
IA-32 Processor Architecture

This chapter is excerpted from *Assembly Language for Intel-Based Computers*, by Kip R. Irvine. Copyright Prentice-Hall Publishing, 2003. All rights reserved. You may print a copy of this chapter, but you may not extract or copy text or illustrations from this document to use in any other publication.

- 2.1 General Concepts
 - 2.1.1 Basic Microcomputer Design
 - 2.1.2 Instruction Execution Cycle
 - 2.1.3 Reading from Memory
 - 2.1.4 How Programs Run
 - 2.1.5 Section Review
- 2.2 IA-32 Processor Architecture
 - 2.2.1 Modes of Operation
 - 2.2.2 Basic Execution Environment
 - 2.2.3 Floating-Point Unit
 - 2.2.4 Intel Microprocessor History
 - 2.2.5 Section Review
- 2.3 IA-32 Memory Management
 - 2.3.1 Real-address Mode
 - 2.3.2 Protected Mode
 - 2.3.3 Section Review
- 2.4 Components of an IA-32 Microcomputer
 - 2.4.1 Motherboard
 - 2.4.2 Video Output
 - 2.4.3 Memory
 - 2.4.4 Input-Output Ports
 - 2.4.5 Section Review
- 2.5 Input-Output System
 - 2.5.1 How It All Works
 - 2.5.2 Section Review
- 2.6 Chapter Summary

2.1 General Concepts

This chapter describes the architecture of the Intel IA-32 processor family and its host computer system from a programmer's point of view. As we pointed out in the first chapter, assembly language is a great tool for learning how a computer works. This chapter is an essential part of the learning process, because you need to learn the basics of system architecture before assembly language can be useful.

We try to strike a balance between concepts that apply to all microcomputer systems and specific information about the IA-32 processor family. It is impossible to know what type of

computer systems you will use in the future, so it would be a mistake for you to only learn about the IA-32. On the other hand, a generalized, superficial knowledge of computer processors might leave you with an empty feeling, not having had enough experience with a single CPU and its assembly language to do anything useful. To use an imperfect analogy, people don't become great cooks by reading cookbooks. They learn to cook a few dishes well, and then build on their experience.

After reading this chapter, you may want to delve further into the IA-32's design. You can begin by reading Intel's well-written and authoritative manual: *IA-32 Intel Architecture Software Developer's Manual, Volume 1: Basic Architecture*. You can download it for free from the Intel Web site (www.intel.com). Far from being a dry, dull reference, this manual can keep your attention for days or weeks. But don't forget about the book you're holding in your hands—it still has a lot to offer you!

2.1.1 Basic Microcomputer Design

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

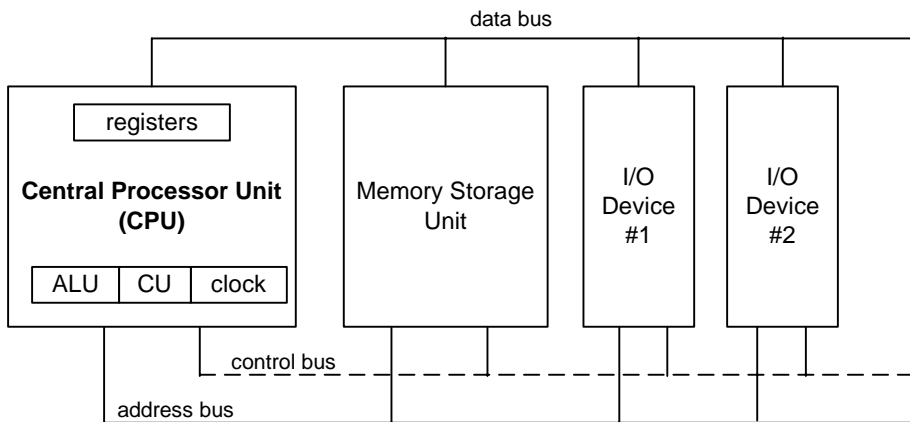
- The clock synchronizes the internal operations of the CPU with other system components.
- The control unit (CU) coordinates the sequencing of steps involved in executing machine instructions.
- The arithmetic logic unit (ALU) performs arithmetic operations such as addition and subtraction, and logical 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 of these pins connect to the data bus, the control bus, and the address bus.

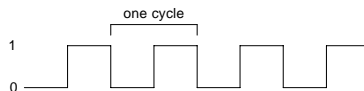
The *memory storage unit* is where instructions and data are held while a computer program is running. The storage unit receives requests for data from the CPU, transfers data from random access memory (RAM) to the CPU, and transfers data from the CPU into memory.

A *bus* is a group of parallel wires that transfer data from one part of the computer to another. The system bus of a computer usually consists of three different buses: the data bus, the control bus, and the address bus. The *data bus* transfers instructions and data between the CPU and memory. The *control bus* uses binary signals to synchronize the 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.

Figure 2–1 Block Diagram of a Microcomputer.



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



(F197)

The duration of a clock cycle is the reciprocal of the clock's speed, 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 between the speed of the CPU, the system bus, and memory circuits. (Recent research suggests that in the near future we may abandon the synchronized computing model in favor of a type of asynchronous operation that would not require a system clock.)

2.1.2 Instruction Execution Cycle

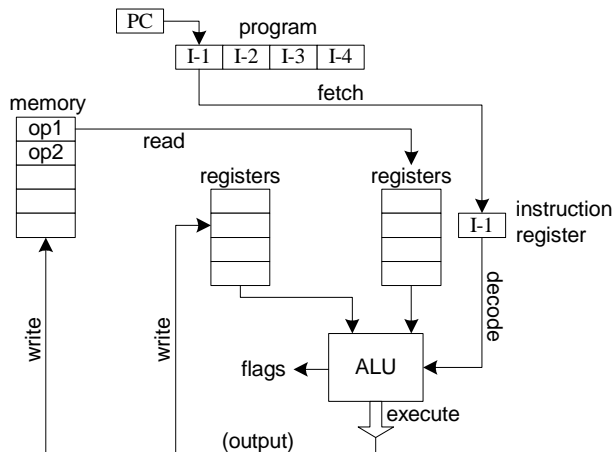
The execution of a single machine instruction can be divided into a sequence of individual operations called the *instruction execution cycle*. When the CPU executes an instruction using a memory operand, it must calculate the address of the operand, place the address on the address

bus, wait for memory to get the operand, and so on.

Before it executes, a program must be loaded into memory. In Figure 2–2, the *program counter* is a register that contains the address of the next instruction about to be executed. The *instruction queue* is a holding area inside the microprocessor into which one or more instructions are copied just before they execute. When the CPU executes a single machine instruction, three primary operations are always necessary: *fetch*, *decode*, and *execute*. Two more steps are required when the instruction uses a memory operand: *fetch operand*, and *store output operand*. In other words, as many as five operations may be required by instructions that access memory.

- **Fetch:** The control unit fetches the instruction, copying it from memory into the CPU and increments the program counter (PC).
- **Decode:** The control unit determines the type of instruction to be executed. It passes zero or more operands to the arithmetic logic unit (ALU) and sends signals to the ALU that indicate the type of operation to be performed.
- **Fetch operands:** If a memory operand is used, the control unit initiates a read operation to retrieve the input operand from memory.
- **Execute:** The arithmetic logic unit executes the instruction, sends its data to the output operand, and updates status flags providing information about the output.
- **Store output operand:** If the output operand is in memory, the control unit initiates a write operation to store the data.

Figure 2–2 Instruction Execution Cycle.



2.1.2.1 Multi-Stage Pipeline

Each step in the instruction cycle takes at least one tick of the system clock, called a *clock cycle*. But this doesn't mean that the processor must wait until all steps are completed before beginning to process the next instruction. The processor can execute the steps in parallel, a technique

known as *pipelining*. The IA-32 processor (beginning with the Intel386) is a pipelined processor with a six-stage execution cycle. The six stages and the parts of the processor that carry them out are listed here:

1. *Bus Interface Unit* (BIU): accesses memory and provides input-output.
2. *Code Prefetch Unit*: receives machine instructions from the BIU and inserts them into a holding area named the *instruction queue*.
3. *Instruction Decode Unit*: decodes machine instructions from the prefetch queue and translates them into microcode.
4. *Execution Unit*: executes the microcode instructions produced by the instruction decode unit.
5. *Segment Unit*: translates logical addresses to linear addresses and performs protection checks.
6. *Paging Unit*: translates linear addresses into physical addresses, performs page protection checks, and keeps a list of recently accessed pages.

Example Let's assume that each execution stage in the processor requires a single clock cycle. Figure 2–3 uses a grid to represent a six-stage *non-pipelined* processor, the type used by Intel prior to the Intel486. When instruction I-1 has finished stage S6, instruction I-2 begins. Twelve clock cycles are required to execute the two instructions. In other words, for k execution stages, n instructions require $(n * k)$ cycles to process.

Figure 2–3 Six-Stage Non-Pipelined Instruction Execution.

		Stages					
		S1	S2	S3	S4	S5	S6
Cycles	1	I-1					
	2		I-1				
	3			I-1			
	4				I-1		
	5					I-1	
	6						I-1
	7	I-2					
	8		I-2				
	9			I-2			
	10				I-2		
	11					I-2	
	12						I-2

Of course, Figure 2–3 represents a major waste of CPU resources because each stage is used only one-sixth of the time.

If, on the other hand, a processor supports pipelining, as in Figure 2–4, a new instruction can enter stage S1 during the second clock cycle. Meanwhile, the first instruction has entered

stage S2. This enables the overlapped execution of the two instructions. Two instructions, I-1 and I-2, are shown progressing through the pipeline. I-2 enters stage S1 as soon as I-1 has moved to stage S2. As a result, only seven clock cycles are required to execute the two instructions. When the pipeline is full, all six stages are in use all the time.

Figure 2–4 Six-Stage Pipelined Execution.

		Stages					
		S1	S2	S3	S4	S5	S6
Cycles	1	I-1					
	2	I-2	I-1				
	3		I-2	I-1			
	4			I-2	I-1		
	5				I-2	I-1	
	6					I-2	I-1
	7						I-2

In general, for k execution stages, n instructions require $k + (n - 1)$ cycles to process. Whereas the non-pipelined processor we showed earlier required 12 cycles to process 2 instructions, the pipelined processor can process 7 instructions in the same amount of time.

2.1.2.2 Superscalar Architecture

A *superscalar* processor has two or more execution pipelines, making it possible for two instructions to be in the execution stage at the same time. In order to better understand why a superscalar processor would be useful, let's consider the preceding pipelined example, in which we assumed that the execution stage (S4) required a single clock cycle. That was an overly simplistic approach. What would happen if stage S4 required two clock cycles? Then a bottleneck would occur, shown in Figure 2–5. Instruction I-2 cannot enter stage S4 until I-2 has completed the stage, so I-2 has to wait one more cycle before entering stage S4. As more instructions enter the pipeline, wasted cycles occur (shaded in gray). In general, for k stages (where one stage requires 2 cycles), n instructions require $(k + 2n - 1)$ cycles to process.

Figure 2–5 Pipelined Execution Using a Single Pipeline.

		Stages					
		S1	S2	S3	S4 ^{exe}	S5	S6
Cycles	1	I-1					
	2	I-2	I-1				
	3	I-3	I-2	I-1			
	4		I-3	I-2	I-1		
	5			I-3	I-1		
	6				I-2	I-1	
	7				I-2	I-1	
	8				I-3	I-2	
	9				I-3	I-2	
	10					I-3	
	11						I-3

When a superscalar processor design is used, multiple instructions can be in the execution stage at the same time. For n pipelines, n instructions can execute during the same clock cycle. The Intel Pentium, which had two pipelines, was the first superscalar processor in the IA-32 family. The Pentium Pro processor was the first to use three pipelines.

Let's introduce a second pipeline into our 6-staged pipeline and assume that stage S4 requires two cycles. In Figure 2–6, odd-numbered instructions enter the *u-pipeline* and even-numbered instructions enter the *v-pipeline*. This removes the wasted cycles, and it is now possible to process n instructions in $(k + n)$ cycles.

Figure 2–6 Superscalar 6-Stage Pipelined Processor.

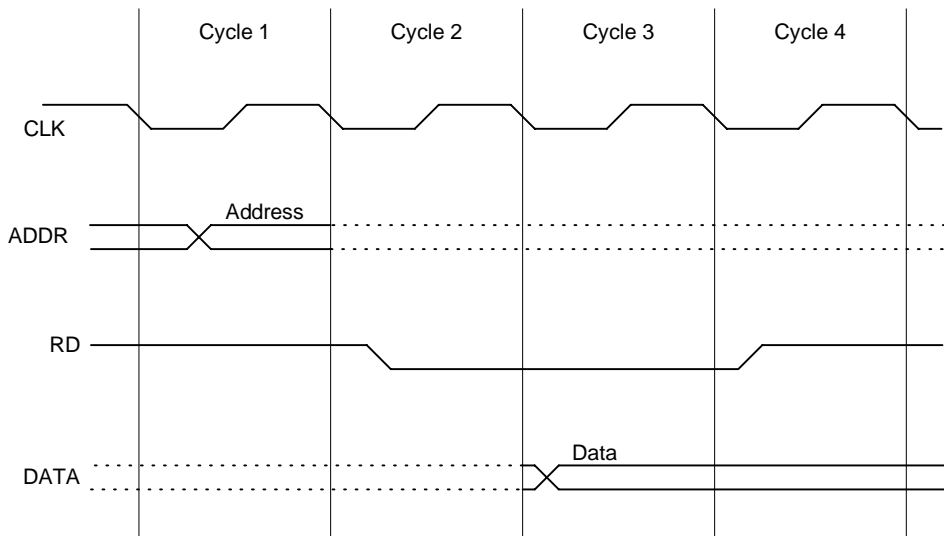
		Stages						
		S1	S2	S3	S4		S5	S6
					u	v		
Cycles	1	I-1						
	2	I-2	I-1					
	3	I-3	I-2	I-1				
	4	I-4	I-3	I-2	I-1			
	5		I-4	I-3	I-1	I-2		
	6			I-4	I-3	I-2	I-1	
	7				I-3	I-4	I-2	I-1
	8					I-4	I-3	I-2
	9						I-4	I-3
	10							I-4

2.1.3 Reading from Memory

Memory access is an important factor when understanding the speed of a program. The CPU clock might be capable of running at 1 or 2 GHz, for example, but access to memory over the system bus is far slower. This forces the CPU to wait one or more clock cycles until operands have been fetched from memory before instructions can execute. These 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 2–7 shows the processor clock (CLK) rising and falling at regular time intervals. In this example, a clock cycle begins as the clock signal changes from high to low. These changes are called *trailing edges*, and they indicate the time taken by the transition between states.

Figure 2-7 Memory Read Cycle.



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

Cycle 1: The address bits of the memory operand are placed on the address bus (ADDR).

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 (RD) goes to 1, signaling that the CPU can now read the values on the data bus.

Cache Memory Because conventional memory is so much slower than the CPU, microcomputers have high-speed *cache memory* that holds the most recently used instructions and data. Whenever possible, the CPU reads from cache memory, giving programs a noticeable boost in performance. There are two types of cache memory in an IA-32 system: *Level-1 cache* is inside the processor itself, and *Level-2 cache* is located on separate high-speed memory chips next to the CPU. Level-1 cache is faster and more expensive than Level-2 cache.

2.1.4 How Programs Run

2.1.4.1 Load and Execute Process

When you tell the computer's operating system (OS) to load and run a program, the following things happen (in sequence):

- The user issues a command to run a certain program. This might be done by typing the program's filename at a command prompt (as in MS-DOS or Linux), or by clicking on an icon or shortcut that identifies the program (as in MS-Windows or Mac OS).
- The 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's filename 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. (This process might involve several steps, but they are transparent to the user.)
- The OS determines the next available location in memory, and loads the program file into memory. It allocates a certain 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 the correct addresses of program data.
- The OS executes a branching instruction that causes the CPU to begin execution of the program's first machine instruction. As soon as the program begins running, it is called a *process*. The OS gives the process an identification number (*process ID*) that makes it possible to keep track of the process while it is running.
- The process runs by itself. It is the OS's job to track the execution of the process and to respond to its requests for system resources. Examples of resources are memory, disk files, and input-output devices.
- When the process ends, its handle is removed and the memory it used is released so it can be used by other programs.

If you're using Windows NT or 2000, 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 with names you might not recognize. Each of those processes is a small program running independently of all the others. Note that each has a PID (program ID), and you can continuously track both the amount of CPU time and the amount of memory used by the program. Most of these run in the background without any visible element. If you know what you're doing, you can shut down a process that was somehow left running in memory by mistake. (On my computer, for example, outlook.exe sometimes.) Of course, if you shut down the wrong process, your computer may stop running, and you'll have to reboot.

2.1.4.2 Multitasking

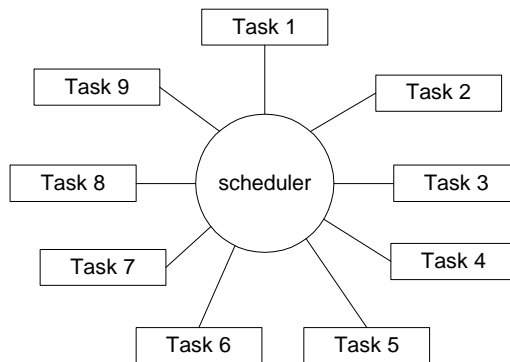
When an operating system is able to run multiple tasks at the same time, it is said to be *multitasking* (or *preemptive multitasking*). A moment ago we were talking about processes, and now we're talking about tasks. A process may optionally contain multiple tasks (or *threads of execution*) that are more or less independent of each other. A game program, for example, with several animated graphics moving independently of each other, might assign each graphic to a separate

task. Some processes consist of only a single task.

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

By rapidly switching tasks, the OS gives the illusion that loaded tasks are running simultaneously. One type of scheduling used by the OS is called *round-robin scheduling*. In Figure 2–8, nine tasks are active. Suppose the *scheduler* arbitrarily assigned 11 milliseconds to each task, and activated them in sequence. One full circuit of the tasks would require a little over 100 milliseconds, which includes time to switch from task to task.

Figure 2–8 Round-Robin Scheduler.



A multitasking OS must run on a processor that supports *task switching*, which means that 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, the task's variables, and its program counter. A multitasking OS will usually assign varying priorities to tasks, giving them relatively larger or smaller time slices.

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. Define *pipelined execution*.
8. In a 5-stage non-pipelined processor, how many clock cycles would it take to execute 2 instructions?
9. In a 5-stage single-pipelined processor, how many clock cycles would it take to execute 8 instructions?
10. What is a *superscalar processor*?
11. Suppose that a 5-stage dual-pipelined processor has one stage that requires two clock cycles to execute, and there are two pipelines for that stage. How many clock cycles would be required to execute 10 instructions?
12. When a program runs, what information does the OS read from the filename's disk directory entry?
13. After a program has been loaded into memory, how does it begin execution?
14. Define *multitasking*.
15. What is the function of the OS scheduler?
16. When the processor switches from one task to another, what values in the first task's state must be preserved?

2.2 IA-32 Processor Architecture

In this section we detail many aspects of the IA-32 processor architecture. Although we mentioned this in Chapter 1, it is worth repeating that IA-32 refers to a family of processors beginning with the Intel386 and continuing up to the latest 32-bit processor, the Pentium 4. Although many enhancements have been made to the processor's performance and implementation, these differences are hidden behind the IA-32 standard. From the programmer's point of view, the IA-32 architecture has not changed substantially since the Intel386. The primary exception is the introduction of a set of high-performance instructions that improve multimedia processing.

2.2.1 Modes of Operation

IA-32 processors have three basic modes of operation: Protected mode, Real-address mode, and System Management mode. In addition, the Virtual-8086 mode is a special case of Protected mode.

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 (called *segments*), and the processor detects any attempt by a program to reference memory outside its assigned segment.

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, even if an MS-DOS program crashes, it will not affect other programs running at the same time. (This feature is often called *Virtual-8086 mode* even though it is not really a separate processor mode.)

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 the other two modes. This mode is available in Windows 98, for example, if you need to run an MS-DOS program that seizes control of the computer's hardware. Old computer games often do this. All Intel processors boot in Real-address mode. After that, the host operating system may switch to another mode.

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

2.2.2 Basic Execution Environment

2.2.2.1 Address Space

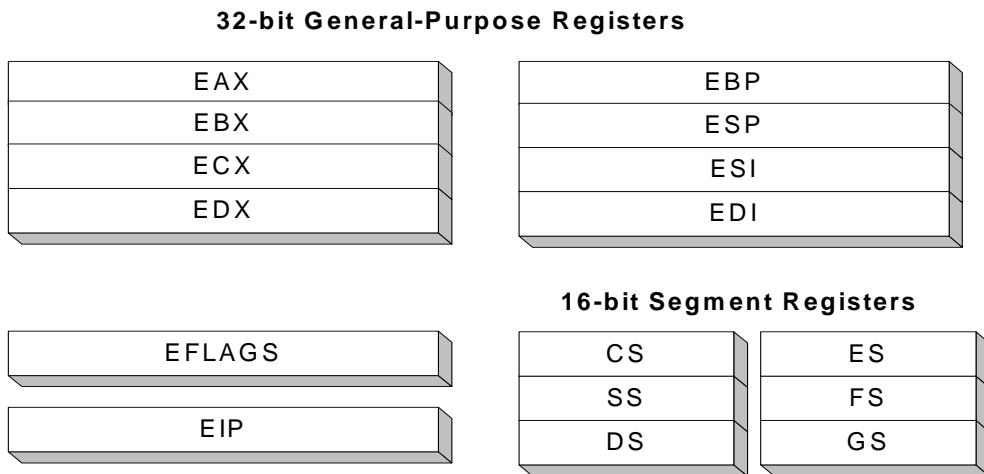
In Protected mode, IA-32 processors can access up to 4GB of memory. This is because 32-bit registers can have values between 0 and $2^{32} - 1$. In Real-address mode, a maximum of 1MB of memory can be accessed. If the processor is in Protected mode and running multiple programs in virtual-8086 mode, each program can access its own separate 1MB area of memory.

2.2.2.2 Basic Program Execution Registers

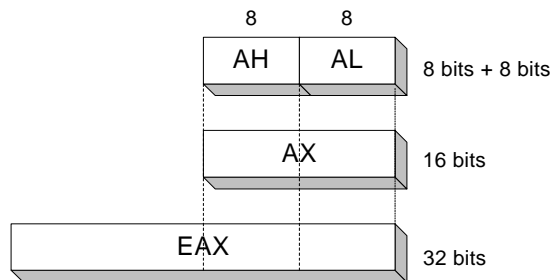
Registers are high-speed storage locations directly inside the CPU, designed to be accessed at much higher speed than conventional memory. When a processing loop is optimized for speed, for example, registers are used inside the loop rather than variables.

Figure 2-9 shows the *basic program execution registers* (as Intel calls them). There are eight general-purpose registers, eight segment registers, a register that holds processor status flags (EFLAGS), and an instruction pointer (EIP).

Figure 2–9 IA-32 Basic Program Execution Registers.



General-Purpose Registers The *general-purpose registers* are primarily used for arithmetic and data movement. As shown in the following figure, each register can be addressed as either a single 32-bit value or two 16-bit values:



Some registers can also be addressed as four separate 8-bit values. For example, the EAX register is 32 bits. Its lower 16 bits are also named AX. The upper 8 bits of AX are named AH, and the lower 8 bits are named AL.

This 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

32-bit	16-bit	8-bit (high)	8-bit (low)
ECX	CX	CH	CL
EDX	DX	DH	DL

The remaining general-purpose registers have separate names for their lower 16 bits, but cannot be divided further. The 16-bit registers shown here are usually used only when writing programs that run in Real-address mode:

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 should never be 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 The *segment registers* are used as base locations for preassigned memory areas called *segments*. Some segments hold program instructions (code), others hold variables (data), and another segment called 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 this address, causing the program to branch to a new location.

EFLAGS Register The EFLAGS (or just *Flags*) register consists of individual binary bits that either control the operation of the CPU or reflect the outcome of some CPU operation. There are

machine instructions that can test and manipulate the processor flags.

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

Control Flags. Individual bits can be set in the EFLAGS register by the programmer to control the CPU's operation. Examples are the *Direction* and *Interrupt* flags. We will cover these on an as-needed basis later in the book.

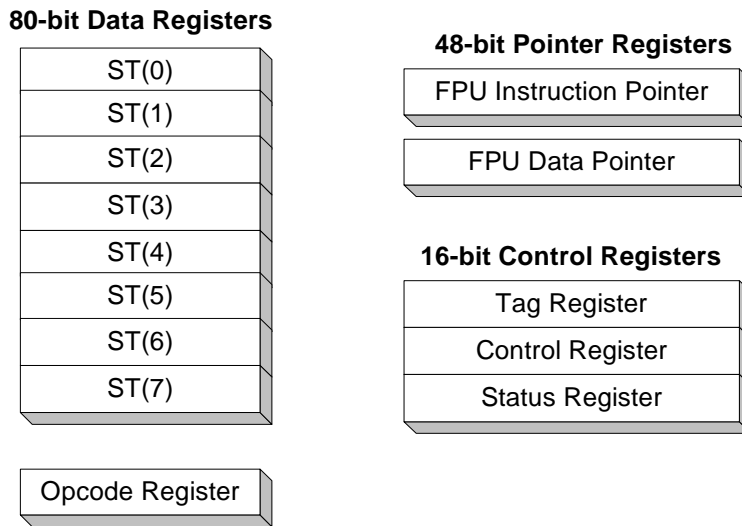
Status Flags. The Status flags reflect the outcomes of arithmetic and logical operations performed by the CPU. They are the Overflow, Sign, Zero, Auxiliary Carry, Parity, and Carry flags. Their abbreviations are shown immediately after their names:

- 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 wide (too many bits) 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 is set when an arithmetic operation causes a carry from bit 3 to bit 4 in an 8-bit operand.
- The **Parity** flag sums the number of bits that are set in a number, and indicates whether the sum is odd or even.

2.2.3 Floating-Point Unit

The IA-32 has a *floating-point unit* (FPU) that is used expressly for high-speed floating-point arithmetic. At one time a separate coprocessor chip was required for this, but beginning with the Intel486, it was 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 of the FPU are shown in Figure 2–10.

Figure 2–10 Floating-Point Unit Registers.

2.2.3.1 Other Registers

In passing, we will mention two other sets of registers used for advanced multimedia programming:

- Eight 64-bit registers for use with the MMX instruction set
- Eight 128-bit XMM registers used for single-instruction, multiple-data (SIMD) operations

2.2.4 Intel Microprocessor History

In this chapter, you're about to get another dose of history from someone (me) who really was around when the first IBM-PC was released, in the dark days when they had 64K of RAM and no hard drives.

Senior programmers love to talk about history and legends because a lot of them were actually around when the history was being written. One of my professors worked on the Mark I computer at Harvard University during World War II. He was given a single register from the Mark I when it was dismantled. (The register was about 2 feet high and weighed 20 pounds!)

Intel 8086 The Intel 8086 processor (created in 1978) marks 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 that allowed programs to address up to 1MB of RAM. This greater access to memory made it possible to write complex business applications. The IBM-PC, introduced around 1980, contained an Intel

8088 processor, which was identical to the 8086 except that it had an 8-bit data bus, making it slightly less expensive to produce. Today, the Intel 8088 is primarily used in low-cost microcontrollers and costs only a few dollars.

Downward Compatibility. It should be noted that each new processor introduced into the Intel family has always been downward-compatible with earlier generations. This has made it possible for the same software to run on the newer computers without modification. Newer software eventually appeared, however, that required the features of the more advanced processors.

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

2.2.4.1 IA-32 Processor Family

The Intel386 processor featured 32-bit registers and a 32-bit address bus and external data path. This was the first member of the IA-32 family, which also includes the Intel486 and various Pentium processors. They support a new way of addressing virtual memory that is larger than the computer's physical memory. Each program was given a 4GB linear address space.

Intel486 Continuing the IA-32 family, the Intel486 processor features an instruction set microarchitecture using *pipelining* techniques that permit multiple instructions to be processed at the same time.

Pentium The Pentium processor added many performance improvements, including a *superscalar* design with two parallel execution pipelines. This permitted two instructions to be decoded and executed simultaneously. The Pentium also introduced a 32-bit address bus and 64-bit internal data path.

2.2.4.2 P6 Processor Family

In 1995, the P6 family of processors was introduced, based on a new micro-architecture design that improved execution speed. At the same time, it extended the basic IA-32 architecture. This family includes, among others, the Pentium Pro, Pentium II, and Pentium III. The Pentium Pro introduced advanced techniques to improve the way instructions were executed. The Pentium II added MMX technology to the P6 family. The Pentium III introduced SIMD (streaming extensions) into the IA-32 architecture, with special 128-bit registers designed to move large amounts of data quickly.

Pentium 4 At the time of this writing, the Pentium 4 is the newest IA-32 processor. It introduced the *NetBurst* micro-architecture that permits the processor to operate at much higher speeds than previous IA-32 processors. It appears to be oriented primarily toward high-performance multimedia applications.

2.2.4.3 CISC and RISC

The earliest Intel processors for the IBM Personal Computer were based on what is called a *Complex Instruction Set* (CISC) approach. The Intel instruction set includes powerful ways to address data, and instructions that are relatively high-level complex operations. The philosophy was that high-level language compilers would have less work to do if individual machine-language instructions were powerful. A major disadvantage to the CISC approach is that complex instructions require a relatively long time for the processor to decode and execute. An interpreter program inside the CPU written in a language called *microcode* decodes and executes each machine instruction. Once Intel committed to a complex instruction set, it was necessary for all subsequent Intel processors to be compatible with the first one. Software written for the original IBM Personal Computer can still run on today's latest Pentium.

A completely different approach to microprocessor design is called *Reduced Instruction Set* (RISC). A RISC machine language consists of a relatively small number of short, simple instructions that can be executed very 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. Unfortunately, these systems have been expensive because the processors were produced in small quantities.

Because of the huge popularity of IBM-PC compatible computers, Intel was able to lower the price of its processors and thus 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 pipelining and superscalar) in its Pentium processors. Meanwhile, the IA-32 instruction set continues to be enormously complex and constantly expanding.

2.2.5 Section Review

1. What are the IA-32 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), which 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 wide to fit into the destination?
8. Which flag is set when the result of an *signed* arithmetic operation is too wide 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 a CISC instruction set.
16. Describe a RISC instruction set.

2.3 IA-32 Memory Management

The IA-32 manages memory according to the basic modes of operation that we discussed earlier in Section 2.2.1.

In *Real-address* mode, only 1MB 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 read and modify any area of RAM (random-access memory), and they can read but not modify any area of ROM (read-only memory). 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 4GB 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 both run in Protected mode.

In *Virtual-8086* mode, the computer runs in Protected mode and creates a virtual 8086 machine with its own 1MB address space that simulates an 80x86 computer running in Real-address mode. Windows NT and 2000, for example, creates 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 and 2000.

In the next two sections (Section 2.3.1 and Section 2.3.2), we will explain details of both Real-address mode and Protected mode. If you want to study this subject in more detail, a good source is the three-volume *IA-32 Intel Architecture Software Developer's Manual*. You can read or download it from Intel's Web site (www.intel.com).

2.3.1 Real-address Mode

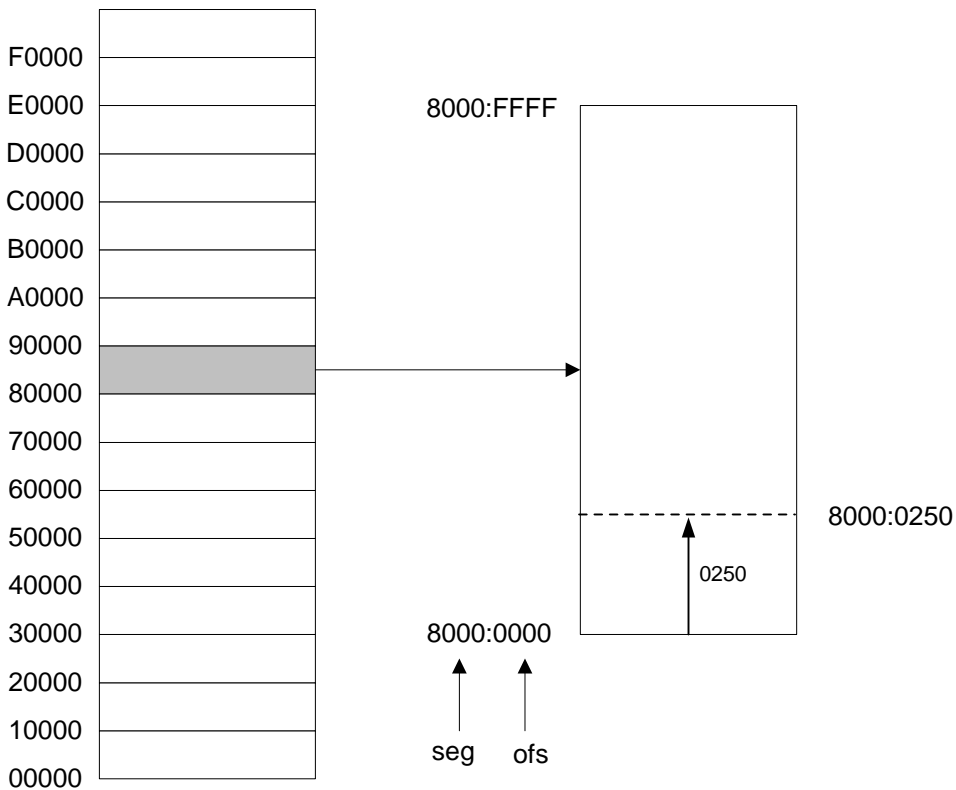
In Real-address mode, the IA-32 processor can access 1,048,576 bytes of memory (1MB) using 20-bit addresses in the range 0 to FFFFF hexadecimal. The basic problem that Intel engineers had to solve was that the original 8086 processor had only 16-bit registers, so it was impossible to directly represent a 20-bit address. They came up with a scheme known as *segmented memory*. All of memory is divided into 64-kilobyte units called *segments*, shown in Figure 2–11.

An analogy might be a large building, where the *segments* represent the floors of the building. A person can ride the elevator to a particular floor, get off, and then begin following the room numbers to locate a single room. The *offset* of a room can be thought of as the distance from the elevator to the room.

Looking again at Figure 2–11, note that each segment begins at an address having a zero for its last hexadecimal digit. Because of this, when segment values are stated, the last zero is dropped. A segment value of C000, for example, refers to the segment that begins at address C0000.

In the same figure, we see an expansion of the segment beginning at 80000. To reach any of the bytes in this segment, we need a 16-bit offset value (0 to FFFF) that can be added to the segment's base location. An example is 8000:0250, which represents an offset of 250 inside the segment beginning at address 80000. The linear address is 80250h.

Figure 2–11 Segmented Memory Map, Real-address Mode



2.3.1.1 20-bit Linear Address Calculation

An *address* is a number that refers to a single location in memory. In Real-address mode, the *linear* (or *absolute*) address is 20 bits, ranging from 0 to FFFFF hexadecimal. But programs cannot use linear addresses directly, so they express addresses using two 16-bit numbers, which are together called a *segment-offset* address:

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

When addresses are expressed this way, the CPU automatically does some arithmetic and converts the segment-offset address to a 20-bit linear address.

Example. Suppose that a variable's hexadecimal segment-offset address is 08F1:0100. The CPU multiplies the segment value by 10 hexadecimal and adds this to the variable's offset:

$$08F1 * 10 = 08F10 \quad (\text{adjusted segment value})$$

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 base locations of a program's code, data, and stack segments:

- 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.

2.3.2 Protected Mode

Now let's turn our attention to Protected mode, the more powerful "native" processor mode. When the processor is running in Protected mode, each program can address up to 4GB of memory, from 0 to FFFFFFFF hexadecimal. This use of a flat address space is also called the *flat memory model* by the Microsoft Assembler. From the programmer's point of view, the flat memory model is very simple to use because it only requires a single 32-bit integer to hold the address of any instruction or variable. The operating system does quite a bit of background work to preserve the illusion of simplicity, aided by the processor's built-in capabilities. The segment registers (CS, DS, SS, ES, FS, GS) point to *segment descriptor tables* that the operating system uses to define the locations of individual program segments.

A typical Protected-mode program has three segments: code, data, and stack. Three segment registers are used all the time:

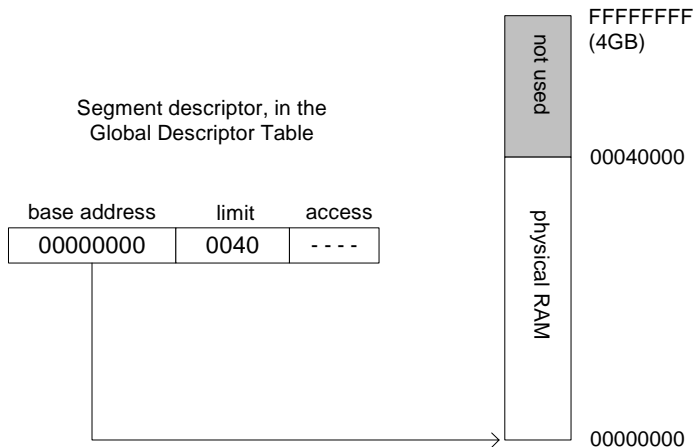
- CS references the descriptor table for the code segment.
- DS references the descriptor table for the data segment.
- SS references the descriptor table for the stack segment.

2.3.2.1 Flat Segmentation Model

In the flat segmentation model, all segments are mapped to the entire 32-bit physical address space of the computer. You have to create at least two segments, one for program code and one for data. Each segment is defined by a *segment descriptor*, a 64-bit value stored in a table known as the *global descriptor table* (GDT). Figure 2–12 shows a segment descriptor whose *base address* field points to the first available location in memory (00000000). The *segment limit* field can optionally indicate the amount of physical memory in the system. In the current figure, the segment limit is 0040. The *access* field contains bits that determine how the segment can be used.

Suppose a computer had 265MB of RAM. The segment limit field would contain 0040 hexadecimal, because its value is implicitly multiplied by 1000 hexadecimal, producing 40000 hexadecimal (256 MB).

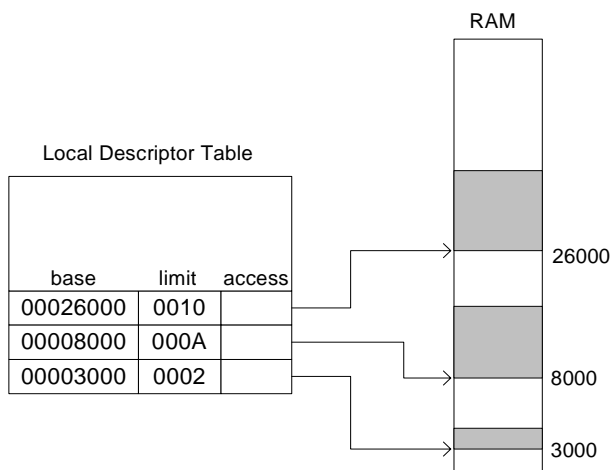
Figure 2–12 Flat Segmentation Model.



2.3.2.2 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 is a separate address space. Figure 2–13 shows that 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 2–13 Multi-Segment Model.



2.3.2.3 Paging

The IA-32 supports 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. Sometimes the complete collection of pages is called *virtual memory*. An operating system will usually include a program called a *virtual memory manager*.

Paging is an important solution to a vexing problem for 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 between them at will. Disk storage, on the other hand, is cheap and plentiful. Paging provides the illusion that memory is almost unlimited in size. Of course, disk access is much slower than main memory access.

When a task is running, parts of it can be stored on disk if they are not currently in use. We say that part of the task has been *paged* (swapped) to disk. Other parts of the task, which are actively executing code, can be in memory. When the CPU needs to execute the part of the task that is currently on disk, 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 (32MB or 64MB), and run 5 or 10 large applications at the same time. You should notice a delay when switching from one program to another, because the OS has to transfer parts 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. This reduces 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 segment-offset address to a linear address: 0950:0100.
5. In Real-address mode, convert the following segment-offset address to a linear address: 0CD1:02E0.
6. In the flat memory model used by the Microsoft Assembler, how many bits are used to 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 the various 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 IA-32 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 an IA-32 Microcomputer

This chapter introduces you to the architecture of IA-32 computers from several points of view. First, the hardware (physical parts of the computer) can be viewed on the *macro* level, looking at peripherals. Then we can look at the internal details of the Intel processor, called the *central processing unit* (CPU). Finally, we look at the software architecture, which is the way the memory is organized, and how the operating system interacts with the hardware.

2.4.1 Motherboard

The heart of any microcomputer is its motherboard. This is a flat board onto which are placed the computer's CPU, supporting processors, 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. Literally dozens of motherboards are

available on the PC market. Although they vary in expansion capabilities and speed, they have a number of elements in common:

- CPU socket: Within the IA-32 family, the socket can be different sizes, depending on the type of processor.
- External cache memory slot: For high-speed cache memory that is used by the CPU to reduce its access to slower conventional RAM.
- Slots to add main memory: Called SIMMs or DIMMs, the memory chips are on small boards that plug into available memory slots.
- BIOS (*basic input-output system*), which is software that has been loaded into a computer chip. Many BIOS chips can be upgraded as the need arises by copying the software from a file supplied by the computer manufacturer. They use a type of memory called *static RAM*.
- IDE cable connectors: For internal fixed disk and CD-ROM drives.
- Sound synthesizer.
- Parallel, serial, USB, video, keyboard, joystick, and mouse ports.
- Network adapter.
- PCI bus connectors for sound cards, graphics cards, data acquisition boards, and other I/O devices.

The following are some of the more important support processors in a typical IA-32 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 8259 *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, as 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.

2.4.1.1 PCI Bus Architecture

The **PCI** (*Peripheral Component Interconnect*) bus was developed by Intel in 1992 to provide a convenient upgrade path for increasingly fast Pentium processors. It is still the dominant bus in today's Pentium systems. The PCI specification supports both 32-bit and 64-bit motherboards. The PCI motherboard provides a connecting bridge between the CPU's local 64-bit bus and the system's external bus.

2.4.1.2 Motherboard Chipset

Most motherboards contain an integrated set of microprocessors and controllers called a *chipset*. The chipset largely determines the capabilities of the computer. The names you see listed here are by Intel, but many motherboards use compatible chipsets from other manufacturers:

- The Intel 8237 Direct Memory Access (DMA) controller transfers data between external devices and RAM, without requiring any work by the CPU.
- The Intel 8259A Interrupt Controller handles requests from the hardware to interrupt the CPU.
- The 8254 Timer Counter handles the system clock that ticks 18.2 times per second, the memory refresh timer, and the time of day clock.
- Microprocessor local bus to PCI bridge.
- System memory controller and cache controller.
- PCI bus to ISA bus bridge.
- Intel 8042 keyboard and mouse microcontroller.

2.4.2 Video Output

The video adapter controls the display of text and graphics on IBM-compatibles. 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.

CRT video monitors use 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 all over.

A direct digital LCD monitor receives a digital bit stream directly from the video controller and does not require raster scanning.

2.4.3 Memory

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

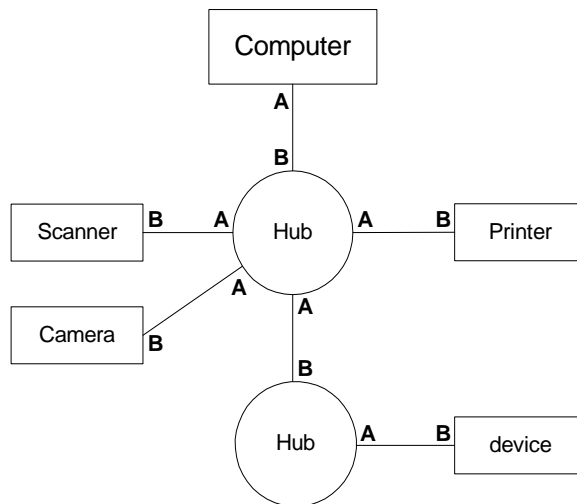
- **ROM** (read-only memory) is memory that is permanently burned into a chip and cannot be erased.
- **EPROM** (erasable programmable read-only memory) can be erased slowly with ultraviolet light, and reprogrammed.

- **Dynamic RAM** is where programs and data are kept when a program is running. It must be refreshed within less than a millisecond or it loses its contents. Because it is inexpensive, it is used for a computer's main memory. Some systems use ECC memory (error checking and correcting), which is memory that is able to detect multiple-bit errors and correct single-bit errors.
- **Static RAM** is the type of RAM chip used primarily for expensive, high-speed cache memory that greatly improves system performance. It keeps its value without having to be constantly refreshed.
- **Video RAM** is used exclusively for storing data that appears on a video display. It is usually located on a video controller board, and is optimized for storing color pixels. Whereas DRAM has only one access port, VRAM is dual-ported, allowing one port to continuously refresh the display while the other port writes data to the display.
- **CMOS RAM** is used on the system motherboard to store system setup information. It is refreshed by a battery, so its contents are retained even when the computer's power is turned off.

2.4.4 Input-Output Ports

USB Port The Universal Serial Bus (USB) port provides an intelligent, high-speed connection between a computer and USB-supported devices. USB ports support data transfer speeds (at present) of up to 12 megabytes per second. You can connect either single-function units (mice, printers) or compound devices that have more than one peripheral sharing the same USB port. A USB hub, shown in Figure 2–14, is a compound device that can be connected to several other devices, including hubs. Each USB cable has two types of connectors (A = upstream), and (B = downstream).

Figure 2–14 USB Hub Configuration.



When a device is attached to the computer via USB, the computer queries the device to get its name, device type, and the type of device driver it supports. This process is called *enumeration*. The computer can suspend power to individual devices and put devices in a suspended state.¹

Parallel Port Most printers connect to a computer via a *parallel port*. By "parallel" we mean that 8 or 16 data bits can travel simultaneously from the computer to the printer. Data can be transferred very quickly 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. The 8255 controller chip is used to program the parallel port.

Serial Port An *RS-232 serial port* sends binary bits one at a time, resulting in slower speeds than the parallel and USB ports, but with the ability to send over larger distances. It can be used to connect a mouse, a modem, or any other serial device to the computer system. The chip that controls the serial ports is the 16550 UART (*Universal Asynchronous Receiver Transmitter*), which is located either on the motherboard or on an adapter card.

1. For more information, read *An Introduction to USB Development*, by Jack G. Ganssle. Embedded Systems Programming (www.embedded.com).

2.4.5 Section Review

1. Describe *external cache memory*.
2. Which Intel processor was behind the creation of the PCI bus?
3. In the motherboard chipset, what task does the Intel 8259A perform?
4. Where is the memory used by the video display located?
5. Describe raster scanning on a CRT video monitor.
6. Name four types of RAM mentioned in this chapter.
7. Which type of RAM is used for Level-2 cache memory?
8. What advantages does a USB device offer over a standard serial or parallel device?
9. What are the names of the two types of USB connectors?
10. Which processor chip controls the serial port?

2.5 Input-Output System

Perhaps you have dreamed of writing computer games. Games are very memory and I/O intensive, and often push their host computers to their absolute limit. Programmers who excel at this know a great deal about video and sound hardware, because they write their code in such a way that it takes advantage of particular hardware design features. You can write hardware-specific code in C++, but usually only after you have learned how to do it in assembly language.

2.5.1 How It All Works

An application program routinely reads input from the keyboard and from files, and it writes output to the screen and to files. This is accomplished not by directly accessing the computer's hardware, but by calling functions in the computer's operating system. Input-output is accomplished via different access levels, similar to the virtual machine concept shown in Chapter 1.

- A high-level programming language such as C++ or Java contains functions that perform input-output. These functions are designed to work on a variety of different computer systems.
- The OS (operating system) is at the next level. It deals with high-level operations, such as writing entire strings and records to files, reading strings from the keyboard, and allocating blocks of memory for application programs.
- The BIOS (Basic Input-Output System) is a collection of functions that communicate directly with hardware devices. The BIOS is installed by the computer's manufacturer, and must be tailored to fit the computer's exact configuration down to the selection of specific

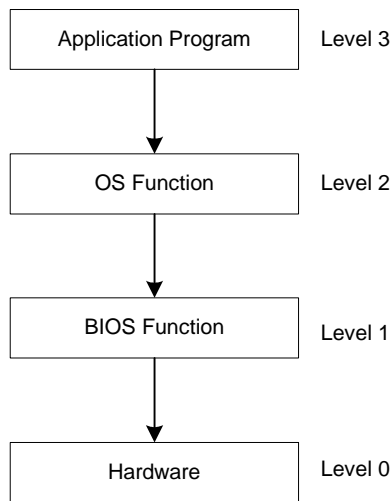
chips on the motherboard. Any operating system installed in the computer must be able to work with the computer's BIOS.

Device Drivers What happens if a new device is installed in a computer that is unknown to the BIOS? When the operating system boots, it loads a device driver (program) that contains functions designed to communicate with the device. The device driver works much like the BIOS, providing input-output functions for the device. An example of such a driver is `CDROM.SYS`, which enables MS-DOS to read CD-ROM drives. It is loaded using a command such as:

```
DEVICE=CDROM.SYS
```

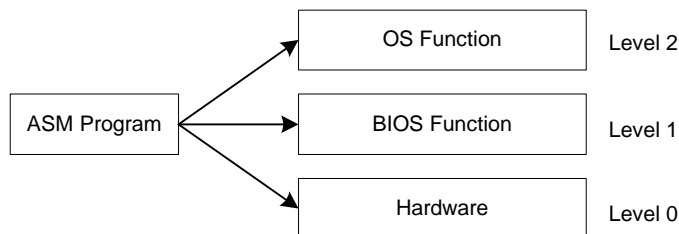
Figure 2–15 shows what happens when an application program displays a string of characters on the screen in a particular color. The following steps are involved:

1. A statement in the application program calls a 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) repeatedly calls a function in the BIOS, passing it the ASCII code and color of each character. The OS calls another BIOS function to advance the cursor to the next position on the screen.
4. The BIOS function (Level 1) receives each 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 2–15 Access Levels for Input-Output Operations.

Programming at Multiple Levels Assembly language programmers have a great deal of power and flexibility when it comes to input-output programming. As shown in Figure 2–16, an assembly language program can choose from the following access levels:

- 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 points, having absolute control over specific devices.

Figure 2–16 Assembly Language Access Levels.

What are the tradeoffs? Control versus portability is the primary one. Level 2 (OS) works on any computer that runs the same OS. If the particular I/O device lacks certain capabilities, the OS

will do its best to compromise. 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 that have a standard BIOS, but will not necessarily produce the same results on all systems. For example, two computers might have video displays with different resolution capabilities. The 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 either with generic devices such as serial ports, or 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 over all operation of the computer. Programs at this level execute as quickly as the hardware will permit.

Suppose, for example, that 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 probably would not be concerned overly much about the card's features. 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. And finally, at the hardware level, you would fine-tune the program for certain brands of audio cards, to take advantage of each card's special features.

Finally, we must point out that 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 Windows NT, 2000, and XP, where 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 three levels of input/output in a computer system, which is the most universal and portable?
2. What distinguishes the BIOS level of 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 is where all the calculations and logic take place. 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 between 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.

Pipelined execution greatly improves the throughput of multiple instructions in a CPU by permitting the overlapped execution of multi-stage instructions. A *superscalar* processor is a pipelined processor with multiple execution pipelines. This is particularly useful when one of the execution stages requires more than a single clock cycle.

A *multitasking* operating system is able to run multiple tasks at the same time. It must run on a processor that supports task switching, which means that the processor saves the state of each task before switching to a new one.

IA-32 processors have three basic modes of operation: Protected mode, Real-address mode, and System Management mode. In addition, the Virtual-8086 mode is a special case of Protected mode.

Registers are named locations within the CPU that can be accessed very quickly. The 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 register consists of individual binary bits that either control the operation of the CPU or reflect the outcome of ALU operations.

The IA-32 has a floating-point unit (FPU) that is expressly for the execution of high-speed floating-point arithmetic.

The Intel 8086 processor marks the beginning of the modern Intel Architecture family. The Intel386 processor, the first of the IA-32 family, featured 32-bit registers and a 32-bit address bus and external data path. The P6 processor family (Pentium Pro onward) is based on a new micro-architecture design that improves execution speed.

The earliest Intel processors for the IBM Personal Computer were based on the *complex instruction set* (CISC) approach. The Intel 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 the IA-32's Real-address mode, only 1MB of memory can be addressed, from hexadecimal 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 4GB of memory. In Virtual-8086 mode, the computer runs in Protected mode and creates a virtual 8086 machine with its own 1MB address space that simulates an 80x86 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). The IA-32 supports 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. The operating system is at the highest level. 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.