CHAPTER

4

Computer Architecture

TABLE OF CONTENTS

An architect takes components such as walls, doors, and ceilings and arranges them together to form a building. Similarly, the computer architect takes components such as input devices, memories, and CPU registers and arranges them together to form a computer.

Buildings come in all shapes and sizes, and so do computers. This fact raises a problem. If we select one computer to study out of the dozens of popular models that are available, then our knowledge will be somewhat obsolete when that model is inevitably discontinued by its manufacturer. Also, this text would be less valuable to people who use the computers we choose not to study.

A virtual computer

But there is another possibility. In the same way that a text on architecture could examine a hypothetical building, this text can explore a virtual computer that contains important features similar to those found on all real computers. This approach has its advantages and disadvantages.

Advantages and disadvantages of a virtual computer

One advantage is that the virtual computer can be designed to illustrate only the fundamental concepts that apply to most computer systems. We can then concentrate on the important points and not have to deal with the individual quirks that are present on all real machines. Concentrating on the fundamentals is also a hedge against obsolete knowledge. The fundamentals will continue to apply even as individual computers come and go in the marketplace.

The primary disadvantage of studying a virtual computer is that some of its details will be irrelevant to those who need to work with a specific real machine at the assembly language level or at the instruction set architecture level. If you understand the fundamental concepts, however, then you will easily be able to learn the details of any specific machine.

There is no 100% satisfactory solution to this dilemma. We have chosen the virtual computer approach mainly for its advantages in illustrating fundamental concepts. Our hypothetical machine is called the *Pep/9 computer*.

The Pep/9 computer

4.1 Hardware

The Pep/9 hardware consists of three major components at the instruction set architecture level, ISA3:

- The central processing unit (CPU)
- The main memory with input/output devices
- The disk

The block diagram of FIGURE 4.1 shows each of these components as a block. The bus is a group of wires that connects the three major components. It carries the data signals and control signals sent between the blocks.

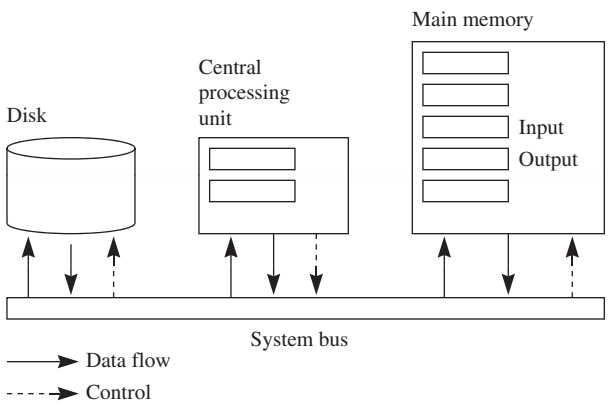


FIGURE 4.1
Block diagram of the Pep/9 computer.

Central Processing Unit (CPU)

The CPU contains six specialized memory locations called registers. As shown in FIGURE 4.2, they are

- The 4-bit status register (NZVC)
- The 16-bit accumulator (A)
- The 16-bit index register (X)
- The 16-bit program counter (PC)
- The 16-bit stack pointer (SP)
- The 24-bit instruction register (IR)

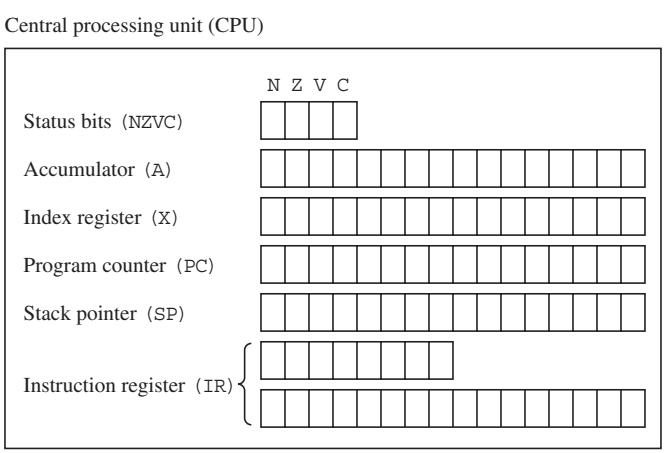


FIGURE 4.2
The CPU of the Pep/9 computer.

The N, Z, V, and C bits in the status register are the negative, zero, overflow, and carry bits (as discussed in Sections 3.1 and 3.2). The accumulator is the register that contains the result of an operation. The next three registers—X, PC, and SP—help the CPU access information in main memory. The index register is for accessing elements of an array. The program counter is for accessing instructions. The stack pointer is for accessing elements on the run-time stack. The instruction register holds an instruction after it has been accessed from memory.

In addition to these six registers, the CPU contains all the electronics (not shown in Figure 4.2) to execute the Pep/9 instructions.

Main Memory

FIGURE 4.3 shows the main memory of the Pep/9 computer. It contains 65,536 eight-bit storage locations. A group of eight bits is called a *byte* (pronounced *bite*). Each byte has an address similar to the number address on a mailbox. In decimal form, the addresses range from 0 to 65,535; in hexadecimal, they range from 0000 to FFFF. Main memory is sometimes called *core memory*.

Figure 4.3 shows the first three bytes of main memory on the first line, the next byte on the second line, the next three bytes on the next line, and, finally, the last two bytes on the last line. Whether you should visualize a line of memory as containing one, two, or three bytes depends on the context of the problem. Sometimes it is more convenient to visualize one byte on a line, sometimes two or three. Of course, in the physical computer a byte is a sequence of eight signals stored in an electrical circuit. The bytes would not be physically lined up as shown in the figure.

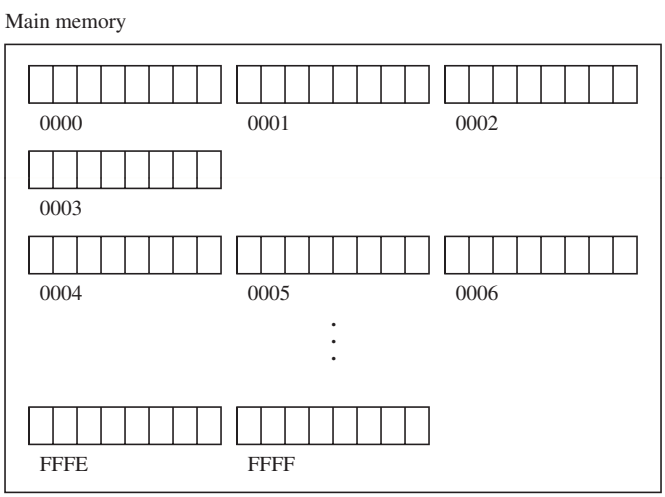
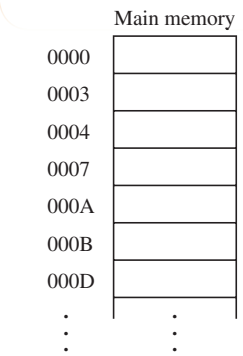


FIGURE 4.3
The main memory of the Pep/9 computer.

Frequently it is convenient to draw main memory as in FIGURE 4.4, with the addresses along the left side of the block. Even though the lines have equal widths visually in the block, a single line may represent one or several bytes. The address on the side of the block is the address of the leftmost byte in the line.

FIGURE 4.4
Another style for depicting main memory.



You can tell how many bytes the line contains by the sequence of addresses. In Figure 4.4, the first line must have three bytes because the address of the second line is 0003. The second line must have one byte because the address of the third line is 0004, which is one more than 0003. Similarly, the third and fourth lines each have three bytes, the fifth has one, and the sixth has two. From the figure, it is impossible to tell how many bytes the seventh line has. The first three lines of Figure 4.4 correspond to the first seven bytes in Figure 4.3.

Regardless of the way the bytes of main memory are laid out on paper, the bytes with small addresses are referred to as the *top* of memory, and those with large addresses are referred to as the *bottom*.

A Pep/9 word

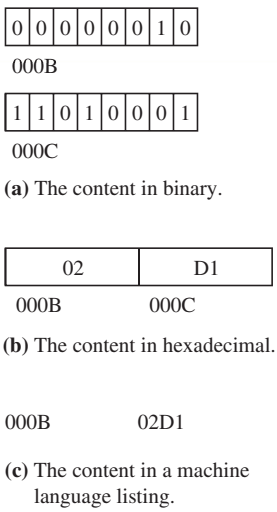
Most computer manufacturers specify a word to be a certain number of bytes. In the Pep/9 computer, a word is two adjacent bytes. A word, therefore, contains 16 bits. Most of the registers in the Pep/9 CPU are word registers. In main memory, the address of a word is the address of the first byte of the word. For example, FIGURE 4.5(a) shows two adjacent bytes at addresses 000B and 000C. The address of the 16-bit word is 000B.

It is important to distinguish between the content of a memory location and its address. Memory addresses in the Pep/9 computer are 16 bits long. Hence, the memory address of the word in Figure 4.5(a) could be written in binary as 0000 0000 0000 1011. The content of the word at this address, however, is 0000 0010 1101 0001. Do not confuse the content of the word with its address. They are different.

To save space on the page, the content of a byte or word is usually written in hexadecimal. Figure 4.5(b) shows the content in hexadecimal of the same word at address 000B. In a machine language listing, the address of the first byte of a group is printed, followed by the content in hexadecimal, as in Figure 4.5(c). In this format, it is especially easy to confuse the address of a byte with its content.

In the example in Figure 4.5, you can interpret the content of the memory location several ways. If you consider the bit sequence 0000 0010 1101 0001 as an integer in two's complement representation, then the first bit is the sign bit, and the binary sequence represents decimal 721. If you consider the rightmost seven bits as an ASCII character, then the binary sequence represents the character Q. The main memory cannot determine which way the byte will be interpreted. It simply remembers the binary sequence 0000 0010 1101 0001.

FIGURE 4.5
The distinction between the content of a memory location and its address.



(a) The content in binary.

(b) The content in hexadecimal.

(c) The content in a machine language listing.

Input/Output Devices

The Pep/9 virtual machine

You may be wondering where this Pep/9 hardware is located and whether you will ever be able to get your hands on it. The answer is, the hardware does not exist! At least it does not exist as a physical machine. Instead, it exists as a set of programs that you can execute on your computer system. The programs simulate the behavior of the Pep/9 virtual machine described in these chapters.

The Pep/9 system simulates two input/output (I/O) modes—interactive and batch. Before executing a program, you must specify the I/O mode. If you specify interactive, the input comes from the keyboard, and both input and output appear in a terminal window. If you specify batch, the input comes from an input pane and the output goes to an output pane. Batch mode simulates input from a file because the input pane must have data in it before the program executes, just as an input file contains data that a program processes.

Memory-mapped I/O

Pep/9 simulates a common computer systems design called *memory-mapped I/O*. The input device is wired into main memory at one fixed address, and the output device is wired into main memory at another fixed address. In Pep/9, the input device is at address FC15 and the output device is at address FC16.

Data and Control

The solid lines connecting the blocks of Figure 4.1 are data flow lines. Data can flow from the input device at address FC15 on the bus to the CPU. It can also flow from the CPU on the bus to the output device at address FC16. Data cannot flow directly from the input device to another memory location without going first to the CPU. Nor can it flow directly from some other memory location to the output device without going first to the CPU. Most computer systems have a mechanism, called *direct memory access* (DMA), that allows data to flow between the disk and main memory directly without going through the CPU. Although this design is common, the Pep/9 simulator does not provide this feature.

The dashed lines are control lines. Control signals all originate from the CPU, which means that the CPU controls all the other parts of the computer. For example, to make data flow from the input device in main memory to the CPU along the solid data flow lines, the CPU must transmit a send signal along the dashed control line to the memory. The important point is that the processor really is central. It controls all the other parts of the computer.

Instruction Format

Each computer has its own set of instructions wired into its CPU. The instruction set varies from manufacturer to manufacturer. It often varies among computers made by the same company, although many manufacturers produce a family of models, each of which contains the same instruction set as the other models in that family.

The instruction specifier and operand specifier

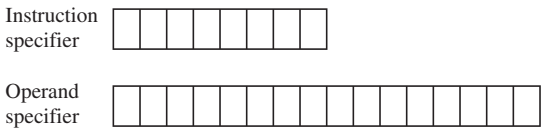
The Pep/9 computer has 40 instructions in its instruction set, shown in FIGURE 4.6. Each instruction consists of either a single byte called the *instruction specifier,* or the instruction specifier followed immediately by a word called the *operand specifier.* Instructions that do not have an operand specifier are called *unary instructions*. FIGURE 4.7 shows the structure of nonunary and unary instructions.

The opcode

The eight-bit instruction specifier can have several parts. The first part is called the *operation code*, often referred to as the *opcode.* The opcode may consist of as many as eight bits and as few as four. For example, Figure 4.6 shows the instruction to move the stack pointer to the accumulator as having an eight-bit opcode of 0000 0011. The add to SP instruction, however, has the five-bit opcode 01010. Instructions with fewer than eight bits in the opcode subdivide their instruction specifier into several fields depending on the instruction. Figure 4.6 indicates these fields with the letters a, r, and n. Each one of these letters can be either 0 or 1.

FIGURE 4.6
The Pep/9 instruction set at Level ISA3.

Instruction Specifier	Instruction
0000 0000	Stop execution
0000 0001	Return from `CALL`
0000 0010	Return from trap
0000 0011	Move SP to A
0000 0100	Move NZVC flags to A⟨12..15⟩
0000 0101	Move A⟨12..15⟩ to NZVC flags
0000 011r	Bitwise invert r
0000 100r	Negate r
0000 101r	Arithmetic shift left r
0000 110r	Arithmetic shift right r
0000 111r	Rotate left r
0001 000r	Rotate right r
0001 001a	Branch unconditional
0001 010a	Branch if less than or equal to
0001 011a	Branch if less than
0001 100a	Branch if equal to
0001 101a	Branch if not equal to
0001 110a	Branch if greater than or equal to
0001 111a	Branch if greater than
0010 000a	Branch if V
0010 001a	Branch if C
0010 010a	Call subroutine
0010 011n	Unimplemented opcode, unary trap
0010 1aaa	Unimplemented opcode, nonunary trap
0011 0aaa	Unimplemented opcode, nonunary trap
0011 1aaa	Unimplemented opcode, nonunary trap
0100 0aaa	Unimplemented opcode, nonunary trap
0100 1aaa	Unimplemented opcode, nonunary trap
0101 0aaa	Add to stack pointer (SP)
0101 1aaa	Subtract from stack pointer (SP)
0110 raaa	Add to r
0111 raaa	Subtract from r
1000 raaa	Bitwise AND to r
1001 raaa	Bitwise OR to r
1010 raaa	Compare word to r
1011 raaa	Compare byte to r⟨8..15⟩
1100 raaa	Load word r from memory
1101 raaa	Load byte r⟨8..15⟩ from memory
1110 raaa	Store word r to memory
1111 raaa	Store byte r⟨8..15⟩ to memory



(a) The two parts of a nonunary instruction.

(b) A unary instruction.

FIGURE 4.7
The Pep/9 instruction format.

Example 4.1 Figure 4.6 shows that the "branch if equal to" instruction has an instruction specifier of 0001 100a. Because the letter a can be zero or one, there are really two versions of the instruction—0001 1000 and 0001 1001. Similarly, there are eight versions of the decimal output trap instruction. Its instruction specifier is 0011 1aaa, where aaa can be any combination from 000 to 111. ∎

FIGURE 4.8 summarizes the meaning of the possible fields in the instruction specifier for the letters a and r. Generally, the letter a stands for addressing mode, and the letter r stands for register. When r is 0, the instruction operates on the accumulator. When r is 1, the instruction operates on the index register. Pep/9 executes each nonunary instruction in one of eight addressing modes—immediate, direct, indirect, stack-relative, stack-relative

FIGURE 4.8
The Pep/9 instruction specifier fields.

aaa	Addressing Mode
000	Immediate
001	Direct
010	Indirect
011	Stack-relative
100	Stack-relative deferred
101	Indexed
110	Stack-indexed
111	Stack-deferred indexed

(a) The addressing-aaa field.

a	Addressing Mode
0	Immediate
1	Indexed

(b) The addressing-a field.

r	Register
0	Accumulator, A
1	Index register, X

(c) The register-r field.

deferred, indexed, stack-indexed, or stack-deferred indexed. Later chapters describe the meaning of the addressing modes. For now, it is important only that you know how to use the tables of Figures 4.7 and 4.8 to determine which register and addressing mode a given instruction uses. The meaning of the letter n in the unary trap instruction is described in a later chapter.

Example 4.2 Determine the opcode, register, and addressing mode of the 1100 1011 instruction. Starting from the left, determine with the help of Figure 4.6 that the opcode is 1100. The next bit after the opcode is the r bit, which is 1, indicating the index register. The three bits after the r bit are the aaa bits, which are 011, indicating stack-relative addressing. Therefore, the instruction loads a word from memory into the index register using stack-relative addressing. ∎

The operand specifier, for those instructions that are not unary, indicates the operand to be processed by the instruction. The CPU can interpret the operand specifier several different ways, depending on the bits in the instruction specifier. For example, it may interpret the operand specifier as an ASCII character, as an integer in two's complement representation, or as an address in main memory where the operand is stored.

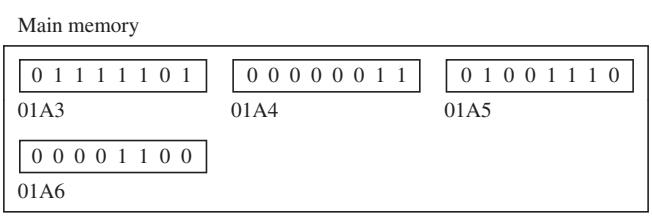
Instructions are stored in main memory. The address of an instruction in main memory is the address of the first byte of the instruction.

Example 4.3 FIGURE 4.9 shows two adjacent instructions stored in main memory at locations 01A3 and 01A6. The instruction at 01A6 is unary; the instruction at 01A3 is not.

In this example, the instruction at 01A3 has

Opcode: 0111
Register-r field: 1
Addressing-aaa field: 101
Operand specifier: 0000 0011 0100 1110

FIGURE 4.9
Two instructions in main memory.



where all the quantities are written in binary. According to the opcode chart of Figure 4.6, this is a subtract instruction. The register-r field indicates that the index register, as opposed to the accumulator, is affected. So this instruction subtracts the operand from the index register. The addressing-aaa field indicates indexed addressing, so the operand specifier is interpreted accordingly. In this chapter, we confine our study to the direct addressing mode. The other modes are taken up in later chapters.

The unary instruction at 01A6 has

Opcode: 0000 110
Register-r field: 0

The opcode indicates that the instruction will do an arithmetic shift right. The register-r field indicates that the accumulator is the register in which the shift will take place. Because this is a unary instruction, there is no operand specifier. ∎

In Example 4.3, the following form of the instructions is called *machine language*:

0111 1101 0000 0011 0100 1110
0000 1100

Machine language

Machine language is a binary sequence—that is, a sequence of ones and zeros—that the CPU interprets according to the opcodes of its instruction set. A machine language listing would show these two instructions in hexadecimal, preceded by their memory addresses, as follows:

01A3 FD034E
01A6 0C

If you have only the hexadecimal listing of an instruction, you must convert it to binary and examine the fields in the instruction specifier to determine what the instruction will do.

4.2 Direct Addressing

This section describes the operation of some of the Pep/9 instructions at Level ISA3. It describes how they operate in conjunction with the direct addressing mode. Later chapters describe the other addressing modes.

The addressing field determines how the CPU interprets the operand specifier. An addressing-aaa field of 001 indicates direct addressing. With direct addressing, the CPU interprets the operand specifier as the address

in main memory of the cell that contains the operand. In mathematical notation,

Direct addressing

$$\text{Oprnd} = \text{Mem}[\text{OprndSpec}]$$

where *Oprnd* stands for *operand*, *OprndSpec* stands for *operand specifier*, and *Mem* stands for *main memory*.

The bracket notation indicates that you can think of main memory as an array and the operand specifier as the index of the array. In C, if `v` is an array and `i` is an integer, `v[i]` is the "cell" in the array that is determined by the value of the integer `i`. Similarly, the operand specifier in the instruction identifies the cell in main memory that contains the operand.

What follows is a description of some instructions from the Pep/9 instruction set. Each description lists the opcode and gives an example of the operation of the instruction when used with the direct addressing mode. Values of N, Z, V, and C are always given in binary. Values of other registers and of memory cells are given in hexadecimal. At the machine level, all values are ultimately binary. After describing the individual instructions, this chapter concludes by showing how you can put them together to construct a machine language program.

The Stop Instruction

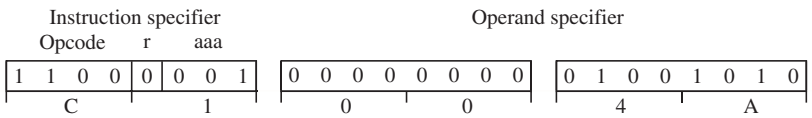
The stop instruction has instruction specifier 0000 0000. When this instruction is executed, it simply makes the computer stop. Because Pep/9 is a simulated computer, you execute it by running the Pep/9 simulator on your computer. The simulator has a menu of command options for you to choose from. One of those options is to execute your Pep/9 program. When your Pep/9 program is executing, if it encounters this instruction it will stop and return the simulator to the menu of command options. The stop execution instruction is unary. It has no operand specifier.

The Load Word Instruction

The load word instruction has instruction specifier 1100 raaa. This instruction loads one word (two bytes) from a memory location into either the accumulator or the index register, depending on the value of r. It affects the N and Z bits. If the operand is negative, it sets the N bit to 1; otherwise it clears the N bit to 0. If the operand consists of 16 0's, it sets the Z bit to 1; otherwise it clears the Z bit to 0. The register transfer language (RTL) specification of the load instruction is

$$r \leftarrow \text{Oprnd} \; ; \; N \leftarrow r < 0 \; , \; Z \leftarrow r = 0$$

FIGURE 4.10
The load word instruction.

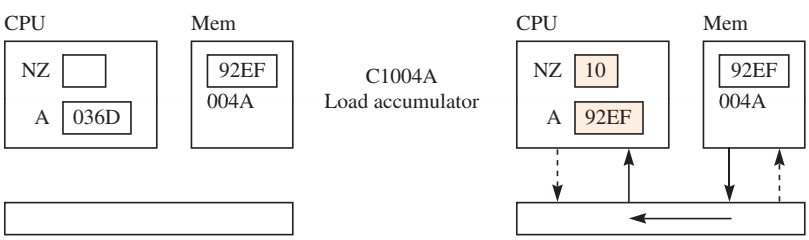


Example 4.4 Suppose the instruction to be executed is C1004A in hexadecimal, which FIGURE 4.10 shows in binary. The register-r field in this example is 0, which indicates a load word to the accumulator instead of the index register. The addressing-aaa field is 001, which indicates direct addressing.

FIGURE 4.11 shows the effect of executing the load word instruction, assuming Mem[004A] has an initial content of 92EF. The load word instruction does not change the content of the memory location. It sends a copy of the two memory cells (at addresses 004A and 004B) to the register. Whatever was in the register before the instruction was executed, in this case 036D, is destroyed. The N bit is set to 1 because the bit pattern loaded has 1 in the sign bit. The Z bit is set to 0 because the bit pattern is not all 0's. The V and C bits are unaffected by the load word instruction.

Figure 4.11 shows the data flow lines and control lines that the load word instruction activates. As indicated by the solid lines, data flows from the main memory on the bus to the CPU, and then into the register. For this data transfer to take place, the CPU must send a control signal (as indicated by the dashed lines) to main memory, telling it to put the data on the bus. The CPU also tells main memory the address from which to fetch the data. ∎

FIGURE 4.11
Execution of the load word instruction.



(a) Before. **(b)** After.

The Store Word Instruction

The store word instruction has instruction specifier 1110 raaa. This instruction stores one word (two bytes) from either the accumulator or the index register to a memory location. With direct addressing, the operand specifies the memory location in which the information is stored. The RTL specification for the store instruction is

Oprnd ← r

Example 4.5 Suppose the instruction to be executed is E9004A in hexadecimal, which FIGURE 4.12 shows in binary. This time, the register-r field indicates that the instruction will affect the index register. The addressing-aaa field, 001, indicates direct addressing.

FIGURE 4.13 shows the effect of executing the store instruction, assuming the index register has an initial content of 16BC. The store instruction does not change the content of the register. It sends a copy of the register to two memory cells (at addresses 004A and 004B). Whatever was in the memory cells before the instruction was executed, in this case F082, is destroyed. The store instruction affects none of the status bits. ∎

FIGURE 4.12
The store word instruction.

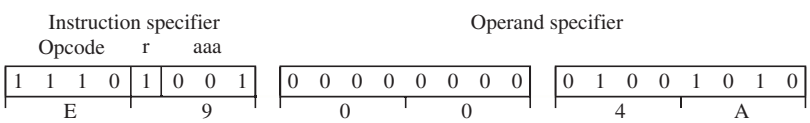
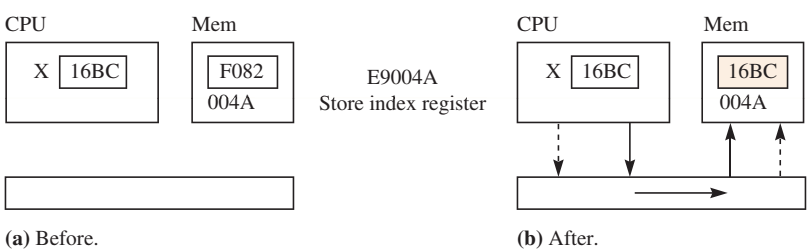


FIGURE 4.13
Execution of the store word instruction.



The Add Instruction

The add instruction has instruction specifier 0110 raaa. It is similar to the load word instruction in that data is transferred from main memory to register r in the CPU. But with the add instruction, the original content of the register is not just written over by the content of the word from main memory. Instead, the content of the word from main memory is added to the content of the register. The sum is placed in the register, and all four status bits are set accordingly. As with the load word instruction, a copy of the memory word is sent to the CPU. The original content of the memory word is unchanged. The RTL specification of the add instruction is

$$r \leftarrow r + Oprnd\ ;\ N \leftarrow r < 0\ ,\ Z \leftarrow r = 0\ ,\ V \leftarrow \{overflow\}\ ,\ C \leftarrow \{carry\}$$

Example 4.6 Suppose the instruction to be executed is 69004A in hexadecimal, which FIGURE 4.14 shows in binary. The register-r field indicates that the instruction will affect the index register. The addressing-aaa field, 001, indicates direct addressing.

FIGURE 4.15 shows the effect of executing the add instruction, assuming the index register has an initial content of 0005 and Mem[004A]

FIGURE 4.14
The add instruction.

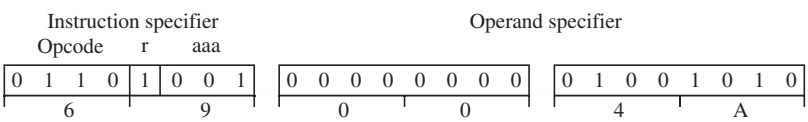
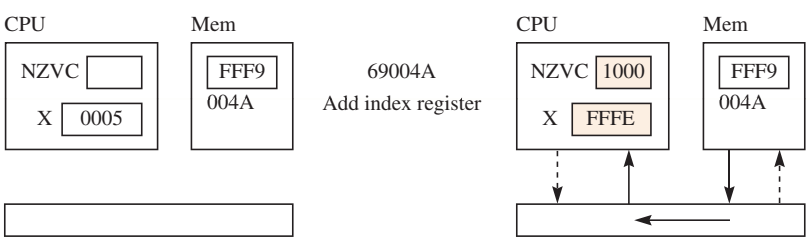


FIGURE 4.15
Execution of the add instruction.



(a) Before. **(b)** After.

has –7 (dec) = FFF9 (hex). In decimal, the sum 5 + (–7) is –2, which is shown as FFFE (hex) in Figure 4.15(b). The figure shows the NZVC bits in binary. The N bit is 1 because the sum is negative. The Z bit is 0 because the sum is not all 0's. The V bit is 0 because an overflow did not occur, and the C bit is 0 because a carry did not occur out of the most significant bit.

The Subtract Instruction

The subtract instruction has instruction specifier 0111 raaa. It is similar to the add instruction, except that the operand is subtracted from the register. The result is placed in the register, and the operand is unchanged. With subtraction, the C bit represents a carry from adding the negation of the operand. The RTL specification of the subtract instruction is

$$r \leftarrow r - Oprnd\ ;\ N \leftarrow r < 0\ ,\ Z \leftarrow r = 0\ ,\ V \leftarrow \{overflow\}\ ,\ C \leftarrow \{carry\}$$

Example 4.7 Suppose the instruction to be executed is 71004A in hexadecimal, which FIGURE 4.16 shows in binary. The register-r field indicates that the instruction will affect the accumulator.

FIGURE 4.17 shows the effect of executing the subtract instruction, assuming the accumulator has an initial content of 0003 and Mem[004A] has 0009. In decimal, the difference 3 – 9 is –6, which is shown as FFFA (hex) in Figure 4.17(b). The figure shows the NZVC bits in binary. The N bit

FIGURE 4.16
The subtract instruction.

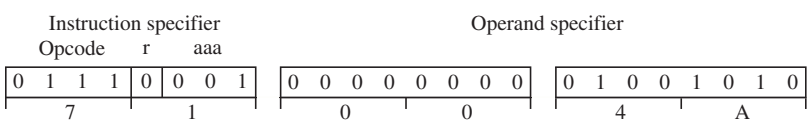
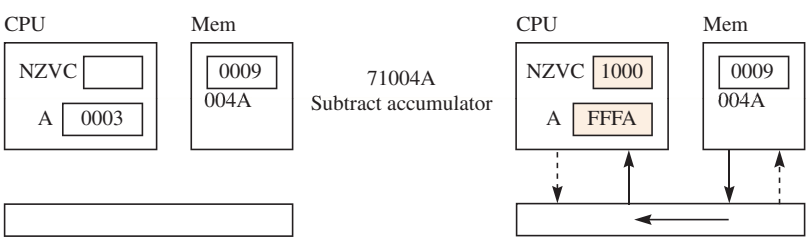


FIGURE 4.17
Execution of the subtract instruction.



(a) Before. **(b)** After.

is 1 because the sum is negative. The Z bit is 0 because the sum is not all 0's. The V bit is 0 because an overflow did not occur, and the C bit is 0 because a carry did not occur when –9 was added to 3. ∎

The And and Or Instructions

The and instruction has instruction specifier 1000 raaa, and the or instruction has instruction specifier 1001 raaa. Both instructions are similar to the add instruction. Rather than add the operand to the register, each instruction performs a logical operation on the register. The and operation is useful for masking out undesired 1 bits from a bit pattern. The or operation is useful for inserting 1 bits into a bit pattern. Both instructions affect the N and Z bits and leave the V and C bits unchanged. The RTL specifications for the and and or instructions are

$$r \leftarrow r \wedge Oprnd\ ;\ N \leftarrow r < 0\ ,\ Z \leftarrow r = 0$$
$$r \leftarrow r \vee Oprnd\ ;\ N \leftarrow r < 0\ ,\ Z \leftarrow r = 0$$

Example 4.8 Suppose the instruction to be executed is 89004A in hexadecimal, which FIGURE 4.18 shows in binary. The opcode indicates that the and instruction will execute and the register-r field indicates that the instruction will affect the index register.

FIGURE 4.19 shows the effect of executing the and instruction, assuming the index register has an initial content of 5DC3 and Mem[004A] has 00FF. In binary, 00FF is 0000 0000 1111 1111. At every position where there is a 1 in Mem[004A], the corresponding bit in the index register is unchanged. At every position where there is a 0, the corresponding bit is cleared to 0. The figure shows the NZ bits in binary. The N bit is 0 because the quantity in the index register is not negative when interpreted as a signed integer. The Z bit is 0 because the index register is not all 0's. ∎

Example 4.9 FIGURE 4.20 shows the operation of the or instruction. The initial state is identical to that of Example 4.8, except that the opcode of the instruction specifier 99 is 1001, which indicates the or instruction. This time, at every position where there is a 0 in Mem[004A], the corresponding

FIGURE 4.18
The and instruction.

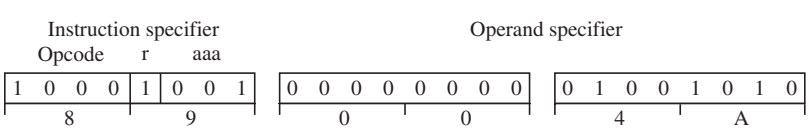
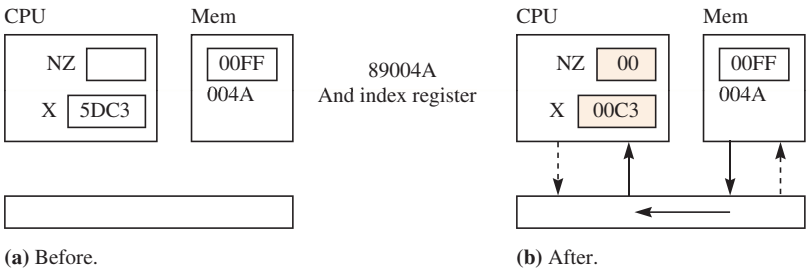
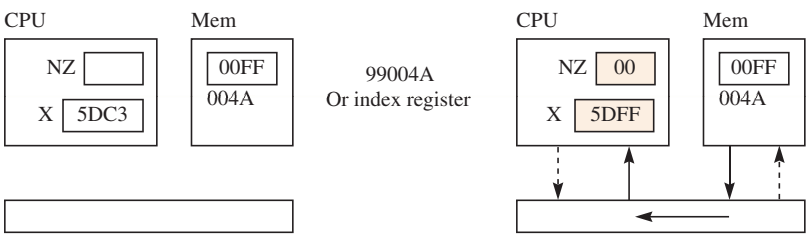


FIGURE 4.19
Execution of the and instruction.



(a) Before. **(b)** After.

FIGURE 4.20
Execution of the or instruction.



(a) Before. **(b)** After.

bit in the index register is unchanged. At every position where there is a 1, the corresponding bit is set to 1. The N bit is 0 because the index register would not be negative if it were interpreted as a signed integer.

The Invert and Negate Instructions

The invert instruction has instruction specifier 0000 011r, and the negate instruction has instruction specifier 0000 100r. Both instructions are unary. They have no operand specifier. The invert instruction performs the NOT operation on the register. That is, each 1 is changed to 0, and each 0 is changed to 1. It affects the N and Z bits. The RTL specification of the invert instruction is

$$r \leftarrow \neg r\,;\, N \leftarrow r < 0\,,\, Z \leftarrow r = 0$$

The negate instruction interprets the register as a signed integer and negates it. The 16-bit register stores signed integers in the range –32768 to 32767. The negate instruction affects the N, Z, and V bits. The V bit is set only if the original value in the register is –32768, because there is no corresponding positive value of 32768. The RTL specification of the negate instruction is

$$r \leftarrow -r\,;\, N \leftarrow r < 0\,,\, Z \leftarrow r = 0\,,\, V \leftarrow \{overflow\}$$

Example 4.10 Suppose the instruction to be executed is 06 in hexadecimal, which FIGURE 4.21 shows in binary. The opcode indicates that the invert instruction will execute, and the register-r field indicates that the instruction will affect the accumulator.

FIGURE 4.22 shows the effect of executing the invert instruction, assuming the accumulator has an initial content of 0003 (hex), which is 0000 0000 0000 0011 (bin). The not instruction changes the bit pattern to 1111 1111 1111 1100. The N bit is 1 because the quantity in the accumulator is negative when interpreted as a signed integer. The Z bit is 0 because the accumulator is not all 0's. ∎

Example 4.11 FIGURE 4.23 shows the operation of the negate instruction. The initial state is identical to that of Example 4.10, except that the opcode of the instruction specifier 1A is 0000 100, which indicates the negate instruction. The negation of 3 is –3, which is 1111 1111 1111 1101 (bin) = FFFD (hex). ∎

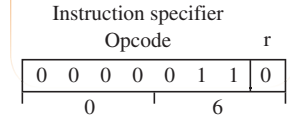


FIGURE 4.21
The invert instruction.

FIGURE 4.22
Execution of the invert instruction.

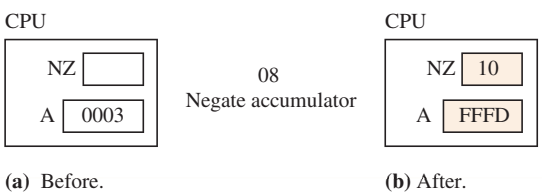


FIGURE 4.23
Execution of the negate instruction.

The Load Byte and Store Byte Instructions

These instructions, along with the two that follow, are byte instructions. Byte instructions operate on a single byte of information instead of a word. The load byte instruction has instruction specifier 1101 raaa, and the store byte instruction has instruction specifier 1111 raaa. The load byte instruction loads the operand into the right half of either the accumulator or the index register, and affects the N and Z bits. It leaves the left half of the register unchanged. The store byte instruction stores the right half of either the accumulator or the index register into a one-byte memory location and does not affect any status bits. The RTL specification of the load byte instruction is

$$r\langle 8..15\rangle \leftarrow \text{byte Oprnd} ; N \leftarrow 0 , Z \leftarrow r\langle 8..15\rangle = 0$$

and the RTL specification of the store byte instruction is

$$\text{byte Operand} \leftarrow r\langle 8..15\rangle$$

Example 4.12 Suppose the instruction to be executed is D1004A in hexadecimal, which FIGURE 4.24 shows in binary. The register-r field in this example is 0, which indicates a load to the accumulator instead of the index register. The addressing-aaa field is 001, which indicates direct addressing.

FIGURE 4.25 shows the effect of executing the load byte instruction, assuming Mem[004A] has an initial content of 92. The N bit is always set to

FIGURE 4.24
The load byte instruction.

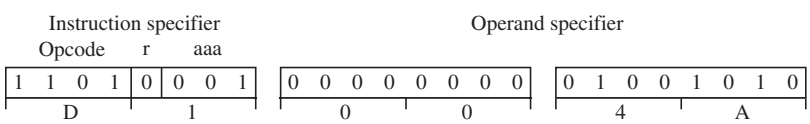
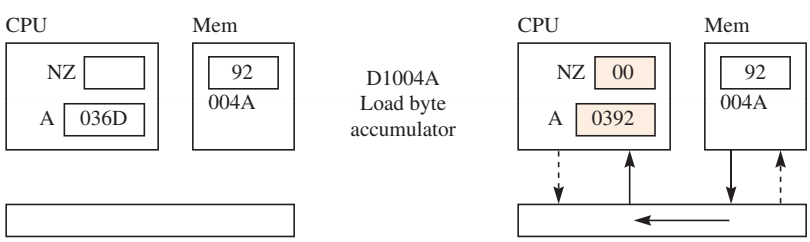
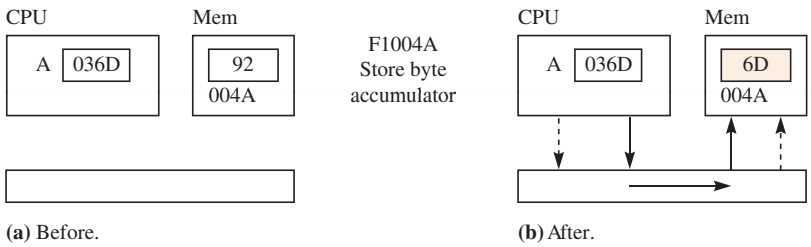


FIGURE 4.25
Execution of the load byte instruction.



(a) Before. **(b)** After.

FIGURE 4.26
Execution of the store byte instruction.



(a) Before. **(b)** After.

0 with this instruction. The Z bit is set to 0 because the eight bits loaded into the right half of the accumulator are not all 0's.

Example 4.13 FIGURE 4.26 shows the effect of executing the store byte instruction. The initial state is the same as in Example 4.12, except that the instruction is store byte instead of load byte. The right half of the accumulator is 6D, which is sent to the memory cell at address 004A.

The Input and Output Devices

The input device is at address FC15, and is attached to an ASCII character input device like a keyboard. You get a character from the input device by executing the load byte instruction from address FC15. The output device is at address FC16 and is attached to an ASCII output device like a screen. You send a character to the output device by executing the store byte instruction to address FC16.

Example 4.14 Suppose the instruction to be executed is D1FC15 in hexadecimal, which is the load byte instruction from the input device with direct addressing. FIGURE 4.27 shows the effect of executing the instruction, assuming that the next character in the input stream is W. The character from the input stream can come from the keyboard or from a file. The figure shows the keyboard wired into the memory location at address FC15. The user has pressed the W key. The ASCII value of the letter W is 57 (hex), which is sent to the accumulator.

The dashed lines from the CPU to main memory represent control signals that instruct the memory subsystem to put the byte from address FC15 onto the system bus. The memory subsystem has a special input circuit that detects when a memory load request is made from address FC15. It then performs all the necessary steps to put the next character from the input stream into Mem[FC15], which is then put on the system bus. This is an example of levels of abstraction in a computer system. The details of how the

FIGURE 4.27
The load byte instruction from the input device.

(a) Before. **(b)** After.

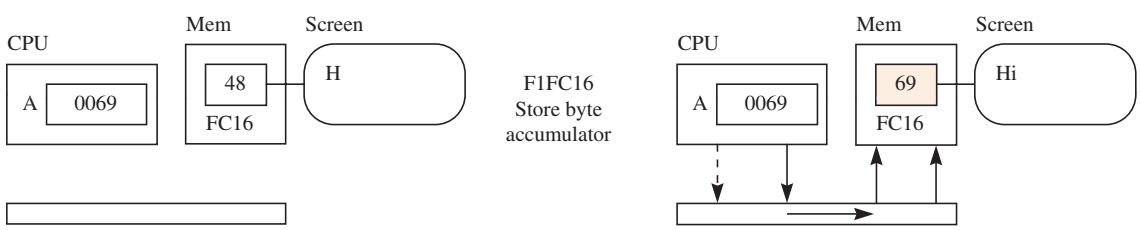
character is transferred from the keyboard to Mem[FC15] are hidden from the Level ISA3 programmer, who only needs to know that to get the next ASCII character from the input stream you load a byte from that address.

Example 4.15 Suppose the instruction to be executed is F1FC16 in hexadecimal, which is the store byte instruction to the output device with direct addressing. FIGURE 4.28 shows the effect of executing the instruction, assuming that 69 (hex), which is the ASCII value for the letter i, is in the right half of the accumulator. The figure shows the screen wired into the memory location at address FC16. The ASCII value of the letter i is sent to Mem[FC16]. As with the input device, the memory subsystem has a special circuit that detects when a byte is stored to Mem[FC16] and routes it to the output stream to be displayed on the screen.

Big Endian Versus Little Endian

There are two CPU design philosophies regarding the transfer of information between the registers of the CPU and the bytes in main memory. The problem arises because main memory is always byte-addressable and a

FIGURE 4.28
The store byte instruction to the output device.



(a) Before. **(b)** After.

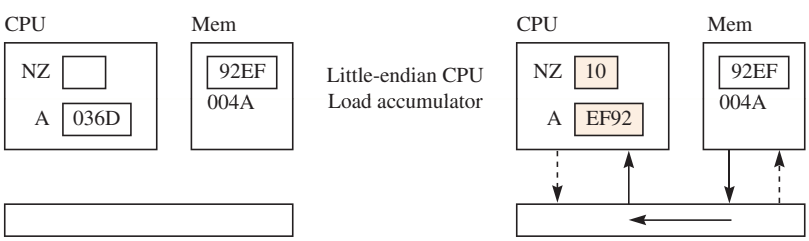
register in a CPU typically contains more than one byte. The design question is, in what order should the sequence of bytes be stored in main memory? There are two choices. The CPU can store the most significant byte at the lowest address, called *big-endian order*, or it can store the least significant byte at the lowest address, called *little-endian order*. The choice of which order to use is arbitrary as long as the same order is used consistently for all instructions in the instruction set.

There is no dominant standard in the computing industry. Some processors use big-endian order, some use little-endian order, and some can use either order depending on a switch that is set by low-level software. Pep/9 is a big-endian CPU. Figure 4.13 shows the effect of the store instruction. The most significant byte in the register is 16, which is stored at the lowest address, 004A. The next byte in the register is BC and is stored at the next higher address, 004B. Figure 4.11 shows the load instruction, which is consistent. The most significant byte of the register gets 92, which is the byte from the lowest address at 004A. The next byte gets EF from the next higher address at 004B.

In contrast, FIGURE 4.29 shows what happens when a load instruction executes in a little-endian CPU. The byte at the lowest address, 004A, is 92 and is put in the least significant byte of the register. The byte from the next higher address, 004B, is put to the left of the low-order byte in the register. FIGURE 4.30 shows the effect of a load instruction in a CPU with 32-bit registers for both big-endian and little-endian ordering. A 32-bit register holds four bytes, which are loaded into the accumulator from most significant to least significant byte, or from least significant to most significant byte, depending on whether the CPU uses big-endian or little-endian ordering, respectively.

The word *endian* comes from Jonathan Swift's 1726 novel *Gulliver's Travels*, in which two competing kingdoms, Lilliput and Blefuscu, have different customs for breaking eggs. The inhabitants of Lilliput break their eggs at the little end, and hence are known as *little endians*, while the inhabitants of

FIGURE 4.29
The load instruction in a little-endian CPU.



(a) Before. **(b)** After.

FIGURE 4.30
The load instruction with a 32-bit register.

	Initial State	Big Endian Final State	Little Endian Final State
Mem[019E]	89	89	89
Mem[019F]	AB	AB	AB
Mem[01A0]	CD	CD	CD
Mem[01A1]	EF	EF	EF
Accumulator		89 AB CD EF	EF CD AB 89

Blefuscu break their eggs at the big end, and hence are known as *big endians*. The novel is a parody reflecting the absurdity of war over meaningless issues. The terminology is fitting, as whether a CPU is big-endian or little-endian is of little fundamental importance.

4.3 von Neumann Machines

In the earliest electronic computers, each program was hand-wired. To change the program, the wires had to be manually reconnected, a tedious and time-consuming process. The Electronic Numerical Integrator and Calculator (ENIAC) computer described in Section 3.1 was an example of this kind of machine. Its memory was used only to store data.

In 1945, John von Neumann had proposed in a report published by the University of Pennsylvania that the United States Ordnance Department build a computer that would store in main memory not only the data, but the program as well. The stored-program concept was a radical idea at the time. Maurice V. Wilkes built the Electronic Delay Storage Automatic Calculator (EDSAC) at Cambridge University in England in 1949. It was the first computer to be built that used von Neumann's stored-program idea. Practically all commercial computers today are based on the stored-program concept, with programs and data sharing the same main memory. Such computers are called *von Neumann machines*, although some believe that J. Presper Eckert, Jr., originated the idea several years before von Neumann's paper.

The von Neumann Execution Cycle

The Pep/9 computer is a classic von Neumann machine. FIGURE 4.31 is a pseudocode description of the steps required to execute a program:

FIGURE 4.31
A pseudocode description of the steps necessary to execute a program on the Pep/9 computer.

```
Load the machine language program
Initialize PC and SP
do {
    Fetch the next instruction
    Decode the instruction specifier
    Increment PC
    Execute the instruction fetched
}
while (the stop instruction does not execute)
```

The `do` loop is called the *von Neumann execution cycle*. The cycle consists of five operations:

The von Neumann execution cycle

- Fetch instruction at Mem[PC].
- Decode instruction fetched.
- Increment PC.
- Execute instruction fetched.
- Repeat the cycle.

The von Neumann cycle is wired into the CPU. The following is a more detailed description of the steps in the execution process.

Load the program

To load the machine language program into main memory, the first instruction is placed at address 0000 (hex). The second instruction is placed adjacent to the first. If the first instruction is unary, then the address of the second instruction is 0001. Otherwise the operand specifier of the first instruction will be contained in the bytes at 0001 and 0002. The address of the second instruction would then be at 0003. The third instruction is placed adjacent to the second similarly, and so on for the entire machine language program.

Initialize PC and SP

To initialize the program counter and stack pointer, PC is set to 0000 (hex), and SP is set to Mem[FFF4]. The purpose of the program counter is to hold the address of the next instruction to be executed. Because the first instruction was loaded into main memory at address 0000, the PC must be set initially to 0000. The purpose of the stack pointer is to hold the address of the top of the run-time stack. A later section explains why SP is set to Mem[FFF4].

Fetch instruction

The first operation in the von Neumann execution cycle is fetch. To fetch an instruction, the CPU examines the 16 bits in the PC and interprets them as an address. It then goes to that address in main memory to fetch the instruction specifier (one byte) of the next instruction. It brings the eight bits of the instruction specifier into the CPU and holds them in the first byte of the instruction register (IR).

Decode instruction specifier

The second operation in the von Neumann cycle is decode. The CPU extracts the opcode from the instruction specifier to determine which instruction to execute. Depending on the opcode, the CPU extracts the register specifier if there is one and the addressing field if there is one. Now the CPU knows from the opcode whether the instruction is unary. If it is not unary, the CPU fetches the operand specifier (one word) from memory and stores it in the last two bytes of the IR.

Increment PC

The third operation in the von Neumann execution cycle is increment. The CPU adds 0001 to the PC if the instruction was unary. Otherwise it adds 0003. Regardless of which number is added to the PC, its value after the addition will be the address of the following instruction because the instructions are loaded adjacent to one another in main memory.

Execute instruction fetched

The fourth operation in the von Neumann execution cycle is execute. The CPU executes the instruction that is stored in the IR. The opcode tells the CPU which of the 40 instructions to execute.

Repeat the cycle

The fifth operation in the von Neumann execution cycle is repeat. The CPU returns to the fetch operation unless the instruction just executed was the stop instruction. Pep/9 will also terminate at this point if the instruction attempts an illegal operation. Some instructions are not allowed to use certain addressing modes. The most common illegal operation that makes Pep/9 terminate is attempting execution of an instruction with a forbidden addressing mode.

FIGURE 4.32 is a more detailed pseudocode description of the steps to execute a program on the Pep/9 computer.

A Character Output Program

The Pep/9 system can take its input from the keyboard and send its output to the screen. These I/O devices are based on the ASCII character set. When you press a key, a byte of information representing a single ASCII character goes from the keyboard and is added to the input stream at the input device at Mem[FC15]. When the CPU sends a byte to the output device at Mem[FC16], the screen interprets the byte as an ASCII character, which it displays.

At Level ISA3, the machine level, computers usually have no input or output instructions for any type of data except bytes. The interpretation of the byte occurs in the input or output device, not in main memory. Pep/9's

FIGURE 4.32
A more detailed pseudocode description of the steps necessary to execute a program on the Pep/9 computer.

```
Load the machine language program into memory starting at address 0000.
PC ← 0000
SP ← Mem[FFF4]
do {
    Fetch the instruction specifier at address in PC
    PC ← PC + 1
    Decode the instruction specifier
    if (the instruction is not unary) {
        Fetch the operand specifier at address in PC
        PC ← PC + 2
    }
    Execute the instruction fetched
}
while ((the stop instruction does not execute) && (the instruction is legal))
```

only input instruction is load byte, which transfers a byte from the input device to a CPU register, and its only output instruction is store byte, which transfers a byte from a CPU register to the output device. Because these bytes are usually interpreted as ASCII characters, the I/O at Level ISA3 of the Pep/9 system is called *character I/O*.

FIGURE 4.33 shows a simple machine-language program that outputs the characters `Hi` on the output device. It uses three instructions: 1101 raaa, which is the load byte instruction from a memory location, 1111 raaa, which is the store byte instruction to the output device, and 0000 0000, which is the stop instruction. The first listing shows the machine language program in binary. Main memory stores this sequence of ones and zeros. The first column gives the address in hex of the first byte of the bit pattern on each line.

The second listing shows the same program abbreviated to hexadecimal. Even though this format is slightly easier to read, remember that memory stores bits, not literal hexadecimal characters as in the second listing. Each line in the listing has a comment that begins with a semicolon to separate it from the machine language. The comments are not loaded into memory with the program.

FIGURE 4.34 shows each step the computer takes to execute the first three instructions of the program. Figure 4.34(a) is the initial state of the Pep/9 computer. Neither the disk nor the input device is shown. Several of

FIGURE 4.33
A machine language program to output the characters `Hi`.

```
Address     Machine Language (bin)
0000        1101 0001 0000 0000 0000 1101
0003        1111 0001 1111 1100 0001 0110
0006        1101 0001 0000 0000 0000 1110
0009        1111 0001 1111 1100 0001 0110
000C        0000 0000
000D        0100 1000 0110 1001

Address     Machine Language (hex)
0000        D1000D ;Load byte accumulator 'H'
0003        F1FC16 ;Store byte accumulator output device
0006        D1000E ;Load byte accumulator 'i'
0009        F1FC16 ;Store byte accumulator output device
000C        00     ;Stop
000D        4869   ;ASCII "Hi" characters


Output
Hi
```

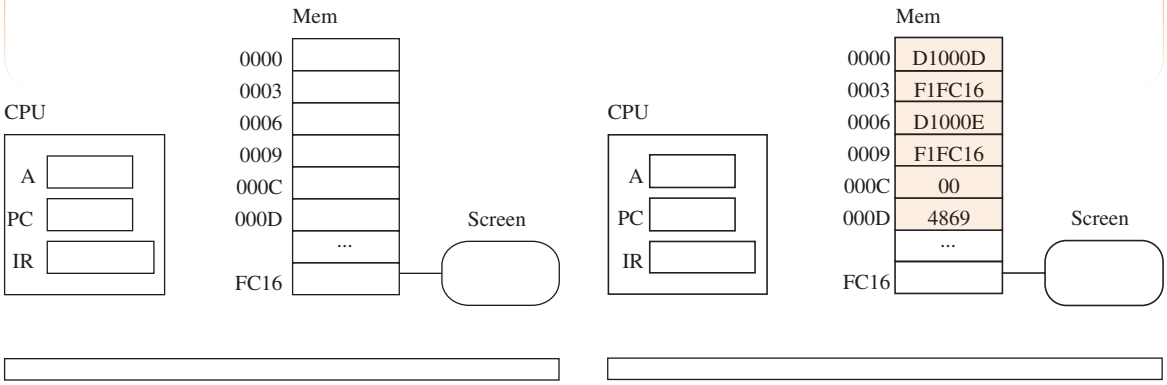
the CPU registers not used by this program are also omitted. Initially, the contents of the main memory cells and the CPU registers are unknown.

Figure 4.34(b) shows the first step of the process. The program is loaded into main memory, starting at address 0000. The details of where the program comes from and what puts it into memory are described in later chapters.

Figure 4.34(c) shows the second step of the process. The program counter is cleared to 0000 (hex). The figure does not show the initialization of SP because this program does not use the stack pointer.

Figure 4.34(d) shows the fetch part of the execution cycle. The CPU examines the bits in the PC and finds 0000 (hex). It signals the main memory to send the byte at that address to the CPU. When the CPU gets the byte, it stuffs it into the first part of the instruction register. Then it decodes the instruction specifier, determines from the opcode that the instruction is not unary, and brings the operand specifier into IR as well. The original bits at addresses 0000, 0001, and 0002 are not changed by the fetch. Main memory has sent a copy of the 24 bits to the CPU.

FIGURE 4.34
The von Neumann execution cycle for the program of Figure 4.33.



(a) Initial state before loading.

(b) Program loaded into main memory.

(c) PC ← 0000 (hex).

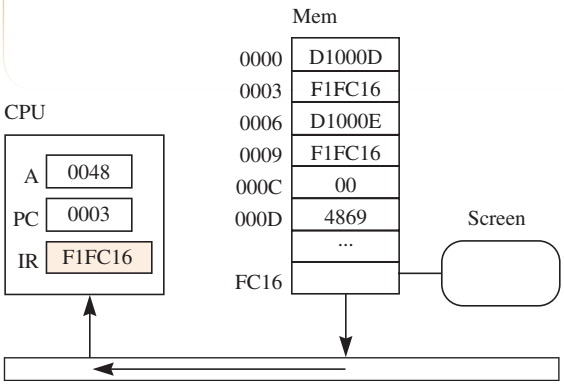
(d) Fetch instruction at Mem[PC].

(e) Increment PC.

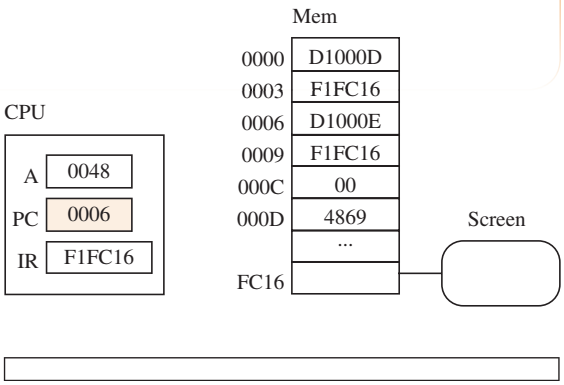
(f) Execute. Load byte for H to accumulator.

(*continues*)

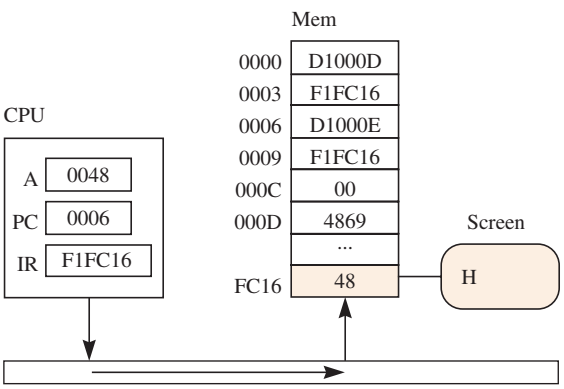
FIGURE 4.34
The von Neumann execution cycle for the program of Figure 4.33. (*continued*)



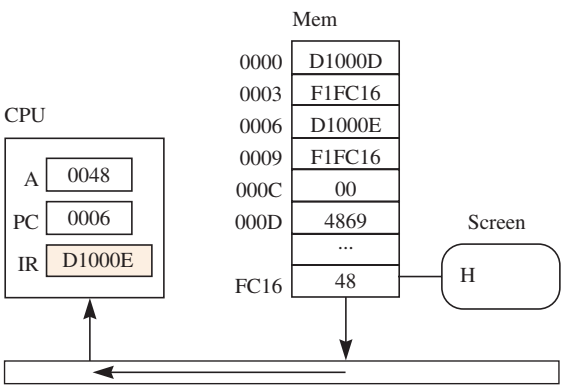
(g) Fetch instruction at Mem[PC].



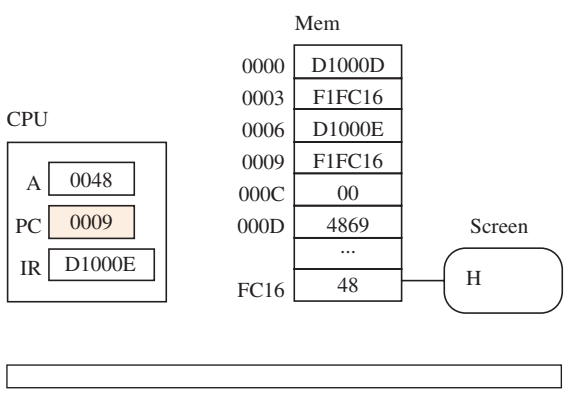
(h) Increment PC.



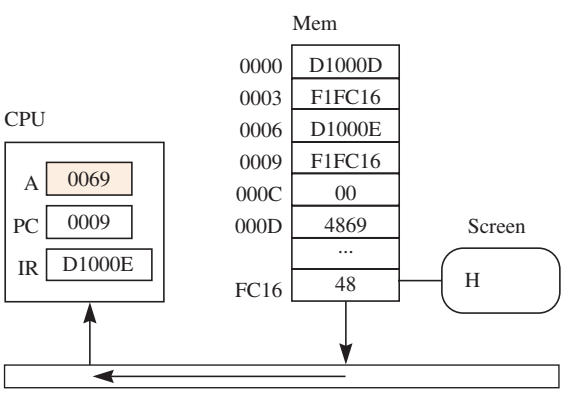
(i) Execute. Store byte from accumulator to output device.



(j) Fetch instruction at Mem[PC].



(k) Increment PC.



(l) Execute. Load byte for i to accumulator.

Figure 4.34(e) shows the increment part of the execution cycle. The CPU adds 0003 to the PC.

Figure 4.34(f) shows the execute part of the execution cycle. The CPU examines the first four bits of IR and finds 1101. This opcode signals the circuitry to execute the load byte instruction. Consequently, the CPU examines the r-field and finds 0, which indicates the accumulator, and the addressing-aaa field and finds 001, which indicates direct addressing. It then examines the operand specifier and finds 000D (hex). It sends a control signal to main memory to go directly to address 000D and put the byte at that address on the system bus. The CPU then retrieves the value from the system bus and puts it into the right half of the accumulator.

Figure 4.34(g) shows the fetch part of the execution cycle. This time the CPU finds 0003 (hex) in the PC. It fetches a copy of the byte at address 0003, determines that the instruction is not unary, and then fetches the word at 0004. As a result, the original content of IR is destroyed.

Figure 4.34(h) shows the increment part of the execution cycle. The CPU adds 0003 to PC, making it 0006 (hex).

Figure 4.34(i) shows the execute part of the execution cycle. The CPU examines the first four bits of IR and finds 1111. This opcode signals the circuitry to execute the store byte instruction. Consequently, the CPU examines the r-field and finds 0, which indicates the accumulator, and the addressing-aaa field and finds 001, which indicates direct addressing. It then examines the operand specifier and finds FC16 (hex). It puts the byte from the right half of the accumulator on the bus and sends a control signal to main memory to store the data from the bus at address FC16, which is the output device. The output device interprets the byte as the ASCII character H and displays the character on the screen.

Figure 4.34(j) shows the fetch part of the execute cycle. Because PC contains 0006 (hex), the byte at that address comes to the CPU. When the CPU examines the opcode, it discovers that the instruction is not unary, so it fetches the word at address 0007.

Figure 4.34(k) shows the increment part of the execution cycle. The CPU adds 0003 to PC, making it 0009 (hex).

Figure 4.34(l) shows the execute part of the execution cycle. As with part (f), the CPU executes the load byte instruction—but this time loads from address 000E, which contains the ASCII code for the letter i. The figure does not show exection of the last two instructions. The following instruction sends the letter i to the output device, and the instruction after that causes the program to halt.

Just outputting two characters may seem a rather involved process, but it all happens quickly in human terms. The fetch part of the execution cycle takes less than about one nanosecond on many computers. Because the

execution part of the execution cycle depends on the particular instruction, a complex instruction may take many nanoseconds to execute, whereas a simple instruction may take a few nanoseconds.

The computer does not attach any meaning to the electrical signals in its circuitry. Specifically, main memory does not know whether the bits at a particular address represent data or an instruction. It remembers only individual 1's and 0's.

von Neumann Bugs

Executing data as instructions

In the program of Figure 4.33, the bits at addresses 0000 to 000C are used by the CPU as instructions, and the bits at 000D and 000E are used as data. The programmer placed the instruction bits at the beginning because she knew the PC would be initially cleared to 0000 and would be incremented by 0001 or 0003 on each iteration of the execution cycle. If the stop instruction (opcode 0000 0000) were omitted by mistake, the execution cycle would continue to fetch the next byte and interpret it as the instruction specifier of the next instruction, even though the programmer intended to have it interpreted as data.

Because programs and data share the same memory, programmers at the machine level must be careful in allocating memory for each of them. Otherwise two types of problems can arise. The CPU may interpret a sequence of bits as an instruction when the programmer intended them to be data. Or the CPU may interpret a sequence of bits to be data when the programmer intended them to be an instruction. Both types of bugs occur at the machine level.

Interpreting instructions as data

Although the sharing of memory by both data and instructions can produce bugs if the programmer is not careful, it also presents an exciting possibility. A program is simply a set of instructions that is stored in memory. The programmer, therefore, can view the program as data for yet another program. It becomes possible to write programs that process other programs. Compilers, assemblers, and loaders are programs that adopt this viewpoint of treating other programs as data.

A Character Input Program

The program of FIGURE 4.35 inputs two characters from the input device and outputs them in reverse order on the output device. It uses the character input instruction with direct addressing to get the characters from the input device.

The first instruction, D1FC15, has an opcode that specifies load byte from the input device at Mem[FC15], register-r field that specifies the accumulator, and addressing-aaa field that specifies direct addressing.

FIGURE 4.35
A machine language program to input two characters and output them in reverse order.

```
Address     Machine Language (bin)
0000        1101 0001 1111 1100 0001 0101
0003        1111 0001 0000 0000 0001 0011
0006        1101 0001 1111 1100 0001 0101
0009        1111 0001 1111 1100 0001 0110
000C        1101 0001 0000 0000 0001 0011
000F        1111 0001 1111 1100 0001 0110
0012        0000 0000

Address     Machine Language (hex)
0000        D1FC15   ;Input first character
0003        F10013   ;Store first character
0006        D1FC15   ;Input second character
0009        F1FC16   ;Output second character
000C        D10013   ;Load first character
000F        F1FC16   ;Output first character
0012        00       ;Stop

Input
up

Output
pu
```

It puts the first character from the input device into the right half of the accumulator. The second instruction, F10013, has an opcode that specifies store byte from the accumulator to Mem[0013]. Although this byte is not shown on the listing, it is surely available because memory goes all the way up to address FFFF.

The third instruction, D1FC15, is identical to the first and inputs the second character into the right half of the accumulator. The fourth instruction, F1FC16, has an opcode that specifies store byte, a register-r field that specifies the accumulator, and an addressing-aaa field that specifies direct addressing. It sends the byte from the accumulator to the output device at Mem[FC16]. The fifth instruction, D10013, loads the first character previously stored in Mem[0013] into the accumulator. The penultimate instruction, F1FC16, sends the second character from

the accumulator to the output device at Mem[FC16]. The last instruction halts the program.

Converting Decimal to ASCII

FIGURE 4.36 shows a program that adds two single-digit numbers and outputs their single-digit sum. It illustrates the inconvenience of dealing with output at the machine level.

The two numbers to be added are 5 and 3. The program stores them at Mem[000D] and Mem[000F]. The first instruction loads the 5 into the accumulator, and then the second instruction adds the 3. At this point the sum is in the accumulator.

Now a problem arises. We want to output this result, but the only output instruction for this Level ISA3 machine is to store a byte in ASCII format to

FIGURE 4.36
A machine language program to add 5 and 3 and output the single-character result.

```
Address     Machine Language (bin)
0000        1100 0001 0000 0000 0000 1101
0003        0110 0001 0000 0000 0000 1111
0006        1001 0001 0000 0000 0001 0001
0009        1111 0001 1111 1100 0001 0110
000C        0000 0000
000D        0000 0000 0000 0101
000F        0000 0000 0000 0011
0011        0000 0000 0011 0000

Address     Machine Language (hex)
0000        C1000D ;A<-first number
0003        61000F ;Add the two numbers
0006        910011 ;Convert sum to character
0009        F1FC16 ;Output the character
000C        00     ;Stop
000D        0005   ;Decimal 5
000F        0003   ;Decimal 3
0011        0030   ;Mask for ASCII char

Output
8
```

the output device at Mem[FC16]. The problem is that our result is 0000 1000 (bin). If the store byte instruction tries to output that, it will be interpreted as the backspace character, BS. (See the ASCII chart of Figure 3.25).

So, the program must convert the decimal number 8, 0000 1000 (bin), to the ASCII character 8, 0011 1000 (bin). The ASCII bits differ from the unsigned binary bits by the two extra 1's in the third and fourth bits. To do the conversion, the program inserts those two extra 1's into the result by ORing the accumulator with the mask 0000 0000 0011 0000, using the OR register instruction:

```
        0000 0000 0000 1000
OR      0000 0000 0011 0000
        0000 0000 0011 1000
```

The accumulator now contains the correct sum in ASCII form. The store byte instruction sends it to the output device.

If you replace the word at Mem[0013] with 0009, what does this program output? Unfortunately, it does not output 14, even though the sum in the accumulator is

14 (dec) = 0000 0000 0000 1110 (bin)

after the add accumulator instruction executes. The OR instruction changes this bit pattern to 0000 0000 0011 1110 (bin), producing an output of >. Because the only output instruction at Level ISA3 is one that outputs a single byte, the program cannot output a result that should contain more than one character. Chapter 5 shows how to remedy this shortcoming.

A Self-Modifying Program

FIGURE 4.37 illustrates a curious possibility based on the von Neumann design principle. Notice that the program from 0006 to 0017 is identical to Figure 4.36 from 0000 to 0011. This program has two instructions at the beginning that are not in Figure 4.36, however. Because the instructions are shifted down six bytes, their operand specifiers are all greater by six than the operand specifiers of the previous program. Other than the adjustment by six bytes, however, the instructions beginning at 0006 would appear to duplicate the processing of Figure 4.36.

In particular, it appears that the load accumulator instruction would load the 5 into the accumulator, the add instruction would add the 3, the OR instruction would change the 8 (dec) to ASCII 8, and the store byte accumulator instruction would send the ASCII character for 8 to the output device at Mem[FC16]. Instead, the output is 2.

Because program and data share the same memory in a von Neumann machine, it is possible for a program to treat itself as data and modify

FIGURE 4.37
A machine language program that modifies itself. The add instruction changes to a subtract instruction.

```
Address     Machine Language (bin)
0000        1101 0001 0000 0000 0001 1001
0003        1111 0001 0000 0000 0000 1001
0006        1100 0001 0000 0000 0001 0011
0009        0110 0001 0000 0000 0001 0101
000C        1001 0001 0000 0000 0001 0111
000F        1111 0001 1111 1100 0001 0110
0012        0000 0000
0013        0000 0000 0000 0101
0015        0000 0000 0000 0011
0017        0000 0000 0011 0000
0019        0111 0001

Address     Machine Language (hex)
0000        D10019 ;Load byte accumulator
0003        F10009 ;Store byte accumulator
0006        C10013 ;A<-first number
0009        610015 ;Add the two numbers
000C        910017 ;Convert sum to character
000F        F1FC16 ;Output the character
0012        00     ;Stop
0013        0005   ;Decimal 5
0015        0003   ;Decimal 3
0017        0030   ;Mask for ASCII char
0019        71     ;Byte to modify instruction

Output
2
```

itself. The first instruction loads the byte 71 (hex) into the right half of the accumulator, and the second instruction puts it in Mem[0009]. What was at Mem[0009] before this change? The instruction specifier of the add accumulator instruction. Now the bits at Mem[0009] are 0111 0001. When the computer gets these bits in the fetch part of the von Neumann execution cycle, the CPU detects the opcode as 0111, the opcode for the subtract register instruction. The register specifier indicates the accumulator, and the addressing mode bits indicate direct addressing. The instruction subtracts 3 from 5 instead of adding it.

Of course, this is not a very practical program. If you wanted to subtract the two numbers, you would simply write the program of Figure 4.36 with the subtract instruction in place of the add instruction. But it does show that in a von Neumann machine, main memory places no significance on the bits it is storing. It simply remembers 1's and 0's and has no idea which are program bits, which are data bits, which are ASCII characters, and so on. Furthermore, the CPU cranks out the von Neumann execution cycle and interprets the bits accordingly, with no idea of their history. When it fetches the bits at Mem[0009], it does not know, or care, how they got there in the first place. It simply repeats the fetch, decode, increment, execute cycle over and over.

The x86 Architecture

The designation *x86* refers to a family of processors beginning with the 8086 introduced by Intel in 1978 and continuing with the 80186, 80286, 80386, 80486 series; the Pentium series; and the Core series. The CPU registers vary in size from 16 bits in the 8086, to 32 bits in the 80386, to 64 bits beginning with the Pentium 4. The processors are generally backward compatible. For example, the 64-bit processors have a 32-bit execution mode so that older software can run unchanged on the newer CPUs.

FIGURE 4.38 shows the registers in a typical 32-bit model. The x86 processors are little-endian and number the bits in a register starting from 0 for the least significant bit. The EFLAGS register has a number

FIGURE 4.38
The registers in a typical 32-bit x86 CPU.

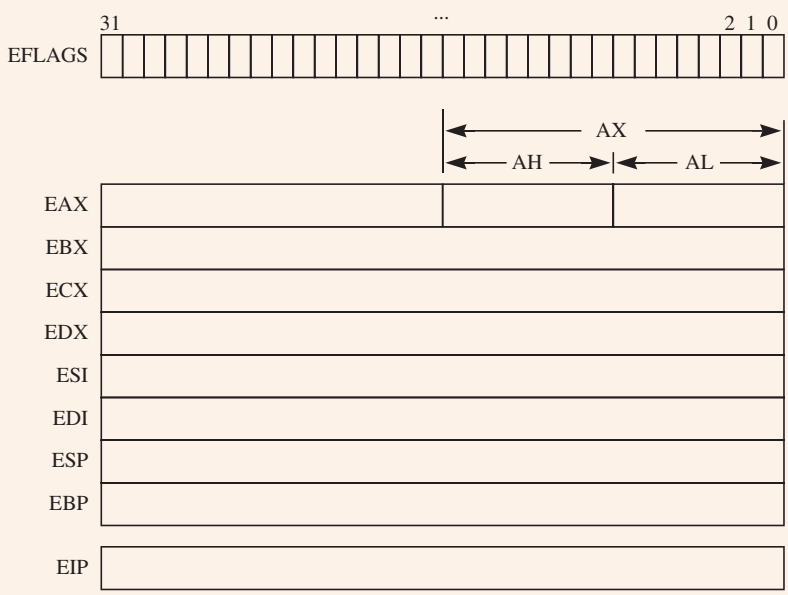


FIGURE 4.39
The x86 status bits corresponding to the Pep/9 status bits.

Status Bit	Intel Name	EFLAGS Position
N	SF	7
Z	ZF	6
V	OF	11
C	CF	0

of status bits besides the four NZVC bits in Pep/9. FIGURE 4.39 shows the location of the four bits that correspond to the Pep/9 status bits. *SF* stands for *sign flag*, and *OF* stands for *overflow flag*.

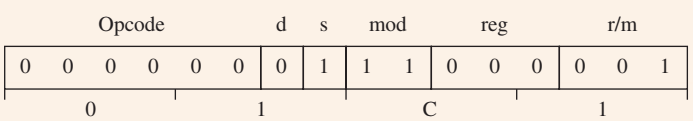
The x86 architecture has four general-purpose accumulators named *EAX*, *EBX*, *ECX*, and *EDX* that correspond to the single Pep/9 accumulator. Figure 4.38 shows that the rightmost two bytes of the EAX register are named *AX*, the left byte of AX is named *AH* for *A-high*, and the right byte of AX is named *AL* for *A-low*. The other accumulators are named accordingly. For example, the rightmost byte of the ECX register is named *CL*.

The x86 architecture has two index registers corresponding to the single X register of Pep/9. ESI is the source index register, and EDI is the destination index register. ESP is the stack pointer, which corresponds to the stack pointer SP of Pep/9. EBP is the base pointer, which points to the bottom of the current stack frame. Pep/9 has no corresponding register. EIP is called the *instruction pointer* in Intel terminology and corresponds to the program counter PC in Pep/9.

FIGURE 4.40 shows a machine language instruction that adds the content of the ECX register to the content of the EAX register and puts the sum in the ECX register. As with all von Neumann machines, a machine language instruction begins with an opcode field, 000000 in this example, which is the opcode for the add instruction. The following fields correspond to the register-r field and the addressing-aaa field of Pep/9 but with meaning specific to the x86 instruction set. The 1 in the s field indicates that the sum is on 32-bit quantities. If s were 0, only a single byte would be added. The 11 in the mod field indicates that the r/m field is a register. The 000 in the reg field, along with the 0 in the d field, indicates the EAX register, and the 001 in the r/m field, along with the 0 in the d field, indicates the ECX register. The hexadecimal abbreviation of the instruction is 01C1.

This is only one format from the x86 instruction set. There are multiple formats with some instruction specifiers preceded by special prefix bytes that change the format of the instruction specifier, and some followed by operand specifiers that might include a so-called scaled indexed byte. The instruction format scheme is complicated because it evolved from a small CPU with the requirement of backward compatibility. Pep/9 illustrates the concepts underlying machine languages in all von Neumann machines without the above complexities.

FIGURE 4.40
The x86 instruction format for the add register instruction.



4.4 Programming at Level ISA3

To program at Level ISA3 is to write a set of instructions in binary. To execute the binary sequence, first you must load it into main memory. The operating system is responsible for loading the binary sequence into main memory.

An operating system is a program. Like any other program, a software engineer must design, write, test, and debug it. Most operating systems are so large and complex that teams of engineers must write them. The primary function of an operating system is to control the execution of application programs on the computer. Because the operating system is itself a program, it must reside in main memory in order to be executed. So main memory must store not only the application programs, but also the operating system.

In the Pep/9 computer, the bottom part of main memory—that is, the part with high memory addresses—is reserved for the operating system. The top part is reserved for the application program. FIGURE 4.41 is a memory map of the Pep/9 computer system. It shows that the operating system starts at memory location FB8F and occupies the rest of main memory. That leaves memory locations 0000 to FB8E for the application program.

The loader is that part of the operating system that loads the application program into main memory so it can be executed. What loads the loader? The Pep/9 loader, along with several other parts of the operating system, is permanently stored in main memory.

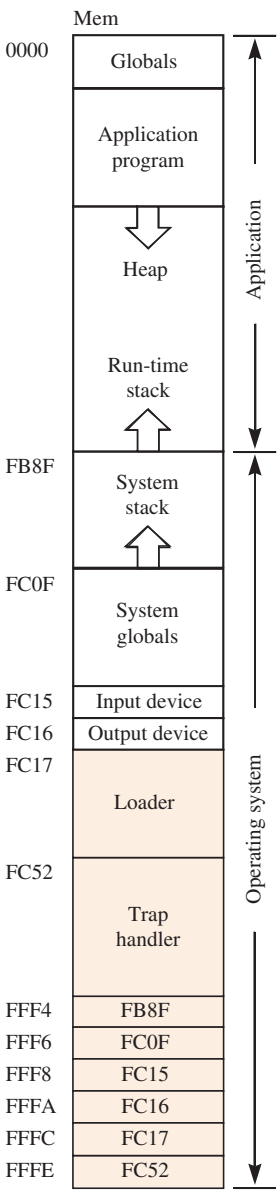
Read-Only Memory

There are two types of electronic-circuit elements from which memory devices are manufactured—read/write circuit elements and read-only circuit elements.

In the program of Figure 4.35, when the store byte instruction, F10013, executed, the CPU transferred the content of the right half of the accumulator to Mem[0013]. The original content of Mem[0013] was destroyed, and the memory location then contained 0111 0101 (bin), the binary code for the letter u. When the load byte instruction, D10013, executed, the bits at location 0013 were sent back to the accumulator so they could be sent to the output device.

The circuit element at memory location 0013 is a read/write circuit. The store byte instruction did a write operation on it, which changed its content. The read byte instruction did a read operation on it, which sent a copy of its content to the accumulator. If the circuit element at location 0013 were a read-only circuit, the store byte instruction would not have changed its content.

FIGURE 4.41
A memory map of the Pep/9 memory. The shaded part is read-only memory.



Both types of main-memory circuit elements—read/write and read-only—are random-access devices, as opposed to serial devices. When the load byte instruction does a read from memory location 0013, it does not need to start at location 0000 and sequentially go through 0001, 0002, 0003, and so on until it gets to 0013. Instead, it can go directly to location 0013. Because it can go to a random location in memory directly, the circuit element is called a *random-access* device.

RAM should be called RWM.

Read-only memory devices are known as *ROM*. Read/write memory devices should be known as *RWM*. Unfortunately, they are known as *RAM*, which stands for *random-access memory*. That name is unfortunate because both read-only and read/write devices are random-access devices. The characteristic that distinguishes a read-only memory device from a read/write memory device is that the content of a read-only device cannot be changed by a store instruction. Because use of the term *RAM* is so pervasive in the computer industry, we also will use it to refer to read/write devices. But in our hearts we will know that ROMs are random also.

Main memory usually contains some ROM devices. Those parts of main memory that are ROM contain permanent binary sequences, which the store instruction cannot alter. Furthermore, when power to the computer is switched off at the end of the day and then switched on at the beginning of the next day, the ROM will retain those binary sequences in its circuitry.

Volatile memory

RAM will not retain its memory if the power is switched off. It is therefore called *volatile*.

There are two ways a computer manufacturer can buy ROM for a memory system. In the first approach, the computer manufacturer specifies to the circuit manufacturer the bit sequences desired in the memory devices. The circuit manufacturer then manufactures the devices accordingly. Alternatively, the computer manufacturer orders a programmable read-only memory (PROM), which is a ROM with all zeros. The computer manufacturer then permanently changes any desired location to a 1, in such a way that the device will contain the proper bit sequence. This process is called "burning in" the bit pattern.

The Pep/9 Operating System

Most of the Pep/9 operating system has been burned into ROM. In Figure 4.41, the ROM part of the operating system is shaded. It begins at location FC17 and continues down to FFFF. That part of main memory is permanent. A store instruction cannot change it. If the power is ever turned off, when it is turned on again, that part of the operating system will still be there. The region from FB8F to FC16 is the RAM part of the operating system for our computer.

The RAM part of the operating system is for the system variables and memory-mapped I/O devices. Variables will change while the operating system program is executing. The ROM part of the operating system contains the loader, which is a permanent fixture. Its job is to load the application program into RAM, starting at address 0000. On the Pep/9 virtual machine, you invoke the loader by choosing the loader option from the menu of the simulator program.

The bottom of the run-time stack for the application program, called the *user stack*, is at memory location FB8F, just above the operating system. The stack pointer register in the CPU contains the address of the top of the stack. When procedures are called, storage for the parameters, the return address, and the local variables are allocated on the stack at successively lower addresses. Hence the stack "grows upward" in memory toward the smaller addresses.

The run-time stack for the operating system begins at memory location FC0F, which is 128 bytes below the start of the user stack. When the operating system executes, the stack pointer in the CPU contains the address of the top of the system stack. Like the user stack, the system stack grows upward in memory. The operating system never needs more than 128 bytes on its stack, so there is no possibility that the system stack will try to store its data in the user stack region.

The Pep/9 operating system consists of two programs—the loader, which begins at address FC17, and the trap handler, which begins at address FC52. You will recall from Figure 4.6 that the instructions with opcodes 0010 011 through 0010 1 are unimplemented at Level ISA3. The trap handler implements these instructions for the assembly language programmer. Chapter 5 describes these instructions at Level Asmb5, the assembly level, and Chapter 8 shows how they are implemented at Level OS4, the operating system level.

Associated with the parts of the operating system are six words at the very bottom of ROM that are reserved for special use by the operating system. They are called *machine vectors* and are at addresses FFF4, FFF6, FFF8, FFFA, FFFC, and FFFE, as shown in Figure 4.41.

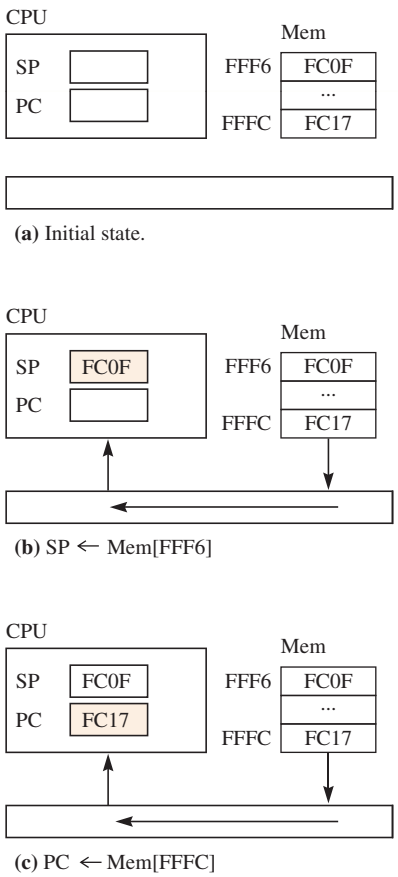
Machine vectors

When you choose the load option from the Pep/9 simulator menu, the following two events occur:

SP ← Mem[FFF6]
PC ← Mem[FFFC]

In other words, the content of memory location FFF6 is copied into the stack pointer, and the content of memory location FFFC is copied into the program counter. After these events occur, the execution cycle begins. FIGURE 4.42 illustrates these two events.

FIGURE 4.42
The Pep/9 load option.



Selecting the load option initializes the stack pointer and program counter to the predetermined values stored at FFF6 and FFFC. It just so happens that the value at address FFF6 is FC0F, the bottom of the system stack. FC0F is the value the stack pointer should have when the system stack is empty. It also happens that the value at address FFFC is FC17. In fact, FC17 is the address of the first instruction to be executed in the loader.

The system programmer who wrote the operating system decided where the system stack and the loader should be located. Realizing that the Pep/9 computer would fetch the vectors from locations FFFA and FFFC when the load option is selected, she placed the appropriate values in those locations. Because the first step in the execution cycle is fetch, the first instruction to be executed after selecting the load option is the first instruction of the loader program.

If you wish to revise the operating system, your loader might not begin at FC17. Suppose it begins at 7BD6 instead. When the user selects the load option, the computer will still go to location FFFC to fetch the vector. So you would need to place 7BD6 in the word at address FFFC.

The advantage of machine vectors

This scheme of storing addresses at special reserved memory locations is flexible. It allows the system programmer to place the loader anywhere in memory that is convenient. A more direct but less flexible scheme would be to design the system to execute the following operations when the user selects the load option:

SP ← FC0F
PC ← FC17

If selecting the load option produced these two events, the loader of the current operating system would still function correctly. However, it would be difficult to modify the operating system. The loader would always have to start at FC57, and the system stack would always have to start at FC4F. The system programmer would have no choice in the placement of the various parts of the system.

Using the Pep/9 System

To load a machine language program on the Pep/9 computer, fortunately you do not need to write it in binary. You may write it with ASCII hexadecimal characters in a text file. The loader will convert from ASCII to binary for you when it loads the program.

The listing in FIGURE 4.43 shows how to prepare a machine language program for loading. It is the program of Figure 4.33, which outputs `Hi`. You simply write in a text file the binary sequence in hexadecimal without any addresses or comments. Terminate the list of bytes with lowercase zz, which the loader detects as a sentinel. The loader will put the bytes in memory one after the other, starting at address 0000 (hex).

The Pep/9 loader is extremely particular about the format of your machine-language program. To work correctly, the very first character in your text file must be a hexadecimal character. No leading blank lines or spaces are allowed. There must be exactly one space between bytes. If you wish to continue your byte stream on another line, you must not leave trailing spaces on the preceding line.

After you write your machine-language program and load it with the loader option, you must select the execute option to run it. The following two events occur when you select the execute option:

SP ← Mem[FFF4]
PC ← 0000

FIGURE 4.43
Preparing a program for the loader.

```
Address     Machine Language (hex)
0000        D1000D ;Load byte accumulator 'H'
0003        F1FC16 ;Store byte accumulator output device
0006        D1000E ;Load byte accumulator 'i'
0009        F1FC16 ;Store byte accumulator output device
000C        00     ;Stop
000D        4869   ;ASCII "Hi" characters

Hex Version for the Loader
D1 00 0D F1 FC 16 D1 00 0E F1 FC 16 00 48 69 zz

Output
Hi
```

Then the von Neumann execution cycle begins. Because PC has the value 0000, the CPU will fetch the first instruction from Mem[0000]. Fortunately, that is where the loader put the first instruction of the application program.

Figure 4.41 shows that Mem[FFF4] contains FB8F, the address of the bottom of the user stack. The application program in this example does not use the run-time stack. If it did, the application program could access the stack correctly because SP would be initialized to the address of the bottom of the user stack.

Enjoy!

Chapter Summary

Virtually all commercial computers are based on the von Neumann design principle, in which main memory stores both data and instructions. The three components of a von Neumann machine are the central processing unit (CPU), main memory with memory-mapped I/O devices, and disk. The CPU contains a set of registers, one of which is the program counter (PC), which stores the address of the instruction to be executed next.

The CPU has an instruction set wired into it. An instruction consists of an instruction specifier and an operand specifier. The instruction specifier,

in turn, consists of an opcode and possibly a register field and an addressing mode field. The opcode determines which instruction in the instruction set is to be executed. The register field determines which register participates in the operation. The addressing mode field determines which addressing mode is used for the source or destination of the data.

Each addressing mode corresponds to a relationship between the operand specifier (OprndSpec) and the operand (Oprnd). In the direct addressing mode, the operand specifier is the address in main memory of the operand. In mathematical notation, Oprnd = Mem[OprndSpec].

To execute a program, a group of instructions and data are loaded into main memory, and then the von Neumann execution cycle begins. The von Neumann execution cycle consists of the following steps: (1) fetch the instruction specified by PC, (2) decode the instruction specifier, (3) increment PC, (4) execute the instruction fetched, and (5) repeat by going to Step 1.

Because main memory stores instructions as well as data, two types of errors at the machine level are possible. You may interpret data bits as instructions, or you may interpret instruction bits as data. Another possibility that is a direct result of storing instructions in main memory is that a program may be processed as if it were data. Loaders and compilers are important programs that take the viewpoint of treating instructions as data.

The operating system is a program that controls the execution of applications. It must reside in main memory along with the applications and data. On some computers, a portion of the operating system is burned into read-only memory (ROM). One characteristic of ROM is that a store instruction cannot change the content of a memory cell. The run-time stack for the operating system is located in random-access memory (RAM). A machine vector is an address of an operating system component, such as a stack or a program, used to access that component. Two important functions of an operating system are the loader and the trap handler.

Exercises

Section 4.1

*1. **(a)** How many bytes are in the main memory of the Pep/9 computer? **(b)** How many words are in it? **(c)** How many bits are in it? **(d)** How many total bits are in the Pep/9 CPU? **(e)** How many times bigger in terms of bits is the main memory than the CPU?

2. **(a)** Suppose the main memory of the Pep/9 were completely filled with unary instructions. How many instructions would it contain? **(b)** What is the maximum number of instructions that would fit in the main memory if none of the instructions is unary? **(c)** Suppose the main memory is completely filled with an equal number of unary and nonunary instructions. How many total instructions would it contain?

*__3.__ Answer the following questions for the machine language instructions 6AF82C and D623D0. **(a)** What is the opcode in binary? **(b)** What does the instruction do? **(c)** What is the register-r field in binary? **(d)** Which register does it specify? **(e)** What is the addressing-aaa field in binary? **(f)** Which addressing mode does it specify? **(g)** What is the operand specifier in hexadecimal?

4. Answer the questions in Exercise 3 for the machine language instructions 7B00AC and F70BD3.

Section 4.2

*__5.__ Suppose Pep/9 contains the following four hexadecimal values:

A: 19AC
X: FE20
Mem[0A3F]: FF00
Mem[0A41]: 103D

If it has these values before each of the following statements executes, what are the four hexadecimal values after each statement executes?

(a) C10A3F **(b)** D10A3F **(c)** D90A41
(d) F10A41 **(e)** E90A3F **(f)** 790A41
(g) 710A3F **(h)** 910A3F **(i)** 07

6. Repeat Exercise 5 for the following statements:

(a) C90A3F **(b)** D90A3F **(c)** F10A41
(d) E10A41 **(e)** 690A3F **(f)** 710A41
(g) 890A3F **(h)** 990A3F **(i)** 06

Section 4.3

7. Determine the output of the following Pep/9 machine language program. The left column is the memory address of the first byte on the line:

```
0000 D10013
0003 F1FC16
0006 D10014
```

```
0009 F1FC16
000C D10015
000F F1FC16
0012 00
0013 4A6F
0015 79
```

8. Determine the output of the following Pep/9 machine language program if the input is `tab`. The left column is the memory address of the first byte on the line:

```
0000 D1FC15
0003 F1001F
0006 D1FC15
0009 F10020
000C D1FC15
000F F10021
0012 D10020
0015 F1FC16
0018 D1001F
001B F1FC16
001E 00
```

9. Determine the output of the following Pep/9 machine language program. The left column in each part is the memory address of the first byte on the line:

*(a)

```
0000 C1000A
0003 81000C
0006 F1FC16
0009 00
000A A94F
000C FFFD
```

(b)

```
0000 C10008
0003 06
0004 F1FC16
0007 00
0008 F0D4
```

Section 4.4

10. Suppose you need to process a list of 31,000 integers contained in Pep/9 memory at one integer per word. You estimate that 20% of the instructions in a typical program are unary instructions. What is the maximum number of instructions you can expect to be able to use in the program that processes the data? Keep in mind that your applications program must share memory with the operating system and with your data.

Problems

Section 4.4

11. Write a machine language program to output your name on the output device. The name you output must be longer than two characters. Write it in a format suitable for the loader and execute it on the Pep/9 simulator.

12. Write a machine language program to output the four characters `Frog` on the output device. Write it in a format suitable for the loader and execute it on the Pep/9 simulator.

13. Write a machine language program to output the three characters `Cat` on the output device. Write it in a format suitable for the loader and execute it on the Pep/9 simulator.

14. Write a machine language program to add the three numbers 2, –3, and 6 and output the sum on the output device. Store the –3 in hexadecimal. Do not use the subtract, negate, or invert instructions. Write the program in a format suitable for the loader and execute it on the Pep/9 simulator.

15. Write a machine language program to input two one-digit numbers, add them, and output the one-digit sum. There can be no space between the two one-digit numbers on input. Write the program in a format suitable for the loader and execute it on the Pep/9 simulator.

16. Write the program in Figure 4.35 in hexadecimal format for input to the loader. Verify that it works correctly by running it on the Pep/9 simulator with an input of `up`. Then modify the store byte instruction at 0003 so that the first character is stored at Mem[FCAA] and the load byte instruction at 000C is also from Mem[FCAA]. What is the output? Explain.