Chapter 4

Computer Architecture

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.

The Pep/10 computer

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/10 computer*.

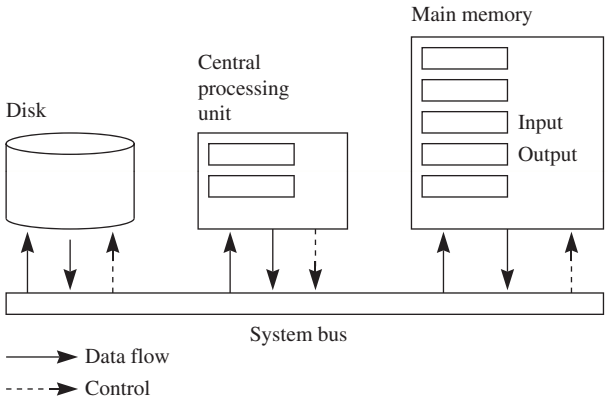


Figure 4.1 Block diagram of the Pep/10 computer.

4.1 Pep/10 Hardware

The Pep/10 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 components of a computer system

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.

Central Processing Unit

The CPU contains six specialized memory locations called *registers*. As Figure 4.2 shows, 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)

Registers in the Pep/10 CPU

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

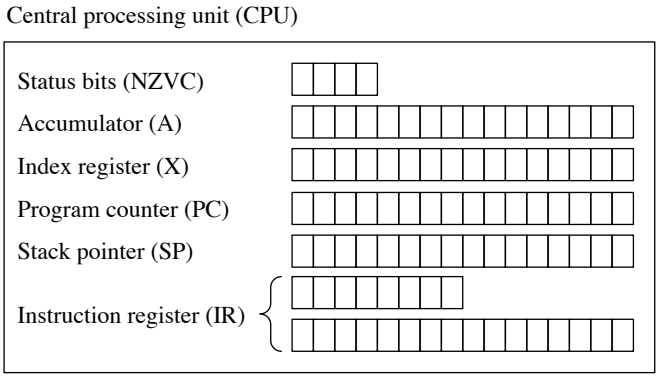


Figure 4.2 The CPU of the Pep/10 computer.

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 seven registers, the CPU contains all the electronics (not shown in Figure 4.2) to execute the Pep/10 instructions.

Main Memory

Eight bits in a byte

Figure 4.3 shows the main memory of the Pep/10 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.

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.

The address of a multi-byte cell in main memory

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.

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

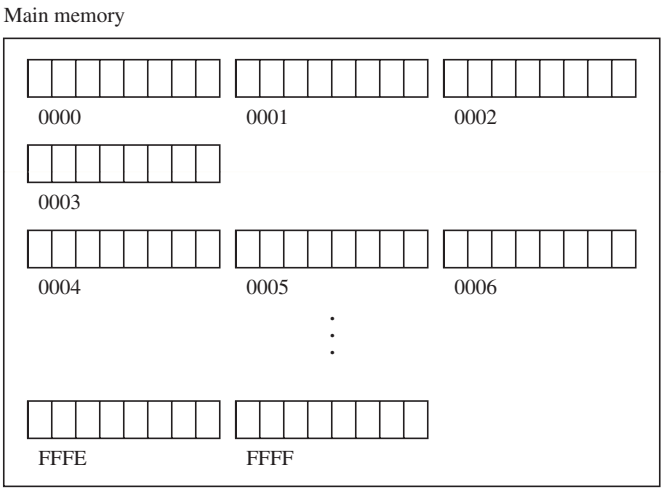


Figure 4.3 The main memory of the Pep/10 computer.

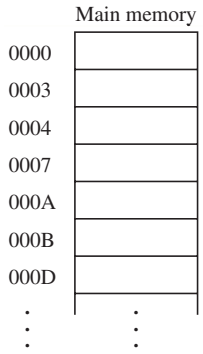


Figure 4.4 Another style for depicting main memory.

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.

How to determine the number of bytes in a memory cell

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.

Most computer manufacturers specify a word to be a certain number of bytes. In the Pep/10 computer, a *word* is two adjacent bytes. A word, therefore, contains 16 bits. Most of the registers in the Pep/10 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.

Definition of a word

Address of a word

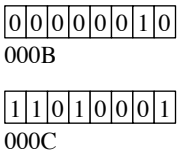
It is important to distinguish between the content of a memory location and its address. Memory addresses in the Pep/10 computer are 16 bits long. Hence, the memory address of the word in Figure 4.5(a) could be written in binary as 0000 0000 0000 1011. The content of the word at this address, however, is 0000 0010 1101 0001. Do not confuse the content of the word with its address. They are different.

Address versus content of a memory cell

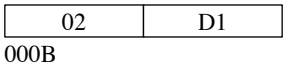
To save space on the page, the content of a byte or word is usually written in hexadecimal. Figure 4.5(b) shows the content in hexadecimal of the same word at address 000B. In a machine language listing, the address of the first byte of a group is printed, followed by the content in hexadecimal, as in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



(a) The content in binary.



(b) The content in hexadecimal.

000B 02D1

(c) The machine language listing.

Figure 4.5 The content of a cell in main memory.

Input/Output Devices

You may be wondering where this Pep/10 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/10 virtual machine described in these chapters.

The Pep/10 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.

Pep/10 simulates a 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/10, the input device is at address FFFD and the output device is at address FFFE.

The Power Off Port

Most computer systems are designed to run continuously for long periods of time. For example, most people rarely turn off their smart phones, which run throughout the day and night. However, all computer systems need a mechanism to power off. Pep/10 has a memory-mapped power off port at memory address FFFF. To shut down the computer system, you can send a byte of information from the CPU to the power off port. It does not matter what information you send to the port. Any information that is sent to the power off port will shut down the Pep/10 virtual computer.

Figure 4.6 is a *memory map* that shows the input and output ports and the power off port for the Pep/10 computer system operating in the bare metal mode. A memory map shows where the parts of the computer system are located in the main memory of the computer. This figure shows that an application program begins at memory address 0000, the data for the application

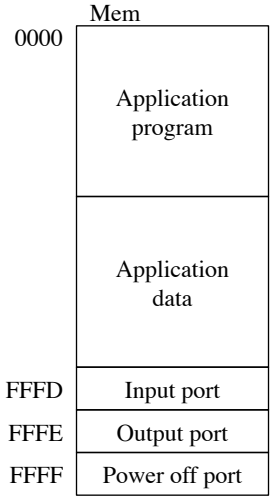


Figure 4.6 The Pep/10 memory map in bare metal mode.

is placed after the program, the input port is at address FFFD, the output port is at address FFFE, and the power off port is at address FFFF.

Recall that we are learning about computer systems at level three, the ISA3 level. At this level, we do not have access to the operating system, which is at level four. When you program in bare metal mode, the last statement in your program should be a statement that sends a byte to the power off port, which will shut down the Pep/10 virtual machine.

How to shut down the Pep/10 virtual machine

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 FFFD on the bus to the CPU. It can also flow from the CPU on the bus to the output device at address FFFE. 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/10 simulator does not provide this feature.

Data lines in the CPU

Direct memory access

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.

Control lines in the CPU

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 set of a CPU

Figure 4.7 shows the 37 instructions in the Pep/10 instruction set. Each instruction consists of either a single byte called the *instruction specifier*, or the instruction specifier followed immediately by a two-byte word called the *operand specifier*. Instructions that have only an instruction specifier are called *monadic instructions*. Instructions that have both an instruction specifier and an operand specifier are called *dyadic instructions*. Figure 4.8 shows the structure of monadic and dyadic instructions.

Instruction and operand specifiers

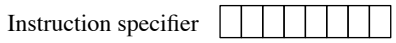
Monadic and dyadic instructions

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 4.7 shows the instruction to move the stack pointer to the accumulator as having

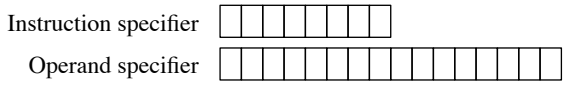
Opcodes

Instruction Specifier	Instruction	Type
0000 0000	Illegal instruction	
0000 0001	Return from call	Monadic
0000 0010	Return from system call	Monadic
0000 0011	Move SP to A	Monadic
0000 0100	Move A to SP	Monadic
0000 0101	Move NZVC flags to A[12 : 15]	Monadic
0000 0110	Move A[12 : 15] to NZVC flags	Monadic
0000 0111	No operation	Monadic
0001 100r	Negate r	Monadic
0001 101r	Arithmetic shift left r	Monadic
0001 110r	Arithmetic shift right r	Monadic
0001 111r	Bitwise Not r	Monadic
0010 000r	Rotate left r	Monadic
0010 001r	Rotate right r	Monadic
0010 010a	Branch unconditional	Dyadic
0010 011a	Branch if less than or equal to	Dyadic
0010 100a	Branch if less than	Dyadic
0010 101a	Branch if equal to	Dyadic
0010 110a	Branch if not equal to	Dyadic
0010 111a	Branch if greater than or equal to	Dyadic
0011 000a	Branch if greater than	Dyadic
0011 001a	Branch if V	Dyadic
0011 010a	Branch if C	Dyadic
0011 011a	Call subroutine	Dyadic
0011 1aaa	System call	Dyadic
0100 0aaa	Add to SP	Dyadic
0100 1aaa	Subtract from SP	Dyadic
0101 raaa	Add to r	Dyadic
0110 raaa	Subtract from r	Dyadic
0111 raaa	Bitwise And to r	Dyadic
1000 raaa	Bitwise Or to r	Dyadic
1001 raaa	Bitwise Exclusive Or to r	Dyadic
1010 raaa	Compare word to r	Dyadic
1011 raaa	Compare byte to r[8 : 15]	Dyadic
1100 raaa	Load word r from memory	Dyadic
1101 raaa	Load byte r[8 : 15] from memory	Dyadic
1110 raaa	Store word r to memory	Dyadic
1111 raaa	Store byte r[8 : 15] to memory	Dyadic

Figure 4.7 The Pep/10 instruction set at Level ISA3.



(a) A monadic instruction.



(b) The two parts of a dyadic instruction.

Figure 4.8 The Pep/10 instruction format.

an eight-bit opcode of 0000 0011. The Add to SP instruction, however, has the five-bit opcode 01000. Instructions with fewer than eight bits in the opcode subdivide their instruction specifier into several fields depending on the instruction. Figure 4.7 indicates these fields with the letters a and r. Each of these letters can be either 0 or 1.

Instruction specifier fields

Example 4.1 Figure 4.7 shows that the Branch if less than instruction has an instruction specifier of 0010 100a. Because the letter a can be zero or one, there are really two versions of the instruction—0010 1000 and 0010 1001. Similarly, there are eight versions of the Add to SP instruction. Its instruction specifier is 0100 0aaa, where aaa can be any combination from 000 to 111. ■

Figure 4.9(a) shows the code for the addressing-aaa field. Pep/10 executes each dyadic instruction in one of eight addressing modes—immediate, direct, indirect, stack-relative, stack-relative 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 Figure 4.9 to determine which addressing mode a given instruction uses.

Addressing-aaa field

Example 4.2 Determine the opcode and addressing mode of the 0100 1000 instruction. Starting from the left, determine with the help of Figure 4.7 that the opcode is 01001, which specifies the Subtract from SP instruction. The next three bits after the opcode are the aaa bits, which are 000, indicating immediate addressing from Figure 4.9(a). Therefore, the instruction subtracts a value from the stack pointer using immediate addressing. ■

Example 4.3 Determine the opcode and addressing mode of the 0010 1001 instruction. Starting from the left, determine with the help of Figure 4.7 that the opcode is 0010100, which specifies the Branch if less than instruction. The next bit after the opcode is the a bit, which is 1, indicating indexed addressing from Figure 4.9(b). Therefore, the instruction branches on the less than condition using indexed addressing. ■

Addressing-a field

Figure 4.9(c) shows the code for the register-r field. When r is 0, the instruction operates on the accumulator. When r is 1, the instruction operates on

Register-r field

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.

Figure 4.9 The Pep/10 instruction specifier fields.

the index register.

Example 4.4 Determine the opcode, register, and addressing mode of the 1100 1011 instruction. Starting from the left, determine with the help of Figure 4.7 that the opcode is 1100, which specifies the Load word r from memory instruction. 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

The operand specifier, for those instructions that are dyadic, 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.

The address of an instruction

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 4.5 Figure 4.10 shows two adjacent instructions stored in main memory at locations 01A3 and 01A6. The instruction at 01A3 is dyadic; the instruction at 01A6 is monadic. In this example, the instruction at 01A3 has

Opcode: 0110
Register-r field: 1
Addressing-aaa field: 101
Operand specifier: 0000 0011 0100 1110

where all the quantities are written in binary. According to the opcode chart of Figure 4.7, this is the Subtract from r instruction. The register-r field indicates that the index register, as opposed to the accumulator, is affected. So,

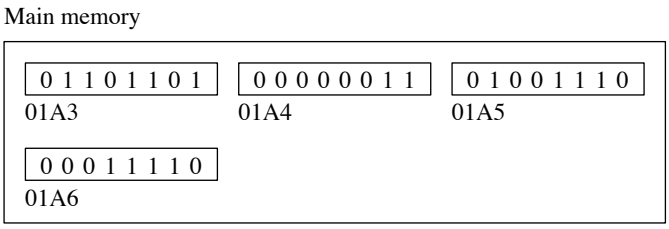


Figure 4.10 Two instructions in main memory.

this instruction subtracts the operand from the index register. The addressing-aaa field indicates indexed addressing, so the operand specifier is interpreted accordingly. This chapter confines our study to the direct addressing mode. Later chapters take up the other modes.

The monadic instruction at 01A6 has

Opcode: 0001 111
Register-r field: 0

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

In Example 4.5, the following form of the instructions is called *machine language*:

Machine language

0110 1101 0000 0011 0100 1110
0001 1110

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:

A machine language listing

01A3 6D034E
01A6 1E

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/10 instructions at Level ISA3. It shows 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,

Oprnd = Mem[OprndSpec]

Direct addressing

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. In hardware, main memory is literally an array of bytes, where the address is the index of the array.

Main memory is an array of bytes.

Following is a description of some instructions from the Pep/10 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 Load Word Instruction

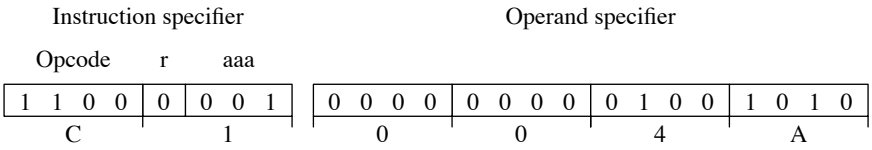
The Load word instruction has instruction specifier 1100 raaa. This instruction loads one word (two bytes) from a memory location into either the accumulator or the index register, depending on the value of r. It affects the N and Z bits. If the operand is negative, it sets the N bit to 1; otherwise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

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

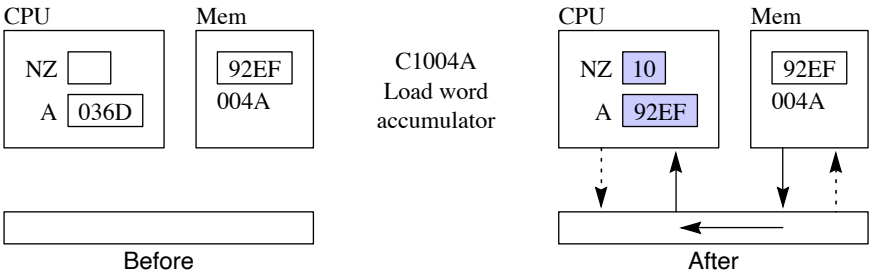
Load word RTL

Example 4.6 Suppose the instruction to be executed is C1004A in hexadecimal, which Figure 4.11(a) shows in binary. The register-r field in this example is 0, which indicates load word to the accumulator instead of the index register. The addressing-aaa field is 001, which indicates direct addressing.

Figure 4.11(b) 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



(a) Instruction format.



(b) Execution

Figure 4.11 The Load word accumulator instruction.

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.

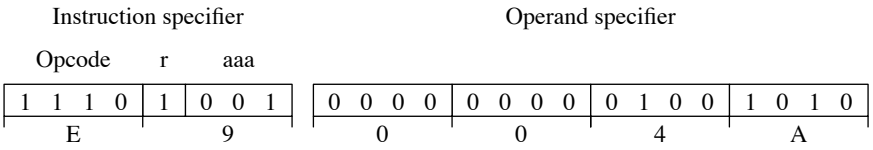
Figure 4.11(b) shows the data flow lines and control lines that the Load word instruction activates. As indicated by the solid lines, data flows from the main memory on the bus to the CPU, and then into the register. For this data transfer to take place, the CPU must send a control signal (as indicated 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. ■

The Store Word Instruction

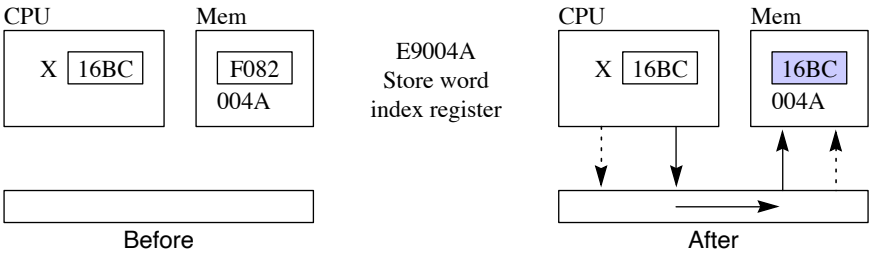
The Store word instruction has instruction specifier 1110 raaa. This instruction stores one word (two bytes) from either the accumulator or the index register to a memory location. With direct addressing, the operand specifies the memory location in which the information is stored. The RTL specification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

Oprnd ← r *Store word RTL*

Example 4.7 Suppose the instruction to be executed is 69004A in hexadecimal, which Figure 4.12(a) 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.



(a) Instruction format.



(b) Execution

Figure 4.12 The Store word index register instruction.

Figure 4.12(b) shows the effect of executing the Store word instruction, assuming the index register has an initial content of 16BC. The Store word 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 word instruction affects none of the status bits. ■

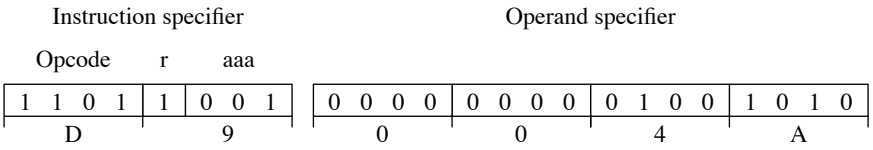
The Load Byte Instruction

This instruction, along with the one that follows, is a byte instruction. Byte instructions operate on a single byte of information instead of a word. The Load byte instruction has instruction specifier 1101 raaa. It loads the operand into the right half of either the accumulator or the index register, and affects the N and Z bits. It clears the left half of the register to all zeros. The RTL specification of the Load byte instruction is

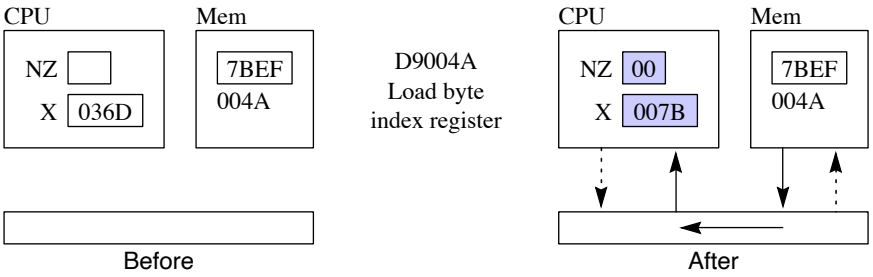
$r[0:7] \leftarrow 0\ ,\ r[8:15] \leftarrow \text{byte Oprnd}\ ;\ N \leftarrow 0\ ,\ Z \leftarrow r[8:15] = 0$ *Load byte RTL*

Example 4.8 Suppose the instruction to be executed is D9004A in hexadecimal, which Figure 4.13(a) shows in binary. The register-r field in this example is 1, which indicates load to the index register instead of the accumulator. The addressing-aaa field is 001, which indicates direct addressing.

Figure 4.13(b) shows the effect of executing the Load byte index register instruction, assuming the initial value of the index register is 036D. The byte at



(a) Instruction format.



(b) Execution

Figure 4.13 The Load byte index register instruction.

Mem[004A] has an initial content of 7B. Execution of the instruction replaces the right half of the index register X[8 : 15] with 7B. The left half of the index register X[0 : 7], which initially contained 03, is cleared to zero. Notice that the address of the EF byte in main memory—next to the 7B byte at address 004A—is at address 004B and has nothing to do with the execution of this Load byte instruction.

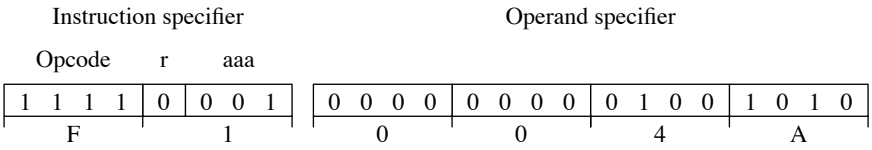
The N bit is always set to 0 with this instruction. The Z bit is set to 0 in this example because the eight bits loaded into the right half of the index register, 7B(hex) or 0111 1011(bin), are not all 0's. ■

The Store Byte Instruction

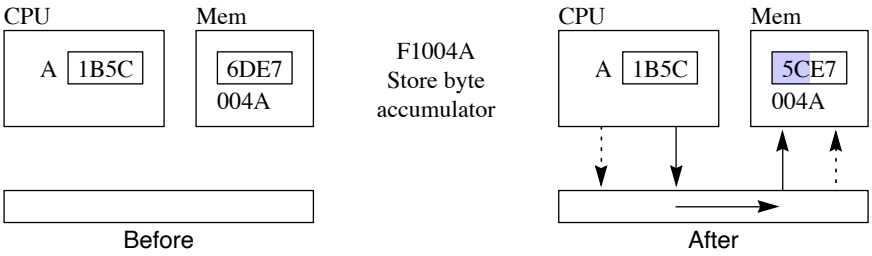
The Store byte instruction has instruction specifier 1111 raaa. It stores the right half of either the accumulator or the index register into a one-byte memory location and does not affect any status bits. The RTL specification of the store byte instruction is

byte Oprnd ← r[8 : 15] *Store byte RTL*

Example 4.9 Suppose the instruction to be executed is F1004A in hexadecimal, which Figure 4.14(a) shows in binary. The register-r field in this example is 0, which indicates store from the accumulator instead of the index register. The addressing-aaa field is 001, which indicates direct addressing.



(a) Instruction format.



(b) Execution

Figure 4.14 The Store byte accumulator instruction.

Figure 4.14(b) shows the effect of executing the Store byte instruction, assuming the byte at Mem[004A] has an initial content of 6D. Execution of the instruction replaces the 6D with 5C, which is the right half of the accumulator A[8 : 15]. Notice that the address of the E7 byte in main memory—next to the initial 6D byte at address 004A—is at address 004B and has nothing to do with the execution of this Store byte instruction. ■

Load is memory to CPU.

Store is CPU to memory.

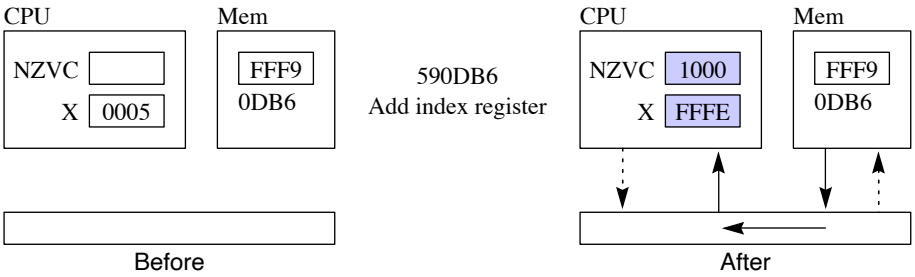
All commercial CPUs have load instructions and store instructions. Like the Pep/10 CPU, load instructions transfer one or more data bytes from main memory over the system bus to a register in the CPU. Store instructions transfer one or more data bytes from a register in the CPU over the system bus to main memory. The solid arrows in part (b) of the preceding four figures show the dataflow between the CPU and main memory.

The Arithmetic Instructions

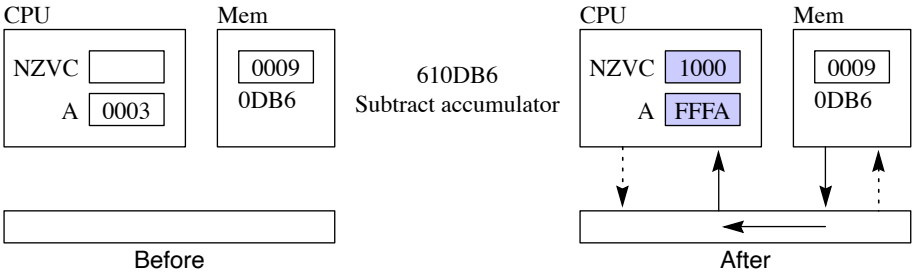
Pep/10 has two dyadic arithmetic instructions—add to r and subtract from r—and three monadic arithmetic instructions—negate r, arithmetic shift left r, and arithmetic shift right r.

The dyadic arithmetic instructions

Both dyadic arithmetic instructions are 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



(a) The Add index register instruction.



(b) The Subtract accumulator instruction.

Figure 4.15 The dyadic arithmetic instructions.

in the register, and all four status bits are set accordingly.

The Subtract from r instruction is identical, except that the content of the word from main memory is subtracted from the content of the register. 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 specifications of the Add and Subtract instructions are

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

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

Example 4.10 Suppose the instruction to be executed is 590DB6 in hexadecimal. You should decode the instruction specifier 59 and determine that the opcode is 0101 indicating the Add instruction, the register-r field is 1 indicating the index register, and the addressing-aaa field is 001 indicating direct addressing.

Figure 4.15(a) shows the effect of executing the Add instruction, assuming the index register has an initial content of 0005 and Mem[0DB6] has -7(dec) $=$ FFF9(hex). In decimal, the sum $5+(-7)$ is -2, which the figure shows as FFFE. 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. ■

Example 4.11 Suppose the instruction to be executed is 610DB6. You should decode the instruction specifier 61 and determine that the opcode is 0110 indicating the Subtract instruction, the register-r field is 0 indicating the accumulator, and the addressing-aaa field is 001 indicating direct addressing.

Figure 4.15(b) shows the effect of executing the Subtract instruction, assuming the accumulator has an initial content of 0003 and Mem[0DB6] has 0009. In decimal, the difference $3-9$ is -6, which the figure shows as FFFA. In practice, the CPU computes the difference $3-9$ by negating the 9 and using the adder hardware to compute $3+(-9)$. 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 a signed overflow did not occur, and the C bit is 0 because a carry did not occur when -9 was added to 3. ■

The monadic arithmetic instructions

The monadic arithmetic instructions are negate, arithmetic shift left, and arithmetic shift right. They do not have operand specifiers and do not access main memory. They operate on a register in the CPU, altering its content. Here are the RTL specifications of the monadic arithmetic instructions.

Negate RTL

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

Arithmetic shift left RTL

$$C \leftarrow r[0]\ ,\ r[0:14] \leftarrow r[1:15]\ ,\ r[15] \leftarrow 0\ ;\ N \leftarrow r < 0\ ,\ Z \leftarrow r = 0\ ,\ V \leftarrow \{\textit{overflow}\}$$

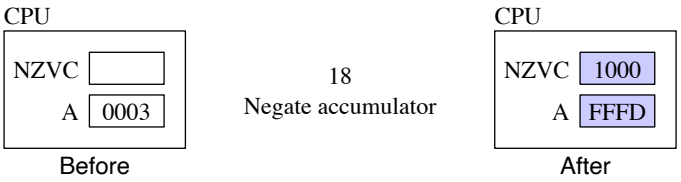
Arithmetic shift right RTL

$$C \leftarrow r[15]\ ,\ r[1:15] \leftarrow r[0:14]\ ;\ N \leftarrow r < 0\ ,\ Z \leftarrow r = 0\ ,\ V \leftarrow 0$$

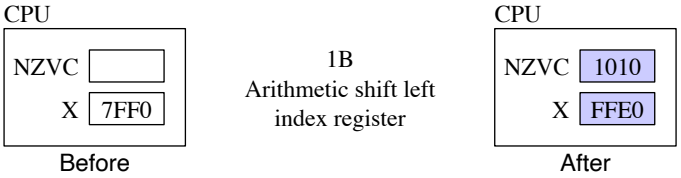
Example 4.12 Figure 4.16(a) shows the operation of the Negate instruction. The instruction specifier is 18 (hex) or 0001 1000 (bin), with an opcode of 0001 100 signifying the Negate instruction, and a register-r field of 0 signifying the accumulator. The initial value in the accumulator is 3. The negation of 3 is -3, which is 1111 1111 1111 1101 (bin) = FFFD (hex). The N bit is 1 because the final result is negative. ■

Example 4.13 Figure 4.16(b) shows the operation of the Arithmetic shift left index register instruction. The instruction specifier is 1B (hex) or 0001 1011 (bin), with an opcode of 0001 101, and a register-r field of 1. The initial value in the index register is 7FF0 (hex) = 0111 1111 1111 0000 (bin). The left shift is FFE0. The V bit is 1 because there was a signed integer overflow. ■

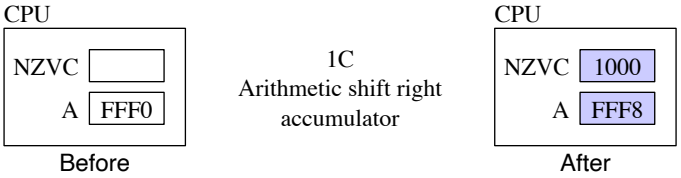
Example 4.14 Figure 4.16(c) shows the operation of the Arithmetic shift right instruction. The instruction specifier is 1C (hex) or 0001 1100 (bin), with an opcode of 0001 110 and a register-r field of 0. The initial value in the accumulator is FFF0 (hex) = 1111 1111 1111 0000 (bin). The right shift is FFF8. The V bit is always 0 because there can never be an overflow with a right shift operation. ■



(a) The Negate accumulator instruction.



(b) The Arithmetic shift left index register instruction.



(c) The Arithmetic shift right index register instruction.

Figure 4.16 The monadic arithmetic instructions.

The Logic Instructions

Pep/10 has six logic instructions—bitwise And to r, bitwise Or to r, bitwise Exclusive Or to r, bitwise Not r, rotate left r, and rotate right r. The first three are dyadic, and the last three are monadic.

The dyadic logic 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 instruction is useful for masking out undesired 1 bits from a bit pattern. The Or instruction is useful for inserting 1 bits into a bit pattern. The Exclusive Or instruction is common in cryptographic algorithms. All three instructions affect the N and Z bits and leave the V and C bits unchanged. The RTL specifications for the dyadic logic instructions are

$r \leftarrow r \wedge Oprnd\ ;\ N \leftarrow r < 0\ ,\ Z \leftarrow r = 0$ *And RTL*

$r \leftarrow r \vee Oprnd\ ;\ N \leftarrow r < 0\ ,\ Z \leftarrow r = 0$ *Or RTL*

$r \leftarrow r \oplus Oprnd\ ;\ N \leftarrow r < 0\ ,\ Z \leftarrow r = 0$ *Exclusive Or RTL*

Example 4.15 Suppose the instruction to be executed is 710DB2 as in the

Before	Instruction	After
A: 9DC3 Mem[0DB2]: FF00	710DB2 Bitwise And accumulator	A: 9D00 NZ: 10 Mem[0DB2]: FF00
X: A56B Mem[0DB2]: 00FF	890DB2 Bitwise Or index register	X: A5FF NZ: 10 Mem[0DB2]: 00FF
A: 83C9 Mem[0DB2]: F00F	910DB2 Bitwise Exclusive Or accumulator	A: 73C6 NZ: 00 Mem[0DB2]: F00F

Figure 4.17 The dyadic logic instructions.

first row of the table in Figure 4.17. The instruction specifier is 71 (hex) = 0111 0001 (bin) with an opcode of 0111 indicating the bitwise And instruction, a register-r field of 0 indicating the accumulator, and an addressing field of 001 indicating direct addressing.

The first row of the table shows the effect of executing the bitwise And accumulator instruction, assuming the accumulator has an initial content of 9DC3 and Mem[0DB2] has FF00 (hex) = 1111 1111 0000 0000 (bin). At every position where there is a 1 in Mem[0DB2], the corresponding bit in the accumulator is unchanged. At every position where there is a 0, the corresponding bit is cleared to 0. The value FF00 is called an *AND mask*, and the operation is *masking out* the low-order half (that is, the the rightmost byte) of the accumulator. 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. ■

The operation of an AND mask

Example 4.16 The second row of Figure 4.17 shows the operation of the bitwise Or index register instruction with an instruction specifier of 89 (hex) = 1000 1001 (bin). In this example, the *OR mask* at Mem[0DB2] is 00FF (hex) = 0000 0000 1111 1111 (bin), and the initial value of the index register is A56B. When used with the bitwise Or index register instruction, everywhere in the mask there is a 1, the corresponding bit in the index register is 1. Everywhere in the mask there is a 0, the corresponding bit in the index register is unchanged. The final value of the index register is A5FF, which is the high-order half (that is, the leftmost byte) unchanged, and the low order half all 1's. ■

The operation of an OR mask

Example 4.17 The third row of Figure 4.17 shows the operation of the bitwise Exclusive Or accumulator instruction with an instruction specifier of 91 (hex) = 1001 0001 (bin). In this example, the *XOR mask* at Mem[0DB2] is F00F (hex) = 1111 0000 0000 1111 (bin), and the initial value of the accumulator is 83C9. When used with the Exclusive Or accumulator instruction, everywhere in the mask there is a 1, the corresponding bit in the accumulator is inverted. Everywhere in the mask there is a 0, the corresponding bit in the

The operation of an XOR mask

Before	Instruction	After
X: 103F	1F Bitwise Not index register	X: EFC0 NZ: 10
A: 7F69 C:1	20 Rotate left accumulator	A: FED3 NZC: 100
A: 7F69 C:1	22 Rotate right accumulator	A: BFB4 NZC: 001

Figure 4.18 The monadic logic instructions.

accumulator is unchanged. The final value of the accumulator is 73C6. The leftmost four bits are inverted from 0111 to 1000. The middle eight bits are unchanged. The rightmost four bits are inverted from 1001 to 0110. Because of this property, the XOR mask is known as a *selective inverter*. The N bit is 0 because the quantity in the accumulator is not negative when interpreted as a signed integer. ■

The selective inverter property of the XOR mask

The monadic logic instructions are bitwise Not, rotate left, and rotate right. They do not have operand specifiers and do not access main memory. Like the monadic arithmetic instructions, they operate on a register in the CPU, altering its content. Here are the RTL specifications of the monadic logic instructions.

The monadic logic instructions

$r \leftarrow \neg r$; $N \leftarrow r < 0$, $Z \leftarrow r = 0$ *Bitwise Not RTL*

$C \leftarrow r[0]$, $r[0:14] \leftarrow r[1:15]$, $r[15] \leftarrow C$; $N \leftarrow r < 0$, $Z \leftarrow r = 0$ *Rotate left RTL*

$C \leftarrow r[15]$, $r[1:15] \leftarrow r[0:14]$, $r[0] \leftarrow C$; $N \leftarrow r < 0$, $Z \leftarrow r = 0$ *Rotate right RTL*

All three monadic logic instructions set the N and Z bits. The rotate instructions also set the C bit.

Example 4.18 The first row of the table in Figure 4.18 shows the operation of the bitwise Not instruction. The instruction specifier is 1F (hex) = 0001 1111 (bin), with an opcode of 0001 111 signifying the bitwise Not instruction, and a register-r field of 1 signifying the index register. The initial value in the index register is 103F (hex) = 0001 0000 0011 1111 (bin). The bitwise Not of that value is 1110 1111 1100 0000 (bin) = EFC0 (hex). ■

Example 4.19 The instruction specifier in the second row of the table is 20 (hex) = 0010 0000 (bin), with an opcode of 0010 000 specifying the rotate left instruction, and a register-r field of 0 specifying the accumulator. The initial value in the accumulator is 7F69 (hex) = 0111 1111 0110 1001 (bin), and the initial value of the carry bit is 1. On execution, the carry bit gets the 0 from A[0], and A[15] gets the 1 from the carry bit. The final value in the accumulator is 1111 1110 1101 0011 (bin) = FED3 (hex).

Before	Instruction	After
A: FFFF Input: `Hello World!`	D1FFFD Load byte accumulator	A: 0048 NZ: 00 Input: `ello World!`
A: 0021 Output: `Hello World`	F1FFFE Store byte accumulator	A: 0021 Output: `Hello World!`
A: 0000	F1FFFF Store byte accumulator	Shut down

Figure 4.19 Accessing the memory-mapped ports.

The third row of the table shows the execution of the rotate right instruction with the same initial state. On execution, the carry bit gets the 1 from A[15], and A[0] gets the 1 from the carry bit. The final value in the accumulator is 1011 1111 1011 0100 (bin) = BFB4 (hex). ■

Accessing the Memory-Mapped Ports

Figure 4.6 (page 60) shows three ports wired into main memory at the last three addresses.

Mem[FFFD]: The input port
Mem[FFFE]: The output port
Mem[FFFF]: The power off port

The input port 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 FFFD. The output port 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 FFFE. The power off port is attached to a circuit that initiates the shut down procedure for the computer. You shut down the computer by executing the store byte instruction to address FFFF.

Input port is ASCII.

Output port is ASCII.

Power off port is shut down.

Example 4.20 The first row of the table in Figure 4.19 shows how to access the next ASCII character from the input stream with the load byte accumulator instruction. The input stream is the string of characters `Hello World!` The first letter in the stream is `H`, and its ASCII value is 48 (hex). Execution of load byte from the input port Mem[FFFD] consumes the `H` from the input stream and loads its ASCII value into the right half of the accumulator. The next time the same instruction executes, it will consume the `e` and will load its ASCII value 65 (hex) into the accumulator. ■

Example 4.21 The second row shows how to send an ASCII character to the output stream with the store byte accumulator instruction. This scenario


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

Address     Machine Language (hex)
0000        D1000F ;Load byte accumulator 'H'
0003        F1FFFE ;Store byte accumulator output port
0006        D10010 ;Load byte accumulator 'i'
0009        F1FFFE ;Store byte accumulator output port
000C        F1FFFF ;Store byte accumulator power off port
000F        4869   ;ASCII "Hi" characters

Output
Hi
```

Figure 4.20 A machine language program to output the characters `Hi`.

assumes that the string of characters `Hello world` was previously sent to the output stream, and the right half of the accumulator contains 21 (hex), which is the ASCII code for the `!` character. Execution of the store byte instruction to the output port sends the `!` to the output stream. ■

Example 4.22 The third row of the table in Figure 4.19 shows how to shut down the computer system by storing a byte to the power off port at Mem[FFFF]. This example sends 00 (hex) to the power off port. However, the system will shut down regardless of the value that you send to the power off port. ■

A Character Output Program

Figure 4.20 shows a simple machine-language program that outputs the characters `Hi` on the output device. It uses two instructions: 1101 raaa, which is the load byte instruction from a memory location, and 1111 raaa, which is the store byte instruction to the output port to output a character. The store byte instruction also stores a byte to the power off port to terminate the program.

The first listing shows the machine language program in binary. The computer system stores this sequence of ones and zeros in main memory starting at Mem[0000]. 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

Cycle	Instruction	State	Output
0	Initial state	A:	
1	Mem[0000]: D1000F	A: 0048	
2	Mem[0003]: F1FFFE	A: 0048	`H`
3	Mem[0006]: D10010	A: 0069	`H`
4	Mem[0009]: F1FFFE	A: 0069	`Hi`
5	Mem[000C]: F1FFFF	Shut down	

Figure 4.21 Trace table for execution of the program in Figure 4.20.

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.

Programs at Mem[0000]

In the Pep/10 system, the program resides at the very top of main memory starting at Mem[0000]. This is a simplification. In physical machines, as opposed to virtual machines like Pep/10, programs reside throughout the memory map.

The von Neumann cycle

Figure 4.21 is a trace table for execution of the program in Figure 4.20. The first column labeled Cycle is the *von Neumann cycle* that is wired into the CPU. The system assumes that the instructions of the program are in main memory adjacent to each other. The CPU cycles through the instructions one by one, executing each instruction in turn until the system shuts down.

Initial state

Cycle 0 shows the initial state of the machine. With this program the only relevant register to keep track of in the CPU is the accumulator (A). Although it appears that the accumulator has nothing in it, all memory locations in the system always have binary values. Something is in the accumulator even on start up, but it is some random value that is irrelevant to the execution of the program. The figure shows the accumulator as blank.

Load byte 'H'

Cycle 1 executes the D1000F instruction, which is load byte accumulator from Mem[000F]. Figure 4.20 shows that 48 (hex) is at Mem[000F]. So, the instruction loads the 48 into A[8:15], the right half of the accumulator.

Output 'H'

Cycle 2 executes the F1FFFE instruction, which is store byte accumulator to Mem[FFFE]. Because Mem[FFFE] is the output port, when it gets the 48 from A[8:15] it interprets the byte as an ASCII character and outputs `H`.

Load and output 'i'

Cycles 3 and 4 are similar to cycles 1 and 2, except that they load the 69 (hex) from Mem[0010], which is the ASCII value of `i`. Cycle 5 sends byte 69 to the power off port, which shuts the system down. Recall that any value sent to the power off port shuts down the system.

Shut down

The program listing in Figure 4.20 illustrates an important feature of all


```
Load program into memory at Mem[0000]
PC ← 0000
do
    Fetch: IR[0:7] ← Mem[PC]
    Decode: Decode instruction specifier IR[0:7]
    Increment: PC ← PC + 1
    if IR[0:7] is a dyadic instruction
        IR[8:23] ← Mem[PC]
        PC ← PC + 2
    Execute: Execute the instruction in the IR
while not shut down && instruction in IR is legal
```

Figure 4.22 The program execution process.

Memory stores instructions and data.

computer systems. Namely, main memory stores both instructions and data. In this program, the bytes in memory addresses 0000 through 000E are instructions and those at addresses 000F and 0010 are data.

4.3 von Neumann Machines

Main memory of ENIAC stored only data.

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 described in Section 3.1 was an example of this kind of machine. Its memory was used only to store data.

Main memory of EDSAC stored instructions and 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 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

Figure 4.22 is a pseudocode description of the program execution process. It consists of two parts—an initialization section and a `do` loop.

The initialization section loads the program into main memory starting at Mem[0000]. The details of how the program gets loaded into memory are beyond the scope of this book. The terminology "load" for this part of the initialization is common but unfortunate. Its use to describe an instruction always means a transfer of data from memory to CPU. Its use in this initialization means a transfer of data from the disk, which is where programs are stored, to memory.

Initialization section: Load program

After loading the program, the initialization section sets the value of the program counter (PC) to 0000. The program counter stores the address of the next instruction to be executed. It is set to 0000 because the next instruction to be executed is the first one of the program, which is stored at address 0000.

Initialization section: Set PC to 0000

The `do` loop in Figure 4.22 is the *von Neumann execution cycle* and is the mechanism by which instructions that are adjacent to each other in memory are executed. Following is a description of the body of the `do` loop.

Fetch: IR[0:7] ← Mem[PC]

Figure 4.2 (page 58) shows the 24-bit instruction register (IR) in the CPU. IR[0:7] is the first byte of the IR, which gets the instruction specifier from memory at the address specified by the PC.

von Neumann cycle: Fetch

Decode: Decode instruction specifier IR[0:7]

Now that the instruction specifier is in IR[0:7], the CPU can detect whether it is monadic or dyadic according to Figure 4.7 (page 62).

von Neumann cycle: Decode

Increment: PC ← PC + 1

The program counter is incremented by 1. If the instruction specifier that was fetched is for a monadic instruction then the PC contains the address of the first byte of the next instruction. On the other hand, if the instruction specifier is for a dyadic instruction then the PC contains the address of the operand specifier of the current instruction.

von Neumann cycle: Increment

```
if IR[0:7] is a dyadic instruction
    IR[8:23] ← Mem[PC]
    PC ← PC + 2
```

The body of the `if` statement executes only if the current instruction is dyadic. In that case, the PC contains the address of the operand specifier, which gets placed in IR[8:23], the rightmost two bytes of the instruction register. Then the program counter is incremented by 2. At this point, regardless of whether the instruction fetched was monadic or dyadic, it is in the IR, and the PC contains the address of the next instruction to be fetched.

Execute: Execute the instruction in the IR

Finally the instruction executes. Notice that the PC does not contain the address

von Neumann cycle: Execute

of the instruction that is executing now. Instead, it contains the address of the instruction that will be fetched in the next cycle after this instruction has finished executing.

The von Neumann cycle can be summarized as follows:

A simplified summary of the von Neumann cycle

- Fetch the instruction at Mem[PC] into the IR.
- Decode the instruction in the IR.
- Increment the PC.
- Execute the instruction in the IR.
- Repeat the cycle.

For monadic instructions, you can think of Fetch as fetching a single byte, and Increment as PC ← PC + 1. For dyadic instructions, you can think of Fetch as fetching three bytes, and Increment as PC ← PC + 3.

The simplification for dyadic instructions

The description in Figure 4.22 shows the above summary is a simplification for dyadic instructions. Fetch does not occur as a single operation that fetches all three bytes at once. Instead, the CPU fetches the instruction specifier byte, and then in a separate operation, it fetches the two bytes of the operand specifier. Similarly, the CPU does not increment PC by 3 in a single operation. Instead, it increments PC by 1 after fetching the instruction specifier, and then by 2 after fetching the operand specifier.

A von Neumann trace table

Figure 4.23 shows the *von Neumann trace table* corresponding to the trace table in Figure 4.21. Each cycle in Figure 4.21 consists of the five steps—fetch, decode, increment, execute, repeat—of the von Neumann cycle. Figure 4.23 expands each cycle of Figure 4.21 by showing three steps—fetch, increment, execute—that change the state of the system. It omits the two steps—decode, repeat—that do not change any values in the CPU or memory.

Cycle 0

Cycle 0 of Figure 4.23 shows the state of the computation after the initialization section executes. The table traces the content of three CPU registers: the accumulator A, the program counter PC, and the instruction register IR. The initialization section initializes PC to 0000.

Cycle 1-fetch

The row labeled 1-fetch fetches the instruction stored at Mem[0000] into IR. Note that this von Neumann trace table uses the simplified summary of the von Neumann cycle because it shows the fetch of instruction D1000F as a single operation. In practice, it takes several steps to do the fetch.

Cycle 1-increment

The row labeled 1-increment increments PC by 3. Again, even though this step of the von Neumann cycle appears as a single operation, in practice it requires several operations. An important detail whose significance will become apparent in the next chapter is the fact that PC contains 0003, which is *not* the address of the instruction D1000F in the instruction register. Instead, 0003 is the address of the next instruction F1FFFE that will be fetched.

Cycle 1-execute

The row labeled 1-execute executes the instruction D1000F that is in the instruction register. Because that instruction is load byte accumulator, A[8:15] gets 48 (hex) from Mem[000F].

Cycle	State			Output
0	A:	PC: 0000	IR:	
1-fetch	A:	PC: 0000	IR: D1000F	
1-increment	A:	PC: 0003	IR: D1000F	
1-execute	A: 0048	PC: 0003	IR: D1000F	
2-fetch	A: 0048	PC: 0003	IR: F1FFFE	
2-increment	A: 0048	PC: 0006	IR: F1FFFE	
2-execute	A: 0048	PC: 0006	IR: F1FFFE	`H`
3-fetch	A: 0048	PC: 0006	IR: D10010	`H`
3-increment	A: 0048	PC: 0009	IR: D10010	`H`
3-execute	A: 0069	PC: 0009	IR: D10010	`H`
4-fetch	A: 0069	PC: 0009	IR: F1FFFE	`H`
4-increment	A: 0069	PC: 000C	IR: F1FFFE	`H`
4-execute	A: 0069	PC: 000C	IR: F1FFFE	`Hi`
5-fetch	A: 0069	PC: 000C	IR: F1FFFF	`Hi`
5-increment	A: 0069	PC: 000F	IR: F1FFFF	`Hi`
5-execute	Shut down			

Figure 4.23 von Neumann trace table for execution of the program in Figure 4.20.

The row labeled 2-fetch fetches the instruction stored at Mem[0003] into IR because PC contains 0003. The row labeled 2-increment increments PC by 3, giving it a value of 0006. The row labeled 2-execute executes the instruction F1FFFE that is in the instruction register. Because that instruction is store byte accumulator, the output port gets 48 (hex) from A[8:15], which appears as the ASCII character `H` on the output device. *Cycle 2*

Cycles 3 and 4 are similar to cycles 1 and 2. In cycle 5-increment, PC gets 000F. However, Mem[000F] never gets fetched because execution of the instruction shuts down the computer.

Address Machine Language (bin)

```
0000     1101 0001 1111 1111 1111 1101
0003     1111 0001 0000 0000 0001 0101
0006     1101 0001 1111 1111 1111 1101
0009     1111 0001 1111 1111 1111 1110
000C     1101 0001 0000 0000 0001 0101
000F     1111 0001 1111 1111 1111 1110
0012     1111 0001 1111 1111 1111 1111
0015     0000 0000
```

Address Machine Language (hex)

```
0000     D1FFFD ;Load byte first char from input port
0003     F10015 ;Store byte first char to 0015
0006     D1FFFD ;Load byte from input port
0009     F1FFFE ;Store byte to output port
000C     D10015 ;Load byte first char from 0015
000F     F1FFFE ;Store byte first char to output port
0012     F1FFFF ;Store byte to power off port
0015     00     ;One byte storage for first char
```

Input

```
up
```

Output

```
pu
```

Figure 4.24 A machine language program to input two characters and output them in reverse order.

A Character Input Program

The program of Figure 4.24 inputs two characters from the input port and outputs them in reverse order to the output port. It uses the load byte instruction with direct addressing to get the characters from the input port.

The instruction specifier D1 of the first instruction, D1FFFD, has an opcode that specifies the load byte instruction, register-r field that specifies the accumulator, and addressing-aaa field that specifies direct addressing. The operand specifier FFFD is the address of the input port. So, the first instruction inputs the ASCII code for the first character from the input port into the right half of the accumulator A[8:15]. The second instruction, F10015, is store byte from A[8:15] to Mem[0015]. The effect of the first two instructions is to transfer the ASCII code for the first character from the input port to Mem[0015].

Input from the input port

Store to memory

The third instruction, D1FFFD, is identical to the first and inputs the ASCII code for the second character from the input port into A[8:15]. The fourth instruction, F1FFFE, is store byte from A[8:15] to the output port at Mem[FFFE].

Input from the input port, store to the output port

Cycle	Instruction	State		Input	Output
0	Initial state	A:	Mem[0015]:	`up`	
1	Mem[0000]: D1FFFD	A: 0075	Mem[0015]:	`p`	
2	Mem[0003]: F10015	A: 0075	Mem[0015]: 75	`p`	
3	Mem[0006]: D1FFFD	A: 0070	Mem[0015]: 75		
4	Mem[0009]: F1FFFE	A: 0070	Mem[0015]: 75		`p`
5	Mem[000C]: D10015	A: 0075	Mem[0015]: 75		`p`
6	Mem[000F]: F1FFFE	A: 0075	Mem[0015]: 75		`pu`
7	Mem[0012]: F1FFFF	Shut down			

Figure 4.25 Trace table for execution of the program in Figure 4.24.

The effect of second pair of instructions is to transfer the second character from the input port to the output port.

Load from memory, store to the output port

The fifth instruction, D10015, loads the first character previously stored in Mem[0015] into A[8:15]. The sixth instruction, F1FFFE, sends A[8:15] to the output port at Mem[FFFE]. The effect of this pair of instructions is to transfer the first character to the output port.

Store to the power off port

The last instruction, F1FFFF, sends a byte to the power off port, which shuts down the computer. Figure 4.25 is a trace table for execution of the program with an input of `up`.

Converting Decimal to ASCII

Figure 4.26 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.

Load 5 from memory, add 3 from memory

The two numbers to be added are 5 and 3. The program stores them at Mem[000F] and Mem[0011]. 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 the output port at Mem[FFFE]. 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.

So, the program must convert the 8 (dec) = 0000 1000 (bin) to the ASCII

Address Machine Language (hex)

```
0000     C1000F ;Load word accumulator 0005 from Mem[000F]
0003     510011 ;Add accumulator 0003 from Mem[0011]
0006     810013 ;Or accumulator 0030 from Mem[0013]
0009     F1FFFE ;Store byte to output port
000C     F1FFFF ;Store byte to power off port
000F     0005   ;Decimal 5
0011     0003   ;Decimal 3
0013     0030   ;Mask for ASCII char
```

Output

```
8
```

Figure 4.26 A machine language program to add two numbers and output their sum.

character `8`, which is 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 OR'ing the accumulator with the mask 0000 0000 0011 0000, using the Or register instruction:

Convert integer value to ASCII

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

Store to the output port

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

If you replace the word at Mem[0011] 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 ASCII byte, the program cannot output a result that contains more than one character. The next chapter shows how to remedy this shortcoming.

Adding Numbers from the Input Port

Figure 4.27 shows a program that inputs two single-digit numbers from the input port and outputs their single-digit sum. It requires the use of an AND mask as well an OR mask to convert the sum to an ASCII character. Following is a description of how the program finds the sum of two and five. The input stream is `25` with no space between the `2` and the `5`.

Address Machine Language (hex)

```
0000     D1FFFD ;Load byte accumulator from input port
0003     E10018 ;Store word accumulator to Mem[0018]
0006     D1FFFD ;Load byte accumulator from input port
0009     510018 ;Add accumulator from Mem[0018]
000C     71001A ;And accumulator from Mem[001A]
000F     81001C ;Or accumulator from Mem[001C]
0012     F1FFFE ;Store byte to output port
0015     F1FFFF ;Store byte to power off port
0018     0000   ;One-word storage for first number
001A     000F   ;AND mask
001C     0030   ;OR mask
```

Input

```
25
```

Output

```
7
```

Figure 4.27 A machine language program to input two numbers and output their sum.

The first instruction D1FFFD loads the ASCII value of `2`, which is 0011 0010, into the right half of the accumulator A[8:15], and 0000 0000 into A[0:7]. The second instruction E10018 stores the word (two bytes) from the accumulator to Mem[0018]. The third instruction loads the ASCII value of `5` into A[8:15].

Input ASCII 2 from the input port, store to memory

Input ASCII 5 from the input port

The fourth instruction 510018 performs the addition as follows.

```
       0000 0000 0011 0010
ADD    0000 0000 0011 0101
       ---------------------
       0000 0000 0110 0111
```

Add ASCII 2 plus ASCII 5

The relevant part of the addition is the rightmost nybble, namely 0010 + 0101 = 0111 (bin), which is 2 + 5 = 7 (dec). The addition of the penultimate nybble, namely 0011 + 0011 = 0110 (bin) is irrelevant. It needs to be stripped away and replaced with 0011 to convert the sum to its equivalent ASCII character.

The fifth instruction 71001A strips away the irrelevant 1's with the AND mask as follows.

```
       0000 0000 0110 0111
AND    0000 0000 0000 1111
       ---------------------
       0000 0000 0000 0111
```

Zero out garbage bits

The AND mask 0000 0000 0000 1111 (bin) = 000F (hex) is at Mem[001A]. Everywhere there is a 0 in the mask, the result is 0. Everywhere there is a 1 the

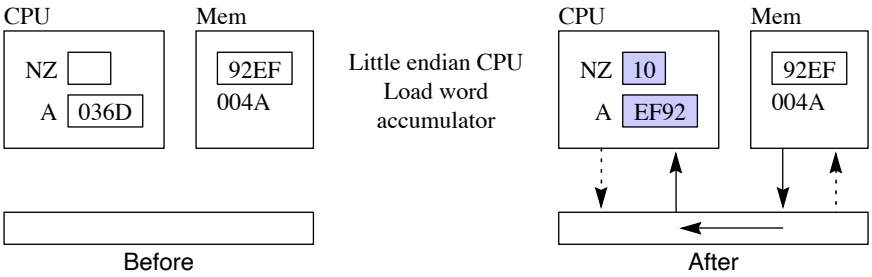


Figure 4.28 The load word instruction in a little-endian CPU.

mask, the result is unchanged. Consequently, this AND mask sets A[0:11] to 0 and leaves A[12:15] unchanged. The accumulator now contains the decimal value of the sum of the two numbers.

The sixth instruction 71001C replaces the penultimate nybble with 0011 using the OR mask as follows.

```
        0000 0000 0000 0111
OR      0000 0000 0011 0000
        ---------------------
        0000 0000 0011 0111
```

Convert integer value to ASCII

A[8:15] now contains the ASCII value of the sum of the two numbers, which the next instruction F1FFFE sends to the output port.

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 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.

Store word with big-endian

Pep/10 is a big-endian CPU. Figure 4.12 (page 68) shows the effect of the store word 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 (page 67) shows the load word 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.

Load word with big-endian

Before	Instruction	After
A: 00 00 00 00 Mem[004A]: 89 AB CD EF	Big-endian CPU Load accumulator	A: 89 AB CD EF Mem[004A]: 89 AB CD EF
A: 00 00 00 00 Mem[004A]: 89 AB CD EF	Little-endian CPU Load accumulator	A: EF CD AB 89 Mem[004A]: 89 AB CD EF

Figure 4.29 The load accumulator instruction with a 32-bit register.

In contrast, Figure 4.28 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.

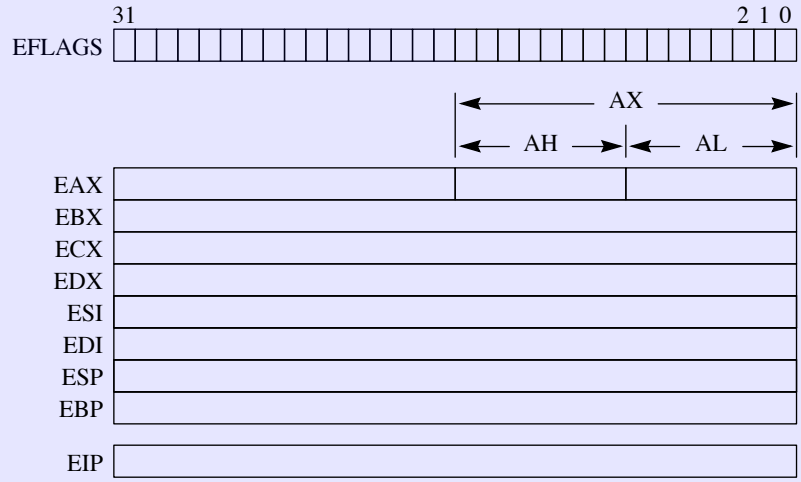
Figure 4.29 shows the effect of a load instruction in a CPU with 32-bit registers for both big-endian and little-endian ordering. A 32-bit register holds four bytes, which are loaded into the accumulator from most significant to least significant byte, or from least significant to most significant byte, depending on whether the CPU uses big-endian or little-endian ordering, respectively. Note that the memory address of the byte EF is 004D.

Etymology of endian

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 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.

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.

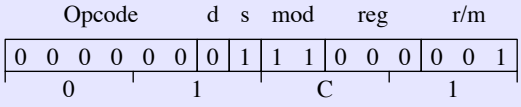


The above figure shows the registers in a typical 32-bit model. The x86 processors are little-endian and number the bits from right to left in a register starting with 0 for the least significant bit. The EFLAGS register has a number of status bits besides the four NZVC bits in Pep/10. The table on the right shows the location of the four bits that correspond to the Pep/10 status bits. SF stands for sign flag, and OF stands for overflow flag.

Pep/10	x86	Position
N	SF	EFLAGS[7]
Z	ZF	EFLAGS[6]
V	OF	EFLAGS[11]
C	CF	EFLAGS[0]

The x86 architecture has four general-purpose accumulators named EAX, EBX, ECX, and EDX that correspond to the single Pep/10 accumulator. The above figure 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/10. 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/10. EBP is the base pointer, which points to the bottom of the current stack frame. Pep/10 has no corresponding register. EIP is called the instruction pointer in Intel terminology and corresponds to the program counter PC in Pep/10.



The above figure 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/10 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/10 illustrates the concepts underlying machine languages in all von Neumann machines without the above complexities.

Written Exercises

Section 4.1

⋆ **4.1** (a) How many bytes are in the main memory of the Pep/10 computer? (b) How many words are in it? (c) How many bits are in it? (d) How many total bits are in the Pep/10 CPU? (e) How many times bigger in terms of bits is the main memory than the CPU?

4.2 Assume the memory map of Figure 4.6 in bare metal mode so that the memory-mapped and power off ports are not available for storage. (a) If main memory were completely filled with monadic instructions how many instructions would it contain? (b) How many instructions would it contain if they were all dyadic?

⋆ **4.3** Answer the following questions for the machine language instructions 5AF82C 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?

4.4 Answer the questions in Exercise 4.3 for the machine language instructions 6B00AC and F70BD3.

Section 4.2

⋆ **4.5** Suppose Pep/10 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) 690A41
(g) 610A3F (h) 810A3F (i) 1F

4.6 Repeat Exercise 4.5 for the following statements:

(a) C90A3F (b) D90A3F (c) F10A41
(d) E10A41 (e) 590A3F (f) 610A41
(g) 790A3F (h) 890A3F (i) 1E

Section 4.3

4.7 Construct the von Neumann trace table similar to Figure 4.23 but for the program in Figure 4.24. Include four columns—Cycle, State, Input, and Output. In the State column, trace A, PC, IR, and Mem[0015].

4.8 Construct the von Neumann trace table similar to Figure 4.23 but for the program in Figure 4.26. Include three columns—Cycle, State, and Output. In the State column, trace A, PC, and IR.

4.9 Construct the trace table similar to Figure 4.25 but for the program in Figure 4.27. Include five columns—Cycle, Instruction, State, Input, and Output. In the State column, trace A and Mem[0018]. Assume the initial value of the Input is `54`.

Programming Problems

Section 4.2

4.10 Write a machine language program to output the following word to the output port. Use direct addressing with the letters of the words in memory after the shutdown instruction. The first letter of the word is uppercase.

(a) `Cat` (b) `Frog` (c) `Tiger`
(d) `Lizard` (e) `Giraffe` (f) `Elephant`
(g) `Butterfly` (h) `Chimpanzee` (i) `Hummingbird`
(j) `Hippopotamus` (k) `Tyrannosaurus`

Section 4.3

4.11 The program in Figure 4.24 has seven instructions and storage in memory for one character. Write the equivalent program with only five instructions and no storage in memory. Use the index register to store the second letter.

4.12 Write a machine language program to input three letters and output them in reverse order. For example, if the input stream is `tab` the output should be `bat`. (a) Do not use the index register. (b) Use the technique of Problem 4.11 to minimize the number of instructions in the program.

4.13 Write a machine language program to input four letters and output them in reverse order. For example, if the input stream is `brag` the output should be `garb`. (a) Do not use the index register. (b) Use the technique of Problem 4.11 to minimize the number of instructions in the program.

4.14 Write a machine language program to add the three numbers 2, –3, and 6, and output the sum to the output port. Use direct addressing with the decimal values of the numbers (including the decimal value of –3) in memory after the shutdown instruction. Do not use the subtract, negate, or bitwise Not instructions.

4.15 Write a machine language program to input three one-digit numbers, add them, and output the one-digit sum. There can be no space between the three one-digit numbers on input. For example, if the input is `143`, the output should be `8`.

4.16 Write a machine language program to input two one-digit numbers, add them, and output the one-digit sum. There should be exactly one space between the two one-digit numbers on input. For example, if the input is `2 5`, the output should be `7`.

4.17 Write a machine language program to input three one-digit numbers, add them, and output the one-digit sum. There should be exactly one space between the three one-digit numbers on input. For example, if the input is `1 4 3`, the output should be `8`.

