Chapter 5

Assembly Language

The Level-ISA3 language is machine language—sequences of 1's and 0's sometimes abbreviated to hexadecimal. Computer pioneers had to program in machine language, but they soon revolted against such an indignity. Memorizing the opcodes of the machine and having to continually refer to ASCII charts and hexadecimal tables to get their programs into binary was no fun. The assembly level was invented to relieve programmers of the tedium of programming in binary.

Chapter 4 describes the Pep/10 computer at Level ISA3, the machine level. This chapter describes Level Asmb5, the assembly level. Between these two levels lies the operating system. Remember that the purpose of levels of abstraction is to hide the details of the system at the lower levels. This chapter illustrates that principle of information hiding. You will use the system calls of the operating system without knowing the details of their implementation. That is, you will learn *what* the system calls do without learning *how* they do it. Chapter 8 reveals the inner workings of the Pep/10 operating system.

5.1 An Assembler at Level 3

The purpose of the Asmb3 level

Some aspects of assembly language are independent of the operating system. Programs written using only those aspects of the language run on the Pep/10 virtual machine in bare metal mode, that is, without an operating system. This Asmb3 level is used only to introduce assembly language. The following section, as well as the remainder of the book, describes assembly language at level Asmb5.

Consider the program of Figure 4.20 (page 77), which outputs `Hi`. It contains two types of bit patterns, one for instructions and one for data. The bits at Mem[0000] to Mem[000E] are instruction bits. For example, the CPU interprets the bits at Mem[0009] to Mem[000B] as the dyadic instruction store byte accumulator. The bits at Mem[000F] to Mem[0010] are data bits. For

Instruction Specifier	Mnemonic	Instruction	Addressing Modes	Status Bits
0000 0000		Illegal instruction		
0000 0001	`RET`	Return from `CALL`	Monadic	
0000 0010	`SRET`	Return from system `CALL`	Monadic	
0000 0011	`MOVFLGA`	Move NZVC flags to A[12 : 15]	Monadic	
0000 0100	`MOVAFLG`	Move A[12 : 15] to NZVC flags	Monadic	NZVC
0000 0101	`MOVSPA`	Move SP to A	Monadic	
0000 0110	`MOVASP`	Move A to SP	Monadic	
0000 0111	`NOP`	No operation	Monadic	
0001 100r	`NEGr`	Negate r	Monadic	NZVC
0001 101r	`ASLr`	Arithmetic shift left r	Monadic	NZVC
0001 110r	`ASRr`	Arithmetic shift right r	Monadic	NZVC
0001 111r	`NOTr`	Bitwise NOT r	Monadic	NZ
0010 000r	`ROLr`	Rotate left r	Monadic	NZC
0010 001r	`RORr`	Rotate right r	Monadic	NZC

Figure 5.1 The Pep/10 instruction set. (*Continues*)

example, the output port interprets the bits that it recieves from Mem[000F] as the ASCII `H` character.

These two types—instruction bit patterns and data bit patterns—are a direct consequence of the von Neumann design, where program and data share the same memory with a binary representation for each. Assembly language contains two types of statements that correspond to these two types of bit patterns. Mnemonic statements correspond to the instruction bit patterns, and pseudo-operations correspond to the data bit patterns.

Mnemonic Statements

Suppose the machine language instruction

```
F1FFFE
```

is stored at some memory location. This is the store byte register r instruction. The register-r bit is 0, which indicates the accumulator as opposed to the index register. The addressing-aaa field is 001, which specifies direct addressing.

This instruction is written in Pep/10 assembly language as

```
STBA 0xFFFE,d
```

The mnemonic `STBA`, which stands for *store byte accumulator*, is written in place of opcode 1100 and register-r field 0. The letter `d` stands for *direct addressing*. The operand specifier is written in hexadecimal `FFFE` preceded by

Instruction Specifier	Mnemonic	Instruction	Addressing Modes	Status Bits
0010 010a	`BR`	Branch unconditional	i, x	
0010 011a	`BRLE`	Branch if less than or equal to	i, x	
0010 100a	`BRLT`	Branch if less than	i, x	
0010 101a	`BREQ`	Branch if equal to	i, x	
0010 110a	`BRNE`	Branch if not equal to	i, x	
0010 111a	`BRGE`	Branch if greater than or equal to	i, x	
0011 000a	`BRGT`	Branch if greater than	i, x	
0011 001a	`BRV`	Branch if V	i, x	
0011 010a	`BRC`	Branch if C	i, x	
0011 011a	`CALL`	Call subroutine	i, x	
0011 1aaa	`SCALL`	System call	i, d, n, s, sf, x, sx, sfx	
0100 0aaa	`ADDSP`	Add to SP	i, d, n, s, sf, x, sx, sfx	
0100 1aaa	`SUBSP`	Subtract from SP	i, d, n, s, sf, x, sx, sfx	
0101 raaa	`ADDr`	Add to r	i, d, n, s, sf, x, sx, sfx	NZVC
0110 raaa	`SUBr`	Subtract from r	i, d, n, s, sf, x, sx, sfx	NZVC
0111 raaa	`ANDr`	Bitwise AND to r	i, d, n, s, sf, x, sx, sfx	NZ
1000 raaa	`ORr`	Bitwise OR to r	i, d, n, s, sf, x, sx, sfx	NZ
1001 raaa	`XORr`	Bitwise XOR to r	i, d, n, s, sf, x, sx, sfx	NZ
1010 raaa	`CPWr`	Compare word to r	i, d, n, s, sf, x, sx, sfx	NZVC
1011 raaa	`CPBr`	Compare byte to r[8 : 15]	i, d, n, s, sf, x, sx, sfx	NZVC
1100 raaa	`LDWr`	Load word r from memory	i, d, n, s, sf, x, sx, sfx	NZ
1101 raaa	`LDBr`	Load byte r[8 : 15] from memory	i, d, n, s, sf, x, sx, sfx	NZ
1110 raaa	`STWr`	Store word r to memory	d, n, s, sf, x, sx, sfx	
1111 raaa	`STBr`	Store byte r[8 : 15] to memory	d, n, s, sf, x, sx, sfx	

Figure 5.1 (*Continued*) The Pep/10 instruction set.

`0x`, which stands for *hexadecimal constant*.

A mnemonic is a memory aid. It is easier to remember that `STBA` stands for the store byte accumulator instruction than to remember that opcode 1111 and register-r value of 0 stand for the load word accumulator instruction. Figure 5.1 lists the mnemonic for each instruction in the Pep/10 instruction set. The lowercase letter `r` in a mnemonic can be either an uppercase `A`, corresponding to a register-r field of 0, or an uppercase `X`, corresponding to a register-r field of 1.

In Pep/10 assembly language, you specify the addressing mode by placing one or more letters after the operand specifier with a comma between them. Figure 5.2 shows the letters that go with each of the eight addressing modes.

Example 5.1 Here are four examples of instructions written in binary machine language and in assembly language.

aaa	Addressing Mode	Letters
000	Immediate	`i`
001	Direct	`d`
010	Indirect	`n`
011	Stack-relative	`s`
100	Stack-relative deferred	`sf`
101	Indexed	`x`
110	Stack-indexed	`sx`
111	Stack-deferred indexed	`sfx`

Figure 5.2 The letters that specify the addressing mode in Pep/10 assembly language.

```
1100 0011 0000 0000 1001 1010    LDWA 0x009A,s
1100 0110 0000 0000 1001 1010    LDWA 0x009A,sx
1100 1011 0000 0000 1001 1010    LDWX 0x009A,s
1100 1110 0000 0000 1001 1010    LDWX 0x009A,sx
```

The opcode for each of the instructions is 1100, which Figure 5.1 shows is the `LDWr` instruction. The first two instructions have a register-r field of 0, and so their mnemonic is `LDWA`. The last two instructions have a register-r field of 1, and so their mnemonic is `LDWX`. The first and third instructions have an addressing-aaa field of 011, and so their addressing mode letter is `s`. The second and fourth instructions have an addressing-aaa field of 110, and so their addressing mode letters are `sx`. ■

Pseudo-Operations

Pseudo-operations (pseudo-ops) are assembly language statements. Pseudo-ops do not have opcodes and do not correspond to any of the 37 instructions in the Pep/10 instruction set. Four of the Pep/10 pseudo-ops are:

`.ASCII`	A string of ASCII bytes
`.BYTE`	A byte value
`.WORD`	A word value
`.BLOCK`	A block of bytes, all 0's

Each of the above pseudo-ops inserts data bits into the machine language program.

Pseudo means false. Pseudo-ops are so called because the bits that they generate do not correspond to opcodes, as do the bits generated by the 37 mnemonic statements. They are not true instruction operations. Pseudo-ops are also called *assembler directives* or *dot commands* because each must be preceded by a `.` in assembly language.

The `.ASCII` pseudo-op expects a string of ASCII characters enclosed in double quotes. It inserts the corresponding string of bytes, one byte for each character.

Example 5.2 The dot command

```
.ASCII "Hello world!"
```

inserts the following string of 12 bytes:

```
48 65 6C 6C 6F 20 77 6F 72 6C 64 21
```

The hexadecimal value `48` is the ASCII code for the letter `H`. The hexadecimal value `65` is the ASCII code for the letter `e`, and so on. ■

The `.BYTE` pseudo-op expects a decimal constant, a hexadecimal constant prefixed with `0x`, or a single ASCII character enclosed in single quotes. It always inserts a single byte. The `.WORD` pseudo-op expects the same thing. However, it always inserts one word (two bytes).

Example 5.3 Here are some examples of the bytes inserted by the `.BYTE` dot command.

```
.BYTE 43       inserts  2B
.BYTE -3       inserts  FD
.BYTE 0xF7     inserts  F7
.BYTE 'H'      inserts  48
.BYTE 365      ERROR: Decimal constant is out of byte range
```

Here are some examples of the words inserted by the `.WORD` dot command.

```
.WORD 43       inserts  002B
.WORD -3       inserts  FFFD
.WORD 0xF7     inserts  00F7
.WORD 'H'      inserts  0048
.WORD 365      inserts  016D
.WORD 0xC42E   inserts  C42E
```

Both dot commands insert the equivalent hexadecimal value of what they are given. If necessary, they pad the hexadecimal value with leading zeros or ones (for negative integers), so that `.BYTE` always inserts exactly one byte and `.WORD` always inserts exactly one word. ■

The `.BLOCK` pseudo-op expects a decimal constant or a hexadecimal constant prefixed with `0x`. It inserts that number of bytes, all of which are 0's.

Example 5.4 Here are some examples of the words inserted by the `.BLOCK` dot command.

```
.BLOCK 3    inserts 00 00 00
.BLOCK 4    inserts 00 00 00 00
.BLOCK 0xC  inserts 00 00 00 00 00 00 00 00 00 00 00 00
```


Assembly Language Source Code

```
LDBA 0x000F,d    ;Load byte accumulator 'H'
STBA 0xFFFE,d    ;Store byte accumulator output port
ldba    0x0010  ,  D  ;Load byte accumulator 'i'
STBA 0xfffe,d    ;Store byte accumulator output port
STBA 0xFFFF,d    ;Store byte power off port
.ASCII "Hi"      ;ASCII "Hi" characters
```

Assembly Language Listing

```
-------------------------------------------------------------------------------
      Object
Addr  code   Symbol   Mnemon  Operand     Comment
-------------------------------------------------------------------------------
0000  D1000F          LDBA    0x000F,d    ;Load byte accumulator 'H'
0003  F1FFFE          STBA    0xFFFE,d    ;Store byte accumulator output port
0006  D10010          LDBA    0x0010,d    ;Load byte accumulator 'i'
0009  F1FFFE          STBA    0xFFFE,d    ;Store byte accumulator output port
000C  F1FFFF          STBA    0xFFFF,d    ;Store byte power off port
000F  4869            .ASCII  "Hi"        ;ASCII "Hi" characters
-------------------------------------------------------------------------------
```

Object Code

```
D1 00 0F F1 FF FE D1 00 10 F1 FF FE F1 FF FF 48
69
```

Figure 5.3 An assembly language program to output the characters `Hi`.

The last example inserts 12 bytes, all 0's, because C (hex) = 12 (dec). ■

Assembler input and output

Figure 5.3 is an assembly language program that is equivalent to the machine language program in Figure 4.20 (page 77). The programmer writes the assembly language *source code*. The *assembler* is a program that takes the source code as input and produces the assembly language *listing* and the *object code* as output.

The programmer can be a little careless about style when writing the source code. In particular, capitalization and spacing need not be consistent. For example, the source code in the figure shows an instance where `D` denotes direct addressing instead of `d`.

The second column of the assembly language listening in Figure 5.3 labeled `Object code` shows the machine language generated by the assembly language statement. The first column labeled `Addr` shows the address in memory where the machine language statement is stored. The remainder of the listing shows the assembly language statement in a standard capitalization and spacing style.

Assembly Language Listing

```
0000  D1FFFD          LDBA    0xFFFD,d    ;Load first char from input port
0003  F10015          STBA    0x0015,d    ;Store first char to 0015
0006  D1FFFD          LDBA    0xFFFD,d    ;Load from input port
0009  F1FFFE          STBA    0xFFFE,d    ;Store to output port
000C  D10015          LDBA    0x0015,d    ;Load first char from 0015
000F  F1FFFE          STBA    0xFFFE,d    ;Store first char to output port
0012  F1FFFF          STBA    0xFFFF,d    ;Store to power off port
0015  00              .BLOCK  1           ;One byte storage for first char
```

Figure 5.4 The listing of an assembly language program to input two characters and output them in reverse order.

Assembly Language Listing

```
0000  C1000F          LDWA    0x000F,d    ;Load 0005 from Mem[000F]
0003  510011          ADDA    0x0011,d    ;Add 0003 from Mem[0011]
0006  810013          ORA     0x0013,d    ;OR 0030 from Mem[0013]
0009  F1FFFE          STBA    0xFFFE,d    ;Store to output port
000C  F1FFFF          STBA    0xFFFF,d    ;Store to power off port
000F  0005            .WORD   5           ;Decimal 5
0011  0003            .WORD   3           ;Decimal 3
0013  0030            .WORD   0x0030      ;Mask for ASCII char
```

Figure 5.5 The listing of an assembly language program to add two numbers and output their sum.

In the assembly language program of Figure 5.3, the first five lines are mnemonic statements with mnemonics `LDBA` and `STBA`. The

```
.ASCII  "Hi"
```

dot command inserts the ASCII code for the letters `H` and `i` into the object code, relieving the programmer of the task of looking up the code manually.

Figure 5.4 is the listing of an assembly language program that is equivalent to the machine language program in Figure 4.24 (page 83). The

```
.BLOCK  1
```

dot command inserts one byte of all 0's into the object code. This is the storage location in main memory for the first letter of the input stream.

Figure 5.5 is the listing of an assembly language program that is equivalent to the machine language program in Figure 4.26 (page 85). Each `.WORD` dot command inserts exactly two bytes into the object code, representing the value given to the dot command.

The difference between `.BLOCK` and `.WORD` is that `.WORD` expects a value, and inserts that value into the object code. Programmers use `.WORD` when

they know the specific value before the program executes, such as the value of a mask. On the other hand, `.BLOCK` expects a numeric value, and inserts that number of bytes into the object code. Programmers use `.BLOCK` to reserve storage for values that are only known at runtime, such as values from the input stream.

Symbols

The assembly language program in Figure 5.5 is certainly easier to write than its machine language version. However, the programmer still has a major irritant—namely, calculation of the addresses 000F, 0011, and 0013 where the data for the program are stored. These addresses depend on how many preceding program statements exist. Anytime you add an instruction to your program, you must re-calculate the addresses for the data at the bottom of your program.

Assembly language symbols eliminate the problem of manually determining addresses. The assembler lets you associate a *symbol* with a *memory address*. Anywhere in the program you need to refer to the address, you can refer to the symbol instead. If you ever modify a program by adding or removing statements, when you reassemble the program the assembler will calculate the new address associated with the symbol. You do not need to rewrite the statements that refer to the changed addresses via the symbols.

The assembly language program of Figure 5.6 produces object code identical to the machine language code of Figure 4.27 (page 86). It defines three symbols—`num`, `andMask`, and `orMask`—and uses three predefined symbols—`charIn`, `charOut`, and `pwrOff`.

The syntax rules for symbols are similar to the syntax rules for C identifiers. The first character must be a letter or the underscore character `_`, and the following characters must be letters, digits, or the underscore character. Symbols can be a maximum of only eight characters long. The characters are case sensitive. For example, `Number` would be a different symbol from `number` because of the uppercase `N`.

You can define a symbol on any assembly language line by placing it at the beginning of the line. When you define a symbol, you must terminate it with a colon `:` . No spaces are allowed between the last character of the symbol and the colon.

In the program of Figure 5.6, the statement

```
num:      .BLOCK  2
```

defines the symbol `num`, in addition to allocating a block of two bytes. Although this line has spaces between the colon and the pseudo-op, the assembler does not require them.

When the assembler detects a symbol definition, it stores the symbol and its value in a *symbol table*. The *value* of a symbol is the the *address* in memory of where the first byte of the object code generated from that line will be inserted into the object code. Figure 5.6(b) shows the symbol table for this program.

Assembly Language Listing

```
0000  D1FFFD          LDBA    charIn,d    ;Input first digit
0003  E10018          STWA    num,d       ;Store to num
0006  D1FFFD          LDBA    charIn,d    ;Input second digit
0009  510018          ADDA    num,d       ;Add num to second digit
000C  71001A          ANDA    andMask,d   ;Zero out the left nybble
000F  81001C          ORA     orMask,d    ;Convert sum to ASCII character
0012  F1FFFE          STBA    charOut,d   ;Output sum
0015  F1FFFF          STBA    pwrOff,d    ;Shut down
0018  0000    num:    .BLOCK  2
001A  000F    andMask: .WORD  0x000F
001C  0030    orMask:  .WORD  0x0030
```

(a) The program.

Symbol	Value	Symbol	Value
num	0018	charIn	FFFD
andMask	001A	charOut	FFFE
orMask	001C	pwrOff	FFFF

(b) The symbol table.

Figure 5.6 An assembly language program with symbols to input two numbers and output their sum.

You can see from the table that the value of the symbol `orMask` is 001C. Furthermore, the address of the first byte of 0030 (the object code generated from that line) is 001C.

When you refer to the symbol, you cannot include the colon. The statement

```
STWA    num,d       ;Store to num
```

refers to the symbol `num`. Because `num` has the value 0018 (hex), this statement generates the same object code that

```
STWA    0x0018,d    ;Store to num
```

would generate. Similarly, the statement

```
LDBA    charIn,d    ;Input first digit
```

generates the same object code that

```
LDBA    0xFFFD,d    ;Input first digit
```


Assembly Language Listing

```
0000  D1FFFD          LDBA    charIn,d    ;Load first char from input port
0003  F10015          STBA    char_1,d    ;Store first char to char_1
0006  D1FFFD          LDBA    charIn,d    ;Load from input port
0009  F1FFFE          STBA    charOut,d   ;Store to output port
000C  D10015          LDBA    char_1,d    ;Load first char from char_1
000F  F1FFFE          STBA    charOut,d   ;Store first char to output port
0012  F1FFFF          STBA    pwrOff,d    ;Store to power off port
0015  00     char_1:  .BLOCK  1           ;One byte storage for first char
```

Figure 5.7 An assembly language program with symbols to input two characters and output them in reverse order.

would generate, because the value of the pre-defined symbol `charIn` is FFFD (hex).

Note that the value of a symbol is an address, not the content of the cell at that address. When this program executes, Mem[001A] will contain 000F, which it gets from the `.WORD` pseudo-op. The value of symbol `andMask` is 001A, which is different from 000F. It might help you to visualize the value of a symbol as coming from the address column on the assembler listing in the line that contains the symbol definition.

Symbols not only relieve you of the burden of calculating addresses manually, they also make your programs easier to read. `num` is easier on the eyes than 0x0018. Good programmers are careful to select meaningful symbols for their programs to enhance readability.

Figure 5.7 is the listing of a program with symbols that produces object code identical to that in Figure 5.4. The symbol `char_1` has value 0015 (hex), so that the statement

```
STBA    char_1,d    ;Store first char to char_1
```

generates the same object code that

```
STBA    0x0015,d    ;Store first char to char_1
```

generates.

Macros

The first two lines of code in Figure 5.7

```
LDBA    charIn,d    ;Load first char from input port
STBA    char_1,d    ;Store first char to char_1
```

occur frequently together in assembly language programs. As a pair they perform a single operation—inputting a character from the input stream to a location in memory. Furthermore, the two lines of code

Assembly Language Source Code

```
         @CHARI  char_1,d    ;Input first char to char_1
         LDBA    charIn,d    ;Load from input port
         STBA    charOut,d   ;Store to output port
         @CHARO  char_1,d    ;Output char_1
         STBA    pwrOff,d    ;Store to power off port
char_1: .BLOCK   1           ;One byte storage for first char
```

Assembly Language Listing

```
             ;        @CHARI  char_1,d    ;Input first char to char_1
0000  D1FFFD           LDBA    charIn,d
0003  F10015           STBA    char_1,d
             ;        End @CHARI
0006  D1FFFD           LDBA    charIn,d    ;Load from input port
0009  F1FFFE           STBA    charOut,d   ;Store to output port
             ;        @CHARO  char_1,d    ;Output char_1
000C  D10015           LDBA    char_1,d
000F  F1FFFE           STBA    charOut,d
             ;        End @CHARO
0012  F1FFFF           STBA    pwrOff,d    ;Store to power off port
0015  00     char_1:   .BLOCK  1           ;One byte storage for first char
```

Figure 5.8 An assembly language program with macros to input two characters and output them in reverse order.

```
LDBA    charIn,d    ;Load first char from input port
STBA    char_1,d    ;Store first char to char_1
```

also occur frequently together. As a pair they perform a single operation—outputting a character from a location in memory to the output stream.

An assembly language *macro* allows the programmer to write a single statement that expands to two or more actual assembly language statements. In the Source Code part of Figure 5.8, the programmer has written a single macro statement

```
@CHARI  char_1,d    ;Input first char to char_1
```

instead of writing the first two statements of Figure 5.7.

Macro mnemonic statements begin with the character `@`. Like mnemonic statements in the instruction set, macro mnemonic statements have operand specifiers and addressing modes. So, you can use them as if they were native statements. In this example, it is as if `@CHARI` is an instruction in the instruction set that transfers a character directly from the input port to main memory.

The assembly language listing shows the macro expansion. In Figure 5.8, the listing shows the macro as a comment, followed by the two statements of

Operation	Macro	Macro Expansion
Character input	`@CHARI` OprndSpec,AddrMode	`LDBA charIn,d` `STBA` OprndSpec,AddrMode
Character output	`@CHARO` OprndSpec,AddrMode	`LDBA` OprndSpec,AddrMode `STBA charOut,d`

Figure 5.9 The `@CHARI` and `@CHARO` macro expansions.

the expansion. The expansion statements are the ones translated by the assembler and executed at runtime. The `@CHARO` macro is similar, but transfers a character from memory to the output port.

Figure 5.9 shows the general macro expansions for `@CHARI` and `@CHARO`. The `@CHARI` macro always uses direct addressing to transfer the character from the input port to the accumulator. It uses the addressing mode supplied by the macro call to transfer the character from the accumulator to memory. In the macro expansion of Figure 5.8, the programmer wrote the macro with direct addressing. Therefore, the macro expansion uses direct addressing for the `STBA` expansion.

Immediate Addressing

With direct addressing, the operand specifier is the address in main memory of the operand. Mathematically,

Oprnd = Mem [OprndSpec]

But with immediate addressing, the operand specifier *is* the operand:

Oprnd = OprndSpec

An instruction that uses direct addressing contains the address of the operand. But an instruction that uses immediate addressing contains the operand itself.

Figure 5.10 shows how to write the program in Figure 4.20 (page 77) with immediate addressing. It outputs the message `Hi`. The assembler translates the load byte instruction

```
LDBA 'H',i
```

into object code D00048 (hex), which is 1101 0000 0000 0000 0100 1000 in binary. A check of Figure 5.1 verifies that 1101 0 is the correct opcode for the `LDBA` instruction. Also, the addressing-aaa field is 000 (bin), which indicates immediate addressing. As Figure 5.2 shows, the `i` specifies immediate addressing.

Character constants are enclosed in single quotes and always generate one byte of code. In the program of Figure 5.10, the character constant is placed

Assembly Language Source Code

```
@CHARO  'H',i       ;Output the H character
@CHARO  'i',i       ;Output the i character
STBA    pwrOff,d    ;Store to power off port
```

Assembly Language Listing

```
             ;        @CHARO  'H',i       ;Output the H character
0000  D00048          LDBA    'H',i
0003  F1FFFE          STBA    charOut,d
             ;        End @CHARO
             ;        @CHARO  'i',i       ;Output the i character
0006  D00069          LDBA    'i',i
0009  F1FFFE          STBA    charOut,d
             ;        End @CHARO
000C  F1FFFF          STBA    pwrOff,d    ;Store to power off port
```

Figure 5.10 A program to output `Hi` using immediate addressing.

in the operand specifier, which occupies two bytes. In this case, the character constant is positioned in the rightmost byte of the two-byte word. That is how the assembler translates the statement to binary. At run time, because the addressing mode is immediate, the CPU interprets 0048 as the operand itself—not the address of the operand—and puts 48 immediately in the accumulator.

What if the programmer makes a mistake and for the second line of code writes

```
@CHARO  'i',d       ;Output the i character
```

using direct addressing instead of immediate addressing? This is not an assembly language error. The first instruction in the macro expansion would be

```
LDBA    'i',d
```

which the assembler would translate to D10069. When that line of code executes, the CPU would interpret 0069 as an address, and it would instruct main memory to put Mem[0069] on the bus to be loaded into the accumulator. The output of the program would be erroneous, but difficult to predict.

5.2 An Assembler at Level 5

The features of assembly language described thus far make programming more convenient than programming in machine language. Mnemonics are easier to remember than hexadecimal codes. Symbols are beneficial because you do not need to calculate the memory address of the data. Macros let you write one line of code instead of the multiple lines of code in the macro expansion.

These features are all available on a machine in bare metal mode, that is, a machine without an operating system. This section describes the operating system, and how assembly language can benefit even further by using the services provided by it.

The Operating System

An operating system is a program that manages applications and supplies services to them. Most operating systems manage multiple applications concurrently. The Pep/10 operating system is simplified in order to teach computer systems concepts. As such, it only manages a single application.

Figure 5.11 is the Pep/10 memory map in full OS mode. The operating system resides at the bottom of memory, and the application—including the program instructions and the application's data—resides at the top of memory. The input port, output port, and power off port are parts of the operating system, and in the same locations at the bottom of memory as they are in the bare metal mode. This chapter describes *what* the Dispatcher and the System calls parts of the Pep/10 operating system do. Chapter 8 presents the details of *how* they do it.

In full OS mode, the application is no longer a standalone program. Instead, it is a *function* that the OS calls. Specifically, the dispatcher is that component of the operating system that calls the application as a function. In general when a function terminates, control passes back to the caller. Because the application is a function called by the dispatcher, when the application terminates it must return control back to the dispatcher.

In bare metal mode, the last statement of an assembly language program is to store a byte to the power off port, which shuts down the system. In full OS mode, the last statement of an assembly language program is to execute the monadic `RET` statement, which returns control back to the dispatcher. The program in Figure 5.12 shows how to use the `RET` statement to terminate an assembly language program in full OS mode. Chapter 6 presents the details of how the `RET` statement operates.

The application program produces the output `Hi`. Then it returns control to the dispatcher. If the dispatcher detects a return error, it prints out an error message.

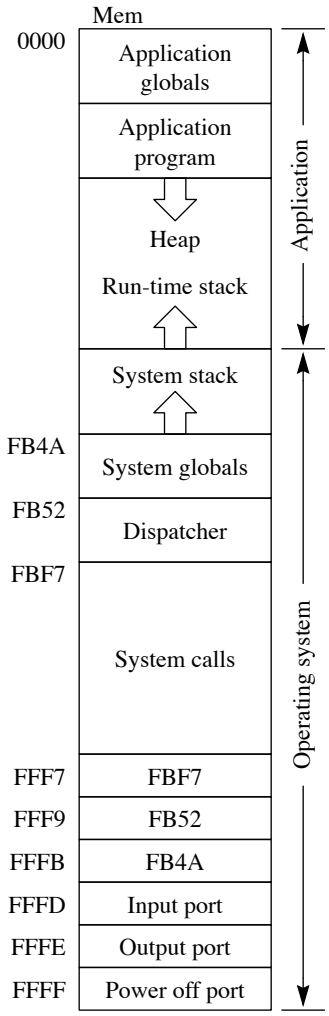


Figure 5.11 The Pep/10 memory map in full OS mode.

The `SCALL` Trap Instruction

Although the assembly language programs at the ISA3 level in the previous section are more convenient to write than the machine language programs in the previous chapter, there are still some major inconveniences.

The program in Figure 5.12 sends a message, one character at a time, to the output port. More convenient would be a function that outputs an arbitrarily long string of characters.

Assembly Language Source Code

```
@CHARO  'H',i       ;Output the H character
@CHARO  'i',i       ;Output the i character
RET
```

Assembly Language Listing

```
             ;        @CHARO  'H',i       ;Output the H character
0000  D00048          LDBA    'H',i
0003  F1FFFE          STBA    charOut,d
             ;        End @CHARO
             ;        @CHARO  'i',i       ;Output the i character
0006  D00069          LDBA    'i',i
0009  F1FFFE          STBA    charOut,d
             ;        End @CHARO
000C  01              RET
```

Figure 5.12 A program in full OS mode to output `Hi`.

The program in Figure 5.6 to input and add two integers is limited to single-digit values, because the input port is an ASCII device. We need a decimal input function that converts a string of ASCII characters from the input port into the two's complement representation of the integer value. Going the other way, we need a decimal output function that converts the two's complement representation of a signed integer and streams the corresponding string of ASCII characters to the output port.

At the HOL6 level of abstraction, the programming language provides these capabilities directly. At the Asmb5 level of abstraction, assembly language does not provide these capabilities at all. Instead, the operating system provides them through the mechanism of a *system call* via the `SCALL` trap instruction.

The Pep/10 operating system provides the following system calls.

- `DECI` Decimal input
- `DECO` Decimal output
- `HEXO` Hexadecimal output
- `STRO` String output
- `SNOP` System call no operation

Each of the above mnemonics is a symbol. However, unlike the symbols presented thus far, the value of each of these symbols is not a memory address. Figure 5.13 is the symbol table for the system calls provided by the Pep/10 operating system. The values number the symbols sequentially starting with 0000 for `DECI` up to 0004 for `SNOP`.

Symbol	Value
`DECI`	0000
`DECO`	0001
`HEXO`	0002
`STRO`	0003
`SNOP`	0004

Figure 5.13 Symbol table for the `SCALL` system calls.

To invoke a system call follow this two-step protocol:

Assembly Language Source Code

```
         LDWA    STRO,i
         SCALL   msg,d
         RET
msg:     .ASCII  "Hello, world!\n\0"
```

Assembly Language Listing

```
0000  C00003          LDWA    STRO,i
0003  390007          SCALL   msg,d
0006  01              RET
0007  48656C msg:     .ASCII  "Hello, world!\n\0"
      6C6F2C
      20776F
      726C64
      210A00
```

Output

```
Hello, world!
```

Figure 5.14 The `STRO` system call.

- Load the symbol for your system call into the accumulator using immediate addressing.
- Execute `SCALL` with an appropriate operand specifier and addressing mode of your choice.

The appropriate operand specifier and addressing mode in the second step, depends on the system call you choose and how you want to use it.

The `STRO` System Call

Figure 5.14 shows how to use the system call protocol to output a string of characters. The string is stored at the bottom of the program after the return statement. As with the C language, you must terminate the string with the NULL character `\0` at the end of the string.

The first step of the protocol is to load `STRO` into the accumulator using immediate addressing. The machine code for this instruction is C00003, with an operand specifier of 0003, which is the value of the symbol `STRO`. Note that 0003 is not a memory address.

The second step of the protocol is to execute `SCALL`, in this case with operand specifier `msg` and direct addressing. The machine code for this instruction is 390007, with an operand specifier of 0007, which is the value of the symbol `msg`. Note that 0007 is the address of the first byte of the string that is output.

In the second step of the protocol, the application is calling a function provided by the operating system. In a HOL6 language like C, when you call a function you can supply a parameter list, which the function accesses to perform its computation. In assembly language, when you execute a system call the operand specifier and the addressing mode are, in effect, the parameter list that the operating system accesses to perform its computation.

The mechanism by which the operating system accesses the operand specifier and the addressing mode is called a *trap*, and `SCALL` is known as a *trap instruction*. The trap mechanism makes a copy of all the registers of the CPU, known as a *process control block* (PCB), and stores the copy in OS memory.

Figure 5.15 shows the process control block for the system call in Figure 5.14. The `SCALL` instruction stores the content of all the registers in the CPU—the instruction register, the stack pointer, the program counter, the index register, the accumulator, and the NZVC status bits—in the PCB.

Recall the steps of the von Neumann cycle—fetch, decode, increment, execute, repeat. The PCB is created at the execution part of the cycle. So, execution of `SCALL` occurs *after* the fetch, decode, and increment parts of the cycle. Therefore, in Figure 5.15:

- The instruction register contains 390007, because the `SCALL` instruction was fetched during the fetch part of the cycle.
- The program counter contains 0006, because PC was incremented by three during the increment part of the cycle.
- The accumulator contains 0003, because the previous instruction loaded the value of the symbol `STRO` into it.

The figure shows the other registers as blank. They actually have values in them that are copied from the CPU. However, it is impossible to know their values from this program.

The code in the operating system can inspect Mem[FB3F] and determine from the 0003 that the call is for the string output operation. It can inspect Mem[FB47] and determine the operand specifier and addressing mode from the 390007. This trap mechanism is how the `SCALL` operand specifier and addressing mode are used, in effect, like parameters in a function call.

After performing the string output operation, the operating system executes the system return instruction `SRET`, which copies the values from the PCB back into the registers of the CPU. One of those registers is the program counter. Figure 5.15 shows 0006, the incremented value of PC, stored in the PCB. Because 0006 is the address of the instruction following the `SCALL` instruction, the instruction following `SCALL` is the one that executes on returning from the operating system call.

FB3E		NZVC
FB3F	0003	A
FB41		X
FB43	0006	PC
FB45		SP
FB47	390007	IR

Figure 5.15 The PCB for the system call in Figure 5.14.

The `DECI` and `DECO` System Calls

Figure 5.16 is the assembler listing of a program that inputs the width and height of a rectangle and computes its perimeter. The program allocates storage for

Assembly Language Listing

```
0000  C00000          LDWA    DECI,i      ;Input width
0003  390047          SCALL   width,d
0006  C00000          LDWA    DECI,i      ;Input height
0009  390049          SCALL   height,d
000C  C10047          LDWA    width,d     ;Compute perimeter of rectangle
000F  510049          ADDA    height,d
0012  1A              ASLA
0013  E1004B          STWA    perim,d
0016  C00003          LDWA    STRO,i      ;Output "width: "
0019  39004D          SCALL   msg1,d
001C  C00001          LDWA    DECO,i      ;Output width
001F  390047          SCALL   width,d
0022  D0000A          LDBA    '\n',i      ;Output newline character
0025  F1FFFE          STBA    charOut,d
0028  C00003          LDWA    STRO,i      ;Output "height: "
002B  390055          SCALL   msg2,d
002E  C00001          LDWA    DECO,i      ;Output height
0031  390049          SCALL   height,d
0034  D0000A          LDBA    '\n',i      ;Output newline character
0037  F1FFFE          STBA    charOut,d
003A  C00003          LDWA    STRO,i      ;Output "perimeter: "
003D  39005E          SCALL   msg3,d
0040  C00001          LDWA    DECO,i      ;Output perimeter
0043  39004B          SCALL   perim,d
0046  01              RET
```

Figure 5.16 A program to input the width and height of a rectangle and compute its perimeter. (*Continues*)

each of these integers in the code immediately following the `RET` instruction.

```
0047  0000   width:   .BLOCK  2
0049  0000   height:  .BLOCK  2
004B  0000   perim:   .BLOCK  2
```

`width` is a symbol whose value is 0047, which is the address of the first byte of the block that stores the integer value of the width of the rectangle. Figure 5.17 shows the values for the other symbols the program defines.

The first two lines of code

```
0000  C00000          LDWA    DECI,i      ;Input width
0003  390047          SCALL   width,d
```

implement the two steps of the decimal input system call protocol. The first line of code loads the `DECI` symbol into the accumulator using immediate ad-


```
0047  0000   width:   .BLOCK  2
0049  0000   height:  .BLOCK  2
004B  0000   perim:   .BLOCK  2
004D  776964 msg1:    .ASCII  "width: \0"
      74683A
      2000
0055  686569 msg2:    .ASCII  "height: \0"
      676874
      3A2000
005E  706572 msg3:    .ASCII  "perimeter: \0"
      696D65
      746572
      3A2000
```

Input

```
24  17
```

Output

```
width: 24
height: 17
perimeter: 82
```

Figure 5.16 (*Continued*) A program to input the width and height of a rectangle and compute its perimeter.

Symbol	Value
`width`	0047
`height`	0049
`perim`	004B
`msg1`	004D
`msg2`	0055
`msg3`	005E

Figure 5.17 Symbol table for the symbols defined in the program of Figure 5.16.

dressing. The second line of code executes the `SCALL` instruction with operand specifier `width` and direct addressing. If the input stream stream is

```
24  17
```

the decimal input system call consumes the ASCII characters `24`, converts them to the equivalent binary value 0018 (hex), and stores that value at Mem[[0047].

Similarly, the next two lines of code consume the ASCII characters `17` from the input stream, converts them to the equivalent binary value 0021 (hex), and stores that value at Mem[0049].

The next four lines of code

```
000C  C10047          LDWA    width,d     ;Compute perimeter of rectangle
000F  510049          ADDA    height,d
0012  1A              ASLA
0013  E1004B          STWA    perim,d
```

compute the perimeter of the rectangle as

$$(\text{width} + \text{height}) \times 2$$

Assembly Language Listing

```
0000  C00001          LDWA    DECO,i
0003  390013          SCALL   num,d
0006  D00020          LDBA    ' ',i
0009  F1FFFE          STBA    charOut,d
000C  C00002          LDWA    HEXO,i
000F  390013          SCALL   num,d
0012  01              RET
0013  FFFD   num:     .WORD   -3
```

Output

```
-3 FFFD
```

Figure 5.18 The `HEXO` system call.

First, the `LDWA` instruction loads the width into the accumulator. The `ADDA` instruction adds the height to the accumulator, leaving the sum in the accumulator. The monadic instruction `ASLA` does an arithmetic shift left on the accumulator, multiplying its value by two. The `STWA` instruction stores the result in the accumulator to Mem[004B], the memory location reserved for the perimeter.

The two lines of code

```
001C  C00001          LDWA    DECO,i      ;Output width
001F  390047          SCALL   width,d
```

implement the two steps of the decimal output system call protocol. The first line of code loads the `DECO` symbol into the accumulator using immediate addressing. The second line of code executes the `SCALL` instruction with operand specifier `width` and direct addressing. The decimal output system call accesses the two bytes at Mem[0047], interprets them as an integer value stored in two's complement binary representation, converts them to the equivalent string of ASCII characters, and sends those characters to the output port.

The `HEXO` System Call

Figure 5.18 shows the operation of the hexadecimal output system call. The program stores –3 at Mem[0013] where is 0013 is the value of symbol `num`. Note that the value stored is FFFD (hex) = 1111 1111 1111 1101 (bin) = –3 (dec). As usual, the value of symbol `num` is the address of the first byte of the memory cell containing the value created by the `.WORD` dot command.

The first two lines of code

```
0000  C00001          LDWA    DECO,i
0003  390013          SCALL   num,d
```


Operation	Macro	Macro Expansion
Decimal input	`@DECI` OprndSpec,AddrMode	`LDWA DECI,i` `SCALL` OprndSpec,AddrMode
Decimal output	`@DECO` OprndSpec,AddrMode	`LDWA DECO,i` `SCALL` OprndSpec,AddrMode
Hexadecimal output	`@HEXO` OprndSpec,AddrMode	`LDWA DECI,i` `SCALL` OprndSpec,AddrMode
String output	`@STRO` OprndSpec,AddrMode	`LDWA STRO,i` `SCALL` OprndSpec,AddrMode
System call no operation	`@SNOP` OprndSpec,AddrMode	`LDWA SNOP,i` `SCALL` OprndSpec,AddrMode

Figure 5.19 The system call macro expansions.

use the `DECO` system call to send the character string `-3` to the output port. The next two lines of code send the space character to the output port.

The two lines of code

```
000C  C00002          LDWA    HEXO,i
000F  390013          SCALL   num,d
```

implement the hexadecimal output system call protocol. The `HEXO` system call works just like the `DECO` system call, except that the `HEXO` system call interprets each of the next four binary nybbles as a single hexadecimal character, and sends the four hexadecimal characters to the output port.

System Call Macros

The system call protocol requires two lines of code to invoke the call. Fortunately, the Pep/10 operating system provides a macro for each of the system calls so you can invoke them with a single line of code. Figure 5.19 shows the macro expansion for each of the five supplied system calls provided by the operating system, and Figure 5.20 shows how to use them.

The benefit of system macros is that you can invoke a system call as if it were a native instruction in the instruction set of the CPU. For example, consider the line of source code

```
@DECO   perim,d     ;Output perimeter
```

from Figure 5.20. It looks like an instruction with mnemonic `@DECO`, operand specifier `perim`, and addressing mode direct. Recall that with direct addressing, the operand specifier is the *address* of the operand. In this line of code, the value of symbol `perim` is the address of the block of two bytes allocated with the line of source code

Assembly Language Source Code

```
          @DECI   width,d     ;Input width
          @DECI   height,d    ;Input height
          LDWA    width,d     ;Compute perimeter of rectangle
          ADDA    height,d
          ASLA
          STWA    perim,d
          @STRO   msg1,d      ;Output "width: "
          @DECO   width,d     ;Output width
          @CHARO  '\n',i      ;Output newline character
          @STRO   msg2,d      ;Output "height: "
          @DECO   height,d    ;Output height
          @CHARO  '\n',i      ;Output newline character
          @STRO   msg3,d      ;Output "perimeter: "
          @DECO   perim,d     ;Output perimeter
          RET
width:    .BLOCK  2
height:   .BLOCK  2
perim:    .BLOCK  2
msg1:     .ASCII  "width: \0"
msg2:     .ASCII  "height: \0"
msg3:     .ASCII  "perimeter: \0"
```

Figure 5.20 The source code of a program with system call macros that is equivalent to the program in Figure 5.16.

```
perim:    .BLOCK  2
```

Figure 5.21 is the assembler listing showing the expansion of the macros. You can see from these two lines in the listing

```
0043  39004B          SCALL   perim,d
004B  0000   perim:   .BLOCK  2
```

that the value of `perim` is 004B.

Suppose the programmer makes a mistake, and uses immediate addressing instead of direct addressing as follows.

```
@DECO   perim,i     ;Output perimeter
```

Recall that with immediate addressing, the operand specifier *is* the operand. The decimal output system call would output `75` because 004B (hex) = 75 (dec).

Macros in general, and system call macros in particular, are examples of abstraction in computer systems. A salient feature of abstraction is the hiding of detail. You can use `@DECO`, together with its operand specifier and addressing

Assembly Language Listing

```
             ;        @DECI   width,d     ;Input width
0000  C00000          LDWA    DECI,i
0003  390047          SCALL   width,d
             ;        End @DECI
             ;        @DECI   height,d    ;Input height
0006  C00000          LDWA    DECI,i
0009  390049          SCALL   height,d
             ;        End @DECI
000C  C10047          LDWA    width,d     ;Compute perimeter of rectangle
000F  510049          ADDA    height,d
0012  1A              ASLA
0013  E1004B          STWA    perim,d
             ;        @STRO   msg1,d      ;Output "width: "
0016  C00003          LDWA    STRO,i
0019  39004D          SCALL   msg1,d
             ;        End @STRO
             ;        @DECO   width,d     ;Output width
001C  C00001          LDWA    DECO,i
001F  390047          SCALL   width,d
             ;        End @DECO
             ;        @CHARO  '\n',i      ;Output newline character
0022  D0000A          LDBA    '\n',i
0025  F1FFFE          STBA    charOut,d
             ;        End @CHARO
             ;        @STRO   msg2,d      ;Output "height: "
0028  C00003          LDWA    STRO,i
002B  390055          SCALL   msg2,d
             ;        End @STRO
             ;        @DECO   height,d    ;Output height
002E  C00001          LDWA    DECO,i
0031  390049          SCALL   height,d
             ;        End @DECO
```

Figure 5.21 The assembler listing of the program in Figure 5.20 showing the expansion of the system call macros. (*Continues*)

mode, as if it were a native CPU instruction. The details of the macro expansion and the processing done by the CPU are hidden from the source code.

5.3 The Assembly Process

This section describes the assembly process, which depends on the following five *von Neumann architecture* principles. These architecture principles follow


```
             ;         @CHARO  '\n',i      ;Output newline character
0034  D0000A           LDBA    '\n',i
0037  F1FFFE           STBA    charOut,d
             ;         End @CHARO
             ;         @STRO   msg3,d      ;Output "perimeter: "
003A  C00003           LDWA    STRO,i
003D  39005E           SCALL   msg3,d
             ;         End @STRO
             ;         @DECO   perim,d     ;Output perimeter
0040  C00001           LDWA    DECO,i
0043  39004B           SCALL   perim,d
             ;         End @DECO
0046  01               RET
0047  0000   width:    .BLOCK  2
0049  0000   height:   .BLOCK  2
004B  0000   perim:    .BLOCK  2
004D  776964 msg1:     .ASCII  "width: \0"
      74683A
      2000
0055  686569 msg2:     .ASCII  "height: \0"
      676874
      3A2000
005E  706572 msg3:     .ASCII  "perimeter: \0"
      696D65
      746572
      3A2000
```

Figure 5.21 (*Continued*) The assembler listing of the program in Figure 5.20 showing the expansion of the system call macros.

from the von Neumann execution cycle—fetch, decode, increment, execute, repeat.

- Main memory stores both instructions and data in binary.
- The decode part of the von Neumann cycle interprets the bits fetched from memory into the IR as a CPU instruction.
- The execute part of the von Neumann cycle interprets the bits processed as data.
- Main memory does not distinguish instruction bits from data bits.
- The CPU can only execute machine language programs. It cannot execute assembly language programs.

The von Neumann architecture is flexible. It allows programs to use other programs as data to perform many useful tasks. For example, the operating system manages all the applications that are running on the computer. From the

point of view of the operating system, those programs are data. One disadvantage of the architecture is that it allows malicious applications like computer viruses to infect other programs.

In contrast to the von Neumann architecture, the *Harvard architecture* maintains separate memories for instructions and data, each with its own separate bus. This architecture makes it difficult to deploy computer viruses.

Because flexibility is so important for general purpose computing, practically all physical CPUs on the market are von Neumann machines. Many virtual machines use the Harvard architecture. For example, WebAssembly (WASM) is a virtual machine designed to provide a secure execution environment for code, primarily for web applications. Even though Pep/10 is a virtual machine, it simulates the von Neumann architecture.

Assemblers

In the early days, programmers wrote in assembly language and then translated each statement into machine language by hand. The translation part was straightforward. It only involved looking up the binary opcodes for the instructions and the binary codes for the ASCII characters in the ASCII table. The hexadecimal operands could similarly be converted to binary with hexadecimal conversion tables. Only after the program was translated could it be loaded and executed.

The translation of a long program was a routine and tedious job. Soon programmers realized that a computer program could be written to do the translation. Such a program is called an assembler. Figure 5.22 illustrates how it functions by translating a program from level Asmb5 to level ISA3.

A computer program solves a problem by accepting input, processing it, and producing output. An assembler is a program whose input is an assembly language program and whose output is that same program translated into machine language in a format suitable for a loader. Input to the assembler is called the *source* program. Output from the assembler is called the *object* program.

Figure 5.23 shows the effect of the Pep/10 assembler on the program of Figure 5.4 to input two characters and output them in reverse order. From the point of view of the assembler, the application program is data to be processed.

An assembler merely translates a program into a format suitable for a loader. It does not execute the program. Translation and execution are separate processes, and translation always occurs first. Figure 5.24 shows the two computer runs necessary for execution of the program to swap the two characters from the input port. The first run is the translation run, and is identical to the run of Figure 5.23. The output of the first run is a machine language program that is executed by the CPU in the second run.

Because the assembler is itself a program, it must be written in some programming language. The computer pioneers who wrote the first assemblers had to write them in machine language. Or, if they wrote them in assembly

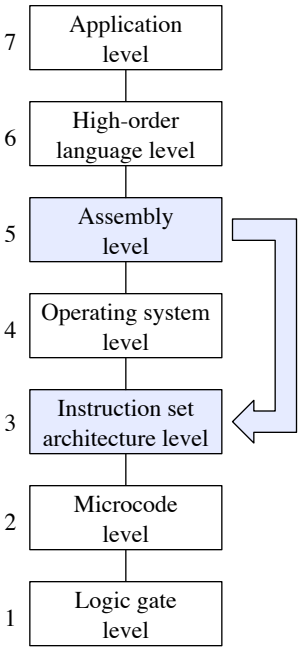


Figure 5.22 The function of an assembler.

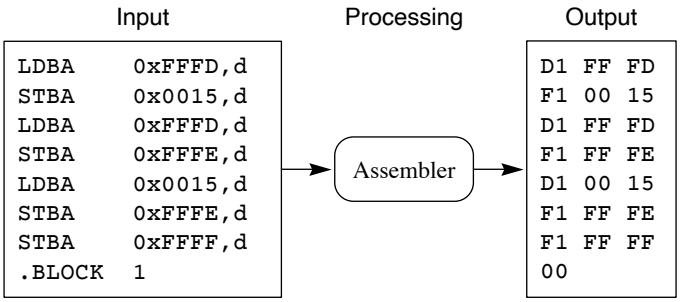


Figure 5.23 The action of an assembler on the program in Figure 5.4 to input two characters and output them in reverse order.

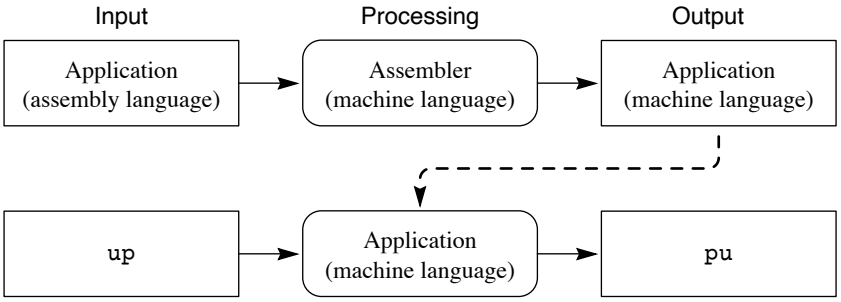


Figure 5.24 Two computer runs necessary for execution of the program in Figure 5.4 to input two characters and output them in reverse order.

language, they had to translate them into machine language by hand because no assemblers were available at the time.

The point is that a machine can execute only programs that are written in machine language.

Cross Assemblers

Machines built by one manufacturer generally have different instruction sets from those in machines built by another manufacturer. Hence, a program in machine language for one brand of computer will not run on a machine with a different brand.

If you write an application in assembly language for a personal computer, you will probably assemble it on the same computer. An assembler written in the same machine language as the language to which it translates is called a *resident assembler*. The assembler resides on the same machine as the application program. Assuming a residence assembler, the two runs of Figure 5.24 would be on the same machine.

However, it is possible for the assembler to be written in Brand X machine language but to translate the application program into Brand Y machine language for a different machine. Then the application program cannot be executed on the same machine on which it is translated. It must first be downloaded from Brand X machine to Brand Y machine.

A *cross assembler* is an assembler that produces an object program for a machine different from the one that runs the assembler. Brand X is called the *host* machine, and Brand Y is called the *target* machine. In Figure 5.24, the first run would be on the host, and the second run would be on the target.

This situation often occurs when the target machine is a small special-purpose computer, such as a mobile device like a watch or a phone. Assemblers are large programs that require significant main memory, as well as input and output peripheral devices. The processor that controls a mobile device has a small main memory. Because it has no input/output (I/O) files, it cannot be used to run an assembler for itself. Its operating system and application programs must be downloaded from a larger host machine that has previously assembled them into the target language.

Interpreting Instructions and Data

Figure 5.25 is a program that illustrates the interpretation of data bits. The fifth von Neumann architecture principle states that the CPU can only execute machine language programs. It cannot execute assembly language programs. Therefore, the principle requires us to consider the translated program.

Here is the translation of the last four statements of the assembly language program and the corresponding symbol table.

```
0031  FFFE   first:   .WORD   0xFFFE
0033  00     second:  .BYTE   0x00
0034  55     third:   .BYTE   'U'
0035  7001   fourth:  .WORD   28673
```

Symbol	Value
`first`	0031
`second`	0033
`third`	0034
`fourth`	0035

In the program, `first` is generated as a hexadecimal value with

```
0031  FFFE   first:   .WORD   0xFFFE
```

but is interpreted as a decimal number with

```
@DECO   first,d     ;Interpret first as dec
```

which outputs `-2`.

Of course, if the programmer meant for the bit pattern FFFE to be interpreted as a decimal number, he would have written the pseudo-op

```
0031  FFFE   first:   .WORD   -2
```

in the first place. This pseudo-op generates the same object code, and the object program would be identical to the original. When `@DECO` executes, it does not

Assembly Language Source Code

```
        @DECO   first,d     ;Interpret first as dec
        @CHARO  ' ',i
        @DECO   second,d    ;Interpret second and third as dec
        @CHARO  ' ',i
        @HEXO   second,d    ;Interpret second and third as hex
        @CHARO  ' ',i
        @CHARO  third,d     ;Interpret third as char
        @CHARO  fourth,d    ;Interpret fourth as char
        RET
first:  .WORD   0xFFFE
second: .BYTE   0x00
third:  .BYTE   'U'
fourth: .WORD   28673
```

Output

```
-2 85 0055 Up
```

Figure 5.25 A program that illustrates the interpretation of data bits.

know how the bits were generated during translation time. It only knows what they are during execution time.

The above analysis is an example of the third Von Neumann architecture principle, which states that the execute part of the von Neumann cycle interprets the bits processed as data. If you think of the `@DECO` instruction abstractly, as if it were a native CPU instruction that executes in a single cycle, the execute part of that cycle is the part that interprets the bits processed as data.

The decimal output instruction

```
@DECO   second,d    ;Interpret second and third as dec
```

interprets the bits at address 0033 as a decimal number and outputs `85`. The `@DECO` instruction always outputs the decimal value of two consecutive bytes. In this case, the bytes are 0055 (hex) = 85 (dec). The fact that the two bytes were generated from two different `.BYTE` dot commands and that one was generated from the hexadecimal constant 0x00 and the other from the character constant `'U'` is irrelevant. During execution, the only thing that matters is what the bits are, not where they came from.

The hexadecimal output instruction

```
@HEXO   second,d    ;Interpret second and third as hex
```

interprets the two bytes beginning at address 0033 as four hexadecimal digits and outputs them with no space between them as `0055`. Again, it does not matter what pseudo-op created the bits. If the `@HEXO` instruction were to output from address 0034, it would print `5570` instead of `0055`.

Assembly Language Source Code

```
here:    @DECO   here,d
         RET
```

Assembly Language Listing

```
             ;here:   @DECO   here,d
0000  C00001 here:    LDWA    DECO,i
0003  390000          SCALL   here,d
             ;        End @DECO
0006  01              RET
```

Output

```
-16384
```

Figure 5.26 A program that illustrates the interpretation of instruction bits as data.

The instruction

```
@CHARO  third,d     ;Interpret third as char
```

interprets the bits at address 0034 as a character. There is no surprise here, because those bits were generated with the `.BYTE` command using a character constant. As expected, the letter `U` is output.

The last instruction

```
@CHARO  fourth,d    ;Interpret fourth as char
```

outputs the letter `p`. Why? Because the bits at memory location 0035 are 70 (hex), which are the bits for the ASCII character `p`. Where did those bits come from? They are the first half of the bits that were generated by

```
0035  7001   fourth:  .WORD   28673
```

It just so happens that 28673 (dec) = 7001 (hex) and the first byte of that bit pattern is 70 (hex).

In the above example, the instructions simply grind through the von Neumann execution cycle. The translation process is different from the execution process, and translation happens before execution. After translation, when the instructions are executing, the origin of the bits is irrelevant. The only thing that matters is what the bits are, not where they came from during the translation phase.

Figure 5.26 is a program that illustrates the interpretation of instruction bits as data. The program assembles and executes without error. It produces an output of `-16384`, which may seem puzzling.

To determine the output, examine the macro expansion in the listing. The first line of the listing reproduces the source line as a comment. The second line

reproduces the definition of the symbol `here`, and associates it with the first line of executable code in the macro expansion. The value of symbol `here` is 0000 because that is the address of the `LDWA` instruction.

The fourth von Neumann architecture principle states that main memory does not distinguish instruction bits from data bits. The `@DECO` system call interprets the two bytes of its operand as a decimal integer stored in two's complement binary representation, and outputs the corresponding stream of ASCII characters. The operand specifier of `@DECO` is 0000. The addressing mode is direct. So, the operand is the two bytes stored at address 0000.

These two bytes are C000 (hex) and were inserted into the object code by the assembler during the translation phase as part of the `LDWA` instruction. The above principle says that it is irrelevant where those bits come from during translation. Now, during execution, they are interpreted as data. Furthermore, C000 (hex) = 1100 0000 0000 0000 (bin) = $-16,384$.

The program in Figure 5.26 interprets bits that are generated as instructions during translation time, as data. The program in Figure 5.27 interprets bits that are generated as data during translation time, as instructions.

It might appear that there is an error in the assembly language source code. The first line of code with `LDWA` and the fourth line of code with `STWA` are valid assembly language statements. However, the second line of code and the third line of code are pseudo-ops, which are used to insert data bits into the object program. In spite of this apparent error in the source code, the assembly language program is perfectly legal and produces an output of `32`.

To determine why the program outputs `32`, consider the assembly language listing. The first dot command `.BYTE` generates `61` because 97 (dec) = 61 (hex). The second dot command `.WORD` generates `0012`. So, the machine language translation of the first four lines of source code is

```
C10010
610012
E10014
```

The second von Neumann architecture principle states that the decode part of the von Neumann cycle interprets the bits fetched from memory into the IR as a CPU instruction. Therefore, after the C10010 instruction executes, the CPU will *fetch* the 61 and *decode* it as the instruction specifier of a CPU instruction. Because 61 (hex) = 0110 0 001 (bin), the CPU interprets 0110 as the opcode, 0 as the register-r field, and 001 as the addressing mode.

Figure 5.1 shows that the 0110 opcode is the subtract instruction, Figure 4.9(c) shows that the 0 register-r field indicates the accumulator, and Figure 4.9(a) shows that the 001 addressing-aaa field indicates direct addressing. So, the object code is the same that would be produced by the mnemonic instruction

```
SUBA    0x0012,d
```

Note that 0x0012 is the value of symbol `num_2`, which is the address of the cell that stores 15.

Assembly Language Source Code

```
         LDWA    num_1,d
         .BYTE   97
         .WORD   0x0012
         STWA    num_3,d
         @DECO   num_3,d
         RET
num_1:   .WORD   47
num_2:   .WORD   15
num_3:   .BLOCK  2
```

Assembly Language Listing

```
0000  C10010          LDWA    num_1,d
0003  61              .BYTE   97
0004  0012            .WORD   0x0012
0006  E10014          STWA    num_3,d
             ;        @DECO   num_3,d
0009  C00001          LDWA    DECO,i
000C  390014          SCALL   num_3,d
             ;        End @DECO
000F  01              RET
0010  002F   num_1:   .WORD   47
0012  000F   num_2:   .WORD   15
0014  0000   num_3:   .BLOCK  2
```

Output

```
32
```

Figure 5.27 A program that illustrates the interpretation of data bits as instructions.

The first three executable statements perform the following computation.

```
C10010 ;Load 47 into the accumulator, A <- 47
610012 ;Subtract 15 from the accumulator, A <- 32
E10014 ;Store 32 to Mem[0014]
```

Consequently, the `@DECO` system call outputs `32`.

To summarize, the programs in Figures 5.25, 5.26, and 5.27 illustrate the von Neumann architecture principles as follows.

Figure 5.25:

The third principle states that the *execute* part of the von Neumann cycle interprets the bits processed as data. Considering each system call to be a native CPU instruction, the *execute* part of the instruction interprets the bits processed as data.

Figure 5.27:
The second principle states that the *decode* part of the von Neumann cycle interprets the bits fetched from memory into the IR as a CPU instruction. The bits inserted into the object code by the `.BYTE` dot command are programmed as if they are data. However, because they are transferred to the IR in the fetch part of the von Neumann cycle, the *decode* part of the cycle interprets them as an instruction.

Figure 5.26:
The bits inserted into the object code by the `LDWA` instruction are interpreted twice. First, when the CPU fetches and then *decodes* them it interprets them as an instruction. Second, when the CPU *executes* `SCALL` it interprets them as data.

The programs in all three figures illustrate the first, fourth, and fifth von Neumann architecture principles. Namely:

- Main memory stores both instructions and data in binary.
- Main memory does not distinguish instruction bits from data bits.
- The CPU can only execute machine language programs. It cannot execute assembly language programs.

Self-Modifying Programs and Viruses

Figure 5.28 illustrates a curious possibility based on the von Neumann design principle. Notice that the assembly language source starting from the third line of code is

```
          LDWA    num_1,d
there:    SUBA    num_2,d
          STWA    num_3,d
          @DECO   num_3,d
          RET
num_1:    .WORD   243
num_2:    .WORD   7
num_3:    .BLOCK  2
```

It appears that the program loads 243 into the accumulator, subtracts 7 from 243 leaving 236 in the accumulator, stores 236 in `num_3`, then outputs `236`. However, the output is `247`. Why?

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 itself. The first instruction

```
LDBA    0x81,i
```


Assembly Language Source Code

```
         LDBA    0x81,i
         STBA    there,d
         LDWA    num_1,d
there:   SUBA    num_2,d
         STWA    num_3,d
         @DECO   num_3,d
         RET
num_1:   .WORD   243
num_2:   .WORD   7
num_3:   .BLOCK  2
```

Assembly Language Listing

```
0000  D00081           LDBA    0x0081,i
0003  F10009           STBA    there,d
0006  C10016           LDWA    num_1,d
0009  610018 there:    SUBA    num_2,d
000C  E1001A           STWA    num_3,d
              ;        @DECO   num_3,d
000F  C00001           LDWA    DECO,i
0012  39001A           SCALL   num_3,d
              ;        End @DECO
0015  01               RET
0016  00F3   num_1:    .WORD   243
0018  0007   num_2:    .WORD   7
001A  0000   num_3:    .BLOCK  2
```

Output

```
247
```

Figure 5.28 A program that modifies itself.

loads the byte 81 (hex) into the right half of the accumulator. The second instruction

```
STBA    there,d
```

puts it in Mem[0009]. What was at Mem[0009] before this change? The instruction specifier of the subtract accumulator instruction. Now the bits at Mem[0009] are 1000 0001. When the CPU gets these bits in the fetch part of the von Neumann execution cycle, it decodes the opcode as 1000, the opcode for the bitwise OR to r instruction. The register-r value of 0 indicates the accumulator, and the addressing-aaa value of 001 indicate direct addressing.

Because 243 (dec) = 0000 0000 1111 0011 (bin), and 7 (dec) = 0000 0000 0000 0111 (bin), the computation is carried out as follows.


```
      0000 0000 1111 0011
OR    0000 0000 0000 0111
      ---------------------
      0000 0000 1111 0111
```

But 0000 0000 1111 0111 (bin) = 247 (dec), which is why the output is `247`.

Of course, this is not a very practical program. If you wanted to OR the two numbers, you would simply write the program of Figure 5.28 without the first two lines of source code and with the OR instruction in place of the subtract instruction.

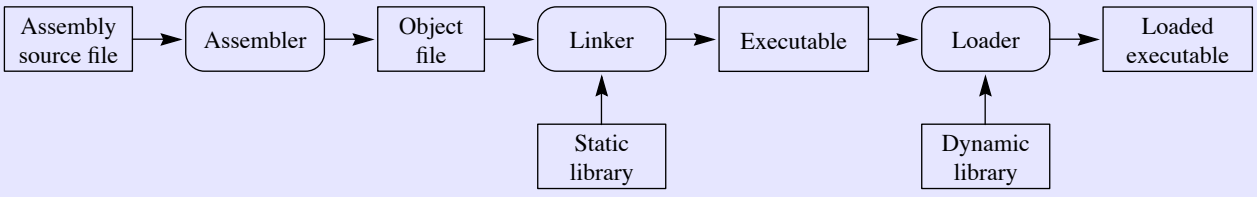
But it does show that in a von Neumann machine, main memory places no significance on the bits it is storing. It simply remembers 1's and 0's and has no idea which are program bits, which are data bits, which are ASCII characters, and so on. Furthermore, the CPU cranks out the von Neumann execution cycle and interprets the bits accordingly—as instructions during the decode part of the cycle and data during the execute part of the cycle. It does not know 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 program of Figure 5.28 modifies itself. A computer *virus* is a program that modifies other programs, including the operating system and other applications. It modifies the opcode of an instruction so that when it is eventually fetched and executed it will perform a malicious operation.

A virus attaches itself to a legitimate program or file, such as a document or email attachment. It exploits vulnerabilities in software or relies on user actions, such as opening an infected file, to gain entry. Once activated, the virus copies itself to other files, programs, or the operating system itself. It may spread via networks, remove all drives, or shared files. The virus executes its malicious payload, which could include corrupting files, stealing data, slowing system performance, or enabling remote control by attackers.

x86 Assembly Language

Figure 5.24 shows two operations necessary to execute an assembly language program—assemble, which translates from assembly language to machine language, followed by execution of the object code. In practice, there are intermediate steps between these two operations—linking and loading—as shown in the following figure.



Like the assembler, the linker is a program that uses another program as data. The linker is necessary if you want your assembly language program to use a previously written module stored in a static library.

For example, it is possible for an assembly language program to call the `printf()` function to send values to the output stream. The code for the function is stored in a static library, and the linker combines a copy of its code in the object file along with the object code from your assembly language program. In a Microsoft system, a static library file has extension `.lib` for library.

The purpose of the loader is to load the object file into main memory. The loader has the additional function of setting up links to the dynamic library, also called the shared library. The idea behind a shared library is to decrease the size of the executable files in the system by not having the code for commonly used libraries duplicated in all the executable files. In a Microsoft system, a dynamic library file has extension `.dll` for dynamic link library.

Programming in assembly language for x86 is complicated by the fact that there are many different incompatible assembly languages for the same x86 instruction set. Also, there are many different incompatible object file formats, depending on the operating system. The following examples compare some Pep/10 assembly language features with those of the Microsoft assembler (MASM) in 32-bit mode available in the Visual Studio IDE.

Here is a code fragment from a Pep/10 assembler listing that allocates storage with some pseudo-ops.

```
0000  FFFE   first:   .WORD 0xFFFE
0002  00     second:  .BYTE 0x00
0003  55     third:   .BYTE 'U'
0004  0470   fourth:  .WORD 1136
0006  000000 fifth:   .BLOCK 4
      00
```

And here is the equivalent code fragment from the MASM listing:

```
00000000              .DATA
00000000  FFFE        first   WORD    0FFFEh
00000002  00          second  BYTE    00h
00000003  55          third   BYTE    'U'
00000004  0470        fourth  WORD    1136
00000006  00000000    fifth   DWORD   ?
```

The data section in a MASM program starts with `.DATA`. There is no dot before the `BYTE` and `WORD` pseudo-ops; nor is there a colon after the definition of a symbol. Hexadecimal constants terminate with the letter h. The leading `0` in `0FFFEh` is necessary to prevent the assembler from interpreting `FFFEh` as a symbol. `DWORD` stands for double word, which is four bytes. `QWORD` stands

for quad word, which is eight bytes. Instead of a separate `.BLOCK` pseudo-op to reserve storage without initialization, a `?` denotes a value of all 0's in a `BYTE`, `WORD`, `DWORD`, or `QWORD`.

Here is a code fragment from a Pep/10 assembly language program.

```
000E  C10003   LDWA num,d   ;A <- the number
0011  500001   ADDA 1,i     ;Add one to it
0014  E10003   STWA num,d   ;Store the sum
```

And here is the equivalent code fragment from the MASM listing.

```
00000000   A1 0000000A   mov eax, num   ;EAX <- the number
00000005   83 C0 01      add eax, 1     ;Add one to it
00000008   A3 0000000A   mov num, eax   ;Store the sum
```

The `mov` mnemonic is used for both load and store. The first argument is always the destination, and the second argument is always the source. In the case of the `add` instruction, the first argument `eax` is the source and the destination. You can see from the listing that the `mov` instructions are five bytes long, and the `add` instruction is three bytes long.

The `mov` instruction can transfer data only between two registers or between a memory cell and a register. It cannot transfer data between two memory cells. In the Pep/10 code, `num` is stored at Mem[0003], while in the MASM code, it is stored at Mem[0000000A].

Written Exercises

Section 5.1

5.1 What is the purpose of the Asmb3 level?

⋆ **5.2** Convert the following machine language instructions into assembly language, assuming that they were not generated by pseudo-ops.

(a) `8AEF2A` (b) `05` (c) `D7003D`

5.3 Convert the following machine language instructions into assembly language, assuming that they were not generated by pseudo-ops.

(a) `72B7DE` (b) `03` (c) `DF63DF`

⋆ **5.4** Convert the following assembly language instructions into hexadecimal machine language.

(a) `ASLA` (b) `@CHARI 0x000F,s`
(c) `BRNE 0x01E6,i`

5.5 Convert the following assembly language instructions into hexadecimal machine language.

(a) `ADDA 0x01FE,i` (b) `@CHARO 0x000D,sf`
(c) `LDWX 0x01FF,s`

⋆ **5.6** Convert the following assembly language pseudo-ops into hexadecimal machine language.

(a) `.ASCII "Bear\0"` (b) `.BYTE 0xF8`
(c) `.WORD 790` (d) `.BLOCK 5`

5.7 Convert the following assembly language pseudo-ops into hexadecimal machine language.

(a) `.BYTE 13` (b) `.ASCII "Frog\0"`
(c) `.WORD -6.` (d) `.BLOCK 7`

5.8 Predict the output of the following Asmb3 assembly language program.

```
LDBA    0x0017,d
STBA    charOut,d
LDBA    0x0016,d
STBA    charOut,d
LDBA    0x0015,d
STBA    charOut,d
STBA    pwrOff,d
.ASCII  "gum"
```

5.9 Predict the output of the following Asmb3 assembly language program.

```
@CHARO  0x0021,d
@CHARO  0x0022,d
@CHARO  0x0023,d
@CHARO  0x0022,d
@CHARO  0x0021,d
STBA    pwrOff,d
.ASCII  "rad"
```

5.10 (a) Predict the output of the following Asmb3 assembly language program if the input is `g`. (b) Predict the output if the input is `A`. (c) Explain the difference between the two results.

```
LDBA charIn,d
ANDA 0x00DF,i
STBA charOut,d
STBA pwrOff,d
```

