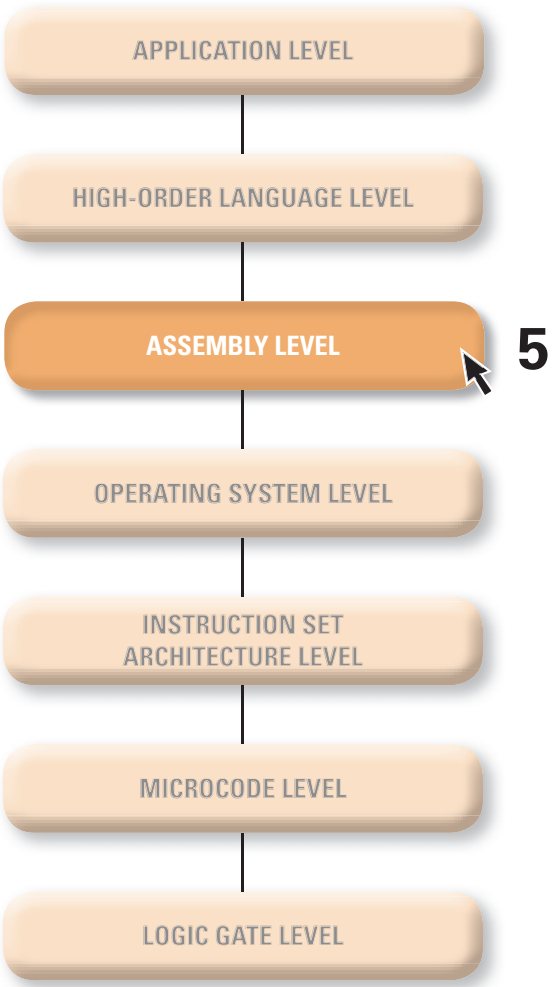
LEVEL

5

Assembly



CHAPTER

5

Assembly Language

TABLE OF CONTENTS

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.

The assembly level uses the operating system below it.

Chapter 4 describes the Pep/9 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 trap handler of the operating system without knowing the details of its operation. That is, you will learn what the trap handler does without learning how the handler does it. Chapter 8 reveals the inner workings of the trap handler.

5.1 Assemblers

The language at Level Asmb5 is called *assembly language*. It provides a more convenient way of writing machine language programs than binary does. The program of Figure 4.33, which outputs `Hi`, contains two types of bit patterns, one for instructions and one for data. These two types are a direct consequence of the von Neumann design, where program and data share the same memory with a binary representation for each.

The two types of bit patterns at Level ISA3

The two types of statements at Level Asmb5

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.

Instruction Mnemonics

Suppose the machine language instruction

```
C0009A
```

is stored at some memory location. This is the load register r instruction. The register-r bit is 0, which indicates the accumulator and not the index register. The addressing-aaa field is 000, which specifies immediate addressing.

This instruction is written in the Pep/9 assembly language as

```
LDWA 0x009A,i
```

A mnemonic for the opcode

The mnemonic `LDWA`, which stands for *load word accumulator*, is written in place of the opcode, 1100, and the register-r field, 0. A mnemonic is a

FIGURE 5.1
The letters that specify the addressing mode in Pep/9 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

memory aid. It is easier to remember that `LDWA` stands for the load word accumulator instruction than to remember that opcode 1100 and register-r 0 stand for the load word accumulator instruction. The operand specifier is written in hexadecimal, `009A`, preceded by `0x`, which stands for *hexadecimal constant*. In Pep/9 assembly language, you specify the addressing mode by placing one or more letters after the operand specifier with a comma between them. FIGURE 5.1 shows the letters that go with each of the eight addressing modes.

Letters for the addressing mode

Example 5.1 Here are some examples of the load word register r instruction written in binary machine language and in assembly language. `LDWX` corresponds to the same machine language statement as `LDWA`, except that the register-r bit for `LDWX` is 1 instead of 0.

```
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
```

FIGURE 5.2 summarizes the 40 instructions of the Pep/9 instruction set at Level Asmb5. It shows the mnemonic that goes with each opcode and the meaning of each instruction. The addressing modes column tells what addressing modes are allowed or whether the instruction is unary (U). The status bits column lists the status bits the instruction affects when it executes.

FIGURE 5.2
The Pep/9 instruction set at Level Asmb5.

Instruction Specifier	Mnemonic	Instruction	Addressing Mode	Status Bits
0000 0000	STOP	Stop execution	U	
0000 0001	RET	Return from CALL	U	
0000 0010	RETTR	Return from trap	U	
0000 0011	MOVSPA	Move SP to A	U	
0000 0100	MOVFLGA	Move NZVC flags to A⟨12..15⟩	U	
0000 0101	MOVAFLG	Move A⟨12..15⟩ to NZVC flags	U	
0000 011r	NOTr	Bitwise invert r	U	NZ
0000 100r	NEGr	Negate r	U	NZV
0000 101r	ASLr	Arithmetic shift left r	U	NZVC
0000 110r	ASRr	Arithmetic shift right r	U	NZC
0000 111r	ROLr	Rotate left r	U	C
0001 000r	RORr	Rotate right r	U	C
0001 001a	BR	Branch unconditional	i, x	
0001 010a	BRLE	Branch if less than or equal to	i, x	
0001 011a	BRLT	Branch if less than	i, x	
0001 100a	BREQ	Branch if equal to	i, x	
0001 101a	BRNE	Branch if not equal to	i, x	
0001 110a	BRGE	Branch if greater than or equal to	i, x	
0001 111a	BRGT	Branch if greater than	i, x	
0010 000a	BRV	Branch if V	i, x	
0010 001a	BRC	Branch if C	i, x	
0010 010a	CALL	Call subroutine	i, x	
0010 011n	NOPn	Unary no operation trap	U	
0010 1aaa	NOP	Nonunary no operation trap	i	

0011 0aaa	DECI	Decimal input trap	d, n, s, sf, x, sx, sfx	NZV
0011 1aaa	DECO	Decimal output trap	i, d, n, s, sf, x, sx, sfx	
0100 0aaa	HEXO	Hexadecimal output trap	i, d, n, s, sf, x, sx, sfx	
0100 1aaa	STRO	String output trap	d, n, s, sf, x	
0101 0aaa	ADDSP	Add to stack pointer (SP)	i, d, n, s, sf, x, sx, sfx	NZVC
0101 1aaa	SUBSP	Subtract from stack pointer (SP)	i, d, n, s, sf, x, sx, sfx	NZVC
0110 raaa	ADDr	Add to r	i, d, n, s, sf, x, sx, sfx	NZVC
0111 raaa	SUBr	Subtract from r	i, d, n, s, sf, x, sx, sfx	NZVC
1000 raaa	ANDr	Bitwise AND to r	i, d, n, s, sf, x, sx, sfx	NZ
1001 raaa	ORr	Bitwise OR 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 5.2 shows the unimplemented opcode instructions replaced by six new instructions:

The unimplemented opcode instructions at Level Asmb5

NOPn	Unary no operation trap
NOP	Nonunary no operation trap
DECI	Decimal input trap
DECO	Decimal output trap
HEXO	Hexadecimal output trap
STRO	String output trap

These new instructions are available to the assembly language programmer at Level Asmb5, but they are not part of the instruction set at Level ISA3. The operating system at Level OS4 provides them with its trap handler. At the assembly level, you may simply program with them as if they were part of the Level ISA3 instruction set, even though they are not. Chapter 8 shows in detail how the operating system provides these

instructions. You do not need to know the details of how the instructions are implemented to program with them.

Pseudo-Operations

Pseudo-operations (pseudo-ops) are assembly language statements. Pseudo-ops do not have opcodes and do not correspond to any of the 40 instructions in the Pep/9 instruction set. Pep/9 assembly language has nine pseudo-ops:

The nine pseudo-ops of Pep/9 assembly language

`.ADDRSS`	The address of a symbol
`.ALIGN`	Padding to align at a memory boundary
`.ASCII`	A string of ASCII bytes
`.BLOCK`	A block of zero bytes
`.BURN`	Initiate ROM burn
`.BYTE`	A byte value
`.END`	The sentinel for the assembler
`.EQUATE`	Equate a symbol to a constant value
`.WORD`	A word value

All the pseudo-ops except `.BURN`, `.END`, and `.EQUATE` insert 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 40 instruction mnemonics. 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 next three programs show how to use the `.ASCII`, `.BLOCK`, `.BYTE`, `.END`, and `.WORD` pseudo-ops. The other pseudo-ops are described later.

The .ASCII and .END Pseudo-ops

The line-oriented nature of assembly language

FIGURE 5.3 is Figure 4.33 written in assembly language instead of machine language. Pep/9 assembly language, unlike C, is line oriented. That is, each assembly language statement must be contained on only one line. You cannot continue a statement onto another line, nor can you place two statements on the same line.

Assembly language comments

Comments begin with a semicolon, `;`, and continue until the end of the line. It is permissible to have a line with only a comment on it, but it must begin with a semicolon. The first four lines of this program are comment lines. The following lines also contain comments, but only after the assembly language statements. As in C, your assembly language programs should contain, at a minimum, your name, the date, and a description of the program. To conserve space in this text, however, the rest of the programs do not contain such a heading.

FIGURE 5.3
An assembly language program to output `Hi`. It is the assembly language version of Figure 4.33.

Assembler Input

```
;Stan Warford
;May 1, 2017
;A program to output "Hi"
;
LDBA      0x000D,d    ;Load byte accumulator 'H'
STBA      0xFC16,d    ;Store byte accumulator output device
LDBA      0x000E,d    ;Load byte accumulator 'i'
STBA      0xFC16,d    ;Store byte accumulator output device
STOP                  ;Stop
.ASCII    "Hi"        ;ASCII "Hi" characters
.END
```

Assembler Output

```
D1 00 0D F1 FC 16 D1 00 0E F1 FC 16 00 48 69 zz
```

Program Output

```
Hi
```

`LDBA` is the mnemonic for the load byte accumulator instruction, and `STBA` is the mnemonic for the store byte accumulator instruction. The statement

```
LDBA 0x000D,d
```

means "Load one byte from Mem[000D] using the direct addressing mode."

The `.ASCII` pseudo-op

The `.ASCII` pseudo-op generates contiguous bytes of ASCII characters. In assembly language, you simply write `.ASCII` followed by a string of ASCII characters enclosed by double quotes. If you want to include a double quote in your string, you must prefix it with a backslash `\`. To include a backslash, prefix it with a backslash. You can put a newline character in your string by prefixing the letter n with a backslash and a tab character by prefixing the letter t with a backslash.

The backslash prefix

Example 5.2 Here is a string that includes two double quotes:

```
"She said, \"Hello\"."
```

Here is one that includes a backslash character:

```
"My bash is \\."
```

And here is one with the newline character:

```
"\nThis sentence will output on a new line."
```

Any arbitrary byte can be included in a string constant using the `\x` feature. When you include `\x` in a string constant, the assembler expects the next two characters to be hexadecimal digits, which specify the byte to be included in the string.

Example 5.3 The dot commands

```
.ASCII "Hello\nworld."
```

and

```
.ASCII "Hello\x0Aworld\x2E"
```

both generate the same sequence of bytes, namely

```
48 65 6C 6C 6F 0A 77 6F 72 6C 64 2E
```

The `.END` *pseudo-op*

You must end your assembly language program with the `.END` command. It does not insert data bits into the program the way the `.ASCII` command does. It simply indicates the end of the assembly language program. The assembler uses `.END` as a sentinel to know when to stop translating.

Assemblers

Compare this program written in assembly language with the same program written in machine language. Assembly language is much easier to understand because of the mnemonics used in place of the opcodes. Also, the characters `H` and `i` written directly as ASCII characters are easier to read.

Unfortunately, you cannot simply write a program in assembly language and expect the computer to understand it. The computer can execute programs only by performing its von Neumann execution cycle (fetch, decode, increment, execute, repeat), which is wired into the CPU. The program must be stored in binary in main memory starting at address 0000 for the execution cycle to process it correctly (as shown in Chapter 4). The assembly language statements must somehow be translated into machine language before they are loaded and executed.

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.

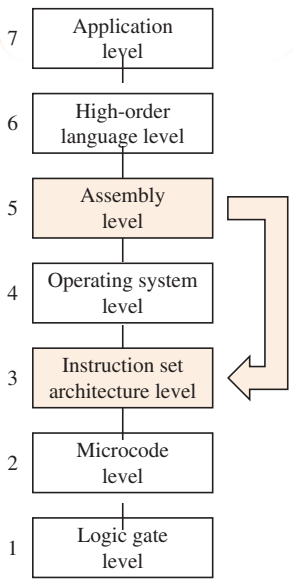
The translation of a long program was a routine and tedious job. Soon programmers realized that a computer program could be written to do the translation. Such a program is called an *assembler*, and FIGURE 5.4 illustrates how it functions.

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 5.5 shows the effect of the Pep/9 assembler on the assembly language of Figure 5.3.

It is important to realize that 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.

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 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.

FIGURE 5.4
The function of an assembler.



The .BLOCK Pseudo-op

FIGURE 5.6 is the assembly language version of Figure 4.35. It inputs two characters and outputs them in reverse order.

FIGURE 5.5
The action of the Pep/9 assembler on the program of Figure 5.3.

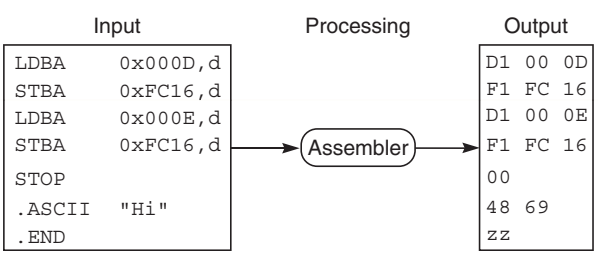


FIGURE 5.6
An assembly language program to input two characters and output them in reverse order. It is the assembly language version of Figure 4.35.

Assembler Input

```
LDBA    0xFC15,d    ;Input first character
STBA    0x0013,d    ;Store first character
LDBA    0xFC15,d    ;Input second character
STBA    0xFC16,d    ;Output second character
LDBA    0x0013,d    ;Load first character
STBA    0xFC16,d    ;Output first character
STOP                ;Stop
.BLOCK  1           ;Storage for first character
.END
```

Assembler Output

```
D1 FC 15 F1 00 13 D1 FC 15 F1 FC 16 D1 00 13 F1
FC 16 00 00 zz
```

Program Input

```
up
```

Program Output

```
pu
```

You can see from the assembler output that the first load statement, `LDBA 0xFC15,d`, translates to D1FC15, and the last store statement, `STBA 0xFC16,d`, translates to F1FC16. After that, the `STOP` statement translates to 00.

The `.BLOCK` pseudo-op generates the next byte of 0's. The dot command

```
.BLOCK 1
```

means "Generate a block of one byte of storage." The assembler interprets any number not prefixed with `0x` as a decimal integer. The digit 1 is therefore interpreted as a decimal integer. The assembler expects a constant after the `.BLOCK` and will generate that number of bytes of storage, setting them to 0's.

The .WORD and .BYTE Pseudo-ops

FIGURE 5.7 is the same as Figure 4.36, computing 5 plus 3. It illustrates the `.WORD` pseudo-op.

Like the `.BLOCK` command, the `.WORD` command generates code for the loader, but with two differences. First, it always generates one

FIGURE 5.7
An assembly language program to add 5 and 3 and output the single-character result. It is the assembly language version of Figure 4.36.

Assembler Input

```
LDWA    0x000D,d    ;A <- first number
ADDA    0x000F,d    ;Add the two numbers
ORA     0x0011,d    ;Convert sum to character
STBA    0xFC16,d    ;Output the character
STOP                ;Stop
.WORD   5           ;Decimal 5
.WORD   3           ;Decimal 3
.WORD   0x0030      ;Mask for ASCII char
.END
```

Assembler Output

```
C1 00 0D 61 00 0F 91 00 11 F1 FC 16 00 00 05 00
03 00 30 zz
```

Program Output

```
8
```

word (two bytes) of code, not an arbitrary number of bytes. Second, the programmer can specify the content of the word. The dot command

```
.WORD 5
```

means "Generate one word with a value of 5 (dec)." The dot command

```
.WORD 0x0030
```

means "Generate one word with a value of 0030 (hex)."

The `.BYTE` command works like the `.WORD` command, except that it generates a byte value instead of a word value. In this program, you could replace

```
.WORD 0x0030
```

with

```
.BYTE 0x00
.BYTE 0x30
```

and generate the same machine language.

You can compare the assembler output of this assembly language program with the hexadecimal machine language of Figure 4.36 to see that they are identical. The assembler was designed to generate output that carefully follows the format expected by the loader. There are no leading blank lines or spaces. There is exactly one space between bytes, with no trailing spaces on a line. The byte sequence terminates with `zz`.

Using the Pep/9 Assembler

Execution of the program in Figure 5.6, the application program that outputs the two input characters in reverse order, requires the computer runs shown in FIGURE 5.8.

First the assembler is loaded into main memory and the application program is taken as the input file. The output from this run is the machine language version of the application program. It is then loaded into main memory for the second run. All the programs in the center boxes must be in machine language.

The Pep/9 system comes with an assembler as well as the simulator. When you execute the assembler, you must provide it with your assembly language program, previously created with the text editor. If you have made no errors in your program, the assembler will generate the object code in a format suitable for the loader. Otherwise, it will protest with one or more error messages and will generate no code. After you generate code from an error-free program, you can use it with the simulator (as described in Chapter 4).

When writing an assembly language program, you must place at least one space after the mnemonic or dot command. Other than that, there are no restrictions on spacing. Your source program may be in any combination of uppercase or lowercase letters. For example, you could write your source of Figure 5.6 as in FIGURE 5.9, and the assembler would accept it as valid and generate the correct code.

FIGURE 5.8
Two computer runs necessary for execution of the program in Figure 5.6.

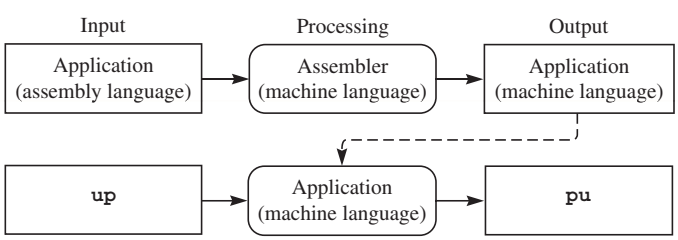


FIGURE 5.9
A valid source program and the resulting assembler listing.

Assembler Input

```
       ldwa 0x000D,d    ;A <- first number
 ADda   0x000F,d  ;Add the two numbers
       ORA    0x0011, d    ;Convert sum to character
  StBA    0Xfc16     ,     d    ;Output the character
       STop    ;Stop
  .WORD 5        ;Decimal 5
.worD     3    ;Decimal 3
    .WORD  0x0030  ;Mask for ASCII char
   .end
```

Assembler Listing

```
------------------------------------------------------------------
      Object
Addr  code      Mnemon  Operand     Comment
------------------------------------------------------------------
0000  C1000D    LDWA    0x000D,d    ;A <- first number
0003  61000F    ADDA    0x000F,d    ;Add the two numbers
0006  910011    ORA     0x0011,d    ;Convert sum to character
0009  F1FC16    STBA    0xFC16,d    ;Output the character
000C  00        STOP                ;Stop
000D  0005      .WORD   5           ;Decimal 5
000F  0003      .WORD   3           ;Decimal 3
0011  0030      .WORD   0x0030      ;Mask for ASCII char
0013            .END
------------------------------------------------------------------
```

In addition to generating object code for the loader, the assembler generates a program listing. The assembler listing converts the source program to a consistent format of uppercase and lowercase letters and spacing. Figure 5.9 shows the assembler listing from the unformatted source program.

The listing also shows the hexadecimal object code that each line generates and the address of the first byte where it will be loaded by the loader. Note that the `.END` command did not generate any object code.

This text presents the remaining assembly language programs as assembler listings, but without the column headings produced by the

assembler, which are shown in the figure. The second column is the machine language object code, and the first column is the address where the loader will place that code in main memory. This layout is typical of most assemblers. It is a vivid presentation of the correspondence between machine language at Level ISA3 and assembly language at Level Asmb5.

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 another machine.

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 language as the language to which it translates is called a *resident assembler*. The assembler resides on the same machine as the application program. The two runs of Figure 5.8 are on the same machine.

Resident assemblers

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 was translated. It must first be moved from Brand X machine to Brand Y machine.

Cross assemblers

A *cross assembler* is an assembler that produces an object program for a machine different from the one that runs the assembler. Moving the machine language version of the application program from the output file of Brand X to the main memory of Brand Y is called *downloading*. Brand X is called the *host machine*, and Brand Y is called the *target machine*. In Figure 5.8, 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 or the computer that controls the cooking cycles in a microwave oven. Assemblers are large programs that require significant main memory, as well as input and output peripheral devices. The processor that controls a microwave oven has a very small main memory. Its input is simply the buttons on the control panel and perhaps the input signal from the temperature probe. Its output includes the digital display and the signals to control the cooking element. Because it has no input/output (I/O) files, it cannot be used to run an assembler for itself. Its program must be downloaded from a larger host machine that has previously assembled the program into the target language.

5.2 Immediate Addressing and the Trap Instructions

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

Oprnd = Mem[OprndSpec]

Direct addressing

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

Oprnd = OprndSpec

Immediate addressing

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

Immediate Addressing

Figure 5.10 shows how to write the program in Figure 5.3 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 5.2 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 5.1 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 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.

Character constants

That is how the assembler translates the statement to binary. But what happens when the loader loads the program and the first instruction executes? If the addressing mode were direct, the CPU would interpret 0048 as an address, and it would instruct main memory to put Mem[0048] on the bus to be loaded into the accumulator. 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 without doing a memory fetch. The third instruction does likewise with 0069.

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

```
0000  D00048    LDBA    'H',i        ;Output 'H'
0003  F1FC16    STBA    0xFC16,d
0006  D00069    LDBA    'i',i        ;Output 'i'
0009  F1FC16    STBA    0xFC16,d
000C  00        STOP
000D            .END

Output
Hi
```

Two advantages of immediate addressing over direct addressing

Immediate addressing has two advantages over direct addressing. The program is shorter because the ASCII string does not need to be stored separately from the instruction. The program in Figure 5.3 has 15 bytes, and this program has 13 bytes. The instruction also executes faster because the operand is immediately available to the CPU in the instruction register. With direct addressing, the CPU must make an additional access to main memory to get the operand.

The DECI, DECO, and BR Instructions

Although the assembly language features we have learned so far are a big improvement over machine language, several irritating aspects remain. They are illustrated in the program of FIGURE 5.11, which inputs a decimal value, adds 1 to it, and outputs the sum.

The first instruction of Figure 5.7,

```
LDWA 0x000D,d ;A <- first number
```

The problem of address computation

puts the content of Mem[000D] into the accumulator. To write this instruction, the programmer had to know that the first number would be stored at address 000D (hex) after the instruction part of the program. The problem with placing the data at the end of the program is that you do not know exactly how long the instruction part of the program will be until you have finished it. Therefore, you do not know the address of the data while writing the instructions that require that address.

Another problem is program modification. Suppose you want to insert an extra statement in your program. That one modification will change the addresses of the data, and every instruction that refers to the data will need to be modified to reflect the new addresses. It would be easier to program at Level Asmb5 if you could place the data at the top of the program. Then you

FIGURE 5.11
A program to input a decimal value, add 1 to it, and output the sum.

```
0000  120005    BR      0x0005      ;Branch around data
0003  0000      .BLOCK  2           ;Storage for one integer
                ;
0005  310003    DECI    0x0003,d    ;Get the number
0008  390003    DECO    0x0003,d    ;and output it
000B  D00020    LDBA    ' ',i       ;Output " + 1 = "
000E  F1FC16    STBA    0xFC16,d
0011  D0002B    LDBA    '+',i
0014  F1FC16    STBA    0xFC16,d
0017  D00020    LDBA    ' ',i
001A  F1FC16    STBA    0xFC16,d
001D  D00031    LDBA    '1',i
0020  F1FC16    STBA    0xFC16,d
0023  D00020    LDBA    ' ',i
0026  F1FC16    STBA    0xFC16,d
0029  D0003D    LDBA    '=',i
002C  F1FC16    STBA    0xFC16,d
002F  D00020    LDBA    ' ',i
0032  F1FC16    STBA    0xFC16,d
0035  C10003    LDWA    0x0003,d    ;A <- the number
0038  600001    ADDA    1,i         ;Add one to it
003B  E10003    STWA    0x0003,d    ;Store the sum
003E  390003    DECO    0x0003,d    ;Output the sum
0041  00        STOP
0042            .END
```

Input

```
-479
```

Output

```
-479 + 1 = -478
```

would know the address of the data when you write a statement that refers to that data.

The problem of restricting numeric operations to a single character

Another irritating aspect of the program in Figure 5.7 is the restriction to single-character results because of the limitations of the output device at Mem[FC16]. Because the device can output only one byte as a single ASCII character, it is difficult to perform I/O on decimal values that require more than one digit for their ASCII representation.

The program in Figure 5.11 alleviates both of these irritations. It is a program to input an integer, add 1 to it, and output the sum. It stores the data at the beginning of the program and permits large decimal values.

When you select the execute option in the Pep/9 simulator, the program counter (PC) gets the value 0000 (hex). The CPU will interpret the bytes at Mem[0000] as the first instruction to execute. To place data at the top of the program, we need an instruction that will cause the CPU to skip the data bytes when it fetches the next instruction. The unconditional branch, BR, is such an instruction. It simply places the operand of the instruction in the PC. In this program,

The unconditional branch, BR

```
BR 0x0005 ;Branch around data
```

places 0005 (hex) in the PC. The RTL specification for the BR instruction is

PC ← Oprnd

During the fetch part of the next execution cycle, the CPU will get the instruction at 0005 instead of 0003, which would have happened if the PC had not been altered.

Because the branch instructions almost always use immediate addressing, the Pep/9 assembler does not require that the addressing mode be specified. If you do not specify the addressing mode for a branch instruction, the assembler will assume immediate addressing and generate 0 for the addressing-a field.

BR defaults to immediate addressing

The correct operation of the BR instruction depends on the details of the von Neumann execution cycle. For example, you may have wondered why the cycle is fetch, decode, *increment*, *execute*, repeat instead of fetch, decode, *execute*, *increment*, repeat. Figure 4.34(f) shows the execution of instruction D1000D to load the byte for H while the value of PC is 0003, the address of instruction F1FC16. If the execute part of the von Neumann execution cycle had been before the increment part, then PC would have had the value 0000 when the instruction at address 0000, which was D1000D, executes. It seems to make more sense to have PC correspond to the *currently executing* instruction instead of the instruction *after* the currently executing one.

The reason increment must come before execute in the von Neumann execution cycle

Why doesn't the von Neumann execution cycle have the execute part before the increment part? Because then BR would not work properly. In Figure 5.11, PC would get 0000; the CPU would fetch the BR instruction, 120005; and BR would execute, placing 0005 in PC. Then PC would increment to 0008. Instead of branching to 0005, your program would branch to 0008. Because the instruction set contains branching instructions, the increment part of the von Neumann execution cycle must be before the execute part.

DECI and DECO are two instructions the operating system provides at the assembly level that the Pep/9 hardware does not provide at the machine

level. DECI, which stands for *decimal input*, converts a sequence of ASCII digit characters to a single word that corresponds to the two's complement representation of the value. DECO, decimal output, does the opposite conversion from the two's complement value in a word to a sequence of ASCII characters.

The DECI *instruction*

DECI permits any number of leading spaces or line feeds on input. The first printable character must be a decimal digit, a +, or a -. The following characters must be decimal digits. DECI sets Z to 1 if you input 0 and N to 1 if you input a negative value. It sets V to 1 if you enter a value that is out of range. Because a word is 16 bits and 2^{16} = 32768, the range is –32768 to 32767 (dec). DECI does not affect the C bit.

The DECO *instruction*

DECO prints a - if the value is negative but does not print + if it is positive. It does not print leading 0's, and it outputs the minimum number of characters possible to properly represent the value. You cannot specify the field width. DECO does not affect the NZVC bits.

In Figure 5.11, the statement

```
DECI 0x0003,d ;Get the number
```

when confronted with input sequence -479, converts it to 1111 1110 0010 0001 (bin) and stores it in Mem[0003]. DECO converts the binary sequence to a string of ASCII characters and outputs them.

The STRO Instruction

You might have noticed that the program in Figure 5.11 requires seven pairs of LDBA and STBA instructions to output the string " + 1 = ", one pair for each ASCII character that is output. The program in FIGURE 5.12 illustrates STRO, which means *string output*. It is another instruction that triggers a trap at the machine level but is a bona fide instruction at the assembly level. It lets you output the entire string of seven characters with only one instruction.

The STRO *instruction*

The operand for STRO is a contiguous sequence of bytes, each one of which is interpreted as an ASCII character. The last byte of the sequence must be a byte of all 0's, which the STRO instruction interprets as the sentinel. The instruction outputs the string of bytes from the beginning up to, but not including, the sentinel. In Figure 5.12, the pseudo-op

```
.ASCII " + 1 = \x00"
```

uses \x00 to generate the sentinel byte. The pseudo-op generates eight bytes including the sentinel, but only seven characters are output by the STRO instruction. Even though you could put the .ASCII pseudo-op at the beginning of the program and branch around it, our coding convention is to always put ASCII strings at the bottom of the program.

FIGURE 5.12
A program identical to that of Figure 5.11 but with the STRO instruction.

```
0000  120005    BR      0x0005      ;Branch around data
0003  0000      .BLOCK  2           ;Storage for one integer
                ;
0005  310003    DECI    0x0003,d    ;Get the number
0008  390003    DECO    0x0003,d    ;and output it
000B  49001B    STRO    0x001B,d    ;Output " + 1 = "
000E  C10003    LDWA    0x0003,d    ;A <- the number
0011  600001    ADDA    1,i         ;Add one to it
0014  E10003    STWA    0x0003,d    ;Store the sum
0017  390003    DECO    0x0003,d    ;Output the sum
001A  00        STOP
001B  202B20    .ASCII  " + 1 = \x00"
      31203D
      2000
0023            .END
```

Input

```
-479
```

Output

```
-479 + 1 = -478
```

The assembler listing allocates room for only three bytes in the object code column. If the string in the .ASCII pseudo-op generates more than three bytes, the assembler listing continues the object code on subsequent lines.

Interpreting Bit Patterns: The HEXO Instruction

Chapters 4 and 5 progress from a low level of abstraction (ISA3) to a higher one (Asmb5). Even though assembly language at Level Asmb5 hides the machine language details, those details are there nonetheless. In particular, the machine is ultimately based on the von Neumann cycle of fetch, decode, increment, execute, repeat. Using pseudo-ops and mnemonics to generate the data bits and instruction bits does not change that property of the machine. When an instruction executes, it executes bits and has no knowledge of how those bits were generated by the assembler. FIGURE 5.13 shows a nonsense program whose sole purpose is to illustrate this fact. It generates data bits with one kind of pseudo-op that are interpreted by instructions in an unexpected way.

FIGURE 5.13
A nonsense program to illustrate the interpretation of bit patterns.

```
0000  120009    BR      0x0009      ;Branch around data
0003  FFFE      .WORD   0xFFFE      ;First
0005  00        .BYTE   0x00        ;Second
0006  55        .BYTE   'U'         ;Third
0007  0470      .WORD   1136        ;Fourth
                ;
0009  390003    DECO    0x0003,d    ;Interpret First as dec
000C  D0000A    LDBA    '\n',i
000F  F1FC16    STBA    0xFC16,d
0012  390005    DECO    0x0005,d    ;Interpret Second and Third as dec
0015  F1FC16    STBA    0xFC16,d
0018  D0000A    LDBA    '\n',i
001B  410005    HEXO    0x0005,d    ;Interpret Second and Third as hex
001E  D0000A    LDBA    '\n',i
0021  F1FC16    STBA    0xFC16,d
0024  D10006    LDBA    0x0006,d    ;Interpret Third as char
0027  F1FC16    STBA    0xFC16,d
002A  D10008    LDBA    0x0008,d    ;Interpret Fourth as char
002D  F1FC16    STBA    0xFC16,d
0030  00        STOP
0031            .END
```

Output

```
-2
85
0055
Up
```

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

```
.WORD 0xFFFE ;First
```

but is interpreted as a decimal number with

```
DECO 0x0003,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

```
.WORD -2 ;First
```

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

The decimal output instruction

```
DECO 0x0005,d ;Interpret Second and Third as dec
```

interprets the bits at address 0005 as a decimal number and outputs 85. DECO 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 0x0005,d ;Interpret Second and Third as hex
```

interprets the two bytes beginning at address 0005 as four hexadecimal digits and outputs them with no space between them. Again, it does not matter what pseudo-op created the bits. If the HEXO instruction were to output from address 0006, it would print 5504 instead of 0055.

The pair of instructions

```
LDBA 0x0006,d ;Interpret Third as char
STBA 0xFC16,d
```

interprets the bits at address 0006 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 pair of instructions

```
LDBA 0x0008,d ;Interpret Fourth as char
STBA 0xFC16,d
```

outputs the letter p. Why? Because the bits at memory location 0008 are 70 (hex), which are the bits for the ASCII character p. Where did those bits come from? They are the second half of the bits that were generated by

```
.WORD 1136 ;Fourth
```

It just so happens that 1136 (dec) = 0470 (hex) and the second byte of that bit pattern is 70 (hex).

In all these examples, the instruction simply grinds through the von Neumann execution cycle. You must always remember that the translation

process is different from the execution process and that 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.

Disassemblers

The one-to-one mapping of an assembler

An assembler translates each assembly language statement into exactly one machine language statement. Such a transformation is called a *one-to-one mapping*. One assembly language statement maps to one machine language statement. This is in contrast to a compiler, which, as we shall see later, produces a *one-to-many mapping*.

Given a single assembly language statement, you can always determine the corresponding machine language statement. But can you do the inverse? That is, given a bit sequence in a machine language program, can you determine the original assembly language statement from which the machine language came?

The nonunique nature of the inverse mapping of an assembler

No, you cannot. Even though the transformation is one-to-one, the inverse transformation is not unique. Given the binary machine language sequence

```
0101 0111
```

you cannot tell if the assembly language programmer originally used an ASCII assembler directive for the ASCII character `W`, or if she wrote the `ADDSP` mnemonic with stack-deferred indexed addressing. The assembler would have produced the exact same sequence of bits regardless of which of these two assembly language statements was in the original program.

Furthermore, during execution, main memory does not know what the original assembly language statements were. It remembers only the 1's and 0's that the CPU processes via its execution cycle.

FIGURE 5.14 shows two assembly language programs that produce the same machine language, and so produce identical output. Of course, a serious programmer would not write the second program because it is more difficult to understand than the first program.

The cause of the nonunique nature of the inverse mapping

Because of pseudo-ops, the inverse assembler mapping is not unique. If there were no pseudo-ops, there would be only one possible way to recover the original assembly language statements from binary object code. Pseudo-ops are for inserting data bits, as opposed to instruction bits, into memory. The fact that data and programs share the same memory is why the inverse assembler mapping is not unique.

The difficulty of recovering the source program from the object program can be a marketing benefit to the software developer. If you write

FIGURE 5.14
Two different source programs that produce the same object program and, therefore, the same output.

Assembly Language Program

```
0000  D10013    LDBA    0x0013,d
0003  F1FC16    STBA    0xFC16,d
0006  D10014    LDBA    0x0014,d
0009  F1FC16    STBA    0xFC16,d
000C  D10015    LDBA    0x0015,d
000F  F1FC16    STBA    0xFC16,d
0012  00        STOP
0013  50756E    .ASCII  "Pun"
0016            .END
```

Assembly Language Program

```
0000  D10013    LDBA    0x0013,d
0003  F1FC16    STBA    0xFC16,d
0006  D10014    LDBA    0x0014,d
0009  F1FC16    STBA    0xFC16,d
000C  D10015    LDBA    0x0015,d
000F  F1FC16    STBA    0xFC16,d
0012  00        STOP
0013  50756E    ADDSP   0x756E,i
0016            .END
```

Program Output

```
Pun
```

The advantage of object code for software distribution

an application in assembly language, there are two ways you can sell it. You can sell the source program and let your customer assemble it. Your customer would then have both the source program and the object program. Or you could assemble it yourself and sell only the object program.

In both cases, the customer has the object program necessary for executing the application program. But if he has the source program as well, he can easily modify it to suit his own purposes. He may even enhance it and then try to sell it as an improved version in direct competition with you, with little effort on his part. Modifying a machine language program would be much more difficult. Most commercial software products are sold only in object form to prevent the customer from tampering with the program.

The open-source software movement is an established development in the computer industry. The idea is that there is a benefit to the customer's having the source program because of support issues. If you own an object program and discover a bug that needs to be fixed or a feature that needs to be added, you must wait for the company who sold you the program to fix the bug or add the feature. But if you own the source, you can modify it yourself to suit your own needs. Some open-source companies give away the source code free of charge and derive their income by providing software support for the product. An example of this strategy is the Linux operating system, which is available for free from the Internet. Although such software is free, it requires a higher level of skill to use.

The advantage of source code for software distribution

A *disassembler* is a program that tries to recover the source program from the object program. It can never be 100% successful because of the nonunique nature of the inverse assembler mapping. The programs in this chapter place the data either before or after the instructions. In a large program, sections of data are typically placed throughout the program, making it difficult to distinguish data bits from instruction bits in the object code. A disassembler can read each byte and print it out several times—once interpreted as an instruction specifier, once interpreted as an ASCII character, once interpreted as an integer with two's complement binary representation, and so on. A person then can attempt to reconstruct the source program, but the process is tedious.

Disassemblers

5.3 Symbols

The previous section introduces `BR` as an instruction to branch around the data at the beginning of the program. Although this technique alleviates the problem of manually determining the address of the data cells, it does not eliminate the problem. You must still determine the addresses by counting in hexadecimal, and if the number of data cells is large, mistakes are likely. Also, if you want to modify the data section, say by removing a `.WORD` command, the addresses of all the data cells following the deletion will change. You must modify any instructions that refer to the modified addresses.

Assembly language symbols eliminate the problem of manually determining addresses. The assembler lets you associate a *symbol*, similar to a C identifier, 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 purpose of assembly language symbols

A Program with Symbols

The assembly language of FIGURE 5.15 produces object code identical to that of Figure 5.12. It uses three symbols, `num`, `msg`, and `main`.

The syntax rules for symbols are similar to the syntax rules for C identifiers. The first character must be a letter, and the following characters must be letters or digits. 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 this program, the statement

```
num: .BLOCK 2 ;Storage for one integer #2d
```

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.

The value of a symbol is an address.

When the assembler detects a symbol definition, it stores the symbol and its value in a symbol table. The value is the address in memory of where the first byte of the object code generated from that line will be loaded. If you define any symbols in your program, the assembler listing will include a printout of the symbol table with the values in hexadecimal. Figure 5.15 shows the symbol table printout from the listing of this program. You can see from the table that the value of the symbol `num` is 0003 (hex).

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

```
LDWA num,d ;A <- the number
```

refers to the symbol `num`. Because `num` has the value 0003 (hex), this statement generates the same code that

```
LDWA 0x0003,d ;A <- the number
```

would generate. Similarly, the statement

```
BR main ;Branch around data
```

generates the same code that

```
BR 0x0005 ;Branch around data
```

would generate, because the value of `main` is 0005 (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[0003] will contain

FIGURE 5.15
A program that adds 1 to a decimal value. It is identical to Figure 5.12 except that it uses symbols.

Assembler Listing

```
-------------------------------------------------------------------------
      Object
Addr  code    Symbol    Mnemon  Operand     Comment
-------------------------------------------------------------------------
0000  120005            BR      main        ;Branch around data
0003  0000    num:      .BLOCK  2           ;Storage for one integer #2d
              ;
0005  310003  main:     DECI    num,d       ;Get the number
0008  390003            DECO    num,d       ;and output it
000B  49001B            STRO    msg,d       ;Output " + 1 = "
000E  C10003            LDWA    num,d       ;A <- the number
0011  600001            ADDA    1,i         ;Add one to it
0014  E10003            STWA    num,d       ;Store the sum
0017  390003            DECO    num,d       ;Output the sum
001A  00                STOP
001B  202B20  msg:      .ASCII  " + 1 = \x00"
      31203D
      2000
0023                    .END
-------------------------------------------------------------------------

Symbol table
---------------------------------------
Symbol    Value         Symbol    Value
---------------------------------------
main      0005          msg       001B
num       0003
---------------------------------------
```

Input

```
-479
```

Output

```
-479 + 1 = -478
```

–479 (dec), which it gets from the input device. The value of `num` will still be 0003 (hex), not –479 (dec), which is different. 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 `0x0003`. Good programmers are careful to select meaningful symbols for their programs to enhance readability.

A von Neumann Illustration

When you program with symbols at Level Asmb5, it is easy to lose sight of the von Neumann nature of the computer. The two classic von Neumann bugs—manipulating instructions as if they were data and attempting to execute data as if they were instructions—are still possible.

For example, consider the following assembly language program:

```
this: DECO this,d
STOP
.END
```

You might think that the assembler would object to the first statement because it appears to be referring to itself as data in a nonsensical way. But the assembler does not look ahead to the ramifications of execution. Because the syntax is correct, it translates accordingly, as shown in the assembler listing in FIGURE 5.16.

During execution, the CPU interprets 39 as the opcode for the decimal output instruction with direct addressing. It interprets the word at Mem[0000], which is 3900 (hex), as a decimal number and outputs its value, 14592.

FIGURE 5.16
A nonsense program that illustrates the underlying von Neumann nature of the machine.

Assembler Listing

```
0000  390000 this:  DECO   this,d
0003  00            STOP
0004                .END
```

Output

```
14592
```

It is important to realize that computer hardware has no innate intelligence or reasoning power. The execution cycle and the instruction set are wired into the CPU. As this program illustrates, the CPU has no knowledge of the history of the bits it processes. It has no overall picture. It simply executes the von Neumann cycle over and over again. The same thing is true of main memory, which has no knowledge of the history of the bits it remembers. It simply stores 1's and 0's as commanded by the CPU. Any intelligence or reasoning power must come from software, which is written by humans.

x86 Assembly Language

Figure 5.8 shows two steps for the Pep/9 system—assemble, which translates from assembly language to machine language, followed by load, which puts the machine language in main memory for execution. FIGURE 5.17 shows an additional step in typical systems called *linking*, which happens after assembly and before loading. Like the assembler and loader, 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 `printf()` 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.

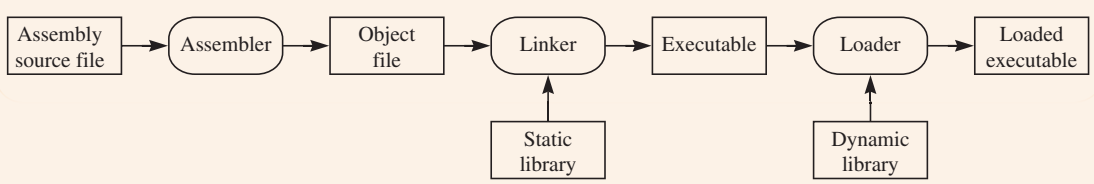
The only function of the Pep/9 loader is to load the object file into main memory. In an actual system, the loader has the additional function of setting up links to the dynamic library, also called the *shared library*. The idea behind a shared library is to decrease the size of the executable files 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/9 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/9 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
```

FIGURE 5.17
Preparation of an assembly language program for execution.



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 the Pep/9 assembler listing for Figure 5.15:

```
000E C10003 LDWA num,d ;A <- the number
0011 600001 ADDA 1,i   ;Add one to it
0014 E10003 STWA num,d ;Store the sum
```

The equivalent code fragment from the MASM listing is at the bottom of this sidebar.

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/9 code, `num` is stored at Mem[0003], while in the MASM code, it is stored at Mem[0000000A].

```
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
```

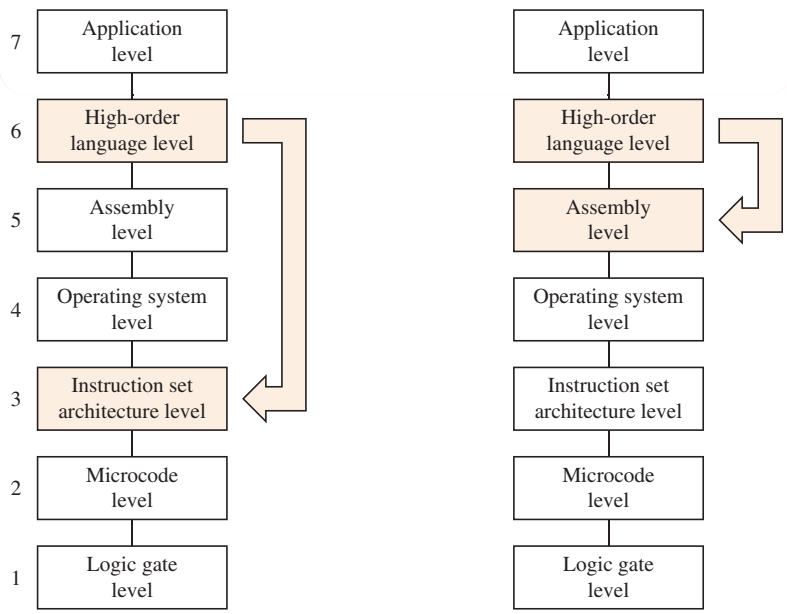
5.4 Translating from Level HOL6

A compiler translates a program in a high-order language (Level HOL6) into a lower-level language, so eventually it can be executed by the machine. Some compilers translate directly into machine language (Level ISA3), as shown in FIGURE 5.18(a). Then the program can be loaded into memory and executed. Other compilers translate into assembly language (Level Asmb5), as shown in Figure 5.18(b). An assembler then must translate the assembly language program into machine language before it can be loaded and executed.

Compilers and assemblers are programs

Like an assembler, a compiler is a program. It must be written and debugged as any other program must be. The input to a compiler is called the *source program*, and the output from a compiler is called the *object program*, whether it is machine language or assembly language. This terminology is identical to that for the input and output of an assembler.

FIGURE 5.18
The function of a compiler.



(a) Translation directly to machine language.

(b) Translation to assembly language.

This section describes the translation process from C to Pep/9 assembly language. It shows how a compiler translates `scanf()`, `printf()`, and assignment statements, and how it enforces the concept of type at the C level. Chapter 6 continues the discussion of the relationship between the high-order language level (Level HOL6) and the assembly level (Level Asmb5).

The Printf() Function

The program in FIGURE 5.19 shows how a compiler would translate a simple C program with one output statement into assembly language.

The compiler translates the single C statement

Translating `printf()`

```
printf("Hello, world!\n");
```

into one executable assembly language statement

```
STRO msg,d
```

and one dot command

FIGURE 5.19
The `printf()` function at Level HOL6 and Level Asmb5.

High-Order Language

```
#include <stdio.h>
int main() {
   printf("Hello, world!\n");
   return 0;
}
```

Assembly Language

```
0000  490004          STRO    msg,d
0003  00              STOP
0004  48656C msg:     .ASCII  "Hello, world!\n\x00"
      6C6F2C
      20776F
      726C64
      210A00
0013                  .END
```

Output

```
Hello, world!
```

```
msg: .ASCII "Hello, world!\n\x00"
```

This is a one-to-two mapping. In contrast to an assembler, the mapping for a compiler generally is not one-to-one, but one-to-many. This program and all the ones that follow place string constants at the bottom of the program. Data that corresponds to variable values is placed at the top of the program to correspond to its placement in the HOL6 program.

Translating `return 0` *in* `main()`

The compiler translates the C statement

```
return 0;
```

into the assembly language statement

```
STOP
```

`return` statements for C functions other than `main()` do not translate to `STOP`. This tranlation of `return` for `main()` is a simplification. A real C compiler must generate code that executes on a particular operating system. It is up to the operating system to interpret the value returned. A common

convention is that a returned value of 0 indicates that no errors occurred during the program's execution. If an error did occur, the program returns some nonzero value, but what happens in such a case depends on the particular operating system. In the Pep/9 system, returning from `main()` corresponds to terminating the program. Hence, returning from `main()` will always translate to `STOP`. Chapter 6 shows how the compiler translates returns from functions other than `main()`.

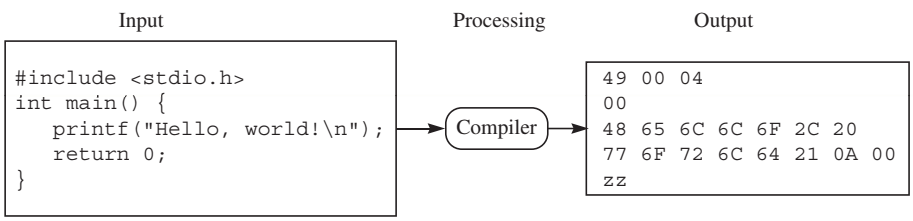
Other elements of the C program are not even translated directly. For example,

```
#include <stdio.h>
```

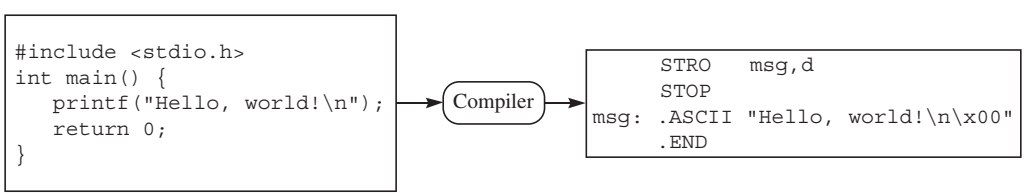
does not appear in the assembly language program at all. A real compiler would use the `#include` statement to make the correct interface to the operating system and its library. The Pep/9 system ignores these kinds of details to keep things simple at the introductory level.

FIGURE 5.20 shows the input and output of a compiler with this program. Part (a) is a compiler that translates directly into machine language. The object program could be loaded and executed. Part (b) is a compiler that translates to assembly language at Level Asmb5. The object program would need to be assembled before it could be loaded and executed.

FIGURE 5.20
The action of a compiler on the program in Figure 5.19.



(a) A compiler that translates directly into machine language.



(b) A compiler that translates into assembly language.

Variables and Types

Every C variable has three attributes—name, type, and value. For each variable that is declared, the compiler reserves one or more memory cells in the machine language program. A variable in a high-order language is simply a memory location in a low-level language. Level-HOL6 programs refer to variables by names, which are C identifiers. Level-ISA3 programs refer to them by addresses. The value of the variable is the value in the memory cell at the address associated with the C identifier.

The symbol table for a compiler

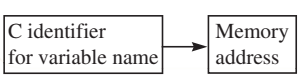
The compiler must remember which address corresponds to which variable name in the Level-HOL6 program. It uses a symbol table to make the connection between variable names and addresses.

The symbol table for a compiler is similar to, but inherently more complicated than, the symbol table for an assembler. A variable name in C is not limited to eight characters, as is a symbol in Pep/9. In addition, the symbol table for a compiler must store the variable's type as well as its associated address.

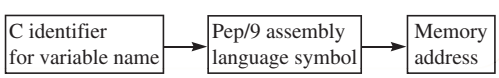
A compiler that translates directly to machine language does not require a second translation with an assembler. FIGURE 5.21(a) shows the mapping produced by the symbol table for such a compiler. The programs in this text illustrate the translation process for a hypothetical compiler that translates to assembly language, however, because assembly language is easier to read than machine language. Variable names in C correspond to symbols in Pep/9 assembly language, as Figure 5.21(b) shows.

The correspondence in Figure 5.21(b) is unrealistic for compilers that translate to assembly language. Consider the problem of a C program that has two variables named `discountRate1` and `discountRate2`. Because they are longer than eight characters, the compiler would have a difficult time mapping the identifiers to unique Pep/9 symbols. Our examples will limit the C identifiers to, at most, eight characters to make clear the correspondence between C and assembly language. Real compilers that translate to assembly language typically do not use assembly language symbols for the variable names.

FIGURE 5.21
The mapping a compiler makes between a Level-HOL6 variable and a Level-ISA3 storage location.



(a) A compiler that translates to machine language.



(b) A hypothetical compiler for illustrative purposes.

Global Variables and Assignment Statements

The C program in FIGURE 5.22 is from Figure 2.4. It shows assignment statements with global variables at Level HOL6 and the corresponding assembly language program, which the compiler produces. The object program contains comments. Real compilers do not generate comments because human programmers usually do not need to read the object program.

The assembly language listing shows two symbols that are used but apparently not defined—`charIn`, used with the `LDBA` instruction, and `charOut`, used with the `STBA` instruction. The assembler automatically includes these two symbols in its symbol table without the programmer needing to define them explicitly. The memory map of the Pep/9 computer in Figure 4.41 shows the input device at Mem[FC15] and the output device at Mem[FC16] within the operating system. Chapter 8 describes the Pep/9 operating system, and Figure 8.2 shows that `charIn` and `charOut` are defined in the operating system assembly language program.

If you modify the operating system, the input device may no longer be at Mem[FC15]. However, its location will still be in the machine vector at FFF8. Similarly, the location of the output device will always be in the machine vector at FFFA. The assembler takes the values of `charIn` and `charOut` from the symbol table of the operating system, which in turn sets up the machine vectors as follows:

Mem[FFF8] has the value of `charIn`
Mem[FFFA] has the value of `charOut`

During execution, the virtual machine uses these vectors to know where the input and output devices are in the memory map. From now on, you should use the symbols `charIn` and `charOut` when accessing the memory-mapped I/O devices, because they will always map to the correct locations in memory regardless of any modifications to the operating system.

Remember that a compiler is a program. It must be written and debugged just like any other program. A compiler to translate C programs can be written in any language—even C! The following program segment illustrates some details of this incestuous state of affairs. It is part of a simplified compiler that translates C source programs into assembly language object programs:

A symbol table definition for a hypothetical compiler

```
typedef int HexDigit;
enum KindType {sInt, sBool, sChar, sFloat};
struct SymbolTableEntry {
   char symbol[32];
   HexDigit value[4];
```

FIGURE 5.22
The assignment statement with global variables at Level HOL6 and Level Asmb5. The C program is from Figure 2.4.

High-Order Language

```
#include <stdio.h>
char ch;
int j;
int main() {
   scanf("%c %d", &ch, &j);
   j += 5;
   ch++;
   printf("%c\n%d\n", ch, j);
   return 0;
}
```

Assembly Language

```
0000  120006          BR      main
0003  00      ch:     .BLOCK  1           ;global variable #1c
0004  0000    j:      .BLOCK  2           ;global variable #2d
              ;
0006  D1FC15  main:   LDBA    charIn,d    ;scanf("%c %d", &ch, &j)
0009  F10003          STBA    ch,d
000C  310004          DECI    j,d
000F  C10004          LDWA    j,d         ;j += 5
0012  600005          ADDA    5,i
0015  E10004          STWA    j,d
0018  D10003          LDBA    ch,d        ;ch++
001B  600001          ADDA    1,i
001E  F10003          STBA    ch,d
0021  D10003          LDBA    ch,d        ;printf("%c\n%d\n", ch, j)
0024  F1FC16          STBA    charOut,d
0027  D0000A          LDBA    '\n',i
002A  F1FC16          STBA    charOut,d
002D  390004          DECO    j,d
0030  D0000A          LDBA    '\n',i
0033  F1FC16          STBA    charOut,d
0036  00              STOP
0037                  .END
```

Input

```
M 419
```

Output

```
N
424
```

```
    KindType kind;
};

SymbolTableEntry symbolTable[100];
```

An entry in a symbol table contains three parts—the symbol itself; its value, which is the address in Pep/9 memory where the value of the variable will be stored; and the kind of value that is stored, that is, the variable's type.

FIGURE 5.23 shows the entries in the symbol table for this program. The first variable has the symbolic name `ch`. The compiler allocates the byte at Mem[0003] by generating the `.BLOCK` command and stores its type as `sChar` in the symbol table, an indication that the variable is a C character. The second variable has the symbolic name `j`. The compiler allocates two bytes at Mem[0004] for its value and stores its type as `sInt`, indicating a C integer. It gets the types from the variable declaration of the C program.

During the code generation phase, the compiler translates

```
scanf("%c %d", &ch, &j);
```

into

```
LDBA 0xFC15,d
STBA 0x0003,d
DECI 0x0004,d
```

It consults the symbol table in Figure 5.23, which was filled at an earlier phase of compilation, to determine the addresses for the operands of the `LDBA` , `STBA`, and `DECI` instructions.

Note that the value stored in the symbol table is not the value of the variable during execution. It is the memory address of where that value will be stored. If the user enters 419 for `j` during execution, then the value stored at Mem[0004] will be 01A3 (hex), which is the binary representation of 419 (dec). The symbol table contains 0004, not 01A3, as the value of the symbol

FIGURE 5.23
The symbol table for a hypothetical compiler that translates the program in Figure 5.22.

	symbol	value	kind
[0]	ch	0003	sChar
[1]	j	0004	sInt
[2]	⋮	⋮	⋮

An assignment statement at Level Asmb5

`j` at translation time. Values of C variables do not exist at translation time. They exist at execution time.

Assigning a value to a variable at Level HOL6 corresponds to storing a value in memory at Level Asmb5. The compiler translates the assignment statement

```
j += 5;
```

into

```
LDWA j,d
ADDA 5,i
STWA j,d
```

where the symbols are shown instead of the addresses. `LDWA` and `ADDA` perform the computation on the right-hand side of the assignment statement, leaving the result of the computation in the accumulator. `STWA` assigns the result back to `j`.

There is a distinction between the word "value" during translation versus during execution. During translation, the value of a symbol is an address. During execution, the value of a variable is the content of a memory cell. This assignment statement illustrates the general rules for accessing global variables:

The rules for accessing global variables

- The value of a symbol for a variable is the address of the variable's value during execution.
- The value of a variable during execution is accessed with direct addressing.

In this case, the value of the symbol for the global variable `j` is the address 0004, and the `LDWA` and `STWA` statements access the value of the variable during execution with direct addressing.

Similarly, the compiler translates

The increment statement at Level Asmb5

```
ch++
```

into

```
LDBA ch,d
ADDA 1,i
STBA ch,d
```

The same instruction that adds 5 to `j`, `ADDA`, performs the increment operation on `ch`. Again, because `ch` is a global variable, its value during translation is the address 0003, and the `LDBA` and `STBA` instructions use direct addressing to access the value of the variable during execution.

The output operator at Level Asmb5

The compiler translates

```
printf("%c\n%d\n", ch, j);
```

into

```
LDBA ch,d
STBA charOut,d
LDBA '\n',i
STBA charOut,d
DECO j,d
LDBA '\n',i
STBA charOut,d
```

using direct addressing to output the values of the global variables `ch` and `j`.

The compiler must search its symbol table to make the connection between a symbol such as `ch` and its address, 0003. The symbol table is an array. If it is not maintained in alphabetic order by symbolic name, a sequential search would be necessary to locate `ch` in the table. If the symbolic names are in alphabetic order, a binary search is possible.

Type Compatibility

To see how type compatibility is enforced at Level HOL6, suppose you have two variables, integer `j` and floating-point `y`, in a C program. Also suppose that you have a computer unlike Pep/9 that is able to store and manipulate floating-point values. The compiler's symbol table for your program might look something like FIGURE 5.24.

Now consider the operation `j % 8` in C. `%` is the modulus operator, which is restricted to operate on integer values. In binary, to perform `j % 8`, you simply set all the bits except the rightmost three bits to 0. For example, if `j` has the value 61 (dec) = 0011 1101 (bin), then `j % 8` has the value 5 (dec)

FIGURE 5.24
The compiler symbol table for a program with a floating-point variable.

	symbol	value	kind
[0]	j	0003	sInt
[1]	y	0005	sFloat
[2]	⋮	⋮	⋮

= 0000 0101 (bin), which is 0011 1101 with all bits except the rightmost three set to 0.

Suppose the following statement appears in your C program:

```
j = j % 8;
```

The compiler would consult the symbol table and determine that `kind` for the variable `j` is `sInt`. It would also recognize 8 as an integer constant and determine that the `%` operation is legal. It would then generate the object code

```
LDWA j,d
ANDA 0x0007,i
STWA j,d
```

Now suppose that the following statement appears in your C program:

Illegal at Level HOL6

```
y = y % 8;
```

The compiler would consult the symbol table and determine that `kind` for the variable `y` is `sFloat`. It would determine that the `%` operation is not legal because it can be applied only to integer types. It would then generate the error message

```
error: float operand for %
```

and would generate no object code. If, however, there were no type checking, the following code would be generated:

Legal at Level Asmb5

```
LDWA y,d
ANDA 0x0007,i
STWA y,d
```

Indeed, there is nothing to prevent an assembly language programmer from writing this code, even though its execution would produce meaningless results.

Type compatibility enforced by the compiler

Having the compiler check for type compatibility is a tremendous help. It keeps you from writing meaningless statements, such as performing a `%` operation on a float variable. When you program directly in assembly language at Level Asmb5, there are no type compatibility checks. All data consists of bits. When bugs occur due to incorrect data movements, they can be detected only at run time, not at translation time. That is, they are logical errors instead of syntax errors. Logical errors are notoriously more difficult to locate than syntax errors.

Pep/9 Symbol Tracer

Pep/9 has three symbolic trace features corresponding to the three parts of the C memory model—the global tracer for global variables, the stack tracer for parameters and local variables, and the heap tracer for dynamically allocated variables. To trace a variable, the programmer embeds trace tags in the comments associated with the variables and single steps through the program. The Pep/9 integrated development environment shows the run-time values of the variables.

There are two kinds of trace tags:

Trace tags

- Format trace tags
- Symbol trace tags

Trace tags are contained in assembly language comments and have no effect on generated object code. Each trace tag begins with the # character and supplies information to the symbol tracer on how to format and label the memory cell in the trace window. Trace tag errors show up as warnings when the code is assembled, allowing program execution without tracing turned on. However, they do prevent tracing until they are corrected.

The global tracer allows the user to specify which global symbol to trace by placing a format trace tag in the comment of the `.BLOCK` line where the global variable is declared. For example, these two lines from Figure 5.22,

```
ch: .BLOCK 1 ;global variable #1c
j:  .BLOCK 2 ;global variable #2d
```

have format trace tags `#1c` and `#2d`. You should read the first format trace tag as "one byte, character." This trace tag tells the symbol tracer to display the content of the one-byte memory cell at the address specified by the value of the symbol, along with the symbol `ch` itself. Similarly, the second trace tag tells the symbol tracer to display the two-byte cell at the address specified by `j` as a decimal integer.

The legal format trace tags are:

The format trace tags

`#1c`	One-byte character
`#1d`	One-byte decimal
`#2d`	Two-byte decimal
`#1h`	One-byte hexadecimal
`#2h`	Two-byte hexadecimal

Global variables do not require the use of symbol trace tags, because the Pep/9 symbol tracer takes the symbol from the `.BLOCK` line on which the trace tag is placed. Local variables, however, require symbol trace tags, which are described in Chapter 6.

The Shift and Rotate Instructions

Pep/9 has two arithmetic shift instructions and two rotate instructions. All four are unary, with the following instruction specifiers, mnemonics, and status bits that they affect:

The shift and rotate instructions

0000 101r	`ASLr`	Arithmetic shift left r	NZVC
0000 110r	`ASRr`	Arithmetic shift right r	NZC
0000 111r	`ROLr`	Rotate left r	C
0001 000r	`RORr`	Rotate right r	C

The shift and rotate instructions have no operand specifier. Each one operates on either the accumulator or the index register, depending on the value of r. A shift left multiplies a signed integer by 2, and a shift right divides a signed integer by 2 (as described in Chapter 3). Rotate left rotates each bit to the left by one bit, sending the most significant bit into C and C into the least significant bit. Rotate right rotates each bit to the right by one bit, sending the least significant bit into C and C into the most significant bit.

The register transfer language (RTL) specification for the `ASLr` instruction is

$$C \leftarrow r\langle 0\rangle \, , \, r\langle 0..14\rangle \, , \leftarrow r\langle 1..15\rangle \, , \, r\langle 15\rangle \leftarrow 0 \, ;$$
$$N \leftarrow r < 0 \, , \, Z \leftarrow r = 0 \, , \, V \leftarrow \{\textit{overflow}\}$$

The RTL specification for the `ASRr` instruction is

$$C \leftarrow r\langle 15\rangle \, , \, r\langle 1..15\rangle \leftarrow r\langle 0..14\rangle \, ; \, N \leftarrow r < 0 \, , \, Z \leftarrow r = 0$$

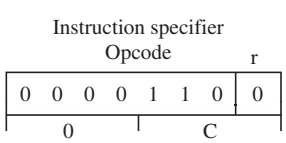
The RTL specification for the `ROLr` instruction is

$$C \leftarrow r\langle 0\rangle \, , \, r\langle 0..14\rangle \leftarrow r\langle 1..15\rangle \, , \, r\langle 15\rangle \leftarrow C$$

The RTL specification for the `RORr` instruction is

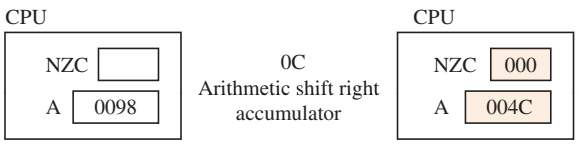
$$C \leftarrow r\langle 15\rangle \, , \, r\langle 1..15\rangle \leftarrow r\langle 0..14\rangle \, , \, r\langle 0\rangle \leftarrow C$$

FIGURE 5.25
The `ASRA` instruction.



Example 5.4 Suppose the instruction to be executed is 0C in hexadecimal, which FIGURE 5.25 shows in binary. The opcode indicates that the `ASRr` instruction will execute, and the register-r field indicates that the instruction will affect the accumulator.

FIGURE 5.26
Execution of the ASRA instruction.



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

FIGURE 5.26 shows the effect of executing the ASRA instruction, assuming the accumulator has an initial content of 0098 (hex) = 152 (dec). The ASRA instruction changes the bit pattern to 004C (hex) = 76 (dec), which is half of 152. The N bit is 0 because the quantity in the accumulator is positive. The Z bit is 0 because the accumulator is not all 0's. The C bit is 0 because the least significant bit was 0 before the shift occurred. ∎

Constants and .EQUATE

.EQUATE is one of the few pseudo-ops to not generate any object code. Furthermore, the normal mechanism of taking the value of a symbol from the address of the object code does not apply. .EQUATE operates as follows:

The operation of .EQUATE

- It must be on a line that defines a symbol.
- It equates the value of the symbol to the value that follows the .EQUATE.
- It does not generate any object code.

The C compiler uses the .EQUATE dot command to translate C constants.

The C program in FIGURE 5.27 is identical to the one in Figure 2.6, except that the variables are global instead of local. It shows how to translate a C constant to machine language. It also illustrates the ASRA assembly language statement. The program calculates a value for score as the average of two exam grades plus a 10-point bonus.

The compiler translates

```
const int bonus = 10;
```

as

```
bonus: .EQUATE 10
```

The assembly language listing in Figure 5.27 is notable on two counts. First, the line that contains the .EQUATE has no code in the machine language column. There is not even an address in the address column because there is no code to which the address would apply. This is consistent

FIGURE 5.27
A program for which the compiler translates a C constant to machine language.

High-Order Language

```
#include <stdio.h>
const int bonus = 10;
int exam1;
int exam2;
int score;

int main() {
   scanf("%d %d", &exam1, &exam2);
   score = (exam1 + exam2) / 2 + bonus;
   printf("score = %d\n", score);
   return 0;
}
```

Assembly Language

```
0000  120009          BR      main
            bonus:    .EQUATE 10          ;constant
0003  0000    exam1:  .BLOCK  2           ;global variable #2d
0005  0000    exam2:  .BLOCK  2           ;global variable #2d
0007  0000    score:  .BLOCK  2           ;global variable #2d
            ;
0009  310003  main:   DECI    exam1,d     ;scanf("%d %d", &exam1,
000C  310005          DECI    exam2,d     ; &exam2)
000F  C10003          LDWA    exam1,d     ;score = (exam1
0012  610005          ADDA    exam2,d     ; + exam2)
0015  0C              ASRA                ; / 2
0016  60000A          ADDA    bonus,i     ; + bonus
0019  E10007          STWA    score,d
001C  490029          STRO    msg,d       ;printf("score = %d\n",
001F  390007          DECO    score,d     ; score)
0022  D0000A          LDBA    '\n',i
0025  F1FC16          STBA    charOut,d
0028  00              STOP
0029  73636F  msg:    .ASCII  "score = \x00"
      726520
      3D2000
0032                  .END
```

```
Symbol table
-------------------------------------
Symbol    Value        Symbol    Value
-------------------------------------
bonus     000A         exam1     0003
exam2     0005         main      0009
msg       0029         score     0007
-------------------------------------
```

Input

```
68 84
```

Output

```
score = 86
```

with the rule that `.EQUATE` does not generate code. Second, Figure 5.27 includes the symbol table from the assembler listing. You can see from the table that symbol `bonus` has the value 000A (hex), which is 10 (dec). In contrast, the symbol `exam1` has the value 5 because the code generated for it by the `.BLOCK` dot command is at address 0005 (hex). But, there is no code for `bonus`, which is set to 000A by the `.EQUATE` dot command.

The I/O and assignment statements are similar to those in previous programs. `scanf()` translates to `DECI`, and `printf()` to `DECO` or `STBA` to `charOut`, all with direct addressing for the global variables. In general, assignment statements translate to

Translating assignment statements

- load register,
- evaluate expression if necessary, and
- store register.

To compute the expression

```
(exam1 + exam2) / 2 + bonus
```

the compiler generates code to load the value of `exam1` into the accumulator, add the value of `exam2` to it, and divide the sum by 2 with the `ASRA` instruction. The `LDWA` and `ADDA` instructions use direct addressing because `exam1` and `exam2` are global variables.

But how does the compiler generate code to add `bonus`? It cannot use direct addressing, because there is no object code corresponding to `bonus`, and hence no address. Instead, the statement

```
ADDA bonus,i
```

uses immediate addressing. In this case, the operand specifier is 000A (hex) = 10 (dec), which is the value to be added. The general rule for translating C constants to assembly language is

Translating C constants

- Declare the constant with `.EQUATE`.
- Access the constant with immediate addressing.

FIGURE 5.28
A translation of the C program in Figure 5.27 with a different placement of instructions and data.

```
0000  310020 main:    DECI    exam1,d     ;scanf("%d %d", &exam1,
0003  310022          DECI    exam2,d     ; &exam2)
0006  C10020          LDWA    exam1,d     ;score = (exam1
0009  610022          ADDA    exam2,d     ; + exam2)
000C  0C              ASRA                ; / 2
000D  60000A          ADDA    bonus,i     ; + bonus
0010  E10024          STWA    score,d
0013  490026          STRO    msg,d       ;printf("score = %d\n",
0016  390024          DECO    score,d     ; score)
0019  D0000A          LDBA    '\n',i
001C  F1FC16          STBA    charOut,d
001F  00              STOP
            ;
            bonus:    .EQUATE 10          ;constant
0020  0000  exam1:    .BLOCK  2           ;global variable #2d
0022  0000  exam2:    .BLOCK  2           ;global variable #2d
0024  0000  score:    .BLOCK  2           ;global variable #2d
0026  73636F msg:     .ASCII  "score = \x00"
      726520
      3D2000
002F                  .END

Symbol table
-------------------------------------
Symbol    Value        Symbol    Value
-------------------------------------
bonus     000A         exam1     0020
exam2     0022         main      0000
msg       0026         score     0024
-------------------------------------
```

In a more realistic program, `score` would have type `float`, and you would compute the average with the real division operator. Pep/9 does not have hardware support for real numbers. Nor does its instruction set contain instructions for multiplying or dividing integers. These operations must be programmed with the shift left and shift right instructions.

Placement of Instructions and Data

The purpose this text is to show the correspondence between the levels of abstraction in a typical computer system. Consequently, the general program structure of an Asmb5 translation corresponds to the structure of the translated HOL6 program. Specifically, global variables appear before the main program in both the Asmb5 program and the HOL6 program. Real compilers do not have that constraint and often alter the placement of programs and data. FIGURE 5.28 is a different translation of the C program in Figure 5.27. One benefit of this translation is the absence of the initial branch to the main program.

Chapter Summary

An assembler is a program that translates a program in assembly language into the equivalent program in machine language. The von Neumann design principle calls for instructions as well as data to be stored in main memory. Corresponding to each of these bit sequences are two types of assembly language statements. For program statements, assembly language uses mnemonics in place of opcodes and register-r fields, hexadecimal instead of binary for the operand specifiers, and mnemonic letters for the addressing modes. For data statements, assembly language uses pseudo-ops, also called *dot commands*.

With direct addressing, the operand specifier is the address in main memory of the operand. But with immediate addressing, the operand specifier is the operand. In mathematical notation, Oprnd = OprndSpec. Immediate addressing is preferable to direct addressing because the operand does not need to be stored separately from the instruction. Such instructions execute faster because the operand is immediately available to the CPU from the instruction register.

Assembly language symbols eliminate the problem of manually determining the addresses of data and instructions in a program. The value of a symbol is an address. When the assembler detects a symbol definition, it stores the symbol and its value in a symbol table. When the symbol is used, the assembler substitutes its value in place of the symbol.

A variable at the high-order language level (Level HOL6) corresponds to a memory location at the assembly level (Level Asmb5). An assignment statement at Level HOL6 that assigns an expression to a variable translates to a load, followed by an expression evaluation, followed by a store at Level Asmb5. Type compatibility at Level HOL6 is enforced by the compiler with the help of its symbol table, which is more complex than the symbol table of an assembler. At Level Asmb5, the only type is bit, and any operation can be performed on any bit pattern.

Exercises

Section 5.1

*1. Convert the following machine language instructions into assembly language, assuming that they were not generated by pseudo-ops:
   (a) 9AEF2A
   (b) 03
   (c) D7003D

2. Convert the following machine language instructions into assembly language, assuming that they were not generated by pseudo-ops:
   (a) 82B7DE
   (b) 04
   (c) DF63DF

*3. Convert the following assembly language instructions into hexadecimal machine language:
   (a) `ASLA`
   (b) `DECI 0x000F,s`
   (c) `BRNE 0x01E6,i`

4. Convert the following assembly language instructions into hexadecimal machine language:
   (a) `ADDA 0x01FE,i`
   (b) `STRO 0x000D,sf`
   (c) `LDWX 0x01FF,s`

*5. Convert the following assembly language pseudo-ops into hexadecimal machine language:
   (a) `.ASCII "Bear\x00"`
   (b) `.BYTE 0xF8`
   (c) `.WORD 790`

6. Convert the following assembly language pseudo-ops into hexadecimal machine language:
(a) `.BYTE 13`
(b) `.ASCII "Frog\x00"`
(c) `.WORD -6`

*__7.__ Predict the output of the following assembly language program:

```
LDBA  0x0015,d
STBA  0xFC16,d
LDBA  0x0014,d
STBA  0xFC16,d
LDBA  0x0013,d
STBA  0xFC16,d
STOP
.ASCII  "gum"
.END
```

8. Predict the output of the following assembly language program:

```
LDBA  0x000E,d
STBA  0xFC16,d
LDBA  0x000D,d
STBA  0xFC16,d
STOP
.ASCII  "is"
.END
```

9. Predict the output of the following assembly language program if the input is `g`. Predict the output if the input is `A`. Explain the difference between the two results:

```
LDBA  0xFC15,d
ANDA  0x000A,d
STBA  0xFC16,d
STOP
.WORD  0x00DF
.END
```

Section 5.2

*__10.__ Predict the output of the program in Figure 5.13 if the dot commands are changed to

```
.WORD 0xFFC7 ;First
.BYTE 0x00   ;Second
.BYTE 'H'    ;Third
.WORD 873    ;Fourth
```

11. Predict the output of the program in Figure 5.13 if the dot commands are changed to

```
.WORD 0xFE63 ;First
.BYTE 0x00   ;Second
.BYTE 'b'    ;Third
.WORD 1401   ;Fourth
```

12. Determine the object code and predict the output of the following assembly language programs:

*(a)

```
DECO  'm',i
LDBA  '\n',i
STBA  0xFC16,d
DECO  "mm",i
LDBA  '\n',i
STBA  0xFC16,d
LDBA  0x0026,i
STBA  0xFC16,d
STOP
.END
```

(b)

```
DECO  'Q',i
LDBA  '\n',i
STBA  0xFC16,d
DECO  0xFFC3,i
LDBA  '\n',i
STBA  0xFC16,d
LDBA  0x007D,i
STBA  0xFC16,d
STOP
.END
```

Section 5.3

***13.** In the following code, determine the values of the symbols `here` and `there`. Write the object code in hexadecimal. (Do not predict the output.)

```
        BR     there
here:   .WORD  9
there:  DECO   here,d
        STOP
        .END
```

14. In the following code, determine the values of the symbols `this`, `that`, and `theOther`. Write the object code in hexadecimal. (Do not predict the output.)

```
           BR     theOther
this:      .WORD  17
that:      .WORD  19
theOther:  DECO   this,d
           DECO   that,d
           STOP
           .END
```

*15. In the following code, determine the value of the symbol `this`. Predict and explain the output of the assembly language program:

```
this:   HEXO    this,d
        STOP
        .END
```

16. In the following code, determine the value of the symbol `this`. Predict and explain the output of the assembly language program:

```
this:   DECO    this,d
        STOP
        .END
```

Section 5.4

17. How are the symbol table of an assembler and a compiler similar? How do they differ?

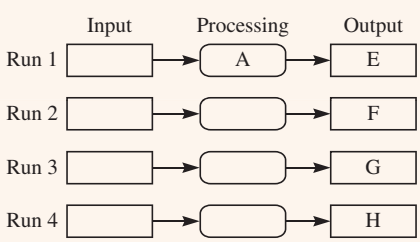
*18. How does a C compiler enforce type compatibility?

19. Assume you have a Pep/9-type computer and the following disk files:

- File A: A Pep/9 assembly language assembler written in machine language
- File B: A C-to-assembly-language compiler written in assembly language
- File C: A C program that will read numbers from a data file and print their median
- File D: A data file for the median program of file C

To compute the median, you must make the four computer runs described schematically in FIGURE 5.29. Each run involves an input file that will be operated on by a program to produce an output file. The output file produced by one run may be used either as the input file or as the program of a subsequent run. Describe the content of files E, F, G, and H, and label the empty blocks in Figure 5.29 with the appropriate file letter.

FIGURE 5.29
The computer runs for Exercise 19.



Problems

Section 5.1

20. Write an assembly language program that prints your first name on the screen. Use the `.ASCII` pseudo-op to store the characters at the bottom of your program. Use the `LDBA` instruction with direct addressing to output the characters from the string. The name you print must contain more than two letters.

Section 5.2

21. Write an assembly language program that prints your first name on the screen. Use immediate addressing with a character constant to designate the operand of `LDBA` for each letter of your name.

22. Write an assembly language program that prints your first name on the screen. Use immediate addressing with a decimal constant to designate the operand of `LDBA` for each letter of your name.

23. Write an assembly language program that prints your first name on the screen. Use immediate addressing with a hexadecimal constant to designate the operand of `LDBA` for each letter of your name.

Section 5.4

The following C programs do not show the `include` statement for `<studio.h>`, which would be required for the programs to compile.

24. Write an assembly language program that corresponds to the following C program:

```
int num1;
int num2;

int main () {
   scanf("%d %d", &num1, &num2);
   printf("%d\n%d\n", num2, num1);
   return 0;
}
```

25. Write an assembly language program that corresponds to the following C program:

```
const char chConst = 'a';
char ch1;
char ch2;
```

```
int main () {
   scanf("%c%c", &ch1, &ch2);
   printf("%c%c%c\n", ch1, chConst, ch2);
   return 0;
}
```

26. Write an assembly language program that corresponds to the following C program:

```
const int amount = 20000;
int num;
int sum;

int main () {
   scanf("%d", &num);
   sum = num + amount;
   printf("sum = %d\n", sum);
   return 0;
}
```

Test your program twice. The first time, enter a value for `num` to make the `sum` within the allowed range for the Pep/9 computer. The second time, enter a value that is in range but that makes `sum` outside the range. Note that the out-of-range condition does not cause an error message but just gives an incorrect value. Explain the value.

27. Write an assembly language program that corresponds to the following C program:

```
int width;
int length;
int perim;

int main () {
   scanf("%d %d", &width, &length);
   perim = (width + length) * 2;
   printf("width = %d\n", width);
   printf("length = %d\n\n", length);
   printf("perim = %d\n", perim);
   return 0;
}
```

28. Write an assembly language program that corresponds to the following C program:

```
char ch;

int main () {
   scanf("%c", &ch);
   ch--;
   printf("%c\n", ch);
   return 0;
}
```

29. Write an assembly language program that corresponds to the following C program:

```
int num1;
int num2;

int main () {
   scanf("%d", &num1);
   num2 = -num1;
   printf("num1 = %d\n", num1);
   printf("num2 = %d\n", num2);
   return 0;
}
```

30. Write an assembly language program that corresponds to the following C program:

```
int num;

int main () {
   scanf("%d", &num);
   num = num / 16;
   printf("num = %d\n", num);
   return 0;
}
```

31. Write an assembly language program that corresponds to the following C program:

```
int num;

int main () {
   scanf("%d", &num);
   num = num % 16;
   printf("num = %d\n", num);
   return 0;
}
```