CHAPTER

6

Compiling to the Assembly Level

TABLE OF CONTENTS

The theme of this text is the application of the concept of levels of abstraction to computer systems. This chapter continues the theme by showing the relationship between the high-order languages level and the assembly level. It examines features of the C language at Level HOL6 and shows how a compiler might translate programs that use those features to the equivalent program at Level Asmb5.

One major difference between Level-HOL6 languages and Level-Asmb5 languages is the absence of extensive data types at Level Asmb5. In C, you can define integers, reals, arrays, Booleans, and structures in almost any combination. But assembly language has only bits and bytes. If you want to define an array of structures in assembly language, you must partition the bits and bytes accordingly. The compiler does that job automatically when you program at Level HOL6.

Another difference between the levels concerns the flow of control. C has `if`, `while`, `do`, `for`, `switch`, and function statements to alter the normal sequential flow of control. You will see that assembly language is limited by the basic von Neumann design to more primitive control statements. This chapter shows how the compiler must combine several primitive Level-Asmb5 control statements to execute a single, more powerful Level-HOL6 control statement.

6.1 Stack Addressing and Local Variables

When a program calls a function, the program allocates storage on the run-time stack for the returned value, the parameters, and the return address. Then the function allocates storage for its local variables. Stack-relative addressing allows the function to access the information that was pushed onto the stack.

You can consider `main()` of a C program to be a function that the operating system calls. You might be familiar with the fact that the main program can have parameters named `argc` and `argv` as follows:

```
int main(int argc, char* argv[])
```

With main declared this way, `argc` and `argv` are pushed onto the run-time stack, along with the return address and any local variables.

A simplification with `main()`

To keep things simple, this text always declares `main()` without the parameters, and it ignores the fact that storage is allocated for the integer returned value and the return address. Hence, the only storage allocated for `main()` on the run-time stack is for local variables. Figure 2.19(a) shows the memory model with the returned value and the return address on the run-time stack. Figure 2.23(a) shows the memory model with this simplification.

Stack-Relative Addressing

With stack-relative addressing, the relation between the operand and the operand specifier is

Stack-relative addressing

Oprnd = Mem[SP + OprndSpec]

The stack pointer acts as a memory address to which the operand specifier is added. Figure 4.41 shows that the user stack grows upward in main memory starting at address FB8F. When an item is pushed onto the run-time stack, its address is smaller than the address of the item that was on the top of the stack.

The stack grows upward in main memory.

You can think of the operand specifier as the offset from the top of the stack. If the operand specifier is 0, the instruction accesses Mem[SP], the value on top of the stack. If the operand specifier is 2, it accesses Mem[SP + 2], the value two bytes below the top of the stack.

The Pep/9 instruction set has two instructions for manipulating the stack pointer directly, `ADDSP` and `SUBSP`. (`CALL`, `RET`, and `RETTR` manipulate the stack pointer indirectly.) `ADDSP` simply adds a value to the stack pointer, and `SUBSP` subtracts a value. The register transfer language (RTL) specification of `ADDSP` is

The `ADDSP` *instruction*

SP ← SP + Oprnd

and the RTL specification of `SUBSP` is

The `SUBSP` *instruction*

SP ← SP – Oprnd

Neither instruction changes the status bits.

Even though you can add to and subtract from the stack pointer, you cannot set the stack pointer with a load instruction. There is no `LDSP` instruction. Then how is the stack pointer ever set? When you select the execute option in the Pep/9 simulator, the following two actions occur:

SP ← Mem[FFF4]
PC ← 0000

The first action sets the stack pointer to the content of memory location FFF4. That location is part of the operating system ROM, and it contains the address of the top of the application's run-time stack. Therefore, when you select the execute option, the stack pointer is initialized correctly. The default Pep/9 operating system initializes the stack pointer to FB8F. The application never needs to set it to anything else. In general, the application only needs to subtract from the stack pointer to push items onto the run-time stack, and add to the stack pointer to pop items off of the run-time stack.

FIGURE 6.1
Stack-relative addressing.

```
0000  D00042   LDBA    'B',i       ;move B to stack
0003  F3FFFF   STBA    -1,s
0006  D0004D   LDBA    'M',i       ;move M to stack
0009  F3FFFE   STBA    -2,s
000C  D00057   LDBA    'W',i       ;move W to stack
000F  F3FFFD   STBA    -3,s
0012  C0014F   LDWA    335,i       ;move 335 to stack
0015  E3FFFB   STWA    -5,s
0018  D00069   LDBA    'i',i       ;move i to stack
001B  F3FFFA   STBA    -6,s
001E  580006   SUBSP   6,i         ;push 6 bytes onto stack
0021  D30005   LDBA    5,s         ;output B
0024  F1FC16   STBA    charOut,d
0027  D30004   LDBA    4,s         ;output M
002A  F1FC16   STBA    charOut,d
002D  D30003   LDBA    3,s         ;output W
0030  F1FC16   STBA    charOut,d
0033  3B0001   DECO    1,s         ;output 335
0036  D30000   LDBA    0,s         ;output i
0039  F1FC16   STBA    charOut,d
003C  500006   ADDSP   6,i         ;pop 6 bytes off stack
003F  00       STOP
0040           .END
```

Output

```
BMW335i
```

Accessing the Run-Time Stack

FIGURE 6.1 shows how to push data onto the stack, access it with stack-relative addressing, and pop it off the stack. The program pushes the string `BMW` onto the stack, followed by the decimal integer 335, followed by the character `'i'`. Then it outputs the items and pops them off the stack.

FIGURE 6.2(a) shows the values in the stack pointer (SP) and main memory before the program executes. The machine initializes SP to FB8F from the vector at Mem[FFF4].

The first two instructions

```
0000 D00042 LDBA 'B',i ;move B to stack
0003 F3FFFF STBA -1,s
```

put an ASCII 'B' character in the byte just above the top of the stack. `LDBA` puts the 'B' byte in the right half of the accumulator, and `STBA` puts it above the stack. The store instruction uses stack-relative addressing with an operand specifier of –1 (dec) = FFFF (hex). Because the stack pointer has the value FB8F, the 'B' is stored at Mem[FB8F + FFFF] = Mem[FB8E]. The next two instructions put 'M' and 'W' at Mem[FB8D] and Mem[FB8C], respectively.

The decimal integer 335, however, occupies two bytes. The program must store it at an address that differs from the address of the 'W' by two. That is why the instruction to store the 335 is

```
0015 E3FFFB STWA -5,s
```

and not `STWA -4,s`. In general, when you push items onto the run-time stack, you must take into account how many bytes each item occupies and set the operand specifier accordingly.

The subtract stack pointer instruction

```
001E 580006 SUBSP 6,i ;push 6 bytes onto stack
```

subtracts 6 from the stack pointer, as Figure 6.2(b) shows. That completes the push operation.

Tracing a program that uses stack-relative addressing does not require you to know the absolute value in the stack pointer. The push operation would work the same if the stack pointer were initialized to some other value, say FA18. In that case, 'B', 'M', 'W', 335, and 'i' would be at Mem[FA17], Mem[FA16], Mem[FA15], Mem[FA13], and Mem[FA12], respectively, and the stack pointer would wind up with a value of FA12. The values would be at the same locations relative to the top of the stack, even though they would be at different absolute memory locations.

FIGURE 6.3 is a more convenient way of tracing the operation and makes use of the fact that the value in the stack pointer is irrelevant. Rather than show the value in the stack pointer, it shows an arrow pointing to the memory cell whose address is contained in the stack pointer. Rather than show the address of the cells in memory, it shows their offsets from the stack pointer. Figures depicting the state of the run-time stack will use this drawing convention from now on.

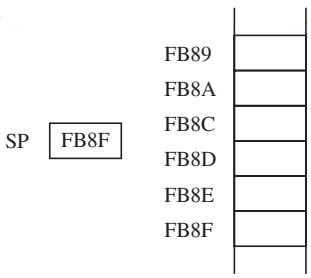
The instruction

```
0021 D30005 LDBA 5,s ;output B
```

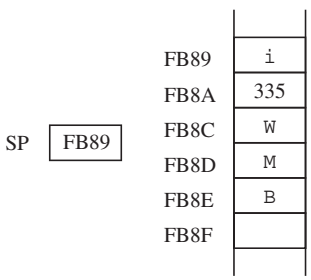
loads the ASCII 'B' character from the stack. Note that the stack-relative address of the 'B' before `SUBSP` executes is –1, but its address after `SUBSP` executes is 5. Its stack-relative address is different because the stack pointer has changed. Both

```
0003 F3FFFF STBA -1,s
```

FIGURE 6.2
Pushing `BMW335i` onto the run-time stack in Figure 6.1.

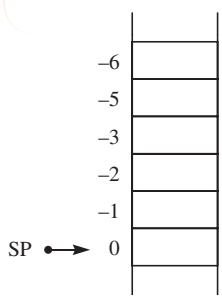


(a) Before the program executes.



(b) After `SUBSP` executes.

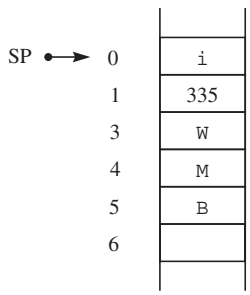
FIGURE 6.3
The stack of Figure 6.2 with relative addresses.



(a) Before the program executes.

(*continues*)

FIGURE 6.3
The stack of Figure 6.2 with relative addresses. (*continued*)



(b) After SUBSP executes.

and

```
0021 D30005 LDBA 5,s ;output B
```

access the same memory cell because SP has a different value when each of them executes. The other items are output similarly using their stack offsets, as shown in Figure 6.3(b).

The instruction

```
003C 500006 ADDSP 6,i ;pop 6 bytes off stack
```

deallocates six bytes of storage from the run-time stack by adding 6 to SP. Because the stack grows upward toward smaller addresses, you push storage by subtracting from the stack pointer, and you pop storage by adding to the stack pointer.

Local Variables

The previous chapter shows how the compiler translates programs with global variables. It allocates storage for a global variable with a .BLOCK dot command and accesses the global variable with direct addressing. Local variables, however, are allocated on the run-time stack. To translate a program with local variables, the compiler

The rules for accessing local variables

- pushes local variables onto the stack with SUBSP,
- accesses local variables with stack-relative addressing, and
- pops local variables off the stack with ADDSP.

The memory model for global versus local variables

An important difference between global and local variables is the time at which the allocation takes place. The .BLOCK dot command is not an executable statement. Storage for global variables is reserved at a fixed location before the program executes. In contrast, the SUBSP statement is executable. Storage for local variables is created on the run-time stack during program execution.

The C program in FIGURE 6.4 is from Figure 2.6. It is identical to the program of Figure 5.27 except that the variables are declared local to main(). Although this difference is not perceptible to the user of the program, the translation performed by the compiler is significantly different. FIGURE 6.5 shows the run-time stack for the program. bonus is a constant and is defined with the .EQUATE command (as in Figure 5.27). However, local variables are also defined with .EQUATE. With a constant, .EQUATE specifies the value of the constant, but with a local variable, .EQUATE specifies the stack offset on the run-time stack. For example, Figure 6.5 shows that the stack offset for local variable exam1 is 4. Therefore, the assembly language program equates the symbol exam1 to 4. Note from the assembly language listing that .EQUATE does not generate any code for the local variables.

.EQUATE specifies the stack offset for a local variable

FIGURE 6.4
A program with local variables. The C program is from Figure 2.6.

High-Order Language

```
#include <stdio.h>

int main() {
   const int bonus = 10;
   int exam1;
   int exam2;
   int score;
   scanf("%d %d", &exam1, &exam2);
   score = (exam1 + exam2) / 2 + bonus;
   printf("score = %d\n", score);
   return 0;
}
```

Assembly Language

```
0000  120003          BR      main
            bonus:    .EQUATE 10          ;constant
            exam1:    .EQUATE 4           ;local variable #2d
            exam2:    .EQUATE 2           ;local variable #2d
            score:    .EQUATE 0           ;local variable #2d
            ;
0003  580006 main:    SUBSP   6,i         ;push #exam1 #exam2 #score
0006  330004          DECI    exam1,s     ;scanf("%d %d", &exam1, &exam2)
0009  330002          DECI    exam2,s
000C  C30004          LDWA    exam1,s     ;score = (exam1 + exam2) / 2 + bonus
000F  630002          ADDA    exam2,s
0012  0C              ASRA
0013  60000A          ADDA    bonus,i
0016  E30000          STWA    score,s
0019  490029          STRO    msg,d       ;printf("score = %d\n", score)
001C  3B0000          DECO    score,s
001F  D0000A          LDBA    '\n',i
0022  F1FC16          STBA    charOut,d
0025  500006          ADDSP   6,i         ;pop #score #exam2 #exam1
0028  00              STOP
0029  73636F msg:     .ASCII  "score = \x00"
      726520
      3D2000
0032                  .END
```

Translation of the executable statements in `main()` differs in two respects from the version with global variables. First, `SUBSP` and `ADDSP` push and pop storage on the run-time stack for the locals. Second, all accesses to the variables use stack-relative addressing instead of direct addressing. Other than these differences, the translation of the assignment and output statements is the same.

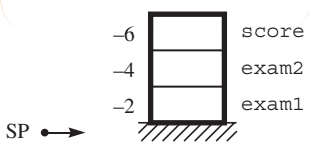
Format trace tags

Figure 6.4 shows how to write trace tags for debugging with local variables. The assembly language program uses the format trace tag `#2d` with the `.EQUATE` pseudo-op to tell the debugger that the values of `exam1`, `exam2`, and `score` should be displayed as two-byte decimal values.

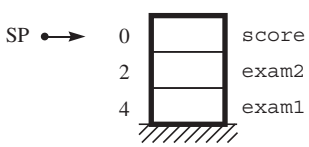
Symbol trace tags

These local variables are pushed onto the run-time stack with the `SUBSP` instruction. Consequently, to debug your program you specify the three symbol trace tags `#exam1`, `#exam2`, and `#score` in the comment for `SUBSP`. When you single-step through the program, the Pep/9 system displays a figure on the screen like that of Figure 6.5(b), with the symbolic labels of the cells on the right of the run-time stack. For the debugger to function accurately, you must list the symbol trace tags in the comment field in the exact order they are pushed onto the run-time stack. In this program, `exam1` is pushed first, followed by `exam2` and then `score`. Furthermore, this order must be consistent with the offset values in the `.EQUATE` pseudo-op.

FIGURE 6.5
The run-time stack for the program of Figure 6.4.



(a) Before `SUBSP` executes.



(b) After `SUBSP` executes.

The variables are popped off the stack with the `ADDSP` instruction. So you must list the variables that are popped off the run-time stack in the proper order. Because the variables are popped off in the opposite order they are pushed on, you list them in the opposite order from the order in the `SUBSP` instruction. In this program, `score` is popped off, followed by `exam2` and then `exam1`.

Although trace tags are not necessary for the program to execute, they serve to document the program. The information provided by the symbol trace tags is valuable for the reader of the program, because it describes the purpose of the `SUBSP` and `ADDSP` instructions. The assembly language programs in this chapter all include trace tags for documentation purposes, and your programs should as well.

6.2 Branching Instructions and Flow of Control

The Pep/9 instruction set has eight conditional branches:

`BRLE`	Branch on less than or equal to
`BRLT`	Branch on less than
`BREQ`	Branch on equal to

`BRNE`	Branch on not equal to
`BRGE`	Branch on greater than or equal to
`BRGT`	Branch on greater than
`BRV`	Branch on V
`BRC`	Branch on C

Each of these conditional branches tests one or two of the four status bits, N, Z, V, and C. If the condition is true, the operand is placed in the program counter (PC), causing the branch. If the condition is not true, the operand is not placed in PC, and the instruction following the conditional branch executes normally. You can think of them as comparing a 16-bit result to 0000 (hex). For example, `BRLT` checks whether a result is less than zero, which happens if N is 1. `BRLE` checks whether a result is less than or equal to zero, which happens if N is 1 or Z is 1. Here is the RTL specification of each conditional branch instruction.

The conditional branch instructions

`BRLE`	$N = 1 \vee Z = 1 \Rightarrow PC \leftarrow Oprnd$
`BRLT`	$N = 1 \Rightarrow PC \leftarrow Oprnd$
`BREQ`	$Z = 1 \Rightarrow PC \leftarrow Oprnd$
`BRNE`	$Z = 0 \Rightarrow PC \leftarrow Oprnd$
`BRGE`	$N = 0 \Rightarrow PC \leftarrow Oprnd$
`BRGT`	$N = 0 \wedge Z = 0 \Rightarrow PC \leftarrow Oprnd$
`BRV`	$V = 1 \Rightarrow PC \leftarrow Oprnd$
`BRC`	$C = 1 \Rightarrow PC \leftarrow Oprnd$

Whether a branch occurs depends on the value of the status bits. The status bits are in turn affected by the execution of other instructions. For example,

```
LDWA num,s
BRLT place
```

causes the content of `num` to be loaded into the accumulator. If the word represents a negative number—that is, if its sign bit is 1—then the N bit is set to 1. `BRLT` tests the N bit and causes a branch to the instruction at `place`. On the other hand, if the word loaded into the accumulator is not negative, then the N bit is cleared to 0. When `BRLT` tests the N bit, the branch does not occur and the instruction after `BRLT` executes next.

Translating the If Statement

FIGURE 6.6 shows how a compiler would translate an `if` statement from C to assembly language. The program computes the absolute value of an integer.

The assembly language comments show the statements that correspond to the high-level program. The `scanf()` function call translates to `DECI`,

FIGURE 6.6
The `if` statement at Level HOL6 and Level Asmb5.

High-Order Language

```
#include <stdio.h>

int main() {
   int number;
   scanf("%d", &number);
   if (number < 0) {
      number = -number;
   }
   printf("%d", number);
   return 0;
}
```

Assembly Language

```
0000  120003          BR      main
            number:   .EQUATE 0           ;local variable #2d
            ;
0003  580002 main:    SUBSP   2,i         ;push #number
0006  330000          DECI    number,s    ;scanf("%d", &number)
0009  C30000 if:      LDWA    number,s    ;if (number < 0)
000C  1C0016          BRGE    endIf
000F  C30000          LDWA    number,s    ;number = -number
0012  08              NEGA
0013  E30000          STWA    number,s
0016  3B0000 endIf:   DECO    number,s    ;printf("%d", number)
0019  500002          ADDSP   2,i         ;pop #number
001C  00              STOP
001D                  .END
```

and the `printf()` function call translates to `DECO`. The assignment statement translates to the sequence `LDWA`, `NEGA`, `STWA`.

The compiler translates the `if` statement into the sequence `LDWA`, `BRGE`. The RTL specification of `LDWr` is

The `LDWr` *instruction*

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

When `LDWA` executes, if the value loaded into the accumulator is positive or zero, the N bit is cleared to 0. That condition calls for skipping the

body of the `if` statement. FIGURE 6.7(a) shows the structure of the `if` statement at Level HOL6. *S1* represents the `scanf()` function call, *C1* represents the condition `number < 0`, *S2* represents the statement `number = -number`, and *S3* represents the statement `printf()` function call. Figure 6.7(b) shows the structure with the more primitive branching instructions at Level Asmb5. The dot following *C1* represents the conditional branch, `BRGE`.

The braces { and } for delimiting a compound statement have no counterpart in assembly language. The sequence

```
Statement 1
if (number >= 0) {
    Statement 2
    Statement 3
}
Statement 4
```

translates to

```
        Statement 1
if:     LDWA number,d
        BRLT endIf
        Statement 2
        Statement 3
endIf:  Statement 4
```

Optimizing Compilers

You may have noticed an extra load statement that was not strictly required in Figure 6.6. You can eliminate the `LDWA` at 000F because the value of `number` will still be in the accumulator from the previous load at 0009.

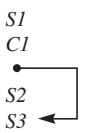
The question is, what would a compiler do? The answer depends on the compiler. A compiler is a program that must be written and debugged. Imagine that you must design a compiler to translate from C to assembly language. When the compiler detects an assignment statement, you program it to generate the following sequence: (a) load accumulator, (b) evaluate expression if necessary, (c) store result to variable. Such a compiler would generate the code of Figure 6.6, with the `LDWA` at 000F.

Imagine how difficult your compiler program would be if you wanted it to eliminate the unnecessary load. When your compiler detected an assignment statement, it would not always generate the initial load. Instead, it would analyze the previous instructions generated and remember the content of the accumulator. If it determined that the value in the accumulator was the same as the value that the initial load put there, it would not generate the initial load.

FIGURE 6.7
The structure of the `if` statement in Figure 6.6.

```
S1
if (C1) {
   S2
}
S3
```

(a) The structure at Level HOL6.



(b) The same structure at Level Asmb5.

In Figure 6.6, the compiler would need to remember that the value of `number` was still in the accumulator from the code generated for the `if` statement.

The purpose of an optimizing compiler

A compiler that expends extra effort to make the object program shorter and faster is called an *optimizing compiler*. You can imagine how much more difficult an optimizing compiler is to design than a nonoptimizing one. Not only are optimizing compilers more difficult to write, they also take longer to compile because they must analyze the source program in much greater detail.

The advantages and disadvantages of an optimizing compiler

Which is better, an optimizing or a nonoptimizing compiler? That depends on the use to which you put the compiler. If you are developing software, a process that requires many compiles for testing and debugging, then you would want a compiler that translates quickly—that is, a nonoptimizing compiler. If you have a large fixed program that will be executed repeatedly by many users, you would want fast execution of the object program—hence, an optimizing compiler. Most compilers offer a wide range of options that allow the developer to specify the level of optimization desired. Software is normally developed and debugged with little optimization and then translated one last time with a high level of optimization for the end users.

The examples in this chapter occasionally present object code that is partially optimized. Most assignment statements, such as the one in Figure 6.6, are presented in nonoptimized form.

Translating the If/Else Statement

FIGURE 6.8 illustrates the translation of the `if/else` statement. The C program is identical to the one in Figure 2.10. The `if` body requires an extra unconditional branch around the `else` body. If the compiler omitted the `BR` at 0015 and the input were 127, the output would be `highlow`.

Unlike Figure 6.6, the `if` statement in Figure 6.8 does not compare a variable's value with zero. It compares the variable's value with another nonzero value using `CPWA`, which stands for *compare word accumulator*. `CPWA` subtracts the operand from the accumulator and sets the NZVC status bits accordingly. `CPWr` is identical to `SUBr` except that `SUBr` stores the result of the subtraction in register r (accumulator or index register), whereas `CPWr` ignores the result of the subtraction. The RTL specification of `CPWr` is

The `CPWr` instruction

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

where T represents a temporary value.

(There is an adjustment to the N bit, $N \leftarrow N \oplus V$, that is not present in the subtract instruction. N is replaced by the exclusive OR of N and V. If the result of the subtraction yields an overflow and the N bit were set as usual, the subsequent conditional branch instruction might execute an erroneous branch. Consequently, if the `CPWr` subtraction operation overflows and sets the V bit,

FIGURE 6.8
The `if/else` statement at Level HOL6 and Level Asmb5. The C program is from Figure 2.10.

High-Order Language

```
#include <stdio.h>

int main() {
   const int limit = 100;
   int num;
   scanf("%d", &num);
   if (num >= limit) {
      printf("high\n");
   }
   else {
      printf("low\n");
   }
   return 0;
}
```

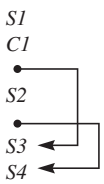
Assembly Language

```
0000  120003          BR      main
            limit:    .EQUATE 100         ;constant
            num:      .EQUATE 0           ;local variable #2d
            ;
0003  580002 main:    SUBSP   2,i         ;push #num
0006  330000          DECI    num,s       ;scanf("%d", &num)
0009  C30000 if:      LDWA    num,s       ;if (num >= limit)
000C  A00064          CPWA    limit,i
000F  160018          BRLT    else
0012  49001F          STRO    msg1,d      ;printf("high\n")
0015  12001B          BR      endIf
0018  490025 else:    STRO    msg2,d      ;printf("low\n")
001B  500002 endIf:   ADDSP   2,i         ;pop #num
001E  00              STOP
001F  686967 msg1:    .ASCII  "high\n\x00"
      680A00
0025  6C6F77 msg2:    .ASCII  "low\n\x00"
      0A00
002A                  .END
```

FIGURE 6.9
The structure of the `if/else` statement in Figure 6.8.

```
S1
if (C1) {
   S2
}
else {
   S3
}
S4
```

(a) The structure at Level HOL6.



(b) The same structure at Level Asmb5.

then the N bit is inverted from its normal value and does not duplicate the sign bit. With this adjustment, the compare operation extends the range of valid comparisons. Even though there is an overflow, the N bit is set as if there were no overflow so that a subsequent conditional branch will operate as expected.)

This program computes `num - limit` and sets the NZVC bits. `BRLT` tests the N bit, which is set if

```
num - limit < 0
```

that is, if

```
num < limit
```

That is the condition under which the `else` part must execute.

FIGURE 6.9 shows the structure of the control statements at the two levels. Part (a) shows the Level-HOL6 control statement, and part (b) shows the Level-Asmb5 translation for this program.

Translating the While Loop

Translating a loop requires branches to previous instructions. FIGURE 6.10 shows the translation of a `while` statement. The C program is identical to the one in Figure 2.13. It echoes ASCII input characters to the output, replacing a space character with a newline character, using the sentinel

FIGURE 6.10
The `while` statement at Level HOL6 and Level Asmb5. The C program is from Figure 2.13.

High-Order Language

```
#include <stdio.h>

char letter;

int main() {
   scanf("%c", &letter);
   while (letter != '*') {
      if (letter == ' ') {
         printf("\n");
      }
      else {
         printf("%c", letter);
      }
      scanf("%c", &letter);
   }
   return 0;
}
```

Assembly Language

```
0000  120004          BR      main
0003  00      letter: .BLOCK  1           ;global variable #1c
              ;
0004  D1FC15  main:   LDBA    charIn,d    ;scanf("%c", &letter)
0007  F10003          STBA    letter,d
000A  D10003  while:  LDBA    letter,d    ;while (letter != '*')
000D  B0002A          CPBA    '*',i
0010  18002E          BREQ    endWh
0013  B00020  if:     CPBA    ' ',i       ;if (letter == ' ')
0016  1A0022          BRNE    else
0019  D0000A          LDBA    '\n',i      ;printf("\n")
001C  F1FC16          STBA    charOut,d
001F  120025          BR      endIf
0022  F1FC16  else:   STBA    charOut,d   ;printf("%c", letter)
0025  D1FC15  endIf:  LDBA    charIn,d    ;scanf("%c", &letter)
0028  F10003          STBA    letter,d
002B  12000A          BR      while
002E  00      endWh:  STOP
002F                  .END
```

technique with * as the sentinel. If the input is `Hello, world!*` on a single line, the output is `Hello`, on one line and `world!` on the next.

The test for a `while` statement is made with a conditional branch at the top of the loop. This program tests a character value, which is a byte quantity. Every `while` loop ends with an unconditional branch to the test at the top of the loop. The unconditional branch at 002B brings control back to the initial test. FIGURE 6.11 shows the structure of the `while` statement at the two levels.

The RTL specification of `CPBr` is

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

where T represents an eight-bit temporary value. The instruction sets the status bits according to the eight-bit value without regard to the high-order byte of register r. The `CPBA` instruction in Figure 6.10(b) would still function correctly even if the accumulator had some 1's in its high-order byte.

The RTL specification of `LDBr` is

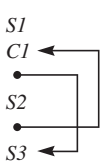
$$r\langle 8..15\rangle \leftarrow \text{byte Oprnd}; N \leftarrow 0, Z \leftarrow r\langle 8..15\rangle = 0$$

As with `CPBr`, the instruction sets the status bits according to the eight-bit value without regard to the high-order byte of register r. If you ever need to

FIGURE 6.11
The structure of the `while` statement in Figure 6.10.

S1
`while` (*C1*) {
  S2
}
S3

(a) The structure at Level HOL6.



(b) The same structure at Level Asmb5.

check for the ASCII NUL byte, you can load the byte into the accumulator and execute BREQ straightaway without using the compare byte instruction.

Translating the Do Loop

A highway patrol officer parks behind a sign. A driver passes by, traveling 20 meters per second, which is faster than the speed limit. When the driver is 40 meters down the road, the officer gets his car up to 25 meters per second to pursue the offender. How far from the sign is the officer when he catches up to the speeder?

The program in FIGURE 6.12 solves the problem by simulation. It is identical to the one in Figure 2.14. The values of cop and driver are the positions of the two motorists, initialized to 0 and 40, respectively. Each

FIGURE 6.12
The do statement at Level HOL6 and Level Asmb5. The C program is from Figure 2.14.

High-Order Language

```
#include <stdio.h>

int cop;
int driver;

int main() {
   cop = 0;
   driver = 40;
   do {
      cop += 25;
      driver += 20;
   }
   while (cop < driver);
   printf("%d", cop);
   return 0;
}
```

Assembly Language

```
0000  120007          BR      main
0003  0000    cop:    .BLOCK  2           ;global variable #2d
0005  0000    driver: .BLOCK  2           ;global variable #2d
              ;
0007  C00000  main:   LDWA    0,i         ;cop = 0
000A  E10003          STWA    cop,d
```

```
000D  C00028          LDWA    40,i        ;driver = 40
0010  E10005          STWA    driver,d
0013  C10003 do:      LDWA    cop,d       ;cop += 25
0016  600019          ADDA    25,i
0019  E10003          STWA    cop,d
001C  C10005          LDWA    driver,d    ;driver += 20
001F  600014          ADDA    20,i
0022  E10005          STWA    driver,d
0025  C10003 while:   LDWA    cop,d       ;while (cop < driver)
0028  A10005          CPWA    driver,d
002B  160013          BRLT    do
002E  390003          DECO    cop,d       ;printf("%d", cop)
0031  00              STOP
0032                  .END
```

execution of the `do` loop represents one second of elapsed time, during which the officer travels 25 meters and the driver 20, until the officer catches the driver.

A `do` statement has its test at the bottom of the loop. In this program, the compiler translates the `while` test to the sequence `LDWA`, `CPWA`, `BRLT`. `BRLT` executes the branch if N is set to 1. Because `CPWA` computes the difference, `cop - driver`, N will be 1 if

```
cop - driver < 0
```

that is, if

```
cop < driver
```

That is the condition under which the loop should repeat. FIGURE 6.13 shows the structure of the `do` statement at Levels 6 and 5.

Translating the For Loop

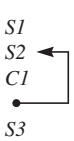
`for` statements are similar to `while` statements because the test for both is at the top of the loop. The compiler must generate code to initialize and to increment the control variable. The program in FIGURE 6.14 shows how a compiler generates code for the `for` statement. It translates the `for` statement into the following sequence at Level Asmb5:

- Initialize the control variable.
- Test the control variable.
- Execute the loop body.
- Increment the control variable.
- Branch to the test.

FIGURE 6.13
The structure of the `do` statement in Figure 6.12.

```
S1
do {
   S2
}
while (C1)
S3
```

(a) The structure at Level HOL6.



(b) The same structure at Level Asmb5.

FIGURE 6.14
The `for` statement at Level HOL6 and Level Asmb5.

High-Order Language

```
#include <stdio.h>

int main() {
   int j;
   for (j = 0; j < 3; j++) {
      printf("j = %d\n", j);
   }
   return 0;
}
```

Assembly Language

```
0000  120003          BR      main
            j:        .EQUATE 0           ;local variable #2d
            ;
0003  580002 main:    SUBSP   2,i         ;push #j
0006  C00000          LDWA    0,i         ;for (j = 0
0009  E30000          STWA    j,s
000C  A00003 for:     CPWA    3,i         ;j < 3
000F  1C002A          BRGE    endFor
0012  49002E          STRO    msg,d       ;printf("j = %d\n", j)
0015  3B0000          DECO    j,s
0018  D0000A          LDBA    '\n',i
001B  F1FC16          STBA    charOut,d
001E  C30000          LDWA    j,s         ;j++)
0021  600001          ADDA    1,i
0024  E30000          STWA    j,s
0027  12000C          BR      for
002A  500002 endFor:  ADDSP   2,i         ;pop #j
002D  00              STOP
002E  6A203D msg:     .ASCII  "j = \x00"
      2000
0033                  .END
```

In this program, `CPWA` computes the difference, `j - 3`. `BRGE` branches out of the loop if N is 0—that is, if

```
j - 3 >= 0
```

or, equivalently,

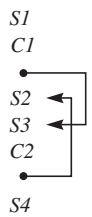
```
j >= 3
```

The body executes three times for `j` having the values 0, 1, and 2. After the last execution, `j` increments to 3, the loop terminates, and the value of 3 is not written by the output statement.

Spaghetti Code

At the assembly level, a programmer can write control structures that do not correspond to the control structures in C. FIGURE 6.15 shows one possible flow of control that is not directly possible in many Level-HOL6 languages. Condition *C1* is tested, and if it is true, a branch is taken to the middle of a loop whose test is *C2*.

FIGURE 6.15
A flow of control not possible directly in many HOL6 languages.



Assembly language programs generated by a compiler are usually longer than programs written by humans directly in assembly language. Not only that, but they often execute more slowly. If human programmers can write shorter, faster assembly language programs than compilers, why does anyone program in a high-order language? One reason is the ability of the compiler to perform type checking (as mentioned in Chapter 5). Another is the additional burden of responsibility that is placed on the programmer when given the freedom of using primitive branching instructions. If you are not careful when you write programs at Level Asmb5, the branching instructions can get out of hand.

The program in FIGURE 6.16 is an extreme example of the problem that can occur with unbridled use of primitive branching instructions. It

FIGURE 6.16
A mystery program.

```
0000  120009          BR      main
0003  0000    n1:     .BLOCK  2           ;#2d
0005  0000    n2:     .BLOCK  2           ;#2d
0007  0000    n3:     .BLOCK  2           ;#2d
              ;
0009  310005  main:   DECI    n2,d
000C  310007          DECI    n3,d
000F  C10005          LDWA    n2,d
0012  A10007          CPWA    n3,d
0015  16002A          BRLT    L1
0018  310003          DECI    n1,d
```

(*continues*)

FIGURE 6.16
A mystery program. (*continued*)

```
001B  C10003          LDWA    n1,d
001E  A10007          CPWA    n3,d
0021  160074          BRLT    L7
0024  120065          BR      L6
0027  E10007          STWA    n3,d
002A  310003 L1:      DECI    n1,d
002D  C10005          LDWA    n2,d
0030  A10003          CPWA    n1,d
0033  160053          BRLT    L5
0036  390003          DECO    n1,d
0039  390005          DECO    n2,d
003C  390007 L2:      DECO    n3,d
003F  00              STOP
0040  390005 L3:      DECO    n2,d
0043  390007          DECO    n3,d
0046  120081          BR      L9
0049  390003 L4:      DECO    n1,d
004C  390005          DECO    n2,d
004F  00              STOP
0050  E10003          STWA    n1,d
0053  C10007 L5:      LDWA    n3,d
0056  A10003          CPWA    n1,d
0059  160040          BRLT    L3
005C  390005          DECO    n2,d
005F  390003          DECO    n1,d
0062  12003C          BR      L2
0065  390007 L6:      DECO    n3,d
0068  C10003          LDWA    n1,d
006B  A10005          CPWA    n2,d
006E  160049          BRLT    L4
0071  12007E          BR      L8
0074  390003 L7:      DECO    n1,d
0077  390007          DECO    n3,d
007A  390005          DECO    n2,d
007D  00              STOP
007E  390005 L8:      DECO    n2,d
0081  390003 L9:      DECO    n1,d
0084  00              STOP
0085                  .END
```

is difficult to understand because of its lack of comments and indentation and its inconsistent branching style. Actually, the program performs a very simple task. Can you discover what it does?

The body of an `if` statement or a loop in C is a block of statements, sometimes contained in a compound statement delimited by braces, `{}`. Additional `if` statements and loops can be nested entirely within these blocks. FIGURE 6.17(a) pictures this situation schematically. A flow of control that is limited to nestings of the `if/else`, `switch`, `while`, `do`, and `for` statements is called *structured flow of control.*

Structured flow of control

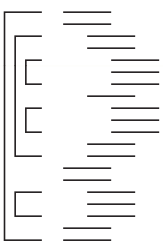
The branches in the mystery program do not correspond to the structured control constructs of C. Although the program's logic is correct for performing its intended task, it is difficult to decipher because the branching statements branch all over the place. This kind of program is called *spaghetti code*. If you draw an arrow from each branch statement to the statement to which it branches, the picture looks rather like a bowl of spaghetti, as Figure 6.17(b) shows.

Spaghetti code

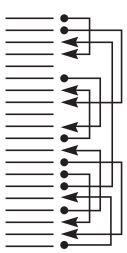
It is often possible to write efficient programs with unstructured branches. Such programs execute faster and require less memory for storage than if they were written in a high-order language with structured flow of control. Some specialized applications require this extra measure of efficiency and are therefore written directly in assembly language.

Balanced against this savings in execution time and memory space is difficulty in comprehension. When programs are hard to understand, they are hard to write, debug, and modify. The problem is economic. Writing, debugging, and modifying are all human activities that are labor intensive and, therefore, expensive. The question you must ask is whether the extra efficiency justifies the additional expense.

FIGURE 6.17
Two different styles of flow of control.



(a) Structured flow.



(b) Spaghetti code.

Flow of Control in Early Languages

Computers had been around for many years before structured flow of control was discovered. In the early days, there were no high-order languages. Everyone programmed in assembly language. Computer memories were expensive, and CPUs were slow by today's standards. Efficiency was all-important. Because a large body of software had not yet been generated, the problem of program maintenance was not appreciated.

The first widespread high-order language was Fortran, developed in the 1950s. Because people were used to dealing with branch instructions, they included them in the language. An unconditional branch in Fortran is

```
GOTO 260
```

A goto statement at Level HOL6

where 260 is the statement number of another statement. It is called a *goto statement*. A conditional branch is

```
IF (NUMBER .GE. 100) GOTO 500
```

where `.GE.` means "is greater than or equal to." This statement compares the value of variable `NUMBER` with 100. If it is greater than or equal to 100, the next statement executed is the one with a statement number of 500. Otherwise, the statement after the `IF` is executed.

Fortran's conditional `IF` is a big improvement over Level-Asmb5 branch instructions. It does not require a separate compare instruction to set the status bits. But notice how the flow of control is similar to Level-Asmb5 branching: If the test is true, do the `GOTO`. Otherwise, continue to the next statement.

As people developed more software, they noticed that it would be convenient to group statements into blocks for use in `if` statements and loops. The most notable language to make this advance was Algol 60, developed in 1960. It was the first widespread block-structured language, although its popularity was limited mainly to Europe.

The Structured Programming Theorem

The preceding sections show how high-level structured control statements translate into primitive branch statements at a lower level. They also show how you can write branches at the lower level that do not correspond to the structured constructs. This raises an interesting and practical question: Is it possible to write an algorithm with goto statements that will perform some processing that is impossible to perform with structured constructs? That is, if you limit yourself to structured flow of control, are there some problems you will not be able to solve that you could solve if unstructured gotos were allowed?

The structured programming theorem

Corrado Bohm and Giuseppe Jacopini answered this important question in a computer science journal article in 1966.[1] They proved mathematically that any algorithm containing gotos, no matter how complicated or unstructured, can be written with only nested `if` statements and `while` loops. Their result is called the *structured programming theorem*.

Bohm and Jacopini's paper was highly theoretical. It did not attract much attention at first because programmers generally had no desire to limit the freedom they had with goto statements. Bohm and Jacopini showed what could be done with nested `if` statements and `while` loops, but left unanswered why programmers would want to limit themselves that way.

1. Corrado Bohm and Giuseppe Jacopini, "Flow-Diagrams, Turing Machines and Languages with Only Two Formation Rules," *Communications of the ACM 9* (May 1966): 366–371.

People experimented with the concept anyway. They would take an algorithm in spaghetti code and try to rewrite it using structured flow of control without goto statements. Usually the new program was much clearer than the original. Occasionally it was even more efficient.

The Goto Controversy

Two years after Bohm and Jacopini's paper appeared, Edsger W. Dijkstra of the Technological University at Eindhoven, the Netherlands, wrote a letter to the editor of the same journal in which he stated his personal observation that good programmers used fewer gotos than poor programmers.[2]

In his opinion, a high density of gotos in a program indicated poor quality. He stated in part:

> More recently I discovered why the use of the goto statement has such disastrous effects, and I became convinced that the goto statement should be abolished from all "higher level" programming languages (i.e., everything except, perhaps, plain machine code). … The goto statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program.

To justify these statements, Dijkstra developed the idea of a set of coordinates that are necessary to describe the progress of the program. When a human tries to understand a program, he must maintain this set of coordinates mentally, perhaps unconsciously. Dijkstra showed that the coordinates to be maintained with structured flow of control were vastly simpler than those with unstructured gotos. Thus he was able to pinpoint the reason that structured flow of control is easier to understand.

Dijkstra acknowledged that the idea of eliminating gotos was not new. He mentioned several people who influenced him on the subject, one of whom was Niklaus Wirth, who had worked on the Algol 60 language.

Dijkstra's letter set off a storm of protest, now known as the famous *goto controversy*. To theoretically be able to program without goto was one thing. But to advocate that goto be abolished from high-order languages such as Fortran was altogether something else.

Old ideas die hard. However, the controversy has died down, and it is now generally recognized that Dijkstra was, in fact, correct. The reason is cost. When software managers began to apply the structured flow of control discipline, along with other structured design concepts, they found that the

2. Edsger W. Dijkstra, "Goto Statement Considered Harmful," *Communications of the ACM 11* (March 1968): 147–648.

resulting software was much less expensive to develop, debug, and maintain. It was usually well worth the additional memory requirements and extra execution time.

Fortran 77 was a more recent version of Fortran standardized in 1977. The goto controversy influenced its design. It contains a block style `IF` statement with an `ELSE` part similar to C. For example,

```
IF (NUMBER .GE. 100) THEN
    Statement 1
ELSE
    Statement 2
ENDIF
```

You can write the `IF` statement in Fortran 77 without goto.

One point to bear in mind is that the absence of gotos in a program does not guarantee that the program is well structured. It is possible to write a program with three or four nested `if` statements and `while` loops when only one or two are necessary. Also, if a language at any level contains only goto statements to alter the flow of control, they can always be used in a structured way to implement `if` statements and `while` loops. That is precisely what a C compiler does when it translates a program from Level HOL6 to Level Asmb5.

6.3 Function Calls and Parameters

A C function call changes the flow of control to the first executable statement in the function. At the end of the function, control returns to the statement following the function call. The compiler implements function calls with the `CALL` instruction, which has a mechanism for storing the return address on the run-time stack. It implements the return to the calling statement with `RET`, which uses the saved return address on the run-time stack to determine which instruction to execute next.

Translating a Function Call

FIGURE 6.18 shows how a compiler translates a function call without parameters. The program outputs three triangles of asterisks.

The `CALL` instruction pushes the content of the program counter onto the run-time stack and then loads the operand into the program counter. Here is the RTL specification of the `CALL` instruction:

The `CALL` *instruction*

SP ← SP − 2 ; Mem[SP] ← PC ; PC ← Oprnd

The return address for the procedure call is pushed onto the stack and a branch to the procedure is executed.

FIGURE 6.18
A procedure call at Level HOL6 and Level Asmb5.

High-Order Language

```
#include <stdio.h>

void printTri() {
   printf("*\n");
   printf("**\n");
   printf("***\n");
   printf("****\n");
}

int main() {
   printTri();
   printTri();
   printTri();
   return 0;
}
```

Assembly Language

```
0000  120019          BR      main
            ;
            ;******* void printTri()
0003  49000D printTri:STRO    msg1,d      ;printf("*\n")
0006  490010          STRO    msg2,d      ;printf("**\n")
0009  490014          STRO    msg3,d      ;printf("***\n")
000C  01              RET
000D  2A0A00 msg1:    .ASCII  "*\n\x00"
0010  2A2A0A msg2:    .ASCII  "**\n\x00"
      00
0014  2A2A2A msg3:    .ASCII  "***\n\x00"
      0A00
            ;
            ;******* int main()
0019  240003 main:    CALL    printTri    ;printTri()
001C  240003          CALL    printTri    ;printTri()
001F  240003          CALL    printTri    ;printTri()
0022  00              STOP
0023                  .END
```

The default addressing mode for CALL *is immediate.*

As with the branch instructions, CALL usually executes in the immediate addressing mode, in which case the operand is the operand specifier. If you do not specify the addressing mode, the Pep/9 assembler will assume immediate addressing.

The RET *instruction*

Here is the RTL specification of RET:

PC ← Mem[SP] ; SP ← SP + 2

The instruction moves the return address from the top of the stack into the program counter. Then it adds 2 to the stack pointer, which completes the pop operation.

In Figure 6.18,

```
0000 120019 BR main
```

puts 0019 into the program counter. The next statement to execute is, therefore, the one at 0019, which is the first CALL instruction. The discussion of the program in Figure 6.1 explains how the stack pointer is initialized to FB8F. FIGURE 6.19 shows the run-time stack before and after execution of the first CALL statement. As usual, the initial value of the stack pointer is FB8F.

The operations of CALL and RET crucially depend on the von Neumann execution cycle: fetch, decode, increment, execute, repeat. In particular, the increment step happens before the execute step. As a consequence, the statement that is executing is not the statement whose address is in the program counter. It is the statement that was fetched before the program counter was incremented and that is now contained in the instruction register. Why is that so important in the execution of CALL and RET?

Figure 6.19(a) shows the content of the program counter as 001C before execution of the first CALL instruction. It is not the address of the first CALL instruction, which is 0019. Why not? Because the program counter was incremented to 001C before execution of the CALL. Therefore, during execution of the first CALL instruction, the program counter contains the address of the instruction in main memory located just after the first CALL instruction.

What happens when the first CALL executes? First, SP ← SP – 2 subtracts two from SP, giving it the value FB8D. Then, Mem[SP] ← PC puts the value of the program counter, 001C, into main memory at address FB8D—that is, on top of the run-time stack. Finally, PC ← Oprnd puts 0003 into the program counter, because the operand specifier is 0003 and the addressing mode is immediate. The result is Figure 6.19(b).

The von Neumann cycle continues with the next fetch. But now the program counter contains 0003. So, the next instruction to be fetched is the one at address 0003, which is the first instruction of the printTri procedure. The output instructions of the procedure execute, producing the pattern of a triangle of asterisks.

FIGURE 6.19
Execution of the first CALL instruction in Figure 6.18.

(a) Before execution of the first CALL.

(b) After execution of the first CALL.

Eventually the RET instruction at 000C executes. FIGURE 6.20(a) shows the content of the program counter as 000D just before execution of RET. This might seem strange, because 000D is not even the address of an instruction. It is the address of the string "*\x00". Why? Because RET is a unary instruction and the CPU incremented the program counter by one. The first step in the execution of RET is PC ← Mem[SP], which puts 001C into the program counter. Then SP ← SP + 2 changes the stack pointer back to FB8F.

The von Neumann cycle continues with the next fetch. But now the program counter contains the address of the second CALL instruction. The same sequence of events happens as with the first call, producing another triangle of asterisks in the output stream. The third call does the same thing, after which the STOP instruction executes. Note that the value of the program counter after the STOP instruction executes is 0023 and not 0022, which is the address of the STOP instruction.

Now you should see why increment comes before execute in the von Neumann execution cycle. To store the return address on the run-time stack, the CALL instruction needs to store the address of the instruction following the CALL. It can do that only if the program counter has been incremented before the CALL statement executes.

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

Translating Call-by-Value Parameters with Global Variables

The allocation process when you call a void function in C is

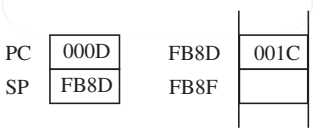
- Push the actual parameters.
- Push the return address.
- Push storage for the local variables.

At Level HOL6, the instructions that perform these operations on the stack are hidden. The programmer simply writes the function call, and during execution the stack pushes occur automatically.

At the assembly level, however, the translated program must contain explicit instructions for the pushes. The program in FIGURE 6.21, which is identical to the program in Figure 2.16, is a Level-HOL6 program that prints a bar chart and the program's corresponding Level-Asmb5 translation. It shows the Level-Asmb5 statements, not explicit at Level HOL6, that are required to push the parameters.

The caller in main() is responsible for pushing the actual parameters and executing CALL, which pushes the return address onto the stack. The callee in printBar() is responsible for pushing storage on the stack for its local variables. After the callee executes, it must pop the storage for the

FIGURE 6.20
The first execution of the RET instruction in Figure 6.18.



(a) Before the first execution of RET.

(b) After the first execution of RET.

FIGURE 6.21
Call-by-value parameters with global variables. The C program is from Figure 2.16.

High-Order Language

```
#include <stdio.h>

int numPts;
int value;
int j;

void printBar(int n) {
   int k;
   for (k = 1; k <= n; k++) {
      printf("*");
   }
   printf("\n");
}

int main() {
   scanf("%d", &numPts);
   for (j = 1; j <= numPts; j++) {
      scanf("%d", &value);
      printBar(value);
   }
   return 0;
}
```

Assembly Language

```
0000  120034          BR      main
0003  0000    numPts:  .BLOCK  2           ;global variable #2d
0005  0000    value:   .BLOCK  2           ;global variable #2d
0007  0000    j:       .BLOCK  2           ;global variable #2d
              ;
              ;******* void printBar(int n)
              n:       .EQUATE 4           ;formal parameter #2d
              k:       .EQUATE 0           ;local variable #2d
0009  580002 printBar:SUBSP   2,i          ;push #k
000C  C00001          LDWA    1,i          ;for (k = 1
000F  E30000          STWA    k,s
0012  A30004 for1:    CPWA    n,s          ;k <= n
0015  1E002A          BRGT    endFor1
0018  D0002A          LDBA    '*',i        ;printf("*")
```

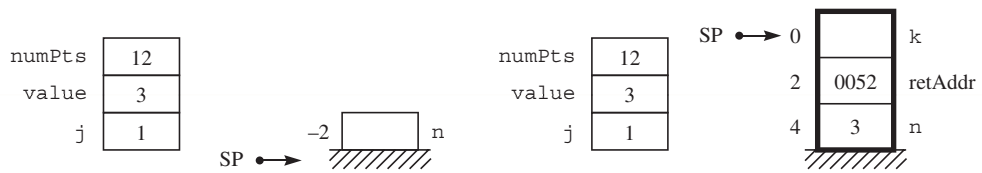
```
001B  F1FC16          STBA    charOut,d
001E  C30000          LDWA    k,s         ;k++)
0021  600001          ADDA    1,i
0024  E30000          STWA    k,s
0027  120012          BR      for1
002A  D0000A endFor1: LDBA    '\n',i      ;printf("\n")
002D  F1FC16          STBA    charOut,d
0030  500002          ADDSP   2,i         ;pop #k
0033  01              RET
              ;
              ;******* main()
0034  310003 main:    DECI    numPts,d    ;scanf("%d", &numPts)
0037  C00001          LDWA    1,i         ;for (j = 1
003A  E10007          STWA    j,d
003D  A10003 for2:    CPWA    numPts,d    ;j <= numPts
0040  1E0061          BRGT    endFor2
0043  310005          DECI    value,d     ;scanf("%d", &value)
0046  C10005          LDWA    value,d     ;move value
0049  E3FFFE          STWA    -2,s
004C  580002          SUBSP   2,i         ;push #n
004F  240009          CALL    printBar    ;printBar(value)
0052  500002          ADDSP   2,i         ;pop #n
0055  C10007          LDWA    j,d         ;j++)
0058  600001          ADDA    1,i
005B  E10007          STWA    j,d
005E  12003D          BR      for2
0061  00      endFor2: STOP
0062                  .END
```

local variables and then pop the return address by executing `RET`. Before the caller can continue, it must pop the actual parameters.

In summary, the caller and callee procedures do the following:

- Caller pushes actual parameters (executes `SUBSP`).
- Caller pushes return address (executes `CALL`).
- Callee pushes storage for local variables (executes `SUBSP`).
- Callee executes its body.
- Callee pops local variables (executes `ADDSP`).
- Callee pops return address (executes `RET`).
- Caller pops actual parameters (executes `ADDSP`).

FIGURE 6.22
Call-by-value parameters with global variables.



(a) After `scanf("%d", &value)`. **(b)** After `printBar(value)`.

Note the symmetry of the operations. The last three operations undo the first three operations in reverse order. That order is a consequence of the last-in, first-out property of the stack.

The global variables in the Level-HOL6 main program—`numPts`, `value`, and `j`—correspond to the identical Level-Asmb5 symbols, whose symbol values are 0003, 0005, and 0007, respectively. These are the addresses of the memory cells that will hold the run-time values of the global variables. FIGURE 6.22(a) shows the global variables on the left with their symbols in place of their addresses. The values for the global variables are the ones after

```
scanf("%d", &value);
```

executes for the first time.

What do the formal parameter, `n`, and the local variable, `k`, correspond to at Level Asmb5? Not absolute addresses, but stack-relative addresses. Procedure `printBar` defines them with

```
n: .EQUATE 4 ;formal parameter #2d
k: .EQUATE 0 ;local variable #2d
```

Remember that `.EQUATE` does not generate object code. The assembler does not reserve storage for them at translation time. Instead, storage for `n` and `k` is allocated on the stack at run time. The decimal numbers 4 and 0 are the stack offsets appropriate for `n` and `k` during execution of the procedure, as Figure 6.22(b) shows. The procedure refers to them with stack-relative addressing.

The statements that correspond to the procedure call in the caller are

```
0046  C10005  LDWA   value,d
0049  E3FFFE  STWA   -2,s
004C  580002  SUBSP  2,i        ;push #n
004F  240009  CALL   printBar   ;printBar(value)
0052  500002  ADDSP  2,i        ;pop #n
```

Because the parameter is a global variable that is called by value, LDWA uses direct addressing. That puts the run-time value of variable value in the accumulator, which STWA then moves onto the stack. The offset is –2 because value is a two-byte integer quantity, as Figure 6.22(a) shows.

The statements that correspond to the procedure call in the callee are

```
0009  580002 printBar:SUBSP  2,i  ;push #k
...
0030  500002           ADDSP  2,i  ;pop #k
0033  01               RET
```

SUBSP subtracts 2 because the local variable, k, is a two-byte integer quantity. Figure 6.22(a) shows the run-time stack just after the first input of global variable value and just before the first procedure call. It corresponds directly to Figure 2.17(d). Figure 6.22(b) shows the stack just after the procedure call and corresponds directly to Figure 2.17(g). Note that the return address, which is labeled ra1 in Figure 2.17, is here shown to be 0052, which is the machine language address of the instruction following the CALL instruction.

The stack address of n is 4 because both k and the return address occupy two bytes on the stack. If there were more local variables, the stack address of n would be correspondingly greater. The compiler must compute the stack addresses from the number and size of the quantities on the stack.

In summary, to translate call-by-value parameters with global variables, the compiler generates code as follows:

The translation rules for call-by-value parameters with global variables

- To get the actual parameter in the caller, it generates a load instruction with direct addressing.
- To get the formal parameter in the callee, it generates a load instruction with stack-relative addressing.

Translating Call-by-Value Parameters with Local Variables

The program in FIGURE 6.23 is identical to the one in Figure 6.21 except that the variables in main() are local instead of global. Although the program behaves like the one in Figure 6.21, the memory model and the translation to Level Asmb5 are different.

You can see that the versions of void function printBar() at Level HOL6 are identical in Figure 6.21 and Figure 6.23. Hence, it should not be surprising that the compiler generates identical object code for the two versions of printBar() at Level Asmb5. The only difference between the

FIGURE 6.23
Call-by-value parameters with local variables.

High-Order Language

```
#include <stdio.h>

void printBar(int n) {
   int k;
   for (k = 1; k <= n; k++) {
      printf("*");
   }
   printf("\n");
}

int main() {
   int numPts;
   int value;
   int j;
   scanf("%d", &numPts);
   for (j = 1; j <= numPts; j++) {
      scanf("%d", &value);
      printBar(value);
   }
   return 0;
}
```

Assembly Language

```
0000  12002E          BR      main
              ;
              ;******* void printBar(int n)
              n:      .EQUATE 4           ;formal parameter #2d
              k:      .EQUATE 0           ;local variable #2d
0003  580002 printBar:SUBSP   2,i         ;push #k
0006  C00001          LDWA    1,i         ;for (k = 1
0009  E30000          STWA    k,s
000C  A30004 for1:    CPWA    n,s         ;k <= n
000F  1E0024          BRGT    endFor1
0012  D0002A          LDBA    '*',i       ;printf("*")
0015  F1FC16          STBA    charOut,d
0018  C30000          LDWA    k,s         ;k++)
```

```
001B  600001          ADDA    1,i
001E  E30000          STWA    k,s
0021  12000C          BR      for1
0024  D0000A endFor1: LDBA    '\n',i      ;printf("\n")
0027  F1FC16          STBA    charOut,d
002A  500002          ADDSP   2,i         ;pop #k
002D  01              RET
            ;
            ;******* main()
            numPts:   .EQUATE 4           ;local variable #2d
            value:    .EQUATE 2           ;local variable #2d
            j:        .EQUATE 0           ;local variable #2d
002E  580006 main:    SUBSP   6,i         ;push #numPts #value #j
0031  330004          DECI    numPts,s    ;scanf("%d", &numPts)
0034  C00001          LDWA    1,i         ;for (j = 1
0037  E30000          STWA    j,s
003A  A30004 for2:    CPWA    numPts,s    ;j <= numPts
003D  1E005E          BRGT    endFor2
0040  330002          DECI    value,s     ;scanf("%d", &value)
0043  C30002          LDWA    value,s     ;move value
0046  E3FFFE          STWA    -2,s
0049  580002          SUBSP   2,i         ;push #n
004C  240003          CALL    printBar    ;printBar(value)
004F  500002          ADDSP   2,i         ;pop #n
0052  C30000          LDWA    j,s         ;j++)
0055  600001          ADDA    1,i
0058  E30000          STWA    j,s
005B  12003A          BR      for2
005E  500006 endFor2: ADDSP   6,i         ;pop #j #value #numPts
0061  00              STOP
0062                  .END
```

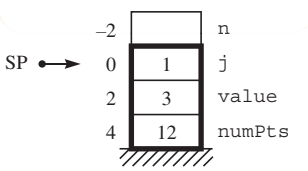
two programs is in `main()`. FIGURE 6.24(a) shows the allocation of `numPts`, `value`, and `j` on the run-time stack in the main program. Figure 6.24(b) shows the stack after `printTri` is called for the first time. Because `value` is a local variable, the compiler generates `LDWA value,s` with stack-relative addressing to get the actual value of `value`, which it then puts in the stack cell for formal parameter `n`.

In summary, to translate call-by-value parameters with local variables, the compiler generates code as follows:

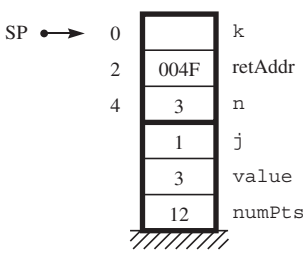
The translation rules for call-by-value parameters with local variables

- To get the actual parameter in the caller, it generates a load instruction with stack-relative addressing.
- To get the formal parameter in the callee, it generates a load instruction with stack-relative addressing.

FIGURE 6.24
The first execution of the function call in Figure 6.23.



(a) After `scanf("%d", &value)`.



(b) After `printBar(value)`.

Translating Non-void Function Calls

The allocation process when you call a function is

- Push storage for the return value.
- Push the actual parameters.
- Push the return address.
- Push storage for the local variables.

Allocation for a non-void function call differs from that for a procedure (void function) call by the extra value that you must allocate for the returned function value.

FIGURE 6.25 shows a program that computes a binomial coefficient recursively and is identical to the one in Figure 2.28. It is based on Pascal's triangle of coefficients (shown in Figure 2.27). The recursive definition of the binomial coefficient is

$$b(n, k) = \begin{cases} 1 & \text{if } k = 0, \\ 1 & \text{if } n = k, \\ b(n-1, k) + b(n-1, k-1) & \text{if } 0 < k < n. \end{cases}$$

FIGURE 6.25
A recursive nonvoid function at Level HOL6 and Level Asmb5. The C program is from Figure 2.28.

High-Order Language

```
#include <stdio.h>

int binCoeff(int n, int k) {
   int y1, y2;
   if ((k == 0) || (n == k)) {
      return 1;
   }
```

```
   else {
      y1 = binCoeff(n - 1, k); // ra2
      y2 = binCoeff(n - 1, k - 1); // ra3
      return y1 + y2;
   }
}
int main() {
   printf("binCoeff(3, 1) = %d\n", binCoeff(3, 1)); // ra1
   return 0;
}
```

Assembly Language

```
0000  12006B          BR      main
            ;
            ;******* int binomCoeff(int n, int k)
            retVal:   .EQUATE 10          ;return value #2d
            n:        .EQUATE 8           ;formal parameter #2d
            k:        .EQUATE 6           ;formal parameter #2d
            y1:       .EQUATE 2           ;local variable #2d
            y2:       .EQUATE 0           ;local variable #2d
0003  580004 binCoeff:SUBSP   4,i         ;push #y1 #y2
0006  C30006 if:      LDWA    k,s         ;if ((k == 0)
0009  180015          BREQ    then
000C  C30008          LDWA    n,s         ;|| (n == k))
000F  A30006          CPWA    k,s
0012  1A001F          BRNE    else
0015  C00001 then:    LDWA    1,i         ;return 1
0018  E3000A          STWA    retVal,s
001B  500004          ADDSP   4,i         ;pop #y2 #y1
001E  01              RET
001F  C30008 else:    LDWA    n,s         ;move n - 1
0022  700001          SUBA    1,i
0025  E3FFFC          STWA    -4,s
0028  C30006          LDWA    k,s         ;move k
002B  E3FFFA          STWA    -6,s
002E  580006          SUBSP   6,i         ;push #retVal #n #k
0031  240003          CALL    binCoeff    ;binCoeff(n - 1, k)
0034  500006 ra2:     ADDSP   6,i         ;pop #k #n #retVal
0037  C3FFFE          LDWA    -2,s        ;y1 = binomCoeff(n - 1, k)
003A  E30002          STWA    y1,s
```

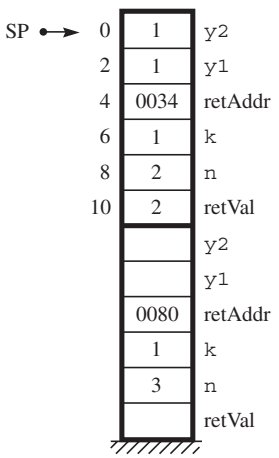
(*continues*)

FIGURE 6.25
A recursive nonvoid function at Level HOL6 and Level Asmb5. The C program is from Figure 2.28. (*continued*)

```
003D  C30008          LDWA    n,s         ;move n - 1
0040  700001          SUBA    1,i
0043  E3FFFC          STWA    -4,s
0046  C30006          LDWA    k,s         ;move k - 1
0049  700001          SUBA    1,i
004C  E3FFFA          STWA    -6,s
004F  580006          SUBSP   6,i         ;push #retVal #n #k
0052  240003          CALL    binCoeff    ;binomCoeff(n - 1, k - 1)
0055  500006 ra3:     ADDSP   6,i         ;pop #k #n #retVal
0058  C3FFFE          LDWA    -2,s        ;y2 = binomCoeff(n - 1, k - 1)
005B  E30000          STWA    y2,s
005E  C30002          LDWA    y1,s        ;return y1 + y2
0061  630000          ADDA    y2,s
0064  E3000A          STWA    retVal,s
0067  500004 endIf:   ADDSP   4,i         ;pop #y2 #y1
006A  01              RET
            ;
            ;******* main()
006B  49008D main:    STRO    msg,d       ;printf("binCoeff(3, 1) = %d\n",
006E  C00003          LDWA    3,i         ;move 3
0071  E3FFFC          STWA    -4,s
0074  C00001          LDWA    1,i         ;move 1
0077  E3FFFA          STWA    -6,s
007A  580006          SUBSP   6,i         ;push #retVal #n #k
007D  240003          CALL    binCoeff    ;binCoeff(3, 1)
0080  500006 ra1:     ADDSP   6,i         ;pop #k #n #retVal
0083  3BFFFE          DECO    -2,s
0086  D0000A          LDBA    '\n',i
0089  F1FC16          STBA    charOut,d
008C  00              STOP
008D  62696E msg:     .ASCII  "binCoeff(3, 1) = \x00"
      ...
009F                  .END
```

The function tests for the base cases with an `if` statement, using the OR Boolean operator. If neither base case is satisfied, it calls itself recursively twice—once to compute $b(n - 1, k)$ and once to compute $b(n - 1, k - 1)$. Figure 2.29 shows the run-time stack produced by a call from the main program with actual parameters (3, 1). The function is called twice more with parameters (2, 1) and (1, 1), followed by a return. Then a call with parameters (1, 0) is executed, followed by a second return, and so on. FIGURE 6.26 shows the run-time stack at the assembly level immediately after the second return. It corresponds directly to the Level-HOL6 diagram of Figure 2.29(g). The return address labeled ra2 in Figure 2.29(g) is 0034 in Figure 6.26, the address of the instruction after the first `CALL` in the function. Similarly, the address labeled ra1 in Figure 2.29(g) is 0080 in Figure 6.26.

FIGURE 6.26
The run-time stack of Figure 6.25 immediately after the second return.



At the start of the main program when the stack pointer has its initial value, the first actual parameter has a stack offset of –4, and the second has a stack offset of –6. In a procedure call (a void function), these offsets would be –2 and –4, respectively. Their magnitudes are greater by 2 because of the two-byte value returned on the stack by the function. The `SUBSP` instruction at 007A allocates six bytes, two for the return value and two each for the actual parameters.

When the function returns control to `ADDSP` at 0080, the value it returns will be on the stack below the two actual parameters. `ADDSP` pops the parameters and return value by adding 6 to the stack pointer, after which it points to the cell directly below the returned value. So `DECO` outputs the value with stack-relative addressing and an offset of –2.

The function calls itself by allocating actual parameters according to the standard technique. For the first recursive call, it computes `n − 1` and `k` and pushes those values onto the stack along with storage for the returned value. After the return, the sequence

```
0034  500006 ra2:  ADDSP  6,i    ;pop #k #n #retVal
0037  C3FFFE       LDWA   -2,s   ;y1 = binomCoeff(n - 1, k)
003A  E30002       STWA   y1,s
```

pops the two actual parameters and return value and assigns the return value to `y1`. For the second call, it pushes `n - 1` and `k - 1` and assigns the return value to `y2` similarly.

Translating Call-by-Reference Parameters with Global Variables

C provides call-by-reference parameters so that the called procedure can change the value of the actual parameter in the calling procedure. Figure 2.20 shows a program at Level HOL6 that uses call by reference to put two global

variables, a and b, in order. FIGURE 6.27 shows the same program together with the object program that a compiler would produce.

Parameters called by reference differ from parameters called by value in C because the actual parameter provides a reference to a variable in the caller instead of the value of the variable. At the assembly level, the code that pushes the actual parameter onto the stack pushes the address of the actual parameter, which corresponds to the & address operator in the parameter list. When the actual parameter is a global variable, its address is available as the value of its symbol. So, the code to get the address of a global variable

FIGURE 6.27
Call-by-reference parameters with global variables. The C program is from Figure 2.20.

High-Order Language

```
#include <stdio.h>

int a, b;

void swap(int *r, int *s) {
   int temp;
   temp = *r;
   *r = *s;
   *s = temp;
}

void order(int *x, int *y) {
   if (*x > *y) {
      swap(x, y);
   }  // ra2
}

int main() {
   printf("Enter an integer: ");
   scanf("%d", &a);
   printf("Enter an integer: ");
   scanf("%d", &b);
   order(&a, &b);
   printf("Ordered they are: %d, %d\n", a, b); // ra1
   return 0;
}
```

Assembly Language

```
0000  12003F          BR      main
0003  0000    a:      .BLOCK  2           ;global variable #2d
0005  0000    b:      .BLOCK  2           ;global variable #2d
              ;
              ;******* void swap(int *r, int *s)
              r:      .EQUATE 6           ;formal parameter #2h
              s:      .EQUATE 4           ;formal parameter #2h
              temp:   .EQUATE 0           ;local variable #2d
0007  580002  swap:   SUBSP   2,i         ;push #temp
000A  C40006          LDWA    r,sf        ;temp = *r
000D  E30000          STWA    temp,s
0010  C40004          LDWA    s,sf        ;*r = *s
0013  E40006          STWA    r,sf
0016  C30000          LDWA    temp,s      ;*s = temp
0019  E40004          STWA    s,sf
001C  500002          ADDSP   2,i         ;pop #temp
001F  01              RET
              ;
              ;******* void order(int *x, int *y)
              x:      .EQUATE 4           ;formal parameter #2h
              y:      .EQUATE 2           ;formal parameter #2h
0020  C40004  order:  LDWA    x,sf        ;if (*x > *y)
0023  A40002          CPWA    y,sf
0026  14003E          BRLE    endIf
0029  C30004          LDWA    x,s         ;move x
002C  E3FFFE          STWA    -2,s
002F  C30002          LDWA    y,s         ;move y
0032  E3FFFC          STWA    -4,s
0035  580004          SUBSP   4,i         ;push #r #s
0038  240007          CALL    swap        ;swap(x, y)
003B  500004          ADDSP   4,i         ;pop #s #r
003E  01      endIf:  RET
              ;
              ;******* main()
003F  490073  main:   STRO    msg1,d      ;printf("Enter an integer: ")
0042  310003          DECI    a,d         ;scanf("%d", &a)
0045  490073          STRO    msg1,d      ;printf("Enter an integer: ")
0048  310005          DECI    b,d         ;scanf("%d", &b)
004B  C00003          LDWA    a,i         ;move &a
```

(*continues*)

FIGURE 6.27
Call-by-reference parameters with global variables. The C program is from Figure 2.20. (*continued*)

```
004E  E3FFFE          STWA    -2,s
0051  C00005          LDWA    b,i         ;move &b
0054  E3FFFC          STWA    -4,s
0057  580004          SUBSP   4,i         ;push #x #y
005A  240020          CALL    order       ;order(&a, &b)
005D  500004 ra1:     ADDSP   4,i         ;pop #y #x
0060  490086          STRO    msg2,d      ;printf("Ordered they are: %d, %d\n"
0063  390003          DECO    a,d         ;, a
0066  490099          STRO    msg3,d
0069  390005          DECO    b,d         ;, b)
006C  D0000A          LDBA    '\n',i
006F  F1FC16          STBA    charOut,d
0072  00              STOP
0073  456E74 msg1:    .ASCII  "Enter an integer: \x00"
      ...
0086  4F7264 msg2:    .ASCII  "Ordered they are: \x00"
      ...
0099  2C2000 msg3:    .ASCII  ", \x00"
009C                  .END
```

is a load instruction with immediate addressing. In Figure 6.27, the code to get the address of `a` is

```
004B C00003 LDWA a,i ;move &a
```

The value of the symbol `a` is 0003, the address of where the `a` is stored. The machine code for this instruction is C00003. C0 is the instruction specifier for the load accumulator instruction with addressing-aaa field of 000 to indicate immediate addressing. With immediate addressing, the operand specifier is the operand. Consequently, this instruction loads 0003 into the accumulator. The store instruction

```
004E E3FFFE STWA -2,s
```

then puts the address of `a` on the run-time stack.

Similarly, the code to push the address of `b` is

```
0051 C00005 LDWA b,i ;move &b
```

The machine code for this instruction is C00005, where 0005 is the address of `b`. This instruction loads 0005 into the accumulator with immediate addressing, after which the store instruction puts it on the run-time stack.

In procedure `order()`, the compiler translates the `if` statement

```
if (*x > *y)
```

at Level HOL6 into the three statements

```
0020  C40004 order:  LDWA  x,sf   ;if (*x > *y)
0023  A40002         CPWA  y,sf
0026  14003E         BRLE  endIf
```

at Level Asmb5. The addressing mode letters for the load and compare instructions are `sf`, which stands for *stack-relative deferred addressing*.

Remember that the relation between the operand and the operand specifier with stack-relative addressing is

Oprnd = Mem[SP + OprndSpec]

Stack-relative addressing

The relation between the operand and the operand specifier with stack-relative deferred addressing is

Oprnd = Mem[Mem[SP + OprndSpec]]

Stack-relative deferred addressing

In other words, Mem[SP + OprndSpec] is the address of the operand, rather than the operand itself.

In the preceding `LDWA` instruction, `x` is the operand specifier. It is the formal parameter stored on the run-time stack. It is the address of `a`, which was put on the stack by the caller. In procedure `order()`, the two expressions `x` and `*x` at Level HOL6 correspond to Level Asmb5 as follows:

- Pointer `x` at Level HOL6 is at Mem[SP + x] at Level Asmb5. You access it with stack-relative addressing, `s`.
- Integer `*x` at Level HOL6 is at Mem[Mem[SP + x]] at Level Asmb5. You access it with stack-relative deferred addressing, `sf`.

Procedure `order()` calls `swap(x, y)`. Because the actual parameter is `x` and not `*x`, procedure `order()` simply transfers the address for `swap()` to use. The statement

```
0029 C30004 LDWA x,s ;move x
```

uses stack-relative addressing to get the address into the accumulator. The next instruction puts it on the run-time stack.

In procedure `order()`, the compiler must translate

```
temp = *r;
```

It must load the value of `*r` into the accumulator and then store it in `temp`. Because it is loading `*r` instead of `r`, it uses stack-relative deferred addressing instead of stack-relative addressing. The compiler generates the following object code to translate the assignment statement:

```
000A   C40006   LDWA   r,sf     ;temp = *r
000D   E30000   STWA   temp,s
```

The next assignment statement in procedure `swap()`

```
*r = *s;
```

has the `*` dereference operator on both variables. Consequently, the compiler generates `LDWA` and `STWA` both with stack-relative deferred addressing.

```
0010   C40004   LDWA   s,sf   ;*r = *s
0013   E40006   STWA   r,sf
```

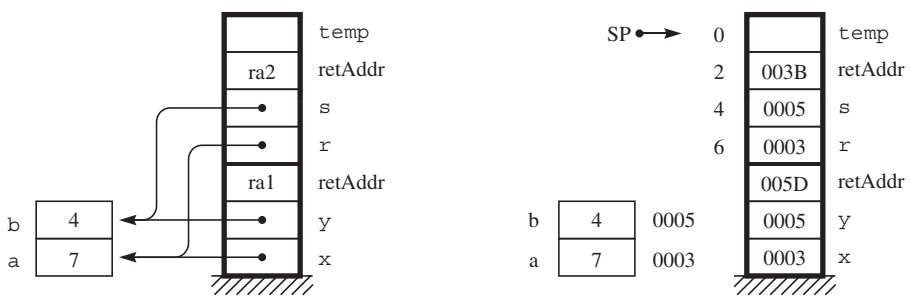
FIGURE 6.28 shows the run-time stack at Level HOL6 and Level Asmb5. The address of `a` is 0003, which `main()` pushes onto the run-time stack for formal parameter `x` when it calls `order()`. Procedure `order()` pushes the same address onto the run-time stack when it calls procedure `swap()`.

In summary, to translate call-by-reference parameters with global variables, the compiler generates code as follows:

The translation rules for call-by-reference parameters with global variables

- To get the actual parameter in the caller, it generates a load instruction with immediate addressing.
- To get the formal parameter `x` in the callee, it generates a load instruction with stack-relative addressing.
- To get the dereferenced formal parameter `*x` in the callee, it generates a load instruction with stack-relative deferred addressing.

FIGURE 6.28
The run-time stack for Figure 6.27 at Level HOL6 and Level Asmb5.



(a) The run-time stack at Level HOL6.

(b) The run-time stack at Level Asmb5.

Translating Call-by-Reference Parameters with Local Variables

FIGURE 6.29 shows a program that computes the perimeter of a rectangle given its width and height. The main program prompts the user for the width and the height, which it inputs into two local variables named `width` and `height`. A third local variable is named `perim`. The main program calls

FIGURE 6.29
Call-by-reference parameters with local variables.

High-Order Language

```
#include <stdio.h>

void rect(int *p, int w, int h) {
   *p = (w + h) * 2;
}

int main() {
   int perim, width, height;
   printf("Enter width: ");
   scanf("%d", &width);
   printf("Enter height: ");
   scanf("%d", &height);
   rect(&perim, width, height);
   // ra1
   printf("Perimeter = %d\n", perim);
   return 0;
}
```

Assembly Language

```
0000  12000E          BR      main
            ;
            ;******* void rect(int *p, int w, int h)
            p:        .EQUATE 6           ;formal parameter #2h
            w:        .EQUATE 4           ;formal parameter #2d
            h:        .EQUATE 2           ;formal parameter #2d
0003  C30004 rect:    LDWA    w,s         ;*p = (w + h) * 2
0006  630002          ADDA    h,s
0009  0A              ASLA
000A  E40006          STWA    p,sf
000D  01              RET
```

(continues)

FIGURE 6.29
Call-by-reference parameters with local variables. (*continued*)

```
                ;
                ;******* main()
                perim:   .EQUATE 4           ;local variable #2d
                width:   .EQUATE 2           ;local variable #2d
                height:  .EQUATE 0           ;local variable #2d
000E  580006 main:    SUBSP   6,i          ;push #perim #width #height
0011  490049          STRO    msg1,d       ;printf("Enter width: ")
0014  330002          DECI    width,s      ;scanf("%d", &width)
0017  490057          STRO    msg2,d       ;printf("Enter height: ")
001A  330000          DECI    height,s     ;scanf("%d", &height)
001D  03              MOVSPA               ;move &perim
001E  600004          ADDA    perim,i
0021  E3FFFE          STWA    -2,s
0024  C30002          LDWA    width,s      ;move width
0027  E3FFFC          STWA    -4,s
002A  C30000          LDWA    height,s     ;move height
002D  E3FFFA          STWA    -6,s
0030  580006          SUBSP   6,i          ;push #p #w #h
0033  240003          CALL    rect         ;rect(&perim, width, height)
0036  500006 ra1:     ADDSP   6,i          ;pop #h #w #p
0039  490066          STRO    msg3,d       ;printf("Perimeter = %d\n", perim);
003C  3B0004          DECO    perim,s
003F  D0000A          LDBA    '\n',i
0042  F1FC16          STBA    charOut,d
0045  500006          ADDSP   6,i          ;pop #height #width #perim
0048  00              STOP
0049  456E74 msg1:    .ASCII  "Enter width: \x00"
      ...
0057  456E74 msg2:    .ASCII  "Enter height: \x00"
      ...
0066  506572 msg3:    .ASCII  "Perimeter = \x00"
      ...
0073                  .END
```

Input/Output

```
Enter width: 8
Enter height: 5
Perimeter = 26
```

a procedure (a void function) named `rect()`, passing `width` and `height` by value and `perim` by reference. The figure shows the input and output when the user enters 8 for the width and 5 for the height.

FIGURE 6.30 shows the run-time stack at Level HOL6 for the program. Compare it to Figure 6.28(a) for a program with global variables that are called by reference. In that program, formal parameters `x`, `y`, `r`, and `s` refer to global variables `a` and `b`. At Level Asmb5, `a` and `b` are allocated at translation time with the `.EQUATE` dot command. Their symbols are their addresses. However, Figure 6.30 shows `perim` to be allocated on the run-time stack. The statement

```
000E 580006 main: SUBSP 6,i
```

allocates storage for `perim`, whose symbol is defined by

```
perim: .EQUATE 4 ;local variable #2d
```

Its symbol is not its absolute address. Its symbol is its address relative to the top of the run-time stack, as FIGURE 6.31(a) shows. Its absolute address is FB8D. Why? Because that is the location of the bottom of the application run-time stack, as the memory map in Figure 4.41 shows.

So, the compiler cannot generate code to push parameter `perim` with

```
LDWA perim,i
STWA -2,s
```

as it does for global variables. If it generated those instructions, procedure `rect()` would modify the content of Mem[0004], and 0004 is not where `perim` is located.

FIGURE 6.30
The run-time stack for Figure 6.29 at Level HOL6.

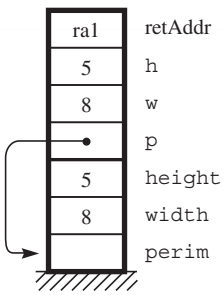
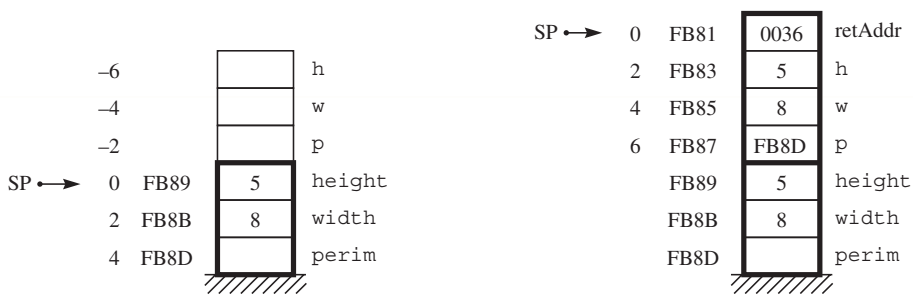


FIGURE 6.31
The run-time stack for Figure 6.29 at Level Asmb5.



(a) Before the procedure call.

(b) After the procedure call.

The absolute address of `perim` is FB8D. Figure 6.31(a) shows that you could calculate it by adding the value of `perim`, 4, to the value of the stack pointer. Fortunately, there is a unary instruction `MOVSPA` that moves the content of the stack pointer to the accumulator. The RTL specification of `MOVSPA` is

The `MOVSPA` instruction

A ← SP

To push the address of `perim`, the compiler generates the following instructions at 001D in Figure 6.29:

```
001D  03      MOVSPA           ;move &perim
001E  600004  ADDA    perim,i
0021  E3FFFE  STWA    -2,s
```

The first instruction moves the content of the stack pointer to the accumulator. The accumulator then contains FB89. The second instruction adds the value of `perim`, which is 4, to the accumulator, making it FB8D. The third instruction puts the address of `perim` in the cell for `p`, which procedure `rect()` uses to store the perimeter. Figure 6.31(b) shows the result.

The compiler translates `*p` in `rect()` as it would any dereferenced call-by-reference parameter. Namely, at 000A it stores the value using stack-relative deferred addressing:

```
000A E40006 STWA p,sf
```

With stack-relative deferred addressing, the address of the operand is on the stack. The operand is

Stack-relative deferred addressing

Oprnd = Mem[Mem[SP + OprndSpec]]

This instruction adds the stack pointer FB81 to the operand specifier 6, yielding FB87. Because Mem[FB87] is FB8D, it stores the accumulator at Mem[FB8D] in the stack.

In summary, to translate call-by-reference parameters with local variables, the compiler generates code as follows:

The translation rules for call-by-reference parameters with local variables

- To get the actual parameter in the caller, the compiler generates the unary `MOVSPA` instruction followed by the `ADDA` instruction with immediate addressing.
- To get the formal parameter `x` in the callee, the compiler generates a load instruction with stack-relative addressing.
- To get the dereferenced formal parameter `*x` in the callee, the compiler generates a load instruction with stack-relative deferred addressing.

Translating Boolean Types

Several schemes exist for storing Boolean values at the assembly level. The one most appropriate for C is to treat the values true and false as integer constants. The values are

```
const int true = 1;
const int false = 0;
```

FIGURE 6.32 is a program that declares a Boolean function named `inRange()`. The library `stdbool.h` defines the `bool` type as if true and false were declared as above.

FIGURE 6.32
Translation of a Boolean type.

High-Order Language

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

const int LOWER = 21;
const int UPPER = 65;

bool inRange(int a) {
   if ((LOWER <= a) && (a <= UPPER)) {
      return true;
   }
   else {
      return false;
   }
}

int main() {
   int age;
   scanf("%d", &age);
   if (inRange(age)) {
      printf("Qualified\n");
   }
   else {
      printf("Unqualified\n");
   }
   return 0;
}
```

(continues)

FIGURE 6.32
Translation of a Boolean type. (*continued*)

Assembly Language

```
0000  120023          BR      main
            true:     .EQUATE 1
            false:    .EQUATE 0
            ;
            LOWER:    .EQUATE 21          ;const int
            UPPER:    .EQUATE 65          ;const int
            ;
            ;******* bool inRange(int a)
            retVal:   .EQUATE 4           ;returned value #2d
            a:        .EQUATE 2           ;formal parameter #2d
0003  C00015 inRange: LDWA    LOWER,i     ;if ((LOWER <= a)
0006  A30002 if:      CPWA    a,s
0009  1E001C          BRGT    else
000C  C30002          LDWA    a,s         ;&& (a <= UPPER))
000F  A00041          CPWA    UPPER,i
0012  1E001C          BRGT    else
0015  C00001 then:    LDWA    true,i      ;return true
0018  E30004          STWA    retVal,s
001B  01              RET
001C  C00000 else:    LDWA    false,i     ;return false
001F  E30004          STWA    retVal,s
0022  01              RET
            ;
            ;******* main()
            age:      .EQUATE 0           ;local variable #2d
0023  580002 main:    SUBSP   2,i         ;push #age
0026  330000          DECI    age,s       ;scanf("%d", &age)
0029  C30000          LDWA    age,s       ;move age
002C  E3FFFC          STWA    -4,s
002F  580004          SUBSP   4,i         ;push #retVal #a
0032  240003          CALL    inRange     ;inRange(age)
0035  500004          ADDSP   4,i         ;pop #a #retVal
0038  C3FFFE          LDWA    -2,s        ;if (inRange(age))
003B  180044          BREQ    else2
003E  49004B then2:   STRO    msg1,d      ;printf("Qualified\n")
0041  120047          BR      endif2
```

```
0044  490056 else2:   STRO    msg2,d      ;printf("Unqualified\n");
0047  500002 endif2:  ADDSP   2,i         ;pop #age
004A  00               STOP
004B  517561 msg1:     .ASCII  "Qualified\n\x00"
      ...
0056  556E71 msg2:     .ASCII  "Unqualified\n\x00"
      ...
0063                   .END
```

Representing false and true at the bit level as 0000 and 0001 (hex) has advantages and disadvantages. Consider the logical operations on Boolean quantities and the corresponding assembly instructions `ANDr`, `ORr`, and `NOTr`. If `p` and `q` are global Boolean variables, then

```
p && q
```

translates to

```
LDWA p,d
ANDA q,d
```

If you AND 0000 and 0001 with this object code, you get 0000 as desired. The OR operation, `||`, also works as desired. The NOT operation is a problem, however, because if you apply NOT to 0000, you get FFFF instead of 0001. Also, applying NOT to 0001 gives FFFE instead of 0000. Consequently, the compiler does not generate the `NOT` instruction when it translates the C assignment statement

```
p = !q
```

Instead, it uses the exclusive-or operation XOR, which has the mathematical symbol $\oplus$. It has the useful property that if you take the XOR of any bit value b with 0, you get b. And if you take the XOR of any bit value b with 1, you get the logical negation of b. Mathematically,

$$b \oplus 0 = b$$
$$b \oplus 1 = \neg\, b$$

Unfortunately, the Pep/9 computer does not have an XORr instruction in its instruction set. If it did have such an instruction, the compiler would generate the following code for the above assignment:

```
LDWA q,d
XORA 0x0001,i
STWA p,d
```

If q is false, it has the representation 0000 (hex), and 0000 XOR 0001 equals 0001, as desired. Also, if q is true, it has the representation 0001 (hex), and 0001 XOR 0001 equals 0000.

The type `bool` was not included in the C-language standard library until 1999. Older compilers use the convention that the Boolean operators operate on integers. They interpret the integer value 0 as false and any nonzero integer value as true. To preserve backward compatibility, current C compilers maintain this convention.

6.4 Indexed Addressing and Arrays

A variable at Level HOL6 is a memory cell at Level ISA3. A variable at Level HOL6 is referred to by its name; at Level ISA3, by its address. A variable at Level Asmb5 can be referred to by its symbolic name, but the value of that symbol is the address of the cell in memory.

At Level Asmb5, the value of the symbol of an array is the address of the first cell of the array.

What about an array of values? An array contains many elements, and so consists of many memory cells. The memory cells of the elements are contiguous; that is, they are adjacent to one another. An array at Level HOL6 has a name. At Level Asmb5, the corresponding symbol is the address of the first cell of the array. This section shows how the compiler translates source programs that allocate and access elements of one-dimensional arrays. It does so with several forms of indexed addressing.

FIGURE 6.33 summarizes all the Pep/9 addressing modes. Previous programs illustrate immediate, direct, stack-relative, and stack-relative

FIGURE 6.33
The Pep/9 addressing modes.

Addressing Mode	aaa	Letters	Operand
Immediate	000	i	OprndSpec
Direct	001	d	Mem[OprndSpec]
Indirect	010	n	Mem[Mem[OprndSpec]]
Stack-relative	011	s	Mem[SP + OprndSpec]
Stack-relative deferred	100	sf	Mem[Mem[SP + OprndSpec]]
Indexed	101	x	Mem[OprndSpec + X]
Stack-indexed	110	sx	Mem[SP + OprndSpec + X]
Stack-deferred indexed	111	sfx	Mem[Mem[SP + OprndSpec] + X]

deferred addressing. Programs with arrays use indexed, stack-indexed, or stack-deferred indexed addressing. The column labeled *aaa* shows the address-aaa field at Level ISA3. The column labeled *Letters* shows the assembly language designation for the addressing mode at Level Asmb5. The column labeled *Operand* shows how the CPU determines the operand from the operand specifier (OprndSpec).

Translating Global Arrays

The C program in FIGURE 6.34 is the same as the one in Figure 2.15, except that the variables are global instead of local. It shows a program at Level HOL6 that declares a global array of four integers named `vector` and a global integer named `j`. The main program inputs four integers into the array with a `for` loop and outputs them in reverse order together with their indexes.

FIGURE 6.34
A global array.

High-Order Language

```
#include <stdio.h>

int vector[4];
int j;

int main() {
   for (j = 0; j < 4; j++) {
      scanf("%d", &vector[j]);
   }
   for (j = 3; j >= 0; j--) {
      printf("%d %d\n", j, vector[j]);
   }
   return 0;
}
```

Assembly Language

```
0000  12000D          BR      main
0003  000000 vector:  .BLOCK  8           ;global variable #2d4a
      000000
      0000
000B  0000   j:       .BLOCK  2           ;global variable #2d
             ;
             ;******* main()
```

(continues)

FIGURE 6.34
A global array. (*continued*)

```
000D  C80000 main:    LDWX    0,i         ;for (j = 0
0010  E9000B          STWX    j,d
0013  A80004 for1:    CPWX    4,i         ;j < 4
0016  1C0029          BRGE    endFor1
0019  0B              ASLX                ;two bytes per integer
001A  350003          DECI    vector,x    ;scanf("%d", &vector[j])
001D  C9000B          LDWX    j,d         ;j++)
0020  680001          ADDX    1,i
0023  E9000B          STWX    j,d
0026  120013          BR      for1
0029  C80003 endFor1: LDWX    3,i         ;for (j = 3
002C  E9000B          STWX    j,d
002F  A80000 for2:    CPWX    0,i         ;j >= 0
0032  160054          BRLT    endFor2
0035  39000B          DECO    j,d         ;printf("%d %d\n", j, vector[j])
0038  D00020          LDBA    ' ',i
003B  F1FC16          STBA    charOut,d
003E  0B              ASLX                ;two bytes per integer
003F  3D0003          DECO    vector,x
0042  D0000A          LDBA    '\n',i
0045  F1FC16          STBA    charOut,d
0048  C9000B          LDWX    j,d         ;j--)
004B  780001          SUBX    1,i
004E  E9000B          STWX    j,d
0051  12002F          BR      for2
0054  00      endFor2: STOP
0055                  .END
```

Input

```
60 70 80 90
```

Output

```
3 90
2 80
1 70
0 60
```

FIGURE 6.35 shows the memory allocation for integer `j` and array `vector`. As with all global integers, the compiler translates

```
int j;
```

at Level HOL6 as the following statement at Level Asmb5:

```
000B 0000 j: .BLOCK 2 ;global variable #2d
```

The compiler translates

```
int vector[4];
```

at Level HOL6 as the following statement at Level Asmb5:

```
0003  000000  vector:  .BLOCK  8  ;global variable #2d4a
      000000
      0000
```

It allocates eight bytes because the array contains four integers, each of which is two bytes. Figure 6.35 shows that 0003 is the address of the first element of the array. The second element is at 0005, and each element is at an address two bytes greater than the previous element.

FIGURE 6.35
Memory allocation for the global array of Figure 6.34.

Format trace tags for arrays

Format trace tags for arrays specify how many cells are in the array as well as the number of bytes. You should read the format trace tag `#2d4a` as "two byte decimal, four cell array." With this specification, the Pep/9 debugger will produce a figure similar to that of Figure 6.35 with each array cell individually labeled.

The compiler translates the first `for` statement

```
for (j = 0; j < 4; j++)
```

as usual. It accesses `j` with direct addressing because `j` is a global variable. But how does it access `vector[j]`? It cannot simply use direct addressing, because the value of symbol `vector` is the address of the first element of the array. If the value of `j` is 2, it should access the third element of the array, not the first.

The answer is that it uses indexed addressing. With indexed addressing, the CPU computes the operand as

Indexed addressing

Oprnd = Mem[OprndSpec + X]

It adds the operand specifier and the index register and uses the sum as the address in main memory from which it fetches the operand.

In Figure 6.34, the compiler translates

```
scanf("%d", &vector[j]);
```

at Level HOL6 as

```
0019  0B      ASLX            ;two bytes per integer
001A  350003  DECI  vector,x  ;scanf("%d", &vector[j])
```

at Level Asmb5. This is an optimized translation. The compiler analyzed the previous code generated and determined that the index register already contained the current value of `j`. A nonoptimizing compiler would generate the following code:

```
LDWX j,d
ASLX
DECI vector,x
```

Suppose the value of `j` is 2. `LDX` puts the value of `j` in the index register. (Or, an optimizing compiler determines that the current value of `j` is already in the index register.) `ASLX` multiplies the 2 times 2, leaving 4 in the index register. `DECI` uses indexed addressing. So, the operand is computed as

```
Mem[OprndSpec + X]
Mem[0003 + 4]
Mem[0007]
```

which Figure 6.35 shows is `vector[2]`. Had the array been an array of characters, the `ASLX` operation would be unnecessary because each character occupies only one byte. In general, if each cell in the array occupies *n* bytes, the value of `j` is loaded into the index register, multiplied by *n*, and the array element is accessed with indexed addressing.

Similarly, the compiler translates the output of `vector[j]` as

```
003E  0B      ASLX            ;two bytes per integer
003F  3D0003  DECO  vector,x
```

with indexed addressing.

In summary, to translate global arrays, the compiler generates code as follows:

The translation rules for global arrays

- To allocate storage for the array, it generates `.BLOCK` *tot*, where *tot* is the total number of bytes occupied by the array.
- To get an element of the array, it generates `LDWX` to put the index into the index register, generates code to multiply the index by the number of bytes per cell (`ASLX` in the case of an array of integers with two bytes per integer), and uses indexed addressing.

Translating Local Arrays

Like all local variables, local arrays are allocated on the run-time stack during program execution. The SUBSP instruction allocates the array and the ADDSP instruction deallocates it. FIGURE 6.36 is a program identical to the one of Figure 6.34 except that the index j and the array vector are local to main().

FIGURE 6.36
A local array. The C program is from Figure 6.34 but with local variables.

High-Order Language

```
#include <stdio.h>

int main() {
   int vector[4];
   int j;
   for (j = 0; j < 4; j++) {
      scanf("%d", &vector[j]);
   }
   for (j = 3; j >= 0; j--) {
      printf("%d %d\n", j, vector[j]);
   }
   return 0;
}
```

Assembly Language

```
0000  120003          BR      main
            ;
            ;******* main ()
            vector:   .EQUATE 2           ;local variable #2d4a
            j:        .EQUATE 0           ;local variable #2d
0003  58000A main:    SUBSP   10,i        ;push #vector #j
0006  C80000          LDWX    0,i         ;for (j = 0
0009  EB0000          STWX    j,s
000C  A80004 for1:    CPWX    4,i         ;j < 4
000F  1C0022          BRGE    endFor1
0012  0B              ASLX                ;two bytes per integer
0013  360002          DECI    vector,sx   ;scanf("%d", &vector[j])
0016  CB0000          LDWX    j,s         ;j++)
```

(continues)

FIGURE 6.36
A local array. The C program is from Figure 6.34 but with local variables. (*continued*)

```
0019  680001          ADDX    1,i
001C  EB0000          STWX    j,s
001F  12000C          BR      for1
0022  C80003 endFor1: LDWX    3,i         ;for (j = 3
0025  EB0000          STWX    j,s
0028  A80000 for2:    CPWX    0,i         ;j >= 0
002B  16004D          BRLT    endFor2
002E  3B0000          DECO    j,s         ;printf("%d %d\n", j, vector[j])
0031  D00020          LDBA    ' ',i
0034  F1FC16          STBA    charOut,d
0037  0B              ASLX                ;two bytes per integer
0038  3E0002          DECO    vector,sx
003B  D0000A          LDBA    '\n',i
003E  F1FC16          STBA    charOut,d
0041  CB0000          LDWX    j,s         ;j--)
0044  780001          SUBX    1,i
0047  EB0000          STWX    j,s
004A  120028          BR      for2
004D  50000A endFor2: ADDSP   10,i        ;pop #j #vector
0050  00              STOP
0051                  .END
```

FIGURE 6.37 shows the memory allocation on the run-time stack for the program of Figure 6.36. The compiler translates

```
int vector[4];
int j;
```

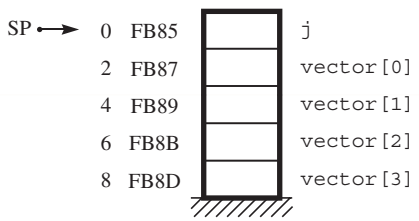
at Level HOL6 as

```
0003 58000A main: SUBSP 10,i ;push #vector #j
```

at Level Asmb5. It allocates eight bytes for `vector` and two bytes for `j`, for a total of 10 bytes. It sets the values of the symbols with

```
vector:   .EQUATE 2  ;local variable #2d4a
j:        .EQUATE 0  ;local variable #2d
```

FIGURE 6.37
Memory allocation for the local array of Figure 6.36.



where 2 is the stack-relative address of the first cell of `vector` and 0 is the stack-relative address of `j` as Figure 6.37 shows.

How does the compiler access `vector[j]`? It cannot use indexed addressing, because the value of symbol `vector` is not the address of the first element of the array. It uses stack-indexed addressing. With stack-indexed addressing, the CPU computes the operand as

Stack-indexed addressing

Oprnd = Mem[SP + OprndSpec + X]

It adds the stack pointer plus the operand specifier plus the index register and uses the sum as the address in main memory from which it fetches the operand.

In Figure 6.36, the compiler translates

```
scanf("%d", &vector[j]);
```

at Level HOL6 as

```
0012  0B      ASLX             ;two bytes per integer
0013  360002  DECI  vector,sx  ;scanf("%d", &vector[j])
```

at Level Asmb5. As in the previous program, this is an optimized translation. A nonoptimizing compiler would generate the following code:

```
LDWX j,d
ASLX
DECI vector,sx
```

Suppose the value of `j` is 2. `LDWX` puts the value of `j` in the index register. `ASLX` multiplies the 2 times 2, leaving 4 in the index register. `DECI` uses stack-indexed addressing. So, the operand is computed as

Mem[SP + OprndSpec + X]
Mem[FB85 + 2 + 4]
Mem[FB8B]

which Figure 6.37 shows is `vector[2]`. You can see how stack-indexed addressing is made for arrays on the run-time stack. SP is the address of the top of the stack. OprndSpec is the stack-relative address of the first cell of the array, so SP + OprndSpec is the absolute address of the first cell of the array. With `j` in the index register (multiplied by the number of bytes per cell of the array), the sum SP + OprndSpec + X is the address of cell `j` of the array.

In summary, to translate local arrays, the compiler generates code as follows:

The translation rules for local arrays

- To allocate storage for the array, it generates `SUBSP` *tot* with immediate addressing where *tot* is the total number of bytes occupied by the array.
- To get an element of the array, it generates `LDWX` to put the index into the index register, generates code to multiply the index by the number of bytes per cell (`ASLX` in the case of an array of integers with two bytes per integer), and uses stack-indexed addressing.

Translating Arrays Passed as Parameters

In C, the name of an array without the square brackets, `[]`, is the address of the first element of the array. When you pass an array, even if you do not use the `&` designation in the actual parameter list, you are passing the address of the first element of the array. The effect is as if you called the array by reference. The designers of the C language reasoned that programmers almost never want to pass an array by value because such calls are so inefficient. They require large amounts of storage on the run-time stack because the stack must contain the entire array. And they require a large amount of time because the value of every cell must be copied onto the stack. Consequently, the default behavior in C is for arrays to be called as if by reference.

FIGURE 6.38 shows how a compiler translates a program that passes a local array as a parameter. The main program passes an array of integers `vector` and an integer `numItms` to procedures `getVect()` and `putVect()`. `getVect()` inputs values into the array and sets `numItms` to the number of items input. `putVect()` outputs the values of the array.

Figure 6.38 shows that the compiler translates the local variables

```
int vector[8];
int numItms;
```

as

```
             vector:  .EQUATE 2     ;local variable #2d8a
             numItms: .EQUATE 0     ;local variable #2d
0057  580012 main:     SUBSP   18,i ;push #vector #numItms
```

FIGURE 6.38
Passing a local array as a parameter.

High-Order Language

```
#include <stdio.h>

void getVect(int v[], int *n) {
   int j;
   scanf("%d", n);
   for (j = 0; j < *n; j++) {
      scanf("%d", &v[j]);
   }
}

void putVect(int v[], int n) {
   int j;
   for (j = 0; j < n; j++) {
      printf("%d ", v[j]);
   }
   printf("\n");
}
int main() {
   int vector[8];
   int numItms;
   getVect(vector, &numItms);
   putVect(vector, numItms);
   return 0;
}
```

Assembly Language

```
0000  120058          BR      main
            ;
            ;******* getVect(int v[], int *n)
            v:        .EQUATE 6           ;formal parameter #2h
            n:        .EQUATE 4           ;formal parameter #2h
            j:        .EQUATE 0           ;local variable #2d
0003  580002 getVect: SUBSP   2,i         ;push #j
0006  340004          DECI    n,sf        ;scanf("%d", n)
0009  C80000          LDWX    0,i         ;for (j = 0
000C  EB0000          STWX    j,s
000F  AC0004 for1:    CPWX    n,sf        ;j < *n
```

(*continues*)

FIGURE 6.38
Passing a local array as a parameter. (*continued*)

```
0012  1C0025          BRGE    endFor1
0015  0B              ASLX                ;two bytes per integer
0016  370006          DECI    v,sfx       ;scanf("%d", &v[j])
0019  CB0000          LDWX    j,s         ;j++)
001C  680001          ADDX    1,i
001F  EB0000          STWX    j,s
0022  12000F          BR      for1
0025  500002 endFor1: ADDSP   2,i         ;pop #j
0028  01              RET
            ;
            ;******* putVect(int v[], int n)
            v2:       .EQUATE 6           ;formal parameter #2h
            n2:       .EQUATE 4           ;formal parameter #2d
            j2:       .EQUATE 0           ;local variable #2d
0029  580002 putVect: SUBSP   2,i         ;push #j2
002C  C80000          LDWX    0,i         ;for (j = 0
002F  EB0000          STWX    j2,s
0032  AB0004 for2:    CPWX    n2,s        ;j < n
0035  1C004E          BRGE    endFor2
0038  0B              ASLX                ;two bytes per integer
0039  3F0006          DECO    v2,sfx      ;printf("%d ", v[j])
003C  D00020          LDBA    ' ',i
003F  F1FC16          STBA    charOut,d
0042  CB0000          LDWX    j2,s        ;j++)
0045  680001          ADDX    1,i
0048  EB0000          STWX    j2,s
004B  120032          BR      for2
004E  D0000A endFor2: LDBA    '\n',i      ;printf("\n")
0051  F1FC16          STBA    charOut,d
0054  500002          ADDSP   2,i         ;pop #j2
0057  01              RET
            ;
            ;******* main()
            vector:   .EQUATE 2           ;local variable #2d8a
            numItms:  .EQUATE 0           ;local variable #2d
0058  580012 main:    SUBSP   18,i        ;push storage for #vector #numItms
005B  03              MOVSPA              ;move (&)vector
005C  600002          ADDA    vector,i
```

```
005F  E3FFFE          STWA    -2,s
0062  03              MOVSPA              ;move &numItms
0063  600000          ADDA    numItms,i
0066  E3FFFC          STWA    -4,s
0069  580004          SUBSP   4,i         ;push #v #n
006C  240003          CALL    getVect     ;getVect(vector, &numItms)
006F  500004          ADDSP   4,i         ;pop #n #v
0072  03              MOVSPA              ;move (&)vector
0073  600002          ADDA    vector,i
0076  E3FFFE          STWA    -2,s
0079  C30000          LDWA    numItms,s   ;move numItms
007C  E3FFFC          STWA    -4,s
007F  580004          SUBSP   4,i         ;push #v2 #n2
0082  240029          CALL    putVect     ;putVect(vector, numItms)
0085  500004          ADDSP   4,i         ;pop #n2 #v2
0088  500012          ADDSP   18,i        ;pop #numItms #vector
008B  00              STOP
008C                  .END
```

Input

```
5 40 50 60 70 80
```

Output

```
40 50 60 70 80
```

The `SUBSP` instruction pushes 18 bytes on the run-time stack, 16 bytes for the eight integers of the array, and 2 bytes for the integer. The `.EQUATE` dot commands set the symbols to their stack offsets, as FIGURE 6.39(a) shows.

The compiler translates

```
getVect(vector, &numItms);
```

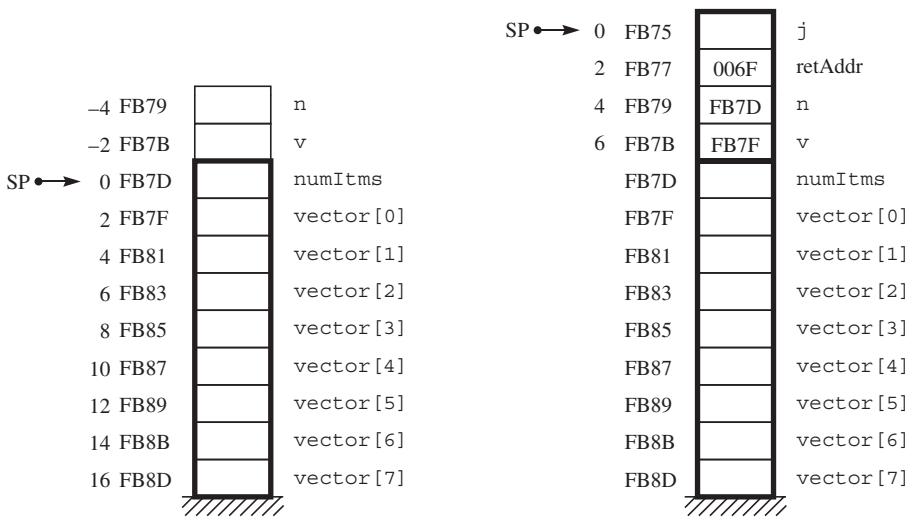
by first generating code to move the address of the first cell of `vector` to the stack

```
005B  03      MOVSPA            ;move (&)vector
005C  600002  ADDA    vector,i
005F  E3FFFE  STWA    -2,s
```

and then by generating code to move the address of `numItms` to the stack.

```
0062  03      MOVSPA            ;move &numItms
0063  600000  ADDA    numItms,i
0066  E3FFFC  STWA    -4,s
```

FIGURE 6.39
The run-time stack for the program of Figure 6.38.



(a) Before calling `getVect( )`. **(b)** After calling `getVect( )`.

Even though the first actual parameter in C is `vector` and not `&vector`, the compiler nevertheless writes code to push the address of `v` with the `MOVSPA` and `ADDA` instructions. As usual, the second actual parameter `&numItms` has the address operator `&` prefixed so the compiler will push its address the same way. Remember that arrays in C are a special case and are called by reference by default without using the `&` addressing operator in the actual parameter list. Figure 6.39(b) shows `v` with FB7F, the address of `vector[0]`, as well as `n` with FB79, the address of `numItms`.

Figure 6.39(b) also shows the stack offsets for the parameters and local variables in `getVect()`. The compiler defines the symbols

```
v: .EQUATE 6 ;formal parameter #2h
n: .EQUATE 4 ;formal parameter #2h
j: .EQUATE 0 ;local variable #2d
```

accordingly. It translates the input statement as

```
0006 340004 DECI n,sf ;scanf("%d", n)
```

where stack-relative deferred addressing is used because `n` is called by reference and the address of `n` is on the stack.

But how does the compiler translate

```
scanf("%d", &v[j]);
```

It cannot use stack-indexed addressing, because the array of values is not in the stack frame for `getVect()`. The value of `v` is 6, which means that the address of the first cell of the array is six bytes below the top of the stack. The array of values is in the stack frame for `main()`. Stack-deferred indexed addressing is designed to access the elements of an array whose address is in the top stack frame but whose actual collection of values is not. With stack-deferred indexed addressing, the CPU computes the operand as

Stack-deferred indexed addressing

Oprnd = Mem[Mem[SP + OprndSpec] + X]

It adds the stack pointer plus the operand specifier and uses the sum as the address of the first element of the array, to which it adds the index register. The compiler translates the input statement as

```
0015  0B      ASLX          ;two bytes per integer
0016  370006  DECI  v,sfx   ;scanf("%d", &v[j])
```

where the letters `sfx` indicate stack-deferred indexed addressing, and the compiler has determined that the index register will contain the current value of `j`.

For example, suppose the value of `j` is 2. The `ASLX` instruction doubles it to 4. The computation of the operand is

Mem[Mem[SP + OprndSpec] + X]
Mem[Mem[FB75 + 6] + 4]
Mem[Mem[FB7B] + 4]
Mem[FB7F + 4]
Mem[FB83]

which is `vector[2]`, as expected from Figure 6.39(b).

The formal parameters in procedures `getVect()` and `putVect()` in Figure 6.39 have the same names. At Level HOL6, the scope of the parameter names is confined to the body of the function. The programmer knows that a statement containing `n` in the body of `getVect()` refers to the `n` in the parameter list for `getVect()` and not to the `n` in the parameter list of `putVect()`. The scope of a symbol name at Level Asmb5, however, is the entire assembly language program. The compiler cannot use the same symbol for the `n` in `putVect()` that it uses for the `n` in `getVect()`, as duplicate symbol definitions would be ambiguous. All compilers must have some mechanism for managing the scope of name declarations in Level-HOL6 programs when they transform them to symbols at Level Asmb5. The compiler in Figure 6.38 makes the identifiers unambiguous by appending the digit 2 to the symbol name. Hence, the compiler translates variable name `n` in `putVect()` at Level HOL6 to symbol `n2` at Level Asmb5. It does the same with `v` and `j`.

With procedure `putVect()`, the array is passed as a parameter but `n` is called by value. In preparation for the procedure call, the address of `vector` is pushed onto the stack as before, but this time the value of `numItms` is pushed, not its address. In procedure `putVect()`, `n2` is accessed with stack-relative addressing

```
0032  AB0004  for2:  CPWX  n2,s  ;j < n
```

because it is called by value. `v2` is accessed with stack-deferred indexed addressing

```
0038  0B      ASLX          ;two bytes per integer
0039  3F0006  DECO  v2,sfx  ;printf("%d ", v[j])
```

as it is in `getVect()`.

In Figure 6.38, `vector` is a local array. If it were a global array, the translations of `getVect()` and `putVect()` would be unchanged. `v[j]` would still be accessed with stack-deferred indexed addressing, which expects the address of the first element of the array to be in the top stack frame. The only difference would be in the code to push the address of the first element of the array in preparation of the call. As in the program of Figure 6.34, the value of the symbol of a global array is the address of the first cell of the array. Consequently, to push the address of the first cell of the array, the compiler would generate an `LDWA` instruction with immediate addressing followed by an `STWA` instruction with stack-relative addressing to do the push.

In summary, to pass an array as a parameter, the compiler generates code as follows:

The translation rules for passing an array as a parameter

- To get the actual parameter in the caller, which is the address of the first element of the array, it generates either (a) `MOVSPA` followed by `ADDA` with immediate addressing for a local array, or (b) `LDWA` with immediate addressing for a global array.
- To get an element of the array in the callee, it generates `LDWX` to put the index into the index register, generates code to multiply the index by the number of bytes per cell (`ASLX` in the case of an array of integers with two bytes per integer), and uses stack-deferred indexed addressing.

Translating the Switch Statement

The program in FIGURE 6.40, which is also in Figure 2.12, shows how a compiler translates the C `switch` statement. It uses an interesting combination of indexed addressing with the unconditional branch, `BR`. The `switch` statement is not the same as a nested `if` statement. If a user

FIGURE 6.40
Translation of a `switch` statement. The C program is from Figure 2.12.

High-Order Language

```
#include <stdio.h>

int main() {
   int guess;
   printf("Pick a number 0..3: ");
   scanf("%d", &guess);
   switch (guess) {
      case 0: printf("Not close\n"); break;
      case 1: printf("Close\n"); break;
      case 2: printf("Right on\n"); break;
      case 3: printf("Too high\n");
   }
   return 0;
}
```

Assembly Language

```
0000  120003          BR      main
            ;
            ;******* main()
            guess:    .EQUATE 0           ;local variable #2d
0003  580002 main:    SUBSP   2,i         ;push #guess
0006  490034          STRO    msgIn,d     ;printf("Pick a number 0..3: ")
0009  330000          DECI    guess,s     ;scanf("%d", &guess)
000C  CB0000          LDWX    guess,s     ;switch (guess)
000F  0B              ASLX                ;two bytes per address
0010  130013          BR      guessJT,x
0013  001B   guessJT: .ADDRSS case0
0015  0021            .ADDRSS case1
0017  0027            .ADDRSS case2
0019  002D            .ADDRSS case3
001B  490049 case0:   STRO    msg0,d      ;printf("Not close\n")
001E  120030          BR      endCase     ;break
0021  490054 case1:   STRO    msg1,d      ;printf("Close\n")
0024  120030          BR      endCase     ;break
0027  49005B case2:   STRO    msg2,d      ;printf("Right on\n")
002A  120030          BR      endCase     ;break
```

(*continues*)

FIGURE 6.40
Translation of a `switch` statement. The C program is from Figure 2.12. (*continued*)

```
002D  490065 case3:   STRO    msg3,d      ;printf("Too high\n")
0030  500002 endCase: ADDSP   2,i         ;pop #guess
0033  00              STOP
0034  506963 msgIn:   .ASCII  "Pick a number 0..3: \x00"
      ...
0049  4E6F74 msg0:    .ASCII  "Not close\n\x00"
      ...
0054  436C6F msg1:    .ASCII  "Close\n\x00"
      ...
005B  526967 msg2:    .ASCII  "Right on\n\x00"
      ...
0065  546F6F msg3:    .ASCII  "Too high\n\x00"
      ...
006F                  .END

Symbol table
--------------------------------------
Symbol    Value        Symbol    Value
--------------------------------------
case0     001B         case1     0021
case2     0027         case3     002D
endCase   0030         guess     0000
guessJT   0013         main      0003
msg0      0049         msg1      0054
msg2      005B         msg3      0065
msgIn     0034
--------------------------------------
```

enters 2 for `guess`, the `switch` statement branches directly to the third alternative without comparing `guess` to 0 or 1. An array is a random access data structure because the indexing mechanism allows the programmer to access any element at random without traversing all the previous elements. For example, to access the third element of a vector of integers, you can write `vector[2]` directly without having to traverse `vector[0]` and `vector[1]` first. Main memory is, in effect, an array of bytes whose addresses correspond to the indexes of the array. To translate the `switch`

Jump tables

statement, the compiler allocates an array of addresses called a *jump table*. Each entry in the jump table is the address of the first statement of a section of code that corresponds to one of the cases of the `switch` statement. With indexed addressing, the program can branch directly to case 2.

Figure 6.40 shows the jump table at 0013 in the assembly language program. The code generated at 0013 is 001B, which is the address of the first statement of case 0. The code generated at 0015 is 0021, which is the address of the first statement of case 1, and so on. The compiler generates the jump table with `.ADDRSS` pseudo-ops. Every `.ADDRSS` command must be followed by a symbol. The code generated by `.ADDRSS` is the value of the symbol. For example, `case2` is a symbol whose value is 0027, the address of the code to be executed if `guess` has a value of 2. Therefore, the object code generated by

The `.ADDRSS` *pseudo-op*

```
.ADDRSS case2
```

at 0017 is 0027.

Suppose the user enters 2 for the value of `guess`. The statement

```
000C CB0000 LDWX guess,s ;switch (guess)
```

puts 2 in the index register. The statement

```
000F 0B ASLX ;two bytes per address
```

doubles the 2, leaving 4 in the index register. The statement

```
0010 130013 BR guessJT,x
```

is an unconditional branch with indexed addressing. The value of the operand specifier `guessJT` is 0013, the address of the first word of the jump table. For indexed addressing, the CPU computes the operand as

Indexed addressing

Oprnd = Mem[OprndSpec + X]

Therefore, the CPU computes

Mem[OprndSpec + X]
Mem[0013 + 4]
Mem[0017]
0027

as the operand. The RTL specification for the `BR` instruction is

PC ← Oprnd

and so the CPU puts 0027 in the program counter. Because of the von Neumann cycle, the next instruction to be executed is the one at address 0027, which is precisely the first instruction for case 2.

The `break` statement in C is translated as a `BR` instruction to branch to the end of the `switch` statement. If you omit the `break` in your C program, the compiler will omit the `BR` and control will fall through to the next case.

If the user enters a number not in the range 0..3, a run-time error will occur. For example, if the user enters 4 for `guess`, the `ASLX` instruction will multiply it by 2, leaving 8 in the index register, and the CPU will compute the operand as

```
Mem[OprndSpec + X]
Mem[0013 + 8]
Mem[001B]
4100
```

so the branch will be to memory location 4100 (hex). The problem is that the bits 001B were generated by the assembler for the `STRO` instruction and were never meant to be interpreted as a branch address. To prevent such indignities from happening to the user, C specifies that nothing should happen if the value of `guess` is not one of the cases. It also provides a `default` case for the `switch` statement to handle any case not encountered by the previous cases. The compiler must generate an initial conditional branch on `guess` to handle the values not covered by the other cases. The problems at the end of the chapter explore this characteristic of the `switch` statement.

Compiling to x86 Assembly Language

The C compiler in Microsoft's Visual Studio IDE normally compiles direct to machine code. However, it has the capability to output to an assembly language called *Microsoft Macro Assembler (MASM)* as an intermediate step, which you can then inspect and compare with your C source code. Following are some code fragments from the assembly language translation of the C program in Figure 6.10.

The program has a single global variable declared as

```
char letter;
```

in C. The compiler translates this declaration to

```
_DATA SEGMENT
COMM _letter:BYTE
_DATA ENDS
```

in MASM. It creates the assembly language symbol `_letter` to represent the global variable `letter` in C. Global variables are stored in a data segment, which is a fixed location in memory.

The compiler translates the loop test

```
while (letter != '*')
```

as shown in FIGURE 6.41(a).

As with the Pep/9 listing, the first column is the address in memory, the second column is the machine code in hexadecimal, the third column is the assembly language mnemonic, and the remaining columns specify one or more operands.

The `movsx` instruction is move with sign extend, which puts the value of `letter` in the EAX register. Rather than having different versions of `mov` for different sizes of operands, the language uses `PTR` to denote the number of bytes in the operand. The `cmp` instruction is the compare instruction, which compares

FIGURE 6.41
MASM translations from C.

```
0003a  0f be 05 00 00
       00 00 00       movsx eax, BYTE PTR _letter
00041  83 f8 2a       cmp   eax, 42 ; 0000002aH
00044  74 62          je    SHORT $LN3@main
```

(a) Translation of the `while` loop test.

```
0007c  a1 00 00 00 00 mov   eax, DWORD PTR _value
00081  50             push  eax
00082  e8 00 00 00 00 call  _printBar
00087  83 c4 04       add   esp, 4
```

(b) Translation of the `printBar()` function call.

```
0001e  c7 45 f8 01 00 ; k = 1
       00 00          mov  DWORD PTR _k$[ebp], 1
00025  eb 09          jmp  SHORT $LN3@printBar
$LN2@printBar:        ; k++
00027  8b 45 f8       mov  eax, DWORD PTR _k$[ebp]
0002a  83 c0 01       add  eax, 1
0002d  89 45 f8       mov  DWORD PTR _k$[ebp], eax
$LN3@printBar:        ; k <= n
00030  8b 45 f8       mov  eax, DWORD PTR _k$[ebp]
00033  3b 45 08       cmp  eax, DWORD PTR _n$[ebp]
00036  7f 19          jg   SHORT $LN1@printBar
```

(c) Translation of the `for` loop.

```
0004f eb d6           jmp SHORT $LN2@printBar
$LN1@printBar:
```

(d) The branch to the top of the loop.

the content of the EAX register with 42 (dec), which is 2A (hex), which is the ASCII character `*`. As with Pep/9, a semicolon is the beginning of a comment, which the compiler produced for the listing. Hexadecimal constants terminate with the letter H. The `je` instruction is *jump if equal*, which is equivalent to the Pep/9 `BREQ` instruction. The compiler generates the symbol `$LN3@main` as the target of the branch, which it defines later in the program.

The code fragment in Figure 6.41(b) is an example of MASM assembly code generated by the function call

```
printBar(value)
```

in the C program of Figure 6.21.

The `push` instruction automatically subtracts 4 from the stack pointer ESP when it pushes `_value` onto the run-time stack. The `add` instruction pops the bytes off the stack.

Although the preceding code fragment is close to the equivalent Pep/9 translation, the code fragment for the setup code at the beginning of `printBar()` is more complex. In addition to the stack pointer, there is a base pointer for the stack frame that must be managed. The compiler sets up values for formal

parameter `n` and local variable `k` in the text segment at the beginning of the `printBar()` function:

```
; COMDAT _printBar
_TEXT SEGMENT
_k$ = -8   ; size = 4
_n$ = 8    ; size = 4
_printBar PROC ; COMDAT
```

Unlike Pep/9, the offsets are relative to the base pointer, not the stack pointer, and can be negative. The setup code at the beginning of the function takes nine instructions and is not shown here.

The code fragment in Figure 6.41(c) is the compiler's translation of

```
for (k = 1; k <= n; k++)
```

inside the function.

The first statement initializes `k` to 1. `DWORD` designates double word, which is four bytes. MASM treats the brackets, `[]`, like the addition operator. Thus, the expression `_k$[ebp]` denotes an addressing mode where the operand is Mem[EBP + OprndSpec], with the value of `_k$` one of the operand specifiers. This addressing mode is equivalent to stack relative in Pep/9 but with the base pointer EBP instead of the stack pointer.

You can see the operand specifiers in the object code. The value of symbol `_k$` is –8 (dec), which is 1111 1000 (bin) or `f8` in the object code. Similarly, you can see that `_n$` is `08` in the object code.

`jmp` is the unconditional jump, equivalent to `BR` in Pep/9. `jg` is the jump if greater than, equivalent to `BRGT` in Pep/9. The two statements in Figure 6.41(d) are at the bottom of the loop. The first branches back to the test, and the second is the symbol that terminates the loop.

The x86 jump instructions use PC-relative addressing, which is common but not used in Pep/9 to keep the translations simple. With PC-relative addressing, the operand specifier is not the address of the target, but rather the offset from the current value of the PC necessary to reach the target. In other words, the operand is PC + OprndSpec.

Example 6.1 Consider the `jmp` instruction at 00025 in Figure 6.41(c). The instruction specifier is `eb` and the operand specifier is 09. The branch is not to Mem[09], but rather to Mem[PC + 09]. After the instruction has been fetched and the program counter incremented, the value of the program counter, EIP in the x86 architecture, is 00027. Thus, the branch is to Mem[00027 + 09], which is Mem[00030], where the addition is in hexadecimal. Sure enough, the target of the branch is `$LN3@printBar:` at 00030. ∎

Example 6.2 Consider the `jmp` instruction at 0004f in Figure 6.41(d). After the instruction has been fetched and the program counter incremented, the value of the program counter is 00051. From the object code listing, the operand specifier is d6. It is considered an eight-bit signed integer 1101 0110, and there is a 1 in the sign bit making it negative—namely –42 (dec). Adding –42 (dec) to 00051 (hex) with the proper base conversions yields the hexadecimal address 00027. As expected, the target of the branch is `$LN2@printBar:` at 00027 in Figure 6.41(c).

The x86 architecture, like Pep/9, describes a complex instruction set (CISC) machine. You can see from the preceding object code that x86 instructions come in many different sizes. Chapter 12 describes the MIPS machine, which is a reduced instruction set (RISC) machine. The primary design goal for RISC machines is for all instructions to have the same size. Chapter 12 shows how the MIPS machine also uses PC-relative addressing for its branch instructions. ∎

6.5 Dynamic Memory Allocation

Abstraction of control

The purpose of a compiler is to create a high level of abstraction for the programmer. For example, it lets the programmer think in terms of a single `while` loop instead of the detailed conditional branches at the assembly level that are necessary to implement the loop on the machine. Hiding the details of a lower level is the essence of abstraction.

Abstraction of data

But abstraction of program control is only one side of the coin. The other side is abstraction of data. At the assembly and machine levels, the only data types are bits and bytes. Previous programs show how the compiler translates character, integer, and array types. Each of these types can be global, allocated with `.BLOCK`, or local, allocated with `SUBSP` on the run-time stack. But C programs can also contain structures and pointers, the basic building blocks of many data structures. At Level HOL6, pointers access structures allocated from the heap with the `malloc()` function. This section shows the operation of a simple heap at Level Asmb5 and how the compiler translates programs that contain pointers and structures.

Translating Global Pointers

FIGURE 6.42 shows a C program with global pointers and its translation to Pep/9 assembly language. The C program is identical to the one in Figure 2.38. Figure 2.39 shows the allocation from the heap as the program executes at Level HOL6. The heap is a region of memory different from the stack. The

FIGURE 6.42
Translation of global pointers. The C program is from Figure 2.38.

High-Order Language

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

int *a, *b, *c;

int main() {
   a = (int *) malloc(sizeof(int));
   *a = 5;
   b = (int *) malloc(sizeof(int));
   *b = 3;
   c = a;
   a = b;
   *a = 2 + *c;
   printf("*a = %d\n", *a);
   printf("*b = %d\n", *b);
   printf("*c = %d\n", *c);
   return 0;
}
```

(continues)

FIGURE 6.42
Translation of global pointers. The C program is from Figure 2.38. (*continued*)

Assembly Language

```
0000  120009          BR      main
0003  0000    a:      .BLOCK  2           ;global variable #2h
0005  0000    b:      .BLOCK  2           ;global variable #2h
0007  0000    c:      .BLOCK  2           ;global variable #2h
              ;
              ;******* main ()
0009  C00002  main:   LDWA    2,i         ;a = (int *) malloc(sizeof(int))
000C  240073          CALL    malloc      ;allocate #2d
000F  E90003          STWX    a,d
0012  C00005          LDWA    5,i         ;*a = 5
0015  E20003          STWA    a,n
0018  C00002          LDWA    2,i         ;b = (int *) malloc(sizeof(int))
001B  240073          CALL    malloc      ;allocate #2d
001E  E90005          STWX    b,d
0021  C00003          LDWA    3,i         ;*b = 3
0024  E20005          STWA    b,n
0027  C10003          LDWA    a,d         ;c = a
002A  E10007          STWA    c,d
002D  C10005          LDWA    b,d         ;a = b
0030  E10003          STWA    a,d
0033  C00002          LDWA    2,i         ;*a = 2 + *c
0036  620007          ADDA    c,n
0039  E20003          STWA    a,n
003C  490061          STRO    msg0,d      ;printf("*a = %d\n", *a)
003F  3A0003          DECO    a,n
0042  D0000A          LDBA    '\n',i
0045  F1FC16          STBA    charOut,d
0048  490067          STRO    msg1,d      ;printf("*b = %d\n", *b)
004B  3A0005          DECO    b,n
004E  D0000A          LDBA    '\n',i
0051  F1FC16          STBA    charOut,d
0054  49006D          STRO    msg2,d      ;printf("*c = %d\n", *c)
0057  3A0007          DECO    c,n
005A  D0000A          LDBA    '\n',i
005D  F1FC16          STBA    charOut,d
0060  00              STOP
```

```
0061  2A6120 msg0:    .ASCII  "*a = \x00"
      3D2000
0067  2A6220 msg1:    .ASCII  "*b = \x00"
      3D2000
006D  2A6320 msg2:    .ASCII  "*c = \x00"
      3D2000
             ;
             ;******* malloc()
             ;        Precondition: A contains number of bytes
             ;        Postcondition: X contains pointer to bytes
0073  C9007D malloc:  LDWX    hpPtr,d     ;returned pointer
0076  61007D          ADDA    hpPtr,d     ;allocate from heap
0079  E1007D          STWA    hpPtr,d     ;update hpPtr
007C  01              RET
007D  007F    hpPtr:  .ADDRSS heap        ;address of next free byte
007F  00      heap:   .BLOCK  1           ;first byte in the heap
0080                  .END
```

Output

```
*a = 7
*b = 7
*c = 5
```

compiler, in cooperation with the operating system under which it runs, must generate code to perform the allocation and deallocation from the heap.

When you program with pointers in C, you allocate storage from the heap with the `malloc()` function. When your program no longer needs the storage that was allocated, you deallocate it with the `free()` function. It is possible to allocate several cells of memory from the heap and then deallocate one cell from the middle. The memory management algorithms must be able to handle that scenario. To keep things simple at this introductory level, the programs that illustrate the heap do not show the deallocation process. The heap is located in main memory at the end of the application program. Function `malloc()` works by allocating storage from the heap, so that the heap grows downward. Once memory is allocated, it can never be deallocated. This feature of the Pep/9 heap is unrealistic but easier to understand than if it were presented more realistically.

Simplification in the Pep/9 heap

The assembly language program in Figure 6.42 shows the heap starting at address 007F, which is the value of the symbol `heap`. The allocation

algorithm maintains a global pointer called `hpPtr`, which stands for *heap pointer*. The statement

```
007D 007F hpPtr: .ADDRSS heap ;address of next free byte
```

initializes `hpPtr` to the address of the first byte in the heap. The application supplies `malloc()` with the number of bytes needed. The `malloc()` function returns the current value of `hpPtr` and then increments it by the number of bytes requested. Hence, the invariant maintained by `malloc()` is that `hpPtr` points to the address of the next byte to be allocated from the heap.

The calling protocol for function `malloc()`

The calling protocol for `malloc()` is different from the calling protocol for other functions. With other functions, information is passed via parameters on the run-time stack. With `malloc()`, the application puts the number of bytes to be allocated in the accumulator and executes the `CALL` statement to invoke the function. The function puts the current value of `hpPtr` in the index register for the application. So, the precondition for the successful operation of `malloc()` is that the accumulator contains the number of bytes to be allocated from the heap. The postcondition is that the index register contains the address in the heap of the first byte allocated by `malloc()`.

The calling protocol for function `malloc()` is more efficient than the calling protocol for other functions. The implementation of `malloc()` requires only four lines of assembly language code, including the `RET` statement. The statement

```
0073 C9007D malloc: LDWX hpPtr,d ;returned pointer
```

puts the current value of the heap pointer in the index register. The statement

```
0076 61007D ADDA hpPtr,d ;allocate from heap
```

adds the number of bytes to be allocated to the heap pointer, and the statement

```
0079 E1007D STWA hpPtr,d ;update hpPtr
```

updates `hpPtr` to the address of the first unallocated byte in the heap.

This efficient calling protocol is possible for two reasons. First, there is no long parameter list as is possible with other functions. The application only needs to supply one value to `malloc()`. The calling protocol for other functions must be designed to handle arbitrary numbers of parameters. If a parameter list had, say, four parameters, there would not be enough registers in the Pep/9 CPU to hold them all. But the run-time stack can store an arbitrary number of parameters. Second, `malloc()` does not call any

other function. Specifically, it makes no recursive calls. The calling protocol for functions must be designed in general to allow for functions to call other functions or to call themselves recursively. The run-time stack is essential for such calls but unnecessary for `malloc()`.

FIGURE 6.43(a) shows the memory allocation for the C program at Level HOL6 just before the first `printf()` statement. It corresponds to Figure 2.39(h). Figure 6.43(b) shows the same memory allocation at Level Asmb5. Global pointers `a`, `b`, and `c` are stored at 0003, 0005, and 0007. As with all global variables, they are allocated with `.BLOCK` by the statements

```
0003 0000 a: .BLOCK 2 ;global variable #2h
0005 0000 b: .BLOCK 2 ;global variable #2h
0007 0000 c: .BLOCK 2 ;global variable #2h
```

Pointers are addresses.

A pointer at Level HOL6 is an address at Level Asmb5. Addresses occupy two bytes. Hence, each global pointer is allocated two bytes. Pointers have format trace tags `#2h` because pointers are addresses, typically displayed in hexadecimal.

The compiler translates the statement

```
a = (int *) malloc(sizeof(int));
```

as

```
0009  C00002  main:  LDWA  2,i     ;a = (int *) malloc …
000C  240073         CALL  malloc  ;allocate #2d
000F  E90003         STWX  a,d
```

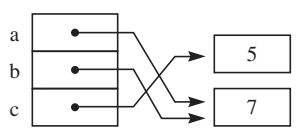
The `LDWA` instruction puts 2 in the accumulator. The `CALL` instruction calls the `malloc()` function, which allocates two bytes of storage from the heap and puts the pointer to the allocated storage in the index register. The comment has format trace tag `#2d` because the cell pointed to by `a` contains a two-byte decimal value. The `STWX` instruction stores the returned pointer in the global variable `a`. Because `a` is a global variable, `STWX` uses direct addressing. After this sequence of statements executes, `a` has the value 007F, and `hpPtr` has the value 0081 because it has been incremented by two.

How does the compiler translate

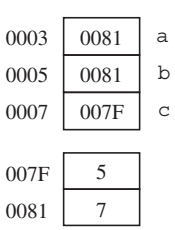
```
*a = 5;
```

At this point in the execution of the program, the global variable `a` has the address of where the 5 should be stored. (This point does *not* correspond to Figure 6.43, which is later in the execution.) The store word instruction cannot use direct addressing to put 5 in `a`, as that would replace the address

FIGURE 6.43
Memory allocation for Figure 6.42 just before the first `printf()` statement.



(a) Global pointers at Level HOL6.



(b) The same global pointers at Level Asmb5.

with 5, which is not the address of the allocated cell in the heap. Pep/9 provides the indirect addressing mode, in which the operand is computed as

Indirect addressing

Oprnd = Mem[Mem[OprndSpec]]

With indirect addressing, the operand specifier is the address in memory of the address of the operand. The compiler translates the assignment statement as

```
0012  C00005  LDWA  5,i  ;*a = 5
0015  E20003  STWA  a,n
```

where `n` in the store word instruction indicates indirect addressing. At this point in the execution, the operand is computed as

Mem[Mem[OprndSpec]]
Mem[Mem[0003]]
Mem[007F]

which is the first cell in the heap. The store word instruction stores 5 in main memory at address 007F.

The compiler translates the assignment of global pointers the same as it would translate the assignment of any other type of global variable. It translates

```
c = a;
```

as

```
0027  C10003  LDWA  a,d  ;c = a
002A  E10007  STWA  c,d
```

using direct addressing. At this point in the program, `a` contains 007F, the address of the first cell in the heap. The assignment gives `c` the same value, the address of the first cell in the heap, so that `c` points to the same cell to which `a` points.

Contrast the access of a global pointer to the access of the cell to which it points. The compiler translates

```
*a = 2 + *c;
```

as

```
0033  C00002  LDWA  2,i  ;*a = 2 + *c
0036  620007  ADDA  c,n
0039  E20003  STWA  a,n
```

where the add and store word instructions use indirect addressing. Whereas access to a global pointer uses direct addressing, access to the cell to which it points uses indirect addressing. You can see that the same principle applies to the translation of the `printf()` statement. Because `printf()` outputs

*a—that is, the cell to which a points—the DECO instruction at 003F uses indirect addressing.

In summary, to translate a global pointer, the compiler generates code as follows:

The translation rules for global pointers

- To allocate storage for the pointer, it generates .BLOCK 2 because an address occupies two bytes.
- To get the pointer, it generates LDWA with direct addressing.
- To get the content of the cell pointed to by the pointer, it generates LDWA or LDBA, depending on the type in the cell with indirect addressing.

Translating Local Pointers

The program in FIGURE 6.44 is the same as the program in Figure 6.42 except that the pointers a, b, and c are declared to be local instead of global. There is no difference in the output of the program compared to the program where the pointers are declared to be global. However, the memory model is quite different because the pointers are allocated on the run-time stack.

FIGURE 6.44
Translation of local pointers.

High-Order Language

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

int main() {
   int *a, *b, *c;
   a = (int *) malloc(sizeof(int));
   *a = 5;
   b = (int *) malloc(sizeof(int));
   *b = 3;
   c = a;
   a = b;
   *a = 2 + *c;
   printf("*a = %d\n", *a);
   printf("*b = %d\n", *b);
   printf("*c = %d\n", *c);
   return 0;
}
```

(continues)

FIGURE 6.44
Translation of local pointers. (*continued*)

Assembly Language

```
0000  120003          BR      main
              ;
              ;******* main()
              a:      .EQUATE 4           ;local variable #2h
              b:      .EQUATE 2           ;local variable #2h
              c:      .EQUATE 0           ;local variable #2h
0003  580006 main:    SUBSP   6,i         ;push #a #b #c
0006  C00002          LDWA    2,i         ;a = (int *) malloc(sizeof(int))
0009  240073          CALL    malloc      ;allocate #2d
000C  EB0004          STWX    a,s
000F  C00005          LDWA    5,i         ;*a = 5
0012  E40004          STWA    a,sf
0015  C00002          LDWA    2,i         ;b = (int *) malloc(sizeof(int))
0018  240073          CALL    malloc      ;allocate #2d
001B  EB0002          STWX    b,s
001E  C00003          LDWA    3,i         ;*b = 3
0021  E40002          STWA    b,sf
0024  C30004          LDWA    a,s         ;c = a
0027  E30000          STWA    c,s
002A  C30002          LDWA    b,s         ;a = b
002D  E30004          STWA    a,s
0030  C00002          LDWA    2,i         ;*a = 2 + *c
0033  640000          ADDA    c,sf
0036  E40004          STWA    a,sf
0039  490061          STRO    msg0,d      ;printf("*a = %d\n", *a)
003C  3C0004          DECO    a,sf
003F  D0000A          LDBA    '\n',i
0042  F1FC16          STBA    charOut,d
0045  490067          STRO    msg1,d      ;printf("*b = %d\n", *b)
0048  3C0002          DECO    b,sf
004B  D0000A          LDBA    '\n',i
004E  F1FC16          STBA    charOut,d
0051  49006D          STRO    msg2,d      ;printf("*c = %d\n", *c)
0054  3C0000          DECO    c,sf
0057  D0000A          LDBA    '\n',i
005A  F1FC16          STBA    charOut,d
005D  500006          ADDSP   6,i         ;pop #c #b #a
```

```
0060  00              STOP
0061  2A6120 msg0:    .ASCII  "*a = \x00"
      3D2000
0067  2A6220 msg1:    .ASCII  "*b = \x00"
      3D2000
006D  2A6320 msg2:    .ASCII  "*c = \x00"
      3D2000
            ;
            ;******* malloc()
            ;        Precondition: A contains number of bytes
            ;        Postcondition: X contains pointer to bytes
0073  C9007D malloc:  LDWX    hpPtr,d     ;returned pointer
0076  61007D          ADDA    hpPtr,d     ;allocate from heap
0079  E1007D          STWA    hpPtr,d     ;update hpPtr
007C  01              RET
007D  007F   hpPtr:   .ADDRSS heap        ;address of next free byte
007F  00     heap:    .BLOCK  1           ;first byte in the heap
0080                  .END
```

FIGURE 6.45 shows the memory allocation for the program in Figure 6.44 just before execution of the first `printf()` statement. As with all local variables, `a`, `b`, and `c` are allocated on the run-time stack. Figure 6.44(b) shows their offsets from the top of the stack as 4, 2, and 0. Consequently, the compiler translates

```
int *a, *b, *c;
```

as

```
a: .EQUATE 4  ;local variable #2h
b: .EQUATE 2  ;local variable #2h
c: .EQUATE 0  ;local variable #2h
```

Because `a`, `b`, and `c` are local variables, the compiler generates code to allocate storage for them with `SUBSP` and deallocates storage with `ADDSP`.

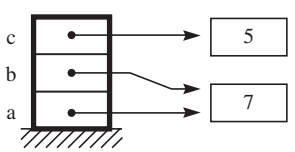
The compiler translates

```
a = (int *) malloc(sizeof(int));
```

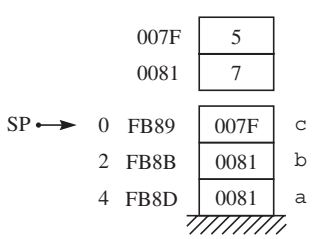
as

```
0006 C00002 LDWA  2,i     ;a = (int *) malloc(sizeof(int))
0009 240073 CALL  malloc  ;allocate #2d
000C EB0004 STWX  a,s
```

FIGURE 6.45
Memory allocation for Figure 6.44 just before the first `printf()` statement.



(a) Local pointers at Level HOL6.



(b) The same local pointers at Level Asmb5.

The LDWA instruction puts 2 in the accumulator in preparation for calling the malloc() function, because an integer occupies two bytes. The CALL instruction invokes malloc(), which allocates the two bytes from the heap and puts their address in the index register. In general, assignments to local variables use stack-relative addressing. Therefore, the STWX instruction uses stack-relative addressing to assign the address to a.

How does the compiler translate the assignment

```
*a = 5;
```

a is a pointer, and the assignment gives 5 to the cell to which a points. a is also a local variable. This situation is identical to the one where a parameter is called by reference in the programs of Figures 6.27 and 6.29. Namely, the address of the operand is on the run-time stack. The compiler translates the assignment statement as

```
000F  C00005  LDWA  5,i   ;*a = 5
0012  E40004  STWA  a,sf
```

where the store instruction uses stack-relative deferred addressing.

The compiler translates the assignment of local pointers the same as it would translate the assignment of any other type of local variable. It translates

```
c = a;
```

as

```
0024  C30004  LDWA  a,s  ;c = a
0027  E30000  STWA  c,s
```

using stack-relative addressing. At this point in the program, a contains 007F, the address of the first cell in the heap. The assignment gives c the same value, the address of the first cell in the heap, so that c points to the same cell to which a points.

The compiler translates

```
*a = 2 + *c;
```

as

```
0030  C00002  LDWA  2,i   ;*a = 2 + *c
0033  640000  ADDA  c,sf
0036  E40004  STWA  a,sf
```

where the add instruction uses stack-relative deferred addressing to access the cell to which c points and the store instruction uses stack-relative deferred addressing to access the cell to which a points. The same principle applies

to the translation of `printf()` statements where the `DECO` instructions also use stack-relative deferred addressing.

In summary, to access a local pointer, the compiler generates code as follows:

The translation rules for local pointers

- To allocate storage for the pointer, it generates `SUBSP` with two bytes for each pointer because an address occupies two bytes.
- To get the pointer, it generates `LDWA` with stack-relative addressing.
- To get the content of the cell pointed to by the pointer, it generates `LDWA` with stack-relative deferred addressing.

Translating Structures

Structures are the key to data abstraction at Level HOL6, the high-order languages level. They let the programmer consolidate variables with primitive types into a single abstract data type. The compiler provides the `struct` construct at Level HOL6. At Level Asmb5, the assembly level, a structure is a contiguous group of bytes, much like the bytes of an array. However, all cells of an array must have the same type and, therefore, the same size. Each cell is accessed by the numeric integer value of the index.

Fields in a structure

With a structure, the cells can have different types and, therefore, different sizes. The C programmer gives each cell, called a *field*, a field name. At Level Asmb5, the field name corresponds to the offset of the field from the first byte of the structure. The field name of a structure corresponds to the index of an array. It should not be surprising that the fields of a structure are accessed much like the elements of an array. Instead of putting the index of the array in the index register, the compiler generates code to put the field offset from the first byte of the structure in the index register. Apart from this difference, the remaining code for accessing a field of a structure is identical to the code for accessing an element of an array.

FIGURE 6.46 shows a program that declares a `struct` named `person` that has four fields named `first`, `last`, `age`, and `gender`. It is identical to the program in Figure 2.40. The program declares a global variable name `bill` that has type `person`. FIGURE 6.47 shows the storage allocation for the structure at Levels HOL6 and Asmb5. Fields `first`, `last`, and `gender` have type `char` and occupy one byte each. Field `age` has type `int` and occupies two bytes. Figure 6.47(b) shows the address of each field of the structure. To the left of the address is the offset from the first byte of the structure. The offset of a structure is similar to the offset of an element on the stack except that there is no pointer to the top of the structure that corresponds to SP.

FIGURE 6.46
Translation of a structure. The C program is from Figure 2.40.

High-Order Language

```
#include <stdio.h>

struct person {
   char first;
   char last;
   int age;
   char gender;
};
struct person bill;

int main() {
   scanf("%c%c%d %c", &bill.first, &bill.last, &bill.age, &bill.gender);
   printf("Initials: %c%c\n", bill.first, bill.last);
   printf("Age: %d\n", bill.age);
   printf("Gender: ");
   if (bill.gender == 'm') {
      printf("male\n");
   }
   else {
      printf("female\n");
   }
   return 0;
}
```

Assembly Language

```
0000  120008          BR      main
            first:    .EQUATE 0  ;struct field #1c
            last:     .EQUATE 1  ;struct field #1c
            age:      .EQUATE 2  ;struct field #2d
            gender:   .EQUATE 4  ;struct field #1c
0003  000000 bill:    .BLOCK  5  ;globals #first #last #age #gender
      0000
            ;
            ;******* main()
0008  C80000 main:    LDWX    first,i     ;scanf("%c%c%d %c",
000B  D1FC15          LDBA    charIn,d    ;&bill.first,
000E  F50003          STBA    bill,x
0011  C80001          LDWX    last,i      ;&bill.last,
```

```
0014  D1FC15          LDBA    charIn,d
0017  F50003          STBA    bill,x
001A  C80002          LDWX    age,i       ;&bill.age,
001D  350003          DECI    bill,x
0020  C80004          LDWX    gender,i    ;&bill.gender)
0023  D1FC15          LDBA    charIn,d
0026  F50003          STBA    bill,x
0029  49006C          STRO    msg0,d      ;printf("Initials: %c%c\n",
002C  C80000          LDWX    first,i     ;bill.first,
002F  D50003          LDBA    bill,x
0032  F1FC16          STBA    charOut,d
0035  C80001          LDWX    last,i      ;bill.last)
0038  D50003          LDBA    bill,x
003B  F1FC16          STBA    charOut,d
003E  D0000A          LDBA    '\n',i
0041  F1FC16          STBA    charOut,d
0044  490077          STRO    msg1,d      ;printf("Age:  %d\n",
0047  C80002          LDWX    age,i       ;bill.age)
004A  3D0003          DECO    bill,x
004D  D0000A          LDBA    '\n',i
0050  F1FC16          STBA    charOut,d
0053  49007D          STRO    msg2,d      ;printf("Gender: ")
0056  C80004          LDWX    gender,i    ;if (bill.gender == 'm')
0059  D50003          LDBA    bill,x
005C  B0006D          CPBA    'm',i
005F  1A0068          BRNE    else
0062  490086          STRO    msg3,d      ;printf("male\n")
0065  12006B          BR      endIf
0068  49008C else:    STRO    msg4,d      ;printf("female\n")
006B  00      endIf:  STOP
006C  496E69 msg0:    .ASCII  "Initials: \x00"
      ...
0077  416765 msg1:    .ASCII  "Age: \x00"
      ...
007D  47656E msg2:    .ASCII  "Gender: \x00"
      ...
0086  6D616C msg3:    .ASCII  "male\n\x00"
      ...
008C  66656D msg4:    .ASCII  "female\n\x00"
      ...
0094                  .END
```

(*continues*)

FIGURE 6.46
Translation of a structure. The C program is from Figure 2.40. (*continued*)

Input
```
bj 32 m
```

Output
```
Initials: bj
Age: 32
Gender: male
```

FIGURE 6.47
Memory allocation for Figure 6.46 just after the `scanf()` statement.

bill.first	b
bill.last	j
bill.age	32
bill.gender	m

(a) A global structure at Level HOL6.

0	0003	b
1	0004	j
2	0005	32
4	0007	m

(b) The same global structure at Asmb5.

The compiler translates

```
struct person {
   char first;
   char last;
   int age;
   char gender;
};
```

with equate dot commands as

```
first:    .EQUATE 0   ;struct field #1c
last:     .EQUATE 1   ;struct field #1c
age:      .EQUATE 2   ;struct field #2d
gender:   .EQUATE 4   ;struct field #1c
```

The name of a field equates to the offset of that field from the `first` byte of the structure. `first` equates to 0 because it is the first byte of the structure. `last` equates to 1 because `first` occupies one byte. `age` equates to 2 because `first` and `last` occupy a total of two bytes. And `gender` equates to 4 because `first`, `last`, and `age` occupy a total of four bytes. The compiler translates the global variable

```
person bill;
```

as

```
0003 000000 bill: .BLOCK 5 ;globals #first #last…
     0000
```

To access a field of a global structure, the compiler generates code to load the index register with the offset of the field from the first byte of the

structure. It accesses the field as it would the cell of a global array using indexed addressing. For example, the compiler translates the `scanf()` of `&bill.age` as

```
001A  C80002  LDWX  age,i   ;&bill.age,
001D  350003  DECI  bill,x
```

The load instruction uses immediate addressing to load the offset of field `age` into the index register. The decimal input instruction uses indexed addressing to access the field.

The compiler translates

```
if (bill.gender == 'm')
```

similarly as

```
0056  C80004  LDWX  gender,i ;if (bill.gender == 'm')
0059  D50003  LDBA  bill,x
005C  B0006D  CPBA  'm',i
```

The load word instruction puts the offset of the `gender` field into the index register. The load byte instruction accesses the field of the structure with indexed addressing and puts it into the rightmost byte of the accumulator. Finally, the compare instruction compares `bill.gender` with the letter `m`.

In summary, to access a global structure, the compiler generates code as follows:

The translation rules for global structures

- It equates each field of the structure to its offset from the first byte of the structure.
- To allocate storage for the structure, it generates `.BLOCK` *tot* where *tot* is the total number of bytes occupied by the structure.
- To get a field of the structure, it generates `LDWX` to load the offset of the field into the index register with immediate addressing, followed by an `LDBA` or `LDWA` instruction with indexed addressing.

The translation rules for local structures

In the same way that accessing the field of a global structure is similar to accessing the element of a global array, accessing the field of a local structure is similar to accessing the element of a local array. Local structures are allocated on the run-time stack. The name of each field equates to its offset from the first byte of the structure. The name of the local structure equates to its offset from the top of the stack. The compiler generates `SUBSP` to allocate storage for the structure and any other local variables, and `ADDSP` to deallocate storage. It accesses a field of the structure by loading the offset of the field into the index register

with immediate addressing followed by an instruction with stack-indexed addressing. Translating a program with a local structure is a problem for the student at the end of this chapter.

Translating Linked Data Structures

Programmers frequently combine pointers and structures to implement linked data structures. The `struct` is usually called a *node*, a pointer points to a node, and the node has a field that is a pointer. The pointer field of the node serves as a link to another node in the data structure. FIGURE 6.48 is

FIGURE 6.48
Translation of a linked list. The C program is from Figure 2.42.

High-Order Language

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

struct node {
   int data;
   struct node *next;
};

int main() {
   struct node *first, *p;
   int value;
   first = 0;
   scanf("%d", &value);
   while (value != -9999) {
      p = first;
      first = (struct node *) malloc(sizeof(struct node));
      first->data = value;
      first->next = p;
      scanf("%d", &value);
   }
   for (p = first; p != 0; p = p->next) {
      printf("%d ", p->data);
   }
   return 0;
}
```

Assembly Language

```
0000  120003          BR      main
            data:     .EQUATE 0           ;struct field #2d
            next:     .EQUATE 2           ;struct field #2h
            ;
            ;******* main ()
            first:    .EQUATE 4           ;local variable #2h
            p:        .EQUATE 2           ;local variable #2h
            value:    .EQUATE 0           ;local variable #2d
0003  580006 main:     SUBSP   6,i         ;push #first #p #value
0006  C00000          LDWA    0,i         ;first = 0
0009  E30004          STWA    first,s
000C  330000          DECI    value,s     ;scanf("%d", &value);
000F  C30000 while:   LDWA    value,s     ;while (value != -9999)
0012  A0D8F1          CPWA    -9999,i
0015  18003F          BREQ    endWh
0018  C30004          LDWA    first,s     ;p = first
001B  E30002          STWA    p,s
001E  C00004          LDWA    4,i         ;first = (...) malloc(...)
0021  24006A          CALL    malloc      ;allocate #data #next
0024  EB0004          STWX    first,s
0027  C30000          LDWA    value,s     ;first->data = value
002A  C80000          LDWX    data,i
002D  E70004          STWA    first,sfx
0030  C30002          LDWA    p,s         ;first->next = p
0033  C80002          LDWX    next,i
0036  E70004          STWA    first,sfx
0039  330000          DECI    value,s     ;scanf("%d", &value)
003C  12000F          BR      while
003F  C30004 endWh:   LDWA    first,s     ;for (p = first
0042  E30002          STWA    p,s
0045  C30002 for:     LDWA    p,s         ;p != 0
0048  A00000          CPWA    0,i
004B  180066          BREQ    endFor
004E  C80000          LDWX    data,i      ;printf("%d ", p->data)
0051  3F0002          DECO    p,sfx
0054  D00020          LDBA    ' ',i
0057  F1FC16          STBA    charOut,d
005A  C80002          LDWX    next,i      ;p = p->next)
005D  C70002          LDWA    p,sfx
```

(*continues*)

FIGURE 6.48
Translation of a linked list. The C program is from Figure 2.42. (*continued*)

```
0060  E30002          STWA    p,s
0063  120045          BR      for
0066  500006 endFor:  ADDSP   6,i         ;pop #value #p #first
0069  00              STOP
              ;
              ;******* malloc()
              ;        Precondition: A contains number of bytes
              ;        Postcondition: X contains pointer to bytes
006A  C90074 malloc:  LDWX    hpPtr,d     ;returned pointer
006D  610074          ADDA    hpPtr,d     ;allocate from heap
0070  E10074          STWA    hpPtr,d     ;update hpPtr
0073  01              RET
0074  0076   hpPtr:   .ADDRSS heap        ;address of next free byte
0076  00     heap:    .BLOCK  1           ;first byte in the heap
0077                  .END
```

Input

```
10 20 30 40 -9999
```

Output

```
40 30 20 10
```

a program that implements a linked list data structure. It is identical to the program in Figure 2.42.

The compiler equates the fields of the struct

```
struct node {
   int data;
   node* next;
};
```

to their offsets from the first byte of the `struct`. `data` is the first field, with an offset of 0. `next` is the second field, with an offset of 2 because data occupies two bytes. The translation is

```
data:   .EQUATE 0  ;struct field #2d
next:   .EQUATE 2  ;struct field #2h
```

The compiler translates the local variables

```
node *first, *p;
int value;
```

as it does all local variables. It equates the variable names with their offsets from the top of the run-time stack. The translation is

```
first:   .EQUATE 4   ;local variable #2h
p:       .EQUATE 2   ;local variable #2h
value:   .EQUATE 0   ;local variable #2d
```

FIGURE 6.49(b) shows the offsets for the local variables. The compiler generates `SUBSP` at 0003 to allocate storage for the locals and `ADDSP` at 0066 to deallocate storage.

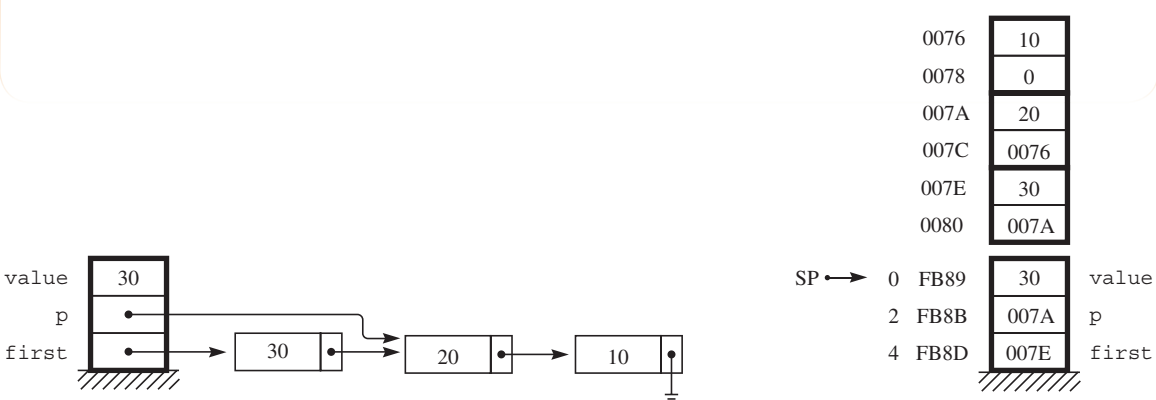
When you use `malloc()` in C, the computer must allocate enough memory from the heap to store the item to which the pointer points. In this program, a node occupies four bytes. Therefore, the compiler translates

```
first = (struct node *) malloc(sizeof(struct node));
```

by allocating four bytes in the code it generates to call `malloc()`. The translation is

```
001E  C00004  LDWA  4,i      ;first = (struct node *) …
0021  24006A  CALL  malloc   ;allocate #data #next
0024  EB0004  STWX  first,s
```

FIGURE 6.49
Memory allocation for Figure 6.48 just before scanning 40 from the input stream.



(a) The linked list at Level HOL6.

(b) The same linked list at Level Asmb5.

The load word instruction puts 4 in the accumulator in preparation for the call to `malloc()`. The call instruction calls `malloc()`, which puts the address of the first byte of the allocated node in the index register. When you allocate a structure, you supply symbol trace tags for the fields, `#data` and `#next` in this case, to be used by the symbolic debugger. The store word instruction completes the assignment to local variable `first` using stack-relative addressing.

How does the compiler generate code to access the field of a node to which a local pointer points? Remember that a pointer is an address. A local pointer implies that the address of the node is on the run-time stack. Furthermore, the field of a `struct` corresponds to the index of an array. If the address of the first cell of an array is on the run-time stack, you access an element of the array with stack-deferred indexed addressing. That is precisely how you access the field of a node. Instead of putting the value of the index in the index register, you put the offset of the field in the index register. The compiler translates

```
first->data = value;
```

as

```
0027  C30000  LDWA  value,s    ;first->data = value
002A  C80000  LDWX  data,i
002D  E70004  STWA  first,sfx
```

Similarly, it translates

```
first->next = p;
```

as

```
0030  C30002  LDWA  p,s        ;first->next = p
0033  C80002  LDWX  next,i
0036  E70004  STWA  first,sfx
```

To see how stack-deferred indexed addressing works for a local pointer to a node, remember that the CPU computes the operand as

Stack-deferred indexed addressing

Oprnd = Mem[Mem[SP + OprndSpec] + X]

It adds the stack pointer plus the operand specifier and uses the sum as the address of the first field, to which it adds the index register. Figure 6.49(b) shows the computation state just after `STWA` at 0036 executes with stack-deferred indexed addressing. The call to `malloc()` has returned the address of the newly allocated node, 007E, and stored it in `first`. The `LDWA` instruction at 0030 has put the value of `p`, 007A at this point in the program, in the accumulator. The `LDWX` instruction at 0033 has put the value of `next`,

offset 2, in the index register. The STWA instruction at 0036 executes with stack-deferred indexed addressing. The operand specifier is 4, the value of first. The computation of the operand is

Mem[Mem[SP + OprndSpec] + X]
Mem[Mem[FB89 + 4] + 2]
Mem[Mem[FB8D] + 2]
Mem[007E + 2]
Mem[0080]

which is the next field of the node to which first points.

In summary, to access a field of a node pointed to by a local pointer, the compiler generates code as follows:

The translation rules for accessing the field of a node to which a local pointer points

- To specify a field of a node, it generates .EQUATE to equate the offset of the field from the first byte of the node.
- To allocate storage for the node, it generates SUBSP *tot* with immediate addressing, where *tot* is the total number of bytes occupied by the structure.
- To get the field pointed to by p, it generates LDWX with stack-relative addressing to move the value of p into the index register, followed by LDWA or LDBA, depending on the type in the cell with stack-deferred indexed addressing.

You should be able to determine how the compiler translates programs with global pointers to nodes. Formulation of the translation rules is an exercise for the student at the end of this chapter. Translation of a C program that has global pointers to nodes is also a problem for the student.

Chapter Summary

A compiler uses conditional branch instructions at the machine level to translate if statements and loops at the high-order languages level. An if/else statement requires a conditional branch instruction to test the if condition and an unconditional branch instruction to branch around the else part. The translation of a while or do loop requires a branch to a previous instruction. The for loop requires, in addition, instructions to initialize and increment the control variable.

The structured programming theorem, proved by Bohm and Jacopini, states that any algorithm containing gotos, no matter how complicated or unstructured, can be written with only nested if

statements and `while` loops. The goto controversy was sparked by Dijkstra's famous letter, which stated that programs without gotos were not only possible but desirable.

The compiler allocates global variables at a fixed location in main memory. Procedures and functions allocate parameters and local variables on the run-time stack. Values are pushed onto the stack by incrementing the stack pointer (SP) and popped off the stack by decrementing SP. The subroutine call instruction pushes the contents of the program counter (PC), which acts as the return address, onto the stack. The subroutine return instruction pops the return address off the stack into the PC. Instructions access global values with direct addressing and values on the run-time stack with stack-relative addressing. A parameter that is called by reference has its address pushed onto the run-time stack. It is accessed with stack-relative deferred addressing. Boolean variables are stored with a value of 0 for false and a value of 1 for true.

Array values are stored in consecutive main memory cells. You access an element of a global array with indexed addressing, and an element of a local array with stack-indexed addressing. In both cases, the index register contains the index value of the array element, which must be multiplied by the number of bytes per cell. An array passed as a parameter always has the address of the first cell of the array pushed onto the run-time stack. You access an element of the array with stack-deferred indexed addressing. The compiler translates the `switch` statement with an array of addresses, each of which is the address of the first statement of a `case`. The array of addresses is called a *jump table*.

Pointer and `struct` types are common building blocks of data structures. A pointer is an address of a memory location in the heap. The `malaloc()` function allocates memory from the heap. You access a cell to which a global pointer points with indirect addressing. You access a cell to which a local pointer points with stack-relative deferred addressing. A `struct` has several named fields and is stored as a contiguous group of bytes. You access a field of a `global` struct with indexed addressing, with the index register containing the offset of the field from the first byte of the `struct`. Linked data structures commonly have a pointer to a `struct` called a *node*, which in turn contains a pointer to yet another node. If a local pointer points to a node, you access a field of the node with stack-deferred indexed addressing.

Exercises

Section 6.1

1. Explain the difference in the C memory model between global and local variables. How is each allocated and accessed?

Section 6.2

2. What is an optimizing compiler? When would you want to use one? When would you not want to use one? Explain.

***3.** The object code for Figure 6.14 has a `CPWA` at 000C to test the value of `j`. Because the program branches to that instruction from the bottom of the loop, why doesn't the compiler generate an `LDWA j,d` at that point before `CPA`?

4. Discover the function of the mystery program of Figure 6.16, and state in one short sentence what it does.

5. Read the papers by Bohm and Jacopini and by Dijkstra that are referred to in this chapter and write a summary of them.

Section 6.3

***6.** Draw the values just before and just after the second `CALL` at 001C of Figure 6.18 executes, as they are drawn in Figure 6.19.

7. Figure 6.26 is the run-time stack just after the second return. Draw the run-time stack, as in that figure, that corresponds to the time just before the second return.

Section 6.4

***8.** In the Pep/9 program of Figure 6.40, if you enter 4 for `Guess`, what statement executes after the branch at 0010? Why?

9. Section 6.4 does not show how to access an element from a two-dimensional array. Describe how a two-dimensional array might be stored and the assembly language object code that would be necessary to access an element from it.

Section 6.5

10. What are the translation rules for accessing the field of a node pointed to by a global pointer?

Problems

Section 6.2

11. Translate the following C program to Pep/9 assembly language.

```
#include <stdio.h>

int main() {
   int number;
   scanf("%d", &number);
   if (number % 2 == 0) {
      printf("Even\n");
   }
   else {
      printf("Odd\n");
   }
   return 0;
}
```

12. Translate the following C program to Pep/9 assembly language.

```
#include <stdio.h>

const int limit = 5;

int main() {
   int number;
   scanf("%d", &number);
   while (number < limit) {
      number++;
      printf("%d ", number);
   }
   return 0;
}
```

13. Translate the following C program to Pep/9 assembly language.

```
#include <stdio.h>

int main() {
   char ch;
   scanf("%c", &ch);
   if ((ch >= 'A') && (ch <= 'Z')) {
      printf("A");
   }
```

```
   else if ((ch >= 'a') && (ch <= 'z')) {
      printf("a");
   }
   else {
      printf("$");
   }
   printf("\n");
   return 0;
}
```

14. Translate the C program in Figure 6.12 to Pep/9 assembly language but with the `do` loop test changed to

```
    while (cop <= driver);
```

15. Translate the following C program to Pep/9 assembly language.

```
#include <stdio.h>

int main() {
   int numItms, j, data, sum;
   scanf("%d", &numItms);
   sum = 0;
   for (j = 1; j <= numItms; j++) {
      scanf("%d", &data);
      sum += data;
   }
   printf("Sum: %d\n", sum);
   return 0;
}
```

Sample Input

```
4 8 -3 7 6
```

Sample Output

```
Sum: 18
```

Section 6.3

16. Translate the following C program to Pep/9 assembly language.

```
#include <stdio.h>

int myAge;

void putNext(int age) {
   int nextYr;
   nextYr = age + 1;
```

```
   printf("Age: %d\n", age);
   printf("Age next year: %d\n", nextYr);
}

int main () {
   scanf("%d", &myAge);
   putNext(myAge);
   putNext(64);
   return 0;
}
```

17. Translate the C program in Problem 16 to Pep/9 assembly language, but declare `myAge` to be a local variable in `main()`.

18. Translate the following C program to Pep/9 assembly language. It multiplies two integers using a recursive shift-and-add algorithm. `mpr` stands for *multiplier* and `mcand` stands for *multiplicand*.

A recursive integer multiplication algorithm

```
#include <stdio.h>

int times(int mpr, int mcand) {
   if (mpr == 0) {
      return 0;
   }
   else if (mpr % 2 == 1) {
      return times(mpr / 2, mcand * 2) + mcand;
   }
   else {
      return times(mpr / 2, mcand * 2);
   }
}

int main() {
   int n, m;
   scanf("%d %d", &n, &m);
   printf("Product: %d\n", times(n, m));
   return 0;
}
```

19. Write a C program that converts an uppercase character to a lowercase character. Declare function

```
char toLower(char ch);
```

to do the conversion. If `ch` is not an uppercase character, the function should return `ch` unchanged. If it is an uppercase character, add the difference of `'a'` and `'A'` to `ch` as the return value. **(a)** Write your

program with a global variable for the actual parameter. Translate your C program to Pep/9 assembly language. **(b)** Write your program with a local variable for the actual parameter. Translate your C program to Pep/9 assembly language.

20. Write a C program that defines

```
int minimum(int j1, int j2)
```

which returns the smaller of `j1` and `j2`. **(a)** Write your program with a global variable for the actual parameter. Translate your C program to Pep/9 assembly language. **(b)** Write your program with a local variable for the actual parameter. Translate your C program to Pep/9 assembly language.

21. Translate to Pep/9 assembly language your C solution from Problem 2.13 that computes a Fibonacci term using a recursive function.

22. Translate to Pep/9 assembly language your C solution from Problem 2.14 that outputs the instructions for the Towers of Hanoi puzzle.

23. The recursive binomial coefficient function in Figure 6.25 can be simplified by omitting `y1` and `y2` as follows:

```
int binCoeff(int n, int k) {
   if ((k == 0) || (n == k)) {
      return 1;
   }
   else {
      return binCoeff(n - 1, k) + binCoeff(n - 1, k - 1);
   }
}
```

Write a Pep/9 assembly language program that calls this function. Keep the value returned from the `binCoeff(n - 1, k)` call on the stack, and push the actual parameters for the call to `binCoeff(n - 1, k - 1)` on top of it. FIGURE 6.50 shows a trace of the run-time stack where the stack frame contains four words (for retVal, `n`, `k`, and retAddr) and the shaded word is the value returned by a function call. The trace is for a call of `binCoeff(3,1)` from the main program.

24. Translate the following C program to Pep/9 assembly language. It multiplies two integers using an iterative shift-and-add algorithm.

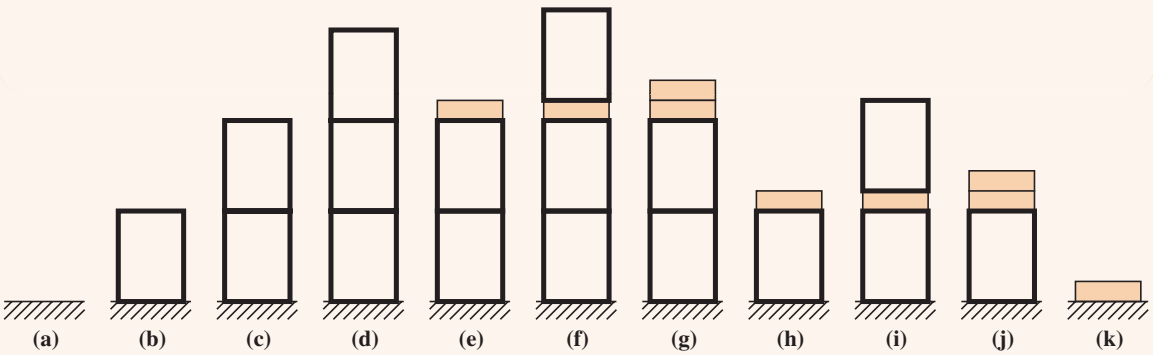
An iterative integer multiplication algorithm

```
#include <stdio.h>

int product, n, m;

void times(int *prod, int mpr, int mcand) {
```

FIGURE 6.50
Trace of the run-time stack for Figure 6.25.




```
    *prod = 0;
    while (mpr != 0) {
        if (mpr % 2 == 1) {
            *prod = *prod + mcand;
        }
        mpr /= 2;
        mcand *= 2;
    }
}

int main () {
    scanf("%d %d", &n, &m);
    times(&product, n, m);
    printf("Product: %d\n", product);
    return 0;
}
```

25. Translate the C program in Problem 24 to Pep/9 assembly language, but declare `product`, `n`, and `m` to be local variables in `main()`.

26. **(a)** Rewrite the C program of Figure 2.22 to compute the factorial recursively, but use procedure `times()` in Problem 24 to do the multiplication. Use one extra local variable in `fact()` to store the product. **(b)** Translate your C program to Pep/9 assembly language.

Section 6.4

27. Translate the following C program to Pep/9 assembly language:

```
#include <stdio.h>

int list[16];
```

```
int j, numItems;
int temp;

int main() {
   scanf("%d", &numItems);
   for (j = 0; j < numItems; j++) {
      scanf("%d", &list[j]);
   }
   temp = list[0];
   for (j = 0; j < numItems - 1; j++) {
      list[j] = list[j + 1];
   }
   list[numItems - 1] = temp;
   for (j = 0; j < numItems; j++) {
      printf("%d ", list[j]);
   }
   printf("\n");
   return 0;
}
```

Sample Input

```
5
11 22 33 44 55
```

Sample Output

```
22 33 44 55 11
```

The test in the second `for` loop is awkward to translate because of the arithmetic expression on the right side of the < operator. You can simplify the translation by transforming the test to the following mathematically equivalent test:

```
j + 1 < numItems;
```

28. Translate the C program in Problem 27 to Pep/9 assembly language, but declare `list`, `j`, `numItems`, and `temp` to be local variables in `main()`.

29. Translate the following C program to Pep/9 assembly language:

```
#include <stdio.h>

void getList(int ls[], int *n) {
   int j;
   scanf("%d", n);
   for (j = 0; j < *n; j++) {
```

```
      scanf("%d", &ls[j]);
   }
}

void putList(int ls[], int n) {
   int j;
   for (j = 0; j < n; j++) {
      printf("%d ", ls[j]);
   }
   printf("\n");
}

void rotate(int ls[], int n) {
   int j;
   int temp;
   temp = ls[0];
   for (j = 0; j < n - 1; j++) {
      ls[j] = ls[j + 1];
   }
   ls[n - 1] = temp;
}

int main() {
   int list[16];
   int numItems;
   getList(list, &numItems);
   putList(list, numItems);
   rotate(list, numItems);
   putList(list, numItems);
   return 0;
}
```

Sample Input

```
5
11 22 33 44 55
```

Sample Output

```
11 22 33 44 55
22 33 44 55 11
```

30. Translate the C program in Problem 29 to Pep/9 assembly language, but declare `list` and `numItems` to be global variables.

31. Translate to Pep/9 assembly language the C program from Figure 2.25 that adds four values in an array using a recursive function.

32. Translate to Pep/9 assembly language the C program from Figure 2.32 that reverses the elements of a local array using a recursive procedure. Initialize `word` to `"star"` with five pairs of load byte, store byte instructions (including the zero sentinel), and translate the `printf()` statement with `STRO` using stack-relative addressing.

33. Translate the following C program to Pep/9 assembly language:

```
#include <stdio.h>

int main () {
   int guess;
   printf("Pick a number 0..3: ");
   scanf("%d", &guess);
   switch (guess) {
      case 0: case 1: printf("Too low"); break;
      case 2: printf("Right on"); break;
      case 3: printf("Too high");
   }
   printf("\n");
   return 0;
}
```

The program is identical to Figure 6.40 except that two of the cases execute the same code. Your jump table must have exactly four entries, but your program must have only three case symbols and three cases.

34. Translate the following C program to Pep/9 assembly language.

```
#include <stdio.h>

int main () {
   int guess;
   printf("Pick a number 0..3: ");
   scanf("%d", &guess);
   switch (guess) {
      case 0: printf("Not close"); break;
      case 1: printf("Too low"); break;
      case 2: printf("Right on"); break;
      case 3: printf("Too high"); break;
      default: printf("Illegal input");
   }
```

```
    printf("\n");
    return 0;
}
```

Section 6.5

35. Translate to Pep/9 assembly language the C program from Figure 6.46 that accesses the fields of a structure, but declare `bill` as a local variable in `main()`.

36. Translate to Pep/9 assembly language the C program from Figure 6.48 that manipulates a linked list, but declare `first`, `p`, and `value` as global variables.

37. Insert the following C code fragment in `main()` of Figure 6.48 just before the `return` statement:

```
sum = 0; p = first;
while (p != 0) {
    sum += p->data;
    p = p->next;
}
printf("Sum: %d\n", sum);
```

and translate the complete program to Pep/9 assembly language. Declare `sum` to be a local variable along with the other locals as follows:

```
struct node *first, *p;
int value, sum;
```

38. Insert the following C code fragment between the declaration of `node` and `main()` in Figure 6.48:

```
void reverse(struct node *list) {
    if (list != 0) {
        reverse(list->next);
        printf("%d ", list->data);
    }
}
```

and the following code fragment in `main()` just before the `return` statement:

```
printf("\n");
reverse(first);
printf("\n");
```

Translate the complete C program to Pep/9 assembly language. The added code outputs the linked list in reverse order.

39. Insert the following C code fragment in `main()` of Figure 6.48 just before the `return` statement:

```
first2 = 0; p2 = 0;
for (p = first; p != 0; p = p->next) {
   p2 = first2;
   first2 = (struct node *) malloc(sizeof (struct node));
   first2->data = p->data;
   first2->next = p2;
}
for (p2 = first2; p2 != 0; p2 = p2->next) {
   printf("%d ", p2->data);
}
printf("\n");
```

Declare `first2` and `p2` to be local variables along with the other locals as follows:

```
struct node *first, *p, *first2, *p2;
int value;
```

Translate the complete program to Pep/9 assembly language. The added code creates a copy of the first list in reverse order and outputs it.

40. Translate to Pep/9 assembly language your C solution from Problem 2.18 that inputs an unordered list of integers with –9999 as a sentinel into a binary search tree, then outputs them with an inorder traversal of the tree.

41. This problem is a project to write a simulator in C for the Pep/9 computer.

(a) Write a loader that takes a Pep/9 object file in standard format and loads it into the main memory of a simulated Pep/9 computer. Declare main memory as an array of integers as follows:

```
int Mem[65536]; // Pep/9 main memory
```

Take your input as a string of characters from the standard input. Write a memory dump function that outputs the content of main memory as a sequence of decimal integers that represents the program. For example, if the input is

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

as in Figure 4.42, then the program should convert the hexadecimal numbers to integers and store them in the first 15 cells of `Mem`. The output should be the corresponding integer values as follows:

```
209 0 13 241 252 22 209 0 14 241 252 22 0 72 105
```

(b) Implement instructions `LDBr`, `STBr`, and `STOP` and addressing modes immediate and direct. Use Figure 4.32 as a guide for implementing the von Neumann execution cycle. If an instruction stores a byte to Mem[FC16], output the corresponding character to the standard output stream. If an instruction loads a byte from Mem[FC15], input the corresponding character from the standard input stream. For example, with the input as in part (a), the output should be `Hi`.

(c) Implement `DECO` as if it were a native instruction. That is, you should not implement the trap mechanism described in Section 8.2.

(d) Implement instructions `BR`, `LDWr`, `STWr`, `SUBSP`, and `ADDSP` and addressing mode stack relative. Test your implementation by assembling the program of Figure 6.1 with the Pep/9 assembler, then inputting the hexadecimal program into your simulator. The output should be `BMW335i`.

(e) Implement instructions `ADDr`, `SUBr`, `ASLr`, and `ASRr`. Implement instructions `DECI` and `STRO` as if they were native instructions. Take the input from the standard input stream and send your output to the standard output stream of C. Test your implementation by executing the program of Figure 6.4.

(f) Implement the conditional branch instructions `BRLE`, `BRLT`, `BREQ`, `BRNE`, `BRGE`, `BRGT`, `BRV`; the unary instructions `NOTr` and `NEGr`; and the compare instructions `CPWr` and `CPBr`. Test your implementation by executing the programs of Figures 6.6, 6.8, 6.10, 6.12, and 6.14.

(g) Implement instructions `CALL` and `RET`. Test your implementation by executing the programs of Figures 6.18, 6.21, 6.23, and 6.25.

(h) Implement instruction `MOVSPA` and addressing mode stack relative deferred. Test your implementation by executing the programs of Figures 6.27 and 6.29.

(i) Implement addressing modes indexed, stack-indexed, and stack-deferred indexed. Test your implementation by executing the programs of Figures 6.34, 6.36, 6.38, 6.40, and 6.48.

(j) Implement the indirect addressing mode. Test your implementation by executing the program of Figure 6.42.