Chapter 6

Compiling to the Assembly Level

The previous chapter describes the translation process from Pep/10 assembly language at Level Asmb5 to Pep/10 machine language at level ISA3. This chapter describes the translation process from C at level HOL6 to Pep/10 assembly language at Level Asmb5.

6.1 The Compilation Process

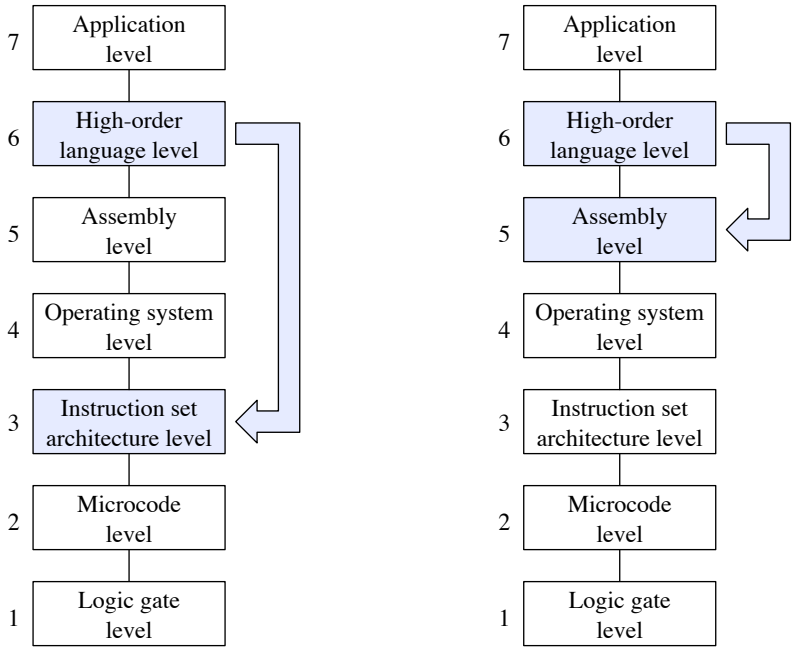
Compilers

A compiler translates a program in a high-order language (Level HOL6) into a lower-level language, so eventually it can be executed by the machine. There are two possibilities for the lower-level language.

Some compilers translate directly into machine language at level ISA3 as in Figure 6.1(a). Then the program can be loaded into memory and executed. Other compilers translate into assembly language at level Asmb5 as in Figure 6.1(b). An assembler then must translate the assembly language program into machine language before it can be loaded and executed.

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

Recall the C memory model described in Chapter 2. C has three kinds of variables—global variables, local variables, and dynamically allocated variables. The value of a variable is stored in the main memory of a computer, but where in memory it is stored depends on the kind of variable. There are three special sections of memory corresponding to the three kinds of variables:



(a) Translation directly to machine language.

(b) Translation to assembly language.

Figure 6.1 The function of a compiler.

- Global variables are stored at a fixed location in memory.
- Local variables and parameters are stored on the run-time stack.
- Dynamically allocated variables are stored on the heap.

Figure 6.2 shows each of these regions in the Pep/10 memory map. Global variables occupy a fixed location above the application program. The heap, which stores dynamically allocated variables, is located directly below the application program and expands downward in memory as needed. The run-time stack, which stores local variables and parameters, is located farther down in memory and expands upward as needed.

Global Variables and the `BR` Instruction

Figure 6.3 shows a C program that inputs a global variable with the `scanf()` function and outputs it with the `printf()` function. The second part of the figure shows the object program in Pep/10 assembly language that the compiler produces.

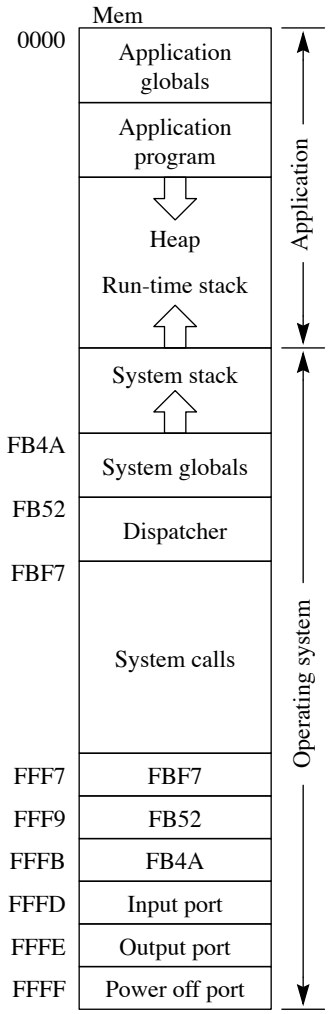


Figure 6.2 The Pep/10 memory map.

High-Order Language

```
#include <stdio.h>
int age;

int main() {
    scanf("%d", &age);
    printf("Age: %d years\n", age);
    return 0;
}
```

Assembly Language

```
         BR      main
age:     .BLOCK  2           ;global variable #2d
;
main:    @DECI   age,d       ;scanf("%d", &age)
         @STRO   msg1,d      ;printf("Age: %d years\n", age)
         @DECO   age,d
         @STRO   msg2,d
         RET                 ;return 0
msg1:    .ASCII  "Age: \0"
msg2:    .ASCII  " years\n\0"
```

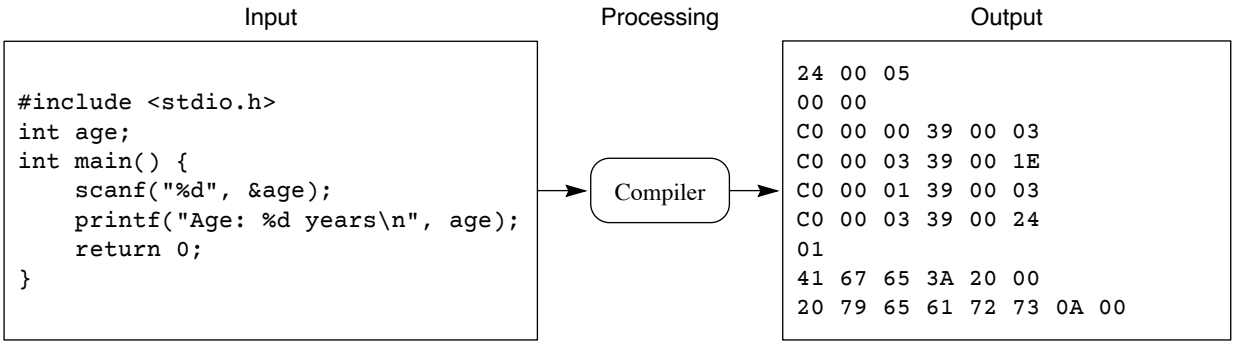
Figure 6.3 Input and output of a global variable. The C program is from Figure 2.x.

Figure 6.4 shows the input and output of a compiler with the source program in Figure 6.3. Part (a) is a compiler that translates directly into machine language. The object program could be loaded and executed. Part (b) is a compiler that translates to assembly language at Level Asmb5. The object program would need to be assembled before it could be loaded and executed.

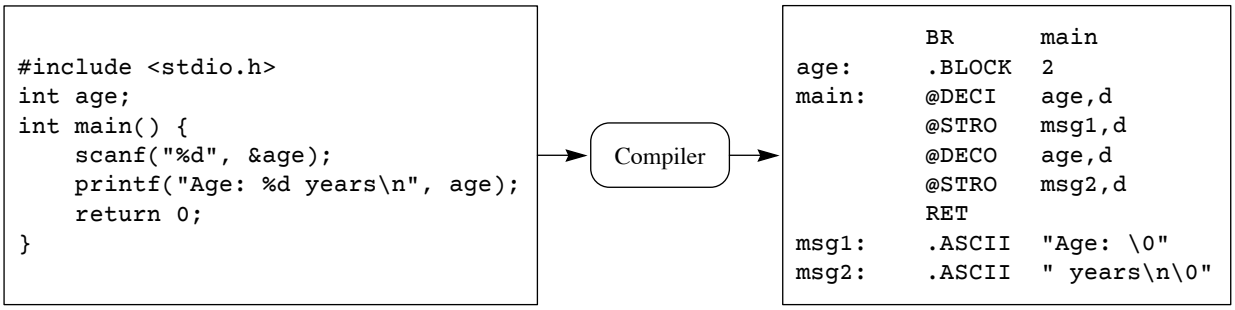
When you select the execute option in the Pep/10 simulator, the program counter (PC) gets the value 0000 (hex). The CPU will interpret the bytes at Mem[0000] as the first instruction to execute. To place global variables at the top of the program, we need an instruction that will cause the CPU to skip over the global variables when it fetches the next instruction. The unconditional branch `BR` is such an instruction.

Because the branch instructions almost always use immediate addressing, the Pep/10 assembler does not require that the addressing mode be specified. If you do not specify the addressing mode for a branch instruction, the assembler will assume immediate addressing and generate 0 for the addressing-a field. In the assembly language program of Figure 6.3, the instruction

```
BR      main
```

(a) Translation directly to machine language.



(b) Translation to assembly language.

Figure 6.4 The action of a C compiler on the source program in Figure 6.3.

is equivalent to

```
BR      main,i
```

This unconditional branch causes the instruction at `main` to execute next, namely the `@DECI` instruction.

The RTL specification of the `BR` instruction is

PC ← Oprnd

The unconditional branch instruction simply places the operand of the instruction in the program counter.

To see how the `BR` instruction executes in this program, consider the assembly language listing for the first three lines of source code.

Cycle	State		
0	A:	PC: 0000	IR:
1-fetch	A:	PC: 0000	IR: 240005
1-increment	A:	PC: 0003	IR: 240005
1-execute	A:	PC: 0005	IR: 240005
2-fetch	A:	PC: 0005	IR: C00000
2-increment	A:	PC: 0008	IR: C00000
2-execute	A: 0000	PC: 0008	IR: C00000

Figure 6.5 von Neumann trace table for execution of the assembly language program in Figure 6.3.

```
0000  240005           BR      main
0003  0000    age:     .BLOCK  2           ;global variable #2d
              ;main:   @DECI   age,d       ;scanf("%d", &age)
0005  C00000  main:    LDWA    DECI,i
0008  390003           SCALL   age,d
              ;        End @DECI
```

`main` is a symbol with value 0005. Therefore, the assembly language instruction `BR main` generates the machine language instruction 240005. The first executable instruction after the global variable `LDWA DECI,i` is stored at Mem[0005]. Figure 6.5 is a von Neumann trace of the execution of the branch instruction (Cycle 1) and the load word instruction (Cycle 2).

1-fetch:
: Because the program counter is 0000, the instruction register gets the branch instruction 240005 at Mem[0000].

1-increment:
: Because the branch instruction is dyadic, the program counter is incremented by three to 0003.

1-execute:
: Because the RTL specification for the branch instruction is PC ← Oprnd, and the addressing mode is immediate, the program counter gets 0005.

2-fetch:
: Because the program counter is 0005, the instruction register gets the load word instruction C00000 at Mem[0005].

2-increment:
Because the load word instruction is dyadic, the program counter is incremented by three to 0008.

2-execute:
Because the addressing mode is immediate, and the value of the `DECI` symbol is 0000, the accumulator gets 0000.

The correct operation of the `BR` instruction depends on the details of the von Neumann execution cycle. For example, you may have wondered why the cycle is fetch, decode, *increment*, *execute*, repeat instead of fetch, decode, *execute*, *increment*, repeat. If the execute part of the von Neumann execution cycle occurs before the increment part, then PC will have the address of the currently executing instruction in the IR. It seems to make more sense to have PC correspond to the currently executing instruction instead of the instruction after the currently executing one.

Why doesn't the von Neumann execution cycle have the execute part before the increment part? Because then `BR` would not work properly. Here is what what would happen to the program in Figure 6.5 if the execute part of the cycle occurred before the increment part.

1-fetch:
Because the program counter is 0000, the instruction register gets the branch instruction 240005 at Mem[0000].

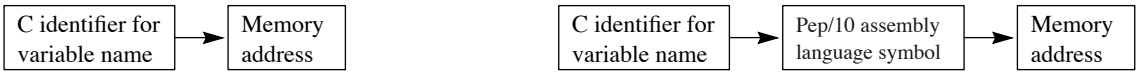
1-execute:
PC has the address of the currently executing instruction.
Because the RTL specification for the branch instruction is PC ← Oprnd, and the addressing mode is immediate, the program counter gets 0005.

1-increment:
Because the branch instruction is dyadic, the program counter is incremented by three to 0008.

2-fetch:
Because the program counter is 0008, the instruction register gets the system call instruction 390003 at Mem[0008].

Instead of branching to 0005, your program would branch to 0008 skipping the load word instruction.

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



(a) Translation directly to machine language. **(b)** Translation first to assembly language.

Figure 6.6 The mapping a compiler makes between a Level-HOL6 variable and a Level-ISA3 storage location.

Symbol	Scope	Type
`age`	global	int

(a) The compiler symbol table.

Symbol	Value
`age`	0003

(b) The assembler symbol table

Figure 6.7 The compiler and assembler symbol tables for the program in Figure 6.3.

The compiler must remember which address corresponds to which variable name in the Level-HOL6 program. It uses a symbol table to make the connection between variable names and addresses. The symbol table for a compiler is similar to, but inherently more complicated than, the symbol table for an assembler. A variable name in C is not limited to eight characters, as is a symbol in Pep/10. In addition, the symbol table for a compiler must store the variable's type as well as its associated address.

A compiler that translates directly to machine language does not require a second translation with an assembler. Figure 6.6(a) shows the mapping produced by the symbol table for such a compiler. The programs in this text illustrate the translation process for a compiler that translates to Pep/10 assembly language, because assembly language is easier to read than machine language. Variable names in C correspond to symbols in Pep/10 assembly language, as Figure 6.6(b) shows.

The correspondence in Figure 6.6(b) is unrealistic for compilers that translate to assembly language. Consider the problem of a C program that has two variables named `discountRate1` and `discountRate2`. Because they are longer than eight characters, the compiler would have a difficult time mapping the identifiers to unique Pep/10 symbols. Our examples will limit the C identifiers to, at most, eight characters to make clear the correspondence between C and assembly language.

Figure 6.7 shows the compiler and assembler symbol tables for the program in Figure 6.3 where the compiler translates to assembly language. The compiler stores the scope of the variable—whether it is global, local, or dynamically allocated. It also stores the variable's type so it can verify that any expression that uses the variable is type correct.

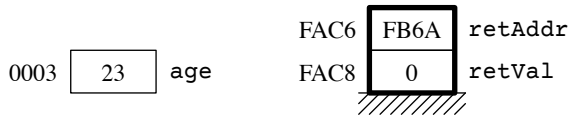


Figure 6.8 The symbol tracer for the program of Figure 6.3.

In general, the compiler has the following rules for generating code for global variables.

- Allocate global variables at a fixed location in memory with `.BLOCK`.
- Access global variables with direct addressing.

The compiler has the following rules for generating code for `scanf()`.

- Translate character input with `@CHARI`.
- Translate integer input with `@DECI`.

The compiler has the following rules for generating code for `printf()`.

- Translate character output with `@CHARO`.
- Translate integer output with `@DECO`.
- Translate string output of any substrings from the format string with `@STRO` or `@CHARO`.

The Pep/10 Symbol Tracer

Pep/10 has three symbolic trace features corresponding to the three parts of the C memory model—the global tracer for global variables, the stack tracer for parameters and local variables, and the heap tracer for dynamically allocated variables. To trace a variable, the programmer embeds trace tags in the comments associated with the variables and single steps through the program. The Pep/10 integrated development environment shows the run-time values of the variables.

There are two kinds of trace tags:

- Format trace tags
- Symbol trace tags

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

The symbol tracer allows the user to specify which global symbol to trace by placing a format trace tag in the comment of the `.BLOCK` line where the global variable is declared. For example, this line from Figure 6.3


```
age:     .BLOCK  2           ;global variable #2d
```

has format trace tag `#2d`. This trace tag tells the symbol tracer to display the two-byte cell at the address specified by `age` as a decimal integer, along with the symbol `age` itself.

Figure 6.8 shows the symbol tracer for the program of Figure 6.3 when the input is `23`. The symbol tracer shows the global variable `age` as the box at a fixed location in memory at address 0003.

The symbol tracer also shows the run-time stack. The `main()` function of this program has no parameters or local variables. Consequently, the only items on the run-time stack for the function call are storage for the return value and the return address. You can see now that the return address from the operating system that the figures in Chapter 2 denoted as "ra0" has the value FB6A, which is an address in the dispatcher of the operating system..

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. To keep things simple, this text always declares `main()` without the parameters. Hence, the only storage allocated for `main()` on the run-time stack is for the return value and any local variables.

The legal format trace tags are:

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

Global variables do not require the use of symbol trace tags, because the Pep/10 symbol tracer takes the symbol from the `.BLOCK` line on which the trace tag is placed. Local variables, however, require symbol trace tags, which are described later.

Constants and the `.EQUATE` Pseudo-Op

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

High-Order Language

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

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

Assembly Language

```
         BR      main
bonus:   .EQUATE 10          ;constant
exam1:   .BLOCK  2           ;global variable #2d
exam2:   .BLOCK  2           ;global variable #2d
score:   .BLOCK  2           ;global variable #2d
;
main:    @DECI   exam1,d     ;scanf("%d %d", &exam1, &exam2)
         @DECI   exam2,d
         LDWA    exam1,d     ;score = (exam1 + exam2) / 2 + bonus
         ADDA    exam2,d
         ASRA
         ADDA    bonus,i
         STWA    score,d
         @STRO   msg,d       ;printf("score = %d\n", score)
         @DECO   score,d
         @CHARO  '\n',i
         RET
msg:     .ASCII  "score = \0"
```

Figure 6.9 A program with a C constant at Level HOL6 and Level Asmb5.
The C program is from Figure 2.x.

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

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

The C program in Figure 6.9 is identical to the one in Figure 2.6, except

that the variables are global instead of local. It shows how to translate a C constant to assembly language. The program calculates a value for `score` as the average of two exam grades plus a 10-point bonus.

The compiler translates

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

as

```
bonus: .EQUATE 10
```

Here is the machine language generated by the declarations in the global space.

```
0000  240009          BR      main
              bonus:  .EQUATE 10          ;constant
0003  0000    exam1:  .BLOCK  2           ;global variable #2d
0005  0000    exam2:  .BLOCK  2           ;global variable #2d
0007  0000    score:  .BLOCK  2           ;global variable #2d
```

The code is notable on two counts.

First, the line that contains the `.EQUATE` has no code in the machine language column. There is not even an address in the address column because there is no code to which the address would apply. This is consistent with the rule that `.EQUATE` does not generate code.

Second, you can see from the symbol table that symbol `bonus` has the value 000A (hex), which is 10 (dec). In contrast, the symbol `exam2` has the value 5 because the code generated for it by the `.BLOCK` dot command is at address 0005 (hex). But, there is no code for `bonus`, which is set to 000A by the `.EQUATE` dot command.

Symbol	Value
`bonus`	000A
`exam1`	0003
`exam2`	0005
`score`	0007

Assignment Statements

An assignment statement in C has the general form

variable `=` expression `;`

where the expression might be a single variable or a more complicated expression. The compiler rule for generating code for the assignment statement is:

- Load the accumulator from the expression with `LDWA` for type `int` or `LDBA` for type `char`.
- Compute the value of the expression if necessary.
- Store the value to the variable with `STWA` or `STBA`.

In Figure 6.9, to compute the expression

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


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

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

```
001C  50000A          ADDA    bonus,i
```

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

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

In a more realistic program, `score` would have type `float`, and you would compute the average with the real division operator. Pep/10 does not have hardware support for real numbers. Nor does its instruction set contain instructions for multiplying or dividing integers. These operations must be programmed in software using the arithmetic instructions.

6.2 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.

Stack-Relative Addressing

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

Oprnd = Mem[SP + OprndSpec]

The stack pointer acts as a memory address to which the operand specifier is added. Figure 6.2 shows that the user stack grows upward in main memory starting at address FAC6. 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.

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.


```
LDBA    'B',i       ;move 'B' to stack
STBA    -1,s
LDBA    'M',i       ;move 'M' to stack
STBA    -2,s
LDBA    'W',i       ;move 'W' to stack
STBA    -3,s
LDWA    335,i       ;move 335 to stack
STWA    -5,s
LDBA    'i',i       ;move 'i' to stack
STBA    -6,s
SUBSP   6,i         ;push 6 bytes onto stack
@CHARO  5,s         ;output B
@CHARO  4,s         ;output M
@CHARO  3,s         ;output W
@DECO   1,s         ;output 335
@CHARO  0,s         ;output i
ADDSP   6,i         ;pop 6 bytes off stack
RET
```

Figure 6.10 Stack-relative addressing.

The Pep/10 instruction set has two instructions for manipulating the stack pointer directly, `ADDSP` and `SUBSP`. `ADDSP` simply adds a value to the stack pointer, and `SUBSP` subtracts a value. The register transfer language (RTL) specification of `ADDSP` is

SP ← SP + Oprnd

and the RTL specification of `SUBSP` is

SP ← SP - Oprnd

Neither instruction changes the status bits.

Accessing the Run-Time Stack

Figure 6.10 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 6.11(a) shows the values in the stack pointer (SP) and main memory before the program executes. The initial value of SP is FAC6 from the function call that the operating system made to `main()`.

The first two instructions

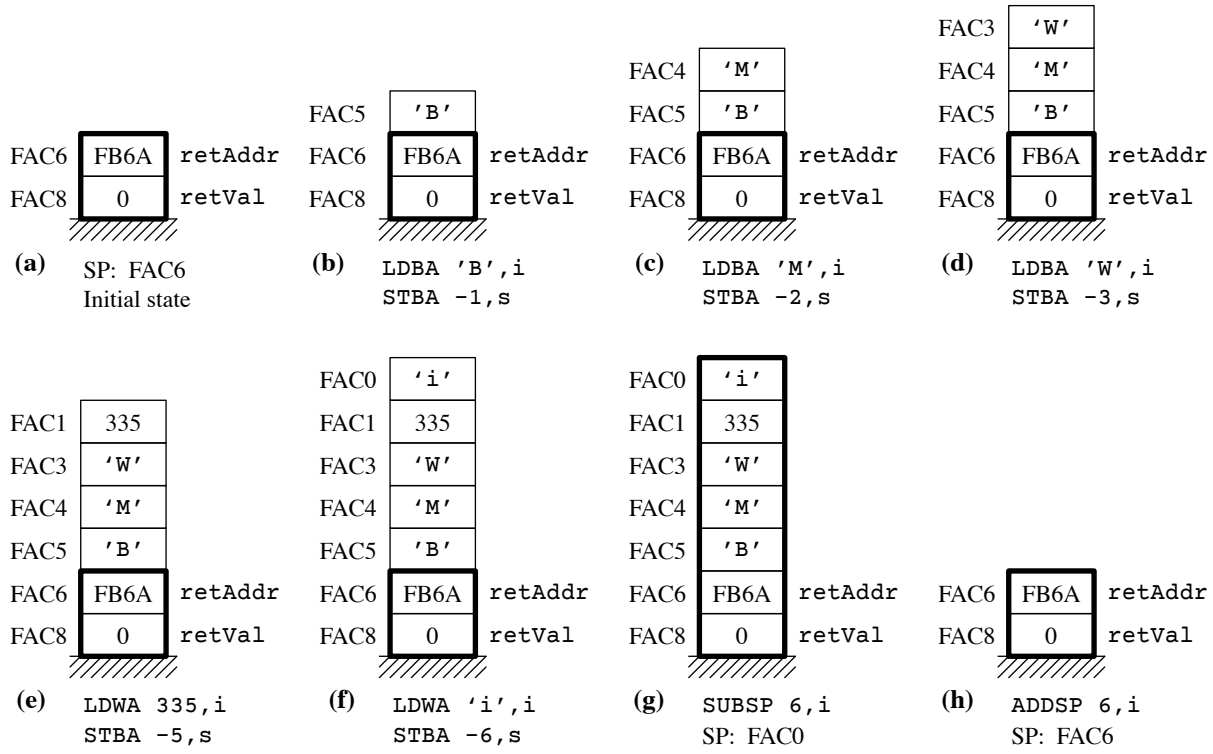


Figure 6.11 A trace of the run-time stack for the program of Figure 6.10.

```
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 FAC6, the `B` is stored at Mem[FAC6 + FFFF] = Mem[FAC5]. The next two instructions put `M` and `W` at Mem[FAC4] and Mem[FAC3], 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 instructions to move the 335

```
0012  C0014F          LDWA    335,i       ;move 335 to stack
0015  E3FFFB          STWA    -5,s
```

use `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.

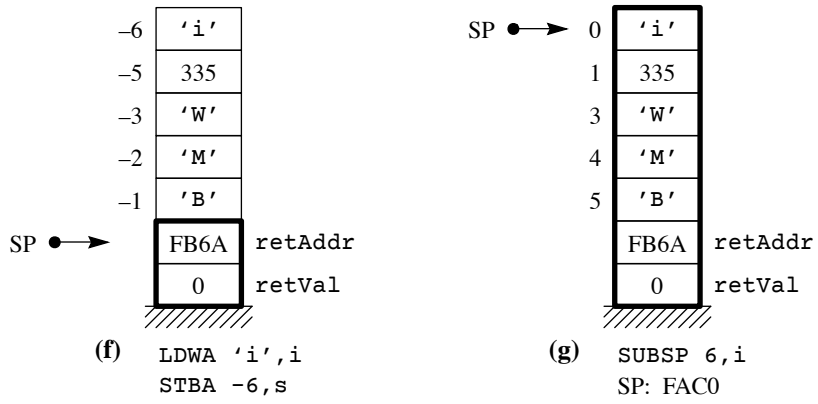


Figure 6.12 The stack-relative addresses of Figure 6.11.

The subtract stack pointer instruction

```
001E  480006          SUBSP   6,i         ;push 6 bytes onto stack
```

subtracts 6 from the stack pointer, as Figure 6.11(g) 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 FBD8. In that case, `B`, `M`, `W`, 335, and `i` would be at Mem[FBD7], Mem [FBD6], Mem[FBD5], Mem[FBD3], and Mem[FBD2], respectively, and the stack pointer would wind up with a value of FBD2. 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 6.12 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

```
              ;         @CHARO  5,s         ;output B
0021  D30005            LDBA    5,s
0024  F1FFFE            STBA    charOut,d
              ;         End @CHARO
```

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. The other items are output similarly using their stack offsets, as shown in Figure 6.3(b).

The instruction

```
003F  400006          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 section 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. The compiler translates a program with local variables as follows.

- Push local variables onto the stack with `SUBSP`.
- Access local variables with stack-relative addressing.
- Pop local variables off of the stack with `ADDSP`.

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 6.13 is from Figure 2.6. It is identical to the program of 6.9 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 6.14 shows the run-time stack for the program. `bonus` is a constant and is defined with the `.EQUATE` command (as in Figure 6.9). 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 6.14 shows that the stack offset for local variable `exam1` is 4. Therefore, the assembly language program equates the symbol `exam1` to 4.

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

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;
}
```

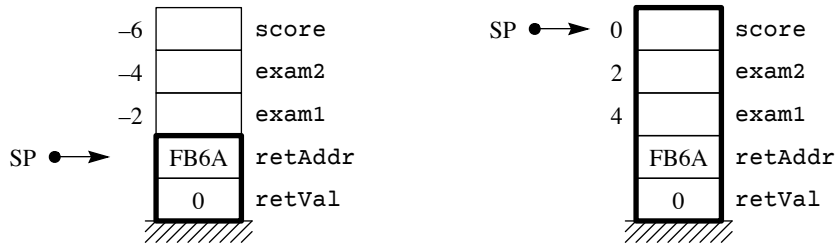
Assembly Language

```
         BR      main
bonus:   .EQUATE 10          ;constant
exam1:   .EQUATE 4           ;local variable #2d
exam2:   .EQUATE 2           ;local variable #2d
score:   .EQUATE 0           ;local variable #2d
;
main:    SUBSP   6,i         ;push #exam1 #exam2 #score
         @DECI   exam1,s     ;scanf("%d %d", &exam1, &exam2)
         @DECI   exam2,s
         LDWA    exam1,s     ;score = (exam1 + exam2) / 2 + bonus
         ADDA    exam2,s
         ASRA
         ADDA    bonus,i
         STWA    score,s
         @STRO   msg,d       ;printf("score = %d\n", score)
         @DECO   score,s
         @CHARO  '\n',i
         ADDSP   6,i         ;pop #score #exam2 #exam1
         RET
msg:     .ASCII  "score = \0"
```

Figure 6.13 A program with local variables. The C program is from Figure 2.6.

these differences, the translation of the assignment and output statements is the same.

Figure 6.13 shows how to write trace tags for debugging with local variables. The assembly language program uses the format trace tag `#2d` with the



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

Figure 6.14 The stack-relative addresses of Figure 6.13.

`.EQUATE` pseudo-op to tell the debugger that the values of `exam1`, `exam2`, and `score` should be displayed as two-byte decimal values.

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/10 system displays a figure on the screen 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.

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.

Return from `main()`

Returning a non-zero value from `main()` indicates an error or abnormal termination. The action taken when a non-zero value is returned depends on the system that calls `main()`. Figure 6.15 shows a C program that returns a non-zero value, and the program's translation by the C compiler to Pep/10 assembly language.

The output in the figure is the result of executing the assembly language

High-Order Language

```
#include <stdio.h>

int main() {
    return 666;
}
```

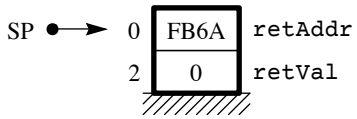
Assembly Language Listing

```
0000  240003          BR      main
            retVal:   .EQUATE 2
0003  C0029A main:    LDWA    666,i
0006  E30002          STWA    retVal,s
0009  01              RET
```

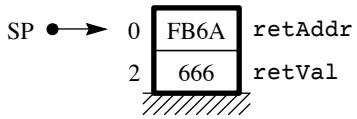
Output

```
Main failed with return value 666
```

Figure 6.15 The result of a non-zero return from `main()`.



(a) Before `STWA retVal,s`.



(b) After `STWA retVal,s`.

Figure 6.16 The run-time stack for Figure 6.15.

program with the Pep/10 operating system. If you run the C program on your computer the output will be different. Most systems output an error message, but the details of the message depends on the system.

6.3 Flow of Control

The Pep/10 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.

`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 6.17 shows how a compiler translates 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`, 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

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

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 6.18(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

High-Order Language

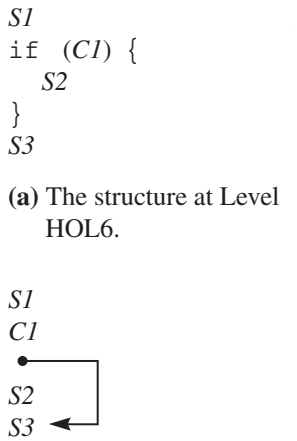
```
#include <stdio.h>

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

Assembly Language

```
         BR      main
number:  .EQUATE 0           ;local variable #2d
;
main:    SUBSP   2,i         ;push #number
         @DECI   number,s    ;scanf("%d", &number)
if:      LDWA    number,s    ;if (number < 0)
         BRGE    endIf
         LDWA    number,s    ;number = -number
         NEGA
         STWA    number,s
endIf:   @DECO   number,s    ;printf("%d", number)
         ADDSP   2,i         ;pop #number
         RET
```

Figure 6.17 The `if` statement at Level HOL6 and Level Asmb5. The C program is from Figure 2.x.



(a) The structure at Level HOL6.

(b) The same structure at Level Asmb5.

Figure 6.18 The structure of the `if` statement in Figure 6.17.

```
number = -number;
```

and *S3* represents the statement `printf()` function call. Figure 6.18(b) shows the structure with the more primitive branching instructions at Level Asmb5. The dot following *C1* represents the conditional branch, `BRGE`.

The definition of the `if` symbol in the statement

```
if:      LDWA    number,s    ;if (number < 0)
```

is not necessary for the program to execute correctly. No other statements in the program refer to the symbol `if`. Its purpose is documentation for the human reader.

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 6.17. You can eliminate the line of code

```
        LDWA    number,s    ;number = -number
```

because the value of `number` will still be in the accumulator from the previous load.

```
if:     LDWA    number,s    ;if (number < 0)
```

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 6.17.

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. In Figure 6.17, the compiler would need to remember that the value of `number` was still in the accumulator from the code generated for the `if` statement.

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.

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 6.19 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`

```
BR      endIf
```

and the input were 127, the output would be `highlow`.

Unlike Figure 6.17, the `if` statement in Figure 6.19 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

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

where T represents a temporary value.

The `CPWA` instruction 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.

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

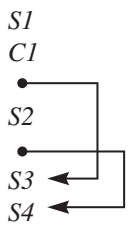
```
         BR      main
limit:   .EQUATE 100          ;constant
num:     .EQUATE 0            ;local variable #2d
;
main:    SUBSP   2,i          ;push #num
         @DECI   num,s        ;scanf("%d", &num)
if:      LDWA    num,s        ;if (num >= limit)
         CPWA    limit,i
         BRLT    else
         @STRO   msg1,d       ;printf("high\n")
         BR      endIf
else:    @STRO   msg2,d       ;printf("low\n")
endIf:   ADDSP   2,i          ;pop #num
         RET
msg1:    .ASCII  "high\n\0"
msg2:    .ASCII  "low\n\0"
```

Figure 6.19 The `if/else` statement at Level HOL6 and Level Asmb5. The C program is from Figure 2.10.

Figure 6.20 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.

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

(a) The structure at Level HOL6.



(b) The same structure at Level Asmb5.

Figure 6.20 The structure of the `if/else` statement in Figure 6.19.

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;
}
```

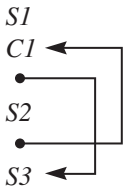
Assembly Language

```
         BR      main
letter:  .BLOCK  1           ;global variable #1c
;
main:    @CHARI  letter,d    ;scanf("%c", &letter)
while:   LDBA    letter,d    ;while (letter != '*')
         CPBA    '*',i
         BREQ    endWh
if:      CPBA    ' ',i       ;if (letter == ' ')
         BRNE    else
         @CHARO  '\n',i      ;printf("\n")
         BR      endIf
else:    @CHARO  letter,d    ;printf("%c", letter)
endIf:   @CHARI  letter,d    ;scanf("%c", &letter)
         BR      while
endWh:   RET
```

Figure 6.21 The `while` loop at Level HOL6 and Level Asmb5. The C program is from Figure 2.13.

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

(a) The structure at Level HOL6.



(b) The same structure at Level Asmb5.

Figure 6.22 The structure of the `while` statement in Figure 6.21.

Translating the `while` Loop

Translating a loop requires branches to previous instructions. Figure 6.21 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 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.

```
Hello,
world!
```

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, with the compare byte accumulator instruction `CPBA`. Every `while` loop ends with an unconditional branch to the test at the top of the loop. The unconditional branch

```
BR      while
```

brings control back to the initial test. Figgure 6.22 shows the structure of the `while` statement at the two levels.

The RTL specification of `CPBr` is

$$T \leftarrow r[8:15] - \text{byte Oprnd} \; ; \; N \leftarrow T < 0 \; , \; Z \leftarrow T = 0 \; , \; V \leftarrow 0 \; , \; C \leftarrow 0$$

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 6.21 would still function correctly even if the accumulator had some 1's in its high-order byte.

The RTL specification of `LDBr` is

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

It clears the left half of the register to zero, and loads the operand into the right half of the register. 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 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 6.23 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

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

```
         BR      main
cop:     .BLOCK  2           ;global variable #2d
driver:  .BLOCK  2           ;global variable #2d
;
main:    LDWA    0,i         ;cop = 0
         STWA    cop,d
         LDWA    40,i        ;driver = 40
         STWA    driver,d
do:      LDWA    cop,d       ;cop += 25
         ADDA    25,i
         STWA    cop,d
         LDWA    driver,d    ;driver += 20
         ADDA    20,i
         STWA    driver,d
while:   LDWA    cop,d       ;while (cop < driver)
         CPWA    driver,d
         BRLT    do
         @DECO   cop,d       ;printf("%d", cop)
         RET
```

Figure 6.23 The do loop at Level HOL6 and Level Asmb5. The C program is from Figure 2.14.

the two motorists, initialized to 0 and 40, respectively. Each 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

```
while:   LDWA    cop,d       ;while (cop < driver)
         CPWA    driver,d
         BRLT    do
```

`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
```

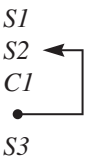
that is, if

```
cop < driver
```

That is the condition under which the loop should repeat. Figure 6.24 shows the structure of the `do` statement at Levels 6 and 5.

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

(a) The structure at Level HOL6.



(b) The same structure at Level Asmb5.

Figure 6.24 The structure of the `do` loop in Figure 6.23.

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 6.25 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.

Here is the code from the program for each of the items in the above sequence.

Initialize the control variable:

```
         LDWA    0,i         ;for (j = 0
         STWA    j,s
```


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

```
         BR      main
j:       .EQUATE 0           ;local variable #2d
;
main:    SUBSP   2,i         ;push #j
         LDWA    0,i         ;for (j = 0
         STWA    j,s
for:     CPWA    3,i         ;j < 3
         BRGE    endFor
         @STRO   msg,d       ;printf("j = %d\n", j)
         @DECO   j,s
         @CHARO  '\n',i
         LDWA    j,s         ;j++)
         ADDA    1,i
         STWA    j,s
         BR      for
endFor:  ADDSP   2,i         ;pop #j
         RET
msg:     .ASCII  "j = \0"
```

Figure 6.25 The `for` loop at Level HOL6 and Level Asmb5. The C program is from Figure 2.x.

Test the control variable:

```
    for:     CPWA    3,i         ;j < 3
             BRGE    endFor
```

Execute the loop body:

```
             @STRO   msg,d       ;printf("j = %d\n", j)
             @DECO   j,s
             @CHARO  '\n',i
```

Increment the control variable:


```
            LDWA    j,s         ;j++)
            ADDA    1,i
            STWA    j,s
```

Branch to the test:

```
            BR      for
```

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.

6.5 Function Calls

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 Void Functions

Figure 6.27 shows how a compiler translates a void function call without parameters. The program outputs three triangles of asterisks. The compiler generates the dyadic instruction `CALL` to call the function and the monadic instruction `RET` to return from the function.

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:

$$\text{SP} \leftarrow \text{SP} - 2 \; ; \; \text{Mem}[\text{SP}] \leftarrow \text{PC} \; ; \; \text{PC} \leftarrow \text{Oprnd}$$

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

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/10 assembler will assume immediate addressing.

Here is the RTL specification of the monadic `RET` instruction:

$$\text{PC} \leftarrow \text{Mem}[\text{SP}] \; ; \; \text{SP} \leftarrow \text{SP} + 2$$

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.

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 6.28 is a partial listing of the program in Figure 6.27. It also shows the run-time stack before and after execution of the first `CALL` and first `RET` statements. As usual, the initial value of the stack pointer is FAC6. The figure shows the Execute part of the cycle after the Increment part.

High-Order Language

```
#include <stdio.h>
void printTri() {
    printf("*\n");
    printf("**\n");
    printf("***\n");
}
int main() {
    printTri();
    printTri();
    printTri();
    return 0;
}
```

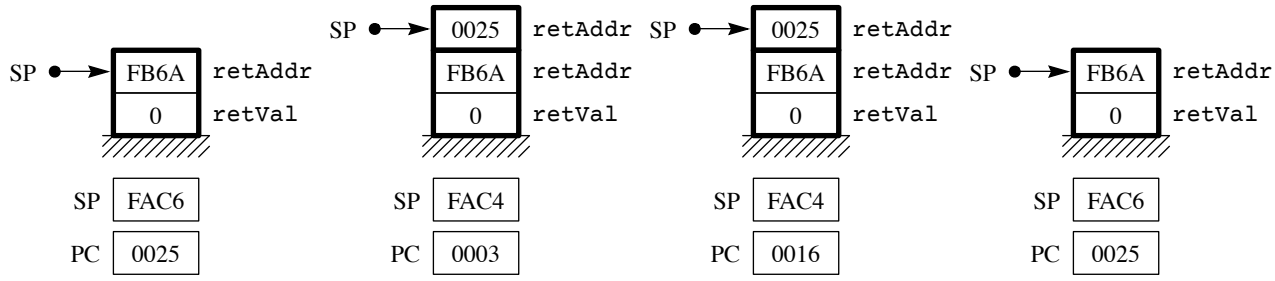
Assembly Language

```
         BR      main
;
;******* void printTri()
printTri:@STRO   msg1,d      ;printf("*\n")
         @STRO   msg2,d      ;printf("**\n")
         @STRO   msg3,d      ;printf("***\n")
         RET
msg1:    .ASCII  "*\n\0"
msg2:    .ASCII  "**\n\0"
msg3:    .ASCII  "***\n\0"
;
;******* int main()
main:    CALL    printTri    ;printTri()
         CALL    printTri    ;printTri()
         CALL    printTri    ;printTri()
         RET
```

Output

```
*
**
***
*
**
***
*
**
***
```

Figure 6.27 A void function call at Level HOL6 and Level Asmb5.



(a) Before first `CALL`. **(b)** After first `CALL`. **(c)** Before first `RET`. **(d)** After first `RET`.

```
...
            ;******* void printTri()
            ;printTri:@STRO   msg1,d      ;printf("*\n")
0003  C00003 printTri:LDWA    STRO,i
0006  390016          SCALL   msg1,d
            ;         End @STRO
...

0015  01              RET
...
            ;******* int main()
0022  360003 main:    CALL    printTri    ;printTri()
0025  360003          CALL    printTri    ;printTri()
...
```

Figure 6.28 A partial listing of the program in Figure 6.27.

Figure 6.28(a) shows the content of the program counter as 0025 before execution of the first `CALL` instruction. It is not the address of the first `CALL` instruction, which is 0022. Why not? Because the program counter was incremented to 0025 before the Execute part of the cycle. Therefore, during the Execute part of the cycle for 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,

$SP \leftarrow SP - 2$

subtracts two from SP, giving it the value FAC4. Then,

$Mem[SP] \leftarrow PC$

puts the value of the program counter, 0025, into main memory at address FAC4—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 6.28(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` function. The output instructions of the function execute, producing the pattern of a triangle of asterisks.

Eventually the `RET` instruction at 0015 executes. Figure 6.28(c) shows the content of the program counter as 0016 just before execution of `RET`. This might seem strange, because 0016 is not even the address of an instruction. It is the address of the string `"*\n\0"`. Why? Because `RET` is a monadic instruction and the CPU incremented the program counter by one. The first step in the execution of `RET` is

PC ← Mem[SP]

which puts 0025 into the program counter. Then

SP ← SP + 2

changes the stack pointer back to FAC6.

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 `RET` instruction in `main()` executes, returning control back to the operating system.

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.

Translating Global Parameters

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 6.29, which is identical

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;
}
```

Figure 6.29 A void function call with global parameters at Level HOL6 and Level Asmb5. The C program is from Figure 2.16. (*continues*)

to the program in Figure 2.16, is a HOL6 program that prints a bar chart and the program's corresponding Asmb5 translation. It shows the 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 local variables and then pop the return address by executing `RET`. Before the caller can continue, it must pop the actual parameters.

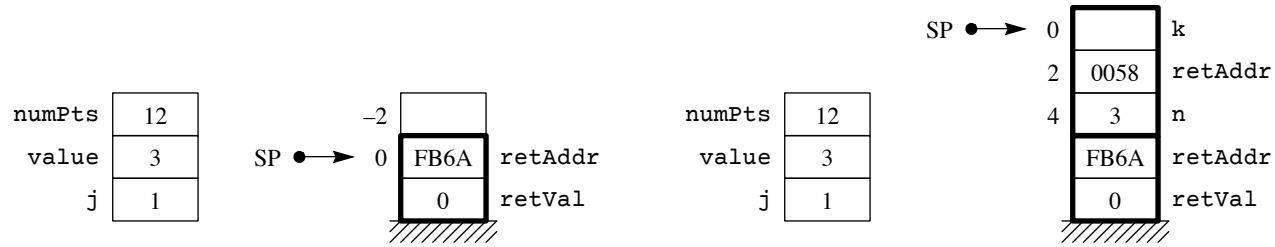
In summary, the caller and callee functions do the following:

- 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`).

Assembly Language

```
         BR      main
numPts:  .BLOCK  2           ;global variable #2d
value:   .BLOCK  2           ;global variable #2d
j:       .BLOCK  2           ;global variable #2d
;
;******* void printBar(int n)
n:       .EQUATE 4           ;formal parameter #2d
k:       .EQUATE 0           ;local variable #2d
printBar:SUBSP   2,i         ;push #k
         LDWA    1,i         ;for (k = 1
         STWA    k,s
for1:    CPWA    n,s         ;k <= n
         BRGT    endFor1
         @CHARO  '*',i       ;printf("*")
         LDWA    k,s         ;k++)
         ADDA    1,i
         STWA    k,s
         BR      for1
endFor1: @CHARO  '\n',i      ;printf("\n")
         ADDSP   2,i         ;pop #k
         RET
;
;******* main()
main:    @DECI   numPts,d    ;scanf("%d", &numPts)
         LDWA    1,i         ;for (j = 1
         STWA    j,d
for2:    CPWA    numPts,d    ;j <= numPts
         BRGT    endFor2
         @DECI   value,d     ;scanf("%d", &value)
         LDWA    value,d     ;move value
         STWA    -2,s
         SUBSP   2,i         ;push #n
         CALL    printBar    ;printBar(value)
         ADDSP   2,i         ;pop #n
         LDWA    j,d         ;j++)
         ADDA    1,i
         STWA    j,d
         BR      for2
endFor2: RET
```

Figure 6.29 (*continued*) A void function call with global parameters at Level HOL6 and Level Asmb5. The C program is from Figure 2.16.



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

Figure 6.30 A void function call with global parameters for the program of Figure 6.29.

- Callee pops return address (executes `RET`).
- Caller pops actual parameters (executes `ADDSP`).

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 HOL6 main program—`numPts`, `value`, `j`—correspond to the identical 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 as the following code from the listing shows.

```
0003  0000   numPts:  .BLOCK  2          ;global variable #2d
0005  0000   value:   .BLOCK  2          ;global variable #2d
0007  0000   j:       .BLOCK  2          ;global variable #2d
```

Figure 6.30(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. Function `printBar` defines them with

```
;******* void printBar(int n)
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 function, as Figure 6.30(b) shows. The function refers to them with stack-relative addressing.

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

```
004C  C10005          LDWA    value,d     ;move value
004F  E3FFFE          STWA    -2,s
0052  480002          SUBSP   2,i         ;push #n
0055  360009          CALL    printBar    ;printBar(value)
0058  400002          ADDSP   2,i         ;pop #n
```

Because the parameter is a global variable, `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 6.30(a) shows.

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

```
0009  480002 printBar:SUBSP   2,i         ;push #k
...
0030  400002          ADDSP   2,i         ;pop #k
0033  01              RET
```

`SUBSP` subtracts 2 because the local variable, `k`, is a two-byte integer quantity. Figure 6.30(a) shows the run-time stack just after the first input of global variable `value` and just before the first function call. It corresponds directly to Figure 2.17(d). Figure 6.30(b) shows the stack just after the function 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 0058, 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 parameters with global variables, the compiler generates code as follows:

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

Translating Local Parameters

The program in Figure 6.31 is identical to the one in Figure 6.29 except that the variables in `main()` are local instead of global. Although the program behaves like the one in Figure 6.29, the memory model and the translation to Level Asmb5 are different.

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;
}
```

Figure 6.31 A void function call with local parameters at level HOL6 and level Asmb5. The C program is from Figure 6.29 except with local variables.
(*continues*)

You can see that the versions of void function `printBar()` at level HOL6 are identical in Figure 6.29 and Figure 6.31. 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 two programs is in `main()`.

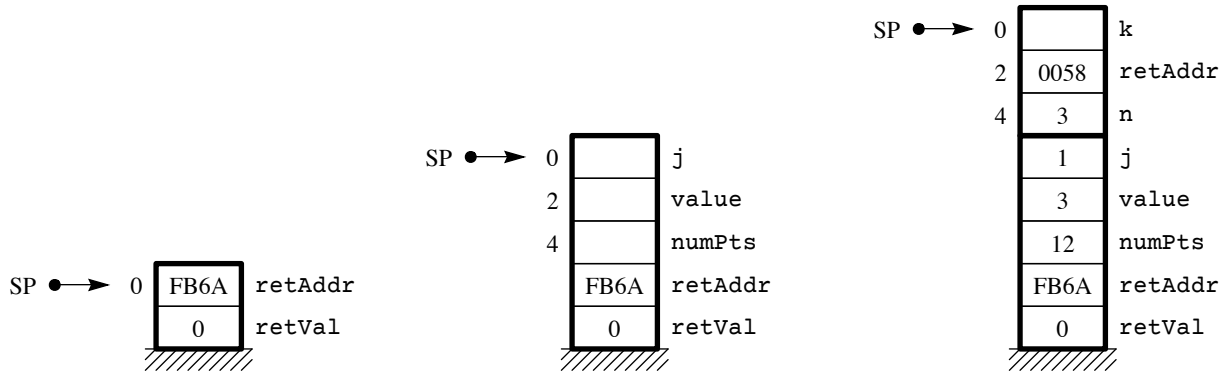
Figure 6.32(a) shows the run-time stack in the main program before it allocates storage for the local variables. Figure 6.32(b) shows the allocation of local variables `numPts`, `value`, and `j` on the run-time stack in the main program. Figure 6.32(c) shows the stack after `printBar` is called for the 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 parameters with local variables, the compiler generates code as follows:

Assembly Language

```
         BR      main
;******* void printBar(int n)
n:       .EQUATE 4           ;formal parameter #2d
k:       .EQUATE 0           ;local variable #2d
printBar:SUBSP   2,i         ;push #k
         LDWA    1,i         ;for (k = 1
         STWA    k,s
for1:    CPWA    n,s         ;k <= n
         BRGT    endFor1
         @CHARO  '*',i       ;printf("*")
         LDWA    k,s         ;k++)
         ADDA    1,i
         STWA    k,s
         BR      for1
endFor1: @CHARO  '\n',i      ;printf("\n")
         ADDSP   2,i         ;pop #k
         RET
;
;******* main()
numPts:  .EQUATE 4           ;local variable #2d
value:   .EQUATE 2           ;local variable #2d
j:       .EQUATE 0           ;local variable #2d
main:    SUBSP   6,i         ;push #numPts #value #j
         @DECI   numPts,s    ;scanf("%d", &numPts)
         LDWA    1,i         ;for (j = 1
         STWA    j,s
for2:    CPWA    numPts,s    ;j <= numPts
         BRGT    endFor2
         @DECI   value,s     ;scanf("%d", &value)
         LDWA    value,s     ;move value
         STWA    -2,s
         SUBSP   2,i         ;push #n
         CALL    printBar    ;printBar(value)
         ADDSP   2,i         ;pop #n
         LDWA    j,s         ;j++)
         ADDA    1,i
         STWA    j,s
         BR      for2
endFor2: ADDSP   6,i         ;pop #j #value #numPts
         RET
```

Figure 6.31 (*continued*) A void function call with local parameters at level HOL6 and level Asmb5. The C program is from Figure 6.29 except with local variables.



(a) Before `SUBSP 6,i` **(b)** After `SUBSP 6,i` **(c)** After `printBar(value)`

Figure 6.32 A void function call with local parameters for the program of Figure 6.31.

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

Translating Non-Void Functions

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 void function call by the extra value that you must allocate for the returned function value.

Figure 6.33 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 trianglee of coefficients (shown in Figure 2.27). The recursive definition of the binomial coefficient is

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

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

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;
}
```

Figure 6.33 A non-void function call at Level HOL6 and Level Asmb5. The C program is from Figure 2.28. (*continues*)

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 6.34(a) shows the run-time stack 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 void function, these offsets would be –2 and –4, respectively. Their magnitudes are greater by 2 because of the two-byte value labeled `retVal` returned on the stack by the function. The `SUBSP` instruction in `main()` allocates six bytes, two for the return value and two each for the actual parameters.

Figure 6.34(b) shows the run-time stack just after the function call from the main program. It corresponds directly to the HOL6 diagram of Figure 2.29(b). The address labeled ra1 in Figure 2.29(b) is 0083 in Figure 6.34(b) as this code from the listing shows.

```
007D  480006          SUBSP   6,i         ;push #retVal #n #k
0080  360003          CALL    binCoeff    ;binCoeff(3, 1)
0083  400006 ra1:     ADDSP   6,i         ;pop #k #n #retVal
```

When the function returns control to `ADDSP` at 0083, 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

Assembly Language

```
         BR      main
;
;******* int binCoeff(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
binCoeff:SUBSP   4,i         ;push #y1 #y2
if:      LDWA    k,s         ;if ((k == 0)
         BREQ    then
         LDWA    n,s         ;|| (n == k))
         CPWA    k,s
         BRNE    else
then:    LDWA    1,i         ;return 1
         STWA    retVal,s
         ADDSP   4,i         ;pop #y2 #y1
         RET
else:    LDWA    n,s         ;move n - 1
         SUBA    1,i
         STWA    -4,s
         LDWA    k,s         ;move k
         STWA    -6,s
         SUBSP   6,i         ;push #retVal #n #k
         CALL    binCoeff    ;binCoeff(n - 1, k)
ra2:     ADDSP   6,i         ;pop #k #n #retVal
         LDWA    -2,s        ;y1 = binCoeff(n - 1, k)
         STWA    y1,s
         LDWA    n,s         ;move n - 1
         SUBA    1,i
         STWA    -4,s
         LDWA    k,s         ;move k - 1
         SUBA    1,i
         STWA    -6,s
         SUBSP   6,i         ;push #retVal #n #k
         CALL    binCoeff    ;binCoeff(n - 1, k - 1)
ra3:     ADDSP   6,i         ;pop #k #n #retVal
```

Figure 6.33 (*continued*) A non-void function call at Level HOL6 and Level Asmb5. The C program is from Figure 2.28. (*continues*)


```
         LDWA    -2,s        ;y2 = binCoeff(n - 1, k - 1)
         STWA    y2,s
         LDWA    y1,s        ;return y1 + y2
         ADDA    y2,s
         STWA    retVal,s
endIf:   ADDSP   4,i         ;pop #y2 #y1
         RET
;
;******* main()
main:    @STRO   msg,d       ;printf("binCoeff(3, 1) = %d\n",
         LDWA    3,i         ;move 3
         STWA    -4,s
         LDWA    1,i         ;move 1
         STWA    -6,s
         SUBSP   6,i         ;push #retVal #n #k
         CALL    binCoeff    ;binCoeff(3, 1)
ra1:     ADDSP   6,i         ;pop #k #n #retVal
         @DECO   -2,s
         @CHARO  '\n',i
         RET
msg:     .ASCII  "binCoeff(3, 1) = \0"
```

Figure 6.33 (*continued*) A non-void function call at Level HOL6 and Level Asmb5. The C program is from Figure 2.28.

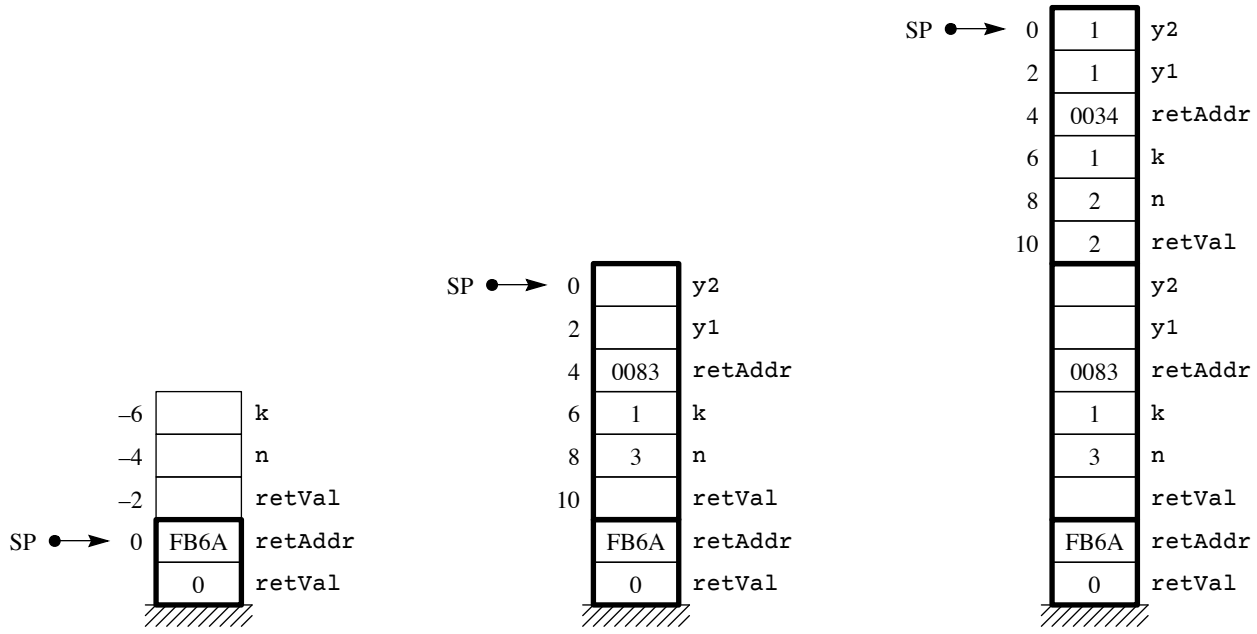
the cell directly below the returned value. So `@DECO` outputs the value with stack-relative addressing and an offset of –2.

Figure 6.34(c) shows the run-time stack at the assembly level just before the third return. It corresponds directly to the HOL6 diagram of Figure 2.29(g). The return address labeled ra2 in Figure 2.29(g) is 0034 in Figure 6.34, the address of the instruction after the first `CALL` in the function as this code from the listing shows.

```
002E  480006          SUBSP   6,i         ;push #retVal #n #k
0031  360003          CALL    binCoeff    ;binCoeff(n - 1, k)
0034  400006 ra2:     ADDSP   6,i         ;pop #k #n #retVal
0037  C3FFFE          LDWA    -2,s        ;y1 = binCoeff(n - 1, k)
```

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

.



(a) Before `binCoeff(3, 1)`. **(b)** Call `binCoeff(3, 1)`. **(c)** Just before third `RET`.

Figure 6.34 A non-void function call for the program of Figure 6.33.

,

