Chapter 8

Code Generation

The final phase in our compiler model is the code generator. It takes as input the intermediate representation (IR) produced by the front end of the compiler, along with relevant symbol table information, and produces as output a semantically equivalent target program, as shown in Fig. 8.1.

The requirements imposed on a code generator are severe. The target program must preserve the semantic meaning of the source program and be of high quality; that is, it must make effective use of the available resources of the target machine. Moreover, the code generator itself must run efficiently.

The challenge is that, mathematically, the problem of generating an optimal target program for a given source program is undecidable; many of the subproblems encountered in code generation such as register allocation are computationally intractable. In practice, we must be content with heuristic techniques that generate good, but not necessarily optimal, code. Fortunately, heuristics have matured enough that a carefully designed code generator can produce code that is several times faster than code produced by a naive one.

Compilers that need to produce efficient target programs, include an optimization phase prior to code generation. The optimizer maps the IR into IR from which more efficient code can be generated. In general, the code-optimization and code-generation phases of a compiler, often referred to as the *back end*, may make multiple passes over the IR before generating the target program. Code optimization is discussed in detail in Chapter 9. The techniques presented in this chapter can be used whether or not an optimization phase occurs before code generation.

A code generator has three primary tasks: instruction selection, register

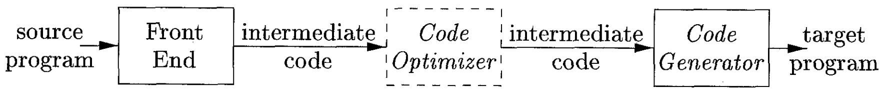

Figure 8.1: Position of code generator

allocation and assignment, and instruction ordering. The importance of these tasks is outlined in Section 8.1. Instruction selection involves choosing appropriate target-machine instructions to implement the IR statements. Register allocation and assignment involves deciding what values to keep in which registers. Instruction ordering involves deciding in what order to schedule the execution of instructions.

This chapter presents algorithms that code generators can use to translate the IR into a sequence of target language instructions for simple register machines. The algorithms will be illustrated by using the machine model in Section 8.2. Chapter 10 covers the problem of code generation for complex modern machines that support a great deal of parallelism within a single instruction.

After discussing the broad issues in the design of a code generator, we show what kind of target code a compiler needs to generate to support the abstractions embodied in a typical source language. In Section 8.3, we outline implementations of static and stack allocation of data areas, and show how names in the IR can be converted into addresses in the target code.

Many code generators partition IR instructions into "basic blocks," which consist of sequences of instructions that are always executed together. The partitioning of the IR into basic blocks is the subject of Section 8.4. The following section presents simple local transformations that can be used to transform basic blocks into modified basic blocks from which more efficient code can be generated. These transformations are a rudimentary form of code optimization, although the deeper theory of code optimization will not be taken up until Chapter 9. An example of a useful, local transformation is the discovery of common subexpressions at the level of intermediate code and the resultant replacement of arithmetic operations by simpler copy operations.

Section 8.6 presents a simple code-generation algorithm that generates code for each statement in turn, keeping operands in registers as long as possible. The output of this kind of code generator can be readily improved by peephole optimization techniques such as those discussed in the following Section 8.7.

The remaining sections explore instruction selection and register allocation.

8.1 Issues in the Design of a Code Generator

While the details are dependent on the specifics of the intermediate representation, the target language, and the run-time system, tasks such as instruction selection, register allocation and assignment, and instruction ordering are encountered in the design of almost all code generators.

The most important criterion for a code generator is that it produce correct code. Correctness takes on special significance because of the number of special cases that a code generator might face. Given the premium on correctness, designing a code generator so it can be easily implemented, tested, and maintained is an important design goal.

8.1.1 Input to the Code Generator

The input to the code generator is the intermediate representation of the source program produced by the front end, along with information in the symbol table that is used to determine the run-time addresses of the data objects denoted by the names in the IR.

The many choices for the IR include three-address representations such as quadruples, triples, indirect triples; virtual machine representations such as bytecodes and stack-machine code; linear representations such as postfix notation; and graphical representations such as syntax trees and DAG's. Many of the algorithms in this chapter are couched in terms of the representations considered in Chapter 6: three-address code, trees, and DAG's. The techniques we discuss can be applied, however, to the other intermediate representations as well.

In this chapter, we assume that the front end has scanned, parsed, and translated the source program into a relatively low-level IR, so that the values of the names appearing in the IR can be represented by quantities that the target machine can directly manipulate, such as integers and floating-point numbers. We also assume that all syntactic and static semantic errors have been detected, that the necessary type checking has taken place, and that type-conversion operators have been inserted wherever necessary. The code generator can therefore proceed on the assumption that its input is free of these kinds of errors.

8.1.2 The Target Program

The instruction-set architecture of the target machine has a significant impact on the difficulty of constructing a good code generator that produces high-quality machine code. The most common target-machine architectures are RISC (reduced instruction set computer), CISC (complex instruction set computer), and stack based.

A RISC machine typically has many registers, three-address instructions, simple addressing modes, and a relatively simple instruction-set architecture. In contrast, a CISC machine typically has few registers, two-address instructions, a variety of addressing modes, several register classes, variable-length instructions, and instructions with side effects.

In a stack-based machine, operations are done by pushing operands onto a stack and then performing the operations on the operands at the top of the stack. To achieve high performance the top of the stack is typically kept in registers. Stack-based machines almost disappeared because it was felt that the stack organization was too limiting and required too many swap and copy operations.

However, stack-based architectures were revived with the introduction of the Java Virtual Machine (JVM). The JVM is a software interpreter for Java bytecodes, an intermediate language produced by Java compilers. The interpreter provides software compatibility across multiple platforms, a major factor in the success of Java.

To overcome the high performance penalty of interpretation, which can be on the order of a factor of 10, *just-in-time* (JIT) Java compilers have been created. These JIT compilers translate bytecodes during run time to the native hardware instruction set of the target machine. Another approach to improving Java performance is to build a compiler that compiles directly into the machine instructions of the target machine, bypassing the Java bytecodes entirely.

Producing an absolute machine-language program as output has the advantage that it can be placed in a fixed location in memory and immediately executed. Programs can be compiled and executed quickly.

Producing a relocatable machine-language program (often called an *object module*) as output allows subprograms to be compiled separately. A set of relocatable object modules can be linked together and loaded for execution by a linking loader. Although we must pay the added expense of linking and loading if we produce relocatable object modules, we gain a great deal of flexibility in being able to compile subroutines separately and to call other previously compiled programs from an object module. If the target machine does not handle relocation automatically, the compiler must provide explicit relocation information to the loader to link the separately compiled program modules.

Producing an assembly-language program as output makes the process of code generation somewhat easier. We can generate symbolic instructions and use the macro facilities of the assembler to help generate code. The price paid is the assembly step after code generation.

In this chapter, we shall use a very simple RISC-like computer as our target machine. We add to it some CISC-like addressing modes so that we can also discuss code-generation techniques for CISC machines. For readability, we use assembly code as the target language . As long as addresses can be calculated from offsets and other information stored in the symbol table, the code generator can produce relocatable or absolute addresses for names just as easily as symbolic addresses.

8.1.3 Instruction Selection

The code generator must map the IR program into a code sequence that can be executed by the target machine. The complexity of performing this mapping is determined by a factors such as

- the level of the IR
- the nature of the instruction-set architecture
- the desired quality of the generated code.

If the IR is high level, the code generator may translate each IR statement into a sequence of machine instructions using code templates. Such statement-by-statement code generation, however, often produces poor code that needs further optimization. If the IR reflects some of the low-level details of the underlying machine, then the code generator can use this information to generate more efficient code sequences.

The nature of the instruction set of the target machine has a strong effect on the difficulty of instruction selection. For example, the uniformity and completeness of the instruction set are important factors. If the target machine does not support each data type in a uniform manner, then each exception to the general rule requires special handling. On some machines, for example, floating-point operations are done using separate registers.

Instruction speeds and machine idioms are other important factors. If we do not care about the efficiency of the target program, instruction selection is straightforward. For each type of three-address statement, we can design a code skeleton that defines the target code to be generated for that construct. For example, every three-address statement of the form x = y + z, where x, y, and z are statically allocated, can be translated into the code sequence

```
LD  R0, y        // R0 = y        (load y into register R0)
ADD R0, R0, z    // R0 = R0 + z   (add z to R0)
ST  x, R0        // x = R0        (store R0 into x)
```

This strategy often produces redundant loads and stores. For example, the sequence of three-address statements

```
a = b + c
d = a + e
```

would be translated into

```
LD  R0, b        // R0 = b
ADD R0, R0, c    // R0 = R0 + c
ST  a, R0        // a = R0
LD  R0, a        // R0 = a
ADD R0, R0, e    // R0 = R0 + e
ST  d, R0        // d = R0
```

Here, the fourth statement is redundant since it loads a value that has just been stored, and so is the third if a is not subsequently used.

The quality of the generated code is usually determined by its speed and size. On most machines, a given IR program can be implemented by many different code sequences, with significant cost differences between the different implementations. A naive translation of the intermediate code may therefore lead to correct but unacceptably inefficient target code.

For example, if the target machine has an "increment" instruction (INC), then the three-address statement a = a + 1 may be implemented more efficiently by the single instruction INC a, rather than by a more obvious sequence that loads a into a register, adds one to the register, and then stores the result back into a:

```
LD  R0, a        // R0 = a
ADD R0, R0, #1   // R0 = R0 + 1
ST  a, R0        // a = R0
```

We need to know instruction costs in order to design good code sequences but, unfortunately, accurate cost information is often difficult to obtain. Deciding which machine-code sequence is best for a given three-address construct may also require knowledge about the context in which that construct appears.

In Section 8.9 we shall see that instruction selection can be modeled as a tree-pattern matching process in which we represent the IR and the machine instructions as trees. We then attempt to "tile" an IR tree with a set of subtrees that correspond to machine instructions. If we associate a cost with each machine-instruction subtree, we can use dynamic programming to generate optimal code sequences. Dynamic programming is discussed in Section 8.11.

8.1.4 Register Allocation

A key problem in code generation is deciding what values to hold in what registers. Registers are the fastest computational unit on the target machine, but we usually do not have enough of them to hold all values. Values not held in registers need to reside in memory. Instructions involving register operands are invariably shorter and faster than those involving operands in memory, so efficient utilization of registers is particularly important.

The use of registers is often subdivided into two subproblems:

1. *Register allocation*, during which we select the set of variables that will reside in registers at each point in the program.

2. *Register assignment*, during which we pick the specific register that a variable will reside in.

Finding an optimal assignment of registers to variables is difficult, even with single-register machines. Mathematically, the problem is NP-complete. The problem is further complicated because the hardware and/or the operating system of the target machine may require that certain register-usage conventions be observed.

Example 8.1: Certain machines require *register-pairs* (an even and next odd-numbered register) for some operands and results. For example, on some machines, integer multiplication and integer division involve register pairs. The multiplication instruction is of the form

```
M x, y
```

where x, the multiplicand, is the even register of an even/odd register pair and y, the multiplier, is the odd register. The product occupies the entire even/odd register pair. The division instruction is of the form

```
D x, y
```

where the dividend occupies an even/odd register pair whose even register is x; the divisor is y. After division, the even register holds the remainder and the odd register the quotient.

Now, consider the two three-address code sequences in Fig. 8.2 in which the only difference in (a) and (b) is the operator in the second statement. The shortest assembly-code sequences for (a) and (b) are given in Fig. 8.3.

```
t = a + b        t = a + b
t = t * c        t = t + c
t = t / d        t = t / d

   (a)              (b)
```

Figure 8.2: Two three-address code sequences

```
L   R1,a         L    R0, a
A   R1,b         A    R0, b
M   R0,c         A    R0, c
D   R0,d         SRDA R0, 32
ST  R1,t         D    R0, d
                 ST   R1, t

   (a)              (b)
```

Figure 8.3: Optimal machine-code sequences

R*i* stands for register *i*. SRDA stands for Shift-Right-Double-Arithmetic and SRDA R0,32 shifts the dividend into R1 and clears R0 so all bits equal its sign bit. L, ST, and A stand for load, store, and add, respectively. Note that the optimal choice for the register into which a is to be loaded depends on what will ultimately happen to t. □

Strategies for register allocation and assignment are discussed in Section 8.8. Section 8.10 shows that for certain classes of machines we can construct code sequences that evaluate expressions using as few registers as possible.

8.1.5 Evaluation Order

The order in which computations are performed can affect the efficiency of the target code. As we shall see, some computation orders require fewer registers to hold intermediate results than others. However, picking a best order in the general case is a difficult NP-complete problem. Initially, we shall avoid the problem by generating code for the three-address statements in the order in which they have been produced by the intermediate code generator. In Chapter 10, we shall study code scheduling for pipelined machines that can execute several operations in a single clock cycle.

8.2 The Target Language

Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator. Unfortunately, in a general discussion of code generation it is not possible to describe any target machine in sufficient detail to generate good code for a complete language on that machine. In this chapter, we shall use as a target language assembly code for a simple computer that is representative of many register machines. However, the code-generation techniques presented in this chapter can be used on many other classes of machines as well.

8.2.1 A Simple Target Machine Model

Our target computer models a three-address machine with load and store operations, computation operations, jump operations, and conditional jumps. The underlying computer is a byte-addressable machine with n general-purpose registers, R0, R1, ... , R$n-1$. A full-fledged assembly language would have scores of instructions. To avoid hiding the concepts in a myriad of details, we shall use a very limited set of instructions and assume that all operands are integers. Most instructions consists of an operator, followed by a target, followed by a list of source operands. A label may precede an instruction. We assume the following kinds of instructions are available:

- *Load* operations: The instruction LD *dst, addr* loads the value in location *addr* into location *dst*. This instruction denotes the assignment *dst* = *addr*. The most common form of this instruction is LD r, x which loads the value in location x into register r. An instruction of the form LD r_1, r_2 is a *register-to-register copy* in which the contents of register r_2 are copied into register r_1.

- *Store* operations: The instruction ST x, r stores the value in register r into the location x. This instruction denotes the assignment $x = r$.

- *Computation* operations of the form *OP dst*, src_1, src_2, where *OP* is a operator like ADD or SUB, and *dst*, src_1, and src_2 are locations, not necessarily distinct. The effect of this machine instruction is to apply the operation represented by *OP* to the values in locations src_1 and src_2, and place the result of this operation in location *dst*. For example, SUB r_1, r_2, r_3 computes $r_1 = r_2 - r_3$. Any value formerly stored in r_1 is lost, but if r_1 is r_2 or r_3, the old value is read first. Unary operators that take only one operand do not have a src_2.

- *Unconditional jumps*: The instruction BR L causes control to branch to the machine instruction with label L. (BR stands for *branch.*)

- *Conditional jumps* of the form B*cond* r, L, where r is a register, L is a label, and *cond* stands for any of the common tests on values in the register r. For example, BLTZ r, L causes a jump to label L if the value in register r is less than zero, and allows control to pass to the next machine instruction if not.

We assume our target machine has a variety of addressing modes:

- In instructions, a location can be a variable name x referring to the memory location that is reserved for x (that is, the l-value of x).

- A location can also be an indexed address of the form $a(r)$, where a is a variable and r is a register. The memory location denoted by $a(r)$ is computed by taking the l-value of a and adding to it the value in register r. For example, the instruction `LD R1, a(R2)` has the effect of setting R1 $=$ *contents*(a $+$ *contents*(R2)), where *contents*(x) denotes the contents of the register or memory location represented by x. This addressing mode is useful for accessing arrays, where a is the base address of the array (that is, the address of the first element), and r holds the number of bytes past that address we wish to go to reach one of the elements of array a.

- A memory location can be an integer indexed by a register. For example, `LD R1, 100(R2)` has the effect of setting R1 $=$ *contents*$(100 +$ *contents*(R2)), that is, of loading into R1 the value in the memory location obtained by adding 100 to the contents of register R2. This feature is useful for following pointers, as we shall see in the example below.

- We also allow two indirect addressing modes: $*r$ means the memory location found in the location represented by the contents of register r and $*100(r)$ means the memory location found in the location obtained by adding 100 to the contents of r. For example, `LD R1, *100(R2)` has the effect of setting R1 $=$ *contents*(*contents*$(100 +$ *contents*(R2))), that is, of loading into R1 the value in the memory location stored in the memory location obtained by adding 100 to the contents of register R2.

- Finally, we allow an immediate constant addressing mode. The constant is prefixed by #. The instruction `LD R1, #100` loads the integer 100 into register R1, and `ADD R1, R1, #100` adds the integer 100 into register R1.

Comments at the end of instructions are preceded by //.

Example 8.2: The three-address statement `x = y - z` can be implemented by the machine instructions:

```
LD  R1, y          // R1 = y
LD  R2, z          // R2 = z
SUB R1, R1, R2     // R1 = R1 - R2
ST  x, R1          // x = R1
```

We can do better, perhaps. One of the goals of a good code-generation algorithm is to avoid using all four of these instructions, whenever possible. For example, y and/or z may have been computed in a register, and if so we can avoid the LD step(s). Likewise, we might be able to avoid ever storing x if its value is used within the register set and is not subsequently needed.

Suppose a is an array whose elements are 8-byte values, perhaps real numbers. Also assume elements of a are indexed starting at 0. We may execute the three-address instruction b = a[i] by the machine instructions:

```
LD  R1, i          // R1 = i
MUL R1, R1, 8      // R1 = R1 * 8
LD  R2, a(R1)      // R2 = contents(a + contents(R1))
ST  b, R2          // b = R2
```

That is, the second step computes $8i$, and the third step places in register R2 the value in the ith element of a — the one found in the location that is $8i$ bytes past the base address of the array a.

Similarly, the assignment into the array a represented by three-address instruction a[j] = c is implemented by:

```
LD  R1, c          // R1 = c
LD  R2, j          // R2 = j
MUL R2, R2, 8      // R2 = R2 * 8
ST  a(R2), R1      // contents(a +  contents(R2)) = R1
```

To implement a simple pointer indirection, such as the three-address statement x = *p, we can use machine instructions like:

```
LD  R1, p          // R1 = p
LD  R2, 0(R1)      // R2 = contents(0 + contents(R1))
ST  x, R2          // x = R2
```

The assignment through a pointer *p = y is similarly implemented in machine code by:

```
LD  R1, p          // R1 = p
LD  R2, y          // R2 = y
ST  0(R1), R2      // contents(0 + contents(R1)) = R2
```

Finally, consider a conditional-jump three-address instruction like

```
if x < y goto L
```

The machine-code equivalent would be something like:

```
LD   R1, x          // R1 = x
LD   R2, y          // R2 = y
SUB  R1, R1, R2     // R1 = R1 - R2
BLTZ R1, M          // if R1 < 0 jump to M
```

Here, `M` is the label that represents the first machine instruction generated from the three-address instruction that has label `L`. As for any three-address instruction, we hope that we can save some of these machine instructions because the needed operands are already in registers or because the result need never be stored. □

8.2.2 Program and Instruction Costs

We often associate a cost with compiling and running a program. Depending on what aspect of a program we are interested in optimizing, some common cost measures are the length of compilation time and the size, running time and power consumption of the target program.

Determining the actual cost of compiling and running a program is a complex problem. Finding an optimal target program for a given source program is an undecidable problem in general, and many of the subproblems involved are NP-hard. As we have indicated, in code generation we must often be content with heuristic techniques that produce good but not necessarily optimal target programs.

For the remainder of this chapter, we shall assume each target-language instruction has an associated cost. For simplicity, we take the cost of an instruction to be one plus the costs associated with the addressing modes of the operands. This cost corresponds to the length in words of the instruction. Addressing modes involving registers have zero additional cost, while those involving a memory location or constant in them have an additional cost of one, because such operands have to be stored in the words following the instruction. Some examples:

- The instruction `LD R0, R1` copies the contents of register `R1` into register `R0`. This instruction has a cost of one because no additional memory words are required.

- The instruction `LD R0, M` loads the contents of memory location `M` into register `R0`. The cost is two since the address of memory location `M` is in the word following the instruction.

- The instruction `LD R1, *100(R2)` loads into register `R1` the value given by $contents(contents(100 + contents(\texttt{R2})))$. The cost is three because the constant 100 is stored in the word following the instruction.

In this chapter we assume the cost of a target-language program on a given input is the sum of costs of the individual instructions executed when the program is run on that input. Good code-generation algorithms seek to minimize the sum of the costs of the instructions executed by the generated target program on typical inputs. We shall see that in some situations we can actually generate optimal code for expressions on certain classes of register machines.

8.2.3 Exercises for Section 8.2

Exercise 8.2.1: Generate code for the following three-address statements assuming all variables are stored in memory locations.

a) `x = 1`

b) `x = a`

c) `x = a + 1`

d) `x = a + b`

e) The two statements

```
x = b * c
y = a + x
```

Exercise 8.2.2: Generate code for the following three-address statements assuming *a* and *b* are arrays whose elements are 4-byte values.

a) The four-statement sequence

```
x = a[i]
y = b[j]
a[i] = y
b[j] = x
```

b) The three-statement sequence

```
x = a[i]
y = b[i]
z = x * y
```

c) The three-statement sequence

```
x = a[i]
y = b[x]
a[i] = y
```

Exercise 8.2.3: Generate code for the following three-address sequence assuming that p and q are in memory locations:

```
    y = *q
    q = q + 4
    *p = y
    p = p + 4
```

Exercise 8.2.4: Generate code for the following sequence assuming that x, y, and z are in memory locations:

```
    if x < y goto L1
    z = 0
    goto L2
L1: z = 1
```

Exercise 8.2.5: Generate code for the following sequence assuming hat n is in a memory location:

```
    s = 0
    i = 0
L1: if i > n goto L2
    s = s + i
    i = i + 1
    goto L1
L2:
```

Exercise 8.2.6: Determine the costs of the following instruction sequences:

a)
```
LD  R0, y
LD  R1, z
ADD R0, R0, R1
ST  x, R0
```

b)
```
LD  R0, i
MUL R0, R0, 8
LD  R1, a(R0)
ST  b, R1
```

c)
```
LD  R0, c
LD  R1, i
MUL R1, R1, 8
ST  a(R1), R0
```

d)
```
LD R0, p
LD R1, 0(R0)
ST x, R1
```

e)

```
LD R0, p
LD R1, x
ST 0(R0), R1
```

f)

```
LD   R0, x
LD   R1, y
SUB  R0, R0, R1
BLTZ *R3, R0
```

8.3 Addresses in the Target Code

In this section, we show how names in the IR can be converted into addresses in the target code by looking at code generation for simple procedure calls and returns using static and stack allocation. In Section 7.1, we described how each executing program runs in its own logical address space that was partitioned into four code and data areas:

1. A statically determined area *Code* that holds the executable target code. The size of the target code can be determined at compile time.

2. A statically determined data area *Static* for holding global constants and other data generated by the compiler. The size of the global constants and compiler data can also be determined at compile time.

3. A dynamically managed area *Heap* for holding data objects that are allocated and freed during program execution. The size of the *Heap* cannot be determined at compile time.

4. A dynamically managed area *Stack* for holding activation records as they are created and destroyed during procedure calls and returns. Like the *Heap*, the size of the *Stack* cannot be determined at compile time.

8.3.1 Static Allocation

To illustrate code generation for simplified procedure calls and returns, we shall focus on the following three-address statements:

- `call` *callee*
- `return`
- `halt`
- `action`, which is a placeholder for other three-address statements.

The size and layout of activation records are determined by the code generator via the information about names stored in the symbol table. We shall first illustrate how to store the return address in an activation record on a procedure call and how to return control to it after the procedure call. For convenience, we assume the first location in the activation holds the return address.

Let us first consider the code needed to implement the simplest case, static allocation. Here, a `call` *callee* statement in the intermediate code can be implemented by a sequence of two target-machine instructions:

```
ST  callee.staticArea, #here + 20
BR  callee.codeArea
```

The ST instruction saves the return address at the beginning of the activation record for *callee*, and the BR transfers control to the target code for the called procedure *callee*. The attribute before *callee.staticArea* is a constant that gives the address of the beginning of the activation record for *callee*, and the attribute *callee.codeArea* is a constant referring to the address of the first instruction of the called procedure *callee* in the *Code* area of the run-time memory.

The operand #*here* + 20 in the ST instruction is the literal return address; it is the address of the instruction following the BR instruction. We assume that #*here* is the address of the current instruction and that the three constants plus the two instructions in the calling sequence have a length of 5 words or 20 bytes.

The code for a procedure ends with a return to the calling procedure, except that the first procedure has no caller, so its final instruction is HALT, which returns control to the operating system. A `return` *callee* statement can be implemented by a simple jump instruction

```
BR  *callee.staticArea
```

which transfers control to the address saved at the beginning of the activation record for *callee*.

Example 8.3: Suppose we have the following three-address code:

```
                // code for c
action1
call p
action2
halt
                // code for p
action3
return
```

Figure 8.4 shows the target program for this three-address code. We use the pseudoinstruction ACTION to represent the sequence of machine instructions to execute the statement `action`, which represents three-address code that is not relevant for this discussion. We arbitrarily start the code for procedure c at address 100 and for procedure p at address 200. We that assume each ACTION instruction takes 20 bytes. We further assume that the activation records for these procedures are statically allocated starting at locations 300 and 364, respectively.

The instructions starting at address 100 implement the statements

```
action1; call p; action2; halt
```

of the first procedure c. Execution therefore starts with the instruction ACTION$_1$ at address 100. The ST instruction at address 120 saves the return address 140 in the machine-status field, which is the first word in the activation record of p. The BR instruction at address 132 transfers control the first instruction in the target code of the called procedure p.

```
                              // code for c
100:  ACTION1                 // code for action1
120:  ST 364, #140            // save return address 140 in location 364
132:  BR 200                  // call p
140:  ACTION2
160:  HALT                    // return to operating system
      ...
                              // code for p
200:  ACTION3
220:  BR *364                 // return to address saved in location 364
      ...
                              // 300-363 hold activation record for c
300:                          // return address
304:                          // local data for c
      ...
                              // 364-451 hold activation record for p
364:                          // return address
368:                          // local data for p
```

Figure 8.4: Target code for static allocation

After executing ACTION$_3$, the jump instruction at location 220 is executed. Since location 140 was saved at address 364 by the call sequence above, *364 represents 140 when the BR statement at address 220 is executed. Therefore, when procedure p terminates, control returns to address 140 and execution of procedure c resumes. □

8.3.2 Stack Allocation

Static allocation can become stack allocation by using relative addresses for storage in activation records. In stack allocation, however, the position of an activation record for a procedure is not known until run time. This position is usually stored in a register, so words in the activation record can be accessed as offsets from the value in this register. The indexed address mode of our target machine is convenient for this purpose.

Relative addresses in an activation record can be taken as offsets from any known position in the activation record, as we saw in Chapter 7. For conve-