Chapter 8

Machine-Code Generation

8.1 Introduction

The intermediate language we have used in chapter 7 is quite low-level and similar to the type of machine code you can find on modern RISC processors, with a few exceptions:

- We have used an unbounded number of variables, where a processor will have a bounded number of registers.
- We have used a complex `CALL` instruction for function calls.
- In the intermediate language, the `IF-THEN-ELSE` instruction has two target labels, where, on most processors, the conditional jump instruction has only one target label, and simply falls through to the next instruction when the condition is false.
- We have assumed that any constant can be an operand to an arithmetic instruction. Typically, RISC processors allow only small constants as operands.

The problem of mapping a large set of variables to a small number of registers is handled by *register allocation*, as explained in chapter 9. Function calls are treated in chapter 10. We will look at the remaining two problems below.

The simplest solution for generating machine code from intermediate code is to translate each intermediate-language instruction into one or more machine-code instructions. However, it is often possible to find a machine-code instruction that covers two or more intermediate-language instructions. We will in section 8.4 see how we can exploit complex instructions in this way.

Additionally, we will briefly discuss other optimisations.

8.2 Conditional jumps

Conditional jumps come in many forms on different machines. Some conditional jump instructions embody a relational comparison between two registers (or a register and a constant) and are, hence, similar to the `IF-THEN-ELSE` instruction in our intermediate language. Other types of conditional jump instructions require the condition to be already resolved and stored in special condition registers or flags. However, it is almost universal that conditional jump instructions specify only one target label (or address), typically used when the condition is true. When the condition is false, execution simply continues with the instructions immediately following the conditional jump instruction.

Converting two-way branches to one-way branches is not terribly difficult: `IF` c `THEN` l_t `ELSE` l_f can be translated to

```
branch_if_c  l_t
jump         l_f
```

where `branch_if_c` is a conditional instruction that jumps when the condition c is true and `jump` is an unconditional jump.

Oftenm, an `IF-THEN-ELSE` instruction is immediately followed by one of its target labels. In fact, this will always be the case if the intermediate code is generated by the translation functions shown in chapter 7 (see exercise 7.5). If this label happens to be l_f (the label taken for false conditions), we can simply omit the unconditional `jump` from the code shown above. If the following label is l_t, we can negate the condition of the conditional jump and make it jump to l_f, *i.e.*, as

```
branch_if_not_c  l_f
```

where `branch_if_not_c` is a conditional instruction that jumps when the condition c is false.

Hence, the code generator (the part of the compiler that generates machine code) should test which (if any) of the target labels follow an `IF-THEN-ELSE` instruction and translate it accordingly. Alternatively, a post-processing pass can be made over the generated machine code to remove superfluous jumps.

If the conditional jump instructions in the target machine language do not allow conditions as complex as those used in the intermediate language, code must be generated to first calculate the condition and put the result somewhere where it can be tested by a subsequent conditional jump instruction. In some machine architectures, *e.g.*, MIPS and Alpha, this "somewhere" can be a general-purpose register. Other machines, *e.g.*, PowerPC or IA-64 (also known as Itanium) use special condition registers, while yet others, *e.g.*, IA-32 (also known as x86), Sparc, PA-RISC and ARM use a single set of arithmetic flags that can be set by comparison or arithmetic instructions. A conditional jump may test various combinations of the flags, so the same comparison instruction can, depending on the subsequent condition, be used for testing equality, signed or unsigned less-than, overflow and several other properties. Usually, any `IF-THEN-ELSE` instruction can be translated into at most two instructions: One that does the comparison and one that does the conditional jump.

8.3 Constants

The intermediate language allows arbitrary constants as operands to binary or unary operators. This is not always the case in machine code.

For example, MIPS allows only 16-bit constants in operands even though integers are 32 bits (64 bits in some versions of the MIPS architecture). To build larger constants, MIPS includes instructions to load 16-bit constants into the upper half (the most significant bits) of a register. With help of these, an arbitrary 32-bit integer can be entered into a register using two instructions. On the ARM, a constant can be an 8-bit number positioned at any even bit boundary. It may take up to four instructions to build a 32-bit number using these.

When an intermediate-language instruction uses a constant, the code generator must check if it fits into the constant field (if any) of the equivalent machine-code instruction. If it does, the code generator genrates a single machine-code instruction. If not, the code generator generates a sequence of instructions that builds the constant in a register, followed by an instruction that uses this register in place of the constant. If a complex constant is used inside a loop, it may be a good idea to move the code for generating this outside the loop and keep it in a register inside the loop. This can be done as part of a general optimisation to move code out of loops, see section 8.5.

8.4 Exploiting complex instructions

Most instructions in our intermediate language are *atomic*, in the sense that each instruction corresponds to a single operation which can not sensibly be split into smaller steps. The exceptions to this rule are the instructions `IF-THEN-ELSE`, which we in section 8.2 described how to handle, and `CALL`, which will be detailed in chapter 10.

CISC (Complex Instruction Set Computer) processors like IA-32 have composite (*i.e.*, non-atomic) instructions in abundance. And while the philosophy behind RISC (Reduced Instruction Set Computer) processors like MIPS and ARM advocates that machine-code instructions should be simple, most RISC processors include at least a few non-atomic instructions, typically for memory-access instructions.

We will in this chapter use a subset of the MIPS instruction set as an example. A description of the MIPS instruction set can be found Appendix A of [39], which is available online [27]. If you are not already familiar with the MIPS instruction set, it would be a good idea to read the description before continuing.

To exploit composite instructions, several intermediate-language instructions can be grouped together and translated into a single machine-code instruction. For example, the intermediate-language instruction sequence

$$\begin{array}{l} t_2 := t_1 + 116 \\ t_3 := M[t_2] \end{array}$$

can be translated into the single MIPS instruction

```
lw r3, 116(r1)
```

where `r1` and `r3` are the registers chosen for t_1 and t_3, respectively. However, is is only possible to combine the two instructions if the value of the intermediate value t_2 is not required later, as the combined instruction does not store this value anywhere.

We will, hence, need to know if the contents of a variable is required for later use, or if it is *dead* after a particular use. When generating intermediate code, most of the temporary variables introduced by the compiler will be single-use and can be marked as such. Any use of a single-use variable will, by definition, be the last use. Alternatively, last-use information can be obtained by analysing the intermediate code using a *liveness analysis*, which we will describe in chapter 9. For now, we will just assume that the last use of any variable is marked in the intermediate code. We assume this is done, and the last use of any variable in the intermediate code is marked by *last*, such as t^{last}, which indicates that this is the last use of the variable t.

Our next step is to describe each machine-code instruction in terms of one or more intermediate-language instructions. We call the sequence of intermediate-language instructions a *pattern*, and the corresponding machine-code instruction its *replacement*, since the idea is to find sequences in the intermediate code that matches the pattern and replace these sequences by instances of the replacement. When a pattern uses variables such as k, t or r_d, these can match any intermediate-language constants, variables or labels, and when the same variable is used in both pattern and replacement, it means that the corresponding intermediate-language constant or variable/label name is copied to the machine-code instruction, where it will represent a constant, a named register or a machine-code label.

For example, the MIPS `lw` (load word) instruction can be described by the pattern/replacement pair

$t := r_s + k$ $r_t := M[t^{last}]$	`lw` $r_t,\ k(r_s)$

where t^{last} in the pattern indicates that the contents of t must not be used afterwards, *i.e.*, that the intermediate-language variable that is matched against t must have a *last* annotation at this place. A pattern can only match a piece of intermediate code if all *last* annotations in the pattern are matched by *last* annotations in the intermediate code. The converse, however, need not hold: It is not harmful to store a value in a register even if it is not used later, so a *last* annotation in the intermediate code need not be matched by a *last* annotation in the pattern.

The list of patterns that in combination describe the machine-code instruction set must cover the intermediate language in full (excluding function calls, which we handle in chapter 10). In particular, each single intermediate-language instruction (with the exception of `CALL`, which we handle separately in chapter 10) must be covered by at least one pattern. This means that we must include the MIPS instruction `lw` $r_t,\ 0(r_s)$ to cover the intermediate-code instruction $r_t := M[r_s]$, even though we have already listed a more general form of `lw`. If there is an intermediate-language instruction for which there are no equivalent single machine-code instruction, a sequence of machine-code instructions must be given for this. Hence, an instruction-set description is a list of pairs, where each pair consists of a *pattern* (a sequence of intermediate-language instructions) and a *replacement* (a sequence of machine-code instructions).

When translating a sequence of intermediate-code instructions, the code generator can look at the patterns and pick the replacement that covers the largest prefix of the intermediate code. A simple way of ensuring that the longest prefix is matched is to list the pairs so longer patterns are listed before shorter patterns. The first pattern in the list that matches a prefix of the intermediate code will now also be the longest mathing pattern.

This kind of algorithm is called *greedy*, because it always picks the choice that is best for immediate profit, *i.e.*, the sequence that "eats" most of the intermediate code in one bite. It will, however, not always yield the best possible solution for the total sequence of intermediate-language instructions.

If costs are given for each machine-code instruction sequence in the pattern/replacement pairs, optimal (*i.e.*, least-cost) solutions can be found for straight-line (*i.e.*, jump-free) code sequences. The least-cost sequence that covers the intermediate code can be found, *e.g.*, using a dynamic-programming algorithm. For RISC processors, a greedy algorithm will typically get close to optimal solutions, so the gain from using a better algorithm is small. Hence, we will go into detail only for the greedy algorithm.

As an example, figure 8.1 describes a subset of the instructions for the MIPS microprocessor architecture in terms of the intermediate language as a set of pattern/replacement pairs. Note that we exploit the fact that register 0 is hardwired to be the value 0 to, *e.g.*, use the `addi` instruction to generate a constant. We assume that we, at this point, have already handled the problem of too-large constants, so any constant that now remains in the intermediate code can be used as an immediate constant in an instruction such a `addi`. Note that we make special cases for `IF-THEN-ELSE` when one of the labels immediately follows the test. Note, also, that we need (at least) two instructions from our MIPS subset to implement an `IF-THEN-ELSE` instruction that uses $<$ as the relational operator, while we need only one for comparison by $=$. Figure 8.1 does not cover all of the intermediate language, but it can fairly easily be extended to do so. It is also possible to add more special cases to exploit a larger subset of the MIPS instruction set.

The instructions in figure 8.1 are listed so that, when two patterns overlap, the longest of these is listed first. Overlap can happen id the pattern in one pair is a prefix of the pattern for another pair, as is the case with the pairs involving `addi` and `lw`/`sw` and for the different instances of `beq`/`bne` and `slt`.

We can try to use figure 8.1 to select MIPS instructions for the following sequence of intermediate-language instructions:

$$\begin{array}{l}
a := a + b^{last} \\
d := c + 8 \\
M[d^{last}] := a \\
\texttt{IF } a = c \texttt{ THEN } label_1 \texttt{ ELSE } label_2 \\
\texttt{LABEL } label_2
\end{array}$$

Only one pattern (for the `add` instruction) in figure 8.1 matches a prefix of this code, so we generate an `add` instruction for the first intermediate instruction. We now have two matches for prefixes of the remaining code: One using `sw` and one using `addi`. Since the pattern using `sw` is listed first in the table, we choose this to replace the next two intermediate-language instructions. Finally, a `beq` instruction matches the last two instructions. Hence, we generate the code

$$\begin{array}{lll}
 & \texttt{add} & a,\ a,\ b \\
 & \texttt{sw} & a,\ 8(c) \\
 & \texttt{beq} & a,\ c,\ label_1 \\
label_2: & &
\end{array}$$

Note that we retain $label_2$ even though the resulting sequence does not refer to it, as some other part of the code might jump to it. We could include single-use annotations for labels like we use for variables, but it is hardly worth the effort, as labels do not generate actual code and hence cost nothing[1].

[1]This is, strictly speaking, not entirely true, as superfluous labels might inhibit later optimisations.

$t := r_s + k$, $r_t := M[t^{last}]$	`lw` r_t, $k(r_s)$
$r_t := M[r_s]$	`lw` r_t, $0(r_s)$
$r_t := M[k]$	`lw` r_t, k(`R0`)
$t := r_s + k$, $M[t^{last}] := r_t$	`sw` r_t, $k(r_s)$
$M[r_s] := r_t$	`sw` r_t, $0(r_s)$
$M[k] := r_t$	`sw` r_t, k(`R0`)
$r_d := r_s + r_t$	`add` r_d, r_s, r_t
$r_d := r_t$	`add` r_d, `R0`, r_t
$r_d := r_s + k$	`addi` r_d, r_s, k
$r_d := k$	`addi` r_d, `R0`, k
`GOTO` *label*	`j` *label*
`IF` $r_s = r_t$ `THEN` $label_t$ `ELSE` $label_f$, `LABEL` $label_f$	`beq` r_s, r_t, $label_t$ $label_f$:
`IF` $r_s = r_t$ `THEN` $label_t$ `ELSE` $label_f$, `LABEL` $label_t$	`bne` r_s, r_t, $label_f$ $label_t$:
`IF` $r_s = r_t$ `THEN` $label_t$ `ELSE` $label_f$	`beq` r_s, r_t, $label_t$ `j` $label_f$
`IF` $r_s < r_t$ `THEN` $label_t$ `ELSE` $label_f$, `LABEL` $label_f$	`slt` r_d, r_s, r_t `bne` r_d, `R0`, $label_t$ $label_f$:
`IF` $r_s < r_t$ `THEN` $label_t$ `ELSE` $label_f$, `LABEL` $label_t$	`slt` r_d, r_s, r_t `beq` r_d, `R0`, $label_f$ $label_t$:
`IF` $r_s < r_t$ `THEN` $label_t$ `ELSE` $label_f$	`slt` r_d, r_s, r_t `bne` r_d, `R0`, $label_t$ `j` $label_f$
`LABEL` *label*	*label*:

Figure 8.1: Pattern/replacement pairs for a subset of the MIPS instruction set

8.4.1 Two-address instructions

In the above we have assumed that the machine code is three-address code, *i.e.*, that the destination register of an instruction can be distinct from the two operand registers. It is, however, not uncommon that processors use two-address code, where the destination register is the same as the first operand register. To handle this, we use pattern/replacement pairs like these:

$r_t := r_s$	`mov` $r_t,\ r_s$
$r_t := r_t + r_s$	`add` r_t, r_s
$r_d := r_s + r_t$	`move` $r_d,\ r_s$ `add` r_d, r_t

that add copy instructions in the cases where the destination register is not the same as the first operand. As we will see in chapter 9, the register allocator will often be able to remove the added copy instruction by allocating r_d and r_s in the same register.

Processors that divide registers into data and address registers or integer and floating-point registers can be handled in a similar way: Add instructions that copy to new registers before operations and let register allocation allocate these to the right type of registers (and eliminate as many of the moves as possible).

Suggested exercises: 8.2.

8.5 Optimisations

Optimisations can be done by a compiler in three places: In the source code (*i.e.*, on the abstract syntax), in the intermediate code, and in the machine code. Some optimisations can be specific to the source language or the machine language, but it makes sense to perform optimisations mainly in the intermediate language, as the optimisations hence can be shared among all compilers that use the same intermediate language. Also, the intermediate language is typically simpler than both the source language and the machine language, making the effort of doing optimisations smaller.

Optimising compilers have a wide array of optimisations that they can employ, but we will mention only a few and just hint at how they can be implemented.

Common subexpression elimination. In the statement `a[i] := a[i]+2`, the address for `a[i]` is calculated twice. This double calculation can be eliminated by storing the address in a temporary variable when the address is first calculated, and then use this variable instead of calculating the address again. Simple methods for common subexpression elimination work on *basic blocks*, *i.e.*, straight-line code without jumps or labels, but more advanced methods can eliminate duplicated calculations even across jumps.

Code hoisting. If part of the computation inside a loop is independent of the variables that change inside the loop, it can be moved outside the loop and only calculated once. For example, in the loop

```
while (j < k) {
  sum = sum + a[i][j];
  j++;
}
```

a large part of the address calculation for `a[i][j]` can be done without knowing `j`. This part can be moved outside the loop so it will only be calculated once. Note that this optimisation ca not be done on source-code level, as the address calculations are not visible there. For the same reason, the optimised version is not shown here.

If `k` may be less than or equal to `j`, the loop body may never be entered and we may, hence, unnecessarily execute the code that was moved out of the loop. This might even generate a run-time error. Hence, we can unroll the loop once to

```
if (j < k) {
  sum = sum + a[i][j];
  j++;
  while (j < k) {
    sum = sum + a[i][j];
    j++;
  }
}
```

The loop-independent part(s) may now without risk be calculated in the unrolled part and reused in the non-unrolled part. Again, this optimisation is not shown.

Constant propagation. A variable may, at some points in the program, have a value that is always equal to a known constant. When such a variable is used in a calculation, this calculation can often be simplified after replacing the variable by the constant that is guaranteed to be its value. Furthermore, the variable that holds the results of this computation may now also become constant, which may enable even more compile-time reduction.

Constant-propagation algorithms first trace the flow of constant values through the program, and then reduce calculations. More advanced methods also look at conditions, so they can exploit that after a test on, *e.g.*, `x = 0`, `x` is, indeed, the constant 0.

Index-check elimination. As mentioned in chapter 7, some compilers insert run-time checks to catch cases when an index is outside the bounds of the array. Some of these checks can be removed by the compiler. One way of doing this is to see if the tests on the index are subsumed by earlier tests or ensured by assignments. For example, assume that, in the loop shown above, `a` is declared to be a $\mathtt{k} \times \mathtt{k}$ array. This means that the entry test for the loop will ensure that `j` is always less than the upper bound on the array, so this part of the index test can be eliminated. If `j` is initialised to 0 before entering the loop, we can use this to conclude that we do not need to check the lower bound either.

8.6 Further reading

Code selection by pattern matching normally uses a tree-structured intermediate language instead of the linear instruction sequences we use in this book. This can avoid some problems where the order of unrelated instructions affect the quality of code generation. For example, if the two first instructions in the example at the end of section 8.4 are interchanged, our simple prefix-matching algorithm will not include the address calculation in the `sw` instruction and, hence, needs one more instruction. If the intermediate code is tree-structured, the order of independent instructions is left unspecified, and the code generator can choose whichever ordering gives the best code. See [35] or [9] for more details.

Descriptions of and methods for a large number of different optimisations can be found in [5], [35] and [9].

The instruction set of (one version of) the MIPS microprocessor architecture is described in [39]. This description is also available online [27].

Chapter 11 describes optimisation in more detail.

Exercises

Exercise 8.1

Add extra inherited attributes to $Trans_{Cond}$ in figure 7.8 that, for each of the two target labels, indicates if this label immediately follows the code for the condition, *i.e.*, a boolean-valued attribute for each of the two labels. Use this information to make sure that the false-destination labels of an `IF-THEN-ELSE` instruction follow immediately after the `IF-THEN-ELSE` instruction.

You can use the function *negate* to negate relational operators so, *i.e.*, $negate(<) = \geq$.

Make sure the new attributes are maintained in recursive calls and modify $Trans_{Stat}$ in figure 7.5 so it sets these attributes when calling $Trans_{Cond}$.

Exercise 8.2

Use figure 8.1 and the method described in section 8.4 to generate code for the following intermediate code sequence:

$d := c + 8$
$:= a + b^{last}$
$M[d^{last}] := a$
`IF` $a < c$ `THEN` $label_1$ `ELSE` $label_2$
`LABEL` $label_1$

Compare this to the example in section 8.4.

Exercise 8.3

In figures 7.3 and 7.5, identify guaranteed last-uses of temporary variables, *i.e.*, places where *last* annotations can be inserted safely.

Exercise 8.4

Choose an instruction set (other than MIPS) and make patterns for the same subset of the intermediate language as covered by figure 8.1. Use this to translate the intermediate-code example from section 8.4.

Exercise 8.5

In some microprocessors, aritmetic instructions use only two registers, as the destination register is the same as one of the argument registers. As an example, copy and addition instructions of such a processor can be described as follows (using notation like in figure 8.1):

$r_d := r_t$	`MOV` r_d, r_t
$r_d := r_d + r_t$	`ADD` r_d, r_t
$r_d := r_d + k$	`ADDI` r_d, k

As in MIPS, register 0 (`R0`) is hardwired to the value 0.

Add to the above table pattern/replacement pairs sufficient to translate the following intermediate-code instructions to sequences of machine-code instructions using only `MOV`, `ADD` and `ADDI` instructions in the replacement sequences:

$$\begin{array}{lcl} r_d & := & k \\ r_d & := & r_s + r_t \\ r_d & := & r_s + k \end{array}$$

Note that neither r_s nor r_t have the *last* annotation, so their values must be preserved. Note, also, that the intermediate-code instructions above are not a sequence, but a list of separate instructions, so you should generate code separately for each instruction.