MITOCW | MIT6_004S17_11-02-02_300k

A compiler is a program that translates a high-level language program into a functionally equivalent sequence of machine instructions, i.e., an assembly language program.

A compiler first checks that the high-level program is correct, i.e., that the statements are well formed, the programmer isn't asking for nonsensical computations, e.g., adding a string value and an integer, or attempting to use the value of a variable before it has been properly initialized. The compiler may also provide warnings when operations may not produce the expected results, e.g., when converting from a floating-point number to an integer, where the floating-point value may be too large to fit in the number of bits provided by the integer. If the program passes scrutiny, the compiler then proceeds to generate efficient sequences of instructions, often finding ways to rearrange the computation so that the resulting sequences are shorter and faster.

It's hard to beat a modern optimizing compiler at producing assembly language, since the compiler will patiently explore alternatives and deduce properties of the program that may not be apparent to even diligent assembly language programmers.

In this section, we'll look at a simple technique for compiling C programs into assembly.

Then, in the next section, we'll dive more deeply into how a modern compiler works.

There are two main routines in our simple compiler: compile_statement and compile_expr.

The job of compile_statement is to compile a single statement from the source program.

Since the source program is a sequence of statements, we'll be calling compile_statement repeatedly. We'll focus on the compilation technique for four types of statements. An unconditional statement is simply an expression that's evaluated once. A compound statement is simply a sequence of statements to be executed in turn. Conditional statements, sometimes called "if statements", compute the value of an test expression, e.g., a comparison such as "A B". If the test is true then statement_1 is executed, otherwise statement_2 is executed. Iteration statements also contain a test expression.

In each iteration, if the test true, then the statement is executed, and the process repeats. If the test is false, the iteration is terminated.

The other main routine is compile_expr whose job it is to generate code to compute the value of an expression, leaving the result in some register.

Expressions take many forms: simple constant values values from scalar or array variables, assignment expressions that compute a value and then store the result in some variable, unary or binary operations that combine the values of their operands with the specified operator.

. . .

Complex arithmetic expressions can be decomposed into sequences of unary and binary operations.

And, finally, procedure calls, where a named sequence of statements will be executed with the values of the supplied arguments assigned as the values for the formal parameters of the procedure. Compiling procedures and procedure calls is a topic that we'll tackle next lecture since there are some complications to understand and deal with. Happily, compiling the other types of expressions and statements is straightforward, so let's get started.

What code do we need to put the value of a constant into a register?

If the constant will fit into the 16-bit constant field of an instruction, we can use CMOVE to load the sign-extended constant into a register.

This approach works for constants between -32768 and +32767.

If the constant is too large, it's stored in a main memory location and we use a LD instruction to get the value into a register. Loading the value of a variable is much the same as loading the value of a large constant. We use a LD instruction to access the memory location that holds the value of the variable. Performing an array access is slightly more complicated: arrays are stored as consecutive locations in main memory, starting with index 0. Each element of the array occupies some fixed number bytes. So we need code to convert the array index into the actual main memory address for the specified array element.

We first invoke compile_expr to generate code that evaluates the index expression and leaves the result in Rx. That will be a value between 0 and the size of the array minus 1. We'll use a LD instruction to access the appropriate array entry, but that means we need to convert the index into a byte offset, which we do by multiplying the index by bsize, the number of bytes in one element.

If b was an array of integers, bsize would be 4.

Now that we have the byte offset in a register, we can use LD to add the offset to the base address of the array computing the address of the desired array element, then load the memory value at that address into a register. Assignment expressions are easy Invoke compile_expr to generate code that loads the value of the expression into a register, then generate a ST instruction to store the value into the specified variable.

Arithmetic operations are pretty easy too. Use compile_expr to generate code for each of the operand expressions, leaving the results in registers.

Then generate the appropriate ALU instruction to combine the operands and leave the answer in a register. Let's

look at example to see how all this works. Here have an assignment expression that requires a subtract, a multiply, and an addition to compute the required value.

Let's follow the compilation process from start to finish as we invoke compile_expr to generate the necessary code. Following the template for assignment expressions from the previous page, we recursively call compile_expr to compute value of the right-hand-side of the assignment. That's a multiply operation, so, following the Operations template, we need to compile the left-hand operand of the multiply.

That's a subtract operation, so, we call compile_expr again to compile the left-hand operand of the subtract. Aha, we know how to get the value of a variable into a register. So we generate a LD instruction to load the value of x into r1.

The process we're following is called "recursive descent".

We've used recursive calls to compile_expr to process each level of the expression tree.

At each recursive call the expressions get simpler, until we reach a variable or constant, where we can generate the appropriate instruction without descending further.

At this point we've reach a leaf of the expression tree and we're done with this branch of the recursion. Now we need to get the value of the right-hand operand of the subtract into a register. In case it's a small constant, so we generate a CMOVE instruction. Now that both operand values are in registers, we return to the subtract template and generate a SUB instruction to do the subtraction.

We now have the value for the left-hand operand of the multiply in r1.

We follow the same process for the right-hand operand of the multiply, recursively calling compile_expr to process each level of the expression until we reach a variable or constant.

Then we return up the expression tree, generating the appropriate instructions as we go, following the dictates of the appropriate template from the previous slide.

The generated code is shown on the left of the slide.

The recursive-descent technique makes short work of generating code for even the most complicated of expressions. There's even opportunity to find some simple optimizations by looking at adjacent instructions. For example, a CMOVE followed by an arithmetic operation can often be shorted to a single arithmetic instruction with the constant as its second operand. These local transformations are called "peephole optimizations" since we're only considering just one or two instructions at a time.