Introduction to Compilers and Translation Engineering

ECE 46800/57300/59500, Fall 2023

Step 3, Control Structures

last update: 31 August at 11:36pm

Table of contents

In this module, you will be asked to build a compiler that can translate a simplified version of C, that we call *microC* (or *uC* for short), into a variant of RiscV assembly. To do so, we will examine three steps of the compilation process:

1. **Building a parser.** You will learn how to use context-free grammars and regular expressions to specify a new language construct and keywords to add a new control structure (if-then-else) to our language.
2. **Generating code.** We will perform code generation for a version of RiscV that has an unlimited number of registers. In the first phase of code generation, you will learn to add *semantic actions* to your parser so it can build an *abstract syntax tree* (AST) for assignment statements and expressions, and then traverse that AST to generate code.
3. **Handling control structures.** We will then complete the code generation process for uC by adding support for generating code for control structures (while loops and if-then-else statements).

Step 3: Control Structures

Suggested date of completion: Sep 22nd 2023, 5PM ET

Due: Dec 1st 2023, 5 pm ET

Setting up your repository

Set up your GitHub classroom repository for step 3 using the link on Brightspace.

Set up your project development and testing environment as directed in the project environment documentations.

Then clone your repository as directed in the cloning and submitting instructions.

Background

We now have a compiler that can generate assembly for straight line code: a sequence of arithmetic operations, assignment statements, and input and output statements. But for a program to do anything useful, it cannot just be straight-line code. It needs to support *control flow*: the ability to change what code executes based on the dynamic execution of the program.

In this step, we will implement code generation for the two control flow constructs in uC: `if` statements and `while` loops.

Branches, Jumps, and Labels

The key to implementing control flow is being able to change the flow of execution of a program. Normally, the machine (or, in this case, our simulator) will execute a program by walking through the instructions one by one, in order. But there are several RISC-V instructions that can change this behavior. Consider the following code:

```
   geti r1
  geti r2
  blt r1, r2, l0
  puti r2
  j l1
l0:
  puti r1
l1:
```

This code implements, roughly, the following piece of uC:

```
 read(x);
read(y);
if (x < y) {
    print(x)
} else {
    print(y)
}
```

The way it does so is through the `blt` and `j` instructions. The `blt` instruction is a conditional branch: if `r1 < r2`, then the branch will jump to label `l0` in the code (and in doing so, will skip executing `puti r1`). The `j` instruction is an unconditional jump: it will immediately jump to label `l1` in the code (and in doing so, will skip executing `puti r2`).

> Risc-V provides 6 different conditional branch instructions for integers:
>
> 1. `blt` for "branch if less than"
> 2. `ble` for "branch if less than or equal to"
> 3. `bgt` for "branch if greater than"
> 4. `bge` for "branch if greater than or equal to"
> 5. `beq` for "branch if equal to"
> 6. `bne` for "branch if not equal to"
>
> We have provided Java classes to model all of those instructions, as well as the unconditional jump, `j`.
>
> Risc-V *does not* provide conditional branch instructions for floating point. Instead, it provides comparison operations that take the form:
>
> `<cmp> dst, src1, src2`
>
> with the semantics: `dst = (src1 <cmp> src2) ? 1 : 0`. This can be combined with integer branch instructions to branch based on floating point comparisons.
>
> The three comparison operators are:
>
> 1. `feq.s` for `==`
> 2. `flt.s` for `<`
> 3. `fle.s` for `<=`
>
> Note that you can synthesize the other three comparisons by changing the polarity of the branch.

Generating code for ifs and whiles

We already know how to generate code for a list of statements. Generating code for `if` statements and `while` loops involves stitching together the "bodies" of those constructs (the then/else blocks for if statements, and the loop body for while loops) with carefully chosen labels, branches, and jumps. See the lecture notes for an explanation of how this works.

What you need to do

We want you to fill in the rest of `CodeGenerator` and `MicroC.g4` to build an AST for if statements and while loops, then generate code for them.

The key challenges you will have to face in this step are:

1. Generating unique and appropriate labels for control constructs.
2. Generating code for the conditional expressions (`CondNode`) in the control constructs. You have to take care to generate the right kind of branch, and the right labels for those branches. The notes in the starter code give some explanation for how to do that.

> Note: don't forget to handle the case where an `if` statement does not have an `else` block!

Running your code

To run your code, you can use the RISC simulator we have provided for you. When using the container environment, this is present in ~/RiscSim. When using the ecegrid environment script, the path to the driver script for it is set in $RISCSIM.

Once your compiler has created an assembly file, you can run it like so:

- In container: `python3 ~/RiscSim/driver.py [assembly file]`
- On ecegrid: `python3 $RISCSIM [assembly file]`

Sample inputs and outputs

The test inputs we will use to test your program are in `tests` . We also have sample outputs in `outputs` . While your compiler may not produce exactly the same assembly as in `outputs` , the code you generate should produce the same results.

Additionally, we have also provided a testall script to automatically compare all test inputs and outputs for you. **NOTE: some tests require user input. The testall script selects a random numerical input between 0 and 100 for each run. it may miss edge cases - make sure you test manually as well!*

What you need to submit

- All of the necessary code for building your compiler.
- A Makefile with the following targets:
  1. `compiler` : this target will build your compiler
  2. `clean` : this target will remove any intermediate files that were created to build the compiler
- A shell script (this *must* be written in bash, which is located at `/bin/bash` on the ecegrid machines) called `runme` that runs your scanner. This script should take in two arguments: first, the input file to the compiler and second, the filename where you want to put the compiler's output. You can assume that we will have run `make clean; make compiler` before running this script, and that we will invoke the script from the root directory of your compiler.

While you may create as many other directories as you would like to organize your code or any intermediate products of the compilation process, both your `Makefile` and your `runme` script should be in the root directory of your repository.

Do not submit any binaries. Your git repo should only contain source files; no products of compilation.

See the submission instructions document for instructions on how to submit. You should tag your step 2 submission as `submission`

Most course announcements will be posted on Piazza and/or the corresponding webpage. Only a few critical announcements will be posted on main page.