Introduction to Compilers and Translation Engineering

ECE 46800/57300/59500, Fall 2023

Step 5, Register Allocation

last update: 31 August at 11:36pm

Table of contents

Step 5: Register Allocation

Suggested date of completion: Nov 3rd 2023, 5PM ET

Due: Dec 1st 2023, 5 pm ET

Setting up your repository

Set up your GitHub classroom repository for step 4 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

Up until now, we have been generating code using a simplifed process, and a simplified target.

1. **Simplified Process** We have generated assembly code directly from the abstract syntax tree
2. **Simplified Target** The instruction set we have been targeting has an unlimited number of registers (your instructions could use as many `t**` registers as you needed)

We are going to fix both of these shortcomings in this module:

1. We will add an *intermediate representation* to our compiler. This will be a form of three address code that encodes the information you need to generate assembly *without* locking you in to particular decisions about when to load/store from memory, or what types of instructions to use.
2. We will perform *register allocation*. In the process of translating from our intermediate representation to assembly, we will make sure that we only use the number of registers that 32-bit RISC-V naturally supports.

You may find the RISC-V documentation useful, especially the assembly documentation, on page 109.

Intermediate Representation

The primary background you need for your intermediate representation is the lecture notes on IRs and three address code.

While we do not prescribe exactly how you implement your three address code, we *suggest* the following.

The instruction classes that we define for you just use `String` s as their operands. So rather than directly generating assembly instructions in your code generator, you can generate *the same kinds of instruction objects* but with different operands to represent 3AC. If an operand starts with a `$` , it represents a 3AC operand rather than a "direct" operand (i.e., a register or a literal). Thus:

```
ADD t1, t2, t3
```

is an `ADD` performed with three registers as operands. In contrast,

```
ADD $t1, $t2, 17
```

is an `ADD` performed with the integer temporary variable (not register) `$t1` as a destination, the integer temporary variable `$t2` as one source operand, and the integer literal `17` as the other source operand.

In addition to having literals or temporaries as operands, 3AC versions of instructions can also have:

1. Floating point temporaries (prefixed with `$f`)
2. Global variable operands (prefixed with `$g` , followed by the name of the global variable)
3. Local variable operands (used for parameters and locals, prefixed with `$l` and the offset from the frame pointer)

Note that generating 3AC is substantially easier than generating assembly! You do not have to worry about l-values and r-values, as you can just put a variable name in any operand position.

> If you would like to add additional operations to your 3AC to make your code generation life easier, you should feel free to do so.

Converting 3AC to Assembly

You should convert 3AC to assembly at the function level. Once you have generated 3AC for all the code in a function, you can make a pass over that code to convert it to assembly. A simple *macro expansion* approach to code generation converts each kind of 3AC into one or more assembly instructions mechanically. For example:

`ADD $t1, $t2, $t3` would turn into: `ADD t1, t2, t3`

in assembly.

On the other hand, `ADD $l4, $t2, $t3` (add temporary 2 to temporary 3 and store the result in the local 4 up from the frame pointer) would convert into:

```
ADD t4, t2, t3
SW t4, 4(fp)
```

> Note how you can easily turn the 3AC operand for a local into an efficient store instruction!

More complicatedly, `ADD $gx, $gy, 7` might turn into:

```
LA t1, <address of y>
LW t2, 0(t1)
ADDI t3, t2, 7
LA t4, <address of x>
SW t3, 0(t4)
```

Note that the fact that we needed to load from `y` to perform the computation was implicit in the fact that it is a source operand. There was no need for special r-value or l-value handling when generating 3AC, it's all handled in the conversion to assembly.

> We also did a small optimization and used an ADDI instruction when one of the operands was a literal. You could have instead loaded `7` into another register.

Note that to perform macro expansion, we often need to use additional registers to hold things like addresses, loaded values, etc. So how do we know what registers to use for that? A simple strategy is to put all temporary variables in their correspondingly numbered register (so `$t7` goes to register `t7` , `$f2` goes to register `f2` , etc.), and then new register you need just count up starting from the temporaries you've already allocated.

Another strategy is to set aside a handful of registers that you will always use for holding intermediate results during macro expansion (how many do you need?).

> Note: you will never need to "remember" the names of these intermediate registers outside of the boundaries of any single 3AC operation, since the final result of any 3AC operation ends up in a local varaible, a global variable, or a named temporary.

> Don't forget to account for the registers you use when generating register save and restore code in your functions!

But a better thing to do, because it will let you generate "real" RISC-V assembly, is to do *register allocation*.

Register Allocation

The most useful background for register allocation is the course notes on this topic.

We will perform register allocation to:

1. Make sure that we do not use more than the available registers on a "real" 32-bit RISC-V machine
2. Minimize the number of loads and stores we need to perform by trying to keep the values of variables in registers when possible.

We will perform *local* register allocation in this step, not *global* register allocation. Thus, to register allocate the 3AC for a function, we will break the list of instructions up into basic blocks, then, for each basic block, compute liveness and perform register allocation.

Basic Blocks

The notes give an algorithm for breaking a sequence of 3AC instructions up into basic blocks.

Liveness

Once you have a list of instructions that constitutes a basic block, you can perform liveness by walking over those instructions in reverse order, using the algorithm given in the notes.

Note that we need to make some conservative assumptions about live variables at the end of each basic block. Because we have no information about the rest of the program (or even the rest of the function), you should assume that all local variables and global variables are live at the end of each basic block.

Register Allocation

Perform register allocation for each basic block by walking *forward* through the basic block and generating assembly code for each 3AC operation, as described in the notes. You will need to create a structure to track which registers hold which temporaries/variables, and which are dirty.

Don't forget to generate stores for each live, dirty register at the end of the basic block.

> The register allocation algorithm we present gives you freedom to choose how to allocate registers for operands, and which registers to spill when necessary. You are free to choose whatever heuristic you like for this purpose.

Which and How Many Registers

We are generating code for a variant of 32-bit RISC-V, with floating point. The RISC-V assembly manual lists which registers are available for what. Note, however, that our calling convention passes all arguments and return values on the stack so, while RISC-V reserves some registers for those purposes, you are free to use them. (Unless you optimized function code generation, in which case you will need to adjust this).

The registers you have available to you *by default* are:

1. Integer registers `x1` through `x31` with the execption of `x1` (the return address, also called `ra`), `x2` (the stack pointer, also called `sp`) and `x8` (the frame pointer, also called `fp`). Note that `x0` is hardwired to 0, so you cannot use it.
2. Floating point registers `f0` through `f31`.

However, you should provide a parameter that allows us to change the number of registers you have access to. (see "what you need to do"). You should a parameter that denotes the maximum integer and floating point register you have access to. In other words, if that parameter is `8`, the maximum available integer register will be `x7`, and the maximum available floating point register will be `f7`. In other words, the default parameter is `32`.

We will never set this parameter to less than 8.

What you need to do

We want you to add an intermediate representation and register allocation, as described above.

468/595 students do not need to perform liveness analysis (for the purposes of the register allocation algorithm, assume that the values in registers are always live, so you never free them before the end of the basic block. At the end of the basic block, you can assume/know that any registers holding temporaries are dead.)

573 students should perform liveness analysis during register allocation to optimize register usage by freeing any registers that hold dead values as soon as they become dead.

The `runme` script is modified to take a third argument: the parameter for the number of registers that will be used during register allocation, as described above. So the script should run as follows:

```
./runme <input file> <output file> <# of registers>
```

Grading

Register allocation has both a correctness aspect and an optimization aspect. On the one hand, your generated assembly should not use more registers than the machine provides (correctness). On the other hand, you need to utilize the provided registers as many as possible, so you won't generate more memory accesses than necessary (Optimization). In the grading of this step, we will evaluate both of these aspects according to the following rules:

For each one of the tests included in the test set:

1. Your generated assembly will be checked against the reference output on the simulator with 32 registers, to ensure that your code correctly behaves as compared to reference. If your code fails this check, you won't receive any credits for this particular test.
2. The number of memory access instructions (`LW, SW, FLW, FSW, ...`) **executed during the simulation run** (Note that this means the actual runtime count, not the count in the asm file) will be collected for both your assembly output (denoted as `S`) and the reference output (denoted `R`). Then, your credits for this particular test would be adjusted by:
   - **For 468/595 students**: `MIN(1, R/S) * 100%`
     - Full credits are granted if your memory performance is the same or better than the reference
     - Otherwise, you get 1% penalty on each 1% of extra memory accesses you perform.
   - **For 573 students**:

     ```
     if S >= R: R/S * 60%
     if S < R: 60% + MIN(25%, 1 - S/R) / 25% * 40%
     ```

     - The reference output and implementation are based on basic-block-only register allocation, without any liveness analysis. Thus we expect your implementation based on liveness to perform better in memory accesses.
     - You will get full credit if you beat the reference memory performance by more than 25%.

Credits for each test cases are distributed evenly. After adjustment of performance, the credits will be accumulated together to reflect your total grades. Then the late submission penalties, if any, would be applied to produce your final grades for the step.

For this step, due to the difficult nature of the assignment, we may curve the grades from the grader based on grades distribution after all grading has been concluded. The curved grades will be uploaded to BrightSpace after the decision is made and grades are calculated accordingly.

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]`

You can also pass an optional second argument that specifies the number of registers to configure the machine with:

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

We would add the functionality to collect the number of memory access instructions executed to the simulator later. Once that is done and the simulator is updated in your environment, you could run the simulator with:

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

to get the number of memory accesses.

We will post a announcement to Piazza once this update is done and live.

Sample inputs and outputs

The inputs and outputs we will test your program on can be found in the `tests` and `outputs` directories.

Note that our output code may look quite a bit different than yours if you make different decisions about register allocation.

For 573 students: the reference register allocations are done with basic-block-based algorithm, and no liveness analysis is performed. You need to reason about the liveness analysis part on your own.

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 three arguments: first, the input file to the compiler; second, the filename where you want to put the compiler's output; and third, the number of registers you will use for register allocation (see above). 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 5 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.