IIT CS443: Compiler Construction

Project 6: Register Allocation and Instruction Selection

Prof. Stefan Muller

Out: Tuesday, Nov. 12
Due: Wedneday, Nov. 27, 11:59pm CDT

Total: 55 points

Logistics

Submission Instructions

Please read and follow these instructions carefully.

- The starter code for Project 6 has been distributed through a pull request to your Project 2 GitHub repo. Merge this pull request into your main branch to start working. (This shouldn't cause merge conflicts, but if it does and you aren't sure how to handle them, ask.)
- When you want to submit, commit the latest changes to your GitHub repo. (A helpful commit message is always, well, helpful, but I'll assume the last commit to your repo is your intended project 6 grade unless you tell me otherwise, since there are no more projects for you to work on.)
- Compile (by running `make`) before submitting. **Submissions that don't compile will not get credit.**
- **A note on late days:** You can submit by Friday, Nov. 29 using **one** late day (not a typo; this is skipping over Thanksgiving Day), or by Saturday Nov. 30 using **two** late days. Because of the timing of the final exam, barring truly exceptional circumstances, I cannot accept any submissions after 11:59pm Saturday Nov. 30.

Collaboration and Academic Honesty

You may work in groups of at most 2 on this project. Read the policy on the website and be sure you understand it.

1 Module Structure and Helpful Functions

The (not very abstract) abstract syntax for RISC-V instructions is in `riscv/riscv_ast.ml`. The main type is `'a inst`. The `'a` type parameter is the type of labels: for your code generation, we will only be working with `label inst`, where `label` is an alias for `string`. That is, as in class (and as in LLVM), you'll be able to jump and branch to string labels which you can place directly in the code (the `Label` "instruction"). Otherwise, instructions are grouped by their type (the ones you'll primarily be working with are R, I, and B). This means that the actual opcode is one of the parameters, **not** the constructor. For example, you'd encode `add rd, rs1, rs2` as `R (Add, rd, rs1, rs2)`. Note:

- `lw` is considered an I-type instruction, with `lw rd, im(rs)` encoded as `I (Lw, rd, rs, im)`.

- `jal`, `sw`, and `lui` are treated separately as indicated in the comments.
- You should not need to use `StoreLabel`, `LoadLabel`, or `LoadAddress` in your code.

Registers are also defined as constructors of type `reg`. The constructors are named `X0`, `X1`, etc. There are also aliases for registers used in various conventions, e.g., `zero`, `ra` (return address), `sp` (stack pointer), `fp` (frame pointer), and `retval` (a0, return value). There are also pre-defined lists of registers that are `callee_saved` and `caller_saved`. The list `general_purpose` contains all of the registers you should consider available for register allocation. Finally, `args` contains the registers (in order) that are used for function arguments. You can access the RISC-V AST using `Riscv.Ast` (or `R` in `codegen.ml`).

The file `riscv_print.ml` (`Riscv.Print`) contains functions for converting various AST components to strings, and for printing instructions. These work much the same way as pretty-printing functions on past assignments. Note that I've included two versions of `string_of_reg`. The default one prints the ABI names, but you can comment that one out and uncomment the other, which prints the numbers.

The file `codegen.ml` already defines a module `VRMap`, which meets the same interface as the maps we know and love, but its keys are of type `var_or_reg` (remember from lecture, this is now the type of variables, which allows us to mix LLVM code that uses variables with code that uses explicit registers).

All of the infrastructure from past projects is still here as well (as well as your solutions to them, unless you replace them with my solutions). In particular, remember the following definitions, which you might find useful:

- `LLVM.Ast.sizeof: LLVM.Ast.typedefs -> LLVM.Ast.typ -> int`
- `Config.word_size: int`

2 Programming Task 1: Instruction Selection (25 points)

Your first task is to complete the translation from LLVM (after register allocation) to RISC-V. I've already implemented a few cases; you need to do the rest, namely:

- `ILabel`
- `ISet`
- `IBinop`
- `IBr`
- `ICondBr`
- `IAlloca`
- `ILoad`
- `IStore`

(Note that these don't necessarily directly correspond to `match` cases you need to implement, as some of these are multiple cases and some can be combined).

This all happens in `codegen_body` in `codegen.ml`. As you can see, there are already a few helper functions here:

- `get_reg_r: bool -> var_or_reg -> R.label R.inst list * R.reg`. This takes a `var_or_reg` and sets you up to use it as an operand. The first component of the result is a list of RISC-V instructions to move the variable into a register (if it isn't in one already), and the second component is the register that the variable will be in once you run those RISC-V instructions. If the variable is already in a register, the list of instructions will be empty and the register will be that register. This is obviously also true if the `var_or_reg` is just a `Register`. If it has to move the variable into a register, it will use `x5` if the first argument is true, otherwise `x6`.

- `get_reg_r_val: bool -> avalue -> R.label R.inst list * R.reg`. Similar to `get_reg_r`, but takes a value (a variable or integer constant). This also handles loading constants into a register, including using `lui` if the constant takes more than 12 bits. Note that this is wasteful if you could use an instruction like `addi`, as `get_reg_r_val` will always move the constant into a register.

- `get_reg_w: var_or_reg -> R.label R.inst list * R.reg`. Gives you the register to use for the *destination* of an instruction. If the `var_or_reg` is a variable stored in a register, it will return that register. If it's spilled, it will return the temporary register `x7` and a list of RISC-V instructions that will store `x7` to the correct location on the stack to update the variable. NOTE that these instructions should be run *after* performing whatever operation you're doing.

- `rop_of_binop: bop -> R.rop`. Returns the RISC-V opcode for the R-type instruction corresponding to an LLVM binop.

- `iop_of_binop: bop -> R.iop`. Returns the RISC-V opcode for the I-type instruction corresponding to an LLVM binop.

The main part of the function matches against the first few LLVM instructions. It should return something of type `R.label R.inst list * ainst list`. That is, a list of RISC-V instructions corresponding to the matched instructions, paired with the rest of the instructions. For example, if `insts` is
`(IBinop(Register R.X7, BAdd, ty, Var (Register R.X5), Var (Register R.X6)))::t`
you'd return
`([R.R (R.Add, R.X7, R.X5, R.X6)], t)`
i.e., a RISC-V instruction corresponding to the first LLVM instruction and the remaining LLVM instructions. If you want to emit a RISC-V instruction corresponding to the first two LLVM instructions (depending on what they are), you'd match `insts` with
`<pattern to match inst 1>::<pattern to match inst 2>::t` and then return
`([<RISC-V instruction(s) for inst1 and inst2>], t)`

You also have access to the following arguments to `codegen_body`, which may be useful:

- `ctx: typdefs` — Typing information about the LLVM code. This can be passed to `LLVM.Ast.sizeof`.

- `alloc: alloc_res VRMap.t` — The result of register allocation.

Finally, your code can/should make the following assumptions (my code does; your code shouldn't have to interact with these assumptions much if at all, but just so you're aware):

- All integer types in the LLVM code you're given are 32 bits or smaller (i.e., you may see `i1` or `i8` or `i32` or even `i5` but not `i64`). And, as has been true so far, pointers are 32 bits.

- Integer types `iN` for $N < 32$ are stored in the lower N bits of a 32-bit memory word or register and sign-extended. For example, -1 as an `i8` would still be stored as `0xffffffff`. Don't try to be cute packing multiple `i8`s together in a single word.

3 Testing Instruction Selection: This is different from past projects!

You can test instruction selection even before implementing register allocation, because I've provided you with a simple (almost-) linear scan register allocator that should work fine for simple test cases (it should be correct for any test case, but may spill more than necessary).

Compile your code using `make` (in the top level of the source tree). This will produce the binary `main`, which you can use as follows to compile test programs:

`./main -O0 <path_to_test_case>`

The test case can be a MiniLLVM, MiniIITRAN, MiniC, or MiniML source file (if it's something other than MiniLLVM, it will be using your solution(s) for the respective projects to comple to LLVM). If you want to optimize also, remove the `-O0` flag and it'll run your project 5 optimizer. In any case, this will parse and type check the file, compile it to LLVM IR (if it isn't already), allocate registers using the provided implementation, and then run instruction selection.

This will output RISC-V assembly code to `<same_path_as_input>/<file>.s`. You can run this using the Venus RISC-V simulator at `https://venus.cs61c.org/`, as demonstrated in class (or other RISC-V simulators/emulators at your own... uh... risk). Remember that the return value of the `main` function becomes the exit code of the whole program—Venus reports this with a banner saying "Program exited with error code N." This isn't actually an error (as long as N is the return value you were expecting).

VERY Important note: The test script run by `make test` **does not** run your generated RISC-V code through an emulator (because I haven't provided you with one that can be run programmatically), and is therefore totally useless for testing this project. The only way to test your code is using the instructions above. It's just as well though. Venus has ***way*** better debugging support than anything I would have coded up, and you can use this to find errors in your code generator.

I did, however give you a few test cases in the `proj6_tests` folder that you can use with the instructions above. The expected results are stored in `proj6_tests/results.txt` (because my LLVM parser doesn't allow comments).

4 Programming Task 2: Register Allocation (30 points)

Your second task is to fill in the implementation of `grcolor` in `codegen.ml` with a graph-coloring register allocator, as we discussed in class. The function returns a pair (list of spilled vertices, allocation), where the second component is of type `alloc_res VRMap.t`: it maps registers to themselves, and variables to either `InReg` of a register or `OnStack` of a stack position. The parts that are already implemented perform a liveness analysis and build an interference graph `igraph` from it. I also gave you the part that initializes the allocation map to pre-color the registers with themselves. The interference graph is of type `IG.t`, and `IG` meets the `GRAPH` signature in `utils/graph.ml`. The main interesting feature of this interface is that nodes in the graph contain data. The data of a node in the interference graph is of type `var_or_reg`: remember that the nodes of an interference graph represent variables (which in our LLVM AST can now be actual variables or registers). To get the `var_or_reg` that a `node` represents, use `IG.get_data`. Also note that the `GRAPH` interface is designed for directed graphs; since interference graphs are undirected, all nodes that have an edge between them have one in both directions. Practically what this means is that, to get the neighbors of a node, you can call either `IG.succs` or `IG.preds` and get the same answer.

You can see how some of this works in action by looking at the code for `greedy`, which is the provided greedy register allocator.

While you have a lot of flexibility in your implementation, the following slight variant on the algorithm we discussed in class is pretty easy to implement functionally given the way things are set up (note that below, "the stack" refers to the stack of graph nodes we build up during simplification; how you implement it is up to you):

1. Iterate over the nodes of the interference graph, removing from the graph and pushing on the stack any node with degree less than K (remember not to remove nodes that are `Register`s)
2. Repeat step 1 until it doesn't remove any more nodes.
3. Spill a node, remove it from the graph and push it on the stack.
4. Repeat steps 1–3 until the graph contains only `Register`s.
5. In the order that they appear on the stack (the reverse of the order they were pushed), pick a color for the nodes. You can use the function
   `get_reg: IG.t -> alloc_res VRMap.t -> IG.node -> R.reg option`
   which takes the interference graph, the allocations so far, and a node, and returns a register that is available to use for the node, if one exists.

Note in particular that you **do not** have to implement coalescing.

When done, change the definition of `regalloc_strategy` in `codegen.ml` from `greedy` to `grcolor` in order to use your allocator.

5 Testing Register Allocation

First, make sure you've changed `regalloc_strategy` to use your allocator (see above) before testing, otherwise you're just testing my greedy allocator!

Compile and run your code as in the "Testing Instruction Selection" section. Run the compiled code in a simulator to make sure your allocator is working correctly. To make sure it's also doing something sensible, you'll need to check the actual allocation decisions it's making. If you don't want to inspect the compiled assembly manually, you can use the `-dumpalloc` command line flag to `./main`, which will print out a table with the allocation decisions after running register allocation.

6 The End

I recommend running `./main` on one of the MiniML test cases from Project 4, and running the generated assembly through the Venus simulator. This is a good stress test for your compiler, but also involves the entire compiler pipeline from Projects 2–6. Depending on which test case you run, it may take a while—the simulator is kinda slow and I'm not gonna say we built the most efficient compiler. But as the program runs, you'll have some time to take a minute and reflect on what you've accomplished this semester: you built a working compiler from (OK, a pretty small subset of) OCaml to assembly. Not bad.

If you find yourself reflecting for too long, you may have introduced an infinite loop into the code. But hopefully (maybe after some debugging) you'll see it spit out the right answer.

See you at the final!