Prof. Jingke Li (FAB120-06, lij@pdx.edu), Class: TR 10:00-11:50 @ FAB 150; Office Hour: TR 13:00-14:00.

Assignment 4: X86-64 Code Generation

(Due Thursday 6/7/18 — Firm deadline!)

In this final assignment, you are going to implement a code generator for converting IR1 programs to X86-64 assembly code. This assignment carries a total of 10 points. There is also an extra credit part, which carries an additional 3 points.

Download and unzip the file `assign4.zip`. You'll see a `assign4` directory with the following items:

`Ir1Grammar.txt` — IR1 grammar

`ir1/` — contains the IR1 representation (`Ir1.java`) and a parser (`Ir1Parser.java`)

`target/X86.java` — X86-64-related representations and utility routines

`tst1/` — contains a set of tests

`gen,run` — scripts for running your `CodeGen` program, and the output `.s` programs

1. Overview

For this assignment, you are implementing a *naive* code generator, *i.e.* one that does not use register allocation at all. As discussed in class, the idea behind a naive generator is very simple:

- Every parameter, variable, and temp is allocated a memory storage in a stack frame.
- Registers are used only as temporary storage to support instruction's execution, and for passing parameters from caller to callee, as required by X86-64's ABI.
- Most instructions are translated into an assembly code block with three parts:
  1. Loading operand(s) from their memory storage to scratch register(s)
  2. Performing the instruction's operation
  3. Storing the result to the destination's memory storage

2. Stack Frame and Variable Storage Mapping

Since no values are held in registers over function calls, there is no need to perform register saving and restoring. The only purpose of a stack frame is to store parameters, variables, and temps of a function.

We assume all values are 4 bytes.[1] Therefore we can use the following formula to compute frame size:

```
frameSize = (paramCnt + varCnt + tempCnt) * 4
```

There is only one small problem. While `paramCnt` and `varCnt` can be directly obtained from the `params` and `locals` components of an `IR1.Func` node, to get a precise `tempCnt`, the generator has to traverse the instruction list of the function to look for all temp occurrences. A simpler solution exists. We can use the instruction count as an approximation for `tempCnt`. This is a safe approximation, since a single instruction can write to at most one temp. (This approach may waste some memory; we ignore that.) With this approach we can modify the formula to:

[1] This is a simplification assumption, and is not totally correct, since IR1 addresses are 8 bytes. However, for our small test programs, the addresses all happen to fit in 4 bytes.

```
frameSize = (paramCnt + varCnt + instCnt) * 4
```

The X86-64 ABI requires that the end address of a stack frame be a multiple of 16, which means that when control is transferred to a new function's entry point, `%rsp + 8` is also a multiple of 16 (since a return address has been pushed on to the stack after the end of the caller's frame). Therefore, you need to include the following statement in the code generator to adjust the frame size when necessary:

```
if ((frameSize % 16) == 0)
  frameSize += 8;
```

Note that `frameSize` defined here does not include the return-address slot.

Since parameters, variables, and temps are all stored in a stack frame, the code generator needs a simple mapping function from names to the frame offsets. A `HashMap<String,Integer>` can be created for each function for this purpose. Parameters need to be added to this map at the beginning of a `Ir1.Func` node's codegen routine, so that function arguments' values can be stored into their frame slots before the function starts its execution. Variables and temps can be added as they appear in the instructions of the function body. (We assume the IR program has been type-checked, so that variables and temps are always defined before their first use.)

3. The CodeGen Program

A starter version of the codegen program is provided in `CodeGen0.java` in the `ir1` sub-directory. The program is organized in a standard syntax-directed form, *i.e.*, a set of codegen routines, one for each IR1 node. At the top, the `main()` method reads in an IR1 program, and invokes the `gen()` routine on the top-level `Ir1.Program` node, which, in turn, calls the `gen()` routine on its components, and so on.

Here are some highlights of codegen issues regarding individual IR1 nodes. In the `CodeGen0.java` program file, you'll find more detailed codegen guidelines.

- `Ir1.Func` — Function is the main codegen unit. For a function node, the generator computes the function's frame size and prepares a `HashMap` data structure to map parameters, variables, and temps to frame offsets. (These two items are packaged into a single `Env` object to be easily passed to all codegen routines.) It will generates code for allocating a frame on the stack and to copy the parameters' values from argument registers to their frame locations. Finally, it invokes the `gen()` routine on the individual instructions to generate their target code.
- `Ir1.Binop` — The code generator separates arithmetic operations from relational operations. For arithmetic operations, corresponding X86-64 instructions exist. However, the division operation needs to be singled out, since it requires two specific registers (`RAX+RDX`). For relational operations, the corresponding X86-64 code should consist of three instructions:

  ```
  cmp     # compare
  set     # set flag
  movzbl  # expand single-byte result to long (for memory storage)
  ```

  Also remember that left and right operands are switched for the `cmp` instruction under Linux/Gnu assembler.
- `Ir1.Call` — The code generator separates out the calls to `printInt` and `printStr` from others. For these two print functions, it generates a call to the system routine `printf`. Since `printf` is a variable argument function, the codegen adds the following instruction before the `call` instruction (This is to communicate the number of arguments to `printf`.):

  ```
  xorl %eax, %eax
  ```

  Also, for `printInt`, the codegen passes a predefined control string to `printf` as the first argument (Hence the argument of `printInt` will become the second argument, and be passed through register `RSI`.):

  ```
  leaq .S0(%rip), %rdi
  ```

For all calls, the code generator generates code for loading arguments from their storage locations into the designated argument registers, *i.e.* `RDI, RSI, RDX, RCX, R8,` and `R9`. If there are more than 6 arguments in the IR program, the code generator just quits.

- `Ir1.Return` — For a return node, the code generator generates an instruction for popping off the function's frame and an `ret` instruction. If there is a return value, it also generates code for loading the value to the return-value register `RAX`.
- `Ir1.StrLit` — For a string literal node, the code generator does three things. (1) Assign it an unique label; (2) Wraps the label in an `X86.AddrName` and generates a `LEA` instruction to load it to a register; and (3) Adds the string literal to a global list for later dumping. A simple strategy for finding an unique label for a string literal is to use its index in the list. For example, if the first two strings in the list are `"%d"`[2] and `"Hello"`, then their labels can be created as `.S0` and `.S1`.

  In the `gen` routine for `Ir1.Program` node, after program's code being generated, the code generator dumps out all the string literals in a `.data` section:

```
        .data
.S0:    .asciz "%d\n"
.S1:    .asciz "Hello\n"
.S2:    ...
```

  Note that a `"\n"` is added to each of the string literals.
- For other operand nodes (*e.g.* `Ir1.Id`, `Ir1.Temp`, `Ir1.IntLit`, and `Ir1.BoolLit`, the codegen routine generates code to load the operand into a register.
- For instruction nodes with a `dst` component (*e.g.* `Ir1.Binop`, `Ir1.Unop`, `Ir1.Move`, and `Ir1.Load`), the code generator checks to see if `dst` (which is either a var or a temp) is in the params/vars/temps `HashMap`; if not, it adds it in.

4. X86-64-Related Utilities

The program file `X86.java` contains x86-64-related utilities: register and operand representations and code emission routines. The code emission routines are just formatted versions of `System.out.print`. They are defined for different instruction cases. Study this program carefully, and use as many utilities provided here as you see fit to help implementing your code generator.

5. Your Task

Your task is to complete the implementation of the code generator. Call your program `CodeGen.java`.

You should test your codegen program with the provided `gen` and `run` scripts and the tests in `tst1`:

```
linux> ./gen tst1/test*.ir
linux> ./run tst1/test*.s
```

There is no requirement that your `.s` programs match the reference version in `.s.ref` files. However, the execution output from your `.s` programs do need to match the reference output in `.out` files.

Extra Credit Work If you can improve the baseline code generator in anyway, you may earn up to 3 extra points. Here are some ideas:

- *Precise temp count* — Instead of using instruction count as an approximation, compute a precise temp count for each function.

[2] This control string for `printf` needs to be added to the global string list by the code generator.

- *Literal operands* — In the current generator, operands are always loaded into registers. A small optimization is to check for integer and boolean literals, and use them directly in instructions when allowed.
- *Register Tracking* — After load values in registers, keep them there; spill them back to memory only when necessary (*e.g.* when there is no more registers available). This work requires tracking register usage and tracking variable locations. Note that you don't have to have variable liveness information for this optimization to work (as discussed in class). Without liveness information, variables may stay in registers longer than necessary. This optimization is much more challenging than the previous two optimizations. You should do this one only if you have extra time.

If you choose to do any extra work, please clearly indicate in your program's comment block what you have done.

Submission Submit a single file, `CodeGen.java`, through the "Dropbox" on the D2L class website. Keep your original file untouched in case there is a need to show its timestamp.