Prof. Jingke Li (FAB120-06, lij@pdx.edu), Tue & Thu 12:00-13:15 @ ASRC 230, Lab: Fri 10:00-11:15/11:30-12:45 @ FAB 170

# Assignment 4: X86-64 Code Generation

(Due Thursday 3/12/15 @ 11:59pm – 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 `hw4.zip`. You'll see a `hw4` directory with the following items:

- `hw4.pdf` — this document
- `CodeGen0.java` — a starter version of the code generator program
- `X86.java` — utility support library for generating X86-64 code
- `IR1Grammar.txt` — the IR1 language's grammar
- `lib.c` — contains the pre-defined functions used in IR programs
- `ir/` — contains the IR1 definition file and an IR1 parser
- `tst/` — contains a set of test programs
- `Makefile` — for building the code-gen
- `gen, run` — scripts for generating and running assembly programs

## 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:

- 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. Registers' usages are not traced.
- Every parameter, variable, and temp has a memory storage in a stack frame.
- Each IR operation instruction is translated into an assembly code block with three parts:
  - Loading operands from their memory storage to registers.
  - Performing the instruction's operation.
  - Storing the result to the destination's memory storage.

  At the beginning of the code block, all registers are available; and at the end, all are released.

Two related issues need to be discussed further.

**Stack Frame Size** Since there are no values held over in registers, there is no need to perform register savings at function call points. The only need for a stack frame is to store parameters, variables, and temps of a function. Therefore we can use the following formula to compute frame size (*Note:* In IR1 all stored values are integers):

```
frameSize = (paramCount + varCount + tempCount) * intSize
```

There is only one small problem. While `paramCount` and `varCount` can be directly obtained from the `params` and `locals` components of an `IR1.Func` node, to get a precise `tempCount`, the generator has to traverse the instruction list of the function to check for all temp occurrences. A simpler solution exists. We can use the instruction count as an approximation for `tempCount`. This is a safe approximation, since

each 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:

```
frameSize = (paramCount + varCount + instCount) * intSize
```

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.

**Variable Storage Mapping** Every reference to a parameter, a variable, or a temp in an IR program is linked to a load or a store from/to its memory storage in the target program. The code generator needs a simple mapping function from names to stack offsets. The solution is to collect all parameters, variables, and temps into a single `ArrayList`, and map their indices to stack offsets with formula: `offset = idx * intSize`. (For example, variable with index 2 will map to offset 8; hence its memory storage will be `8[%rsp]`.) Parameters and variables can be added to the list using the `params` and `locals` components, while temps can be added incrementally when they are encountered in instructions.

## The CodeGen Program Structure

A starter version of the code-gen program is provided in `CodeGen0.java`. The program is organized in a standard syntax-directed form, *i.e.*, a set of code-gen routines, one for each IR1 syntax node. At the top, the `main()` method handles the input of an IR1 program, and invokes the code-gen routine, `gen()`, on the top-level `IR1.Program` node. It, in turn, calls the `gen()` routine on each of the `IR1.Func` nodes in the program.

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

- `IR1.Func` — Function is the main code-gen unit. For a function node, the generator creates a list, `allVars`, and initializes it with its `params` and `locals`. It computes the function's frame size and generate code to allocate a new frame on the stack. It also generates code to move the incoming actual arguments from argument registers to their frame locations. Finally, it generates code for the function's instructions by recursively invoking the `gen()` routine on their corresponding nodes.
- `IR1.Binop` — The code generator needs to separate 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 under Linux/Gnu assembler.
- `IR1.Call` — To prepare for a call, the code generator needs to generate 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 needs to generate an instruction for popping off the function's frame, and then the `ret` instruction. It may also need to generate code for loading return value to the return-value register `RAX`.

- For IR1 nodes with a `dst` component (*e.g.* `IR1.Binop`, `IR1.Unop`, `IR1.Move`, and `IR1.Load`), if the `dst`'s name is not in the `allVars` list, add it in.
- For `IR1.Src` nodes (*e.g.* `IR1.Id`, `IR1.Temp`, `IR1.IntLit`, `IR1.BoolLit`, and `IR1.StrLit`), the code-gen routine is called `to_reg()`, indicating that it generates code for loading the source into a register.

## X86-64 Utility Support Library

The program file `X86.java` contains an x86-64 utility support library, which has two main parts: register and operand representations and code emission routines. Study this program carefully, since you'll be using many of the representations and the utility routines defined here.

One set of useful definitions is the classified registers:

```
// Indices of standard argument registers
static Reg[] allRegs = {RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP,
                        R8,  R9,  R10, R11, R12, R13, R14, R15};
static Reg[] argRegs = {RDI, RSI, RDX, RCX, R8, R9};
static Reg[] calleeSaveRegs = {RBX, RBP, R12, R13, R14, R15};
static Reg[] callerSaveRegs = {RAX, RCX, RDX, RSI, RDI, R8, R9, R10, R11};
```

The code emission routines are just formatted versions of `System.out.print()`. They are defined for different instruction cases. You should select the proper one to use for each individual case. These routines' type signatures are listed below.

```
static void emit(String s) {...}
static void emit0(String op) {...}
static void emit1(String op, Operand rand1) {...}
static void emit2(String op, Operand rand1, Operand rand2) {...}
static void emitLabel(Label lab) {...}
static void emitString(String s) {...}
```

## Your Task (*10 points*)

Your task is to complete the implementation of the code generator. Copy `CodeGen0.java` to `CodeGen.java` and edit the new program. Read the provided guidelines carefully. Another useful information source are the `.s` programs in the `tst` directory. In those programs you can see instruction-by-instruction corresponding examples of IR1 code and X86-64 code.

## Extra Credit Work (*Up to 3 points*)

The extra credit work is to improve this code generator in anyway you can. Here are a few ideas.

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

## Requirements, Grading, and What to Turn In

This assignment will be graded mostly on the correctness of the generated X86-64 code. The provided `.s.ref` files are for reference only. Your code generator does not need to match them. However, when running your generated `.s` programs with the `run` script, their output should match those in the `.out.ref` files.

The minimum requirement for receiving a non-F grade is that your `CodeGen.java` program generates at least one correct X86-64 program for an IR1 input.

Submit your program, `CodeGen.java`, through the "Dropbox" on the D2L class website.