Prof. Jingke Li (FAB120-06, lij@pdx.edu), Tue 10:00-11:50 @UTS 203, Labs: Tue 12:00-13:50 & Wed 12:00-13:50 @FAB 88-10

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 100 points. There is also an extra credit part, which carries an additional 30 points.

Preparation

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
- `RegAlloc.java` — a linear-scan register allocator
- `Liveness.java` — provides liveness information for the register allocator to use
- `IR1Grammar.txt` — the IR1 language's grammar
- `x86-64.pdf` — a brief summary of the x86-64 architecture and instruction set
- `lib.c` — contains the pre-defined functions used in IR programs
- `ir1/` — contains the IR1 definition file `IR1.java` and files of an IR1 parser
- `tst/` — contains a set of test programs
- `Makefile` — for building the code-gen
- `gen, run` — scripts for generating and running IR programs

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 function declaration 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 code generator calls the register allocator (`RegAlloc.linearScan()`) to get a register assignment for *all* variables and temps in the function. It computes the function's frame size and generate code to allocate a new frame on the stack. It also generates code to save any callee-save registers that get assigned by the register allocator. 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 treated specially, since it requires two specific registers (`RAX+RDX`). For relational operations, the corresponding X86-64 code should consists of three instructions:

```
cmp     # compare
set     # set flag
movzbq  # expand single-byte result to full size
```

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 moving arguments from their assigned registers into the designated argument registers, *i.e.* `RDI, RSI, RDX, RCX, R8,` and `R9`. In the X86-64 utility support library, there is a parallel move routine, which can be used for this purpose. If there are more than 6 arguments in the IR1 program, the code generator just quits.
- `IR1.Return` — For a return node, the code generator needs to generate the function's *exit sequence*, *i.e.* instructions for restoring the saved registers and for popping off the function's frame, before the final `ret` instruction.
- For IR1 nodes with a `.dst` component (*e.g.* `IR1.Binop`, `IR1.Unop`, `IR1.Move`, and `IR1.Load`), if the corresponding variable or temp does not have an assigned register, then it means the register allocator decided that the variable or temp is dead. In this case, no target code needs to be generated for this node.
- For `IR1.Src` nodes (*e.g.* `IR1.Id`, `IR1.Temp`, `IR1.IntLit`, `IR1.BoolLit`, and `IR1.StrLit`), the code-gen routine (`gen_source()`) takes an additional parameter: a scratch register. For `IR1.Id` and `IR1.Temp`, the routine returns their assigned register. For other nodes, their values are loaded into the scratch register, and the routine returns that register.
- There are two registers that are set aside as scratch registers: `R10` and `R11`.

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 emitGLabel(GLabel lab) {...}
static void emitString(String s) {...}
static void emitMov(Size size, Operand from, Operand to) {...}
```

Register Allocation

Register allocation is performed at function level. The code generator calls the allocation routine `linearScan()` (defined in `RegAlloc.java`) with an `IR1.Func` node as argument. The allocator returns a register mapping

for all variables and temps in the function (except for those that are determined to be dead). The mapping is represented by a Java HashMap:

```
Map<IR1.Dest,X86.Reg> regMap;
```

Note that `IR1.Dest` is an interface, which is implemented by two concrete nodes: `IR1.Id` and `IR1.Temp`.

If the register allocator cannot find enough registers to cover all variables and temps, it will raise an exception and quit. Your code generator does not need to continue in this case.

Stack Frames

Since all variables and temps are mapped to registers, the only use of a stack frame is to keep the return address and to save the callee-save registers that are assigned to temps or variables.

Note that 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 calculate a function's frame allocation size as follows:

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

where `calleeSaveSize` represents the total size (in bytes) of all callee-registers that need to be saved. Note that `frameSize` defined here does not include the return-address slot.

Your Task (*100 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 (*30 points*)

(The extra points you earn from this part can be used to offset any points you lost in this or previous assignments.)

Implement a second code generator. The only requirement for this code generator is that it *does not* use the register allocator provided in `RegAlloc.java`.

What this means is that you need to implement either the *naive* code generation method, which use registers only as scratch storage, or the method that performs register allocation along with code generation. Both methods have been discussed in class. The second method is more challenging, since it requires tracking register usage and variable locations. You should choose this one only if you have extra time.

With the naive method, there is one new issue. All temporary results need to be stored into memory right after they are computed, and be fetched back to registers when they are needed. Your generator needs to allocate space in the function's stack frame for these temporaries. You need to come up with a size for allocation and a mapping scheme for assigning storage index to each variable and temp.

To do this precisely, you need to know the total number of variables and temps in a function. While this count can be obtained through a traversal over the function's instructions, you can use a simple conservative approximation: just use the total instruction count, since each instruction can write to at most one variable or temp.

Requirements, Grading, and What to Turn In

This assignment will be graded mostly on the correctness of the generated X86-64 code. The provided test outputs (*i.e.* the `.s` programs) are for reference only. Your code generator does not need to match those outputs.

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(s), `CodeGen.java` (and `CodeGen2.java`) if you have one), through the "Dropbox" on the D2L class website.