Prof. Jingke Li (FAB 120-06, li@cs.pdx.edu), Classes: MW 16:40-18:30, URBAN 204, Office Hours: MW 14:00-15:00.

Project 5: SPARC Assembly Code Generation

(Due Wednesday 3/14/12 *** Firm Deadline ***)

This last project is to implement the final component of the MINI compiler, a SPARC assembly code generator for the IR Tree language. Like the interpreter, this codegen module is to be implemented as a visitor to the IR nodes.

The CodeVI Interface

The visitor interface for this project is called `CodeVI`, in which the visit routines for expression nodes return an `Operand` object (see below), while all other routines do not have anything to return.

```
public interface CodeVI {
  public void visit(PROG t) throws Exception;
  ...
  public void visit(MOVE t) throws Exception;
  ...
  public Operand visit(EXPlist t) throws Exception;
  public Operand visit(MEM t) throws Exception;
  ...
}
```

Operand Representation

In the codegen, all operands, including constants, addresses, and intermediate results, are stored in objects of the class `Operand`. There are three operand forms: registers, register-offsets, and immediate values, each is represented by a subclass. For immediate values, small 13-bit values are specially marked, since they can be included directly in many instructions.

```
abstract class Operand {}

public class Reg extends Operand {                  // register
  static final int FREE=0, USE=1, TEMP=2, RSRV=3;
  String name;
  int status;
}
public class RegOff extends Operand {               // register-offset
  Reg r;           // register
  int o;           // offset
}
public class Immed extends Operand {                // immediate value
  int i;
  Boolean is13b;   // flag for 13-bit value
}
```

Register Management

A simple register allocator is provided in `Sparc.java`. The register allocator keeps track of every register's status. Upon an allocation request, the allocator allocates the next `FREE` register, and changes its status to

USE. If there is no more registers available, it simply gives up and throws an exception. When a temp is assigned to a register, it should stay there until the end of the FUNC scope. The status TEMP is used for this case to prevent the register from being freed. **Note:** It is your codegen's responsibility to change a register's status from USE to TEMP, since the register allocator cannot know what a selected register is to be used for. Registers that cannot be allocated for general use (such as sp, fp, and g5-g7) are marked with the status RSRV. They are not included in the allocation pool. The following are the available register routines:

`Reg getReg()` — request an (arbitrary) free reg

`void getReg(Reg r)` — request a specific reg

`void freeReg(Reg r)` — free a used reg, skip if it holds a temp

`void freeAllRegs()` — free all registers, including those holding temps

The set of special registers are given symbolic names, which can be handy when you need to request them or just refer to them:

```
regSP, regFP, regG0, regI7, regY , regO0, regO1, regI0
```

Instruction Emission

A set of instruction emission routines are also provided in `Sparc.java`:

`emit0(String op)` — for emitting an instruction with no operand

`emit2(String op, Operand x, Operand y)` — for emitting an instruction with two operands

`emit3(String op, Operand x, Operand y, Operand z)` — for emitting an instruction with three operands

`emitLoad(Operand addr, Reg r)` — for emitting a load instruction

`emitStore(Reg r, Operand addr)` — for emitting a store instruction

`emitString(String s)` — for emitting a string literal

`emitNonInst(String s)` — for emitting a non-instruction line (e.g. label, comment, or white-space)

Use only these routines to emit instructions, since they have a built-in instruction-counting mechanism.

The CodegenVisitor Class

The main codegen program should be called `CodegenVisitor.java` and placed in a subdirectory `codegen`. You may consider declaring the following global constants and variables:

```
public class CodegenVisitor implements CodeVI {
  private final int maxTemps = 512;
  private final int maxStrs  = 32;
  private final int wordSize = Sparc.WORDSIZE;
  private Reg[] tempReg = new Reg[maxTemps];    // hold temp's reg assignment
  private String[] strBuf = new String[maxStrs]; // hold string literals
  private int tempCnt = 0;
  private int strCnt = 2;    // two pre-defined strings

  public CodegenVisitor() {}
  ...
}
```

In particular, it is useful to declare an array for holding the mapping from temps to registers and another one for saving string literals for emitting them at the end of the assembly program.

Codegen Routine Details

The following are guidelines for handling individual IR nodes:

- `MOVE` — Generate code for *src* and bring the result to a reg; if *dst* is a temp, emit a `mov` instruction; otherwise, translate *dst* to an address and then emit a `st` (store) instruction. **Note:** In the case *dst* is a `MEM` node, the recursive `accept` call should be made on `MEM`'s *exp* component, instead of on `MEM` itself. This is to avoid translating `MEM` into a load instruction.
- `JUMP` — Emit a `ba` instruction followed by a `nop`.
- `CJUMP` — First emit a `cmp` instruction; then emit a conditional branch instruction.
- `CALLST/CALL` — For system routines (i.e. *print*, *error*, and *malloc*), pass parameters through the registers `regO0` and `regO1`. For user-defined routines, pass parameters on the stack. In both cases, if there is a return value, it will be in `regO0` after the call returns.
- `LABEL` — Straightforward.
- `RETURN` — If there is an expression, generate code for it and bring the result to `regI0`. Then emit instructions `ret` and `restore`.
- `BINOP` — Emit a corresponding binop instruction. **Note:** Sparc's operands are quite restrictive: in a three-operand instruction, the first operand must be a register; the second one may be a register or a 13-bit immediate value. It is probably easiest just to bring all values into registers uniformly, at least to start with. The integer division instruction `sdiv` requires the use of a special register `regY`. The content of `regY` needs to be cleared before the execution of `sdiv`.
- `MEM` — First, generate code for *exp*, which should result in an `Operand` representing an address. Then emit a `ld` (load) instruction to load the value to a register. **Note:** `MEM` node appearing in the lhs of a `MOVE` node should be handled directly in that node, rather than going through this visit routine.
- `TEMP ---` Check the `tempReg` array to see if the temp has been assigned a register already; if so, return that register; otherwise, get a new register and save the info in the array.
- `VAR ---` Note that the SPARC supports register+offset addresses in `ld` and `st` instructions. So for a `VAR` node, the codegen should generate a `RegOff` operand, using `regFP` as the base register. The first var should have an offset of `-4`; the second, `-8`, and so on.
- `PARAM ---` Similar to the `VAR` case; but the offset value needs to be adjusted by the reserved register-window dumping space. So the first param, `(PARAM 0)`, should have an offset of `68`; the second, `72`, and so on.
- `FIELD ---` Bring the base address (i.e. the *obj* component) into a register and then generate a `RegOff` operand. The first field should have an offset of `0`.
- `(NAME wSZ) ---` Translate to an `Immed` operand with value `4`.
- `CONST ---` Translate it to an `Immed` operand.
- `FLOAT ---` Convert the floating-point value to an integer, then translate it to an `Immed` operand.

Code Organization and What to Turn In

As usual, a file `proj5-code.tar` is provided to create the directory structure. The script for running the codegen is called `runc`. Another script, `runa`, is for further compiling the Sparc assembly code to an executable code and running it. (*Remember:* You can only do this on a Sparc machine! Use `unix.cs.pdx.edu`) Your codegen's correctness will be measured by the output of `runa`. **Note:** There is no need to match your `.s` files with the reference ones. However, you should try to make the register usage count and instruction count of your programs to match or be close to those of the references'.

Submit your program `CodegenVisitor.java` through the "Dropbox" on the D2L class webpage.