Project 5 Hints: SPARC Codegen

Jingke Li

Portland State University

SPARC Assembly Code Generation

- *Input:* An IR Tree program
- *Output:* A Sparc assembly program
- *Strategy:* A visitor to the IR tree nodes
- *Register Management:*
  - A simple allocator: Upon a request, allocate the next available register; if there is no more registers available, simply issue an error and die.
  - No register saving/restore at call/return points: Works fine for simple programs.

The CodeVI Interface

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

Expression nodes return an `Operand` object.

Operand Representation

```
abstract class Operand {}

public class Reg extends Operand {
  static final int FREE=0, USE=1, TEMP=2, RSRV=3;
  String name;
  int status;
}

public class RegOffset extends Operand {
  Reg r;
  int o;
}

public class Immed extends Operand {
  int i;
  Boolean is13b; // flag for small value [-4096,4096)
}
```

Sparc.java

A Sparc library consisting of register definitions, register-allocation routines, instruction-emission routines, and and some other misc. routines. All routines are defined as stitic routines, so use them with a "`Sparc.`" prefix.

```
public class Sparc {
  static private final int MAXREGS=22;     // number of registers
  static private final int MINARGSIZE=24; // min arg-building area size
  static final int ARGOFFSET=68;           // offset of 1st arg in frame
  static final int WORDSIZE=4;             // word size
  static private int instCnt = 0;          // track inst count
  static private int regUseCnt = 0;        // track reg-use
  static private int maxRegCnt = 0;        // max reg-use count
  ...
```

Register Definitions

```
// special registers
static final Reg
    regSP = new Reg("%sp", Reg.RSRV), // stack ptr
    regFP = new Reg("%fp", Reg.RSRV), // frame ptr
    regG0 = new Reg("%g0", Reg.RSRV), // always zero
    regI7 = new Reg("%i7", Reg.RSRV), // return address
    regY  = new Reg("%y",  Reg.RSRV), // for div op
    regO0 = new Reg("%o0"),           // first arg
    regO1 = new Reg("%o1"),           // second arg
    regI0 = new Reg("%i0");           // return value

// general registers
static private final Reg[] reg = {
    new Reg("%l0"), new Reg("%l1"), new Reg("%l2"), new Reg("%l3"),
    new Reg("%l4"), new Reg("%l5"), new Reg("%l6"), new Reg("%l7"),
    new Reg("%i1"), new Reg("%i2"), new Reg("%i3"), new Reg("%i4"),
    new Reg("%i5"), new Reg("%g1"), new Reg("%g2"), new Reg("%g3"),
    new Reg("%g4"), new Reg("%o2"), new Reg("%o3"), new Reg("%o4"),
    new Reg("%o5"), new Reg("%o7") };
```

Simple Register Allocation

```
static Reg getReg() throws Exception {
  for (int i=0; i<MAXREGS; i++) {
    if (reg[i].status == Reg.FREE) {
      reg[i].status = Reg.USE;
      if (++regUseCnt > maxRegCnt) maxRegCnt = regUseCnt;
      return reg[i];
    } }
  throw new CodegenException("Out of registers");
}

static void getReg(Reg r) throws Exception {
  if (r.status == Reg.FREE) {
    r.status = Reg.USE;
    if (++regUseCnt > maxRegCnt) maxRegCnt = regUseCnt;
  } else { throw new CodegenException(...); }
}
```

Note: When a register is allocated to a temp, its status needs to be changed to TEMP, so that it would not be freed.

Freeing Registers

```
// free register r, skip if it holds a temp
static void freeReg(Reg r) {
  if (r.status == Reg.USE) {
    r.status = Reg.FREE;
    regUseCnt--;
  }
}

// free all regs, including those holding temps
static void freeAllRegs() {
  for (int i=0; i<MAXREGS; i++) {
    if ((reg[i].status == Reg.USE) ||
        (reg[i].status == Reg.TEMP)) {
      reg[i].status = Reg.FREE;
    }
    regUseCnt = 0;
  }
}
```

Instruction Emission Routines

```
static void emit0(String op) {
  emit("\t" + op + "\n");
  instCnt++;
}
static void emit2(String op, Operand x, Operand y) {
  emit("\t" + op + " ");
  x.emit(); emit(",");
  y.emit(); emit("\n");
  instCnt++;
}
static void emit3(String op, Operand x, Operand y, Operand z) {...}
static void emitLoad(Operand addr, Reg reg) {...}
static void emitStore(Reg reg, Operand addr) {...}

// for emitting labels and string literals
static void emitString(String s) {...}

// for emitting comment and white space only
static void emitNonInst(String s) { emit(s); }
```

Misc. Routines

```
static private int roundup(int x, int p) {
  return ((x+p-1)/p) * p;
}

// compute stack frame size
static int frameSize(int varCnt, int argCnt) {
  int localSize = varCnt * WORDSIZE;
  int argSize = argCnt * WORDSIZE;
  if (argSize < MINARGSIZE) argSize = MINARGSIZE;
  return roundup(ARGOFFSET + localSize + argSize, 8);
}

// print reg and inst counts
static void printStats() {
  emit("\n!Total regs:  " + maxRegCnt);
  emit("\n!Total insts: " + instCnt + "\n");
}
```

The CodegenVisitor Class

```
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];
  private String[] strBuf = new String[maxStrs];
  private int tempCnt = 0;
  private int strCnt = 2;    // two pre-defined strings

  public CodegenVisitor() {
    strBuf[0] = "L$0:\t.asciz \"Array bounds check error\\n\"";
    strBuf[1] = "L$1:\t.asciz \"%d\\n\"";
  }
  public void visit(PROG p) throws Exception {
    p.funcs.accept(this);
    printStrConsts();
    Sparc.printStats();
  }
  ...
```

Example: Hello World

```
-----------------------------------------------------
class hello {
  public static void main(String[] a) {
    System.out.println("Hello World!");
  }
}

IR_PROGRAM
main (locals=0, max_args=0) {
 [CALLST (NAME print) ( (STRING "Hello World!"))]
}
-----------------------------------------------------

        .global main
        .align 4
main:
!locals=0, max_args=0
        save %sp,-96,%sp
! [CALLST (NAME print) ( (STRING "Hello World!"))]
        sethi %hi(L$2),%o0
        call printf
        or %o0, %lo(L$2),%o0
        ret
        restore

L$2:    .asciz "Hello World!\n"

!Total regs:  1
!Total insts: 10
```

Handling FUNClist and STMTlist

```
public void visit(FUNClist fl) throws Exception {
  for (int i=0; i<fl.size(); i++)
    fl.elementAt(i).accept(this);
}

public void visit(STMTlist sl) throws Exception {
  for (int i=0; i<sl.size(); i++) {
    STMT s = sl.elementAt(i);
    Sparc.emitNonInst("!");
    s.dump();
    s.accept(this);
  }
}
```

Handling FUNC

```
public void visit(FUNC f) throws Exception {
  int framesize = Sparc.frameSize(f.varCnt, f.argCnt);
  if (f.label.equals("main"))
    Sparc.emit0(".global main");
  Sparc.emit0(".align 4");
  Sparc.emitString(f.label + ":");
  Sparc.emitNonInst("!locals=" + f.varCnt + ",
                    max_args=" + f.argCnt + "\n");
  Sparc.emit0("save %sp,-" + framesize + ",%sp");
  f.stmts.accept(this);
  Sparc.freeAllRegs();
  ...
}
```

Handling MOVE

```
public void visit(MOVE s) throws Exception {
  // generate code for s.src and bring result to a reg r
  ...
  if (s.dst instanceof TEMP) {
    Operand dst = s.dst.accept(this);
    Sparc.emit2("mov", src, dst);
    freeOperand(src);
  } else if (s.dst instanceof MEM) {
    Operand dst = ((MEM) s.dst).exp.accept(this);
    Sparc.emitStore(r, dst);
    ...
  } else {
    ...
  }
}
```

If `s.dst` is an `MEM`, do not invoke `s.dst.accept(this)`, since that would emit a load instruction.

Handling JUMP/CJUMP

```
public void visit(JUMP s) throws Exception {
  if (s.target instanceof NAME) {
    emit0("ba " + ((NAME) s.target).id);
    emit0("nop");
  } else
    throw new CodegenException("JUMP target not a NAME");
}

public void visit(CJUMP s) throws Exception {
  // bring operands to regs r1 and r2
  ...
  Sparc.emit2("cmp", r1, r2);
  Sparc.emit0(relopCode(s.op) + " " + label);
  Sparc.emit0("nop");
  ...
}
```

Handling CALLST/RETURN

```
public void visit(CALLST s) throws Exception {
  String fname = s.func.id;
  if (fname.equals("print"))       genPrint(s.args);
  else if (fname.equals("error")) genError();
  else                             genCall(fname, s.args);
}
void genCall(String label, EXPlist args) throws Exception {
  for (int i=0; i<args.size(); i++) {
    // load arg_i to a reg, then store it to
    // the i-th param slot in stack frame
  }
  Sparc.emit0("call " + label);
  Sparc.emit0("nop");
}
void genReturn(RETURN s) throws Exception {
  if (s.exp != null)
    // generate code for s.exp and bring result to RegRV
  emit0("ret");
  emit0("restore");
}
```

Handling CALLST/RETURN (cont.)

```
void genPrint(EXPlist args) throws Exception {
  if (args != null && args.size() == 1
        && (args.elementAt(0) instanceof STRING)) {
    // print a string
    String str = ((STRING) args.elementAt(0)).s;
    String lab = "L$" + strCnt;
    strBuf[strCnt++] = lab + ":\t.asciz \"" + str + "\\n\"";
    Sparc.getReg(Sparc.reg00);
    Sparc.emit0("sethi %hi(" + lab + "),%o0");
    Sparc.emit0("or %o0, %lo(" + lab + "),%o0");
    Sparc.emit0("call printf");
    Sparc.emit0("nop");
    Sparc.freeReg(Sparc.reg00);
  } else if (...) {
    // print an integer: pass two arguments to printf,
    // one control string at L$1 and one integer
    ...
  } else {
    ...
  }
}
```

Handling BINOP

- The first operand of a binop instruction must be a register.
- 13-bit immediate value can be used directly as an operand. A large immediate value must be load to a reg first (with a `set` instruction).
- `DIV` operation is implemented with the instruction `sdiv`, which require the use of a special register `%y`. The content of `%y` needs to be cleared before the execution of `udiv`.

```
// a naive version
public Operand visit(BINOP e) throws Exception {
  // Generate code for the two operands, and bring them to
  // registers r1 and r2; allocate a register r3 for result
  ...
  if (e.op == BINOP.DIV)
    Sparc.emit3("wr", Sparc.regG0, Sparc.regG0, Sparc.regY);
  Sparc.emit3(binopCode(e.op), r1, r2, r3);
  ...
  return r3;
}
```

Handling BINOP (cont.)

A BINOP Example:

```
! (BINOP - (BINOP + (VAR 1) (CONST 4096))
!          (BINOP / (BINOP * (VAR 2) (CONST 4095)) (CONST 15)))
        ld [%fp-4],%l0
        set 4096,%l1
        add %l0,%l1,%l0
        ld [%fp-8],%l1
        smul %l1,4095,%l1
        wr %g0,%g0,%y
        sdiv %l1,15,%l1
        sub %l0,%l1,%l0
```

Handling MEM

- First, generate code for the *exp* component, which should result in an `Operand` node representing an address.
- Then generate a load instruction.

```
! [RETURN (MEM (BINOP + (VAR 1) (CONST 4)))]
        ld [%fp-4],%l2
        add %l2,4,%l2
        ld [%l2],%l2
        mov %l2,%i0
        ret
        restore
```

Handling CALL

- Distinguish the system routine `malloc` from user-defined routines. For `malloc`, pass its parameter through the register %o0.
- For user-defined routines, pass parameters through the stack.

```
! [MOVE (TEMP 1) (CALL (NAME malloc) ( (NAME wSZ)))]
        mov 4,%o0
        call malloc
        nop

! [CALLST (NAME body_print) ( (VAR 1))]
        ld [%fp-4],%l1
        st %l1,[%sp+68]
        call body_print
        nop
```

Handling Other EXP Nodes

- `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 table.

```
public Operand visit(TEMP t) throws Exception {
  Reg r = tempReg[t.num];
  if (r == null) {
    r = Sparc.getReg();
    ...
  }
  return r;
}
```

- `VAR` --- Generate a `RegOffset` operand; using `%fp` as the base register:

```
public Operand visit(VAR e) {
  return new RegOff(Sparc.regFP, - e.idx * wordSize);
}
```

Handling Other EXP Nodes (cont.)

- `PARAM` --- Similar to `VAR`; but the offset value needs to be adjusted by the constant `Sparc.ARGOFFSET` (i.e. 68 bytes).
- `FIELD` --- Bring the base address (i.e. the `obj` component) into a register and then generate a `RegOffset` operand.
- `CONST` --- Generate an `Immed` operand.
- `FLOAT` --- Convert the floating-point value to an integer, then generate an `Immed` operand.

Tips on Register Handling

- Always request a regiter before use and free it after use.
- You might want to define some utility routines:

```
// load operand t to reg r
void toReg(Operand t, Reg r) throws Exception {
  if (t instanceof Immed) {
    if (((Immed) t).is13b)
      Sparc.emit2("mov", t, r);
    else
      Sparc.emit2("set", t, r);
  } else if (t instanceof RegOff) {
    Sparc.emitLoad(t, r);
    ...
  } else { // t instanceof Reg
    ...
  }
}
```

Possible Optimizations

- Put an useful instruction in the delay slot.
- Recognize 13-bit immediate values. If one appears as a second operand, don't load it to a reg; use it directly.
- For `BINOP` nodes, try to reuse operand's register for result.
- For `MEM` nodes, identify `RegOff` address mode. For instance, if we have

  ```
  (MEM (BINOP + (TEMP 1) (CONST 4)))
  ```

  then we can generate one instruction: `ld [r1+4], r2`
  instead of two:

  ```
  add r1, 4, r2
  ld [r3], r3
  ```

- When the pre-defined string `L$0` or `L$1` are not needed, don't generate them.