CS179E Phase 4: Instruction Selection

by Brandon Yi

Requirements and Specifications

This phase of the project implements instruction selection and activation records managment. At a high level, this phase takes a valid VaporM program from Phase 3 and returns a MIPS assembly language program. The MIPS program manages its own stack by maintaing a frame pointer ($fp) and a stack pointer ($sp).

Design

Overview

Phase 4 uses the same API used in Phase 3 to parse Vapor(M) programs. The API provides a syntax tree which can be used to do a depth-first traveresal on the input VaporM program. One pass is sufficient to translate VaporM into MIPS. The translation of VaporM to MIPS is handled by the `VaporVisitor` class.

`VaporVisitor`

The `VaporVisitor` class has methods which allow the program to traverse a VaporM syntax tree in a depth-first fashion. This allows for line-by-line translation. For example, here is how an assignment instruction in VaporM gets translated to MIPS:

```
public void visit(VAssign a) throws E {
        String currLine = "";
        int relPos = getRelativePos(a.sourcePos.line);

        String destReg = a.dest.toString();

        if (a.source instanceof VVarRef) {
            currLine += "move " + destReg + " " + a.source.toString();
        } else if (a.source instanceof VLitInt) {
            currLine += "li " + destReg + " " + a.source.toString();
        } else {
            currLine += "la " + destReg + " " + a.source.toString().replace(":",
"");
        }
```

```
        addLine(relPos, currLine);
}
```

Since the 3rd phase translated all local variables into stack locations or general purpose registers, translating an assignment is a matter of selection the correct instruction based on what datatype the source value is. In the example above, if the source is a register, a simple `move` instruction will work just fine. Likewise, if source is an immediate value, `li` works. However, if the source is a label, you must remove the colon from the label identifier and then use `la` to assign its value to the destination register.

Sometimes, the general purpose registers are not enough. In cases like these, `$t9` and can be used. Consider the translation of the multiplication (`mul`) instruction:

```
if (c.args[0] instanceof VOperand.Static) {
  currLine += "li $t9 " + c.args[0].toString() + "\n";
  currLine += "mul " + c.dest.toString() + " $t9 " + c.args[1].toString();
} else {
  currLine += "mul " + c.dest.toString() + " " + c.args[0].toString() + " " +
c.args[1].toString();
}
```

Since the `mul` instruction only takes registers as its operands, we must use `$t9` to store the static value of an operand. This situation arises when translating the following instruction in VaporM: `$s0 = MulS(0, $s1)`. The `0` must be stored in `$t9` in order to use the `mul` instruction in MIPS.

Stack Maintenance for Activation Records

Two pieces of code must be inserted at the beginning and end of every function. Here are the two pieces of code:

```
// Shared data
int spSize = spSize = (currFunc.stack.out * 4) + (currFunc.stack.local * 4) + 8;

// on entry
String onEntry = "";
onEntry += "sw $fp -8($sp)\n";
onEntry += "move $fp $sp\n";
onEntry += "subu $sp $sp " + spSize + "\n";
onEntry += "sw $ra -4($fp)";

/* ... */

// on exit
String onExit = "";
```

```
onExit += "lw $ra -4($fp)\n";
onExit += "lw $fp -8($fp)\n";
onExit += "addu $sp $sp " + spSize + "\n";
```

These pieces of code allows for the maintence of activiation records for every function. The allow for the stack to be setup so that local variable and arguments can be passed between activation records. Here is a diagram of the stack:

```
Higher address
          ----------------------
          |
          ----------------------
          |
$fp ->    | In / Caller Out
          ----------------------
          | ra
          ----------------------
          | fp
          ----------------------
          |
          | Local
          ----------------------
          |
$sp ->    | Out
          ----------------------
          | ra
          ----------------------
          | fp
          ----------------------

          Callee frame

Lower address
```

As you can see, the VaporM stacks (In, Out and Local) get translated into one stack in memory. This memory model is very similar to the memory model used by nearly all computer executables.

Testing and Verification

Along with the basic tests provided by this couse, I added a couple more special cases. These cases include:

Test File	Description
Call.vaporm	A more simple function call test.
ManyLocals.vaporm	A test which uses more local variables than registers available.

While this project does use `gradle`, you must supply the test files to the program manually and use a MIPS interpreter to verfity the results (e.g. `java -jar mars.jar nc P.s`).