On the course web page, you’ll find C code for a just-in-time compiler handling a small subset of JVM instructions. The Java subset supported includes integer arithmetic, integer arrays, static methods, and a minimal set of output facilities, just as in (the solution to) Homework 1. However, programs are required to have an empty operand stack at each control flow join-point; in practice, this rules out most uses of the Java conditional (?:) expression. Also, programs are restricted to 6 local variables and a maximum operand stack size of 8.

The compiler is defined by a set of C files (jit2.c, sparc.[ch], class.[ch], basics.[ch], bytecode.h), and a makefile, which generates an executable jit2. Make sure to download the latest version of these files before beginning work, and be on the lookout for possible bug fixes during the week, which will be publicized on the class mailing list. jit2 can be invoked just like the usual java interpreter on a single class file, but fails on programs outside the supported subset. Before executing the generated code for a method, jit2 displays it on standard error.

Your assignment is to improve jit2 so that it doesn’t have to generate instructions for loading the stack from local variables or with small constants, duplicating stack elements, or swapping stack elements. Moreover, you should avoid generating unnecessary mov instructions to store from the stack into local variables in cases where this can be easily detected by applying a two-instruction peephole.

Details

The provided jit2 takes a very naive approach to register allocation: local variables 0 through 5 are always stored in SPARC registers %i0 through %i5, and stack slots 0 through 7 are always stored in %l0 through %l7. We reserve the %o registers for passing values to functions we call, and the %g registers for temporary scratch values (which are not preserved over function calls). (For more details on SPARC register conventions, and for other facts about the SPARC architecture, see the SPARC V9 architecture manual referenced on the course web page.) To determine which stack slots are accessed by a given instruction, it is necessary to keep track of the stack size at each program point, which is guaranteed by the JVM specification to be uniquely defined for any verified program. A more realistic implementation would handle more than 6 locals or 8 stack slots by spilling registers to memory when necessary, but we’ll ignore this issue here.

The main disadvantage of this approach is that it generates a register-register move instruction for each load of the stack from a local variable or a constant and each store to a local variable. An alternative approach is to maintain a flexible assignment from stack slots to registers or constants; when a load occurs, the assignment is updated but data aren’t actually moved. We continue to keep local variables in fixed %i or %l registers; note that the local variables holding arguments must continue to be allocated in %i0,%i1, etc., because this is where the Sparc calling convention puts them. Any registers not needed for locals can be used to store stack values. In many cases, though, the contents of a stack slot will be identical to a local var register or a small constant, so no extra register will be needed. We can represent stack slots using a structure like this:
typedef struct slot_st {
    bool is_reg; // true ==> value is in register r; false ==> value is c
    union {
        reg r;
        int c; // must be in range [-4096,4095]
    }
} Slot;

Slot stack[method->max_stack];
int stack_size;

Stack operations now work primarily on the stack data structure. For example, an instruction like ILOAD_2 is processed by something like this:

    stack[stack_size] = (Slot) {true, varreg[2]};
    stack_size++;

Similarly, ICONST_2 is something like:

    stack[stack_size] = (Slot) {false,2};
    stack_size++;

Pure stack operations like SWAP can also be implemented just by manipulating stack, without emitting any instructions at all.

Storing is a little more complicated; ISTORE_2 is roughly:

    if (stack[stack_size-1].is_reg)
        EMIT(GEN_MOV(varreg[index],stack[stack_size-1].r));
    else
        EMIT(GEN_MOV_IMM(varreg[index],stack[stack_size-1].c));

But if varreg[index] appears in some stack slot, its value needs to be copied to a fresh register first (with an appropriate update to the stack state).

Finally, operations that produce a fresh value always need a fresh register. For example, IADD is processed by something like:

    reg target = get_reg(stack_size);
    if (stack[stack_size-2].is_reg && stack[stack_size-1].is_reg)
        EMIT(gen_op(ADD_OP,target,stack[stack_size-2].r,stack[stack_size-1].r));
    else
        ... // more cases to consider here
    stack[stack_size-2].r = (Slot) {true,target};
    stack_size--;
Here we assume that routine `get_reg(s)` returns a fresh register that is unused in slots `stack[0]` through `stack[s-1]`.

Allowing flexible stack representation helps get rid of unnecessary `mov`s corresponding to stack loads, but it leaves all the `mov`’s corresponding to stack stores. Many of these can be removed by applying a form of peephole optimization, as follows. Each time we process a bytecode instruction, we note whether the generated code computes a value into a fresh stack register. If such an instruction is immediately followed by a `STORE` to a local variable (and that variable is not held in a stack slot), we can avoid a `mov` by overwriting the target register field of the previously generated instruction contain the local variable register instead. (The `sparc.h` function `patch_rdest` does the necessary work.)

To show all these techniques at work, consider the Java Function

```java
static void foo(int a)
    int b = 20;
    int c = (a - b) * (b - a);
    a = b + c;
```

here’s what we’d like to generate:

<table>
<thead>
<tr>
<th>BYTE CODE</th>
<th>SPARC CODE</th>
<th>EMITTED</th>
<th>STACK SLOT</th>
<th>STATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>save %o6,-96,%o6</td>
<td>s0:20</td>
<td>s0:%i1</td>
<td>s0:%i2</td>
<td>s0:%i3</td>
</tr>
<tr>
<td>bipush 20</td>
<td>s0:%i0</td>
<td>s0:%i0</td>
<td>s0:%i1</td>
<td>s0:%i2</td>
</tr>
<tr>
<td>istore_1</td>
<td>s0:%i2</td>
<td>s0:%i2</td>
<td>s0:%i3</td>
<td>s0:%i4</td>
</tr>
<tr>
<td>iload_0</td>
<td>s0:%i3</td>
<td>s0:%i3</td>
<td>s0:%i4</td>
<td>s0:%i5</td>
</tr>
<tr>
<td>iload_1</td>
<td>s0:%i4</td>
<td>s0:%i4</td>
<td>s0:%i5</td>
<td>s0:%i6</td>
</tr>
<tr>
<td>isub</td>
<td>s0:%i5</td>
<td>s0:%i5</td>
<td>s0:%i6</td>
<td>s0:%i7</td>
</tr>
<tr>
<td>iload_0</td>
<td>s0:%i6</td>
<td>s0:%i6</td>
<td>s0:%i7</td>
<td>s0:%i8</td>
</tr>
<tr>
<td>imul</td>
<td>s0:%i7</td>
<td>s0:%i7</td>
<td>s0:%i8</td>
<td>s0:%i9</td>
</tr>
<tr>
<td>istore_2</td>
<td>s0:%i8</td>
<td>s0:%i8</td>
<td>s0:%i9</td>
<td>s0:%i10</td>
</tr>
<tr>
<td>iload_1</td>
<td>s0:%i9</td>
<td>s0:%i9</td>
<td>s0:%i10</td>
<td>s0:%i11</td>
</tr>
<tr>
<td>iload_2</td>
<td>s0:%i10</td>
<td>s0:%i10</td>
<td>s0:%i11</td>
<td>s0:%i12</td>
</tr>
<tr>
<td>iadd</td>
<td>s0:%i11</td>
<td>s0:%i11</td>
<td>s0:%i12</td>
<td>s0:%i13</td>
</tr>
<tr>
<td>istore_0</td>
<td>s0:%i12</td>
<td>s0:%i12</td>
<td>s0:%i13</td>
<td>s0:%i14</td>
</tr>
<tr>
<td>return</td>
<td>s0:%i13</td>
<td>s0:%i13</td>
<td>s0:%i14</td>
<td></td>
</tr>
</tbody>
</table>
• SPARC instructions like add and cmp (which is really a subcc) only allow a constant as the second operand (not the first). The simplest way to handle this is just to force the first operand into a fresh register if it isn’t already in one. For commutative operators you can do better if the second operand is non-constant, by simply switching the operands. You can also do this for comparisons (remembering, of course, to switch the sense of the comparison too!). If both operands are constants, you can do the arithmetic at compile time.

• You need to be able to deal with a constant appearing in any stack slot. For example, even an array address might be a constant (namely 0, corresponding to null), so the exception checking code must handle this. Again, the simplest thing to do is simply force such operands into registers, but you can be more elaborate if you like. (But remember that there’s no point generating efficient code that you know precedes the raising of an exception.)

**Partial Steps; Partial Credit**

This is a potentially lengthy assignment, with a fair amount of infrastructure to build. I suggest starting by getting the stack tracking and get_reg() machinery to work without special handling for constants. (In other words, just define stack as reg stack[], and force constants into temporary registers.) Getting this far will be worth 65% of your score. Then extend your machinery to handle constants. This is straightforward, but requires a fair amount of tedious case analysis, particularly if you try to do a good job with commutative operators, etc. (Think about defining some utility functions and/or macros to help.) A decent job here is worth another 25% of your score. Finally, the peephole optimization for STOREs is actually quite easy; its worth the last 10% of your score.

**How to submit your homework.**

Submit the homework by email to cs577apt@cs.pdx.edu prior to the beginning of class on the due date. You should submit a file jit2a.c, containing your modified version of jit2.c, as an attachment to your mail.