CSE 378 Autumn 1999

Machine Organization and Assembly Language Programming

Program Assignment #5

Due: Thursday, November 4

This assignment is the continuation of Assignment #4. You are going to complete the simulation of the JVM and test your program on a JVM program that you will write yourself.

The JVM machine
The JVM is a stack machine. This means as you have seen in Assignment #2 that arithmetic instructions will take their source operands from the top of the *stack* (popping twice) and store (push) the result on top of the stack (by convention, the JVM stack grows "upwards", i.e., towards increasing addresses). In addition to the stack, the JVM provides *local storage* for variables. The instructions for loading/storing local variables to/from the stack are described below.

Instruction semantics
In Assignment #4 you were given the *syntax* of the bytecodes, i.e., the opcode and the type of operands following a given opcode. We now describe the *semantics* of the bytecodes, i.e., the meaning associated with each opcode.

The *arithmetic instructions* (**IADD,ISUB,IMUL,IDIV**) all operate on signed 32-bit integers but you don't have to worry about overflow. For ISUB, the result is (v2 - v1) (v1 is at the top of the stack). For IDIV, the result is the integer quotient of (v2/v1). For IDIV, you don't have to worry about the remainder. For IMUL, you can assume that the result will fit into a single 32-bit word.

The *local variable instructions* (**ILOAD, ISTORE**) have a one-byte operand that has to be interpreted as an **unsigned index value**. This index determines the location (a word) within the *local storage* area that should be Pushed on top of the stack (ILOAD) or Poped from the stack in the location (ISTORE). You may assume that 256 variables will be enough and that this area won't overflow (of course in a real implementation you might have a larger index as well as routines to check for overflow and underflow). Each variable stored in the *local storage* is a 32-bit signed integer.

The two other *local variable instructions* (**IALOAD, IASTORE**) also load/store to/from the stack in a location in local storage but now instead of the index being given in the instruction, the value of the index is the value of the location on top of the stack. That is if the local storage area looks like (the left column is the index, the right column is the contents of the local storage area):

```
           ----------
  0           17
           ----------
  1           73
           ----------
  2            4
           ----------
  3           27
           ----------
  4           13
           ----------
 etc.
```

and if the stack looks like

```
           ----------
             214
           ----------
              19        <---- ``below'' top of the stack
           ----------
               3        <---- top of the stack
           ----------
```

(Note that the convention in this example is to have the top of the stack point to the last value pushed on top of the stack.)

The bytecode IALOAD will result in the new stack configuration

```
           ----------
             214
           ----------
              19
           ----------
              27       <---- top of the stack
           ----------
```

that is we have performed something like "POP v1" followed by "Push local(v1)'.

IASTORE stores the element below the top of the stack at the local storage location whose index is on the top of the stack. Then the two top entries on the stack are popped. That is with the same initial conditions as before the IALOAD above, IASTORE would result in

```
           ----------
  0            17
           ----------
  1            73
           ----------
  2             4
           ----------
  3            19
           ----------
  4            13
           ----------
 etc.
```

and

```
           ----------
              214       <---- top of the stack
           ----------
```

The *immediate instructions* (**BIPUSH, SIPUSH**) are followed respectively by an 8-bit and a 16-bit signed immediate value. When BIPUSH is executed, the 8-bit immediate value is to be sign-extended to 32 bits and Pushed on top of the stack. For SIPUSH, the immediate value is calculated as

$(\mathit{immed}1 \ << \ 8)\ \mathit{OR}\ \mathit{immed}2$

(where $<<$ is a logical left shift) and then sign-extended to 32 bits and Pushed on top of the stack (i.e., "SIPUSH 0x80 0x01" will push the 32-bit value 0xffff8001 on top of the stack and "SIPUSH 0x01 0x80" will push the 32-bit value 0x00000180).

The *duplicate instructions* duplicate the top of the stack (**DUP**) or the two top locations of the stack (**DUP2**). With the initial stack:

```
           ----------
              214
           ----------
               19
           ----------
                3       <---- top of the stack
           ----------
```

DUP would result in

```
----------
   214
----------
    19
----------
     3
----------
     3        <---- top of the stack
----------
```

and DUP2 would result in

```
----------
   214
----------
    19
----------
     3
----------
    19
----------
     3        <---- top of the stack
----------
```

The *branch instructions* (**IFNE, IFEQ, IFLE, IFLT, IFGE, IFGT**) compare the top of the stack to the value 0 and then pop the top of the stack. These instructions as well as **GOTO** are followed by 2 bytes: *offset1* and *offset2*. If the comparison is successful and in the case of the GO TO, the control in the interpreted program is transferred to the instruction whose offset, IN BYTES, relative to the start address of the branch instruction is computed as $(offset1 \ << \ 8)\ OR\ offset2$ (e.g., "GOTO 0x00 0x03" is a no-op since it transfers to the instruction following the GOTO; "GOTO 0xff 0xfd" will transfer to the instruction whose start is 3 bytes before the GOTO).

Finally, the **POP** instruction discards the top of the stack and the **IRETURN** instruction signals the end of the computation (in this simplified machine we don't have call/return facilities). The value at the top of the stack is popped and used as the return value to the main program.

Your task

Complete the JVM interpreter that you started in Assignment #4 Recall that the basic structure of the interpreter should be a loop that will go through the 5 steps

1. Fetch the next bytecode
2. Decode it
3. Fetch the operands (if any)
4. Execute the operation
5. Store the results (if any)

You should test your interpreter on test programs of your own so that every bytecode is used at least once.

You will be **turnin** in both:

- your JVM interpreter function (written in SPIM), and
- an array computation program (written in JVM)

The JVM program you have to write is very similar to Part II Problem 1 of Assignment #3 and is described below. It has the same specifications but what you have to compute is slightly different, namely:

- the number of elements strictly greater than the last element
- the minimum element
- the maximum element
- the (integer) average of all elements

(I did not want you to compute the number of even elements because the (partial) JVM that you have has no shift instructions and the IDIV is lame.)

These results should be put in the first few locations of the *local storage* area with, of course, the program being in the JVM program area.

The local area should contain (in this order)

- the number of elements strictly greater than the last element (initialized to 0)
- the minimum element (initialized to 0)
- the maximum element (initialized to 0)
- the (integer) average of all elements (initialized to 0)
- the size of the sorted array

- the array itself

(Be careful that the array and the few extra variables that you will need must fit in the local storage area which is limited to 256 variables).

The environment provided to you

The environment is as in Assignment #4, i.e.,

- $a0 the address of the program array (that will contain your test program)
- $a1 – the initial address of the top of the JVM stack
- $a2 – the address of location 0 of the local storage of the JVM