Problem Set #3

Due: Part I Wednesday April 23rd
Due: Part II Wednesday April 30th

This assignment is a little longer and parts of it are tedious (such is life) so (1) It is due in two parts and (2) you can do it in groups of 2, if you so desire.

The assignment has two goals: (1) give you an idea of how behavioral machine simulators are built, and (2) give you an introduction to the Java Virtual machine (JVM), a machine-independent representation of high-level programming languages (or another form of Assembly Language), which is based on the concept of stack architectures (recall Problem Set #2).

The JVM machine
The JVM is a stack machine. This means as you have seen in Problem Set #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. A small subset of the JVM instruction set is in the table below:

<table>
<thead>
<tr>
<th>Name</th>
<th>Bcode</th>
<th>Instr. format</th>
<th>Stack operations</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IADD</td>
<td>0x60</td>
<td>IADD</td>
<td>POP v1, POP v2, PUSH res</td>
<td>Integer add</td>
</tr>
<tr>
<td>ISUB</td>
<td>0x64</td>
<td>ISUB</td>
<td>POP v1, POP v2, PUSH res</td>
<td>Integer sub</td>
</tr>
<tr>
<td>IMUL</td>
<td>0x68</td>
<td>IMUL</td>
<td>POP v1, POP v2, PUSH res</td>
<td>Integer mul</td>
</tr>
<tr>
<td>IDIV</td>
<td>0x6b</td>
<td>IDIV</td>
<td>POP v1, POP v2, PUSH res</td>
<td>Integer div (quotient)</td>
</tr>
<tr>
<td>ILOAD</td>
<td>0x15</td>
<td>ILOAD index</td>
<td>PUSH res</td>
<td>St to loc. var. at index</td>
</tr>
<tr>
<td>ISTORE</td>
<td>0x36</td>
<td>ISTORE index</td>
<td>POP v1</td>
<td>St from local</td>
</tr>
<tr>
<td>ILOAD</td>
<td>0x2e</td>
<td>ILOAD</td>
<td>see text</td>
<td>St to local</td>
</tr>
<tr>
<td>IASTORE</td>
<td>0x4f</td>
<td>IASTORE</td>
<td>see text</td>
<td>St from local and pop</td>
</tr>
<tr>
<td>SIPUSH</td>
<td>0x10</td>
<td>BIPUSH imm</td>
<td>PUSH res</td>
<td>Byte immediate push</td>
</tr>
<tr>
<td>SIPUSH</td>
<td>0x17</td>
<td>SIPUSH imm1 imm2</td>
<td>PUSH res</td>
<td>Short immediate push</td>
</tr>
<tr>
<td>DUP</td>
<td>0x59</td>
<td>DUP</td>
<td>DUP</td>
<td>Duplicate top of stack</td>
</tr>
<tr>
<td>DUP2</td>
<td>0x5c</td>
<td>DUP2</td>
<td>Dupl. top 2 st. entries</td>
<td></td>
</tr>
<tr>
<td>POP</td>
<td>0x57</td>
<td>POP</td>
<td>POP v1</td>
<td>POP value, discard</td>
</tr>
<tr>
<td>IFEQ</td>
<td>0x99</td>
<td>IFEQ offset1 offset2</td>
<td>POP v1</td>
<td>Branch on v1 == 0</td>
</tr>
<tr>
<td>IFNE</td>
<td>0x9a</td>
<td>IFNE offset1 offset2</td>
<td>POP v1</td>
<td>Branch on v1 !=0</td>
</tr>
<tr>
<td>IFLT</td>
<td>0x9b</td>
<td>IFLT offset1 offset2</td>
<td>POP v1</td>
<td>Branch on v1 &lt;0</td>
</tr>
<tr>
<td>IFLE</td>
<td>0x9c</td>
<td>IFLE offset1 offset2</td>
<td>POP v1</td>
<td>Branch on v1 &lt;=0</td>
</tr>
<tr>
<td>IFGT</td>
<td>0x9d</td>
<td>IFGT offset1 offset2</td>
<td>POP v1</td>
<td>Branch on v1 &gt;0</td>
</tr>
<tr>
<td>IFGE</td>
<td>0x9e</td>
<td>IFGE offset1 offset2</td>
<td>POP v1</td>
<td>Branch on v1 &gt;=0</td>
</tr>
<tr>
<td>GOTO</td>
<td>0xa7</td>
<td>GOTO offset1 offset2</td>
<td>none</td>
<td>unconditional branch</td>
</tr>
<tr>
<td>IRETURN</td>
<td>0xac</td>
<td>IRETURN</td>
<td>POP v1</td>
<td>Integer return</td>
</tr>
</tbody>
</table>

JVM Instruction Syntax
Each byte code is 1 byte long. An index (as in ILOAD index) is a 1 byte long unsigned integer. imm1
and imm2 are both 1 byte long and when concatenated form a signed immediate value of 2 bytes. Thus BIPUSH is followed by 1 byte of immediate while SIPUSH is followed by 2 bytes of immediate. offset1 and offset2 are also 1 byte long meaning that each branch instruction is followed by a 16-bit offset.

Registers v1 and v2 are part of the register component of the JVM machine. The latter will be described in more detail on page 3.

**JVM Instruction Semantics**

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. Note that the JVM has a dedicated register used as a stack pointer.

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 Popped 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. In other words, IALOAD corresponds semantically to POP followed by ILOAD local(v1) but this last instruction is not available as such. So if the value 3 was at the top of the stack and the value 27 in word “local + 3” the value 27 would replace the value 3 on top of the stack.

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. So, if the value 3 was on top of the stack and the value 27 below it, then 27 would be stored in “local + 3” and the stack popped twice.

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 \((immed1 << 8) OR immed2\) (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 0x00” will push the 32-bit value 0x00000100).

The duplicate instructions duplicate the top of the stack (DUP) or the two top locations of the stack (DUP2).

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. These offsets are RELATIVE to the JVM’s program counter PC. Of course the JVM has a dedicated register used as PC If the comparison is successful and in the case of the GOTO, 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 stores the value at the top of the stack in v1 and pops the stack. 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.

An example “program” (that does not do anything) would be like:

```
ILOAD 18
ILOAD 6
IADD
DUP
BIPUSH -95
ISUB
ISTORE 1
```

**Your task. Part I. Writing an interpreter for the JVM in SPIM**

Your task is to write an interpreter for JVM in SPIM and to test it snippets of programs written in JVM machine language (like the one above but translated into byte code as in:

0x15, 0x12, 0x15, 0x06, 0x60, 0x59, 0x10, 0xa1, 0x64, 0x36, 0x01

The basic basic structure of the interpreter, i.e., the SPIM code, 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)

The JVM machine that you need to simulate has 4 components:

- Registers:
  - the JVM program counter $PC_{JV_{M}}$ which points to the next bytecode to be interpreted
  - the JVM stack pointer $SP_{JV_{M}}$ that points to the top of the JVM stack
  - the registers v1 and v2 as shown in the table.

Note that these registers are NOT THE SAME as those used in SPIM. For example $sp$ is a register that points to the top of the SPIM stack while $SP_{JV_{M}}$ points to the top of the JVM machine that you are simulating. You should assign either a memory location, or preferably a SPIM register that does not have a special purpose in SPIM to the JVM stack pointer. This is true also of course of $PC_{JV_{M}}$, v1 and v2.

- The JVM program to be interpreted, i.e., a sequence of byte codes.
- The JVM stack
- The JVM local area

A skeleton program to show this lay-out is included at the end of this hand-out.

Although the JVM program that you will write (see below) might not use all the bytecodes defined in the table, your simulator should include the simulation of all the bytecodes.

**Your task. Part II. Writing a program in JVM bytecodes**

The JVM program you have to write is very similar to Problem 5 of Problem Set #2. 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

(We 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 JVM local storage 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).

**Your task. Part III. “Executing” the program written in Part II on the simulator written in Part I**

Instructions for **turnin** will be given shortly. What you will have to turn in will be:

- Part I: your JVM interpreter function (written in SPIM). It might be difficult to test the correctness of the interpretation but at the very least you should be able to recognize and decode all bytecodes and attempt to execute them. You can do that on very simple JVM bytecode programs

- Parts II and III: the array computation program (written in JVM) and its correct execution on the interpreter. You will be given help in “assembling” the JVM program soon.
The skeleton program

#--------{ global data section }----------#
# declare all global variables and string #
# constants in this section. #
#------------------------------------------#
.data

# A sample set of bytecodes.
jvmbytecode: .byte 0x60, 0x15, 0x10, 0x10, 0xfe, 0x17
      .byte 0x18, 0xe4, 0x99, 0x05, 0xe8, 0x00
      .align 2
jvmstack: .space 1024 # Allocate stack space
jvmlocal : .word 0,0,0,0 # Reserve space for the 4 answers
      .word 12 # length of the array to be examined
      .word 13,5,7,2,13,234,13,19,20,17,23,14 # an example array
      .space 956

#--------{ code section }------------------#
# place all main code and procedure code in #
# this section of the file #
#------------------------------------------#
.text
# declare main as a global symbol
.globl main
main:

    subu $sp, $sp, 4 # create 3 new words on the stack
    sw $ra, 4($sp)  # store the return address

# call the JVM interpreter after storing its arguments in the
# appropriate registers
    la $a0, jvmbytecode # store the bytecode address in $a0
    la $a1, jvmstack   # store the stack address in $a1
    la $a2, jvmlocal   # store the local address in $a2
    jal JVM             # call the Java Virtual Machine

# When the JVM is done, we have nothing to do, so just exit our
# program.

    lw $ra, 4($sp)    # restore the return address
    addu $sp, $sp, 4  # restore the stack
    j $ra             # return