Machine Organization and Assembly Language Programming

# Problem Set #3

Due: Part I Wednesday April 21st
Due: Parts II and III Wednesday April 28th

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:

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

**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. *offfset1* 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 (for ILOAD) or be overwritten with an element that is Popped from the stack (for ISTORE). The **POP** instruction discards the top of the stack and moves the stack pointer accordingly. However, implementation-wise, the value at the top of the stack is stored in v1. 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 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 (i.e., the top of the stack and the one "below" it) (**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 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 **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. Another way to end the computation is to have a bytecode of 0x00.

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

```
poi:     ILOAD 18    #the use of the label will become clear
         ILOAD  6    #when you ‘‘assemble’’ bytecode programs
         IADD        #see page 4
         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 on 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 $PC_{JVM}$ which points to the next bytecode to be interpreted
  - the JVM stack pointer $SP_{JVM}$ 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 $SP_{JVM}$ 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 $PC_{JVM}$, 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 similar to Problem 4 of Problem Set #2, i.e., given an array of integers and its size, find:

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

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 turnin 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.

**Assembling a byte code program**
The *perl* language is useful for translating patterns into other patterns. The perl program $jvm-assemble\_pl$ available from the Web page, will transform a program written in JVM assembly language to a sequence of byte codes that can be pasted into your .data section of a SPIM program. The command line in Linux is:

```
perl jvm-assemble_pl <bytecode file> <start label>  > <output file>
```

For example if the "program" on page 3 were in a file named *abc*, then the command line in Linux:

```
perl jvm_assemble_pl abc poi > def
```

would generate the output file:

```
            .align  2
poi:
            .byte   0x15, 0x12              # iload
            .byte   0x15, 0x06              # iload
            .byte   0x60                    # iadd
            .byte   0x59                    # dup
            .byte   0x10, 0xa1              # bipush
            .byte   0x64                    # isub
            .byte   0x36, 0x01              # istore
            .byte   0x00                    #
            .align  2
end_poi:
```

**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 1 new word 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
```