CSE 378 Winter 1998

Machine Organization and Assembly Language Programming

# Problem Set #5

Due: **Wednesday, February 11**

This assignment is the continuation of Assignment #3. You are going to complete the simulation of the JVM and test your program on a small JVM program that you will write yourself.

In order to write the program, you will need one more bytecode, namely:

| Name | Bcode | Instr. format | Stack operations | Description |
|---|---|---|---|---|
| IDIV | 0x6b | IDIV | Pop v1, Pop v2, Push res | Integer div |

**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 #3 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). For IDIV, the result is the integer quotient of (v2/v1). For IDIV, you don't have to worry about the remainder.

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 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 *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 *branch instructions* (**GOTO, 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 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). Since we don't have a JVM assembler, you'll will have to compute these addresses yourself in the test programs you will write.

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 #3. 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.

When you **turnin** your program, the test program that you should turn in is a binary search algorithm (cf. details below). Since the basic JVM that you interpret does not have call/return instructions, use an iterative version of binary search (any elementary programming textbook should have a version of this algorithm).

### The environment provided to you

The environment is almost the same as in Assignment #3. A slightly modified version of `jbm.s` can be found in:

`/cse/courses/cse378/CurrentQtr/ProblemSets/assn5/jvm.s`

and recall that

- Your interpreter routine should be called `java_interpret`.
- Your test program should be loaded in the `program` area. This is done by loading $a1 with the address of the test program and $a2 with its length before the call to memcpy (`jal memcpy`)
- `java_interpret` is called with 3 arguments

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

In the case of the binary search test program, you will have to fill the JVM local storage area. This should be done in `jvm.s` using `memcpy` as is done for the test program. The local area should contain (in this order)

- the key that you want to search for (at location 0)
- the size of the sorted array (at location 1 – counted in words)
- the sorted array itself ( use only positive integers)

(Be careful that the size of the array in words in the local storage area and the size of the local area in bytes that you will pass in $a2 to `memcpy` are not the same!).

The value that you should return is the value of the match if the search is successful, and the value "-1" if it is not. You should put that value in $v0 before returning to `jvm.s`