THE UNIVERSITY OF WESTERN ONTARIO

Computer Science 2208a - Fall 2012
Fundamentals of Computer Organization

ASSIGNMENT #5

Due Dec 5, 2012 at 11:59 PM

1 Implementing a Mini JVM

As you know, a program written in Java is compiled by a Java compiler into a machine independent language called *bytecode*, output into a file with the extension `.class`. A Java Virtual Machine(JVM) for your computer then interprets the bytecodes in the `.class` file, thereby executing the program on your machine.

We normally cannot view the `.class` files, since they are binary files. The Java `.class` file format contains a lot of information in addition to the bytecodes for the methods in the file (e.g. header information, class inheritance information, table of constants, multiple methods, etc.).

In this assignment, we will be using a *mini* version of a `.class` file (so we will call them `.miniclass` files). Our `.miniclass` files will contain only bytecodes, and only bytecodes for one method; however, the format of the bytecodes is the same as that found in real `.class` files. You will be writing part of a Java Virtual Machine (JVM) for a SPARC: code to interpret the bytecodes in a `.miniclass` file.

2 Learning Objectives

The purpose of this assignment is to have you:

1. gain an understanding of how Java bytecode works, and how a Java Virtual Machine interprets bytecode
2. use local variables and data structures in SPARC assembly language
3. gain more experience with passing parameters to functions
4. gain some experience in understanding and building on someone else's code
5. follow specifications
6. practice good programming style (including writing good comments)

3 Provided Code

To help you get started, you have been provided with the following code:

- `mini_jvm.m`: a program that already implements a subset of a JVM for the SPARC

You have also been provided the following code, which you will modify as part of the assignment:

- `asn5.m`: a main program for this assignment
- `get_codes.m`: a helper function described later

3.1 `mini_jvm.m`

The program in `mini_jvm.m` interprets bytecodes needed for integer arithmetic, as detailed below. `mini_jvm.m` reads bytecodes from the keyboard. Figure 1 shows a simple example of a run of the provided `mini_jvm` program, showing the bytecodes being input from the keyboard. The bytecode program that it executes calculates the sum of the integers 5 and -3 and outputs the result. Figure 2 explains each line of input to `mini_jvm` in Figure 1.

```
obelix[11]% ./mini_jvm
> 16 5
> 16 -3
> 96
> 187
* 2
> 177
```

Figure 1: Sample output of provided `mini_jvm.m`

Bytecodes	Instruction / Operands	Description
`16 5`	`bipush 5`	Push the integer 5 on the stack
`16 -3`	`bipush -3`	Push the integer -3 on the stack
`96`	`iadd`	Pop and add top two elements on the stack; push result back on stack
`187`	`iprint`	Print top element on stack
`177`	`return`	Exit the program

Figure 2: Deciphering the input in Figure 1

The bytecodes that are already implemented and interpreted by the provided code in `mini_jvm.m` are described in Figure 3, but you will be modifying some of them. Note that `iprint` is not a real Java bytecode operation. In *real* bytecode, code `187` is unused. However, for our purposes, we will use it to display integer values to the screen so we can see the results of our programs.

Mnemonic	Code	Args	Description
`bipush`	`16`	`b`	Push signed byte `b` onto the stack ($-128 \leq b \leq 127$)
`pop`	`87`		Pop top item off stack (it is discarded)
`iadd`	`96`		Pop and add the top two items on the stack; push result back on stack
`isub`	`100`		Pop and subtract the top two items: `(second from top) - (top)`; push result back on stack
`imul`	`104`		Pop and multiply the top two items; push result back on stack
`idiv`	`108`		Pop and integer divide the top two items: `(second from top) / (top)`; push result back on stack
`return`	`177`		Return from method
`iprint`	`187`		Print the top item on the stack (without popping it off the stack)

Figure 3: Bytecodes implemented in `mini_jvm.m`

4 Task

You will be building upon the provided code to implement a mini JVM (`asn5.m`) that reads bytecodes from a `.miniclass` file specified as a command line parameter (with the help of the provided function in `get_codes.m`), and interprets each bytecode, one at a time (using a function which you will write in `execute_codes.m`).

4.1 `asn5.m`

You have been provided with the bare bones of a main program for our Java Virtual Machine in a file called `asn5.m`. You will modify this file, as described below, to take the filename of the `.miniclass` file as a command line parameter. For example, if the executable for your assignment is called `asn5`, to run it on a file called `sample.miniclass`, you would type:

```
obelix[11]% ./asn5 sample.miniclass
```

The main program calls a function `get_codes`, which reads the bytecode stream from the file into a byte array. A description of this function is provided in Figure 4.

Function:	`get_codes`
Description:	Reads the bytecode stream from the `.miniclass` file into a bytecode array in memory
Parameters:	1. The address of the string containing the filename of the `.miniclass` file
	2. The address of the bytecode array into which the file will be read
Returns:	The number of bytes read from the file, or `-1` if the file was not opened or read successfully

Figure 4: `get_codes` function

The second parameter to `get_codes` is already setup for you in `asn5.m`, but you will need to provide the first parameter, which you will read from the command line arguments, as noted above.

The main program then calls a function `execute_codes` (which you will write, as described in the next section) that interprets the bytecodes contained in the bytecode array. You will make five modifications to this main program:

1. code to check whether the command line has the correct number of parameters.
   - If it does not, print an error message.
   - Exit the program **and return `1` to the operating system**.
2. code to read the name of the `.miniclass` file as a command line parameter. See Topic 7 for a refresher.
3. code to check if the filename passed on the command line does not exist or there was an error in reading the file (see the description of `get_codes` below).
   - In either case, print an error message.
   - Exit the program **and return `2` to the operating system**.
4. after the call to `execute_codes`, check the return value of `execute_codes`. If the bytecode program resulted in a division by zero error, **return `3` to the operating system**.
5. at the end of the program (assuming we didn't exit with an error), you must **return `0` to the operating system** to indicate a successful termination.

You must not make any other changes to the main program `asn5.m`.

4.1.1 Return Values

A `main` function can return an integer value to the operating system, which provides an explanation of why the program terminated. This is why we often see C programs written as in Figure 5.

```
int main(int argc, char** argc)
{
    ...
    return 0;
}
```

Figure 5: Return value from a C program

By convention, when a program exits successfully, it returns `0` to the operating system. If a program exits with an error, it returns a non-zero value.

When using a shell, a program's exit status is always stored in the special variable `$?`. Figure 6 shows an invocation of an executable `errorprogram`, which exits with an error and returns 5 to the operating system.

```
obelix[11]% ./errorprogram
obelix[12]% echo $?
5
obelix[13]% echo $?
0
```

Figure 6: Examining the return value of a program

Observe that the first time we checked the value of `$?`, it printed `5`, since that was the return value of `errorprogram`. The second time we checked the value of `$?`, it printed `0`, since that was the return value of the previous invocation of `echo`.

Do not forget to return the appropriate values from your `main` function to indicate the program exit status to the operating system. How do you do this? To answer this, think for a moment about how any function returns a value to its caller.

Figure 7 reiterates the exit values that you should use in `asn5.m`.

Exit Status	Description
0	Normal program termination
1	Invalid number of command line parameters
2	Error reading specified `.miniclass` file
3	Division by zero occurred while interpreting the bytecode

Figure 7: Exit statuses for `asn5.m`

4.2 `execute_codes.m`

You will write the function `execute_codes` and place it in a file `execute_codes.m`. It must be a non-leaf function and must conform to the SPARC protocol for calling functions. Figure 8 provides an overview of `execute_codes`.

Function:	`execute_codes`
Description:	Interprets the bytecodes contained in the bytecode array passed to it
Parameters:	1. The address of the bytecode array
	2. The number of bytes in the bytecode stream
Returns:	`0`, if the bytecode program executed successfully
	`-1`, if the bytecode program resulted in a division by zero error

Figure 8: `execute_codes` function

The code in `execute_codes` MUST be based on the code that you are given in `mini_jvm.m`, and you MUST use the existing register and constant names (and add others, of course). The main difference is that you will read each instruction from the bytecode array rather than reading it from the keyboard.

Execution will begin at the first byte in the array, and will continue until the `return` bytecode is reached OR until there is a division by 0 error.

You may design and write other functions for modularity, if you wish. Keep in mind that you do NOT want a lot of small functions, since that will reduce the runtime efficiency of the JVM. These functions may be leaf functions, but all functions must conform to the SPARC protocol for calling functions and follow SPARC conventions as in the lecture notes.

4.2.1 Interpreting the Bytecodes

The following provides a suggested algorithm for interpreting the Java bytecodes in the bytecode stream:

Define a local register as a *program counter* and another local register as an *instruction register*. Execution then proceeds as follows:

1. Initialize the program counter to the first byte in the bytecode array.
2. Load the instruction pointed to by the program counter into the instruction register.
3. Determine the type of instruction in the instruction register and perform the appropriate action for that instruction.
4. Update the program counter to point to the next instruction to be executed (either the next instruction in the byte array, or the target of a branch).
5. Go back to step 2.

4.2.2 The Operand Stack

You MUST use the operand stack as implemented in the provided code. Note that this operand stack is implemented as an array that is a local variable. In the provided code, it is a local variable in the main program in `mini_jvm.m`. In Assignment 5, it will be a local variable in the function `execute_codes`.

In Java, an integer is stored in a word, so the operand stack is a stack of **words**. It uses a local register as the *stack pointer*, and is set up so that it grows toward **smaller** addresses.

4.2.3 Storage for Local Variables of the Java Method

Local variables in a Java method are stored in an array of **words**, and are identified by their offset or *index* from the beginning of the array, with the index starting at 0. We will assume that the first local variable declared in the Java method is at index 0, the second at index 1, etc.

This *local variable* array will be implemented as a local variable of your `execute_codes` function. You may assume that there will be no more than **12** word-sized local variables in a Java method.

4.2.4 Division by Zero

The provided code prints an error message if a division by zero is attempted for `idiv`. You should do the same when interpreting `irem`: if the operand by which to divide is zero, print an error message.

Additionally, in both `idiv` and `irem`, after printing an error message, you should immediately return `-1` to the calling function to indicate that an error occurred.

4.2.5 Return Instruction

The `return` instruction should return `0` to the calling function to indicate that all bytecodes were interpreted successfully.

4.2.6 Assumptions

- A JVM would assume that all bytecode operations are valid, since they were generated by a Java compiler. Therefore, you may also assume that the bytecodes are valid.
- You may assume that the offset for a branching instruction will be such that a branch will not go outside the bytecode stream. Again, this would be normal since the offsets would have been generated by a Java compiler.
- You may assume that the operand stack is large enough so that it will not overflow nor underflow, so that you do not need to check for this in your code.
- For simplicity in this assignment, assume that the bytecode stream will be no longer than 100 bytes.
- Also for simplicity, you may assume that there will be no more than 12 word-sized local variables in a Java method.
- You may assume that the last bytecode in any bytecode stream will be `177` (`return`).

4.3 Overview of Bytecodes to Implement

This section gives an overview of the bytecodes that you will be implementing. More specific implementation details are provided later. The bytecodes that you will add to your JVM are a sampling from different types of instructions, including some arithmetic, logical, branching, and load/store instructions. Figure 10 summarizes the bytecodes that you will be implementing.

Mnemonic	Code	Arguments	Description
`bipush`	16	`b`	This operation is already implemented for you in `mini_jvm.m`, but you will need to modify it to read from the bytecode stream, rather than from the keyboard, as explained later
`iconst_0`	3		Push constant `0` onto the stack
`iload`	21	n	Push local variable at index `n` onto the stack
`istore`	54	n	Pop stack and store in local variable at index `n`
`dup`	89		Duplicate the top item on the stack
`idiv`	108		This operation is already implemented for you in `mini_jvm.m`, but you will need to modify it to return a specific value if division by zero occurs. This is described later.
`irem`	112		Find the remainder on integer division: `(second from top) / (top)` push result back on stack
`ishr`	122	n	Arithmetic shift right: pop and shift the top item on the stack `n` bit positions to the right with an arithmetic shift, where `n` is the value in the low 5 bits of the next item on the stack. It too is popped, and the shifted result is pushed back on the stack.
`iushr`	124	n	Logical shift right: same as ishr, except it is a logical shift to the right
`iinc`	132	`n d`	Increment local variable at index `n` by `d`
`i2b`	145		Int to byte: pop the top item from the stack, truncate it to a byte, then sign-extend it and push the result back onto the operand stack
`i2c`	146		Int to char: pop the top item from the stack, truncate it to a byte, then zero-extend it andpush the result back onto the operand stack
`ifne`	154	`offset`	Pop top item off the stack, branch to `offset` if it is not equal to zero. `offset` is a signed 16-bit value.
`goto`	167	`offset`	Branch to `offset`. `offset` is a signed 16-bit value.

Figure 9: Bytecodes to Implement

4.4 Bytecode Pseudocode

The following describes how each of the bytecode operations work, for the ones that require more detail than that given in the tables above:

- `bipush`: The `bipush` bytecode is immediately followed by a single byte argument `b` which specifies the value to be pushed onto the stack.
    - Read the byte argument `b`.
    - Push that value onto the operand stack. Note that it will be stored on the operand stack as a **word**.

- `irem`:
  - Pop the top two items off the operand stack.
  - Find the remainder.
  - Push the result back onto the operand stack.

  The pseudocode for the other arithmetic , type conversion and logical operations is similar (idiv, ishr, iushr, i2b, i2c).

- `dup`: the `dup` bytecode has the effect of duplicating the top item on the stack:
  - Pop the top item off the operand stack.
  - Push two new copies of that item onto the operand stack.

- `iload`: the `iload` bytecode is immediately followed by a single byte argument `n` which specifies the offset into the local variable array (starting from index 0):
  - Read the byte argument `n`.
  - Copy the value from element `n` of the array and push it onto the operand stack.

- `istore`: the `istore` bytecode is immediately followed by a single byte argument `n` which specifies the offset `n` into the local variable array (starting from index 0):
  - Read the byte argument `n`.
  - Pop the top value off the operand stack and store it into the element with index `n` in the local variable array.

- `iinc`: the `iinc` bytecode is immediately followed by two byte arguments `n` and `d`, which specify:
  1. the offset `n` into the local variable array (starting from index 0).
  2. the value `d` by which to increment that variable.

  - Read the byte argument `n`.
  - Read the byte argument `d`.
  - Add the value `d` to the value in the element with index `n` in the local variable array.

- `ifeq`: the `ifeq` bytecode is immediately followed by a single signed 16-bit argument `n` which specifies the offset for the branch which should be taken if the comparison to zero fails:
  - Read the signed 16-bit argument `n`.
  - Pop the top element off the operand stack.
  - Branch to the offset if it is not equal to zero, i.e. branch to the bytecode instruction that is at the current instruction (the `ifeq`) plus the offset. Otherwise, drop through to the next bytecode instruction in the bytecode stream.

4.4.1 Efficiency Note

Note that the descriptions for the bytecodes may say, for example, that you pop an operand and push the result. However, you do not need to *physically* do a pop and a push in your implementation. Note how the sample code does not always actually change the stack pointer when doing a `pop` followed by a `push`!

5 Resources

In completing this assignment, you may find the following resources helpful:

- Lecture Notes, Topics 7 and 11
- The 2nd edition of the textbook has a section on the Java Virtual Machine (section 1.6, pages 20-29)
- The book, *The Java Virtual Machine Specification*, Java SE 7 Edition, is an excellent resource on the JVM. The entire book is available on the web at:

  http://docs.oracle.com/javase/specs/jvms/se7/html/index.html

  You might find Chapter 6, *The Java Virtual Machine Instruction Set* useful for this assignment. The whole book is interesting since it contains everything you need to know to write a complete Java Virtual Machine

6 Testing Your Program

On the assignment 5 web page, you will find `.miniclass` files that you can use for testing your assignment. These are binary files and therefore cannot be viewed.

The *source code* for these `.miniclass` files is provided in `.minijava` files. You can use these files to trace the algorithms implemented in each `.miniclass` file and determine whether or not your program is producing the correct results.

Be sure to also test the exit status codes from your program, as summarized in Figure 7. Once again, you can check the value of the special shell variable `$?` after invoking your program to view its exit status.

7 General Implementation Specifications

1. It is not required that you fill *all* possible delay slots. However, you should be filling the *obvious* delay slots in your program.
2. Use macros to give symbolic names to constants. You should not be using "magic numbers". By convention, constant names are *capitalized*, and multiple words in a constant are separated by underscores (e.g. `NUM_CHARS`).
3. Use only local registers (`%l0` to `%l7`) for temporary storage, and registers `%o0` and `%o1` for function parameters as per convention. Efficiency note: you can leave a result that is returned in `%o0`, if you will then be using it as the input parameter in `%o0` for the next function call.
4. Use macros to give symbolic names to the registers that you use, EXCEPT FOR the registers `%o0` and `%o1` (i.e. do **not** use symbolic names for registers `%o0` and `%o1`). By convention, symbolic register names are all in lowercase and end in `_r` (for example, `time_r`). Multiple words are separated by underscores and are not camel-cased (e.g. `user_age_r` and not `userAge_r`).
5. You are expected to use good programming style and include complete commenting (see the **Commenting Guidelines** on the CS 2208 web site).

8 Bonus- Floating Point Numbers (10%)

To qualify for bonus marks, you must have implemented all instructions in the assignment and must pass 80% of the automated tests on your code.

Implement the following additional instructions, which deal with floating point values.

Mnemonic	Code	Arguments	Description
`fconst_0`	`11`		Push floating point constant `0.0f` onto the stack
`fload`	`23`	n	Push local floating point variable at index `n` onto the stack
`fstore`	`56`	n	Pop stack and store floating point in local variable at index `n`
`fadd`	`98`		Pop the top two items from the stack as floating point values, add them, and push the result to the stack.
`fdiv`	`110`		Pop the top two items from the stack as floating point values, divide them, and push the result to the stack. This should still return the divide by zero error as above.
`i2f`	`134`		Int to float: pop the top item from the stack, convert it to a floating point, then push the result to the stack
`f2i`	`139`		float to int: pop the top item from the stack as a floating point value, convert it to an int, then push the result to the stack
`fprint`	`186`		This is not a real Java bytecode – in real bytecode, code 186 is used for the `invokedynamic` instruction. You will implement the fprint byte code to print the top item on the stack as a floating point number (without popping it off the stack)

Figure 10: Bytecodes to Implement

Once you have implemented all of the above function, you must then write a small program, using the java byte code mnemonics, to show the correct functionality of each of byte code, in a file called fp.minijava.

You can compile your `fp.minijava` file using the `minijavac` compiler provided on the Assignment 5 web page.

Note: There will be no bonus marks given for implementing *only* the instructions in this section. You must implement **all** of the instructions **and** `fp.minijava` to receive bonus marks for this part.

9 Submission Instructions

1. Create a directory `asn5` in your Git repository – **not** `Asn5`, `Assign5`, etc.
2. In this directory, the following files should exist (in exactly the case and spelling shown):
   - `asn5.m`
   - `execute_codes.m`
   - `fp.minijava` (if you did Bonus)

   If you do the bonus, make sure you hand in your `minijava` file (the source code files) and not your `miniclass` files.
3. Do not put code in sub-directories of `asn5` – please put all the files listed above in the `asn5` directory.
4. We use automated scripts to assemble your programs and they will be expecting the filenames and directory structure discussed above. Failing to adhere to the requirements dictated here will prevent us from automatically testing your programs, and will result in a deduction of a few marks.
5. When you have committed your code and are ready to submit it, tag it with the Git tag `asn5` – **not** `Asn5`, `Assign5`, etc.

IMPORTANT: Failing to tag your commit properly is tantamount to not submitting your assignment. Do not forget to tag the commit that represents your assignment submission with the tag `asn5`. Otherwise, you will not have submitted your assignment – even if you pushed your code to GitHub. We have no way of knowing which of your commits represents your assignment submission unless you tag your "submission commit" properly.