Advanced Programming languages
Programming Project 4

A Virtual Machine Project

The goals of this project are:

- to improve the structure and functionality of a virtual machine program we have discussed in class
- to extend the instruction set of the virtual machine including to execute native code
- to develop two programs that utilize the enhanced capability.

1. Improving the structure and functionality

Download the Program by clicking this link

The program as written is a poor exemplar of an object-oriented program. The program contains many redundant, static variables. Additionally, several example programs are hard-coded into the tester class. The first goal of this assignment is to make the program less static, to create a file structure for programs, and to modify the program to choose the file to be run through a JFileChooser instance. The file structure should be as follows:

```
<globalHeaderComment>    // Author, date, etc., and an annotated version of the program
~                        // tilde delimiting the code comment at the top
<numOpCodesAndOperands>  // number of ints for the opcodes and operands in the program
<opcodesAndOperands>     // integer op codes and operands
<numFuncs>               // number of functions in the program - for metadata
repeat <numFuncs> times
  <String>               // function name
  <integer> <integer> <integer> // numArgs, numLocals and address of function
<literalcount>           // number of string literals that appear next
<StringLiterals>
```

We will extend the VM to include an additional operator and to execute native code.

2. Adding to the Instruction Set

Recall that we currently have 18 opcodes in our VM:

```
    final short IADD = 1;      // int add
    final short ISUB = 2;
    final short IMUL = 3;
    final short ILT  = 4;      // int less than
    final short IEQ  = 5;      // int equal
    final short BR   = 6;      // branch
    final short BRT  = 7;      // branch if true
    final short BRF  = 8;      // branch if false
    final short ICONST = 9;    // push constant integer
    final short LOAD   = 10;   // load from local context
    final short GLOAD  = 11;   // load from global memory
    final short STORE  = 12;   // store in local context
    final short GSTORE = 13;   // store in global memory
    final short PRINT  = 14;   // print stack top
    final short POP  = 15;     // throw away top of stack
    final short CALL = 16;
    final short RET  = 17;     // return with/without value
    final short HALT = 18;
```

We should add a MOD command (returns the remainder in integer division) and an EXEC command which will execute native code. Adding MOD should be simple. In order to execute native code, we will use the string literal pool at the end of the input file containing the program. The operand of the EXEC command will be an index into the String literal pool that indicates the location of the command that is to be executed. In order to maintain backward compatibility with previously written programs, we should not change any of the commands above, but rather add the new commands onto the end.

3. Executing native code:

We can execute a Java system call and get its output. This is an extremely useful general capability, allowing interaction with the operating system at the application level. The following program will run as-is in the Linux environment. You can also execute executable files such as shell scripts and binaries. Your extensions to the virtual machine should be able to handle both.

```
import java.io.*;
```

```
public class JavaRunCommand {

    public static void main(String args[]) {

        String s = null;

        try {

        // run the Unix "ls -l" command
            // using the Runtime exec method. NOTE: Windows requires more work as demonstrated in class
            Process p = Runtime.getRuntime().exec("ls -l");

            BufferedReader stdInput = new BufferedReader(new
                 InputStreamReader(p.getInputStream()));

            BufferedReader stdError = new BufferedReader(new
                 InputStreamReader(p.getErrorStream()));

            // read the output from the command
            System.out.println("Here is the standard output of the command:\n");
            while ((s = stdInput.readLine()) != null) {
                System.out.println(s);
            }

            // read any errors from the attempted command
            System.out.println("Here is the standard error of the command (if any):\n");
            while ((s = stdError.readLine()) != null) {
                System.out.println(s);
            }

            System.exit(0);
        }
        catch (IOException e) {
            System.out.println("exception happened - here's what I know: ");
            e.printStackTrace();
            System.exit(-1);
        }
    }
}
```

Note that the above program captures all of the output of the program that has been executed. Your program should capture the output and the bytecodes must know how to handle the return value.

4. Developing and Running GCD two ways:

You should write GCD in our bytecodes, including a main function and a separate function for gcd. Name the file `GCDFunc.txt`. This version will be interpreted by the VM. Note that the `factorial()` example is the template for passing parameters to a function and getting back the result. Additionally, you should write GCD in C, compile it, and then `EXEC` it from the VM, passing two parameters as command line arguments. This IR program should be named `GCDEXEC.txt`. Set up BOTH versions of the GCD programs (the one the virtual machine executes and the C program) to compute the gcd of 1983 and 1530. The answer should be 3. For the external call, your program should path to:

```
c:\\test\\<yourFolderName>\\gcd.exe 1983 1530
```

Your compiled program should be there (and named gcd.exe). Note the command-line parameters in the literal. ON Windows, the escape sequence IS needed (the "\\" in the path).

Program Execution
When your program runs, it will pop up a `JFileChooser` to select the input bytecode file. The JFileChooser should include a statement to make it look in the current directory, instead of in `MyDocuments`:

```
    fileChooser.setCurrentDirectory(new File("."));
```

Other Policies
All projects will be individual.

Deliverables:

You will submit all your work bundled up in WinZip (on PC) to the DropBox for the course on `elearning.uwf.edu`. Please make your name is very clear on the zipfile you upload and inside all files contained within the zipfile. Place all your files in a single directory named `<firstInitial><lastname>-p<projNumber>`. For instance, my submission would be `jcoffey-p4`. zip that folder into a .zip file and upload it to the appropriate dropbox.

So, submit a single zip file containing a single folder, with the following items in the folder:

- Source code for the entire program (all the Java files)
- A User's manual regarding exactly what this submission does and how to run it
- Your `GCDEXEC.txt` and `GCDFunc.txt`.
- Your gcd.c for the EXEC, including a binary of the executable version, in the correct location to path to it as specified above.