ASSIGNMENT 7: PROGRAMMING THE MIC1
due Friday, November 15.

OVERVIEW

In this assigment, we will use the Mic-1 simulation tools develloped by Ray Ontko. We will first learn how to drive the simulation tools around; then we will use them to implement new instructions on the Mic-1.

INTRODUCTION

The Mic-1 tools are written in Java, so you'll be able to make use of the JDK installation you did earlier in the semester.

Download the tools from the course web site (use the Software link) or directly from Ray's site, `http://www.ontko.com/mic1/`. There is a distribution for Windows-based PCs and Unix machines.

There are a bunch of different files that are used. Here is a summary:

Suffix	Meaning
`.jas`	IJVM assembly source file. IJVM is the ISA (instruction set architecture) that we are implementing on the Mic-1. This is at the same level as 6811 assembly code. Sample instructions are BIPUSH, DUP, IADD, etc.
`.ijvm`	IJVM binary file. When you run the `.jas` program through the `ijvmasm` assembler, you get an `.ijvm` file. This is the "executable" for the Mic-1 simulator.
`.conf`	Configuration file for the `ijvmasm` assembler. This file contains a listing of all of the IJVM instructions, plus their corresponding anchor locations in the Mic-1 control store (i.e., their op-codes).
`.mal`	Micro-assembly language sourc e file. These are the microinstructions that define the Mic-1 and thereby implement the IJVM instruction set.
`.mic1`	Compiled control store that gets loaded into the Mic-1. When you run the `.mal` source through the `mic1asm` compiler, you get the `.mic1` file.

There are several separate Java programs you will use. Here is a summary:

Name	Purpose
`ijvmasm`	Assembles IJVM assembly source (`.jas` file) into binary form (`.ijvm` file) for loading into the Mic-1 simulator.
`mic1asm`	Compiles Mic-1 specification (`.mal` file) into binary control store form (`.mic1` file).
`mic1sim`	The Mic-1 simulator itself. This takes as an argument the Mic-1 control store file (e.g., `mic1ijvm.mic1`)

GETTING STARTED: PRINTING AN ASTERISK

Ray has helpfully implemented two new IJVM instructions not mentioned in the book: IN and OUT. These accept keyboard input (IN) and print ASCII characters to an output window in the

simulator (OUT). IN reads a keypress and pushes its ASCII value on the stack (or zero, if no key was pressed). OUT pops a word from the stack and displays it as an ASCII character.

To begin, we'll write a trivial IJVM program that loads 0x2a onto the stack (the asterisk character), outputs it to the screen, and then loops endlessly to terminate.

Copy the following code into a file named `asterisk.jas`:

```
.main

L1:     BIPUSH 0x2a     // asterisk
        OUT             // display it

DONE:   GOTO DONE       // loop to end
```

Now, assemble your code into a `.ijvm` file with the following command:

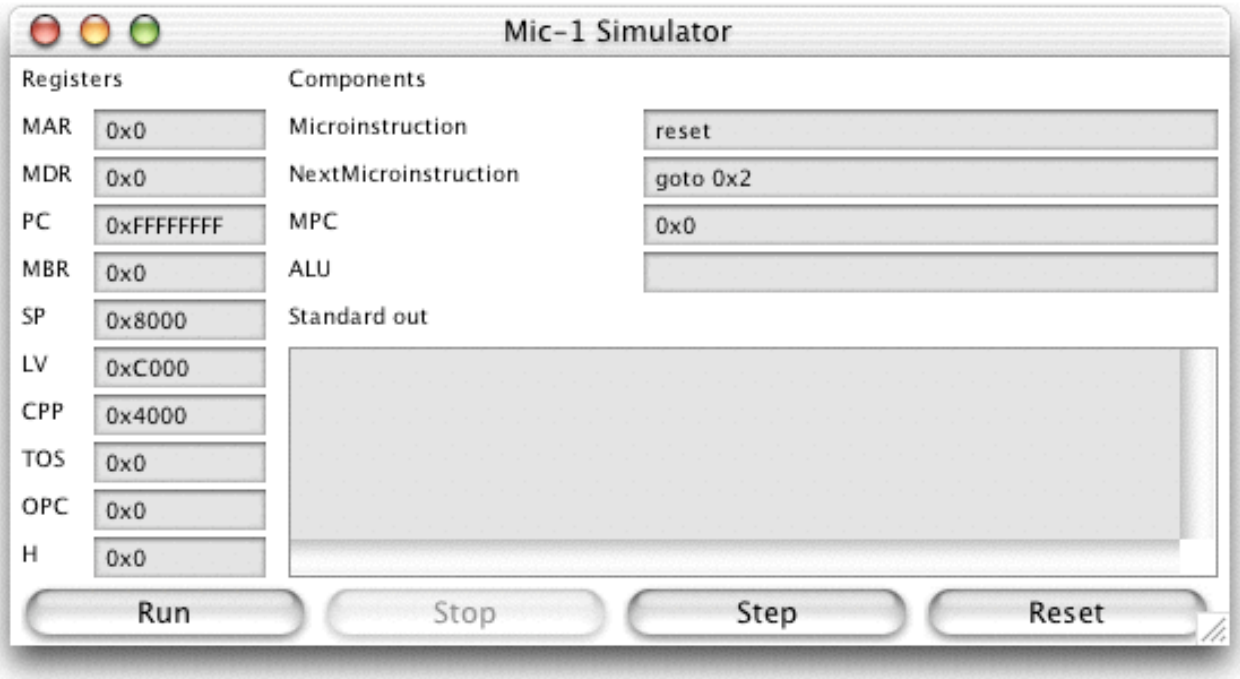
```
java ijvmasm asterisk.jas asterisk.ijvm
```

Note: **You will have to set the CLASSPATH environment variable** before you can get this to work. See Ray's documentation `user_guide.html` for details.

Once you have the `asterisk.ijvm` file, you can run the Mic-1 simulator. Type:

```
java mic1sim mic1ijvm.mic1 asterisk.ijvm
```

This should bring up the graphical simulator as shown below:



Click "Run," and you should see a single asterisk (*) in the Standard out area. The MPC value will flip around as the machine continually executes the DONE: GOTO DONE instruction.

Problem 1A.

Re-write the `asterisk.jas` program to print three asterisks and then terminate with the endless loop. *Turn in the code of your `3asterisk.jas` program.*

Problem 1B.

If necessary, modify your 3-asterisk version to accomplish the task with no more than 10 bytes of object code in the resultant `.ijvm` file (including the DONE: GOTO DONE jump).

Hints:

- You can display the bytes in the assembled `.ijvm` file with the dump utility that's provided with Ray's package: `java dump asterisk.ijvm`. The object code starts at location 20 in the output (byte 0x10 is the BIPUSH instruction). Also, location 19 tells you how big the object code is.
- Page 222 of your book is a table of the IJVM instruction set.
- Look at Ray's sample `.jas` programs for the trick that will let you solve this.

Turn in the code of your new `3asterisk.jas` program.

PROBLEM 2:
ADDING AN INSTRUCTION TO THE IJVM ISA
BY MODIFYING THE MIC-1 MICROPROGRAM.

(If you understand the problem statement, you're halfway there.)

In this exercise, we will extend the capabilities of the Mic-1 machine by modifying the definition of its control store to include a new instruction. Then we will add the details for this instruction into the definition file for the `ijvmasm` assembler. Finally we'll write some `.jas` code that exercises the new instruction, and run it in the simulator to make sure it works.

The instruction we will add is COM, for complement. The instruction will calculate the one's complement of the word on the stack and push it back on the stack.

The microcode for this is fairly straightforward. The point of this exercise is to walk through all the steps, to set you up for doing some more interesting microprograms.

Here's the code for COM:

```
com1    MDR = TOS = NOT TOS       // calc 1s compl, save in TOS and MDR
com2    MAR = SP; wr; goto Main1  // set MAR from SP, write, recycle
```

The top-of-stack value is already present in the machine, in the TOS register (by definition of how the TOS register works). We calculate its one's complement with the expression `NOT TOS`. This gets stuffed into both the MDR register (for writing to the RAM) and the TOS register (since it's now the new top-of-stack). In a second cycle, we set up the MAR from the SP, for writing the new TOS value out to memory. In this same cycle, we instruct the memory system to write the new value to RAM (with `wr`); then we `goto Main1` to execute the next opcode.

This line has to be installed in the `mic1ijvm.mal` file, which is the source definition file for the Mic-1 control store. Please **start out by making a new directory for work on this problem, and then copy the supplied mic1ijvm.mal file into it.** If you are working on a Unix machine, you'll

have to give yourself write permissions for the new `mic1ijvm.mal` file.

The com1 definition line can go anywhere after a line that ends with a specific goto. I put it after the ior3 definition.

In addition, we need to choose an opcode for the COM instruction. This goes both in the `mic1ijvm.mal` file and also in the assembler definition file `ijvm.conf` . At the beginning section of the `mic1ijvm.mal` file, the opcodes are defined in numerical order. Choose an unused opcode for your new COM instruction, and define it. (I choose the opcode 0x19.)

The opcode also needs to be installed in the `ijvm.conf` file. In this file, the IJVM instructions are listed in alphabetical order. After the line that defines BIPUSH, add:

```
0x19    COM             // Complement top word on stack
```

Now, you need to run the tool that compiles the `mic1ijvm.mal` microprogram definition file into its binary control store form. Do this with:

```
java mic1asm mic1ijvm.mal mic1ijvm.mic1
```

Your new Mic-1 machine should be ready to run. You just need an IJVM program to test it.

Think about a simple way to test that the new COM instruction works, and write a `.jas` program to demonstrate. Assemble and run the resulting `.ijvm` file with your new Mic-1 simulator. Does it work?

To turn in:
a) A copy of the program you used to demonstrate that the COM instruction worked, a short discussion of the output you expect from your test program, and screensnap of the simulator run showing that result.
b) Answer the following: Why is it not necessary to modify the stack pointer register (SP) to accomplish COM instruction?

PROBLEM 3: IMPLEMENTING A BIT-SHIFT INSTRUCTION.

The Mic-1 hardware includes a bit-shifter unit after the ALU. The shifter has the capability to perform a logical shift left by 8 bits (filling from the right with zeroes). This is specified with the notation "<< 8"; for example, see the `wide_istore2` line in the `mic1ijvm.mal` file. The shifter can also perform an arithmetic shift right by one bit. The notation ">> 1" specifies this.

Create an instruction, ASR, that takes a one-byte argument indicating how many bits to shift the operand. The operand is the top-of-stack word.

In the `ijvm.conf` file, define the ASR instruction as:

```
0x19    ASR byte        // Arithmetic Shift Right TOS word
```

Here, I used 0x19 as the opcode byte; you can use whatever you like.

To turn in:
a) The lines you added to `mic1ijvm.mal` that implement the ASR instruction.
b) A copy of the program you used to test that the ASR instruction, a short discussion of the

output you expect from your test program, and screensnap of the simulator run showing that result.

PROBLEM 4: IMPLEMENTING A 'STACK-UNDER' INSTRUCTION.

The Forth programming language doesn't have an LV (local variable) pointer structure to index arguments to functions. Instead, it provides stack operations that let you peek down into the stack and copy words arbitrarily deep in the stack as new stack pushes.

For example, UNDER 1 would locate the word one deep in the stack and push it as the new top-of-stack. That word is still on the stack, but afterward it would be two deep. UNDER 0 is equivalent to DUP (which already exists in the IJVM ISA).

Implement this new UNDER instruction. It takes an immediate byte argument, which is the unsigned 0–255 count of how many levels deep the word to be extracted lives.

When you write the test/demonstration code, make sure that you demonstrate correct functionality of (at least) UNDER 0 and UNDER 3.

To turn in:
a) The lines you added to `mic1ijvm.mal` that implement the UNDER instruction.
b) A copy of the program you used to test that the UNDER instruction, a short discussion of the output you expect from your test program, and screensnap of the simulator run showing that result. *Make sure that you have demonstrated UNDER 0 and also UNDER 3.*

HONORS/EXTRA CREDIT
PROBLEM 5: MICROCODED MULTIPLY.

It was suggested in class that it might be interesting to implement a multiply instruction on the Mic-1. Suppose it were to operate like IADD, ISUB, etc.—it would pull two words off the stack, multiply them, and push the result.

Let's simplify a bit, and presume that the operation is only required to do a 16x16 bit multiply (using the lower halves of the two 32-bit argument words).

Is this feasible? If yes, sketch out a solution. If no, provide an argument as to why, or how the Mic-1 architecture would need to be extended to make the multiply operation possible.

In terms of speed, how would a microcoded multiply compare to the equivalent multiply implemented as a subroutine in the IJVM ISA?