SIMPSIM – A SIMPLIFIED STACK BASED MICROPROCESSOR SIMULATOR

You will be building a CPU simulator in Java of a fairly simple CPU. **SIMPSIM** simulates a stack-based machine, also known as a zero-address computer. The opcodes for a stack-based machine do not have any arguments per se, they process items on a stack. For example, the ADD instruction adds together the top two items on the stack and then pushes the result back onto the stack. Stack based machines were popular in the early days of computing but are limited in terms of building efficient high-speed machines in a modern context. Nonetheless, they have some appealing characteristics in an educational setting that will allow you to explore some ideas regarding CPU simulation more deeply. In terms of building a software simulator, they are not significantly less efficient than register based machines and many VM architectures are, at least in a limited sense, stack based. **SIMPSIM** is based on Bruce Land's (Cornell) Pancake machine which is similar to a machine by Nakano et al. If you're interested in reading more about stack machines the book by Koopman, referenced below, is freely downloadable.

OVERVIEW

The **SIMPSIM** simulator takes in a simplified text-based object file and simulates the execution of a CPU. The object file must conform to the same file format that Logisim takes as a memory image file, and simply contains a list of numbers in hexadecimal format. There is a one line header which reads "v2.0 raw", and after that each line will contain a single hexadecimal number that matches the bit width of the memory cell.

Make sure that you understand this idea. You are not feeding assembly language mnemonics into the computer to simulate, rather, you are feeding in the machine language op-codes that results from the assembly of those mnemonics.

For this project we will be a using a 4k X 16-bit memory width so each line of your input file will contain a single four hex digit, sixteen-bit number. Hence, data values are 16-bit and address values are 12-bit. The stack is a separate 4k memory that, of course, holds 16-bit values and has a depth of 4096, or 4k entries. Note, there are some requirements of tracking used stack depth discussed below.

In total then, your CPU has 8k x 16-bit memory cells. 4k are organized as program memory and this is where you will initially load your instructions and this is also the memory that your program counter will address as it executes each instruction. The other 4k is your stack memory and this is organized as a stack. You have no ability to address this memory directly, you may only PUSH elements to the top of the stack or POP elements from the top of the stack. Note that you can move data to and from program memory to the stack via the PUSH and POP commands. In this sense, the program memory is also a data memory because we can use it to store the final results of operations computed on the stack.

This program's task is simple: to simulate the execution of a single program in memory, one instruction at a time. The simulator, will first load the instructions from the program into a simulated program memory *(this is not the stack)*. Then, it will start the main loop that simulates the CPU: fetch, decode, execute. It will continue to do this until it reaches the special instruction HALT, that we will use to indicate the program should be finished. To simulate a CPU, you'll have to keep track of stack and memory state and a few other things, for example, the program pointer register. Each instruction just updates some subset of the machine state. Because the simulated computer is rather simple, you won't have to spend too much effort decoding the instructions. Each instruction is represented by a 16-bit opcode described below.

Often students struggle with the concepts of this assignment. Try to focus on what you are doing. Your Java code is simulating what each of the instructions would do in a real CPU. Often this requires that you think in terms very similar to the CPU machinery. However, most of this exercise is about *behavioral simulation*. What do we mean by this? We mean that you are simulating the behavior of the CPU, not a precise interpretation of the underlying digital logic. For example, when you add two numbers in java you simply use the '+' operator. Consequently, when you encounter an ADD instruction in the machine language input, you will want to use Java's '+' operator to add the two items on the top of the stack. In order to do this, you will need to retrieve the two items from the top of the stack. You should know how to do this because you should have discussed how to implement stacks in your data structures course. Yes, there are some gotchas, but your goal is to take a machine language program defined in a text file, and simulate the execution of that program by simulating the *behavior* of each instruction using Java language features.

CPU INSTRUCTION SET

The instructions listed below define a simplified stack-based CPU. Many standard operations have not been included here in order to keep your simulator reasonably minimal. That said, this simple simulator is capable of interesting computation. With this set of instructions, we can support small high-level languages reasonably easily. Many of the operations refer to *top* and *next*. These are the top value of the stack and the entry just below the top of the stack. An operation such as ADD will POP the two top elements off of the stack, compute the sum and the PUSH the result back onto the stack.

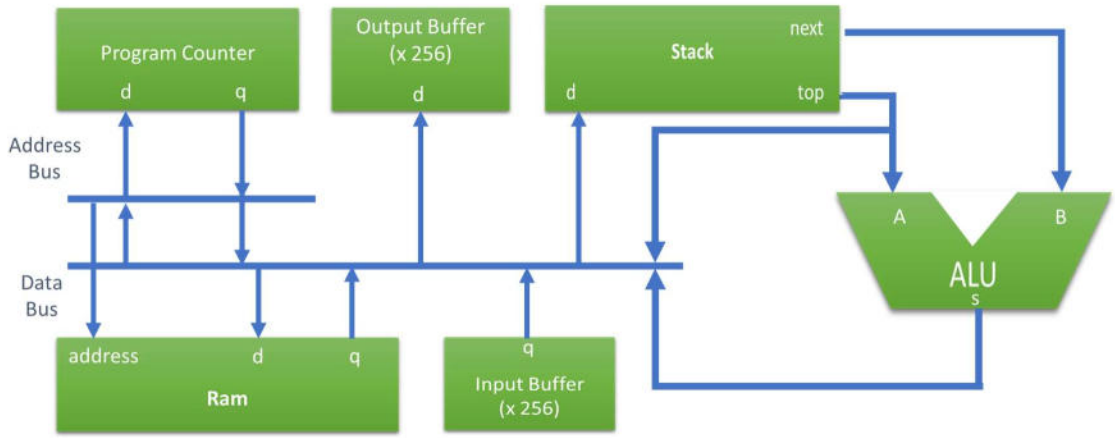
For this exercise, unless specified otherwise, all numbers are 16-bit signed integers represented in 2's complement format. This is important and it behooves you to choose the correct data type to represent them in your code. Note that the immediate values, shown as I in the table, are 12 bit 2's complement signed integers and will need to be sign extended to 16 bits in your code. Finally, the address values, shown as A in the table, are 12-bit unsigned integers. Finally, the port values, shown as P in the table, are 8-bit unsigned values. This means that this CPU can access 256 input and output ports. There are some special considerations for your simulator with respect to this detailed below.

SIMPSIM INSTRUCTION SET TABLE

Instruction Mnemonic	Machine Code in Hexadecimal	Action (Microcode)	Comments
NOP	0000	Do Nothing	
HALT	0F00	Stop CPU Execution	
PUSHI I	1000 + I	I → top	
PUSH A	2000 + A	mem[A] → top	
POP A	3000 + A	top →mem[A]	
JMP A	4000 + A	A → pc	
JZ A	5000 + A	A → pc if top == 0	
JNZ A	6000 + A	A → pc if top != 0	
IN	D000 + P	in[port P] → top	
OUT	E000 + P	top → out[port P]	
ADD	F000	next + top → top	
SUB	F001	next – top → top	
MUL	F002	next * top → top	
MOD	F003	next % top → top	Integer modulus
SHL	F004	next << top → top	
SHR	F005	next >> top → top	
BAND	F00A	next & top → top	
BOR	F00B	next | top → top	
BXOR	F00C	next ^ top → top	
EQ	F010	next == top → top	
NE	F011	next != top → top	
GE	F012	next >= top → top	
LE	F013	next <= top → top	
GT	F014	next > top → top	
LT	F015	next < top → top	
NEG	F006	-top → top	Unary minus
BNOT	F00D	~top → top	Bitwise inversion

DATAPATH

This is the basic datapath for the CPU. This is provided so that you can see at a glance how this should work. Keep in mind that in a software *behavioral* simulation that each of these elements is somewhat abstract. For example, there are no wires for the data bus and address bus, however, you should keep the distinction in mind when modeling your memory. Note also that the stack does not have any address input. This should tell you that the address locations of the stack are inaccessible to the computing machine, hence, they are inaccessible to your simulator.



STACK BASED COMPUTATION

In a stack-based instruction set architecture, all computation moves through the stack. For example, suppose that we wanted to add two numbers together, we could execute the following code:

```
PUSHI 225     ; push 225 onto the stack
PUSHI 15      ; push 15 onto the stack
ADD           ; add 225 to 15, push result onto stack
POP k         ; store result at location k
HALT          ; halt the simulator
k: 0          ; define location k and initialize to 0
```

With this small program our CPU would place two numbers on the stack. The ADD instruction would then operate on the two numbers that are on top of the stack by adding them together and then returning the sum to the top of the stack. Since you have no way to access the memory locations of the stack, it should be clear to you that the ADD instruction must pop the two data elements from the stack, add them together, and then push the elements back onto the stack. Do not get confused here. We are not talking about the PUSH and POP instructions which instruct the CPU to push and pop elements from and to memory locations respectively. We are talking about the intrinsic operations of a stack. In a *behavioral* simulation, we do not have to think about the digital logic that makes a stack work correctly. We will talk about this later in the course. We are simply using Java's ability to work with a stack of a specific data type. Now is a great time to practice creating a Java class using Java Generics for your Stack. You may use Java's Stack class from the Collections library; however, I will caution you that it is not ideal. A good object-oriented simulation should attempt to model the *behavioral* limitations of the underlying system. For maximum points, write your own Stack class. You may use any underlying Java Collections data structure as your backing store. You should have heard language like this in your data structures course, if It is not familiar to you then feel free to ask questions in the discussion forums.

This series of instructions listed above would ***assemble*** to the following opcodes using the table provided above. This machine is simple enough that you can hand assemble fairly easily. Hand assembly is the process of manually converting each assembly instruction into its machine language representation. For each line of the assembly listing above, look up the *opcode* in the table given above and translate this code into a machine language instruction by following the rules given for computing the machine language value.

ASSEMBLED CODE

```
*** LABEL LIST ***
k       005

*** MACHINE PROGRAM ***
000:10E1        PUSHI 225
001:100F        PUSHI 15
002:F000        ADD
003:3005        POP k
004:0F00        HALT
005:0000        k: 0
```

You would then want these instructions one word, in hexadecimal, per line in the ***object*** file listed below. Note, an object file format does not have to be a binary file. In our case it will be a simple text file containing a header followed by the list of 16-bit hexadecimal words given above. Your program must read in this format. The grading scripts will assume that you can read this format without error. It behooves you to put some effort into doing adequate testing of this code. Often students lose points because they make poor assumptions and submit unreliable code. I suggest that you save this file with an **.out** extension. Hence, file containing the machine instructions below might be called ***example1.out***.

```
-------------------------snip--------------------------
v2.0 raw
10E1
100F
F000
3005
0F00
0000
-------------------------snip--------------------------
```

After executing these instructions, the result, 240, will still be stored in memory location k and the simulator will stop running. For this program your program should output nothing as there are no output operations defined. You may limit the run of your simulator to 10,000 machine cycles. You may find this helpful for debugging. If you choose to do this then you must print an error that indicates that the simulator exceeded 10,000 cycles whenever it does so.

HOW TO SIMULATE **SIMPSIM**: INSTRUCTIONS

How should a simulation work? A simulator models the behavior of the CPU. The first thing your simulator will do is to fetch and then decode the instruction that starts at address zero. Once you have decoded the instruction, you can simulate its execution. For example, if the instruction was PUSHI 5, you would move the value 5 that's embedded into the 16-bit opcode onto the stack. You then move the program pointer (sometimes called program counter) ahead by one to fetch the next instruction and repeat. Think about how each instruction works and then manipulate the state of the machine accordingly. Use Java's high-level operations whenever appropriate and always think about the format of the input data. Is the data an unsigned integer, a signed integer, a byte? Is the data 16 bits, 12 bits, 8 bits, something else? A good solution to this problem will think carefully about the data.

Don't lose sight of the following: The data input for your Simpsim.java program written in the Java language is another program written in SIMPSIM machine language. This machine language program is passed in as a parameter on the command line. This is no different than executing Java. The data input for the Java compiler is a program that is written in the Java language and passed in as a parameter on the command line. None of the machine language examples above will appear anywhere inside your java program. These are always separate files that are passed in for your simulator to interpret.

If these ideas are confusing to you, then now is a really good time to review the prep material on the differences between assembler and machine language and what an assembler does. Feel free to post questions in the discussion forums.

COMMAND LINE INPUTS AND OUTPUT FORMAT

Your simulator will run from the command line using the Java interpreter. Your program must be saved in a single source file called Simpsim.java. You may include multiple non-public classes in your file, but ***your file must run with Java 11's single source execution***. You would then run it from the command line as follows:

```
Your command line prompt> java Simpsim.java [options] <object file> [integer in value]
```

Your simulator will execute the code and output integers, one per line, from the OUT instruction as described below. You are free to add additional options beyond those listed below to help you debug. At minimum I suggest at least a partial memory dump and maybe a disassembly line that gives you all information for each instruction being processed. It is important that when no options are specified other than what are described above, that your program prints only the OUT values described below. When no parameters are specified, then the program must print out usage information, e.g.:

```
Your command line prompt> java Simpsim.java
USAGE:  java Simpsim.java [options] <object file> [integer in value]
```

You must provide an option to see the maximum stack depth used. It's an interesting parameter to see for moderately complex programs. This option will be specified as **-s.** You may use

Hence, the following execution.

```
Your command line prompt> java Simpsim.java -s example1.out
```

Will run the above example which does not print any data output. However, your simulator must then print the following:

```
Maximum Stack Depth: 2
```

This indicates that the example program above used a maximum of two stack locations in its execution.

You should be familiar with standard notation for command line program specifications. That is, you should know that square brackets represent an optional component. So, in the above example we specified an option **-s**, but such options don't have to be specified because the options specification is in square brackets. Similarly, the above example did not specify an integer input value, this is because that is also optional. Finally, the specification for object file is enclosed in angle brackets. This indicates some variable input to the program. In this case, you replace that phrase in your command with the name of the object file.

Note that the option for stack depth is preceded by a hyphen character. This is a standard way to provide options and you must use this format. You may not allow any other character and you may not use any option processing libraries. Any additional options that you provide must work in the same way. This gives you an easy way to test to see if an argument is an option. I consider all of this notation to be review for senior students and if it is not for you then now is a great time to learn this industry standard notation.

IN/OUT INSTRUCTION

Simpsim includes IN and OUT instructions to gather input and provide output. The IN and OUT parameters operate on the stack and have a port embedded in their instruction format. For this exercise, we are only going to use port 0, which need not be specified as the default opcode will use port 0. You can be certain that any code that I pass to your simulator will automatically use port 0 so it is not necessary to do error checking on the value at this time. This is meant to simplify this aspect of the simulator for this assignment.

When an OUT instruction is encountered, the CPU pops the value off of the top of the stack to output it to the port. Your simulator will print this single integer to the screen followed by a carriage return. During normal program execution your program will print nothing else. You must support a single optional IN value on the command line. This is a signed 16-bit integer. Whenever the CPU encounters an IN instruction for port 0, the simulator will push this value onto the stack. This allows you to specify a single integer on the command line to be used as a run time parameter.

TESTING REQUIREMENTS

The best way to test your simulator is to write several small assembly language programs and execute them with your simulator. At the moment, you will have to hand assemble these programs. It is relatively easy to write an assembler for such a CPU. If you are feeling adventurous, that's a useful activity, however, it's not required. You must run the test code that will be provided to you and your output must match exactly. This will not be the only input file that I use to grade your assignment and I suggest that you write enough assembler to test all of the instructions in multiple ways.

Writing your own test inputs is an important skill for real world programming. You will not have an instructor on the job who will give you everything that you need to make sure that your code works. If your code doesn't work then your customer may drop you or your boss may fire you. It's your responsibility to think about testing the code that you write and that includes using your imagination to think of all the possible ways that it might fail.

SUBMISSION REQUIREMENTS

You must submit your Simpsim.java file as a Java 11 single course executable. It must run from the command line without explicit compilation as shown above. You must also provide a short writeup that describes the options and demonstrates their execution. In your writeup you must show the execution of the example code that will be provided to you in the assignment and the example code above from this assignment. For maximum points you should also show the execution of at least one, no more than two other assembly language programs that you have written and assembled. Do not overthink this, a simple example would be to print the first k Fibonacci numbers for some fixed k. A better example would be to print the first k Fibonacci numbers where k is an input parameter presented on the command line.

Your writeup should be neatly formatted and should give the grader a quick overview of your project as well as to demonstrate the above execution of example programs.

For maximum points, put some effort into capturing your screen output. Dark backgrounds look terrible and it is trivial to change it so that your console background either matches the document or is a light color, such as light gray, and looks good against the white document background. Similarly, it's trivial to reasonably match font size and be consistent across your examples. Don't say that you haven't been warned. My neatness standards are high for maximum points.

REFERENCES

Nakano, K.; Ito, Y., *Processor, Assembler, and Compiler Design Education Using an FPGA*, Parallel and Distributed Systems, 2008. ICPADS '08. 14th IEEE International Conference on; 8-10 Dec. 2008 pages: 723 - 728 (Nakano, K.; Ito, Y.; Dept. of Inf. Eng., Hiroshima Univ., Higashi-Hiroshima, Japan)

Nakano, K.; Kawakami, K.; Shigemoto, K.; Kamada, Y.; Ito, Y. *A Tiny Processing System for Education and Small Embedded Systems on the FPGAs*, Embedded and Ubiquitous Computing, 2008. EUC '08. IEEE/IFIP International Conference, Dec. 2008 pages: 472 - 479

Philip, K., and Jr Koopman. "Stack Computers, the new Wave." (1989).
http://users.ece.cmu.edu/~koopman/stack_computers/index.html

Bruce Land's Pancake Machine *http://people.ece.cornell.edu/land/courses/ece5760/DE2/Stack_cpu.html*