STUMPED – A SMALL STACK BASED MICROPROCESSOR FOR UNDERGRAUDATE EDUATION

You will be building a CPU simulator in Java of a fairly simple CPU. **STumpEd** is 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. **STumpEd** 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 **STumpEd** 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 op-codes into the computer to simulate, rather, you are feeding in the machine language that results from the assembly of those op-codes.

For this project we will be a using a 4k X 16-bit memory width so each line will contain a single four hex digit, sixteen-bit number. Hence, data values are 16-bit and address values are 12-bit. The stack, of course, holds 16 bit values and, for now, is of arbitrary depths. We may revisit this idea in a later project. Note, there are some requirements of tracking used stack depth discussed below.

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 memory. 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.

CPU INSTRUCTION SET

The instructions listed below define a fairly comprehensive, yet simple stack-based CPU. 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. I will caution you that this is a non-trivial aspect of this exercise. 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.

Pay close attention to the difference between bitwise and logical operations. The logical operations are based on the C standard for interpretation of integers as values for *true* and *false* where false is represented by zero and true is any other value. You will want to make sure that your Java code does a proper translation of these values.

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
HLT           ; stop the simulator
```

This series of instructions would assemble to the following opcodes using the table provided above. This machine is simple enough that you can hand assemble fairly easily.

```
*** 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, and 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 hex words given above. Your program must read in this format.

```
-------------------------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 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.

HOW TO SIMULATE **STUMPED**: 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.

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 Stumped.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 Stumped.java <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 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. I also highly recommend that you provide an option to see the maximum stack depth used. It's an interesting parameter to see for moderately complex programs. 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 Stumped.java
USAGE:  java Stumped.java <object file> [integer in value]
```

IN/OUT INSTRUCTION

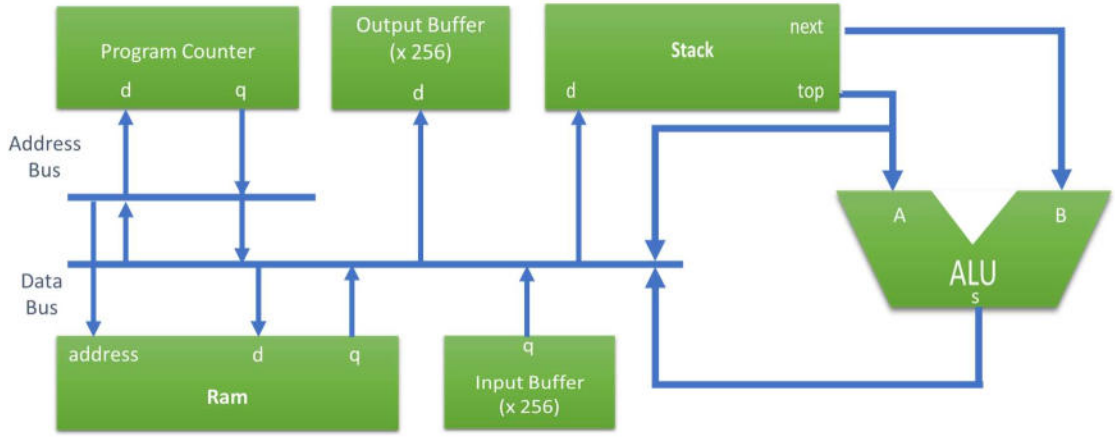
Stumped includes IN and OUT instructions to gather input and provide output. The IN and OUT parameters operate on the stack and have a port embedded in their instruction format. For this exercise, we are only going to use port 0, which need not be specified in your assembler as the default opcode will use port 0.

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 may provide 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.

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 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.



STUMPED INSTRUCTION SET TABLE

Instruction Mnemonic	Machine Code in Hexadecimal	Action (Microcode)	Comments
NOP	0000	Do Nothing	
HALT	0F00	Stop CPU Execution	
PUSHPC	0100	PC → top	Pay close attention to sign issues here
POPPC	0200	top → PC	
LD	0300	mem[top] → top	
ST	0400	top → mem[next]	Both are popped off of the stack
DUP	0500	Copies top of stack	Stack: top → top top
DROP	0600	Removes top of stack	Stack: next top → next Note: this is like pop but the top of the stack is discarded.
OVER	0700	Copies next to top	Stack: next top → next top next
DNEXT	0800	Drops next	Stack: next top → top
SWAP	0900	Swaps top and next	Stack: next top → top next
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	
DIV	F003	next / top → top	Integer division, division by zero halts CPU
MOD	F004	next % top → top	Integer modulus
SHL	F005	next << top → top	
SHR	F006	next >> top → top	
BAND	F007	next & top → top	
BOR	F008	next | top → top	
BXOR	F009	next ^ top → top	
AND	F00A	next && top → top	Logical operations, not bitwise. Non-zero implies true, zero is false
OR	F00B	next || top → top	
EQ	F00C	next == top → top	
NE	F00D	next != top → top	
GE	F00E	next >= top → top	
LE	F00F	next <= top → top	
GT	F010	next > top → top	
LT	F011	next < top → top	
NEG	F012	-top → top	Unary minus
BNOT	F013	~top → top	Bitwise inversion
NOT	F014	!top → top	Logical negation

SUBMISSION REQUIREMENTS

In addition to writing the simulator in .java, you are also required 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. In addition to building the simulator, you will also have to write the following assembler programs to run on Stumped.

1) Convert the example 8,2 in your book to the stack format, then hand assemble and run. Your code will also have to fill the array. Include startup *assembler* code that loads the first eight prime numbers into the first eight bytes of the array. Note that there is no concept of an array in the assembly code, labels are merely representative of an address. You may use a defined variable name to indicate the first word of the "array" and then use the provided instructions to figure out how to index subsequent elements. Note, this must be code that loads each prime as an immediate value and stores it in the correct memory location. You cannot just hardcode the primes where the array starts. Output each value of the array as you store it and then output the sum of the array when you are finished.

2) Write a number of short test files to test your simulator. There is no requirement for the number of test files, however, your test files much test each instruction at least once. Write a short report discussing any choices that you made. Include each test file and describe what it is testing. In the report include both your assembler code and your object file listings (as shown above). You will be uploading a .pdf of your report and a .java file of your simulator.

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*