Homework 1: Stack Machine

See Webcourses and the syllabus for due dates.

Purpose

In this homework you will form a team [Collaborate] and implement a word-addressable stack-based virtual machine [UseConcepts] [Build].

Directions

There are two parts to this homework:

1. (10 points) Email our TA, Mana Mostaani (mana.mostaani@Knights.ucf.edu) with:

   1. the email's subject being `"Our team for COP 3402"`.
   2. a list of the full names of all the team members, of which there may be either 2 or 3 total,
   3. the name of the person who will be the "team communicator"; this person will submit all assignments and be responsible for responding to emails from the course staff, and
   4. the name of a different person, who will be the "team facilitator"; this person will be responsible for making sure that all team members understand everything about the solution.

   Your team will jointly do all project implementations in the course.

   All team members must all be registered for the same lecture section (0001 for Dr. Leavens), and it is recommended that they all be in the same lab section as well.

   We will randomly ask questions of students in the team to ensure that all team members understand their solution; there will be penalty of up to 10 points (deducted from all team members' scores for that assignment) if some team member does not understand some part of the solution to an assignment.

2. (100 points) Implement and submit your VM code as described in the rest of this document.

   For the implementation, your code must be written in ANSI standard C and must compile with gcc and run correctly on Eustis. (See http://newton.i2lab.ucf.edu/wiki/Help:Eustis for information on how to access Eustis.) We recommend using the gcc flag `-Wall` and fixing all warnings before turning in this assignment.

What to Turn In

Your team must submit on Webcourses:

1. A plain text file named `sources.txt` that lists the names of all the `.c` files needed to compile your program, all on one line separated by spaces. For example, if you have files named `base.c`, `instruction.c`, and `vm.c`, then your file `sources.txt` would look contain (only) the following line of text naming these files:

```
base.c instruction.c vm.c
```

If there is only one file in your program, then put its name in your `sources.txt` file.

2. Each source file that is needed to compile your VM with `gcc` on Eustis, including all needed header files (if there are any).

3. The output of a test program running in the virtual machine. Please provide a copy of the initial state of the stack and the state of stack after the execution of each instruction. Please see the example in Appendix B.

We will take some points off for: code that does not work properly, duplicated code, code with extra unnecessary cases, or code that is excessively hard to follow. Avoid duplicating code by using helping functions, or library functions. It is a good idea to check your code for these problems before submitting. Don't hesitate to contact the staff if you are stuck at some point. Your code should compile properly; if it doesn't, then you probably should keep working on it. Email the staff with your code file if you need help getting it to compile or have trouble understanding error messages. If you don't have time to get your code to compile, at least tell us that you didn't get it to compile in your submission.

What to Read

You should read *Systems Software: Essential Concepts* (by Montagne) in which we recommend reading chapters 1-3.

Overview

In this assignment, you will implement a word-addressable stack-based virtual machine (VM).

The following subsections specify the interface between the Unix operating system (as on Eustis) and the VM as a program.

Inputs

The VM is passed a single file name as its only command line argument; this file should be the name of a (readable) text file containing the program that the VM should execute. For example, if the executable is named `vm` and the program it should run is contained in the file named `hw1-test1.txt` (and both these files are in the current directory), then the VM should execute the program in the file `hw1-test1.txt` by executing the following command in the Unix shell (e.g., at the command prompt on Eustis):

```
./vm hw1-test1.txt
```

When the program executes a CHI instruction to read a character, that character will be read from standard input (`stdin`). However, note that if you want the program to read a character, typing a single character (say `c`) into the terminal (i.e., to the shell) while the program is running will not send that character immediately to the program, as standard input is buffered by default. To send characters to the program it is best to use a pipe or file redirection in the Unix shell, for example, to send the two characters `c` and `d` to the VM running the program `progfile.txt` one could use the following command at the Unix shell:

```
echo cd | ./vm progfile.txt
```

Another command that would accomplish the same thing is to put the characters to be input into a file (using a text editor), say `cd-input.txt` and then to use the following Unix command.

```
./vm progfile.txt < cd-input.txt
```

Outputs

The VM prints its tracing output to standard output (`stdout`); furthermore, characters printed using the CHO instruction are also printed to standard output.

All error messages (e.g., for division by zero) should be sent to standard error output (`stderr`).

Exit Code

When the machine halts normally, it should exit with a zero error code (which indicates success on Unix). However, when the machine encounters an error it should halt and the program should stop with a non-zero exit code (which indicates failure on Unix).

1 VM Architecture

The VM you are to implement is a stack machine that conceptually has two memory stores: the "stack," which is organized as a LIFO queue of C int values and contains the data to be used by instruction evaluation, and the "code," which is organized as a list of instructions. The code list contains the instructions for the VM in order of execution.

1.1 Registers

The VM has a few built-in registers[1] used for its execution: The registers are named:

- base pointer (BP),
- stack pointer (SP), which points to the next location in the stack to allocate (i.e., one above the current top of the stack), and
- program counter (PC).

The use of these registers will be explained in detail below.

1.2 Instruction Format

The Instruction Set Architecture (ISA) of the VM has instructions that each have two components, which are integers (i.e., they have the C type int) named as follows:

OP	is the operation code
M	depending on the operator it indicates: either: (a) A number (when OP is LIT or INC), or (b) A program address (when OP is JMP, JPC, or CAL).

The list of instructions and details on their execution appears in Appendix A.

[1] What we call “registers” in this document are simply important concepts that simulate what would be registers in a hardware implementation of the virtual machine. In the VM as a C program, these would be implemented as variables.

1.3 VM Cycles

The VM instruction cycle conceptually does the following for each instruction:

1. Let *IR* be the instruction at the location that PC indicates. (Note that *IR* could be considered to be the contents of a register.)
2. The PC is made to point to the next instruction in the code list.
3. The instruction *IR* is executed using the "stack" memory. (This does not mean that the instruction is stored in the "stack.") The OP component of this instruction (*IR*.OP) indicates the operation to be executed. For example, if *IR*.OP encodes the instruction ADD, then the machine adds the top two elements of the stack, popping them off the stack in the process, and stores the result in the top of the stack (so in the end SP is one less than it was at the start). Note that arithmetic overflows and underflows happen as in C **int** arithmetic.

1.4 VM Initial/Default Values

When the VM starts execution, BP, SP, and PC are all 0. This means that execution starts with the "code" element 0. Similarly, the initial "stack" store values are all zero (0).

1.5 Size Limits

The following constants define the size limitations of the VM.

- `MAX_STACK_HEIGHT` is 2048
- `MAX_CODE_LENGTH` is 512

1.6 Invariants

The VM enforces the following invariants and will halt with an error message (written to stderr) if one of them is violated:

- $0 \leq \mathsf{BP} \ \wedge \ \mathsf{BP} \leq \mathsf{SP} \ \wedge \ 0 \leq \mathsf{SP} \ \wedge \ \mathsf{SP} < \texttt{MAX\_STACK\_HEIGHT}$
- $0 \leq \mathsf{PC} \ \wedge \ \mathsf{PC} < \texttt{MAX\_CODE\_LENGTH}$

A Appendix A

In the following tables, italicized names (such as p) are meta-variables that refer to integers. If an instruction's field is notated as $-$, then its value does not matter (we use 0 as a placeholder for such values in examples). Note that stack[$SP - 1$] is the top element of the stack.

A.1 Basic Instructions

OP Code Num.	OP Mnemonic	M	Comment (Explanation)
1	LIT	n	Literal push: stack[SP] $\leftarrow n$; SP $\leftarrow$ SP $+ 1$
2	RTN	–	Returns from a subroutine and restores the caller's AR: PC $\leftarrow$ stack[SP $- 1$]; BP $\leftarrow$ stack[SP $- 2$]; SP $\leftarrow$ SP $- 2$
3	CAL	p	Call the procedure at code index p, generating a new activation record and setting PC to p: stack[SP] $\leftarrow$ BP; // dynamic link stack[SP $+ 1$] $\leftarrow$ PC; // return address BP $\leftarrow$ SP; SP $\leftarrow$ SP $+ 2$; PC $\leftarrow p$;
4	POP	–	Pop the stack: SP $\leftarrow$ SP $- 1$;
5	PSI	–	Push the element at address stack[SP $- 1$] on top of the stack: stack[SP $- 1$] $\leftarrow$ stack[stack[SP $- 1$]]
6	PRM	o	Parameter at stack[BP $- o$] is pushed on the stack: stack[SP] $\leftarrow$ stack[BP $- o$]; SP $\leftarrow$ SP $+ 1$
7	STO	o	Store stack[SP $- 2$] into the stack at address stack[SP $- 1$] $+ o$ and pop the stack twice: stack[stack[SP $- 1$] $+ o$] $\leftarrow$ stack[SP $- 2$]; SP $\leftarrow$ SP $- 2$
8	INC	m	Allocate m locals on the stack: SP $\leftarrow$ SP $+ m$
9	JMP	–	Jump to the address in stack[$SP - 1$] and pop: PC $\leftarrow$ stack[SP $- 1$]; SP $\leftarrow$ SP $- 1$
10	JPC	a	Jump conditionally: if the value in stack[SP $- 1$] is **not** 0, then jump to a and pop the stack: **if** stack[SP $- 1$] $\neq 0$ **then** {PC $\leftarrow a$} ; SP $\leftarrow$ SP $- 1$
11	CHO	–	Output of the value in stack[SP $- 1$] to standard output as a character and pop: **putc**(stack[SP $- 1$], `stdout`); SP $\leftarrow$ SP $- 1$
12	CHI	–	Read an integer, as character value, from standard input and store it in the top of the stack, but on EOF or error, store -1: stack[SP] $\leftarrow$ **getc**(`stdin`); SP $\leftarrow$ SP $- 1$
13	HLT	–	Halt the program's execution
14	NDB	–	Stop printing debugging output

A.2 Arithmetic/Logical Instructions

For comparisons, note that 0 represents false and 1 represents true. That is, the result of a logical operation, such as $A > B$ is defined as 1 if the condition was met and 0 otherwise. Arithmetic is interpreted as **int** arithmetic as for C **int** values. Errors such as division by 0 (or modulo by 0) cause the VM to halt with an appropriate error message printed on stderr.

OP Codd	Number Mnemonic	M	Comment (Explanation)
15	NEG	–	Negate the value in the top of the stack: stack[SP − 1] ← −stack[SP − 1]
16	ADD	–	Add the top two elements in the stack: stack[SP − 2] ← stack[SP − 1] + stack[SP − 2]; SP ← SP − 1
17	SUB	–	Subtract the 2nd to top element from the top one: stack[SP − 2] ← stack[SP − 1] − stack[SP − 2]; SP ← SP − 1
18	MUL	–	Multiply the top two elements in the stack: stack[SP − 2] ← stack[SP − 1] × stack[SP − 2]; SP ← SP − 1
19	DIV	–	Divide the top element by the 2nd from top element: stack[SP − 2] ← stack[SP − 1]/stack[SP − 2]; SP ← SP − 1
20	MOD	–	Modulo, result is the remainder of the top by the 2nd from top element of the stack: stack[SP − 2] ← stack[SP − 1]modstack[SP − 2]; SP ← SP − 1
21	EQL	–	Are (the contents of) the top and 2nd from top element equal? stack[SP − 2] ← stack[SP − 1] = stack[SP − 2]; SP ← SP − 1
22	NEQ	–	Are (the contents of) the top and 2nd from top element different? stack[SP − 2] ← stack[SP − 1] ≠ stack[SP − 2]; SP ← SP − 1
23	LSS	–	Is (the contents of) the top element strictly less than the 2nd from top element? stack[SP − 2] ← stack[SP − 1] < stack[SP − 2]; SP ← SP − 1
24	LEQ	–	Is (the contents of) the top element no greater than the 2nd from top element? stack[SP − 2] ← stack[SP − 1] ≤ stack[SP − 2]; SP ← SP − 1
25	GTR	–	Is (the contents of) the top element strictly greater than the 2nd from top? stack[SP − 2] ← stack[SP − 1] > stack[SP − 2]; SP ← SP − 1
26	GEQ	–	Is (the contents of) the top element no less than the contents of the 2nd from top element? stack[SP − 2] ← stack[SP − 1] ≥ stack[SP − 2]; SP ← SP − 1
27	PSP	–	Push SP (i.e., the address itself) on top of the stack: stack[SP] ← SP; SP ← SP + 1

A.3 Examples

As an example, consider the instruction `ADD 0`, which is input as the line `15 0`, where SP is 10, so this means to place in stack[8] the sum of the values in stack[8] and stack[9], and then setting SP to 9.

As another example: if we have instruction `LIT 9`, which is input as the line `1 9`, then this means to push the integer 9 on the top of the stack: stack[SP] ← 9; SP ← SP + 1.

B Appendix B: Examples

B.1 A Simple Example Showing Output Formatting

The following very simple example shows the expected formatting. Suppose the input is the following file (`hw1-test0.txt`, the name of this file is passed to the VM on the Unix command line):

```
8 2
13 0
```

Running the VM with the above input produces the following output (written to stdout). Note that there are two parts to the output: (1) a listing of the instructions in the program one per line, following a header, with mnemonics for each instruction and (2) a trace of the program's execution, following the line `Tracing ...` (all on standard output). The trace of execution shows the state of the built-in registers (PC, BP, and SP) and the stack's values at addresses between BP and SP $-$ 1 (inclusive), and then it shows the instruction being executed (following the text `==> addr:  `); this consists of: (a) the address of the instruction being executed, then (b) the instruction with its mnemonic and M value, then after showing the instruction being executed (and after the instruction's execution by the VM) the state is again shown. The output of the instruction and the resulting state are show after is each instruction executed.

```
Addr  OP    M
0     INC   2
1     HLT   0
Tracing ...
PC: 0 BP: 0 SP: 0
stack:
==> addr: 0     INC   2
PC: 1 BP: 0 SP: 2
stack: S[0]: 0 S[1]: 0
==> addr: 1     HLT   0
PC: 2 BP: 0 SP: 2
stack: S[0]: 0 S[1]: 0
```

B.2 A Slightly More Involved Example

The following example is a bit more involved and shows some of the details of the machine's execution.

B.2.1 Input File

The following is the contents of the file `hw1-test1.txt`:

```
8 2
1 0
1 1
1 5
1 7
16 0
1 12
22 0
10 11
13 0
1 78
11 0
1 13
```

```
11 0
13 0
```

B.2.2 Output (To Stdout)

Running the VM with the above input produces the following output (written to stdout).

```
Addr  OP    M
0     INC   2
1     LIT   0
2     LIT   1
3     LIT   5
4     LIT   7
5     ADD   0
6     LIT   12
7     NEQ   0
8     JPC   11
9     HLT   0
10    LIT   78
11    CHO   0
12    LIT   13
13    CHO   0
14    HLT   0
Tracing ...
PC: 0 BP: 0 SP: 0
stack:
==> addr: 0     INC   2
PC: 1 BP: 0 SP: 2
stack: S[0]: 0 S[1]: 0
==> addr: 1     LIT   0
PC: 2 BP: 0 SP: 3
stack: S[0]: 0 S[1]: 0 S[2]: 0
==> addr: 2     LIT   1
PC: 3 BP: 0 SP: 4
stack: S[0]: 0 S[1]: 0 S[2]: 0 S[3]: 1
==> addr: 3     LIT   5
PC: 4 BP: 0 SP: 5
stack: S[0]: 0 S[1]: 0 S[2]: 0 S[3]: 1 S[4]: 5
==> addr: 4     LIT   7
PC: 5 BP: 0 SP: 6
stack: S[0]: 0 S[1]: 0 S[2]: 0 S[3]: 1 S[4]: 5 S[5]: 7
==> addr: 5     ADD   0
PC: 6 BP: 0 SP: 5
stack: S[0]: 0 S[1]: 0 S[2]: 0 S[3]: 1 S[4]: 12
==> addr: 6     LIT   12
PC: 7 BP: 0 SP: 6
stack: S[0]: 0 S[1]: 0 S[2]: 0 S[3]: 1 S[4]: 12 S[5]: 12
==> addr: 7     NEQ   0
PC: 8 BP: 0 SP: 5
stack: S[0]: 0 S[1]: 0 S[2]: 0 S[3]: 1 S[4]: 0
==> addr: 8     JPC   11
PC: 9 BP: 0 SP: 4
stack: S[0]: 0 S[1]: 0 S[2]: 0 S[3]: 1
==> addr: 9     HLT   0
```

```
PC: 10 BP: 0 SP: 4
stack: S[0]: 0 S[1]: 0 S[2]: 0 S[3]: 1
```

C Appendix C: Hints

Other sample programs are included in the homework directory, with file names ending in `.txt`.

C.1 Recommended Struct for Instructions

We recommend using the following structure for instructions:

```
typedef struct {
    int op; /* opcode */
    int m; /* M */
} instruction;
```