Stack Machine Revised

Gary T. Leavens
Department of Computer Science
University of Central Florida
Leavens@ucf.edu

March 22, 2023

Abstract

In creating the code generator (for homework 4), we found that the stack machine as described originally (for homework 1) had some features that could be changed to make code generation easier. In this document, the changed and revised features are **highlighted in bold font**.

1 Overview

The virtual machine (VM) is a word-addressible stack-based machine.

The vm's source code is provided in the hw4-tests.zip file in the subidirectory named `vm`.

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

1.1 Inputs

The VM understands the `-n` command line option, which turns off the printing of the program and the VM's tracing output; otherwise the VM prints the program and an execution trace, on `stderr`, by default. The execution trace can be turned off during a program's execution by using the NDB instruction.

The VM takes a single file name as a 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/vm` and the program it should run is contained in the file named `hw1-test1.txt` (and both the input file and the `vm` subdirectory 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/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 `stdin` is buffered. 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/vm progfile.txt
```

One could also put those characters in a file say `cd-input.txt` and then to use the following Unix command: `vm/vm progfile.txt < cd-input.txt`

1.2 Outputs

The VM prints its tracing output to the Unix standard error 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`).

1.3 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).

2 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.

2.1 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.

2.2 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: (a) A number (when OP is LIT or INC), (b) A program address offset (when OP is JMP or JPC), or (c) A program address (when OP is 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 are implemented as (global) variables.

2.3 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 **short int** arithmetic.[2]

2.4 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).

2.5 Size Limits

The following constants define the size limitations of the VM.

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

2.6 Invariants

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

- $0 \leq \mathsf{BP} \wedge \mathsf{BP} \leq \mathsf{SP} \wedge 0 \leq \mathsf{SP} \wedge \mathsf{SP} <$ `MAX_STACK_HEIGHT`
- $0 \leq \mathsf{PC} \wedge \mathsf{PC} <$ `MAX_CODE_LENGTH`

A Appendix A

In the following tables, italicized names (such as p) are meta-variables that refer to (short) 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.

[2]The VM's arithmetic was **changed to be C's `short int` arithmetic in this revision**.

A.1 Basic Instructions

The main changes in the basic instructions are as follows:

- **An explicit no-op instruction (NOP) was added.**
- **The CAL and RTN instructions were changed to save and restore the static link.**
- **The PRM instruction was changed to a LOD instruction that loads an offset from the address on the top of the stack into the top of the stack.**
- **The JMP and JPC instructions were changed to jump relative to the current instruction's address PC by a (possibly negative) offset.**
- **The STO instruction was changed to put the frame's address on the stack first (at stack[SP − 2]) and the value to be stored on the top of the stack (at stack[SP − 1]).**

OP Code	OP Mnemonic	M	Comment (Explanation)
0	**NOP**	–	**do nothing (no-op)**
1	LIT	n	Literal push: stack[SP] ← n; SP ← SP + 1
2	RTN	–	Returns from a **procedure** and restores the caller's AR: PC ← stack[SP − 1]; BP ← stack[SP − 2]; **SP** ← **SP** − 3

3	CAL	p	Call the procedure at code index p, generating a new activation record and setting PC to p:
			stack[**SP**] ← **stack**[**BP**]; // static link
			stack[SP**+ 1**] ← BP; // dynamic link
			stack[SP**+ 2**] ← PC; // return address
			BP ← SP; SP ← SP**+ 3**; PC ← p;

4	POP	–	Pop the stack: SP ← SP − 1;
5	PSI	–	Push the element at address stack[SP − 1] on top of the stack: stack[SP − 1] ← stack[stack[SP − 1]]
6	**LOD**	o	**The value at the address in the top of the stack + offset o is put on top of the stack:** **stack**[**SP** − 1] ← **stack**[**stack**[**SP** − 1] + o]
7	STO	o	**Store stack**[**SP** − 1] **into the stack at address stack**[**SP** − 2] + o and pop the stack twice: **stack**[**stack**[**SP** − 2] + o] ← **stack**[**SP** − 1]; SP ← SP − 2
8	INC	m	Allocate m locals on the stack: SP ← SP + m
9	JMP	o	**Jump relative to the current instruction's code index**: **PC** ← **PC** − 1 + o
10	JPC	o	**Jump conditionally relative to the current instruction's code index**: **if** stack[SP − 1] ≠ 0 **then** {**PC**←**PC-1+o**}
11	CHO	–	Output of the value in stack[SP − 1] to standard output as a character and pop: **putc**(stack[SP − 1], `stdout`); SP ← SP − 1
12	CHI	–	Read an integer, as character value, from standard input and push it in the top of the stack, but on EOF or error, push -1: stack[SP] ← **getc**(`stdin`); SP ← SP + 1
13	HLT	–	Halt the program's execution
14	NDB	–	Stop printing debugging output

A.2 Arithmetic/Logical Instructions

The major changes to these instructions are that the arguments are pushed on the stack with the left argument of a binary operation (or comparison) first and then the right argument. This makes the biggest difference for the SUB, DIV, MOD, and non-equality comparison 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 operations are performed as C's `short int` arithmetic.**[3] Errors such as division by 0 (or modulo by 0) cause the VM to halt with an appropriate error message printed on stderr.

OP Code	Number Mnemonic	M	Comment (Explanation)
15	NEG	–	Negate the value in the top of the stack: $\text{stack}[\mathsf{SP}-1] \leftarrow -\text{stack}[\mathsf{SP}-1]$
16	ADD	–	Add the top two elements in the stack: $\text{stack}[\mathsf{SP}-2] \leftarrow \text{stack}[\mathsf{SP}-2] + \text{stack}[\mathsf{SP}-1]; \mathsf{SP} \leftarrow \mathsf{SP}-1$
17	SUB	–	**Subtract the top element from the 2nd to top one:** $\textbf{stack}[\textbf{SP}-2] \leftarrow \textbf{stack}[\textbf{SP}-2] - \textbf{stack}[\textbf{SP}-1]; \mathsf{SP} \leftarrow \mathsf{SP}-1$
18	MUL	–	Multiply the top two elements in the stack: $\text{stack}[\mathsf{SP}-2] \leftarrow \text{stack}[\mathsf{SP}-2] \times \text{stack}[\mathsf{SP}-1]; \mathsf{SP} \leftarrow \mathsf{SP}-1$
19	DIV	–	**Divide the 2nd from top element by the top one:** $\textbf{stack}[\textbf{SP}-2] \leftarrow \textbf{stack}[\textbf{SP}-2]/\textbf{stack}[\textbf{SP}-1]; \mathsf{SP} \leftarrow \mathsf{SP}-1$
20	MOD	–	**Modulo, result is the remainder of the 2nd from top element by the top element:** $\textbf{stack}[\textbf{SP}-2] \leftarrow \textbf{stack}[\textbf{SP}-1]\textbf{modstack}[\textbf{SP}-2]; \mathsf{SP} \leftarrow \mathsf{SP}-1$
21	EQL	–	Are (the contents of) the top two elements equal? $\text{stack}[\mathsf{SP}-2] \leftarrow \text{stack}[\mathsf{SP}-2] = \text{stack}[\mathsf{SP}-1]; \mathsf{SP} \leftarrow \mathsf{SP}-1$
22	NEQ	–	Are (the contents of) the top two elements different? $\text{stack}[\mathsf{SP}-2] \leftarrow \text{stack}[\mathsf{SP}-2] \neq \text{stack}[\mathsf{SP}-1]; \mathsf{SP} \leftarrow \mathsf{SP}-1$
23	LSS	–	**Is (the contents of) the second from the top element strictly less than the contents of the top element?** $\textbf{stack}[\textbf{SP}-2] \leftarrow \textbf{stack}[\textbf{SP}-2] < \textbf{stack}[\textbf{SP}-1]; \mathsf{SP} \leftarrow \mathsf{SP}-1$
24	LEQ	–	**Is (the contents of) the 2nd from top element no greater than the contents of the top element?** $\textbf{stack}[\textbf{SP}-2] \leftarrow \textbf{stack}[\textbf{SP}-2] \leq \textbf{stack}[\textbf{SP}-1]; \mathsf{SP} \leftarrow \mathsf{SP}-1$
25	GTR	–	**Is (the contents of) the 2nd from top element strictly greater than the contents of the top element?** $\textbf{stack}[\textbf{SP}-2] \leftarrow \textbf{stack}[\textbf{SP}-2] > \textbf{stack}[\textbf{SP}-1]; \mathsf{SP} \leftarrow \mathsf{SP}-1$
26	GEQ	–	**Is (the contents of) the 2nd from top element no less than the contents of the top element?** $\textbf{stack}[\textbf{SP}-2] \leftarrow \textbf{stack}[\textbf{SP}-2] \geq \textbf{stack}[\textbf{SP}-1]; \mathsf{SP} \leftarrow \mathsf{SP}-1$

A.3 VM State Examination Instructions

These instructions allow the state of the VM to be examined and used in computation.

The PBP and PPC instructions were added to allow access to the VM's BP and PC registers in a computation.

[3] **The definition of arithmetic for the arithmetic operators was changed to `short int` arithmetic in this revision.**

OP Code	Number Mnemonic	M	Comment (Explanation)
27	PSP	–	Push SP (i.e., the address itself) on top of the stack: stack[SP] ← SP; SP ← SP + 1
28	**PBP**	–	**Push BP (i.e., the address itself) on top of the stack: stack[SP] ← BP; SP ← SP + 1**
29	**PPC**	–	**Push PC (i.e., the address itself) on top of the stack: stack[SP] ← PC; SP ← SP + 1**

A.4 Examples

As an example, consider the instruction `ADD 0`, which is input as the line `16 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):

Running the VM with the above input produces the following output (written to stderr). 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   3
1     HLT   0
Tracing ...
PC: 0 BP: 0 SP: 0
stack:
==> addr: 0     INC   3
PC: 1 BP: 0 SP: 3
stack: S[0]: 0 S[1]: 0 S[2]: 0
==> addr: 1     HLT   0
PC: 2 BP: 0 SP: 3
stack: S[0]: 0 S[1]: 0 S[2]: 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 3
1 0
1 1
1 5
1 7
16 0
1 12
22 0
10 2
13 0
1 78
11 0
1 13
11 0
13 0
```

B.2.2 Output (to stderr)

Running the VM with the above input produces the following output (written to stderr) (assuming that the `-n` option is not used.

```
Addr  OP    M
0     INC   3
1     LIT   0
2     LIT   1
3     LIT   5
4     LIT   7
5     ADD   0
6     LIT   12
7     NEQ   0
8     JPC   2
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   3
PC: 1 BP: 0 SP: 3
stack: S[0]: 0 S[1]: 0 S[2]: 0
==> addr: 1     LIT   0
PC: 2 BP: 0 SP: 4
stack: S[0]: 0 S[1]: 0 S[2]: 0 S[3]: 0
==> addr: 2     LIT   1
PC: 3 BP: 0 SP: 5
stack: S[0]: 0 S[1]: 0 S[2]: 0 S[3]: 0 S[4]: 1
==> addr: 3     LIT   5
```

```
PC: 4 BP: 0 SP: 6
stack: S[0]: 0 S[1]: 0 S[2]: 0 S[3]: 0 S[4]: 1 S[5]: 5
==> addr: 4     LIT   7
PC: 5 BP: 0 SP: 7
stack: S[0]: 0 S[1]: 0 S[2]: 0 S[3]: 0 S[4]: 1 S[5]: 5 S[6]: 7
==> addr: 5     ADD   0
PC: 6 BP: 0 SP: 6
stack: S[0]: 0 S[1]: 0 S[2]: 0 S[3]: 0 S[4]: 1 S[5]: 12
==> addr: 6     LIT   12
PC: 7 BP: 0 SP: 7
stack: S[0]: 0 S[1]: 0 S[2]: 0 S[3]: 0 S[4]: 1 S[5]: 12 S[6]: 12
==> addr: 7     NEQ   0
PC: 8 BP: 0 SP: 6
stack: S[0]: 0 S[1]: 0 S[2]: 0 S[3]: 0 S[4]: 1 S[5]: 0
==> addr: 8     JPC   2
PC: 9 BP: 0 SP: 5
stack: S[0]: 0 S[1]: 0 S[2]: 0 S[3]: 0 S[4]: 1
==> addr: 9     HLT   0
PC: 10 BP: 0 SP: 5
stack: S[0]: 0 S[1]: 0 S[2]: 0 S[3]: 0 S[4]: 1
```