COP 3402: Systems Software
Spring 2015

Programming Assignment Module #1 (P-Machine)

Due February 10th 2015 by 11:59 PM

The P-Machine

In this assignment, you will implement a virtual machine (VM) known as the P-machine (PM/0) in C, such that it can be compiled and run on the Eustis servers. The specifications listed in this assignment on the PM/0 should be sufficient for an unambiguous implementation of the machine. A brief example of execution listed of the machine is also provided for concrete understanding.

Composition of the P-Machine

The PM/0 is a stack machine with two memory stores and four registers:

- The “Code” – a list of instructions for the VM to execute
- The “Stack” – a stack of data to be used by the PM/0 CPU
- Base Pointer (BP) – The base of the current activation record in the Stack
- Stack Pointer (SP) – The current top of the Stack
- Program Counter (PC) – The current address of the next instruction in the code to execute
- Instruction Register (IR) – The current instruction to execute

Special Values of the P-Machine

The machine will start with the initial values as follows:

$$SP \leftarrow -1; BP \leftarrow 0; PC \leftarrow 0; IR \leftarrow N/A; Stack \leftarrow [0,0,0,\dots]$$

In addition, the maximum stack height at any point of execution is no more than 2000, the maximum code length is 500, and no procedure will be more than three levels down from the main procedure (the maximum lexicographical level).

The Instruction Cycle

The PM/0 instruction cycle is carried out in two steps:

- **Fetch Step**: An instruction is fetched from the Code and placed into the IR register ($IR \leftarrow code[PC]$). Afterwards, the program counter is incremented by 1 to point to the next instruction to be executed($PC \leftarrow PC + 1$).
- **Execute Step**: Following the Fetch Step, the Execute Step will execute the instruction stored in the IR register. Execution is done as specified by the PM/0 ISA, located later in the assignment.

Input and Output

The submitted program should always read from a file named “mcode.txt”. The text file will contain several lines of instructions in PM/0 ISA, all represented as whitespace-separated 32-bit integer values.

The output of the execution (Via the System out instruction) should print “Output:” to the console, and for each value that the machine needs to output, print these values on separate lines in the order that the machine prints them. Furthermore, the program should also write to a file as “debug” information. The name of the file must be “stacktrace.txt”. In this file, the program should first write out the interpreted assembly code with line numbers, followed by the state of the machine during its execution. Format will be specified in the example.

Submission Details

On Webcourses, you must submit a zip file named “[Last name]_[First name]-HW1.zip”. Within this zip file, you *must* include the following:

- The source code of your PM/0 VM (again, in C)
- A readme document (“readme.txt”) indicating how to compile and run the VM
- The input code used to test your machine as well as the output stack trace file generated by it

PM/0 Instruction Set Architecture

The Instruction Set Architecture has 22 instructions, all in the format of three integers (OP, L, M) separated by whitespace. OP is the operation code (or opcode), L indicates the lexicographical level, and M has a variety of meanings that is narrowed down by the opcode. The instructions are listed as follows:

OP	L	M	Name	Meaning	Pseudo Code
1	0	M	LIT	Push constant value (literal) **M** onto the stack.	$SP \leftarrow SP + 1;$ $Stack[SP] \leftarrow M;$
2	0	M	OPR	Depending on **M**, it could have the following meanings:	
		0	RET	Returns from the current procedure. *Note: if this operation clears the stack (SP is -1), then the VM should terminate.*	$SP \leftarrow BP - 1;$ $PC \leftarrow Stack[SP + 3];$ $BP \leftarrow Stack[SP + 2];$
		1	NEG		$Stack[SP] \leftarrow -Stack[SP];$
		2	ADD		$SP \leftarrow SP - 1;$ $Stack[SP] \leftarrow Stack[SP] + Stack[SP + 1];$
		3	SUB		$SP \leftarrow SP - 1;$ $Stack[SP] \leftarrow Stack[SP] - Stack[SP + 1];$
		4	MUL		$SP \leftarrow SP - 1;$ $Stack[SP] \leftarrow Stack[SP] * Stack[SP + 1];$
		5	DIV		$SP \leftarrow SP - 1;$ $Stack[SP] \leftarrow Stack[SP] \div Stack[SP + 1];$
		6	ODD		$Stack[SP] \leftarrow Stack[SP] mod 2$
		7	MOD		$SP \leftarrow SP - 1;$ $Stack[SP] \leftarrow Stack[SP] mod\ Stack[SP + 1];$
		8	EQL	*Note: For $8 \le M \le 13$, the value set is 1 if the comparison is true, 0 otherwise.*	$SP \leftarrow SP - 1;$ $Stack[SP] \leftarrow Stack[SP] \equiv Stack[SP + 1];$

		9	NEQ		$SP \leftarrow SP - 1;$ $Stack[SP] \leftarrow Stack[SP] \neq Stack[SP + 1];$
		10	LSS		$SP \leftarrow SP - 1;$ $Stack[SP] \leftarrow Stack[SP] < Stack[SP + 1];$
		11	LEQ		$SP \leftarrow SP - 1;$ $Stack[SP] \leftarrow Stack[SP] \leq Stack[SP + 1];$
		12	GTR		$SP \leftarrow SP - 1;$ $Stack[SP] \leftarrow Stack[SP] > Stack[SP + 1];$
		13	GEQ		$SP \leftarrow SP - 1;$ $Stack[SP] \leftarrow Stack[SP] \geq Stack[SP + 1];$
3	L	M	LOD	Load value from the Stack location at offset **M** from **L** lexicographical levels up. Then push this loaded value to the top of the Stack.	$SP \leftarrow SP + 1;$ $Stack[SP] \leftarrow Stack[base(L, BP) + M];$
4	L	M	STO	Pop of the value at the top of the Stack and store it at the Stack location at offset **M** from **L** lexicographical levels up.	$Stack[base(L, BP) + M] \leftarrow Stack[SP];$ $SP \leftarrow SP - 1;$
5	L	M	CAL	Call procedure at code index **M**.	$Stack[SP + 1] \leftarrow base(L, BP)$ $Stack[SP + 2] \leftarrow BP$ $Stack[SP + 3] \leftarrow PC$ $BP \leftarrow SP + 1$ $PC \leftarrow M$
6	0	M	INC	Allocate **M** locals at the top of the Stack. The first three are the **Static Link** (SL), **Dyanmic Link** (DL), and the **Return Address** (RA).	$SP \leftarrow SP + M;$
7	0	M	JMP	Jump to instruction **M**.	$PC \leftarrow M;$
8	0	M	JPC	Pop off the value from the top of the Stack. Jump to instruction **M** if that value is 0.	$if\ Stack[SP] \equiv 0\ then\ \{pc \leftarrow M;\}$ $SP \leftarrow SP - 1;$
9	0	0	OUT	Pop off the value from the top of the Stack. Print out that value.	$\mathrm{P}rint(Stack[SP]);$ $SP \leftarrow SP - 1;$
10	0	0	IN	Read in a value and push it to the top of the Stack.	$SP \leftarrow SP + 1;$ $Read(Stack[SP]);$

Input/Output Example

Input File:

```
7 0 10
7 0 2
6 0 5
1 0 13
4 0 3
1 0 1
4 1 3
1 0 7
4 0 4
2 0 0
6 0 5
1 0 3
4 0 3
1 0 0
4 0 4
5 0 2
2 0 0
```

Output to Console:

```
Output:
```

Output to File:

```
Line   OP    L     M
 0     jmp   0     10
 1     jmp   0     2
 2     inc   0     5
 3     lit   0     13
 4     sto   0     3
 5     lit   0     1
 6     sto   1     3
 7     lit   0     7
 8     sto   0     5
 9     opr   0     0
10     inc   0     5
11     lit   0     3
12     sto   0     3
13     lit   0     0
14     sto   0     4
15     cal   0     2
16     opr   0     0

                        pc    bp    sp    stack
Initial values          0     0     -1
```

```
 0    jmp   0     10    10    0     -1
10    inc   0     5     11    0     4     0 0 0 0 0
11    lit   0     3     12    0     5     0 0 0 0 0 3
12    sto   0     3     13    0     4     0 0 0 3 0
13    lit   0     0     14    0     5     0 0 0 3 0 0
14    sto   0     4     15    0     4     0 0 0 3 0
15    cal   0     2     2     5     4     0 0 0 3 0
 2    inc   0     5     3     5     9     0 0 0 3 0 | 0 0 16 0 0
 3    lit   0     13    4     5     10    0 0 0 3 0 | 0 0 16 0 0 13
 4    sto   0     3     5     5     9     0 0 0 3 0 | 0 0 16 13 0
 5    lit   0     1     6     5     10    0 0 0 3 0 | 0 0 16 13 0 1
 6    sto   1     3     7     5     9     0 0 0 1 0 | 0 0 16 13 0
 7    lit   0     7     8     5     10    0 0 0 1 0 | 0 0 16 13 0 7
 8    sto   0     4     9     5     9     0 0 0 1 0 | 0 0 16 13 7
 9    opr   0     0     16    0     4     0 0 0 1 0
16    opr   0     0     0     0     -1
```

NOTE: It is necessary to separate each Activation Record with a bracket “|”.

Tips

Instruction Data Structure

We recommend using the following structure for your instructions:

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

Base Function

This function will be helpful to find a variable in a different Activation Record some **L** levels down:

```
/**********************************************/

/*        Find base L levels down             */

/**********************************************/


int base(l, base) // l stand for L in the instruction format
{
  int b1; //find base L levels down
  b1 = base;
  while (l > 0)
  {
    b1 = stack[b1 + 1];
    l--;
  }
  return b1;
}
```