University of Central Florida
Department of Computer Science
COP 3402: Systems Software
Spring 2020

Homework #1 (P-Machine)

Due Sunday, February 2nd 2020 by 11:59 p.m.

The P-machine:

In this assignment, you will implement a virtual machine (VM) known as the P-machine (PM/0). The P-machine is a stack machine with two memory stores: the "stack" which is an array, organized as a stack in the upper end and a data area for global variables in the lower end. The stack segment contains data to be used by the PM/0 CPU. The other segment is the "text" segment, which is organized as an array and contains the instructions to be executed by PM/0 CPU. The PM/0 CPU has five registers. These registers are named base pointer (BP), stack pointer (SP), global pointer (GP), program-counter (PC) and instruction register (IR). They will be explained in detail later on in this document.

The Instruction Set Architecture (ISA) of the PM/0 has 23 instructions and the instruction format is as follows: "OP L M"

Each instruction contains three components (OP, L, M) that are separated by one space.

OP is the operation code.
L indicates the lexicographical level.
M depending of the operators it indicates:
- A number (instructions: LIT, INC).
- A program address (instructions: JMP, JPC, CAL).
- A data address (instructions: LOD, STO)
- The identity of the operator OPR
  (e.g. OPR 0, 2 (ADD) or OPR 0, 4 (MUL)).

The list of instructions for the ISA can be found in Appendix A and B.

P-Machine Cycles

The PM/0 instruction cycle is carried out in two steps. This means that it executes two steps for each instruction. The first step is the Fetch, where the actual instruction is fetched from the "text" segment in memory. The second step is the Execute, where the instruction that was fetched is executed using the "data-stack" segment in memory and the CPU. This does not mean the instruction is stored in the "stack."

Fetch Step:

In the Fetch Step, an instruction is fetched from the "text" segment and is placed in the IR register (IR ← code[PC]). Then, the program counter is incremented by 1 to point to the next instruction to be executed (PC← PC + 1).

Execute Step:

In the Execute Step, the instruction that was fetched is executed by the PM/0. The OP component that is stored in the IR register (IR.OP) indicates the operation to be executed. For example, if IR.OP is the ISA instruction OPR (IR.OP = 02), then the M component of the instruction in the IR register (IR.M) is used to identify the operation and execute the appropriate arithmetic or relational instruction.

PM/0 Initial/Default Values:

Initial values for PM/0 CPU registers:

SP = MAX_DATA-STACK_HEIGHT;
BP = 0;
PC = 0;
IR = 0;
GP = -1
;

Initial "data-stack" store values:

data-stack[0] =0; data-stack[1] =0; …; data-stack[n-1] =0;

Constant Values:

MAX_DATA-STACK_HEIGHT is 23
MAX_CODE_LENGTH is 500
MAX_LEXI_LEVELS is 3

Assignment Instructions and Guidelines:

1. The VM must be written in C and **must run on Eustis**. If it runs in your PC but not on Eustis, for us it does not run.
2. Submit to Webcourses:
   a) A readme document indicating how to compile and run HW1.

   b) The source code of your PM/0 VM.

c) 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 C.

d) Team assignment (Team size: minimum one student and max. two students)

e) If you program does not follow the specifications, the grade will be zero)

f) The name of all team members must be written at the beginning of the program.

g) Only one student in the team must submit.

h) The team member(s) must be the same for all projects. In case of problems within the team. The team will be split and each member must continue working as a one member team.

i) On late submissions:

One day late 10% off.
Two days late 20% off.
After two days the grade will be zero.

Appendix A

Instruction Set Architecture (ISA)

There are 13 arithmetic/logical operations that manipulate the data within data-stack. These operations are indicated by the **OP** component = 2 (OPR). When an **OPR** instruction is encountered, the **M** component of the instruction is used to select the particular arithmetic/logical operation to be executed (e.g. to multiply the two elements at the top of the stack, write the instruction "2 0 4").

ISA:

01 – **LIT** **0, M** Push constant value (literal) **M** onto the data or stack

02 – **OPR** **0, M** Operation to be performed on the data at the top of the stack (See in Appendix B)

03 – **LOD** **L, M** Load value to top of stack from the stack location at offset **M** from **L** lexicographical levels down

04 – **STO** **L, M** Store value at top of stack in the stack location at offset **M** from **L** lexicographical levels down

05 – **CAL** **L, M** Call procedure at code index **M** (generates new Activation Record and pc ← **M**)

06 – **INC** **0, M** Allocate **M** locals (increment sp by M). First three are **Static Link (SL)**, **Dynamic Link (DL)**, and **Return Address (RA)**

07 – **JMP** **0, M** Jump to instruction **M**

08 – **JPC** **0, M** Jump to instruction **M** if top stack element is 0

09 – **SIO** **0, 1** Write the top stack element to the screen

10 – **SIO** **0, 2** Read in input from the user and store it on top of the stack.

11 – **SIO** **0, 3** End of Program.

Appendix B

ISA Pseudo Code

01 – **LIT 0, M** if (bp = = 0) {gp = gp + 1 ; stack[gp] ← M ;}
else {sp ← sp - 1; stack[sp] ← M; }

02 – **OPR 0, #** (0 ≤ # ≤ 13)

0	RET	sp ← bp + 1 bp ← stack[sp - 3] pc ← stack[sp - 4])
1	NEG	(-stack[sp])
2	ADD	(sp ← sp + 1; stack[sp] ← stack[sp] + stack[sp - 1])
3	SUB	(sp ← sp + 1; stack[sp] ← stack[sp] - stack[sp - 1])
4	MUL	(sp ← sp + 1; stack[sp] ← stack[sp] * stack[sp - 1])
5	DIV	(sp ← sp + 1; stack[sp] ← stack[sp] / stack[sp - 1])
6	ODD	(stack[sp] ← stack[sp] mod 2) or ord(odd(stack[sp]))
7	MOD	(sp ← sp + 1; stack[sp] ← stack[sp] mod stack[sp - 1])
8	EQL	(sp ← sp + 1; stack[sp] ← stack[sp] = = stack[sp - 1])
9	NEQ	(sp ← sp + 1 and stack[sp] ← stack[sp] != stack[sp - 1])
10	LSS	(sp ← sp + 1 and stack[sp] ← stack[sp] < stack[sp - 1])
11	LEQ	(sp ← sp + 1 and stack[sp] ← stack[sp] <= stack[sp - 1])
12	GTR	(sp ← sp + 1 and stack[sp] ← stack[sp] > stack[sp - 1])
13	GEQ	(sp ← sp + 1 and stack[sp] ← stack[sp] >= stack[sp - 1])

03 – **LOD L, M**

```
if (base(L, bp) = = 0)
  {gp = gp + 1; stack[gp] ← stack[ base(L, bp) + M];}
else
  { sp ← sp - 1; stack[sp] ← stack[ base(L, bp) - M];}
```

04 – **STO L, M**

```
if (base(L, bp) = = 0)
  {stack[ base(L, bp) + M] ← stack[gp]; gp = gp – 1;}
else
  {stack[ base(L, bp) - M] ← stack[sp]; sp ← sp + 1;}
```

05 - **CAL L, M**

```
if (sp – 4 ≤ gp) Error_StakOverflow ( );
stack[sp - 1] ← 0                    /* space to return value
stack[sp - 2] ← base(L, bp);         /* static link (SL)
stack[sp - 3] ← bp;                  /* dynamic link (DL)
stack[sp - 4] ← pc                   /* return address (RA)
bp ← sp - 1;
pc ← M;
```

06 – **INC 0, M** if (sp – M ≤ gp) Error_StakOverflow ( );
if (bp = = 0) gp = gp + **M**
else sp ← sp - **M**;

07 – **JMP 0, M** pc ← **M**;

08 – **JPC 0, M** **if** stack[sp] == 0 **then {** pc ← **M; }**
sp ← sp + 1;

09 – **SIO 0, 1** print(stack[sp]);
sp ← sp + 1;

10 - **SIO 0, 2** sp ← sp - 1;
read(stack[sp]);

11 - **SIO 0, 3** Set **Halt** flag to zero;

NOTE: The result of a logical operation such as (A > B) is defined as 1 if the condition was met and 0 otherwise.

Appendix C

Example of Execution

This example shows how to print the stack after the execution of each instruction. The following PL/0 program, once compiled, will be translated into a sequence code for the virtual machine PM/0 as shown below in the **INPUT FILE**.

```
const n = 9;
int i,h;
procedure sub;
  const k = 7;
  int j,h;
  begin
    j:=n;
    i:=1;
    h:=k;
  end;
begin
  i:=3; h:=0;
  call sub;
end.
```

INPUT FILE

For every line, there must be 3 integers representing **OP**, **L** and **M**.

```
7 0 10
7 0 2
6 0 6
1 0 9
4 0 4
1 0 1
4 1 4
1 0 7
4 0 5
2 0 0
6 0 6
1 0 3
4 0 4
1 0 0
4 0 5
5 0 2
11 0 3
```

we recommend using the following structure for your instructions:

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

OUTPUT FILE

1) Print out the program in interpreted assembly language with line numbers:

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

2) Print out the execution of the program in the virtual machine, showing the stack and registers pc, bp, and sp:

				gp	pc	bp	sp	data	stack
Initial values				-1	0	0	23		
0	jmp	0	10	-1	10	0	23		
10	inc	0	6	5	11	0	23	0 0 0 0 0 0	
11	lit	0	3	6	12	0	23	0 0 0 0 0 0 3	
12	sto	0	4	5	13	0	23	0 0 0 0 3 0	
13	lit	0	0	6	14	0	23	0 0 0 0 3 0 0	
14	sto	0	5	6	15	0	23	0 0 0 0 3 0	
15	cal	0	2	6	2	0	23	0 0 0 0 3 0 | 0 0 0 0 0 0 0 0 0 0 0 0 0 16 1 1 0	
2	inc	0	6	5	3	22	17	0 0 0 0 3 0 |	0 0 16 1 1 0
3	lit	0	9	5	4	22	16	0 0 0 0 3 0 |	9 0 0 16 1 1 0
4	sto	0	4	5	5	22	17	0 0 0 0 3 0 |	0 9 16 1 1 0
5	lit	0	1	5	6	22	16	0 0 0 0 3 0 |	1 0 9 16 1 1 0
6	sto	1	4	5	7	22	17	0 0 0 0 1 0 |	0 9 16 1 1 0
7	lit	0	7	5	8	22	16	0 0 0 0 1 0 |	7 0 9 16 1 1 0
8	sto	0	5	5	9	22	17	0 0 0 0 1 0 |	7 9 16 1 1 0
9	opr	0	0	5	16	0	23	0 0 0 0 1 0	
16	sio	0	3	5	17	0	23		

NOTE: It is necessary to separate each Activation Record with a bar "|".

Appendix D

Helpful Tips

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;
}
```

For example in the instruction:

STO L, M - you can do stack[base(ir[pc].**L**, bp) + ir[pc].M] = stack[sp] to store in a variable **L** levels down.