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

Homework #1 (P-Machine)
This is a team project (Max. 2 students per team.)

Due September 24, 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 segments: the "stack," which is organized as a data-stack which contains the data segment (first AR) and the stack to be used by the PM/0 CPU, and the "text", which contains the instructions for the VM to execute. The PM/0 CPU has four registers to handle the stack and text segments: The registers are named base pointer (BP), stack pointer (SP), program counter (PC) and instruction register (IR). They will be explained in detail later on in this document. The machine also has a register file (RF) with eight (8) registers (0-7).

The Instruction Set Architecture (ISA) of PM/0 has 22 instructions and the instruction format has four fields: "OP R L M". They are separated by one space.

OP is the operation code.
R refers to a register in the register file
L indicates the lexicographical level or a register in arithmetic and relational instructions. (L or R)
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)
- A register (R) in arithmetic and logic instructions.
  (e.g. ADD RF[1], RF[2], RF[3] meaning RF[1] ← RF[2] + RF[3])

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 cycle, where the instruction pointed to by the program counter (PC) is fetched from the "text" segment and placed in the instruction register (IR). The second step is the Execute cycle, where the instruction placed in the IR is executed using the "stack" segment and the register file (RF). This does not mean the instruction is stored in the "stack."

Fetch Cycle:

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

Execute Cycle:

In the Execute cycle, the instruction placed in IR, is executed by the VM-CPU. 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 instruction ADD (IR.OP = 11), then R, L, M components of the instruction in IR are used as registers numbers to execute the instruction ADD R L M (RF [R] ← RF [L] + RF [M])

PM/0 Initial/Default Values:

Initial values for PM/0 CPU registers:

SP = MAX_STACK_HEIGHT;
BP = SP -1;
PC = 0;
IR = 0;

Initial "stack" segment values are all zero:

stack[0] =0, stack[2] =0, stack[3] =0…..stack[n-1] = 0.

All registers in the register file have initial value zero (R0 = 0, R1= 0, R3 = 0….. R7 = 0.

Constant Values:

MAX_STACK_HEIGHT is 1000
MAX_CODE_LENGTH is 500

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. "The input file name should be read as a command line argument at runtime, for example: $ ./a.out input.txt".

3. "Program output should be printed to the screen, and should follow the formatting of the example in Appendix C"

4. Submit to Webcourses:
   a) A readme document indicating how to compile and run the VM.

   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) Student names must be shown at the beginning of the program.

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

   g) Include comments in your program.

   h) Do not implement each VM instruction with a function. If you do, a penalty of -5 per function will be applied to your grade in the assignment.

   i) 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 for all other projects.

   j) 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 the register file. These operations will be explained after the 09 basic instructions of PM/0.

ISA:

01	–	**LIT R, 0, M**	Loads a constant value (literal) **M** into Register **R**
02	–	**RTN 0, 0, 0**	Returns from a subroutine and restore the caller environment.
03	–	**LOD R, L, M**	Load value into a selected register from the stack location at offset **M** from **L** lexicographical levels up
04	–	**STO R, L, M**	Store value from a selected register in the stack location at offset **M** from **L** lexicographical levels up
05	–	**CAL 0, L, M**	Call procedure at code index **M** (generates new Activation Record and PC ← **M**)
06	–	**INC 0, 0, M**	Allocate **M** memory words (increment SP by M). First three are reserved to **Static Link (SL)**, **Dynamic Link (DL)**, and **Return Address (RA)**
07	–	**JMP 0, 0, M**	Jump to instruction **M (PC ← M)**
08	–	**JPC R, 0, M**	Jump to instruction **M** if **R** = 0
09	–	**SYS R, 0, 1**	Write a register to the screen
		SYS R, 0, 2	Read in input from the user and store it in a register
		SYS 0, 0, 3	End of program (**Set Halt flag to zero**)

Appendix B

ISA Pseudo Code

01 – **LIT R, 0, M** RF[R] ← **M;**

02 – **RTN 0, 0, 0**
```
sp ← bp + 1;
bp ← stack[sp - 2];
pc ← stack[sp - 3];
```

03 – **LOD R, L, M** RF[R] ← stack[base(**L, bp**) - **M**];

04 – **STO R, L, M** stack[base(**L, bp**) - **M]** ← RF[R];

05 - **CAL 0, L, M**
```
stack[sp - 1] ← base(L, bp);      /* static link (SL)
stack[sp - 2] ← bp;               /* dynamic link (DL)
stack[sp - 3] ← pc;               /* return address (RA)
bp ← sp - 1;
pc ← M;
```

06 – **INC 0, 0, M** sp ← sp - **M**;

07 – **JMP 0, 0, M** pc ← **M**;

08 – **JPC R, 0, M** **if** RF[R] == 0 **then** { pc ← **M; }**

09 – **SYS R, 0, 1** print(RF[R]);

SYS R, 0, 2 read(RF[R]);

SYS 0, 0, 3 **Set Halt flag to zero (End of program).**

10 - **NEG R, L, M** (RF[R] ← - RF[R]
11 - **ADD R, L, M** (RF[R] ← RF[L] + RF[M])
12 - **SUB R, L, M** (RF[R] ← RF[L] - RF[M])
13 - **MUL R, L, M** (RF[R] ← RF[L] * RF[M])
14 - **DIV R, L, M** (RF[R] ← RF[L] / RF[M])
15 - **ODD R, L, M** (RF[R] ← RF[R] mod 2) or ord(odd(RF[R]))
16 - **MOD R, L, M** (RF[R] ← RF[L] mod RF[M])
17 - **EQL R, L, M** (RF[R] ← RF[L] = = RF[M])
18 - **NEQ R, L, M** (RF[R] ← RF[L] != RF[M])
19 - **LSS R, L, M** (RF[R] ← RF[L] < RF[M])

20 - **LEQ R, L, M** (RF[R] ← RF[L] <= RF[M])])
21 - **GTR R, L, M** (RF[R] ← RF[L] > RF[M])
22 - **GEQ R, L, M** (RF[R] ← RF[L] >= RF[M])

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

NOTE: If we have the instruction ADD 4 5 6 (11 4 5 6), we have to interpret this as: RF[4] ← RF[5] + RF[6]

Another example: if we have instruction LIT 5 0 9 (1 5 0 9), we have to interpret this as: RF[5] ← 9

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 = 8;
int i,h;
procedure sub;
  const k = 7;
  int j,h;
  begin
    j:=n;
    i:=1;
    h:=k;
  end;
begin
  i:=3; h:=9;
  call sub;
end.
```

INPUT FILE

For every line, there must be 4 integers representing **OP**, **R**, **L** and **M**.

```
7 0 0 10  ← PC =0
7 0 0 2
6 0 0 5
1 0 0 8
4 0 0 3
1 0 0 1
4 0 1 3
1 0 0 7
4 0 0 4
2 0 0 0
6 0 0 5
1 0 0 3
4 0 0 3
1 0 0 9
4 0 0 4
5 0 0 2
9 0 0 3
```

we recommend using the following structure for your instructions:

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

OUTPUT FILE

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

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

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

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

```
                                        pc     bp     sp
Initial values                          0      999    1000
Registers: 0  0  0  0  0  0  0  0
Stack: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

                                        pc     bp     sp
0  jmp  0  0  10                        10     999    1000
Registers: 0  0  0  0  0  0  0  0
Stack:

10  inc   0  0   5                      11     999     995
Registers: 0  0  0  0  0  0  0  0
Stack: 0 0 0 0 0

11  lit    0  0   3                     12     999     995
Registers: 3  0  0  0  0  0  0  0
Stack: 0 0 0 0 0
```

12 sto 0 0 3 13 999 995
Registers: 3 0 0 0 0 0 0 0
Stack: 0 0 0 3 0

13 lit 0 0 9 14 999 995
Registers: 9 0 0 0 0 0 0 0
Stack: 0 0 0 3 0

14 sto 0 0 4 15 999 995
Registers: 9 0 0 0 0 0 0 0
Stack: 0 0 0 3 9

15 cal 0 0 2 2 994 995
Registers: 9 0 0 0 0 0 0 0
Stack: 0 0 0 3 9

2 inc 0 0 5 3 994 990
Registers: 9 0 0 0 0 0 0 0
Stack: 0 0 0 3 9 | 999 999 16 0 0

3 lit 0 0 8 4 994 990
Registers: 8 0 0 0 0 0 0 0
Stack: 0 0 0 3 9 | 999 999 16 0 0

4 sto 0 0 3 5 994 990
Registers: 8 0 0 0 0 0 0 0
Stack: 0 0 0 3 9 | 999 999 16 8 0

5 lit 0 0 1 6 994 990
Registers: 1 0 0 0 0 0 0 0
Stack: 0 0 0 3 9 | 999 999 16 8 0

6 sto 0 1 3 7 994 990
Registers: 1 0 0 0 0 0 0 0
Stack: 0 0 0 1 9 | 999 999 16 8 0

7 lit 0 0 7 8 994 990
Registers: 7 0 0 0 0 0 0 0
Stack: 0 0 0 1 9 | 999 999 16 8 0

```
 8   sto   0  0  4                        9        994  990
Registers: 7  0  0  0  0  0  0  0
Stack:  0 0 0 1 9 | 999 999 16 8 7

 9   rtn   0  0  0                        16       999  995
Registers: 7  0  0  0  0  0  0  0
Stack:  0 0 0 1 9

16  sio    0  0  3                        17       999   995
Registers: 7  0  0  0  0  0  0  0
Stack:  0 0 0 1 9
```

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 up:

```
/*******************************************/
/*            Find base L levels up        */
/*                                         */
/*******************************************/

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

For example in the instruction:

STO R, L, M - You can do stack[base(ir.**L**, bp) + ir[IR.**M**] = RF[IR.R] to store the content of register into the stack **L** levels up from the current AR. RF stand for register file.