University of Central Florida
College of Engineering & Computer Science
COP 3402: Systems Software
Summer 2018

Homework #1 (P-Machine)

Due Sunday, June 3, by 11:59 p.m.

Team project (Max. 2 students)

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 organized as a stack and contains the data to be used by the PM/0 CPU, and the "code," which is organized as a list and contains the instructions for the VM. The PM/0 CPU has four registers to handle the stack and code 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 sixteen (16) registers (0-15).

The Instruction Set Architecture (ISA) of the PM/0 has 24 instructions and the instruction format is as follows: "OP R L M"

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

OP is the operation code.
R refers to a register
L indicates the lexicographical level or a register in arithmetic and logic instructions.
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 in arithmetic and logic instructions.
  (e.g. ADD R[1], R[2], R[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 actual instruction is fetched from the "code" memory store. The second step is the Execute Cycle, where the instruction that was fetched is executed using the "stack" memory store 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 "code" store and placed in the IR register (IR ← code[PC]). Afterwards, 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 that was fetched is executed by the VM. 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 ADD (IR.OP = 12), then the R, L, M component of the instruction in the IR register (IR.R, IR.L, IR.M) are used as a register and execute the appropriate arithmetic or logical instruction.

PM/0 Initial/Default Values:
Initial values for PM/0 CPU registers:
SP = 0; BP = 1; PC = 0;IR = 0;

Initial "stack" store values are all zero: We just show the first three stack locations:
stack[1] =0, stack[2] =0, stack[3] =0…..stack[1999] = 0.

All registers in the register file have initial value zero (R0 = 0, R1= 0, R3 = 0….R15 = 0.
Constant Values:
MAX_STACK_HEIGHT is 2000
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**.
2. 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.

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 9 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 down
04	–	**STO** R, **L, M**	Store value from a selected register in the stack location at offset **M** from **L** lexicographical levels down
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** locals (increment sp by M). First four are **Functional Value, Static Link (SL)**, **Dynamic Link (DL)**, and **Return Address (RA)**
07	–	**JMP 0, 0, M**	Jump to instruction **M**
08	–	**JPC R, 0, M**	Jump to instruction **M** if **R** = 0
09	–	**SIO** R, **0, 1**	Write a register to the screen
09	**–**	**SIO R, 0, 2**	Read in input from the user and store it in a register
09	**–**	**SIO 0, 0, 3**	End of program (program stops running)

Appendix B

ISA Pseudo Code

```
01 – LIT    R, 0,  M     R[i] ← M;


02 – RTN  0, 0, 0        sp ← bp - 1;
                         bp ← stack[sp + 3];
                         pc ← stack[sp + 4];

03 – LOD R, L, M         R[i] ← stack[ base(L, bp) + M];

04 – STO R, L, M         stack[ base(L, bp) + M] ← R[i];

05 - CAL   0, L, M       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, 0, M       sp ← sp + M;

07 – JMP   0, 0, M       pc ← M;

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

09 – SIO   R, 0, 1       print(R[i]);

09 - SIO   R, 0, 2       read(R[i]);

09 – SIO   0, 0, 3       Set Halt flag to one;

10 - NEG   (R[i] ← -R[j])
11 - ADD   (R[i] ← R[j] + R[k])
12 - SUB   (R[i] ← R[j] - R[k])
13 - MUL   (R[i] ← R[j] * R[k])
14 - DIV   (R[i] ← R[j] / R[k])
15 - ODD   (R[i] ← R[i] mod 2) or ord(odd(R[i]))
16 - MOD   (R[i] ← R[j] mod  R[k])
```

17 - **EQL** (R[i] ← R[j] = = R[k])
18 - **NEQ** (R[i] ← R[j] != R[k])
19 - **LSS** (R[i] ← R[j] < R[k])
20 - **LEQ** (R[i] ← R[j] <= R[k])
21 - **GTR** (R[i] ← R[j] > R[k])
22 - **GEQ** (R[i] ← R[j] >= R[k])

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

NOTE: in all instructions, “i” refers to operand R, “j” refers to operand L, and “k” refers to operand M. For example, if we have the instruction ADD 7 8 9 (11 7 8 9), we have to interpret this as:
R[7] ← R[8] + R[9]

Another example: if we have instruction LIT 5 0 9 (1 5 0 9), we have to interpret this as:
R[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**.

INPUT FILE

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

```
Factorial Program:
6 0 0 6
1 0 0 3
4 0 0 4
1 0 0 1
4 0 0 5
5 0 0 7
7 0 0 19
6 0 0 4
3 0 1 4
3 1 1 5
13 1 0 1
4 1 1 5
1 1 0 1
12 0 0 1
4 0 1 4
18 0 0 1
8 0 0 18
5 0 1 7
2 0 0 0
3 0 0 5
9 0 0 1
9 0 0 3
```

```
we recommend using the following structure for your
instructions:

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

```
Factorial Op Printout:
0 inc 0 0 6
1 lit 0 0 3
2 sto 0 0 4
3 lit 0 0 1
4 sto 0 0 5
5 cal 0 0 7
6 jmp 0 0 19
7 inc 0 0 4
8 lod 0 1 4
9 lod 1 1 5
10 mul 1 0 1
11 sto 1 1 5
```

```
12 lit 1 0 1
13 sub 0 0 1
14 sto 0 1 4
15 neq 0 0 1
16 jpc 0 0 18
17 cal 0 1 7
18 rtn 0 0 0
19 lod 0 0 5
20 sio 0 0 1
21 sio 0 0 3
```

```
Factorial Stack Trace:
Inital Values           pc   bp   sp
0    inc  0    0    6    1    1    6    0 1 0 0 0 0
1    lit  0    0    3    2    1    6    0 1 0 0 0 0
2    sto  0    0    4    3    1    6    0 1 0 0 3 0
3    lit  0    0    1    4    1    6    0 1 0 0 3 0
4    sto  0    0    5    5    1    6    0 1 0 0 3 1
5    cal  0    0    7    7    7    6    0 1 0 0 3 1
7    inc  0    0    4    8    7    10   0 1 0 0 3 1 | 0 1 1 6
8    lod  0    1    4    9    7    10   0 1 0 0 3 1 | 0 1 1 6
9    lod  1    1    5    10   7    10   0 1 0 0 3 1 | 0 1 1 6
10   mul  1    0    1    11   7    10   0 1 0 0 3 1 | 0 1 1 6
11   sto  1    1    5    12   7    10   0 1 0 0 3 3 | 0 1 1 6
12   lit  1    0    1    13   7    10   0 1 0 0 3 3 | 0 1 1 6
13   sub  0    0    1    14   7    10   0 1 0 0 3 3 | 0 1 1 6
14   sto  0    1    4    15   7    10   0 1 0 0 2 3 | 0 1 1 6
15   neq  0    0    1    16   7    10   0 1 0 0 2 3 | 0 1 1 6
16   jpc  0    0    18   17   7    10   0 1 0 0 2 3 | 0 1 1 6
17   cal  0    1    7    7    11   10   0 1 0 0 2 3 | 0 1 1 6
7    inc  0    0    4    8    11   14   0 1 0 0 2 3 | 0 1 1 6 | 0 1 7 18
8    lod  0    1    4    9    11   14   0 1 0 0 2 3 | 0 1 1 6 | 0 1 7 18
9    lod  1    1    5    10   11   14   0 1 0 0 2 3 | 0 1 1 6 | 0 1 7 18
10   mul  1    0    1    11   11   14   0 1 0 0 2 3 | 0 1 1 6 | 0 1 7 18
11   sto  1    1    5    12   11   14   0 1 0 0 2 6 | 0 1 1 6 | 0 1 7 18
12   lit  1    0    1    13   11   14   0 1 0 0 2 6 | 0 1 1 6 | 0 1 7 18
13   sub  0    0    1    14   11   14   0 1 0 0 2 6 | 0 1 1 6 | 0 1 7 18
14   sto  0    1    4    15   11   14   0 1 0 0 1 6 | 0 1 1 6 | 0 1 7 18
15   neq  0    0    1    16   11   14   0 1 0 0 1 6 | 0 1 1 6 | 0 1 7 18
16   jpc  0    0    18   18   11   14   0 1 0 0 1 6 | 0 1 1 6 | 0 1 7 18
18   rtn  0    0    0    18   7    10   0 1 0 0 1 6 | 0 1 1 6
18   rtn  0    0    0    6    1    6    0 1 0 0 1 6
6    jmp  0    0    19   19   1    6    0 1 0 0 1 6
19   lod  0    0    5    20   1    6    0 1 0 0 1 6
20   sio  0    0    1    21   1    6    0 1 0 0 1 6
21   sio  0    0    3    0    1    0    0
```

```
Factorial Output:
6
```

NOTE: It is necessary to separate each Activation Record with a bar “|”.
NOTE 2: After printing each line you must print the content of all registers.
For instance:

```
lit 1   0   1   13  11  14  0 1 0 0 2 6 | 0 1 1 6 | 0 1 7 18
```

RF: 0, 1, 0, 0, 0, 0, 0, 0

```
lit 5   0   9   13  11  14  0 1 0 0 2 6 | 0 1 1 6 | 0 1 7 18
```

RF: 0, 1, 0, 0, 0, 9, 0, 0

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 R, L, M - you can do stack[base(ir[pc].**L**, bp) + ir[pc].M] = R[i] to store the content of register into the stack **L** levels down from the current AR.