University of Central Florida

Department of Electrical Engineering & Computer Science

COP 3402: Systems Software

Summer 2012

Homework #1 (P-Machine)

Due Sunday, June 5th 2012 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 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. 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 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 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. 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 OPR (IR.OP = 02), then the M component of the instruction in the IR register (IR.M) is used to identify the operator 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:

stack[1] =0;
stack[2] =0;
stack[3] =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.
   d) Programming teams: You can work on teams. Max. two (2) students per team

Appendix A

Instruction Set Architecture (ISA)

There are 13 arithmetic/logical operations that manipulate the data within 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 stack
02	–	**OPR**	**0, M**	Operation to be performed on the data at the top of the stack (detailed 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 at the top of the stack

Appendix B

ISA Pseudo Code

```
01 – LIT    0, M        sp ← sp + 1;
                        stack[sp] ← M;

02 – OPR   0, #   ( 0  ≤  #  ≤  13 )
                  0   RET    (sp ← bp -1 and pc ← stack[sp + 3] and bp ← stack[sp + 2])
                  1   NEG    (-stack[sp])
                  2   ADD    (sp ← sp – 1 and  stack[sp] ← stack[sp] + stack[sp + 1])
                  3   SUB    (sp ← sp – 1 and  stack[sp] ← stack[sp] - stack[sp + 1])
                  4   MUL    (sp ←sp – 1 and  stack[sp] ← stack[sp] * stack[sp + 1])
                  5   DIV    (sp ←sp – 1 and 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 and  stack[sp] ← stack[sp] mod stack[sp + 1])
                  8   EQL    (sp ← sp – 1 and  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         sp ←  sp + 1;
                        stack[sp] ← stack[ base(L, bp) + M];

04 – STO   L, M         stack[ base(L, bp) + M] ← stack[sp];
                        sp ← sp - 1;

05 - CAL   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, M        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]);
```

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 = 13;
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 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
```

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	5
3	lit	0	13
4	sto	0	3
5	lit	0	1
6	sto	1	3
7	lit	0	7
8	sto	0	4
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

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

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

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

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];
    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 a variable **L** levels down.