COP 3402: Systems Software
Summer 2015

Programming Project Module #1 (P-Machine)

Due on Sunday 05/31 by 11:59 PM

The P-machine:

Your task is to implement a virtual machine (VM) known as the P-machine (PM/0). The P-machine is a stack machine with two memory stores: `stack`, which is organized as a stack and contains the data to be used by the PM/0 CPU, and `code`, which is organized as a list and contains the instructions for the VM. The PM/0 CPU has four registers. The registers are base pointer `bp`, stack pointer `sp`, program counter `pc` and instruction register `ir`.

The Instruction Set Architecture (ISA) of the PM/0 has 23 instructions. Each instructions is made of three integers, and the instruction format is as follows.

Each instruction contains three components `OP L M` that are separated by one space.

- **OP** operation code (opcode).
- **L** lexicographical level.
- **M** depending on the opcode it indicates:
  - Number (instructions: `LIT, INC`).
  - Program Address (instructions: JMP, JPC, `CAL`).
  - Data Address (instructions: `LOD, STO`)
  - Identity of the operator OPR
    For example, `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.

Fetch Cycle:

In the Fetch Cycle, an instruction is fetched from `code` store and placed in `ir`:

```
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: We just show the first three stack locations:

```
stack[1] =0;
stack[2] =0;
stack[3] =0;
```

Constant Values:

```
MAX_STACK_HEIGHT = 2000;
MAX_CODE_LENGTH = 500;
MAX_LEXI_LEVELS = 3;
```

Assignment Instructions and Guidelines:

1. The VM must be written in C and **must run on Eustis**.
2. Submit to Webcourses (inside a single zip file):
   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 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”).

op		L	M	
01	**LIT**	**0**	**M**	Push value **M** onto stack
02	**OPR**	**0**	**M**	(arithmetic or logical operations defined in detail below)
03	**LOD**	**L**	**M**	Get value at offset **M** in frame **L** levels down and push it
04	**STO**	**L**	**M**	Pop stack and insert value at offset **M** in frame **L** levels down
05	**CAL**	**L**	**M**	Call procedure at **M** (generates new stack frame)
06	**INC**	**0**	**M**	Allocate **M** locals on stack
07	**JMP**	**0**	**M**	Jump to **M**
08	**JPC**	**0**	**M**	Pop stack and jump to **M** if value is equal to 0
09	**SIO**	**0**	**0**	Pop stack and print out value
09	**SIO**	**0**	**1**	Read in input from user and push it
09	**SIO**	**0**	**2**	Halt the machine

Appendix B

ISA Pseudo Code

```
01  LIT  0  M
              sp = sp + 1;
              stack[sp] = M;

02  OPR  0  M

 0 RET  sp = bp – 1; pc = stack[sp + 4]; bp = stack[sp + 3];

 1 NEG  stack[sp] = -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;

 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; stack[sp] = stack[sp] != stack[sp + 1];

10 LSS  sp = sp – 1; stack[sp] = stack[sp] <  stack[sp + 1];

11 LEQ  sp = sp – 1; stack[sp] = stack[sp] <= stack[sp + 1];

12 GTR  sp = sp – 1; stack[sp] = stack[sp] >  stack[sp + 1];

13 GEQ  sp = sp – 1; 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] = 0;            /* return value (FV)
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
              sp = sp + M;

07  JMP  0  M
              pc = M;

08  JPC  0  M
              if ( stack[ sp ] == 0 ) then { pc = M; }
              sp = sp - 1;

// minor correction: the numbers of the modifier component
// for the SIO operation below were not correct.  The incorrect
// values were 1, 2, and 3. (Corrected // 05/26 at 10:00AM)

09  SIO  0  0
              print(stack[ sp ]);
              sp = sp – 1;

09  SIO  0  1
              sp = sp + 1;
              read(stack[ sp ]);

09  SIO  0  2  halt;
```

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 execution of P-code

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

INPUT FILE

The input file must be named "mcode.txt". For every line, there must be 3 integers representing **OP**, **L** and **M**.

```
7 0 10
7 0 2
6 0 6
1 0 13
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
9 0 2 // NOTE: the modifier is 2 now (it used to be 3).
```

we recommend using the following structure for your instructions:

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

OUTPUT FILE

The Virtual Machine must output a file named "stacktrace.txt". This file must contain two parts:

1) 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     13
 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     2
```

2) 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    jmp   0     10      10    1     0
10    inc   0     6       11    1     6     0 0 0 0 0 0
11    lit   0     3       12    1     7     0 0 0 0 0 0 3
12    sto   0     4       13    1     6     0 0 0 0 3 0
13    lit   0     0       14    1     7     0 0 0 0 3 0 0
14    sto   0     5       15    1     6     0 0 0 0 3 0
15    cal   0     2       2     7     6     0 0 0 0 3 0
 2    inc   0     6       3     7     12    0 0 0 0 3 0 | 0 1 1 16 0 0
 3    lit   0     13      4     7     13    0 0 0 0 3 0 | 0 1 1 16 0 0 13
 4    sto   0     4       5     7     12    0 0 0 0 3 0 | 0 1 1 16 13 0
 5    lit   0     1       6     7     13    0 0 0 0 3 0 | 0 1 1 16 13 0 1
 6    sto   1     4       7     7     12    0 0 0 0 1 0 | 0 1 1 16 13 0
 7    lit   0     7       8     7     13    0 0 0 0 1 0 | 0 1 1 16 13 0 7
 8    sto   0     5       9     7     12    0 0 0 0 1 0 | 0 1 1 16 13 7
 9    opr   0     0       16    1     6     0 0 0 0 1 0
16    sio   0     2       17    1     6     0 0 0 0 1 0
```

NOTE: It is necessary to separate each Activation Record with a bracket "|".
NOTE: The first element in the stack `stack[0]` is never shown.

Appendix D: Hints

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(level, b) {
  while (level > 0){
    b = stack[b + 2];  // NOTE THE CHANGE! We now have + 2 !
    level--;
  }
  return b;
}
```

For example in the instruction:

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