University of Central Florida

Department of Computer Science

COP 3402: Systems Software

Summer 2021

Homework #1 (P-Machine)

This project can be done solo OR with one partner

Due June 4, 2021 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 one memory area called the process address space (PAS). The process address space is divide into two contiguous segments: the "text", which contains the instructions for the VM to execute and the "stack," which is organized as a data-stack to be used by the PM/0 CPU.

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 stack grows upwards.

The Instruction Set Architecture (ISA) of PM/0 has 24 instructions and the instruction format has three fields: "OP L M". They are separated by one space.

OP is the operation code.
L indicates the lexicographical
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 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. 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 $\leftarrow$ text[PC]) and the program counter is incremented by 3 to point to the next instruction to be executed (PC$\leftarrow$ PC + 3).

Execute Cycle:
In the Execute cycle, the instruction placed in IR, is executed by the VM-CPU. The op-code (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 OPR (IR.OP = 2), then the M fieldt of the instruction in the IR register (IR.M) is used to identify the operator and execute the appropriate arithmetic or relational instruction.

PM/0 Initial/Default Values:
Initial values for PM/0 CPU registers will be set up once the program has been uploaded in the process address space:

SP = points to the value "M" in the last instruction in text;
BP = SP + 1;
PC = 0;
IR = null;

Initial process address space values are all zero:

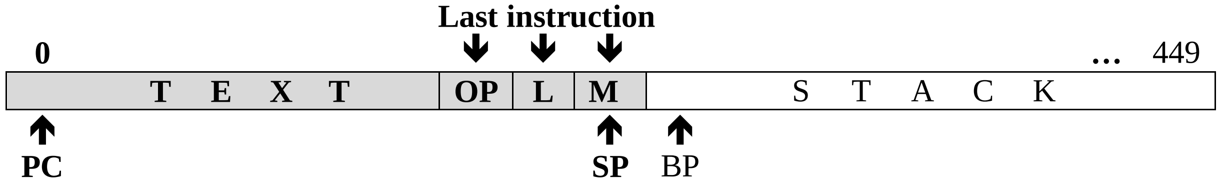
pas[0] =0, pas[1] =0, pas[3] =0…..[n-1] = 0.

Constant Values:

MAX_PAS_LENGTH is 500

You will never be given an input file with more than text length greater than 150 lines of code. However as programs to be run in PM/0 has different length, The values of SP and BP will be set up dynamically once the program had been uploaded.

The figure bellow illustrate the process address space:



Assignment Instructions and Guidelines:

1. The VM must be written in C and must run on Eustis3. 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 (A deduction of 5 points will be applied to submissions that do not implement this).
3. Program output should be printed to the screen, and should follow the formatting of the example in Appendix C. A deduction of 5 points will be applied to submissions that do not implement this.
4. Submit to Webcourses:
   a) A readme text file indicating how to compile and run the VM.
   b) The source code of your PM/0 VM which should be named "vm.c"
   c) Student names should be written in the header comment of each source code file, in the readme, and in the comments of the submission
   d) Do not change the ISA. Do not add instructions or combine instructions. Do not change the format of the input. If you do so, your grade will be zero.
   e) Include comments in your program.
   f) Do not implement each VM instruction with a function. If you do, a penalty of -15 will be applied to your grade. You should only have two functions: main and base (Appendix D).
   g) 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.
   h) On late submissions:
      - **One day late 10% off.**
      - **Two days late 20% off.**
      - **Submissions will not be accepted after two days.**
      - **Resubmissions are not accepted after two days.**
      - **Your latest submission is the one that will be graded.**

We will be using a bash script to test your programs. This means your program should follow the output guidelines listed (see Appendix C for an example). You don't need to be concerned about whitespace beyond newline characters. We use diff -w.

Output specifications:

- The first line should contain column headers: "PC", "BP", "SP", and " segment" in that order.
- The second line should contain "Initial values: " followed by the initial values of each register.

- Each line of the trace should have the following values:
  - The input line number
  - Three letter interpreted opcode in upper case (ex. 6 = INC), opcode 2 should be interpreted based on the M value (ex. 2 0 4 = MUL)
  - The code line's L value
  - The code line's M value
  - PC
  - BP
  - SP
  - The values of the stack printed from pas[first position] to pas[SP] inclusive
- If you encounter the read instruction (9 0 2), print out "Please Enter an Integer:" followed by a scanf, then the stack trace for this line.
- If you encounter the write instruction (9 0 1), print out "Top of Stack Value:" followed by the aforementioned value.
- Your program should have a single empty line at the end
- In total, output should have 3 lines + 1 for every instruction executed + 1 for every 9 0 2 or 9 0 1 instruction

You can check that your program is following the specifications by downloading the contents of the HW1 Example file on webcourses. It contains the files tester.sh, test1.txt, output1.txt, test2.txt and output2.txt All you need to do is make sure your C file is in the same directory as these three files and run "$ bash tester.sh" It will let you know if you're passing the test cases. If your program follows the output specifications, you can earn 5 bonus points.

Rubric:

-100 – Does not compile
10 – Compiles
20 – Produces lines of meaningful execution before segfaulting or looping infinitely
5 – Follows IO specifications (takes command line argument for input file name and prints output to console)
5 – README.txt containing author names
5 – Fetch cycle is implemented correctly
15 – Instructions are not implemented with individual functions
5 – Well commented source code
5 – Arithmetic instructions are implemented correctly
5 – Read and write instructions are implemented correctly
10 – Load and store instructions are implemented correctly
10 – Call and return instructions are implemented correctly
5 – Follows formatting guidelines correctly, source code is named vm.c

Appendix A

Instruction Set Architecture (ISA) – (eventually we will use "stack" to refer to the stack segment)

There are 13 arithmetic/logical operations that manipulate the data within the stack segment. 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 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**	Pushes a constant value (literal) **M** onto the stack
02 –	**OPR 0, M**	Operation to be performed on the data at the top of the stack.
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** memory words (increment SP by M). First four are reserved to **Static Link (SL)**, **Dynamic Link (DL)**, **Return Address (RA), and Parameter (P)**
07 –	**JMP 0, M**	Jump to instruction **M (PC ← M)**
08 –	**JPC 0, M**	Jump to instruction **M** if top stack element is 1
09 –	**SYS 0, 1**	Write the top stack element to the screen
	SYS 0, 2	Read in input from the user and store it on top of the stack
	SYS 0, 3	End of program (**Set Halt flag to zero**)

Appendix B

ISA Pseudo Code

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

02 – OPR  0, #         (0 <= # <= 13)
                       0     RTN      sp ← bp - 1;
                                      bp ← pas[sp + 2];
                                      pc ← pas[sp + 3];

                       1     NEG      pas[sp] ← -1 * pas[sp]

                       2     ADD      sp ← sp – 1;
                                      pas[sp] ← pas[sp] + pas[sp + 1]

                       3     SUB      sp ← sp – 1;
                                      pas[sp] ← pas[sp] - pas[sp + 1]

                       4     MUL      sp ← sp – 1;
                                      pas[sp] ← pas[sp] * pas[sp + 1]

                       5     DIV      sp ← sp – 1;
                                      pas[sp] ← pas[sp] / pas[sp + 1]

                       6     ODD      pas[sp] ← pas[sp] mod 2

                       7     MOD      sp ← sp – 1;
                                      pas[sp] ← pas[sp] mod pas[sp + 1]

                       8     EQL      sp ← sp – 1;
                                      pas[sp] ← pas[sp] == pas[sp + 1]

                       9     NEQ      sp ← sp – 1;
                                      pas[sp] ← pas[sp] != pas[sp + 1]

                       10    LSS      sp ← sp – 1;
                                      pas[sp] ← pas[sp] < pas[sp + 1]

                       11    LEQ      sp ← sp – 1;
                                      pas[sp] ← pas[sp] <= pas[sp + 1]
```

```
                  12    GTR         sp ← sp – 1;
                                    pas[sp] ← pas[sp] > pas[sp + 1]

                  13    GEQ         sp ← sp – 1;
                                    pas[sp] ← pas[sp] >= pas[sp + 1]

03 – LOD L, M     sp ← sp + 1;
                  pas[sp] ← pas[base(L) + M];

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

05 - CAL   L, M   pas[sp + 1]  ←  base(L);     /* static link (SL)
                  pas[sp + 2]  ← bp;           /* dynamic link (DL)
                  pas[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 pas[sp] == 1 then { pc ← M; }
                  sp ← sp - 1;

09 – SYS   0, 1   printf("%d", pas[sp]);
                  sp ← sp - 1;

     SYS   0, 2   sp ← sp + 1;
                  scanf("%d", pas[sp]);

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

NOTE: The result of a logical operation such as (A < B) is defined as 1 if the condition was met and 0 otherwise. Take a look to the following example (LSS) to understand the implementation of relational operations:

```
LSS:
    sp = sp -1;
    if (pas[sp] ← pas[sp] < pas[sp + 1])
        stack[sp] = 1;
    else
        stack[sp] = 0;
```

Appendix C

Example of Execution

This example shows how to print the stack after the execution of each instruction.

INPUT FILE

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

```
7 0 45
7 0 6
6 0 4
1 0 4
1 0 3
2 0 4
4 1 4
1 0 14
3 1 4
2 0 10
8 0 39
1 0 7
7 0 42
1 0 5
2 0 0
6 0 5
9 0 2
5 0 6
9 0 1
9 0 3
```

When the input file (program) is read in to be stored in the text segment starting at location 0 in the process address space, each instruction will need three memory locations to be stored. Therefore, the PC must be incremented by 3 in the fetch cycle.

0			**3**			**6**			**9**			**12**			**15**			…
7	**0**	**45**	7	0	6	**6**	**0**	**4**	1	0	4	**1**	**0**	**3**	2	0	4	etc

The initial CPU register values for the example in this appendix are:

SP = 59;
BP = SP + 1;
PC = 0;
IR = null;

Hint: Each instruction uses 3 array elements and each data value just uses 1 array element.

<u>OUTPUT FILE</u>
Print out the execution of the program in the virtual machine, showing the stack and pc, bp, and sp.

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

```
                PC   BP   SP   stack
Initial values: 0    60   59

 0 JMP  0 45    45   60   59
45 INC  0  5    48   60   64   0 0 0 0 0
Please Enter an Integer: 3
48 SYS  0  2    51   60   65   0 0 0 0 0  3
51 CAL  0  6     6   66   65   0 0 0 0 0  3
 6 INC  0  4     9   66   69   0 0 0 0 0  3 |60 60 54 0
 9 LIT  0  4    12   66   70   0 0 0 0 0  3 |60 60 54 0 4
12 LIT  0  3    15   66   71   0 0 0 0 0  3 |60 60 54 0 4 3
15 MUL  0  4    18   66   70   0 0 0 0 0  3 |60 60 54 0 12
18 STO  1  4    21   66   69   0 0 0 0 12 3 |60 60 54 0
21 LIT  0 14    24   66   70   0 0 0 0 12 3 |60 60 54 0 14
24 LOD  1  4    27   66   71   0 0 0 0 12 3 |60 60 54 0 14 12
27 LSS  0 10    30   66   70   0 0 0 0 12 3 |60 60 54 0 0
30 JPC  0 39    33   66   69   0 0 0 0 12 3 |60 60 54 0
33 LIT  0  7    36   66   70   0 0 0 0 12 3 |60 60 54 0 7
36 JMP  0 42    42   66   70   0 0 0 0 12 3 |60 60 54 0 7
42 RTN  0  0    54   60   65   0 0 0 0 12 3
Output result is: 3
54 SYS  0  1    57   60   64   0 0 0 0 12
57 SYS  0  3    60   60   64   0 0 0 0 12
```

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(int L)
{
        int arb = BP;      // arb = activation record base
        while ( L > 0)     //find base L levels down
        {
                arb = pas[arb];
                L--;
        }
        return arb;
}
```

For example in the instruction:

STO L, M - You can do stack [base (IR.**L**) + IR.**M**]= pas[SP] to store the content of the top of the stack into an AR in the stack, located **L** levels down from the current AR.

Note: we are working at the CPU level therefore the instruction format must have only 3 fields. Any program whose number of fields in the instruction format is grater than 3 will get a zero.