University of Central Florida
Department of Computer Science
COP 3402: Systems Software
Fall 2021

Homework #1 (PM/0-Machine)

This project can be done solo OR with Group

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 four segments: the "text", which contains the instructions for the VM to execute, the "data" segment to store global variables, the "heap" to support programming languages which handle memory dynamically, and the "stack," to handle subroutine linkage and is organized as a data-stack. All these segments, in the process address space, are handled by the PM/0-CPU.

The PM/0 CPU has several registers to handle the process address space: The registers are named base pointer (BP), stack pointer (SP), program counter (PC), data pointer (DP), global data pointer (GP), the heap pointer (FREE), instruction counter (IC), and instruction register (IR). They will be explained in detail later on in this document. The stack grows downwards.

The Instruction Set Architecture (ISA) of PM/0 has 22 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 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 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 "data", "heap", and "stack" segments. You should be aware that the only segment were instructions are placed is in the "text segment.

Fetch Cycle:

In the Fetch cycle, an instruction is fetched from the "text" segment and placed in the IR register (IR ← pas[PC]) and the program counter is incremented by 3 to point to the next instruction to be executed (PC← PC + 3). PC is incremented by 3 because each instruction require 3 memory locations.

Execute Cycle:

In the Execute cycle, the instruction placed in IR, is executed by the VM/0-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 field of the instruction in the IR register (IR.M) is used to identify the operator and execute the appropriate arithmetic or relational instruction. (see chapter 2 in the textbook)

PM/0 Initial/Default Values:

The Process Address Space (PAS) represent the process memory and it is represented by an array of length 500. Initial values for PM/0 CPU registers will be set up once the program has been uploaded in the process address space. A variable called Instruction Counter (IC) will be used to load the program. Initial value for IC= 0. Each time an instruction is placed in the text segment, IC is incremented by 3. Once the program is uploaded, all others CPU registers can be set up. See illustration below.

```
GP = IC;                    //Global Pointer – Points to DATA segment
DP = IC -1                  //Data Pointer – To access variables in Main
FREE = IC + 40              // FREE points to Heap
BP = IC                     // Points to base of DATA or activation records
PC = 0;                     // Stack pointer – Points to top of stack
SP = MAX_PAS_LENGTH;
```

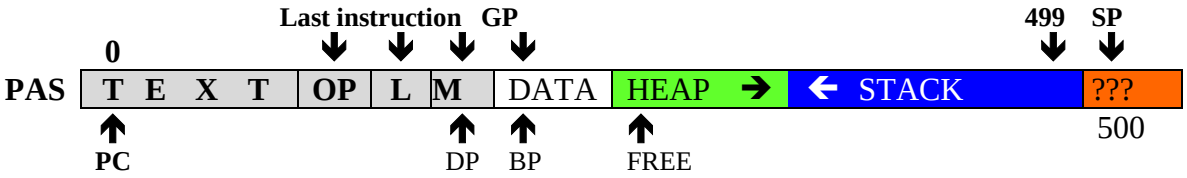
Initial process address space values are all zero:

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

Constant Values:

MAX_PAS_LENGTH is 500

The IR can be implemented in variety of ways including as a struct or as an array.
You will never be given an input file with more than 150 lines of code. However, as programs to be run in PM/0 have different lengths, The registers will be set up dynamically once the program had been uploaded. The figure bellow illustrates 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 Eustis3, for us it does not compile and receives a zero grade.
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 may only have three functions: main and base (Appendix D) and a print function if desired.**
   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.**

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”, “DP”, and “data” in that order.
- The second line should contain “Initial values: “ followed by the initial values of each register.
- After each line is executed two lines of output should be printed for the trace:
    - The first 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
        - The values of the following registers: PC, BP, SP, DP
        - The values of the data section printed from pas[gp] to pas[dp] inclusive.
    - The second line should print the stack starting with a header “stack : “ followed by the values printed from pas[MAX_PAS_LENGTH - 1] to pas[SP] inclusive.
- If you encounter the read instruction (9 0 2), print out “Please Enter an Integer: ” followed by a scanf, before 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.

Rubric:

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, return, and increase instructions are implemented correctly
5 – jump instructions are implemented correctly

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 (or DATA)
02 –	**OPR 0, M**	Operation to be performed on the data at the top of the stack. (Or in data segment)
03 –	**LOD L, M**	Load value to top of stack from the stack location at offset **M** from **L** lexicographical levels down (Or load a value into the data segment)
04 –	**STO L, M**	Store value at top of stack in the stack location at offset **M** from **L** lexicographical levels down (Or store a value into the data segment)
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). (Or increment dp)
07 –	**JMP 0, M**	Jump to instruction **M (PC ← M)**
08 –	**JPC 0, M**	Jump to instruction **M** if top stack or data element is 0
09 –	**SYS 0, 1**	Write the top stack element or data element to the screen
	SYS 0, 2	Read in input from the user and store it on top of the stack (or store it in data section)
	SYS 0, 3	End of program (**Set Halt flag to zero**)

Appendix B

ISA Pseudo Code

01 – **LIT 0, M**

```
if bp == gp then {dp ← dp + 1; pas[dp] ← M;}
             else {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

```
if bp == gp then pas[dp] ← -1 * pas[dp]
           else  pas[sp] ← -1 * pas[sp]
```

2 ADD

```
if bp == gp
then {dp ← dp - 1;
      pas[dp] ← pas[dp] + pas[dp + 1]}
else  {sp ← sp + 1;
       pas[sp] ← pas[sp] + pas[sp - 1]}
```

3 SUB

```
if bp == gp
then { dp ← dp - 1;
       pas[dp] ← pas[dp] - pas[dp + 1]}
else  {sp ← sp + 1;
       pas[sp] ← pas[sp] - pas[sp - 1]}
```

4 MUL

```
if bp == gp
then {dp ← dp - 1;
      pas[dp] ← pas[dp] * pas[dp + 1]}
else  {sp ← sp + 1;
       pas[sp] ← pas[sp] * pas[sp - 1]}
```

5 DIV

```
if bp == gp
then {dp ← dp - 1;
      pas[dp] ← pas[dp] / pas[dp + 1]}
else  {sp ← sp + 1;
       pas[sp] ← pas[sp] / pas[sp - 1]}
```

6 ODD

```
if bp == gp  then pas[dp] ← pas[dp] mod 2
           else  pas[sp]  ← pas[sp] mod 2
```

```
7    MOD     if bp == gp
             then {dp ←dsp - 1;
                   pas[dp] ← pas[dp] mod pas[dp + 1]}
             else  {sp ← sp + 1;
                    pas[sp] ← pas[sp] mod pas[sp - 1]}

8    EQL     if bp == gp
             then {dp ← dp - 1;
                   pas[dp] ← pas[dp] == pas[dp + 1]}
             else  {sp ← sp + 1;
                    pas[sp] ← pas[sp] == pas[sp - 1]}

9    NEQ     if bp == gp
             then {dp ← dp - 1;
                   pas[dp] ← pas[dp] != pas[dp + 1]}
             else  {sp ← sp + 1;
                    pas[sp] ← pas[sp] != pas[sp - 1]}

10   LSS     if bp == gp
             then {dp ← dp - 1;
                   pas[dp] ← pas[dp] < pas[dp + 1]}
             else  {sp ← sp + 1;
                    pas[sp] ← pas[sp] < pas[sp - 1]}

11   LEQ     if bp == gp
             then {dp ← dp - 1;
                   pas[dp] ← pas[dp] <= pas[dp + 1]}
             else  {sp ← sp + 1;
                    pas[sp] ← pas[sp] <= pas[sp - 1]}

12   GTR     if bp == gp
             then {dp ← dp - 1;
                   pas[dp] ← pas[dp] > pas[dp + 1]}
             else  {sp ← sp + 1;
                    pas[sp] ← pas[sp] > pas[sp - 1]}

13   GEQ     if bp == gp
             then {dp ← dp - 1;
                   pas[dp] ← pas[dp] >= pas[dp + 1]}
             else  {sp ← sp + 1;
                    pas[sp] ← pas[sp] >= pas[sp - 1]}
```

03 – **LOD L, M**

```
if bp  == gp then
        {dp ← dp + 1; pas[dp] ← pas[gp + M];}
else
        {if base(L) == gp then
                {sp ← sp - 1;  pas[sp] ← pas[gp + M];}
        else
                {sp ← sp - 1;  pas[sp] ← pas[base(L) - M];}}
```

04 – **STO L, M**

```
if bp == gp then
        {pas[gp + M] ← pas[dp]; dp = dp – 1;}
else
        {if base(L) == gp then
                {pas[gp + M] ← pas[sp]; sp = sp + 1;}
        else
                {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**

```
if bp == gp then dp ← dp + M
                 else  sp ← sp - M;
```

07 – **JMP 0, M**

```
pc ← M;
```

08 – **JPC 0, M**

```
if bp == gp then
        if pas[dp] == 0 then { pc ← M; }
        dp ← dp - 1;
else
        if pas[sp] == 0 then { pc ← M; }
        sp ← sp + 1;
```

09 – **SYS 0, 1**

```
if bp == gp then {printf("%d", pas[dp]); dp ← dp -1;}
                 else  {printf("%d", pas[sp]) ; sp ← sp + 1;}
```

SYS 0, 2

```
if bp == gp then {dp ← dp + 1; scanf("%d", pas[dp]);}
                 else  {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 3
1 0 4
1 0 3
2 0 4
4 1 0
1 0 14
3 1 0
2 0 10
8 0 39
1 0 7
7 0 42
1 0 5
2 0 0
6 0 1
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**	**3**	1	0	4	**1**	**0**	**3**	2	0	4	etc

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

OUTPUT FILE

Print out the execution of the program in the virtual machine, showing the stack and pc, bp, and sp.

```
                    PC   BP   SP   DP   data
Initial values:     0    60   500  59
 0   JMP  0    45   45   60   500  59
     stack :
15   INC  0    1    48   60   500  60   0
     stack :
Please Enter an Integer: 3
16   SYS  0    2    51   60   500  61   0 3
     stack :
17   CAL  0    6    6    499  500  61   0 3
     stack :
 2   INC  0    3    9    499  497  61   0 3
     stack : 60 60 54
 3   LIT  0    4    12   499  496  61   0 3
     stack : 60 60 54 4
 4   LIT  0    3    15   499  495  61   0 3
     stack : 60 60 54 4 3
 5   MUL  0    4    18   499  496  61   0 3
     stack : 60 60 54 12
 6   STO  1    0    21   499  497  61   12 3
     stack : 60 60 54
 7   LIT  0    14   24   499  496  61   12 3
     stack : 60 60 54 14
 8   LOD  1    0    27   499  495  61   12 3
     stack : 60 60 54 14 12
 9   LSS  0    10   30   499  496  61   12 3
     stack : 60 60 54 0
10   JPC  0    39   39   499  497  61   12 3
     stack : 60 60 54
13   LIT  0    5    42   499  496  61   12 3
     stack : 60 60 54 5
14   RTN  0    0    54   60   500  61   12 3
     stack :
Top of Stack Value: 3
18   SYS  0    1    57   60   500  60   12
     stack :
19   SYS  0    3    60   60   500  60   12
     stack :
```

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.

Function for printing:

```
void print_execution(int line, char *opname, int *IR, int PC, int BP, int SP, int DP, int *pas,
int GP)
{
      int i;
      // print out instruction and registers
      printf("%2d\t%s\t%d\t%d\t%d\t%d\t%d\t%d\t", line, opname, IR[1], IR[2], PC, BP,
SP, DP);

      // print data section
      for (i = GP; i <= DP; i++)
```

```
                printf("%d ", pas[i]);
        printf("\n");

        // print stack
        printf("\tstack : ");
        for (i = MAX_PAS_LENGTH - 1; i >= SP; i--)
                printf("%d ", pas[i]);
        printf("\n");
}
```