University of Central Florida
Department of Computer Science
COP 3402: Systems Software
Almalki, Spring 2022

Homework #1 (PM/0-Machine)

MAKE SURE YOU HAVE READ THE PROJECT OVERVIEW, ESPECIALLY THE PROJECT INSTRUCTIONS AND GUIDELINES SECTION

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 areas called the "stack," which is organized as a data-stack to store variables and which grows downwards, and the "text", which contains the instructions for the VM to execute. The machine also has a register file where arithmetic and logic instructions are executed. Additionally, the P-machine has several registers to handle the stack: 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 PM/0 follows the following instruction format. There are three fields: "OP L M". They are separated by one space.

OP is the operation code, specifying which instruction to execute
R indicates which register the operation should place output in or store from
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 list of instructions for the ISA can be found in Appendix A.

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 and register stacks. You should be aware that the only segment where 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 ← code[PC]) and the program counter is incremented to point to the next instruction to be executed.

Execute Cycle:
In the Execute cycle, the instruction placed in IR, is executed by the PM/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 LOD (IR.OP = 3), then the R filed indicate which register the instruction should load data to, the L field indicates which procedure the machine should load from, and the M field is used to identify the address of the variable within the procedure.

PM/0 Initial/Default Values:
The data stack is represented by an array of length 100. The register file is represented by an array of length 10. Initial values for PM/0 CPU registers will be set up once the program has been uploaded in the text segment.

```
BP = MAX_STACK_LENGTH - 1     // Base Pointer – Points to the base of the
                                  current AR
SP = BP + 1                   // Stack Pointer – Points to the top entry of
                                  the stack
PC = 0                        // Program Counter – Points to next
                                  instruction
IR                            // Instruction Register - holds the current
                                  instructioncan be an array or a struct
```

Initial stack and register file values are all zero

Constant Values:
REG_FILE_SIZE is 10
MAX_STACK_LENGTH is 100

You will never be given an input file with more than 150 lines of code.

Error Handling

There are two errors we need to check for:

1. Stack Overflow Error - when SP < 0, must be checked for in INC
2. Out of Bounds Access Error - when a LOD or STO instruction tries to access a value < 0 or >= MAX_STACK_LENGTH

When an error is found, print the error and set halt equal to true, then print the trace.

Assignment Instructions and Guidelines:

1. Source code named vm.c. If you have changed the structure of the package or altered provided files, you must leave an explanation in your readme and as a comment on your submission (you will lose points if you don't).
2. Instructions on how to use the package and author names in a readme text file. (If you didn't alter the package structure, your readme will just be author names)
3. Program output should be printed to the screen, and should follow the formatting of the examples If your output does not follow this format, you'll experience a delay in grading. We've given you a print function, so you shouldn't have any trouble
4. Only one submission per team. Student names should be written in the header comment of each source code file, in the readme, and in the comments of the submission

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

Rubric:

15	Compiles
20	Produces lines of meaningful execution (which clearly vary depending on the input file) before segfaulting, looping infinitely, or erroring on all inputs
5	Reading Instructions (readme.txt indicates if provided support files were changed, author names are in the header comment, readme, and submission comment)
5	Fetch cycle is implemented correctly
10	Arithmetic and logic instructions are implemented correctly
5	Read and write instructions are implemented correctly
15	Load and store instructions are implemented correctly
10	Call, return, and increase instructions are implemented correctly
5	jump instructions are implemented correctly
10	Correct error handling

Appendix A

Instruction Set Architecture (ISA)

Because the stack grows down, the end of it where the most recent values are is the "bottom".

ISA:

01	**LIT**	**R**	**0**	**M**	RF[**IR.R**] = **IR.M**
02	**RET**	**0**	**0**	**0**	Return from current procedure (X) to the last procedure (Y). SP = the index of the bottom of Y's AR BP = dynamic link value from X's AR PC = return address value from X's AR
03	**LOD**	**R**	**L**	**M**	Load value to register **IR.R** from the stack location at offset RF[**IR.M**] from **L** lexicographical levels up RF[**IR.R**] = stack[base(**L**) - RF[**IR.M**]] Before performing the load, check for Out of Bounds Access Error by checking if base(**L**) - **M** is less than zero or greater than or equal to MAX_STACK_LENGTH
04	**STO**	**R**	**L**	**M**	Store value from register **IR.R** to the stack location at offset RF[**IR.M**] from **L** lexicographical levels down stack[base(**L**) - RF[**IR.M**]] = RF[**IR.R**] Before performing the load, check for Out of Bounds Access Error by checking if base(**L**) - **M** is less than zero or greater than or equal to MAX_STACK_LENGTH
05	**CAL**	**0**	**L**	**M**	Call procedure at code index **M**. This will generate a new Activation Record. There are three values in the AR: 1st - static link = base(**L**) 2nd - dynamic link = BP 3rd - return address = PC After creating the activation record, BP = the index of the first entry of the new AR PC = **IR.M**
06	**INC**	**0**	**0**	**M**	Decrement SP by **M**, check for Stack Overflow Error which can occur if SP < 0 after the decrement
07	**JMP**	**0**	**0**	**M**	Jump to instruction **M**

08	**JPC**	**R**	**0**	**M**	Jump to instruction **M** if register **R** is 0
09	**WRT**	**R**	**0**	**0**	Print register **R**
10	**RED**	**R**	**0**	**0**	Register **R** equals scanf()
11	**HAL**	**0**	**0**	**0**	End of program (**Set Halt flag to true**)
12	**NEG**	**R**	**0**	**M**	Negate the register **R**
13	**ADD**	**R**	**L**	**M**	Add the registers **L** and **M** and store the result in register **R**
14	**SUB**	**R**	**L**	**M**	Subtract register **M** from register **L** and store the result in register **R**
15	**MUL**	**R**	**L**	**M**	Multiply registers **L** and **M** and store the result in register **R**
16	**DIV**	**R**	**L**	**M**	Divide register **L** by register **M** and store the result in register **R**
17	**MOD**	**R**	**L**	**M**	Set register **R** equal to register **L** modulo register **M**
18	**EQL**	**R**	**L**	**M**	If register **L** equals register **M**, set register **R** to 1. Otherwise set register **R** to 0
19	**NEQ**	**R**	**L**	**M**	If register **L** does not equal register **M**, set register **R** to 1. Otherwise set register **R** to 0
20	**LSS**	**R**	**L**	**M**	If register **L** is less than register **M**, set register **R** to 1. Otherwise set register **R** to 0
21	**LEQ**	**R**	**L**	**M**	If register **L** is less than or equal to register **M**, set register **R** to 1. Otherwise set register **R** to 0
22	**GTR**	**R**	**L**	**M**	If register **L** is greater than register **M**, set register **R** to 1. Otherwise set register **R** to 0
23	**GEQ**	**R**	**L**	**M**	If register **L** is greater than or equal to register **M**, set register **R** to 1. Otherwise set register **R** to 0

Appendix B

If the input is:

```
var x, y[4];
begin
     read y[1];
     y[3] := 4 * (7 + 13);
     if y[1] <> x ?
          y[0] := 2
     : y[2] := 0;
end.
```

The output should be:

```
                        PC   SP   BP
Initial values:                 0    100  99
 0   JMP  0    0    1    1    100  99        0 0 0 0 0 0 0 0 0 0
stack:
 1   INC  0    0    8    2    92   99        0 0 0 0 0 0 0 0 0 0
stack:   0 0 0 0 0 0 0 0
 2   LIT  0    0    1    3    92   99        1 0 0 0 0 0 0 0 0 0
stack:   0 0 0 0 0 0 0 0
 3   LIT  1    0    4    4    92   99        1 4 0 0 0 0 0 0 0 0
stack:   0 0 0 0 0 0 0 0
 4   ADD  0    0    1    5    92   99        5 4 0 0 0 0 0 0 0 0
stack:   0 0 0 0 0 0 0 0
Please Enter a Value: 8 *in the output file this won't be printed
 5   RED  1    0    0    6    92   99        5 8 0 0 0 0 0 0 0 0
stack:   0 0 0 0 0 0 0 0
 6   STO  1    0    0    7    92   99        5 8 0 0 0 0 0 0 0 0
stack:   0 0 0 0 0 8 0 0
 7   LIT  0    0    3    8    92   99        3 8 0 0 0 0 0 0 0 0
stack:   0 0 0 0 0 8 0 0
 8   LIT  1    0    4    9    92   99        3 4 0 0 0 0 0 0 0 0
stack:   0 0 0 0 0 8 0 0
 9   ADD  0    0    1    10   92   99        7 4 0 0 0 0 0 0 0 0
stack:   0 0 0 0 0 8 0 0
10   LIT  1    0    4    11   92   99        7 4 0 0 0 0 0 0 0 0
stack:   0 0 0 0 0 8 0 0
11   LIT  2    0    7    12   92   99        7 4 7 0 0 0 0 0 0 0
stack:   0 0 0 0 0 8 0 0
12   LIT  3    0    13   13   92   99        7 4 7 13 0 0 0 0 0 0
stack:   0 0 0 0 0 8 0 0
13   ADD  2    2    3    14   92   99        7 4 20 13 0 0 0 0 0 0
stack:   0 0 0 0 0 8 0 0
14   MUL  1    1    2    15   92   99        7 80 20 13 0 0 0 0 0 0
stack:   0 0 0 0 0 8 0 0
15   STO  1    0    0    16   92   99        7 80 20 13 0 0 0 0 0 0
```

```
stack:   0 0 0 0 0 8 0 80
16   LIT 0    0    1    17   92   99       1 80 20 13 0 0 0 0 0 0
stack:   0 0 0 0 0 8 0 80
17   LIT 1    0    4    18   92   99       1 4 20 13 0 0 0 0 0 0
stack:   0 0 0 0 0 8 0 80
18   ADD 0    0    1    19   92   99       5 4 20 13 0 0 0 0 0 0
stack:   0 0 0 0 0 8 0 80
19   LOD 0    0    0    20   92   99       8 4 20 13 0 0 0 0 0 0
stack:   0 0 0 0 0 8 0 80
20   LIT 1    0    3    21   92   99       8 3 20 13 0 0 0 0 0 0
stack:   0 0 0 0 0 8 0 80
21   LOD 1    0    1    22   92   99       8 0 20 13 0 0 0 0 0 0
stack:   0 0 0 0 0 8 0 80
22   NEQ 0    0    1    23   92   99       1 0 20 13 0 0 0 0 0 0
stack:   0 0 0 0 0 8 0 80
23   JPC 0    0    30   24   92   99       1 0 20 13 0 0 0 0 0 0
stack:   0 0 0 0 0 8 0 80
24   LIT 0    0    0    25   92   99       0 0 20 13 0 0 0 0 0 0
stack:   0 0 0 0 0 8 0 80
25   LIT 1    0    4    26   92   99       0 4 20 13 0 0 0 0 0 0
stack:   0 0 0 0 0 8 0 80
26   ADD 0    0    1    27   92   99       4 4 20 13 0 0 0 0 0 0
stack:   0 0 0 0 0 8 0 80
27   LIT 1    0    2    28   92   99       4 2 20 13 0 0 0 0 0 0
stack:   0 0 0 0 0 8 0 80
28   STO 1    0    0    29   92   99       4 2 20 13 0 0 0 0 0 0
stack:   0 0 0 0 2 8 0 80
29   JMP 0    0    35   35   92   99       4 2 20 13 0 0 0 0 0 0
stack:   0 0 0 0 2 8 0 80
35   HLT 0    0    0    36   92   99       4 2 20 13 0 0 0 0 0 0
stack:   0 0 0 0 2 8 0 80
```

For reference, the assembly code for this program is given next, but you don't need to worry about how it is generated in this assignment. HW3 will address code generation. We give input in PL/0 rather than assembly because it is significantly easier to write test cases in PL/0 than it is in assembly. To see the assembly for any input, run ./magic input.txt -a the -a tag tells the package to print the assembly

```
Line     OP Code OP Name R    L    M
0    7    JMP 0    0    1
1    6    INC 0    0    8
2    1    LIT 0    0    1
3    1    LIT 1    0    4
4    13   ADD 0    0    1
5    10   RED 1    0    0
6    4    STO 1    0    0
7    1    LIT 0    0    3
```

```
8   1   LIT 1   0   4
9   13  ADD 0   0   1
10  1   LIT 1   0   4
11  1   LIT 2   0   7
12  1   LIT 3   0   13
13  13  ADD 2   2   3
14  15  MUL 1   1   2
15  4   STO 1   0   0
16  1   LIT 0   0   1
17  1   LIT 1   0   4
18  13  ADD 0   0   1
19  3   LOD 0   0   0
20  1   LIT 1   0   3
21  3   LOD 1   0   1
22  19  NEQ 0   0   1
23  8   JPC 0   0   30
24  1   LIT 0   0   0
25  1   LIT 1   0   4
26  13  ADD 0   0   1
27  1   LIT 1   0   2
28  4   STO 1   0   0
29  7   JMP 0   0   35
30  1   LIT 0   0   2
31  1   LIT 1   0   4
32  13  ADD 0   0   1
33  1   LIT 1   0   0
34  4   STO 1   0   0
35  11  HLT 0   0   0
```

Appendix C

Helpful Tips

This function will be helpful to find a variable in a different Activation Record some **L** levels up:

```
/*********************************************/
/*          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 = data_stack[arb];
                L--;
        }
        return arb;
}
```

In the following example, the Red cells belong to the first instance of procedure B, the yellow cells belong to the first instance of procedure C, the green cells belong to the second instance of procedure C, the blue cells belong to the third instance of procedure C which is the current AR. Procedure C is a sub procedure of procedure B and calls itself (thus the multiple instances). Procedure B has two variables: x and y. Procedure C has one variable: k

	…	…	Further up the stack
	64	37	Static link of B
	65	37	Dynamic link of B
	66	68	Return address of B
	67		x
	68		y
	69	64	Static link of 1st instance of C

	70	64	Dynamic link of 1st instance of C
	71	31	Return address of 1st instance of C
	72		k
	73	64	Static link of 2nd instance of C
	74	69	Dynamic link of 2nd instance of C
	75	18	Return address of 2nd instance of C
	76		k
BP	77	64	Static link of 3rd instance of C
	78	73	Dynamic link of 3rd instance of C
	79	18	Return address of 3rd instance of C
SP	80		k

Interesting things to notice:

- B's static link is equal to it's dynamic link. This means it is a sub procedure of the same procedure that called it.
- All three instances of C have the same static link. This is because they are all sub procedures of B, so their static link points to B
- All the dynamic links point to the base of the last AR
- The 2nd and 3rd instance of C have the same return address, but the 1st instance has a different one. The return address points to a code index. The 2nd and 3rd instances were called from within the C procedure, so their return addresses point to the same instruction within C. But the first instance was called from procedure B