HW #1 (Virtual Machine)

COP 3402: Systems Software

Spring 2026

Instructor: Jie Lin, Ph.D.

Due Date: **Monday, February 9th, 2026**

Last updated: January 25, 2026

Disclaimer: This document may not cover all possible scenarios—when in doubt, ask the instructor or a TA.

All official updates (test cases, clarifications, etc.) will be posted as **Webcourses announcements**.

Check Webcourses regularly for critical updates.

Pseudocode note: The pseudocode in Appendix A is a starting point and may not be 100% aligned with the required implementation. Your responsibility is to understand the ISA first, then adapt details as needed before coding.

Contents

1 Assignment Overview

In this assignment you will work either individually or in a two-person team to implement a *virtual machine* called the **P-machine** (also referred to as the PM/0). The P-machine is a simple stack machine used to execute programs composed of a small instruction set. Your job is to read an input file containing P-machine instructions, store them in memory, and interpret them exactly according to the specifications described below. **This assignment must be implemented in C only; no other language will be permitted.** Throughout execution the machine maintains a program counter, base pointer and stack pointer while manipulating a contiguous address space consisting of a *text segment* at the bottom for instructions and a *stack segment* for runtime data near the top, with a reserved unallocated region at the top of PAS (addresses 499 down to 481). The stack grows downward.

At a high level your implementation must:

1. **Understand the architecture.** Read the description of the P-machine architecture, including how its process address space is organized and how the stack grows. Familiarize yourself with the available registers (PC, BP, SP, and the instruction register IR).

2. **Implement the fetch-execute cycle.** Write a loop that repeatedly fetches the next instruction from the text segment (three integers per instruction), *increments* the program counter appropriately and then executes the instruction by manipulating the stack and registers. See the P-machine Review for details.

3. **Handle all instruction types.** The PM/0 instruction set includes LIT, OPR, LOD, STO, CAL, INC, JMP, JPC and SYS instructions. Each must be implemented exactly as described, without adding new opcodes or changing the instruction format. Appendix A lists all opcodes and their meanings.

4. **Run on Eustis using ANSI C (C only).** Your program must be written in standard C, compiled with `gcc` using the `-Wall` flag and run correctly on the university server *Eustis*. If your code runs on your personal machine but not on Eustis, it will be graded as not working.

5. **Follow the input and output formats.** Your program must read a single input file specified on the command line (see Section 2). It must not prompt for the filename. The output must follow the example shown in Appendix B: print the values of the program counter, base pointer and stack pointer after each instruction and display the current contents of the stack, separating activation records with vertical bars.

6. **Required documentation.** Include the required header comment in your `vm.c` source file (see Section 5.1). Do not change the ISA, do not add instructions, and do not change the format of the input; otherwise your grade will be zero.

The remainder of this document provides the detailed specification, input/output formats, an overview of the P-machine architecture, hints for implementation and sample input/output. Refer back to it regularly as you develop your solution.

2 Command Line Parameters

Invoke your program from the terminal on the **Eustis** server. The executable must accept **exactly one** parameter:

1. **Input file.** A text file containing the P-machine program to execute. Each line must have exactly three integers: the opcode (OP), the lexicographical level (L) and the M field, separated by whitespace. Lines may not contain comments or extra fields.

Do *not* prompt the user for the filename—if the argument count is incorrect, print an error and exit. Input and output formats are shown in Appendix B.

The assignment specifies strict formats for the input and output.

2.1 Input File

The input to your virtual machine is a plain-text file whose contents define the program to run. Each line represents exactly one instruction and consists of three integers OP, L and M, separated by one or more spaces or tab characters. Your program must automatically split each line into OP, L and M tokens and store them into the correct PAS indexes.

The three numbers must correspond to a valid opcode and its parameters as defined in Appendix A. Your program should read the entire file, split each line into an $\langle OP, L, M\rangle$ triple, and store each triple in the process address space starting from address **0** and moving upward (see Section 3): the first instruction's fields are placed at addresses $0 \to \mathrm{OP}_0,\ 1 \to L_0,\ 2 \to M_0$; the second at $3 \to \mathrm{OP}_1,\ 4 \to L_1,\ 5 \to M_1$; and so on until EOF.

2.2 Output Format

For each instruction executed, print the current values of the program counter (PC), base pointer (BP) and stack pointer (SP) followed by the contents of the stack from the top of the stack down to the base pointer. Separate activation records with a vertical bar (`|`). The first line printed should display the initial register values before any instruction executes. When executing `SYS 0 1` you must print the top of the stack to standard output and then pop it. When executing `SYS 0 2` you must prompt the user to enter an integer exactly as shown in the sample output ("`Please Enter an Integer: `") and push the integer onto the stack. The `SYS 0 3` instruction halts the program and should be printed as the last instruction executed.

See Appendix B for an example of the exact formatting.

3 P-Machine Review

The P-machine is a simple stack machine with a single process address space (PAS) of fixed size **500**. **You must create a static array of 500 integers in your program to represent the entire PAS, with all entries initialized to zero.** The PAS contains a text segment, a stack segment, and a reserved unallocated region:

- **Text segment (code):** stored at the *bottom* of PAS and growing *upward*. The first instruction occupies addresses $0 \to \mathrm{OP}_0,\ 1 \to L_0,\ 2 \to M_0$; the second occupies $3, 4, 5$; etc.
- **Stack segment (data):** begins at address **480** and grows *downward* as values are pushed. All runtime data are stored here.
- **Unallocated region (reserved):** addresses **499** down to **481** are unused.

After loading the code from the input file into the text segment, initialize registers as follows: set **PC** = 0 prior to the first fetch; set **BP** = 480; set **SP** = 481 (so the first push decrements into the stack).

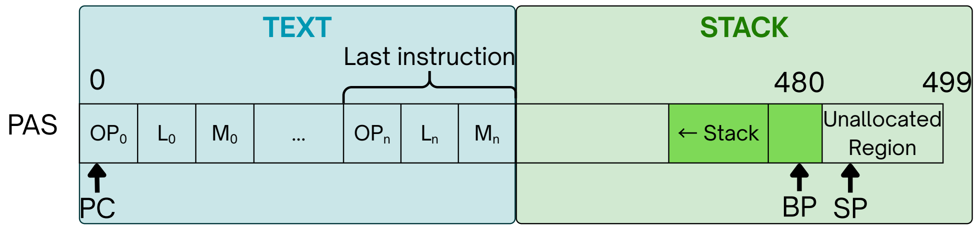

Figure 1: PAS layout: text segment from 0 upward; stack segment grows downward from 480; reserved region 499–481. PC starts at 0; BP=480 and SP=481.

3.1 Registers

The PM/0 CPU maintains the following registers:

- **PC (Program Counter):** points to the next instruction in the text segment.
- **BP (Base Pointer):** points to the base of the current activation record on the stack.
- **SP (Stack Pointer):** points to the top of the stack. The stack grows downward (decrementing SP) when values are pushed and upward when values are popped.
- **IR (Instruction Register):** holds the OP, L, M fields of the instruction currently being executed.

3.2 Instruction Format

Each instruction consists of three integer fields:

1. **OP:** the operation code specifying the instruction to execute (LIT, OPR, LOD, STO, CAL, INC, JMP, JPC, SYS).
2. **L:** the lexicographical level for instructions that access variables in other activation records.
3. **M:** a parameter whose meaning depends on the opcode. It may be a literal value, an address in the text segment, an offset within an activation record or a sub-opcode for arithmetic and logical operations.

3.3 Fetch-Execute Cycle

The virtual machine repeatedly performs the following two steps until a `SYS 0 3` instruction is encountered:

Fetch cycle: Copy the instruction at address PC in the text segment into the IR and *increment* PC by 3 (since each instruction occupies three slots and code is stored upward). In pseudocode:

```
IR.OP ← PAS[PC]
IR.L ← PAS[PC+1]
IR.M ← PAS[PC+2]
PC ← PC + 3
```

Execute cycle: Examine IR.OP and perform the operation corresponding to that opcode, modifying SP, BP and/or PC and the stack as appropriate. See Appendix A for a full specification of each opcode.

At startup, load the text segment from the input file as described, then set PC = 0, BP = 480, and SP = 481. The PAS array is initially filled with zeros. The reserved region 499–481 remains unused. Remember that the stack grows downward: pushing decreases SP, and popping increases it.

3.4 Activation Records and the Base Function

Procedures create *activation records* on the stack. Each activation record contains three words reserved for the static link (SL), dynamic link (DL) and return address (RA), followed by the procedure's local variables. The `CAL` instruction uses the lexicographical level to determine how far down the static chain to traverse when linking to the called procedure's enclosing scope. A helper function `base(bp, L)` can be used to find the base pointer of the activation record L levels down from the current record. In C it could be written as:

Base function example

```
/* Find base L levels down from the current activation record */
int base(int BP, int L) {
    int arb = BP;        // activation record base
    while (L > 0) {
        arb = pas[arb];  // follow static link
        L--;
    }
    return arb;
}
```

Keep in mind that **no dynamic memory allocation**, **no pointer arithmetic**, and **no function-like macros** are allowed. If you use any of these or implement instructions using separate functions, your program will receive a score of zero.

4 Build and Execution

Compile your program on Eustis using the C compiler. Only C is accepted for this assignment. Use the `-Wall` flag and adhere strictly to ANSI C (C11 is recommended). Example commands:

Compilation Commands

```
# Compile and run on Eustis
gcc -O2 -Wall -std=c11 -o vm vm.c
./vm input.txt
```

5 Submission Instructions

Submit your work on **Webcourses**. Programs are compiled and tested on **Eustis**. Follow these rules to avoid deductions.

5.1 Code Requirements

- **Program name.** Name your program `vm.c`. The compiled executable should therefore be named `vm`.
- **Command line.** Accept exactly one argument (the input file). If a user provides a different number of arguments, print an error message and exit.
- **Header comment.** Place the following non-breaking box at the top of your `vm.c` file. It will not split across pages and lines will not wrap inside the box.

Required Header Comment

```
/*
Assignment:
vm.c - Implement a P-machine virtual machine

Authors: <Your Name(s) Here>

Language: C (only)

To Compile:
  gcc -O2 -Wall -std=c11 -o vm vm.c

To Execute (on Eustis):
  ./vm input.txt

where:
  input.txt is the name of the file containing PM/0 instructions;
  each line has three integers (OP L M)

Notes:
  - Implements the PM/0 virtual machine described in the homework
    instructions.
  - No dynamic memory allocation, no pointer arithmetic, and
    no function-like macros.
  - Does not implement any VM instruction using a separate function.
  - Runs on Eustis.

Class: COP 3402 - Systems Software - Spring 2026

Instructor : Dr. Jie Lin

Due Date: Monday, February 9th, 2026
*/
```

- **Commenting.** Include comments in your source code explaining the purpose of variables, major code blocks and edge cases. A well-commented program earns extra credit; an uncommented one receives zero.
- **No dynamic memory, pointer arithmetic, or function-like macros.** Apart from file handles and the stack links described above, dynamic memory allocation (`malloc`, `calloc`, `realloc`, etc.), pointer arithmetic, and function-like macros are forbidden. If used, your grade will be zero.
- **Function count limit.** Your submission may define at most **three** functions: `main`, the provided `base` helper, and a single print function (e.g., to print the stack/trace). If your program contains more than three functions, your grade will be **0**.
- **No instruction functions.** Do not implement each VM instruction as a separate function. Your main loop should implement the fetch-execute cycle directly. You may

write the `base()` helper for static link traversal and one auxiliary print function, but instruction logic must remain in the main loop.

5.2 What to Submit

- Your source code (**can only be .c**).
- **If you used AI:** Submit the AI Usage Disclosure Form *and* a separate markdown file describing your AI usage.
- **If you did not use AI:** Submit no additional files.

5.3 Submission Guidelines

- Submit on Webcourses before the due date. Late submissions incur penalties as described below. Resubmissions are not accepted after two days.
- Your program should not write to any files. All output must go to standard output exactly as specified.
- Do not modify the PM/0 instruction set or input format. Changing the ISA (e.g., adding an opcode or altering the number of fields) will result in a zero.
- Only C is allowed for this assignment.

6 Academic Integrity, AI Usage and Plagiarism Policy

6.1 AI Usage Disclosure

If you use AI tools while completing this assignment, you must disclose this usage. Complete the **AI Usage Disclosure Form** provided with this assignment. If you used AI, include a separate markdown file describing:

- The name and version of the AI tool used.
- The dates used and specific parts of the assignment where the AI assisted.
- The prompts you provided and a summary of the AI output.
- How you verified the AI output against other sources and your own understanding.
- Reflections on what you learned from using the AI.

If you used AI and did not submit both the disclosure form and the markdown file, it will be treated as plagiarism and will result in a 0 for this submission.

6.2 Plagiarism Detection and Writing Standards

All submissions will be processed through plagiarism detection tools (e.g., JPlag). If the similarity score between your submission and others exceeds a threshold, your code will be considered plagiarized and you will receive an F for the course. This applies regardless of whether you used AI and regardless of whether you submitted an AI disclosure form. **If AI usage is detected and the required AI disclosure form/markdown are missing, the submission will be treated as plagiarism and receive a zero.**

While AI tools may assist with brainstorming or high-level guidance, the final submission must represent your own work and understanding. **Do not copy and paste AI-generated code.** AI tends to produce highly similar solutions across students; use it only to understand ideas, then implement the code yourself. Do not copy previous semester solutions or share your code with others.

7 Submission Deadlines and Late Policy

All deadlines use U.S. Eastern Time (Orlando, FL). The Webcourses submission timestamp is authoritative. A submission is considered late if it is uploaded after the posted due date/time. Late submissions are accepted for up to 48 hours after the due date, subject to the following point penalties (deductions are applied to the assignment's maximum score, not a percentage):

- 0:00:01-12:00:00 late → −5 points
- 12:00:01-24:00:00 late → −10 points
- 24:00:01-36:00:00 late → −15 points
- 36:00:01-48:00:00 late → −20 points
- After 48:00:00 → Not accepted; recorded as missed (0 points)

Resubmissions after 48 hours are not accepted.

8 Grading

Your assignment will be graded based on both functionality and strict compliance with the specification. Hidden tests will exercise every instruction and error path. The grading policy below is intentionally strict.

- **-100 points (automatic zero):** Program does not compile on Eustis or cannot be built using the provided commands in Section 4.
- **Immediate Zero:**
  - Plagiarism.

    - Using AI without the required disclosure form and markdown file, or evidence of direct copying from AI/another student (including very high similarity).
    - Changing the instruction set (ISA) or input/output format.
    - Using dynamic memory allocation (`malloc`, `calloc`, `realloc`, etc.).
    - Using pointer arithmetic.
    - Using function-like macros (e.g., `#define F(x) ...`).
    - Implementing VM instructions as separate functions.
    - More than three functions defined (only `main`, `base`, and one print function are allowed).
    - Omitting the required header comment or failing to update it with the correct author name(s).
    - Failing to accept command-line arguments as required.
    - Producing no output in the terminal after a successful compile.
    - Writing any output to a file.

- **-10 points:** Fetch cycle or fetch-execute order implemented incorrectly (PC/IR updates out of order).
- **-5 points per error:** Each incorrectly implemented instruction or operation detected during grading.
- **-5 points (formatting):** Major deviations from Appendix B output format (e.g., missing activation-record separator bar). Minor alignment differences (e.g., spacing of bars) are not penalized.

If additional mistakes are discovered during grading that are not listed above, we reserve the right to deduct points. As a baseline, **at least 5 points may be deducted for each mistake found**. Not all situations are covered in the grading policy.

A Instruction Set Architecture (ISA)

The PM/0 supports nine opcodes. Each instruction is encoded by a three-number tuple $\langle OP, L, M\rangle$. The tables below summarize each opcode along with a brief description and pseudocode. See Table 2 for `OPR` sub-operations.

Pseudocode note: The pseudocode in Appendix A is a starting point and may not be 100% aligned with the required implementation. Your responsibility is to understand the ISA first, then adapt details as needed before coding.

Table 1: PM/0 Instruction Set (Core)

Opcode	Mnemonic	L	M	Description	Pseudocode
01	LIT	0	n	**Push literal.**	`sp ← sp +1` `pas[sp] ← ` n
02	OPR	0	m	**Operation code.**	*See OPR Table 2 for operation details*

03	LOD	n	a	**Load to stack.**	`sp ← sp +1`
					`pas[sp] ← pas[base(bp,`n`) +`a`]`

04	STO	n	o	**Store from stack.**	`pas[base(bp,`n`) +`o`] ← pas[sp]`
					`sp ← sp −1`

05	CAL	n	a	**Call procedure.**	`pas[sp+1] ← base(bp,`n`)`
					`pas[sp+2] ← bp`
					`pas[sp+3] ← pc`
					`bp ← sp+1`
					`pc ← ` a

06	INC	0	n	**Allocate locals.**	`sp ← sp +`n
07	JMP	0	a	**Unconditional jump**	`pc ← ` a
08	JPC	0	a	**Conditional jump.**	`if pas[sp] = 0 then pc ← ` a `sp ← sp −1`

09	SYS	0	1	**Output integer.**	`print(pas[sp])`
					`sp ← sp −1`

09	SYS	0	2	**Read integer.**	`sp ← sp +1`
					`pas[sp] ← readInt()`

09	SYS	0	3	**Halt the program.**	`halt`

Table 2: PM/0 Arithmetic and Relational Operations (OPR, opcode 02, L=0)

Opcode	**Mnemonic**	**L**	**M**	**Description**	**Pseudocode**

02	RTN	0	0	**Return from subroutine.**	`sp ← bp −1`
					`bp ← pas[sp+2]`
					`pc ← pas[sp+3]`

02	NEG	0	1	**Negation (unary minus).**	`pas[sp] ← −pas[sp]`

02	ADD	0	2	**Addition.**	`pas[sp−1] ← pas[sp−1] + pas[sp]`
					`sp ← sp −1`

02	SUB	0	3	**Subtraction.**	`pas[sp−1] ← pas[sp−1] − pas[sp]`
					`sp ← sp −1`

02	MUL	0	4	**Multiplication.**	`pas[sp−1] ← pas[sp−1] * pas[sp]`
					`sp ← sp −1`

02	DIV	0	5	**Integer division.**	`pas[sp−1] ← pas[sp−1] / pas[sp]`
					`sp ← sp −1`

02	EQL	0	6	**Equality comparison**	`pas[sp−1] ← (pas[sp−1] == pas[sp])`
					`sp ← sp −1`

02	NEQ	0	7	**Inequality comparison**	`pas[sp−1] ← (pas[sp−1] ≠ pas[sp])`
					`sp ← sp −1`

02	LSS	0	8	**Less-than comparison**	`pas[sp−1] ← (pas[sp−1] < pas[sp])`
					`sp ← sp −1`

02	LEQ	0	9	**Less-or-equal comparison**	`pas[sp−1] ← (pas[sp−1] ≤ pas[sp])`
					`sp ← sp −1`

02	GTR	0	10	**Greater-than comparison**	`pas[sp−1] ← (pas[sp−1] > pas[sp])`
					`sp ← sp −1`

02	GEQ	0	11	**Greater-or-equal comparison**	`pas[sp−1] ← (pas[sp−1] ≥ pas[sp])`
					`sp ← sp −1`

B Sample Inputs and Outputs

B.1 Sample Input File

The following is the sample PM/0 program used in examples. Each line has three integers corresponding to OP, L and M.

Sample Input Content

```
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 8
8 0 39
1 0 7
7 0 42
1 0 5
2 0 0
6 0 5
9 0 2
5 0 3
9 0 1
9 0 3
```

B.2 Sample Program Output

The console output for the sample program illustrates formatting of PC, BP, SP and the stack after each instruction. Activation records are separated by a vertical bar (|).

Sample Execution Trace

```
        L       M    PC   BP   SP   stack
Initial values:      0    480  481
JMP     0       45   45   480  481
INC     0       5    48   480  476  0 0 0 0 0
Please Enter an Integer: 8
SYS     0       2    51   480  475  0 0 0 0 0  8
CAL     0       3    3    474  475  0 0 0 0 0  8
JMP     0       6    6    474  475  0 0 0 0 0  8
INC     0       4    9    474  471  0 0 0 0 0  8 | 480 480 54 0
LIT     0       4    12   474  470  0 0 0 0 0  8 | 480 480 54 0 4
LIT     0       3    15   474  469  0 0 0 0 0  8 | 480 480 54 0 4  3
MUL     0       4    18   474  470  0 0 0 0 0  8 | 480 480 54 0 12
STO     1       4    21   474  471  0 0 0 0 12 8 | 480 480 54 0
LIT     0       14   24   474  470  0 0 0 0 12 8 | 480 480 54 0 14
LOD     1       4    27   474  469  0 0 0 0 12 8 | 480 480 54 0 14 12
LSS     0       8    30   474  470  0 0 0 0 12 8 | 480 480 54 0 0
JPC     0       39   39   474  471  0 0 0 0 12 8 | 480 480 54 0
LIT     0       5    42   474  470  0 0 0 0 12 8 | 480 480 54 0 5
RTN     0       0    54   480  475  0 0 0 0 12 8
Output result is: 8
SYS     0       1    57   480  476  0 0 0 0 12
SYS     0       3    60   480  476  0 0 0 0 12
```