# HW #1 (Virtual Machine)

COP 3402: Systems Software

Fall 2025

Instructor: Jie Lin, Ph.D.

Due Date: **Friday, September 12th, 2025**

Last updated: September 3, 2025

> **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.

## 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* for instructions and a *stack segment* for data (no unused region). 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), *decrements* 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 6.1). 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 and store each triple in the process address space starting from address **499** and moving downward (see Section 3): the first instruction's fields are placed at addresses $499 \to \text{OP},\ 498 \to L,\ 497 \to M$; the second at $496 \to \text{OP},\ 495 \to L,\ 494 \to M$; 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 two segments (no unused region):

- **Text segment (code):** stored at the *top* of PAS and growing *downward*. The first instruction occupies addresses $499 \to \text{OP},\ 498 \to L,\ 497 \to M$; the second occupies $496, 495, 494$; etc.
- **Stack segment (data):** begins just below the code and grows *downward* as values are pushed.

After loading the code from the input file into the text segment, initialize registers as follows: set **PC** $= 499$ prior to the first fetch; set **SP** to the address of the last $M$ word loaded (i.e., the lowest address used by code); set **BP** $= \text{SP} - 1$.

### 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 $\text{OP}, L, M$ fields of the instruction currently being executed.

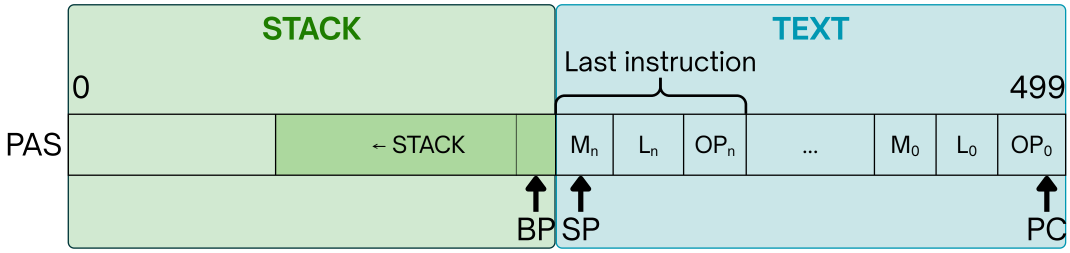


Figure 1: PAS layout: text segment from 499 downward; stack segment grows downward. PC starts at 499; after loading, SP is the last M index and BP=SP−1.

### 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 *decrement* PC by 3 (since each instruction occupies three slots and code is stored downward). 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 = 499, set SP to the last loaded $M$ address, and set BP = SP − 1. The PAS array is initially filled with zeros. 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 and no pointer arithmetic are allowed. If any instruction is implemented in your code using a separate function or if you use dynamic memory, 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 What We Are Providing

You are given this instruction document as guidance. There are no separate test files or automated scripts provided for this project; instead, Appendix B includes a sample input file and the corresponding execution trace. Use it as a reference to verify your program's formatting and semantics. Your grader will compile and run your program on a suite of hidden tests following the exact specification described herein.

# 6 Submission Instructions

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

## 6.1 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 or pointer arithmetic.
  - Does not implement any VM instruction using a separate function.
  - Runs on Eustis.

Class: COP 3402 - Systems Software - Fall 2025

Instructor : Dr. Jie Lin

Due Date: Friday, September 12th, 2025
*/
```

- **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 pointers/dynamic memory.** Apart from file handles and the stack links described above, pointer usage and dynamic memory (`malloc`, `calloc`, `realloc`, etc.) 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.

### 6.2 What to Submit

- Your source code (**can only be .c**).
- Team contribution sheet (**signed**).
- The AI Usage Disclosure Form with your signature.
- **If you used AI:** A separate markdown file describing your AI usage.
- **If you did not use AI:** Only the signed disclosure form is needed.

### 6.3 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. C++ and Rust are not permitted here.

## 7 Academic Integrity, AI Usage and Plagiarism Policy

### 7.1 AI Usage Disclosure

If you plan to 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 did not use any AI tools, check the appropriate box on the form. Submit the signed form and the markdown file (if applicable) along with your assignment. Failure to disclose AI usage will be treated as academic dishonesty.

### 7.2 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. While AI tools may assist with brainstorming, the final submission must represent your own work and understanding. Do not copy previous semester solutions or share your code with others.

## 8 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.

## 9 Grading

Your assignment will be graded based on both functionality and compliance with the specification. Hidden tests will exercise every instruction and error path. Roughly, the grading rubric is:

- **-100 points:** Program does not compile on Eustis or cannot be built via the provided commands.
- **Immediate Zero:** Plagiarism, changing the instruction set, using dynamic memory or pointer references, implementing instructions as separate functions, omitting the required header comment, more than three functions defined (only `main`, `base`, and one print function are allowed), or source is not in C.
- **10 points:** Program compiles successfully and runs.
- **25 points:** Program produces meaningful output for some instructions before crashing or looping infinitely.

- **5 points:** Accepts exactly one argument and prints output to the console or command line (no file output or saving).
- **5 points:** Required header comment present (with author names and compile/run instructions) in `vm.c`; builds on Eustis using `gcc`.
- **5 points:** Fetch cycle implemented correctly (correctly updates PC and IR).
- **5 points:** Well-commented source code.
- **10 points:** All OPR operations implemented correctly.
- **10 points:** SYS 0 1 and SYS 0 2 implemented correctly.
- **10 points:** Load and store instructions (LOD, STO) implemented correctly.
- **10 points:** Call and return instructions (CAL, RTN) implemented correctly.
- **5 points:** Follows formatting guidelines (output matches sample) and source code is named `vm.c`.

Conversely, programs that pass all tests with no errors will earn a perfect score.

# 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.

# 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 $M$.

Table 1: PM/0 Instruction Set (Core)

| Opcode | OP Mnemonic | L | M | Description & Pseudocode |
|---|---|---|---|---|
| 01 | LIT | 0 | $n$ | **Literal push.**<br>`sp` $\leftarrow$ `sp` $-1$<br>`pas[sp]` $\leftarrow$ $n$ |
| 02 | OPR | 0 | $m$ | **Operation code;** see Table 2 for specific operations.<br>*See OPR table for operation details* |
| 03 | LOD | $n$ | $a$ | **Load value to top of stack** from offset $a$ in the AR $n$ static levels down.<br>`sp` $\leftarrow$ `sp` $-1$<br>`pas[sp]` $\leftarrow$ `pas[base(bp,`$n$`)` $-a$`]` |
| 04 | STO | $n$ | $o$ | **Store top of stack** into offset $o$ in the AR $n$ static levels down.<br>`pas[base(bp,`$n$`)` $-o$`]` $\leftarrow$ `pas[sp]`<br>`sp` $\leftarrow$ `sp` $+1$ |
| 05 | CAL | $n$ | $a$ | **Call procedure** at code address $a$; create activation record.<br>`pas[sp`$-1$`]` $\leftarrow$ `base(bp,`$n$`)`<br>`pas[sp`$-2$`]` $\leftarrow$ `bp`<br>`pas[sp`$-3$`]` $\leftarrow$ `pc`<br>`bp` $\leftarrow$ `sp`$-1$<br>`pc` $\leftarrow$ $a$ |
| 06 | INC | 0 | $n$ | **Allocate** $n$ **locals** on the stack.<br>`sp` $\leftarrow$ `sp` $-n$ |
| 07 | JMP | 0 | $a$ | **Unconditional jump** to address $a$.<br>`pc` $\leftarrow$ $a$ |
| 08 | JPC | 0 | $a$ | **Conditional jump:** if value at top of stack is 0, jump to $a$; pop the stack.<br>`if pas[sp] = 0 then pc` $\leftarrow$ $a$<br>`sp` $\leftarrow$ `sp` $+1$ |
| 09 | SYS | 0 | 1 | **Output integer** value at top of stack; then pop.<br>`print(pas[sp])`<br>`sp` $\leftarrow$ `sp` $+1$ |
| 09 | SYS | 0 | 2 | **Read an integer** from stdin and push it.<br>`sp` $\leftarrow$ `sp` $-1$<br>`pas[sp]` $\leftarrow$ `readInt()` |
| 09 | SYS | 0 | 3 | **Halt the program.**<br>`halt` |

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

| Opcode | OP Mnemonic | L | M | Description & Pseudocode |
|---|---|---|---|---|
| 02 | RTN | 0 | 0 | **Return from subroutine** and restore caller's AR.<br>`sp ← bp +1`<br>`bp ← pas[sp−2]`<br>`pc ← pas[sp−3]` |
| 02 | ADD | 0 | 1 | **Addition.**<br>`pas[sp+1] ← pas[sp+1] + pas[sp]`<br>`sp ← sp +1` |
| 02 | SUB | 0 | 2 | **Subtraction.**<br>`pas[sp+1] ← pas[sp+1] − pas[sp]`<br>`sp ← sp +1` |
| 02 | MUL | 0 | 3 | **Multiplication.**<br>`pas[sp+1] ← pas[sp+1] * pas[sp]`<br>`sp ← sp +1` |
| 02 | DIV | 0 | 4 | **Integer division.**<br>`pas[sp+1] ← pas[sp+1] / pas[sp]`<br>`sp ← sp +1` |
| 02 | EQL | 0 | 5 | **Equality comparison** (result 0/1).<br>`pas[sp+1] ← (pas[sp+1] == pas[sp])`<br>`sp ← sp +1` |
| 02 | NEQ | 0 | 6 | **Inequality comparison** (result 0/1).<br>`pas[sp+1] ← (pas[sp+1] ≠ pas[sp])`<br>`sp ← sp +1` |
| 02 | LSS | 0 | 7 | **Less-than comparison** (result 0/1).<br>`pas[sp+1] ← (pas[sp+1] < pas[sp])`<br>`sp ← sp +1` |
| 02 | LEQ | 0 | 8 | **Less-or-equal comparison** (result 0/1).<br>`pas[sp+1] ← (pas[sp+1] ≤ pas[sp])`<br>`sp ← sp +1` |
| 02 | GTR | 0 | 9 | **Greater-than comparison** (result 0/1).<br>`pas[sp+1] ← (pas[sp+1] > pas[sp])`<br>`sp ← sp +1` |
| 02 | GEQ | 0 | 10 | **Greater-or-equal comparison** (result 0/1).<br>`pas[sp+1] ← (pas[sp+1] ≥ pas[sp])`<br>`sp ← sp +1` |

Sample Input Content

```
7 0 45
7 0 6
6 0 4
1 0 4
1 0 3
2 0 3
4 1 4
1 0 14
3 1 4
2 0 7
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:       499   439   440
JMP     0       45    454   439   440
INC     0       5     451   439   435   0 0 0 0 0
Please Enter an Integer: 8
SYS     0       2     448   439   434   0 0 0 0 0  8
CAL     0       3     496   433   434   0 0 0 0 0  8
JMP     0       6     493   433   434   0 0 0 0 0  8
INC     0       4     490   433   430   0 0 0 0 0  8 | 439 439 445 0
LIT     0       4     487   433   429   0 0 0 0 0  8 | 439 439 445 0 4
LIT     0       3     484   433   428   0 0 0 0 0  8 | 439 439 445 0 4  3
MUL     0       3     481   433   429   0 0 0 0 0  8 | 439 439 445 0 12
STO     1       4     478   433   430   0 0 0 0 12 8 | 439 439 445 0
LIT     0       14    475   433   429   0 0 0 0 12 8 | 439 439 445 0 14
LOD     1       4     472   433   428   0 0 0 0 12 8 | 439 439 445 0 14 12
LSS     0       7     469   433   429   0 0 0 0 12 8 | 439 439 445 0 0
JPC     0       39    460   433   430   0 0 0 0 12 8 | 439 439 445 0
LIT     0       5     457   433   429   0 0 0 0 12 8 | 439 439 445 0 5
RTN     0       0     445   439   434   0 0 0 0 12 8
Output result is: 8
SYS     0       1     442   439   435   0 0 0 0 12
SYS     0       3     439   439   435   0 0 0 0 12
```