# University of Central Florida

## Department of Computer Science

## COP 3402: Systems Software

## Fall 2022

**Homework #1 (PM/0-Machine)**

**Course Project Overview:**

This semester you will be implementing a compiler and virtual machine for a mini-language PM/0. This project will be a multi-file C program. There are two support files we will give you (a driver to read in the initial program and call subprograms; and a header file which declares the functions and structures all the programs share). The remaining three files contain the meat of the project, which you will be implementing. The programs are described as follows:

- lex.c - implements the lexical analyzer, which takes the input program as a string, tokenizes the program, and returns these tokens as the lexeme list
- parser.c - implements the compiler, which takes the lexeme list, parses for any errors, creates the symbol table, and generates/returns the assembly code
- vm.c - implements the virtual machine which executes the assembly code generated by the parser.

For the first homework, you will be implementing vm.c. Because this is the last file executed by the program (and thus it's dependent on lex.c and parser.c), we will provide compiled versions of lex.c and parser.c for you.

**Assignment Overview:**

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: the "text" section 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.

You will implement the function

*void execute(int trace_flag, instruction *code);*

within the project file **vm.c**.

The instruction array, **code**, functions as the "text" section and contains all the instructions to be executed. The instruction struct is defined in compiler.h, we include a copy of the definition in Appendix D. In affect, each index of the code array has three fields:

- IR.op - the operation code which specifies the type of instruction
    - LIT, OPR, LOD, STO, CAL, RTN, INC, JMP, JPC, SYS

- IR.l - the lexicographical level which is only used by LOD, STO, and CAL instructions to indicate the activation record being referenced in the instruction
- IR.m - has different functions depending on the function type
    - For LIT and INC it holds a literal value
    - For OPR and SYS it holds the secondary operation code
    - For LOD and STO it holds a data address
    - For CAL, JMP, and JPC it holds a program address

Your VM will never receive code with errors and it will always include a halt instruction. It is not necessary for your program to know how many instructions there are in the input.

You will need to declare an integer array to serve as the "stack" section. compiler.h contains a constant, ARRAY_SIZE, you can use for the size of the stack. All entries of the stack should initialize with zero values. The stack grows upwards.

The PM/0 CPU has five registers to handle the stack and text segments: The registers are named base pointer (BP), stack pointer (SP), program counter (PC), instruction register (IR), and halt.

- Base pointer (BP) points the base of the current activation record (the static link). The initial value of BP is 0
- Stack pointer (SP) points to the top of the stack. The initial value of SP is -1
- Program counter (PC) points to the next instruction in the text. The initial value of PC is 0.
- Instruction Register (IR) points to the current instruction in the text. Initially, the IR should be declared, but not initialized.
- The halt flag tells the P-Machine when the program is done. It's initial value is 0 (false).

BP, SP, PC, and halt should be integers. IR should be an instruction. (See Appendix D or compiler.h for the definition of the instruction struct).

## 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.

### 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 (PC← PC + 1).

**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 JPC (IR.op = 8), then the machine will check the value at the top of the stack. If it's equal to 0, then PC will be set to the M value of the instruction (PC ←IR.m). This uses up the value at the top of the stack, so SP must be decremented (SP ←SP - 1).

**Implementing the P-Machine**
After setting up the stack and its support registers, implement the P-Machine as a while loop. Loop as long as the halt flag is false. Within the loop, you'll have three steps. The two steps specified above, Fetch and Execute (which can be implemented with either a switch case or an if-else-if structure), followed by a print step. We've given you a print function in the vm.c skeleton and in Appendix D. Make sure to only print if the trace_flag is true (1). The execution should only be printed if the trace_flag is true, but the prompts for read and write instructions should always be printed. The descriptions for the instructions are given in Appendices A and B.

**Testing Your Program**
As explained in the project overview, your assignment is one part of a multi-file C program. We understand that this is the first time many of you are working with a multi-file C program. To test your program you need to upload all of the project files to Eustis3:

- driver.c - the driver program. It contains the main function and reads in the input program. You should not alter driver.c.
- compiler.h - the header file. It contains all the structure, function, and constant declarations for the project. You should not alter compiler.h
- lex.o and parser.o - compiled versions of our implementation of lex.c and parser.c. You are welcome to open them, but they'll look like gibberish because compiled files are not human readable. They're written in machine code. They're a lot like a.out files or other executable files. They were compiled on Eustis3. Because they were compiled on Eustis3, they include references to libraries that the Eustis3 operating system has. It's possible that your computer also has those libraries, so you'd be able to run the project on your home computer, but this is unlikely, especially if you have a windows machine. If you try to run these files on a machine that doesn't have the necessary libraries, you'll get a segfault.
- vm.c - where you'll write your code
- Any input and output files you want to use.

To compile, use the command "gcc driver.c lex.o parser.o vm.c"
To execute, use the command "./a.out input.txt -v"

- Replace input.txt with the name of whatever input file you're using
- "-v" is a tag for the driver so your program knows to print the virtual machine trace. This sets the trace_flag to true.
- There are three other flags supported by the driver:
    - "-l" which will print the lexeme list
    - "-s" which will print the symbol table
    - "-c" which will print the assembly code

To compare your output to correct output, use the command "./a.out input -v > output.txt" to generate your output. The program will still stop and wait for input if there is a read instruction. Then compare your output to correct output with the command "diff -w -B your_output.txt correct_output.txt" This will print out any differences between the two files. If the command doesn't print anything, then the two files are exactly the same (the desired outcome).

**Making Your Own Test Cases**
We're providing you with two test cases to use when developing your program, but we will be using different test cases to grade, so you may want to write your own test cases. Input files are written in PL/0, which is fairly simple. We've included the grammar in Appendix E. To get the correct output for your test cases, we're providing the "magic" file. "magic" is a compiled version of our implementation of the project. It works like a.out. You must be on Eustis3 to run it. You may get an error with permissions, use the command "chmod +x magic" if this happens. To get the assembly code for your input program, use the command "./magic input.txt -c" To get the vm trace for you input program, use the command "./magic input.txt -v". If your input program has any errors, the project will print them, and if there are read commands, it will always wait for input. To write the output to a file, add "> output.txt" to the end of the command.

**Administrative 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 run. If you need help setting up your computer to access Eustis3, reach out to a TA or Dr. Aedo.
2. Do not change the ISA.
3. Do not add instructions or combine instructions.
4. Do not change the format of the input.
5. Do not try to implement lex.c or parser.c.
6. Do not add a main function to vm.c.
7. Include comments in your program.
8. Do not implement each VM instruction with a function. Use a switch case or if-else-if structure.

9. If you submit a program from another semester or section or from the internet, this is considered plagiarism. We regard this as cheating. At a minimum, you will receive a zero on this assignment.
10. Submit to Webcourses:
    a) The source code of your PM/0 VM which should be named “vm.c”
    b) Student names should be written in the header comment of each source code file and in the comments of the submission

**Rubric:**

20 – Compiles
20 – Produces lines of meaningful execution before segfaulting or looping infinitely
10 – Well commented source code
5 - Program is structured correctly
5 – Fetch cycle is implemented correctly
5 – Arithmetic instructions are implemented correctly
5 - Logic instructions are implemented correctly
5 – Read and write instructions are implemented correctly
5 - Jump instructions are implemented correctly
10 – Load and store instructions are implemented correctly
10 – Call and return instructions are implemented correctly

# Appendix A

## Instruction Set Architecture (ISA)

Arithmetic and logical 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. The same is true for system operations, OP = 10 (SYS).

**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 – | **RTN 0, 0** | Return from the current procedure. Restores the environment using information from the activation record. |
| 07 – | **INC 0, M** | Allocate **M** memory words (increment SP by M). First three are reserved to **Static Link (SL)**, **Dynamic Link (DL)**, **and Return Address (RA)** |
| 08 – | **JMP 0, M** | Jump to instruction **M (PC 🡐 M)** |
| 09 – | **JPC 0, M** | Jump to instruction **M** if top stack element is 0 |
| 10 – | **SYS 0, M** | Operation to be performed by the system. |

# Appendix B

**ISA Pseudo Code**

The base function takes the stack, current bp, and **L** value and returns the base of the desired activation record. It is provided for you in both the vm.c skeleton and Appendix D.

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

02 – OPR  0, M          M
                        1     ADD       sp ← sp – 1;
                                        stack[sp] ← stack[sp] + stack[sp + 1]

                        2     SUB       sp ← sp – 1;
                                        stack[sp] ← stack[sp] - stack[sp + 1]

                        3     MUL       sp ← sp – 1;
                                        stack[sp] ← stack[sp] * stack[sp + 1]

                        4     DIV       sp ← sp – 1;
                                        stack[sp] ← stack[sp] / stack[sp + 1]

                        5     EQL       sp ← sp – 1;
                                        stack[sp] ← stack[sp] == stack[sp + 1]

                        6     NEQ       sp ← sp – 1;
                                        stack[sp] ← stack[sp] != stack[sp + 1]

                        7     LSS       sp ← sp – 1;
                                        stack[sp] ← stack[sp] < stack[sp + 1]

                        8     LEQ       sp ← sp – 1;
                                        stack[sp] ← stack[sp] <= stack[sp + 1]

                        9     GTR       sp ← sp – 1;
                                        stack[sp] ← stack[sp] > stack[sp + 1]

                        10    GEQ       sp ← sp – 1;
                                        stack[sp] ← stack[sp] >= stack[sp + 1]
```

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

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

05 - CAL   L, M      stack[sp + 1]  ←  base(stack, bp, L);       // static link (SL)
                     stack[sp + 2]  ← bp;                        // dynamic link (DL)
                     stack[sp + 3]  ← pc;                        // return address (RA)
                     bp ← sp + 1;
                     pc ← M;

06 – RTN  0, 0       sp ← bp - 1;
                     bp ← stack[sp + 2];      // dynamic link (DL)
                     pc ← stack[sp + 3];      // return address (RA)

07 – INC   0, M      sp ← sp + M;

08 – JMP   0, M      pc ← M;

09 – JPC   0, M      if stack[sp] == 0 then { pc ← M; }
                     sp ← sp - 1;

10 – SYS, 0, M       M
                     1      WRT        printf("\nOutput : %d", stack[sp]);
                                       printf("\n\t\t\t\t");
                                       sp ← sp – 1;

                     2      RED        sp ← sp + 1;
                                       printf("\nInput : ");
                                       scanf("%d", &stack[sp]);
                                       printf("\t\t\t\t");

                     3      HLT        halt ← 1 (true)
```

# Appendix C

**Example Program**

***Input Program (*“mult.txt”*):***

```
var x;
var y;
const one := 1;
var result;
procedure MULT;
begin
    read x;
    read y;
    result := 0;
    call MULT;
    def MULT {
        begin
            if y == 0 then
                return;
            result := result + x;
            y := y - one;
            call MULT
        end
    };
    write result
end.
```

***Assembly Code (*“./a.out mult.txt -c”*)***

```
Line OP Name  L M
0    7  INC   0 6
1    10 RED   0 2
2    4  STO   0 3
3    10 RED   0 2
4    4  STO   0 4
5    1  LIT   0 0
6    4  STO   0 5
7    5  CAL   0 9
8    8  JMP   0 25
9    7  INC   0 3
10   3  LOD   1 4
11   1  LIT   0 0
12   2  EQL   0 5
13   9  JPC   0 15
14   6  RTN   0 0
```

```
15   3  LOD   1 5
16   3  LOD   1 3
17   2  ADD   0 1
18   4  STO   1 5
19   3  LOD   1 4
20   1  LIT   0 1
21   2  SUB   0 2
22   4  STO   1 4
23   5  CAL   1 9
24   6  RTN   0 0
25   3  LOD   0 5
26   10 WRT   0 1
27   10 HLT   0 3
```

***Virtual Machine Output** (”./a.out mult.txt -v”)**/**(“mult_out.txt”)**:***

```
                    PC  BP  SP  stack
Initial Values:     0   0   -1
0   INC 0   6   1   0   5   0 0 0 0 0 0
1   RED 0   2
Input : 7       2   0   6   0 0 0 0 0 0 7
2   STO 0   3   3   0   5   0 0 0 7 0 0
3   RED 0   2
Input : 2       4   0   6   0 0 0 7 0 0 2
4   STO 0   4   5   0   5   0 0 0 7 2 0
5   LIT 0   0   6   0   6   0 0 0 7 2 0 0
6   STO 0   5   7   0   5   0 0 0 7 2 0
7   CAL 0   9   9   6   5   0 0 0 7 2 0
9   INC 0   3   10  6   8   0 0 0 7 2 0 0 0 8
10  LOD 1   4   11  6   9   0 0 0 7 2 0 0 0 8 2
11  LIT 0   0   12  6   10  0 0 0 7 2 0 0 0 8 2 0
12  EQL 0   5   13  6   9   0 0 0 7 2 0 0 0 8 0
13  JPC 0   15  15  6   8   0 0 0 7 2 0 0 0 8
15  LOD 1   5   16  6   9   0 0 0 7 2 0 0 0 8 0
16  LOD 1   3   17  6   10  0 0 0 7 2 0 0 0 8 0 7
17  ADD 0   1   18  6   9   0 0 0 7 2 0 0 0 8 7
18  STO 1   5   19  6   8   0 0 0 7 2 7 0 0 8
19  LOD 1   4   20  6   9   0 0 0 7 2 7 0 0 8 2
20  LIT 0   1   21  6   10  0 0 0 7 2 7 0 0 8 2 1
21  SUB 0   2   22  6   9   0 0 0 7 2 7 0 0 8 1
22  STO 1   4   23  6   8   0 0 0 7 1 7 0 0 8
23  CAL 1   9   9   9   8   0 0 0 7 1 7 0 0 8
9   INC 0   3   10  9   11  0 0 0 7 1 7 0 0 8 0 6 24
10  LOD 1   4   11  9   12  0 0 0 7 1 7 0 0 8 0 6 24 1
11  LIT 0   0   12  9   13  0 0 0 7 1 7 0 0 8 0 6 24 1 0
12  EQL 0   5   13  9   12  0 0 0 7 1 7 0 0 8 0 6 24 0
13  JPC 0   15  15  9   11  0 0 0 7 1 7 0 0 8 0 6 24
15  LOD 1   5   16  9   12  0 0 0 7 1 7 0 0 8 0 6 24 7
16  LOD 1   3   17  9   13  0 0 0 7 1 7 0 0 8 0 6 24 7 7
```

```
17   ADD 0    1    18   9    12   0 0 0 7 1 7 0 0 8 0 6 24 14
18   STO 1    5    19   9    11   0 0 0 7 1 14 0 0 8 0 6 24
19   LOD 1    4    20   9    12   0 0 0 7 1 14 0 0 8 0 6 24 1
20   LIT 0    1    21   9    13   0 0 0 7 1 14 0 0 8 0 6 24 1 1
21   SUB 0    2    22   9    12   0 0 0 7 1 14 0 0 8 0 6 24 0
22   STO 1    4    23   9    11   0 0 0 7 0 14 0 0 8 0 6 24
23   CAL 1    9    9    12   11   0 0 0 7 0 14 0 0 8 0 6 24
9    INC 0    3    10   12   14   0 0 0 7 0 14 0 0 8 0 6 24 0 9 24
10   LOD 1    4    11   12   15   0 0 0 7 0 14 0 0 8 0 6 24 0 9 24 0
11   LIT 0    0    12   12   16   0 0 0 7 0 14 0 0 8 0 6 24 0 9 24 0 0
12   EQL 0    5    13   12   15   0 0 0 7 0 14 0 0 8 0 6 24 0 9 24 1
13   JPC 0    15   14   12   14   0 0 0 7 0 14 0 0 8 0 6 24 0 9 24
14   RTN 0    0    24   9    11   0 0 0 7 0 14 0 0 8 0 6 24
24   RTN 0    0    24   6    8    0 0 0 7 0 14 0 0 8
24   RTN 0    0    8    0    5    0 0 0 7 0 14
8    JMP 0    25   25   0    5    0 0 0 7 0 14
25   LOD 0    5    26   0    6    0 0 0 7 0 14 14
26   WRT 0    1
Output : 14
                   27   0    5    0 0 0 7 0 14
27   HLT 0    3    28   0    5    0 0 0 7 0 14
```

## Appendix D

```
typedef struct instruction {
    int op;
    int l;
    int m;
} instruction;

int base(int *stack, int BP, int L)
{
    while (L > 0)
    {
        BP = stack[BP];
        L--;
    }
    return BP;
}
```

**Use this function after Fetch, but before Execute, make sure to only use if trace_flag is true (1)**

```
void print_instruction(int PC, instruction IR)
{
    char opname[4];

    switch (IR.op)
    {
        case LIT : strcpy(opname, "LIT"); break;
        case OPR :
            switch (IR.m)
            {
                case ADD : strcpy(opname, "ADD"); break;
                case SUB : strcpy(opname, "SUB"); break;
                case MUL : strcpy(opname, "MUL"); break;
                case DIV : strcpy(opname, "DIV"); break;
                case EQL : strcpy(opname, "EQL"); break;
                case NEQ : strcpy(opname, "NEQ"); break;
                case LSS : strcpy(opname, "LSS"); break;
                case LEQ : strcpy(opname, "LEQ"); break;
                case GTR : strcpy(opname, "GTR"); break;
                case GEQ : strcpy(opname, "GEQ"); break;
                default : strcpy(opname, "err"); break;
            }
            break;
        case LOD : strcpy(opname, "LOD"); break;
        case STO : strcpy(opname, "STO"); break;
        case CAL : strcpy(opname, "CAL"); break;
        case RTN : strcpy(opname, "RTN"); break;
        case INC : strcpy(opname, "INC"); break;
        case JMP : strcpy(opname, "JMP"); break;
        case JPC : strcpy(opname, "JPC"); break;
        case SYS :
            switch (IR.m)
            {
                case WRT : strcpy(opname, "WRT"); break;
                case RED : strcpy(opname, "RED"); break;
                case HLT : strcpy(opname, "HLT"); break;
                default : strcpy(opname, "err"); break;
            }
            break;
        default : strcpy(opname, "err"); break;
    }

    printf("%d\t%s\t%d\t%d\t", PC - 1, opname, IR.l, IR.m);
}
```

**Use this function after Execute, make sure to only use if trace_flag is true (1)**

```
void print_stack(int PC, int BP, int SP, int *stack)
{
    int i;
    printf("%d\t%d\t%d\t", PC, BP, SP);
    for (i = 0; i <= SP; i++)
        printf("%d ", stack[i]);
    printf("\n");
}
```

# Appendix E

## PL/0 Grammar

Use for writing test cases.

**EBNF of tiny PL/0:**

program ::= block "**.**" **.**
block ::= declarations statement**.**
declarations ::= { const | var | proc }.
const ::= ["**const**" ident ":=" number "**;**"]**.**
var ::= [ "**var** "ident "**;**"]**.**
proc ::= [ "**procedure**" ident "**;**" ].
statement ::= [ ident "**:=**" expression
  | "**call**" ident
  | "**begin**" statement { "**;**" statement } "**end**"
  | "**if**" condition "**then**" statement
  | "**while**" condition "**do**" statement
  | "**read**" ident
  | "**write**" expression
  | "**def**" ident "**{**" block "**}**"
  **|** "**return**"] **.**
condition ::= expression rel-op expression**.**
rel-op ::= "=="|"**!**="|"<"|"<="|">"|">="**.**
expression ::= term { ("+"|"**-**") term}**.**
term ::= factor {("*"|"/") factor}**.**
factor ::= ident | number | "**(**" expression "**)**" .
number ::= digit {digit}**.**
ident ::= letter {letter | digit}**.**
digit ::= "**0**" | "**1**" | "**2**" | "**3**" | "**4**" | "**5**" | "**6**" | "**7**" | "**8**" | "**9**"**.**
letter ::= "**a**" | "**b**" | … | "**y**" | "**z**" | "**A**" | "**B**" | ... | "**Y**" | "**Z**"**.**

**Based on Wirth's definition for EBNF we have the following rule:**
**[ ] means an optional item.**
**{ } means repeat 0 or more times.**
**Terminal symbols are enclosed in quote marks.**
**A period is used to indicate the end of the definition of a syntactic class.**