Lab-07 A PL/0 p-code virtual machine

In this lab, we are going to implement a virtual machine for executing the p-code in binary format.

1 Objectives

1. Construct a virtual machine incrementally.

2 Learning Outcomes

By completing this lab, learners should be able to

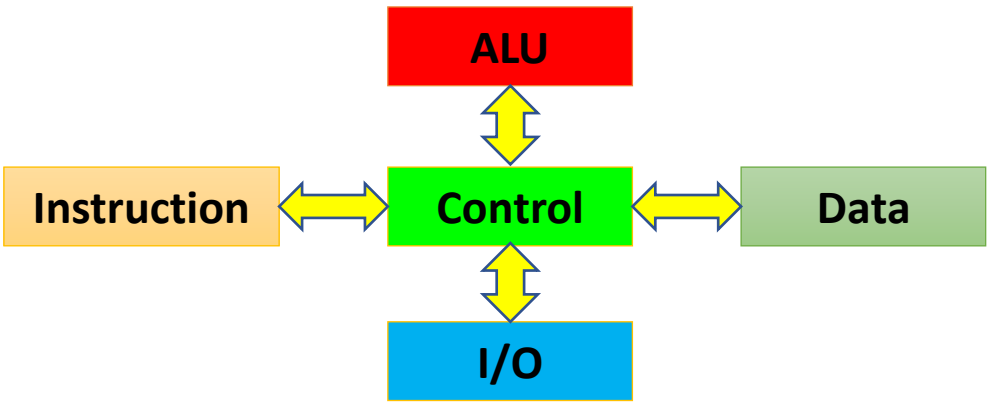
1. build a simple virtual machine incrementally

3 A PL/0 p-code virtual machine

The virtual machine that we are going to build is a simple **REPL stack machine** with **Harvard architecture**.

3.1 Harvard architecture

Harvard architecture is a computer architecture where the storages for instruction and the data are separate. Hence, the instruction will never overwrite the data and vice versa. The architecture greatly simplifies the implementation of machine because the instruction memory and data memory does not need to have the same characteristics.



3.2 Stack Machine

Stack machine is a machine using the stack operations (push and pop) to access the data memory. The main advantages of stack machine are:

- No explicit memory addressing is required because the operands are implicit when the stack is used.
- Stack machine usually leads to more compact machine codes.
- Easier to build compilers or interpreters for stack machines.

3.3 A P-code virtual machine

A PL/0 P-code virtual machine that we are building has the following characteristics.

- It is a stack machine with Harvard architecture.
- It has 3 registers:
  - B (base pointer) points to the base of the current stack.
  - P (program counter) points to the address of the next instruction to be executed.
  - T (top-of-stack) points to one address after the top of the stack.
- The initial values of each pointer are as follows:
  - B: 0 since at the beginning, there is only 1 stack; hence, the base of the stack is 0.
  - P: 0 since the first instruction to be executed is at instruction memory location 0.
  - T: 0 since at the beginning, there is no value in the data memory; in other words, no data are on the stack.
- The instruction memory stores the instruction in the decoded format. E.g. the machine code E001 will be stored as (7,0,1).

4 REPL

A **read-eval-print loop** (**REPL**, pronounced rep-ple) also called an **interactive top-level shell** or a **language shell** is a type of interpreters that read the instruction, evaluate the instruction, then show the result of the evaluation. Examples of REPLE includes traditional calculators, Python interactive shell, Windows Command Prompt, PowerShell, Bash shell for Linux-like OS, etc.

A REPL usually performs the following jobs:

1. **R**ead the user input (command, statement, expressions, etc.).
2. **E**valuates or executes the inputs from step 1.
3. **P**rint the results from step 2 (the evaluation step).
4. **L**oop back to step 1.

In this lab, we will use **?>** as the prompt for the shell.

5 Tasks

We are going to step-by-step build the virtual machine. The virtual machine should be able to read a file containing binary machine code similar to the one in lab-05. In this lab, the commands for the emulator are ***case-sensitive*** and use only lower-case letters.

When the emulator starts, it should display the following message:

```
*** Welcome to a PL/0 P-code machine shell! ***
?>
```

5.1 Phase I: The quit command.

In this phase, we will implement a **`quit`** command. This command is used to exit the loop; in other words, to quit the shell. Hence, once this command is evaluated by the virtual machine, the virtual machine should stop and return to the environment of the caller.

```
*** Welcome to a PL/0 P-code machine shell! ***
?> quit
>
```

5.2 Phase II: The dump command.

In this phase, we will implement a **dump** command. This command displays the content of the virtual machine's registers, the content of the data memory and the program. You may start implement this phase with only 100 words of data memory and 20 words for instruction memory (later on we will extend the memory size).

```
*** Welcome to a PL/0 P-code machine shell! ***
?>dump
```

Sample output of the **dump** command (derived from [1]):

```
REGISTERS:
B (Base):            0000
P (Program Counter)  0000
T (Top of stack):    0000

INSTRUCTION MEMORY:
         0            1            2            3            4
00   (0,0,0)      (0,0,0)      (0,0,0)      (0,0,0)      (0,0,0)
05   (0,0,0)      (0,0,0)      (0,0,0)      (0,0,0)      (0,0,0)
10   (0,0,0)      (0,0,0)      (0,0,0)      (0,0,0)      (0,0,0)
15   (0,0,0)      (0,0,0)      (0,0,0)      (0,0,0)      (0,0,0)

DATA MEMORY:
       0    1    2    3    4    5    6    7    8    9
0    0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
10   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
20   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
30   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
40   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
50   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
60   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
70   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
80   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
90   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

Note

1. the instruction memory stores the instruction in the decoded form.
2. The stored instructions are in the following form: (F,L,V) where
    a. F: the function to be executed
    b. L: the level of the function call
    c. V: the value to use or the operation to be executed

The sample output provides only the minimal output you should display. Please feel free to design the output that you prefer. For instance, you may

- Display the instruction in the instruction memory at the location pointed by the P register.

- Display the assembly code for the instruction mentioned above.
- Display the size of available data memory.
- Display the size of available instruction memory.
- Etc.

5.3 Phase III: The load command

In this phase, we will implement the **load** command. This command will prompt the user for the file name of a P-code machine code (.mc) and load the content of the file into the memory of virtual machine. For instance, if the file contains the following machine codes.

```
000F
E002
E000
```

After executing the **load** and **dump** command, the instruction memory of the virtual machine should look similar to the picture below.

Note that, this command should affect only the instruction memory, not the data memory. The registers also should not be affected by this command.

```
*** Welcome to a PL/0 P-code machine shell! ***
?>load
Please enter filename: sample.mc
?>dump
```

```
REGISTERS:
B (Base):            0000
P (Program Counter)  0000
T (Top of stack):    0000

INSTRUCTION MEMORY:
        0           1           2           3           4
00   (0,0,15)    (7,0,2)     (7,0,0)     (0,0,0)     (0,0,0)
05   (0,0,0)     (0,0,0)     (0,0,0)     (0,0,0)     (0,0,0)
10   (0,0,0)     (0,0,0)     (0,0,0)     (0,0,0)     (0,0,0)
15   (0,0,0)     (0,0,0)     (0,0,0)     (0,0,0)     (0,0,0)

DATA MEMORY:
       0    1    2    3    4    5    6    7    8    9
0    0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
10   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
20   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
30   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
40   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
50   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
60   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
70   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
80   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
90   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

5.4 Phase IV: Implement the opcode

In this phase, we will implement the virtual machine so that it can execute the P-code. Use the following pseudocode as a guideline for implementation.

Function	Description	Pseudocode
0	Load the literal V onto the stack	stack[T] = V T = T + 1
1	Allocate the storage on the stack	T = T + V
2	Load the variable onto the stack at level L	stack[T] = stack[base(L) + V] T = T + 1
3	Assign the variable with the value at the top of the stack	T = T – 1 stack[base(L) + V] = stack[T]
4	Call a routine (V) at level L	stack[T] = base(l) stack[T+1] = B stack[T+2] = P B = T P = V
5	Unconditional Jump	P = V

6	Conditional Jump	if stack[T – 1] == 0
		  P = V
		  T = T – 1
		endif

7	Operation specified by V
	V == 0: stop
	V == 1: return
	V == 2: negate the number
	V == 3: Addition
	V == 4: Subtraction
	V == 5: Multiplication
	V == 6: Division
	V == 7: Is the operand odd?
	V == 8: Are the operands equal?
	V == 9: Are the operands inequal?
	V == 10: Is a < b ?
	V == 11: Is a <= b ?
	V == 12: Is a > b ?
	V == 13: Is a >= b ?
		
		V == 0: Stop the program execution
		V == 1: T = B – 1; P = stack[t + 2]; B = stack[t+1]
		V == 2: stack[T – 1] = – stack[T – 1]
		V == 3: T = T – 1; stack[T – 1] = stack[T – 1] + stack[T]
		V == 4: T = T – 1; stack[T – 1] = stack[T – 1] - stack[T]
		V == 5: T = T – 1; stack[T – 1] = stack[T – 1] * stack[T]
		V == 6: T = T – 1; stack[T – 1] = stack[T – 1] / stack[T]
		V == 7: stack[T – 1] = int(stack[T – 1] is odd)
		V == 8: T = T – 1;
		  stack[T – 1] = int(stack[T – 1] = stack[T];
		V == 9: T = T – 1;
		  stack[T – 1] = int(stack[T – 1] != stack[T];
		V == 10: T = T – 1;
		  stack[T – 1] = int(stack[T – 1] < stack[T]);
		V == 11: T = T – 1;
		  stack[T – 1] = int(stack[T – 1] <= stack[T]);
		V == 12: T = T – 1;
		  stack[T – 1] = int(stack[T – 1] > stack[T]);
		V == 13: T = T – 1;
		  stack[T – 1] = int(stack[T – 1] >= stack[T]);

5.5 Phase V: The step command

In this phase, we will implement the **`step`** command. This command will execute exactly one instruction in the instruction memory pointed by the PC register. The suggested internal execute steps are

1. Fetching: Read the instruction memory at the location indicated by the P register.
2. Incrementing: Increase the value of the P register by 1.
3. Executing: Execute the opcode.

Be careful about when the jump instructions are executed.

```
?>step
?>dump
```

For instance, executing the **step** command after the command from the previous sample, the virtual machine should display the following screen. The command **(0,0,15)** is **LIT 0 15** loading the value 15 (or F in hexadecimal) onto the top of the stack. Hence,

- the B register does not change.
- the P is now 1 (points to the next instruction at instruction memory location 1)
- the T register is now 1 (the number 15 is stored on the stack; hence, the top of the stack is increased by 1).
- The value at data memory location 0 is now 15 (or F in hexadecimal).

```
REGISTERS:
B (Base):            0000
P (Program Counter)  0001
T (Top of stack):    0001

INSTRUCTION MEMORY:
        0           1           2           3           4
00   (0,0,15)    (7,0,2)     (7,0,0)     (0,0,0)     (0,0,0)
05   (0,0,0)     (0,0,0)     (0,0,0)     (0,0,0)     (0,0,0)
10   (0,0,0)     (0,0,0)     (0,0,0)     (0,0,0)     (0,0,0)
15   (0,0,0)     (0,0,0)     (0,0,0)     (0,0,0)     (0,0,0)

DATA MEMORY:
       0    1    2    3    4    5    6    7    8    9
0    000F 0000 0000 0000 0000 0000 0000 0000 0000 0000
10   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
20   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
30   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
40   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
50   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
60   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
70   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
80   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
90   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

After executing another **step** command:

```
?>step
?>dump
```

The command **(7,0,1)** is **OPR 0 1**, negating the value at the top of the stack. Hence,

- the B register does not change.

- the P is now 2 (points to the next instruction at instruction memory location 2)
- the T does not change because the operation OPR 0 1 is performed at the top of the stack.
- The value at data memory location 0 is now -15 (or FFF1 in hexadecimal).

```
REGISTERS:
B (Base):            0000
P (Program Counter)  0002
T (Top of stack):    0001

INSTRUCTION MEMORY:
        0            1            2            3            4
00  (0,0,15)     (7,0,2)      (7,0,0)      (0,0,0)      (0,0,0)
05  (0,0,0)      (0,0,0)      (0,0,0)      (0,0,0)      (0,0,0)
10  (0,0,0)      (0,0,0)      (0,0,0)      (0,0,0)      (0,0,0)
15  (0,0,0)      (0,0,0)      (0,0,0)      (0,0,0)      (0,0,0)

DATA MEMORY:
       0    1    2    3    4    5    6    7    8    9
0   FFF1 0000 0000 0000 0000 0000 0000 0000 0000 0000
10  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
20  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
30  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
40  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
50  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
60  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
70  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
80  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
90  0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
```

5.6 Phase VI: the run command:

In this phase, we will implement the **run** command. The virtual machine will execute the instructions until it encounters the **stop** command (**OPR 0 0**).

For instance, if we **load** the program onto the virtual machine and execute the **run** command. The virtual machine will execute the instructions until it stops.

```
*** Welcome to a PL/0 P-code machine shell! ***
?>load
Please enter filename: sample.mc
?>run
?>dump
```

Once, the virtual machine stops the execution, the following output from the virtual machine may be followed.

```
REGISTERS:
B (Base):            0000
P (Program Counter)  0003
```

```
T (Top of stack):   0001

INSTRUCTION MEMORY:
        0           1           2           3           4
00   (0,0,15)    (7,0,2)     (7,0,0)     (0,0,0)     (0,0,0)
05   (0,0,0)     (0,0,0)     (0,0,0)     (0,0,0)     (0,0,0)
10   (0,0,0)     (0,0,0)     (0,0,0)     (0,0,0)     (0,0,0)
15   (0,0,0)     (0,0,0)     (0,0,0)     (0,0,0)     (0,0,0)

DATA MEMORY:
MEMORY:
       0    1    2    3    4    5    6    7    8    9
0    FFF1 0000 0000 0000 0000 0000 0000 0000 0000 0000
10   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
20   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
30   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
40   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
50   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
60   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
70   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
80   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
90   0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
```