# Lab-05 A Simple Bytecode Disassembler

In this lab, we are going to implement a disassembler for a simple bytecode instruction set.

## 1 Objectives

1. Construct a disassembler iteratively.
2. Check the correctness of machine code.

## 2 Learning Outcomes

By completing this lab, learners should be able to

1. build a simple disassembler.
2. build an object code verifier.
3. recognize the differences between incremental development and iterative development.

## 3 Bytecode

**Bytecode** (also called **portable code** or **p-code**) is a form of object code designed for efficient execution. While bytecode is typically interpreted by a virtual machine, it may be translated into actual machine code to be executed by hardware. Examples of bytecode include Java bytecode, Common Intermediate Language (CIL) used by .NET, MATLAB m-code, Python byte code, Pascal p-code, etc.

In this lab, we will implement a disassembler for a ***slightly modified PL/0 p-code instruction set*** described in [1,2]. A PL/0 p-code is a 16-bit bytecode with the following format:

| **Component** | **Function (F)** | | | **Level (L)** | | **Value (V)** | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Bit** | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| **Byte** | Byte 1 | | | | | | | | Byte 0 | | | | | | | |

Where

- **Function** (**F**) stands for the function to perform (3 bits).
- **Level** (**L**) stands for the level of the call (2 bits).
- **Value** (**V**) stands for a number, a data address, a program address, or an operator depends on the value of **F** (11 bits).

***Note that: The order of functions of the virtual machine code for this lab is organized differently from the one described in [1,2]. The order would not affect the assembly code (p-code). However, if you use the PL/0 machine code from Internet, you need to change the opcode (function) accordingly.***

This lab will form a basis for building a simple virtual machine for executing p-code. Therefore, you need to meticulously carry out this lab.

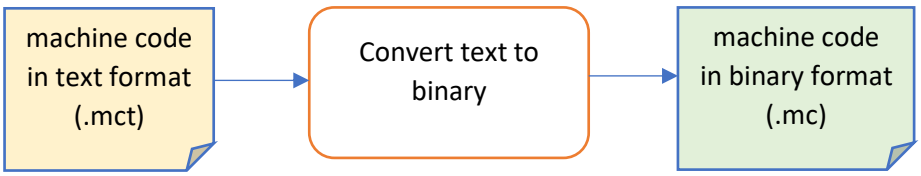
Our PL/0 p-code instruction set is as follow:

| Function | Mnemonic | | | Description |
|---|---|---|---|---|
| 0 | LIT | 0 | number | Load the literal (number; **V**) onto the stack |
| 1 | INT | 0 | number | Allocate storage on the stack |
| 2 | LOD | level | data address | Load the value of variable (at data address) with respect to level (**L**) onto the stack. i.e. stack ← variable |
| 3 | STO | level | data address | Store the value from the top of the stack onto the variable (at data address) with respect to level (**L**). i.e. variable ← stack |
| 4 | CAL | level | program address | Call a subroutine (**V**) at level (**L**) |
| 5 | JMP | 0 | program address | Unconditional jump |
| 6 | JPC | 0 | program address | Conditional jump |
| 7 | OPR | 0 | operator | Perform either arithmetic or relational operations |

# 4 Tasks

## 4.1 Phase I: Convert text to binary

In this phase, we will convert a text representation of our PL/0 machine code to a binary machine code. This phase acts as a convenient tool for constructing machine code in binary format. The source file for this phase is a text file (with extension .mct) where each line contains 3 numbers representing function, level, and value respectively.




Sample contents of an input file and its corresponding output file. Note that the size of the output file for this example should be 10 bytes (5 instructions or 5 x 16-bit). The space between digits of the binary representation is just for easy manual verification. The binary and decimal numbers shown in the table is merely for your convenience; they should **NOT** appear in the output file.

| The input file (.mct) | The output file (.mc) | | |
|---|---|---|---|
| | Binary | Hexadecimal | Decimal |
| 0 0 0 | 000 00 000 0000 0000 | 00 00 | 0 |
| 7 0 1 | 111 00 000 0000 0001 | E0 01 | 57345 |
| 2 1 2047 | 010 01 111 1111 1111 | 4F FF | 20479 |
| 3 2 79 | 011 10 000 0100 1111 | 70 4F | 28751 |
| 5 0 123 | 101 00 010 1111 1101 | A2 FD | 41725 |

## 4.2 Phase II: Convert text to bytecode with verification.

In this phase, add a machine code verifier to ensure that the input contains only valid machine codes. The following minimal verification are required:

- The number of bits for **function** (**F**) is 3; therefore, the value of the **F** component must be in the range of 0 and 7, inclusively.

- The number of bits for **level** (**L**) is 2; therefore, the value of the **L** component must be in the range of 0 and 3, inclusively.
- The number of bits for **value** (**V**) is 11; therefore, the value of the **V** component must be in the range of 0 and 2047, inclusively.
- In addition, our virtual machine will support only 13 operations; therefore, the value of the **V** component when the function **F** is (**7** or **OPR**) must be in the range of 0 and 12, inclusively

For instance, if the contents of the text machine code (.mct) are

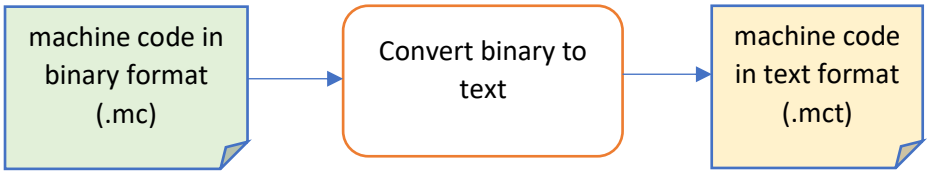
| Line no. | Code |
|---|---|
| 1 | 0 0 7 |
| 2 | 3 0 6 |
| 3 | 2 4 2048 |
| 4 | 5 1 10 |
| 5 | 7 0 0 |
| 6 | 8 0 1 |
| 7 | 7 0 13 |
| 8 | 6 0 5 |

If such text (machine code) file is passed to the text-to-binary converter, the converter should produce no output file and display the following error messages:

```
Line 3: Invalid level [4]
Line 3: Invalid value [2048]
Line 4: The function does not support level [1]
Line 6: Invalid function [8]
Line 7: Invalid operation [13]
```

## 4.3 Phase III: Displaying machine codes

In this phase, we will build another translator for converting the machine code in binary format to a text format. It is basically the reverse process of the Phase I.




## 4.4 Phase IV: Naïve bytecode disassembler

In this phase, we will implement a naïve disassembler where no verification is performed to determine the validity of the bytecode. The task is to generate assembly codes from machine codes.


For instance, the sample code from phase I should produce the following output

| The input file (.mc) in binary | The output file (assembly file with .asm extension) |
|---|---|
| 0000 0000 0000 0000 | LIT 0 0 |
| 1110 0000 0000 0001 | OPR 0 1 |
| 0100 1111 1111 1111 | LOD 1 2047 |
| 0111 0000 0100 1111 | STO 2 79 |
| 1010 0010 1111 1101 | JMP 0 123 |

### 4.5 Phase V: Binary Machine code

In this phase, the input file will be a binary file instead of a text file (with .mc instead of .mct). Therefore, each machine code is now a 16-bit binary code. Therefore, instead of displaying the line number, the program should display the location (the starting location in location 0 or byte 0) of the error. For instance, if the following input (in binary), the program should produce the following output:

Input:

| Byte | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Values | 00 | 00 | E0 | 01 | A0 | FF | A8 | 0B | E0 | 0D |

```
Bytes 6-7: The function does not support level [1]
Bytes 8-9: Invalid operation [13]
```

## 5 References


[1] Wikipedia contributors. 2022. P-code machine. *Wikipedia, The Free Encyclopedia*. Retrieved November 15, 2022 from https://en.wikipedia.org/w/index.php?title=P-code_machine&oldid=1103004966

[2] Niklaus Wirth. 1976. *Algorithms + Data structures = Programs*. Prentice-Hall, Englewood Cliffs, N.J.