**16-bit CPU LC-3 [Little Computer-3] Virtual Machine**

Contents

[Objective 3](#_Toc193480509)

[Materials Required 3](#_Toc193480510)

[Theory 3](#_Toc193480511)

[Architecture of LC-3 3](#_Toc193480512)

[1. Memory Architecture 3](#_Toc193480513)

[2. Register File 3](#_Toc193480514)

[3. Instruction Set Architecture (ISA) 4](#_Toc193480515)

[4. Condition flags 5](#_Toc193480516)

[5. Addressing Modes 5](#_Toc193480517)

[5. Control Unit 5](#_Toc193480518)

[6. Input/Output (I/O) Mechanisms 5](#_Toc193480519)

[7. Data Representation 5](#_Toc193480520)

[Example Compatible Assembly Program with LC-3 VM: 6](#_Toc193480521)

[Logical Flow of the Program 6](#_Toc193480522)

[Pseudocode & Program Architecture 8](#_Toc193480523)

[Conclusion 10](#_Toc193480524)

The LC-3 is a simplified model used to teach fundamental concepts of computer architecture. It operates with a 16-bit word size and a 2^16 (65,536) location address space. The architecture includes eight general-purpose registers (R0-R7), a program counter (PC), and a condition flags register (COND) to track the status of operations. The LC-3 instruction set comprises 16 opcodes, facilitating operations like arithmetic, logic, control flow, and memory access.​

## Architecture of LC-3

### 1. Memory Architecture

* **Address Space:** The LC-3 features a 16-bit address space, allowing for 2^16 (65,536) unique memory locations, each capable of storing a 16-bit word. ​
* **Addressability:** Each memory location is 16 bits wide, aligning with the word size of the architecture.​

#define MEMORY\_MAX (1 << 16)

uint16\_t memory[MEMORY\_MAX]; /\* 65536 locations \*/

### 2. Register File

* **General-Purpose Registers:** The LC-3 includes eight general-purpose registers, labeled R0 through R7, each 16 bits in width. These registers are used for various computational tasks and data storage during program execution. ​
* **Special Registers:**
  + **Program Counter (PC):** Holds the address of the next instruction to be executed.​
  + **Condition Codes (CC):** Indicate the result of the last executed instruction, setting flags for positive (P), zero (Z), or negative (N) outcomes.​

enum

{

R\_R0 = 0,

R\_R1,

R\_R2,

R\_R3,

R\_R4,

R\_R5,

R\_R6,

R\_R7,

R\_PC, /\* program counter \*/

R\_COND,

R\_COUNT

};

### 3. Instruction Set Architecture (ISA)

* **Instruction Format:** Each instruction is 16 bits in length, with the first 4 bits representing the opcode, defining the operation to be performed. ​
* **Opcodes:** The LC-3 supports 16 opcodes, facilitating a range of operations:​
  + **Arithmetic Operations:** ADD, AND​
  + **Data Movement:** LD (Load), LDI (Load Indirect), LDR (Load Base + Offset), LEA (Load Effective Address), ST (Store), STI (Store Indirect), STR (Store Base + Offset)​
  + **Control Flow:** BR (Branch), JMP (Jump), JSR/JSRR (Jump to Subroutine)​
  + **Other Operations:** NOT (Bitwise NOT), RTI (Return from Interrupt), TRAP (System Call)​

enum

{

OP\_BR = 0, /\* branch \*/

OP\_ADD, /\* add \*/

OP\_LD, /\* load \*/

OP\_ST, /\* store \*/

OP\_JSR, /\* jump register \*/

OP\_AND, /\* bitwise and \*/

OP\_LDR, /\* load register \*/

OP\_STR, /\* store register \*/

OP\_RTI, /\* unused \*/

OP\_NOT, /\* bitwise not \*/

OP\_LDI, /\* load indirect \*/

OP\_STI, /\* store indirect \*/

OP\_JMP, /\* jump \*/

OP\_RES, /\* reserved (unused) \*/

OP\_LEA, /\* load effective address \*/

OP\_TRAP /\* execute trap \*/

};

### 4. Condition flags

The R\_COND register stores condition flags which provide information about the most recently executed calculation. This allows programs to check logical conditions such as

if (x > 0) { ... }. Each CPU has a variety of condition flags to signal various situations. The LC-3 uses only 3 condition flags which indicate the sign of the previous calculation.

enum

{

FL\_POS = 1 << 0, /\* P \*/

FL\_ZRO = 1 << 1, /\* Z \*/

FL\_NEG = 1 << 2, /\* N \*/

};

### 5. Addressing Modes

* **Immediate:** The operand is specified within the instruction itself.​
* **Register:** The operand is located in one of the general-purpose registers.​
* **PC-Relative:** The address is determined by adding a sign-extended offset to the current value of the PC.​
* **Base+Offset:** The address is calculated by adding a sign-extended offset to the contents of a base register.​
* **Indirect:** The instruction specifies an address that points to another memory location containing the actual operand.

### 5. Control Unit

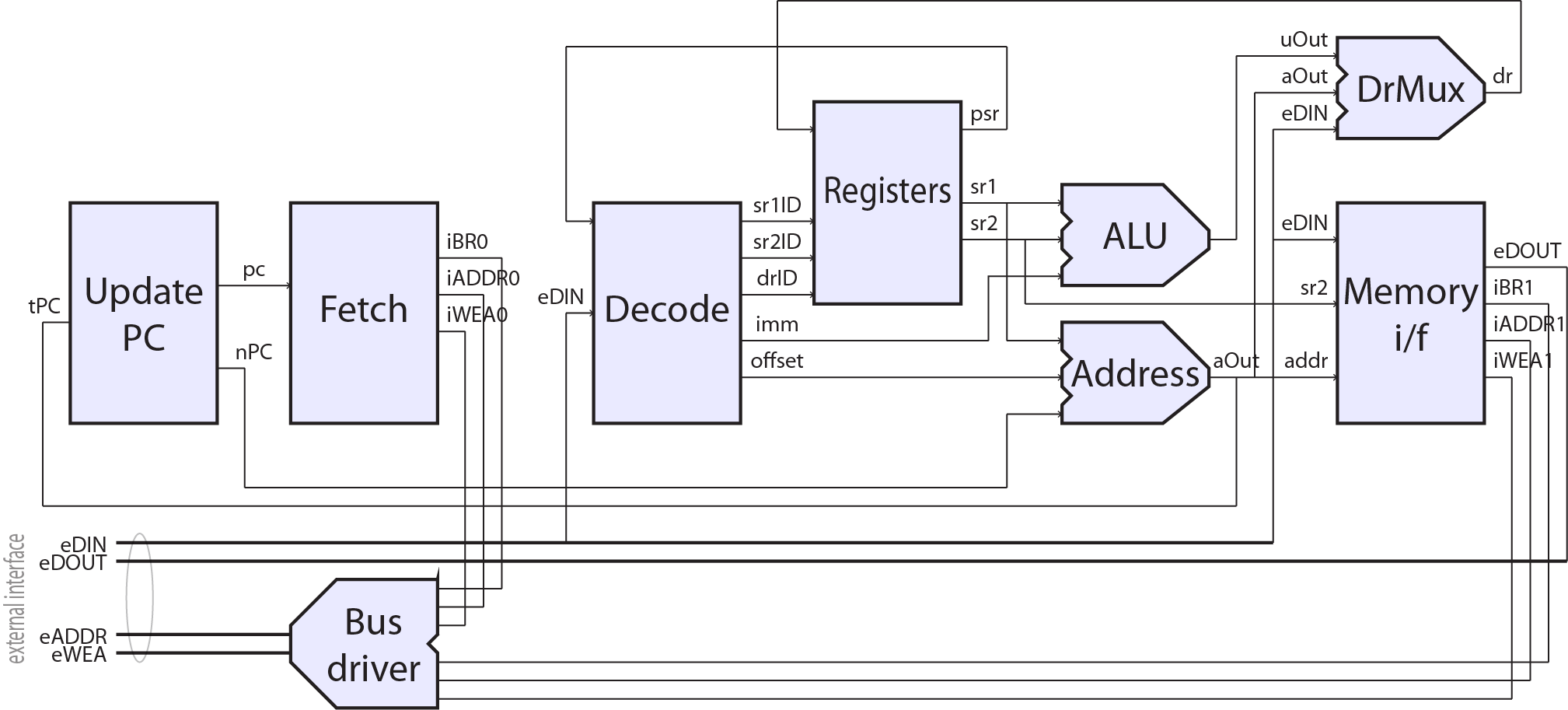
* **Finite State Machine (FSM):** Manages the sequencing of instruction execution through states such as fetch, decode, evaluate address, fetch operands, execute, and store result.​

### 6. Input/Output (I/O) Mechanisms

* **Memory-Mapped I/O:** Certain memory addresses are designated for I/O device registers, enabling communication between the CPU and peripheral devices.​
* **TRAP Routines:** Special system calls that facilitate common I/O operations, such as reading a character from the keyboard or displaying a character on the screen.​

### 7. Data Representation

* **Two's Complement:** Utilized for representing signed integers, allowing for straightforward arithmetic operations.​
* **ASCII Encoding:** Employed for character data, enabling standard representation of textual information.​

The LC-3's architecture, with its limited yet functional instruction set and straightforward design, serves as an effective tool for learning the essentials of computer organization and assembly language programming

## Example Compatible Assembly Program with LC-3 VM:

.ORIG x3000 ; this is the address in memory where the program will be loaded

LEA R0, HELLO\_STR ; load the address of the HELLO\_STR string into R0

PUTs ; output the string pointed to by R0 to the console

HALT ; halt the program

HELLO\_STR .STRINGZ "Hello World!" ; store this string here in the program

.END ; mark the end of the file

# Logical Flow of the Program

The simulator's operation can be broken down into the following steps:​

1. **Initialization:**
   * Set up memory and registers.​[jmeiners.com](https://www.jmeiners.com/lc3-vm/)
   * Load the LC-3 program into memory.​
2. **Fetch Cycle:**
   * Retrieve the instruction from memory at the address specified by the PC.​
   * Increment the PC to point to the next instruction.​
3. **Decode Cycle:**
   * Extract the opcode and operands from the fetched instruction.​
4. **Execute Cycle:**
   * Perform the operation specified by the opcode.​
   * Update registers, memory, and condition flags as necessary.​
5. **Interrupt Handling:**
   * Check for and handle any interrupts or exceptions.​
6. **Repeat:**
   * Continue the fetch-decode-execute cycle until a halt instruction is encountered.

A diagram of a program

AI-generated content may be incorrect.

# Pseudocode & Program Architecture

initialize\_memory\_and\_registers()

load\_program\_into\_memory(program)

while not halted:

instruction = fetch\_instruction(PC)

PC = PC + 1

opcode, operands = decode\_instruction(instruction)

execute\_instruction(opcode, operands)

if opcode == HALT:

halted = true

1. @@{Includes}
2. @@{Memory Mapped Registers}
3. @@{TRAP Codes}
4. @@{Memory Storage}
5. @@{Register Storage}
6. @@{Input Buffering}
7. @@{Handle Interrupt}
8. @@{Sign Extend}
9. @@{Swap}
10. @@{Update Flags}
11. @@{Read Image File}
12. @@{Read Image}
13. @@{Memory Access}
14. @@{Main Loop}

A white background with black and red lines and numbers

AI-generated content may be incorrect.A screenshot of a computer code

AI-generated content may be incorrect.A screen shot of a computer program

AI-generated content may be incorrect.

Figure1: Register Contents after Performing Addition

Figure 2 (b): 2048 game on LC-3

Figure 2(a): 2048 Game on LC-3