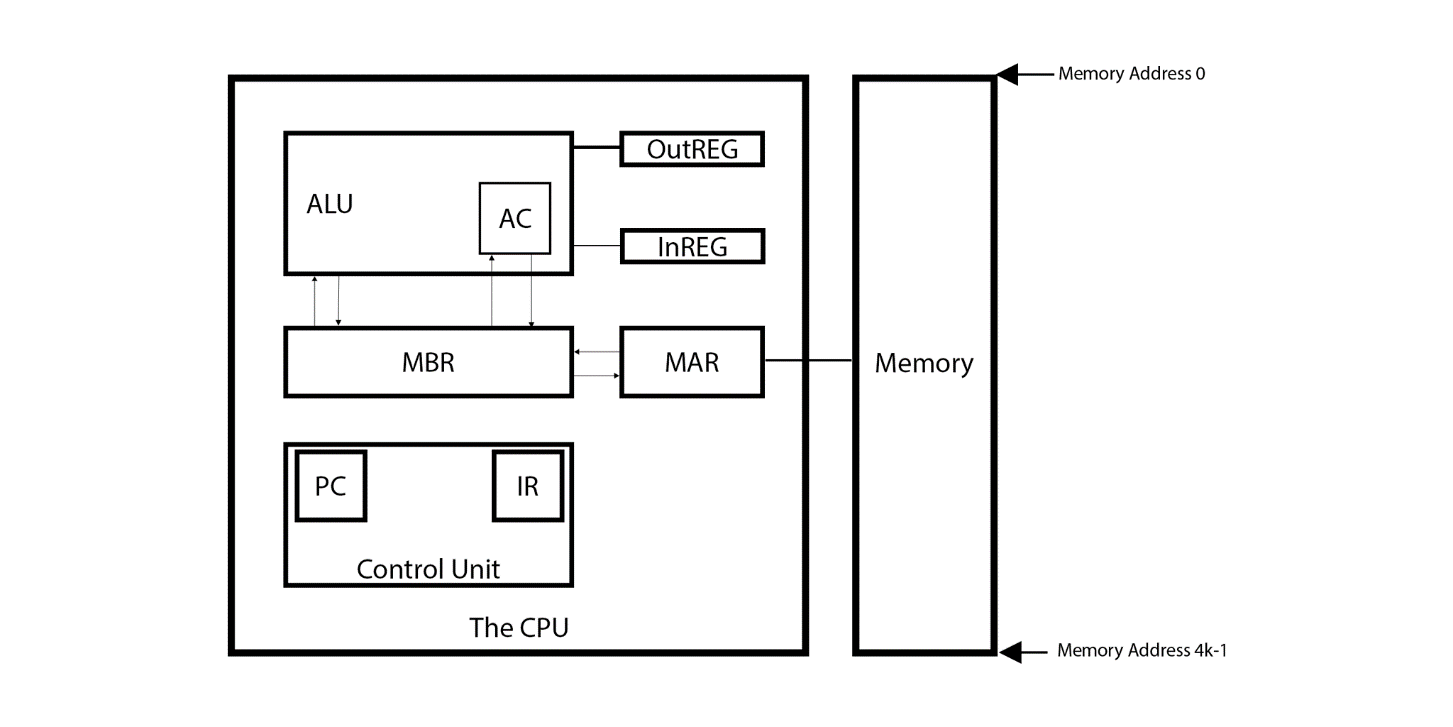
MARIE ('**M**achine **A**rchitecture that is **R**eally **I**ntuitive and **E**asy') is a machine architecture and assembly language served only for educational purposes from The Essentials of Computer Organization and Architecture (Linda Null, Julia Lobur). In addition, the publisher provides a set of simulator programs for the machine, written in Java.

MARIE.js is a JavaScript version implementation of MARIE. It aims to be as faithful to the original Java programs as it can, while improving on features to make concepts more intuitive and easier to understand.

The basic idea, is that the MARIE assembly language is a simple implementation of the von Neumann architecture as shown below.



An assembly language is the lowest level of abstraction you can get away from machine language, which is binary code. Each instruction corresponds to its binary representation.

There are several assembly languages, one for each machine architecture. More familiar architectures like x86, ARM and MIPS are fairly complicated (x86 even more so than ARM and MIPS), which is why MARIE is designed to be easy to understand (hence its name).

So in MARIE (as well as in other architectures) we have a collection of registers. These registers are shown below:

* **AC** or Accumulator: intermediate data is stored within the AC
* **PC** or Program Counter: as the name suggests it stores the current position of the instruction, with each instruction having its own address
* **MAR** or Memory Access Register: stores or fetches the 'data' at the given address
* **MBR** or Memory Buffer Register: stores the data when being transferred to or from memory
* **IR** or Instruction Register: holds the current instruction

In MARIE, each instruction is 16 bits long with the first 4 bits representing the opcode and the remaining 12 bits are being used to represent the address.

For example the instruction CLEAR, the Opcode is A in HEX and 1010 in binary so the instruction will look something like `1010............`

**Code Table**

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Instruction** | **Hex Opcode** | **Summary** |
| Arithmetic | Add X | 3 | Adds value in AC at address X into AC, AC ← AC + X |
| Subt X | 4 | Subtracts value in AC at address X into AC, AC ← AC - X |
| AddI X | B | Add Indirect: Use the value at X as the actual address of the data operand to add to AC |
| Clear | A | AC ← 0 |
| Data Transfer | Load X | 1 | Loads Contents of Address X into AC |
| Store X | 2 | Stores Contents of AC into Address X |
| I/O | Input | 5 | Request user to input a value |
| Output | 6 | Prints value from AC |
| Branch | Jump X | 9 | Jumps to Address X |
| Skipcond (C) | 8 | Skips the next instruction based on C: if (C) =  - 000: Skips if AC < 0 - 400: Skips if AC = 0 - 800: Skips if AC > 0 |
| Subroutine | JnS X | 0 | Jumps and Store: Stores value of PC at address X then increments PC to X+1 |
| JumpI X | C | Uses the value at X as the address to jump to |
| Indirect Addressing | LoadI | D | Loads value from indirect address into AC e.g. LoadI addresspointer Gets address value from addresspointer, loads value at the address into AC |
| StoreI | E | Stores value in AC at the indirect address. e.g. StoreI addresspointer Gets value from addresspointer, stores the AC value into the address |
|  | Halt | 7 | End the program |

**Register Transfer Language** or **RTL** shows how the CPU (Assembler) works. Within the CPU there are many components including:

* **AC** or Accumulator : intermediate data is stored within the AC
* **PC** or Program Counter : as the name suggests it counts the current position of the code, each line has its own address
* **MAR** or Memory Access Register , **stores** or **fetches** the 'data' at the given address
* **MBR** or Memory Buffer Register , stores the data when being transferred
* **IR** or Instruction Register

Note that the end of each code, you will need to increment the **PC** by 1, so:

PC ← PC + 1

# RTL of MARIE Code

## Direct Addressing

### Load X

**Load X** loads the value from address X into the **AC**

MAR ← X # load X (address) into MAR

MBR ← M[MAR] # load value stored at address into MBR

AC ← MBR # load value in MBR into AC

### Store X

**Store X** stores the current value from the **AC** into address X

MAR ← X # load address into MAR

MBR ← AC # load AC value into MBR

M[MAR] ← MBR # write MBR value into the Memory of the address indicated by the MAR

### Add X

**Add X** adds the value stored at address X into **AC**

MAR ← X # load X into MAR

MBR ← M[MAR] # load value stored at address X into MBR

AC ← AC + MBR # add value in AC with MBR value and store it back into AC

### Subt X

**Subt X** subtracts the value in AC with the value stored at address X

MAR ← X

MBR ← M[MAR]

AC ← AC - MBR

### Jump X

**Jump X** jumps to address X

PC ← X

## Indirect Addressing

### LoadI X

**LoadI X** loads the value which is stored at address of the address X into the AC

MAR ← X # load value X into MAR

MBR ← M[MAR] # load value stored at address X into MBR

MAR ← MBR # load value back into MAR (MAR cant write itself)

MBR ← M[MAR] # load value stored at the address indicate by the value of address X

AC ← MBR # Load value into AC.

### AddI X

**AddI X** adds the value which is stored at address of the address X to the value in AC and stores it back into AC

MAR ← X # load value X into MAR

MBR ← M[MAR] # load value stored at address X into MBR

MAR ← MBR # load value back into MAR (MAR cant write itself)

MBR ← M[MAR] # load value stored at the address indicate by the value of address X

AC ← MBR + AC # Load value into AC

### JnS X

**JnS X** or Jumps and Stores: Stores PC at address X and jumps to X+1

MAR ← X # loads value X into MAR

MBR ← PC + 1 # loads value of PC into MBR

M[MAR] ← MBR # stores value in MBR into address of MAR

AC ← X + 1 # increments X by 1 and stores it into AC

PC ← AC # jumps program counter to the address indicated by the AC

### JumpI X

**JumpI X** uses the value at X as the address to jump to

MAR ← X # loads value X into MAR

MBR ← M[MAR] # loads value stored at address X into MBR

PC ← MBR # loads the value into PC