Introduction to MARIE, A Basic CPU Simulator

2nd Edition

Jason Nyugen, Saurabh Joshi, Eric Jiang

THE MARIE.JS TEAM (HTTPS://GITHUB.COM/MARIE-JS/)

Introduction to MARIE, A Basic CPU Simulator

Copyright © 2016

Second Edition Updated August 2016

First Edition – July 2016

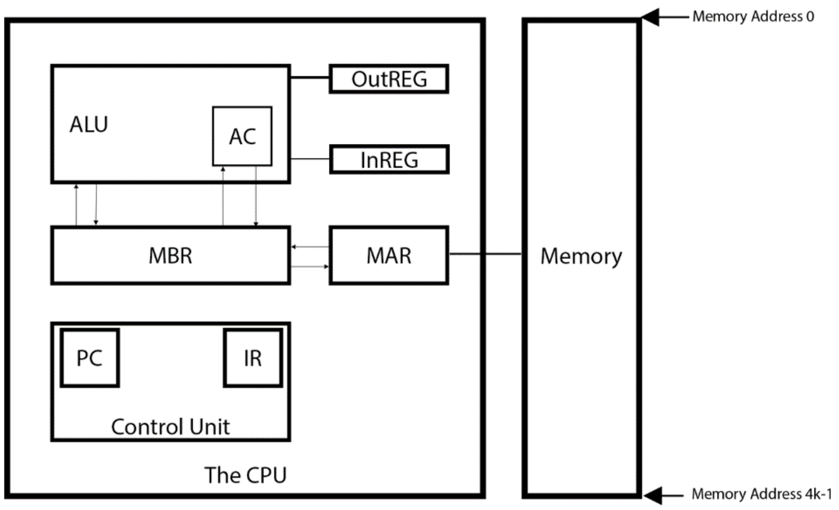
By Jason Nyugen, Saurabh Joshi and Eric Jiang

Please reference this document at https://marie-js.github.io/MARIE.js/book.pdf for Student Integrity Policies for more information please visit your university's Student Integrity Policy.

Contents

Introduction to MARIE and MARIE.js

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. In this book we will use MARIE.js this is available at: https://marie-js.github.io/MARIE.js/ 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

MARIE Instruction Set

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............`

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) is - 000: Skips if AC < 0 - 400: Skips if AC = 0 - 800: Skips if AC > 0
Subroutine	JnS X	0	Jumps and Store: Stores PC at address X and jumps to X+1
	JumpI X	C	Uses the value at X as the address to jump to
Indirect Addressing	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
	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
	Halt	7	End the program

Register Transfer Language

Introduction

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 it's 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 Basic MARIE Code

Direct Addressing

Load X

As explained earlier **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  # writes MBR value into the Memory of 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 into MBR stored at the address indicate by MAR
AC  ← MBR      # 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 address indicated by 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
MAR ← MBR             # loads value back into MAR
MBR ← M[MAR]          # fetches the value at the address into MBR
PC  ← MBR             # loads the value into PC
```

Subroutines

Subroutines

A subroutine is a sequence of instructions that is modular and can be executed multiple times. If you are familiar with any programming language, you may see that subroutines are similar to *functions*. If you only dealt with functions in mathematics, you could view subroutines in terms of inputs and outputs. However the way they perform operations can be quite different. Furthermore, MARIE doesn't provide a way to specify input or output values (for programmers, parameter or return values).

MARIE provides a way to call these subroutines by using the `JnS` instruction, and normal program execution can be resumed once the subroutine exits by using the `JumpI` instruction.

Example

Here is an example that prints the variable x, then halts the program:

```
/ Enter subroutine PrintVariableX
JnS PrintVariableX
Halt

PrintVariableX, HEX 000 / Used for storing return address
                Load X
                Output

                / Exit subroutine PrintVariableX
                JumpI PrintVariableX

X, DEC 42
```

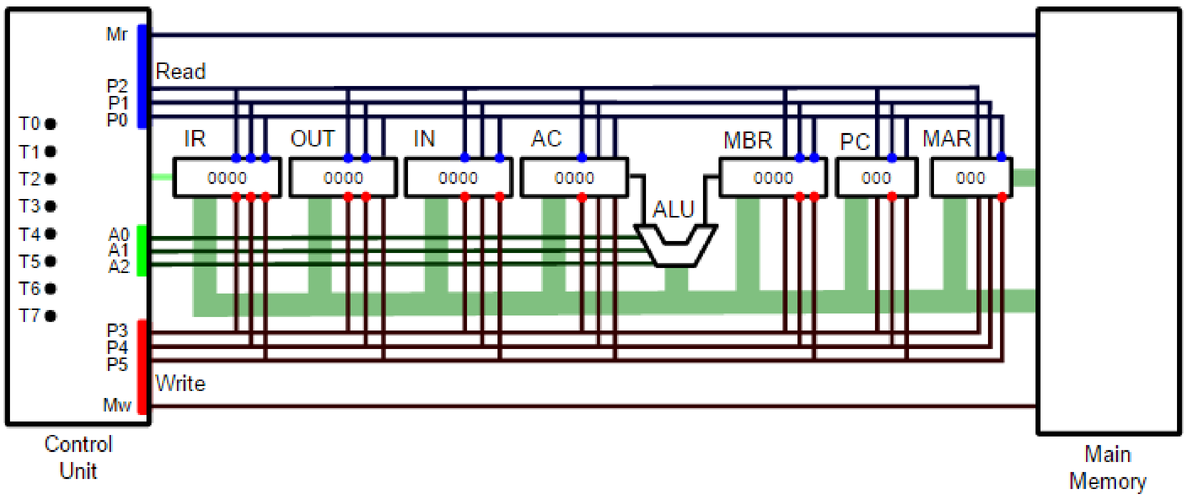
The `JnS` instruction stores the address of the next instruction after it was called. In this case, the memory address points to the `Halt` instruction. This is the return address which will be later used to restore program execution. Once it has done that, it then jumps to the instruction immediately below the label `PrintVariableX`, by using the memory address of `PrintVariableX` and incrementing that value once.
Once the subroutine performs its intended task, and runs the `JumpI` instruction, it loads the memory address stored at the `PrintVariableX` label, and stores it into the `PC` register. Note that we don't need to increment the `PC` register here as it is already taken care of in the fetch part of the fetch-decode-execute cycle before it entered the subroutine. Program execution is resumed from where it was, and the program halts as the `Halt` instruction is executed.

The major part of subroutines is that it can be reused. Here is a slightly modified example that illustrates this.

```
/ Call subroutine PrintVariableX
JnS PrintVariableX
Load X
Add Y
Store X
JnS PrintVariableX / Call subroutine PrintVariableX again
Halt

PrintVariableX, HEX 000 / Used for storing return address
                Load X
                Output

                / Exit subroutine PrintVariableX
                JumpI PrintVariableX

X, DEC 42
Y, DEC 5
```

It doesn't matter where you call the subroutines (unless the subroutine calls itself), or how many calls you make to the subroutines. The return address will be overwritten, and program execution will always be resumed to where it was once the subroutine exits.

Datapath Simulator



The datapath simulator is incorporated into MARIE.js, and can be accessed via the menu: `View → Datapath`.
The purpose of this visualisation is to give an understanding of how instructions and micro-instructions relate to sequence of physical signals.

Register bank

The MARIE simulator register bank is a set of 7 registers used for different purposes. For example, the `PC` register holds the memory address that points to the next instruction. Here is a list of the registers used in the MARIE simulator.

Name	Opcode	Abbreviation	# of bits stored
Memory Address Register	`001`	`MAR`	12
Program Counter	`010`	`PC`	12
Memory Buffer Register	`011`	`MBR`	16
Accumulator	`100`	`AC`	16
Input	`101`	`IN`	16
Output	`110`	`OUT`	16
Instruction Register	`111`	`IR`	16

Memory

The memory stores data in a sequence of locations. At this point of time, nothing much is shown in the memory, apart from whether data is being read from or written to memory.

It is important to know that the data in each memory cell has no meaning in itself. For example, a memory cell may represent as data `0000` (which is very common as usually most of memory cells are empty), but can also be seen as a JnS instruction with memory address `000`, as the highest hexadecimal value is the same as the opcode for the JnS instruction.

Read control bus

The read control bus tells which register (or memory) to output data into the data bus.

Abbreviation	Opcode	Activate Wires
`M[MAR]`	`000`*	`Mr`
`MAR`	`001`	`P0`
`PC`	`010`	`P1`
`MBR`	`011`	`P1 P0`
`AC`	`100`	`P2`
`IN`	`101`	`P2 P0`
`OUT`	`110`	`P2 P1`
`IR`	`111`	`P2 P1 P0`

* While the memory opcode is `000`, it technically means that we do not want to access any register. This is the reason why we have a separate memory read wire so that we can tell the memory exactly when we want to fetch the contents of one memory cell.

Write control bus

The write control bus tells which register (or memory) to read from the data bus and override its value.

Abbreviation	Opcode	Activate Wires
`M[MAR]`	`000`*	`Mw`
`MAR`	`001`	`P3`
`PC`	`010`	`P4`
`MBR`	`011`	`P4 P3`
`AC`	`100`	`P5`
`IN`	`101`	`P5 P3`
`OUT`	`110`	`P5 P4`
`IR`	`111`	`P5 P4 P3`

* Like what is said previously in the read control bus section, this opcode just means do not write to any register. A separate memory write wire is activated instead when we need to write to memory

Data bus

The data bus is 16 bits long, and is used for transferring data (which may hold memory addresses) between registers and/or the memory. It is connected to all registers as well as the memory.

Address bus

The address bus is 12-bits long, and is connected to both the `MAR` register and the memory.

Decode bus

The "decode bus" is 4-bits long, and is connected to both the `IR` register and the control unit. Only the highest 4 bits of the `IR` register is connected to the decode bus, which is used as input for decoding which instruction is needed to be executed.

The control unit, and putting it all together

The control unit handles both the register bank, the memory, and the ALU. It does this by generating a sequence of signals, depending on what instruction it has decoded. All instructions begin with the fetch cycle, which the control unit fetches the next instruction from memory, and increments the program counter. Once the instruction is decoded, it executes the instruction by performing the corresponding sequence of RTL operations. Each RTL operation has its own set of signals that needs to be generated.

The active 'LED' in the time sequence signal labelled `Tn` where `n` is an unsigned integer, shows how many RTL operations have been performed before the current one within the current instruction. These sequential signals are reset once the control unit has finished executing the current instruction and is ready to execute the next instruction.
The first three (`T0`, `T1`, `T2`) time sequence signals are dedicated to the fetch part of the fetch-decode-execute cycle. The rest of the time sequence depends on what instruction the control unit has decoded from the `IR`.

Tutorials

A Simple Calculator

In this tutorial we are going to write some code which assembles a simple addition calculator.

Concepts

Variables in `MARIE` are in string, for example both the variables `X` and `value` work in MARIE. Variables are typically declared at the bottom of the code.
The instruction **INPUT** takes a user input and loads it into the `Accumulator` or `AC`

Coding

The main idea is to get the user to input two values and store it into two variables: X and Y. A simple way to do this is:

```
INPUT
Store X
INPUT
Store Y
```

This will store the user input into two variables: X and Y. For the purposes we recommend you set the input and output value types to **DEC** (Decimal) mode. The next part requires us to Load X into the AC, then Add Y, output it then Halt the program.

```
Load X
Add Y
Output
Halt
```

Now, declare these variables into 'temporary' values, note that declaration is usually at the end of the code.

```
X, DEC 0
Y, DEC 0
```

There we have it. The code should something like

```
INPUT
Store X
INPUT
Store Y
Load X
Add Y
Output
Halt

X, DEC 0
Y, DEC 0
```

Multiplication in MARIE

The goal of this section is for you to write some code for multiplying two integers together, especially negative integers. This page explains how to multiply integers that are both non-negative.

Since we can only add and subtract in MARIE, there is no multiply command in MARIE. Instead we can use a combination of `Add` and `Subt` instructions to perform iterative addition and terminate the program once it computes the result.

Explanation

Iterative addition is the simplest method for performing multiplication on integers. For example, the expression `3*4` can be expressed as 3 added to itself 4 times: i.e. `3+3+3+3`. As multiplication is commutative (i.e. order of multiplying numbers does not matter), this is also the same as 4 is added to itself 3 times: `4+4+4`.

Selection statements

There are **no** `if` or `else` statements in MARIE, as implementing them is a bit complicated and would require several instructions. Instead we use `Skipcond`. `Skipcond (num)` skips the next line if a certain condition is true (which is explained in the MARIE Instruction Set Section).

Writing the Code

So what we need for the expression `X*Y` are two variables, `X` and `Y`.
What we can do is to allow the user to input integers into the two variables like so:

```
INPUT
Store X
INPUT
Store Y
```

The idea is we are going to add X, Y Times. For this code, we need to load the variable num into the accumulator, then adds X to it and stores it back into the variable num. Then it loads Y into the accumulator and subtracts 1 from it and stores it back into Y. Then it uses the Skipcond 400statement to check if Y is equal to 0. If it is it will 'Jump' over the Jump loop command and loads the number and outputs the result before halting. Now we are going to look at the main loop - this is because we need to check when Y is equal to 0. So the main loop will look something like:

```
loop,    Load num
         Add X
         Store num
         Load Y
         Subt one
         Store Y
         Skipcond 400
         Jump loop
         Load num
         Output
         Halt
```

Now, we need to declare the variables. So:

```
X, DEC 0
Y, DEC 0
one, DEC 1
num, DEC 0
```

Full Code

```
/ Prompt user to type in integers
INPUT
Store X
INPUT
Store Y

/ Loop for performing iterative addition
loop,    Load num
         Add X
         Store num

         Load Y
         Subt one
         Store Y

         Skipcond 400 / have we completed the multiplication?
         Jump loop / no; repeat loop
         / yes, so exit the loop

/ Output result to user then halt program
Load num
Output
Halt

/ Declare labels here
X, DEC 0
Y, DEC 0
one, DEC 1
num, DEC 0
```