



## UNIT 8B

### Computer Organization: Machine Level Instructions

15110 Principles of Computing,  
Carnegie Mellon University - CORTINA

1

## von Neumann Architecture

- Most computers follow the **fetch-decode-execute** cycle introduced by John von Neumann.
  - Fetch next instruction from memory.
  - Decode instruction and get any data it needs (possibly from memory).
  - Execute instruction with data and store results (possibly into memory).
  - Repeat.

15110 Principles of Computing,  
Carnegie Mellon University - CORTINA

2

# Programming a Machine

- All instructions for a program are stored in computer memory in binary, just like data.
- A program is needed that translates human readable instructions (e.g. in Ruby) into binary instructions (“machine language”).
  - An interpreter is a program that translates one instruction at a time into machine language to be executed by the computer.
  - A compiler is a program that translates an entire program into machine language which is then executed by the computer.

15110 Principles of Computing,  
Carnegie Mellon University - CORTINA

3

# MARS

Memory Array Redcode Simulator

- A simulated computer system that we can use to explore how to run instructions at the machine level.
  - To use this in Ruby, we need to run `include MARSLab`
- We can program this virtual machine in assembly language (a human readable form of machine language) called Redcode.

15110 Principles of Computing,  
Carnegie Mellon University - CORTINA

4

## MARS details

- Memory is simulated by an array of “words”.
- Each word is either an instruction or a data value.
- Instructions are executed in sequence one at a time unless we execute an instruction that causes the virtual machine to “jump” to a location somewhere else in memory for the next instruction.

## Simple MARS Program (simple.txt)

```
labels  opcodes  operands
↓       ↓       ↓
x      DAT #4
y      DAT #7
simple  ADD x, y ; add x to y
          DAT #0 ; 0 is 'halt'
          end simple
```

**DAT** specifies a data value. Data values can also be instructions (e.g. “halt”)

## Running the Program in irb (cont'd)

```

> include MARSLab
=> Object
> m = make_test_machine("simple.txt")
=> #<MiniMARS mem = [DAT #0 #4,...] pc = [*2]>
> m.dump
0000: DAT #0 #4
0001: DAT #0 #7
0002: ADD -2 -1
0003: DAT #0 #0
=> nil

```

Program starts at address 2 in "memory" ↓

x	DAT #4
y	DAT #7
simple	ADD x, y
	DAT #0

add the data 2 words back to the data 1 word back

"memory"addresses

15110 Principles of Computing,  
Carnegie Mellon University - CORTINA

7

## Running the Program in irb (cont'd)

```

> m.step
=> ADD -2 -1
> m.dump
0000: DAT #0 #4
0001: DAT #0 #11 ← y has been updated
0002: ADD -2 -1
0003: DAT #0 #0
=> nil
> m.status
Run: continue PC: [ *3 ]

```

x	DAT #4
y	DAT #7
simple	ADD x, y
	DAT #0

PC = Program Counter  
The PC indicates where the next instruction is located (e.g. address 3).

15110 Principles of Computing,  
Carnegie Mellon University - CORTINA

8

## Running the Program in irb (cont'd)

```
> m.step  
=> DAT #0 #0
```

```
> m.dump  
0000: DAT #0 #4  
0001: DAT #0 #11  
0002: ADD -2 -1  
0003: DAT #0 #0
```

```
=> nil
```

```
> m.status
```

```
Run: halt
```

x	DAT #4
y	DAT #7
simple	ADD x, y
	DAT #0

nothing has changed

The MARS simulator  
executed an instruction  
with opcode 0 (halt)  
and has halted.

15110 Principles of Computing,  
Carnegie Mellon University - CORTINA

9

## Looping Example

Multiply  $x * y$ .

Algorithm: Add  $x$  to an accumulator  $y$  times.

Example: Compute  $5 * 9$ :

```
x          DAT #5  
y          DAT #9  
acc        DAT #0  
mult       ADD x, acc      ; add x to acc  
           SUB #1, y       ; subtract 1 from y  
           JMN mult, y     ; jump to label mult  
           ; if y is not zero  
  
end mult
```

15110 Principles of Computing,  
Carnegie Mellon University - CORTINA

10

## Running the Program in irb

```
> include MARSLab
=> Object
> m = make_test_machine("mult.txt")
=> #<MiniMARS mem = [DAT #0 #5,...] pc = [*3]>
> m.run
=> 28 ← number of instructions executed
> m.dump(0,2) ← dump (display) the
0000: DAT #0 #5      words from "memory"
0001: DAT #0 #0      in this range only
0002: DAT #0 #45
=> nil
```

15110 Principles of Computing,  
Carnegie Mellon University - CORTINA

11

## Example: Fahrenheit to Celsius

$$\text{cels} = (\text{fahr} - 32) * 5 / 9$$

<code>fahr</code>	<code>DAT #82</code>	<code>; fahrenheit value</code>
<code>cels</code>	<code>DAT #0</code>	<code>; store result here</code>
<code>ftmp</code>	<code>DAT #0</code>	<code>; save fahr-32 here</code>
<code>acc</code>	<code>DAT #0</code>	<code>; accumulate answer</code>
<code>count</code>	<code>DAT #5</code>	<code>; counter for mult.</code>

(program continues on next page)

15110 Principles of Computing,  
Carnegie Mellon University - CORTINA

12

## Example: Fahrenheit to Celsius

```
start      MOV fahr, ftmp      } set ftmp = fahr - 32
           SUB #32, ftmp
mult       ADD ftmp, acc      } add ftmp to acc
           SUB #1, count     } 5 times
           JMN mult, count   } (count starts off at 5)
div        SUB #9, acc       } divide acc by 9:
           SLT #0, acc       } subtract 9 from acc
           DAT #0 ; halt     } and add 1 to cels to
           ADD #1, cels      } see how many times
           JMP div           } 9 divides into acc
           end start        } always jump to
                               label div
```

skip next instruction if 0 is less than acc