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.
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.

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.
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)

<table>
<thead>
<tr>
<th>labels</th>
<th>opcodes</th>
<th>operands</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DAT #4</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>DAT #7</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>ADD x, y</td>
<td>; add x to y</td>
</tr>
<tr>
<td>simple</td>
<td>DAT #0</td>
<td>; 0 is ‘halt’</td>
</tr>
<tr>
<td></td>
<td>end simple</td>
<td></td>
</tr>
</tbody>
</table>

**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”

“memory” addresses

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

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

Running the Program in irb (cont’d)

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

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

y has been updated
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

nothing has changed

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

Looping Example

Multiply x \times y.
Algorithm: Add x to an accumulator y times.

Example: Compute 5 \times 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
Running the Program in irb

```ruby
> 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)
0000: DAT #0 #5
0001: DAT #0 #0
0002: DAT #0 #45
=> nil
```

Example: Fahrenheit to Celsius

```ruby
cels = (fahr - 32) * 5 / 9
```

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

(program continues on next page)
Example: Fahrenheit to Celsius

start
MOV fahr, ftmp
SUB #32, ftmp

mult
ADD ftmp, acc
SUB #1, count
JMN mult, count

div
SUB #9, acc
SLT #0, acc
DAT #0 ; halt
ADD #1, cels
JMP div

end start

set ftmp = fahr - 32
add ftmp to acc
5 times (count starts off at 5)
divide acc by 9: subtract 9 from acc and add 1 to cels to see how many times 9 divides into acc
always jump to label div

skip next instruction if 0 is less than acc