von Neumann Architecture

• Modern computers include **control logic** that implements 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.
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 do:
    include MARSLab
- We can program this virtual machine in assembly language (a human readable form of machine language) called Redcode.
MARS Instruction Format

• Memory is an array of **words**. (How many bits per word? Don’t worry about it.)
• Each word is divided into several fields:

```
+--------+------------+------------+
| OpCode | Operand A  | Operand B  |
|        |            |            |
|        |            |            |

Address mode A

Address mode B
```

• OpCodes: ADD, SUB, MOV, JMP, DAT, etc.

MARS Address Modes

1. **Direct** mode: the operand is at the specified address relative to the current instruction.
2. **Immediate** mode (marked by #): the operand is given in the instruction itself.
3. Other modes are provided (e.g., indirect addressing, auto-increment) but we won’t cover them here. Google “RedCode” if you want to know more.
Addressing Example

\texttt{MOV \#500, 10}

- Opcode is MOV (“move data”).
- Move the value 500 to the memory location that is 10 beyond the current instruction.
- Addressing modes used:
  \texttt{A} = immediate (\#), operand is 500
  \texttt{B} = direct, operand is 10

Addressing Example

\texttt{MOV \#500, 10}  
\texttt{ADD \#30, 9}

- Move the value 500 to the memory location that is 10 beyond the current instruction.
- Then add the value 30 to that same location.
Memory Map

Memory Map

Program Counter

MOV #500, 10
ADD #30, 9
DAT #0, #0

DAT #0, #0

DAT #0, #0
Encountering a DAT instruction halts execution.
Stupid MARS Tricks

600: MOV #123, 10
601: DAT #0
602: JMP -2
603: JMP -1
604: JMP -1
605: JMP 1
606: JMP 0
607: JMP 2
608: JMP 1
609: MOV #-42, 1
610: DAT #97

Program Counter
What Does This Do?

• JMN means “jump if non-zero”
• Syntax: \texttt{JMN place, value}

\begin{verbatim}
ADD #1, 2
JMN -1, 1
DAT #-2000
\end{verbatim}

• To test this, do: \texttt{m.run(9999)}

Assembly Language

• Slightly more abstract than machine language.
• Handles some of the drudgery for us.

\begin{verbatim}
myprog
MOV #500, foo
ADD #30, foo
DAT #0, #0
; other stuff ...

foo
DAT #0, #0
end myprog
\end{verbatim}

\begin{tabular}{l}
  \texttt{foo} = 500 \\
  \texttt{foo} = \texttt{foo} + 30
\end{tabular}
Simple MARS Program (simple.txt)

labels    opcodes     operands
    ↓     ↓     ↓
    x    DAT    #4
    y    DAT    #7
    simple    ADD    x,    y    ;    add    x    to    y
                DAT    #0        ;    DAT    will    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”

“memory”addresses

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

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

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

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

The MARS simulator executed a DAT instruction and has halted.
Looping Example

Multiply \( x \times y \) without using the MUL instruction.

Algorithm: Add \( x \) to an accumulator \( y \) times.

Example: Compute \( 5 \times 9 \):

\[
\begin{align*}
x &\quad \text{DAT} \ #5 \\
y &\quad \text{DAT} \ #9 \\
\text{acc} &\quad \text{DAT} \ #0 \\
\text{mult} &\quad \text{ADD} \ x, \ \text{acc} \quad \text{; add } x \ \text{to acc} \\
&\quad \text{SUB} \ #1, \ y \quad \text{; subtract } 1 \ \text{from } y \\
&\quad \text{JMN} \ \text{mult}, \ y \quad \text{; jump to label } \text{mult} \\
&\quad \text{end } \text{mult}
\end{align*}
\]

Flowchart of Multiply Example
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) dump (display) the words from “memory” in this range only
0000: DAT #0 #5
0001: DAT #0 #0
0002: DAT #0 #45
=> nil
```

What Does This Do?

<table>
<thead>
<tr>
<th>puzzle</th>
<th>MOV #1, x</th>
</tr>
</thead>
<tbody>
<tr>
<td>loop</td>
<td>JMP 2</td>
</tr>
<tr>
<td></td>
<td>ADD x, x</td>
</tr>
<tr>
<td></td>
<td>SUB #1, count</td>
</tr>
<tr>
<td></td>
<td>SLT count, zero</td>
</tr>
<tr>
<td></td>
<td>JMP loop</td>
</tr>
<tr>
<td>x</td>
<td>DAT #0 ; 1 2 4 8 16 32</td>
</tr>
<tr>
<td>count</td>
<td>DAT #5 ; 4 3 2 1 0 -1</td>
</tr>
<tr>
<td>zero</td>
<td>DAT #0</td>
</tr>
<tr>
<td>end puzzle</td>
<td></td>
</tr>
</tbody>
</table>
Example: Fahrenheit to Celsius

cels = (fahr – 32) * 5 / 9

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)