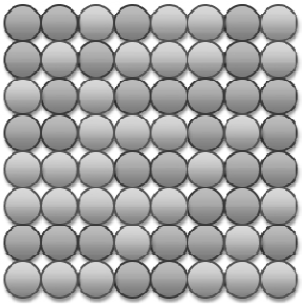

UNIT 8C

Computer Organization: The Machine's Language

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)

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)

Program starts at address 2 in "memory"

```
> 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
```

```
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

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 ]
```

```
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).

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.

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
```

Running the Program in irb

```
> include MARSLab
=> Object
> m = make_test_machine("mult.txt")
=> #<MiniMARS mem = [DAT #0 #5,...] pc = [*3]>
> m.run
=> 28
> m.dump(0,2)
0000: DAT #0 #5
0001: DAT #0 #0
0002: DAT #0 #45
=> nil
```

number of instructions executed

dump (display) the words from "memory" in this range only

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)

Example: Fahrenheit to Celsius