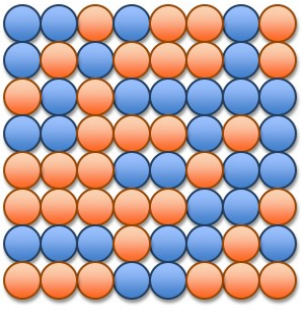


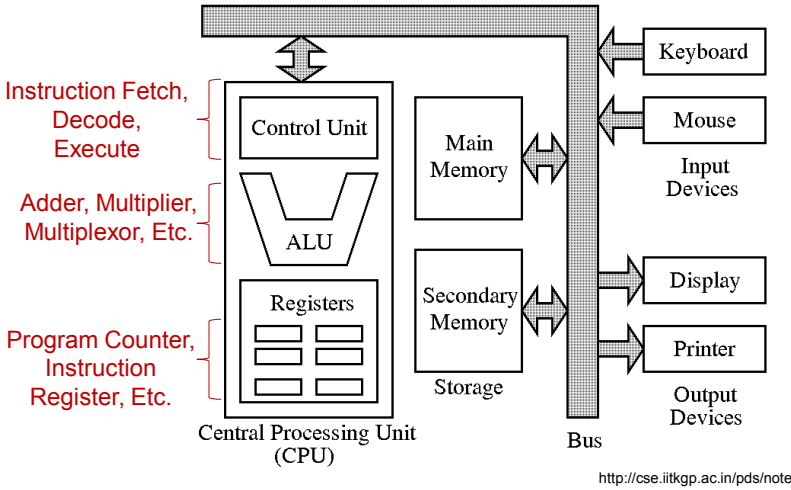
UNIT 8C

Computer Organization: The Machine's Language

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.

Stored Program Computer



http://cse.iitkgp.ac.in/pds/notes/intro.html

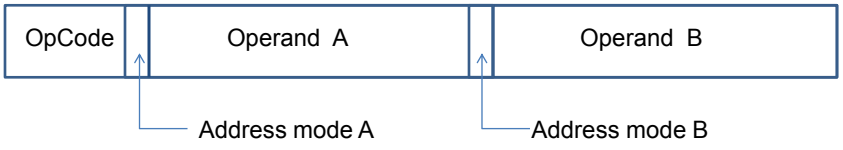
MARS

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:



- 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

```
MOV  #500, 10
```

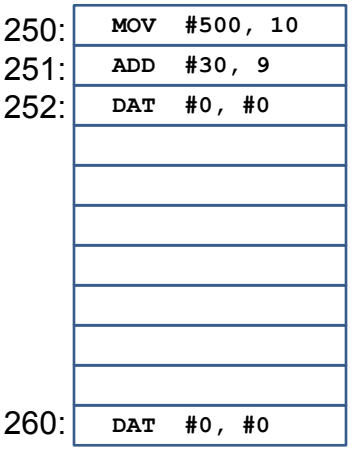
- Opcode is MOV (“move data”).
- Move the value 500 to the memory location that is 10 beyond the current instruction.
- Addressing modes used:
  A = immediate (#), operand is 500
  B = direct, operand is 10

Addressing Example

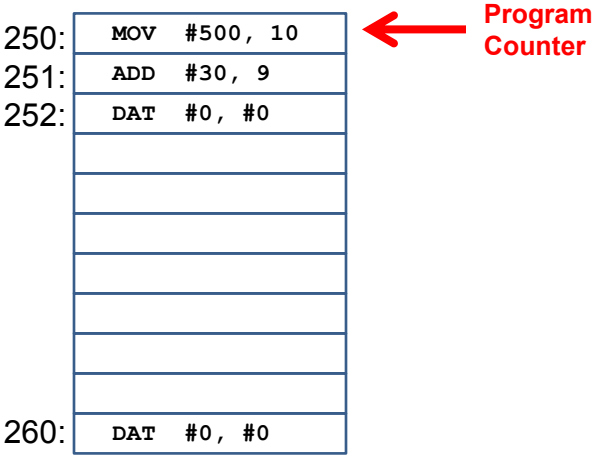
```
MOV  #500, 10
ADD  #30, 9
```

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

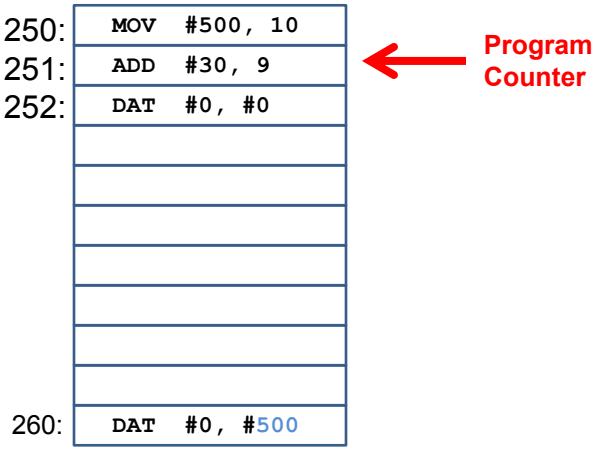
Memory Map



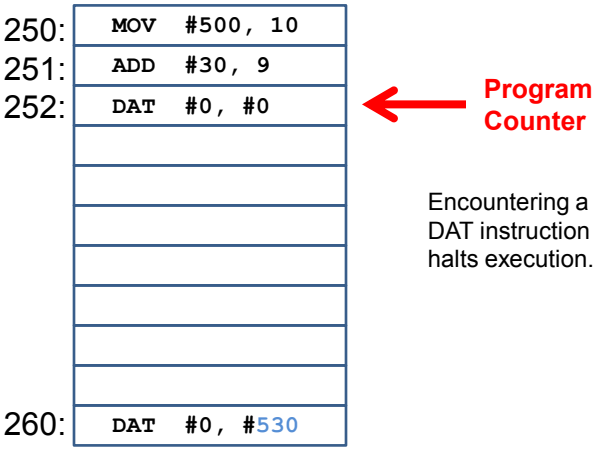
Memory Map



Memory Map



Memory Map



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
```

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
```

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
```

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
```

What Does This Do?

- JMN means "jump if non-zero"
- Syntax: **JMN** *place, value*

```
ADD #1, 2
JMN -1, 1
DAT #-2000
```

- To test this, do: `m.run(9999)`

Assembly Language

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

```
foo = 500
foo = foo + 30
```

```
myprog      MOV #500, foo
            ADD #30, foo
            DAT #0, #0
            ; other stuff ...
foo         DAT #0, #0
            end myprog
```

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)

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 a DAT instruction and has halted.

Looping Example

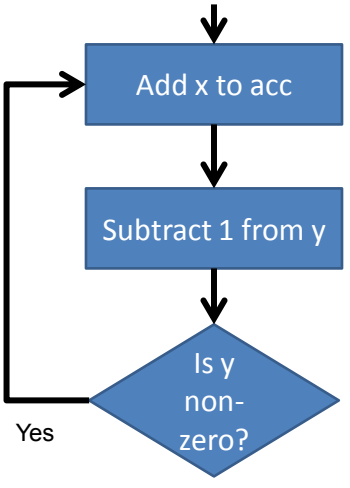
Multiply x * y without using the MUL instruction.
Algorithm: Add x to an accumulator y times.

Example: Compute 5 * 9:

```
begin
  acc = acc + x
  y = y - 1
end while y != 0
```

```
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
```


Flowchart of Multiply Example



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

What Does This Do?

```
puzzle    MOV #1, x
          JMP 2
loop      ADD x, x
          SUB #1, count
          SLT count, zero
          JMP loop
x         DAT #0     ; 1  2  4  8  16  32
count     DAT #5     ; 4  3  2  1   0  -1
zero      DAT #0
          end puzzle
```

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

```
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

skip next
instruction
if 0 is less
than acc

always jump to
label div