basicArchitectures

What we are looking for

-- A general design/organization

-- Some concept of generality and completeness

-- A completely abstract view of machines
a definition
a completely (?) general framework

-- An introduction to a common, standard ISA

-- An introduction to the LC3

Getting in tune with the current scene, listen to

Dave Patterson:

Computer Architecture is Back:
Parallel Computing Landscape

http://www.youtube.com/watch?v=On-k-E5HpcQ

Turing

Completely General, Abstract

Any computing machine (?)
--- Define "Computation"

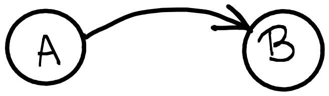
Can prove mathematical statements
--- Define Algorithm
--- The UN-computable
--- Limits on Time/Space

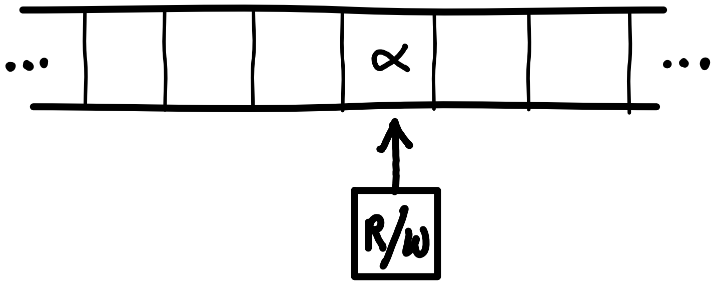
Machine Description
a Table of rules

State	input	output	move	Next State
A	α	λ	L	B
A	β			
A	λ			
…				
A	Ω			
B	α			
B	β			
…				

a "rule"

(input = α output = λ move = L)





Σ = alphabet of symbols = $\{\alpha, \beta, \ldots, \Omega\}$

32-bit symbols → 4G symbol set

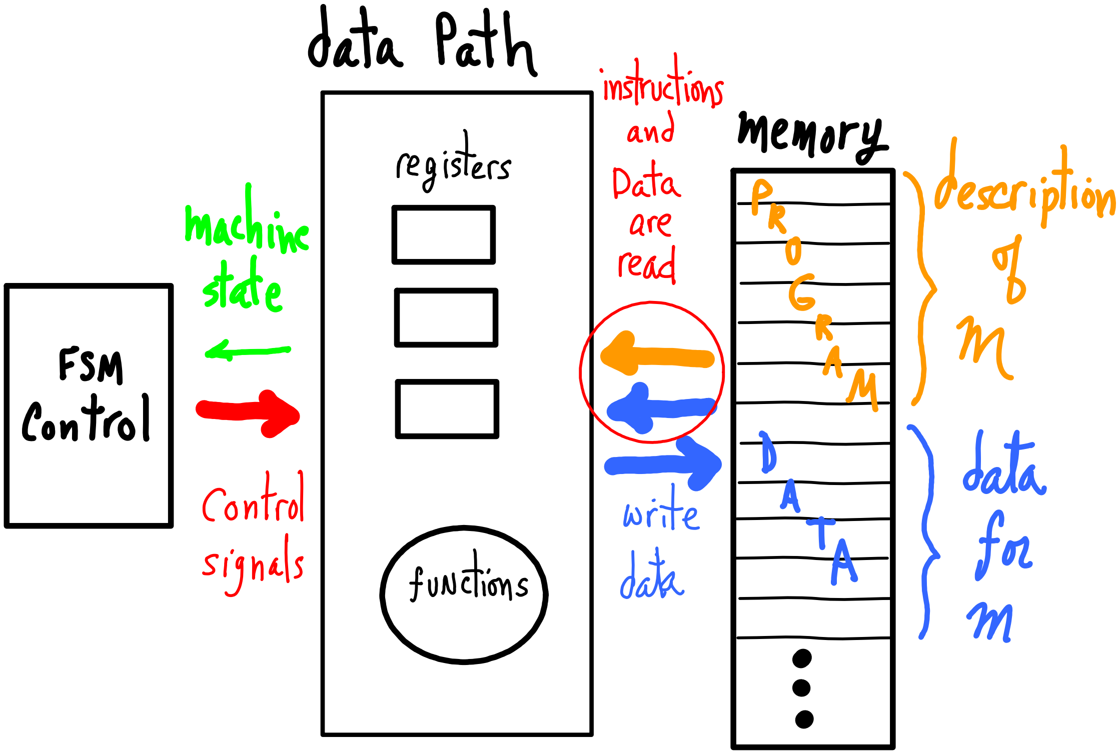
Description of machine M:
--- M's states
--- M's symbols
--- M's rules

Von Neumann

Simulate M

Building calculating machines

-- SSEM
-- Manchester Mark 1
-- Ferranti Mark 1
-- IAS
-- ENIAC

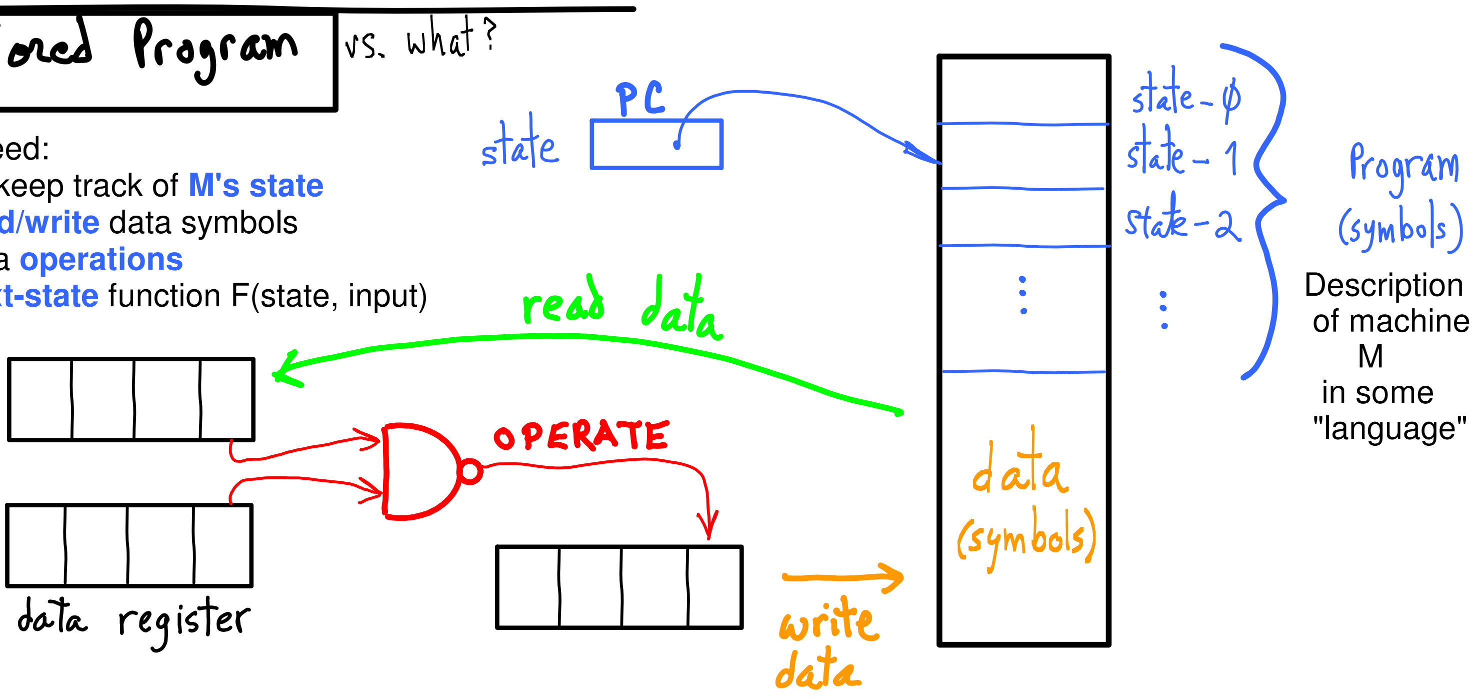


Program describes completely, how
--- Machine M changes state
--- next state depends on current state and current input
--- output depends on input and M's current state
--- M "moves" to another location to read next symbol

STored Program

vs. what?

We need:
1) To keep track of **M's state**
2) **read/write** data symbols
3) data **operations**
4) **next-state** function F(state, input)



Description of machine M in some "language"

desriptions of functions

nand(a, b) ===> f
nand(c, d) ===> e

is sufficient for any function

We need a language rich enough to describe

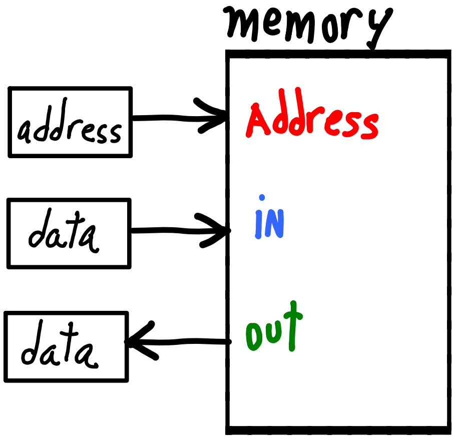
---- **any function**: next-state or data operation
---- **read/write** operations
---- **state change** for M

with this we can **describe** ANY machine, **and simulate** it.

Memory == an array
address == array index.
Two Operations:

store data: **in** ==> **Memory[address]**

retrieve data: **out** <== **Memory[address]**



And, what else?

NB--It's not obvious what capabilities we need.
Can we find a model that could tell us that?

convenience/performance

ADD SUB
MULTIPY DIVIDE
...

read/write data, stack operations

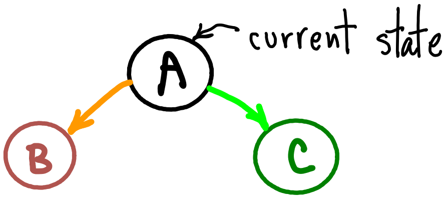
Load Reg, Address
Store Reg, Address

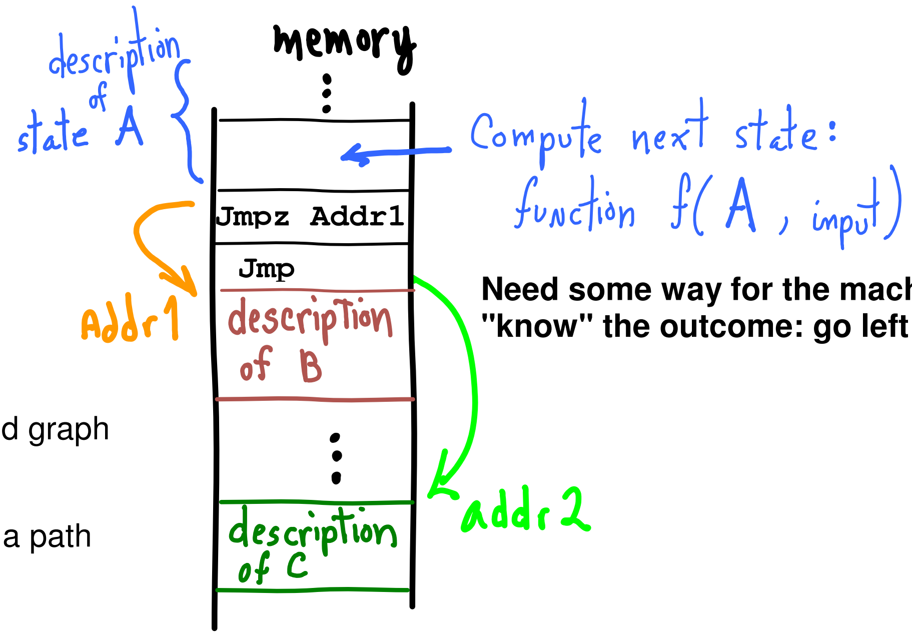
Push Reg
Pop Reg

index/address	content (symbols)
000	α
001	β
010	λ
011	δ
100	e
101	α
110	λ
111	σ

Content of Memory[001]
is symbol β

Necessary for TM completeness:
--- Changing M's state
--- Go to other part of description
(depending on data)





Need some way for the machine to "know" the outcome: go left or right.

Recreate branched graph in linear memory.

Execution follows a path through graph.

Turing View of a Computer

--- a machine can be described as a table of rules
(current-state, input ===> output, next-state)

--- input, a symbol, is read from "memory"
--- a rule is applied according to
--- what the current state is
--- which symbol was read
--- output, a symbol, goes to "memory"
--- the machine changes state
--- repeat

von Neumann View of a Computer

--- a "program" is a sequence of
step-by-step instructions

--- instructions are read from "memory"
--- the instruction is read into a "register"
--- a controller "executes" the instruction
--- data is read from "memory"
--- a "register" remembers what was read
--- an operation changes the data
--- a register stores the result
--- the result is written to "memory"
--- repeat

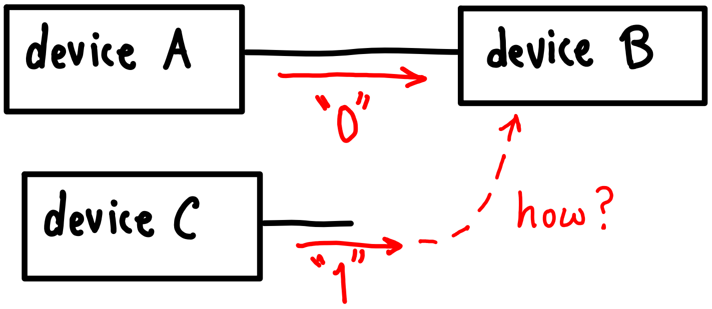
How they correspond:

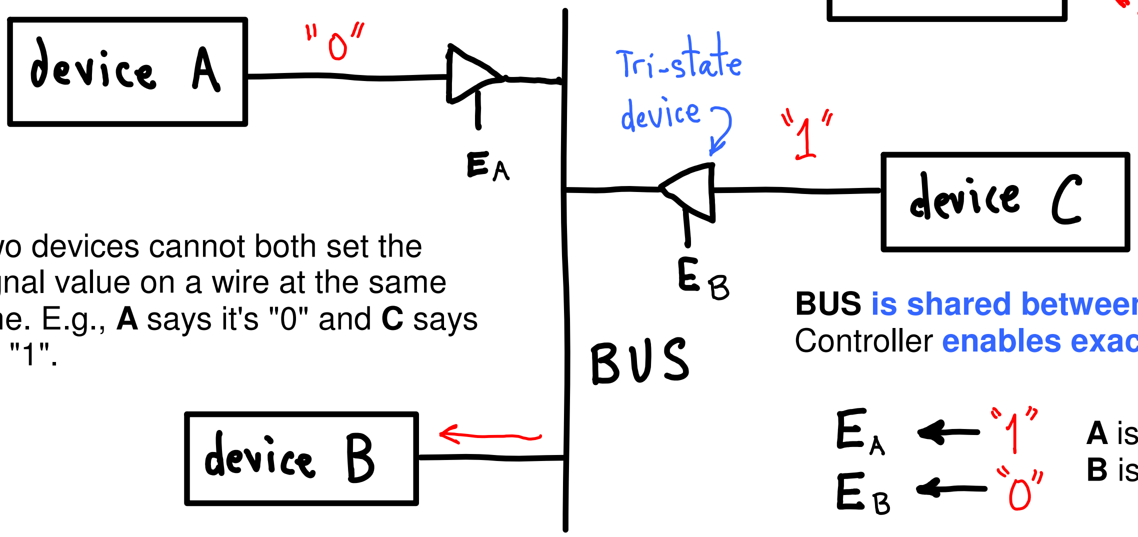
Turing	**von Neumann**
a State	A section of the program
a change of state	jumping to a different section
a rule's output part	a section that produces a new output
a rule's state-change part	a section that calculates the next jump

Busses (because you see them here and there)

signals travel on a wire:
A sends a "0" to **B**.

C cannot use same wire,
even at a different time?
A's output tied to C's output?





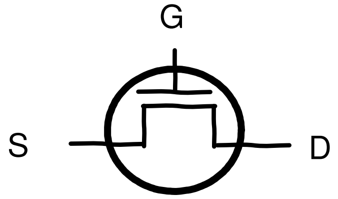
Two devices cannot both set the signal value on a wire at the same time. E.g., **A** says it's "0" and **C** says it's "1".

BUS is shared between devices.
Controller **enables exactly one at a time**; e.g.,

$E_A \leftarrow$ "1" **A** is **connected** to **BUS**
$E_B \leftarrow$ "0" **B** is **disconnected** from **BUS**

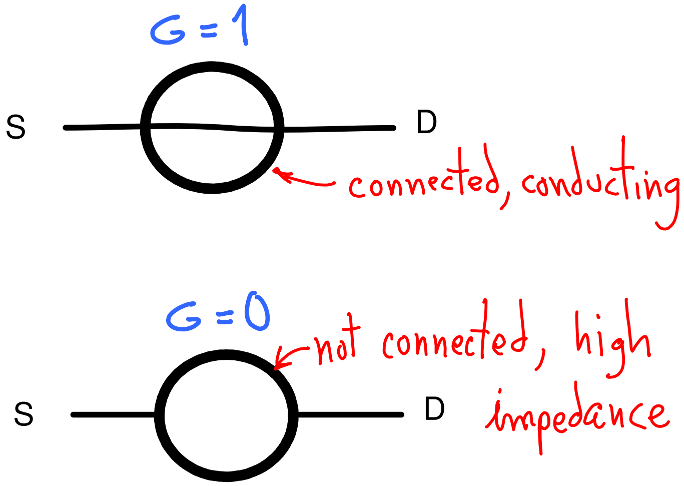
A "tri-state buffer"

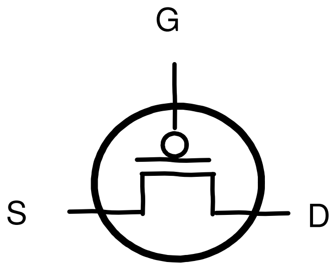
CMOS



n-transistor
– switch

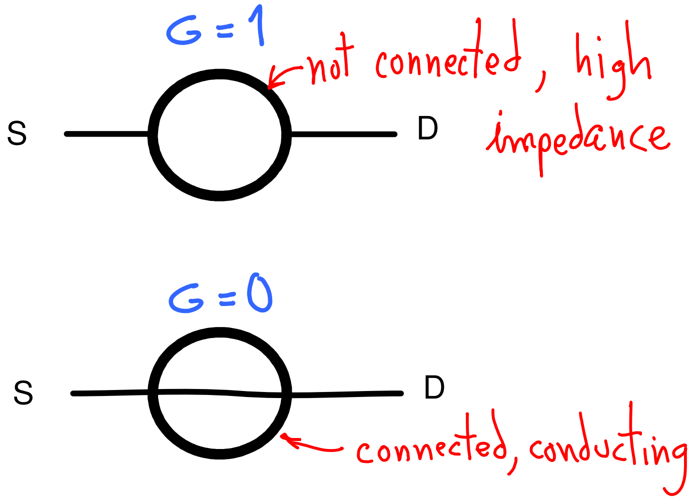
Passes a "0" cleanly

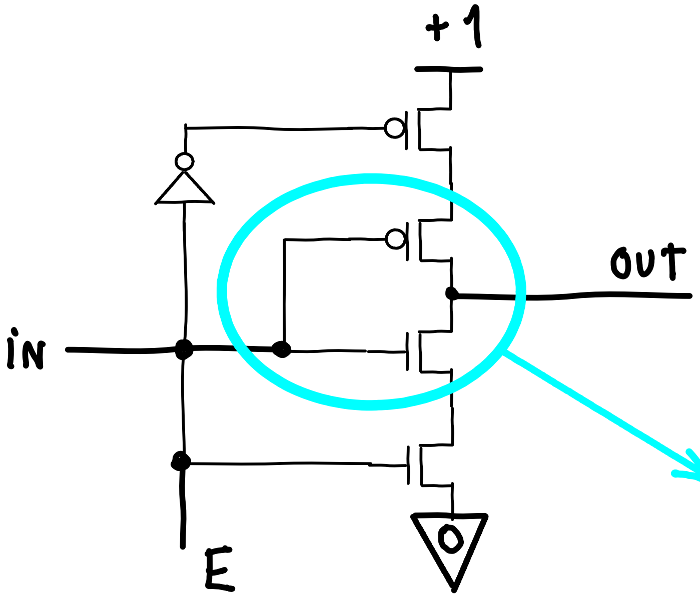




p-transistor
– switch

Passes a "1" cleanly

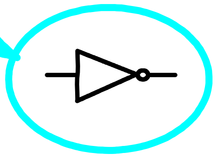




A Tri-state, inverting buffer

E = 1 : Passes 1 or 0 cleanly (inverted)

E = 0 : Passes neither, no conduction,
high impedance



an inverter

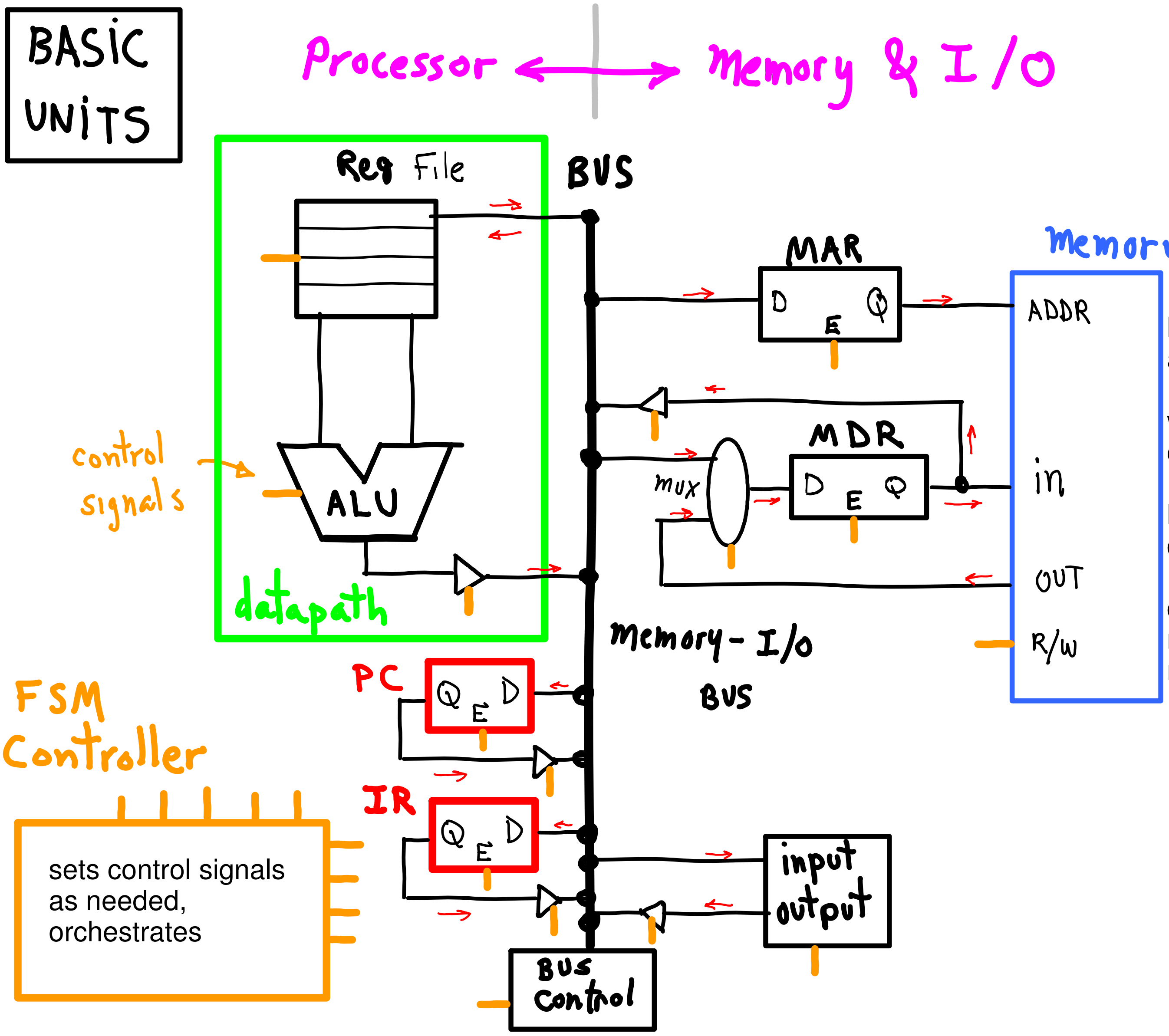
in == 0 then out == 1
in == 1 then out == 0

Verilog wire signal values

X -- unknown (for various reasons)
Z -- high impedance

1 -- voltage for "1"
0 -- voltage for "0"

BASIC UNITS



sets control signals as needed, orchestrates

MAR, memory address register:

which memory cell is accessed.

MDR, memory data register:

data sent to, or received from memory.

Machine Cycle

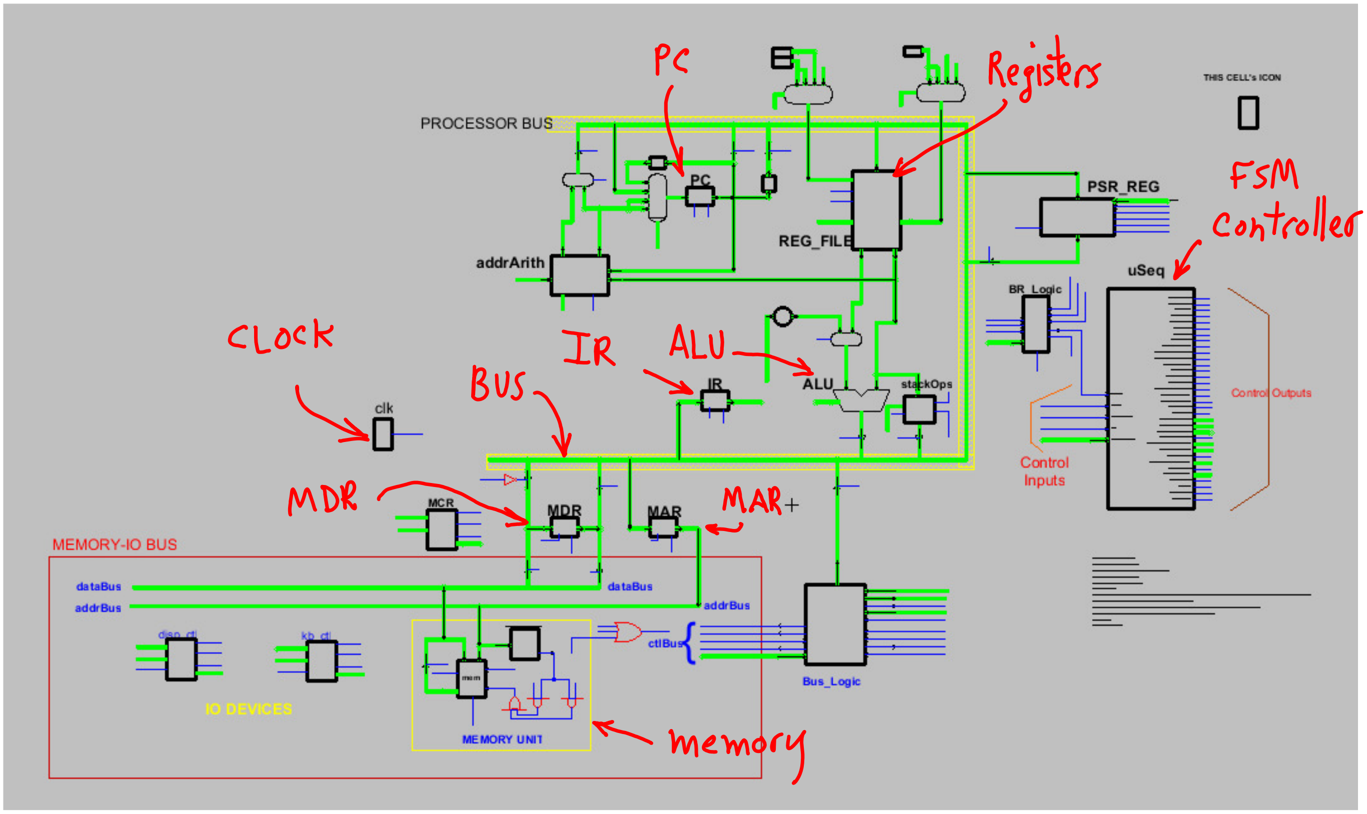
================

execution phases

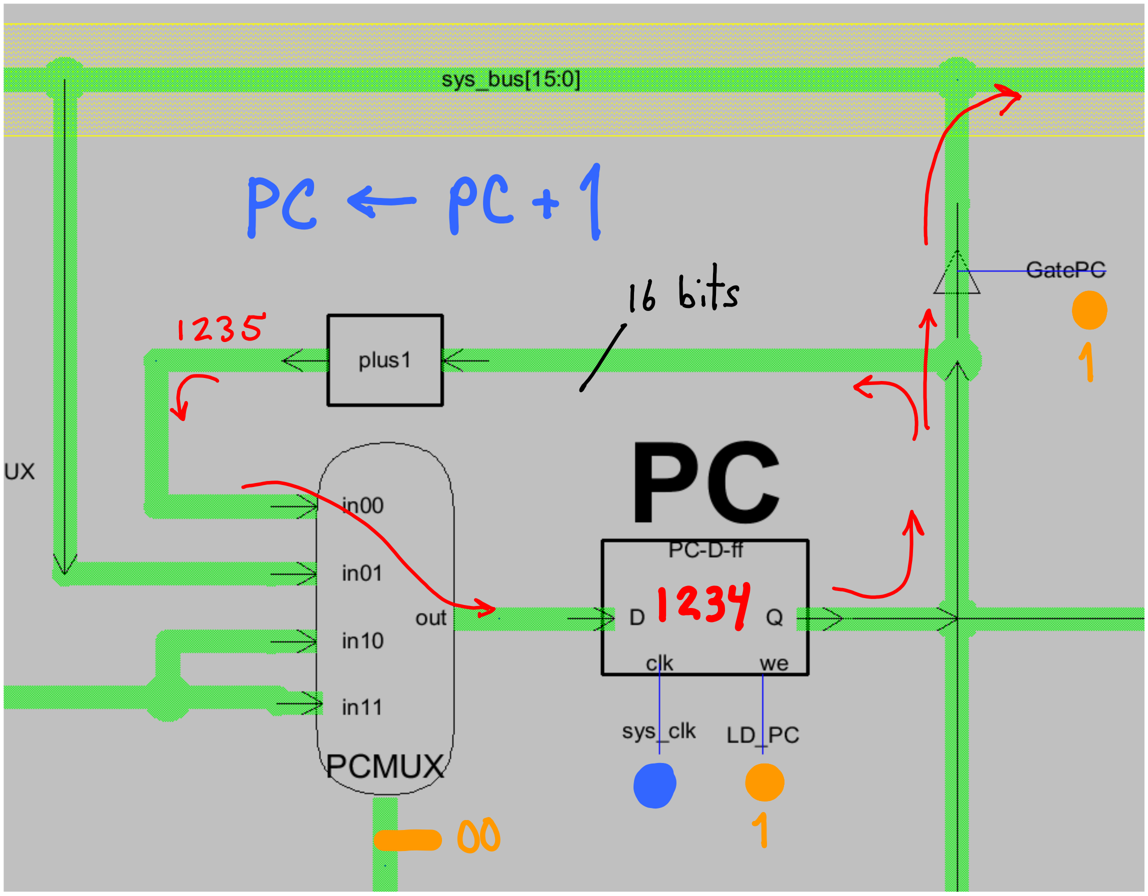
Fetch instruction
Decode instruction
Evaluate address
Fetch operands
Execute instruction
Store result

forever, or?

LC3, a von Neumann architecture



Instruction Execution

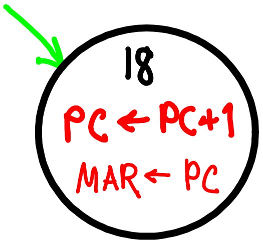


PHASE:
Fetch instruction

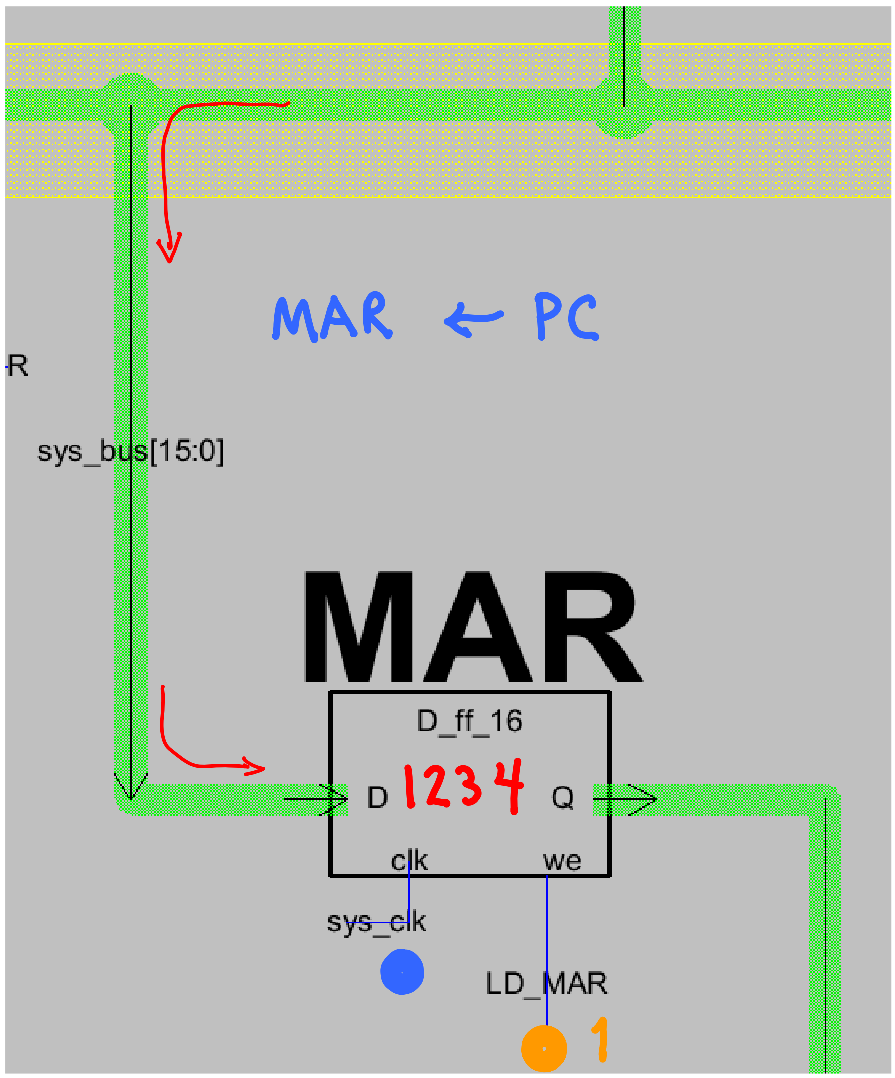
-- 1st step, state 18

PC <== PC+1
MAR <== PC

(Happens in parallel, finalized when clock pulse arrives.)



FSM Controller
State, 18

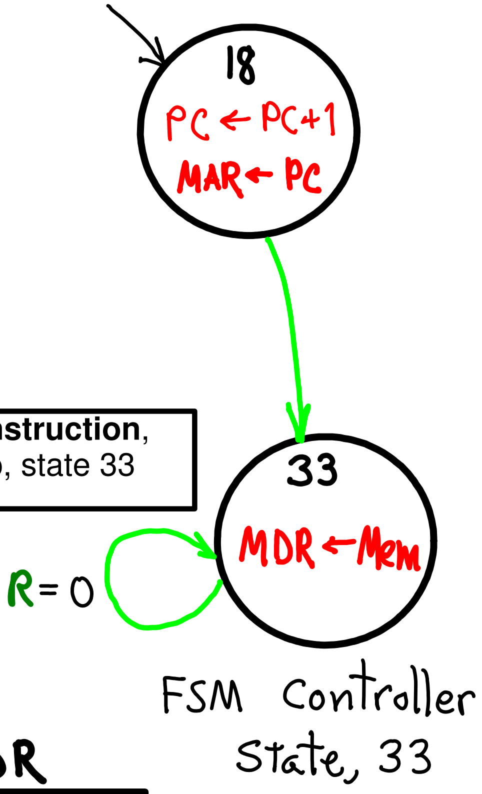


(continue)
Fetch instruction, first step:

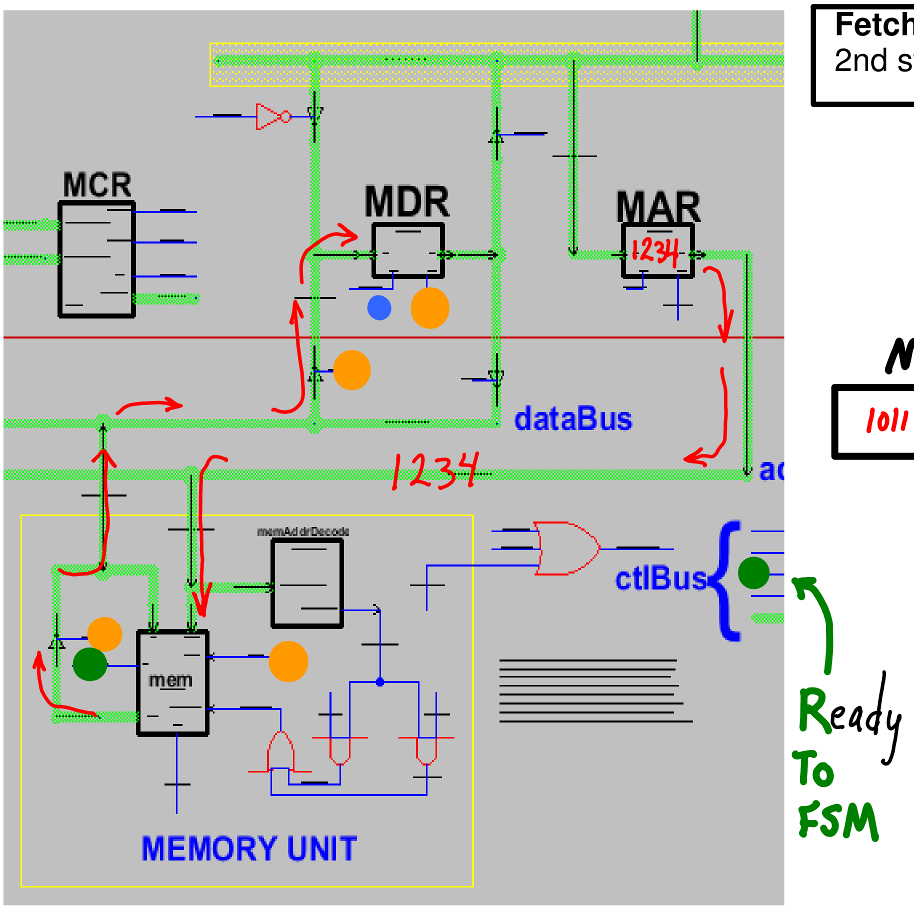
Load MAR w/ address in PC

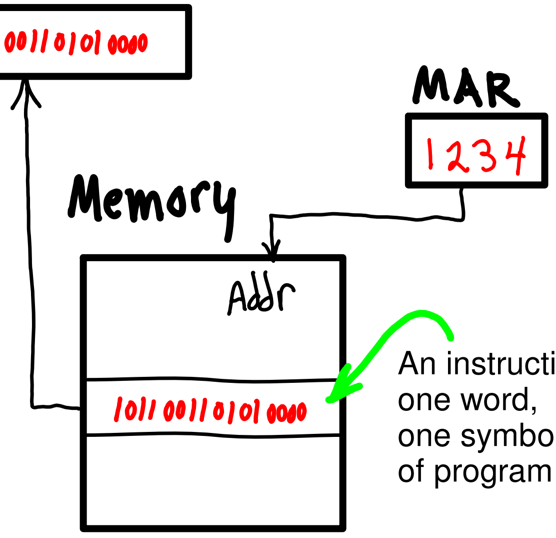
FSM Controller must set
signal values for State-18:

GatePC <== 1'b1
PCMUX <== 2'b00
LD_PC <== 1'b1
LD_MAR <== 1'b1



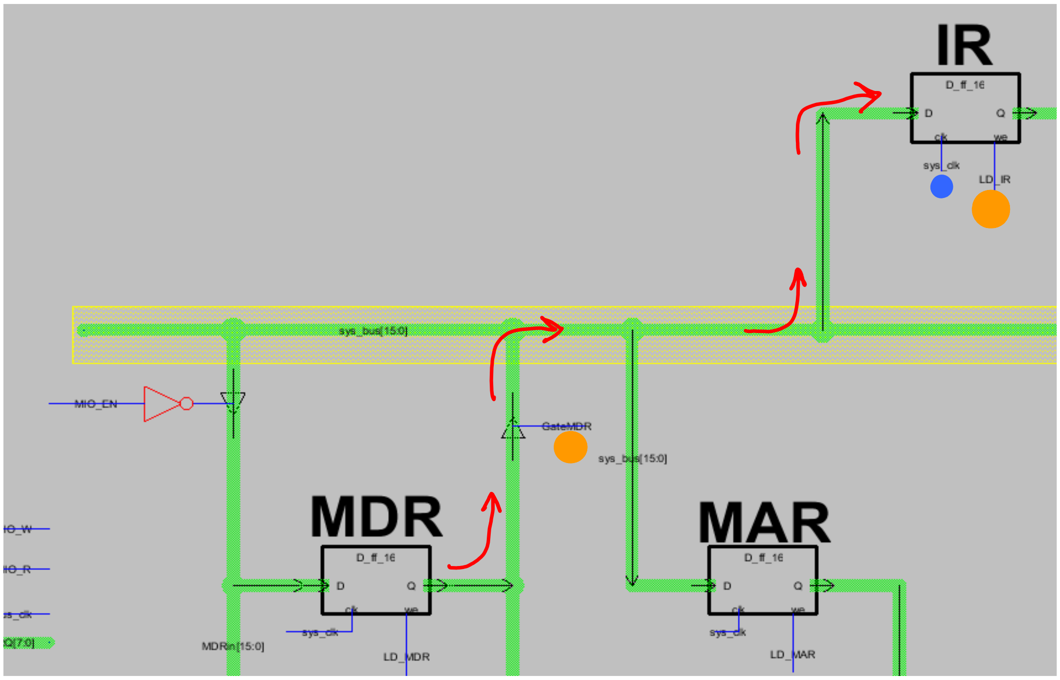
Fetch-instruction,
2nd step, state 33

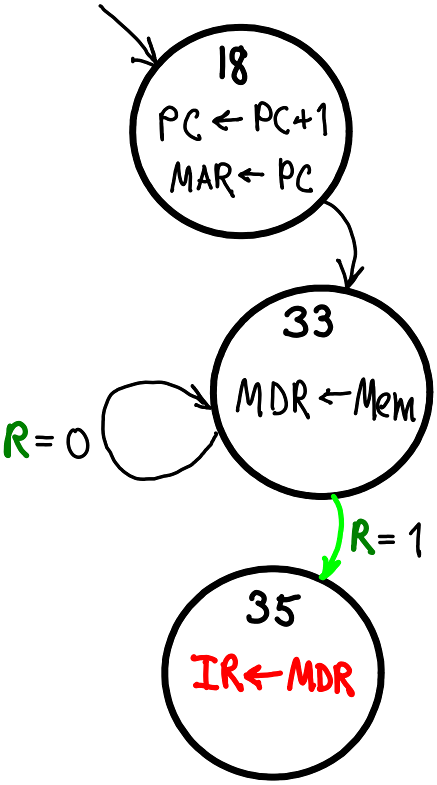




An instruction,
one word,
one symbol
of program.

Continue to next state when R ← 1





Fetch-instruction,
3rd step

Instruction is remembered, ie., "registered" in **IR**

Control signals for state-35:

GateMDR <== 1'b1
LD_IR <== 1'b1

This completes the **Fetch-instruction** phase.

Overall effects:

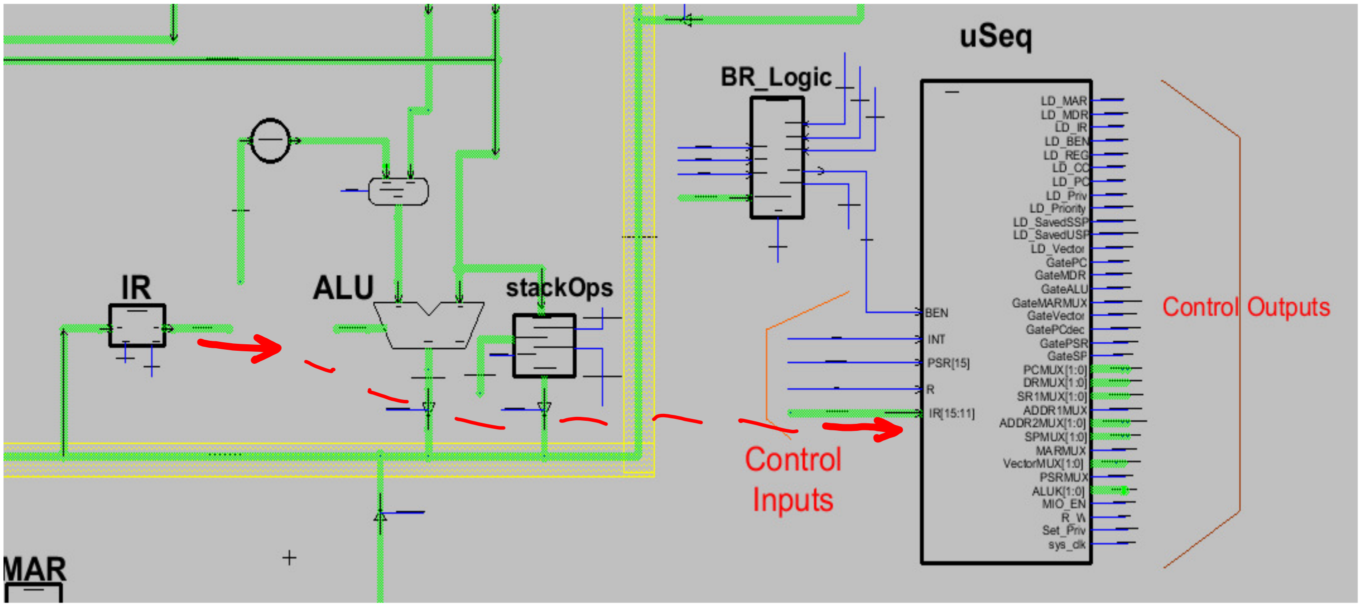
IR <== 16'b 1011 0011 0101 0000 (16-bit instruction, from memory)

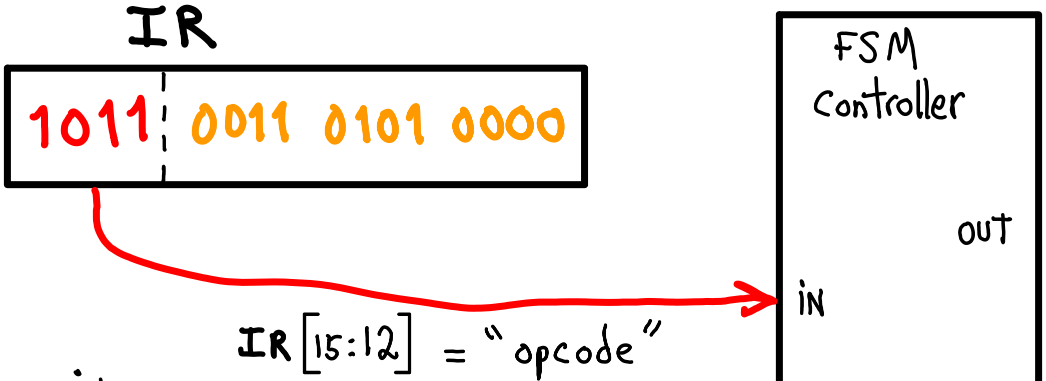
PC <== 16'b 0001 0010 0011 0101 (a 16-bit memory address)

Or, to put it in other ways:

IR <== 16'h B350
IR <== Mem[16'h 1234])
PC <== 16'h 1235

Decode phase, what instruction is this?





Decode phase

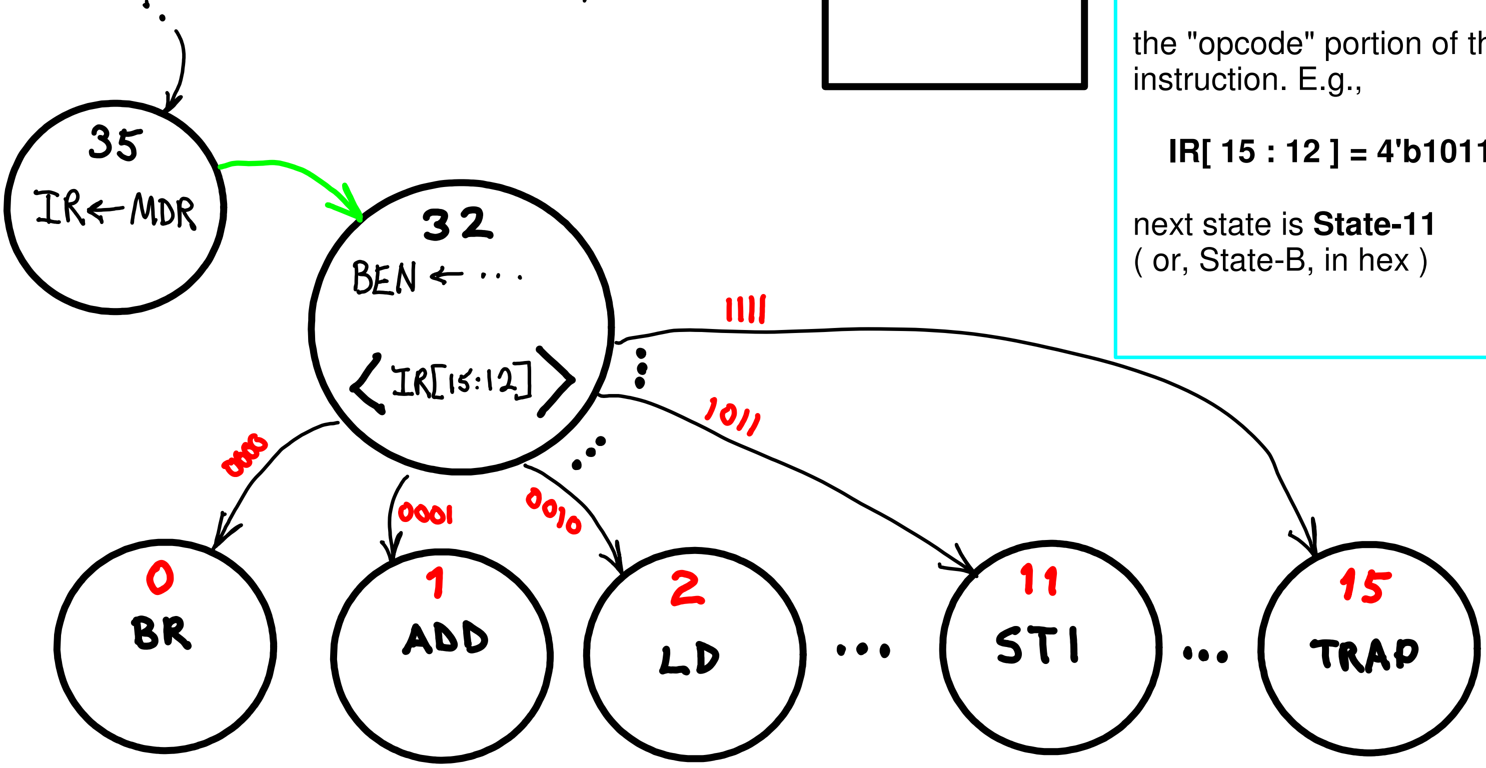
Next state of controller is determined by

IR[15 : 12]

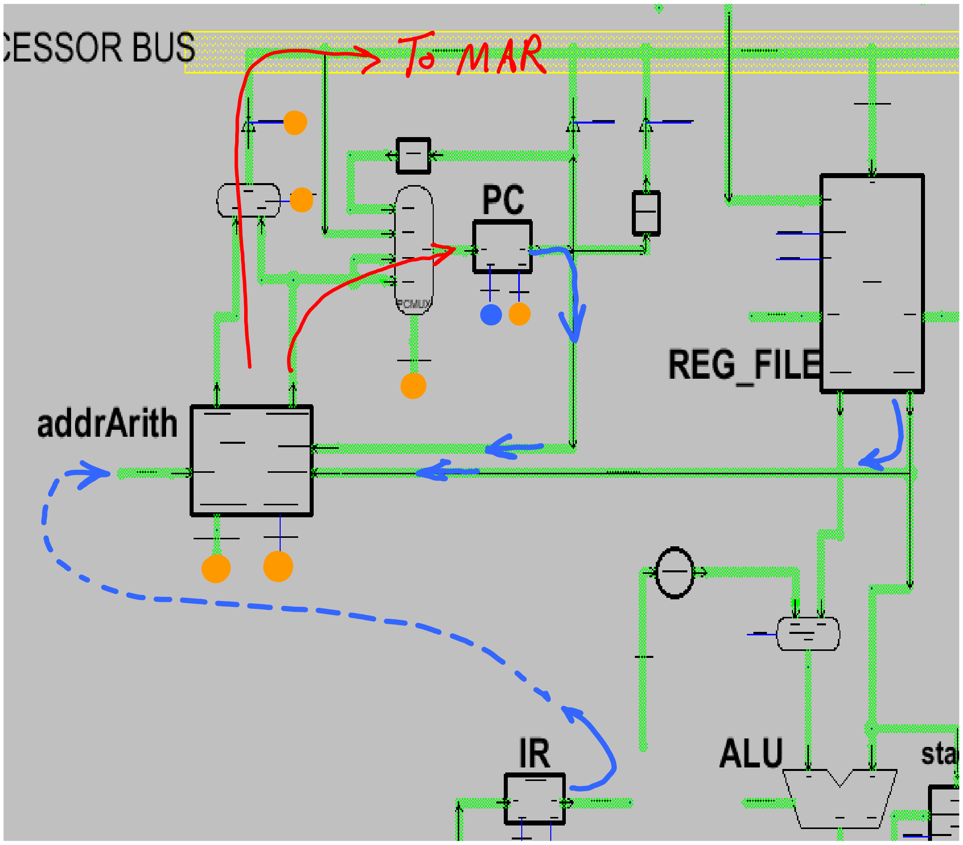
the "opcode" portion of the instruction. E.g.,

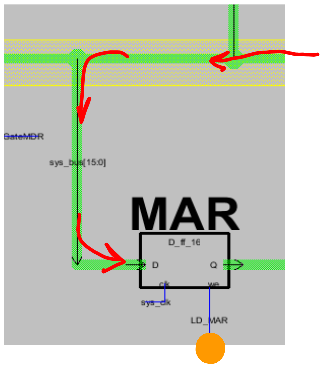
IR[15 : 12] = 4'b1011

next state is **State-11**
(or, State-B, in hex)



Evaluate Address Phase





Evaluate Address phase

Calculated in **addrArith**.
Sources for calculation are:

--- RegFile (a register)
--- PC
--- IR

Resulting address sent to:

--- PC, **change state**
jump to different part of program
(instructions JMP, BR, INT, TRAP)

OR

--- MAR, **data transfer**
memory-to-register
register-to-memory
(instructions LDR, STR)

What's a Register File ?

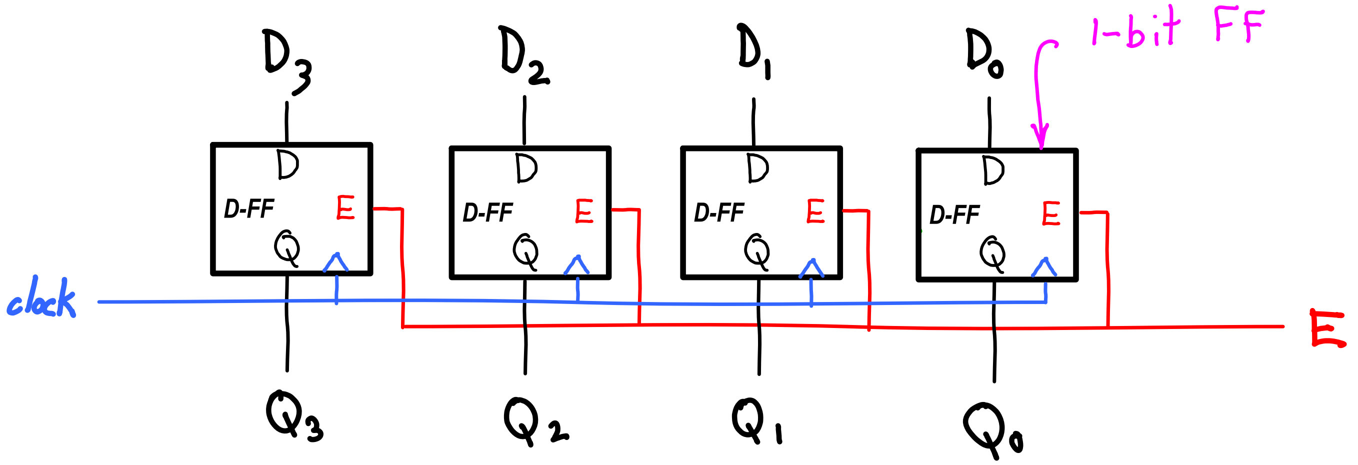
Some registers (flip-flops),
and **a way to select,**
--- **which to output**
--- **which to write**

a register?

N 1-bit D-FlipFlops

4-bit register

Holds a 4-bit "word"



E=1: data D written to FF **on next clock pulse**

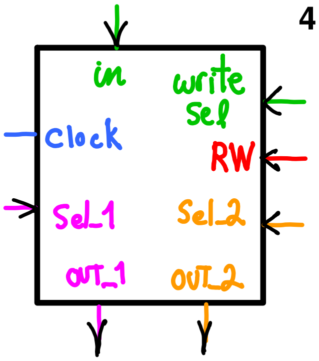
4-register RegFile

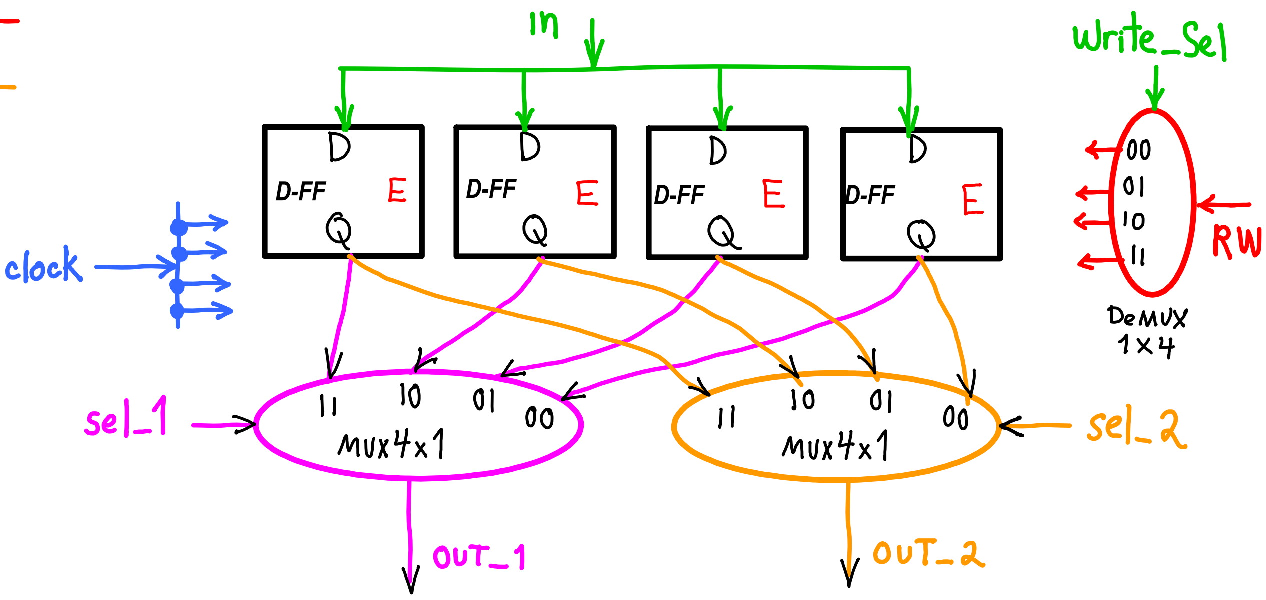
4 N-bit registers

data output
--- **out_1**
--- **out_2**
data select
--- **Sel_1**
--- **Sel_2**

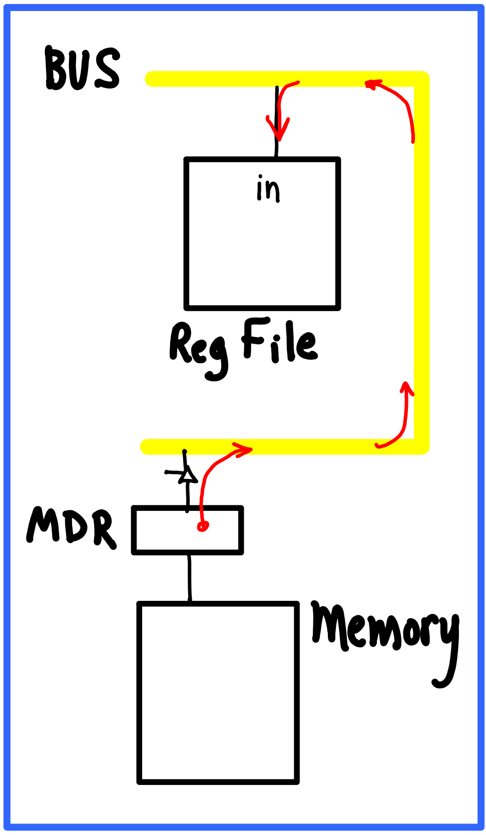
data input
--- **in**
write select
-- **write_Sel**
read/write control
--- **RW**

clock input
--- **clock**





Fetch Operands Phase

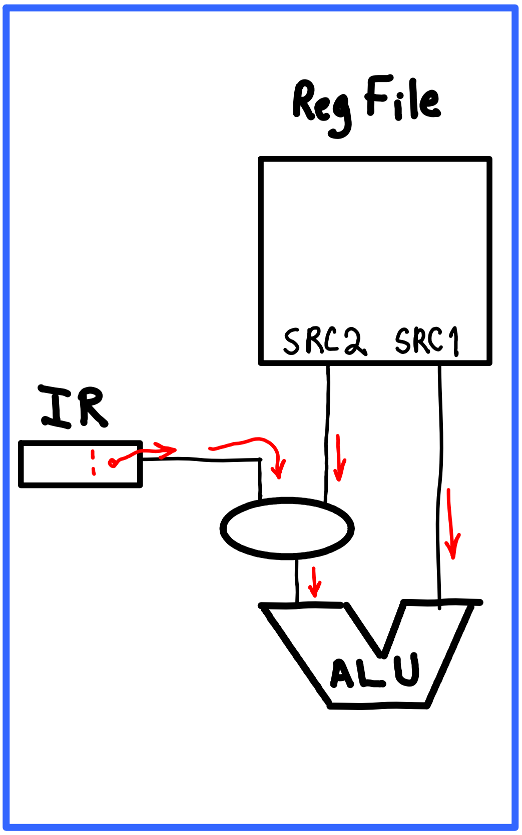


move data from memory to a **register** (e.g., LD)

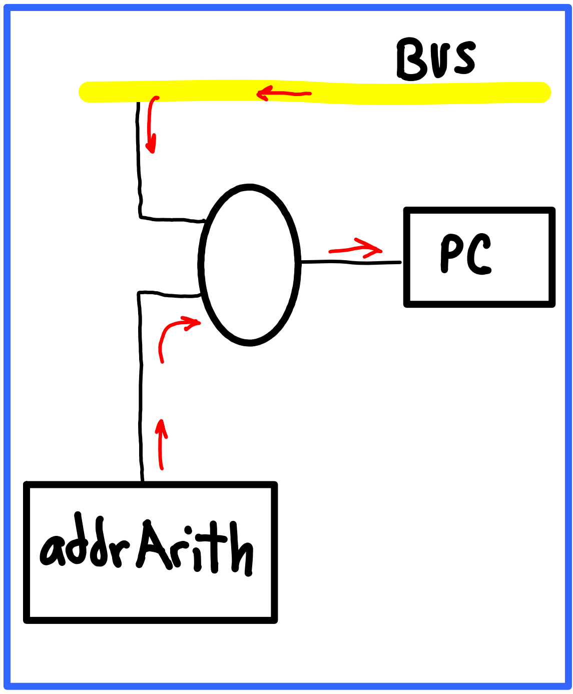
OR

send data to ALU from a **register** **or** some of the **IR**'s bits

LC3's "load from memory" instructions (LD, LDR, LDI) do nothing after copying from the **MDR** to a **register**.



Execute Phase

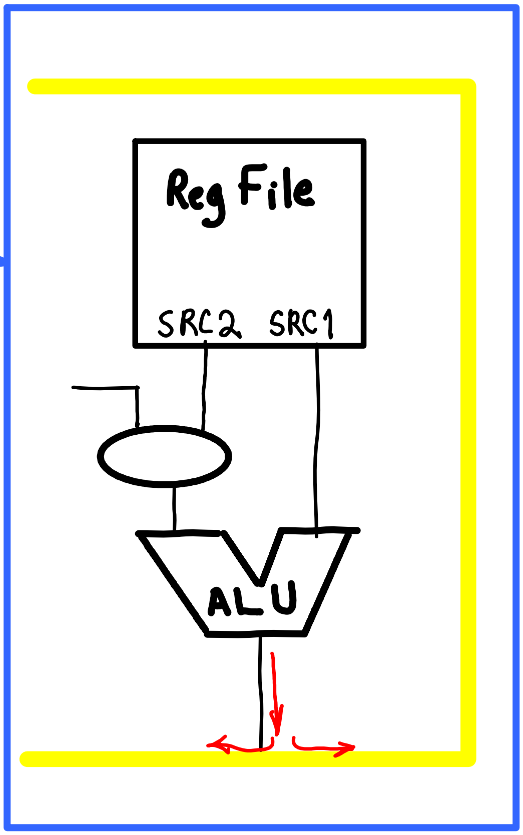


ALU operation and making result available on **BUS** (e.g., ADD, SUB, NOT)

OR

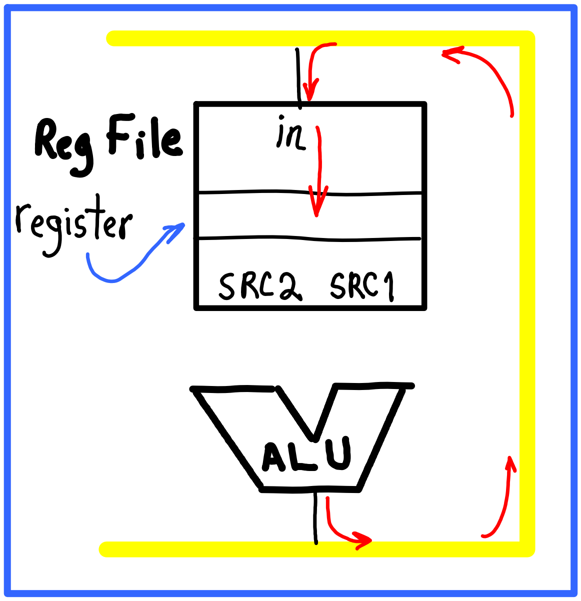
Load the PC with an address calculated in Evaluate Address phase, or from another source (e.g., interrupt vector).

Instructions that do (2.) are **JMP**, **BR**, **TRAP**, as well as system generated action from hardware, **interrupts** and **exceptions**.



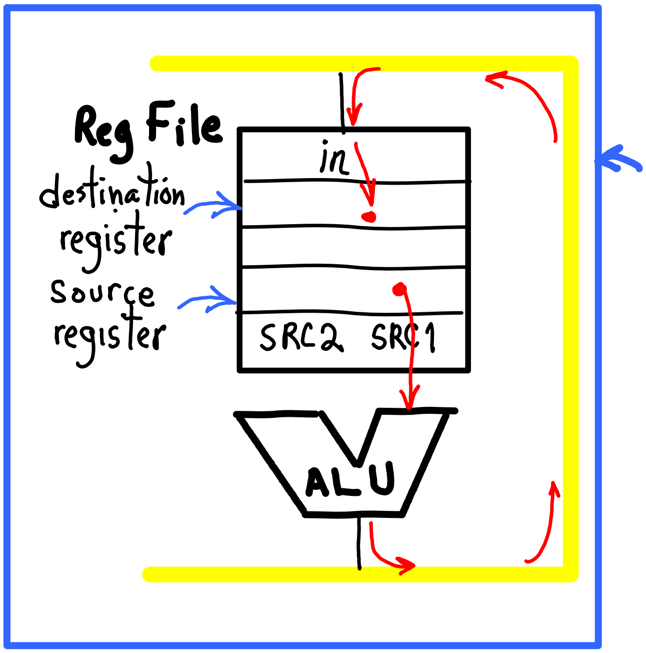
Store Phase

write ALU result to a **register**.



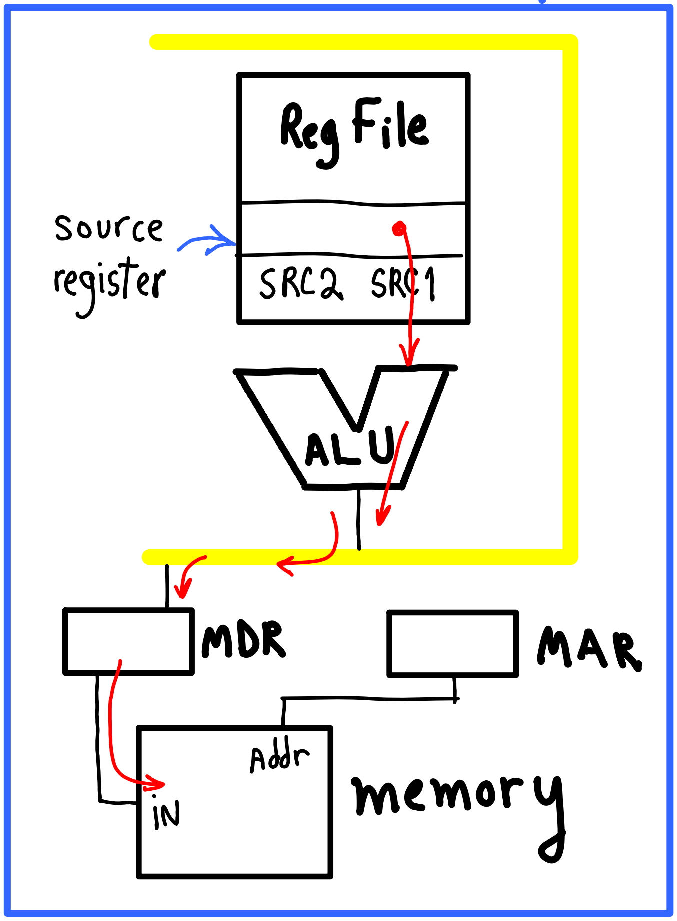
OR

copy from **source register** to **destination register**.



OR

copy from **register** to **memory location**.



Execution Phases

Machine Cycle

================

Fetch instruction
Decode instruction
Evaluate address
Fetch operands
Execute instruction
Store result

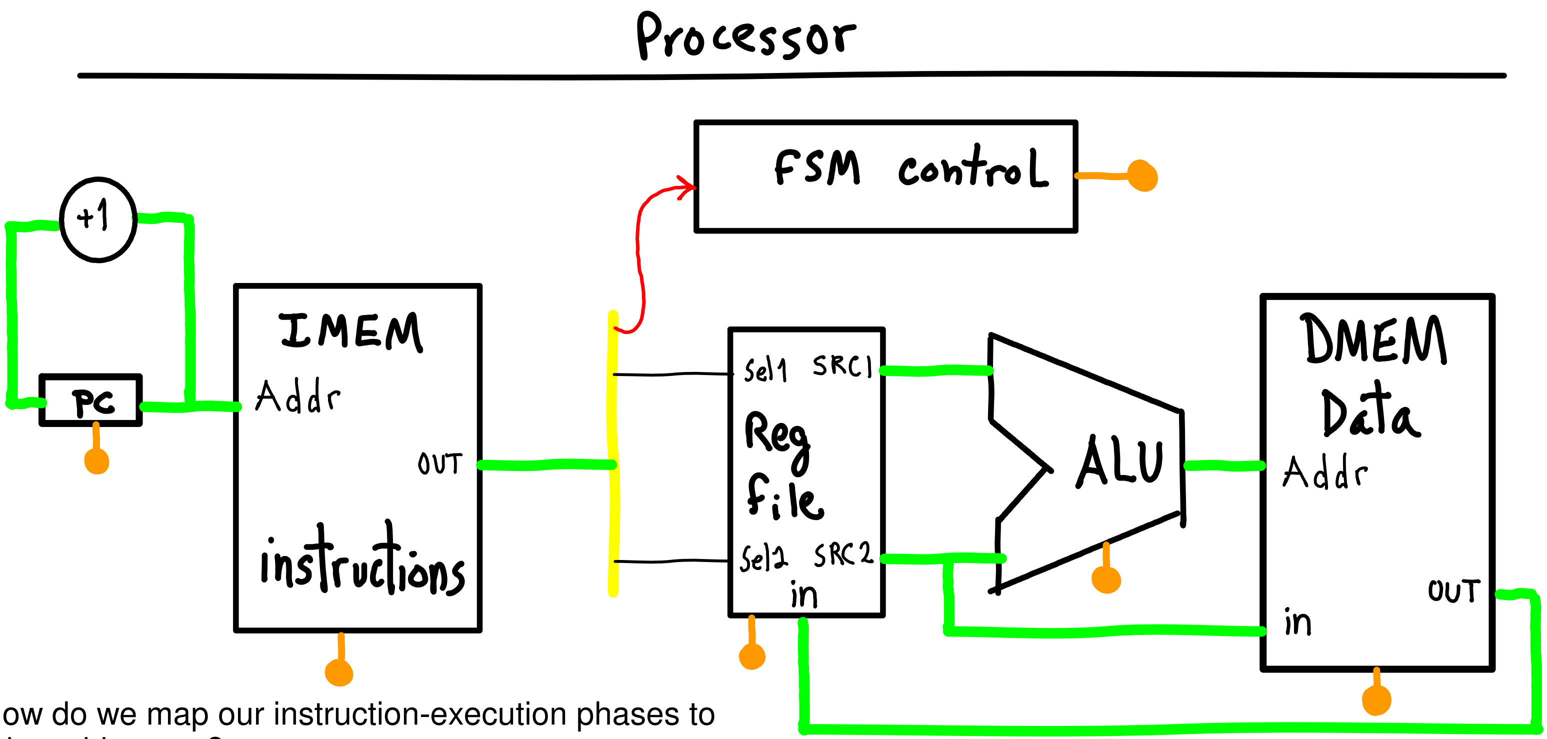
--- No instruction uses every phase.

--- Multiple instructions **could be** simultaneously in different phases. (How about same phase?)

--- Some phases must wait for the previous phase to complete (eg., memory access)

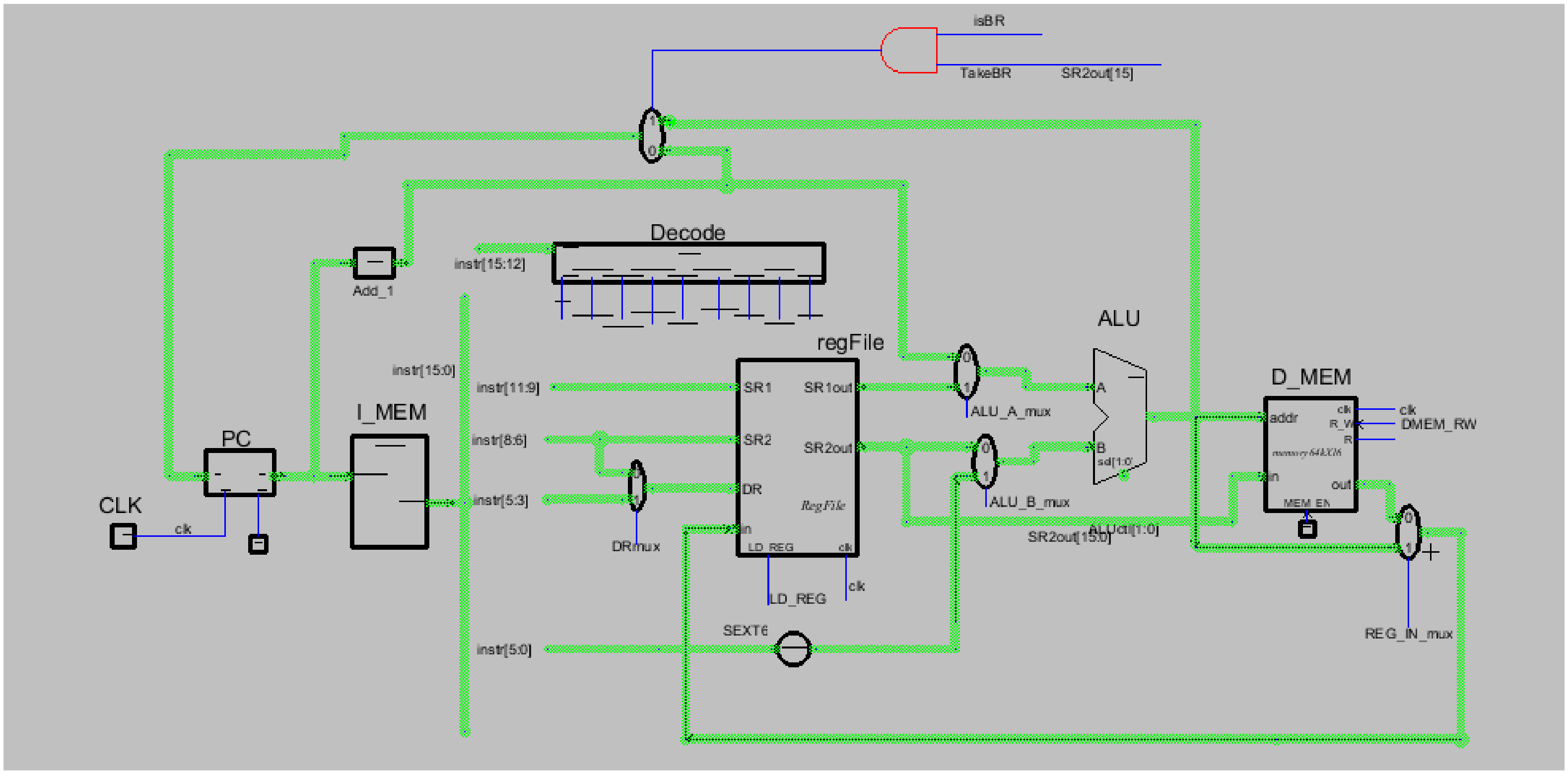
Harvard Architecture

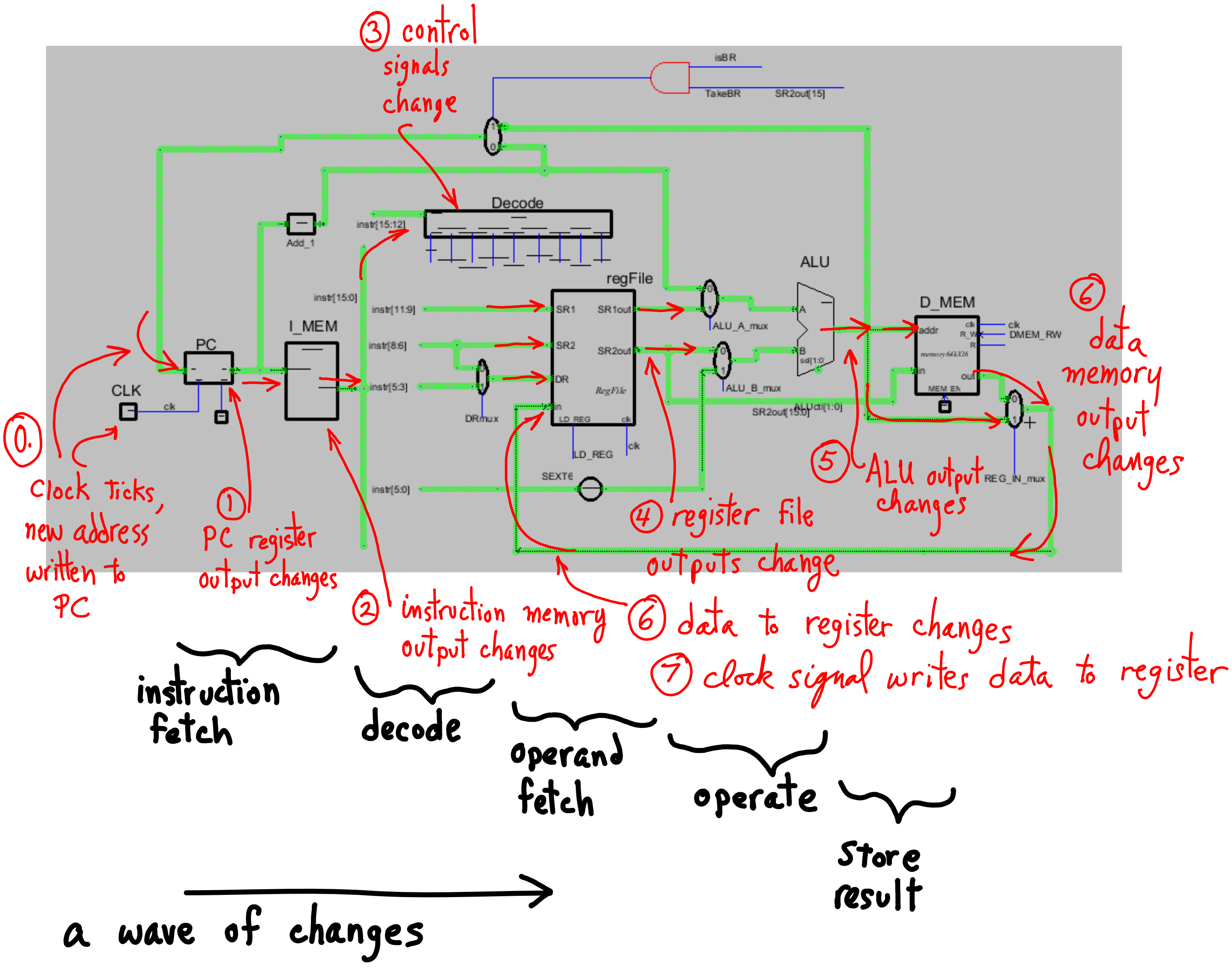
Two memories, one for instructions, one for data.
Layed out as a "pipeline" ==> more parallelism.



How do we map our instruction-execution phases to this architecture?

LC4, Harvard Architeture version of LC3





③ control signals change

0. Clock Ticks, new address written to PC

① PC register output changes

② instruction memory output changes

④ register file outputs change

⑤ ALU output changes

⑥ data memory output changes

⑥ data to register changes

⑦ clock signal writes data to register

instruction fetch

decode

operand fetch

operate

Store result

a wave of changes

LC4, Only **ONE cycle** per instruction

--- **Two memories**, instructions and data
--- **controller sets all control signals** for proper flow
--- **Does not need IR**, instruction memory output is stable

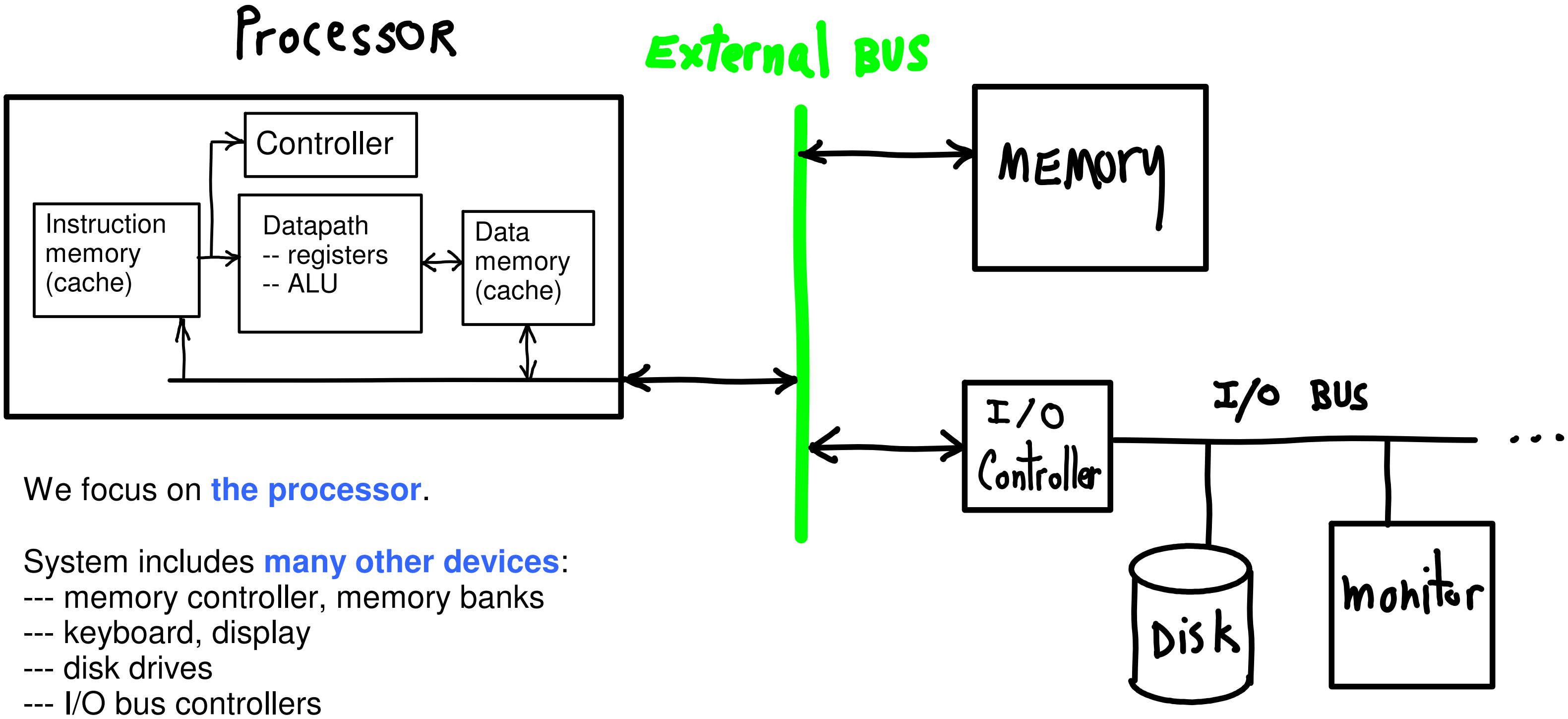
vs.

LC3, **many cycles** per instruction

--- **One memory**, used first for instruction, then data
--- **controller handles parts of operation** at a time
--- **Needs IR** to remember instruction

System (von Neumann)

At large scale,
system is von **Neumann Architecture**.
At small scale,
processor might be **Harvard Architecture**.



We focus on **the processor**.

System includes **many other devices**:
--- memory controller, memory banks
--- keyboard, display
--- disk drives
--- I/O bus controllers
--- network interfaces
--- communication channels for video
--- GPUs
--- etc.

2 basic Architectures

— von Neumann
— Harvard

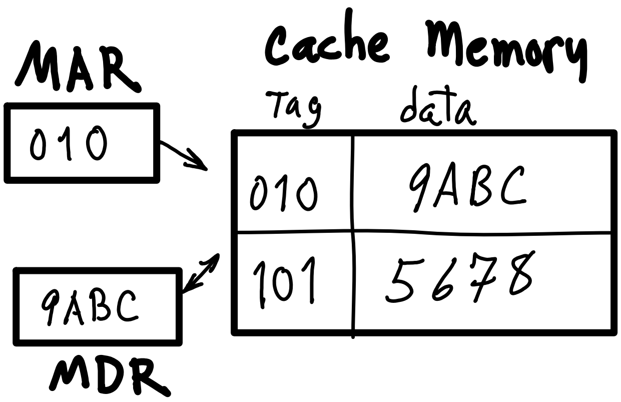
Processors can be implement either way.

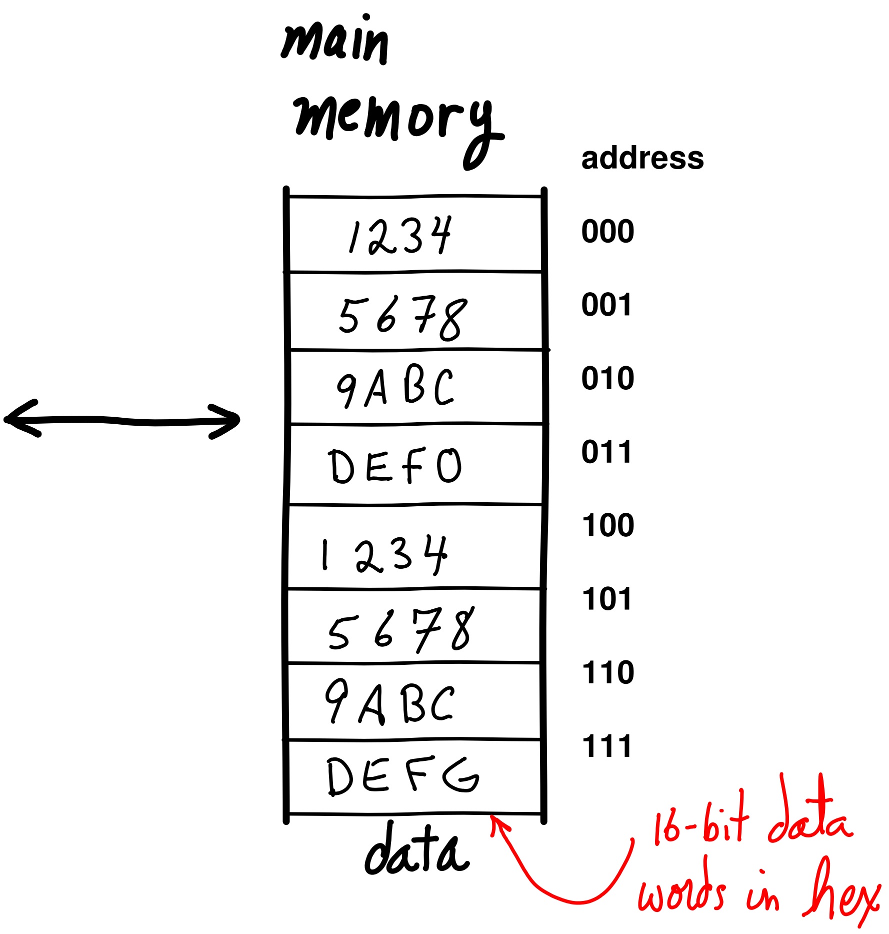
Systems are typically von Neumann.

Processors can be either von Neumann or Harvard (Intel x86, Pentium, ...).

Enhancements (?)
--- more processor hardware, built-in functionality (caches, branch prediction, ...)
--- multiple processors
--- simpler, low-power
--- wider (more bits per register)
--- wider (copied functional units)
--- wider (multiple, different functional units)

what's a cache?





A cache is a small memory.

```
Cache Memory Read Operation

    Get address input.

    Search all cache cells {
        if address == cell's tag,
            send cell's data to output
        else
            try next cell
    }
    if Search failed {
        cache "miss";
        get data from Main Memory;
        write tag+data into cell;
        send data to output.
    }
```

Writes are similar, but data goes the other way.

Data is transfered between cache and main memory as needed.

Cache is fast, main memory is slow.

Cache ABSTRACTION provides illusion that,

---- Access is just memory access:
address goes in, data comes out.

---- Two Memories, IMEM and DMEM:
separate, independent accesses

---- Single, unified, von Neumann memory:
one memory, one address space.

Advantages

---- Memory access can appear to be very fast, when in fact it is very slow: If data is reused, cache can respond immediately, without waiting for slow main memory response.

---- Two caches allow two memory accesses in parallel (simultaneously).

Cache Operation Complications
Handling the new problems created:

---- tag+data is not in cache?
Must **stall the processor** when cache misses?
How do we stall a processor?

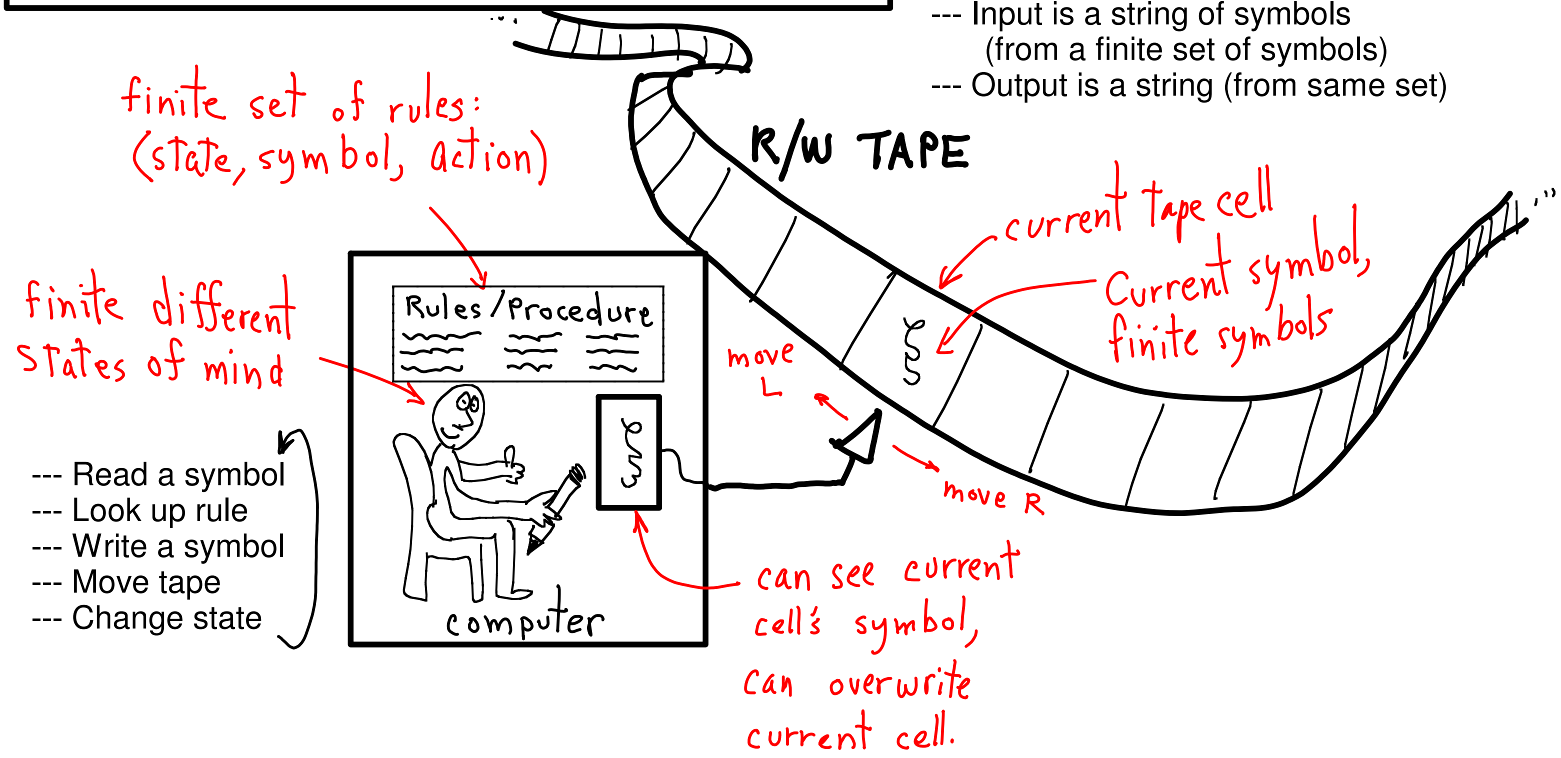
---- write operation?
Will cache and main memory data differ?
Should both be written at same time?
Should we wait and write main memory later?

--- overlapping accesses from both caches?
IMEM reads instruction that DMEM is writing?
Should IMEM wait until DMEM finishes?
How would IMEM know DMEM is writing?

Turing Machine Concept

Question: What can a person (computer) do, given a set of instructions to follow.

--- Works for any person (unambiguos)
--- Input is a string of symbols
(from a finite set of symbols)
--- Output is a string (from same set)



--- Read a symbol
--- Look up rule
--- Write a symbol
--- Move tape
--- Change state

A Few Details

--- Starts in a particular state
--- Stops in a "Halting" state, or not at all
--- Can go forever
--- Can always get more tape:
--- more input (or maybe finite input)
--- more output

Clearly, goes beyond human capabilities, but good as a fundamental abstraction.

What Is A State (of mind)?

Version-1:
--- I know I am doing addition
--- I know I am adding the 5th column
--- I know I have seen the number 5 in the top row
--- I know I have seen the number 2 in the bottom row

Version-2:
--- Physical state is momentary value of all measurables
--- State change is affected by interactions w/ environment
--- Instantaneous environmental impact is current symbol
--- Rule-based state change
--- Instantaneous effect on environment is output symbol

An example: add

```
  1 2
+ 3 4
-----
    6
```

- start up-right
- see "2"
- write: nothing
- move down
- next state: "seen 2"
- see "4"
- write: nothing
- move down
- next state: "seen 2 & 4"
- see "␣" (blank)
- write: "6"
- move up-left

...

Universal TM

A Turing Machine
that
Simulates other Turing Machines

Every computation can be modeled as some Turing Machine.

Doing computation X means building and running TM-x.

<span style="color:red">**Big idea**</span>: **don't build new hardware,**

Build one simulator

For every other (new/special) machine, describe and simulate.

Build one simulator, and many descriptions.

--- **describing == programming**
--- **simulating == executing**

Language for describing TMs?

--- The rule table describes a TM. Simple!
--- Or, devise a programming language. More productive.
--- Is the language Turing complete (can describe any TM)?

UTM

Simultation-step-1:
Find M's R/W location, read input symbol, **A**

Simulation-step-2:
Find M's state, **S**

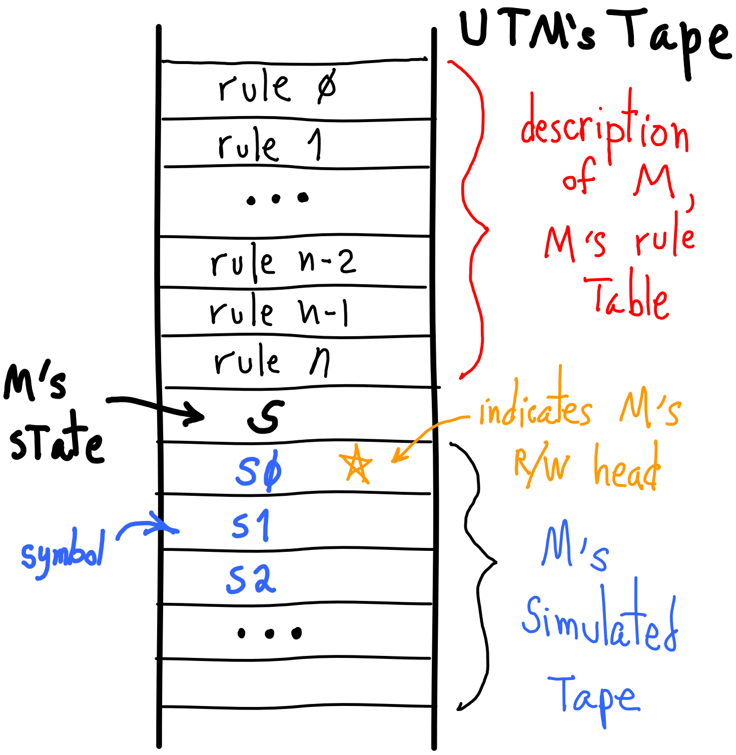
Simulation-step-3:
Find Rule Table

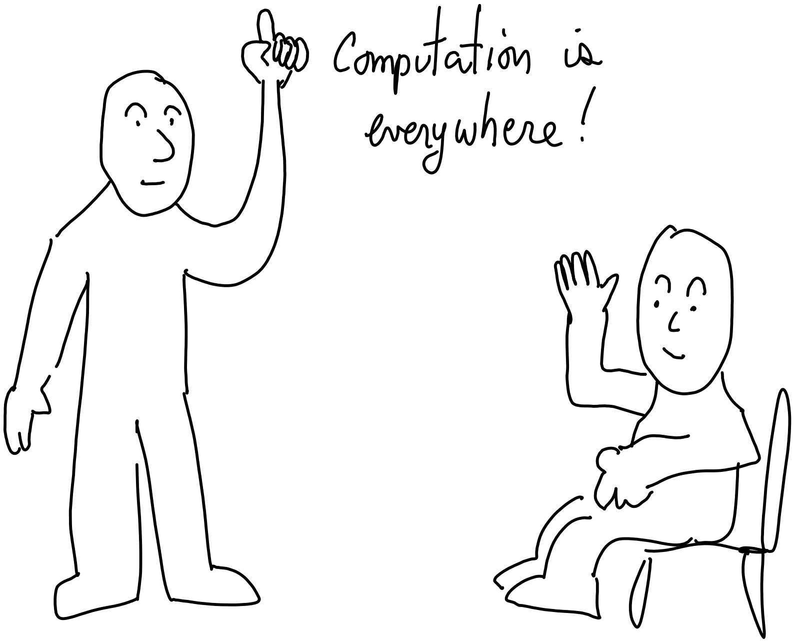
Simulation-step-4:
Search for rule for State **S**
Check if input == **A**
If **S** and **A** do not match Rule, find next Rule

Simulation-step-5:
Find Rule's output symbol, **B**
Find R/W head's cell
Write **B**

Simulation-step-6:
Find Rule's move **G** = (L or R)
Find R/W head's cell
Write R/W head location mark to L or R cell

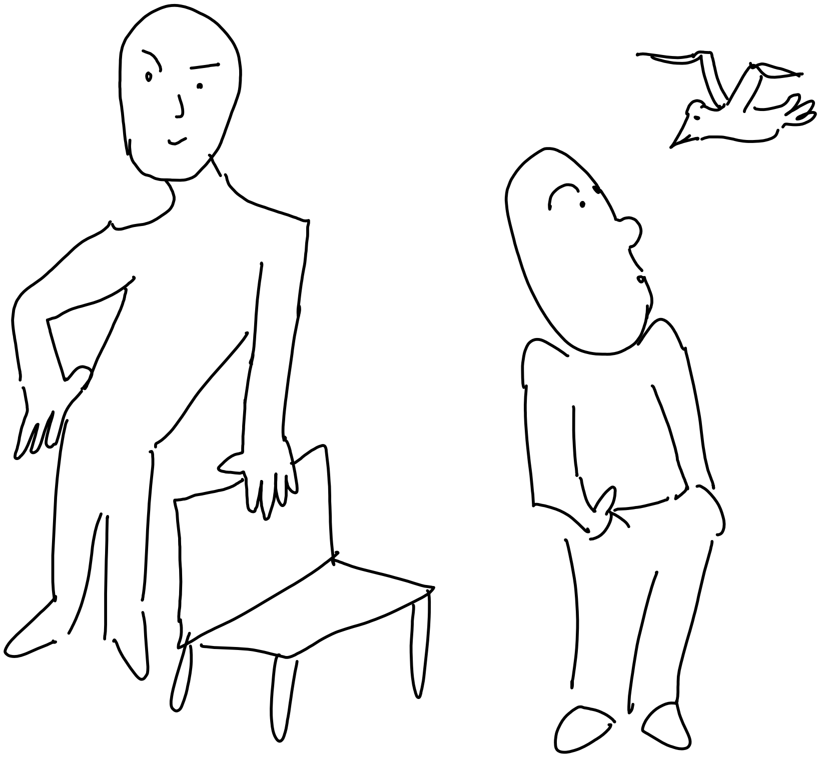
Simulation-step-7:
Find Rule's next-State, **N**
Find M's current-state cell
Write **N**

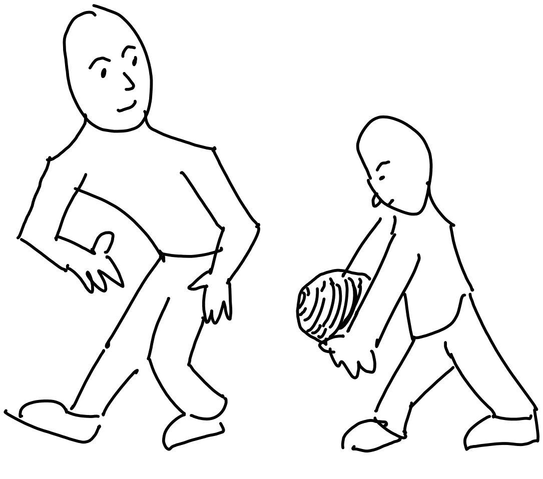




Eham: Computation is everywhere.
Drah: Where?
E: Everywhere!
D: A car crash?
E: Yes.
D: A doll house?
E: Yes.
D: Me?
E: Yes.
D: What is the same about them?
E: They all change.
D: So, computation is change?
E: Yes.
D: Everything changes, so computation is everywhere?
E: Yes.
D: What is computation?
E: Change.

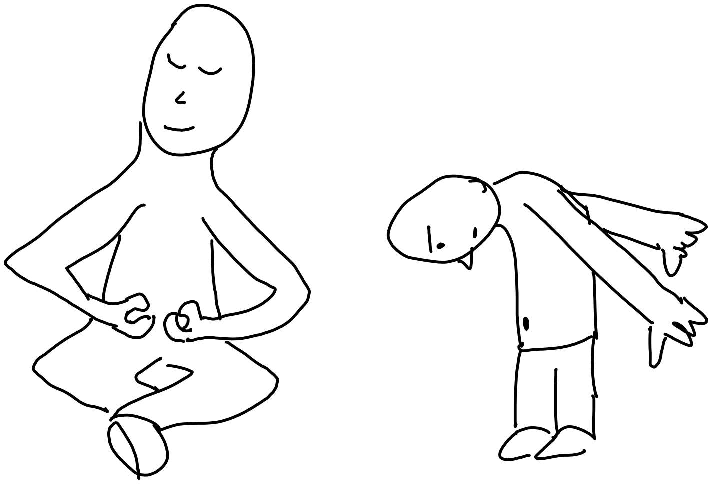
D: So, everything changes, and because everything changes, everything is computation, and computation is change.
E: Yes!
D: Oh.
E: You see, it is really quite simple.
D: How simple?
E: There is a model.
D: A model?
E: Yes.
D: How is there a model?
E: Things are one way, then they are another.
D: And that means there is a model?
E: Exactly.
D: How do I know there is a model?
E: That is an existence proof.
D: What is?
E: I just said there is a model, didn't I?



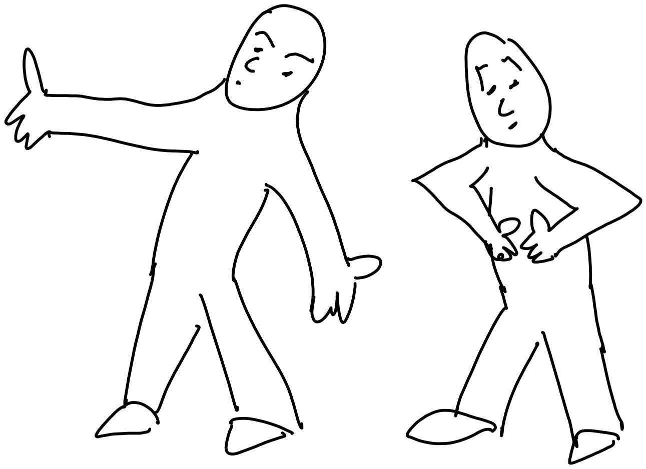


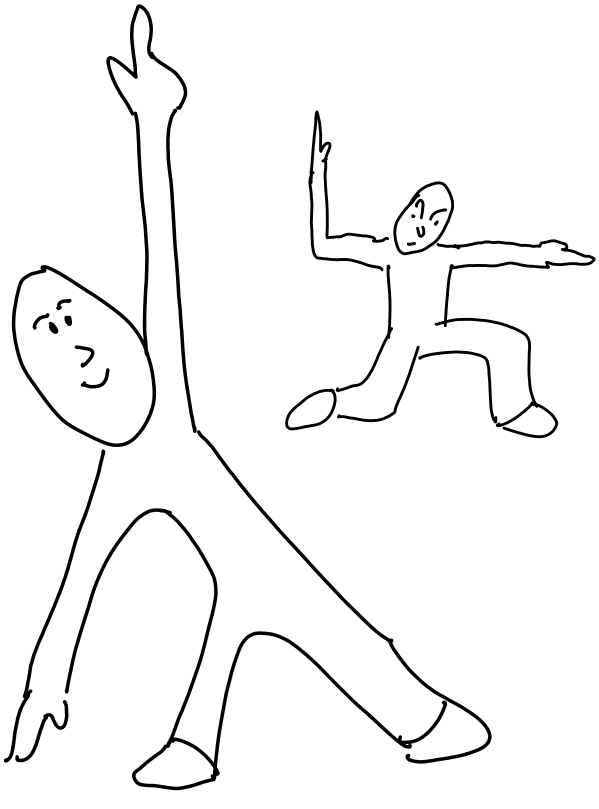
D: And a model means things are one way, then another.
E: Now you've got it.
D: Isn't that the same as change?
E: Quite right.
D: So, a model is change and change is computation and change is computation because there is a model?
E: See, now you're getting the hang of it.
D: Oh.

D: So, what is a computer?
E: Something that does computation.
D: Doing computation?
E: That's it, computing.
D: So, computers compute?
E: Obviously.
D: And computing is change?
E: What else could it be?
D: Everything changes, so everything is a computer?
E: Yes, absolutely.



D: I am a computer?
E: Without a doubt. When you change, which you do constantly, you are computation.
D: Then, I'm not me before, nor me after, but I'm me as I change?
E: Computation is everything and everywhere, all things are changing, you are changing, you are computation.
D: What if I don't change?
E: Everything changes.
D: So, there is nothing that doesn't change?
E: That's right, nothing doesn't change.
D: So nothing isn't computation. Does nothing exist?
E: Of course nothing exists. There is zero, zero exists.





D: So 0 is not computation?
E: That's right, because 0 is nothing. If it were something, then it would be computation, because all things change.
D: So, does 1 exist.
E: As surely as anything exists, as certainly as zero exists.
D: But they don't change, 0 and 1, I mean?
E: Of course not.
D: Then something exists which is not computation?
E: Absolutely.
D: But, if computation is everywhere, where are 0 and 1?
E: Right there.
D: Where? On the ceiling?
E: Of course. See that thing there? There is only 1 of them there.
D: So that's the existence of 1?
E: What could be clearer?