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

Von Neumann

versus

data Path

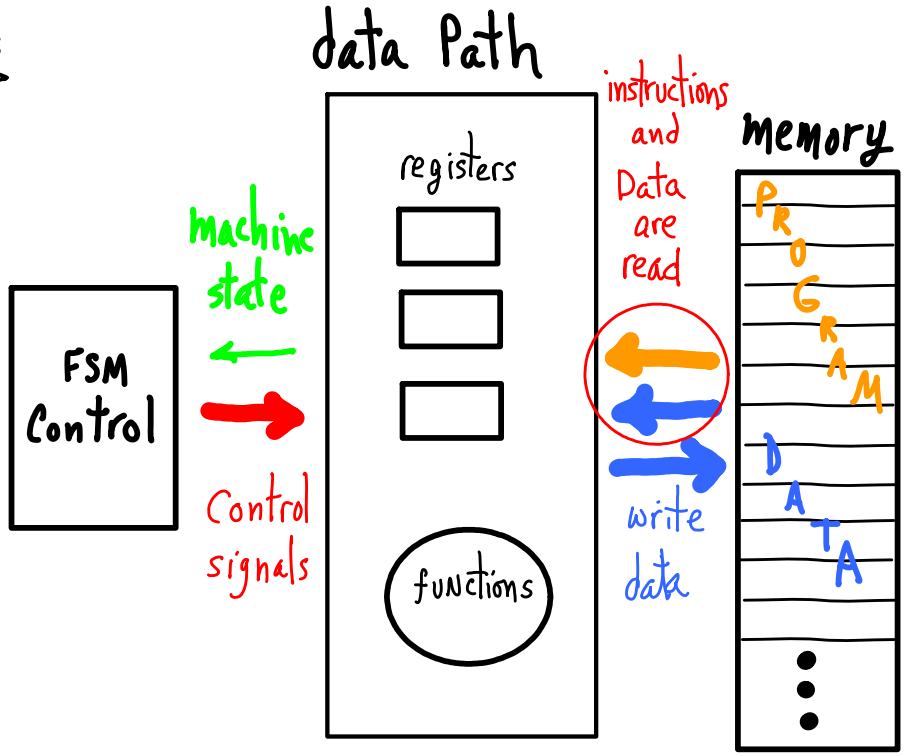
Machine Description (Turing),
a table of rules

State	input	output	move	Next
A	S1	S5	L	B
	S2			
	S3			
	...			
	Sn			
B	S1			
	S2			
	...			

a "rule"

Σ = alphabet of symbols

32-bit symbols → 4G symbol set



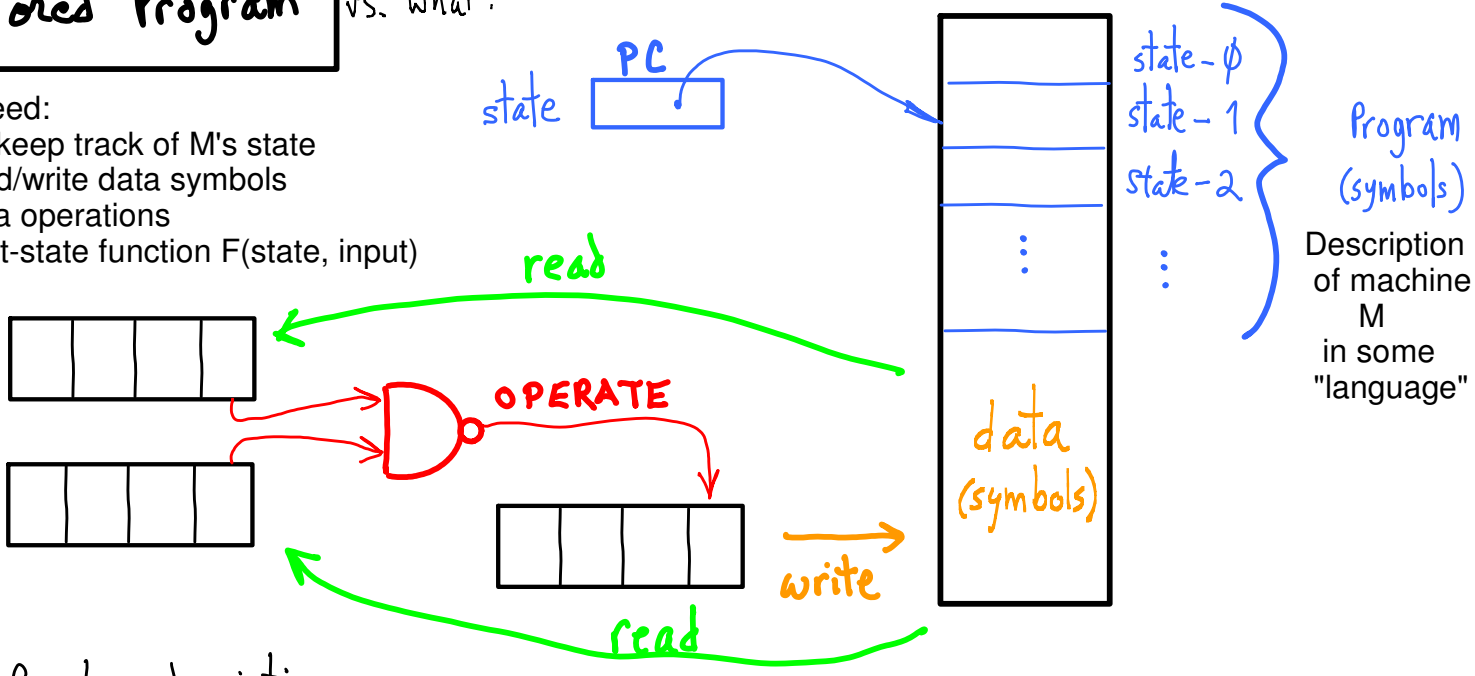
(input = s1 / output = s5 / move = L)

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)



function descriptions

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
---- how to change to next state of M

then we can describe ANY machine, and simulate it.

And, what else?

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

Memory can be thought of as an array, the address is the array index:

Memory[address **]** <== in
out <== **Memory[** address **]**

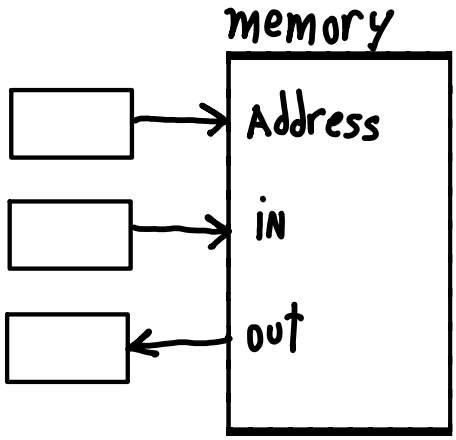
convenience/ performance

ADD SUB
MULTIPY DIVIDE
...

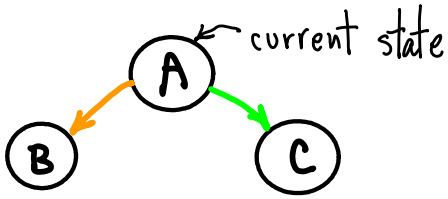
read /write data, stack operations

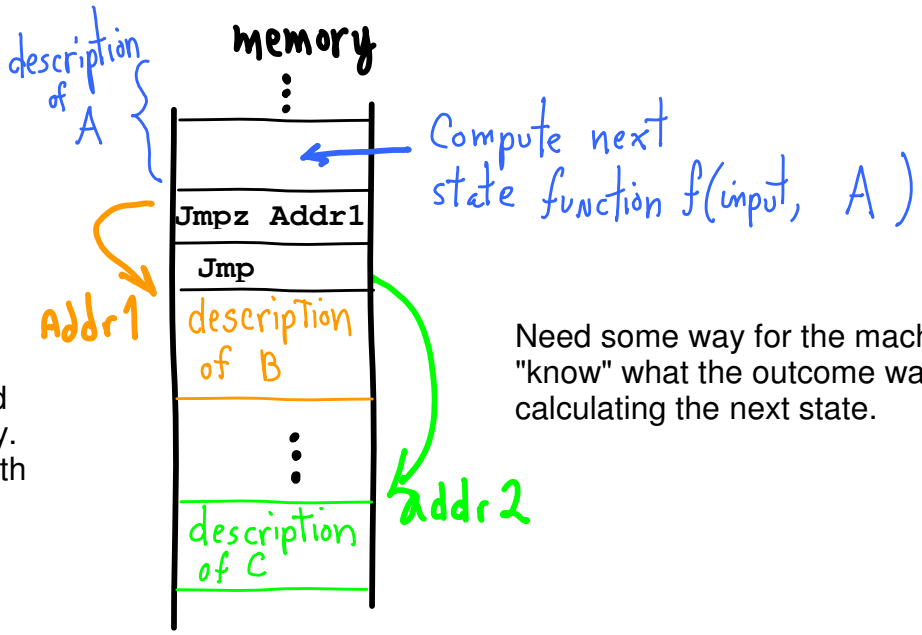
Load Reg, Address
Store Reg, Address

Push Reg
Pop Reg



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





Recreate the branched graph in linear memory. Execution follows a path through graph.

Need some way for the machine to "know" what the outcome was of calculating the next state.

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

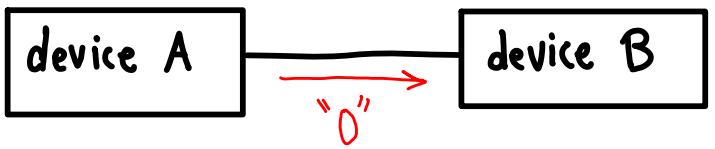
--- instructions are read from "memory"
--- the instruction is read into a "register"
--- a controller "executes" the instruction
--- data is read from "memory"
--- a "register" remembers what was read
--- an operation changes the data
--- a register stores the result
--- the result is written to "memory"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

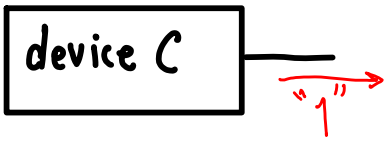
How they correspond:

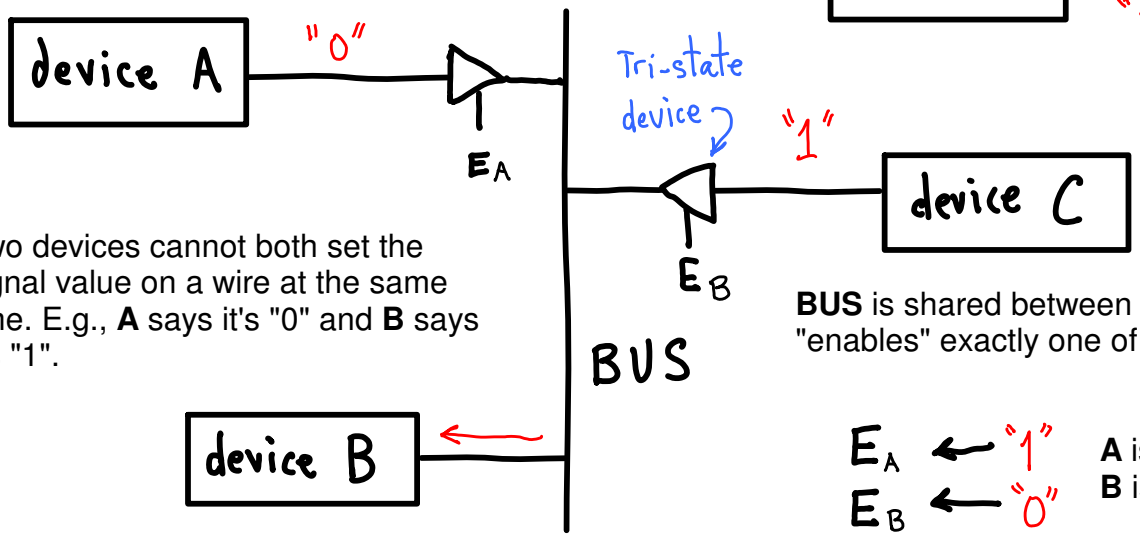
Turing	**von Neumann**
a State	A section of the program
a change of state	jumping to a different section
a rule's output part	a section that produces a new output
a rule's state-change part	a section that calculates the next jump

Busses (because you see them here and there)

We usually think of a signal as going from one place to another along a wire. E.g., device **A** sends a "0" to device **B**.
Device **C** cannot use the same wire to send to **B**, even at a different time. But, maybe we can?







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

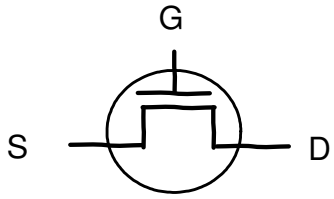
BUS is shared between devices. The controller "enables" exactly one of the tri-states at a time; e.g.,

$E_A \leftarrow$ "1"
$E_B \leftarrow$ "0"

A is connected to **BUS**
B is disconnected from **BUS**

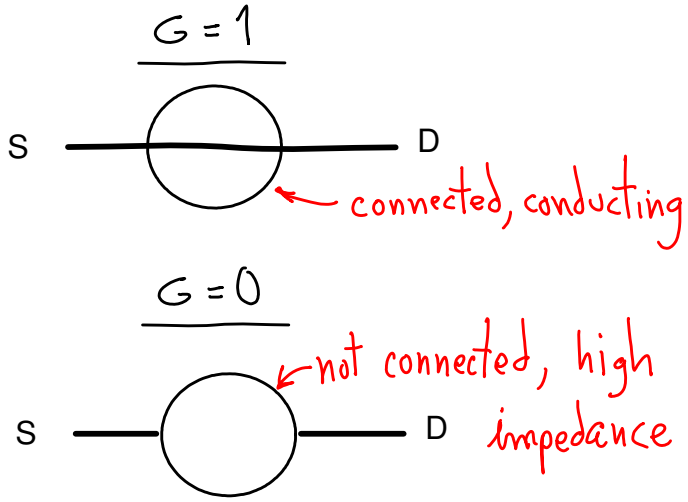
A "tri-state buffer"

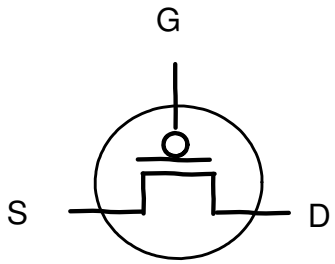
CMOS



n-transistor
– switch

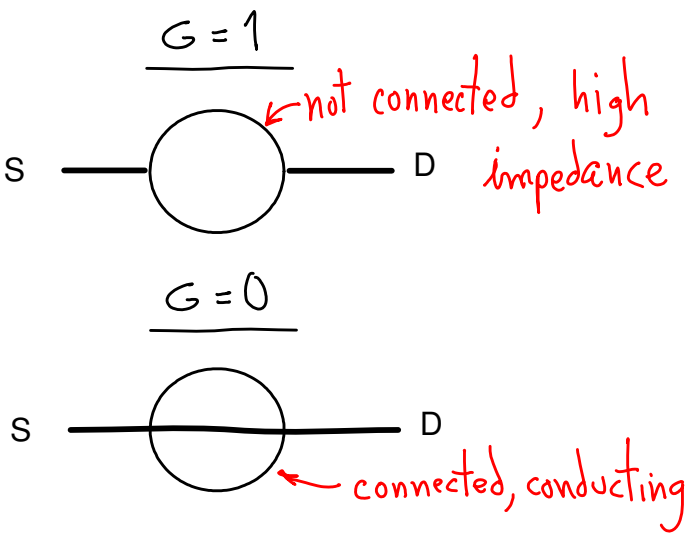
Passes a "0" cleanly

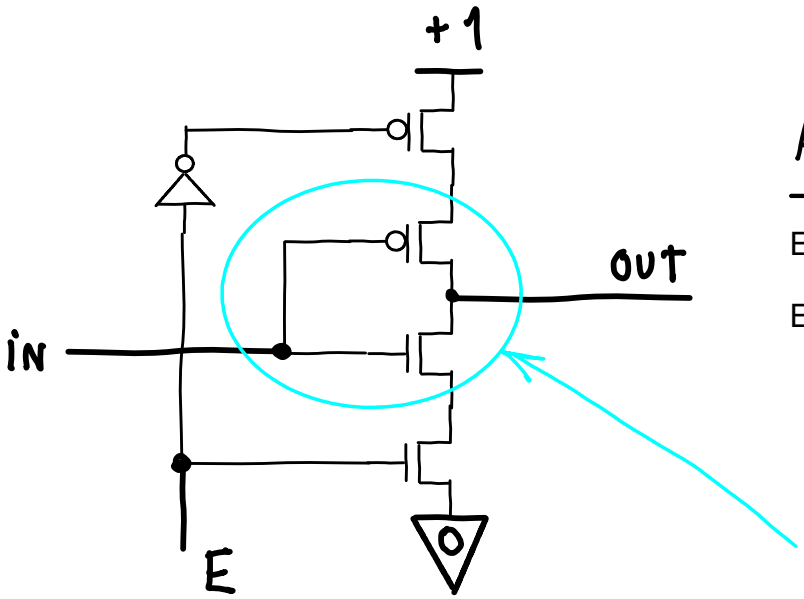




p-transistor
– switch

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

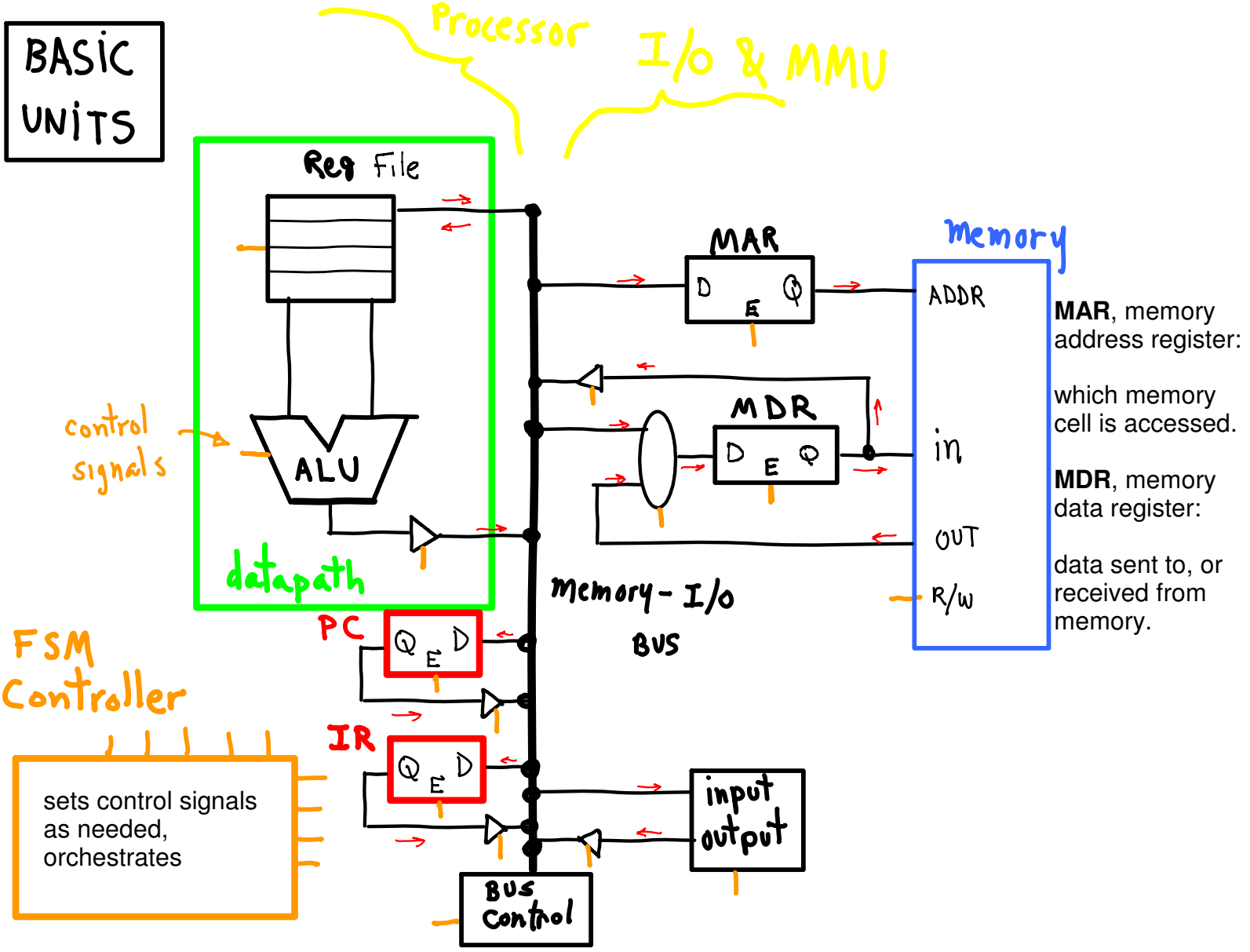
an inverter

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



MAR, memory address register:

which memory cell is accessed.

MDR, memory data register:

data sent to, or received from memory.

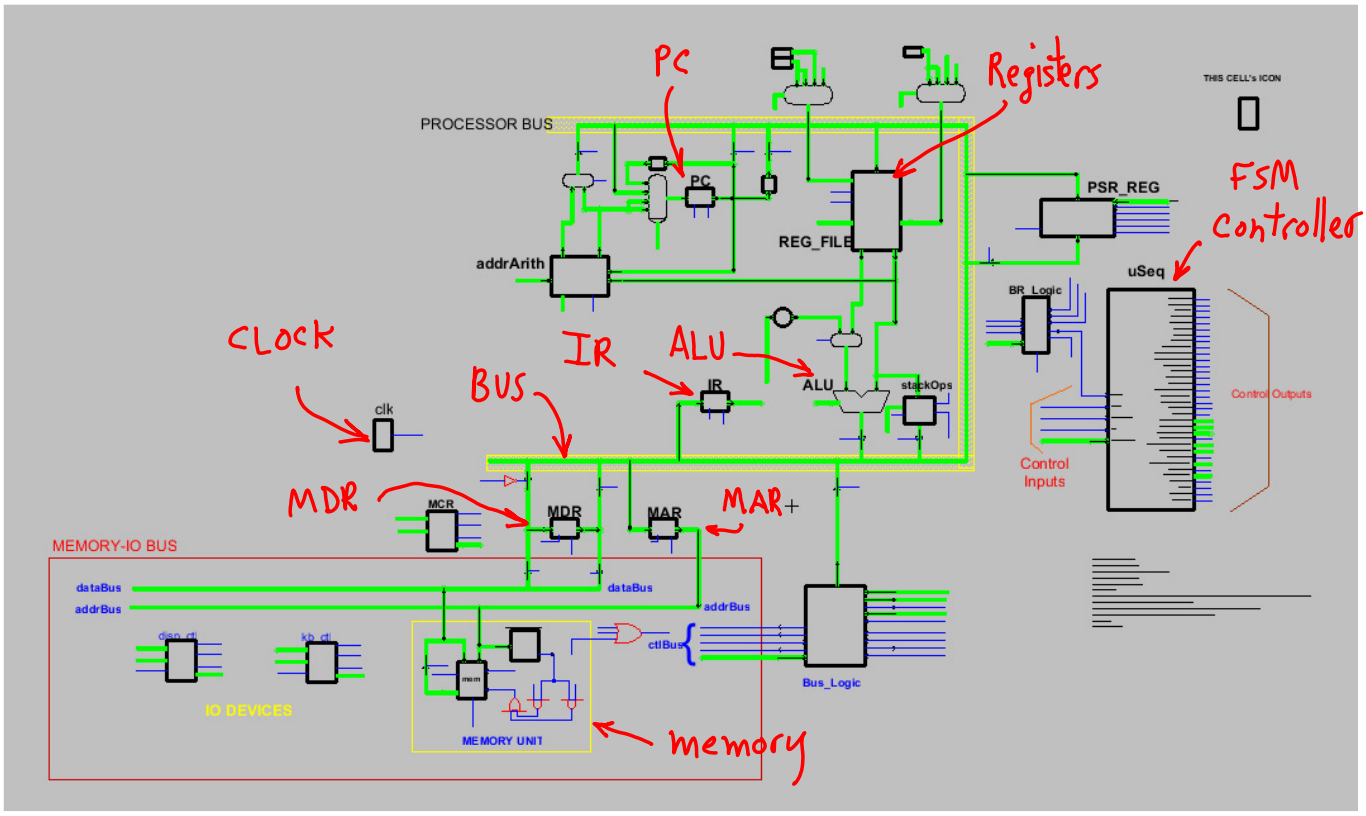
Machine Cycle

execution phases

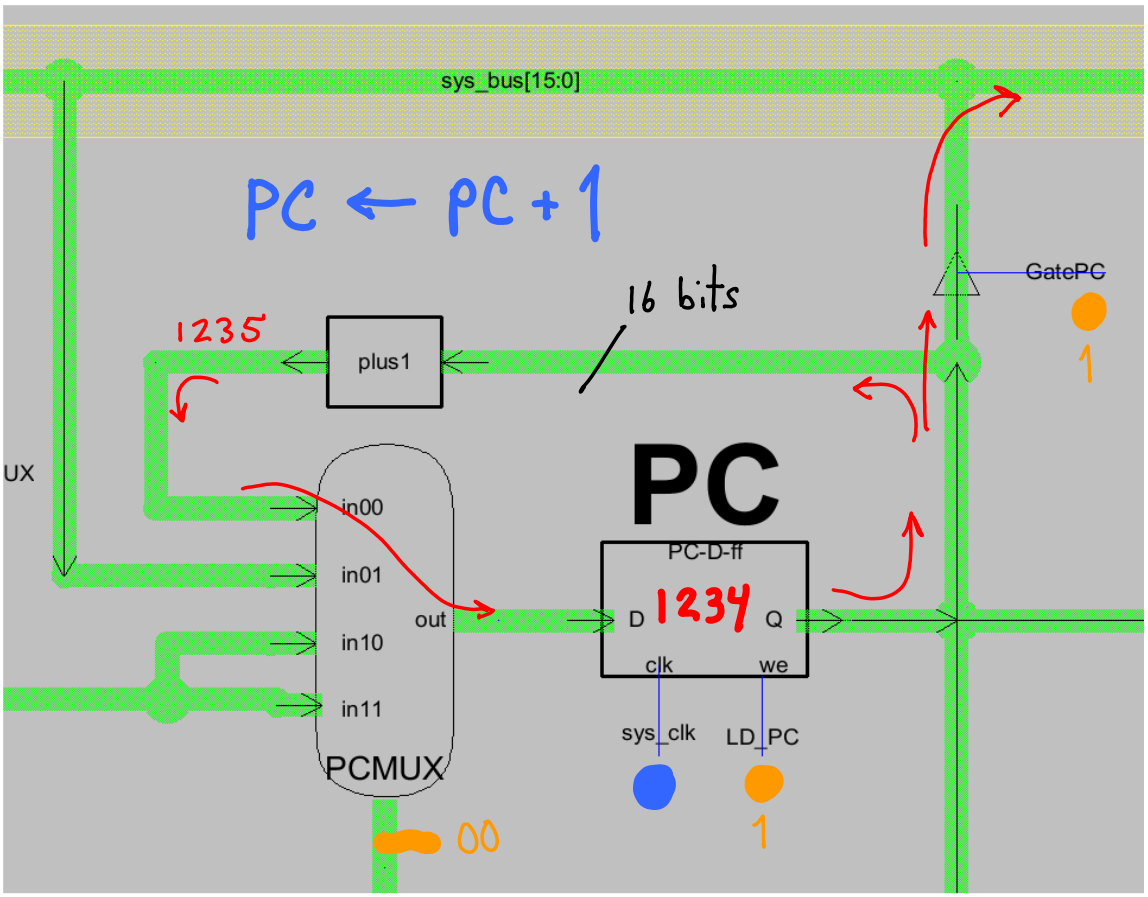
1. **Fetch** instruction
2. **Decode** instruction
3. **Evaluate** address
4. **Fetch** operands
5. **Execute** instruction
6. **Store** result

forever, or?

LC3, a von Neumann architecture



Instruction Execution

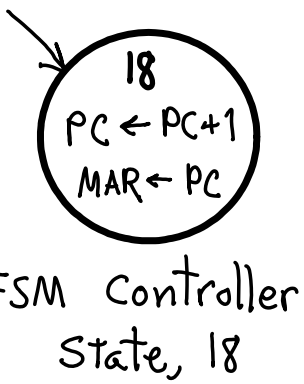


PHASE:
Fetch instruction

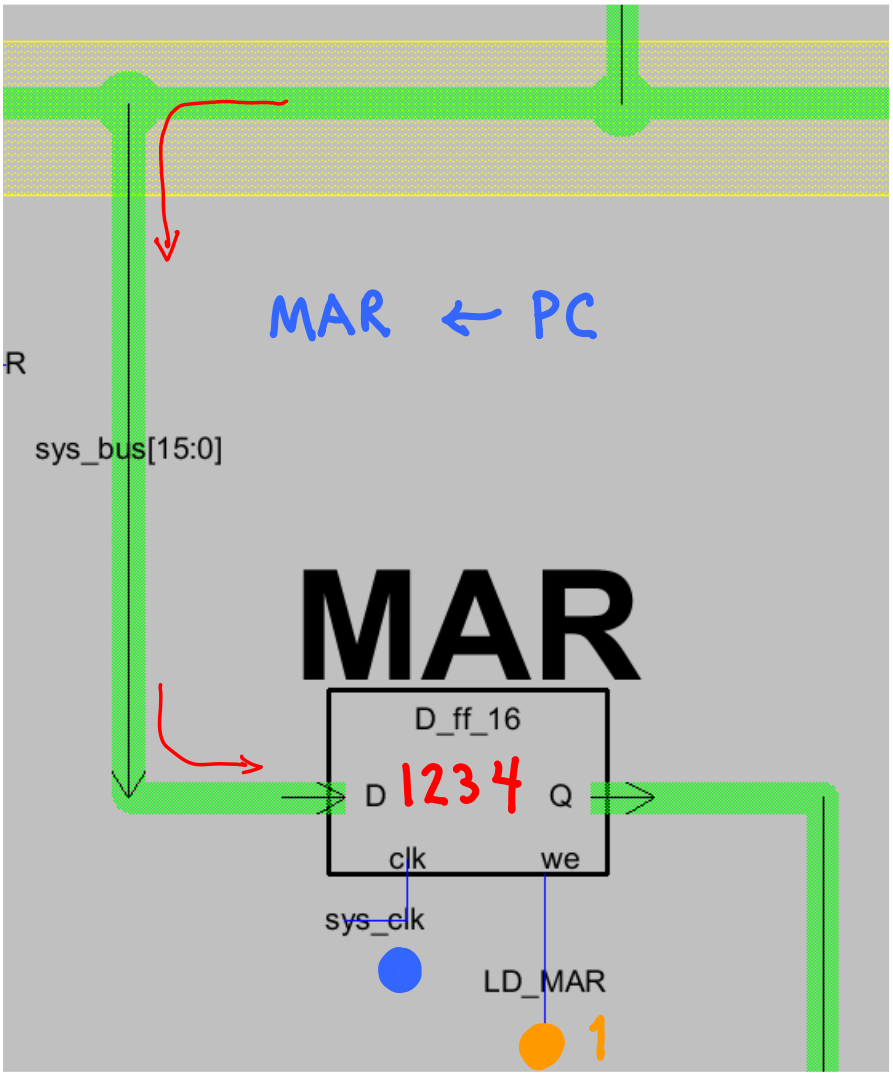
-- 1st step, state 18

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

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



FSM Controller State, 18

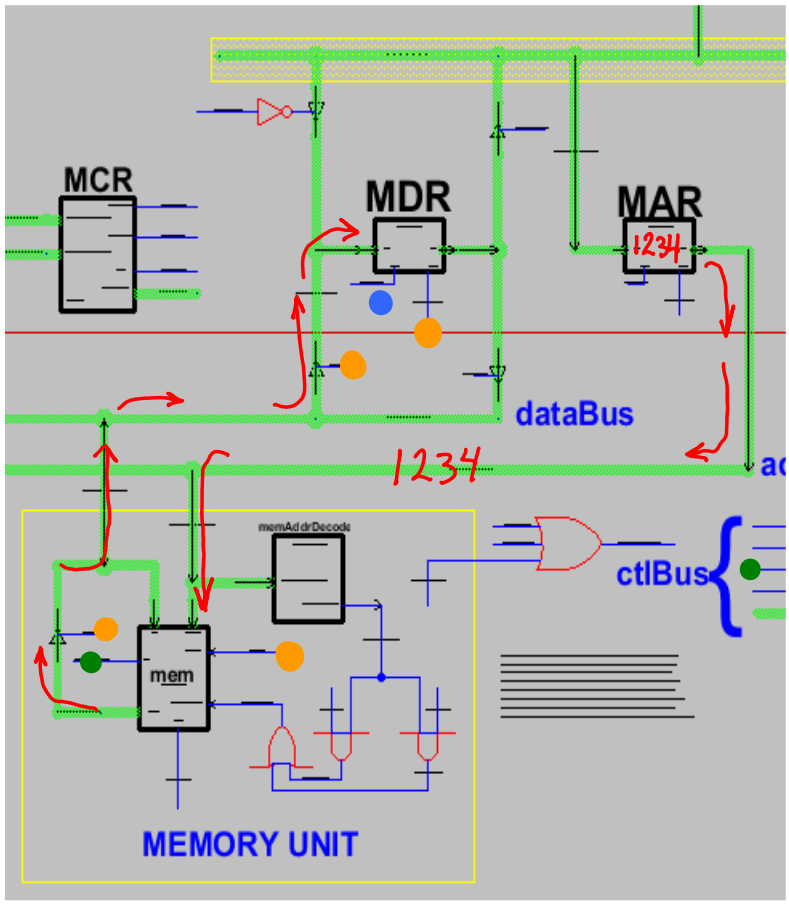


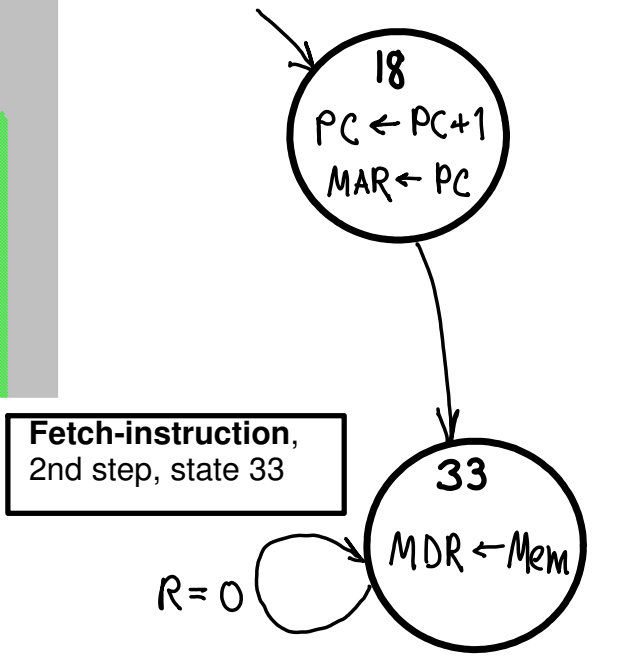
(continue)
Fetch instruction, first step:

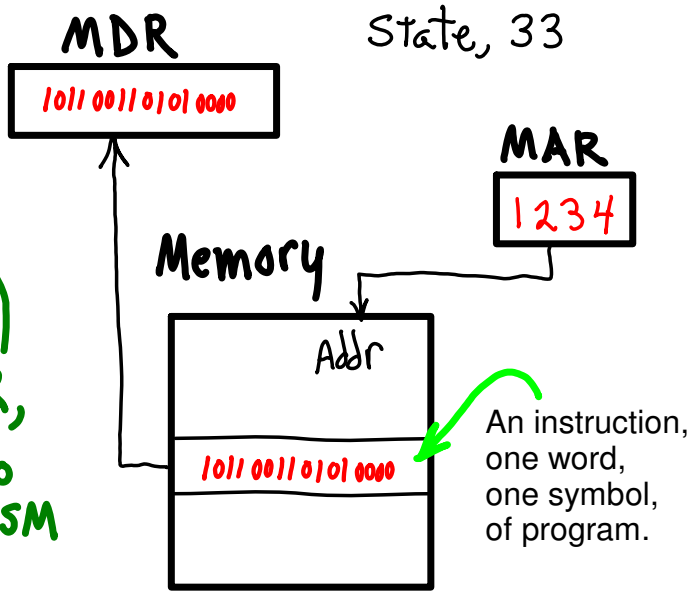
Load MAR w/ address in PC

FSM Controller must set signal values for State-18:

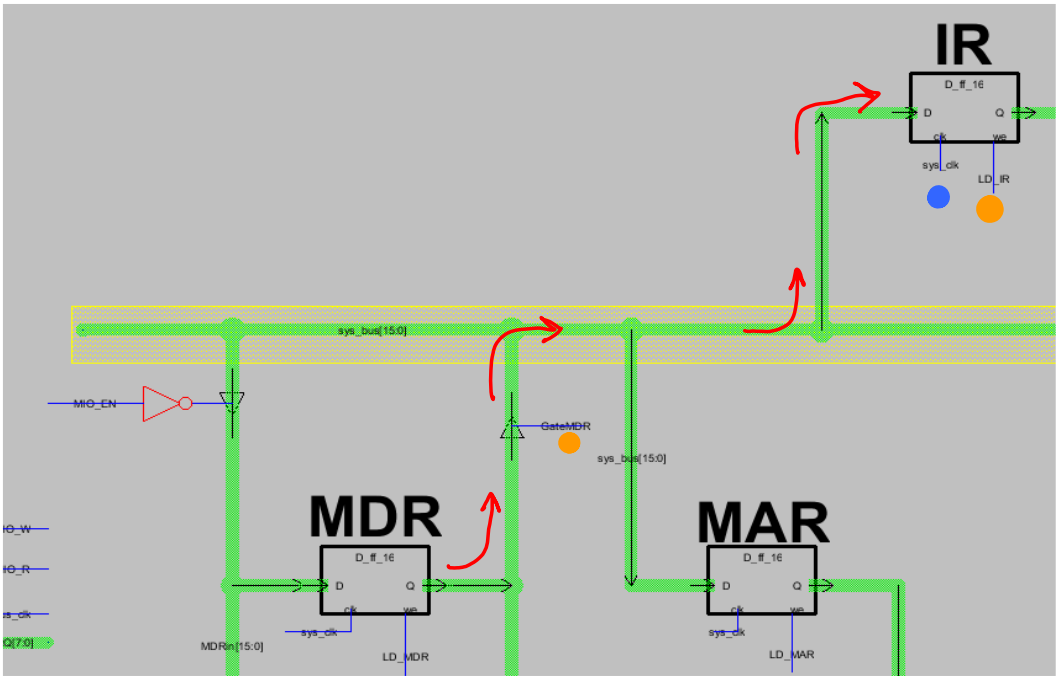
GatePC <== 1'b1
PCMUX <== 2'b00
LD_PC <== 1'b1
LD_MAR <== 1'b1

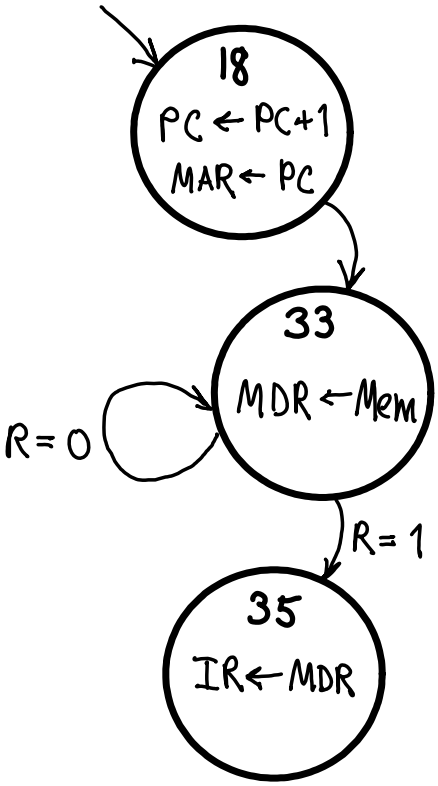






Continue to next state when R ← 1





Fetch-instruction,
3rd step

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

Control signals for state-35:

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

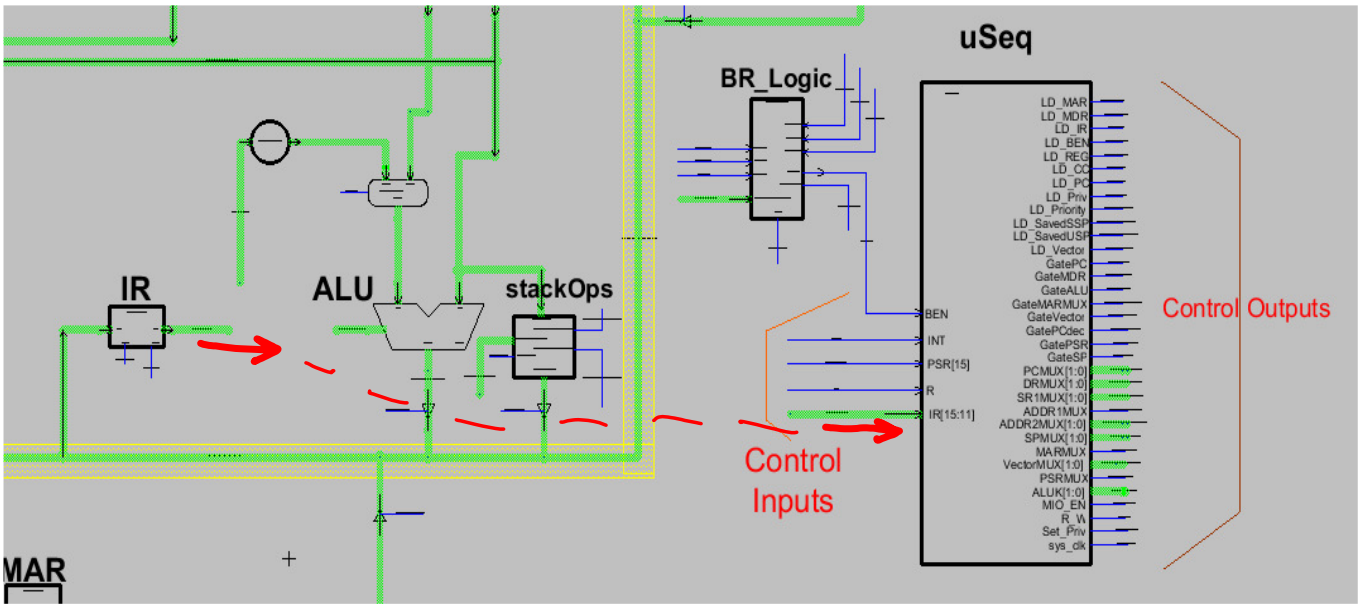
This completes the **Fetch-instruction** phase.

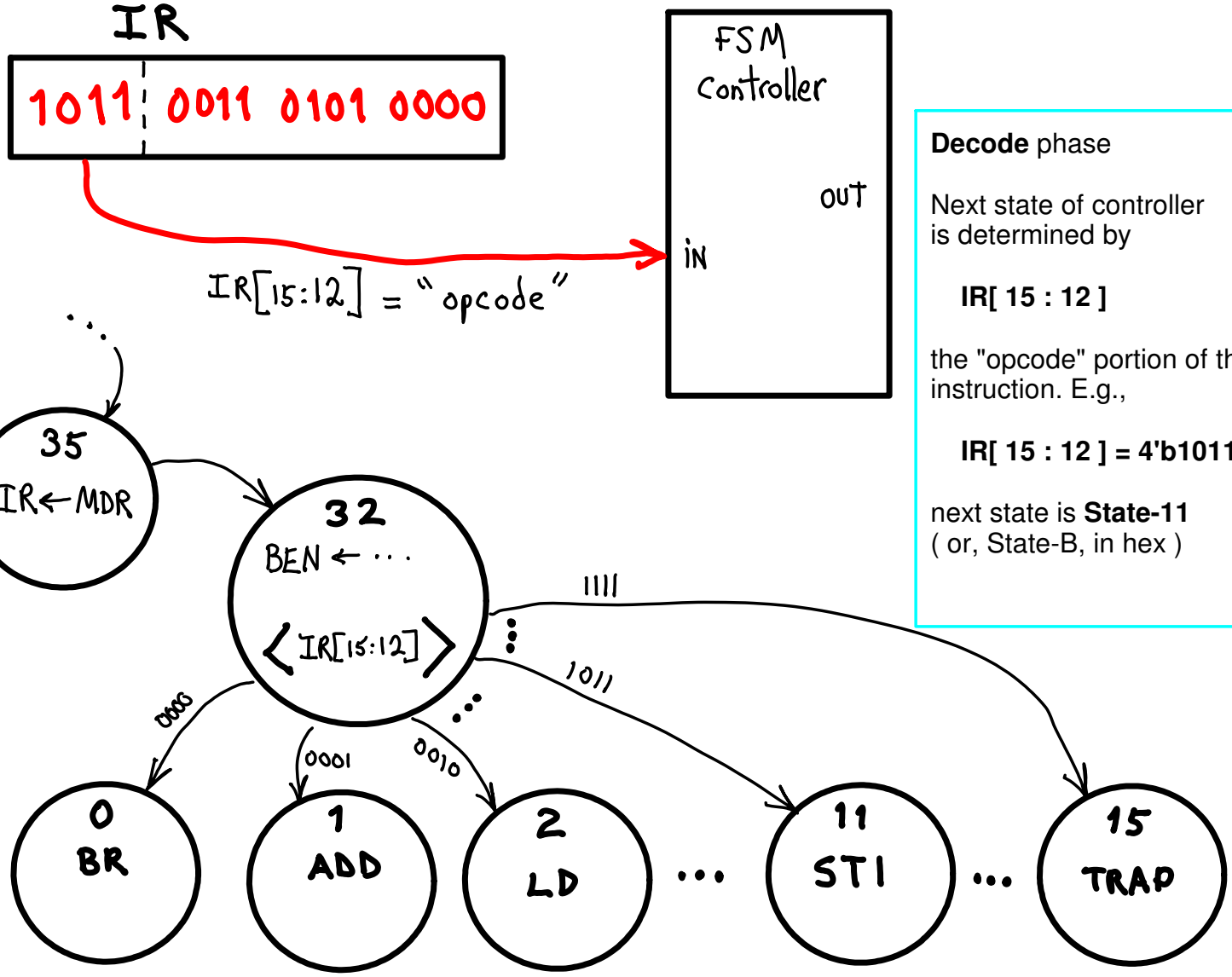
Overall effects:

IR <== 16'b 1011 0011 0101 0000 (an instruction, content of memory word)
(**IR** <== 16'h B350; or **IR** <== Mem[16'h 1234])

PC <== 16'b 0001 0010 0011 0100 (a 16-bit memory address)
(**PC** <== 16'h 1235)

Decode phase, what instruction is this?





Decode phase

Next state of controller is determined by

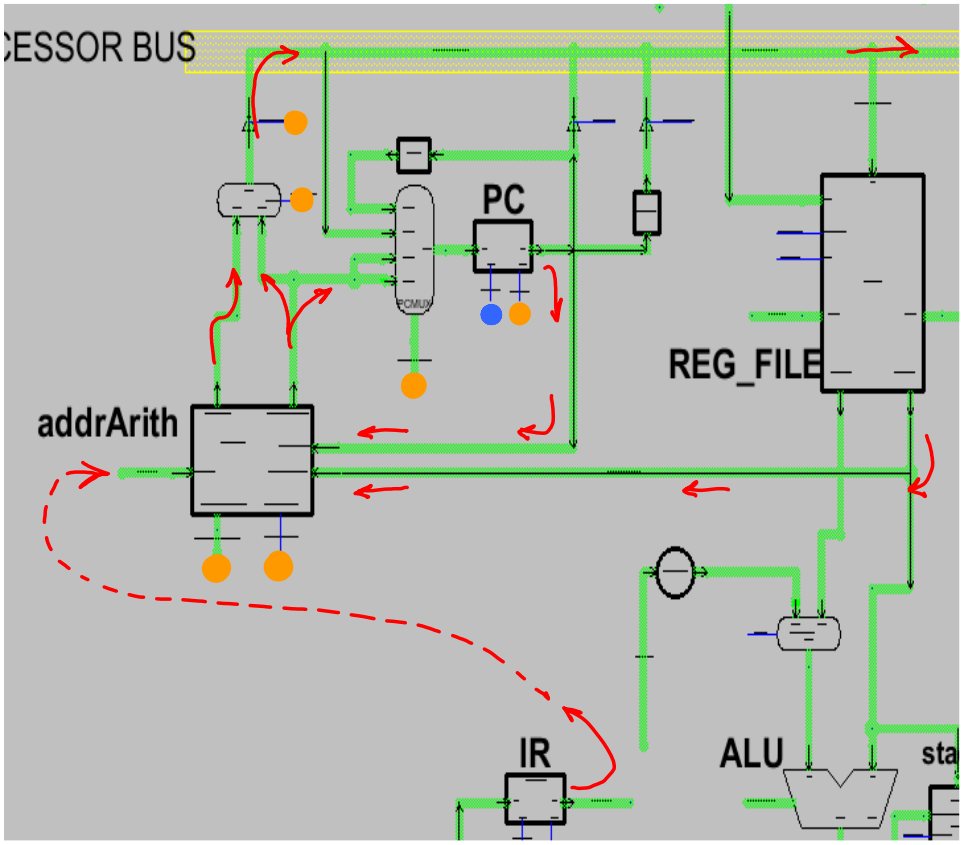
IR[15 : 12]

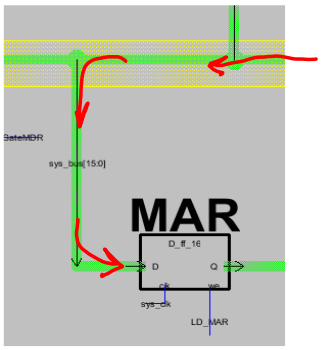
the "opcode" portion of the instruction. E.g.,

IR[15 : 12] = 4'b1011

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

Evaluate Address Phase





Evaluate Address phase

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

--- any of 8 registers in **RegFile**
--- **PC**
--- **IR**

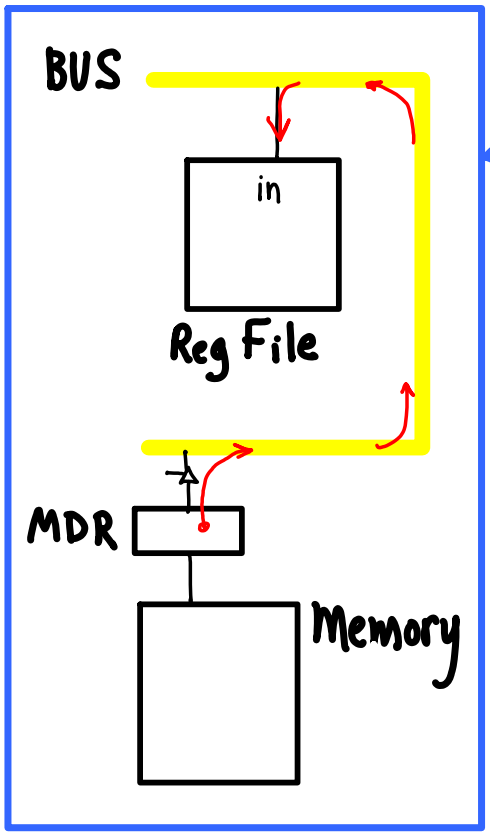
Destination of calculated address is,

--- (1) **MAR**, to transfer data between memory and register: LDR, STR

or

--- (2) **PC**, to jump to next instruction: JMP, BR, INT, TRAP

Fetch Operands Phase



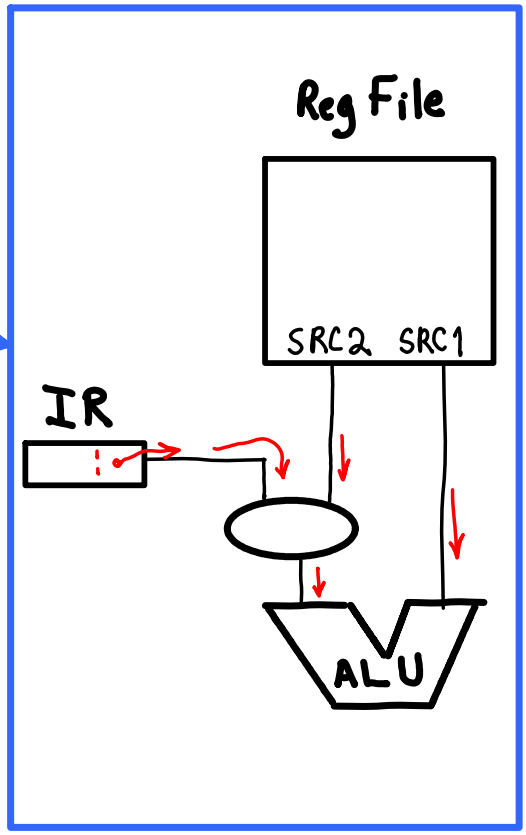
Fetching Operands could mean,

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

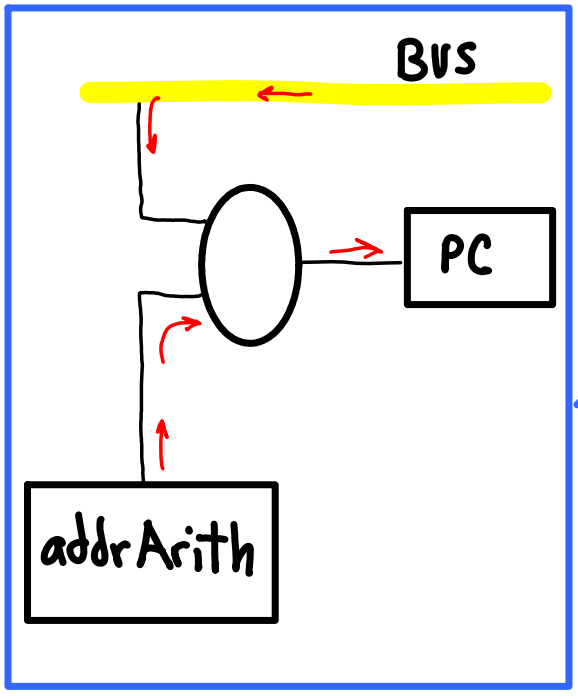
or

(2.) make data available to **ALU** (e.g., ADD) possible from some of the **IR**'s bits, but usually from a **register**.

LC3's "load from memory" instructions (LD, LDR, LDI) do nothing after copying from the **MDR** to a **register**; ie., they have no further execution phases.



Execute Phase



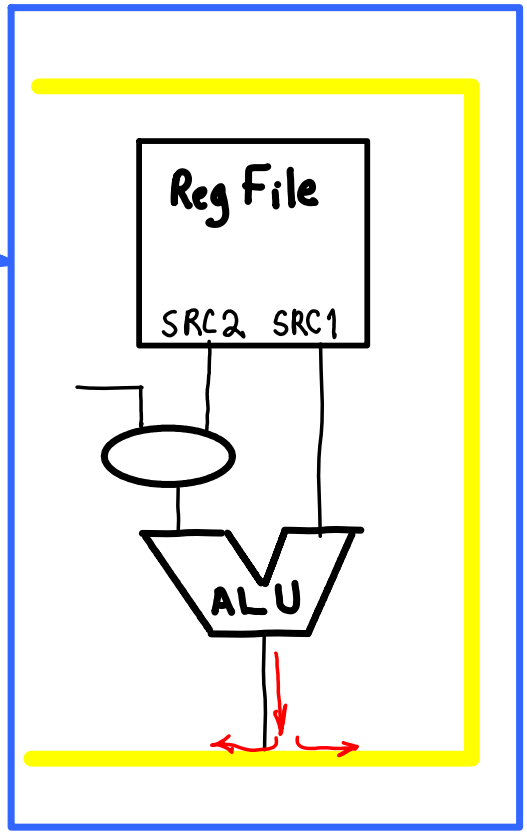
Execution could mean,

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

or

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

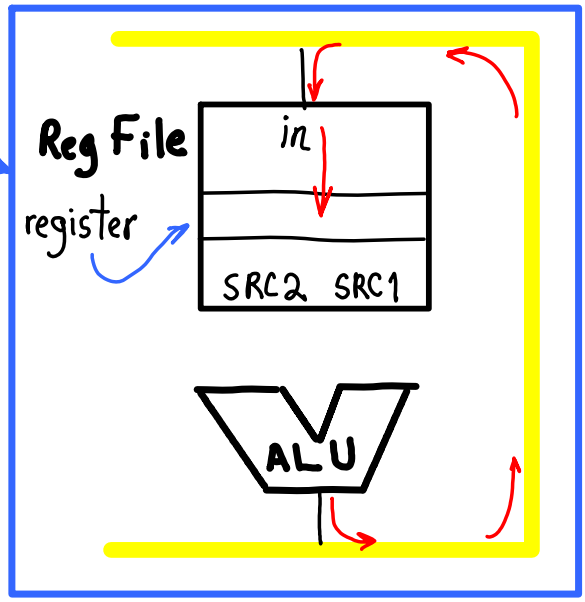
Instructions that do (2.) are **JMP**, **BR**, **TRAP**, as well as system generated action from hardware, **interrupts** and **exceptions**.



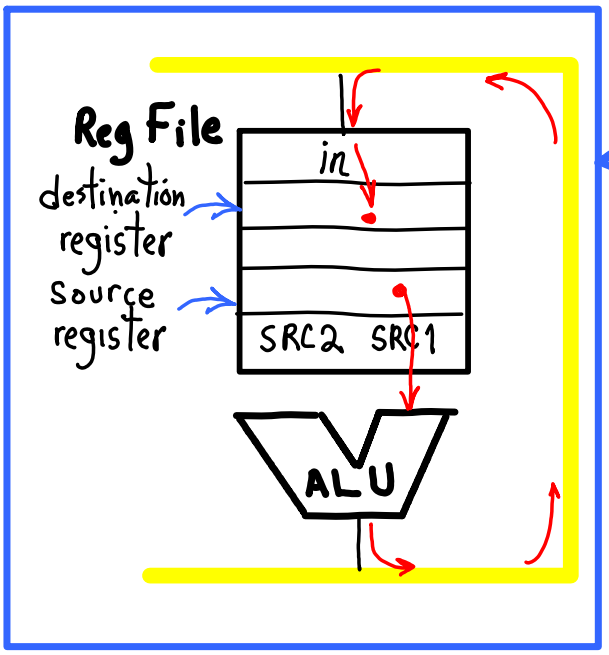
Store Phase

Store could mean,

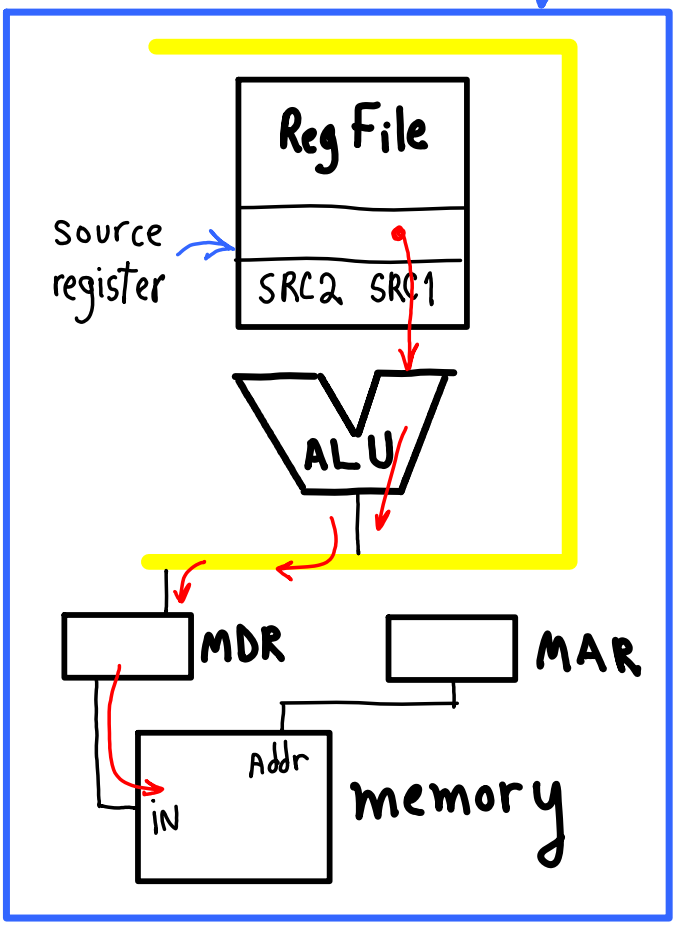
(1.) write **ALU** result to a **register**.



(2.) copy from a **source register** to **destination register**.



(3.) copy from a **register** into a **memory location**.



Execution Phases

Machine Cycle
==========
Fetch instruction
Decode instruction
Evaluate address
Fetch operands
Execute instruction
Store result

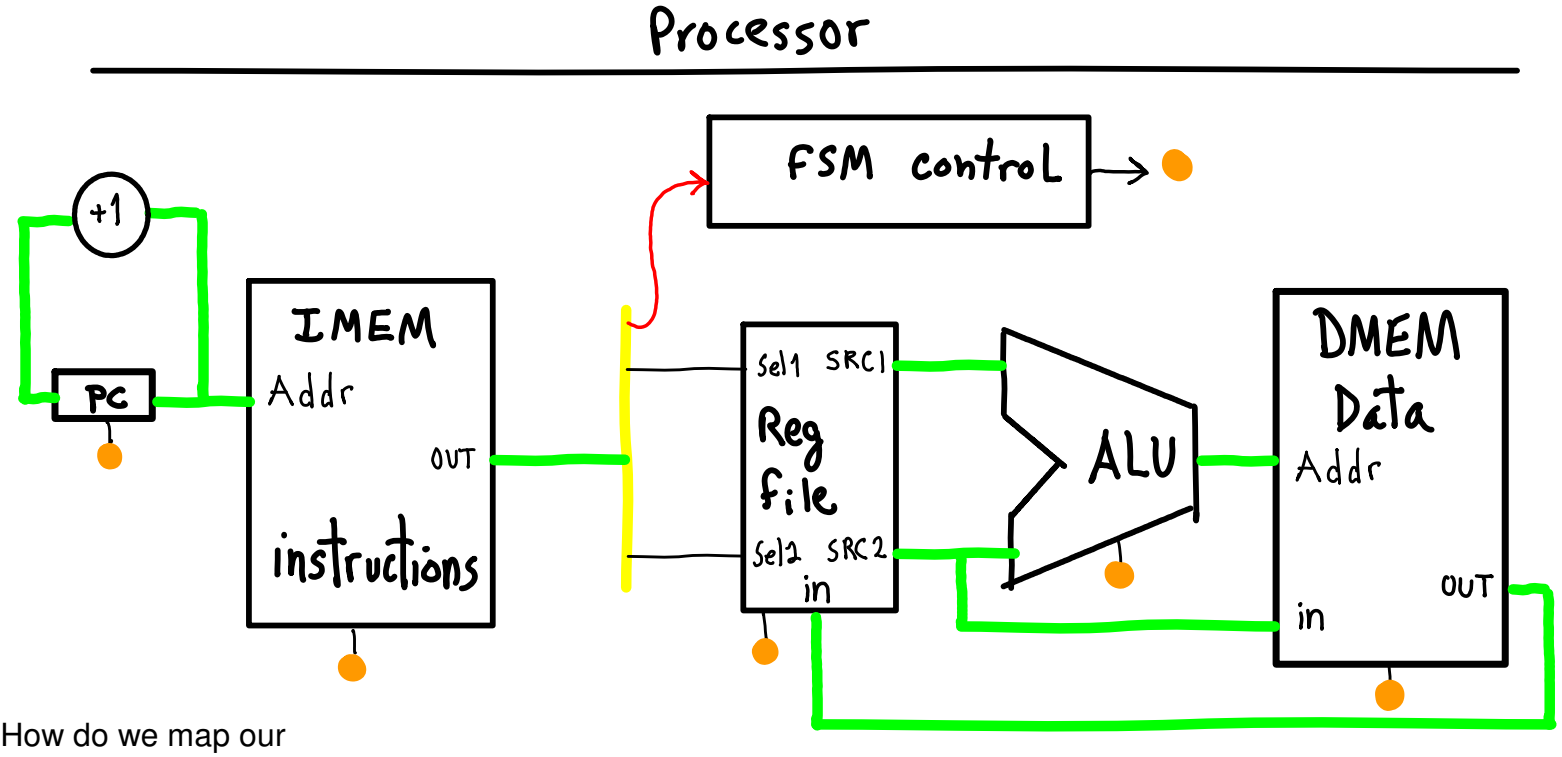
--- No instruction executes every phase.

--- 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)

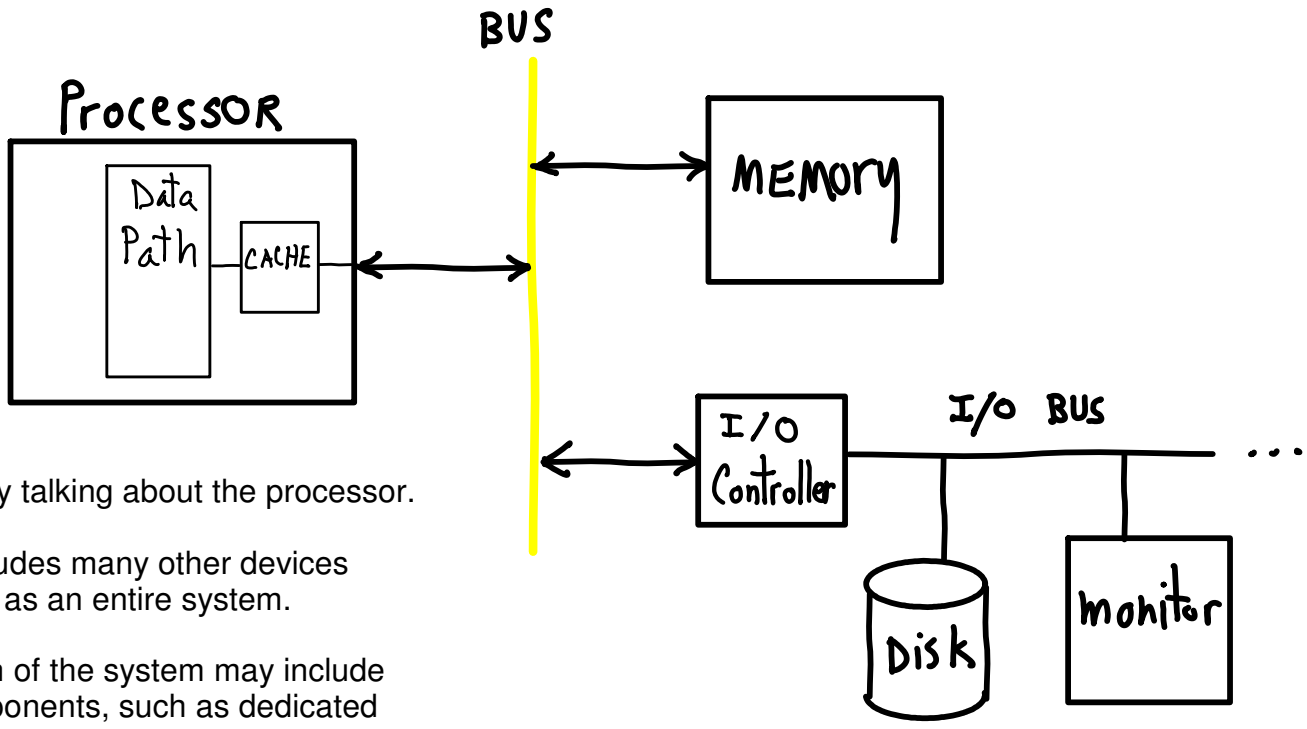
Harvard Architecture

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



How do we map our

System (von Neumann)



We are principally talking about the processor.

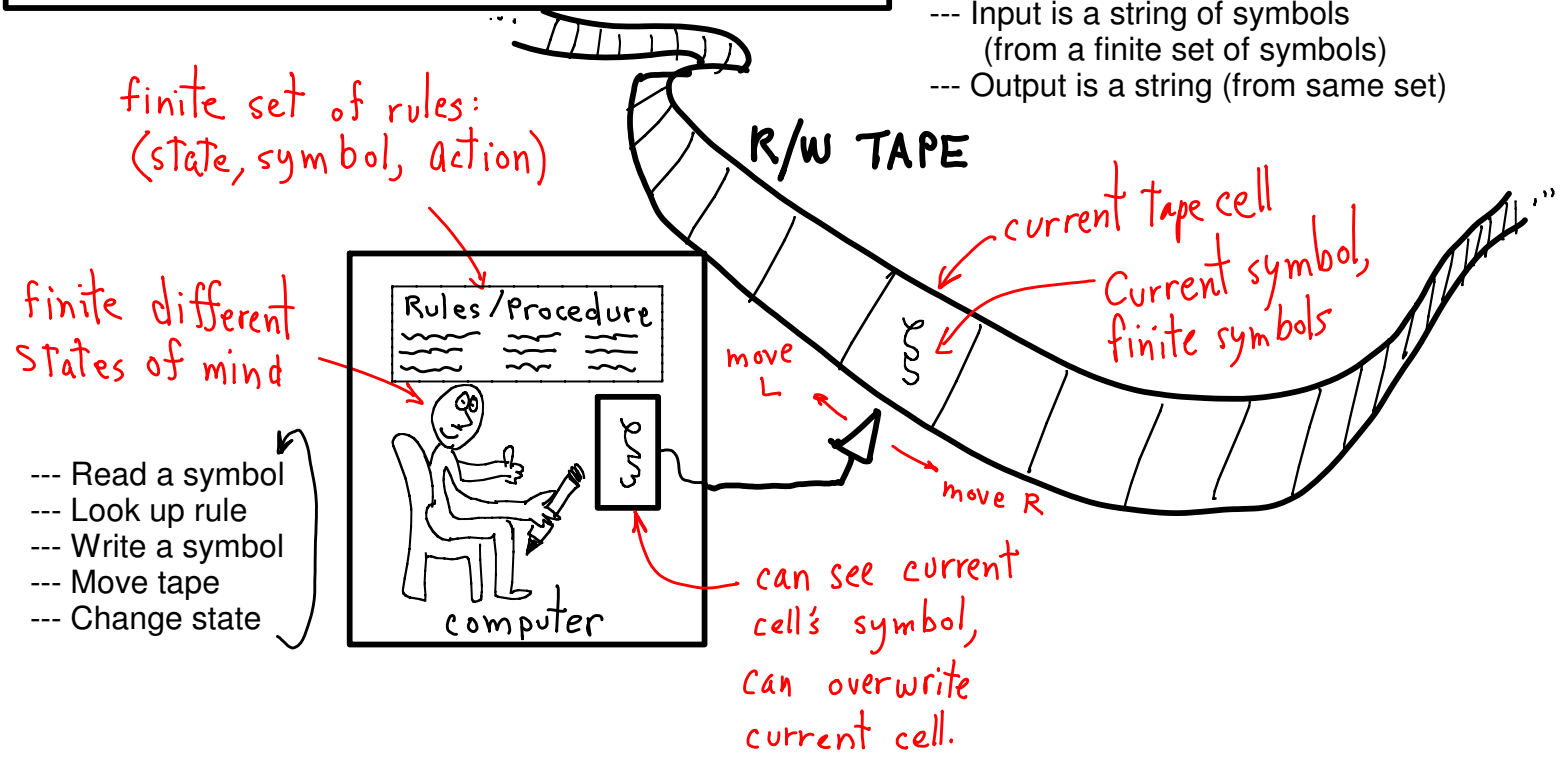
A wider view includes many other devices coupled together as an entire system.

The configuration of the system may include many other components, such as dedicated communication channels for video, GPUs, ...

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)



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

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.

Big idea: **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**

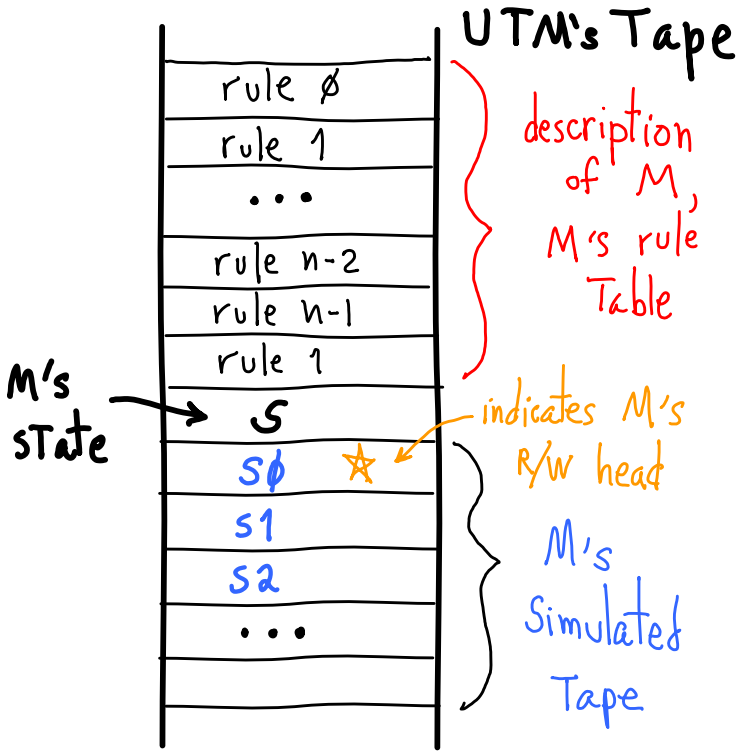
Simulation-step-3:
Find Rule Table

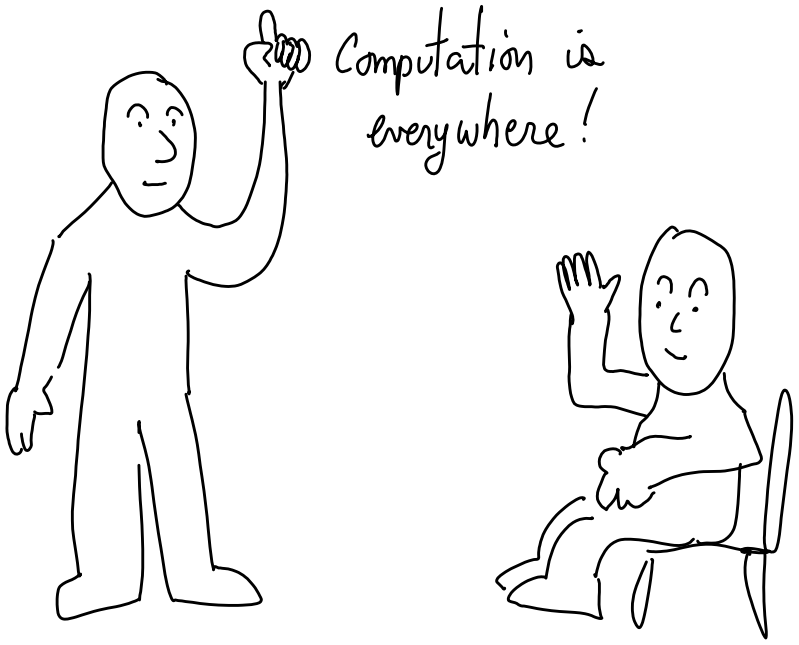
Simulation-step-4:
Search for rule for State **S**
Check if input == **A**
If **S** and **A** do not match Rule, find next Rule

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

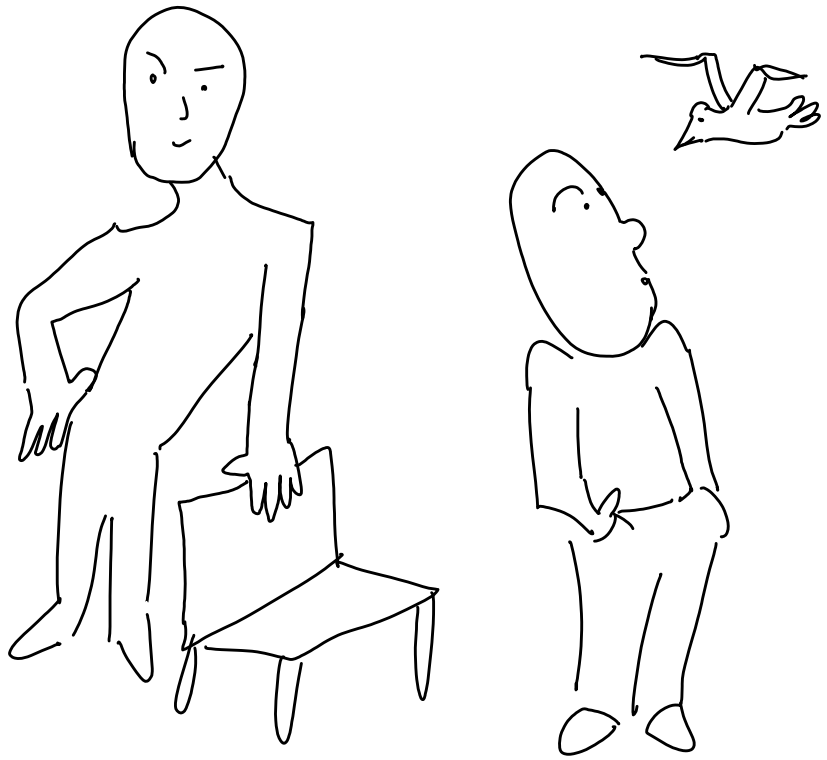
Simulation-step-7:
Find Rule's next-State, **N**
Find M's current-state cell
Write **N**

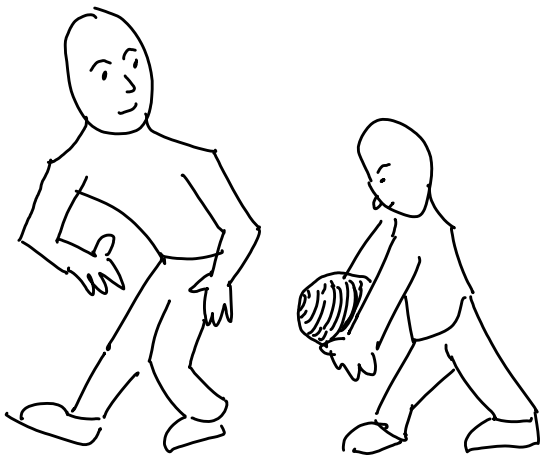




Eham: Computation is everywhere.
Drah: Where?
E: Everywhere!
D: A car crash?
E: Yes.
D: A doll house?
E: Yes.
D: Me?
E: Yes.
D: What is the same about them?
E: They all change.
D: So, computation is change?
E: Yes.
D: Everything changes, so computation is everywhere?
E: Yes.
D: What is computation?
E: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

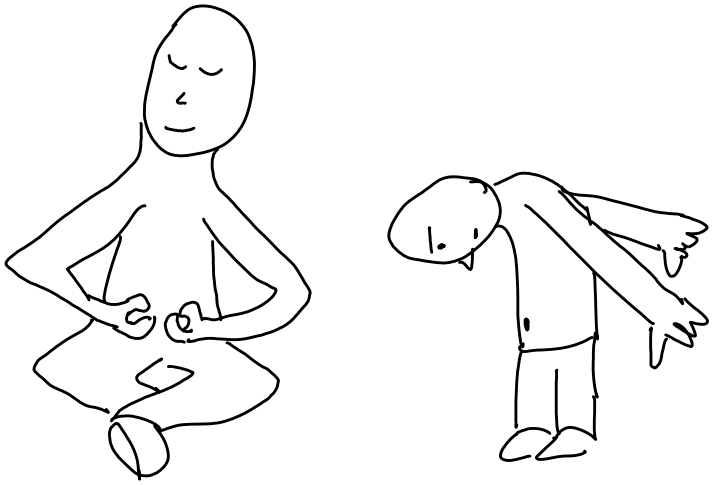
D: So, everything changes, and because everything changes, everything is computation, and computation is change.
E: Yes!
D: Oh.
E: You see, it is really quite simple.
D: How simple?
E: There is a model.
D: A model?
E: Yes.
D: How is there a model?
E: Things are one way, then they are another.
D: And 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?



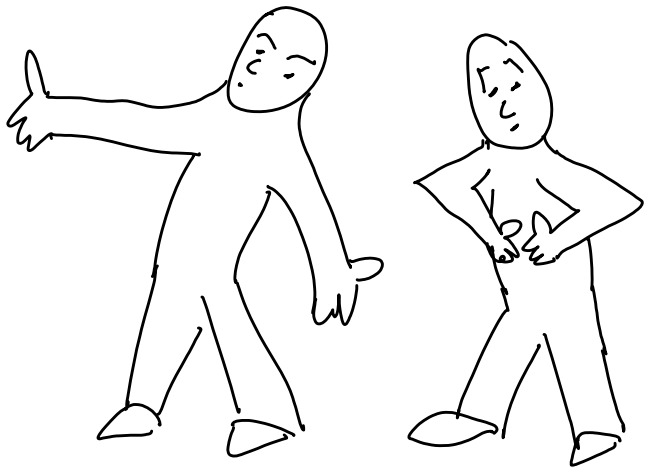


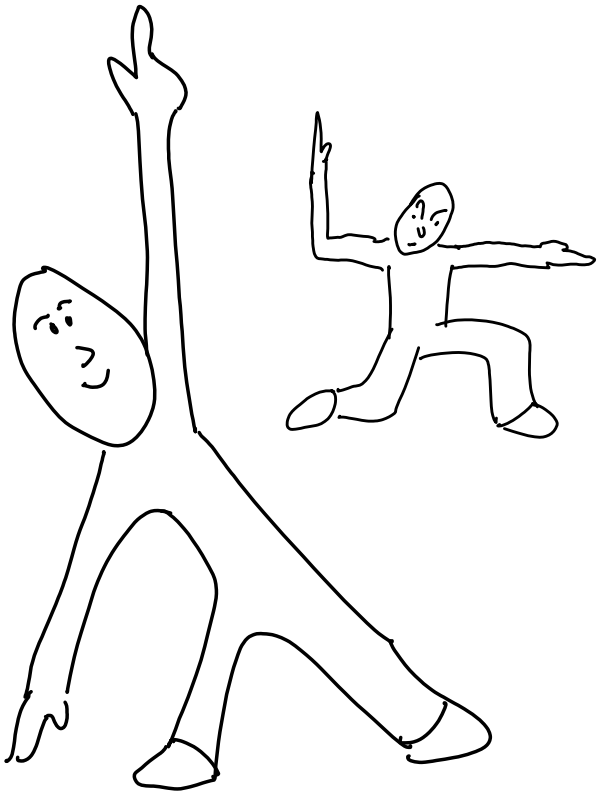
D: And a model means things are one way, then another.
E: Now you've got it.
D: Isn't that the same as change?
E: Quite right.
D: So, a model is change and change is computation and change is computation because there is a model?
E: See, now you're getting the hang of it.
D: Oh.

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?