Digital Logic

- Boolean functions

$f: \{0,1\}^n \to \{0,1\}$

$\{0,1\}^n$: the set of all n-tuples

AND(F, F) = F
AND(F, T) = F
AND(T, F) = F
AND(T, T) = T

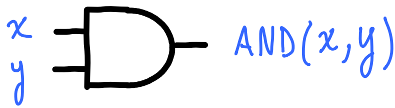
TRUTH Table (x, y: variables)

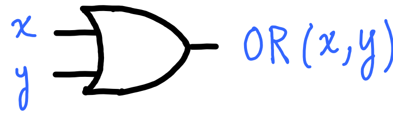
x	y	AND
0	0	0
0	1	0
1	0	0
1	1	1

$x \cdot y$

x is a proposition
x = T or x = F

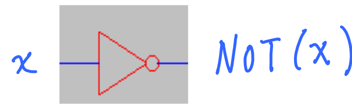
- gates





x	y	OR
0	0	0
0	1	1
1	0	1
1	1	1

$x + y$



x	NOT
0	1
1	0

$\bar{x}$

- function Primitives (function composition)
  min-terms, max-terms

$f(x) = s(g(x))$ $\quad f(x,y) = r(g(x,y), h(x,y))$

$f(x,y) = \text{OR}(\text{AND}(\cdot,\cdot), \text{AND}(\cdot,\cdot), \ldots \text{AND}(\cdot,\cdot))$

(first argument: x or $\bar{x}$; second argument: y or $\bar{y}$)

= min-term expansion of f

$= \text{AND}(\text{OR}(\cdot,\cdot)\ \text{OR}(\cdot,\cdot), \ldots \text{OR}(\cdot,\cdot))$

= max-term expansion of f

x	y	f(x,y)
0	0	0
0	1	1
1	0	1
1	1	0

=

x	y	g(x,y)
0	0	0
0	1	1
1	0	0
1	1	0

+ OR

x	y	h(x,y)
0	0	0
0	1	0
1	0	1
1	1	0

$f(0,0) = g(0,0) + h(0,0)$
$f(0,1) = g(0,1) + h(0,1)$
$f(1,0) = g(1,0) + h(1,0)$
$f(1,1) = g(1,1) + h(1,1)$

what is this function?

x	y	$g(x,y)$
0	0	0
0	1	1
1	0	0
1	1	0

AND(NOT(0) , 0) = 0

AND(NOT(0) , 1) = 1

AND(NOT(1) , 0) = 0

AND(NOT(1) , 1) = 0

$g(x,y)$ = AND(NOT(x) , y) $= \overline{x} \cdot y$

= min-term: $m_{01}(x,y)$ aka $m_1(x,y)$

x	y	$h(x,y)$
0	0	0
0	1	0
1	0	1
1	1	0

→ $x \cdot \overline{y}$ or m_2

the others

$m_0 = \overline{x} \cdot \overline{y}$

$m_3 = x \cdot y$

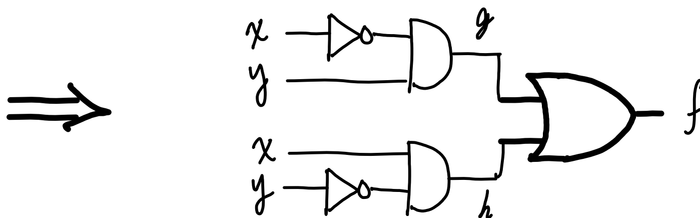
$\Longrightarrow f(x,y) = g(x,y) + h(x,y)$

$= \overline{x} \cdot y + x \cdot \overline{y} = m_1 + m_2$

(in general $f(\cdot) = \sum_I m_i$)

$I \subseteq \{0, 1, \ldots 2^n\}$

$\Longrightarrow$



Any n-ary function can be built as an OR of minterms.

Minterms are ANDs of variables or their negations.

Max Terms

Algebra

$A + 0 = A$ $\begin{cases} \text{if } A = 1, & A+0 = 1 = A \\ \text{if } A = 0, & A+0 = 0 = A \end{cases}$

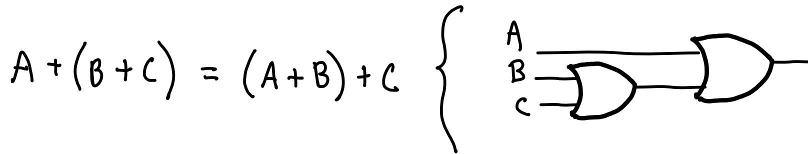
$A \cdot 0 = 0$

$A + B = B + A$

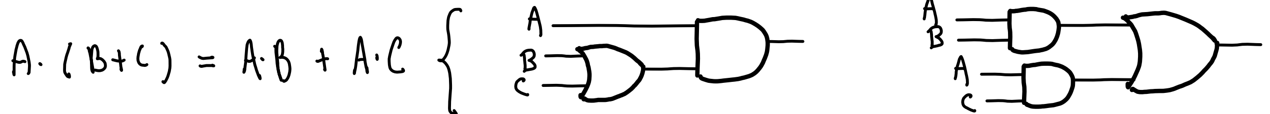
A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

A	B	B+A
0	0	0
0	1	1
1	0	1
1	1	1

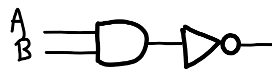
$A + (B + C) = (A + B) + C$

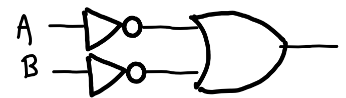


$A \cdot (B + C) = A \cdot B + A \cdot C$



$\overline{A \cdot B} = \overline{A} + \overline{B}$ (De Morgan's Law I)





Equality $(0, 1, \cdot, +)$ $\Longrightarrow$ Equality $(1, 0, +, \cdot)$ (Duality)

$\overline{A \cdot B} = \overline{A} + \overline{B}$ $\Longrightarrow$ $\overline{A + B} = \overline{A} \cdot \overline{B}$ (De Morgan's Law II)

$A \cdot 0 = 0$ $\Longrightarrow$ $A + 1 = 1$

A	B	f
0	0	1
0	1	0
1	0	0
1	1	1

$\Longrightarrow$

A	B	$\overline{f}$
0	0	0
0	1	1
1	0	1
1	1	0

$\Longrightarrow \overline{f} = \overline{A} \cdot B + B \cdot \overline{A}$

$\Longrightarrow f = \overline{\overline{f}} = \overline{\overline{A} \cdot B + B \cdot \overline{A}}$

$= \overline{\overline{A} \cdot B} \cdot \overline{B \cdot \overline{A}} = \underbrace{(A + \overline{B})}_{\text{Max-term}} \cdot \underbrace{(\overline{A} + B)}_{\text{Max-term}}$

$$\Rightarrow \quad (A+\overline{B})\cdot(\overline{A}+B) = A\cdot(\overline{A}+B) + \overline{B}\cdot(\overline{A}+B) \quad \text{(dist.)}$$
$$= A\cdot\overline{A} + A\cdot B + \overline{B}\cdot\overline{A} + \overline{B}\cdot B \quad \text{(dist.)}$$
$$= 0 + A\cdot B + \overline{B}\cdot\overline{A} + 0$$
$$= A\cdot B + \overline{B}\cdot\overline{A}$$

min-terms for f

Simplification

$$g = A\cdot B + \overline{A}\cdot B = (A+\overline{A})\cdot B = 1\cdot B = B$$

Karnaugh map

A \ B	0	1
0	0	1
1	0	1

as long as B = 1, it doesn't matter what A is.

f

AB \ C	0	1
00	0	0
01	0	0
11	1	1
10	1	1

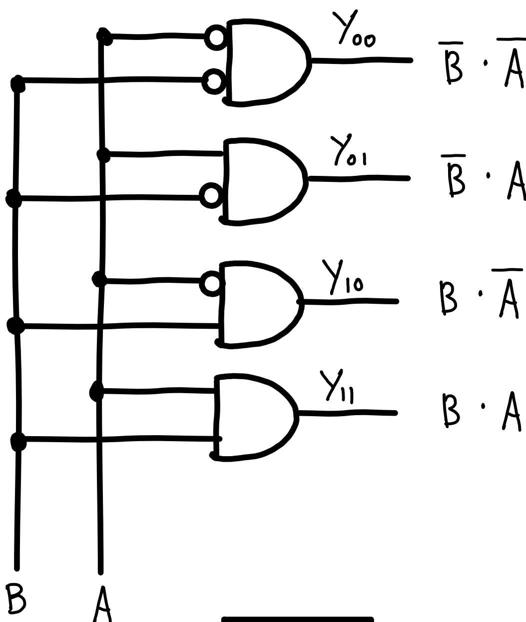
(A=1)

A

(A=1) • (C=0)

$A\overline{C}$

Decoder



$\overline{B}\cdot\overline{A}$

$\overline{B}\cdot A$

$B\cdot\overline{A}$

$B\cdot A$

B	A	Y_{00}
0	0	1
0	1	0
1	0	0
1	1	0

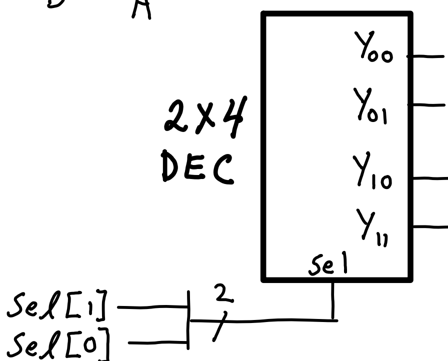
code = 00

B	A	Y_{01}
0	0	0
0	1	1
1	0	0
1	1	0

code = 01

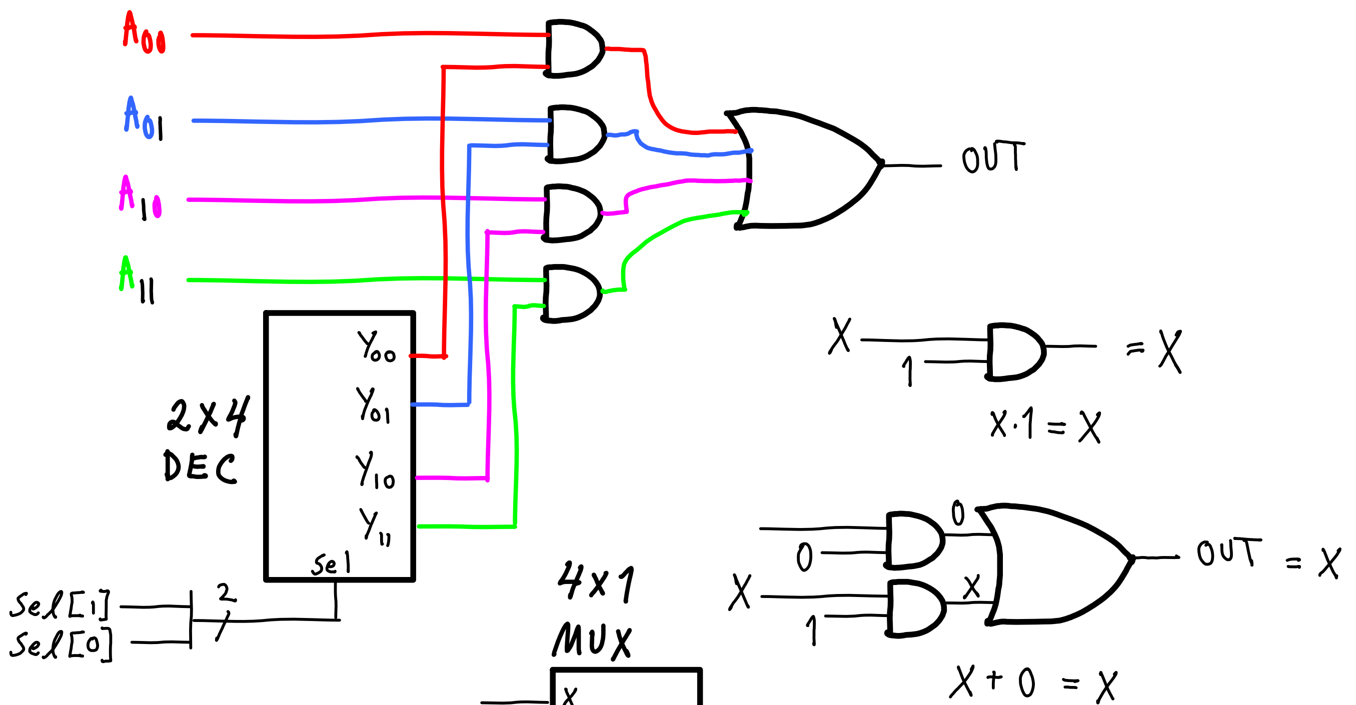
B	A	Y_{10}
0	0	0
0	1	0
1	0	1
1	1	0

B	A	Y_{11}
0	0	0
0	1	0
1	0	0
1	1	1



B	A	Y_{00}	Y_{01}	Y_{10}	Y_{11}
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

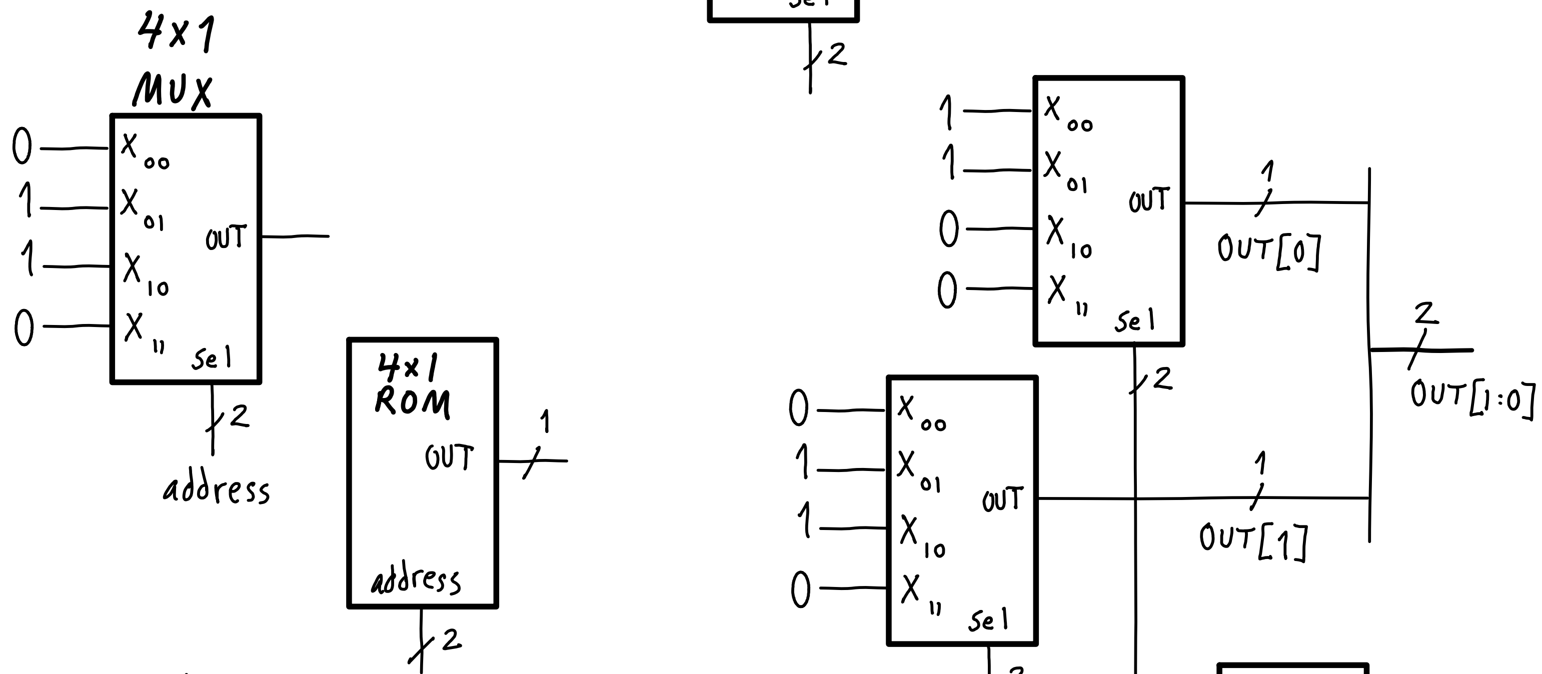
} exactly one output = 1

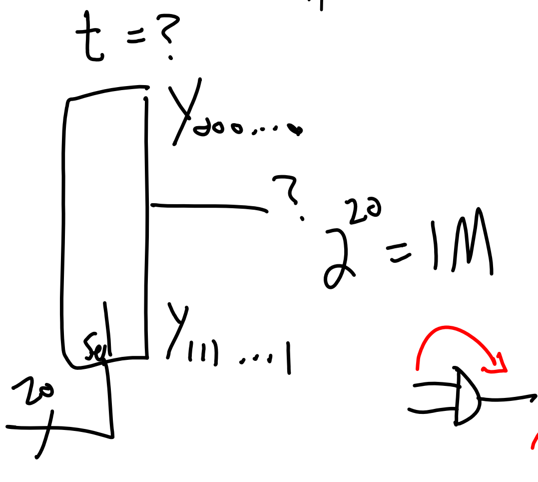


4x1
MUX

0
X
0
1
X
OUT = X

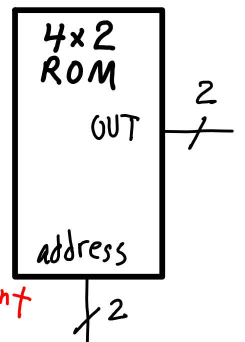
$X + 0 = X$





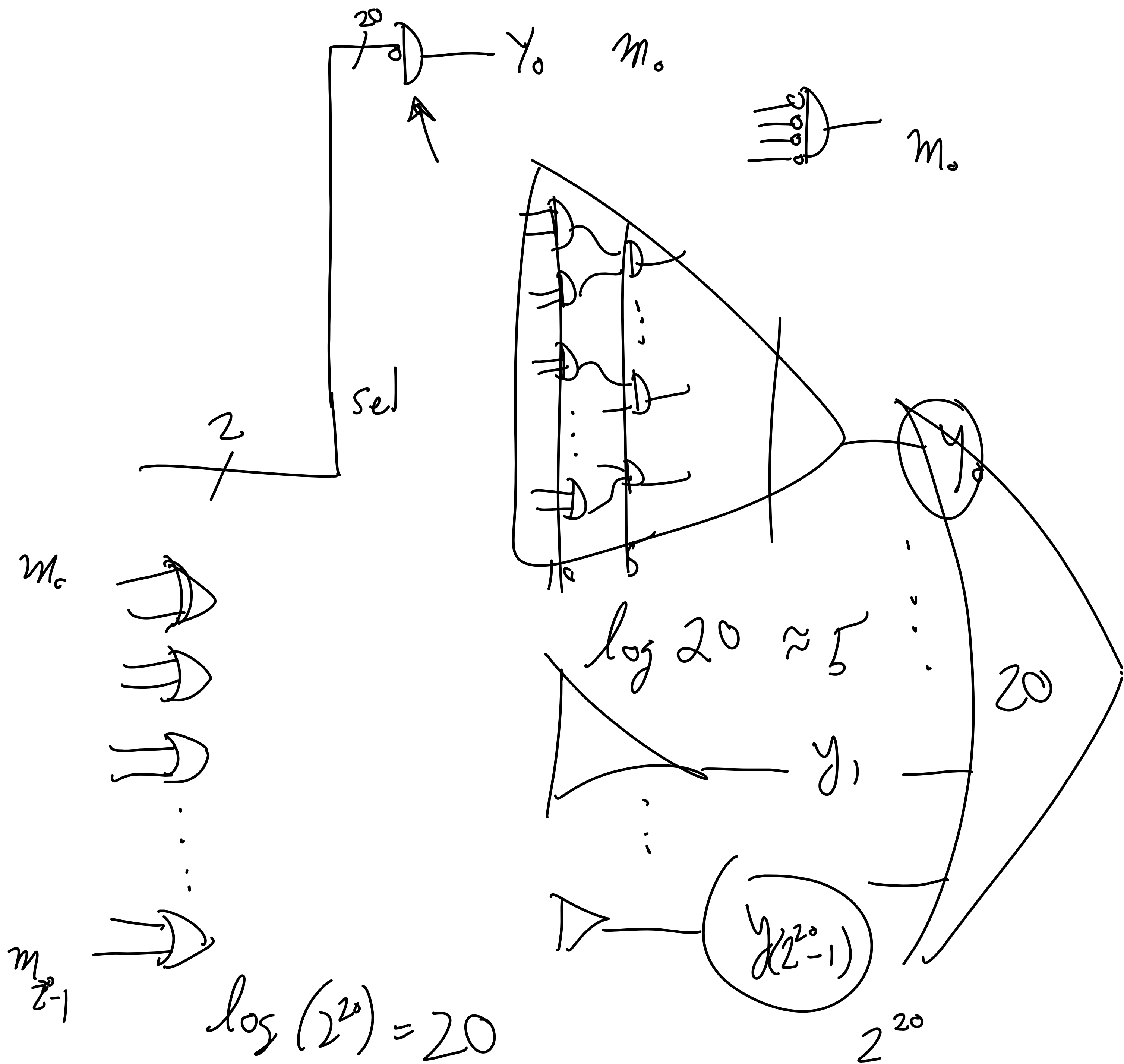
$Word_{00}$	0	1
$Word_{01}$	1	1
$Word_{10}$	1	0
$Word_{11}$	0	0
	b_1	b_0

Content of ROM



Q. How much is the time delay for the 4X1 MUX? Assume only basic, 2-input gates are used (AND, OR, NOT). Assume each basic, 2-input gate has a delay of 1 unit from the time its input changes until its output changes correspondingly.

Q. How much is the time delay for a (1 M)X1 MUX? 1M is 2^20.



Don't Cares

B	A	Y
0	0	0
0	1	1
1	0	1
1	1	1

⟹

B	A	Y
0	0	0
0	1	1
1	X	1

when B = 0, we care what A is.

if B = 1 we don't care what A is.

Suppose our machine works like this:

if B = 0, then Y = A

But, if B = 1, it makes no difference what Y is.

B	A	Y
0	0	0
0	1	1
1	0	X
1	1	X

output don't cares: can be either 0 or 1, whichever is convenient.

B	A	Y
0	0	0
0	1	1
1	X	X

ADDER

C_{in}	B	A	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

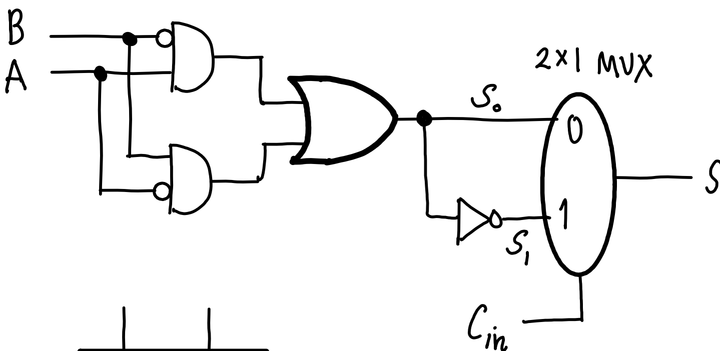
B	A	S_0
0	0	0
0	1	1
1	0	1
1	1	0

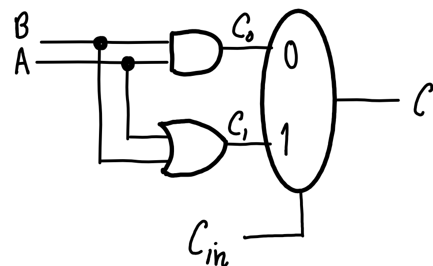
$$S_0 = \overline{B} A + B \overline{A}$$

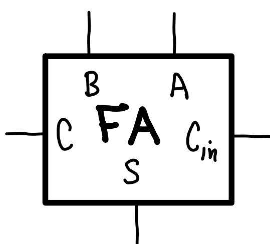
B	A	S_1
0	0	1
0	1	0
1	0	0
1	1	1

$$S_1 = \overline{S_0}$$

$$S = \overline{C_{in}} \cdot \overline{B} \cdot A$$
$$+$$
$$\overline{C_{in}} \cdot B \cdot \overline{A}$$
$$+$$







1-bit Full Adder

The above Full Adder (FA) includes two 1-bit, 2X1 MUXes.

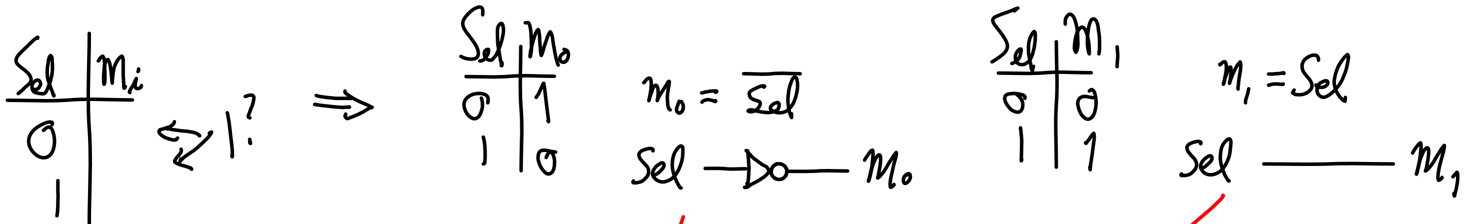
Q. What are the min-term functions for one-variable Boolean functions?

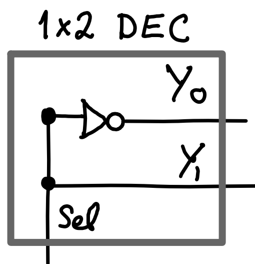
Q. Build a 1X2 DEC.

Q. Show a logic design for a 1-bit, 2X1 MUX. Use the organization shown above for the 1-bit, 4X1 MUX. That is, use a DEC to gate input signals to an OR.

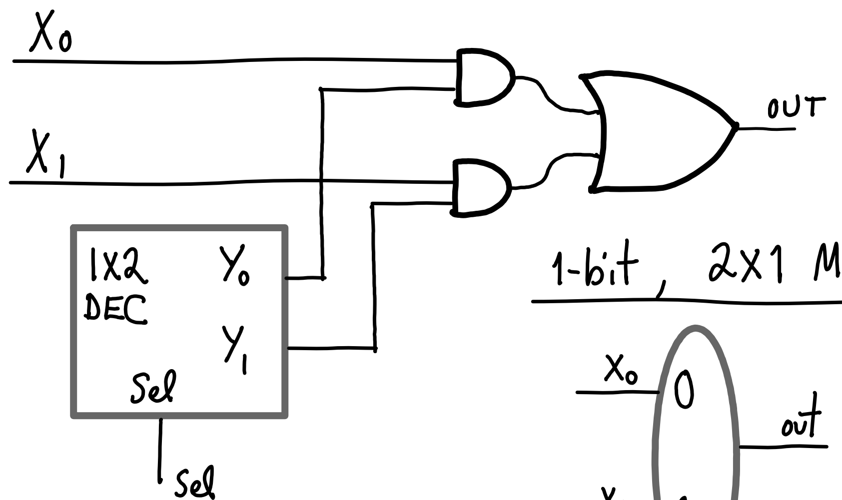
Q. How would we build a 3-bit, 2X1 MUX? Show a design using 1-bit, 2X1 MUXes.

minterms for 1-bit input?

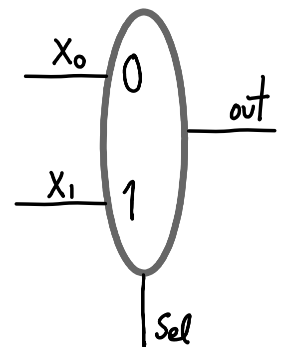




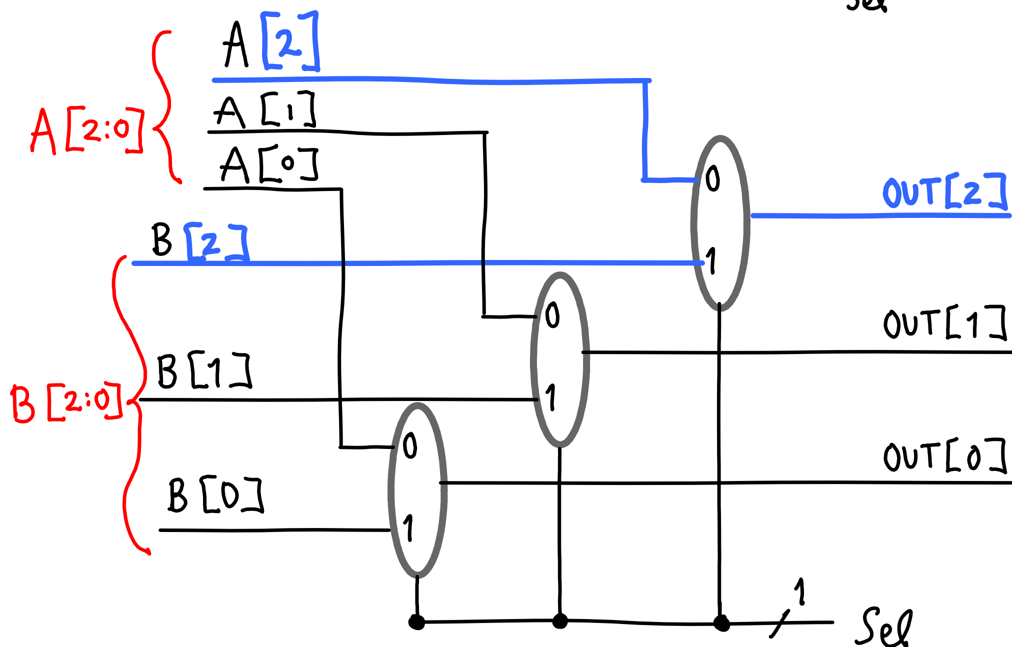
every minterm possible

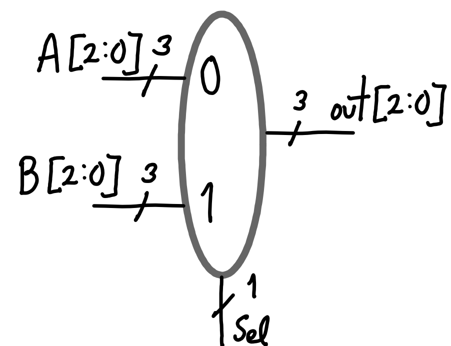


1-bit, 2X1 MUX

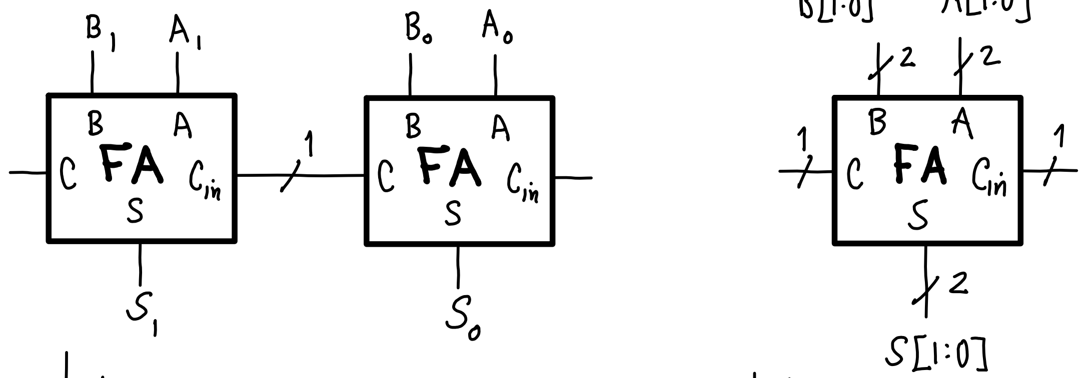


3-bit, 2X1 MUX

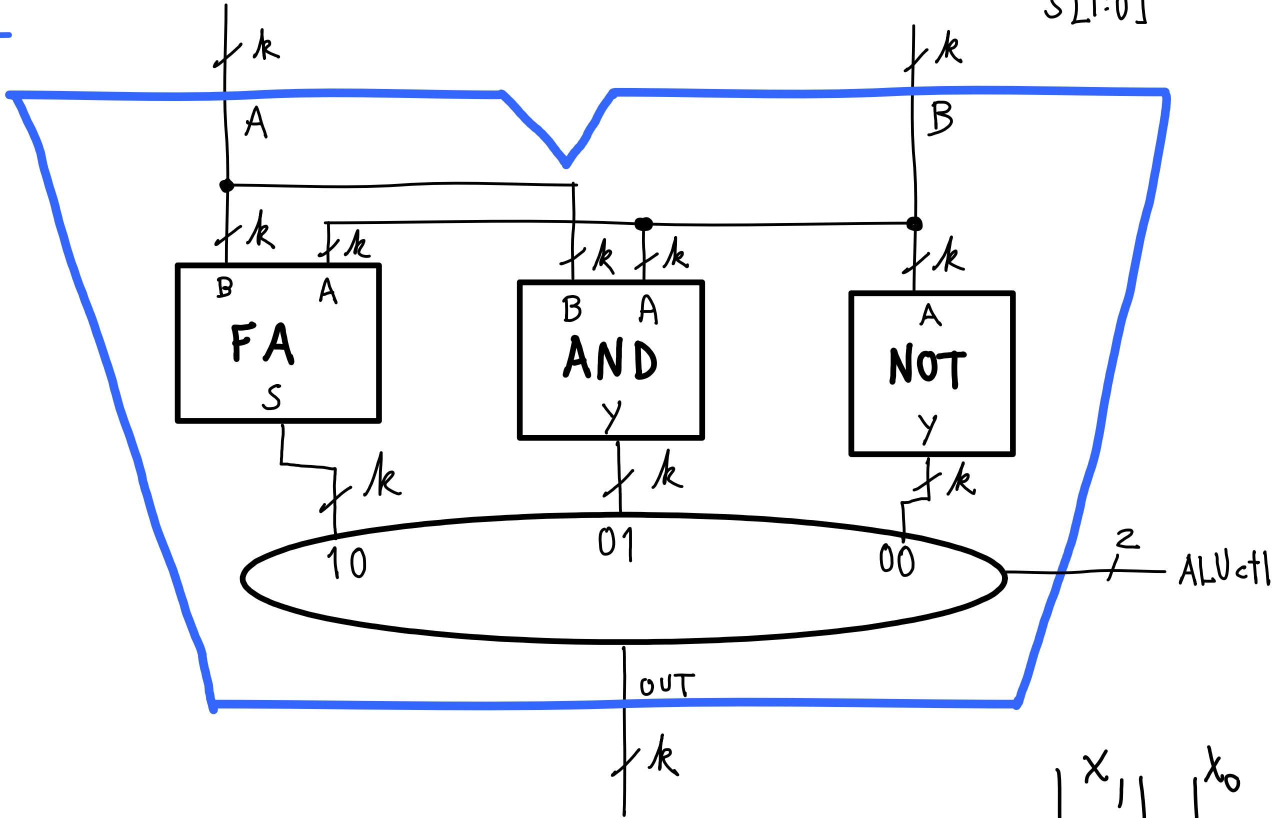


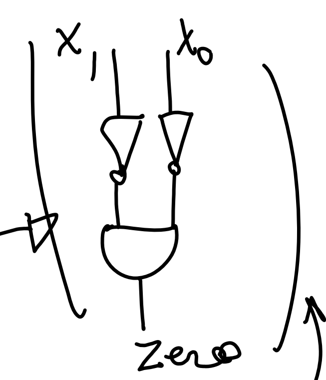


2-bit FA



ALU

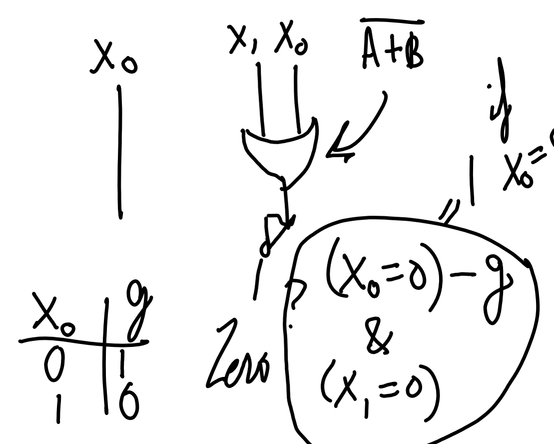





```
module MIPSALU (ALUctl, A, B, ALUOut, Zero);

input [3:0] ALUctl;
input [31:0] A,B;

output reg [31:0] ALUOut;
output Zero;

assign Zero = (ALUOut==0); //-- Zero is true if ALUOut is 0

always @(ALUctl, A, B) begin //-- reevaluate if these change

    case (ALUctl)
    0:  ALUOut <= A & B;
    1:  ALUOut <= A | B;
    2:  ALUOut <= A + B;
    6:  ALUOut <= A - B;
    7:  ALUOut <= A < B ? 1 : 0;
    12: ALUOut <= ~(A | B); // result is nor
    default: ALUOut <= 0;
    endcase

end

endmodule
```

4'b0000 →

$\bar{A} \cdot \bar{B} =$



Algorithms and Computers

We would like to have:
-- A simple concept of computation/computing/computers
Why?
--- 1. When we build one, we can tell what we want: can it do what it is supposed to do?
--- 2. When we see one, we can recognize it (eg. is a QM machine a computer?)
--- 3. When we look at a complex system, we can identify its fundamental structure: abstraction.
--- 4. We can define what we mean by an algorithm (ie., TM that always halts).

BIG IDEA: Define computation (automatic procedure)

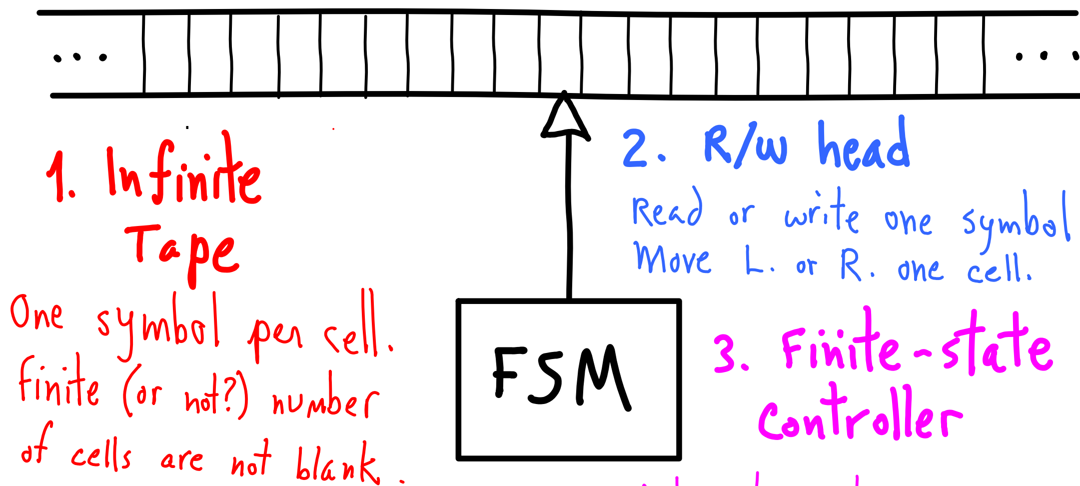
Church-Turing Thesis:

Any computation can be done by some Turing Machine (TM).

(Efficiently?)

Can't prove, but works so far.

TM model of Computation



In one state at a time. Finite number of states. Changes state deterministically, next state depends on symbol read and current state. Specific start state.

4. Finite symbol set, Σ.

For example, $\Sigma = \{0, 1\}$

or

$\Sigma = \{\#, 0, \ldots, 9, a \ldots g\}$ ($\#$ ← blank)

5. User manual

Defines:

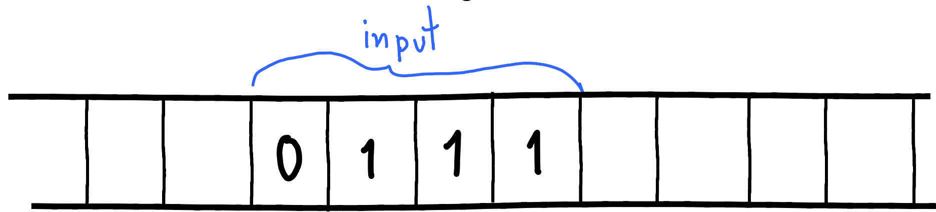
1. Proper input
2. Proper output
3. Interpretation
4. Halting states
5. Start state

Operation

1. Read one symbol
2. Depending on state and symbol,
   --- write one symbol
   --- move R/W head one cell L. or R.
   --- change state
3. Repeat, or if current state is a "halting state", stop operating.

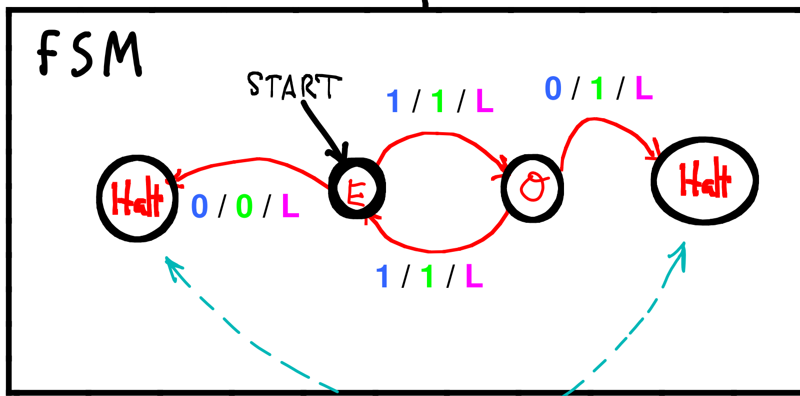
Finite-State Machine Controller

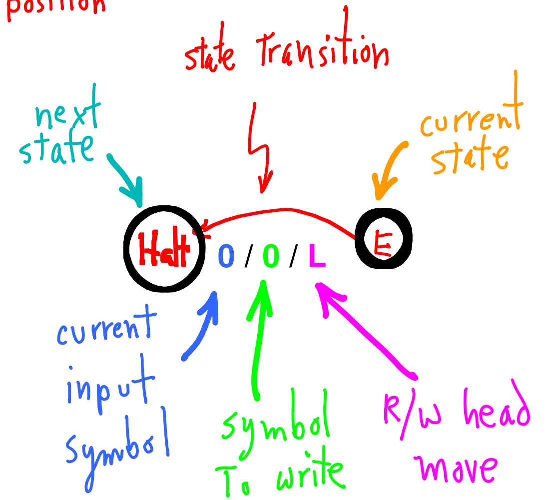
State Transition diagrams



initial Tape configuration

current cell, initial R/W head position





Starts in state E.

These represent same state.

a state-transition rule →

If (current state == E) AND
(current symbol == 0)
then
(write symbol == 0)
(move R/W head L.)
(new state == Halt)

What does it do?

On this specific input?

On any general input?

Alternate representation of FSM:

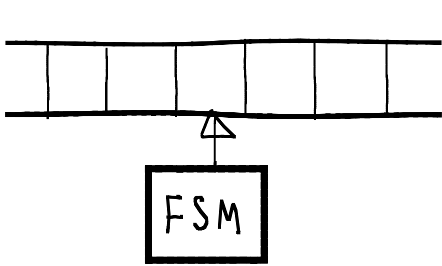
(start state = E)

(state, input)	(output, move, next state)
(E, 1)	(1, L, O)
(E, 0)	(0, L, Halt)
(O, 1)	(1, L, E)
(O, 0)	(1, R, Halt)

Same information as diagram.

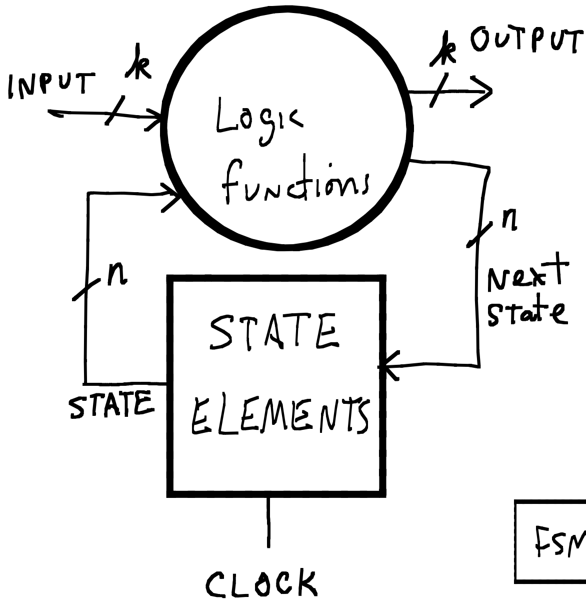
→ An encoding of the FSM.

TM-implementation



k-bit symbols

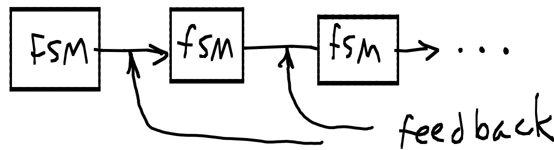
e.g., $\Sigma = \{000, 001, \ldots, 111\}$

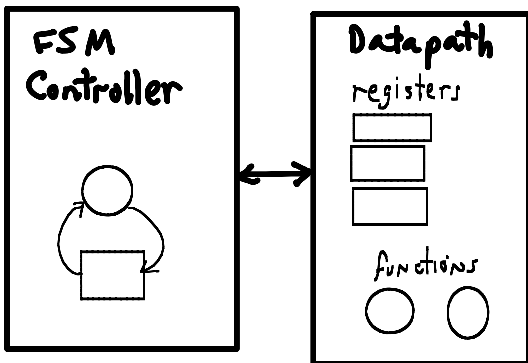


FSM has input/output, but from/to where?

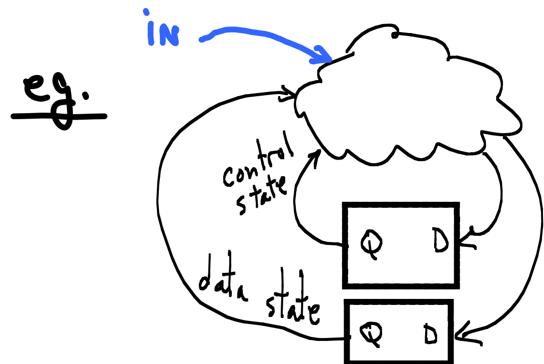
(1) Other FSMs in same machine

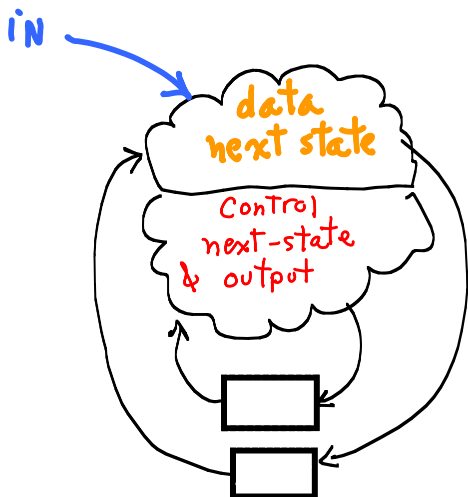
(2) Feedback loops

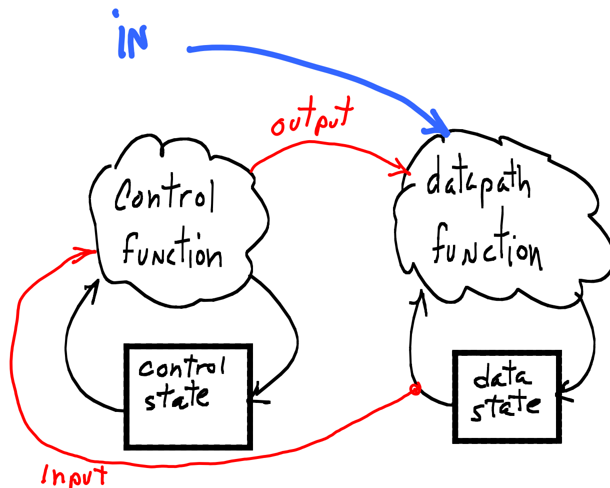




We like to separate **state** into two "types", **control** and **data**. Eg., some state elements are for "control" state, and some are for "data" state.





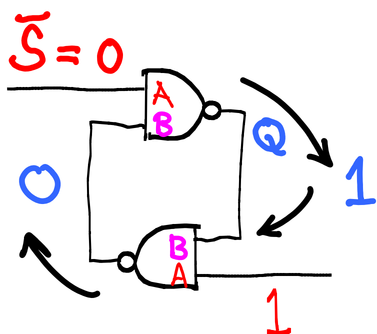


Basic Sequential elements

We need to remember our "state".

--- Stay in one state: use feedback.
--- State is output Q.
--- How do we change state?

Force Q=1.
Stable.



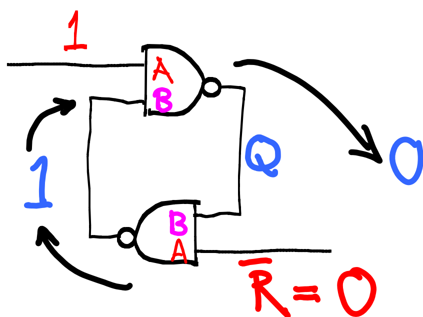
NAND

A	B	Q
0	0	1
0	1	1
1	0	1
1	1	0

A=0 ⇒ Q = 1

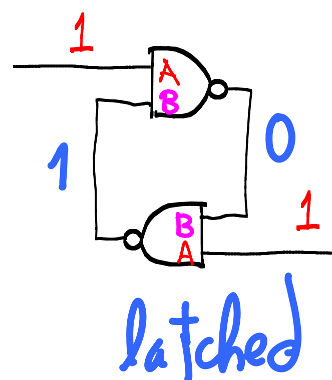
$Q = \bar{B}$

When A=0,
NAND=1.
When A=1,
NAND=(-B).

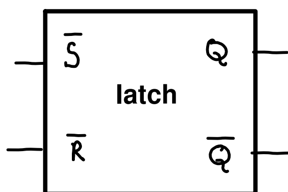


Force Q=0.
Stable.

R=1



$\bar{S}$	$\bar{R}$	
1	1	stable
0	1	set Q=1
1	0	reset Q=0



Use a basic latch to build an SR-latch:

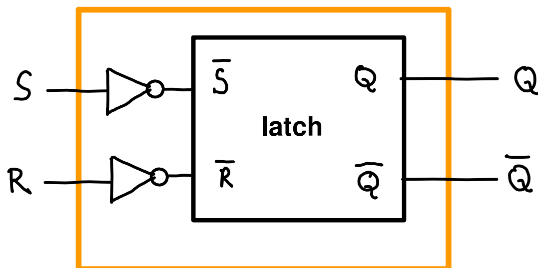
invert S, R inputs.

S = R = 0 stable

S=1, R=0 Q=1

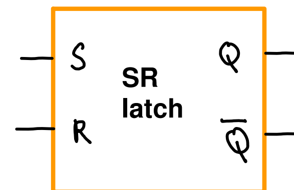
S=0, R=1 Q=0

S = R = 0 stable



SR latch

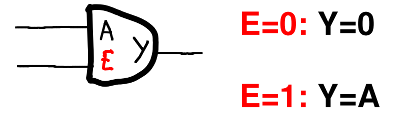
⇒



a basic element fn state

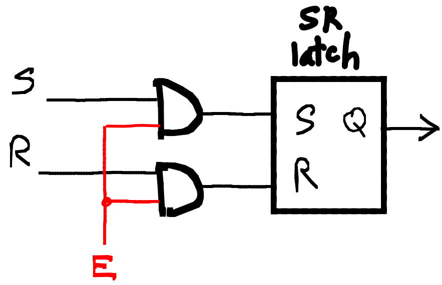
Clocking, part 1: gating/enable

"gating" a signal, A.



E=0: Y=0

E=1: Y=A



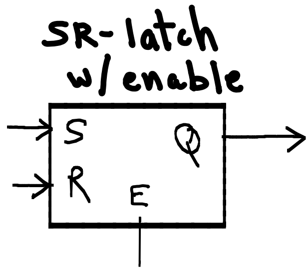
SR-latch w/ enable

E=0: ignores S-R inputs
E=1: transparent

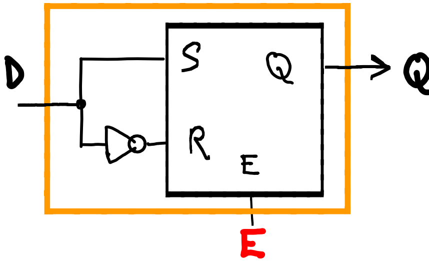
E	A	Y = (AB)
0	0	0
0	1	0
1	0	0
1	1	1

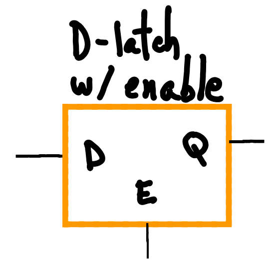
input ignored

passes input to output



simplify



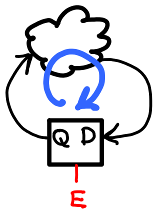


E=1
D=0 (S=0, R=1): Q=0
D=1 (S=1, R=0): Q=1

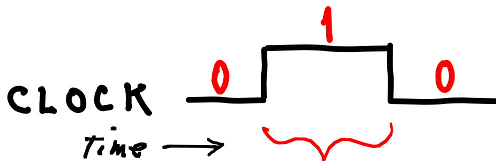
E=0
stable

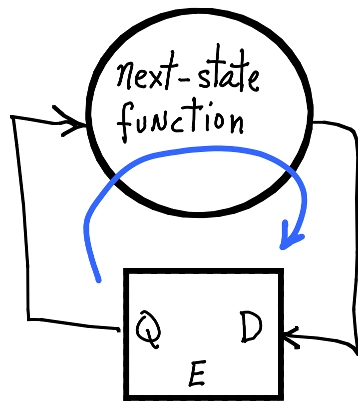
2-phase clocking, D-FF

Problem



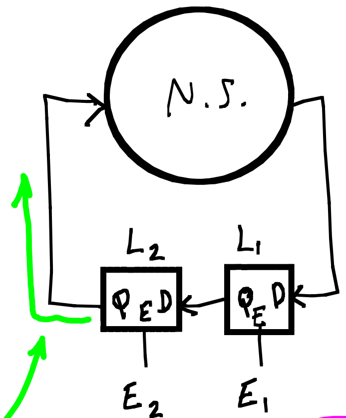
If E=1,
How many times will update to state occur before E=0?

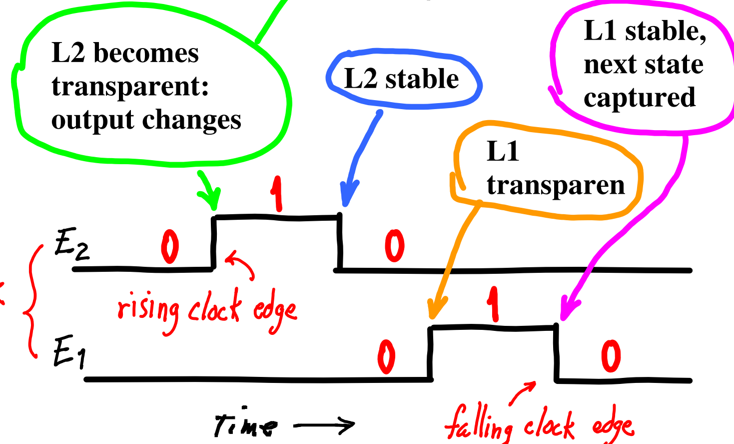




Latch is transparent during the time clock=1.

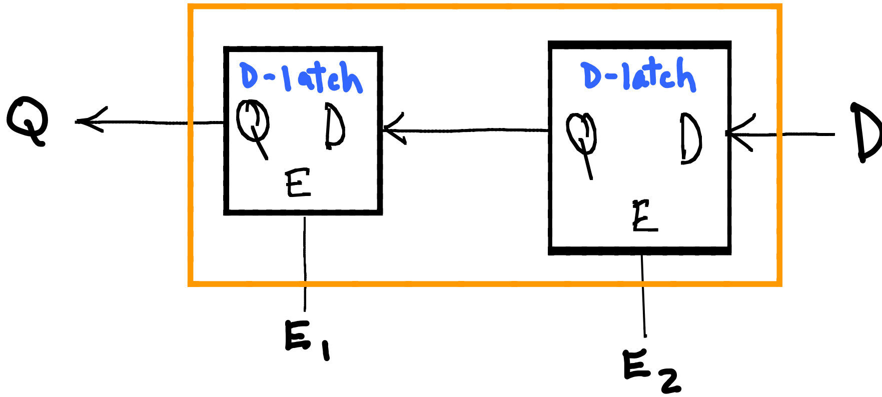
Need to prevent feedback, until ready.

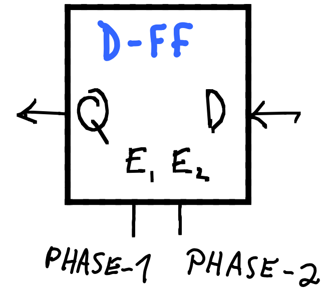




2-phase clock

D-FF

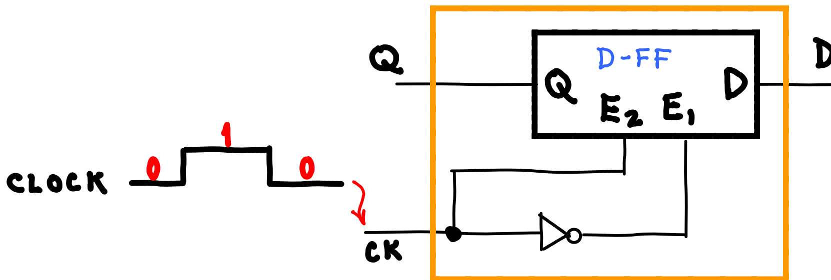


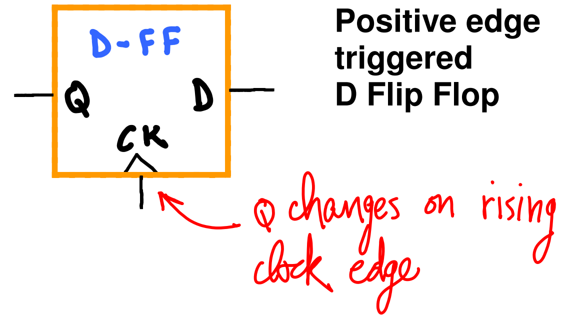


2-Phase Clocking

Separate signals for each latch's enable in FlipFlop. On breadboard we connect PHASE-1 to one data switch, PHASE-2 to another.

implement 2-phases

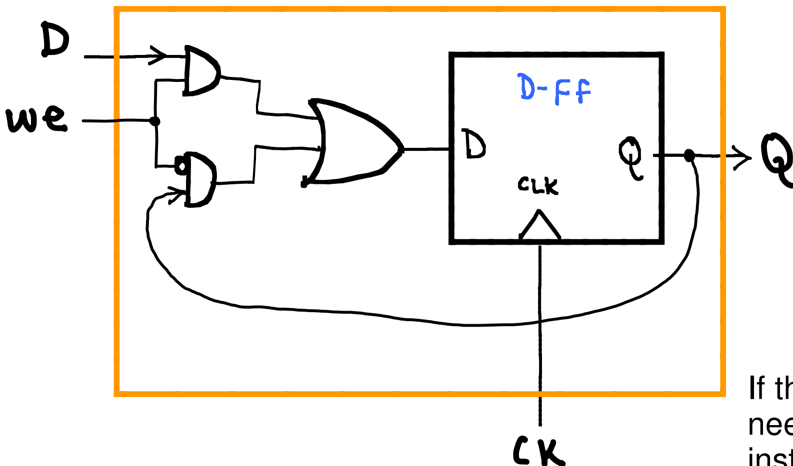


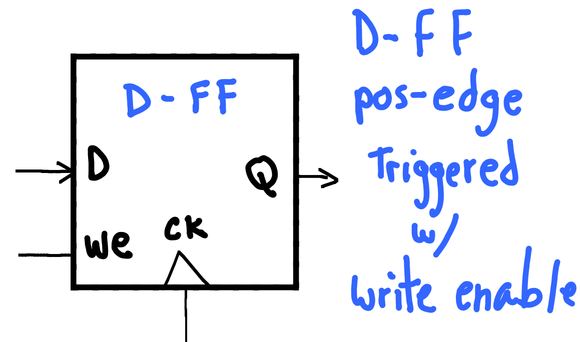


Positive edge triggered D Flip Flop

D-FF, posedge Triggered w/ write enable

We often want to control whether or not the FF will be written into when the clock pulse arrives: add an "enable" input. When enable is 0, the current state is written back into the FF. Otherwise, D is written.





If there is no feedback path from Q to D, we do not need a flip-flop, we can use a write-enable latch instead. Datapaths sometimes can use latches.

FSM in ROM (n-bit state, i-bit input, k-bit FSM output)

(STATE, INPUT) is ROM address
n bits + i bits ===> 2^(n+i) ROM locations

(NEXT-STATE, FSM-OUTPUT) is ROM output
n bits + k bits ====> (n+k) bits per location

===> 2^(n+i) location by (n+k)-bit word ROM

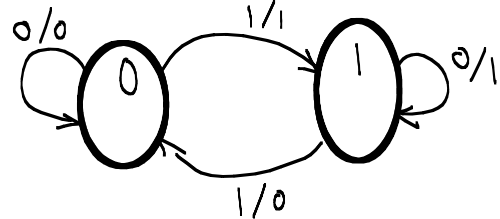
ANY FSM (Mealy or Moore) can be built as a ROM

NOTE: A Moore machine's output depends only on state
===> use n-bit addresses, one ROM location per state.

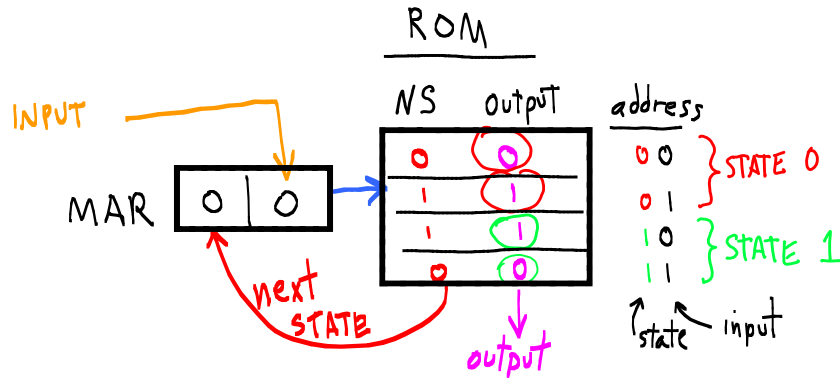
BUT, next-state depends on current-state+input. Encode part of next-state function in ROM word as NS-CODE, and use external logic to calculate next-state function: next-state = f(INPUT, NS-CODE). This is what is done in the LC3's micro-coded controller.

Every possible FSM can be built as a ROM.

ROM is very large since there is a word for every possible {state, input} combination.



address		ROM data word (row)	
STATE	INPUT	next-state	output
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0

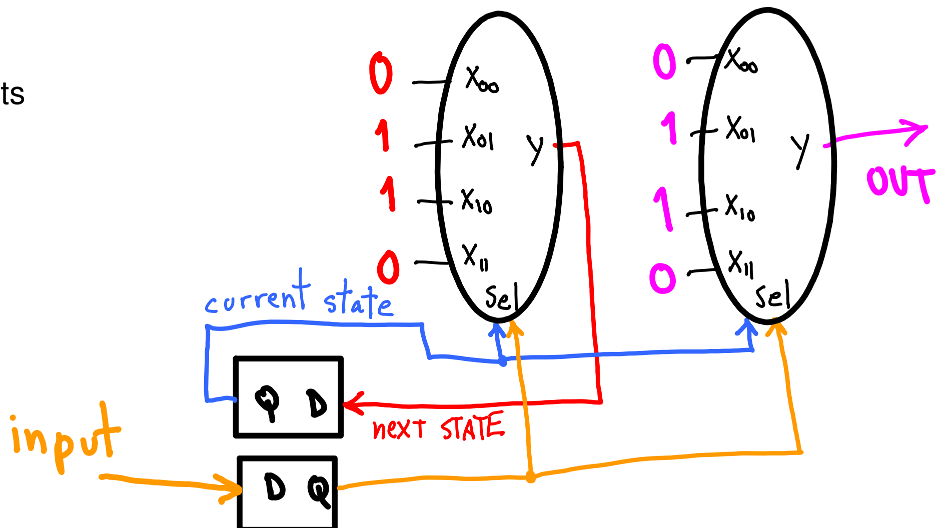


at clock tick:

-- { current state, current input } captured
-- output changes to match captured state/input

-- Every state row has same output
===> Moore Machine

-- Rows for state S have differing outputs
===> Mealy Machine.



We can enumerate all ROMs (and consequently all TMs/digital-computers):

Concatenate ROM content from all words:

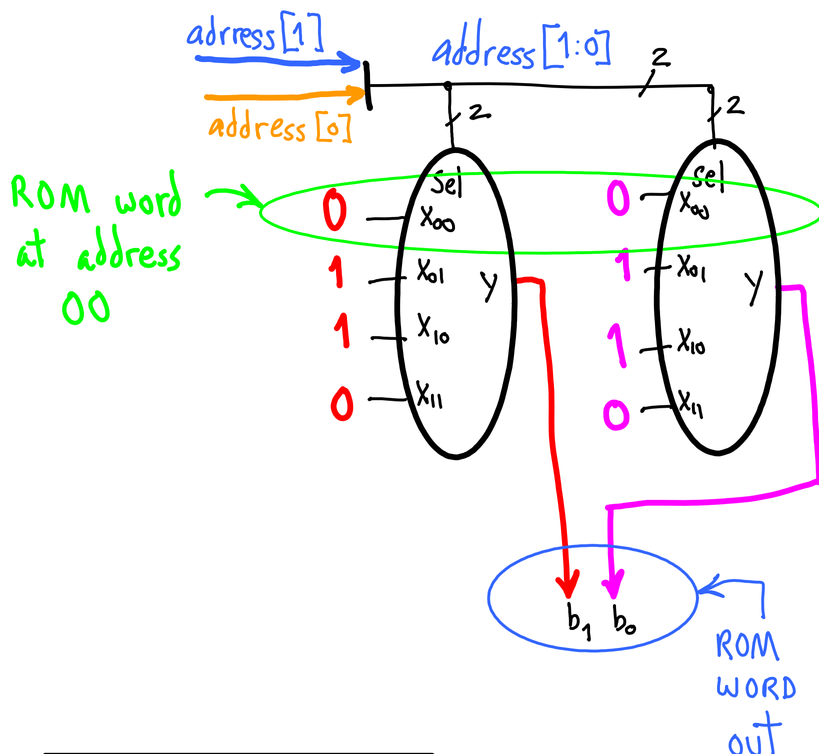
address	content
00	00
01	11
10	11
11	00

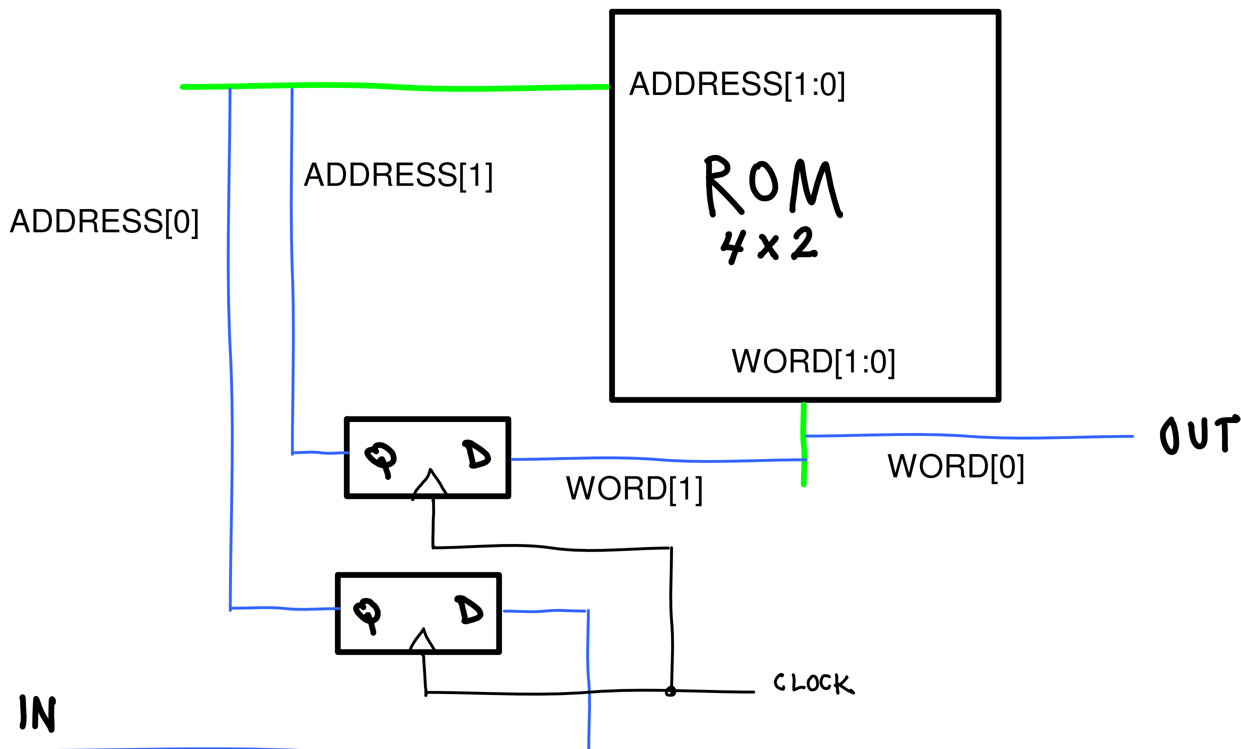
==> 01111000

List all n = i = k = 1 machines:
FSM-0, FSM-1, ..., FSM-256

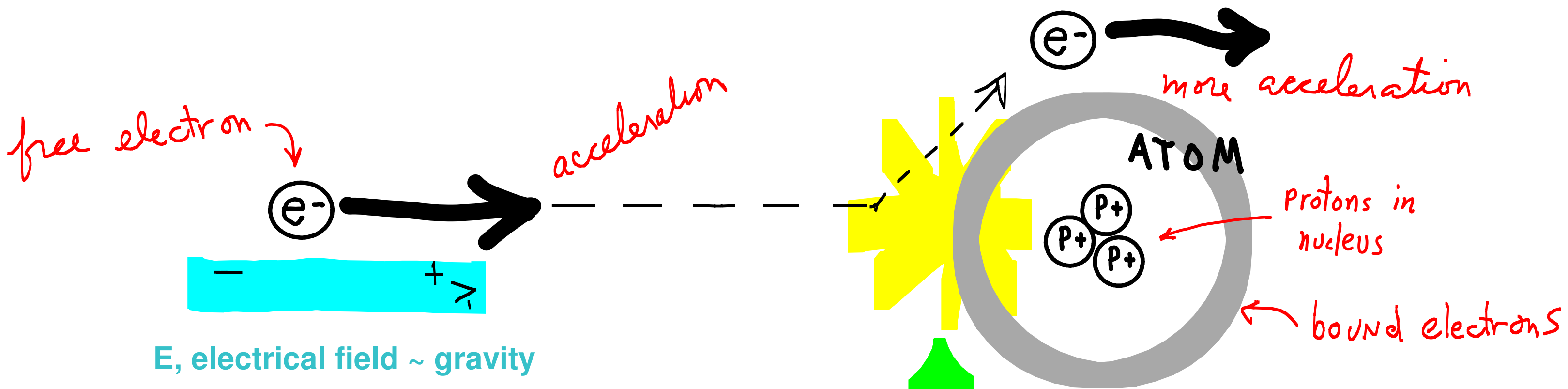
List all n = i = k = 2 machines:
FSM-257, FSM-258, ...

and so on.





Electrons & Heat

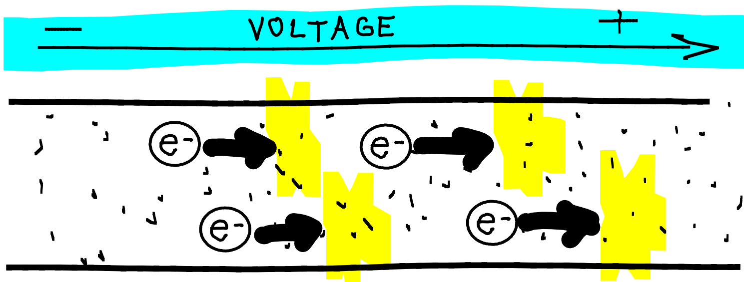


Collision transfers energy to atoms of material, e.g., wire.

Atom motion = Heat.

~All collision energy radiates as heat.

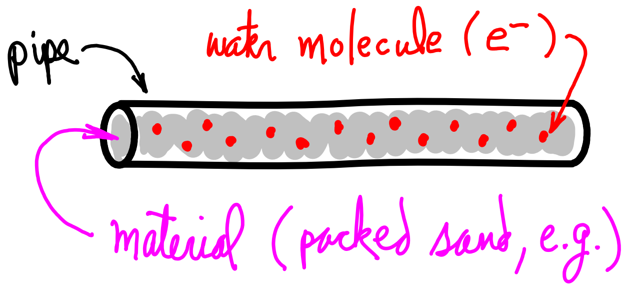
E X distance = Voltage

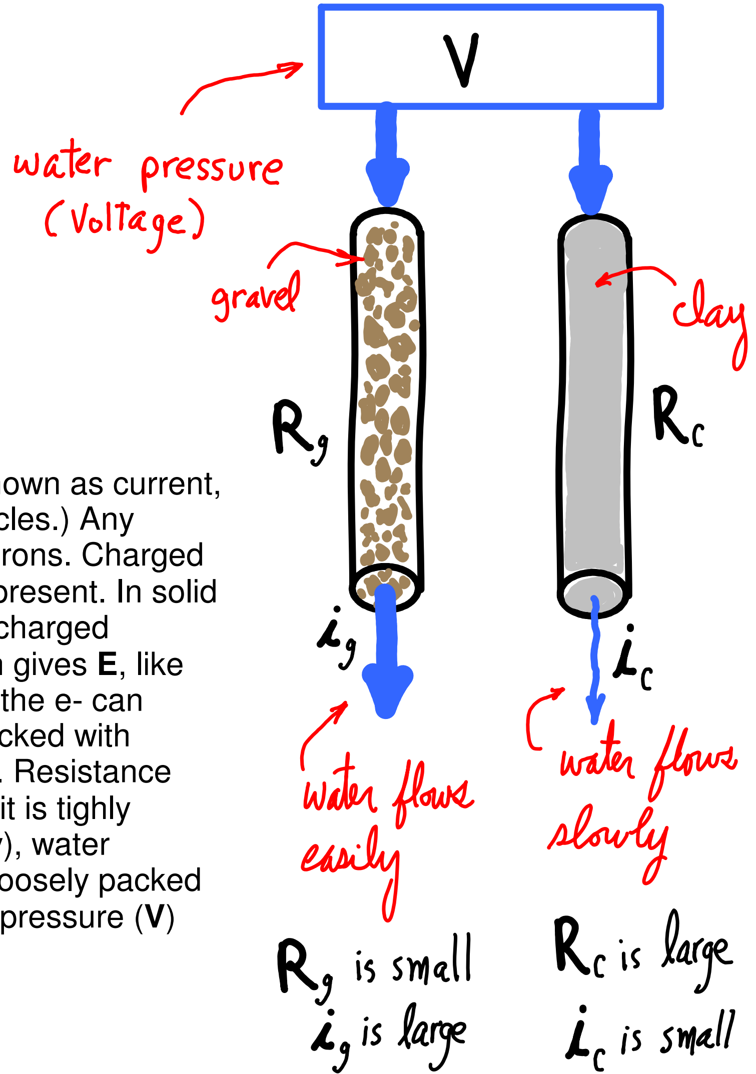


High Current ==> Many e- Moving
High Voltage ==> Large E Field
Large E ==> Fast Acceleration
High Voltage + High Current ==> Lots of Heat

$$\text{Heat} = \frac{\text{energy}}{\text{sec}} = \text{Power} = I \cdot V$$

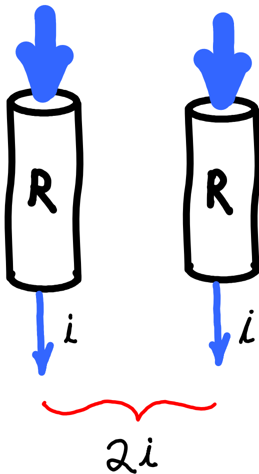
Basic Electricity, The Water Analogy





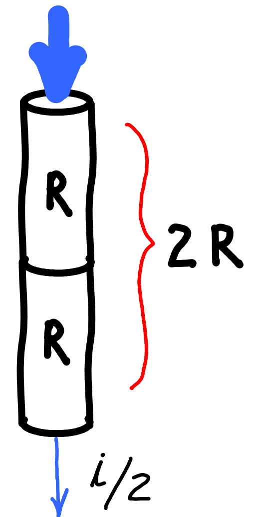
Conductors

Conduction is the movement of electrons (**e-**), also known as current, i. (Conduction can also be by positively charged particles.) Any material conducts, if we pull hard enough on the electrons. Charged things, such as e-, move when an electric field (**E**) is present. In solid materials, the nucleus of an atom contains positively charged protons (**p+**). Protons and e- attract each other, which gives **E**, like gravity. In solid material the p+ are fixed in place, but the e- can move. We can think of the solid material as a pipe packed with something; e.g., sand, and the e- as water molecules. Resistance (**R**) is how tightly packed the material in the pipe is: if it is tighly packed (the material is very fine material such as clay), water molecules have a hard time making it through; if it is loosely packed (large gravel), water drains through easily. The water pressure (**V**) and **R** together determine i.



Parallel circuit
Suppose we have two identical pipes side by side, and they both have resistance, **R**, and each has current, i. The current through both is twice the current through one, $2i$.

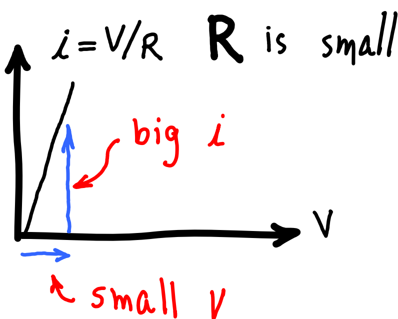
Series circuit
If we connect them end to end instead, we might expect the total resistance to be **R+R**, and the current to be $(1/2)\ i$.

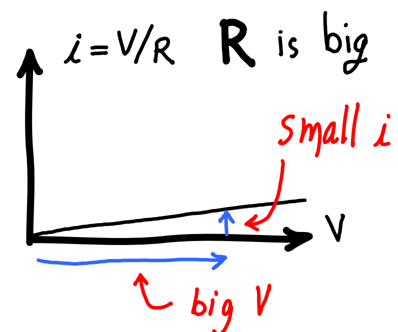


Ohm's Law devices
Different materials and devices have different relationships between i, **R**, and **V**.If the relationship can be expressed as,

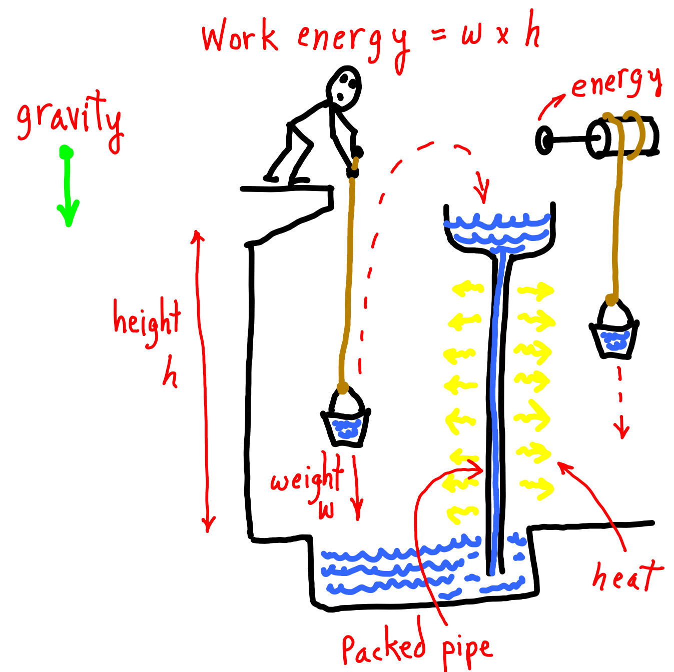
$$i\ \mathbf{R} = \mathbf{V}$$

then we call the device an Ohm's Law device. Of course, this is only an approximate model. If **R is very big**, the device **is a non-conducting insulator**. A **big pressure V** only gives **a little flow**. If **R is very small**, the device **is a conductor**. A very **small V** gives a **large flow**.



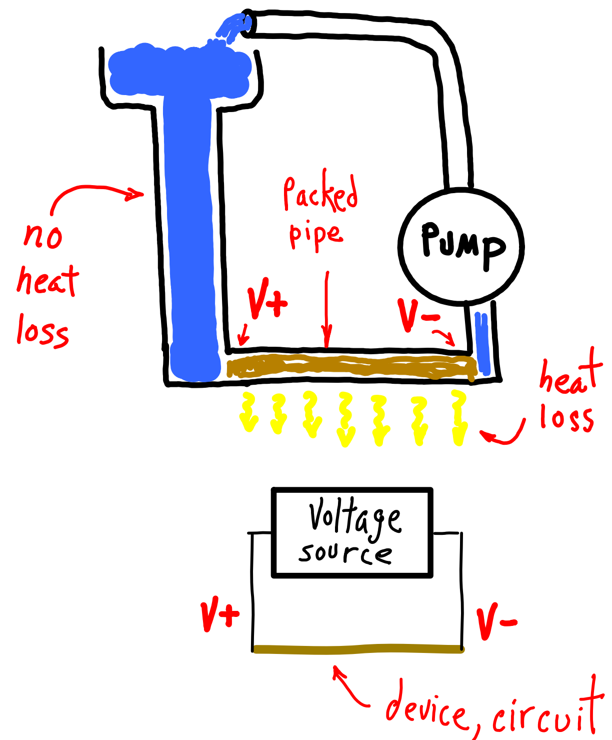


It takes work to get water pressure. Suppose we have a water tower. We pull the water up. **Pulling the weight w up the tower's height h** is the **work** we do, **w X h**.



We can get that same amount of energy back from the water in the tank. We can drop a bucket of water and use the pull to do work of some sort. That energy is used up in our packed pipe as the **water falls** through the pipe: it **heats the packing** as the water collides with the packing. The **heat escapes** by radiating away.

The downward force on the water is caused by the gravity **field E** acting on the water's mass: the **weight** of the water is **E X mass**. On the moon, the same mass of water weighs less because the moon's gravity field pulls less than earth's. You can jump high easily on the moon, for instance. **Smaller E** would mean it takes **less energy** to move the water: **less pressure** in the pipe, and **less flow**, and **less heat**.



Our model of an electrical **voltage source** is a tower and **very large pipe without packing**. It supplies water that flows through our packed pipe, and an **energetic process pumps** water back up. The pressure at the pump inlet is **V-** and the pressure at the tank end is **V+**. The **pressure difference V** drives water through the **packed pipe**.

Voltage Source = Pump + Tank + Big Pipe

Device/Circuit = Packed Pipe

The **energy lost** in the packed pipe by the water that flows through it is the same it took to pump the water into the tank:

energy = weight X h
weight = mass X **g** (**g** is gravitational acceleration)
mass = volume X density (let density = 1)
volume = Area X **h**

energy = (Area X h X 1) X **g** X **h**

Here, we are assuming the volume of water that flowed is equal to the volume of the big pipe whose cross sectional area is Area. Because the big pipe is h tall, its volume is Area X h. Suppose we want to see **how much energy** an amount of water of **mass m** loses. We first find the energy lost per unit mass by dividing the above by the mass (Area X h X 1):

energy-per-unit-mass = **g** X **h**

The **energy from mass m** is then:

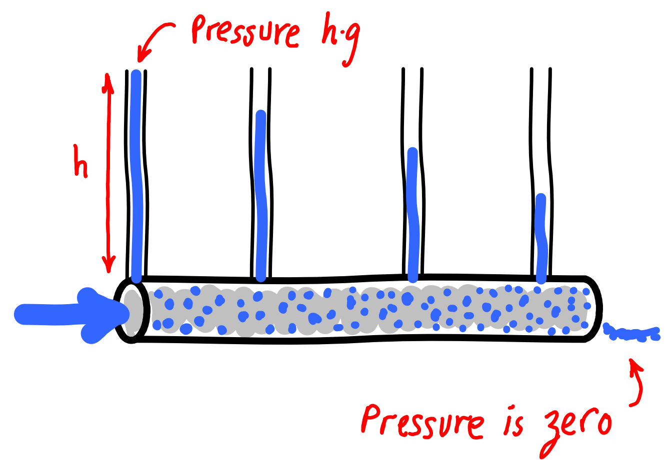
energy-m = **m** X **g** X **h**

Define **V** (short for voltage-across-the-device) as (**g** X **h**):

V = (**g** X **h**)

The energy for mass **m** is then,

energy-m = **m** X **V**



A packed pipe with water flowing through it has more pressure on the inlet side than the outlet side. (If it were the other way around, the flow would go backward.) The pressure drops along the pipe. At the inlet side, the pressure is just the total weight of the water in the big pipe pressing down divided by its area:

$$\textbf{pressure-h} = (\text{Area} \times h \times 1) \times g \,/\, \text{Area}$$

$$= g \times h$$

$$= V$$

So, the voltage **V is** the same as the **water pressure** supplied by the source. **Exit pressure is zero** because the **pump is pulling** the water from that end of the pipe.

Suppose **k** units of water flow per second. The **power loss in the device** is,

$$\textbf{Power} = \textbf{energy-per-unit-mass} \times (\mathbf{k}/\text{sec})$$

$$= \mathbf{V} \times (\mathbf{k}/\text{sec})$$

Current is i and is equal to (**k**/sec):

$$\textbf{Power} = \mathbf{V} \times i$$

Electrons and water molecules are equivalent. They just differ in their respective fields (**E** and **g**) and the properties those fields affect (**charge** and **mass**). **Power loss is heat** (mostly). Note that we have used **E** and **g** interchangeably, and applied electrical terminology to water.

Suppose our packed pipes can be modeled by Ohm's Law. **Power loss** is then **proportional to the square** of **V**. It can also be expressed as proportional to the square of i. (Both are shown at right.)

$$\text{power} = iV \qquad V = iR$$
$$= i(iR) = i^2R$$

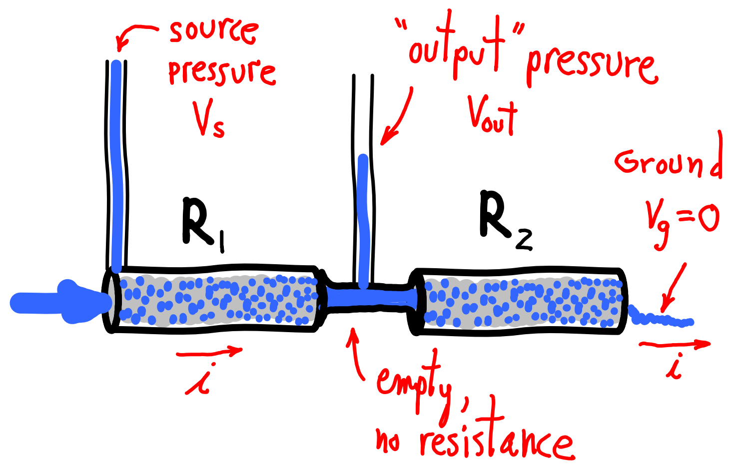
$$\text{power} = iV \qquad i = V/R$$
$$= (V/R)V = V^2/R$$

Note for unchanging **V**, that as resistance **R goes to 0**, the **power loss goes to infinity**. This is a short circuit. Before power loss actually goes to infinity, the heat will melt or vaporize the device. Of course, as **R goes to infinity**, nothing will flow, and **no power is lost**.

$$\lim_{R \to 0} \text{power} = \lim_{R \to 0} V^2/R$$

Voltage Divider

At right are two devices connected in series: Between them is a section of empty pipe whose resistance is relatively close to zero (wire).



The pressure (voltage) across **device**$_1$ is the source voltage $\mathbf{V}_s$ minus the "output" voltage $\mathbf{V}_{out}$. The current exiting **device**$_2$ has pressure $\mathbf{V}_g = 0$. So, the voltage across **device**$_2$ is $\mathbf{V}_{out}$.

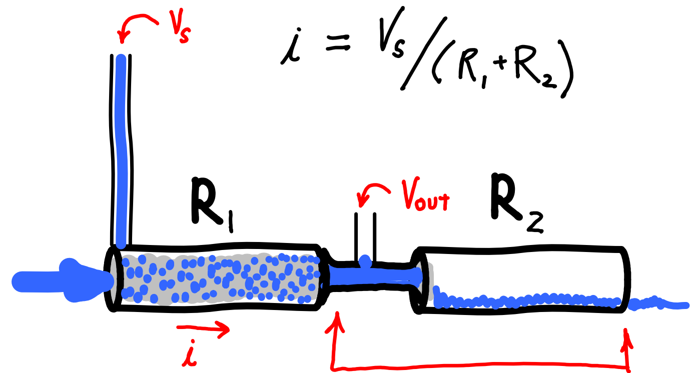
The "output" of this system $\mathbf{V}_{out}$ depends on the two resistances, $\mathbf{R}_1$ and $\mathbf{R}_2$. The current i is the same through both resistors.

total voltage difference

is $(V_s - V_g) = V_s$

current is

Suppose $\mathbf{R}_2$ **is nearly 0** (a resistance-less wire). The total voltage difference over both resistors is $(\mathbf{V}_s - \mathbf{V}_g) = \mathbf{V}_s$. The output voltage is the voltage difference across $\mathbf{R}_2$. Because $\mathbf{R}_2$ is about 0 (i.e., it has no packing, water passes through easily) 0 volts is almost all that is needed to move water through it. That is to say, no water pressure can build up on the inlet side of $\mathbf{R}_2$ because when it starts to build up, water flows through before any pressure can build up. The current i is completely determined by $\mathbf{R}_1$.



$V_{out} - V_g = V_{out}$

$i = V_s/(R_1+R_2)$

$= V_s/R_1$

$V_{out} = i R_2$

$\approx i \cdot 0 = 0$

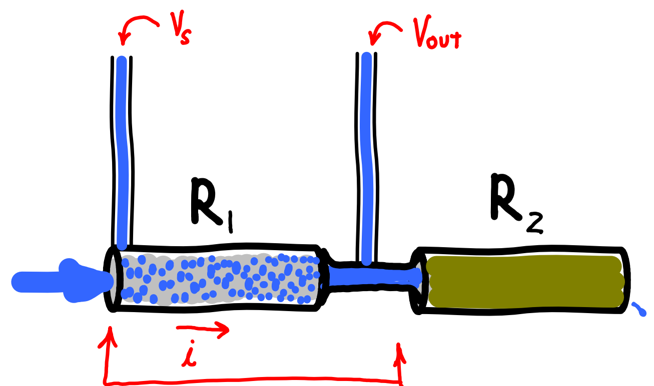
In the other extreme, suppose $\mathbf{R}_2$ is **nearly infinite** (an open circuit or switch). No matter how much pressure there is, almost no current flows.

$$i = (\mathbf{V}_s - \mathbf{V}_g) / (\mathbf{R}_1 + \mathbf{R}_2) = \mathbf{V}_s / (\mathbf{R}_1 + \mathbf{R}_2) \sim \mathbf{0}$$

Pressure will build up as flow exits $\mathbf{R}_1$ and gets stopped by $\mathbf{R}_2$. Water would flow through $\mathbf{R}_2$ if the pressure at one end were different from the pressure at the other end. But, no current flows. So, the pressure at both ends must be the same. That is, $\mathbf{V}_{out}$ is the same as $\mathbf{V}_s$.

$$(\mathbf{V}_s - \mathbf{V}_{out}) = i\,\mathbf{R}_1 \sim 0\,\mathbf{R}_1 = \mathbf{0}$$

$$\mathbf{V}_s \sim \mathbf{V}_{out}$$



$i = (V_s - V_g)/(R_1+R_2)$

$\approx V_s/R_2 \approx 0$

$(V_s - V_{out}) = i R_1$

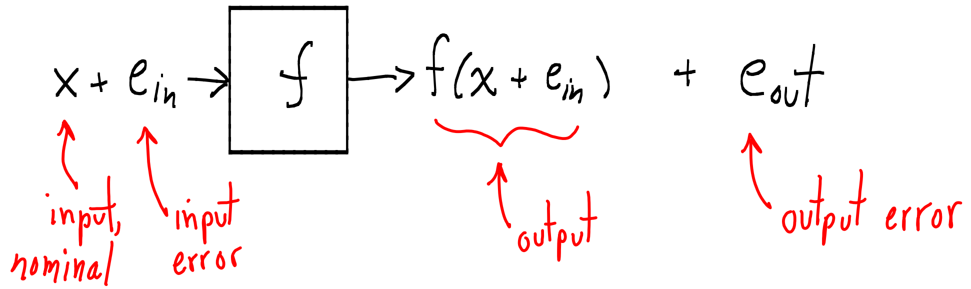
$V_s - V_{out} \approx 0 \cdot R_1 = 0$

$V_s \approx V_{out}$

We need Signal-Restoring, Non-Linear Logic. Ohm's Law devices are LINEAR.

Suppose we had only linear devices (or something very nearly linear), then signal output has errors proportional to input errors.

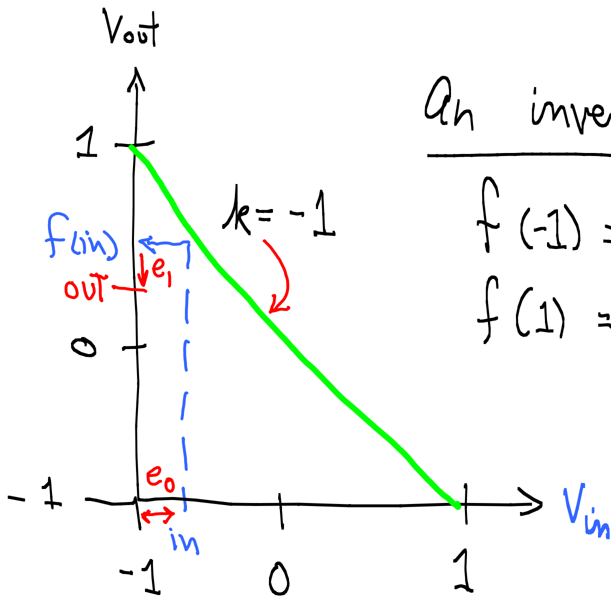
Errors/noise :



Linearity:

$$f(v+e_{in}) = k(v+e_{in}) + e_{out} = kv + ke_{in} + e_{out}$$

($ke_{in} + e_{out}$: Total error)



An inverter ckt.

$$f(-1) = 1$$
$$f(1) = -1$$

in → (inverter) → f(in)

Suppose we connect 2 in series

IN → f_1 → f_2 → $= f_2(f_1(in))$

(v is nominal signal: ± 1)

$$f_1(in) = f_1(v+e) = kv + ke_0 + e_1$$
$$= (-1)(-1) + (-1)e_0 + e_1$$
$$= 1 - e_0 + e_1$$

$$f_2(f_1(in)) = f_2(1 - e_0 + e_1)$$
$$= -1 + e_0 - e_1 + e_2$$

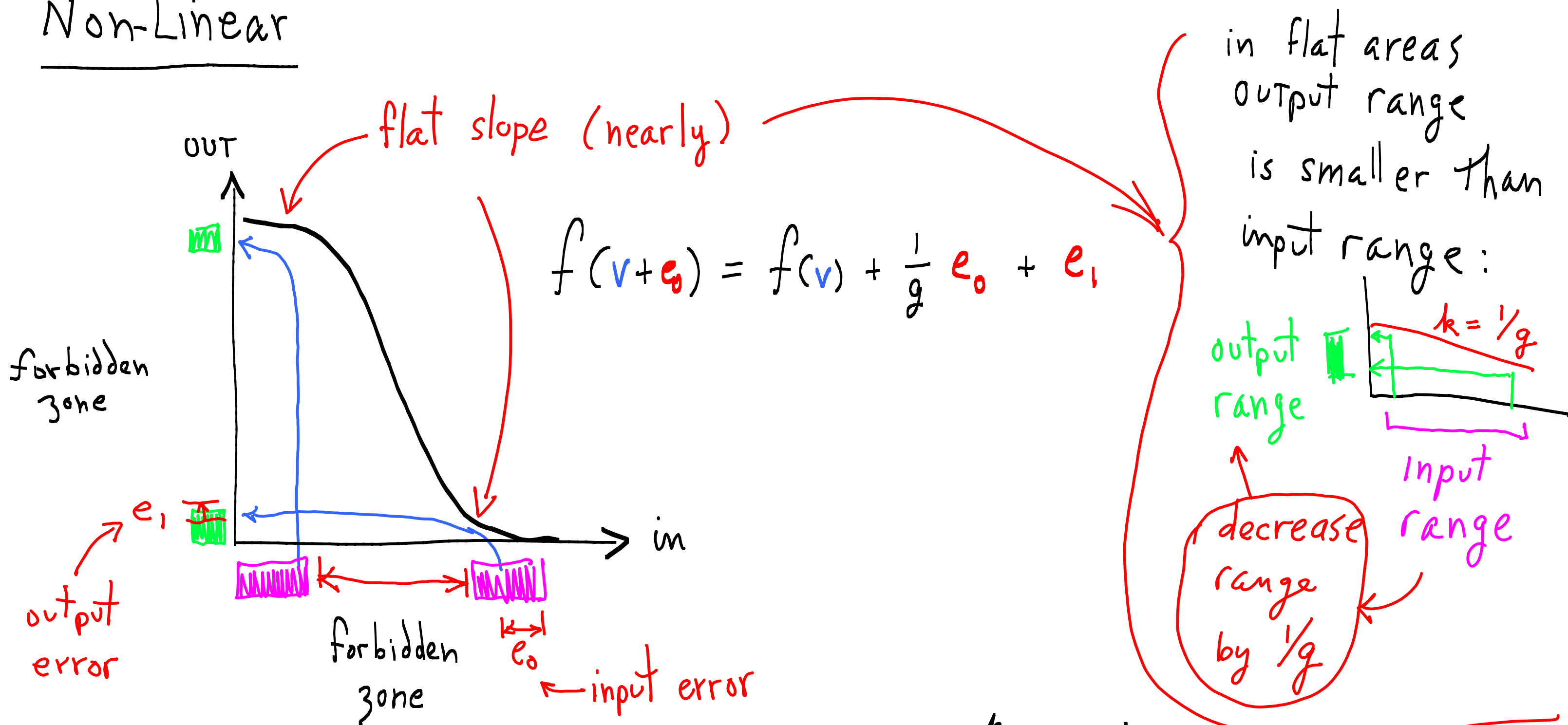
k stages $\Rightarrow$ $1 + \sum_{i=0}^{k} (-1)^i e_i$

The errors include signs = random walk with random size steps.

Errors independently random w/ average 0 ==> variance increases w/ k.
Total error grows w/o bound!

Take random step (either in the -1 or +1 direction).
How far from 0 can you expect to be after k steps? About k^1/2 away.
With probability 0 you will be at 0, and error gets unboundedly large.

We must Reduce error at each stage ==> exponentially decreasing effect in later stages.



after k stages: $= 1 + \sum_{i=0}^{k} \left(\frac{1}{g}\right)^i e_i$

Randomly ±

Converges!

If g is large enough (flat areas of curve are flat enough),

and

if output error size is not too big,

Then

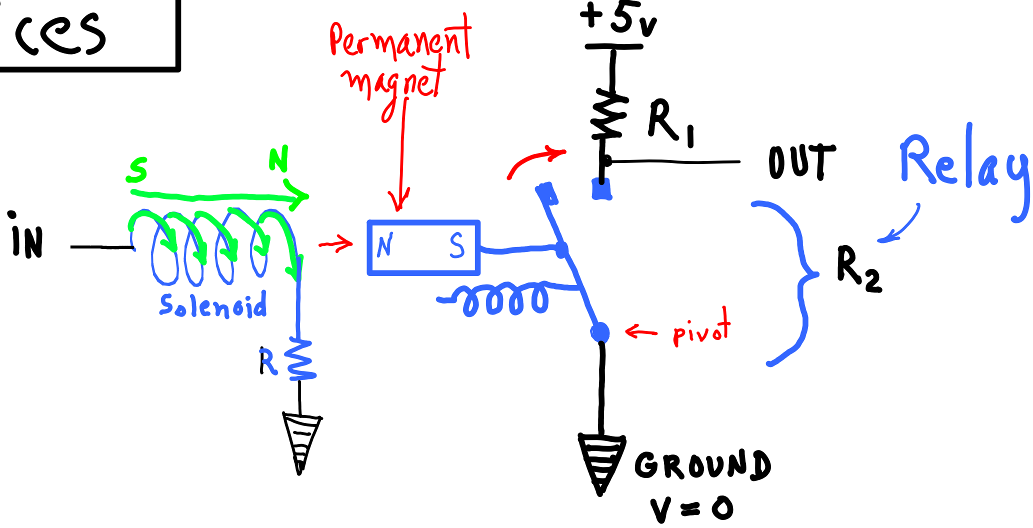
output after k stages never hits FORBIDDEN ZONE.

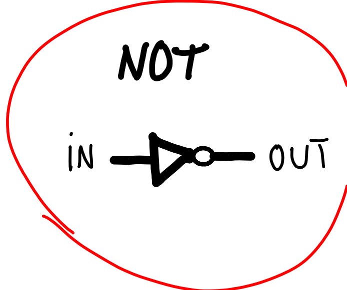
So, if we plan to have a circuit with long device chains, we must have non-linear devices w/ suitable response curves.

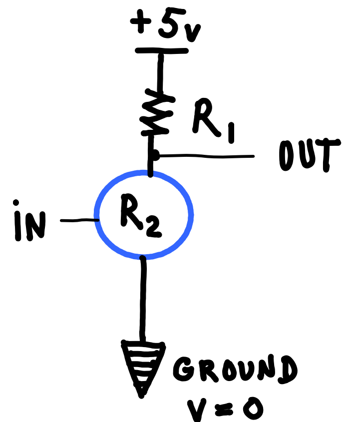
Do we plan to have long chains? YES:

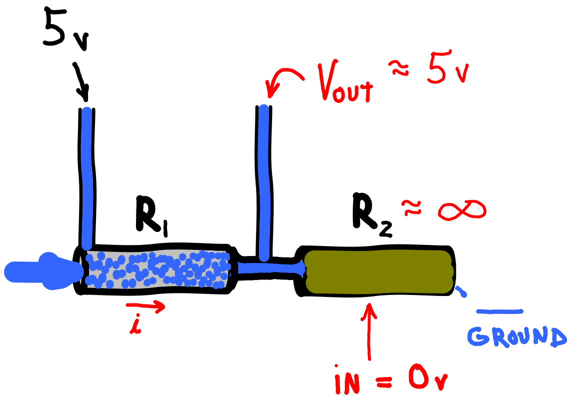
(1) feedback in system,
(2) chained data operations: D1 ==> D2 ==> D3 ==> D4 ...
(3) 1 Billion devices per cpu

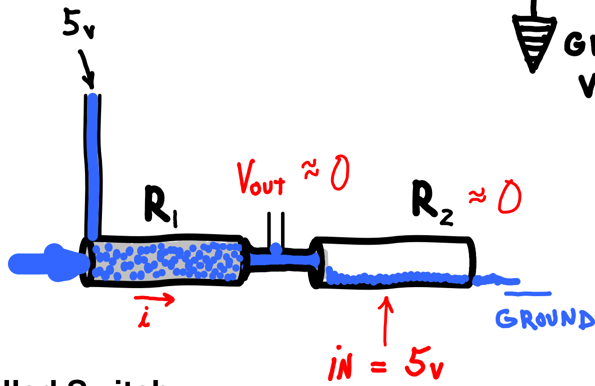
Devices





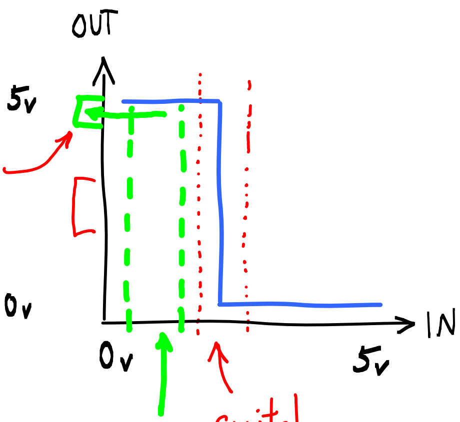






Voltage Controlled Switch

far from forbidden zone



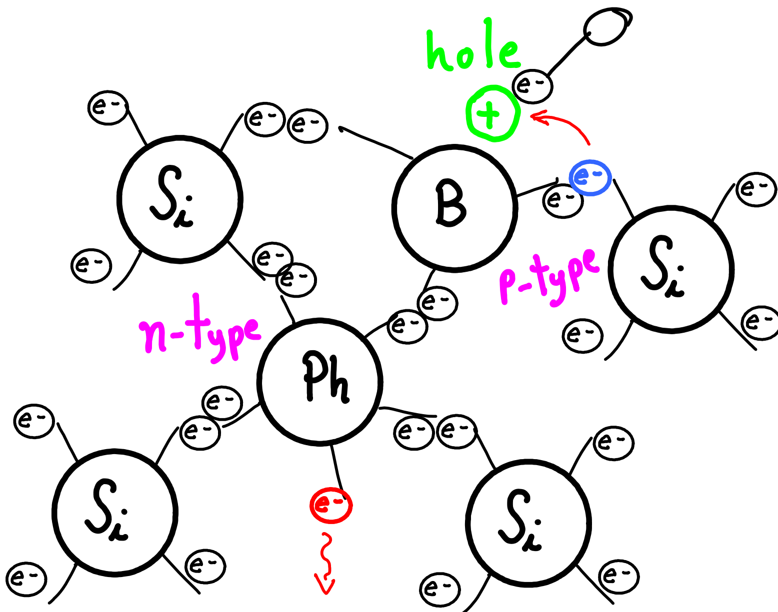
switch closes, varies for each device ⇒ FORBIDDEN ZONE

--- Non-linear response

--- Any voltage in allowed range will yield "clean" output.

--- Easy to prevent forbidden input voltage.

Solid State devices – Semiconductors



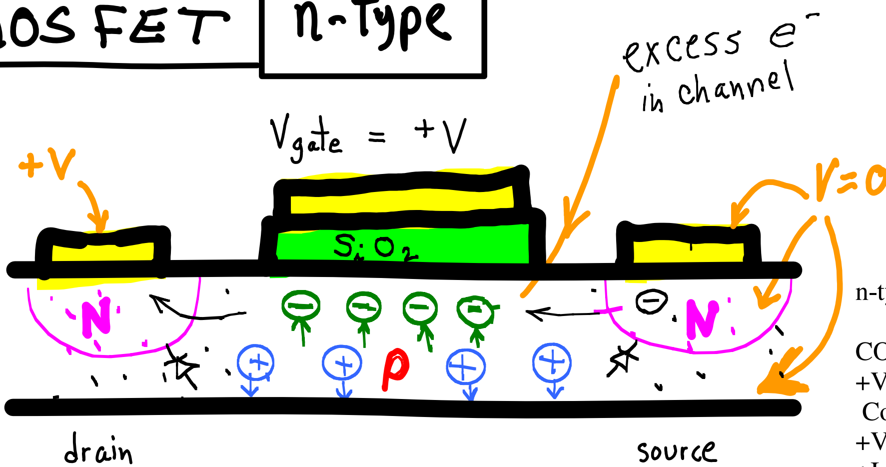
Phosphorous impurity:
e- leaves easily, becomes "hot"
conduction-band e- .

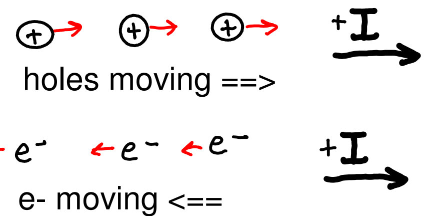
Boron impurity:
"cold" valence-band e- arrives, leaves behind +valence "hole" which moves.

Holes and e- move in opposite directions, but current direction is same.

Easy e- flow from n-type to p-type, but reverse flow hard: "cold" valence-band e- need too much energy to become conduction band e-.

MOSFET n-type

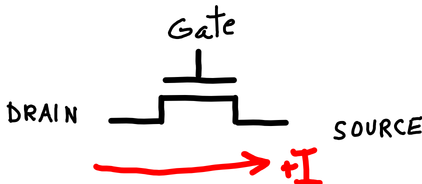




n-type MOSFET (n-transistor)

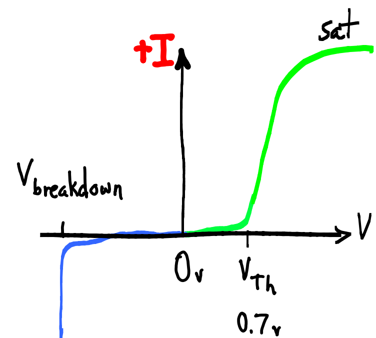
CONDUCTING (Vgate = +V, R-drain-to-source ~= 0):
+V on gate drives holes away from P-type channel.
Conduction-band e- move from source N-type well.
+V on drain pulls conduction-band e- off.
+I current flows left-to-right.

NOT-CONDUCTING (Vgate = 0, R-drain-source = BIG)
Vgate = 0, holes populate channel.
Source N-well e- drop into valence band in channel.
+V at drain cannot pull valence-band e- from P-type to N-type.

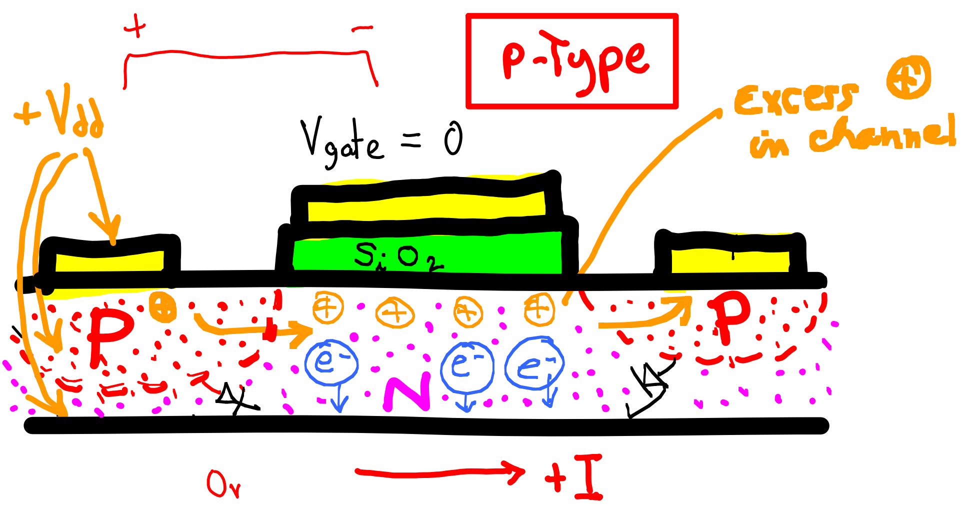


n-type transistor

Vgate	R	
0	∞	not conducting
+V	0	conducting



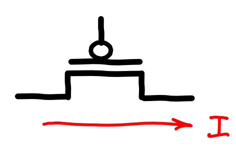
Just what we want: nice non-linear switch.



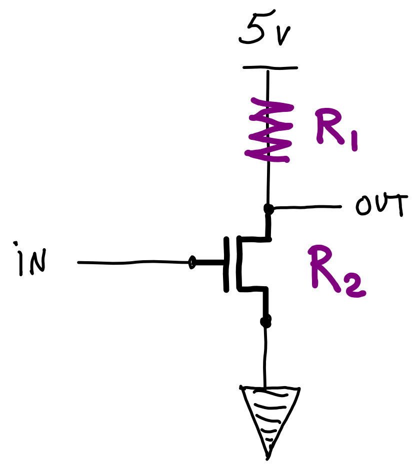
P-type (n-channel) transistor

Vgate = 0:
Vgate = -Vdd wrt to base,
pushes e- away from channel, leaves excess holes, current flows.
R = 0, conducting.

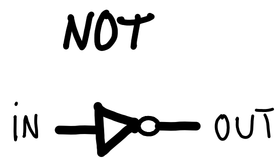
Vgate = Vdd:
Vgate = 0 wrt to base
channel is neutral
only random thermal e- available for current flow.
R = infinity, not conducting

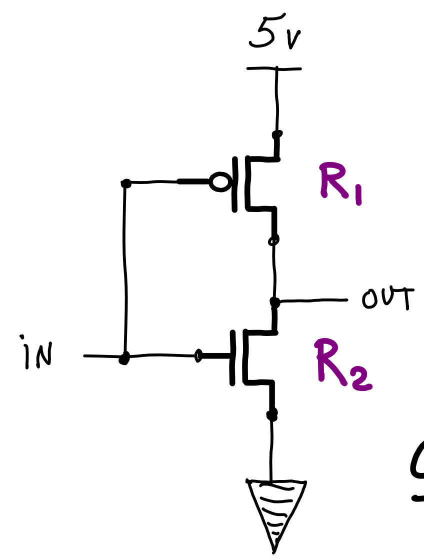


Vgate	R
0	0 conducting
+V	∞ not conducting



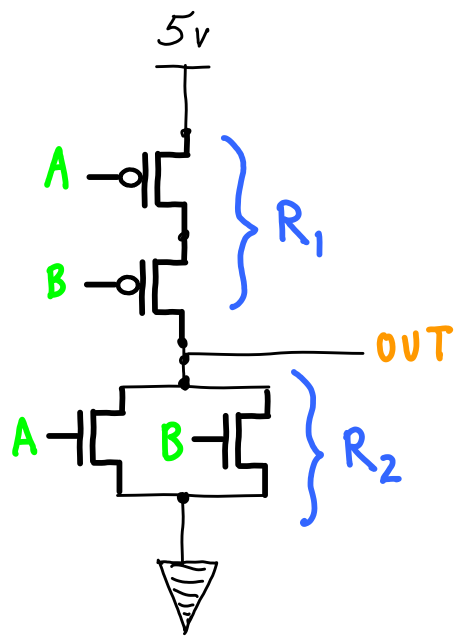
NOT





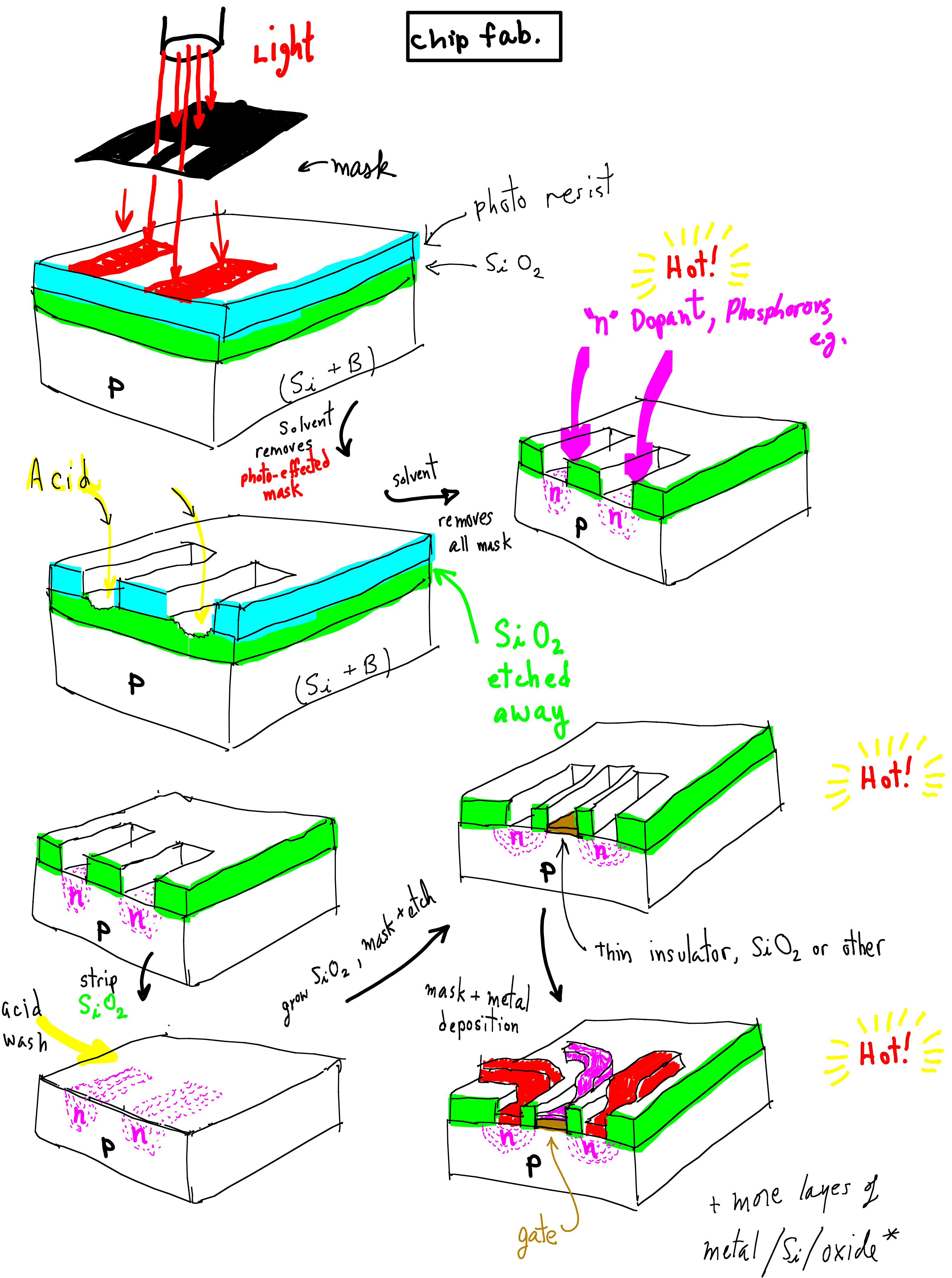
$P = iV$

CMOS

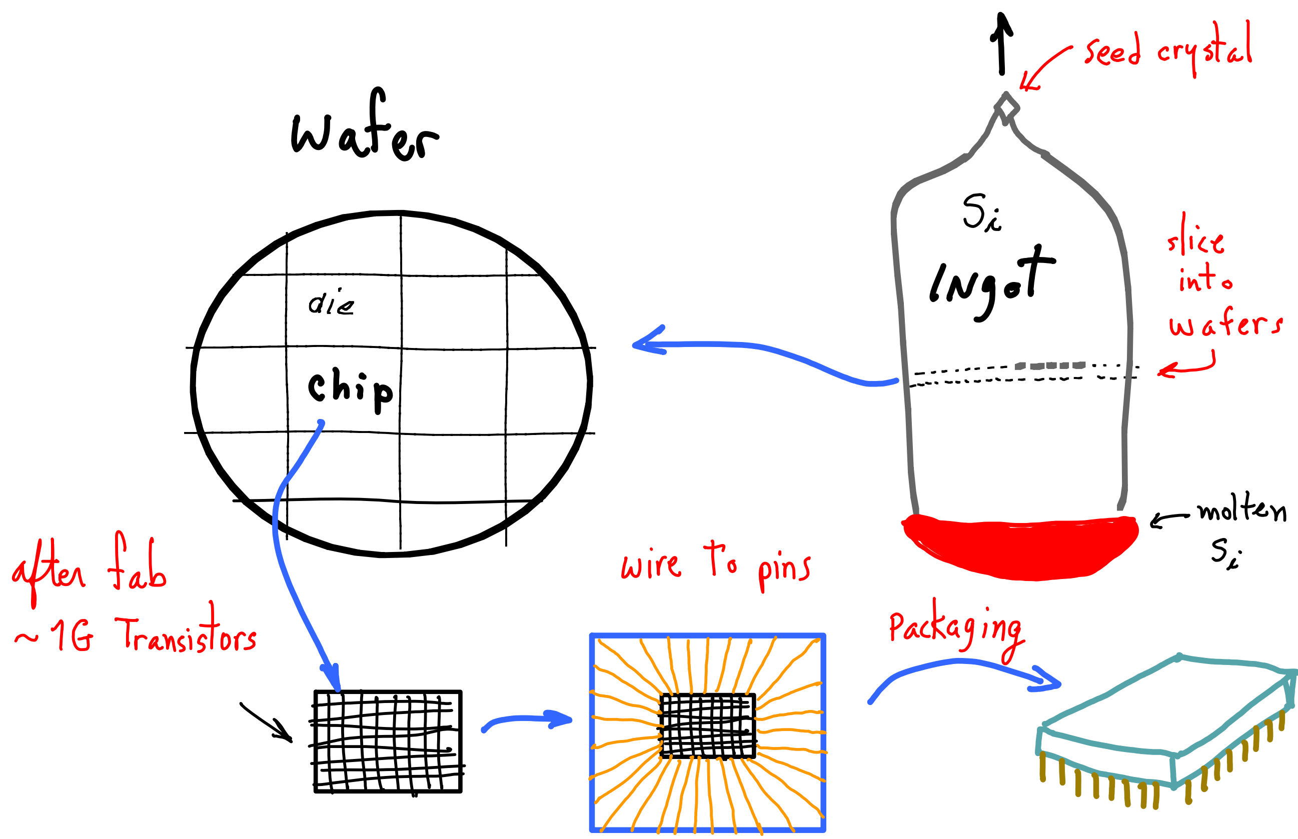


A	B	OUT
0	0	1
0	1	0
1	0	0
1	1	0

$OUT = \overline{A+B}$

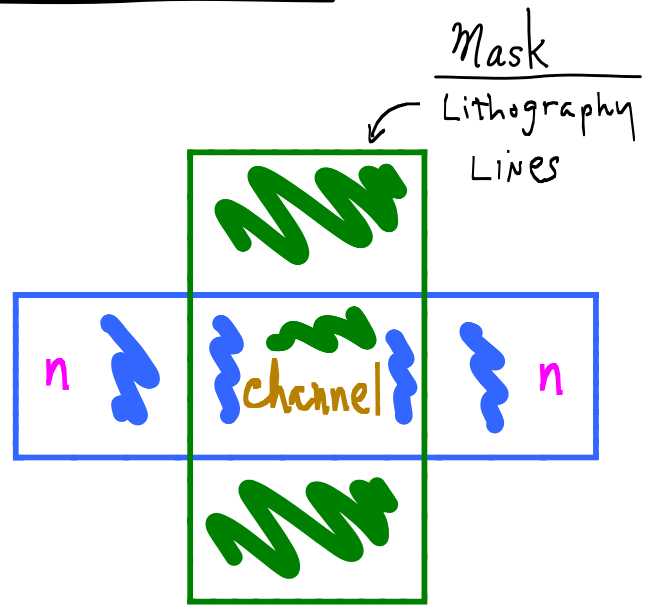

chip fab.
Light
mask
photo resist
SiO_2
P
$(Si + B)$
Solvent removes
photo-effected mask
Hot!
"n" Dopant, Phosphorous, e.g.
Acid
solvent
removes all mask
n
n
P
P
$(Si + B)$
SiO_2 etched away
Hot!
n
n
P
n
n
P
strip SiO_2
grow SiO_2, mask + etch
thin insulator, SiO_2 or other
mask + metal deposition
acid wash
n
n
P
Hot!
n
n
P
gate
+ more layers of metal/Si/oxide*

Lithography



Line Sizes

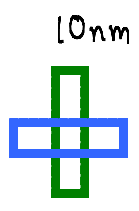
Mask
Lithography Lines



Better Process
more expenisve equipment
+$

lines shrink

10nm



Faster switching
→ faster clock

more transistors
→ more function
→ more usage
→ more sales
→ same price

smaller features → More defects, lower die yield, but smaller die → Moore's Law

Copyright © The McGraw-Hill Companies, Inc. Permission required for reproduction or display.

LC-3 Overview: Memory and Registers

Memory

- address space: 2^{16} locations (16-bit addresses)
- addressability: 16 bits

Registers

- temporary storage, accessed in a single machine cycle
  - accessing memory generally takes longer than a single cycle
- eight general-purpose registers: R0 - R7
  - each 16 bits wide
  - how many bits to uniquely identify a register?
- other registers
  - not directly addressable, but used by (and affected by) instructions
  - PC (program counter), condition codes (PSR)

LC-3 Overview: Instruction Set

Opcodes

- 15 opcodes
- *Operate* instructions: ADD, AND, NOT
- *Data movement* instructions: LD, LDI, LDR, LEA, ST, STR, STI
- *Control* instructions: BR, JSR/JSRR, JMP, RTI, TRAP
- some opcodes set/clear *condition codes*, based on result:
  - N = negative, Z = zero, P = positive (> 0) → PSR.CC

Data Types

- 16-bit 2's complement integer

Addressing Modes

- How is the location of an operand specified?
- non-memory addresses: *immediate*, *register*
- memory addresses: *PC-relative*, *indirect*, *base+offset*

Assembly Language

foo.asm, an assembly language source code file (ASCII codes, created in any text editor)

```
               .orig x3000

               ld r1, six
               ld r2, number
               and r3,r3,#0

again          add r3,r3,r2
               add r1,r1,#-1
               brp again

               halt

number         .blkw 1
six            .fill x0006
msg            .string "abc"
               .end
```

Assembler lc3as

foo.obj : load object's BITS:

```
0011000000000000   <- header
0010001000000111   } calculated offset
0010010000000101
0101011011100000
0001011011000010
0001001001100001
0000001111111101
1111000000100101   <- Translation of "halt"
0000000000000000   BLKW 1
0000000000000110   .FILL x0006
0000000001100001   'a' x0061
0000000001100010   'b' x0062
0000000001100011   'c' x0063
0000000000000000   NUL x0000
```

opcode

Assembler (lc3as) **Directives** (to control the assembly process):
.orig: puts a load address into the .obj load-object file's header.
.end: tells assembler, this is the end of source code.
.blkw: tells assembler, create *n* blank words (all zeroes).
.fill: tells assembler, put these bits into a word.
.string: convert text to .FILL w/ one ascii code per word, NUL terminated.

The assembler produces machine code words:
--- ONE PER LINE expressing an LC3 instruction
--- ONE PER LINE where there is a .fill directive
--- n PER LINE where there is a .blkw directive
The assembler also calculates offsets for us using **symbols**. Symbols stand for memory addresses (starting for the .orig address). Offsets are calculated by subtraction. Symbols refer to the next instruction's location.

f.asm

```
        .orig x0200
main:
        ADD R1, R2, R3
        ADD R4, R5, var
foo:    .FILL x1234
var:    .FILL x012F
        .end
```

assemble
LC3as

f.obj

```
0000001000000000
0001001001000011
0001100101100001
0001001000110100
0000000100101111
```

203
- (201+1)
1

load

mem

```
0200:  0001001001000011
0201:  0001100101100001
0202:  0001001000110100
0203:  0000000100101111
```

← PC

NOT (Register)

ADD/AND (Register)

this zero means "register mode"

ADD/AND (Immediate)

this one means "immediate mode"

PROCESSOR BUS

PC

DR

SR2

REG_FILE

SR1

addrArith

IR

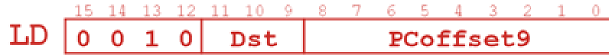
ALU

stackOps

MDR

MAR

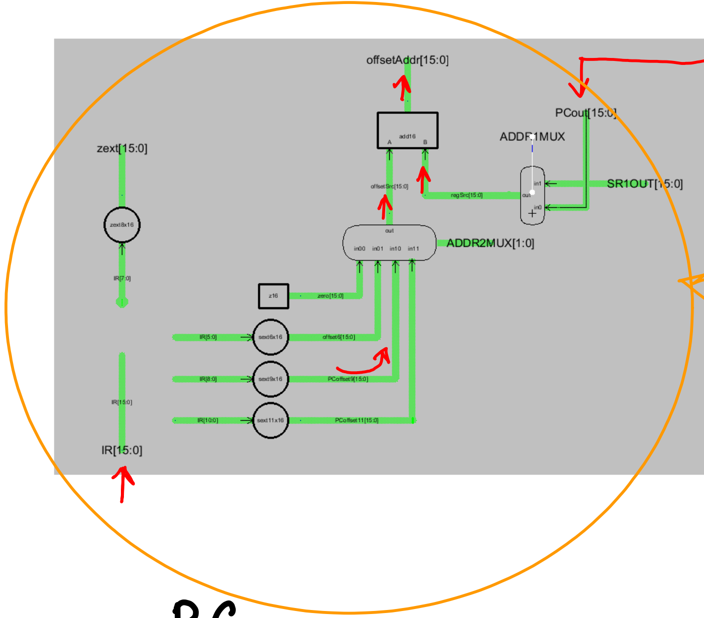
LD (PC-Relative)

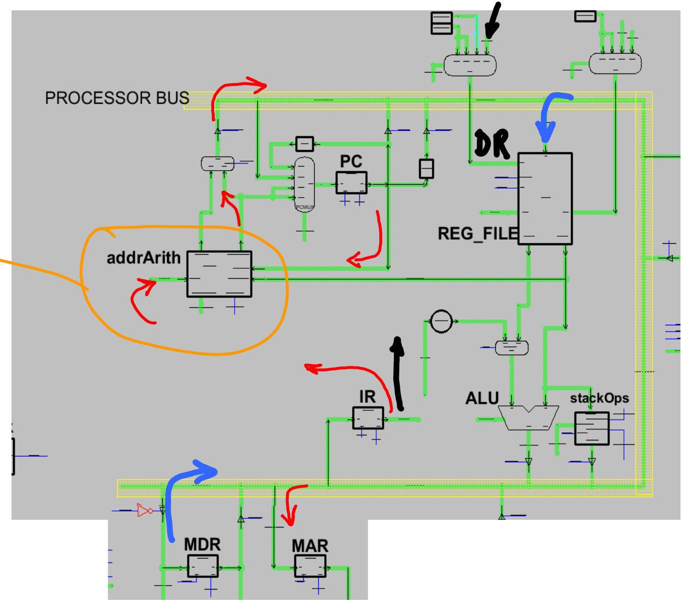


DR

ST (PC-Relative)

SR





PC

IR | DR | PCoffset9

Sext

+

MAR

Mem

023F

MDR

IN

023F

Reg File

foo.asm

main:	**LD** R3, var
	...
var:	.FILL x023F

0010 011 000000111 IR

ST

symbol	value
"main"	x0200
"var"	x0208

LC3as load

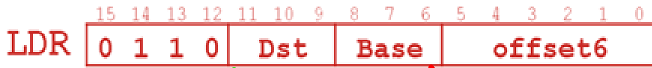
Mem

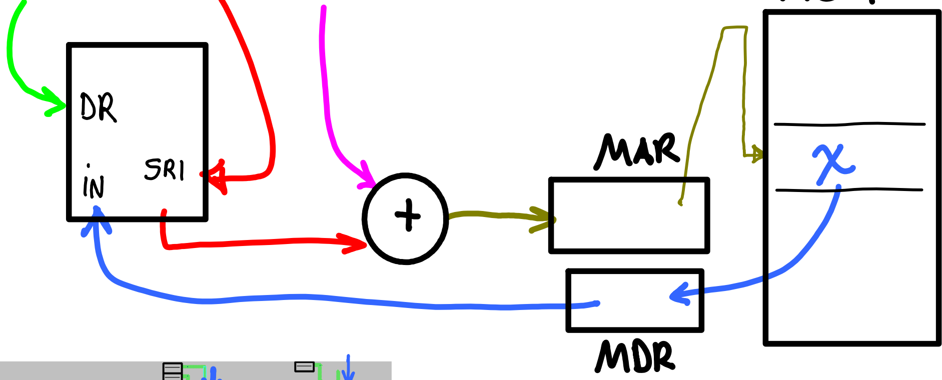
PC →

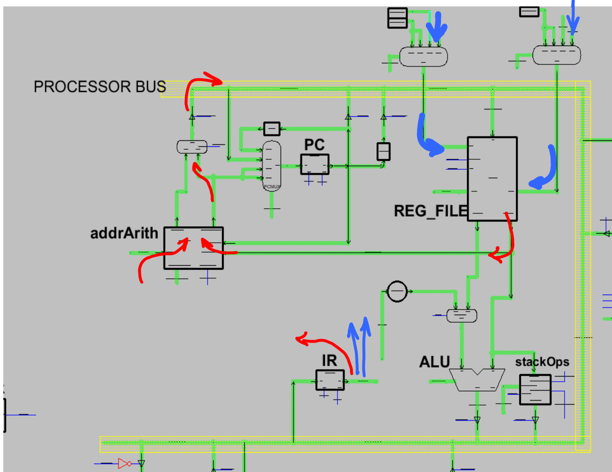
0200:	0010 011 000000111
...	...
0208:	0000 0010 0011 1111

208
\- 201
7

LDR (Base+Offset)

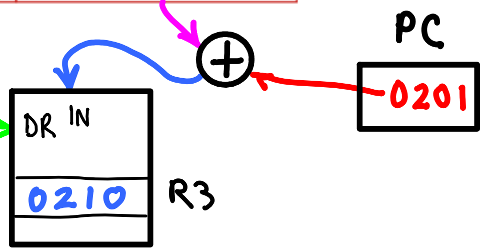







```
main:     LD R2, tablePTR
          LDR R1, R2, #2
                  ...
tablePTR:
           .FILL table
           ...
table:    .FILL x0000
          .FILL x0001
          .FILL x0002
```

"main"	x0200
"tablePTR"	x02F0
"table"	xFF00

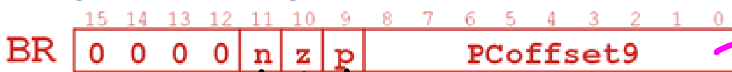


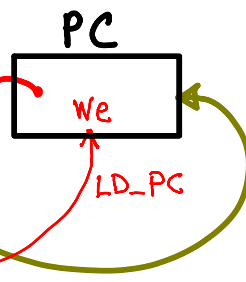
Source Code
main: LEA R3, array
...
array: .BLKW 100

Symbol Table
"main" x0200
"array" x0210

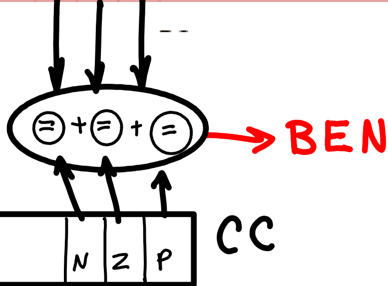
Memory
0200: 1110 011 000001111
... ...
0210: ????
0211: ????
... ...

210
\- 201
F





9-bit offset:
PC +/- 256



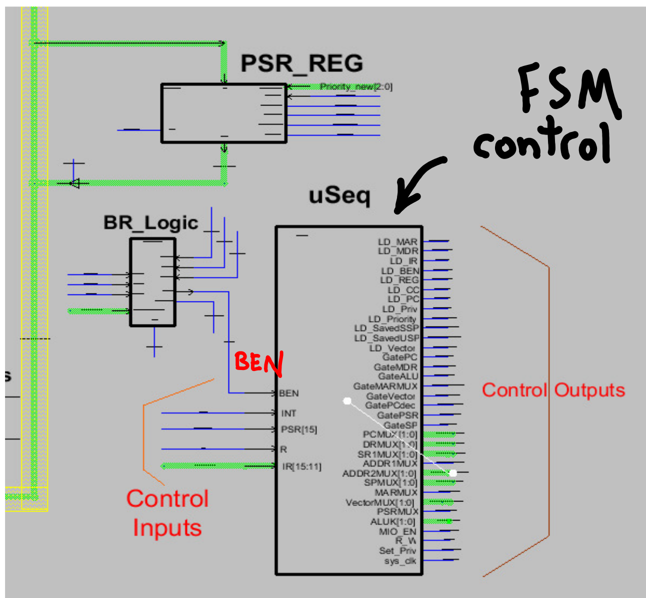
BEN

Source Code
main: ADD R0, R0, 1
BRp main

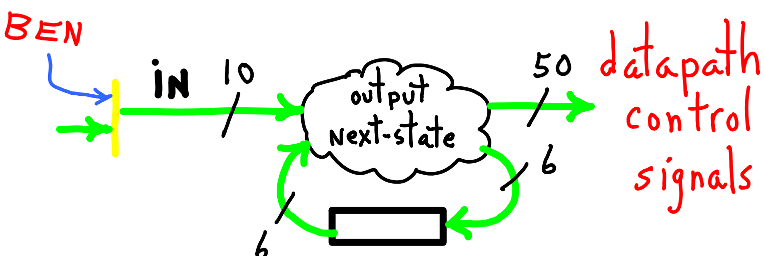
Symbol Table
"main" x0200

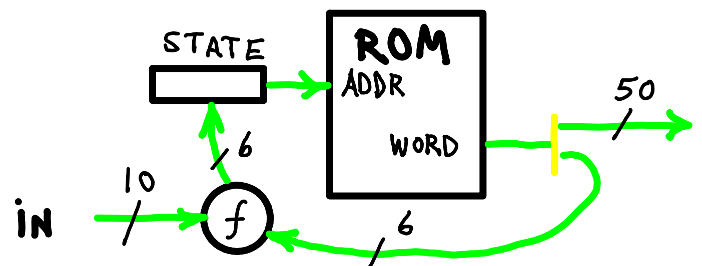
Memory
0200: 0001 000 000 0 00001
0201: 0000 001 111111110

200
\- 202
-2



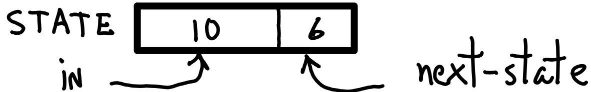
- only certain instructions set the codes (ADD, AND, NOT, LD, LDI, LDR, LEA)

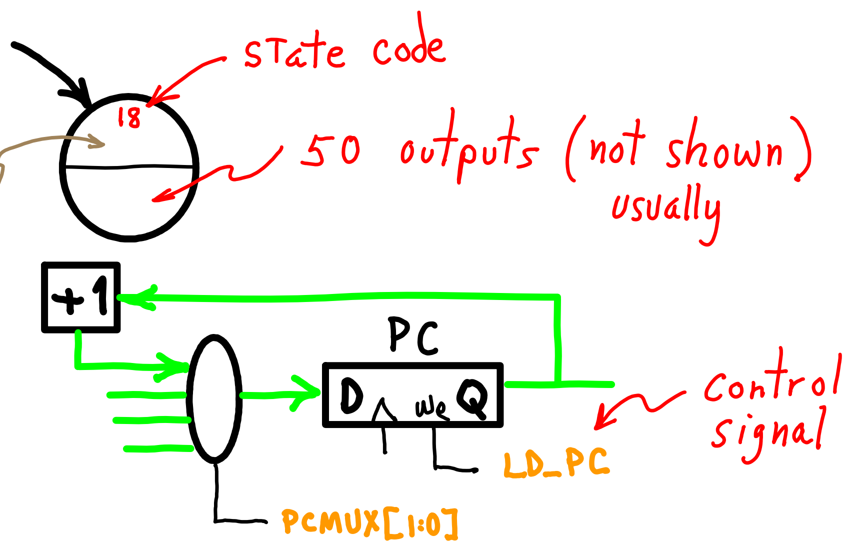




STATE DIAGRAM

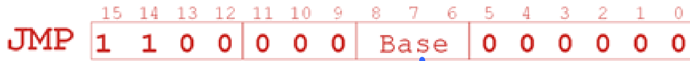
Moore FSM
--- Output not a function of input
--- Output determined by state only





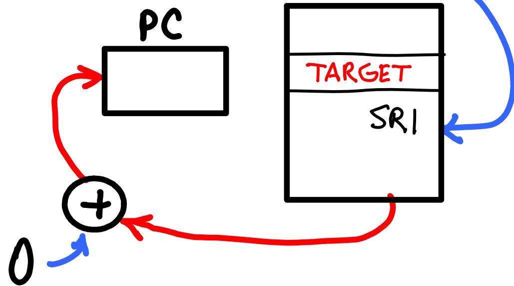
→ 64 word ROM

vs



Mealy
FSM

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JMP	1	1	0	0	0	0	0	Base			0	0	0	0	0	0



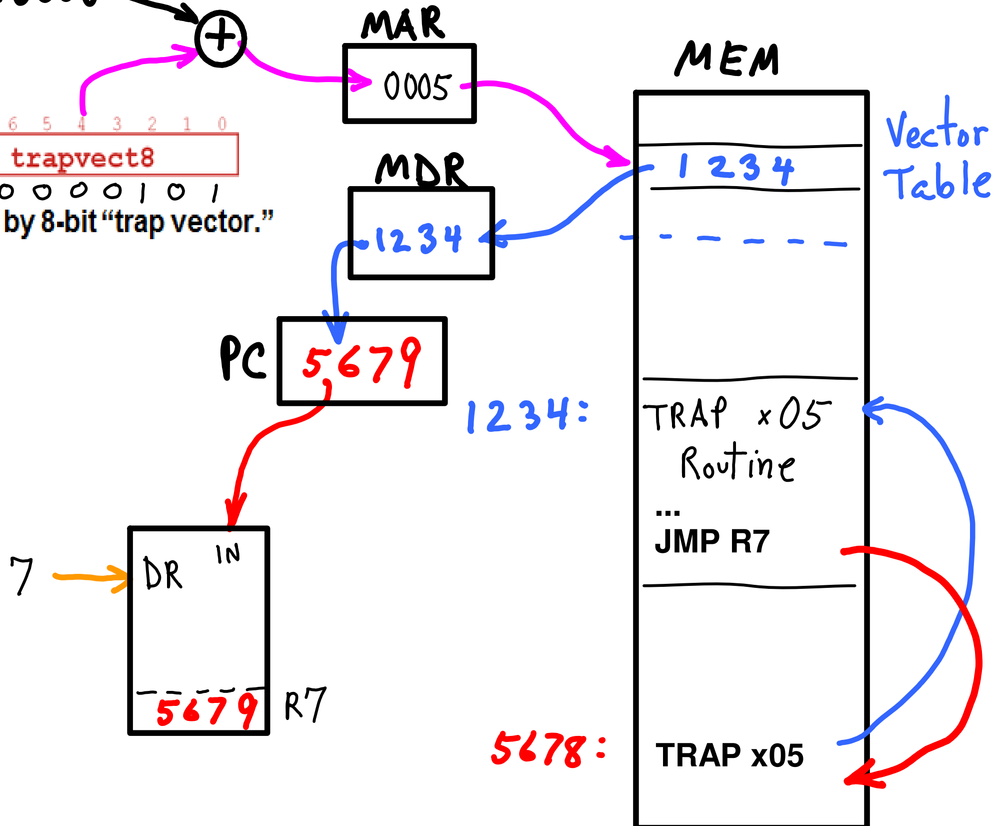
Source Code:

```
main:   LEA R7, next
        BRnzp foo
next:   ADD R0, R1, #11

...
foo:    ADD R0, R1, #10
        JMP R7
```

TRAP

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TRAP	1	1	1	1	0	0	0	0	trapvect8							
									0	0	0	0	0	1	0	1

Calls a service routine, identified by 8-bit "trap vector."



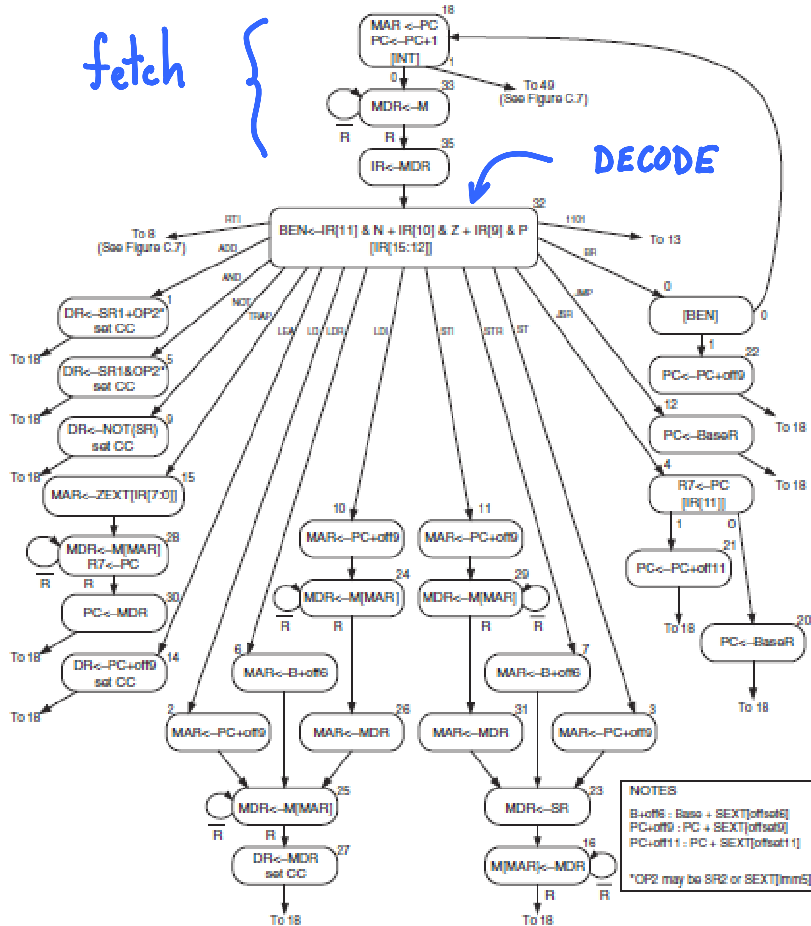


Figure C.2 A state machine for the LC-3

RISC

eliminate 5 opcodes

LD, LDI, ST, STI

JSR/JSRR

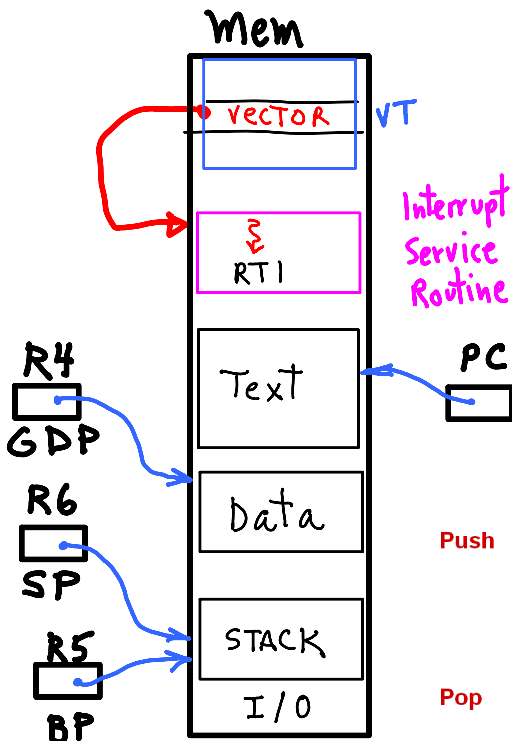
eliminate JMP?

change BR → BRR

eliminate TRAP?

change BR → BRR+

Interrupts, Exceptions



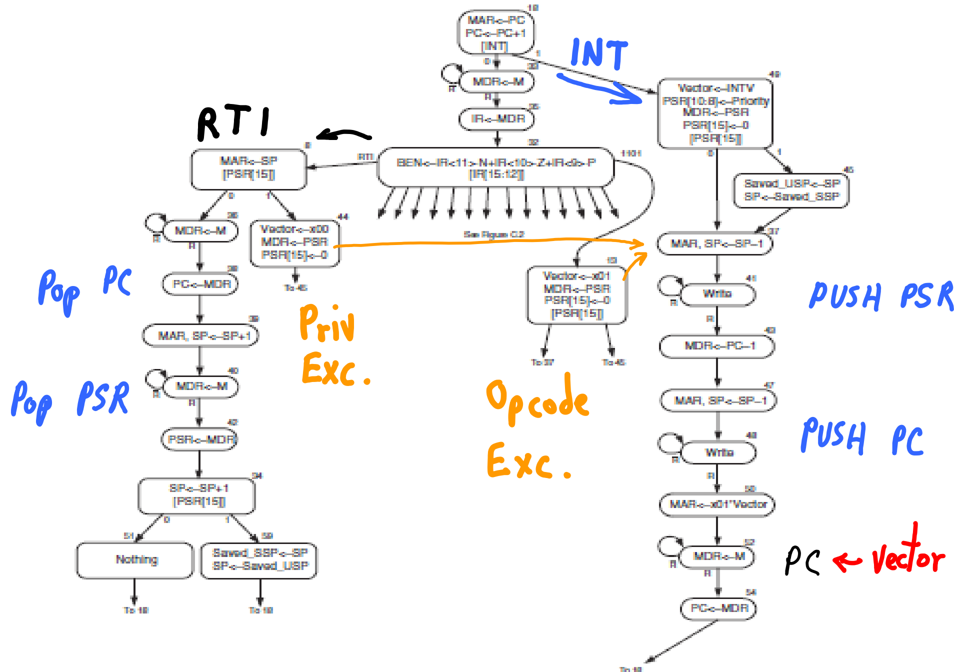


Figure C.7 LC-3 state machine showing interrupt control

Push

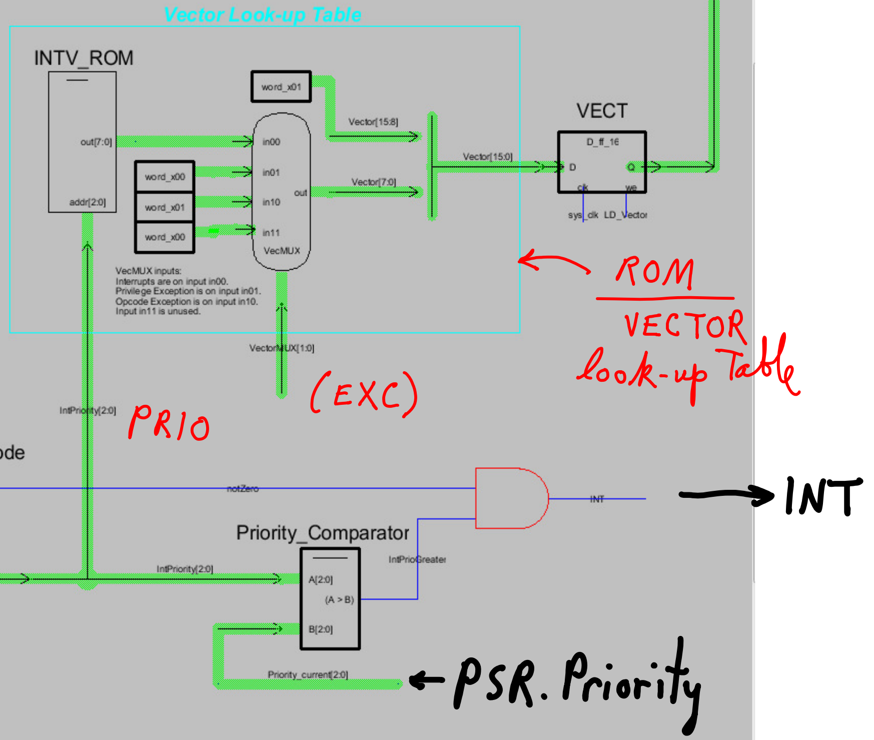
```
ADD  R6, R6, #-1   ; decrement stack ptr
STR  R0, R6, #0    ; store data (R0)
```

Pop

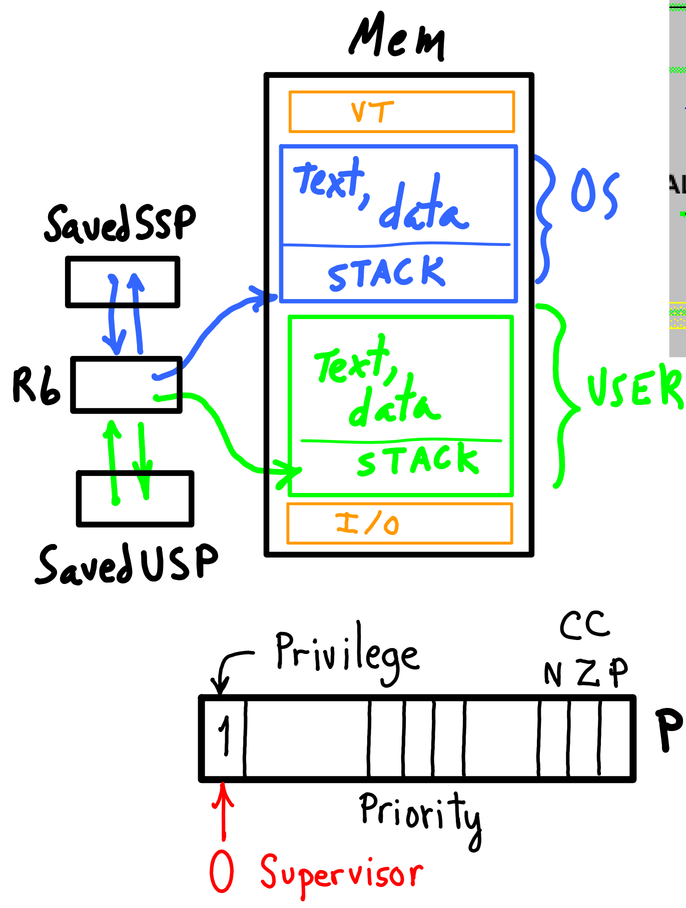
```
LDR  R0, R6, #0    ; load data from TOS
ADD  R6, R6, #1    ; decrement stack ptr
```

+1

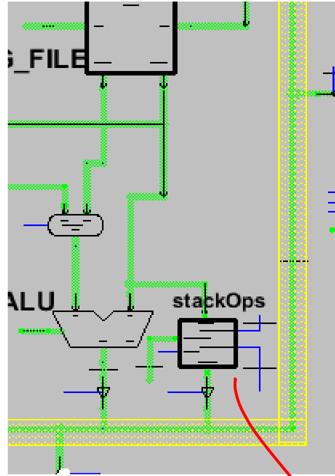
R7 - linkage } Hardware
R6 - SP } defined



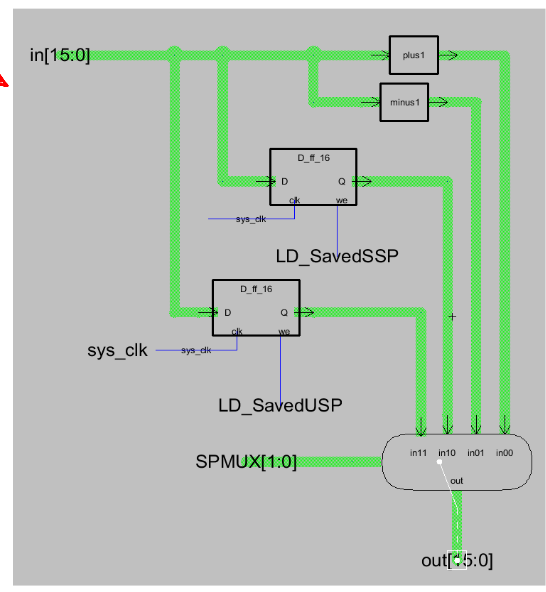
MAR
Interrupt_Logic
Note: The priority of the device attempting to generate an interrupt is compared with the priority of the currently executing instruction. In P&P, this comparison is "A >= B". However, that would cause infinitely nested interrupts: The priority of a device's interrupt handler is set to the priority of the device generating the interrupt; the first instruction of the handler would be interrupted. Here, the comparison is "A > B". This effectively masks interrupts for that device and solves the nested interrupt problem. A side-effect is that priority-0 devices never interrupt. Another is that all interrupts can be masked by setting priority to 7 (via RTI).
Vector Look-up Table
INTV_ROM
VECT
ROM
VECTOR
look-up Table
(EXC)
PRIO
INTpriorityEncode
Unused IRQ inputs are set to 0.
Keyboard: kb_ctl, IRQ[4], priority 4
Display: disp_ctl, IRQ[5]
MCR: IRQ[6]
Priority_Comparator
INT
IRQ
PSR.Priority



Switching Super ↔ User
Mem
VT
Text, data
STACK
OS
Text, data
STACK
USER
I/O
Saved SSP
R6
Saved USP
Privilege
CC
N Z P
1
PSR
Priority
0 Supervisor



ALU
stackOps



in[15:0]
plus1
minus1
LD_SavedSSP
sys_clk
LD_SavedUSP
SPMUX[1:0]
out[15:0]