Introduction

Digital System: All signals have 2 states (on/off, high/low, etc.)

↳ When encoding information in systems, we examine how to interpret/manipulate that info

abstract pieces of digital logic → functional units (i.e. blocks) → add memory/stored info

- **abstraction**: generalizing concrete details

a **system** takes in info and uses it to create new information

↳ data (input) → [system] → new data (output)

↳ **ex.** speed of a wheel → □ → number on speedometer 

Analog System: Any fluctuations in the system adjusts the information

↳ **ex.** if the wheels are inflated slightly more, the output changes

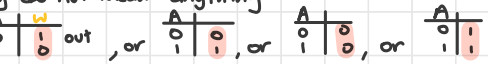
↳ Solution to an analog system: divide info into 2 values, ex. 0–30 mph and 30–∞ mph

↳ everything becomes very predictable

Main benefits of a binary, boolean, or digital system: robust, predictable, repeatable, because there are NO incremental changes — billions of reliable switches that only work in 2 states

Boolean Logic and Expressions

Two states: 0 and 1 ← only representations, they do not mean anything

↳ System example: A → □ → 

↳ 2 input System:

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

X: XOR function ($A \oplus B$), "exclusive or"

3 main:
- Y: AND function ($A \cdot B$)
- Z: OR function ($A + B$), "inclusive or"
- W: NOT function ($\bar{A}$)

definition, not equality

↳ **ex.** $Q = \overline{A+B}$

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

$2^{\#\text{of inputs}}$ = # of rows

Order of Operations:
1. NOT
2. AND
3. OR

Controlling Values

↳ OR: if either input is 1, out is 1

↳ AND: if either input is 0, out is 0

↳ **ex.** $T = \overline{A \cdot B + \bar{C}}$ ← if C is 0, $\bar{C}$ is 1, OR's controlling value is 1, and T is 0

A	B	C	T
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

↳ 1. if C is 0, T is 0

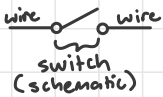
↳ 2. if A or B are 0, $A \cdot B$ is 0. Assuming C is 1, T is 1

↳ 3. Finally, if both A and B are 1, T is 0

Truth Table: Displays all the boolean outputs of a logic expression

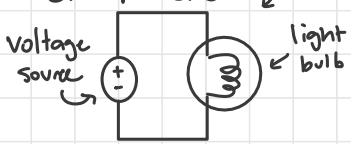
Combinational Logic: It is possible to specify a value for every combo of inputs — aka, can you create a complete truth table?

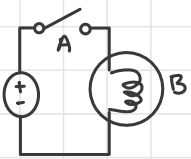
Switches and Switch Networks



→ Switches act as a "push" button, not a "toggle"
↳ inherently digital — they are either activated (1) or not (0)

example circuit 2





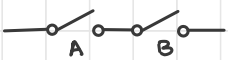
A	B
not activated	off
activated	on

→

A	B
0	0
1	1

B = A

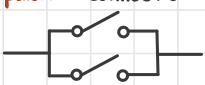
"series connection"



A	B	Is there a connection?
0	0	0
0	1	0
1	0	0
1	1	1

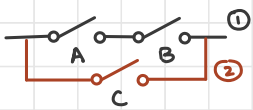
→ Connection = $A \cdot B$ (AND)

"parallel connection"



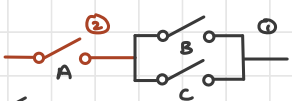
A	B	Is there a connection?
0	0	0
0	1	1
1	0	1
1	1	1

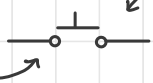
→ Connection = $A + B$ (OR)

↳ ex. Connection = $A \cdot B + C$ → ① ② 

↳ When drawing, follow the order of operations: AND first, OR second

→ Wire convention is that:  if ambiguous

↳ ex. $A \cdot (B + C)$ → 

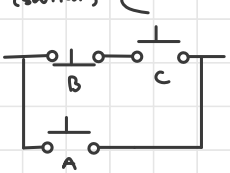
Schematic symbols:  is equivalent to normally open (not making a connection) 

resting state: open

A	Conn
0	1
1	0

NOT (switch) ← normally closed →

resting state: closed

↳ Ex. 

Conn = $\bar{B} \cdot C + A$

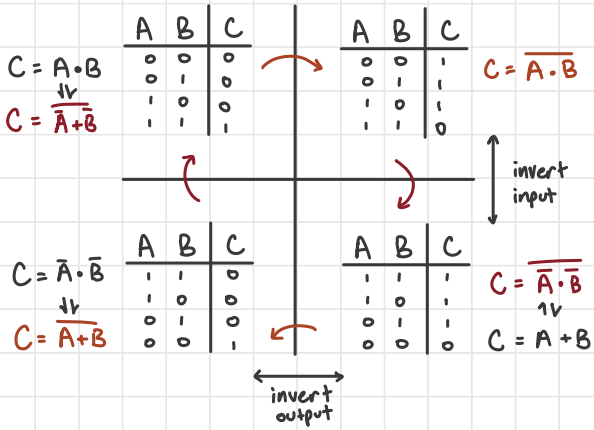
Algebraic Multiplication and Simplification

De Morgan's Theorem: $\overline{A \cdot B} \rightleftharpoons \bar{A} + \bar{B}$, $\overline{A + B} \rightleftharpoons \bar{A} \cdot \bar{B}$ "$\rightleftharpoons$" means logically equivalent

↳ used to simplify logic to make circuitry possible (get rid of the big bars)

↳ numerous other theorems, can be proved using circuitry

Proving DeMorgan's Theorem:



Applying DeMorgan's Theorem: $\overline{A+B}$

1) invert inputs $(A \to \bar{A})$ $\overline{\bar{A}+\bar{B}}$
2) change operators $(\cdot \to +,\ + \to \cdot)$ $\overline{\bar{A}\cdot\bar{B}}$
3) invert output (final result) $\overline{\overline{A}\cdot\overline{B}} = \bar{A}\cdot\bar{B}$

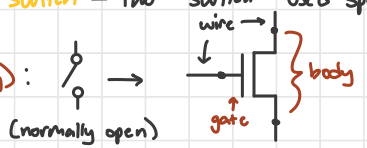
↳ **ex.** $\overline{A\cdot(B+C)}$ is $\overline{A\cdot(\text{something})}$
↳ $\bar{A}\cdot\overline{(\text{something})} \to \bar{A}+(\bar{B}\cdot\bar{C})$

↳ **ex.** $\overline{A\cdot B+C}$ 2nd to last step in evaluating, so first in DeMorgan's
↳ $\overline{(\text{something})}\cdot\bar{C} \to (\bar{A}+\bar{B})\cdot\bar{C}$
↳ need parenthesis to keep order of operations

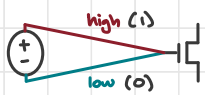
Transistors, logic levels, CMOS & Gate-level design

FET (Field-effect transistor) switch – the "switch" used specifically for ECE 2020

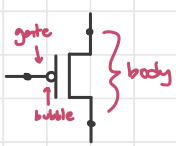
↳ N-FET (negative silicon doping):



(normally open)

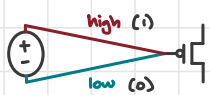


↳ P-FET (positive silicon doping):



(normally closed)

gate/switch activation

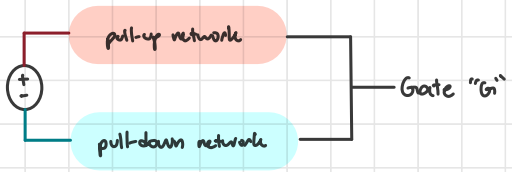


↳ To force something to a specific voltage, it must be connected to a specific voltage source
↳ if a P or N-FET is connected to high voltage then disconnected, the gate is still "set" to high ("floating")

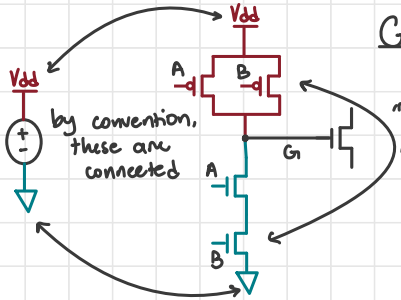
gate	Connection? N-FET	P-FET
low (0) (Connected to low side of voltage)	open (0)	closed (1)
high (1) (Connected to high side of voltage)	closed (1)	open (0)

↳ **ex.** $G_1 = \overline{A\cdot B}$

A	B	pull up?	pull down?
0	0	1	0
0	1	1	0
1	0	1	0
1	1	0	1



pull-up network = $\overline{A\cdot B}$ = $\bar{A}+\bar{B}$ } complements
pull-down network = $A\cdot B$



Gate G

"A" switches and "B" switches are also connected

MOSFET - metal oxide semiconductor

CMOS - complementary MOS

↳ a way of using MOSFETs to make a network to connect to low vs high voltages

Why are complementary pull-up / pull-down networks needed?

↳ **ex.** if we have pu/pd networks such that:

A	B	pu	pd	Gate P
0	0	0	1	0
0	1	0	0	Floating (Z) ← unstable
1	0	1	0	1
1	1	1	1	short-circuit ← 🔥

NOT a CMOS

Convention (necessary)

↳ pu networks are P-FETs

↳ pd networks are N-FETs

↳ **ex.** $S = A \cdot B$

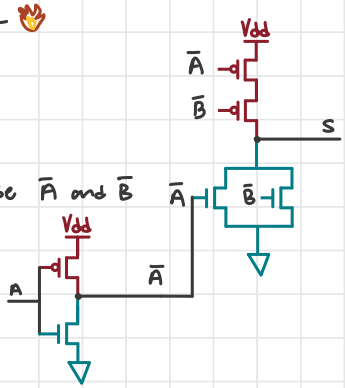
↳ $pu = A \cdot B$, $pd = \overline{A \cdot B} = \bar{A} + \bar{B}$

↳ pu uses P-FETs, so must use $\bar{A}$ and $\bar{B}$

↳ vice versa for pd

↳ **ex.** how do we get $\bar{A}$?

↳ $pu = \bar{A}$, $pd = \bar{\bar{A}} = A$



Gate Design

Logic schematics have certain symbols for basic functions, and flow (input → output) left to right

↳ AND

↳ OR

↳ NOT

bubble can be added to signify inversion

↳ NAND

↳ NOR

Gate Schematic - a diagram of how gates connect to each other

↳ **ex.** $A = \overline{A \cdot B + C}$ ↳ schematic ONLY emphasizes the logic

↳ 

Mixed Logic - decoupling intention from implementation

↳ ex. Imagine if there is no paper in a printer, it sends a high voltage

↳ This circuit needs to be implemented when low voltage = false

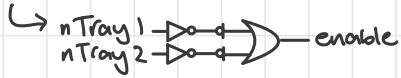
↳ In mixed logic, the "activation level" is separated from the logic operations

↳ nTray1, nTray2 → enable

→ prefix "n" or bar specifies inputs are "active-low"

→ bars on the wires are where inversions need to be

↳ To resolve the neccessary inversions (bars), inverters must be implemented

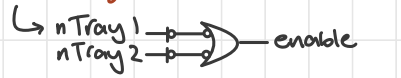
↳ 

→ bubbles are placed on bars to show they are handled

→ BUT, bubbles and bars are NOT physical devices

↳ only inverters are physically integrated

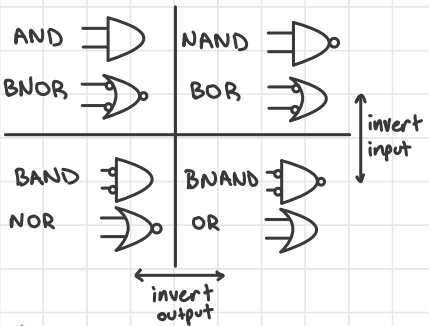
↳ Mixed logic also has flexible implementation (via DeMorgan's Theorem)

↳ 

→ OR has been bubbled (if either input is low, output high)

→ "bubbled OR" or BOR has a physical equivalent, NAND

← "BOR" and "BAND" gates are not sold, as there are physical equivalents — but they MUST be used to resolve the mixed logic

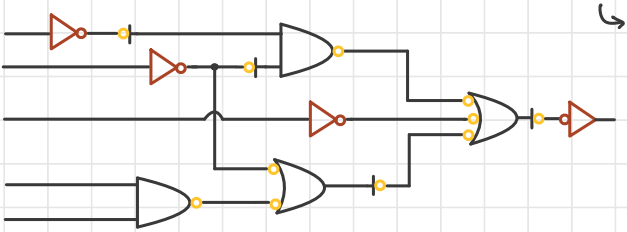


Mixed Logic Manipulation — 2 main goals for the flexibility of a mixed logic schematic:

1) Minimize # of transistors if implementing in CMOS (implementation cost)

2) Use a specific kind of gate (often to use only 1 kind of gate)

↳ ex. The goal for this circuit is to use only NAND gates and inverters



↳ 1) All gates converted to NAND

↳ ○ output for AND, ○ input for OR

2) inverters added where neccessary

↳ Make sure to keep bubble pairs together

Logic Simplification with k-Maps and SoP

↳ ex. Turn this truth table into a boolean expression

A	B	Y	Z	P
0	0	0	0	0
0	1	0	0	0
1	0	0	1	1
1	1	1	0	1

→ $Y = A \cdot B$

→ $Z = A \cdot \overline{B}$

↳ $P = A \cdot B + A \cdot \overline{B}$

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

($B + \overline{B}$: always true)

$P = A$

↳ ex.

A	B	C	W
0	0	0	1 ←
0	0	1	0
0	1	0	1 ←
0	1	1	1 ←
1	0	0	0
1	0	1	0
1	1	0	1 ←
1	1	1	0

→ Look at all input combos where W=1, then list:

↳ $W = \overline{A} \cdot \overline{B} \cdot \overline{C} + \overline{A} \cdot B \cdot \overline{C} + \overline{A} \cdot B \cdot C + A \cdot B \cdot \overline{C}$

1- "minterm" (output is 1), combo of inputs ANDed together

↳ Canonical Sum of Products — full list of all combos

↳ How do we simplify?

Karnaugh Map (K-Map) - grid where # of rows (inputs) = # of cells, and moving one cell to another only changes 1 input

↳ How? With a gray code (i.e. 00, 01, 11, 10), where only 1 input changes at a time

↳ Looking at adjacent "1" cells tells us that one of the inputs don't matter, helping to simplify the canonical sum of solutions

A	B	C	W
0	0	0	1 ←
0	0	1	0
0	1	0	1 ← ←
0	1	1	1 ←
1	0	0	0
1	0	1	0
1	1	0	1 ←
1	1	1	0

ex.

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

→ If A and C are both 0, B doesn't matter ($\bar{A} \cdot \bar{C}$)

→ If A is 0 and B is 1, C doesn't matter ($\bar{A} \cdot B$)

→ If B is 1 and C is 0, A doesn't matter ($B \cdot \bar{C}$)

↳ $W = \bar{A} \cdot \bar{C} + \bar{A} \cdot B + B \cdot \bar{C}$

A	B	C	J
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

ex.

A \ BC	00	01	11	10
0	0	0	0	0
1	1	1	1	1

$A \cdot \bar{B}$ & $A \cdot B$ → A

↳ Implicants of the same size/shape can be combined (if they share the same inputs)

↳ These will form rectangles with 2^n in each dimension (2×2, 1×4, etc.)

↳ Large as possible implicants are called prime implicants

↳ as simple as possible for the minimal sum of products (SoP)

ex.

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

→ $B \cdot D + A \cdot \bar{C} + \bar{B} \cdot \bar{C} \cdot \bar{D}$

↳ essential prime implicants covers something no other prime implicant covers

↳ in this example, all prime implicants are essential

ex.

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

→ $A \cdot B + \bar{C} \cdot D$

↳ only essential prime implicants needed

↳ nonessential: $B \cdot D$

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

→ minimal SoP can be made out of different options (non-essential)

implicants wrap around

↳ ex. Say some outputs don't matter

A	B	C	G
0	0	0	0
0	0	1	x ←
0	1	0	x ←
0	1	1	0
1	0	0	x ←
1	0	1	1
1	1	0	0
1	1	1	1

how can the logic be simplest? (x as 0 or 1)

A \ BC	00	01	11	10
0	0	x	0	x
1	x	1	1	0

→ $A \cdot C$

↳ only prime imp

potential prime imp

ex.

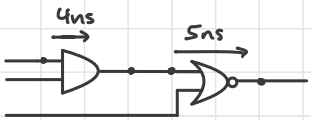
A \ BC	00	01	11	10
0	0	x	1	1
1	x	1	x	0

→ $V = \bar{A} \cdot B + C$

↳ Simpler if some X's are 1

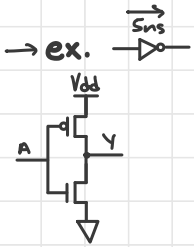
non-essential (multiple cover the 1)

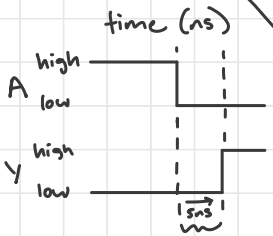
Circuit Timing



Circuit timings – abstractions give an easier way to estimate the delay between an input and output
↳ the amount of time it takes for a change to propagate through a circuit

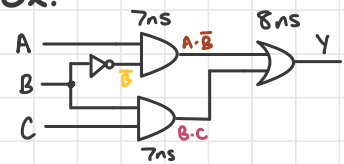
→ ex.

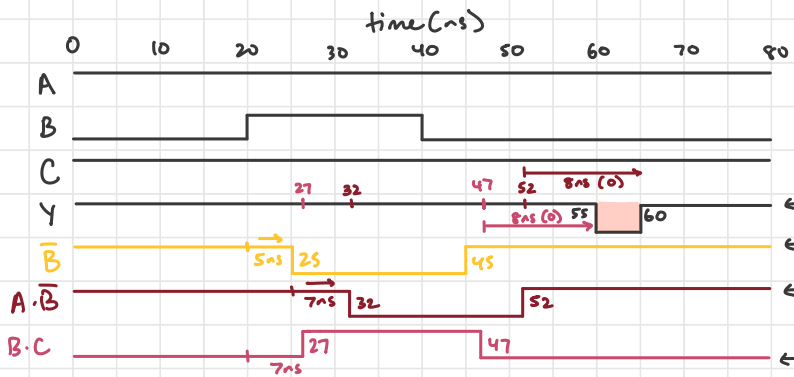




propagation delay

→ ex.

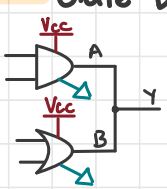




glitch in Y due to propagation delay
← due to change in A·B̄, B·C
← due to change in B
← due to change in B̄
← due to change in B

Gate Design (cont'd)

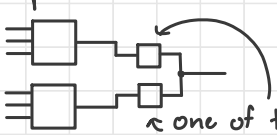
↳ connecting the output of 2 gates is problematic



↳

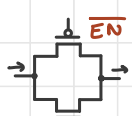
A	B	Y
0	0	0
0	1	} short-circuit
1	0	
1	1	

↳ solution:



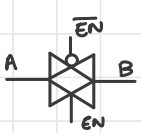
↑ one of these are disconnected

↳ Solution 1: Use transistors



EN	connection?
0	disconnected
1	connected

→

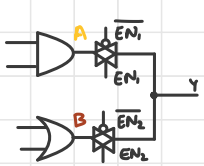


transmission gate
(t-gate)
(pass gate)

EN	A	Y
0	x	Z
1	0	0
1	1	0
1	Z	Z

if disconnected, A doesn't matter

↳ ex.



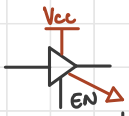
EN_1	EN_2	A	B	Y
0	0	x	x	z
0	1	0	0	0
0	1	0	1	1
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	0
1	0	1	0	1
1	0	1	1	1
1	1	0	0	0
1	1	0	1	short
1	1	1	0	short
1	1	1	1	

covers 4 rows

$Y = B$

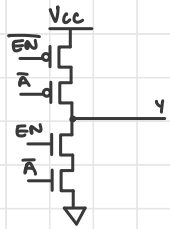
$Y = A$

↳ Solution 2: connect directly to high/low



tristate driver/buffer

EN	A	Y	pu	pd
0	0	z	no	no
0	1	z	no	no
1	0	0	no	yes
1	1	1	yes	no



pu: $EN \cdot A$

pd: $EN \cdot \bar{A}$

Number Systems

decimal	encoding
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

n number of digits can produce 2^n combinations

↳ range of unsigned binary: $[0, 2^n - 1]$

↳ converting Base 2 → Base 10

$0\,1\,1\,0\,1\,1\,0$

$2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^0$

$0 + 32 + 16 + 0 + 4 + 2 + 0$

↳ ex. Addition:

$$\begin{array}{r} 1\,1\,1\,1\,1 \\ 0\,1\,1\,0\,1\,1\,0 \\ +\,0\,0\,1\,1\,1\,0\,1 \\ \hline 1\,0\,1\,0\,0\,1\,1 \end{array}$$

$0\,1\,1\,0\,1\,1\,0 \rightarrow 54$

$0\,0\,1\,1\,1\,0\,1 \rightarrow 29 = 0 + 0 + 16 + 8 + 4 + 0 + 1$

$1\,0\,1\,0\,0\,1\,1 \rightarrow 83$

overflow – when some operation results in a value outside range of values we can represent

↳ ex.

$$\begin{array}{r} 1\,1\,1\,0\,1 \\ +\,0\,1\,1\,0 \\ \hline 0\,0\,1\,1\,1 \end{array}$$

(1) indicates overflow in this particular encoding scheme

→ negative values can be shown with signed magnitude, where the first value indicates sign:

neg: $1\,0\,0\,1 \rightarrow -1$

pos: $0\,0\,1\,1 \rightarrow 3$

$0\,0\,1\,0 \rightarrow 2$

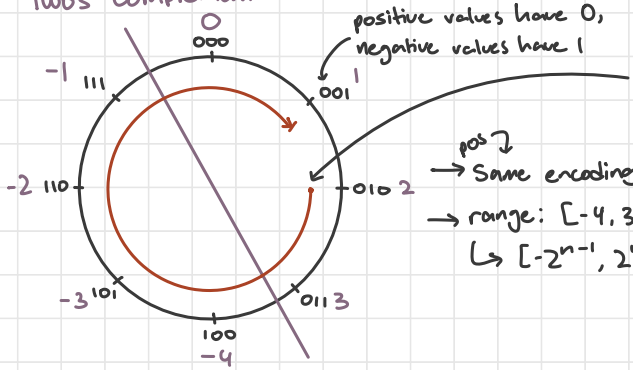
01	1
00	0
10	0
11	-1

↳ $1\,0\,0\,1\,0$ ↳ can't do

sign | magnitude bit addition!

↳ range: $[-(2^{n-1}-1), 2^{n-1}-1]$ because you have pos/neg 0

Two's Complement



positive values have 0, negative values have 1

pos → same encoding as unsigned binary

→ range: $[-4, 3]$

↳ $[-2^{n-1}, 2^{n-1}-1]$

carry out, but NO overflow

$0\,1\,0 \rightarrow 2$

$+\,1\,1\,1 \rightarrow -1$

$0\,0\,1 \rightarrow 1$

overflow occured

$1\,0\,1 \rightarrow -3$

$+\,1\,1\,0 \rightarrow -2$

$0\,1\,1 \rightarrow 3$, not -5

Negating → in signed magnitude

$0\,0\,1\,0\,1$ pos → $1\,0\,1\,0\,1$ neg

1) flip sign

in Two's complement

$0\,0\,1\,0\,1$ → flip → $1\,1\,0\,1\,0$ → +1 → $1\,1\,0\,1\,1$ → flip → $0\,0\,1\,0\,0$

1) flip every digit

2) add 1

↳ ex. $\begin{array}{r} 0010111 \\ -\ 0111100 \end{array}$ $A - B = A + (-B)$ → $\begin{array}{r} 0010111 \\ +\ 1000100 \\ \hline 1011111 \end{array}$ → 11 → negate, then count → 0111100, 28, so −28

↳ −17 = 11 − 28

Base 16: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F (10, 11, 12, 13, 14, 15) ↳ ex. 1101 0011 1111 = D F

$13 \times 16^2 + 3 \times 16^1 + 15 \times 16^0$

↳ hexadecimal is represented by DEF_{16} or 0xDEF

↳ binary is represented by 11011_2 or 0b11011

Fractions/Decimals: 2 methods

↳ floating-point: in code, determine where the fractional separator is

↳ fixed-point: as the engineer, choose where it is within the system

↳ ex. 101.011 (fractional separator)

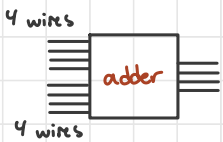
$2^2\ 2^1\ 2^0\ 2^{-1}\ 2^{-2}\ 2^{-3}$

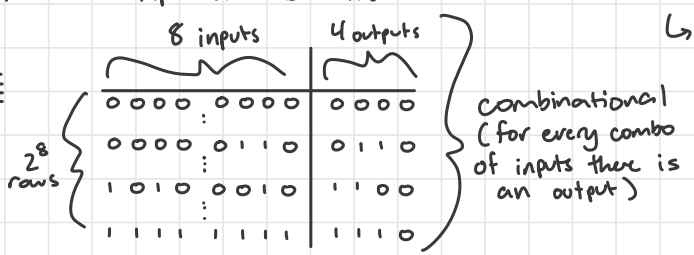
$4 + 0 + 1 + 0 + 1/4 + 1/8 \rightarrow 5.375_{10}$

Building Blocks

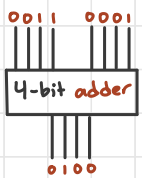
abstraction — a system with some functional behavior

↳ an adder, takes in inputs and adds them

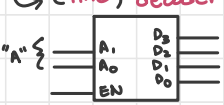




↳ ex.



↳ (line) decoder — a combination of inputs → activates one output (selected by inputs)



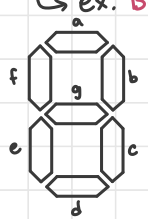
EN	A_1	A_0	D_3	D_2	D_1	D_0
0	x	x	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

enable

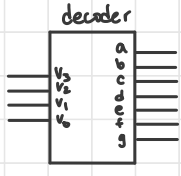
$D_0 = EN \cdot \overline{A_0} \cdot \overline{A_1}$, $D_1 = EN \cdot \overline{A_0} \cdot A_1$, etc.

↳ able to be implemented with gates

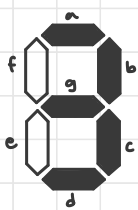
↳ ex. BCD-to-7 segment decoder



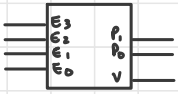
number	binary
0	0000
1	0001
2	
3	
4	
5	
6	
7	
8	1000
9	1001



V_3	V_2	V_1	V_0	a	b	c	d	e	f	g
0	0	0	0	1	1	1	1	1	1	0
0	0	0	1	0	1	1	0	0	0	0
0	0	1	0	1	1	0	1	1	0	1
0	0	1	1	1	1	1	1	0	0	0
1	0	0	1	1	1	1	0	1	1	1



↳ (priority) encoder — one activated input → a combination of outputs



highest priority →

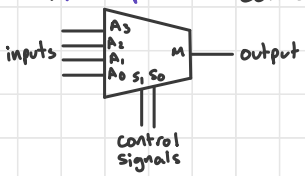
E_3	E_2	E_1	E_0	P_0	P_1	V
0	0	0	0	x	x	0
0	0	0	1	0	0	1
0	0	1	x	0	1	1
0	1	x	x	1	0	1
1	x	x	x	1	1	1

← valid input (at least 1 input active)

E_3 E_2 E_1 E_0 (high → priority → low)

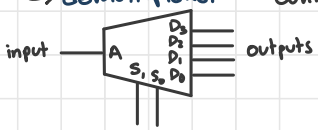
↳ ex. if E_3 true, E_2, E_1, and E_0 don't matter

↳ **multiplexer** — connects one of the inputs to the output



S_1	S_0	M
0	0	A_0
0	1	A_1
1	0	A_2
1	1	A_3

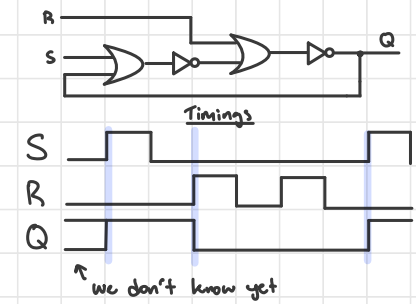
↳ **demultiplexer** — connects the input to one of the outputs



S_0	S_1	A	D_3	D_2	D_1	D_0
0	0	0	Z	Z	Z	0
0	0	1	Z	Z	Z	1
0	1	0	Z	Z	0	Z
0	1	1	Z	Z	1	Z
1	0	0	Z	0	Z	Z
1	0	1	Z	1	Z	Z
1	1	0	0	Z	Z	Z
1	1	1	1	Z	Z	Z

floating, because disconnected

Latches and Flip-Flops



→ Designing something that can hold information

↳ $Q = \overline{(\overline{Q+S}+R)}$

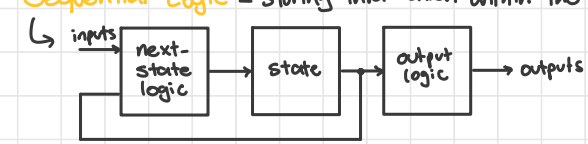
→ $Q = \overline{\overline{Q+0}+0} = \overline{\overline{Q}} = Q$

→ using S and R, we can 'force' Q to be 0 or 1

→ trying to go from 0 to 1 causes propagation delay

S	R	Q
0	0	Q
0	1	0
1	0	1
1	1	unstable (?)

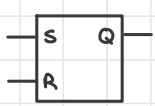
Sequential Logic — storing information within the circuit (different from combinational logic)



↳ set & reset

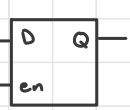
↳ **S-R Latch**

S	R	Q
0	0	Q
0	1	0
1	0	1
1	1	0

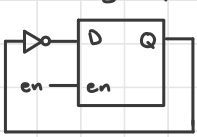


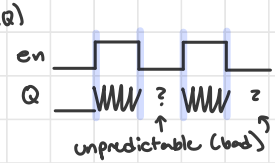
↳ **D-Latch**

en	D	Q
0	0	Q
0	1	Q
1	0	0
1	1	1



→ **ex.** inverting output (Q)



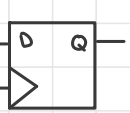


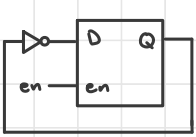
Solution: new device

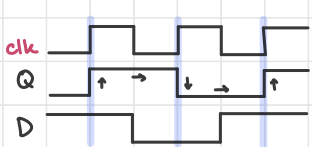
↳ clocked D flip-flop

↳ **Clocked D flip-flop**

clk	D	Q
⌐	0	Q
⌐	1	Q
⌙	0	0
⌙	1	1







↳ depends on 'rising' and 'falling' edges on clk

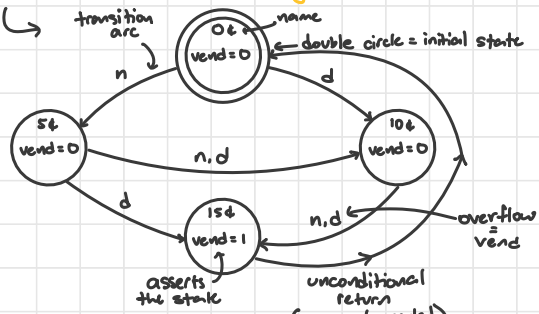
↳ event sensitivity → allows sequential logic to be predictable

Finite State Machines

→ ex. a gumball machine that takes nickels and dimes, and gumballs cost 15¢

↳ 1) Do we need sequential logic? Do we need to keep track of diff. states?

↳ Yes; use a State Diagram – decide which info to hold



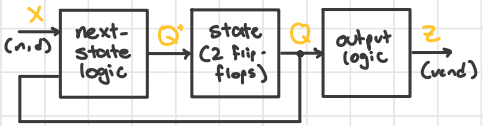
→ Based on the current state, figure out what the output is

↳ State encoding table

Name	Q_1	Q_0
0¢	0	0
5¢	0	1
10¢	1	0
15¢	1	1

needs at least 2 flip-flops to hold onto 2 values

↳



Transition Table

↳

Current State	Q_1	Q_0	X_1 ← d	X_0 ← n	Next State	Q_1^+	Q_0^+	Z ← vend	
0¢	0	0	0	0	0¢	0	0	0	
0¢	0	0	0	1	5¢	0	1	0	
0¢	0	0	1	0	10¢	1	0	0	
0¢	0	0	1	1	?	X	X	0	← figure out later for next-state logic
5¢	0	1	0	0	5¢	0	1	0	
5¢	0	1	0	1	10¢	1	0	0	
5¢	0	1	1	0	15¢	1	1	0	
5¢	0	1	1	1	?	X	X	0	
10¢	1	0	0	0	10¢	1	0	0	
10¢	1	0	0	1	15¢	1	1	0	
10¢	1	0	1	0	15¢	1	1	0	
10¢	1	0	1	1	?	X	X	0	
15¢	1	1	0	0	0¢	0	0	1	
15¢	1	1	0	1	0¢	0	0	1	
15¢	1	1	1	0	0¢	0	0	1	
15¢	1	1	1	1	0¢	0	0	1	

impossible, be only 1 coin slot

state changes first

state = 15¢, then vend

unconditional transition

→ Output Logic is combinational

↳ for any state (Q_1, Q_0), vend is known

Q_1	Q_0	Z
0	0	0
0	1	0
1	0	0
1	1	1

$Z = Q_1 \cdot Q_0$

↳ Next-State Logic is also combinational, depends on $Q_1 Q_0$ and $X_1 X_0$

↳ Two outputs, so two K-Maps (Q_1^+, Q_0^+)

Q_1^+:

$Q_1 Q_0$ \ $X_1 X_0$	00	01	11	10
00	0	0	X	1
01	0	1	X	1
11	0	0	0	0
10	1	1	X	1

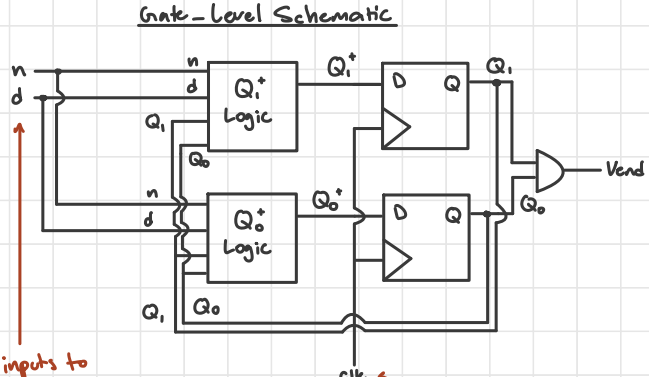
$Q_1^+ = Q_1\overline{Q_0} + \overline{Q_1}X_1 + \overline{Q_1}Q_0X_0$

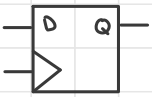
Q_0^+:

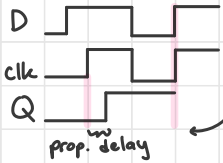
$Q_1 Q_0$ \ $X_1 X_0$	00	01	11	10
00	0	1	X	0
01	1	0	X	1
11	0	0	0	0
10	0	1	X	1

$Q_0^+ = \overline{Q_0}X_0 + \overline{Q_1}Q_0\overline{X_0} + Q_1\overline{Q_0}X_1$

Gate-Level Schematic





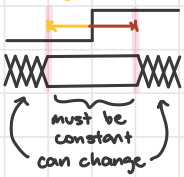


What happens? unpredictable!

↳ solution: have requirements when D can change before and after a rising clk edge

↳ t_{su} t_h → t_{su}, setup time

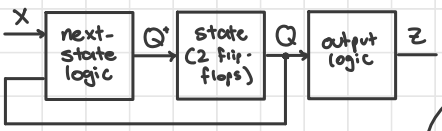
→ t_h, hold time

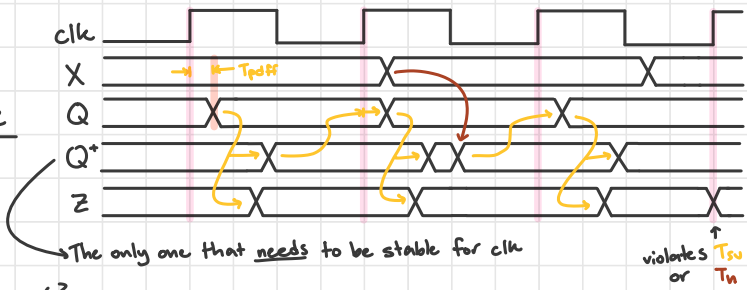


↳ NOT physical delays, just requirements to ensure predictable behavior

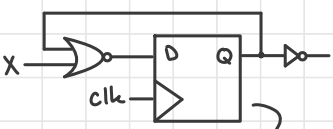
↳ There is prop. delay from clk → Q, but not from D → Q

→ **ex.** timing delays





→ **ex.** when should we NOT change X?

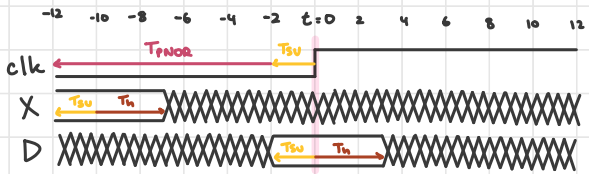


T_{PNOR} = 10ns

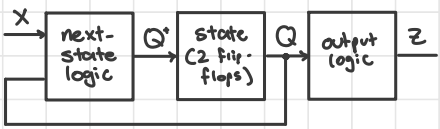
T_{PNOT} = 4ns

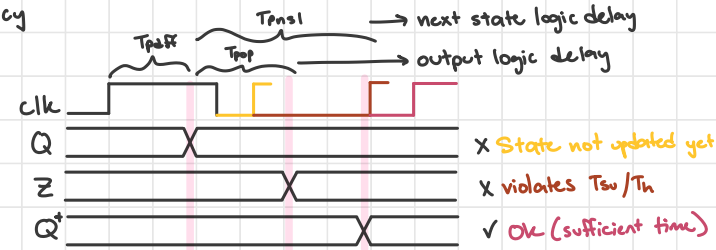
T_{pdff} = 6ns

T_{su} = 2ns, T_h = 3ns

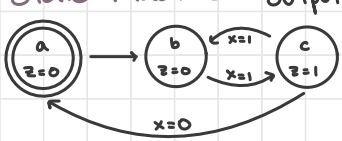


→ **ex.** min clk period / max frequency





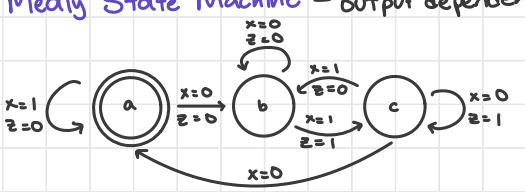
Moore State Machine – output only dependent on state



→ only need info about state to find output

→ inputs only give info about next state

Mealy State Machine – output dependent on state and inputs



→ need info about state and inputs to find output

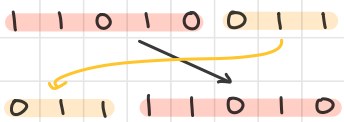
→ inputs give info about next state and output

Shifting

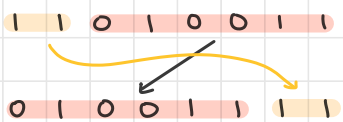
→ typically to the left or right, where the number of bits going in = number going out

barrel shift – take the fall-out and re-insert it

↳ ex. right shift 3

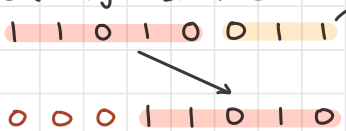


↳ ex. left shift 2

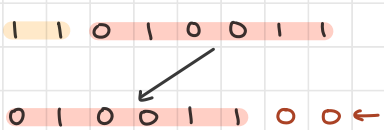


logical shift – replace gaps with 0s

↳ ex. right shift 3



↳ ex. left shift 2



arithmetic shift – goal → if right n bits, number × 2^{-n}

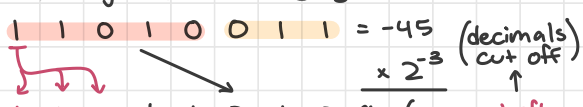
↳ if left n bits, number × 2^{n}

↳ if left, fill in 0s

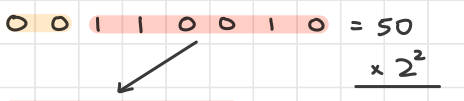
if right, fill in 0s IF positive, use 1 IF negative

↳ can think of distributing the 1st bit to all gaps if right shift

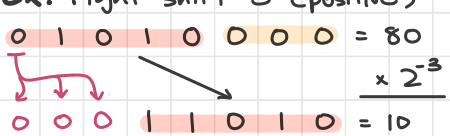
↳ ex. right shift 3 (negative)



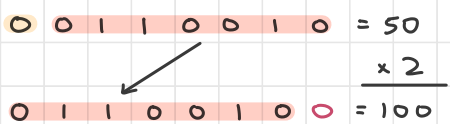
↳ ex. left shift 2



↳ ex. right shift 3 (positive)



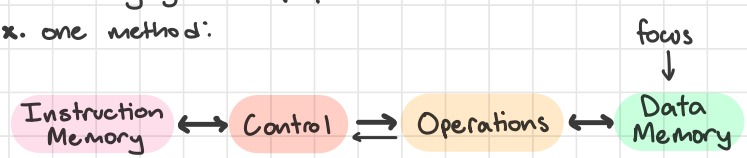
↳ ex. left shift 1



Memory and Data Storage

Goal: Creating general-purpose hardware

↳ ex. one method:



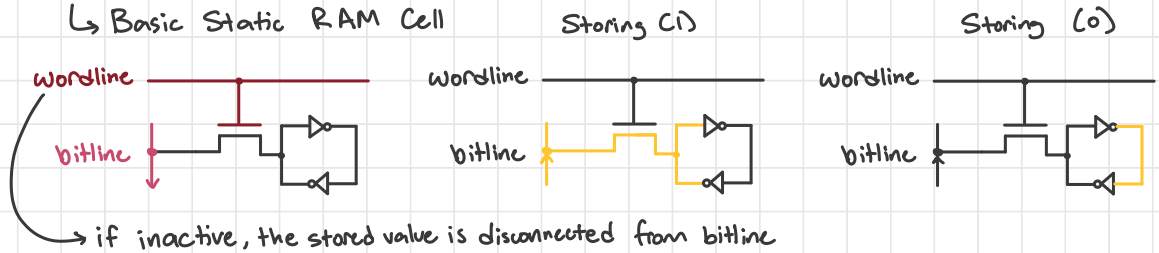
→ instruction memory – holds instructions

→ controls – controls the system, synchronized across system

→ operations – executes instructions (general purpose)

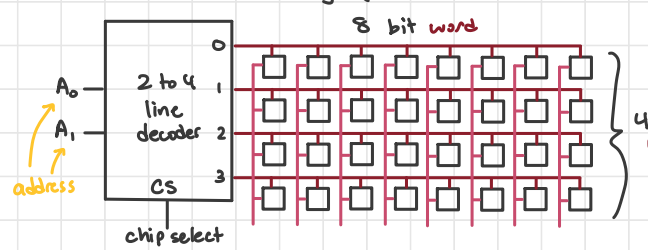
→ data memory – values are stored

Data Memory → has a driven input address, outputs data at address
↳ Control lines enable outputs + control data reading/writing
↳ a "word" is a value, one "word" at each address
↳ N bits of address can uniquely identify 2^n words
↳ M words needs $\log_2(M)$ addresses
↳ ex. 8 bits of address can find $2^8 = 256$ words
↳ bit memory - used to store a single 1 or 0
↳ several bit memory units can be used to store a word, depending on how many bits a word is
↳ Basic Static RAM Cell

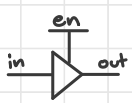


↳ if inactive, the stored value is disconnected from bitline

Bit Cell Arrays - if words are B-bits wide, and there are W words, you need B·W bit cells
↳ usually arranged in a rectangular format
↳ choosing which bit cells are active is done through decoders
↳ ex. 4 x 8 Memory (word count x size)

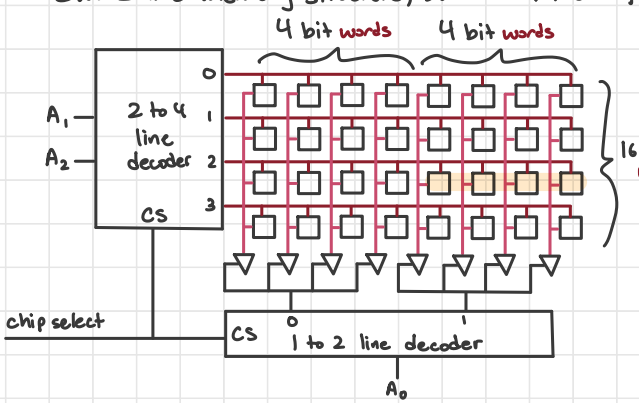


→ unidirectional transmission gate



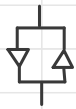
↳ tristate drivers are needed bc memory capacitors are too weak to drive circuits

↳ ex. Same memory structure, but 8x4 Memory



→ can break up the rectangular array for diff. number of bits
↳ chooses which "column" to look at, by determining which words connect to the output
↳ ex. address 5 is 101, since $A_0 = 1$, it looks at the right column, then wordline 10, or 2

/RW – additional logic to choose between read/write settings

↳ can be implemented with bidirectional tristate drivers → 

Memory Blocks – once a reasonable amount of memory is designed, they can be abstracted to a functional block

Abbreviations – CS → "chip select", $2^{10} = 1{,}024 \rightarrow$ "1K", $2^{20} = 1{,}048{,}576 \rightarrow$ "1M"

↳ **ex.** 4M words needs how many address bits? ↳ $\log_2(4M) = \log_2(2^2 \cdot 2^{20}) = 22$ address bits

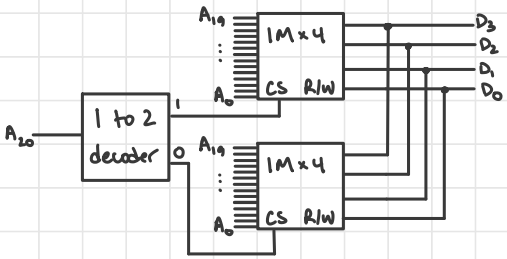
↳ **ex.** 256K 16-bit words, how many bit cells? ↳ $2^8 \cdot 2^{10} \cdot 2^4 = 2^{22}$ bits, or $2^{20} \cdot 2^2 = 4M$ bits

Combining Memory Blocks

↳ **ex.** given 1M×4 memory blocks, how can we make 2M×4 memory?

1) need to double # of words, so 1 more address bit

2) only have 1M×4, so need 2 blocks



→ CS, or "chip signal", if 0 disables memory block by making the decoder output all 0's

→ if A_{20} ($\downarrow_{20}\ \overline{\ }_{19} \cdots \overline{\ }_{0}$) is 1, the top memory block is active, if A_{20} is 0, the bottom memory block is active

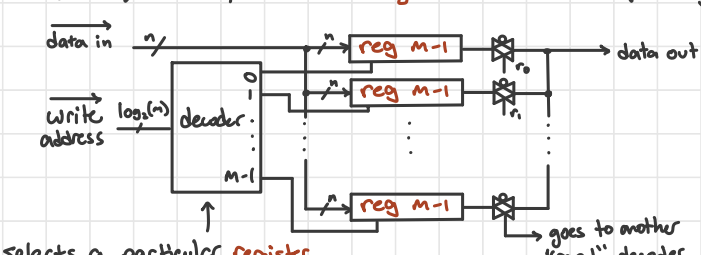
Datapath Elements

Datapath – memory to store temporary data for currently executing operations

↳ Instruction Memory ↔ Control ⇄ Operations ↔ Data Memory [focus]

Register – a set of flip flops all clocked together to store a multi-bit value

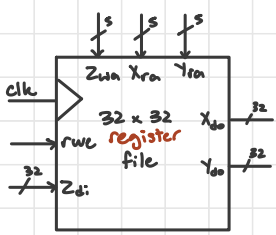
↳ arranged to form a "register file" to temporarily store values



↳ individual "write enables" cause one register to store incoming data

↳ 1 databus (n wires) and 1 decoder to pick whichever register to read/write from

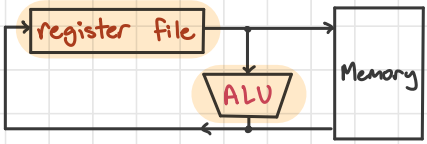
MIPS Architecture – an example computer architecture framework



- ↳ Xra, Yra – read address
- ↳ Xdo, Ydo – data out
- ↳ Zwa, Zdi – write address, data in
- ↳ rwe – register write enable

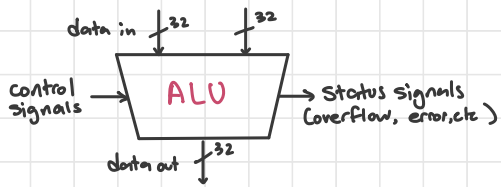
} 32 bit Zdi, so 32 registers = 2^5, 5 address bits to select any register

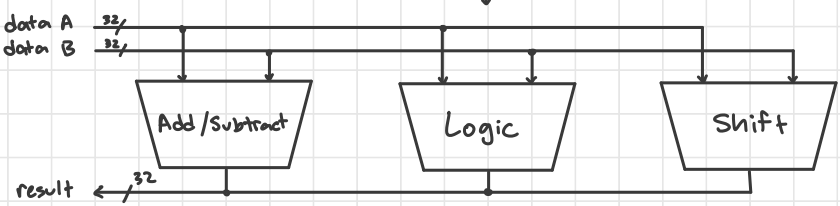
General Data Flow (datapath) →



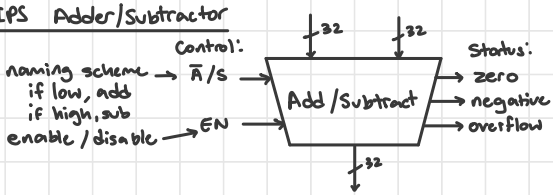
ALU – arithmetic and logic unit

- ↳ collection of devices
  - ↳ adder/subtractor (generally)
  - ↳ logic unit (AND, OR, XOR, NOT)
  - ↳ shifter





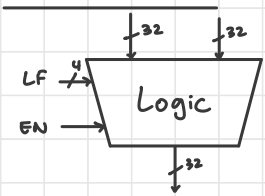
MIPS Adder/Subtractor



Can represent any logic operation

A	B									
0	0	0	0	0	0	0	0	0	0	1
0	1	0	0	0	0	1	1	1	1	1
1	0	0	0	1	1	0	0	1	1	1
1	1	0	1	0	1	0	1	0	1	1
		0	A·B	A·B̄	. . .				A+B	1

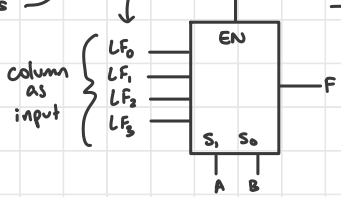
MIPS Logic Unit



→ performs bitwise operations

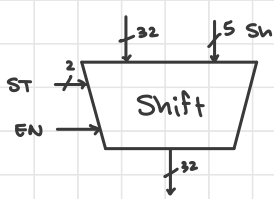
↳ ex. Logical AND

```
0 0 1 1 0 0 1 1
0 1 0 1 0 1 0 1
0 0 0 1 0 0 0 1
```



→ Logic unit has 32 multiplexers, one for each X and Y bit (A and B) to compare

MIPS Shift Unit

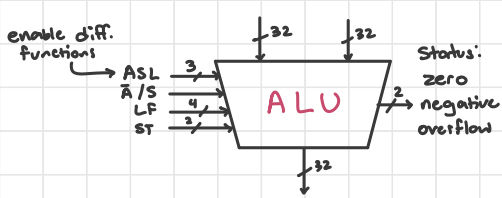


→ ShAmt → Shift Amount

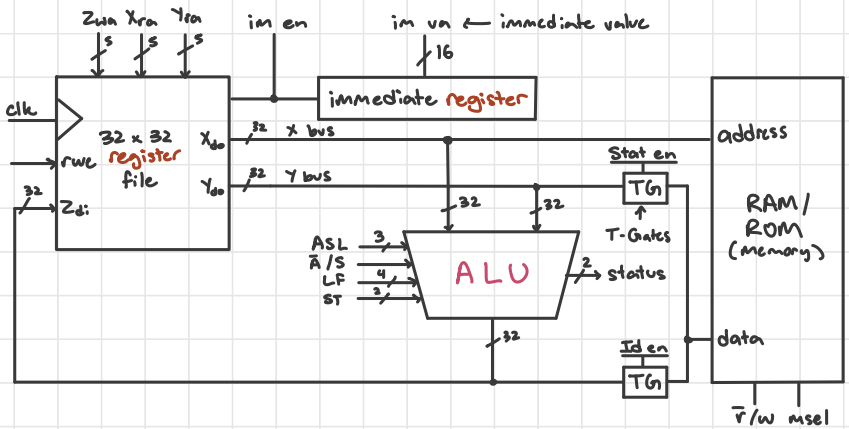
↳ 1 bit to choose left/right

↳ 4 bits to choose amount (up to 16)

MIPS ALU



Full Datapath + Memory



Single-Cycle Datapath

Microcode – control signals that directly interact with datapath

Machine Code – encoding of microcode into more compact form to store in memory

Assembly Code – human-readable way to write machine code

↳ ex. "ADD" to stand in for adding within machine code

Deriving microcode – placing all necessary signals/labels into a table

↳ ex adding two numbers and storing into a specific register

↳ ASM format: add $1, $3, $5 ← specific to MIPS

↳ RTL format: R1 = R3 + R5 ← more generic syntax

↳ enable adder, disable others

↳ address 3 onto Xra, address 5 onto Yra

↳ store result into 1, so Z_{di} to 1

↳ register write enabled

↳ immediate value disabled

↳ disconnect memory, so st en and id en inactive

↳ ex. storing an immediate value into a register

↳ addi $2, $1, 89 → immediate value is only 16 bits, so you need to add lower/upper

↳ several ways to do this, i.e. add immediate with 0 (R0 is always 0)

↳ ex. accessing memory (loading a word)

↳ lw $2, ($1) ← parenthesis indicates an address

→ shifts require an input from immediate value

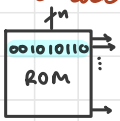
→ storing a value in memory requires placing value into register first, then → memory

Instruction-to-Code Decoder — stores all signal combinations into a read-only memory

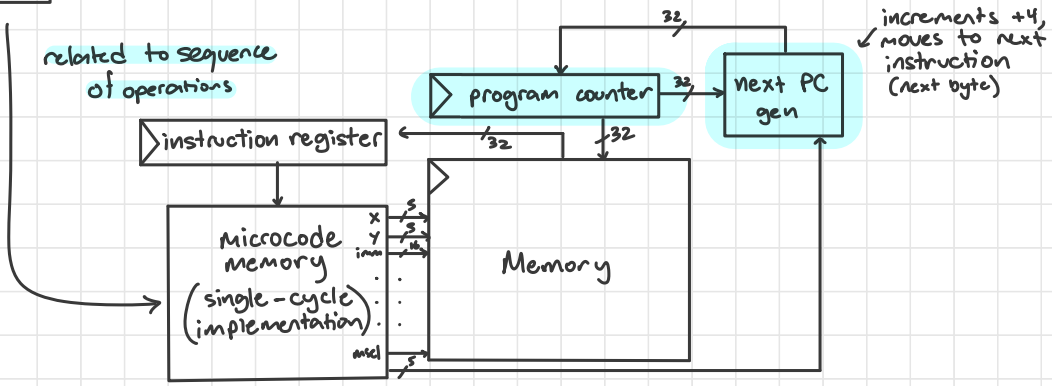
↳ Machine code instruction used as address into ROM

↳ → control signals are the data stored at each address

↳ uses memory as translator between microcode and machine code



related to sequence of operations



Program Counter (PC) — holds address of current instruction

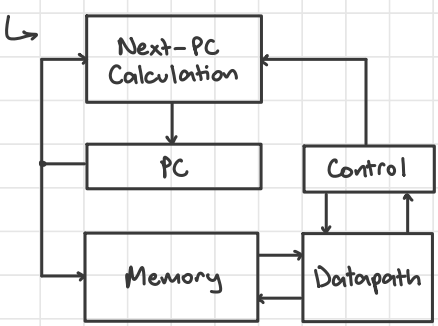
↳ changing PC moves execution to somewhere else in memory (i.e. BRANCH or JUMP)

↳ starts at 0, increments by 4 each cycle

↳ instructions are 32 bit, and memory is addressed to the byte

↳ PCs are $\underbrace{XXXX\ldots}_{30}00$

Fetching an instruction



↳ ex. RTL example: 25+N, 25-N, N-25

```
label ←  main:  R2 = 25          * immediate value → R2
                R3 = MEM[value]
                R4 = R2 + R3
                R5 = R2 - R3
                R6 = R3 - R2
label (address) ← value: .word 12
```

↳ branch if equal and branch if not equal } actual implementation: subtract R3 and R0 and branch if result ≠ 0

↳ ex. If R3 = R0 GOTO [label]

↳ beq $3, $0, [offset]

↳ ex. multiply using add, 5×7

```
main: R1 = MEM[M1]
      R2 = MEM[M2]
      R4 = 0
```

↳

```
M1:     .word 5
M2:     .word 7
result: .word 0
```

```
        IF R1 = R0 GOTO end     * if not true, PC goes to next instruction
loop:   R4 = R4 + R2            * adds M2 to result
        R1 = R1 - 1             * decrements M1
        IF R1 != R0 GOTO loop   * add R2 to R4 R1 times (R4 = R2 . R1)
end:    MEM[result] = R4
```

↳ ex. max()

```
main:   R1 = MEM[V1]        ↳ V1: .word 3     result: .word 0
        R2 = MEM[V2]        ↳ V2: .word 5
        R3 = R1 < R2        * if true, R3 = 1 ; if not, R3 = 0
        IF R3 != R0 GOTO V2Max
        MEM[Result] = R1    * if R2 isn't max, then R1 is
        GOTO end
V2Max:  MEM[Result] = R2
end:    GOTO end
```