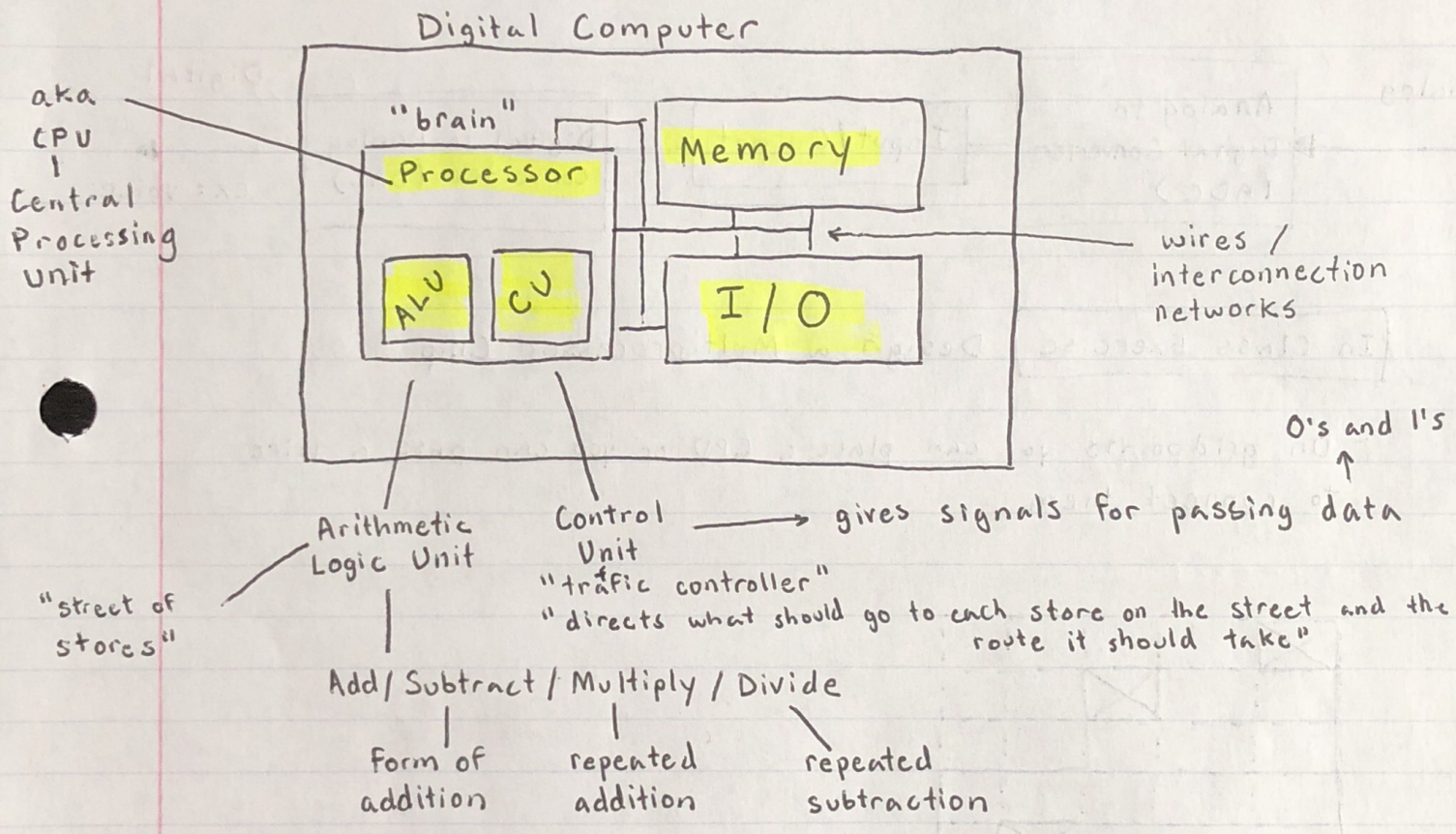
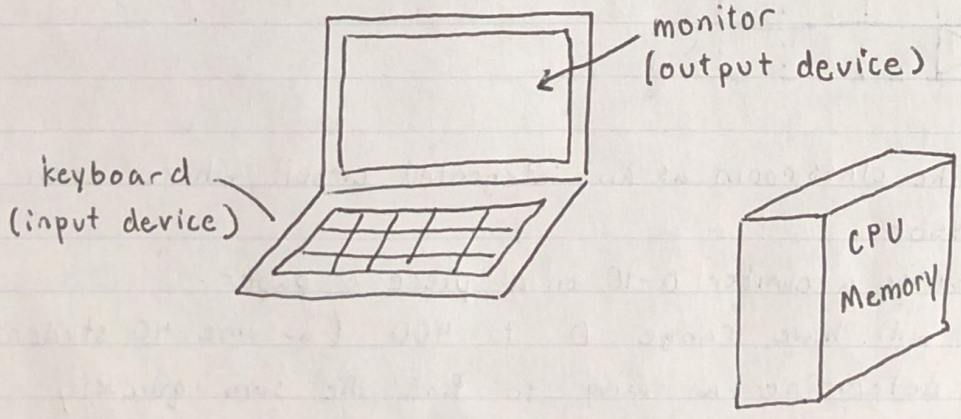
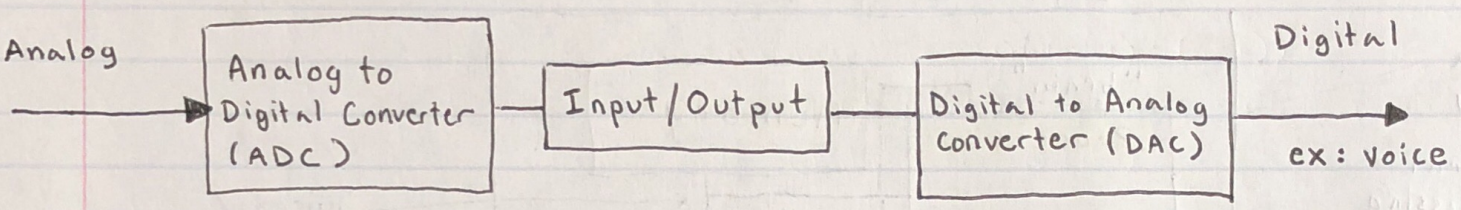
W3 L2 2-26-18 Binary Numbers, Logical Gates

Recap of Last Lec
- History of Computers video
  - Big / slow / expensive → smaller / faster / cheaper over time
- Number Systems (briefly)



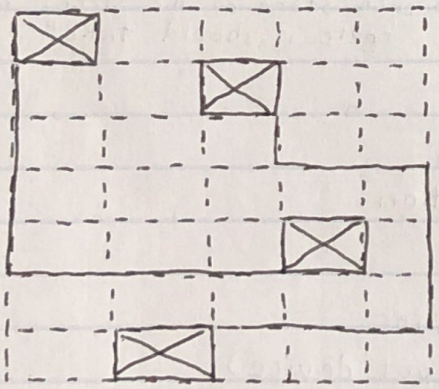


Computers
- General Purpose
- Embedded Systems — program is hard-wired
  - ex: microwave



In Class Exercise Design a Multiprocessor Chip

- On gridpoints you can place a CPU or you can pass a wire to connect them



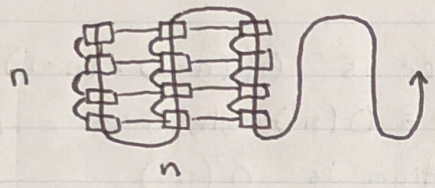
- Think of the classroom as an integrated circuit where each one of you is a processor
- Everyone write a number 0-10 on a piece of paper
- The sum will have range 0 to 400 (assume 40 students)

Problem: determine a way to find the sum quickly

- A slow method would be to visit each processor one at a time and add up the numbers. This would take 40 steps and only one processor would be active at a time.

- Now assume the grid is $n \times n$ processors w/ wires between adjacent processors
- To add n^2 numbers:

Algorithm 1



$O(n^2)$

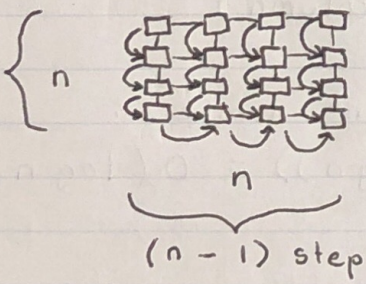
"as though you have a factory of 1,000 employees where only 1 works at a time"

Speedup = $\frac{\text{Unimproved time}}{\text{improved time}}$

time it would take w/ one CPU

Speedup = $\frac{O(n^2)}{O(n^2)} = 1$ → no speedup
→ bad algorithm

Algorithm 2



$O((n-1)+(n-1)) = O(2n-2) = O(n)$

"each row ~~acts~~ passes number to the row behind it at the same time. first row 1, then row 2, etc"

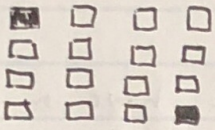
speedup = $\frac{O(n^2)}{O(n)} = n$ → slightly improved algorithm

"you hired 100 employees and got 10 times speed improvement"

Chip Diameter = shortest distance between two points that are farthest apart

→ lower bound on time complexity of solving any problem

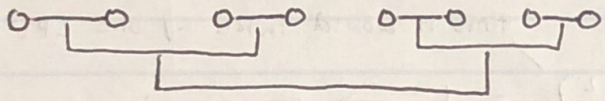
For previous architecture:



→ chip diameter is $O((n-1)+(n-1)) =$
$O(2n-2) = O(n)$ steps
∴ best solution is $O(n)$

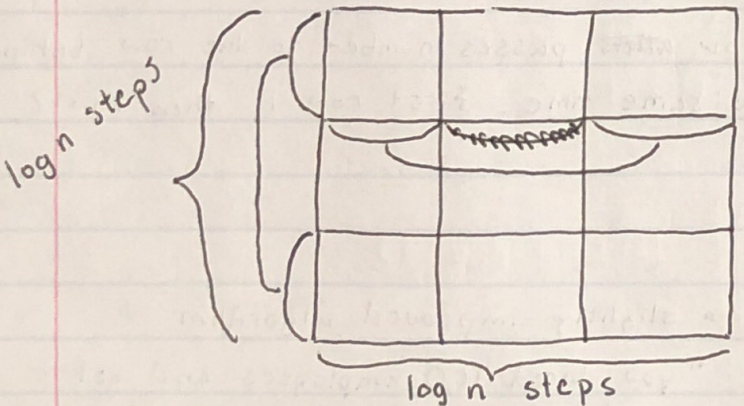
New Architecture

Stepping back from a computer architecture point of view, the fastest way to add n numbers is in log n steps with a binary tree



$\log_2 8 = 3$ steps

We can add binary tree wire interconnections to the grid of processors (to each row and column)



To add n^2 numbers:
$O(2(\log n)) = O(\log n)$

- Therefore it takes log n steps to add the numbers in one column
- speedup = $\frac{O(n^2)}{O(\log n)}$ → Is this optimal? Optimal would be $O(n^2)$, so we're off by a factor of $O(\log n)$, so not bad at all
- diameter = $O(2(\log n)) = O(\log n)$ → ∴ "diameter optimal"

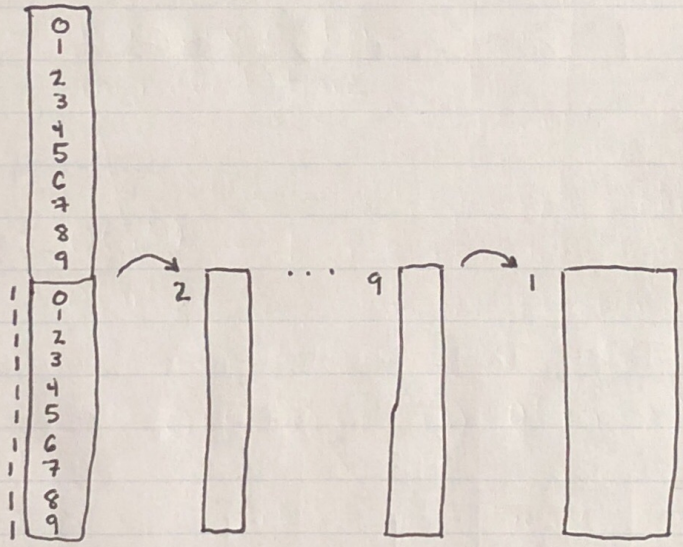
Importance of example on previous page: interconnection networks have a big impact on how Past things are done
high level / advanced perspective

Break

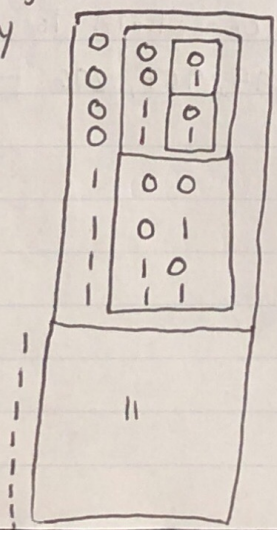
Number Systems

- Decimal 0, 1, 2, ..., 9
- Octal 0, ..., 7
- Binary 0, 1

Counting Up in Decimal



Counting Up in Binary



To represent up to n-1 in binary, need log n bits

Powers of Two

$2^1 = 2$
$2^2 = 4$
$2^3 = 8$
$2^4 = 16$
$2^5 = 32$
$2^6 = 64$
$2^7 = 128$
$2^8 = 256$
$2^9 = 512$
[$2^{10} = 1,024$ ← kilobyte
$2^{11} = 2,048$
$2^{12} = 4,096$
$2^{13} = 8,192$
$2^{14} = 16,384$
$2^{15} = 32,768$
$2^{16} = 65,536$
$2^{17} = 131,072$
$2^{18} = 262,144$
$2^{19} = 524,288$
[$2^{20} = 1,048,576$ ← megabyte

⋮

[$2^{30} = 1,073,741,824$ ← gigabyte

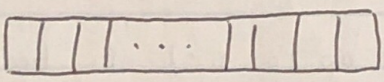
⋮

[$2^{40} = 1,099,511,627,776$ ← terabyte

⋮

[$2^{64} = 18,446,744,073,709,551,616$ ← bubbabyte

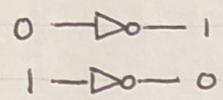
For A 32 bit register → largest # it can hold is $2^{32} - 1$
$= 4,294,967,296 - 1$



Basic Logic Gates for Binary Numbers

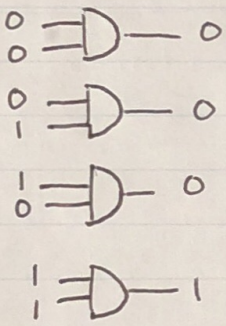
NOT Gate

inverts the input bit to an opposite bit



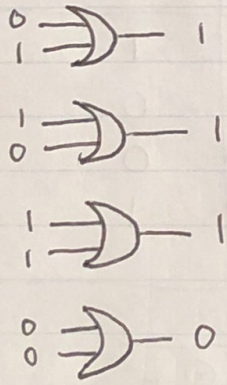
AND Gate $a \cdot b$

produces an output of 1 when both of its inputs are 1

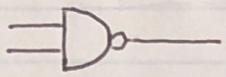


OR Gate $a + b$

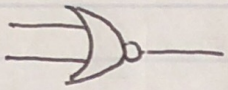
produces an output of 1 when one or more of its inputs are 1



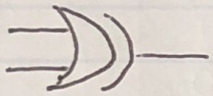
NAND Gate ("not and") $\overline{a \cdot b}$



NOR Gate ("not or") $\overline{a + b}$



XOR Gate ("exclusive or") $a \oplus b$



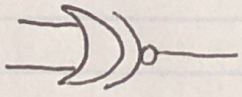
Truth Table for the Basic Logic Gates

		$a \cdot b$	$a + b$	$\overline{a \cdot b}$	$\overline{a + b}$	$a \oplus b$	$\overline{a \oplus b}$
b	a	AND	OR	NAND	NOR	XOR	XNOR
0	0	0	0	1	1	0	1
0	1	0	1	1	0	1	0
1	0	0	1	1	0	1	0
1	1	1	1	0	0	0	1

a	$\bar{a}$ NOT
0	1
1	0

XNOR Gate $\overline{a \oplus b}$

"exclusive nor"



Universal Set : set of logic gates which can be used to create any digital circuit; can implement NOT, OR, and AND

NOT
OR
AND
→ universal set

NAND → universal set

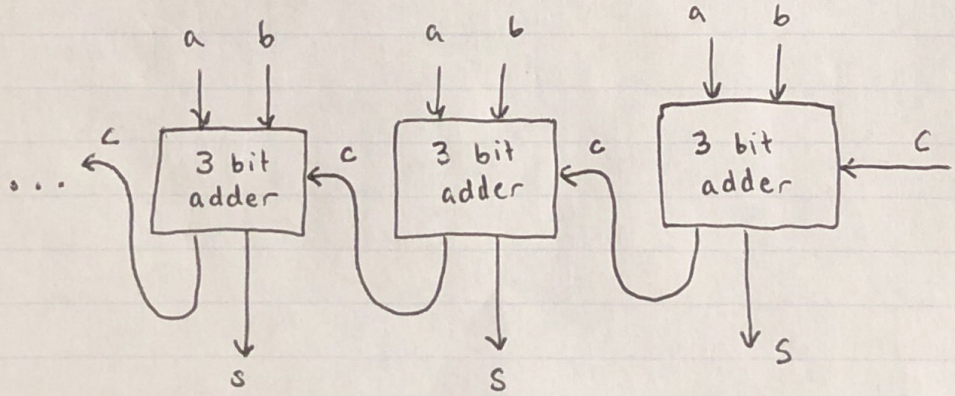
NOR → universal set

Simple Binary Addition

```
  1 1 0 1 0
  0 1 0 1 1
+ 1 1 0 1 0
-----------
1 0 0 1 0 1
```

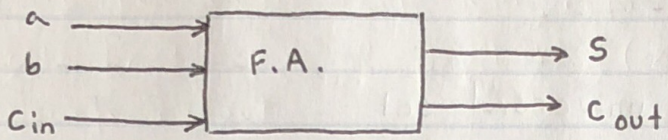
each column is a 3 bit adder

A 3 bit adder is called a Full Adder (F.A.)



× 6 (one for each column)

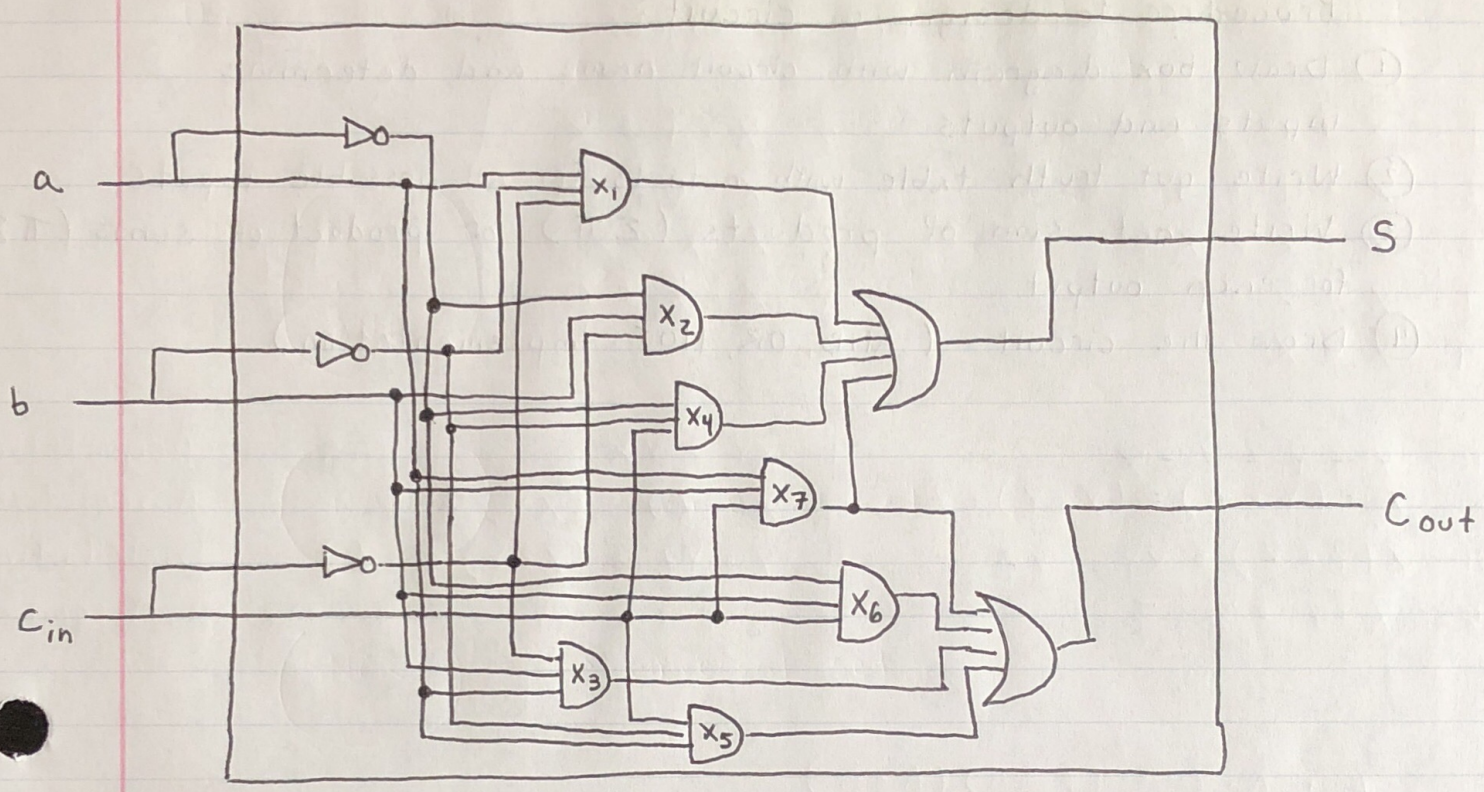
Example Design a 3 bit adder (aka Full Adder)



	C_{in}	b	a	S	C_{out}
x_0	0	0	0	0	0
x_1	0	0	1	1	0
x_2	0	1	0	1	0
x_3	0	1	1	0	1
x_4	1	0	0	1	0
x_5	1	0	1	0	1
x_6	1	1	0	0	1
x_7	1	1	1	1	1

$$Sum_{true} = x_1 + x_2 + x_4 + x_7$$
$$= \bar{C}_{in}\,\bar{b}\,a + \bar{C}_{in}\,b\,\bar{a} + C_{in}\,\bar{b}\,\bar{a} + C_{in}\,b\,a$$

$$C_{out\,true} = x_3 + x_5 + x_6 + x_7$$
$$= \bar{C}_{in}\,b\,a + C_{in}\,\bar{b}\,a + C_{in}\,b\,\bar{a} + C_{in}\,b\,a$$


a
b
C_{in}
X_1
X_2
X_4
X_7
X_6
X_3
X_5
S
C_{out}

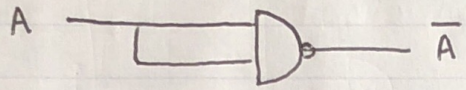
Procedure to design a circuit:

1. Draw box diagram with circuit name and determine inputs and outputs
2. Write out truth table with outputs for all possible inputs
3. Write out sum of products ($\Sigma\Pi$) or product of sums ($\Pi\Sigma$) for each output
4. Draw the circuit (AND, OR, NOT implementation)

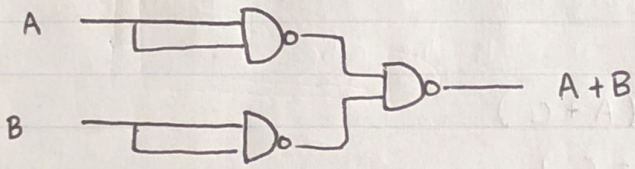
W4 L3 3-5-18 Boolean Algebra, Circuit Simplification

In Class Exercise — Prove NAND gate is universal (Gate Conversion)

NOT $\overline{A \cdot A} = \overline{A}$



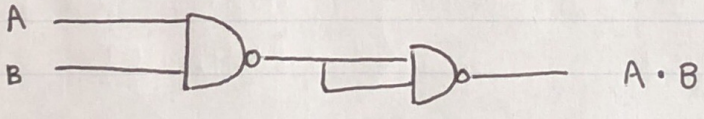
OR $\overline{(\overline{A \cdot A})(\overline{B \cdot B})} = \overline{\overline{A} \cdot \overline{B}} = A + B$



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

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

$= A \cdot B$



Another way to simplify this:

$\overline{(\overline{A \cdot B})(\overline{A \cdot B})} = \overline{\overline{(A \cdot B)}} = A \cdot B$

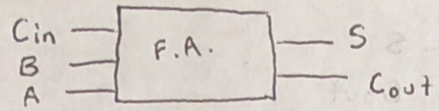
Basic Identities of Boolean Algebra

1.	$X + 0 = X$	2. $X \cdot 1 = X$	these are the "dual" of the column on the left
3.	$X + 1 = 1$	4. $X \cdot 0 = 0$	
5.	$X + X = X$	6. $X \cdot X = X$	
7.	$X + \overline{X} = 1$	8. $X \cdot \overline{X} = 0$	
9.	$\overline{\overline{X}} = X$		

10.	$X + Y = Y + X$	11. $XY = YX$	Commutative
12.	$X + (Y + Z) = (X + Y) + Z$	13. $X(YZ) = (XY)Z$	Associative
14.	$X(Y + Z) = XY + XZ$	15. $X + YZ = (X + Y)(X + Z)$	Distributive
16.	$\overline{X + Y} = \overline{X} \cdot \overline{Y}$	16. $\overline{X \cdot Y} = \overline{X} + \overline{Y}$	DeMorgan's

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

dual: switch the AND's to OR's and vice versa; switch the 1's to 0's and vice versa



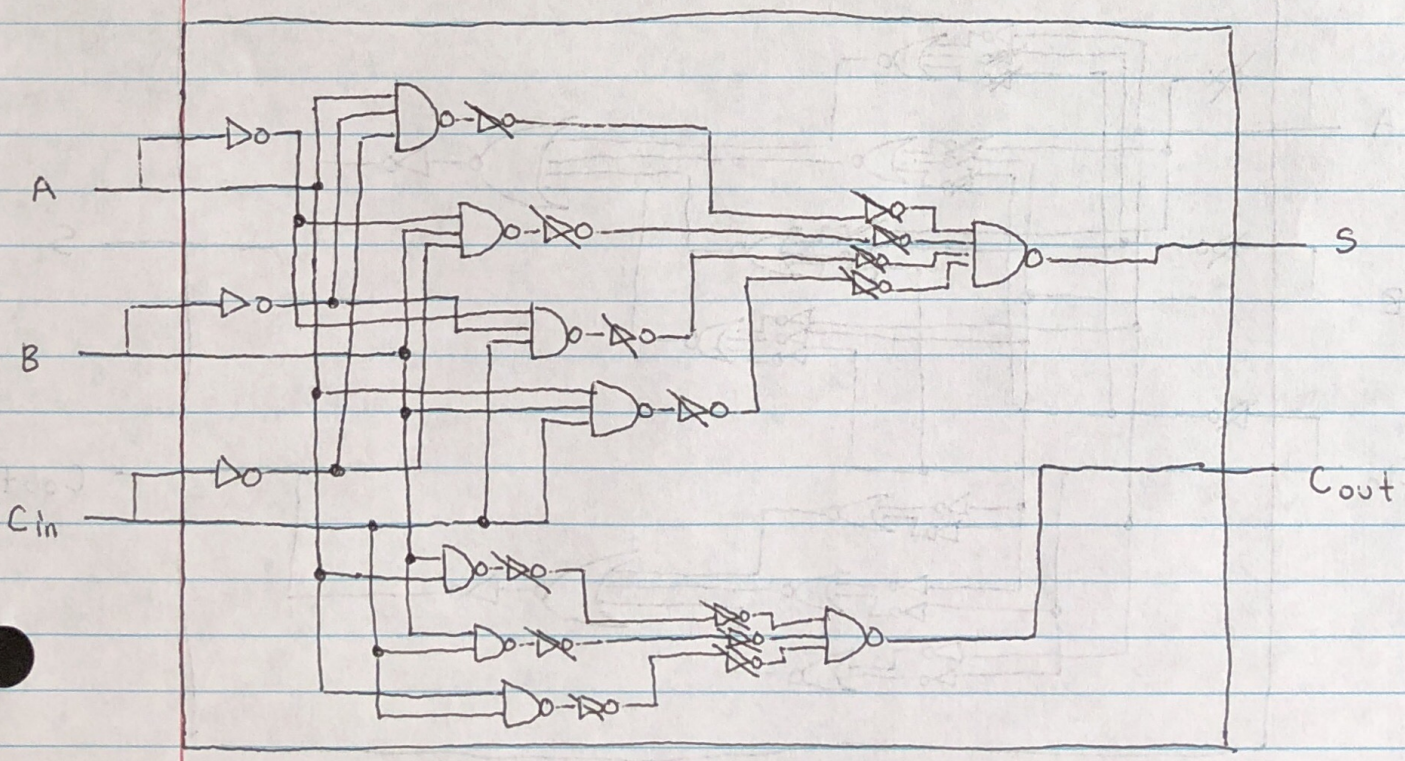
Design a Full Adder with NAND gates

Cin	B	A	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$S = \bar{C}_{in}\bar{B}A + \bar{C}_{in}B\bar{A} + C_{in}\bar{B}\bar{A} + C_{in}BA$

$C_{out} = \bar{C}_{in}BA + C_{in}\bar{B}A + C_{in}B\bar{A} + C_{in}BA$

$= AB + BC_{in} + AC_{in}$ by K-map minimization technique

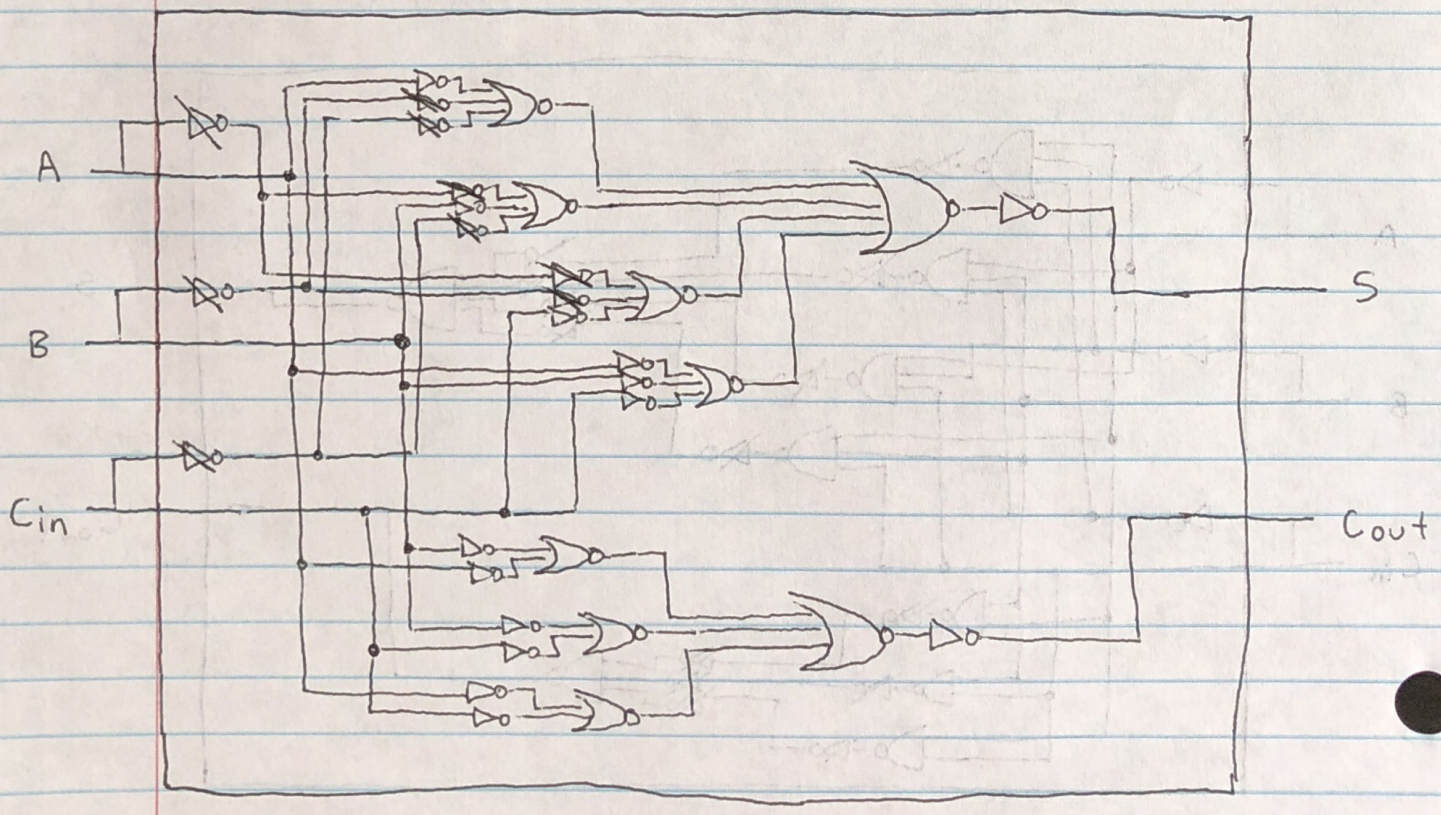


Design a Full Adder with NOR gates

Cin	B	A	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

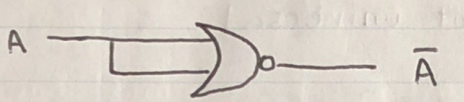
$$S = \bar{C}_{in}\,\bar{B}\,A + \bar{C}_{in}\,B\,\bar{A} + C_{in}\,\bar{B}\,\bar{A} + C_{in}\,B\,A$$

$$C_{out} = AB + BC_{in} + AC_{in}$$



In Class Exercise **Prove NOR gate is universal** (Gate Conversion)

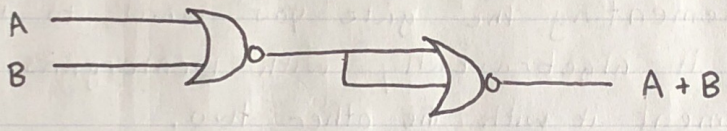
NOT $\overline{A + A} = \overline{A}$



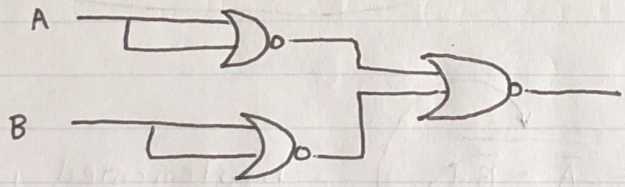
OR $\overline{(\overline{A+B}) + (\overline{A+B})} = \overline{\overline{(A+B)}\ \overline{(A+B)}}$

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

$= A + B$



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



To prove a gate is universal is to prove you can use one or more of that type of gate to implement NOT, AND, and OR.

XOR ⟶ not universal

AND ⟶ not universal

If you can implement OR, NOT or AND, NOT then you can also implement the third gate to make a universal set.

By double complementing the gate you want to implement, you can simplify it algebraically with DeMorgan's law to see how to implement it with the other two.

AND

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

implemented with OR, NOT

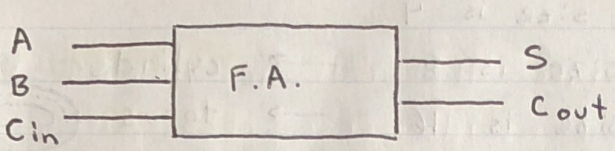
OR

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

implemented with AND, NOT

UNIVERSAL	NOT UNIVERSAL
AND, OR, NOT	XOR
NAND	AND
NOR	
NOT AND	
NOT OR	

Example Design a Full Adder



	Cin	B	A	S	Cout
m_0	0	0	0	0	0
m_1	0	0	1	1	0
m_2	0	1	0	1	0
m_3	0	1	1	0	1
m_4	1	0	0	1	0
m_5	1	0	1	0	1
m_6	1	1	0	0	1
m_7	1	1	1	1	1

sum of products is equal to product of sums

$\Sigma\Pi$	$\Pi\Sigma$
- solve for 1 to find minterms	- solve for 0 to find maxterms
- complement inputs equal to 0	- complement inputs equal to 1

$S = \Sigma\Pi = m_1 + m_2 + m_4 + m_7$ ← minterms

$= \bar{C}_{in}\bar{B}A + \bar{C}_{in}B\bar{A} + C_{in}\bar{B}\bar{A} + C_{in}BA$

$S = \Pi\Sigma = (M_0)(M_3)(M_5)(M_6)$ ← maxterms

$= (C_{in} + B + A)(C_{in} + \bar{B} + \bar{A})(\bar{C}_{in} + B + \bar{A})(\bar{C}_{in} + \bar{B} + A)$

$C_{out} = \Sigma\Pi = m_3 + m_5 + m_6 + m_7$ ← minterms

$= \bar{C}_{in}BA + C_{in}\bar{B}A + C_{in}B\bar{A} + C_{in}BA$

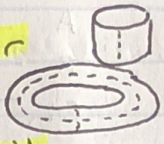
$C_{out} = \Pi\Sigma = (M_0)(M_1)(M_2)(M_4)$ ← maxterms

$= (C_{in} + B + A)(C_{in} + B + \bar{A})(C_{in} + \bar{B} + A)(\bar{C}_{in} + B + A)$

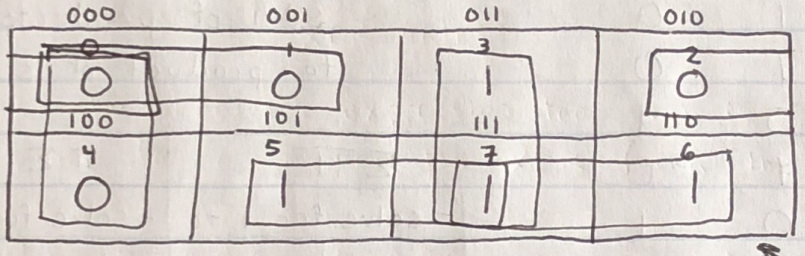
Note: If all boxes in a K-map are 1, then the function is equal to 1

K-map Minimization Technique

- one map for each output
- 2 variables → map size is 4
- 3 variables → map size is 8 → cylinder
  4 variables → map size is 16 → torus
- adjacent boxes have to differ by at most one bit
  ↳ reason for non-sequential ordering

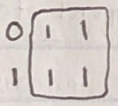


Carry (C_{out})



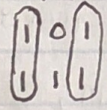
1's correspond to minterms

3,7 → BA



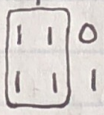
+

5,7 → $C_{in}A$



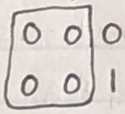
+

6,7 → $C_{in}B$



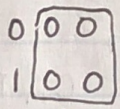
$$C_{out} = \Sigma\Pi = BA + C_{in}A + C_{in}B$$

0,1 → $C_{in}+B$



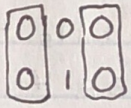
•

0,4 → $B+A$



•

0,2 → $C_{in}+A$



$$C_{out} = \Pi\Sigma = (C_{in} + B)(B + A)(C_{in} + A)$$

K-map Minimization Procedure

1. Draw map size based on number of input variables
2. Number the boxes (non-sequential b/c adjacent boxes can differ by maximum one bit)
3. Write out the binary equivalent of the box numbers
4. Fill in 1's corresponding to the minterms for this output
5. Identify rectangles of 1's that are powers of two in size, going from largest to smallest (16, 8, 4, 2, 1)
6. Do step 5 until all the boxes with 1's are covered
7. Write the expression for each rectangle (minterm) by comparing the bits in common between the boxes. When the bit in common is a 0, complement the input variable.

Sum (s)

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

↓

Nightmare case

↓

Can't minimize b/c no rectangles larger than one square

Don't Care Conditions : the output is neither 0 or 1; indicated by an X on the K-map

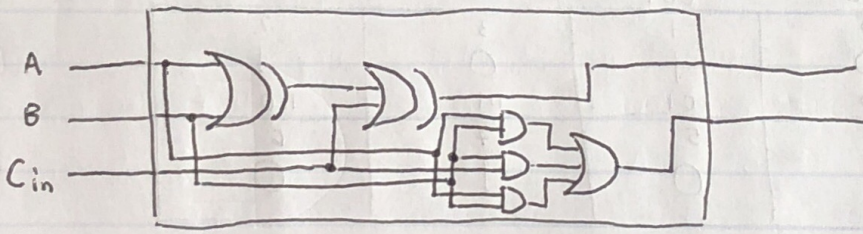
e.g. "if we had an adder that will not have a value higher than two, we wouldn't care about the higher order bits"

In Class Exercise Can you implement a full adder with XOR gates?

A: Yes, the boolean expression for sum is equal to the expression for a 3 variable XOR (aka the odd function).

$$S = A \oplus B \oplus C_{in} = A\bar{B}\bar{C}_{in} + \bar{A}B\bar{C}_{in} + \bar{A}\bar{B}C_{in} + ABC_{in}$$

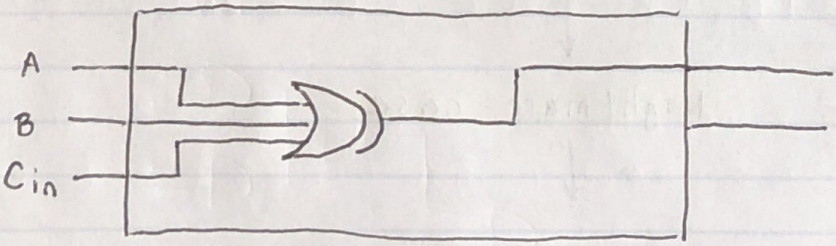
2-input XOR gates



$S = A \oplus B \oplus C_{in}$

$C_{out} = AB + BC_{in} + AC_{in}$

3-input XOR gates



Parity Bit : one way senders and receivers verify transmissions is by using a parity bit

- even parity → even number of 1's
- odd parity → odd number of 1's
- XOR is also used to generate the parity bit (XOR outputs 1 when the number of 1's is odd)

4 Variable K-map

0000	0001	0011	0010
0	1	3	2
0100	0101	0111	0110
4	5	7	6
1100	1101	1111	1110
12	13	15	14
1000	1001	1011	1010
8	9	11	10

Example Gate Propogation Delay of F.A. with 2-input XOR's vs. with 3-input XOR's

Assume all gates have the same amount of delay equal to 1.

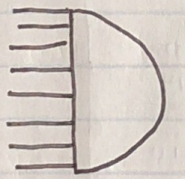
□ Why?

	Gate Propogation Delay
F.A. with 2-input XOR's	2
F.A. with 3-input XOR's	1

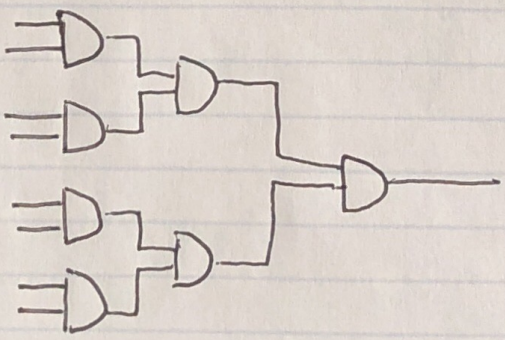
(just count the # of gates in shortest path to find delay)

Example More restrictions on the number of inputs to gates results in more levels of gates

e.g.



vs.



W5 L4 3-12-18 Ripple Carry Adder, Carry Look Ahead Adder, Subtractor

Example Sum of products is equal to product of sums

$$\Sigma\Pi = \Pi\Sigma$$

Analogy: Assume the colors of the rainbow are Red, Orange, Yellow, Green, Blue, Purple.

- To get into the class you have to wear either blue or green.
- You should not wear red and orange and yellow and purple.

↳ two ways of saying the same thing

$\Sigma\Pi = B + G$ ← equal to 1

$\Pi\Sigma = (R)(O)(Y)(P)$ ← equal to 0

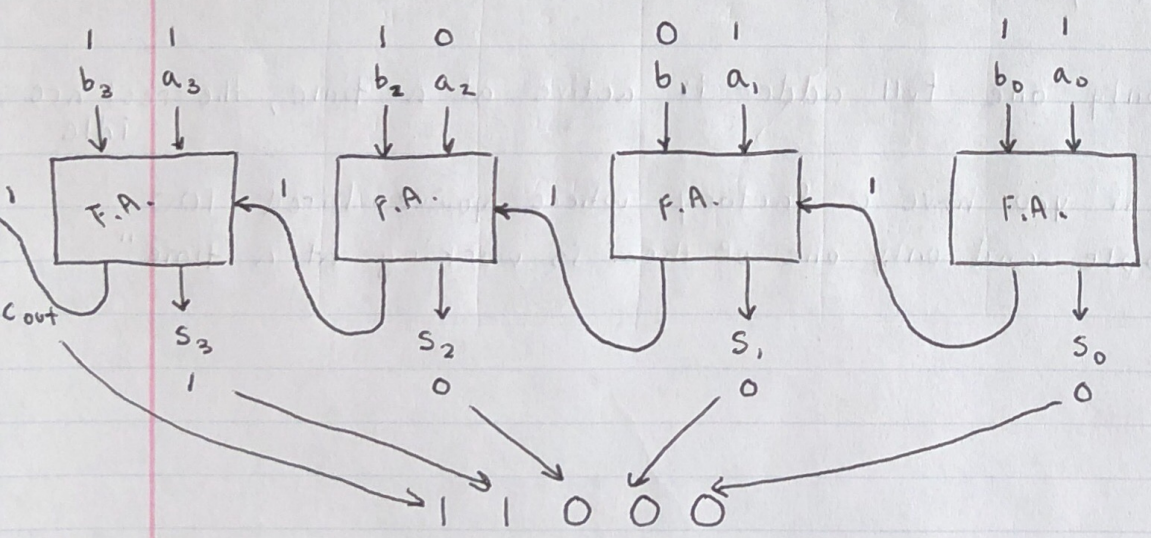
Ripple Carry Adder: add two n bit numbers using n Full Adders ("stupid adders")

$a_3\ a_2\ a_1\ a_0$

1 0 1 1 ← A

+ 1 1 0 1 ← B

$b_3\ b_2\ b_1\ b_0$



Ripple Carry Adder

- assume each adder has $O(1)$ cycles
  - ↳ $T = O(n)$ (time complexity)
- assume each adder takes $O(1)$ space
  - ↳ $S = O(n)$ (space complexity)
- Space time product
  - ↳ $ST = O(n^2)$
- Utilization factor
  - ↳ only $\frac{1}{n}$ $\left(\text{total work} = \frac{n}{n^2} = \frac{1}{n}\right)$
    - ↳ only one full adder is active at a time, the rest are idle
    - ↳ "as if you have a factory where you've hired 100 people and only one of them is working at a time"

□ Why?

Subtraction Using 2s Complement (Subtraction by method of addition) (assumes regular binary representation)

Regular Algebra $A - B = A + (-B)$

Binary $A - B = A + (2\text{'s complement of } B)$

2's Complement of B = 1's Complement of B + 1

flip the bits

Example $5 - 2 = 5 + (-2)$

$5 \rightarrow 101$

$2 \rightarrow 010$ Binary

$\downarrow$

$10'1$ flip the bits

$+ \quad 1$ add 1

110 2's Complement

$$\begin{array}{r} 1101 \\ +\ 110 \\ \hline (1)011 \end{array}$$

ignore the carry

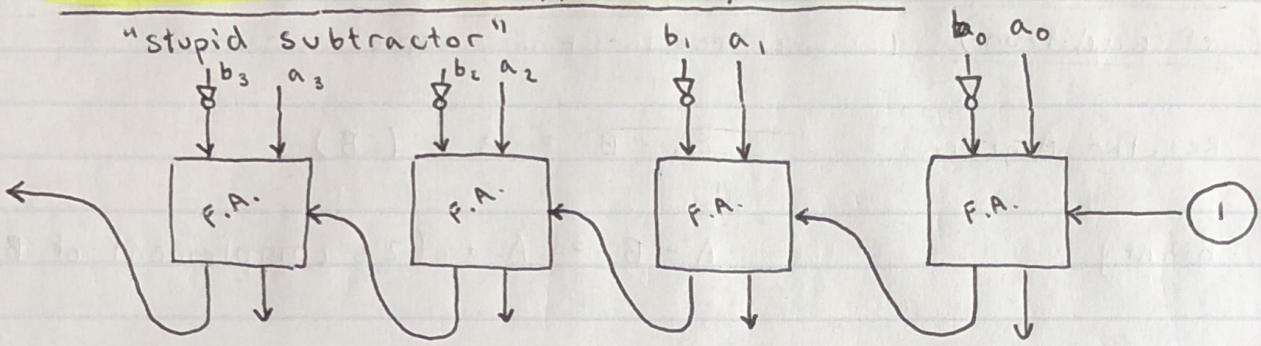
$\downarrow$

solution $\longrightarrow 011 = 3$

Steps for Subtraction Using 2's Complement

1. Add A to 2's complement of B
2. Ignore the carry

Subtractor (from Ripple Carry Adder)



- By complementing each of the b bits and adding an initial carry of one, we can turn the ripple carry adder into a subtractor
- Complementing the b bits and adding the initial carry of 1 is equivalent to taking the 2's complement of the b operand

2's Complement Representation vs. Regular Binary Representation

$0 + 2^2 = 0$

	Binary			2's Complement	
	2^2	2^1	2^0		
0 ←	0	0	0	0	
1 ←	0	0	1	1	
2 ←	0	1	0	2	
3 ←	0	1	1	3	
4 ←	1	0	0	-4	$= -1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
5 ←	1	0	1	-3	$= -1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$
6 ←	1	1	0	-2	$= -1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$
7 ←	1	1	1	-1	$= -1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$

$-1 \times 2^2 = -4$

- For 2's Complement representation

sign bit
- IF Most Significant Bit is 0 → number is positive
- IF Most Significant Bit is 1 → number is negative
- IF number is negative, add -1×2^{MSB} when summing the values of the digits
- Take the 2's Complement of a number in 2's Complement form to get the regular binary representation
- Technically also do this for positive numbers: $-0 \times 2^{MSB} = 0$

IF you add two positive numbers and the result is negative → Overflow

IF you add two negative numbers and the result is positive → Overflow

Data Range of 2's Complement Numbers

	Regular Binary	2's Complement
3 bits	$0 \rightarrow 2^3 - 1$	$-2^{3-1} \rightarrow 2^{3-1} - 1$
	$0 \rightarrow 7$	$-4 \rightarrow 3$
n bits	$0 \rightarrow 2^n - 1$	$-2^{n-1} \rightarrow 2^{n-1} - 1$
4 bits	$0 \rightarrow 2^4 - 1$	$-2^{4-1} \rightarrow 2^{4-1} - 1$
	$0 \rightarrow 15$	$-8 \rightarrow 7$

can derive these equation by writing out the truth table

- 2's complement data range is roughly half of binary data range for the same number of bits, except it has both a positive and negative range

- In order to not have overflow, a binary arithmetic result must be within the data range for a given number of bits and given number representation format

Subtraction Using 2's Complement also works when the numbers are in 2's Complement format

$$2\text{'s } A - 2\text{'s } B = 2\text{'s } A + (2\text{'s } (2\text{'s } B))$$

Placeholder Digits for 2's Complement Numbers

Zero Fill

Decimal → can always add 0's

$248 = 0248 = 00248 = 00\ldots0248$

Binary → can also just add 0's

$100 = 0100 = 00100 = 00\ldots0100$

$\downarrow$

4

Sign Extension

2's Complement Representation Negative Numbers → you can't just add 0's, but you can add 1's

$100 = -4 = 1100 = 11100 = 11\ldots1100$

$\downarrow$

$100 = -4 + 0 + 0 = -4$

$1100 = -8 + -4 + 0 + 0 = -4$

$11\,100 = -16 + 8 + 4 + 0 + 0 = -4$

2's Complement Representation Positive Numbers → same as regular binary so you can just add 0's

Example Exam Question Design an adder capable of adding these two numbers (or this range). The numbers are represented in 2's Complement format.

How to find the 2's Complement of a binary number

(e.g. when doing Subtraction using 2's Complement)

1. Take the 1's Complement (flip all the bits)
2. Add 1

How to read a number in 2's Complement Format

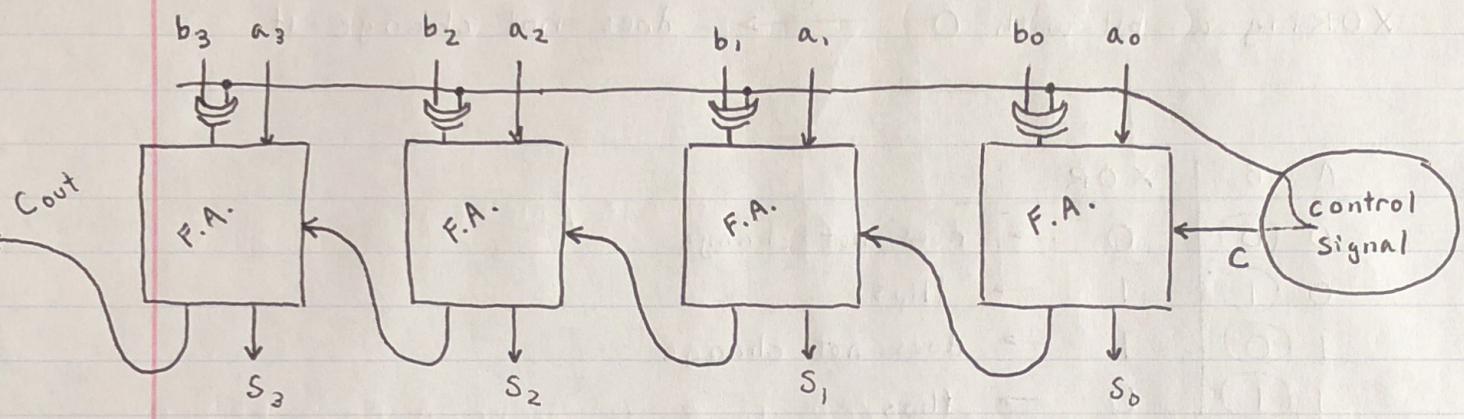
1. Sum the values of each bit position as if it were a regular binary number, except add -1 times the value of the Most Significant Bit (MSB) position.

$$-1 \times \_ \times 2^{MSB} + \ldots + \_ \times 2^2 + \_ \times 2^1 + \_ \times 2^0$$

Why K-maps use non-sequential ordering for the boxes

The box numbers for adjacent boxes can differ by at most one bit, if used sequential order would have boxes that differ by two bits.

In Class Exercise Take the Ripple Carry Adder and make it do both addition and subtraction.



When the control signal is 1, it adds 1 as well as complements the b bits of the b operand. This is because XORing a bit with 1 flips it. In this case the circuit performs subtraction.

When the control signal is 0, the circuit has no initial carry and the control signal does not change the b bits of the b operand. This is because XORing a bit with 0 does not change it. In this case the circuit performs addition as if it were just a basic Ripple Carry Adder.

This example is akin to signals sent by the control unit in the processor.

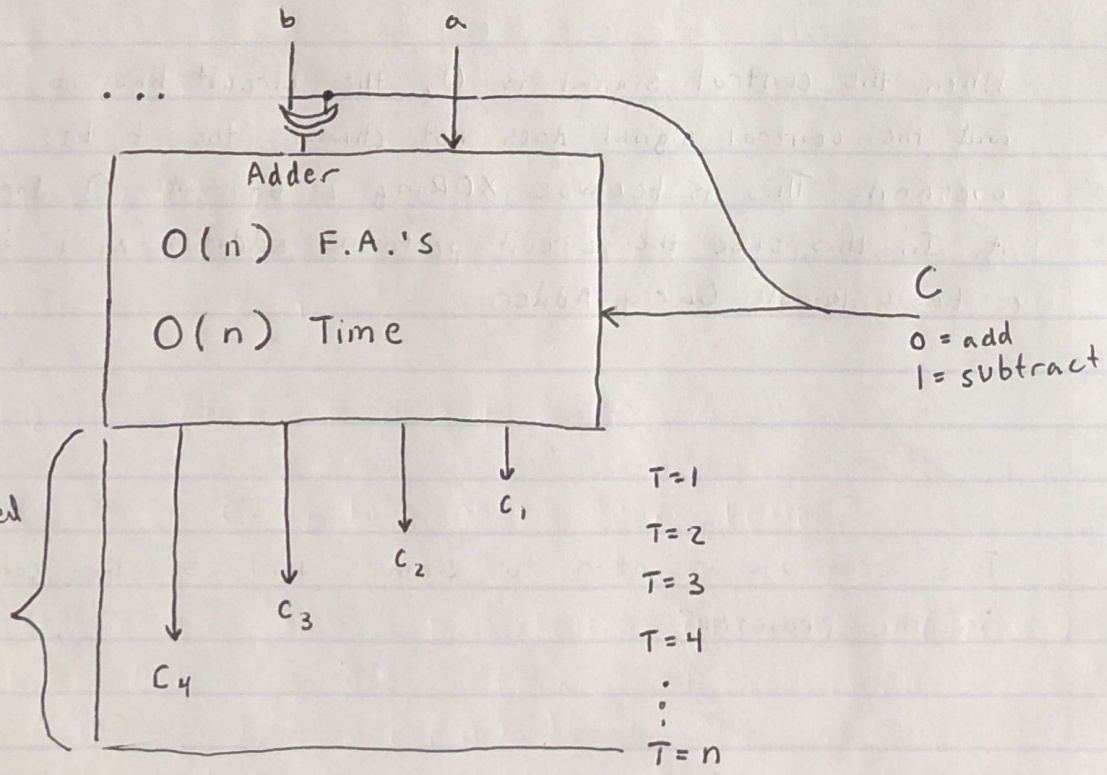
Properties of XOR

XORing a bit with 1 ⟶ flips the bit (complements it)

XORing a bit with 0 ⟶ does not change it

A	B	XOR	
0	0	0	→ does not change
0	1	1	→ flips
1	0	1	→ does not change
1	1	0	→ flips

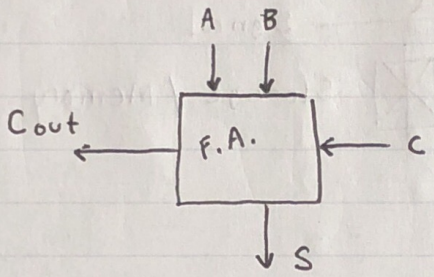
Time Complexity of Ripple Carry Adder



Carry Look Ahead Adder

Goal : generate all carries at the same time (in parallel)

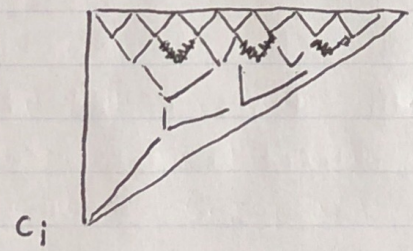
- lower bound $O(\log n)$ time



$$C_{out\ i+1} = \underbrace{a_i b_i}_{\text{Generate}} + \underbrace{c_i (a_i + b_i)}_{\text{Propogate}}$$

Don't have to know for exam

try to generate them at the same time then combine them



$O(\log n)$ steps

$O(\log n)$ Time

What to know about Carry Look Ahead Adders

$T = O(\log n)$

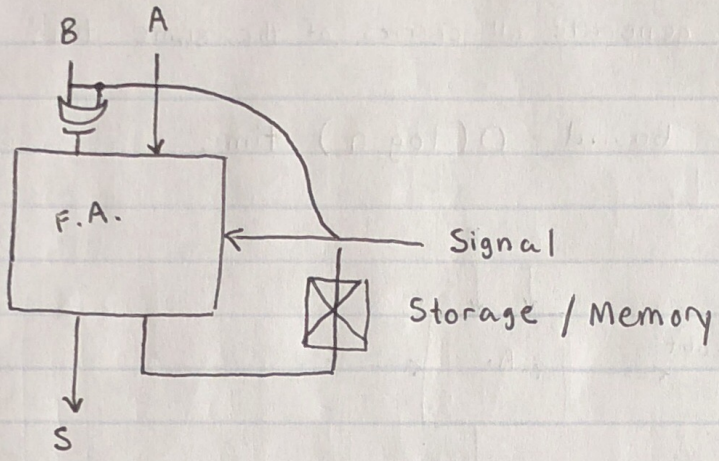
"generate partial carries simultaneously and then they collectively come up with the final carry"

$Space = S = O(n)$

$Space\ Time\ Product = ST = O(n \log n)$

$Total\ Work = \frac{n}{n \log n} = \frac{1}{\log n} = Utilization\ Factor$

Single Adder



- can add signal and XOR to B to make it do subtraction too

$$T = O(n)$$

$$S = O(1)$$

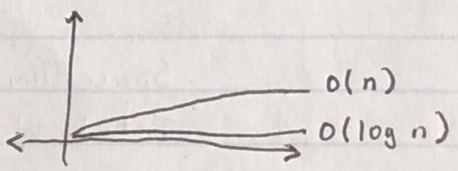
$$ST = O(n)$$

Total Work = n

$$\text{Utilization Factor} = \frac{n}{n} = 1 = 100\%$$

Is this better or is Carry Look Ahead Adder better?

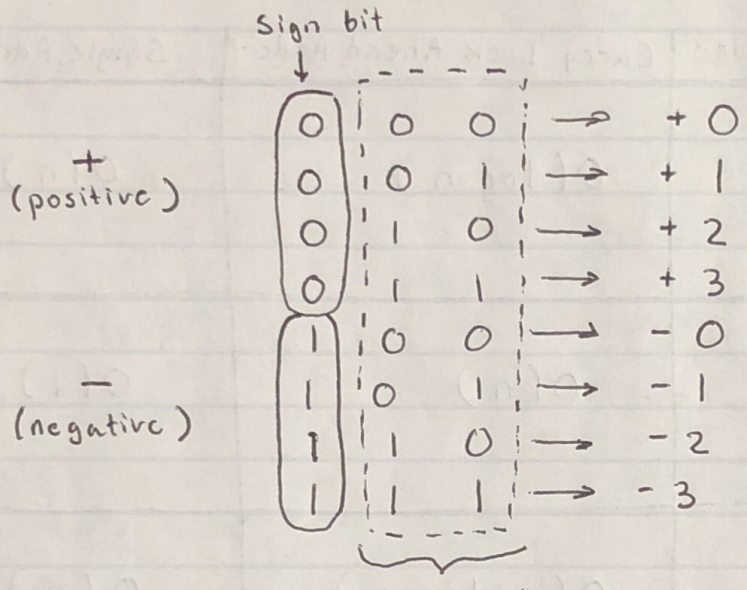
Even though Single Adder has better utilization (100%), Carry Look Ahead is ~~better because it~~ is much faster ($O(\log n)$ vs $O(n)$).
So CLA is more time efficient while the Single Adder is more space efficient.



Comparison of Adders

	Ripple Carry Adder	Carry Look Ahead Adder	Single Adder (with memory)
Time Complexity	$O(n)$	$O(\log n)$	$O(n)$
Space Complexity	$O(n)$	$O(n)$	$O(1)$
Space / Time Product	$O(n^2)$	$O(n \log n)$	$O(n)$
Total Work	n	n	n
Utilization Factor	$\frac{n}{n^2} = \frac{1}{n}$	$\frac{n}{n \log n} = \frac{1}{\log n}$	$\frac{n}{n} = 1$ (100%)
Speedup	$\frac{O(n)}{O(n)} = 1$ (no speedup)	$\frac{O(n)}{O(\log n)} = \frac{n}{\log n}$	$\frac{O(n)}{O(n)} = 1$ (no speedup)

Signed Magnitude Representation



- two values for 0 (+0, −0)

Adding a Control Signal to an Adder

This can be done with any of the types of adders to make it do both addition and subtraction.

Signed Binary Numbers

Decimal	Signed 2's Complement	Signed Magnitude
+7	0 1 1 1	0 1 1 1
+6	0 1 1 0	0 1 1 0
+5	0 1 0 1	0 1 0 1
+4	0 1 0 0	0 1 0 0
+3	0 0 1 1	0 0 1 1
+2	0 0 1 0	0 0 1 0
+1	0 0 0 1	0 0 0 1
+0	0 0 0 0	0 0 0 0
-0	—	1 0 0 0
-1	1 1 1 1	1 0 0 1
-2	1 1 1 0	1 0 1 0
-3	1 1 0 1	1 0 1 1
-4	1 1 0 0	1 1 0 0
-5	1 0 1 1	1 1 0 1
-6	1 0 1 0	1 1 1 0
-7	1 0 0 1	1 1 1 1
-8	1 0 0 0	—

W6 L5 3-19-18 Decoders, Encoders, Multiplexers, DeMux, Comparators

Today

- Comparators
- Decoders
- Encoders
- Multiplexers
- Demultiplexers

Comparators

Suppose you want to compare two decimal numbers. How would you do it?

A = 5 1 8 6
B = 1 3 9 8
5 > 1
A > B

A = 5 1 3
B = 5 1 8
5=5 1=1 3<8
B > A

A: Compare each digit starting with the most significant one.

The same method would work for ~~[illegible]~~ comparing two binary numbers.

A = 1 1 0 1 1
B = 1 1 0 0 1
1=1 1=1 0=0 1>0
A > B

Algorithm :

Start from the most significant bit

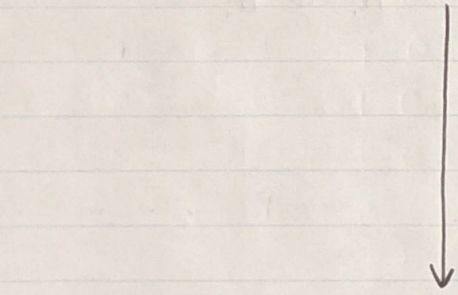
Compare A_n , B_n

If $A_n > B_n$ then $A > B$

Else If $B_n > A_n$ then $B > A$

Else $B_n = A_n$

↳ Decrement the index and continue the loop (compare the next bit)



We want to do this in hardware (with gates)

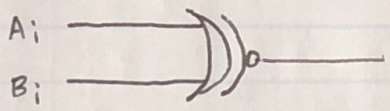
We can convert our algorithm into gates by writing out a truth table and Boolean expression for each possible result of comparing two bits in the algorithm:

- $A_i = B_i$
- $A_i > B_i$
- $A_i < B_i$

$A_i = B_i$

A_i	B_i	$A_i = B_i$	same as →	$\overline{XOR}$ = XNOR
0	0	1		1
0	1	0		0
1	0	0		0
1	1	1		1

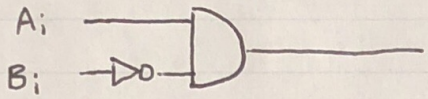
$$A_i = B_{i\ true} = \overline{A_i}\,\overline{B_i} + A_i B_i = \overline{A_i \oplus B_i}$$



$A_i > B_i$

A_i	B_i	$A_i > B_i$
0	0	0
0	1	0
1	0	1
1	1	0

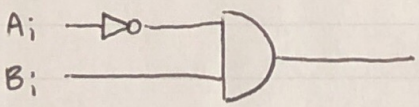
$A_i > B_{i\ true} = A_i \overline{B_i}$



$A_i < B_i$

A_i	B_i	$A_i < B_i$
0	0	0
0	1	1
1	0	0
1	1	0

$A_i < B_{i\ true} = \overline{A_i} B_i$



Example **Design a 3 bit comparator**

Assume we're comparing $A_3 A_2 A_1$ vs $B_3 B_2 B_1$

$$A > B = \underbrace{A_3 \bar{B}_3}_{\text{1st bit greater}} + \underbrace{(\overline{A_3 \oplus B_3}) A_2 \bar{B}_2}_{\text{2nd bit greater}} + \underbrace{(\overline{A_3 \oplus B_3})(\overline{A_2 \oplus B_2}) A_1 \bar{B}_1}_{\text{3rd bit greater}}$$

$$A < B = \bar{A}_3 B_3 + (\overline{A_3 \oplus B_3}) \bar{A}_2 B_2 + (\overline{A_3 \oplus B_3})(\overline{A_2 \oplus B_2}) \bar{A}_1 B_1$$

$$A = B = (\overline{A_3 \oplus B_3})(\overline{A_2 \oplus B_2})(\overline{A_1 \oplus B_1})$$

$T = O(\log n)$

↓

This is because we do the comparison for each bit one at a time

↓

e.g. for $n = 1000$ it would take $\log_2 1000 = 10$ steps because you need 10 bits to represent 1000 in binary ($2^{10} = 1024$)

Algorithmic Design of a Comparator

This example is significant because it is the first time we've designed a circuit algorithmically. In contrast, if we had used our previous method of writing out all possible inputs & outputs in a truth table, for a 6 input truth table we would've had $2^6 = 64$ rows, and writing out sum of products / minimizing the expressions would be a nightmare.

e.g. 6 inputs

B_3 B_2 B_1	A_3 A_2 A_1	A > B	A < B	A = B
0 0 0	0 0 0	0	0	1
0 0 0	0 0 1	1	0	0
0 0 0	0 1 0	1	0	0
0 0 0	0 1 1	1	0	0
⋮	⋮		⋮	

$2^6 = 64$ rows

The truth table method of writing out all inputs/outputs works for any problem. But for large numbers of inputs and outputs the truth table could be huge which is why we want to do the 9th algorithmic method.

In Class Exercise Design a 2 bit comparator (algorithmically)

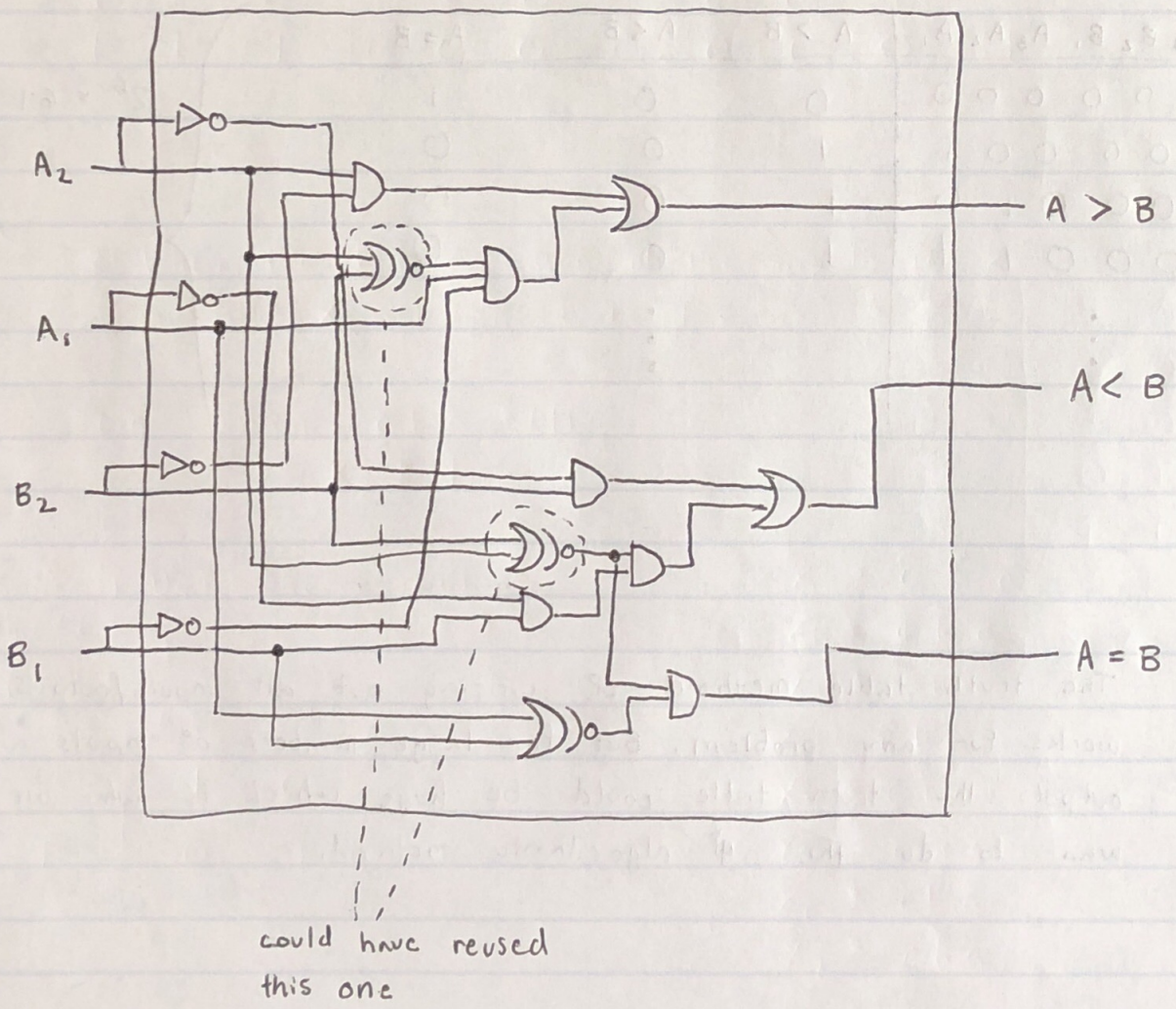
Assume we're comparing A_2A_1 vs B_2B_1

Using the same algorithm we developed for the 3 bit comparator:

$$A > B = A_2\overline{B_2} + \overline{(A_2 \oplus B_2)}\, A_1\overline{B_1}$$

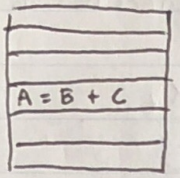
$$A < B = \overline{A_2}\, B_2 + \overline{(B_2 \oplus A_2)}\, \overline{A_1}\, B_1$$

$$A = B = \overline{(A_2 \oplus B_2)}\,\overline{(A_1 \oplus B_1)}$$



Where a Decoder is used

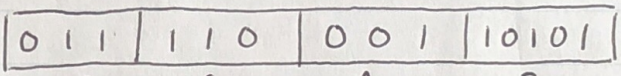
High Level Program



Assembly Language

ADD A, B, C

Machine Code



decoded by a Decoder

[redacted] generates signal

i.e.

1 = subtract

0 = add

Decoder : a decoder has $\log n$ bits as input and generates n outputs

"decodes an input of 1's and 0's and generates a signal"

$\log n = \log 8 = 3$ inputs; $n = 8$ outputs

c	b	a	x_0	x_1	x_2	x_3	x_4	x_5	x_6	x_7
0	0	0	1	0	0	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0
0	1	0	0	0	1	0	0	0	0	0
0	1	1	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	1	0	0
1	1	0	0	0	0	0	0	0	1	0
1	1	1	0	0	0	0	0	0	0	1

When the binary value of the inputs is k, output x_k is equal to 1

e.g. when $c\,b\,a = 011$,
$x_3 = 1$

Control Unit : decoder with $\log 2 = 1$ input and $n = 2$ outputs

(ex) z	Add	Subtract
0	1	0
1	0	1

$x_0 = \bar{c}\,\bar{b}\,\bar{a}$
$x_1 = \bar{c}\,\bar{b}\,a$
$x_2 = \bar{c}\,b\,\bar{a}$
$x_3 = \bar{c}\,b\,a$
$x_4 = c\,\bar{b}\,\bar{a}$
$x_5 = c\,\bar{b}\,a$
$x_6 = c\,b\,\bar{a}$
$x_7 = c\,b\,a$

Encoder: the input and output of a Decoder are switched

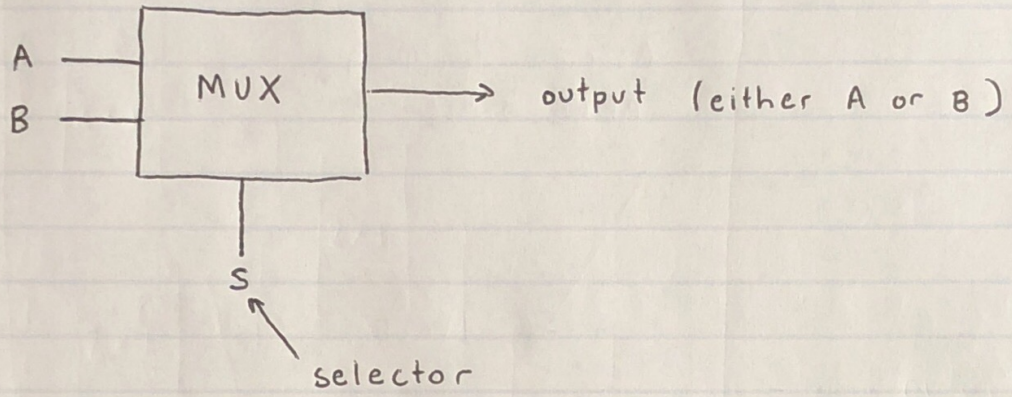
i.e. solve for when c, b, a are true

- Prof said don't worry about this one as much

Multiplexer : a MUX is a selector, it selects one of the inputs to become output; it selects one out of n inputs to become output, needs log n selector bits

aka MUX

(ex) 2 input MUX (one selector bit)



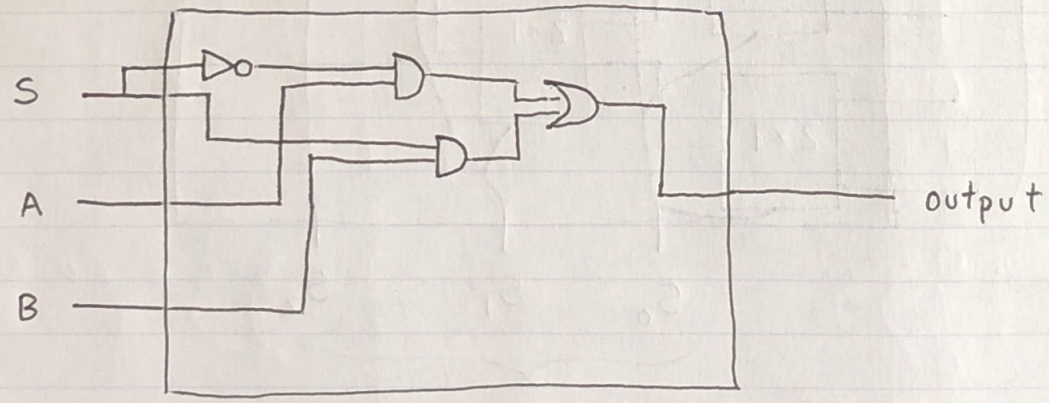
- If the selector is 0, then the output is A
- If the selector is 1, then the output is B

- With a MUX you can draw the truth table for anything and put the output as input to the MUX, but it uses a lot of resources

In class Exercise — Design the circuit for a MUX

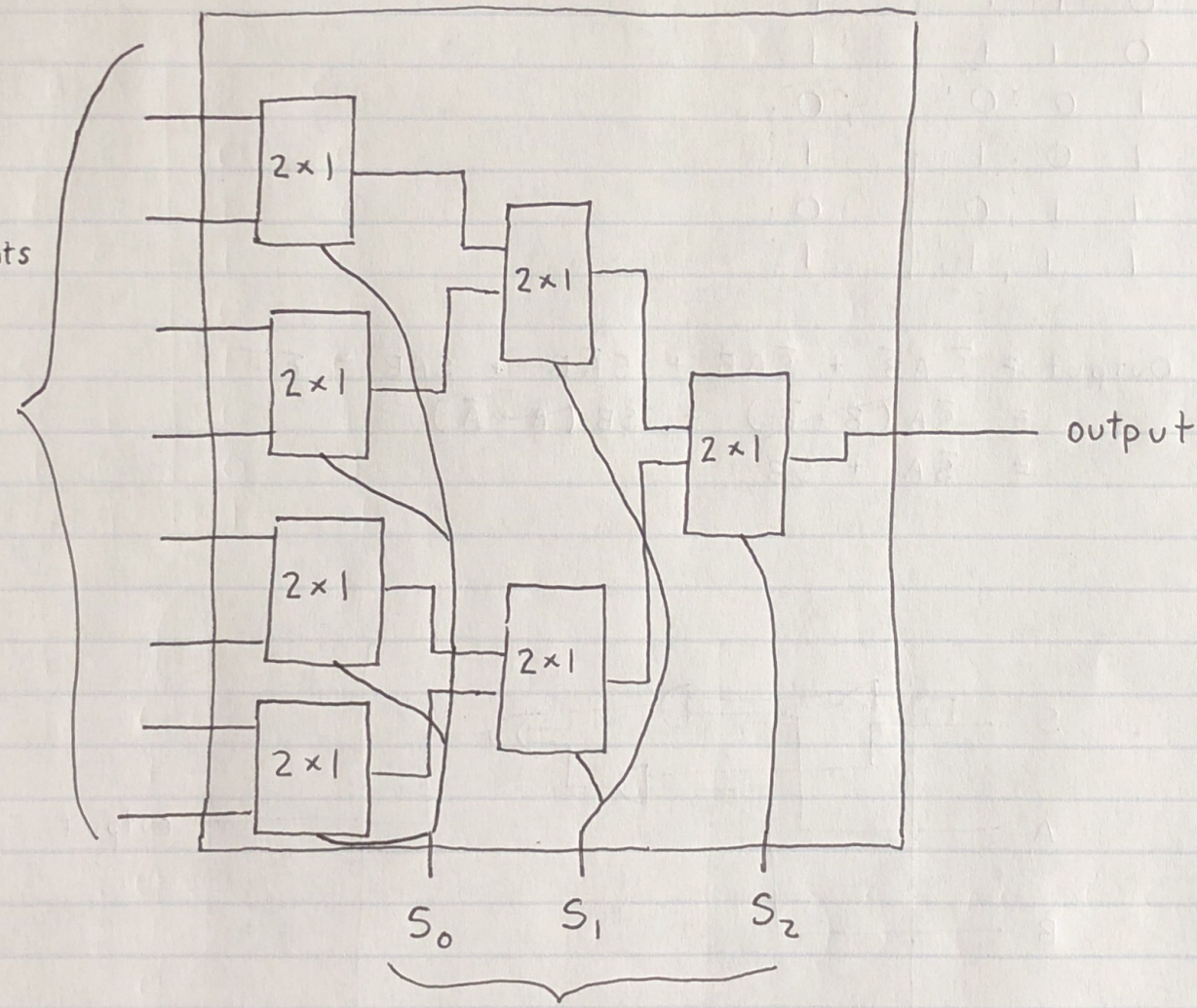
S	A	B	Output
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

$$\text{Output} = \bar{S}A\bar{B} + \bar{S}AB + S\bar{A}B + SAB = \Sigma\Pi$$
$$= \bar{S}A(\underbrace{B + \bar{B}}_{1}) + SB(\underbrace{A + \bar{A}}_{1})$$
$$= \bar{S}A + SB$$



We can use 2×1 MUX's to design a larger MUX

8 × 1 MUX



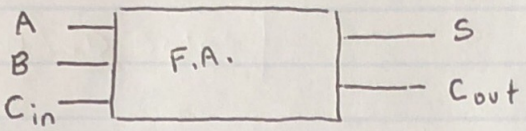
log n = log 8 = 3 selector bits

- Each selector bit is connected to a level of MUX's

Demultiplexer: the input and output of a Multiplexer are switched

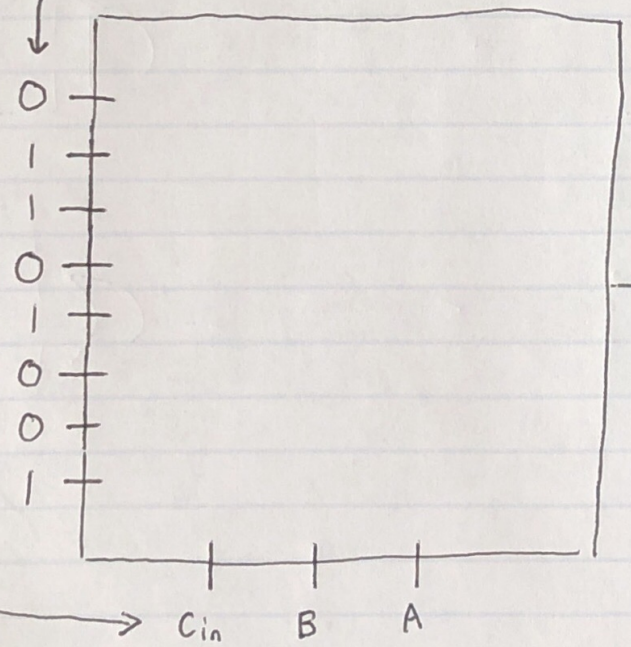
aka DeMux "broadcast"

Example Design a Full Adder with MUX's

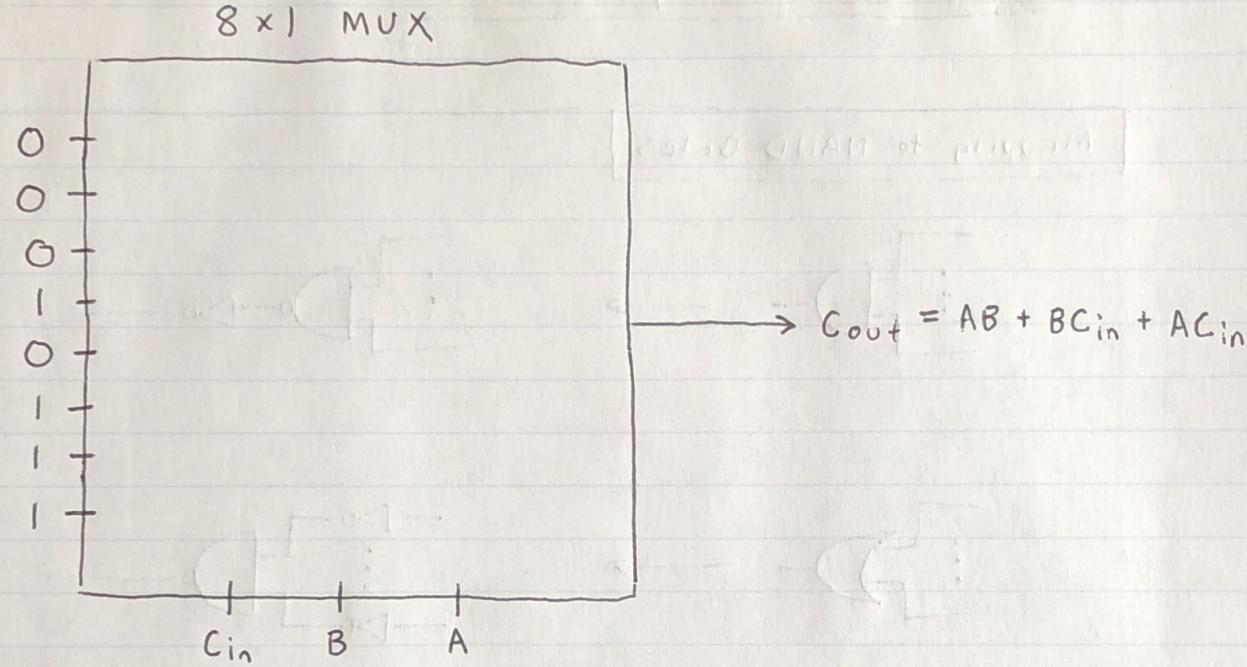


Cin	B	A	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

8 × 1 MUX



$S = A \oplus B \oplus C_{in}$

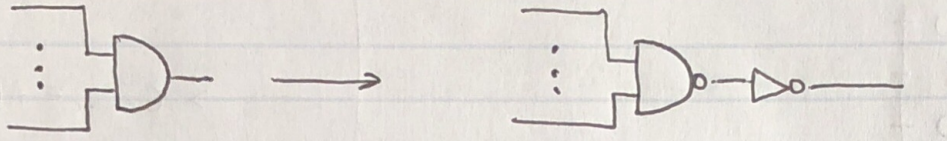


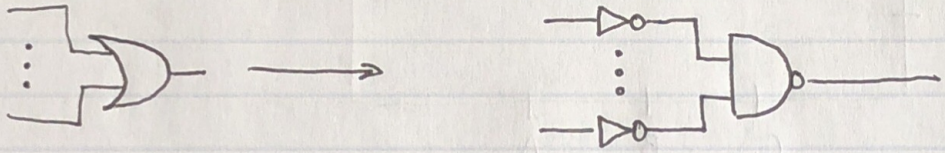
- Need a MUX for each output
- Use the 8 output values as inputs for the MUX
- Use the 3 inputs as selector bits for the MUX
- The MUX then "selects" an output for a given input

Although it is easy to implement the circuit with MUX's, it is not hardware optimal because it requires a lot of gates.

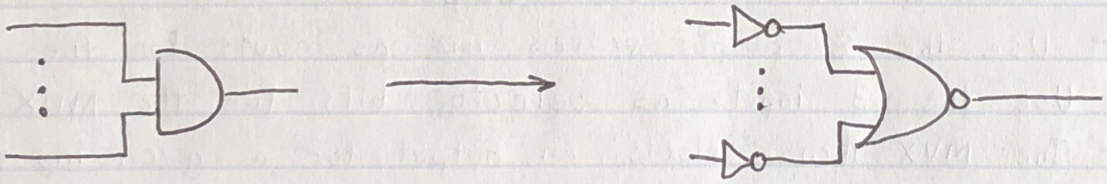
Gate Replacement / Conversion

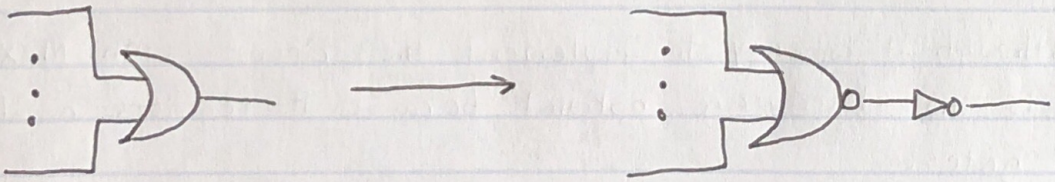
Mapping to NAND Gates





Mapping to NOR Gates





Comparison of Component Types

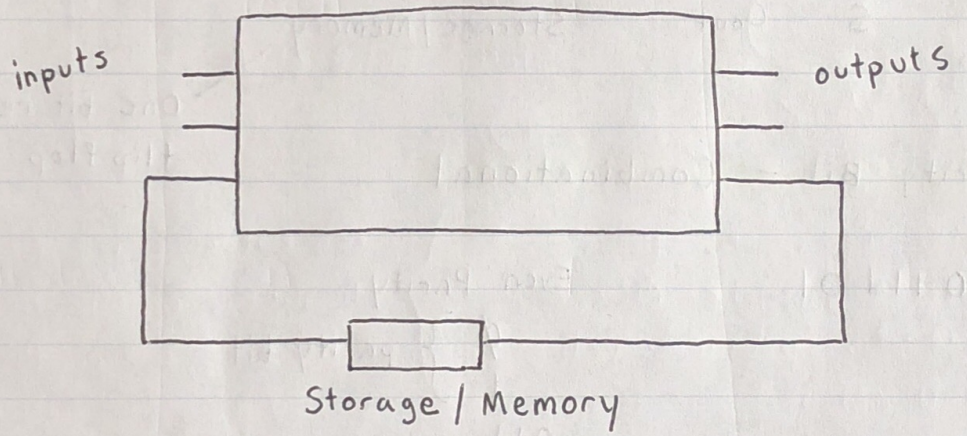
	# Inputs	# Outputs	# Selector Bits (if applicable)
Encoder	n	$\log n$	
Decoder	$\log n$	n	
Multiplexer	n	1	$\log n$
Demultiplexer	1	n	$\log n$

W7 LG 3-26-18 Flip Flops, Counters, Sequential Circuits

Today

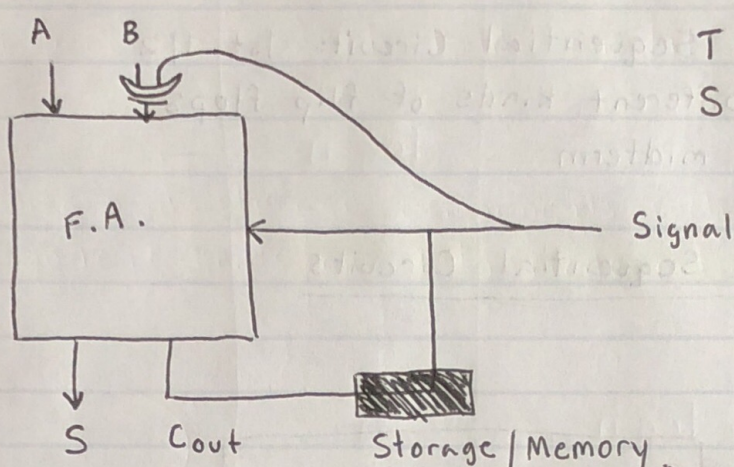
- Ch 4 Sequential Circuits 1st 1/2
  - Different kinds of flip flops
- Hints midterm

Ch 4 Sequential Circuits



- a sequential circuit's current output depends on a previous output
- has storage / memory
- has concept of sequence
- synchronous clock

Example Full Adder as Sequential Circuit



T = O(N)
S = O(1)

One bit register / flip flop

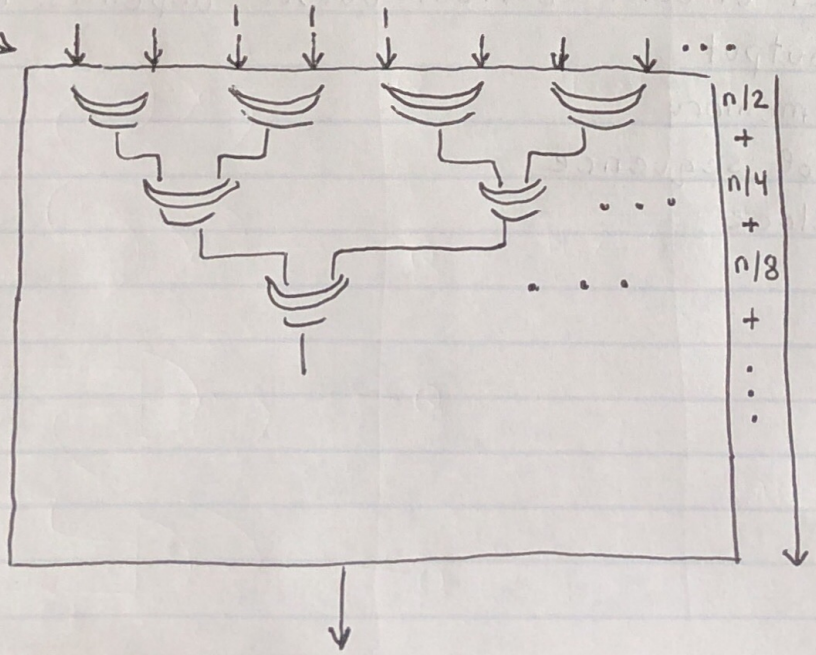
Example Parity Bit - Combinational

1 1 1 1 0 1 1 1 0 1

XOR all the bits

Even Parity
0 } parity bit
1
Odd

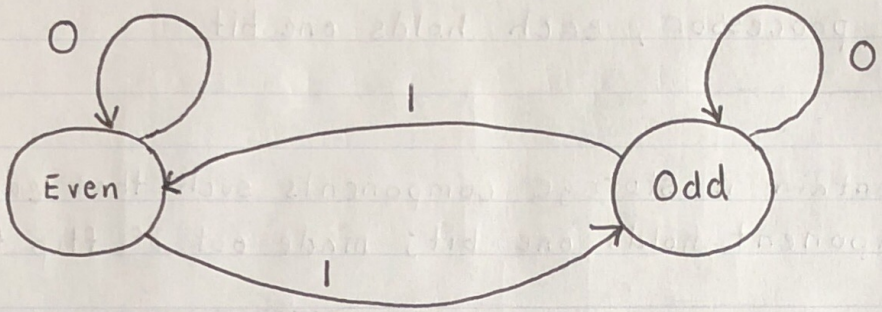
Parity bit is calculated w/ XOR

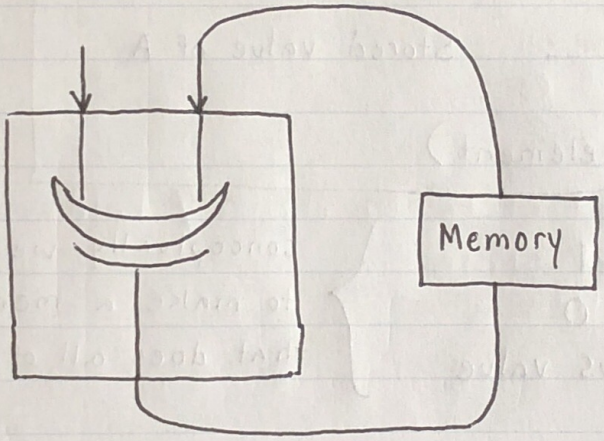


log n levels

O(n) XOR gates
O(log n) time

Example **Parity Bit - Sequential**



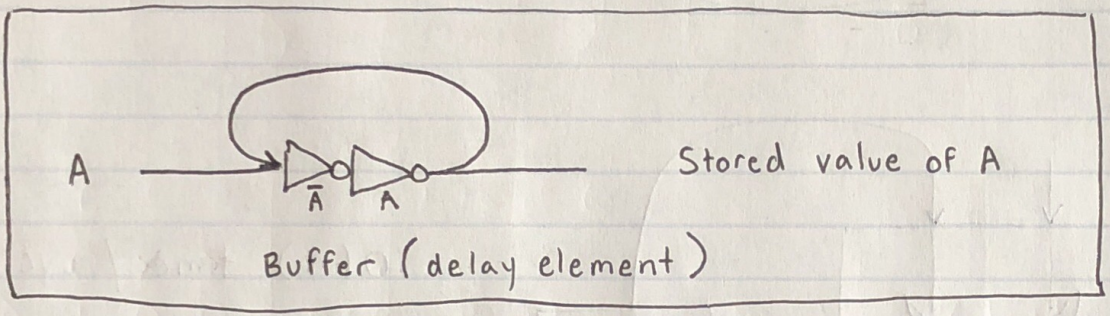


O(n) time / cycles

Flip Flops

- memory in processor, each holds one bit

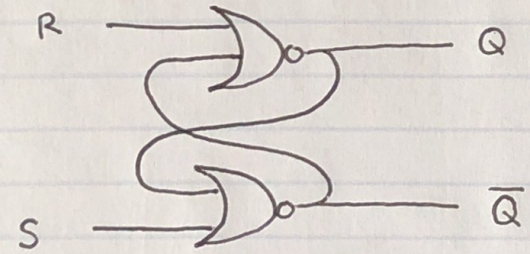
registers : contain n storage components such that each component holds one bit; made out of flip flops



- Set → write 1
- Reset → write 0
- Hold → previous value

} conceptually we want to make a memory unit that does all of this

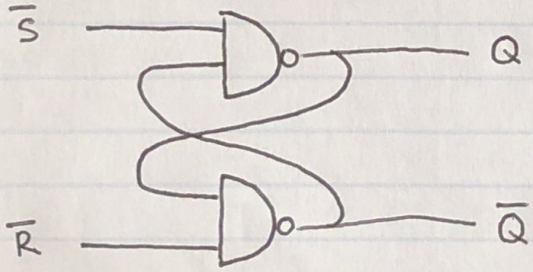
SR Latch



Next State

$\overline{\text{Next State}}$

NOR Implementation



NAND Implementation

SR Latch - Excitation Table

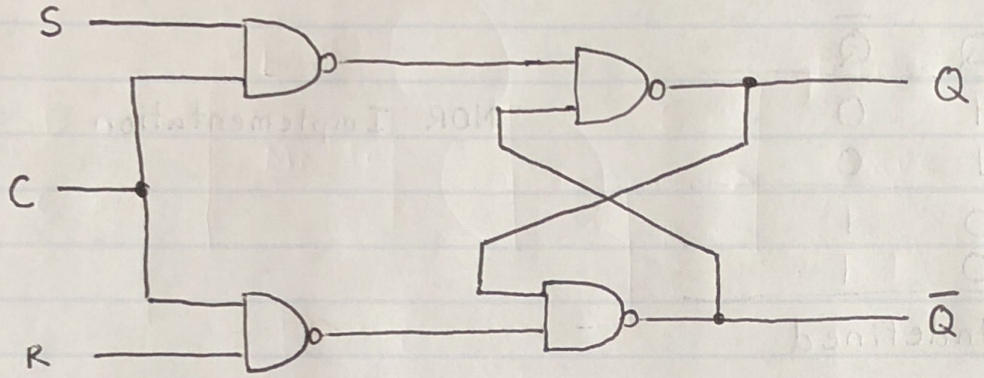
	S	R	Q	$\overline{Q}$
Sets	1	0	1	0
Holds	0	0	1	0
Resets	0	1	0	1
Holds	0	0	0	1
	1	1	Undefined	

NOR Implementation

	$\overline{S}$	$\overline{R}$	Q	$\overline{Q}$
Sets	0	1	1	0
Holds	1	1	1	0
Resets	1	0	0	1
Holds	1	1	0	1
	0	0	Undefined	

NAND Implementation

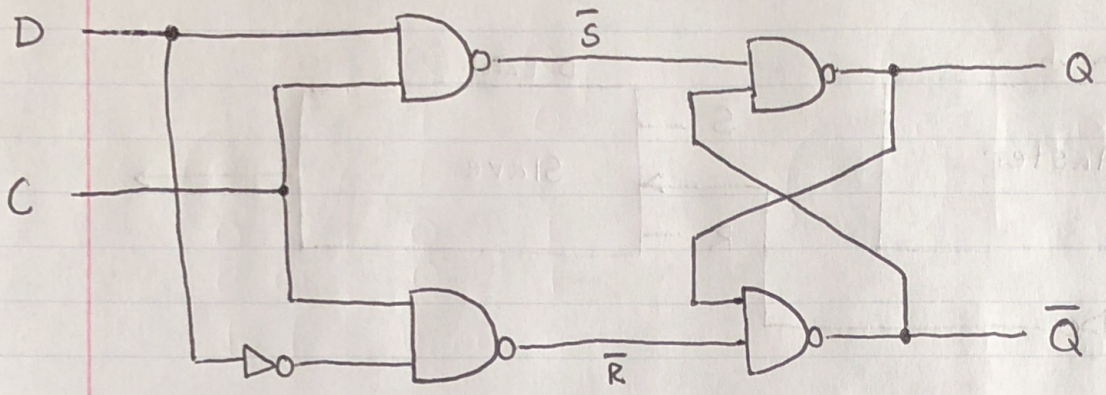
SR Latch with Control Input



C	S	R	Next state of Q
0	X	X	No change / holds
1	0	0	~~No change / holds~~ No change / holds
1	0	1	Q = 0; Reset state
1	1	0	Q = 1; Set state
1	1	1	Undefined

activated

D Latch

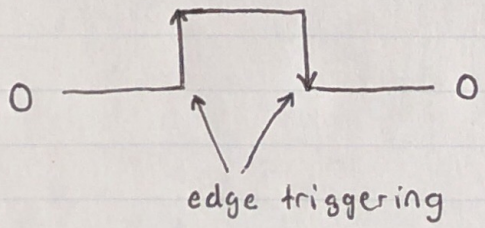


C	D	Next state of Q
0	X	No change / holds
1	0	Q = 0; Reset state
1	1	Q = 1; Set state

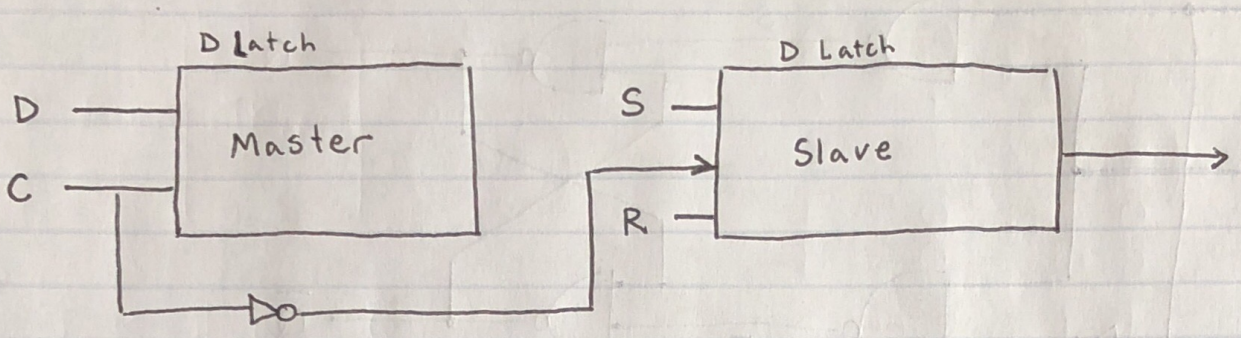
Edge Triggering

When is it 1?

- When it goes up — edge positive
- When it goes down — edge negative



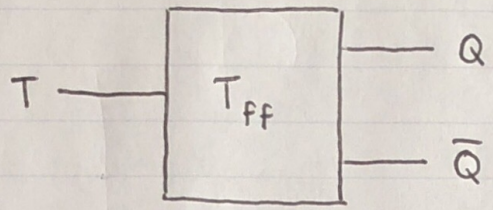
D Flip Flop



- By using one D Latch activated when control is 1 and one D Latch activated when control is 0, we only get an output when control has been both 1 and 0 (when an edge triggering has occurred)

D	Q	$\overline{Q}$
0	0	1
1	1	0

T Flip Flop

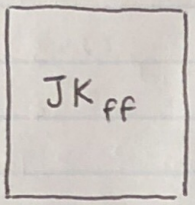


T	Q	$\bar{Q}$
0	Q	$\bar{Q}$
1	$\bar{Q}$	Q

→ when T = 0 acts like D_{ff} / Holds
→ toggles bit

- Can implement with XOR to toggle it

JK Flip Flop



J	K	Q	$\bar{Q}$	
0	0	Q	$\bar{Q}$	Holds "D"
0	1	0	1	(R)esets
1	0	1	0	(S)ets
1	1	$\bar{Q}$	Q	(T)oggle

In Class Exercise **Conceptually describe what a flip flop is.**

A flip flop is a component used as memory in the processor to store a single bit. There are different kinds of flip flops: D_{ff}, T_{ff}, JK_{ff}. Some functions of a flip flop include setting, resetting, holding, and toggling the bit. Flip flops are used to make registers, fast access memory storage internal to the processor. It is implemented as a sequential circuit.

W9 L7 4-16-18 Microprocessor, PLA, ROM, Register Transfer, HDL

- Overview
  - Ch 1 - Intro
  - Ch 2 > Combinational
  - Ch 3 >
  - Ch 4 - Sequential
- Ch 5 - PLD
- Ch 6 - Registers
  - ↓ Rest of the course
- Final Exam

- microprocessor - when you take ALU, CU, registers and integrate it into a single chip

- Today's processors are VLSI

- Approx. 10 gates in one flip flop

- Why use PLD's —— flexible
  - reusable

- Application Specific Integrated Circuit (ASIC) - opposite of PLD

- All PLD's we'll discuss are used in the ALU

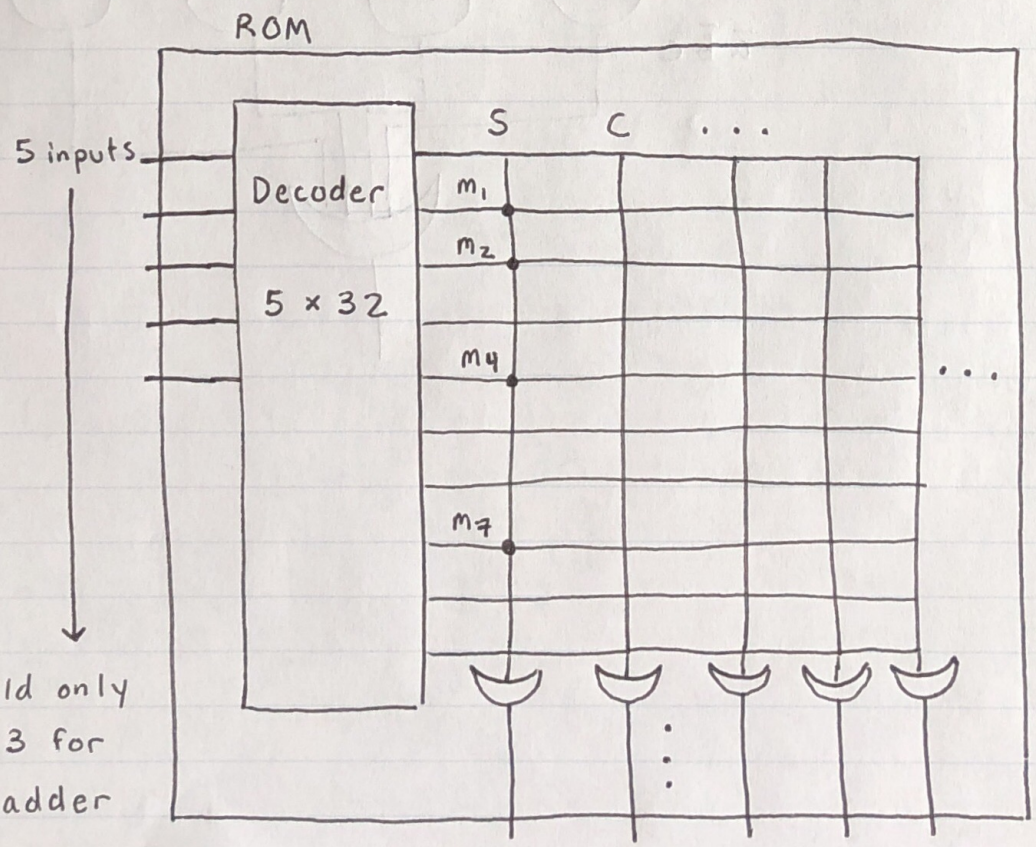
Programmable Logic Devices (PLDs)

Levels of Integration

SSI	Small Scale Integration	few - 10's
MSI	Medium Scale Integration	10's - 100's
LSI	Large Scale Integration	100's - 1000's
VLSI	Very Large Scale Integration	10,000's - 100,000's
ULSI	Ultra Scale Integration	
WSI	Wafer Scale Integration	

PLD # 1 Read Only Memory (ROM)

F. A.	S C
0 0 0	0 0
0 0 1	1 0
0 1 0	1 0
0 1 1	0 1
1 0 0	1 0
1 0 1	0 1
1 1 0	0 1
1 1 1	1 1

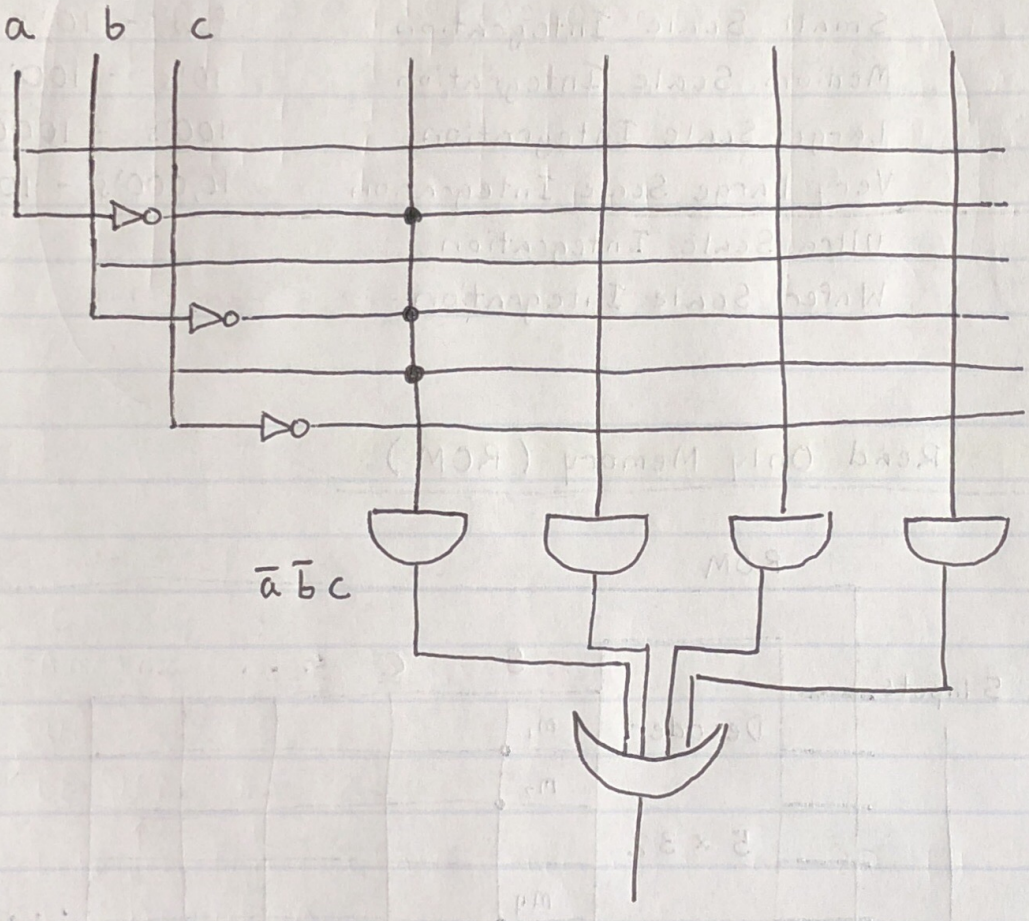


would only use 3 for full adder

$S = m_1 + m_2 + m_4 + m_7$

PLD #2 Programmable Logic Arrays (PLAs)

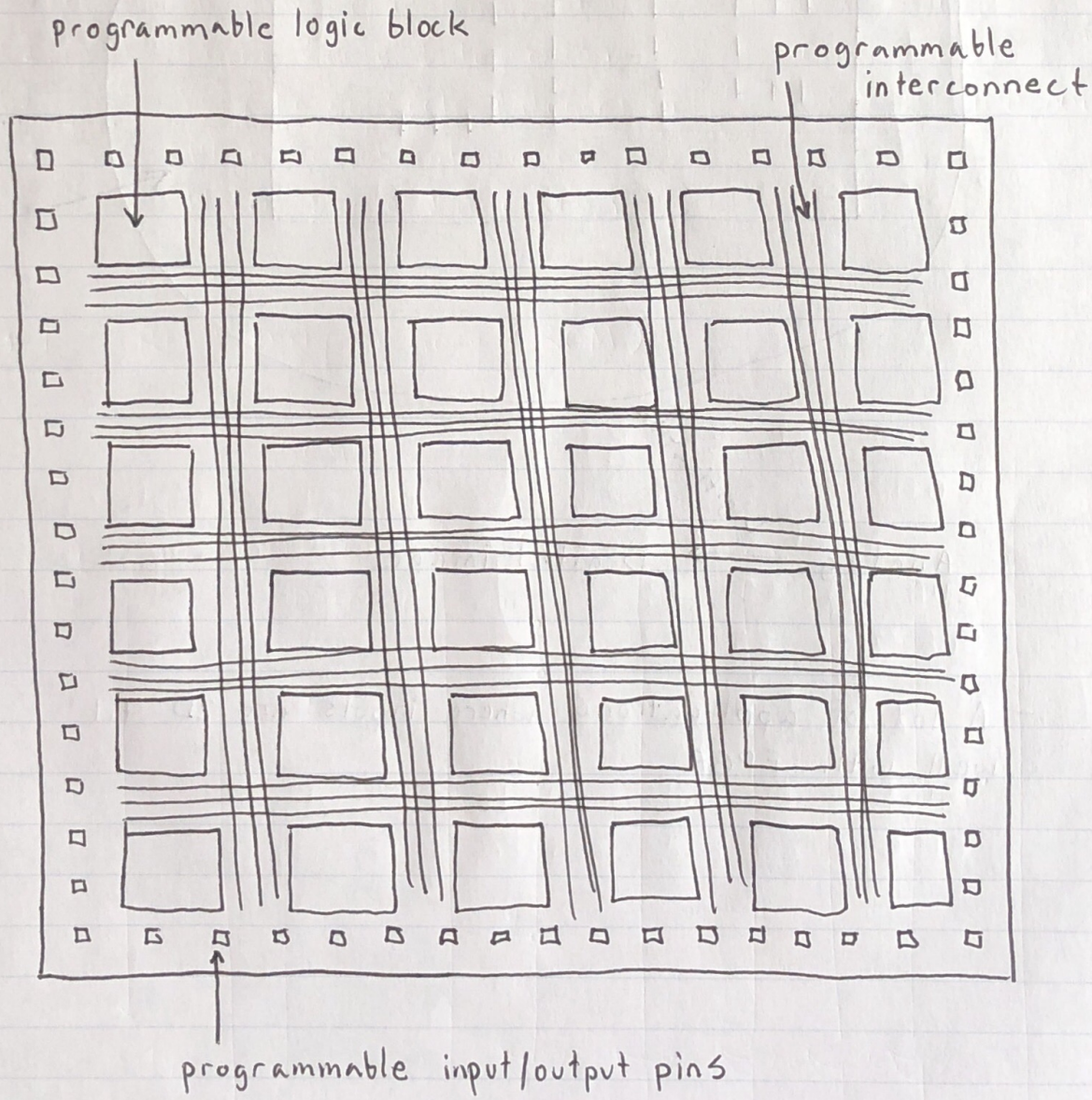
- like a "bar"
- like sum of products



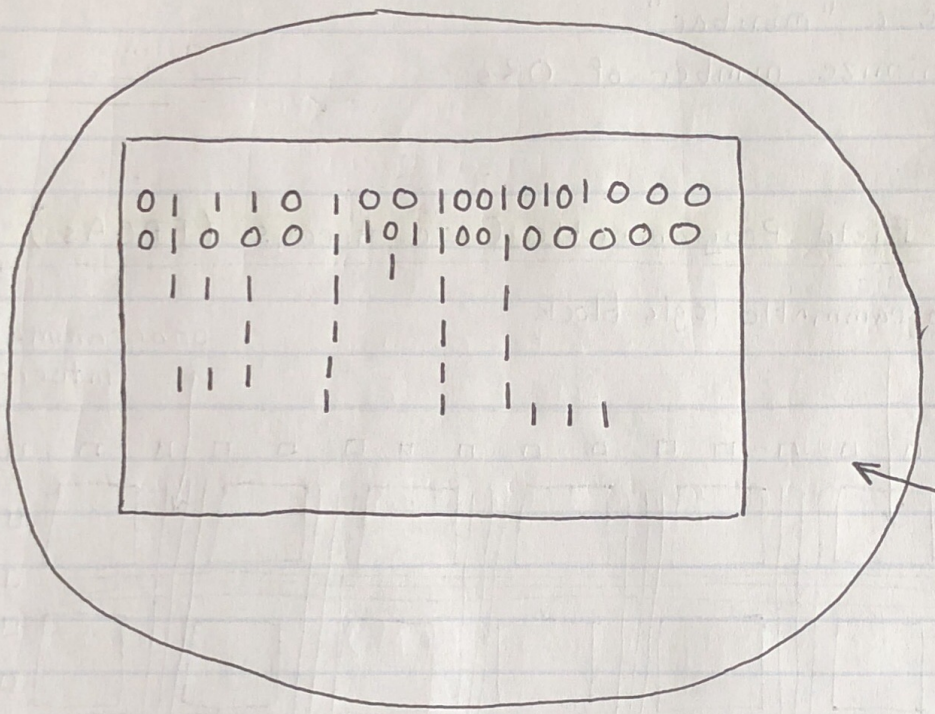
PLD #3 Programmable Array Logic Devices (PALs)

- like a "minibar"
- minimize number of ORs

PLD #4 Field Programmable Gate Arrays (FPGAs)



Example Image Processing (not on exam)



Digitization

Analog → Digital image

- A lot of applications where inputs are 2D fit well with FPGA

Registers and Register Transfer

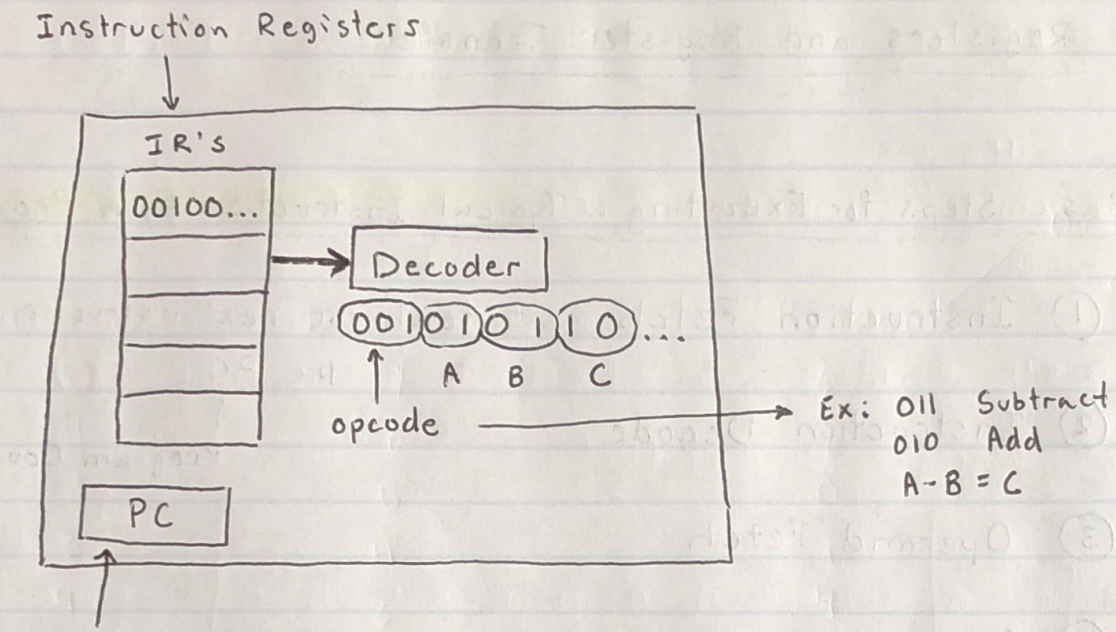
Basic Steps for Executing a Typical Instruction by a Processor

1. Instruction Fetch - fetch the next instruction pointed to by the PC
   PC = Program Counter
2. Instruction Decode
3. Operand Fetch
4. Execute
5. Write back (WB) / Store - put it back in register/memory

Access Time

- Registers - ns (10^{-9})
- Secondary Memory - ms (10^{-3})
- I/O - s (10^{1})

Example Internal Registers of Hypothetical 16 bit Processor



Program counter - address of next instruction

Assembly

SUB R1, R2, R3

011 ( ) ( ) ( )

Reg1 Reg2 Reg3

SUB R1, #5, R3

SUB 2086, #5, R3

Address — The number 5

} Format of instruction depends on processor

RISC - operands must be in registers

CISC - operands can be in memory

Register Transfer Operations

Subtract

[redacted] A, B, C

$A - B = A + \overline{B} + 1$

Say we want to do this operation

↓

```
LOAD  A, R1
LOAD  B, R2
COMP  R2          (complement)
ADD   R2, 1, R2
ADD   R1, R2, R3
```

This would be the operation at the register level

- Registers are made out of flip flops
  - An n bit register is made out of n flip flops

Operations with Registers

Counters

On Exams "Crazy Counter"

Upcounter	Downcounter	Random
0	7	6
1	6	2
2	5	0
3	4	1
4	3	4
5	2	
6	1	
7	0	

- Counters are used for:
  - multiplication → repeated addition → count down
  - division → repeated subtraction → count up
  - many things in the processor

Project Preview Design a computer with single ALU, counter, where the numbers can be signed.

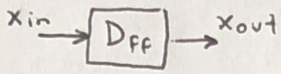
When it's signed, do you always:
- add for multiplication
- subtract for division
- count down for multiplication
- count up for division

?

↓

Hint: No

D Flip Flops



$x_{in} = x_{out}$

∴ Next State and D flip flops are equal.

★ On Exam

Example Design a Counter that counts up from 0 to 7 with both D flip flops and T flip flops.

	current state a b c	next state	D_2 D_1 D_0	T_2 T_1 T_0
0	0 0 0	0 0 1	0 0 1	0 0 1
1	0 0 1	0 1 0	0 1 0	0 1 1
2	0 1 0	0 1 1	0 1 1	0 0 1
3	0 1 1	1 0 0	1 0 0	1 1 1
4	1 0 0	1 0 1	1 0 1	0 0 1
5	1 0 1	1 1 0	1 1 0	0 1 1
6	1 1 0	1 1 1	1 1 1	0 0 1
7	1 1 1	0 0 0	0 0 0	1 1 1

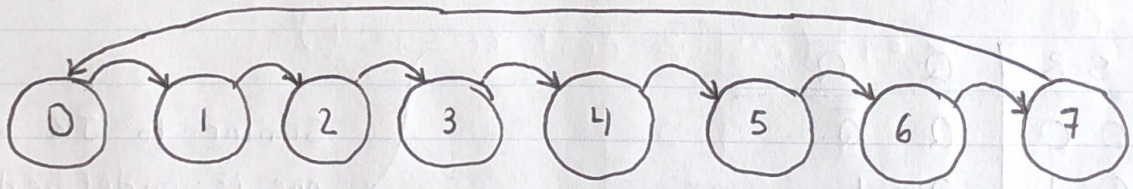
= (current state ↔ next state)

Compare the bits of Current State and Next State. If a bit changes, the flip flop in the same position is a 1.

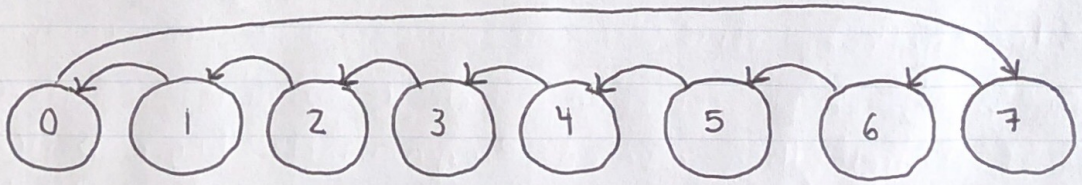
3 bit register

$T_2 = \Sigma\Pi$ $T_1 = \Sigma\Pi$ $T_0 = 1$

State Diagram of Upcounter (0 - 7)



State Diagram of Downcounter (7 - 0)



Recap Flip Flop Excitation Tables

J	K	Q	$\bar{Q}$	
0	0	Q	$\bar{Q}$	
0	1	0	1	
1	0	1	0	
1	1	$\bar{Q}$	Q	Toggle

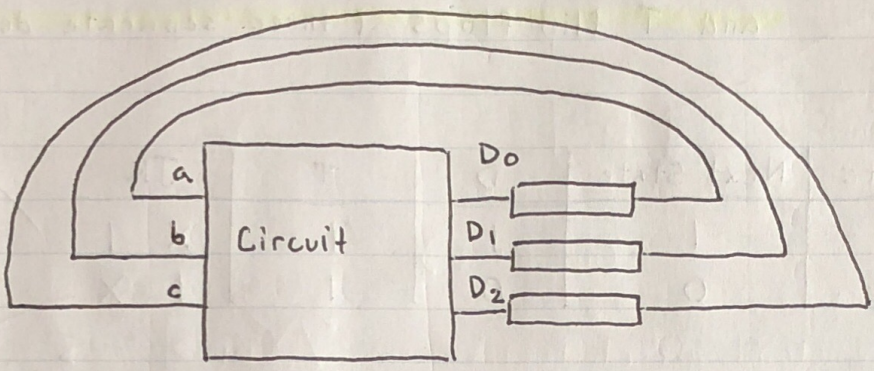
T	Q	$\bar{Q}$
0	Q	$\bar{Q}$
1	$\bar{Q}$	Q

S	R	Q	$\bar{Q}$
0	0	Q	$\bar{Q}$
0	1	0	1
1	0	1	0
1	1	Undefined	

} Similar to JK except last one is undefined

* On Exam

Example Design a counter cont'd



To design a counter at the logic gate level find the sum of products using the current state.

For the D flip flop solution:

$$D_0 = \underset{m_0}{\bar{a}\bar{b}\bar{c}} + \underset{m_2}{\bar{a}b\bar{c}} + \underset{m_4}{a\bar{b}\bar{c}} + \underset{m_6}{ab\bar{c}}$$

$$D_1 = m_1 + m_2 + m_5 + m_6$$
$$= \bar{a}\bar{b}c + \bar{a}b\bar{c} + a\bar{b}c + ab\bar{c}$$

$$D_2 = m_3 + m_4 + m_5 + m_6$$
$$= \bar{a}bc + a\bar{b}\bar{c} + a\bar{b}c + ab\bar{c}$$

- If you use D flip flops you'll end up using a lot of gates
- If you use JK or T flip flops you will use less gates
  - Why? Because JK and T are more complex while D is simple
  - If using JK flipflops you'd have two inputs to the flip flop and would have to solve for $\Sigma\Pi$ of both J and K

★ On Exam (2 bit or 3 bit)

In Class Exercise Design a small counter that counts from 0 to 1 with D flip flops, JK flip flops, and T flip flops (three separate designs)

a Current State	Next State	D	T	JK
0	1	1	1	X 1
1	0	0	1	1 X

Either:
0 1
1 1 (Toggle)

Either:
1 0
1 1 (Toggle)

$D = \bar{a}$

$T = 1$

$J = 1$ (assume don't care is 1)

$K = 1$ (assume don't care is 1)

Today we connected the first part of the course to the second half by revisiting the diagram of the components of the computer. We reviewed which components we covered in the first half then covered Programmable Logic Devices such as ROM, PLA, FPGA that are used in the ALU. Next we covered registers and register transfers (Ch 6) which built on Ch 4 in which we learned about sequential circuits and flip flops, which registers are made of. We learned about upcounters and downcounters and how to design a counter made out of D, T, and JK flip flops and combinations of the three. Overall we also transitioned from low level to high level.

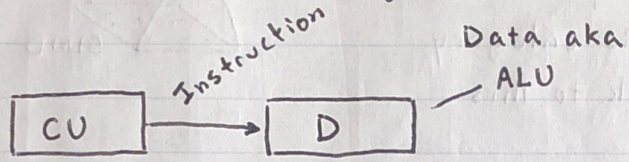
WIO LB 4-23-18 Processor & Control Design, Computer Classifications

Today
- Computer Classifications (not in book)
- Processor Design (Ch 7)

Flynn's Computer Classifications

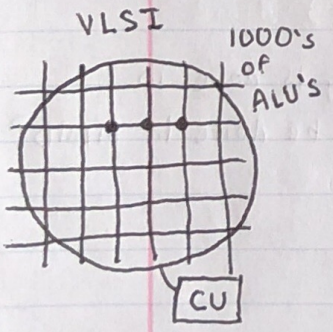
- 4 Groups of Types of Processor Design
- Computer Architecture Classifications based on processors

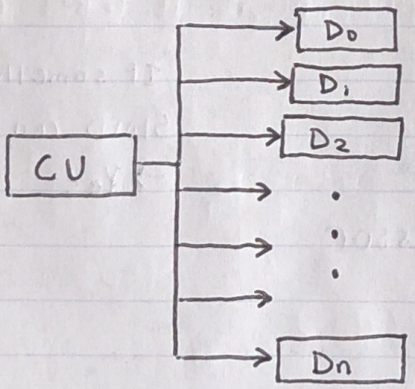
1. Single Instruction Single Data (SISD)



- "uni-processors"
- one ALU
- one instruction at a time
- what we have seen so far

2. Single Instruction Multiple Data (SIMD)

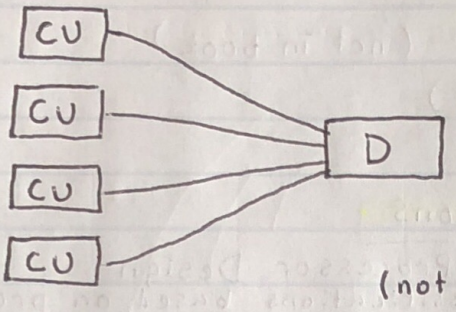




"imagine class is a processor, prof is control unit, each desk is ALU, raise your right hand, raise your right hand if you have glasses"

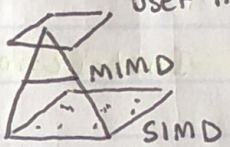
- tightly coupled multiprocessor
- All ALU's do the same thing at the same time, except if they are masked out
  - they hear the same instruction
- synchronized
- synchronous
- very much in use now esp. in GPU's

3. Multiple Instruction Single Data (MISD)



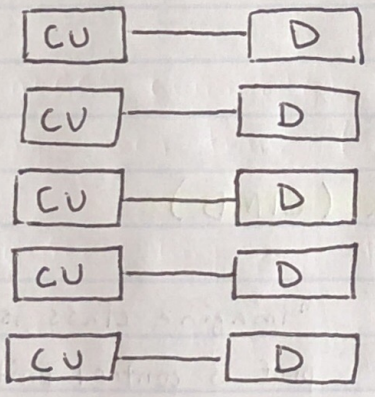
(not MISD)

- image understanding architecture
- ? what falls in this category
  - ↳ will come back to this



→ possible to have multiple types in the same computer

4. Multiple Instruction Multiple Data (MIMD)

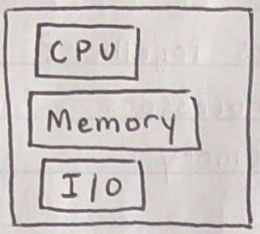


If something is done in SIMD can it be done in MIMD?
→ Yes

- loosely coupled multiprocessor
- asynchronous
- examples:
  - cloud (distributed)
  - IP address
  - broadcast
  - supercomputer (within a single computer)
  - clusters
  - IOT

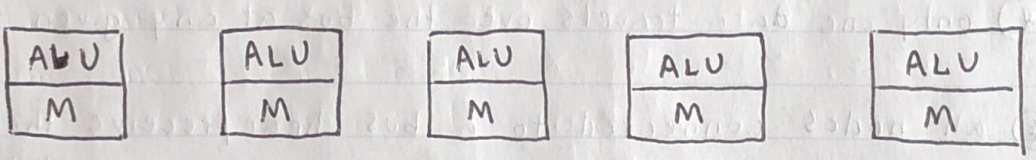
Computer Classifications Based on Memory

1.

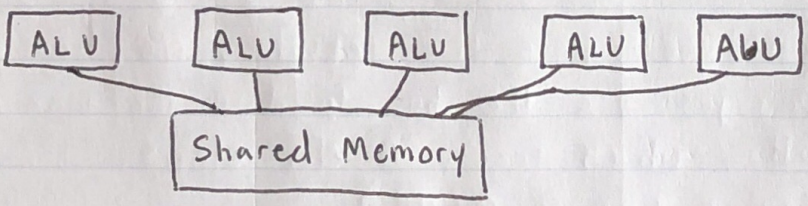


2. What happens when you have multiple data processors (ALU's)?

A) Distributed Memory — works for SIMD, MIMD



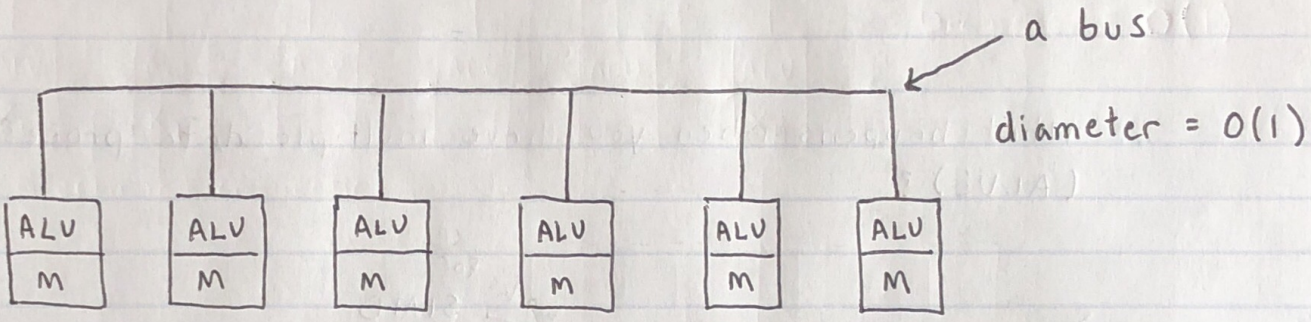
B) Shared Memory



Buses

How do we interconnect multiple modules together?

- processors
- memory



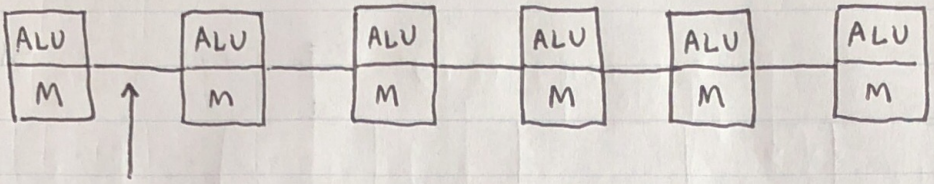
(*) only one data travels over the bus at any given time

(*) all nodes connected to a bus hear/receive the same data at any given time (the data is "broadcast")

(*) to go from any point to another point it takes O(1) time

Compare buses with:

distributed memory

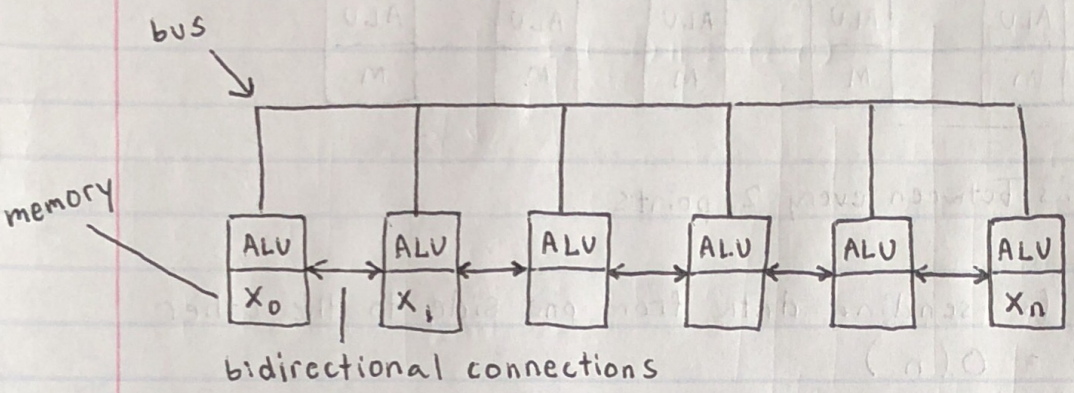


connections between every 2 points

- $T = O(n)$ for sending data from one side to the other
- Diameter = $O(n)$
- Like RCA, stupid structure

- Diameter: shortest distance between two points farthest apart

In Class Exercise Find max of all x_i's with a bus. (without a bus would be $O(n)$ time)



SIMD

Diameter = $O(1)$

- If we divide n by x $(\frac{n}{x})$ we get x # of groups.
- If we want to make the size of each group equal to the # of groups, solve for x where $x^2 = n$:

$$x^2 = n$$
$$x = \sqrt{n}$$

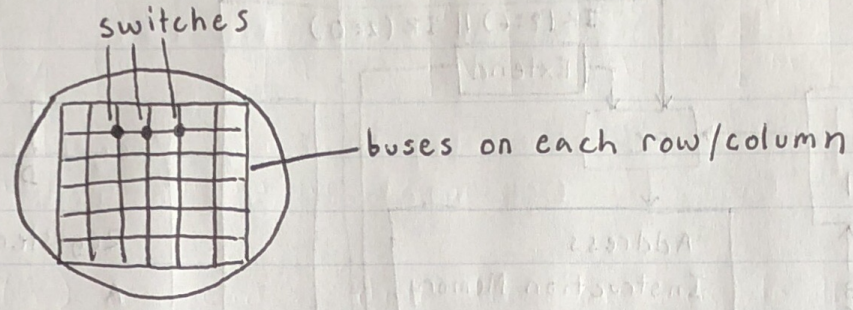
- Take groups of size $\sqrt{n}$
- For all $\sqrt{n}$ groups do the following:
  - Find the max using neighboring connections until you have the result for each group (in parallel)
  - At this point you'll have $\sqrt{n}$ results, one from each group
  - Now you can just use the bus to find the max of the the $\sqrt{n}$ results, which takes $\sqrt{n}$ steps

  ∴ the running time is $O(\sqrt{n})$

- By adding a simple bus we get a huge speedup (one wire)

Reconfigurable Arrays

- 2D
- GPU's

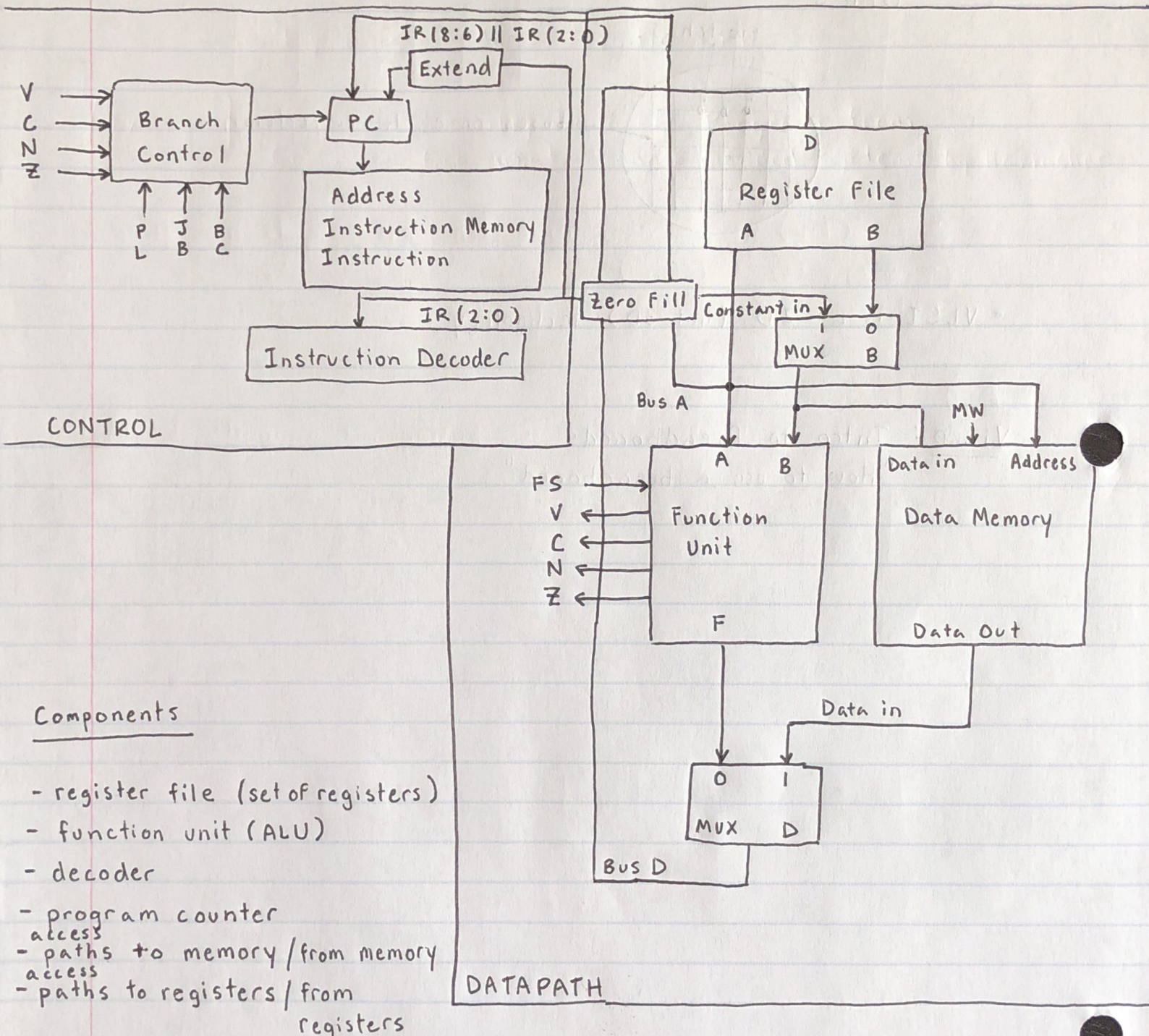


- VLSI is a 2D (not 3D) technology

Video: Intro to Breadboards
"How to use a breadboard"

Group In Class Exercise

Design a CPU at the block diagram level



Components

- register file (set of registers)
- function unit (ALU)
- decoder
- program counter
- paths to memory / from memory access
- paths to registers / from registers

Components needed for 5 Steps for Executing a Basic Instruction:

① Instruction Fetch
- program counter register

② Instruction Decode
- decoder → decodes a word

③ Operand Fetch
- bus to memory / registers

④ Execute
- ALU aka functional unit
  ↳ adder / subtractor
  ↳ result goes to register

⑤ Write Back
- bus to memory / registers

Textbook References
- Figure 8-1
- Figure 8-15
- Figure 8-10
- Figure 8-11

- Control unit - decoder and signals it generates from the instruction

W11 L9 4-30-18 Instruction Set Architecture, Operation Cycle, Addressing

Instruction Set Architecture

- assembly language for a specific CPU

Example Hypothetical Assembly Language Program from "Computer Organization and Embedded Systems" by Hamacher

	Memory address label	Operation	Addressing or data information
Assembler directive		ORIGIN	100
Statements that generate machine instructions		LD	R2, N
		CLR	R3
		MOV	R4, #NUM1
	LOOP:	LD	R5, (R4)
		ADD	R3, R3, R5
		ADD	R4, R4, #4
		SUB	R2, R2, #1
		BGT	R2, R0, LOOP
		ST	R3, SUM
		next instruction	
Assembler directives		ORIGIN	200
	SUM:	RESERVE	4
	N:	DATAWORD	150
	NUM1:	RESERVE	600
		END	

Figure 2.13 Assembly language representation for the program in Figure 2.12.

32 bit

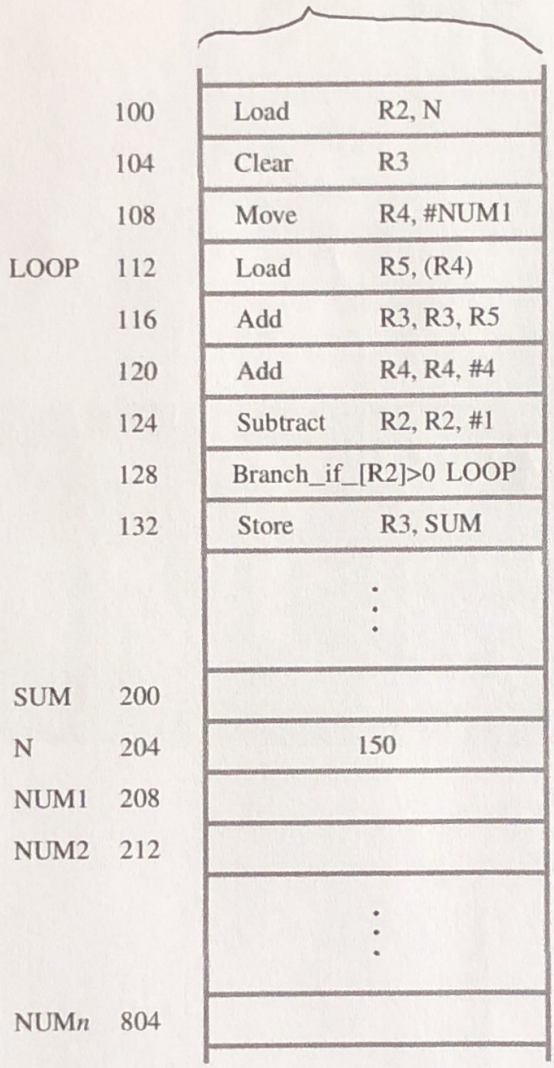


Figure 2.12 Memory arrangement for the program in Figure 2.8.

Trace the program

32 bit

R0 [0]

R1 [ ]

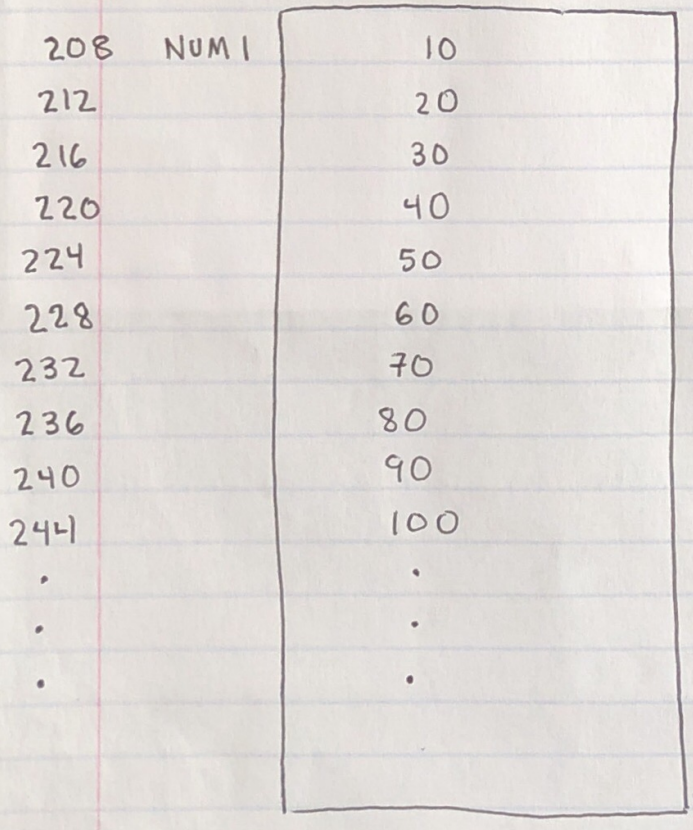
R2 [150 - 1 = 149] - 1 = 148

R3 [0 + 10 = 10] + 20 = 30

R4 [208 + 4 = 212] + 4 = 216

R5 [~~10~~ 20]

Assume Memory Contents



The program is adding these numbers with a counter of 150, stops when the counter reaches 0

- # means put immediate

- ( ) means indirect

★ On Exam

In Class Exercise

Trace previous example with N: DATAWORD 10 and the memory contents below:

- Show contents of all registers

		Memory
208	NUM1	1
212		2
216		3
220		4
224		5
228		6
232		7
236		8
240		9
244		10

Assume R0 contains 0

Iteration 1

	R0	R1	R2	R3	R4	R5
LD R2, N	ϕ		10			
CLR R3	ϕ		10	ϕ		
MOV R4, #NUM1	ϕ		10	ϕ	208	
LD R5, (R4)	ϕ		10	ϕ	208	1
ADD R3, R3, R5	ϕ		10	1	208	1
ADD R4, R4, #4	ϕ		10	1	212	1
SUB R2, R2, #1	ϕ		9	1	212	1
BGT R2, R0, LOOP		9 > ϕ → LOOP				

Iteration 2

	R0	R1	R2	R3	R4	R5
LD R5, (R4)	ϕ		9	1	212	2
ADD R3, R3, R5	ϕ		9	3	212	2
ADD R4, R4, #4	ϕ		9	3	216	2
SUB R2, R2, #1	ϕ		8	3	216	2
BGT R2, R0, LOOP		8 > ϕ → LOOP				

Iteration 3

R0	R1	R2	R3	R4	R5
ϕ		8	3	216	3
ϕ		8	6	216	3
ϕ		8	6	220	3
ϕ		7	6	220	3
	7 > ϕ → LOOP				

Iteration 4

R0	R1	R2	R3	R4	R5
ϕ		7	6	220	4
ϕ		7	10	220	4
ϕ		7	10	224	4
ϕ		6	10	224	4

$6 > \phi \rightarrow$ LOOP

Iteration 5

R0	R1	R2	R3	R4	R5
ϕ		6	10	224	5
ϕ		6	15	224	5
ϕ		6	15	228	5
ϕ		5	15	228	5

$5 > \phi \rightarrow$ LOOP

Iteration 6

R0	R1	R2	R3	R4	R5
ϕ		5	15	228	6
ϕ		5	21	228	6
ϕ		5	21	232	6
ϕ		4	21	232	6

$4 > \phi \rightarrow$ LOOP

Iteration 7

R0	R1	R2	R3	R4	R5
ϕ		4	21	232	7
ϕ		4	28	232	7
ϕ		4	28	236	7
ϕ		3	28	236	7

$3 > \phi \rightarrow$ LOOP

Iteration 8

R0	R1	R2	R3	R4	R5
ϕ		3	28	236	8
ϕ		3	36	236	8
ϕ		3	36	240	8
ϕ		2	36	240	8

$2 > \phi$ → LOOP

Iteration 9

R0	R1	R2	R3	R4	R5
ϕ		2	36	240	9
ϕ		2	45	240	9
ϕ		2	45	244	9
ϕ		1	45	244	9

$1 > \phi$ → LOOP

Iteration 10

R0	R1	R2	R3	R4	R5
ϕ		1	45	244	10
ϕ		1	55	244	10
ϕ		1	55	248	10
ϕ		ϕ	55	248	10

ϕ is not greater than ϕ → STOP

Section 9.2 Operand Addressing

ISA Classifications by Operand Addressing

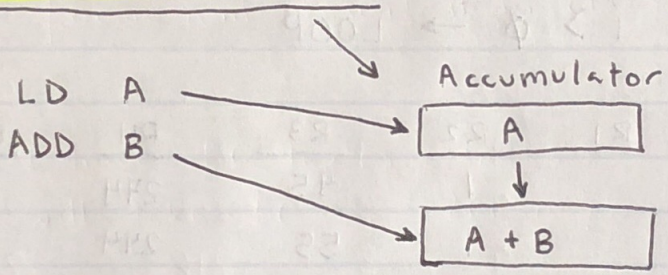
3 - Address Instructions

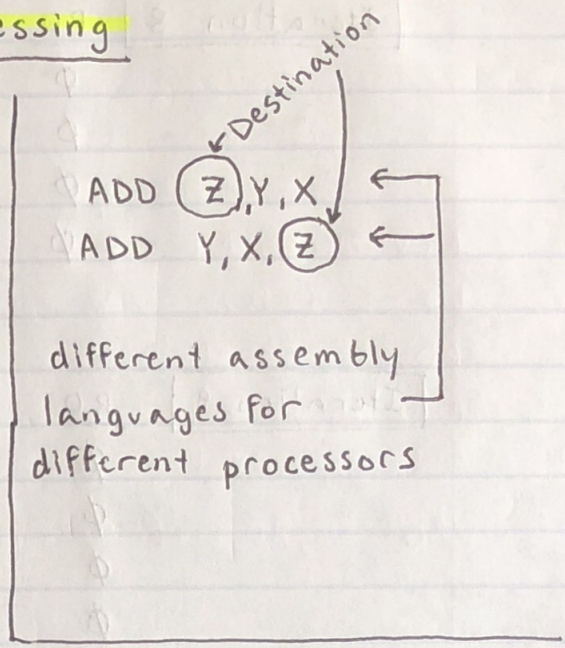
```
ADD  R1, A, B
```

2 - Address Instructions

```
MOV  R1, A
ADD  R1, B
```

1 - Address Instructions





Addressing Modes

- # immediate addressing
- () indirect addressing - like a pointer
- no symbol direct addressing

Types of Instructions

- Logic - AND, OR
- Arithmetic - ADD, SUB
- Branch
- Shift
- Memory access

- Shift left 1 bit = multiply by two

- Shift right 1 bit = divide by two

W12 L10 5-7-18 Pipelining vs. Multiprocessing, RISC vs. CISC

Example — Multiprocessing

[redacted] Analogy - 50 research projects 4 steps × 1 day

Research Project

Day 1 — Go to library / find related material

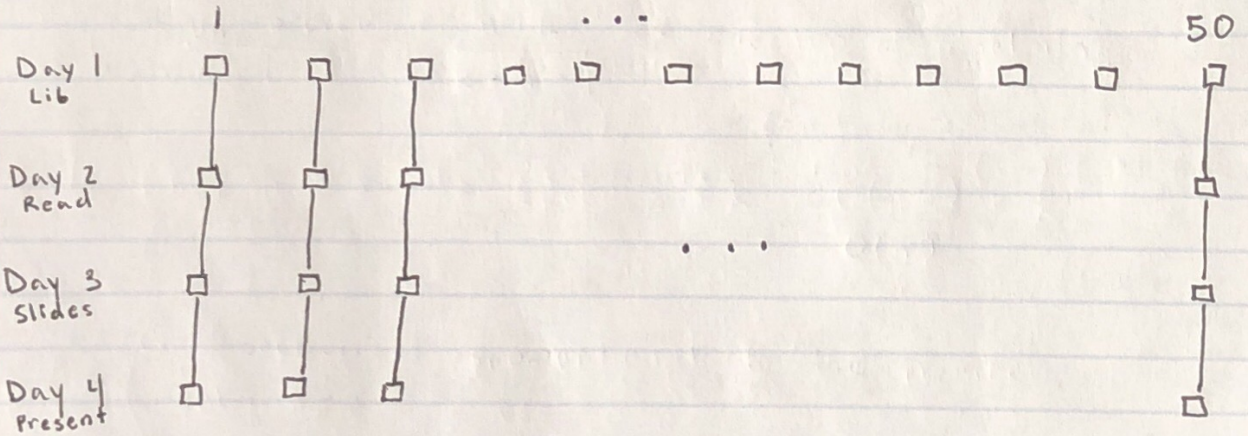
Day 2 — Read and summarize it ← Brain

Day 3 — Make a powerpoint presentation / slides

Day 4 — Come and present it in class

- If I have 50 topics / tasks
- It will take 50 × 4 = 200 days

Multiprocessing:



$$\text{Speedup} = \frac{\text{unimproved}}{\text{improved}} = \frac{50 \times 4}{4} = 50$$

I used 50 people and got speedup of 50 → Linear speedup

Drawbacks of multiprocessing

- we need 50 people
- we need 50 people who can do everything

Pipelining

- overlap tasks
- break down tasks into subtasks

Example — Pipelining Analogy (same research project analogy)

- 12 tasks → w/o pipelining takes 48 days
- 4 people
- each person in charge of one day subtask

- pipeline flushing time - 4 days
- like a factory
- after 4th person presents task 1, he can present task 2 the next day
- every task is called a stage
- every stage can be used at the same time
- throughput: rate of output
  - no pipeline - 1 output per 4 cycles / days
  - pipeline - 1 output per 1 cycle / day
    - ↑ after initial delay

- don't have to have 4 people that do everything
- could apply this to steps 1-5 of instruction execution
- data lines up to be processed

key feature of pipelining →
- each stage must have the same amount of delay as the others to avoid bottlenecks
- slowest part of a pipeline becomes the clock rate for the pipeline
- 4 different parts good for different purposes (ALU, etc.)
- for every stage you need registers / buffers

super scalar processing : pipelining and multiprocessing

Speedup for doing pipelining as compared to sequential

- for n tasks where each task takes k units of time

$$\frac{\text{unimproved}}{\text{improved}} = \frac{n \cdot k}{k + (n-1)} = \text{speedup of pipelining}$$

↑ first task takes T = k ; remaining (n-1) tasks take T = 1

when $n \to \infty$ → Speedup = k $\quad \frac{O(nk)}{O(n)} = k$

∴ Speedup is proportional to the number of stages you use

RISC

- processor design in which all instructions are relatively simple and take the same amount of time
- uses pipelining
- smaller number of instructions
- use multiple instructions to do one complex operation
- don't really need scheduling when pipelining, only when introduce multiprocessing

Example Hypothetical CISC vs. RISC

	CISC	RISC	
$M = x^3$	CUBE X	MUL X, X, Y MUL Y, X, M	20ns ↑ Each instruction takes the same time in RISC
10ns 50ns	Each instruction takes different time in CISC		

CISC

- time per instruction varies
- simpler for programming
- optimizes per instruction
- scheduling is not easy w/ CISC

Exam Question — Is RISC or CISC better and why?

CISC → mainly b/c optimization but scheduling becomes a problem
→ simpler for programming

RISC → smaller size of RISC processors is one reason why ARM processors (RISC) are used in mobile phones

Where does pipelining fall in Flynn's Computer Classifications

SISD
~~SIMD~~
MISD → if you think of single data (processor) split up into multiple stages, each stage is working on a different instruction
~~MIMD~~

In Class Exercise Describe in your own words what is pipelining.

Pipelining is done by overlapping tasks and breaking down a task into subtasks. Each subtask is called a step or stage and each stage must have the same amount of delay to avoid bottlenecks. The slowest stage of the pipeline becomes the clock rate for the pipeline. A good analogy is a production line where data lines up to be processed, each step is working at the same time, but each step does not have/need to be capable of performing the tasks for all other steps.

In Class Exercise Give an example of pipelining.

Given six instructions $i_6\ i_5\ i_4\ i_3\ i_2\ i_1$, the steps of instruction execution can be pipelined as follows:

T = 1	IF_{i_1}				
T = 2	IF_{i_2}	D_{i_1}			
T = 3	IF_{i_3}	D_{i_2}	OF_{i_1}		
T = 4	IF_{i_4}	D_{i_3}	OF_{i_2}	E_{i_1}	
T = 5	IF_{i_5}	D_{i_4}	OF_{i_3}	E_{i_2}	S_{i_1}

Start : Not on exam

Design a multiprocessor.

Example Given a program, determine how many ALU's to use.
(ASICs - Application Specific ICs)

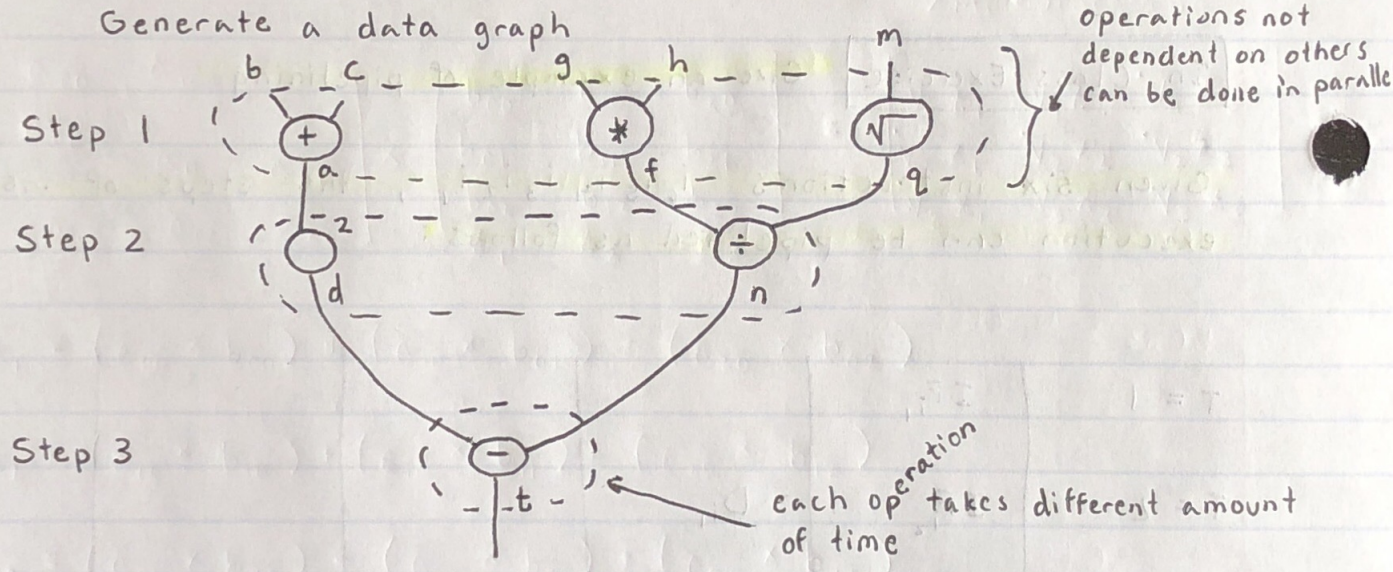
i_1 $a = b + c$
i_2 $d = (a^2)$
i_3 $f = g * h$
$q = \sqrt{m}$
$n = f \div q$
$t = n - d$

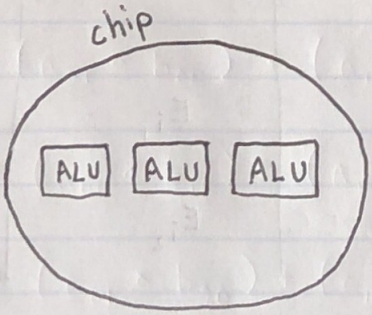
dependent on i_1 ↓

out of order execution : $i_1 \rightarrow i_3$ ↓

also used for pipelining

Generate a data graph



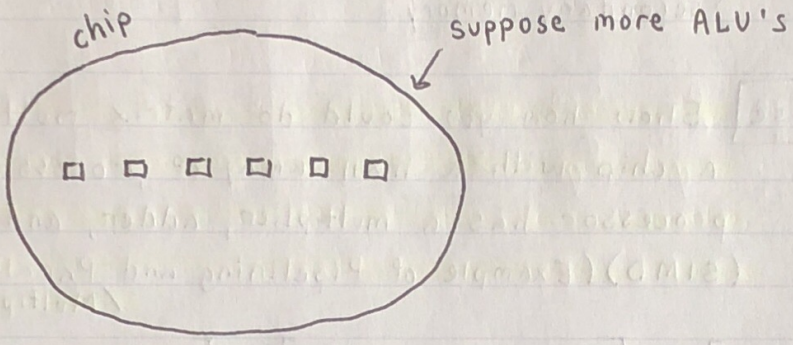


synthesize problem → Fastest possible → 3 ALU's

↑ b/c at the top of data graph you have 3 independent calculations

Example — Scheduling Problem

Given a task described by a data dependency graph show how to map and schedule it for execution on available processors.



End: Not on exam

Project — Design a multifunctional ALU which can add / subtract / divide / multiply using least amount of hardware wherein the numbers are signed numbers represented in 2's complement format.

- You can use a few counters and MUX's but minimize the devices used.

- Numbers can have sign:
  - −
  - − +
  + −
  + +

- Show everything at gate level

W13 L11 5-14-18 Interrupts, DMA, Memory Systems, Cache, RAM, DISC

Today

1. One example from last week's topic
2. Memory — cache — placement strategies
3. I/O — main memory
4. DMA — secondary memory

spatial / temporal locality

In Class Exercise Show how you could do matrix multiplication on a chip with a $n \times n$ array of processors. Each processor has a multiplier, adder, and registers. (SIMD) (Example of Pipelining and Parallel Processing / Multiprocessing)

$$\begin{matrix} V_1 \\ V_2 \\ V_3 \end{matrix} \begin{bmatrix} a_{00} & a_{10} & a_{20} \\ a_{01} & a_{11} & a_{21} \\ a_{02} & a_{12} & a_{22} \end{bmatrix} \times \begin{bmatrix} b_{00} & b_{10} & b_{20} \\ b_{01} & b_{11} & b_{21} \\ b_{02} & b_{12} & b_{22} \\ Z_1 & Z_2 & Z_3 \end{bmatrix} = \begin{bmatrix} V_1 \cdot Z_1 & V_1 \cdot Z_2 & V_1 \cdot Z_3 \\ V_2 \cdot Z_1 & V_2 \cdot Z_2 & V_2 \cdot Z_3 \\ V_3 \cdot Z_1 & V_3 \cdot Z_2 & V_3 \cdot Z_3 \end{bmatrix}$$

$$V_1 \cdot Z_1 = (a_{00} \times b_{00}) + (a_{10} \times b_{01}) + (a_{20} \times b_{02})$$

$$V_1 \cdot Z_2 = (a_{00} \times b_{10}) + (a_{10} \times b_{11}) + (a_{20} \times b_{12})$$

$$V_1 \cdot Z_3 = (a_{00} \times b_{20}) + (a_{10} \times b_{21}) + (a_{20} \times b_{22})$$

$$V_2 \cdot Z_1 = (a_{01} \times b_{00}) + (a_{11} \times b_{01}) + (a_{21} \times b_{02})$$

$$V_2 \cdot Z_2 = (a_{01} \times b_{10}) + (a_{11} \times b_{11}) + (a_{21} \times b_{12})$$

$$V_2 \cdot Z_3 = (a_{01} \times b_{20}) + (a_{11} \times b_{21}) + (a_{21} \times b_{22})$$

$$V_3 \cdot Z_1 = (a_{02} \times b_{00}) + (a_{12} \times b_{01}) + (a_{22} \times b_{02})$$

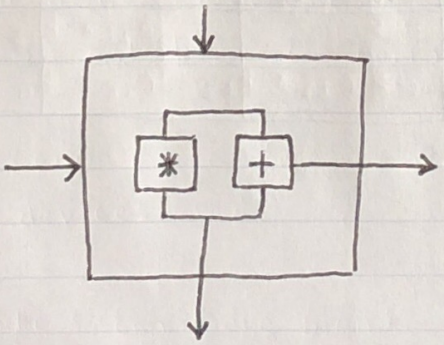
$$V_3 \cdot Z_2 = (a_{02} \times b_{10}) + (a_{12} \times b_{11}) + (a_{22} \times b_{12})$$

$$V_3 \cdot Z_3 = (a_{02} \times b_{20}) + (a_{12} \times b_{21}) + (a_{22} \times b_{22})$$

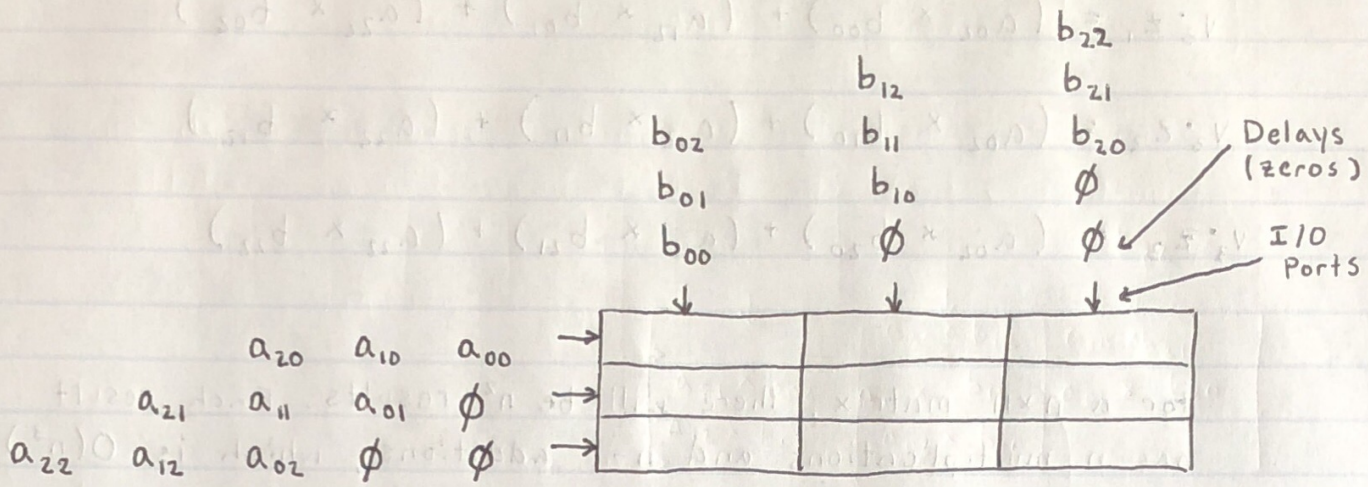
For a $n \times n$ matrix, there will be n^2 results. Each result has n multiplications and $n-1$ additions, which is $O(n^3)$ total operations. Given n^2 ALU's, should be able to solve in $O(n)$ time.

Every super step:

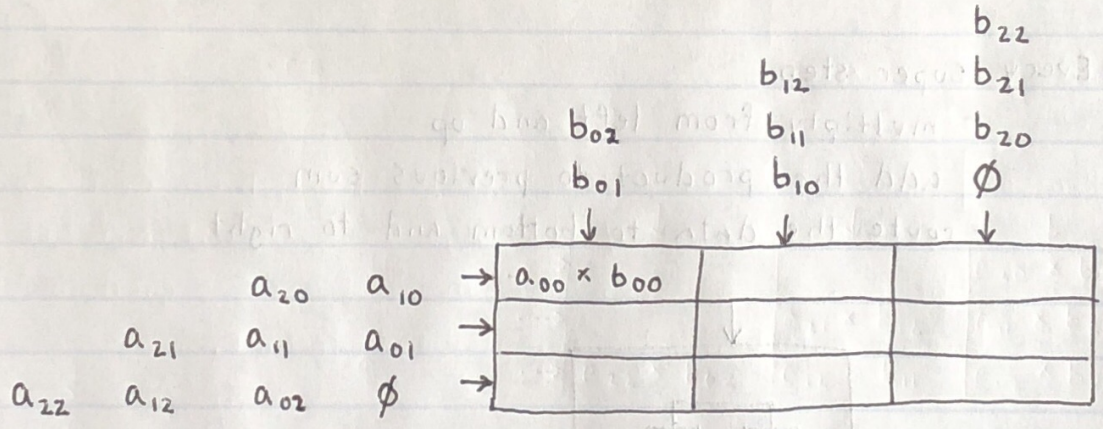
- multiply from left and up
- add the product to previous sum
- route the data to bottom and to right



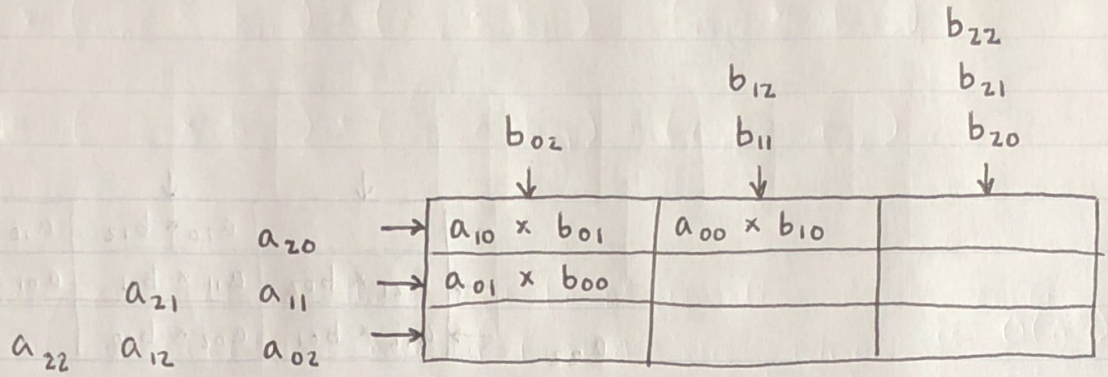
T = 0


b_{22}
b_{12} b_{21}
b_{02} b_{11} b_{20}
b_{01} b_{10} ϕ
b_{00} ϕ ϕ
Delays (zeros)
I/O Ports
a_{20} a_{10} a_{00}
a_{21} a_{11} a_{01} ϕ
a_{22} a_{12} a_{02} ϕ ϕ

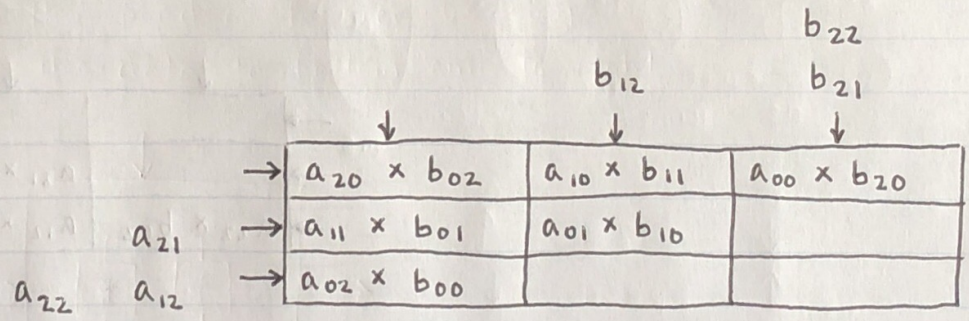
T = 1


b_{22}
b_{12} b_{21}
b_{02} b_{11} b_{20}
b_{01} b_{10} ϕ
a_{20} a_{10}
$a_{00} \times b_{00}$
a_{21} a_{11} a_{01}
a_{22} a_{12} a_{02} ϕ

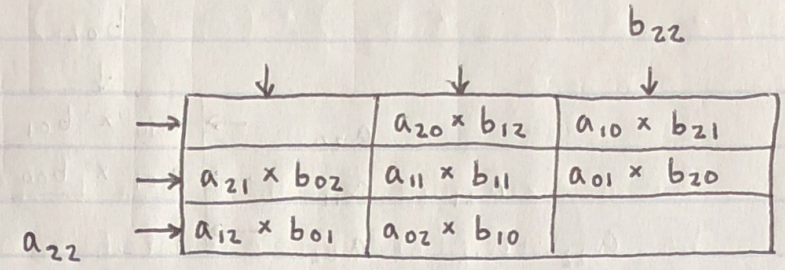
T = 2


b_{22}
b_{12} b_{21}
b_{02} b_{11} b_{20}
a_{20}
a_{21} a_{11}
a_{22} a_{12} a_{02}
$a_{10} \times b_{01}$
$a_{00} \times b_{10}$
$a_{01} \times b_{00}$

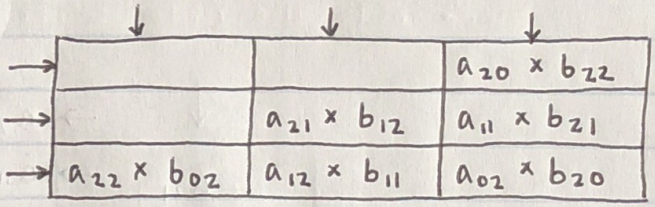
T = 3


b_{22}
b_{12} b_{21}
$a_{20} \times b_{02}$
$a_{10} \times b_{11}$
$a_{00} \times b_{20}$
a_{21}
$a_{11} \times b_{01}$
$a_{01} \times b_{10}$
a_{22} a_{12}
$a_{02} \times b_{00}$

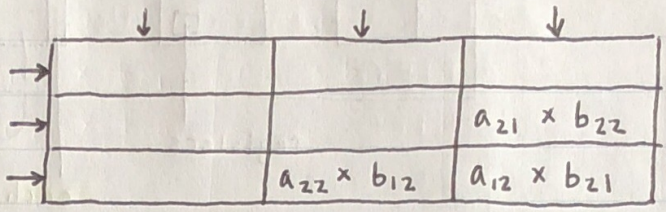
T = 4


b_{22}
a_{22}
$a_{20} \times b_{12}$
$a_{10} \times b_{21}$
$a_{21} \times b_{02}$
$a_{11} \times b_{11}$
$a_{01} \times b_{20}$
$a_{12} \times b_{01}$
$a_{02} \times b_{10}$

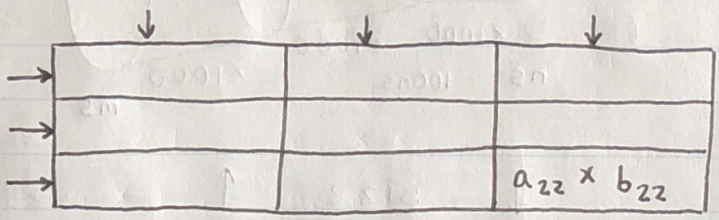
T = 5


$a_{20} \times b_{22}$
$a_{21} \times b_{12}$
$a_{11} \times b_{21}$
$a_{22} \times b_{02}$
$a_{12} \times b_{11}$
$a_{02} \times b_{20}$

T = 6



T = 7

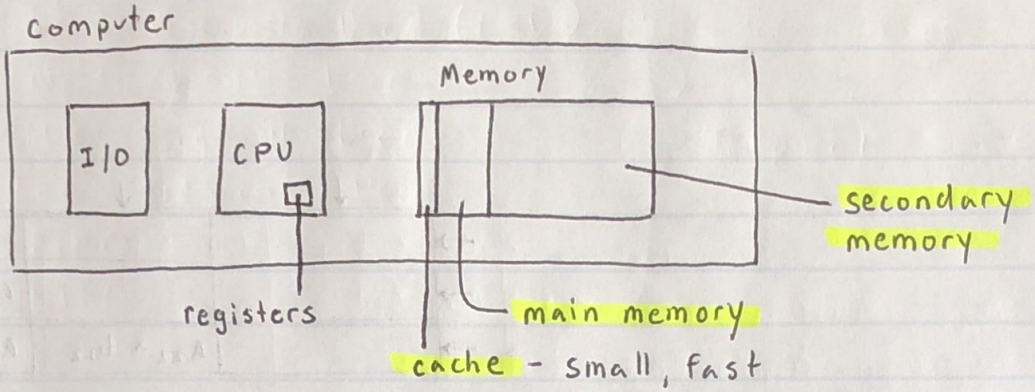


For a $n \times n$ matrix this solution will be done in $2n + 1 = 2(3) + 1 = 6 + 1 = 7$ steps.

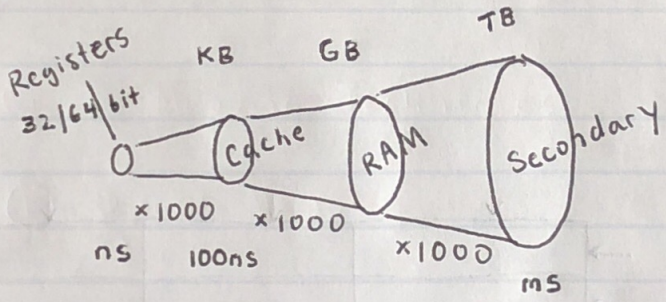
$$\text{Speedup} = \frac{\text{unimproved}}{\text{improved}} = \frac{O(n^3)}{O(n)} = n^2$$

Speedup is equal to the number of processors, ∴ the solution is linear speedup optimal.

Memory



- Hierarchical Memory Structure



↓ size ↑
↑ speed ↓
↑ price/byte ↓

Analogy

- fridge = cache
- corner 7-11 = main memory
- huge walmart = secondary

- If data is not in cache, check main memory, if not there, go to secondary memory

- **Unit of access**: how much is moved between each level

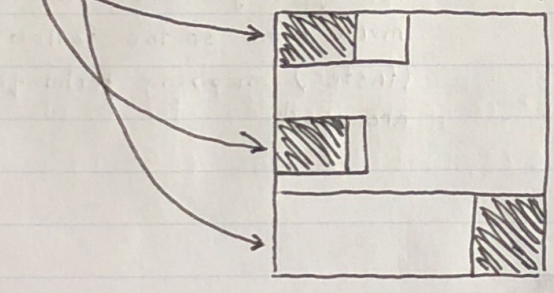
- **Spatial locality**: pick things that are in the same space / neighborhood

- **Temporal locality**: concept of time, making decision based on time

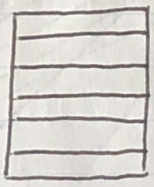
Placement Strategies (there is room)

spatial

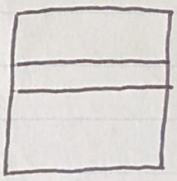
1. **First Fit** - fast, find first one that fits
2. **Best Fit** - lots of small fragmentations so not actually the best
3. **Worst Fit** - some people say this is best b/c you take a big area and just keep putting things in, end up w/ a lot of space open



- **Pages**: fixed size



- **Segmentation**: variable size



Replacement Strategies (there is no room, decide who to replace)

- Least Recently Used (LRU)
- Most Recently Used
- Most Frequently Used
- Least Frequently Used
- FIFO
- LIFO
- Random

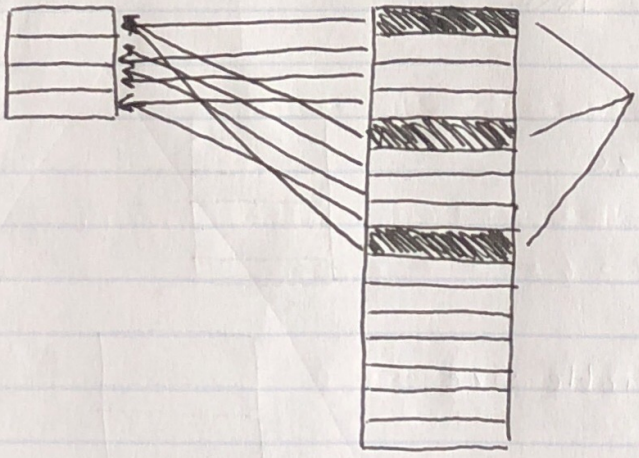
} Temporal

Used for:

- Secondary → Main Memory
- Main Memory → Cache → for this the replacement stategy algorithms take too much time so the following (faster) mapping techniques are used

Cache Mapping Techniques

(1) Direct Mapping

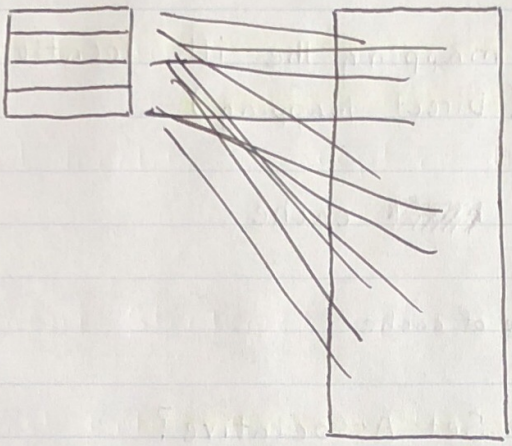


all map to same spot in cache

spaces in memory

Drawback: multiple [redacted] can only be mapped to the same location in the cache, meaning they can't both be in the cache at the same time

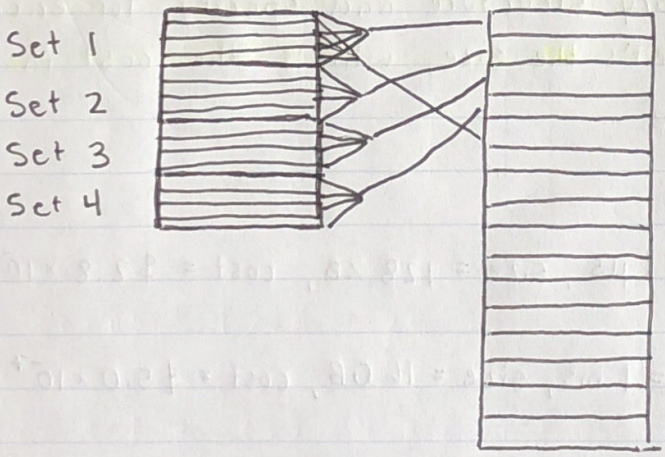
(2) Fully Associated



Function:

- anywhere
- "random"

(3) Set Associative



In Class Exercise

① What is the function for mapping the ith location of main memory to cache? (Direct Mapping)

Assume k blocks in cache

Function: $i \bmod k$

(i: main memory address; k: size of cache)

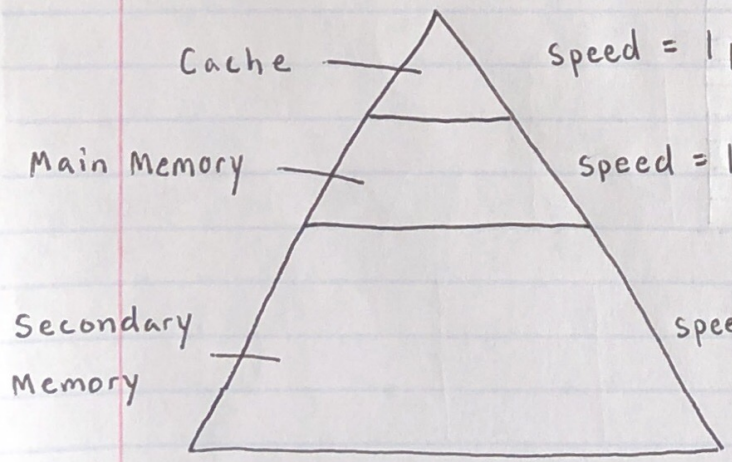
② What is the function for Set Associative?

Assume S sets in cache

Function: $i \bmod S$ gives you the set to map to, then it can go anywhere within that set

(i: main memory address; S: # sets)

③ Draw a hierarchical memory structure and specify for each level what's the speed, what's the size, what's the cost in dollars / byte. (speed: access time)



Cache: speed = 1 µs, size = 128 KB, cost = $2.8 × 10^-3 / byte

Main Memory: speed = 1 ms, size = 16 GB, cost = $5.0 × 10^-7 / byte

Secondary Memory: speed = 1 s, size = 1 TB, cost = $1 × 10^-12 / byte

- Approx 10^3 difference in speed, size, cost between levels

keyboard - order of seconds
cpu - order of ns

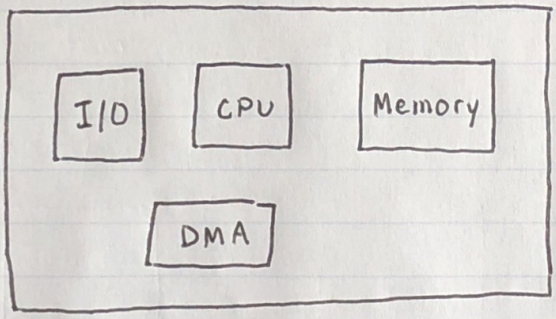
I/O - extremely slow

↓

interrupts:
- when an I/O request comes in, the processor gets interrupted
- processor puts contents of what it was working on on a "hold" register then resumes later
- analogy: put phone call on hold, switch, resume
- processor can also be interrupted b/c of infinite loops, divide by zero errors, etc.

Direct Memory Access (DMA)

- "another processor" dedicated for accessing memory / I/O so don't have to interrupt processor



Example: Print location 100 of memory (I/O request)

↓

would have to interrupt CPU if no DMA