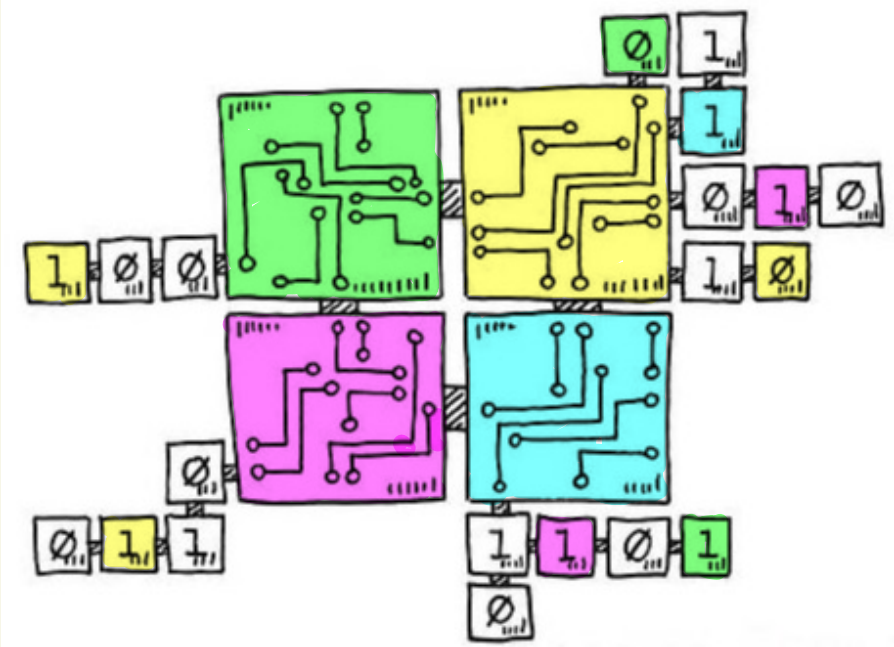
IE2104



Number System

- BIT (BInary digIT) → 0 (False) & 1 (True)
- Binary$_2$ (0, 1) → 3-bit string (000, 001, 010, 011 ...), 4 bit string (0000, 0001, 0010, ...)
- Decimal$_{10}$ (0–9)
- Octal$_8$ (0–7) → 0, 1, 2, 3, 4, 5, 6, 7, 10, 11, 12, 13, 14, 15 ...
- Hexadecimal$_{16}$ (0–F) → 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

Binary to Decimal

E.g. 4-bit binary number 1011_2 is

$1\times2^3+0\times2^2+1\times2^1+1\times2^0 = 11_{10}$ → decimal

- 1011_2 → binary
- position −1 [1 − 1 = 0]
- for octal this is 8
- most significant bit (MSB)
- least significant bit (LSB)

hexadecimal to decimal

E.g. $B5E_{16} = 11\times16^2+5\times16^1+14\times16^0$ (positions 2 1 0)

$= 2910_{10}$

9	A	B	C	D	E	F
	10	11	12	13	14	15

Division to get smt from decimal

E.g.

Quotient	Remainder
168	
84	0
42	0
21	0
10 (21 ÷ 2)	1 (remainder)
5	0
2	1
1	0
0	1

write it backwards & that is the ans

$168_{10} = 10101000_2$

E.g.

Quotient	Remainder
2793	
174	9
10	14 (E)
0	10 (A)

$2793_{10} = AE9_{16}$

Conversion bw binary & octal

↳ Each Octal digit equals to 3 bits

$3\ 6\ 2_8$ → $011\ 110\ 010_2$

$010\ 101\ 110$ → $2\ 5\ 6_8$

000	0
001	1
010	2
011	3
⋮	⋮

If have 10100, just add 0s infront → 010 100 → groups of 3

Conversion bw binary & hex

↳ Each hexadecimal digit equals to 4 bits

$E\ 2\ 8\ 0_{16}$ → $1110\ 0010\ 1000\ 0000_2$

$F\ B\ 1_{16}$ → $1111\ 1011\ 0001_2$

Representation of Signed Number

↳ Normally use −/+ to represent −ve & +ve → binary uses smt else:

1. Signed & Magnitude Representation
   - If MSB = 0 → number +ve [01101 = +13]
   - If MSB = 1 → number −ve [11101 = −13]
     - first no. not included in value calculations → do not affect value
   - Disadvantage: a) 2 patterns represent 0 [000 = +0, 100 = −0]

b) Handle sign bit separately

2. One's Complement Representation

- Positive & corresponding negative no. complement each other. (Logic NOT)
  - E.g. $01101 = +13$ → 0 infront means ⊕ve
    $10010 = -13$ → all the bits are inverted (0→1, 1→0)
- Advantages: Symmetry & ease of complementation
- Disadvantages: 2 patterns represent 0 [$000 = +0$, $111 = -0$]

3. Two's Complement Representation

- Positive & negative numbers are their corresponding 1's complement number +1.
  - E.g. 01101 (complement first then add 1) $= +13$
    $10010 + 1 = 10011 = -13$
    $1111+1 = 10000 = +0$ (discarded Caluags like that)
- Value of 2's Complement Number of 11011

$$\text{Value} = (-1)1\times2^4 + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$$
$$= -16 + 8 + 0 + 2 + 1 = -5$$

Binary Addition

E.g.

carry in: 1 1 1 1 1 1 1 0

	1 0 1 1 1 1 1 1 0	190
+	1 0 0 0 1 1 0 1	+ 141
	1 0 1 0 0 1 0 1 1	331

Binary Subtraction

E.g. borrow: 0 0 1 1 0

	1 X X 0 0 X 0 1	229
−	0 0 1 0 1 1 1 0	− 46
	1 0 1 1 0 1 1 1	183

Two's Complement Addition & Subtraction

- Any carry beyond MSB ignored.

E.g.

	1 1 1 1 0
+	1 0 1 0
	1 0 1 0 0 0

ignore ← 10000 → answer

$-2 \longrightarrow 0010 \rightarrow +2$
$+-6$ (−8)
$1101+1 = 1110 = -2$

→ if $-2-6$ → from +2 & +6 changed both to ⊖ve → add

E.g. $85_{16} - 3B_{16} = 85_{16} + (-3B_{16})$

$3B_{16} = 0011\ 1011$
$-3B_{16} = 1100\ 0101 = C5_{16}$

	8 5 16
+	C 5 16
discard carry →	1 4 A 16 → Ans

Value Range

For n-bit binary, range is:

Unsigned	$0 \longrightarrow +2^n - 1$
1's complement	$-(2^{n-1}-1) \rightarrow +2^{n-1}-1$
2's Complement	$-2^{n-1} \rightarrow +2^{n-1}-1$

- Addition of 2 number different sign → never overflow. But if addition of same sign → get a different sign → overflow

Binary – Coded Decimal (BCD) [8421 code]

↳ Each decimal digit represented by 4 bit. Weight for BCD bit are 8, 4, 2 & 1.

0	0000
1	0001
2	0010
3	0011
4	0100

5	0101
6	0110
7	0111
8	1000
9	1001

* Binary 1010 to 1111 not used ↘ 10 ↙ 15

E.g. 38_{10} → 0011 1000 0110 (3, 8, 6)

combined bit not binary number ✱

2421 Code

↳ weights of 2421.

↳ Advantage: self complementing, 2 BCD number with sum of 9 have 2421 code complement each other

E.g. $1011 \rightarrow 2\times1 + 4\times0 + 2\times1 + 1\times1 = 5$

Excess-3 code

↳ BCD code plus 3, self-complementing for the sum of 9 for BCD.

↳ BCD (0000) → Excess-3 (0011) ↳ calculate same way as BCD

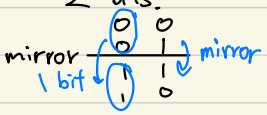
Gray Code

↳ Adjacent numbers have only 1 bit different, not a weighted code

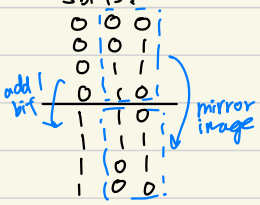
Gray Code	Decimal
0000	0
0001	1
0011	2
0010	3
0110	4
0111	5
0101	6
0100	7

↳ Gray code can be produced:

2 bits:



3 bits:

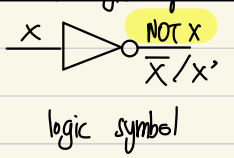


ASCII Code

Digital Circuits

Truth table
- list all the combinations of input & output
- number of rows is 2^n → n is the number of variables. Binary combinations obtained from $0-(2^n-1)$
- Basic logic operations → AND, OR, NOT

NOT logic operation (inversion operation)

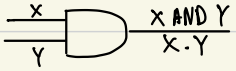


logic symbol

X	NOT X
0	1
1	0

← truth table

AND logic operation

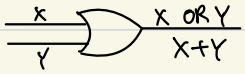


logic symbol

X	Y	XY
0	0	0
0	1	0
1	0	0
1	1	1

← truth table → multiply ($0\times0=0$, $0\times1=0$)

OR logic operation

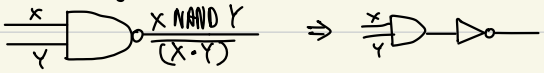


logic symbol

X	Y	X+Y
0	0	0
0	1	1
1	0	1
1	1	1

← truth table → addition ($0+0=0$, $0+1=1$)

NAND logic operation (not AND) → faster than AND gate

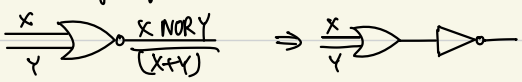


logic symbol

X	Y	$\overline{(X\cdot Y)}$
0	0	1
0	1	1
1	0	1
1	1	0

← logic table

NOR logic operation (not OR) → faster than OR gate

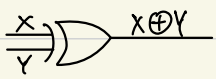


logic symbol

X	Y	$\overline{(X+Y)}$
0	0	1
0	1	0
1	0	0
1	1	0

← logic table

XOR (Exclusive-OR) logic operation



logic symbol

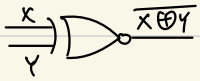
X	Y	X⊕Y
0	0	0
0	1	1
1	0	1
1	1	0

→ change last value X+Y

X	Y	X+Y
1	1	(1)

inverse

XNOR (Exclusive-NOR) logic operation



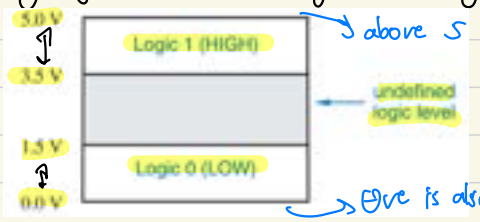
X	Y	$\overline{X\oplus Y}$
0	0	1
0	1	0
1	0	0
1	1	1

→ change last value of X+Y ★

CMOS Logic

↳ CMOS → Complementary Metal-Oxide Semiconductor Field-Effect Transistor (CMOSFET) ← transistor called this

↳ Any logic circuit → range of voltages interpreted as logic 0 & another non-overlapping range → logic 1



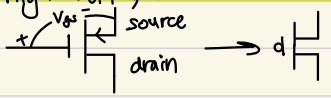
↳ above 5 is also 1

↳ below is also 0

↳ may damage transistor

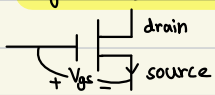
1) p-channel MOS (PMOS)

↳ high volt, '1' → transistor will be "OFF"



2) n-channel MOS (NMOS) transistor

↳ high volt, '1' → transistor will be "ON"

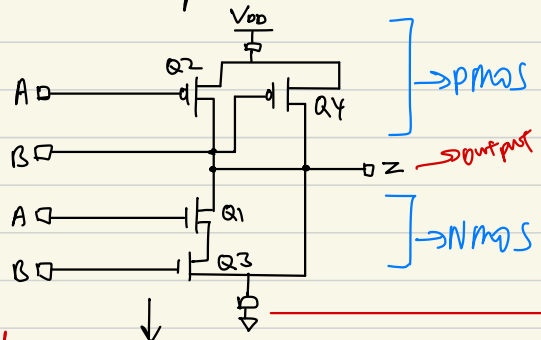


For NMOS circuit: OR / + → parallel

AND / X → series

NMOS & PMOS circuit complementary → If NMOS //, PMOS series & vice versa

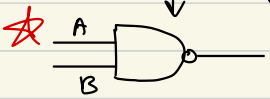
CMOS 2-input NAND Gate



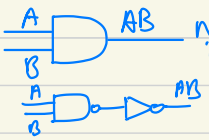
function table ← Z output

A	B	Q1	Q2	Q3	Q4	Z
L	L	OFF	ON	OFF	ON	H
L	H	OFF	ON	ON	OFF	H
H	L	ON	OFF	OFF	ON	H
H	H	ON	OFF	ON	OFF	L

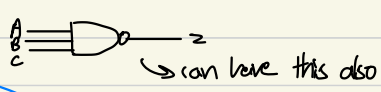
If both Q1 & Q3 ON & Q2 ON → Z will still be low as current will go to ground



If want this



need add inverter (6 transistors)

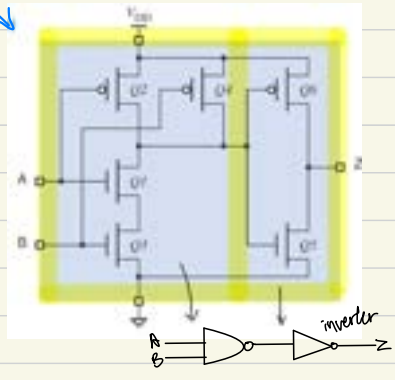


↳ can have this also

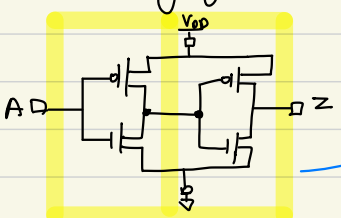
Fan-in

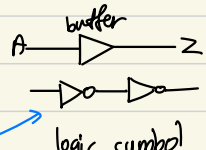
↳ number of inputs that gate can have in a particular logic family

↳ fan-in limit of CMOS gates → 4 for NOR gates, 6 for NAND gates



Non-inverting gate

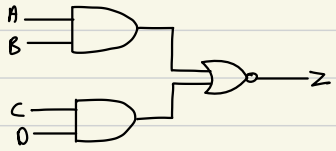




output = input

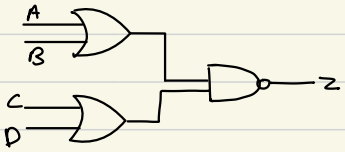
logic symbol

CMOS AND-OR-INVERT gate



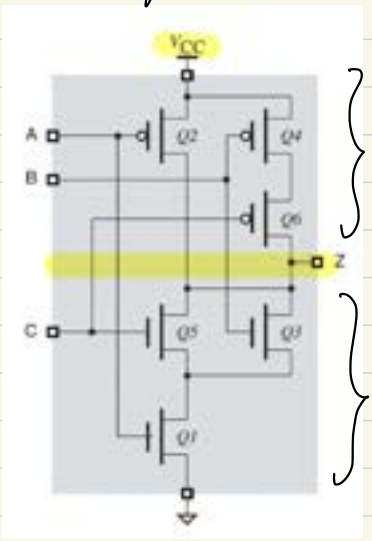
$\overline{(CD+AB)}$

CMOS OR-AND-INVERT gate

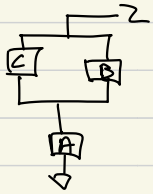


$\overline{(A+B)\cdot(C+D)}$

CMOS equation



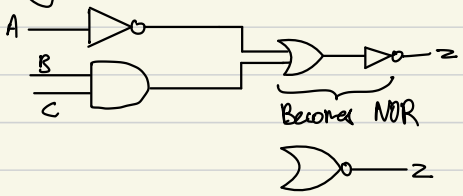
→ To get equation, just look at NMOS circuit
1) Draw out circuit & write out according to input



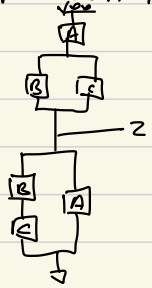
→ C & B // , A in series with B+C

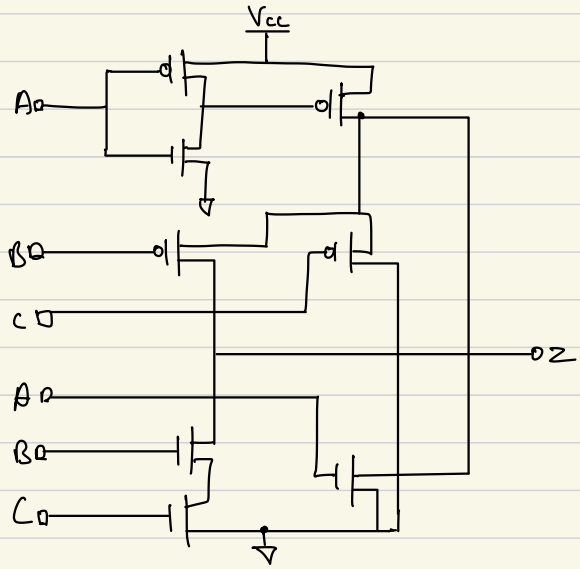
$\overline{A\cdot(C+B)}$ ← add inverse

E.g. Design a CMOS circuit that has the functional behaviour shown below. (Only eight transistors are required.)



$\overline{A+BC}$





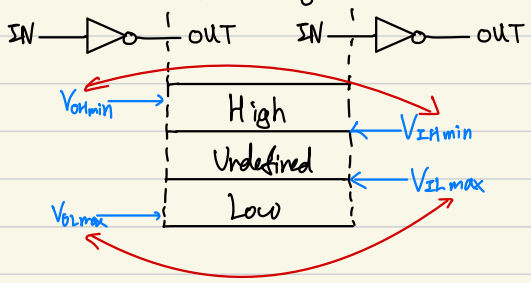
Input/Output Voltages

a) $V_{OH min}$: The minimum output voltage for the HIGH state

b) $V_{IH min}$: The minimum input voltage recognised as HIGH

c) $V_{IL max}$: The maximum input voltage recognised as LOW.

d) $V_{OL max}$: The maximum output voltage for the LOW state.



→ DC noise margin is a measure of how much noise it takes to corrupt a worst-case output voltage into a value that may be recognised properly by an input

→ Low-state DC noise margin = $V_{IL max} - V_{OL max}$

→ High-state DC noise margin = $V_{OH min} - V_{IH min}$

who is more sensitive to noise → lower voltage diff

Input Current

→ I_{IH} = The maximum current that flows into the input in HIGH state.

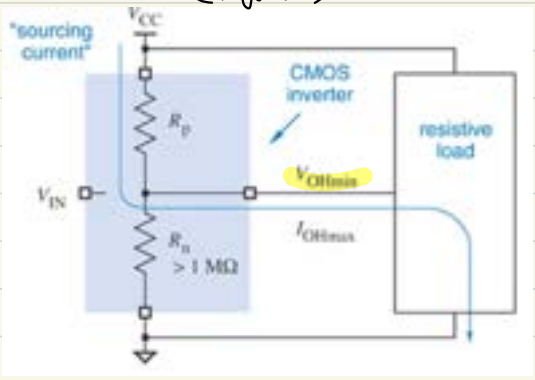
→ I_{IL} = The maximum current that flows into the input in LOW state.

→ $I_{OL max}$ = The maximum current that the output can sink in the LOW state while still maintaining an output voltage no greater than $V_{OL max}$. (Figure 2)

→ $I_{OH max}$ = The maximum current that the output can source in the HIGH state while still maintaining an output voltage no less than $V_{OH min}$. (Figure 1)

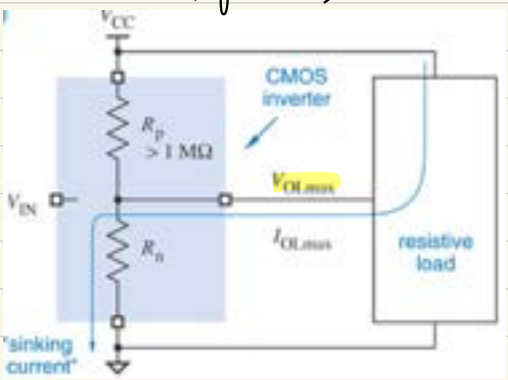
if not will get wrong logic

(Figure 1)



cannot source too much current

(Figure 2)



cannot sink too much current

Fanout

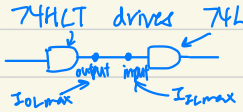
→ Number of inputs that the gate can drive without exceeding its worst-case loading specifications (logic level still correct)

→ High state: fanout = $\frac{I_{OH max}}{I_{IH max}}$

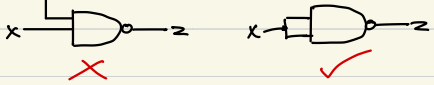
→ Low state: fanout = $\frac{I_{OL max}}{I_{IL max}}$

★ fanout = min of the 2.

74HCT drives 74LS

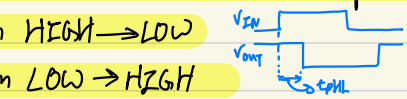


* Unused CMOS input should never be left unconnected. CMOS input have high impedance → only takes small amount of circuit noise to make input look high!

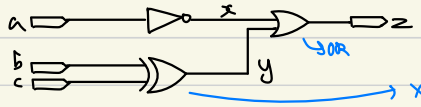


Transition time

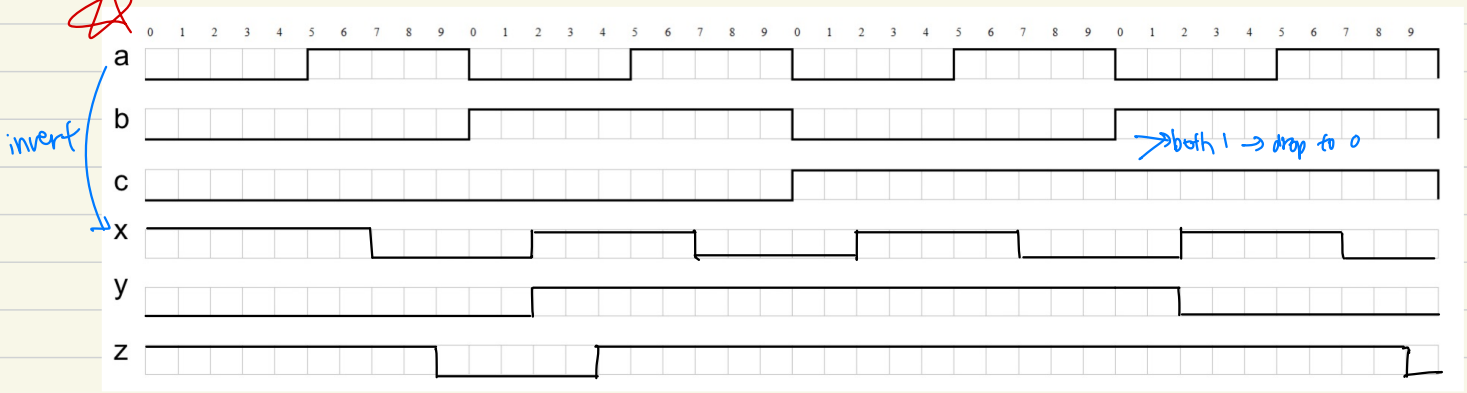
- Amount of time that output of logic circuit takes to change from one state to another:
  - a) LOW to HIGH → rise time (t_r)
  - b) HIGH to LOW → fall time (t_f)
- Propagation delay (t_p) → amount of time it takes for input signal to produce a change in output signal:
  - a) t_{pHL} → output Δ from HIGH → LOW
  - b) t_{pLH} → output Δ from LOW → HIGH



E.g. Given the logic circuit below, complete waveforms x, y, z → propagation delay of 2ns for all gates

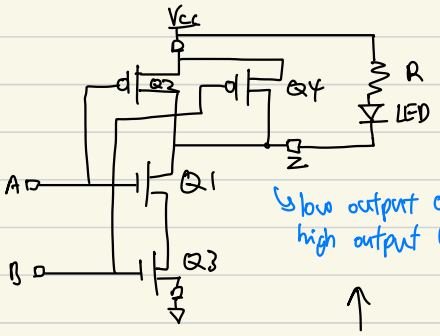


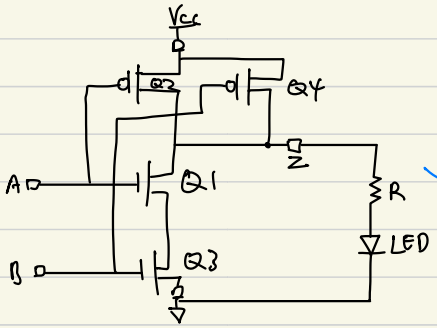
XOR (X | Y | X⊕Y : 1 | 1 | 0)



Driving LEDs

- A LED can be turned "on" with an ordinary CMOS output



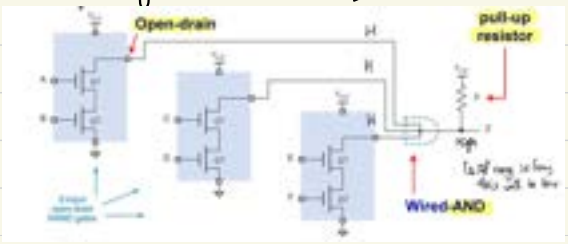


E.g. A particular has a voltage drop of ~2V in the "on" state & requires ~5mA of current for normal brightness. Determine an appropriate value for the pull-up resistor when the LED is connected to a NAND gate, which has $V_{OL\,max}$ of 0.37V with V_{CC} = 5V

Voltage drop across R: $V_R = 5 - 2 - 0.37 = 2.63V$

$R = \frac{V}{I} = \frac{2.63}{0.005} = 526\Omega$ → single 510Ω resistor can be used

Wired Logic (AND function)



CMOS Logic families

* 74FAMnn
  - FAM: alphabetic family
  - nn: numeric function designation → different fam but same this → perform same function

→ The first two 74-series CMOS families are:

a) HC (High-speed CMOS)

b) HCT (High-speed CMOS, TTL compatible)

↳ Designed to recognise CMOS input level → can use any power-supply voltage 2–6V

↳ higher voltage is used for higher speed & lower voltage for lower power dissipation

$P = C_{PD} \cdot V_{DD}^2 \cdot f$ (V_{DD} → voltage; f → transition frequency)

P = power dissipation; C_{PD} = power-dissipation capacitance

→ Two most recent & probably most versatile CMOS families are:

a) AHC (Advanced high-speed CMOS)

b) AHCT (Advanced high-speed CMOS, TTL compatible)

↳ 2–3 times as fast as HC/HCT

→ For discrete logic families → trend is to produce parts that operate & produce output at lower voltage but can also tolerate inputs at the higher voltage → 3.3-V CMOS families can operate with 5-V CMOS & TTL families

→ basic CMOS series for 3.3-V category & their designations follows:

a) 74LV & 74LVC → low-voltage CMOS

b) 74AHVCMOS → Advanced low-voltage CMOS

Bipolar logic

→ perform logic operations (diode logic)

↳ use semiconductor diodes & Bipolar Junction Transistors (BJT) as the basic building blocks of logic circuits

↳ original Transistor-Transistor Logic (TTL) families used transistors both to perform logic functions & to boost their output drive capability

↳ Emitter-Coupled Logic (ECL) families used transistors as current switches to achieve very high speed.

under

Transistor-Transistor Logic

↳ Advantage: not sensitive to electrostatic discharge → CMOS is

↳ use same logic levels as TTL-compatible CMOS → low: 0–0.8V, high: 2.0–5.0V

Single-Variable Theorems

↳ AND operation → logic multiplication (·)

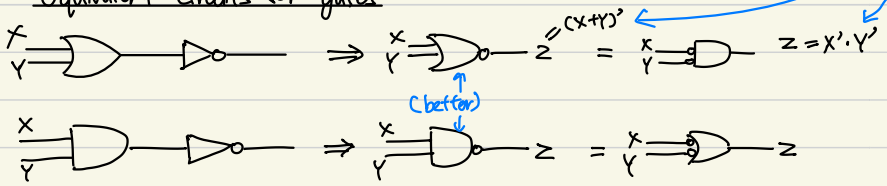
↳ OR operation → logic addition (+)

1. Identity elements → $X+0=X$ & $X \cdot 1 = X$
2. Null elements → $X+1=1$ & $X \cdot 0 = 0$
3. Idempotency → $X+X=X$ & $X \cdot X = X$
4. Complements → $X+X'=1$ & $X \cdot X' = 0$

Two & Three-Variable Theorems

1. Distributivity → $X \cdot Y + X \cdot Z = X \cdot (Y+Z)$ & $(X+Y) \cdot (X+Z) = X + Y \cdot Z$
2. Covering → $X + X \cdot Y = X$ & $X \cdot (X+Y) = X$ [Absorption law]
3. Combining → $X \cdot Y + X \cdot Y' = X$ & $(X+Y) \cdot (X+Y') = X$
4. Consensus → $X \cdot Y + X' \cdot Z + Y \cdot Z = X \cdot Y + X' \cdot Z$
   → $(X+Y) \cdot (X'+Z) \cdot (Y+Z) = (X+Y) \cdot (X'+Z)$
5. DeMorgan Theorem → "Break the Bar & change the sign"
   → $\overline{x+y} = \bar{x} \cdot \bar{y}$ & $\overline{x \cdot y} = \bar{x} + \bar{y}$

Equivalent Circuits for gates



Operator Precedence

1. Parentheses
2. NOT
3. AND
4. OR

e.g. $f = x \cdot z + x \cdot z'$

Precedence: z', $x \cdot y$, $x \cdot z'$, $x \cdot y + x \cdot z'$

2-Variable Function From Truth Table

↳ Boolean Function can be derived from Truth Table → expression has all the 1 rows OR tgt

x	y	f	
0	0	0	
0	1	1	→ $\bar{x}y = 1$
1	0	0	
1	1	1	→ $xy = 1$

$f = x'y + xy$ (all those with $f=1$)

$f' = x'y' + xy'$ (all those with $f=0$)

$f = \overline{x'y' + xy'} = (\overline{x'y'})(\overline{xy'})$ → De-Morgan

$= (\bar{\bar{x}} + \bar{\bar{y}})(\bar{x} + \bar{y})$

$= (x+y)(x'+y) = x'y + xy$

→ sum-of-products (SOP) → $x' + x \cdot y' + x' \cdot y \cdot z$

→ product-of-sums (POS) → $x' \cdot (x+y') \cdot (x'+y+z)$

Minterm (SOP)

↳ minterm is a product which consists of all variables in normal/complement form but NOT BOTH

e.g. 2 variables → $x \cdot y'$ is minterm, but x' is NOT a minterm

3 variables → $x'yz'$ is minterm, but xy' is NOT a minterm

Maxterm (POS)

↳ sum in which each variable appears once & only once either in its normal form / its complement form but NOT BOTH

e.g. 2 variables → $x'+y'$ / $x'+y$ / $x+y'$ / $x+y$

Minterm Boolean Expression

	x	y	z	f_1	f_2
m_0	0	0	0	0	0
m_1	0	0	1	1	0
m_2	0	1	0	0	1
m_3	0	1	1	0	0
m_4	1	0	0	1	1
m_5	1	0	1	0	0
m_6	1	1	0	1	1
m_7	1	1	1	0	1

$f_1(x,y,z) = m_1 + m_4 + m_6$ → these when $f=1$

$= \Sigma m(1,4,6) = \Sigma(1,4,6)$

$= x'y'z + xy'z' + xyz'$

$f_2(x,y,z) = m_2 + m_4 + m_6 + m_7$

$= \Sigma m(2,4,6,7) = \Sigma(2,4,6,7) = 1$ (canonical sum)

Maxterm Boolean Expression

	x	y	z	f_1	f_2	f_1'
M_0	0	0	0	0	0	1
M_1	0	0	1	1	0	0
M_2	0	1	0	0	1	1
M_3	0	1	1	0	0	1
M_4	1	0	0	1	1	0
M_5	1	0	1	0	0	1
M_6	1	1	0	1	1	0
M_7	1	1	1	0	1	1

$f_1 = M_0 \cdot M_2 \cdot M_3 \cdot M_5 \cdot M_7$ → take these where $f=0$

$= \pi M(0,2,3,5,7) = \pi(0,2,3,5,7)$ → 011 → 0, so $x+y'+z'$

$= (x+y+z)\cdot(x+y'+z)\cdot(x+y'+z')\cdot(x'+y+z')\cdot(x'+y'+z')$ → can be converted from f_1' (min term) & use De-morgan

$f_2 = M_0 \cdot M_1 \cdot M_3 \cdot M_5$

$= \pi M(0,1,3,5) = \Sigma(2,4,6,7)$

= 0 (canonical product)

Conversion to minterms

E.g. $f(a,b,c) = a' + bc' + ab'c$.

To convert a' to minterm, 2 variables (b,c) must be added without changing the function

$a' = a'\cdot 1$ & $1 = b+b'$ → $a'(b+b') = a'b + a'b'$

$a'b = a'b(c+c') = a'bc + a'bc'$ $\qquad bc' = bc'(a+a') = abc' + a'bc'$

$a'b' = a'b'(c+c') = a'b'c + a'b'c'$

$f = a'bc + a'bc' + a'b'c + a'b'c' + abc' + a'bc' + ab'c$ → same can cancel & just write one

(011, 001, 000, 110, 101)

$= a'bc + a'b'c + a'b'c' + abc' + ab'c$

$= \Sigma(0,1,3,5,6) = \pi(4,7)$

Express Boolean Function in Maxterms

E.g. $f(a,b,c) = a' + bc'$

$= (a'+b)(a'+c')$ → Distribution Law

$= (a'+b+cc')(a'+c'+bb')$ (0, 0)

$= (a'+b+c)(a'+b+c')(a'+c'+b)(a'+c'+b')$

$= (a'+b+c)(a'+b+c')(a'+c'+b')$ (1,0,0 / 1,0,1 / 1,1,1)

$= \pi(4,5,7) = \Sigma(0,1,2,3,6)$

Boolean Function Manipulation E.g.

$f = x\cdot y' + x\cdot y\cdot z + x'\cdot z = x(y'+yz) + x'\cdot z$

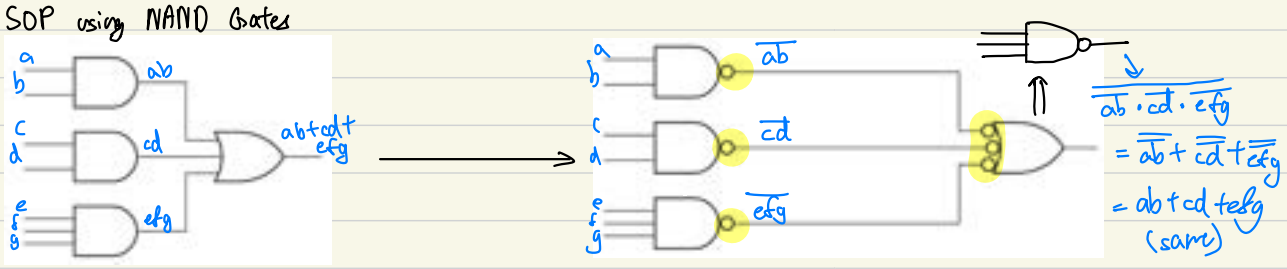
$= x[(y'+y)\cdot(y'+z)] + x'z = xy' + xz + x'z$ (y+y' = 1; Distribution Law)

$= x\cdot y' + (x+x')\cdot z = xy' + z$ (x+x' = 1)

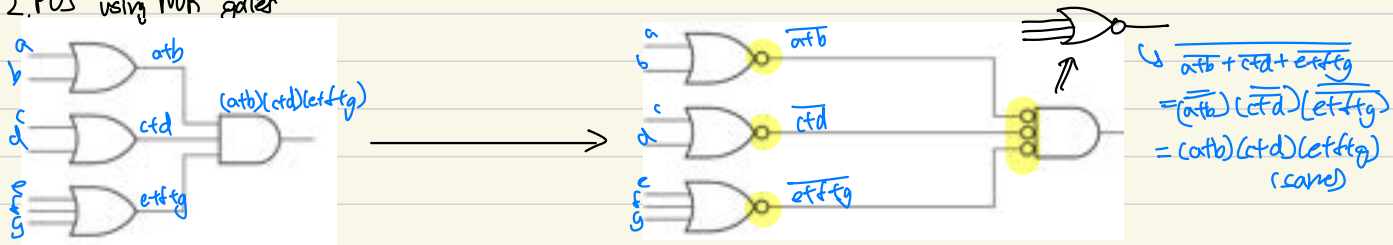
Circuit Manipulations

↳ we like to use **NAND & NOR gates → faster than AND & OR gates**

1. SOP using NAND Gates



2. POS using NOR gates



K-map (Karnaugh Map)

↳ Diagram made up of squares → each square represents exactly one minterm of the logic function

1. 2-variable K-map

b \ a	0	1
0	m_0	m_2
1	m_1	m_3

$f = a'b + ab' = 1$ (01 / 10) — fill in 1 in K-map for these → rest put 0

$f = \Sigma m(1, 2)$

b \ a	0	1
0	0	1
1	1	0

2. 3-variable K-map

↳ 8 squares → binary sequence of abc follows **Gray code → enable logic minimisation**

c \ ab	00	01	11	10
0	m_0	m_2	m_6	m_4
1	m_1	m_3	m_7	m_5

$f = \Sigma m(2, 6)$

$= a'bc' + abc' = bc'$

c \ ab	00	01	11	10
0	0	1	1	0
1	0	0	0	0

Can group together → ignore different one & get bc'

Logic minimisation

a	b	c	f
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

$f = \Sigma m(1, 3, 4, 6) = 001 + 011 + 100 + 110$

$= a'b'c + a'bc + ab'c' + abc'$

$= a'c(b' + b) + ac'(b' + b)$

$= a'c + ac'$

fill it in using truth table

c \ ab	00	01	11	10
0	0	0	1	1
1	1	1	0	0

110 & 100 → ac'

001 & 011 → $a'c$

K-map covering

↳ We can group 1, 2, 4, 8 squares

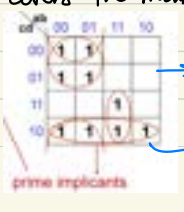
↳ A function is said to have been **covered** if all logic-1 squares are grouped

→ Implicants

↳ each group gives a **product term** & is called an implicant of the function

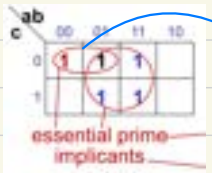
1. **Prime implicants** (group this 1st)

↳ group that covers the maximum possible number of adjacent squares



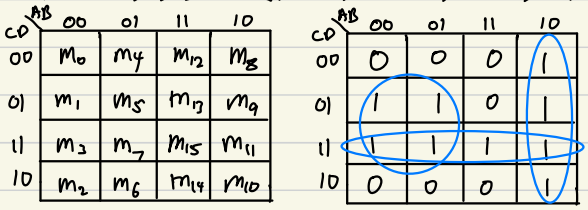
2. Essential Prime Implicant (group this 2nd)

↳ prime implicant that covers a minterm which is not covered by any other prime implicants.



need to be covered so need draw another circle

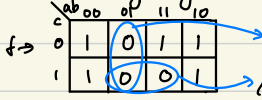
E.g. Plot $f=(A,B,C,D)=\Sigma m(1,3,5,7,8,9,10,11,15)$ on K-map & list all implicants. (Prime & essential prime)



∴ 9+12+4 = 25 ∴ 25 implicants total

When group 1 individual by itself → 4 variable implicant (e.g. $AB\bar{C}D$) → 9
When group 2 tgt → 3 variable implicants ($\bar{A}\bar{C}D$, $A\bar{B}D$) → 12
When group 4 tgt → 2 variable implicant ($A\bar{B}$, $\bar{B}D$, CD, $A\bar{D}$) → 4
When group 8 tgt → 1 variable implicant

→ Covering logic-1 squares in K-map gives us logic function in sum-of-products form
→ Covering logic-0 squares gives us logic functions in product-of-sum form

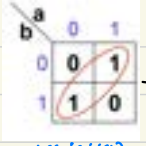


001 & 011 → $(a+b')$
111 & 011 → $(b'+c')$

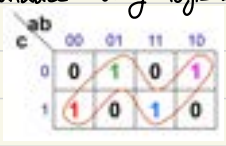
$f=(a+b')(b'+c')$

K-map for XOR & XNOR

1. XOR (3-letter) → Odd Function (odd no. of variables having logic 1 → logic 1)

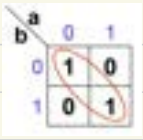


→ $ab'+a'b=a\oplus b$

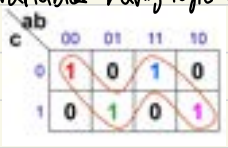


→ $a\oplus b\oplus c$

2. XNOR (4-letter) → Even Function (even no. of variables having logic 1 → logic 0)



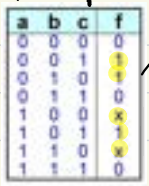
→ $a'b'+ab=(a\oplus b)'$



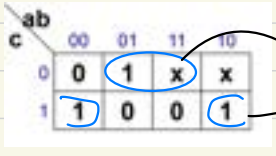
→ $(a\oplus b\oplus c)'$

Don't-Care Conditions

↳ Don't-Care conditions usually denoted by 'X' / 'd'
↳ K-map enter X into don't care squares → can treat as either 0/1 → make use to either to make larger groups



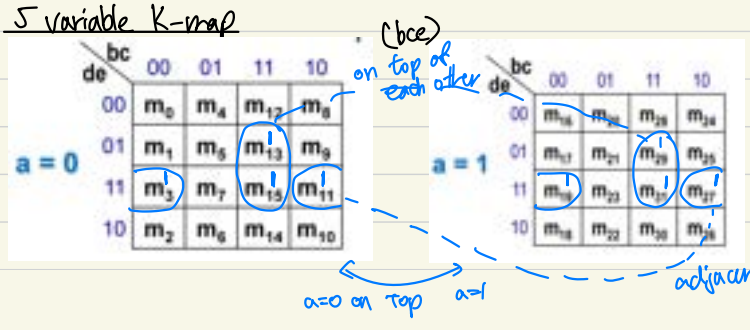
→ Usually expressed as:
$f(a,b,c)=\Sigma m(1,2,5)+d(4,6)$



010 110
101 001

$f=bc'+b'c$

5 variable K-map



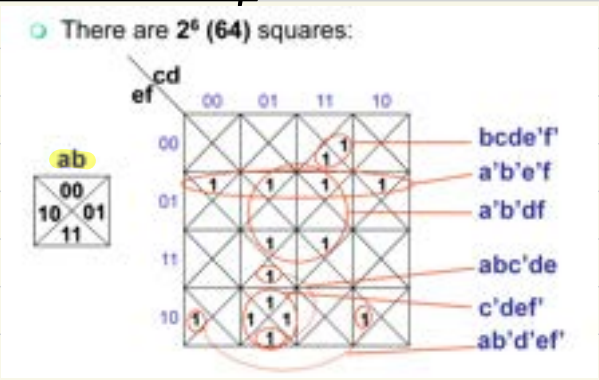
(bce) on top of each other

a=0 on top a=1

adjacent (c'de)

$f=(a,b,c,d,e)$
$=\Sigma m(3,11,13,15,19,27,29,31)$
$=bce+c'de$

6 variable K-map



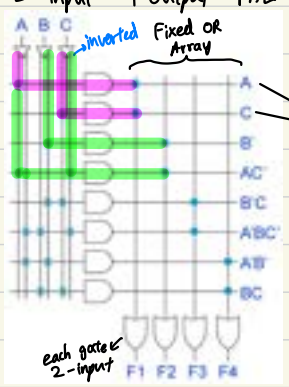
Programmable Logic Devices (PLDs)

↳ Sum of Products (SOP) of logic functions can be implemented

1. Programmable Array Logic (PAL)

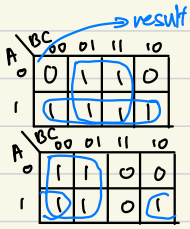
- programmable AND array & a fixed OR array
- AND plane has $2n$ programmable connection where n is the number of inputs
- no. of product terms for each OR gate output is fixed during manufacturing process & cannot be changed by user

E.g. 3-input 4-output PAL



$F_1 = \sum m(1,3,4,5,6,7)$

$= A + C$



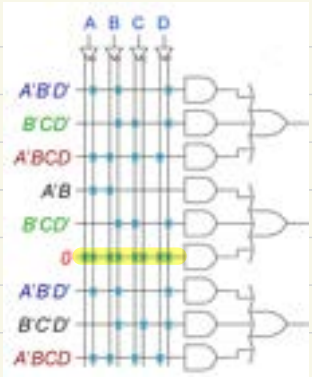
$F_2 = \sum m(0,1,4,5,6)$

$= B' + AC'$

$F_3 = \sum m(1,2,5) = B'C + A'BC'$

$F_4 = \sum m(0,1,3,7) = A'B' + BC$

E.g.



$F_1 = A'B'D' + B'C'D' + A'BCD$

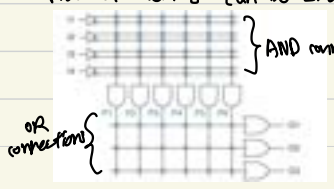
For PAL → need write both $2B'C'D'$

$F_2 = A'B + B'C'D' + 0$

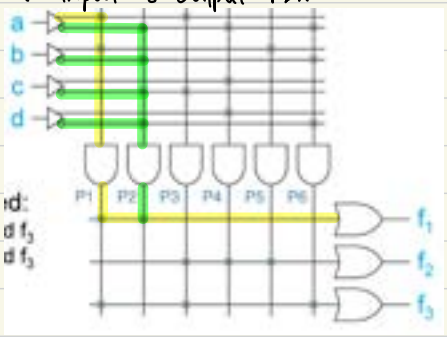
$F_3 = A'B'D' + B'C'D' + A'BCD$

2. Programmable Logic Array (PLA)

- Both AND array & OR array are programmable
- Number of product terms for each OR gate output is programmable by the user
- Product terms can be shared by PLA outputs, but not by PAL outputs due to their fixed OR array



E.g. 4-input 3-Output PLA



$f_1 = P1 + P2 = ab + \bar{a}\bar{b}\bar{c}\bar{d}$

$f_2 = P3 + P4 + P5$

$= a\bar{c} + \bar{a}cd + b$

$f_3 = P1 + P3 + P6$

$= ab + a\bar{c} + \bar{a}\bar{b}\bar{d}$

→ P1 is shared by f_1 & f_3

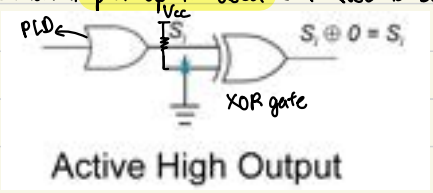
→ P3 is shared by f_2 & f_3

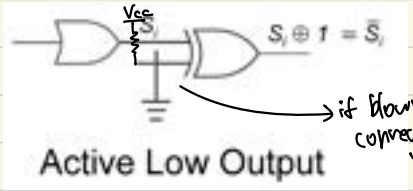
Programmable Output Polarity

↳ XOR gate added at output

↳ Connection to ground is programmable

↳ Internal pull-up is used (If fuse is blown → connection is to 1)





Advantages of Using PLDs

1. Reduce PCB size → PLD can replace many small scale indication (SSI)/medium scale indication (MSI) packages → reduce PCB size & cost.
2. Easier to assemble → takes less time & effort to assemble fewer components → reduce cost
3. Shorten design time → PLD implementation much faster than application static integration circuit (ASICs)/ full custom ICs.
4. Allow design changes → reprogrammable PLDs allow logic functions to be changed easily
5. Improve reliability → fewer component means greater reliability

Binary Decoder

↳ n input & 2^n output (n-to-2^n decoder)

E.g. 3-to-8 Decoder → 1 output corresponds to one input

Function Table →

Inputs			Outputs							
A	B	C	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0
0	1	1	0	0	0	0	1	0	0	0
1	0	0	0	0	0	1	0	0	0	0
1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0	0	0

$Y_0 = A'B'C' = \sum(m_0)$

$Y_1 = A'B'C = \sum(m_1)$

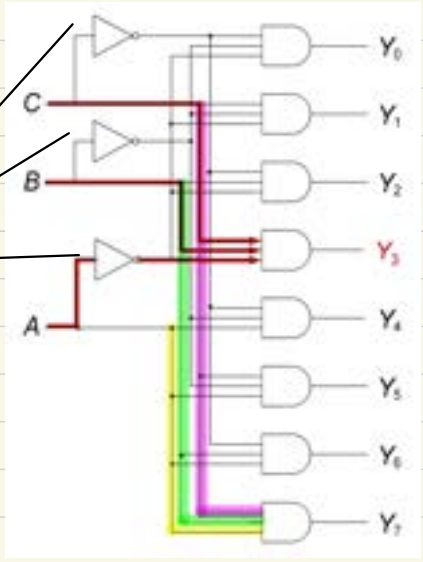
$Y_2 = A'BC' = \sum(m_2)$

$Y_3 = A'BC = \sum(m_4)$

⋮

} Logic Function

3 inverters



E.g. 3-to-4 Decoder → NOT a binary decoder

↳ have more than one output activated

Inputs			Outputs			
X	Y	Z	Q_3	Q_2	Q_1	Q_0
0	0	1	0	1	1	1
0	1	0	1	0	1	1
0	1	1	0	0	1	1
1	0	1	0	1	0	0
1	1	0	1	0	0	0

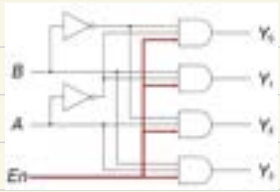
E.g. 2-to-4 Decoder with Output Enable ($\rightarrow 2^n$)

Function Table

Inputs			Outputs			
En	A	B	Y_3	Y_2	Y_1	Y_0
0	X	X	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

enable

IF En=0, A & B does not matter



$Y_0 = EnA'B'$ $Y_1 = EnA'B$ $Y_2 = EnAB'$ $Y_3 = EnAB$

MSI (Medium Scale Integration) Decoders

↳ e.g. 2-to-4, 3-to-8, 4-to-16 Decoders, BCD-to-decimal decoder, BCD-to-Seven Segment Decoder

E.g. 74LS138 3-to-8 Decoder [Low Power Schottky TTL]

3 enable pins

Inputs						Outputs							
G1	G2A	G2B	C (MSB)	B	A (LSB)	Y_7	Y_6	Y_5	Y_4	Y_3	Y_2	Y_1	Y_0
0	x	x	x	x	x	1	1	1	1	1	1	1	1
x	1	x	x	x	x	1	1	1	1	1	1	1	1
x	x	1	x	x	x	1	1	1	1	1	1	1	1
1	0	0	0	0	0	1	1	1	1	1	1	1	0
1	0	0	0	0	1	1	1	1	1	1	1	0	1
1	0	0	0	1	0	1	1	1	1	1	0	1	1
1	0	0	0	1	1	1	1	1	1	0	1	1	1
1	0	0	1	0	0	1	1	1	0	1	1	1	1
1	0	0	1	0	1	1	1	0	1	1	1	1	1
1	0	0	1	1	0	1	0	1	1	1	1	1	1
1	0	0	1	1	1	0	1	1	1	1	1	1	1

to enable decoder

all output are ones

only activated by 0 → active low output.

using 74LS138

$f = Y_1 \times Y_5 \times Y_7 = M_1 \times M_5 \times M_7$

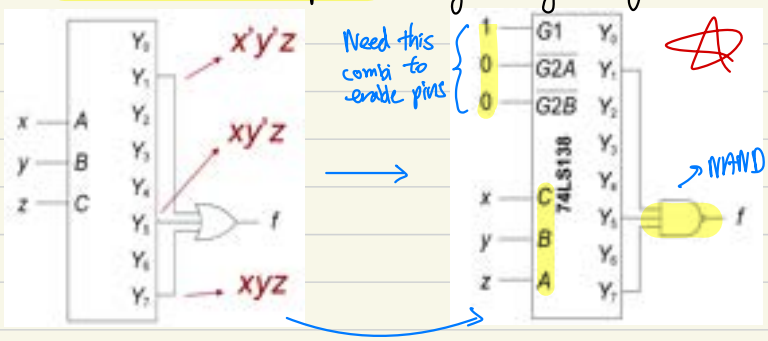
$= (A'+B+C)(A'+B+C')(A'+B'+C')$

$= (x+y+z')(x'+y+z')(x'+y'+z')$

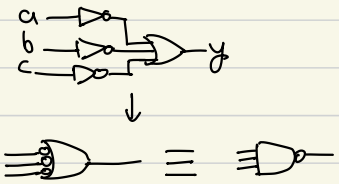
$\overline{(x+y+z)(x'+y'+z')(x'+y'+z')}$

$= x'y'z + xy'z + xyz$

Use a 3-to-8 decoder to implement: $f = x'y'z + xy'z + xyz = (m_1 + m_5 + m_7)$



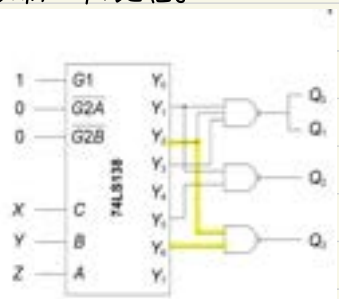
$y = \bar{a} + \bar{b} + \bar{c} = \overline{a \cdot b \cdot c}$



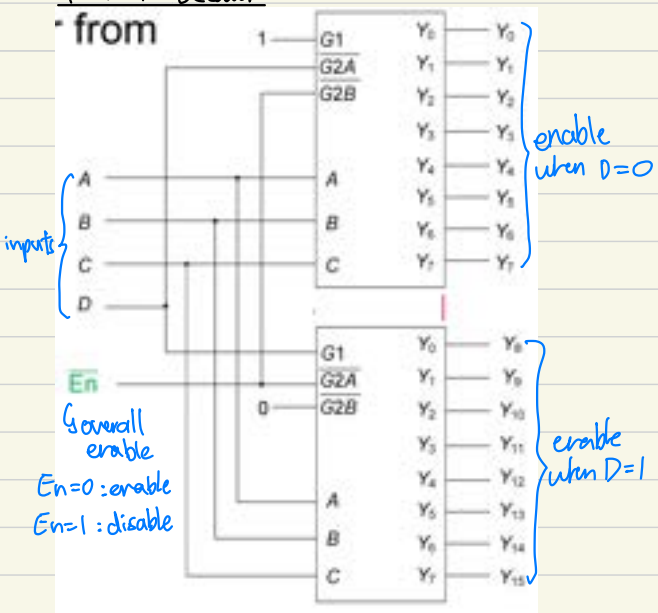
E.g. 3-to-4 Decoder with 74LS138

Inputs			Outputs			
X	Y	Z	Q_3	Q_2	Q_1	Q_0
0	0	1	0	1	1	1
0	1	0	1	0	1	1
0	1	1	0	0	1	1
1	0	1	0	1	0	0
1	1	0	1	0	0	0

- $Q_3 = m_2 + m_6$
- $Q_2 = m_1 + m_5$
- $Q_1 = Q_0 = m_1 + m_2 + m_3$



4-to-16 Decoder from



Binary Encoders (can be built from OR gates)

- perform **reverse operation of decoder**
- 2^n input & n outputs (2^n-to-n)
- Normal encoder have problem → use priority encoder (assign priority to inputs)

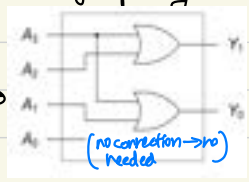
E.g. 4-to-2 Priority Encoder

A3 has the **highest priority**

A0 has the **lowest priority**

Inputs				Outputs	
A_3	A_2	A_1	A_0	Y_1	Y_0
1	x	x	x	1	1
0	1	x	x	1	0
0	0	1	x	0	1
0	0	0	1	0	0

→



Inputs				Outputs	
A_3	A_2	A_1	A_0	Y_1	Y_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

→ same result diff input (problem)

0 0 0 0 0 0

E.g. 74148 8-to-3 Priority Encoder

group select

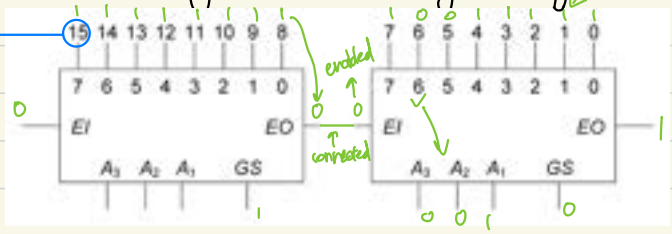
0 to enable, 1 to disable

Inputs									Outputs				
EI	7	6	5	4	3	2	1	0	A3	A2	A1	GS	EO
1	x	x	x	x	x	x	x	x	1	1	1	1	1
0	1	1	1	1	1	1	1	1	1	1	1	1	0
0	0	x	x	x	x	x	x	x	0	0	0	0	1
0	1	0	x	x	x	x	x	x	0	0	1	0	1
0	1	1	0	x	x	x	x	x	0	1	0	0	1
0	1	1	1	0	x	x	x	x	0	1	1	0	1
0	1	1	1	1	0	x	x	x	1	0	0	0	1
0	1	1	1	1	1	0	x	x	1	0	1	0	1
0	1	1	1	1	1	1	0	x	1	1	0	0	1
0	1	1	1	1	1	1	1	0	1	1	1	0	1

→ input activated & have corresponding code for it

→ 16-to-4 Priority Encoder → built by cascading 2 74148 8-to-3 priority encoders

highest priority



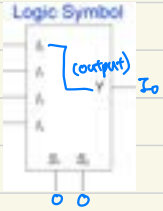
Multiplexers (MUX)/Data Selector

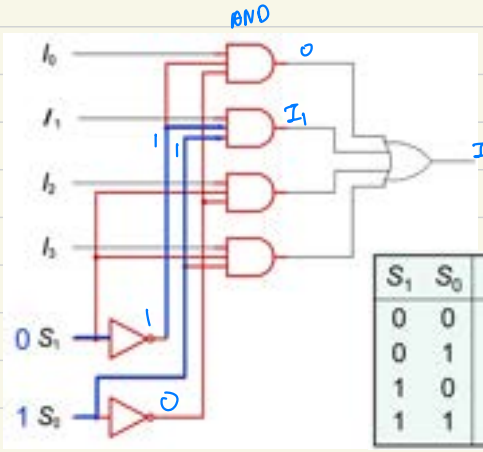
- Many input lines & 1 output line → select the input to go to the output
- If 2^n inputs → n selected signals are needed

E.g. 4-to-1 MUX

S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

If 0 0, means I_0 connect to output



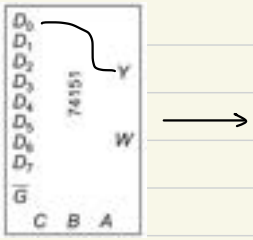


S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

→ $Y = S_1'S_0'I_0 + S_1'S_0I_1 + S_1S_0'I_2 + S_1S_0I_3$

MSI MUX (get these into online)

- 74151 : 8-to-1

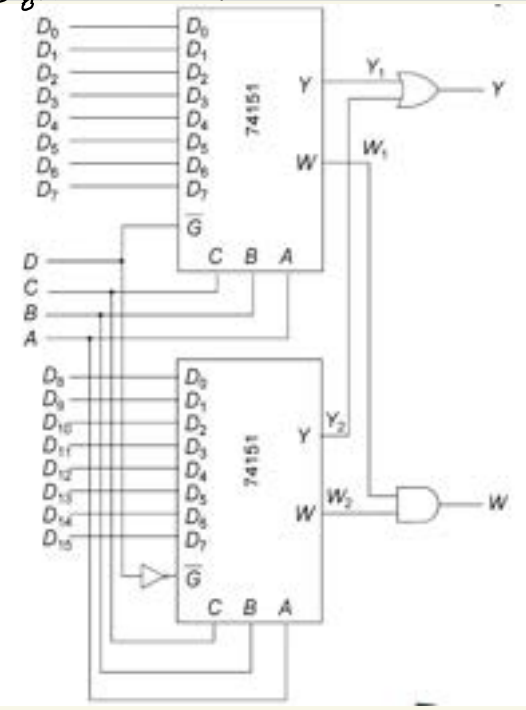


Inputs Select			Strobe	Outputs	
C	B	A	G	Y	W
x	x	x	1	0	1
0	0	0	0	D_0	$\overline{D_0}$
0	0	1	0	D_1	$\overline{D_1}$
0	1	0	0	D_2	$\overline{D_2}$
0	1	1	0	D_3	$\overline{D_3}$
1	0	0	0	D_4	$\overline{D_4}$
1	0	1	0	D_5	$\overline{D_5}$
1	1	0	0	D_6	$\overline{D_6}$
1	1	1	0	D_7	$\overline{D_7}$

enable use

$W = \overline{Y}$

E.g. 16-to-1 MUX



Built with two 74151 → $D_0 - D_7$

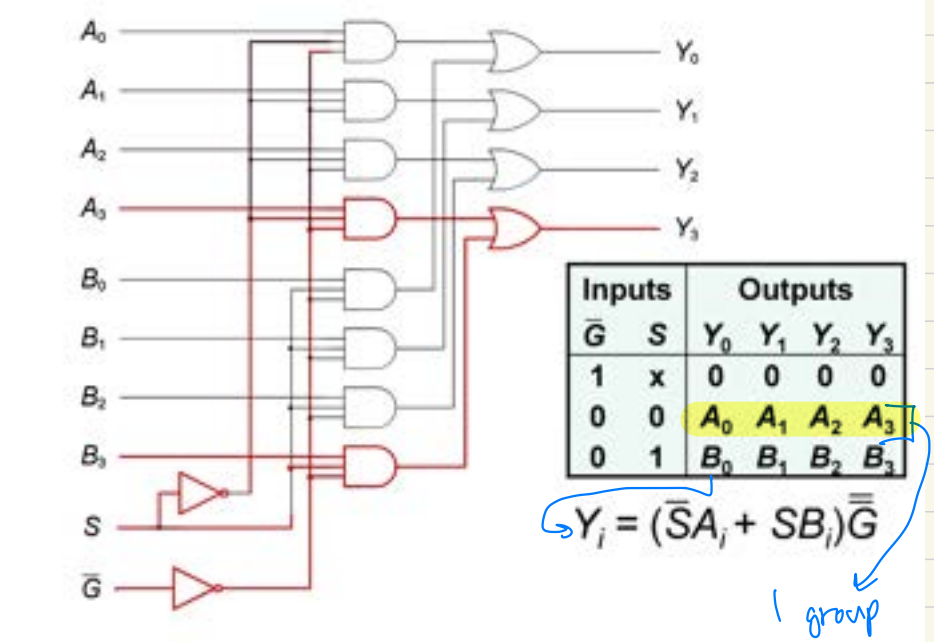
D = 0 : Top MUX enabled / Bottom MUX disabled

D = 1 : Bottom MUX enabled / Top MUX disabled → $D_8 - D_{15}$

$$W = \overline{Y}$$
$$= \overline{Y_1 + Y_2}$$
$$= \overline{\overline{W_1} + \overline{W_2}}$$
$$= W_1 \cdot W_2$$

Quadruple 2-to-1 MUX

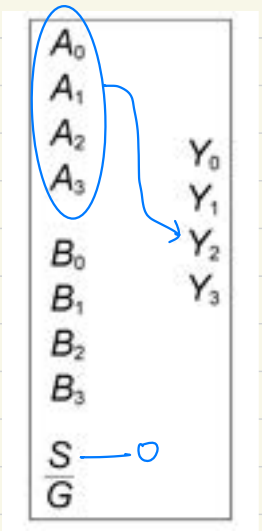
- 2-to-1 MUX with 4 bits for each input
- When 1 input is selected, the whole group of 4 bits goes to the output



Inputs		Outputs			
$\overline{G}$	S	Y_0	Y_1	Y_2	Y_3
1	x	0	0	0	0
0	0	A_0	A_1	A_2	A_3
0	1	B_0	B_1	B_2	B_3

$$Y_i = (\overline{S}A_i + SB_i)\overline{\overline{G}}$$

1 group



Implement Logic Function with MUX

E.g. Use 4-to-1 MUX to implement the same function:

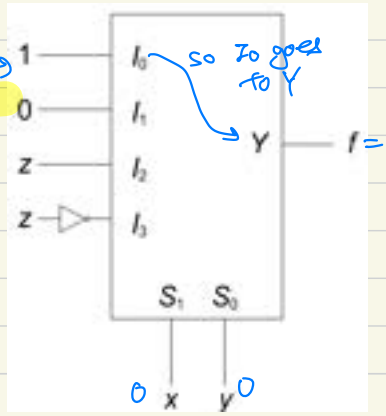
x	y	z	f
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

1 → according to table → will be output 1

0

z

$\bar{z}$



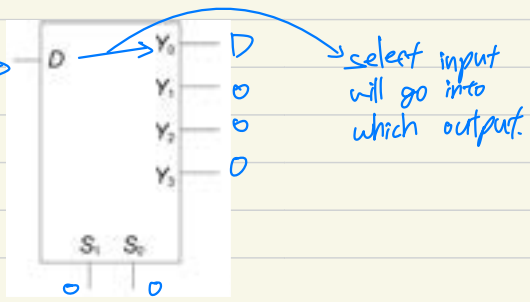
Demultiplexer (DMUX)

↳ Reverse the operation of MUX

↳ 1 data input line, n select signals, 2^n data output lines

E.g. 1-to-4 DMUX

S_1	S_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	D
0	1	0	0	D	0
1	0	0	D	0	0
1	1	D	0	0	0



select input will go into which output.

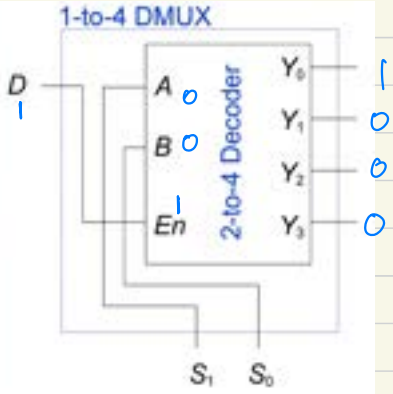
Use Decoders as DMUX

2-to-4 Decoder

En	A	B	Y_3	Y_2	Y_1	Y_0
0	X	X	0	0	0	0
1	0	0	0	0	0	1
1	0	1	0	0	1	0
1	1	0	0	1	0	0
1	1	1	1	0	0	0

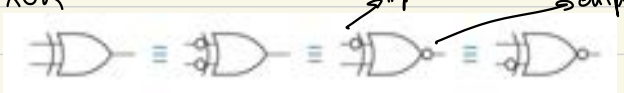
1-to-4 DMUX

S_1	S_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	D
0	1	0	0	D	0
1	0	0	D	0	0
1	1	D	0	0	0

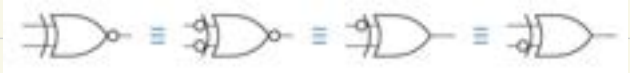


XOR & XNOR Equivalent Gates

↳ XOR



↳ XNOR



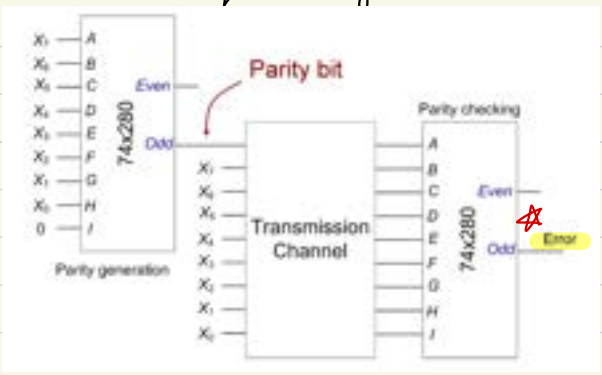
Parity Circuits

- Parity → Even / Odd quality of number of 1's in binary code
- Odd Parity → odd number of inputs are 1 → output = 1
- Even Parity → even number of inputs are 1 → output = 1

[1 0 0 1 0 1 1 1] → Odd Parity
[1 0 1 1 1] → Even Parity
→ can be realised by N-1 XOR gates

Parity Checking Applications

- Binary code = original code + 1 parity bit
- If original code has even no. of '1' → parity bit is '1', else, parity bit is '0'
  - Eg. 0110 → 2 bit of '1's → add 1 (even)
  - 10110 → odd parity → add 0 → 010110 (still odd)
- Most common technique for single error detection



Comparator

- compares binary numbers → equal = 1 → true. not equal = 0 → not true.

a	b	a=b	a>b	a<b
0	0	1	0	0
0	1	0	0	1
1	0	0	1	0
1	1	1	0	0

$a=b: \bar{a}\bar{b} + ab \;/\; \overline{\bar{a}\bar{b} + a\bar{b}}$ (NOT XOR)

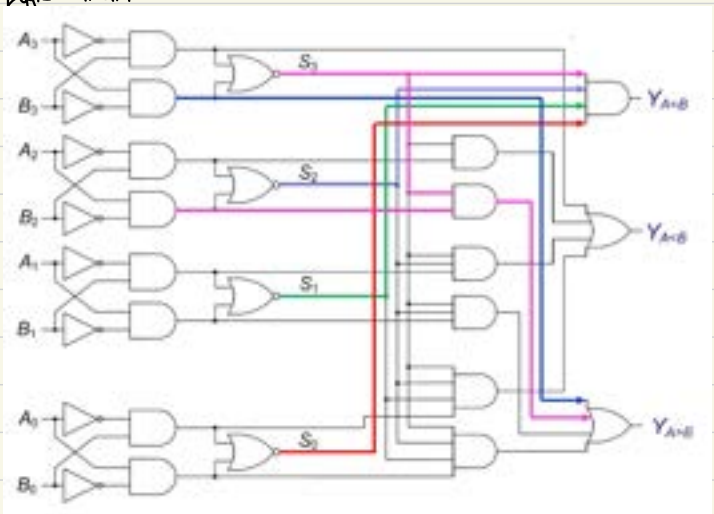
$a>b: a\bar{b}$

$a<b\ \bar{a}\,b$

Logic for A<B / B>A

- A_3B_3 is 01 → B > A the rest $A_0 \to A_2$, $B_0 \to B_2$ don't care (MSB)
- $A_3 = B_3$ → $A_2B_2 = 01$ → B > A ...

Logic Circuit



don't care

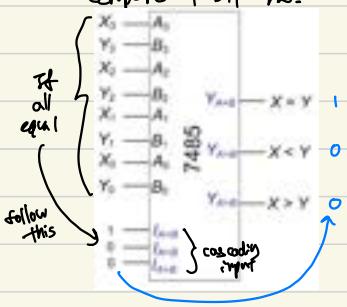
Input Data				Cascading Inputs			Outputs		
A_3B_3	A_2B_2	A_1B_1	A_0B_0	$I_{A>B}$	$I_{A<B}$	$I_{A=B}$	$Y_{A>B}$	$Y_{A<B}$	$Y_{A=B}$
$A_3>B_3$	X	X	X	X	X	X	H	L	L
$A_3<B_3$	X	X	X	X	X	X	L	H	L
$A_3=B_3$	$A_2>B_2$	X	X	X	X	X	H	L	L
$A_3=B_3$	$A_2<B_2$	X	X	X	X	X	L	H	L
$A_3=B_3$	$A_2=B_2$	$A_1>B_1$	X	X	X	X	H	L	L
$A_3=B_3$	$A_2=B_2$	$A_1<B_1$	X	X	X	X	L	H	L
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0>B_0$	X	X	X	H	L	L
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0<B_0$	X	X	X	L	H	L
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0=B_0$	H	L	L	H	L	L
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0=B_0$	L	H	L	L	H	L
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0=B_0$	X	X	H	L	L	H
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0=B_0$	L	L	L	H	H	L
$A_3=B_3$	$A_2=B_2$	$A_1=B_1$	$A_0=B_0$	H	H	L	L	L	L

follow this

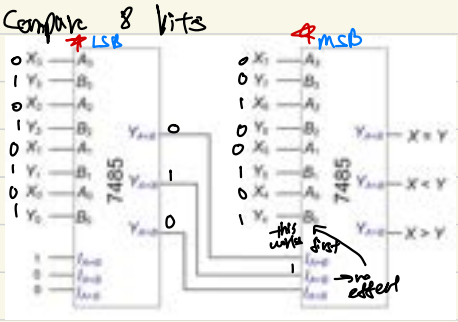
input all equal

not logical

Compare 4 bit no.



Compare 8 bits

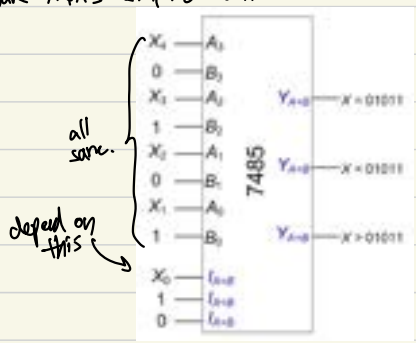


MSB LSB

X = 0100 0000

Y = 0011 1111

Compare $X_4X_3X_2X_1X_0$ with 01011



→ If $X_0 = 1$, $X = 01011$

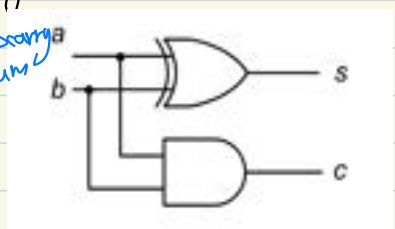
If $X_0 = 0$, $X = 01010 < 01011$

Half adder

↳ Addition of 2 single bits

a	b	s	c
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

carry
sum



$s = a \oplus b$

$c = a \cdot b$

1 ← carry

1

+ 1

0 → sum

Full adder

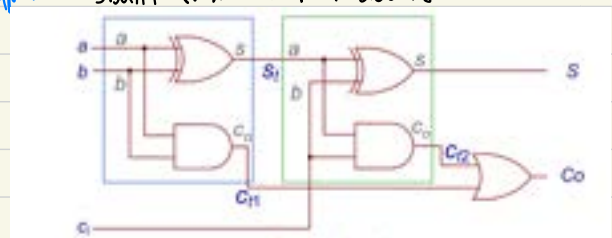
carry input

carry output

a	b	c_i	s	c_o
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

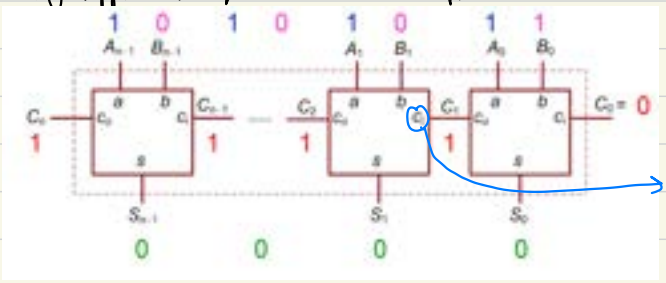
$s = a \oplus b \oplus c_i$

→ built from 2 half adders

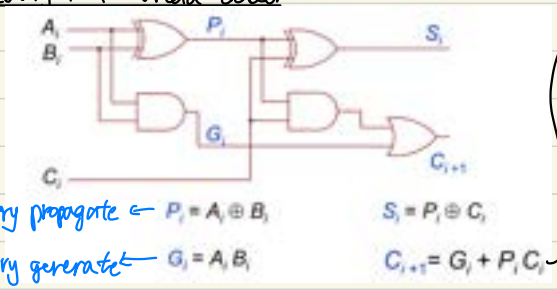


n-bit Ripple Adder

- A_{n-1} & B_{n-1} are MSB & A_0 & B_0 are LSB
- Carry ripples through chain → but simple & slow → carries have to propagate through chain



Carry-look-ahead adder



carry propagate ← $P_i = A_i \oplus B_i$ $\quad S_i = P_i \oplus C_i$

carry generate ← $G_i = A_i B_i$ $\quad C_{i+1} = G_i + P_i C_i$

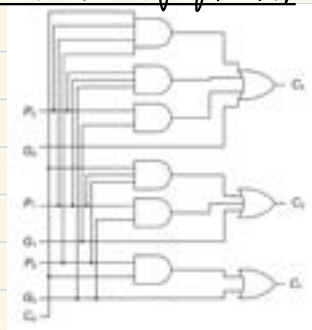
$C_1 = G_0 + P_0 C_0$

$C_2 = G_1 + P_1 (C_1)$

$= G_1 + P_1 (G_0 + P_0 C_0)$

↳ carry no longer depend on its previous state

Look-ahead carry generator

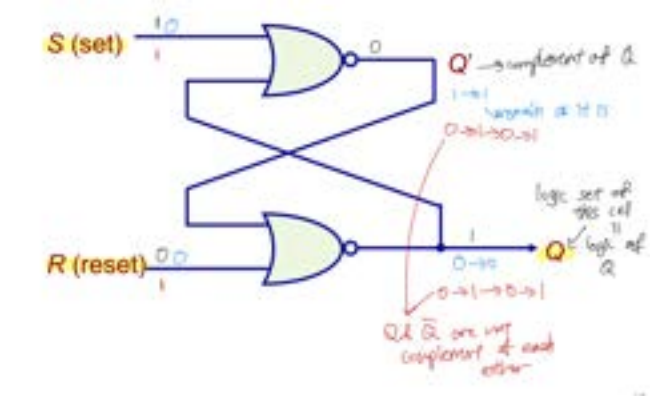


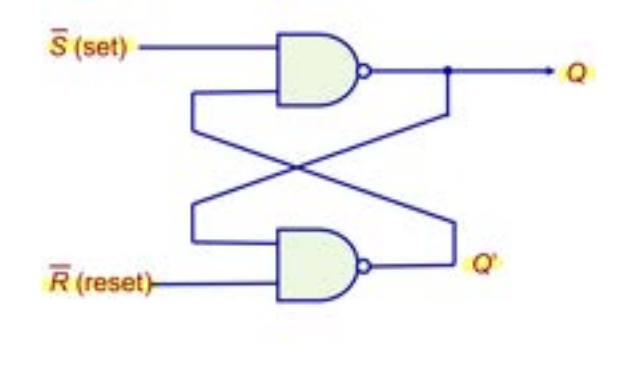
- Speed: 2 gate delay for all carry, independent of the number of bit in the adder
  $t_{C_i} = 2t_{gate}$
- Cost: more gates

Digital Circuits

1. Combinational Circuits
   - ↳ output depend only on present value & input & not past value
   - ↳ Random Logic (SOP/POS), Array Logic (PLA/PAL), Adders
2. Sequential Circuits
   - ↳ output depend on both present & past input values

NOR





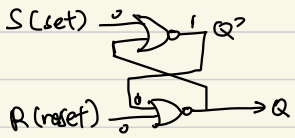
- Combinational Circuits → output depend only on present value of inputs
- Sequential Circuits → output depend on both present & past input values → have memory element
  a) Latch → memory element where output only be initiated by input
  b) Flip-flop → has control signal (clock) → any output only initiated by clock
- CLK

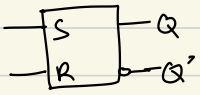
CLK-L (falling-edge)
↳ duty cycle = $\frac{t_L}{t_{per}}$

Basic Binary Cell

→ SET condition → stores a '1'
→ RESET condition → stores a '0'
→ REST condition unchanged

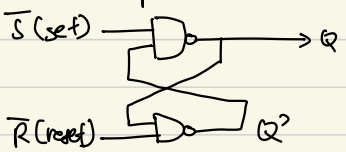
1. Cross-Coupled NOR Gate Cell

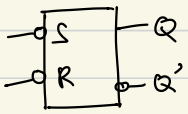




S(set)	R(reset)	Q	Q'	
1	0	1	0	set
0	0	1	0	rest
0	1	0	1	reset
0	0	0	1	rest
1	1	0	0	not allows.

2. Cross-Coupled NAND Gate Cell





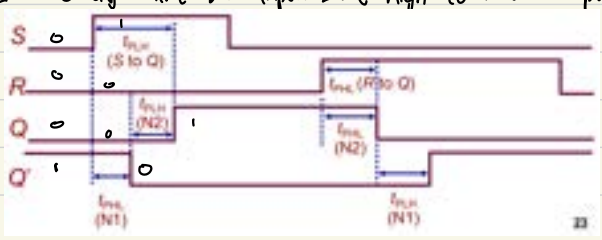
input ✓

S(set)	R(reset)	Q	Q'	
0	1	1	0	set
1	1	1	0	rest
1	0	0	1	reset
1	1	0	1	rest
0	0	1	1	not allows.

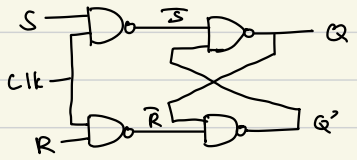
Delay Parameters

1. t_{PLH} → delay time b/w input Δ& low-to-high output transition
2. t_{PHL} → delay time b/w input Δ& high to low output transition.

NOR circuit.



SR Flip-flop (Clocked flip-flop)



↳ S, R only triggered when CLK = 1
CLK = 0 → S R don't care.

S	R	Q_t	Q_{t+1}	
0	0	0	0	Rest Mode
0	0	1	1	
0	1	0	0	Reset Mode
0	1	1	0	
1	0	0	1	Set Mode
1	0	1	1	
1	1	0	X	Not Allowed
1	1	1	X	

next state (when CLK triggers it)

each are either 0/1

same

reset

don't care condition

$Q_t \to Q_{t+1}$	S	R	S	R
0 → 0	0	X	0	0
			0	1
0 → 1	1	0	1	0
1 → 0	0	1	0	1
1 → 1	X	0	0	0
			1	0

either reset/rest

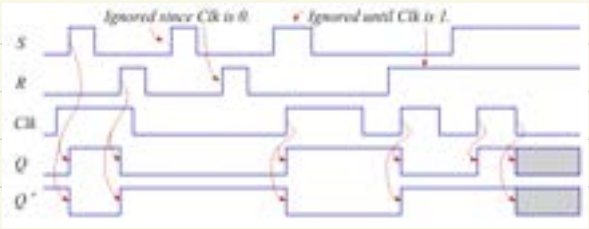
→ set

→ reset

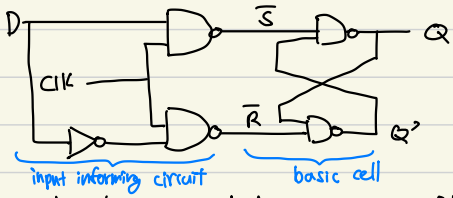
either set/rest

$Q_t \backslash SR$	00	01	11	10
0	0	0	X	1
1	1	0	X	1

$Q_{t+1} = R'Q_t + S$



D Flip-flop (Delay-latch)



↳ CLK pulse reaches high → Q output follows D input

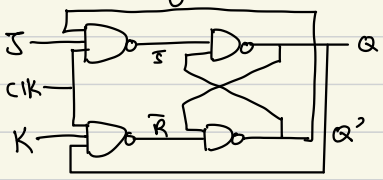
$Q_t \to Q_{t+1}$	D
0 → 0	0
0 → 1	1
1 → 0	0
1 → 1	1

D	Q_t	Q_{t+1}	
0	0	0	Reset condition
0	1	0	
1	0	1	Set Condition
1	1	1	

$Q_{t+1} = D$

JK Flip-flop

↳ functionally identical to SR flip-flop (J → S, K → R)



$Q_t \to Q_{t+1}$	J	K
0 → 0	0	X
0 → 1	1	X
1 → 0	X	1
1 → 1	X	0

J	K	Q_t	Q_{t+1}	
0	0	0	0	Rest
0	0	1	1	Rest
0	1	0	0	Reset
0	1	1	0	Reset
1	0	0	1	Set
1	0	1	1	Set
1	1	0	1	Toggle
1	1	1	0	Toggle

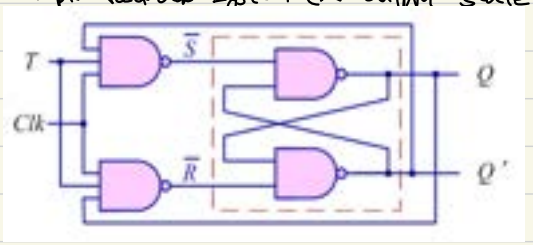
Toggle

$Q_t \backslash JK$	00	01	11	10
0	0	0	1	1
1	1	0	0	1

$JQ_t' + K'Q_t$

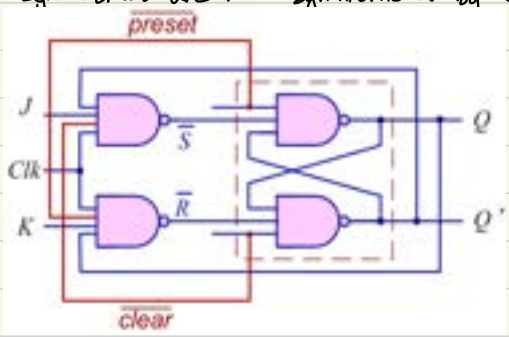
T Flip-flop (Toggle)

↳ multiple feedback system (Δ output state when T input is asserted



Asynchronous Preset & Clear

↳ Asynchronous are NOT synchronised by Clk signal & affect output immediately when asserted



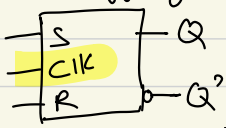
Normally either one = 0 & other 1

→ Both = 0 → Q & Q' = 1 → does not make sense → can use a reset dominant flip-flop

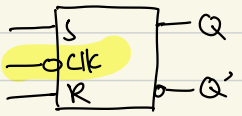
→ Both = 1 → does not matter → clk & other stuff matters

Triggering Schemes of FFs

1. Level Triggering (either logic 1/0)



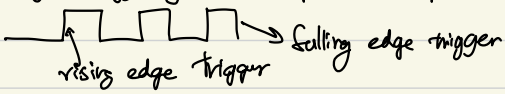
high-level triggered

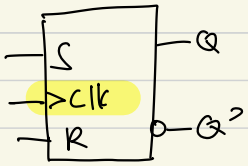


low level triggered

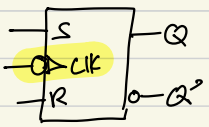
2. Pulse Triggering → normally with master-slave system → give rise to problem "ones catching" & "zeros catching" → falsely triggered due to noise input

3. Edge-Triggering → output of flip-flop only Δ with clock edge



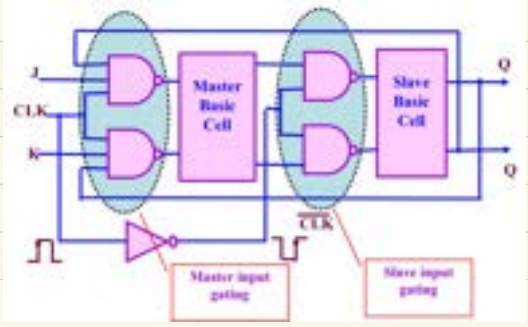


Rising edge triggered



Falling edge triggered

Master/Slave JK Flip-flop

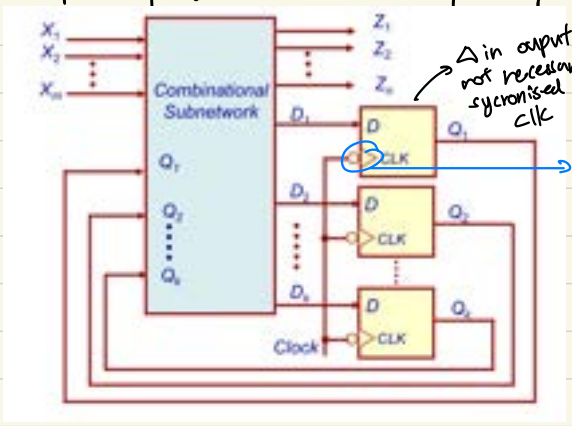


1. On rising-edge of clock pulse: Master active, slave inactive
   - ↳ Master basic cell loaded in accordance to the input condition of 1st SET/RESET decoder ($J, K, Q, \bar{Q}$)
   - ↳ New state blocked by NAND gates
2. Falling-edge of clock pulse: Master inactive, slave active.
   - ↳ Data stored in Master cell is passed through the slave cell & output Q & $\bar{Q}$ are updated.
   - ↳ New state is fed back to the input of the 1st SET/RESET decoder

☆ → Master/slave changes state on falling-edge of clock

Mealy Network

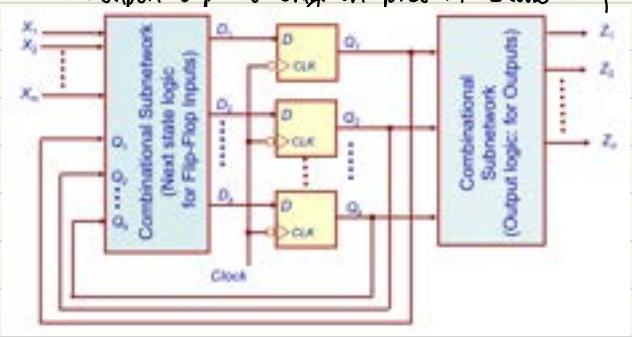
↳ output depend on both external input & present state



Moore Network

output ↑

↳ Output depend only on present state



↳ when Δ input → only take effect when clk triggers ↑

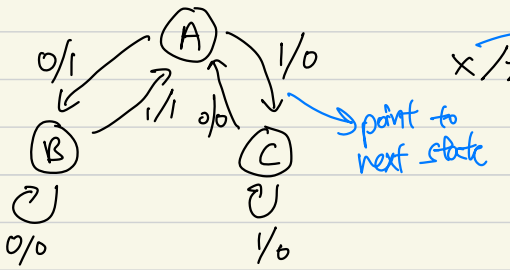
Mealy Model state diagram

S	Input (X)	
	0	1
A	B, 1	C, 0
B	B, 0	A, 1
C	A, 0	C, 0
	S*, Z	

present state → A; next state → B, 1 / C, 0; output

OR

S	S*		Output (Z)	
	X=0	X=1	X=0	X=1
A	B	C	1	0
B	B	A	0	1
C	A	C	0	0



X/Z → input to next state / output from

→ point to next state

E.g. Determine output response to sequential circuit of input X = 0110 1011 00, assuming initial state is A.

Clk Pulse:	0	1	2	3	4	5	6	7	8	9	10
X:	0	1	1	0	1	0	1	1	0	0	
Present state:	A	B	A	C	A	C	A	C	C	A	B
Next state:	B	A	C	A	C	A	C	C	A	B	
output:	1	1	0	0	0	0	0	0	0	1	

∴ Y = 1100 0000 01 & final state is B

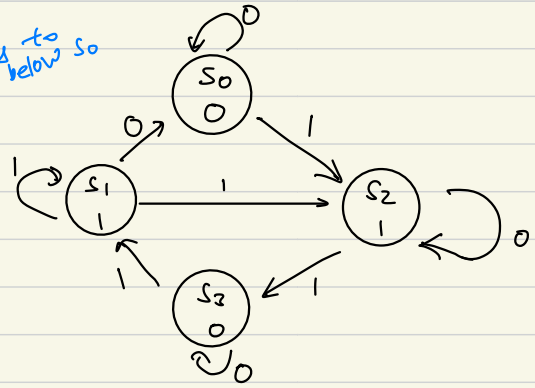
→ output Z can change any time either due to input/state changes

→ output of Mealy model circuit should be sampled only when the circuit has stabilised after input change, else false output arises → network assumed a new state but old input associated with previous state still present

Moore Model State Diagram & Table

S	Input (X) 0	Input (X) 1	Output (Z)
S_0	S_0	S_2	0
S_1	S_0	S_2	1
S_2	S_2	S_3	1
S_3	S_3	S_1	0
	S^*		

related to state below so

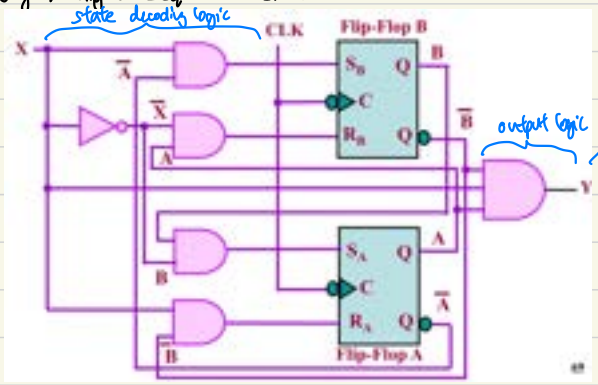


→ output change only when FFs Δ state (triggering edge of clock) → output is displaced in time with respect to the input sequence

Analysis of Synchronous Sequence Circuits

1. Identify functional blocks of system
2. Write Boolean expression for each outputs of NEXT-STATE Decoder (input of flip-flops) & output of the circuit
3. Plot K-maps (state map) from Boolean expressions
4. Construct Present & next state table using K-map & characteristic table of flip-flop
5. Use present state, input, output & next state to develop a state diagram
6. Describe behaviour based on state diagram

Eg. Analyse sequencial circuit



does it depend on input directly?
(if yes → connected to X → mealy circuit

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

$S_A = \overline{X} B$

(2)

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

$R_A = X \overline{B}$

(1) → get from diagram above & fill in k table.

	00	01	11	10
0	0	0	0	0
1	1	1	0	0

$S_B = X \overline{A}$

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

$R_B = \overline{X} A$

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

$Y = (X A \overline{B})$

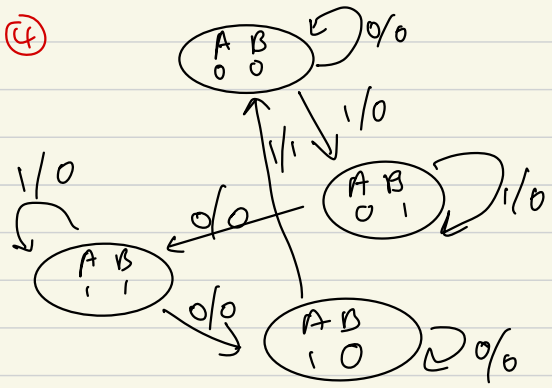
③

depends on SR FF

Present state		Input	Next state				output	True next state	
A	B	X	S_A	R_A	S_B	R_B	Y	A(t+1)	B(t+1)
0	0	0	0	0	0	0	0	0	0
0	0	1	0	1	1	0	0	0 (reset)	1 (set)
0	1	0	1	0	0	0	0	1 (set)	1 (no ch)
0	1	1	0	0	1	0	0	0	1
1	0	0	0	0	0	0	0	1	0
1	0	1	0	1	0	1	1	0	0
1	1	0	1	0	0	0	0	1	0
1	1	1	0	0	0	0	0	1	1

(present state) → (next state)

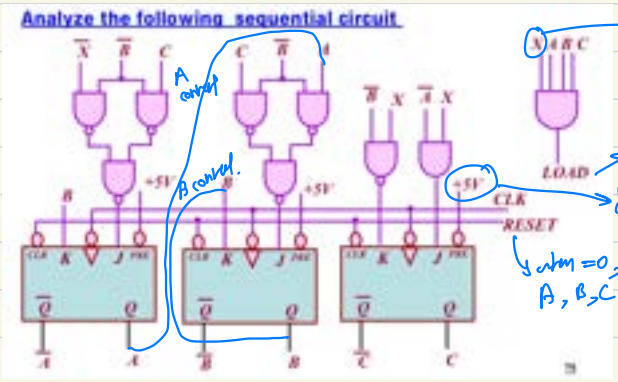
④



X/Y → input/output.

⑤ ↳ Diagram shows machine generate a proper sequence from state 00 to 01, 11, 10 & back to 00.
↳ Only time generate output Y is in state 10 to 00 when input X is asserted high
↳ Two flip-flop 4 useful states

E.g. 2.



→ meely circuit. X → SSC → load

→ output

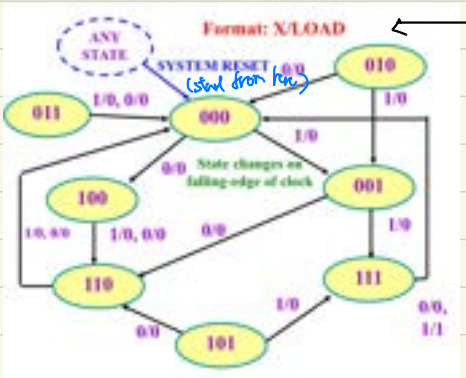
→ disabled → not in use
↳ connect to 5V

↳ when =0, A, B, C =

→ K-map →

	A	B	C	X	Load	A*	B*	C*
State	0	0	0	0	0	1	0	0
0	0	0	0	1	0	0	0	1
State	0	0	1	0	0	1	1	0
1	0	0	1	1	0	1	1	1
State	0	1	0	0	0	0	0	0
2	0	1	0	1	0	0	0	1
State	0	1	1	0	0	0	0	0
3	0	1	1	1	0	0	0	0
State	1	0	0	0	0	1	1	0
4	1	0	0	1	0	1	1	0
State	1	0	1	0	0	1	1	0
5	1	0	1	1	0	1	1	1
State	1	1	0	0	0	0	0	0
6	1	1	0	1	0	0	0	0
State	1	1	1	0	0	0	0	0
7	1	1	1	1	1	0	0	0

state diagram



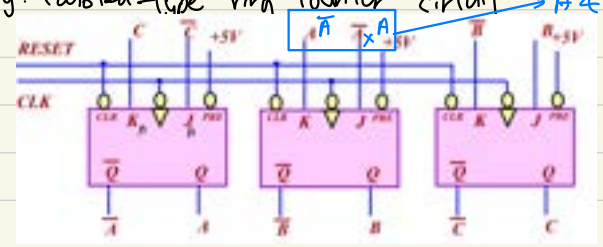
→ states 000, 001, 100, 110 & 111 → sequential states

→ states 010, 011 & 101 → unused states (never reached) → no arrow point to them

↳ Hold states will not stay for more than one clk cycle

→ D flip-flop, next true state follow input D-flip flop

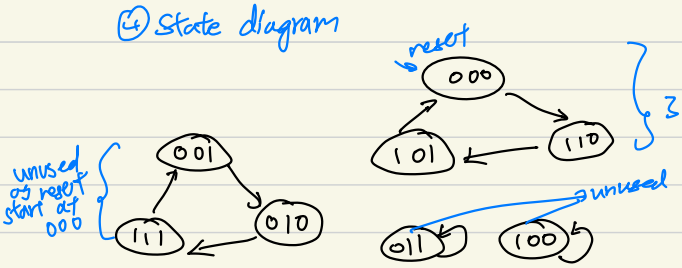
E.g. Twisted-tape ring counter circuit → $\bar{A}$ & $\bar{\bar{A}}$ wrong correction.



① Equation ② K-map ③ Present & next state tabulation

PRESENT STATE			INPUT	NEXT STATE CODE						OUTPUT	TRUE NEXT STATE		
A	B	C		J_A	K_A	J_B	K_B	J_C	K_C		A	B	C
0	0	0	-	1	0	1	0	0	1	-	1	1	0
0	0	1	-	0	1	1	0	0	1	-	0	1	0
0	1	0	-	1	0	1	0	1	0	-	1	1	1
0	1	1	-	0	1	1	0	1	0	-	0	1	1
1	0	0	-	1	0	0	1	0	1	-	1	0	0
1	0	1	-	0	1	0	1	0	1	-	0	0	0
1	1	0	-	1	0	0	1	1	0	-	1	0	1
1	1	1	-	0	1	0	1	1	0	-	0	0	1

④ State diagram

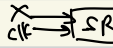


3 useful state. → 2 flip-flop 2^2 enough ↳ start wrong.

Design Synchronous Sequential Circuits

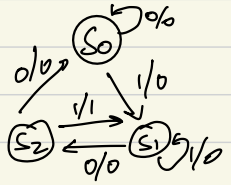
1. Study circuit specifications
2. Draw state diagram & determine no. of flip-flop & state variables
3. Develop present & next state table
4. Derive K-table & logic circuit

E.g. Design a circuit with one input X & one output Z that recognises input sequence 101 & overlapping sequences



X = 0011011001010100

Z = 0000010000010100



$2^{N_F-1} < N_S \leq 2^{N_F}$ (N_S: no. of states, N_F: no. of flip-flop)

$2^1 < 3 \leq 2^2$ → 2 flip-flop → 2 state variable Q_1 & Q_0

$S_0 : Q_1Q_0 = 00$ $S_1 : Q_1Q_0 = 01$ $S_2 : Q_1Q_0 = 10$

③ ① ④ ② decide what flip-flop you want ⑤

Present state		I/P	Next state		O/P	Next state decoder	
A	B	X	A_{t+1}	B_{t+1}	Z	D_A	D_B
0	0	0	0	0	0	0	0
0	0	1	0	1	0	0	1
0	1	0	1	0	0	1	0
0	1	1	0	1	0	0	1
1	0	0	0	0	0	0	0
1	0	1	0	1	1	0	1
1	1	0	X	X	X	X	X
1	1	1	X	X	X	X	X

follow

X \ AB	00	01	11	10
0	0	1	d	0
1	0	0	d	0

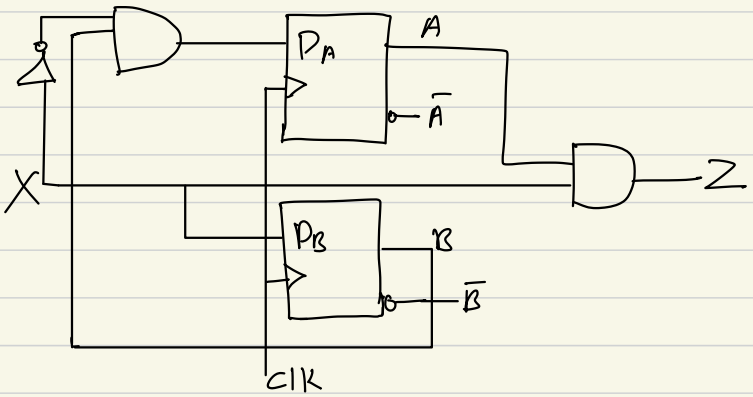
$D_A = X'B$

X \ AB	00	01	11	10
0	0	0	d	0
1	1	1	d	1

$D_B = X$

X \ AB	00	01	11	10
0	0	0	d	0
1	0	0	d	1

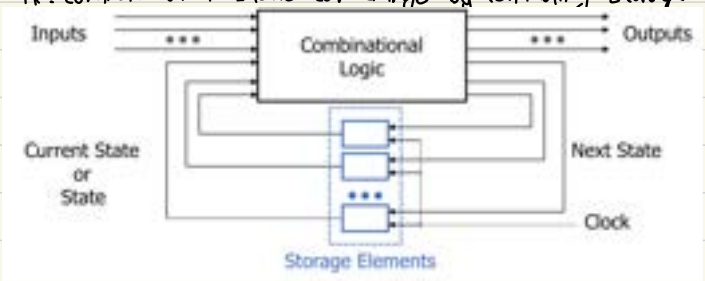
$Z = XA$



Lecture → Robot & Maze & Vending Machine

Finite-State Machine : FSM

- States : determined by possible values in sequential storage elements
- Transition : change of state
- Clk : controls when state can change by controlling storage elements

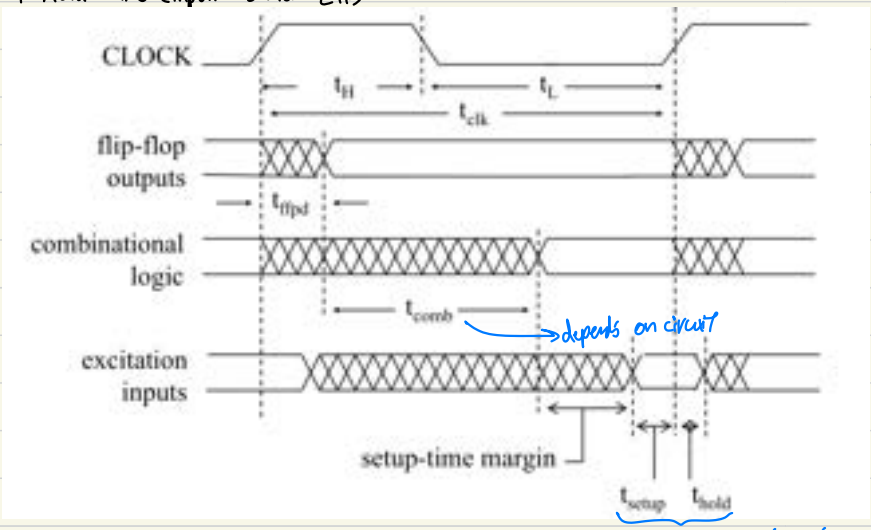


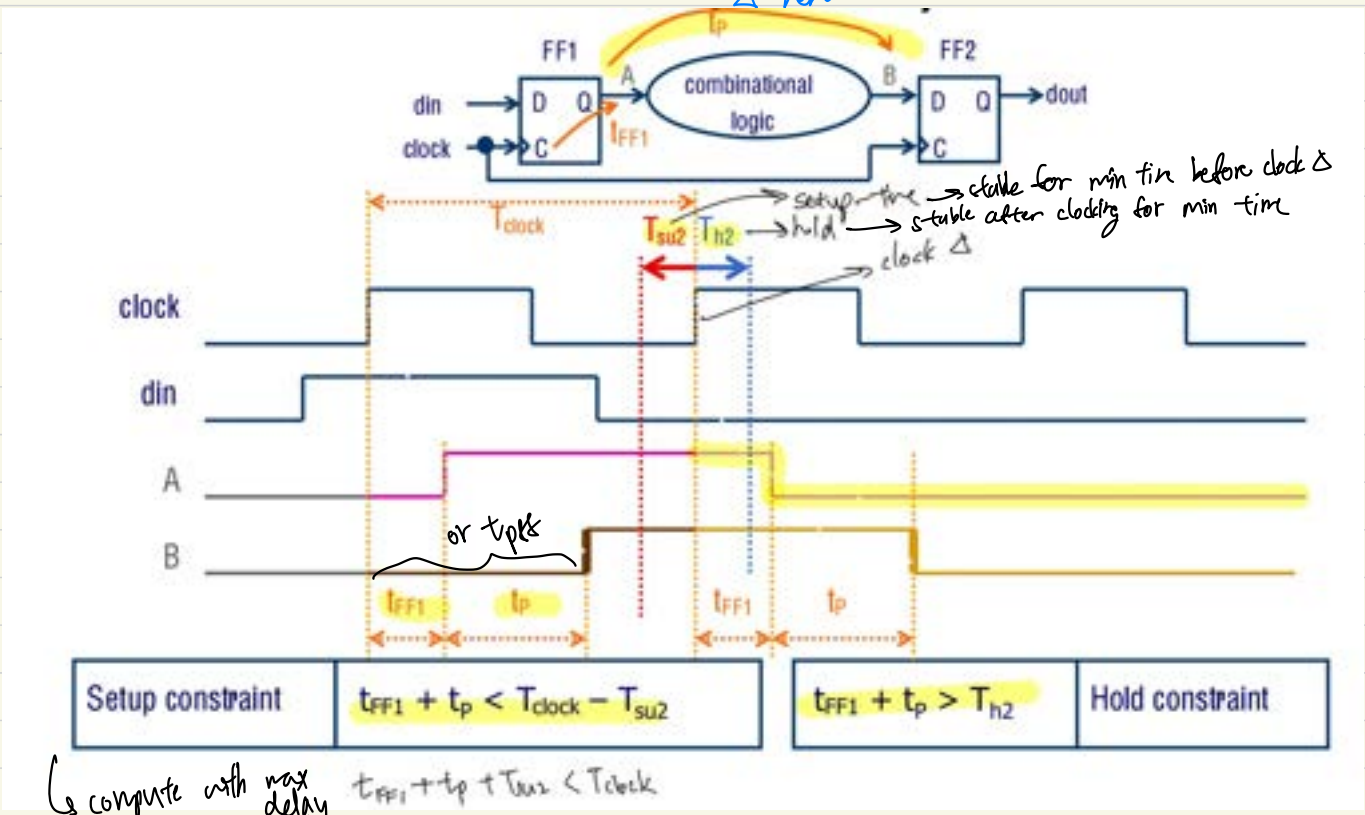
Look at: 1. Maximum propagation delay (clk to output)

2. Minimum propagation delay (clk to output)

3. Setup time (input before clk)

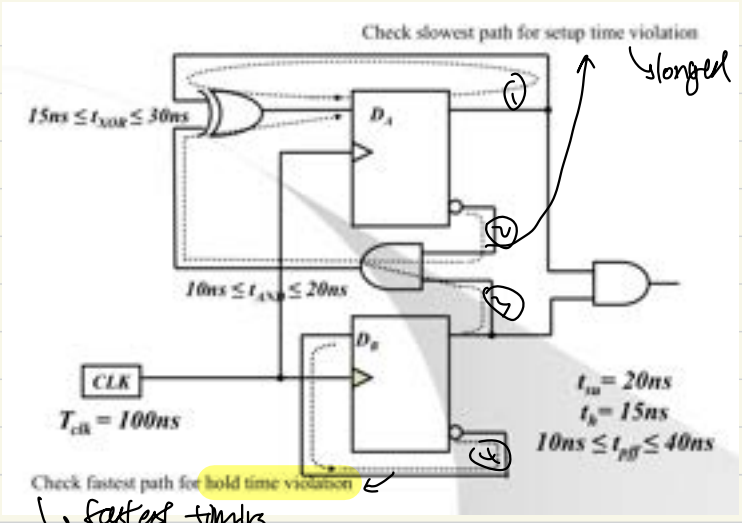
4. Hold time (input after clk)





↳ compute with max delay $t_{FF1} + t_p + T_{su2} < T_{clock}$

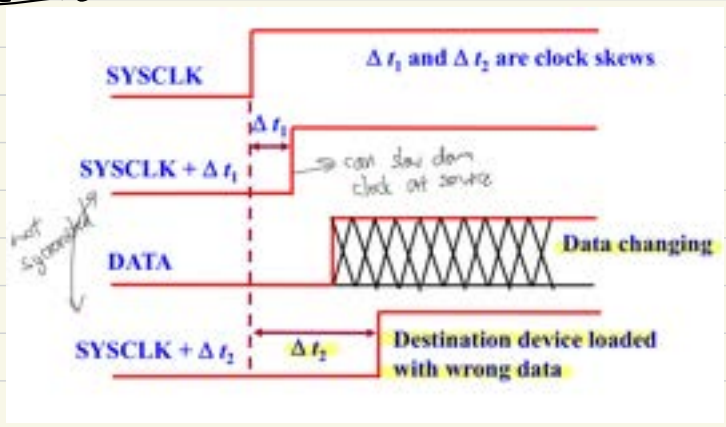
Delay through gate Δ supply voltage, temp & manufacturing process



longest timing

↳ fastest timing

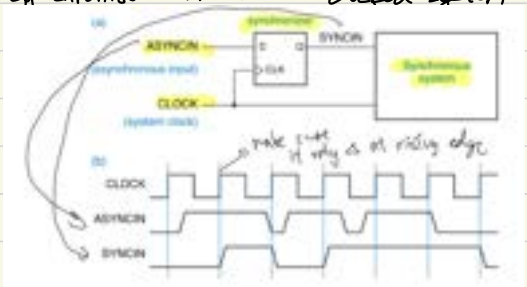
Clk skew



Metastability → cannot determin if it is 0/1

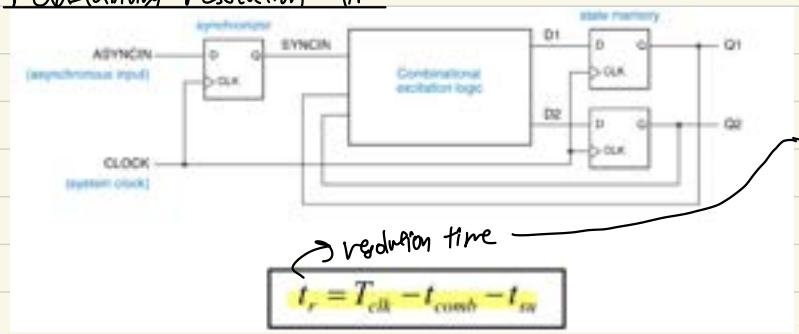
Asynchronous input

↳ synchronise them to the clocked system using synchroniser



→ Synchronising more than once → problem

Metastability Resolution Time



→ resolution time → maximum that output can remain metastable without causing synchroniser failure

OR

→ use faster flip-flop & lengthen clk period

$$\mathrm{MTBF} = \frac{e^{t_w/\tau}}{f_e \cdot f_{clk} \cdot t_a}$$

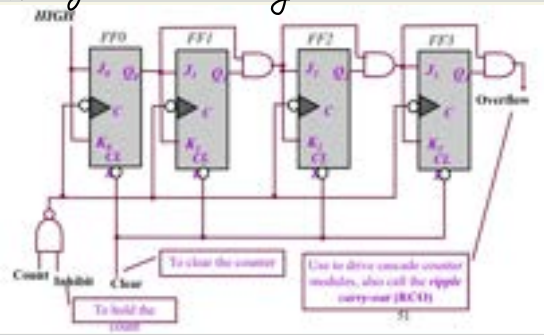
Mean Time Between synchronizer Failures

- t_w : waiting time or resolution time
- f_{clk} : frequency of the flip-flop clock
- f_e : asynchronous input changes per second applied to the flip flop
- τ is the regeneration time constant, i.e. one complete feedback signal of flip-flop
- t_a is the aperture time of flip-flop and in general equals to setup time + hold time

Counters

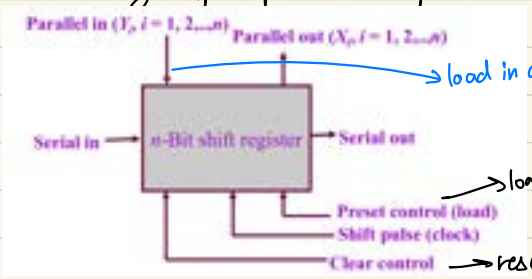
↳ a clocked sequential circuit that sequences through a fixed set of patterns

Adding control circuitry

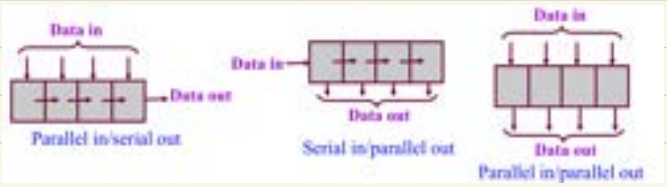


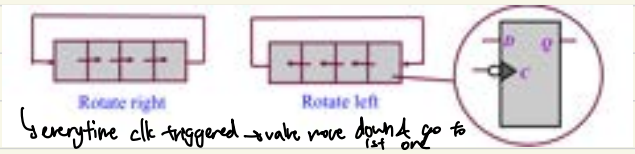
Universal shift register

↳ constructed by flip-flops that manipulates bit position of binary data by shifting data bits to the left/right



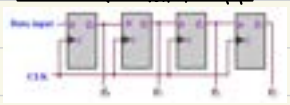
1. 

2. 

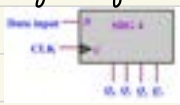
3. 

everytime clk triggered → value move down 1st one go to

Serial in/Parallel out

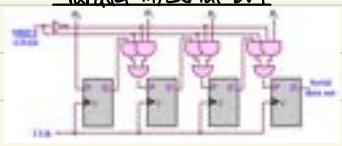


Logic Diagram

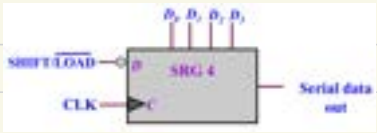


Logic Symbol

Parallel in/serial out



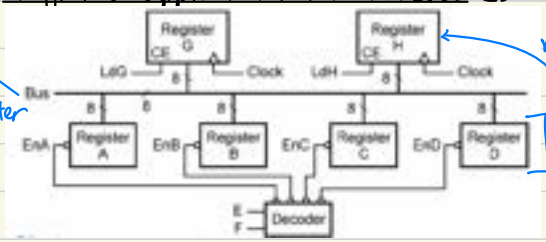
Logic Diagram



Logic Symbol

Registers application: Data Transfers (Data Highway)

Data can be placed on bus & read by designated register



move data from 4 block to 2 block

EF = 00 → A

EF = 01 → B

EF = 10 → C

EF = 11 → D

↳ C is enabled → go on bus → C$_3$ can be activated by clk & lock value of register C into flip-flop

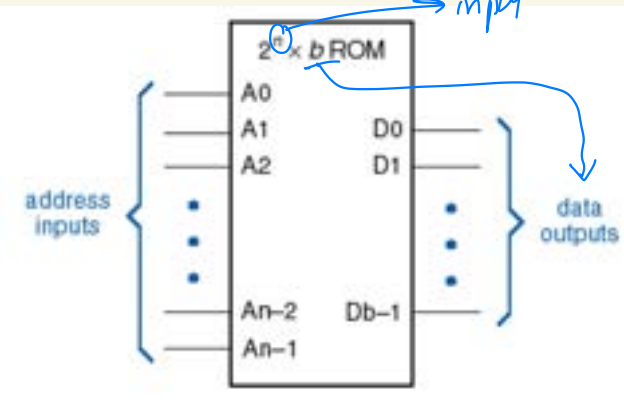
Read-Only Memories (ROM)

↳ program storage
  ↳ Boot ROM for personal computers
  ↳ Complete application storage for embedded systems

1 byte = 8 bits

$k = 2^{10} = 1024$

$M = 2^{20} = k^2 = k \times k$

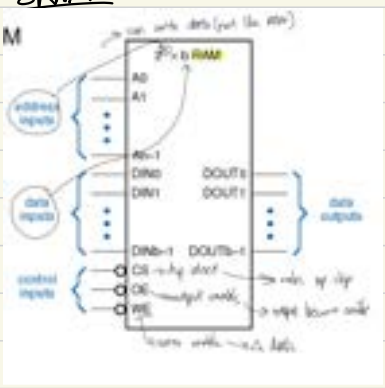


Read/Write Memories (RAM ⇒ random access memory)

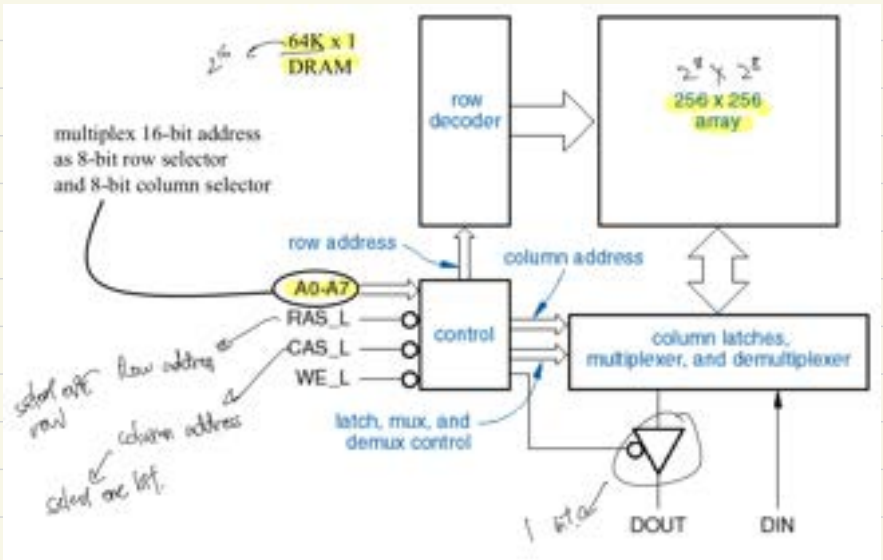
↳ Lose their memory when power removal.

- SRAM (static RAM) → behave like latches & flip-flop
- DRAM (dynamic memory) → memory only last a few milliseconds → must "refresh" location by reading/writing

SRAM



DRAM



8 byte → 2^3 byte → ③ data i/x
8×2^{23} bits → ② address i/x.

1 byte → 8 bits → 2^3 bits

1k bits → 2^{10} bits M bits → 2^{20} bits

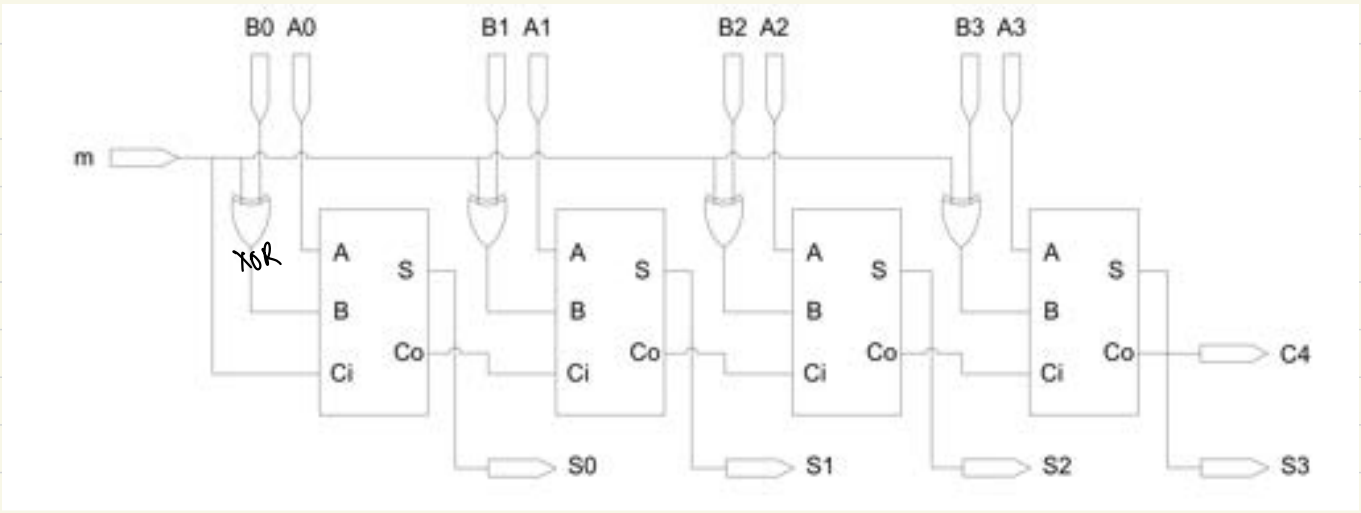
8 Mbyte = $2^{23} \times 8$ bits

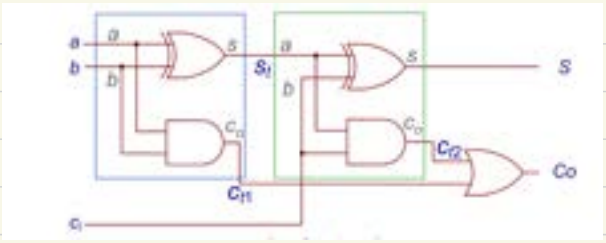
256 k bits → $2^8 \times 2^{10} = 2^{18}$

256M bits = $2^8 \times 2^{20} = 2^{28}$ bits 32 data bits = 2^5 data bits.

total address bit = $\frac{2^{28}}{2^5} = 2^{23}$

no of address pins needed = $\frac{23}{2} = 11.5 \Rightarrow 12$





→ toggle

T	Q	Q_{t+1}
0	0	0
0	1	1
1	0	1
1	1	0