Logic Design

In general base-r system has coefficients multiplied by powers of r: $(a_n a_{n-1} a_{n-2} \dots a_0 . a_{-1})_r$

$a_n \cdot r^n + a_{n-1} \cdot r^{n-1} + a_{n-2} \cdot r^{n-2} + \dots + a_0 \cdot r^0 + a_{-1} \cdot r^{-1} + \dots \, a_{-m} \cdot r^{-m}$

★Hexa Decimal: 0, 1, 2, 3, …, 9, A, B, C, D, E, F
10 11 12 13 14 15

★In computer:
Kilo ⟷ 2^{10}, Mega ⟷ 2^{20}, Giga ⟷ 2^{30}, tera ⟷ 2^{40}

★computer capacity → Bytes → 8 Bits

★Arithmetic operations: $0+0=0$, $0+1=1$, $1+0=1$, $1+1=10$

★Number-Base conversions:

Decimal → r system

Integer: divide by r; fraction: multiply by r

Octal → 8 → 2^3, Hexa-Decimal → 16 → 2^4

3 Binary Digits, 4 Binary Digits

★complement of Numbers: r's complement, (r-1)'s complement

r's complement: $[(r^n-1)-N]+1$

(r-1)'s complement: $(r^n-1)-N$, ★if Binary

★in Binary
سيب لحد أول 1
وبعدها أقلب

★Subtraction with complements: unsigned M − N in base r

① take r's complement to N, ② Add to M,

③ if M ≥ N → Nothing to do And carry is discarded

④ if M < N → Take r's complement As No carry Discarded

★Signed Binary Numbers:

Decimal	Signed-1's complement	Signed-1's comp	Signed Magnitude
+4	0100	0100	0111
+3	0011	0011	0011
+2	0010	0010	0010
+1	0001	0001	0001
+0	0000	0000	0000
−0	—	1111	1000
−1		1110	1001
−2		1101	1010
−3		1100	1011
−4		1011	1100

overflow:
(+Ve) + (+Ve) = −Ve
(−Ve) + (−Ve) = +Ve

Arithmetic Subtraction:
$(\pm A) - (+B) = (\pm A) + (-B)$
$(\pm A) - (-B) = (\pm A) + (+B)$

★n-code bit
→ 0 → $2^n - 1$ element

BCD: 0000 = 0, 0001 = 1, 0010 = 2, 0011 = 3, 0100 = 4, 0101 = 5, 0110 = 6, 0111 = 7, 1000 = 8, 1001 = 9

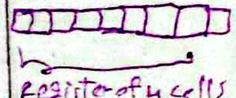
★BCD Addition:
→ Less than or equal 1001 (without carry)
→ greater than or equal 1010 XX
Add +6 (0110)
→ Have A carry? Add 6 (0110)

Decimal Digit	BCD 8421	2421	Excess-3	8, 4, -2, -1
0	0000	0000	0011	0000
1	0001	0001	0100	0111
2	0010	0010	0101	0110
3	0011	0011	0110	0101
4	0100	0100	0111	0100
5	0101	1000	1000	1011
6	0110	1001	1001	1010
7	0111	1010	1010	1001
8	1000	1011	1011	1000
9	1001	1100	1100	1111

★2421, excess-3 Are self-complementing

★Excess-3 unweighted code

★Register: A group of Binary cells.



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

★ASCII → 7-Bit, 94 Graphic characters, 34 Non-printing

★Error-Detecting code:

ASCII A = 1000001

Even-Parity: 01000001 — نخلي عدد ال1 زوجي

odd Parity: 11000001 — نخلي عدد ال1 فردي

AND

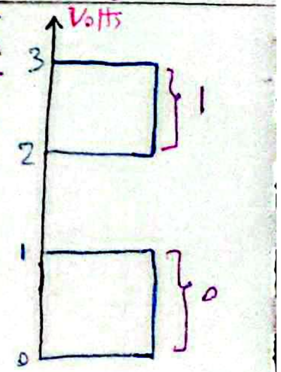
x	y	x·y
0	0	0
0	1	0
1	0	0
1	1	1

OR

x	y	x+y
0	0	0
0	1	1
1	0	1
1	1	1

Not

x	x'
0	1
1	0



☆ Most common Postulates formulate Various Algebric structure

① closure: A subset of a given set is closed under an operation on the larger set if Performing that operation on members of the subset always Produce a member of that subset.

② Associative Law: $(x*y)*z = x*(y*z)$ for all $x, y, z \in S$ نبدل ترتيب operators وليس Brackets

③ commutative Law: $x*y = y*z$ for all $x, y \in S$

④ Identity element: $e*x = x*e = x$ for every $x \in S$

⑤ Inverse: $x*y = e$

⑥ Distributive Law: $x*(y \cdot z) = (x*y) \cdot (x*z)$

☆ A field → Algebric structure → set of elements
→ two Binary operators with Properties 1→6

☆ Axiometic Definition of Boolean Algebra:

Boolean Algebra → Algebric structure defined by set of Elements B
→ with two binary operators +, •
→ satisfy Huntington Postulates

Huntington Postulates
- → ① structure closed under + operator, • operator
- → ② 0 → Identity for +, 1 → Identity for •
- → ③ $x+y = y+x$, $x \cdot y = y \cdot x$ commutative commutative
- → ④ $x \cdot (y+z) = (x \cdot y) + (x \cdot z)$, $x + (y \cdot z) = (x+y) \cdot (x+z)$ Distributive
- → ⑤ there is a complement for every $x \to x'$ → $x + x' = 1$, → $x \cdot x' = 0$
- → ⑥ At Least two elements $x, y \in B$, $x \neq y$

☆ Two-Valued Boolean Algebra

AND

x	y	x·y
0	0	0
0	1	0
1	0	0
1	1	1

OR

x	y	x+y
0	0	0
0	1	1
1	0	1
1	1	1

NOT

x	x'
0	1
1	0

Duality:
OR ⟷ AND
0 ⟷ 1
DeMorgan:
OR ⟷ AND
0 ⟷ 1
Variable ⟷ complement

Operator Precedence:
(1) Parentheses
(2) Not
(3) AND
(4) OR

☆ consensus Theorem: $\underbrace{xy + x'z}_{\text{complemented Variable}} + yz = xy + x'z$ → Redundancy Term

☆ Canonical AND standard Forms: n-Variable → 2^n - minterm, → 2^n - maxterm

☆ Minterm → AND → 0-Variable', → 1-Variable   ☆ Maxterm → OR → 0-Variable, → 1-Variable'

☆ $F = \underset{+}{\Sigma}(\text{minterms of one's}) = \underset{\text{AND}}{\Pi}(\text{Maxterms of zero})$, ☆ $\text{minterm}_j' = \text{Maxterm}_j$

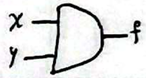
☆ standard Forms → Sum of Products (SOP)
→ Product of Sums (POS)

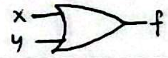
x	y	f0	f1	f2	f3	f4	f5	f6	f7	f8	f9	f10	f11	f12	f13	f14	f15
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

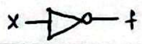
Boolean function	operator symbol	Name	comments
$F_0 = 0$		Null	Binary constant zero
$F_1 = xy$	$x \cdot y$	AND	x and y
$F_2 = xy'$	x/y	Inhibition	x, but Not y
$F_3 = x$		Transfer	x
$F_4 = x'y$	y/x	Inhibition	y, but Not x
$F_5 = y$		Transfer	y
$F_6 = xy' + x'y$	$x \oplus y$	Exclusive OR (XOR)	x or y but Not both
$F_7 = x + y$	$x + y$	OR	x or y
$F_8 = (x+y)'$	$x \downarrow y$	NOR	Not-OR
$F_9 = xy + x'y'$	$(x \oplus y)'$	Equivalence (XNOR)	x equal y
$F_{10} = y'$	y'	Complement	Not y
$F_{11} = x + y'$	$x \subset y$	Implication	If y, then x
$F_{12} = x'$	x'	Complement	Not x
$F_{13} = x' + y$	$x \supset y$	Implication	If x, then y
$F_{14} = (xy)'$	$x \uparrow y$	NAND	Not-AND
$F_{15} = 1$	Identity	Identity	Binary constant 1

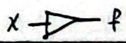
☆ Positive & Negative Logic

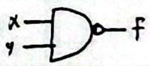
Logic Value	Signal Value	Logic Value	Signal Value
1	H	0	H
0	L	1	L

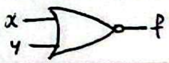

AND  $f = x \cdot y$

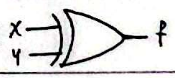
OR  $f = x + y$

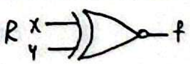
Inverter  $f = x'$

Buffer  $f = x$

NAND  $f = (xy)'$

NOR  $f = (x+y)'$

XOR  $f = xy' + x'y = x \oplus y$

XNOR  $f = xy + x'y' = (x \oplus y)'$

GATE-Level Minimization : Map Method

only one bit changes in value from one adjacent column to next

☆ Two-Variable K-Map :

	y=0	y=1
x=0	m_0 $x'y'$	m_1 $x'y$
x=1	m_2 xy'	m_3 xy

ال map ان تتجمع المربعات اللى جنب بعض بس مش Diagonal يعنى فوق بعض تحت بعض من اليمين للشمال

☆ Three-Variable K-Map :

x \ yz	00	01	11	10
0	m_0	m_1	m_3	m_2
1	m_4	m_5	m_7	m_6

→ Gray code

x \ yz	00	01	11	10
0	$x'y'z'$	$x'y'z$	$x'yz$	$x'yz'$
1	$xy'z'$	$xy'z$	xyz	xyz'

☆ Four-Variable K-Map : Any two Adjacent squares in the map differ by only one variable

m_0	m_1	m_3	m_2
m_4	m_5	m_7	m_6
m_{12}	m_{13}	m_{15}	m_{14}
m_8	m_9	m_{11}	m_{10}

wx \ yz	00	01	11	10
00	$w'x'y'z'$	$w'x'y'z$	$w'x'yz$	$w'x'yz'$
01	$w'xy'z'$	$w'xy'z$	$w'xyz$	$w'xyz'$
11	$wxy'z'$	$wxy'z$	$wxyz$	$wxyz'$
10	$wx'y'z'$	$wx'y'z$	$wx'yz$	$wx'yz'$

1 square → 4 Literals
2 squares → 3 Literals
4 squares → 2 Literals
8 // → 1 Literal
16 // → equal $\boxed{1}$
zero // → equal $\boxed{0}$

☆ **Prime Implicant :** Product term obtained by combining the Maximum Possible Number of Adjacent squares in the map — أكبر عدد مربعات جنب بعض

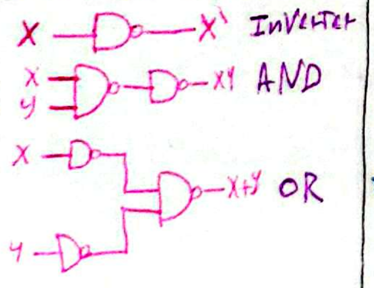
☆ **Essential Prime Implicant :** if it is the only Prime Implicant that covers the Minterm — لو بيغطى minterm معين مفيش Prime Implicants تانى بيغطيه

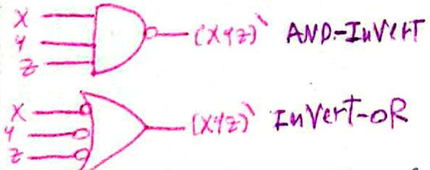
☆ لما بتجمع الوحايد بتنشأ معادلة فى صورة SOP ← Sum of Products ← ( )+( )+( )

☆ لما بتجمع الاصفار وتقلب المتغيرات بتنشأ معادلة فى صورة POS ← Product of Sums ← (+ +)(+ +)

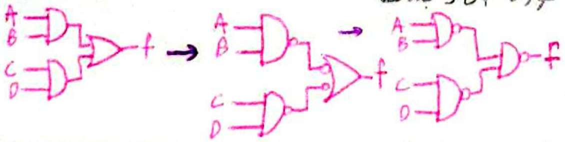
☆ **Don't-Care conditions :** مش كل الاحتمالات صفر وواحد وارد يكون فى احتمالات التطبيق الخاص بنا غير مهتم بها فبالتالى بتبقى Don't Care X وبنستخدمها فى تسهيل العملية لتصغير المعادلة

☆ **NAND circuits :**





☆ **Two-Level Implementations :** جمع الدالة فى صورة SOP بعدها



NAND-Procedure :

① simplify the function and express it in SOP form
② Draw a NAND gate for each product term of the expression that has at least two literals. the inputs to each NAND gate are the literals of the term. this procedure produces a group of first-level gates
③ second Level → AND-Invert, Invert-OR
④ complement the Literal of the term with one Literal

Multi Level AND-OR → NAND :
① AND → AND-Invert (conversion)
② OR → Invert-OR (conversion)

③ فرقع كل التنين Bubbles على نفس الخط لو ملقيتش زوز واحدة

☆ **NOR-circuits :** POS → Bubbles لو فى ال second Level ولو single Literal لو complement

☆ **other two Level Implementations :**

Equivalent Nondegenerate form (a)	(b)*	Implements the function	simplify F into
AND-NOR	NAND-AND	AND-OR-Invert	SOP zero's in map
OR-NAND	NOR-OR	OR-AND-Invert	POS is in map and complement variables

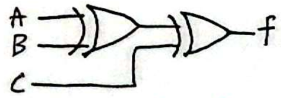
☆ **Exclusive-OR function :** $X \oplus Y = XY' + X'Y$

☆ odd function : the odd patterns of Input → output

☆ N-Variable XOR → $2^n/2$ minterms odd numbers of 1's

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

odd $F = A \oplus B \oplus C$



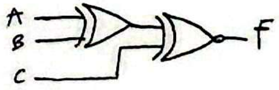
3-Input odd function

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

odd $F = A \oplus B \oplus C \oplus D$

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

Even $F = (A \oplus B \oplus C)'$

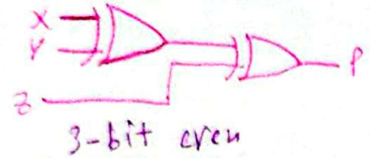


3-Input Even function

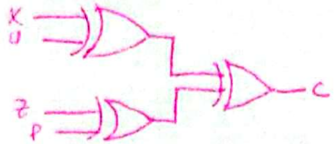
AB \ CD	00	01	11	10
00	1		1	
01		1		1
11	1		1	
10		1		1

Even $F = (A \oplus B \oplus C \oplus D)'$

★ Parity Generating and checking:



3-bit even Parity Generator



4-bit even Parity checker

★ بص يا معلم انت لو بتدوا على إنك تعمل نمط يكون زوجي فانت محتاج حاجه لو النمط فردي تحط 1 لو النمط زوجي تحط صفر فبالتالي دي XOR وإنما لو عايزه فردي هتجيب ال complement بس مش أكتر

★ بالنسبة لل checker فانت لو بتصمم checker لعدد زوجي يبقى محتاجه يطلع واحد لو عدد الدخلات كان فردي ويطلع صفر لو عدد الدخلات زوجي بالتالي XOR ولو checker فردي حط complement

★ HDL

Design Entry → functional simulation or Verification → Logic synthesis → timing Verification → fault simulation

★ test Bench: written in HDL, More complex To Prove circuit is functionally correct

// → single Line comment, /* ... */ → Multiple Line comment

★ Module Declaration: اسم الmodule, Port List: inputs, outputs Here

```
module (moduleName) (outputs, inputs);
  :      output ..., ..., ...;  } inputs, And outputs
         input  ..., ..., ...;  }
         wire   ..., ..., ...;  → سلك وسيط
         and  InstanceName (output, Input_1, Input_2, ...);
         or     //     (  //  ,  //  ,  //  , ...);
         not    //     (  //  ,  //  );
endmodule
```

★ Gate Delays: `timescale 1ns/100ps → Precision
(1ns ↓ unit measurement for time Delays)

and #(Time_Delay) ... ; طريقة الكتابة

★ في ال test bench ال outputs بيتحطوا في wire, وال inputs في reg

begin
⋮ } Block
end

inputA = (BitNumbers)'b(Sequence); → Assign Value for Input
initial
⋮
initial #(time-Delay) $finish; } اساوي وانهي

★ Boolean expressions: assign (output) = (expression);

|| → OR , && → AND , ! → Not

★ User-Defined Primitives (UDP):

① Declared by Primitive, name, Port List

② one output, first in Port List, (يكتب في ال output)

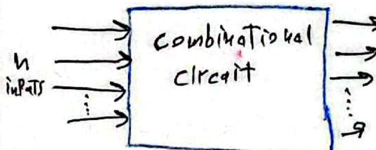
③ Any Number of inputs, بس بشرط يكتبوا بنفس الترتيب في ال table

④ table
   inputs : output;
   ⋮
   end table

⑤ end Primitive

Combinational Logic

★ the diagram of a combinational circuit has Logic gates with no Feedback Paths or memory Elements



★ For n inputs → 2^n Possible combinations of Binary input

★ Analysis Procedure: Boolean Functions from Logic Diagrams:

● Label all gate outputs, Determine Boolean functions for each gate

★ Design Procedure:

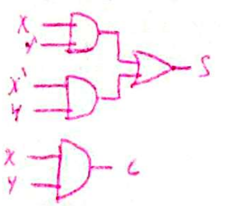
① Determine inputs, outputs, Assign Labels to each, ② Derive truth table

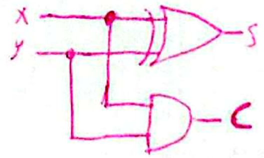
③ get simplified Boolean functions, ④ Draw Logic diagram, verify correctness

★ Binary-Adder-Subtractor:

★ Half Adder:

X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0





$S(\text{sum}) = X'Y + XY' = X \oplus Y$

$C(\text{carry}) = XY$

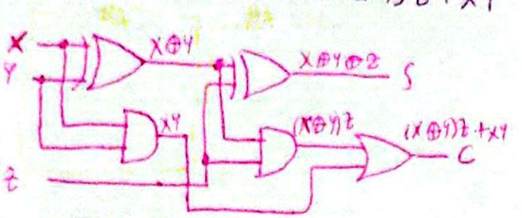
★ Full-Adder

$S = x'y'z + x'yz' + xy'z' + xyz = x \oplus y \oplus z$

$C = xy + xz + yz = (x \oplus y)z + xy$

x	y	z	C	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

بيجمع x,y,z



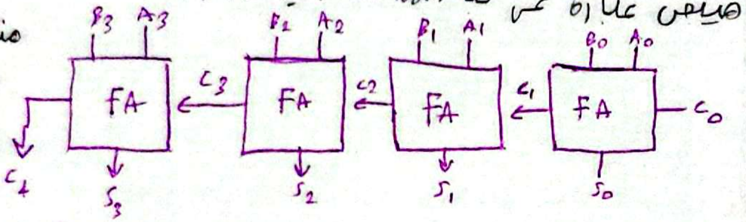
Full Adder

★ Binary Adder:

هيبقى عبارة عن صف full Adders بحيث يكونوا بعدد الـ Bits

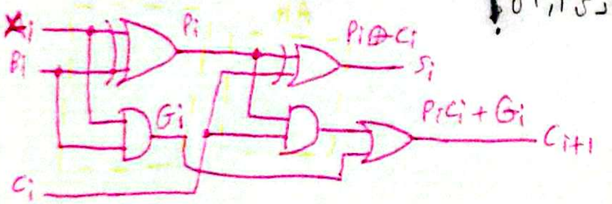
هنا مثلاً ده 4-bit Adder

بتدخل الـ carry بتاع الـ Full Adder للي بعده وهكذا



★ Carry Propagation:

في مشكلة في الـ Binary Adder العادي إنه كدا علشان نوصل لآخر FA محتاجين كل اللي قبله يكون طلع الـ carry علشان يا خده input ليه فهيبقى كل FA مستني اللي قبله فهنحل المشكلة دي ازاي؟



$P_i = A_i \oplus B_i, \; G_i = A_i B_i$

$S_i = P_i \oplus C_i, \; C_{i+1} = G_i + P_i C_i$

$C_0 = \text{input carry}, \; C_1 = G_0 + P_0 C_0$

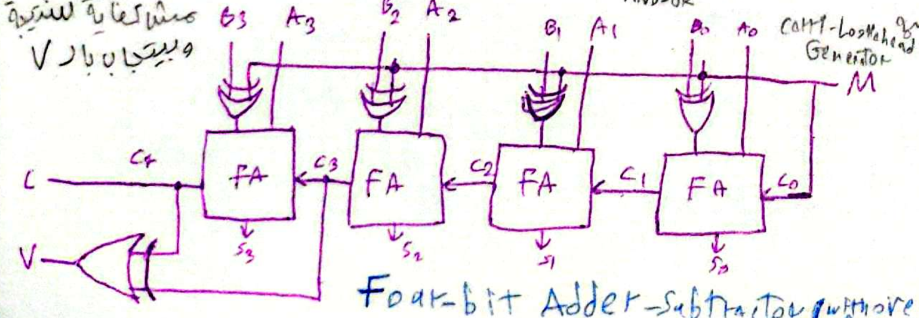
$C_2 = G_1 + P_1 G_0 + P_1 P_0 C_0$

⋮

كدا الـ C هيطلع كله مرة واحدة ومش هيحصل delay وبقا 2-level AND-OR واسمه Carry-Lookahead Generator

★ Binary Subtractor:

لما الـ M بصفر بيبقى Adder لكن لما الـ M بواحد بيبقى subtractor والـ overflow بيحصل لما اتنين signed يتجمعوا والمكان مش كفاية للنتيجة وبيتجاب بالـ V

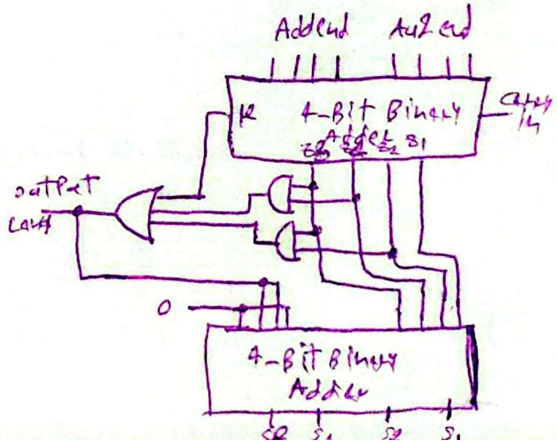


Four-bit Adder-Subtractor (with overflow Detection)

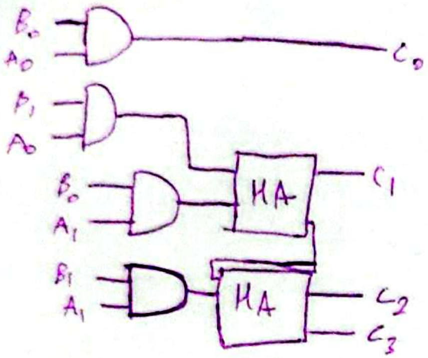
★ BCD Adder

Binary Sum					BCD Sum					Decimal
K	Z_8	Z_4	Z_2	Z_1	C	S_8	S_4	S_2	S_1	
0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	1	1
0	0	0	1	0	0	0	0	1	0	2
0	0	0	1	1	0	0	0	1	1	3
0	0	1	0	0	0	0	1	0	0	4
0	0	1	0	1	0	0	1	0	1	5
0	0	1	1	0	0	0	1	1	0	6
0	0	1	1	1	0	0	1	1	1	7
0	1	0	0	0	0	1	0	0	0	8
0	1	0	0	1	0	1	0	0	1	9
0	1	0	1	0	1	0	0	0	0	10
0	1	0	1	1	1	0	0	0	1	11
0	1	1	0	0	1	0	0	1	0	12
0	1	1	0	1	1	0	0	1	1	13
0	1	1	1	0	1	0	1	0	0	14
0	1	1	1	1	1	0	1	0	1	15
1	0	0	0	0	1	0	1	1	0	16
1	0	0	0	1	1	0	1	1	1	17
1	0	0	1	0	1	1	0	0	0	18
1	0	0	1	1	1	1	0	0	1	19

$C = K + Z_8 Z_4 + Z_8 Z_2$

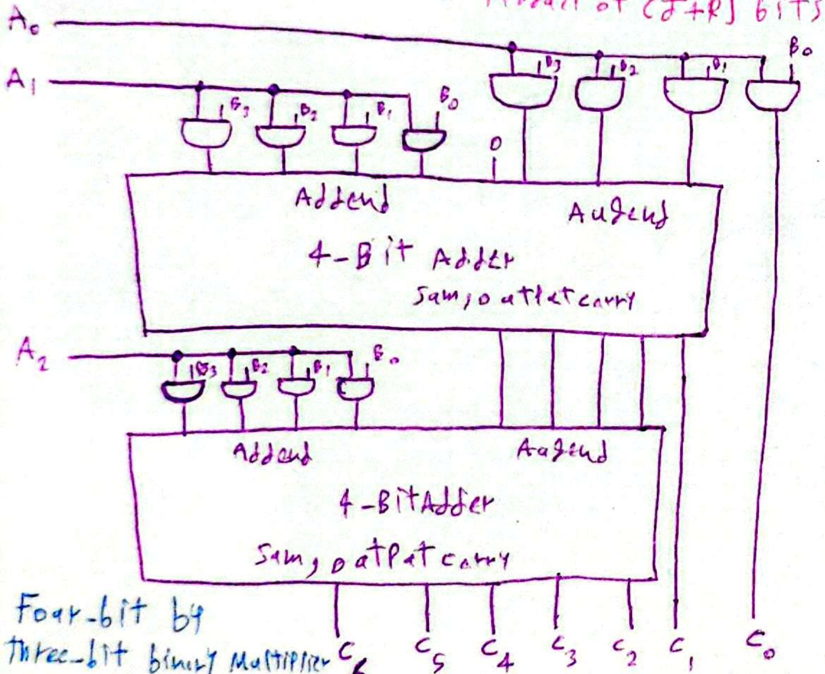


★ Binary Multipliers



		B_1	B_0
		A_1	A_0
		A_0B_1	A_0B_0
	A_1B_1	A_1B_0	
C_3	C_2	C_1	C_0

★ For J multiplier bits and K multiplicand bits, we need (J × K) AND gates and (J − 1) K-bit adders to produce a product of (J + K) bits



Four-bit by three-bit binary Multiplier

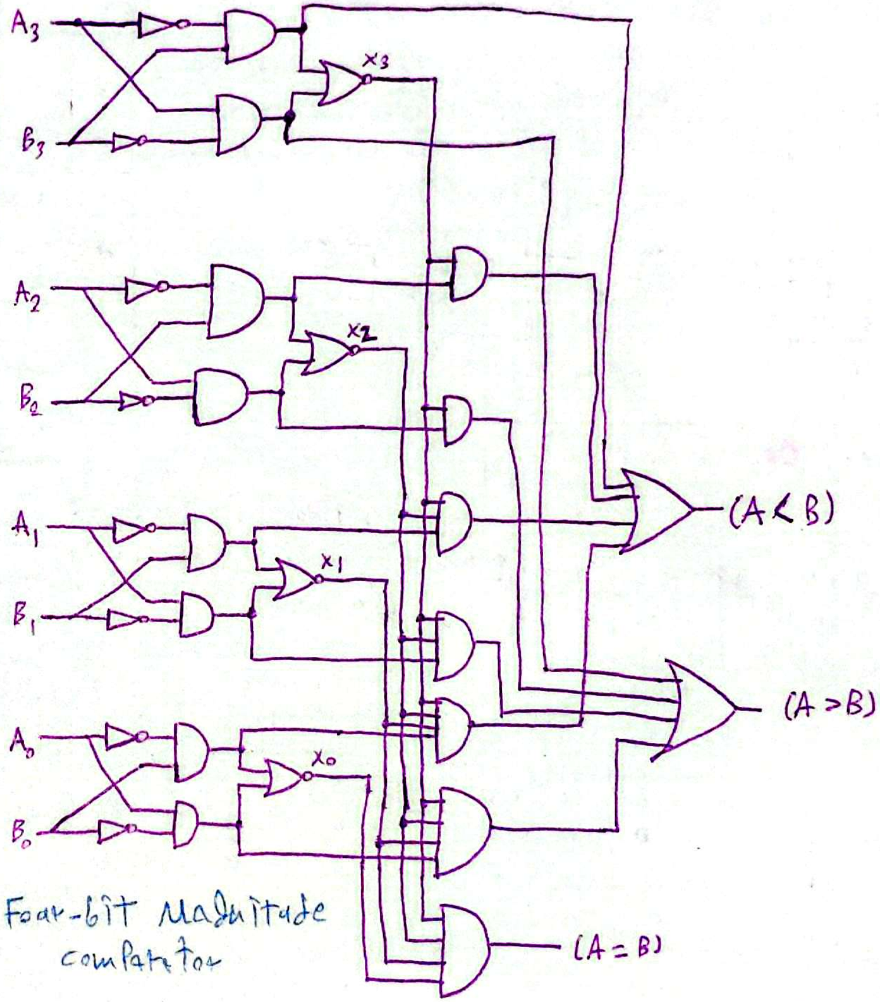
وهكذا نفس العملية مستكملك بيها مع باقي ال Multipliers ومستخدم ال Adders

★ Magnitude comparator: $x_i = A_iB_i + A_i'B_i' = (A_i \oplus B_i)'$ Equivalence pair

★ $(A = B) = x_3x_2x_1x_0$ → equal

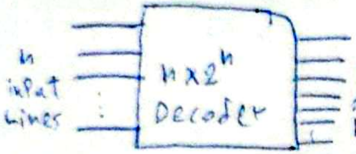
★ $(A > B) = A_3B_3' + x_3A_2B_2' + x_3x_2A_1B_1' + x_3x_2x_1A_0B_0'$ → greater than

★ $(A < B) = A_3'B_3 + x_3A_2'B_2 + x_3x_2A_1'B_1 + x_3x_2x_1A_0'B_0$ → Less than

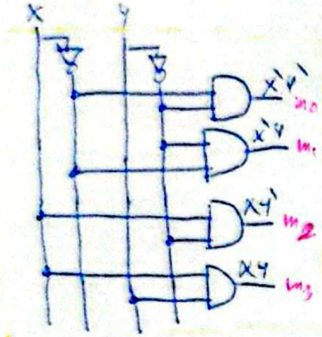


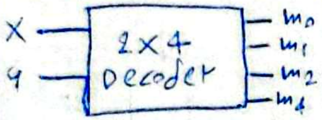
Four-bit Magnitude comparator

★ **Decoders:** each output line Represents certain minterm of the [redacted] Variables, we can Add enable to the Decoder to switch it on and off. فك تشغيل

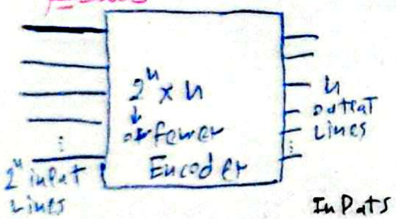


★ To Represent a function from Decoder you can OR the minterms of the function, and you could NOR the Maxterms of the function





★ **Encoder:** coding the input to certain outputs
تشفير



$x = D_4 + D_5 + D_6 + D_7$

$y = D_2 + D_3 + D_6 + D_7$

$z = D_1 + D_3 + D_5 + D_7$

Inputs								outputs		
D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	x	y	z
1	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	1	0	0
0	0	0	0	0	1	0	0	1	0	1
0	0	0	0	0	0	1	0	1	1	0
0	0	0	0	0	0	0	1	1	1	1

Octal-to-Binary Encoder

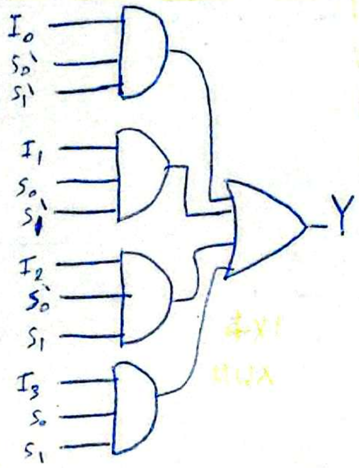
★ **Priority Encoder:**

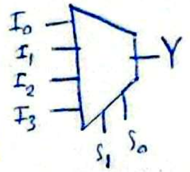
Inputs				outputs		
D_0	D_1	D_2	D_3	x	y	V
0	0	0	0	X	X	0
1	0	0	0	0	0	1
X	1	0	0	0	1	1
X	X	1	0	1	0	1
X	X	X	1	1	1	1

هو Encoder عادى بس لو فيه داس على زرارين يختار بـ Priority معينة زى هنا ال subscript الأكبر يبقى أولوية أعلى و ال V بيعبر عن إذا كان الرقم الى خارج مظبوط ولا لا إذا داس حد على ال inputs علشان لو محدش داس يدى صفر

★ **Multiplexers:** بنختار input معين بناءً على ال selection lines وندي لل output

★ 2^n input Line – n selection Line – 1 output Line



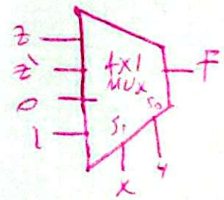


S_1	S_0	Y
0	0	I_0
0	1	I_1
1	0	I_2
1	1	I_3

★ نقدر تفعيل Enable Line برضو

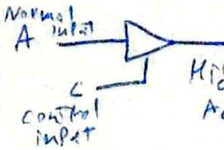
★ نقدر نعمل MUX أكبر من MUX أصغر عن طريق ال Input Lines

x	y	z	f	
0	0	0	0	f = z
0	0	1	1	
0	1	0	1	f = z'
0	1	1	0	
1	0	0	0	f = 0
1	0	1	0	
1	1	0	1	f = 1
1	1	1	1	



★ **three state Gates:**

★ Gates عادية بس بيبقى فى Input Line إضافية بتتحكم فى ال Gate



output Y = A if C = 1
High-impedance if C = 0
Act As open circuit

★ النوع ده من ال Buffers نقدر نستخدمه كـ MUX علشان ال control line

Verilog Model
combinational Logic

- User-defined Primitive → Truth table
- continuous Assignment (Dataflow) → Boolean equation
- Primitive (gate) (Gate-Level) → Schematic

Boolean equation ↔ Truth table, Boolean equation ↔ Schematic

☆ Predefined Primitives:

- and, - nand, - or, - nor, - xor, - xnor, - not, - buf

☆ Vectors:

output [0:3] y; → [0], [1], [2], [3]   (MSB = [0])
wire [7:0] x; → [7], ..., [0]   (MSB = [7])

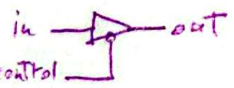
☆ Hierarchical description Design Methodologies:

Top Down — Bottom up

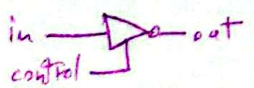
☆ three-state Gates:

gatename (output, input, control);

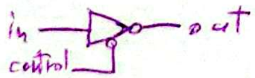
bufif1



bufif0



notif1



notif0

☆ لو عندك output معين و عايز يبقى tri-state بعد ما عملت Declaration
output y;
tri y; ← لازم تكتب tri زي

☆ nets: wire, wor, wand, tri, supply1, supply0

Dataflow Modeling: assign statement

Symbol	operation	Symbol	operation
+	Binary addition		
−	binary subtraction		
&	bitwise AND	&&	Logical AND
|	bitwise OR	||	Logical OR
^	bitwise XOR		
~	bitwise Not	!	Logical NOT
==	equality		
>	greater than		
<	Less than		
{}	concatenation		
?:	conditional		

☆ << → Logical left shift
☆ >> → // right //
☆ <<< → Arithmetic left //
☆ >>> → // right //

☆ condition ? true-expression : false-expression;

☆ Behavioral Modeling (always)

```
module ---;
output  y;
input   A, B, select;
reg     y;
always @(A or B or select)
  :
endmodule
```

كل outputs لازم يبقوا في ال reg

control expression

(A or B or select) → sensitivity list

اي تغيير دول الكود في ال block بيشتغل

☆ if-else: if (select) out = A;
else out = B;

☆ case (select)

```
2'b00: out = in_0;
2'b01: out = in_1;
2'b10: out = in_2;
2'b11: out = in_3;
default: ....;
endcase
```

default case يستحسن يبقى في

casex → x, z don't care
casez → z only don't care

☆ 'b, 'd, 'o, 'h
Binary, decimal, octal, Hexa

☆ repeat (no. of Repetitions)

☆ writing a simple Test Bench

module test_module_name;
// Declare local reg and wire identifiers
// Instantiate the design module under test
// Specify a stopwatch, using $finish to terminate the simulation
// Generate stimulus, using initial and always statements
// Display the output response (text or graphics (or both))
endmodule

inputs → reg, outputs → wire

$display() → بتكتب و تسيب سطر
$write() → بتكتب بس مش بتسيب سطر
$monitor() → كل ما يحصل تغيير بتكتب
$time → بتجيب الوقت الحالي
$finish → اقفل ال simulation

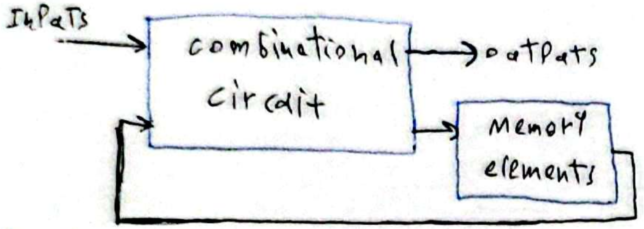
Task-name (format specification, argument list)
" ", A, B, ...)

%b → binary placeholder ← %B
%d → Decimal placeholder ← %D
%h → Hexa placeholder ← %H
%o → octal placeholder ← %O

$dumpfile("file-name.vcd")
$dumpvars(0, test_module_name)
ال WF

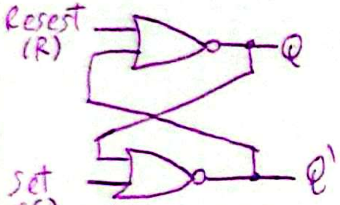
☆

Sequential Logic:



★ sequential circuit is specified by a time sequence of inputs, outputs, internal states

★ storage Elements: Latches: SR Latch:



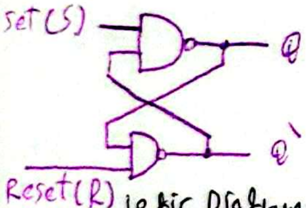
Logic Diagram

S	R	Q	Q'
1	0	1	0
0	0	1	0
0	1	0	1
0	0	0	1
1	1	0	0 (forbidden)

ثابت 00

Function table

NOR-Representation



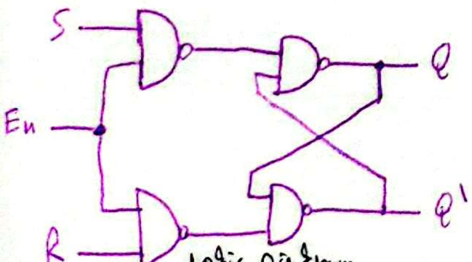
Logic Diagram

S	R	Q	Q'
1	0	0	1
1	1	0	1
0	1	1	0
1	1	1	0
0	0	1	1 (forbidden)

ثابت 11

Function table

NAND-Representation



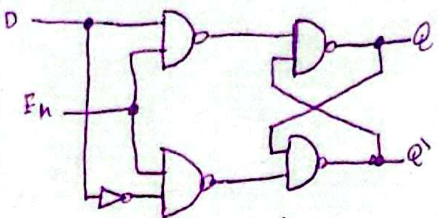
Logic Diagram

En	S	R	Next Q
0	X	X	No change
1	0	0	No change
1	0	1	Q=0; Reset state
1	1	0	Q=1; set state
1	1	1	Forbidden

Function table

SR Latch with control input

D-Latch:



Logic Diagram

En	D	next state of Q
0	X	No change
1	0	Q=0; reset state
1	1	Q=1; set state

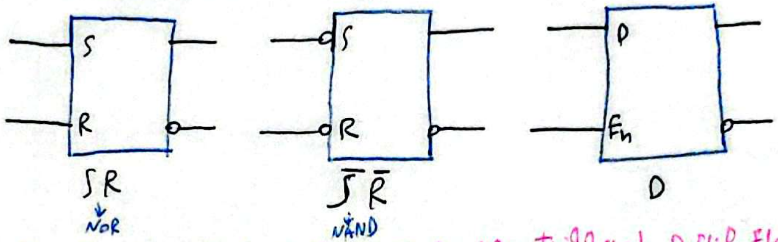
Function table

D	Q(t+1)	
0	0	reset
1	1	set

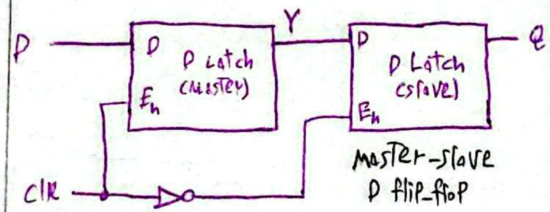
characteristic table

$$Q(t+1) = D$$

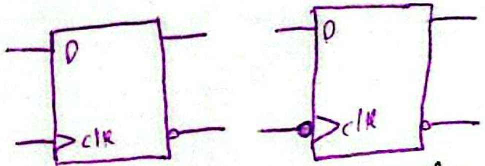
characteristic equation



★ storage Elements: Flip-Flops: Edge-triggered D flip-flop

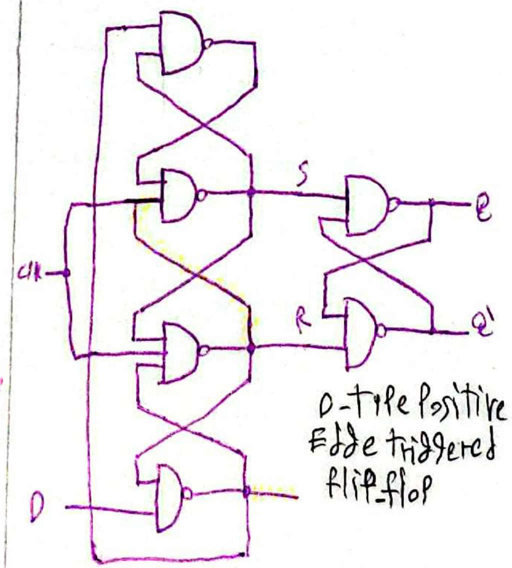


master-slave D flip-flop

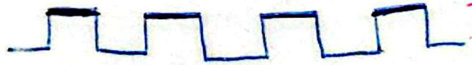


Positive edge Negative edge

D-flip-flop

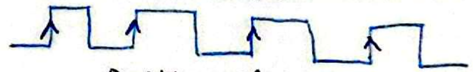


D-type Positive Edge triggered flip-flop

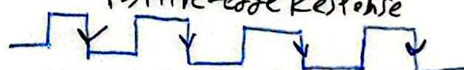


Response to Positive Level

LATCH



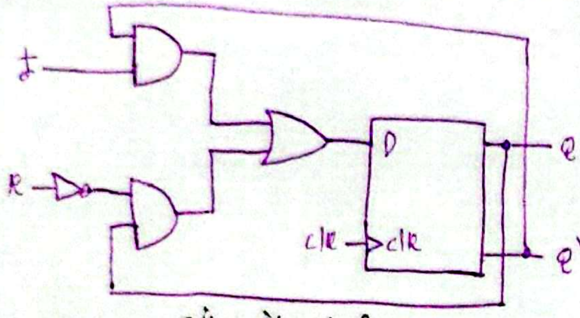
Positive-edge Response



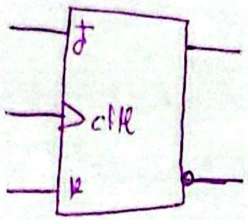
Negative-edge Response

flip-flop

★ Other flip-flops: JK flip-flop



Circuit Diagram



Graphic symbol

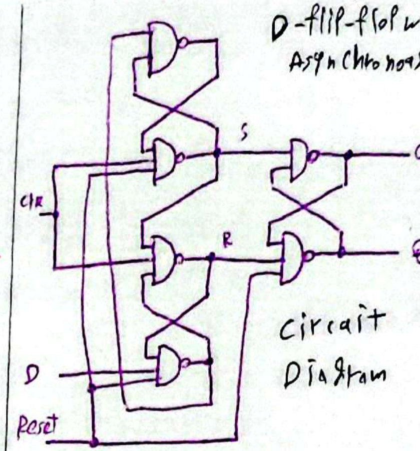
J	K	Q(t+1)	
0	0	Q(t)	No change
0	1	0	Reset
1	0	1	set
1	1	Q'(t)	complement

Characteristic table

$Q(t+1) = JQ' + K'Q$

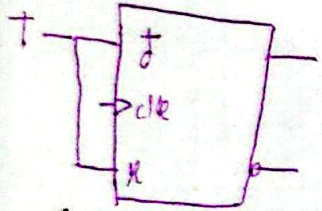
characteristic equation

D-flip-flop with Asynchronous Reset

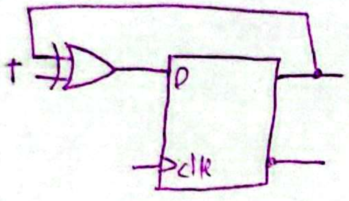


Circuit Diagram

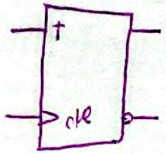
★ T-flip-flop



from J-K flip-flop



from D-flip-flop



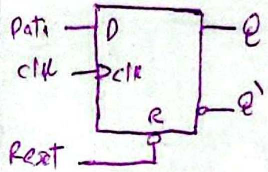
Graphic symbol

T	Q(t+1)	
0	Q(t)	No change
1	Q'(t)	complement

characteristic table

$Q(t+1) = T \oplus Q = TQ' + T'Q$

characteristic equation



Graphic symbol

R	clk	D	Q	Q'
0	X	X	0	1
1	↑	0	0	1
1	↑	1	1	0

Function table

★ Analysis of clocked sequential circuits

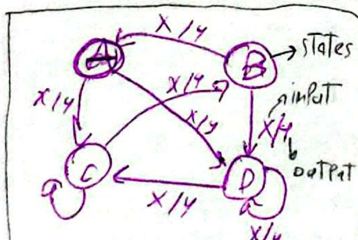
★ Circuit diagram → Equations → state table → state diagram

Present state		Input	Next state		output
A	B	X	A	B	y

state table → m flip-flops, n inputs → 2^{m+n} rows in state table

0 → $2^{m+n} - 1$

Present state		Next state				output	
		X=0		X=1		X=0	X=1
A	B	A	B	A	B	y	y



State Diagram

★ Next-state Values in JK-or-T-flipflops:

① Determine → flip-flop input equation

② List Binary Values of each input Equation

③ Use flip-flop characteristic Table

OR:

① Determine → flip-flop input equation

② Substitute input equations → flip-flop charakhteristic equation

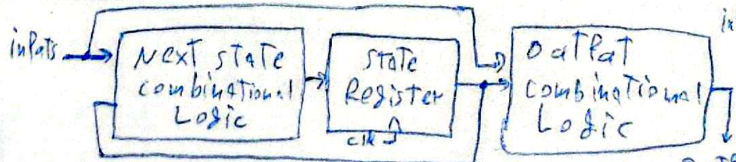
★ Mealy Vs. Moore FSM

★ ال output في ال moore معتمدة على ال P.S لكن ال mealy معتمدة على ال P.S وال inputs

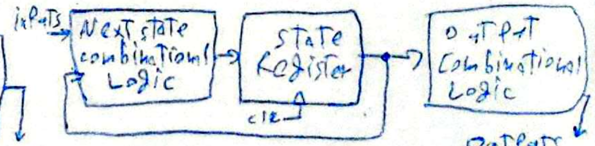
★ في ال moore ال outputs بتتغير مع ال clk علشان معتمدة على ال P.S فقط إنما ال mealy ممكن ال output يتغير مع تغيير ال input

★ ال output بتاع ال Mealy باخده قبل ال Active edge مباشرة

Mealy machine



Moore machine



★ Truth table describes a combinational circuit
★ state table describes a sequential circuit
★ characteristic table describes operation of a flip-flop
★ excitation table gives the values of flip flop inputs for a given state transition

★ **State Reduction:** اعمل state table وقارن
لو اتنين states عندهم نفس ال Next states لنفس
ال input وكذلك عندهم نفس ال output اقدر استغنى
عن واحد منهم

★ **State Assignment:** m-states,
n-bits $\rightarrow 2^n \geq m$

State	Assignment 1, Binary	Assignment 2, Gray code	Assignment 3, one-hot
a	000	000	0-00000-0001
b	001	001	0-00000-0010
c	010	011	0-00000-0100
d	011	010	0-00000-1000
e	100	110	0-00001-0000
f	101	111	0-00010-0000
g	110	101	0-00100-0000
h	111	100	0-01000-0000

بعدها بكتب ال state table

Present state	Next state x=0	Next state x=1	output x=0	output x=1

★ **Design Procedure:**
1. From the word description and specifications of the desired operation, derive a state diagram for the circuit
2. Reduce No. of states if necessary
3. Assign binary values to the states
4. obtain the binary-coded state table
5. choose the type of flip-flops to be used
6. Derive the simplified flip-flop input equations and output equations
7. Draw the Logic Diagram

★ **Excitation tables:**

J-K flip-flop

Q(t)	Q(t+1)	J	K
0	0	0	X
0	1	1	X
1	0	X	1
1	1	X	0

T-flip-flop

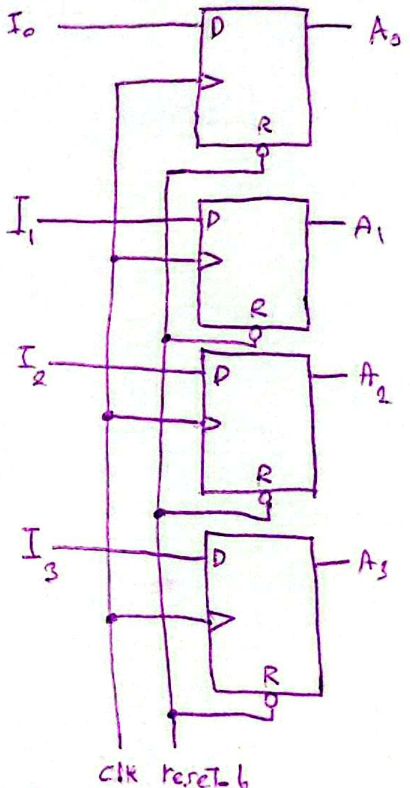
Q(t)	Q(t+1)	T
0	0	0
0	1	1
1	0	1
1	1	0

★ **self-correcting machine:**
this indicates that the system or machine is designed to handle errors or invalid states by redirecting them to predefined or valid states. this is a common feature in finite state machines (FSMs) or digital systems to ensure robustness

★ **Register:** group of flip-flops, each one of which shares a common clock and is capable of storing one bit of information.
n-bit register → store n bits of info.
↳ have n flip-flops

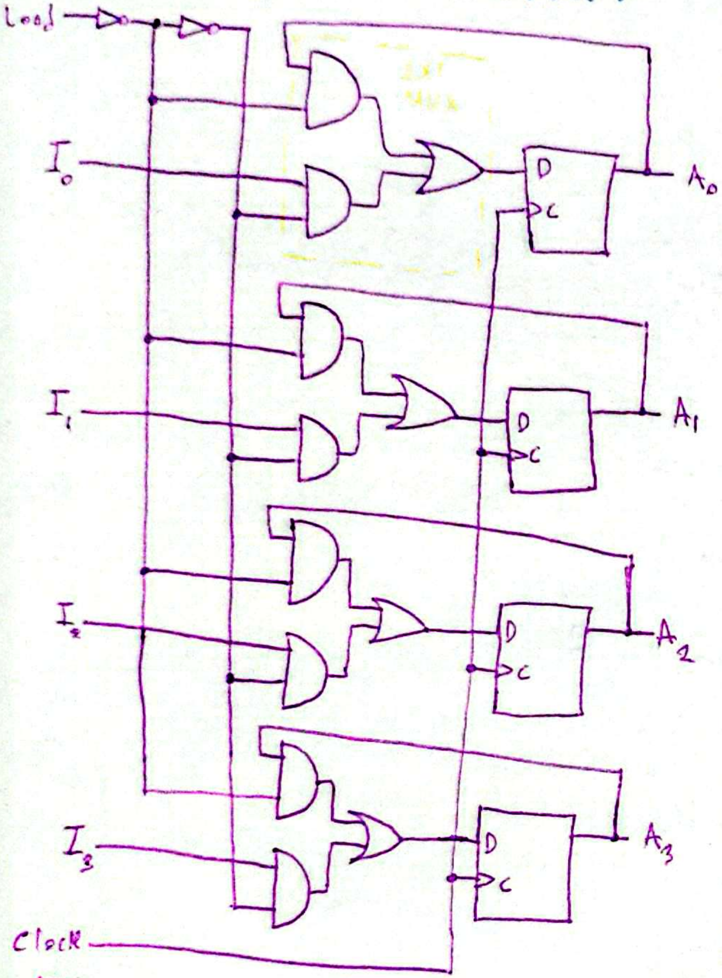
★ **counter:** Register that goes through a predetermined sequence of binary states

★ **four-bit Register:**



★ it's Register with Parallel Loads (All bits are loaded simultaneously with a common clock pulse)

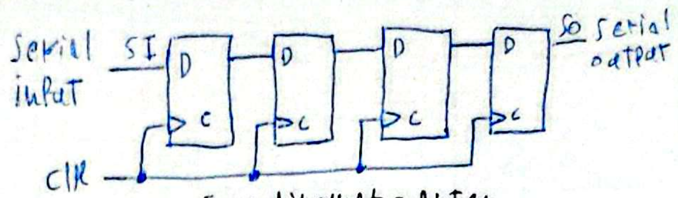
☆ four-bit register with Parallel Load :



☆ Load = 0 → No change

☆ Load = 1 → $(A_3A_2A_1A_0) = (I_3I_2I_1I_0)$

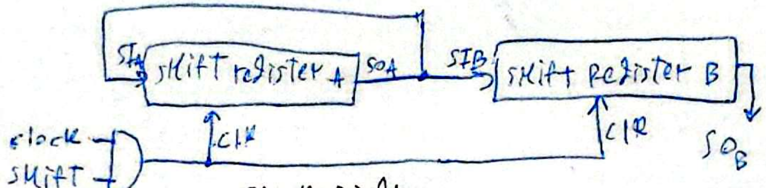
☆ Shift Registers :



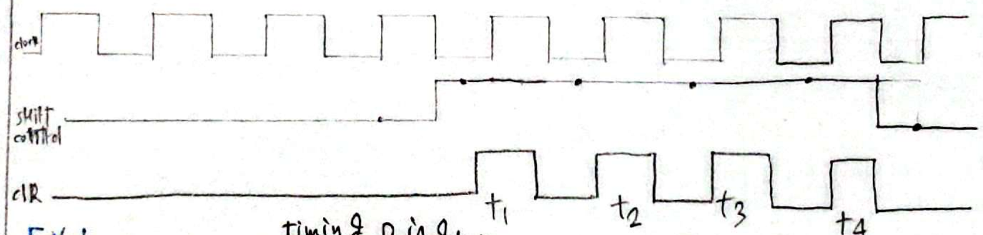
four-bit shift register

☆ serial is slower than Parallel but serial operations have the Advantage of Requiring fewer hard-ware components

☆ serial transfer : the datapath of a digital system is said to operate in serial Mode when information is transfered and manipulated one bit at a time



Block Diagram



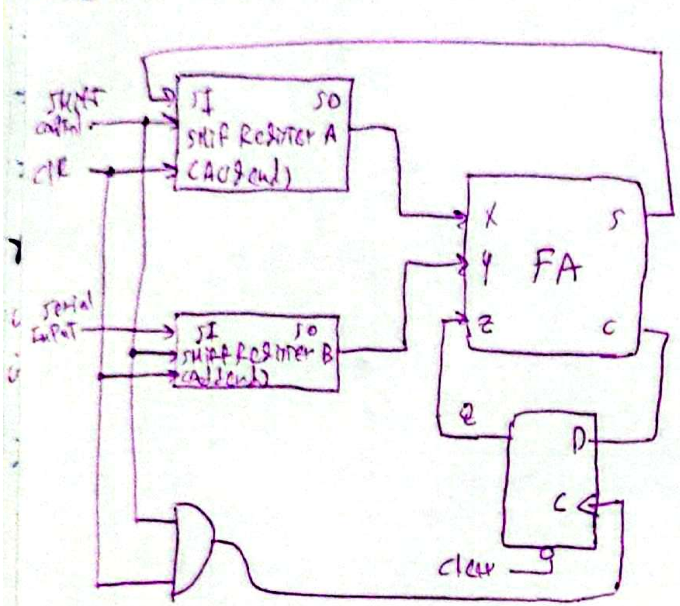
Timing Diagram

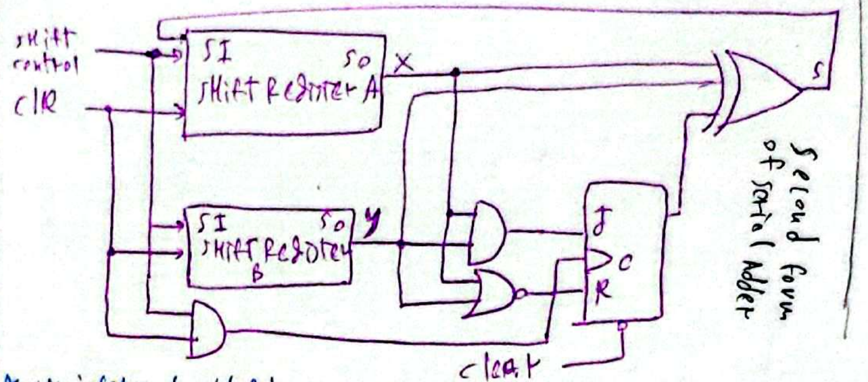
serial transfer from Register A to Register B

EX :

Timing Pulse	shift Register A	shift Register B
Initial Value	1 0 1 1	0 0 1 0
After t_1	1 1 0 1	1 0 0 1
After t_2	1 1 1 0	1 1 0 0
After t_3	0 1 1 1	0 1 1 0
After t_4	1 0 1 1	1 0 1 1

★ Serial Adder





mode control

S_1	S_0	Register operation
0	0	No change
0	1	shift right
1	0	shift left
1	1	Parallel Load

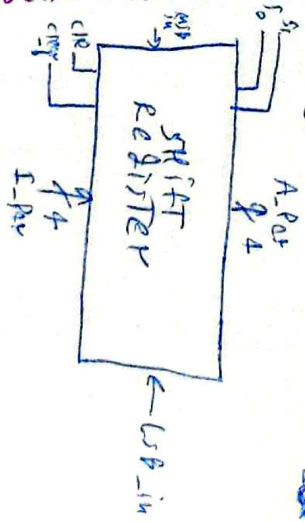
Present state	Inputs		Next state	output	flip-flop inputs	
Q	x	y	Q	S	J_Q	K_Q
0	0	0	0	0	0	X
0	0	1	0	1	0	X
0	1	0	0	1	0	X
0	1	1	1	0	1	X
1	0	0	0	1	X	1
1	0	1	1	0	X	0
1	1	0	1	0	X	0
1	1	1	1	1	X	0

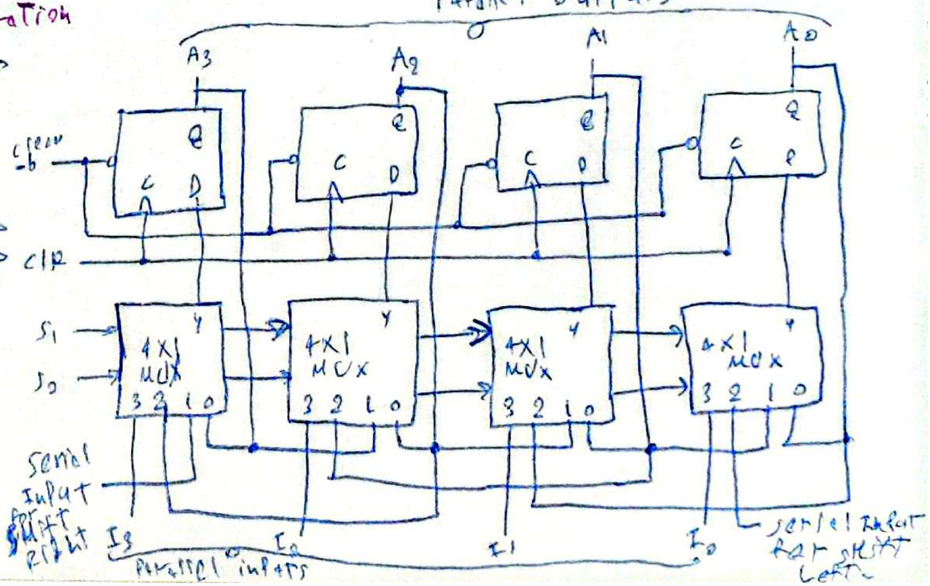
state table for serial adder

$J_Q = xy$, $K_Q = x'y' = (x+y)'$, $S = x \oplus y \oplus Q$

★ Universal shift Register:

① A clear control to clear the register to 0, ② A clock input to synchronize the operations
③ A shift-right control to enable the shift-right operation and serial input and output lines associated with the shift right
④ A shift-left control to enable the shift-left operation and serial input and output lines associated with the shift left
⑤ A parallel-load control to enable a parallel transfer and the n input lines associated with the parallel transfer ⑦ A control state that leaves the information in the register unchanged in response to the clock
⑥ n parallel output lines
other shift registers may have only some of the preceding functions, with at least one shift operation





★ Behavioral Modeling:

initial — always

initial	always
single-pass behaviour	cyclic behaviour

```
initial
  begin
    clock = 1'b0;
    repeat (30)   -> no. of transitions
      #10 clock = ~clock;
  end
```

```
initial
  begin
    clock = 1'b0;
  end
initial #300 $finish;
always #10 clock = ~clock;
```

clock

Indefinite loop

```
initial begin clock = 0; forever #10 clock = ~clock; end
```

event control operator

```
always @ (event control expression) begin
  // Procedural assignment statements that execute
  // when condition is met
end
```

لازم الى هنا و هيكون عليه شمال يكون من نوع reg

```
always @ (A, B, C)
```

هنا مع اى تغيير فيهم هينفذ الكود

```
always @ (posedge clock, negedge reset)
```

عند ال transition ايس

★ Procedural Assignment:

Blocking	nonblocking
(=)	(<=)
executed sequentially	executed concurrently
cyclic Behaviour use	edge sensitive
ex: combinational circuits	Modeling latches Behaviour

بيحل كل expressions الى على اليمين وبعدها ب الـ Assign

★ HDL models of flip-flops:

★ (D-Latch):

```
module D_Latch (Q, D, enable);
  output Q;
  input D, enable;
  reg Q;
  always @ (enable or D)
    if (enable) Q <= D;
endmodule
```

اى تغيير فيهم هينفذ الكود

```
// Verilog 2001, 2005:
module D_Latch (output reg Q, input enable, D);
  always @ (enable, D)
    if (enable) Q <= D;
endmodule
```

اى تغيير فيهم هينفذ الكود

★ (D-Type flip-flop):

```
module D_FF (Q, D, Clk);
  output Q;
  input D, Clk;
  reg Q;
  always @ (posedge Clk)
    Q <= D;
endmodule
```

```
// V2001, V2005:
module DFF (output reg Q, input D, Clk, rst);
  always @ (posedge Clk, negedge rst)
    if (!rst) Q <= 1'b0;
    else Q <= D;
endmodule
```

★ Rules to enable the tool to infer the clock correctly:

1. if, else_if → Asynchronous event
2. last else → clock event
3. Asynchronous events are tested first

★ Alternative flip-flop models

JK-flip-flop, T-flip-flop from D-flip-flop:

```
module TFF (Q, T, Clk, rst);
  output Q;
  input T, Clk, rst;
  wire DT;
  assign DT = Q ^ T;
  DFF TF1 (Q, DT, Clk, rst);
endmodule
```

assign DT = Q ^ T;
DFF TF1 (Q, DT, Clk, rst);
continuous assignment

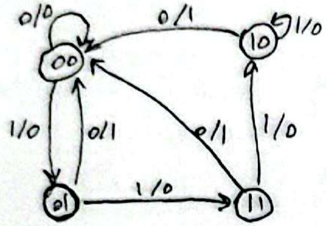
```
module JKFF (output reg Q, input J, K,
  Clk, rst);
  wire JK = (J & ~Q) | (~K & Q);
  DFF JK1 (Q, JK, Clk, rst);
endmodule
```

```
module DFF (output reg Q, input D, Clk,
  rst);
  always @ (posedge Clk, negedge rst)
    if (!rst) Q <= 1'b0;
    else Q <= D;
endmodule
```

★★ JK Flip-flop:

```
module JK_FF (input J, K, Clk,
  output reg Q,
  output Q_b);
  assign Q_b = ~Q;
  always @ (posedge Clk)
    case ({J, K})
      2'b00: Q <= Q;
      2'b01: Q <= 1'b0;
      2'b10: Q <= 1'b1;
      2'b11: Q <= !Q;
    endcase
endmodule
```

★ State Diagram-Based HDL Models: Mealy Machine: Zero Detector

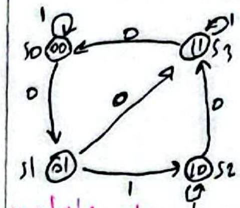


P.S		N.S x=0		N.S x=1		Output x=0	Output x=1
A	B	A	B	A	B	y	y
0	0	0	0	0	1	0	0
0	1	0	0	1	1	1	0
1	0	0	0	1	0	1	0
1	1	0	0	1	0	1	0

```
module Mealy_Zero_Detector (
   output reg y_out,          → output
   input x_in, clock, reset   → inputs
);
   reg [1:0] state, next_state;   → states
   parameter S0 = 2'b00, S1 = 2'b01, S2 = 2'b10, S3 = 2'b11;
                                  ↑ constants defined by parameter in Verilog
   always @(state, x_in)   → combinational circuit changes by level
     case (state)
 Blocking
       S0: if (x_in)  next_state = S1; else next_state = S0;
       S1: if (x_in)  next_state = S3; else next_state = S0;
       S2: if (~x_in) next_state = S0; else next_state = S2;
       S3: if (x_in)  next_state = S2; else next_state = S0;
     endcase
   always @(state, x_in)   → combinational circuit changes by level
 Blocking
     case (state)
       S0: y_out = 0;
       S1, S2, S3: y_out = ~x_in;
     endcase
   always @(posedge clock, negedge reset)   → edge transition → flip-flop
 Non Blocking
     if (!reset) state <= S0;
     else state <= next_state;
end module
```

fork ... join → in TB executes lines in parallel

Moore Machine: Zero Detector



P.S		N.S x=0		N.S x=1	
A	B	A	B	A	B
0	0	0	1	0	0
0	1	1	1	1	0
1	0	1	1	1	0
1	1	0	0	1	1

```
module Moore_Model_Fig_5_19C
   output [1:0] y_out,
   input x_in, clock, reset
);
   reg [1:0] state;   → No Need for N.S
   parameter S0 = 2'b00, S1 = 2'b01,
             S2 = 2'b10, S3 = 2'b11;
   always @(posedge clock, negedge reset)
     if (!reset) state <= S0;   → reset Asynchronous
     else case (state)
 State Transition At Posedge (Non Blocking)
       S0: if (~x_in) state <= S1;
           else state <= 0;
       S1: if (x_in) state <= S2;
           else state <= S3;
       S2: if (~x_in) state <= S3;
           else state <= S2;
       S3: if (~x_in) state <= S0;
           else state <= S3;
     endcase
   assign y_out = state;   → continuous Assignment
end module
```

★ Structural Description of Clocked Sequential Circuits:

★ بتعمل ال combinational بال assign و ال sequential always
ونشيل الحاجات القديمة زي ال structural
بتاع ال combinational

★ Ripple Counters:

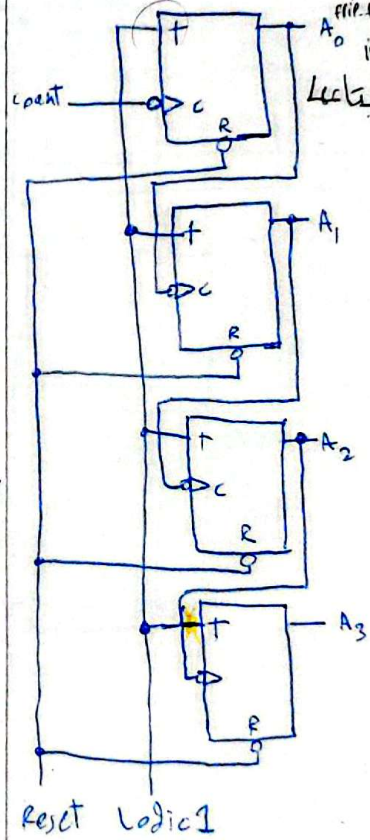
★ Binary Ripple Counter:

A_3	A_2	A_1	A_0
0	0	0	0
0	0	0	1
0	0	1	0
0	0	1	1
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	0	0	0
1	0	0	1

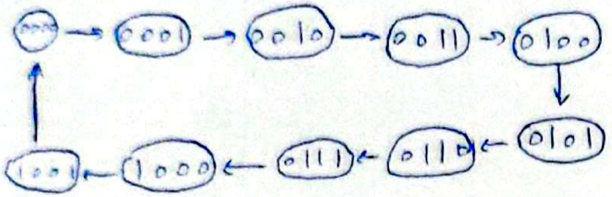
Negative Edge clock

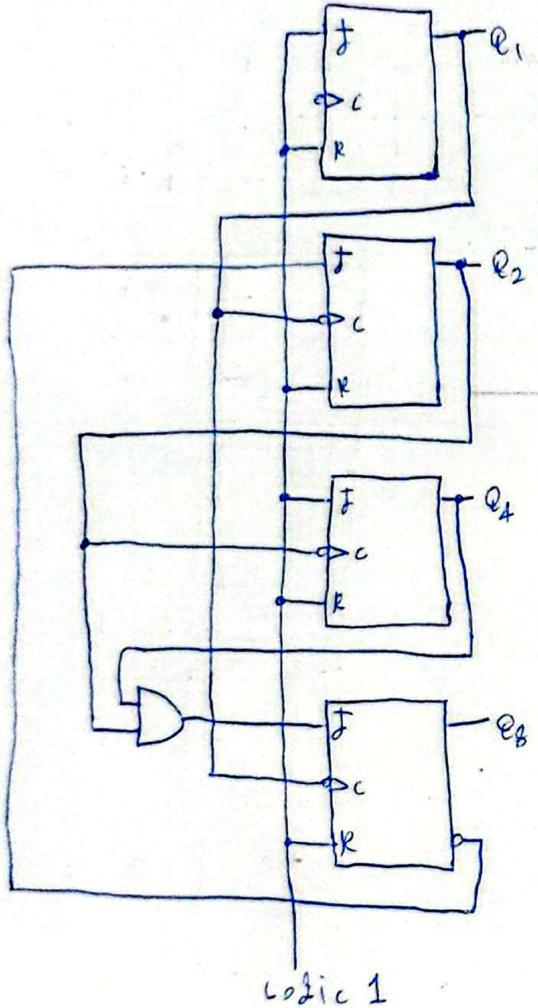
Positive edge clock

تقدر تعملها بال D flip-flop.
بس هتخلي ال input بتاع ال D هو $\bar{Q}$ بتاعها



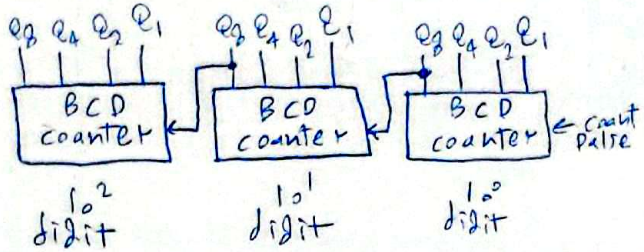
✯ BCD Ripple counter:



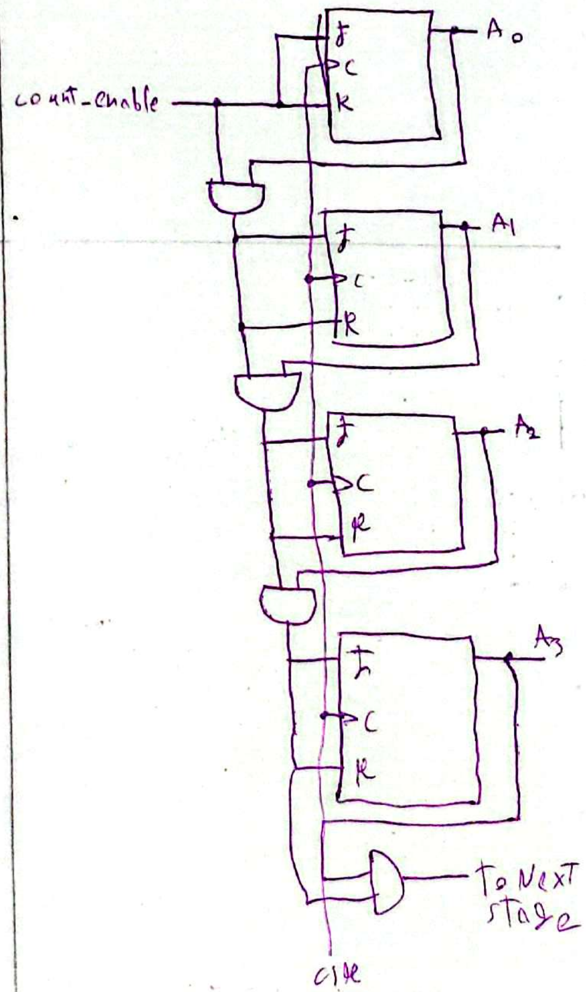


✯ Decade counter:

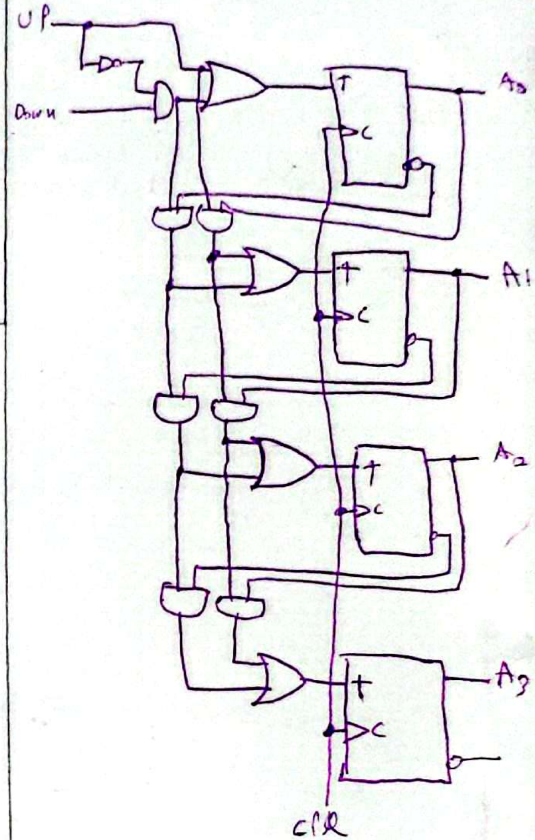
كل لما يوصل لـ 9 وبعدها ينزل لـ 0 اللي بعده يعد وهكذا



✯ Four-bit synchronous Binary counter:



✯ Four-bit-up-down binary counter:



☆ BCD counter:

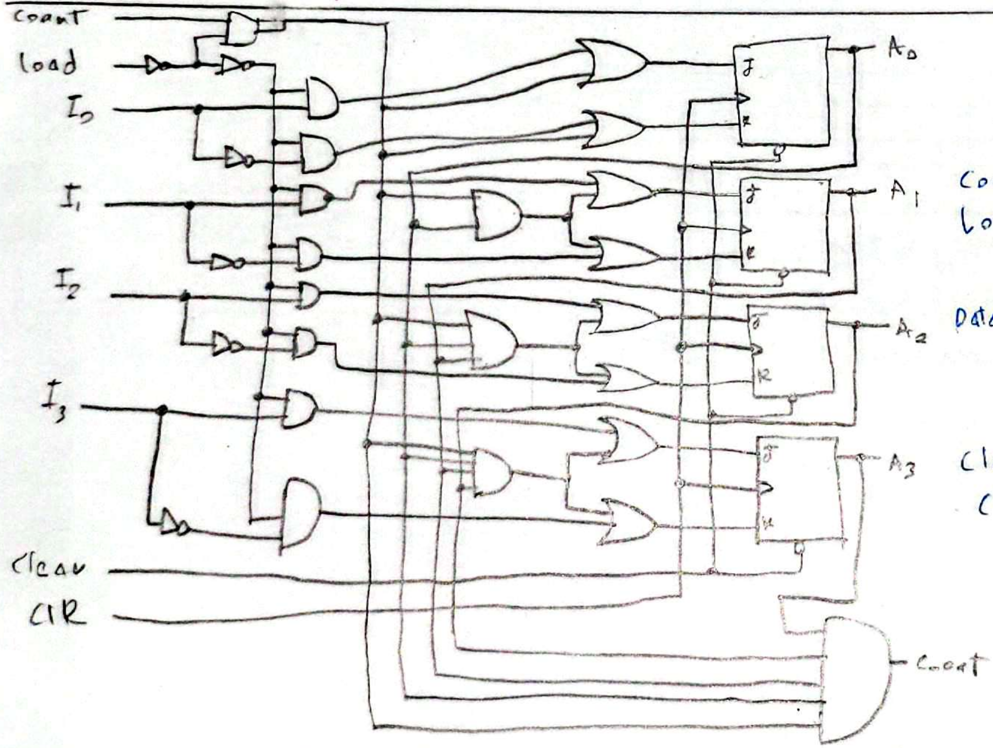
Present state				Next state				output	Flip-flops Inputs			
Q_8	Q_4	Q_2	Q_1	Q_8	Q_4	Q_2	Q_1	y	T_{Q_8}	T_{Q_4}	T_{Q_2}	T_{Q_1}
0	0	0	0	0	0	0	1	0	0	0	0	1
0	0	0	1	0	0	1	0	0	0	0	1	1
0	0	1	0	0	0	1	1	0	0	0	0	1
0	0	1	1	0	1	0	0	0	0	1	1	1
0	1	0	0	0	1	0	1	0	0	0	0	1
0	1	0	1	0	1	1	0	0	0	0	1	1
0	1	1	0	0	1	1	1	0	0	0	0	1
0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	0	0	1	0	0	0	0	1
1	0	0	1	0	0	0	0	1	1	0	0	1

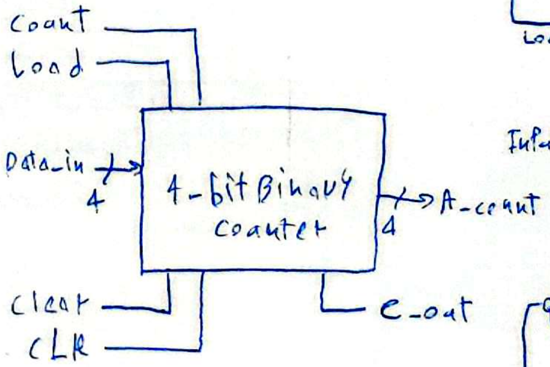
$T_{Q_1} = 1$, $T_{Q_2} = Q_8' Q_1$, $T_{Q_4} = Q_2 Q_1$

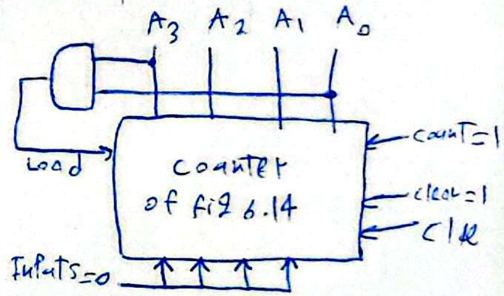
$T_{Q_8} = Q_8 Q_1 + Q_4 Q_2 Q_1$, $y = Q_8 Q_1$

☆ Binary counter with Parallel Load:

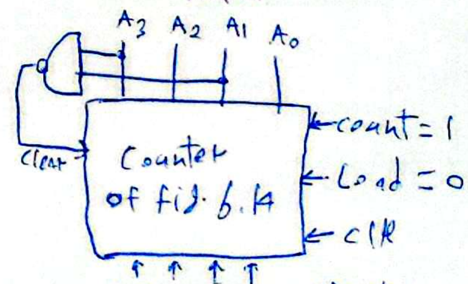
Clear	CLK	Load	count	Function
0	X	X	X	Clear to 0
1	↑	1	X	Load inputs
1	↑	0	1	count Next Binary state
1	↑	0	0	No change







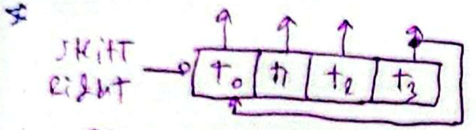
BCD counter using Load input



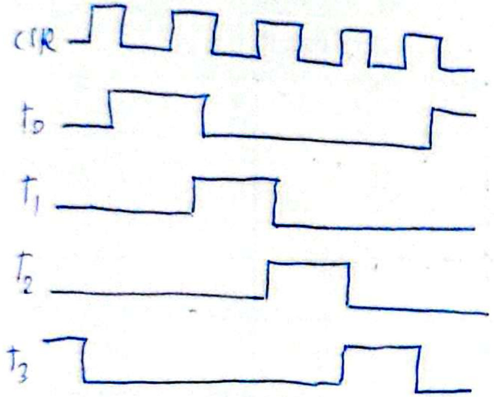
Inputs has no effect

BCD counter using the clear input

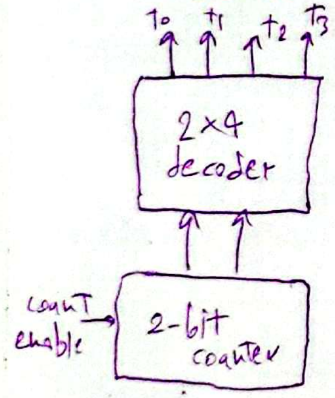
☆ Ring counter:



Ring-counter (Initial value=1000)

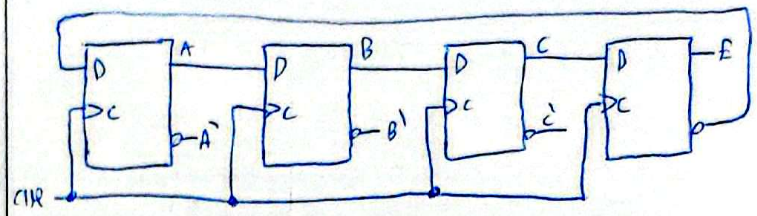


sequence of four timing signals



2^n Timing signals → 2^n flip-flops (shift register)
→ n-bit counter + n-to-2^n Line Decoder

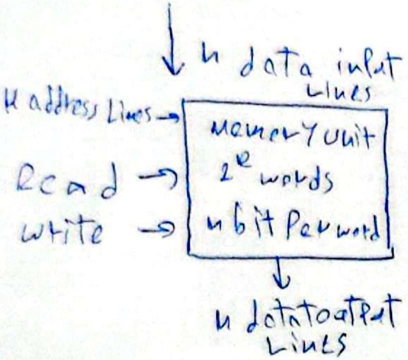
☆ Johnson counter:



Sequence Number	Flip-flop outputs A	B	C	E	AND gate required for output
1	0	0	0	0	A' E'
2	1	0	0	0	A B'
3	1	1	0	0	B C'
4	1	1	1	0	C E'
5	1	1	1	1	A E
6	0	1	1	1	A' B
7	0	0	1	1	B' C
8	0	0	0	1	C' E

☆ inter-Assignment Delay: $Q <= \#2 \sim Q$
effect of postponing Assignment
هينفذ بعد ثانيتين بيستخدم في ال Ripple counter مثلا

☆ Random-Access Memory (RAM):



$2^k \geq m$

m → total number of words

k → No. of Addresses

1024 × 16 memory

Memory address		
Binary	Decimal	Memory content
0000_0000	0	16 Bit
0000_0001	1	16 Bit
0000_0010	2	16 Bit
⋮	⋮	⋮
1111_1101	1021	16 Bit
1111_1110	1022	16 Bit
1111_1111	1023	16 Bit

☆ Write & Read operations:

Write steps:

① Apply the binary Address of the desired word to the address lines

② Apply the data bits that must be stored in memory to the data input lines

③ Activate the write inputs

Read steps:

① Apply the binary address of the desired word to the address lines

② Activate the read input

reg $\underbrace{[15:0]}_{\text{word Length}}$ memword $\underbrace{[0:1023]}_{\text{no. of words}}$;

☆ Access time: time required to select a word and read it

☆ cycle time: time required to complete a write operation

Ram → SRam (latch store Binary)
↳ DRam (Power off Lose info.)

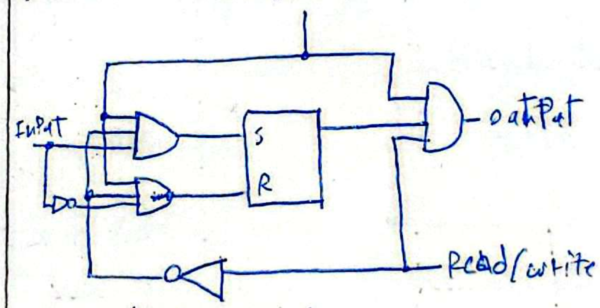
(electric charges on capacitors on MOS chip)

(Must Refresh memory to Restore decaying charge)

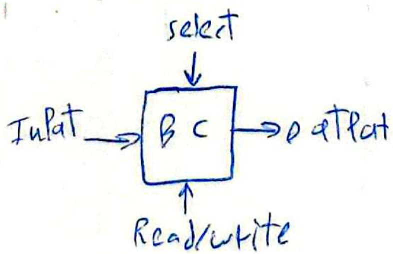
Volatile

☆ Magnetic Disk & ROM Nonvolatile

☆ Memory cell:

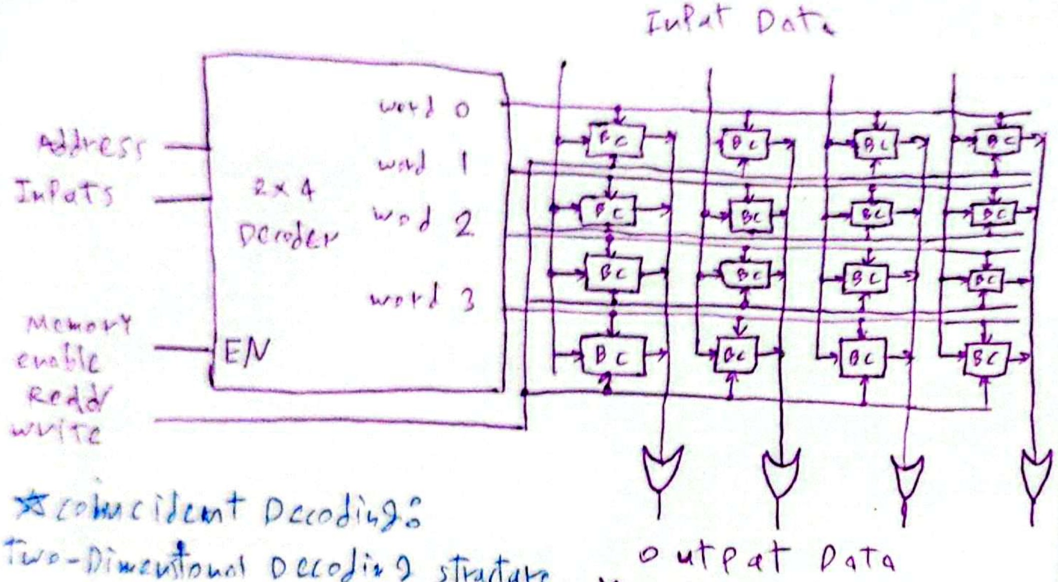


Logic Diagram



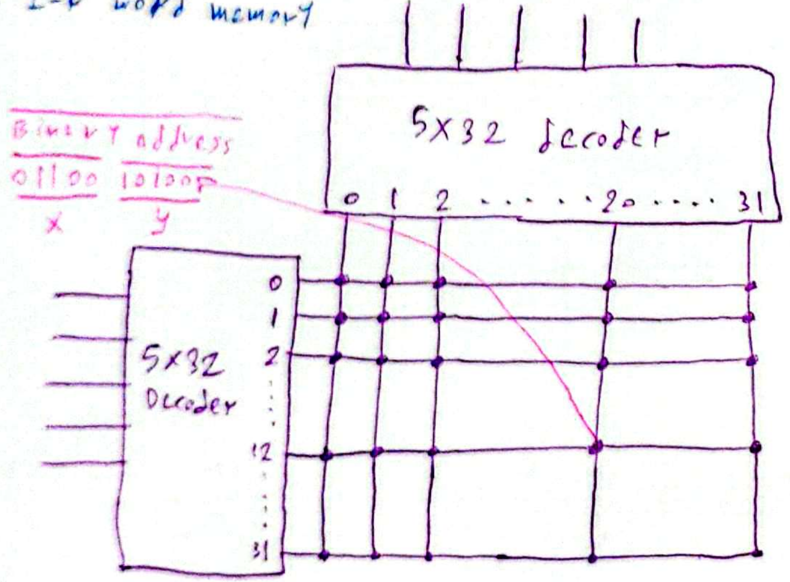
Block diagram

4x4 RAM:

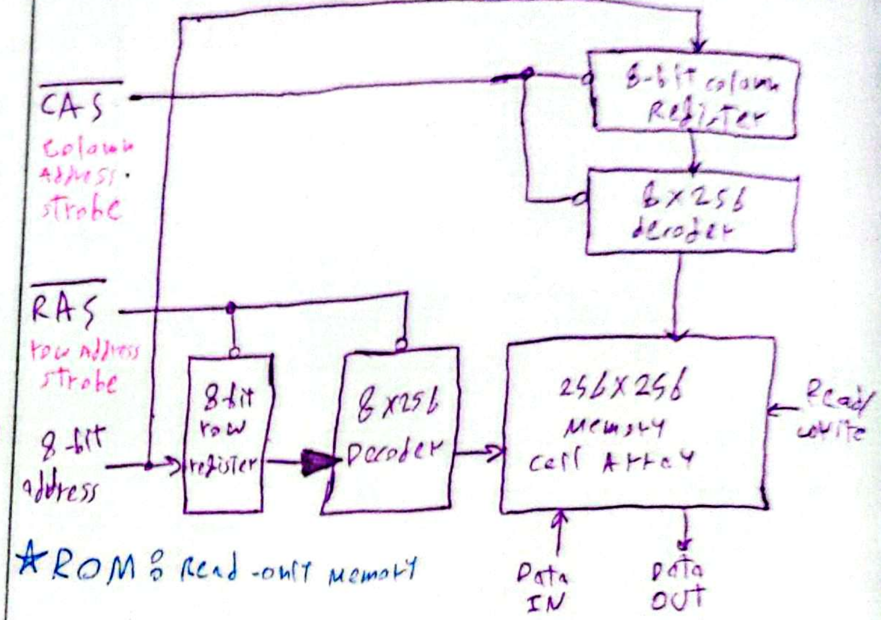


☆ Coincident Decoding:

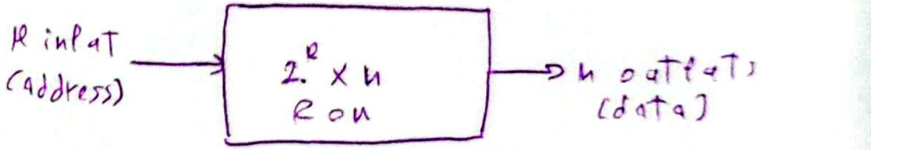
Two-Dimensional Decoding structure for a 1-K word memory



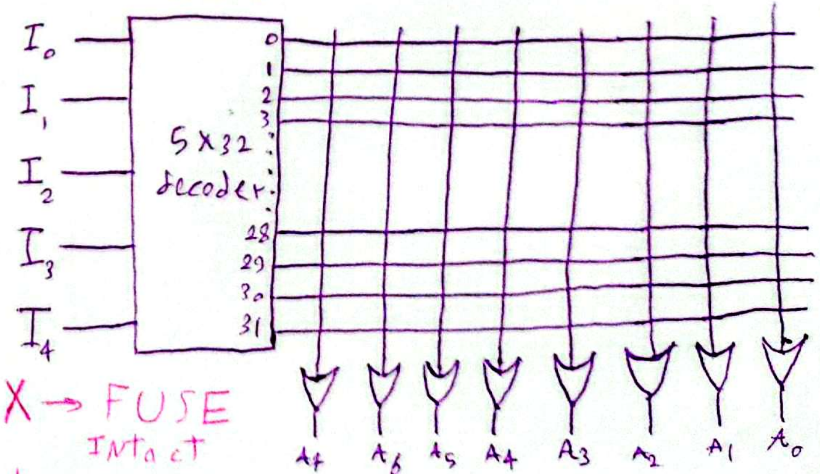
☆ Address Multiplexing for a 64K DRAM:



☆ ROM: Read-only memory



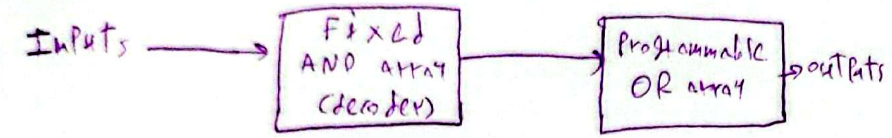
32X8 ROM:



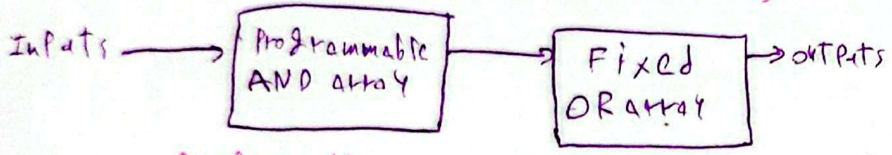
X → FUSE Intact

+ → FUSE blown

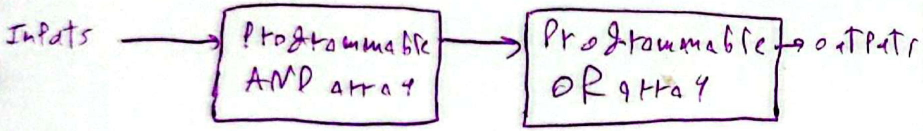
☆ types of ROMs:



Programmable read-only memory (PROM)



Programmable array Logic (PAL)



Programmable Logic Array (PLA)

Combinational PLDs (Programmable Logic Device) →
- PROM
- PAL
- PLA

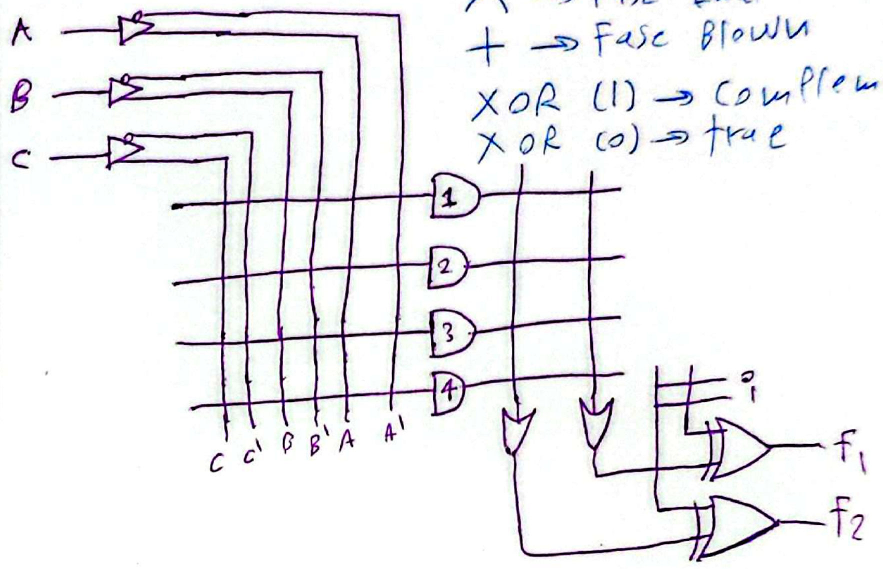
☆ PLA: three Inputs, four Product term, two outputs

X → fuse Intact

\+ → fuse Blown

XOR (1) → complement

XOR (0) → true



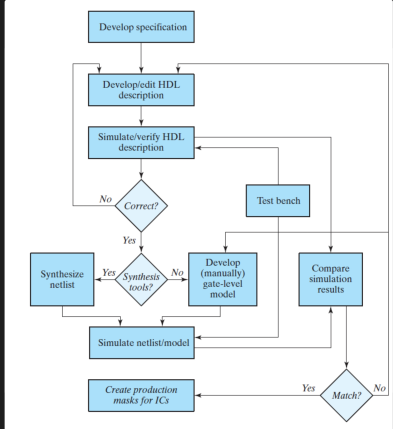


FIGURE 8.1
A simplified flowchart for HDL-based modeling, verification, and synthesis

For synthesizable sequential circuits, the event control expression must be sensitive to the positive or the negative edge of the clock (synchronizing signal), but not to both.

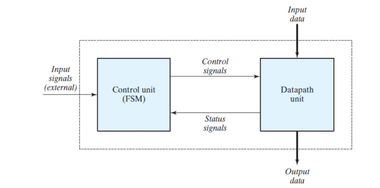


FIGURE 8.2
Control and datapath interaction

Thus, a synthesis tool will interpret the logic to be combinational; failure to assign a value to every variable on every path of logic implies the need for a transparent latch (memory) to implement the logic. Synthesis tools will provide the latch, wasting silicon area.

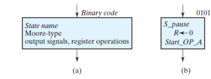


FIGURE 8.3
ASM chart state box

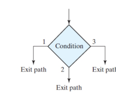
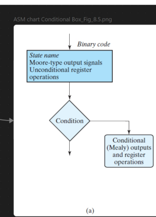


FIGURE 8.4
ASM chart decision box

ALGORITHMIC STATE MACHINES (ASMs)



An algorithmic state machine (ASM) chart is a flowchart that has been developed to specifically define algorithms for execution on digital hardware.

Consequently, we now consider three levels of design: structural description, RTL description, and algorithmic based behavioral description.

The steps to form an ASMD chart are:

1. Form an ASM chart showing only the states of the controller and the input signals 2 that cause state transitions,

2. Convert the ASM chart into an ASMD chart by annotating the edges of the ASM chart to indicate the concurrent register operations of the datapath unit (i.e., register operations that are concurrent with a state transition), and

3. Modify the ASMD chart to identify the control signals that are generated by the controller and that cause the indicated operations in the datapath unit.

Digital System in RTL

RTL Representations

- The set of registers in the system.
- The operations that are performed on the data stored in the registers.
- The control that supervises the sequence of operations in the system.

Operations In Digital Systems

1. **Transfer operations, which transfer (i.e., copy) data from one register to another.**

2. **Arithmetic operations, which perform arithmetic (e.g., multiplication) on data in registers.**

3. **Logic operations, which perform bit manipulation (e.g., logical OR) of non numeric data in registers.**

4. **Shift operations, which shift data between registers.**

Operations In Digital Systems Examples

R1dR1 + R2 Add contents of R2 to R1 (R1 gets R1 + R2)

R3dR3 + 1 Increment R3 by 1 (count upwards)

R4dshr R4 Shift right R4

R5d0 Clear R5 to 0

In general, the

blocking assignment operator (=) is used in a procedural assignment statement only

when it is necessary to specify a sequential ordering of multiple assignment statements.

RTL in HDL

(a) assign S = A + B; // Continuous assignment for addition operation

(b) always @ (A, B) // Level-sensitive cyclic behavior

S = A + B; // Combinational logic for addition operation

(c) always @ (negedge clock) // Edge-sensitive cyclic behavior

begin

RA = RA + RB; // Blocking procedural assignment for addition

RD = RA; // Register transfer operation

end

(d) always @ (negedge clock) // Edge-sensitive cyclic behavior

begin

RA <= RA + RB; // Nonblocking procedural assignment for addition

RD <= RA; // Register transfer operation

end

<= ----> Non-Blocking

= ----> Blocking

Table 8.1
Verilog 2001 HDL Operators

Operator Type	Symbol	Operation Performed
Arithmetic	+	addition
	–	subtraction
	*	multiplication
	/	division
	%	modulus
	**	exponentiation
Bitwise or Reduction	~	negation (complement)
	&	AND
	|	OR
	^	exclusive-OR (XOR)
Logical	!	negation
	&&	AND
	||	OR
Shift	>>	logical right shift
	<<	logical left shift
	>>>	arithmetic right shift
	<<<	arithmetic left shift
	{ , }	concatenation
Relational	>	greater than
	<	less than
	==	equality
	!=	inequality
	===	case equality
	!==	case inequality
	>=	greater than or equal
	<=	less than or equal

Loop Statements

repeat Statement

```
initial
begin
    clock = 1'b0;
    repeat (16)
    #5 clock = ~ clock;
end
```

Forever Loop

```
initial
begin
    clock = 1'b0;
    forever
    #10 clock = ~ clock;
end
```

While Loop

```
integer count;
initial
begin
    count = 0;
    while (count < 64)
    #5 count = count + 1;
end
```

For Loop

- An initial condition.
- An expression to check for the terminating condition.
- An assignment to change the control variable.

```
for (j = 0; j < 8; j = j + 1)
    begin
    // procedural statements go here
    end
```

Table 8.2
Verilog Operator Precedence

+ – ! ~ & ~& | ~| ^ ~^ ^~ (unary)	Highest precedence
**	
* / %	
+ – (binary)	
<< >> <<< >>>	
< <= > >=	
== != === !==	
& (binary)	
^ ^~ ~^ (binary)	
| (binary)	
&&	
||	
?: (conditional operator)	
{ } { { } }	Lowest precedence