Computer Aided Digital Design

- Hitesh Pranav

Unit - 1 Combinational Logic Design

Levels of abstraction for an electronic computing System

→ The critical technique for managing complexity is abstraction

Abstraction : hiding details which aren't important

→ System can be viewed from many different levels of abstraction

→ 9 levels are there

Application Software ⇒ Uses the facilities provided by operating system to solve a problem for the user

↑

Operating System ⇒ Handles low level details like accessing hard drive or managing memory

↑

Architecture ⇒ Particular architecture can be implemented by one of many different microarchitectures with different price/performance/power trade-offs

↑

Microarchitecture ⇒ Links logic & architecture levels of abstraction

Involves combining logic elements to execute the instructions defined by architecture

↑

Logic ⇒ Complex Structures like adders/memories are built from digital circuits

↑

Digital Circuits ⇒ Logic gates restrict voltages to discrete ranges which are indicated by 0 & 1

↑

Analog Circuits ⇒ Devices are assembled to create components like amplifiers

↑

Devices ⇒ System is constructed from electronic devices called transistors

↑

Physics ⇒ Motion of electrons

(PDADLMAOA)

Physics
Devices
Analog circuits
Digital circuits
Logic
Microarchitecture
Architecture
Application software

Number System

→ Decimal Numbers

→ Referred as base 10

ex: $9742_{10} = \underbrace{9 \times 10^3}_{\text{Nine Thousands}} + \underbrace{7 \times 10^2}_{\text{Seven Hundreds}} + \underbrace{4 \times 10^1}_{\text{Four Tens}} + \underbrace{2 \times 10^0}_{\text{Two Ones}}$

→ Binary Numbers

→ Bits represents one of 2 values, 0 & 1 joined together to form binary numbers

→ 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048... column weights

ex: $1011_2 = \underbrace{1 \times 2^3}_{\text{one eight}} + \underbrace{0 \times 2^2}_{\text{no four}} + \underbrace{1 \times 2^1}_{\text{one two}} + \underbrace{1 \times 2^0}_{\text{one one}} = 11_{10}$

→ Decimal to Binary

ex: 84_{10} → Binary ?

Method 1

84_{10}	64 x 1
84 - 64 = 20	16 x 1
20 - 16 = 4	4 x 1
	1010100_2

Method 2

2 | 84		
2 | 42	—	0
2 | 21	—	0
2 | 10	—	1
2 | 5	—	0
2 | 2	—	1
2 | 1	—	0

$\Rightarrow 1010100_2$

Q. $(41.6875)_{10}$ = ?

A.

2 | 41		
2 | 20	—	1
2 | 10	—	0
2 | 5	—	0
2 | 2	—	1
2 | 1	—	0
0	—	1

0.6875	x 2 =	1.375	→ 1
0.375	x 2 =	0.75	→ 0
0.75	x 2 =	1.5	→ 1
0.5	x 2 =	1	→ 1

$(41.6875)_{10} = (101001.1011)_2$

→ Hexadecimal Numbers

→ Uses digits 0 to 9 and letters A - F

→ Columns in base 16 — 1, 16, 256, 4096 ...

0 - 0000	8 - 1000
1 - 0001	9 - 1001
2 - 0010	A - 1010
3 - 0011	B - 1011
4 - 0100	C - 1100
5 - 0101	D - 1101
6 - 0110	E - 1110
7 - 0111	F - 1111

→ Hexadecimal to Binary & Decimal

ex: $2ED_{16}$

$2_{16} = 0010$

$E_{16} = 1110$

$D_{16} = 1101$

→ $2ED_{16} = 001011101101_2$

$2ED_{16} = 2 \times 16^2 + E \times 16^1 + D \times 16^0 = 749_{10}$

Two — Two hundred fifty six's

Fourteen — Sixteens

Thirteen — Ones

https://www.rapidtables.com/convert/number/decimal-to-binary.html?x=41.6875

use for decimal to binary conversion checking

→ Binary to Hexadecimal

ex: 1111010_2 → Hexadecimal ?

Read from right

0111 1010

7 A ⇒ $7A_{16}$

$2^{10} = 1024 \Rightarrow 1$ Kilo

$2^{20} = 1$ million $\Rightarrow 1$ Mega

$2^{30} = 1$ billion $\Rightarrow 1$ Giga

1024 bytes = 1 KB → Most used

1024 bits = 1 Kb

Communication speed in bits/sec

→ Decimal to Hexadecimal

ex: 333_{10} → Hexadecimal ?

333	256 x 1
333 - 256 = 77	16 x 4
77 - 64 = 13	

$14D_{16}$

→ Bytes

→ Group of 8 bits is a byte

→ $2^8 = 256$ possibilities

→ Size of objects in computer usually measured in bytes

→ Nibble

→ Group of 4 bits is a nibble

→ $2^4 = 16$ possibilities

→ One hexadecimal digit stores 1 nibble

Two hexadecimal digits store 1 byte

→ Binary Addition

→ Simple rules

$0 + 0 = 0$

$1 + 0 = 1$

$0 + 1 = 1$

$1 + 1 = 0$ with 1 Carry

$1 + 1 + 1 = 1$ with 1 Carry

ex: $(0\ 1\ 1\ 1)_2 + (0\ 1\ 0\ 1)_2$

Carry 1 1 1

0 1 1 1

0 1 0 1

$(1\ 1\ 0\ 0)_2$ $\Rightarrow$ $7 + 5 = 12$

→ Addition with overflow

→ Addition is said to overflow if result is too big to fit in available digits

ex: Carry 1 1 1 $\quad 13 + 5 = 18$

0 1 1 0 1

\+ 0 1 0 1

1 0 0 1 0

→ Signed Binary Numbers

→ Several schemes exist to represent binary numbers but the 2 most widely used are

i) Sign / Magnitude Numbers

ii) Two's Complement

→ Sign / Magnitude Number

→ Uses most significant bit as the sign & remaining N-1 bits as magnitude

→ 0 indicates positive

1 indicates negative

ex: 5 & -5

↳ 0101 → 1101

→ Two's Complement

→ Identical to unsigned binary numbers except that most significant bit position has a weight of -2^{N-1} instead of 2^{N-1}

→ Process consists of inverting all of the bits in the number & then adding 1 to least significant bit position

ex: To get -2_{10}, invert 2_{10} and add 1

$2_{10} = 0010_2$

invert $\Rightarrow 1101_2$

add 1 $\Rightarrow 1110_2 = -2_{10}$

ex: $3_{10} - 5_{10}$

$= 3_{10} + (-5)_{10}$

$3_{10} = 0011$

$5_{10} = 0101 \Rightarrow -5_{10} = 1010 + 1 = 1011$

$3_{10} + (-5)_{10} =$ 0 0 1 1

\+ 1 0 1 1

1 1 1 0 $= -2_{10}$

ex: $-7_{10} + 7_{10}$

$= 1001_2 + 0111_2 = 10000_2$

5th bit is discarded, leaving correct 4 bit result 0000_2

→ N-bit two's complement numbers represent one of 2^N possible values but the values are split b/w +ve & -ve numbers

So, a 4 bit two's complement number represents 16 values : −8 to 7

in general $[-2^{N-1}, 2^{N-1}-1]$

→ Adding 2 N-bit +ve numbers or -ve numbers may cause overflow if result $> 2^{N-1}-1$ or result $< -2^{N-1}$

But adding a +ve number to a -ve number never causes overflow

ex: $4_{10} + 5_{10} = 0100_2 + 0101_2 = 1001_2 = 9_{10}$ or -7_{10}

The result overflows range of 4-bit +ve two's complement numbers, producing an incorrect negative result. If computation had been done using 5 or more bits, the result $01001_2 = 9_{10}$ would have been correct

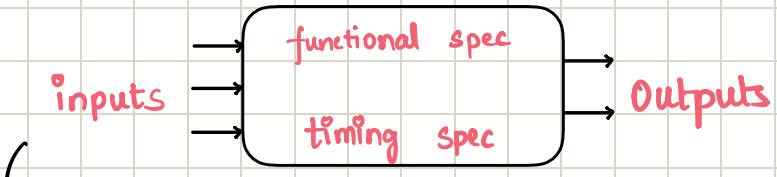
→ When a two's complement number is extended to more bits, the sign bit must be copied into the most significant bit positions. This is called sign extension

Comparision of Number System

System	Range
Unsigned	$[0, 2^N-1]$
Sign / Magnitude	$[-2^{N-1}+1, 2^{N-1}-1]$
Two's Complement	$[-2^{N-1}, 2^{N-1}-1]$

→ In digital electronics, a circuit is a network that processes discrete-valued variables

→

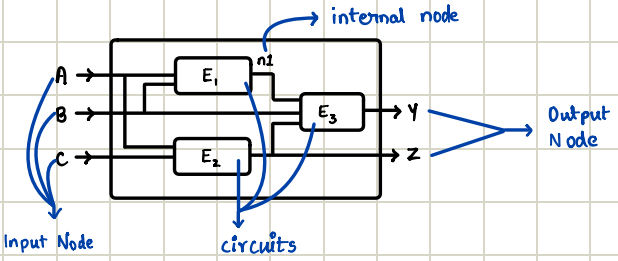


↳ A black box with one or more discrete-valued input terminals one or more discrete-valued output terminals a functional specification describing the relationship b/w inputs & outputs a timing specification describing the delay b/w inputs changing and outputs responding

→ Circuits have nodes & elements

An element itself is a circuit with inputs, outputs & specification

A node is a wire, whose voltage conveys a discrete-valued variable



→ Combinational Circuit

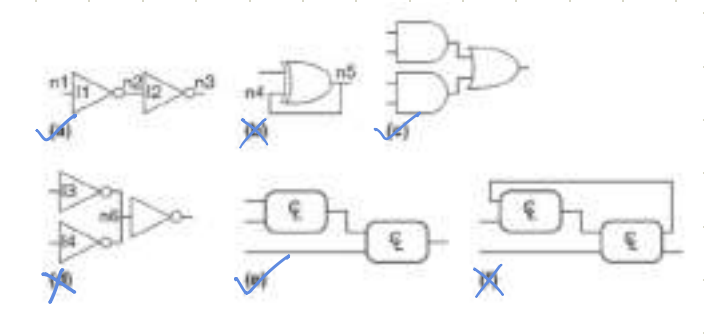
→ For a circuit to be combinational:

→ Every circuit element is itself combinational

→ Every node of circuit is designated as an input to the circuit or connects to exactly one output terminal of a circuit element

→ The circuit contains no cyclic paths

ex:



→ Multilevel Combinational Logic

→ Logic in sum-of-products form is called two-level logic because it consists of literals connected to a level of AND gates connected to a level of OR gates

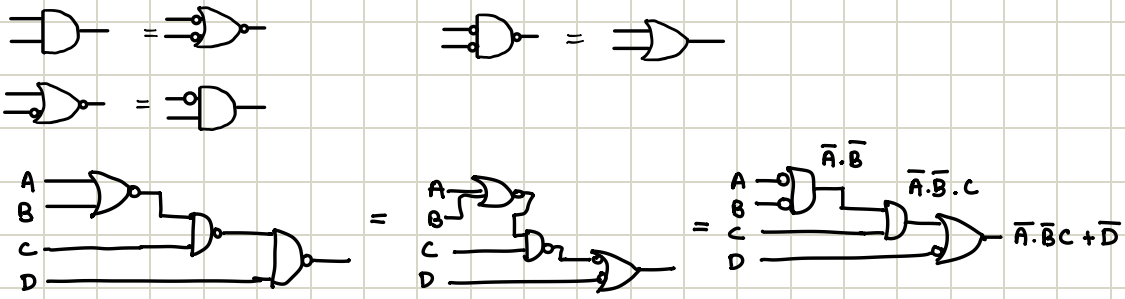
→ Hardware reduction:

→ Some logic functions require enormous amount of hardware when built using two-level logic

→ 8 input XOR using 2 level techniques requires 128 eight input AND gates and one 128-input OR gate for a two-level sum-of-products implementation but better to use a tree of two-input XOR gates

→ Bubble Pushing

→ Shortcut method for forming equivalent logic circuits, based on De-Morgan's Theorem



→ X's and Z's

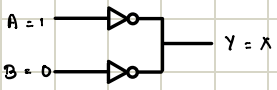
→ Boolean algebra is limited to 0's and 1's, however, real circuits can also have illegal & floating values, represented symbolically by X and Z

→ Illegal Value : X

→ X indicates that the circuit node has unknown or illegal value

→ Commonly appears if it is driven to both 0 and 1 at same time

ex:



→ This is called contention which is considered error & must be avoided
Usually voltage on a contented node is between 0 and V_{DD}
It is often, but not always, in forbidden zone

→ Contention also causes large amounts of power to be flow between the gates & circuit getting hot & possibly damaged

→ X values are also sometimes used by circuit simulators to indicate an uninitialized value and digital designers also use 'X' to indicate "don't care" values in truth tables

→ Floating value : Z

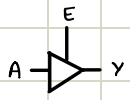
→ Z indicates node is being driven neither HIGH nor LOW
which means node is said to be floating, high impedance, or high Z

→ In reality, floating node might be 0, 1 or at some voltage in between, depending on the history of the system

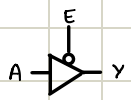
→ A floating node doesn't always mean there is an error in the circuit

→ One common way to produce a floating node is to forget to connect a voltage to a circuit input, or to assume that an unconnected input is same as an input with the value of 0.

→ Tristate buffer has 3 outputs : 0, 1, Z



A	E	Y
0	0	Z
1	0	Z
0	1	0
1	1	1



A	E	Y
0	0	0
1	0	1
0	1	Z
1	1	Z

→ Commonly used on busses connecting multiple chips

→ For a system, each chip can connect to a shared memory bus using tristate buffers but only one chip is allowed to assert its signal to drive a value onto the bus at a time.
Other chips must produce floating values to avoid contention.

ex:

Karnaugh Maps

→ They are graphical method for simplifying Boolean equations

3 Input function:

a) Truth Table

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

b) K-map

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

→ Grey Code

$Y = \bar{A}\bar{B}\bar{C} + \bar{A}B\bar{C}$

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

c) K-map showing minterms

Y: C \ AB	00	01	11	10
0	$\bar{A}\bar{B}\bar{C}$	$\bar{A}B\bar{C}$	$AB\bar{C}$	$A\bar{B}\bar{C}$
1	$\bar{A}\bar{B}C$	$\bar{A}BC$	ABC	$A\bar{B}C$

→ Simply circle all the rectangular block of 1's in the map, using fewest no. of circles and each circle should be as large as possible, Then read off the implicants that were circled

→ A boolean equation is minimized when it is written as sum of the fewest no. of prime implicants and each circle on K-map represents a prime implicant

→ Each circle must span a rectangular block that is a power of 2 (1, 2, 4, 8)

Q. Suppose we have the function $Y = F(A, B, C)$ with K-map shown, Minimise the equation using the K-map

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

A.

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

$A'B' + AC' + AB' = AC' + B'$

↓ prime implicants

Use for checking if your answer is correct or not → http://www.32x8.com/index.html

Q. $F(A, B, C) = \Sigma(0, 1, 5)$

A.

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

$\Rightarrow \bar{A}\bar{B} + \bar{B}C$

Q. $F(A, B, C) = \Sigma(0, 1, 5, 7)$

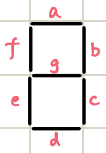
A.

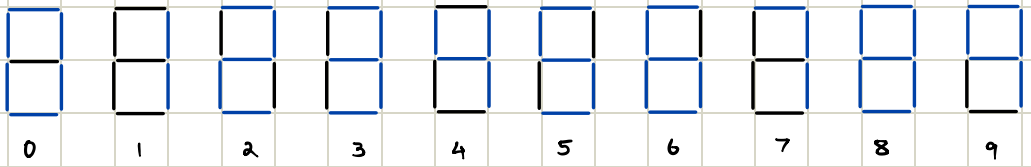
F: A \ BC	00	01	11	10
0	1	1	0	0
1	0	1	1	0

$= \bar{A}\bar{B} + AC$

ex: SEVEN SEGMENT DISPLAY DECODER

→ 7-segment decoder takes a 4-bit data input $D_{3:0}$ and produces 7 outputs to control LEDs to display digit from 0 to 9. The 7 outputs are often called segments 'a' to 'g', or $S_a - S_g$.

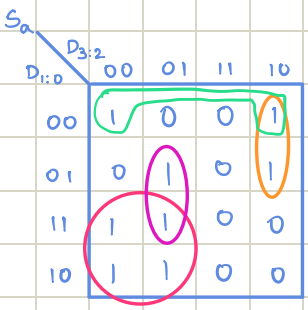


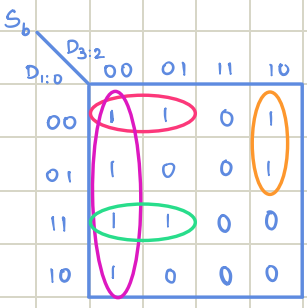


Q. Write truth table for outputs & use K-map to find boolean equation for outputs S_a & S_b (Assume illegal input values (10-15) produce blank readout)

A.

$D_{3:0}$ ($D_3D_2D_1D_0$) Input	S_a	S_b	S_c	S_d	S_e	S_f	S_g
0000 – 0	1	1	1	1	1	1	0
0001 – 1	0	1	1	0	0	0	0
0010 – 2	1	1	0	1	1	0	1
0011 – 3	1	1	1	1	0	0	1
0100 – 4	0	1	1	0	0	1	1
0101 – 5	1	0	1	1	0	1	1
0110 – 6	1	0	1	1	1	1	1
0111 – 7	1	1	1	0	0	0	0
1000 – 8	1	1	1	1	1	1	1
1001 – 9	1	1	1	0	0	1	1
others (10-15)	0	0	0	0	0	0	0





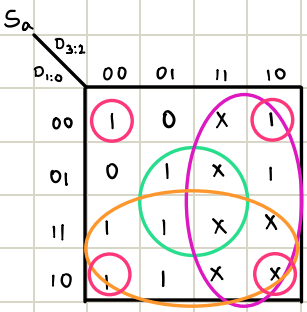
$S_a = \bar{D}_3 D_1 + D_3 \bar{D}_2 \bar{D}_1 + \bar{D}_2 \bar{D}_1 \bar{D}_0 + \bar{D}_3 D_2 D_1$

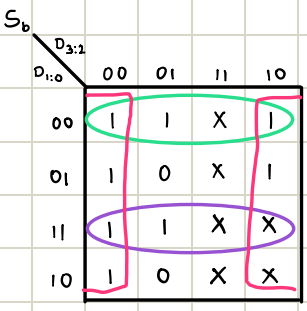
$S_b = \bar{D}_3 \bar{D}_2 + \bar{D}_3 \bar{D}_1 \bar{D}_0 + \bar{D}_3 D_1 D_0 + D_3 \bar{D}_2 \bar{D}_1$

→ Karnaugh maps : Don't cares

- → They are indicated by the symbol X, which means entry can be either 0 or 1
- → Don't cares also appear in truth table outputs where output value is unimportant or the corresponding input combination can never happen
- → They can be treated as either 0's or 1's allowing for logic minimization
- → They can be circled if they help & don't have to be circled if not helpful.

ex: SEVEN-SEGMENT DISPLAY DECODER WITH DON'T CARES





$S_a = D_1 + D_3 + D_2 D_0 + \bar{D}_2 \bar{D}_0$

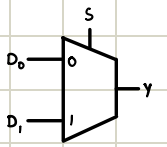
$S_b = \bar{D}_2 + \bar{D}_1 \bar{D}_0 + D_1 D_0$

→ Combinational Building Blocks

- → Combinational logic is often grouped into larger building blocks to build more complex systems
- → This is an application of the principle of abstraction, hiding the unnecessary gate-level details to emphasize the function of the building block

→ Multiplexers

- → They are the most commonly used combinational circuits
- → They choose an output from among several possible inputs based on the value of a select signal
- → 2:1 Multiplexer



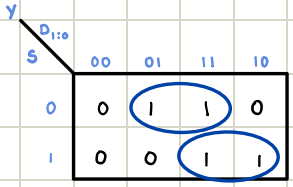
Select input (or) Control Signal

Controls what the multiplexer does

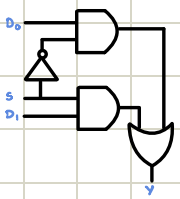
S	D_1	D_0	Y
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

Data Inputs

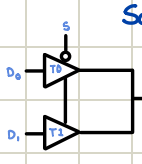
Output



$Y = D_0 \bar{S} + D_1 S$



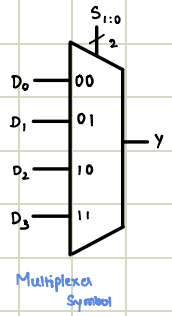
Schematic with tristate buffer



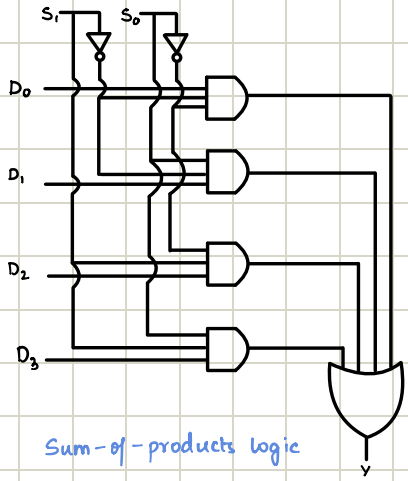
$Y = D_0 \bar{S} + D_1 S$

→ Wider Multiplexers

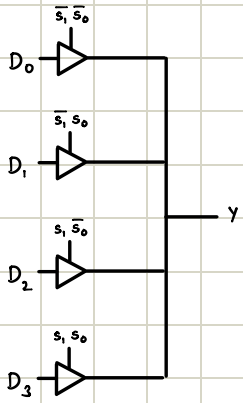
→ 4:1 Multiplexer has 4 data inputs and one output



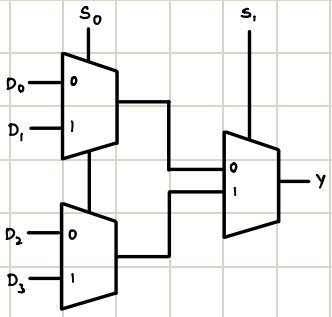
Multiplexer Symbol



Sum-of-products logic



Tristates



Hierarchical

→ Wider multiplexers, such as 8:1 & 16:1 multiplexers can also be built by expanding

→ In general, N:1 multiplexer needs $\log_2 N$ select lines

→ Multiplexer can be used as lookup tables to perform logic operations

S_0	S_1	Y
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

$\Rightarrow \overline{S_0}\overline{S_1}D_0 + \overline{S_0}S_1D_1 + S_0\overline{S_1}D_2 + S_0S_1D_3$

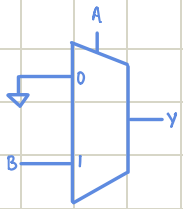
Q. Show the implementation of two-input AND and XOR functions with 2:1 multiplexers

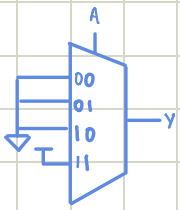
A.

$Y = AB$

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

A	Y
0	0
1	B



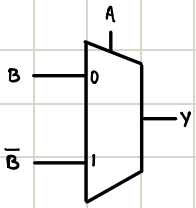


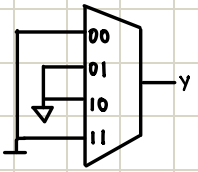
→ 2-input XOR Gate

$Y = A \oplus B$

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

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

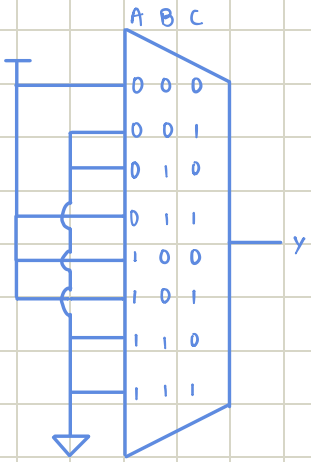




Q. Alyssa P. Hacker needs to implement the function $Y = A\bar{B} + \bar{B}\bar{C} + \bar{A}BC$ to finish her senior project, but when she looks in her lab kit, the only part she has left is an 8:1 MUX. How does she implement the function?

A.

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

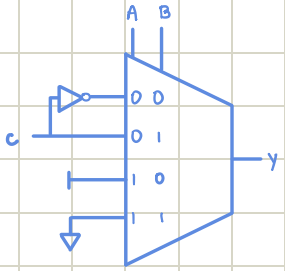


Q. Alyssa turns on her circuit one more time before the final presentation and blows up the 8:1 multiplexer. She begs her friends for spare parts and they gave her a 4:1 multiplexer and an inverter. Can she build her circuit with only these parts?

A.

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

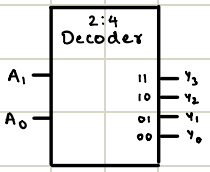
A	B	Y
0	0	$\bar{C}$
0	1	C
1	0	1
1	1	0



→ Decoders

→ A decoder has N inputs & 2^N outputs. It asserts exactly one of its output depending on input combination

→ Outputs are called one-hot because exactly one is 'hot' (HIGH) at a given time



A_1	A_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	1
0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	0	0	0

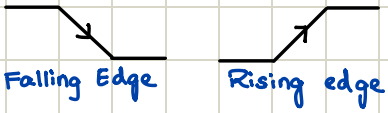
$$Y_0\bar{A_0}\bar{A_1} + Y_1A_0\bar{A_1} + Y_2\bar{A_0}A_1 + Y_3A_0A_1$$

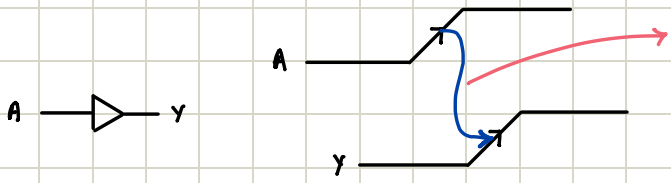
→ an $N:2^N$ decoder can be constructed from 2^N N-input AND gates

→ Timing

→ An output takes time to change in response to an input change

→ A timing diagram portrays the transient respone of the buffer circuit when an input changes



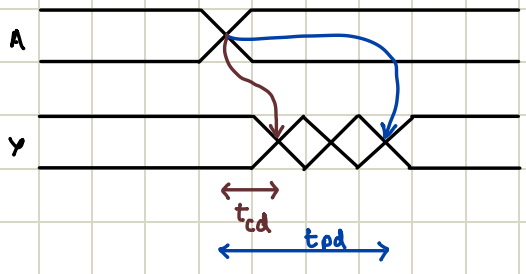


Blue arrow indicates rising edge of Y that is caused by rising edge of A
The delay is measured from 50% point of input signal (A) to 50% point of output signal (Y) b/w LOW & HIGH changes

→ Combinational logic is 2 types:

1) Propagational delay (t_{pd}) -
output final value
Max time from when an input changes until output(s) reach their final value

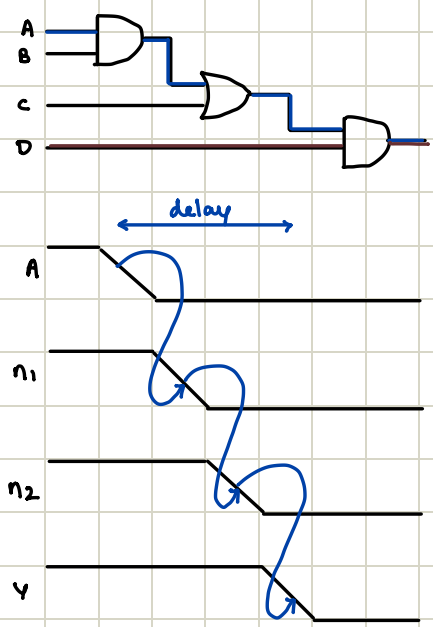
2) Contamination delay (t_{cd}) -
output change in value
Min time from when an input changes until output(s) until any output starts to change its value



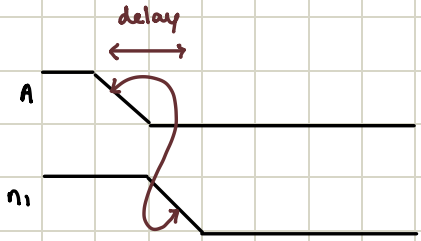
→ Critical path is the longest therefore the slowest path
→ Short path is the shortest therefore the longest path

→ Propagation delay of a combinational circuit is sum of propagation delays through each element on circuit path
→ Contamination delay of a combinational circuit is sum of contamination delays through each element on circuit path

ex:



- Critical Path
- Short Path



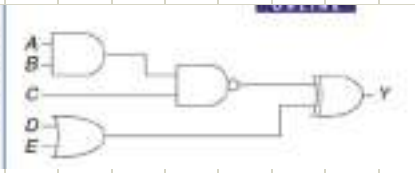
Short Path

$t_{cd} = t_{cd\_AND}$

Critical Path

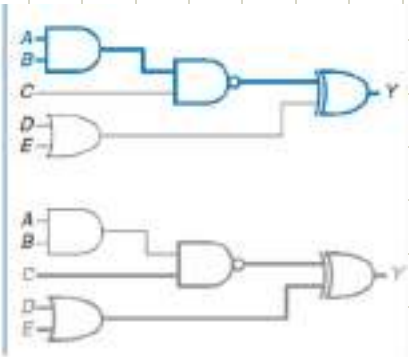
$t_{pd} = 2t_{pd\_AND} + t_{pd\_OR}$

Q. Ben Bitdiddle needs to find the propagation delay and contamination delay of the circuit shown in Figure. According to his data book, each gate has a propagation delay of 100 picoseconds (ps) and a contamination delay of 60 ps.



A. Solution:

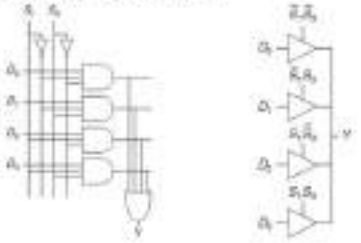
- Ben begins by finding the critical path and the shortest path through the circuit.
- Hence, t_{pd} is three times the propagation delay of a single gate, or 300 ps.
- There are only two gates in the shortest path, so t_{cd} is 120 ps.

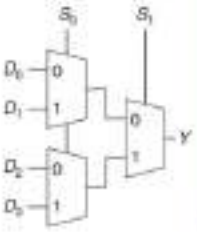


CONTROL-CRITICAL VS. DATA-CRITICAL

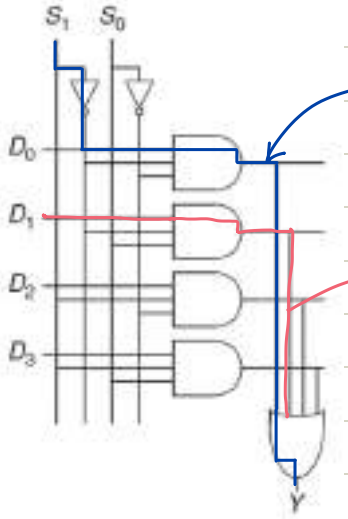
Q. Compare the worst-case timing of the three four-input multiplexer designs shown in Figure. Table lists the propagation delays for the components. What is the critical path for each design? Given your timing analysis, why might you choose one design over the other?

Gate	t_{pd} (ps)
NOT	30
2-input AND	60
3-input AND	80
4-input OR	90
tristate (A to Y)	50
tristate (enable to Y)	35





A.



$t_{pd\_sy} = t_{pd\_INV} + t_{pd\_AND3} + t_{pd\_OR4}$

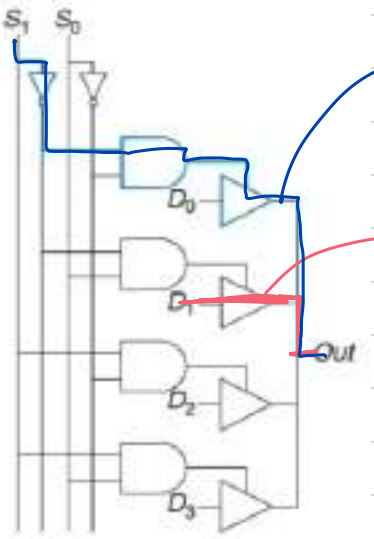
$= 30 + 80 + 90$

$= 200\,ps$

$t_{pd\_dy} = t_{pd\_AND3} + t_{pd\_OR4}$

$= 170\,ps$

$\max(t_{pd\_sy}, t_{pd\_dy}) = 200\,ps \rightarrow$ Worst of the two



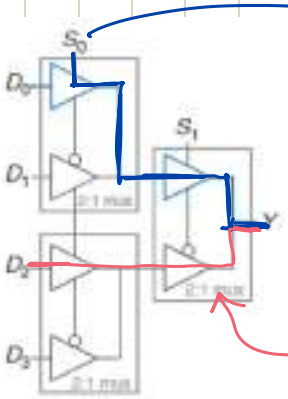
$t_{pd\_sy} = t_{pd\_INV} + t_{pd\_AND2} + t_{pd\_TRI\_sy}$

$= 30 + 60 + 35$

$= 125\,ps$

$t_{pd\_dy} = t_{pd\_TRI\_ay}$

$= 50\,ps$



$t_{pd\_s0y} = t_{pd\_TRI\_sy} + t_{pd\_TRI\_ay} = 85\,ps$

$t_{pd\_dy} = 2\,t_{pd\_TRI\_ay} = 100\,ps$

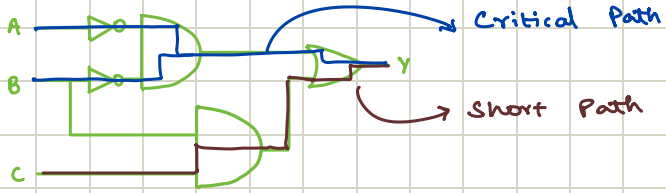
→ Best choice not only depends on critical path & input arrival times, but also power, cost, availability of parts etc.,

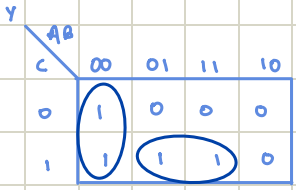
Glitches

→ Usually single input transition causes single output transition, but multiple output transitions are also possible, and are called Glitches (or) Hazards

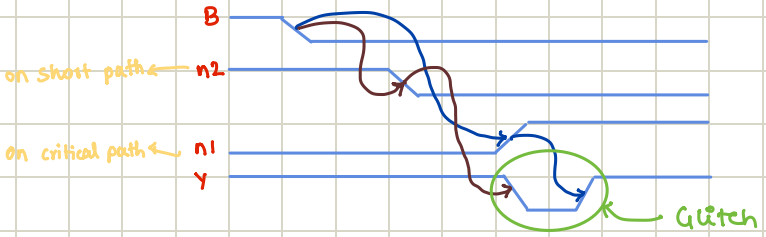
→ Glitches don't actually cause any harm, but we must recognize them

ex





when A & C = 0 and B transitions from 1 to 0, n2 falls before n1 rises



→ If the circuitry implementing one of the prime implicants turns off before the circuitry of other prime implicant can turn on, there is a glitch

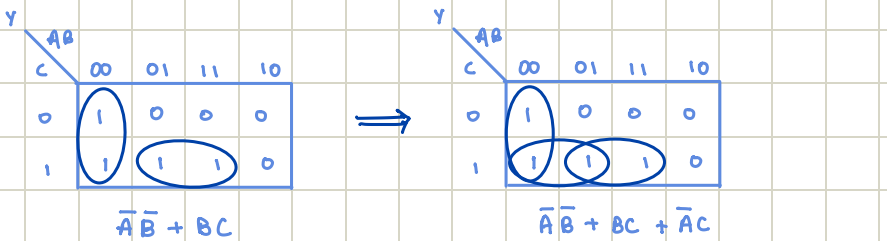
when n1 rises, Y returns to 1

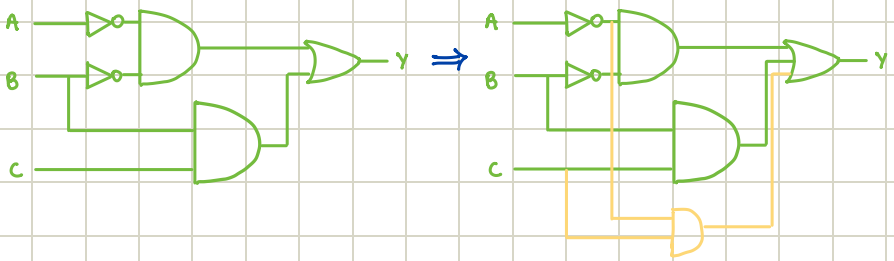
Y initially starts at 1, and ends at 1, but glitches to 0

We can avoid the glitches by adding another gate

In the previous example, to fix it, we add another covering prime implicant boundary adding a redundant term and additional hardware

(NOTE: Glitches cannot be FIXED by adding hardware)





Unit-2 Sequential Logic Design

→ Outputs of sequential logic depend on both current & previous inputs Hence, Sequential Logic has memory

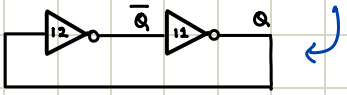
→ It might explicitly remember certain previous inputs or it might distill previous inputs into smaller ground of information called **state of system**

→ State of digital sequential circuit is set of bits called **state variables** containing info about past necessary to explain future behaviour

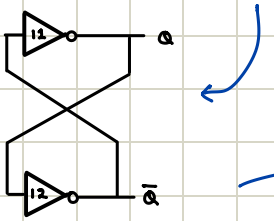
Bistable Element

→ A fundamental building block of memory with 2 stable states

→ Consists of pair of inverters connected in a loop

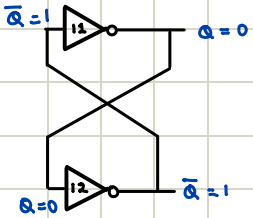


→ Inverters are cross-coupled (input of I1 = Output of I2 & vice-versa)

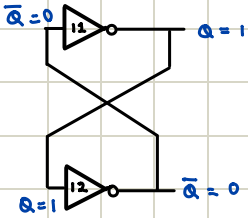


Circuit has no inputs, Only 2 outputs (Q & $\bar{Q}$)

→ Case (i): Q = 0



Case (ii): Q = 1



(A third state also is possible with both outputs approximately b/w 0 & 1 called as **metastable state**)

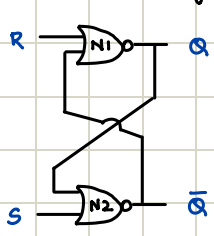
SR Latch

→ Simplest sequential circuit which is composed of 2 cross coupled NOR gates

→

S (Set)	R (Reset)	Q	$\bar{Q}$
0	0	Q_{prev}	$\bar{Q}_{prev}$
0	1	0	1
1	0	1	0
1	1	0	0

(Here $\bar{Q}$ is not the inverse of Q)



→ If either NOR gates see one TRUE (1) input, they produce FALSE (0) output

→ To SET means to make it TRUE (Q = 1 & $\bar{Q}$ = 0)

To RESET means to make it FALSE (Q = 0 & $\bar{Q}$ = 1)

Turning both off means the previous inputs are remembered

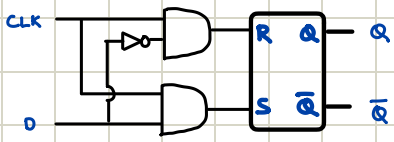
Turning both on simultaneously doesn't make sense, hence impossible

D Latch

→ D Latch solves the problem of SR Latch

→ Data Input (D) controls what next state should be

Clock Input (CLK) controls what state should change



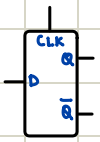
→

CLK	D	Q	$\bar{Q}$
0	X	Q_{prev}	$\bar{Q}_{prev}$
1	0	0	1
1	1	1	0

$D = S$

$\bar{D} = R$

(Also called transparent latch / level-sensitive latch)



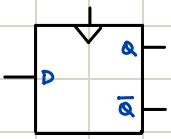
D Flip-Flop

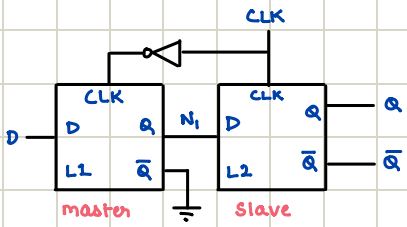
→ Built from 2 back-to-back D latches controlled by complementary clocks

→ First Latch, L1 ⇒ Master

Second Latch, L2 ⇒ Slave

Node between them ⇒ N1





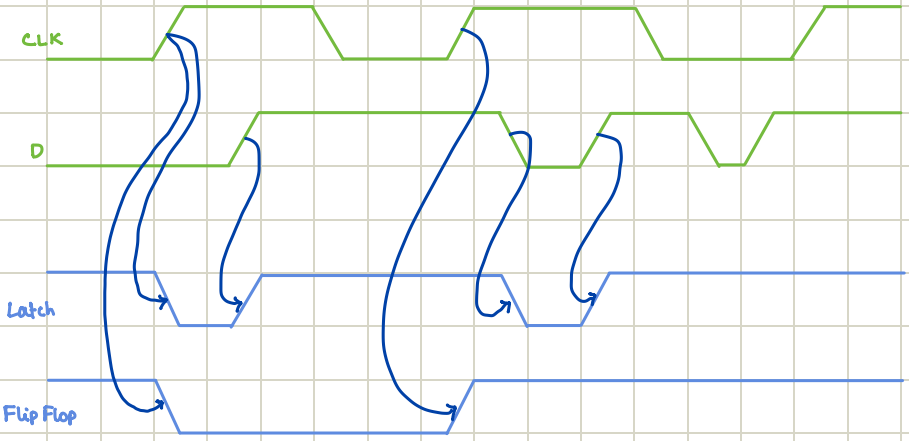
→

CLK	D	Q	$\bar{Q}$
0	X	Q_{prev}	$\bar{Q}_{prev}$
↓	0	0	1
↓	1	1	0

(Also called master-slave flip flop, edge-triggered flip flop or positive edge triggered flip flop)

Q. Ben Bitdiddle applies the D and CLK inputs shown to D Latch & D Flip Flop. Determine the output, Q, for both devices

A.



Q. How many transistors are needed to build D-Flip flop & D-latch ?

A. S-R Latch is made of 2 NOR Gates ⇒ 2 x 4 = 8 Transistors

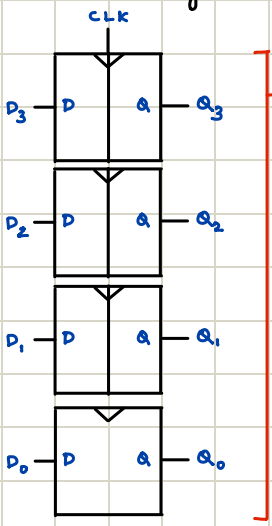
AND Gate is made of NAND and NOR Gate ⇒ 4 + 2 = 6 Transistors

So, D Latch is made of S-R Latch, 2 AND Gates, NOT Gate ⇒ 8 + (2×6) + 2 = 22

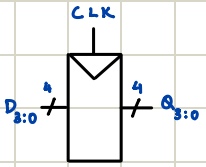
D Flip flop is made of 2 D Latches, 1 NOT gate ⇒ (2×22) + 2 = 46 Transistors

Register

→ N-bit register is a bank of N flip flops sharing common CLK input
↳ all bits update simultaneously



→ Schematic of 4-bit register



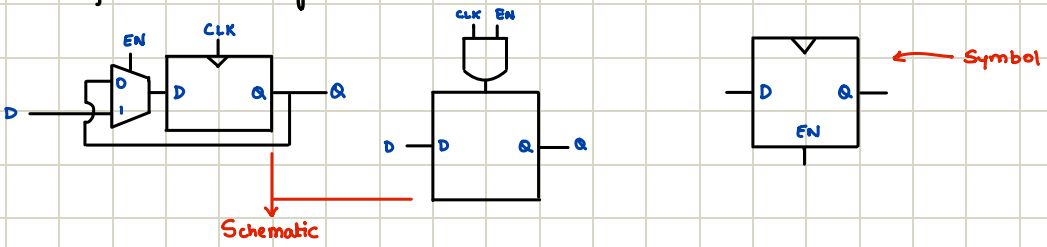
→ Symbol of 4 bit register

Enabled Flip Flop

→ It adds another input EN or ENABLED to determine if data is loaded on clock edge

→ When EN is true, behaves like ordinary D flip flop
When EN is false, ignores clock & retains its state

→ It is used to load a new value into a flip-flop only some time instead of every clock edge



Resettable Flip flop

→ It adds another input RESET

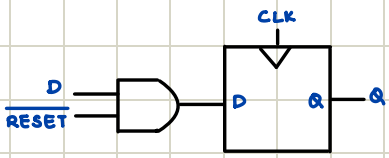
→ When RESET is false, behaves like ordinary D flip flop
When RESET is true, ignores D & resets output to 0

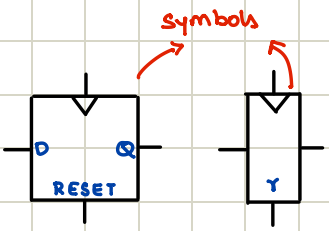
→ Used to force a known state (0) into all flip-flops when turned on

→ 2 Types : SYNCHRONOUS and ASYNCHRONOUS

SYNCHRONOUS: Resets themselves only on rising edge of CLK

ASYNCHRONOUS: Resets as RESET becomes TRUE independant of CLK

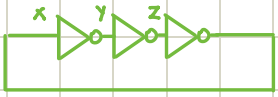




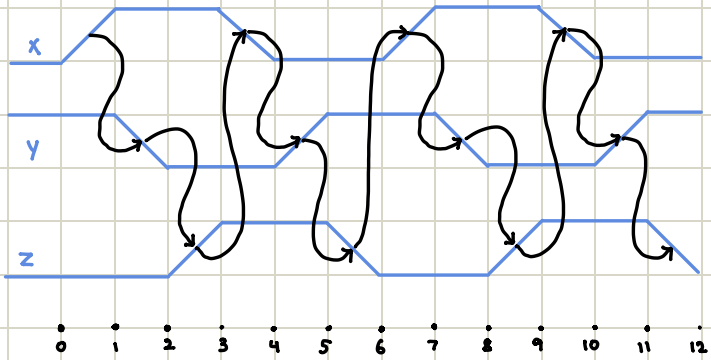
If $\overline{\text{RESET}}$ = FALSE ⇒ D is forced on flipflop input

If $\overline{\text{RESET}}$ = TRUE ⇒ D is passed on flipflop input

Q. Alyssa P Hacker encounters 3 misbegotten inverters who have tied themselves in a loop The 3rd inverter's output is fed back to 1st inverter. Each inverter has a propagation delay of 1ns. Determine what the circuit does



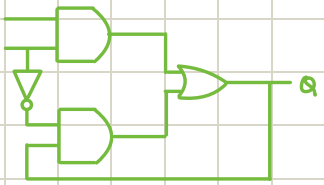
A. If initially $X = 0$, then $Y = 1$ & $Z = 0$ and $X = 1$ which is inconsistent
The circuit is unstable (or) astable



→ Pattern repeats every 6ns
→ This circuit is called **ring oscillator**

Q. Ben Bitdiddle designed a new D latch claiming to be better than the one in the figure because it has fewer gates.
The truth table is given, based on which he derived boolean equations. Q_{prev} is obtained by feeding the output Q.

CLK	D	Q_{prev}	Q
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1



$Q = CLK.D + \overline{CLK}.Q_{prev}$

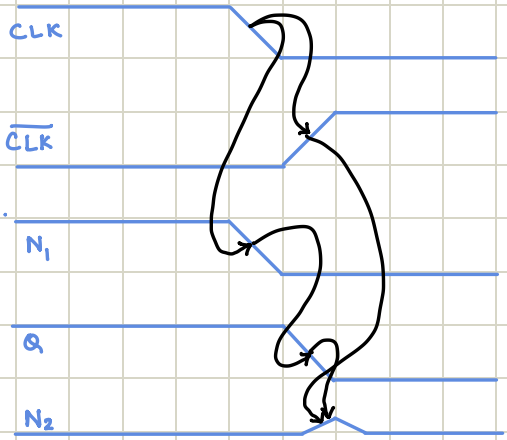


Figure shows that the circuit has race condition that causes it to fail when certain gates are slower than others

A. If $CLK = D = 1$, Latch is transparent & passes D through to make $Q = 1$
Now CLK falls, Latch should remember its old value, Keeping $Q = 1$
However, suppose delay through inverter from CLK to $\overline{CLK}$ is rather long compared to delays of AND and OR
Then N_1 & Q may both fall before $\overline{CLK}$ rises, Then N_2 will never rise & Q becomes stuck at 0.

Synchronous Sequential Circuits

→ Loops called cyclic paths are sequential circuits, in which outputs are fed back to inputs

→ Usually they have undesirable races or unstable behaviour & analyzing them is very time consuming

→ To avoid such problems, designers break cyclic paths by inserting registers which contain the state of system only at clock edge, so we say state is synchronized to clock

→ If clock is sufficiently slow, so that inputs to all registers settle before next clock edge, all races are eliminated

→ This way of adopting usage of registers is the proper definition of synchronous sequential circuits

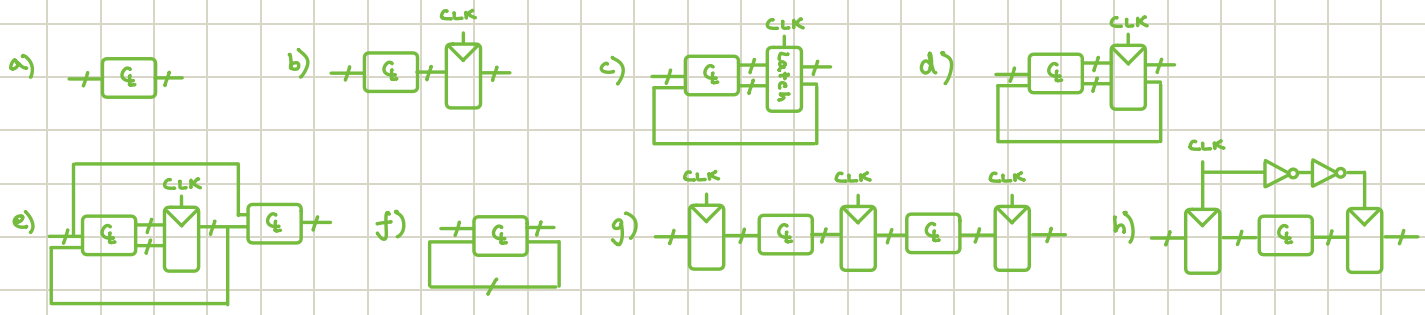
Rules of Synchronous Sequential Circuit

1) Every circuit element is either a register or combinational circuit
2) At least one circuit element is a register
3) All registers receive same clock signal
4) Every cyclic path contains atleast 1 register

→ Simplest synchronous sequential circuit = flip-flop → 1 input (D), 1 clock (CLK), 1 output (Q), 2 states (0,1)

→ 2 common synchronous sequential circuits ⇒ Finite State Machines & Pipelines

Q. Which of the following are synchronous sequential circuit



A. a) Combinational (No registers)
b) Simple Sequential
c) Neither Combinational nor Synchronous Sequential (Because Latch isn't register or combinational)
d) Synchronous Sequential (FSM)
e) Synchronous Sequential (FSM)
f) Neither Combinational nor Synchronous Sequential (Cyclic path without any register in path)
g) Synchronous Sequential (Pipeline)
h) Not Synchronous Sequential (2nd register receives different clock signal than 1st cuz of inverter delays)

Finite State Machines

→ A circuit with K registers can be in one of a finite number (2^K) of unique states

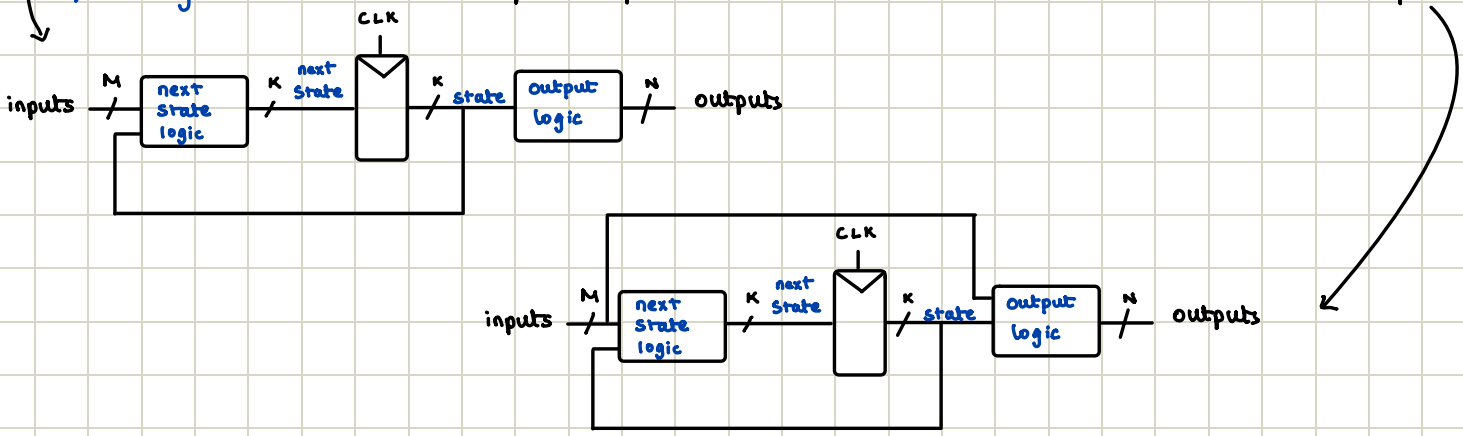
→ FSM has M inputs, N outputs & K bits of state
Also receives a clock & optionally, reset signal

→ FSM consists 2 blocks of combinational logic
next state logic
registers

→ On each clock edge, FSM advances to next state

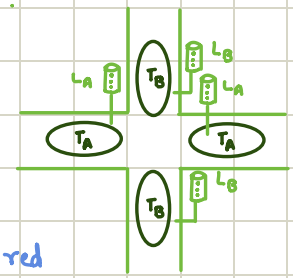
→ There are 2 classes of FSM's :

1) Moore Machines - Output depends only on current state of machines
2) Mealy Machines - Output depends on both current state & current inputs



Q. Consider a traffic light intersection, where a traffic light must be invented. The rule is that they can't go in perpendicular paths and can only go straight from the signal.

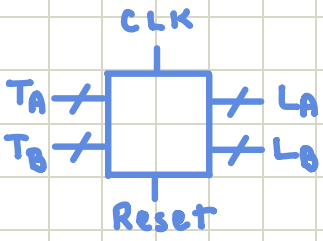
A. We can install 2 traffic sensors, T_A & T_B
They must indicate TRUE if traffic is present
FALSE if traffic is absent



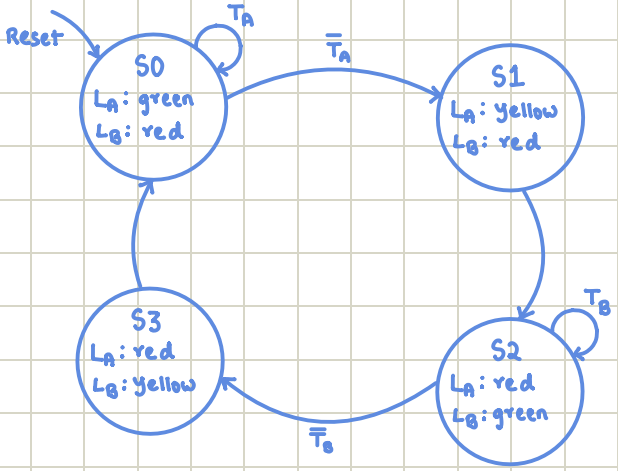
We can also install 2 traffic lights, L_A & L_B which receive input to tell whether it should be green, yellow, red

So FSM has 2 inputs → T_A & T_B
2 outputs → L_A & L_B

Let's provide a clock with 5-second period, lights change for every rising edge
Also a RESET button so a physical technician can control it to a known initial state
When system is RESET, lights are GREEN at L_A & RED at L_B
It is repeated till there's no traffic detected at T_A.
When T_A is FALSE, L_A is YELLOW, and then from S0 to S1.
After S1, it goes to S2, L_A is red, L_B is green,
T_B is passed till it is FALSE to which it goes to
S_3 where L_B is YELLOW, and finally returning to S0



Current State S	Inputs T_A	Inputs T_B	Next State S'
S0	0	X	S0
S0	1	X	S1
S1	X	X	S2
S2	X	0	S3
S2	X	1	S2
S3	X	X	S0



The State transition diagram is abstract,
It uses {S0, S1, S2, S3} and {red, yellow, green}
But to build real circuit, we must use binary encodings
Like

State	Binary Encoding
S0	00
S1	01
S2	10
S3	11

Output	Binary Encoding
Green	00
Yellow	01
Red	10

→ Updated State Transition State using binary encodings

Current State S_1	Current State S_0	Inputs T_A	Inputs T_B	Next State s_1'	Next State s_0'
0	0	1	X	0	0
0	0	0	X	0	1
0	1	X	X	1	0
1	0	X	0	1	1
1	0	X	1	1	0
1	1	X	X	0	0

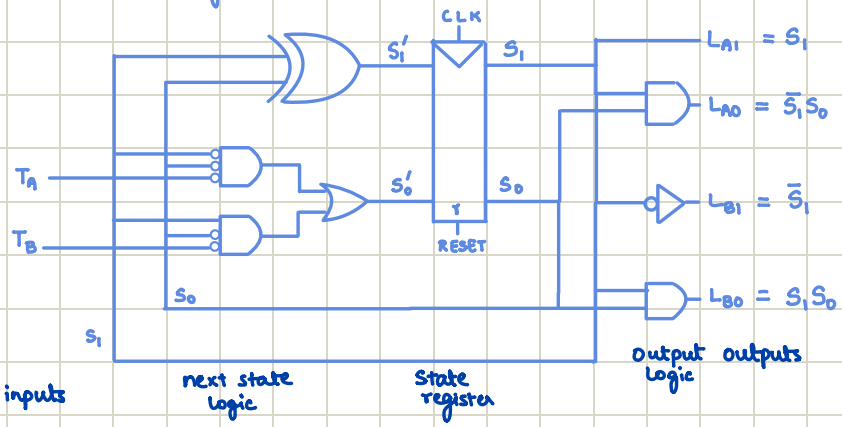
$$\Rightarrow S_1' = \overline{S}_1 S_0 + S_1 \overline{S}_0 \overline{T}_B + S_1 \overline{S}_0 T_B = S_1 \overline{S}_0 + \overline{S}_1 S_0 = S_1 \oplus S_0$$
$$S_0' = \overline{S}_1 \overline{S}_0 \overline{T}_A + S_1 \overline{S}_0 \overline{T}_B$$

→ Truth Table for Outputs

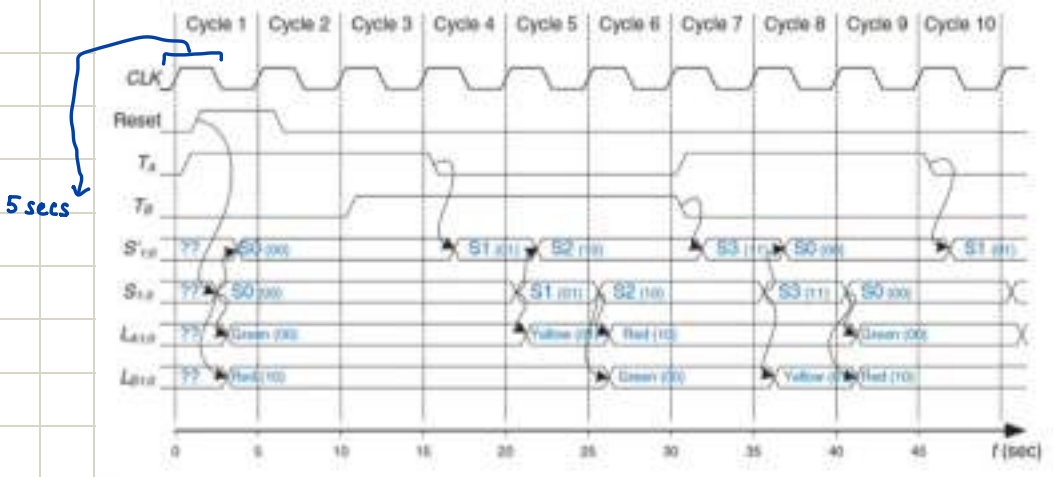
Current State S_1	Current State S_0	Outputs L_{A1}	Outputs L_{A0}	Outputs L_{B1}	Outputs L_{B0}
0	0	0	0	1	0
0	1	0	1	1	0
1	0	1	0	0	0
1	1	1	0	0	1

$$\Rightarrow L_{A1} = S_1 \overline{S}_0 + S_1 S_0 = S_1$$
$$L_{A0} = \overline{S}_1 S_0$$
$$L_{B1} = \overline{S}_1 \overline{S}_0 + \overline{S}_1 S_0 = \overline{S}_1$$
$$L_{B0} = S_1 S_0$$

→ Diagramic Representation



→ Timing Diagram



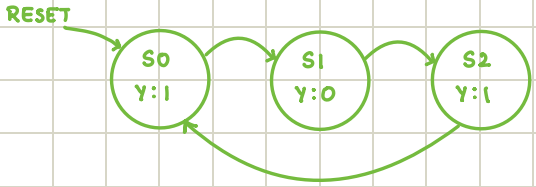
State Encodings

→ Like the previous example where we used arbitrary encodings, different choices would give different outputs.
But we can get the set of possible encodings & select a reasonable one by using Computer-Aided Design

→ There are 2 choices of state encoding

- **Binary Encoding**: Each state is represented as binary number
- **One-hot Encoding**: Seperate bit of state is used for each state & One bit is hot or TRUE at any time hence, the name
  - FSM with 3 states can be stored as 001, 010, 100
  - but this requires more flip-flops
  - but output logic is simpler, hence less gates

Q. Design a counter that doesn't take any input, but has 1 output, such that it counts upto 3 and resets with binary & one-hot encoding



A

Current State	Next State
S0	S1
S1	S2
S2	S0

⇒

Current State	Output
S0	1
S1	0
S2	0

⇒

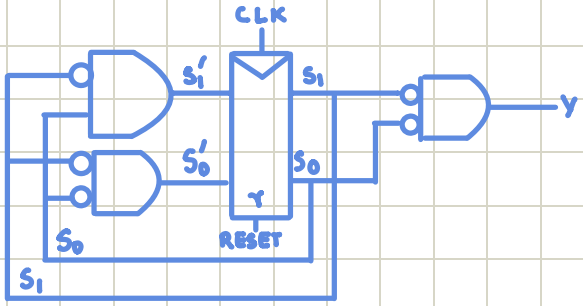
State	OHC S_2	OHC S_1	OHC S_0	BC S_1	BC S_0
S_0	0	0	1	0	0
S_1	0	1	0	0	1
S_2	1	0	0	1	0

S_2'	S_1'	S_0'
0	1	0
1	0	0
0	0	1

$S_2' = S_1$
$S_1' = S_0$
$S_0' = S_2$
$Y = S_0$

$S_1' = \bar{S}_1 S_0$
$S_0' = \bar{S}_1 \bar{S}_0$
$Y = \bar{S}_1 \bar{S}_0$

S_1'	S_0'
0	1
1	0
0	0



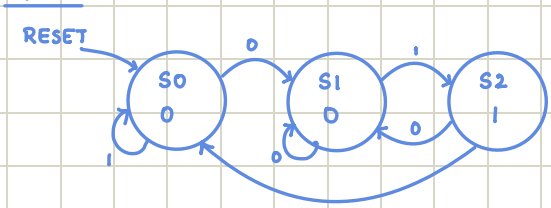
Moore & Mealy Machines

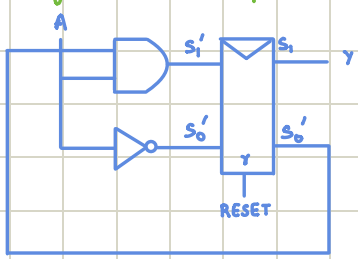
→ In state transition diagrams for Moore machines, outputs labeled in circles
→ In state transition diagrams for Mealy machines, outputs labeled on arcs

Q. Alyssa P Hacker owns a pet robotic snail with an FSM brain. The snail crawls from left to right along a paper tape containing a sequence of 0's & 1's. On each clock cycle, the snail crawls to the next bit. The snail smiles when the last 2 bits that it has crawled over are 01. Design the FSM to compute when the snail should smile. The input A is the bit underneath the snail's antennae. The output Y is TRUE when the snail smiles. Compare Moore & Mealy state machine designs. Sketch a timing diagram for each machine showing the input, states, & output as Alyssa's snail crawls along the sequence 01000110111:

A.

Moore:



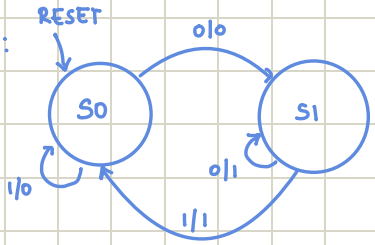


Current State S	Input A	Next State S′
S0	0	S1
S0	1	S0
S1	0	S1
S1	1	S2
S2	0	S1
S2	1	S0

In terms of state encodings,

Current State S_1	S_0	Input A	Next State S_1'	S_0'
0	0	0	0	1
0	0	1	0	0
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	0	0

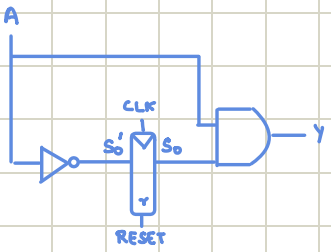
Mealy :

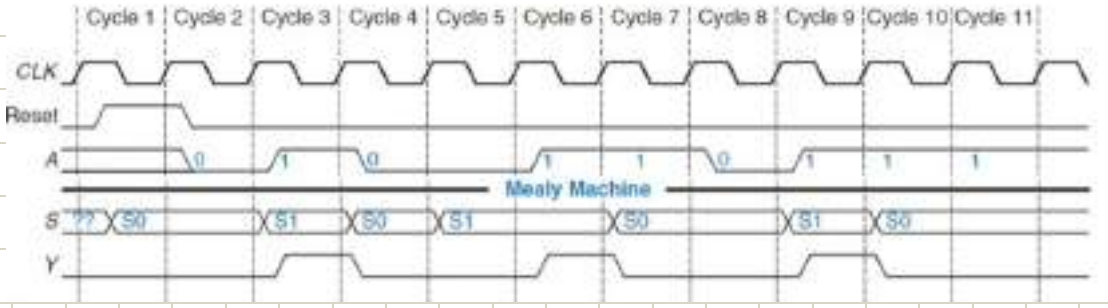


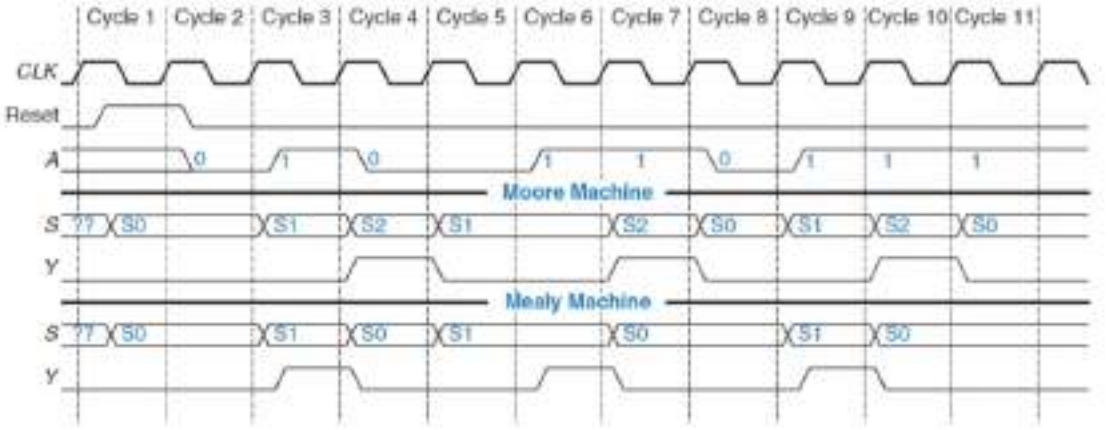
Current State S	Input A	Next State S′	Output Y
S0	0	S1	0
S0	1	S0	0
S1	0	S1	0
S1	1	S0	1

In terms of state encodings

Current State S	Input A	Next State S′	Output Y
S0	0	S1	0
S0	1	S0	0
S1	0	S1	0
S1	1	S0	1







Procedure to design FSM

1) Identify inputs & outputs
2) Sketch state transition diagram
3) For Moore machine, write state transition table & write output table
4) For Mealy machine, write combined state transition & output table
5) Select state encodings
6) Write boolean equations for the next state & output logic
7) Sketch the circuit schematic

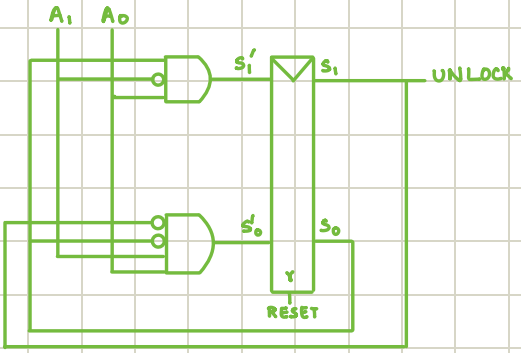
Deriving an FSM from a Schematic

→ Deriving the state transition diagram from a schematic follows nearly the reverse process of FSM design.

→ This process can be necessary
ex: reverse engineering somebody else's system

→ Procedure to derive an FSM from its circuit
1) Examine circuit by stating inputs, outputs & state bits
2) Write next state & output tables
3) Create next state & output tables
4) Reduce the next stable table to eliminate unreachable states
5) Assign each valid state bit combination a name
6) Rewrite next state & output tables with state names
7) Draw state transition diagram
8) State in words what the FSM does

Q. Alyssa P Hacker arrives home, but her keypad lock has been rewired and her old code no longer works. A piece of paper is taped to it showing the circuit diagram in Figure. Alyssa thinks the circuit could be a finite state machine & decides to derive the state transition diagram to see if it helps her get in the door.



A. input is $A_{1:0}$, output is Unlock
Moore machine (Output depends on state bits)
Next state & output equations ⇒ $S_1' = S_0 \bar{A}_1 A_0$
$S_0' = \bar{S}_1 \bar{S}_0 A_1 A_0$
$Unlock = S_1$

Output Table ⇒

S_1	S_0	Output
0	0	0
0	1	0
1	0	1
1	1	1

Next state table ⇒

S_1	S_0	A_1	A_0	S_1'	S_0'
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	0	0
1	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	0	0
1	0	1	1	0	0
1	1	0	0	0	0
1	1	0	1	1	0
1	1	1	0	0	0
1	1	1	1	0	0

Since $S_{1:0} = 11$ can never come from previous states, it is unused & unreachable because of the FSM's design. So we can ignore it And for $S_{1:0} = 10$, next state is always $S_{1:0} = 00$, so don't cares are inserted

S_1	S_0	A_1	A_0	S_1'	S_0'
0	0	0	0	0	0
0	0	0	1	0	0
0	0	1	0	0	0
0	0	1	1	0	1
0	1	0	0	0	0
0	1	0	1	1	0
0	1	1	0	0	0
0	1	1	1	0	0
1	0	x	x	0	0

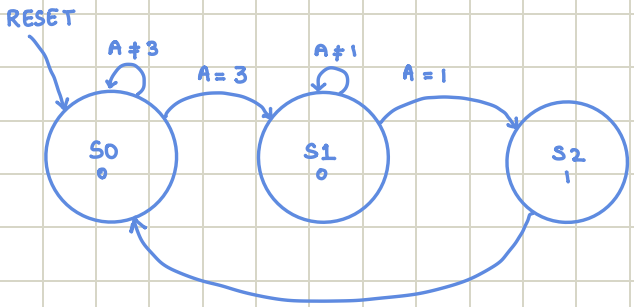
↳ Reduce Next State Table

⇒

S_1	S_0	O/P
0	0	0
0	1	0
1	0	1

↳ Reduced Output Table

⇒



↳ State Transition Diagram

Timing of Sequential Logic

→ Sampling is the process where a flip-flop copies input to output on rising edge of clock

→ A sequential element has an aperture time (some period of time) around the clock edge during which input must be stable for flip-flop to produce well-defined output.

→ The aperture of sequential element is defined by setup time & hold time before & after clock edge

Static discipline limits to use logic levels within forbidden zone

Dynamic discipline limits to use logic levels outside forbidden zone

↳ When clock rises, the output may start to change after clock-to-Q contamination delay & must settle to final value within clock-to-Q propagation delay.

But for circuit to operate properly, there must be stabilized at least for some setup time before rising edge of clock.

Even after rising edge of clock, input must remain stable for some hold time

So, Dynamic Discipline states "inputs of synchronous sequential circuit must be stable during setup & hold aperture time around clock edge

System Timing

→ Clock period (or) Cycle Time, T_c, is time b/w rising edges of a repetitive clock signal

→ $f_c = \frac{1}{T_c}$ ⇒ Clock frequency

Setup time Constraint

→ Minimum Clock period ⇒ $T_c \geq$ Clock to Q Propagation Delay (t_{pcq}) + Propagation Delay (t_{pd}) + setup delay (t_{setup})

$t_{pd} \leq T_c - (t_{pcq} + t_{setup})$

↳ Setup Time Constraint

Hold Time Constraint

→ Clock to Q Contamination Delay (t_{ccq}) + Contamination Delay (t_{cd}) ≥ Hold Delay (t_{hold})

⇒ $t_{cd} \geq t_{hold} - t_{ccq}$

↳ Hold Time Constraint

Q. Ben Bitdiddle designed the following circuit. According to the datasheets, Flip flops have t_{ccq} = 30ps, t_{pd} = 80ps, t_{setup} = 50ps, t_{hold} = 60ps. Each logic gate has t_{pd} = 40ps & t_{cd} = 25ps. Help Ben determine max clock frequency & whether any hold time violations could occur?

A.

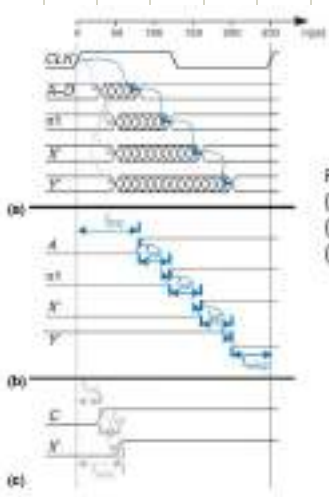
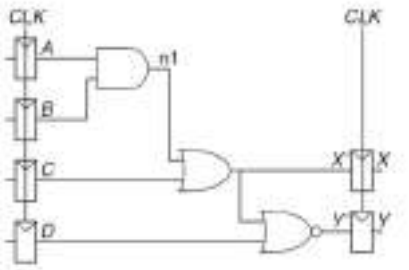


Figure: Timing diagram: (a) general case, (b) critical path, (c) short path



$T_c \geq t_{pcq} + (\text{Overall } t_{pd}) + t_{setup}$

$T_c \geq 80 + 3(40) + 50$

$T_c \geq 250 \text{ ps}$

$f_c = \frac{1}{T_c} = \frac{1}{250\,p} = 4 \times 10^9$

$t_{hold} \leq t_{ccq} + t_{cd} = 30 + 25 = 55 \text{ ps}$

but given t_{hold} = 60 ps, Hence there is hold time variation

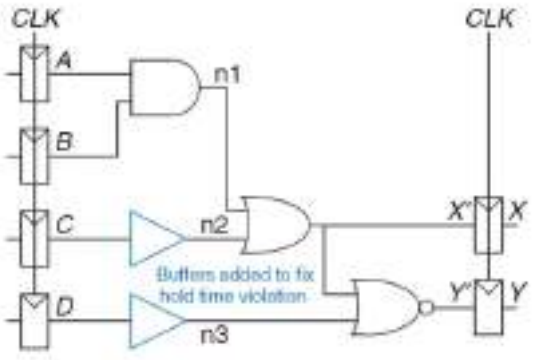
Q. How can the previous circuit be fixed?

A. We can add a buffer to increase gate delays

The critical path will be unaffected, So, f_c = 4GHz

Short path will change, $t_{hold} \leq t_{ccq} + 2t_{cd} = 30 + 2(25) = 80\text{ps}$

This is after 60ps hold time has elapsed, so circuit operates correctly now.



CADD - U - 3

Introduction

→ For higher productivity, designers used a lower level of abstraction by using a CAD Tool to produce optimized gates

→ Specifications are given in Hardware description Language

- System Verilog
- VHDL

Bitwise Operators

→ They act on single-bit signal or multi-bit busses

ex: input logic a;

ex: input logic [3:0] a;

→ INVERTERS

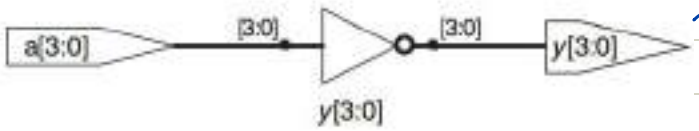
System verilog ⇒

```
module inv(input  logic [3:0] a,
           output logic [3:0] y);

  assign y = ~a;
endmodule
```

4 bit bus ⇒ Little Endian Order

3 2 1 0

↑ MSB ↑ LSB



→ Synthesized Circuit

→ Usually endianness is irrelevant. It matters only for operators like addition etc.,

→ LOGIC GATES

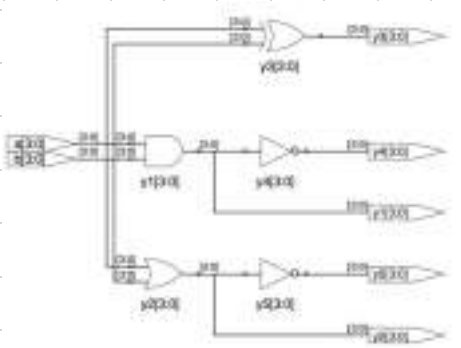
System verilog ⇒

```
module gates(input  logic [3:0] a, b,
             output logic [3:0] y1, y2,
                                y3, y4, y5);

  /* five different two-input logic
     gates acting on 4-bit busses */
  assign y1 = a & b;     // AND
  assign y2 = a | b;     // OR
  assign y3 = a ^ b;     // XOR
  assign y4 = ~(a & b);  // NAND
  assign y5 = ~(a | b);  // NOR
endmodule
```

→ &, |, ^, ~ are operators

a, b, y_1, y_2, y_3, y_4 are operands

Synthesized Circuit



Comments and White Spaces

→ Both the languages aren't worried about spaces, tabs, line breaks but for readability we use proper indenting

→ For both languages,

Multiline Comments ⇒ /* */ (Just like in C!)

Single line Comments ⇒ //

→ SystemVerilog is CASE-SENSITIVE

VHDL is NOT CASE-SENSITIVE

Reduction Operators

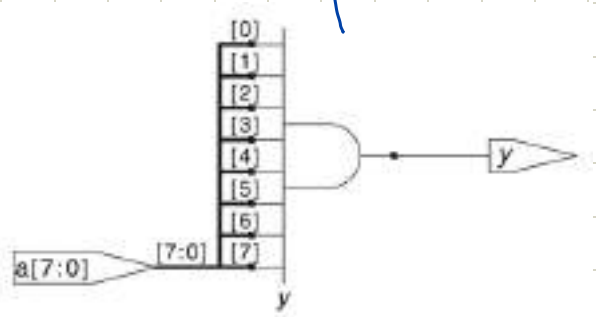
→ Implies a multiple-input gate acting on a single bus

→ EIGHT INPUT AND

System verilog ⇒

```
module and8(input  logic [7:0] a,
            output logic       y);

  assign y = &a;

  // &a is much easier to write than
  // assign y = a[7] & a[6] & a[5] & a[4] &
  //            a[3] & a[2] & a[1] & a[0];
endmodule
```

Synthesized Circuit



→ Analogous reduction operator exists for OR, XOR, NAND, NOR and XNOR

Internal Variables

→ Usually its better to write complex functions into intermediate steps

→ For a FULL ADDER,

$S\ (sum) = A \oplus B \oplus C$

$C\ (carry) = AB + AC_{in} + BC_{in}$

$P = A \oplus B$

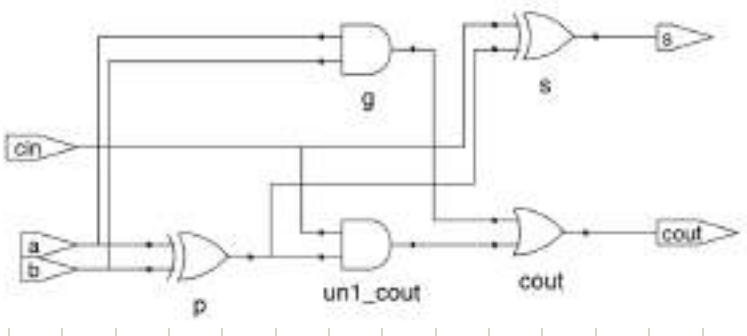
$G = AB$ (internal variables)

$\Rightarrow$ $S = P \oplus C_{in}$, $C = G + PC_{in}$

System verilog ⇒

```
module fulladder(input  logic a, b, cin,
                 output logic s, cout);

  logic p, g;

  assign p = a ^ b;
  assign g = a & b;

  assign s = p ^ cin;
  assign cout = g | (p & cin);
endmodule
```

Synthesized Circuit



Operator Precedence

→ System Verilog precedence order (highest to lowest)

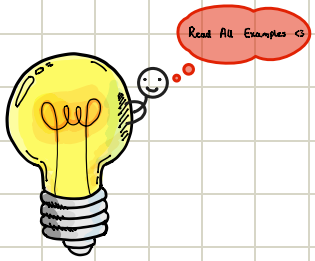
NOT (~)
MUL (*), DIV (/), MOD (%)
PLUS (+), MINUS (-)
Logical Left Shift (<<), Logical Right Shift (>>)
Arithmetic Left Shift (<<<), Arithmetic Right Shift (>>>)
Relative Comparision (< , <= , > , >=)
Equality Comparision (== , !=)
AND (&), NAND(~&)
XOR (^), XNOR (~^)
OR (|), NOR (~|)
Conditional (?:)

Numbers

→ Numbers can be binary (2), octal (8), decimal (10) or hexadecimal (16)

→ The format for declaring constants in System Verilog is N'B Value (Size, Base, Value)

ex: 3'b110	⇒	3 bits,	base 2,	Value = 6
8'b 1010_1111	⇒	8 bits,	base 2,	Value = 171 (underscores are ignored, to break values into small chunks)
8'b 11	⇒	8 bits,	base 2,	Value = 3
'b11	⇒	? bits,	base 2,	Value = 3 (Value Store = 000...0011)
3'd6	⇒	3 bits,	base 10,	Value = 6
6'o42	⇒	6 bits,	base 8,	Value = 34
8'h AB	⇒	8 bits,	base 16,	Value = 171
42	⇒	? bits,	base 10,	Value = 42

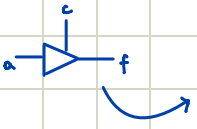


Z's and X's

→ Z indicates floating value (Used in Tristate Buffer)

X indicates invalid logic level

If bus is simultaneously driven to 0 & 1 by 2 enabled Tristate buffers resulting X, indicating contention



c	a	f
0	0	z
0	1	z
1	0	0
1	1	1

System verilog ⇒

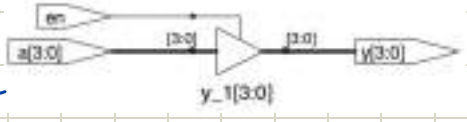
```
module tristate(input  logic [3:0] a,
                input  logic       en,
                output tri   [3:0] y);

  assign y = en ? a : 4'bz;
endmodule
```

z = floating value

Logic signals can have only single driver,
Tristate busses have multiple drivers, so declared as net
→ 1 driver is active & net takes that value (tri / trireg)
But when none are active, tri takes z, trireg retains previous value

System writes it as 4'bzzzz
→ padding

Synthesized Circuit



→ How SystemVerilog deals with X's & Z's

&		A			
		0	1	z	x
B	0	0	0	0	0
	1	0	1	x	x
	z	0	x	x	x
	x	0	x	x	x

Values of A

Values of B

Bit Swizzling

→ It refers to manipulating the bits of a signal or group of signals for purposes of combining, selecting or rearranging them.

→ We must operate on a subset of a bus (or) concatenate multiple signals together to form a wider bus

{n{}} is used to repeat a signal/value n times

System verilog ⇒

```
assign y = {c[2:1], {3{d[0]}}, c[0], 3'b101};
```

{} concatenates busses

assume c = 3'b110 & d = 1'b1

then, y = 11_111_0_101

c[2:1] 3{d[0]} c[0] 3'b101

y = 9'b111110101

Delays

→ Delays can be specified in HDL to control the timing during simulation

→ Delays help in

1) Predicting Circuit Behaviour (we can simulate & find how fast a circuit will operate)
2) Debugging (Help understand cause & effect in complex designs)

→ Delays are ignored during synthesis of circuit

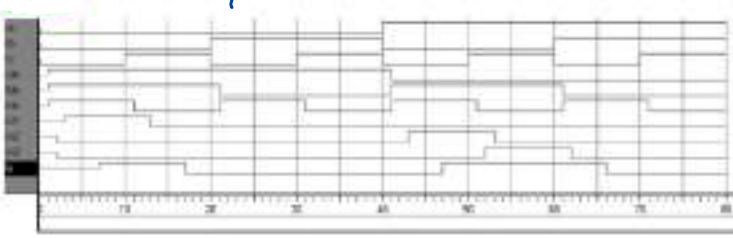
Actual Delays in hardware depends on t_{pd} and t_{cd} specifications, not on numbers in HDL code

ex: take $y = \bar{a}\bar{b}\bar{c} + a\bar{b}\bar{c} + a\bar{b}c$

System Verilog ⇒

```
`timescale 1ns/1ps
module example(input logic a, b, c,
               output logic y);

  logic ab, bb, cb, n1, n2, n3;

  assign #1 {ab, bb, cb} = ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```

Waveform with delays



timescale unit/precision

unit → 1ns, precision → 1ps

is used to indicate no. of units of delays

Truth Tables with Don't Cares

→ They are used for logic simplification

System Verilog ⇒

```
module priority_casez(input  logic [3:0] a,
                      output logic [3:0] y);
  always_comb
    casez(a)
      4'b1???: y <= 4'b1000;
      4'b01??: y <= 4'b0100;
      4'b001?: y <= 4'b0010;
      4'b0001: y <= 4'b0001;
      default: y <= 4'b0000;
    endcase
endmodule
```

Logic is evaluated whenever any of its input changes (always combinational)

casez statement acts like a case statement except that it also recognizes ? as 'Don't care'

So, by logic,

If a[3] = 1 ⇒ y = 4'b1000

else if a[3:2] = 01 ⇒ y = 4'b0100

else if a[3:1] = 001 ⇒ y = 4'b0010

else if a[3:0] = 0001 ⇒ y = 4'b0001

else y = 4'b0000

Blocking and Nonblocking Assignments

→ Blocking and Nonblocking Assignments Guidelines

1) Use always_ff @(posedge clk) and non-blocking assignments to model synchronous sequential logic

```
always_ff @(posedge clk)
  begin
    s1 <= d; // nonblocking
    q <= s1; // nonblocking
  end
```

(<= → non-blocking)

2) Use continuous assignments to model simple combinational logic

```
assign y = s ? d1 : d0;
```

3) Use always_comb & blocking assignments to model more complicated combinational logic where the always statement is helpful

```
always_comb
  begin
    p = a ^ b; // blocking
    g = a & b; // blocking
    s = p ^ cin;
    cout = g | (p & cin);
  end
```

(= → blocking)

4) Don't make assignments to the same signal in more than one always statement or continuous assignment statement

→ Combinational circuits are modelled better using blocking assignments

→ Sequential circuits are modelled better using non-blocking assignments

CORRECT MODELLING USING BLOCKING ASSIGNMENT

ex:

```
module fulladder(input  logic a, b, cin,
                 output logic s, cout);
  logic p, g;

  always_comb
    begin
      p = a ^ b;          // blocking
      g = a & b;          // blocking

      s = p ^ cin;        // blocking
      cout = g | (p & cin);  // blocking
    end
endmodule
```

Initially everything is 0

After some time, a changes to 1,

always gets trigerred

Order of Evaluation :

1) p ← 1 ⊕ 0 = 1
2) g ← 1 . 0 = 0
3) s ← 1 ⊕ 0 = 1
4) cout ← 0 + 1.0 = 0

NON-RECOMMENDED MODELLING USING NON-BLOCKING ASSIGNMENT

```
// nonblocking assignments (not recommended)
module fulladder(input  logic a, b, cin,
                 output logic s, cout);
  logic p, g;

  always_comb
    begin
      p <= a ^ b; // nonblocking
      g <= a & b; // nonblocking

      s <= p ^ cin;
      cout <= g | (p & cin);
    end
endmodule
```

In contrast if we use non-blocking for combinational,

Let a rise from 0 to 1, b & cin = 0

1) p ← 1 ⊕ 0 = 1
2) g ← 1 . 0 = 0
3) s ← 0 ⊕ 0 = 0
4) cout ← 0 + 0.0 = 0 → Which is NOT TRUE !

But p changes from 0 to 1, triggering always and evaluates second time,

5) p ← 1 ⊕ 0 = 1
6) g ← 1 . 0 = 0
7) s ← 1 ⊕ 0 = 1
8) cout ← 0 + 1.0 = 0 → Eventually reaching TRUE ans.

Even though same hardware, always is evaluated twice slowing down the simulation

Also, HDL produces wrong result if intermediate values are forgotten in sensitivity list.

CORRECT MODELLING USING NON-BLOCKING ASSIGNMENT

ex:

```
module sync(input  logic clk,
            input  logic d,
            output logic q);
  logic n1;

  always_ff @(posedge clk)
    begin
      n1 <= d; // nonblocking
      q <= n1; // nonblocking
    end
endmodule
```

On rising edge of clk, d is copied to n1 at same time as n1 is copied to q

Initially ⇒ d = 0, n1 = 1, q = 0

Later ⇒ n1 ← d = 0

q ← n1 = 1

NON-RECOMMENDED MODELLING USING BLOCKING ASSIGNMENT

```
// Bad implementation of a synchronizer using blocking
// assignments
module syncbad(input  logic clk,
               input  logic d,
               output logic q);
  logic n1;

  always_ff @(posedge clk)
    begin
      n1 = d; // blocking
      q = n1; // blocking
    end
endmodule
```

On rising edge of clk, d is copied to n1 but new value of n1 is copied to q

Initially ⇒ d = 0, n1 = 1, q = 0

Later ⇒ n1 ← d = 0

q ← n1 = 0

Finite State Machines

→ FSM's consists of state register & 2 blocks of combinational logic to compute the next state & output, given current state & input

```
module divideby3FSM(input  logic clk,
                    input  logic reset,
                    output logic y);
  typedef enum logic [1:0] {S0, S1, S2} statetype;
  statetype [1:0] state, nextstate;

  // state register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // next state logic
  always_comb
    case (state)
      S0:      nextstate <= S1;
      S1:      nextstate <= S2;
      S2:      nextstate <= S0;
      default: nextstate <= S0;
    endcase

  // output logic
  assign y = (state == S0);
endmodule
```

divide-by-3 FSM

typedef – statement defines statetype to be a two-bit logic value with 3 possibilities : S0, S1 or S2

state and nextstate are statetype signals

S0 : 00, S1 : 01, S2 : 10

↳ Encodings must be set by user

In the example, we can use 3-bit one-hot values

typedef enum logic [2:0] {S0 = 3'b001, S1 = 3'b010, S2 = 3'b100}

case is used to define State transition table

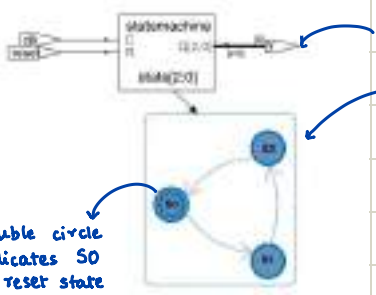
Output y is 1 when state is S0

The equality comparision a == b evaluates to 1 if a = b & 0 otherwise

The inequality comparision a != b does the inverse, evaluating to 1 if a != b, 0 otherwise

→ Synthesis tools produce a block-diagram & state transition diagram for state machines

It doesn't show logic gates, inputs or outputs on the arcs & states

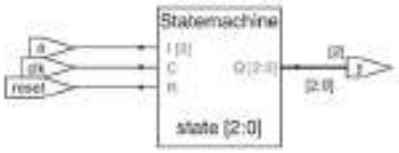


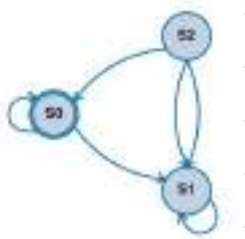
PATTERN RECOGNIZER MOORE FSM

```
module patternMoore(input  logic clk,
                    input  logic reset,
                    input  logic a,
                    output logic y);

  typedef enum logic [1:0] {S0, S1, S2} statetype;
  statetype state, nextstate;

  // state register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // next state logic
  always_comb
    case (state)
      S0: if (a) nextstate = S0;
          else   nextstate = S1;
      S1: if (a) nextstate = S2;
          else   nextstate = S1;
      S2: if (a) nextstate = S0;
          else   nextstate = S1;
      default:   nextstate = S0;
    endcase

  // output logic
  assign y = (state == S2);
endmodule
```

Non-blocking for sequential (state register)

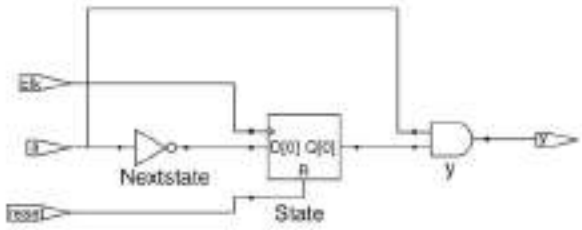
Blocking for combinational (next state)





PATTERN RECOGNIZER MEALY FSM

```
module patternMealy(input  logic clk,
                    input  logic reset,
                    input  logic a,
                    output logic y);

  typedef enum logic {S0, S1} statetype;
  statetype state, nextstate;

  // state register
  always_ff @(posedge clk, posedge reset)
    if (reset) state <= S0;
    else       state <= nextstate;

  // next state logic
  always_comb
    case (state)
      S0: if (a) nextstate = S0;
          else   nextstate = S1;
      S1: if (a) nextstate = S0;
          else   nextstate = S1;
      default:   nextstate = S0;
    endcase

  // output logic
  assign y = (a & state == S1);
endmodule
```



Data Types

→ Verilog primarily had 2 types : reg and wire
  ↳ Ultimately caused lot of confusion

→ In verilog, if a signal appears on LHS of = or <=, it must be declared reg
  others are declared as wire

```
module flop(input           clk,
            input     [3:0] d,
            output reg [3:0] q);
  always @(posedge clk)
    q <= d;
endmodule
```

→ By default, input & output ports are declared wire
→ Unless mentioned that they are reg

→ System Verilog introduced logic which is a synonym for reg & avoids misleading users about whether it is actually a flip-flop

→ System Verilog also relaxes rules on assign statements & hierarchial port instantations so, logic is used even outside always block where wire would have been required

→ But signals with multiple drivers are declared as net which allows System Verilog to generate error message instead of x when logic is accidentaly connected to multiple drivers

→ Most common type of net is called wire or tri
  ↳ Used for single drivers
  Used for multiple drivers

But we already use logic for single drivers, So, wire is obsolete

→ When net is driven to a certain value by 1 or more drivers, it takes up that value
When undriven, it floats (z)
When driven to different value (0, 1, x) by multiple drivers, it is in contention (x)

Net Type	No Driver	Conflicting Drivers
tri	z	x
trireg	previous value	x
triand	z	0 if there are any 0
trior	z	1 if there are any 1
tri0	0	x
tri1	1	x

How net resolves differently when driven/undriven by multiple sources.

Parameterized Modules

→ HDL permits usage of variable bit widths using parameterized modules.

ex: Parameterized N-Bit 2:1 Multiplexers

```
module mux2
  #(parameter width=8)
   (input  logic [width-1:0] d0, d1,
    input  logic             s,
    output logic [width-1:0] y);

   assign y=s ? d1 : d0;
endmodule
```

→ #(parameter variable = x) allows to define parameters

```
module mux4_8(input  logic [7:0] d0, d1, d2, d3,
              input  logic [1:0] s,
              output logic [7:0] y);

  logic [7:0] low, hi;

  mux2 lowmux(d0, d1, s[0], low);
  mux2 himux(d2, d3, s[0], hi);
  mux2 outmux(low, hi, s[1], y);
endmodule
```

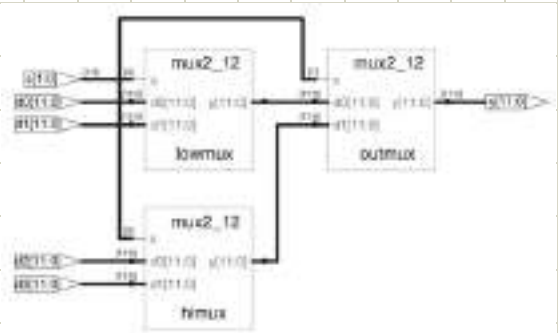
→ 8 bit 4:1 MUX instantiates 3 2:1 MUX using their default widths

```
module mux4_12(input logic  [11:0] d0, d1, d2, d3,
               input logic  [1:0]  s,
               output logic [11:0] y);

  logic [11:0] low, hi;

  mux2 #(12) lowmux(d0, d1, s[0], low);
  mux2 #(12) himux(d2, d3, s[0], hi);
  mux2 #(12) outmux(low, hi, s[1], y);
endmodule
```

→ But for 12-bit 4:1 MUX, we override the existing parameter value using #( ) before instance name

→ NOTE: # ⇒ Delays

#(...) ⇒ Parameter Defining & Overriding



SYNTHESIZED 12-BIT 4:1 MUX

ex: PARAMETERIZED N: 2^N DECODER

```
module decoder
  #(parameter N=3)
   (input  logic [N-1:0]    a,
    output logic [2**N-1:0] y);

  always_comb
    begin
      y=0;
      y[a]=1;
    end
endmodule
```

→ A larger DECODER would have to specify with cases

But parameterized code sets appropriate output bit to 1.
(Decoder uses blocking assignments to set all bits to 0, then changes to appropriate bit to 1)

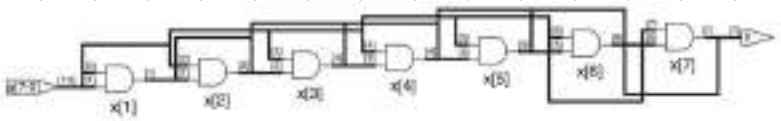
→ HDL also provides generate statement to produce variable amount of hardware depending on value of a parameter.

ex: PARAMETERIZED N-INPUT AND GATE

```
module andN
  #(parameter width=8)
   (input  logic [width-1:0] a,
    output logic             y);

  genvar i;
  logic [width-1:0] x;

  generate
    assign x[0]=a[0];
    for(i=1; i<width; i=i+1) begin: forloop
      assign x[i]=a[i] & x[i-1];
    end

  endgenerate

  assign y=x[width-1];
endmodule
```

→ for statement loops through i = 1, 2..., width-1 to produce many consecutive AND gates

→ The begin in generate for loop must be followed by ":" and an arbitrary label
↳ ex: forloop



Testbenches

→ HDL module that is used to test another module called device under test (DUT)

→ It contains statements to apply inputs to DUT & check that correct outputs are produced

Input & desired output patterns are called test vectors

→ Testbenches are simulated but not synthesizable

ex-

```
module testbench1();
  logic a, b, c, y;

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // apply inputs one at a time
  initial begin
    a=0; b=0; c=0; #10;
    c=1;           #10;
    b=1; c=0;      #10;
    c=1;           #10;
    a=1; b=0; c=0; #10;
    c=1;           #10;
    b=1; c=0;      #10;
    c=1;           #10;
  end
endmodule
```

→ Initiate which device to test

→ Must be used only for testbenches & no where else.

(Sillyfunction = $\bar{a}\bar{b}\bar{c} + a\bar{b}\bar{c} + a\bar{b}c$)

→ After initial begin starts executing statements in its body

So, first is 000, wait 10 units of time, then 001, again wait and so on till all inputs are applied

→ But checking for correct outputs is tedious & error prone. Its easy to determine the correct output now, but later on, even if minor changes are made, determining correct output becomes a hassle. Instead we can use self-checking testbench

```
module testbench2();
  logic a, b, c, y;

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // apply inputs one at a time
  // checking results
  initial begin
    a=0; b=0; c=0; #10;
    assert (y === 1) else $error("000 failed.");
    c=1;  #10;
    assert (y === 0) else $error("001 failed.");
    b=1; c=0;  #10;
    assert (y === 0) else $error("010 failed.");
    c=1;  #10;
    assert (y === 0) else $error("011 failed.");
    a=1; b=0; c=0;  #10;
    assert (y === 1) else $error("100 failed.");
    c=1;  #10;
    assert (y === 1) else $error("101 failed.");
    b=1; c=0;  #10;
    assert (y === 0) else $error("110 failed.");
    c=1;  #10;
    assert (y === 0) else $error("111 failed.");
  end
endmodule
```

→ assert verifies if a specified condition is true.

If not, executes else.

$error prints an error message describing assertion failure

assert is ignored during synthesis

→ == , != effective for signals without x and z

=== , !== effective for signals with x and z

→ Writing code for each test vector also becomes tedious, especially for modules that require large number of vectors. Better approach is to place test vectors in a seperate file

```
module testbench3();
  logic        clk, reset;
  logic        a, b, c, y, yexpected;
  logic [31:0] vectornum, errors;
  logic [3:0]  testvectors[10000:0];

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // generate clock
  always
    begin
      clk=1; #5; clk=0; #5;
    end

  // at start of test, load vectors
  // and pulse reset
  initial
    begin
      $readmemb("example.tv", testvectors);
      vectornum=0; errors=0;
      reset=1; #27; reset=0;
    end

  // apply test vectors on rising edge of clk
  always @(posedge clk)
    begin
      #1; {a, b, c, yexpected} = testvectors[vectornum];
    end

  // check results on falling edge of clk
  always @(negedge clk)
    if (~reset) begin // skip during reset
      if (y !== yexpected) begin  // check result
         $display("Error: inputs=%b", {a, b, c});
         $display(" outputs=%b (%b expected)", y, yexpected);
         errors=errors+1;
      end
      vectornum = vectornum+1;
      if (testvectors[vectornum] === 4'bx) begin
        $display("%d tests completed with %d errors",
                 vectornum, errors);
        $finish;
      end
    end
endmodule
```

always/process generates a clock without sensitivity list which is continuously reevaluated

It reads test vectors from text file & pulses reset for 2 cycles.

example.tv

```
000_1
001_0
010_0
011_0
100_1
101_1
110_0
111_0
```

Overkill for such a simple circuit 😭
But better for complex circuits

Parallelism

→ The speed of a system is characterized by latency & throughput of information moving through it.

- Throughput: no. of tokens that can be produced per unit time
- Latency: Time required for 1 token to pass through the system from start to end
- group of inputs that are produced to give group of outputs

Q. Ben Bitdiddle is throwing a milk & cookies party to celebrate the installation of his traffic light controller. It takes 5 minutes to roll cookies & place them on his tray. It then takes 15 minutes for the cookies to bake in the oven. Once the cookies are baked, he starts another tray. What is Ben's throughput & latency for a tray?

A. Latency = 5min + 15min = 20 min = 1/3 hour

Throughput = 1 tray / 20 min = 3 trays/hr

→ Throughput can be improved by processing several tokens at same time, this is called parallelism. It comes in 2 forms: spatial (parallelism) & temporal (pipelining)

- spatial (parallelism) → Avoided cuz its ambiguous
  - Multiple copies of the hardware are provided so multiple tasks can be done at same time
- temporal (pipelining)
  - Task is broken into stages & Multiple tasks can be spread across stages
  - Each task must pass through all stages but can go in any order, hence multiple tasks can overlap

Q. Ben Bitdiddle has 100 friends coming to his party & needs to bake cookies faster. He is considering using spatial and/or temporal parallelism

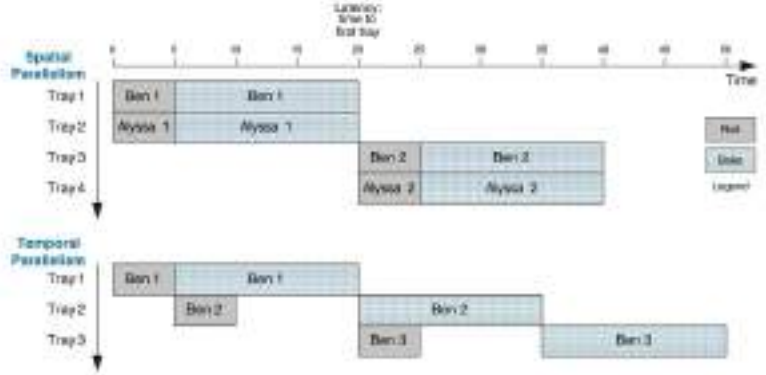
A. Spatial Parallelism: Ben asks Alyssa P Hacker to help & she has her own tray & oven

Latency = 20 mins

Throughput = 6 trays/hr

Temporal Parallelism: Ben get's another tray & starts rolling cookies on it after he puts the first tray in the oven

Throughput = 4 trays/hr



If Ben & Alyssa uses both techniques, they can bake 8 trays/hour

→ Consider a task with Latency L

If there wasn't any parallelism, throughput is $\frac{1}{L}$

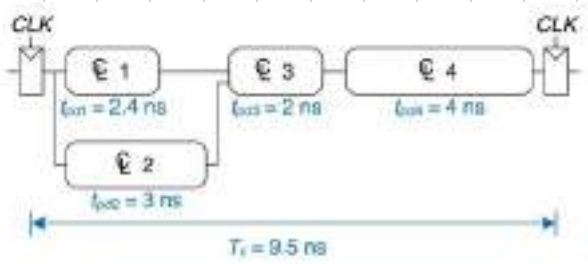
→ For spatially parallel system with N copies of hardware, throughput is $\frac{N}{L}$

→ For temporally parallel system, task ideally broken into N stages of equal length, throughput is also $\frac{N}{L}$ & only 1 copy is required

But finding N steps of equal length is difficult, So, if longest step has latency L_1, the pipelined throughput is $\frac{1}{L_1}$

Q. Assume a circuit with no pipelining. Critical path passes through 2,3,4.
Assume register has clock-to-Q propagation delay of 0.3ns & setup time 0.2ns

A.

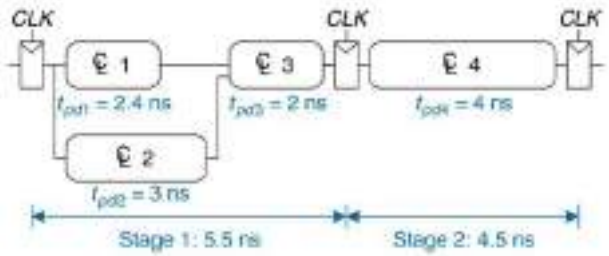


$T_c = 0.2 + 3 + 2 + 4 + 0.3 = 9.5ns \Rightarrow$ latency

Throughput $= \frac{1}{9.5n} = 105$ MHz

Q. Now assume same circuit partitioned into 2 stage pipeline by adding register

A



First stage ⇒ 0.3 + 3 + 2 + 0.2 = 5.5 ns

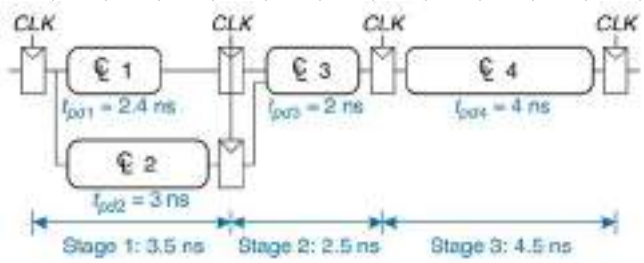
Second stage ⇒ 0.3 + 4 + 0.2 = 4.5 ns

Latency ⇒ 2 cycles ⇒ 2 × 5.5 = 11 ns

Throughput $= \frac{1}{5.5n} = 182$ MHz

Q. Now assume same circuit partitioned into 3 stage pipeline by adding register

A.



First stage ⇒ 0.2 + 3 + 0.3 = 3.5 ns

Second stage ⇒ 0.2 + 2 + 0.3 = 2.5 ns

Third stage ⇒ 0.2 + 4 + 0.3 = 4.5 ns

Latency ⇒ 3 cycles ⇒ 3 × 4.5 = 13.5 ns

Throughput $= \frac{1}{4.5n} = 222$ MHz

→ In conclusion, adding pipeline stage improves throughput in expense for latency

Unit-4 Digital Building Blocks

Arithmetic Circuits

→ They are the central building blocks of computers

→ They perform arithmetic functions like addition, subtraction, comparision, shift, multiplication, and division.

Adders

→ **Half adder :**

→ Here we add 1 bit binary numbers

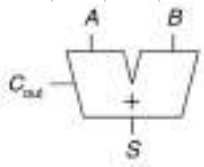
→ It has 2 inputs (A & B) and 2 outputs (S & C_{out})

S = Sum of A & B

C_{out} = Carry out

ex : 1 + 1 = 2 ⇒ can't be represented on binary as 2, so S = 0 & C = 1

→ Half adder is made of XOR and AND Gate



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

$S = A \oplus B$

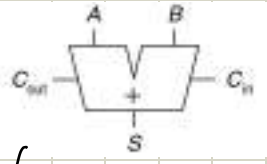
$C = A \cdot B$

→ **Full Adder :**

→ In a single bit half adder, C_{out} can be used to display second bit

But for multi bit addition, Half adder can't accept a C_{in} to perform addition in second column

So we use Full adder which accepts C_{in}



↳ 1 bit Full Adder

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

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

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

→ Carry Propagate Adder (CPA)

→ N-bit adder adds 2 N-bit inputs A & B and C_{in} to produce N-bit result S and carry out C_{out} which propagates into next bit.

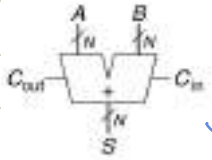
→ It is similar to full adder but A, B, S are busses rather than single bits

→ Common Implementations are :

i) Ripple-Carry adders

ii) Carry-Lookahead adders

iii) Prefix Adders



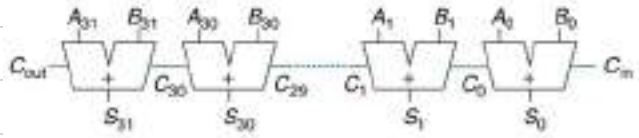
↳ CPA

→ Ripple Carry Adder

→ Ripple carry adder is built by chaining together N full adders

→ It carries ripples through the carry chain creating a delay

ex : For a 32-bit addition, S_{31} depends on C_{30}, S_{30} depends on C_{29} and so on



$t_{ripple} = N\, t_{FA}$ ↳ delay of each F.A

↳ N-bits

→ Carry-Lookahead Adder (CLA)

→ CLA solves the problem of ripple carry adder by dividing the adder into blocks & providing circuitry to quickly determine C_{out} as soon as C_{in} is known.

→ CLA's use generate (G) and propagate (P) signals that describe how a column or block determines the C_{out}.

→ Generate : Carry will definitely be produced in that column regardless of previous carries

Propagate : Carry is received from previous bit, it will pass through to the next bit.

→ i^{th} column of adder will generate carry if it produces C_{out} independant of C_{in} ($A_i = B_i = 1$)

i^{th} column of adder will propagate carry if it produces C_{out} when there is C_{in} (A_i or $B_i = 1$)

i^{th} column of adder will generate carry out C_i if it generates carry G_i or propagates carry in $P_i C_{i-1}$

So, $C_i = A_i B_i + (A_i \oplus B_i) C_{i-1} = G_i + P_i C_{i-1}$

→ Block will generate carry if it produces carry out independant of carry in to block G_i

Block will propagate carry if it produces carry out whenever there is carry in to block P_i

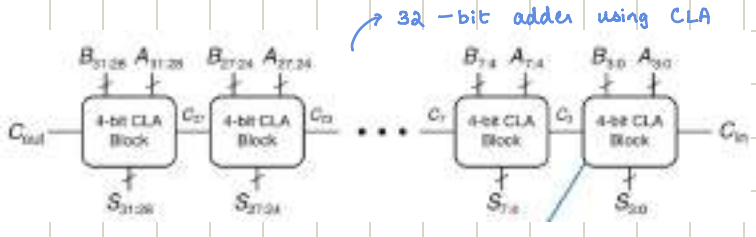
So, $G_{3:0} = G_3 + P_3 (G_2 + P_2 (G_1 + P_1 G_0))$

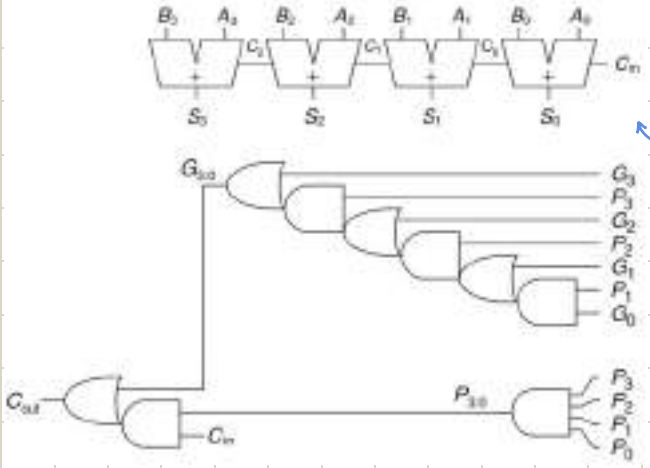
and, $P_{3:0} = P_3 P_2 P_1 P_0$

→ Carry out can be computed as

$C_i = G_{i:j} + P_{i:j} C_{j-1}$

↳ 32-bit adder using CLA





4-bit CLA Block

→ All CLA blocks generate & propogate signals simultaneously

→ N-bit adder divided into K-bit blocks has a delay

$$t_{CLA} = t_{pg} + t_{pg\_block} + \left(\frac{N}{K} - 1\right) t_{AND-OR} + K t_{FA}$$

- t_{pg}: delay of the individual generate/propagate gates to generate P_i & G_i (Single AND/OR gates)
- t_{pg_block}: Delay to find the generate/propagate signals $P_{i:j}$ & $G_{i:j}$ for K-bit block
- t_{AND-OR}: Delay from C_{in} to C_{out} through final AND/OR logic of K-bit CLA block

Q. Compare the delays of a 32-bit ripple-carry adder and a 32-bit carry-lookahead adder with 4-bit blocks. Assume that each 2-input gate delay is 100ps & full adder delay is 300 ps

A. $t_{ripple} = N t_{FA}$

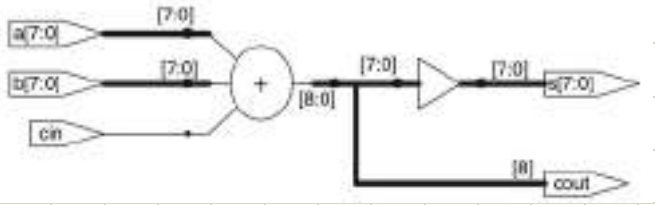
$= 32 \times 300 \times 10^{-12} = 9.6$ ns

$$t_{CLA} = t_{pg} + t_{pg\_block} + \left(\frac{N}{K} - 1\right) t_{AND-OR} + K t_{FA}$$

$$= \left(100 + 600 + \left(\frac{32}{4} - 1\right) 200 + (4 \times 300)\right) \times 10^{-12} = 3.3 \text{ ns}$$

For 4 input AND/OR gate, 3 AND and 3 OR Gates according to this diagram

```
module adder #(parameter N=8)
              (input  logic [N-1:0] a, b,
               input  logic         cin,
               output logic [N-1:0] s,
               output logic         cout);

  assign {cout, s} = a + b + cin;
endmodule
```



→ **Prefix Adder**

→ Prefix adder extend generate & propagate signals of carry-lookahead adder to perform faster addition

→ Uses P_i and G_i signals to calculate carries more quickly by combining columns (bits) into groups and doubling the size of the groups in each step (2, 4, 8...)

→ The circuit grows logarithmically in complexity

→ $C_i = G_{i-1} + P_{i-1} C_{-1}$

$S_i = (A_i \oplus B_i) \oplus C_{i-1}$

→ $G_{-1} = C_{in}$

$P_{-1} = 0$

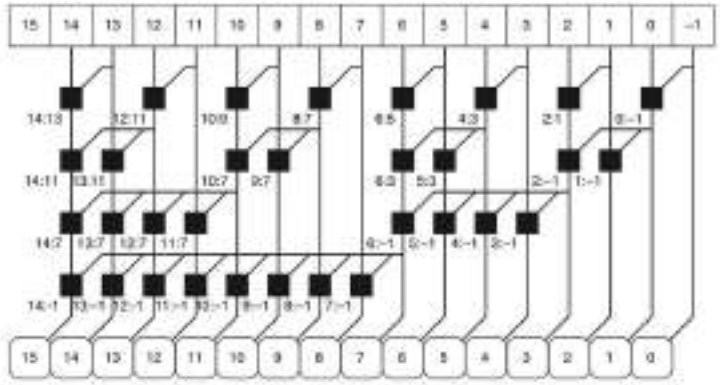
$C_0 = G_{-1}$

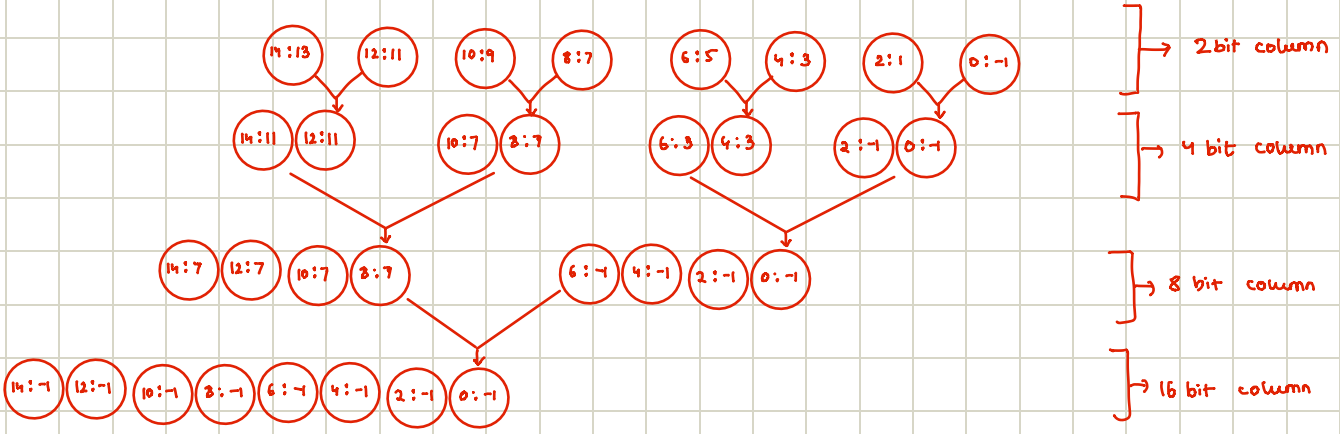
$S_i = (A_i \oplus B_i) \oplus G_{i-1:-1}$

$\Rightarrow G_{i:j} = G_{i:k} + P_{i:k} G_{k-1:j}$

$P_{i:j} = P_{i:k} P_{k-1:j}$

→





→ $t_{PA} = t_{pg} + \log_2 N \, (t_{pg\_prefix}) + t_{XOR}$

t_{pg_prefix} → delay of black prefix cell

Q. Compute delay of 32-bit prefix adder. Assume 2-input gate delay = 100ps

A. $t_{pg\_prefix} = 200\,ps$ (2 gate delays in each black prefix cell)

$t_{pg} = 100ps$

$N = 32 \Rightarrow \log_2 32 = 5$

$T_{XOR} = 100\,ps$

$t_{PA} = 100 + (5 \times 200) + 100 = 1.2\,ns$

Subtractors

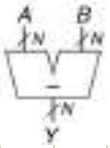
→ Adders can add +ve & -ve numbers using 2's complement number representation

→ $Y = A - B$

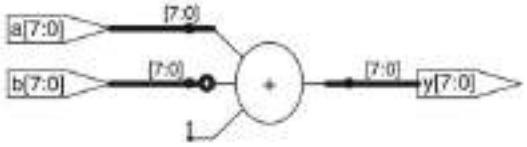
$-B = \overline{B} + 1$

$\Rightarrow Y = A + \overline{B} + 1$

This can be performed by carry propagate adder ($A + \overline{B}$ with $C_{in} = 1$)




```
module subtractor #(parameter N = 8)
                   (input  logic [N-1:0] a, b,
                    output logic [N-1:0] y);

  assign y = a - b;
endmodule
```



Comparators

→ Determines if 2 binary numbers are = or > or < to each other

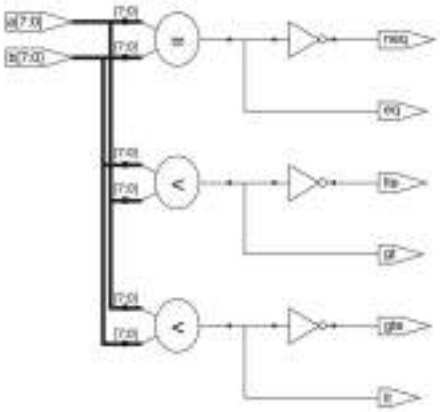
→ It receives 2 N-bit binary number A and B

→ 2 Types of Comparators

i) Equality Comparator - Produces single output indicating whether A == B by using XNOR gates to verify

ii) Magnitude Comparator - Produces one or more outputs indicating relative values of A & B by computing A - B and looking at sign (MSB) of result (1 = Negative) (0 = Positive)

but functions incorrectly upon overflow




```
module comparator #(parameter N = 8)
                   (input  logic [N-1:0] a, b,
                    output logic eq, neq, lt, lte, gt, gte);

  assign eq  = (a == b);
  assign neq = (a != b);
  assign lt  = (a < b);
  assign lte = (a <= b);
  assign gt  = (a > b);
  assign gte = (a >= b);
endmodule
```

Arithmetic Logic Unit

→ Combines variety of mathematical & logical operations into single unit

→ Typical ALU can perform addition, subtraction, AND, OR operations

→ ALU forms heart of most computer systems

→ ALU receives control signal F that specifies which function to perform

↳ usually in blue to distinguish from data

→ Some ALUs produce extra outputs called flags ↗ indicate info abt ALU

ex : Overflow flag - indicates adder overflowed

Zero flag - ALU output is 0

C flag - When adder produces C_{out} & ALU is performing addition/subtract

$F_{2:0}$	Function
000	A AND B
001	A OR B
010	A + B
011	not used
100	A AND $\overline{B}$
101	A OR $\overline{B}$
110	A − B
111	SLT ↗ Magnitude Comparision

Shifters and Registers

→ Move bits & multiply or divide by powers of 2

→ Shifters ⇒ Shifts binary number left / right by specified no. of pos^ns

→ Logical Shifter – Shifts number to left or right and fills empty spots with 0's

11001 LSR 2 ⇒ 00110 ⇒ >>

11001 LSL 2 ⇒ 00100 ⇒ <<

(Special Case of Multiplication by 2^N → 000011 << 4 = 110000 ↳ $3 \times 2^4 = 48$)

→ Arithmetic Shifter – Same as logical shifter but right shift fills MSB with copy of old MSB (Useful for multiplying & dividing signed numbers) ↳ Only for ASR

ASL = LSL (Fills with 0)

11001 ASR 2 ⇒ 11110 ⇒ >>>

(Special case of Division by 2^N → 11100 >>> 2 = 111111 ↳ $-4 / 2^2 = -1$)

11001 ASL 2 ⇒ 00100 ⇒ <<

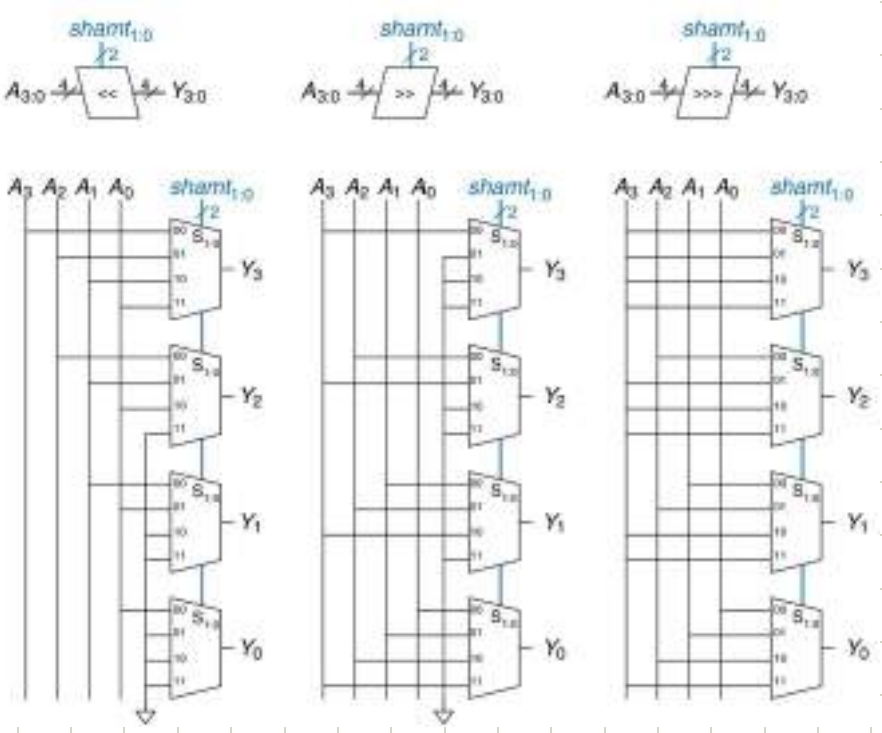
→ Rotators ⇒ Rotates numbers in a circle such that empty spots are filled with bits shifted off the other end

11001 ROR 2 ⇒ 01110

11001 ROL 2 ⇒ 00111

→ N-bit shifter built from N 'N:1 MUX'

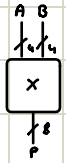
Input is shifted by 0 to N-1 bits depending on $\log_2 N$ – bit select lines



Multiplication

→ Similar to decimal multiplication but involves 1's & 0's

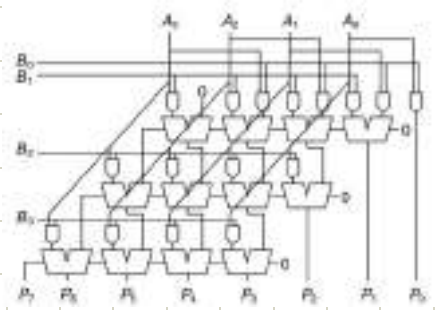
→ N × N multiplier multiplies 2 N-bit numbers and produces a 2N-bit result



$$
\begin{array}{cccccccc}
 & & & A_3 & A_2 & A_1 & A_0 & \\
 & & \times & B_2 & B_2 & B_1 & B_0 & \\
\hline
 & & & A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 & \\
 & & A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 & & \\
 & A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 & & & \\
A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 & & & & \\
\hline
P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
\end{array}
$$

→ N partial products

→ N-1 stages of 1-bit adders

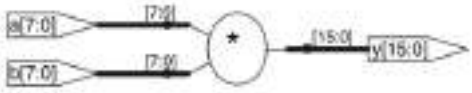


Q.

$$
\begin{array}{cccc}
 & 0 & 1 & 0 & 1 \\
\times & 0 & 1 & 1 & 1 \\
\hline
 & 0 & 1 & 0 & 1 \\
 & 0 & 1 & 0 & 1 \\
 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & \\
\hline
0 & 1 & 0 & 0 & 0 & 1 & 1
\end{array}
$$

A.

```
module multiplier #(parameter N = 8)
                  (input  logic [N-1:0]   a, b,
                   output logic [2*N-1:0] y);

  assign y = a * b;
endmodule
```

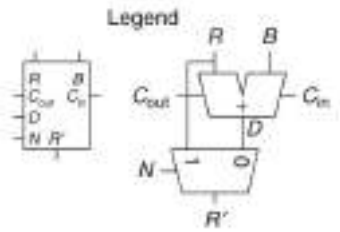


→ This operation also called MACs (used in DSP)

Division

→ Algorithm for N-bit unsigned in the range $[0, 2^{N-1}]$;

```
K = 0
for i = N-1 to 0
    R = {R' << 1, Ai}
    D = R < 0
    else
    R = R'
```



→ Divider computes $\frac{A}{B}$ and produces quotient Q and remainder R

→ Each row calculates D = R − B

Signal 'N' indicates whether D is −ve (row's MUX selects MSB = 1)

Q_i = 0 if D = −ve else 1 otherwise

MUX passes R to next row if difference is −ve & D otherwise

Number System

→ Now we must consider fixed & floating point number system (Before only signed & unsigned integers)

Fixed Point Number System

→ It has a decimal point between integer & fraction bits

similar to decimal point between integer & fraction digits of decimal no.

ex: 01101100 ⇒ 0110.1100

→ $2^2 + 2^1 + 2^{-1} + 2^{-2} = 6.75$

(integer bits) high word ← → low word (fractional bits)

→ Signed fixed-point numbers can use either 2's complement or sign/magnitude notation

ex : 0 0 1 0 . 0 1 1 0

2's complement → 1010.0110

sign & magnitude → 1101.1010

Converting Decimal to Binary & Vice-versa for Fraction

ex : 3.125

→ Write 3 in binary as usual $= (011)_2$

→ Take fractional part & keep multiplying with 2 till you get 1

$0.125 \times 2 = 0.25$

$0.25 \times 2 = 0.5$

$0.5 \times 2 = 1$

$\rightarrow (0.001)_2$

→ Write together

$(3.125)_{10} = (011.001)_2$

ex : 011.001

→ Multiply by 2^N N is position

0 1 1 . 0 0 1

2 1 0 -1 -2 -3

$0\times2^2 + 1\times2^1 + 1\times2^0 + 0\times2^{-1} + 0\times2^{-2} + 0.2^{-3}$

$= 0+2+1+0+0+0.125$

$= 3.125$

$(011.001)_2 = (3.125)_{10}$

Q. Compute 0.75 - 0.625 using fixed point numbers

A. $0.625 \Rightarrow (0000.1010)_2$

Applying 2's complement,

$(1111.0101) + 0.0001$

$-0.625 = (1111.0110)_2$

$0.75 = (0000.1100)_2$

$0.75 + (-0.625) = 0.125$ ⇒ $0000.1100 + 1111.0110 = 10000.0010$

But 1 overflows

So, 0000.0010

Floating Point Number System

→ It is similar to scientific notation and avoids the limitation of having constant no. of integer & fractional bits

→ It has $\pm M \times E^B$ (Sign, Mantissa, Base, Exponent)

Q. Represent 228_{10} in floating point representation

A. $228_{10} = 11100100_2 = 1.11100100 \times 10^7$

0	00000111	11100100 00000000 00000000
Sign	Exponent	Mantissa

We make 2 modifications

i) Since first bit of mantissa is always 1, we don't have to store it (implicit leading one)

So we can improve efficiency

0	00000111	11100100 00000000 00000000

ii) The exponent needs to represent +ve & -ve exponents, Hence we used a biased exponent

(Biased exponent = Original Exponent + Constant Bias (127)) ⇒ $7 + 127 = 134 = 10000110$

0	10000110	11100100 00000000 00000000

→ IEEE 754 Standard

→ But all numbers can't be represented like some special cases which are given special notations

No.	Sign	Exponent	Fraction
0	X	00000000	00000000000000000000000
∞	0	11111111	00000000000000000000000
−∞	1	11111111	00000000000000000000000
NaN	X	11111111	Non-zero

→ We have 2 types of precision

i) Single-precision / Single / Float → Sign Bits = 1 , Exponent Bits = 8 , Fraction Bits = 23
   → $\pm 1.175494 \times 10^{-38}$ to $\pm 3.402824 \times 10^{38}$

ii) Double-precision / Double → Sign Bits = 1 , Exponent Bits = 11 , Fraction Bits = 52
   → $\pm 2.2250738585072O \times 10^{-308}$ to $\pm 1.79769313486232 \times 10^{308}$

Q. Express the base 10 Numbers in IEEE 754 single precision floating-point format

i) −13.5625

ii) 42.3125

iii) −17.15625

A. i) $-13.5625 = -(13.5625) = -(1101.1001) = -(1.1011001 \times 10^3)$

$3 + 127 = 130 \Rightarrow 10000010$

1	10000010	10110010000000000000000

ii) $42.3125 = 101010.0101 = 1.01010101 \times 10^5$

$5 + 127 = 132 \Rightarrow 10000100$

0	10000100	01010101000000000000000

iii) $-17.15625 = -(17.15625) = -(10001.00101) = -(1.0001100101 \times 10^4)$

$4 + 127 = 131 \Rightarrow 10000011$

1	10000011	00010010100000000000000

Floating Point Addition

→ Steps to follow:

i) Extract exponent & fraction bit

ii) Add leading 1 to form mantissa

iii) Compare Exponents

iv) Shift smaller mantissa if necessary

v) Add mantissas

vi) Normalize mantissa & adjust exponent if necessary

vii) Round result

viii) Assemble exponent & fraction back into floating point number

Rounding

→ Arithmetic results that fall outside of available precision must round to a neighboring number

→ Rounding modes - round down, round up, round towards 0, round to nearest

→ Overflow - Number is too large to be represented

→ Underflow - Number is too tiny to be represented

Q. Solve 7.875 + 0.1875 by floating point addition

A.

1) 7.875 =

0	1000 0001	1111 100 000 0000 0000 0000 00

0.1875 =

0	0111 1100	1000 0000 000 000 000 000000

2)

0	1000 0001	1.1111 100 000 0000 0000 0000 00
0	0111 1100	1.1000 0000 0000 0000 0000 000000

3) $7.875 = 1.11111 \times 2^{2}$

$0.1875 = 1.1 \times 2^{-3}$

Powers difference = $2-(-3) = (5)_{10} = (101)_{2}$

Shift amount = 101

4)

0	1000 0001	1.1111 100 000 0000 0000 0000 00
0	0111 1100	0.0000 11 0000 0000 0000 0000 000000

+101

00000

5)

0	1000 0001	1.1111 100000 0000 000 0000 00
0	1000 0001	0.0000 11 0000 0000 0000 0000 000000

10.0000011 0000000 0000000000

6)

0	1000 0001

10.0000011 0000000 0000000000 >>1

+1

0	1000 0010

1.0000001 1000000 0000000000

7) No Rounding Required

8)

Answer

0	10000010	0000001 10000000 0000000000

1.0000001×2^{3}

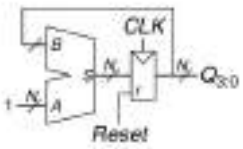
$= 1000.0001 \Rightarrow 8.0625$ //

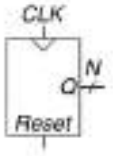
Sequential Building Blocks

→ Counters

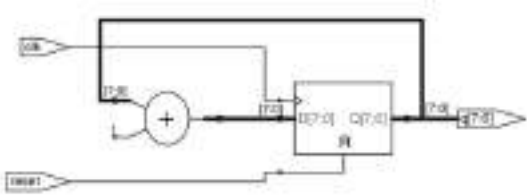
→ It is a sequential arithmetic circuit with CLK and RESET as inputs & N-bit output Q.

→ After every cycle, counter adds 1 to value stored in register





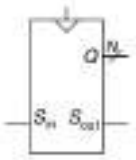

```
module counter #(parameter N = 8)
                (input  logic clk,
                 input  logic reset,
                 output logic [N-1:0] q);

  always_ff @(posedge clk, posedge reset)
    if (reset) q <= 0;
    else       q <= q + 1;
endmodule
```



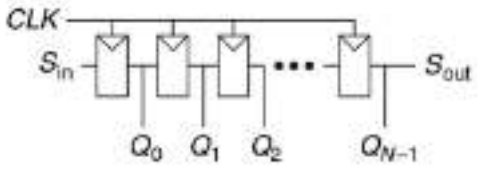
→ Shift Registers

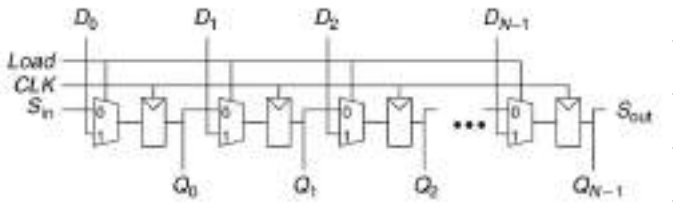
→ It has CLK, S_{in}, S_{out}, N parallel outputs $Q_{N-1:0}$

→ On rising edge of CLK, new bit is shifted from S_{in} & subsequent contents are shifted forward & last bit in shift register is available at S_{out}



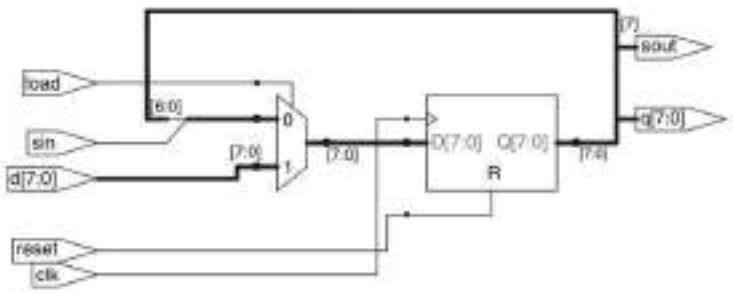
→ They are also called Serial-to-parallel converters because input is given at S_{in} (1 bit at a time) and after N cycles, the past N inputs are in parallel at Q.





→ Shift register can also be modified to perform serial-to-parallel & parallel-to-serial operations by adding parallel input $D_{N-1:0}$ & control signal 'Load'

```
module shiftreg #(parameter N = 8)
                 (input  logic          clk,
                  input  logic          reset, load,
                  input  logic          sin,
                  input  logic [N-1:0]  d,
                  output logic [N-1:0]  q,
                  output logic          sout);

  always_ff @(posedge clk, posedge reset)
    if (reset)      q <= 0;
    else if (load)  q <= d;
    else            q <= {q[N-2:0], sin};

  assign sout = q[N-1];
endmodule
```



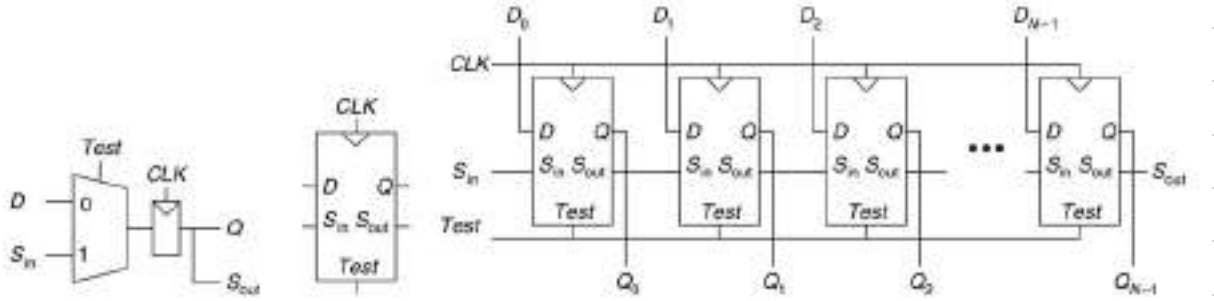
Scan Chains

→ Shift registers test sequential circuits using 'scan chains'
It is relatively difficult compared to combinational circuits testing

→ Designer must directly observe & control all states of machine by adding a test mode in which contents of all flip-flops can be read out or loaded with desired values.

i) In normal mode, flip-flops load data from their D input & ignore

ii) In test mode, flip-flops serially shift contents out & shift in new content using S_{in} & S_{out} respectively



Q. Develop verilog model for the circuit that counts 16 clock cycles & produces control signal ctrl, that is 1 during every 8th & 12th cycle

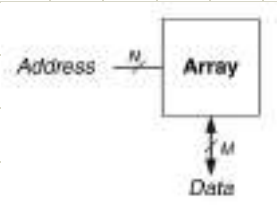
A.

```verilog
// Verilog model for a circuit that counts 16 clock cycles and produces a control signal 'ctrl'
`timescale 1ns/1ps

module clock_counter (
    input clk,      // Clock signal
    input rst,      // Reset signal
    output reg ctrl // Control signal
);

    // 4-bit counter to count up to 16
    reg [3:0] count;

    // Sequential logic: Counter increments on every clock edge
    always @(posedge clk or posedge rst) begin
        if (rst) begin
            count <= 4'd0; // Reset counter to 0
        end else begin
            count <= count + 4'd1; // Increment counter
            if (count == 4'd15) begin
                count <= 4'd0; // Wrap around after 16
            end
        end
    end

    // Combinational logic: Set ctrl high during the 8th and 12th cycles
    always @(*) begin
        if (count == 4'd7 || count == 4'd11) begin
            ctrl = 1'b1;
        end else begin
            ctrl = 1'b0;
        end
    end

endmodule
```

Memory Arrays

→ Memory is organized as 2D array of memory cells

Memory reads/writes contents of one of the rows of the arrays



→ Row is specified by an address

Value read/written is called Data

Array with N-bit addresses & M-bit data has 2^N rows & M columns

Each row of data is called word

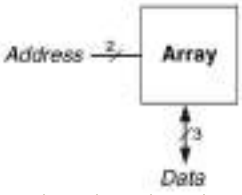
So, array contains 2^N M-bit words

↳ Depth: No of words ↳ Width: No. of columns

ex: $2^2 \times 3$-bit array

No. of words = $2^N = 2^2 = 4$

Width = 3 bits



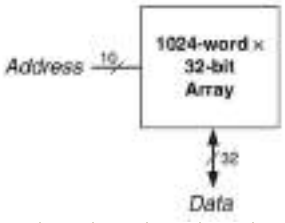
Data is random
No specific table

Address	Data	
1 1	0 1 0	
1 0	1 0 0	depth
0 1	1 1 0	
0 0	0 1 1	

← width →

ex: $2^{10} \times 32$-bit array

Size = 32 Kb



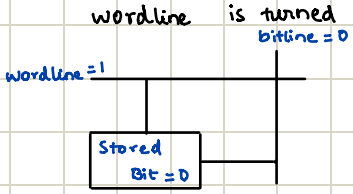
Bit Cells

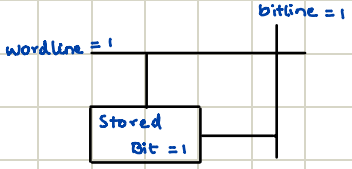
→ Memory arrays are built as an array of bit cells each of which stores 1 bit of data and each bit cell is connected to a wordline & bitline

→ For each combination of address bits, memory asserts a single wordline that activates the bitcell in me in that row

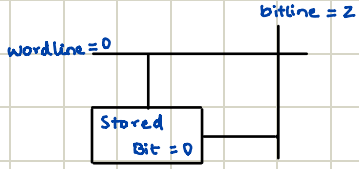
→ Operation:

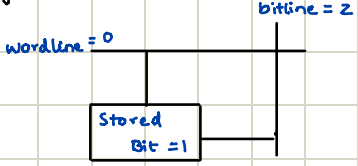
i) Read Operation - To write a bit cell, bitline is initially left floating (Z) wordline is turned ON, allowing stored value to drive the bitline to 0 or 1





ii) Write Operation - To write a bit cell, bitline is strongly driven to desired value, then wordline is turned ON, connecting bitline to stored bit





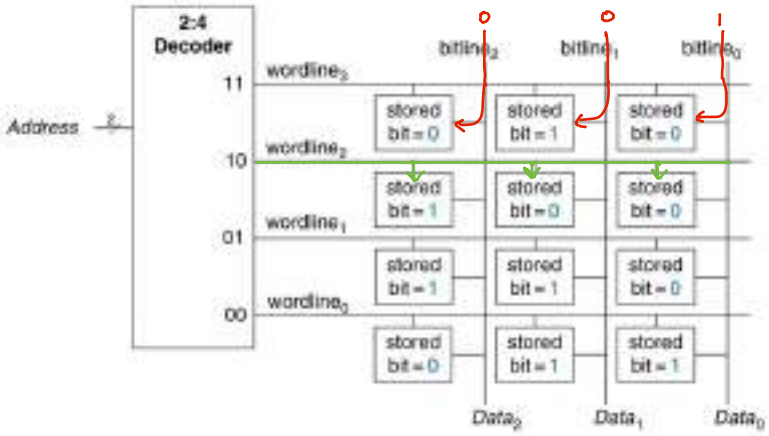
Organization of 4x3 Memory Array

→ Read: a wordline is asserted & corresponding row of bit cells drives the bitlines HIGH or LOW

to read 10, bitlines are left floating, decoder asserts wordline 2 & then data stored in bit cells is read out to Data bitlines

→ Write: Bitlines are driven HIGH or LOW & then wordline is asserted, allowing bitline values to be stored in that row of bit cells

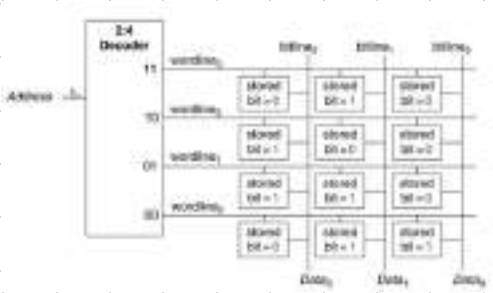
To write 001 to adress 11, bitlines are driven to 001 & wordline3 is asserted & value of 001 is stored



= READ MECHANISM

= WRITE MECHANISM

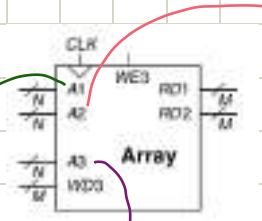
CLEAN COPY FOR REFERENCE



Memory Ports

→ All memories have 1 or more ports and each port gives read and/or write access to one memory address

→ Multiported memories can access several addresses simultaneously



Port 1 reads data from A1 onto RD1

Port 2 reads data from A2 onto RD2

→ Port 3 writes the data WD3 into memory location A3 on rising edge of CLK or when WE3 in enabled

Memory Types

→ Classified based on how bits are stored in bit cell

Random Access Memory

→ Volatile (Loses data when turned OFF)

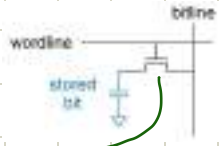
→ Read & written quickly

→ Main memory (DRAM) in computer

DRAM

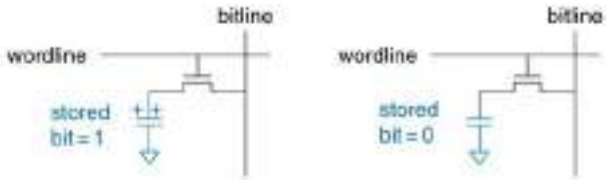
→ Stores data bits on capacitor

→ Dynamic because data needs to be rewritten periodically



→ NMOS behave like switch & when wordline is asserted, NMOS is ON & stored bit value transfers to / from bitline

→ As time passes, noise degrades value of stored bit



SRAM

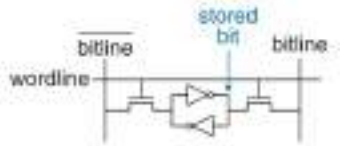
→ Static, cuz data doesn't need to be refreshed

→ Data is stored on cross-coupled inverters

→ Each cell has 2 outputs, bitline & $\overline{\text{bitline}}$

→ When wordline is asserted both NMOS are ON & data values transferred to / from bitlines

→ Unlike DRAM, value is restored in cross-coupled inverter



Comparision of Memory

Memory Type	Transistors per bit cell	Latency	
Flip-flop	20	fast	→ More area but less delay
RAM	6	medium	
RAM	1	slow	→ less area but more latency

Read - Only Memory

→ Non-volatile (Retains data when turned OFF)

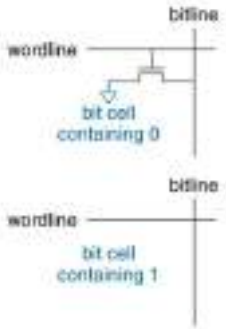
→ Reads quickly but writing is impossible / slow

→ Flash memory in devices

→ Stores a bit as presence / absence of transistor

→ To read the cell, bitline is weakly pulled HIGH & wordline is turned ON

→ If transistor is present, pulls bitline LOW. If absent, bitline remains HIGH



Type	Name	Description
ROM	Read Only Memory	Chip is hardwired with presence or absence of transistors. Changing requires building a new chip.
PROM	Programmable ROM	Fuses in series with each transistor are blown to program bits. Can't be changed after programming.
EPROM	Electrically Programmable ROM	Charge is stored on a floating gate to activate or deactivate transistor. Erasing requires exposure to UV light.
EEPROM	Electrically Erasable Programmable ROM	Like EPROM, but erasing can be done electrically.
Flash	Flash Memory	Like EEPROM, but erasing is done on large blocks to amortize cost of erase circuit. Low cost per bit, dominates nonvolatile storage today.

4x3 ROM Dot Notation

→ Contents of ROM is indicated using DOT Notation where DOT represents intersection of wordline (row) & bitline (column) that the data bit is 1

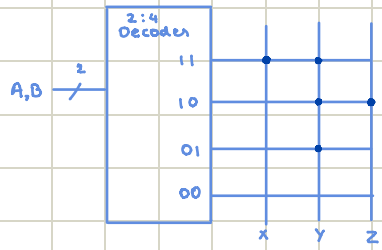
Q. Implement the following functions using a $2^2 \times 3$-bit ROM

$X = AB$

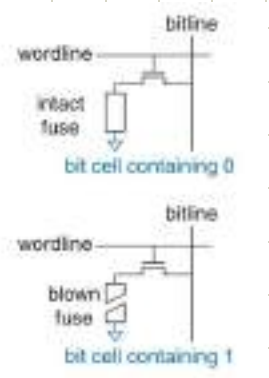
$Y = A + B$

$Z = AB'$

A.



Bit-cell for a fuse-programmable ROM (One-Time Programmable ROM)



→ User programs ROM by applying high voltage to selectively blow fuses

→ If fuse present, transistor connected to ground & cell holds 0.
If fuse destroyed, transistor disconnected from ground & cell holds 1.

Logic using memory arrays

→ Memory arrays can also perform combinational logic functions

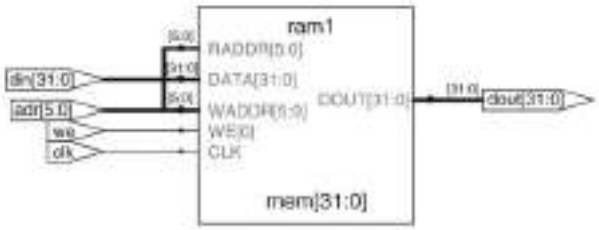
→ Like in the previous question, X is AND gate, Y is OR gate

→ Memory arrays that perform logic are called lookup tables (LUTs)

Memory HDL

→ 2^N-word x M-bit RAM

```
module ram #(parameter N = 6, M = 32)
            (input  logic          clk,
             input  logic          we,
             input  logic [N-1:0] adr,
             input  logic [M-1:0] din,
             output logic [M-1:0] dout);

  logic [M-1:0] mem [2**N-1:0];

  always_ff @(posedge clk)
    if (we) mem[adr] <= din;

  assign dout = mem[adr];
endmodule
```



→ 4 word x 3-bit ROM

```
module rom(input  logic [1:0] adr,
           output logic [2:0] dout);

  always_comb
    case(adr)
      2'b00: dout <= 3'b011;
      2'b01: dout <= 3'b110;
      2'b10: dout <= 3'b100;
      2'b11: dout <= 3'b010;
    endcase
endmodule
```

Logic Arrays

→ Gates organized into regular arrays

→ If connections are made programmable, then any function can be performed w/o specific ways to connect wires

→ Logic arrays are also reconfigurable

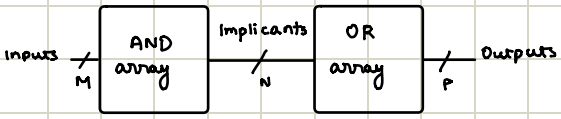
→ 2 Types :

i) Programmable Logic Arrays (PLA)

ii) Field Programmable Gate Arrays (FPGA)

PLAs

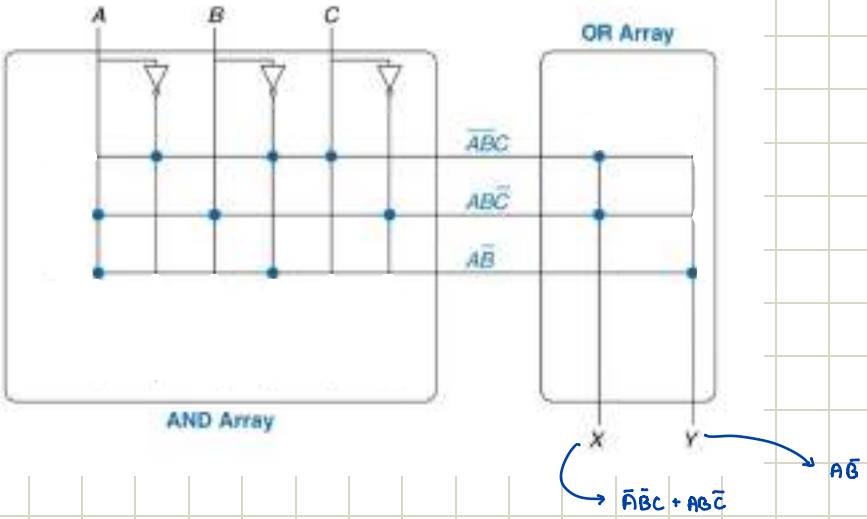
→ Implement 2-level combinational logic is SOP form

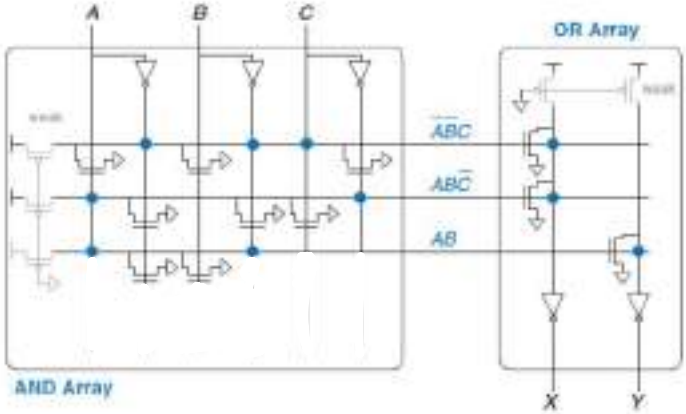


⇒ M x N x P - bit PLA

Q. Dot Notation for 3x3 x 2 -bit PLA performing $X = A'B'C + ABC'$ & $Y = AB'$

A.





To minimize size & cost, ROMs & PLAs use pseudo NMOS or dynamic circuits instead of conventional logic gates

FPGAs

→ Array of reconfigurable gates.

→ Software tools like Xilinx Vivado can be used to implement designs onto it using HDL or schematic

→ FPGAs are powerful & flexible than PLAs

→ They can implement both combinational & sequential logic

→ Can implement multi-level functions

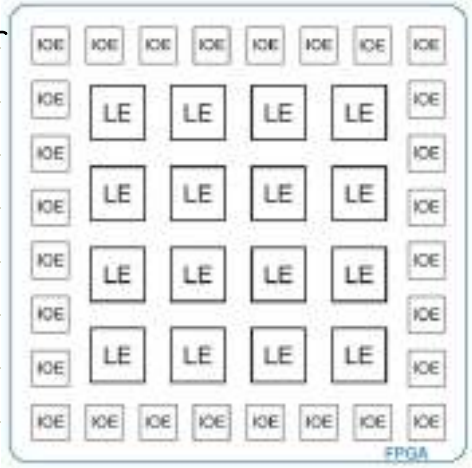
→ Composed of Logic Elements, Input / Output Elements, Programmable Interconnection

- Logic Elements → Perform Logic
- Input / Output Elements → Interface with outside world
- Programmable Interconnection → Connect LEs & IOEs

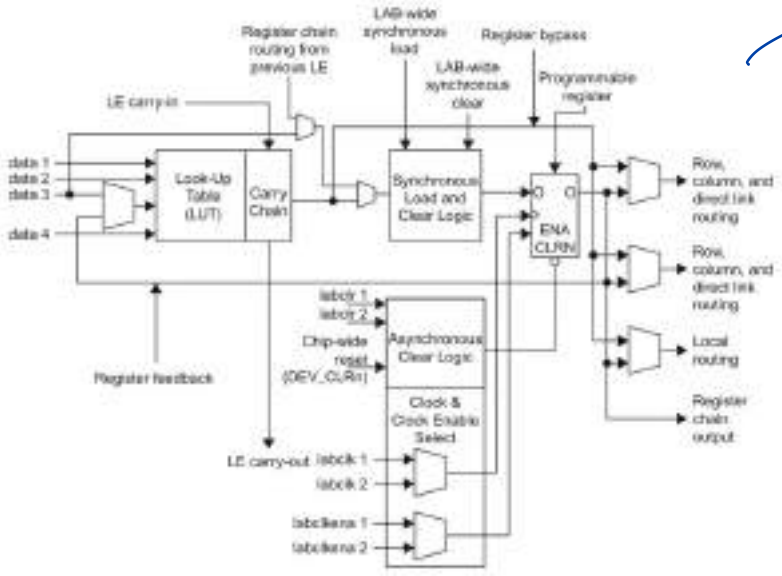
Composed of LUTs, FFs, MUXs

- LUTs → Perform Combinational Logic
- FFs → Perform Sequential Logic
- MUXs → Connect LUTs & FFs

General FPGA Layout



→ Leading FPGA manufactures - Altera & Xilinx



→ Cyclone IV LE

→ Only 1 Logic element

Consists of

i) 1 4-input LUT

ii) 1 registered output

iii) 1 combinational output

Alyssa P. Hacker is building a finite state machine that must run at 200 MHz. She uses a Cyclone IV GX FPGA with the following specifications: $t_{LE} = 381$ ps per LE, $t_{setup} = 76$ ps, and $t_{pcq} = 199$ ps for all flip-flops. The wiring delay between LEs is 246 ps. Assume the hold time for the flip-flops is 0. What is the maximum number of LEs her design can use?

Solution: Alyssa uses Equation 3.13 to solve for the maximum propagation delay of the logic: $t_{pd} \leq T_c - (t_{pcq} + t_{setup})$.

Thus, $t_{pd} = 5$ ns $- (0.199$ ns $+ 0.076$ ns), so $t_{pd} \leq 4.725$ ns. The delay of each LE plus wiring delay between LEs, $t_{LE+wire}$, is 381 ps + 246 ps = 627 ps. The maximum number of LEs, N, is $Nt_{LE+wire} \leq 4.725$ ns. Thus, $N = 7$.

Example 5.6 FUNCTIONS BUILT USING LEs

Explain how to configure one or more Cyclone IV LEs to perform the following functions: (a) $X = \overline{A}\,\overline{B}C + AB\overline{C}$ and $Y = A\overline{B}$ (b) $Y = JKLMPQR$; (c) a divide-by-3 counter with binary state encoding (see Figure 3.29(a)). You may show interconnection between LEs as needed.

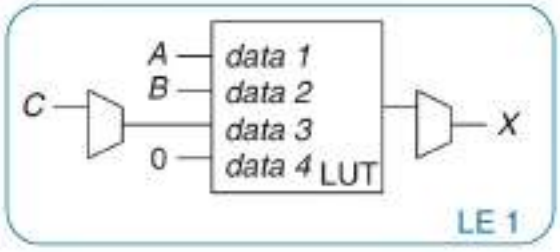
Solution: (a) Configure two LEs. One LUT computes X and the other LUT computes Y, as shown in Figure 5.59. For the first LE, inputs *data 1*, *data 2*, and *data 3* are A, B, and C, respectively (these connections are set by the routing channels). *data 4* is a don't care but must be tied to something, so it is tied to 0. For the second LE, inputs *data 1* and *data 2* are A and B; the other LUT inputs are don't cares and are tied to 0. Configure the final multiplexers to select the combinational outputs from the LUTs to produce X and Y. In general, a single LE can compute any function of up to four input variables in this fashion.

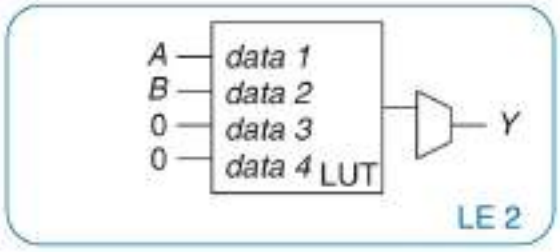
(b) Configure the LUT of the first LE to compute $X = JKLM$ and the LUT on the second LE to compute $Y = XPQR$. Configure the final multiplexers to select the combinational outputs X and Y from each LE. This configuration is shown in Figure 5.60. Routing channels between LEs, indicated by the dashed blue lines, connect the output of LE 1 to the input of LE 2. In general, a group of LEs can compute functions of N input variables in this manner.

(c) The FSM has two bits of state ($S_{1:0}$) and one output (Y). The next state depends on the two bits of current state. Use two LEs to compute the next state from the current state, as shown in Figure 5.61. Use the two flip-flops, one from each LE, to hold this state. The flip-flops have a reset input that can be connected to an external *Reset* signal. The registered outputs are fed back to the LUT inputs

(A) *data 1*	(B) *data 2*	(C) *data 3*	*data 4*	(X) LUT output
0	0	0	X	0
0	0	1	X	1
0	1	0	X	0
0	1	1	X	0
1	0	0	X	0
1	0	1	X	0
1	1	0	X	1
1	1	1	X	0

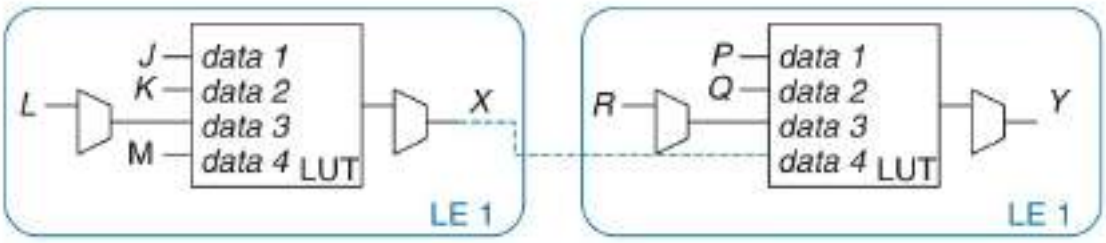
(A) *data 1*	(B) *data 2*	*data 3*	*data 4*	(Y) LUT output
0	0	X	X	0
0	1	X	X	0
1	0	X	X	1
1	1	X	X	0





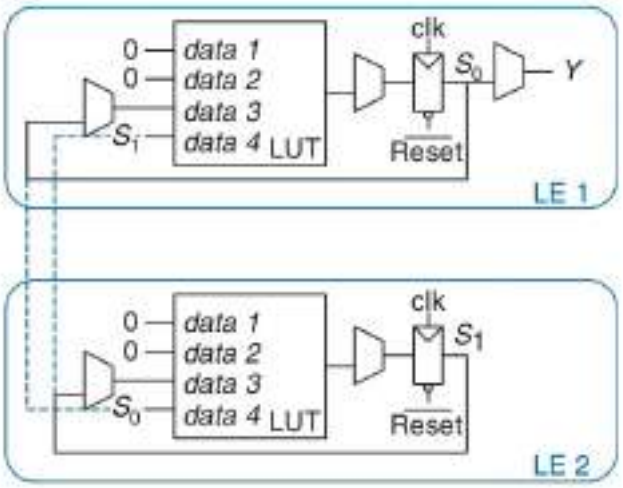
(J) data 1	(K) data 2	(L) data 3	(M) data 4	(X) LUT output
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1

(P) data 1	(Q) data 2	(R) data 3	(X) data 4	(Y) LUT output
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	1



(S_0) data 1	data 2	data 3	(S_1) data 4	(S_0') LUT output
X	X	0	0	1
X	X	0	1	0
X	X	1	0	0
X	X	1	1	0

(S_1) data 1	data 2	data 3	(S_0) data 4	(S_1') LUT output
X	X	0	0	0
X	X	0	1	1
X	X	1	0	0
X	X	1	1	0



using the multiplexer on *data 3* and routing channels between LEs, as indicated by the dashed blue lines. In general, another LE might be necessary to compute the output Y. However, in this case $Y = S_0$, so Y can come from LE 1. Hence, the entire FSM fits in two LEs. In general, an FSM requires at least one LE for each bit of state, and it may require more LEs for the output or next state logic if they are too complex to fit in a single LUT.