



Bridge of Life
Education

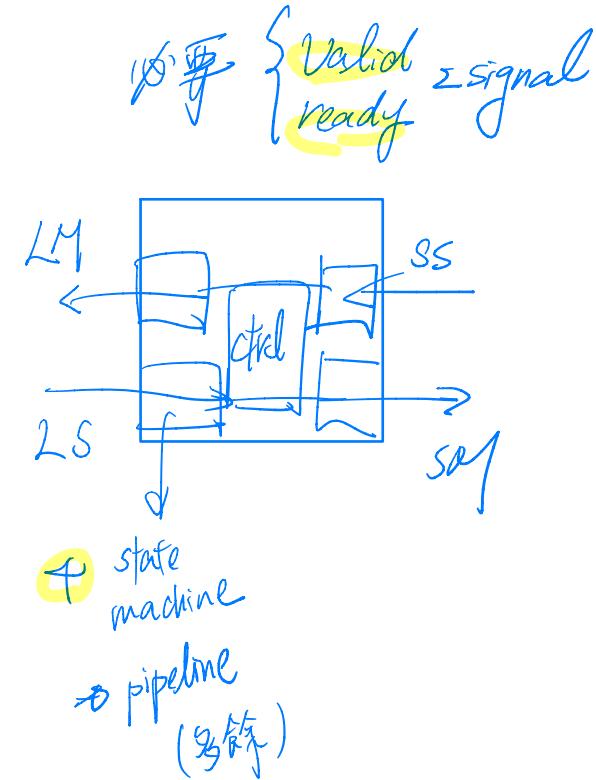
SOC Design Verilog Logic Design

Jiin Lai

Level of Proficiency in Verilog Design

1. Code matches simulation result (Testbench + RTL)
2. Pre-synthesis (RTL) matches Post-synthesis (RTL = Gate-level)

3. Design Quality (PPA – Power/Performance/Area)
plz 暫解 一下 module
4. System/Application Level Optimization



Agenda

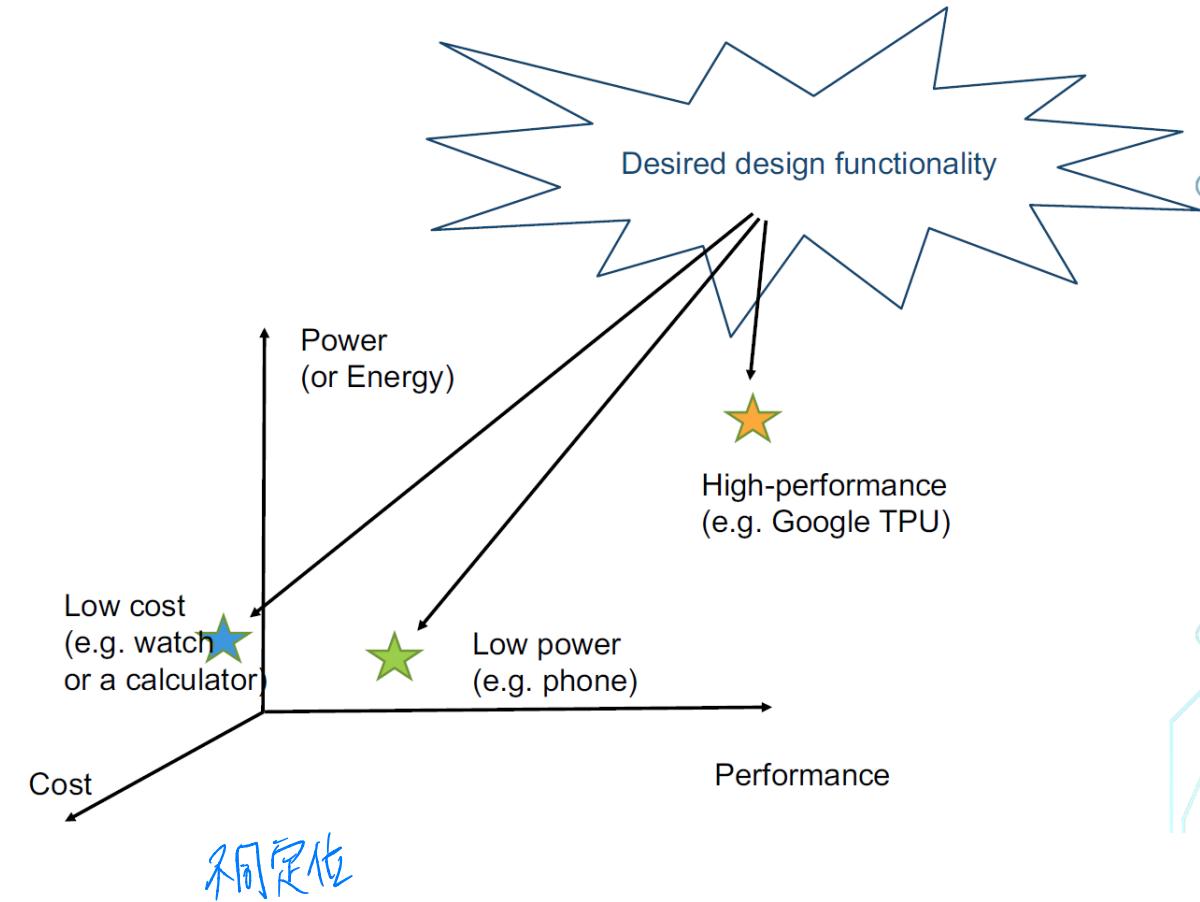
1. Design Background
2. Combinational Design
3. Sequential Design

Advanced Design

- Reset – Synchronous/Asynchronous, Power-on Reset
- Clock – clock tree balance, clock switching, CDC, useful skew
- Asynchronous FIFO
- Source Synchronous
- Latch Design – Time borrowing
- State encoding
- Hierarchical Design

Design Metric – Performance, Power, Area (Cost) - PPA

- The desired functionality can be implemented with different performance, power or cost targets
- Design is the tradeoff of **“Performance”, “Area”, “Cost”**



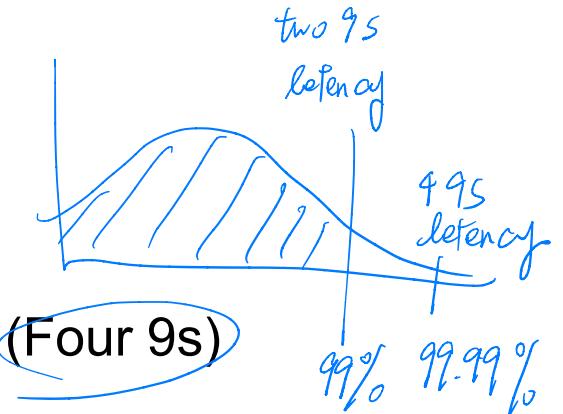
Performance

• Throughput task/time (MB/s or op/s)

- Number of tasks performed in a unit of time (operations per second)
 - E.g. Google TPUv3 board performs 420 TFLOPS (10^{12} floating-point operations per second, where a floating-point operation is BFLOAT16) *Term* *16-bit FLOAT*
- Watch out for ‘op’ definitions – can be a 1-b ADD or a double-precision FP add (or more complex task)
- Peak vs. average throughput *no 固定標準*
- Resource utilization – How efficient the computing resource is used.

• Latency

- How long does a task take from start to finish
 - E.g. facial recognition on a phone takes 10's of ms
- Sometimes expressed in terms of clock cycles
- Average vs. ‘tail’ latency, e.g. latency for 99.99% of transaction *(Four 9s)*



Power and Energy

- Energy (in joules (J))
 - Needed to perform a task, e.g. Add two numbers or fetch a datum from memory
 - Active and standby
- Power (in watts (W))
 - Energy dissipated in time ($W = J/s$)
 - Peak power vs average power

load-store v.s. Dataflow architecture.

L-S von Neumann

Energy/instruction $\frac{E}{I}$

Data non von-

in3pJ.

→ Application Acceleration : Dataflow E_d

Area / Cost

- **Non-recurring engineering (NRE) costs**

- Fixed, one-time cost to research, design, and verify a new piece of HW.
- Fixed, mask design
- Amortized over all units shipped
 - e.g. \$20M in development adds \$.20 to each of 100M units

$$\text{cost per IC} = \text{variable cost per IC} + \frac{\text{fixed cost}}{\text{volume}}$$

• Recurring costs

- Cost to manufacture, test and package a unit
- Processed wafer cost is which yields:
 - 1 Cerebras chip
 - 50-100 large FPGAs or GPUs
 - 200 laptop CPUs
 - >1000 cell phone SoCs

$$\text{variable cost} = \frac{\text{cost of die} + \text{cost of die test} + \text{cost of packaging}}{\text{final test yield}}$$

Verilog Introduction

Module – A component in a circuit

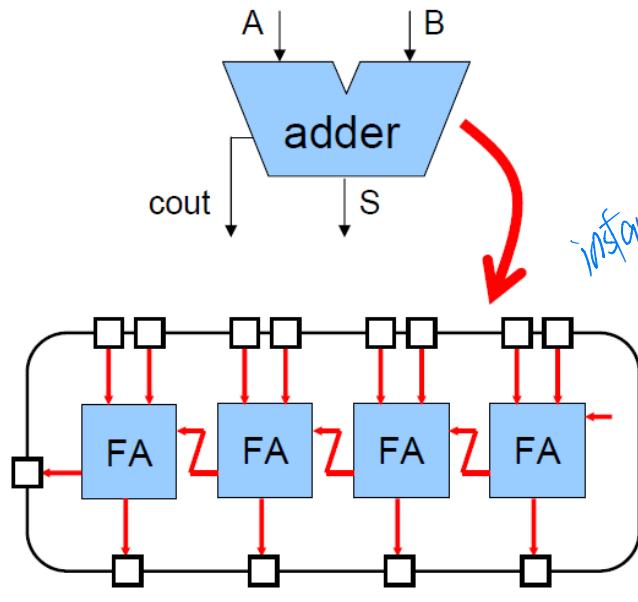
- **Structural Verilog** *module → connection*
 - List of sub-components and how they are connected
 - Just like ~~schematics~~, but using text
 - tedious to write, hard to decode
 - You get precise control over circuit details
 - May be necessary to map to special resources of the FPGA/ASIC
- **Behavioral Verilog** *no for, while, delay...*
 - Describe what a component does, not how it does it
 - ~~Synthesized~~ into a circuit that has this behavior
 - Result is only as good as the tools
- Hierarchy of modules. Top-level module is your entire design (or the environment to test your design).

Verilog Modules (Declaration) and Instantiation (Instance)

```
name          port list  
module addr_cell(a, b, cin, s, cout);  
    input  a, b, cin;  
    output s, cout;           port declarations (input,  
                                                output, or inout)  
                                module body  
  
endmodule  
  
module adder (A, B, S);           Instance of addr_cell  
    addr_cell ac1( ... connections ... );  
  
endmodule
```

Difference between a Verilog “**module**” and a C “**function**” ?

Hierarchical Modules



module FA(a, b, cin, cout, sum)

Ports are attached to nets either **by position**

```
module adder( input [3:0] A, B,  
              output cout,  
              output [3:0] S );  
  
    wire c0, c1, c2;  
    FA fa0( A[0], B[0], 0, c0, S[0] );  
    FA fa1( A[1], B[1], c0, c1, S[1] );  
    FA fa2( A[2], B[2], c1, c2, S[2] );  
    FA fa3( A[3], B[3], c2, cout, S[3] );  
  
endmodule
```

Module are connected together with nets

Ports are attached to nets **by name** *prefer*

```
module adder( input [3:0] A, B,  
              output cout,  
              output [3:0] S );  
  
    wire c0, c1, c2;  
    FA fa0( .a(A[0]), .b(B[0]),  
            .cin(0), .cout(c0),  
            .sum(S[0]) );  
  
    FA fa1( .a(A[1]), .b(B[1]),  
            ...  
    );  
  
endmodule
```

Structural Model - XOR example

```
module xor_gate( out, a, b );
    input a, b;
    output out;
    wire aBar, bBar, t1, t2;
    not invA (aBar, a);
    not invB (bBar, b);
    and and1 (t1, a, bBar);
    and and2 (t2, b, aBar);
    or  or1 (out, t1, t2);
endmodule
```

module name: `xor_gate` (highlighted by a dashed box)

port list: `(out, a, b)` (highlighted by a dashed box)

port declarations: `input a, b;` and `output out;`

internal signal declarations: `wire aBar, bBar, t1, t2;`

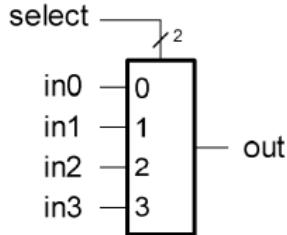
Built-in gates: `not`, `and`, `or` (highlighted by a blue oval)

instances: `invA`, `invB`, `and1`, `and2`, `or1` (highlighted by red arrows)

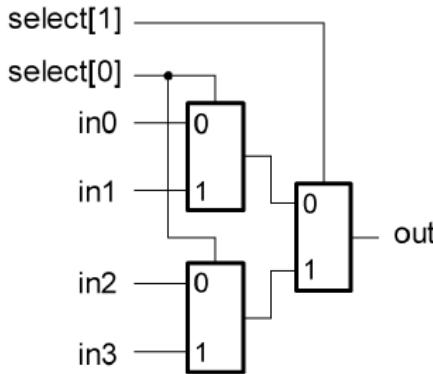
Instance name: `invA`, `invB`, `and1`, `and2`, `or1` (highlighted by red arrows)

Interconnections (note output is first): A logic diagram showing the internal connections of the module. Input `a` connects to the `invB` gate (inverted to `bBar`) and to one input of the `and1` gate (inverted to `t1`). Input `b` connects to the `invA` gate (inverted to `aBar`) and to one input of the `and2` gate (inverted to `t2`). The outputs of `invB` and `invA` are connected to the inputs of `and1` and `and2` respectively. The outputs of `and1` (`t1`) and `and2` (`t2`) are connected to the inputs of the `or1` gate, which produces the final output `out`.

Instantiation, Signal Array, Named ports



a) 4-input mux symbol



b) 4-input mux implemented with 2-input muxes

```
/* 2-input multiplexor in gates */
module mux2 (in0, in1, select, out);
    input in0,in1,select;
    output out;
    wire s0,w0,w1;
    not (s0, select);
    and (w0, s0, in0),
        (w1, select, in1);
    or (out, w0, w1);
endmodule // mux2
```

```
module mux4 (in0, in1, in2, in3, select, out);
    input in0,in1,in2,in3;
    input [1:0] select;           ----- Signal array. Declares select[1], select[0]
    output out;
    wire w0,w1;
    mux2
        m0 (.select(select[0]), .in0(in0), .in1(in1), .out(w0)),
        m1 (.select(select[0]), .in0(in2), .in1(in3), .out(w1)),
        m3 (.select(select[1]), .in0(w0), .in1(w1), .out(out));
endmodule // mux4
```

Named ports. Highly recommended.

Continuous Assignment Examples

not too compliment

Assign values whenever there is a change in the RHS.

Model combinational logic without specifying an interconnection of gates.



```
wire [3:0] A, X,Y,R,Z;  
wire [7:0] P;  
wire r, a, cout, cin;
```



```
assign R = X | (Y & ~Z);  
assign r = &X; ← example reduction operator  
assign R = (a == 1'b0) ? X : Y; ← conditional operator  
assign P = 8'hff; ← example constants  
assign P = X * Y; ← arithmetic operators (use with care!)  
assign P[7:0] = {4{X[3]}, X[3:0]}; ← (ex: sign-extension)  
assign {cout, R} = X + Y + cin; ← bit field concatenation  
assign Y = A << 2; ← bit shift operator  
assign Y = {A[1], A[0], 1'b0, 1'b0}; ← equivalent bit shift
```

Non-Continuous (**Procedure**) Assignments (always @)

```
module and_or_gate (out, in1, in2, in3);
    input      in1, in2, in3;
    output     out;
    reg        out;      "reg" type declaration. Not really a register in
                        this case. Just a Verilog idiosyncrasy.
```

```
always @ (in1 or in2 or in3) begin
    out = (in1 & in2) | in3;
end
```

LHS (lefthand side) declare reg

keyword

“sensitivity” list, triggers the action in the body.

blocking statement

brackets multiple statements (not necessary in this example.)

Isn't this just: assign out = (in1 & in2) | ins3; Why bother ?

Verilog was originally designed as a simulation language

Always Blocks – Case, If ... else

```
module mux4 (in0, in1, in2, in3, select, out);  
    input in0,in1,in2,in3;  
    input [1:0] select;  
    output      out;  
    reg         out;  
  
    always @ (in0 in1 in2 in3 select)  
        case (select)  
            2'b00: out=in0;  
            2'b01: out=in1;  
            2'b10: out=in2;  
            2'b11: out=in3;  
        endcase  
endmodule // mux4
```

Which constant matches
“select” get applied.

```
module mux4 (in0, in1, in2, in3, select, out);  
    input in0,in1,in2,in3;  
    input [1:0] select;  
    output      out;  
    reg         out;  
  
    always @ (in0 in1 in2 in3 select)  
        if (select == 2'b00)  
            out=in0;  
        else if (select == 2'b01)  
            out=in1;  
        else if (select == 2'b10)  
            out=in2;  
        else out=in3;  
    endmodule // mux4
```

Nested if structure leads to “priority logic”
structure: with different delays for
different inputs
(in3 to out delay > than in0 to out delay).
Case version treats all inputs the same.

Generator : An example Ripple Adder

```
module Adder(A, B, R);  
  parameter N = 4;           Declare a parameter with default value.  
  input [N-1:0] A;          Note: this is not a port. Acts like a “synthesis-time” constant.  
  input [N-1:0] B;  
  output [N:0] R;  
  wire [N:0] C;  
  
  genvar i;                Replace all occurrences of “4” with “N”.  
  
  generate  
    for (i=0; i<N; i=i+1) begin:bit  
      FullAdder add(.a(A[i]), .b(B[i]), .ci(C[i]), .co(C[i+1]), .r(R[i]));  
    end  
  endgenerate  
  
  assign C[0] = 1'b0;  
  assign R[N] = C[N];  
endmodule
```

not hardware resource

variable exists only in the specification - not in the final circuit.

Keyword that denotes synthesis-time operations

For-loop creates instances (with unique names)

Parameters give us a way to generalize your designs.
e.g.
Adder #(N(64)) adder64 (...);

Are these combinational circuits correct ?

```
// Example A: 3-Input Adder  
always @(a or b) begin  
    out = a + b + c;  
end
```

3 input → simulation mismatch

```
// Example B:  
assign out = in & out;
```

combinational
loopback

```
// Example C:  
always @(*) begin  
    case (select)  
        2'b00: out=in0;  
        2'b01: out=in1;  
        2'b10: out=in2;  
    endcase  
end
```



latch up

Basics

- Combinational logic: → (组合逻辑)
procedure block

- Continuous Assignment:

- `assign a = b & c;`

- Always block with @(*)

- `always @(*) begin`
`a = b & c; // blocking statement`
`end`

- Sequential logic:

- Always block with @(posedge clk)

- `always @(posedge clk) begin`
`a <= b & c; // nonblocking statement`
`end`

Combinational Logic

Combinational Logic Representation

input
多
數清楚

Truth Table

≈ case

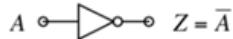
if else
???

C	B	A	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

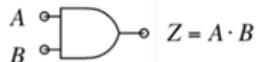
• Boolean Expression

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

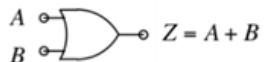
INVERTER:



AND:



OR:



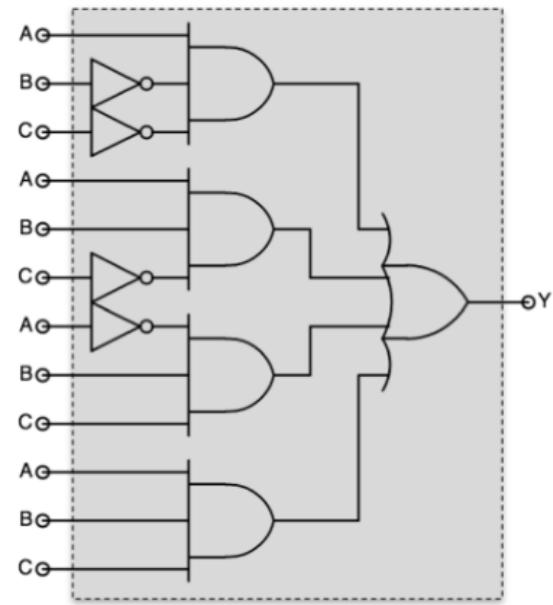
A	Z
0	1
1	0

A	B	Z
0	0	0
0	1	0
1	0	0
1	1	1

A	B	Z
0	0	0
0	1	1
1	0	1
1	1	1

- Sum-of-Product
- 3 levels of INV, AND, OR
- What if # inputs increase?

Schematic View



Building Blocks – NAND, NOR, XOR

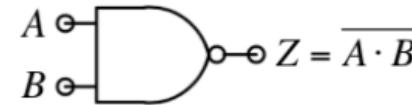
- CMOS is naturally inverting
- NAND – a unit of a gate (4 transistors)
- NAND/NOR are not associative

$$\overline{A \& B \& C} \neq A \& \overline{B \& C}$$

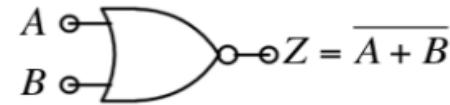
- XOR – parity and arithmetic logic
- Associative

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

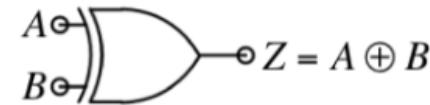
NAND (not AND)



NOR (not OR)



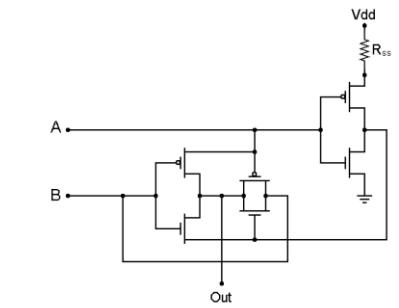
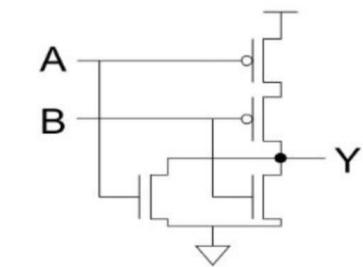
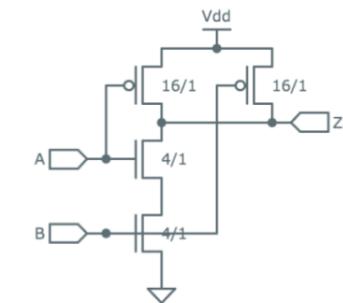
XOR (exclusive OR)



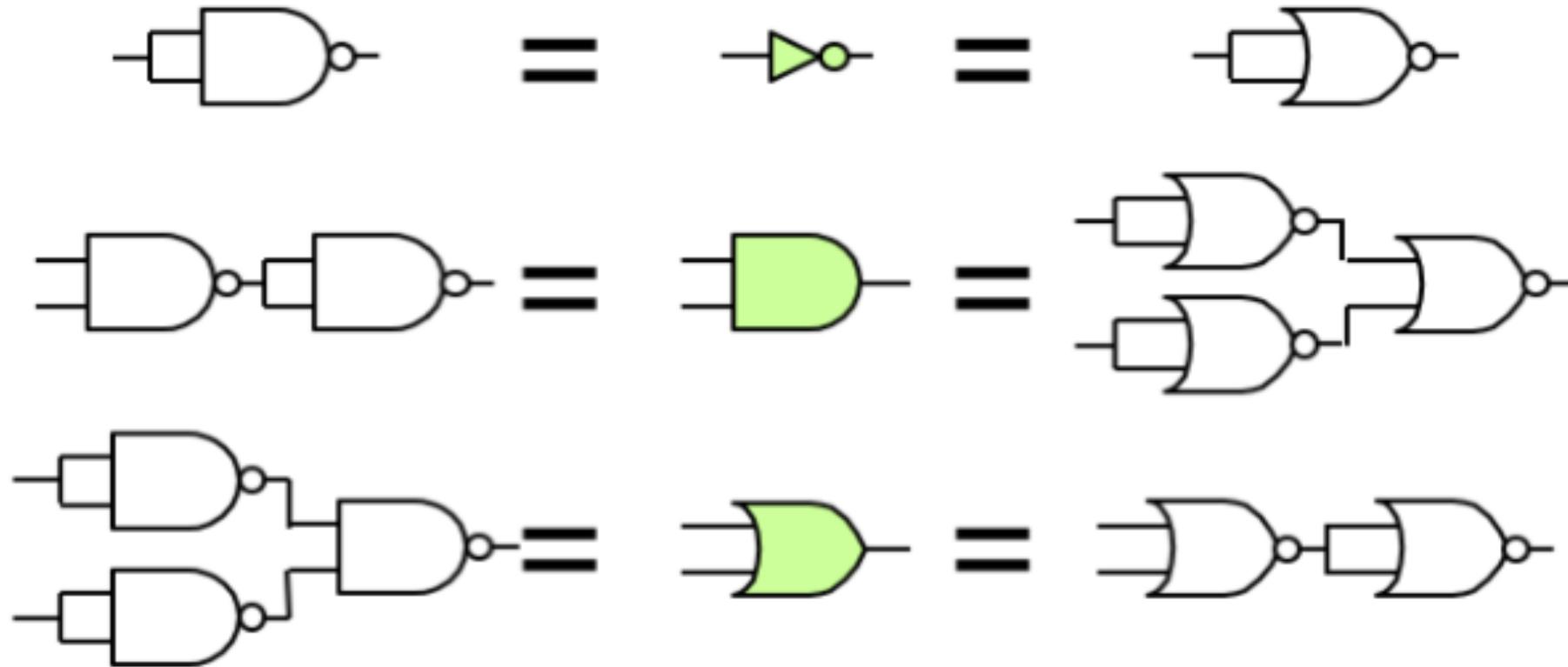
A	B	Z
0	0	1
0	1	1
1	0	1
1	1	0

A	B	Z
0	0	1
0	1	0
1	0	0
1	1	0

A	B	Z
0	0	0
0	1	1
1	0	1
1	1	0



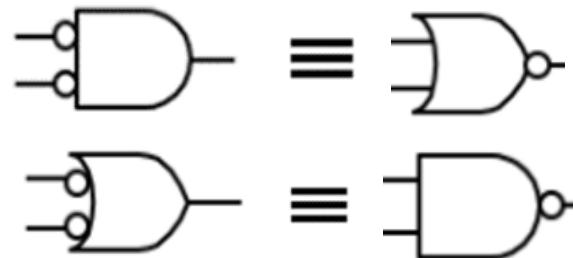
Universal Building Blocks – NAND, NOR



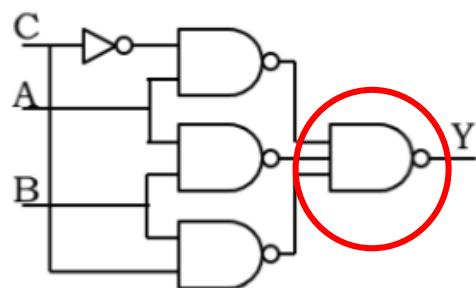
Demorgan's Laws

Demorgan's
Laws:

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



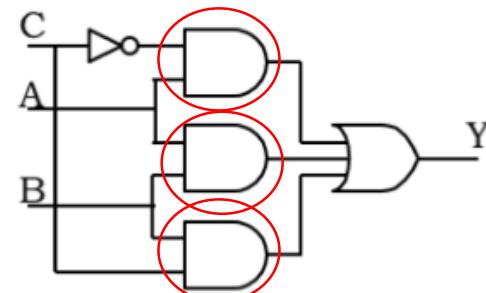
NAND-NAND



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



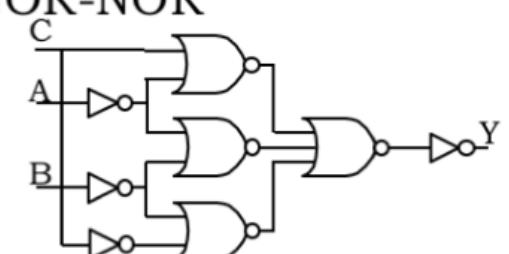
"Pushing Bubbles"



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



NOR-NOR



Logic Minimization: Truth Table

C	B	A	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

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

$\rightarrow \bar{C}A$

$\rightarrow CB$

$\rightarrow BA$

- Truth Table with “Don’t Cares”
- Don’t Cares input is irrelevant

Logic Minimization: Boolean Minimization (Boolean Algebra)

Boolean Algebra

OR rules: $a + 1 = 1, a + 0 = a, a + a = a$

AND rules: $a1 = a, a0 = 0, aa = a$

Commutative: $a + b = b + a, ab = ba$

Associative: $(a + b) + c = a + (b + c), (ab)c = a(bc)$

Distributive: $a(b+c) = ab + ac, a + bc = (a+b)(a+c)$

Complements: $a + \bar{a} = 1, a\bar{a} = 0$

Absorption: $a + ab = a, a + \bar{a}b = a + b, a(a + b) = a, a(\bar{a} + b) = ab$

Reduction: $ab + \bar{a}b = b, (a + b)(\bar{a} + b) = b$

DeMorgan's Law: $\bar{a} + \bar{b} = \bar{ab}, \bar{ab} = \bar{a} + \bar{b}$

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

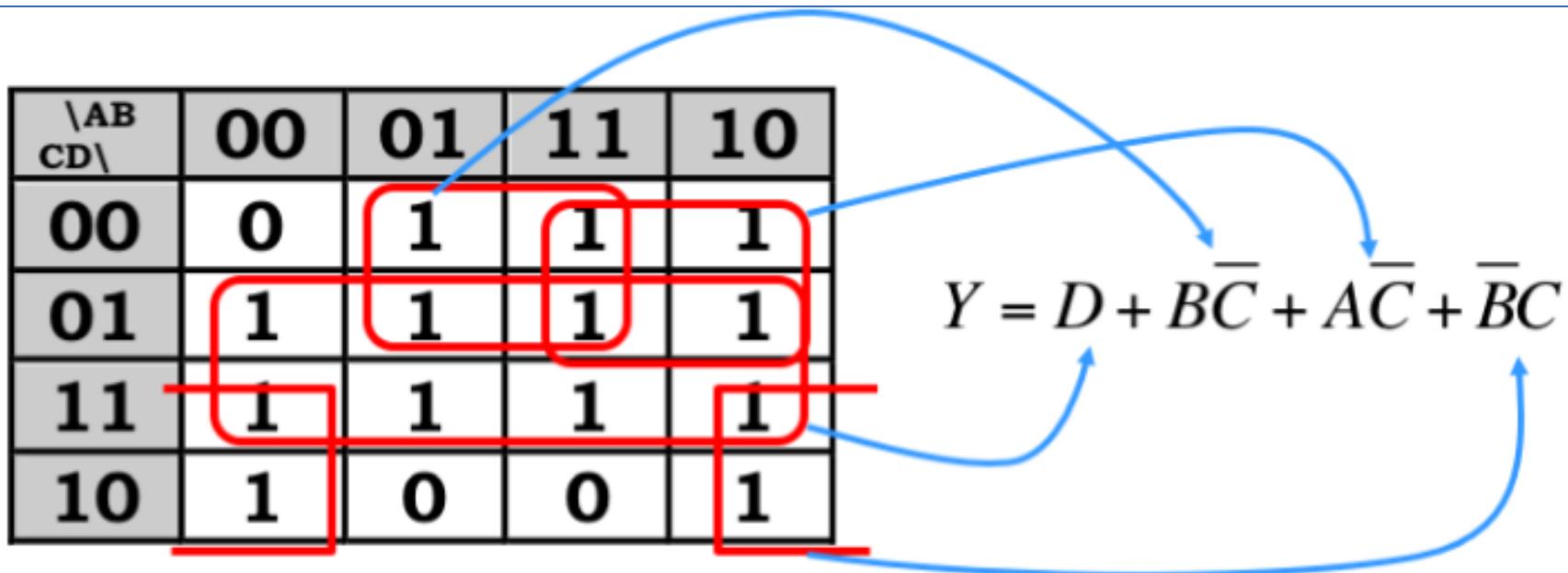
$$Y = \bar{C}\bar{B}A + CB + \bar{C}BA$$

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

Logic Minimization – Karnaugh Maps

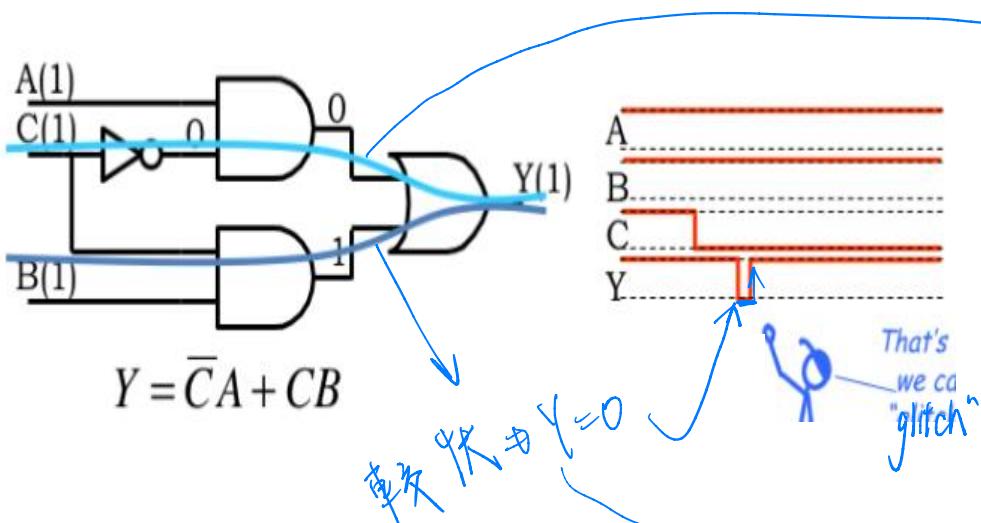
Karnaugh Maps

- Find implicants – rectangular region function with “1”
- Prime implicant – not completely contained in any other implicant
- Pick prime implicant cover all “1”

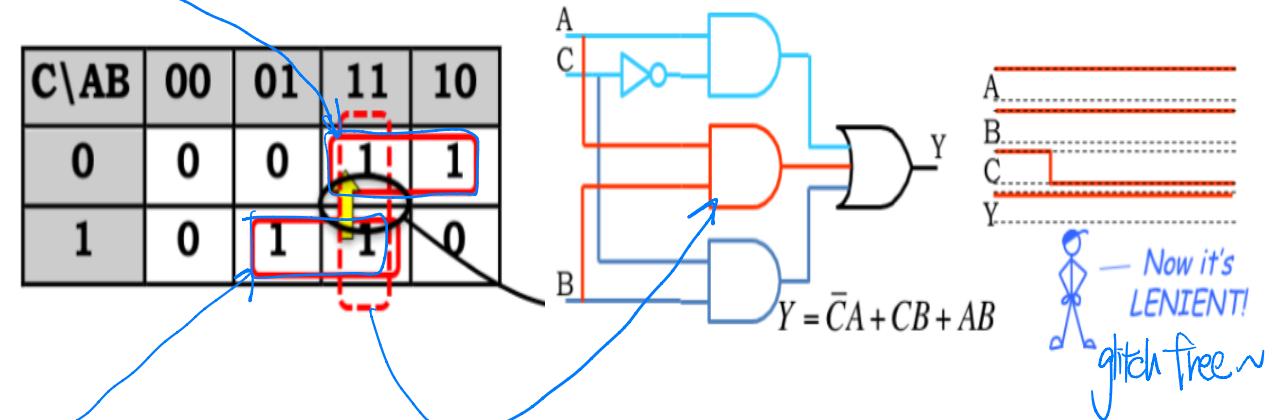


“Hazard” circuit

- “glitch” or “hazard” circuit

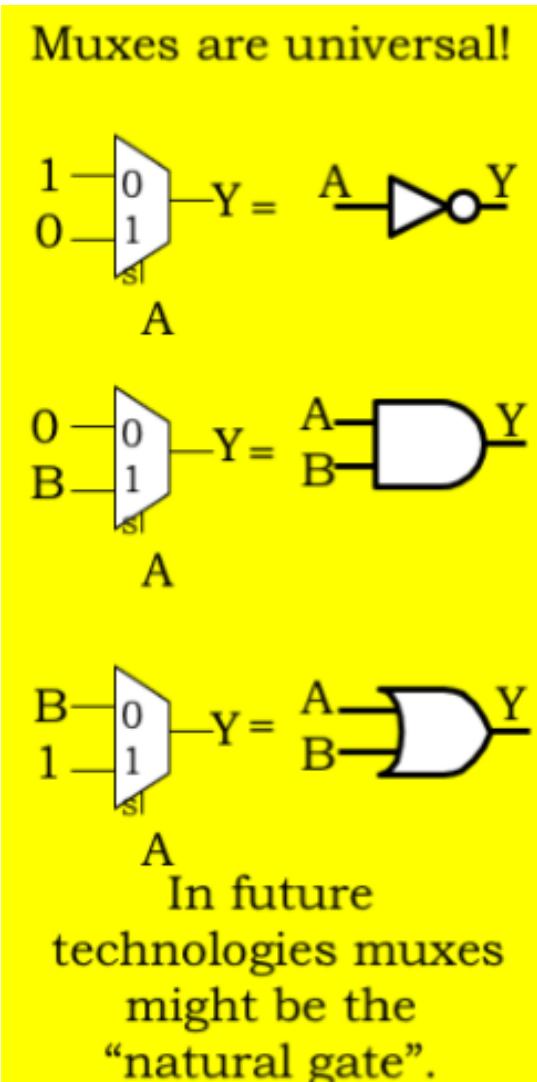


- To make glitch-free, include product terms link for all prime implicants

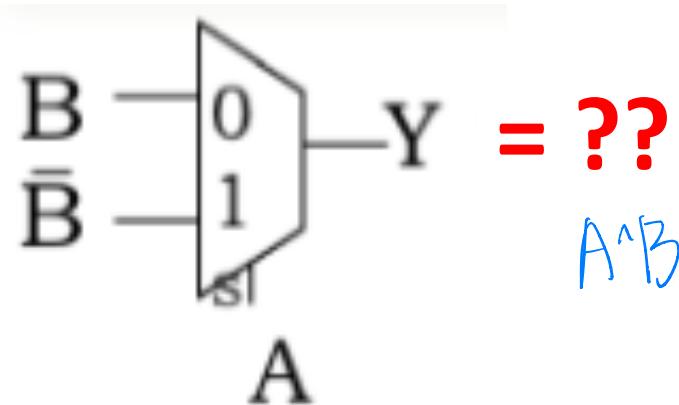


- For synchronous design, glitch does not matter as long as settled before sampled
- However, it does increase delay for settling the glitch

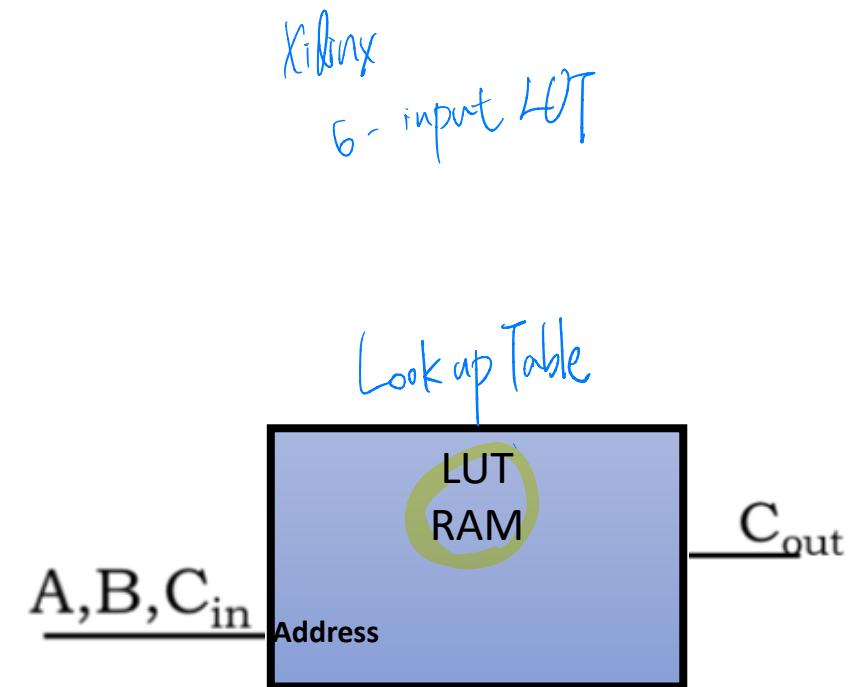
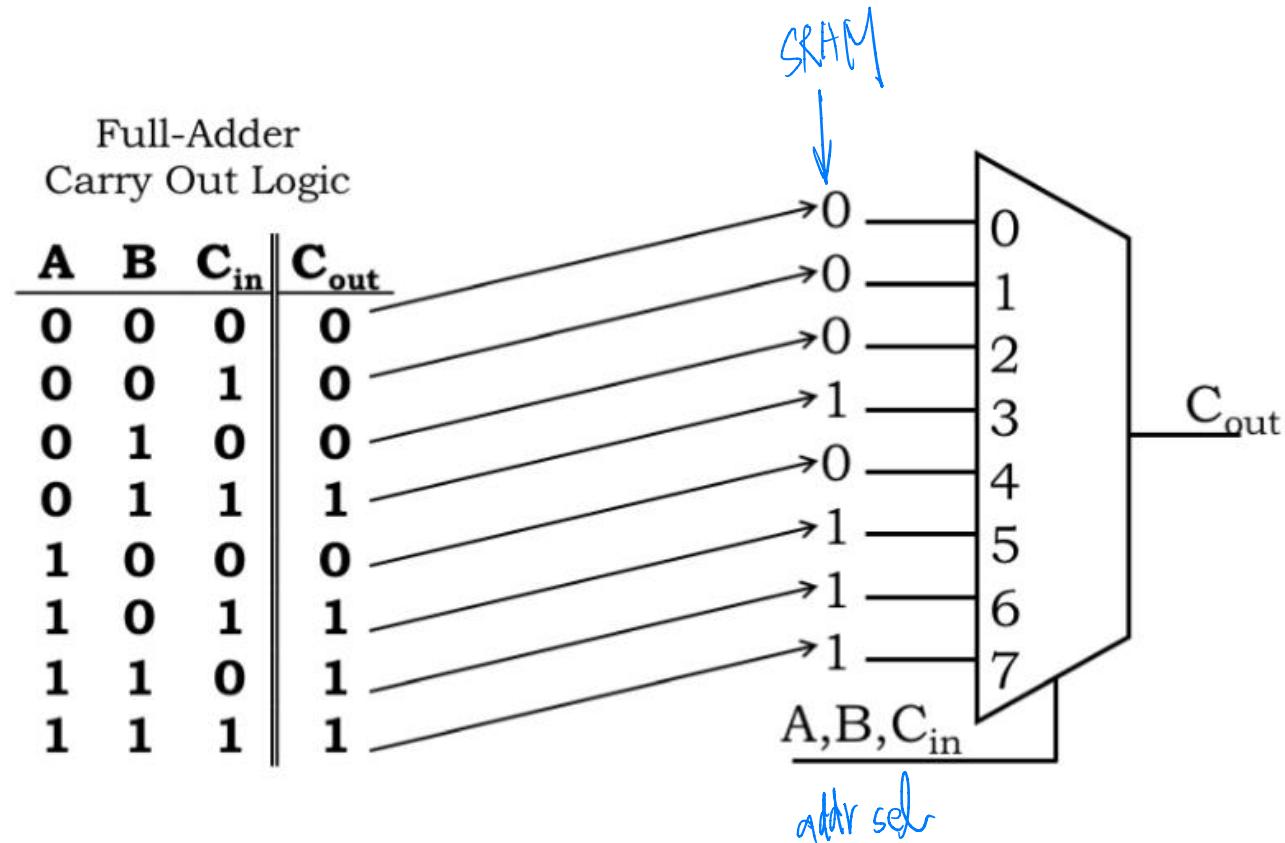
Another Universal Gate - MUX



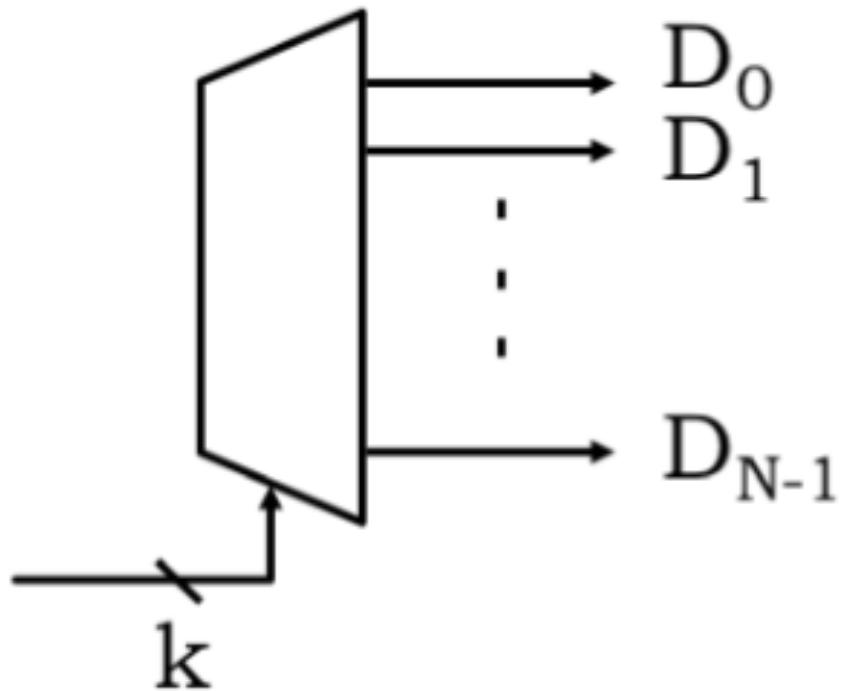
Transmission Gate



Encoder



Decoder

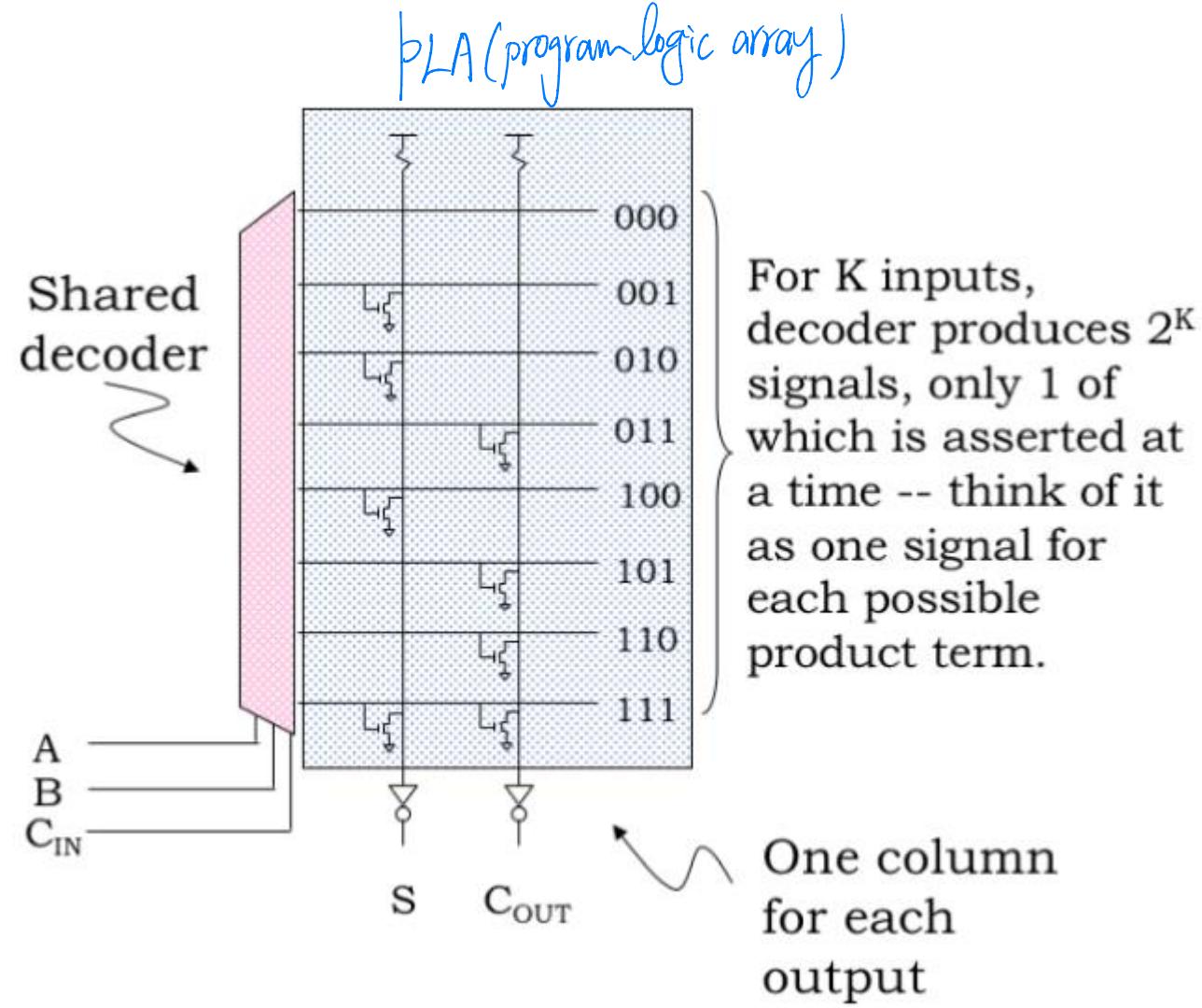
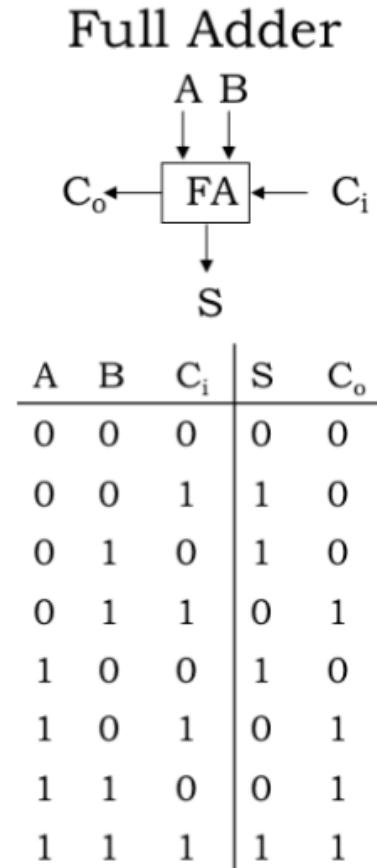


DECODER:

- k SELECT inputs,
- $N = 2^k$ DATA OUTPUTs.

Select inputs choose one of the D_j to assert HIGH, all others will be LOW.

Decode + Encoder Loop-Up Table, ROM – Basic FPGA



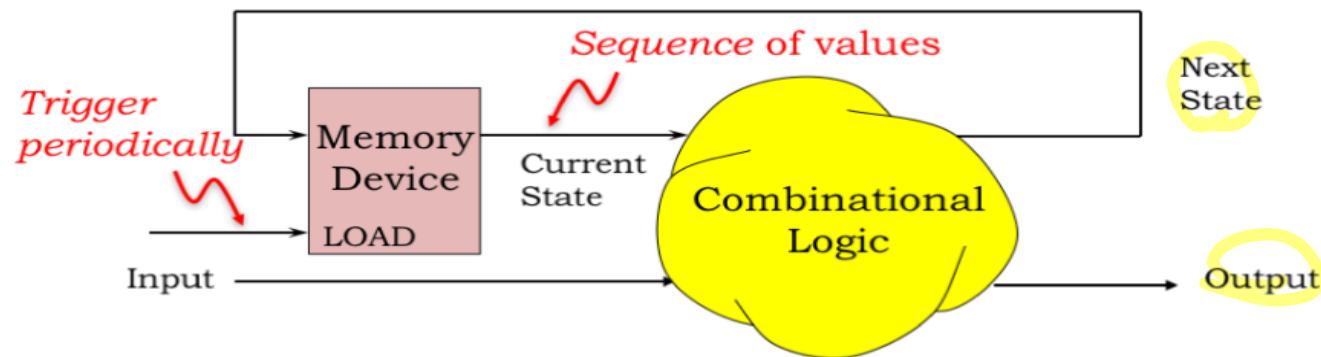
Recap – Combinational Logic

- Logic representation:
 - Truth Table, Boolean Equation, Schematic (NOT-AND-OR) – SOP
- Basic building block
 - NAND, NOR, MUX
 - How to represent NOT-AND-OR
- Logic Minimization – Demorgan law,
 - Boolean algebra
 - Karnaugh
 - Hazard & how to eliminate
- Look-up table (RAM) to implement the logic – encoder/decoder

Sequential Logic

Sequential logic = Combinational Logic + Memory Device

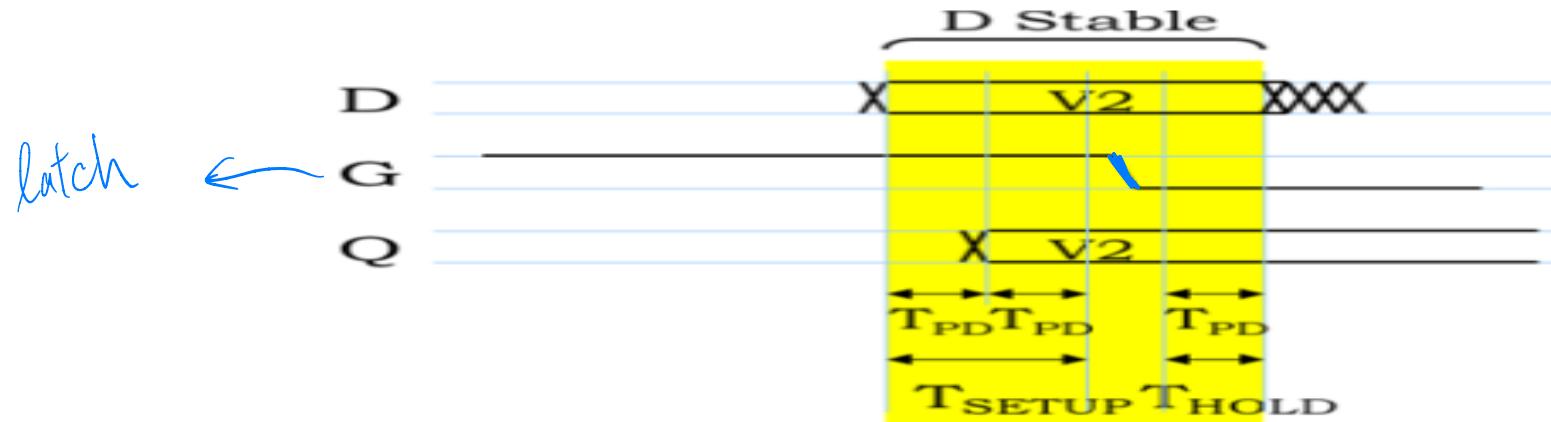
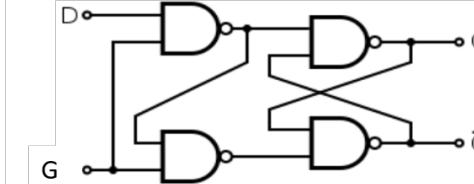
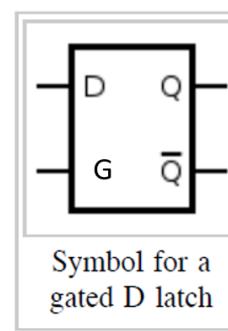
- A toggle switch –
 - If light is on -> turn off
 - If light is off -> turn on light
 - output changed by a input “event” (switch) and a current state (light)
 - An event (push the button, a transition not an level)
 - and a state need to memorized
- Output = $F(\text{Input}, \text{CurrentState})$ - combinational
- $\text{NextState} = Y(\text{Input}, \text{CurrentState})$ – combinational
- $\text{CurrentState} = \text{NextState} @ \text{Trigger (Clock)}$ – memory device



{ mealy
moore } machine

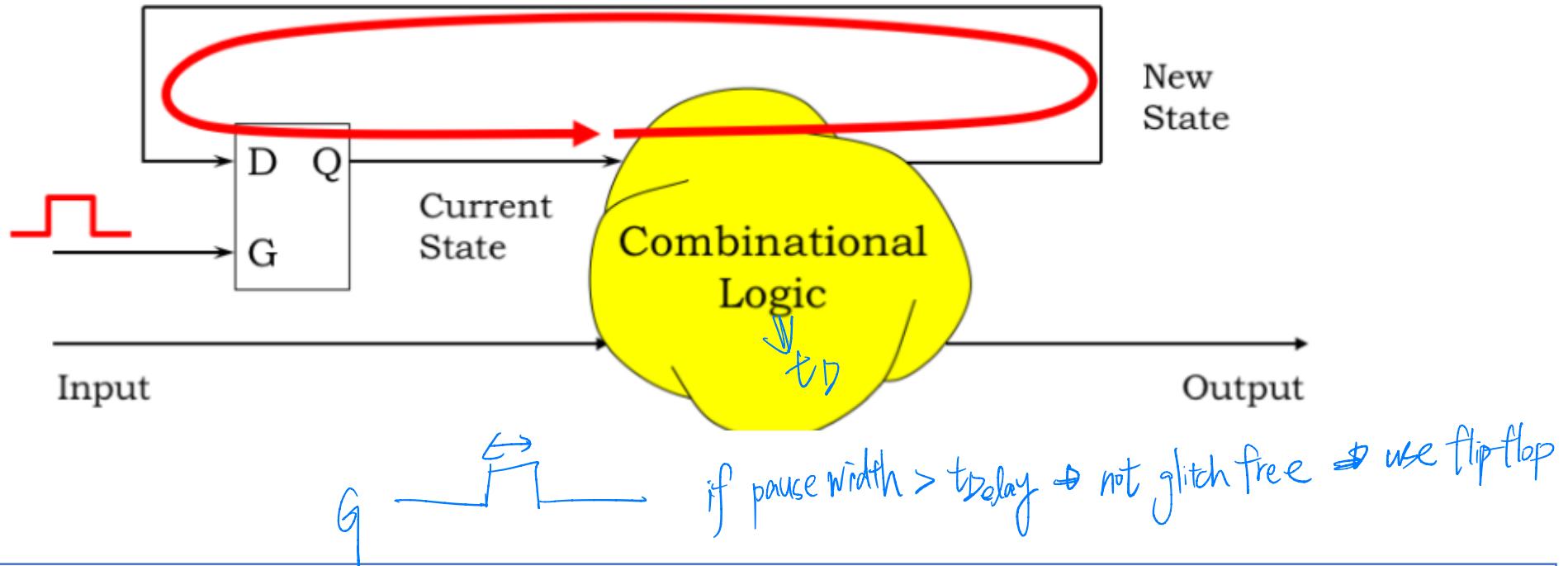
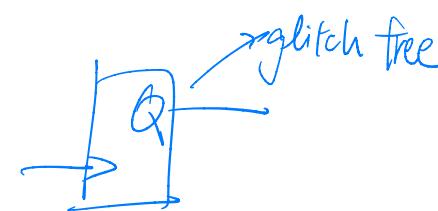
Memory Devices – Latch

D-latch truth table				
G	D	Q	\bar{Q}	Comment
0	X	Q_{prev}	\bar{Q}_{prev}	No change
1	0	0	1	Reset
1	1	1	0	Set



- **Setup Time (T_{setup})** = $2T_{\text{pd}}$: interval prior to G transition for which D must be stable and valid.
- **Hold Time (T_{hold})** = T_{pd} : interval following G transition for which D must be stable & valid.

Memory Devices – Latch

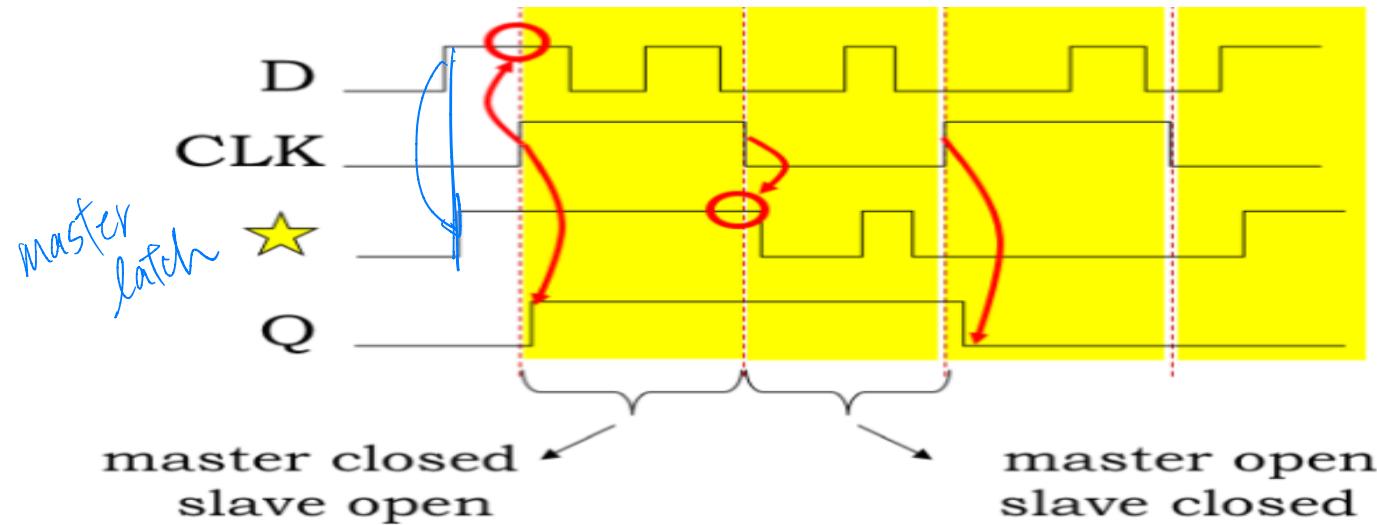
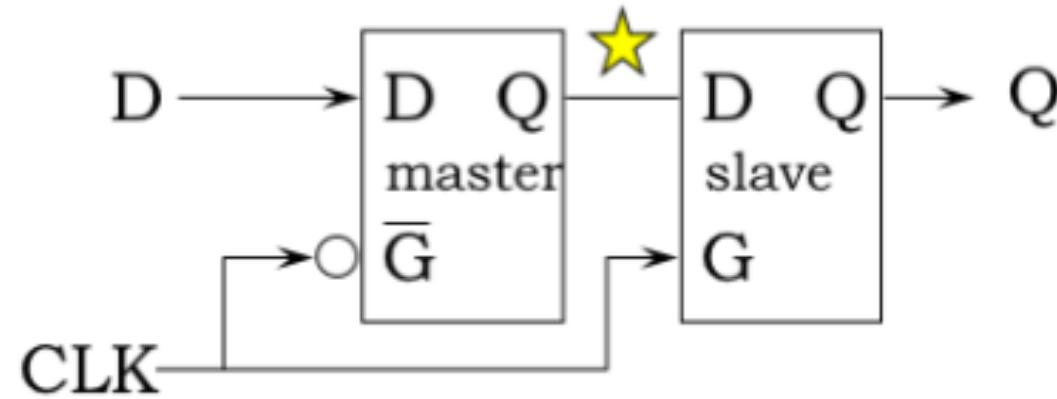


When G=1, latch is Transparent

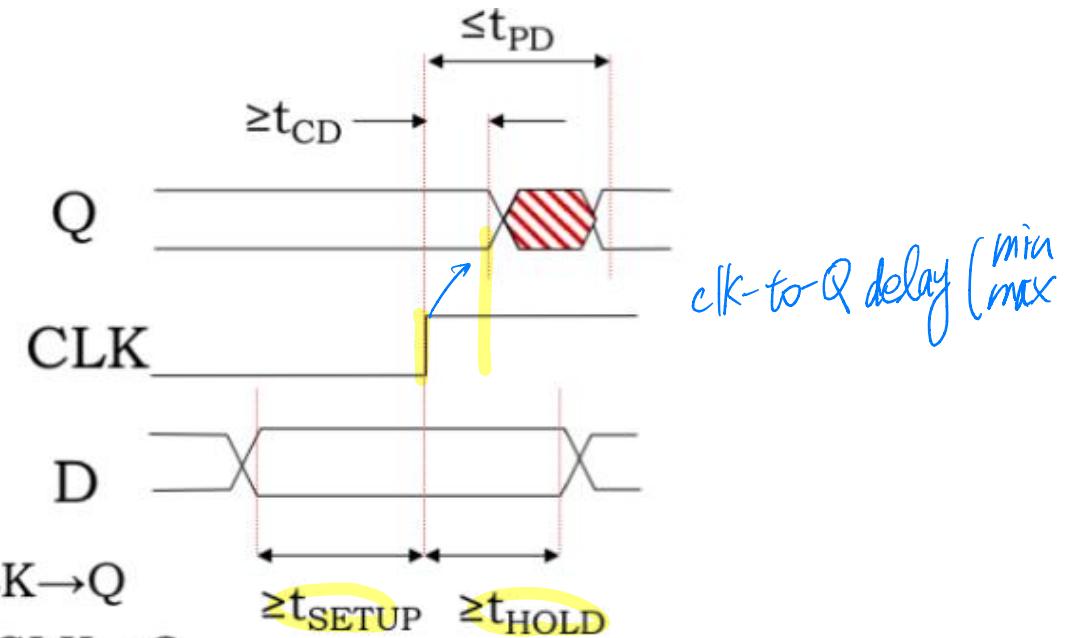
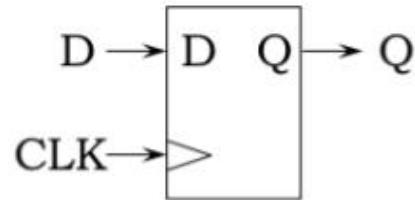
What is the G pulse-width should be?

so the glitch from the combination logic won't create glitch on Q output?

Memory Devices – Flip-flop (edge-triggered by a clock edge)



Memory Devices – Flip-flop Timing



t_{PD} : maximum propagation delay, CLK \rightarrow Q

t_{CD} : minimum contamination delay, CLK \rightarrow Q

t_{SETUP} : setup time

guarantee that D has propagated through feedback path before master closes

t_{HOLD} : hold time

guarantee master is closed and data is stable before allowing D to change

Sequential Timing

$t_{CD,reg1}$: minimum contamination delay of reg1, CLK->Q

$t_{PD,reg1}$: maximum propagation delay of reg1, CLK ->Q

$t_{CD,L}$: minimum delay of combinational logic L

$t_{PD,L}$: maximum delay of combinational logic L

$t_{CD, Input}$: minimum arrival time of Input

$t_{PD, Input}$: maximum arrival time of Input

t_{CLK} : clock period.

Minimum delay (reg2: Hold time)

$$t_{CD,Input} + t_{CD,L} \geq t_{HOLD,reg2}$$

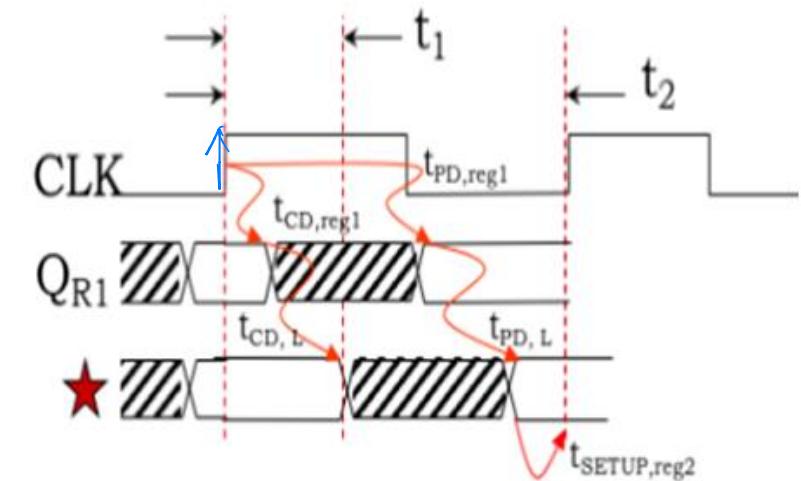
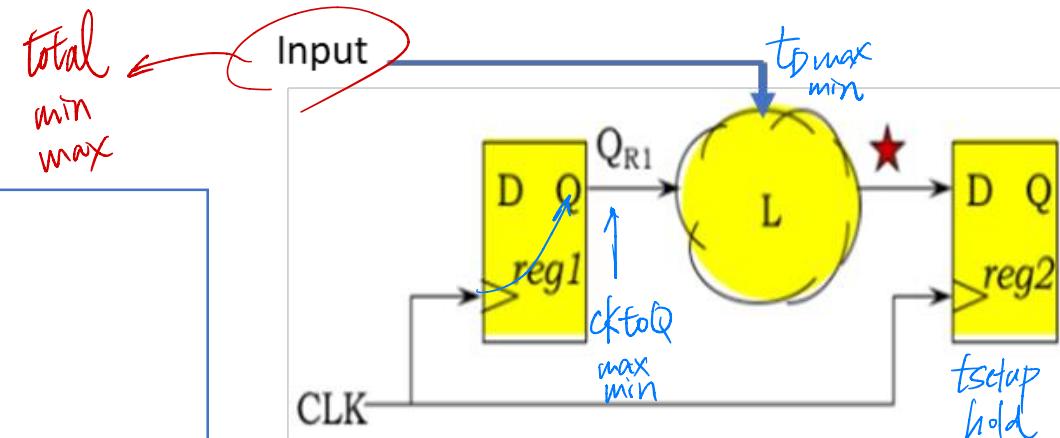
$$t_{CD,reg1} + t_{CD,L} \geq t_{HOLD,reg2}$$

Maximum delay (reg2: Setup time)

$$t_{PD,reg1} + t_{PD,L} + t_{SETUP,reg2} \leq t_{CLK}$$

$$t_{PD,Input} + t_{PD,L} + t_{SETUP,reg2} \leq t_{CLK}$$

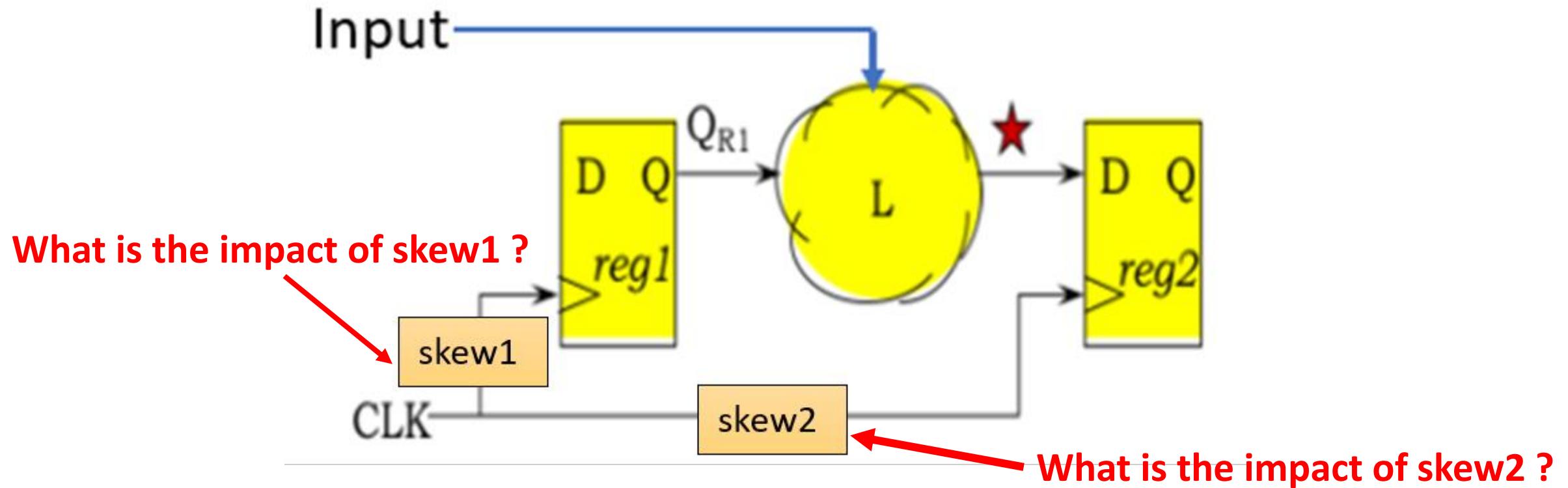
1. How much time for useful work (i.e. combination logic L) $\xrightarrow{t_{setup}}$
2. Constraints on $t_{CD,L}$? $\xrightarrow{-t_{hold}}$
3. Minimum clock period Tclock?
4. Setup, Hold time for the input ?



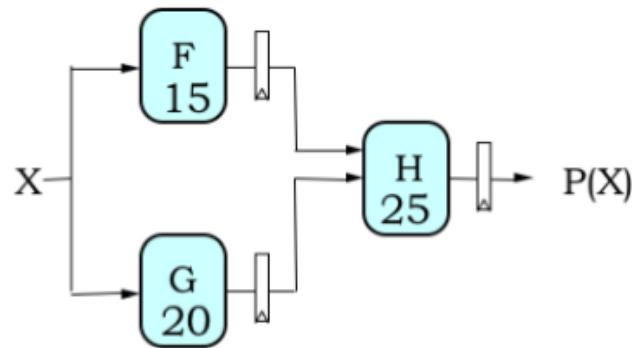
$$t_1 = t_{CD,reg1} + t_{CD,L} \geq t_{HOLD,reg2}$$

$$t_2 = t_{PD,reg1} + t_{PD,L} + t_{SETUP,reg2} \leq t_{CLK}$$

Clock Skew

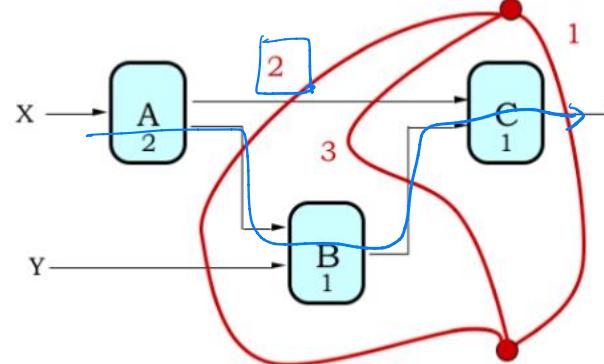


Pipeline – Latency v.s. Throughput

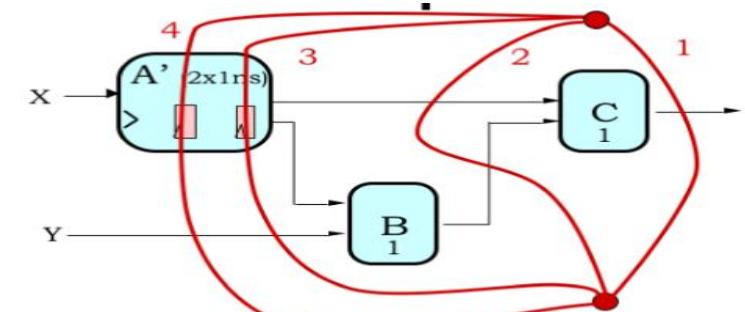


	<u>latency</u>	<u>throughput</u>
unpipelined	45	$1/45$
2-stage pipeline	50	$1/25$

Pipeline partition



- Pipelined version of A



4-stage pipeline, throughput=1

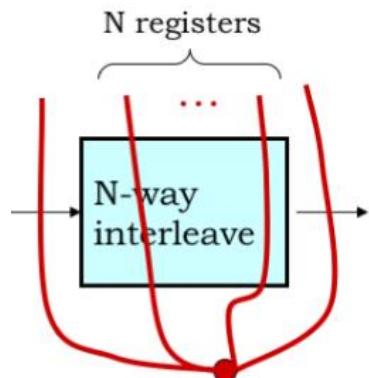
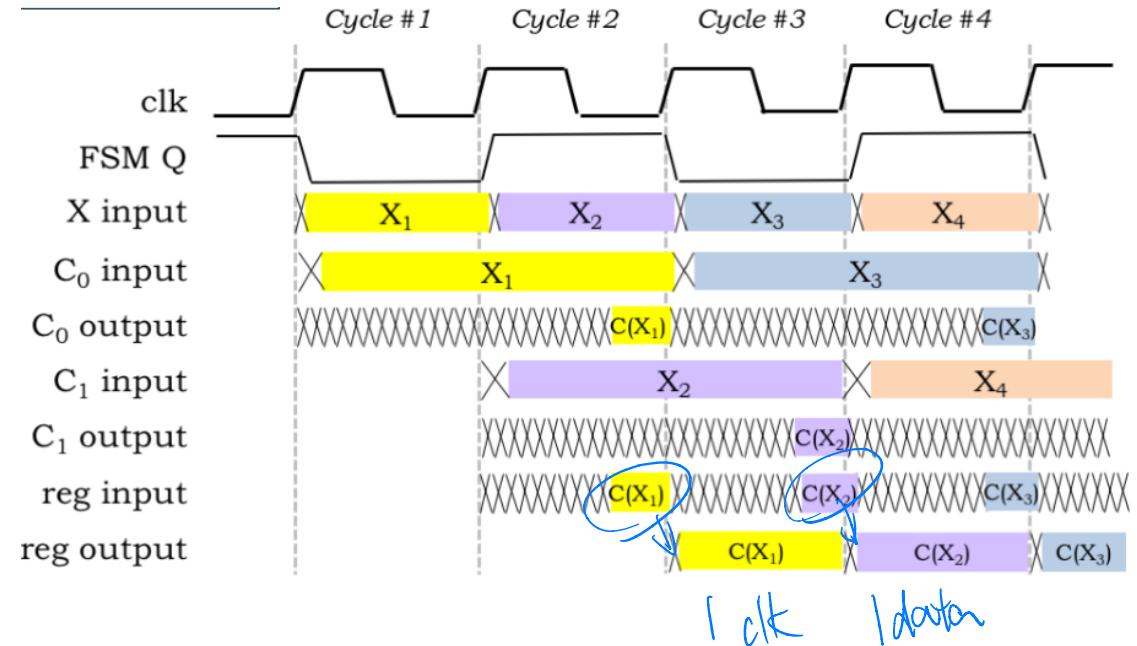
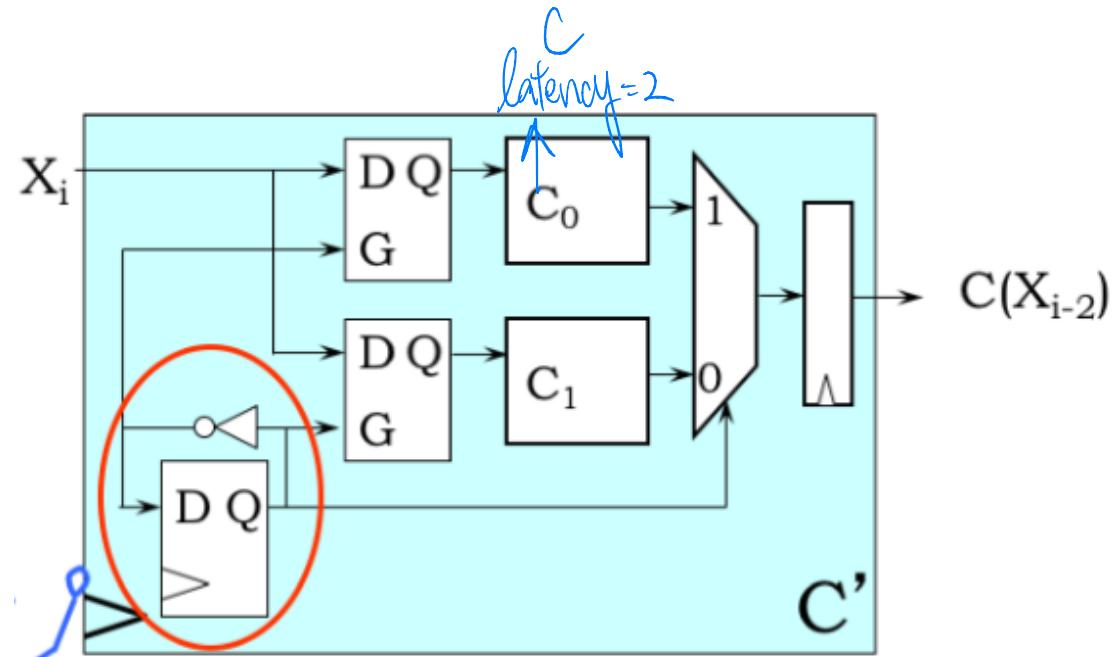
	LATENCY	THROUGHPUT
0-pipe:	4	$1/4$
1-pipe:	4	$1/4$
2-pipe:	4	$1/2$
3-pipe:	6	$1/2$

4-stage pipeline

Latency : 4 · Throughput= 1

But how to pipeline A block ?

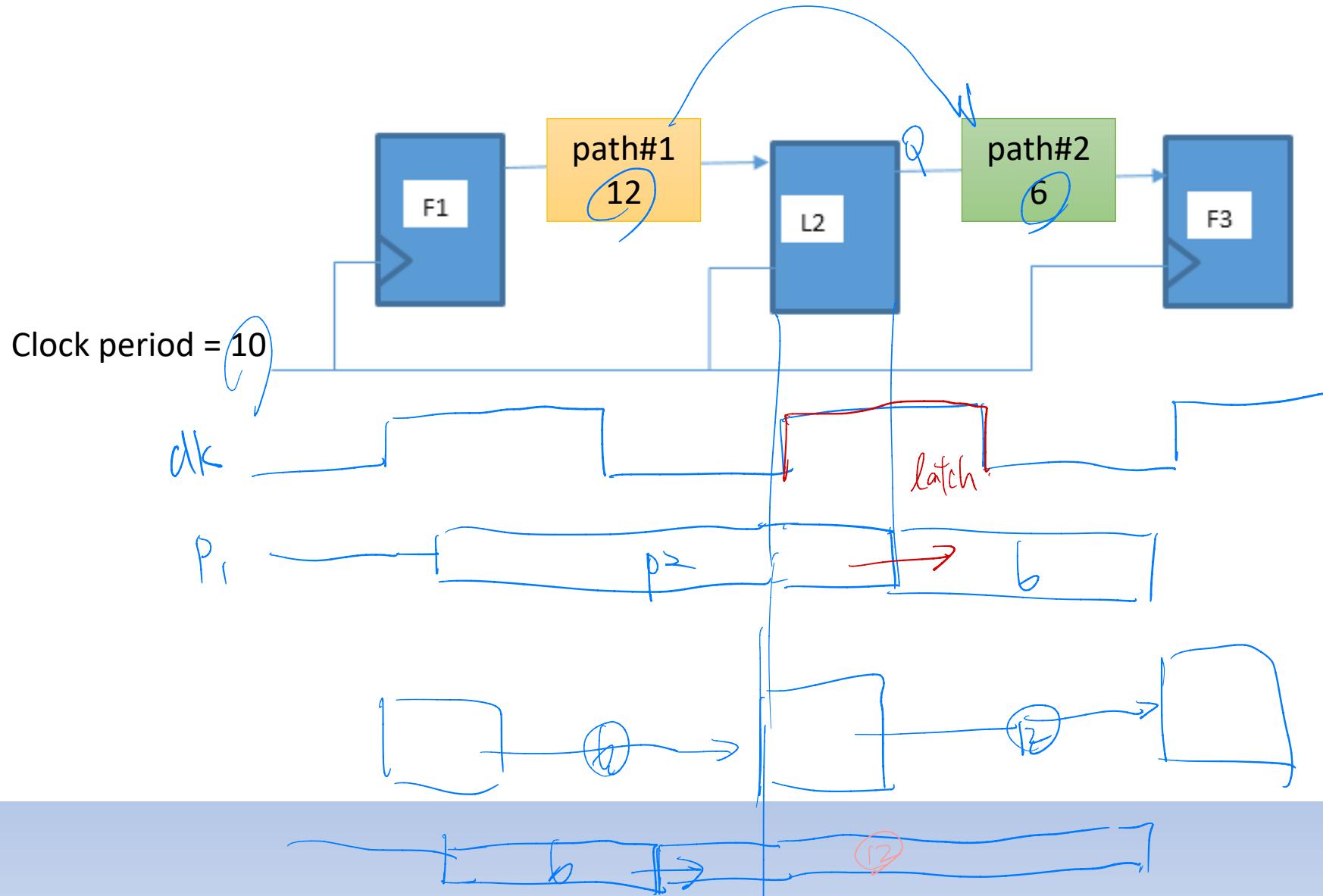
If no pipeline version -> Interleaving



N-way interleaving = N pipeline Stages

n instances
n throughput

Time Borrowing



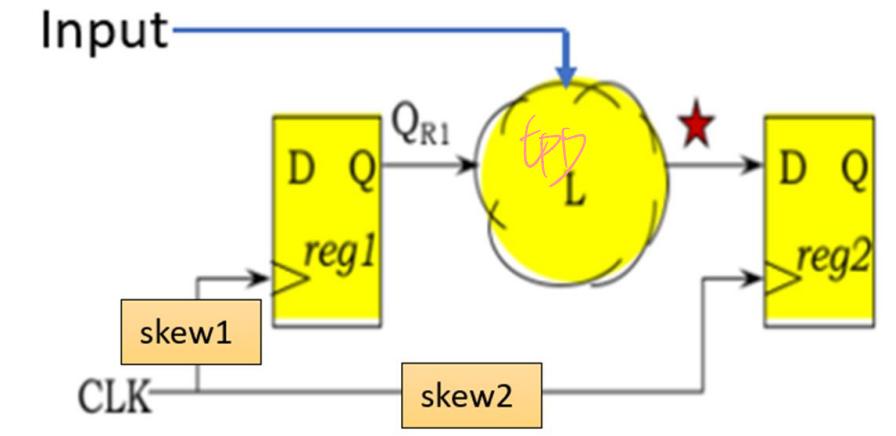
Recap – Sequential Logic

- Introduce the concept of state and memory device
- Latch & Flip-flop and its timing requirement – setup/hold
- Sequential timing – path delay to meet flip-flop setup/hold
 - Clock skew
- Introduce Pipeline & Pipeline timing
 - Latency v.s. Throughput
- Interleave to increase throughput
- Time borrowing

Fj Shie
(Fayyaz)
→ brain

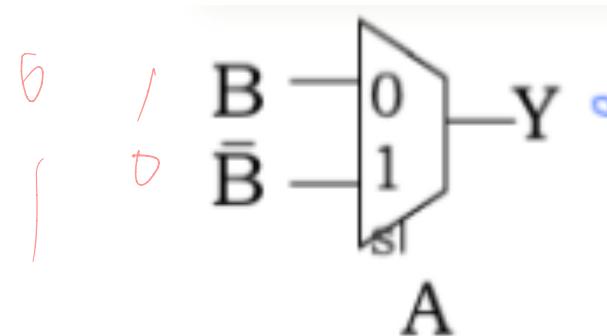
Question#1

1. tPD,L could not meet the following equation, we can choose add delay at skew1 or skew2 ?
 $tPD,reg1 + tPD,L + tSETUP,reg2 \leq tCLK$



Question#2

2. What gate realized by the MUX on the left ? AND, OR, NOT, XOR, NOR



A truth table for a logic function. The columns are labeled A and B, and the rows are labeled 00, 01, 10, and 11. The output column is labeled Y. The values in the Y column are 0, 1, 0, and 1 respectively. To the right of the table, there is handwritten text: "Y = A + B" with a crossed-out "OR" symbol, indicating that the function is actually an XOR gate.

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

Question#3

- How we change the following design to do time borrowing in the following case?

