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)
4. System/Application Level Optimization

Agenda

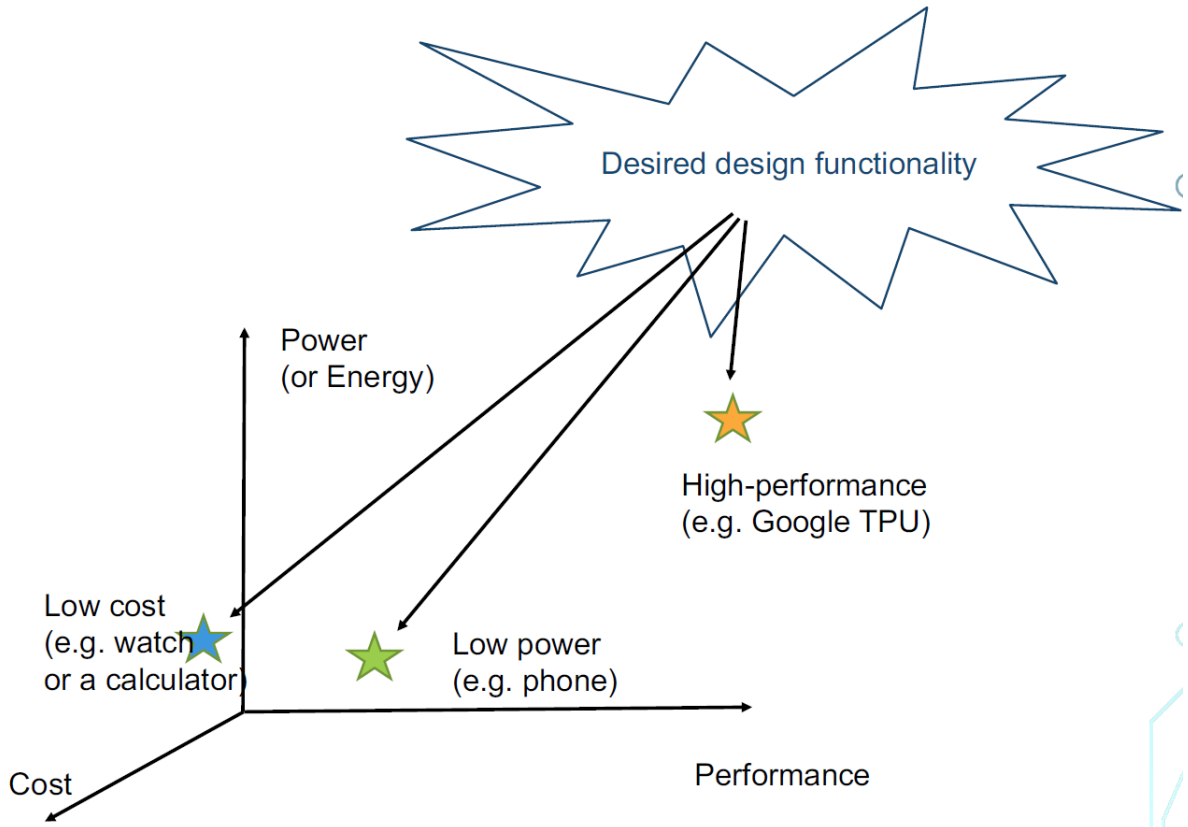
1. Design Background
2. Combinational Design
3. Sequential Design

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**
  - 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)
  - 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
  - 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
  - Sometime 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

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

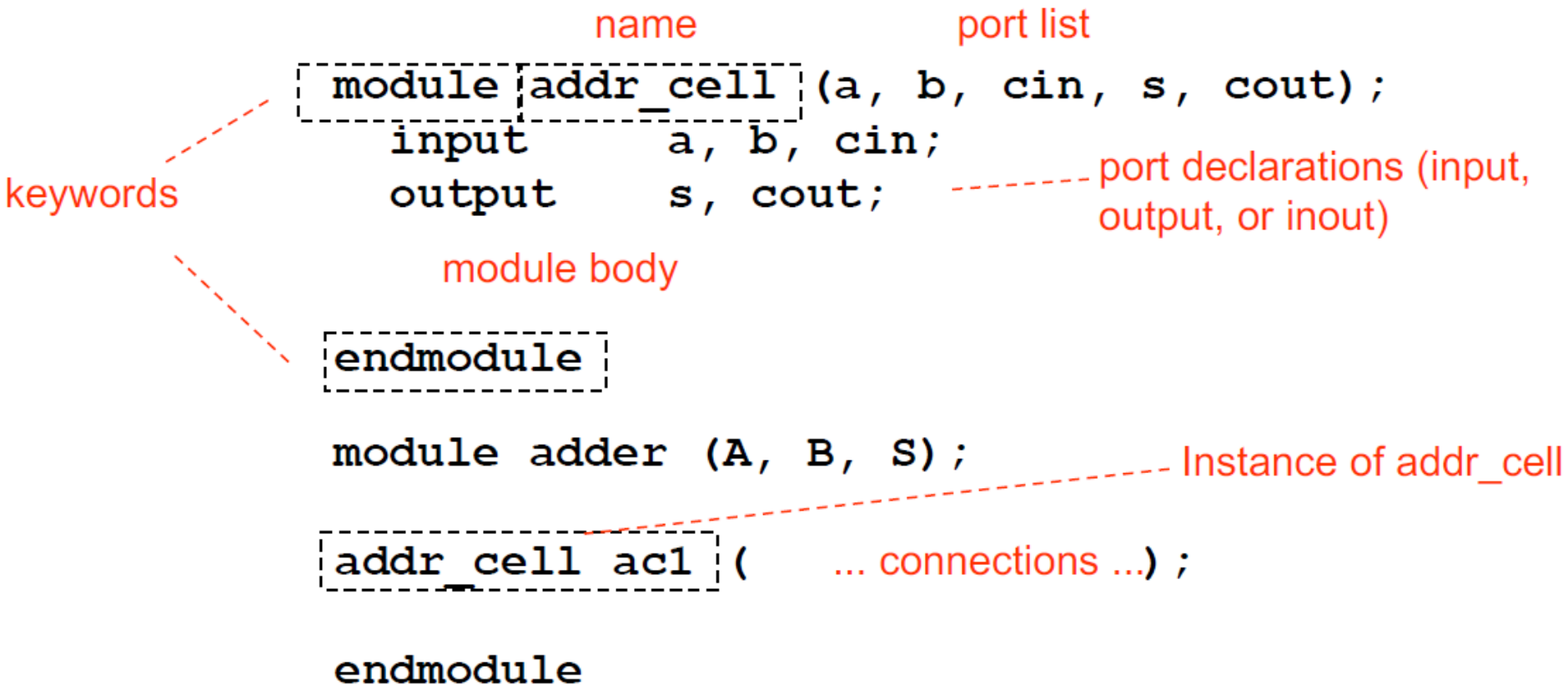
Module – A component in a circuit

- **Structural Verilog**
  - 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**
  - 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

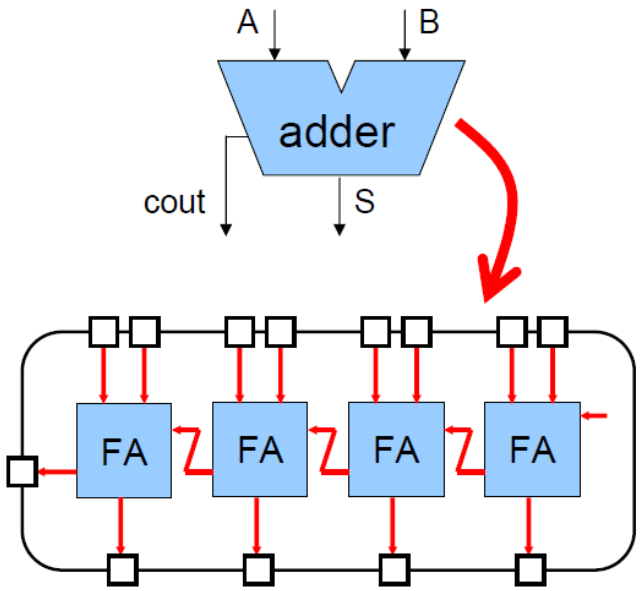
- Hierarchy of modules. Top-level module is your entire design (or the environment to test your design).

Verilog Modules (Declaration) and Instantiation (Instance)



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
```

Carry Chain

Module are connected together with nets

Ports are attached to nets by name

```
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

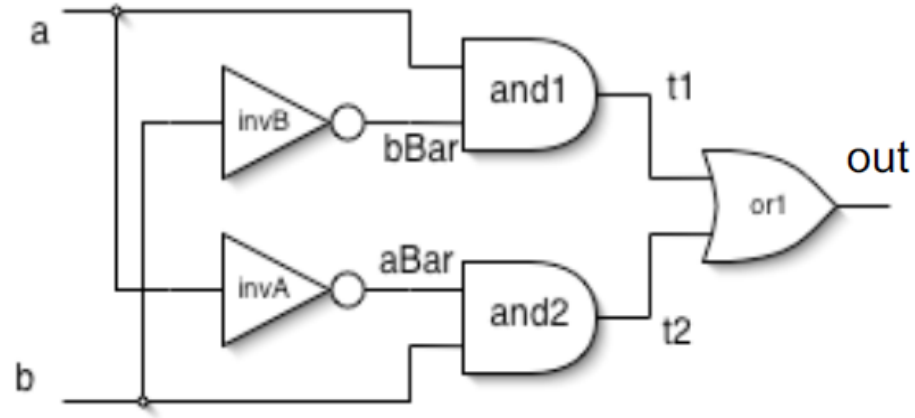
port list

port declarations

internal signal declarations

Built-in gates

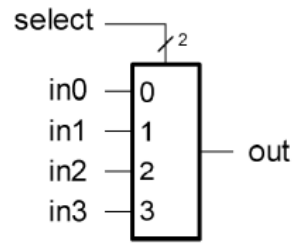
instances



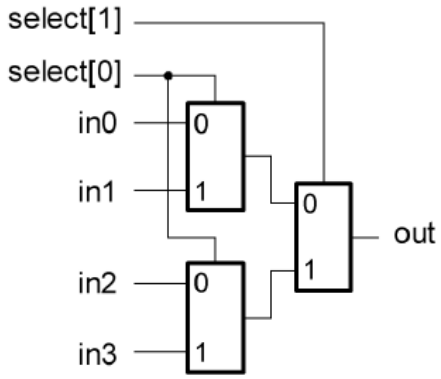
Interconnections (note output is first)

Instance name

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;
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
```

Signal array. Declares select[1], select[0]

Named ports. Highly recommended.

Continuous Assignment Examples

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;

assign R = (a == 1'b0) ? X : Y;

assign P = 8'hff;

assign P = X * Y;

assign P[7:0] = {4{X[3]}, X[3:0]};

assign {cout, R} = X + Y + cin;

assign Y = A << 2;

assign Y = {A[1], A[0], 1'b0, 1'b0};
```

use of bit-wise Boolean operators

example reduction operator

conditional operator

example constants

arithmetic operators (use with care!)

(ex: sign-extension)

bit field concatenation

bit shift operator

equivalent bit shift

Non-Continuous (Procedure) Assignments (always @)

```
module and_or_gate (out, in1, in2, in3);
  input     in1, in2, in3;
  output    out;
  reg       out;

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

endmodule
```

"reg" type declaration. Not really a register in this case. Just a Verilog idiosyncrasy.

keyword

"sensitivity" list, triggers the action in the body.

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
```

keyword

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;
  input [N-1:0] A;
  input [N-1:0] B;
  output [N:0] R;
  wire [N:0] C;

  genvar i;

  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
```

Declare a parameter with default value.

Note: this is not a port. Acts like a "synthesis-time" constant.

Replace all occurrences of "4" with "N".

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
```

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

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

Basics

- Combinational logic:
  - 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

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

• Boolean Expression

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

INVERTER: $A \rightarrow Z = \bar{A}$

A	Z
0	1
1	0

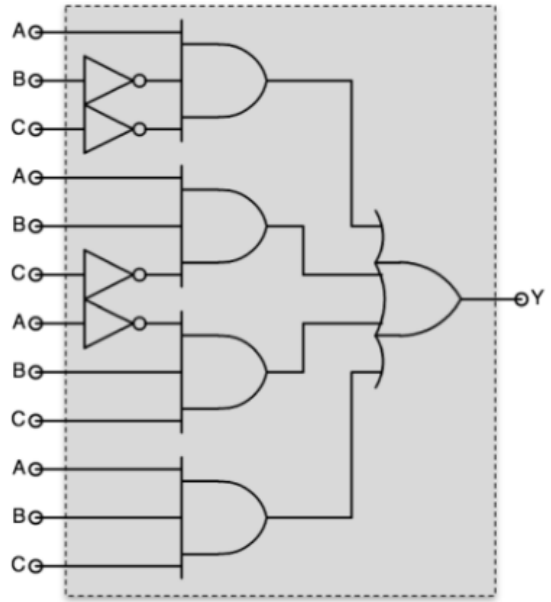
AND: $A, B \rightarrow Z = A \cdot B$

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

OR: $A, B \rightarrow Z = A + B$

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

Schematic View

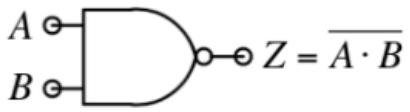


- Sum-of-Product
- 3 levels of INV, AND, OR
- What if # inputs increase?

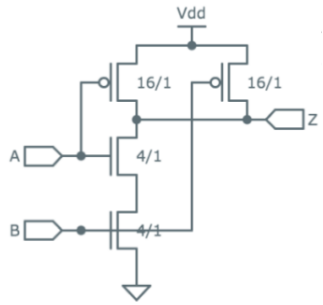
Building Blocks – NAND, NOR, XOR

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

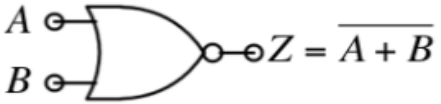
NAND (not AND)



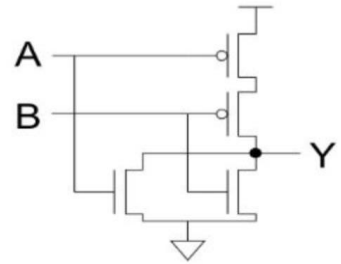
A	B	Z
0	0	1
0	1	1
1	0	1
1	1	0



NOR (not OR)

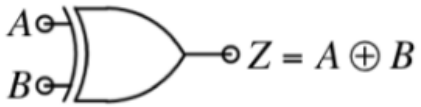


A	B	Z
0	0	1
0	1	0
1	0	0
1	1	0

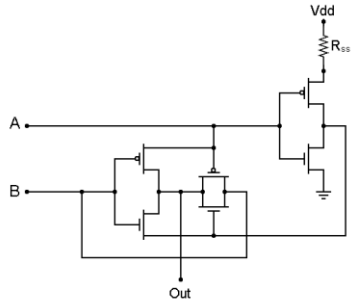


- **XOR – parity and arithmetic logic**
- **Associative**

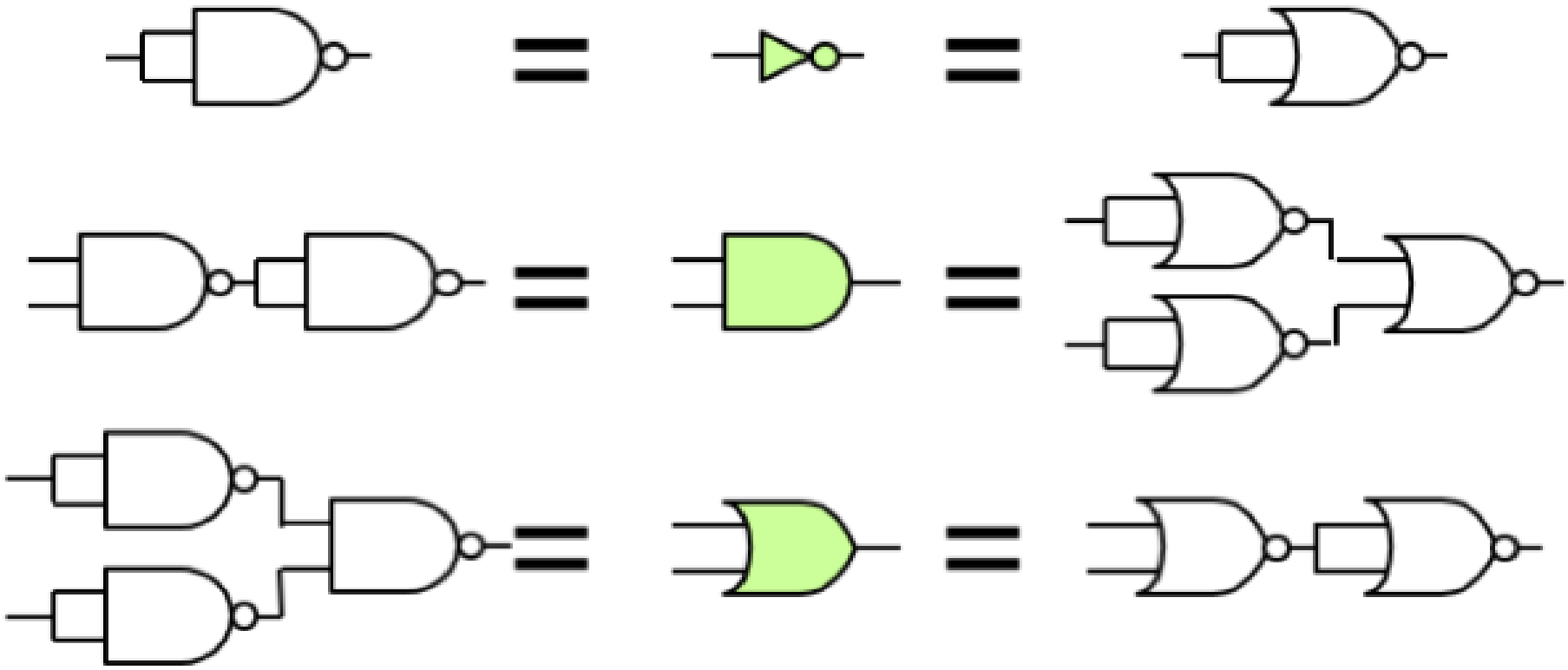
XOR (exclusive OR)



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



Universal Building Blocks – NAND, NOR

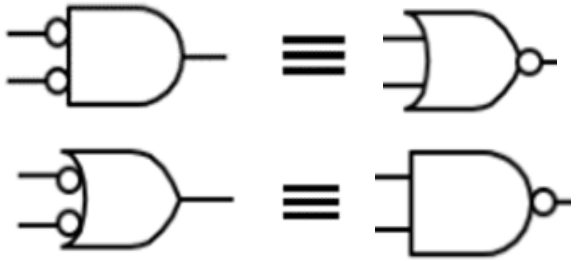


Demorgan's Laws

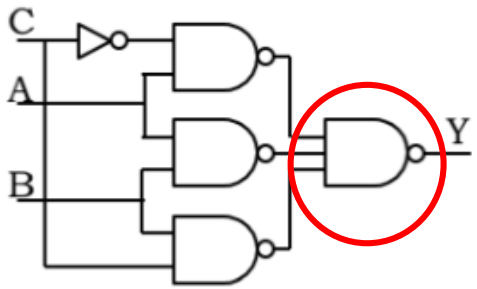
Demorgan's Laws:

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

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

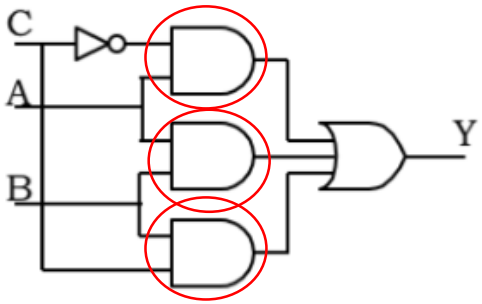


NAND-NAND



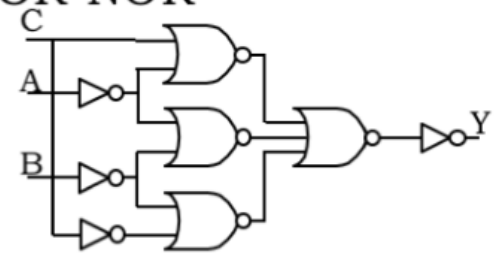
$\overline{AB}=\overline{A}+\overline{B}$

"Pushing Bubbles"

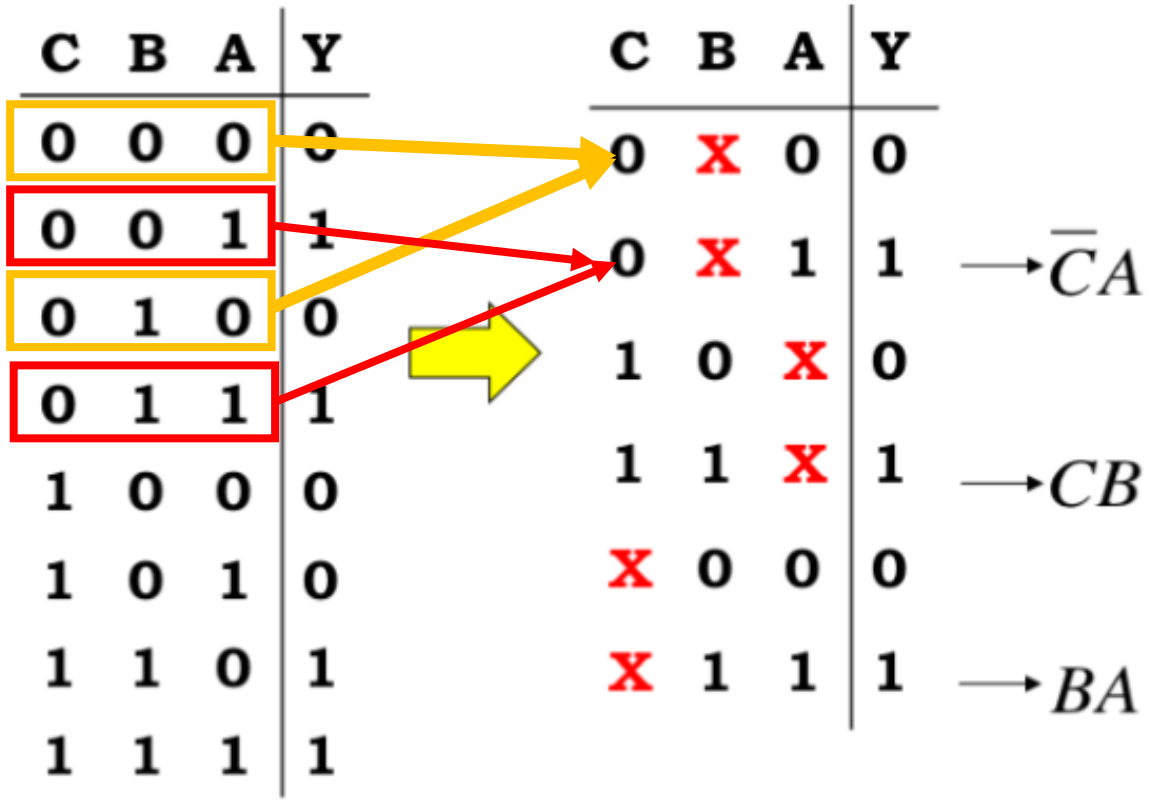


$\overline{\overline{A}\,\overline{B}}=\overline{A+B}$

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	$\rightarrow \overline{C}A$
1	0	X	0	
1	1	X	1	$\rightarrow CB$
X	0	0	0	
X	1	1	1	$\rightarrow BA$

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

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

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

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

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

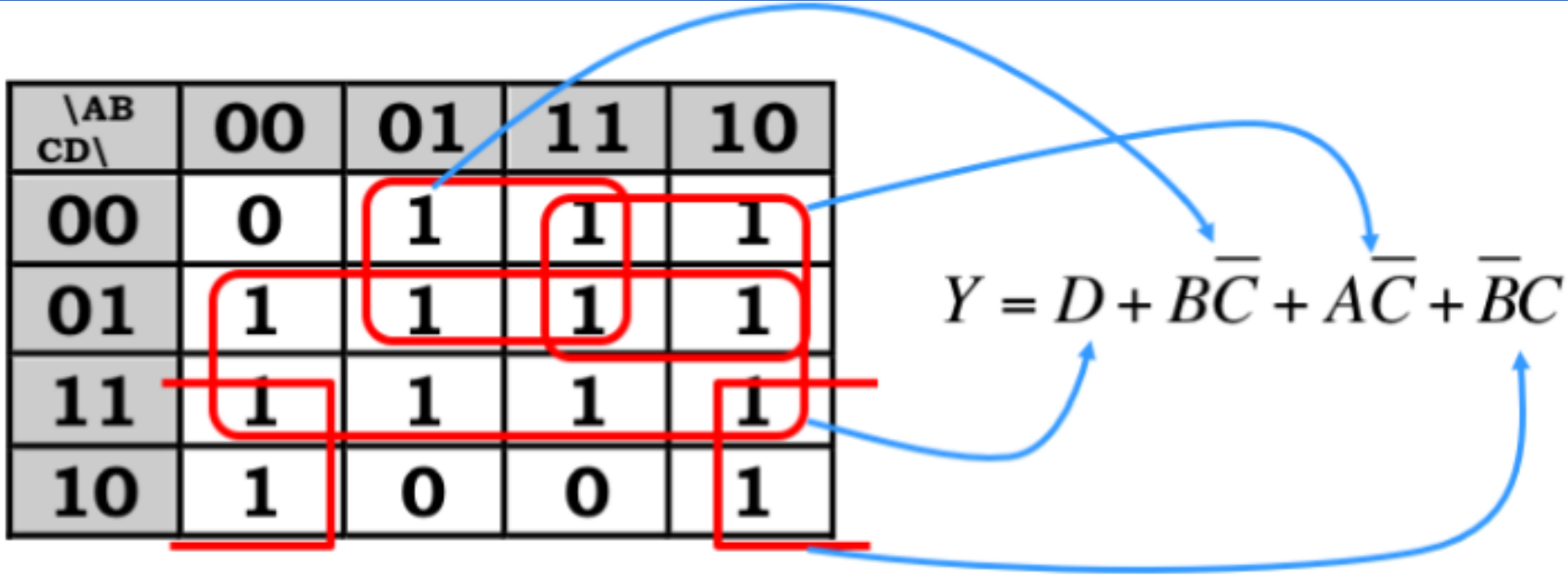
$$Y=\bar{C}\bar{B}A+CB+\bar{C}BA$$

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

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”

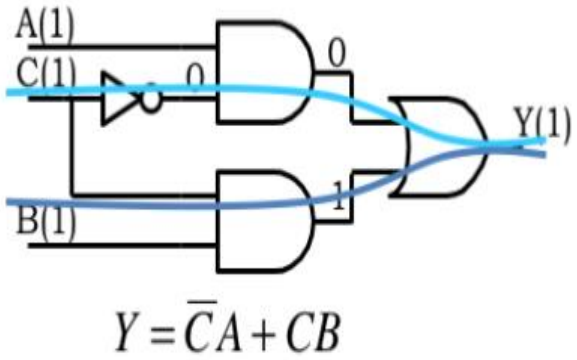


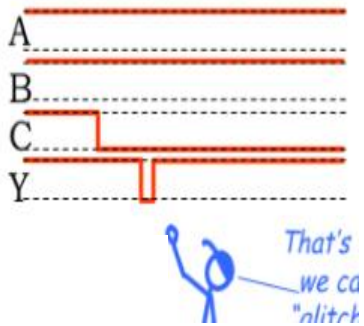
\AB CD\	00	01	11	10
00	0	1	1	1
01	1	1	1	1
11	1	1	1	1
10	1	0	0	1

$$Y = D + B\overline{C} + A\overline{C} + \overline{B}C$$

“Hazard” circuit

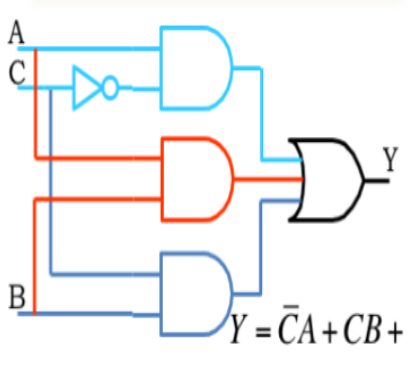
- “glitch” or “hazard” circuit

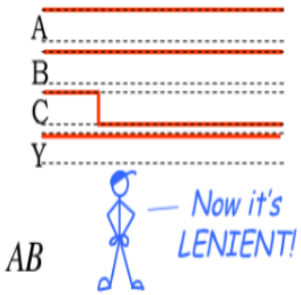




- To make glitch-free, include product terms link for all prime implicants

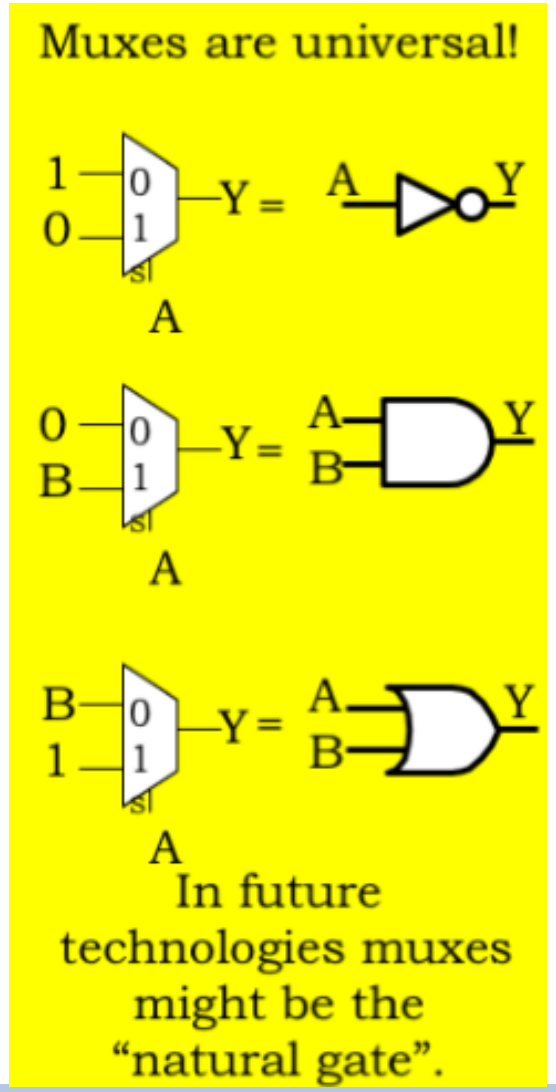
C\AB	00	01	11	10
0	0	0	1	1
1	0	1	1	0

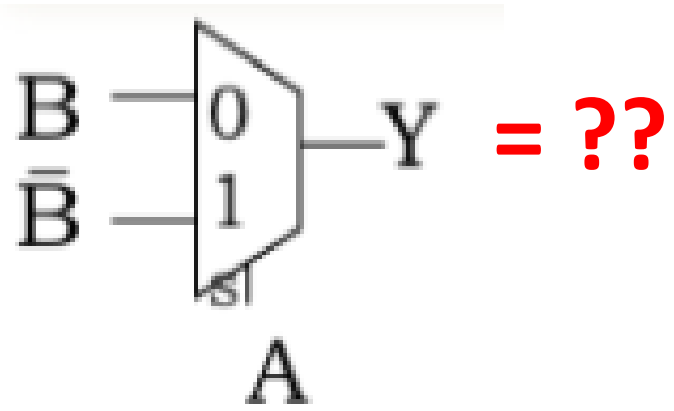




- For synchronous design, glitch does not matter as long as settled before sampled
- However, it does increase delay for settling the glitch

Another Universal Gate - MUX

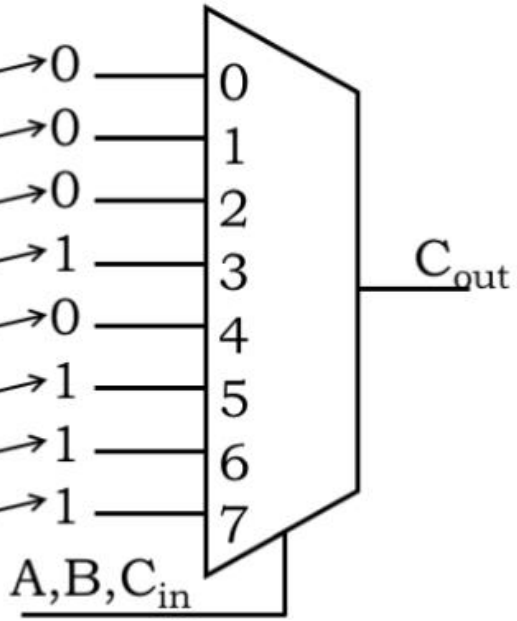


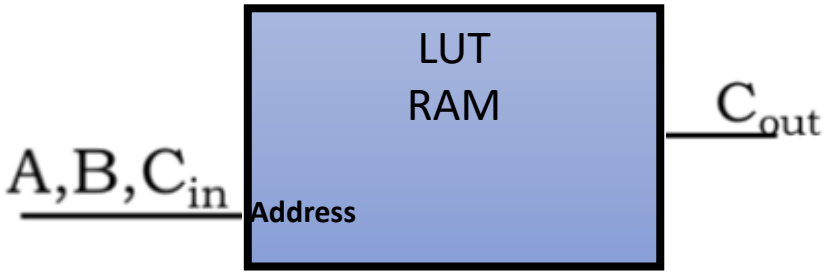


Encoder

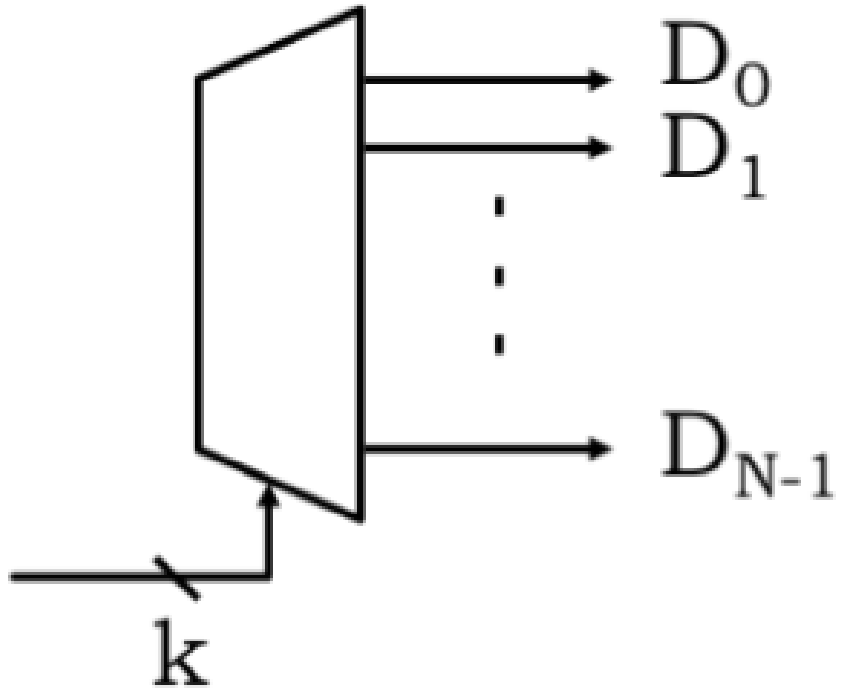
Full-Adder
Carry Out Logic

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





Decoder

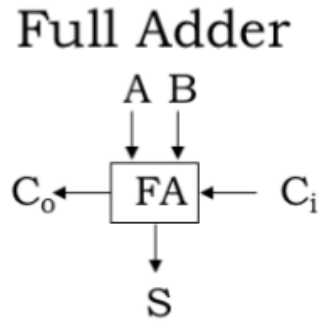


DECODER:

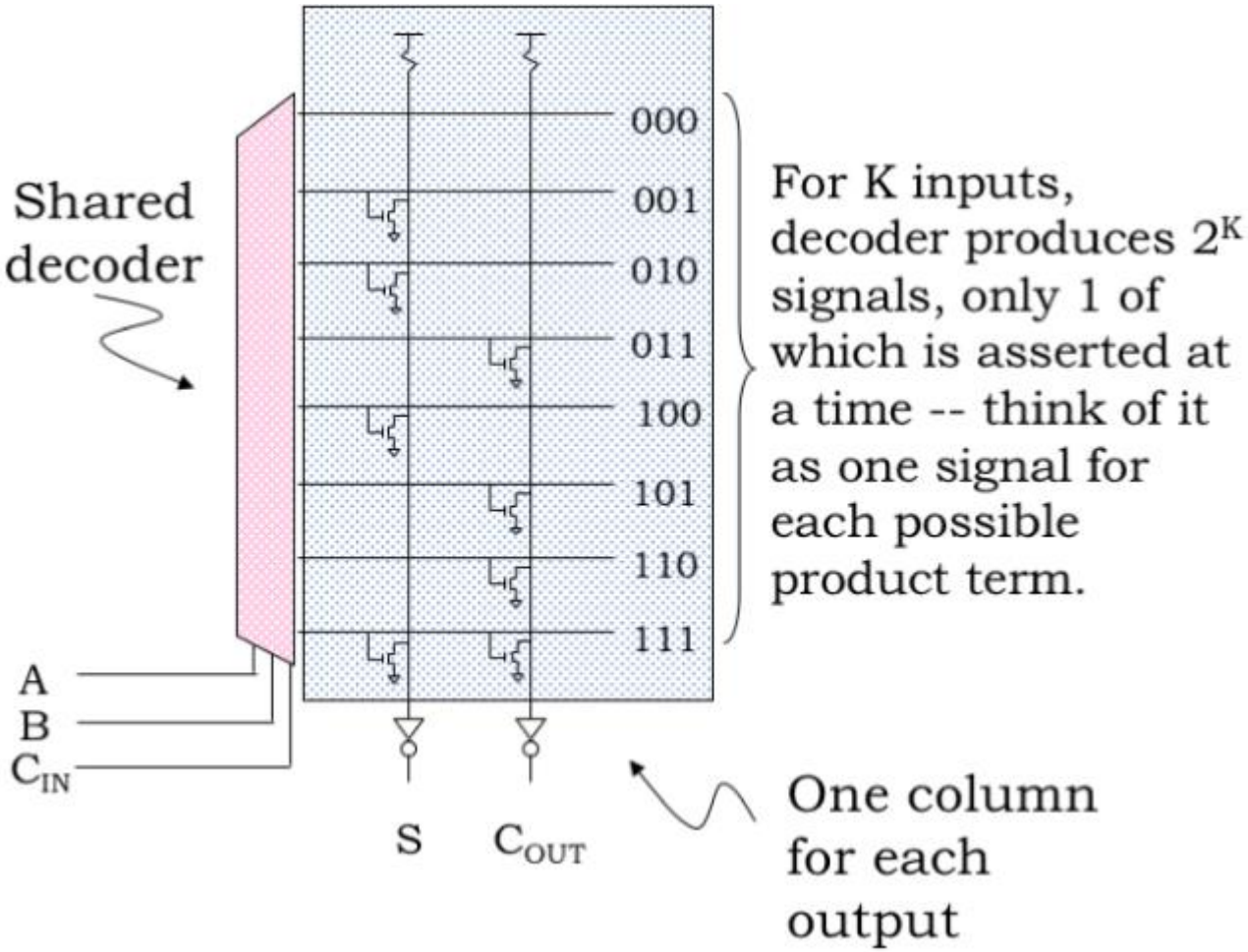
- k SELECT inputs,
- $N = 2^k$ DATA OUTPUTs.

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

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



A	B	C_i	S	C_o
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1



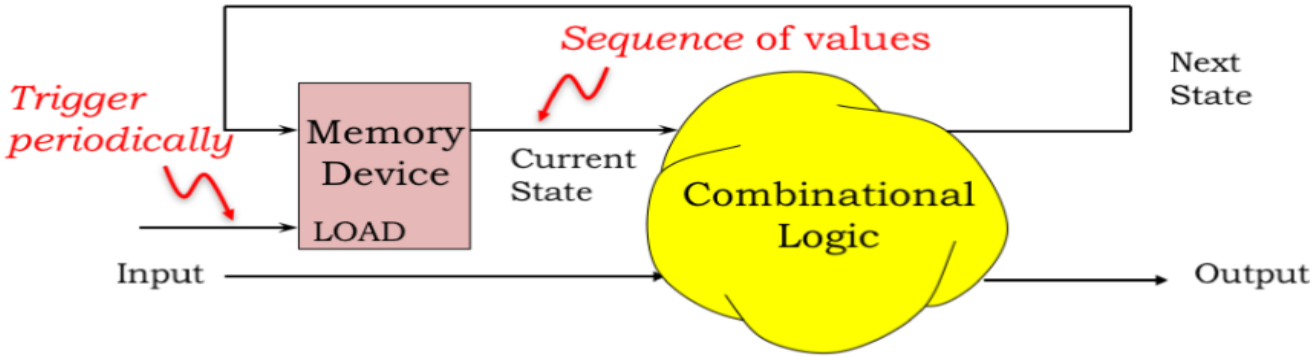
Recap – Combinational Logic

- 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

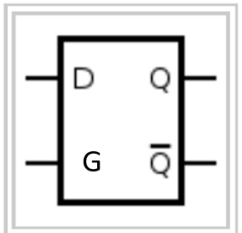
- A toggle switch –
  - If light is on -> turn off
  - If light is off -> turn on light
  - output changed by a input “event” (switch) and a current state (light)
  - An event (push the button, a transition not an level)
  - and a state need to memorized
- Output = F(Input, CurrentState) - combinational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



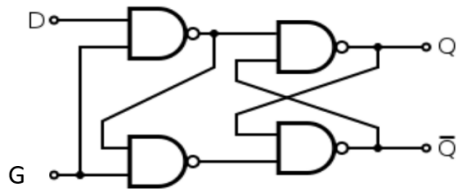
Memory Devices – Latch

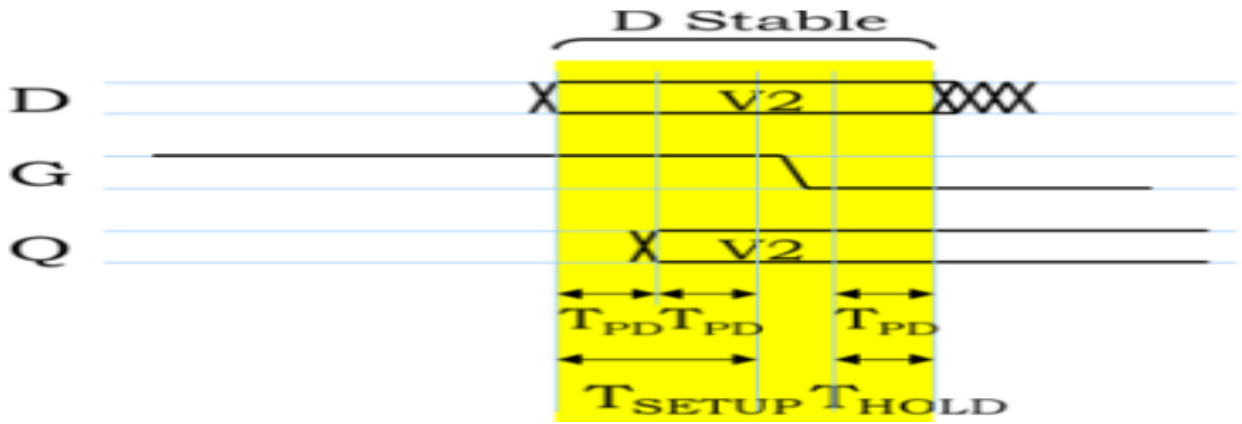
D-latch truth table

G	D	Q	$\overline{Q}$	Comment
0	X	Q_{prev}	$\overline{Q}_{prev}$	No change
1	0	0	1	Reset
1	1	1	0	Set



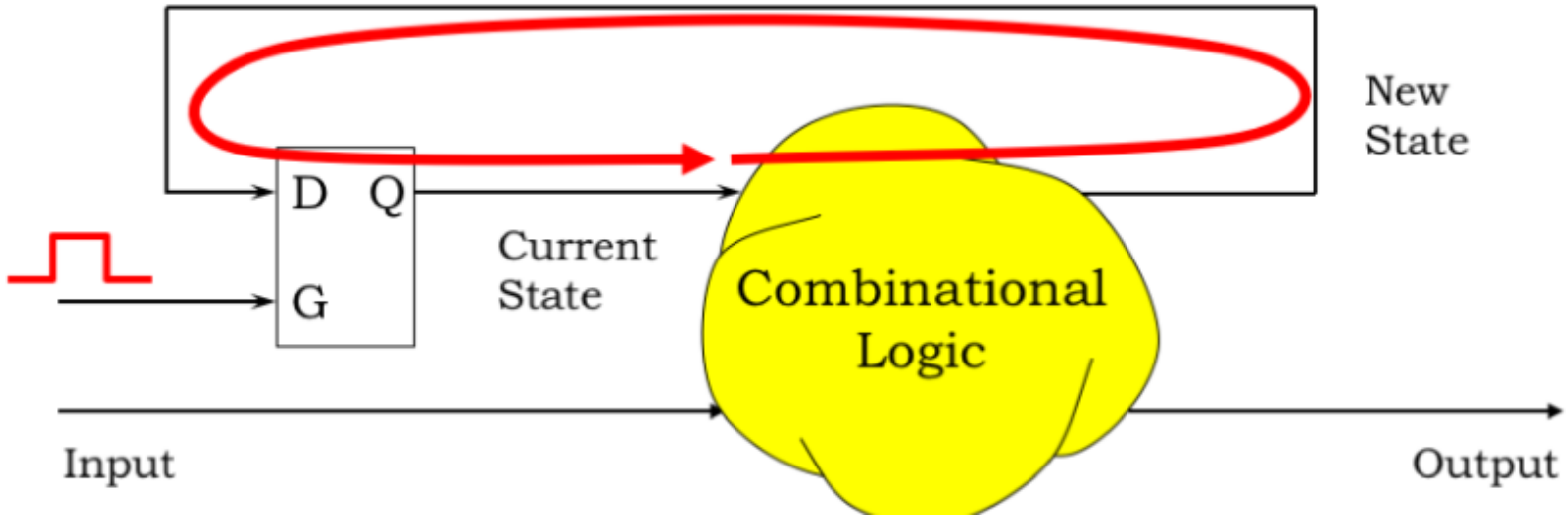
Symbol for a gated D latch





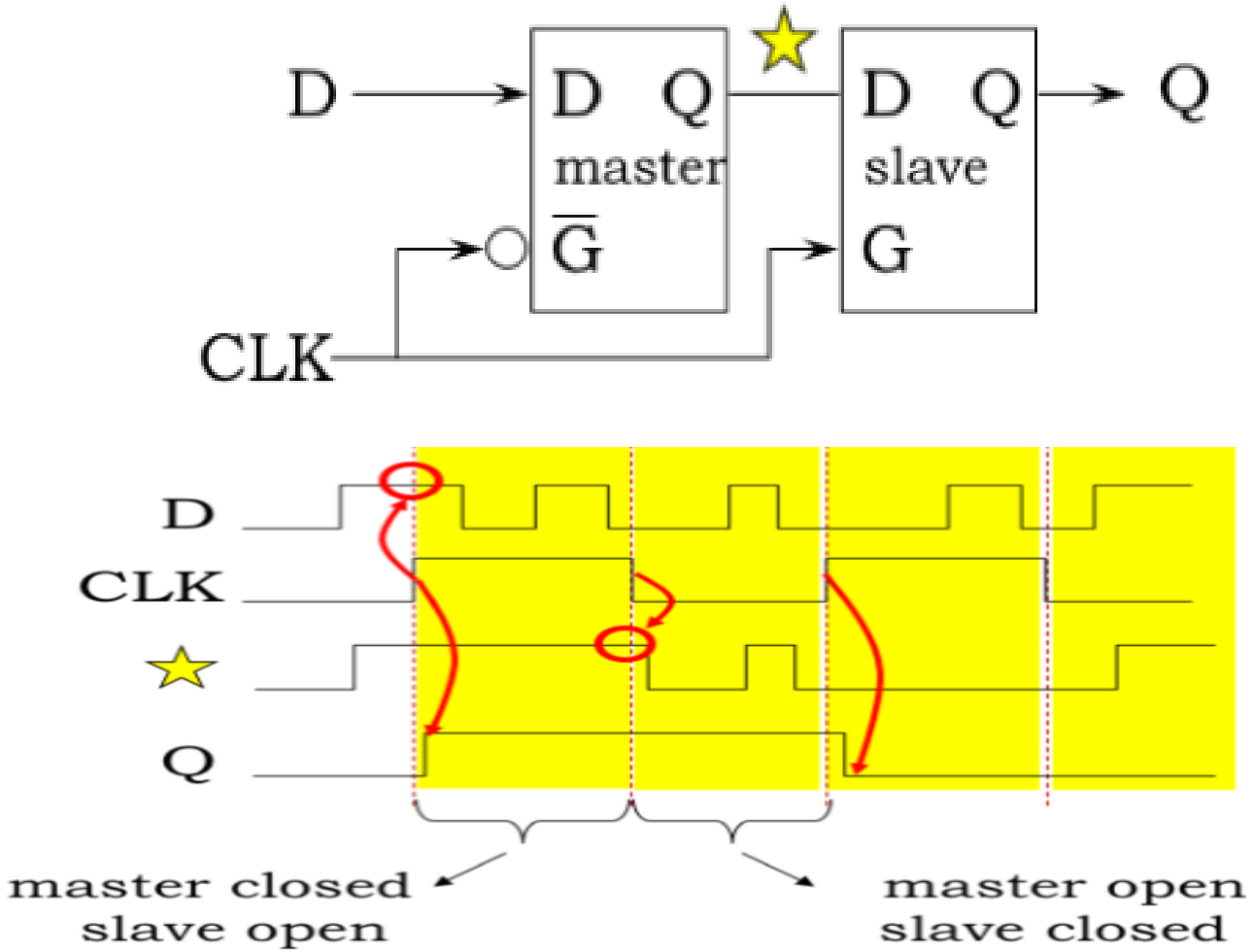
- **Setup Time (T_{setup})**= 2Tpd: interval prior to G transition for which D must be stable and valid.
- **Hold Time (T_{hold})** = Tpd: interval following G transition for which D must be stable & valid.

Memory Devices – Latch

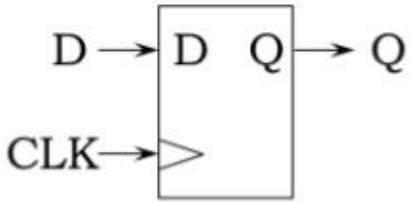


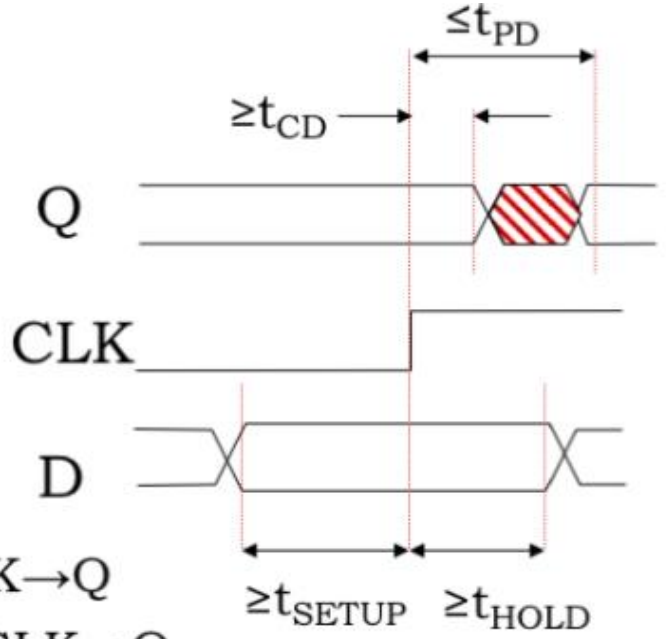
When G=1, latch is Transparent
What is the G pulse-width should be?
so the glitch from the combination logic wont’ create glitch on Q output?

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



Memory Devices – Flip-flop Timing





t_{PD}: maximum propagation delay, CLK→Q

t_{CD}: minimum contamination delay, CLK→Q

t_{SETUP}: setup time

guarantee that D has propagated through feedback path before master closes

t_{HOLD}: hold time

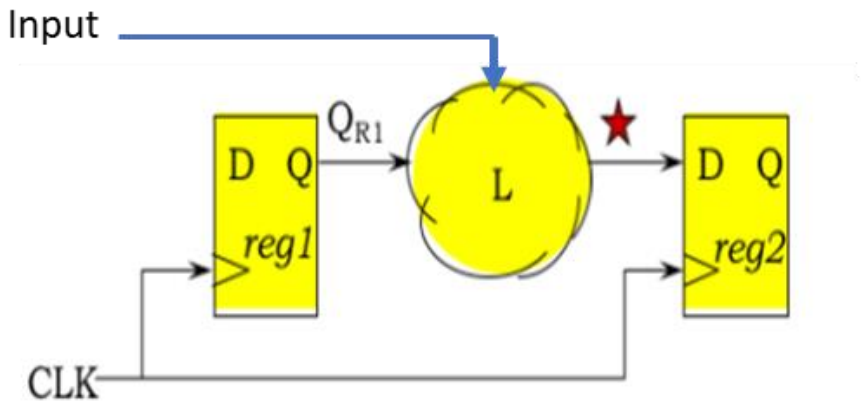
guarantee master is closed and data is stable before allowing D to change

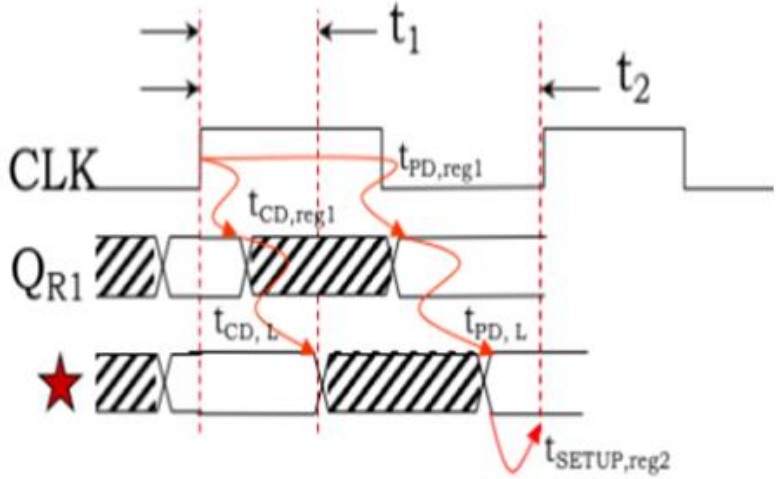
Sequential Timing

tCD,reg1 : minimum contamination delay of reg1, CLK->Q
tPD,reg1 : maximum propagation delay of reg1, CLK ->Q
tCD,L : minimum delay of combinational logic L
tPD,L : maximum delay of combinational logic L
tCD, Input: minimum arrival time of Input
tPD, Input: maximum arrival time of Input
tCLK : clock period.

Minimum delay (reg2: Hold time)
tCD,Input+ tCD,L >= tHOLD,reg2
tCD,reg1 + tCD,L >= tHOLD,reg2
Maximum delay (reg2: Setup time)
tPD,reg1 + tPD,L + tSETUP,reg2 <= tCLK
tPD,Input + tPD,L + tSETUP,reg2 <= tCLK

1. How much time for useful work (i.e. combination logic L)
2. Constrains on tCD,L ?
3. Minimum clock period Tclock?
4. Setup, Hold time for the input ?

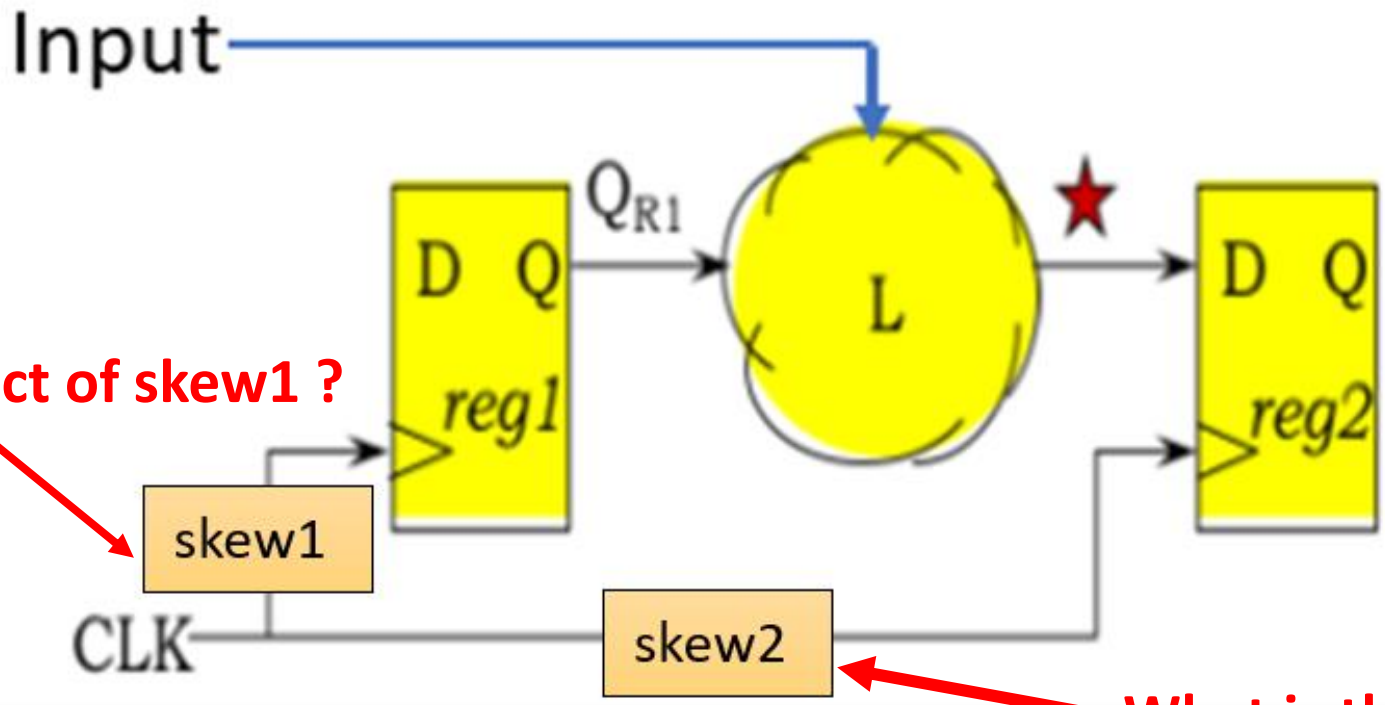




$$t_1 = t_{CD,reg1} + t_{CD,L} \geq t_{HOLD,reg2}$$

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

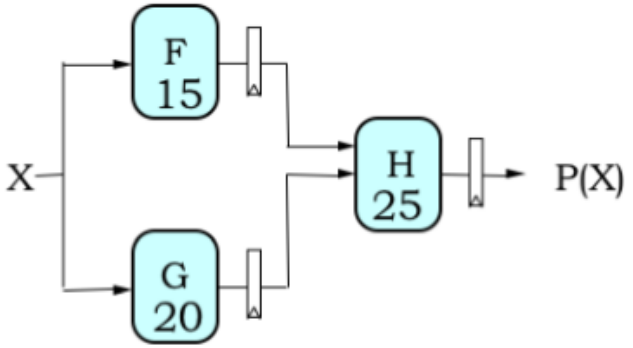
Clock Skew



What is the impact of skew1 ?

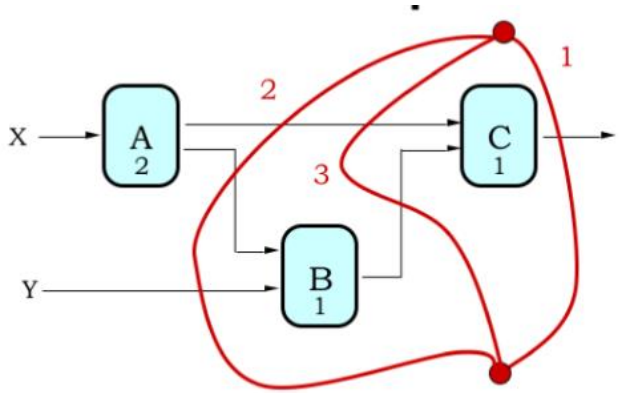
What is the impact of skew2 ?

Pipeline – Latency v.s. Throughput



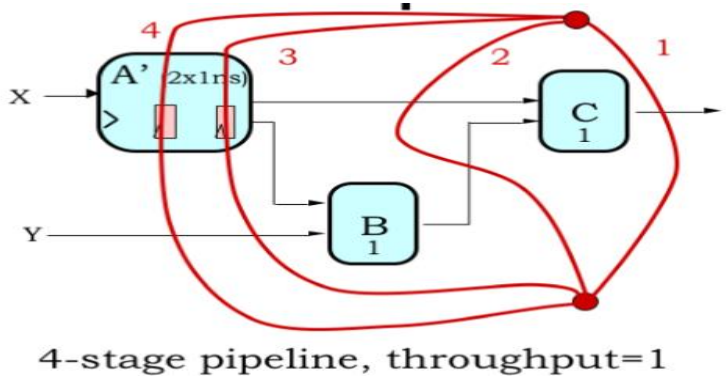
	latency	throughput
unpipelined	45	1/45
2-stage pipeline	50	1/25

Pipeline partition



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

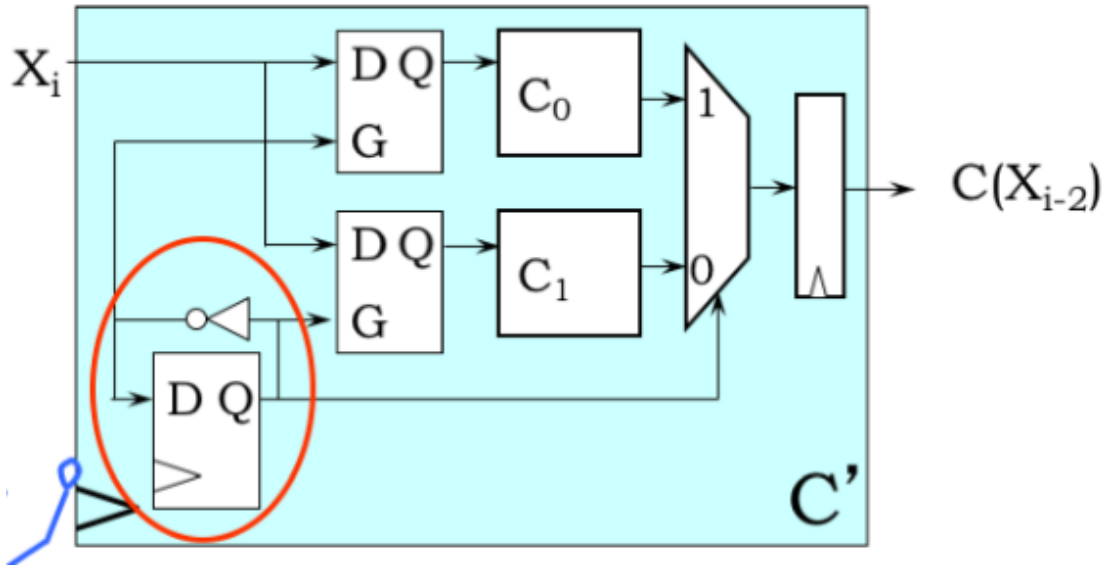
- Pipelined version of A

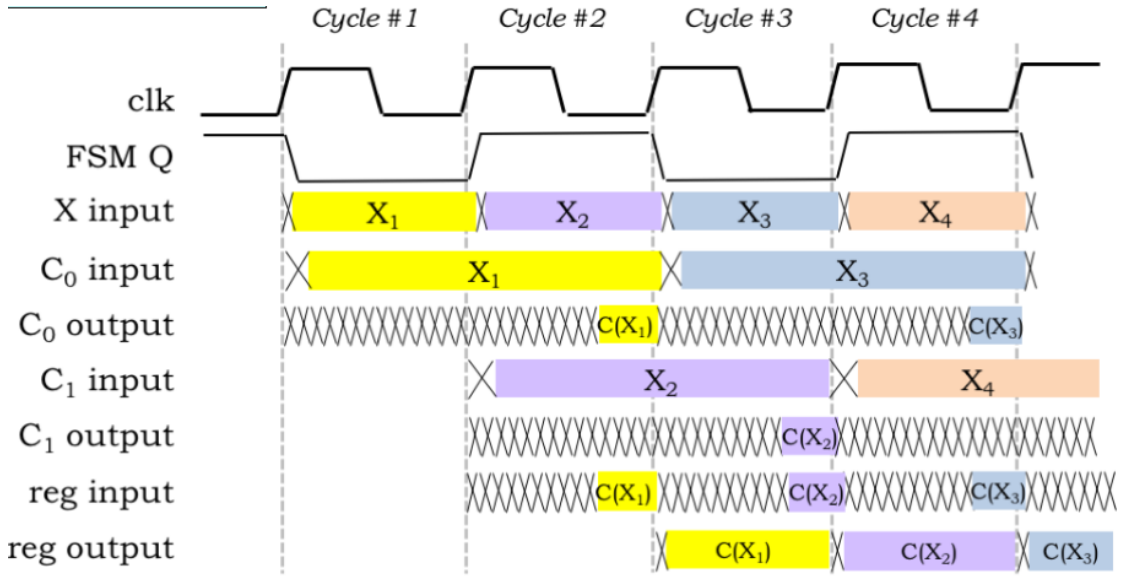


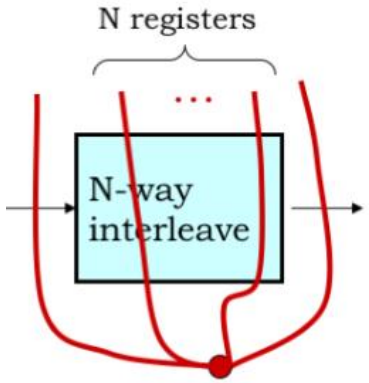
4-stage pipeline, throughput=1

4-stage pipeline
Latency：4， Throughput= 1
But how to pipeline A block ?

If no pipeline version -> Interleaving

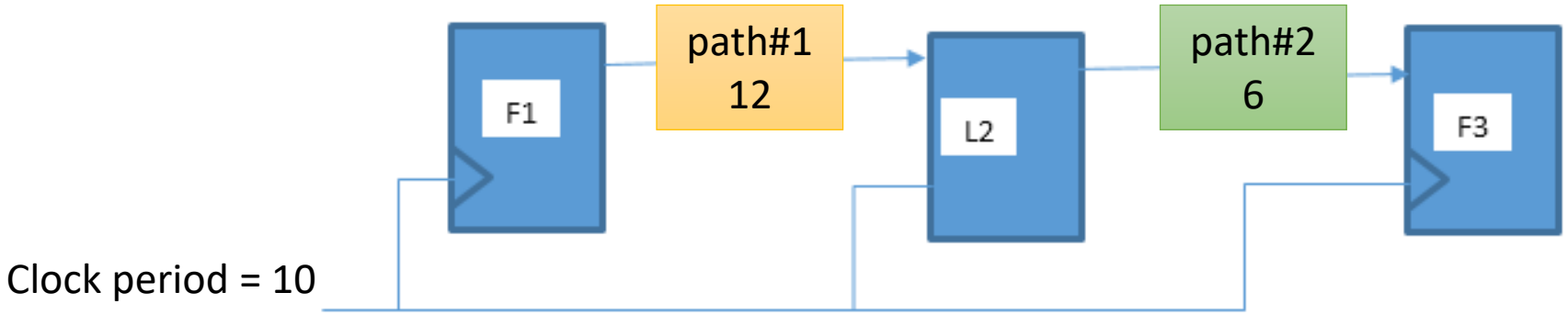






N-way interleaving = N pipeline Stages

Time Borrowing



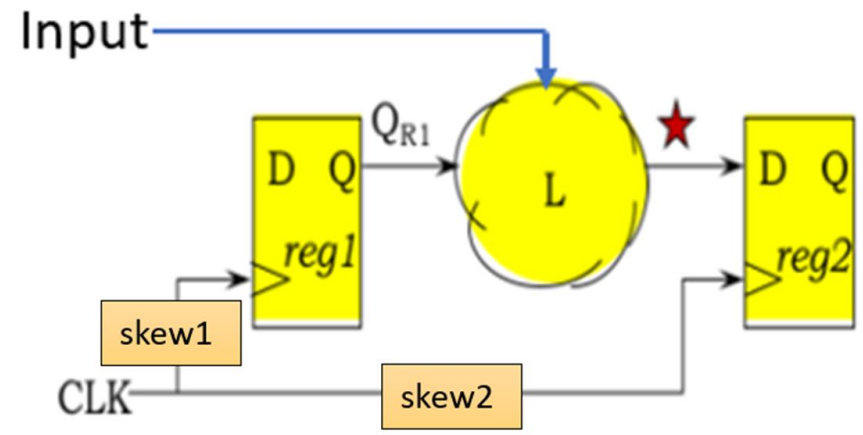
Recap – Sequential Logic

- 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

Question#1

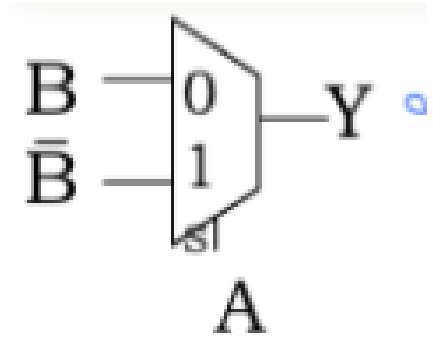
1. tPD,L could not meet the following equation, we can choose add delay at skew1 or skew2 ?

   tPD,reg1 + tPD,L + tSETUP,reg2 <= tCLK



Question#2

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



Question#3

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