



**PUF Academy**

A PUFsecurity Alliance

# Digital Logic Design

- Lecture 3
- Sequential Logic ■

2025 Spring

# Agenda ■

1. Sequential Logic
2. Static Timing Analysis
3. Generate Construct
4. Wavedrom tool

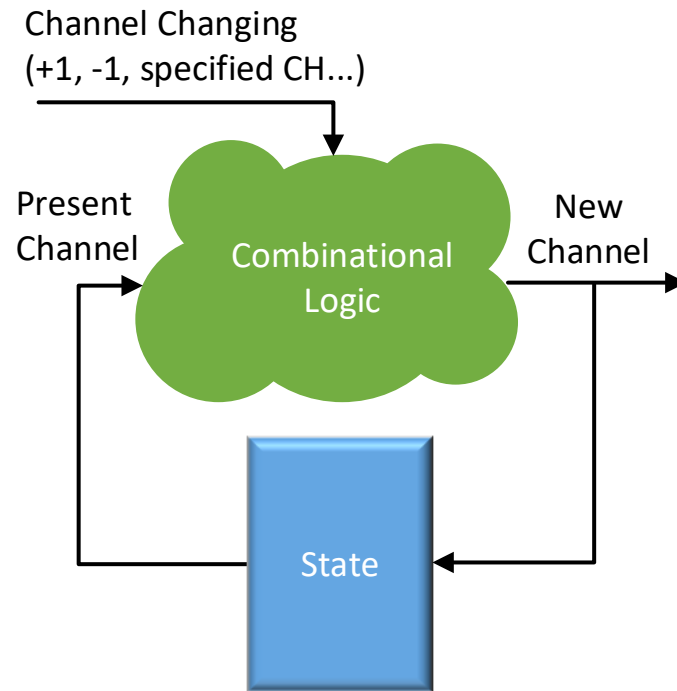
# Why We Need Sequential Logics? ■



- If the television is on channel 5, pressing “channel up” switches it to channel 6. Pressing “channel up” again switches it to channel 7. Why the same inputs get different outputs?
- Sequential logic has **state (memory)** while combi national logic does not
- The output of sequential logic depends on the **sequence of past inputs** and present input signals

# Sequential Logic ■

- We can imagine that a TV channel receiver would be like this:



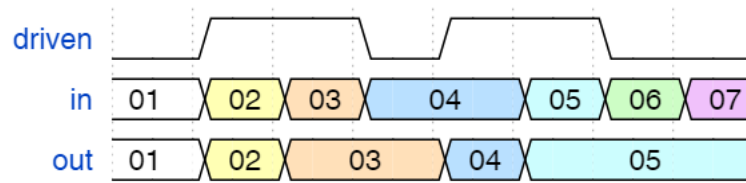
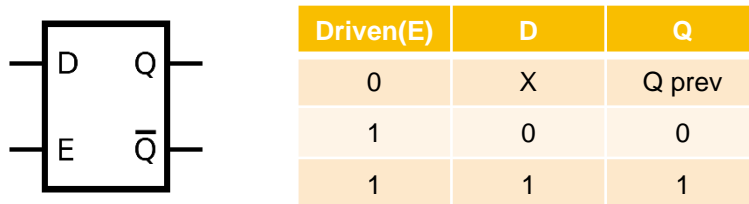
- IC circuit is a circuit composed of a group of standard components, what kind of components are the state in the circuit?



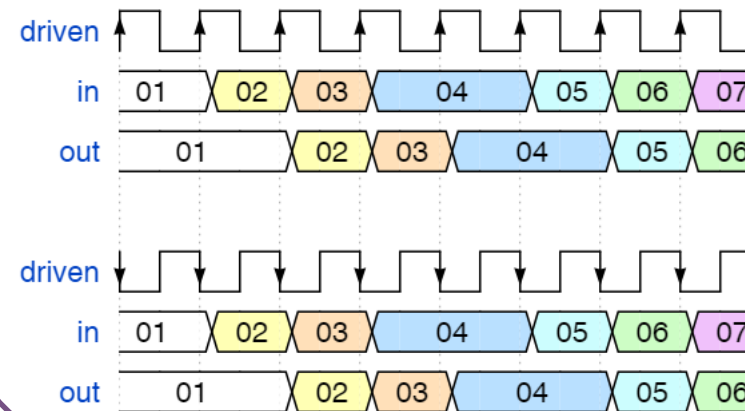
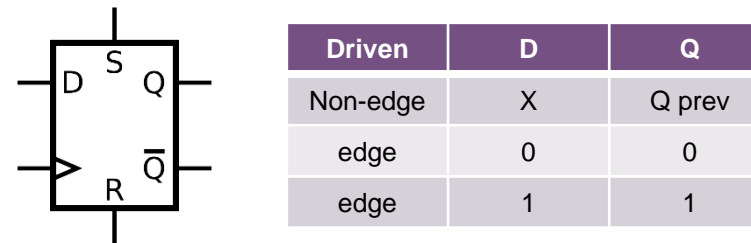
# Two Types Sequential Logics

- Latches and Flip-flops are the common circuits that have two stable states (0 or 1) that can store information

## D-Latch (level trigger)



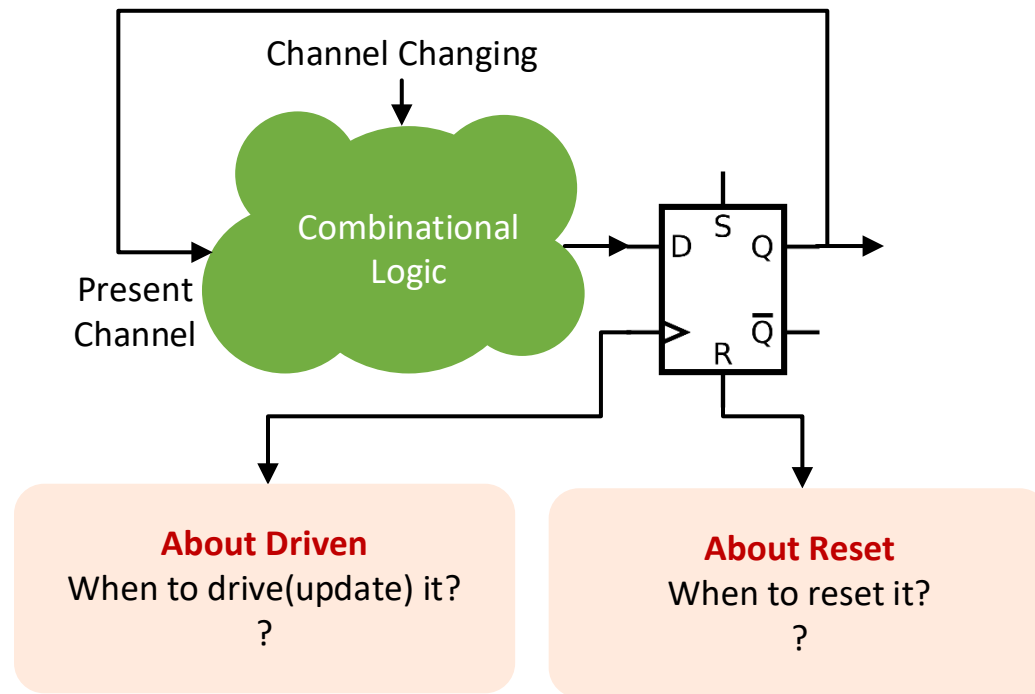
## D-Flip-Flop (edge trigger)





# Sequential Logic ■

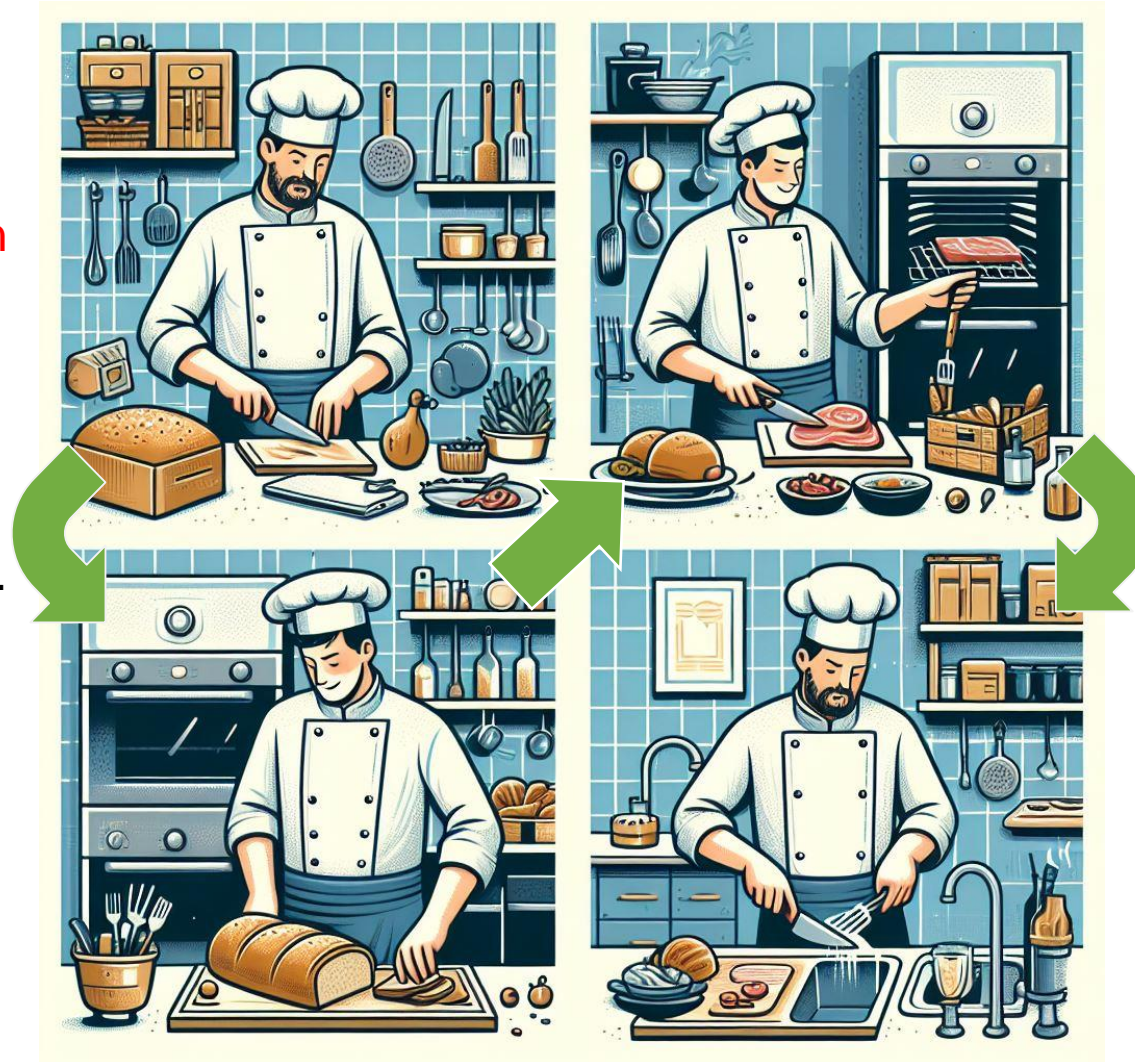
- For the convenience of Static Timing Analysis(STA), we almost only use D-Flip-Flop and not D-Latch, let's talk about it later
- Let's improve the circuit a little bit:



Drive it when TV receives a channel change signal, any problem?

# Four Chefs in The Kitchen ■

- If you have four chefs. Everyone completes their work and passes it on to the next person
- Paul cuts the meat. It takes 4~6 mins
- John makes the bread. It takes 5~7 mins

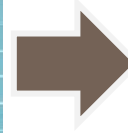


- George uses the oven. It takes 3~7 mins
- Ringo sets the dishes. It takes 3~5 mins
- How do you ensure that every dish is served quickly and without mistakes?



# Four Chefs in The Kitchen

- John waits for one dish to be finished before moving on to the next



One dish is served every 20 mins on average

- Everyone finishes their part and immediately moves on to the next one



Hurry!!



Hurry!!



Hurry!!



Hurry!!



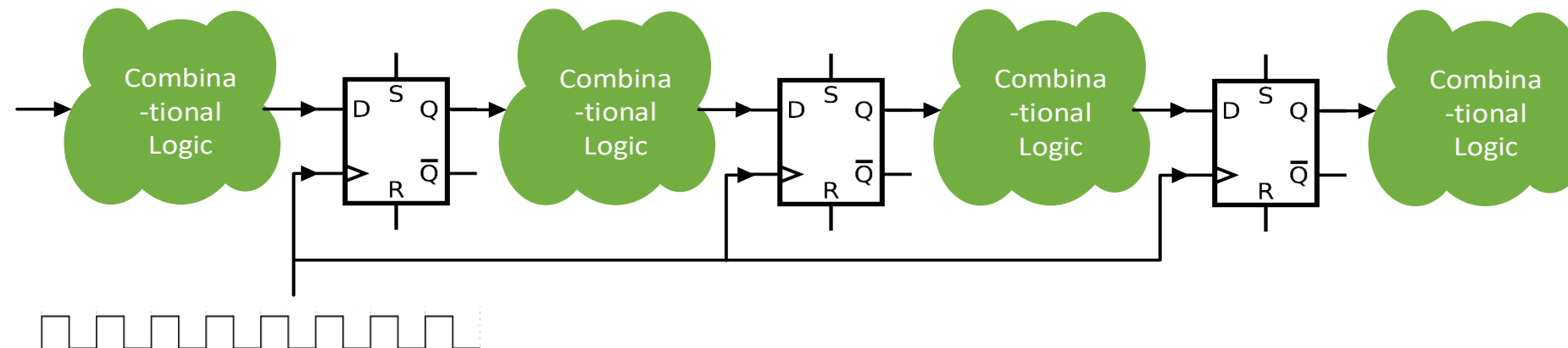
George may roast two pieces of meat at once

# Four Chefs in The Kitchen

- John waits for one dish to be finished before moving on to the next

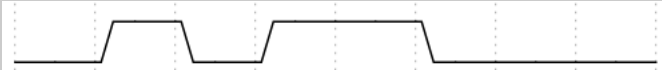





- The same is true in IC design



# Sequential Logic ■

- Driven are divided into **Asynchronous** and **Synchronous** types

Asynchronous	Synchronous
	
 <p data-bbox="810 639 1294 815">It is NOT synchronized by a clock signal (trigger when needed)</p>	 <p data-bbox="1837 639 2277 815">It is synchronized by a clock signal (periodically)</p>
<p data-bbox="471 996 1123 1086">The speed limited by <b>propagation delay of logic gates</b></p>	<p data-bbox="1610 996 2007 1086">The speed limited by <b>clock period</b></p>



# Sequential Logic ■

- **Synchronous circuit** is a better solution for pipeline construction
- Is it the perfect ? Not exactly
  - The maximum clock rate is determined by the slowest logic path



When a combinational circuit is complex,  
it will reduce the overall output efficiency

To split complex operations into several  
simple operations



- High frequency clock consumes a large amount of power and heat

Clock signal must be distributed to every flip-flop.

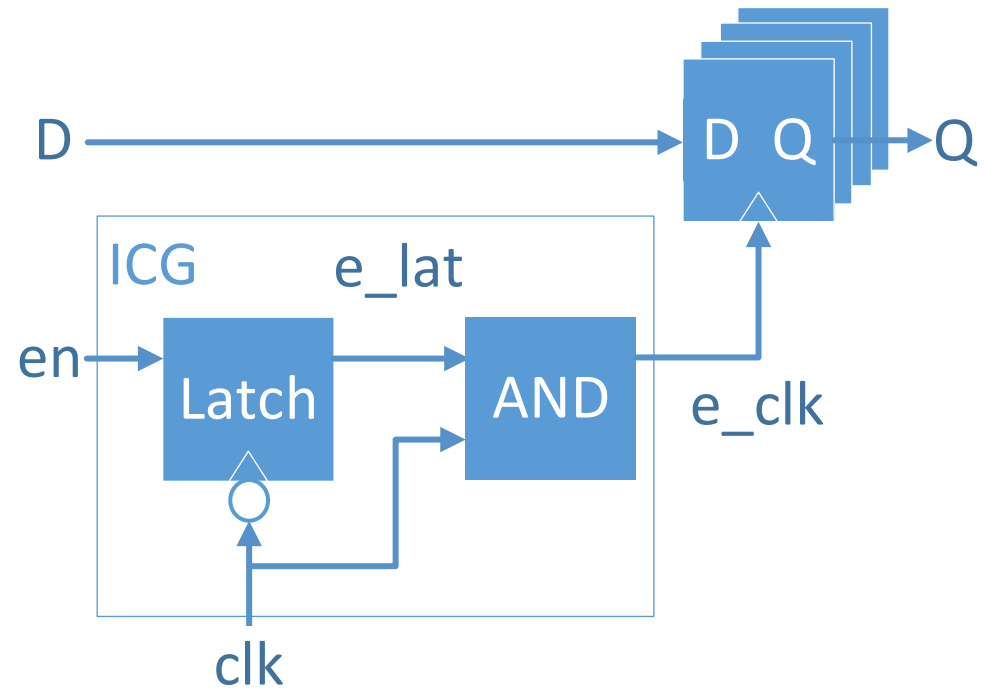
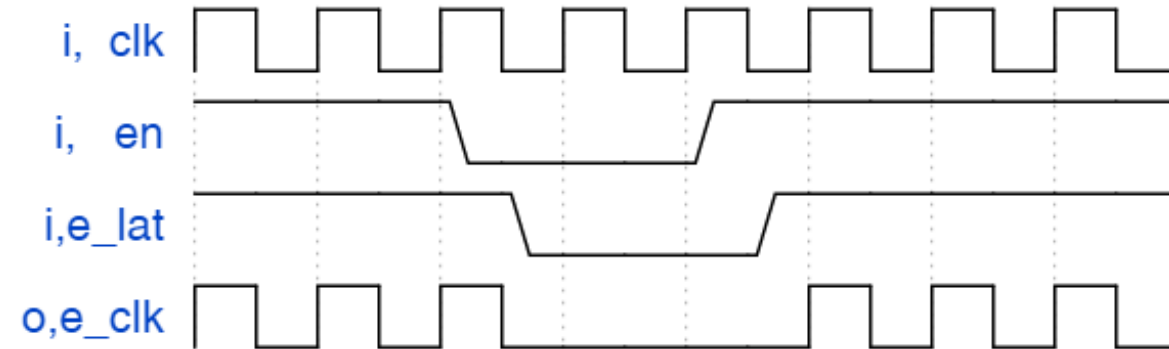
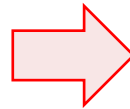
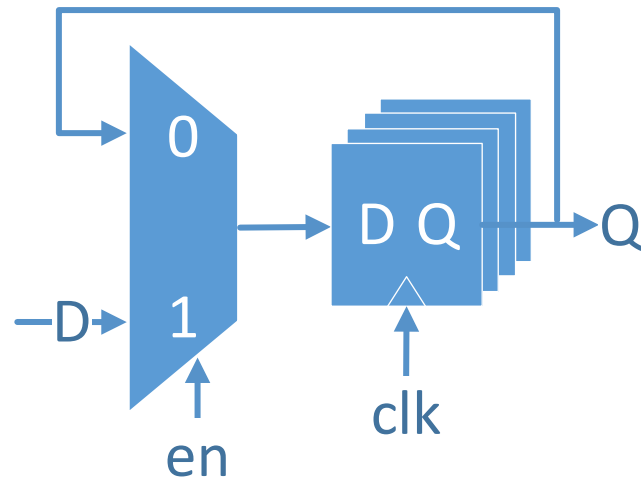
This distribution consumes a large power even the flip-flops  
that are doing nothing

To temporarily turn off the clock while the  
device is not being actively used



# Integrated Clock Gating

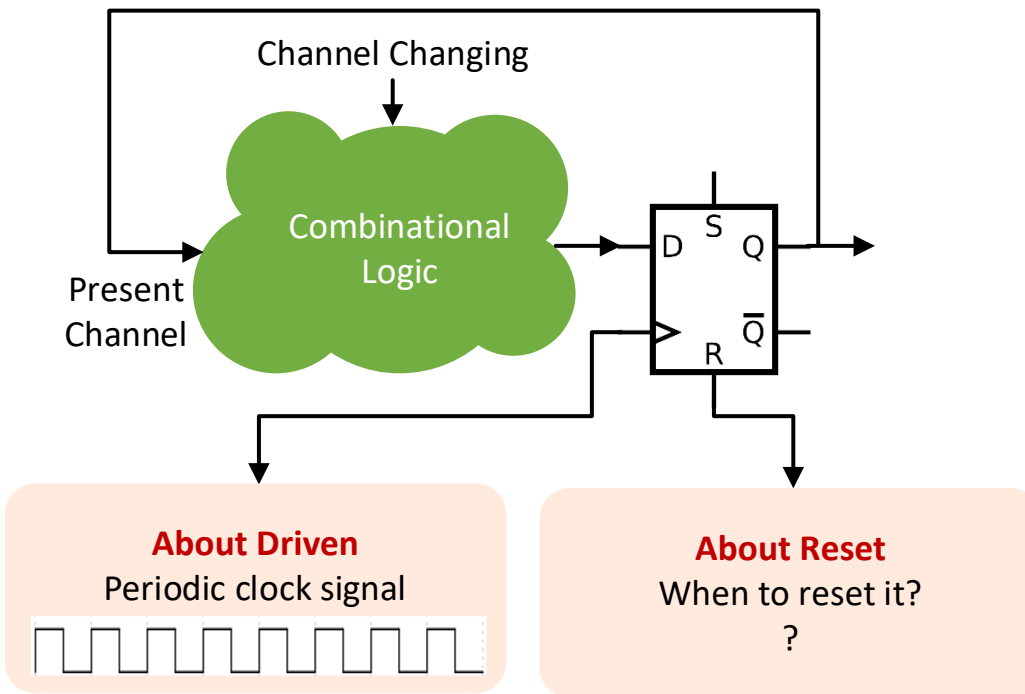
- ICG cell basically stops the clock propagation through it when we apply a low clock enable signal on it
- Reduce clock switching
- Reduce dynamic power





# Sequential Logic ■

- We recommend choose periodic clock signal as the driven signal
- Let's solve the reset signal:

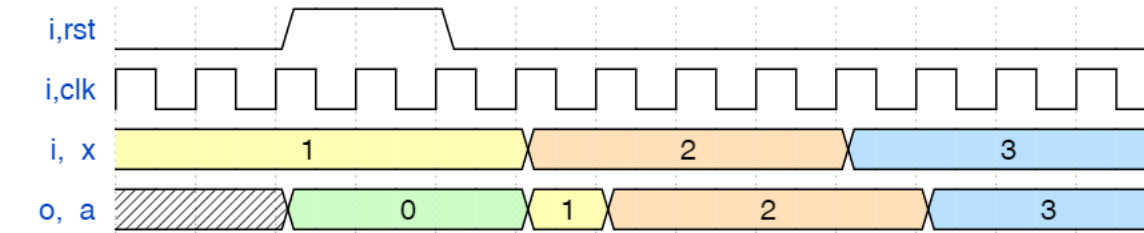


When the power is just turned on,  
which channel does the TV stop at?

# Active High and Active Low

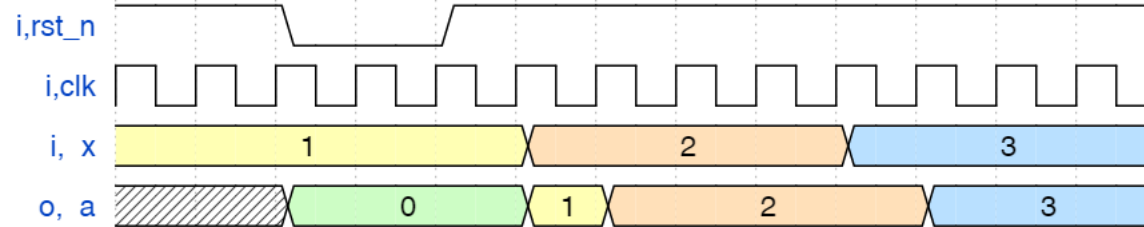
## ■ Active high reset

- Input goes high (more positive) to reset the circuit



## ■ Active low reset

- Input goes low (more negative) to reset the circuit



## ■ We prefer to use **Active Low Reset**

- The point of high voltage isn't defined and may vary from device
- Active low reset is already active the instant you switch power on

# Synchronous Reset

## ■ Synchronous Reset

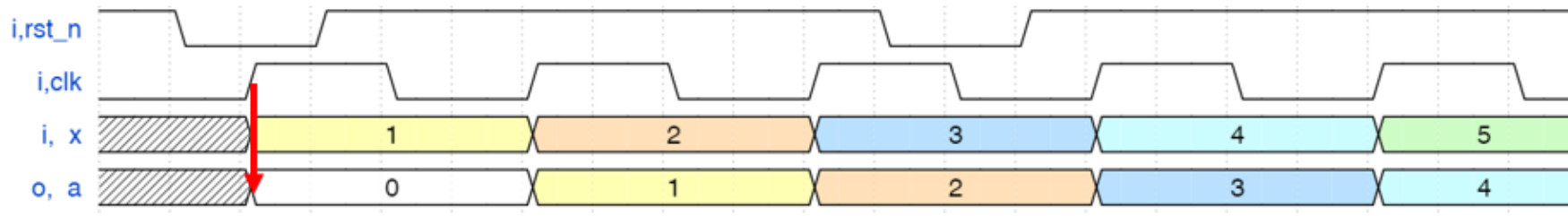
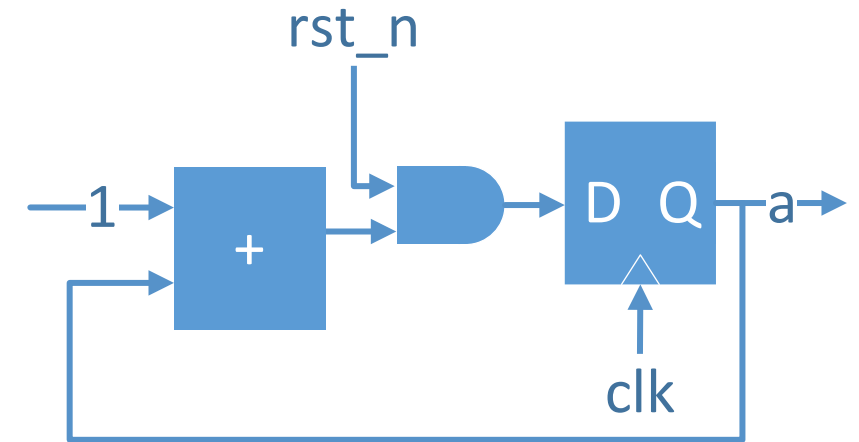
- Reset signal will only affect on the active edge of a clock

## ■ Advantages

- Glitch filtering from reset combinational logic

## ■ Disadvantages

- Can't be reset without clock signal
- May not be able come out unknow X during simulation
- Adds delay to data path



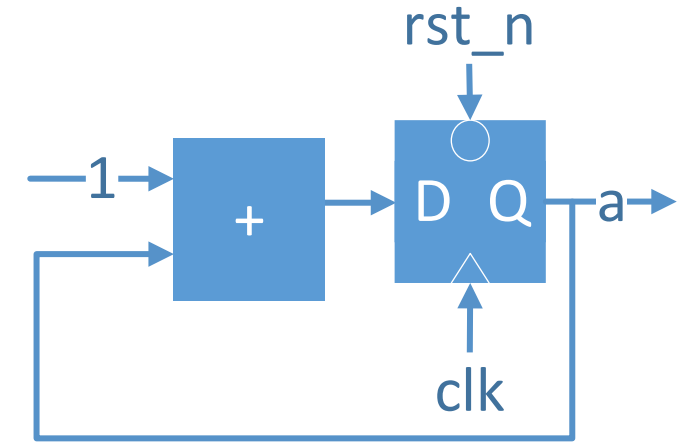
# Asynchronous Reset

## ■ Asynchronous Reset

- The flip-flop incorporate a reset pin into itself

## ■ Advantages

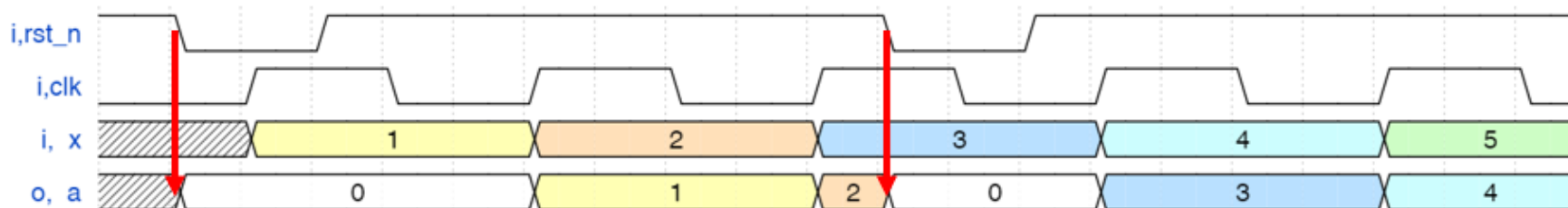
- Reset is independent of clock signal
- No problem related to unknown X propagation in simulation
- Does not interfere or add extra delay to data path
- Reset is immediate



## ■ Disadvantages

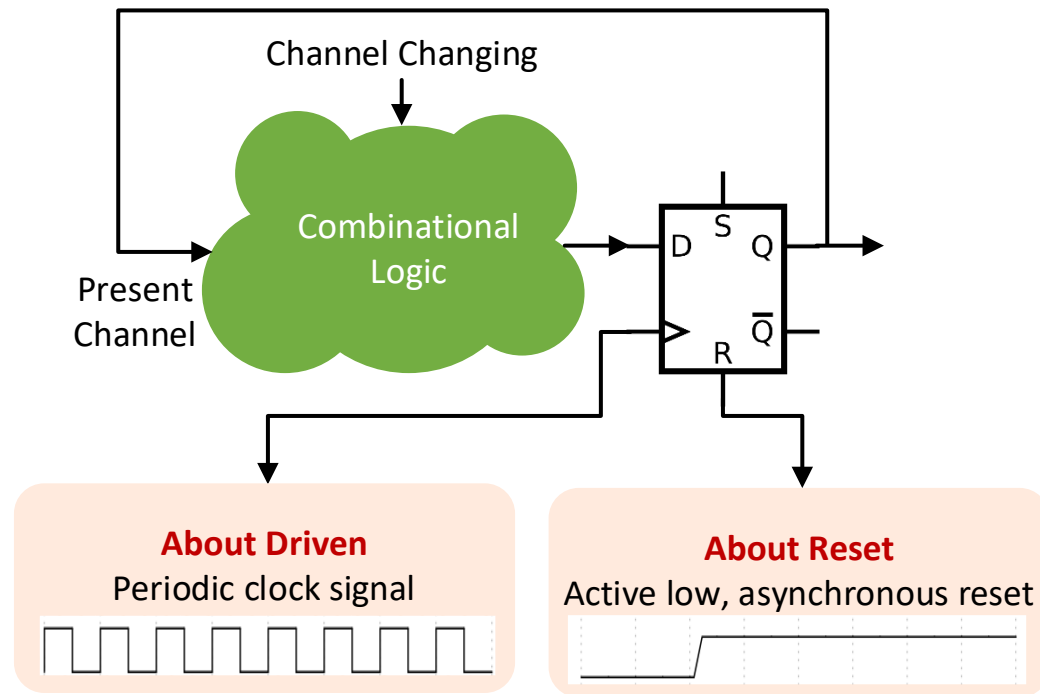
- Noisy reset line could cause unwanted reset => can be solved by advanced process
- Reset signal cannot be too close to the clock edge => be solved by advanced process

## ■ We like to use Asynchronous Reset more



# Sequential Logic ■

- We recommend choose asynchronous signal as reset signal
- Let's improve the circuit a little bit:



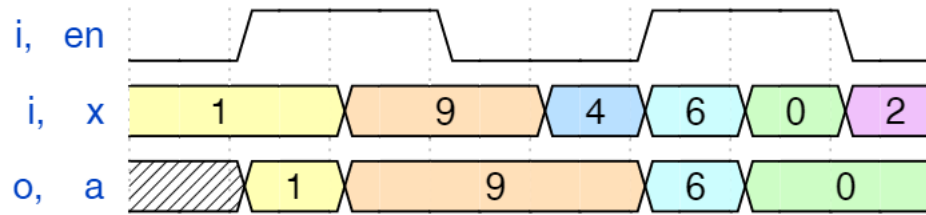
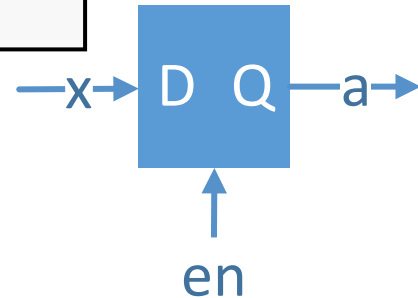
We have an idea of what the circuit looks like in our minds, but how to we explain it to the computers?



# Basic Sequential Logic(1/4) ■

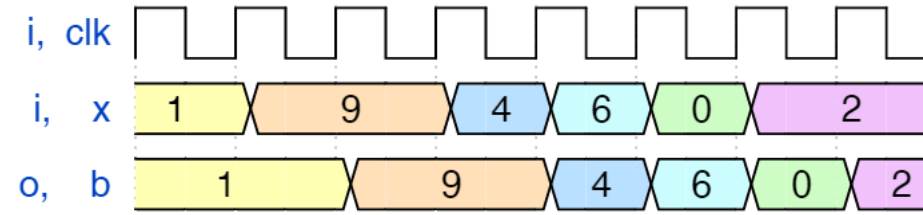
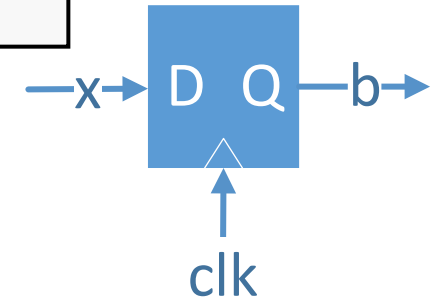
## ■ Latch

```
7 //latch
8 wire [3:0] x;
9 reg [3:0] a;
10 always@* begin
11     if(en==1'b1)    a = x ;
12 //else            a = a ; //keep value
13 end
```



## ■ Flip-flop

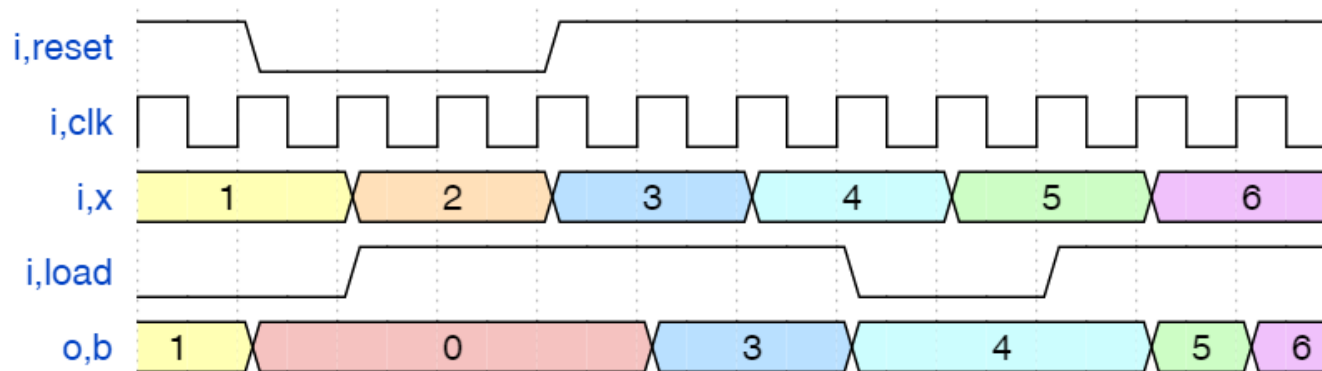
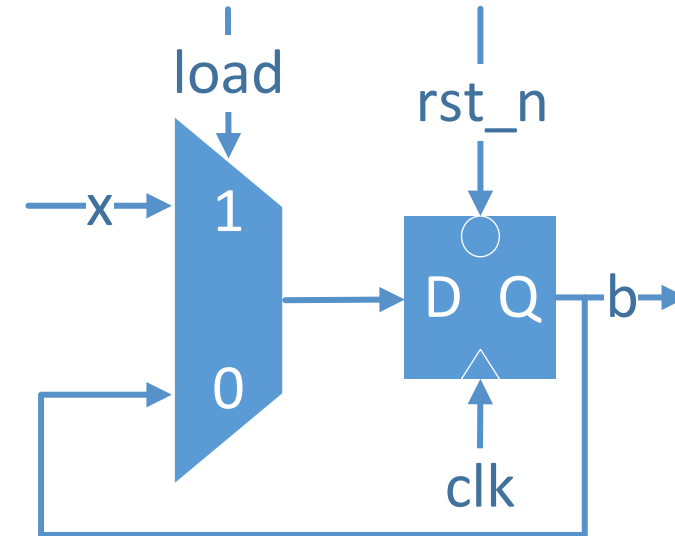
```
15 //flip-flop
16 wire [3:0] x;
17 reg [3:0] b;
18 always@(posedge clk) begin //edge trigger
19     b <= x ;
20 end
```



# Basic Sequential Logic(2/4)

## Flip-Flop with Reset (edge trigger)

```
22 //flip-flop with reset
23 wire [3:0] x;
24 wire      load;
25 reg  [3:0] b;
26 always@(posedge clk or negedge rst_n) begin //clock or reset only
27     if(~rst_n)      b <= 4'b0 ; //put reset in the 1st condition
28     else if(load)    b <= x ;
29     /*else          b <= b ;*/ //keep value if no define
30 end
```

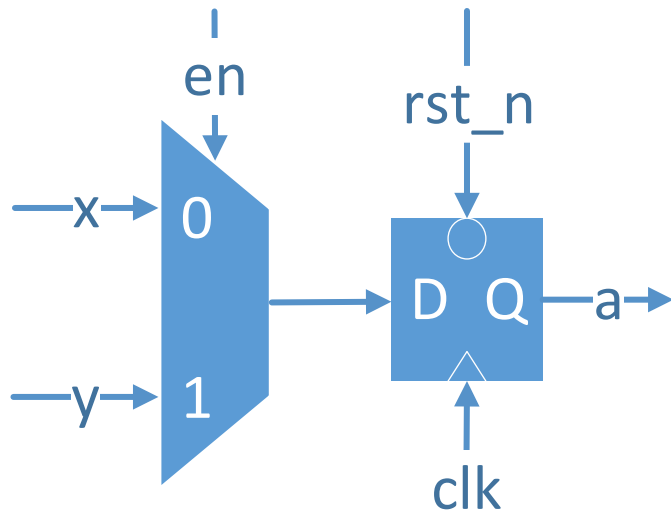


# Basic Sequential Logic(3/4) ■

- Put a flip-flop in one always block



```
16 //good
17 always@(posedge clk or negedge rst_n) begin
18     if(~rst_n) a <= 1'b0;
19     else if(en) a <= y;
20     else       a <= x;
21 end
```



- Put a flip-flop in separate always block



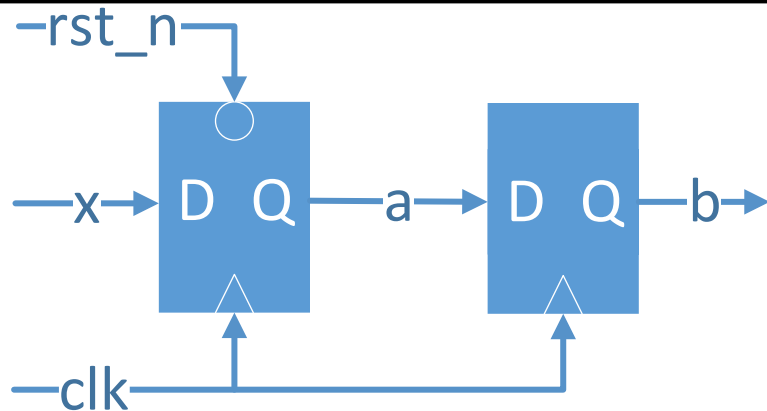
```
8 //bad
9 always@(posedge clk or negedge rst_n) begin
10     if(~rst_n) a <= 1'b0;
11     else       a <= x;
12 end
13 always@(posedge clk) begin
14     if(en)     a <= y;
15 end
```

# Basic Sequential Logic(4/4) ■

- Put dissimilar flip-flops in separate always blocks
  - Except: shift regs, similar properties regs



```
19 //good
20 always@(posedge clk or negedge rst_n) begin
21     if(~rst_n) begin
22         a <= 1'b0;
23     end
24     else begin
25         a <= x;
26     end
27 end
28
29 always@(posedge clk) begin
30     b <= a;
31 end
```



- Put dissimilar flip-flops in the same always block

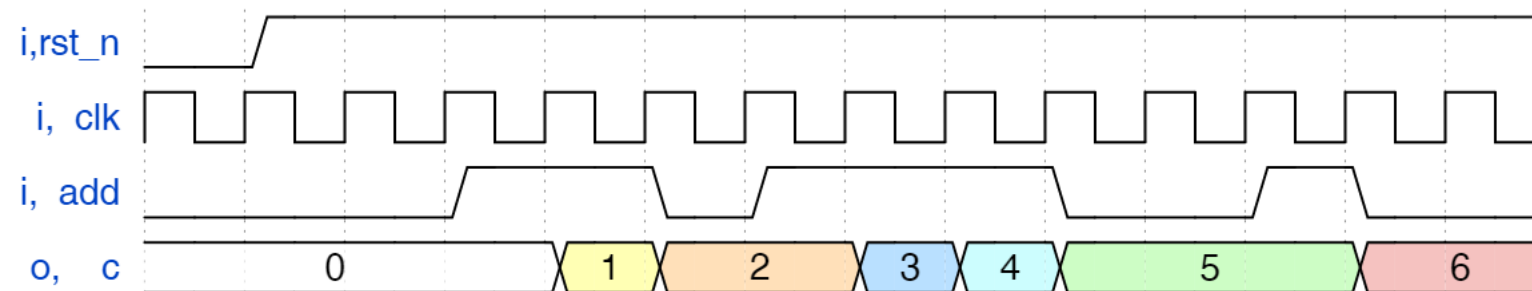
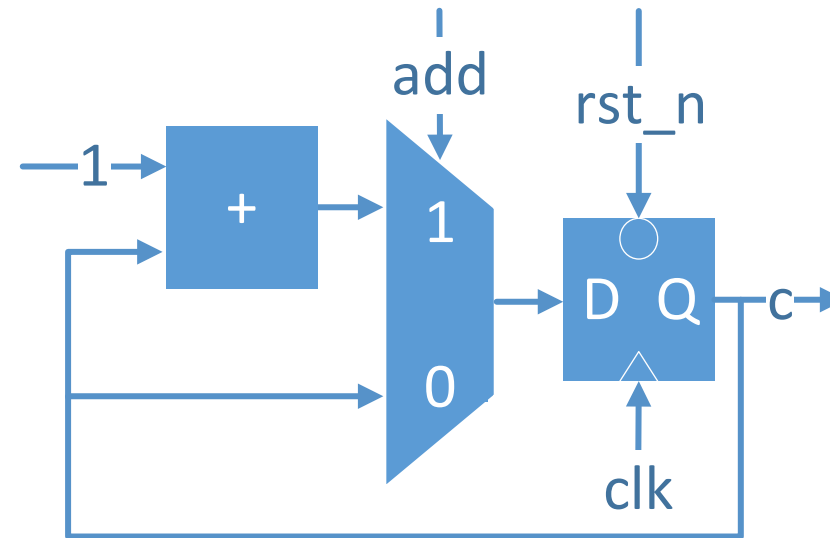


```
8 //bad
9 always@(posedge clk or negedge rst_n) begin
10     if(~rst_n) begin
11         a <= 1'b0;
12     end
13     else begin
14         a <= x;
15         b <= a;
16     end
17 end
```

# Counter

## 4-bit counter

```
32 //counter
33 wire    add;
34 reg [3:0] c;
35 always@(posedge clk or negedge rst_n) begin
36     if(~rst_n)      c <= 4'b0 ;
37     else if(add)    c <= c + 4'b1 ;
38     /*else          c <= c;*/
39 end
```

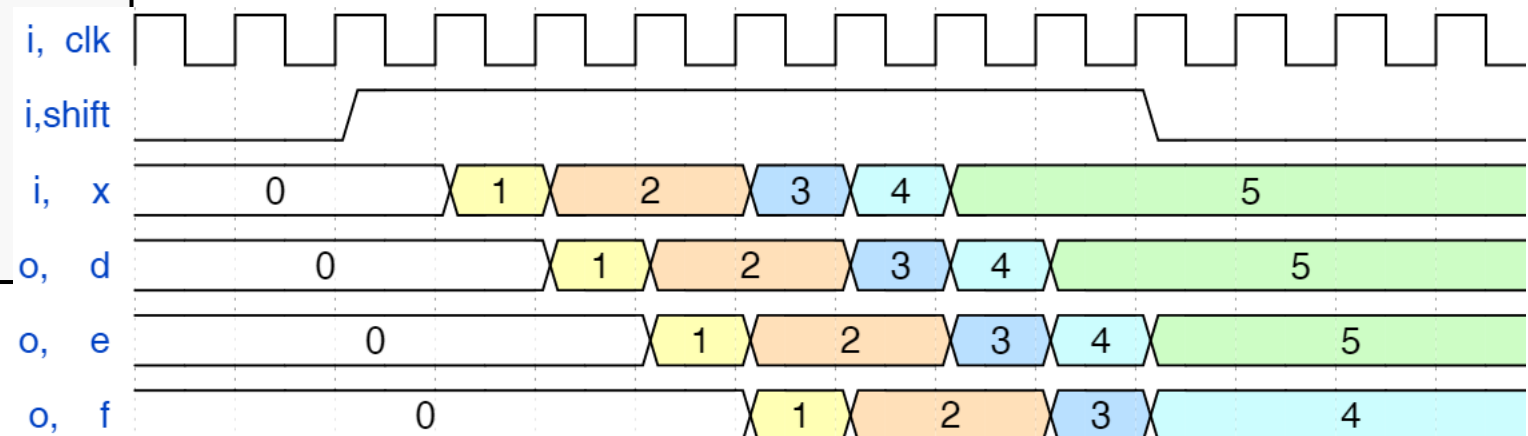
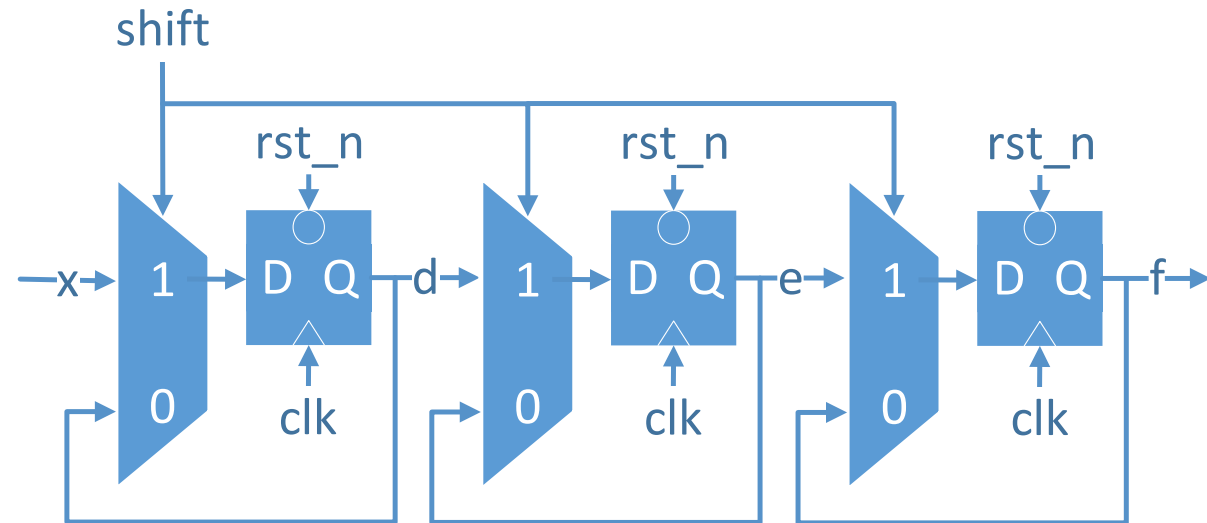




# Shift Registers

## Shift registers

```
41 //shift registers
42 wire [3:0] x;
43 wire      shift;
44 reg  [3:0] d, e, f;
45 always@(posedge clk or negedge rst_n) begin
46     if(~rst_n) begin
47         d <= 4'b0;
48         e <= 4'b0;
49         f <= 4'b0;
50     end
51     else if(shift) begin
52         d <= x;
53         e <= d;
54         f <= e;
55     end
56     /*else begin
57         d <= x;
58         e <= d;
59         f <= e;
60     end*/
61 end
```

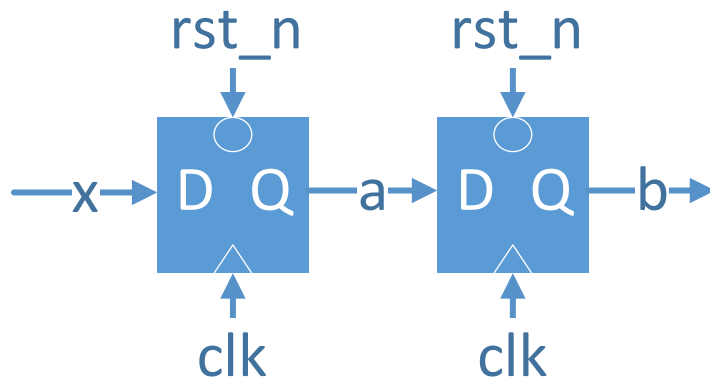


# Blocking and Non-Blocking

- Non-blocking assignment “<=”



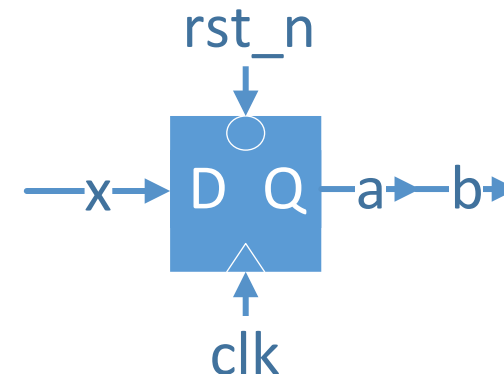
```
21 //non-blocking
22 wire [3:0] x;
23 reg [3:0] a, b;
24 always@(posedge clk or negedge rst_n) begin
25     if(~rst_n) begin
26         a <= 4'b0;
27         b <= 4'b0;
28     end
29     else begin
30         a <= x;
31         b <= a;
32     end
33 end
```



- Blocking assignment “=”

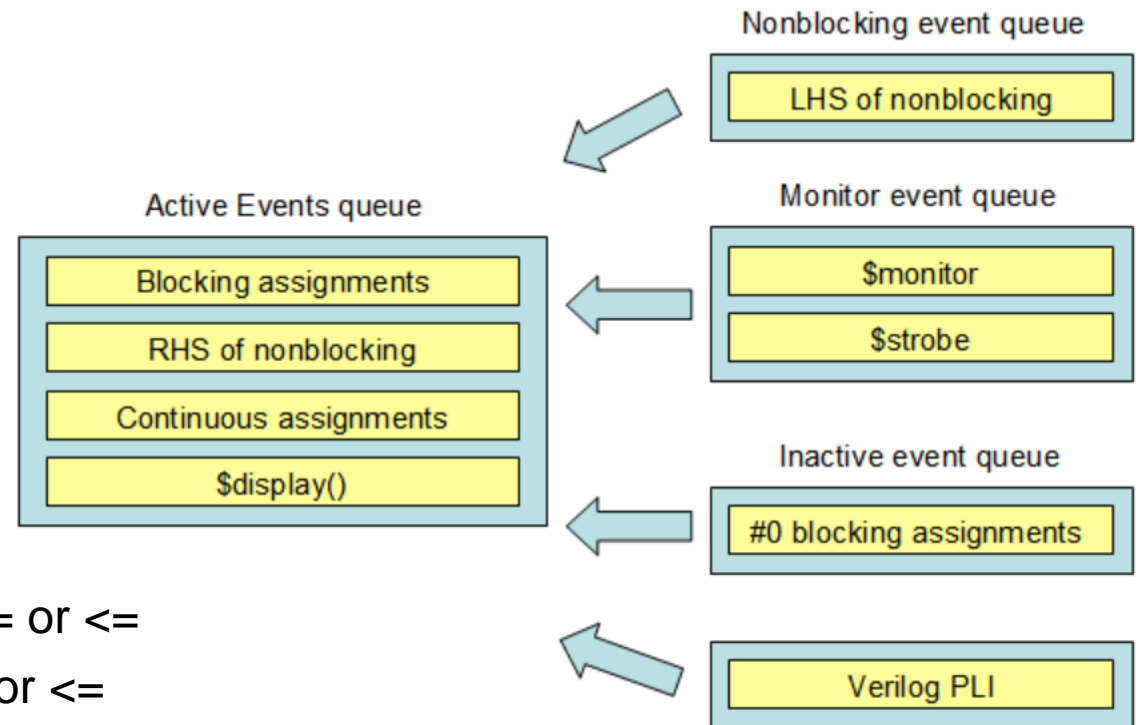


```
8 //blocking, DO NOT USE IT
9 wire [3:0] x;
10 reg [3:0] a, b;
11 always@(posedge clk or negedge rst_n) begin
12     if(~rst_n) begin
13         a = 4'b0;
14         b = 4'b0;
15     end
16     else begin
17         a = x;
18         b = a;
19     end
20 end
```



# Blocking and Non-Blocking

- **Blocking:** The next line cannot be executed until one line has been executed
- **Non-blocking:** Each line is executed simultaneously in one block
- Hardware circuit can be executed in parallel, but the computer software itself is executed sequentially, so we need a non-blocking description to solve it
- **RHS:** An expression or variable to the right of = or <=
- **LHS:** An expression or variable to the left of = or <=
- Whole non-blocking behavior
  - Perform RHS at the beginning of the clk rising edge
  - Perform LHS near the end of the clk rising edge






# Clock ■

- Coding style about clocks
  - Multiple clocks in the same blocks
  - Different edge trigger in the same design
  - Multiple clocks in different blocks



```
11 //multiple clocks(resets) in the same blocks
12 always@(posedge clk1 or posedge clk2 or negedge rst1_n) begin
13     if(~rst1_n)      a <= 4'b0;
14     else              a <= x;
15 end
16
17 //different edge trigger in the same design
18 always@(posedge clk1 or negedge rst1_n) begin
19     if(~rst1_n)      a <= 4'b0;
20     else              a <= x;
21 end
22 always@(negedge clk1 or negedge rst1_n) begin
23     if(~rst1_n)      b <= 4'b0;
24     else              b <= y;
25 end
26
27 //multiple clocks in different blocks
28 always@(posedge clk1 or negedge rst1_n) begin
29     if(~rst1_n)      a <= 4'b0;
30     else              a <= x;
31 end
32 always@(posedge clk2 or negedge rst2_n) begin
33     if(~rst2_n)      b <= 4'b0;
34     else              b <= y;
35 end
```

# Reset(1/4) ■

- Coding style about Resets
  - Reset value can be 0 or 1 
  - Active-low <-> negative edge  
Active-high <-> positive edge 
  - Reset value must be constant 

```
48 //reset value could be 1
49 always@(posedge clk or negedge rst_n) begin
50     if(~rst_n)        f <= 4'b1111;
51     else if(clear)     f <= 4'b0000;
52     else if(add)       f <= f + 4'b1;
53 end

32 //active low <-> negative ; active high <-> posedge
33 always@(posedge clk or posedge rst_n) begin
34     if(~rst_n)        d <= 4'b0000;
35     else if(clear)     d <= 4'b0000;
36     else if(add)       d <= d + 4'b1;
37 end
38
39
40 //reset value must be constant
41 always@(posedge clk or negedge rst_n ) begin
42     if(~rst_n)        e <= x;
43     else if(clear)     e <= 4'b0000;
44     else               e <= e + 4'b1;
45 end
46
```



# Reset(2/4) ■

- Coding style about Resets
  - 2(or more) edge-trigger resets



- Edge-trigger reset combines other signals



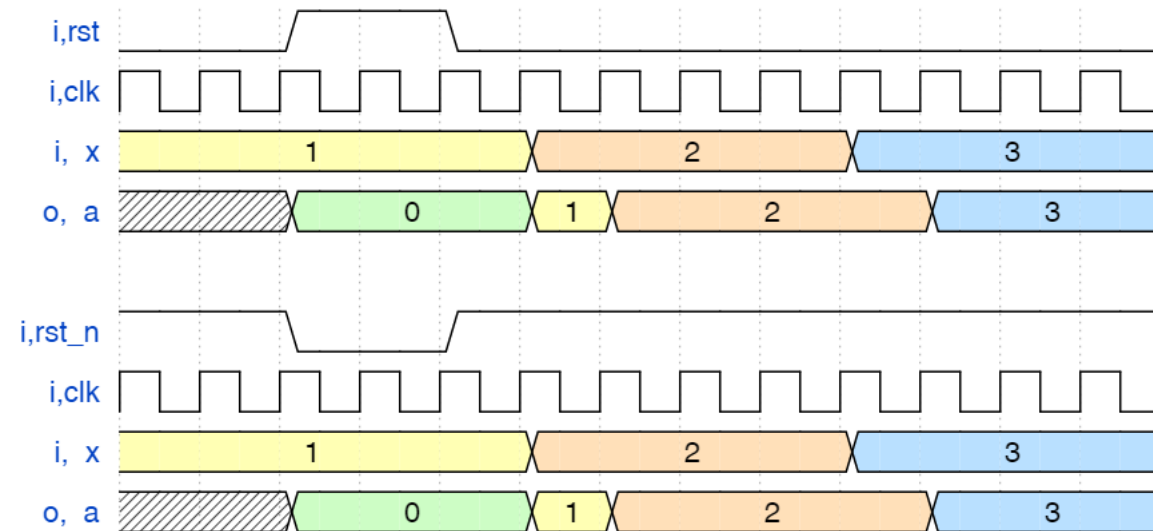
- Put edge-trigger reset at the first condition



```
9 //2(or more) edge-trigger resets
10 always@(posedge clk or negedge rst_n or negedge clear) begin
11     if(~rst_n)          a <= 4'b0000;
12     else if(clear)      a <= 4'b0000;
13     else                 a <= a + 4'b1;
14 end
15
16
17 //edge-trigger reset combines other signal
18 always@(posedge clk or negedge rst_n) begin
19     if(~rst_n | clear) b <= 4'b0000;
20     else if(add)      b <= b + 4'b1;
21 end
22
23
24 //put edge-trigger reset at the first condition
25 always@(posedge clk or negedge rst_n) begin
26     if(clear)          c <= 4'b0000;
27     else if(~rst_n)    c <= 4'b0000;
28     else if(add)      c <= c + 4'b1;
29 end
```

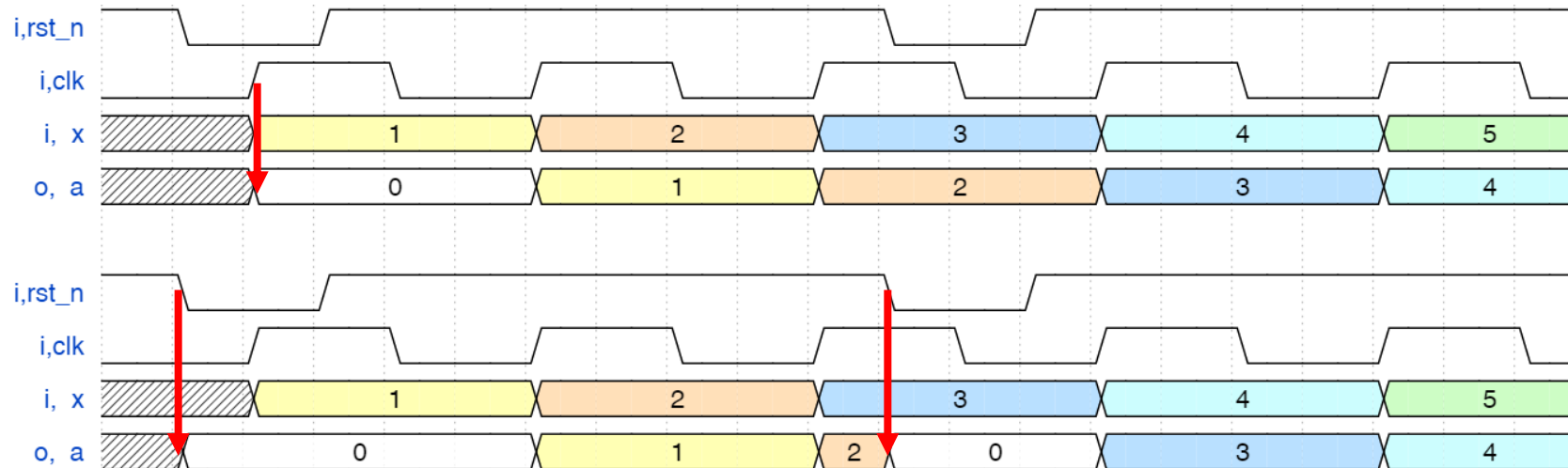
# Reset(3/4) ■

```
11 wire [3:0] x;  
12 reg  [3:0] a;  
13 reg  [3:0] b;  
14 //active high  
15 always@(posedge clk or posedge rst) begin  
16     if(rst) a <= 4'b0;  
17     else   a <= x;  
18 end  
19 endmodule  
20  
21  
22 //active low  
23 always@(posedge clk or negedge rst_n) begin  
24     if(~rst_n) b <= 4'b0;  
25     else      b <= x;  
26 end
```



# Reset(4/4) ■

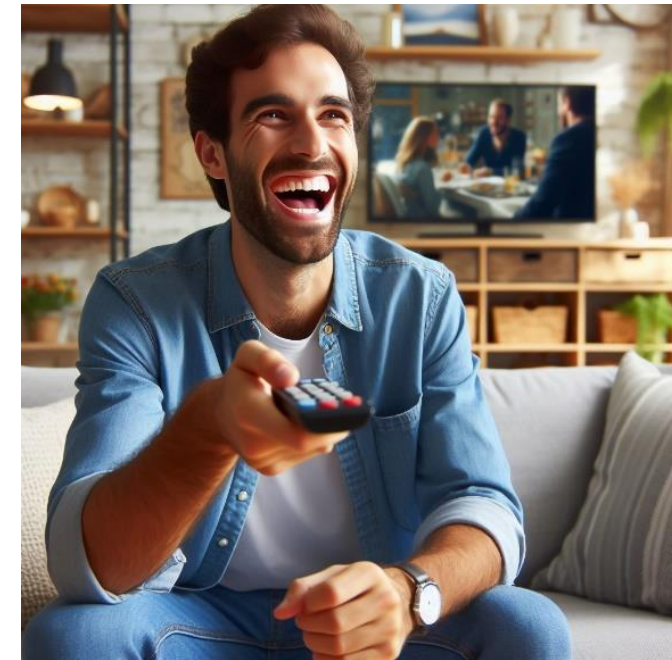
```
9 //synchronous
10 //rst_n is not in the sensitivity list
11 always@(posedge clk) begin
12     if(~rst_n) a <= 4'b0;
13     else      a <= x;
14 end
15
16 //asynchronous
17 //rst_n is in the sensitivity list
18 always@(posedge clk or negedge rst_n) begin
19     if(~rst_n) a <= 4'b0;
20     else      a <= x;
21 end
```



# Quick Tips ■

- **Draw block diagrams before you write codes !**
- Use `always@(...)` block in sequential logic, and put clock and reset only
- All registers can be reset (either synchronous or asynchronous)
- Choose `positive` or `negative` edge trigger of clock and reset
- Put reset in 1<sup>st</sup> condition and keep it clean
- Use non-blocking “`<=`” in sequential logic
- Don't put one register in different always block
- Registers name are distinguishable as much as possible
- Avoid using case-only different names (ex: Data & data)

```
22 //flip-flop with reset
23 wire [3:0] x;
24 wire      load;
25 reg  [3:0] b;
26 always@(posedge clk or negedge rst_n) begin //clock or reset only
27     if(~rst_n)      b <= 4'b0 ; //put reset in the 1st condition
28     else if(load)   b <= x ;
29     /*else          b <= b ;*/ //keep value if no define
30 end
```

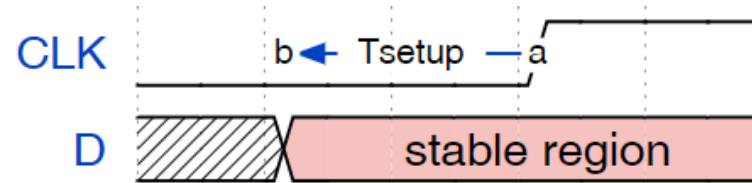


# Agenda ■

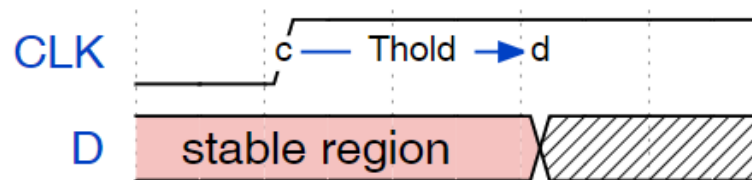
1. Sequential Logic
2. Static Timing Analysis
3. Generate Construct
4. Wavedrom tool

# Timing of D Flip-Flop

- Setup time ( $T_{\text{setup}}$ )
  - The setup time indicates a data signal remains stable for a minimum specified time **before** a transition in an enabling (clock event)



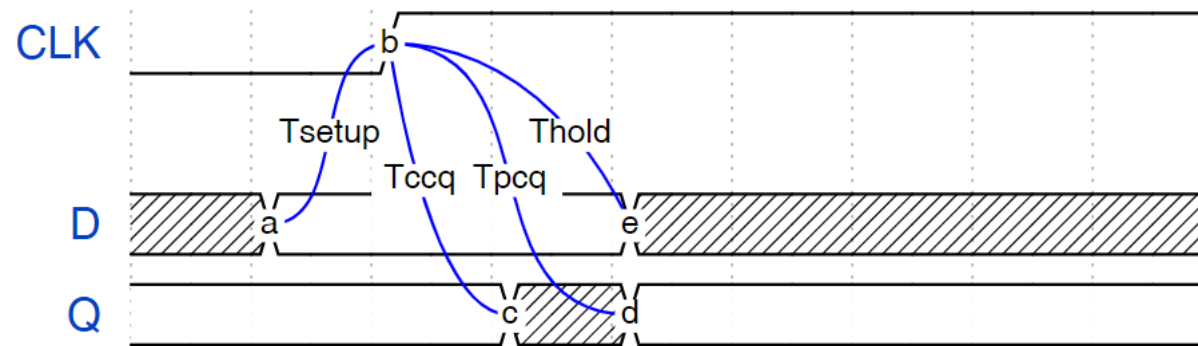
- Hold time ( $T_{\text{hold}}$ )
  - The hold time indicates a data signal remains stable for a minimum specified time **after** a transition in an enabling (clock event)



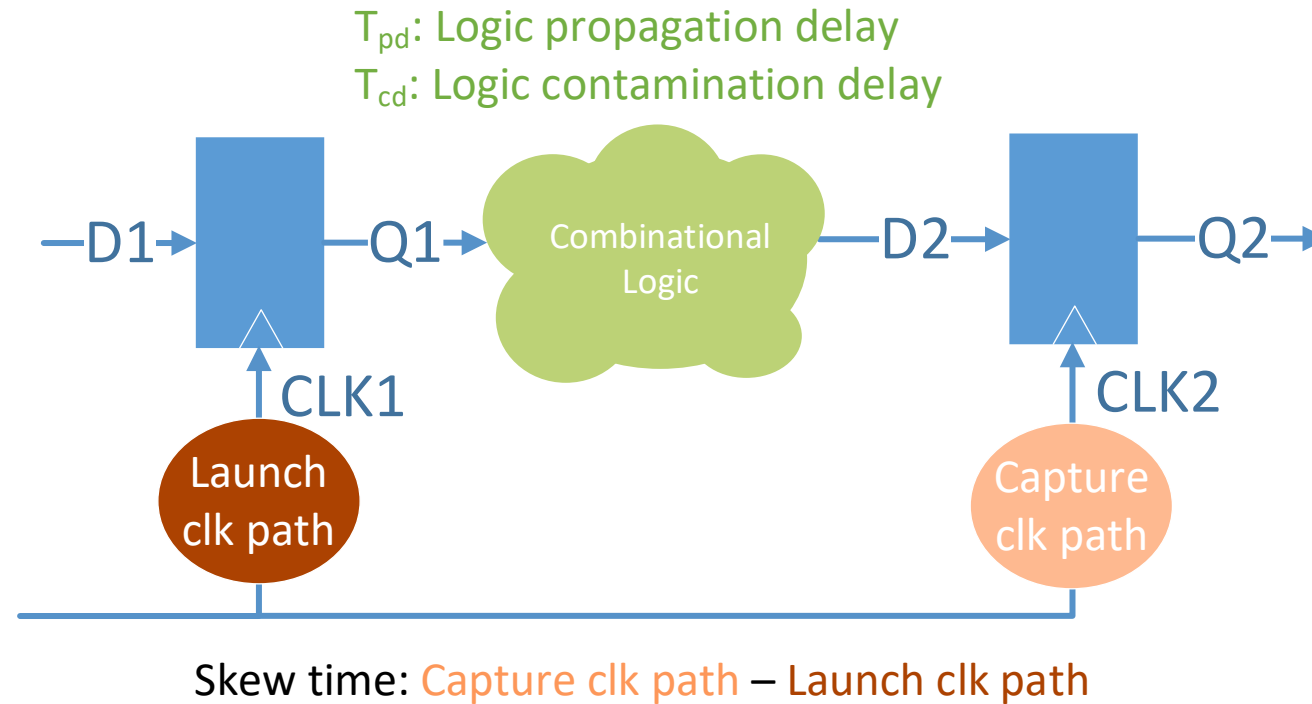


# Timing of D Flip-Flop ■

- CLK-to-Q contamination delay ( $T_{ccq}$ )
  - The contamination time that Q is first changed after the clock edge
- CLK-to-Q propagation delay ( $T_{pcq}$ )
  - The propagation time that Q reaches steady after the clock edge



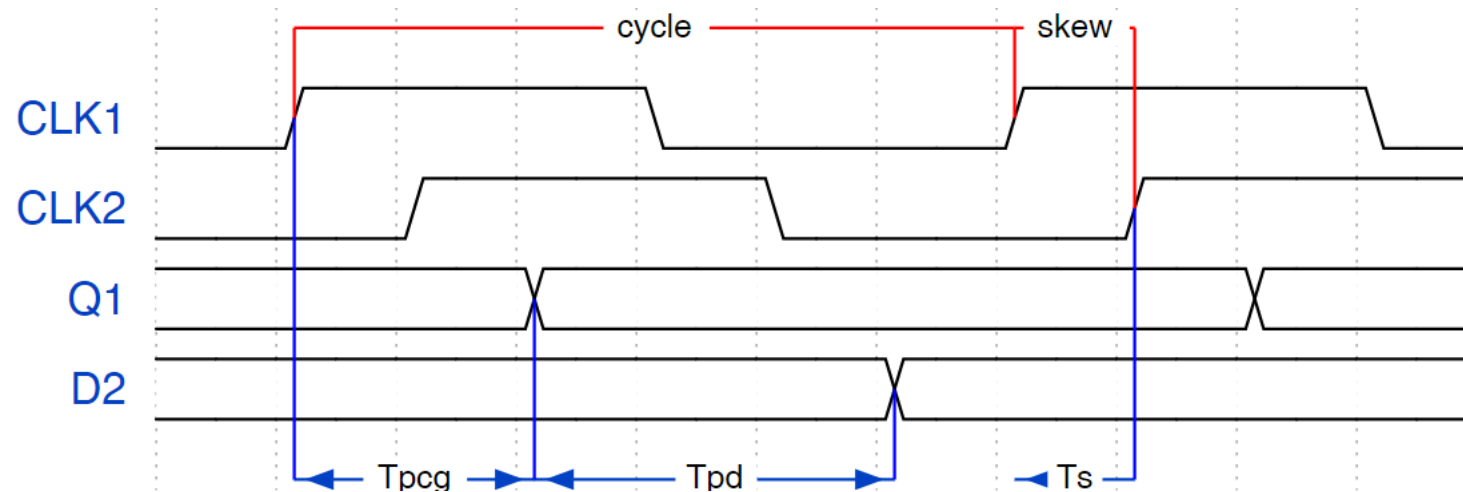
# Setup/Hold Time Criterion ■



- Setup time margin
  - $(T_{\text{cycle}} + T_{\text{skew}}) > (T_{\text{pcq}} + T_{\text{pd}} + T_{\text{setup}})$
- Hold time margin
  - $(T_{\text{ccq}} + T_{\text{cd}}) > (T_{\text{skew}} + T_{\text{hold}})$

# Setup Time Criterion ■

- Setup time margin
- $(T_{\text{cycle}} + T_{\text{skew}}) > (T_{\text{pcq}} + T_{\text{pd}} + T_{\text{setup}})$

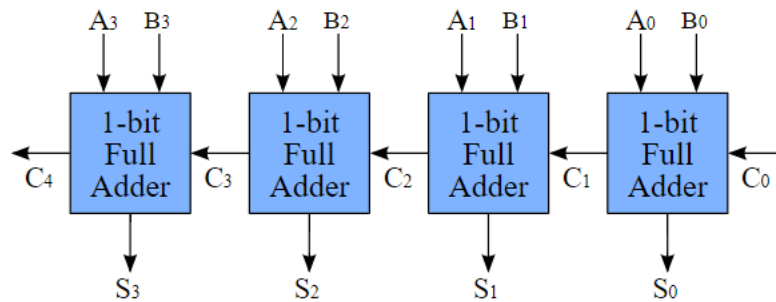


# Fix Setup Violation ■

## ■ Optimize Netlist

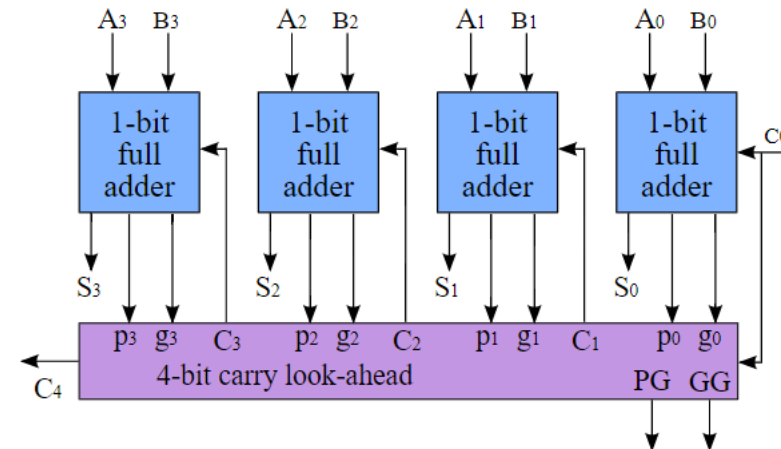
- Change to faster architecture

### Carry-Ripple Adder



->

### Carry-Lookahead Adder



$$\begin{aligned} C_4 = & G_3 \mid \\ & (G_2 \& P_3) \mid \\ & (G_1 \& P_2 \& P_3) \mid \\ & (G_0 \& P_1 \& P_2 \& P_3) \mid \\ & (C_0 \& P_0 \& P_1 \& P_2 \& P_3) \end{aligned}$$

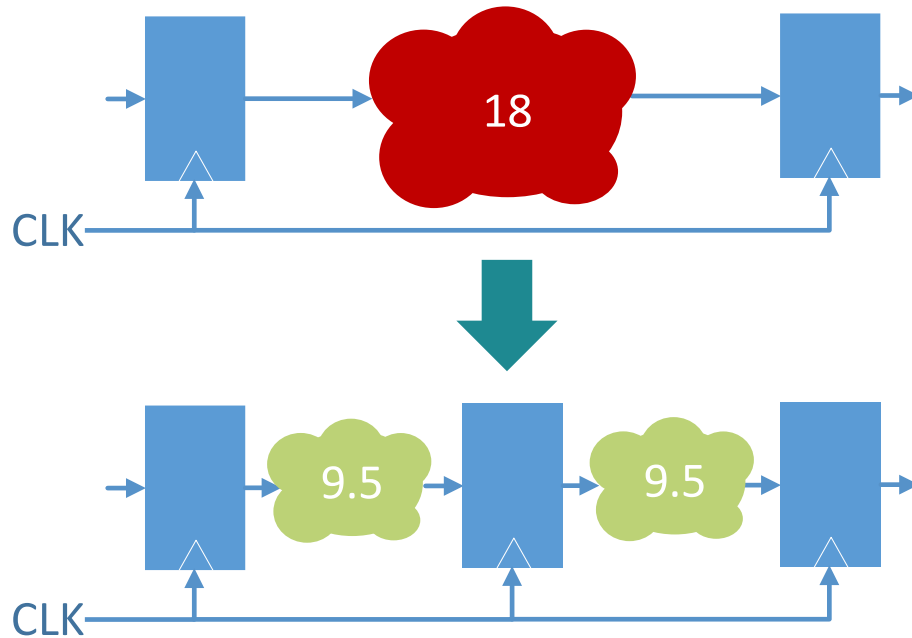
Ref. [https://en.wikipedia.org/wiki/Adder\\_\(electronics\)](https://en.wikipedia.org/wiki/Adder_(electronics))

Ref. [https://en.wikipedia.org/wiki/Carry-lookahead\\_adder](https://en.wikipedia.org/wiki/Carry-lookahead_adder)

# Fix Setup Violation .

## ■ Optimize Netlist

### – Pipeline



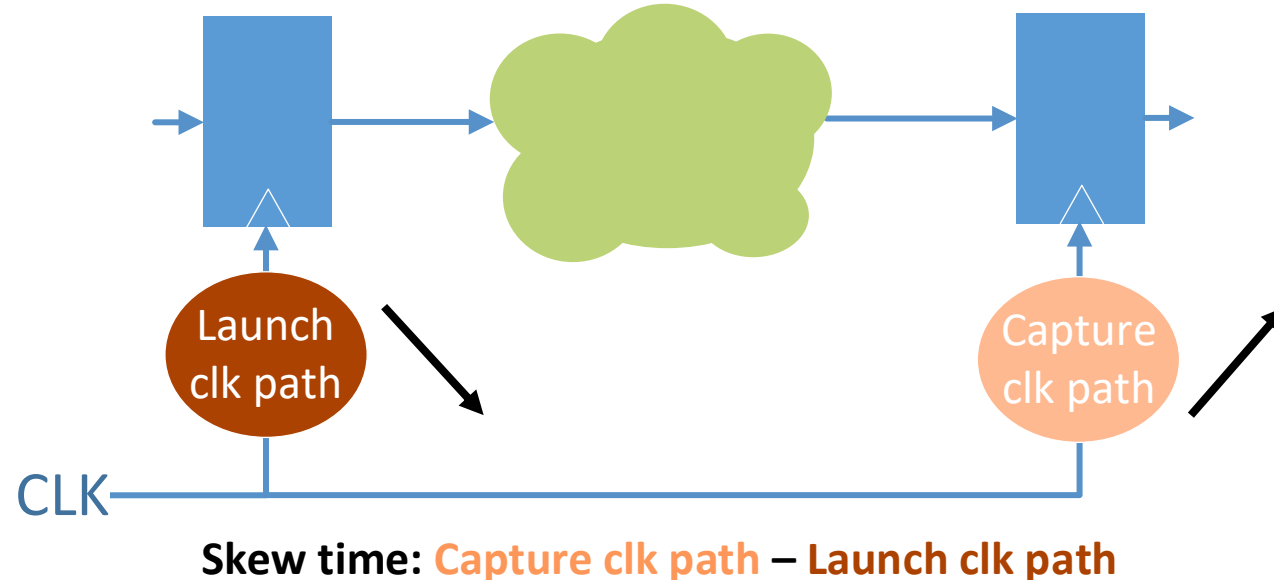
### – Retiming



# Fix Setup Violation ■

## ■ Adjust clock path

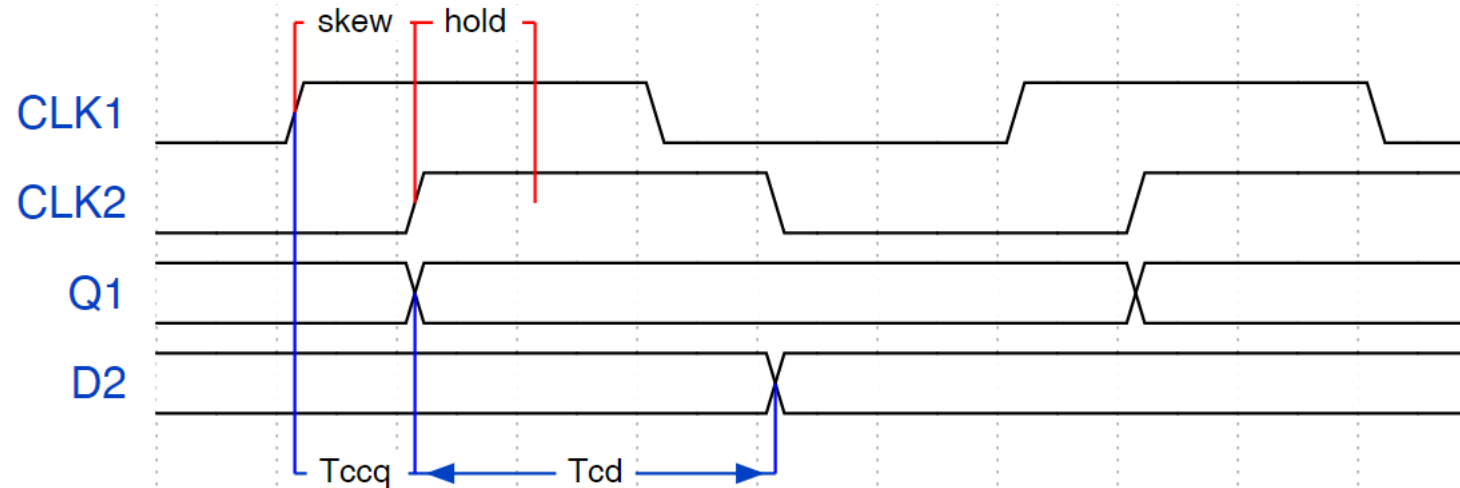
- Decrease **Launch clock path**
- Increase **Capture clock path**
- Make sure there are a setup margin on the next register from the capture clock and a hold margin on the previous register from the launch clock





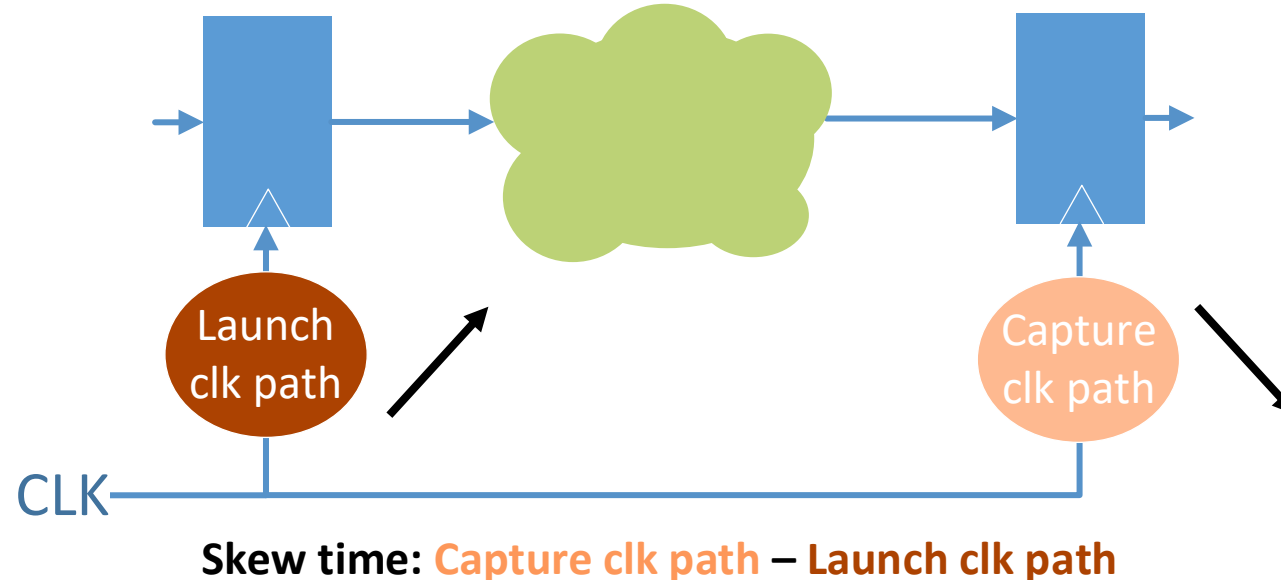
# Hold Time Criterion ■

- Hold time margin
- $(T_{ccq} + T_{cd}) > (T_{skew} + T_{hold})$
- Insert cells between Q1 and D2 to met hold time margin



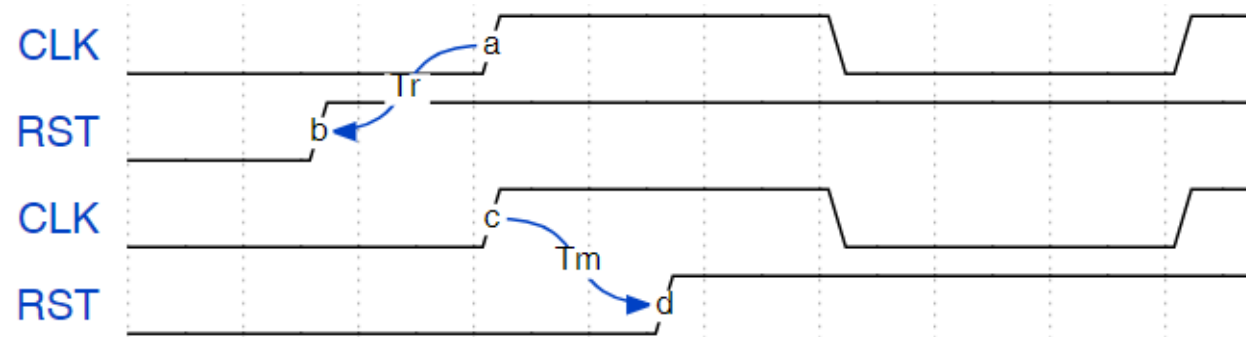
# Fix Hold Violation

- **Optimize Netlist**
  - Increase combination logic contamination delay (Tcd)
- **Adjust clock path**
  - Increase **Launch clock path**
  - Decrease **Capture clock path**



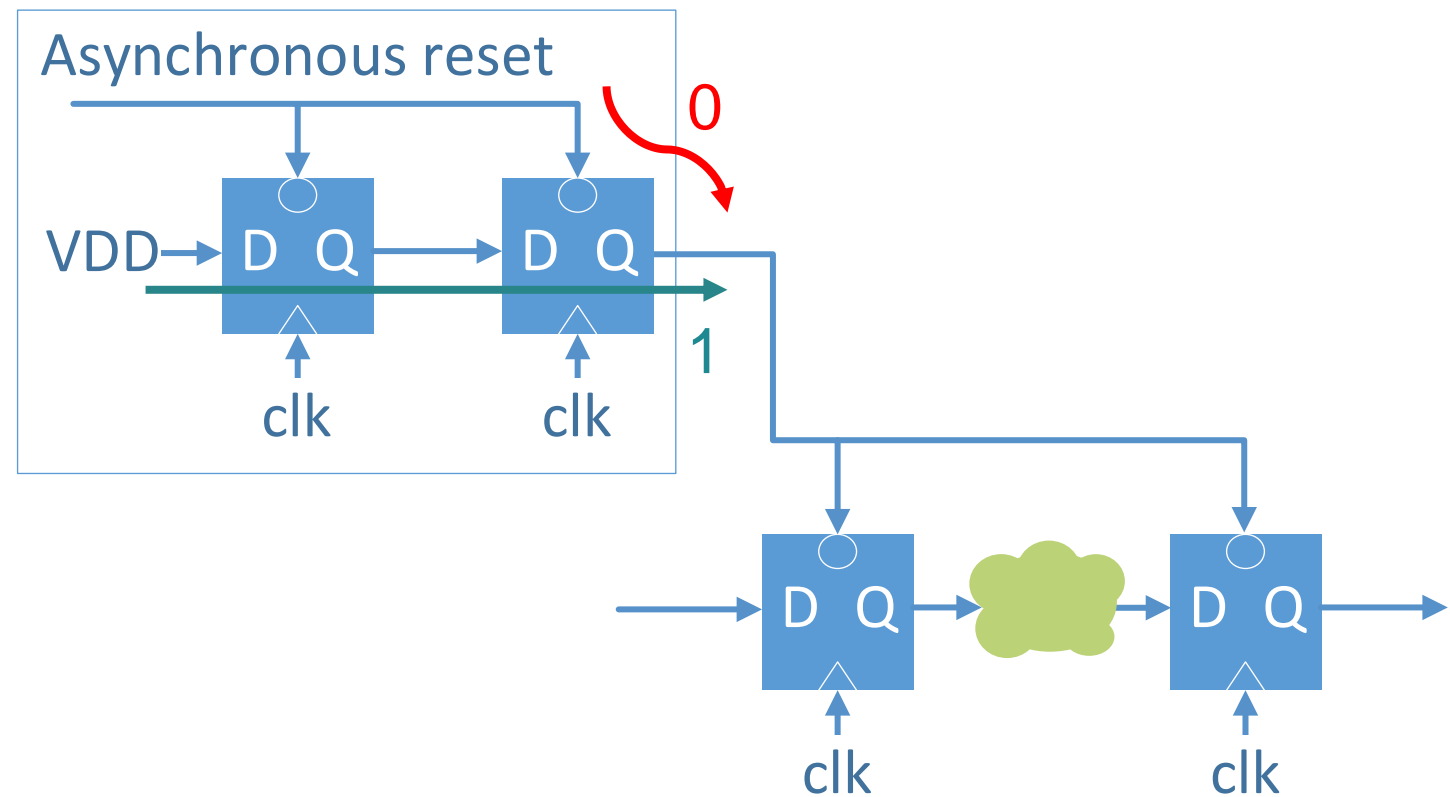
# Timing of D Flip-Flop ■

- Recovery time ( $T_r$ ) (for asynchronous reset)
  - The time that the reset signal must be stabilized before the clock edge
- Removal time ( $T_m$ ) (for asynchronous reset)
  - The time that the reset signal must be stabilized after the clock edge



# Reset Assertion And De-Assertion

- Asynchronous reset assertion timing scenarios
  - Reset assertion **immediately** causes the output to go 0
  - Reset de-assertion **waits for clock edge** to propagate the value 1 at input to output



# Fix Setup Violation by Synthesis ■

## ■ Set *group\_path* (Design Compiler command)

- DC will optimize every group, always from the worst path, until any of the following conditions are met
  - All the paths in the group meet the timing requirements
  - The worst path in the group is no longer optimized
- Sometimes we need to optimizing **non-worst paths**, modifying the group order by using the *group\_path* and setting weights

```
group_path [-weight weight_value] [-critical_range range_value] -default | -name group_name [-from from_list | -rise_from rise_from_list | -fall_from fall_from_list] [-through through_list | -rise_through rise_through_list | -fall_through fall_through_list] [-to to_list | -rise_to rise_to_list | -fall_to fall_to_list] [-comment comment_string]
```

- **critical\_range**: use the worst path as a reference, and optimize the slack within a certain range of paths
- **weight**: how much effort to spend on this group, take the value 0~100 (float), the larger value spend the more effort

# Fix Setup Violation by Back-end ■

## ■ Adjust placement and floorplan

- Interrelated macro to be placed together
- Reserved for space for the pins of macro
- Some macro need to be close to the port

## ■ Minimize data path delay

- **Change different Vt (threshold voltage) cells**
  - Speed:  $LVT > RVT > HVT$
  - Power:  $HVT < RVT < LVT$
- **Insert buffer:** shorting the net or decrease the fanout
- **Size up cell:** change X1 cell to X4, X8...
- **Layer assignment:** High level metal has less resistance



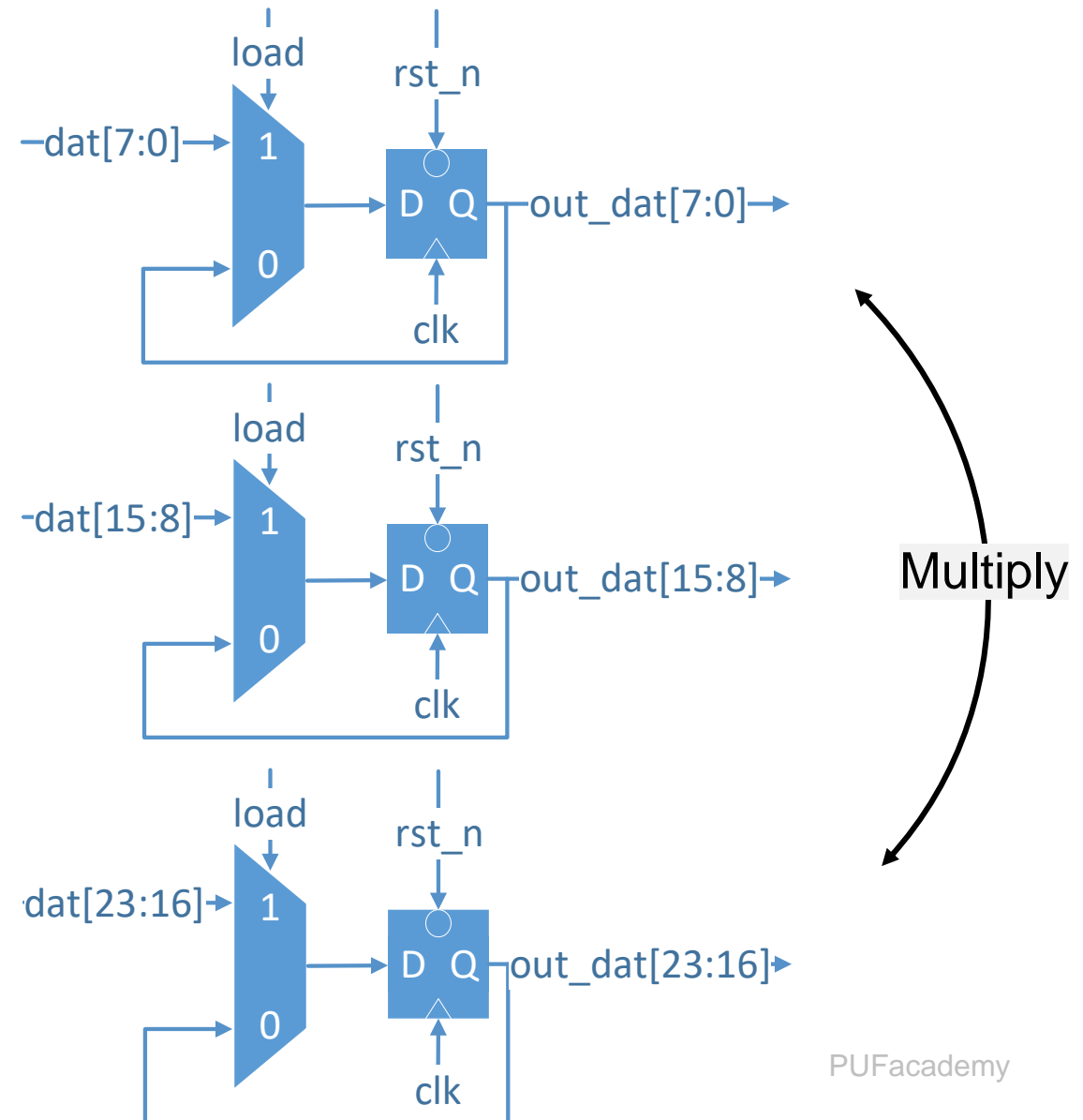
# Agenda ■

1. Sequential Logic
2. Static Timing Analysis
3. Generate Construct
4. Wavedrom tool

# Generate Construct ■

- Generate constructs are used to either **multiply** or **conditionally** instantiate generate blocks into a model
- Loop generate constructs (for multiply)

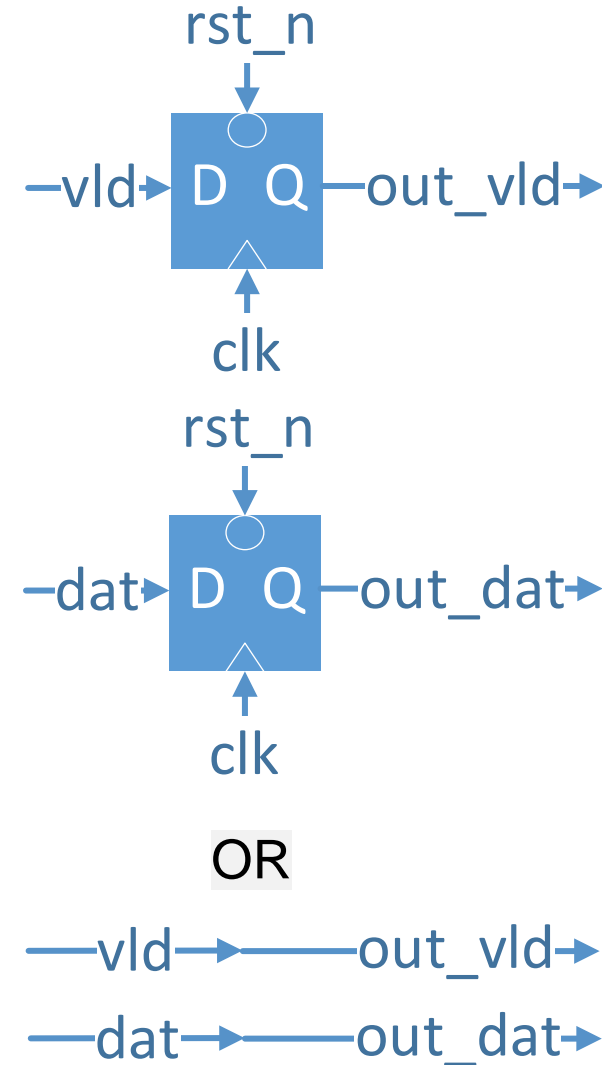
```
13 generate
14   genvar i;
15   for(i=0;i<DAT_BYTE;i=i+1) begin : DATA_LOAD
16     always@(posedge clk or negedge rst_n) begin
17       if(~rst_n) begin
18         out_dat[i*8+:8] <= {8'b0};
19       end
20       else if(load)begin
21         out_dat[i*8+:8] <= dat[i*8+:8];
22       end
23     end
24   end
25 endgenerate
```



# Generate Construct

- Conditional generate constructs (for conditionally)

```
12 generate
13   if(REG_FLAG)begin : REG_OUT
14     always@(posedge clk or negedge rst_n) begin
15       if(~rst_n) begin
16         out_vld <= 1'b0;
17         out_dat <= 32'b0;
18       end
19       else begin
20         out_vld <= vld;
21         out_dat <= dat;
22       end
23     end
24   end
25   else begin : NO_REG_OUT
26     always@(*) begin
27       out_vld = vld;
28       out_dat = dat;
29     end
30   end
31 endgenerate
```



# Generate Construct ■

- genvar cannot be reused in one file (even in different generate for)



```
12 //genvar reused
13 generate
14   genvar i;
15   for(i=0;i<10;i=i+1) begin : RTL1
16     //...
17   end
18 endgenerate
19
20 generate
21   genvar i;
22   for(i=0;i<10;i=i+1) begin : RTL2
23     //...
24   end
25 endgenerate
```

- Don't give a generate block the same name as other signal



```
50 //the same name with a signal
51 wire [31:0] dat;
52 generate
53   genvar i;
54   for(i=0;i<10;i=i+1) begin : dat
55     //...
56   end
57 endgenerate
```

- Two generate can not share one genvar



```
27 //sharing one genvar
28 genvar i;
29 generate
30   for(i=0;i<10;i=i+1) begin : RTL1
31     //...
32   end
33 endgenerate
34
35 generate
36   for(i=0;i<10;i=i+1) begin : RTL2
37     //...
38   end
39 endgenerate
```

- Give every if-generate case a name



```
41 //give a generate for a name
42 generate
43   genvar i;
44   for(i=0;i<10;i=i+1) begin : RTL1
45     //...
46   end
47 endgenerate
```

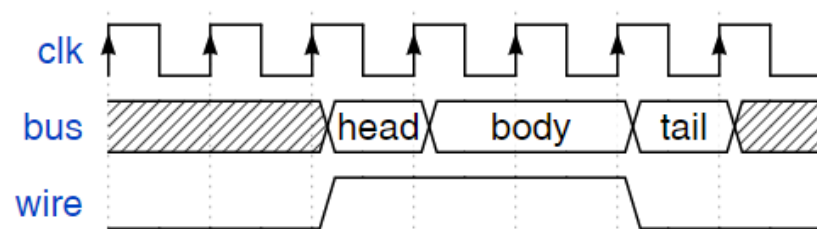
# Agenda ■

1. Sequential Logic
2. Static Timing Analysis
3. Generate Construct
4. Wavedrom tool

# Wavedrom

- <https://wavedrom.com/>
- Wavedrom is a JavaScript application which is a format describes digital timing diagrams

```
1 { signal: [  
2   { name: "clk", wave: "P....." },  
3   { name: "bus", wave: "x.==.x", data: ["head", "body", "tail", "data"] },  
4   { name: "wire", wave: "0.1..0." }  
5 ]}
```



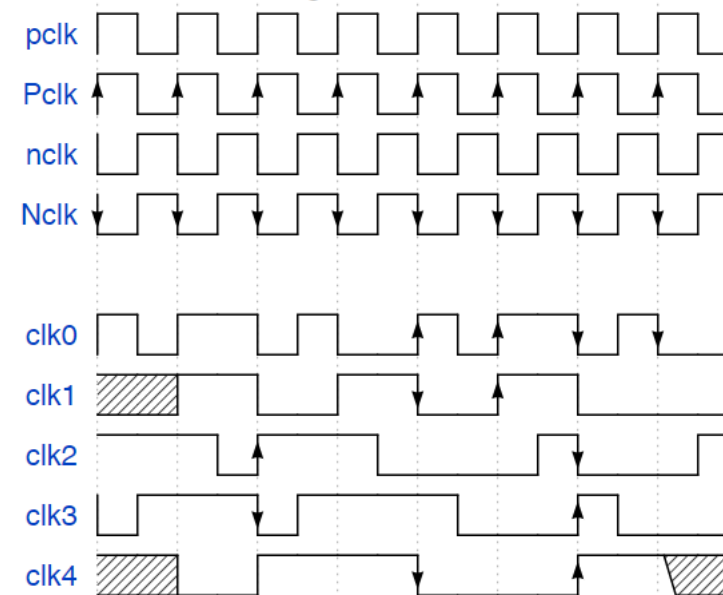


# Adding Clock

## ■ Adding clock: **pPnNhl**

```
1 { signal: [  
2   { name: "pclk", wave: 'p.....' },  
3   { name: "Pclk", wave: 'P.....' },  
4   { name: "nclk", wave: 'n.....' },  
5   { name: "Nclk", wave: 'N.....' },  
6   {} ,  
7   { name: 'clk0', wave: 'phnlPHNL' },  
8   { name: 'clk1', wave: 'xhlhLHl.' },  
9   { name: 'clk2', wave: 'hpHpLnLn' },  
10  { name: 'clk3', wave: 'nhNhplPl' },  
11  { name: 'clk4', wave: 'xlh.L.Hx' },  
12 ] }
```

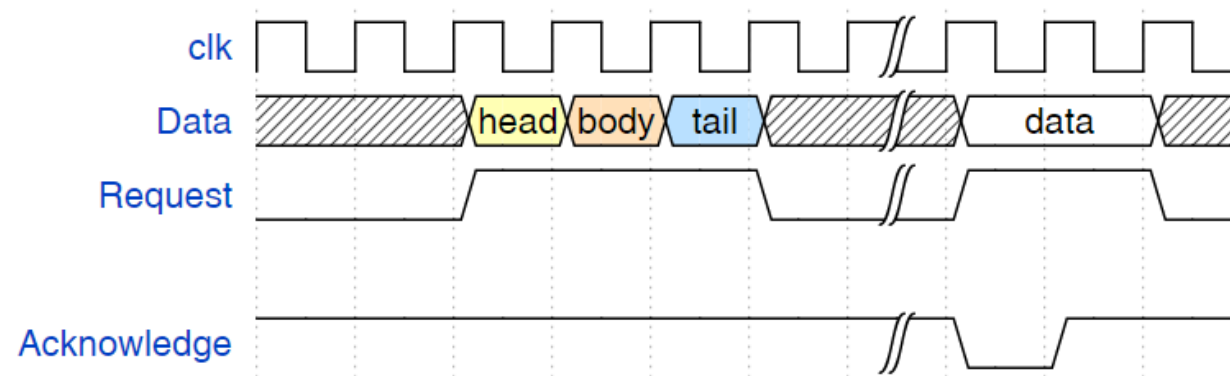
and the rendered diagram:



# Colors, Spacers And Gaps

- Colors : 3~9
- Spacers: {}
- Gaps: |

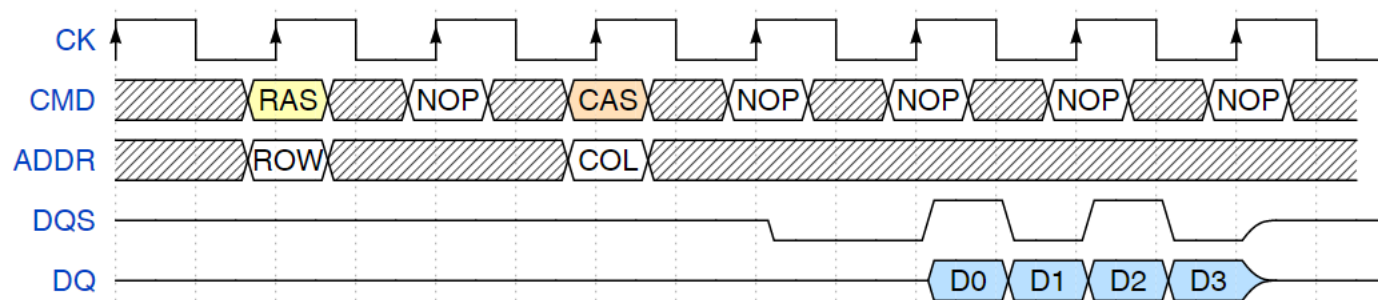
```
1 { signal: [  
2   { name: "clk",      wave: "p.....|..." },  
3   { name: "Data",     wave: "x.345x|=.x", data: ["head", "body", "tail", "data"] },  
4   { name: "Request",  wave: "0.1..0|1.0" },  
5   {}  
6 { name: "Acknowledge", wave: "1.....|01." }  
7 ]}
```



# Period And Phase

- Period : Scale up waveform N times
- Phase: Shift waveform N cycle

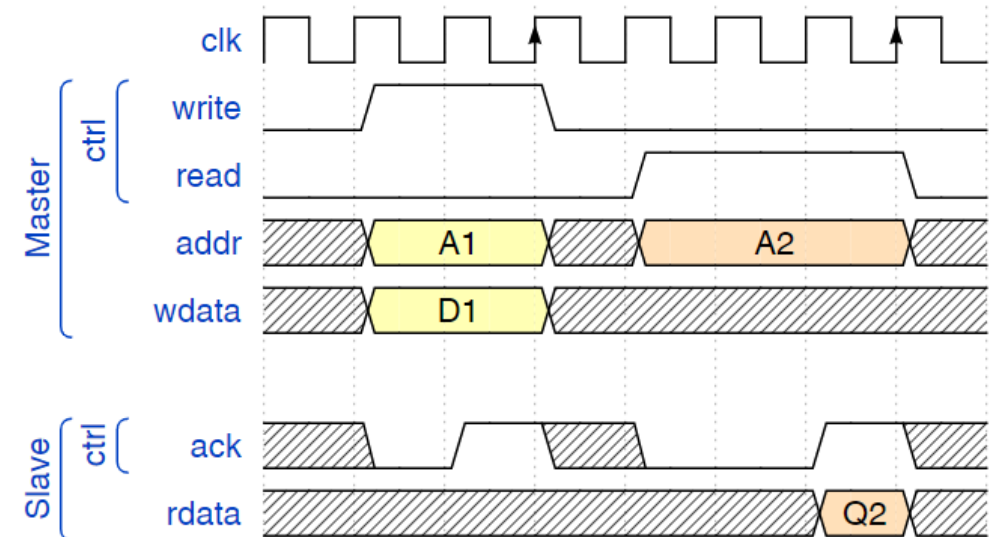
```
1 { signal: [  
2   { name: "CK", wave: "P.....", period: 2 },  
3   { name: "CMD", wave: "x.3x=x4x=x=x=x", data: "RAS NOP CAS NOP NOP NOP NOP", phase: 0.5 },  
4   { name: "ADDR", wave: "x.=x..=x.....", data: "ROW COL", phase: 0.5 },  
5   { name: "DQS", wave: "z.....0.1010z." },  
6   { name: "DQ", wave: "z.....5555z.", data: "D0 D1 D2 D3" }  
7 ]}
```



# Groups

## ■ Classify groups

```
1 { signal: [  
2   { name: 'clk', wave: 'p..Pp..P'},  
3   ['Master',  
4     ['ctrl',  
5       {name: 'write', wave: '01.0....'},  
6       {name: 'read', wave: '0...1..0'}  
7     ],  
8     { name: 'addr', wave: 'x3.x4..x', data: 'A1 A2'},  
9     { name: 'wdata', wave: 'x3.x....', data: 'D1' },  
10  ],  
11  {}},  
12  ['Slave',  
13    ['ctrl',  
14      {name: 'ack', wave: 'x01x0.1x'},  
15    ],  
16    { name: 'rdata', wave: 'x.....4x', data: 'Q2'},  
17  ]  
18 ]}
```



Feedback to us ■

NTHU 晶片安全設計 回饋單



<https://forms.office.com/r/DYDu8vLaWN>

# Thank you!



Visit our website: [pufacademy.com](https://pufacademy.com)

Contact us: [PUFacademy@pufsecurity.com](mailto:PUFacademy@pufsecurity.com)

