

AI ASIC: Design and Practice (ADaP) Fall 2024 Advanced Verilog HDL

燕博南

Outline



- Part 1
 - Verilog HDL grammar, operators, synthesizable design
- Part 2
 - Finite State Machine
 - Pitfalls
- Part 3
 - Timing Disaster

HDL: Hardware Description Language

LUT: Look-up table

Stdcell: standard cell

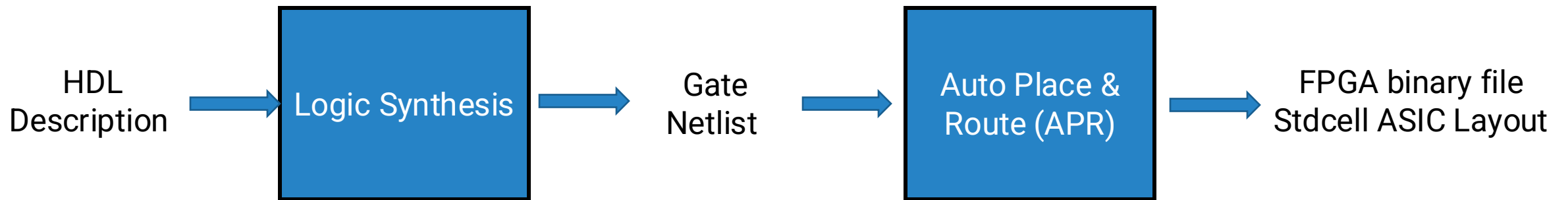
Part 1

Verilog HDL grammar, operators, synthesizable design

Digital Circuit Design Flow

- Using Verilog you can write an executable functional specification that
- documents exact behavior of all the modules and their interfaces
- can be tested & refined until it does what you want

An HDL description is the first step in a mostly automated process to build an implementation directly from the behavioral model



- HDL -> Logic
- Map to target lib (stdcell/LUTs)
- Optimize speed, area

- Create floorplan blocks
- Place cells in blocks
- Route interconnect
- Optimize iteratively

Basic Building - Module

```
1 // single-line comments
2 /* multi-line
3 comments
4 */
5 module name(
6     input a,b,
7     input [31:0] c,
8     output z,
9     output reg [3:0] s
10    ); ← Don't forget ";" here
11 // declarations of internal signals, registers
12 // combinational logic: assign
13 // sequential logic: always @ (posedge clock)
14 // module instances
15 endmodule
```

In Verilog we design modules, one of which will be identified as our top-level module. Modules usually have named, directional ports (specified as **input**, **output**) which are used to communicate with the module.

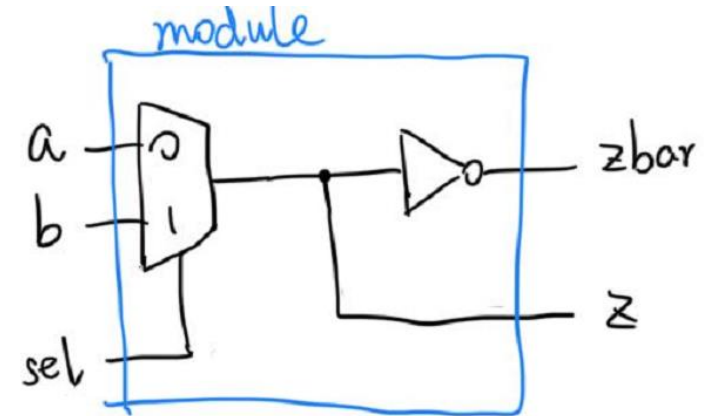
- Format: HDL ignores space " ", it only recognize ";"

Wires & Registers

Wires

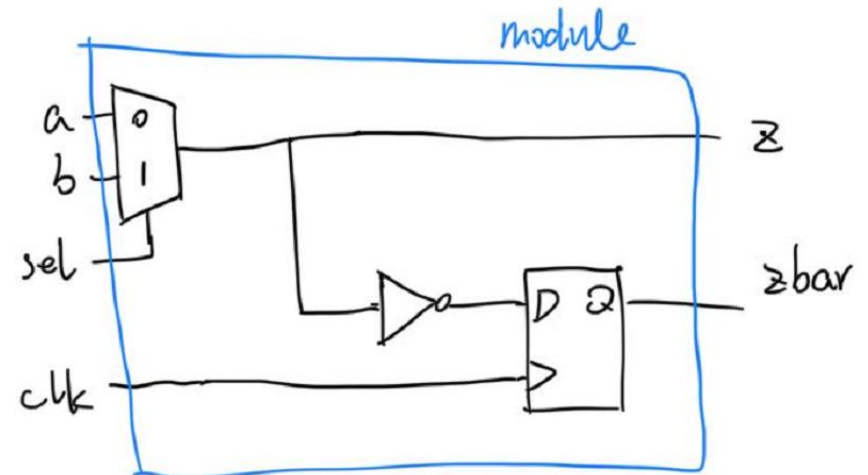
```
1 // 2-to-1 multiplexer with dual-polarity outputs
2 module mux2(
3     input a,b,sel,
4     output z,zbar
5 );
6 // again order doesn't matter (concurrent execution!)
7 // syntax is "assign LHS = RHS" where LHS is a wire/bus
8 // and RHS is an expression
9 assign z = sel ? b : a;
10 assign zbar = ~z;
11 endmodule
```

Directly connect!



Regs

```
1 // 2-to-1 multiplexer with dual-polarity outputs
2 module weird_mux2(
3     input a,b,sel,
4     output z,
5     output reg zbar
6 );
7 // again order doesn't matter (concurrent execution!)
8 // syntax is "assign LHS = RHS" where LHS is a wire/bus
9 // and RHS is an expression
10 assign z = sel ? b : a;
11
12 always @(posedge clk) begin
13     zbar = ~z;
14 end
15
16 endmodule
```



Secrets of Wires & Regs

- Without “reg” declaration, variables are always wires
- Reg can only be output and inside signals
- Caution!
 - Wires can only be changed using “assign” outside “always”
 - Regs can only be changed inside “always”

```
10  assign zbar = ~z;
```

```
12  always @(posedge clk) begin  
13      |      zbar = ~z;  
14  end
```

Operators

Arithmetic	*	Multiply
	/	Division
	+	Add
	-	Subtract
	%	Modulus
	+	Unary plus
	-	Unary minus
Logical	!	Logical negation
	&&	Logical and
		Logical or
Relational	>	Greater than
	<	Less than
	>=	Greater than or equal
	<=	Less than or equal
	Equality	==
!=		inequality
Shift		>>
	<<	Left shift
	<<<, >>>	Arithmetic shift

Reduction	~	Bitwise negation
	~&	nand
		or
	~	nor
	^	xor
	^~	xnor
	~^	xnor

Concatenation	{}	Concatenation
Conditional	?	conditional

What are the difference between (logic) shift and arithmetic shift?

Numeric Constants

Constant values can be specified with a specific width and radix:

123 // default: decimal radix, 32-bit width

'd123 // 'd = decimal radix

1'd3 ??

'h7B // 'h = hex radix

'o173 // 'o = octal radix

is 1'd1

'b111_1011 // 'b = binary radix, “_” are ignored

'hxx // can include X, Z or ? in non-decimal constants

16'd5 // 16-bit constant 'b0000_0000_0000_0101

11'h1X? // 11-bit constant 'b001_XXXX_ZZZZ

By default constants are unsigned and will be extended with 0's on left if need be (if high-order bit is X or Z, the extended bits will be X or Z too).

You can specify a signed constant as follows:

8'shFF // 8-bit twos-complement representation of -1

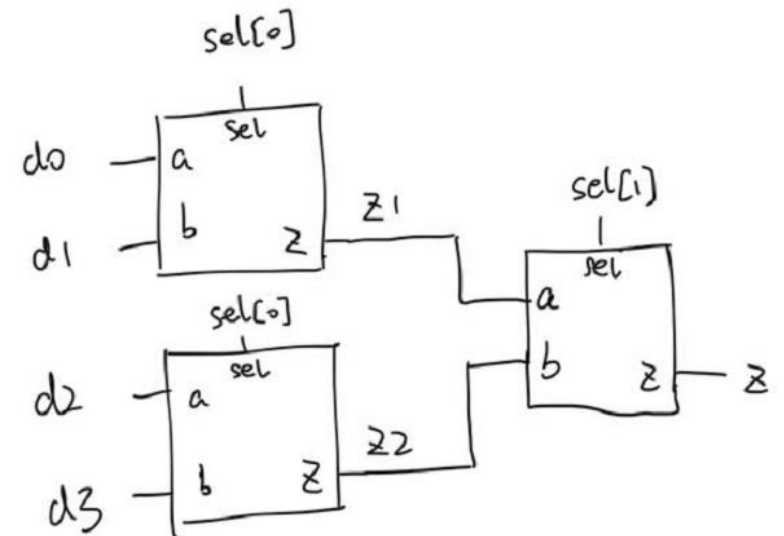
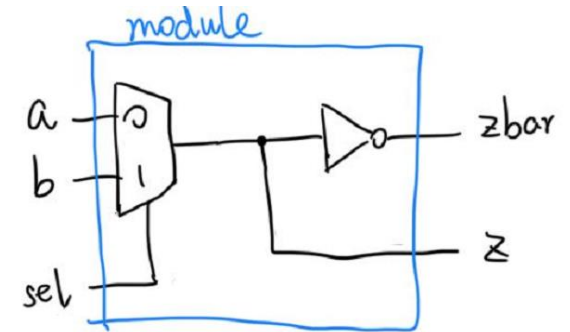
To be absolutely clear in your intent it's usually best to explicitly specify the width and radix.

Hierarchy: module instances

```

1 // 4-to-1 multiplexer
2 module mux4(input d0,d1,d2,d3, input [1:0] sel, output z);
3     wire z1,z2;
4     // instances must have unique names within current module.
5     // connections are made using .portname(expression) syntax.
6     // once again order doesn't matter...
7     mux2 m1(.sel(sel[0]),.a(d0),.b(d1),.z(z1)); // not using zbar
8     mux2 m2(.sel(sel[0]),.a(d2),.b(d3),.z(z2));
9     mux2 m3(.sel(sel[1]),.a(z1),.b(z2),.z(z));
10    // could also write "mux2 m3(z1,z2,sel[1],z,)" NOT A GOOD IDEA!
11 endmodule
    
```

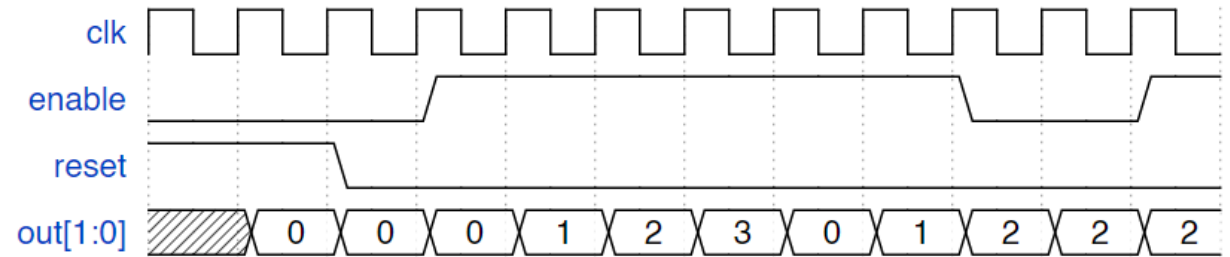
- Write all original names (style requirement)
- Connection are concurrently executed



Example 1: A counter

```
1 module counter (  
2   out      , // Output of the counter  
3   enable   , // enable for counter  
4   clk      , // clock Input  
5   reset    // reset Input  
6 );  
7  
8 output [1:0] out; //Output Ports  
9 input  enable, clk, reset; //Input Ports  
10  
11 reg [1:0] out; //Internal Variables  
12  
13 always @(posedge clk)  
14 if (reset) begin  
15 | out <= 2'b0 ;  
16 end else if (enable) begin  
17 | out <= out + 1;  
18 end  
19  
20 endmodule
```

What is the timing diagram?



- Beware of “before” & “after” clock edge

Verification: Simulation & Testbench

Design:

```
1 module counter (  
2   out      , // Output of the counter  
3   enable   , // enable for counter  
4   clk      , // clock Input  
5   reset    // reset Input  
6 );  
7  
8 output [1:0] out; //Output Ports  
9 input enable, clk, reset; //Input Ports  
10  
11 reg [1:0] out; //Internal Variables  
12  
13 always @(posedge clk)  
14 if (reset) begin  
15   out <= 2'b0 ;  
16 end else if (enable) begin  
17   out <= out + 1;  
18 end  
19  
20 endmodule
```

Testbench:

```
1 `timescale 1ns/1ps  
2 module Testbench;  
3  
4 wire [1:0] OUT;  
5 reg EN, CLK, RST;  
6  
7 initial CLK = 0;  
8 always #2 CLK=~CLK;  
9  
10 initial begin  
11   #1  
12   EN = 0;  
13   RST = 1;  
14   #4  
15   EN = 1;  
16   RST = 0;  
17   #(4*7)  
18   EN = 0;  
19 end  
20  
21 counter u1(  
22   .out (OUT)      , // Output of the counter  
23   .enable (EN)    , // enable for counter  
24   .clk (CLK)      , // clock Input  
25   .reset (RST)    // reset Input  
26 );  
27  
28 endmodule
```

Testbench has no ports

Synthesizable

Not synthesizable



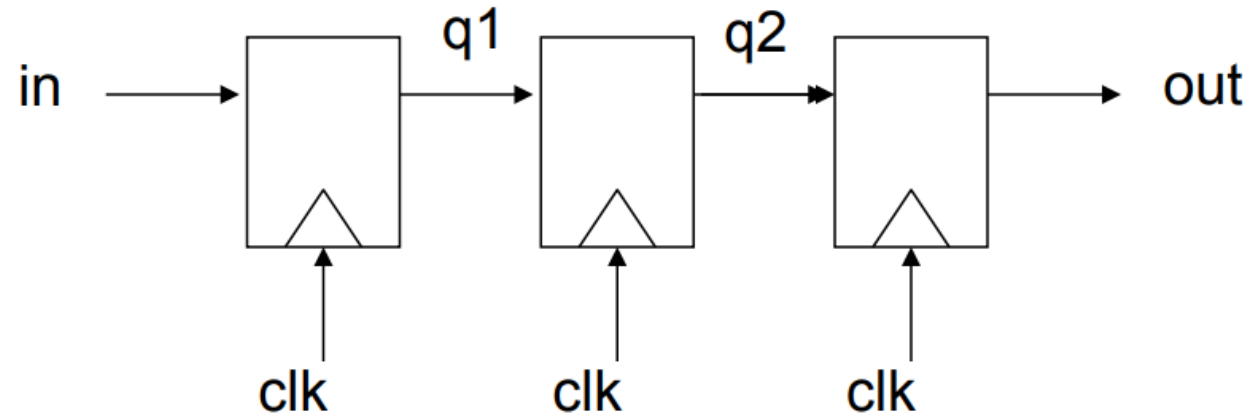
Cannot easily converted to circuits

Initial
#

```
1  `timescale 1ns/1ps
2  module Testbench;
3
4  wire [1:0] OUT;
5  reg EN, CLK, RST;
6
7  initial CLK = 0;
8  always #2 CLK=~CLK;
9
10 initial begin
11     #1
12     EN = 0;
13     RST = 1;
14     #4
15     EN = 1;
16     RST = 0;
17     #(4*7)
18     EN = 0;
19 end
20
21 counter u1(
22     .out (OUT)    , // Output of the counter
23     .enable (EN) , // enable for counter
24     .clk (CLK)   , // clock Input
25     .reset (RST) // reset Input
26 );
27
28 endmodule
```



Blocking and Non-blocking



```
1 // shift register
2 reg q1,q2,out;
3 always @(posedge clk) begin
4     q1 = in;
5     q2 = q1;
6     out = q2;
7 end
```

Wrong!

```
1 // shift register
2 reg q1,q2,out;
3 always @(posedge clk) q1 <= in;
4 always @(posedge clk) q2 <= q1;
5 always @(posedge clk) out <= q2;
```

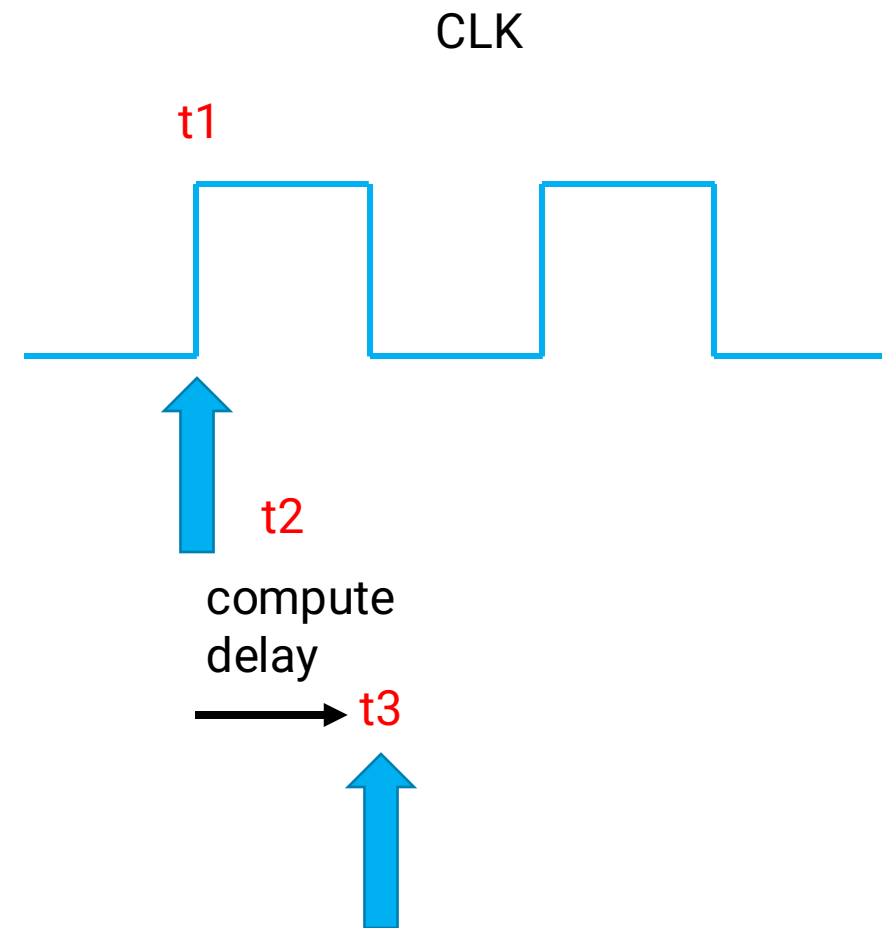
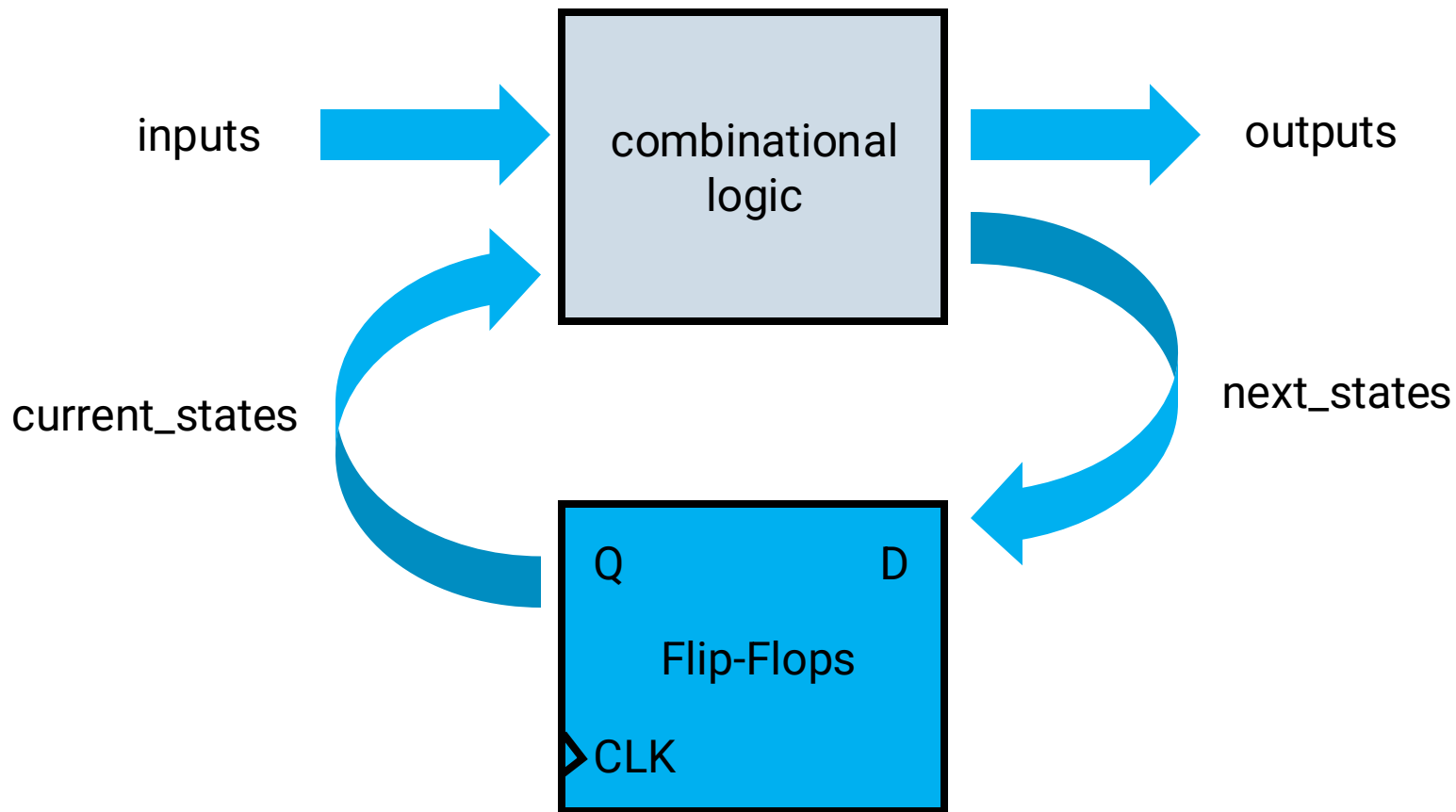
Correct!

Part 2

Finite State Machine

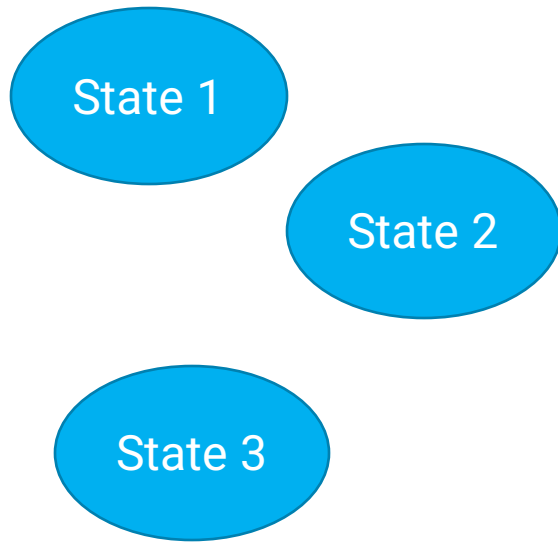
Finite State Machine

- Hardware/Circuits

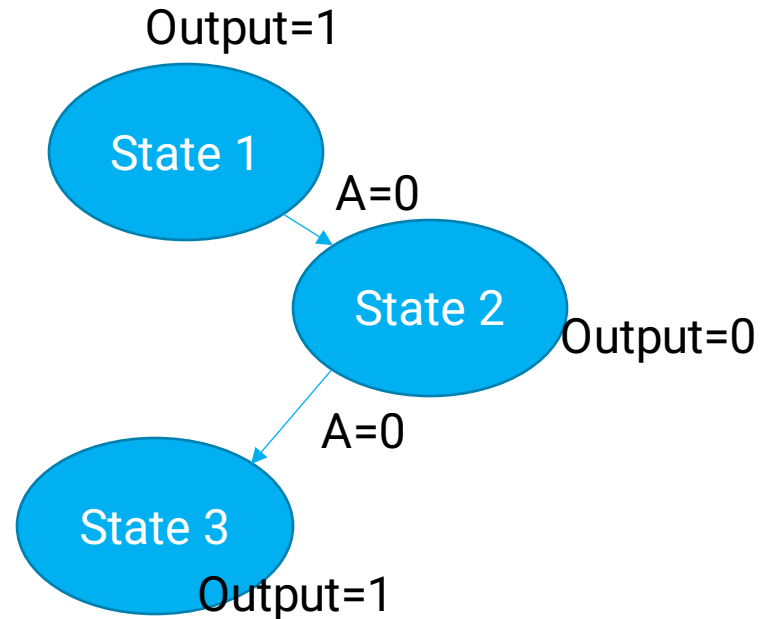


Time point: t_1, t_2, t_3

Design Methodologies & Templates



Step 1: define states



Step 2: draw state-transfer diagram

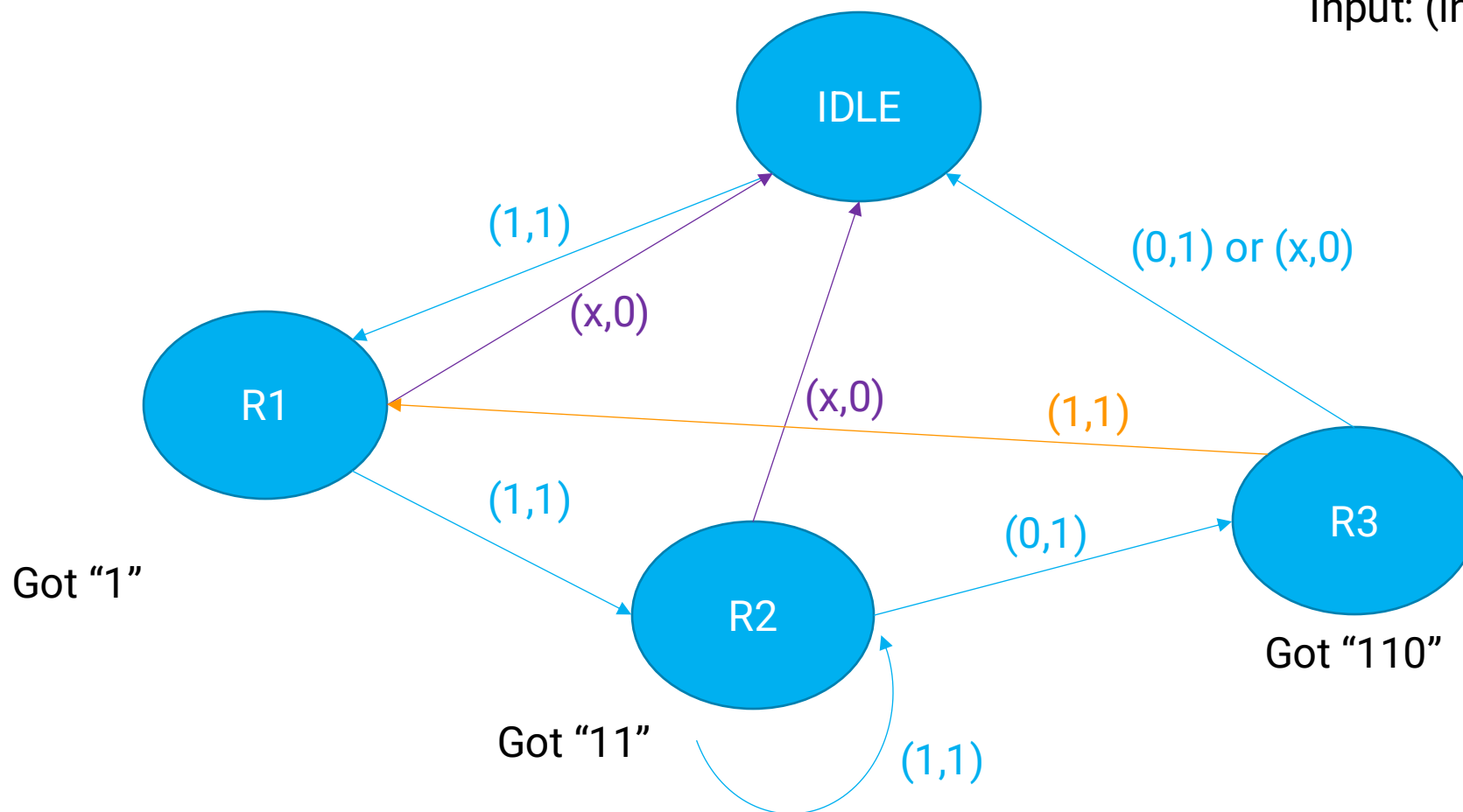
```
1 module safe_state_machine (
2     input  clk, data_in, reset,
3     output reg [1:0] data_out
4 );
5
6     reg [1:0] state;
7     // Declare states
8     parameter S0 = 0, S1 = 1, S2 = 2, S3 = 3;
9     // Output depends only on the state
10    always @ (state) begin
11        case (state)
12            S0:
13                data_out = 2'b01;
14            S1:
15                data_out = 2'b10;
16            S2:
17                data_out = 2'b11;
18            S3:
19                data_out = 2'b00;
20            default:
21                data_out = 2'b00;
22        endcase
23    end
24
25    // Determine the next state
26    always @ (posedge clk or posedge reset) begin
27        if (reset)
28            state <= S0;
29        else
30            case (state)
31                S0:
32                    state <= S1;
33                S1:
34                    if (data_in)
35                        state <= S2;
36                    else
37                        state <= S1;
38                S2:
39                    if (data_in)
40                        state <= S3;
41                    else
42                        state <= S1;
43                S3:
44                    if (data_in)
45                        state <= S2;
46                    else
47                        state <= S3;
48            endcase
49    end
```

Step 3: fill in the template

FSM Example 1 – Step 1 & 2

Question: build circuits that outputs 1 pulse, whenever receives “110”

Input: (input, rstb)



FSM Example 1 – Step 3

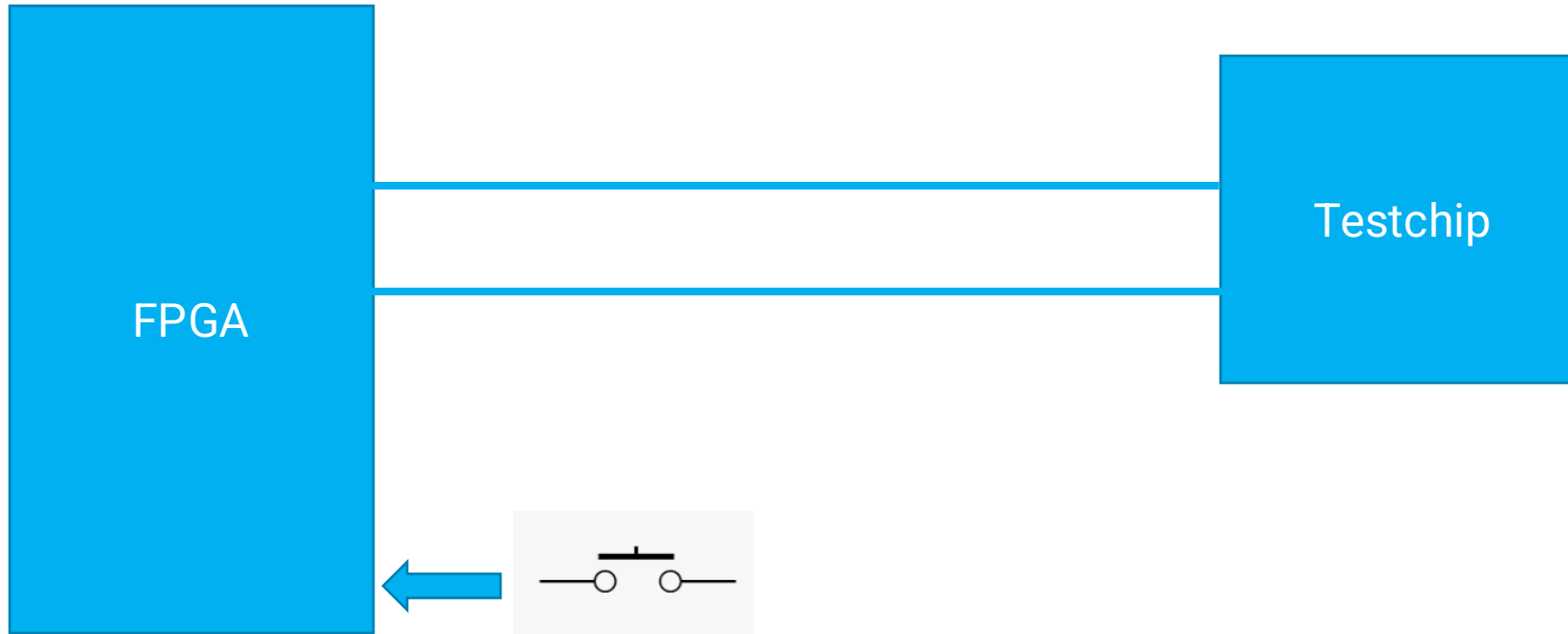
Question: build circuits that outputs 1 pulse, whenever receives “110”

```
1  module PasscodeDetector (  
2      input  clk, data_in, rstb,  
3      output reg data_out  
4  );  
5  
6      reg [1:0] state;  
7      // Declare states  
8      parameter STAT_IDLE = 0,  
9              STAT_R1 = 1,  
10             STAT_R2 = 2,  
11             STAT_R3 = 3;  
12     // Output depends only on the state  
13     always @ (state) begin  
14         if(state == STAT_R3) begin  
15             data_out <= 1; // alarming!  
16         end  
17         else begin  
18             data_out <= 0;  
19         end  
20     end
```

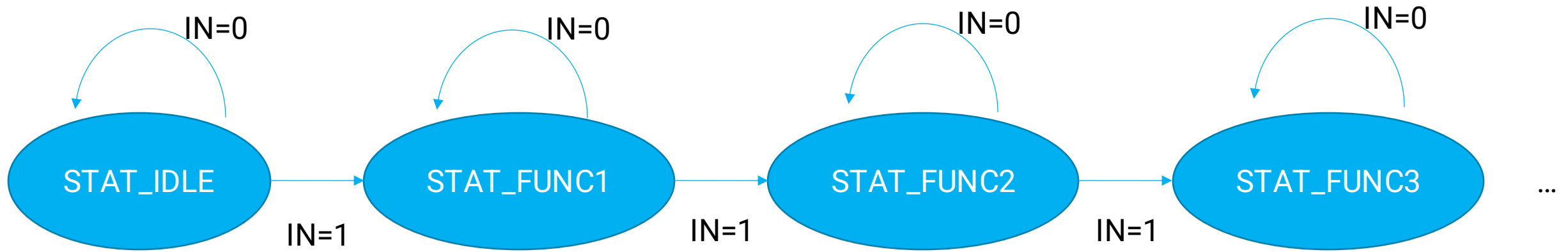
```
22     // Determine the next state  
23     always @ (posedge clk) begin  
24         if (~rstb)  
25             state <= STAT_IDLE;  
26         else  
27             case (state)  
28                 STAT_IDLE: begin  
29                     if(data_in==1) begin  
30                         state <= STAT_R1;  
31                     end  
32                 end  
33                 STAT_R1: begin  
34                     if (data_in==1)  
35                         state <= STAT_R2;  
36                     else  
37                         state <= STAT_IDLE;  
38                 end  
39                 STAT_R2: begin  
40                     if (data_in==0)  
41                         state <= STAT_R3;  
42                     else  
43                         state <= STAT_R2;  
44                 end  
45                 STAT_R3: begin  
46                     if(data_in==1) begin  
47                         state <= STAT_R1;  
48                     end  
49                     else begin  
50                         state <= STAT_IDLE;  
51                     end  
52                 end  
53             endcase  
54         end  
55     endmodule
```

FSM Example 2 Auto Chip Testing Environment

Push button to test Function 1, Function 2, Function 3, ... in series



FSM Example 2 Step 1&2



Step 3 is so easy... that you can fill it yourself.

Another Question

Philosophy behind:
Use “state variable” to label timing

Another Q:
How to generate custom waveform
after entering some state?

Solution: setup an counter variable
Do everything at its pace
[! DO NOT ABUSE. This may cause
large comparing logic.]



Possibility of Nested State Machine!

```
1  always @ (posedge clk or posedge reset) begin
2      if (reset)
3          state <= S0;
4      else begin
5          case (state)
6              S0:
7                  data_out = 2'b01;
8              S1: begin
9                  if(cnt==10'd1) begin
10                     //do what you want
11                 end
12                 else if (cnt<=10'd5) begin
13                     //do what you want
14                 end
15                 else if (cnt<=10'd20) begin
16                     //do what you want
17                 end
18                 else begin
19                     //do what you want
20                 end
21             end
22         end
23         S2:
24             data_out = 2'b11;
25         S3:
26             data_out = 2'b00;
27         default:
28             data_out = 2'b00;
29     endcase
30 end
31 end
```

- State Machine vs. AI
 - State machines exhaustively cover all cases, which is impossible in AI agent design.
- State machine is appropriate for small scale designs.
- Nested state machines:
 - normally, do not nest inside FSM in >2 folds.
 - Otherwise, too complicated **timing path** dependency.

Verilog HDL中的For循环

```
module addergen1 (co, sum, a, b, ci);  
    parameter SIZE = 4;  
    output [SIZE-1:0] sum;  
    output co;  
    input [SIZE-1:0] a, b;  
    input ci;  
    wire [SIZE :0] c;  
    wire [SIZE-1:0] t [1:3];  
    genvar i;  
    assign c[0] = ci;
```

```
    for(i=0; i<SIZE; i=i+1) begin:bit  
        xor g1 ( t[1][i], a[i], b[i]);  
        xor g2 ( sum[i], t[1][i], c[i]);  
        and g3 ( t[2][i], a[i], b[i]);  
        and g4 ( t[3][i], t[1][i], c[i]);  
        or g5 ( c[i+1], t[2][i], t[3][i]);  
    end  
    assign co = c[SIZE];  
endmodule
```

Hierarchical gate instance names are:

xor gates: bit[0].g1 bit[1].g1 bit[2].g1 bit[3].g1
bit[0].g2 bit[1].g2 bit[2].g2 bit[3].g2
and gates: bit[0].g3 bit[1].g3 bit[2].g3 bit[3].g3
bit[0].g4 bit[1].g4 bit[2].g4 bit[3].g4
or gates: bit[0].g5 bit[1].g5 bit[2].g5 bit[3].g5
Generated instances are connected with
multidimensional nets t[1][3:0] t[2][3:0] t[3][3:0]
(12 nets total)

“Generate”

- 用For循环生成电路

```
module gray2bin1 (bin, gray);  
    parameter SIZE = 8; output [SIZE-1:0] bin;  
    input [SIZE-1:0] gray;  
    // this module is parameterizable  
    genvar i;  
    generate  
        for (i=0; i<SIZE; i=i+1) begin:bit  
            assign bin[i] = ^gray[SIZE-1:i];  
            // i refers to the implicitly defined localparam whose  
            // value in each instance of the generate block is  
            // the value of the genvar when it was elaborated.  
        end  
    endgenerate  
endmodule
```

Part 3

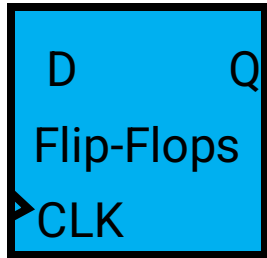
Timing

- Central Question: How to make faster computer? (performance)
- Q1: What are the limitations for timing?
 - Timing & Delay Mechanism
- Q2: How to Analysis Timing in VLSI?
 - Static Timing Analysis (STA)
- Q3: How to improve the design?
 - Retiming in HDL

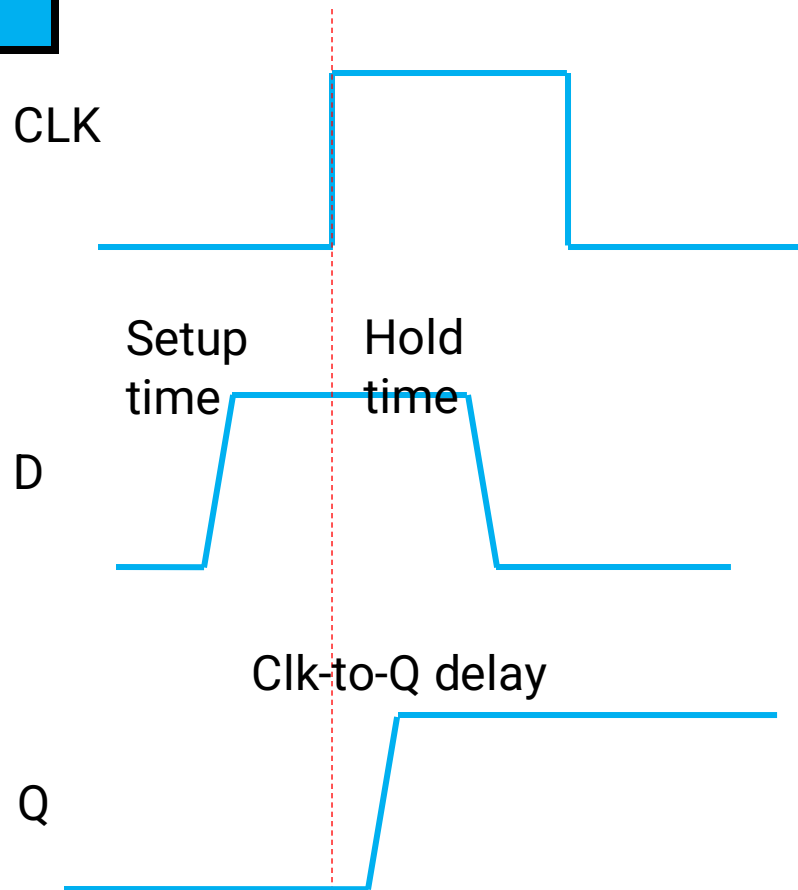
Timing Part: Q1

What are the limitations for timing?

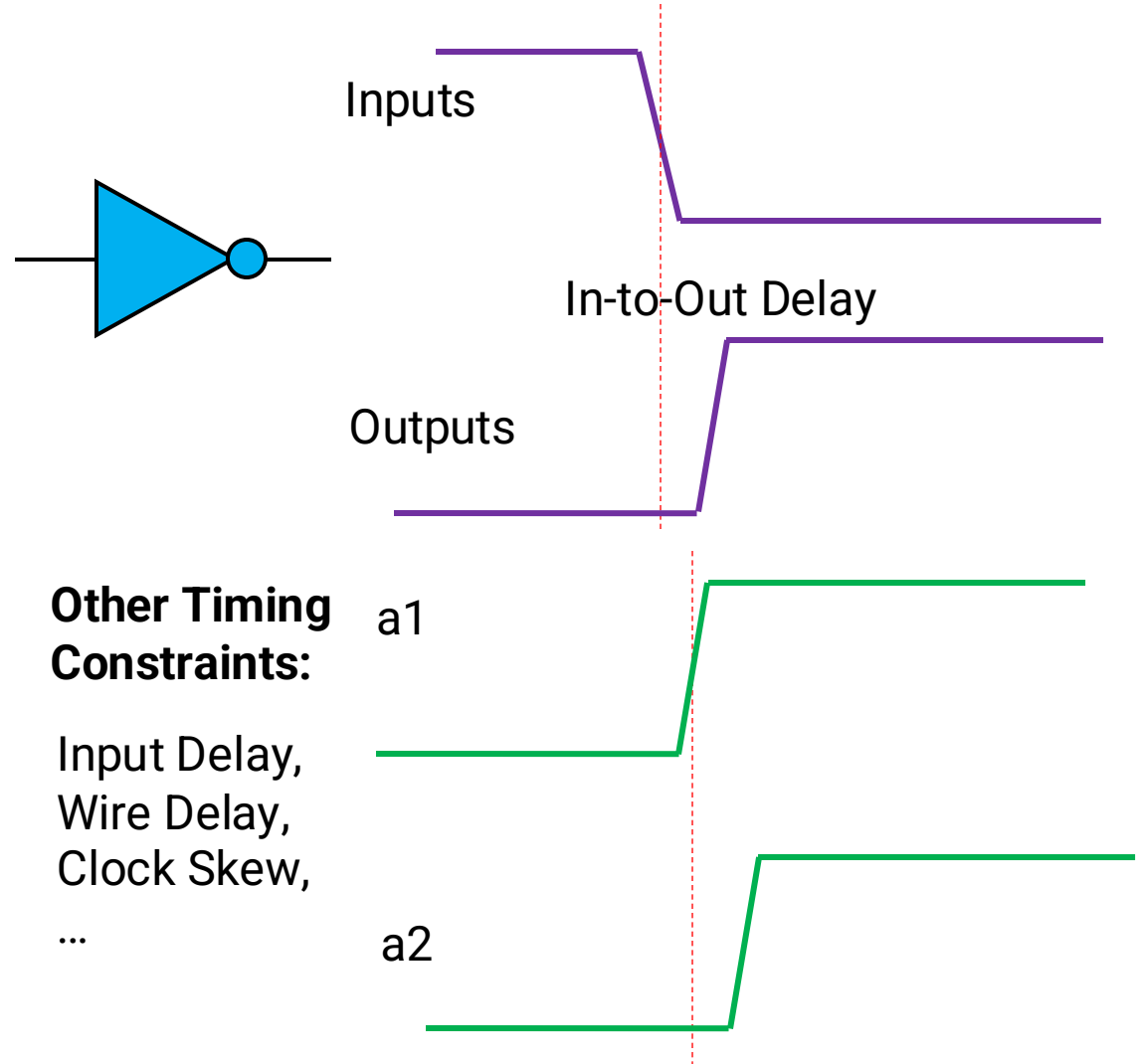
Timing Limitations



Flip-Flop (FF) Delay

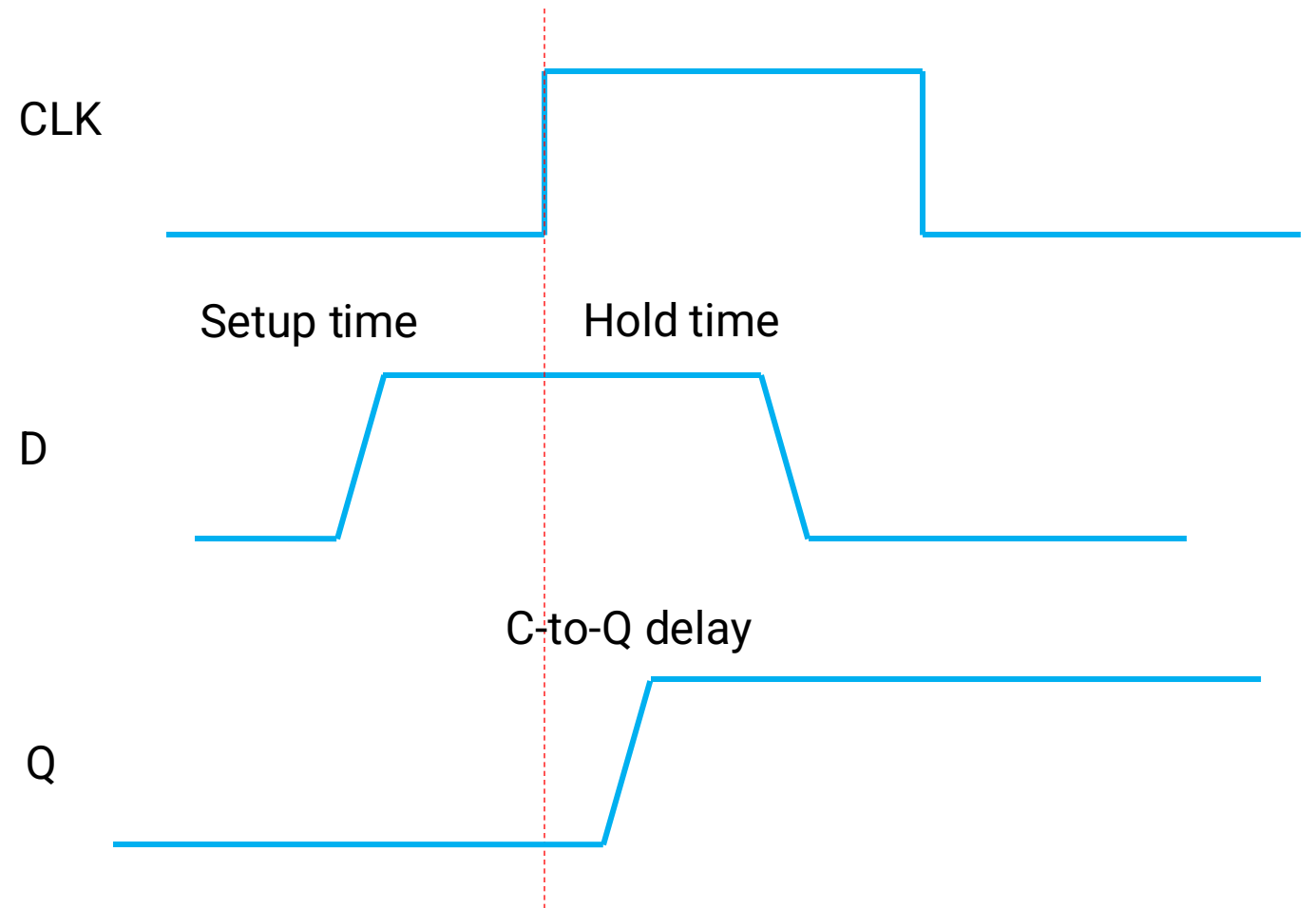
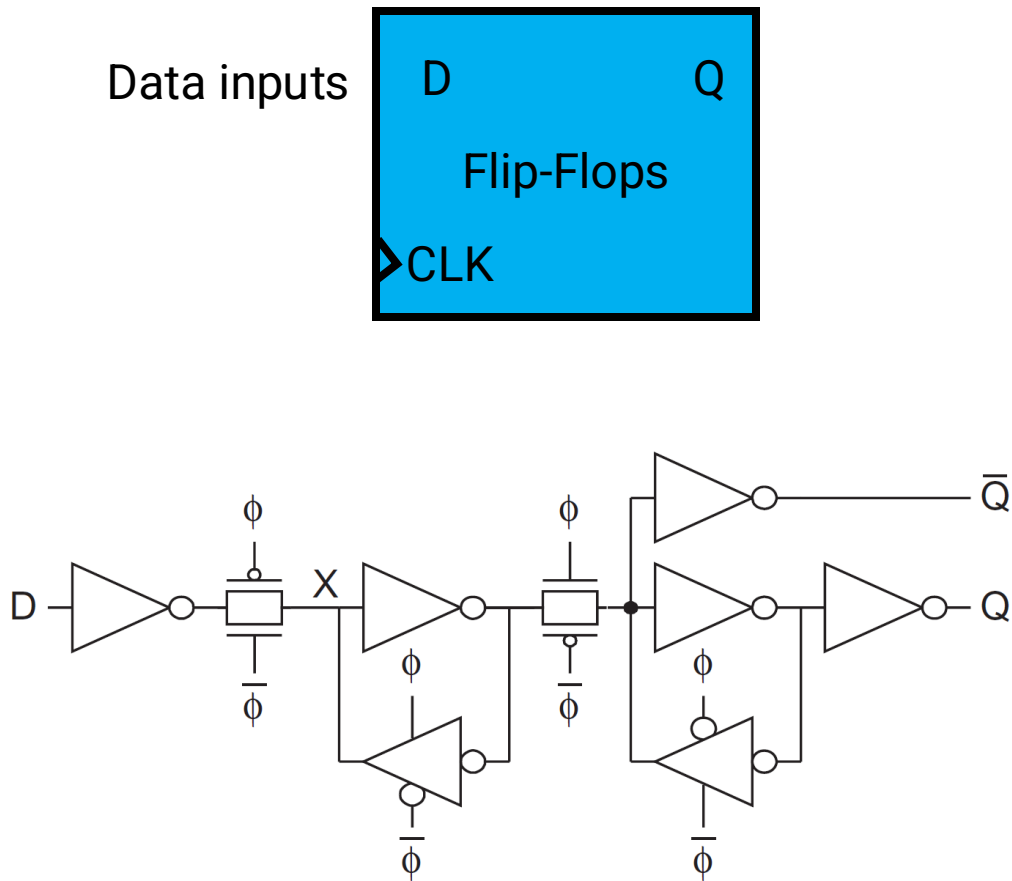


Combinational Logic (CL) Delay

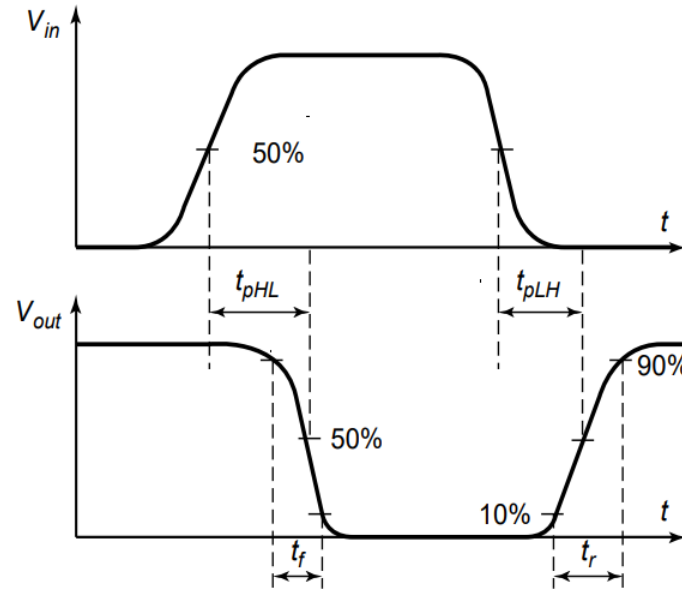
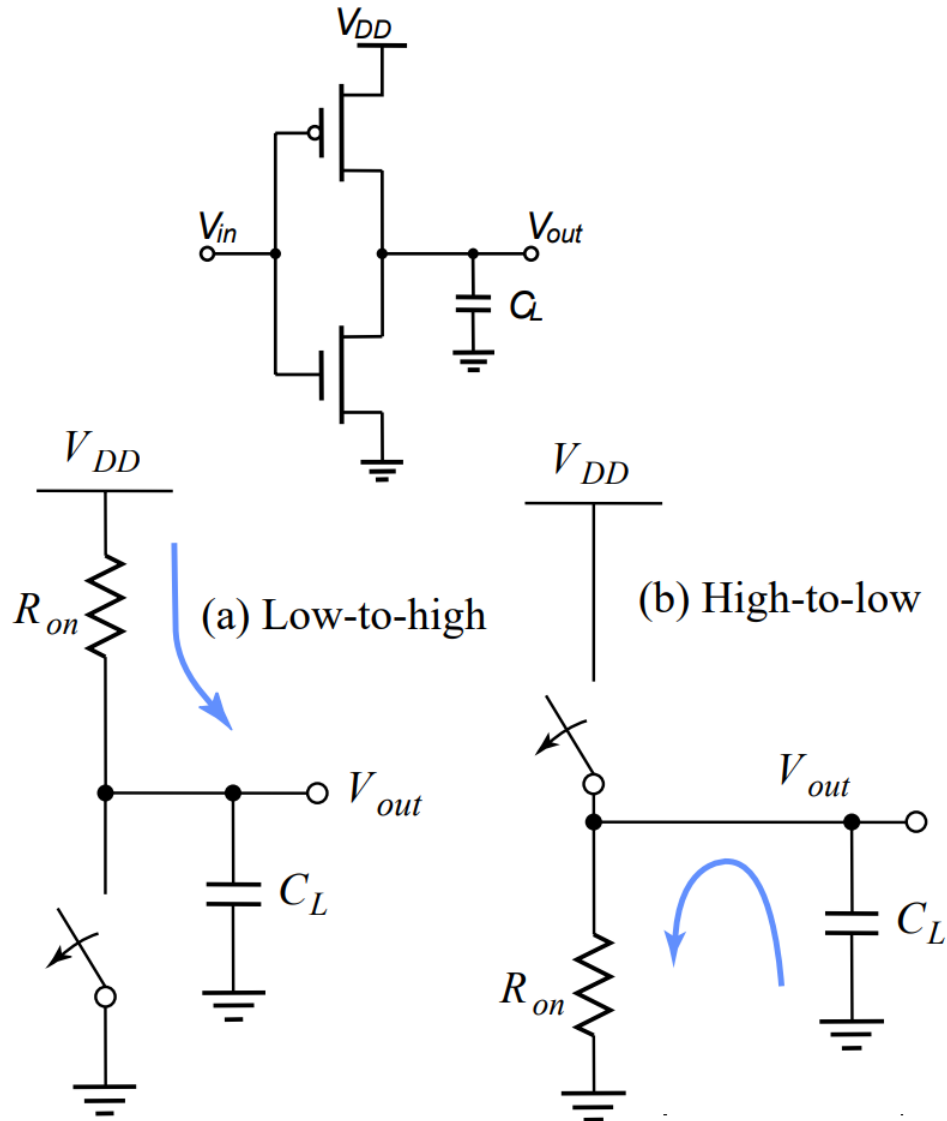


1 - FF Timing

- Flip-flops need time to a ready & adequate data inputs

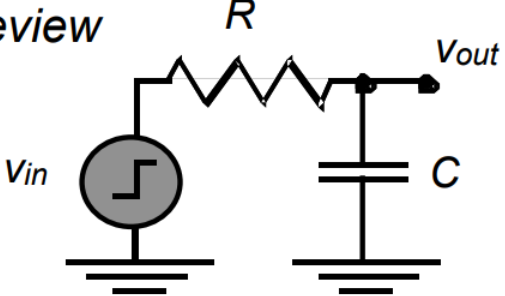


2 - Combinational Logic Delay



Application Example:
load with unit "pF"

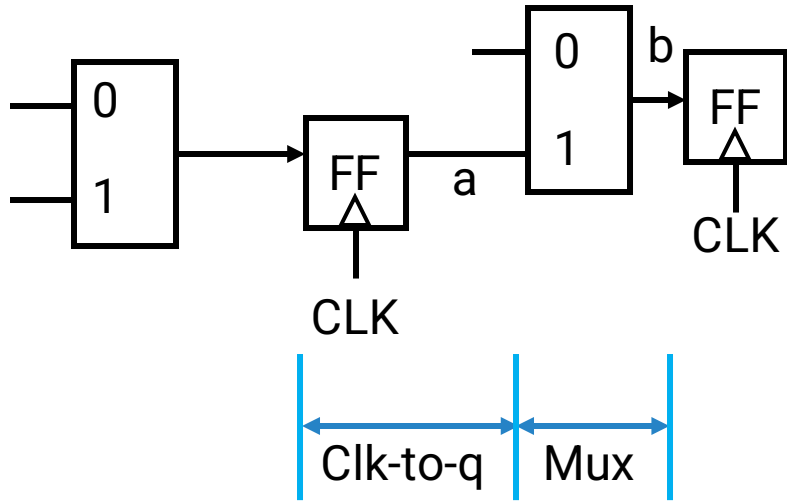
review



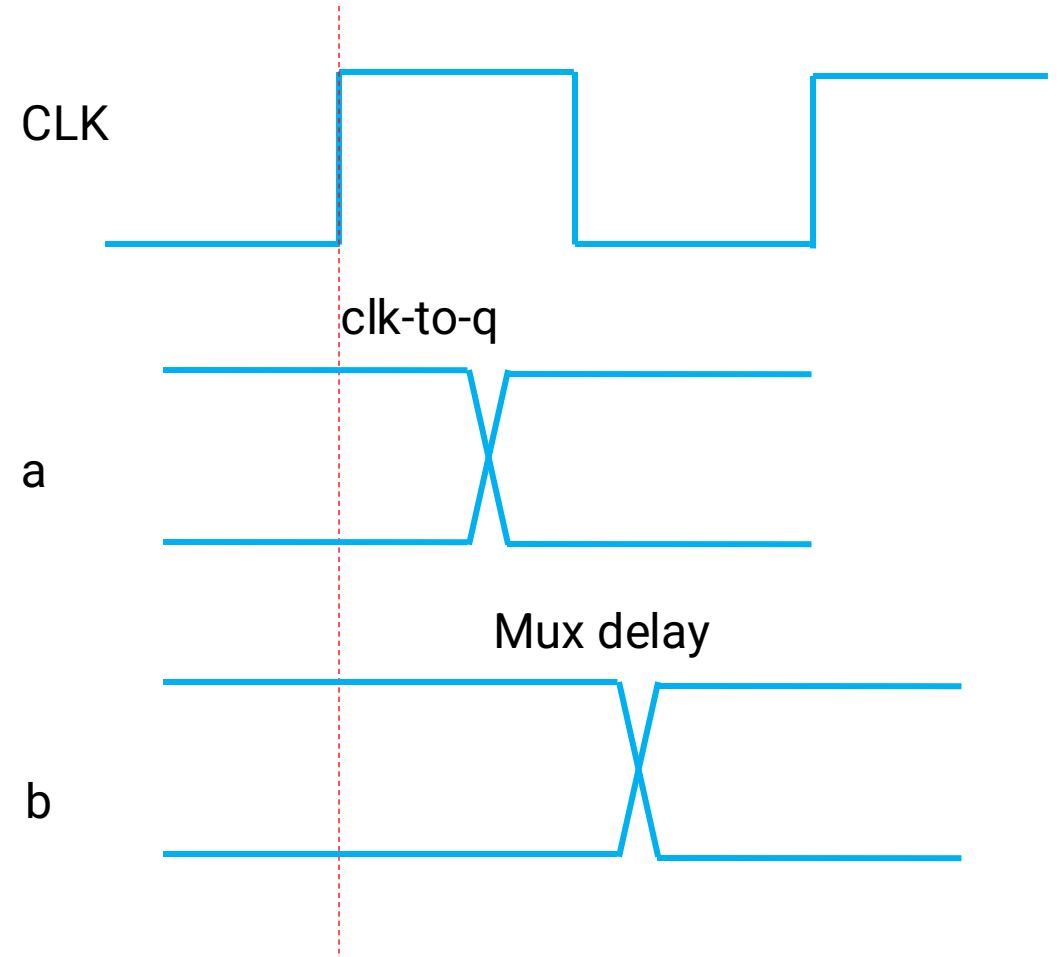
$v_{out}(t) = (1 - e^{-t/\tau}) V$

$t_p = \ln(2) \tau = 0.69 RC$

Things to Do in One Cycle



Parallel to serial converter circuit

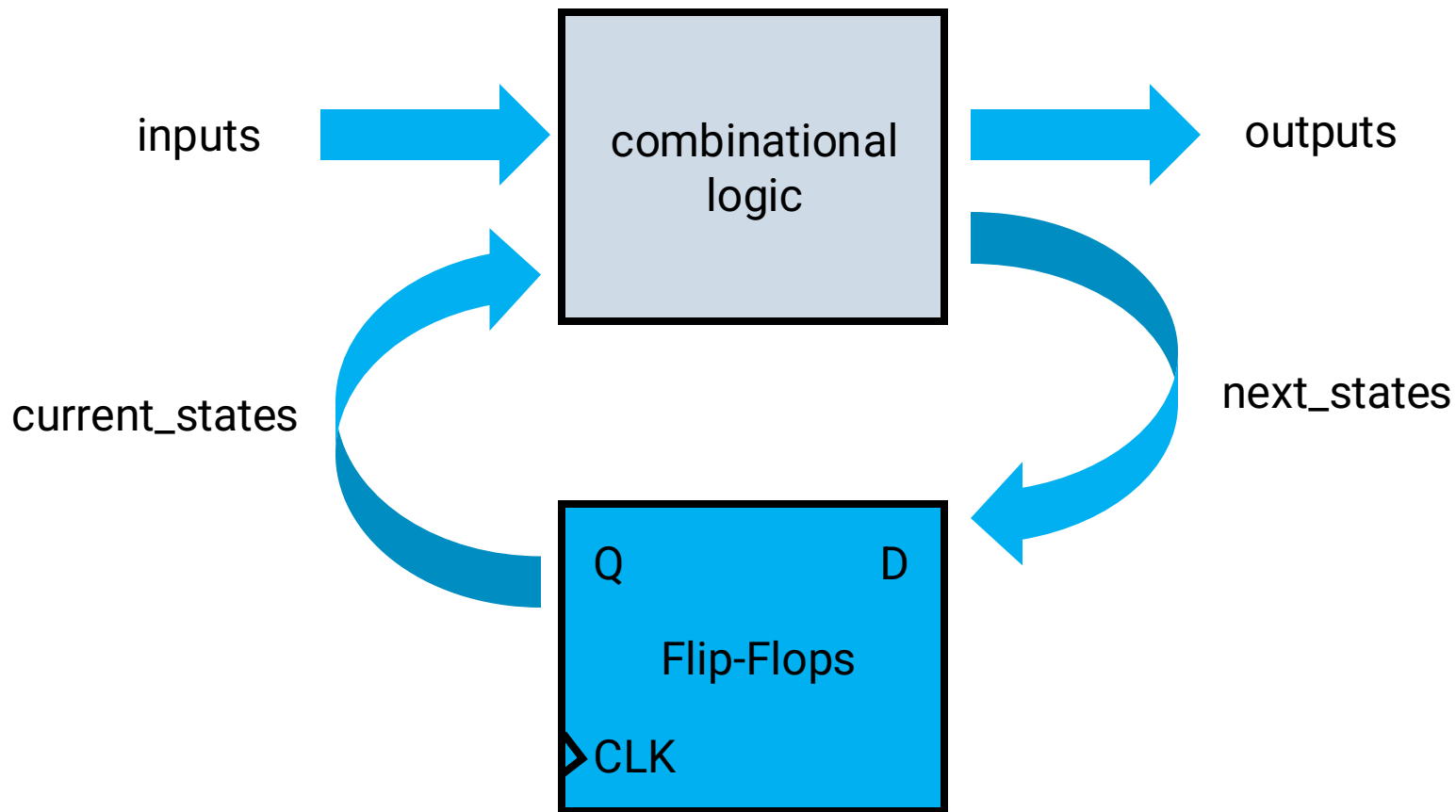


Clock Period T:

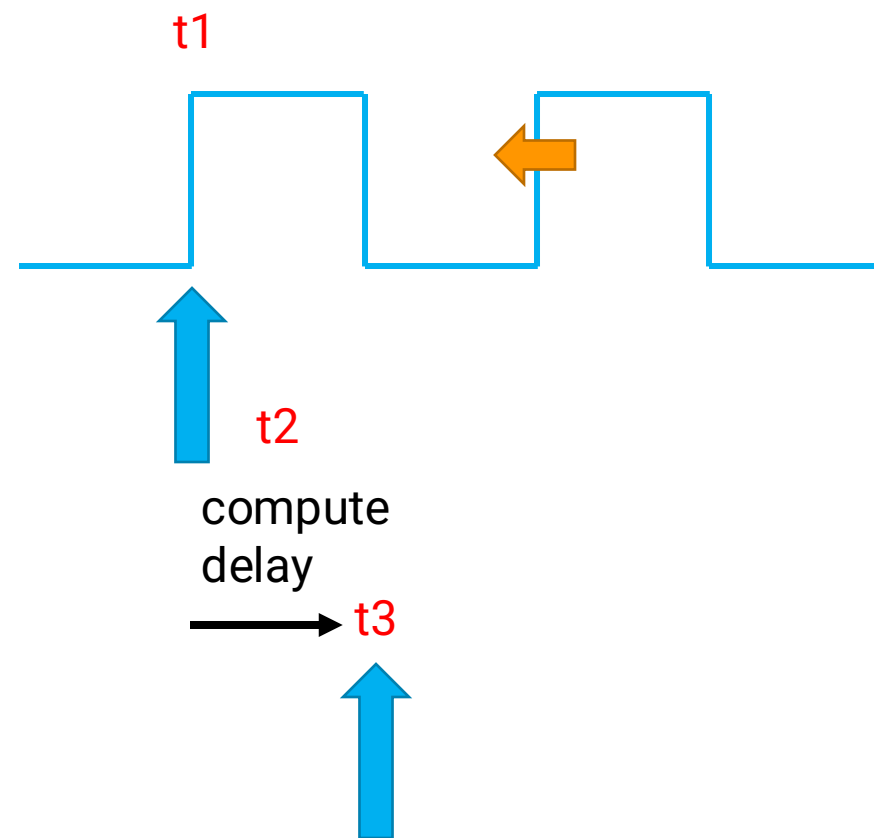
$$T > \text{Time}(\text{Clk-to-q}) + T(\text{mux}) + T(\text{setup})$$

Some Thoughts on FSM Timing

- Hardware/Circuits



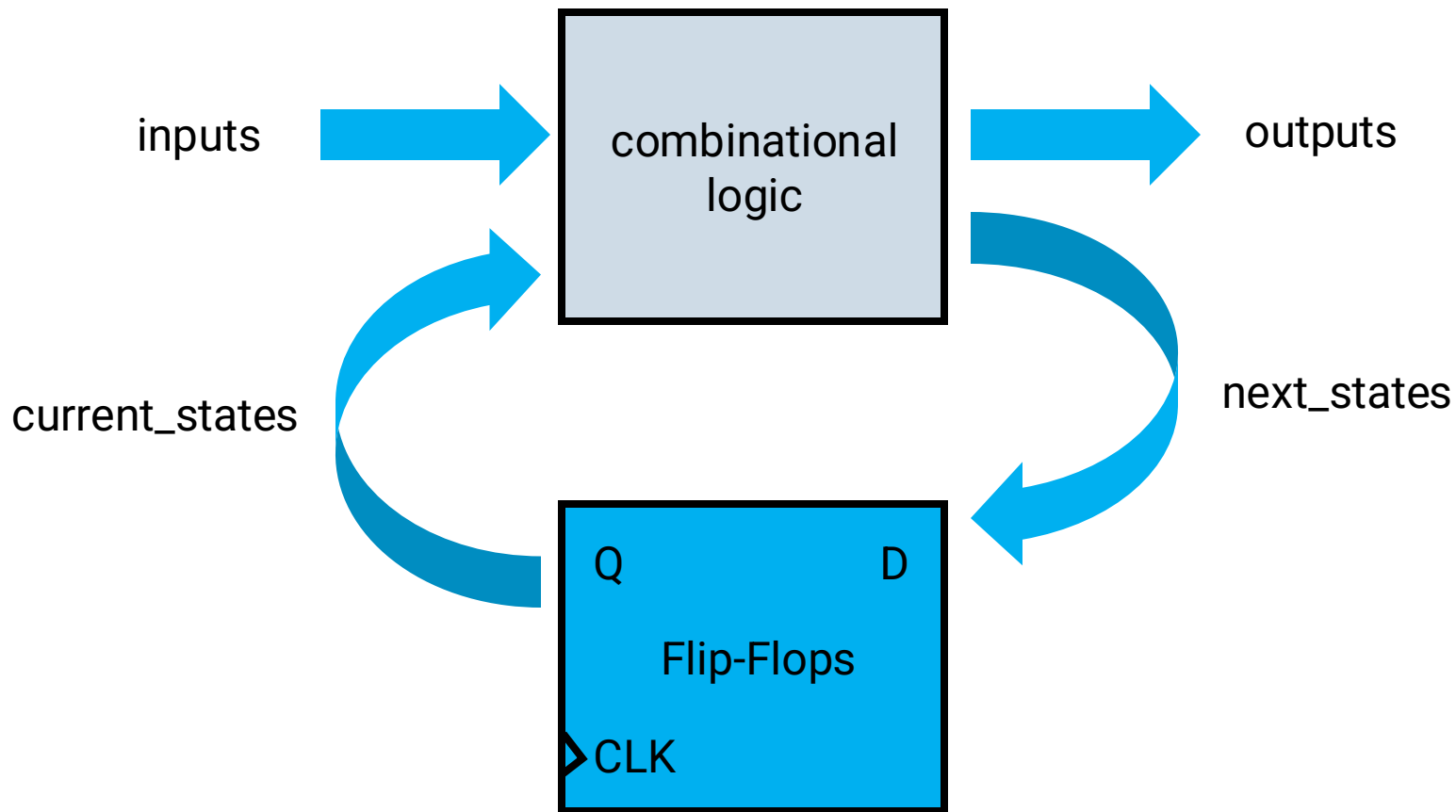
What if CLK keep shrinking?



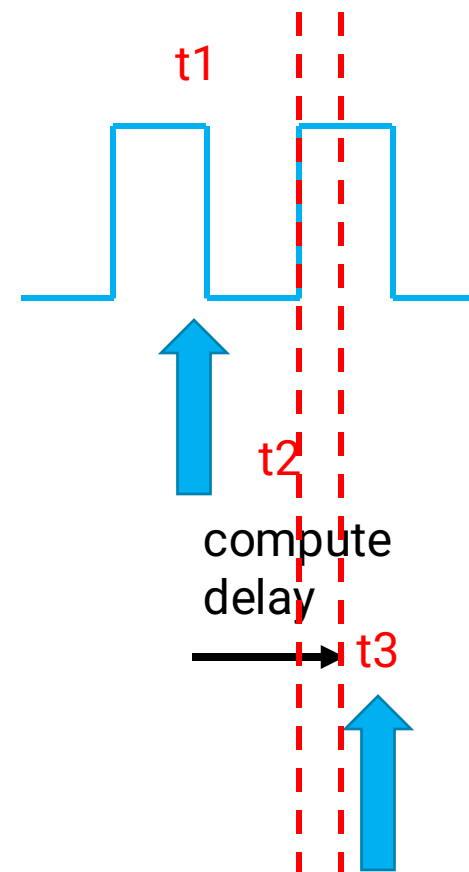
Time point: t_1, t_2, t_3

Some Thoughts on FSM Timing

- Hardware/Circuits

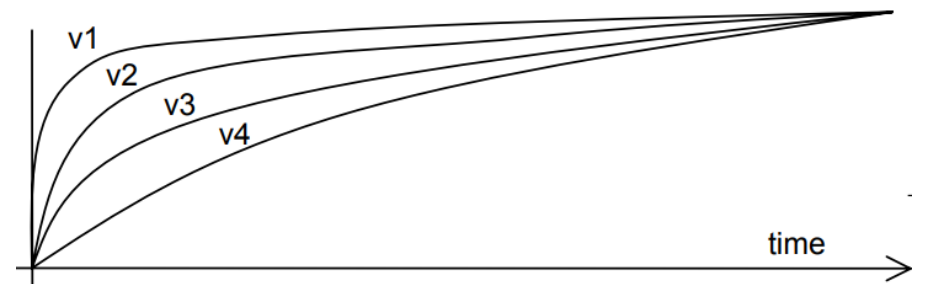
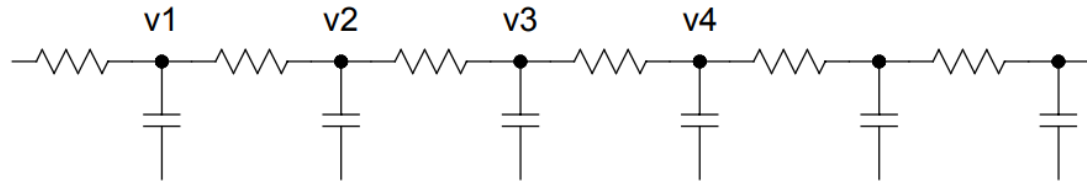
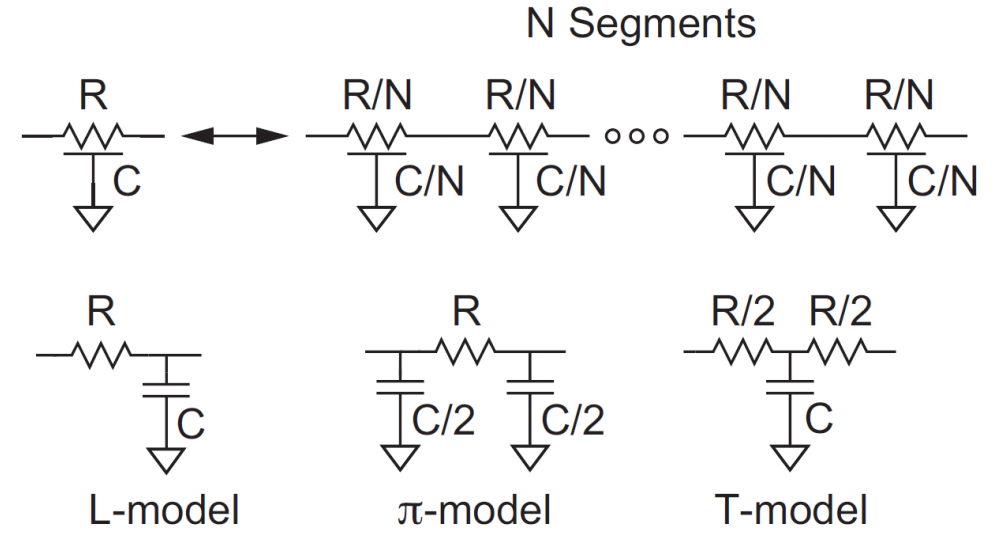
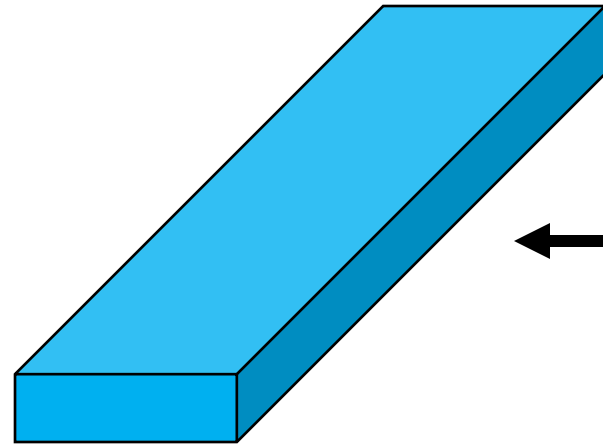
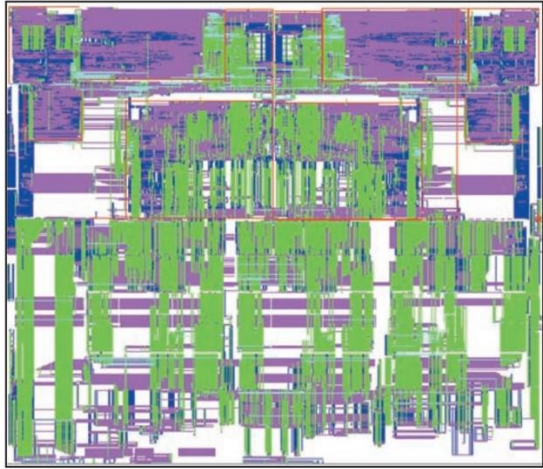


What is CLK keep shrinking?

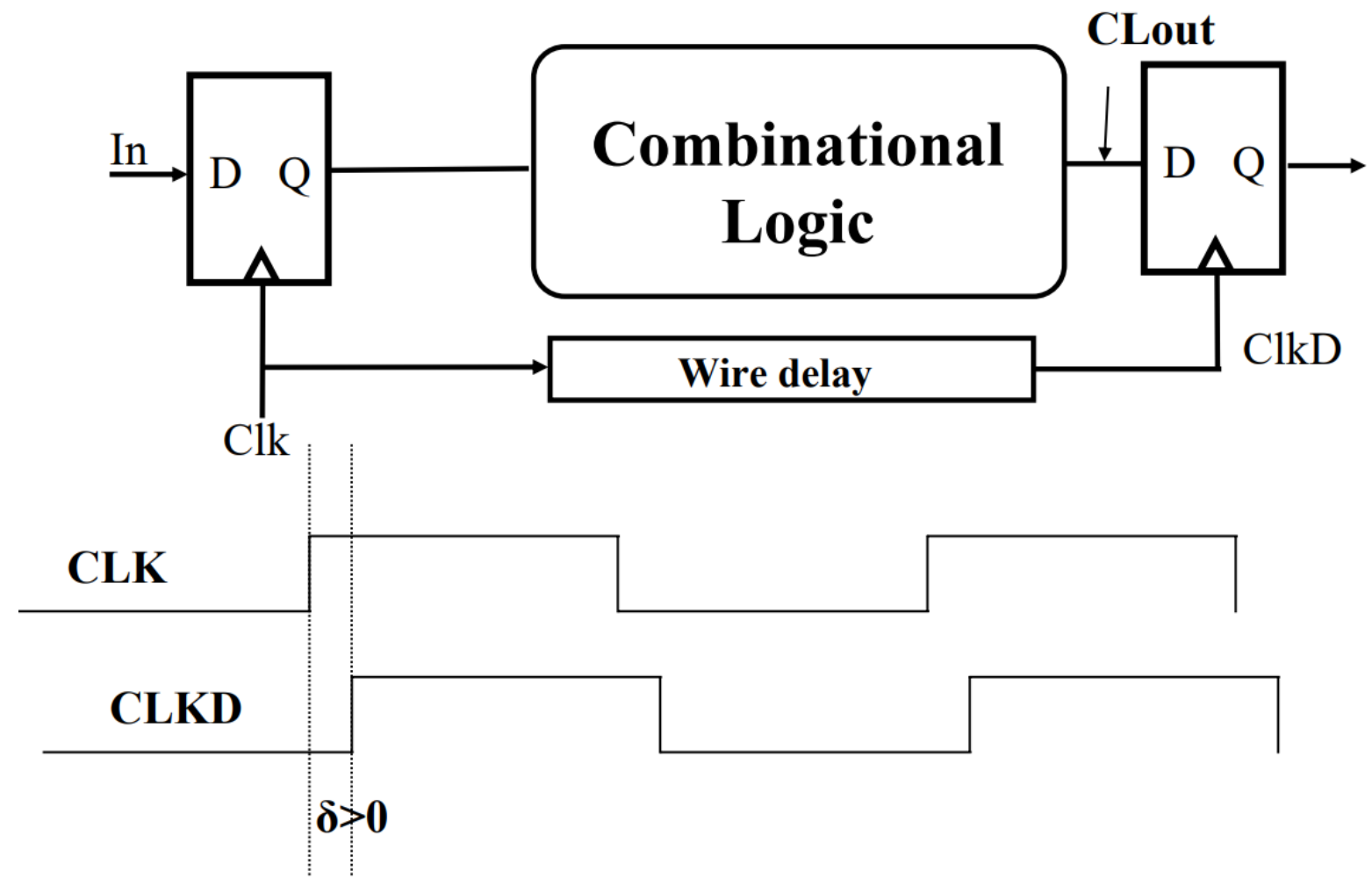


Time point: t1, t2, t3

3 - Wire Delay



4 - Clock Skew



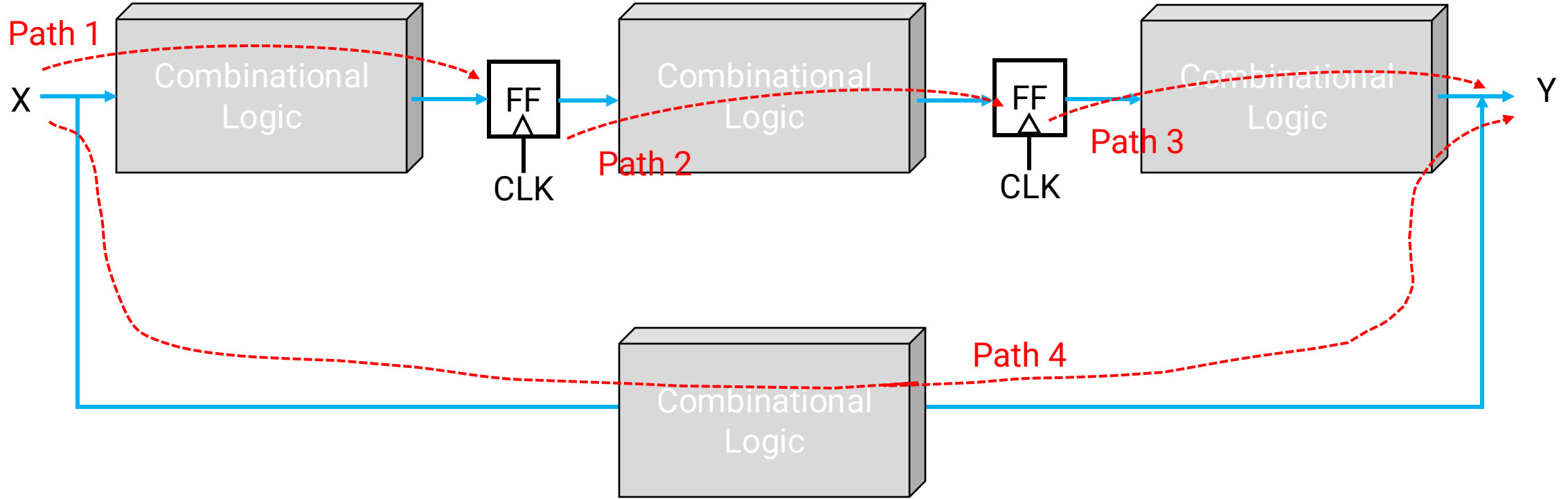
Timing Quality to Blame

	Foundry	Library Developer	CAD Tool	Designer (You!)
Gate Delay	Physical parameters	Cell topology, Transistor sizing	Cell selection	Choose Design Corner to Consider
Wire Delay	Physical parameters		Place & Route	Layout
Cell Input Capacitance	Physical parameters	Cell topology, Transistor sizing	Cell selection	
Cell Fanout			synthesis	HDL
Cell Drive Strength	Physical parameters	Transistor sizing	Cell selection	

Timing Part: Q2

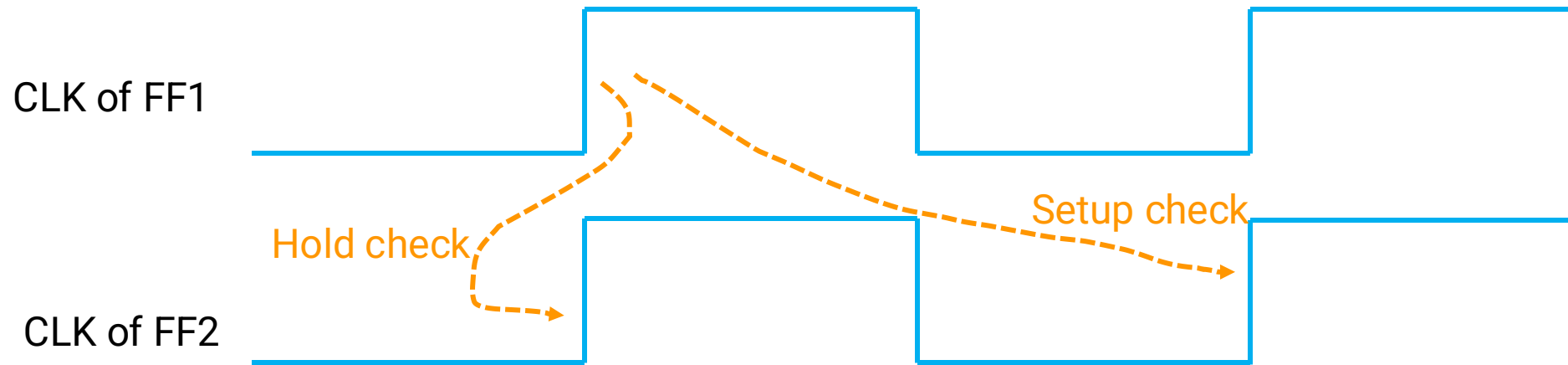
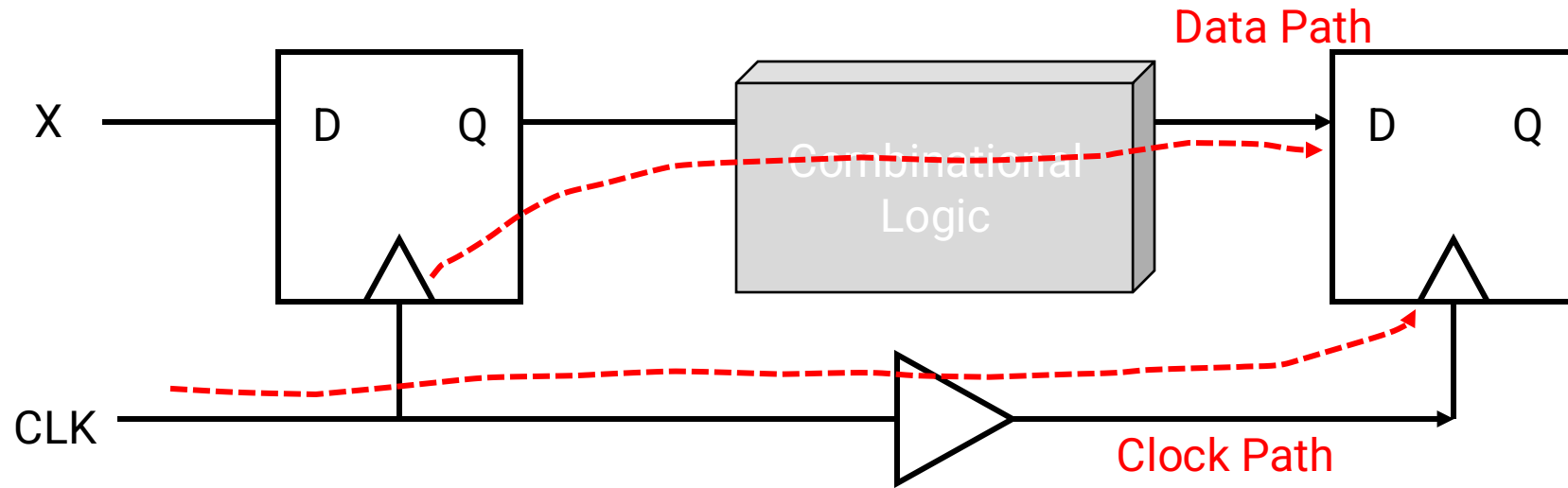
How to Analysis Timing in VLSI? ... STA

Timing Analysis for VLSI



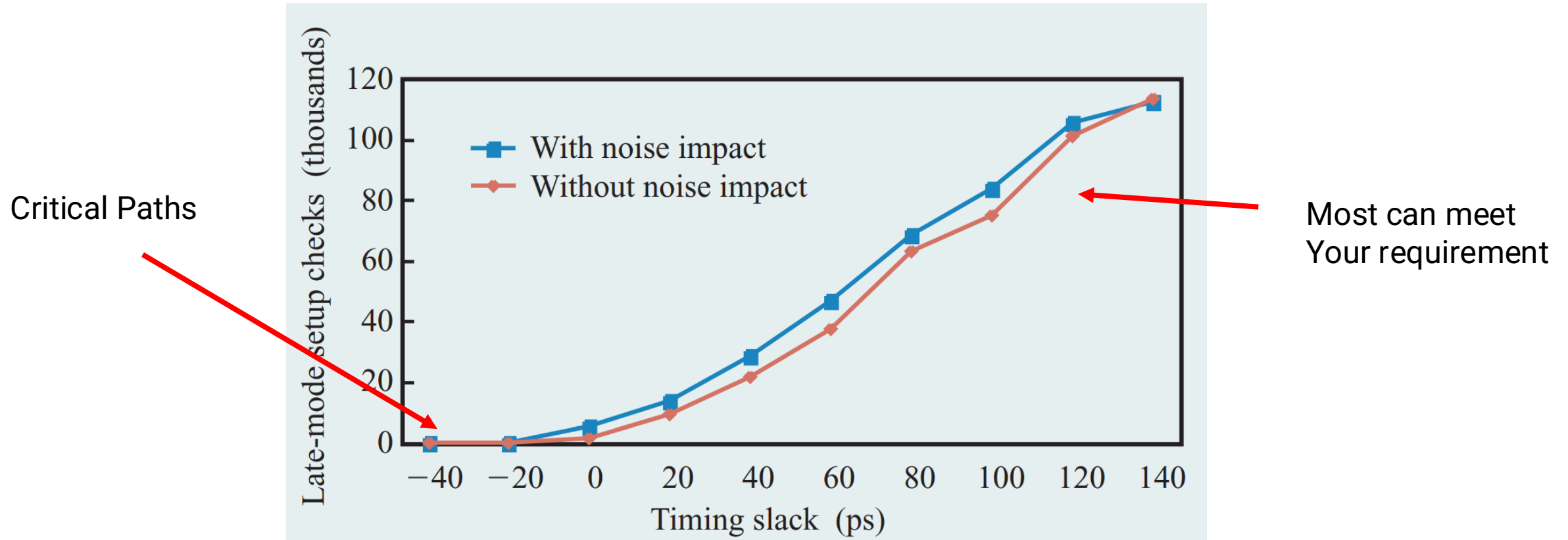
Definition of Paths

Timing Analysis for VLSI



STA: Check gate-level netlist to find the timing for all paths

Timing Analysis for VLSI



Timing Part: Q3

How to improve the design?

Technique 1 – Pipelining

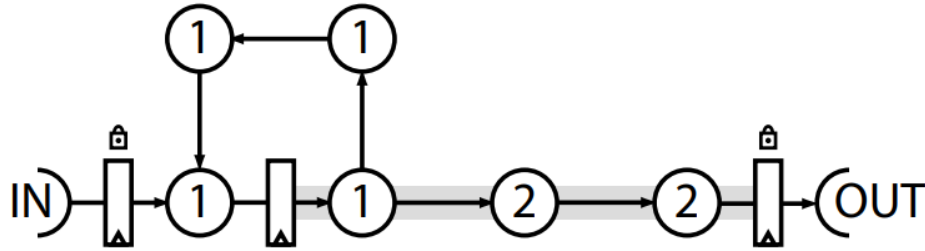


Figure 1: A small graph before retiming. The nodes represent logic delays, with the inputs and outputs passing through mandatory, fixed registers. The critical path is 5.

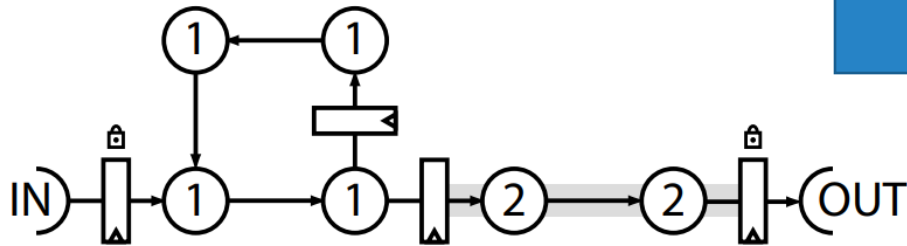


Figure 2: The example in Figure 2 after retiming. The critical path is reduced from 5 to 4.

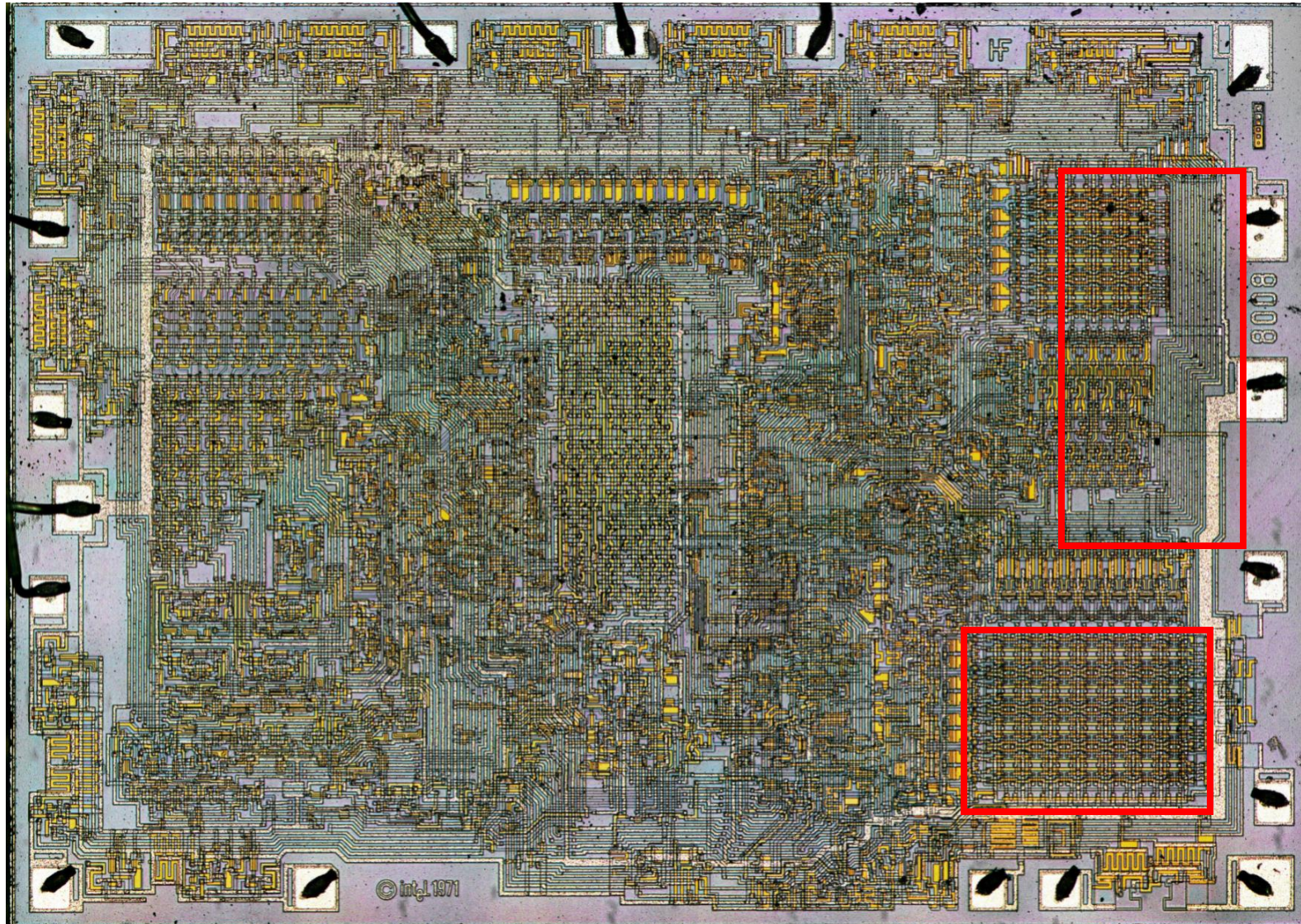
Do less things in one cycle for faster cycles

```

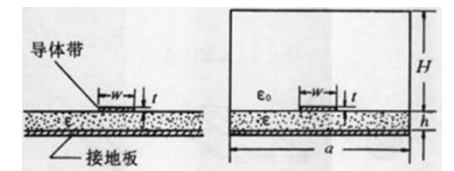
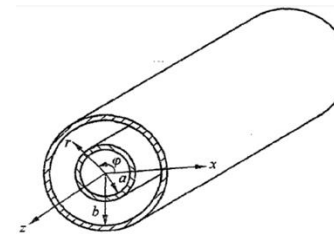
1  always @ (posedge clk or posedge reset) begin
2      if (reset)
3          state <= S0;
4      else begin
5          case (state)
6              S0:
7                  data_out = 2'b01;
8              S1: begin
9                  if(cnt==10'd1) begin
10                     //do what you want
11                 end
12                 else if (cnt<=10'd5) begin
13                     //do what you want
14                 end
15                 else if (cnt<=10'd20) begin
16                     //do what you want
17                 end
18                 else begin
19                     //do what you want
20                 end
21             endcase
22         end
23         S2:
24             data_out = 2'b11;
25         S3:
26             data_out = 2'b00;
27         default:
28             data_out = 2'b00;
29     endcase
30 end
31 end

```

Technique 2 – Floor planning



1. Make module connection natural:
 - Find good neighbors
2. Leave bus channels
3. Make bus wave guides!



Die photo of Intel first 8b processor 8008