Timing and Verification

Design of Digital Circuits 2017

Srdjan Capkun
Onur Mutlu
(Guest starring: Frank K. Gürkaynak and Aanjhan Ranganathan)

http://www.syssec.ethz.ch/education/Digitaltechnik_17

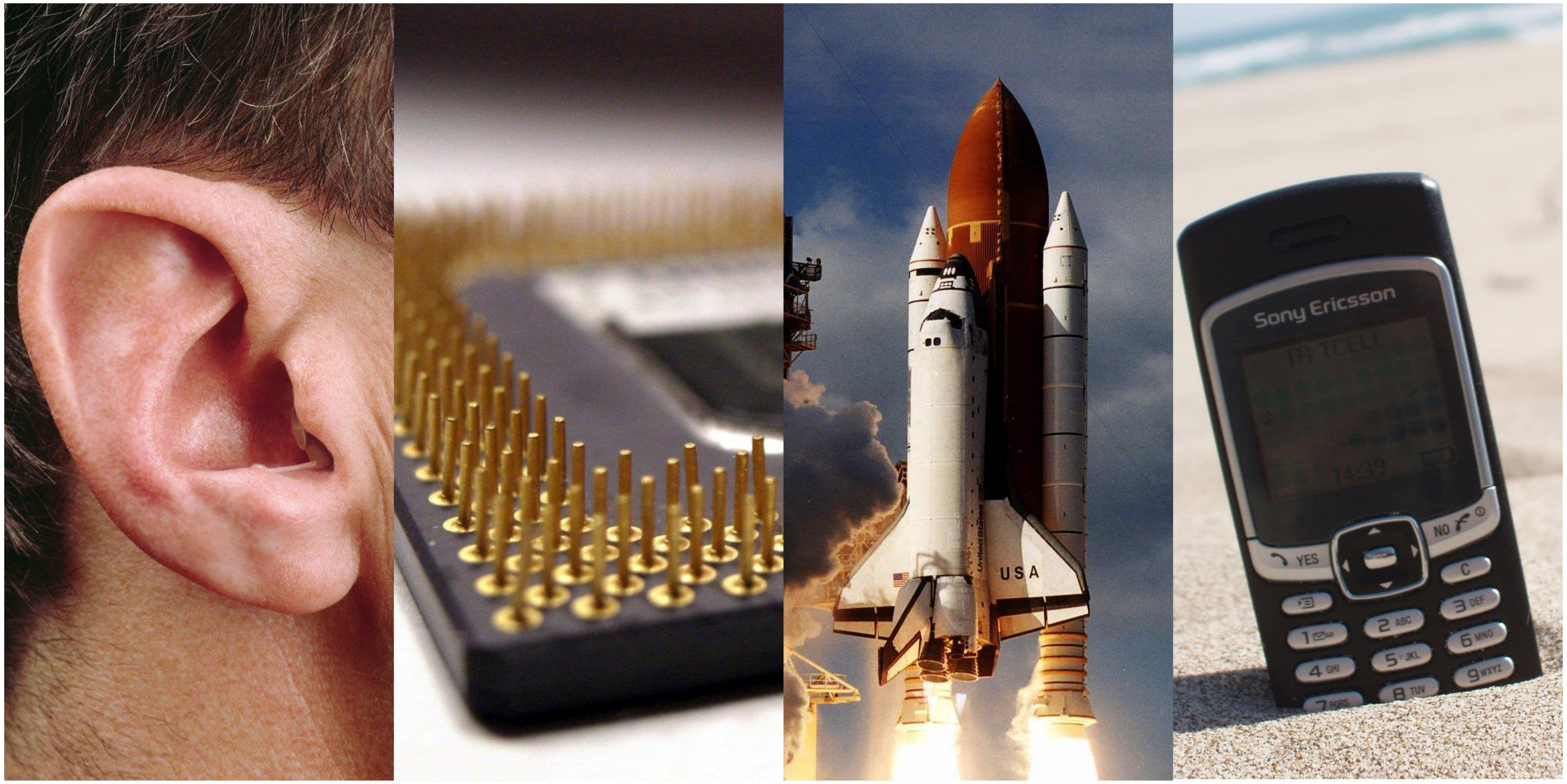
What will we learn

- **Timing in Combinational circuits**
  - Propagation and Contamination Delays
- **Timing for Sequential circuits**
  - Setup and Hold time
  - How fast can my circuit work?
- **How timing is modeled in Verilog**
- **Verification using Verilog**
  - How can we make sure the circuit works correctly
  - Designing Testbenches

The Goal Of Circuit Design Is To Optimize:

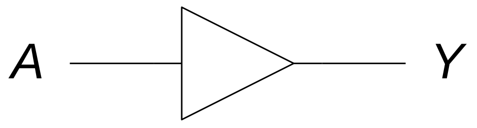
- **Area**
  - Net circuit area is proportional to the cost of the device
- **Speed / Throughput**
  - We want circuits that work faster, or do more
- **Power / Energy**
  - Mobile devices need to work with a limited power supply
  - High performance devices dissipate more than $100W/cm^2$
- **Design Time**
  - Designers are expensive
  - The competition will not wait for you

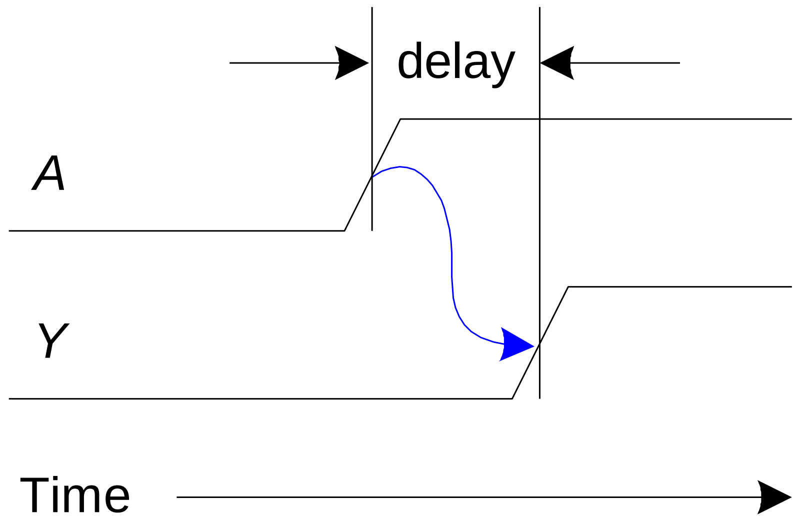
Requirements Depend On Application



Timing

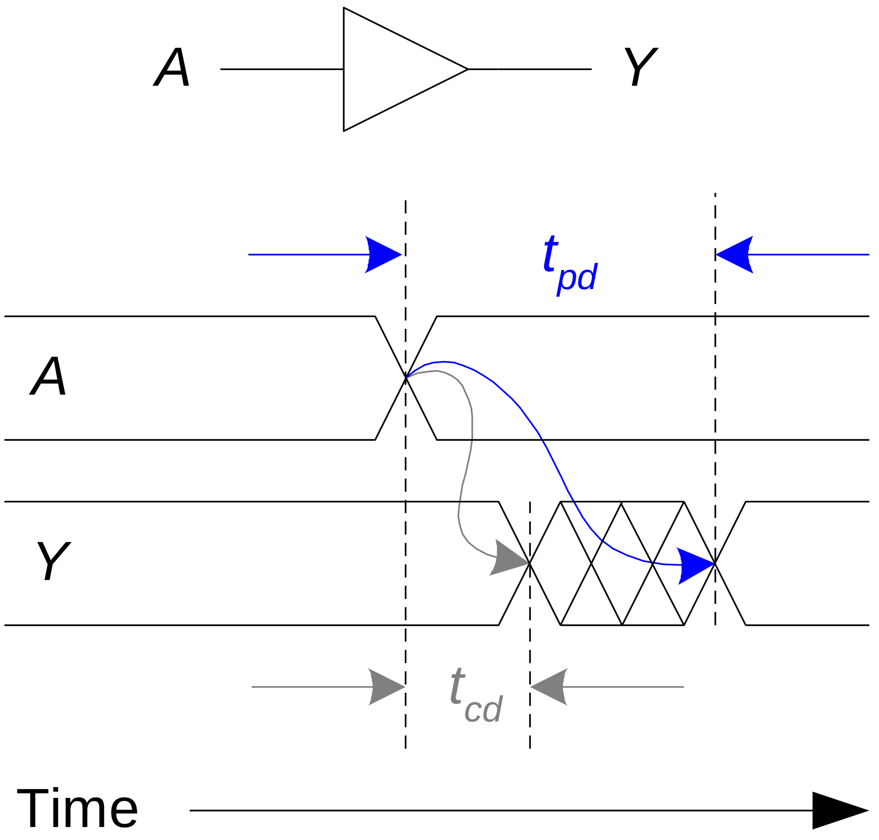
- **Until now, we investigated mainly functionality**
- **What determines how fast a circuit is and how can we make faster circuits?**





Propagation and Contamination Delay

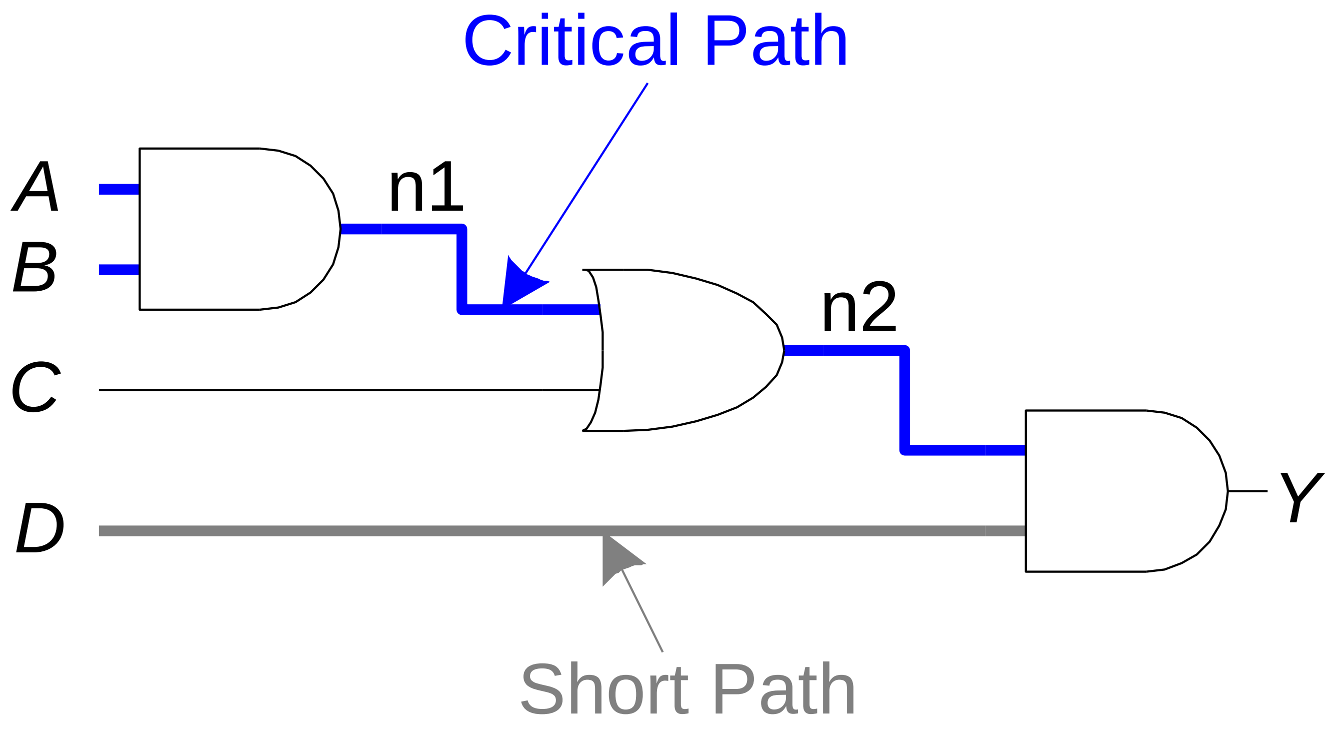
- **Propagation delay**: t_{pd} = max delay from input to output
- **Contamination delay**: t_{cd} = min delay from input to output



Propagation & Contamination Delay

- **Delay is caused by**
  - Capacitance and resistance in a circuit
  - Speed of light limitation (not as fast as you think!)
- **Reasons why t_{pd} and t_{cd} may be different:**
  - Different rising and falling delays
  - Multiple inputs and outputs, some of which are faster than others
  - Circuits slow down when hot and speed up when cold

Critical (Long) and Short Paths



- **Critical (Long) Path:** $t_{pd} = 2\ t_{pd\_AND} + t_{pd\_OR}$
- **Short Path:** $t_{cd} = t_{cd\_AND}$

Propagation times

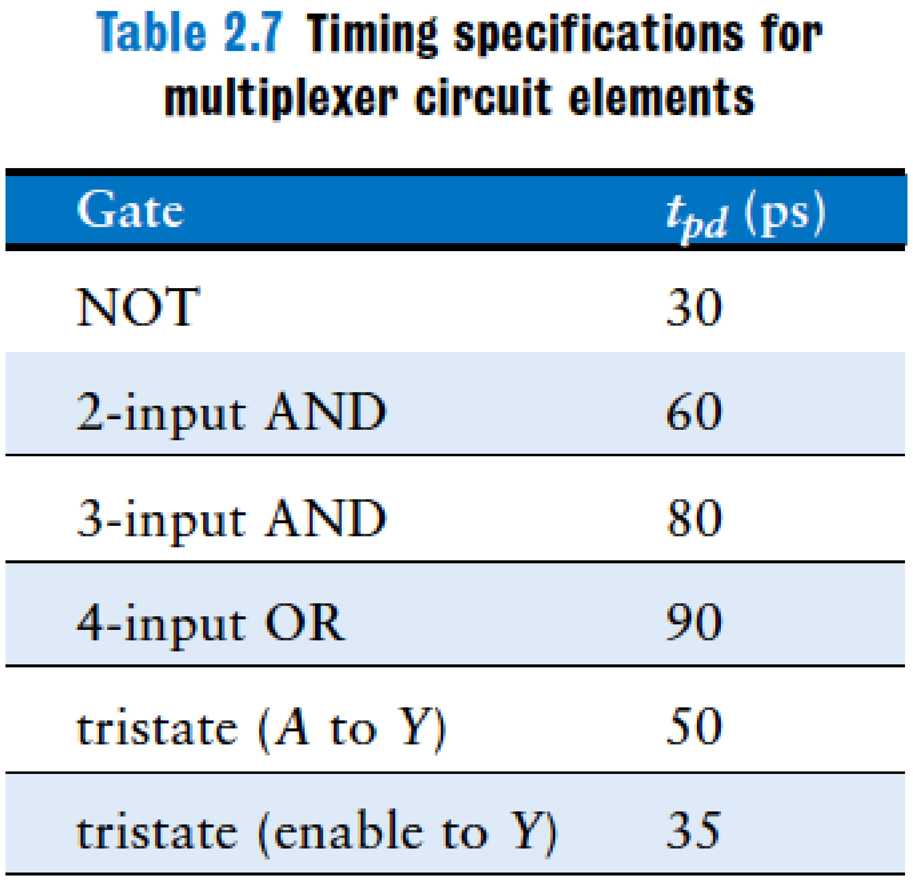
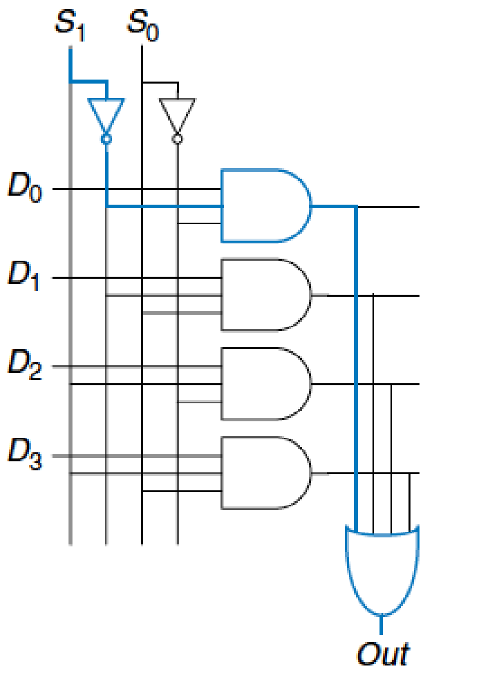
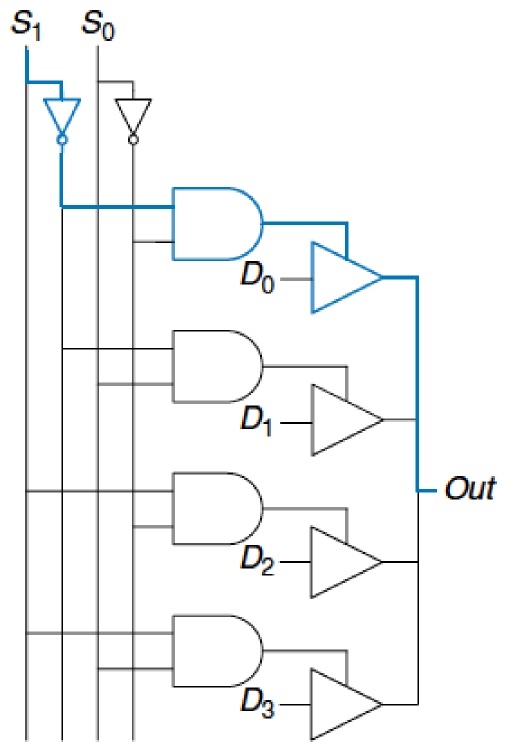


Table 2.7 Timing specifications for multiplexer circuit elements

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

Propagation times





(a)

$$\begin{aligned}t_{pd\_sy} &= t_{pd\_INV} + t_{pd\_AND3} + t_{pd\_OR4}\\ &= 30\text{ ps} + 80\text{ ps} + 90\text{ ps}\\ &= \mathbf{200\text{ ps}}\\ t_{pd\_dy} &= t_{pd\_AND3} + t_{pd\_OR4}\\ &= \mathbf{170\text{ ps}}\end{aligned}$$

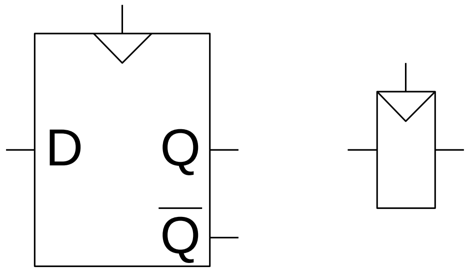
(b)

$$\begin{aligned}t_{pd\_sy} &= t_{pd\_INV} + t_{pd\_AND2} + t_{pd\_TRI\_SY}\\ &= 30\text{ ps} + 60\text{ ps} + 35\text{ ps}\\ &= \mathbf{125\text{ ps}}\\ t_{pd\_dy} &= t_{pd\_TRI\_AY}\\ &= \mathbf{50\text{ ps}}\end{aligned}$$

Figure 2.73 4:1 multiplexer propagation delays: (a) two-level logic, (b) tristate

Sequential Timing

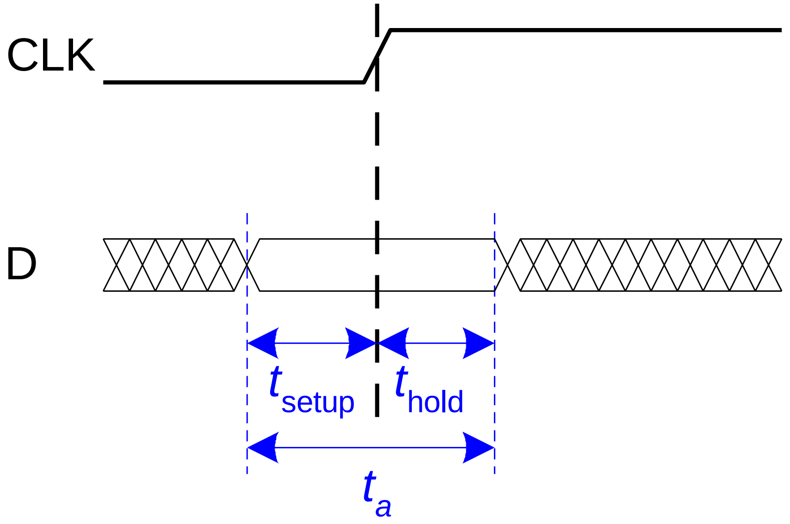
D Flip-Flop Symbols



- **Flip-flop samples D at clock edge**
- **D must be stable when it is sampled**
- **Similar to a photograph, D must be stable around the clock edge**
- **If D is changing when it is sampled, *metastability* can occur**
  - Recall that a flip-flop copies the input D to the output Q on the rising edge of the clock. This process is called sampling D on the clock edge. If D is stable at either 0 or 1 when the clock rises, this behavior is clearly defined. But what happens if D is changing at the same time the clock rises?

Input Timing Constraints

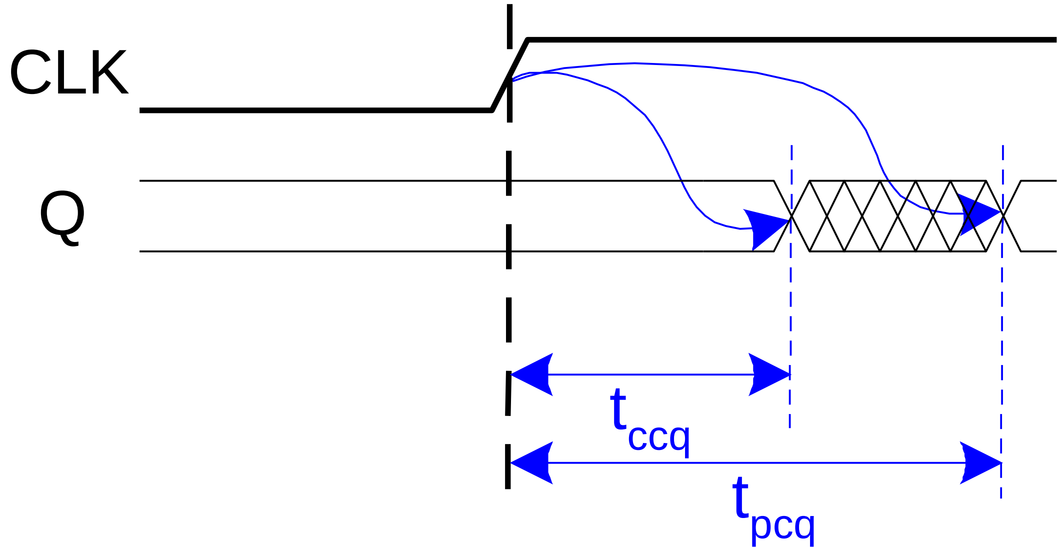
- ***Setup time***: t_{setup} = time before the clock edge that data must be stable (i.e. not changing)
- ***Hold time***: t_{hold} = time after the clock edge that data must be stable
- ***Aperture time***: t_a = time around clock edge that data must be stable ($t_a = t_{setup} + t_{hold}$)



Output Timing Constraints

- ***Propagation delay***: t_{pcq} = time after clock edge that the output Q is guaranteed to be stable (i.e., to stop changing)

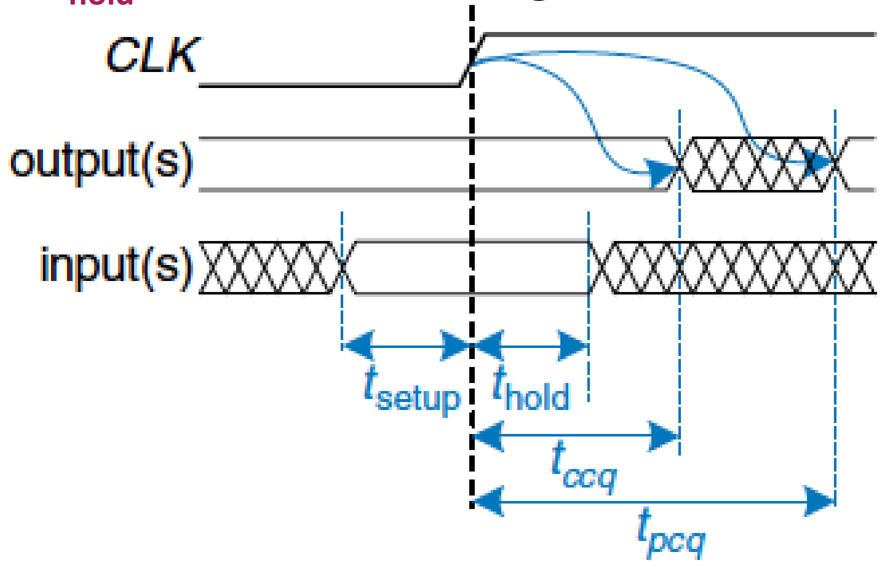
- ***Contamination delay***: t_{ccq} = time after clock edge that Q might be unstable (i.e., start changing)



Dynamic Discipline

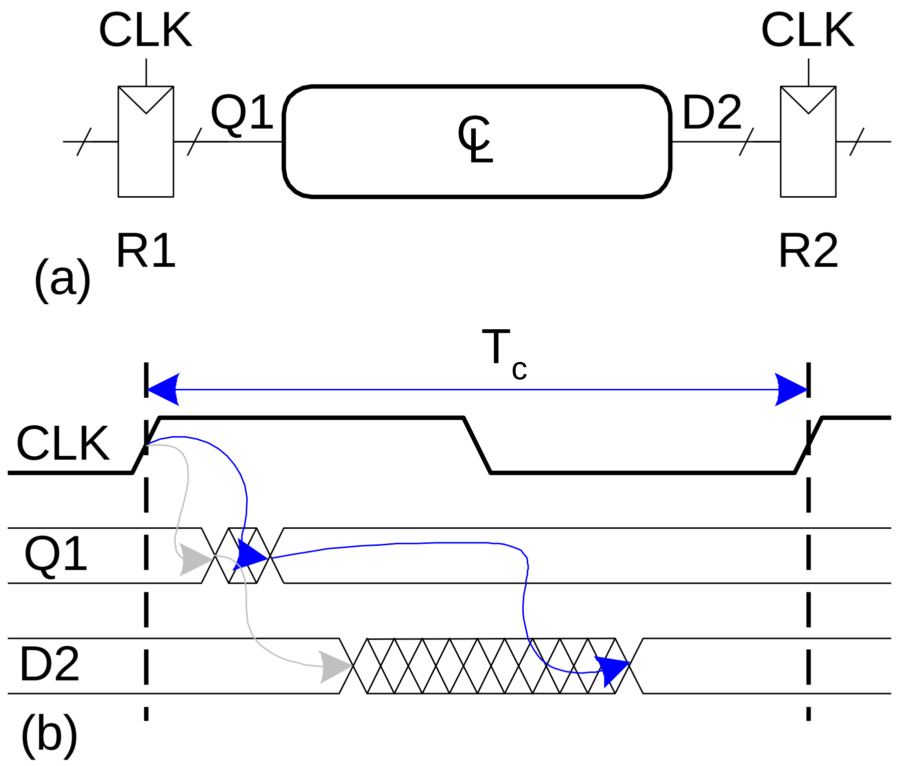
- **The input to a synchronous sequential circuit must be stable during the aperture (setup and hold) time around the clock edge.**

- **Specifically, the input must be stable**
  - at least t_{setup} **before** the clock edge
  - at least until t_{hold} **after** the clock edge



Dynamic Discipline

- The delay between registers has a minimum and maximum delay, dependent on the delays of the circuit elements

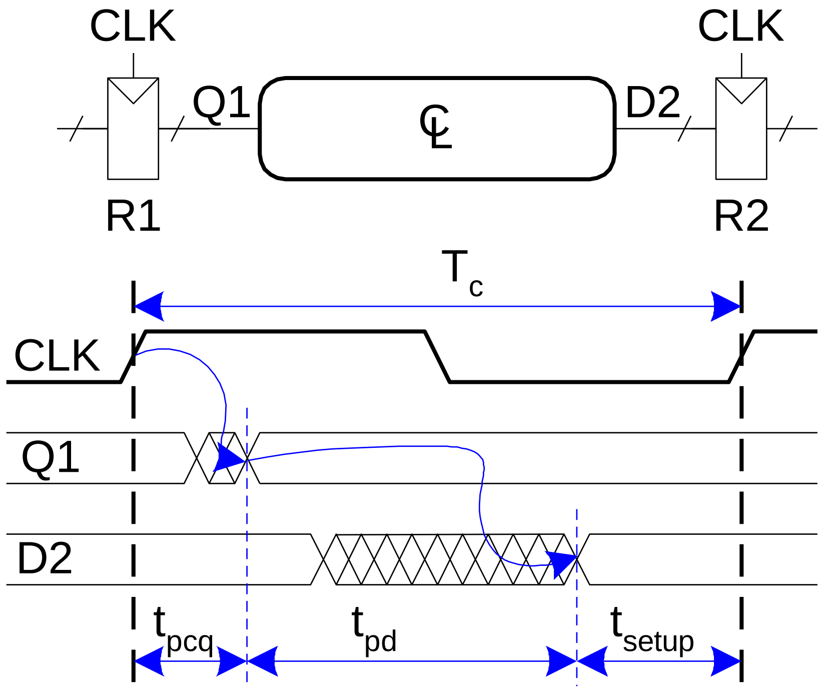


Setup Time Constraint

- **The clock period or cycle time, T_c, is the time between rising edges of a repetitive clock signal. Its reciprocal, $f_c=1/T_c$, is the clock frequency.**
- **All else being the same, increasing the clock frequency increases the work that a digital system can accomplish per unit time.**
- **Frequency is measured in units of Hertz (Hz), or cycles per second:**
  - 1 megahertz (MHz) 10^6 Hz
  - 1 gigahertz (GHz) 10^9 Hz.

Setup Time Constraint

- The setup time constraint depends on the maximum delay from register R1 through the combinational logic.
- The input to register R2 must be stable at least t_{setup} before the clock edge.

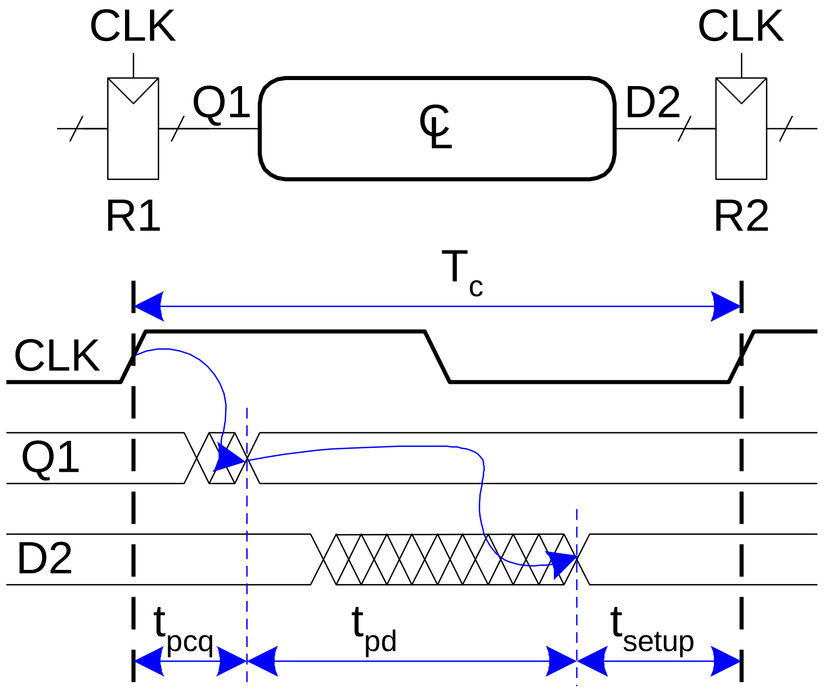


T_c >=

Setup Time Constraint

- The setup time constraint depends on the maximum delay from register R1 through the combinational logic.

- The input to register R2 must be stable at least t_{setup} before the clock edge.

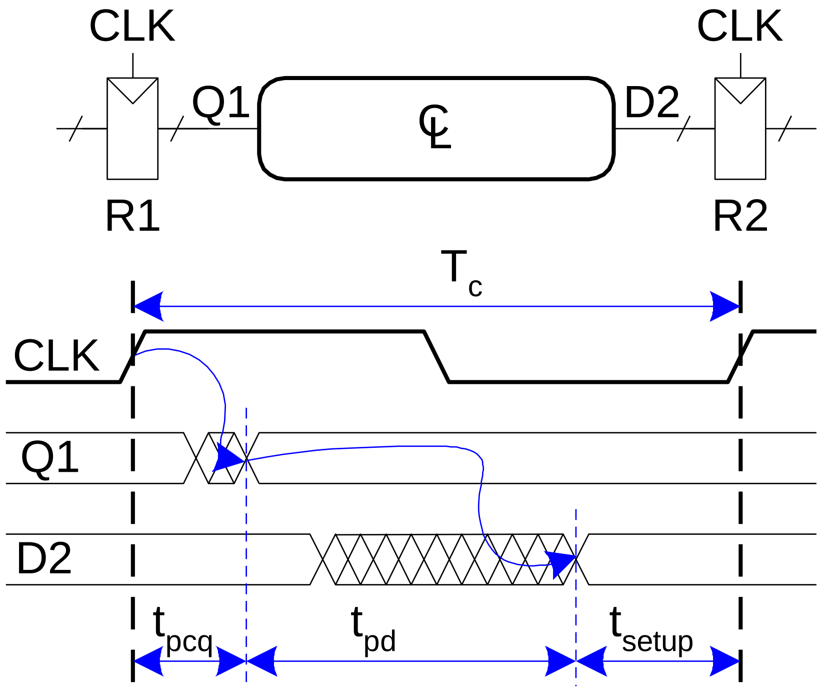


$T_c >= t_{pcq} + t_{pd} + t_{setup}$

$t_{pd} <=$

Setup Time Constraint

- **The setup time constraint depends on the maximum delay from register R1 through the combinational logic.**
- **The input to register R2 must be stable at least t_{setup} before the clock edge.**



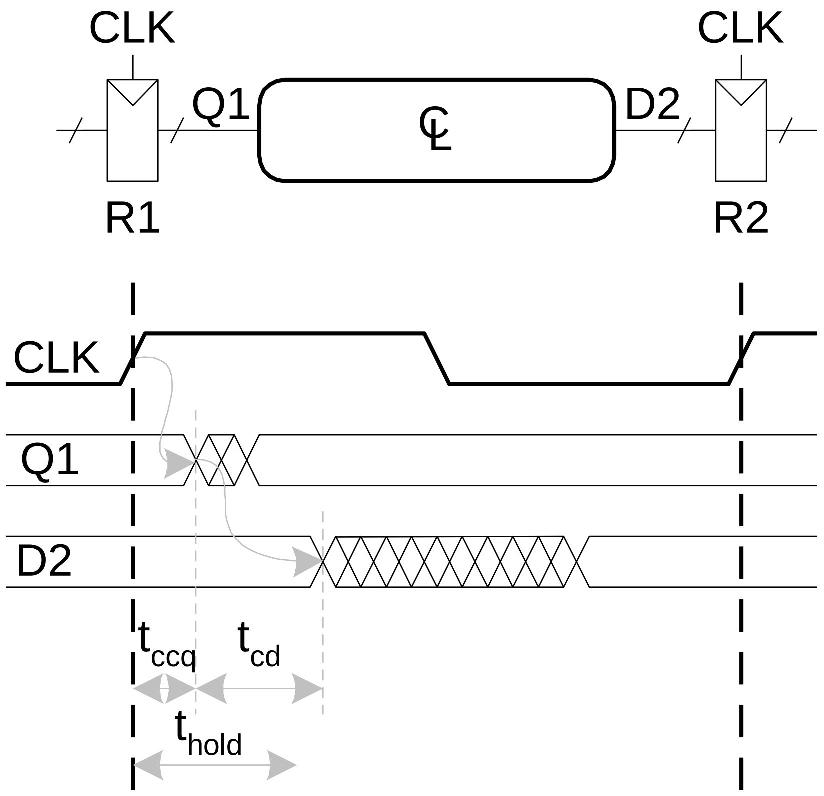
$$T_c \geq t_{pcq} + t_{pd} + t_{setup}$$

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

Hold Time Constraint

- The hold time constraint depends on the minimum delay from register R1 through the combinational logic.

- The input to register R2 must be stable for at least t_{hold} after the clock edge.

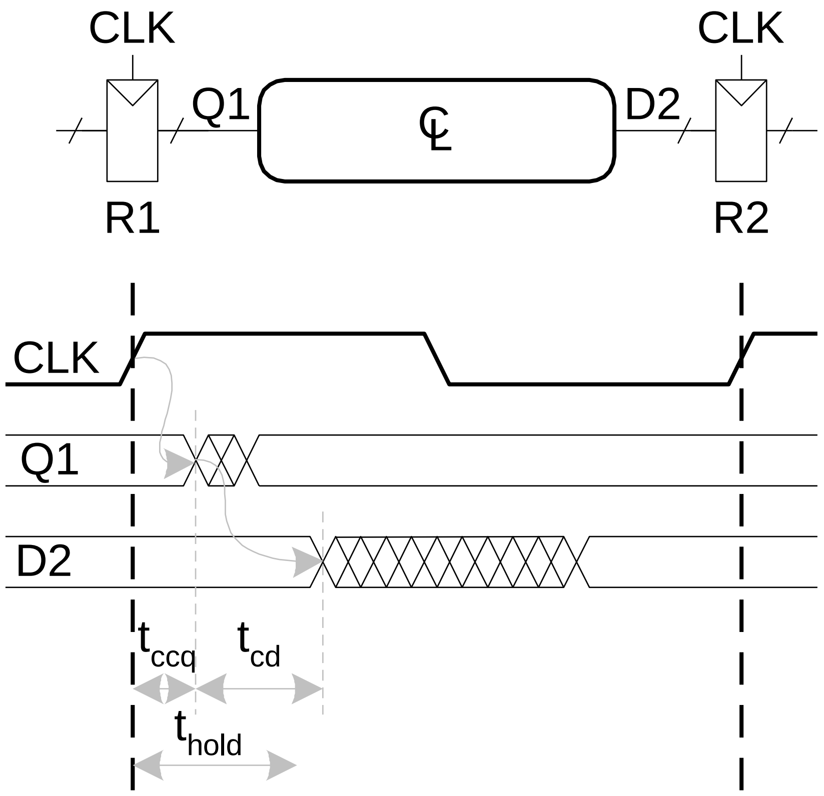


t_{hold} <

Hold Time Constraint

- The hold time constraint depends on the minimum delay from register R1 through the combinational logic.

- The input to register R2 must be stable for at least t_{hold} after the clock edge.

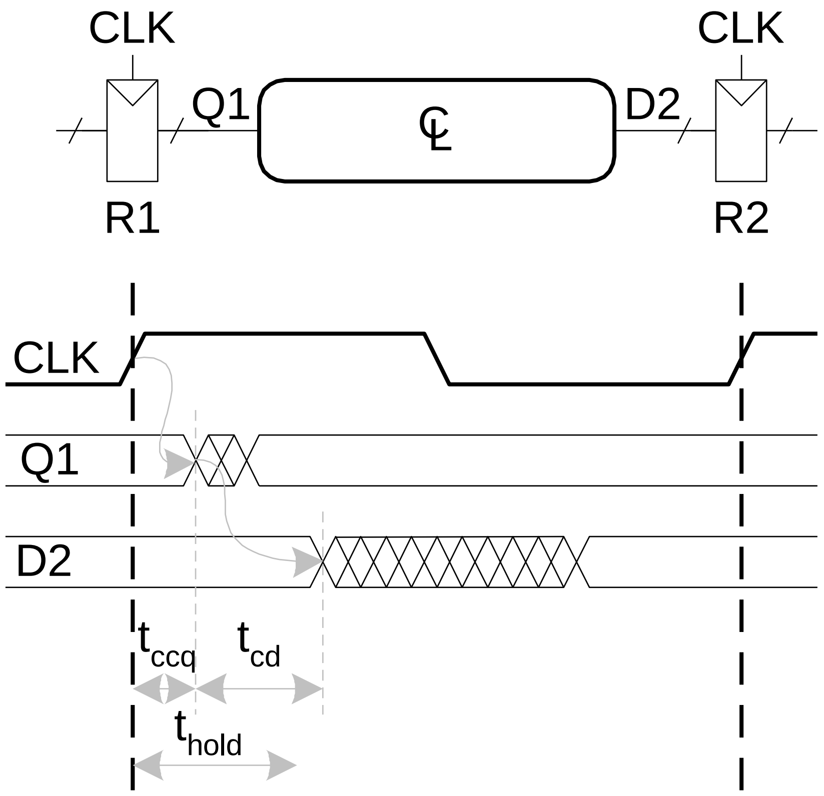


$t_{hold} < t_{ccq} + t_{cd}$

$t_{cd} >$

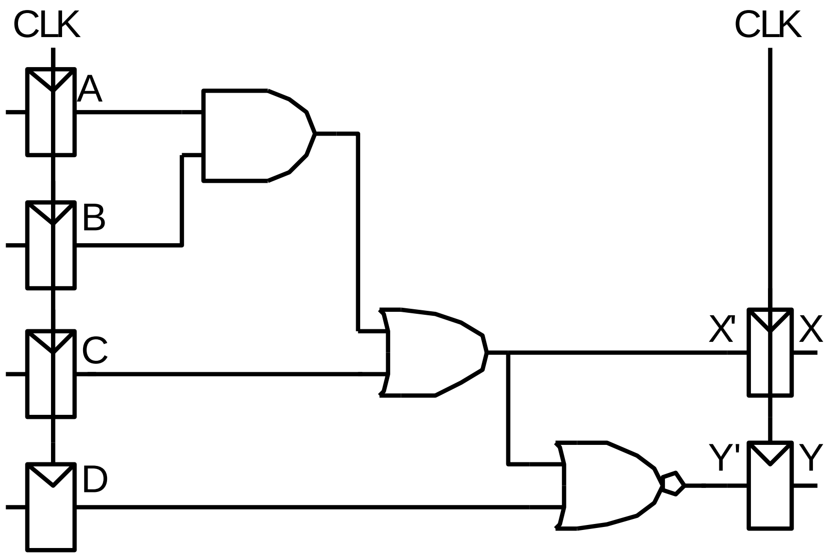
Hold Time Constraint

- The hold time constraint depends on the minimum delay from register R1 through the combinational logic.
- The input to register R2 must be stable for at least t_{hold} after the clock edge.



$$t_{hold} < t_{ccq} + t_{cd}$$
$$t_{cd} > t_{hold} - t_{ccq}$$

Timing Analysis



Timing Characteristics

t_{ccq}	= 30 ps
t_{pcq}	= 50 ps
t_{setup}	= 60 ps
t_{hold}	= 70 ps

per gate

t_{pd}	= 35 ps
t_{cd}	= 25 ps

t_{pd} =

t_{cd} =

Setup time constraint:

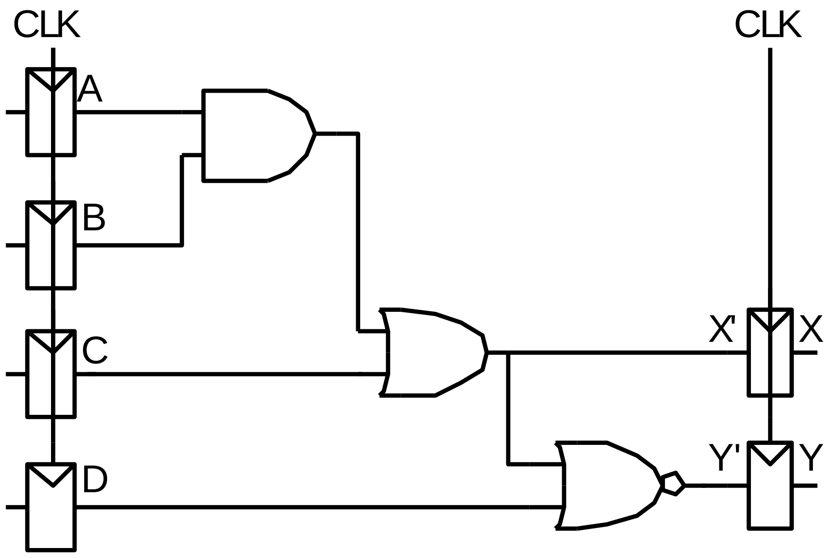
$T_c \geq$

$f_c = 1/T_c =$

Hold time constraint:

$t_{ccq} + t_{cd} > t_{hold}$?

Timing Analysis



t_{pd} = 3 x 35 ps = 105 ps

t_{cd} = 25 ps

Timing Characteristics

t_{ccq}	= 30 ps
t_{pcq}	= 50 ps
t_{setup}	= 60 ps
t_{hold}	= 70 ps
per gate t_{pd}	= 35 ps
per gate t_{cd}	= 25 ps

Setup time constraint:

$T_c \geq (50 + 105 + 60)$ ps = 215 ps

$f_c = 1/T_c$ = 4.65 GHz

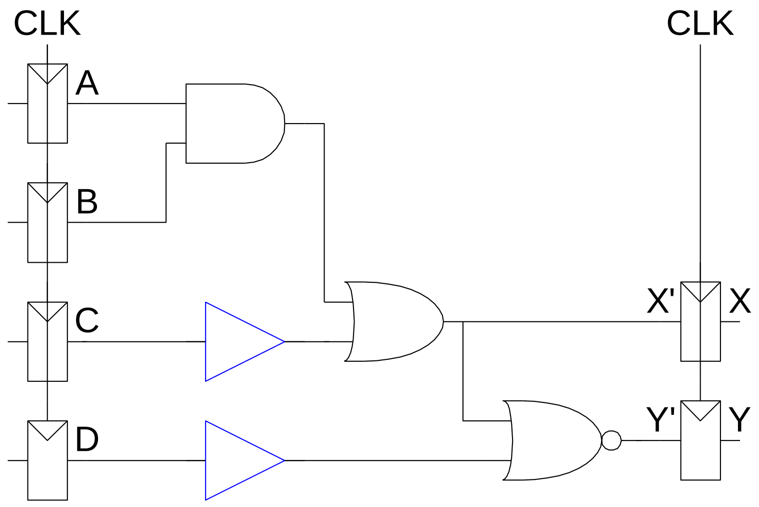
Hold time constraint:

$t_{ccq} + t_{cd} > t_{hold}$?

(30 + 25) ps > 70 ps ? **No!**

Fixing Hold Time Violation

Add buffers to the short paths:



t_{pd} =

t_{cd} =

Setup time constraint:

$T_c \geq$

f_c =

Timing Characteristics

t_{ccq}	= 30 ps
t_{pcq}	= 50 ps
t_{setup}	= 60 ps
t_{hold}	= 70 ps

per gate

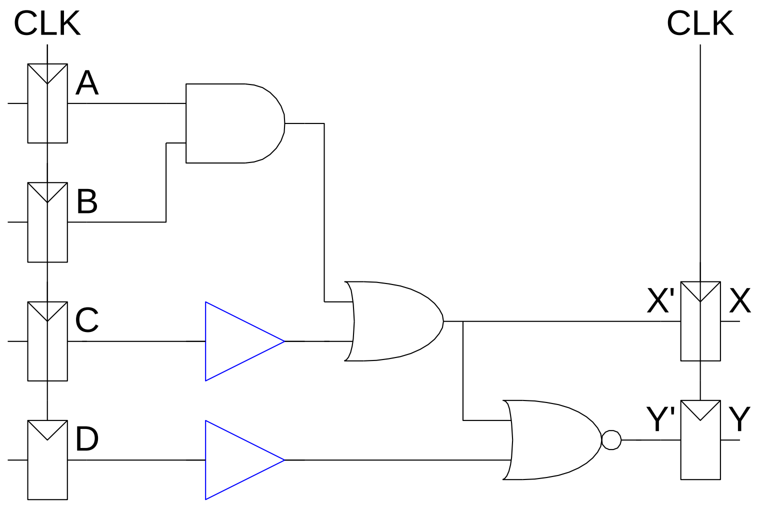
t_{pd}	= 35 ps
t_{cd}	= 25 ps

Hold time constraint:

$t_{ccq} + t_{cd} > t_{hold}$?

Fixing Hold Time Violation

Add buffers to the short paths:



t_{pd} = 3 x 35 ps = 105 ps

t_{cd} = 2 x 25 ps = 50 ps

Timing Characteristics

t_{ccq}	= 30 ps
t_{pcq}	= 50 ps
t_{setup}	= 60 ps
t_{hold}	= 70 ps

per gate

t_{pd}	= 35 ps
t_{cd}	= 25 ps

Setup time constraint:

$T_c \geq (50 + 105 + 60)$ ps = 215 ps

$f_c = 1/T_c$ = 4.65 GHz

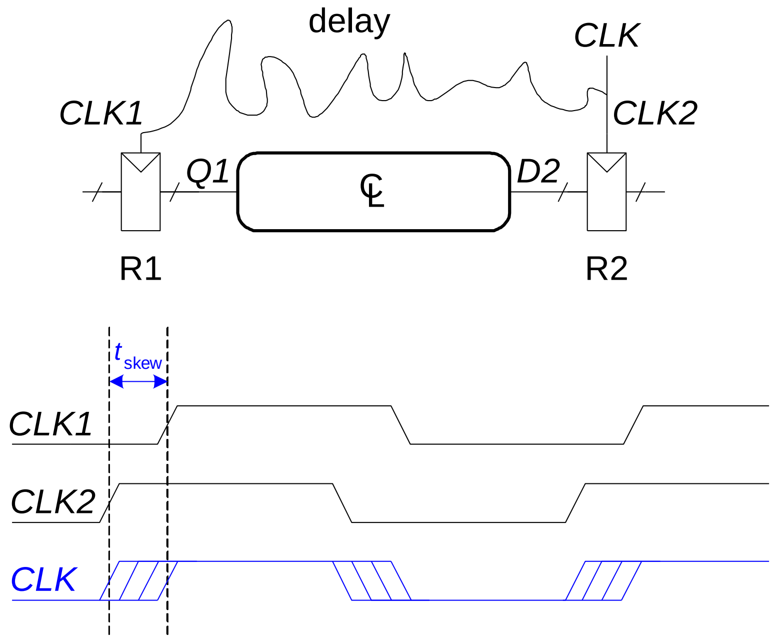
Hold time constraint:

$t_{ccq} + t_{cd} > t_{hold}$?

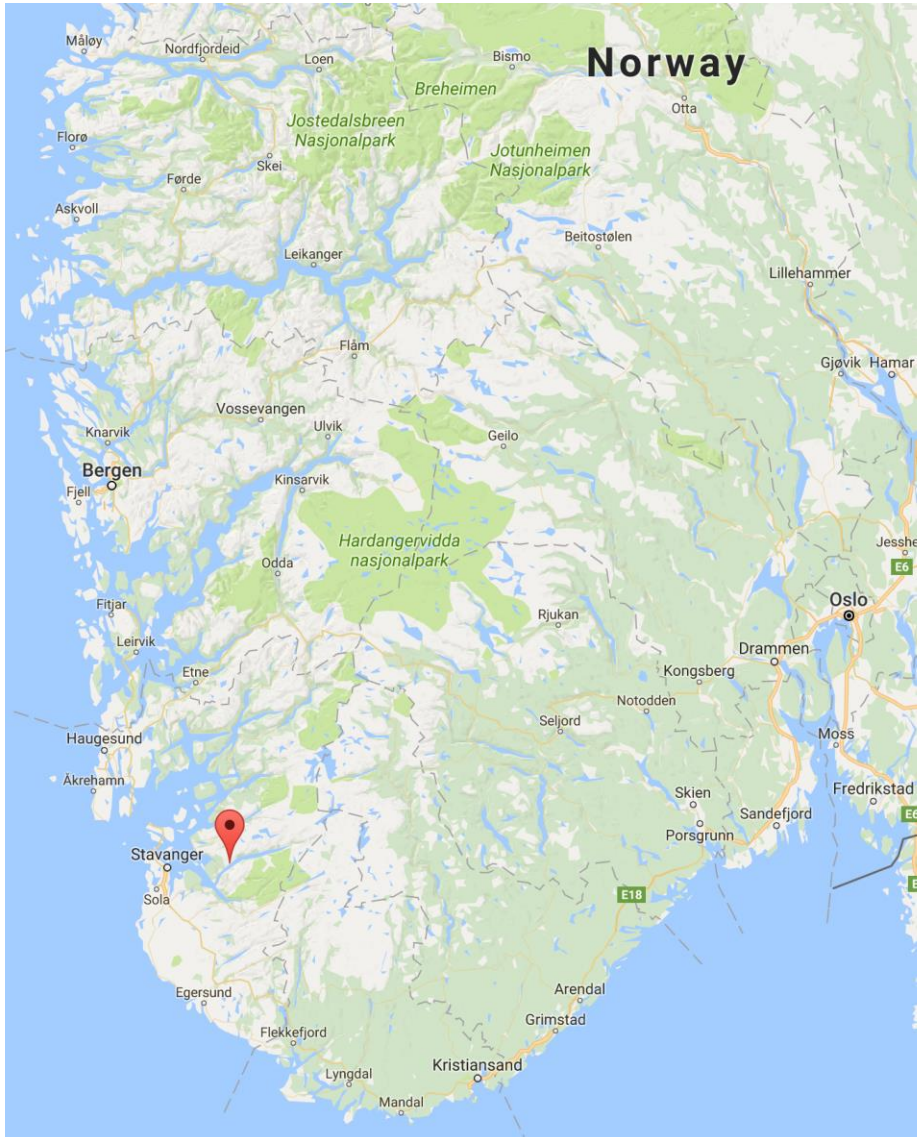
(30 + 50) ps > 70 ps ? **Yes!**

Clock Skew

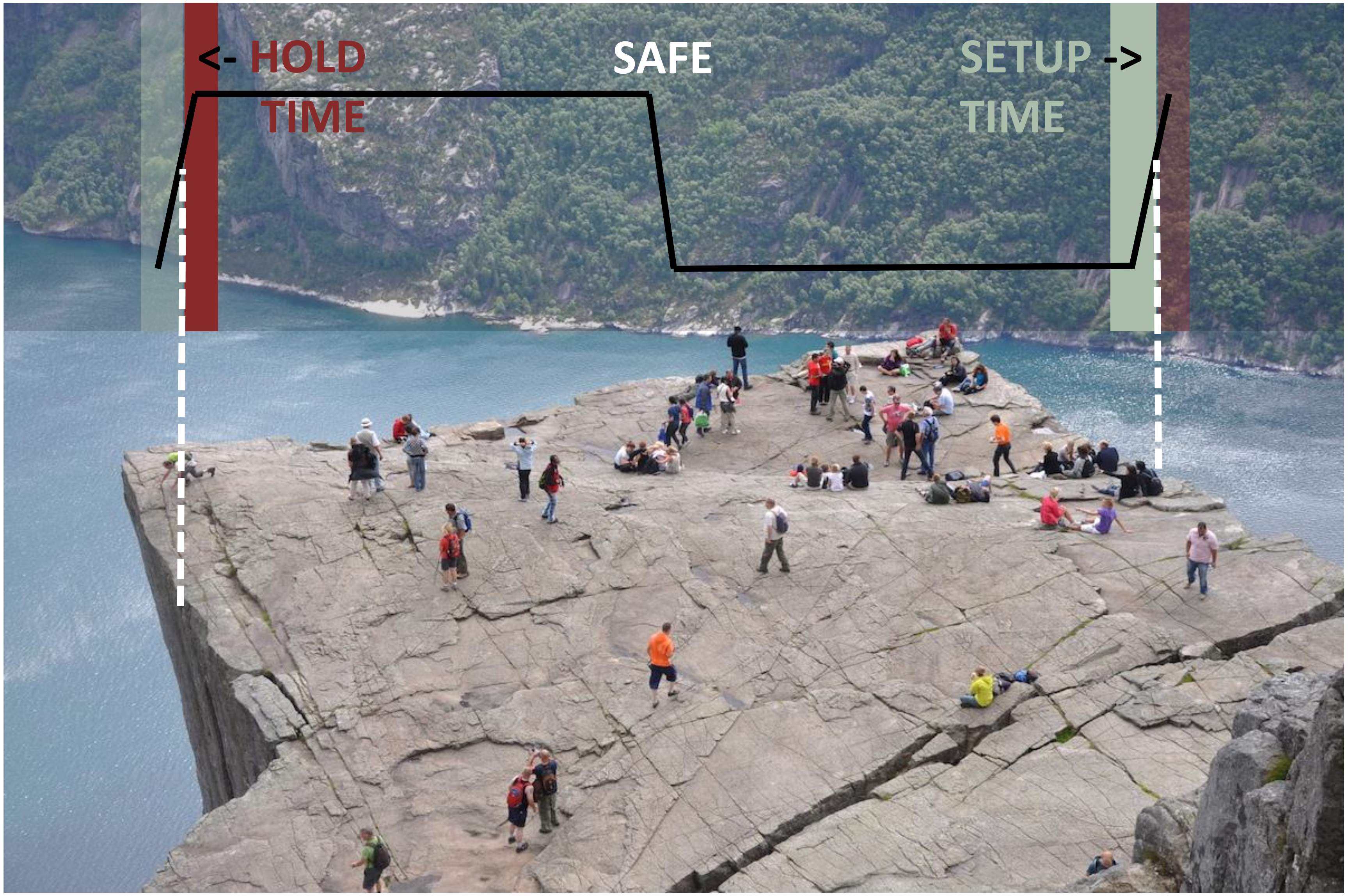
- The clock doesn’t arrive at all registers at the same time
- Skew is the difference between two clock edges
- Examine the worst case to guarantee that the dynamic discipline is not violated for any register – many registers in a system!



Preikestolen - Norway



Stay away from both HOLD and SETUP !



How Do You Know That A Circuit Works?

- **You have written the Verilog code of a circuit**
  - Does it work correctly?
  - Even if the syntax is correct, it might do what you want?
  - What exactly it is that you want anyway?

- **Trial and error can be costly**
  - You need to ‘test’ your circuit in advance

- **In modern digital designs, functional verification is the most time consuming design stage.**

The Idea Behind A Testbench

- **Using a computer simulator to test your circuit**
  - You instantiate your design
  - Supply the circuit with some inputs
  - See what it does
  - Does it return the “correct” outputs?

Testbenches

- **HDL code written to test another HDL module, the *device under test* (dut), also called the *unit under test* (uut)**

- **Not synthesizeable**

- **Types of testbenches:**
  - Simple testbench
  - Self-checking testbench
  - Self-checking testbench with testvectors

Example

- **Write Verilog code to implement the following function in hardware:**

$$y = (\overline{b} \cdot \overline{c}) + (a \cdot \overline{b})$$

- **Name the module sillyfunction**

Example

- **Write Verilog code to implement the following function in hardware:**

$$y = (\overline{b} \cdot \overline{c}) + (a \cdot \overline{b})$$

- **Name the module sillyfunction**

```
module sillyfunction(input  a, b, c,
                     output y);

  assign y = ~b & ~c | a & ~b;
endmodule
```

Simple Testbench

```
module testbench1(); // Testbench has no inputs, outputs
  reg  a, b, c;      // Will be assigned in initial block
  wire y;

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

  // apply inputs one at a time
  initial begin                    // sequential block
    a = 0; b = 0; c = 0; #10; // apply inputs, wait 10ns
    c = 1; #10;               // apply inputs, wait 10ns
    b = 1; c = 0; #10;        // etc .. etc..
    c = 1; #10;
    a = 1; b = 0; c = 0; #10;
  end
endmodule
```

Simple Testbench

- **Simple testbench instantiates the design under test**
- **It applies a series of inputs**
- **The outputs have to be observed and compared using a simulator program.**
  - This type of testbench does not help with the outputs
- `initial` **statement is similar to** `always`**, it just starts once at the beginning, and does not repeat.**
- **The statements have to be blocking.**

Self-checking Testbench

```
module testbench2();
  reg  a, b, c;
  wire y;

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

  // apply inputs one at a time
  initial begin
     a = 0; b = 0; c = 0; #10; // apply input, wait
     if (y !== 1) $display("000 failed."); // check
     c = 1; #10;               // apply input, wait
     if (y !== 0) $display("001 failed."); // check
     b = 1; c = 0; #10;        // etc.. etc..
     if (y !== 0) $display("010 failed."); // check
  end
endmodule
```

Self-checking Testbench

- **Better than simple testbench**
- **This testbench also includes a statement to check the current state**
- **`$display` will write a message in the simulator**
- **This is a lot of work**
  - Imagine a 32-bit processor executing a program (thousands of clock cycles)
- **You make the same amount of mistakes when writing testbenches as you do writing actual code**

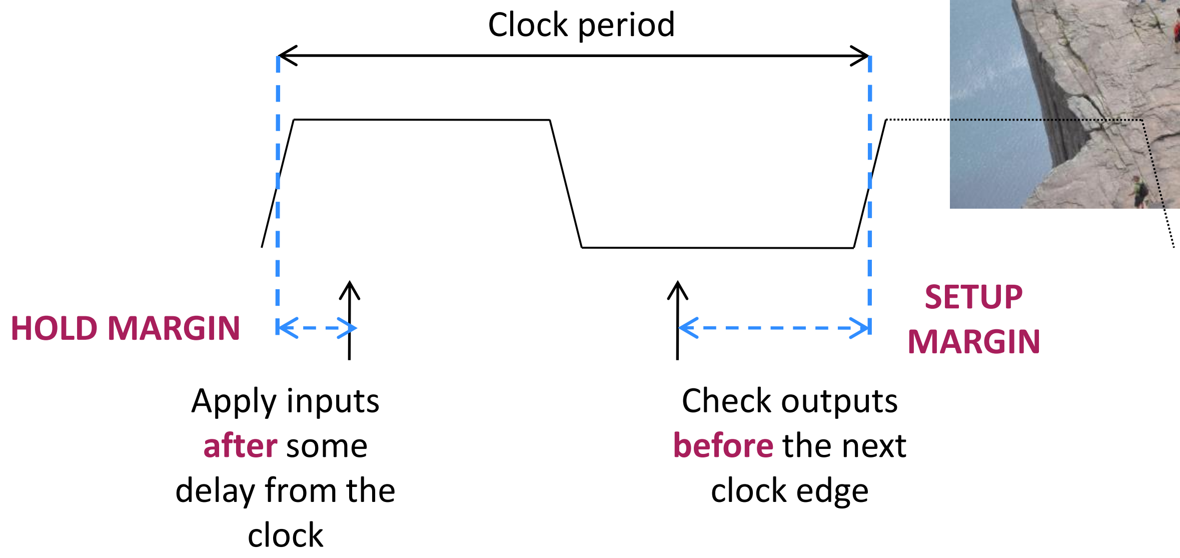
Testbench with Testvectors

- **The more elaborate testbench**

- **Write testvector file: inputs and expected outputs**
  - Usually can use a high-level model (golden model) to produce the ‘correct’ input output vectors

- **Testbench:**
  - Generate clock for assigning inputs, reading outputs
  - Read testvectors file into array
  - Assign inputs, get expected outputs from DUT
  - Compare outputs to expected outputs and report errors

Testbench with Testvectors


Clock period
HOLD MARGIN
SETUP MARGIN
Apply inputs after some delay from the clock
Check outputs before the next clock edge

- **A testbench clock is used to synchronize I/O**
  - The same clock can be used for the DUT clock
- **Inputs are applied following a hold margin**
- **Outputs are sampled before the next clock edge**
  - The example in book uses the falling clock edge to sample

Testvectors File

- **We need to generate a testvector file (somehow)**
- **File: example.tv – contains vectors of abc_yexpected**

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

Testbench: 1. Generate Clock

```
module testbench3();
  reg          clk, reset;    // clock and reset are internal
  reg          a, b, c, yexpected;  // values from testvectors
  wire         y;                   // output of circuit
  reg  [31:0] vectornum, errors;    // bookkeeping variables
  reg  [3:0]  testvectors[10000:0];// array of testvectors

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

  // generate clock
  always     // no sensitivity list, so it always executes
    begin
      clk = 1; #5; clk = 0; #5;     // 10ns period
    end
```

2. Read Testvectors into Array

```
// at start of test, load vectors
// and pulse reset

 initial               // Will execute at the beginning once
    begin
      $readmemb("example.tv", testvectors); // Read vectors
      vectornum = 0; errors = 0;            // Initialize
      reset = 1; #27; reset = 0;       // Apply reset wait
    end


// Note: $readmemh reads testvector files written in
// hexadecimal
```

3. Assign Inputs and Expected Outputs

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

- **Apply inputs with some delay (1ns) AFTER clock**
- **This is important**
  - Inputs should not change at the same time with clock
- **Ideal circuits (HDL code) are immune, but real circuits (netlists) may suffer from hold violations.**

4. Compare Outputs with Expected Outputs

```
// check results on falling edge of clk
   always @(negedge clk)
    if (~reset)                    // skip during reset
      begin
      if (y !== yexpected)
      begin
        $display("Error: inputs = %b", {a, b, c});
        $display("  outputs = %b (%b exp)",y,yexpected);
        errors = errors + 1;
      end
// Note: to print in hexadecimal, use %h. For example,
//       $display(“Error: inputs = %h”, {a, b, c});
```

4. Compare Outputs with Expected Outputs

```
// increment array index and read next testvector
      vectornum = vectornum + 1;
      if (testvectors[vectornum] === 4'bx)
      begin
          $display("%d tests completed with %d errors",
                    vectornum, errors);
          $finish;                        // End simulation
      end
    end
endmodule

// Note: === and !== can compare values that are
// x or z.
```

Golden Models

- **A golden model represents the ideal behavior of your circuit.**
  - Still it has to be developed
  - It is difficult to get it right (bugs in the golden model!)
  - Can be done in C, Perl, Python, Matlab or even in Verilog

- **The behavior of the circuit is compared against this golden model.**
  - Allows automated systems (very important)

Why is Verification difficult?

- **How long would it take to test a 32-bit adder?**
  - In such an adder there are 64 inputs = 2^{64} possible inputs
  - That makes around 1.85 10^{19} possibilities
  - If you test one input in 1ns, you can test 10^{9} inputs per second
    - or 8.64 x 10^{14} inputs per day
    - or 3.15 x 10^{17} inputs per year
  - we would still need **58.5 years** to test all possibilities

- **Brute force testing is not feasible for all circuits, we need alternatives**
  - Formal verification methods
  - Choosing ‘critical cases’
  - Not an easy task