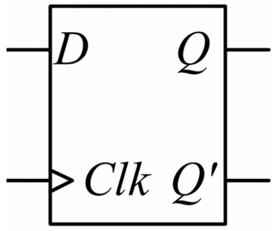
Digital Logic Design

Lecture 7

Timing of sequential logic

A flip-flop copies the input *D* to the output *Q* on the rising edge of the clock.

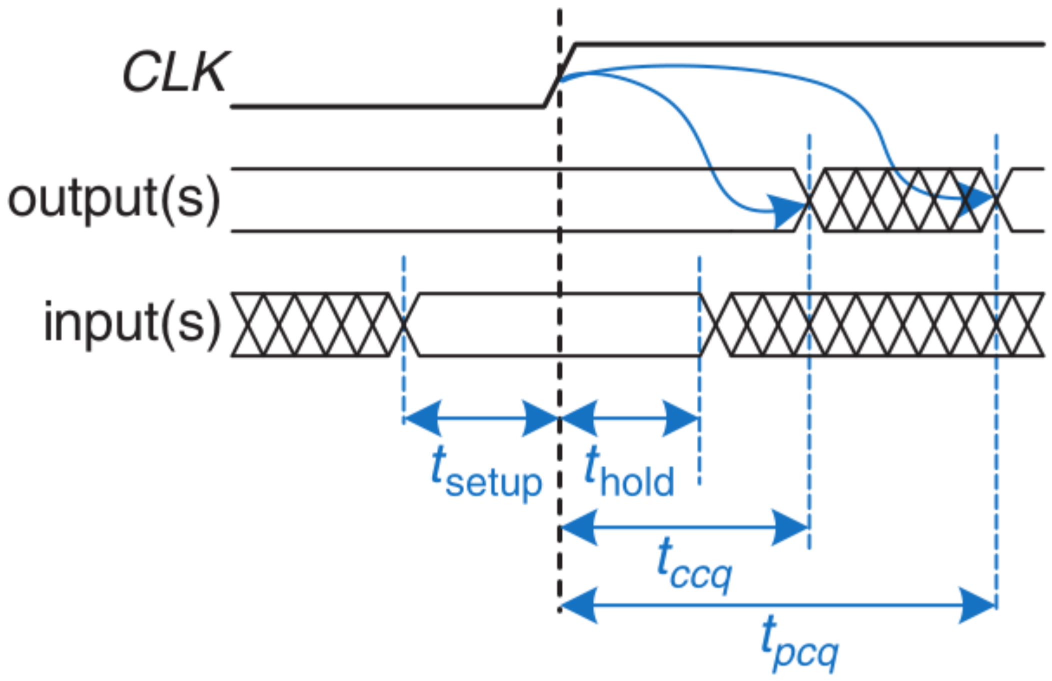


This process is called **sampling** *D* on the clock edge.

If *D* is **stable** at either 0 or 1 when the clock rises, this behavior is clearly defined.

If *D* is changing at the same time the clock rises, we may have timing problems.

Timing specification for synchronous sequential circuit



When the clock rises, the output may start to change after the **clock-to-Q contamination delay**,

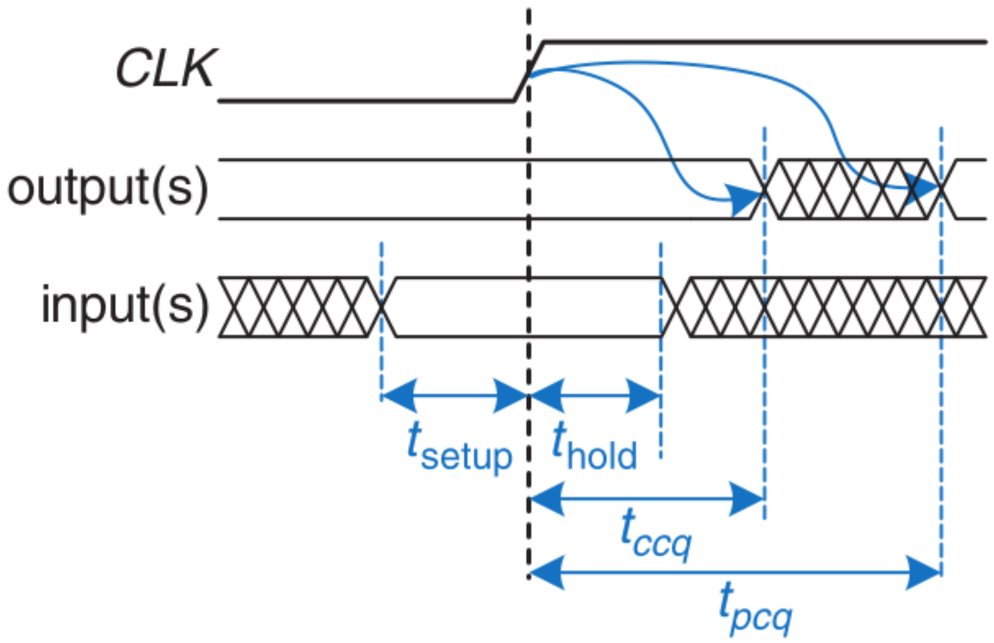
$$t_{ccq}$$

The output must definitely settle to the final value within the **clock-to-Q propagation delay**,

$$t_{pcq}$$

For the circuit to sample its input correctly, the input must have stabilized at least some **setup time**,

$$t_{setup}$$



before the rising edge of the clock

and must remain stable for at least some **hold time**,

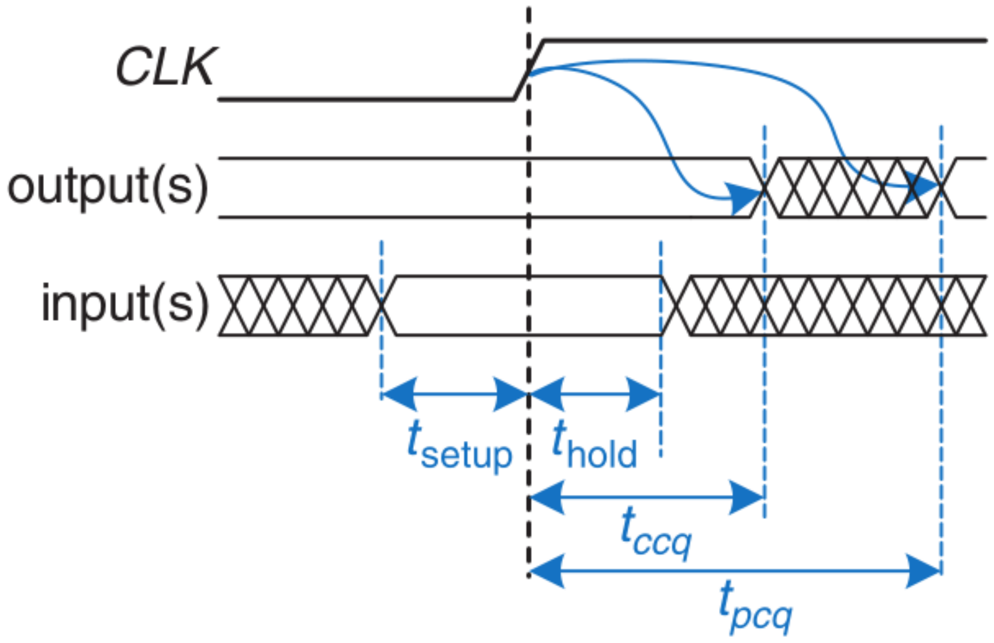
$$t_{hold}$$

after the rising edge of the clock.

The **aperture time** of the circuit:

$$t_{aperture} = t_{setup} + t_{hold}$$

The **dynamic discipline** states that the inputs of a synchronous sequential circuit must be stable during the aperture time around the clock edge.

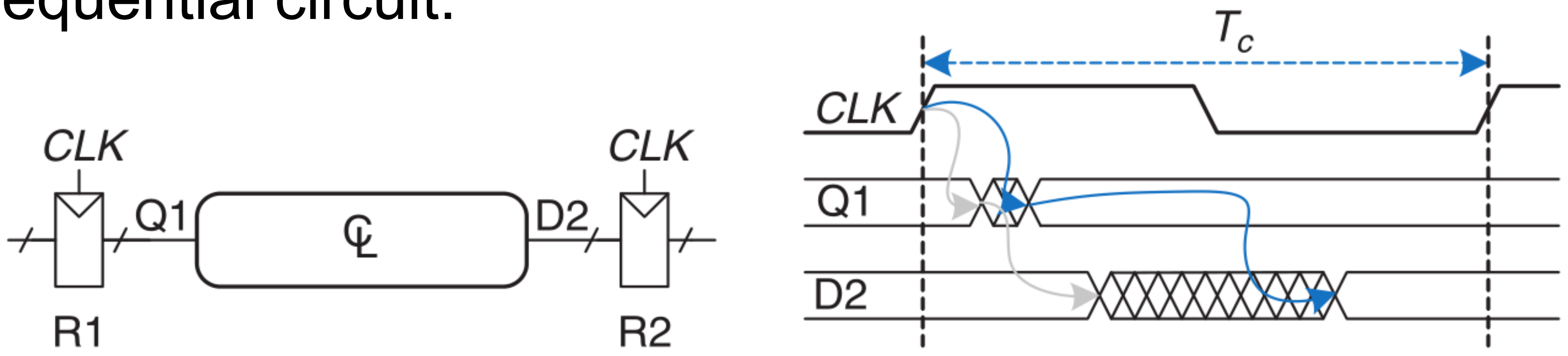


By imposing this requirement, we guarantee that the flip-flops sample signals while they are not changing.

Because we are concerned only about the final values of the inputs at the time they are sampled, we can treat signals as discrete in time as well as in logic levels.

We write *A*[*n*], the value of signal *A* at the end of the *n*th clock cycle, where *n* is an integer, rather than *A*(*t*), the value of *A* at some instant *t*, where *t* is a real number.

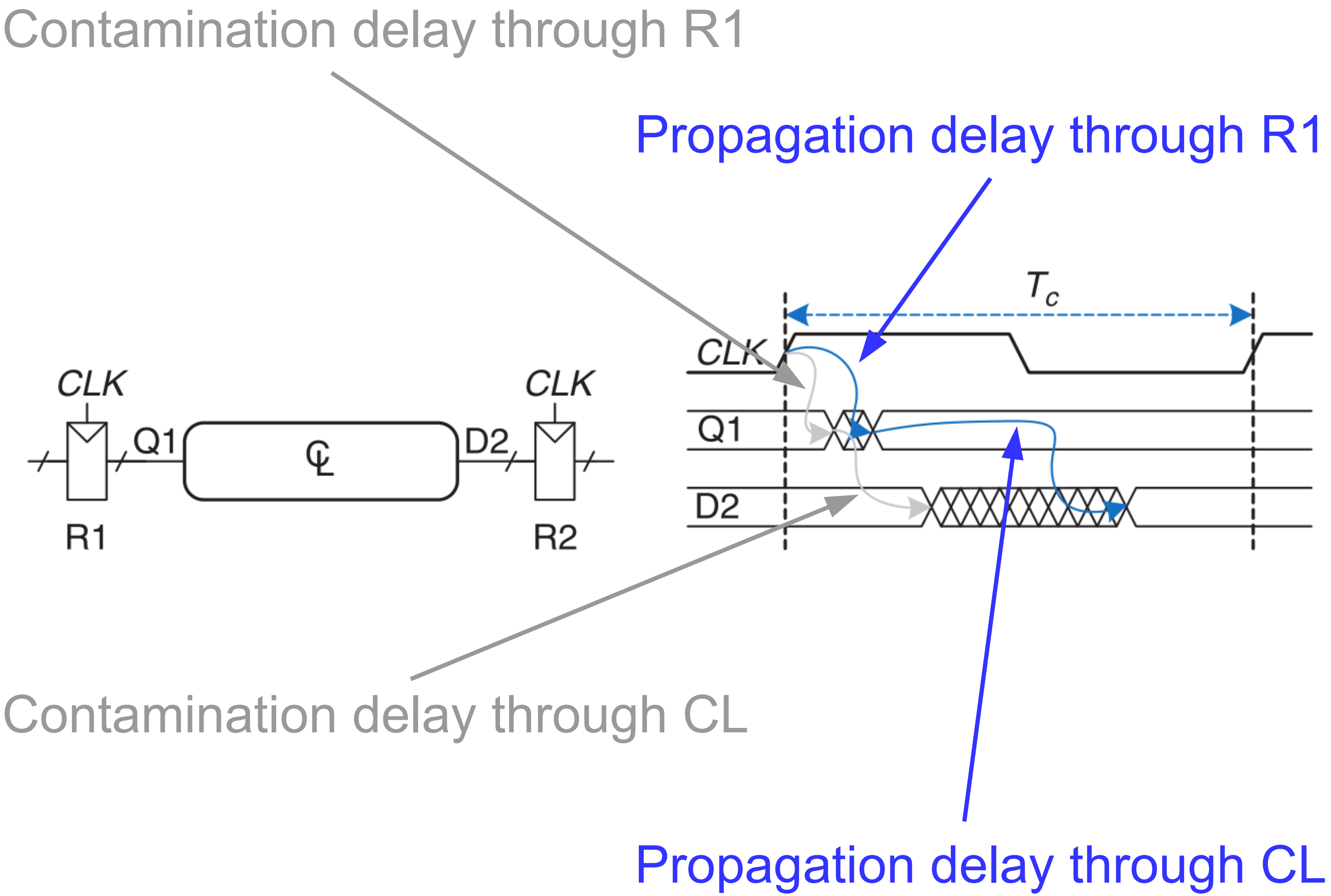
A generic path in a synchronous sequential circuit:

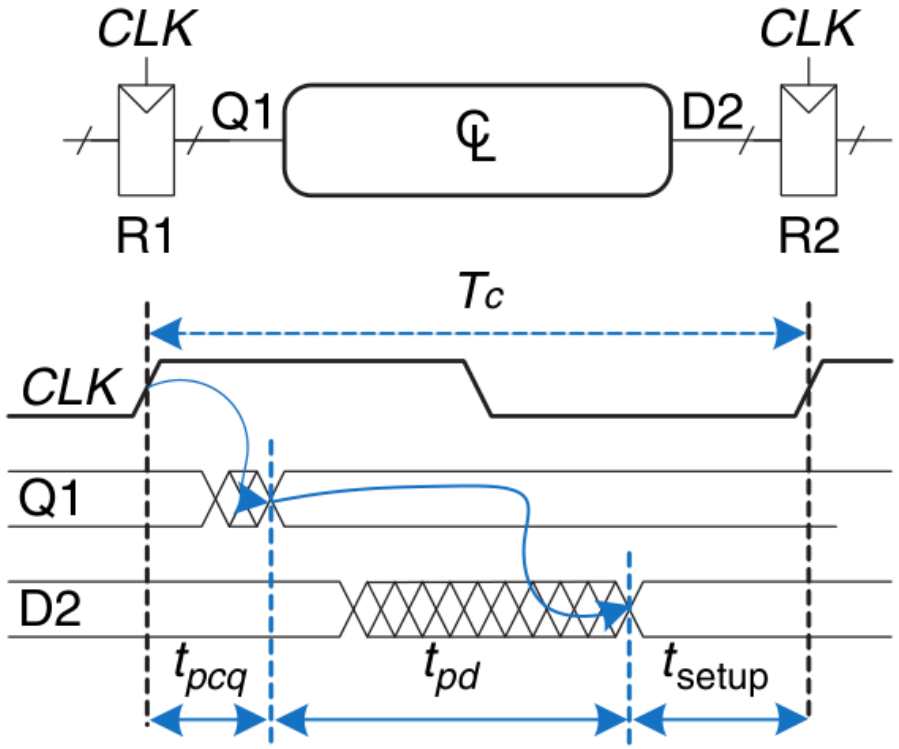


On the rising edge of the clock, register R1 produces output Q1.

The signal enter a block of combinational logic, producing D2, the input to register R2.

Each output signal may start to change a contamination delay after its input changes and settles to the final value within a propagation delay after its input settles.


Contamination delay through R1
Propagation delay through R1
T_c
CLK
Q1
D2
CLK
CLK
Q1
CL
D2
R1
R2
Contamination delay through CL
Propagation delay through CL



To satisfy the setup time of R2, D2 must settle no later than the setup time before the next clock edge.

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

From this inequality, we get **the setup time constraint** also known as **max-delay constraint**

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

$t_{pcq} + t_{setup}$ is called the **sequencing overhead**

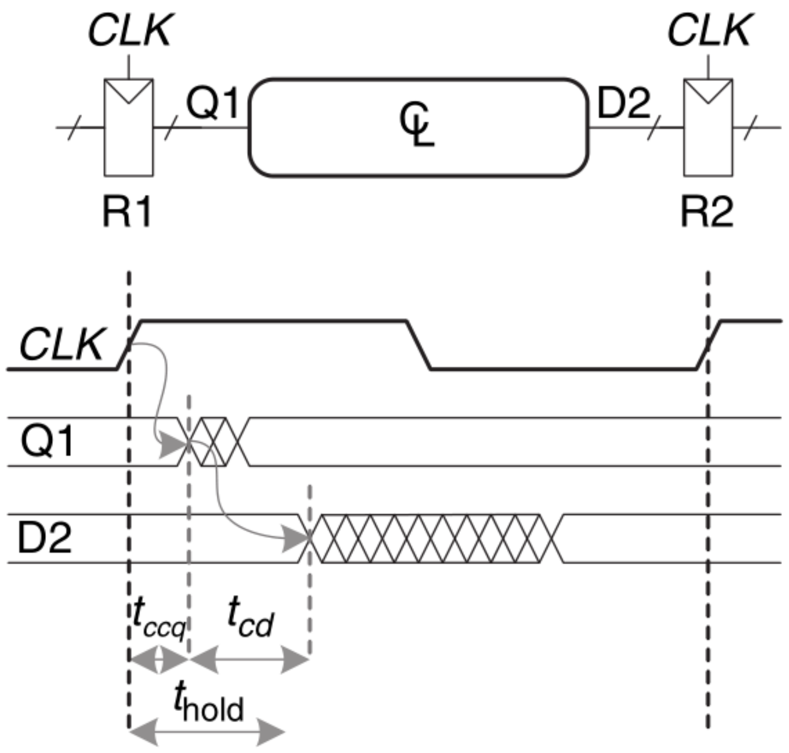
$$t_{pd} \leq T_c - \left(t_{pcq} + t_{setup}\right)$$

If the propagation delay through the combinational logic is too great, D2 may not have settled to its final value by the time R2 needs it to be stable and samples it.

Hence, R2 may sample an incorrect result or even an illegal logic level, a level in the forbidden region.

In such a case, the circuit will malfunction.

The problem can be solved by increasing the clock period or by redesigning the combinational logic to have a shorter propagation delay.



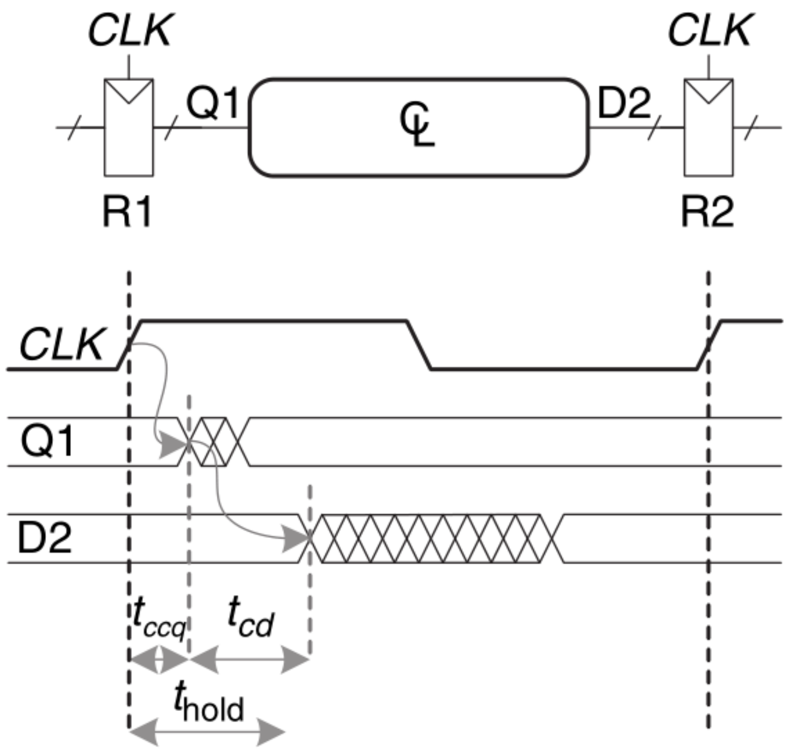
The register R2 also has a **hold time constraint**.

Its input, D2, must not change until some time, t_{hold} after the rising edge of the clock

D2 might change as soon as $t_{ccq}+t_{cd}$ after the rising edge of the clock.

Hence,

$$t_{ccq}+t_{cd}\geq t_{hold}$$



$$t_{ccq} + t_{cd} \geq t_{hold}$$

Rearranging, we solve for the minimum contamination delay through the combinational logic:

$$t_{cd} \geq t_{hold} - t_{ccq}$$

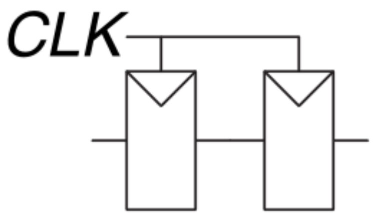
This equation is also called the **hold time constraint** or **min-delay constraint**

It limits the minimum delay through combinational logic.

$$t_{cd} \geq t_{hold} - t_{ccq}$$

We have assumed that any logic elements can be connected to each other without introducing timing problems.

In particular, we would expect that two flip-flops may be directly cascaded:



In this case, $t_{cd} = 0$

Thus, it is required that $t_{cd} \geq t_{hold}$

Often, flip-flops are designed with $t_{hold} = 0$

therefore, the hold time requirement is satisfied.

If $t_{hold} > 0$

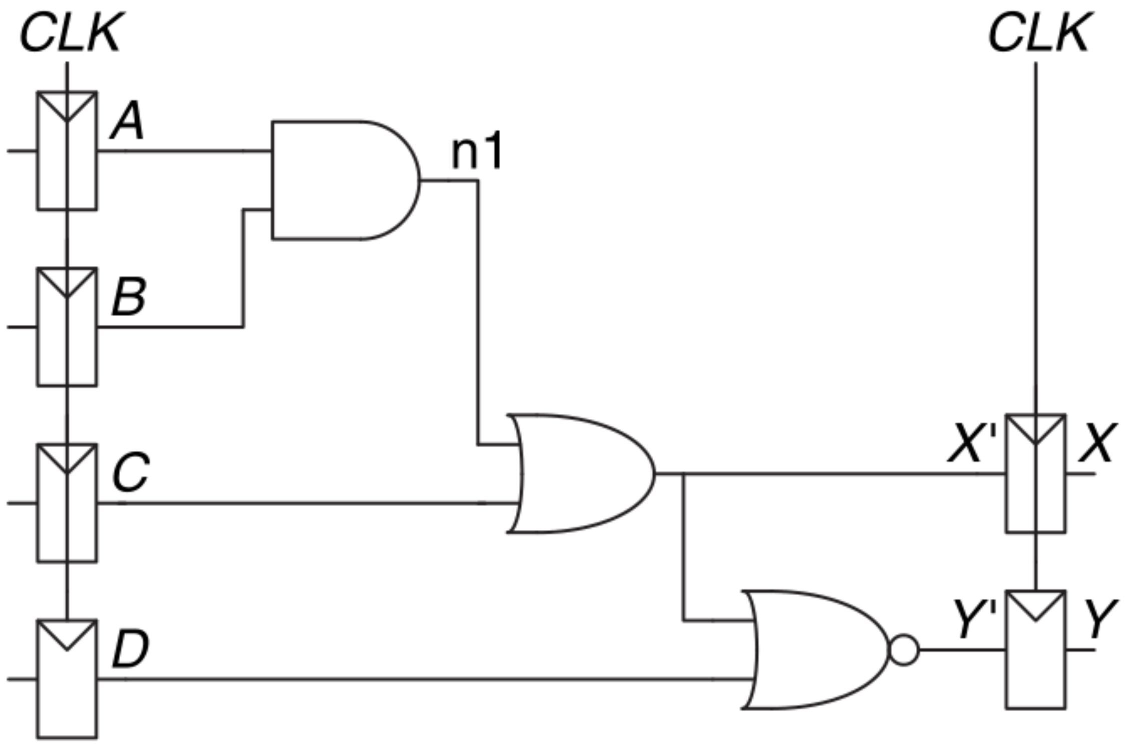
If hold time constraints are violated, the only solution is to increase the contamination delay through the logic.

This requires redesigning the circuit.

Unlike setup time constraints, hold time constraints cannot be fixed by adjusting the clock period.

Redesigning an integrated circuit and manufacturing the corrected design takes months and millions of dollars, so hold time violations must be taken extremely seriously.

Example



Flip-flops: $t_{ccq} = 30\ ps$

$t_{pcq} = 80\ ps$

$t_{setup} = 50\ ps$

$t_{hold} = 60\ ps$

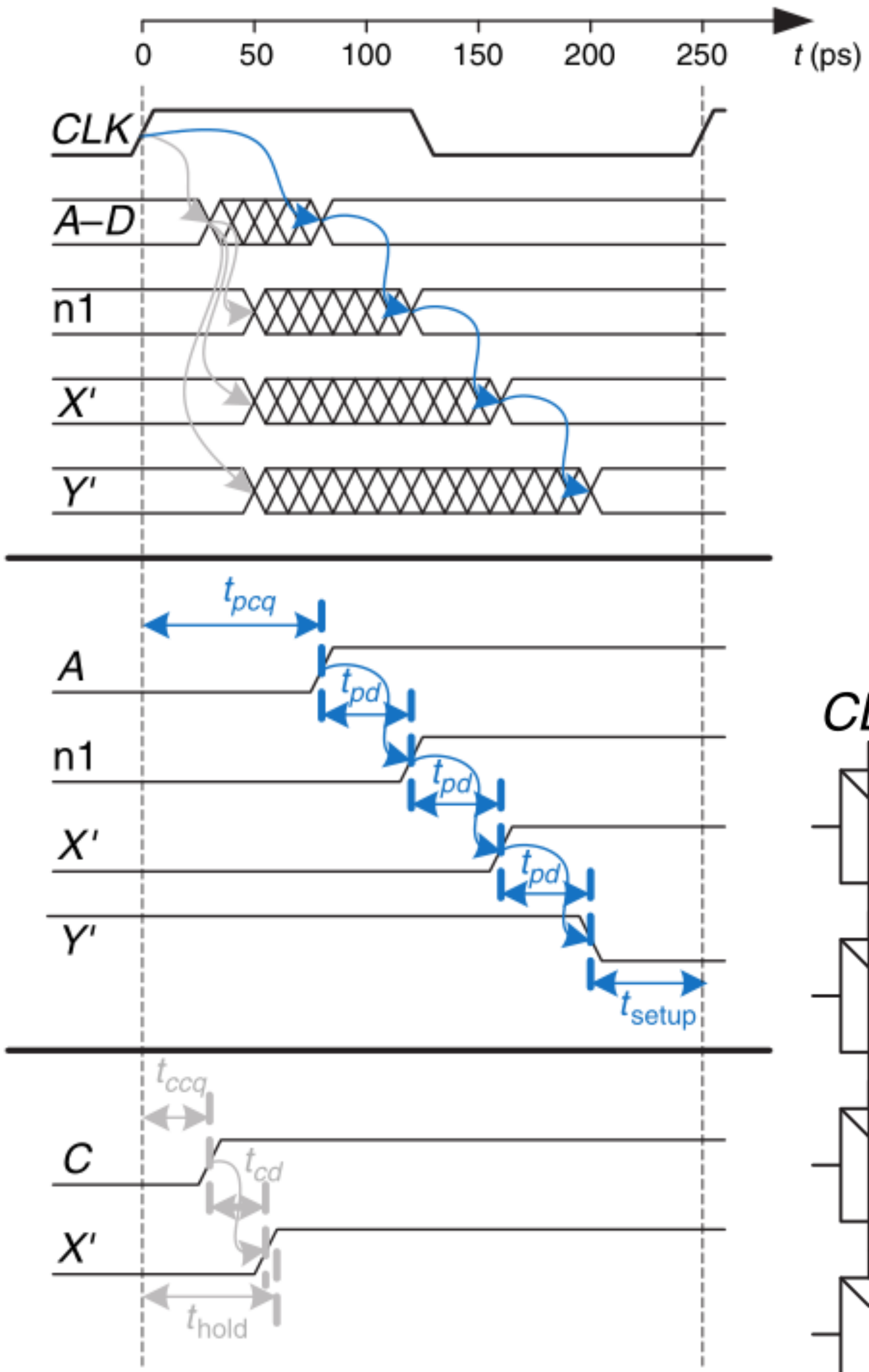
Each logic gate:

$t_{pd} = 40\ ps$

$t_{cd} = 25\ ps$

What is the maximum clock frequency for this circuit?

Are there any hold time violations?



Flip-flops: $t_{ccq} = 30\ ps$

$t_{pcq} = 80\ ps$

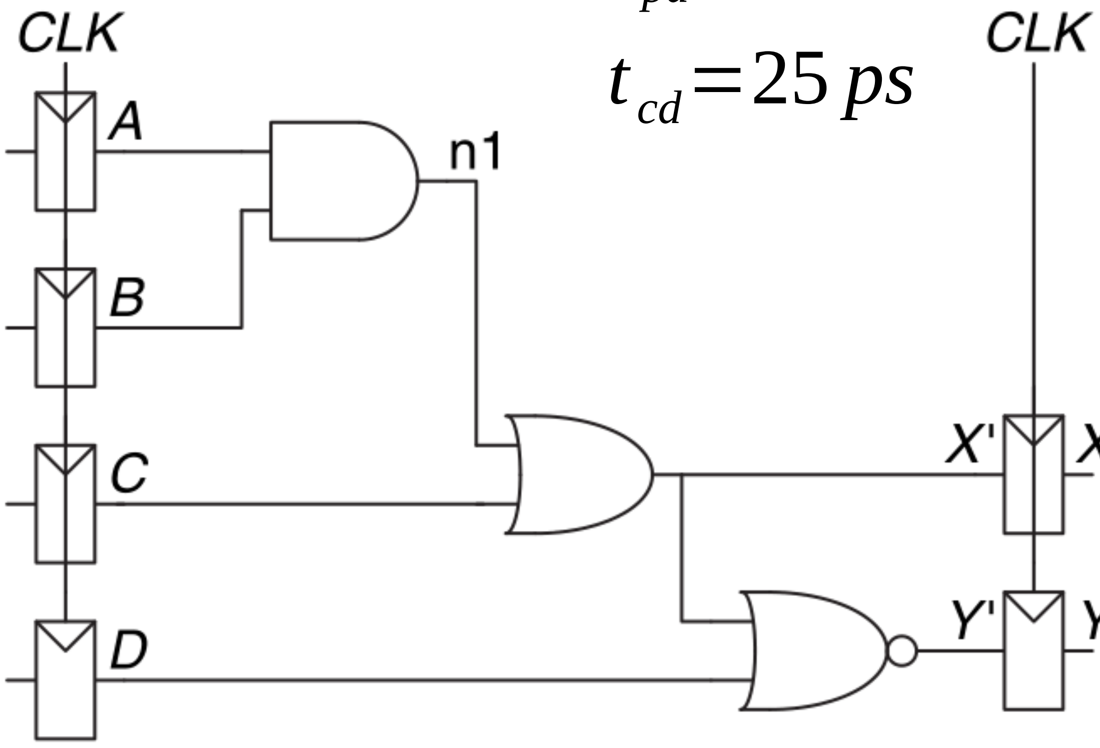
$t_{setup} = 50\ ps$

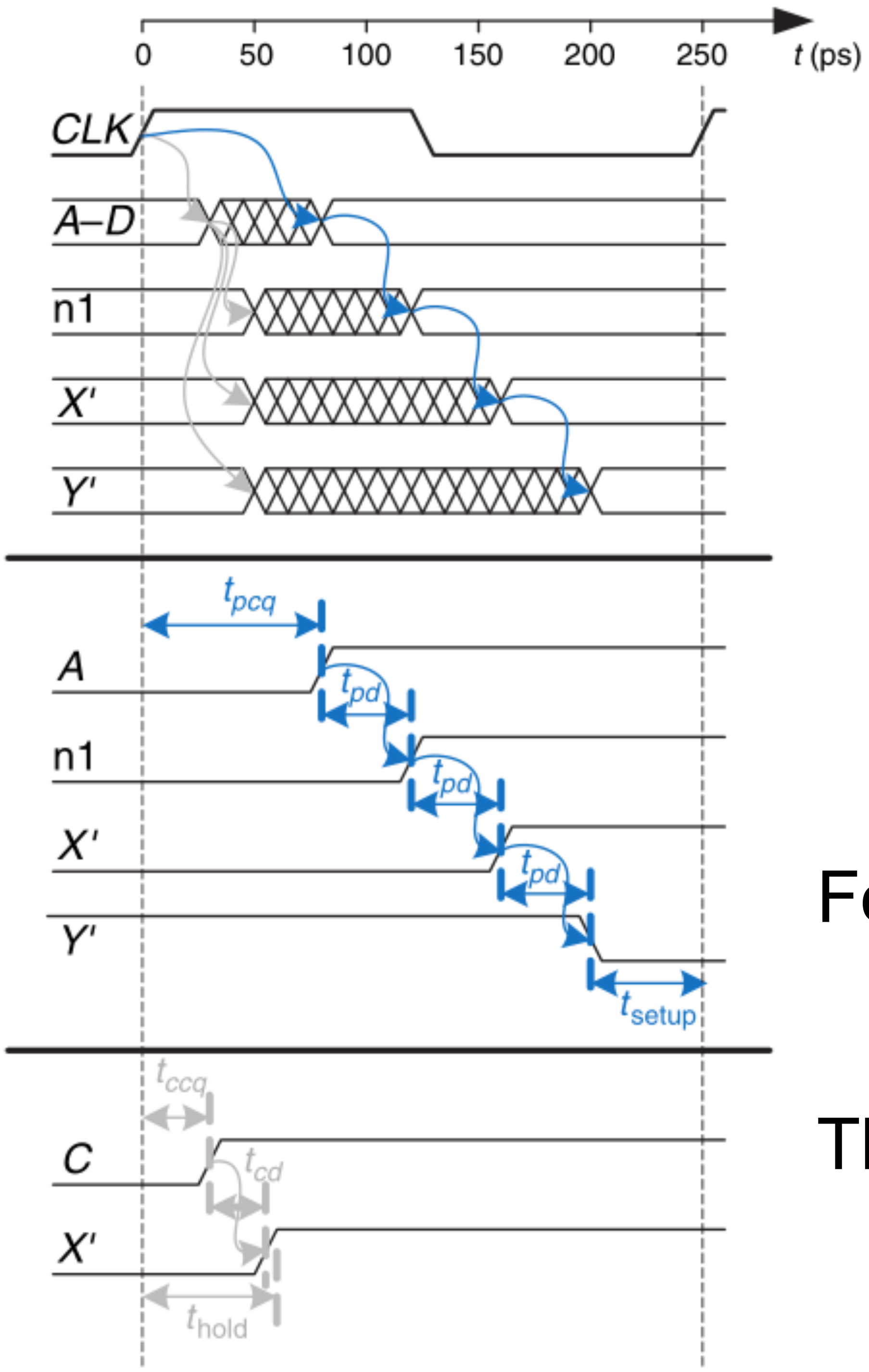
$t_{hold} = 60\ ps$

Each logic gate:

$t_{pd} = 40\ ps$

$t_{cd} = 25\ ps$





Flip-flops: $t_{ccq} = 30\ ps$

$t_{pcq} = 80\ ps$

$t_{setup} = 50\ ps$

$t_{hold} = 60\ ps$

Each logic gate:

$t_{pd} = 40\ ps$

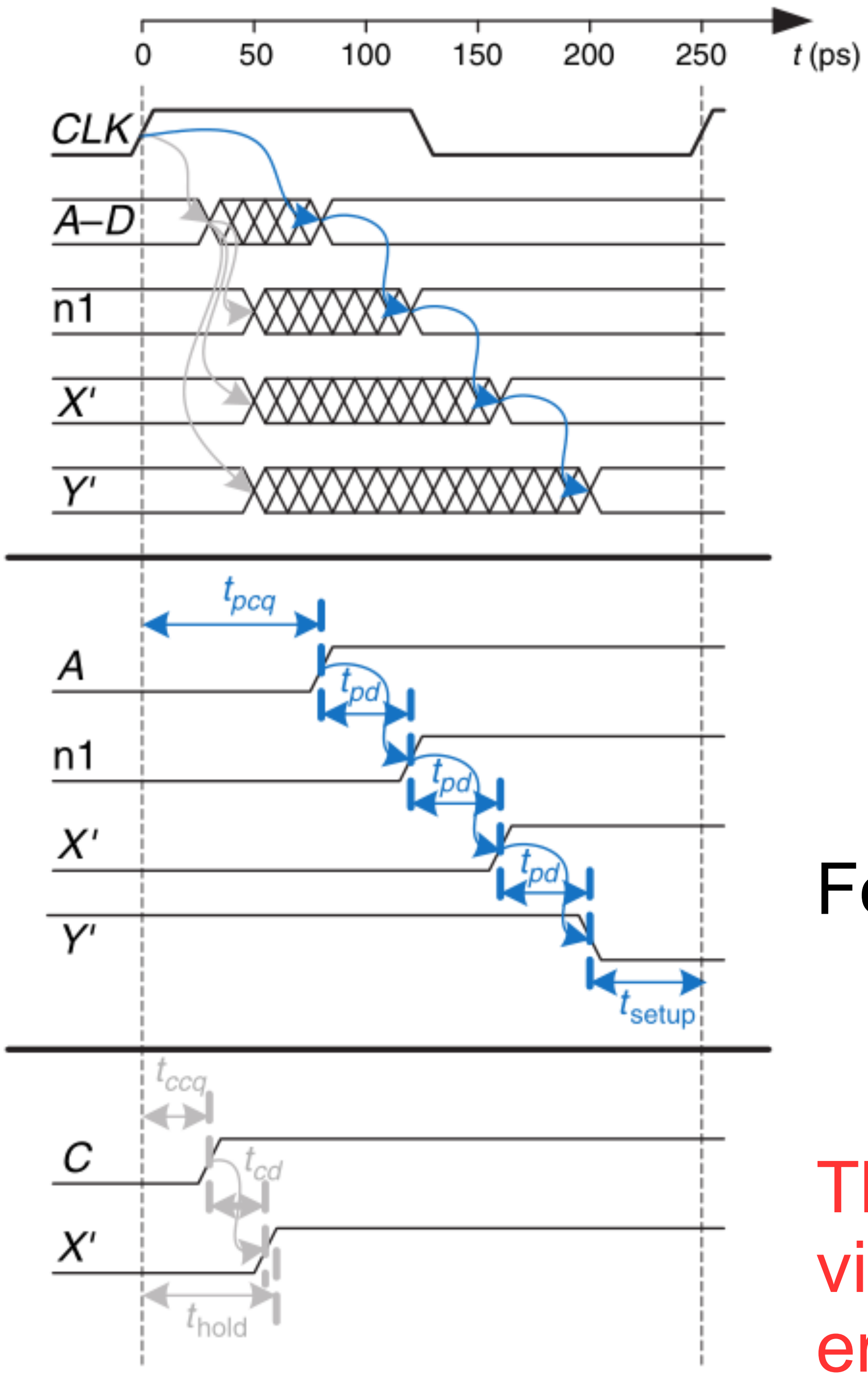
$t_{cd} = 25\ ps$

For the critical path,

$$T_c \geq t_{pcq} + 3\,t_{pd} + t_{setup} = 250\ ps$$

The maximum clock frequency:

$$f = \frac{1}{T_c} = 4\,GHz$$



Flip-flops: $t_{ccq} = 30\ ps$

$t_{pcq} = 80\ ps$

$t_{setup} = 50\ ps$

$t_{hold} = 60\ ps$

Each logic gate:

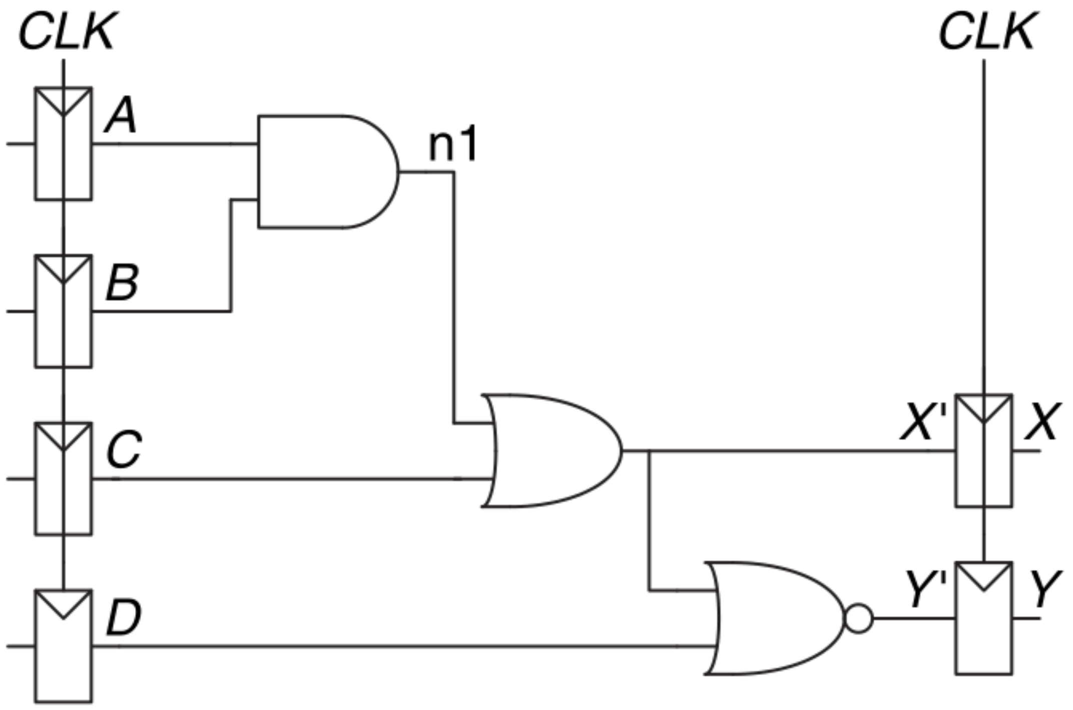
$t_{pd} = 40\ ps$

$t_{cd} = 25\ ps$

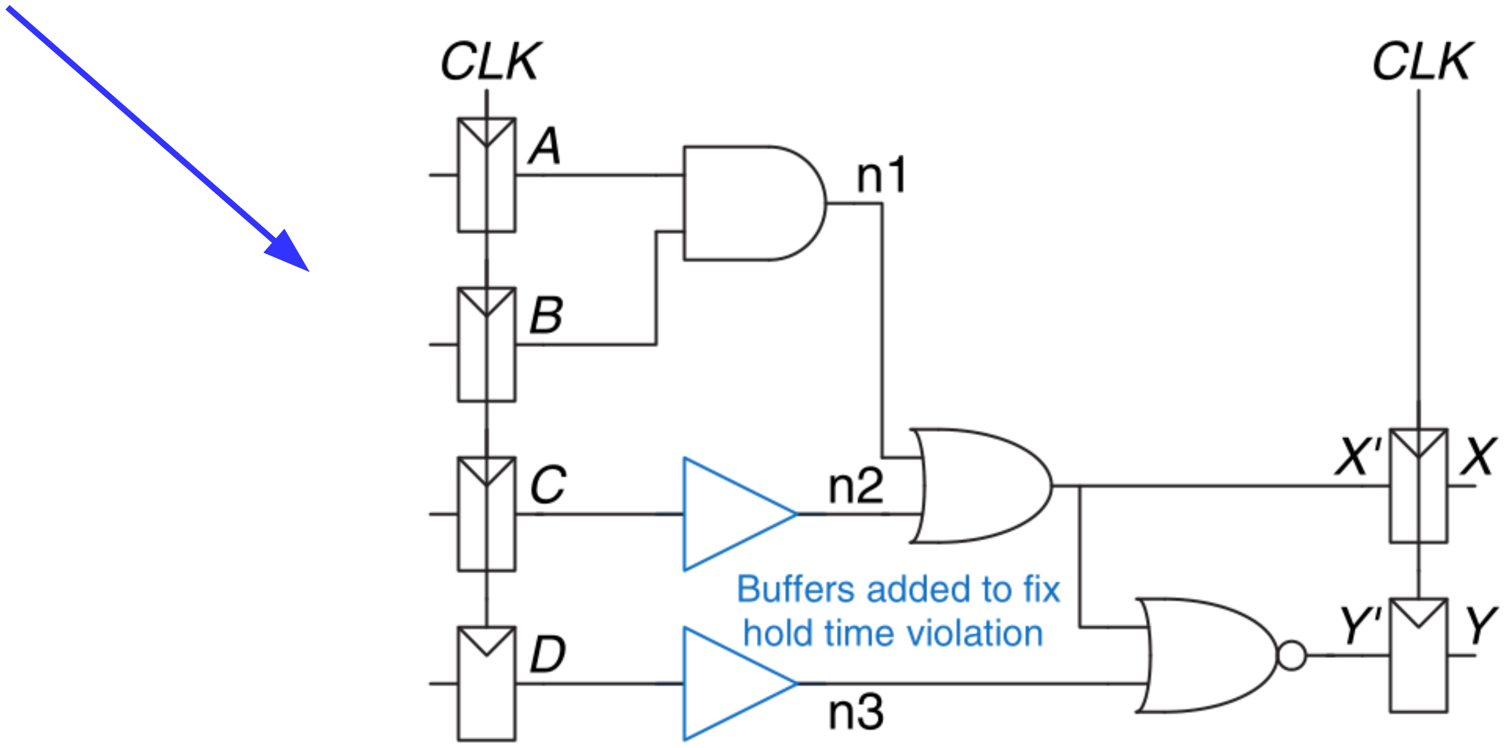
For the short path,

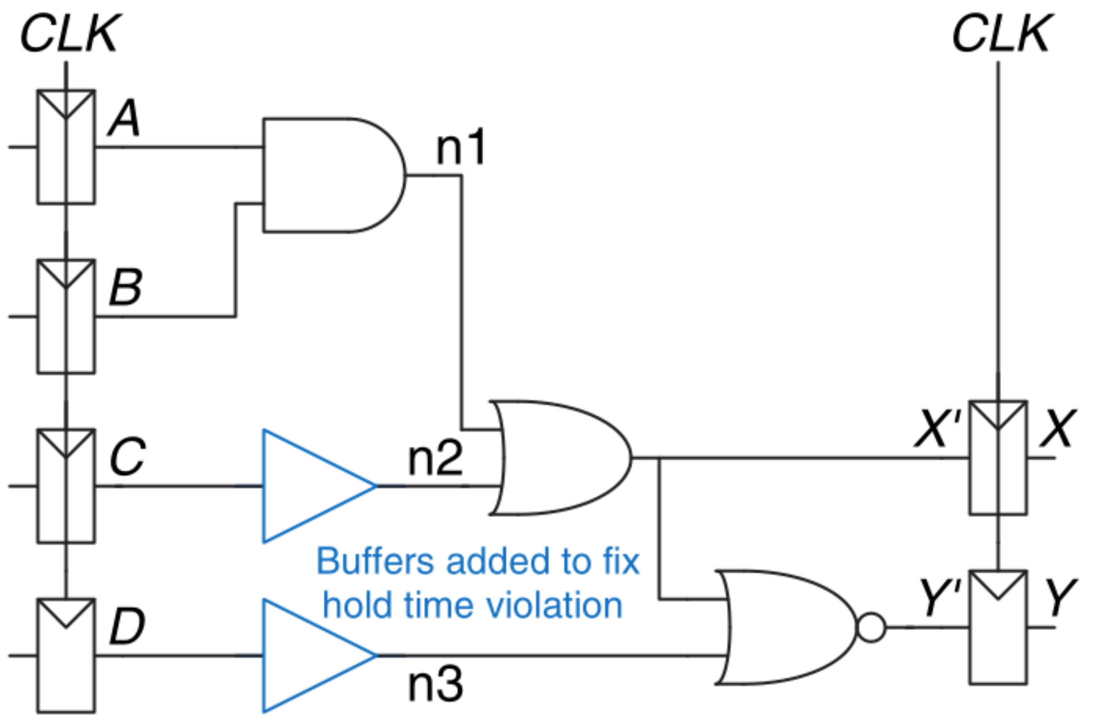
$$t_{ccq} + t_{cd} = 55\ ps \ < \ t_{hold}$$

The circuit has a hold time violation and may behave erratically at any clock frequency!



Fixing hold time violation





Flip-flops: $t_{ccq} = 30\ ps$

$t_{pcq} = 80\ ps$

$t_{setup} = 50\ ps$

$t_{hold} = 60\ ps$

Each logic gate:

$t_{pd} = 40\ ps$

$t_{cd} = 25\ ps$

The critical path is unaffected, therefore the maximum clock frequency is still 4 GHz.

For the short path, $t_{ccq} + 2t_{cd} = 80\ ps\ >\ t_{hold}$

so the circuit now operates correctly.

Clock skew

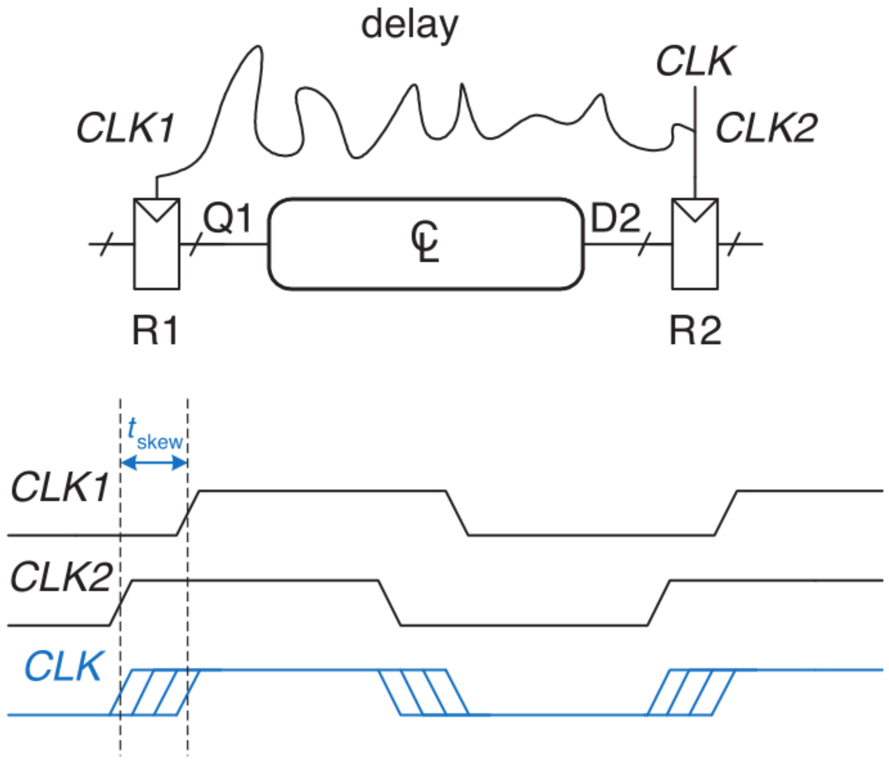
In the previous analysis, we assumed that the clock reaches all registers at exactly the same time.

In reality, there is some variation in this time.

This variation in clock edges is called **clock skew**.

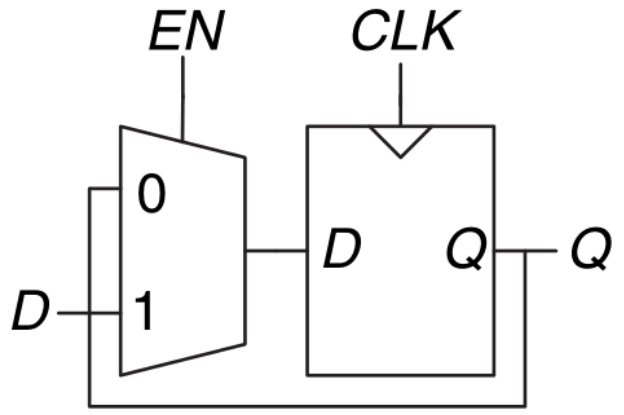
Why clock skew occurs:

- wires have different lengths
- noise
- clock gating

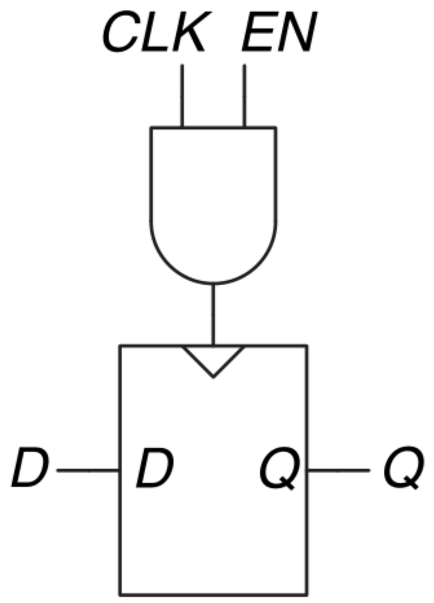


Clock gating

Two ways to construct an enabled flip-flop from a D flip-flop:



1) through a mux



2) clock gating

If *EN* = 0, the *CLK* input is also 0 and the flip-flop retains its old value.

EN must not change while *CLK* = 1, lest the flip-flop see a clock glitch

Clock gating

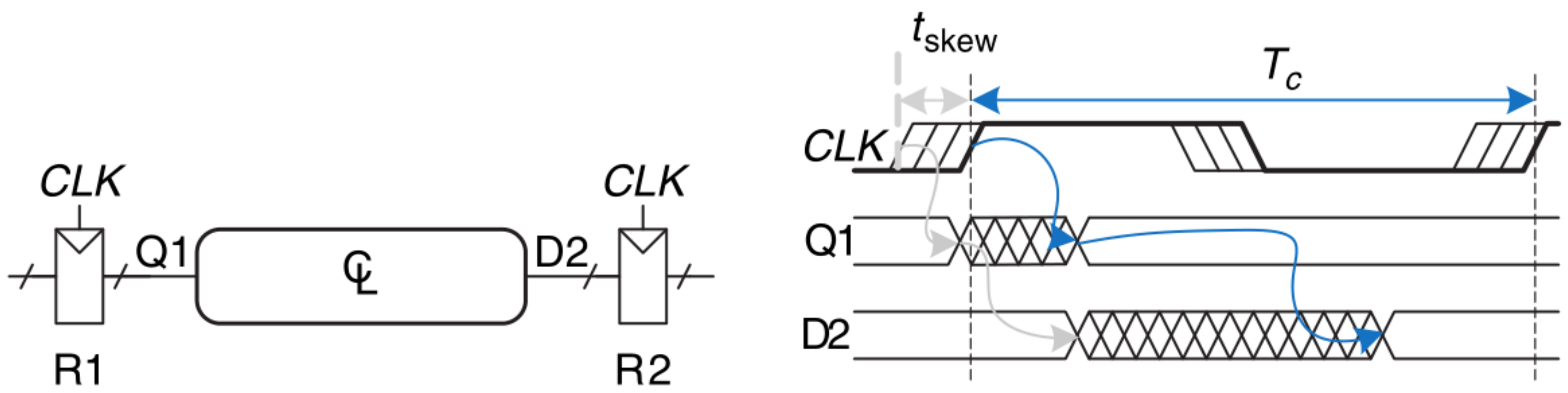
Performing logic on the clock is a bad idea.

Clock gating delays the clock and can cause timing errors.

If some clocks are gated and others are not, there will be substantial skew between the gated and ungated clocks.

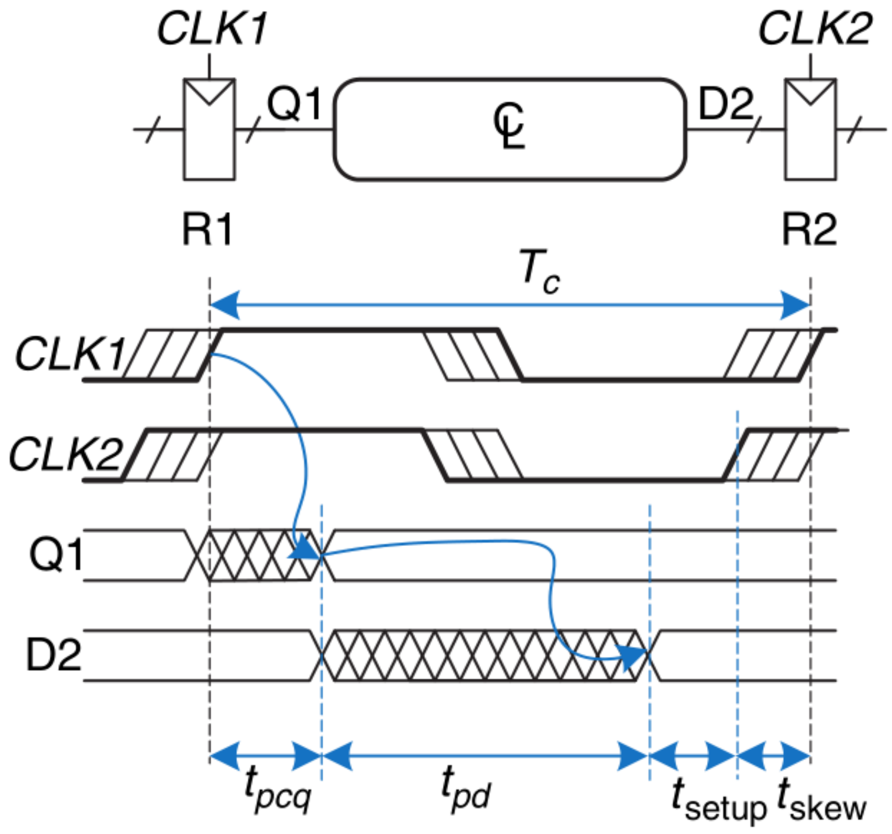
Do clock gating only if you are sure you know what you are doing.

When doing timing analysis, we consider the worst-case scenario, so that we can guarantee that the circuit will work under all circumstances.



heavy line: the latest time at which the clock signal might reach any register

hashed lines: the clock might arrive up to t_{skew} earlier



Consider the setup time constraint.

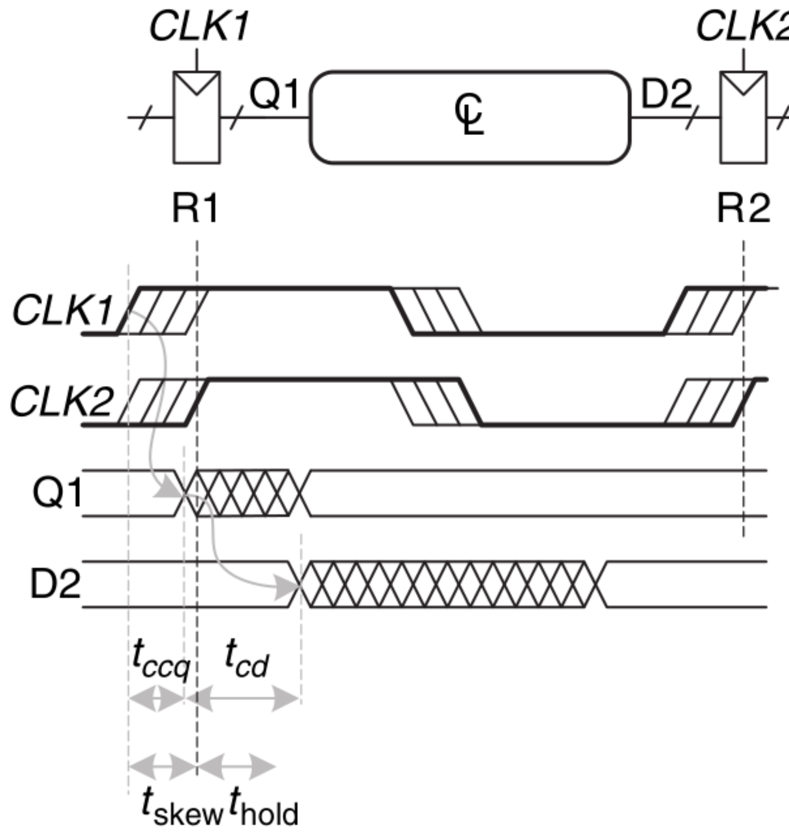
In the worst case, R1 receives the latest skewed clock and R2 receives the earliest skewed clock.

The data propagates through the register and combinational logic and must setup before R2 samples it.

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

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

Clock skew effectively increases the setup time



Consider the hold time constraint.

In the worst case, R1 receives an early skewed clock, and R2 receives a late skewed clock.

The data zips through the register and combinational logic but must not arrive until a hold time after the late clock.

$$t_{ccq} + t_{cd} \geq t_{hold} + t_{skew}$$

$$t_{cd} \geq t_{hold} + t_{skew} - t_{ccq}$$

Clock skew effectively increases the hold time

Clock skew effectively increases both the setup time and the hold time.

It reduces the time available for useful work in the combinational logic.

It also increases the required minimum delay through the combinational logic.

Even if $t_{hold} = 0$

a pair of back-to-back flip-flops will violate

$$t_{cd} \geq t_{hold} + t_{skew} - t_{ccq} \qquad \text{if} \qquad t_{cd} < t_{skew} - t_{ccq}$$

To prevent serious hold time failures, designers must not permit too much clock skew.

Clock fan-out

It is important that the clock circuit's output has sufficient fan-out capability to drive the necessary number of ICs requiring a clock input.

It is also important that the clock signal is not degraded in amplitude, speed of its rise and fall times or accuracy of its frequency.

The waveform should be kept as close as possible to a perfect square wave shape.

These qualities may degrade as the number of flip-flop increases.

Circuit capacitance

Because the clock must feed many gates, the small capacitance of each of these gates will add, to become a large capacitance, which loads the clock output tending to slow the rise and fall time of the clock signal.

To avoid this, the clock output must have a low enough impedance to rapidly charge and discharge any natural capacitance in the circuit.

The usual way to achieve this is to feed the clock signal via a special clock buffer gate, which will have the necessary low output impedance and a large fan out factor.

Cross-talk

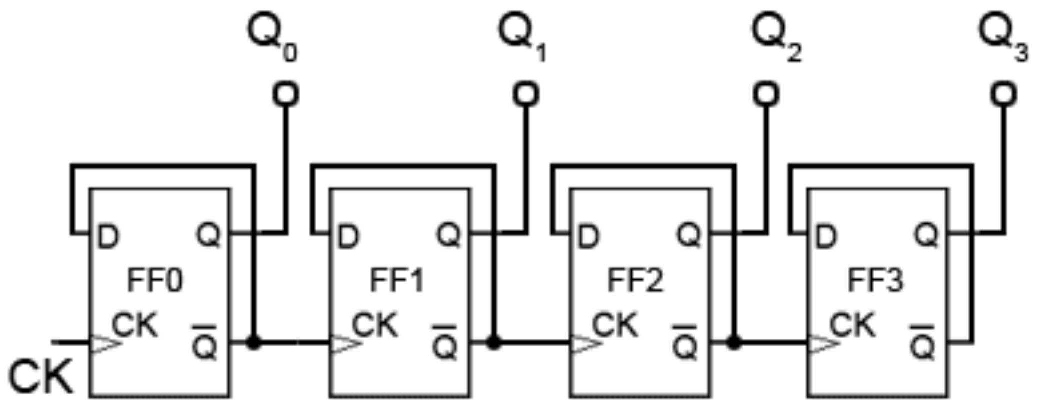
Where the clock signal has to be distributed around large circuits, there is a greater chance of introducing noise, and possible **cross-talk** where data in one conductor is radiated into another nearby conductor.

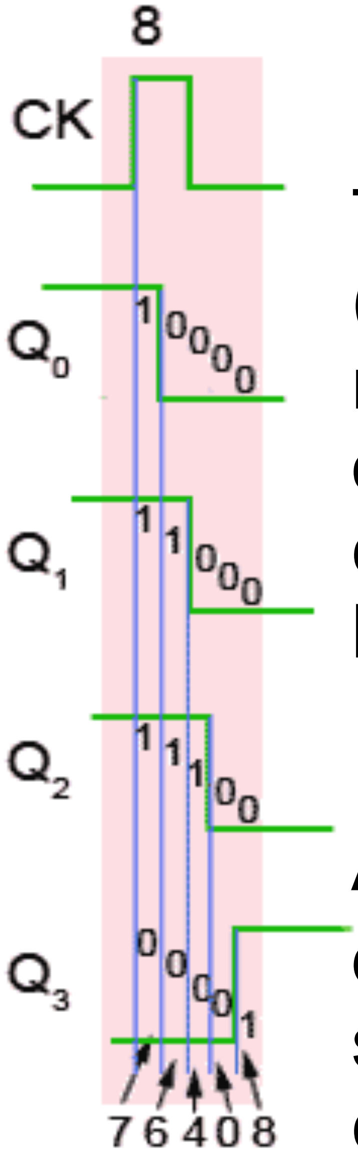
Problems such as this will increase the likelihood of skew errors, due to small changes in the phase of some of the distributed clock signals.

Miniaturisation brought about by surface mount technology can help minimise these problems.

Clock ripple

– in asynchronous counters

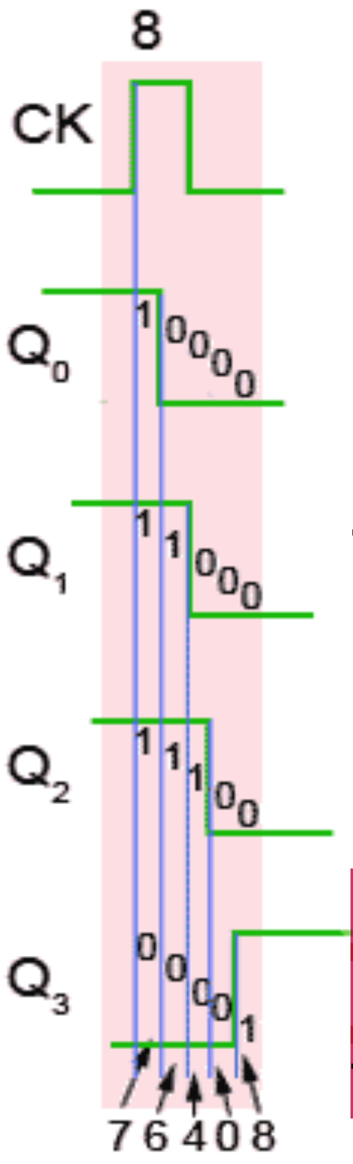




The propagation delays in each flip-flop (indicated by the blue vertical lines) add, over a number of flip-flops, to form a significant amount of delay between the time at which the output changes at the first flip flop (the least significant bit), and the last flip flop (the most significant bit).

As the Q_0 to Q_3 outputs each change at different times, a number of different output states occur as any particular clock pulse causes a new value to appear at the outputs.

At CK pulse 8 for example, the outputs Q0 to Q3 should change from 7 to 8, however what really happens is that the outputs change in the following sequence:



7, 6, 4, 0, 8

Unexpected values appear at the Q outputs for a very short time.

This problem prevents the circuit being used as a reliable counter.

Synchronous counters don't have this problem because the clock pulses are fed to every flip-flop in the chain at exactly the same time.

Synchronous versus asynchronous counters

Advantage of synchronous counters:

they don't have the timing ripple problem

Disadvantages of synchronous counters:

1) With every stage operating at very high clock frequencies, stray capacitive coupling between the counter and other components and within the counter itself is more likely occur, so that interference can be transferred between different stages of the counter, upsetting the count if adequate decoupling is not provided.

2) Because the clock pulses must charge, and discharge the input capacitance of every flip-flop simultaneously; synchronous counters having many flip-flops will cause large pulses of charge and discharge current in the clock driver circuits every time the clock changes logic state.

This can cause unwelcome spikes on the supply lines that could cause problems elsewhere in the digital circuitry.

Advantages of asynchronous counters:

1) Only the first flip-flop runs at the clock frequency.

Each subsequent flip-flop runs at half the frequency of the previous one.

2) The clock is only driving the first flip-flop in the counter chain → large pulses of charge and discharge current in the clock driver circuits.

Asynchronous counters are mostly used for frequency division applications and for generating time delays.

In either of these applications the timing of individual outputs is not likely to cause a problem to external circuitry.

The fact that most of the stages in the counter run at much lower frequencies than the input clock, greatly reduces any problem of high frequency noise interference to surrounding components.

Metastability

It is not always possible to guarantee that the input to a sequential circuit is stable during the aperture time.

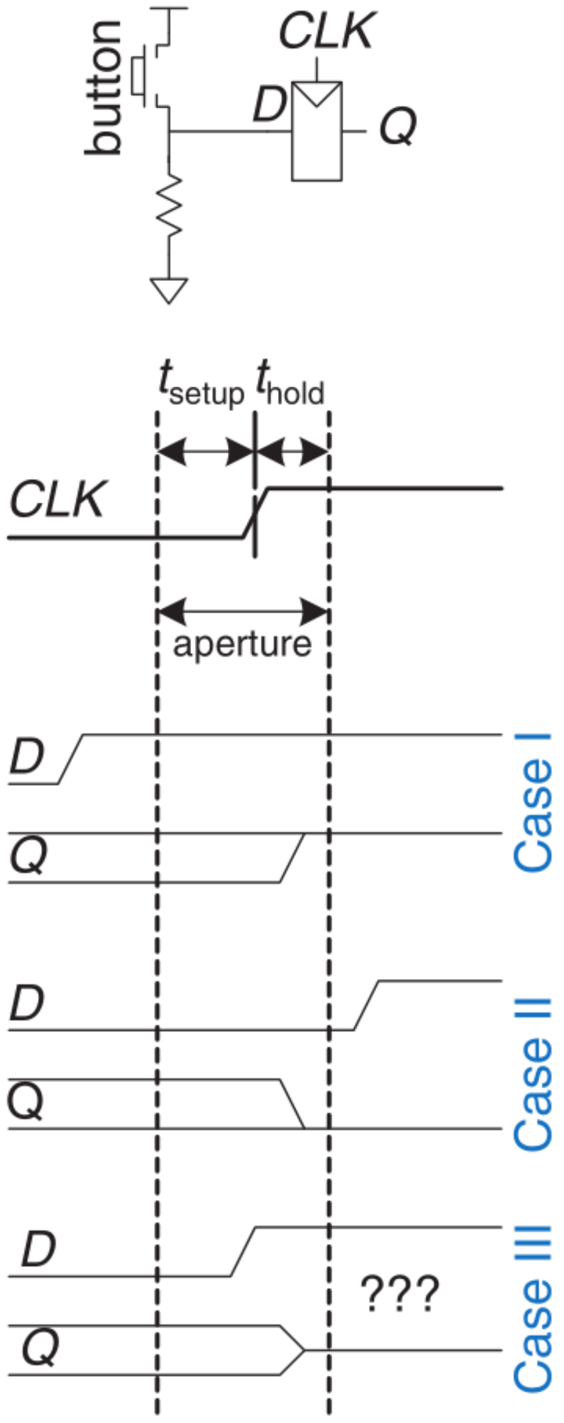
E.g. when the input arrives from the external world.

Example: a button connected to the input of a flip-flop.

When the button is not pressed, $D = 0$

When the button is pressed, $D = 1$

The button is pressed at some random time relative to the rising edge of *CLK*.



We want to know the output Q after the rising edge of CLK.

Case I:

The button is pressed much before CLK, Q = 1

Case II:

The button is not pressed until long after CLK, Q = 0

Case III:

The button is pressed sometime between t_{setup} before CLK and t_{hold} after CLK

The input violates the dynamic discipline and the output is undefined.

Metastable state

When a flip-flop samples an input that is changing during its aperture, the output Q may momentarily take on a voltage between 0 and V_{DD} that is in the forbidden zone.

This is called a **metastable state**

Eventually, the flip-flop will resolve the output to a stable state of either 0 or 1.

However, the resolution time required to reach the stable state is unbounded.

Resolution time

If a flip-flop input changes at a random time during the clock cycle, the **resolution time**,

$$t_{res}$$

– the time required to resolve to a stable state is a random variable.

If the input changes outside the aperture, then $t_{res} = t_{pcq}$

But if the input happens to change within the aperture, the resolution time can be much longer.

The probability that the resolution time exceeds some arbitrary time, *t* is

$$P(t_{res} > t) = \frac{T_0}{T_c} e^{-t/\tau}$$

T_c – the clock period

T_0, τ – characteristic of the flip-flop

The equation is valid only for $t_{res} \gg t_{pcq}$

T_0/T_c describes the probability that the input changes during the aperture time

τ is a time constant indicating how fast the flip-flop moves away from the metastable state.

It is related to the delay through the cross-coupled gates in the flip-flop.

Synchronizers

Inputs to digital systems from the real world are often asynchronous.

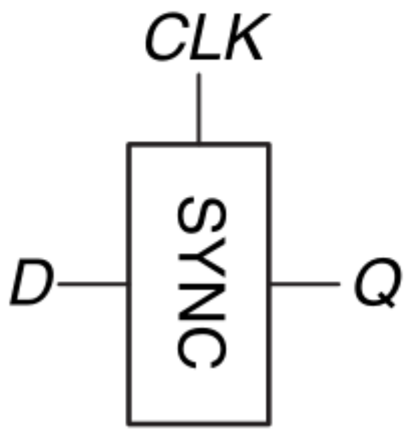
If handled carelessly, they can lead to metastable voltages within the system.

This may cause erratic system failures that are difficult to track down and correct.

The goal of a digital system designer should be to ensure that, given asynchronous inputs, the probability of encountering a metastable voltage is sufficiently small.

To guarantee good logic levels, all asynchronous inputs should be passed through **synchronizers**.

Synchronizers



A **synchronizer** receives an asynchronous input *D* and a clock *CLK*.

It produces an output Q within a bounded amount of time.

The output has a valid logic level with extremely high probability.

If D is stable during the aperture, Q should take on the same value as D.

If D changes during the aperture, Q may take on either a HIGH or LOW value but must not be metastable.

The speed of a system is characterized by the **latency** and **throughput** of information moving through it.

A **token** is a group of inputs that are processed to produce a group of outputs.

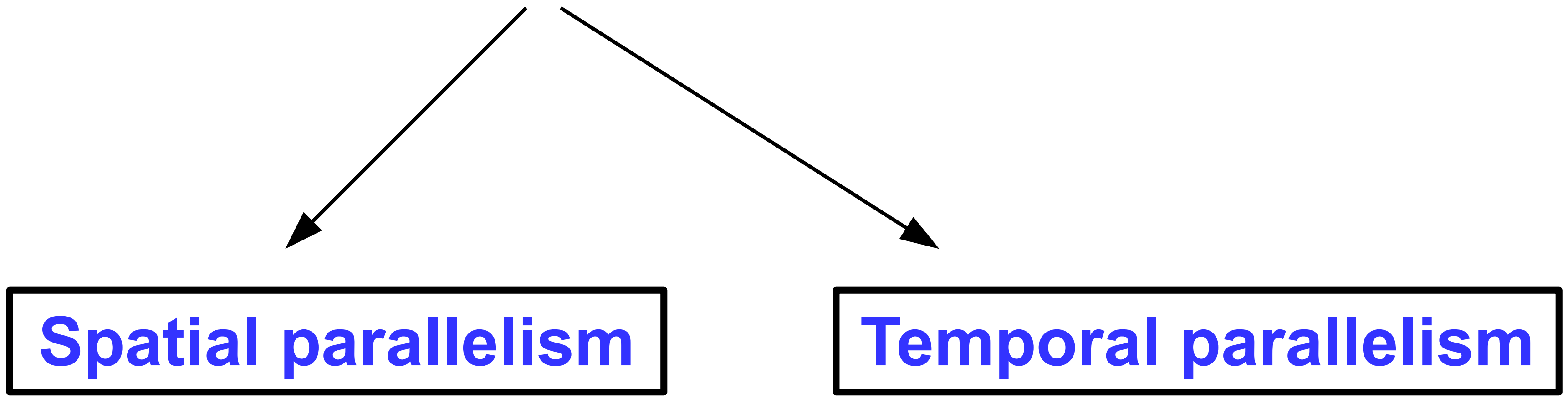
The **latency** of a system is the time required for one token to pass through the system from start to end.

The **throughput** is the number of tokens that can be produced per unit time.

Parallelism

The throughput can be improved by processing several tokens at the same time.

This is called **parallelism**



With **spatial parallelism**, multiple copies of the hardware are provided so that multiple tasks can be done at the same time.

With **temporal parallelism**, a task is broken into stages, like an assembly line.

Multiple tasks can be spread across the stages.

Although each task must pass through all stages, a different task will be in each stage at any given time so multiple tasks can overlap.

Temporal parallelism is also called **pipelining**.

Consider a task with latency L

without parallelism, the throughput is $1/L$

spatially parallel system with N copies of the hardware:
the throughput is N/L

temporally parallel system:
the task is ideally broken into N stages, of equal length

In such a case, the throughput is also N/L, and only one copy of the hardware is required.

However, finding N steps of equal length is often impractical.

If the longest step has a latency L_1, the pipelined throughput is $1/L_1$

Temporal parallelism may speed up a circuit without duplicating the hardware.

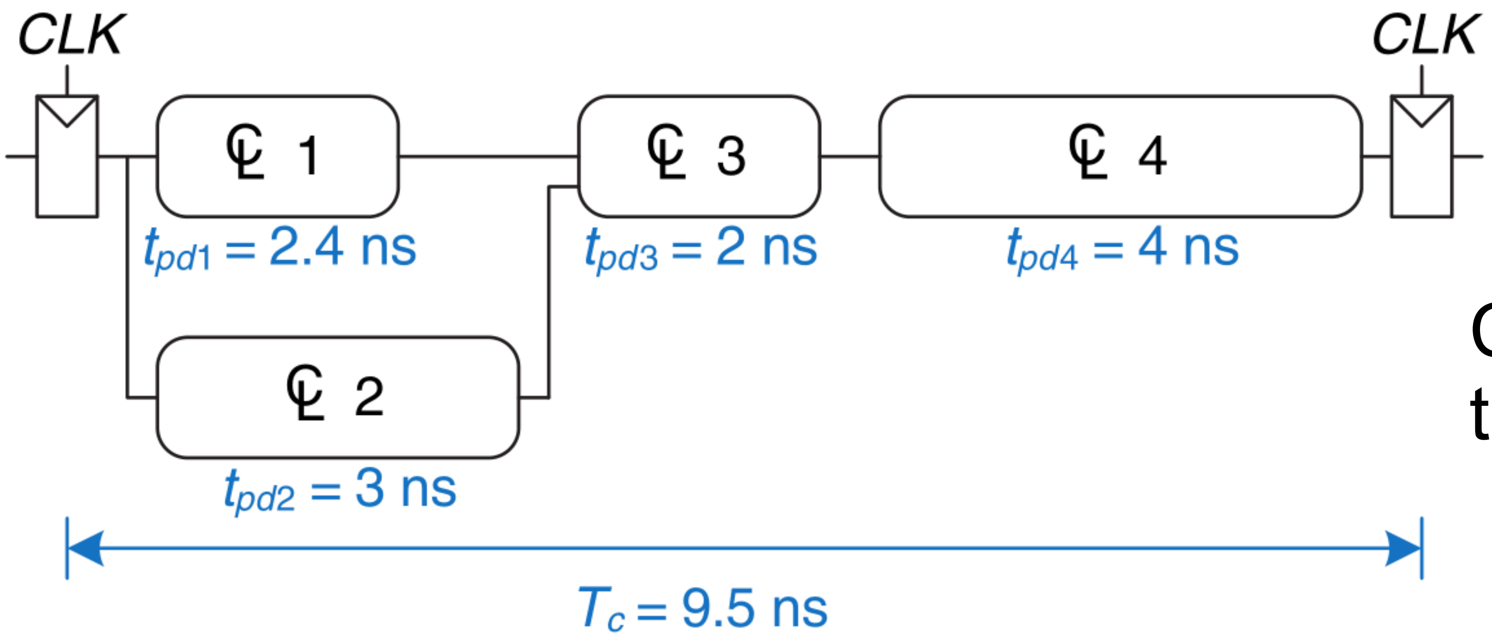
Instead, registers are placed between blocks of combinational logic to divide the logic into shorter stages that can run with a faster clock.

The registers prevent a token in one pipeline stage from catching up with and corrupting the token in the next stage.

Circuit with no pipelining

$t_{pcq} = 0.3\,ns$

$t_{setup} = 0.2\,ns$



Critical path: through 2, 3, 4

$$T_c = 0.3 + 3 + 2 + 4 + 0.2 = 9.5\,ns$$

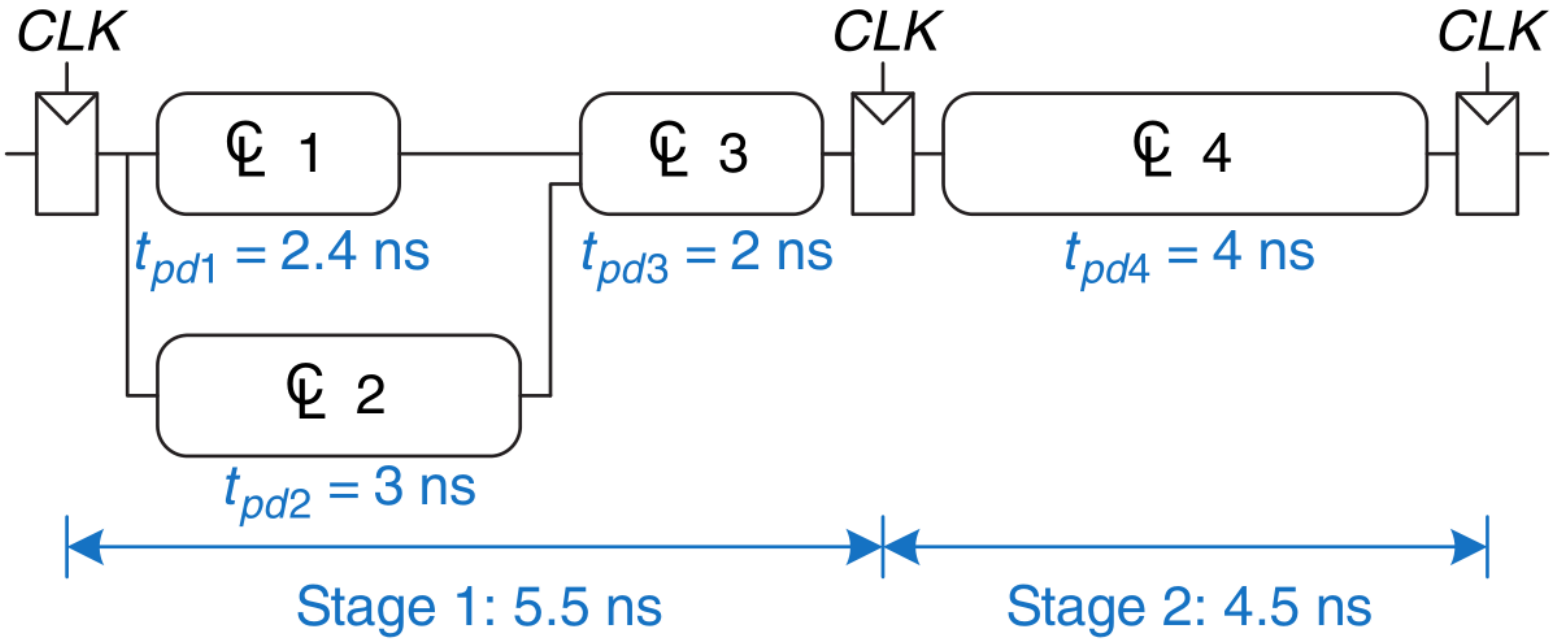
Latency: $9.5\,ns$

Throughput: $1/9.5\,ns = 105\,MHz$

Circuit with two-stage pipeline

$t_{pcq} = 0.3\,ns$

$t_{setup} = 0.2\,ns$



Min clock period for the 1st stage: $0.3 + 3 + 2 + 0.2 = 5.5\,ns$

Min clock period for the 2nd stage: $0.3 + 4 + 0.2 = 4.5\,ns$

The clock must be slow enough for all stages to work

$$T_c = 5.5\,ns$$

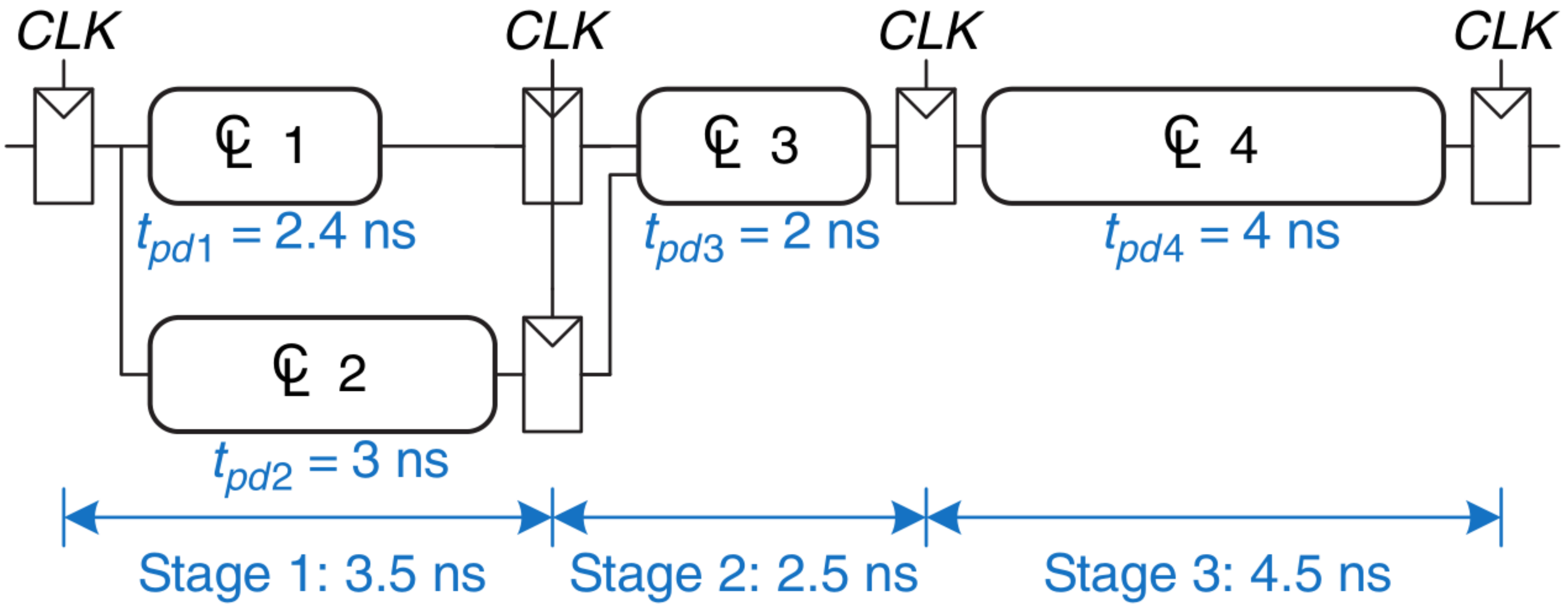
Latency: $11\,ns$ (two clock cycles)

Throughput: $1/5.5\,ns = 182\,MHz$

Circuit with three-stage pipeline

$t_{pcq} = 0.3\,ns$

$t_{setup} = 0.2\,ns$



The cycle time is limited bythe 3rd stage $T_c = 4.5\,ns$

Latency: $13.5\,ns$ (three cycles)

Throughput: $1/4.5\,ns = 222\,MHz$

Adding pipeline stages improves throughput at the expense of some latency.