Hardware Modeling [VU] (191.011)
– WS24 –
Synchronous Design Style

Guest Lecture by Prof. Steininger

WS 2024/25

In this guest lecture you will be introduced to the synchronous design style. As we will see, this style revolves around sequential circuit elements and a global clock signal to synchronize them. The lecture will motivate why we need such synchronization.

Hardware Modeling [VU] (191.011)
– WS24 –
Synchronous Design Style

Guest Lecture by Prof. Steininger

WS 2024/25

Modified: 2025-03-12, 16:33 (21636bb)

Logic gates are the elementary blocks of a digital circuit (e.g. AND, OR, XOR)

Each digital circuit, not mattering how complex it is, is composed of logic gates. In fact, these gates are internally built from transistors, but as digital designers we abstract this detail away and consider gates as the atomic building blocks available to us. Well known examples of such gates are the AND, OR and XOR gate.

Combinational Logic Gates

- Logic gates are the elementary blocks of a digital circuit (e.g. AND, OR, XOR)

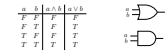
- Logic gates are the elementary blocks of a digital circuit (e.g. AND, OR, XOR)
- Such gates without memory are called **combinational**
  - Their outputs only depend on their inputs (c.f. mathematical function like $sin(x)$)

We call a gate combinational if its output is fully determined by its inputs alone, just as for mathematical functions like the sine function or pure functions in VHDL. This is the case for gates that have no internal memory.

Combinational Logic Gates

- Logic gates are the elementary blocks of a digital circuit (e.g. AND, OR, XOR)
- Such gates without memory are called **combinational**
  - Their outputs only depend on their inputs (c.f. mathematical function like $sin(x)$)

- Logic gates are the elementary blocks of a digital circuit (e.g. AND, OR, XOR)
- Such gates without memory are called **combinational**
  - Their outputs only depend on their inputs (c.f. mathematical function like $sin(x)$)
- We can express their functionality using a *truth table*
  - Enumerate all inputs and write down output

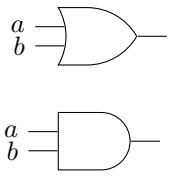


As you certainly know, the function of a gate can be expressed through a truth table, where for each combination of input values the corresponding output is given. The slides illustrate this for the two-input AND gate, as well as for the two-input OR gate.

Combinational Logic Gates

- Logic gates are the elementary blocks of a digital circuit (e.g. AND, OR, XOR)
- Such gates without memory are called **combinational**
  - Their outputs only depend on their inputs (c.f. mathematical function like $sin(x)$)
- We can express their functionality using a *truth table*
  - Enumerate all inputs and write down output

a	b	$a \wedge b$	$a \vee b$
F	F	F	F
F	T	F	T
T	F	F	T
T	T	T	T



Gates with a memory are called **sequential**
Output depends on inputs and **previous state**
⇒ Expressed via truth table containing previous state or state diagram

As soon as a gate contains a memory, it has a state that is determined by the value stored in that memory. Now the gate's output for a certain combination of input values may also be dependent on its state. In the truth table this can be expressed by treating the current state as an input for calculating the next output. We will see an example shortly. Furthermore, you probably know that state diagrams, as used for automata, can alternatively be used to describe the function of sequential gates.

Sequential Logic Gates

- Gates with a memory are called **sequential**
  - Output depends on inputs and **previous state**

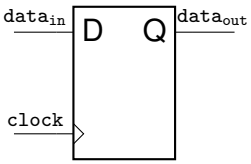
  ⇒ Expressed via truth table containing previous state or state diagram

- Gates with a memory are called **sequential**
  - Output depends on inputs and **previous state**
  - ⇒ Expressed via truth table containing previous state or state diagram
- Prominent example: **flip-flop**
  - At each rising edge of the clock (CLK) the input data (D) is copied to the output (Q)
  - Between rising clock edges the output is stable

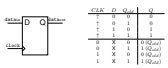
A very important sequential gate is the flip-flop. In its minimalistic form it has two inputs, namely clock and data, and one output. Its function is to copy the value from its data input to its output whenever it sees a rising edge at its clock input. This seemingly simple function has a conceptually fundamental detail, as the operation is controlled by a transition, an not just by a static logic value. On the slide you can see the symbol for such a flip-flop, where the output is called ”Q”. The triangle at the clock input highlights that the flip-flop is sensitive to rising edges of this input.

Sequential Logic Gates

- Gates with a memory are called **sequential**
  - Output depends on inputs and **previous state**
  - ⇒ Expressed via truth table containing previous state or state diagram
- Prominent example: **flip-flop**
  - At each rising edge of the clock (CLK) the input data (D) is copied to the output (Q)
  - Between rising clock edges the output is stable



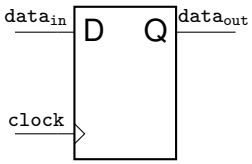
- Gates with a memory are called **sequential**
  - Output depends on inputs and **previous state**
  - ⇒ Expressed via truth table containing previous state or state diagram
- Prominent example: **flip-flop**
  - At each rising edge of the clock (CLK) the input data (D) is copied to the output (Q)
  - Between rising clock edges the output is stable



Next to the flip-flop symbol you can now see its truth table. The arrow in the clock column stands for a rising edge and the X in the input data column for an arbitrary value. Observe how a rising clock edge always results in the input data being copied to Q and how the flip-flop will otherwise keep its output stable.

Sequential Logic Gates

- Gates with a memory are called **sequential**
  - Output depends on inputs and **previous state**
  - ⇒ Expressed via truth table containing previous state or state diagram
- Prominent example: **flip-flop**
  - At each rising edge of the clock (CLK) the input data (D) is copied to the output (Q)
  - Between rising clock edges the output is stable



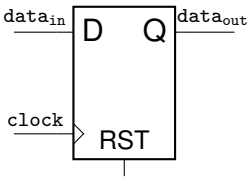
CLK	D	Q_{old}	Q
↑	0	0	0
↑	0	1	0
↑	1	0	1
↑	1	1	1
0	X	0	$0\ (Q_{old})$
0	X	1	$1\ (Q_{old})$
1	X	0	$0\ (Q_{old})$
1	X	1	$1\ (Q_{old})$

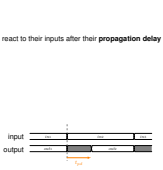
- Gates with a memory are called **sequential**
  - Output depends on inputs and **previous state**
  - ⇒ Expressed via truth table containing previous state or state diagram
- Prominent example: **flip-flop**
  - At each rising edge of the clock (CLK) the input data (D) is copied to the output (Q)
  - Between rising clock edges the output is stable
- Optional: (synchronous or asynchronous) reset input (RST)

In addition, there may be a reset input to force the output to a specific value, independent of the data input. If this forcing is bound to the occurrence of a rising clock edge, we call this a synchronous reset. If it happens instantly, without respecting the clock, we call it an asynchronous reset. Both versions have their use in practice.

Sequential Logic Gates

- Gates with a memory are called **sequential**
  - Output depends on inputs and **previous state**
  - ⇒ Expressed via truth table containing previous state or state diagram
- Prominent example: **flip-flop**
  - At each rising edge of the clock (CLK) the input data (D) is copied to the output (Q)
  - Between rising clock edges the output is stable
- Optional: (synchronous or asynchronous) reset input (RST)

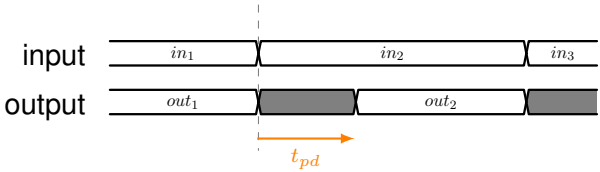


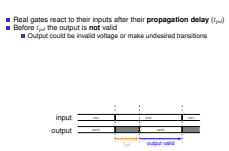


In a truth table it looks like an input change leads to an immediate output reaction. However, there is no truly immediate reaction in real life. A logic gate needs some time, the so-called propagation delay, to update its output after an input change. The image on the slide illustrates this via the dark gray output value which stands for invalid. Only after the propagation time the output corresponding to the second input is provided by the gate. This has important consequences.

Timing Conditions for Proper Operation of Gates

- Real gates react to their inputs after their **propagation delay** (t_{pd})

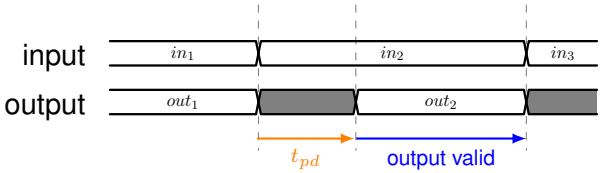


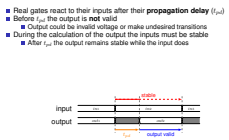


After we change an input, we need to give the output time to stabilize. If we use the result too early, we may see unwanted transitions or incorrect values.

Timing Conditions for Proper Operation of Gates

- Real gates react to their inputs after their **propagation delay** (t_{pd})
- Before t_{pd} the output is **not** valid
  - Output could be invalid voltage or make undesired transitions

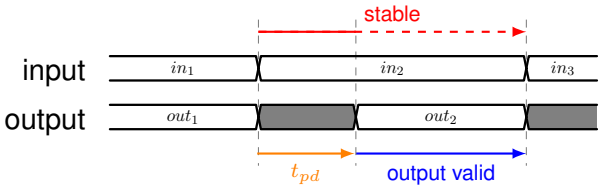


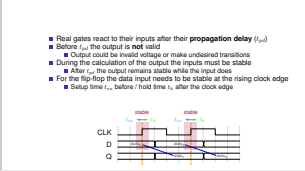


Also, in order to keep the output stable, we need to keep all inputs stable until the result is complete. Otherwise, we get an incomplete computation which is obviously not useful. After the propagation delay, the output stays valid until the input changes again. In the figure the red arrow atop the input illustrates this. It is drawn solid during the time where the input must be stable and drawn dashed afterwards as from now on the input is can change at an arbitrary time without corrupting the computation of the output.

Timing Conditions for Proper Operation of Gates

- Real gates react to their inputs after their **propagation delay** (t_{pd})
- Before t_{pd} the output is **not** valid
  - Output could be invalid voltage or make undesired transitions
- During the calculation of the output the inputs must be stable
  - After t_{pd} the output remains stable while the input does

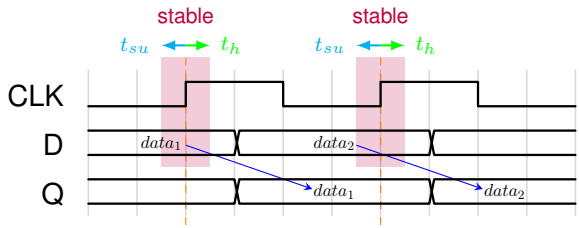


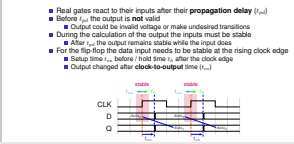


For the flip flop it must be clear which logic value to copy to the output with a rising clock edge. To ensure this, data must not change shortly before or after the rising clock edge. This critical interval depends on a time where data must be stable before a clock edge, the so-called setup-time, and a time after the clock edge, called hold-time. The respective interval around the clock edge is called the setup and hold window.

Timing Conditions for Proper Operation of Gates

- Real gates react to their inputs after their **propagation delay** (t_{pd})
- Before t_{pd} the output is **not** valid
  - Output could be invalid voltage or make undesired transitions
- During the calculation of the output the inputs must be stable
  - After t_{pd} the output remains stable while the input does
- For the flip-flop the data input needs to be stable at the rising clock edge
  - Setup time t_{su} before / hold time t_h after the clock edge

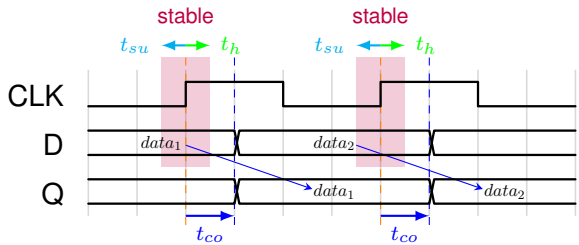




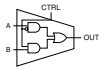
Furthermore, it takes some time for a flip flop to copy the input value to the output. This time is the clock to output time.

Timing Conditions for Proper Operation of Gates

- Real gates react to their inputs after their **propagation delay** (t_{pd})
- Before t_{pd} the output is **not** valid
  - Output could be invalid voltage or make undesired transitions
- During the calculation of the output the inputs must be stable
  - After t_{pd} the output remains stable while the input does
- For the flip-flop the data input needs to be stable at the rising clock edge
  - Setup time t_{su} before / hold time t_h after the clock edge
  - Output changed after **clock-to-output** time (t_{co})



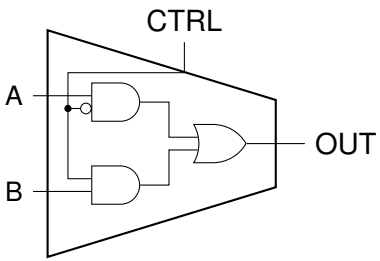
Larger functions are composed of many simple gates



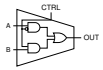
By appropriate composition of gates larger and more complex functions can be realized. For example, the slide shows the gate-level implementation of a multiplexer circuit. You should already familiar with this circuit where depending on the control input either the input ”A” or ”B” is forward to the output.

Building Functions from Gates

- Larger functions are composed of many simple gates



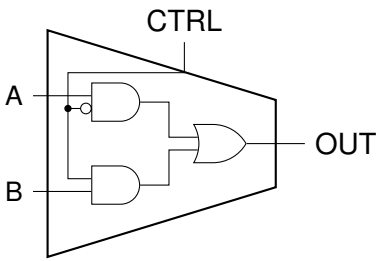
- Larger functions are composed of many simple gates
  - Gates operate concurrently
  - Each gate has an individual delay
  - Some gates will provide inputs for others



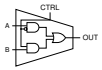
In such a larger function all gates operate in parallel and independent of each other. The only coupling between gates happens when one gate is providing the input for another. This enormous parallelism gives hardware its high performance but also makes it hard to design it.

Building Functions from Gates

- Larger functions are composed of many simple gates
  - Gates operate concurrently
  - Each gate has an individual delay
  - Some gates will provide inputs for others



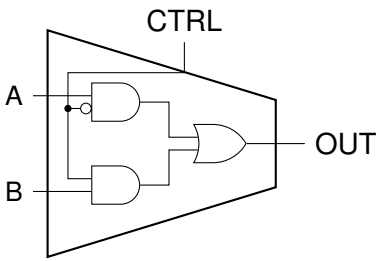
- Larger functions are composed of many simple gates
  - Gates operate concurrently
  - Each gate has an individual delay
  - Some gates will provide inputs for others
- How to ensure proper operation?



But how can we guarantee proper operating conditions for such a receiving gate under these circumstances? The gates providing its individual inputs may have different propagation delays, and they may even have received their respective inputs at different points in time. And how does the gate know when an individual input is stable at all?

Building Functions from Gates

- Larger functions are composed of many simple gates
  - Gates operate concurrently
  - Each gate has an individual delay
  - Some gates will provide inputs for others
- How to ensure proper operation?



- Larger functions are composed of many simple gates
  - Gates operate concurrently
  - Each gate has an individual delay
  - Some gates will provide inputs for others
- How to ensure proper operation?
- ⇒ Requires coordination!

Obviously we need to somehow coordinate these activities.

Building Functions from Gates

- Larger functions are composed of many simple gates
  - Gates operate concurrently
  - Each gate has an individual delay
  - Some gates will provide inputs for others
- How to ensure proper operation?

⇒ Requires coordination!

To better understand the problem and maybe get an idea for a solution principle, let us look at a real-world example: In an orchestra we have individual musicians whose activities need to be coordinated. How is that achieved? Well, we all know that there is a conductor who takes care of this. But what is the abstract principle? Can we use this principle when designing circuits and coordinating gates?

Coordination in Real Life

There is no global notion of time ⇒ the conductor introduces one

The general problem is that there is no notion of time shared by the musicians. This prevents them from coordinating the play themselves as they do not know when other musicians will start or stop playing and when the time has come where they should play themselves. This is where the conductor comes in. In fact, with their batons that are visible for all, a conductor provides a global notion of time, namely the beat.

The Orchestra's Coordination Principle

- There is no global notion of time ⇒ the conductor introduces one

There is no global notion of time ⇒ the conductor introduces one
Each musician knows their specific schedule

Each musician knows, for their specific instrument, how to time their activities relative to this time base. They do not need to communicate with others, not even know about their schedule. They can simply know when and what to play with based on this global beat.

The Orchestra's Coordination Principle

- There is no global notion of time ⇒ the conductor introduces one
- Each musician knows their specific schedule

There is no global notion of time ⇒ the conductor introduces one
Each musician knows their specific schedule
A global plan ensures the desired result as a sum of all activities

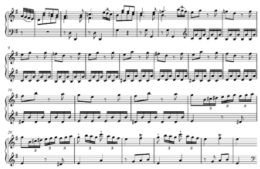


The brilliant thing now is that the composer has puzzled out a global plan that makes sure that the contributions of the individual instruments sum up to a marvelous symphony if they all adhere to the conductor.

The Orchestra's Coordination Principle

- There is no global notion of time ⇒ the conductor introduces one
- Each musician knows their specific schedule
- A global plan ensures the desired result as a sum of all activities



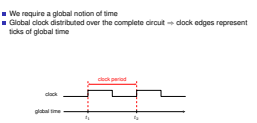


We require a global notion of time

Can we apply this principle to our circuit problem? In fact, the musicians correspond to the logic gates, and the circuit designer takes the role of the ingenious composer. Thus, the only thing missing is a pendant to the conductor that introduces a global notion of time.

Coordination in Synchronous Logic

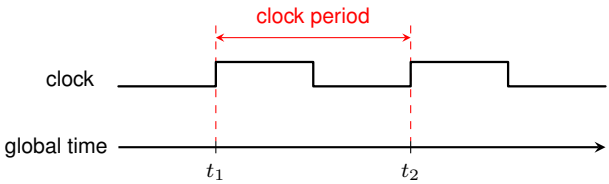
- We require a global notion of time

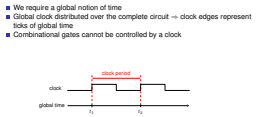


In a synchronous circuit the idea is to introduce a clock generator whose output is distributed over the whole circuit to establish a global time base. This clock corresponds to the role of the conductor in an orchestra. However, instead making the progress of time visible to all musicians via a moving baton, we use the edges of the clock signal when designing a synchronous circuit.

Coordination in Synchronous Logic

- We require a global notion of time
- Global clock distributed over the complete circuit ⇒ clock edges represent ticks of global time

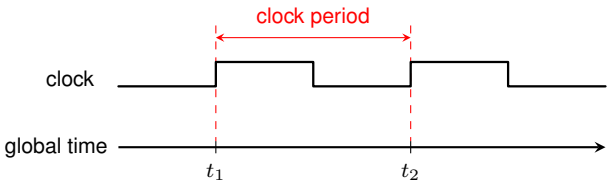


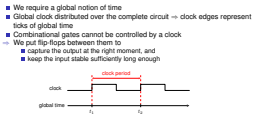


On the slide you can see an illustration of how the clock can be used to introduce a notion of time. Typically, the clock is an oscillating signal with rectangular shape. The time between two rising clock edges is the period of the clock signal. By knowing this period, we can relate the progression of time to the occurrence of clock edges. In this illustration we can see that each rising edge is associated with a global time.

Coordination in Synchronous Logic

- We require a global notion of time
- Global clock distributed over the complete circuit ⇒ clock edges represent ticks of global time
- Combinational gates cannot be controlled by a clock



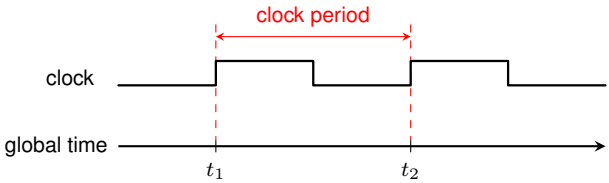


However, the combinational logic gates that implement our circuit's logic function operate continuously and do not respect any triggering by a clock. At this point the flip-flop comes to the rescue: Recall that its function is to copy it data input to the output upon a rising clock edge. This is exactly what we need.

Coordination in Synchronous Logic

- We require a global notion of time
- Global clock distributed over the complete circuit ⇒ clock edges represent ticks of global time
- Combinational gates cannot be controlled by a clock

⇒ We put flip-flops between them to
  - capture the output at the right moment, and
  - keep the input stable sufficiently long enough



- We require a global notion of time
- Global clock distributed over the complete circuit ⇒ clock edges represent ticks of global time
- Combinational gates cannot be controlled by a clock

⇒ We put flip-flops between them to
  - capture the output at the right moment, and
  - keep the input stable sufficiently long enough

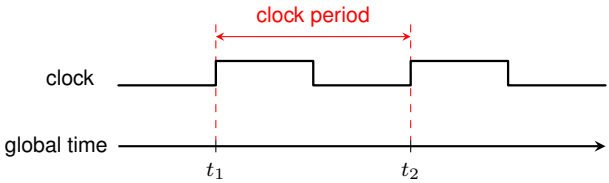
clock period
clock
global time
t_1
t_2

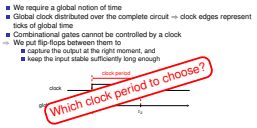
So at some well-chosen places within our circuit we introduce flip-flops into the data flow. They can provide the two essential functions we need for proper gate operation: Capture the output after it stabilized and keep the input stable for most of the time.

Coordination in Synchronous Logic

- We require a global notion of time
- Global clock distributed over the complete circuit ⇒ clock edges represent ticks of global time
- Combinational gates cannot be controlled by a clock

⇒ We put flip-flops between them to
  - capture the output at the right moment, and
  - keep the input stable sufficiently long enough



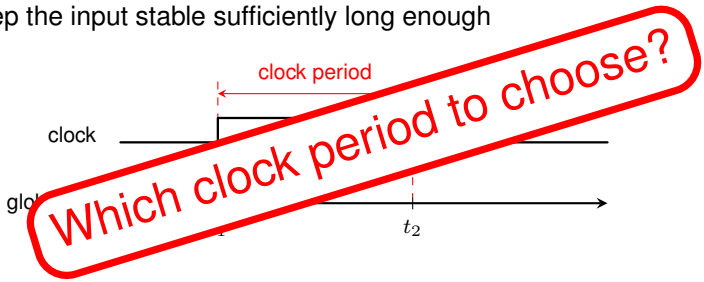


Naturally this only works if we choose an appropriate clock. However – when is a clock appropriate? Can the clock period be arbitrary?

Coordination in Synchronous Logic

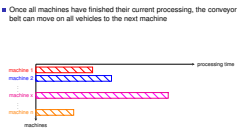
- We require a global notion of time
- Global clock distributed over the complete circuit ⇒ clock edges represent ticks of global time
- Combinational gates cannot be controlled by a clock

⇒ We put flip-flops between them to
- capture the output at the right moment, and
- keep the input stable sufficiently long enough



To find an answer, let us again look into a real-world example. This time we use an assembly line for cars. It comprises a number of stations, where a machine does some specific processing on the chassis of the car currently in front of it. All machines do their processing in parallel, each on a different car. When that is finished, the belt moves the cars forward, each to the respective next machine for the next processing step. For the time between two steps we need to consider that all machines do their processing in parallel and we cannot move the belt forward before the longest processing step has finished. Otherwise, we would propagate car chassis that are not yet ready for subsequent steps. This will introduce errors in the final car, maybe amplified by other stages not being able to correctly perform their task due to the chassis being in the wrong state.

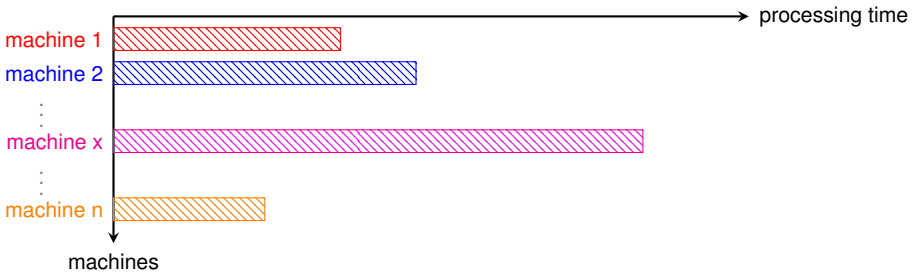
Assembly Line Optimization

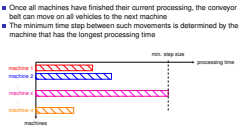


As a consequence, the conveyor belt must only move once all machines are done with their processing steps. With all vehicles being moved at once by the conveyor belt, we must obviously always wait for the slowest processing step. The image on the slide illustrates this. For all ”n” machines participating in the assembly line we have plotted the processing time they require.

Timing the Assembly Line

- Once all machines have finished their current processing, the conveyor belt can move on all vehicles to the next machine

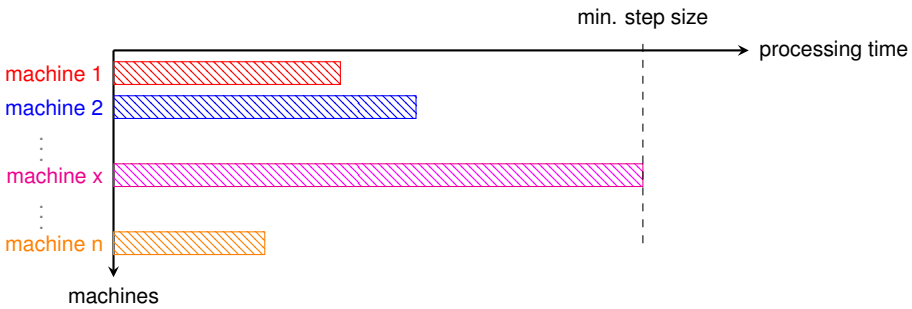




One processing time will be greater or equal to all other machines' times. This is the minimal step size of the conveyor belt. This longest processing time is the minimum interval between two conveyor belt movements. The pace for moving the belt must consider that. Of course, it can move slower than that, but never faster without compromising the whole assembly line. In our example the slowest one is machine "X".

Timing the Assembly Line

- Once all machines have finished their current processing, the conveyor belt can move on all vehicles to the next machine
- The minimum time step between such movements is determined by the machine that has the longest processing time



In a synchronous design the circuit is partitioned into blocks through the insertion of flip-flops

How can we apply this insight to our problem? By inserting flip-flops as mentioned before, we have partitioned the logic into smaller blocks. These flip-flops take the role of the conveyor belt: They capture an output and convey it to the next stage. Between each two flip-flops we have a combinational logic block that performs a function on the output of one flip-flop and provides its result to the input of the next flip-flop. This combinational logic thus takes the role of the machines along the conveyor belt with the propagation delay of each combinational function corresponding to the processing time of the machines.

Static Timing Analysis (STA)

- In a synchronous design the circuit is partitioned into blocks through the insertion of flip-flops

- In a synchronous design the circuit is partitioned into blocks through the insertion of flip-flops
- To identify the minimum clock period we use **static timing analysis** (STA)

To determine the pace for moving, which is the clock frequency in our case, we need to make sure all computations are finished before moving on. This means that we have to determine the propagation of each combinational function and to ensure that the clock period is sufficiently high. Otherwise, invalid outputs might be forwarded by the flip-flops, just as with the unfinished chassis on the conveyor belt. This is the purpose of the so-called static timing analysis.

Static Timing Analysis (STA)

- In a synchronous design the circuit is partitioned into blocks through the insertion of flip-flops
- To identify the minimum clock period we use **static timing analysis** (STA)

- In a synchronous design the circuit is partitioned into blocks through the insertion of flip-flops
- To identify the minimum clock period we use **static timing analysis** (STA)
  - Determine the signal delays through each block (take the slowest)
  - The longest of all such block delays is the **critical path**

First we determine the longest propagation path from any of its inputs to its output for each combinational logic block. Here, a propagation path is the sum of propagation delays of all the gates the respective signal is running through. Since we cannot change the clock-period on the fly in general, we have to ensure that the clock supports the longest possible path through the block, such that it will always finish with its computation within a clock period. This longest path is also called the critical path of the block. Next we search for the longest critical path among all blocks. It corresponds to the longest possible processing time of a machine along the conveyor belt.

Static Timing Analysis (STA)

- In a synchronous design the circuit is partitioned into blocks through the insertion of flip-flops
- To identify the minimum clock period we use **static timing analysis** (STA)
  - Determine the signal delays through each block (take the slowest)
  - The longest of all such block delays is the **critical path**

- In a synchronous design the circuit is partitioned into blocks through the insertion of flip-flops
- To identify the minimum clock period we use **static timing analysis** (STA)
  - Determine the signal delays through each block (take the slowest)
  - The longest of all such block delays is the **critical path**
  - After this delay even the slowest output is stable
  - We must also ensure stable flip-flop inputs around clock edges

After the delay of this critical path, even the slowest output in our circuit has stabilized. Recall that we previously also mentioned that the input data of a flip flop must be stable at the clock edge where it captures the data and that it takes some time after each clock edge until the new output data is provided. We therefore also have to consider the output delay of the flip-flop that feeds the critical path input, as well as the setup time of the flip-flop that captures the block output for the clock period limit.

Static Timing Analysis (STA)

- In a synchronous design the circuit is partitioned into blocks through the insertion of flip-flops
- To identify the minimum clock period we use **static timing analysis** (STA)
  - Determine the signal delays through each block (take the slowest)
  - The longest of all such block delays is the **critical path**
  - After this delay even the slowest output is stable
  - We must also ensure stable flip-flop inputs around clock edges

- In a synchronous design the circuit is partitioned into blocks through the insertion of flip-flops
- To identify the minimum clock period we use **static timing analysis** (STA)
  - Determine the signal delays through each block (take the slowest)
  - The longest of all such block delays is the **critical path**
  - After this delay even the slowest output is stable
  - We must also ensure stable flip-flop inputs around clock edges
- This critical path delay determines the minimum clock period

In conclusion, this critical path delay of our design constitutes the lower limit for our clock period. The clock period must not be lower than this delay to ensure safe operation of the circuit.

Static Timing Analysis (STA)

- In a synchronous design the circuit is partitioned into blocks through the insertion of flip-flops
- To identify the minimum clock period we use **static timing analysis** (STA)
  - Determine the signal delays through each block (take the slowest)
  - The longest of all such block delays is the **critical path**
  - After this delay even the slowest output is stable
  - We must also ensure stable flip-flop inputs around clock edges
- This critical path delay determines the minimum clock period

- In a synchronous design the circuit is partitioned into blocks through the insertion of flip-flops
- To identify the minimum clock period we use **static timing analysis** (STA)
  - Determine the signal delays through each block (take the slowest)
  - The longest of all such block delays is the **critical path**
  - After this delay even the slowest output is stable
  - We must also ensure stable flip-flop inputs around clock edges
- This critical path delay determines the minimum clock period
  - The maximum clock frequency is the inverse of this period

Often when referring to clocks the clock frequency rather than period is given. Converting between these two quantities happens via inversion. Note that the lower limit for the clock period can be converted into an upper limit for the clock frequency by inversion.

Static Timing Analysis (STA)

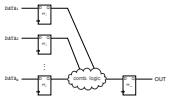
- In a synchronous design the circuit is partitioned into blocks through the insertion of flip-flops
- To identify the minimum clock period we use **static timing analysis** (STA)
  - Determine the signal delays through each block (take the slowest)
  - The longest of all such block delays is the **critical path**
  - After this delay even the slowest output is stable
  - We must also ensure stable flip-flop inputs around clock edges
- This critical path delay determines the minimum clock period
  - The maximum clock frequency is the inverse of this period

- In a synchronous design the circuit is partitioned into blocks through the insertion of flip-flops
- To identify the minimum clock period we use **static timing analysis** (STA)
  - Determine the signal delays through each block (take the slowest)
  - The longest of all such block delays is the **critical path**
  - After this delay even the slowest output is stable
  - We must also ensure stable flip-flop inputs around clock edges
- This critical path delay determines the minimum clock period
  - The maximum clock frequency is the inverse of this period
- For the best performance we choose our clock frequency close to the maximum from the STA ⇒ more in a later lecture

Naturally we want to approach the highest possible clock frequency, in order to obtain the best performance and thus want to achieve a small clock period. We will discuss methods to reduce the clock period in future lectures.

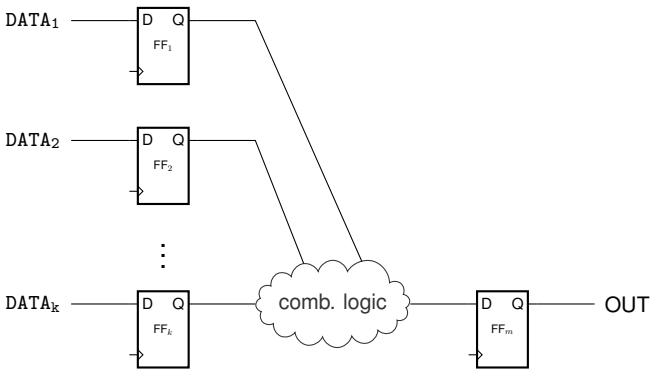
Static Timing Analysis (STA)

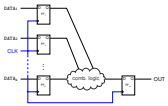
- In a synchronous design the circuit is partitioned into blocks through the insertion of flip-flops
- To identify the minimum clock period we use **static timing analysis** (STA)
  - Determine the signal delays through each block (take the slowest)
  - The longest of all such block delays is the **critical path**
  - After this delay even the slowest output is stable
  - We must also ensure stable flip-flop inputs around clock edges
- This critical path delay determines the minimum clock period
  - The maximum clock frequency is the inverse of this period
- For the best performance we choose our clock frequency close to the maximum from the STA ⇒ more in a later lecture



Let us look at an illustration of what you heard on the previous slide. On the slide you can see a circuit consisting of ”k” input flip-flops that capture the bits of a data word provided to them. The outputs of these flip-flops are connected to a block of some combinational logic and the result of this logic is then captured by a flip-flop and provided at its output.

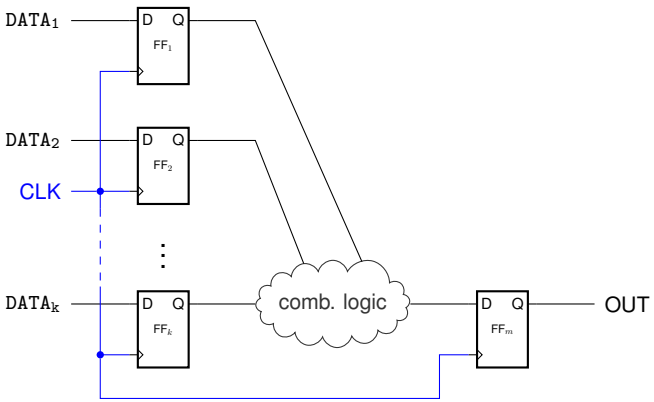
Static Timing Analysis Illustration

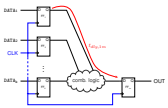




All flip-flops are connected to our global clock and the goal of the ”STA” is to determine the smallest possible clock period, or equivalently the highest possible clock frequency.

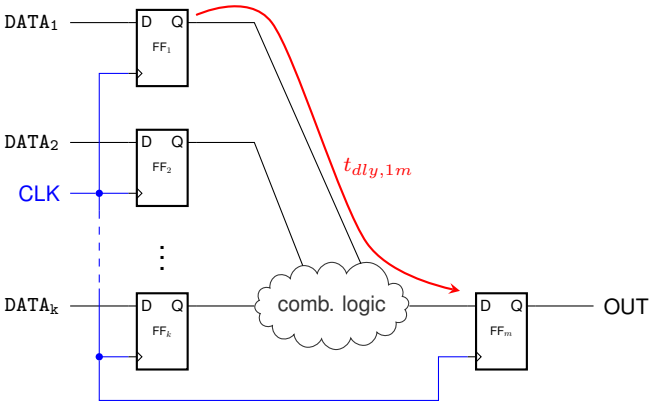
Static Timing Analysis Illustration

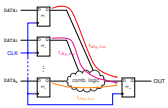




We start by determining the worst-case delay on the path from the output of the first flip-flop, through the combinational logic, to the input of the final flip flop. This path is highlighted in red.

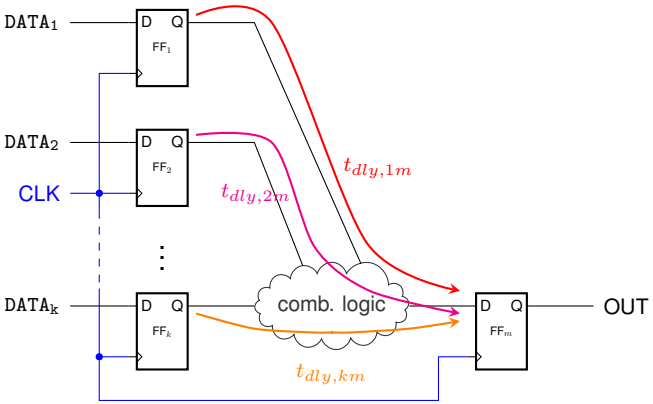
Static Timing Analysis Illustration

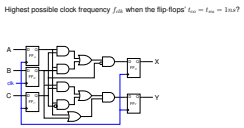




Likewise we determine all other such paths from the remaining input flip-flops. Note that the propagation delay through the combinational logic can and will usually be different for the different flip-flop outputs, as the signals run through different gates.

Static Timing Analysis Illustration

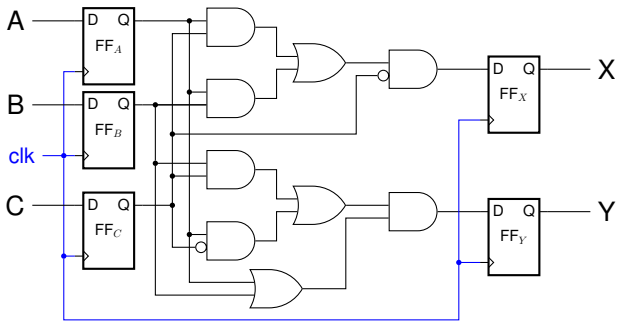


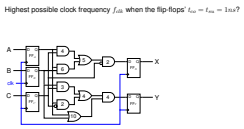


Let us now consider a bit less abstract example for a static timing analysis. On the slide you can see a circuit consisting of three input flip-flops, a combinational function, and two output flip-flops. We want to determine the maximum possible clock frequency such that only valid values are captured by the flip-flops. Naturally we require information about the timing properties of the individual elements of the circuit to perform the ”STA”. Both the clock-to-output and the setup time for the flip-flops are given to be one nanosecond. This means that it takes one nanosecond each time the flip-flop samples a value until it is available at its output, and that the flip-flop inputs must be stable for at least one nanosecond before they get sampled.

Calculation Example

Highest possible clock frequency f_{clk} when the flip-flops' $t_{co} = t_{su} = 1ns$?

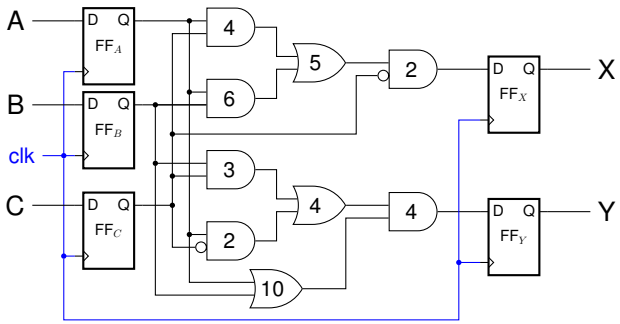


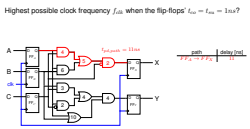


The combinational gates have different propagation delays, shown in nanoseconds in the respective symbol. Now we have to consider all possible paths from the output of a flip-flop to the input of a flip-flop.

Calculation Example

Highest possible clock frequency f_{clk} when the flip-flops' $t_{co} = t_{su} = 1ns$?

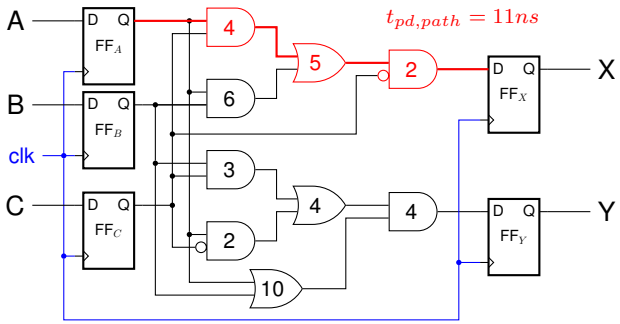




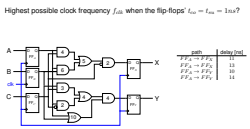
For example, one such path goes from the output of flip-flop ”A” via and ”AND” gate, an ”OR” gate, and finally and ”AND” gate again to the input of flip-flop ”X”. The total propagation delay along this path is the sum of the gate delays and thus eleven nanoseconds.

Calculation Example

Highest possible clock frequency f_{clk} when the flip-flops' $t_{co} = t_{su} = 1ns$?



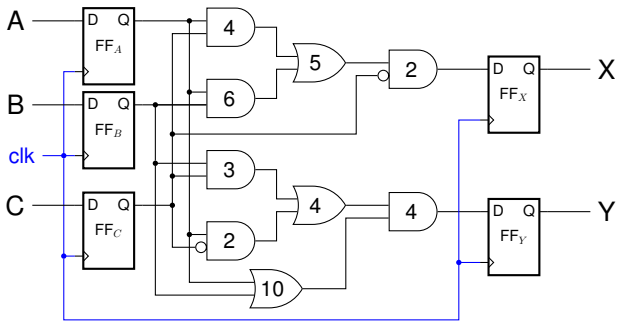
path	delay [ns]
$FF_A \rightarrow FF_X$	11



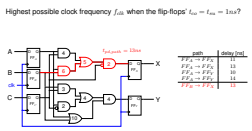
We can systematically continue to analyze all paths from flip-flop ”A” to one of the output flip-flops. The table to the right contains the determined path delays.

Calculation Example

Highest possible clock frequency f_{clk} when the flip-flops' $t_{co} = t_{su} = 1ns$?



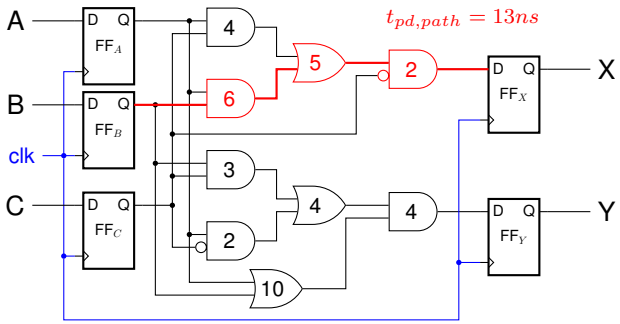
path	delay [ns]
$FF_A \rightarrow FF_X$	11
$FF_A \rightarrow FF_X$	13
$FF_A \rightarrow FF_Y$	10
$FF_A \rightarrow FF_Y$	14



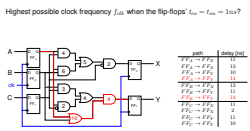
We then continue with the next flip-flop, in this case the one labelled ”B” and determine all paths to any output flip-flop. The slide shows one such path with a delay of thirteen nanoseconds.

Calculation Example

Highest possible clock frequency f_{clk} when the flip-flops' $t_{co} = t_{su} = 1ns$?



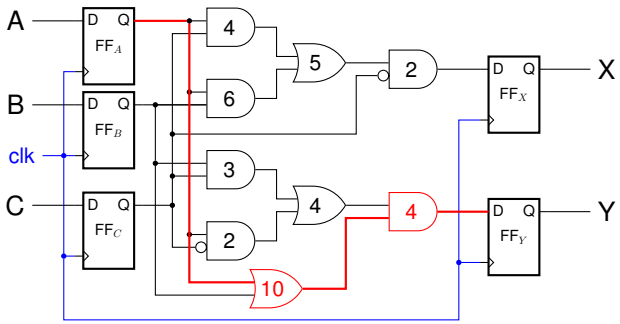
path	delay [ns]
$FF_A \rightarrow FF_X$	11
$FF_A \rightarrow FF_X$	13
$FF_A \rightarrow FF_Y$	10
$FF_A \rightarrow FF_Y$	14
$FF_B \rightarrow FF_X$	13



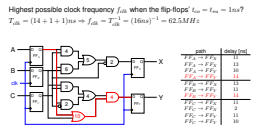
After we are done enumerating all paths and determining their respective delay value, we can easily determine the critical one. In this case we have two paths through the combinational logic with an overall delay of fourteen nanoseconds. They are both highlighted in red in the table. Furthermore, one of them is highlighted in the circuit as well.

Calculation Example

Highest possible clock frequency f_{clk} when the flip-flops' $t_{co} = t_{su} = 1ns$?



path	delay [ns]
$FF_A \rightarrow FF_X$	11
$FF_A \rightarrow FF_X$	13
$FF_A \rightarrow FF_Y$	10
$FF_A \rightarrow FF_Y$	14
$FF_B \rightarrow FF_X$	13
$FF_B \rightarrow FF_Y$	11
$FF_B \rightarrow FF_Y$	14
$FF_C \rightarrow FF_X$	11
$FF_C \rightarrow FF_X$	2
$FF_C \rightarrow FF_Y$	11
$FF_C \rightarrow FF_Y$	10

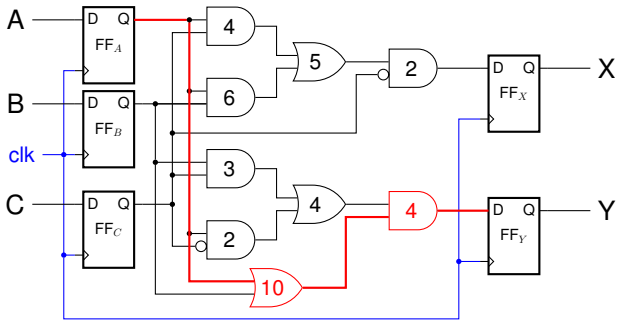


Knowing the longest path through the combinational logic we can finally determine the highest possible clock period. To do so we take the clock-to-output and setup times of our flip-flops, both one nanosecond, and add it to the critical path we determined. This tells us that the smallest possible clock period is sixteen nanoseconds, resulting in a maximum clock frequency of 62.5 MHz.

Calculation Example

Highest possible clock frequency f_{clk} when the flip-flops' $t_{co} = t_{su} = 1ns$?

$$T_{clk} = (14 + 1 + 1)ns \Rightarrow f_{clk} = T_{clk}^{-1} = (16ns)^{-1} = 62.5MHz$$



path	delay [ns]
$FF_A \rightarrow FF_X$	11
$FF_A \rightarrow FF_X$	13
$FF_A \rightarrow FF_Y$	10
$FF_A \rightarrow FF_Y$	14
$FF_B \rightarrow FF_X$	13
$FF_B \rightarrow FF_Y$	11
$FF_B \rightarrow FF_Y$	14
$FF_C \rightarrow FF_X$	11
$FF_C \rightarrow FF_X$	2
$FF_C \rightarrow FF_Y$	11
$FF_C \rightarrow FF_Y$	10

Now that we understand the concept of synchronous design, let us review its benefits.

Benefits of Synchronous Design

Discretization of Time
Concentrate on points in time where all inputs and outputs are stable
Designing synchronous circuits is relatively easy and efficient

First of all, the discretization of time through the clock makes it much easier to design a circuit. We do not need to consider unstable signals or unwanted transitions. We just operate with stable values on a fixed time grid. By just concentrating on the points in time, where all signals are stable, we can comparably easy and efficiently design circuits.

Benefits of Synchronous Design

- Discretization of Time
  - Concentrate on points in time where all inputs and outputs are stable

  $\Rightarrow$ Designing synchronous circuits is relatively easy and efficient

- Discretization of Time
  - Concentrate on points in time where all inputs and outputs are stable
  - Designing synchronous circuits is relatively easy and efficient
- High efficiency
  - Just one single signal required to coordinate all activities in the circuit
  - This periodic clock signal is easy to generate

But also the implementation is very efficient. If we need to implement some coordination for our circuit, what could be simpler than using a ”single” signal for this purpose! And a periodic signal can be easily generated by an oscillator. In practice, most often a crystal oscillator is used due to its good stability.

Benefits of Synchronous Design

- Discretization of Time
  - Concentrate on points in time where all inputs and outputs are stable
  - $\Rightarrow$ Designing synchronous circuits is relatively easy and efficient
- High efficiency
  - Just one single signal required to coordinate all activities in the circuit
  - This periodic clock signal is easy to generate

- Discretization of Time
  - Concentrate on points in time where all inputs and outputs are stable
  - Designing synchronous circuits is relatively easy and efficient
- High efficiency
  - Just one single signal required to coordinate all activities in the circuit
  - This periodic clock signal is easy to generate
- Proven in practice
  - Billion working designs

And finally, synchronous design has proven useful in billions of working designs, and engineers have lots of experiences with it.

Benefits of Synchronous Design

- Discretization of Time
  - Concentrate on points in time where all inputs and outputs are stable
  - $\Rightarrow$ Designing synchronous circuits is relatively easy and efficient
- High efficiency
  - Just one single signal required to coordinate all activities in the circuit
  - This periodic clock signal is easy to generate
- Proven in practice
  - Billion working designs

However, there are a couple of issues with synchronous design as well.

Issues with Synchronous Design

Clock distribution
Clock edges must arrive at all flip flops at (nearly) the same time
⇒ The clock network is power hungry and challenging to design

A central assumption for our coordination approach was that the clock establishes a global time base that is perceived by all flip-flops in the exact same way. This means all flip-flops must receive the clock edges at the same time. This is close to impossible, since the distribution of the clock requires a huge tree of signal lines. These signal lines have delays, and for signal integrity purposes there may be buffers inserted that add extra delay. So a very careful balancing of these delays and clever arrangement of the tree needs to be done. In addition, the large tree wit its numerous buffers consumes a lot of power.

Issues with Synchronous Design

- Clock distribution
  - Clock edges must arrive at all flip flops at (nearly) the same time

  $\Rightarrow$ The clock network is power hungry and challenging to design

- Clock distribution
  - Clock edges must arrive at all flip flops at (nearly) the same time
  - ⇒ The clock network is power hungry and challenging to design
- Delay uncertainties
  - Propagation delays vary with temperature, supply voltage and are subject to fabrication tolerances
  - ⇒ Require worst-case assumptions, wasting performance

Another major issue comes from the fact that propagation delays are in practice not well known. They vary with temperature and supply voltage, and in the nanoscale, fabrication delay is also subject to considerable tolerances. So in the static timing analysis we need to stay on the safe side and assume the worst conceivable conditions, the so-called worst case. This then results in a conservative clock limit for the whole design, while most of the individual chips implementing the circuit could in fact go faster.

Issues with Synchronous Design

- Clock distribution
  - Clock edges must arrive at all flip flops at (nearly) the same time
  - ⇒ The clock network is power hungry and challenging to design
- Delay uncertainties
  - Propagation delays vary with temperature, supply voltage and are subject to fabrication tolerances
  - ⇒ Require worst-case assumptions, wasting performance

- Clock distribution
  - Clock edges must arrive at all flip flops at (nearly) the same time
  - ⇒ The clock network is power hungry and challenging to design
- Delay uncertainties
  - Propagation delays vary with temperature, supply voltage and are subject to fabrication tolerances
  - ⇒ Require worst-case assumptions, wasting performance
- Rigid timing, no graceful degradation
  - Propagation delay exceeds clock period ⇒ completely wrong results produced

Finally, if a delay ever exceeds the worst case assumption, the circuit will deliver a completely wrong result. There is no gradual way of failing like a delayed or just slightly incorrect result.

Issues with Synchronous Design

- Clock distribution
  - Clock edges must arrive at all flip flops at (nearly) the same time
  - ⇒ The clock network is power hungry and challenging to design
- Delay uncertainties
  - Propagation delays vary with temperature, supply voltage and are subject to fabrication tolerances
  - ⇒ Require worst-case assumptions, wasting performance
- Rigid timing, no graceful degradation
  - Propagation delay exceeds clock period ⇒ completely wrong results produced

- Clock distribution
  - Clock edges must arrive at all flip flops at (nearly) the same time
  - ⇒ The clock network is power hungry and challenging to design
- Delay uncertainties
  - Propagation delays vary with temperature, supply voltage and are subject to fabrication tolerances
  - ⇒ Require worst-case assumptions, wasting performance
- Rigid timing, no graceful degradation
  - Propagation delay exceeds clock period ⇒ completely wrong results produced
- However: synchronous design is the most widely used design style
  - Alternatives exist (advanced courses)

In spite of all that, synchronous design is by far the most widely adopted design style and the standard way of doing digital design. However, it should be noted that it is also possible to achieve the required coordination without a clock using the asynchronous design paradigm. You can learn more about that in more advanced courses.

Issues with Synchronous Design

- Clock distribution
  - Clock edges must arrive at all flip flops at (nearly) the same time
  - ⇒ The clock network is power hungry and challenging to design
- Delay uncertainties
  - Propagation delays vary with temperature, supply voltage and are subject to fabrication tolerances
  - ⇒ Require worst-case assumptions, wasting performance
- Rigid timing, no graceful degradation
  - Propagation delay exceeds clock period ⇒ completely wrong results produced
- However: synchronous design is the most widely used design style
  - Alternatives exist (advanced courses)

Thank you for listening! We recommend you to immediately take the self-check test in TUWEL, to see if you understood the material presented in this lecture.

Lecture Complete!