Nexys-4 7-Segment Display Control

Recommendations and Good Design Practices

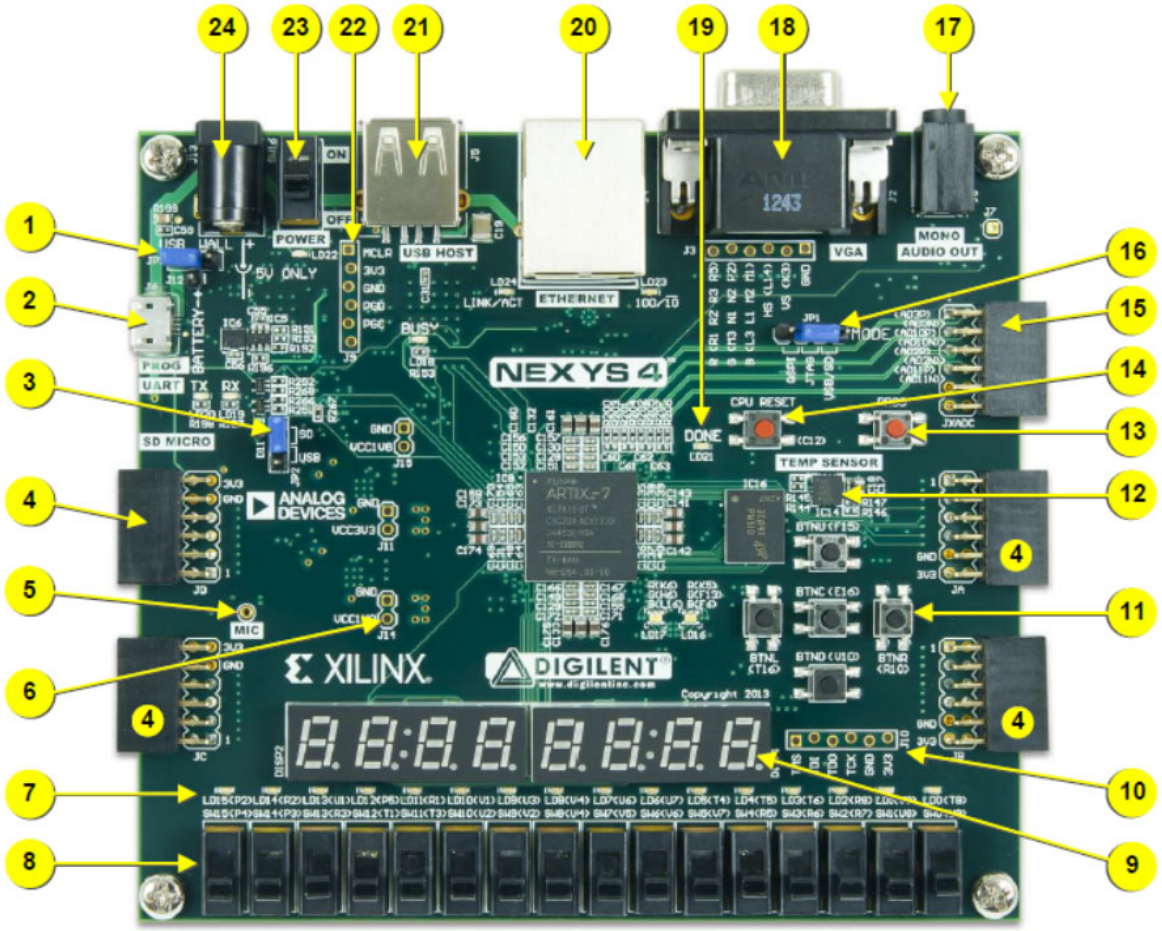
Constraints Specification and Timing Analysis

LECTURE 3

IOULIIA SKLIAROVA

Nexys-4 Development Board

- 7-segment displays
- …



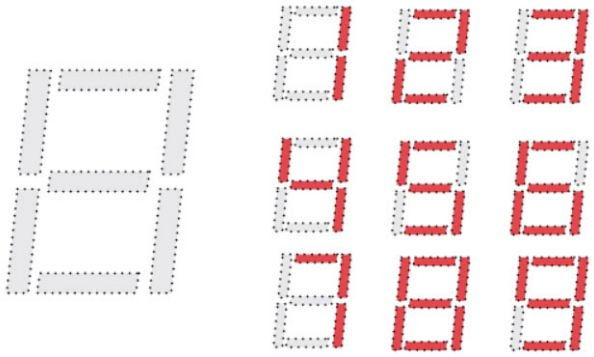
FPGA: xc7a100Tcsg324-1

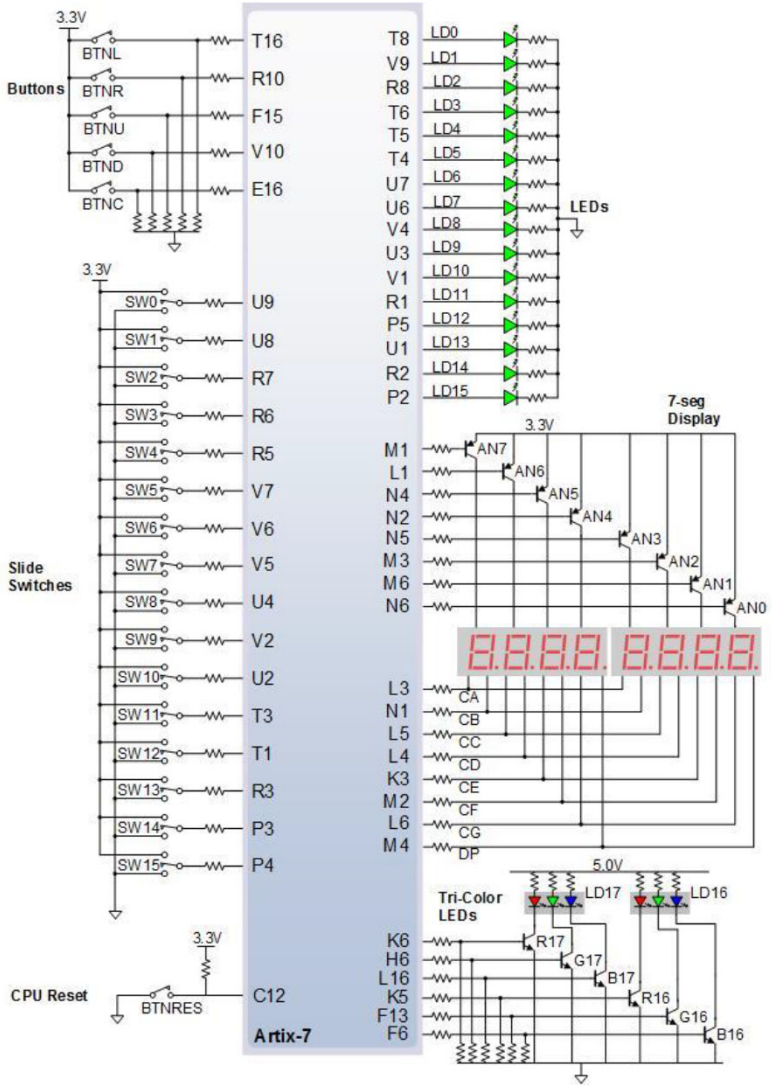
7-Segment Displays

The Nexys-4 board contains two four-digit common anode seven-segment LED displays, configured to behave like a single eight-digit display.

Each of the eight digits is composed of seven segments arranged in a “figure 8” pattern, with an LED embedded in each segment.

Segment LEDs can be individually illuminated, so any one of 128 patterns can be displayed on a digit by illuminating certain LED segments and leaving the others dark.





7-Segment Displays

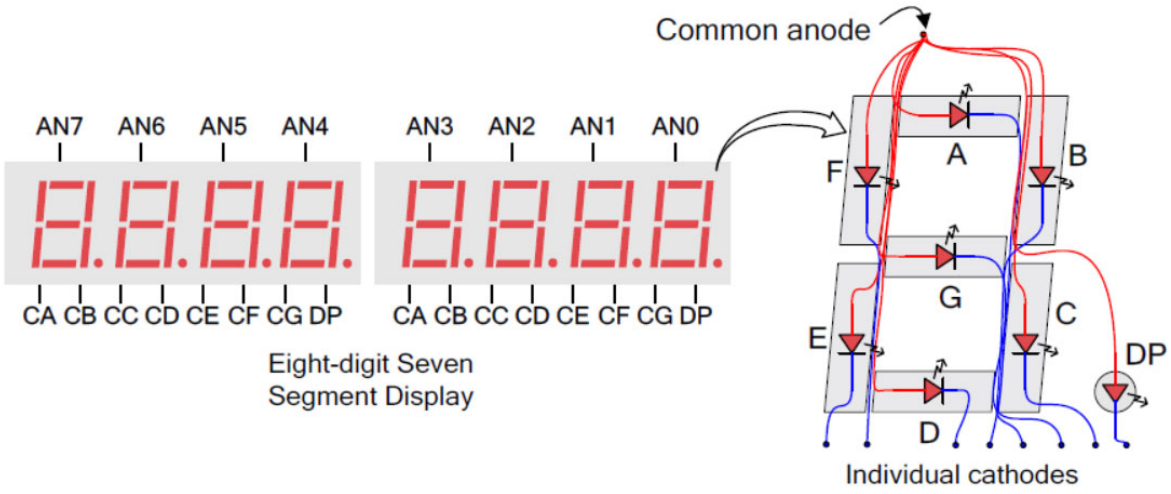
The anodes of the seven LEDs forming each digit are tied together into one “common anode” circuit node, but the LED cathodes remain separate.

The common anode signals are available as eight “digit enable” input signals to the 8-digit display.

The cathodes of similar segments on all four displays are connected into seven circuit nodes labeled CA through CG.

This signal connection scheme creates a multiplexed display, where the cathode signals are common to all digits but they can only illuminate the segments of the digit whose corresponding anode signal is asserted.

Both the AN0..7 and the CA..G/DP signals are driven low when active.



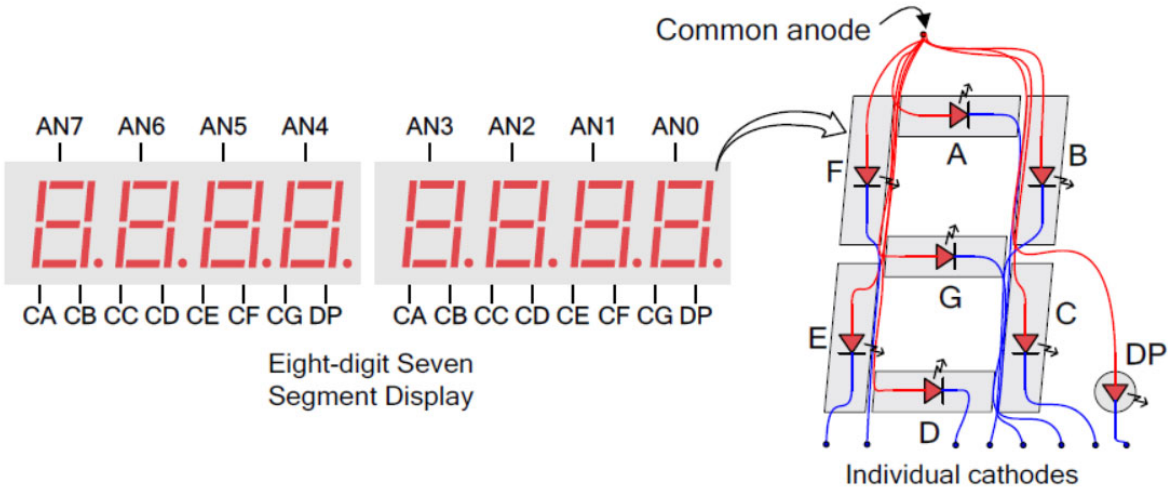
7-Segment Displays Controller

A scanning display controller circuit can be used to show an eight-digit number on this display.

This circuit drives the anode signals and corresponding cathode patterns of each digit in a repeating, continuous succession at an update rate that is faster than the human eye can detect.

Each digit is illuminated just one-eighth of the time, but because the eye cannot perceive the darkening of a digit before it is illuminated again, the digit appears continuously illuminated.

If the update, or “refresh”, rate is slowed to around 45Hz, a flicker can be noticed in the display.

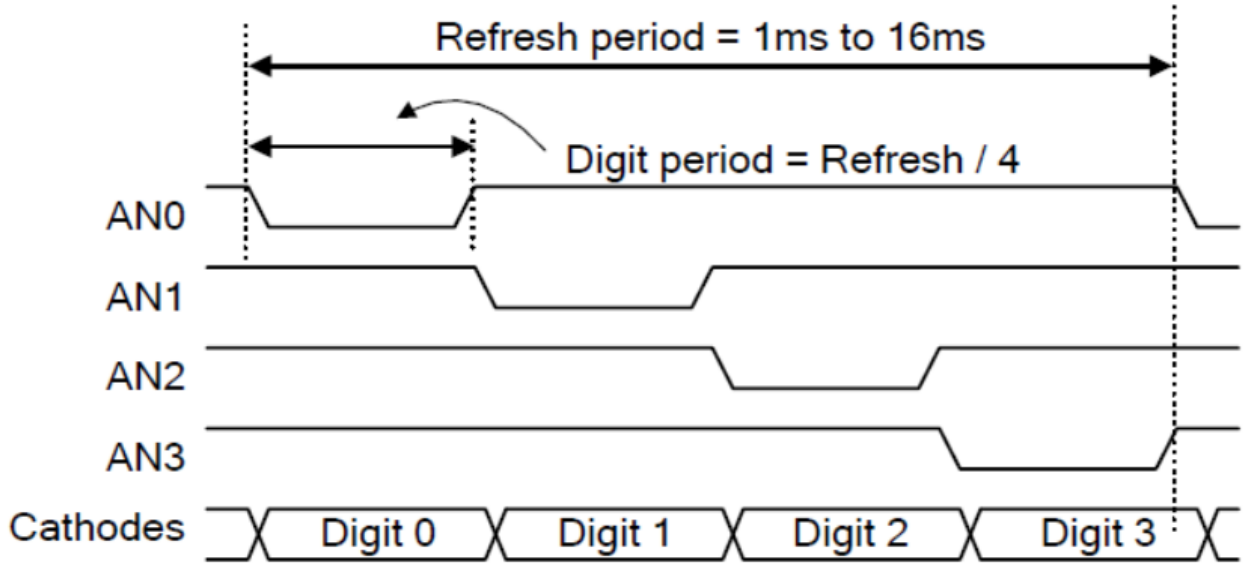


7-Segment Displays Controller

For each of the eight digits to appear bright and continuously illuminated, all eight digits should be driven once every 1 to 16ms, for a refresh frequency of about 1KHz to 60Hz.

For example, in a 62.5Hz refresh scheme, the entire display would be refreshed once every 16ms, and each digit would be illuminated for 1/8 of the refresh cycle, or 2ms.

The controller must drive low the cathodes with the correct pattern when the corresponding anode signal is driven low.



7-Segment Displays Controller

Design suggestions:

- you have to switch between digits slow enough for our human eyes to see the light
- … but fast enough so that there is no flicker
- apply a 800Hz clock (clock period = 1.25ms)
- this means that the refresh rate of all 8 digits is 100Hz and each digit is illuminated for 1.25 ms within a 10 ms period.

Asynchronous Signals

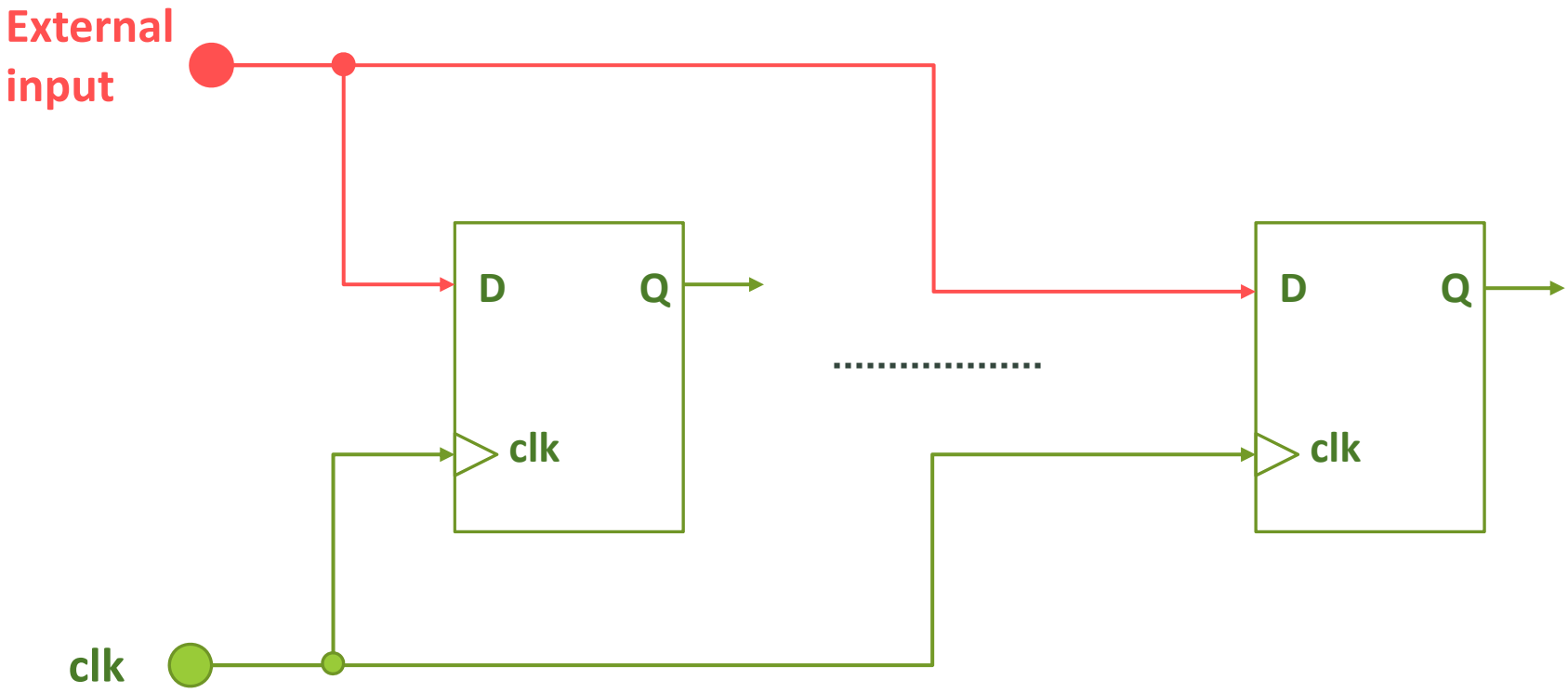
Clocked circuits are **synchronous**

Unclocked circuits or signals are circuits or signals are asynchronous

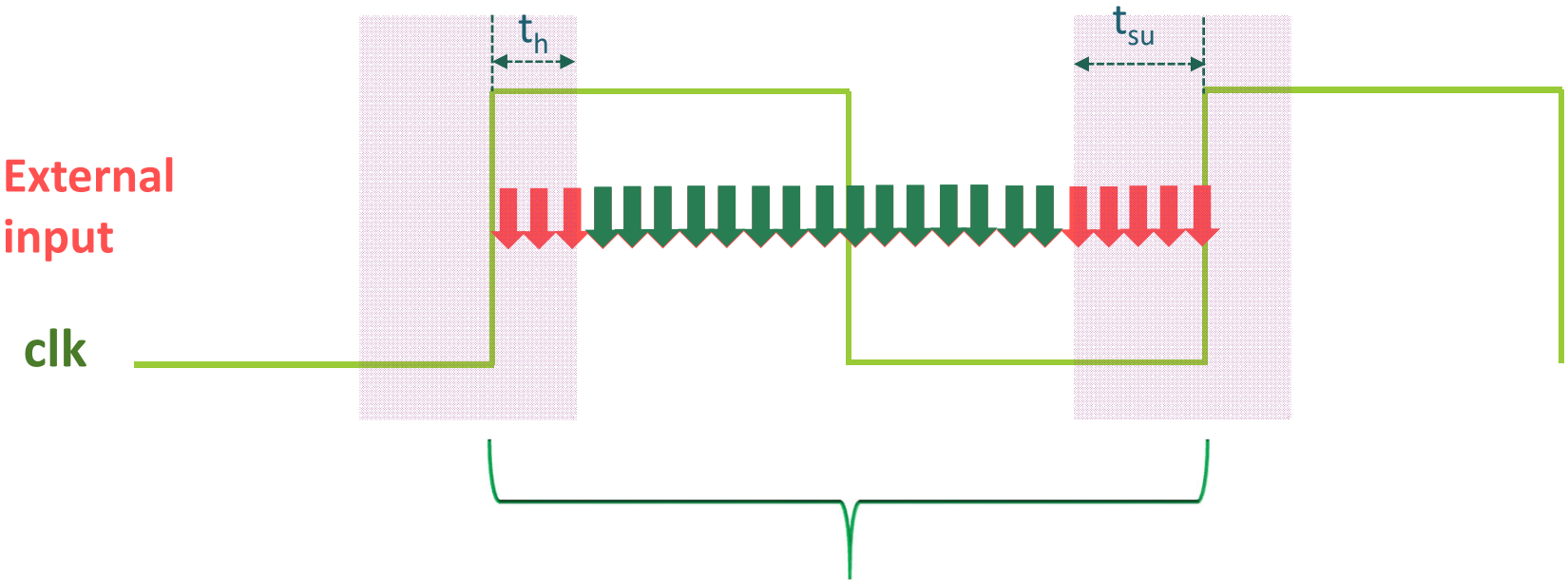
Synchronous circuits have asynchronous inputs:

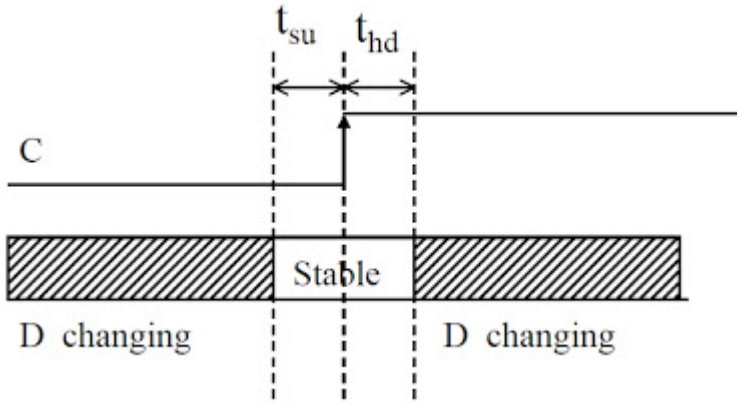
- Reset signal, memory wait, user input, etc.
- Inputs can change at any time
- We must synchronize the input to our clock
- Inputs can violate flip-flop setup/hold times

Asynchronous Inputs



Asynchronous Signals





Metastability problem:

- Occurs when input changes near active clock edge
- Not every signal transition that violates a register's t_{su} or t_h results in a metastable output
- The likelihood that a register enters a metastable state and the time required to return to a stable state vary depending on the process technology used to manufacture the device and on the operating conditions
- In most cases, registers will quickly return to a stable defined state

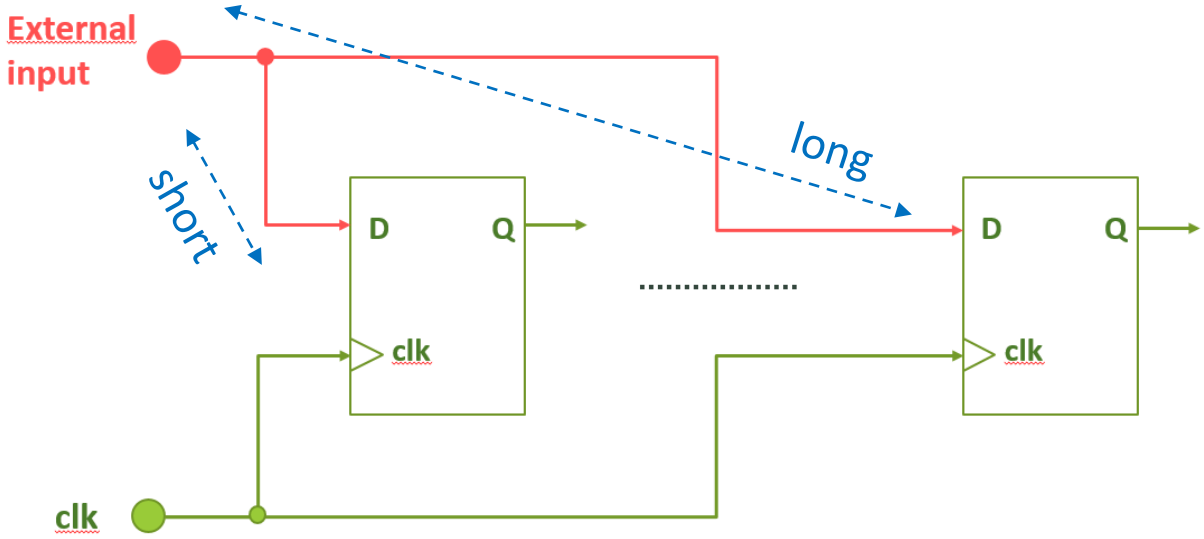
Asynchronous Inputs

In addition to metastability:

Slight delay differences mean that the registers can disagree on the input value

"Inconsistent value problem"

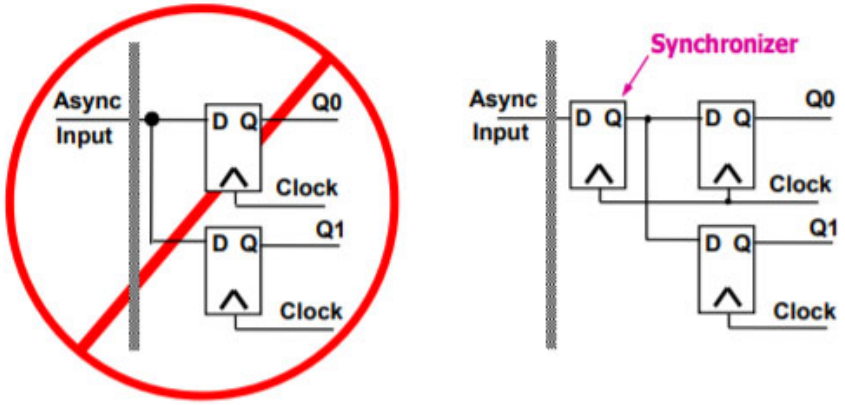
- Two paths from input to two different registers:



Handling Asynchronous Inputs

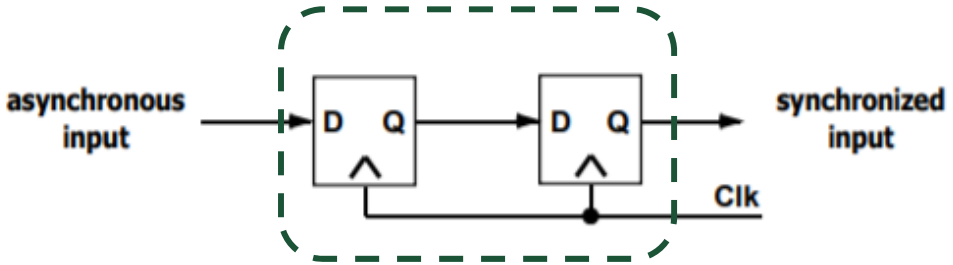
Never fan-out asynchronous inputs

- Synchronize at circuit boundary – **synchronization register**
- Fan-out synchronized signal
- Failure probability can never be zero

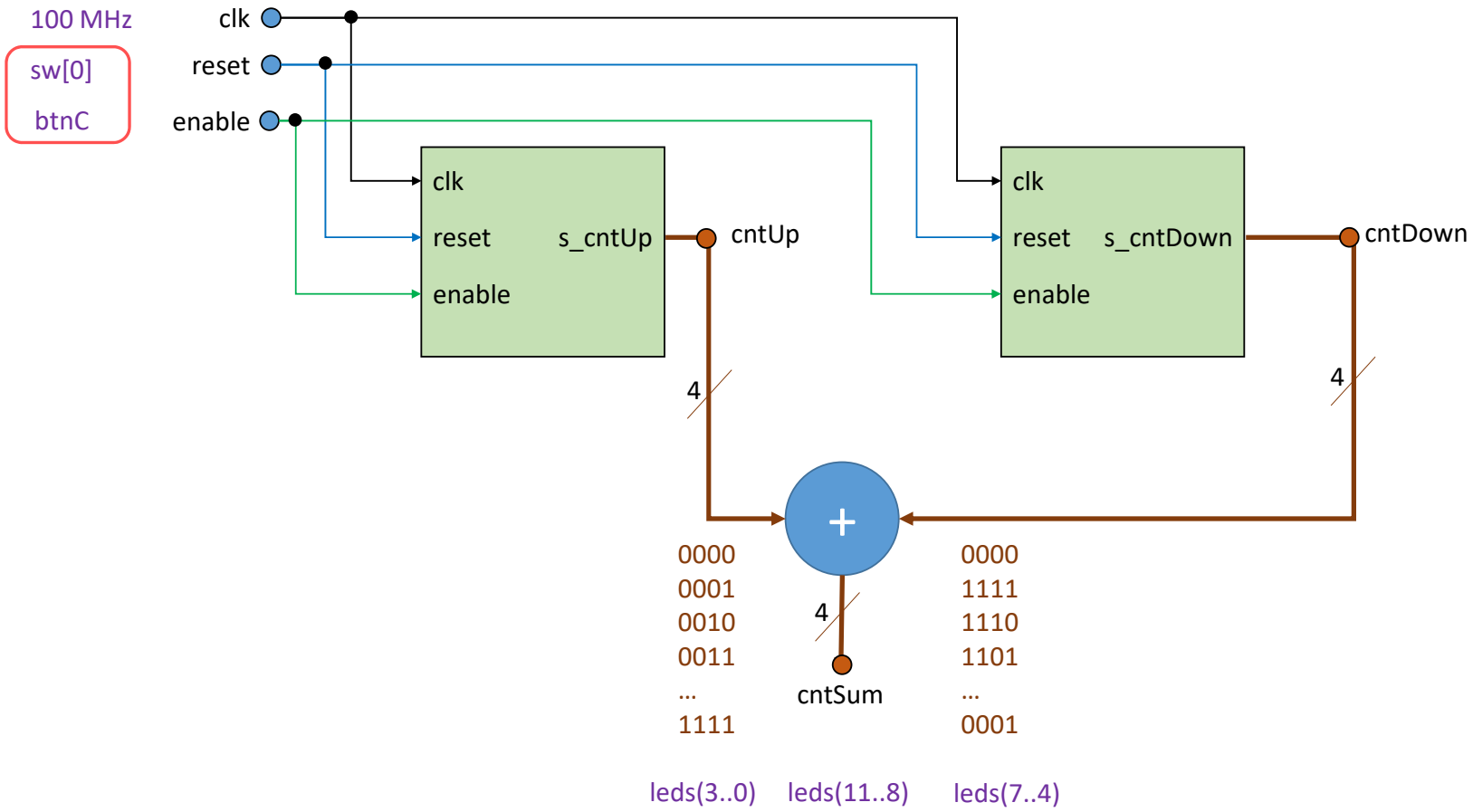


- To minimize the failures due to metastability in asynchronous signal transfers cascade two (or more) flip-flops
  - Effectively synchronizes twice
  - Both would have to fail for system to fail

Synchronization chain



Example Project (Vivado)



Murphy's law: Anything that can go wrong will go wrong

Timing Analysis

Timing analysis is the methodical analysis of a digital circuit to determine if the **timing constraints** imposed by components or interfaces are met. Typically, this means that you are trying to prove that all set-up, hold, and pulse-width times are being met.

Timing analysis can be **static** or **dynamic**.

- **Dynamic timing analysis (DTA)** verifies functionality of the design by applying input vectors and checking for correct output vectors. This approach is an extension of simulation and ensures that circuit timing is tested in its functional context.
- **Static timing analysis (STA)** checks static delay requirements of the circuit without any input or output vectors.

DTA has to be accomplished and functionality of the design must be cleared before the design is subjected to STA.

DTA and STA are not alternatives to each other.

Quality of the DTA increases with the increase of input test vectors.

DTA can be used for synchronous as well as asynchronous designs.

STA can't run on asynchronous designs and hence DTA is the best way to analyze asynchronous designs.

DTA is also best suitable for designs having clocks crossing multiple domains.

What is Timing Analysis for?

Questions

- What is the longest delay in your circuit?
- Can the system operate at a target clock frequency?
- Are the timing requirements met for all paths in your design?
- How to determine the maximum operating frequency of a system?

Static Timing Analysis

Static timing analysis is a method of validating the timing performance of a design by checking all possible paths for timing violations.

It considers the “worst” possible delay through each logic element, but not the logical operation of the circuit.

In comparison to circuit simulation, static timing analysis is faster and more thorough.

However, static timing analysis checks the design only for proper timing, not for correct logical functionality.

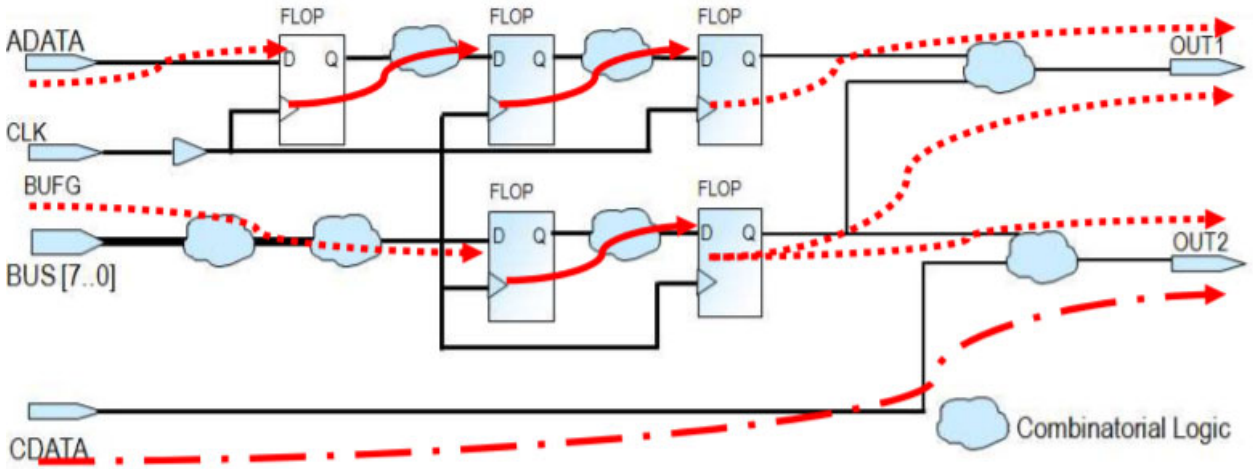
How Does STA Work?

Every device path in a design must be analyzed with respect to timing specifications/requirements

- Catch timing-related errors faster and easier than gate-level simulation & board testing

Designer must enter timing requirements & exceptions

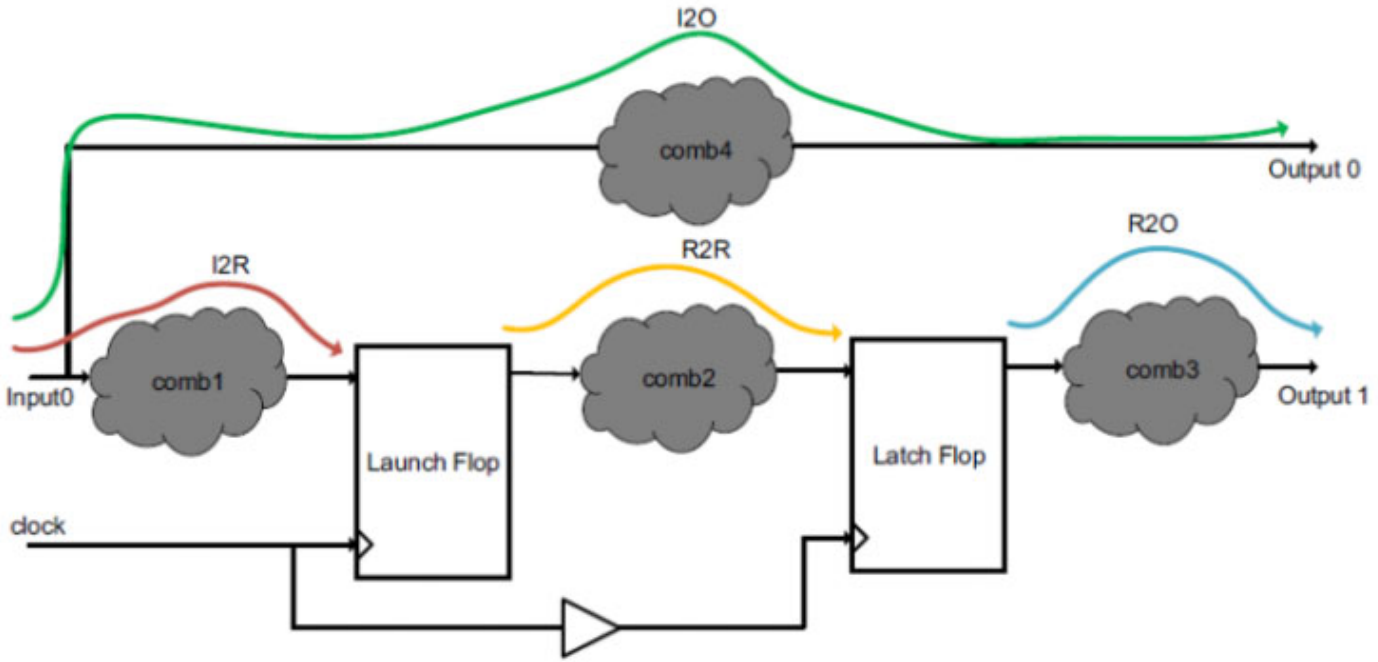
- Used to guide the FPGA tools during placement and routing
- Used to compare against actual results (post-place and route)



Path Categories

Any digital design can be divided into four categories of paths for STA:

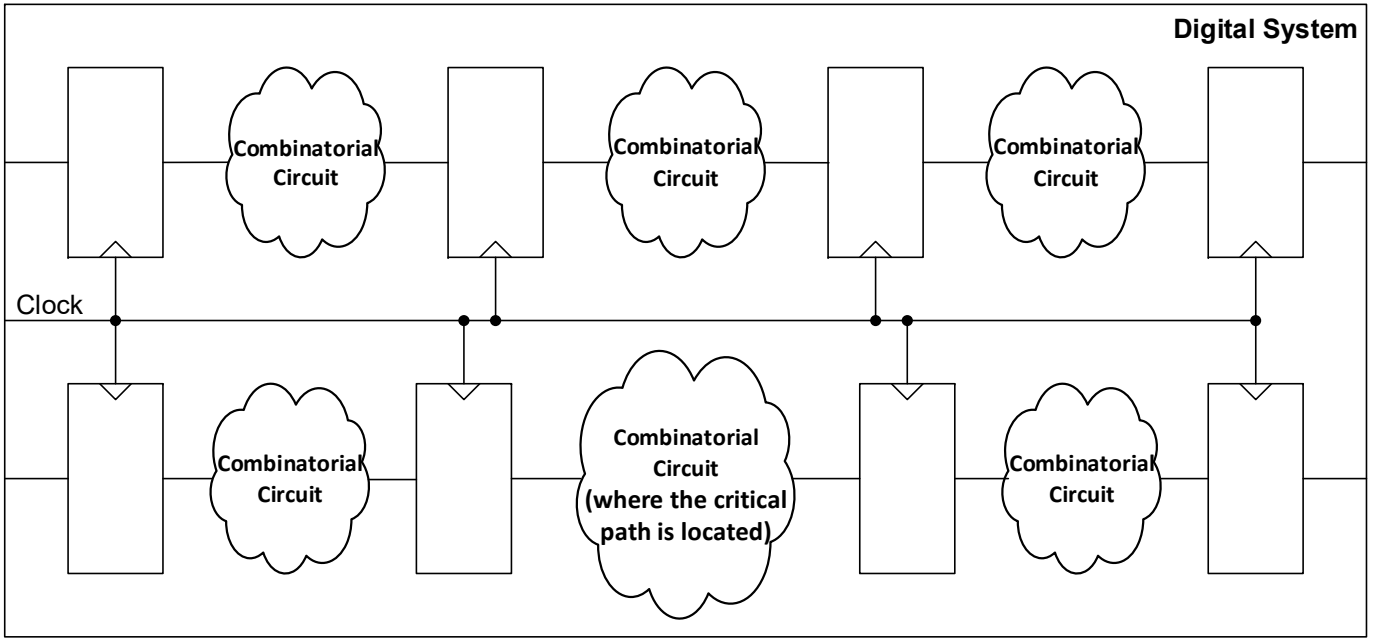
- Input to Output (I2O)
- Input to Register (I2R)
- Register to Register (R2R)
- Register to Output (R2O)



Basic Assumptions

In CR we will make the following assumptions/simplifications:

- The design is synchronous
- The system operates within a single clock domain (single clock signal)
- All system inputs and outputs are registered (at the top-level)
- The external (top-level) I/O timings (including setup and hold times) are ignored



Timing Paths

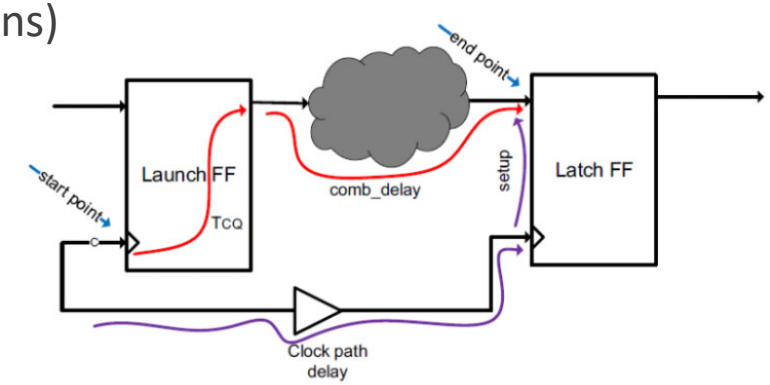
A path is a route from a startpoint to an endpoint

Startpoint, a.k.a **Primary Inputs (PI)**:

- Clock pins of flip-flops
- Input ports

Endpoints, a.k.a **Primary Outputs (PO)**:

- Input pins of-flip flops (except the clock pins)
- Output ports
- Memories
- Hard macros



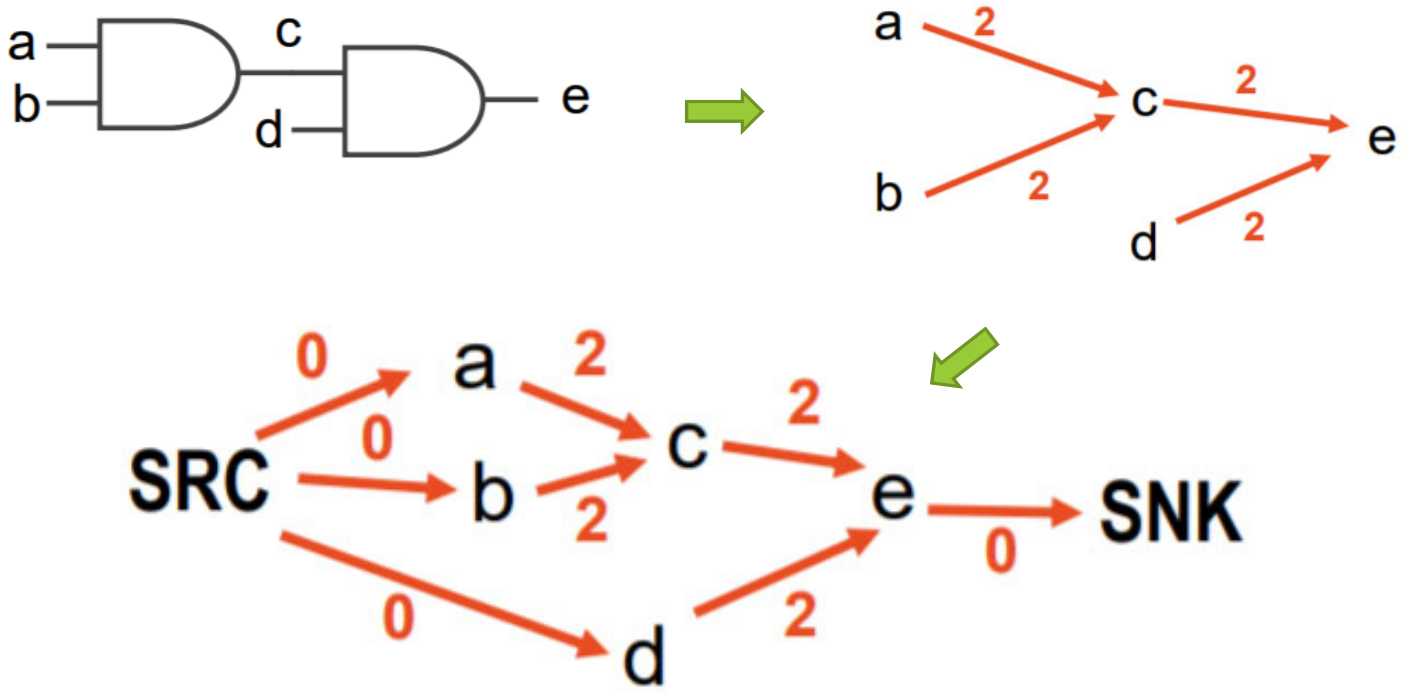
There can be:

- Many paths going to any one endpoint
- Many paths for each start-point and end-point combination

Simple Path Representation

Node-oriented timing analysis:

- For each node, find the worst delay to the node along any path
  - build a delay graph with nodes representing wires and edges representing logical elements
  - add source/sink nodes such that all paths start and end at a single node



Definitions

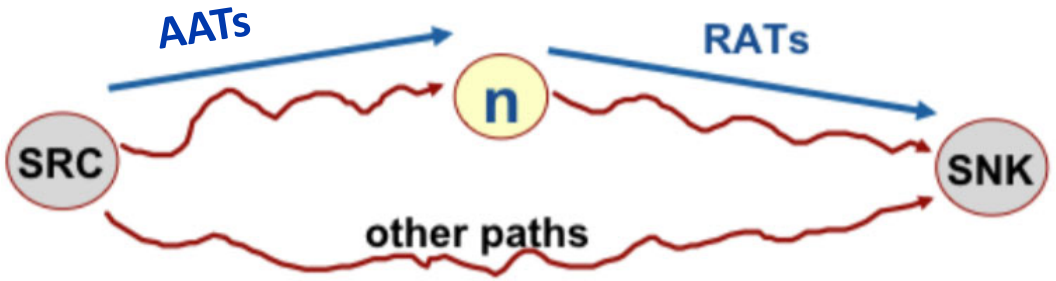
Actual Arrival Time at a node (AAT): the longest path from the source to the node.

Required Arrival Time at node (RAT): the latest time the signal is allowed to leave the node to make it to the sink in time.

The AAT at a node is the maximum of the AATs at the predecessor nodes plus the delay from that node.

The RAT to a node is the minimum of the RATs at the successor nodes minus the delay to that node.

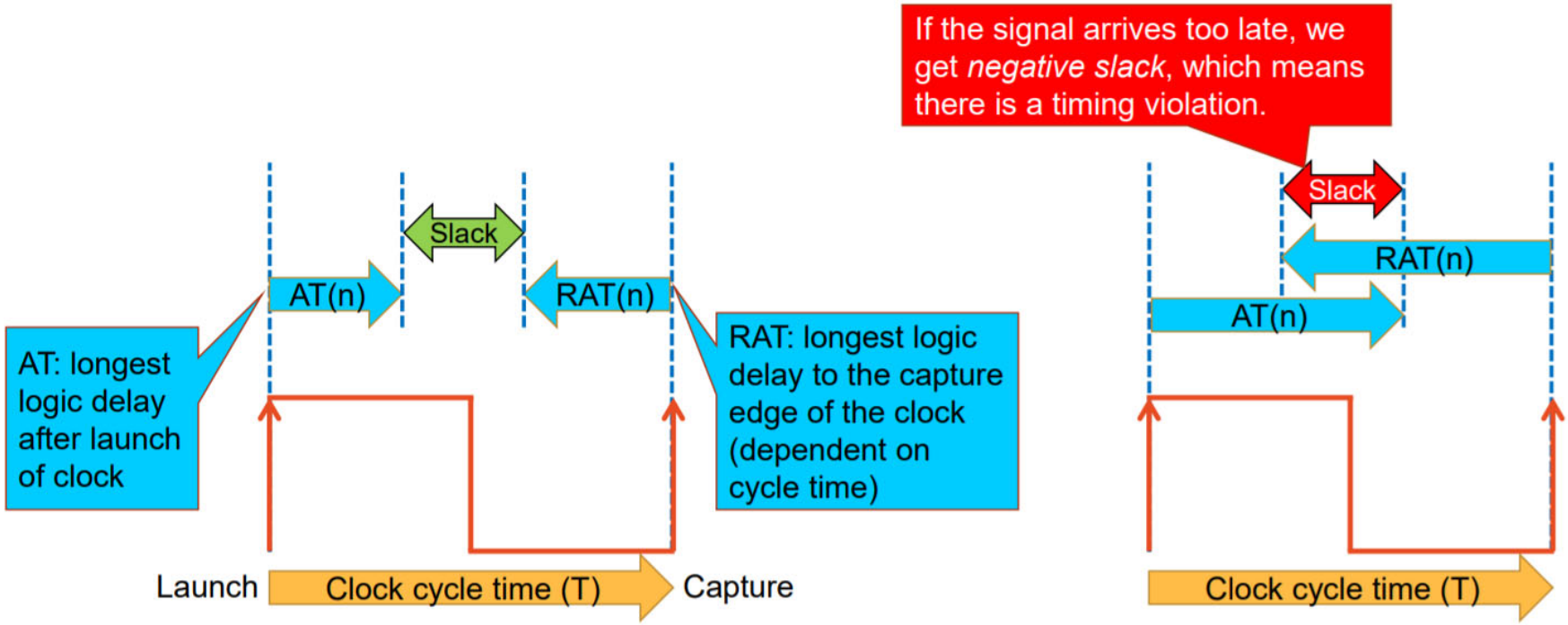
Slack at a node *n* is defined as: RAT(*n*) – AAT(*n*)



The Slack

Positive slack = OK

Negative slack = problem: **timing violation**

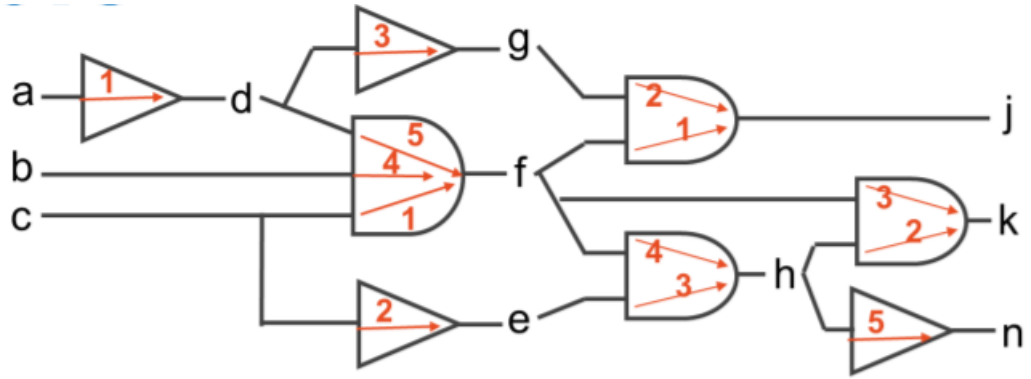


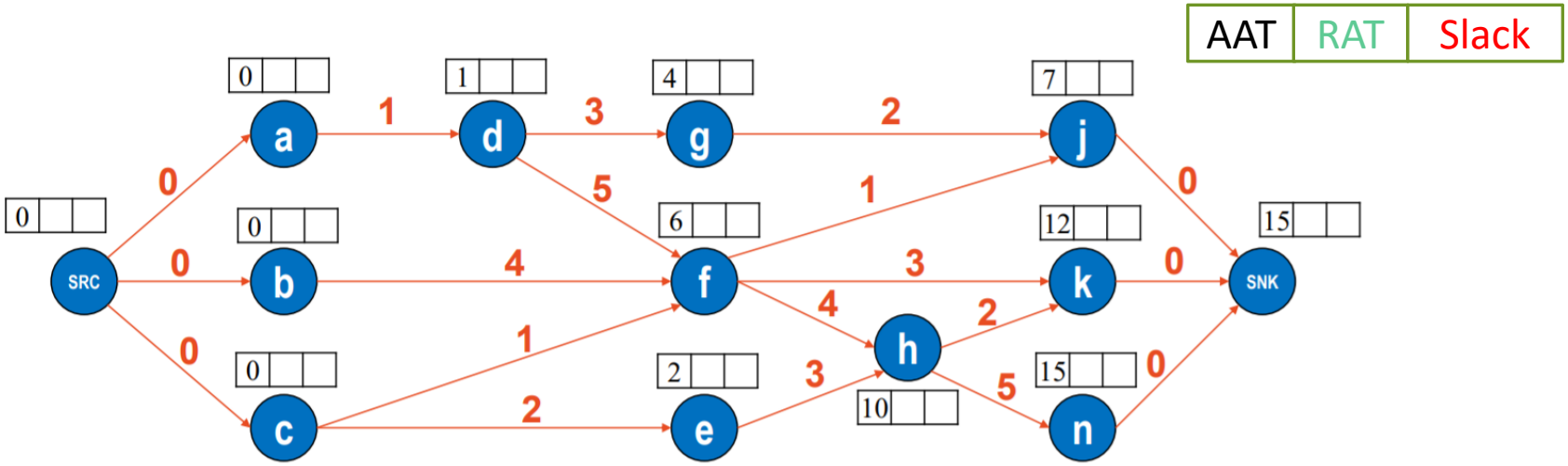
*AT=AAT

Example

Does the circuit meet a cycle time of T=12?

What is the slack? What is the critical path?

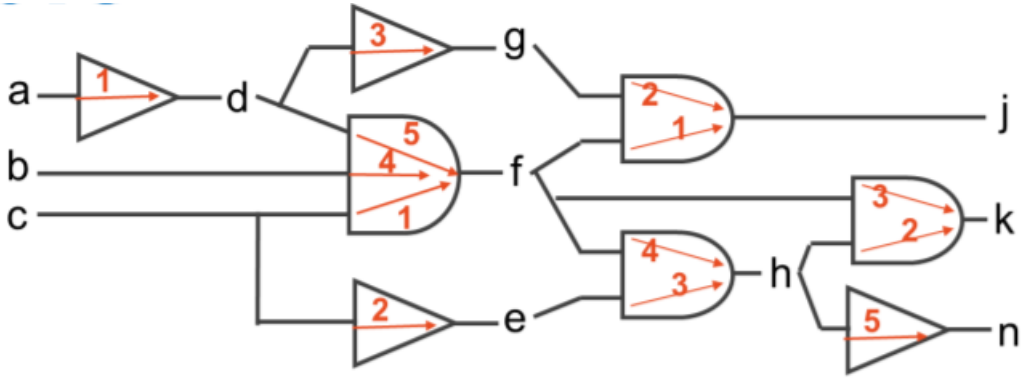


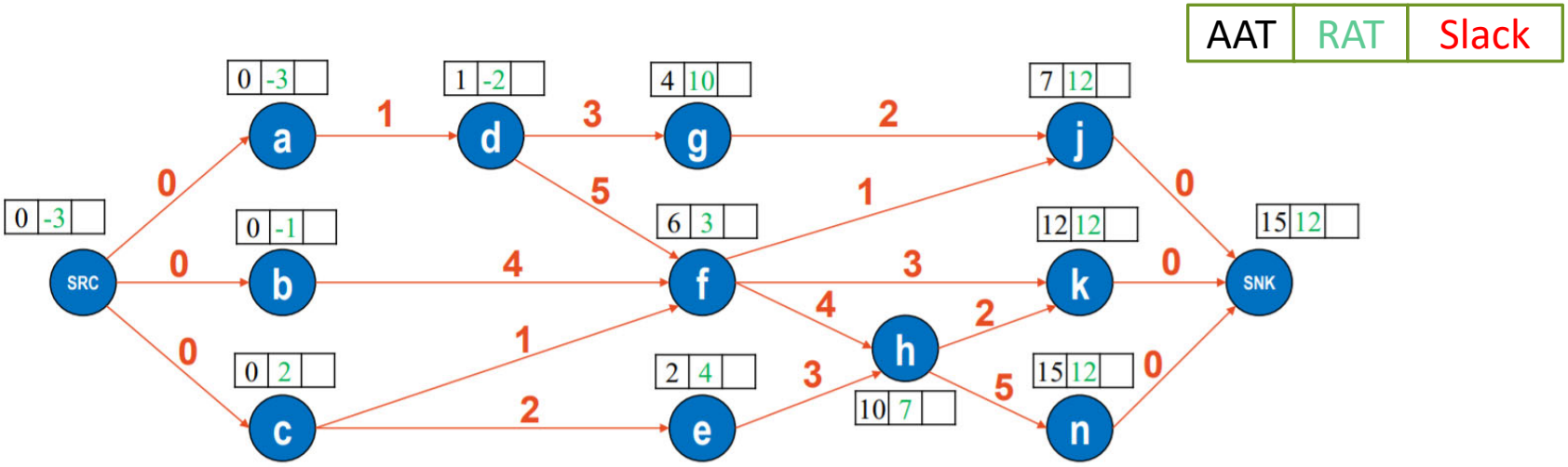


Example

Does the circuit meet a cycle time of T=12?

What is the slack? What is the critical path?

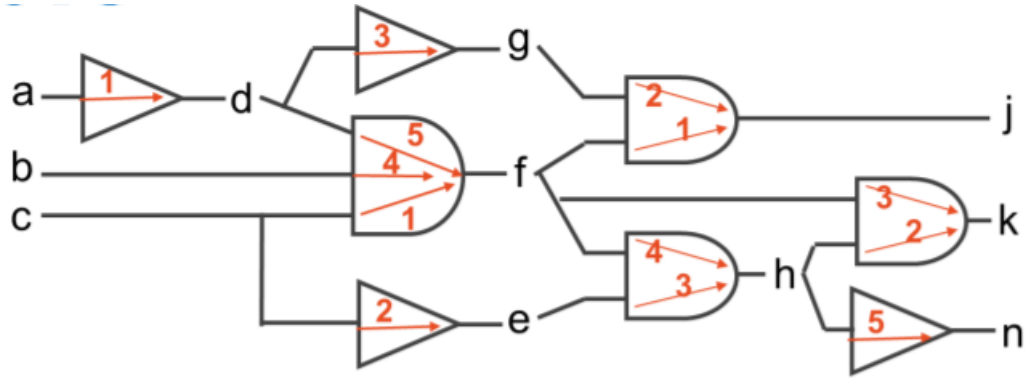


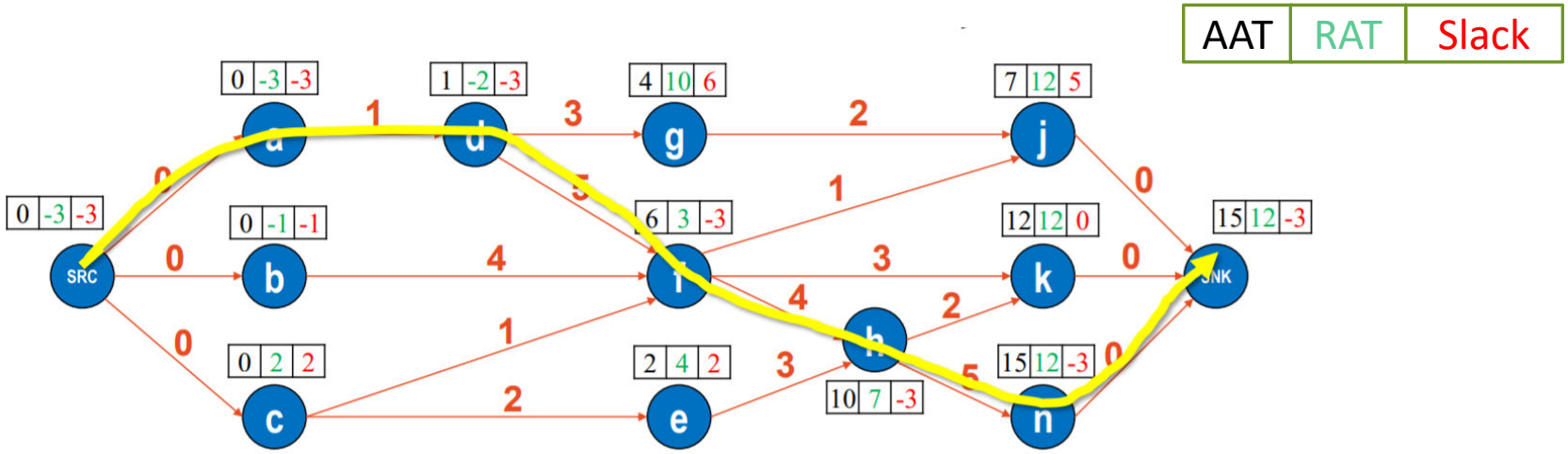


Example

Does the circuit meet a cycle time of T=12?

What is the slack? What is the critical path?





Timing Constraints

How does the STA tool know what the required clock period is?

We have to tell it!

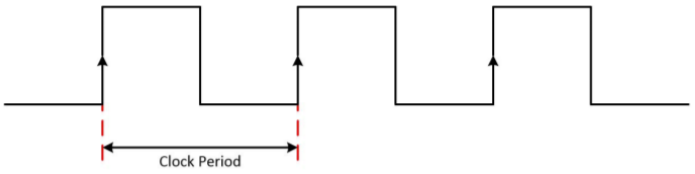
We have to define constraints for the design.

This is usually done using the Synopsys Design Constraints (SDC) syntax.

In Vivado, timing constraints must be passed to the synthesis engine by means of one or more XDC files.

To start, we must define a clock:

- Where does the clock come from?
  - (i.e., input port, output of PLL, etc.)
- What is the clock period?
  - => operating frequency
- What is the duty-cycle of the clock?



XDC Clock Definitions

SDC command syntax for creating a clock:

```
create_clock [-add] [-name <clock_name>] -period <value>
[-waveform <edge_list>] [<targets>]
```

The clock, named `my_clock` (used to refer to the clock in other commands), entering device through the port `clk`, with period of 20 ns and 50% duty cycle (default value):

```
create_clock -period 20 -name my_clock [get_ports clk]
```

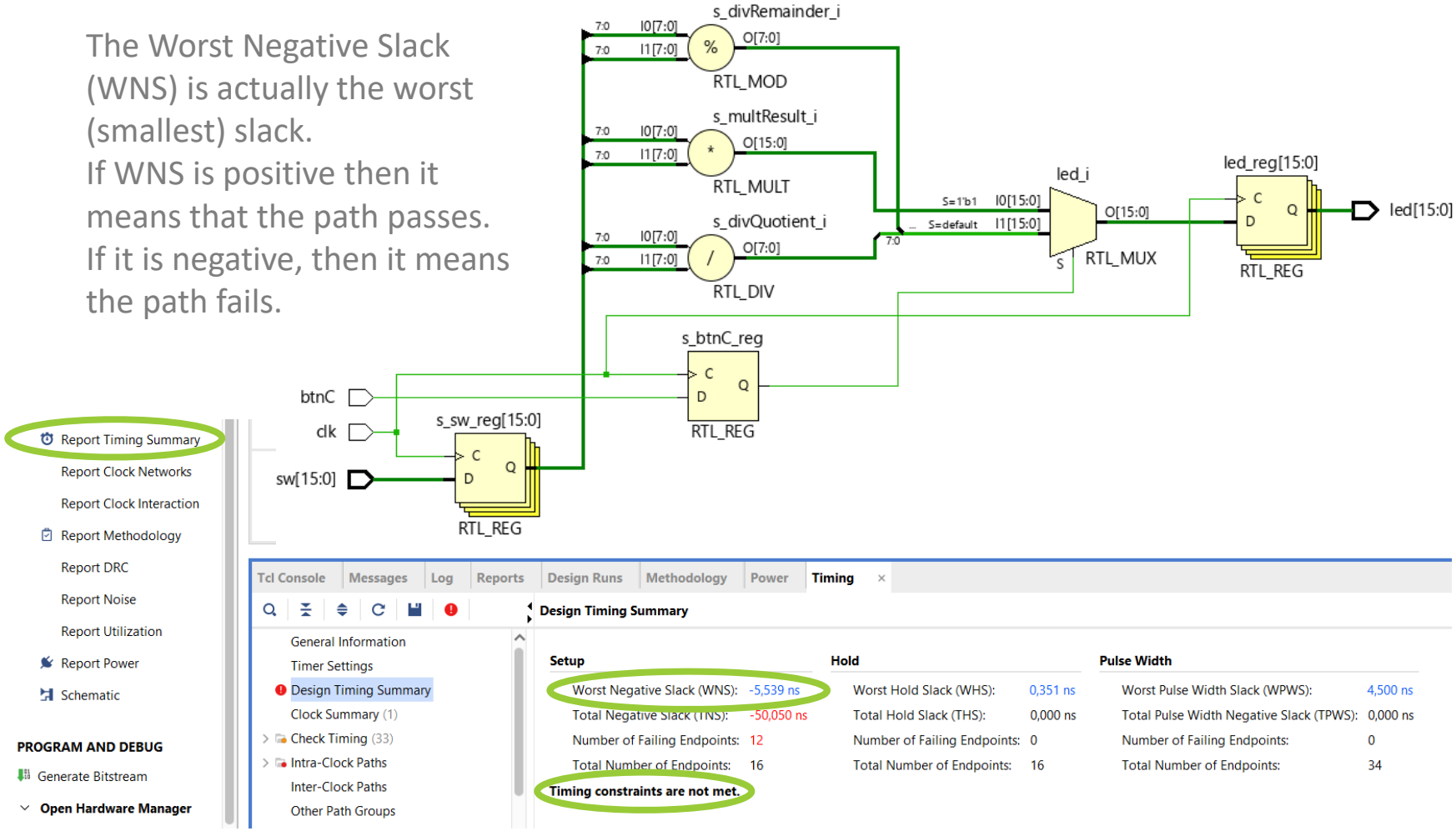
[get_ports string] – returns all ports that match string

Nexys4_Master.xdc:

```
create_clock -add -name sys_clk_pin -period 10.00 -
waveform {0 5} [get_ports clk]
```

Example Project (Vivado)

The Worst Negative Slack (WNS) is actually the worst (smallest) slack.
If WNS is positive then it means that the path passes.
If it is negative, then it means the path fails.



Maximum Frequency

The maximum frequency (f_{max}) value is not explicitly given in the report_timing/ report_timing_summary report.

The maximum frequency a design can run on hardware in a given implementation:

- f_{max} = 1/(t-WNS), where *t* is the target clock period.

The maximum frequency a design can run on a given architecture:

- The user will have to decrease *t* and re-run synthesis/implementation until WNS<0.
- Different strategies might be needed to get the best achievable f_{max}.

Final Remarks

At the end of this lecture you should

- understand how to control Nexys-4's 7-segment displays
- be able to follow the following recommendations:
  - Use a single clock signal for all the project's components
  - Register all asynchronous input signals
  - Define timing constraints
  - Detect and correct timing violations

To do:

- Test the given projects on Nexys-4 kit