EE 531: ADVANCED VLSI DESIGN

Timing Analysis

Nishith N. Chakraborty

January, 2025

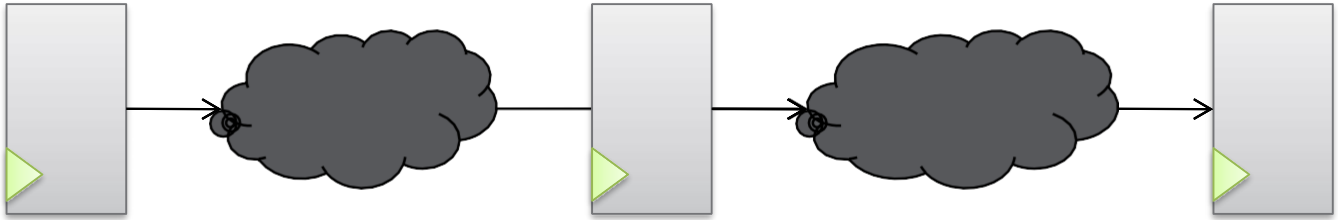
TIMING ANALYSIS

- Static Timing Analysis (STA) – method of computing expected timing of digital circuit without simulation
- Referred to as static because it does not depend on input vectors
- Each cell or module in design is accompanied by timing information (e.g., rise and fall times, delays)

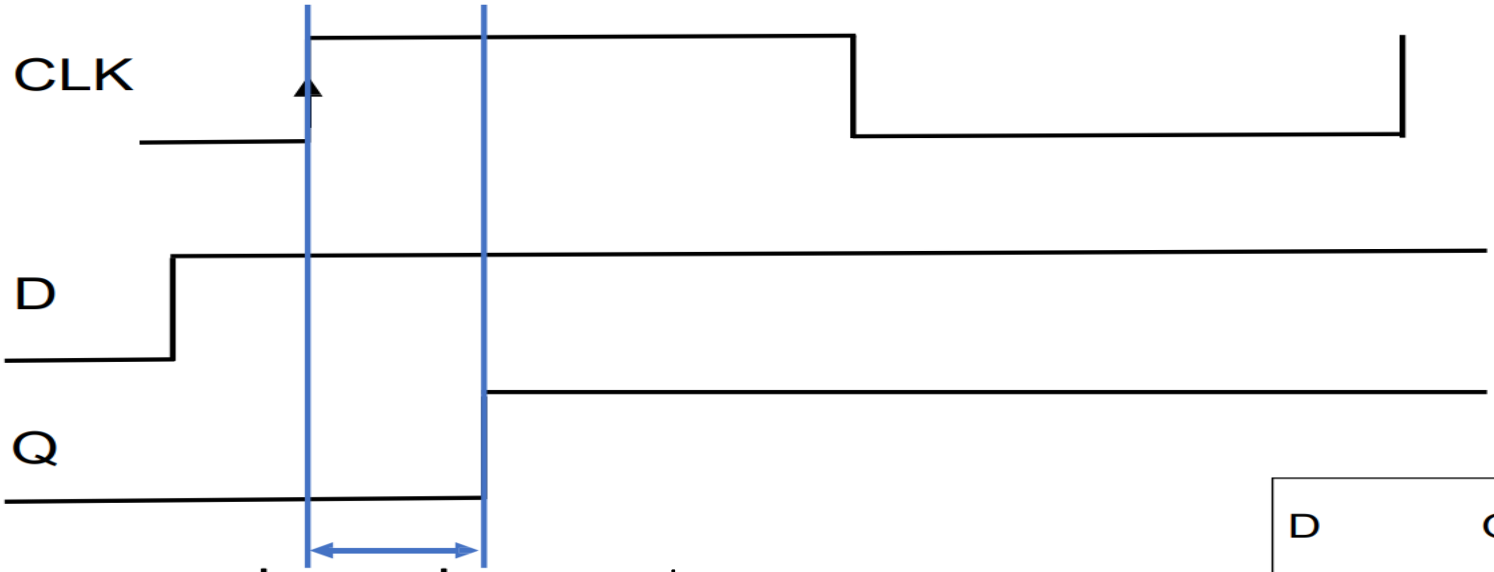
- Timing analysis (usually via STA) must occur between multiple points in the design flow and is used to optimize synthesis, placement, & routing

SYNCHRONOUS DESIGN - REVISIT

- The majority of digital designs are Synchronous and constructed with Sequential Elements.
  - Synchronous design eliminates races.
  - Pipelining increases throughput.
- We will assume that all sequentials are Edge-Triggered, using D-Flip Flops as registers.
- D-Flip Flops have three critical timing parameters:
  - t_{cq} – clock to output: essentially a propagation delay
  - t_{setup} – setup time: the time the data needs to arrive before the clock
  - t_{hold} – hold time: the time the data has to be stable after the clock

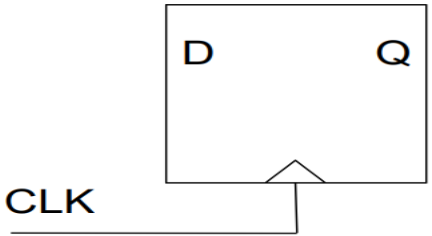


TIMING METRICS: RISING EDGE TRIGGERED FLIP-FLOP

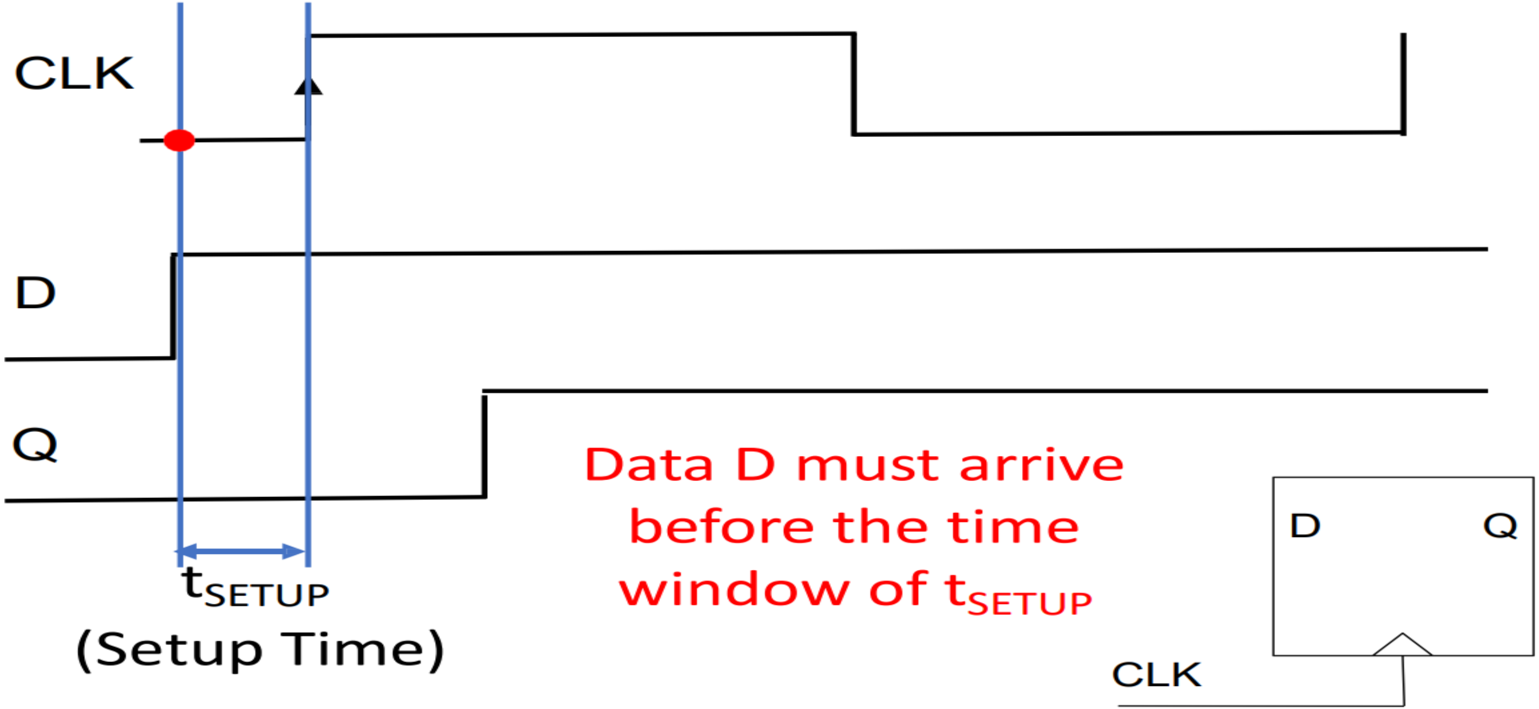


$t_{PFF} = t_{CLK \rightarrow Q} = t_{pcq}$

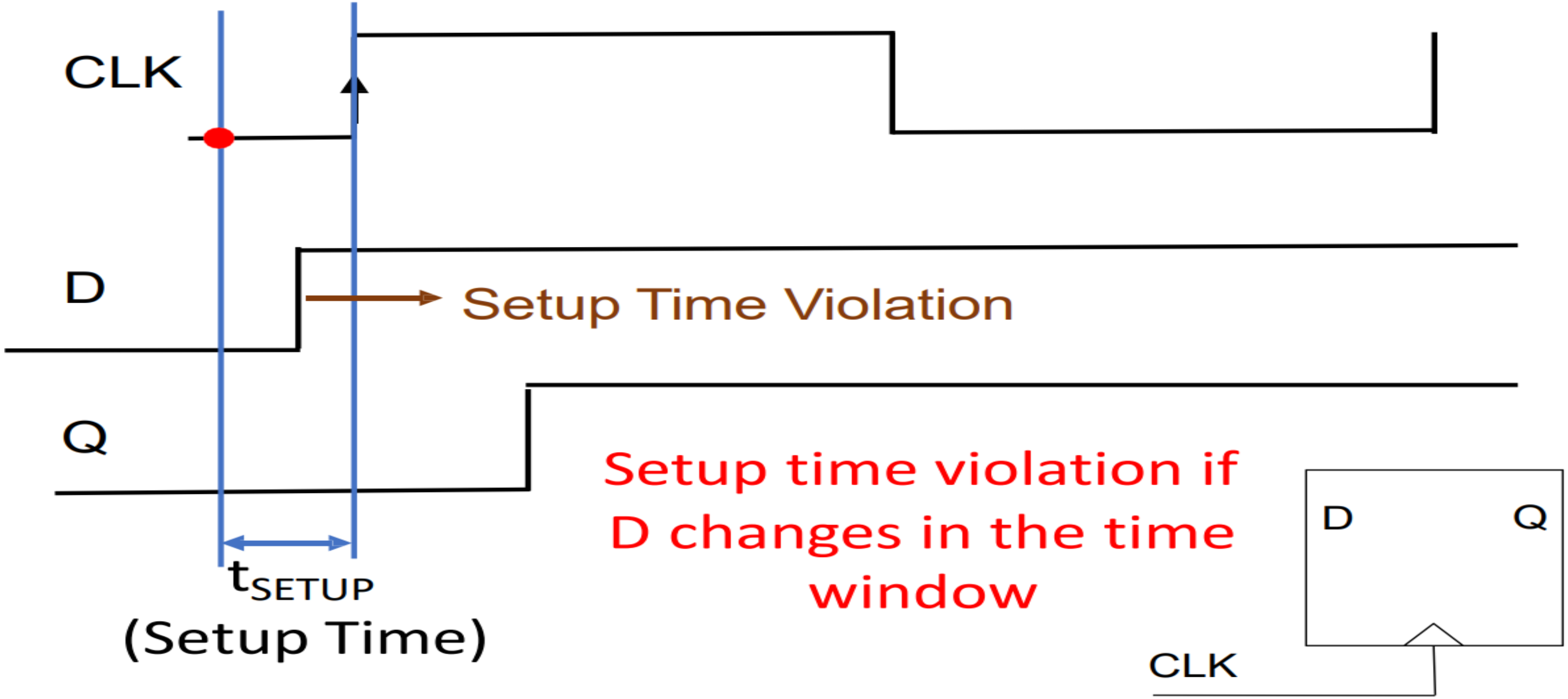
(Delay from CLK to Q)



TIMING METRICS: RISING EDGE TRIGGERED FLIP-FLOP

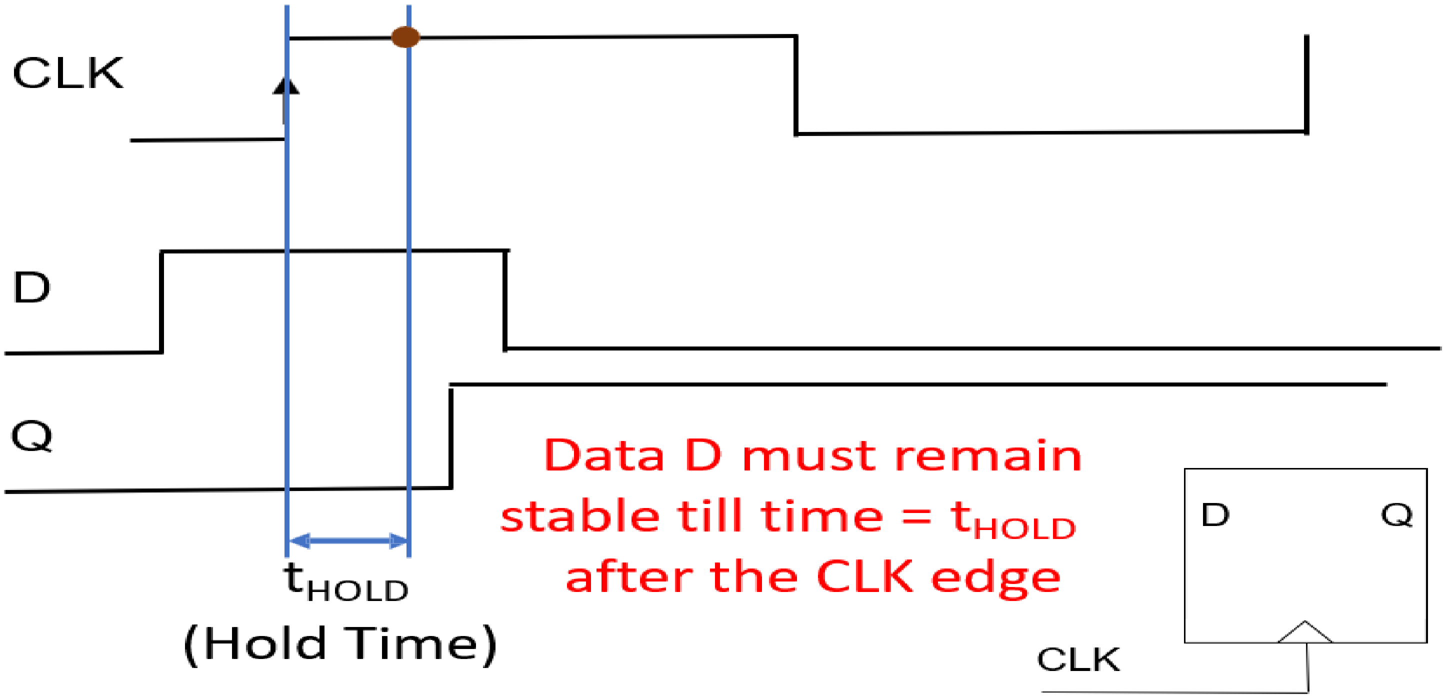


TIMING METRICS: RISING EDGE TRIGGERED FLIP-FLOP



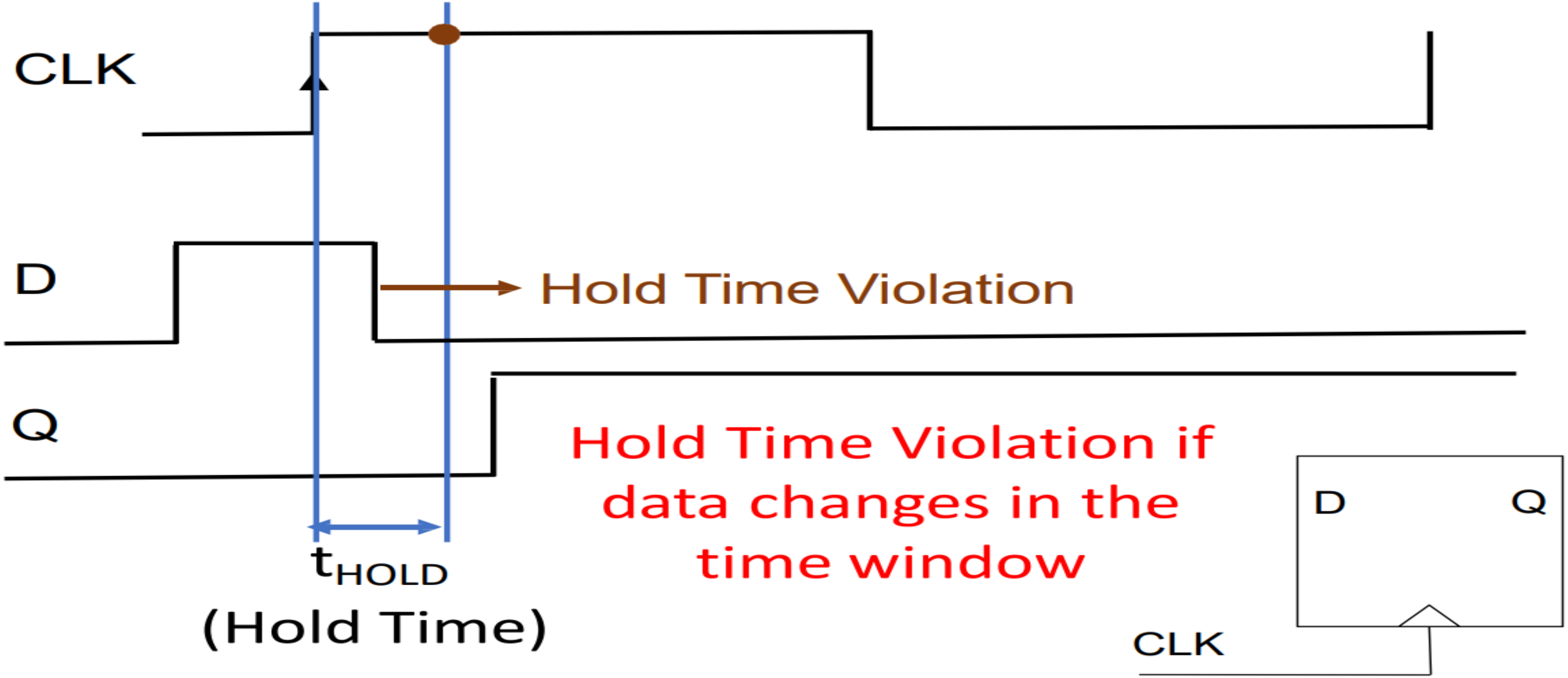
Setup time violation if D changes in the time window

TIMING METRICS: RISING EDGE TRIGGERED FLIP-FLOP



Data D must remain stable till time = t_{HOLD} after the CLK edge

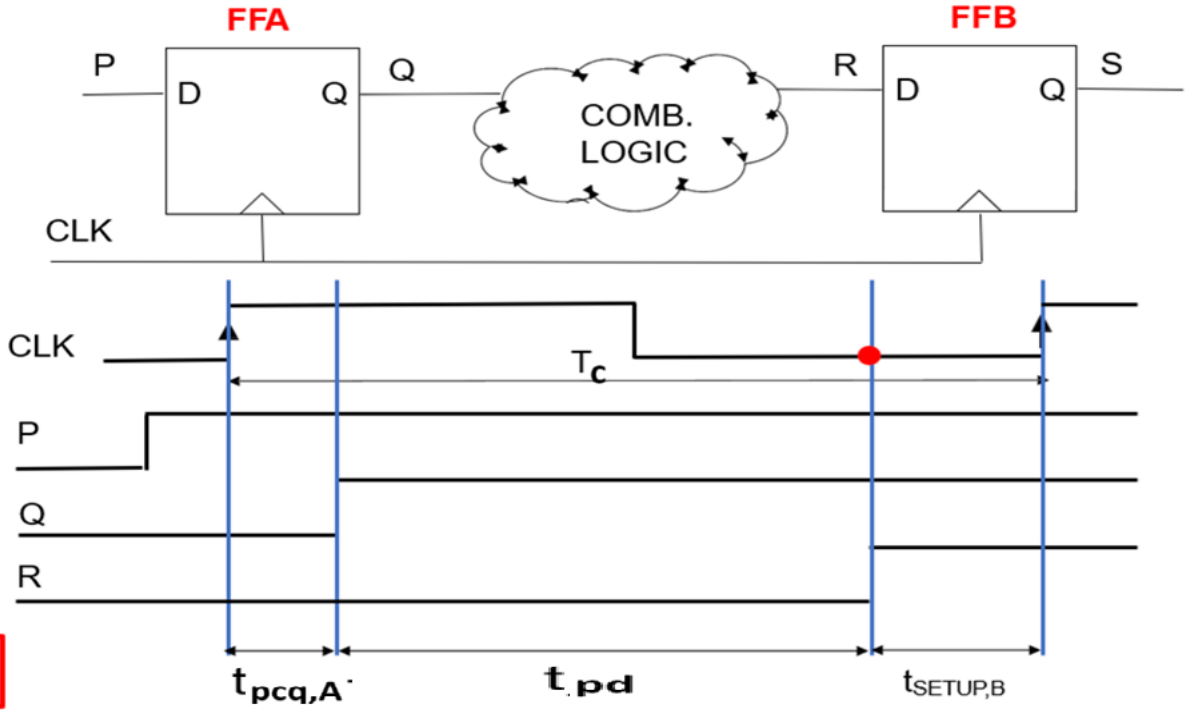
TIMING METRICS: RISING EDGE TRIGGERED FLIP-FLOP



TIMING CONSTRAINTS

- There are two main problems that can arise in synchronous logic:
  - Max Delay: The data doesn't have enough time to pass from one register to the next before the next clock edge.
  - Min Delay: The data path is so short that it passes through several registers during the same clock cycle.
- Max delay violations are a result of a slow data path, including the registers' t_{setup}, therefore it is often called the "Setup" path.
- Min delay violations are a result of a short data path, causing the data to change before the t_{hold} has passed, therefore it is often called the "Hold" path.

SETUP TIME CONSTRAINT



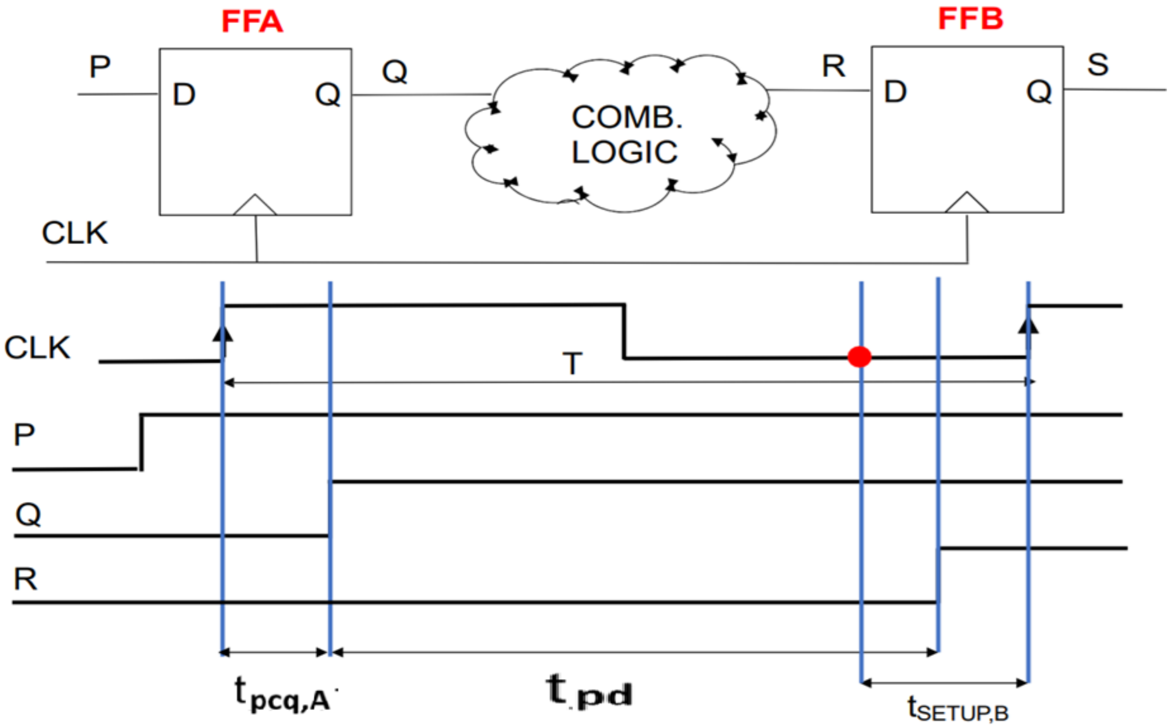
Must be ensured by design

$\boxed{T_c >= t_{pcq,A} + t_{pd} + t_{SETUPB}}$

SETUP TIME CONSTRAINT

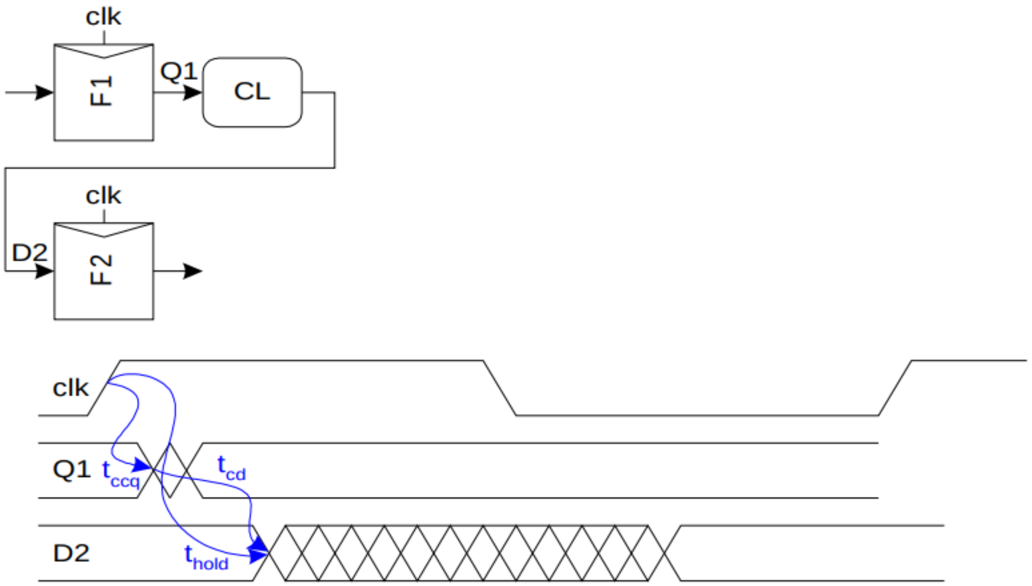
$$\boxed{T_c < t_{pcq,A} + t_{pd} + t_{SETUPB}}$$

Setup time violation!
Must be avoided



HOLD TIME CONSTRAINT

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

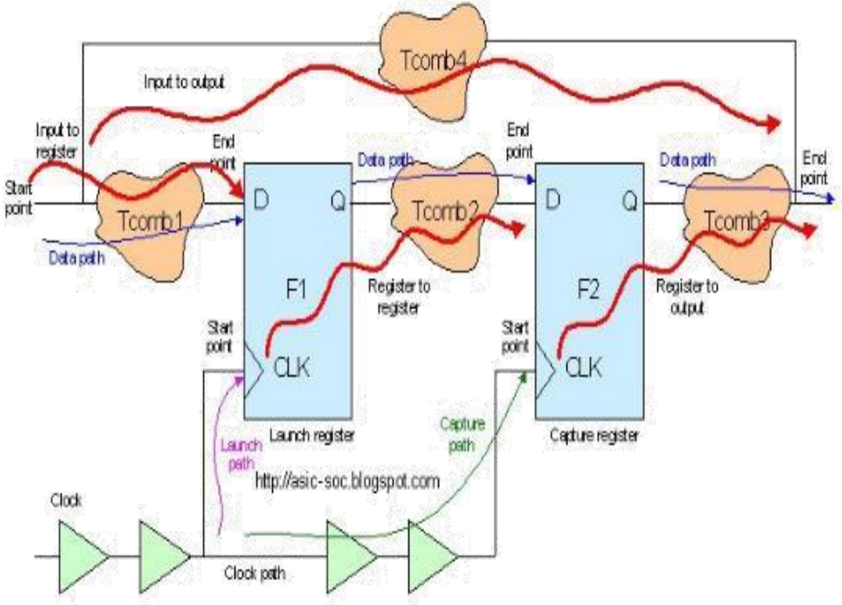


TIMING CONSTRAINTS: SUMMARY

- For Setup constraints, the data has to propagate fast enough to be captured by the next clock edge:
  - This sets our maximum frequency.
  - If we have setup failures, we can always just slow down the clock.
- For Hold constraints, the data path delay has to be long enough so it isn't accidentally captured by the same clock edge:
  - This is independent of clock period.
  - If there is a hold failure, you can throw your chip away!

STATIC TIMING ANALYSIS (STA)

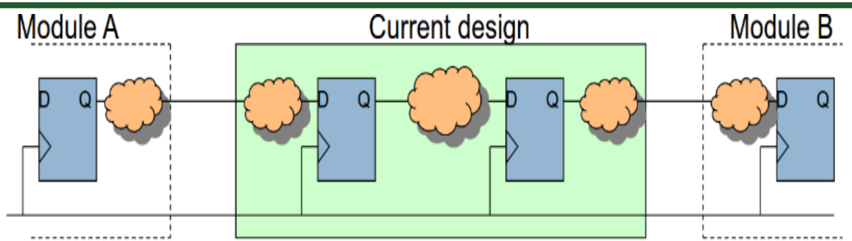
- STA checks the worst case propagation of all possible vectors for min/max delays.
- Advantages:
    - Much faster than timing-driven, gate-level simulation
    - Exhaustive, i.e., every (constrained) timing path is checked.
    - Vector generation NOT required



STATIC TIMING ANALYSIS (STA)

- Disadvantages:
  - Proper circuit functionality is NOT checked
  - Must define timing requirements/exceptions
- Limitations:
  - Only useful for synchronous design
  - Cannot analyze combinatorial feedback loops
    - e.g., a flip-flop created out of basic logic gates
  - Cannot analyze asynchronous timing issues
    - Such as clock domain crossing
  - Will not check for glitching effects on asynchronous pins
    - Combinatorial logic driving asynch (set/reset) pins of sequential elements will not be checked for glitching

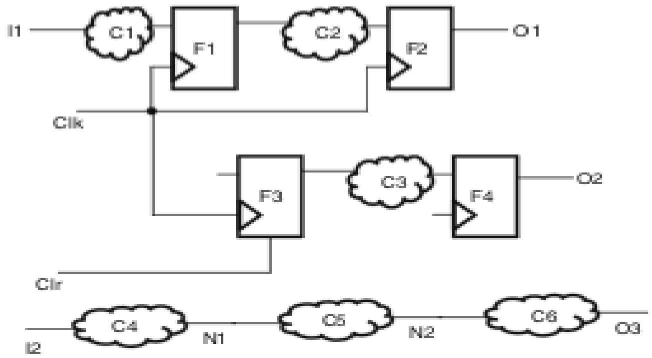
TIMING PATHS

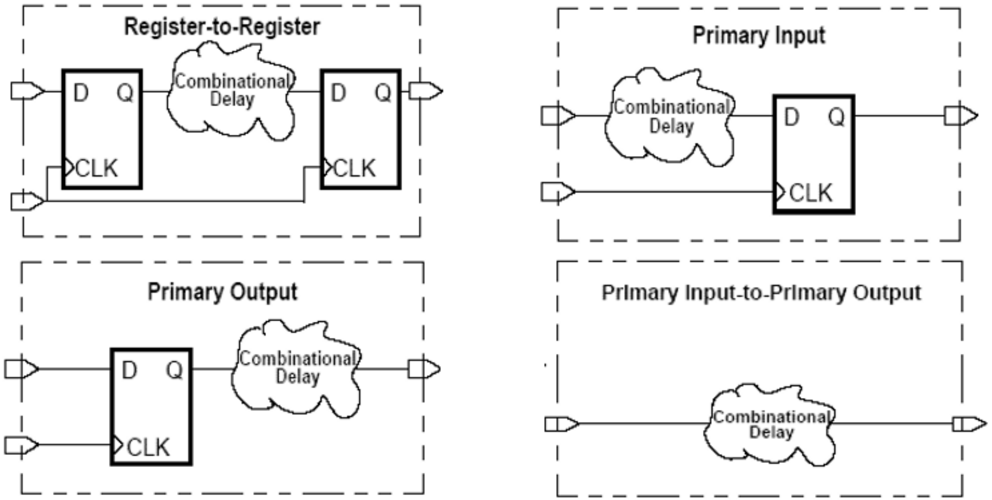


- A path is a route from a Startpoint to an Endpoint.
- Startpoint (SP)
  - Clock pins of the flip flops
  - Input ports , a.k.a Primary Inputs (PI)
- Endpoints (EP)
  - Input pins of the flip flops, except the clock pins
  - Output ports, a.k.a Primary Outputs (PO)
  - Memories / Hard macros
- There can be:
  - Many paths going to any one endpoint
  - Many paths for each start-point and end-point combination

STATIC TIMING ANALYSIS

- Four categories of timing paths
  - Register to Register (reg2reg)
  - Register to Output (reg2out)
  - Input to Register (in2reg)
  - Input to Output (in2out)





GOALS OF STATIC TIMING ANALYSIS

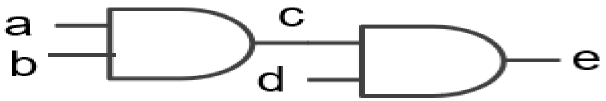
- Verify max delay and min delay constraints are met for all paths in a design.
  - Start with a Gate-Level Netlist.
  - Timing Models are provided for every gate in the library.
  - Static Timing Analysis needs to report if any path violates the max/min delay constraints.
- But is this enough?
  - No!
  - We want to know all the paths that violate the timing constraints.
  - In fact, we want to know the timing of all paths reported in order of length.
  - And we want to know where the problems are so we can go about fixing them.
- Let’s see the basic idea of how this can be done.

SOME BASIC ASSUMPTIONS

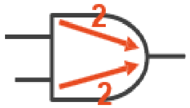
- Our design is synchronous
  - In addition, we will only be showing how to deal with combinational elements and max delay constraints.
- We will assume a pin-to-pin delay model
  - In other words, each gate has a single, constant delay from input to output.
  - In the real world, gate delay is affected by many factors, such as gate type, loading, waveform shape, transition direction, particular pin, and random variation.
  - As we saw earlier, a real design gets all this data from the .lib files.
- We will take a topological approach
  - In other words, we disregard the logical functionality of the gates and therefore, consider all paths, though some of them cannot logically happen.
  - More on this later…

SIMPLE PATH REPRESENTATION

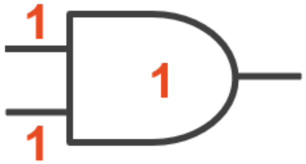
- Let’s say we have the circuit:



  - ➢ And the timing model of our AND gate is:

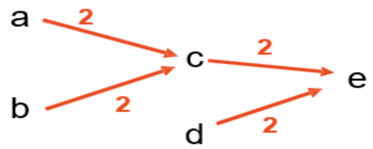


  - ➢ Sometimes also described as a combination of input wire delay and gate delay:

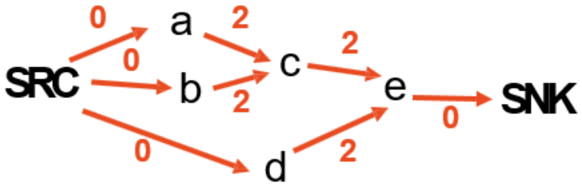


SIMPLE PATH REPRESENTATION

- We will build a graph:
  - Vertices: Wires, 1 per gate output and 1 for each SP and EP.
  - Edges: Gates, input pin to output pin, 1 edge per input with a delay for each edge.

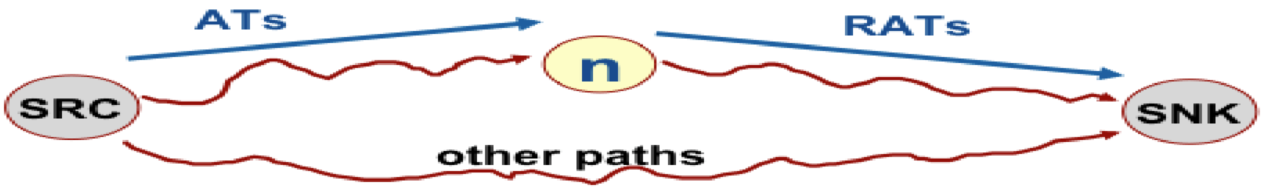


- Finally, add Source/Sink Nodes:
  - 0-weight edge to each SP and from each EP.
  - That way all paths start and end at a single node.



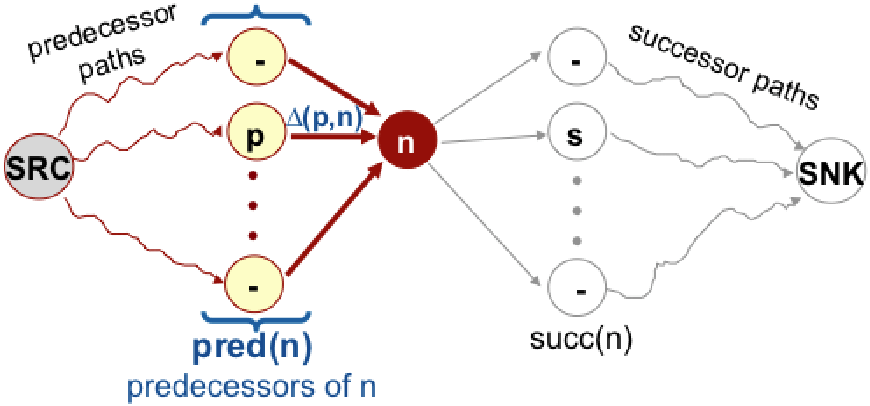
NODE ORIENTED TIMING ANALYSIS

- If we would enumerate every path, we would quickly get exponential explosion in the number of paths.
- Instead, we will use node-oriented timing analysis
  - For each node, find the worst delay to the node along any path.
- For this, we need to define two important values:
  - Arrival Time at a node (AT): 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.
- Slack at node n is defined as: *Slack(n) = RAT(n) – AT(n)*



HOW DO WE COMPUTE ATS AND RATS?

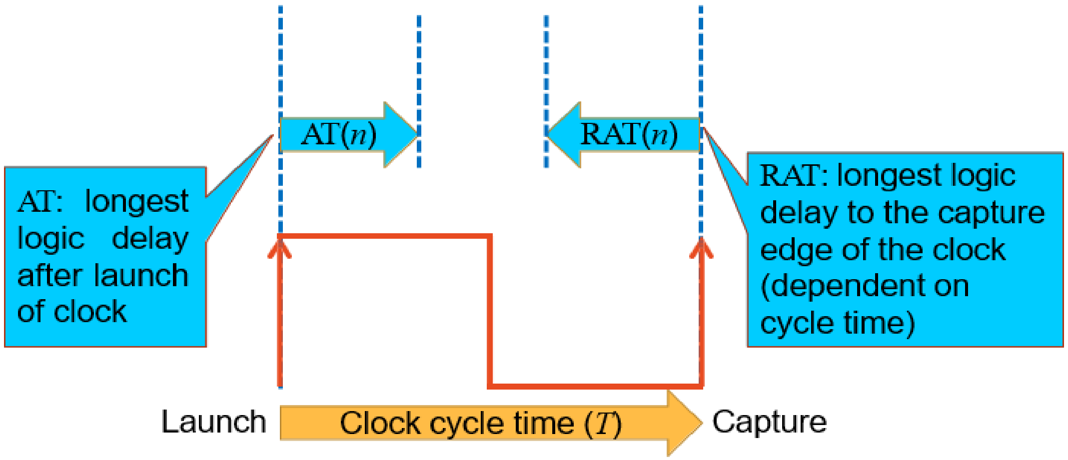
- Recursively!
  - The Arrival Time at a node is just the maximum of the ATs at the predecessor nodes **plus** the delay from that node.
  - The Required Arrival Time to a node is just the minimum of the RATs at the successor nodes **minus** the delay to that node.



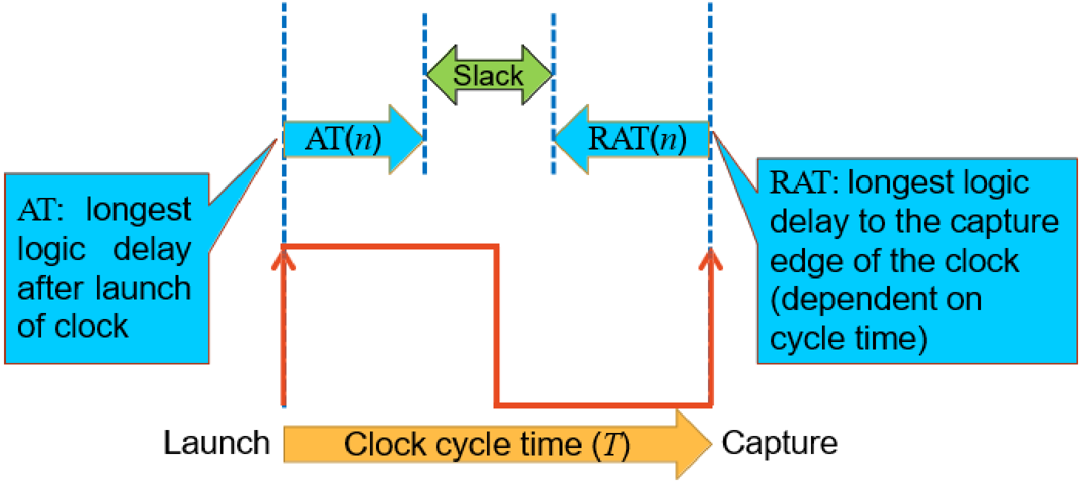
$$AT(n) = \begin{cases} 0 & n = SRC \\ \max\limits_{p \in pred(n)} \left[AT(p) + \Delta(p,n) \right] & n \neq SRC \end{cases}$$

$$RAT(n) = \begin{cases} T & n = SNK \\ \min\limits_{s \in succ(n)} \left[RAT(s) - \Delta(n,s) \right] & n \neq SNK \end{cases}$$

AT, RAT, AND SLACK: GRAPHICAL OVERVIEW

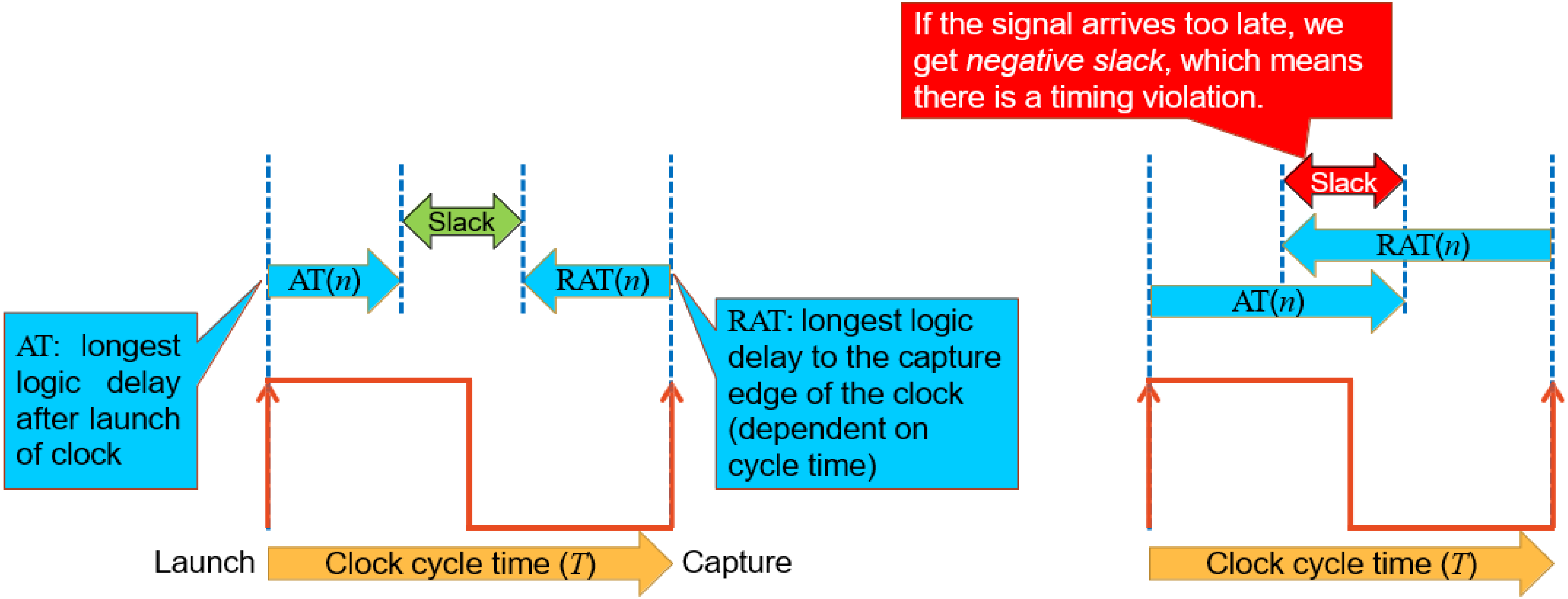


AT, RAT, AND SLACK: GRAPHICAL OVERVIEW



CAL POLY
SAN LUIS OBISPO

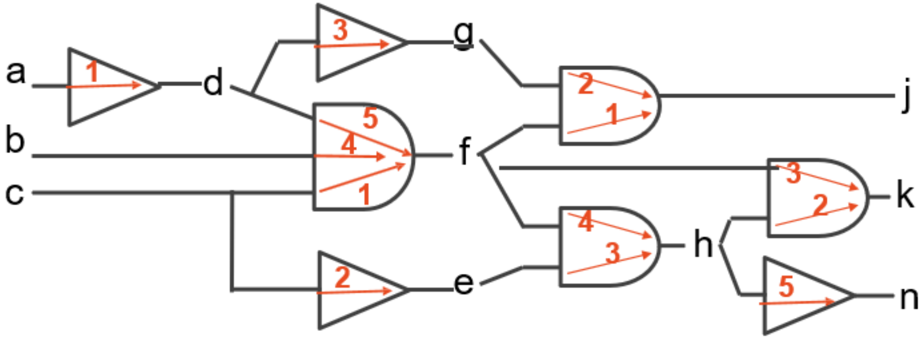
AT, RAT, AND SLACK: GRAPHICAL OVERVIEW



CAL POLY
SAN LUIS OBISPO

STA: EXAMPLE

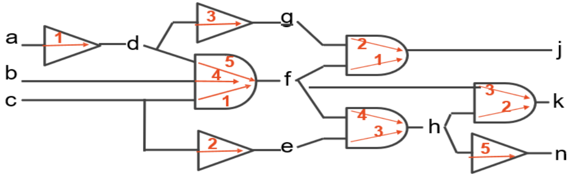
- Just look at this path and try to find the worst path.
  - Does it meet a cycle time of T=12 ?

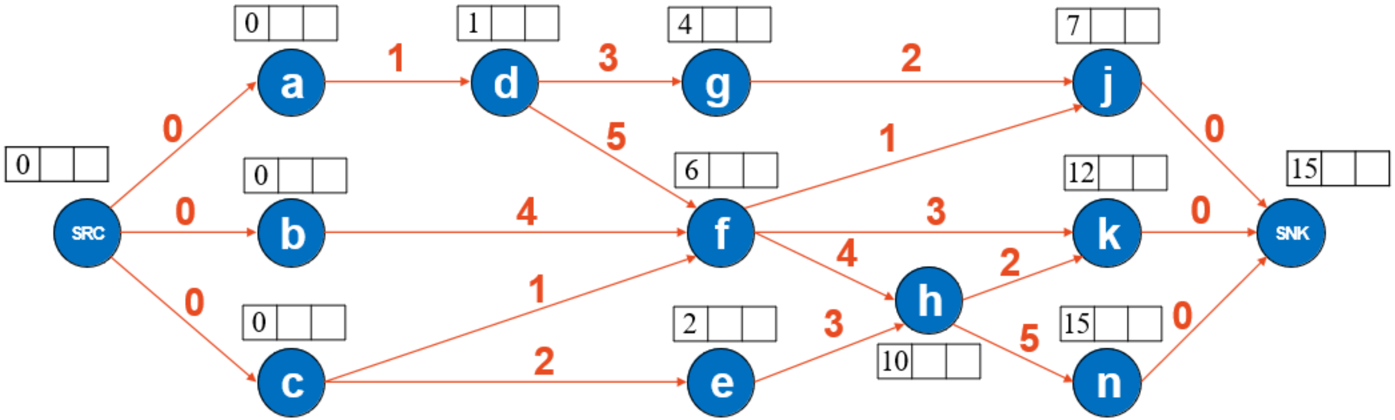


- Now let's fill in the RAT, AT, and SLACK of each node and:
  - Quickly find out if we meet timing
  - Figure out what the worst path is

STA: EXAMPLE

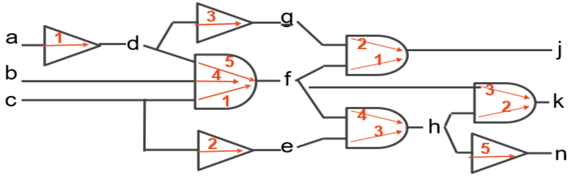
- We'll start by representing it as a directed acyclic graph (DAG)
- Next, we'll compute **ATs** from SRC to SNK

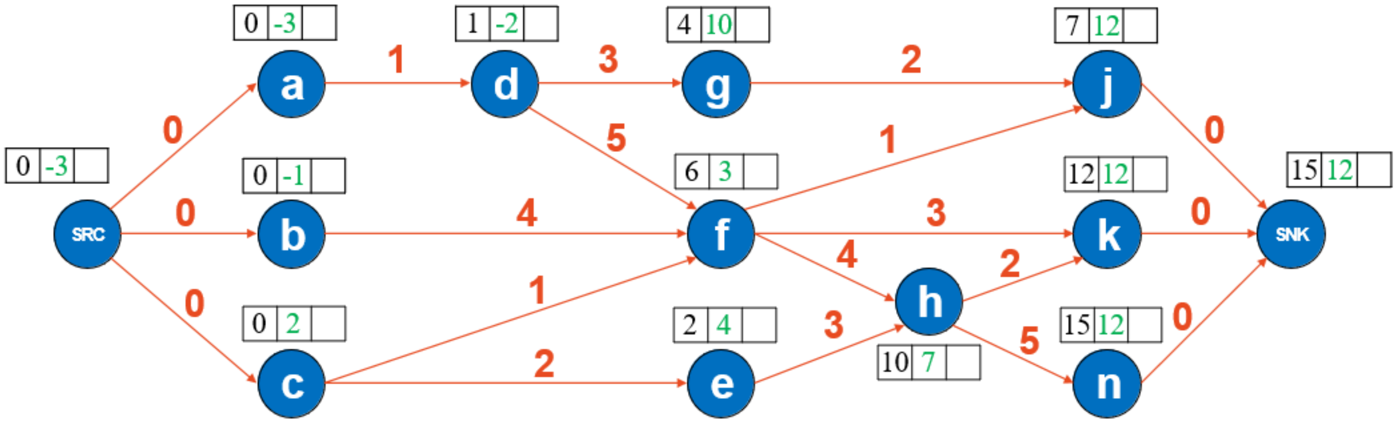




STA: EXAMPLE

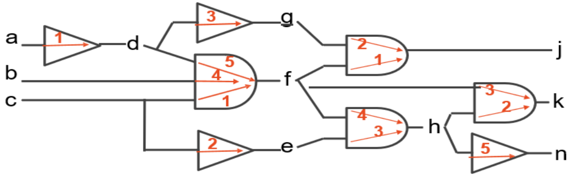
- And now RAT from SNK to SRC

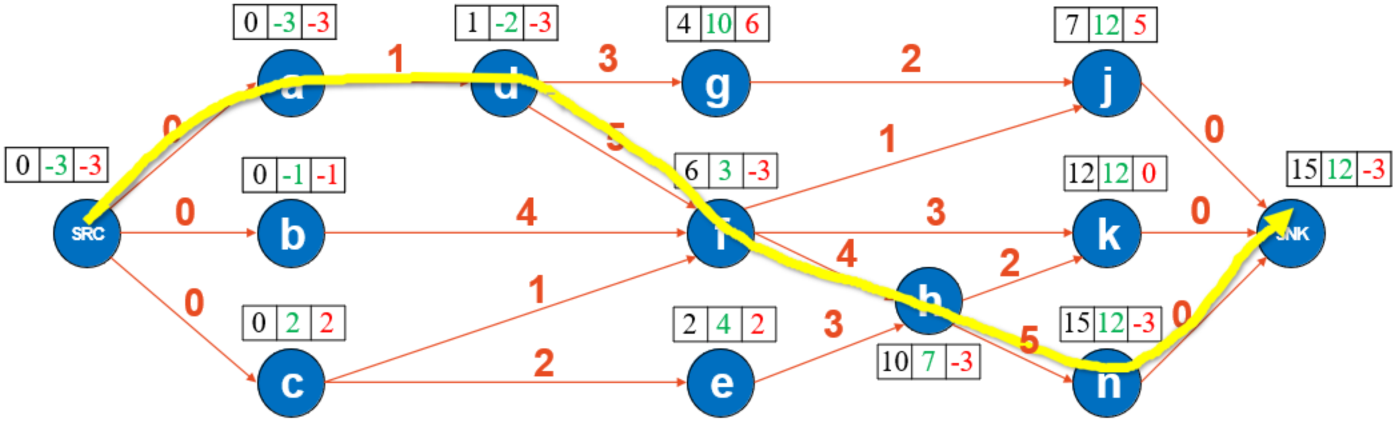




STA: EXAMPLE

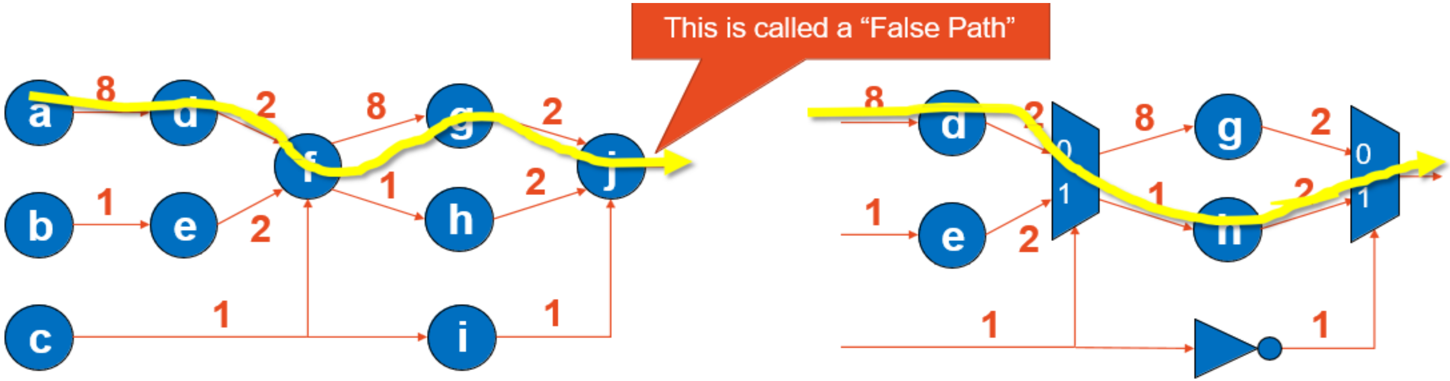
- And finally, we can calculate the slack.
- And guess what – we found the critical path!





FALSE PATHS

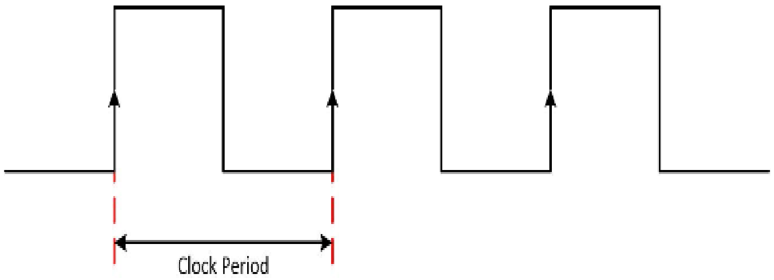
- We saw how to find the RAT, AT and Slack at every node.
  - All of this can be done very efficiently and be adapted for min timing, sequential elements, latch-based timing, etc.
  - Even better, we can quickly report the order of the critical paths.
- However, this was all done topologically (i.e., without looking at logic).
  - Let's see why this is a problem



Design Constraints: Revisited

TIMING CONSTRAINTS

- **Question:** How does the STA tool know what the required clock period is?
- **Answer:**
  - We have to tell it!
  - We have to define constraints for the design.
  - This is usually done using the Synopsys Design Constraints (SDC) syntax, which is a superset of TCL.
- Three main categories of timing constraints:
  - Clock definitions
  - Modeling the world external to the chip
  - Timing exceptions



TIMING CONSTRAINTS

- EDA tools sometimes use a different data structure called a “collection”
- A collection is similar to a TCL list, but:
  - The value of a collection is not a string, but rather a pointer, and we need to use special functions to access its values.
  - For example, if you were to run foreach on a collection, it would just have one element (the pointer to the collection). Instead, use foreach_in_collection.
  - I won’t go into the specifics here, but these are some of the collection accessing functions:

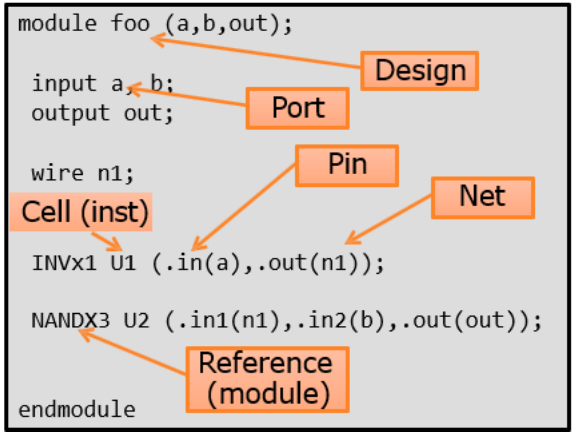
```
foreach_in_collection
index_collection
sizeof_collection
sort_collection
```

```
filter_collection
add_to_collection
compare_collections
```

```
copy_collection
get_object_name
remove_from_collection
```

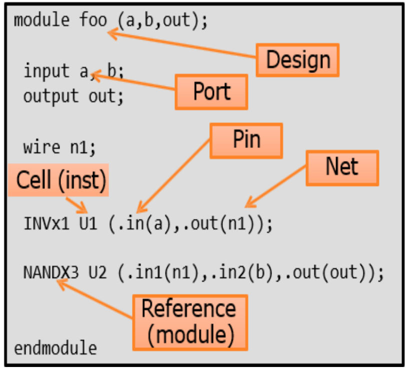
DESIGN OBJECTS

- **Design**: A circuit description that performs one or more logical functions (i.e Verilog module).
- **Cell**: An instantiation of a design within another design (i.e Verilog instance).
- **Reference**: The original design that a cell "points to" (i.e Verilog sub-module)
  - ➢ Called a module in Stylus Common UI.
- **Port**: The input, output or inout port of a Design.
- **Pin**: The input, output or inout pin of a Cell in the Design.
- **Net**: The wire that connects Ports to Pins and/or Pins to each other.
- **Clock**: Port of a Design or Pin of a Cell explicitly defined as a clock source.



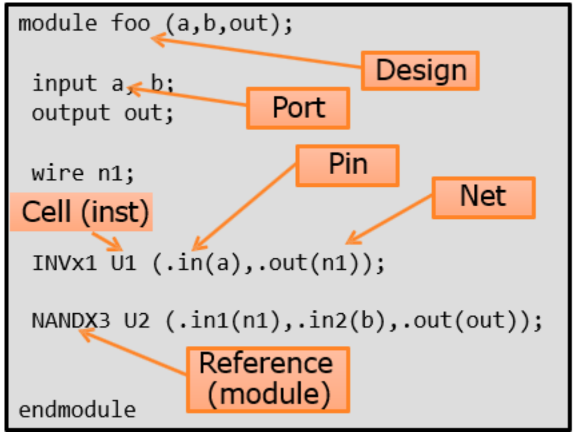
SDC HELPER FUNCTIONS

- Before starting with constraints, let's look at some very useful built in commands:
  - Note that all of these return collections and not TCL lists!
  - These will only work after design elaboration!
- "get" commands:
  - [get_ports string] – returns all ports that match string.
  - [get_pins string] – returns all cell/macro pins that match string.
  - [get_nets string] – returns all nets that match string.
    - Note that adding the –*hier* option will search hierarchically through the design.



SDC HELPER FUNCTIONS

- "all" commands:
  - [all_inputs] – returns all the primary inputs (ports) of the block.
  - [all_outputs] – returns all the primary outputs (ports) of the block.
  - [all_registers] – returns all the registers in the block.



CAL POLY
SAN LUIS OBISPO

CLOCK DEFINITIONS

- 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?

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

- Can there be more than one clock in a design?
  - Yes, but be careful about clock domain crossings!
  - If a clock is produced by a clock divider, define a "generated clock":

```
create_generated_clock -name gen_clock \
       -source [get_ports clk] -divide_by 2 [get_pins FF1/Q]
```

CLOCK DEFINITIONS

- But during synthesis, we assume the clock is ideal, so:

```
set_ideal_network [get_ports clk]
```

- However, for realistic timing, it should have some transition:

```
set_clock_transition 0.2 [get_clocks my_clock]
```

- And we may want to add some jitter, so:

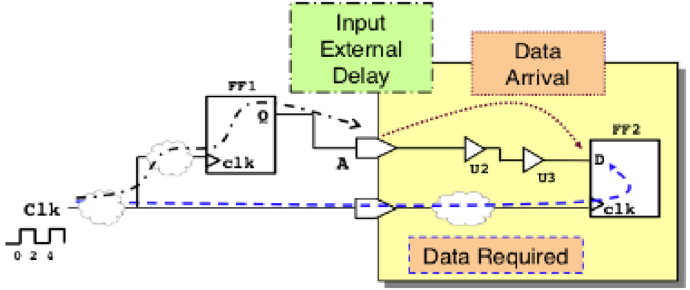
```
set_clock_uncertainty 0.2 [get_clocks my_clock]
```

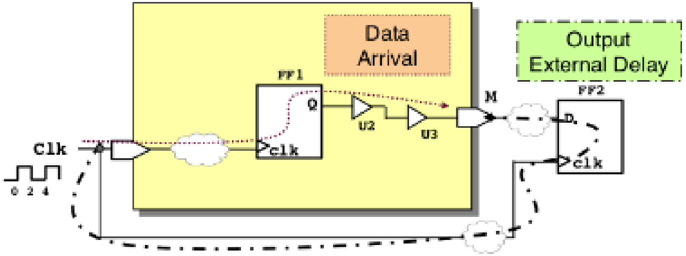
- Finally, after building a clock tree, we do not want the clock to be ideal anymore, so:

```
set_propagated_clock [get_clocks my_clock]
```

I/O CONSTRAINTS

- Now that the clock is defined, reg2reg paths are sufficiently constrained.
- However, what about in2reg, reg2out, and in2out paths?
  - First, what clock toggles an I/O port?
  - And what about the time needed outside the chip?
- Define I/O constraints:
  - Input and output delays model the length of the path outside the block:






```
set_input_delay 0.8 -clock clk \
        [remove_from_collection [all_inputs] [get_ports clk]]
set_output_delay 2.5 -clock clk [all_outputs]
```

I/O CONSTRAINTS

- An alternative approach is to define max delays to/from I/Os:

```
set_max_delay 5 \
        -from [remove_from_collection [all_inputs] [get_ports clk]]
set_max_delay 5 -to [all_outputs]
```

- Additionally, we must model the transitions on the inputs:

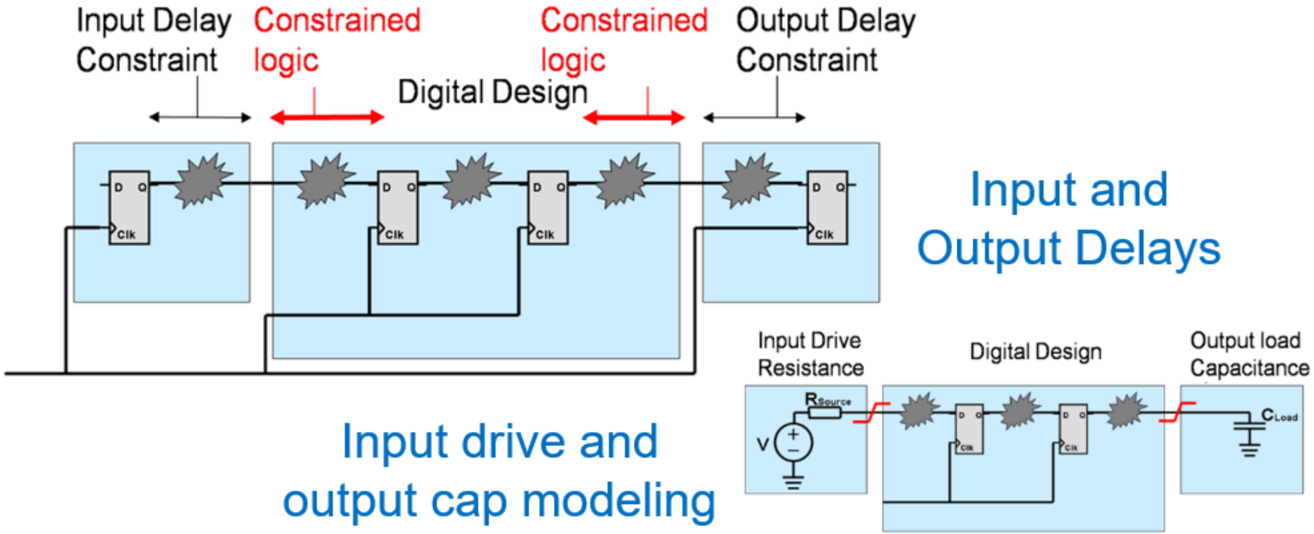
```
set_driving_cell -cell [get_lib_cells MYLIB/INV4] -pin Z \
        [remove_from_collection [all_inputs] [get_ports clk]]
```

- And capacitance of the outputs:

```
set_load $CIN_OF_INV [all_outputs]
```

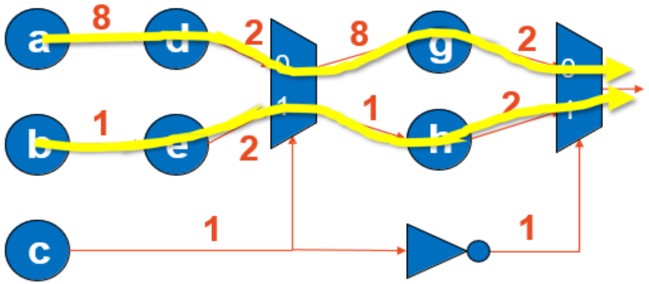
I/O CONSTRAINTS

- Graphically, we can summarize the I/O constraints, as follows:



TIMING EXCEPTIONS

- There are several cases when we need to define exceptions that should be treated differently by STA.
- For example, looking into the topology of the network we saw earlier:

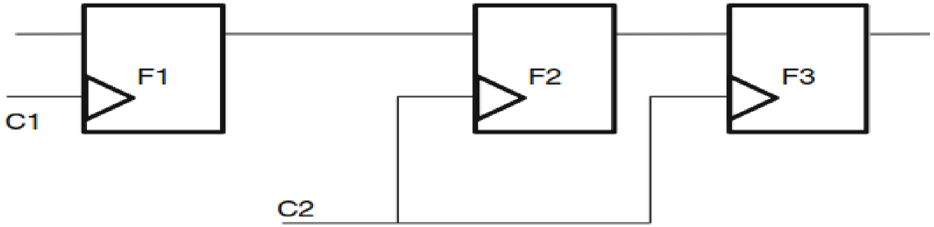


- In this case, we would define a false path:

```
set_false_path -through [get_pins mux1/I0] -through  [get_pins  mux2/I0]
set_false_path -through [get_pins mux1/I1] -through  [get_pins  mux2/I1]
```

TIMING EXCEPTIONS

- Another common case of a false path is a clock domain crossing through a synchronizer:



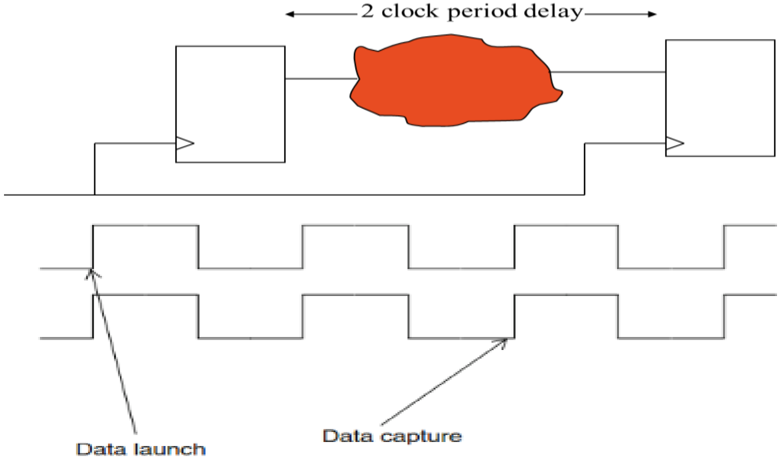

```
set_false_path –from F1/CP –to F2/D
```

- Alternatively, this can be defined with:

```
set_clock_groups –logically_exclusive \
        –group [get_clocks C1] –group [get_clocks C2]
```

TIMING EXCEPTIONS

- If an equal-phase (divided) slow clock is sending data to a faster clock, a multi-cycle path may be appropriate:




```
set_multicycle_path -setup -from F1/CP -to F2/D        2
set_multicycle_path -hold     -from F1/CP -to F2/D     1
```

CAL POLY
SAN LUIS OBISPO

TIMING EXCEPTIONS

- A common case for designs is that some value should be assumed constant
  - For example, setting a register for a certain operating mode.
- In such cases, many timing paths are false
  - For example, if the constant sets a multiplexer selector.
  - Or a ‘0’ is driven to one of the inputs of an AND gate.
- To propagate these constants through the design and disable irrelevant timing arcs, a set_case_analysis constraint is used:

```
set_case_analysis 0 [get_ports TEST_MODE]
```

DESIGN RULE VIOLATIONS (DRV)

- You can set specific design rules that should be met, for example:
  - Maximum transition through a net.

```
set_max_transition $MAX_TRAN_IN_NS
```

  - Maximum Capacitive load of a net.

```
set_max_capacitance $MAX_CAP_IN_PF
```

  - Maximum fanout of a gate.

```
set_max_fanout $MAX_FANOUT
```

Thank you!