What is STA (Static Timing Analysis) ?

Static Timing Analysis is a technique of analysing timing paths in a digital logic by adding up delays along a timing path (both gate and interconnect) and comparing it with constraints (clock period) to check whether the path meets the constraint.

Static timing analysis is a method for determining if a circuit meets timing constraints without having to simulate. So, it validates the design for desired frequency of operation, without checking the functionality of the design.

Static timing analysis is a method of validating the timing performance of a design by checking all possible paths for timing violations under worst-case conditions. *It considers the worst possible delay through each logic element, but not the logical operation of the circuit.*

: What are all the items that are checked by static timing analysis ?

A partial list of things it checks is here :

- Setup Timing
- Hold timing
- Removal and Recovery Timing on resets
- Clock gating checks
- Min max transition times
- Min/max fanout
- Max capacitance
- Max/min timing between two points on a segment of timing path.
- Latch Time Borrowing
- Clock pulse width requirements

What Is Setup Time?

Setup time is the amount of time before the clock edge that the input signal needs to stable to guarantee it is properly accepted on the clock edge.

What Is Hold Time?

Hold time is the amount of time after the clock edge that the input should be stable to guarantee it is properly accepted on the clock edge.

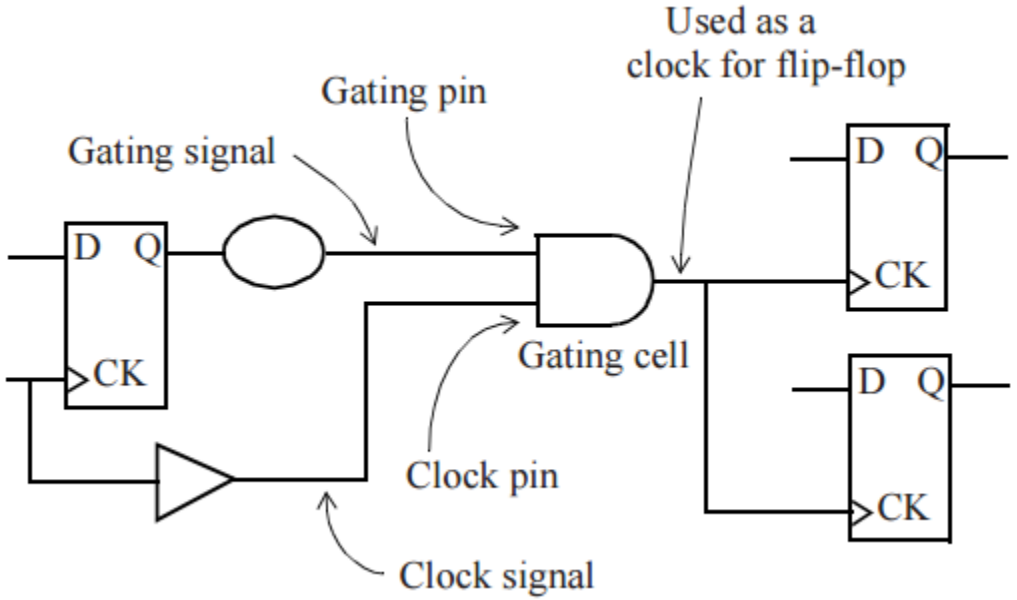
Removal and Recovery Timing on resets ?

Recovery Time is the minimum required time to the next active clock edge the after the reset (or the signal under analysis) is released

Removal Time is the minimum required time after the clock edge after which reset can be released.

Clock gating checks

A clock gating check occurs when a **gating signal** can control the path of a **clock signal** at a logic cell. An example is shown in Figure 1. The pin of logic cell connected to clock is called **clock pin** and pin where gating signal is connected to is **gating pin**. Logic cell where clock gating occurs is also referred to as **gating cell**.



Min max transition times

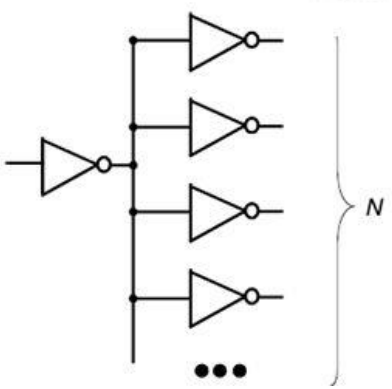
The transition time is **the time needed for a signal to pass from 10% to 90% or from 20% to 80% of its final state**.

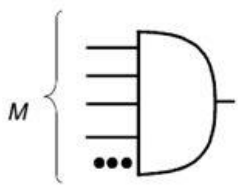
Min/max fanout

Fan-In and Fan-Out

- The fan-in of a gate is defined as the number of inputs to the gate.
- The fan-out denotes the number of load gates N that are connected to the output of the driving gate.
- Increasing the fan-out of a gate can affect its logic output levels.

$$Fanout = \frac{I_{OL(\max)}}{I_{IL(\max)}} \quad \text{หรือ} \quad Fanout = \frac{I_{OH(\max)}}{I_{IH(\max)}}$$





(b) Fan-in *M*

How can I reduce my FPGA power consumption?

One of the more powerful techniques is to **maximize the use of hard IP blocks available on chip**, because FPGA vendors design the hard IP to use only the exact resources required to achieve a given protocol or architecture. Another technique is to simply suspend all or part of the FPGA when it's not in use.

Describe a timing path

Timing path is defined as **the path between start point and end point** where start point and end point is defined as follows: Start Point: All input ports or clock pins of a sequential element are considered as valid start point. End Point: All output port or D pin of sequential element is considered as End point.

What is timing critical path?

The critical path is defined as **the path between an input and an output with the maximum delay**.

The Way STA is performed on a given Circuit:

To check a design for violations or say to perform STA there are 3 main steps:

- Design is broken down into sets of timing paths,
- Calculates the signal propagation delay along each path
- And checks for violations of timing constraints inside the design and at the input/output interface.

Timing Paths:

Timing paths can be divided as per the type of signals (e.g clock signal, data signal etc).

Types of Paths for Timing analysis:

- Data Path
- Clock Path
- Clock Gating Path
- Asynchronous Path

http://www.vlsi-expert.com/2011/03/static-timing-analysis-sta-basic-timing.html

Other types of Paths:

There are few more types of path which we usually use during timing analysis reports. Those are subset of above mention paths with some specific characteristics. Since we are discussing about the timing paths, so it will be good if we will discuss those here also.

Few names are

- Critical path
- False Path
- Multi-cycle path
- Single Cycle path
- Launch Path
- Capture Path
- Longest Path (also know as Worst Path, Late Path, Max Path , Maximum Delay Path)
- Shortest Path (Also Know as Best Path, Early Path, Min Path, Minimum Delay Path)

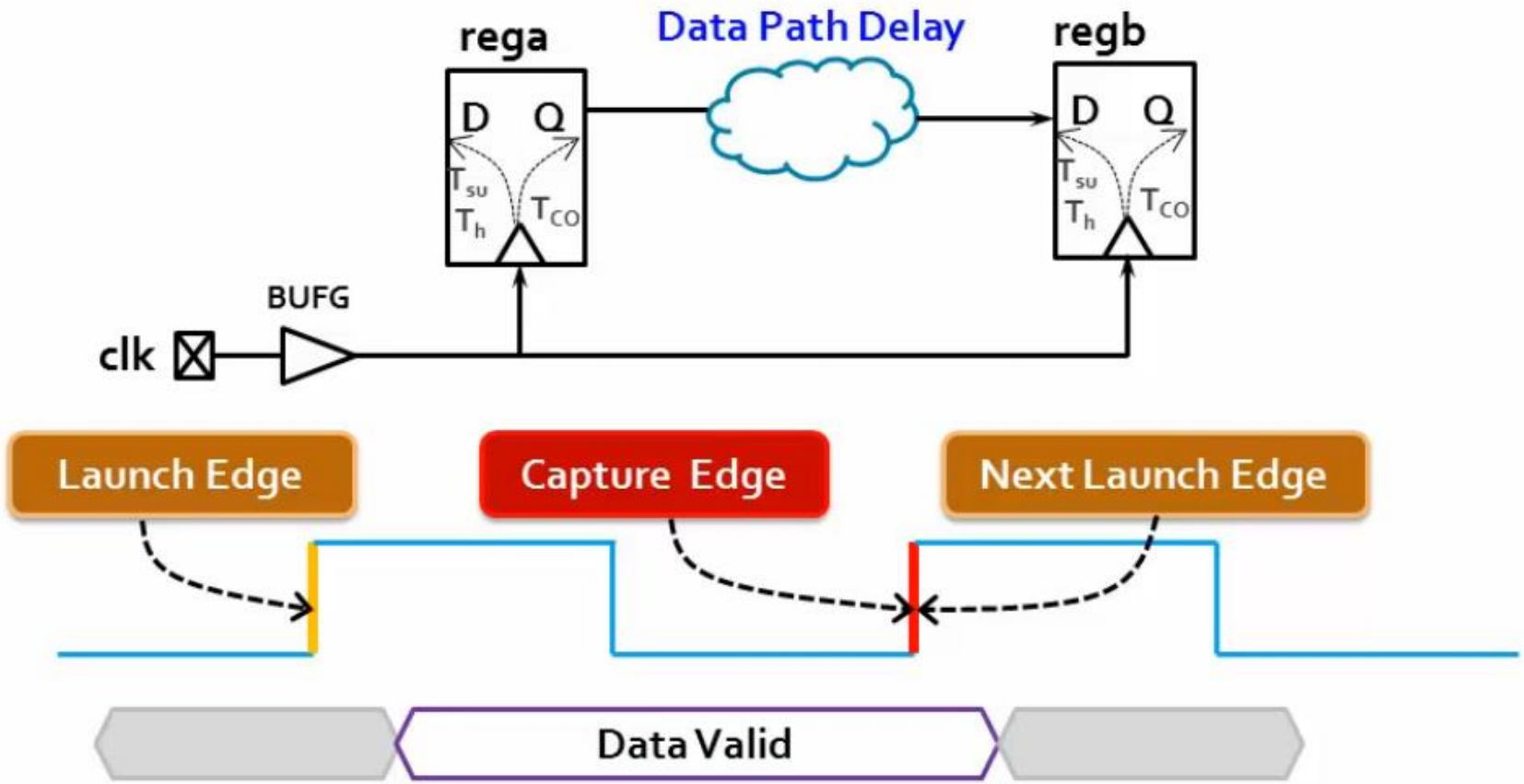
Latch Time Borrowing

Time Borrowing is permitting the logic to automatically borrow time from next cycle, thereby reducing the time available for data to arrive for the following cycle OR permitting the logic to use slack from the previous cycle, in the current cycle

Clock pulse width requirements

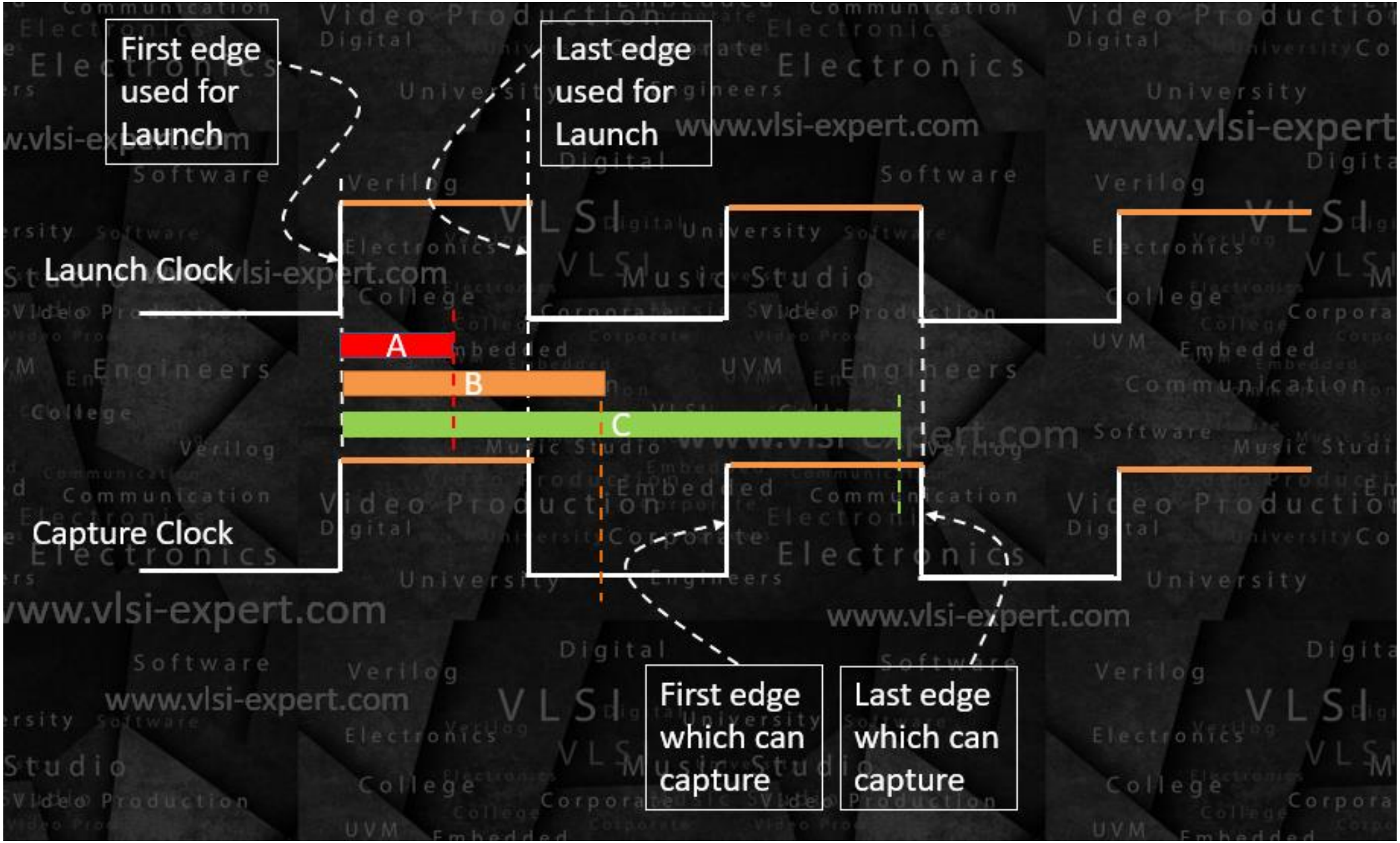
Minimum pulse width checks are done **to ensure that width of the clock signal is wide enough for the cell's internal operations to complete**. i.e. to get a stable output you need to ensure that the clock signal at the clock pin of the flop is at least of a certain 'minimum' width

Launch vs. Capture Edges



Launch Edge : the edge which "launches" the data from source register

Capture Edge : the edge which "captures" the data at destination register (with respect to the launch edge, selected by timing analyzer; typically 1 cycle)


First edge used for Launch
Last edge used for Launch
Launch Clock
A
B
C
Capture Clock
First edge which can capture
Last edge which can capture

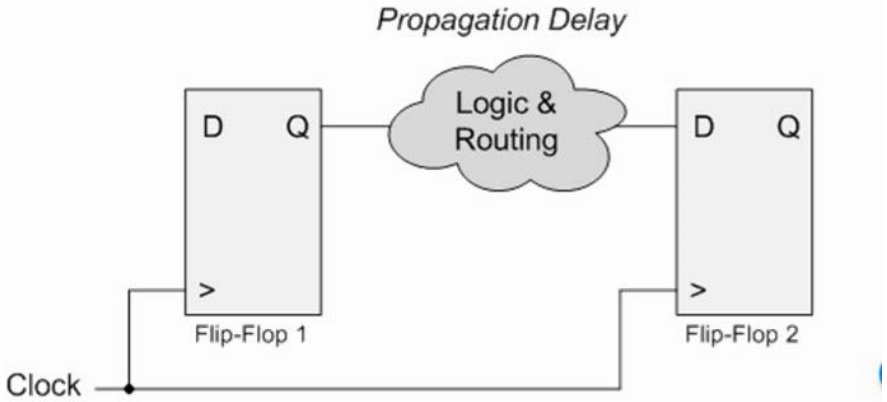
What does the setup/Hold time of a flop depend upon ?

Setup/Hold time of a flip-flop depends upon the Input data slope, Clock slope and Output load.

Explain signal timing propagation from one flip-flop to another flip-flop through combinational delay.

Propagation Delay

- Time it takes for a signal to travel from a source to a destination
- Voltages in wires take time!
- The longer the propagation delay, the slower your clock is able to run.



Explain setup failure to a flip-flop

if we reduce frequency, our cycle time increases and eventually FF2_in will be able to make it in time and there will not be a setup failure. Also notice that a clock skew is observed at the second flop. The clock to second flop clk2 is not aligned with clk1 anymore and it arrives earlier, which exacerbates the setup failure.

What is design sign off?

Sign off **typically implies that the design is final and no further revisions are possible**.

If hold violation exists in design, is it OK to sign off design? If not, why?

No you can not sign off the design if you have hold violations. Because hold violations are functional failures. Setup violations are frequency dependent. You can reduce frequency and prevent setup failures. Hold violations stemming from the same clock edge race, are frequency independent and are functional failures because you can end up capturing unintended data, thus putting your state machine in an unknown state.

What are setup and hold checks for clock gating and why are they needed ?

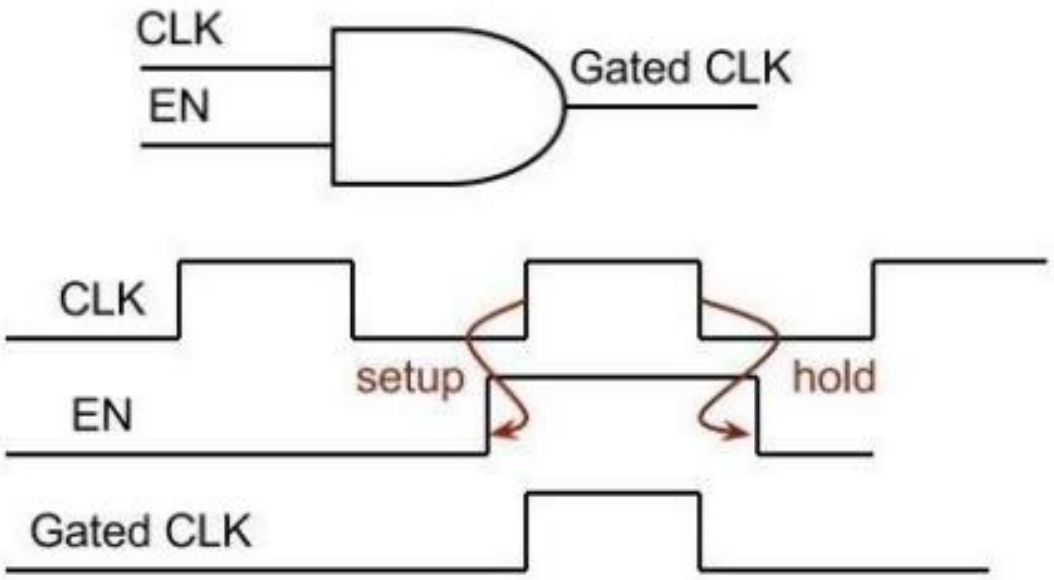


Figure S13. Clock gating setup and hold check

What determines the max frequency a digital design will work on. Why hold time is not included in the calculation for the above ?

Worst max margin will decide the max frequency a design will work on. As setup failure is frequency dependent. Hold failure is not frequency dependent hence it is not factored into the frequency calculation.

One chip which came back after being manufactured fails setup test and another one fails a hold test. Which one may still be used how and why ?

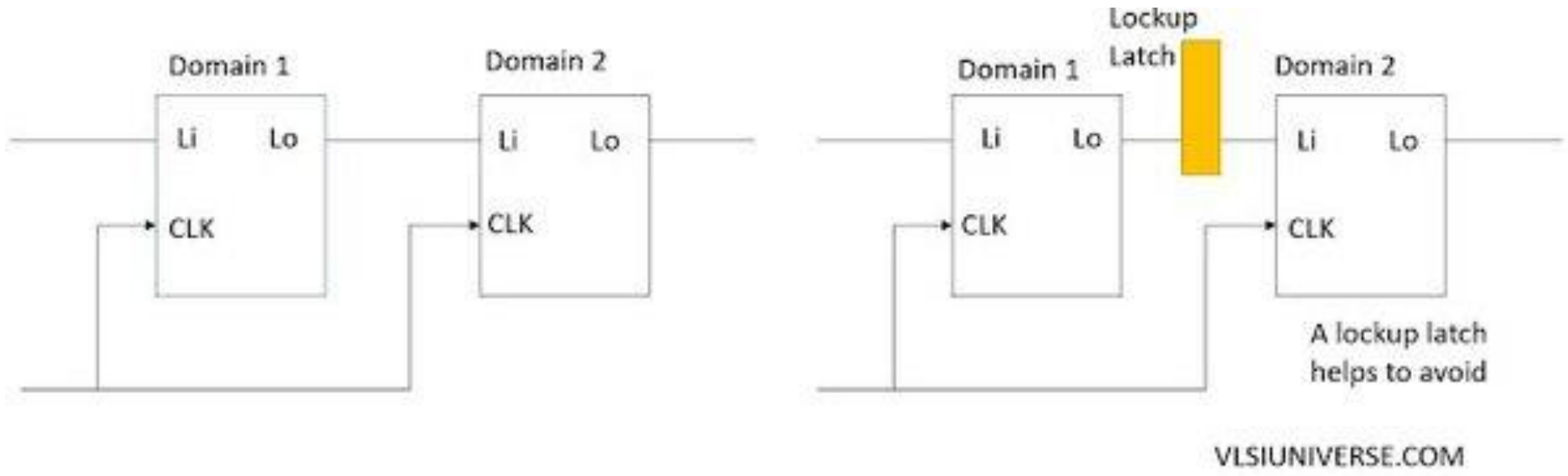
Setup failure is frequency dependent. If certain path fails setup requirement, you can reduce frequency and eventually setup will pass. This is because when you reduce frequency you provide more time for the flop/latch input data to meet setup. Hence we call setup failure a frequency dependent failure. While hold failure is not frequency dependent. Hold failure is functional failure.

: Are clock domain crossing issues detected by STA tool ?

No clock Domain crossing issues are not detected by Static Timing Analysis tool. As mentioned earlier, tool simply tries to find out the worst case setup and hold checks between launch and capture edge. Designer has to design for clock domain crossings.

What are these lockup latches?

Lockup latch is simply a transparent latch (D Latch). These lockup latches are used in scan-based designs, i.e., in between to scan flip flops which have large probability of hold failure. The lockup latches are used to avoid large clock skew problems.

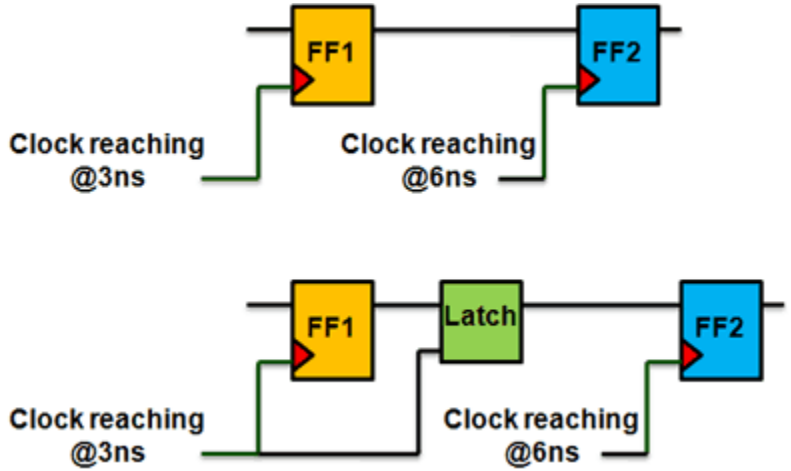


Difference between lockup latches and lockup registers.

Both lockup latch and lockup register do serve the same purpose, but they do have some differences. Due to which one is preferred among the other one. Let see who wins the quiz.
Coming to the area perspective a lockup latch is half the size of a lockup register. So, lockup latch is area efficient compared to the use of lockup registers. Also, we can say that lockup latches are power efficient by considering the same point.

Advantages of inserting lockup latches

First this by inserting lockup latches we can do timing closure for hold failure during SCAN-SHIFT mode. That is the case when there is a large, uncommon path between launch and capture flip flops. This scenario is AREA efficient and POWER efficient when compared to lockup registers, which will be discussed in the later section.



Does location of lockup latch matter ?

The location of lockup latch very much matters. When you introduce lockup latch in between two flops, you are essentially breaking timing path into two segments. One path from the original launch flop to the lockup latch and other timing path from the lockup latch to the original capture flop.

What are your options to fix a timing path ?

There are several different possibilities for fixing a timing path.

- Obvious logic optimization
- Better placement
- - More pipelining.
- Move logic to previous pipe stage ?
- Replicate drivers
- Parallelism in RTL
- Use of Macro.
- Synthesized if...elseif...elseif series.
- One Hot instead of Binary coded State Registers
- Physical design techniques.
- Power trade off techniques

What are multi cycle paths ?

Multicycle paths are **data paths between two registers that operate at a sample rate slower than the FPGA clock rate and therefore take multiple clock cycles to complete their execution**.

What is a false path?

False Paths are **those timing arcs in design where changes in source registers are not expected to get captured by the destination register within a particular time interval**.

What is signal integrity ?

Signal Integrity is **the ability of an electrical signal to carry information reliably and resist the effects of high-frequency electromagnetic interference from nearby signals**

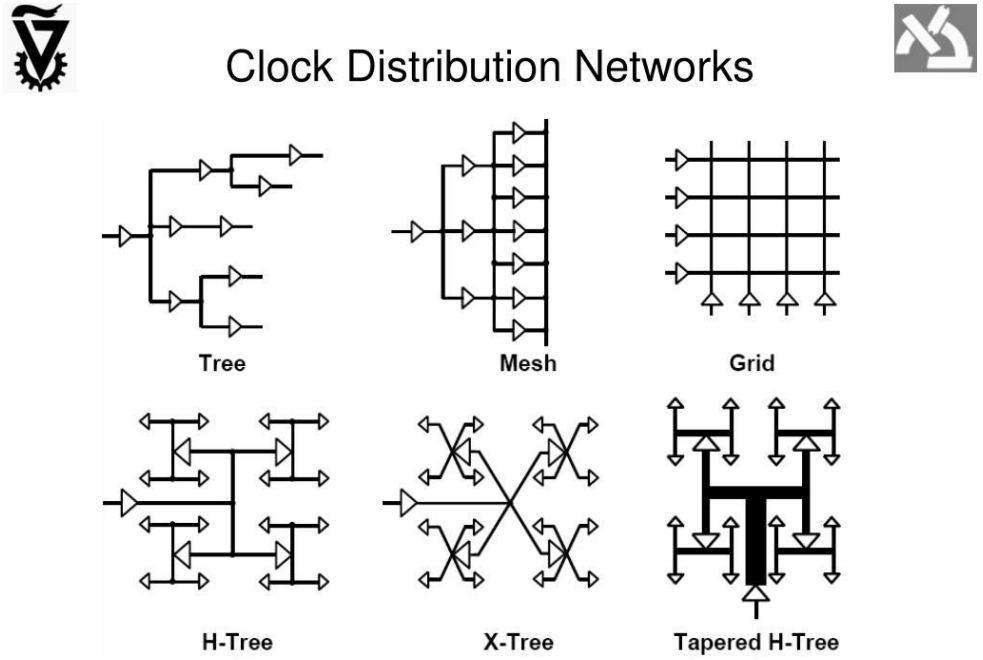
What is crosstalk glitch and how do you fix it ?

1. Break the victim net by inserting an buffer in between .
2. Provide sufficient spacing between the victim and aggressor net to reduce coupling capacitance.
3. Route the victim / aggressor net in a different metal layer to remove coupling capacitance.
4. Strengthen the signal on the victim net by using a lower VT or higher drive strength cell .
5. Weaken the strength of the aggressor net cell . (Least preferred) .

: What are the main clock distribution styles used ?

In digital designs there are two main clock distribution styles that are used.

1) Clock mesh or Clock grid distribution system.
2) 2) CTS(Clock tree synthesis), or Clock tree distribution system



What is a clock tree synthesis?

Clock Tree Synthesis (CTS) is **the technique of balancing the clock delay to all clock inputs by inserting buffers/inverters along the clock routes of an ASIC design**. As a result, CTS is used to balance the skew and reduce insertion latency

Why is clock gating done ?

As you can see in the figure by clock gating we can mask certain clock pulses, or in other words we can control the clock toggling activity. Clock is usually a very high fanout signal which is distributed throughout the chip. Because it usually drives large number of elements and normally continuously toggles, it account for major portion of dynamic power dissipation of the chip. Ability to turn off clock toggling when not needed is the most effective dynamic power saving mechanism. From timing perspective, one has to ensure that clock gating doesn’t introduce glitches or changes the shape of the clock pulse. There are setup and hold checks to ensure this.

What is metastability and what are its effects ?

In metastable states, **the circuit may be unable to settle into a stable '0' or '1' logic level within the time required for proper circuit operation**. As a result, the circuit can act in unpredictable ways, and may lead to a system failure, sometimes referred to as a "glitch".

: How do you synchronize between 2 clock domains?

Common methods for synchronizing data between clock domains are:

- Using m-FF based synchronizers.
- Using MUX based synchronizers.
- Using Handshake signals.
- Using FIFOs (First In First Out memories).
- Using Toggle synchronizers
- Using Xilinx specific clock domain crossing (CDC) tools
- Using PLL and DLL

How is FIFO depth/size determined ?

Size of the FIFO depends on both read and write clock domain frequencies, their respective clock skew and write and read data rates. Data rate can vary depending on the two clock domain operation and requirement and frequency. FIFO has to be able to handle the case when data rate of writing operation is maximum and for read operation is minimum.

Design a flip flop using a mux.

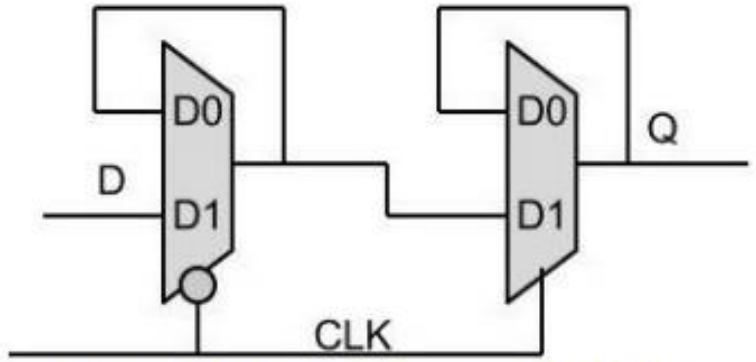


Figure MI1c. Flip-flop using 2:1 MUX

How will a flip flop respond if the clock and D input of a D flipflop are shorted and clock connected to this shorted input?

One can expect this flip flop to be in metastable state most of the time. Because with clock and data input tied together, every time clock rises the data will also rise and will definitely violate the setup time and hold time for that flip flop. With the continuous violation of setup and hold time we can expect flip flop to be in metastable state for at least very large amount of time.

: How do you fix timing path from latch to latch ?

Latch to latch setup time violation is fixed just like flop to flop path setup time violation, where you either speed up the data from latch to latch by either optimizing logic count, or speeding up the gate delays and/or speeding up wire delays

What is the difference between a latch and a flip-flop

Latch is level sensitive device, while flip-flop is edge sensitive. Actually a D flip-flop is made from two back to back latches, in master-slave configuration

What happens to delay if you increase load capacitance?

Usually device slows down if load capacitance is increased. Device delay depends on three parameters, 1) the strength of device, which usually is the width of the device 2) input slope or slew rate and 3) output load capacitance. Increasing strength or the width increases self load as well.

Handling timing closure

- Placement: You can use pblock constraints to try to guide placement. This is hit or miss and is beyond the scope of this book.
- Routing: If you have a very high utilization, routing can become congested. Trying to redesign some paths to ease timing may help.
- Too much logic in a path: This can be resolved by adding pipelining or breaking up long paths if possible.
- Lack of DSP pipelining: This falls under the previous bullet of too much logic. If you are trying to do more than the DSP can handle without using some of the internal resources, you may need to evaluate adding pipeline stages

How to pipeline a design

We've seen that too much logic in a path can cause timing closure problems.

This can be addressed in a few ways:

- Using aggressive optimizations
- Pipelining logic
- Pipelining DSP elements

Performance Metrics for Circuits

Performance Metrics for Circuits

Circuit Latency (L): time between arrival of new input and generation of corresponding output.

For combinational circuits this is just t_{PD}.

Circuit Throughput (T): Rate at which new outputs appear.

For combinational circuits this is just $1/t_{PD}$ or $1/L$.

What is speed grade and how do you select FPGA as per requirements

Speed Grade is what that determines how max a clock can run in FPGA. Companies use different values, while -1, -2 indicates the scale. Higher the grade, higher the cost of FPGA

What is the maximum possible speed achievable for a given device say Virtex6 device (some speed grade)

The Fmax is determined by Flop-to-Flop timing using shortest route (CLB) with least clock skew. To put it simple, usually this is calculated based on logic levels between a source to destination path

How do you code to reduce power in FPGA design

- Avoid reset for FPGA
- Clock Gate
- Use synchronous design
- Avoid overconstraining
- Reduce Device temperature (cooling solution)
- Use clk_en and control enable for all Memory
- Use LUT for smaller memory. BRAM takes more power

How do you manage multiple clocks and how do you route them

- CDC tools can help this like Spyglass, etc. But asynchronous transfers must be handles carefully in design and later they can be assigned false path for the tool to go easy on compilation

How do you control reset logic

- For FPGA, global reset is sufficient. Use async reset for internal logic and sync that reset in main clk (if they are in same clock, else use accordingly)
- Always assert the reset asynchronously and de-assert synchronously with clock

Some questions about RTL coding for FPGA primitive components, what are the primitive components and what have you used.

- BUFGMUX, ibuff, obuff, etc

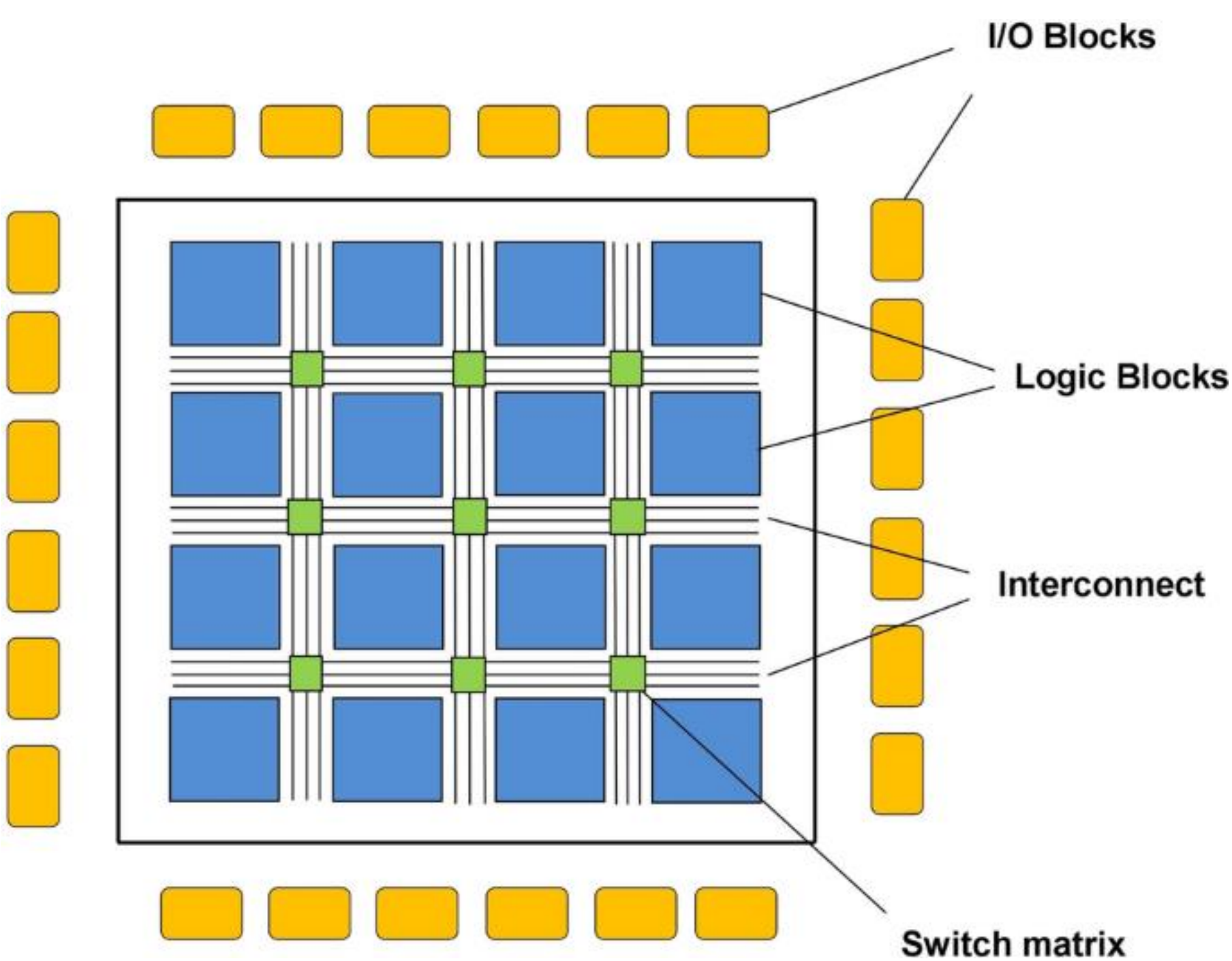
What is the difference between PAL and PLD?

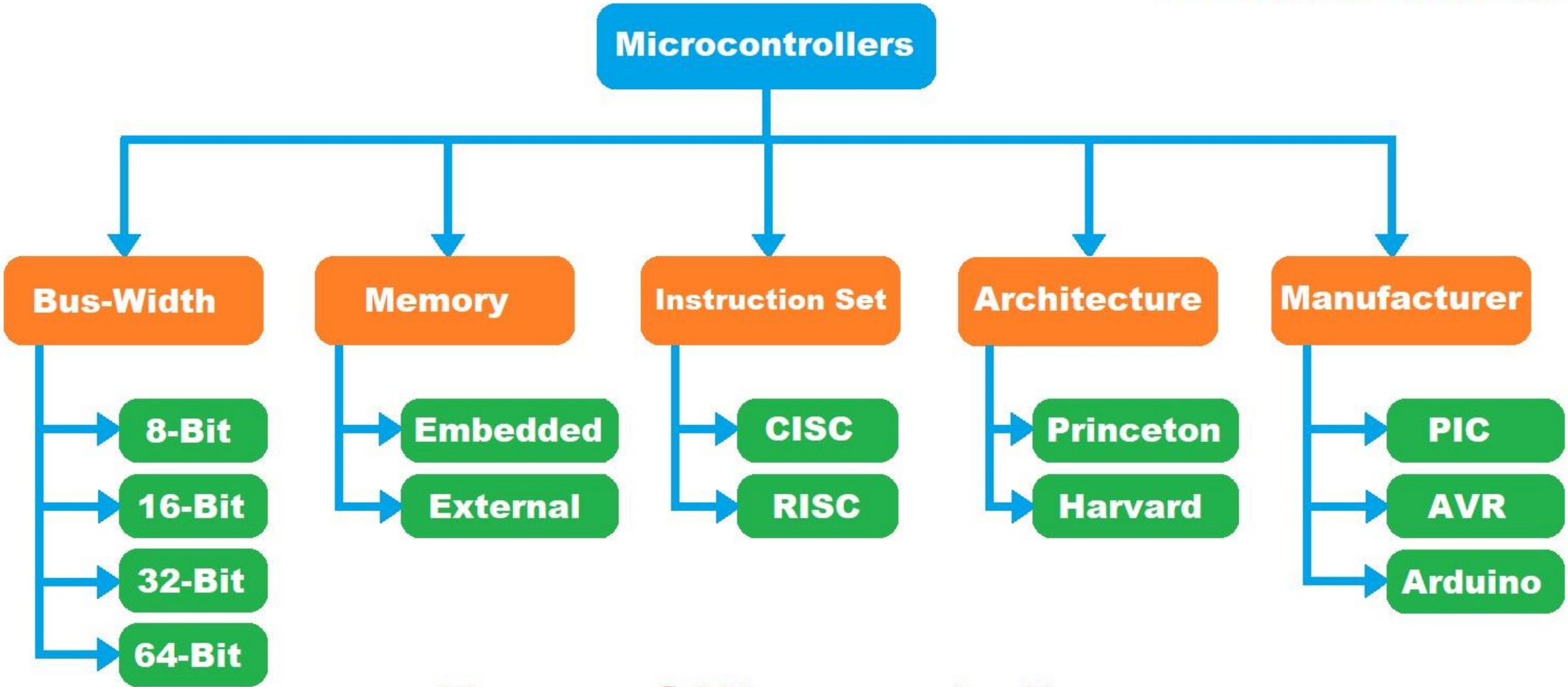
PLD (programmable logis devices) has a programmable OR plane, and fixed AND plae, PLA (programmable logis arrays) has programmable OR, and programmable AND palne, PAL (programmable array logic) has fixed OR, and programmable AND plane.

What do you understand by transport delay and inertial delay?

Transport delay: Transport delay is a type of delay caused by the wires that connect to the gates. Due to the wire's resistance and inductance, it delays the signal.

Inertial delay: The inertial delay is the time it takes for a gate to change its output.

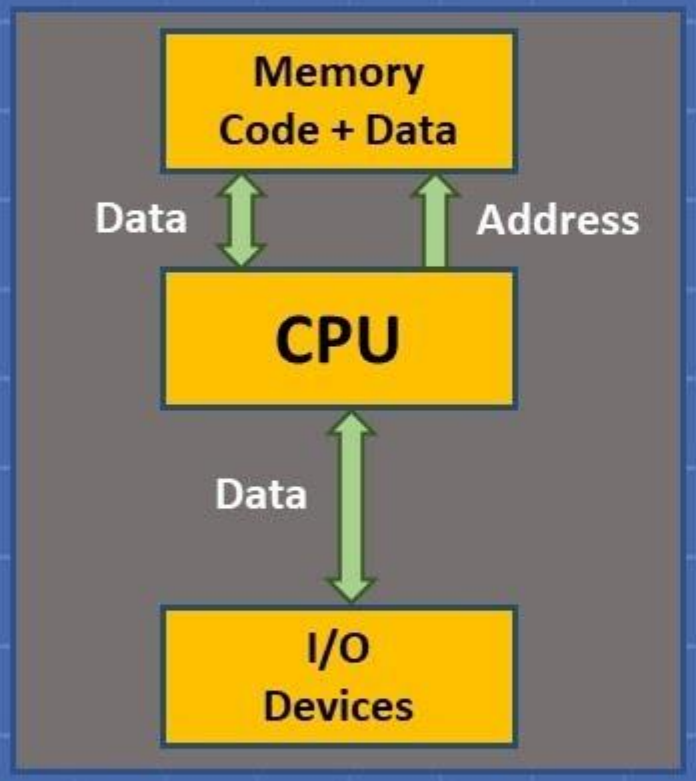

I/O Blocks
Logic Blocks
Interconnect
Switch matrix



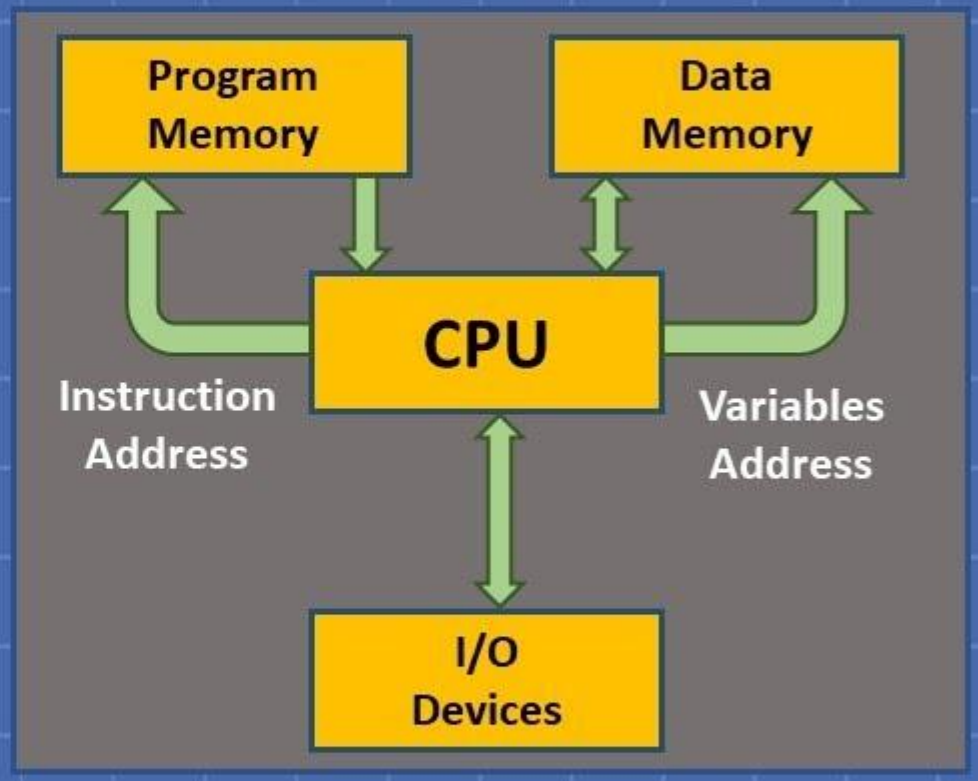
Types of Microcontrollers

Microprocessor & Microcontroller

Von-Neumann



Harvard Architecture



3 Difference between Von-Neumann & Harvard Architectures

Von-Neuman	Harvard
First digital computer architecture. Introduced stored program concept	Computer architecture based on Harva Mark 1
One memory module for data and instructions	Have different memory modules for da and instructions.
common bus for data and instructions	Individual buses for data and instructio
CPU takes 2 clocks to execute one instruction. Because fetch data before executing an instruction.	Can execute instruction one clock cycle
CPU can not fetch instructions and data read/write at the same time.	CPU can not fetch instructions and data read/write at the same time.
Slow	Fast

CISC: (Complex Instruction Set Computer)	RISC: (Reduced Instruction Set Computer)
More complicated processor design	Simpler processor design
Uses complex instructions	Uses simpler instructions
Each instruction may take multiple machine cycles	Each instruction takes one machine cycle
Does not allow pipelining as the instruction is not completely clear.	Allows pipelining as the instructions are more clear. However, Dependency can affect pipelining.
Longer Instruction Set	Smaller Instruction Set
Many instructions are available	Limited number of instructions are available
An instruction can perform complex tasks so no need to combine multiple instructions	An instruction performs a simple task so complex tasks can be achieved by combining multiple instructions
A task may be able to be completed in a single machine cycle	A task many take a number of machine cycles
Many addressing modes are available	Fewer addressing modes are available
Uses a single register set	Uses one or more register sets
Requires less RAM	Require more RAM
Programs run more slowly due to complicated circuit	Program run faster due to simple instructions

SRAM VS DRAM

SRAM	DRAM
It has less storage capacity	It has large storage capacity
SRAMs are low density devices.	DRAMs are high density devices.
These are used in cache memories.	These are used in main memories.
SRAM is expensive than DRAM.	DRAM is cheaper than SRAM

Why might you choose to use an FPGA in your design?

FPGAs are highly customizable. Some reasons to use an FPGA in your design might be: many unique peripherals required, lots of input/output needed, very fast processing speed required, lots of math operations, high data throughputs, interfaces to high bandwidth external memory, and reprogrammability are common reasons.

What does a for loop do in synthesizable code? Does it work the same way as in a software language like C?

A for loop in VHDL and Verilog is not the same thing as a for loop in python or C. In synthesizable code, for loops are used to replicate logic. It saves having to type the same thing over and over again, but it does not produce a loop in the same way that software programming loops work. In general, for loops should be avoided in synthesizable code for beginners unless they clearly understand how they work. Further Reading:

What is the purpose of a PLL?

PLL stand for Phase-locked loop and is commonly used inside FPGAs to generate desired clock frequencies. PLLs are built-in to the FPGA fabric and are able to take an input clock and derive a unique out-of-phase clock from that input clock. They are very useful if your design requires several unique clocks to be running internally.

Describe the difference between inference and instantiation

Inference is when you write VHDL or Verilog to “infer” or tell the synthesis tools to place some type of component down. For example, by creating a large memory storage register, you might be inferring a Block RAM. Instantiation is when you directly create the primitive component for the Block RAM based on the particular vendor’s user guide for how to instantiate primitive components. Inferrence is more portable across FPGA technologies. Instantiation might be better if you need to be very explicit about the primitive that you want to work with, or apply some unique settings to it.

What is FIFO?

FIFO stands for First In First Out. It is a commonly used FPGA component. A FIFO is a storage element, usually made of a Block RAM for large FIFOs and registers for short FIFOs. They are used to buffer data, for example when reading and writing to external memory, or for when crossing clock domains, or for storing pixels coming out of a camera. Further Reading

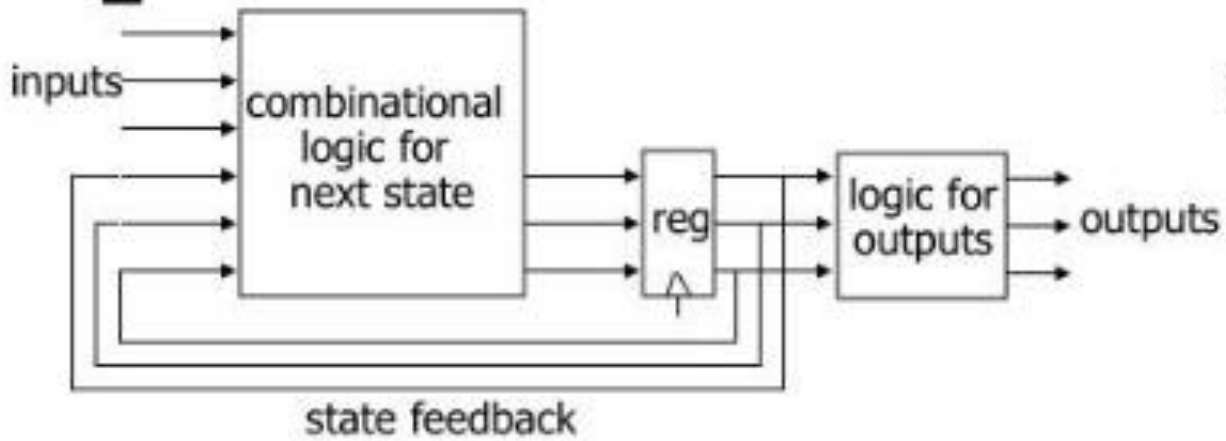
What is a Block RAM?

A Block RAM is a specific part of an FPGA that is usually a 16k or 32k bits storage element. It can have dynamic width and depth and is useful for many applications inside of an FPGA. They are used in Dual-port memories, FIFOs, and LUTs to name a few.

What is a shift register in an FPGA?

A shift register is a method of moving data from a source to a destination in some number of clock cycles. It is useful for creating delays inside of your FPGA, or for converting serial data to parallel data, or for converting parallel data to serial data.

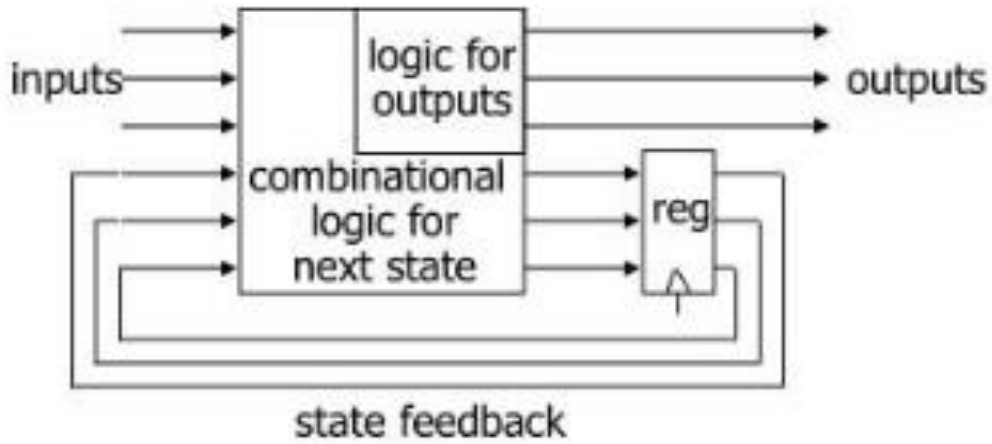
Moore vs. Mealy machines



Moore machine

Outputs are a function of current state

Outputs change synchronously with state changes



Mealy machine

Outputs depend on state and on inputs

Input changes can cause immediate output changes **(asynchronous)**

What is the purpose of the synthesis tools?

The synthesis tools are provided by the FPGA vendor and are used to translate your VHDL or Verilog code into logic that the FPGA is built from (e.g. Flip-Flops, Look-Up Tables, Block RAMs, etc).

What happens during Place and Route?

The synthesis process is usually followed by place and route, which takes the primitives and places them inside the FPGA and checks that the entire design meets your timing constraints. The timing constraints tell the FPGA the clock rates and the specific I/O to use, and the place and route process ensures that your design is able to work at those speeds.

What is a SERDES transceiver and where are they used?

SERDES stands for SERializer/DESerializer. These are high-speed transmitters and receivers that are used to send serial data across a point to point link. Usually these are used at speeds of > 1 Gbps. Current FPGA technology can have SERDES transceivers that operate at > 50 Gbps for a single data link. Data integrity at these speeds is very challenging, so lots of tricks are employed to ensure data is able to pass successfully. They are used commonly in RF applications, high-speed video applications, communication interfaces such as PCI Express, SATA, Gigabit Ethernet, etc.

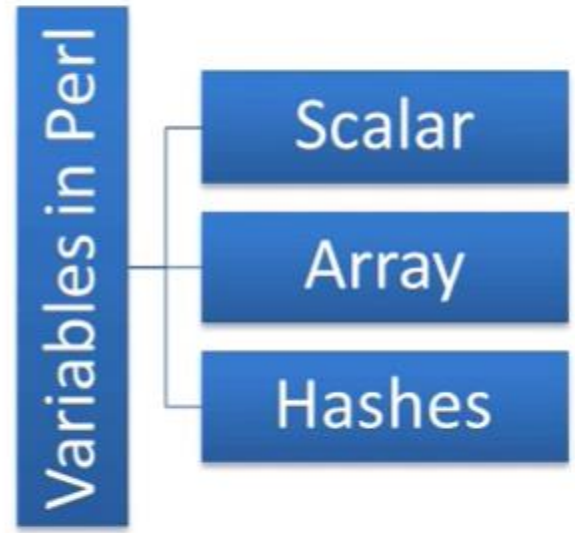
Features of Perl

- There is no requirement of specifying the data type in Perl
- It only has three types – scalar, array and hashes
- Scalar must start with $
- Array must start with @
- It automatically takes int or string (both are scalar values for perl)
- By default, all scalars have initial value as 0

Defining Variables

- $employee_name = “Sagar”; # String assignment
- $employee_age = 23; # An integer assignment
- $employee_salary = 440.5 # Floating point number

```
print "Age = $employee_age\n";
print "Name = $employee_name\n";
print "Salary = $employee_salary\n";
```

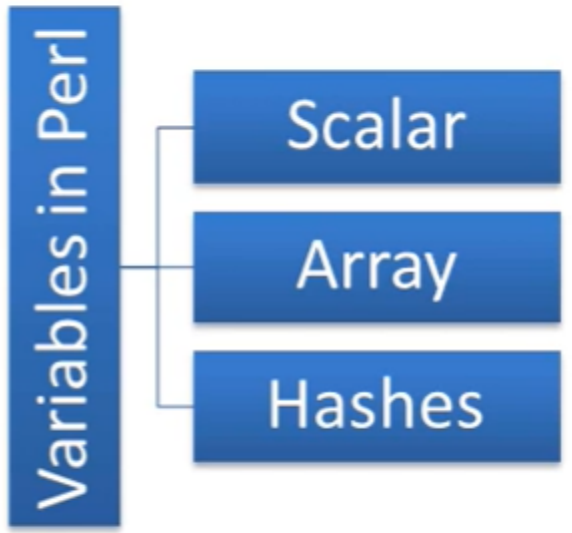


Defining Variables

- Array variables are preceded by “@” unlike “$” in scalar
- @names = (”Mohan", ”Ekansh", ”Henna");
- @ages = (25, 28, 31);

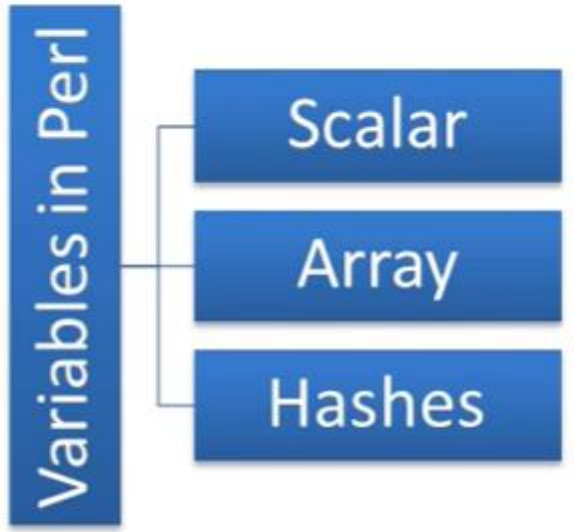
```
print "\$ages[0] = $ages[0]\n";
print "\$ages[1] = $ages[1]\n";
print "\$ages[2] = $ages[2]\n";
print "\$names[0] = $names[0]\n";
print "\$names[1] = $names[1]\n";
print "\$names[2] = $names[2]\n";
```

$ages[0] = 25
$ages[1] = 28
$ages[2] = 31
$names[0] = Shobhit
$names[1] = Ekansh
$names[2] = Henna



Defining Variables

- **Hash** variables are preceded by "%" sign
- %emp_data = ('Danish', 28, 'Raju', 40, 'Ritesh', 25);




```
print "\$data{'Danish'} = $data{Danish'}\n";
print "\$data{'Raju'} = $data{'Raju'}\n";
print "\$data{'Ritesh'} = $data{'Ritesh'}\n";
```

```
$data{'Danish'} = 28
$data{'Raju'} = 40
$data{'Ritesh'} = 25
```

Strings

- Perl is strong in operating on strings
- You can concatenate, increment, decrement, repeat etc. to the strings based on requirement
- For example,

```
$string1 = "potato";
$string2 = "head";
$newstring = $string1 . $string2;
```

- The value of `$newstring` is "potatohead"

Strings

- Multiply Operation:

```
$stringvar = "abc";
print($stringvar * 2);
```

- It will result in `$stringvar = 0`
- The multiply operator treats string as 0 by default after parsing
- Multiply Operation on alphanumeric string:

```
$stringvar = "12P34";
print($stringvar * 2);
```

- It will result in `$stringvar = 24`

Strings

- Increment Operation:

```
$stringvar = "abc";
$stringvar++;
```

- It will result in `$stringvar = "abd"`
- Repeat Operation:

```
$newstring = "t" x 5;
```

- The `$newstring` will be `"ttttt"`

Strings

```
$str = "z";
$str++;
```

- The value of `$str` will be `"aa"`.

```
$str = "1.2P34";
$str++;
```

- The value of `$str` will be `"2.2"`.

Array vs Lists

```
$scalar = 'text';
@array = (1, 2, 3);
%hash = (key1 => 'val1', key2 => 'val2');
#In above 3 examples, all three are lists of scalar, array and hash
#type.
```

- An array is a *variable*, but all of Perl's data types (scalar, array and hash) can provide a *list*, which is simply an ordered/Unordered set of scalars.

Lists

- Generally speaking, Lists are collection of scalars

```
#list of characters
@list1 = (a,b,c,d);
#list of integers
@list2 = (1,2,3,4);
#list of strings
@list3 = ("this", "is", "a","list");
```

Lists

Danish	Satish	Rajesh	Manju	Uma	Vipin	Suresh

Declaring a list

```
@names=(Danish, Satish, Rajesh, Manju, Uma, Vipin, Suresh);
```

Accessing a list element

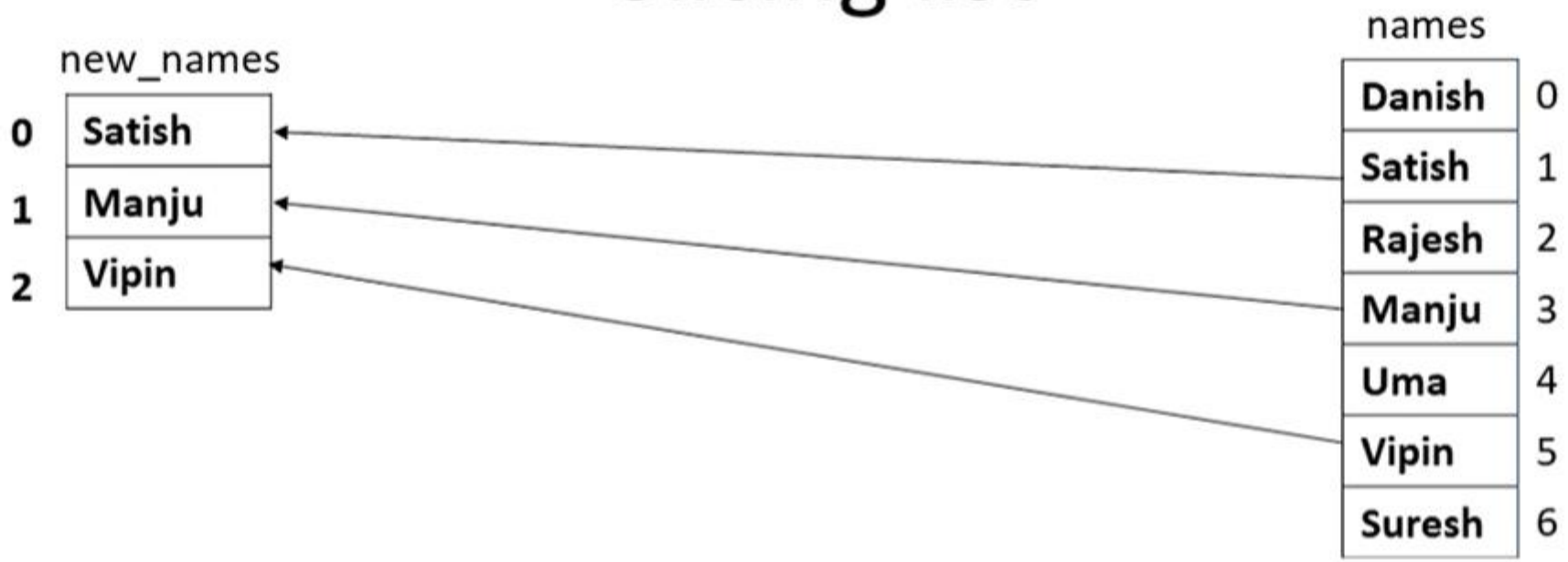
```
$array_element = $names[4];
```

Accessing last element of list

```
$last_name = $names[-1];
```

```
%> print "$names[4];
Uma
%> print "$names [-1];
Su
```

Slicing list




```
@names = (Danish, Satish, Rajesh, Manju, Uma, Vipin, Suresh);
@new_names[0,1,2] = @names[1,3,5];
print "$new_names[1]";
```

```
%> print "$new_names[1]";
Manju
```

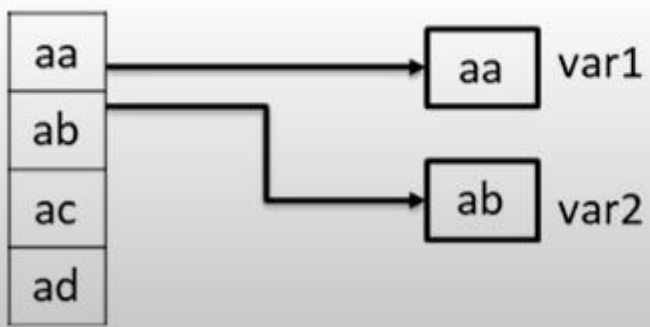
Lists

- `@list1 = (1..10);`
  `print(@list1);`
- Defining a part of list: `(2, 5..9, 11)`
- Lists with floating point values will be like this `(2.1..6.3)`

```
%> print (@list1);
12345678910
%> print (2, 5..9, 11);
25678911
```

Lists

- `@list1 = (aa..ad);`
  `print(@list1);`
- `print(@list1)` and `print("@list1")` have different outputs.
- `$var1, $var2 = @list1;`




```
%> print (@list1);
aaabacad
%> print ("@list1");
aa ab ac ad
```

- `@numbers = (9,2,8,4,1);`
- `@names = (" rosy "," mahesh "," ruby "," john ");`
- `@sorted_num = sort @numbers;`
- `@sorted_names = sort @names;`
- `@descend_num = reverse sort @numbers;`
- `@reverse_names = reverse sort @names;`

```
%> print (@sorted_num);
12489
%> print (@sorted_names);
 john  mahesh  rosy  ruby
%> print (@descend_num);
98421
%> print (@reverse_names);
ruby  rosy  mahes
```

- Merge the elements of an array into a single string

```
string = join (array);                    #syntax of join command
$string1 = join(" ", "this", "is", "a", "string");
$string2 = join("::", "words", "and", "colons");
@list = ("Here", "is", "a");
$string3 = join(" ", @list, "string");
```

```
%> print ($string1);
This is a string

%> print ($string2);
words::and::colons
```

- Split a string into array elements :

```
array = split (string);
$string = "words::separated::by::colons";
@array = split(/::/, $string);
```

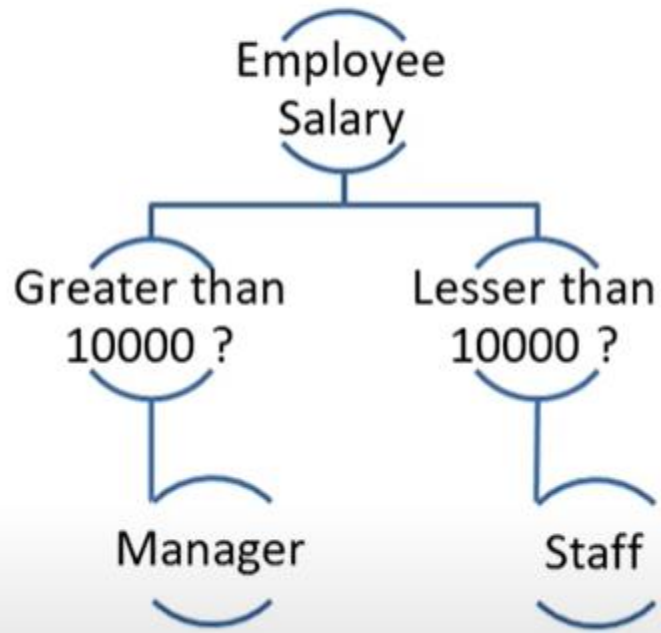
- The resulting string will be :

```
print("@array");
words separated by colons
$string1 = "abcde";
@array1 = split(//,$string1);
```

Conditional Statements

if-else condition:

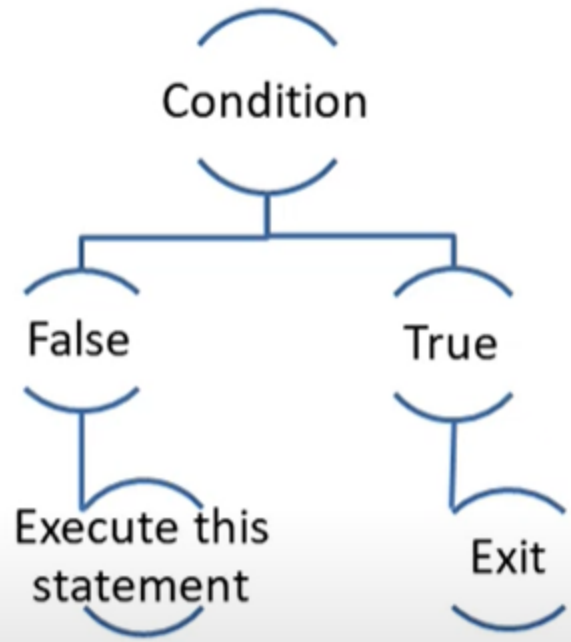
```
$salary = 10500
if( $salary > 10000 ) {
print "Employee is Manager\n";
}
elsif ($salary < 10000) {
print "Employee is staff\n";
}
```



Conditional Statements

- Unless condition:

```
$a = 22
unless( $a < 20 )
{
print( "a is greater than 20\n");
}
```



Conditional Statements

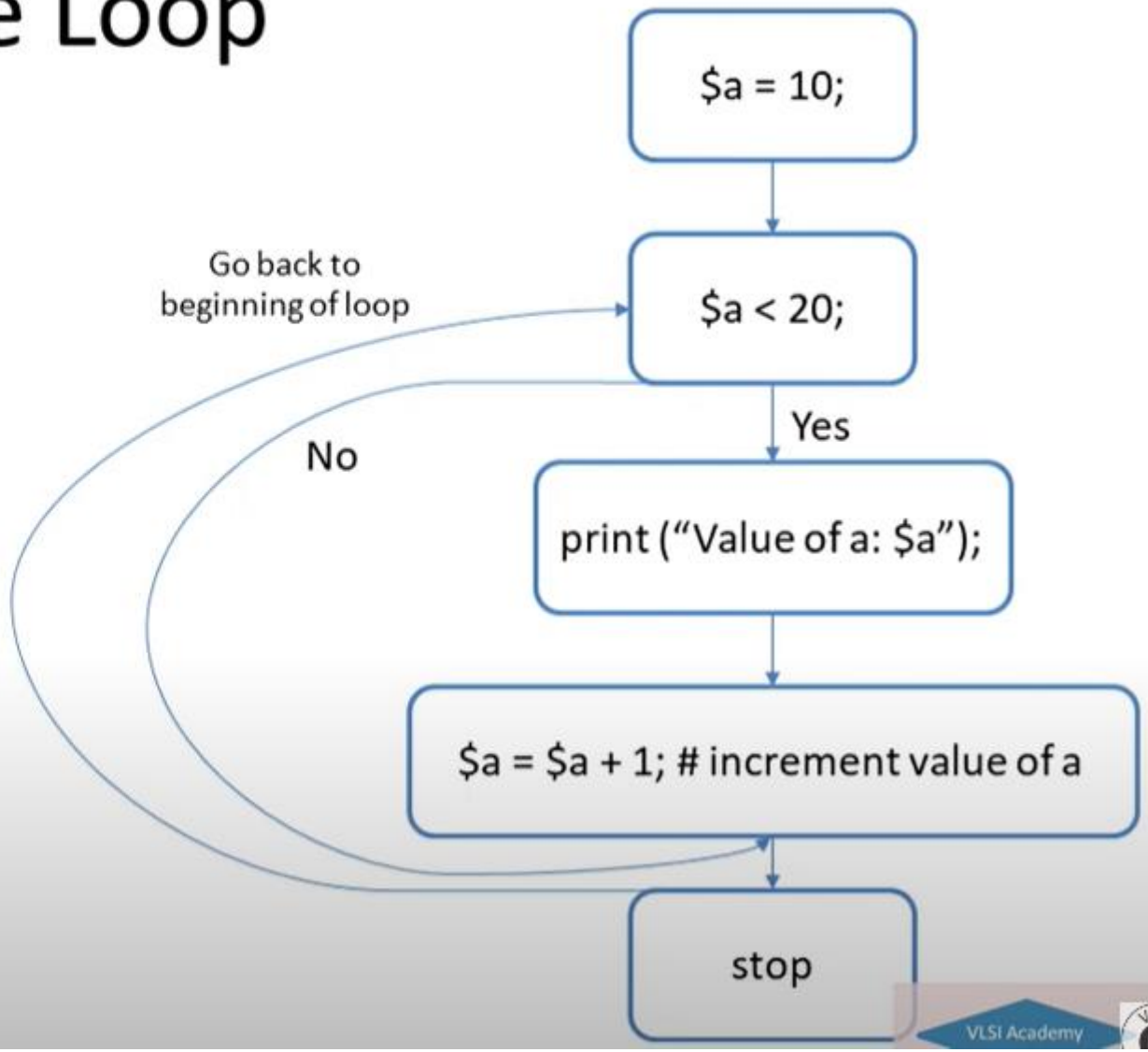
```
use Switch;
$var = 30;
@array = (10, 20, 30);
%hash = ('key1' => 10, 'key2' => 20);
switch($var){
    case 10             { print "number 100\n" }
    case "a"            { print "string a" }
    case [1..10,42]     { print "number in discontinuous list" }
    case (\@array)      { print "number in array list" }
    case (\%hash)       { print "entry in hash" }
    else                { print "previous case not true" }
```

While Loop

- **While** Loop:

repeatedly executes a target statement as long as a given condition is true.

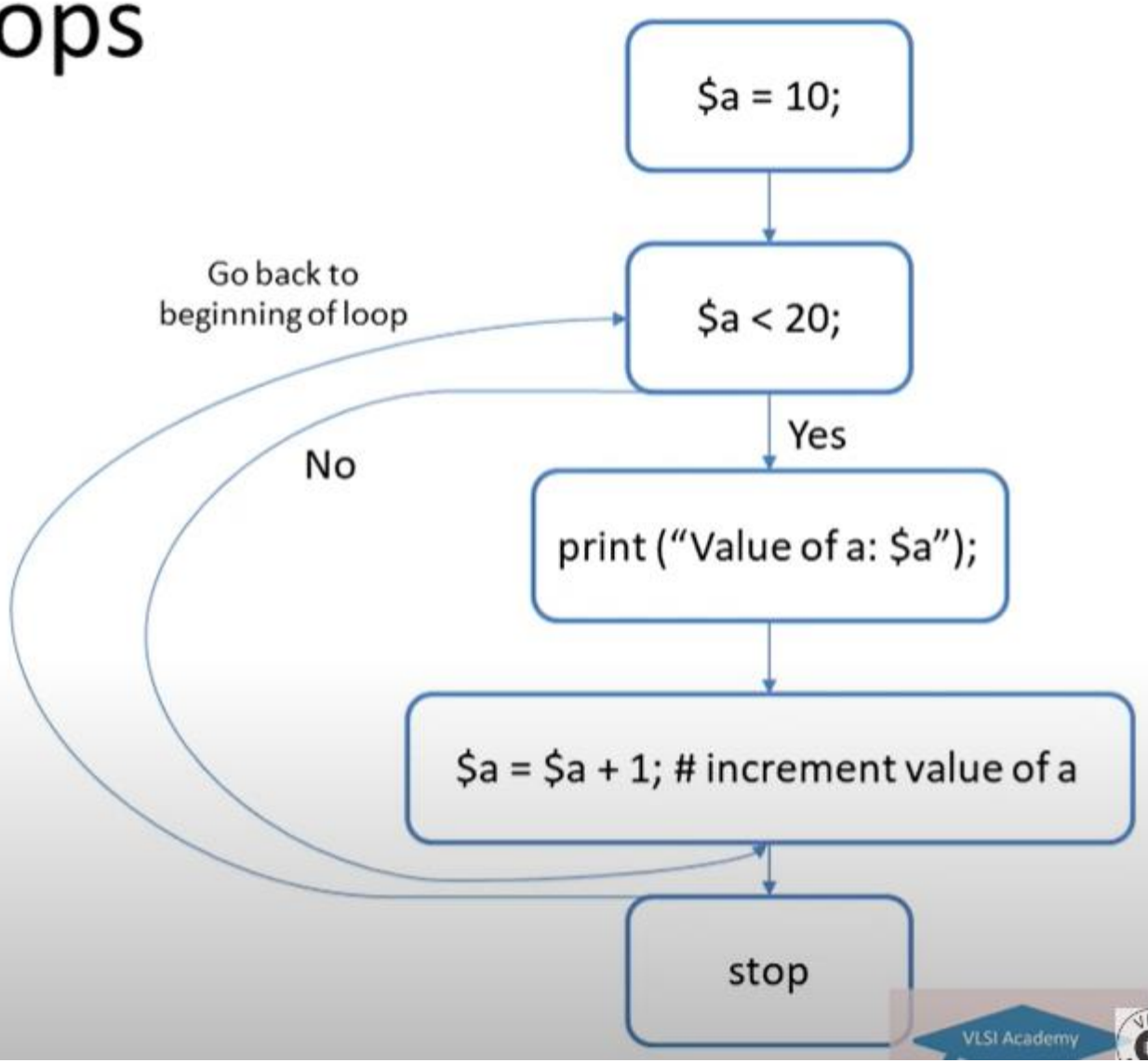
```
$a = 10;
while( $a < 20 )
{
print( "Value of a: $a\n");
$a = $a + 1;
}
```



Loops

- **For** Loop:
- Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

```
for( $a = 10; $a < 20; $a = $a + 1 )
{ print "value of a: $a\n"; }
```



Loops

```
@names = (“Ragav”,”Yogita”,”Ankit”,”Vyas”,”Pradeep”,”Pavan”);
$size = @names;
for( $i = 0; $i < $size; $i = $i + 1 ) {
   print “$names[i]”;
}
```

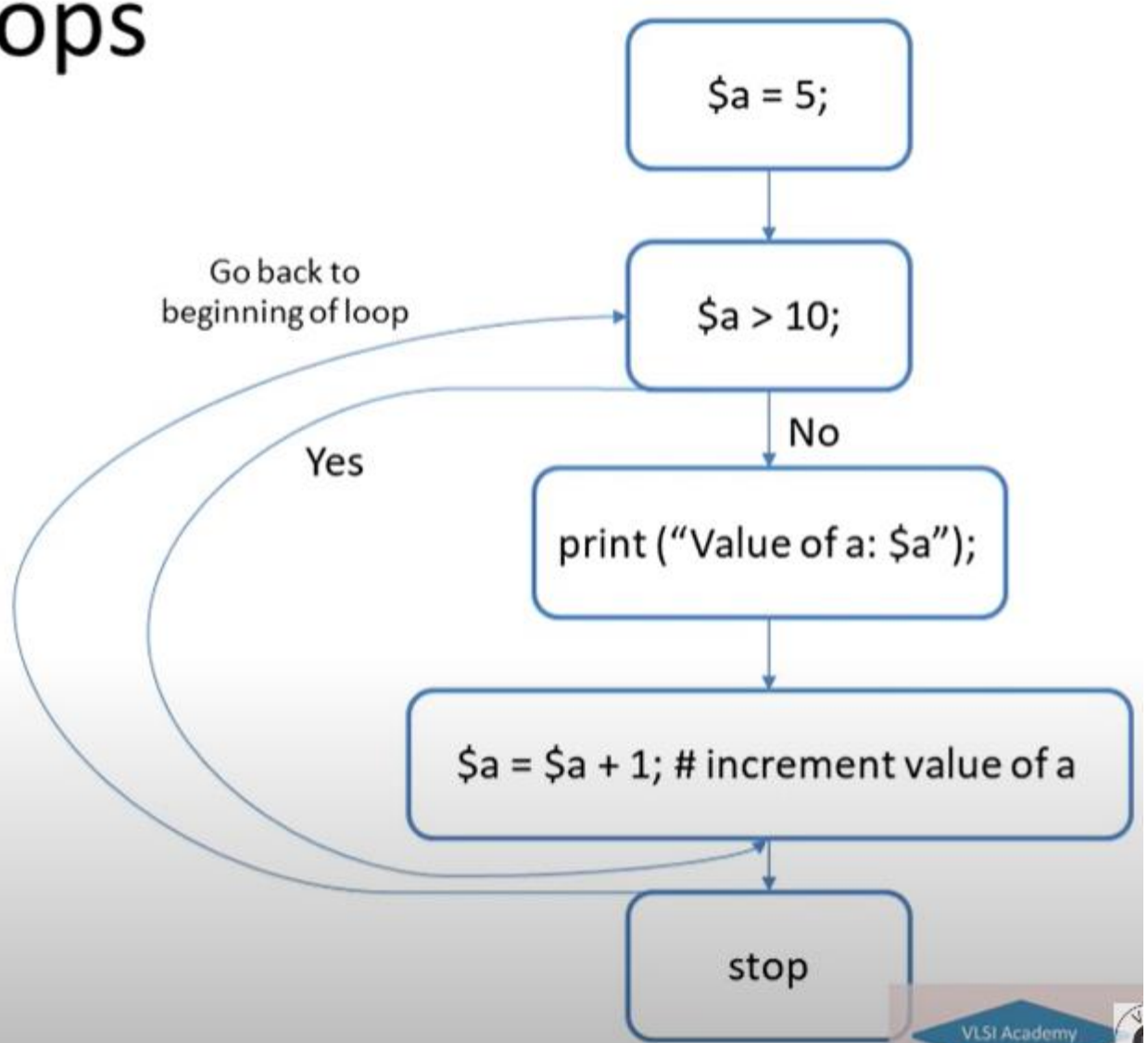
@names	Raghav	Yogita	Ankit	Vyas	Pradeep	Pavan
Index →	0	1	2	3	4	5

Loops

- **Until** Loop:

repeatedly executes a target statement as long as a given condition is false.

```
$a = 5
until( $a > 10 )
{
print( "Value of a: $a\n");
$a = $a + 1;
}
```



Loops

- **Foreach** Loop
- The foreach loop iterates over array by referencing the value directly instead of referencing the index of array.

```
foreach $employee (@names)
{ print "name of employee: $employee\n"; }
```

@names	Raghav	Yogita	Ankit	Vyas	Pradeep	Pavan
Index →	0	1	2	3	4	5

```perl
use List::Util qw(min max);
@price;
open($fh, '<', "leaders.rpt");
while($line = <$fh>) {
    @data = split(' ',$line);
    if ($data[-1]!= 'price') {
    push(@price,$data[-1]);
    }
    }

    print("@price\n");
    $maxprice = max @price;
    print $maxprice;
```

```
leader  country Phone   price
trump   America iphone4 600
modi    India   pixel3  500
boris   Britain iphoneX 800
putin   Russia  HTC9    400
imran   Pakstan nokia   20
jinping China   huawei  350
frank   Germany iphone6 700
```

```
PS F:\perl> perl .\leader_summary.pl
600 500 800 400 20 350 700
800
PS F:\perl>
```

Functions (Sub-routines)

- Syntax:

```
sub subroutine_name
{ body of the subroutine }
```

- Calling function

```
subroutine_name( list of arguments );
```

- To access the arguments inside the function, use the special array `@_`
- The first argument to the function is in `$_[0]`, the second is in `$_[1]`, and so on

In function definition you don't have to specify the number of arguments to be passed.

this is a system array @_, it will be containing all the list of arguments and you can clearly see how we can access each argument one by one here

Passing Array to Subroutine

- Example:

```
# Function definition
sub PrintMyList{
my @list = @_;
print "Given list is @list\n";
}
$x = 10;
@y = (1, 2, 3, 4);
# Function call with list parameter
PrintMyList($x, @y);
```

new 1

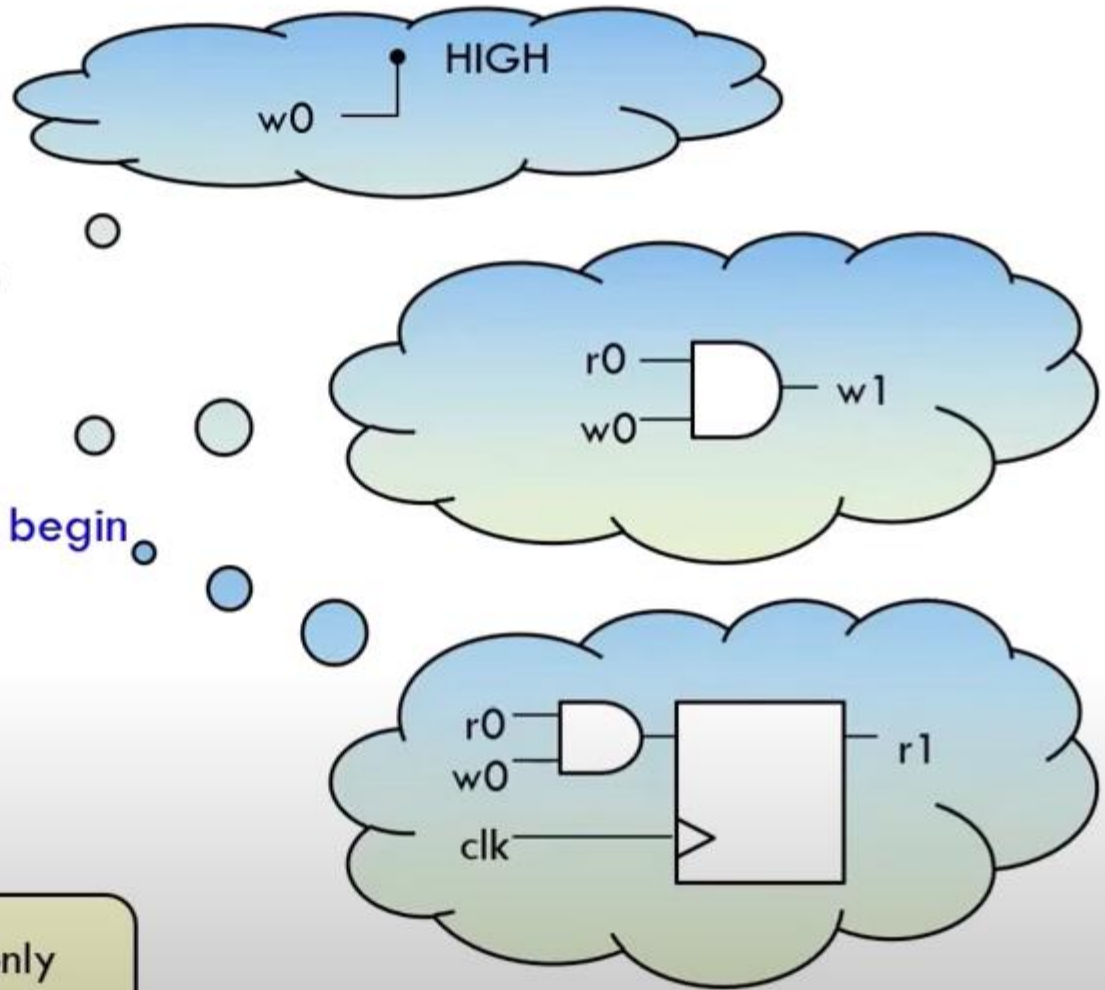
```
1  %leader = ('Modi', India, 'Trump', USA, 'Putin', Russia);
2  print "\$leader{'Modi'} = $leader{'Modi'}\n";
3  print "\$leader{'Trump'} = $leader{'Trump'}\n";
```

```
PS F:\perl> perl .\lec9.pl
$leader{'Modi'} = India
$leader{'Trump'} = USA
PS F:\perl>
```

```perl
%leader = ('Modi', India, 'Trump', USA, 'Putin', Russia);
print "\$leader{'Modi'} = $leader{'Modi'}\n";
print "\$leader{'Trump'} = $leader{'Trump'}\n";

$leader{'Boris'} = Britain;

@names = keys %leader;

@country = values %leader;

print "@names   ";
print "@country ";
#iterating over hashes
while (($key,$value) = each %leader) {
    print "$key => $value \n";
```

```
PS F:\perl> perl .\lec9.pl
$leader{'Modi'} = India
$leader{'Trump'} = USA
Trump Boris Putin Modi   USA Britain Russia India
Trump => USA
Boris => Britain
Putin => Russia
Modi => India
PS F:\perl>
```

```
module dut ( input wire clk );

    wire w0, w1;
    reg r0, r1;
    assign w0 = 1'b1;
    assign w1 = r0 & w0;
    always @( posedge clk ) begin
        r1 = r0 & w0;
    end
    initial r0 = 1'b1;

endmodule
```



- Simulation only
- Not for synthesis

```
module dut ( input wire clk );

    wire w0, w1;
    reg r0, r1;
    assign w0 = 1'b1;
    assign w1 = r0 & w0;
    always @( posedge clk ) begin
        r1 = r0 & w0;
    end
    initial r0 = 1'b1;

endmodule
```

```
always @( posedge clk or posedge reset ) begin
    if ( reset ) begin
        r1 <= 0;
    end else begin
        r1 <= r0 & w0;
    end
end
```

```
module dut ( input wire clk );

    logic l0, l1, l2;
    initial l0 = 1'b1;                    // Procedural statement
    always @( posedge clk ) begin         // Sequential statement
        l1 = l0;
    end
    assign l2 = l1;                       // Continuous assignment

endmodule
```

```
module test ();
        wire w0, w1, w2;
        assign w0 = w1; // Multiple drivers for w0
        assign w0 = w2; // Multiple drivers for w0
        logic l0, l1, l2;
        assign l0 = l1;
        assign l0 = l2;    // Illegal multiple drivers for l0
endmodule
```

```
module test ();
    logic I0;
    initial I0 = 1'b0; // Initial block #1
    initial I0 = 1'b1; // Initial block #2
endmodule
```

- Nope, not multiple drivers.
- This is a race condition.

```
module dut ( input wire clk );
        logic l0;
        always @( posedge clk ) begin // Sequential statement #1
                l0 = 1'b0;
        end
        always @( posedge clk ) begin // Sequential statement #2
                l0 = 1'b1;
        end
endmodule
```

- Nope, not multiple drivers.
- This is a race condition.

```
module dut ( input wire clk );
        logic l0;
        assign l0 = 1'b0;
        always @( posedge clk ) begin
                l0 = 1'b1;
        end
endmodule
```

- This is a multiple drivers case and therefore it will fail compilation.

SUMMARY

- *logic* is a new data type introduced in SystemVerilog.
- It can be used with procedural statements , sequential statements and continuous assignments.
- It cannot be used with multiple drivers; only *wire* can support that.

TYPE	WIDTH	STATE	SIGNED
integer	32	4 states: 0,1,Z,X	signed
reg	1	4 states: 0,1,Z,X	1bit
wire	1	4 states: 0,1,Z,X	1bit
logic	1	4 states: 0,1,Z,X	1bit
bit	1	2 states: 0,1	1bit
byte	8	2 states: 0,1	signed
shortint	16	2 states: 0,1	signed
int	32	2 states: 0,1	signed
longint	64	2 states: 0,1	signed

```
module test ();
        byte byte0;         // 2 states variable
        logic[7:0] I8;      // 4 states variable
        initial begin
                byte0 = I8;
                if ( byte0 == I8 ) $display( "byte0 == I8" );
                if ( byte0 === I8 ) $display( "byte0 === I8" );
                if ( $isunknown( I8 ) ) $display( "I8 is unknown" );
                I8 = 8'b00000000;
                if ( $isunknown( I8 ) ) $display( "I8 is unknown" );
        end
endmodule
```

- Initialized as 0, byte0=8'b00000000
- Initialized as X, I8 =8'bxxxxxxxx

- byte0 =8'b00000000
- I8 =8'bxxxxxxxx



```
module test ();
        byte byte0;    // 2 states variable
        logic[7:0] l8; // 4 states variable
        initial begin
                l8 = 255;
                byte0 = -1;
                if ( l8 == byte0 ) $display( "same" );
                $display( "byte0 = %0d, %0b", byte0, byte0 );
                $display( "l8 = %0d, %0b", l8, l8 );
        end
endmodule
```

- byte0 =0
- l8 =X
- l8 =255
- byte0 =-1
- 255 = -1???

Print: byte0 = -1, 11111111

Print: l8 = 255, 11111111

```
module test ();
        byte byte0;
        initial begin
                for ( byte0=0 ; byte0<200 ; byte0++ ) begin
                        $display( "byte0=%0d", byte0 );
                end
                $display( "Count finished" );
        end
endmodule
```

- byte0 is signed (-128 to 127)
- byte0 is always <200

```
byte0=0
byte0=1
byte0=2
byte0=3
...
byte0=127
byte0=-128
byte0=-127
```

```
module test ();

    byte unsigned byte0;
    initial begin
        for ( byte0=0 ; byte0<200 ; byte0++ ) begin
            $display( "byte0=%0d", byte0 );
        end
        $display( "Count finished" );
    end

endmodule
```

- Byte unsigned range: 0 - 255

- byte0 is signed (-128 to 127)
- byte0 is always <200

```
byte0=0
byte0=1
byte0=2
byte0=3
...
byte0=127
byte0=-128
byte0=-127
```

```
module test ();

    typedef byte unsigned ubyte;

    byte unsigned   ubyte0;
    byte unsigned   ubyte1;
    ubyte           ubyte00;
    ubyte           ubyte11;

endmodule
```

type

variable

```
module test ();


        int i0;
        int i1;
        initial begin
                i0 = 32'd1;
                $display( "i0=%0d", i0 );
                i1= 32'd0;
                $display( "i1=%0d", i1 );
        end
endmodule
```

```
module test ();
        parameter RED=1, GREEN=2;
        parameter DAY=0, NIGHT=1;
        int colour;
        int mode;
        initial begin
                colour = RED;
                $display( "colour=%0d", colour );
                mode = DAY;
                $display( "mode=%0d", mode );
        end
endmodule
```

```
module test ();

        enum int {RED=1, GREEN=2} colour;
        enum int {DAY,NIGHT}      mode;

        initial begin
                colour = RED;
                $display( "colour=%0d", colour );
                mode = DAY;
                $display( "mode=%0d", mode );
        end
endmodule
```

type

variable

```
module test ();
        parameter RED=1, GREEN=2;
        parameter DAY=0, NIGHT=1;
        int colour;
        int mode;
        initial begin
                colour = RED; // colour = NIGHT
                $display( "colour=%0d", colour );
                mode = DAY;
                $display( "mode=%0d", mode );
        end
endmodule
```

```
enum {R, G, B}  colour; // int colour

enum {RED, GREEN, BLUE}  colour; // int colour

enum int {R=0, G=1, B=2}  colour;

enum bit[1:0] {R, G, B}  colour; // bit[1:0] colour

enum {R=1, G, B}  colour; // R=1,G=2, B=3

enum {R, G=2,B}  colour; // R=0, G=2, B=3

enum {R=0, G=2,B=3}  colour; // R=0, G=2, B=3
```

```
module test ();
        enum {RED, GREEN}  colour;
        initial begin
                colour = RED;
                colour = 0;
                $cast( colour, 0 );
                $cast( colour, 2 );
        end
endmodule
```

Enum can only takes the values in its define

Cast executes the assignment if the underlying types are compliant

Fail simulation as 2 is not a valid value for colour; RED=0, GREEN=1

```
module test ();
        enum {RED, GREEN}  colour;
        enum {DAY, NIGHT}  mode;
        initial begin
                colour = RED;
                colour = DAY;
        end
endmodule
```

Enum variable is scoped.
RED != DAY even if they are both 0

```
module test ();
        enum {RED, GREEN}  colour;
        initial begin
                colour = RED;
                colour = colour.first(); // Get the first value: RED
                colour = colour.next(); // Get the next colour: GREEN
                colour = colour.next(); // Get the next colour: RED again
                colour = colour.last(); // Get the last colour: GREEN
                $display( "colour=%0d,%0s", colour, colour.name() );
        end
endmodule
```

Print: colour=1,GREEN

SUMMARY

- *enum* is a number data type which provides name to its value.
- It has predefined built-in functions.

```
module test (); // Understanding formatter
        logic[7:0] i = 4'd12;
        initial begin
                $display( "i=%d", i );
                $display( "i=%0d", i );
                $display( "i=%0b", i );
                $display( "i=%0h", i );
                $display( "i=%0d, %0b, %0h", i, i, i );
        end
endmodule
```

Print: i= 12

Print: i=12

Print: i=1100

Print: i=c

Print: i=12, 1100, c

```
module test (); // String concatenation
        string name = "John";
        string name2 = {name, name};
        initial begin
                $display( "Hello %s", name );
                $display( "Hello %s", name2 );
                $display( "Hello %s", {name, " ", name} );
        end
endmodule
```

name2="JohnJohn"

Print: Hello John

Print: Hello JohnJohn

Print: Hello John John

```
module test (); // String comparison
        string myName = "Leo";
        string yourName = "John";
        initial begin
                if ( myName == yourName ) begin
                        $display( "%s==%s", myName, yourName );
                end
        end
endmodule
```

```
module test (); // String built-in functions
        string name = "John";
        initial begin
                $display( "Hello %s", name );
                $display( "Hello %s", name.toupper() );
                $display( "Hello %s", name.tolower() );
                $display( "Hello %s", name.getc(0) );
                $display( "Hello %s", name.substr(0,1) );
        end
endmodule
```

Print: Hello John

Print: Hello JOHN

Print: Hello john

Print: Hello J

Print: Hello Jo

```
module test (); // String and number conversion
        string str = "1234";
        int i;
        initial begin
                i = str.len();
                i = str.atoi();
                i = 5678;
                str.itoa( i );
        end
endmodule
```

str="1234"; i=4

str="1234"; i=1234

str="1234"; i=5678

str="5678"; i=5678

```
module test (); // Writing to file
        integer fd;
        string content = “Hello”;
        initial begin
                fd = $fopen ( “file.txt”, “w” );
                if ( fd ) begin
                        $fdisplay ( fd, content ); // Write one line
                end
                $fclose( fd );
        end
endmodule
```

```
module test (); // Reading from file
        integer fd;
        string content;
        initial begin
                fd = $fopen ( “file.txt”, “r” );
                if ( fd ) begin
                        $fgets( content, fd ); // Read one line
                end
                $fclose( fd );
        end
endmodule
```

```
module dut ( input wire clk, reset );

  enum {OFF, ON} state;

  always @( posedge clk or posedge reset ) begin
      if ( reset ) begin
          state = OFF;
      end else begin
          state = (state==ON) ? OFF : ON;
      end
  end

endmodule
```

```
module dut ( input wire clk, reset );

  string state;

  always @( posedge clk or posedge reset ) begin
      if ( reset ) begin
          state = "OFF";
      end else begin
          state = (state=="ON") ? "OFF" : "ON";
      end
  end

endmodule
```

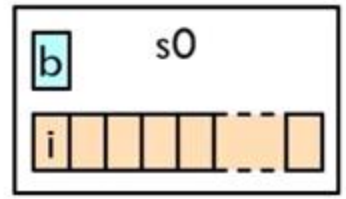
SUMMARY

- *string* is a data type used to display messages in a simulation.
- It has predefined built-in functions.

```
module test ();
    bit[7:0] red0, red1, red2;
    bit[7:0] green0, green1, green2;
    bit[7:0] blue0, blue1, blue2;
    initial begin
        red0=8'h0;
        green0=8'h0;
        blue0=8'hff;
        red1=red0; green1=green0; blue1=blue0;
        red2=red0; green2=green0; blue2=blue0;
    end
endmodule
```

```
module test ();
    struct {
        bit[7:0] red, green, blue;
    } pixel0, pixel1, pixel2;
    initial begin
        pixel0.red=8'h0;
        pixel0.green=8'h0;
        pixel0.blue=8'hff;
        pixel1 = pixel0;
        pixel2 = pixel0;
    end
endmodule
```

```
module test ();

    struct { bit b; int i; } s0;
    initial begin
        $display( "%0b", s0.b );
        $display( "%0b", s0.i );
        $display( "%0p", s0 );
        $display( "%0b", s0 );
    end

endmodule
```

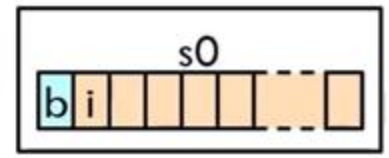


Print: 0

Print: 0

Print: '{b=0,i=0}

```
module test ();

    struct packed { bit b; int i; } s0;
    initial begin
        $display( "%0b", s0.b );
        $display( "%0d", s0.i );
        $display( "%0p", s0 );
        $display( "%0b", s0 );
    end

endmodule
```



Print: 0

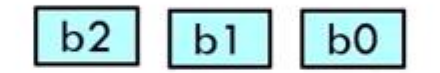
```
module test ();

        struct { bit b; int i; }   s0, s1;
        struct packed { bit b; int i; }   sp0, sp1;
        initial begin
                s1 = s0;
                sp1 = sp0;
                s1 = sp0;
        end


endmodule
```

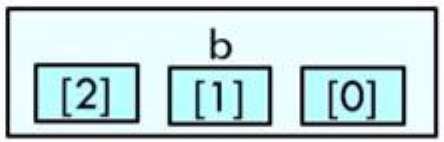
Type mismatch

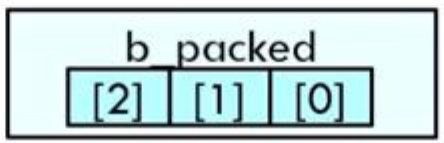
SUMMARY

- *struct* is an aggregate data type which can contains multiple variables of different types.

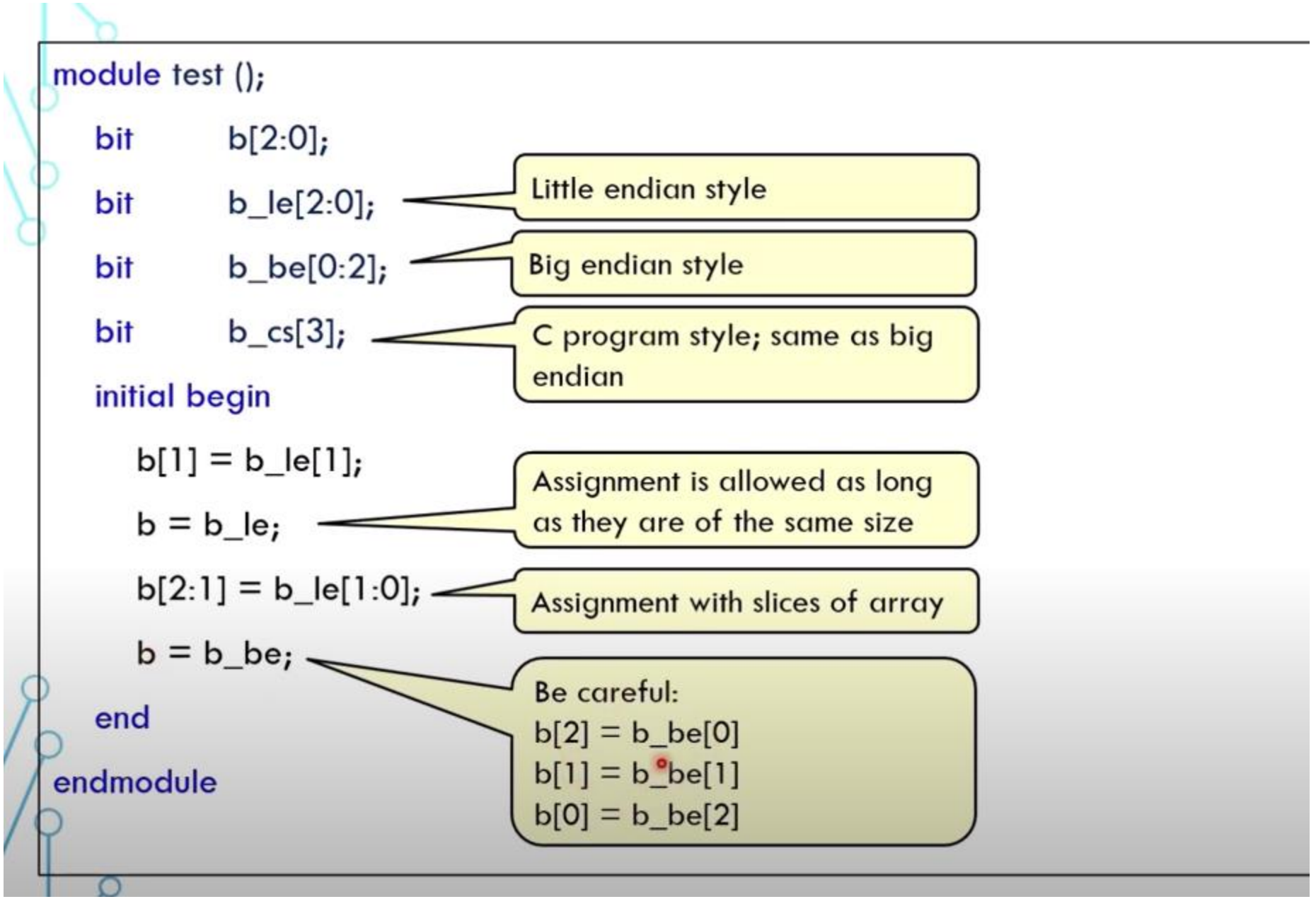
```
module test ();
    bit         b0, b1, b2;
    bit         b[2:0];
    bit[2:0]    b_packed;
    byte        by0, by1, by2;
    byte        by[2:0];
    byte[2:0]   by_packed;
endmodule
```







Only single bit vector can be packed: bit, logic, reg, wire


module test ();
bit b[2:0];
bit b_le[2:0];
bit b_be[0:2];
bit b_cs[3];
initial begin
b[1] = b_le[1];
b = b_le;
b[2:1] = b_le[1:0];
b = b_be;
end
endmodule
Little endian style
Big endian style
C program style; same as big endian
Assignment is allowed as long as they are of the same size
Assignment with slices of array
Be careful:
b[2] = b_be[0]
b[1] = b_be[1]
b[0] = b_be[2]

```
module test ();
    byte b[99:0];
    byte dynamicArray[];
    byte queue[$];
    byte associativeArray[*];
endmodule
```

Used in test bench;
not synthesizable

```
module test ();
    byte b[99:0];
    byte dynamicArray[];
    byte queue[$];
    byte associativeArray[*];
endmodule
```

Used in test bench;
not synthesizable

```
module test (); // No function
  int a, b, result;

  initial begin
    a=1; b=2; result=a+b;
    a=3; b=4; result=a+b;
    a=5; b=6; result=a+b;
  end
endmodule
```

Future modification in multiple places

```
module test (); // Simple function
  int a, b, result;
  function void calculate();
    result=a+b; $display( "%d",result );
  endfunction
  initial begin
    a=1; b=2; calculate();
    a=3; b=4; calculate();
    a=5; b=6; calculate();
  end
endmodule
```

Future modification in only one place

```
module test (); // Simple function
  int a, b, result;
  function void calculate();
    result=a+b;
  endfunction
  initial begin
    a=1; b=2; calculate();
    a=3; b=4; calculate();
    a=5; b=6; calculate();
  end
endmodule
```

```
module test (); // Function & arguments
  int result;
  function void calculate(int a, b);
    result=a+b;
  endfunction
  initial begin
    calculate( 1, 2 );
    calculate( 3, 4 );
    calculate( 5, 6 );
  end
endmodule
```

a and b now "belong" to function.

```
module test (); // Function & arguments
  int result;
  function void calculate( int a, b );
    result=a+b;
  endfunction
  initial begin
    calculate( 1, 2 );
    calculate( 3, 4 );
    calculate( 5, 6 );
  end
endmodule
```

```
module test (); // Function & return value
  int result , c;
  function int calculate( int a, b );
    return (a+b);
  endfunction
  initial begin
    result = calculate( 1, 2 );
    result = calculate( 3, 4 );
    c      = calculate( 5, 6 );
  end
endmodule
```

Function is decoupled from "result". It is independent and can be used by others

```
module test (); // No automatic syntax
  function void increment();
    int a;
    a = a+1;
  endfunction
  initial begin
    increment();
    increment();
    increment();
  end
endmodule
```

a is "hardware" signal; it exists from the beginning till the end.

a = 1

a = 2

a = 3

```
module test (); // automatic syntax
  function automatic void increment();
    int a;
    a = a+1;
  endfunction
  initial begin
    increment();
    increment();
    increment();
  end
endmodule
```

a is "software" variable; it's created when the function is called and destroyed when the function call finishes.

a = 1

a = 1

a = 1

```
module test (); // function
  int result;
  function int calculate( int a, b );
    return (a+b);
  endfunction
  initial begin
    #1; result = calculate( 1, 2 );
    #1; result = calculate( 3, 4 );
    #1; result = calculate( 5, 6 );
  end
endmodule
```

Function has return value capability

```
module test (); // task
  int result;
  task calculate( int a, b );
    #1; result = (a+b);
  endtask
  initial begin
    calculate( 1, 2 );
    calculate( 3, 4 );
    calculate( 5, 6 );
  end
endmodule
```

Task has timing control capability

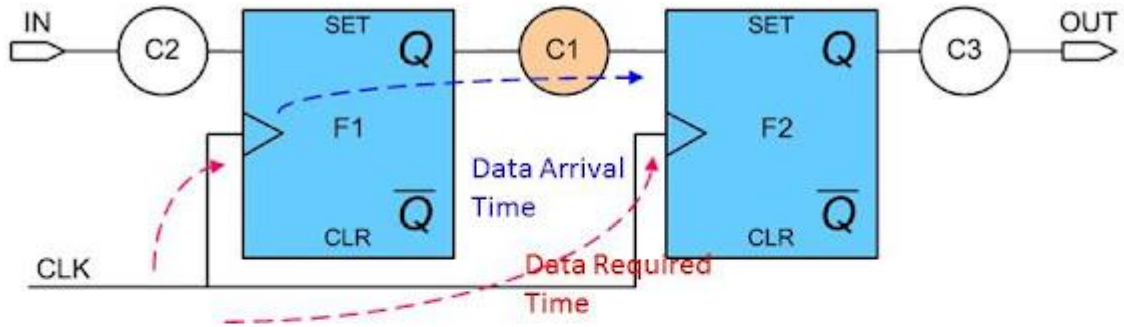
```
module test (); // Function call


  function void fSimple( int a, int b );
    $display( "a=%0d, b=%0d", a, b );
  endfunction


  function void fDefArgVal( int a, int b=1 );
    $display( "a=%0d, b=%0d", a, b );
  endfunction
```

```
  initial begin


    fSimple( 0, 1 );
    fSimple( .a(0), .b(1) );


    fDefArgVal( 0, 1 );          // a=0, b=1
    fDefArgVal( .a(0), .b(1) );  // a=0, b=1
    fDefArgVal( 0 );             // a=0, b=1(default)
    fDefArgVal( .a(0) );         // a=0, b=1(default)


  end
endmodule
```

Argument values provided by position a=0, b=1

Argument values provided by name

Setup and Hold Slack



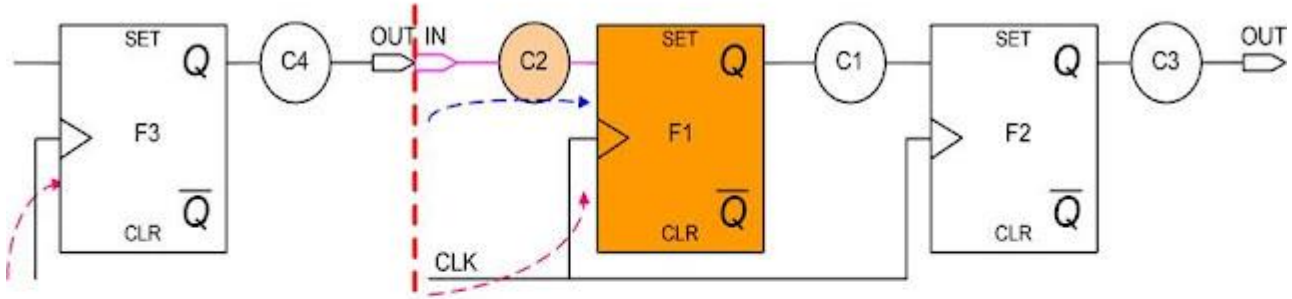
- Setup and hold slack is defined as the difference between data required time and data arrival time.

setup slack= Data Required Time- Data Arrival Time

hold slack= Data Arrival Time- Data Required Time

- A +ve setup slack means design is working at the specified frequency and it has some more margin as well.
- Zero setup slack specifies design is exactly working at the specified frequency and there is no margin available.
- Negative setup slack implies that design doesn't achieve the constrained frequency and timing. This is called as setup violation.

Input to Reg

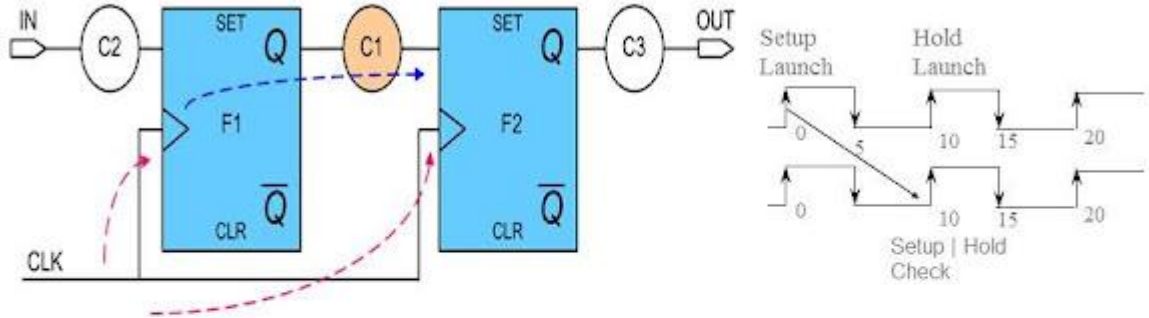


- Data arrival time is the time required for the data to start from input port and propagate through combinational logic and end at data pin of the flip-flop.

Arrival time *=Tcombo*
Required time *= Tclk-Tsetup*

setup slack *= Required Time- Arrival Time*
=(Tclock-Tsetup) – Tcombo

Reg to Reg Path: Setup and Hold Equations



- Data arrival time is the time required for data to propagate through source flip flop, travel through combinational logic and routing and arrive at the destination flip-flop before the next clock edge occurs.

Arrival Time ***= Tclk-q+Tcombo+Tsetup***

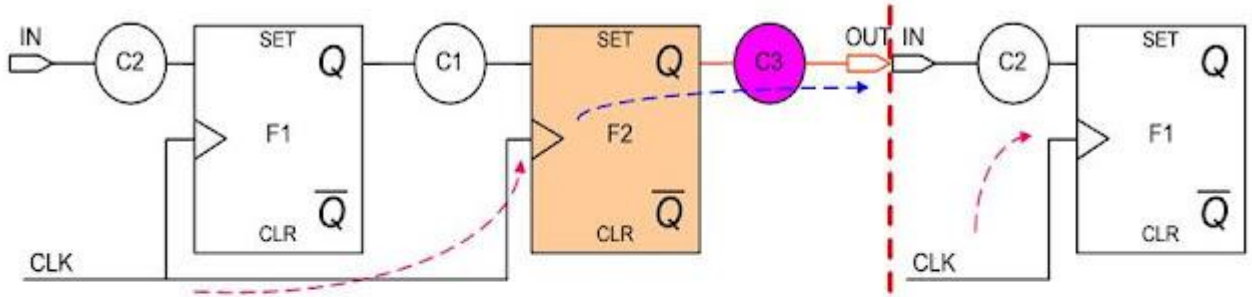
Required Time ***=Tclock***

setup slack ***= Required Time- Arrival Time***

=Tclock– (Tclk-q+Tcombo+Tsetup)

Hold Slack ***= (Tclk-q+Tcombo)-hold***

Reg to Output



- Data arrival time is the time required for data to leave source flip-flop, travel through combinational logic and interconnects and leave the chip through output port.

Arrival time =Tclk-q+Tcombo

Required Time = Unconstrained

Data Required Time

the sum of the delay from a clock source to the clock pin of the destination register, minus the micro setup time of the destination register.

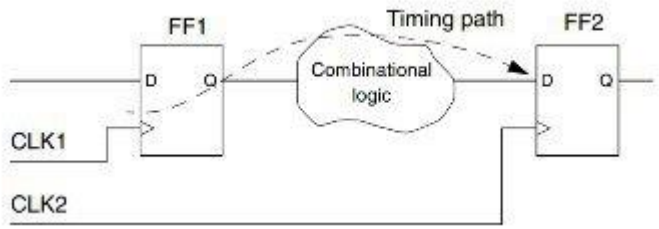
Clock latency

The time taken by Clock signal to reach from clock source to the clock pin of a particular flip flop is called as **Clock latency**. Clock skew can also be termed as the difference between the capture clock latency and the launch clock latency for a set of flops.

False path and multicycle paths are the timing exceptions in the design.

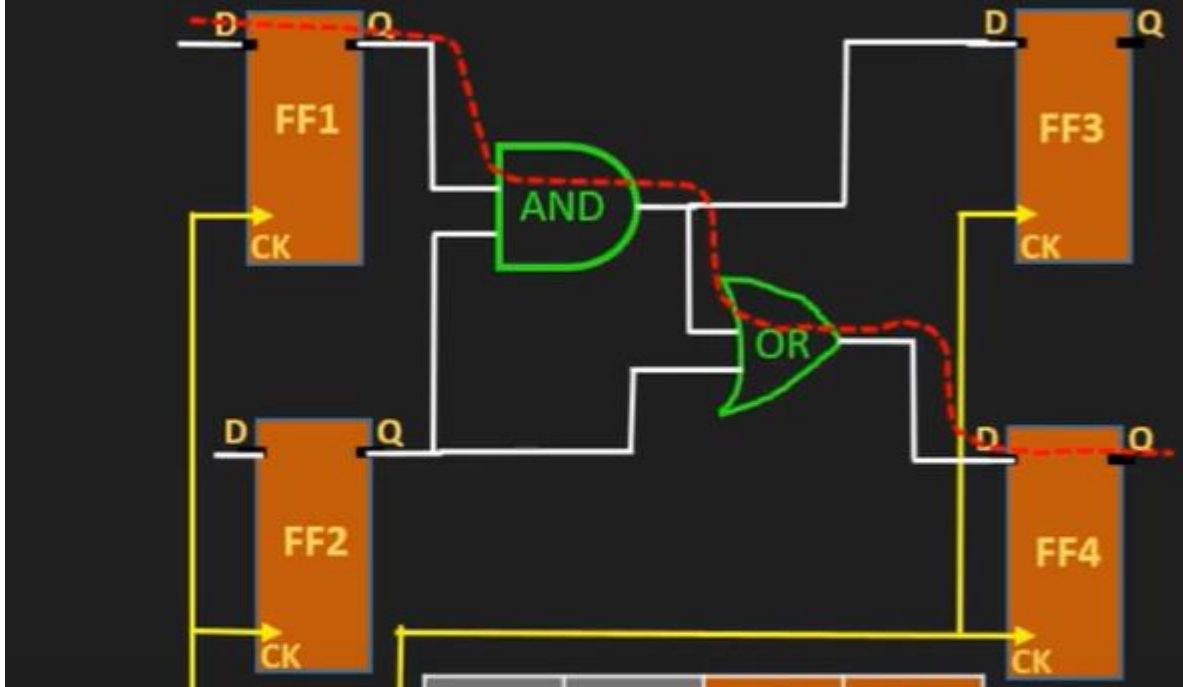
False paths:

Paths in the design which doesn't require timing analysis are called **False paths**. These paths are timing exceptions in the design. Which is commonly occured in the blocks at which more than one clock is involving in the functionality

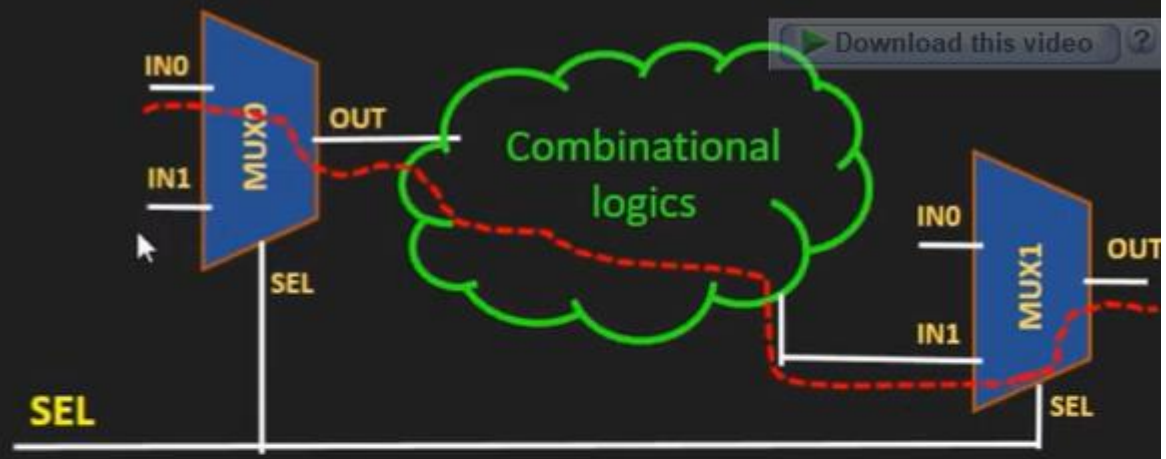


Examples:

2. False Path



FF1/D	FF2/D	FF3/D	FF3/D
0	0	0	0
0	1	0	1
1	0	0	0
1	1	1	1

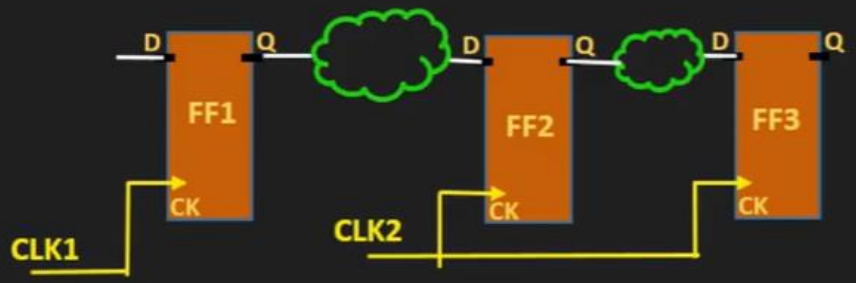


Architectural False Path

SEL	OUT
0	MUX1/IN0
1	MUX0/IN1 → Combo → MUX1/IN1

For any combination of inputs, path
MUX0/IN0 → MUX0/OUT → Combo → MUX1/IN1 → MUX1/OUT
Can not get activated and must be excluded from timing analysis.

Examples:



Clock Domain Crossing (CDC)

SDC Syntax

2. False Path

set_false_path

Specify a false path exception, removing paths from timing analysis

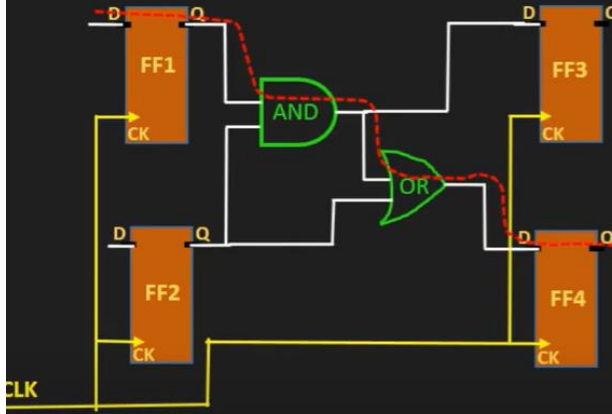
```
set_false_path [-from <names>] [-through <names>] [-to <names>][-setup][-hold]
[-fall_from <names>][-fall_to <names>] [-rise_from <names>] [-rise_to <names>]
```

Examples:

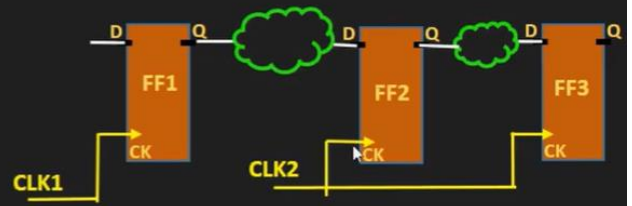
```
Set false path between two unrelated clocks:
      set_false_path -from [get_clocks CLKA] -to [get_clocks CLKB]
Set false path between two flops:
      set_false_path -from {FF1} -to {FF2}
Set false path through U1/Z:
      set_false_path -through [get_pins U1/Z]
```

Examples:

2. False Path



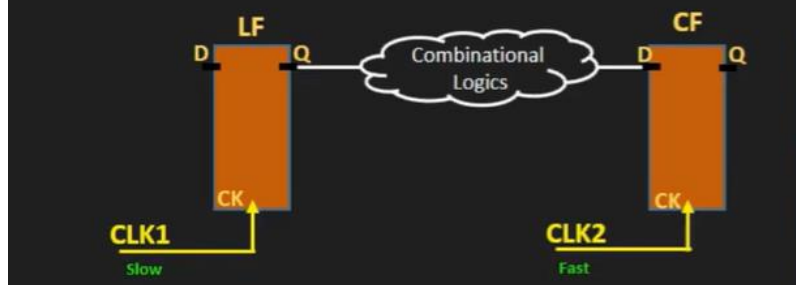
set_false_path -from {FF1} -to {FF4}

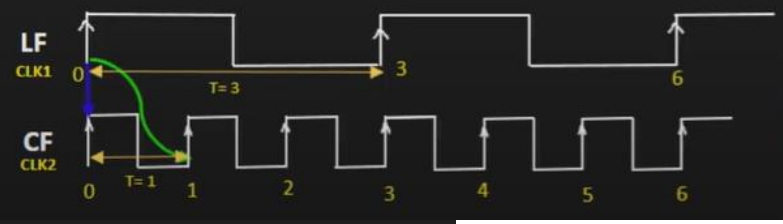


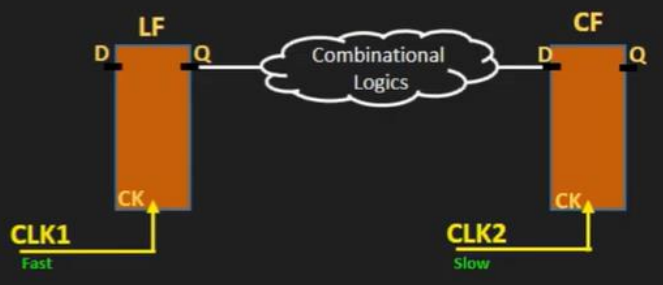
set_false_path -from [get_clock *CLK1*]
-to [get_clock *CLK2*]

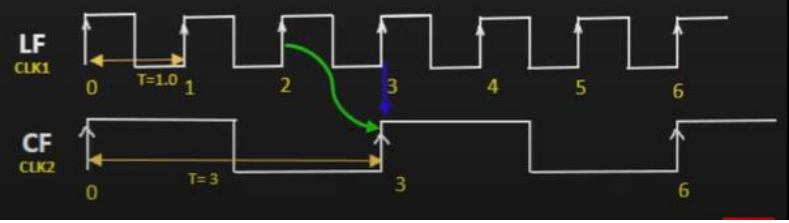
Default setup and hold checks?

Multi frequency clocks



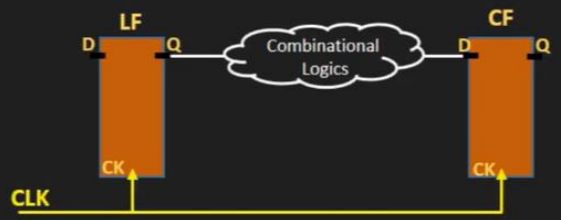






Default setup and hold checks?

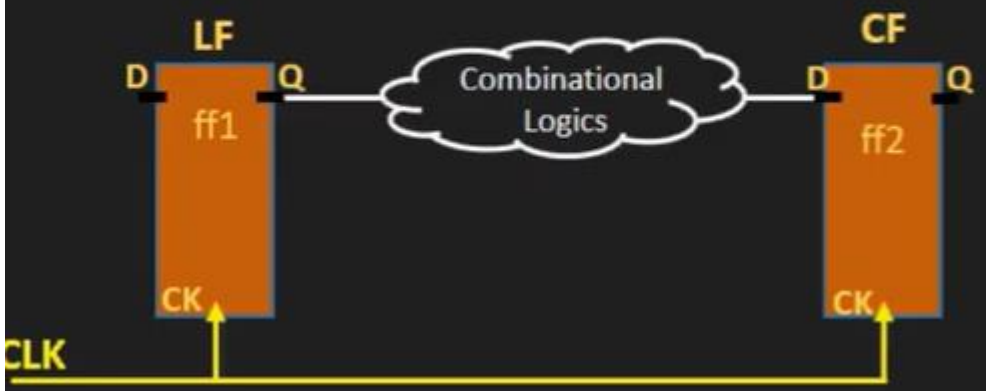
Single clock



How to set Multi Cycle Paths

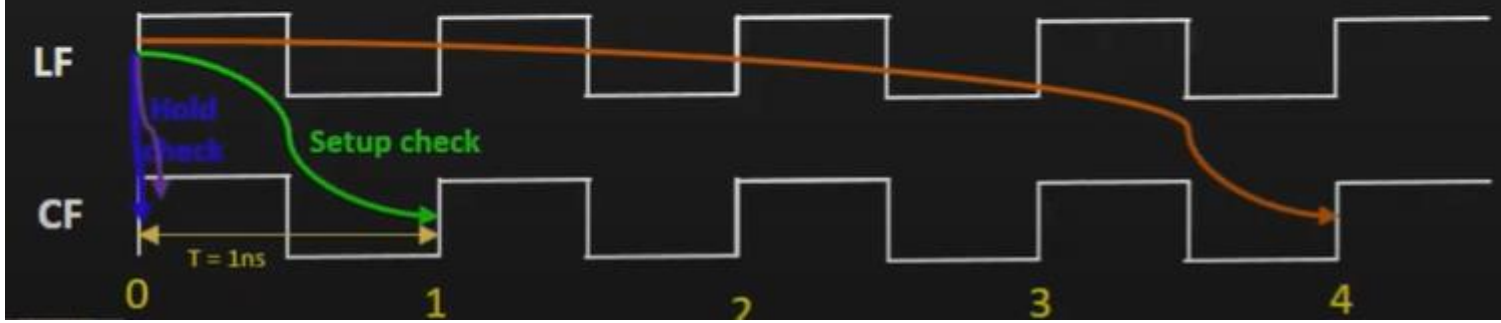
2. Multi Cycle Path

SDC file



set_multicycle_path -setup 4 -from ff1/Q -to ff2/D

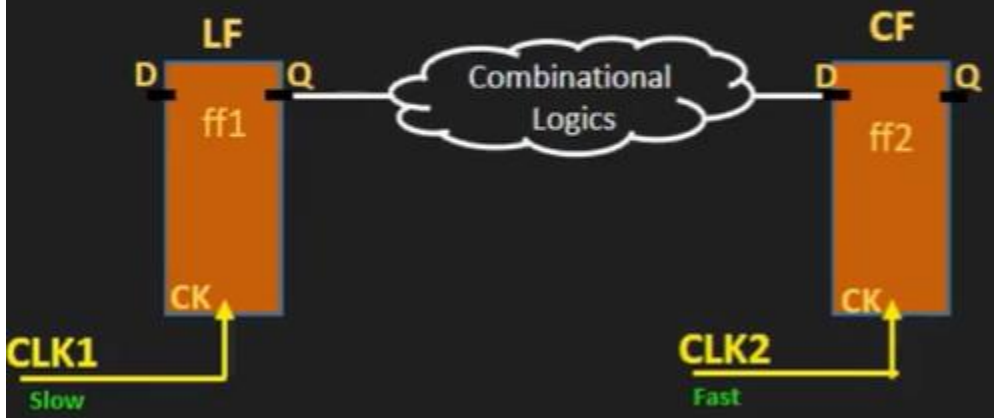
set_multicycle_path -setup 4 -from ff1/Q -to ff2/D
set_multicycle_path -hold 3 -from ff1/Q -to ff2/D



- Applying MCP for setup, hold get affected by same no. of cycles in same direction
- Applying MCP for hold, does not affect the setup check time

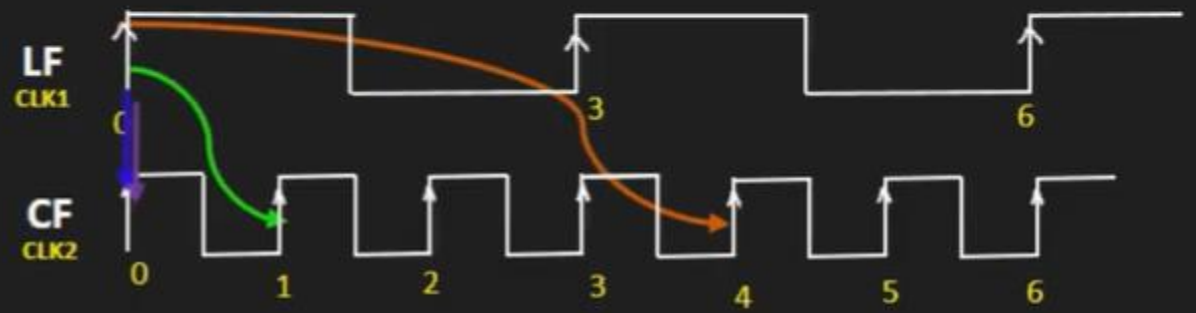
How to set Multi Cycle Paths

2. Multi Cycle Path



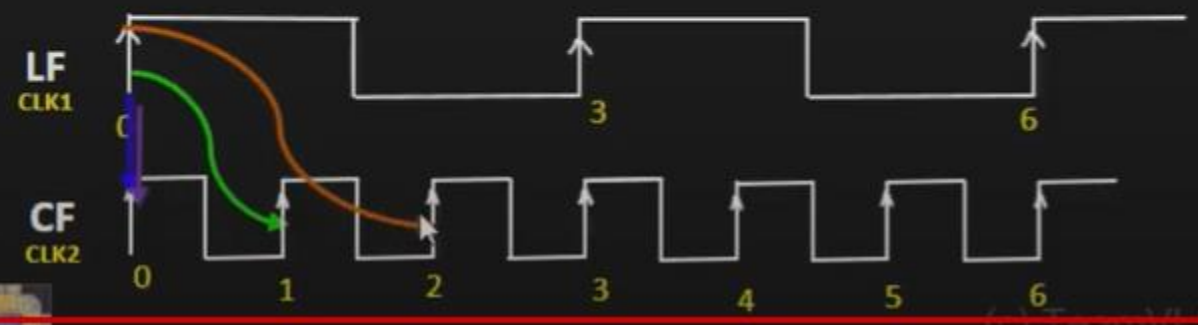
set_multicycle_path -setup 2 -from ff1/Q -to ff2/D -start

set_multicycle_path -hold 1 -from ff1/Q -to ff2/D -start



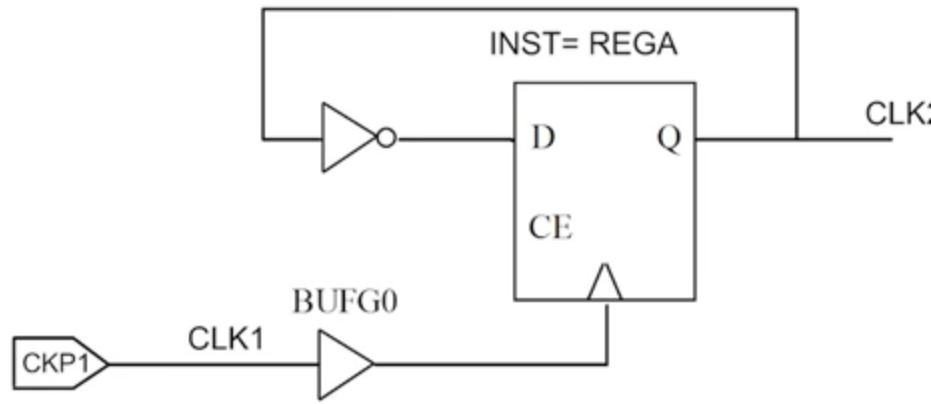
set_multicycle_path -setup 2 -from ff1/Q -to ff2/D -end

set_multicycle_path -hold 1 -from ff1/Q -to ff2/D -end

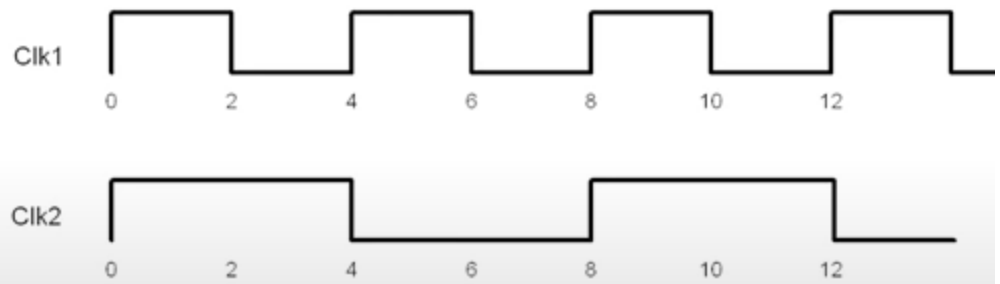


User-Defined Generated Clocks

- **User-created clock modification**
  - No automatic generation



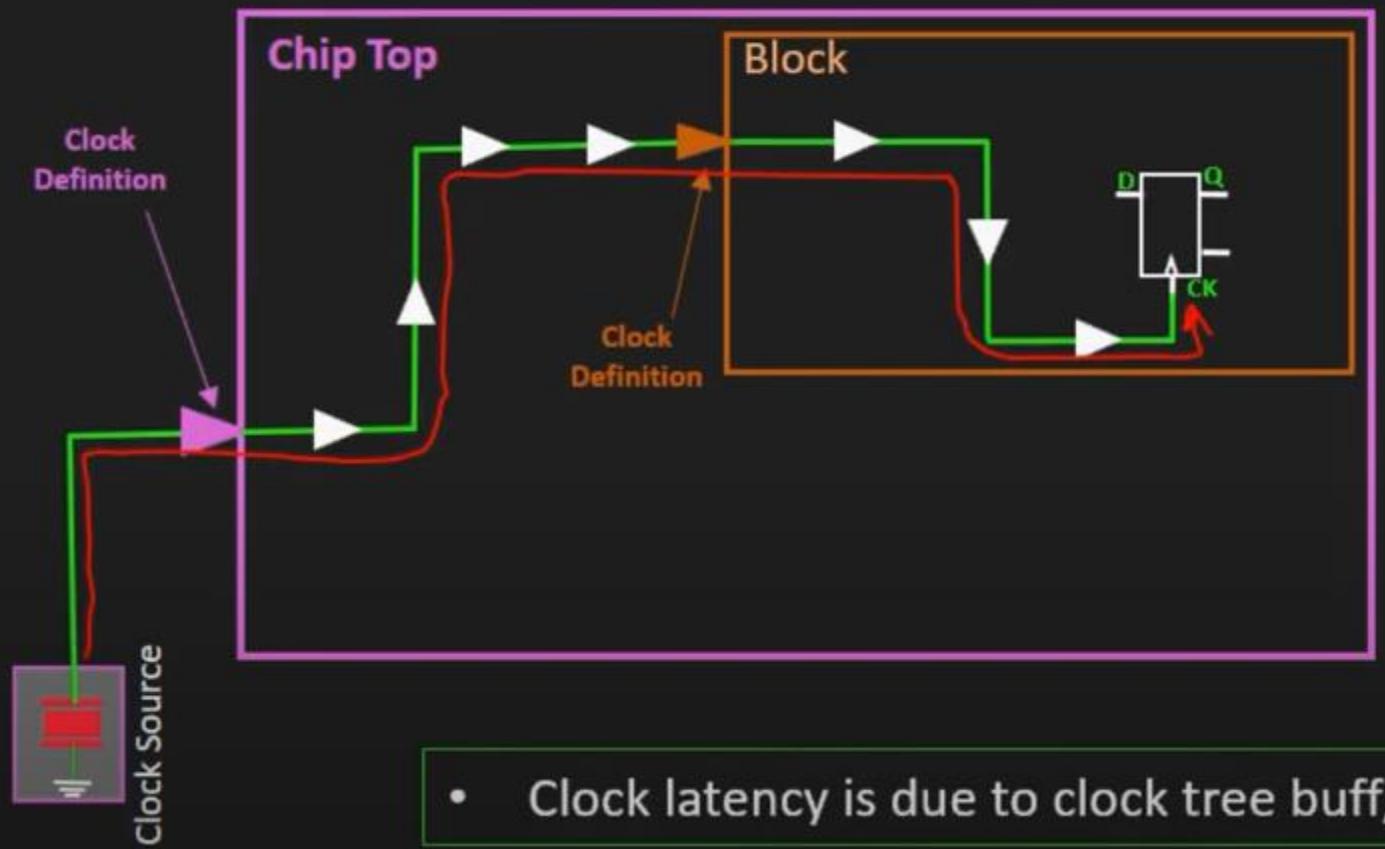
create_clock –name clk1 –period 4 [get_ports CKP1]
*create_generated_clock -name clk2 *
*–source [get_ports CKP1] *
*–divide_by 2 *
[get_pins REGA/Q]

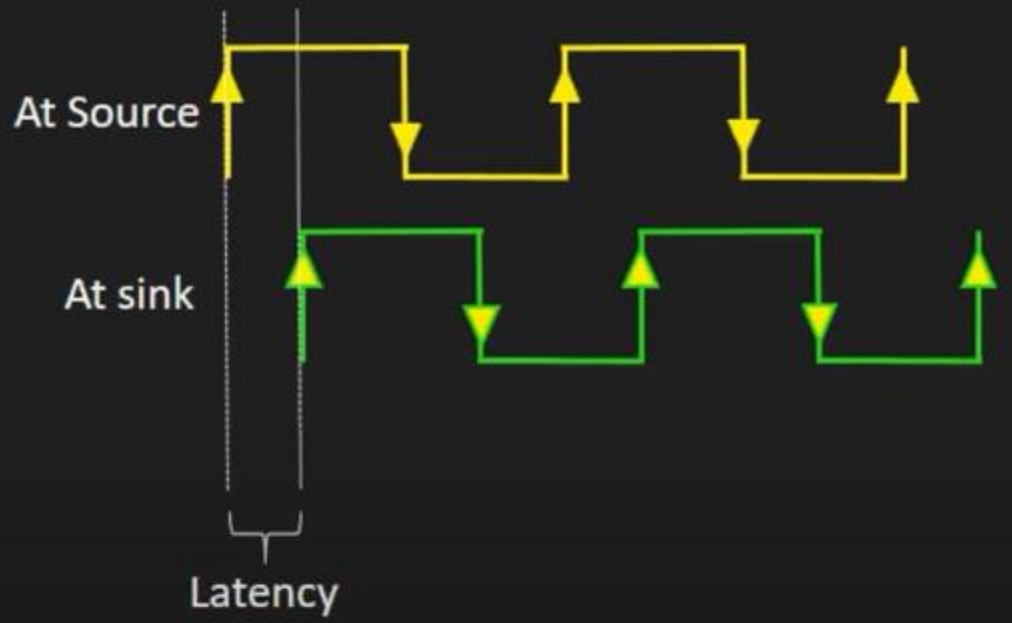


Clock Latency

STA Terminology

- Clock Latency is the **time required** to reach the clock signal from the **clock source pin** to the **sink pin / Destination pin**

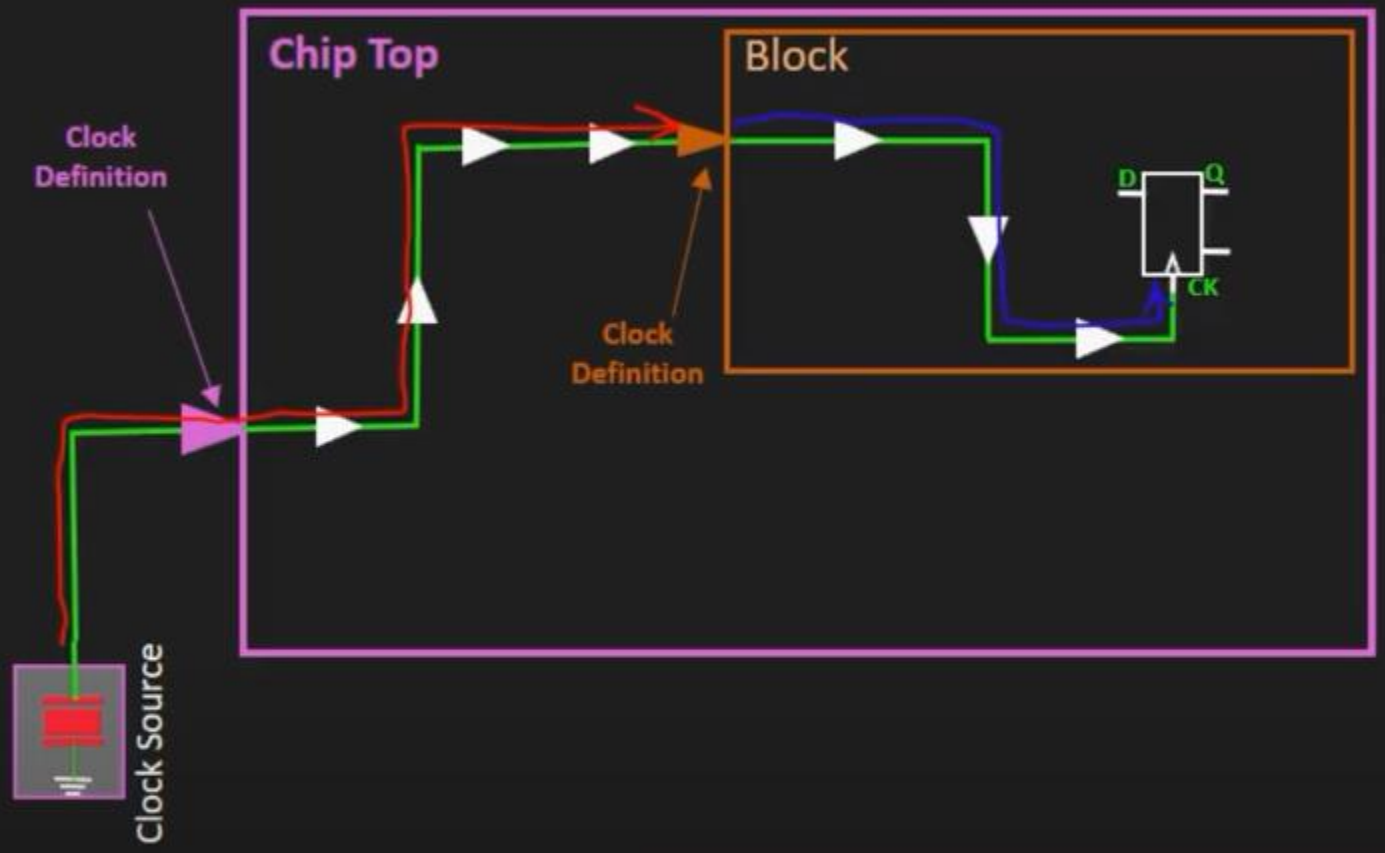




- Clock latency is due to clock tree buff/inv **cell delay** plus the clock **net delay**

Source Latency & Network Latency

STA Terminology



- Clock latency further could be divided into parts:
  - I. Source Latency
  - II. Network Latency

I. Source Latency:

- The time required to reach the clock signal to clock definition point from the clock source point is called Source Latency

II. Network Latency:

- The time required to reach the clock signal to the sink pin from clock definition point is called Network Latency

Clock Latency = Source Latency + Network Latency

Latency in SDC file

Clock latency information is provided in SDC file through the sdc command

set_clock_latency

Example:

- **Set network latency**

  set_clock_latency <value> [get_clocks <CLK_name>]

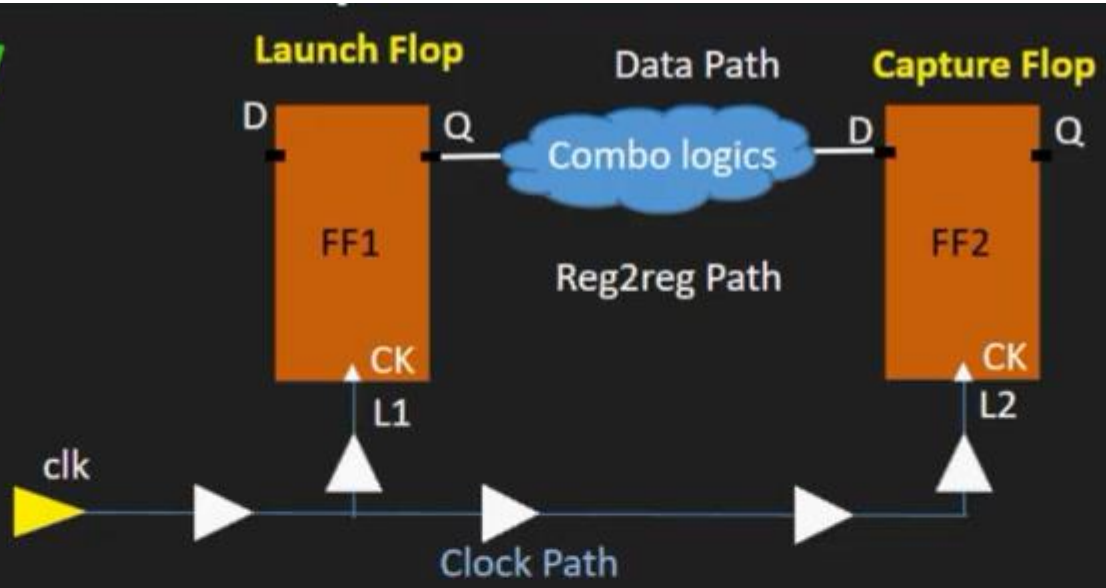
  set_clock_latency 0.8 [get_clocks coreClk]

- **Set Source latency**

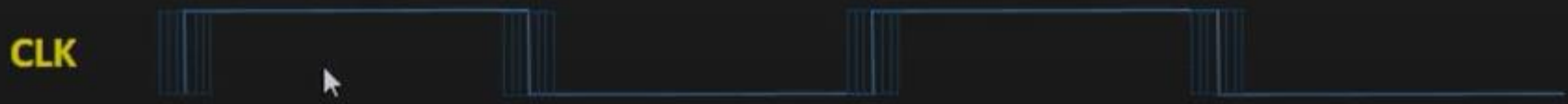
  set_clock_latency <value> [get_clocks <CLK_name>] –source

  set_clock_latency 1.2 [get_clocks coreClk] -source

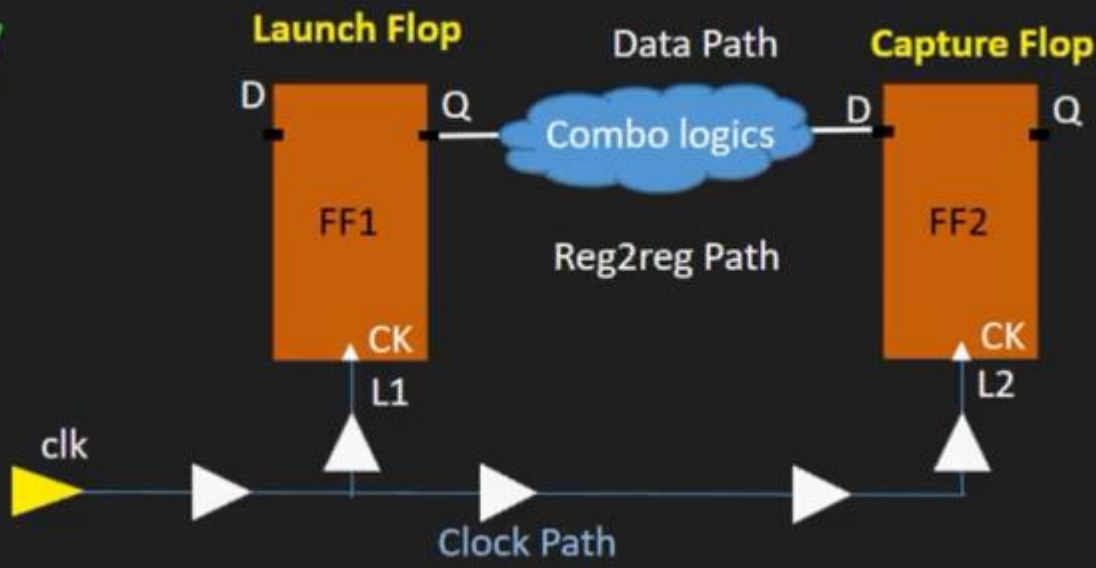
Clock uncertainty



- In practice no clock source could be so ideal that it is free from jitter, There is always some variation in clock period in the source of clock itself



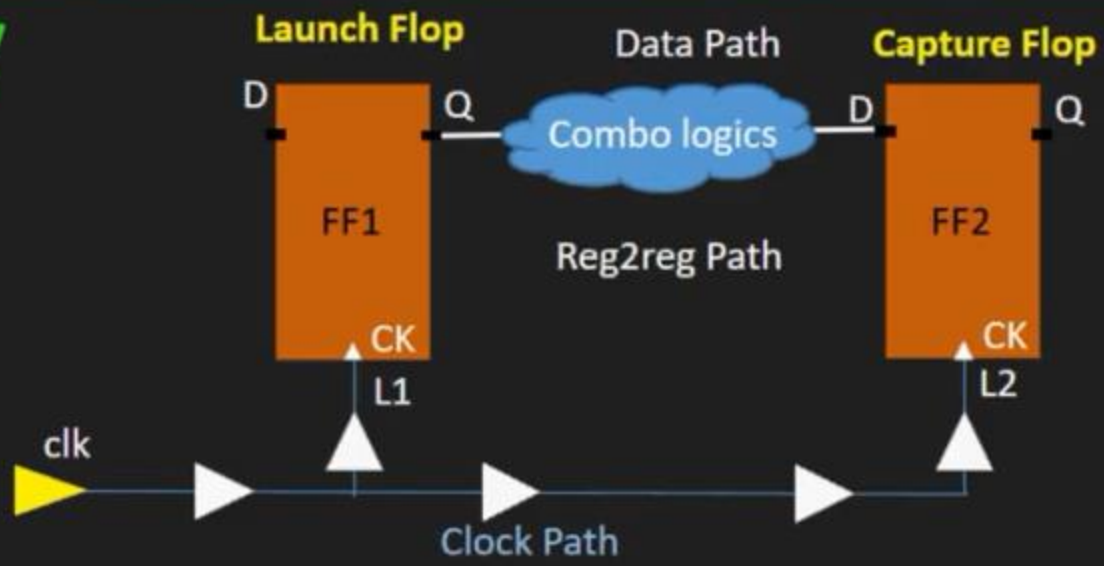
Clock uncertainty



- There is always some skew in a clock tree, so launch and capture flop will get the clock edge different point of time

Clock uncertainty

STA Terminology

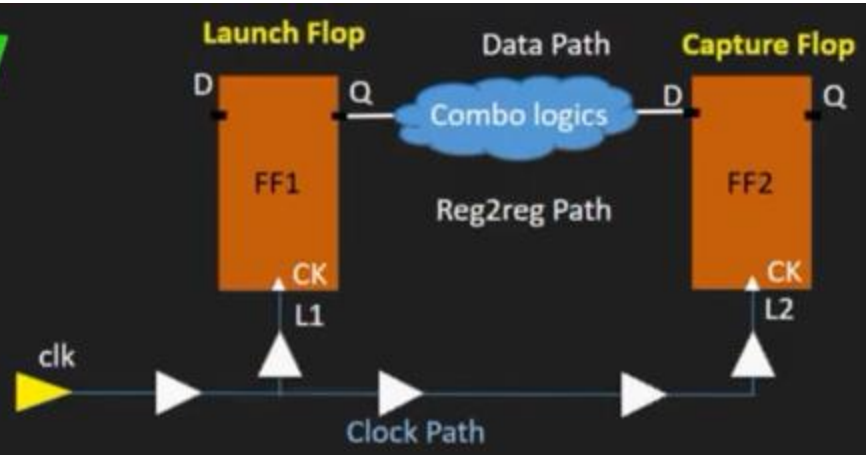


- Clock tree may suffer from many variations (like: crosstalk, IR, Process Variation etc) which can affect the clock tree delay of individual clock paths. These variation may vary the clock edge arrival time at sink pin
- Clock Uncertainty is used to model all these factors (discussed here) which could affect the clock period
- Clock Uncertainty specifies a window within which clock edge may arrive at any point of time

Clock uncertainty

STA Terminology

Clock uncertainty



STA Terminology

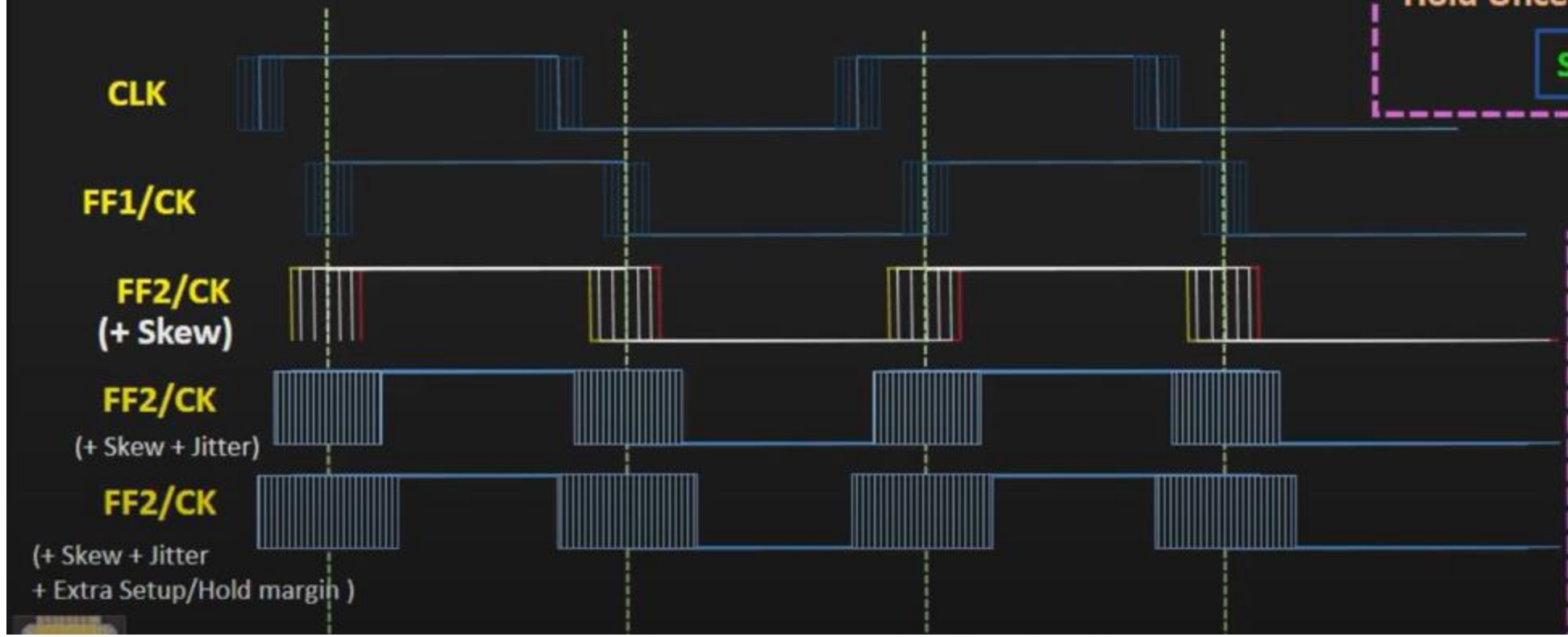
PreCTS Stage:

Setup Uncertainty:

Jitter + Skew + Setup margin

Hold Uncertainty:

Skew + Hold margin



PostCTS Stage:

Setup Uncertainty:

Jitter + Setup margin

Hold Uncertainty:

Hold margin

t Clock Uncertainty

- set_clock_uncertainty

```
set_clock_uncertainty –setup 0.15 [get_clock clk_core]
set_clock_uncertainty –hold 0.05 [get_clock clk_core]
```

Setup analysis (AAT < RAT):

Required Arrival Time (RAT)

= T + Skew – Setup Uncertainty – setup

= 1000ps +100ps – 150ps – 30ps

= 920ps

Hold analysis (AAT > RAT):

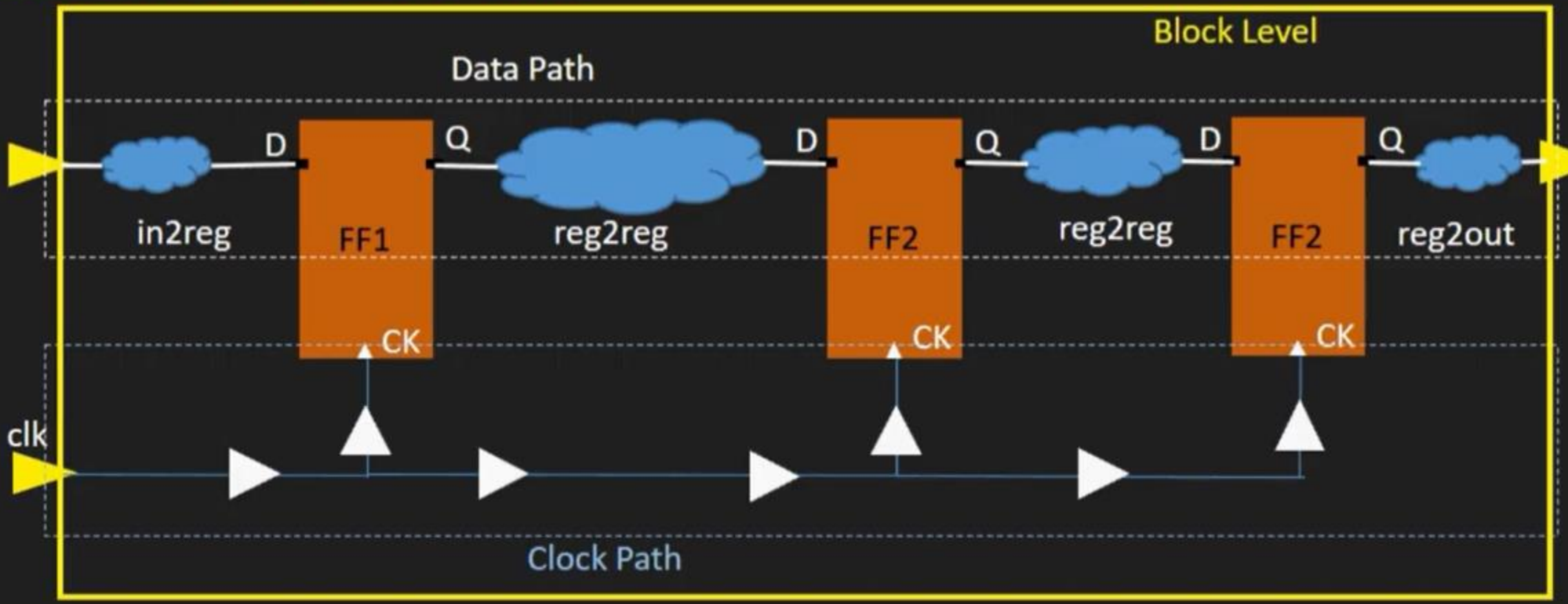
RAT

= Hold time + Skew + Hold Uncertain

= 20ps + 100ps + 50ps

= 170ps

Data path and Clock Path



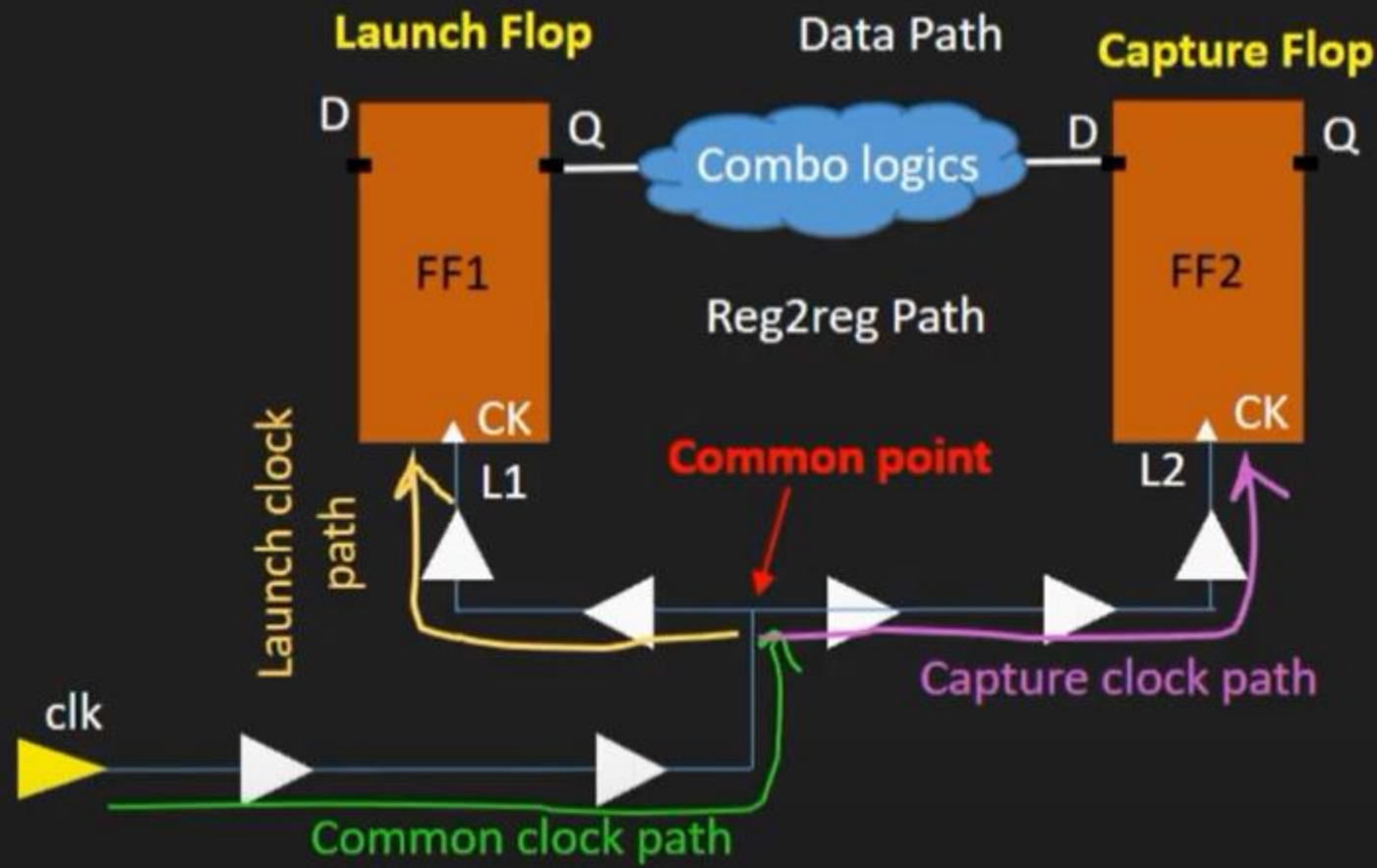
Data path:

- The path through which data travels from input pin to output pin of block.

Clock path:

- The path through which clock is being distributed to all sequential elements in the design from the clock source

Launch Clock path and Capture clock path



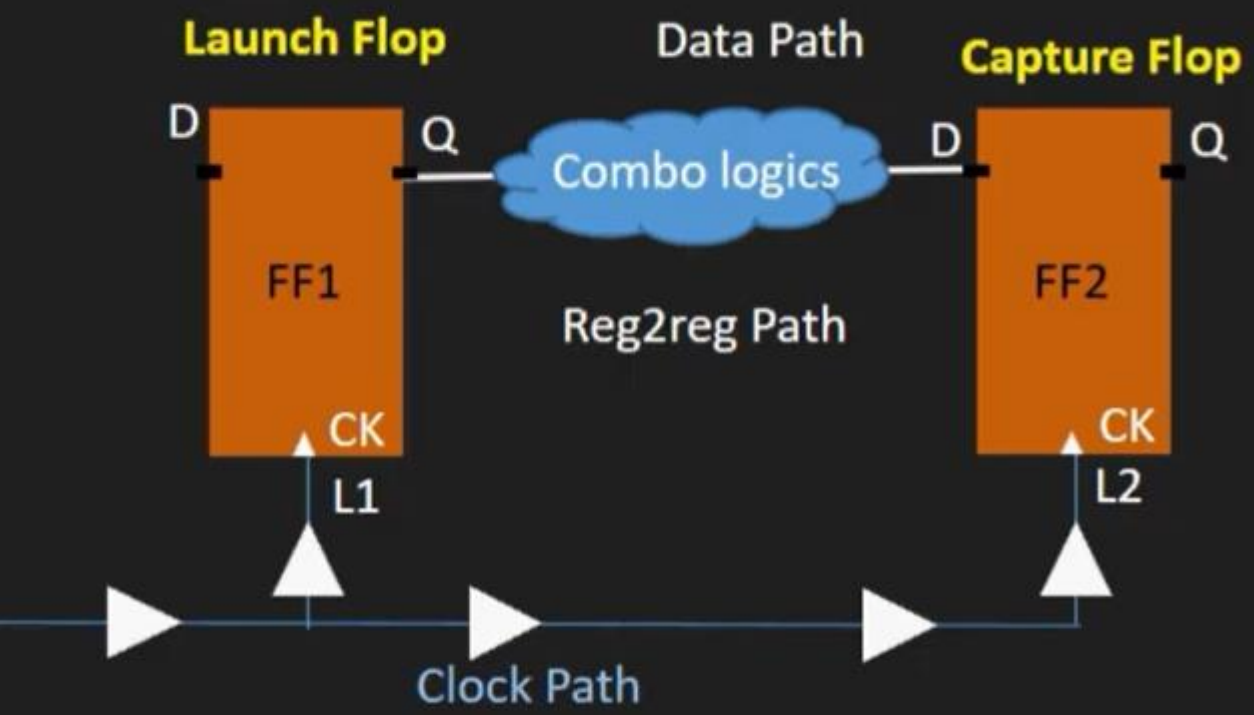
Launch clock path:

- The clock path from the common point to the launch flop is called Launch clock path

Capture clock path:

- The clock path from the common point to the capture flop is called Capture clock path

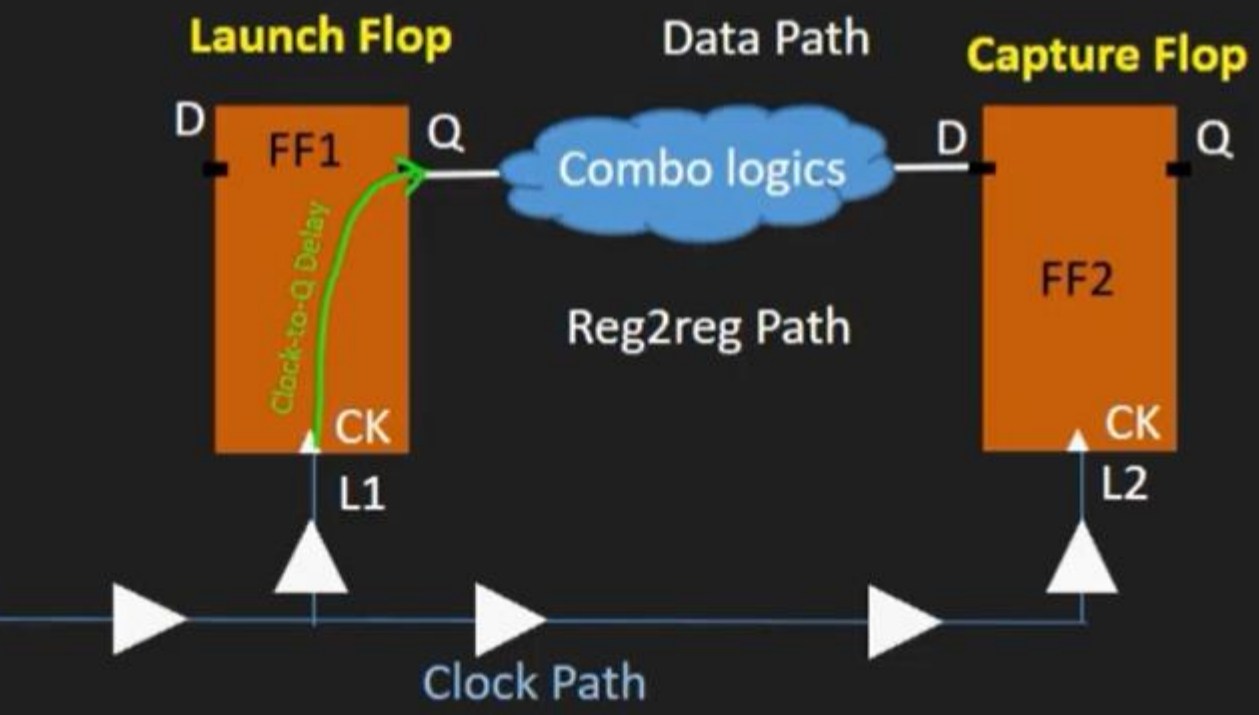
aunch Flop and Capture Flop



- In reference of reg2reg path, Data is being launched FF1 on a positive edge of the clock and the it will tra through the combinational path and finally will captured by FF2 on the next clock edge by default
- The flop which launches the data (FF1) is called **lau flop** and the flop which captures the data (FF2) is ca **Capture flop**.

- In a path group it is not necessary that both end of path will always have a register (e.g : in2reg, reg2c reg2mem)
- In such cases instead of saying launch and capture flop, They are generally called by **start point** and **end poi**

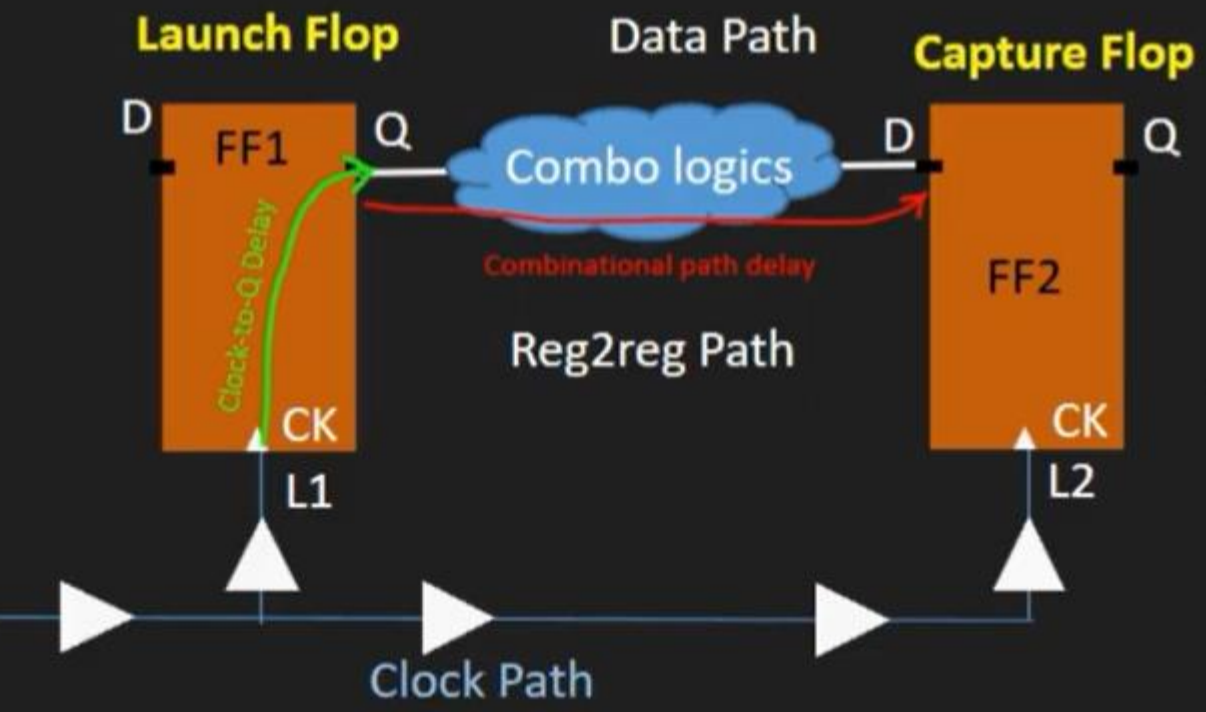
Clock to Q Delay



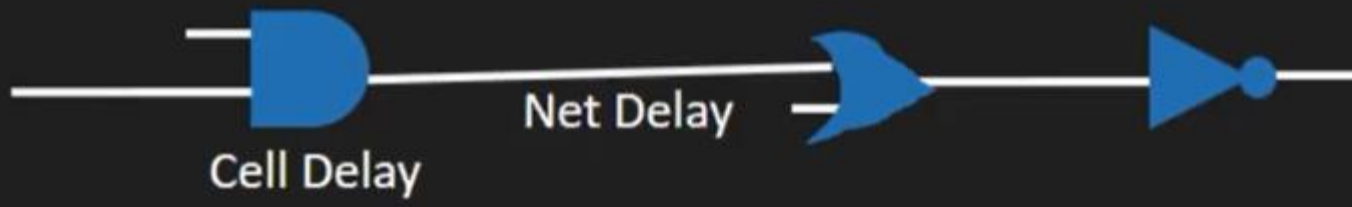
- The moment when active edge of clock reach the flop, data is not being launched instantly. It take so fixed amount of time to launch the data.
- The delay between arrival of active edge of clock Data launch is called **Clock-to-Q delay**

Data arrival time = Clock-to-Q delay + Combinational path delay

et Delay and Cell Delay



Data arrival time =
Clock-to-Q delay + Combinational path delay



Combinational Delay = Σ Cell Delays + Σ Net Delays

- Each cell has some specific propagation delay defied in the lib file for different process corners
- A net is combination of R and C, so each segment of net has some delay which is called net delay
- Net Delay variation is based on RC corners
- A combinational path delay is sum of Cell delays on the path and net delays