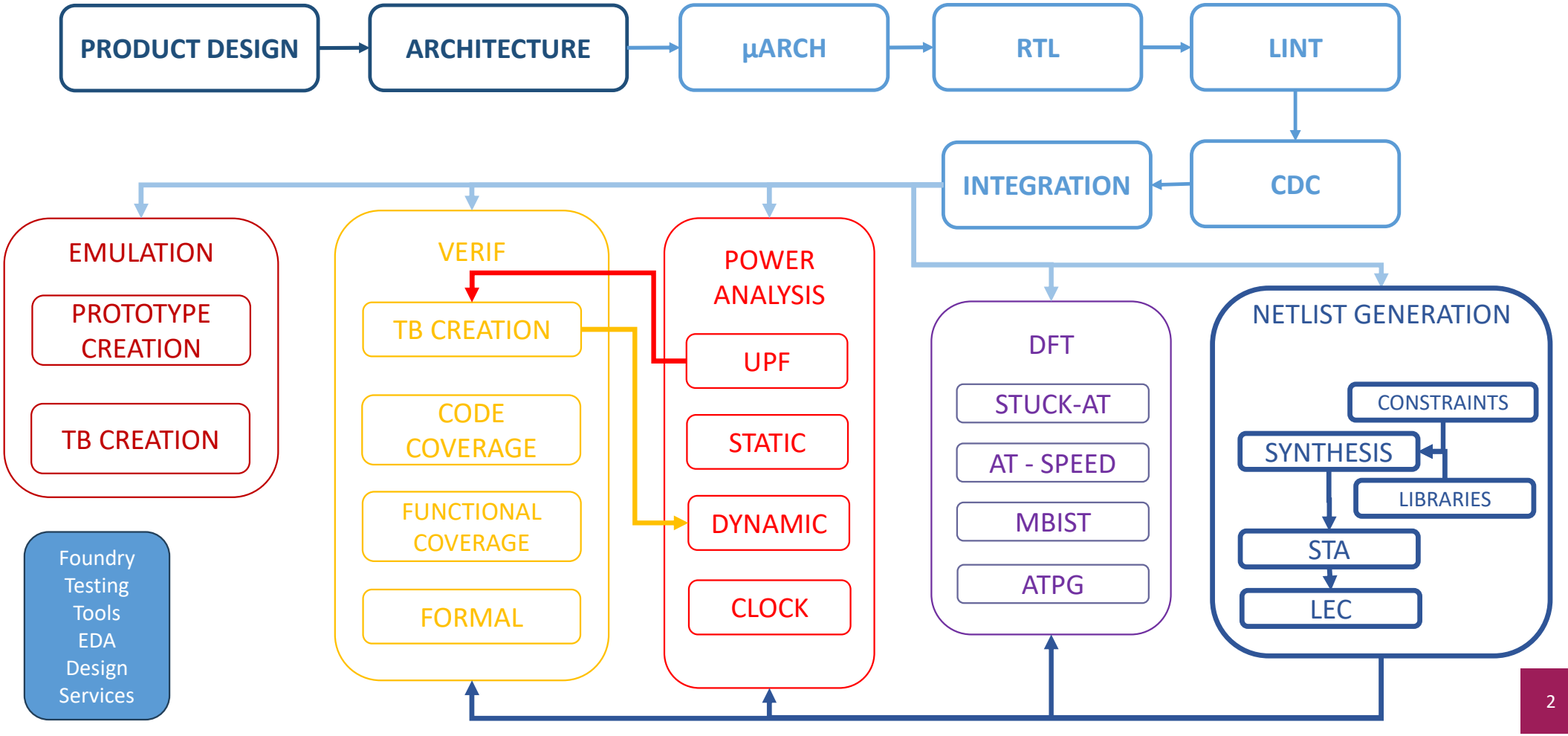
RAMAIAH
SKILL ACADEMY

VERILOG for Synthesis

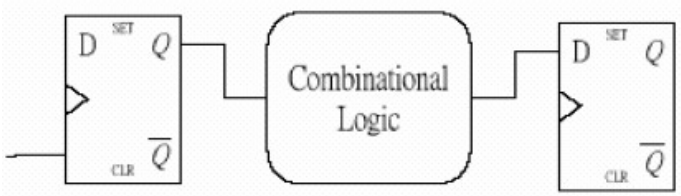
Vasudev Murthy

ASIC Design : FRONT END

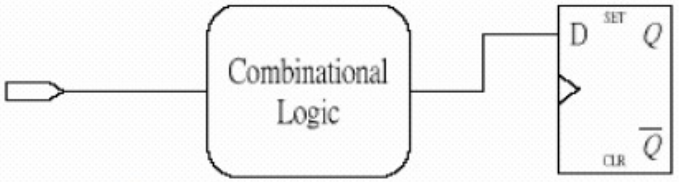
RAMAIAH SKILL ACADEMY



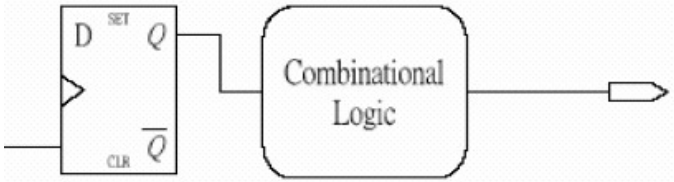
Timing paths



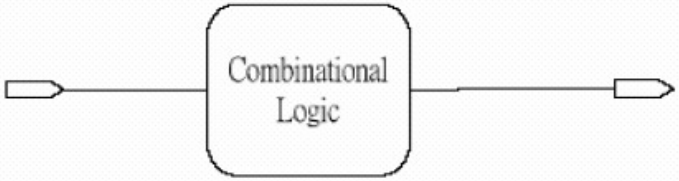
Start Point:
clock pin of sequential device
End Point:
data input pin of sequential devices



Start Point:
primary input port
End Point:
data input pin of sequential devices

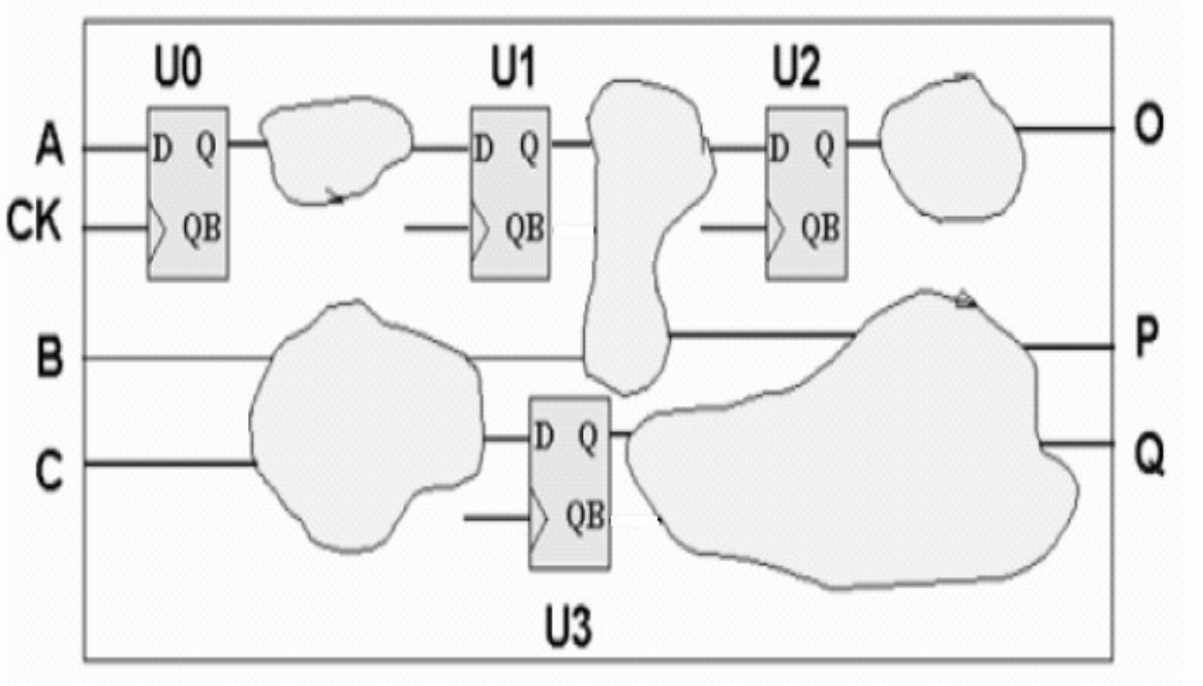


Start Point:
clock pin of sequential device
End Point:
primary output port



Start Point:
primary input port
End Point:
primary output port

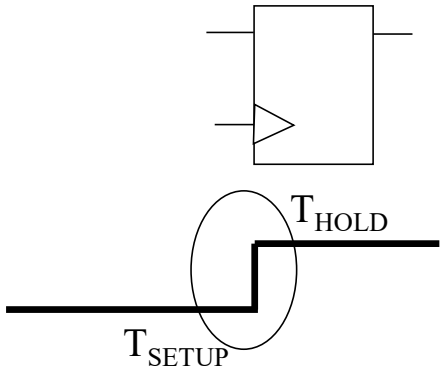
Timing paths



- This circuit has
- How many start points ?
- How many end points ?
- How many paths ?

Timing Terminologies

- Setup Time : Time for which data should be stable at the input of the ff **before** the arrival of clock at the ff's clock pin
- Hold Time : Time for which data should be stable at the input of the ff **after** the arrival of clock at the ff's clock pin



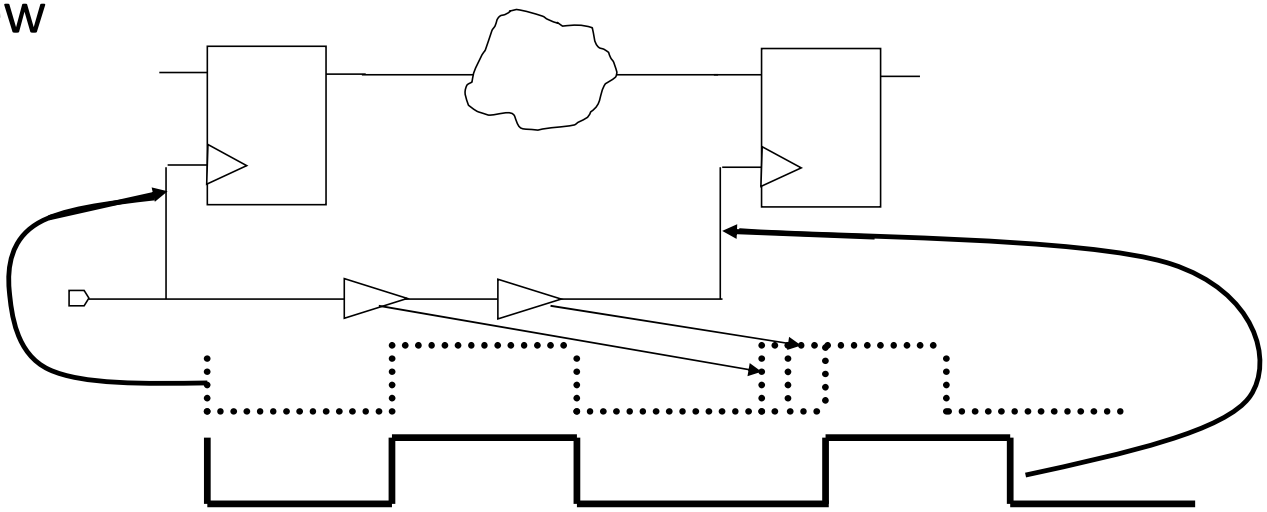
Timing Terminologies

RAMAIAH SKILL ACADEMY

- Critical Path : Theoretically path which has maximum delay
- Arrival Time : Time taken by data to reach a particular end point from a specific start point. Depends on complexity of logic through which data traverses
- Required Time : Time at which data is required at a particular end point. Depends on the requirements / specifications
- Slack : Difference in required time and arrival time. For a design to work the slack value should always be positive

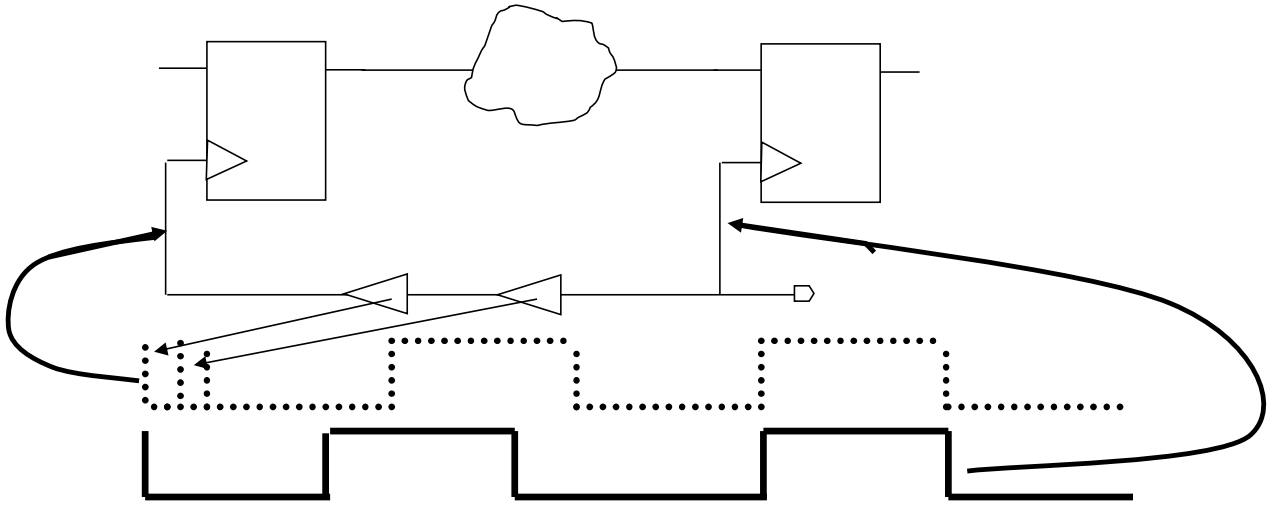
Clock Skew

- Clock feeds multimillion flip-flops
- Theoretically clock should arrive at same instance at all flip-flops – Practically impossible
- Frequency of clock may vary from cycle to cycle
- Variation in arrival of clock at clock pin of subsequent / consecutive flip-flops is known as skew



Clock Skew

- The difference in arrival of clock at different flip-flops could be due to
  - Jitter – cycle to cycle variation in clock period due to aging of oscillator, anomalies in the pll etc
  - Clock network delay – Uncertainty in arrival of clock due to clock network and clock network component delays

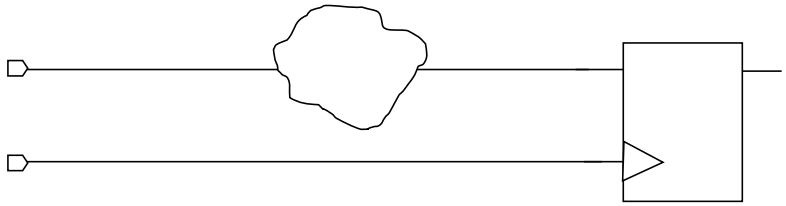


Clock Skew

- Skew can be considered to be of two types
  - When clock arrives earlier than expected – generally known as negative skew
  - When clock arrives later than expected – generally known as positive skew
- +ve skew can occur when data & clock travel in same direction
- -ve skew can occur when data & clock travel in opposite directions

Frequency Calculation

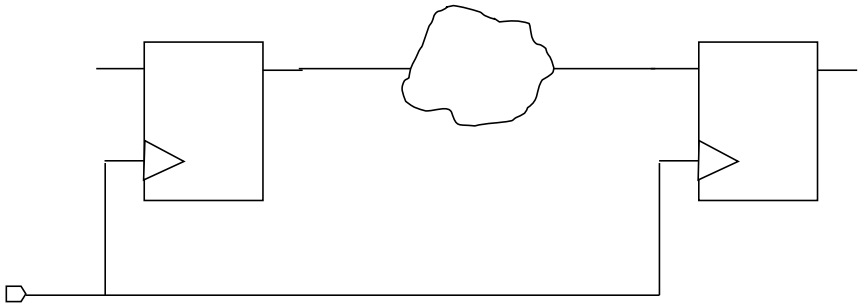
- Arrival Time = $T_{COMBO, Max}$



- Required Time = $T_{CP} + T_{Clock\_Insertion\_Delay} - T_{SETUP}$
  - Since there is only one clock here instead of skew the possible delay on the clock line is denoted as insertion delay

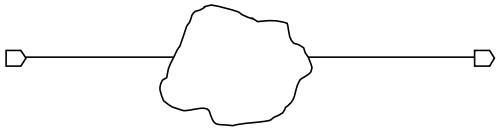
- Arrival Time : $T_{Clk-Q} + T_{Combo, MAX}$

- Required Time = $T_{CP} + T_{Clock\_Skew} - T_{SETUP}$
  - Due to presence of 2 clocks the possible difference in arrival of clock is denoted as skew



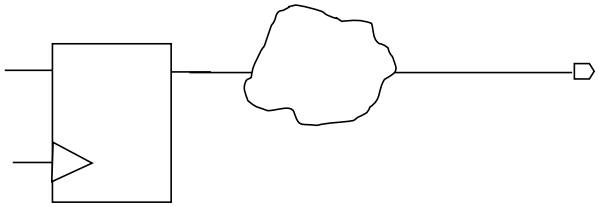
Frequency Calculation

- Arrival Time : $T_{Combo, MAX}$



- Required Time : Explicit Timing Constraint
  (If Specified)

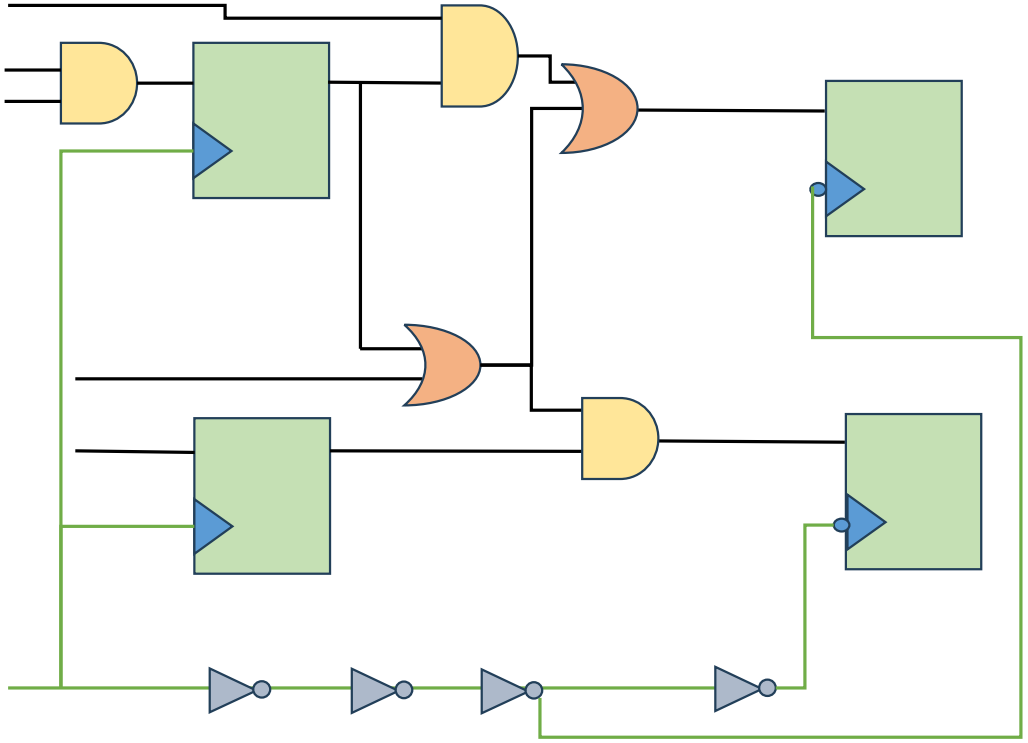
- Arrival Time = $T_{Clk\text{-}Q} + T_{Combo, MAX}$



- Required Time = Typically Unconstrained Path
  (Has no requirement set)

Max Frequency Calculation

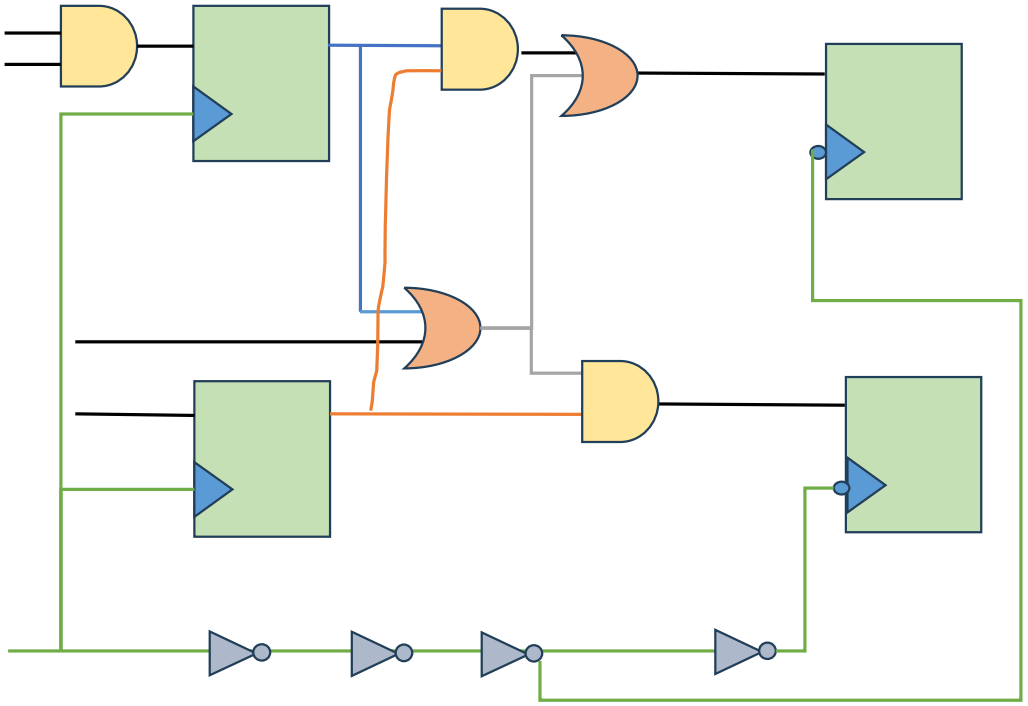
- Find the maximum Frequency for the circuit shown in figure below



CLK-Q	2
Setup	1
AND	1.5
OR	1
INV	0.5

Max Frequency Calculation

- Find the maximum Frequency for the circuit shown in figure below



	R Max	R Min	F Max	F Min
CLK-Q	2	1.9	2.2	2.1
Setup	1	1	1	1
AND	1.5	1.4	1.3	1.2
OR	1.0	0.9	1.2	1.1
INV	0.5	0.5	0.5	0.5

Synthesis Simulation Mismatches

- RTL engineers write code to meet the functional requirements of a specification
- The functional code may not be optimized for synthesis
- Any RTL developed without due emphasis on synthesis is bound to fail
- A functionally perfect code may just not work after synthesis
- For design to be functionally correct even after synthesis some basic practices that need to be followed will be discussed

Incomplete Sensitivity List

- Any process / procedural block which has incomplete sensitivity list is bound to create a mismatch between synthesis and simulation
- Generation of hardware is not dependent on sensitivity list but simulation does depend on it

```
module mux ( output reg z, input a,b, select)

        always@(a,b)
                if(select) z = a;
                else z = b;
endmodule
```

Synthesizes to a mux but output does not follow 'select' in simulation

Blocking in Sequential block

- Blocking mimics the combinational logic hardware
- Should be used with care in sequential blocks

```
module wrong_blocking(output reg z , input a,b,c)
        reg d;

        always@(posedge c)
                begin
                        d = a & b;
                        z = d;
                end
endmodule
```

During simulation ‘d’ changes only at posedge clock whereas the output of synthesized ‘and’ gate (real hardware) changes as soon as input changes

Non blocking in combo block

- Non-Blocking mimics the sequential logic hardware
- Should not be used combinational blocks

```
module wrong_non_blocking(output reg z , input a,b,c)
        reg d;

        always@(a,b,c)
                begin
                        d <= a & b;
                        z <= d & c;
                end
endmodule
```

During simulation ‘z’ changes only after there are changes in ‘a,b,c’ at two different instances whereas ‘z’ from the synthesized ‘and’ gates changes as soon as there is change in inputs

Read before write

```
module rbw (
output reg out,
input a, b, c, d );

reg temp;

always @(a,b,c,d)
  begin
        o = a & b | temp;
        temp = c & d;
  end
endmodule
```

Whenever there is a change in any of the inputs ‘o’ gets updated with the previous value of temp in simulation But in actual hardware this doesn’t happen

Operators

- Some operators are either not synthesizable or synthesize into something else than its own functionality
- Divide operator synthesizes in ASIC unconditionally unlike in FPGAs
- Power operators synthesize iff the operands involved are in powers of 2
- Arithmetic right shift operators synthesize as normal right shift operators
  - Leading to synthesis simulation mismatches

For Loops

```
module memory (
output reg [7:0] mem_out ,
input [7:0] mem_in,
input [6:0] addr,
input rw, clear_n, clock);

integer j;
reg [7:0] memory [127:0];

always@(posedge clock)
    if(!clear_n)
            for(j = 0; j < 128; j = j + 1)
            memory[j] = 8'b0;
    else
            for( j = 0; j < 128; j = j +1)
            memory[j] = mem_in;
endmodule
```

- For loops execute in one delta
- There is no delay between any two iterations of the for loop

For Loops

- Unnecessary for loops
- Leads to extra hardware

```
module for_loop (
output reg [3:0] sum,
input clr);

reg [3:0] value [6:0];
integer j;

   always@(*)
         if(clr)     sum = 4'b0;
         else
                     for (j =0; j < 3; j = j +1)
                     sum = sum + value[j]
endmodule
```

Incomplete Ifs

RAMAIAH SKILL ACADEMY

'If' the single most important cause for most design failures

```
module incomplete_if (
output reg what,
input data, control);

  always@(*)
        if(control)
        what <= data;

endmodule
```

```
module in_complete_if (
output reg what,
input data, next, control);

        always@(*)
            begin
                if(control)  what <= data;
                if(!control) what <= next;
              end
endmodule
```

Incomplete Ifs

- If statement can be incomplete even in the presence of else !!
- If statements can be incomplete in spite of exercising all the options !!

```
module incomplete_if_else (
output reg y,z,
input data, in, control);

  always@(*)
        if(control) y <= data;
        else        z <=  in;

endmodule
```

```
module mux_4_1
(output reg z,
input i0, i1, i2, i3,
input [1:0] s);

always@(*)
        if (s == 2'd0) z = i0;
  else  if (s == 2'd1) z = i1;
        else     if (s == 2'd2) z = i1;
                 else     if (s == 2'd3) z = i1;
endmodule
```

Incomplete Ifs

```
module incomplete_iff (
output reg what, when,
input data, control, clock);

        always@(posedge clock)
                begin
                    if(control)
what = data;
                    when = what +
control;
                end

endmodule
```

```
module incomplete_iff (
output reg what, when,
input data, control, clock);

        always@(posedge clock)
            begin
                if(control) what <= data;
                when <= what + control;
            end


endmodule
```

- Non-Blocking turns the tide in flip-flops favour

Incomplete Case

```
module alu #(parameter size = 5)
( output reg [2*size-1:0] result,
   input       [size-1:0] data_1, data_2,
    input       [2:0]    operation);

always@(operation, data_1, data_2)
   case(operation)
          3'b000 : result = data_1 + data_2;
          3'b001 : result = data_1 – data_2;
          3'b010 : result = data_1 + 1'b1;
          3'b011 : result = data_1 – 1'b1;
          3'b100 : result = data_1 * data_2;
   endcase

endmodule
```

Imperfect Case

```
module imperfect_case  (
output reg [2:0] result_1, result_i,
input       [1:0] data_1, data_2,
input       [1:0]    operation);

always@(operation, data_1, data_2)
  case(operation)
        2'b00 : result_1 = data_1 + data_2;
        2'b01 : result_1 = data_1 – 1'b1;
        2'b10 : result_i = data_1 + 1'b1;
        2'b11 : result_i = data_1 – data_2;
  endcase
endmodule
```

Synthesis of Case

- Well constructed Case statement synthesizes into multiplexer
- Case can be considered well constructed when it is both full as well as parallel
- A case can be considered as full if for an ‘n’ bit case selector the case has 2^n branches (case items)
- A case can be considered as parallel if all the branch conditions are unique

Full Case?

```
module full_case  (
output reg [2:0] result_1, result_i,
input       [1:0] data_1, data_2,
input       [1:0]    operation);

always@(operation, data_1, data_2)
   case(operation) // synopsys_full_case
          2'b00 : result_1 = data_1 + data_2;
          2'b01 : result_1 = data_1 – 1'b1;
          2'b10 : result_i = data_1 + 1'b1;
          2'b11 : result_i = data_1 – data_2;
   endcase
endmodule
```

Priority of Signals

RAMAIAH SKILL ACADEMY

```
module some_gate (
output reg q,
input a,b,c,d);

  always@(negedge a, negedge b, posedge
  d)
                if (b) q = 1'b0;
        else    if (a) q = c;

endmodule
```

Blocking and Non-Blocking

```
module ordering (
output reg e,
input a, clock);

reg b , c , d;

   always @ (posedge clock)
         begin
                  b = a;
                  c = b;
                  d = c;
                  e = d;
         end

endmodule
```

```
module ordering (
output reg e,
input a, clock);

reg b , c , d;

   always @ (posedge clock)
         begin
                  b <= a;
                  c <= b;
                  d <= c;
                  e <= d;
         end

endmodule
```

Blocking and Non-Blocking

```
always @ (posedge clock)
        begin
                c = b;
                b = a;
                d = c;
                e = d;
        end
```

```
always @ (posedge clock)
        begin
                c <= b;
                b <= a;
                d <= c;
                e <= d;
        end
```

```
always @ (*)
        begin
                b = a;
                c = b;
                d = c;
                e = d;
        end
```

```
always @ (*)
        begin
                b <= a;
                c <= b;
                d <= c;
                e <= d;
        end
```

Synthesis results

```
module results (
output reg d,
input a,b,c);

always@(*)
        if (a && b)    d = c;
  else if (a)          d = ~c;
  else           d = 1'bx;

endmodule
```

Bad Style

```
module circuit (
output reg [5:0] Qout,
input [1:0] data, clock, reset);

reg [1:0] inter;
reg [2:0] val;
reg [2:0] ue;
reg [5:0] temp;

always@(posedge clock)
   if (reset)
      begin
         inter  <= 0; val      = 0;
         ue     <= 0; temp    = 0;
      end
   else
      begin
         inter <= data;
         val     = data + inter;
         ue    <= val;
         temp   = ue + Qout;
         Qout <= temp;
      end

endmodule
```

Resource Sharing

With proper coding styles designs can be built with minimum hardware

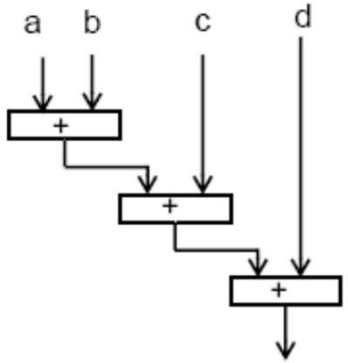
```
module waste (
output reg SUM,
input control, data, in, signal);

always@(*)
  if(control) SUM = data + in;
  else SUM = data + signal;

endmodule
```

```
module share (
output reg SUM,
input control, data, in, signal);

reg t;

always@(*)
   if(control) t = in;
   else t = signal;

assign SUM = data + t;

endmodule
```

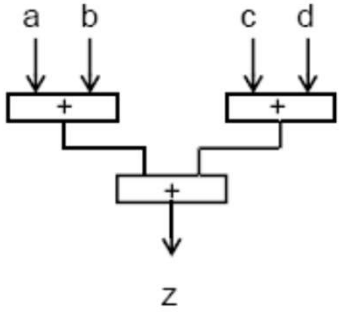
Restructuring

Timing performance of designs can be improved by proper structuring of logic

```
module adders (
output [3:0] z ,
input [1:0] a,b,c,d);

assign z = a + b + c + d;

endmodule
```

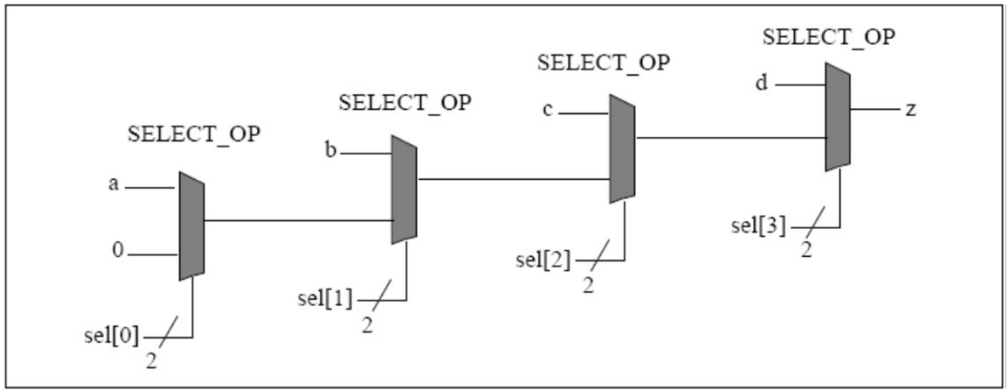



```
module adders (
output [3:0] z ,
input [1:0] a,b,c,d);

assign z = (a + b) + (c + d);

endmodule
```



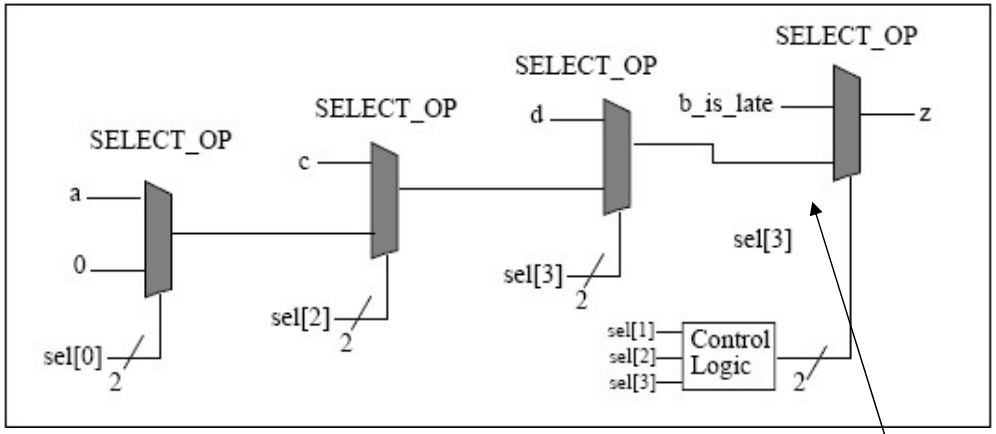
Restructuring

```
module mult_if(
output reg z,
input a, b, c, d,
input [3:0] sel );

always @(*)
        begin
                        z = 0;
                if (sel[0]) z = a;
                if (sel[1]) z = b;
                if (sel[2]) z = c;
                if (sel[3]) z = d;
        end
endmodule
```



Restructuring

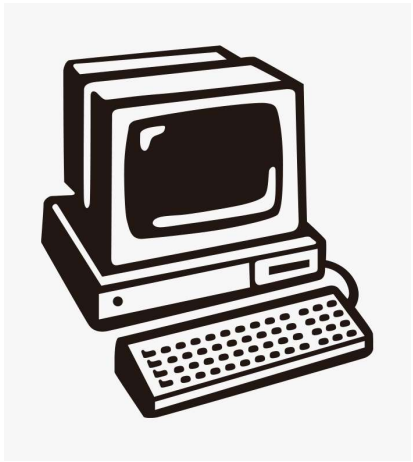
```
module mult_if_improved(
output reg z,
input a, b_is_late, c, d,
input [3:0] sel) ;

always @(*)
begin
                        z = 0;
        if (sel[0]) z = a;
        if (sel[2]) z = c;
        if (sel[3]) z = d;
        if (sel[1] & ~(sel[2]|sel[3])) z = b_is_late;
end
endmodule
```

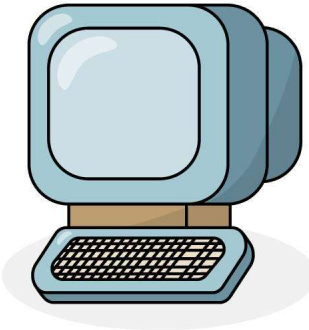


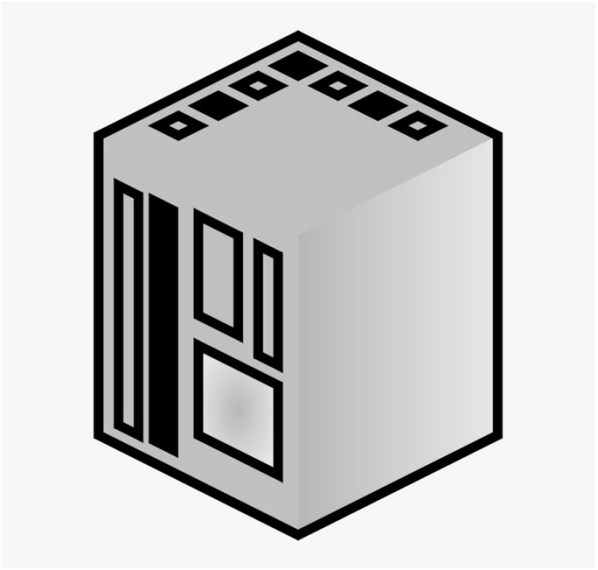
Push late signals towards the output

Coding & Design Guidelines

- Every sequential system should have a reset
- Active low signals should be appended with '_n'
- Variables should not have long names
- While modeling FSMs parameters should be used for state assignments
- Outputs should always be register to have better control
- Each file should have only one module
- File name and module name should be same and meaningful
- The same variable should not be updated from two blocks
- Ensure clock blocks are separated from logic







Projects

Sravani	Sathvika	Swati	Sumanth
Kadambari	Nithyashree	Harita	Vishwa
Rajesh	Rohan	Yashwanth	Amogh
Akhil	Melrin	Hemanth	Ajay
Praveen	Umesh	Vivian	Naresh
Samba Siva	Saiteja	Sudeep	Toushif
			Thanush
CL : Darshan	CL : Darshan	CL : Priyanka	CL : Priyanka