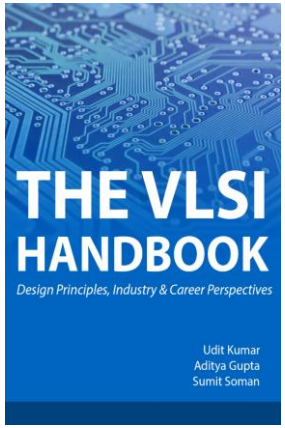
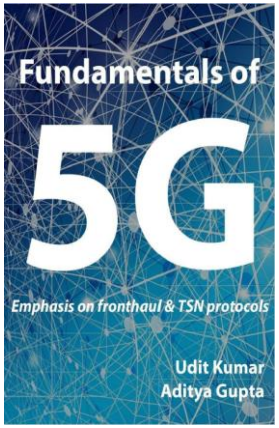
# HDL Design using Verilog

Udit Kumar, PhD, IIT Delhi.

15+ years experience, Author

https://www.linkedin.com/in/udit-kumar-phd-iit-delhi




Learn VLSI

Website: https://www.sites.google.com/view/learnvlsi

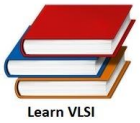
LinkedIn: https://www.linkedin.com/company/learnvlsi

Note: Views expressed here are personal views and not endorsed by present or past employer.

# Disclaimer

- **The intention of this presentation is information sharing. So consider this material as information purpose only.**
- We explicitly disclaim any liability for mistakes and omissions in the material presented.
- We have done our best to ensure the correctness of the material and have no obligation or duty to any person or organization for any loss or damages stemming from the contents.
- We make no claim, promises, or guarantees regarding the correctness, completeness, patent infringement, or sufficiency of the same.
- Take prior approval for Commercial usage of this information.

# Outline



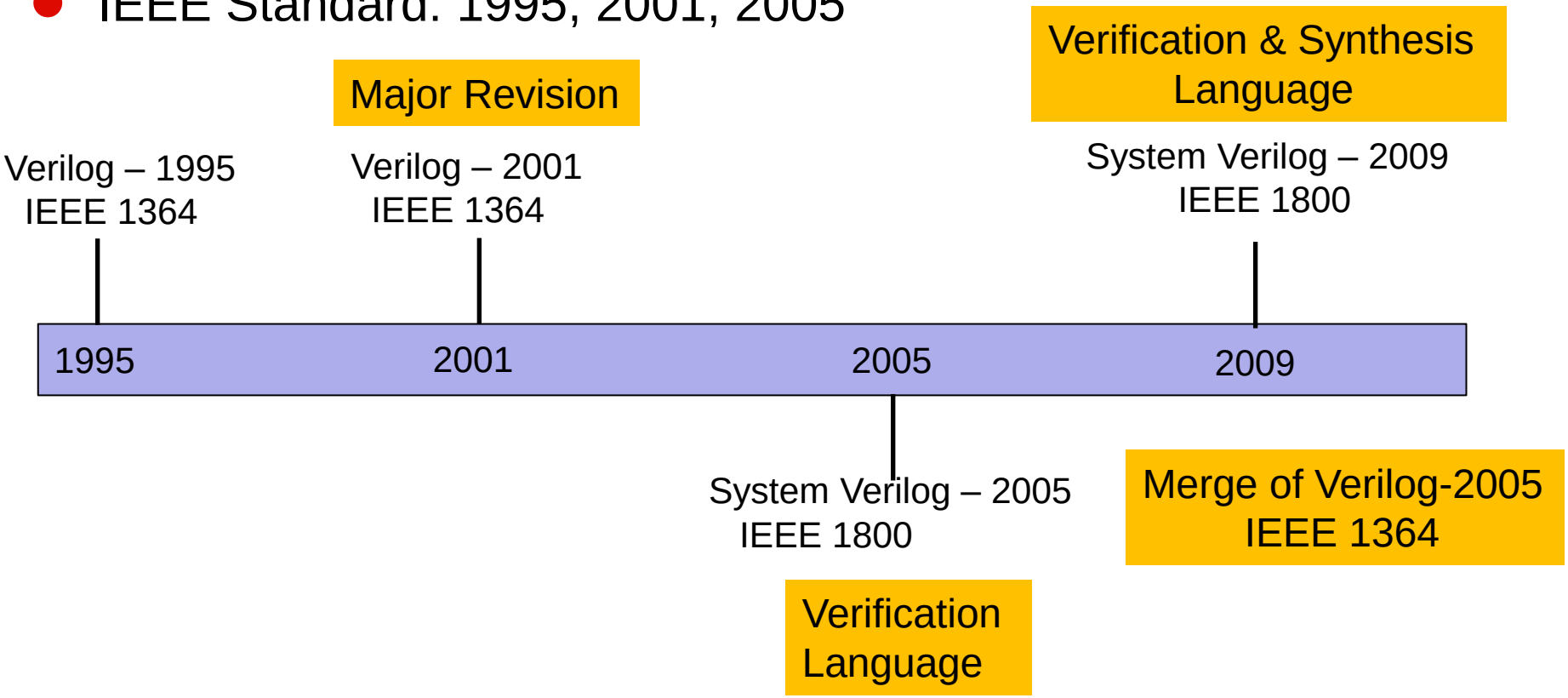

- Need for HDL Language?
- History
- Where to start: VHDL, Verilog, System Verilog?
- First Step for HDL
- Verilog fundamentals
  - Get familiarize with keywords, syntax, operators, features etc.
  - Testbench
  - Blocking, non blocking, Operators, Flow controls etc.
  - Timescales, inter n intra assignment delay
- Delta Delay
- Steps for RTL Design
- Simulations tools
- Learning resources

# Need for HDL Language

- HDL offers a way to model piece of hardware using a text based programming language.
- Synthesis tools convert HDL into equivalent logic gates.
- HDL languages
  - Verilog
  - System Verilog
  - VHDL
- Difference from Software
  - Need hardware mindset
  - Work in parallel not sequential like software

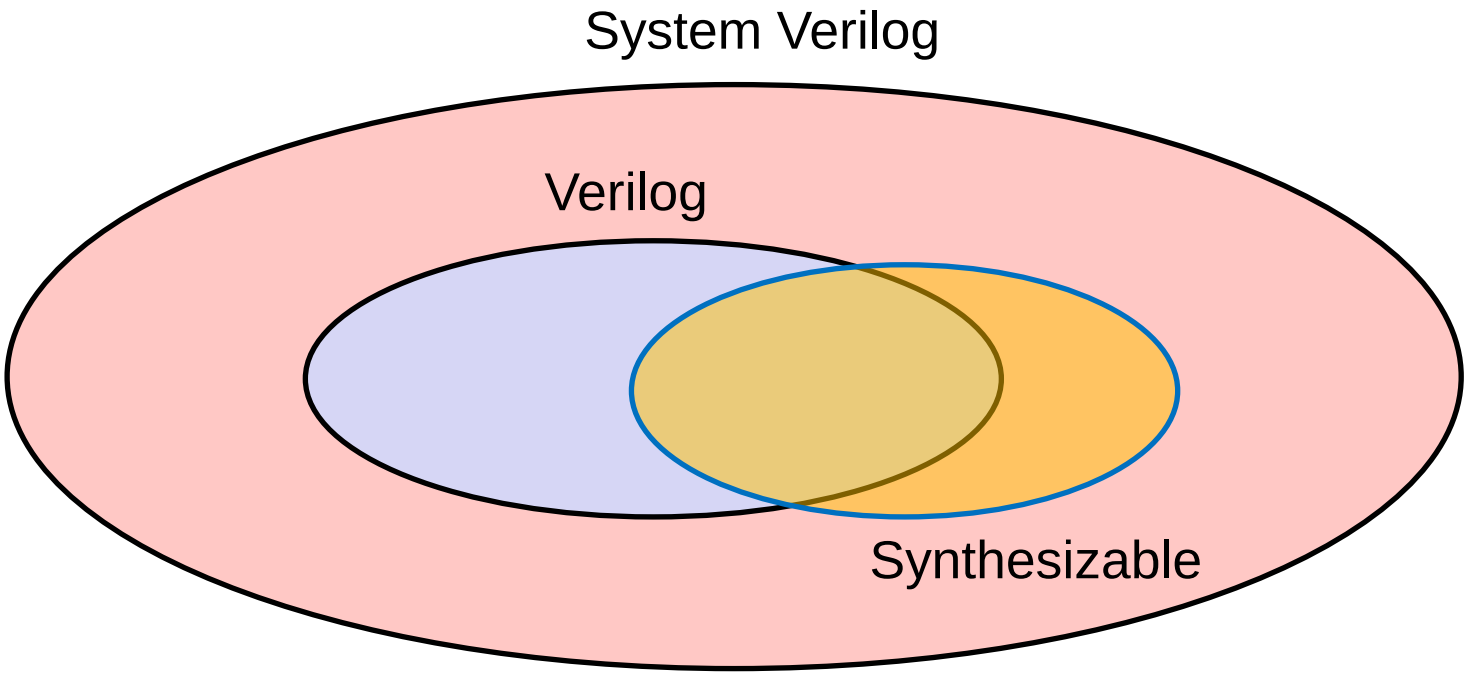
# History

- Developed by Gateway Design Automation (Proprietary Language): 1984
- Verilog made an open Standard: 1990
- IEEE Standard: 1995, 2001, 2005



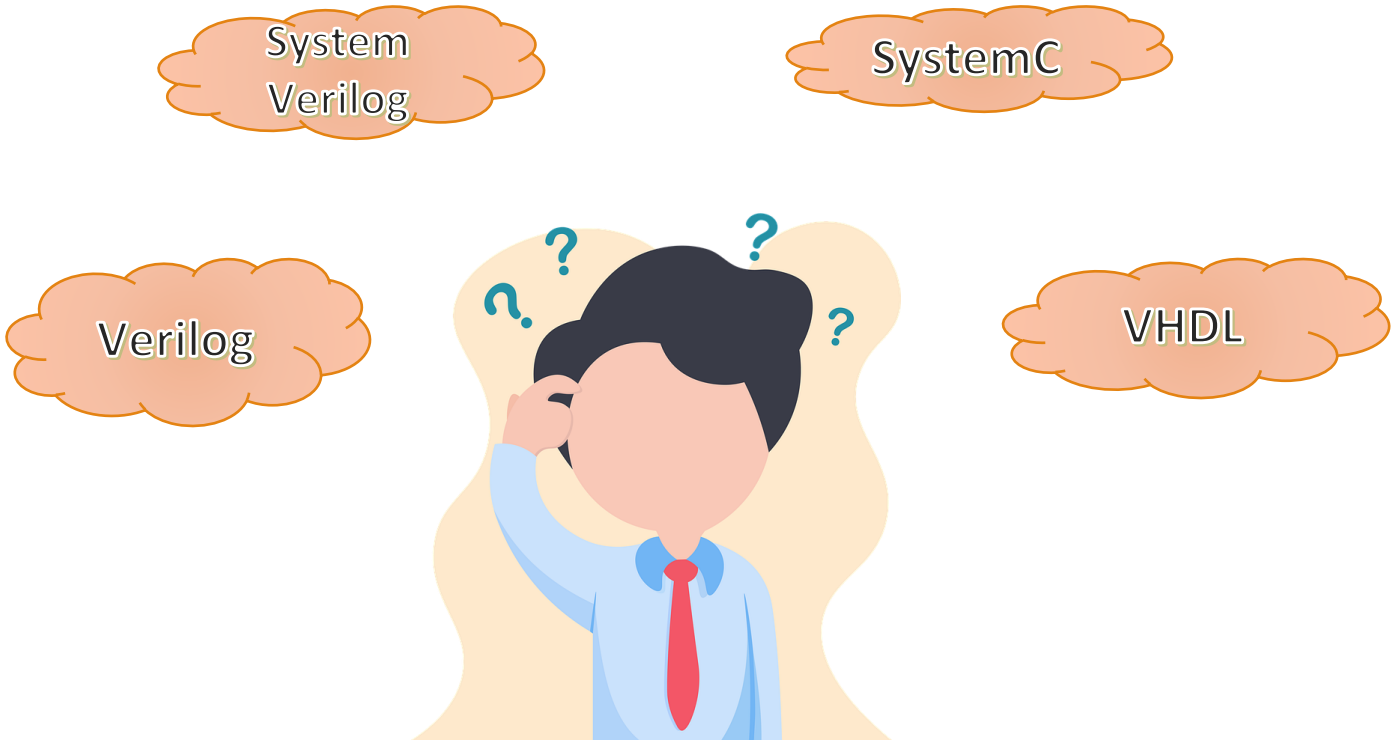
# Verilog and System Verilog

- System Verilog is superset of Verilog.

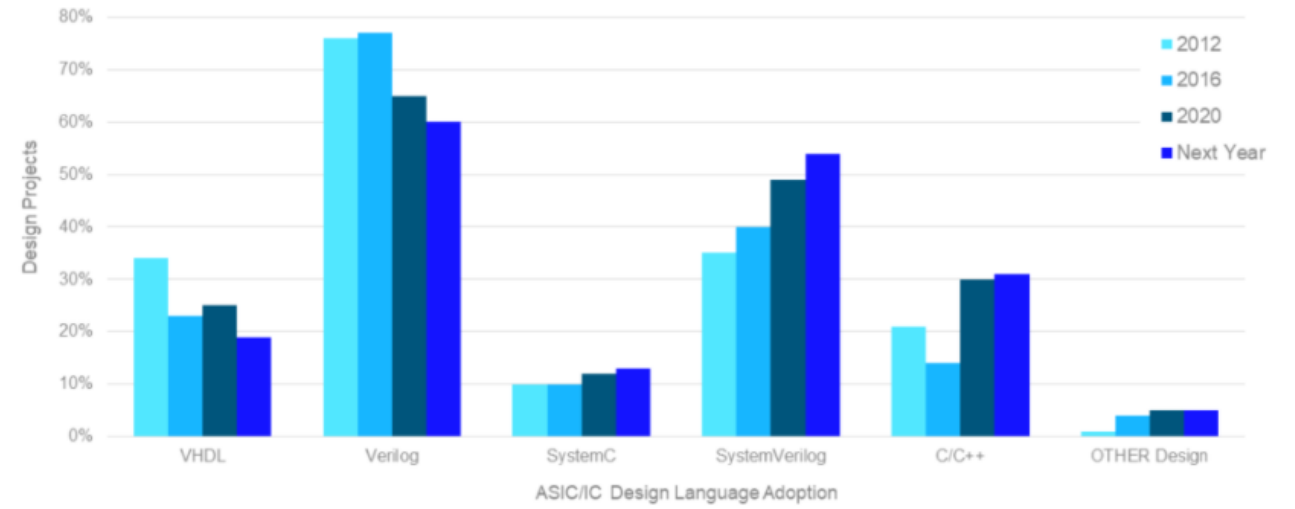



- Both Verilog and System Verilog contains Synthesizable and non synthesizable constructs.

# Confusion: Where to start?



# ASIC/IC Design Language



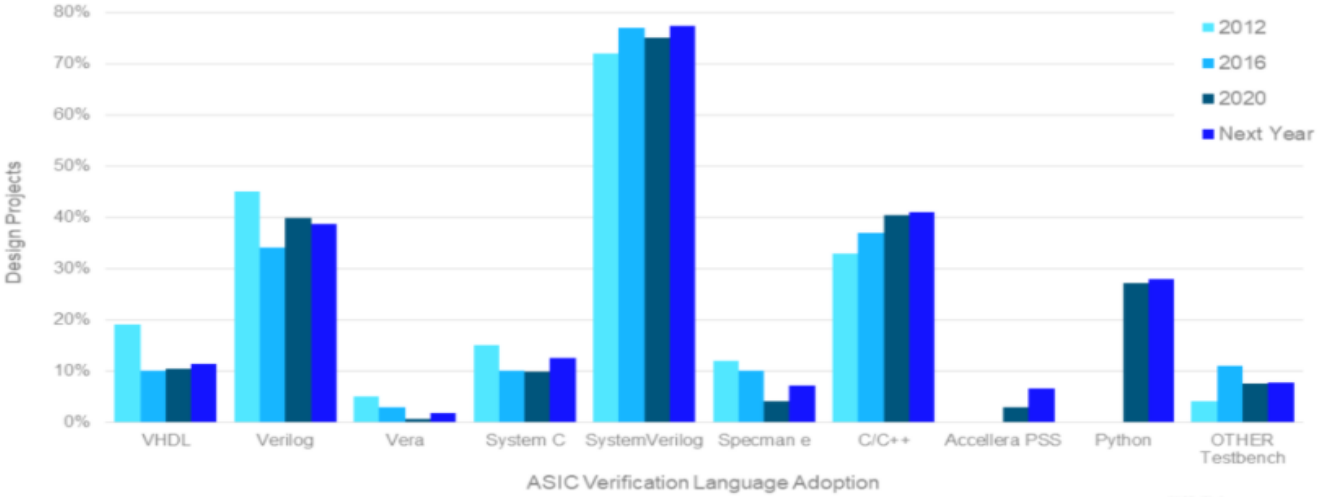

Source: Wilson Research Group and Mentor, A Siemens Business, 2020 Functional Verification Study

** Multiple answers possible




SIEMENS

Design




Source: Wilson Research Group and Mentor, A Siemens Business, 2020 Functional Verification Study

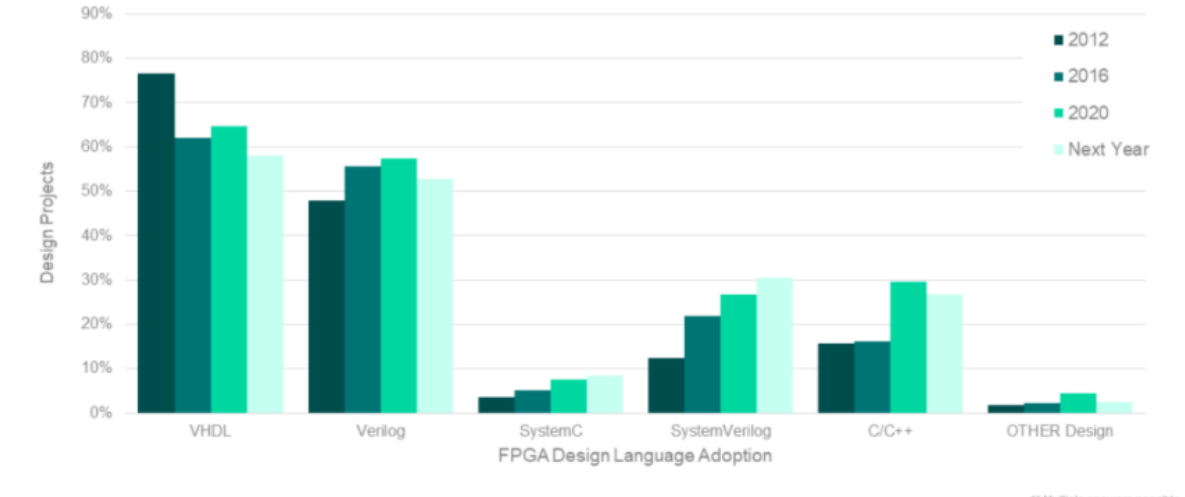
** Multiple answers possible




SIEMENS

Verification

# FPGA Design and Verification



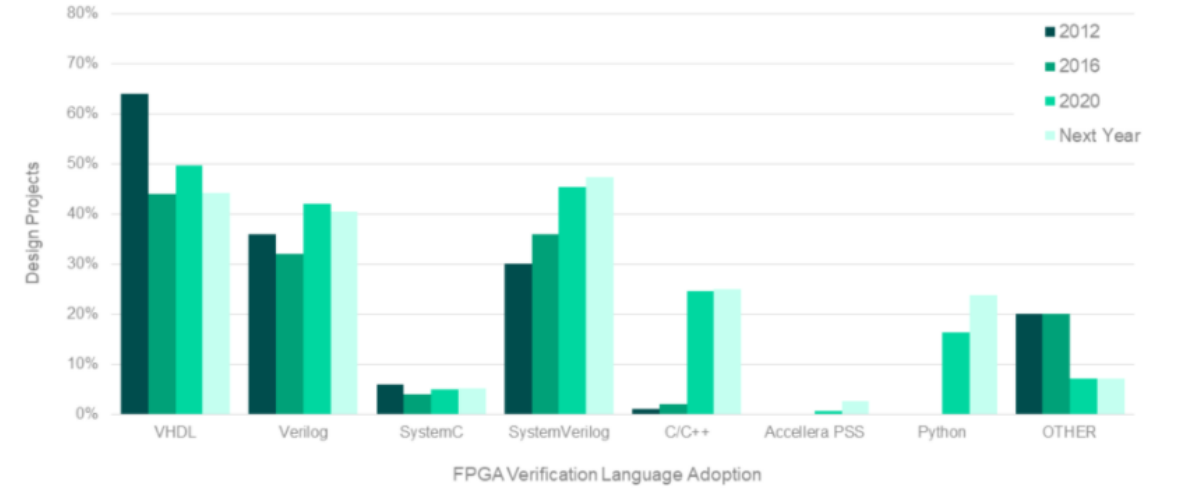

Source: Wilson Research Group and Mentor, A Siemens Business, 2020 Functional Verification Study

** Multiple answers possible




SIEMENS

Design




Source: Wilson Research Group and Mentor, A Siemens Business, 2020 Functional Verification Study

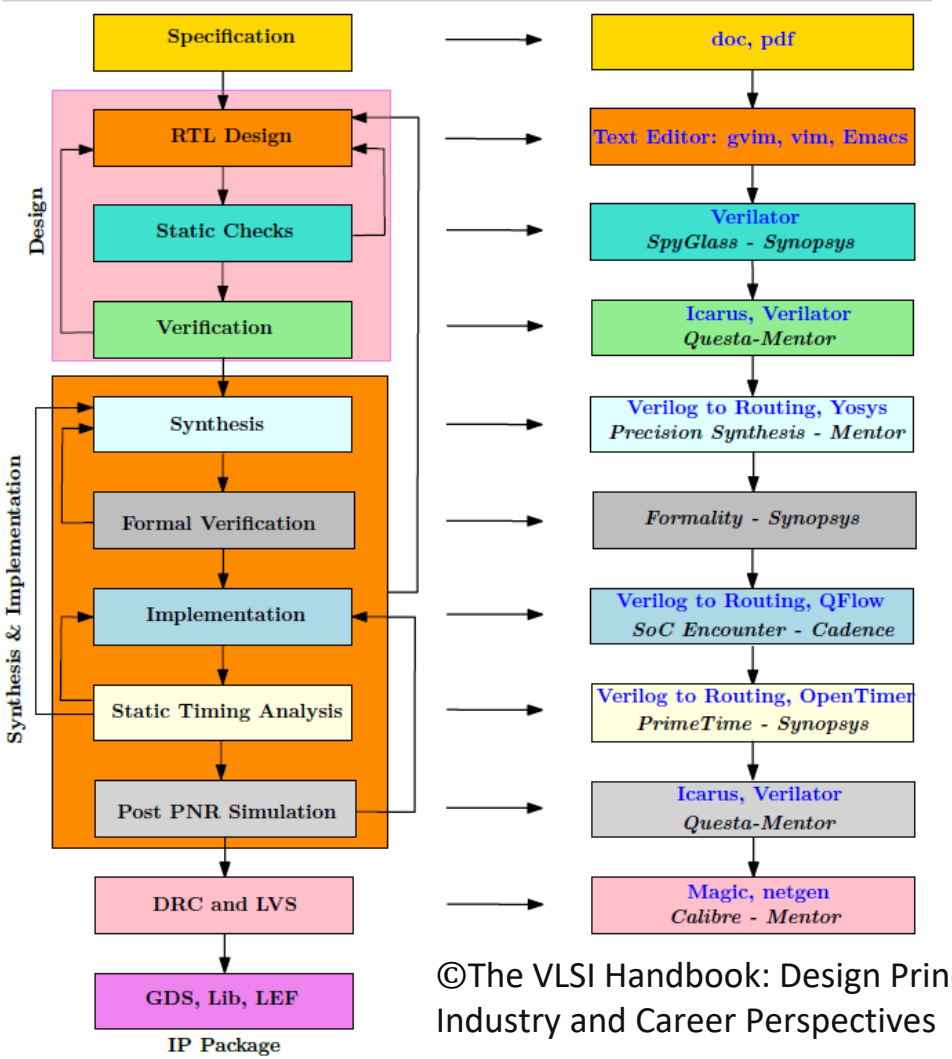
** Multiple answers possible




SIEMENS

Verification

# Hard Digital IP Design Flow

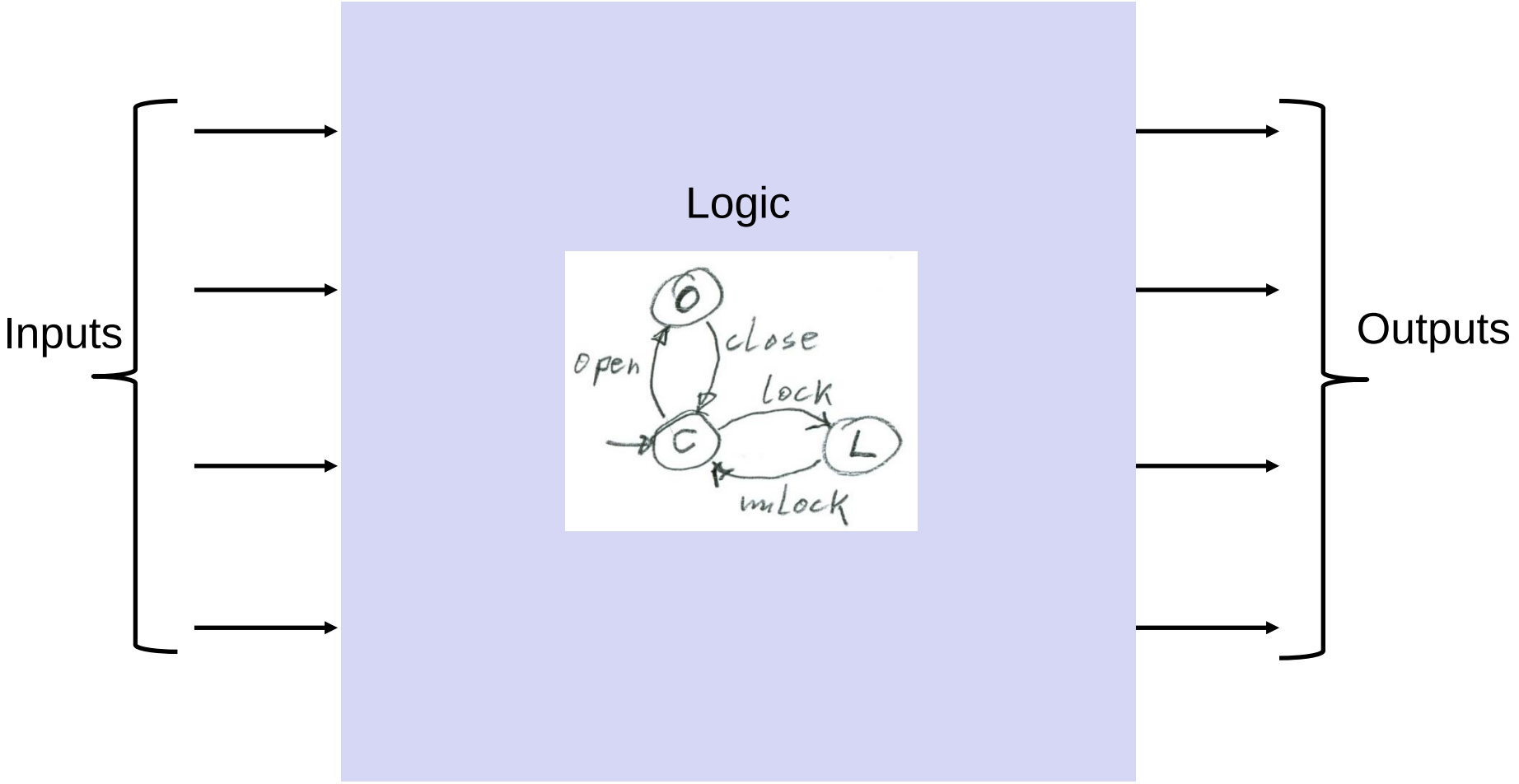



©The VLSI Handbook: Design Principles, Industry and Career Perspectives

Tools written in blue color are open source tools

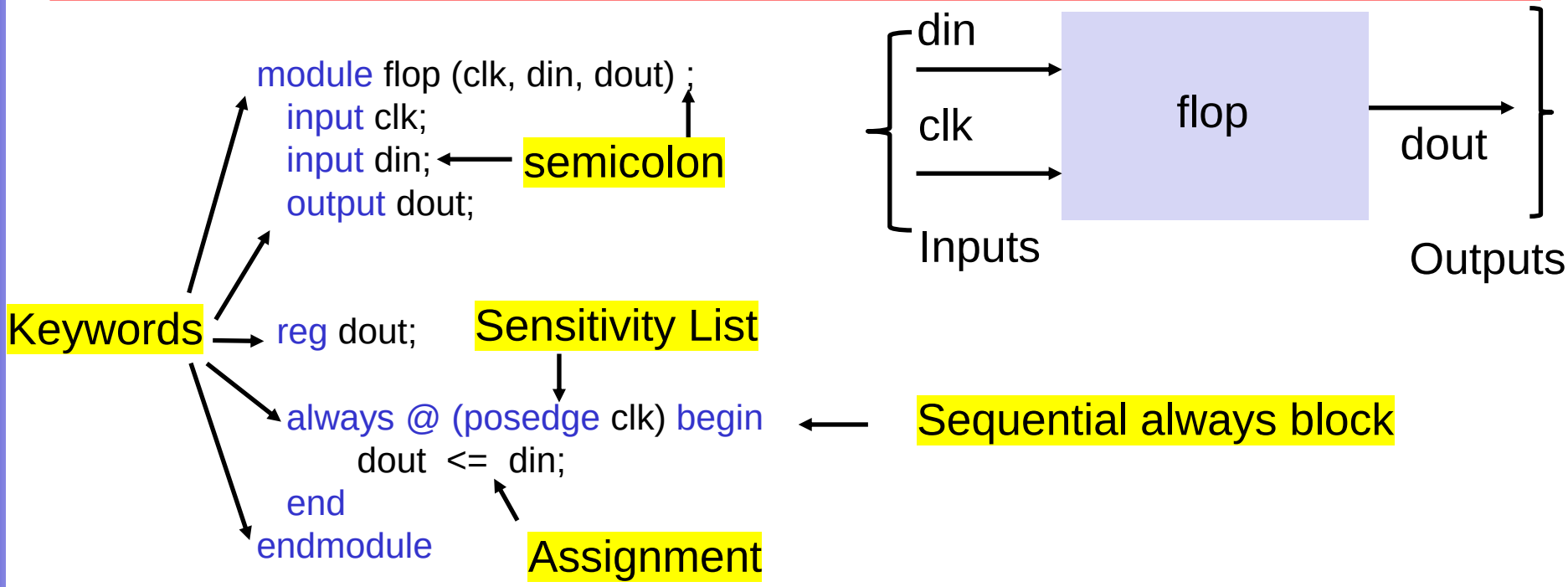
Reference: Book "The VLSI Handbook: Design Principles, Industry and Career Perspectives", Udit Kumar, Aditya Gupta, Sumit Soman

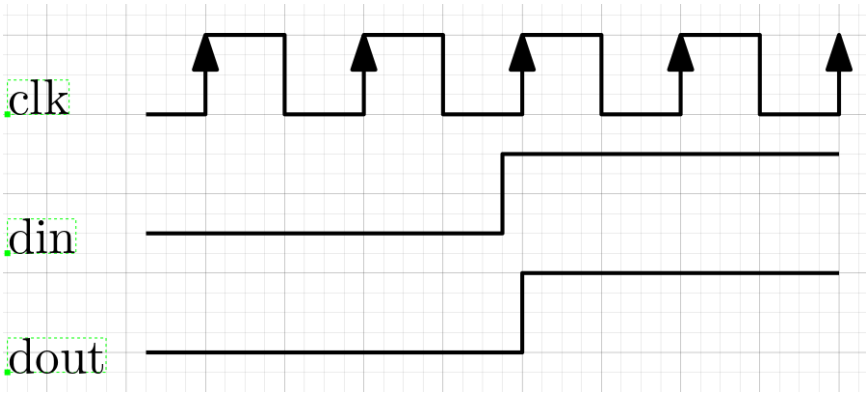
# First Step for HDL: Hardware Mindset




**Thinking in terms of digital design is key.**

# An Example


module flop (clk, din, dout) ;
input clk;
input din;
output dout;
semicolon
Keywords
reg dout;
Sensitivity List
always @ (posedge clk) begin
dout <= din;
end
endmodule
Assignment
Sequential always block
din
clk
Inputs
flop
dout
Outputs


clk
din
dout

# Synchronous and Asynchronous Reset

```
module flop (clk, rst_n, din, dout) ;
  input clk;
  input rst_n;
  input din;
  output dout;

  reg dout;

  always @ (posedge clk or negedge rst_n) begin
      if (!rst_n) begin
          dout <= 1'b0;
      end else begin
          dout  <=  din;
      end
  end
endmodule
```

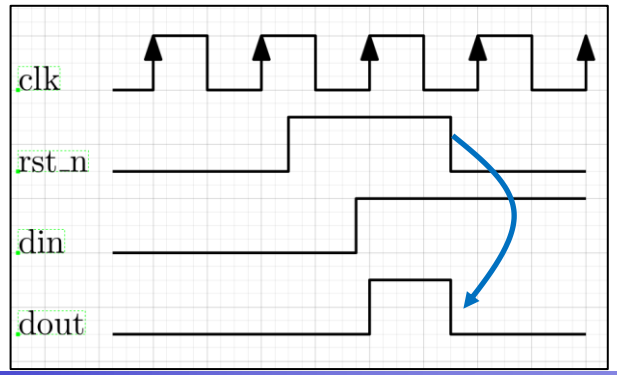
Difference in the sensitivity list

Reset Check

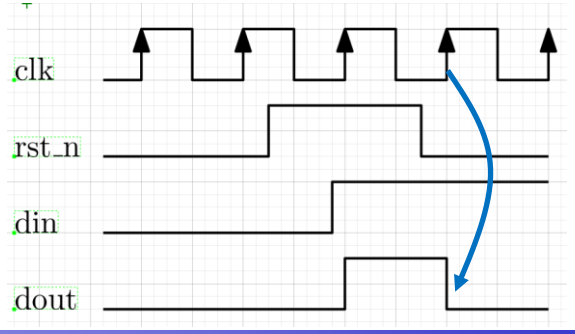
Conditional statement

```
module flop (clk, rst_n, din, dout) ;
  input clk;
  input rst_n;
  input din;
  output dout;

  reg dout;

  always @ (posedge clk) begin
      if (!rst_n) begin
          dout <= 1'b0;
      end else begin
          dout  <=  din;
      end
  end
endmodule
```

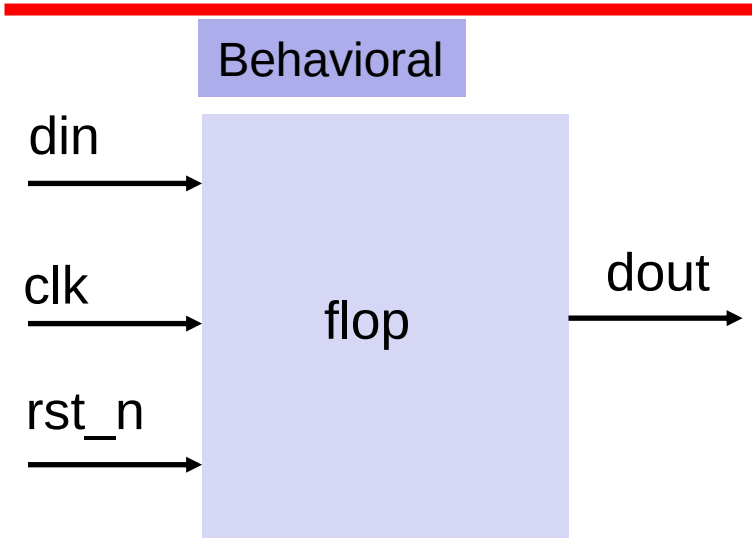
Async Reset

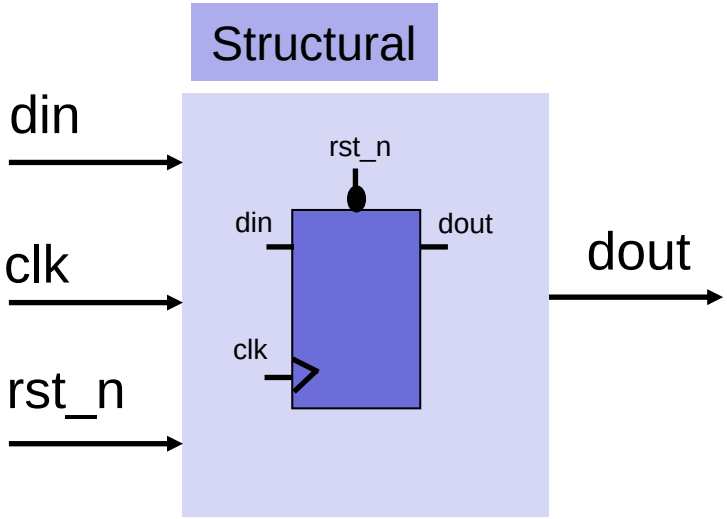



Sync Reset



# Behavioral Vs structural




```
module flop (clk, rst_n, din, dout) ;
  input clk;
  input rst_n;
  input din;
  output dout;

 reg dout;

  always @ (posedge clk or negedge rst_n) begin
      if (!rst_n) begin
         dout <= 1'b0;
      end else begin
         dout  <=  din;
      end
  end
endmodule
```

```
module flop (clk, rst_n, din, dout) ;
  input clk;
  input rst_n;
  input din;
  output dout;

  DFF dff_inst (
     .din(din),
     .clk(clk),
     .rst_n(rst_n),
     .dout(dout));
endmodule
```

Instance name

Port connection by name

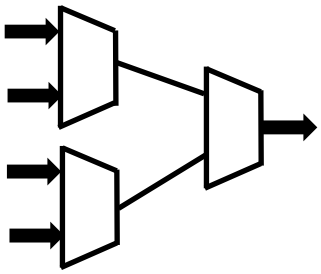
```
DFF dff_inst(clk, rst_n, din, dout);
```

Port connection by ordered list

# Behavioral Vs Structural

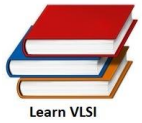
- Behavioral
  - Program describes input/output behavior of circuit
  - Many structural implementations could have same behavior.
  - Easier to write and understand


- Structural
  - Explicit structure of the circuit
  - Useful to build bigger designs.



**Behavioral modelling is mostly used to describe functionality. Structural modelling is used to connect various blocks.**

# Continuous and Procedural Statement




**Continuous Assignment**

```
module flop (D0, D1, DOUT) ;
  input D0, D1;
  output DOUT;

assign DOUT = D0 & D1;

endmodule
```

Operators

**Procedural Assignment**

```
module flop (D0, D1, DOUT) ;
  input D0, D1;
  output DOUT;

always @ (D0 or D1) begin
  DOUT = D0 & D1;
end ;

endmodule
```

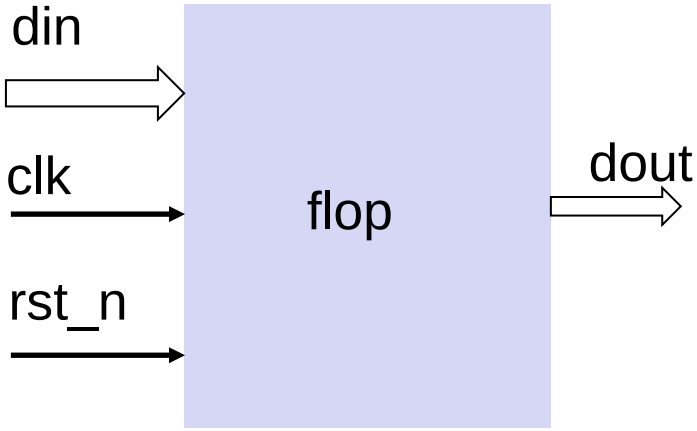
Blocking assignment

- Continuous Statement (assign)
  - Always evaluated,
  - All statement evaluates in parallel
- Procedural Statement (always, initial)
  - Execute in the defined order
  - Two types
    - always: Executed continuously based upon sensitivity list
    - initial: Execute only once

# Parameter

Default value

```
module flop #(parameter WIDTH=8) (clk, rst_n, din, dout) ;
   input clk, rst_n;
   input [WIDTH -1 : 0] din;
   output [WIDTH -1 : 0] dout;


  reg [WIDTH -1 : 0] dout;

   always @ (posedge clk or negedge rst_n) begin
         if (!rst_n) begin
            dout <= {WIDTH{1'b0}};
         end else begin
            dout  <=  din;
         end
   end
endmodule
```



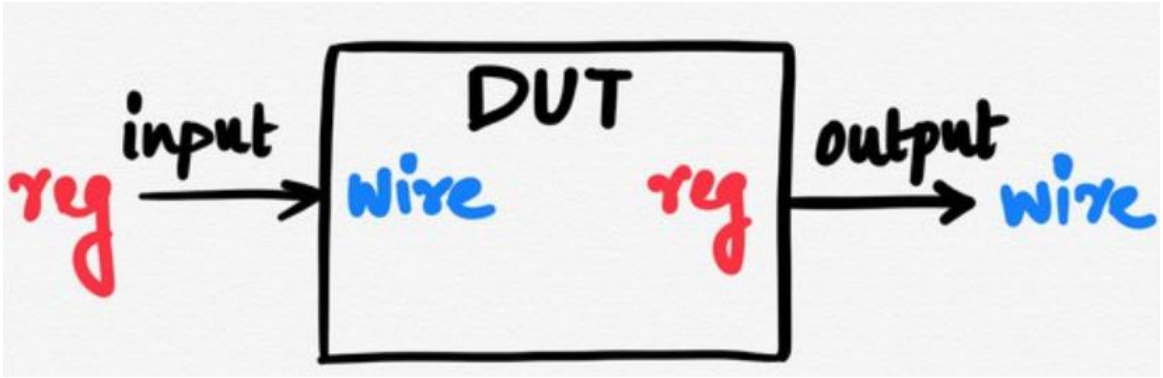
# Let’s summarize

```
module <module_name> #(<parameters>) (<port list>) ;

    <Declarations>
    <Continuous statements>
    <Behavioral code>
    <Instantiations>
    <Task and functions>

endmodule
```

# Testbench

## Testbench

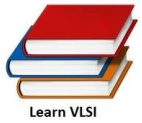
```
module testbench () ;
   reg clk = 1'b0;
   reg rst_n;
   reg din;
   wire dout;

initial begin
  forever
     #10 clk = ! clk;
end

initial begin
   din = 1'b0;
   rst_n = 1'b0;

   #25 rst_n = 1'b1;
  # 7 din     = 1'b1;
end

flop F1 (clk, rst_n, din, dout);

endmodule
```




## DUT

```
module flop (clk, rst_n, din, dout) ;
  input clk;
  input rst_n;
  input din;
  output dout;

 reg dout;

  always @ (posedge clk or negedge rst_n) begin
       if (!rst_n) begin
          dout <= 1'b0;
       end else begin
          dout  <=  din;
       end
  end
endmodule
```

# Can we use always Vs initial forever for clock generation?




- Yes, both approach are fine
- Approach 1

```
reg clk = 1'b0;
always
      #10 clk = ! clk;
 end
```

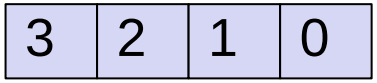
- Approach 2

```
reg clk = 1'b0;
initial begin
   forever
      #10 clk = ! clk;
 end
```

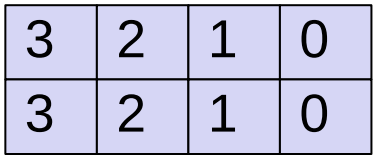
# Scaler, Vectors and Arrays

```
wire busA; // Single bit scaler net
```

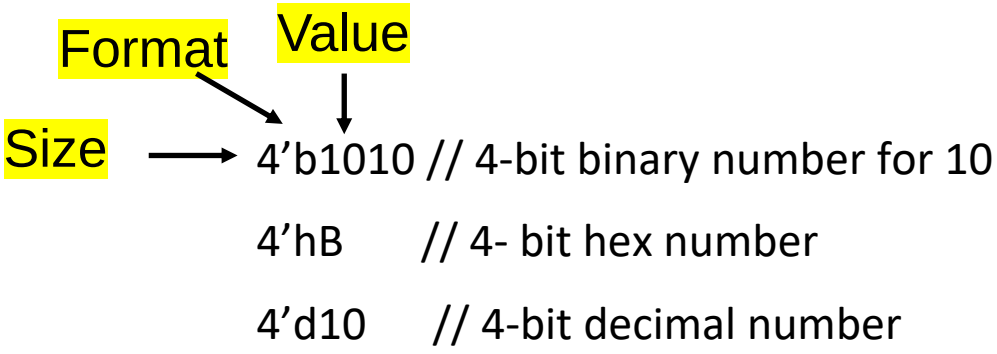
```
wire [3:0] busA // 4-bit vector net
```




```
wire [3:0] busA [2]
```



# Numbers representation



# Operators

- Arithmetic Operators
  - +, - , ..*
- Bitwise Operators
  - &, |, ^ ..: Work on each bits (A=3'b101 B= 3'b100, Q = A &B = 3'b100)
- Reduction Operators
  - &, |, ^, ~.. : Reduces a vector to a single bit value (&A = 'b0)
- Relational Operators
  - >, <, >= ...
- Logical Operators
  - !, &&, || ..
- Shift Operators
  - <<, >>, <<<, >>> (A=8'b010, B = A << 2 = 8'b01000)
- Equality Operators
  - ==, ===, !=,

Additional Details: https://www.chipverify.com/verilog/verilog-operators

# Find out issue in the code?

```
module flop #(parameter WIDTH=8) (clk, rst_n, din, dout) ;
  input clk, rst_n;
  input [WIDTH -1 : 0] din;
  output dout;


 reg dout;

  always @ (posedge clk or negedge rst_n) begin
       if (!rst_n) begin
          dout <= {WIDTH{1'b0}};
       end else begin
          dout  <=  din;
       end
  end
endmodule
```

```
module flop (clk, rst_n, din, dout) ;
  parameter WIDTH=8;
  input clk, rst_n;
  input [WIDTH -1 : 0] din;
  output dout;


 reg dout;

  always @ (posedge clk or negedge rst_n) begin
       if (!rst_n) begin
          dout <= {WIDTH{1'b0}};
       end else begin
          dout  <=  din;
       end
  end
endmodule
```

# Flow controls

- Control statement
  - If, for , while
  - case, casex, casez
  - …

- Initial/forever

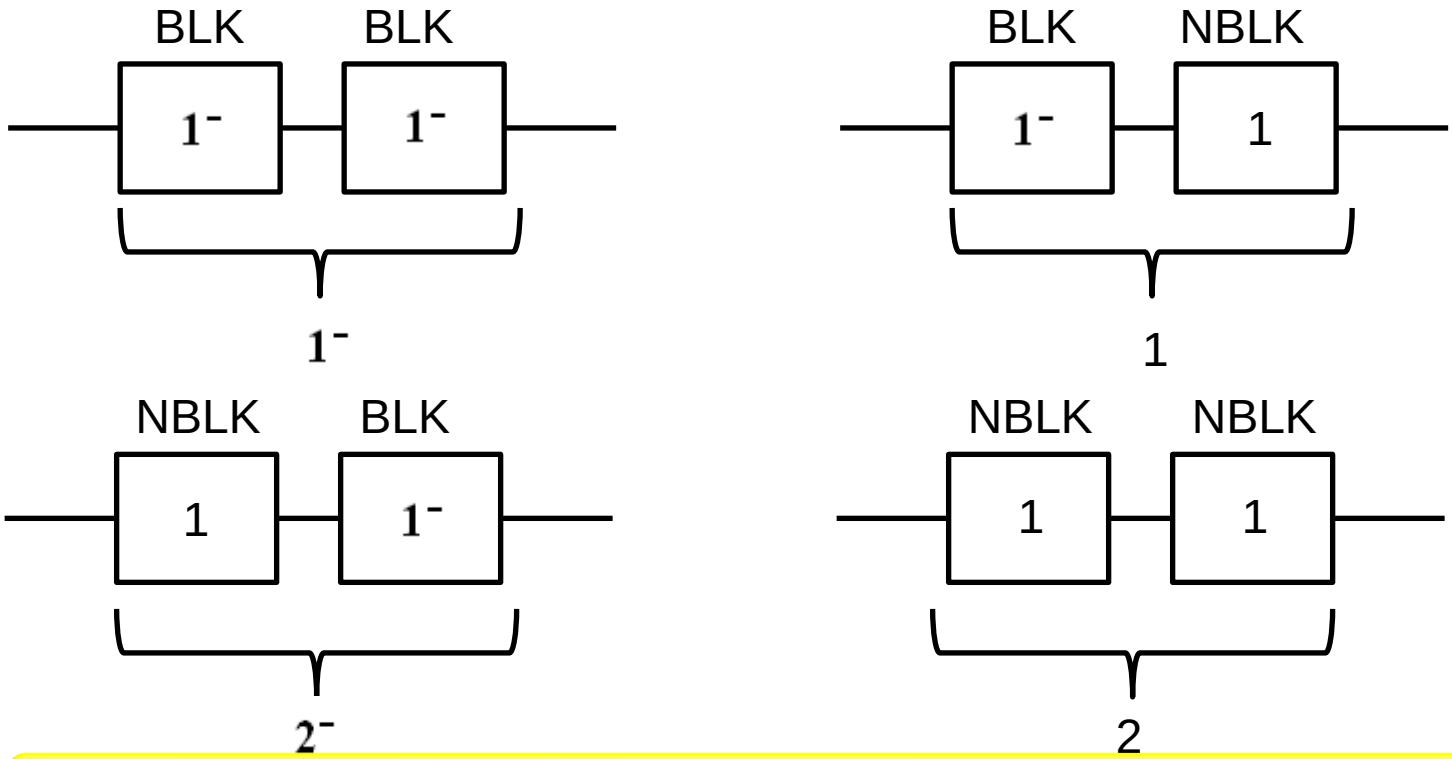
```
module forever_example (clk_out) ;
output clk_out;

 reg clk_out = 1'b0;

initial begin
    #10 clk_out = ! Clk_out;
end

endmodule
```

```
module forever_example (clk_out) ;
output clk_out;

 reg clk_out = 1'b0;

initial begin
  forever
    #10 clk_out = ! Clk_out;
end

endmodule
```

# Blocking vs non-blocking assignments

- Delta delay occurrence in such assignments

$1^-$ Start of delta time




Blocking Blocks delta time

BLK: Blocking assignment, NBLK: Non-Blocking assignment

# An example

```
// Blocking assignment
always @ (A or B)
begin
   S1 =  (A || B);
end


always @ (posedge clk)
begin
   Q =  D;
end

 -- No delta cycle consumed
```

```
// Non blocking assignment

always @ (A or B)
begin
  S2 <= (A || B );
end

always @ (posedge clk)
begin
   Q <=  D;
end

-- 1 delta cycle consumed
```

# Verilog timescale

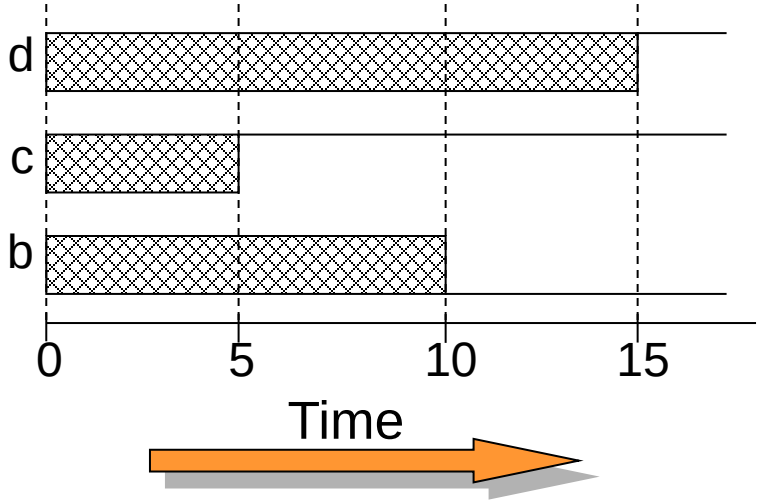
- Verilog uses time delays defined in units.
- Mapping time units to “real” time is done using `timescale
- `timescale <time_unit> / <time_precision>
  - Time_unit: this is the time unit used to defined delays
  - Time_precision: Minimum time incremented by simulator
  - Example

```
`timescale 1ns/1ps

initial begin
      q = 1'b0;
  #5 q = 1'b1;  // ← 5 ns delay will be observed
end
```

# Fork/Join

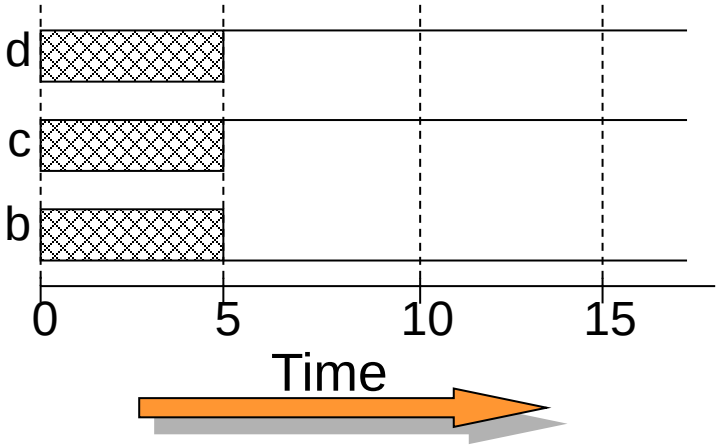
```
initial begin
    #5  c = 1;
    #5  b = 0;
    #5 d  = c;
end

endmodule
```

Execute sequentially




```
initial begin
  fork
    #5  c = 1;
    #5  b = 0;
    #5 d  = c;
  join
end

endmodule
```

Execute Parallelly




Waveform credit: Thanasis Oikonomou

# Intra and Inter assignment delay

- Intra assignment delay

<LHS> = #<delay>  <RHS>

```
module flop (clk, rst_n, din, dout) ;
   input clk;
   input din;
   output dout;


   reg dout;

   always @ (posedge clk) begin
         dout  <=#1  din;
   end
endmodule
```

Considered as transport delay

- Inter assignment delay

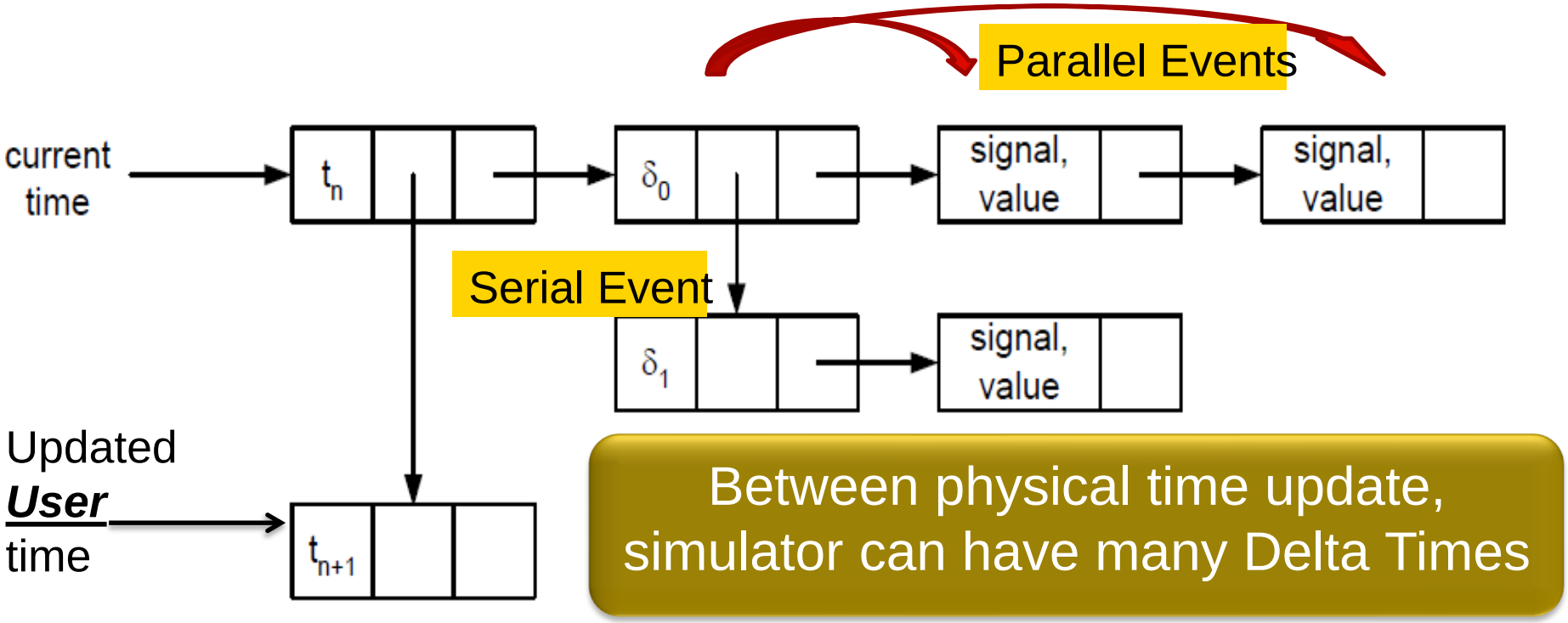
#<delay> <LHS> = <RHS>

```
module forever_example (clk_out) ;
output clk_out;

 reg clk_out = 1'b0;

initial begin
    #10 clk_out = ! Clk_out;
end

endmodule
```

Wait for elapse of the time and than evaluate the expression

# What is Delta Delay?

- A delta time, used inside simulator database, is infinitely small but useful…
- Each event consumes one *delta* time-step.



# Verilog has event queue

```
always @ (A or B)
begin
   S1 =  (A || B);
   S2 <= A && B;
end
```

**Active event**

- Blocking assignments
- Continuous assignment
- Evaluate RHS of non-blocking assignment

**Inactive events**

- #0 blocking assignments

**Non-blocking**

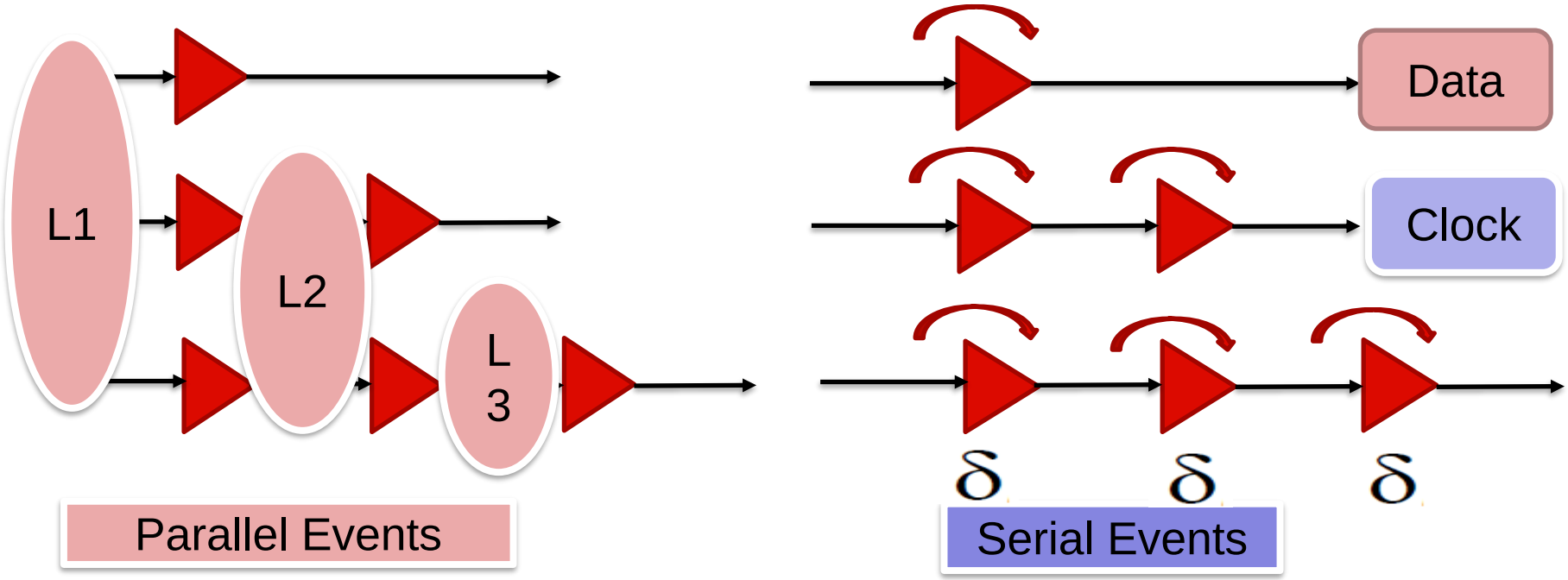
- Update LHS of non-blocking assignments

**Monitor events**

- $monitor & $strobe

**Multiple events are grouped together and execute in defined order.**

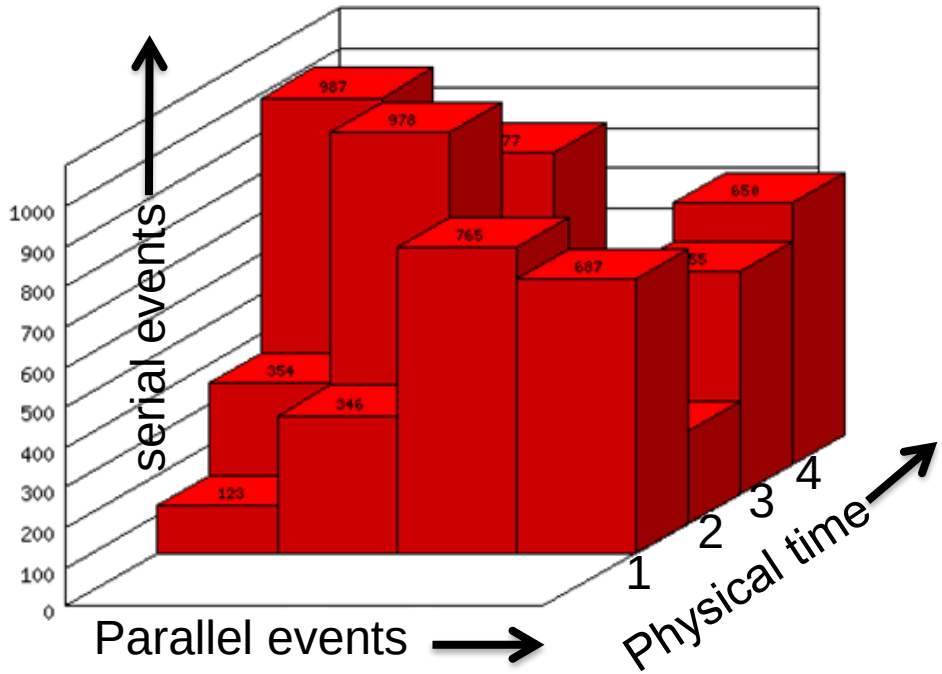
# Concept of Parallel & Serial events




- All events are software objects & consume “Delta” delay.
- Simulators do not understand relationship between data & clock events.

# Discrete Event Simulation

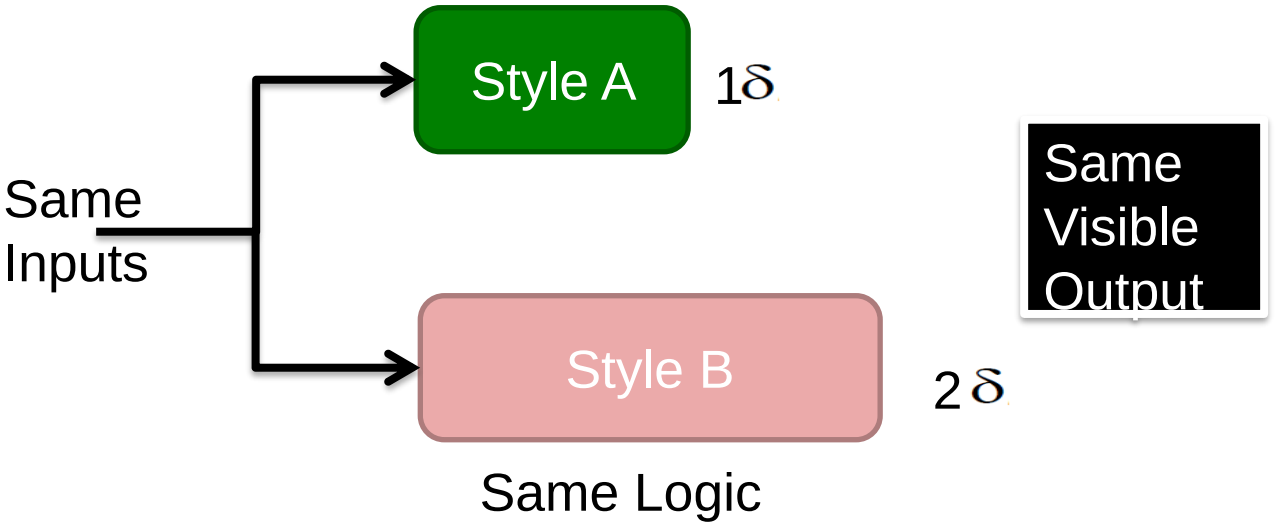
- A method to mimic events in physical systems, using software.




- Language Rules (LRM) specifies how and when events are created and updated.
- Simulator manages millions of events and their order (explained later)
- Understanding this is useful to debug corner cases.

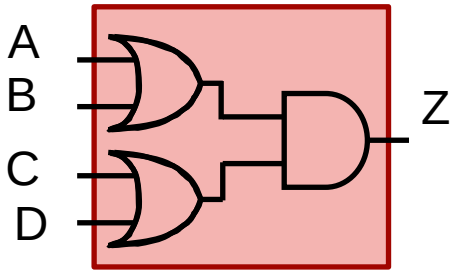
# HDL writing affects events

- HDL writing affects creation and flow of events.
- The result visible to user is the same, ir-respective of HDL style. As simulator do not show simulated result for each delta to the users.

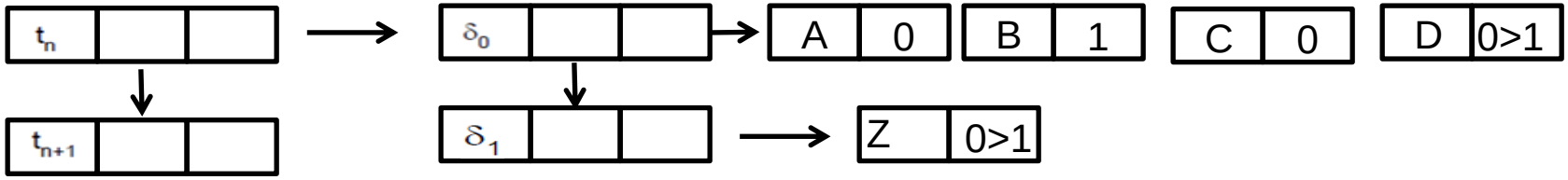


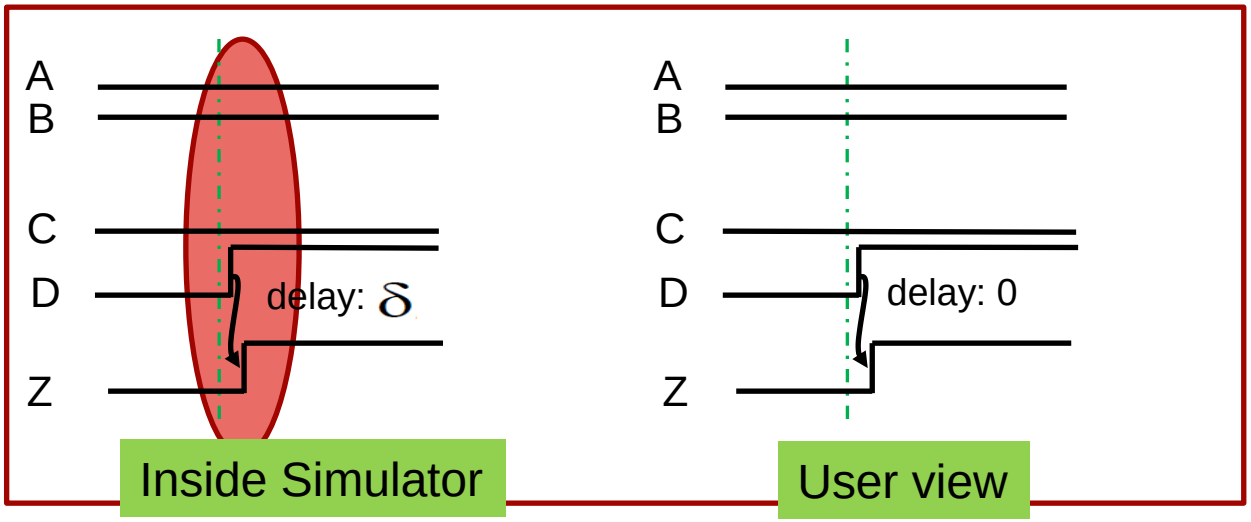
# Style A: Output coming after 1 Delta

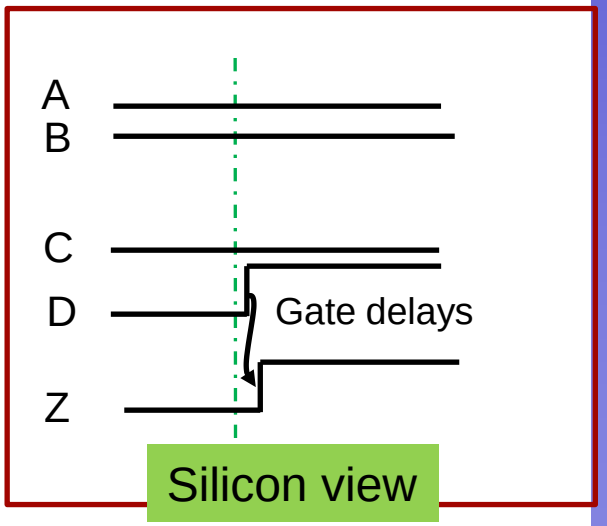
```
Always  (A,B,C,D) begin
   Z <=  (A OR B) AND (C OR D);
end
```




Test case

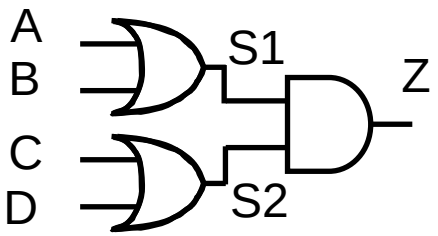


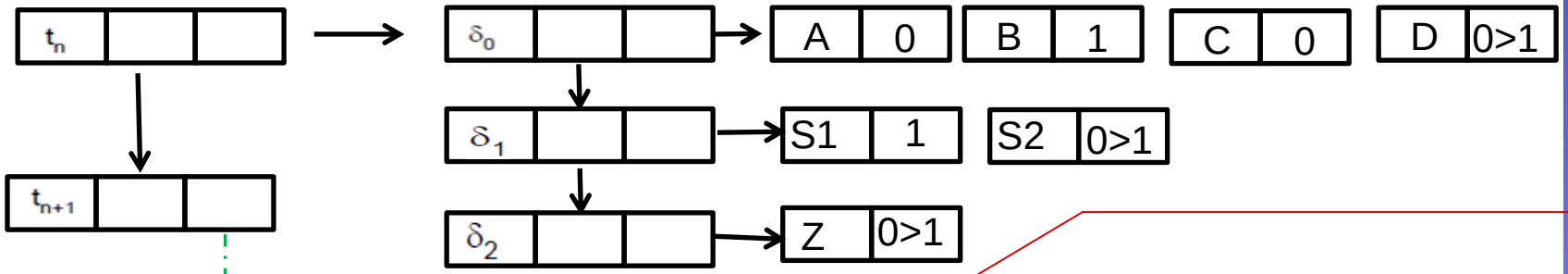



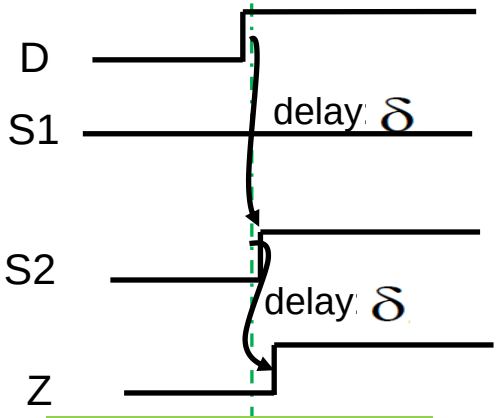
# Style B: Output coming after 2 Delta

```
process (A,B,C,D,S1,S2) begin
   S1 <=  (A OR B);
   S2 <=  (C OR D);
   Z <= S1 AND S2;
end
```

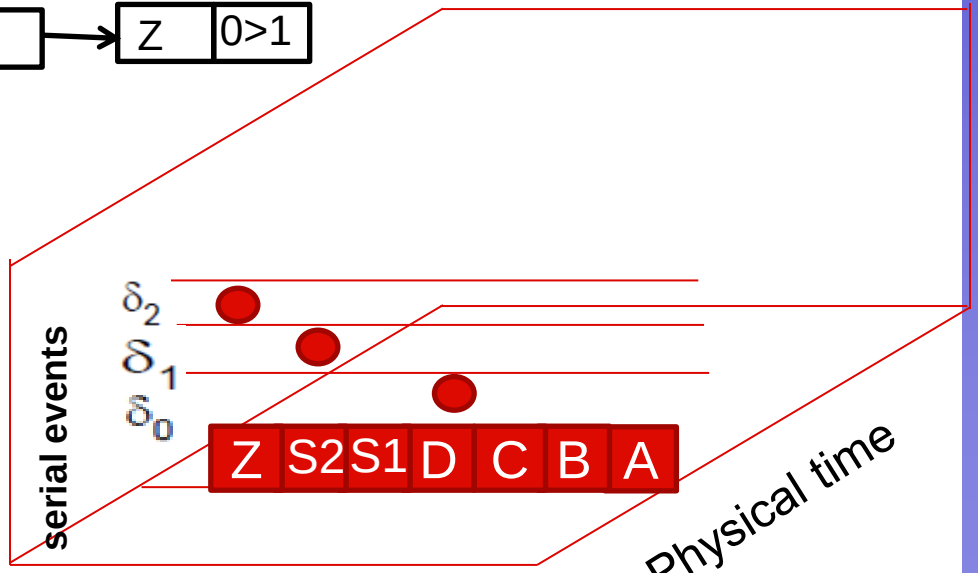



Test Case



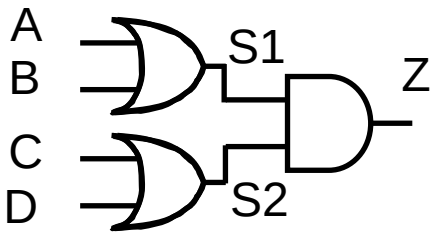

Inside Simulator



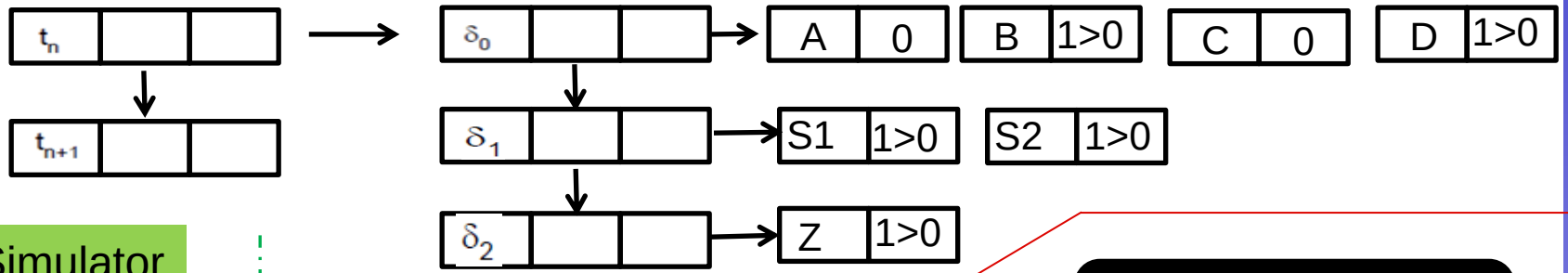

Parallel events

# Style B with Multiple Input Changes

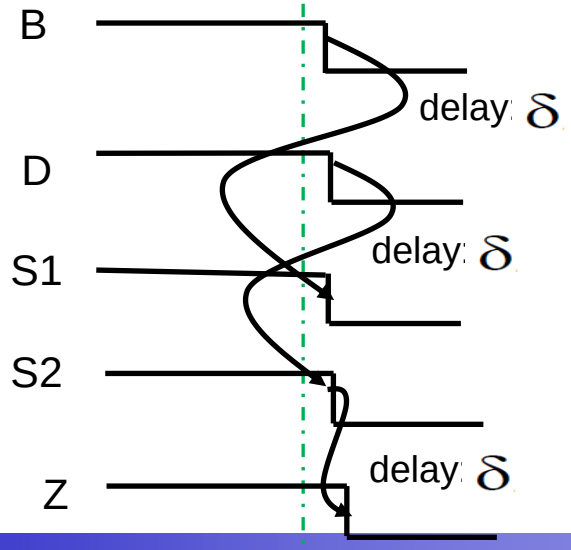
```
process (A,B,C,D,S1,S2) begin
   S1 <=  (A OR B);
   S2 <=  (C OR D);
   Z <= S1 AND S2;
end
```

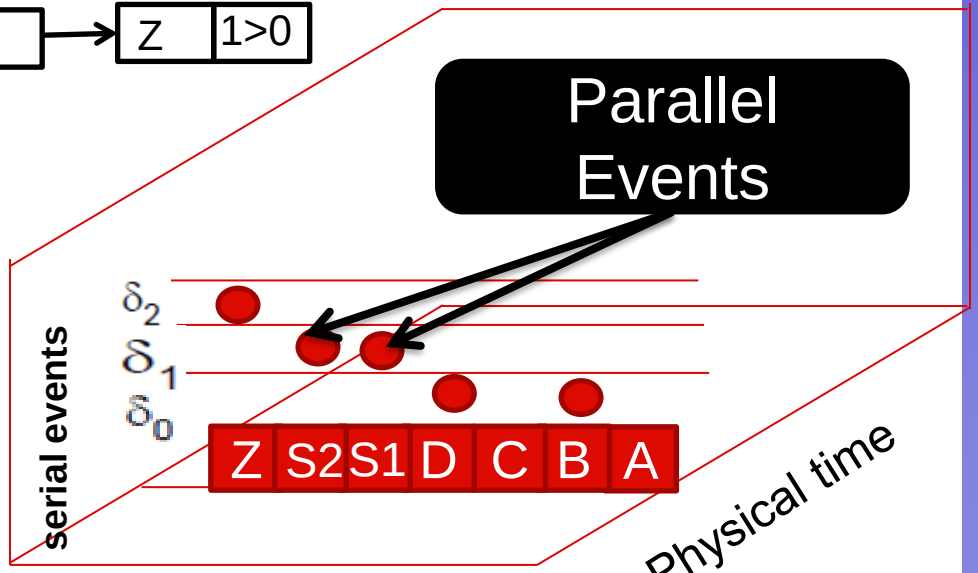



Test Case




Inside Simulator

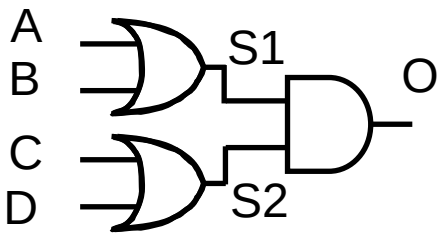


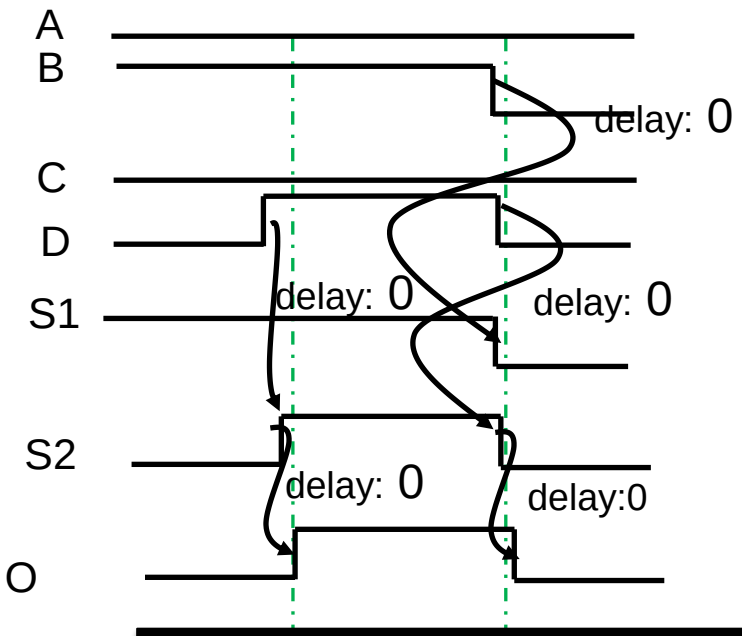

Parallel events

# Style B: RTL & Silicon view

```
process (A,B,C,D) begin
   S1 <=  (A OR B);
   S2 <=  (C OR D);
   O <= S1 AND S2;
end
```

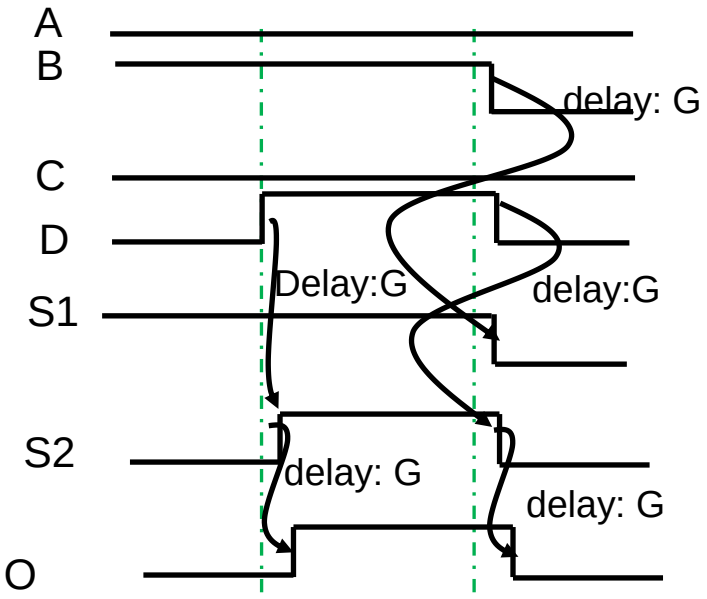



RTL view




No delay from I/P to O/P

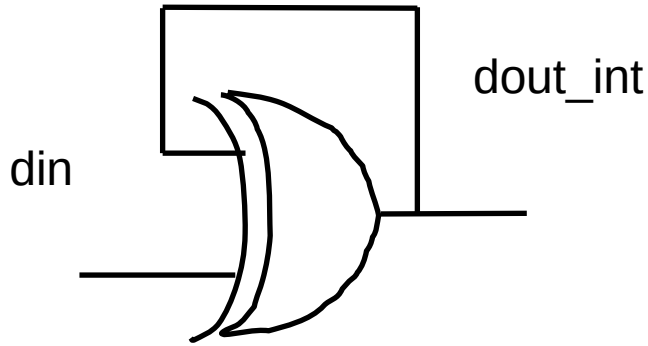
Silicon view




O/P after gate Delay

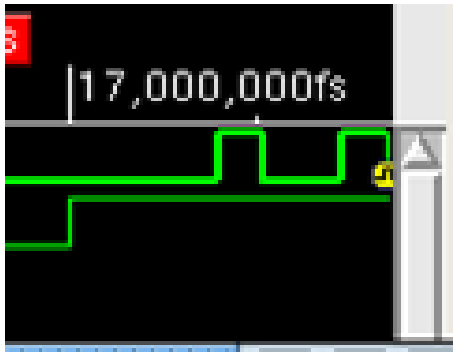
# Delta delay and Physical delay

```
reg dout_int=0;
always @ (din or dout_int) begin
  dout_int <= dout_int ^ din;
end
```




- RTL Simulation will hang

- Gate level simulation will produce train of pulse



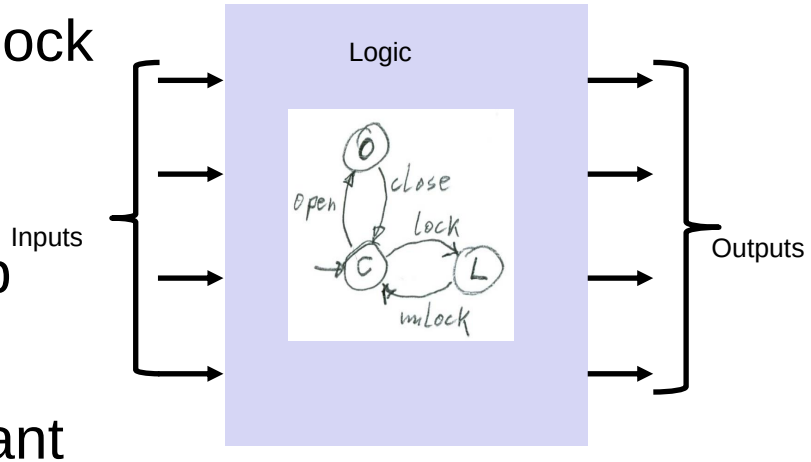

RTL Simulation hangs but gate level simulation produces train of pulses.

# Design examples

- Clock gating
- Clock dividers
- Up counter
- Level to pulse generator
- Pulse to level generator
- Sequence detector
- FSM designs
- ..

# Steps for RTL Design

- Draw a block diagram and mention input and outputs.
- Decide on the clock and reset
- Do you need to break functionality in multiple blocks? If yes, draw sub block level diagrams.
- Does your design require some sequence of operation? If yes, map desired functionality into a FSM.
- Think in terms of what state you want to store on every clock cycle.
  - This will help you to find out required flops, that you are going to put into the sequential always block.



# UP Counter


```
module flop #(parameter WIDTH=4)
        (input clk, rst_n, en, output [WIDTH-1:0] count)

 reg [WIDTH-1:0] counter;

 always @ (posedge clk or negedge rst_n) begin
      if (!rst_n) begin
         count <= 1'b0;
      end else begin
         if (en) begin
            count  <=  count + 1;
         end
      end
 end
endmodule
```

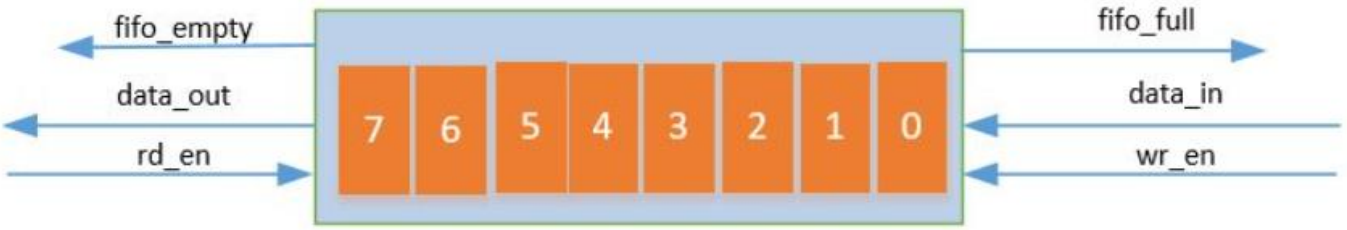
Sequential always block

Combinational logic

Flops

# Synchronous FIFO Design

Assignment: Lets design a Synchronous FIFO with depth 32, and width 8.




Write Control

RAM

Read Control

FIFO full and FIFO Empty

Reference: https://vlsicoding.blogspot.com/2013/11/verilog-code-for-synchronous-fifo.html

# Simulation tools

- Open Source tools
  - Icarus Verilog: www.iverilog.icarus.com
  - Verilator: www.veripool.org/wiki/verilator
- Commercial tools
  - Cadence: ncsim
  - Mentor: Questa
  - Synopsys: vcs
- Web based simulator
  - https://www.edaplayground.com/
- Useful utilities: https://edautils.com/DownloadLinks.html

# Learning Resources

- Book: VERILOG HDL, by Samir Palnitkar
- Verilog FAQ by Shivakumar
  - Various FAQ on Verilog
  - Good collection of interview Questions: A must read before Interviews
- https://www.sites.google.com/view/learnvlsi
- https://www.vlsiguru.com/mentor-verilog-hep-ppt/
- https://www.chipverify.com/verilog/

# Questions and Answers

- Async reset should be used as active low or high?
  - Usage of active low async reset is higher.
  - Async reset is helpful to save power, and better noise immunity, better control to bring whole design into reset state.
- Can we use posedge clock along with posedge of rst_n in sensitivity list?
  - Yes, we can usge.
- Can we use “and” in sensitivity list
  - No, this is not allowed.
- Which one is better, Async or sync reset?
  - Async reset is good when we want to control design reset externally and want reset immediately. Majority of design using Async reset.
  - Sync reset is part of data path and generally internally generated signals.  This is used on need basis.
  - For area optimization in the data path, there are flops with async or sync reset.

# Thank you

Next webinars:

**RTL Design Guidelines (Planned for 26-Feb-2022)**

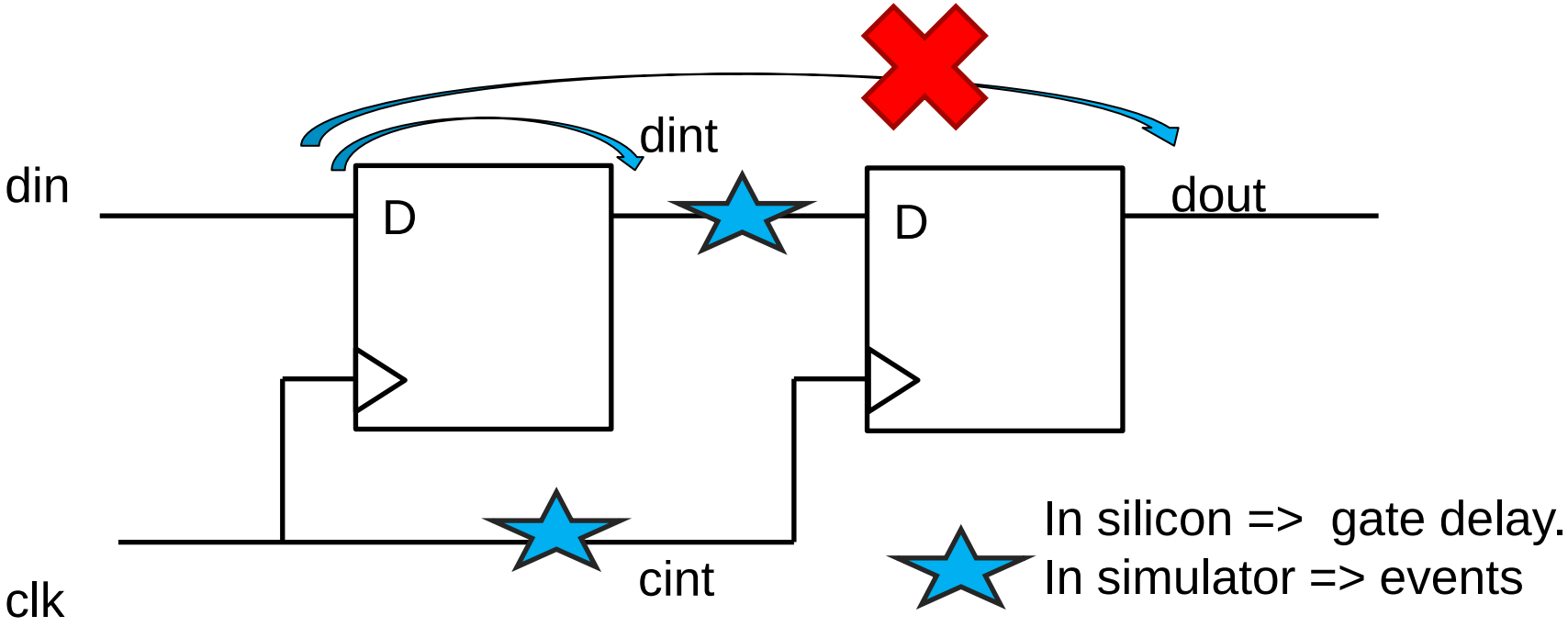
**Low Power RTL Design (Tentative: 26th March 2022)**

**For more updates, follow Learn VLSI LinkedIn Page :**
https://www.linkedin.com/company/learnvlsi

# ADDITIONAL INFORMATION

# What is Shoot-thru?

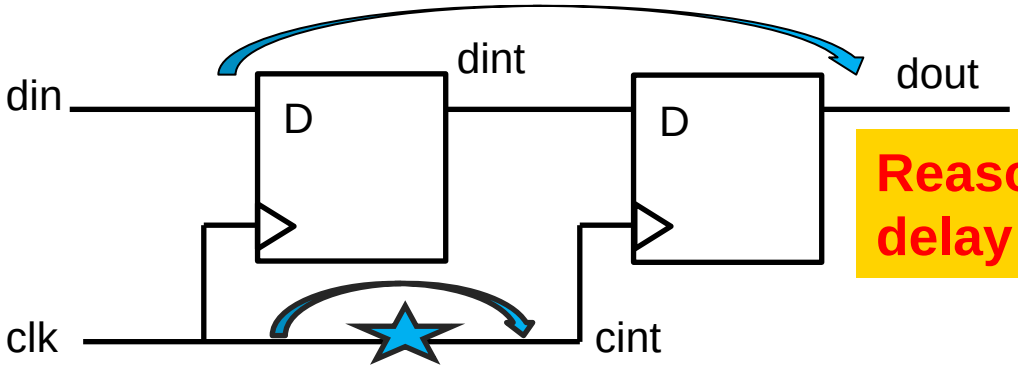
- When any signal jumps over an *extra* register within one clock cycle.
  - E.g. Input “din” crosses 2 register within one clock cycle.



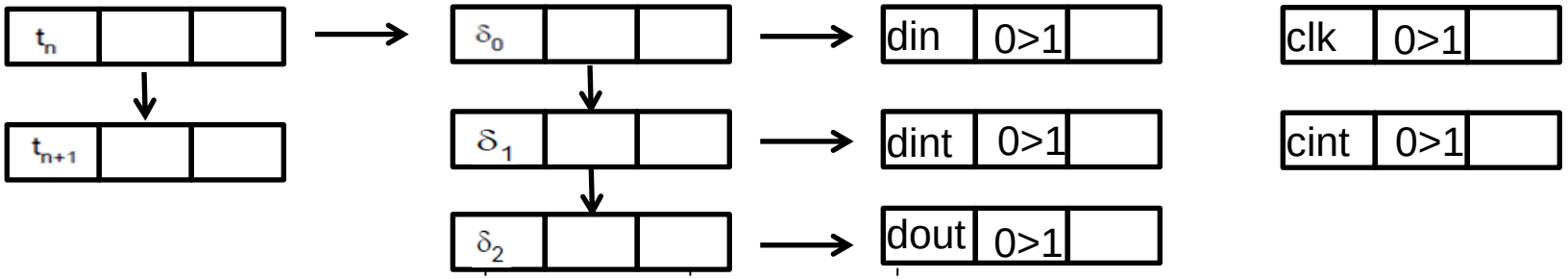

**Shoot-thru:**

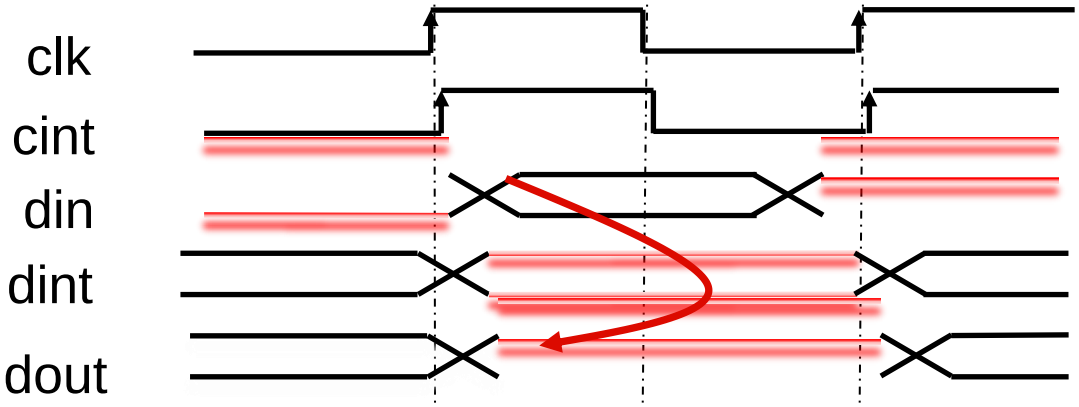
**(events in data path ) <= (events in clock path)**

# How Simulation Shoot-thru occurs?




**Reason : Extra Delta delay in Clock Path**

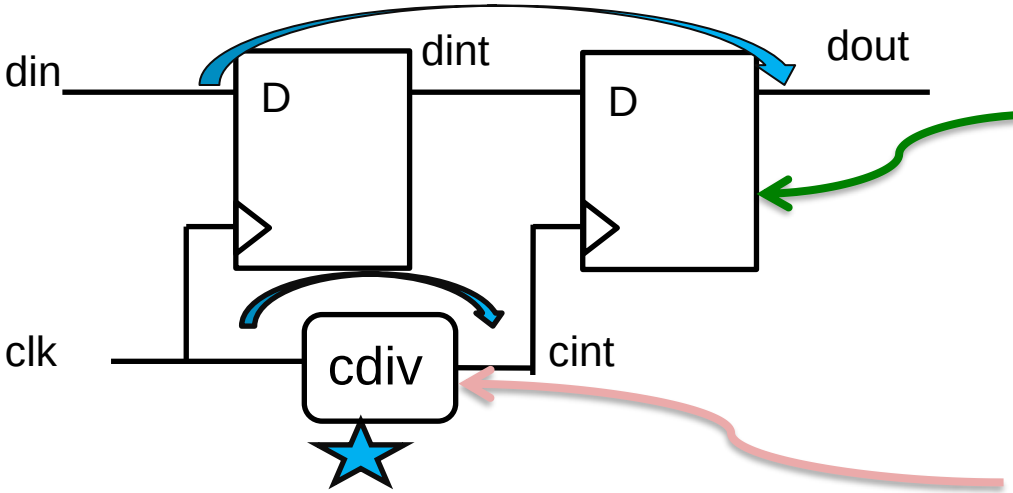



Consequence : dout same as din within single clock

# Will Verilog coding rules help?

- Can blocking assignment in combination blocks and non-blocking in sequential avoid simulation shoot-thru?
  - Not, always….



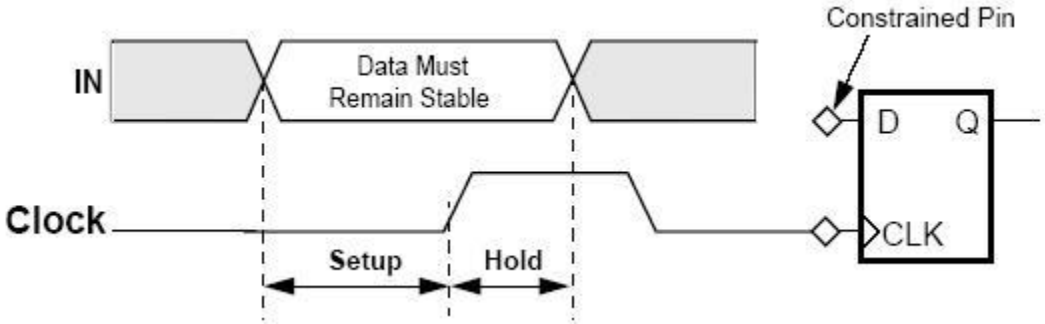

**Extra Delta-delay in Clock Path**

```
// Input data sampling
always @ (posedge clk, negedge rst_n) begin
    if (rst_n==1'b0)
                dint  <= din;
     else
                dint  <= din;
 end
// Capture data on divided clock
always @ (posedge cint, negedge rst_n) begin
    if (rst_n==1'b0)
                dout <= 1'b0;
     else
                dout <= dint;
 end
// Clock generation
always @ (posedge clk, negedge rst_n) begin
    if (rst_n==1'b0)
                cint     <= 1'b0;
   else
                cint    <= !cint;
```

**Non-blocking assignments consume delta delay**

# So, How to detect shoot-thru issues?

- By using an improved simulator?
  - **NO,** as root cause of problem is that RTL Simulation has no notion of timing delay & Hold checks.




- By doing Gate level simulation?
  - **Yes** but it is very costly.
- By using robust RTL coding rules?
  - **Yes** this is easy and efficient (see subsequent slides)

# Rule 1 - Force scheduling of data

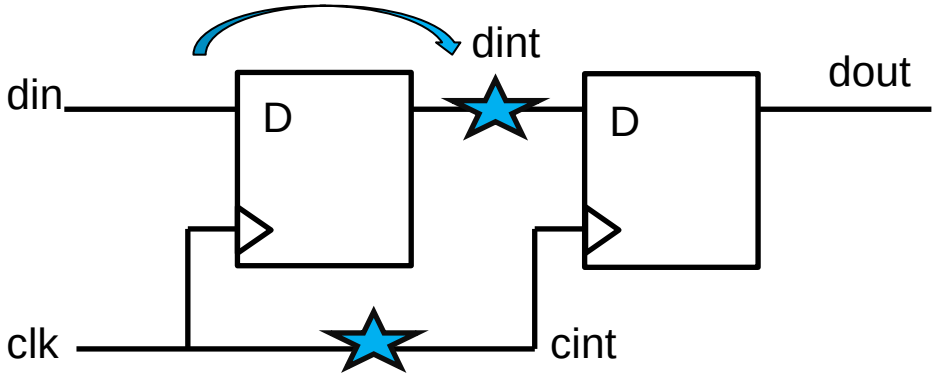
- Force **physical** time delay in sequential data paths i.e.
  - **Sequential data assignment** is delayed in ‘physical time’, instead of ‘delta’ time.

**Verilog**

```
dint <= #1 din ;
```

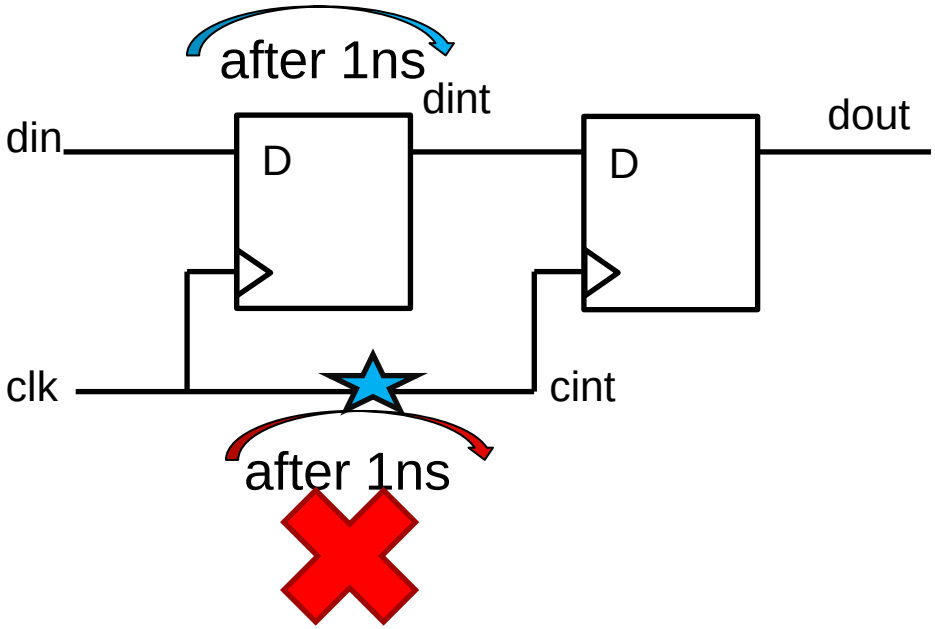
**VHDL**

```
dint<= din after 1 ns;
```




This ensures there is extra delay in data.

# Rule 2 - No 'physical delay' in clock path




- Delay on clock paths competes with delay on Data path
- Clock vs Data scheduling still un-reliable

This ensures a minimum delay on clock path.

Rule1 + Rule2 ensure that data is updated after clock → no risk of shoot-thru