HDL Design using Verilog
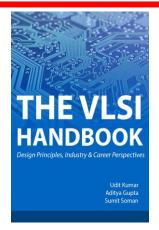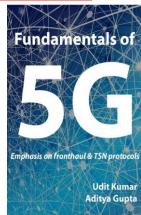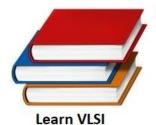
Udit Kumar, PhD, IIT Delhi.

17+ years experience, Author

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

THE VLSI HANDBOOK
Design Principles, Industry & Career Perspectives
Udit Kumar
Aditya Gupta
Sumit Soman

Fundamentals of 5G
Emphasis on fronthaul & TSN protocols
Udit Kumar
Aditya Gupta

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

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

Learn VLSI

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, copyright violation or sufficiency of the same.

- Some of materials were copied from the sources freely available on the internet. When possible, these sources have been cited; however, some references may have been cited incorrectly or overlooked. If you feel that a picture, graph, or code example has been copied from you and either needs to be cited or removed, please feel free to email: elearn.vlsi@gmail.com and we will address this as soon as possible.

- **Take prior approval for Commercial usage of this information.**

- **Views expressed here are personal views and not endorsed by present or past employer.**

Learn VLSI

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
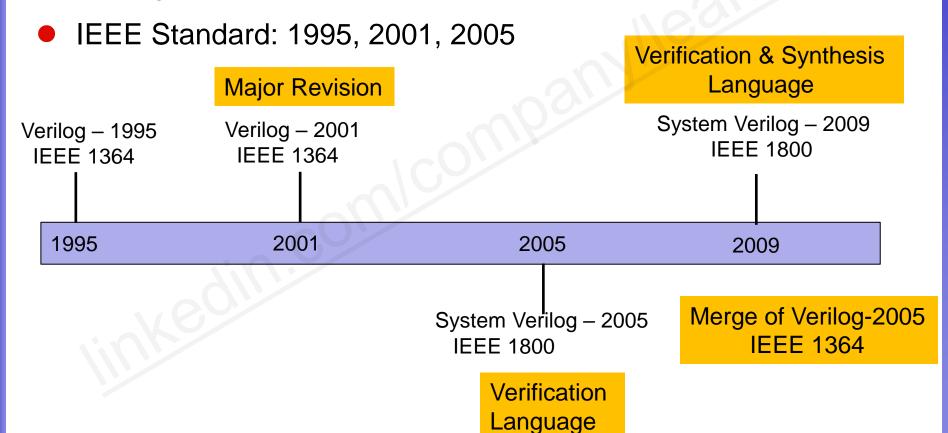
Need for HDL Language

- HDL offers a way to model piece of hardware using a text based programming language.

- Synthesis tools convert HDL into logic gates.

- HDL languages
  - Verilog
  - System Verilog
  - VHDL

- Difference from Software
  - Need hardware mindset
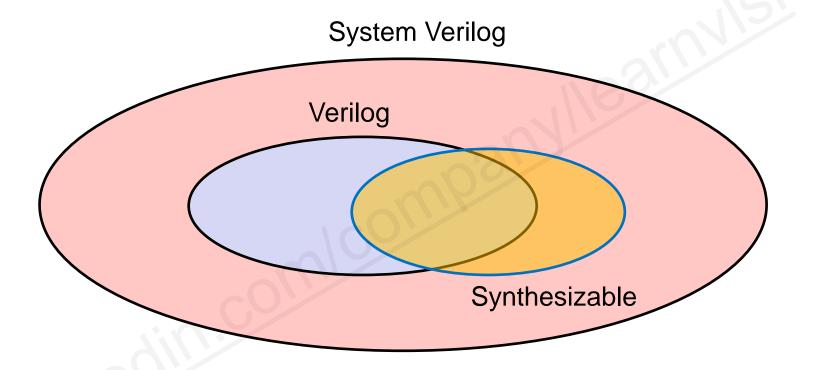 - Work in parallel not sequential like software

History

- Developed by Gateway Design Automation (Proprietary Language): 1984

- Verilog made an open Standard: 1990

- IEEE Standard: 1995, 2001, 2005

Major Revision

Verification & Synthesis Language

Verilog – 1995
IEEE 1364

Verilog – 2001
IEEE 1364

System Verilog – 2009
IEEE 1800

| 1995 | 2001 | 2005 | 2009 |

System Verilog – 2005
IEEE 1800

Merge of Verilog-2005
IEEE 1364

Verification Language
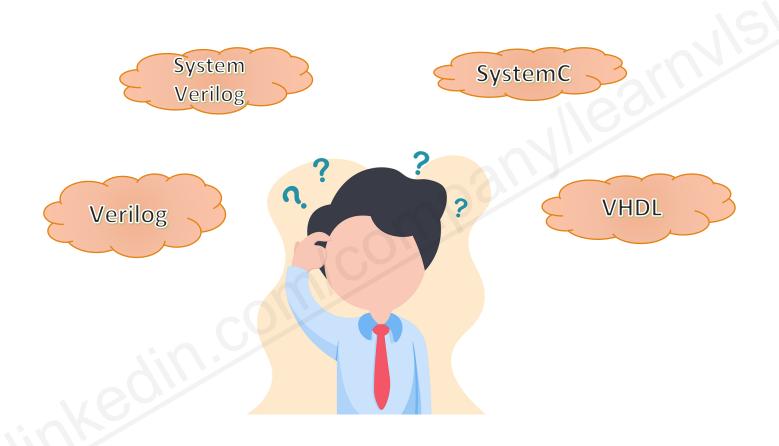
Verilog and System Verilog

- System Verilog is superset of Verilog.

- Both Verilog and System Verilog contains Synthesizable and non synthesizable constructs.
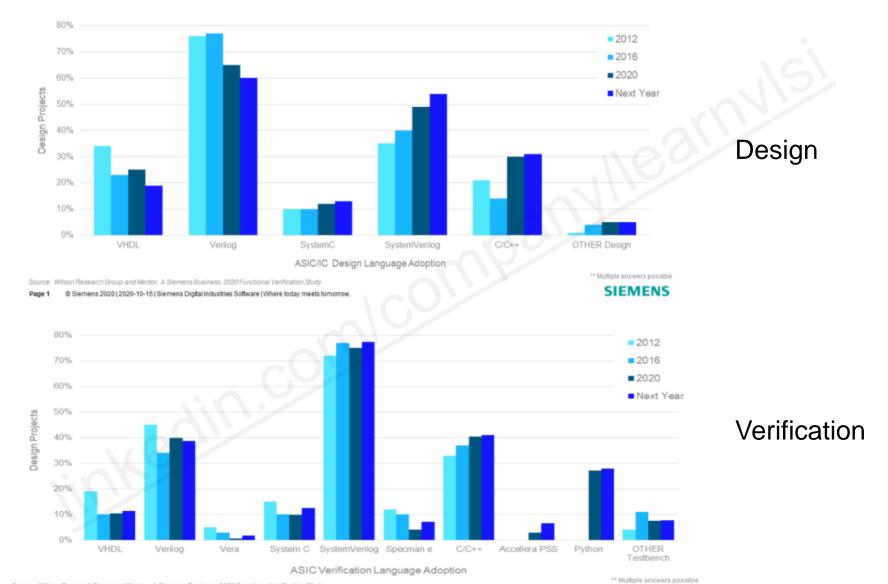
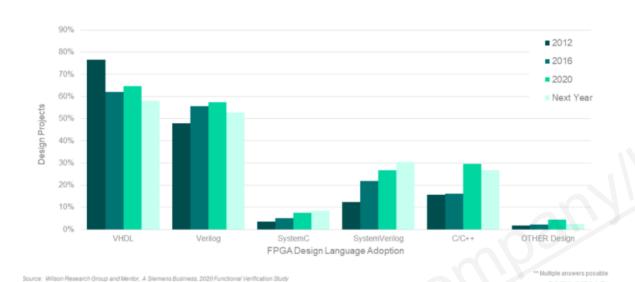# Confusion: Where to start?

ASIC/IC Language

Design

Verification
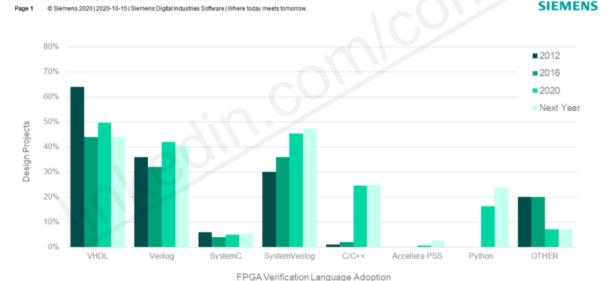
FPGA Design and Verification

Design

Verification
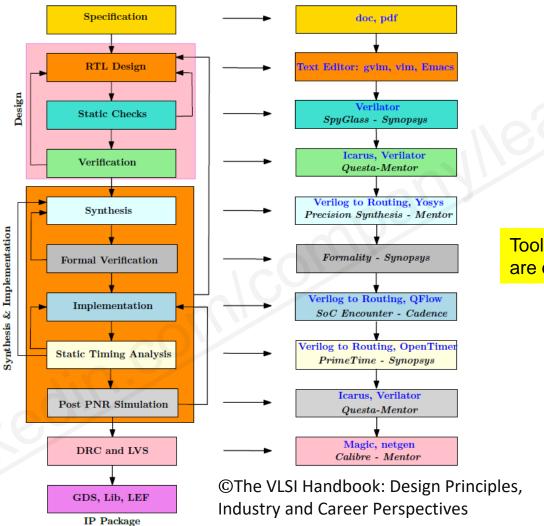
Hard Digital IP Design Flow
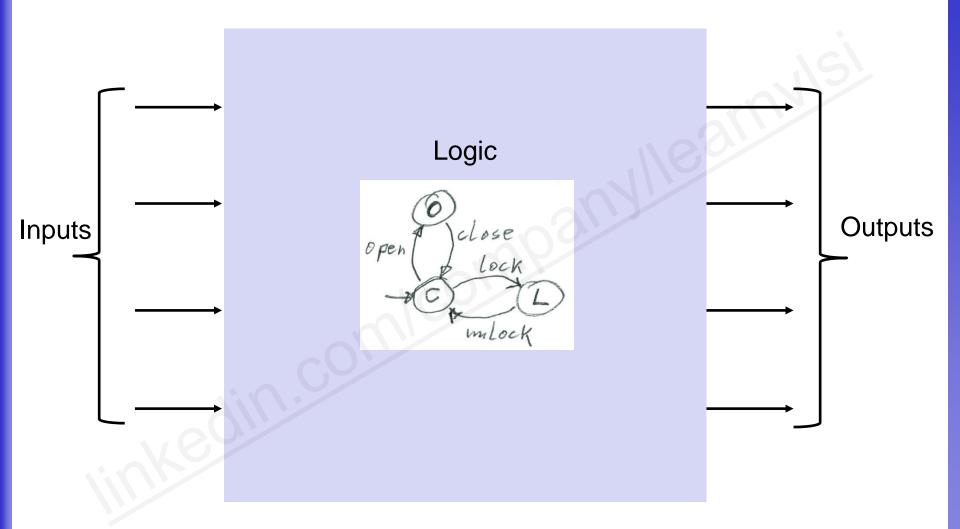
Tools written in blue color are open source tools

Design Flow	Tool
Specification	doc, pdf
RTL Design	Text Editor: gvim, vim, Emacs
Static Checks	Verilator / SpyGlass - Synopsys
Verification	Icarus, Verilator / Questa-Mentor
Synthesis	Verilog to Routing, Yosys / Precision Synthesis - Mentor
Formal Verification	Formality - Synopsys
Implementation	Verilog to Routing, QFlow / SoC Encounter - Cadence
Static Timing Analysis	Verilog to Routing, OpenTimer / PrimeTime - Synopsys
Post PNR Simulation	Icarus, Verilator / Questa-Mentor
DRC and LVS	Magic, netgen / Calibre - Mentor
GDS, Lib, LEF (IP Package)	

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

Reference: Book "The VLSI Handbook: Design Principles, Industry and Career Perspectives", Udit Kumar, Aditya Gupta, Sumit Soman
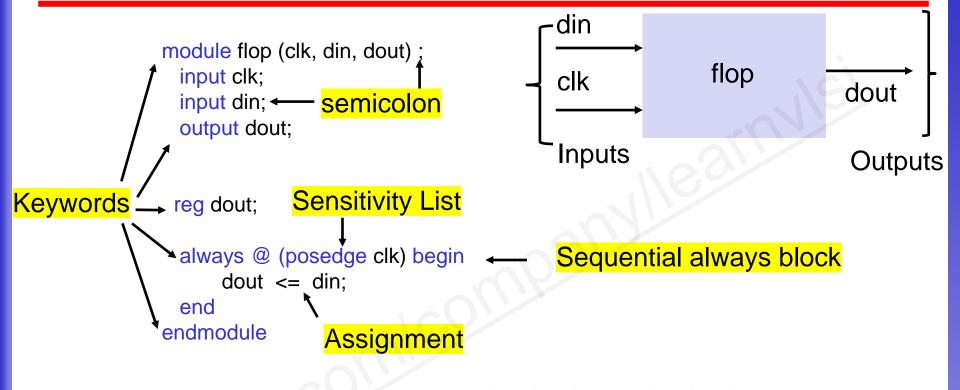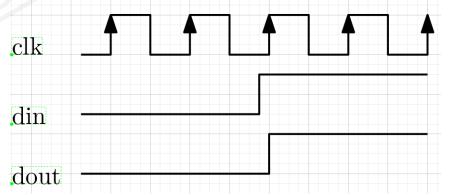
First Step for HDL: Hardware Mindset

Logic

Inputs

Outputs

Thinking in terms of digital design is key.

An Example

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

    reg dout;

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

semicolon

Keywords

Sensitivity List

Sequential always block

Assignment

din

clk

flop

dout

Inputs

Outputs

clk

din

dout

Synchronous and Asynchronous Reset

```verilog
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
```

```verilog
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
```

Difference in the sensitivity list

Reset Check

Conditional statement

Async Reset

Sync Reset

Behavioral Vs structural

Behavioral

din

clk

dout

flop

rst_n

```verilog
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
```

Structural

din

clk

rst_n

din | rst_n | dout

clk

dout

```verilog
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 behavioral implementations could have same behavior.
  - ⊙ Easier to write and understand

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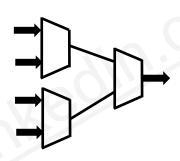 - ⊙ Explicit structure of the circuit
  - ⊙ Useful to build bigger designs.

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
```

Procedural Assignment

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


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

endmodule
```

Operators

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

```verilog
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
```

din →

clk →

rst_n →

flop

→ dout

Let's summarize

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

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

endmodule

Testbench

```verilog
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

```verilog
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

always @ (*)
    #10 clk = ! clk;
end

Always @ (clk)
    #10 clk = ! clk;
end


always @ (clk)
    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`

3	2	1	0

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

3	2	1	0
3	2	1	0

Numbers representation

Format   Value

Size   ⟶   4'b1010 // 4-bit binary number for 10

4'hB   // 4- bit hex number

4'd10   // 4-bit decimal number

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?

```verilog
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
```

```verilog
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: What will be output for these code?**

```
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
```
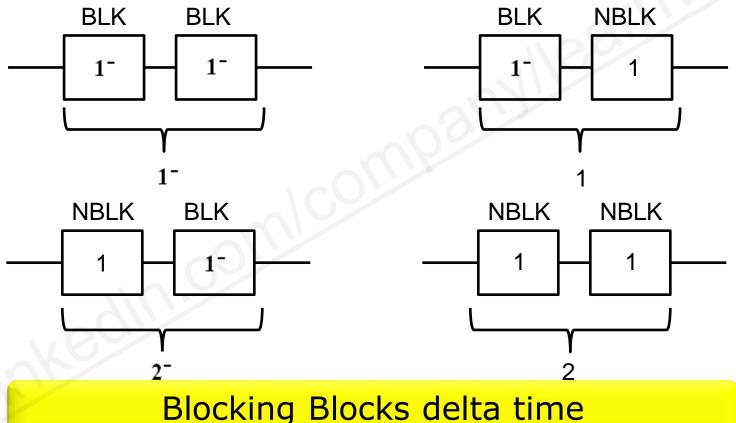
Blocking vs non-blocking assignments

- Delta delay occurrence in such assignments

1^- Start of delta time

BLK   BLK
1^- 1^-
1^-

BLK   NBLK
1^- 1
1

NBLK   BLK
1 1^-
2^-

NBLK   NBLK
1 1
2

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 =  S1;
end


 -- No delta cycle consumed
```

```
// Non blocking assignment

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

always @ (posedge clk)
begin
  S3 <=  S2;
  Q   <= S3;
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
    ```
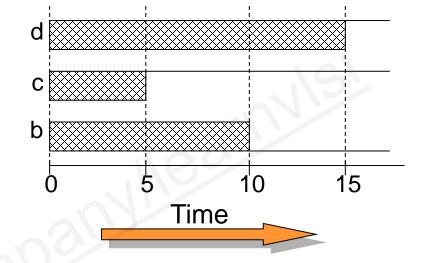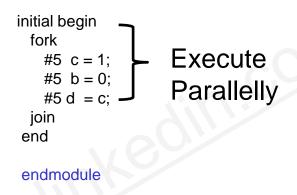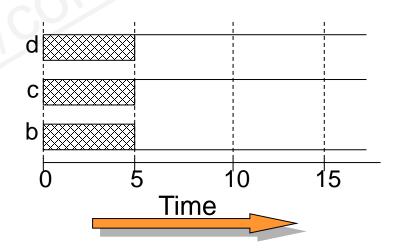
Fork/Join

```
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

Time

Time

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
```
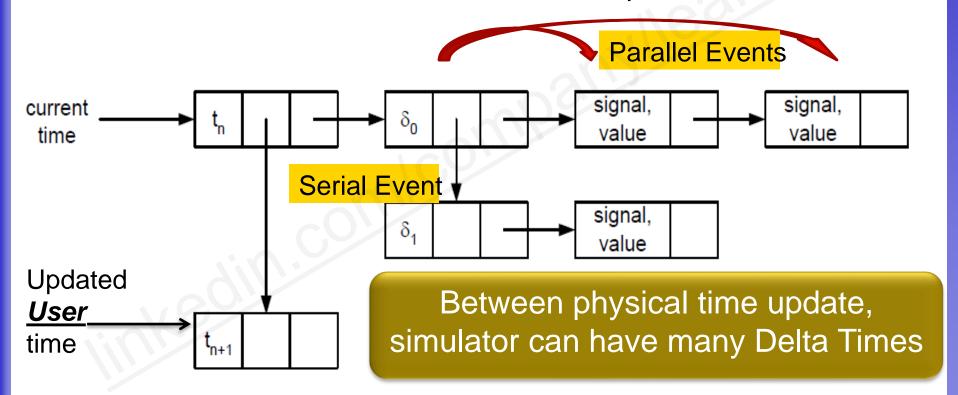
Wait for elapse of the time and than evaluate the expression

What is Delta Delay?

- A delta time, used <u>inside</u> simulator database, is infinitely small but useful…

- Each event consumes one *delta* time-step.

Parallel Events

Serial Event

current time → t_n → δ_0 → signal, value → signal, value

δ_1 → signal, value

Updated ***User*** time → t_{n+1}

Between physical time update, simulator can have many Delta Times

Verilog has event queue

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

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

Inactive events
- #0 blocking assignments

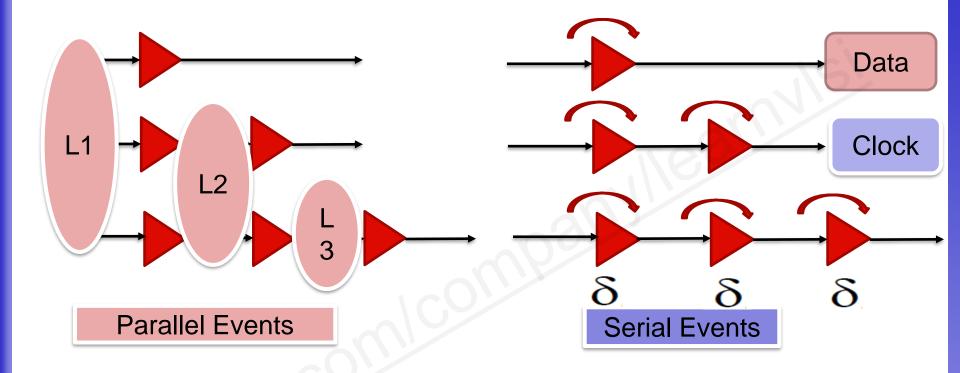**Non-blocking**
- Update LHS of non-blocking assignments

Monitor events
- $monitor & $strobe

Multiple events are grouped together and execute in defined order.

Concept of Parallel & Serial events

Parallel Events
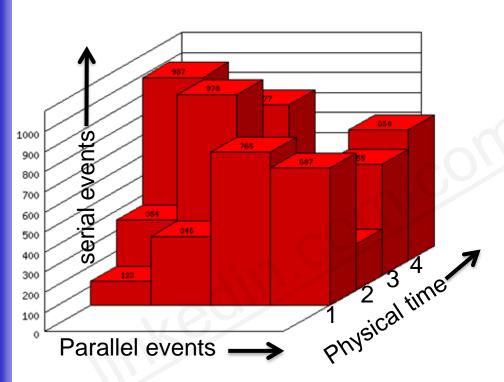
Serial Events

Data

Clock

- All events are software objects & consume "Delta" delay.

- Simulators do not understand relationship between data & clock events.

Discrete Event Simulation

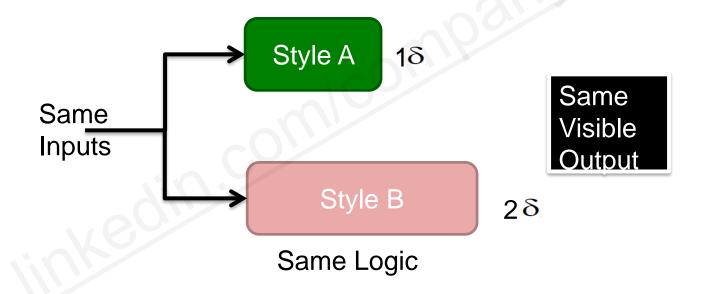- A method to mimic events in physical systems, using software.

  ⊙ 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.
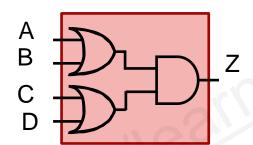
HDL writing affects events

- 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.
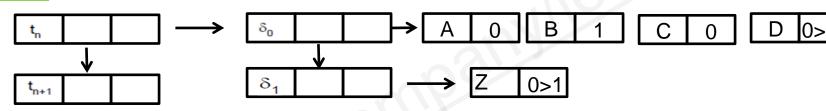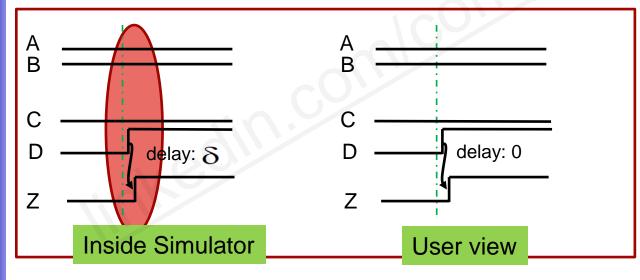

Same Inputs

Style A 1δ

Style B 2δ

Same Logic

Same Visible Output

Style A: Output coming after 1 Delta
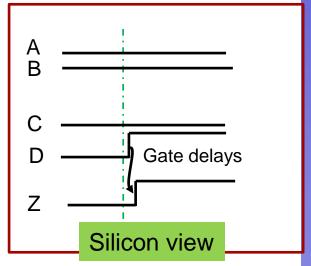
```
always  @ (A or B or C or D) begin
  Z <=  (A || B) && (C || D);
end
```


Test case

t_n		

t_{n+1}		

δ_0		

δ_1		

| A | 0 | | B | 1 | | C | 0 | | D | 0>1 |

| Z | 0>1 |

Inside Simulator delay: δ

User view delay: 0

Silicon view Gate delays
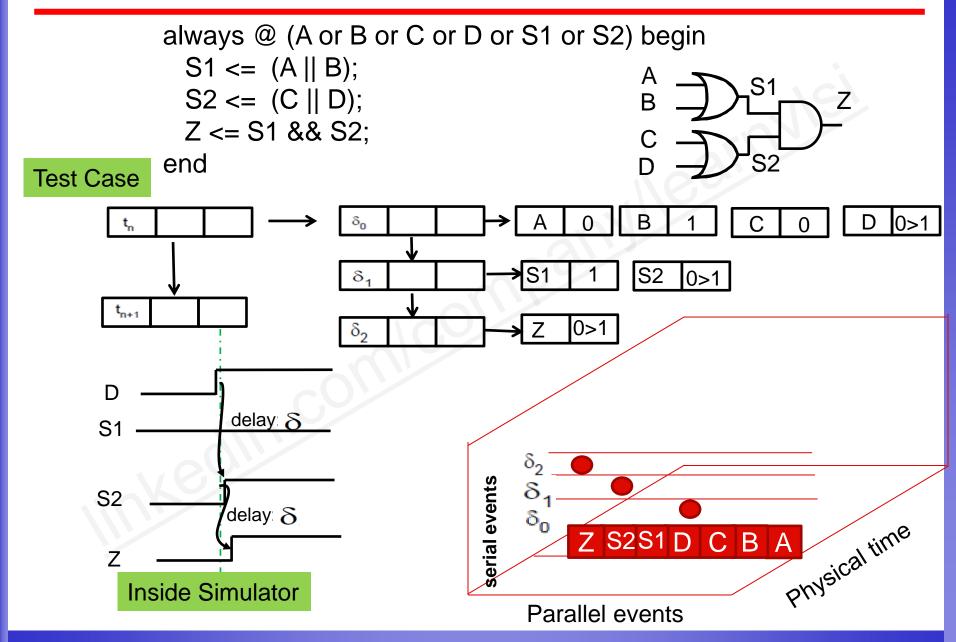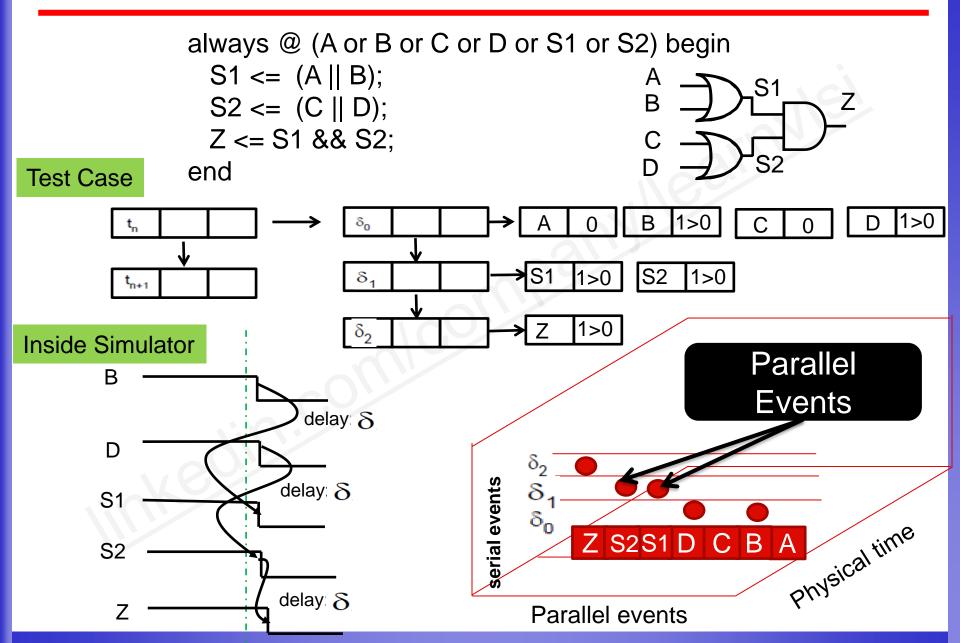
Style B: Output coming after 2 Delta

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

Test Case

Inside Simulator

serial events

Parallel events

Physical time

Style B with Multiple Input Changes

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

Test Case

| δ_0 | | | A | 0 | B | 1>0 | C | 0 | D | 1>0 |

| δ_1 | | | S1 | 1>0 | S2 | 1>0 |

| δ_2 | | | Z | 1>0 |

Inside Simulator

B

delay: δ

D

delay: δ

S1

S2

delay: δ

Z

Parallel Events

δ_2
δ_1
δ_0

serial events

Z S2 S1 D C B A

Physical time

Parallel events

Learn VLSI
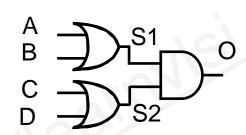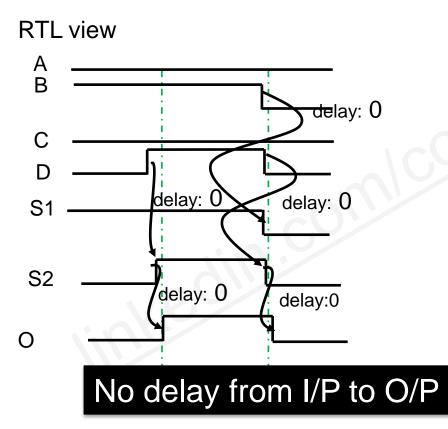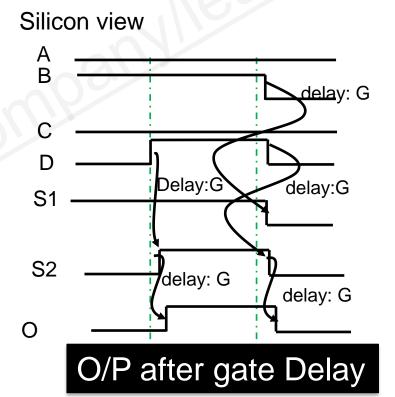
```
Always @ (A or B or C or D) begin
  S1 <=  (A || B);
  S2 <=  (C || D);
  O <= S1 && S2;
end
```


RTL view

delay: 0

delay: 0   delay: 0

delay: 0   delay:0

No delay from I/P to O/P

Silicon view

delay: G

Delay:G   delay:G

delay: G   delay: G

O/P after gate Delay
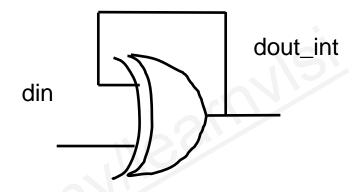
Delta delay and Physical delay
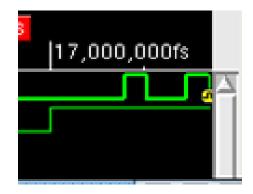
```
reg dout_int=0;
always @ (din or dout_int) begin
  dout_int <= dout_int ^ din;
end
```


- RTL Simulation will hang

- Gate level simulation will produce train of pulse

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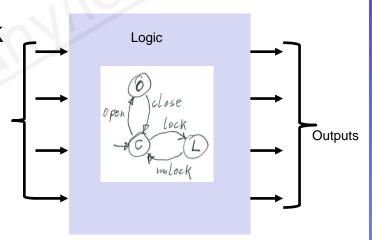
Design examples

- 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.
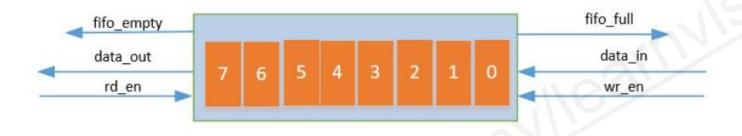

Logic

Inputs

Outputs

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        ◄─── Sequential always block
      if (!rst_n) begin
        count <= 1'b0;
      end else begin
        if (en) begin        ◄─── Combinational logic
          count  <=  count + 1;        ◄──
        end
      end
  end
endmodule        ◄── Flops
```

din
clk
en

count

Outputs

count
clk

| 0 ns | | 50 ns | | 100 ns | | 150 ns | | 200 ns | | 250 ns | | 3 |

Synchronous FIFO Design

Write Control

RAM

Read Control

FIFO full and FIFO Empty

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.
  - Active low 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 use.
- 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 without async or sync reset.

Link for other presentations

- HDL Design using Verilog
  - https://www.linkedin.com/feed/update/urn:li:activity:6901101173491798016

- RTL Design Guidelines
  - https://www.linkedin.com/feed/update/urn:li:activity:6903289386536968192

- Clock domain crossing:
  - https://www.linkedin.com/feed/update/urn:li:activity:6906226196003061760

- Low Power Design:
  - https://www.linkedin.com/feed/update/urn:li:activity:6913837834302885888

- Low Power RTL Design
  - https://www.linkedin.com/feed/update/urn:li:activity:6914792874811297793

- Power optimization at Architecture and implementation level
  - https://www.linkedin.com/feed/update/urn:li:activity:6915873128191590400

- VLSI Design Flows and Open source tools:
  - https://www.linkedin.com/feed/update/urn:li:activity:6886886690405924864

Learn VLSI website: https://www.sites.google.com/view/learnvlsi/webinar

Telegram Channel: https://t.me/elearnvlsi

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

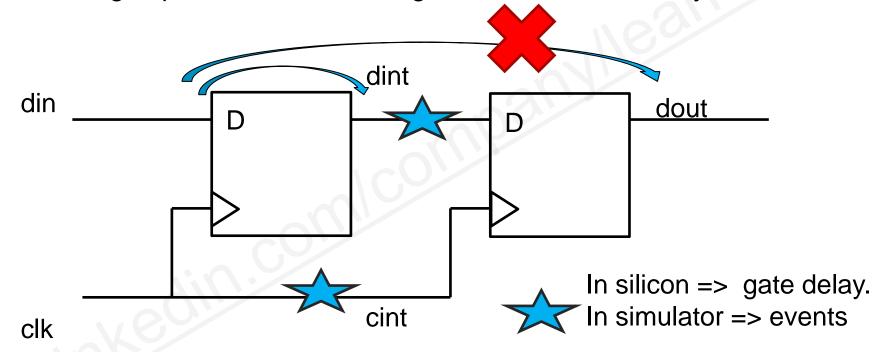**Feedback/Errata: Please send email to elearnvlsi@gmail.com**

After each improvement, the updated slides will be available at website:
https://www.sites.google.com/view/learnvlsi/webinar

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.

In silicon => gate delay.
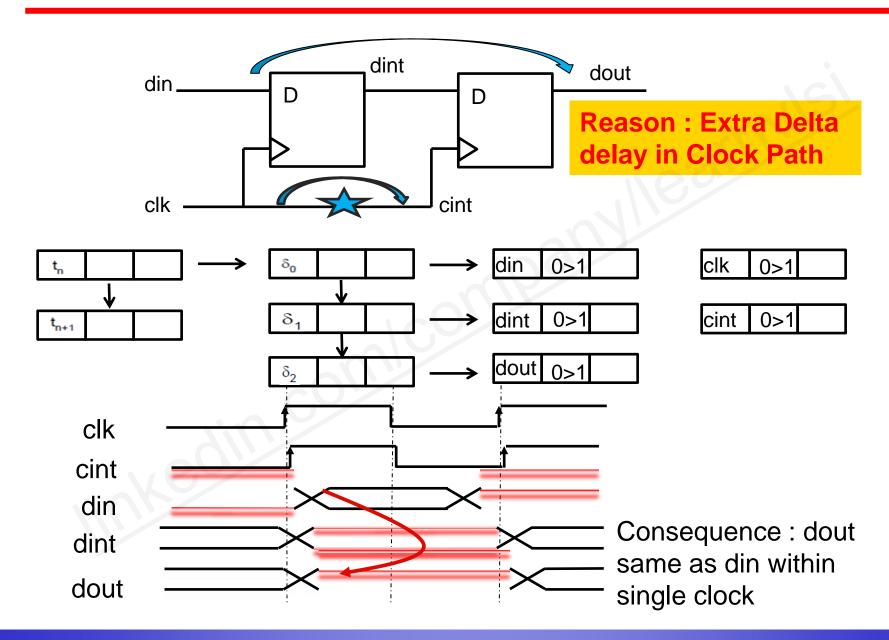In simulator => events

Shoot-thru:
        (events in data path) <= (events in clock path)

How Simulation Shoot-thru occurs?

Reason : Extra Delta delay in Clock Path

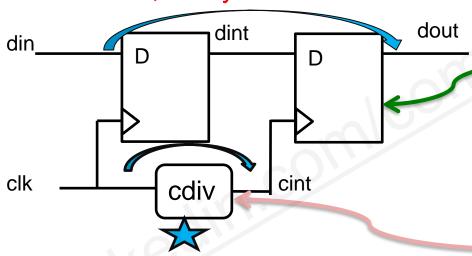Consequence : dout same as din within single clock

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

Non-blocking assignments consume delta delay

```
// Input data sampling
always @ (posedge clk, negedge rst_n) begin
    if (rst_n==1'b0)
                dint  <= 1'b0;
      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
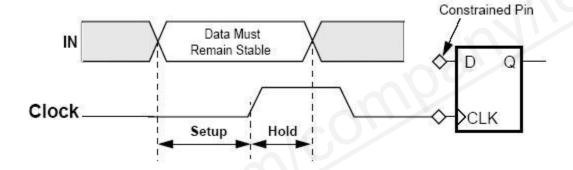
```
always @ (posedge clk, negedge rst_n) begin
    if (rst_n==1'b0)
                cint    <= 1'b0;
      else
                cint    <= !cint;
```

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.
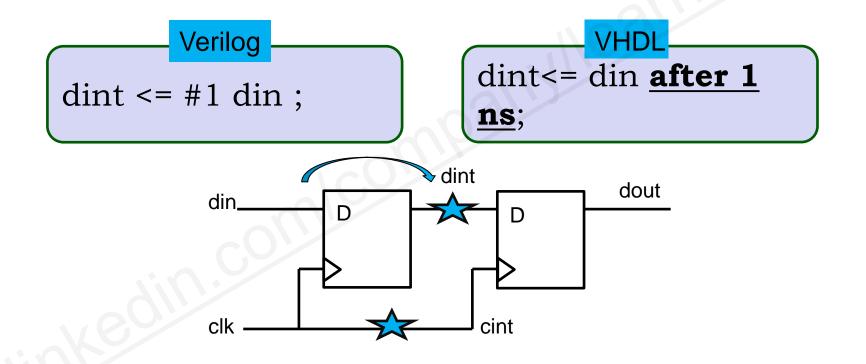

- **By doing Gate level simulation?**
  - ◉ **Yes** but it is very costly.

- **By using robust RTL coding rules?**
  - ◉ **Yes** this is easy and efficient (see subsequent slides)
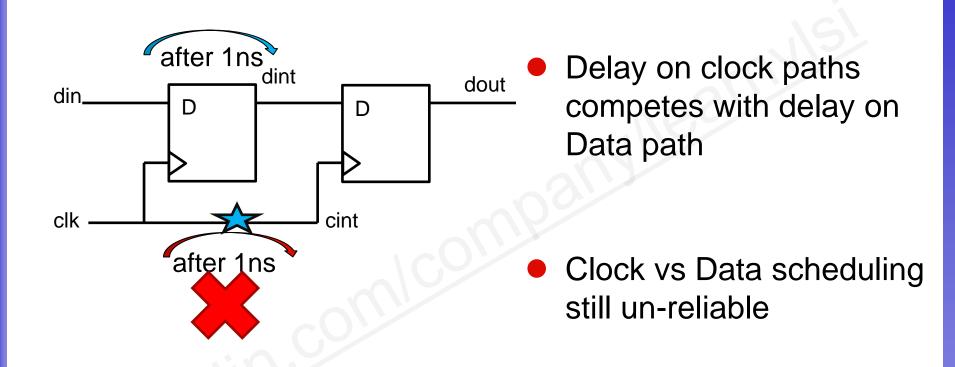
Rule 1 - Force scheduling of data

- 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

$$dint <= din\ \underline{\textbf{after 1 ns}};$$

This ensures there is extra delay in data.

Rule 2 - No 'physical delay' in clock path

after 1ns

dint

din ____ [D] dout

[D]

clk ____ ⭐ cint

after 1ns

❌

- 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