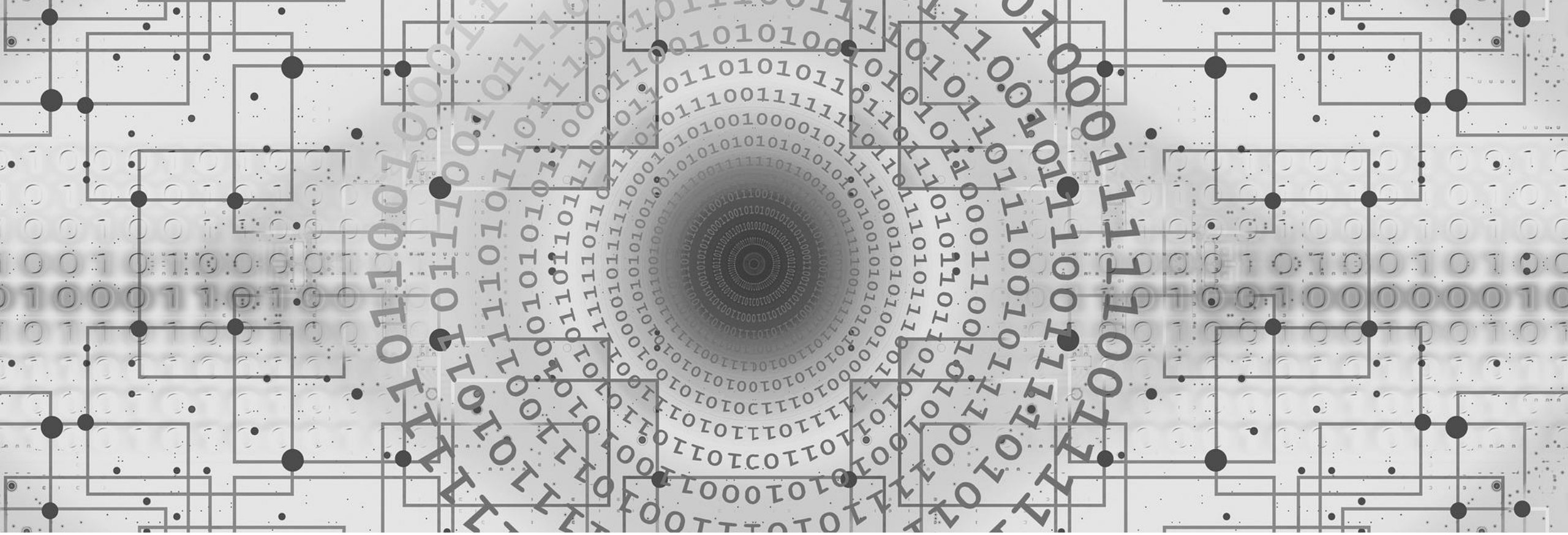


Lecture-8 and 9 Procedural Blocks

ECE-111 Advanced Digital Design Project

Vishal Karna

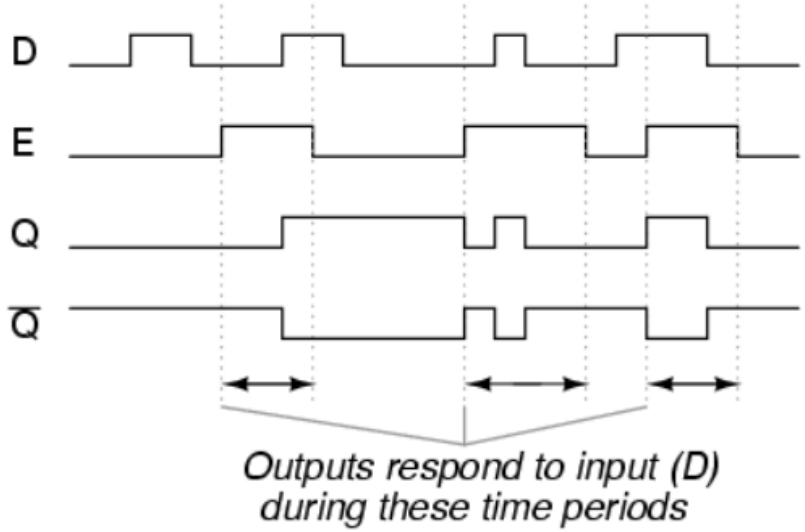
UC San Diego
JACOBS SCHOOL OF ENGINEERING
Electrical and Computer Engineering

Winter 2022

Difference Between a Latch and a Flip-Flop

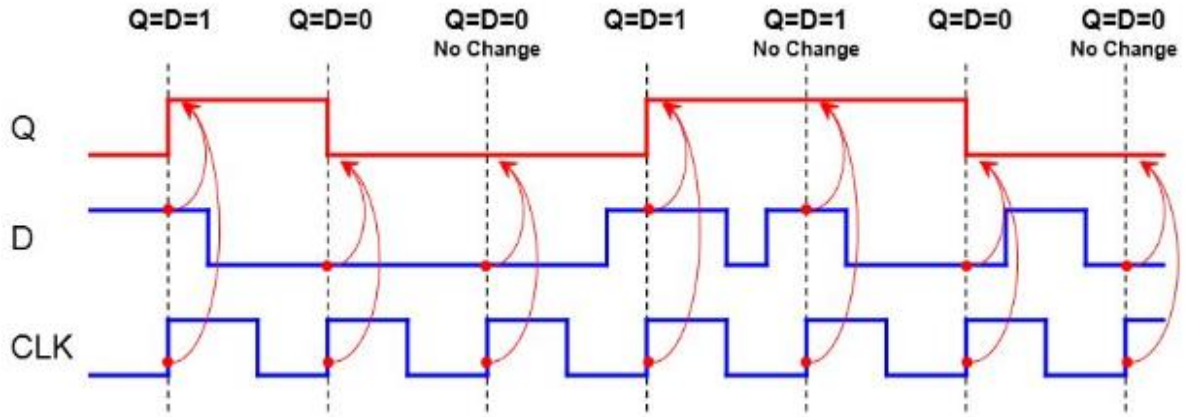
Latch

- Latch is controlled by level triggered enable signal (either positive or negative level triggered)
- Latch is **asynchronous** (its output state changes as soon as its input changes and when active level is maintained at Enable input)



Flip-Flop

- Flip-Flop is controlled by edge triggered control signal, such as clock (either positive or negative edge triggered).
- Flip-Flop is **synchronous** (its output responds to the changes in the input only at positive or negative edge of input clock signal)



System Verilog Procedural block

- **SystemVerilog provides two types of procedural blocks :**
  - initial
  - always
    - **always@, always_ff, always_comb, always_latch**

- **always blocks are synthesizable procedural blocks and it is used for developing RTL code which specifies design behavior**
  - Synthesize compiler converts **always** block into actual hardware logic

- **Initial block is a non-synthesizable procedural block and it is used for simulation purpose**
  - It is typically used for developing testbench code which contains stimulus for design
  - Synthesize compiler will ignore initial block if specified within design modules
    - **initial** block will not be synthesized into hardware logic

Always Block : always@

- **always procedural blocks are used to describe events that should happen under certain conditions**
  - always block runs continuously throughout the simulation
  - module can have multiple always blocks and each of them running concurrently at given time step

- **Syntax and structure of always block** :

  always@(sensitivity list) **begin**

  <procedural statements> } **body**

  end

Example :

```
logic sum;
always@(a,b) begin // a, b is a trigger condition for always block
 sum = a + b;   // statement will execute if either a or b changes
end
```

- **begin** and **end** is required if there are multiple procedural statement, otherwise it is optional

- **The sensitivity list tells the always block when to execute its body**
  - Whenever any variable in the sensitivity list changes, the **always** executes its procedural statements enclosed within **begin** and **end**
  - Depending on the way the sensitivity list is specified, either combinational logic or sequential logic (such as flip-flop) will be inferred
- **Variables on the LHS of procedural statements** inside **always** block must be of type **logic** or **reg**

Always Block : always@

- **Always procedure can be used to model :**
  - Combinational logic
  - Clocked sequential logic (such as flipflops)
  - Level sensitive logic (such as latches)

- **Sensitivity list can have any number of signals and it can be specified in multiple different ways :**
  - Edge sensitive list used for **sequential circuit**
    - **always**@(**posedge** clk) // always block will trigger whenever there is a rising edge of clk
    - **always**@(**negedge** clk) // always block will trigger whenever there is a negedge edge of clk
    - **always**@(**posedge** clk or **negedge** reset) // always block will trigger if there is a rising edge of clk or falling edge of reset
  - Level sensitive list used for **combinational circuit**
    - **always**@(address) // always block will trigger whenever address value changes
    - **always**@(mode or select)
    - **always**@(mode, select)

    There is no difference and no advantage using "or" over "," separator In sensitivity list representation. Using "or" is more verbose however could be confusing and might be treated as "or" gate. Suggestion is to use comma as a separator in sensitivity list

- **Synthesis compiler will infer a flip-flop (sequential logic) if sensitivity list** has **posedge** or **negedge**

Types of always procedural block

Category	Usage Example	Purpose	Introduced in Verilog or SV ?
always@(<level sensitivity list>)	**always@**(a, b) **begin** // assignment statements **end**	Model Combinational Logic	Verilog, SystemVerilog
always@(<edge sensitivity list>)	**always@**(**posedge** clk, **negedge** reset) **begin** // assignment statements **end**	Model Sequential Logic	Verilog, SystemVerilog
always@(*)	**always@**(*) **begin** // assignment statements **end**	Model Combinational or Sequential Logic	Verilog, SystemVerilog
always_comb	**always_comb begin** // assignment statements **end**	Model Combinational Logic	SystemVerilog
always_ff@(<edge sensitivity list>)	**always_ff@**(**posedge** clk, **negedge** reset) **begin** // assignment statements **end**	Model Sequential Logic	SystemVerilog
always_latch	**always_latch begin** // assignment statements **end**	Model a Latch (Level Sensitive Sequential Logic)	SystemVerilog

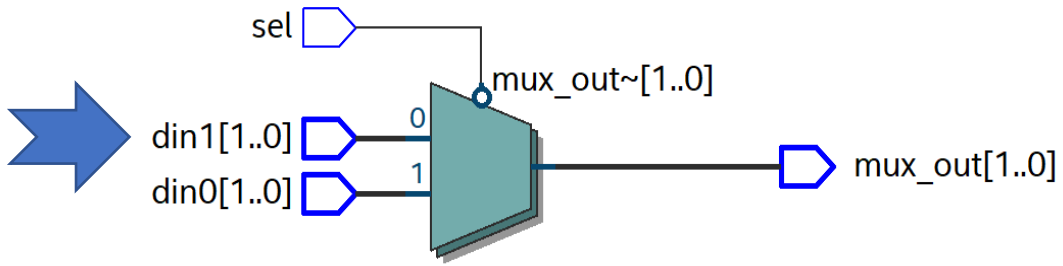
Always Block : always@

- Multiplexer using always@ procedural block

```
module mux(
  input  logic[1:0] din0,
  input  logic[1:0] din1,
  input  logic sel,
  output logic[1:0] mux_out
);

 // always block to describe 2to1 multiplexor
 always@(sel,din0,din1) // complete sensitivity list
  begin
    if(sel == 1'b0) begin
      mux_out = din0;
    end else begin
      mux_out = din1;
    end
  end
endmodule: mux
```

Body of always block with procedural and conditional statements

Synthesis compiler will generate 2to1 MUX



Always Block : always@

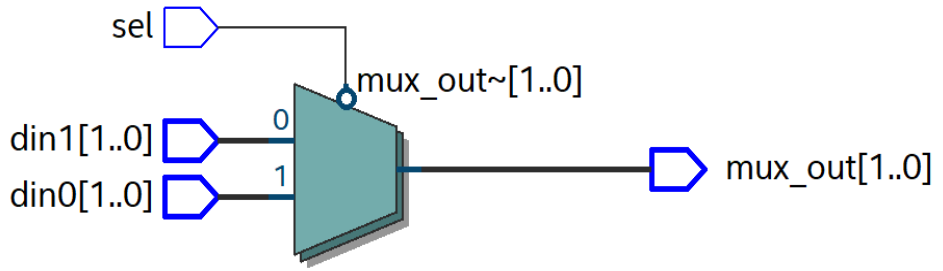
- The sensitivity list must include all input signals used by procedural statement within always block to properly model combinational logic.
  - Omission of any input signal which impacts behavior of logic, can lead to simulation and synthesis mismatches.

- In below mentioned example din1 input signal is missed out in sensitivity list specification,
  - Synthesis result would still produce a 2to1 MUX, however when simulating design any change in din1 will not propagate to output signal mux_out even when sel value is set to 1'b0.

```
module mux(
 input  logic[1:0] din0, din1
 input  logic sel,
 output logic[1:0] mux_out );

 // always block to describe 2to1 multiplexor
 always@(sel, din0) // din1 missed out in sensitivity list
  begin
   if(sel == 1'b0) begin
     mux_out = din0;
   end else begin
     mux_out = din1;
   end
  end
endmodule: mux
```

- Multiple procedural statements with blocking "=" assignments
- Change in value of din1 will not propagate to output of mux due to din1 signal not specified in sensitivity list of always block

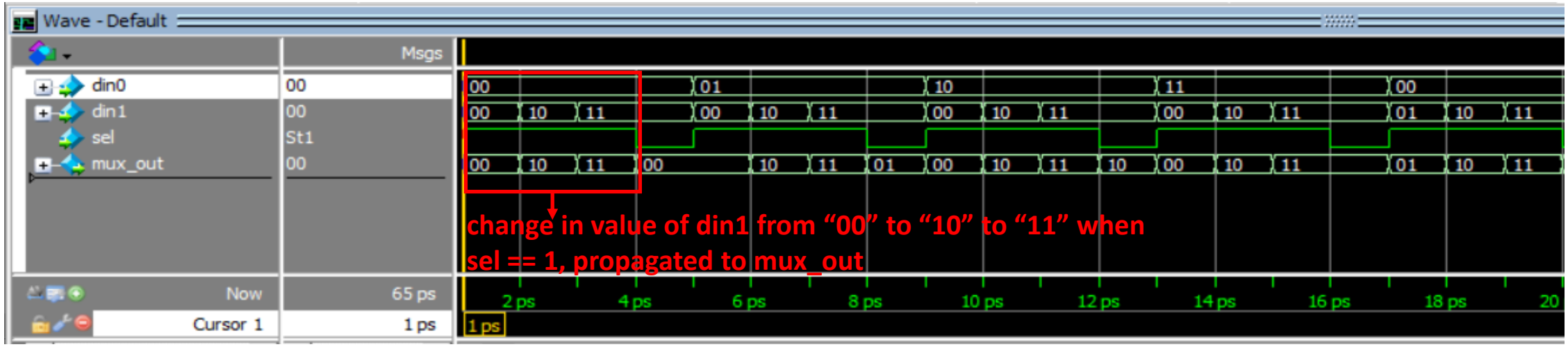
Synthesis compiler will still generate 2to1 MUX even with din1 signal missing in sensitivity list.



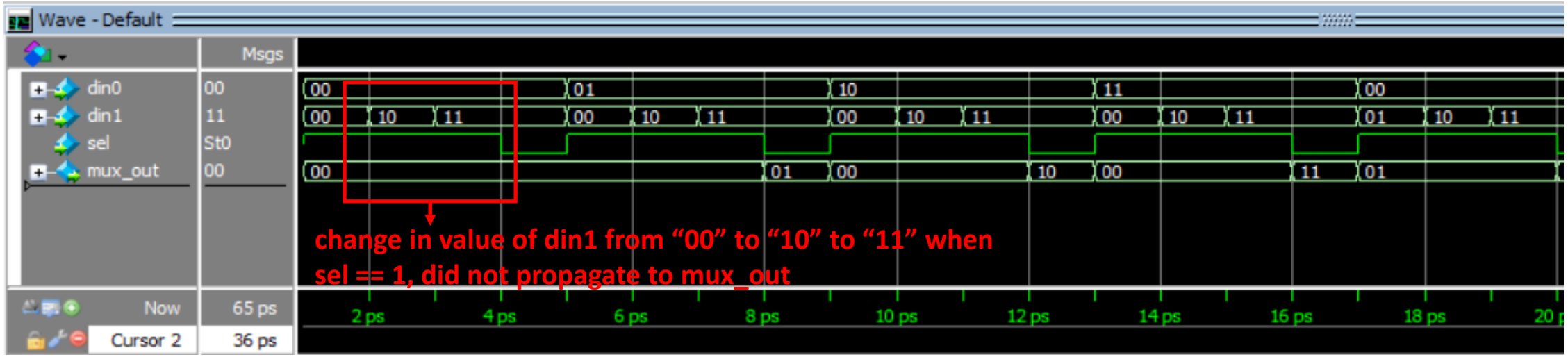
Simulation vs Synthesis results mis-match due to incomplete sensitivity list !!!

Always Block : always@

- Simulation result for mux implementation with complete sensitivity list



- Simulation result for mux implementation with din1 missing in sensitivity list



Always Block : always@*

- Verilog-2001 attempted to address incomplete sensitivity list with the addition of special token **@*** that would infer a complete sensitivity list
  - **always@*** or **always@(*)** both representation means the same

```
module mux(
  input  logic[1:0] din0,
  input  logic[1:0] din1,
  input  logic sel,
  output logic[1:0] mux_out
);

 // always block to describe 2to1 multiplexor
 always@(*) // automatically infers sel, din0, din1 in sensitivity list
  begin
    if(sel == 1'b0) begin
      mux_out = din0;
    end else begin
      mux_out = din1;
    end
  end
endmodule: mux
```

due to din0, din1 all signals in RHS automatically inferred in sensitivity list, any change in value of din0 and din1 will propagate at the output of the mux when sel is 0 and 1 respectively

always@* Limitations

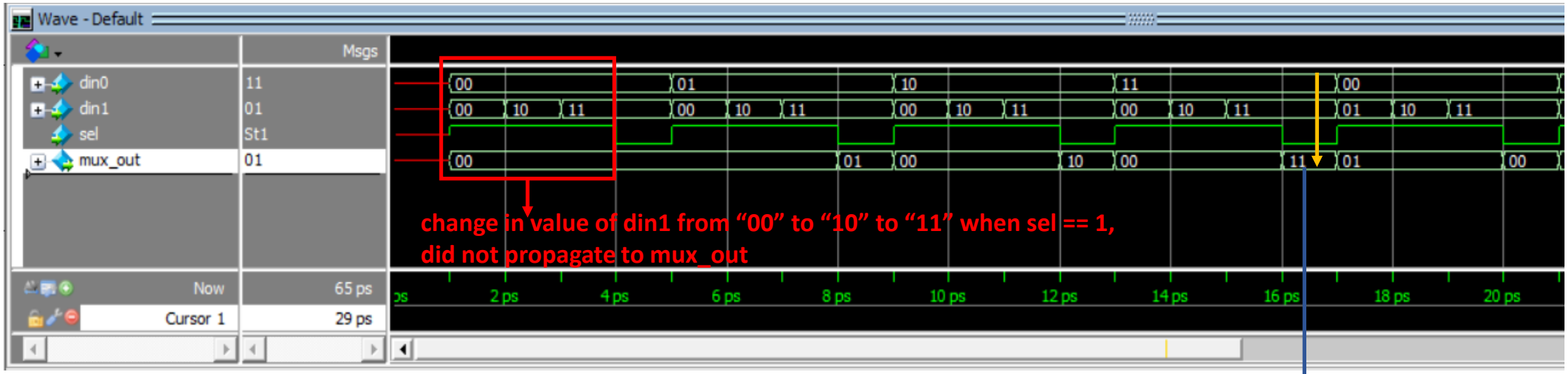
- always@* does not infer a complete sensitivity list when the **always @*** block contains functions.
  - it will not infer sensitivity to signals that are externally referenced in a function or a task that is called from the **always** block.
  - it will only be sensitive to the signals passed into the function or task

```
module mux(
 input  logic[1:0] din0, din1,
 input  logic sel,
 output logic[1:0] mux_out
);
// function to return selected input value
function logic[1:0] func_mux(logic l_sel)
 begin
  if(l_sel == 1'b0) begin
   func_mux = din0;
  end else begin
   func_mux = din1;
  end
 end
endfunction
// example of incomplete sensitivity list inference
always@(*) begin //@(*) will not automatically infer din0 and din1 in sensitivity list
  mux_out = func_mux(sel);
end
endmodule: mux
```

change in **"din0"** or **"din1"** value will not trigger the function **func_mux** since **din0** and **din1** are not part of function arguments and hence values will not propagate to mux_out until **"sel"** value changes

always@* Limitations

- Simulation result of always@(*) with functional call inside always block
  - Due to function call does not have din0 and din1 arguments, **always@(*)** will **not** automatically infer din0 and din1 in sensitivity list.
  - Change in values of din0 and din1 will not propagate to output of mux if sel value does not change



Why did din0 value propagated to mux_out ?
Reason >> since **"sel"** value changed, function **func_mux** gets triggered which in turn caused **"din0"** value to get assigned to **"mux_out"**

How to address always@* incomplete sensitivity list inference?

- **Approach A** : If function or task is called from the always procedural block, replace **always@*** with **always@** having all input signals specified in its sensitivity list.

```
module mux(
  input  logic[1:0] din0, din1,
  input  logic sel,
  output logic[1:0] mux_out
);
// function to return selected input value
function logic[1:0] func_mux(logic l_sel)
  begin
    if(l_sel == 1'b0) begin
      func_mux = din0;
    end else begin
      func_mux = din1;
    end
  end
 endfunction
 // example of complete sensitivity list inference
 always@(sel, din0, din1) begin //sel, din0 and din1 all input signals are in sensitivity list
   mux_out = func_mux(sel);
 end
endmodule: mux
```

change in **"din0"** or **"din1"** value will trigger the function **func_mux** even though **din0** and **din1** are not part of function arguments but **din0** and **din1** are specified in always@ sensitivity list.

How to address always@* incomplete sensitivity list inference ?

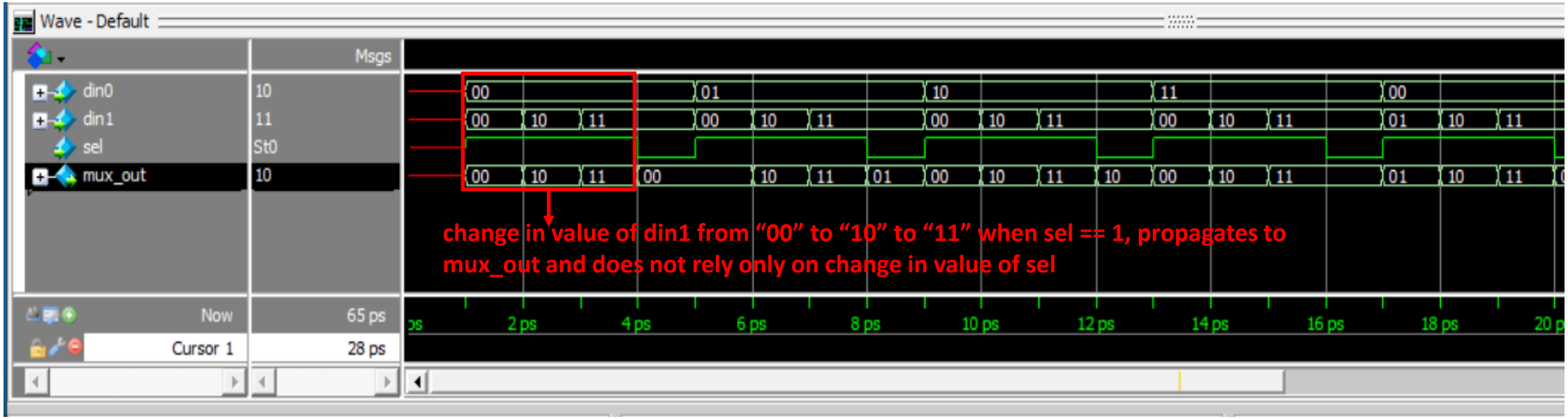
- **Approach B** : List all input signals (**din0, din1** and **sel**) as part of function call argument and update function signature to have additional arguments
  - This will enforce **always@(*)** to automatically infer all **sel, din0 and din1** in its sensitivity list

```
module mux(
 input  logic[1:0] din0, din1,
 input  logic sel,
 output logic[1:0] mux_out
);
// function to return selected input value
function logic[1:0] func_mux(logic l_sel, logic din_0, logic din_1)
 begin
  if(l_sel == 1'b0) begin
   func_mux = din_0;
  end else begin
   func_mux = din_1;
  end
 end
endfunction
// example of complete sensitivity list inference
always@(*) begin //sel, din0 and din1 are automatically inferred in sensitivity list
  mux_out = func_mux(sel, din0, din1);
end
endmodule: mux
```

change in **"din0"** or **"din1"** value will trigger the function **func_mux** since **din0** and **din1** are provided as arguments to **func_max** causing **always@*** to infer automatically din0 and din1 in its sensitivity list

always@* Limitations

- Simulation result of **Approach A** and **Approach B** RTL model implementations



- **Approach A** requires changing always@(*) with always@(sel, din0, din1)
  - Responsibility of designer to ensure all required input signals are specified in sensitivity list
  - In case of complex RTL Models, there is a possibility designer might unintentionally forget to mention some of the required input signals in sensitivity list, resulting in simulation vs synthesis mis-matches
- **Approach B** requires change in function argument list and potentially change in code inside function
  - In case of complex RTL Models, changing function implementation might not be always feasible
- Is there a better way to address this limitation ?
  - **Yes, use "always_comb" supported by SystemVerilog !**

Always block : always_comb

- An **always_comb** will infer an accurate sensitivity list for combinational logic without designer to explicitly specify all required input signals in always@ sensitivity list
  - Within **always_comb,** function calls does not have to have all input signals as part of the argument list, since all required input signals are automatically inferred in sensitivity list

```
module mux(
  input  logic[1:0] din0, din1,
  input  logic sel,
  output logic[1:0] mux_out
);
// function to return selected input value
function logic[1:0] func_mux(logic l_sel) begin
    if(l_sel == 1'b0) begin
      func_mux = din0;
    end else begin
      func_mux = din1;
    end
  end
 endfunction
 // example of automatic complete sensitivity list inference
 always_comb begin //sel, din0 and din1 all input signals are automatically inferred in sensitivity list
   mux_out = func_mux(sel);
 end
endmodule: mux
```

change in **"din0"** or **"din1"** value will trigger the function **func_mux** even though **din0** and **din1** are not part of function arguments since always_comb will automatically infer **din0** and **din1** in sensitivity list.

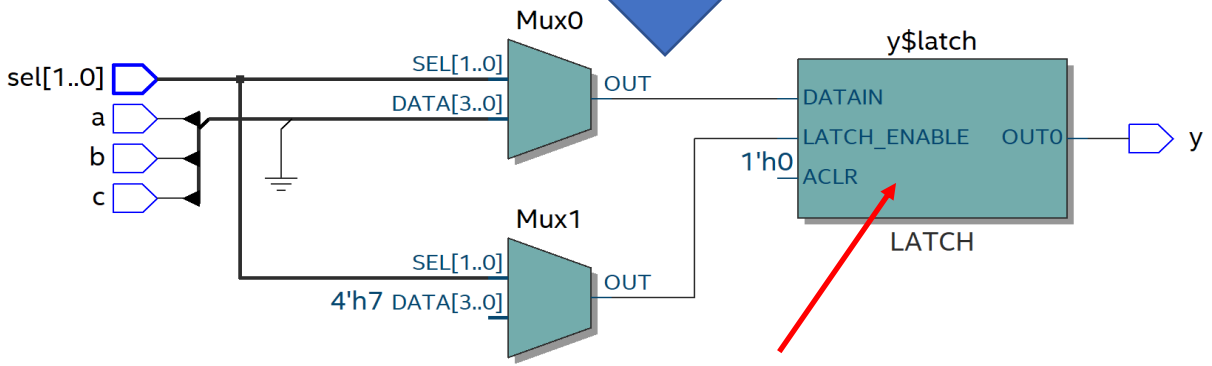
Always block : always_comb

- **always_comb** has several advantages over **always@* and always@**(<sensitivity_list>) :
  - **always_comb** automatically executes once at time zero, whereas **always@* and always@(a,b)** waits until a change occurs on a signal in the inferred sensitivity list.
    - Hence outputs of the combinational logic procedure are updated to match the values of the inputs at the start of the simulation. This is not true with **always@***
  - **always_comb** is sensitive to changes within the contents of a function, whereas **always@*** is only sensitive to changes to the arguments of a function. Also **always@*** doesn't infer sensitivity from tasks.
  - **always_comb** enforces some coding rules to ensure proper combinational logic is modeled
    - Only one source can write to variables on the LHS of assignments within an **always_comb** procedure, whereas **always @*** permits multiple processes to write to the same variable.
      - This avoids multiple driver ahead of the time during RTL code development
    - **always_comb** will give compile time error when having incomplete case statement to avoid unintentional latch.
      - With **always @* and always@(a,b)** there might be warning from synthesis compiler however code will synthesis and will infer unintentional latch causing simulation vs synthesis mist-match **(see next slide)**

- **In SystemVerilog always_comb is a better version of always @* and it is a best practice to use always_comb for modeling combinational circuit !!**

Always block : always_comb

```
module mux_3x1(
 input logic a, b, c,
 input logic [1:0] sel,
 output logic y
);
always@(a, b, c, sel) begin
  case (sel)
    2'b00: y = a;
    2'b01: y = b;
    2'b10: y = c;
    // Missing case expression for 2'b11
  endcase
end
endmodule: mux_3x1
```



Synthesis compiler infers hardware and does not generate error. Unwanted latch created at output 'y' by Synthesis compiler due to missing case expression.

```
module mux_3x1(
 input logic a, b, c,
 input logic [1:0] sel,
 output logic y
);
always_comb begin
  case (sel)
    2'b00: y = a;
    2'b01: y = b;
    2'b10: y = c;
    // Missing case expression for 2'b11
  endcase
end
endmodule: mux_3x1
```

Synthesis compiler throws error and synthesis process fails due to missing case item expression when using **always_comb** construct :

Warning : Incomplete case statement has no default case statement.

Warning : Inferring latches for variable "y", which holds is previous value in one or more paths through always constuct

SystemVerilog RTL Coding Error : always_comb construct does not infer purely combination logic !

Always block : always_ff

- **always_ff** procedure is used to model sequential flip-flop logic
  - **always_ff** differs from **always_comb**; in **always_ff** sensitivity list must be specified by designer
    - This is required since synthesis and compiler tools cannot infer the clock name and edge automatically from the body of **always_ff**
    - Synthesis and compiler tools does not know whether a reset is asynchronous or synchronous. If asynchronous then reset information is required to be specified in sensitivity list

  - **Example Syntax :**

```
always_ff@(posedge clk)
  q<=d

always_ff@(posedge clock or posedge reset) //both "or" and "," as a separator are allowed
begin
  if(reset) out <= 0;
  else  out <= out + 1;
end
```

Always block : always_ff

- **always_ff** enforces many of the synthesis requirements for RTL sequential logic coding :
  - Sensitivity list must specify either **posedge** or **negedge** of a clock required to update state of flip-flop
  - Sensitivity list must specify **posedge** or **negedge** of any asynchronous set or reset signals
  - Mixing of single edge and double edge expressions are not allowed within sensitivity list
  - Other than clock, asynchronous set/reset signals, sensitivity list cannot contain any other signals such as D input or an enable input.
  - Variables written on the left-hand side of assignments within **always_ff** procedure cannot be assigned by any other procedure or continuous assignment statement
  - Cannot mix blocking and non-blocking assignments to a same variable within **always_ff**

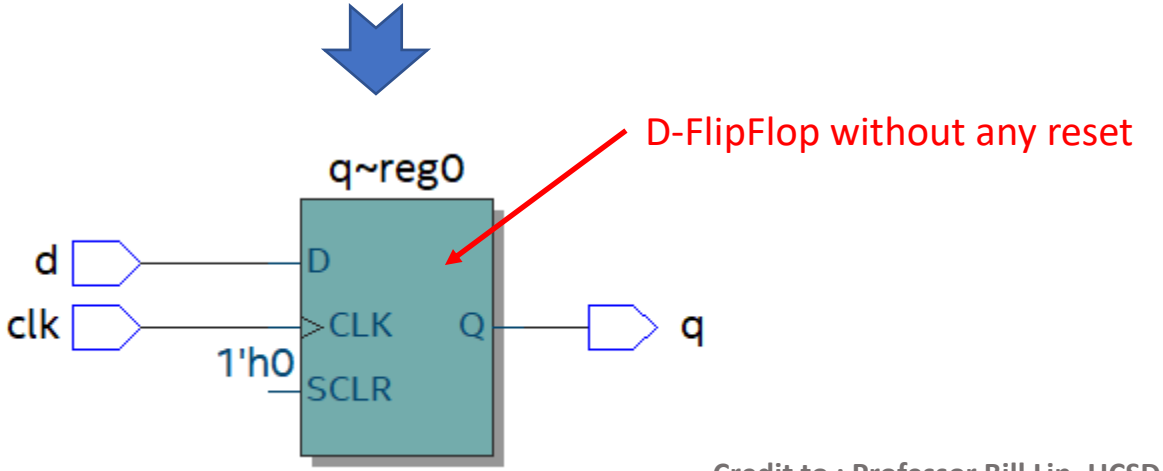
- **Violation of any rules mentioned above while modeling sequential logic, there will be syntax error from synthesis compiler**

D-FlipFlop Model Without Reset

```
module dff(
 input logic clk, d,
 output logic q
);

always_ff@(posedge clk) begin
  q <= d;  // 'q' gets 'd'
end
endmodule: dff
```

No reset signal specified

- SystemVerilog calls "**<=**" a "**non-blocking**" assignment.
- It means "**wait until next positive edge of clk**" **before** updating "q"
- This is why synthesis will produce a positive edge-trigged D-FF
- "always_ff" indicates that this is a "clocked always statement



D-FlipFlop Model with Reset using always_ff

D-FlipFlop with Synchronous Active High Reset

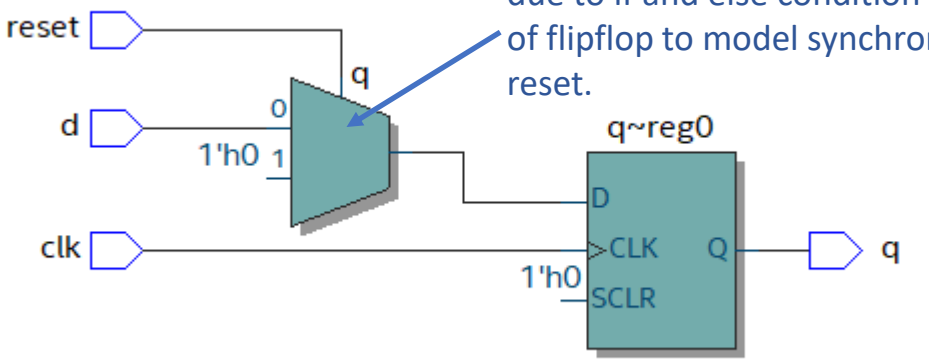
```
module dff(
 input logic clk, d, reset,
 output logic q
);
always_ff@(posedge clk) begin
if(reset)
  q <= 0;
else
  q <= d;
end
endmodule: dff
```

For synchronous reset, "reset" signal should ***not*** be specified in **always_ff** sensitivity list

reset signal is checked only on each positive edge of the clock, hence it is synchronous to "clk"

non-blocking assignment with clock edge sensitivity will infer a flipflop

mux is inferred before D input due to if and else condition logic of flipflop to model synchronous reset.

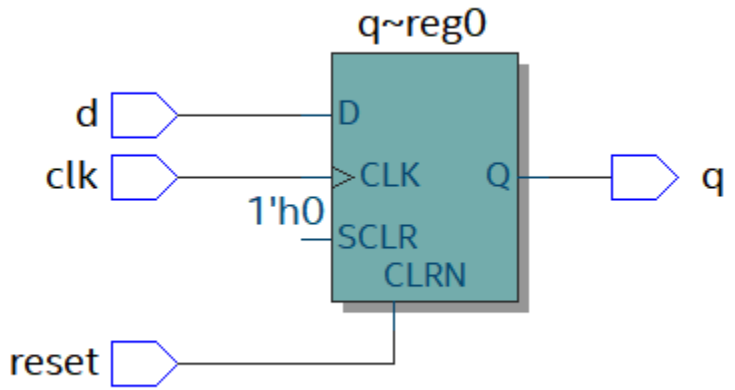


D-FlipFlop with Asynchronous positive Edge Reset

```
module dff(
 input logic clk, d, reset,
 output logic q
);
always_ff@(posedge clk, posedge reset) begin
 if(reset)
  q <= 0;
 else
   q <= d;
end
endmodule: dff
```

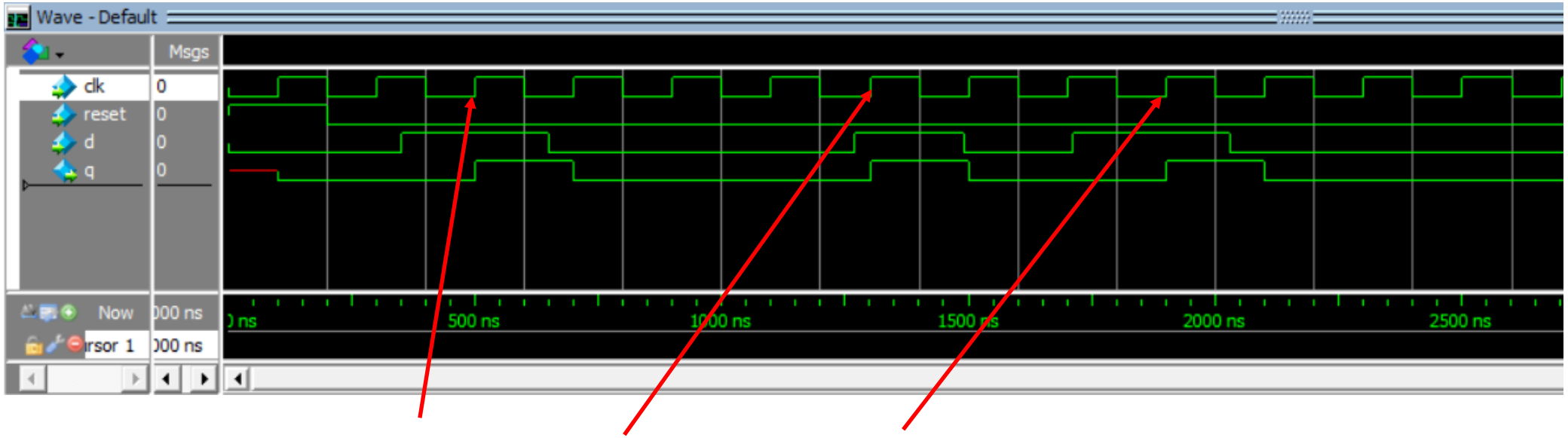
- Having reset in sensitivity list indicates to synthesizer to create asynchronous reset.
- Change in reset from '0' to '1' anytime will cause procedural statements within always_ff block to evaluate

if condition must have matching polarity of reset signal, since posedge is specified before "reset" signal in always_ff sensitivity list



No Mux inferred before D input of flipflop for asynchronous reset

Simulation Result for D-Flipflop with Synchronous Reset



Input d = 1 is sampled on this positive edge of rising clock and propagates to output ‘q’ and ouput q=1 is retained for the entire clock cycle

Mixing Single and Double Edge in Sensitivity List

D-FlipFlop with Asynchronous Reset

```
module dff1(
 input logic clk, d, reset,
 output logic q
);
always_ff@(posedge clk, reset)
begin
  if(reset)
    q <= 0;
  else
    q <= d;
end
endmodule: dff1
```

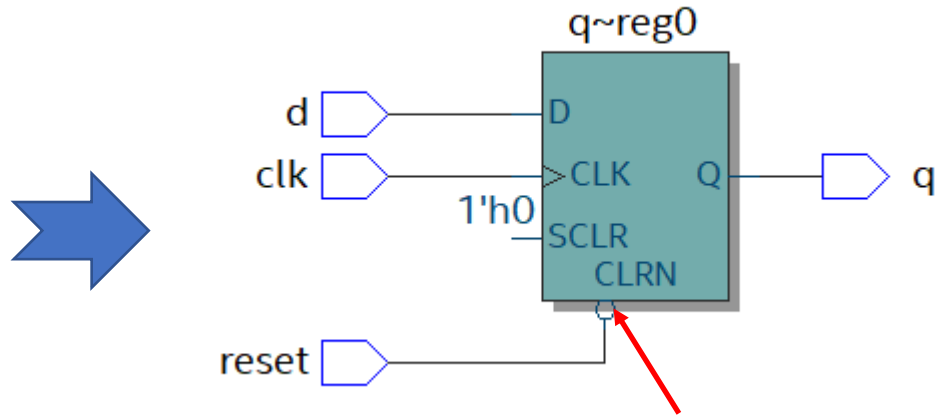
Mixing single and double edge in sensitivity list is not allowed and Code will not Synthesis

```
10122 Verilog HDL Event Control error at dff1.v(5): mixed single- and double-edge expressions are not supported
10235 Verilog HDL Always Construct warning at dff1.v(9): variable "d" is read inside the Always Construct but isn't in the Always Construct's Event Control
12153 Can't elaborate top-level user hierarchy
      Quartus Prime Analysis & Synthesis was unsuccessful. 2 errors, 2 warnings
```

Resettable D-FlipFlop Model

D-FlipFlop with Asynchronous Negedge Reset

```
module dff(
 input logic clk, d, reset,
 output logic q
);
always_ff@(posedge clk, negedge reset) begin
  if(!reset)
    q <= 0;
  else
    q <= d;
end
endmodule: dff
```



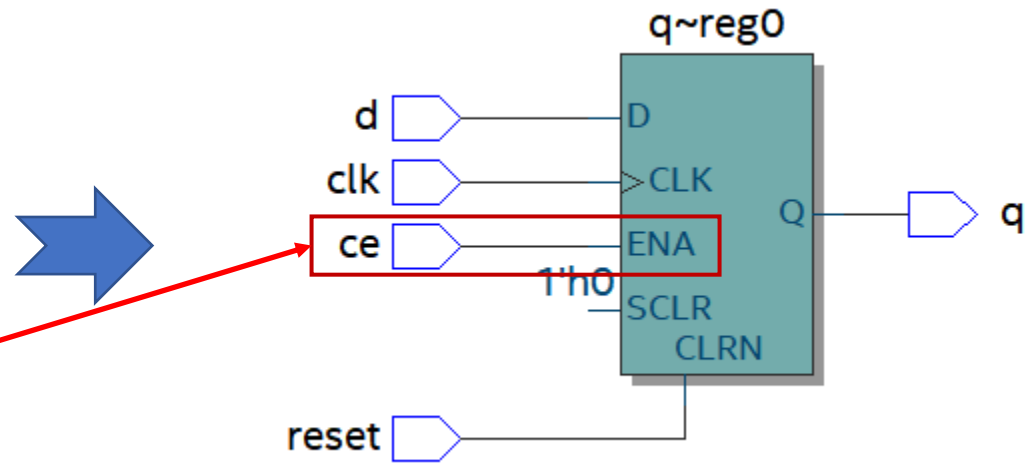
Inverter is inferred by Synthesizer due to negative edge reset mentioned in D-Flipflop model

- By specifying the negedge reset signal here, synthesis tools will infer asynchronous resettable D-FFs
- Due to **negedge** specified before signal "reset" in sensitivity list, if condition should have "!" before "reset" signal, otherwise synthesizer will give error.

D-FlipFlop Model with clock enable using always_ff

D-FlipFlop with Clock Enable and Asynchronous Reset

```
module dff(
 input logic clk, d, reset, ce,
 output logic q
);
always_ff@(posedge clk, posedge reset)
begin
  if(reset)
    q <= 0;
  else
    if(ce) // synchronous clock enable
      q <= d;
end
endmodule: dff
```



Dissimilar D-FlipFlops (Bad SystemVerilog Coding)

Synchronous Reset D-FlipFlop with Non-Reset Follower FlipFlop

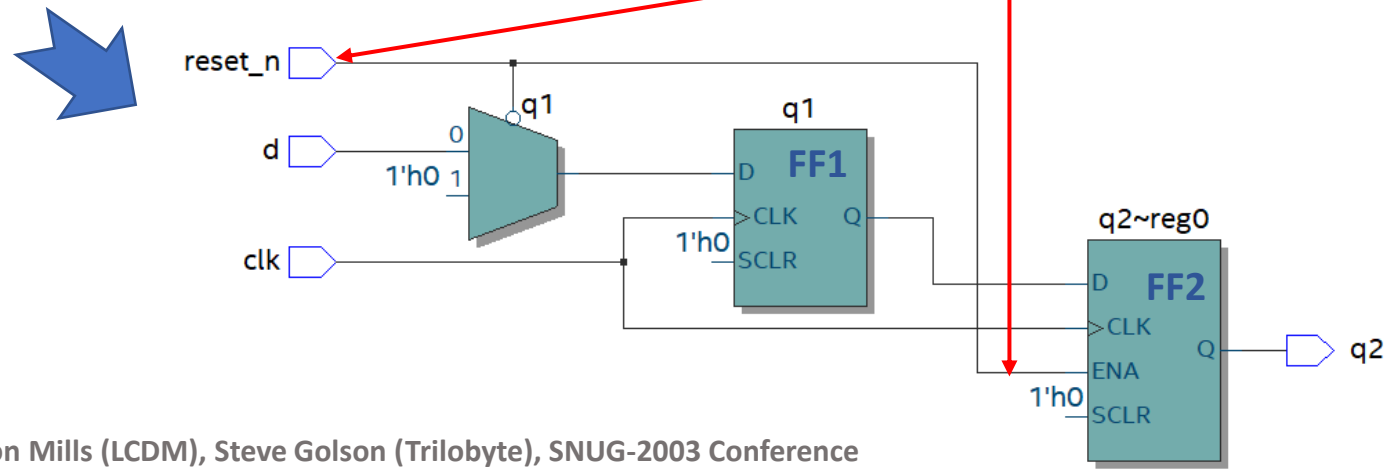
```
module bad_FF_coding(
 input logic clk, d, reset_n,
 output logic q2);
logic q1;
always_ff@(posedge clk) begin
  if(!reset_n)
    q1 <= 1'b0;
  else begin
    q1 <= d;
    q2 <= q1;
  end
end
endmodule: bad_FF_coding
```

Creates synchronous reset for first stage Flipflop FF1

Creates first stage flipflop FF1 with Synchronous reset

Creates second stage flipflop FF2 with no reset since q2<=0 is not mentioned under if(!reset_n)

Since the two flip-flops (FF1 and FF2) were inferred in the same procedural always_ff block and second flip-flop FF2 is not resettable, the reset signal reset_n will be used as a data enable for the second flop FF2



Dissimilar D-FlipFlops (Good SystemVerilog Coding)

Synchronous Reset D-FlipFlop with Non-Reset Follower FlipFlop

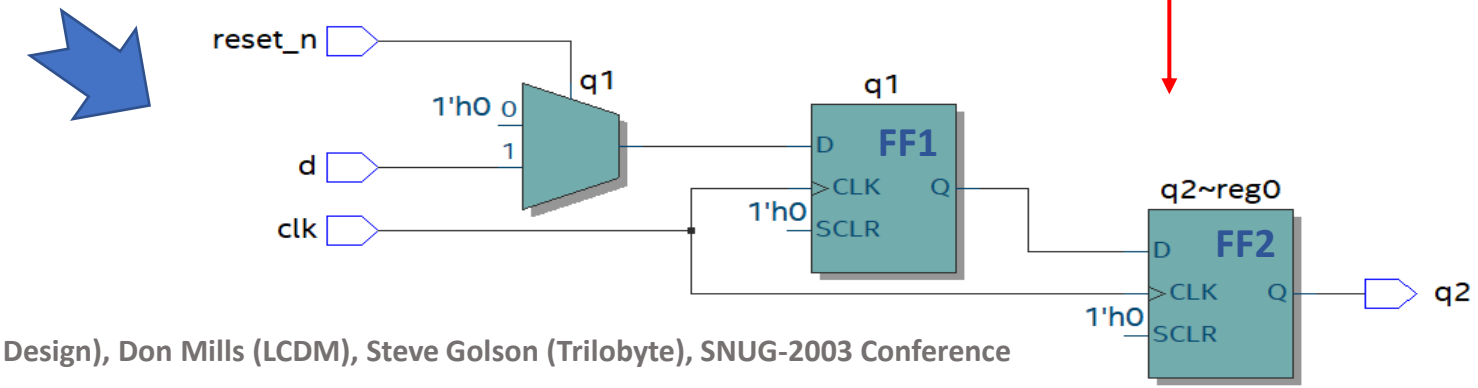
```
module good_FF_coding(
 input logic clk, d, reset_n,
 output logic q2);
logic q1;
always_ff@(posedge clk) begin
  if(!reset_n)
    q1 <= 1'b0;
  else
    q1 <= d;
  end
end
always_ff@(posedge clk) begin
   q2 <= q1;
end
endmodule: good_FF_coding
```

Creates synchronous reset for first stage Flipflop FF1

Creates first stage flipflop FF1 with Synchronous reset

Creates second stage flipflop FF2 with no reset

Since the two flip-flops (FF1 and FF2) were inferred in different procedural always_ff, the reset signal reset_n will not be used as a data enable for the second flop FF2



Credit : Clifford Cummings (Sunburst Design), Don Mills (LCDM), Steve Golson (Trilobyte), SNUG-2003 Conference

Multiple assignment statement on same variable

Multiple assignments on same variable

```
module adder(
 input logic clk, add1, add2,
 output logic result
);
always_ff@(posedge clk) begin
  if(add1) result <= result + 1;
  if(add2) result <= result + 2;
end
endmodule: adder
```

If add1 and add2 both are ‘1’ then result is assigned result+1 and result+2 simultaneously due to non-blocking assignment which is ambiguous.
Code will synthesis however circuit will not function correctly

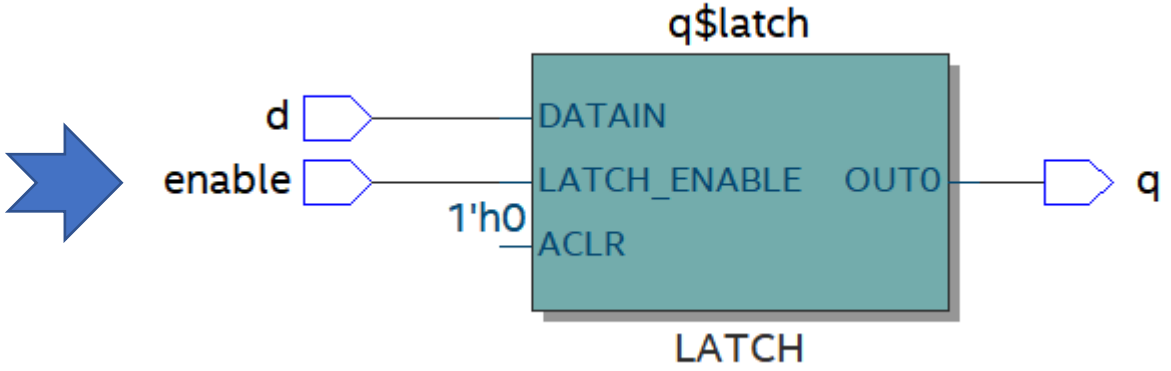
Always block : always_latch

- **always_latch** is introduced in SystemVerilog to model latch logic.
  - Similar to **always_comb,** all signals read within **always_latch** block are automatically inferred in sensitivity list. Including in case of function calls with partial list of signals in argument list.
  - **always_latch** indicates to EDA tools, that designer of RTL code intents to model a latch

- **always_latch** enforces some of the coding guidelines for synthesis
  - Any constructs such as #,@ and wait, which delays execution of statements are not permitted within always_latch
  - Variables assigned in always_latch cannot be assigned by any other procedure or continuous assignment statement.

```
module latch(
 input logic d, enable,
 output logic q
);
always_latch
begin
  if(enable)
    q <= d;
end
endmodule: latch
```

Inputs ‘d’ and ‘enable’ automatically inferred in sensitivity list by always_ff



Always block : always_latch

- Synthesis compiler will infer a latch logic in all three implementations shown below

Approach-A : Latch Model using always@(<explicit sensitivity list>)

```
module latch(
 input logic d, enable,
 output logic q
);
always@(d, enable) begin
  if(enable)
    q <= d;
end
endmodule: latch
```

Missing **else** branch will result in latch inference

Approach-B: Latch Model using always@(*) auto sensitivity list

```
module latch(
 input logic d, enable,
 output logic q
);
always@(*) begin
  if(enable)
    q <= d;
end
endmodule: latch
```

Missing **else** branch will result in latch inference

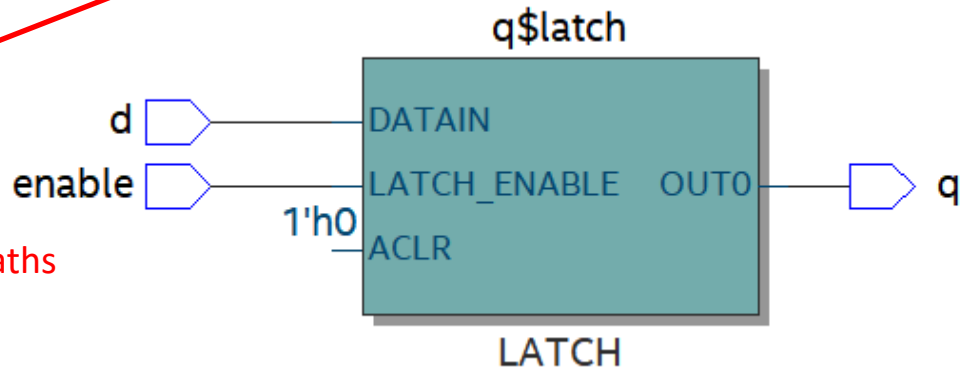
Approach-C Latch Model using always_latch auto sensitivity list

```
module latch(
 input logic d, enable,
 output logic q
);
always_latch begin
  if(enable)
    q <= d;
end
endmodule: latch
```

Best Practice

Synthesis compiler throws warning for potential unitential latch for designer to confirm

Warning (10240): Verilog HDL Always Construct warning at latch.v(6) : inferring latch(es) for variable "q", which holds its previous value in one or more paths through the always construct

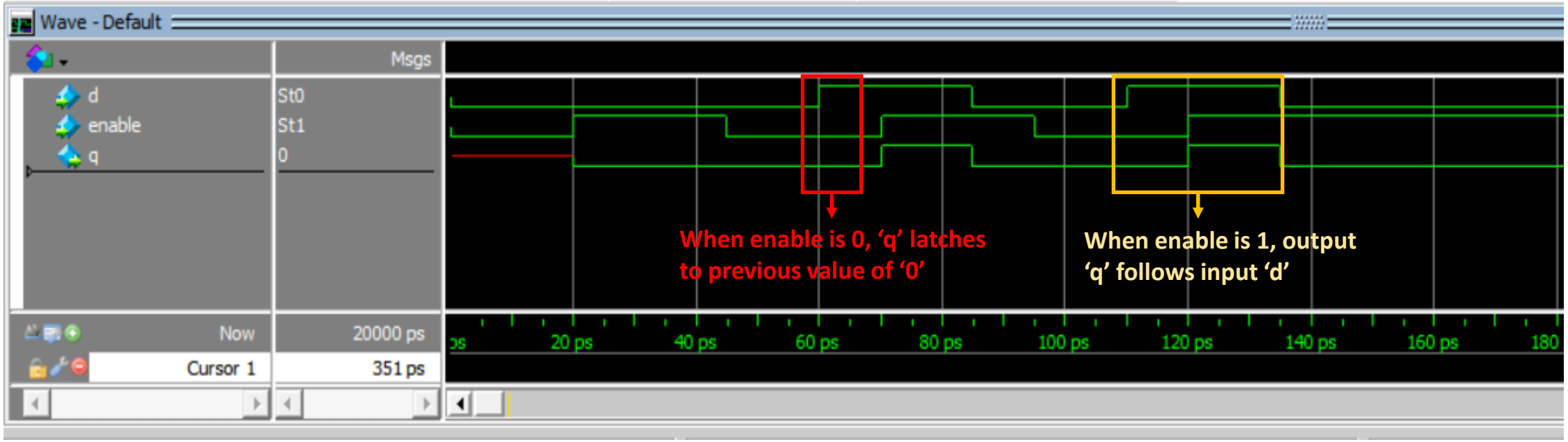


Synthesis compiler does not generate any warning message since always_ff instructs synthesis compiler to infer an intentional latch. Instead generates a info message

Info (10041): Inferred latch for "q" at latch.v(6)

Latch Model Simulation Result

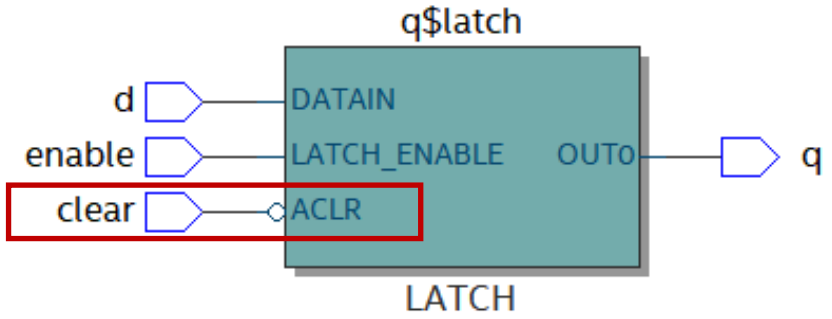
- Same simulation result for latch RTL modeled with **Approach-A, Approach-B** and **Approach-C**



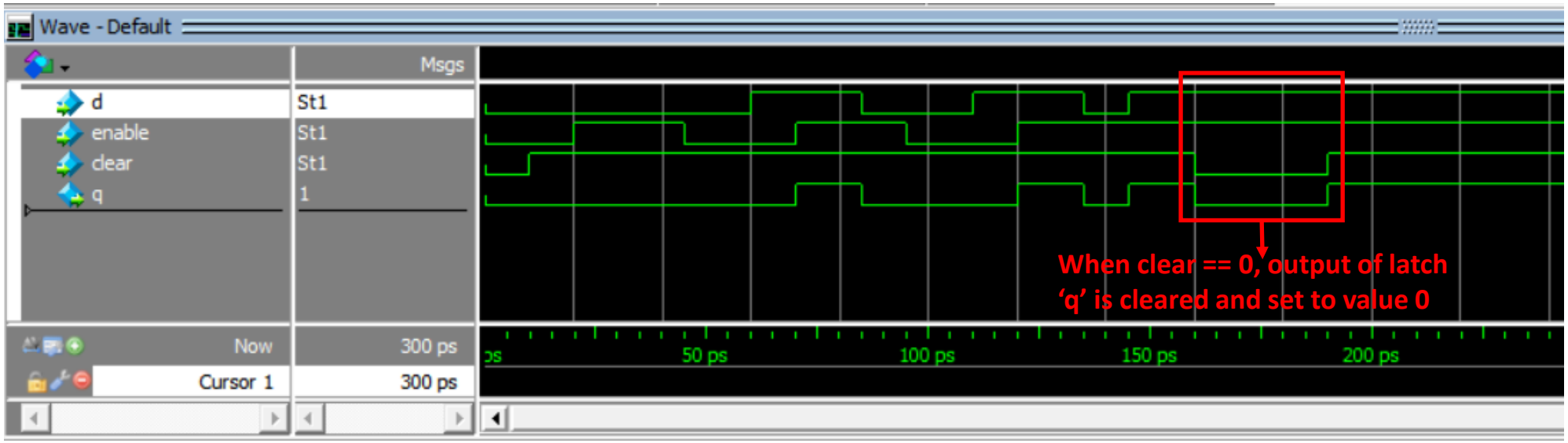
Latch with Asynchronous Clear

- RTL model of latch with asynchronous negedge clear using **always_latch**

```
module latch(
 input logic d, enable, clear,
 output logic q
);
always_latch begin
  if(!clear) // asynchronous !clear
    q <= 0;
  else if(enable)
    q <= d;
end
endmodule: latch
```



Simulation result of latch with asynchronous clear



Latch Model with Function Call

- Function calls within **always_latch** does not have to have all input signals as part of the argument list, since all required input signals are inferred in sensitivity list.
- However with **always@(*)** latch implementation, explicitly designer has to list down all required input signals as arguments in function call for function to trigger when input changes.
- Both implementations will infer a latch when synthesizing the logic, however simulation behavior will be different with **always_latch** vs **always@(*)** implementations.

Latch RTL model using always@(*)

```
module latch(
 input logic d, enable,
 output logic q
);

function logic func_latch();
  func_latch = d;
endfunction

always@(*) begin
  if(enable)
    q <= func_latch();
end
endmodule: latch
```

func_latch will not trigger when there is change in value of "d" and when enable is '1' but enable value did not change

Same synthesis results. (latch will be inferred)

=

Different simulation results

≠

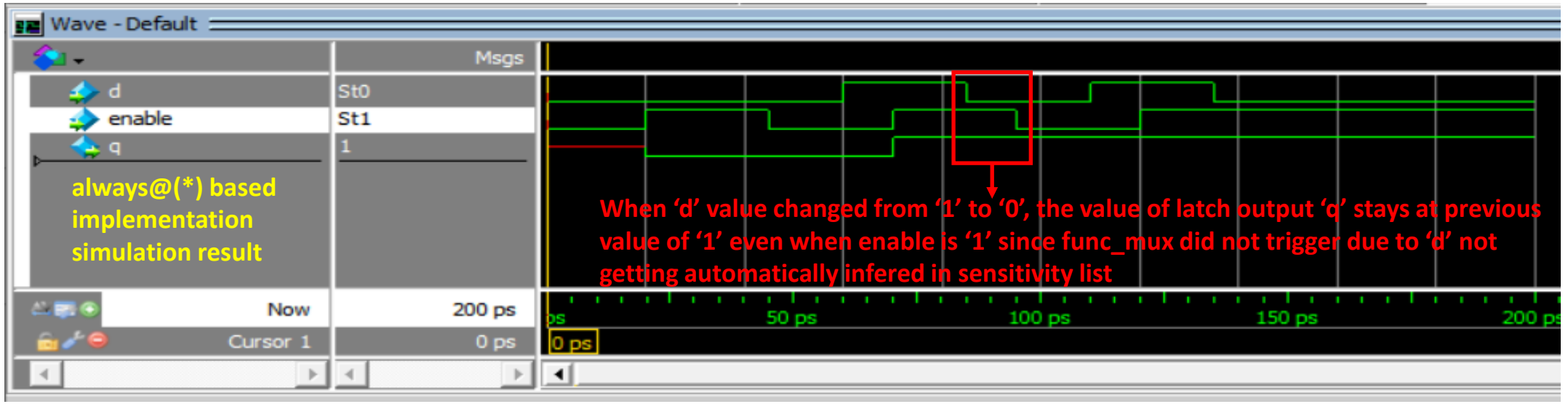
Latch RTL model using always_latch

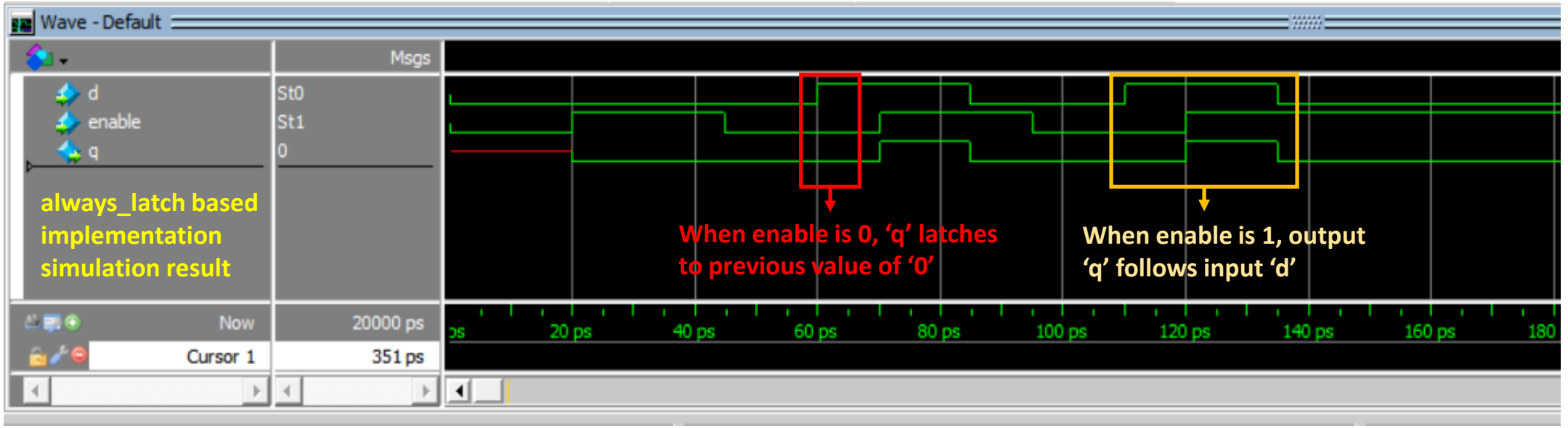
```
module latch(
 input logic d, enable,
 output logic q
);

function logic func_latch();
  func_latch = d;
endfunction

always_latch begin
  if(enable)
    q <= func_latch();
end
endmodule: latch
```

func_latch will be triggered whenever there is change in value of "d" and when enable is '1' but enable value did not change

Latch Model Simulation Result

- **always@(*)** vs **always_latch** simulation result of RTL model of latch with function call





Summary on always Procedures

- **Verilog-2001 supported below mentioned always procedures :**
  - **always@(<explicit sensitivity list>)** to model combinational and sequential logic
  - **always@(*)** to model combinational and sequential logic

  Note : Avoid Verilog **always@(*)** usage !!

- **SystemVerilog introduced three more always procedures which brings some enhanced capabilities and addressed ambiguity in Verilog Language :**
  - **always_comb** to model combinational logic
  - **always_latch** to model latch
  - **always_ff@(<explicity sensitivity list>)** to model sequential logic

- **SystemVerilog always procedures addressed some of the limitations of Verilog-2001 always procedures and clearly conveys design intent to EDA tools**
  - **Hence suggestion is to use SystemVerilog always procedures wherever possible in RTL model !!**

Summary on always Procedures

- **always_comb** and **always_latch** will execute at time zero of the simulation, ensuring the variables on the left-hand side of assignments within the block correctly reflect the values on the right and side at time 0.

- **always_comb** and **always_latch** are sensitive to signal changes within a function called by the procedural block, and not just the function arguments, which was a bug with **always @(*)**

- **always_latch** and **always_ff** procedural blocks are ***huge***! They can prevent serious modeling errors, and they enable software tools to verify that design intent has been met.

- ***Recommendation —*** Use **always_comb**, **always_latch** and **always_ff** in all RTL code.
  - Only use the general purpose **always** procedure in models that are not intended to be synthesized, such as bus-functional models, abstract RAM models, and verification testbenches.

RTL model for 4-bit counter using always_ff

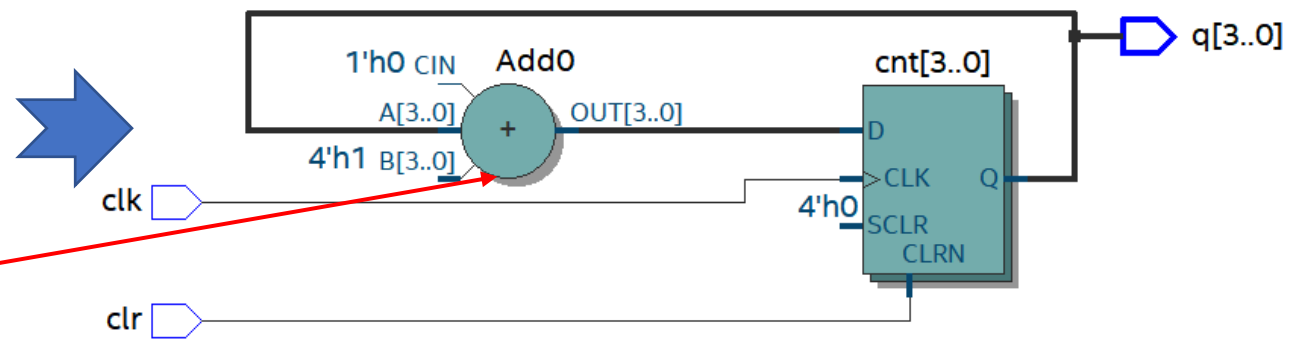
N-bit unsigned up counter with asynchronous clear

```
module counter #(parameter N=4)
(
  input   logic clk, clr,
  output logic[N-1:0] q
);
  logic[N-1:0] cnt;
  always_ff@(posedge clk or posedge clr)
  begin
   if(clr)
     cnt <= 'b0;
   else
     cnt <= cnt + 1'b1;
  end
  assign q = cnt;
endmodule: counter
```

will infer asynchronous clear

'+' operator will infer an adder

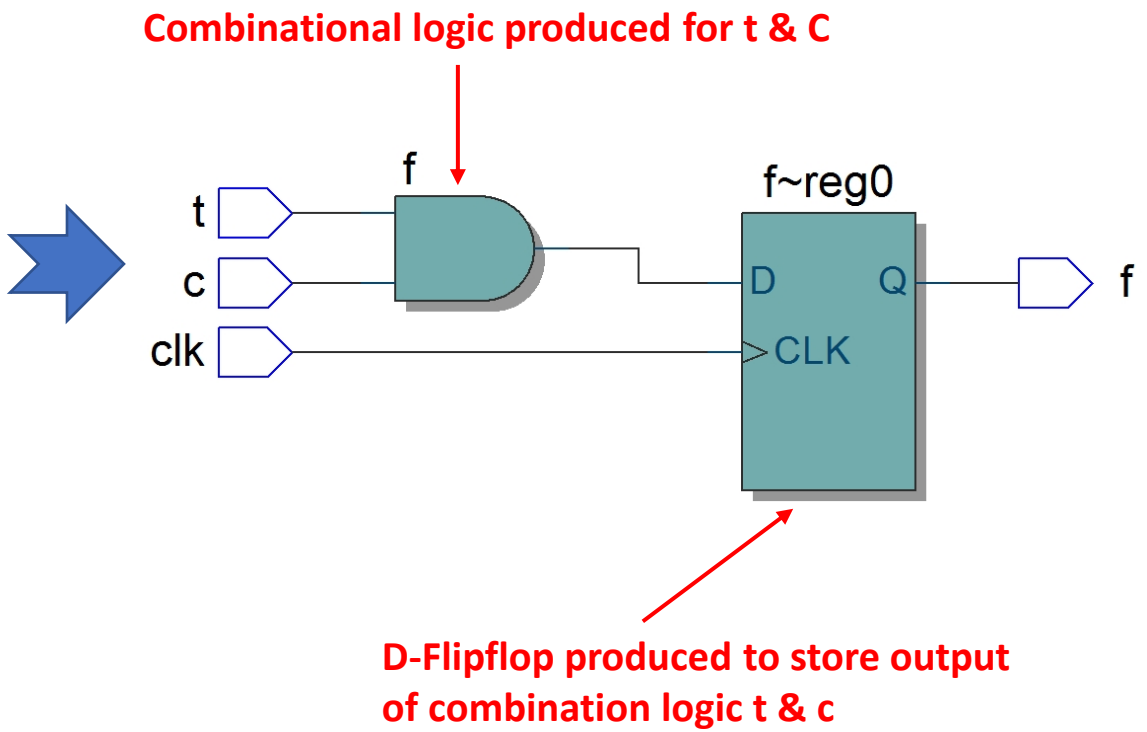
Feedback from q to input of adder is created due to counter incrementing from the previous output value



Creating Combinational Logic and D-flipflop

```
module ex1(
  input logic clk,
  input logic t, c,
  output logic f
);

  always_ff@(posedge clk)
  begin
    f <= t & c;
  end
endmodule: ex1
```

- Positive edge triggered D-Flipflop produced for "f"
- "And" gate produced for t & C



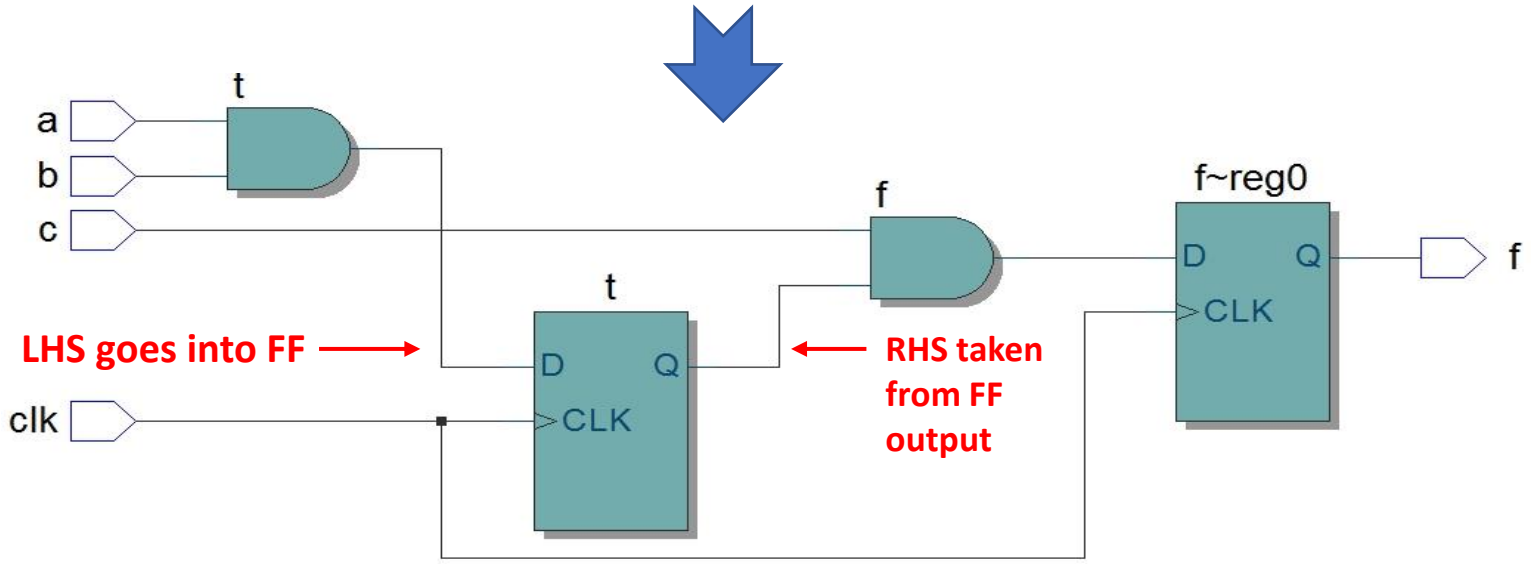
Clocked always_ff Block

```
module ex2(
  input logic clk,
  input logic a, b, c,
  output logic f);

  logic t;
  always_ff@(posedge clk) begin
    t <= a & b;
    f <= t & c;
  end
endmodule: ex2
```

D-FF produced for "t" and "f"

LHS stores output of AND-gate to "t" register

RHS reads from "t" register



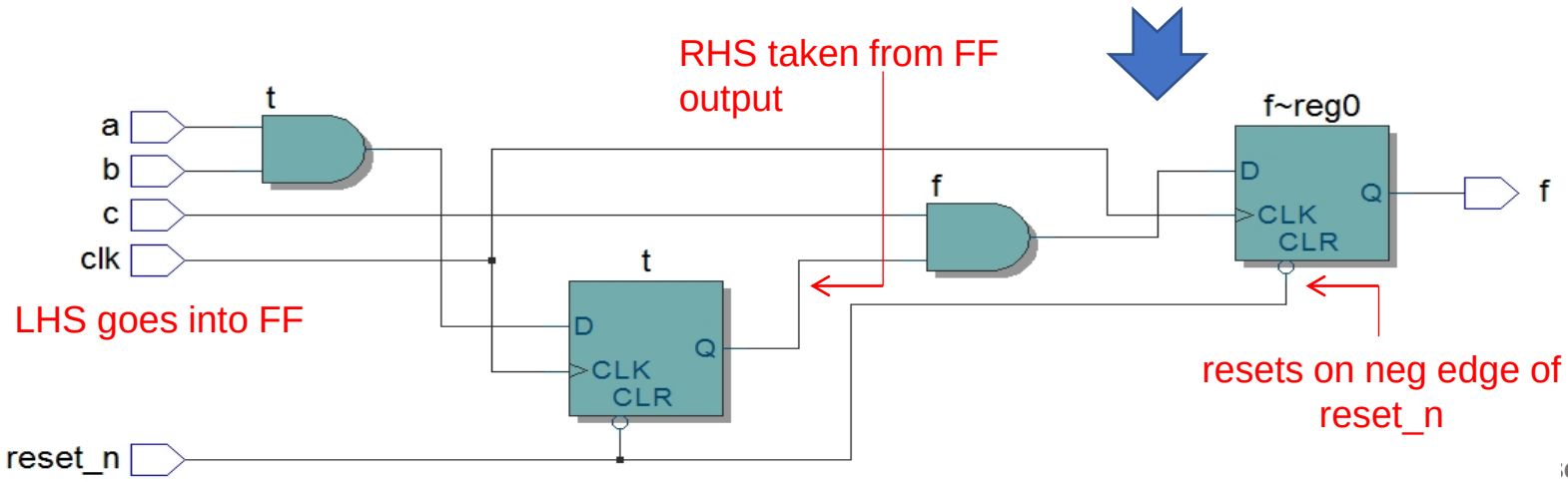
Credit to : Professor Bill Lin, UCSD, ECE Department

Clocked always_ff Block

```
module ex3(
  input logic clk, reset_n, a, b, c,
  output logic f);
  logic t;
  always_ff@(posedge clk, negedge reset_n) begin
    if(!reset_n) begin
      t <= 0;
      f <= 0;
    end else begin
      t <= a & b;
      f <= t & c;
    end
  end
endmodule: ex3
```

this **if** part specifies how registers should be initialized

- Template specifies async reset edge-triggered D-FFs
- **Positive** edge-triggered on “clk”
- Async reset **negative** edge on “reset_n”



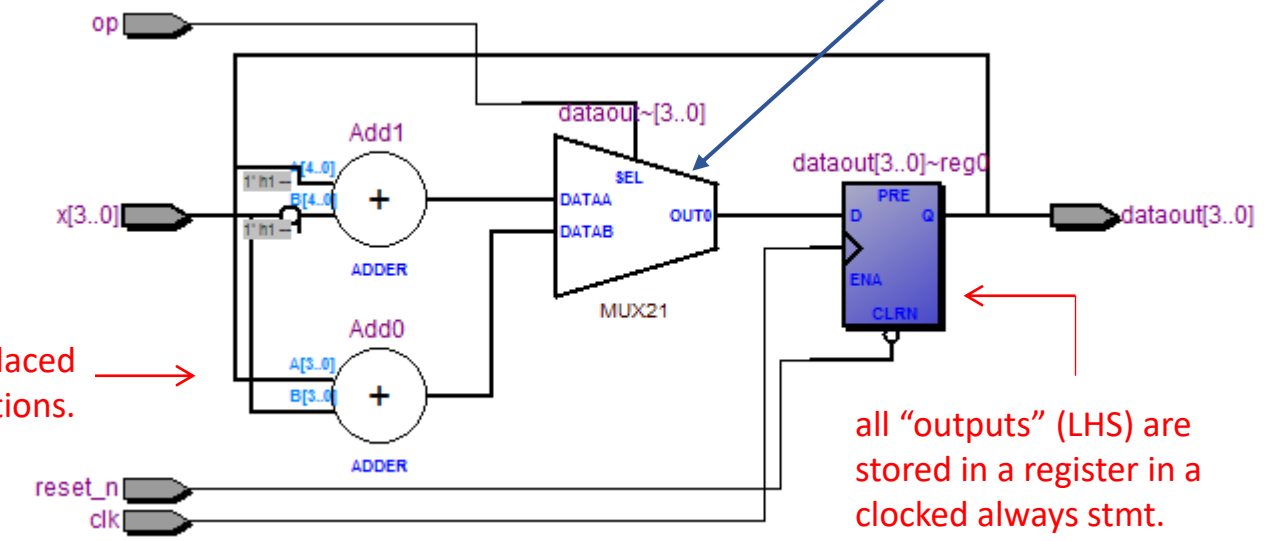
Clocked always_ff Statement

```
module ex4 (
  input logic clk, reset_n, op,
  input logic[3:0] x,
  output logic[3:0] dataout);
  always_ff@(posedge clk, negedge reset_n) begin
  if(!reset_n)
    dataout <= 4'b0000;
  else
    if (op)
      dataout <= dataout + x;
    else
      dataout <= dataout - x;
  end
endmodule: ex4
```

this **if** part specifies how registers should be initialized

- this **else** part specifies how registers should be updated at each cycle.
- all "outputs" (LHS) are stored in a register in a clocked always stmt. LHS specifies "**new**" value that will be stored after next clk tick.
- should use "**<=**" instead of "**=**", which means on the RHS, the value is the "**current**" value of the register

If-then-else & case stmts translated as MUXes



Arithmetic operators (e.g. "+") are replaced with library logic implementations.

all "outputs" (LHS) are stored in a register in a clocked always stmt.

Multiple always_ff Clocked Statement

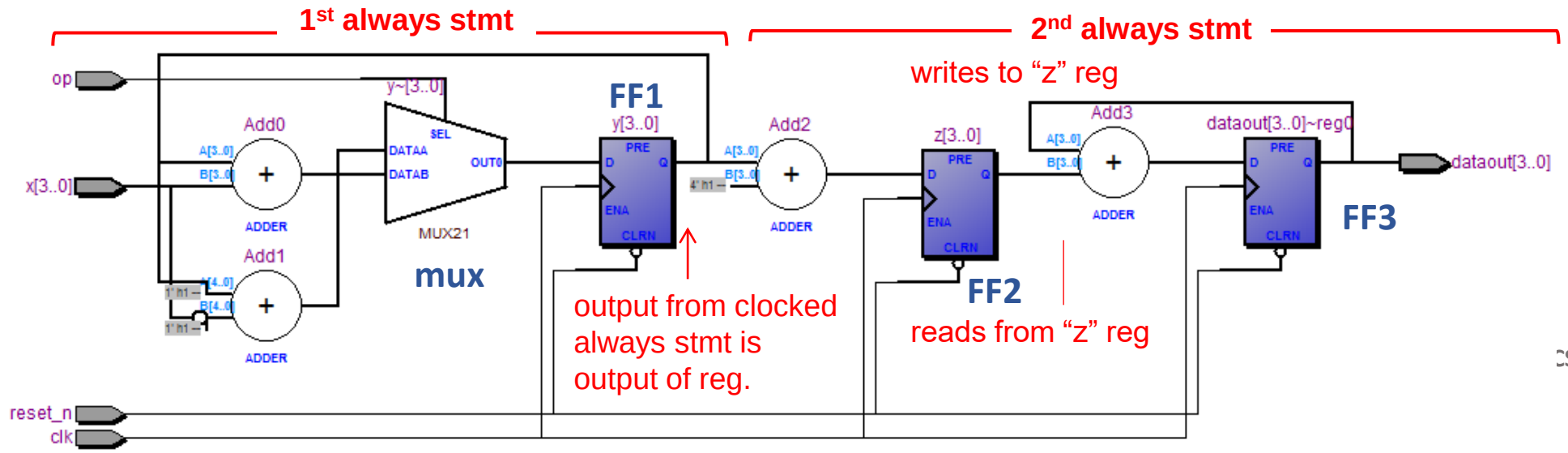
```
module ex5 (
 input logic clk, reset_n, op,
 input logic[3:0] x,
 output logic[3:0] dataout);
 logic[3:0] y, z;
 always_ff@(posedge clk, negedge reset_n) begin
 if(!reset_n)
  y <= 4'b0000;
 else
  if (op)
   y <= y + x;
  else
   y <= y - x;
 end
```

- this nested **if-else** produced mux
- And non-blocking assignments inside Clocked always_ff produced FF1

Both always_ff procedural blocks will run concurrently at any given time steps

```
always_ff@(posedge clk, negedge reset_n) begin
 if(!reset_n) begin
  dataout <= 4'b0000;
  z <= 4'b0000;
 end else begin
  z <= y + 1;
  dataout <= dataout + z;
 end
endmodule: ex5
```

- This logic produced counter with output stored in flipflop FF3



Mixing Procedural Blocks and Continuous Assignment Statements

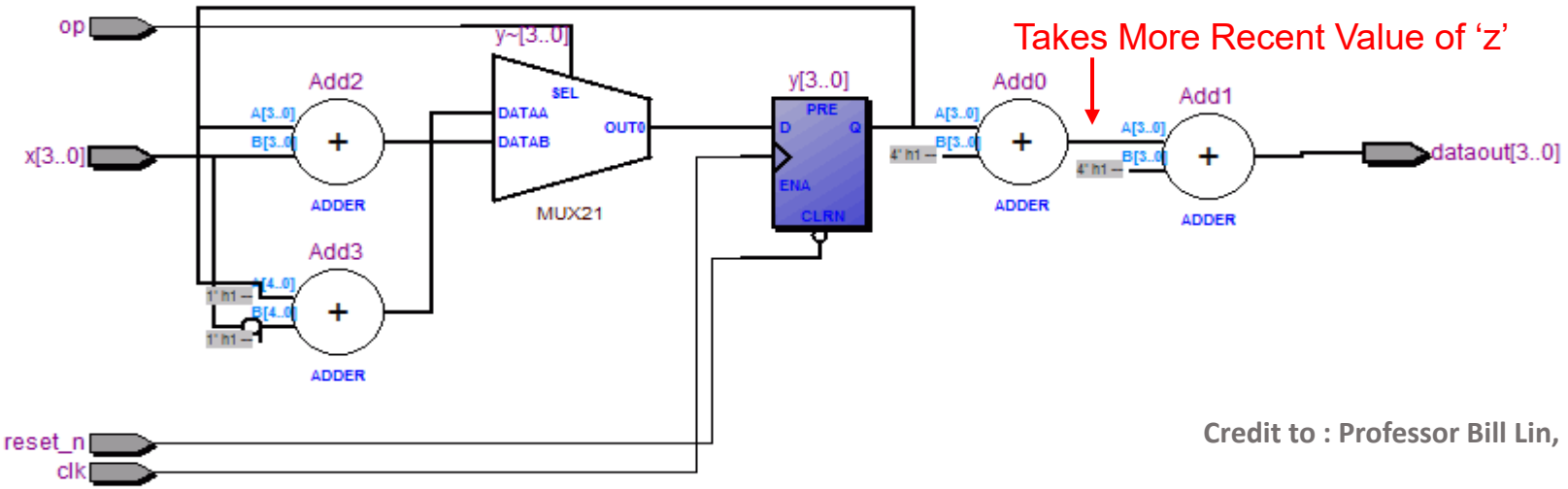
```
module ex6 (input logic clk, reset_n, op,
  input logic [3:0] x,
  output logic [3:0] dataout);
  logic [3:0] y, z;

always_ff@(posedge clk, negedge reset_n) begin
 if (!reset_n)
  y <= 4'b0000;
 else
  if (op)
    y <= y + x;
  else
    y <= y - x;
 end
```

always_ff and all continuous assignments runs concurrently at any given time steps

```
 assign z = y + 1;
 assign dataout = z + 1;
endmodule: ex6
```

always_ff statement

Two continuous assign statements



Mixing Procedural Blocks and Continuous Assignment Statements

```
module ex7 (input logic clk, reset_n, op,
  input logic [3:0] x,
  output logic [3:0] dataout);
  logic [3:0] y, z;

always_ff@(posedge clk, negedge reset_n) begin
 if (!reset_n)
  y <= 4'b0000;
 else
  if (op)
    y <= y + x;
  else
    y <= y - x;
end
```

same behavior as ex6, but using always_comb instead of 2 assign statements

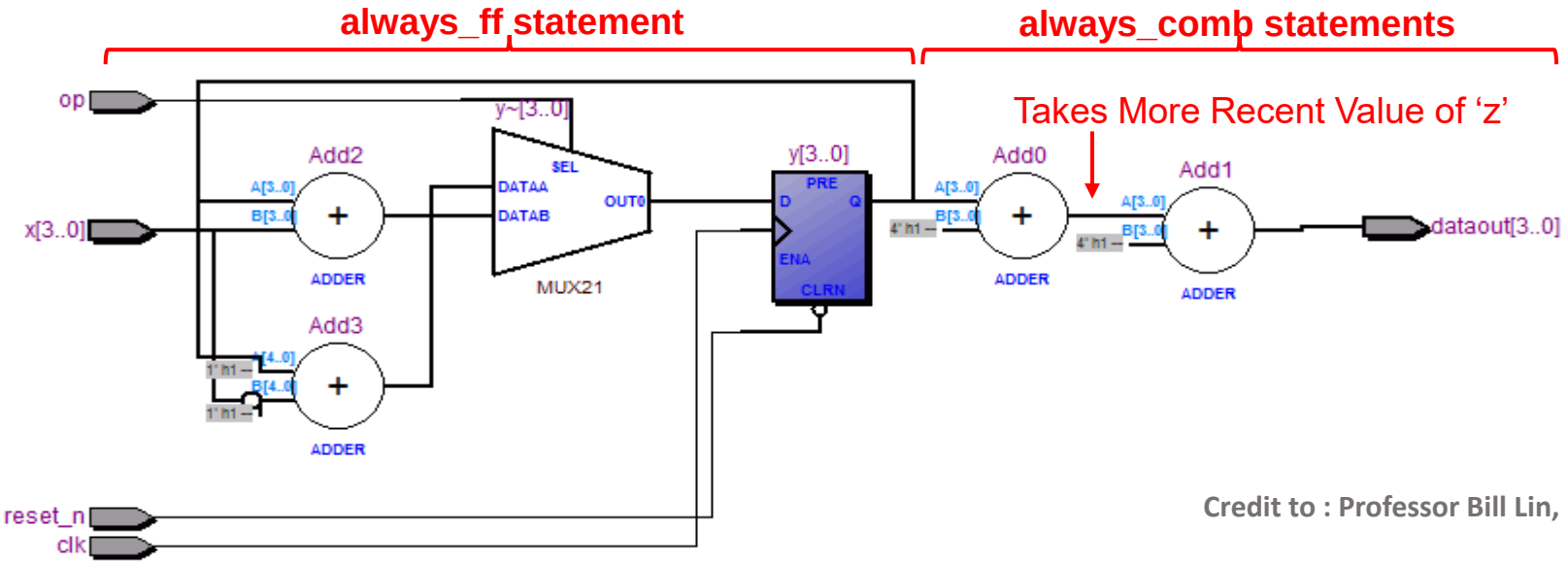
Always_comb representation is similar to having 2 continuous assignments

assign z = y + 1;

assign dataout = z + 1;

```
always_comb
begin
  z = y + 1;
 dataout = z + 1;
end
endmodule: ex7
```

should use "**=**" in comb. logic always stmts. no reg. RHS just takes output from the previous eqn.



Initial block

- **Initial block contains one or more programming statements and timing information to instruct simulators when to drive and which values to drive ports of design block**
  - It is specified using keyword **initial**
  - **begin** and **end** keywords must be used to specify multiple statements within **initial** block
  - **Syntax :**

```
initial
   <optional delay> <programming statement>
end
```

```
Initial begin
   <optional delay> <programming statement>
   ….
   ….
   <optional delay> <programming statement>
end
```

- **Initial block is used to develop testbench code to simulate design / RTL code :**
  - Create and specify how stimulus will be applied to input signals in design
  - Specifies initialization of local variables in testbench code and forcing of internal design signals
  - Specifies how the design signals are monitored and displayed
  - Specifies simulation pass and fail criteria and checking mechanism
  - Specifies the file where the signal waveform information is to be dumped

Initial Procedural block

- Initial block is executed at the beginning of the simulation at time 0

- Initial block is executed **only once** in simulation
  - Initial block finishes once all the statements within the block are executed and does not execute again for a given run of a simulation
  - Delay statements within initial block will advance simulation

```
module tesbench;
  logic reset, enable;
  initial begin
    reset = 1;
    enable = 0;
    #10ns reset = 0;
    enable = 1;
    #5ns;
    enable = 0;
  end
endmodule
```

initial block starts execution at time 0

- reset will get value 1 and enable will be get value 0 at time 0ns
- simulation will advance to 10ns and then reset will get new value 0 and enable is assigned new value 1;
- simulation will advance 5ns to reach at 15ns (10ns + 5ns) timestamp and then enable will be assigned new value 0

initial block terminates forever once enable is assigned value 0 at time 15ns

Multiple Initial blocks

- SystemVerilog module can have multiple initial blocks
  - All initial blocks executes concurrently and starts it execution at time 0

```
module multiple_initial_block_test;
 logic reset, enable;
 // initial block-1
 initial begin
   reset = 1;
   #20ns reset = 0;
 end
 // initial block-2
 initial begin
   enable = 0;
   #15ns enable = 1;
   #25ns enable = 0;
 end
 // initial block-3
 initial begin
   $monitor("time=%g ns, reset=%b, enable=%b\n", $time, reset, enable);
   #50ns;
   $display("time=%g ns, End of Simulation!\n", $time);
   $finish;
 end
endmodule
```

Each initial block starts execution at time 0 as three separate threads running concurrently

$monitor is a system task in SystemVerilog which executes whenever there is change in value of variables it is monitoring, in this example each time reset or enable value changes $monitor task will be called

$finish is a system task in SystemVerilog which terminates simulation

Simulation Result

- time=0ns, reset=1, enable=0
- time=15ns, reset=1, enable=1
- time=20ns, reset=0, enable=1
- time=40ns, reset=0, enable=0
- Time=50ns, End of Simulation!

All initial blocks executes only once during given simulation !!

Total Simulation Time = 50ns

- initial block-1 has total delay of 20ns
- initial block-2 has total delay of 35ns
  - block-2 (15ns + 25ns) = 35ns
- initial block-3 has total delay of 50ns

Since initial block-3 has longest delay of 50ns, hence simulation will run until 50ns and it will then terminate

Initial block – Pre-mature Termination !

❑ One initial block can pre-maturely terminate execution of other initial blocks !!

```
module multiple_initial_block_test;
  logic reset, enable;
  // initial block-1
  initial begin
    reset = 1;
    #20ns reset = 0;
  end
  // initial block-2
  initial begin
    enable = 0;
    #15ns enable = 1;
    #25ns enable = 0;
  end
  // initial block-3
  initial begin
    $monitor("time=%g ns, reset=%b, enable=%b\n", $time, reset, enable);
    #30ns;   // changed delay from 50ns to 30ns
    $display("time=%g ns, End of Simulation!\n", $time);
    $finish;
  end
endmodule
```

#25ns enable=0 which is scheduled for execution at simulation timestamp 40ns will not execute since initial block-3 will pre-maturely terminate initial block-2 and entire simulation due to $finish executed in initial block-3 at time 30ns.

Simulation Result

- time=0ns, reset=1, enable=0
- time=15ns, reset=1, enable=1
- time=20ns, reset=0, enable=1
- time=30ns, End of Simulation!

Note : simulation is not able to advance to time 40ns to drive enable = 0 from initial block-2 since $finish is encountered at time 30ns from initial block-3, which pre-maturely terminated initial block-2 !

Total Simulation Time = 30ns

- initial block-1 has total delay of 20ns
- initial block-2 has total delay of 35ns
  - block-2 (15ns + 25ns) = 35ns
- initial block-3 has total delay of 30ns

Since initial block-2 has longest delay of 35ns, however since initial block-3 has **$finish** system task, which executes at 30ns, simulation will terminate **at 30ns**

Initial block – Race Condition !

- Initial blocks can have race conditions since they are executing concurrently !!
  - there is no specific order in which statements across initial block executes at given time stamp
  - it is choice of a simulator to pick the order. Order of execution will impact output value.

```
module multiple_initial_block_with_race;
  logic reset, enable;
  // initial block-1
  initial begin
    reset = 1;
    #20ns reset = 0;
  end
  // initial block-2
  initial begin
    enable = 0;
    #20ns enable = reset;
  end
  // initial block-3
  initial begin
    $monitor("time=%g ns, reset=%b, enable=%b\n", $time, reset, enable);
    #30ns;
    $display("time=%g ns, End of Simulation!\n", $time);
    $finish;
  end
endmodule
```

There is a race condition between initial block-1 and initial block-2

- **@20ns from initial block-1 reset is getting assigned with new value 0**
- **@20ns from initial block-2 reset value is assigned to enable**

Which value of reset will be assigned to enable @20ns ?

- **Previous value 1 or new value 0 ?**
- **Based on order of execution picked by simulator, enable will have value 1 or 0**

Possible Simulation Result-1

- time=0ns, reset=1, enable=0
- time=20ns, reset=0, enable=0
- time=30ns, End of Simulation!

Note : if simulator decided to first execute statement (#20ns reset=0) from initial block-1 then enable will get assigned new value of reset which is 0 in initial block-2 at timestamp @20ns

Possible Simulation Result-2

- time=0ns, reset=1, enable=0
- time=20ns, reset=0, enable=1
- time=30ns, End of Simulation!

Note : if simulator decided to first execute statement (#20ns enable=reset) from initial block-2 then enable will get assigned old value of reset which is 1 in initial block-2 at timestamp @20ns.

Summary on initial and always procedure block

initial	always
❑ Not synthesizable and does not generate hardware logic	❑ Synthesizable and it can generate hardware logic
❑ Used for simulation purpose	❑ Used for specifying design behavior (RTL code)
❑ Starts execution from beginning of a simulation at time 0	❑ Starts execution from beginning of a simulation at time 0
❑ Executes only once during simulation and it terminates when all statements within it are executed	❑ Executes repeatedly. Its activity shall cease only when the simulation is terminated
❑ Any number of initial blocks can be defined within a module	❑ Any number of always blocks can be defined within a module
❑ Multiple initial blocks executes concurrently	❑ Multiple always blocks executes concurrently

- ❑ **There is no implied order of execution between initial and always constructs.**
  - **initial blocks are not necessarily required to be scheduled and executed before the always constructs**