Lecture 9: More Procedural Statements

UCSD ECE 111

Prof. Farinaz Koushanfar

Fall 2024

Some slides are courtesy of Prof. Lin or Prof. Karna ECE111 courses

Important announcements (Details in Canvas)

- **Next class on Tues 10/29 will be virtual – posted online (Not in person)**
- **Class on Thurs 10/31 will be in person as usual so please be here in the class**
- **Prof office hours next week:** Thurs 10/31 from 12-2pm or by email appointment
- **TA office hours:** are now MWF 9-11am for Fall'24
  - Zoom Meeting ID: 948 6397 0932; Passcode 004453
  - https://ucsd.zoom.us/j/94863970932?pwd=iXcQXbLYjOaAQTdOJlrmglCYMSyeir.1
- **TA Discussion session:** Wed 4-4:50PM (in person) Location: CNTRE 214
- **Oct 22:** Homework 4 was posted on Canvas
  - Due on Wed Oct 30, **10/30/24**
  - **Late homework policy:** Max of 2 late days, with 20% grade reduction per day

Homework 4 overview

- Design of Linear Feedback Shift Register (LFSR), Barrel Shifter, Gray to Binary Code Convertor
- You will learn how to:
  - Create synthesizable SystemVerilog code
  - Better learn how to use testbenches
  - Design functional SystemVerilog code that can compile post synthesis.
- There will be three parts for this homework:
  - **Homework-4a**: Developing a Synthesizable SystemVerilog Model for a Linear Feedback Shift Register (LFSR)
  - **Homework-4b**: Developing a synthesizable SystemVerilog model of a Barrel Shifter
  - **Homework-4c**: Developing a synthesizable SystemVerilog model of a Gray to Binary Code Convertor

Recap

always Procedural Blocks

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

Which Statements are True?

- always_comb does infer a complete sensitivity list when the always_comb block contains functions
- always_comb automatically executes once at time zero, whereas always@(*) waits until a change occurs on a signal in the inferred sensitivity list
- always@(*) may not infer a complete sensitivity list when the always @(*) block contains functions
- always_comb does infer a complete sensitivity list when the always_comb block contains functions

always_ff Block

- **always_ff** procedure is used to model **sequential flip-flop logic**

- In always_ff**, sensitivity list** must be specified by designer
  - This is because 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

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

```
always_ff@(posedge clock or posedge reset)
begin
    if (reset) out <= 0;
    else  out <= out + 1;
end
```

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
```

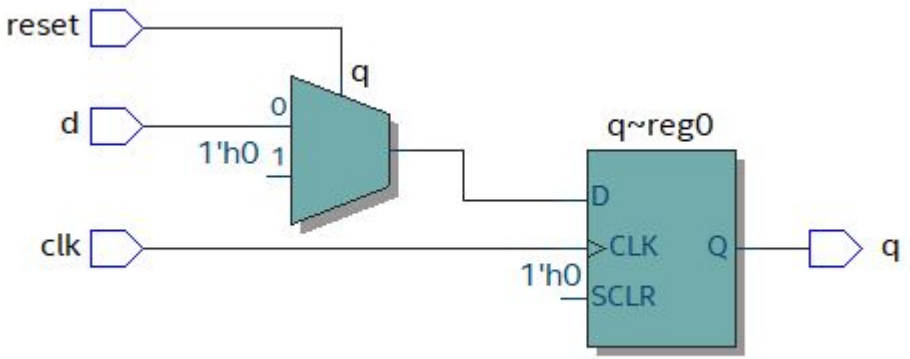
Mixing single and double edge in sensitivity list is not allowed and code will not synthesize

```
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
```

D-FF model with Reset

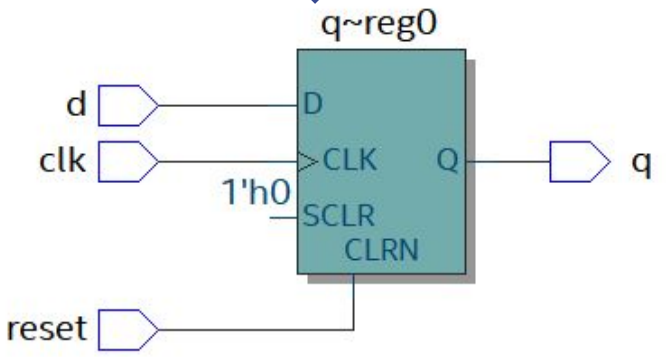
D-FlipFlop with Synchronous Active High Reset

```
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
```



D-FlipFlop with Asynchronous positive edge reset

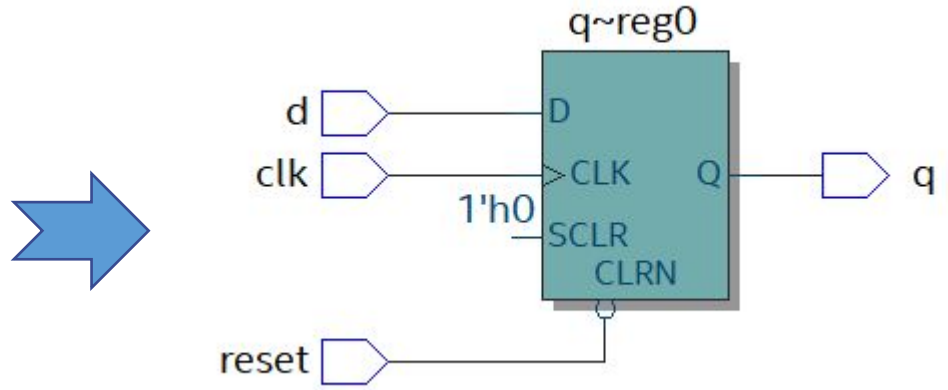
```
module dff(
 input logic clk, d, reset,
 output logic q
);
always_ff@(posedge clk, posedge reset) begin
  if(reset)
    q <= 0;
  else
    q <= d;
end
endmodule: dff
```



Resettable D-FF Model

D-FlipFlop with Asynchronous Negedge Reset

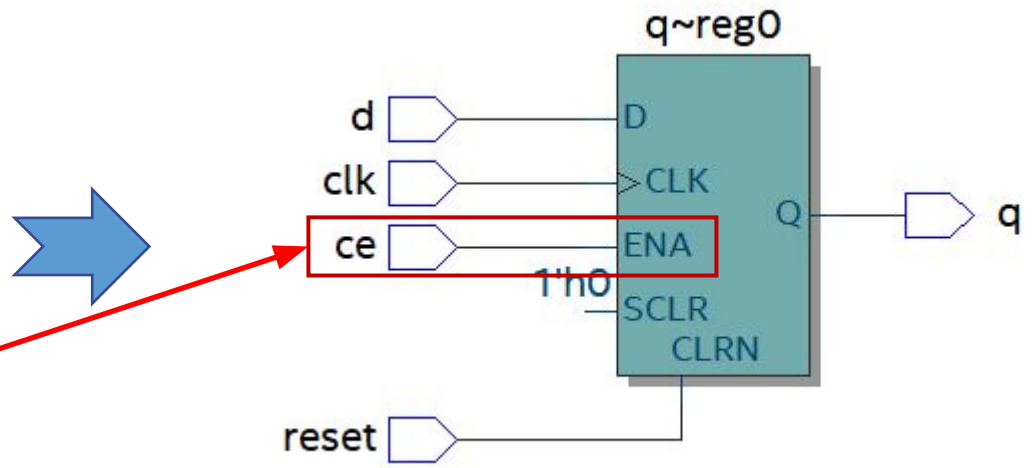
```
module dff(
 input logic clk, d, reset,
 output logic q
);
always_ff@(posedge clk, negedge reset) begin
  if(!reset)
    q <= 0;
  else
    q <= d;
end
endmodule: dff
```



D-FF Model with Clock Enable and Asynch Reset

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
```



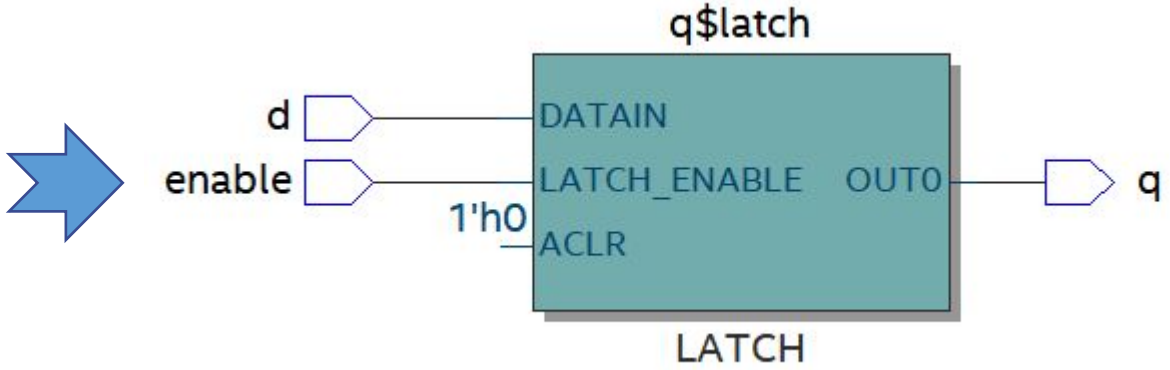
always_latch

always Block: always_latch

- **always_latch** indicates to EDA tools, that designer of RTL code intents to model a latch

- Similar to **always_comb,** i.e., 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.

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

always_latch begin
  if(enable)
    q <= d;
end

endmodule: latch
```



always_latch Coding Rules

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

always Block: always_latch

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

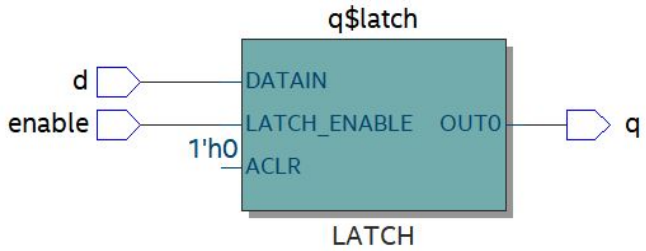
Approach-C Latch Model using always_latch auto sensitivity list

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

Best Practice

Synthesis compiler throws a warning for a potentially unintentional latch

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

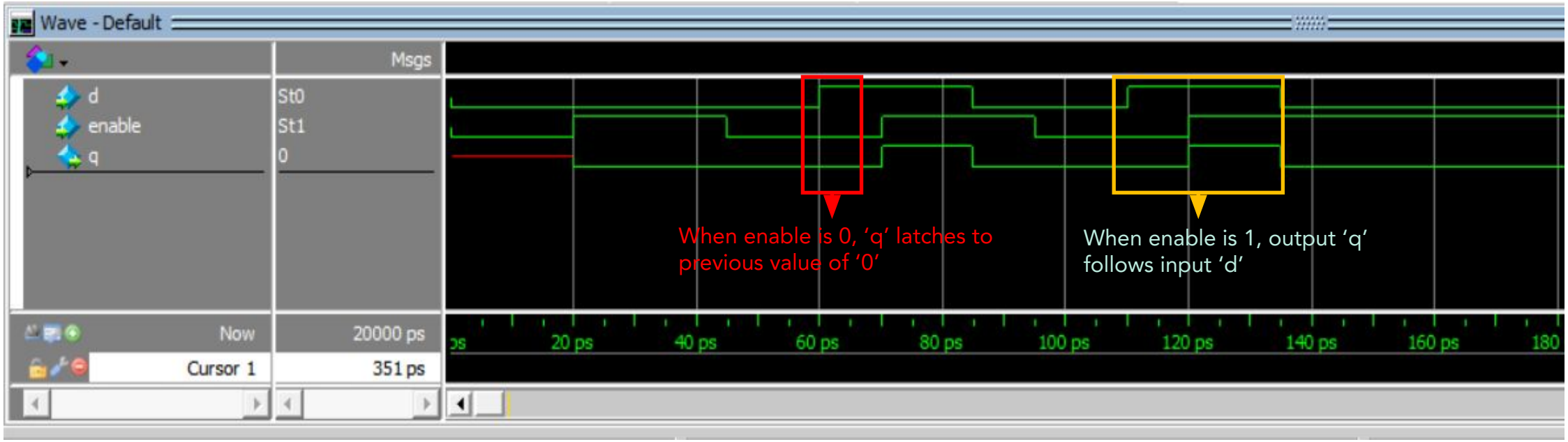


Synthesis compiler does not generate any warning message since always_ff instructs synthesis compiler to infer an intentional latch. Instead generates a info message

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

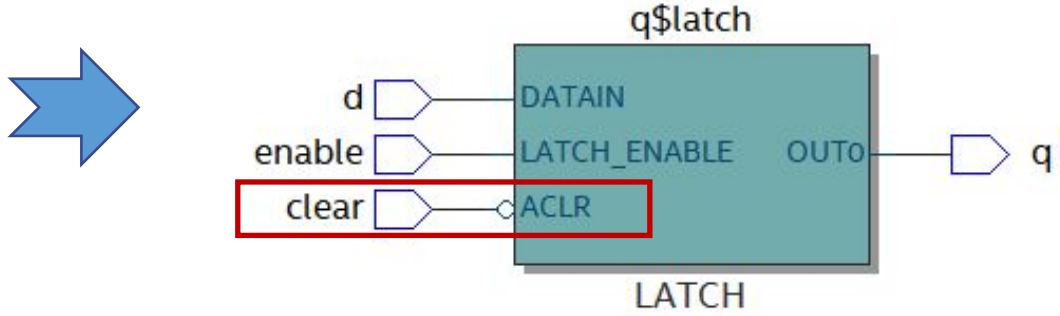
Latch Model Simulation Results

- All three approaches give an identical simulation result

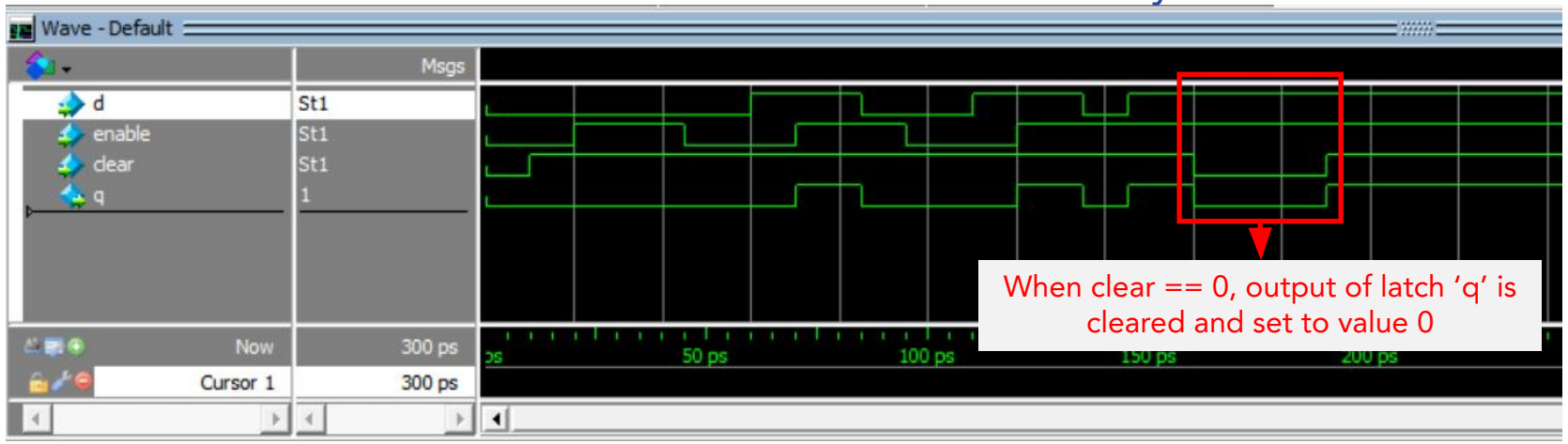


Latch with Asynchronous Negedge clear

```
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 Calls

- 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

≠

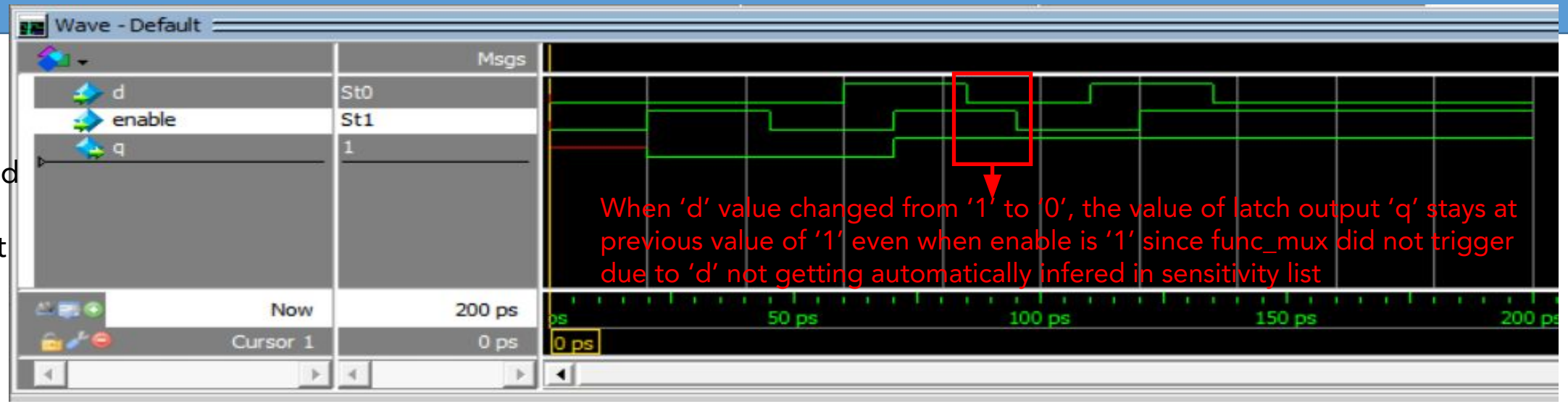
Latch RTL model using always_latch

```
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
```

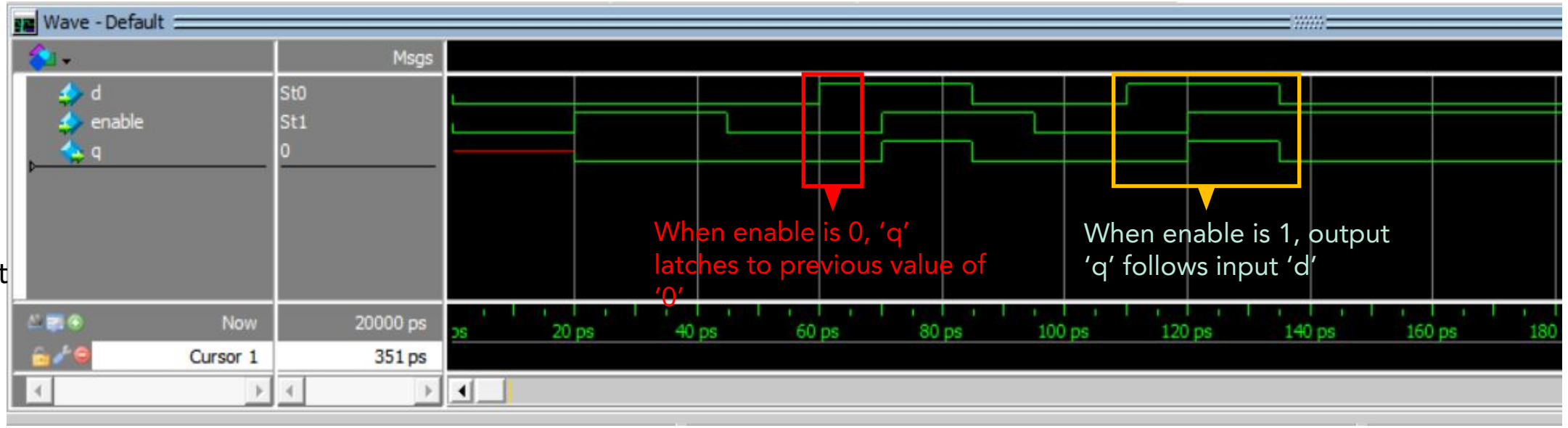
func_latch will be triggered whenever there is change in value of "d" and when enable is '1' but enable value did not change

Latch Model Simulation Results

always@(*) based implementation simulation result



always_latch based implementation simulation result



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 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.

always Procedural Blocks Example…

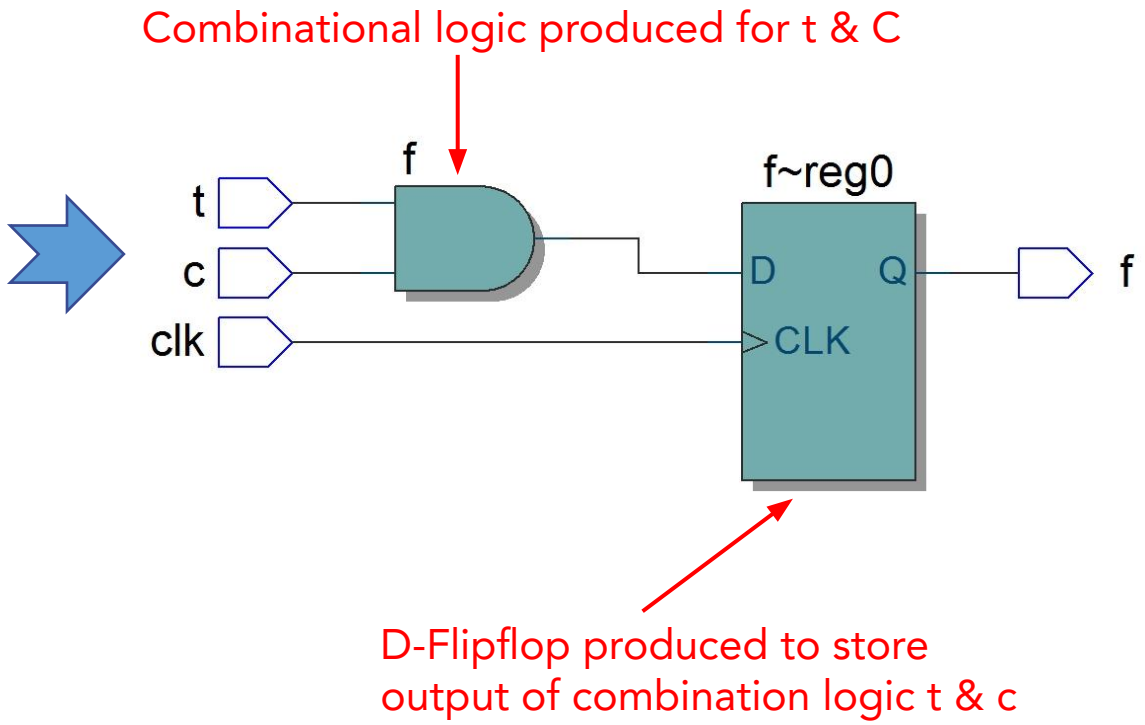
Example 1

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

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

Example 1: 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 Flip-flop produced for "f"
- "And" gate produced for t & C



Example 2

```
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
```

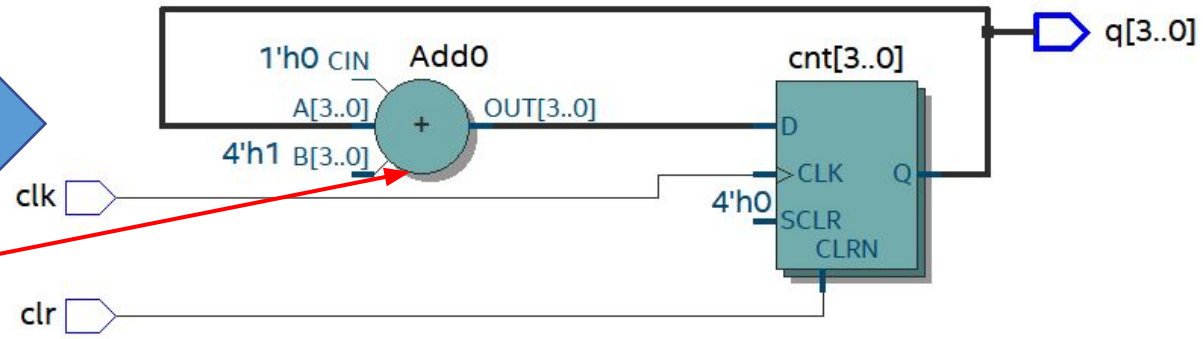
Example 2: N-bit Up-Counter/Asynch clear

```
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



Example 3

```
module ex3(
  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: ex3
```

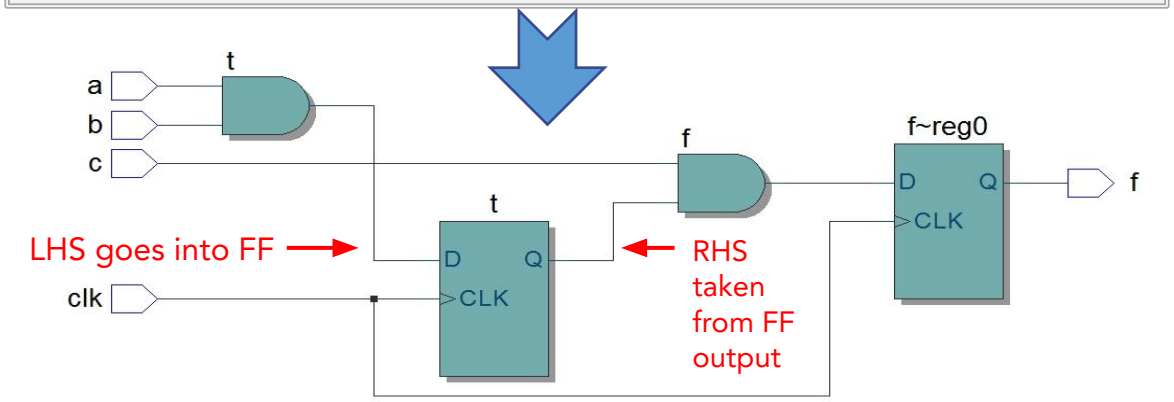
Example 3: Serially connected D Flip-flops with Combination Logic

```
module ex3(
  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: ex3
```

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

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

RHS reads from "t" register



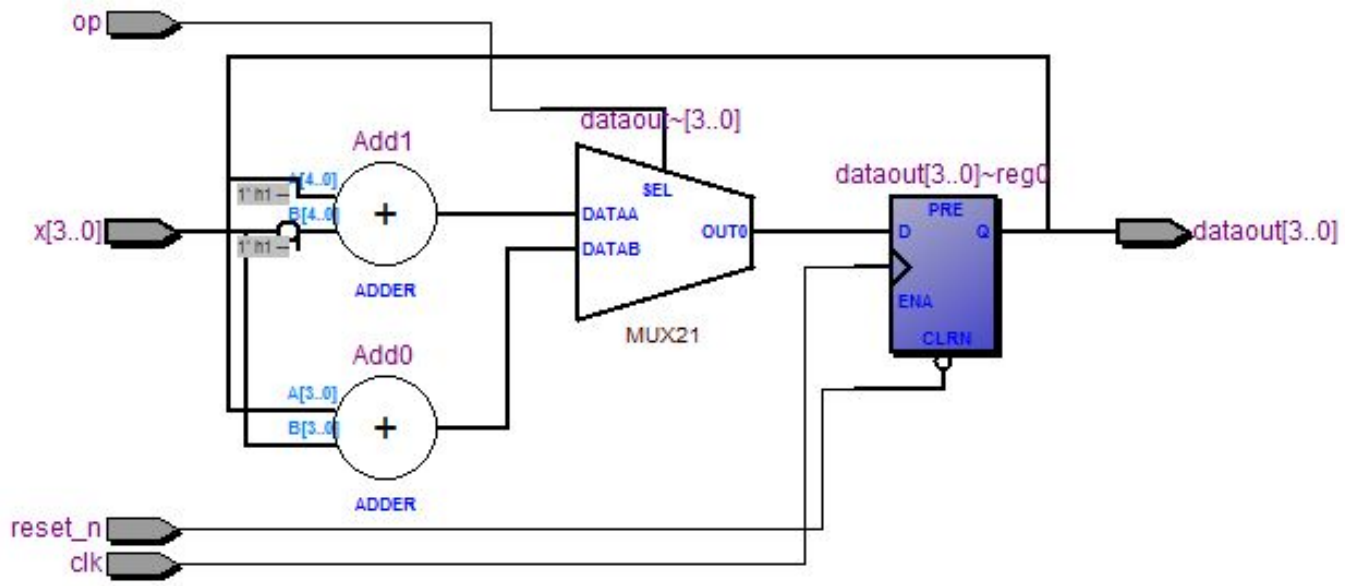
Example 4

```
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
```

Credit to : Professor Bill Lin, UCSD, ECE Department

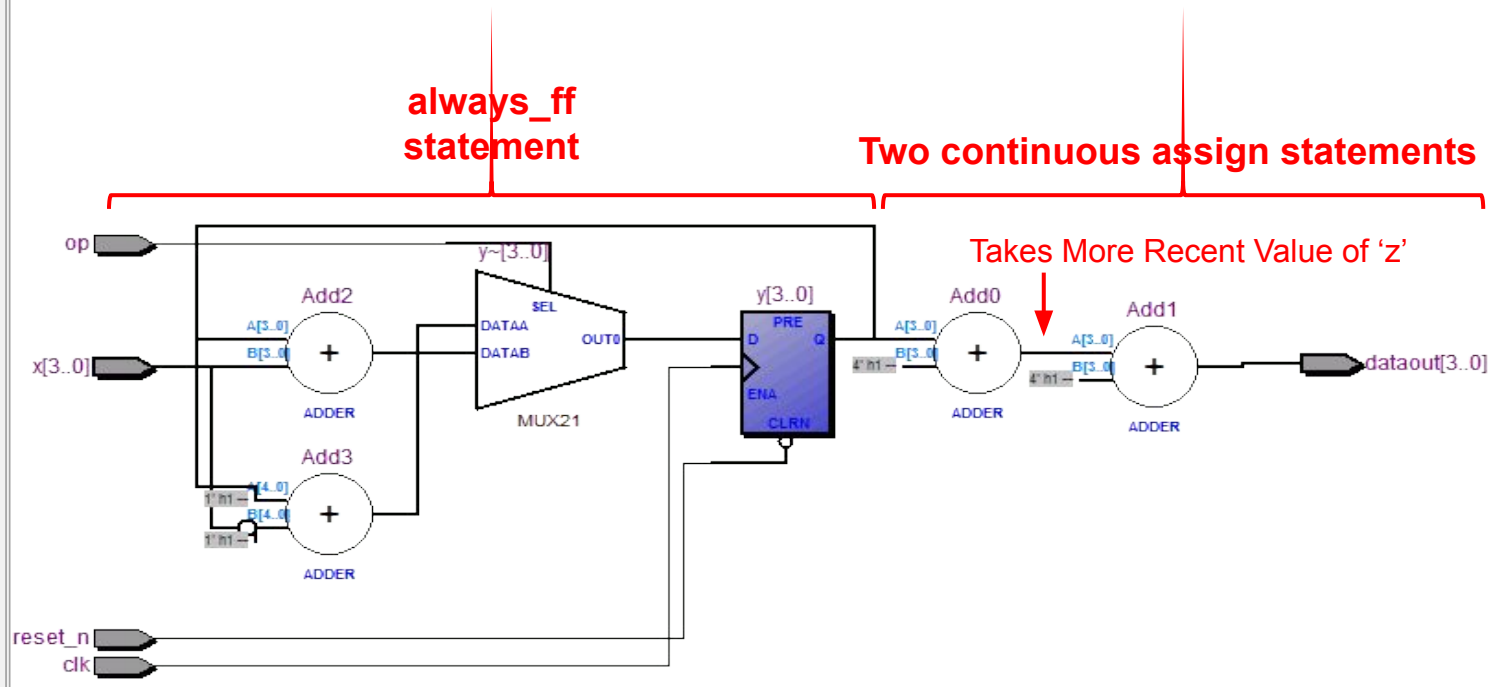
Example 4: Clocked always_ff Statement with (Negedge) Reset

```
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
```



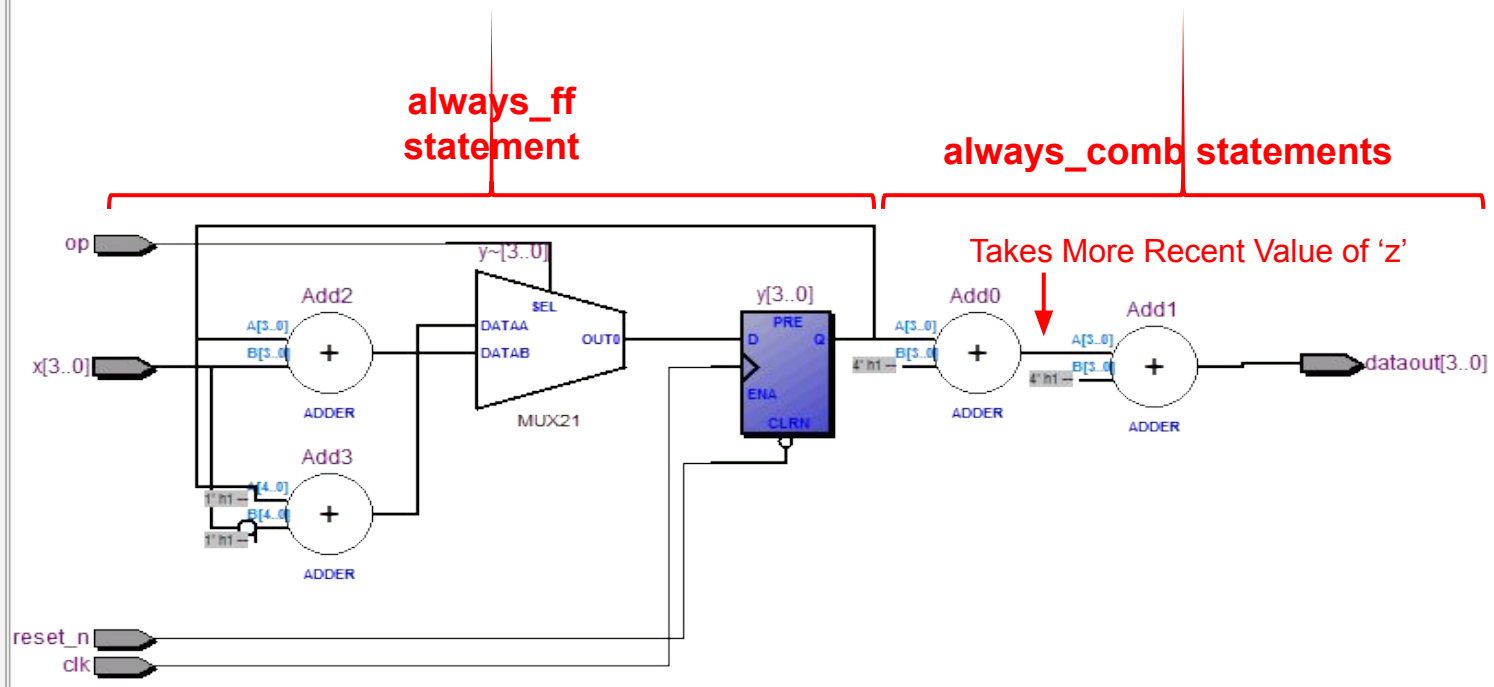
Example 5: Mixing Procedural Blocks and Continuous Assignment Statements

```
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

 assign z = y + 1;
 assign dataout = z + 1;
endmodule: ex5
```



Example 6: 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
end
always_comb begin
   z = y + 1;
   dataout = z + 1;
end
endmodule: ex6
```



always_ff Good and Bad Coding Styles

D-Similar D-FF (Bad Coding…)

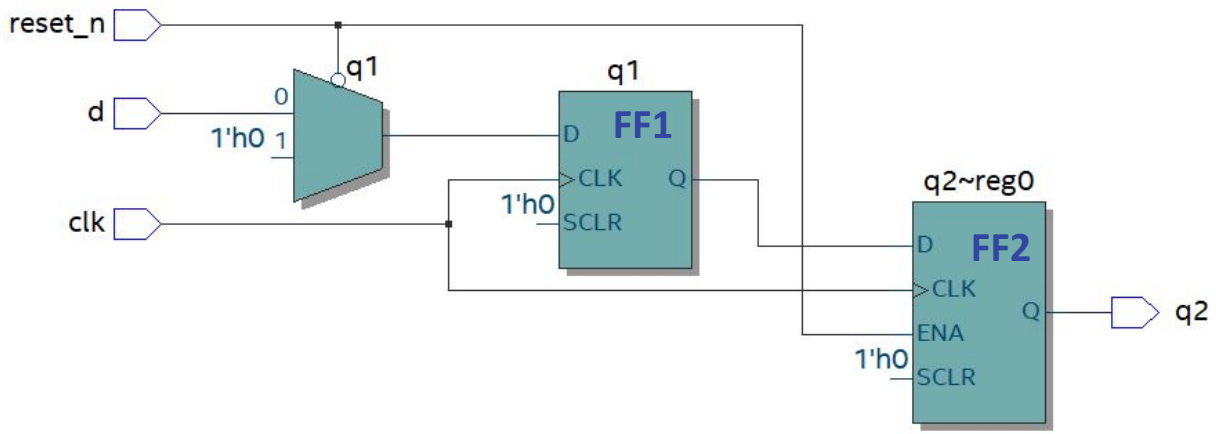
```
module bad_FF_coding(
 input logic clk, d, reset_n,
 output logic q2);

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
```

D-Similar D-FF (Bad 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
```

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

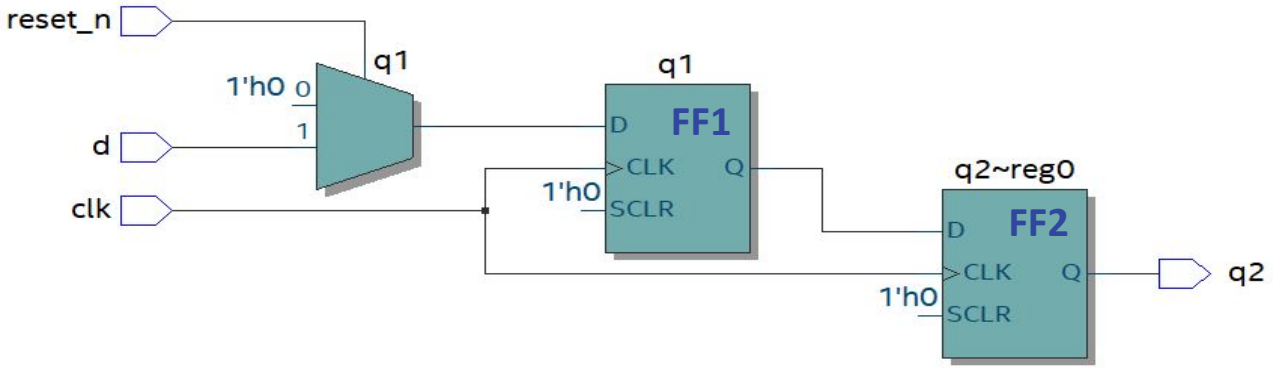


D-Similar D-FF (Good Coding…)

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
```

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



Multiple Assignment Statements 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
```

Code will synthesize however circuit will not function correctly.

Why? (watch the lecture discussion!)

Initial Procedural Blocks

Initial Procedural Blocks

- **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

- **S**

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

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

Initial Procedural Blocks

- 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
  - 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 can result in Race Conditions

```
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
endmodule
```

Not recommended!

Brief Summary on initial and always Procedural Blocks

Initial Procedural Blocks

- **Not synthesizable** and does not generate hardware logic
- Used for **simulation purpose**
- 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
- **Any number** of initial blocks can be defined within a module
- **Multiple** initial blocks executes **concurrently**

Always Procedural Blocks

- **Synthesizable** and it can generate hardware logic
- Used for **specifying design behavior** (RTL code)
- Starts **execution from beginning** of a simulation at time 0
- **Executes repeatedly**. Its activity shall cease only when the simulation is terminated
- **Any number** of always blocks can be defined within a module
- **Multiple** always blocks executes **concurrently**

Quiz