Lecture 8: Blocking vs. Non Blocking II

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)

- **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 15:** Homework 3 was posted on Canvas
  - Due on Wednesday, **10/23/24**
- **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

Blocking vs. Non Blocking

Blocking

```
sum = a + b;
prod = sum * c;
```

- Evaluation and assignment in a single step
  - Expression on RHS of (=) assignment is evaluated and the variable on LHS is updated immediately before the next sequential statement in the procedural block is evaluated and executed
- Each blocking assignment statement executes sequentially in the order it is specified in a procedural block
  - Order matters!
- Used to model combinational logic

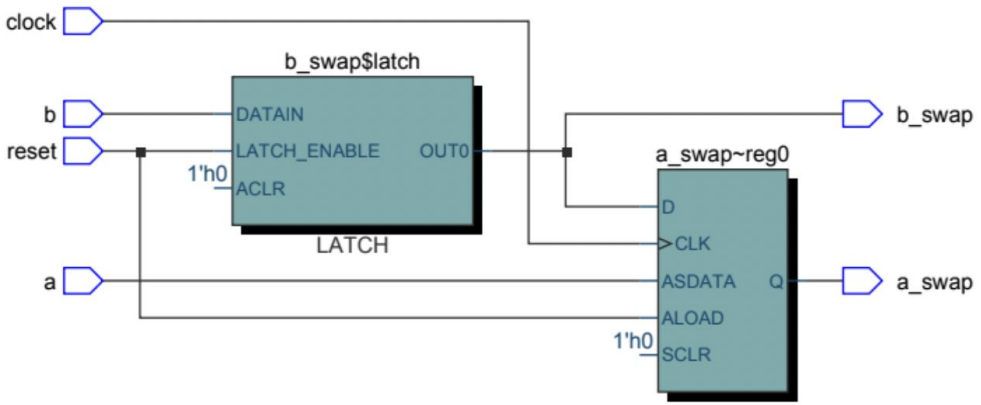
Non-blocking

```
sum <= a + b;
prod <= sum * c;
```

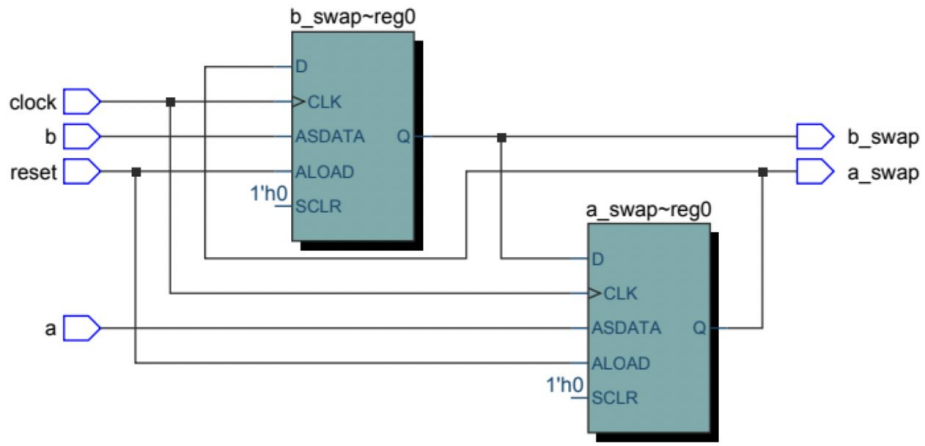
- Evaluation and assignment in two separate steps
  - Expression on RHS of (=) assignment is evaluated but the LHS variable is update is postponed till all the statements in the procedural block are evaluated and executed
- Each nonblocking assignment statement executes concurrently (i.e., in parallel) without blocking each other
  - Order does not matter!
- Used to model sequential logic

Which Code would swap a_swap and b_swap?

```
module blocking_assignment (
 input logic clock, rst, a, b,
 output logic a_swap, b_swap
);

always@(posedge rst, posedge clock)
 begin
  if (rst == 1) begin
     a_swap = a;
     b_swap = b;
  end
  else begin
    a_swap = b_swap;
    b_swap = a_swap;
  end
 end
endmodule
```




```
module non_blocking_assignment (
 input logic clock, rst, a, b,
 output logic a_swap, b_swap
);

always@(posedge rst, posedge clock)
 begin
  if (rst == 1) begin
     a_swap <= a;
     b_swap <= b;
  end
  else begin
    a_swap <= b_swap;
    b_swap <= a_swap;
  end
 end
endmodule
```



The Final (p,q) Values at the Next Clock Edge…

- Assume Initial Value of p=5 and q=8

```
always@ (posedge clock) begin
   p = q;
end
always@ (posedge clock) begin
   q = p;
end
```

Both always blocks will execute concurrently and there is a race condition between two always procedural assignments

Two possibilities:

1. p=8 and q=8
2. p=5 and q=5

No swapping of values of p and q in either case!

```
always@(posedge clock) begin
   p = q;
   q = p;
end
```

Simulator will execute p = q statement first and then execute the statement q = p

One possibility:

- p=8 and q=8

No swapping of values of p and q!

```
always@(posedge clock) begin
  tmp1 = p;
  tmp2 = q;
  p = tmp2;
  q = tmp1;
end
```

Simulator will execute four statements in order: (i) tmp1 = p, (ii) tmp2 = q, (iii) p = tmp2, (iv) q = tmp1.

One possibility:

- p=8 and q=5

Swapping of values of p and q happens!

The Final (p,q) Values at the Next Clock Edge…

- Assume Initial Value of p=5 and q=8

```
always@ (posedge clock) begin
   #0 p = q;
end
always@ (posedge clock) begin
   q = p;
end
```

q=p statement will always execute before the p=q statement as the #0 delay pushes the first statement to the "Inactive Region"

Single possibiliy:

1. p=5 and q=5

No swapping of values of p and q in either case!

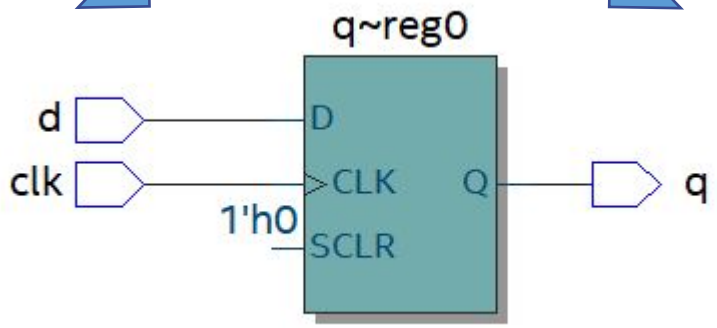
Synthesis of Blocking and Non Blocking Assignments

Which code would synthesis a D-FF?

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

always@(posedge clk) begin
    q = d;
end
endmodule: blocking_dff
```

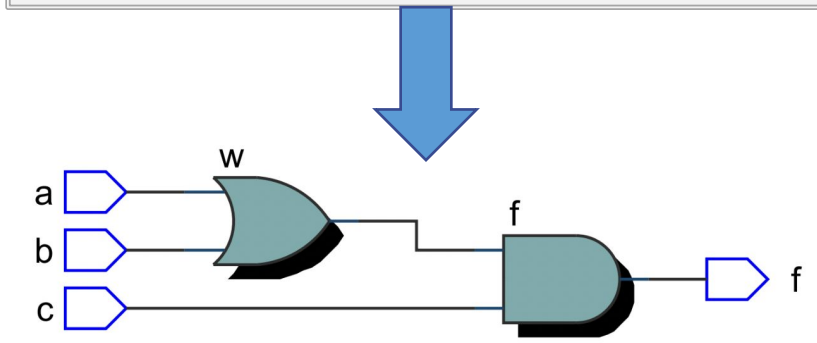
Both!

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

always@(posedge clk) begin
    q <= d;
end
endmodule: non_blocking_dff
```

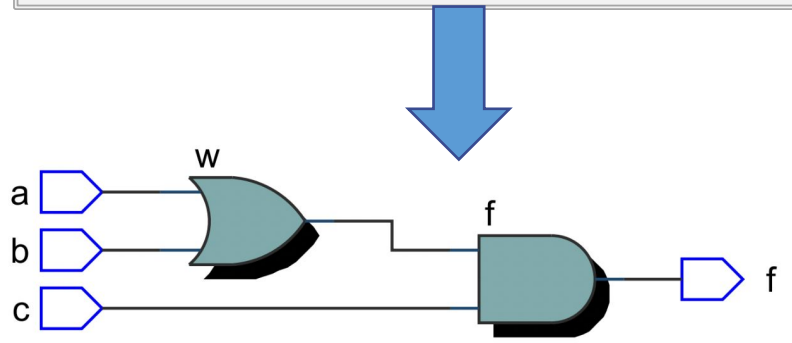


Combinational Logic: Blocking vs. Non!

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

logic w;
always @(*) begin
    w = a | b;
    f = w & c;
end
endmodule: blocking_comb
```

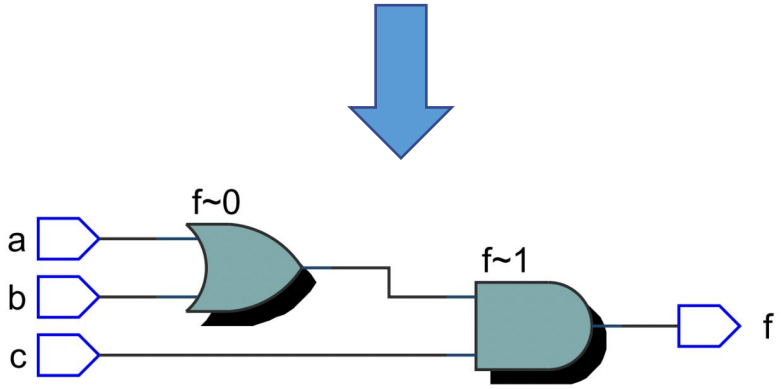



```
module non_blocking_comb (
    input logic a, b, c,
    output logic f
);

logic w;
always @(*) begin
    w <= a | b;
    f <= w & c;
end
endmodule: non_blocking_comb
```



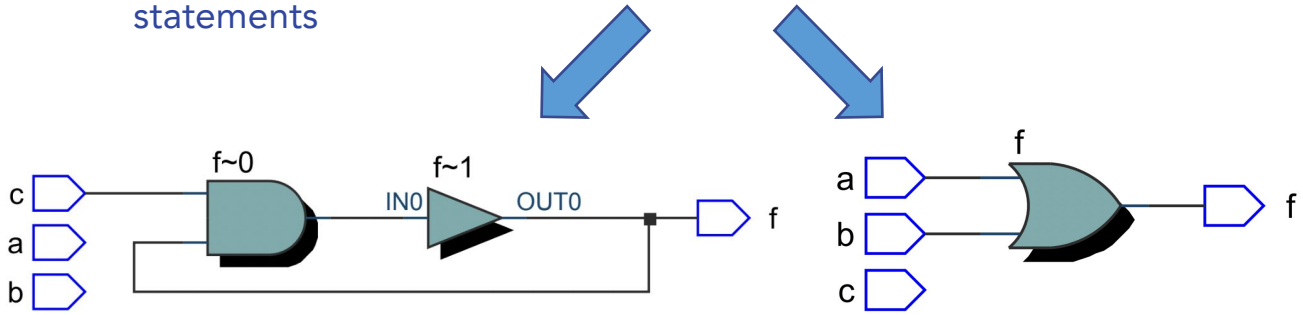
Combinational Logic: Blocking vs. Non!

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

always @(*) begin
    f = a | b;
    f = f & c;
end
endmodule: blocking_comb
```




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

always @(*) begin
    f <= a | b;
    f <= f & c;
end
endmodule: non_blocking_comb
```

Synthesis compiler can provide ambiguous netlist results when the same net is simultaneously driven by multiple non-blocking statements

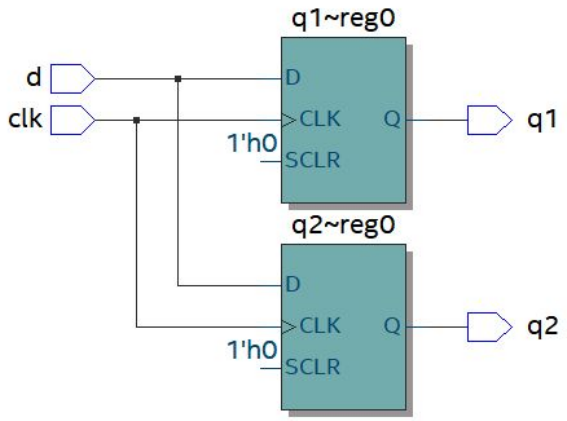


Sequential Logic: Blocking vs. Non!

```
module blocking_seq (
    input logic clk, d,
    output logic q1, q2
);

always @(posedge clk) begin
    q1 = d;
    q2 = q1;
end
endmodule: blocking_seq
```

q1 is not connected to q2 and ~~both q1 and~~ q2 will get the value of d in the same clock cycle

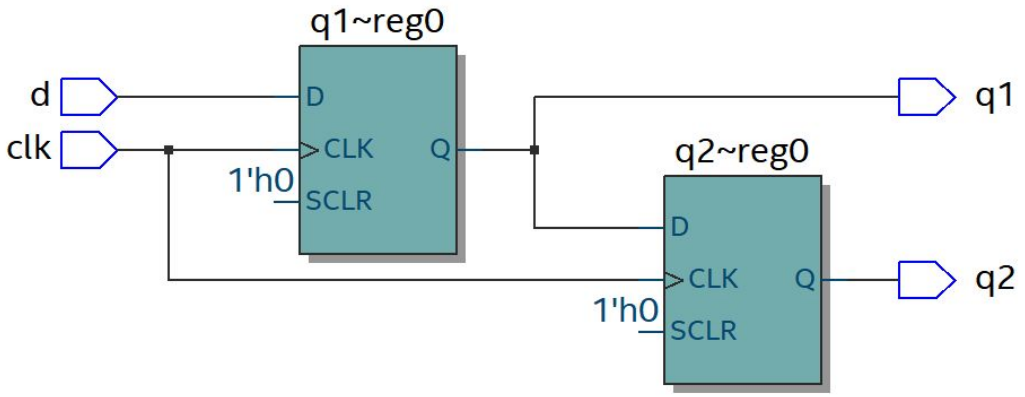
Synthesis compiler will create two registers in parallel




```
module non_blocking_seq (
    input logic clk, d,
    output logic q1, q2
);

always @(posedge clk) begin
    q1 <= d;
    q2 <= q1;
end
endmodule: non_blocking_seq
```

New value of d is propagated to q1 and one clock cycle later it will propagate to q2

Synthesis compiler will create two serially chained registers and circuit will behave as a two bit shift register

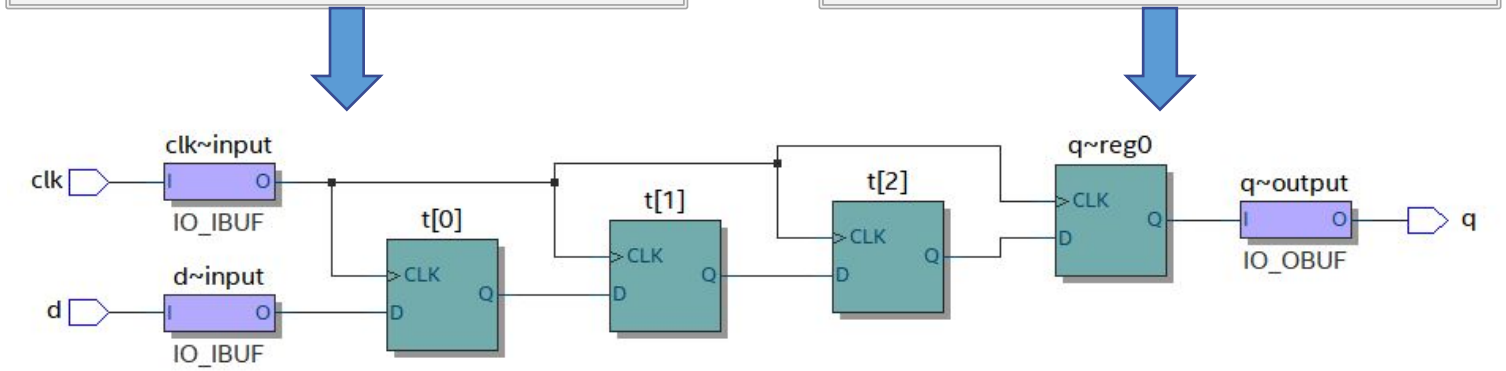


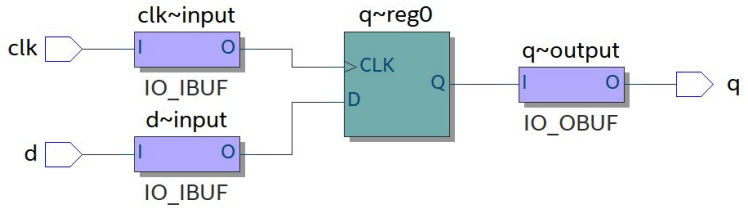
Sequential Logic: Blocking vs. Non!

```
module shift_register_1 (
    input logic clk, d,
    output logic q
);
logic[2:0] t;
always@(posedge clk)
begin
    t[0] <= d;
    t[1] <= t[0];
    t[2] <= t[1];
    q  <= t[2];
end
endmodule
```

```
module shift_register_2 (
    input logic clk, d,
    output logic q
);
logic[2:0] t;
always@(posedge clk)
begin
    q  = t[2];
    t[2] = t[1];
    t[1] = t[0];
    t[0] = d;
end
endmodule
```

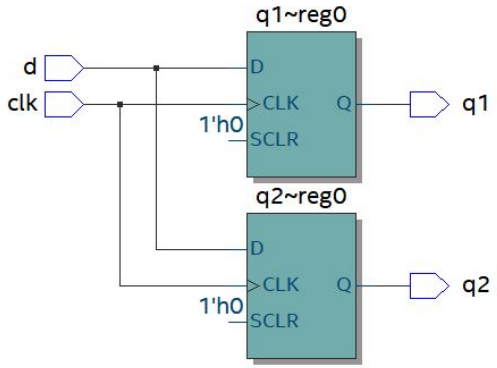
```
module incorrect_shift_register (
    input logic clk, d,
    output logic q
);
logic[2:0] t;
always@(posedge clk)
begin
    t[0] = d;
    t[1] = t[0];
    t[2] = t[1];
    q  = t[2];
end
endmodule
```



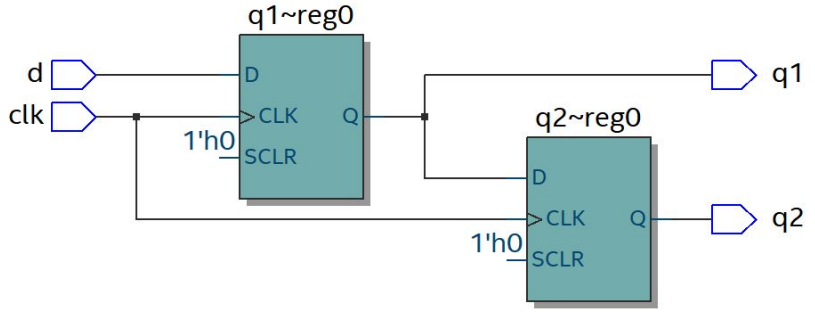


Reordered Blocking Assignments for Sequential can lead to **Inconsistencies**…

```
module shift_register_2 (
    input logic clk, d,
    output logic q1, q2
);
always@(posedge clk)
begin
    q1  = d;
    q2  = q1;
end
endmodule
```



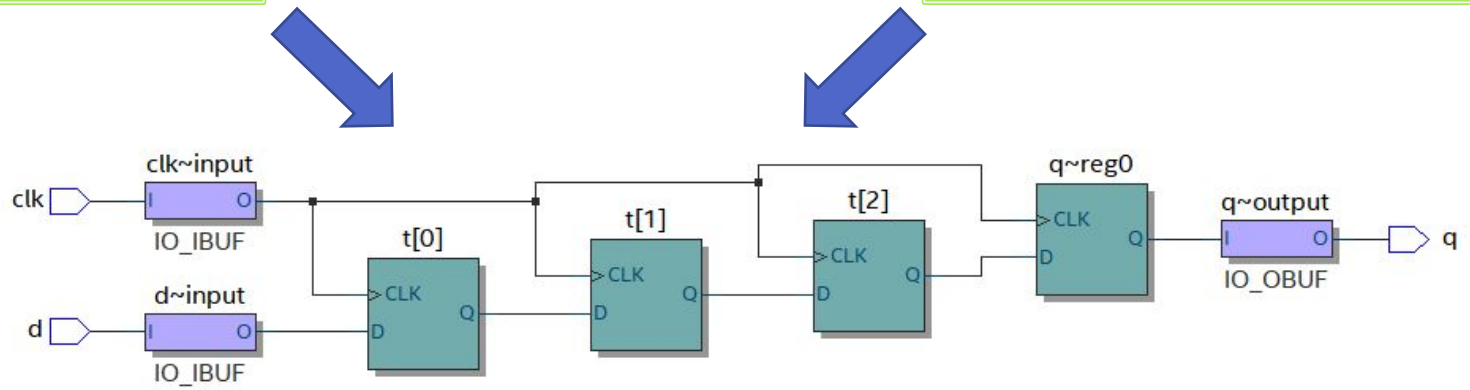

```
module incorrect_shift_register (
    input logic clk, d,
    output logic q1, q2
);
always@(posedge clk)
begin
    q2  = q1;
    q1  = d;
end
endmodule
```



Reordered Blocking Assignments for Sequential provides **Consistent Results**

```
module shift_register_1 (
    input logic clk, d,
    output logic q
);
logic[2:0] t;
always@(posedge clk)
begin
    t[0] <= d;
    t[1] <= t[0];
    t[2] <= t[1];
    q  <= t[2];
end
endmodule
```

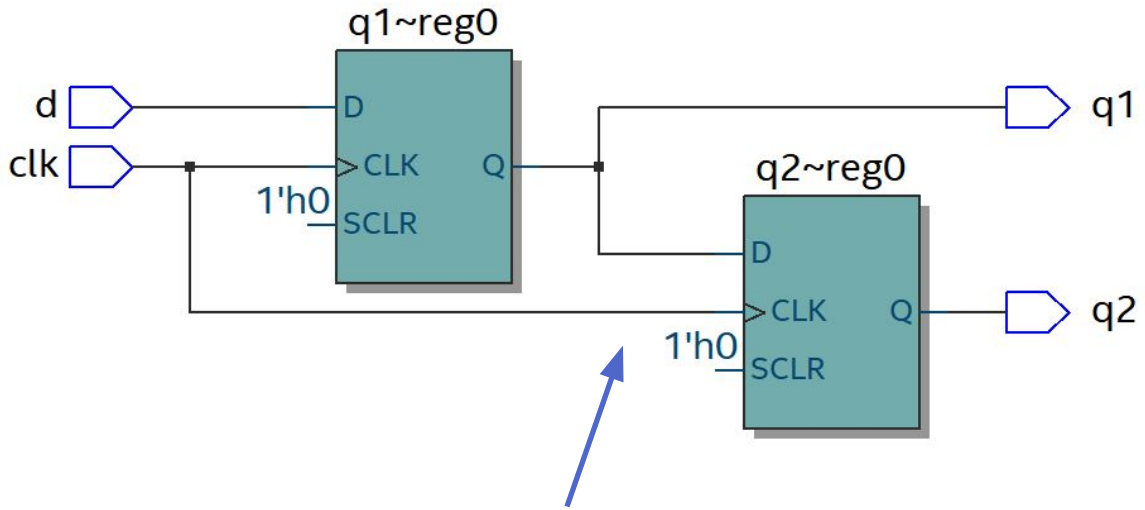
```
module shift_register_2 (
    input logic clk, d,
    output logic q
);
logic[2:0] t;
always@(posedge clk)
Begin
    q  <= t[2];
    t[1] <= t[0];
    t[0] <= d;
    t[2] <= t[1];
end
endmodule
```



Splitting Blocking Assignments in Separate always Blocks

```
module shift_register (
  input logic clk, d,
  output logic q1, q2
);

always@(posedge clk)
 begin
   q1 = d;
 end

always@(posedge clk)
 begin
    q2 = q1;
 end

endmodule
```

Splitting blocking assignment statements in two separate always block will result in a two-bit shift register upon synthesis since both assignments will execute in parallel



Synthesis compiler will connect q1 to q2 and circuit will behave as a two-bit shift register

Note : However in simulation due to race condition between two always procedural block based on which always block executes first it might behave as a 2-bit shift register or 1-bit parallel registers.

- If always block with q2=q1 assignment executes first over other always block having q1=d then circuit will behave as a 2-bit shift register
- If always block with q1=d assignment executes first over other always block having q2=q1 then circuit will behave as 1-bit parallel register

Guidelines: Blocking and Non-Blocking Assignments

1. Use non-blocking assignments inside **`always_ff`** (next lecture) procedural blocks to model synchronous sequential logic

2. Use blocking assignments inside **`always_comb`** (next lecture) procedural blocks to model combinational logic

3. Do not make assignments to the same signal in more than one always statement or continuous assignment

```
always@(a,b) begin
  c <= a + b;
  c <= a * b;
end
```

4. Do not mix blocking and non-blocking assignments in the same always block

```
always@(a,b) begin
  sum = a + b;
  prod <= a * b;
end
```

(Harris & Harris book)

Procedural Assignments with Delays

Inter and Intra-Delay Syntax

- Inter assignment delay (delay on the LHS)
  - Execution of the entire statement is delayed
  - Syntax: `#<delay> <LHS> = <RHS>`
  - Example: `#5 a = b | c;`

- Intra assignment delay (delay on the RHS)
  - Only the RHS of the assignment operator is delayed
  - Syntax: `<LHS> = #<delay> <RHS>`
  - Example: `q <= #5 (a & b) | c;`

Sequential Procedural Assignments with Inter-Delays

- A sequential blocking assignment evaluates and assigns before continuing within procedural block
  - Timing control before an assignment statement (inter assignment delay) will postpone when the next statement is evaluated and updated
  - Order of evaluation is deterministic

```
always@(posedge clk) begin
  a = 1;
  #4 b = a + 2;
end
```

`a = 1;` → Evaluate RHS and assign to 'a' immediately

`#4 b = a + 2;` → After execution of previous statement, advance 4 time units due to inter assignment delay, then evaluate RHS (a+2) and assign it to 'b' immediately

equivalent

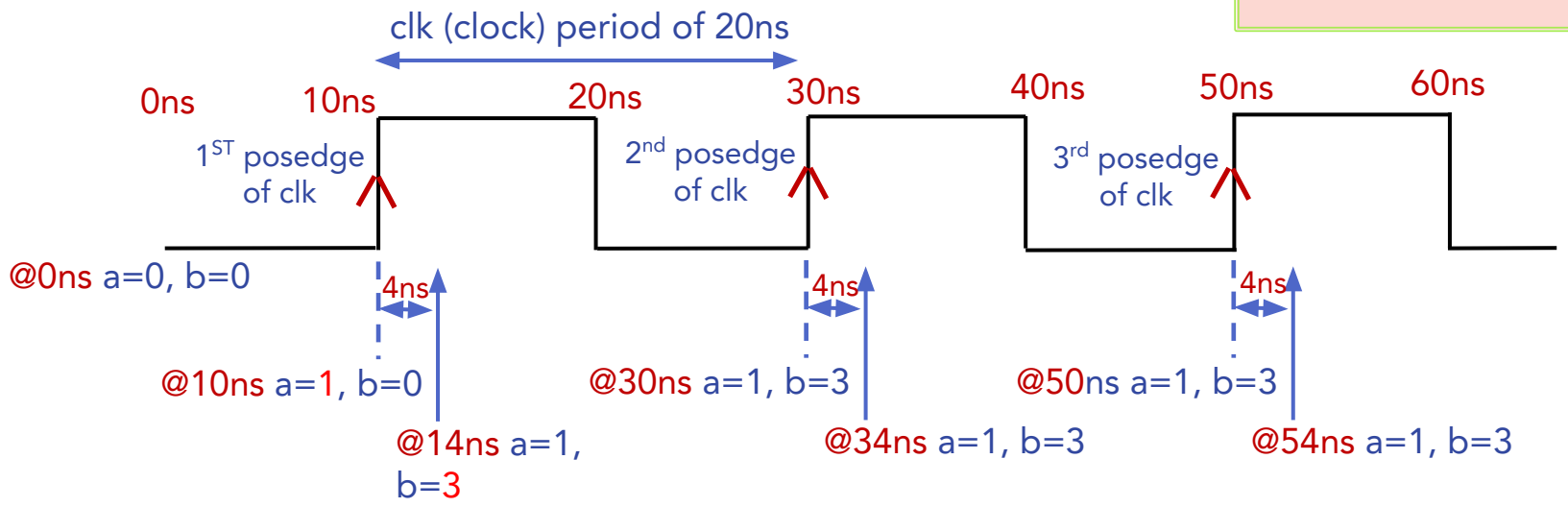
```
always@(posedge clk) begin
  a = 1;
  #4 ns;
  b = a + 2;
end
```

Simulation Output

Assume At 0ns a=0, b=0

At 1st posedge of clk a=1

At 1st posedge of clk+4ns b=3



Sequential Procedural Assignments with Inter-Delays

- Timing control before a non-blocking assignment statement (inter assignment delay) will postpone when the next assignment is evaluated and updated
  - Order of evaluation is deterministic

```
always@(posedge clk) begin
  a <= 1;
  #4 b <= a + 2;
end
```

a <= 1; → Evaluate RHS immediately; then assign to 'a' at the end of the time step

#4 b <= a + 2; → Delay 4 time units due to inter assignment delay, then evaluate RHS; then assign to 'b' at the end of time step (1 clock period + 4ns)

equivalent

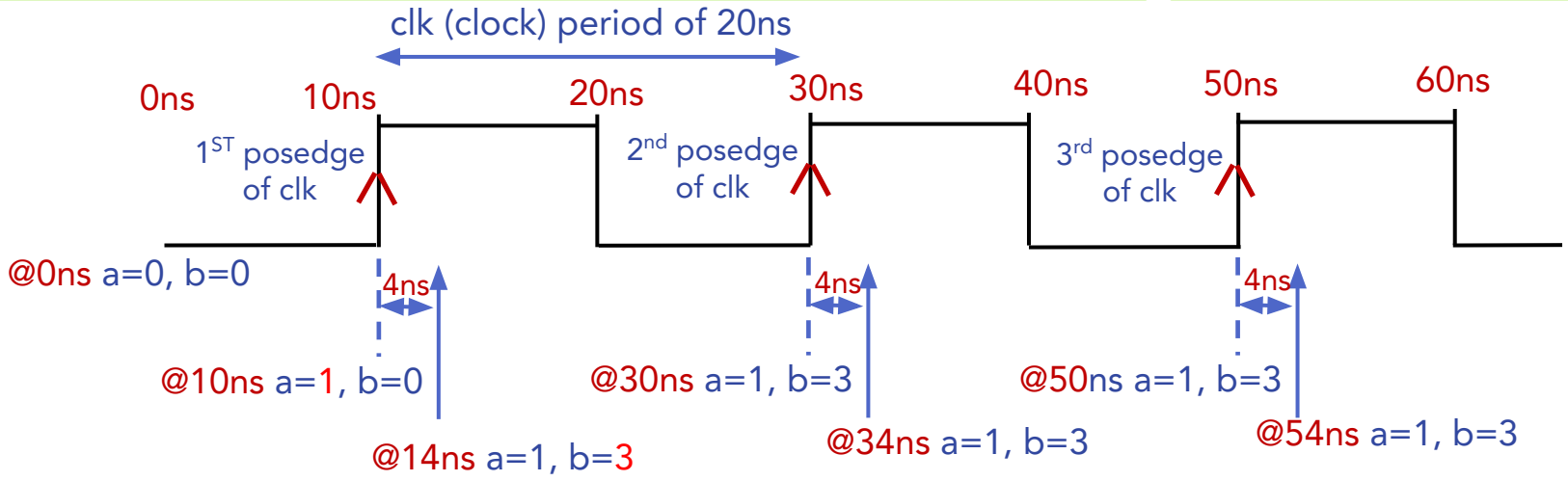
```
always@(posedge clk) begin
  a <= 1;
  #4 ns;
  b <= a + 2;
end
```

Simulation Output

Assume At 0ns a=0, b=0

At 1st posedge clk, a=1

At 1st posedge clk+4ns, b=3



Note : Event driven simulator will not re-evaluate value of 'a' and 'b' at 34ns and 54ns since there was no change in 'a' or 'b' value

Concurrent Procedural Assignment with Inter-Delays

- Concurrent blocking assignments have unpredictable results due to race condition
  - Order of concurrent evaluation is indeterministic and unpredictable simulation result!

```
always@(posedge clk) begin
  #4 a = a + 2;
end

always@(posedge clk) begin
  #4 b = a + 3;
end
```

Delay 4 time units due to inter assignment delay, then evaluate RHS and assign to 'a', immediately

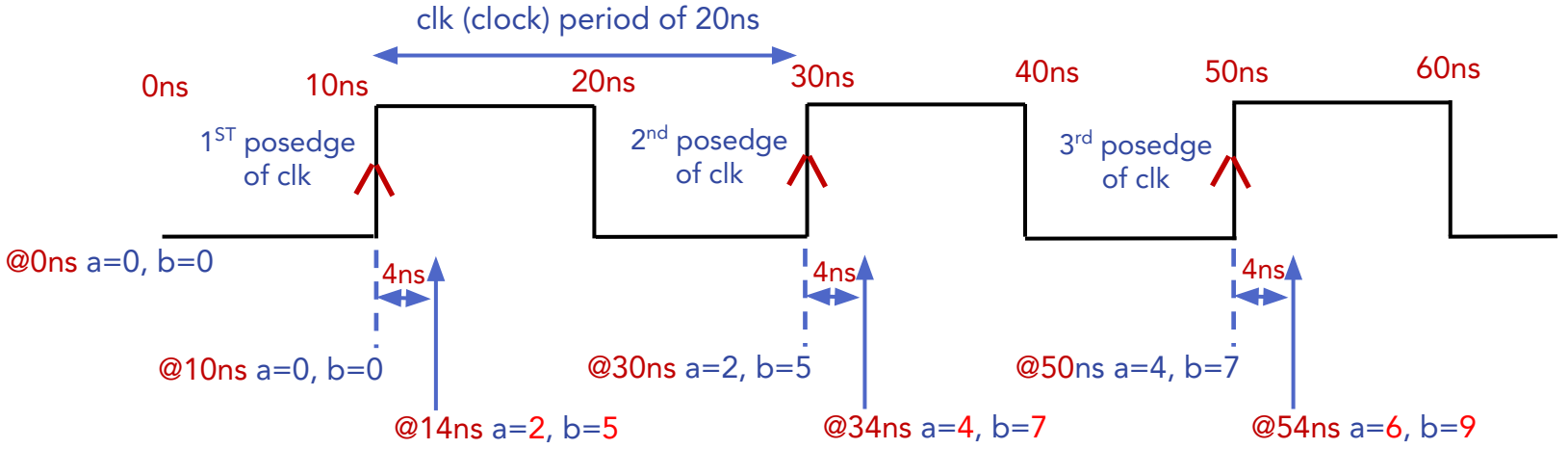
Delay 4 time units due to inter assignment delay, then evaluate RHS and assign to 'b', immediately

Simulation Output

Assume At 0ns a=0, b=0

Unpredictable Result !!

new value of 'b' could be evaluated before or after 'a' changes



Note : Result shown above is in case when simulator evaluates 'b' after 'a' is evaluated

Concurrent Procedural Assignment with Inter-Delays

- Concurrent non-blocking assignments have predictable results
  - Order of concurrent evaluation is indeterministic, but predictable simulation result!

```
always@(posedge clk) begin
  #4 a <= a + 2;
end

always@(posedge clk) begin
  #4 b <= a + 3;
end
```

Delay 4 time units due to inter assignment delay, then evaluate RHS; then assign to 'a' at the end of time step (1 clock period + 4ns)

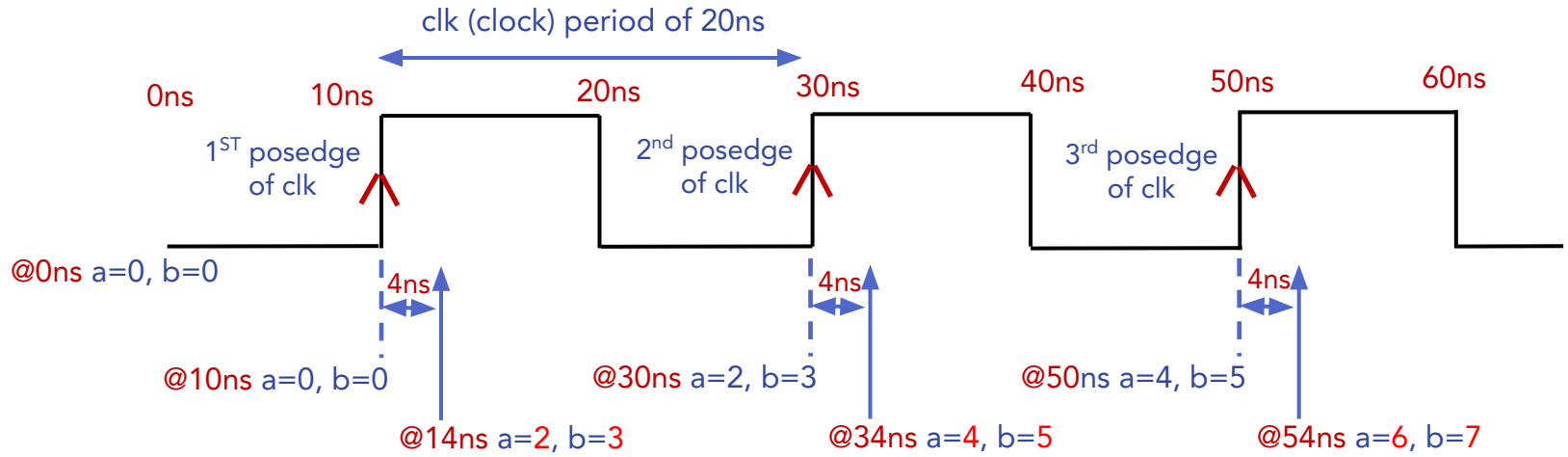
Delay 4 time units due to inter assignment delay, then evaluate RHS; then assign to 'b' at the end of time step (1 clock period + 4ns)

Simulation Output

Assume At 0ns a=0, b=0

Predictable Result !!

After 1st posedge clk + 4ns
a=2 and b=3



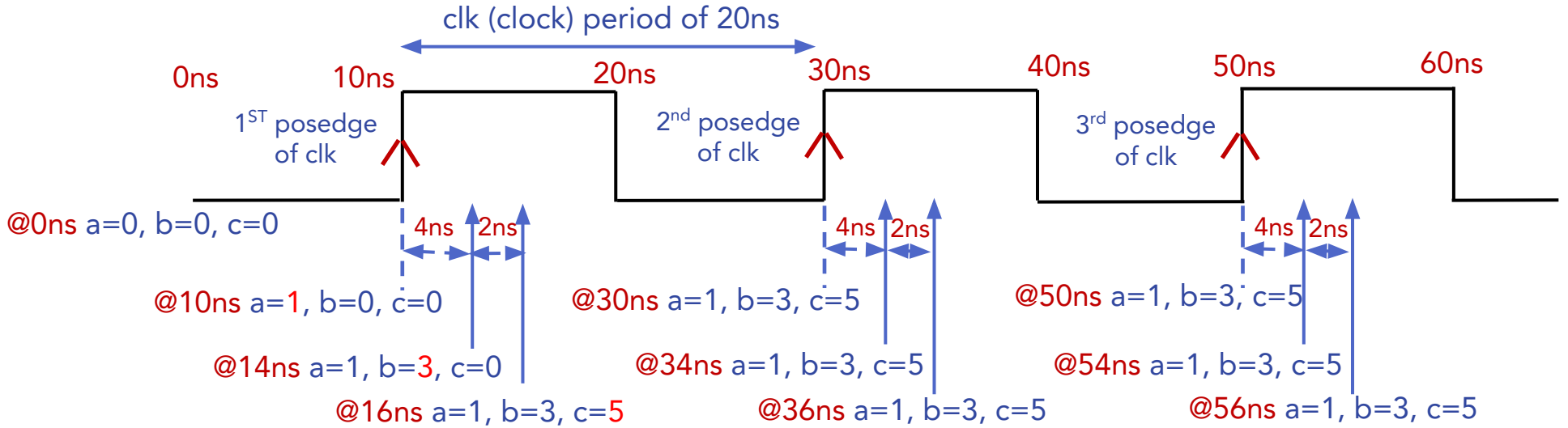
Procedural Concurrent Assignments with Intra-Delays

- Blocking statements evaluated and assigned sequentially
- Right-hand side is evaluated before the delay
- Left-hand side is assigned after the delay

```
always@(posedge clk) begin
  a = 1;          → Evaluate RHS and assign to 'a' immediately
  b = #4 a + 2;   → After execution of previous statement, evaluate RHS expression immediately,
                    then after intra delay of 4 time units assign to 'b'
  c = #2 b + 2;   → After execution of previosus statement, evaluate RHS expression immediately,
                    then after intra delay of 2 time units assign to 'c'
end
```

Simulation Output

Assume At 0ns a=0, b=0, c=0
At 1st posedge clk, a=1
At 1st posedge clk+4ns, b=3
At 1st posedge clk+6ns, c=5



Procedural Non-Blocking Assignments with Intra-Delays

- Right-hand side of non-blocking statements evaluated concurrently before the delay
- Left-hand side is assigned after the delay

```
always@(posedge clk) begin
  a <= 1;
  b <= #4 a + 2;
  c <= #2 b + 2;
end
```

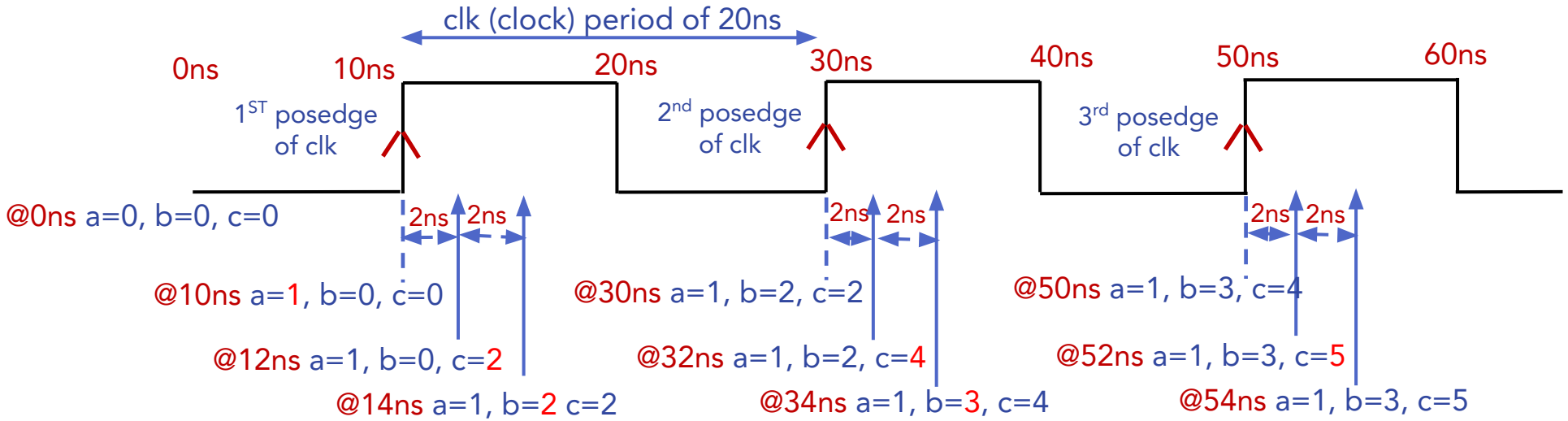
Evaluate RHS immediately; then assign to 'a' at the end of the time step

Evaluate RHS expression immediately, then after intra delay of 4 time units assign to 'b'

Evaluate RHS expression immediately, then after intra delay of 2 time units assign to 'c'

Simulation Output

Assume At 0ns a=0, b=0
At 1st posedge clk, a=1
At 1st posedge clk+2ns, c=2
At 1st posedge clk+4ns, b=2



The **always** block…

Overview of always Block

- Event in sensitivity list can be specified in multiple different ways :
  - Edge (posedge, negedge)
  - Level or dual edge (any change in value of signal)

```
// always block sensitive to
// posedge event of clock
always@(posedge clock) begin
  dout = din;
end
```

```
// always block sensitive to
// negedge event of clock
always@(negedge clock) begin
  dout = din;
end
```

```
// always block is sensitive to both
posedge and negedge event of interrupt
always@(interrupt) begin
  abort = 1;
end
```

Overview of always Block

- Always procedure can be used to model :
  - Combinational logic
  - Sequential logic
    - Edge-sensitive sequential logic (such as flip-flops)
    - Level-sensitive sequential logic (such as latches)

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

 always@(posedge clk)
 begin
   q <= d;
 end
endmodule
```

Sequential Logic

```
module comb (
  input logic inv, input logic [3:0] data,
  output logic [3:0] result);

  always@(inv, data)  begin
    if(inv) result = ~data;
    else    result = data;
  end
endmodule
```

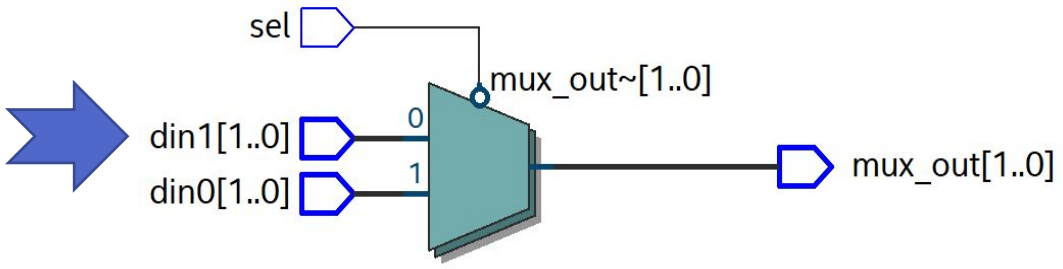
Combinational Logic

More always Block: Complete Sensitivity List

```
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

Synthesis compiler will generate 2to1 MUX



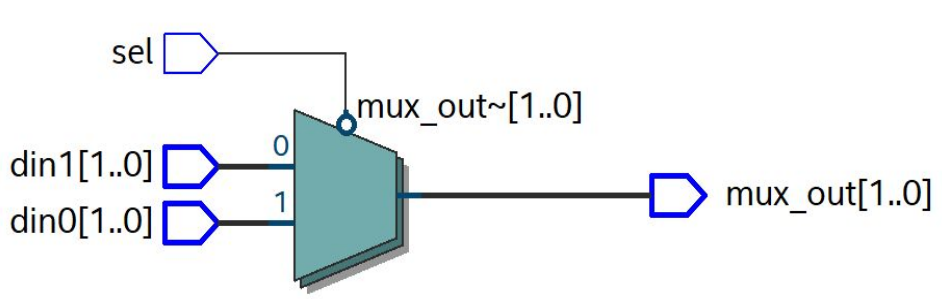
More always Block: Complete Sensitivity List

- In the example below, 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.

- Omission of any input signal in the sensitivity list which impacts behavior of logic can lead to simulation and synthesis mismatches

```
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
```

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

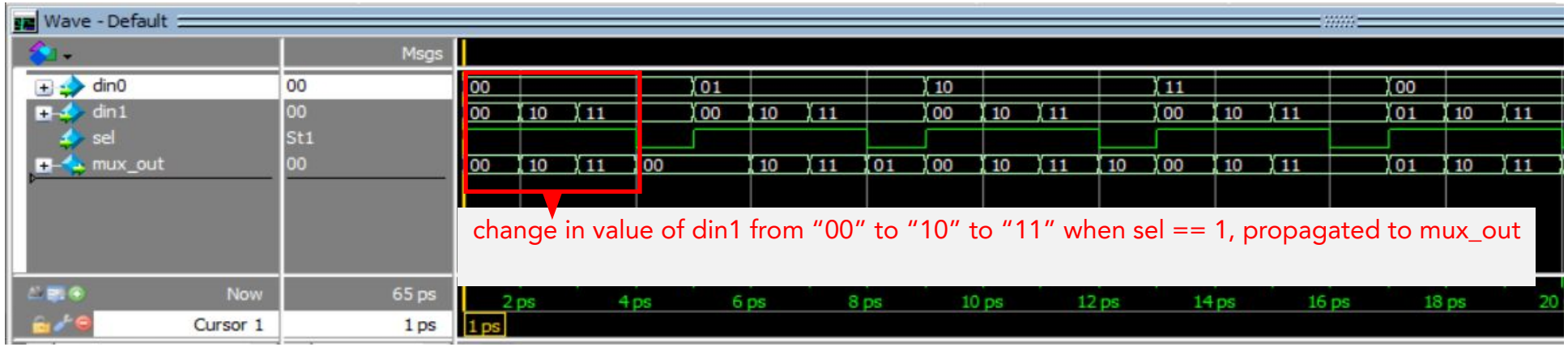


Simulation vs Synthesis results mismatch due to incomplete sensitivity list!

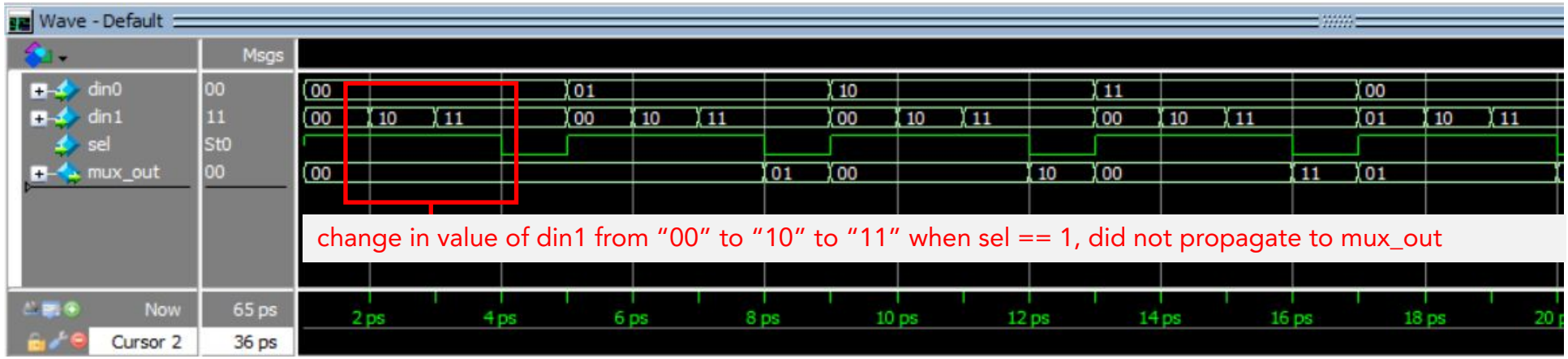
31

Complete vs. Incomplete Sensitivity List for always block

- Complete sensitivity list:



- Incomplete sensitivity list (i.e. with din1 is missing from the sensitivity list):



How about 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@(*)
begin
   if(sel == 1'b0) begin
      mux_out = din0;
   end else begin
      mux_out = din1;
   end
end
endmodule: mux
```

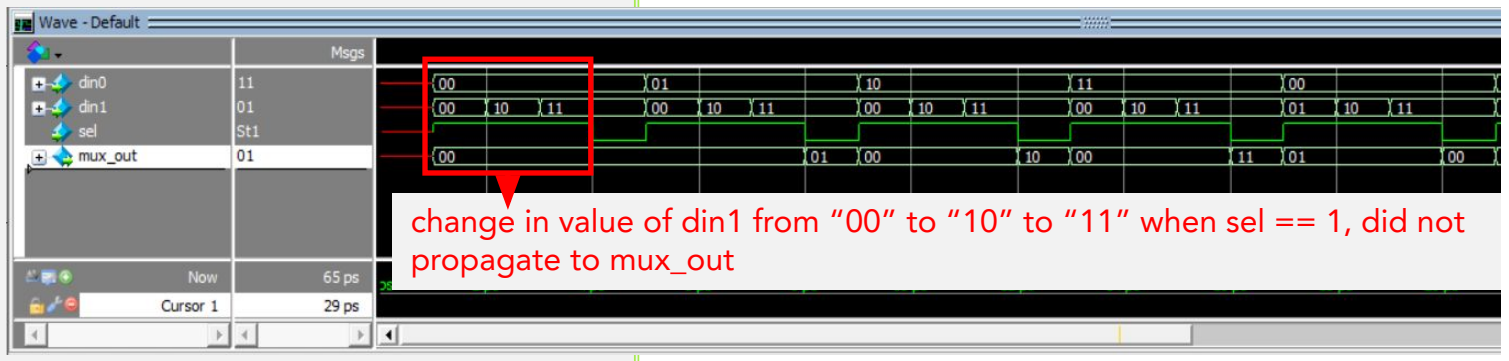
always@(*) automatically infers sel, din0, din1 in the sensitivity list

Limitations of always@*

- Unfortunately, always@* does not always infer a complete sensitivity list
  - For e.g., always@* will only be sensitive to the signals passed into the function or task called within the block and will not infer sensitivity to signals that are externally referenced by 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



Simulation result

Limitations of always@*: How to fix?

Solution 1: Explicitly specify the correct 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
    mux_out = func_mux(sel);
end
endmodule: mux
```

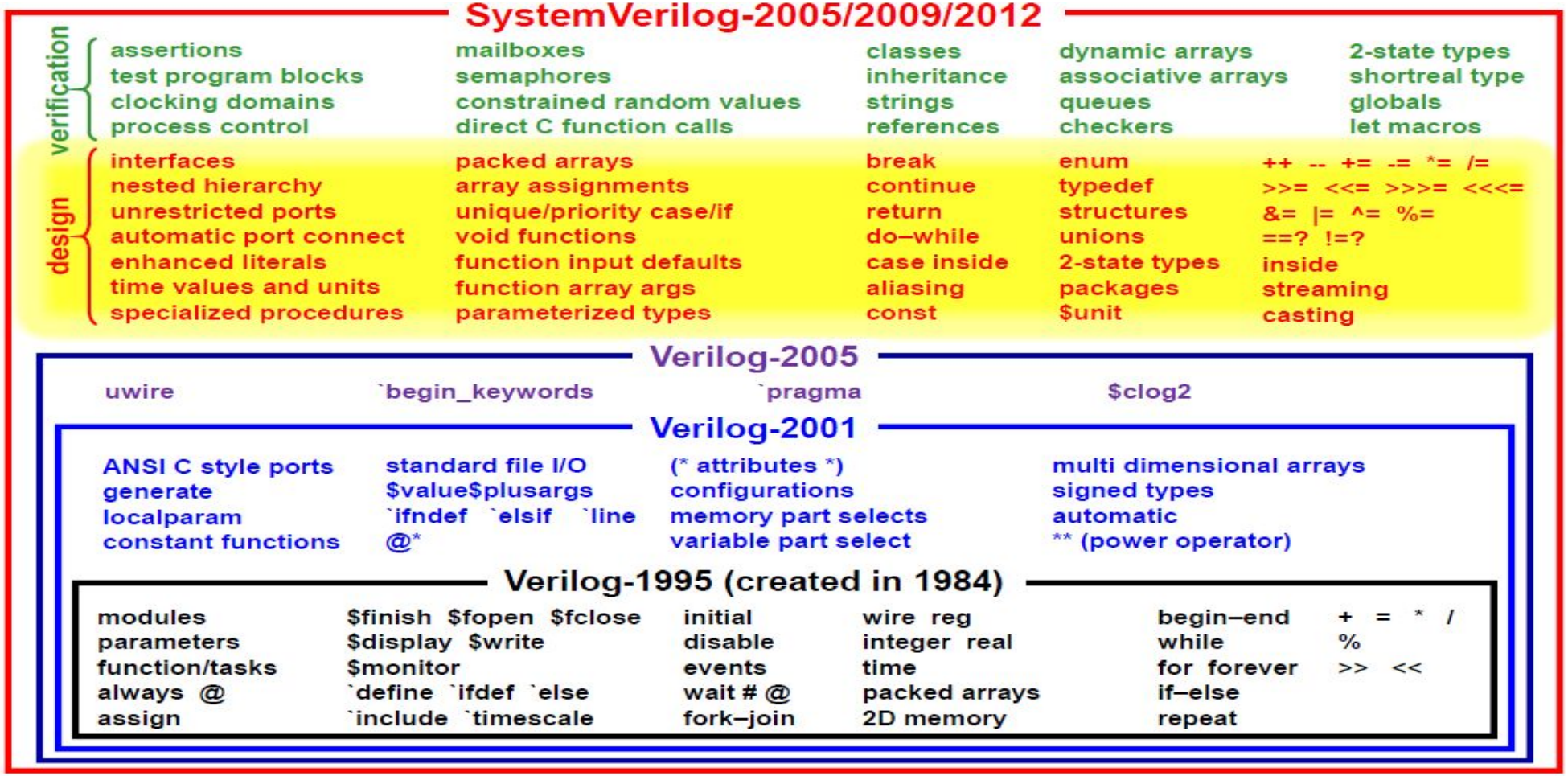
sel, din0 and din1 all input signals are in sensitivity list

Solution 2: Specify all signals in the argument list of the function call

```
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
     mux_out = func_mux(sel, din0, din1);
end
endmodule: mux
```

sel, din0 and din1 are automatically inferred in sensitivity list

Solution 3: New Standard?



Procedural Block Types with always

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

Combinational Logic with 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.

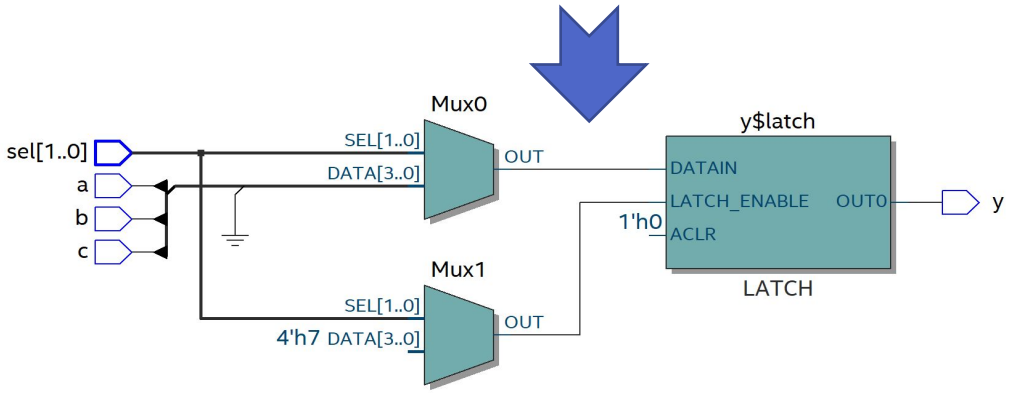
Rule 1 for always_comb: Incomplete case Statements

- This avoids unintentional latches

```
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
```

```
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 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 !

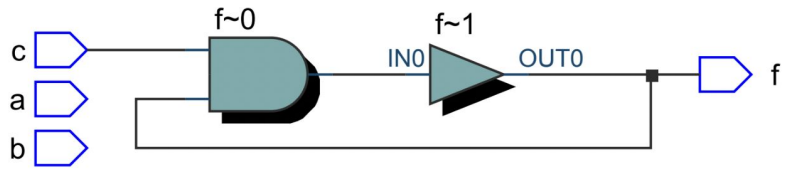


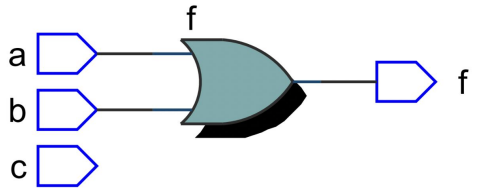
Rule 2 for always_comb: No Multiple Drivers

- Avoids ambiguous netlists

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

always_comb begin
    f <= a | b;
    f <= f & c;
end
endmodule: non_blocking_comb
```

Not allowed!





Sequential 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
- Examples:

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

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

Sequential always_ff Block Rules

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

Quiz – which one is not synthezising correctly and why?

```
module shift_register_left ( input logic clk, input logic reset,
input logic data_in, output logic [3:0] data_out );
logic [3:0] shift_reg;
always_ff @(posedge clk or posedge reset)
        begin
        if (reset)
                begin
                shift_reg[3] = 1'b0;
                shift_reg[2] = 1'b0;
                shift_reg[1] = 1'b0;
                shift_reg[0] = 1'b0;
                end
        else begin
                shift_reg[3] = shift_reg[2];
                shift_reg[2] = shift_reg[1];
                shift_reg[1] = shift_reg[0];
                shift_reg[0] = data_in;
                end
        end
assign data_out = shift_reg;
endmodule
```

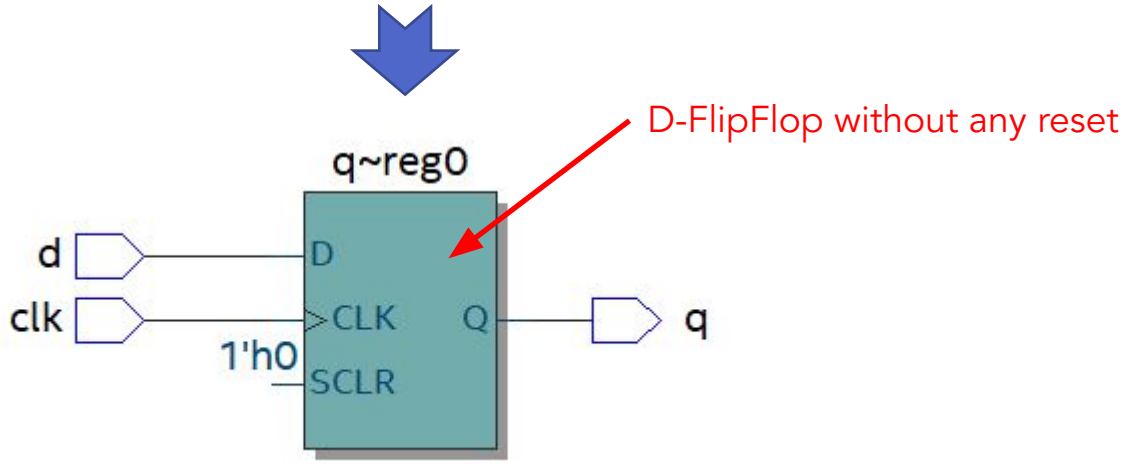
```
module shift_register_right ( input logic clk, input logic reset,
input logic data_in, output logic [3:0] data_out );
logic [3:0] shift_reg;
always_ff @(posedge clk or posedge reset)
        begin
        if (reset)
                begin
                shift_reg[3] = 1'b0;
                shift_reg[2] = 1'b0;
                shift_reg[1] = 1'b0;
                shift_reg[0] = 1'b0;
                end
        else begin
                shift_reg[0] = data_in;
                shift_reg[1] = shift_reg[0];
                shift_reg[2] = shift_reg[1];
                shift_reg[3] = shift_reg[2];
                end
        end
assign data_out = shift_reg;
endmodule
```

D-Flip Flop Model without reset with always_ff

```
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-Flip Flop Model with reset with always_ff

D-FlipFlop with Synchronous Active High Reset

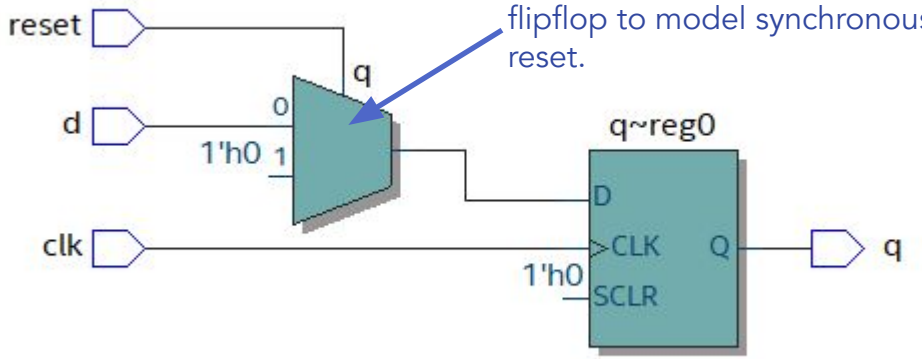
```
module dff(
 input logic clk, d, reset,
 output logic q
);
always_ff@(posedge clk) begin
if(reset)
  q <= 0;
else
  q <= d;
end
endmodule: dff
```

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.

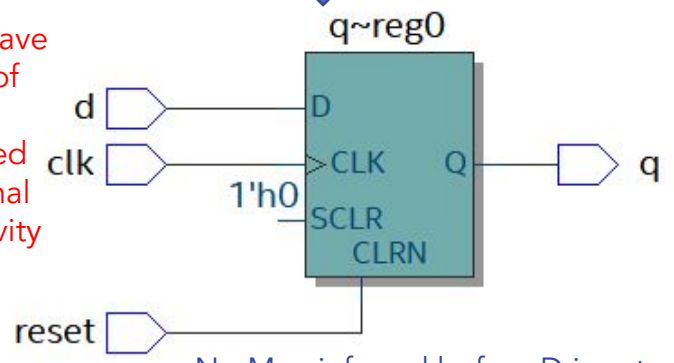


D-FlipFlop with Asynchronous positive Edge Reset

```
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
```

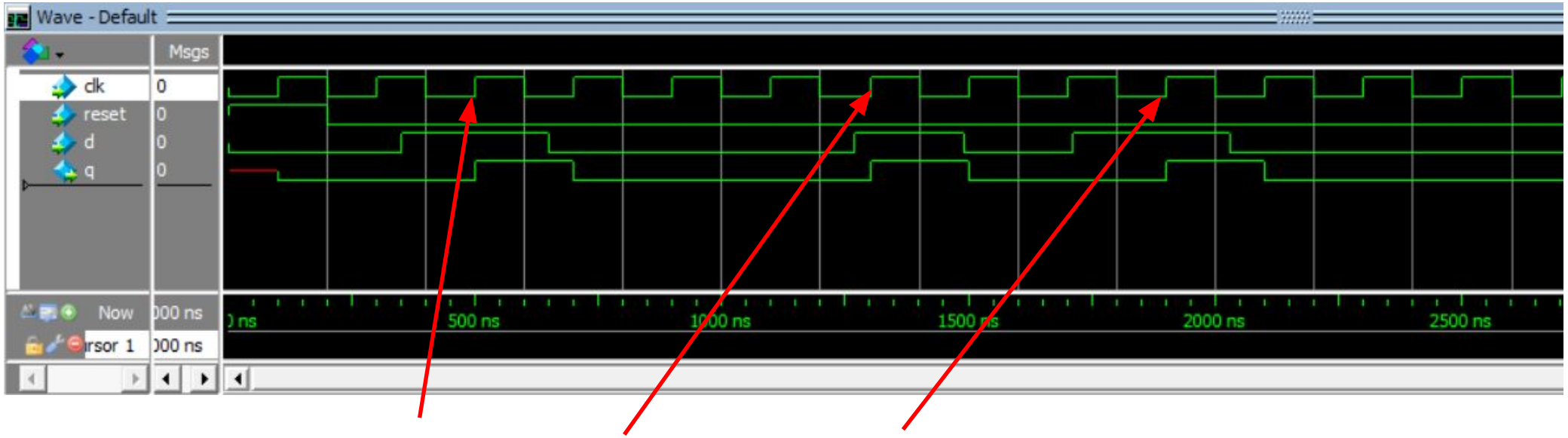
- Having reset in sensitivity list indicates to synthesizer to create asynchronous reset.
- Change in reset from '0' to '1' anytime will cause procedural statements within always_ff block to evaluate

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

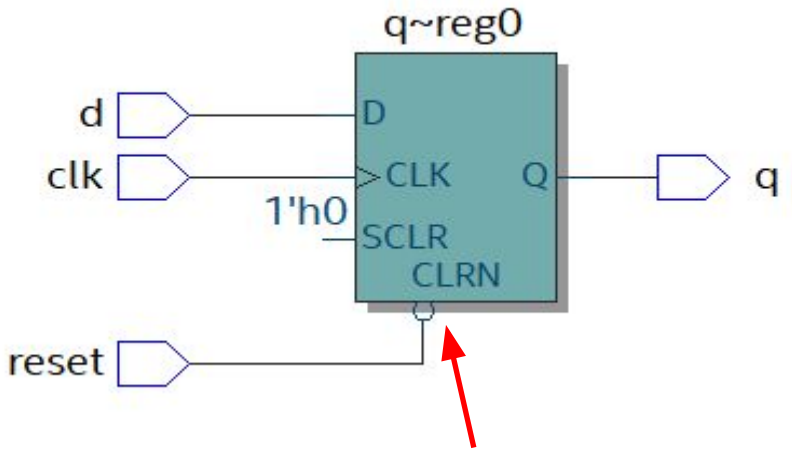
Simulation Results for D-FF with synch reset



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

D-FF 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 the synthesizer due to negative edge reset mentioned in D-Flipflop model

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

Mixing Single and Double Edge in Sensitivity is **Not Allowed**

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
```