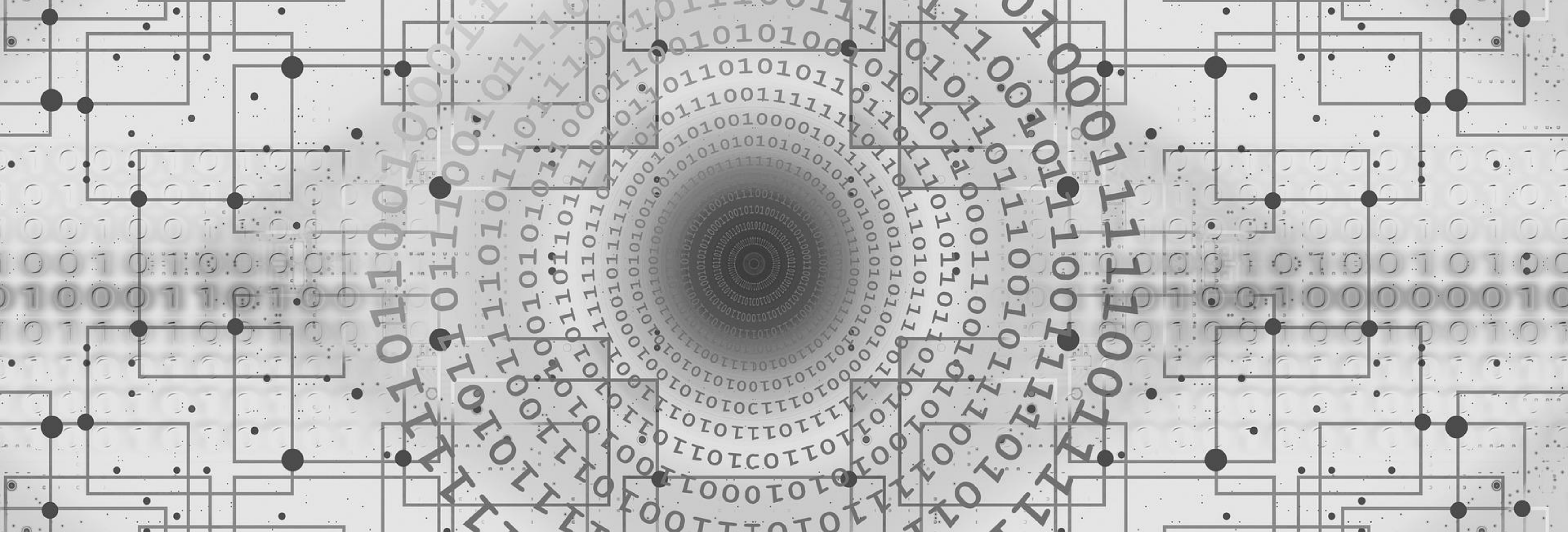


Lecture-6 & 7: Blocking and Non-blocking Assignments

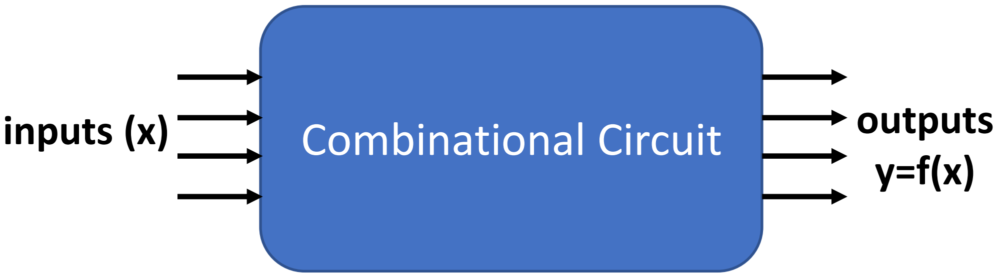
ECE-111 Advanced Digital Design Project

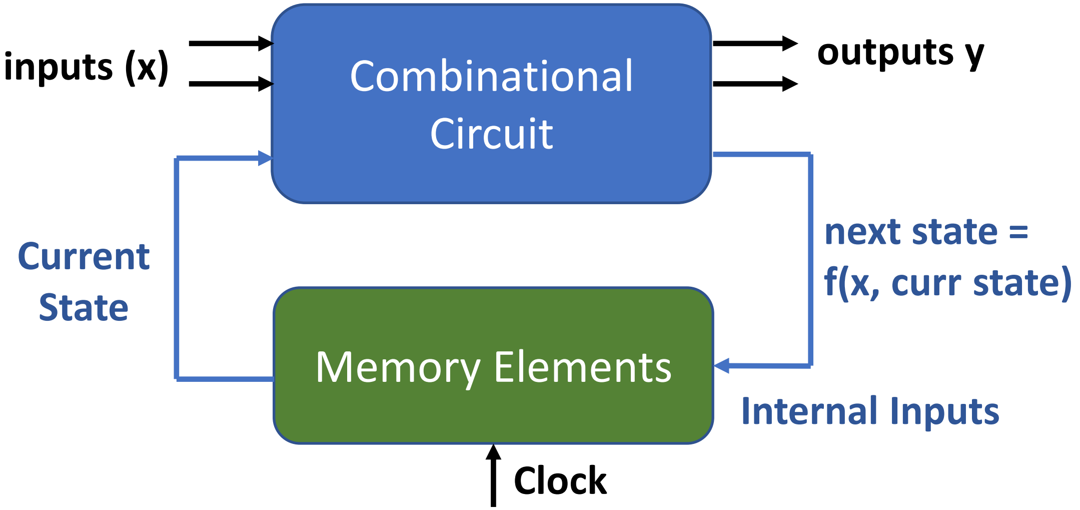
Vishal Karna

Winter 2022

UC San Diego
JACOBS SCHOOL OF ENGINEERING
Electrical and Computer Engineering

Combinational vs Sequential Circuit





Combinational Circuit

- At any instance of time present value of outputs depends solely on present value of inputs
- Does not store any intermediate values and hence does not require any memory elements
- Behavior is described by set of output functions
- **Examples:** Full Adder, half adder, comparator, multiplexer, decoder, encoder, etc

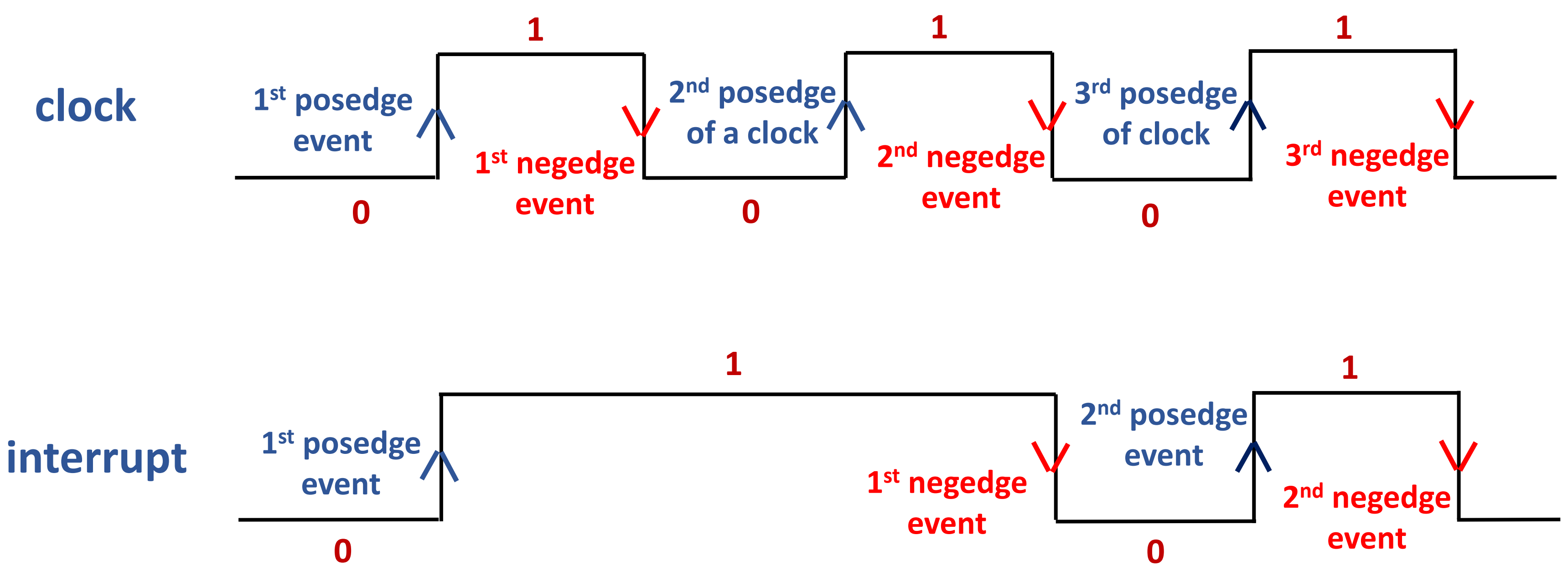
Sequential Circuit

- Present values of outputs are determined from present values of inputs and past state (i.e. sequence of past inputs or known as past outputs)
- Behavior is described by set of output functions and set of next states functions stored in memory
- It contains memory elements to store past outputs
- **Examples:** Flipflop, Latch, Shift Register, etc

posedge and negedge event

- ❑ Positive edge (**posedge**) defines a **rising edge** of a signal
- ❑ Posedge event triggers when a signal transitions from:
  - 0 to 1
  - 0 to X
  - X to 1
  - 0 to Z
  - Z to 1

- ❑ Negative edge (**negedge**) defines a **falling edge** of a signal
- ❑ Negedge event triggers when a signal transitions from:
  - 1 to 0
  - 1 to X
  - X to 0
  - 1 to Z
  - Z to 0



Overview of Always Block

```
always@(<sensitivity list>) begin
  <procedural statements>
end
```

```
always@(a or b) begin
  c = a ^ b; // executes if value of 'a' or 'b' changes
end
```

- **always procedural blocks are used to describe events that should happen under certain conditions**
  - whenever any event in the sensitivity list occurs, the procedural statements are executed
  - sensitivity list can have one or more signals specified
  - **always block runs continuously throughout the simulation !**
- **Event in sensitivity list can be specified in multiple different ways :**
  - Edge (posedge, negedge)
  - Level (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
  - Clocked sequential logic (such as flipflops)
  - Level sensitive logic (such as latches)

- **Sequential Logic is triggered by a 'CLOCK' event**
  - Latches are sensitive to level of the signal
  - Flip-flops are sensitive to the transitioning of clock
    - Synthesis compiler will infer a flip-flop if sensitivity list has **posedge** or **negedge** event

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

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

Always block triggers upon rising edge of clk. posedge CLOCK event and "<=" will infer storage and sequential logic

Sequential Logic using Posedge CLOCK Event

```
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
```

Always block triggers if either inv or data value changes. No storage created. Purely combinational logic inferred

Combinational Logic

SystemVerilog Procedural Assignments

- **SystemVerilog has two types of procedural assignments to model combinational and sequential circuits/logic**
  - **Blocking Assignments** represented with equal sign (**=**) to model combinational logic
  - **Non-Blocking Assignments** represented with less-than-equal (**<=**) to model sequential logic, such as flip flops, latches, shift registers, etc

Blocking Assignment

❑ Blocking Assignment :

- **Syntax :** LHS Variable_Name = [delay or event control] RHS_Expression;
- **Example :**

```
integer a, b, c, sum, prod;
initial begin
  a=5; b=10; c=4; sum=2; prod=8;
  sum = a + b;
  prod = sum * c;
end
```

Statements with Blocking assignment will execute one at a time in the order it is specified. Hence **sum = a + b** line will block execution of line **prod = sum *c** until a + b addition is computed and new value of addition is assigned to variable sum

Simulation Result

- **Initial values of a=5, b=10, c=4, sum=2, prod=8**
- **sum becomes (5 + 10) = 15**
- **prod becomes (15 * 4) = 60**

- **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.
- Execution flow within the procedure is blocked until the current assignment is completed
  - Hence blocking assignment statement is used to model **combinational logic**.
- Each blocking assignment statement executes sequentially in the order it is specified in a procedural block

Non-Blocking Assignment

❑ Non-Blocking assignment :

- **Syntax :** LHS Variable_Name **<=** [delay or event control] RHS_Expression;
- **Example :**

```
integer a, b, c, sum, prod;
initial begin
  sum <= a + b;
  prod <= sum * c;
end
```

Both statements with non-blocking assignment will execute concurrently without blocking each other

Simulation Result

- **At time 0 ns, Initial values a=5, b=10, c=4,**
- **At time 1 ns,**
  - **sum becomes (5 + 10) = 15**
  - **prod becomes (2 * 4) = 8**

- **Evaluation and assignment is a two-step process**
  - Expression on RHS of (**<=**) assignment is evaluated immediately
  - Assignment to LHS variable is postponed until other evaluations in **current time step** is completed
  - Latest RHS value is assigned to target LHS variable at the end of the simulation cycle
- Each non-blocking assignment statement within the procedural block executes in parallel (concurrently) without blocking each other
  - Hence the **order of specifying non-blocking assignment statement does not matter !**
- Clock-to-Q propagation delay of flip-flop behavior is represented by non-blocking assignments in RTL model even though RTL code is modeled with zero-delays.
  - Hence non-blocking is used to model **sequential logic**

Blocking vs Non-Blocking Assignment

```
module blocking_assignment (
  input logic clock, a,
  output logic b
);
always@(posedge clock)
 begin
   a = 1;
   // a is '1'
   b = a;
   // b is now '1' as well
 end
endmodule
```

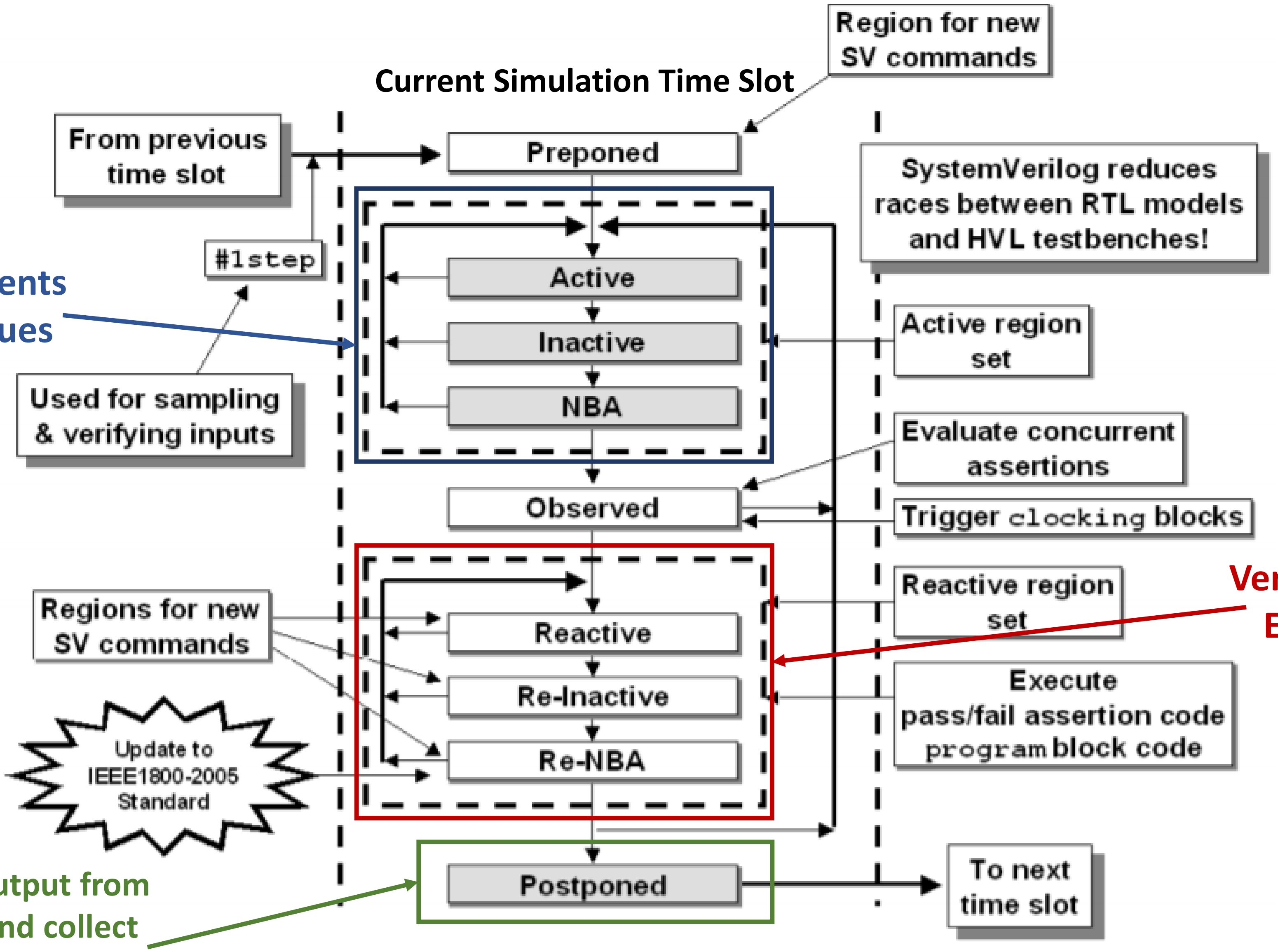
```
module non_blocking_assignment (
  input logic clock, a,
  output logic b
);
always@(posedge clock)
 begin
   a <= 1;
   b <= a;
   // all assignments are made
   // b is not yet '1'
 end
endmodule
```

- **Value is assigned immediately.**
- **Process waits until the first assignment is complete, it blocks progress**

- **Values are assigned at the end of the block.**
- **All assignments are made in parallel, process flow is not-blocked.**

SystemVerilog RTL Modeling and Verification Events Scheduling Flow

RTL Modeling Events Scheduling Queues



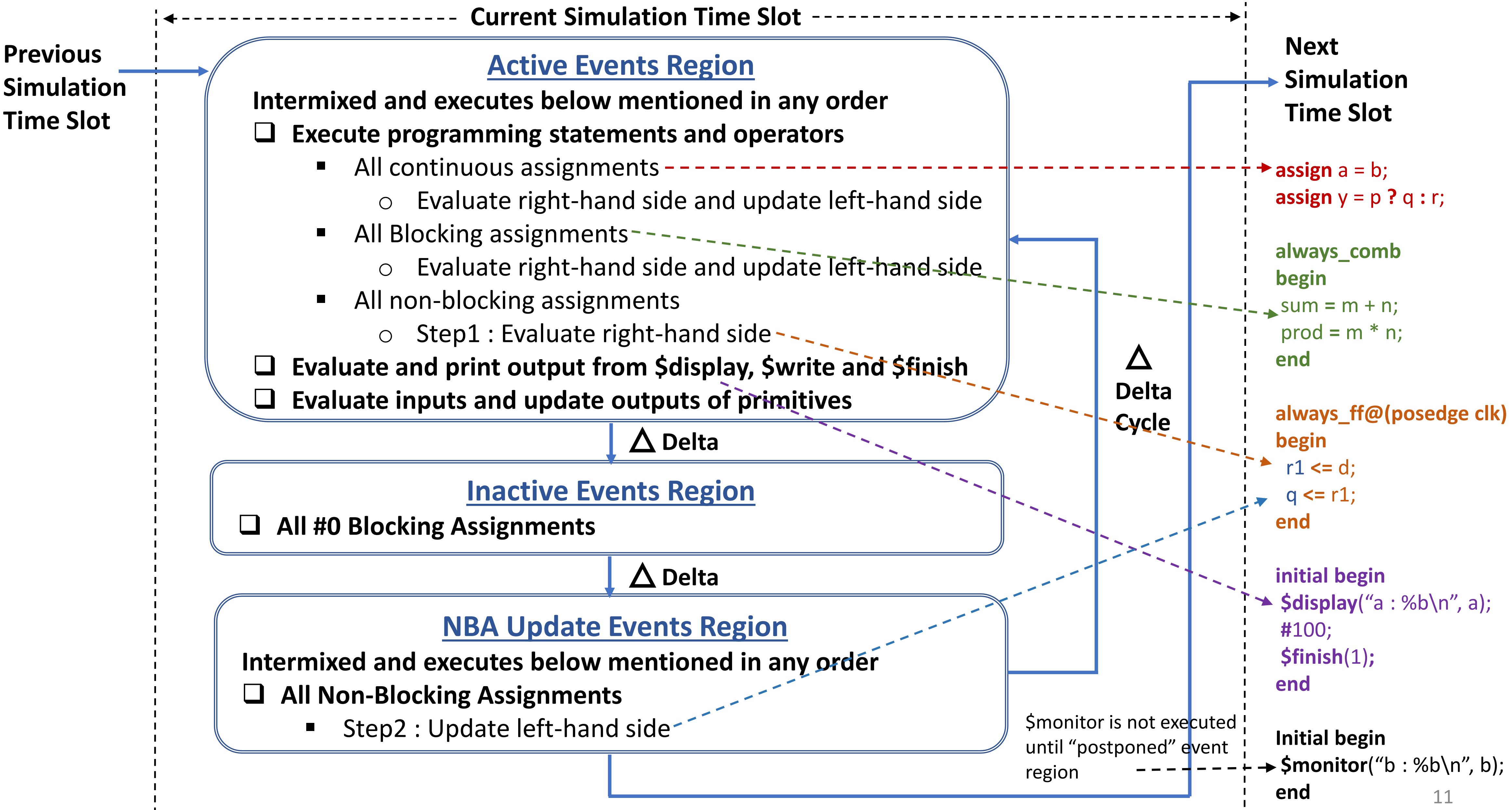
Verification Testbench Events Scheduling Queues

Evaluate and print output from $monitor, $strobe and collect functional coverage for items using strobe sampling

Figure 5 - SystemVerilog-2005 event regions with PLI regions omitted

Credit : Sunburst design, Clifford E. Cummings and Synopsys Artulo Salz

SystemVerilog RTL Modeling Events Scheduling Flow



Credit : Stuart Sutherland, Source : RTL Modeling with SystemVerilog for Simulation and Synthesis Book

Swapping of variables using blocking and non-blocking assignment

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

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

Simulation Result

- ❑ Both always blocks will execute concurrently and there is a **race condition** between two always procedural assignments
- ❑ Assume Initial Value of **p**=5 and **q**=8
  - If simulator executes always block with p = q first, before q = p, then both p and q will get value of q (Final values : **p**=8 and **q**=8)
  - If simulator executes always block with q = p first, before p = q, then both p and q will get value of p (Final values : **p**=5 and **q**=5)
  - **No Swapping of values of p and q !!**

```
always@ (posedge clock)
  p <= q;

always@ (posedge clock)
  q <= p;
```

Simulation Result

- ❑ Both always blocks will execute concurrently, however there is no race condition and **actual swapping of values of p and q happens !!**
- ❑ RHS variables are read first and assigned to LHS at the end of the simulation cycle and new RHS variable is available at the next posedge of the clock
- ❑ Assume Initial Value of **p**=5 and **q**=8
  - p gets previous value of q, hence **p**=8
  - q get previous value of p, hence **q**=5

Swapping of variables using blocking assignment

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

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

Simulation Result

- Simulator executes p = q statement first and then executes q = p.
- Assume Initial Value of **p**=5 and **q**=8
  - Both p and q will get value of q. **There is no swapping of values of p and q !!**
  - Final value of **p**=8 and **q**=8

Simulation Result

- **Swapping of values of p and q happens !!**
- To swap values of p and q using blocking assignment requires temporary variables as shown above
- Assume Initial value of **p**=5 and **q**=8
  - tmp1 will get assigned 5
  - Tmp2 will get assigned 8
  - p will get assigned value of tmp2, which is 8, same as value of q
  - q will get assigned value of tmp1, which is 5, same as previous value of p

Sequential Procedural Assignments with Inter-Delay

- A sequential **blocking assignment** evaluates and assigns before continuing within procedure 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**

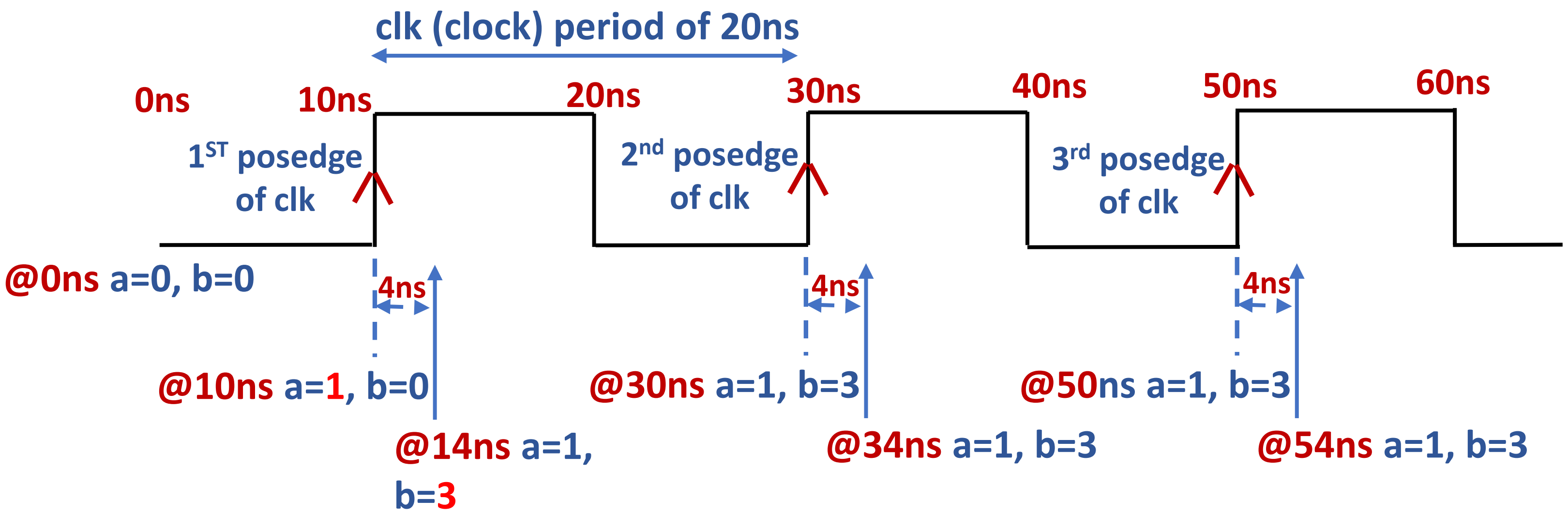
always@(posedge clk) begin
a = 1; → Evaluate RHS and assign to 'a' immediately
#4 b = a + 2; → After execution of previous statement, advance 4 time units due to inter assignment delay, then evaluate RHS (a+2) and assign it to 'b' immediately
end

Simulation Output

Assume At **0ns** a=0, b=0
At 1st posedge of clk a=1
At 1st posedge of clk+**4ns** b=3

equivalent

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



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

Sequential Procedural Assignments with Inter-Delay

- A sequential **non-blocking** assignment evaluates, then continues on to the next timing control before next non-blocking assignment evaluates. And then finally updates all LHS variable
  - Timing control before an 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;          → 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
end                  assign to 'b' at the end of time step (1 clock period + 4ns)
```

Simulation Output

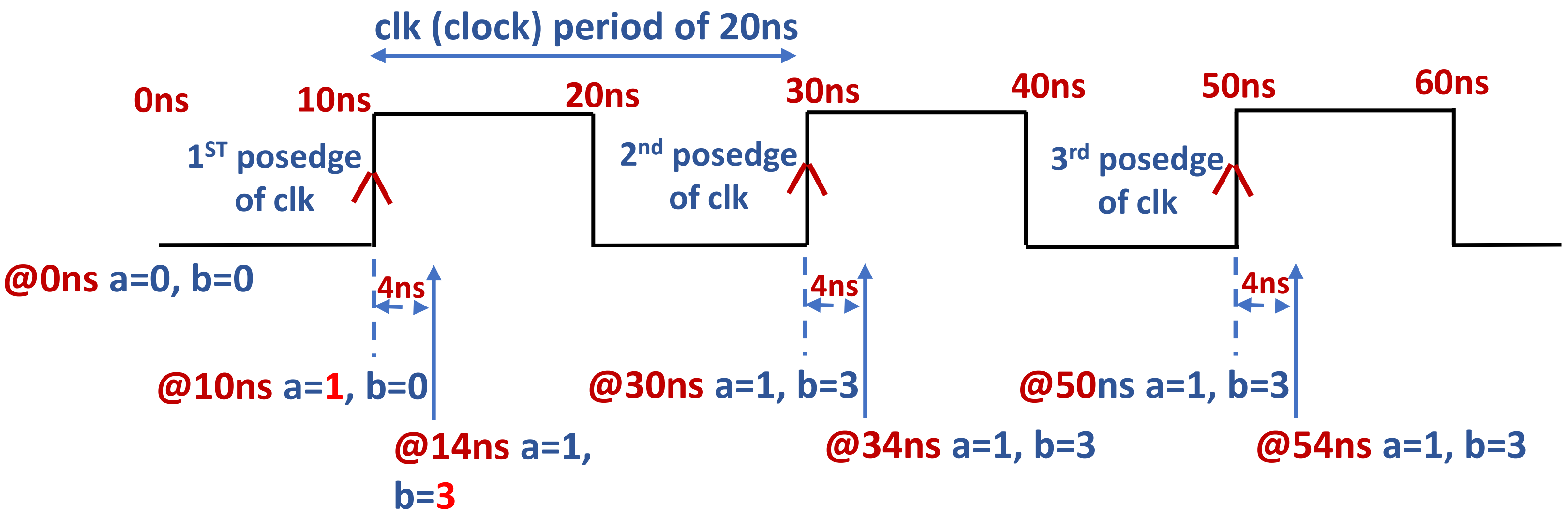
Assume At **0ns** a=0, b=0

At 1^{st} posedge clk**,** a=1

At 1^{st} posedge clk+**4ns**, b=3

equivalent

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



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 Assignments with Inter-Delay

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

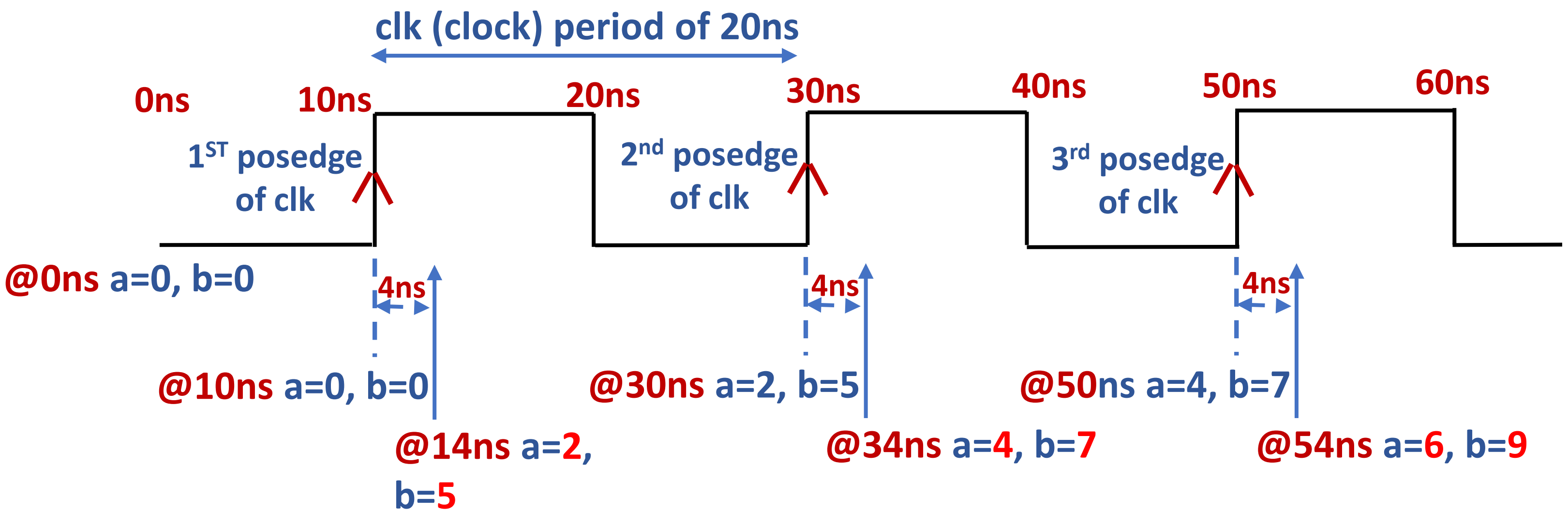
```
always@(posedge clk) begin
  #4 a = a + 2;   → Delay 4 time units due to inter assignment delay, then evaluate RHS
end                 and assign to 'a', immediately

always@(posedge clk) begin
  #4 b = a + 3;   → Delay 4 time units due to inter assignment delay, then evaluate RHS
end                 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 Assignments with Inter-Delay

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

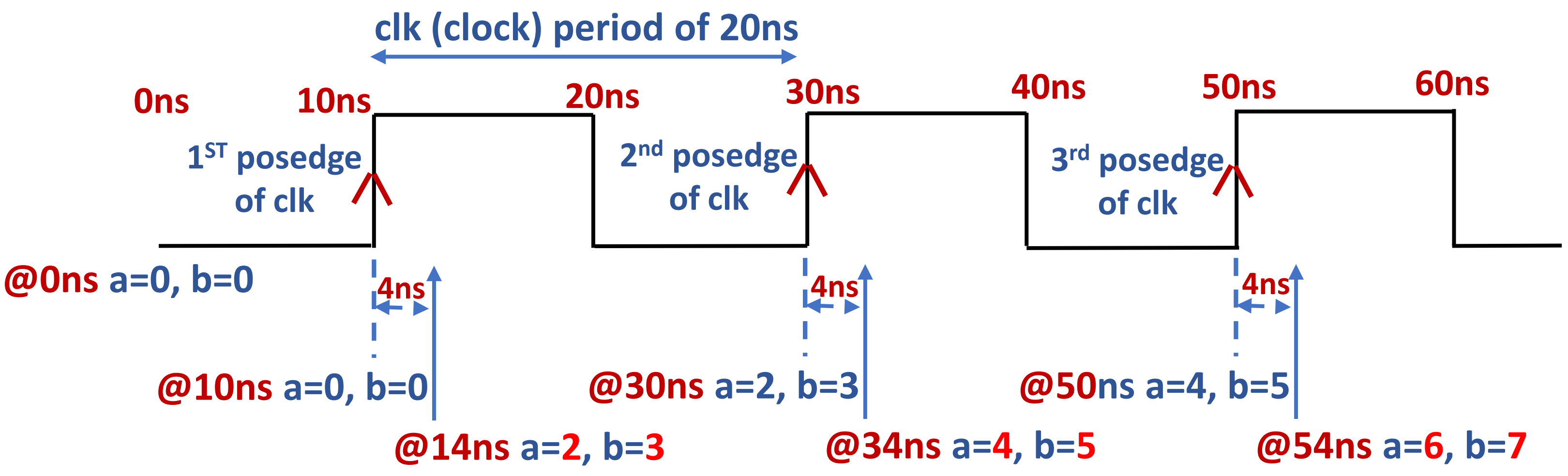
```
always@(posedge clk) begin
  #4 a <= a + 2;   → Delay 4 time units due to inter assignment delay, then evaluate RHS;
end                  then assign to 'a' at the end of time step (1 clock period + 4ns)

always@(posedge clk) begin
  #4 b <= a + 3;   → Delay 4 time units due to inter assignment delay, then evaluate RHS;
end                  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 Assignments with Intra-Delays

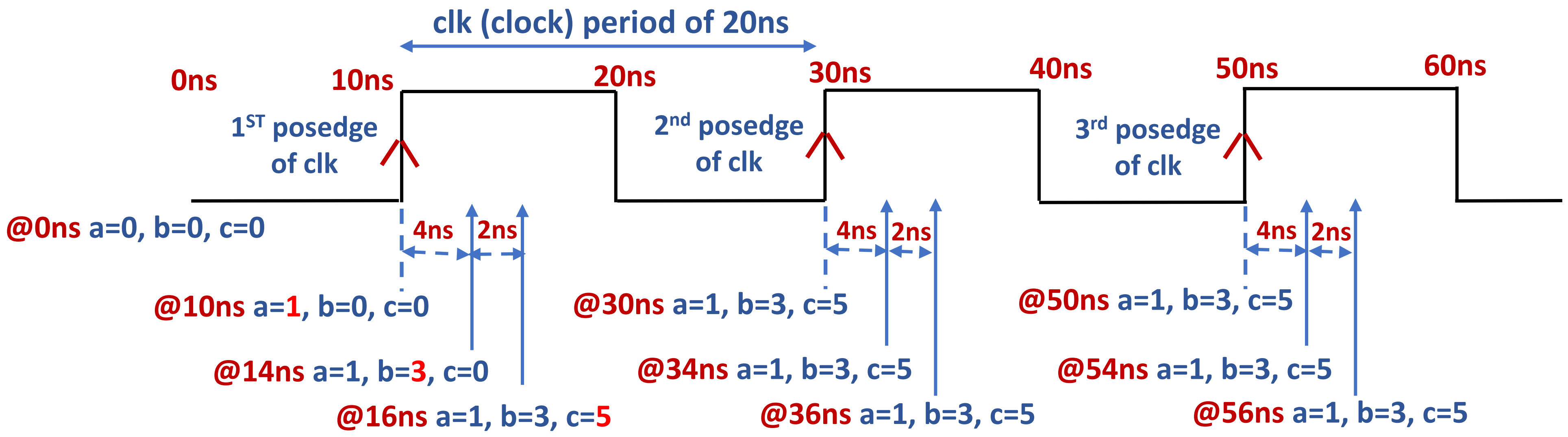
- An intra-assignment delay places the timing control after the assignment token
  - Right-hand side is evaluated before the delay
  - Left-hand side is assigned after the delay

- A sequential **blocking** assignment with intra-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+4**ns,** b=3
At 1st posedge clk+6**ns,** c=5



Procedural Assignments with Intra-Delays

❑ A sequential **non-blocking** assignment with intra-delay

```
always@(posedge clk) begin
  a <= 1;        → Evaluate RHS immediately; then assign to 'a' at the end of the time step
  b <= #4 a + 2; → Evaluate RHS expression immediately, then after intra delay of 4 time units
                   assign to 'b'
  c <= #2 b + 2; → Evaluate RHS expression immediately, then after intra delay of 2 time units
end                assign to 'c'
```

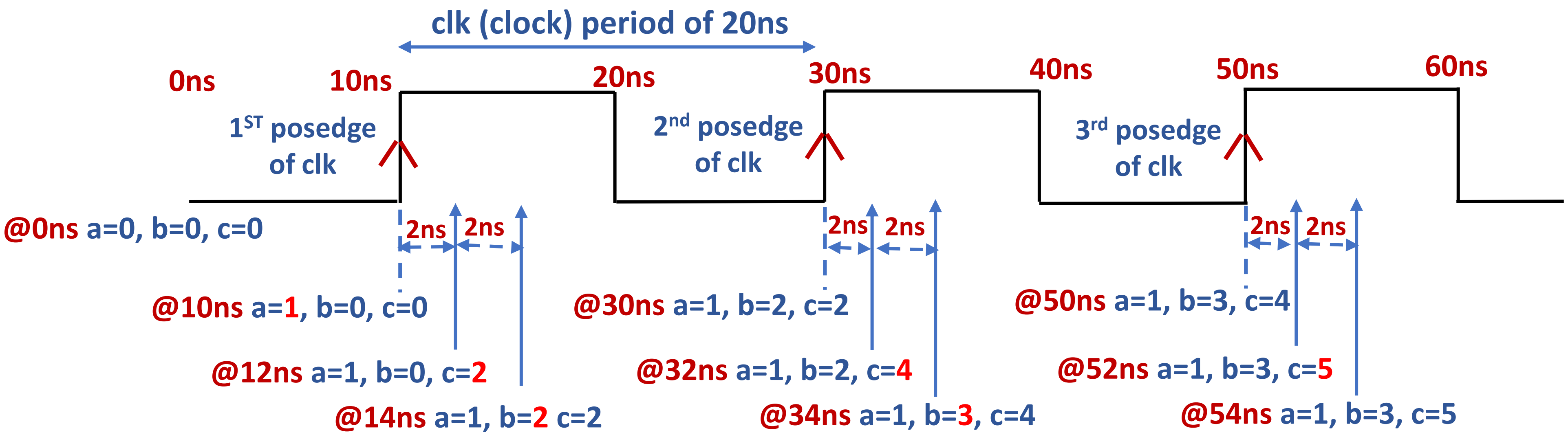
Simulation Output

Assume At **0ns** a=0, b=0

At 1st posedge clk**,** a=1

At 1st posedge clk+2**ns,** c=2

At 1st posedge clk+**4ns**, b=2



Rules

- It is not recommended to have both blocking and non-blocking assignment statements in same always block.

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

- Synthesis compiler will ignore inter and intra delays in both blocking and non-blocking procedural assignment statement
  - If delays are used, then expect mis-match between RTL and synthesized netlist simulation result

```
always@(a,b) begin
  c = #2 (a + b); //#2 delay ignored by synthesizer, however in RTL simulation effect of #2 will be observed
end
```

- Same variable cannot have both blocking and non-blocking assignments to it. Below mentioned is not allowed !!

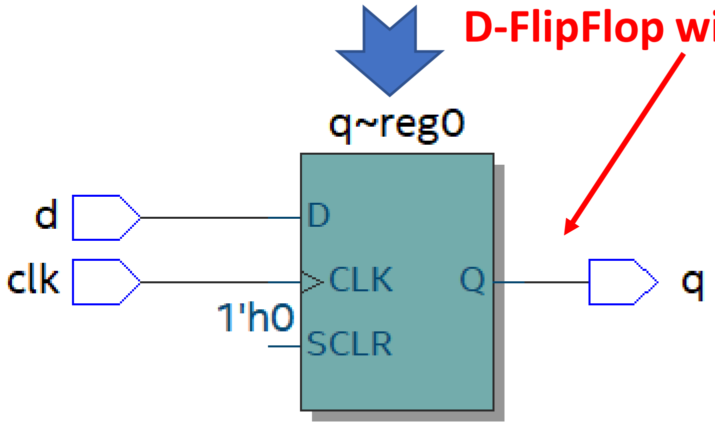
```
always@(a,b) begin
  c <= a + b;
  c <= a * b;
end
```

D-FlipFlop Model Using Non-blocking assignment

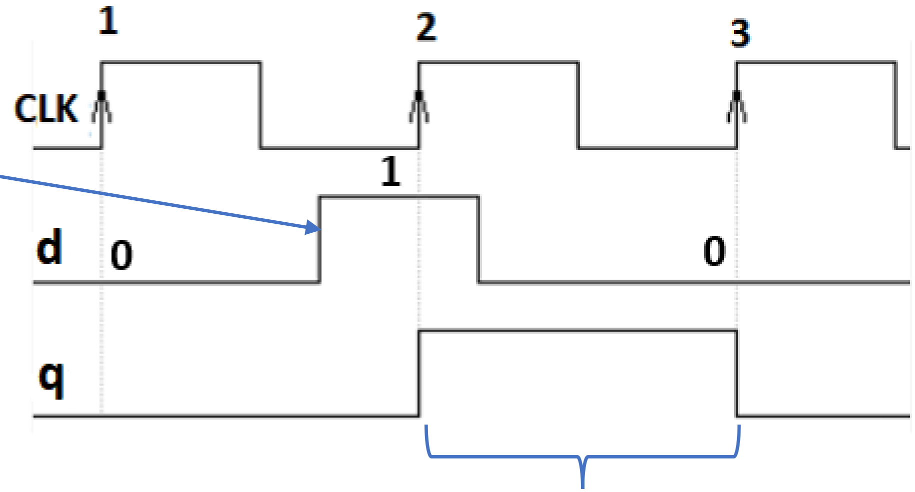
```
module dff(
 input logic clk, d,
 output logic q
);

always@(posedge clk) begin
  q <= d;  // when clk rises copy 'd' to 'q'
end
endmodule: dff
```

- SystemVerilog calls "<=" a "non-blocking" assignment.
- It means "wait until end of current simulation cycle before updating "q".
- When clk is not rising, value of q is preserved (memorized)
- Synthesis will produce a positive edge-trigged D-FF

D-FlipFlop without any reset



Input 'd' = 1 captured at CLK edge '2' will appear at output 'q' and value of 'q'= 1 will then be preserved until CLK edge '3'



Value of 'q' is retained until next edge of clk

Example of blocking vs non-blocking assignment

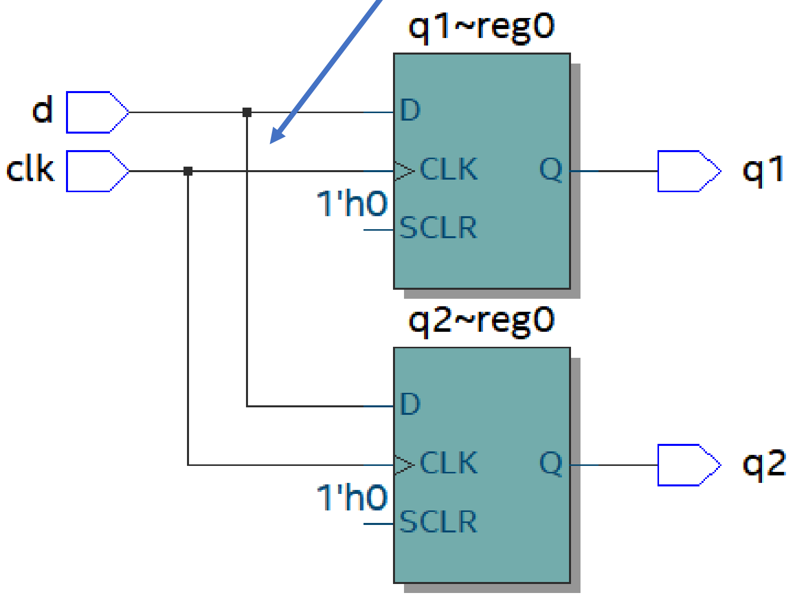
```
module parallel_registers (
  input logic clk, d,    // clk is a clock
  output logic q1, q2
);
always @(posedge clk)
 begin
   q1 = d;
   q2 = q1;
 end
endmodule
```

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

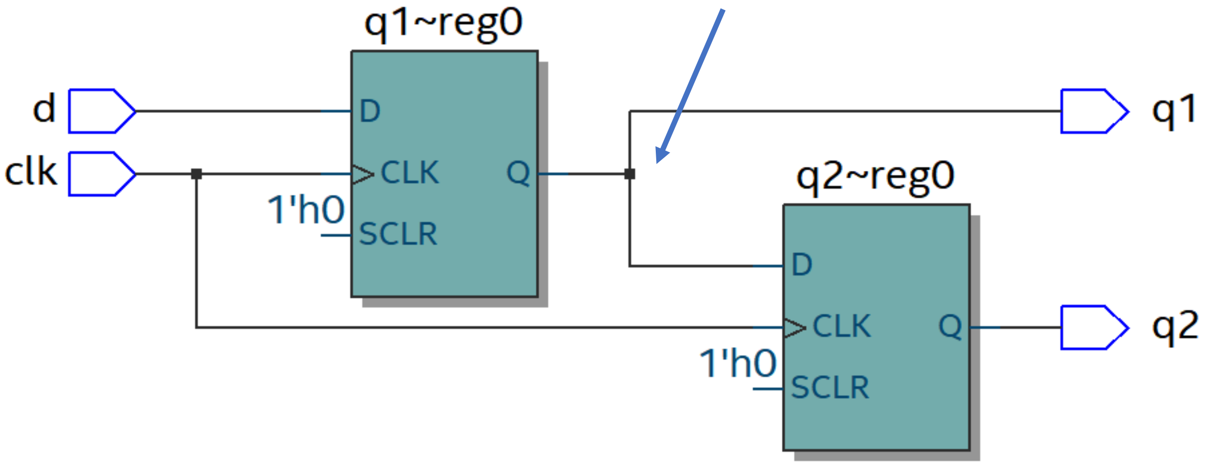
```
module shift_register (
  input logic clk, d, // clk is a clock
  output logic q1, q2
);
always @(posedge clk)
 begin
   q1 <= d;
   q2 <= q1;
 end

endmodule
```

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

Synthesis compiler will create two registers in parallel



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

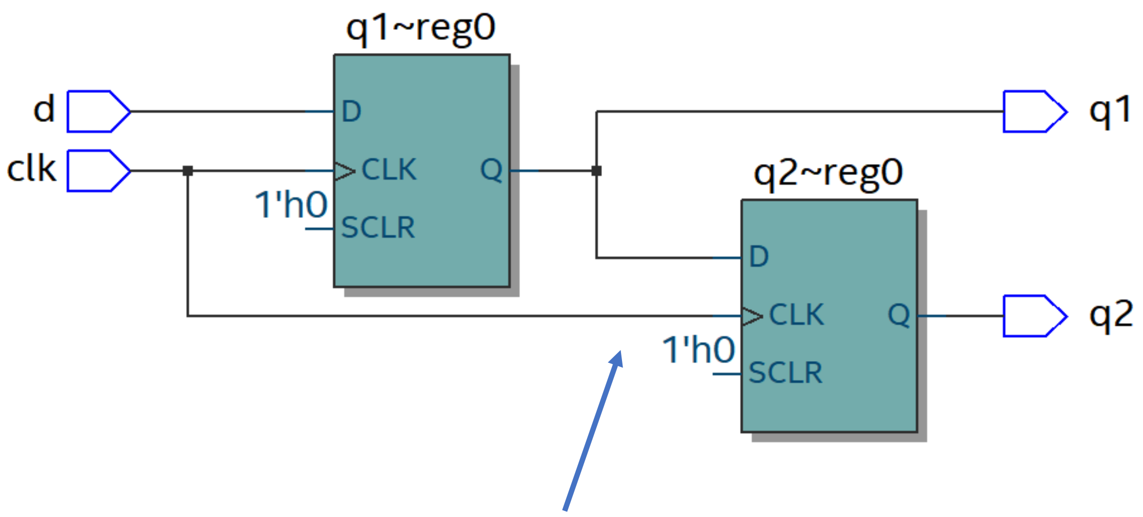


Note : Serially chained register is known as shift register

Splitting Blocking Assignments in Separate Always Block

```
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 sysnthesis 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.

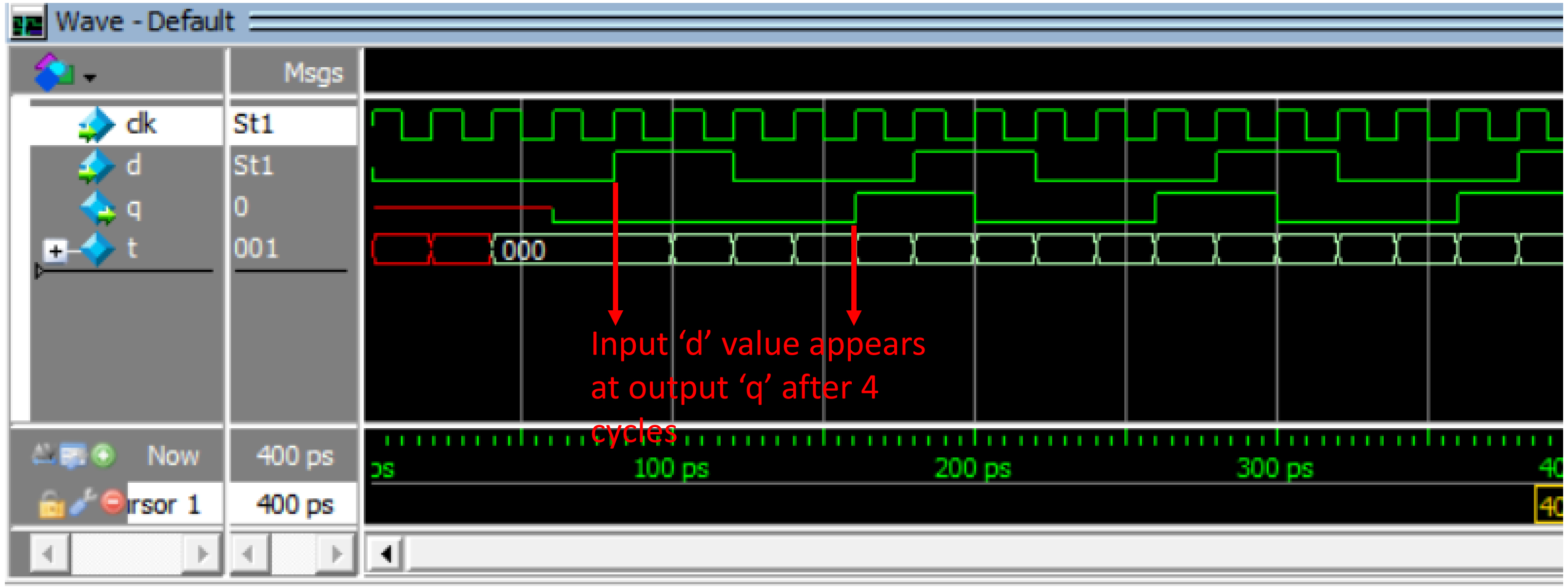
- If always block with q2=q1 assignment executes first over other always block having q1=d then circuit will behave as a 2-bit shift register
- If always block with q1=d assignment executes first over other always block having q2=q1 then circuit will behave as 1-bit parallel register

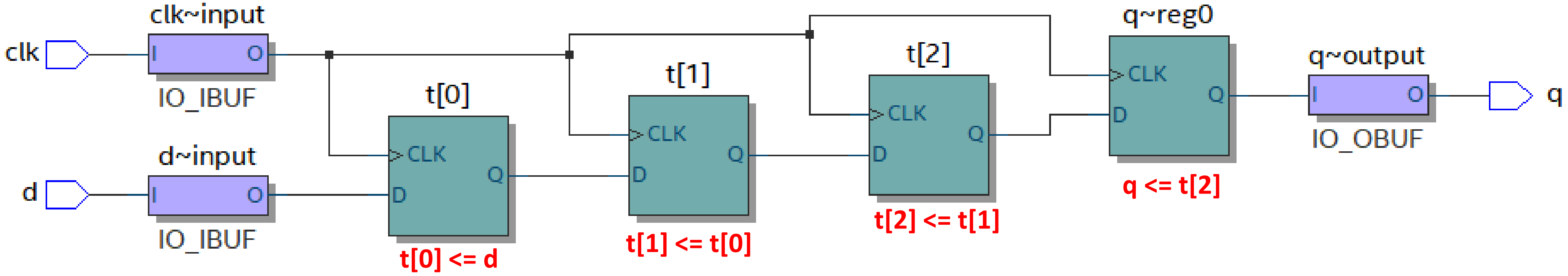
Shift Register using Non-Blocking Assignments

```
module 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
```

Since all RHS expression will be evaluated in parallel, non-blocking assignments can be specified in any order

Simulation Result



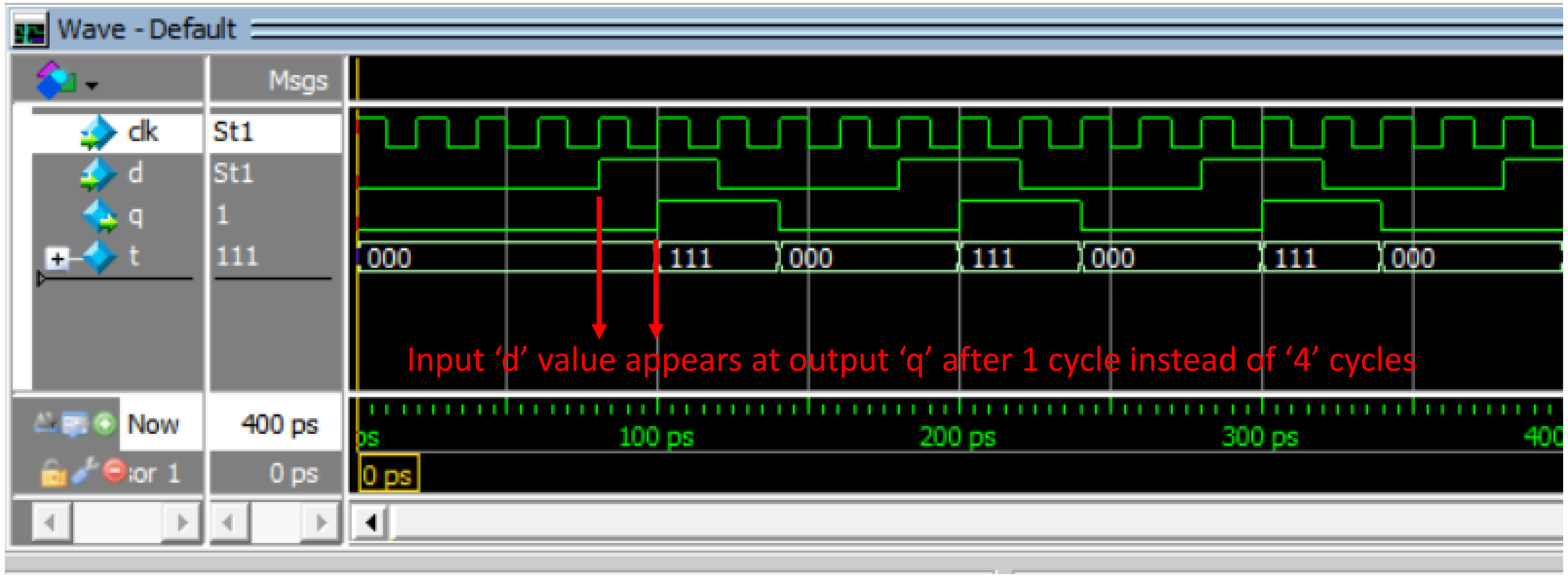


Shift Register using Blocking Assignments (Incorrect Usage)

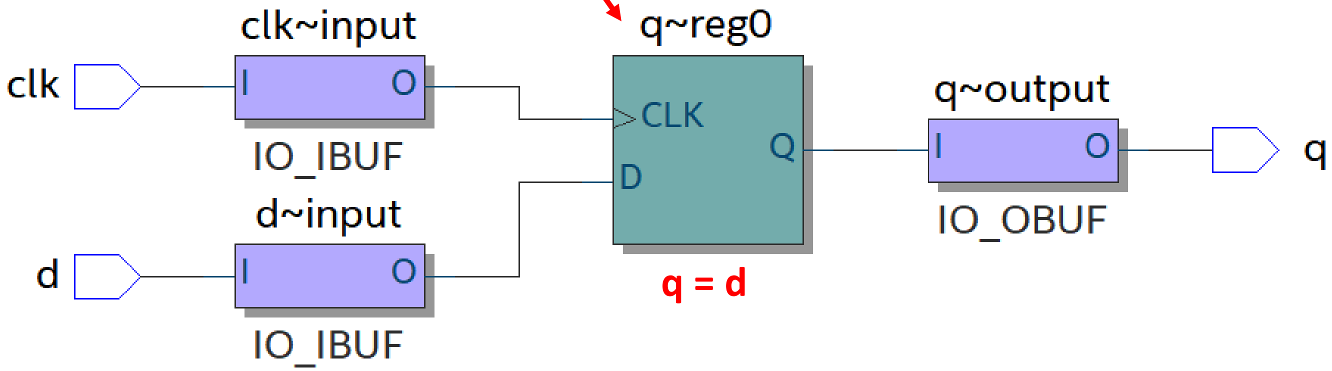
```
module incorrect_shift_register (
  input logic clk, d,
  output logic q
);
 logic[2:0] t;
 always@(posedge clk)
 begin
   t[0] = d;
   t[1] = t[0];
   t[2] = t[1];
     q  = t[2];
  end
endmodule
```

Due to blocking assignment, this code can be seen as d=q and this will behave as a 1- bit register.

Simulation Result



Instead of behaving as a 4-bit shift register, synthesizer will generate as a 1-bit register/flipflop



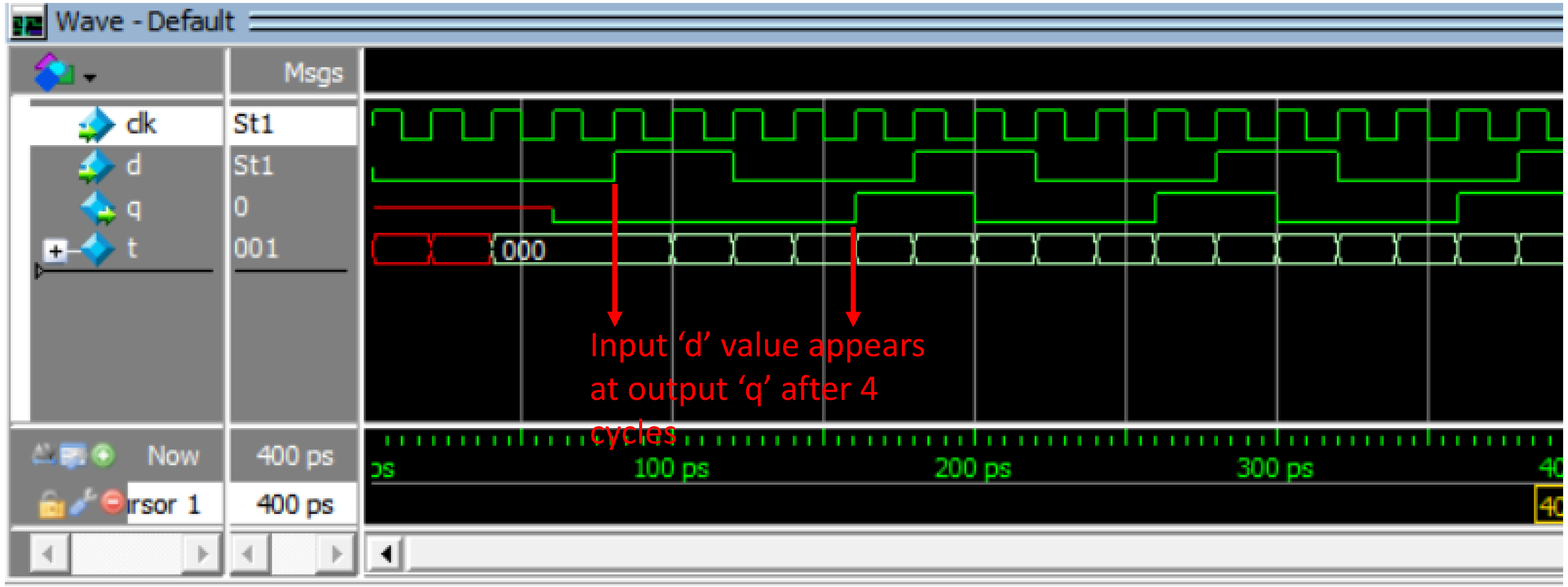
q = d

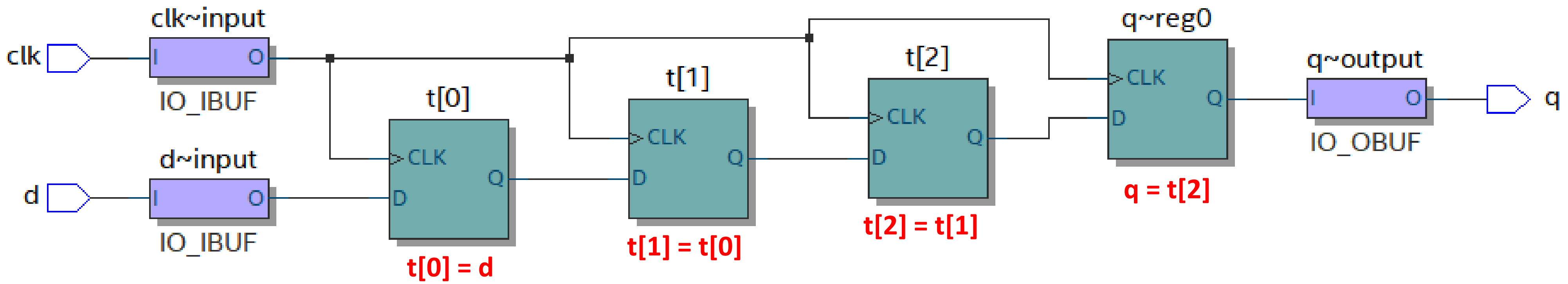
Shift Register using Re-ordered Blocking Assignments

```
module shift_register (
  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
```

Ordering of blocking assignment is important. Re-ordering blocking assignment statements can result in a different circuit, in this case it became a shift register

Simulation Result





References

- **SystemVerilog Event Regions, Race Avoidance & Guidelines :** SNUG Conference 2006
  - **Paper Authors :** Clifford E. Cummings (Sunburst Design), Arturo Salz (Synopsys)
  - http://www.sunburst-design.com/papers/CummingsSNUG2006Boston_SystemVerilog_Events.pdf