Lecture 7: Blocking and Non-blocking Assignments

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:** Tues 10/22 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 Wed Oct 23, **10/23/24**
    - **Late homework policy:** Max of 2 late days, with 20% grade reduction per day

Homework 3 overview

- You will learn how to:
  - Create synthesizable SystemVerilog code
  - Learn about primary ports
  - Better learn how to use testbenches
  - Design functional SystemVerilog code that can post synthesis.
- There will be two parts for this homework:
  - **Homework-3a**: Developing a Synthesizable SystemVerilog Model for a 4-bit Johnson Counter
  - **Homework-3b**: Developing a synthesizable SystemVerilog model of a 4-bit Universal Shift Register.

Review

Review: System Verilog Data

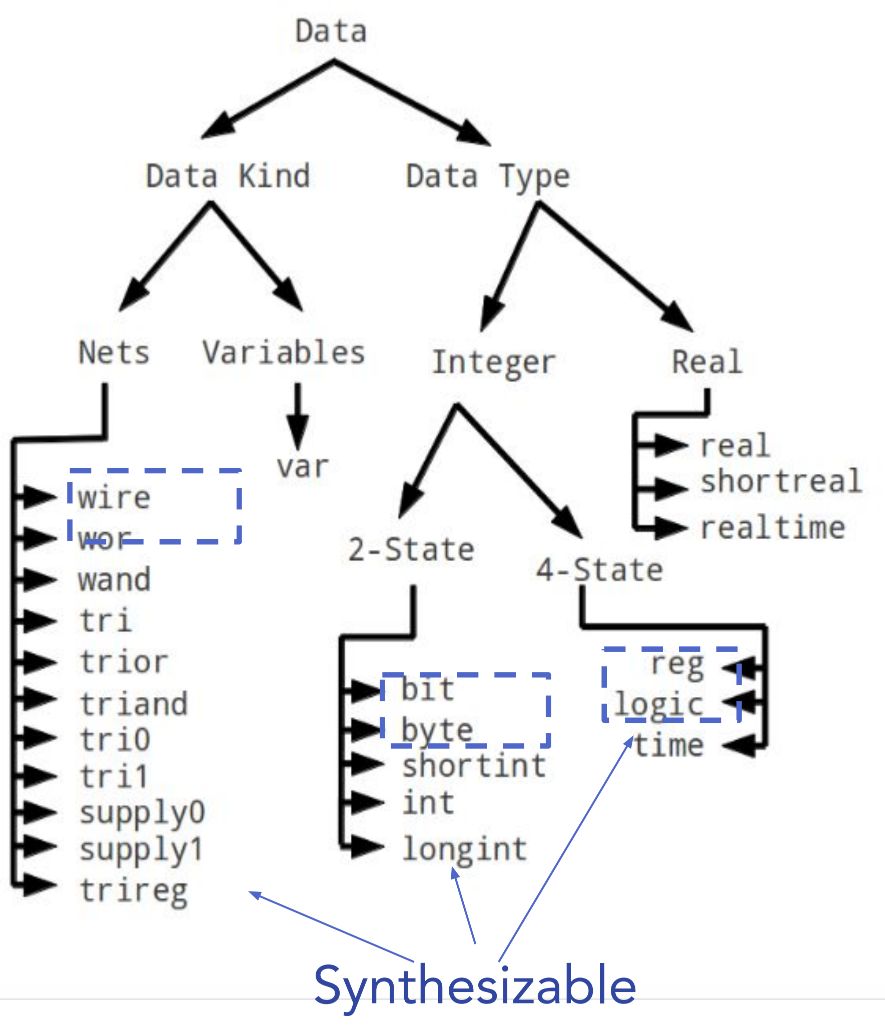
- SystemVerilog data can be specified using two properties : (i) Kind and (ii) Type
  - Data kind refers to usage as net or variable
  - Data type refers to possible values a data kind can take (e.g. integer, float etc.)
- Any data can be declared using the syntax:

  <data_kind> <data_type> <literal_name>

  example: var logic carry_out;

  wire logic [2:0] sum;

  Here :
  - carry_out is declared as a variable which can store 4-state values
  - sum is declared as a net which can store 4-state values
  - logic specification after wire is optional (wire [2:0] sum;)



Truth Table for Some Net Types

Wire/tri

	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	z

Wand/triand

	0	1	x	z
0	0	0	0	0
1	0	1	x	1
x	0	x	x	x
z	0	1	x	z

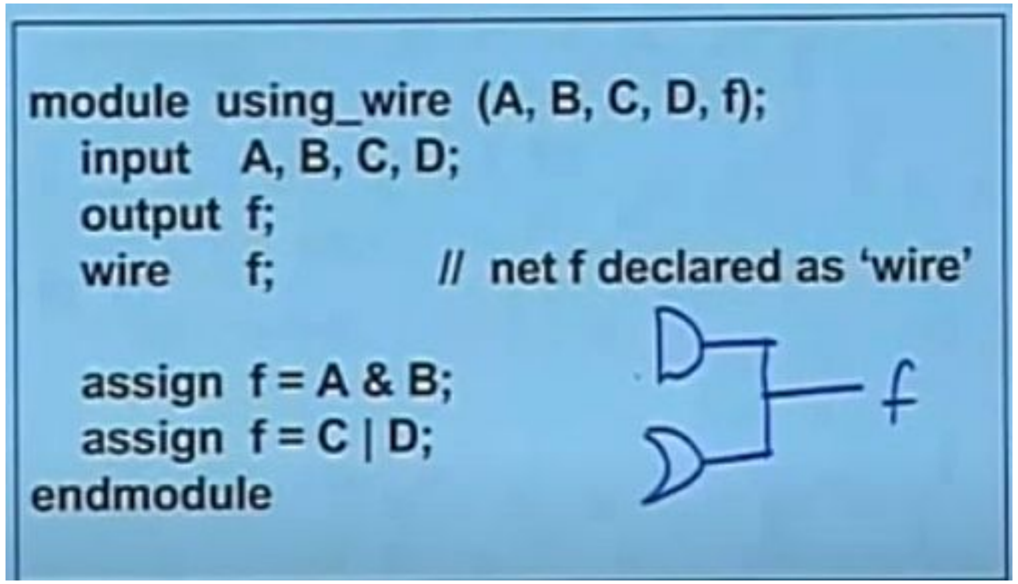
Wor/trior

	0	1	x	z
0	0	1	x	0
1	1	1	1	1
x	x	1	x	x
z	0	1	x	z

tri0

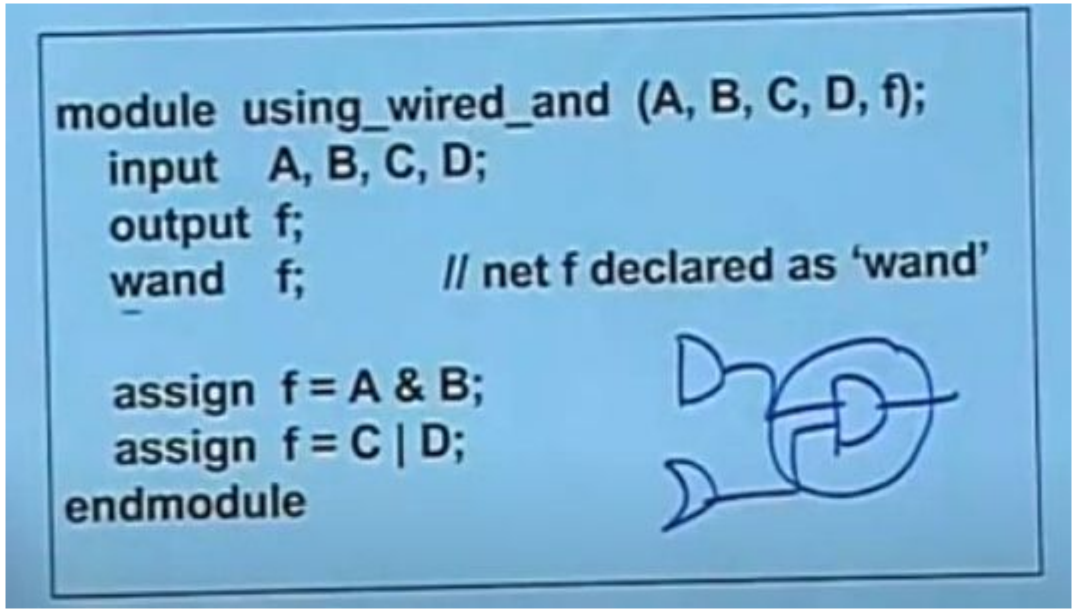
	0	1	x	z
0	0	x	x	0
1	x	1	x	1
x	x	x	x	x
z	0	1	x	0

Wand Data Type Example



```
module  using_wire  (A, B, C, D, f);
    input    A, B, C, D;
    output  f;
    wire      f;            //  net f declared as 'wire'

    assign  f = A & B;
    assign  f = C | D;
endmodule
```

Synthesis tool will give design Error where wire 'f' is assigned from two continuous assign statements !



```
module  using_wired_and  (A, B, C, D, f);
    input    A, B, C, D;
    output  f;
    wand     f;          // net f declared as 'wand'

    assign  f = A & B;
    assign  f = C | D;
endmodule
```

Synthesis tool will connect output of OR and AND gate and explicitly add AND gate at the output.

Use of 2-State Variables with Caution

- Avoid all 2-state data types in RTL modeling

  - 2-state data types can hide design bugs !

  - It can lead to **simulation** vs **synthesis** mismatches
    - **Synthesis** treats bit, byte, shortint, int and longint 2-state data types as a 4-state reg variable. **Simulation** treats as 2-State variable.
    - **Simulation** might start with a value 0 in each bit whereas **synthesized** implementation might power-up with each bit a 0 or 1.
    - Any x or z driven values on **2-state variables**, are **converted to** 0. These data types are initialized to 0 at the start of simulation and may not trigger an event for active low signals.

  - Exception : use 2-state **int** data type variable for the iterator variable in for-loops where x and z is not required.

Verilog Vs. SystemVerilog Data Types

Verilog

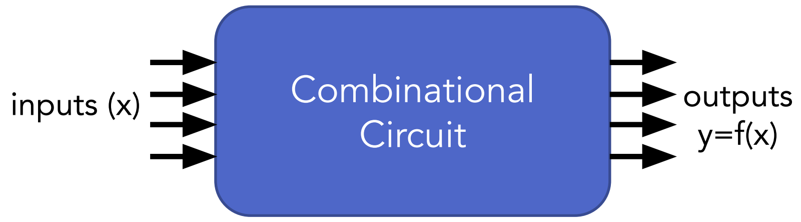
- For synthesizable variables supports only **4-state** (0,1,X,Z) variables

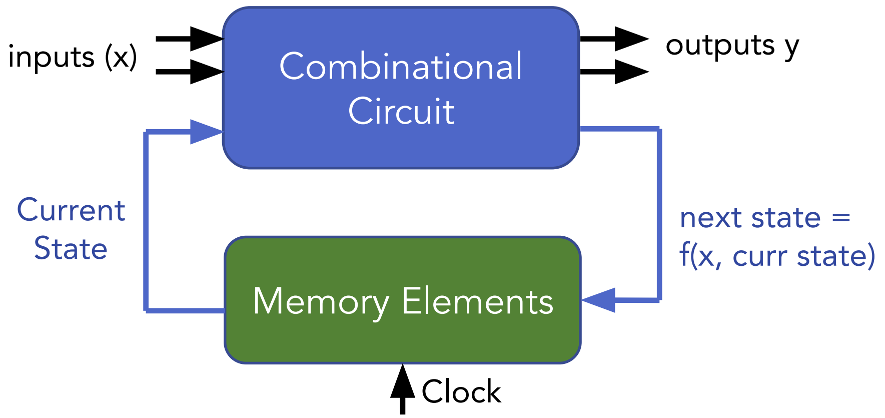
SystemVerilog

- Synthesizable **2-state** (0,1) data type added
- **2-state** variable can be used in testbench code where X,Z are not required
- **2-state** variable in RTL Model improves simulation performance and 50% reduced memory usage as compared to **4-state** variables

Blocking and Non-Blocking Statements

Combinational Vs. Sequential Circuits





- Output solely depends on current **input values**
- Does not require **memory elements** or **clock** signals
- Behavior defined by a set of **output functions**
- E.g.: Mux, Full adder, comparator, decoder etc.

- Output depends on current **input values** and **state** (i.e. determined by the sequence of past inputs)
- Requires **clock signal** and **memory elements** for storing the 'state'
- Behavior defined by a set of **output functions** and **next state function**
- E.g.: Flip-flop, latch, shift register, etc.

Overview

- Today's lecture applies to procedural blocks in SystemVerilog

- Two types of procedural blocks:
  1. Initial: execute only once at time zero
  2. Always: execute over and over again (always)

```
initial begin
   <procedural statements>
end
```

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

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
  - Sequential logic
    - Edge-sensitive sequential logic (such as flipflops)
    - 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

Posedge and Negedge Events

Positive edge (**posedge**) defines a **rising edge** of a signal

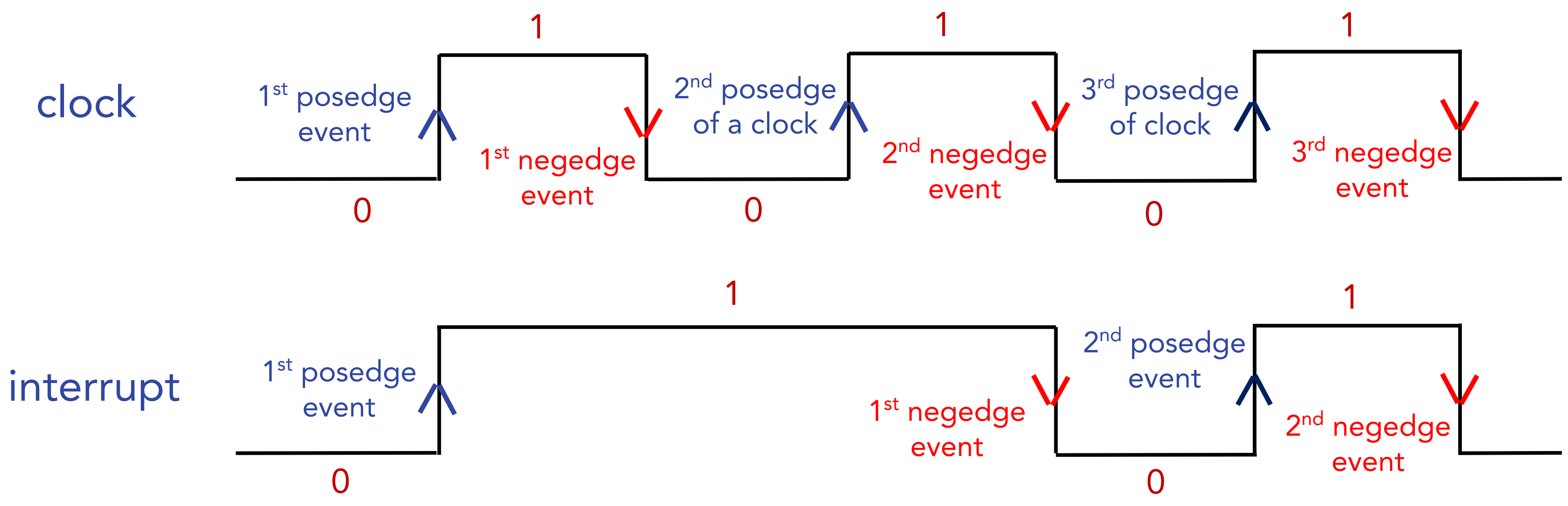
Posedge event triggers when a signal transitions from:

- 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



System Verilog Procedural Assignments: Blocking and Non Blocking

- SystemVerilog has two types of procedural assignments to model combinational and sequential circuits/logic
  - Blocking Assignments represented with equal sign (=) to model combinational logic
    - Syntax : LHS Variable_Name = [delay or event control] RHS_Expression;
    - Example: sum = a + b;
    - sum = #5 a + b;
  - Non-Blocking Assignments represented with less-than-equal (<=) to model sequential logic, such as flip flops, latches, shift registers, etc
    - Syntax : LHS Variable_Name <= [delay or event control] RHS_Expression;
    - Example: sum <= a + b;

Blocking and Non Blocking Statements

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 blocking assignment **statement executes concurrently** (i.e., in parallel) without blocking each other
  - Order does not matter!
- Used to model **sequential logic**

Blocking and Non Blocking Statements

Blocking

```
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
```

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

Non-Blocking

```
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
```

Simulation Result

- At 1st step, a=5, b=10, c=4, sum=2, prod=8
  - Compute next sum value: (5 + 10) = 15
  - Compute next prod value: (2 * 4) = 8
- At 2nd step,
  - sum becomes 15
  - prod becomes 8

Blocking and Non Blocking Statements

```
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**

Quiz: Blocking / Non Blocking Statements?

```
module blocking_assignment (
  input logic clock
  output logic [2:0] a=0,b=0,c=0
);

always@(posedge clock)
 begin
   a =  #5 2;
   b =  #5 4;
   c =  #10 5;
 end
endmodule
```

```
module non_blocking_assignment (
  input logic clock,
  output logic [2:0] a=0,b=0,c=0
);

always@(posedge clock)
 begin
   a <=  #5 2;
   b <=  #5 4;
   c < = #10 5;
 end
endmodule
```

Draw the timing waveforms for when a, b, and c values change?

System Verilog Modeling and Verification Events Scheduling Flow

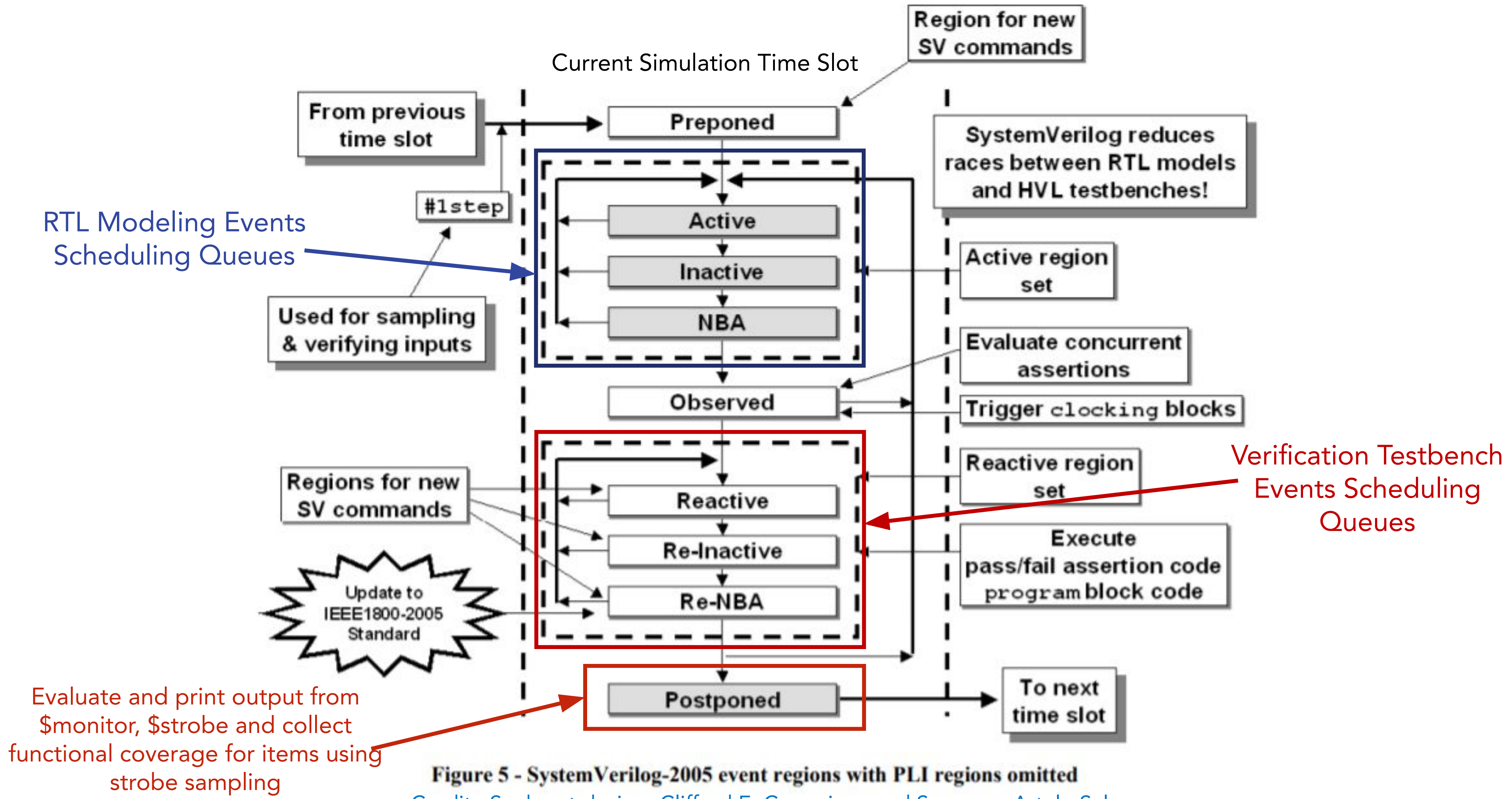
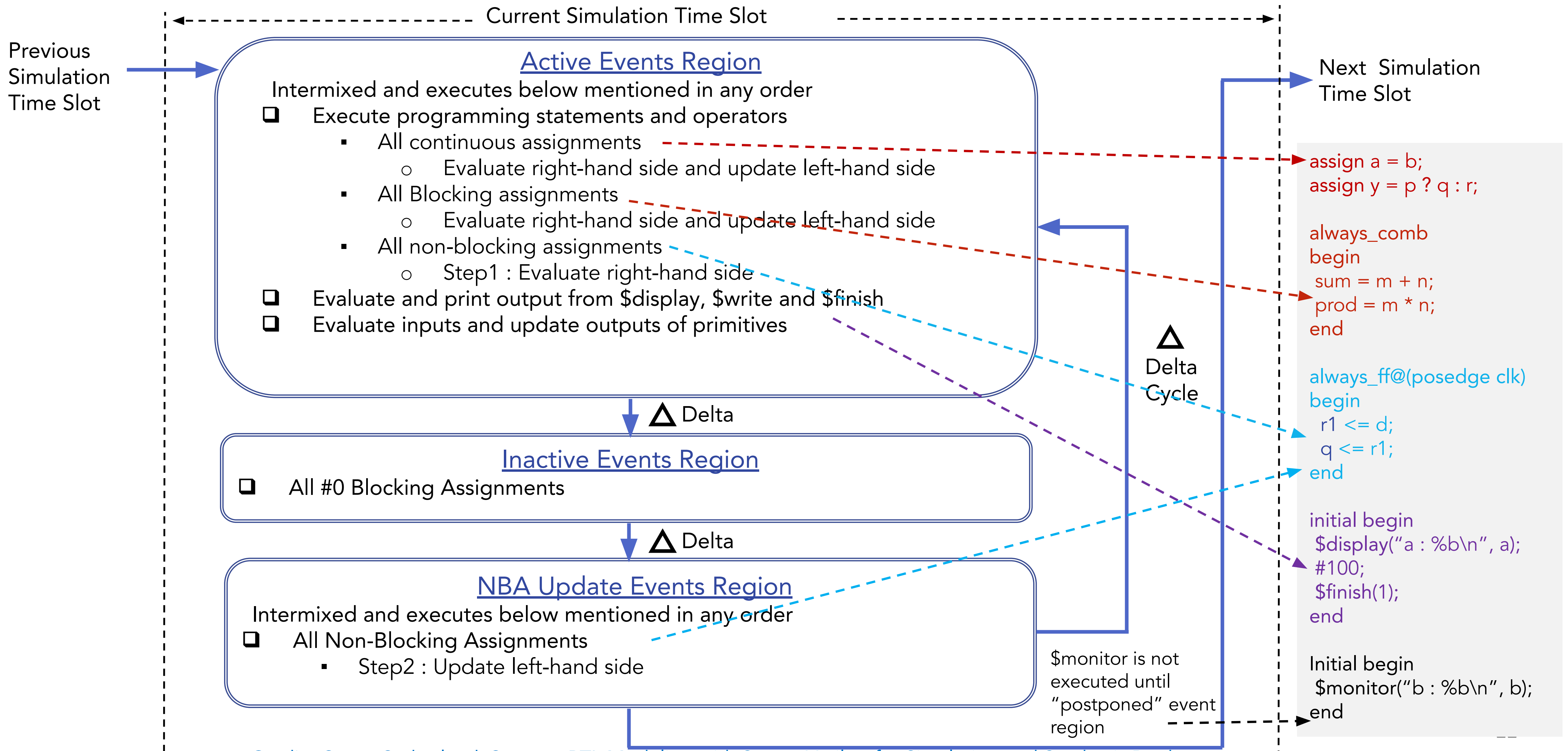

Current Simulation Time Slot
Region for new SV commands
From previous time slot
Preponed
#1step
SystemVerilog reduces races between RTL models and HVL testbenches!
RTL Modeling Events Scheduling Queues
Active
Inactive
NBA
Active region set
Used for sampling & verifying inputs
Evaluate concurrent assertions
Observed
Trigger clocking blocks
Regions for new SV commands
Reactive region set
Verification Testbench Events Scheduling Queues
Reactive
Re-Inactive
Re-NBA
Execute pass/fail assertion code program block code
Update to IEEE1800-2005 Standard
Postponed
To next time slot
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

System Verilog RTL Modeling Schedule Flow



Current Simulation Time Slot
Previous Simulation Time Slot
Next Simulation Time Slot
Delta
Delta
Delta Cycle

Active Events Region

Intermixed and executes below mentioned in any order

- ❑ Execute programming statements and operators
  - ▪ All continuous assignments
    - o Evaluate right-hand side and update left-hand side
  - ▪ All Blocking assignments
    - o Evaluate right-hand side and update left-hand side
  - ▪ All non-blocking assignments
    - o Step1 : Evaluate right-hand side
- ❑ Evaluate and print output from $display, $write and $finish
- ❑ Evaluate inputs and update outputs of primitives

Inactive Events Region

- ❑ All #0 Blocking Assignments

NBA Update Events Region

Intermixed and executes below mentioned in any order

- ❑ All Non-Blocking Assignments
  - ▪ Step2 : Update left-hand side

$monitor is not executed until "postponed" event region

```
assign a = b;
assign y = p ? q : r;

always_comb
begin
 sum = m + n;
 prod = m * n;
end

always_ff@(posedge clk)
begin
  r1 <= d;
  q <= r1;
end

initial begin
 $display("a : %b\n", a);
 #100;
 $finish(1);
end

Initial begin
 $monitor("b : %b\n", b);
end
```

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

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

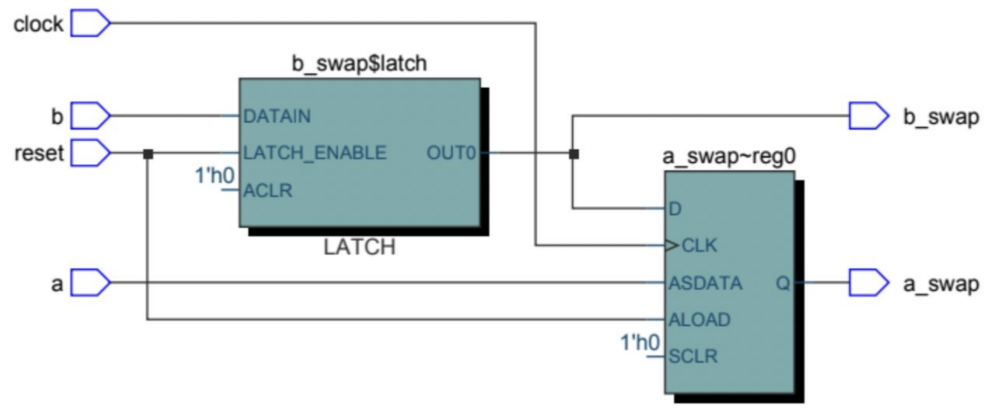
Which Code Will Swap a_swap and b_swap?

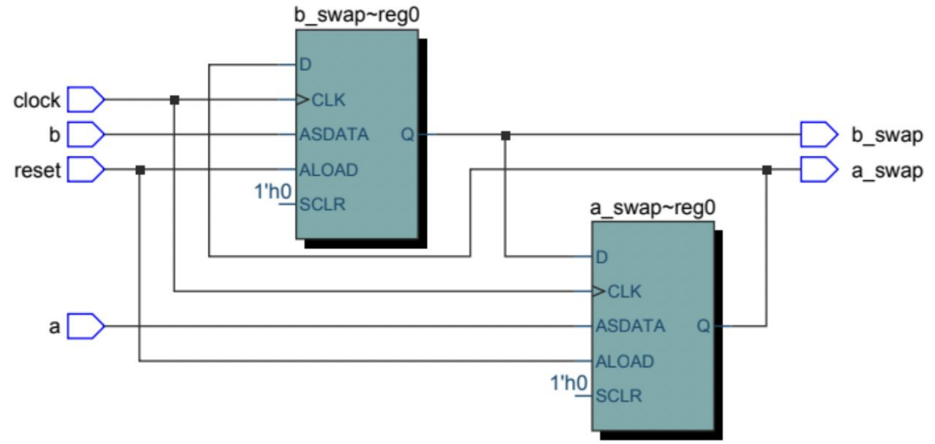
```
module blocking_assignment (
 input logic clock, rst, a, b,
 output logic a_swap, b_swap
);

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!

Quiz: What Would the Synthesis Tool do?

```
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

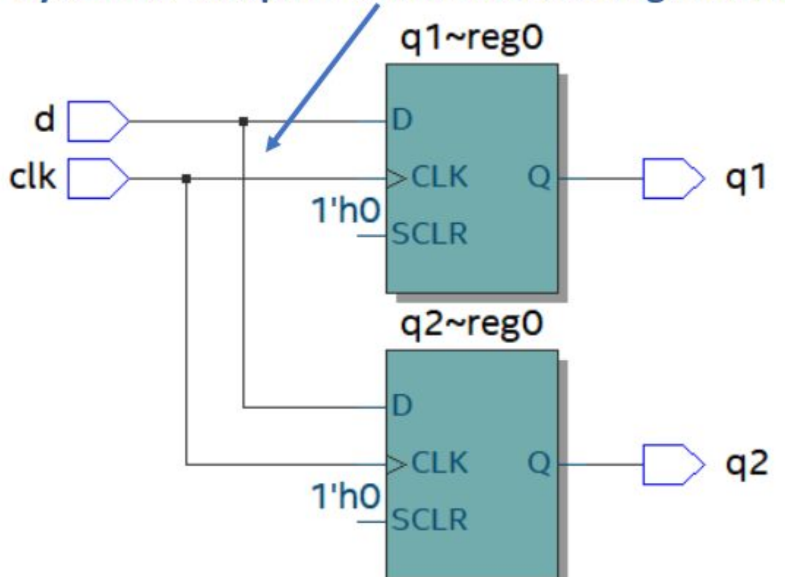
Synthesis compiler will create two registers in parallel

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

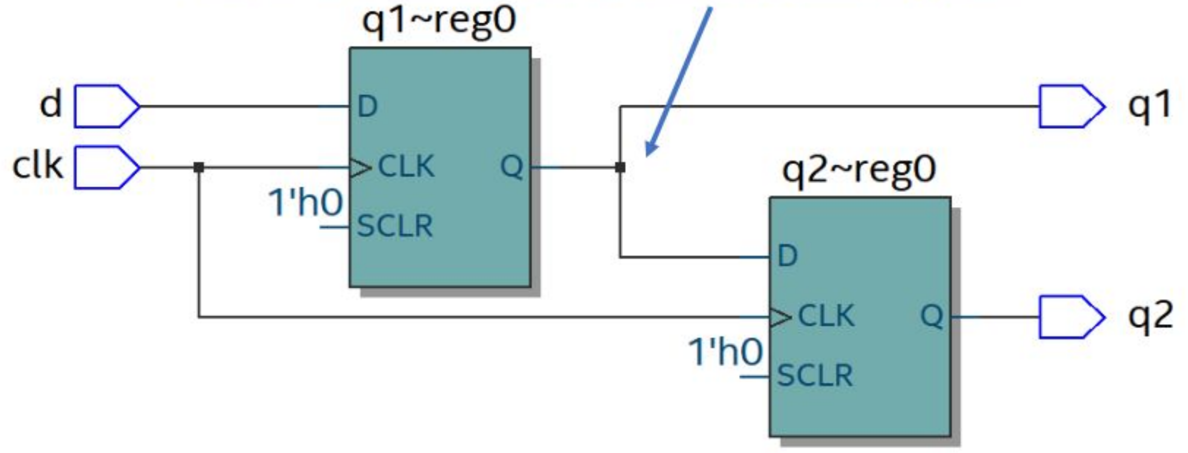
Synthesis compiler will create two registers in parallel



```
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 serially chained registers and circuit will behave as a two bit shift register


q1~reg0
d
clk
D
CLK
Q
1'h0
SCLR
q1
q2~reg0
D
CLK
Q
1'h0
SCLR
q2