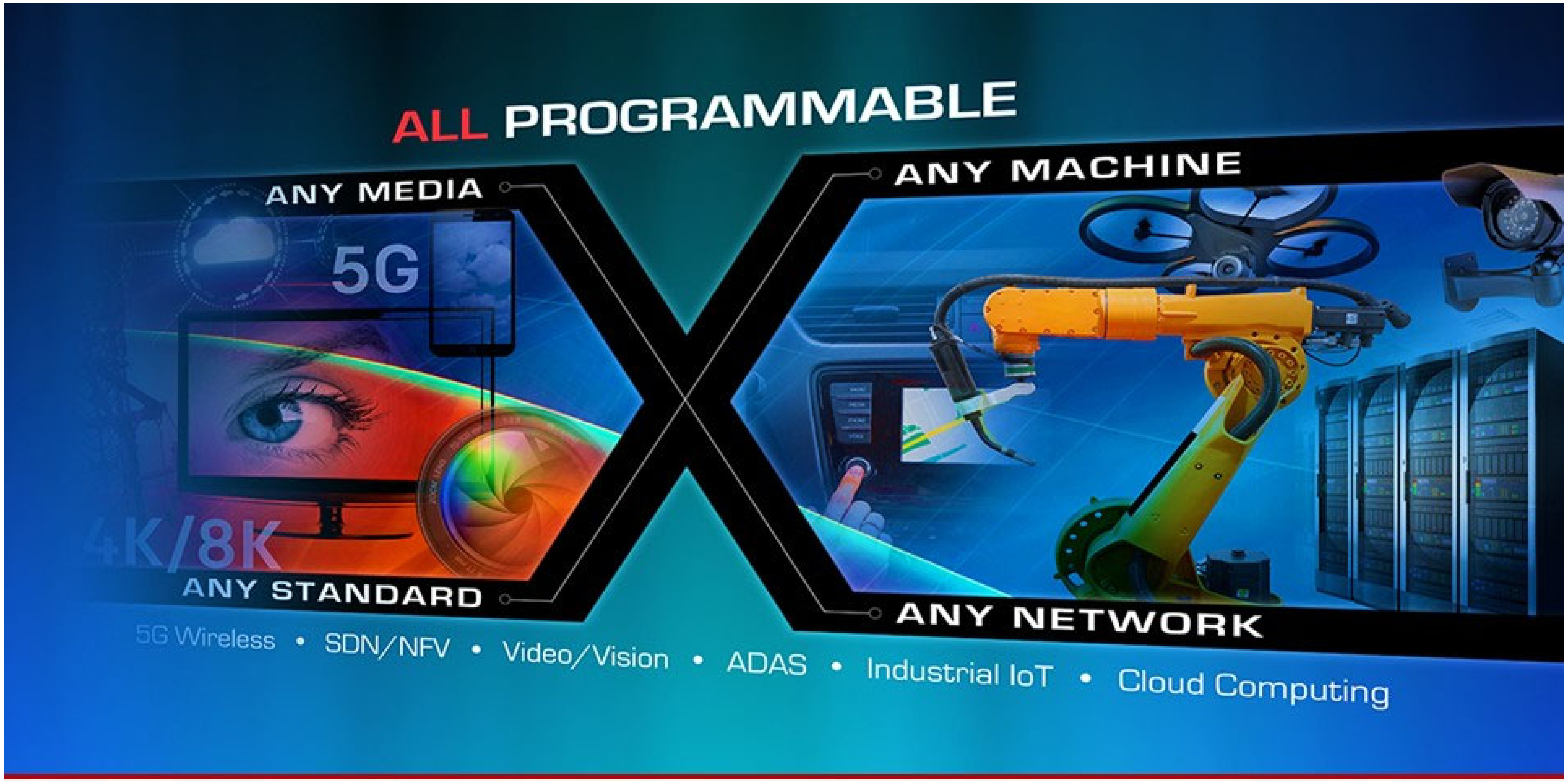
Seminar

FPGAXpert - Verilog

Lecturer

Nir Balulu

).


ALL PROGRAMMABLE
ANY MEDIA
ANY MACHINE
ANY STANDARD
ANY NETWORK
5G
4K/8K
5G Wireless • SDN/NFV • Video/Vision • ADAS • Industrial IoT • Cloud Computing

Introduction to Testbenches

2015.3

Design Verification

- A key advantage to an HDL design entry approach is the ability to verify the design before implementation
- Verilog HDL allows you to first verify the source code and then the resulting netlist
- As designs get more complex, an increasing portion of the total design cycle is devoted to verification

Objectives

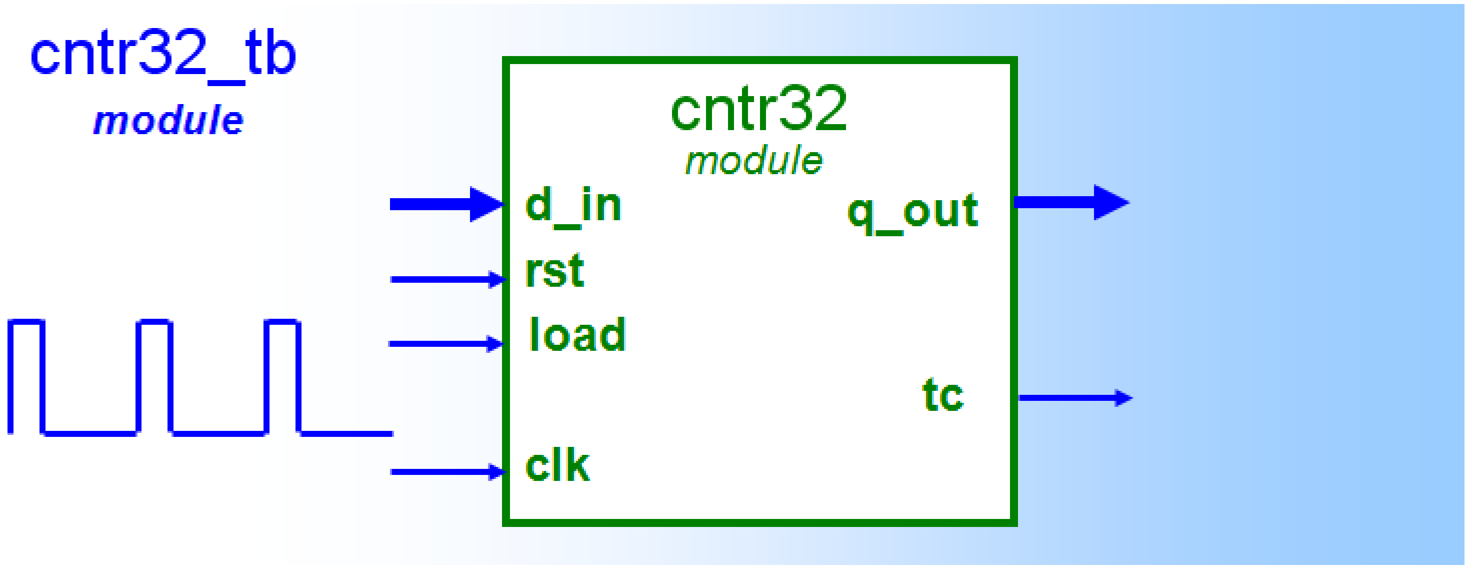
After completing this module, you will be able to:

- Describe the concept and general application of a testbench
- Distinguish between simulation only and synthesizable constructs
- List some basic recommendations for effective design verification
- List the necessary components for creating and executing a Verilog testbench
- Create basic input stimulus and model input clocks
- Use *$display* and *$monitor* appropriately for simulator tasks

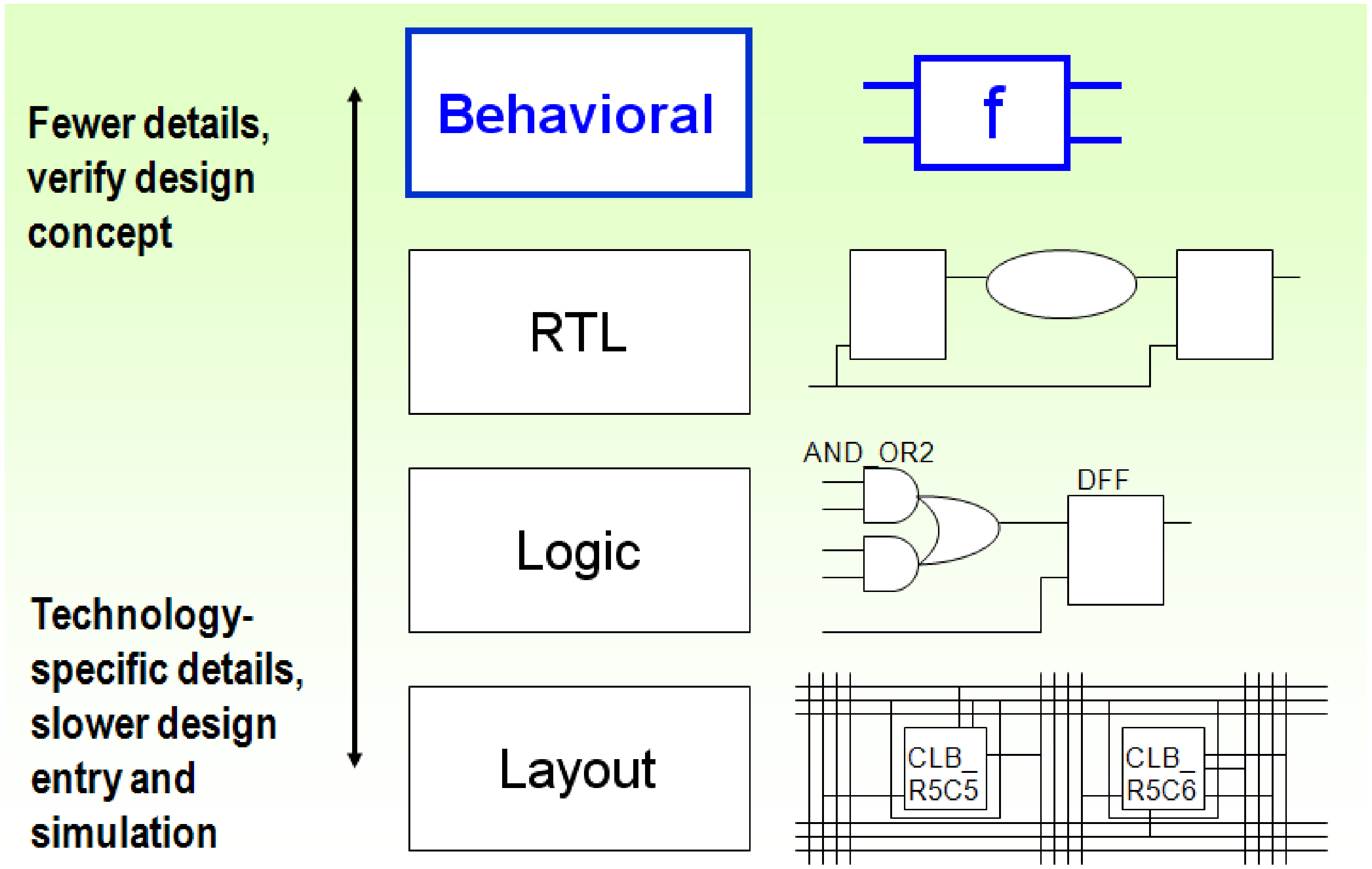
The Testbench Concept

Testbench Concept

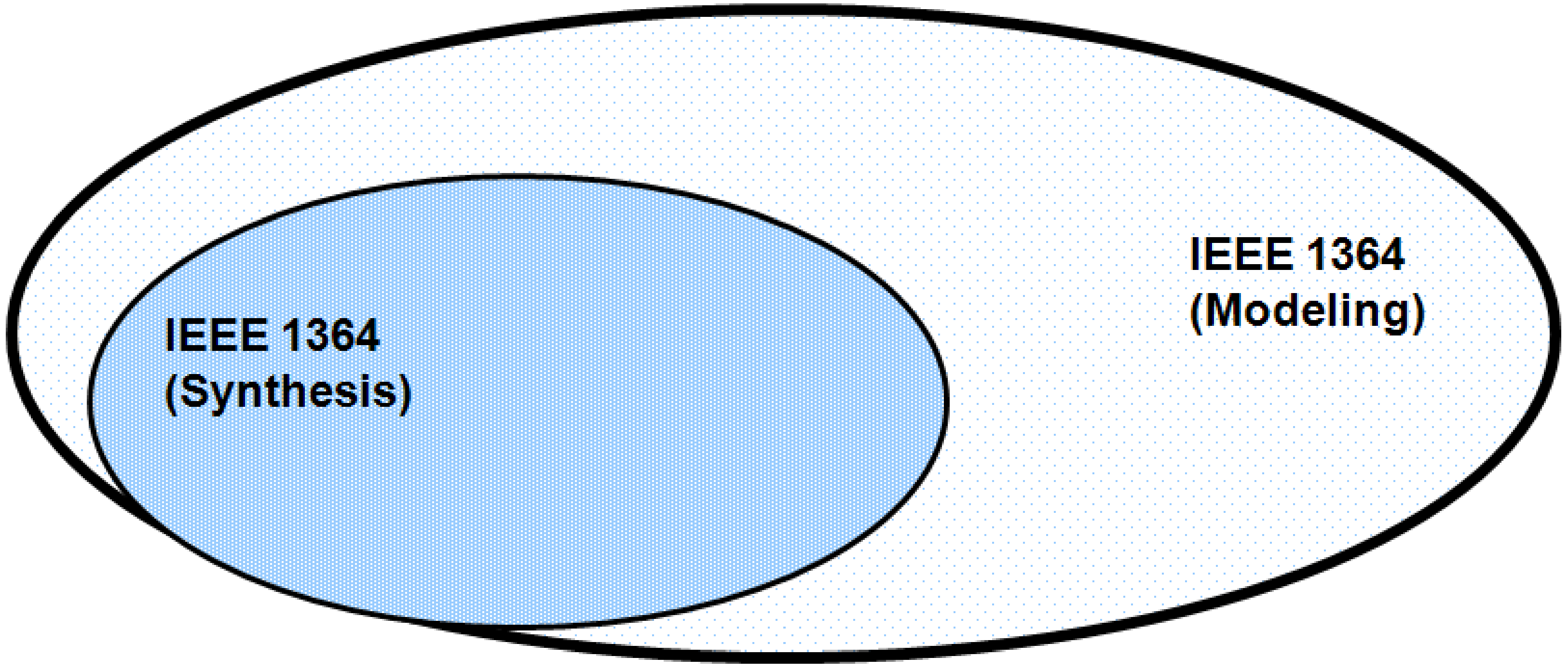
- A Verilog testbench (also called test fixture) is a virtual "test bed"
  - An upper-level hierarchical Verilog structure applies input stimulus to a Unit Under Test (UUT) and monitors the output to verify functionality



Application of a Testbench


Fewer details, verify design concept
Technology-specific details, slower design entry and simulation
Behavioral
f
RTL
Logic
AND_OR2
DFF
Layout
CLB_R5C5
CLB_R5C6

Complete IEEE 1364



Behavioral modeling in Verilog utilizes the broad capabilities of the language

Components of a Testbench

- *`timescale* declaration
  - Specify time unit for all delays
- Module, which defines the testbench top-level structure
  - A testbench usually does not have ports
- Internal signals, which will drive the stimuli into the UUT and monitor the response from the UUT
  - Signal to drive and monitor
- UUT instantiation
- Stimuli generation
  - Write statements to create stimulus and procedural block
- Response monitoring and comparing
  - Self-testing statements that will report values, error, and warnings
  - $display, $write, $strobe, and/or $monitor system tasks

Apply Your Knowledge

1. Which of the following does not appear on or within a testbench?
   a) Internal signals
   b) Unit under test
   c) Ports
   d) Stimulus

2. Can a testbench be synthesized?

3. What are the advantages of performing all stages of verification with the same test fixture?

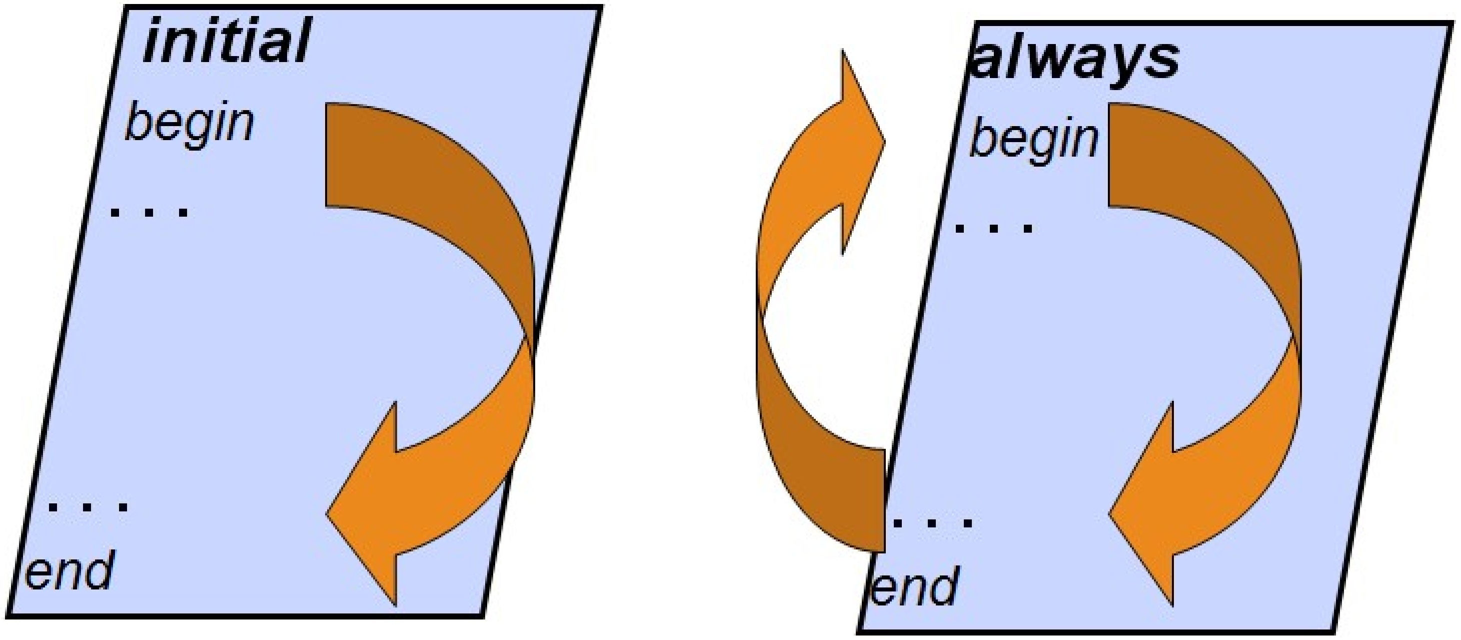
Behavioral and Procedural Coding

Procedural Statements

- In Verilog HDL, behavioral-level modeling is characterized by the use of procedural statements, which describe the method of assignment and/or updating signal values
- There are two procedural statements: *initial* and *always*
- In practical usage, *initial* statements are commonly used within the behavioral (simulation) environment, whereas *always* statements are used in both behavioral and RTL coding

initial versus always

- By definition, all statements within an *initial* block execute only once; the statements within an *always* block continually execute for the duration



Procedural Coding

- A testbench (behavioral) description has flexibility not available in RTL-level code or actual hardware
  - Values stored in signal
    - ***reg   sig1 = 1’b0 ;***
  - Model bus input and output
    - ***reg data_bus = 32’hafc175b4 ;***
  - No need to consider propagation delay
  - Model timing as necessary
    - ***always @ (a, b)***
    - ***#4   y <= a & b ;***

Before the Testbench

- Before you actually start building the testbench, creating a concise written verification strategy or some form of test plan is often helpful
  - This document should be included in the overall design documentation to enhance readability and module reuse
- The verification strategy document should include, but is not limited to
  - Brief summary of the hardware
  - Details on critical functionality
  - Overall verification goals
  - Nature and source of input stimulus
  - Tool-specific or compiler-specific considerations (if applicable)

Apply Your Knowledge

1. Statements within an *initial* blocks execute
   a) Whenever the *initial* block is triggered
   b) Once per simulation cycle
   c) After the simulation clock starts
   d) Once per simulation

2. Why is it important to verify each submodule before the design hierarchy is completed?

Testbench Examples

- The Testbench Concept
- Behavioral and Procedural Coding
- **Testbench Examples**
- Using $display and $monitor
- More on Testbenches
- Summary

`timescale Directives

- Timescales are used to specify delay units and simulation resolution within a given module
  - *`timescale* <reference_time_unit> / <resolution>

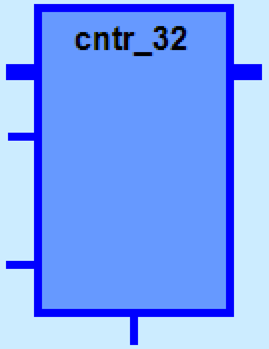
```
`timescale  1 ns / 100 ps
module dff_tb ( );
reg   rst ;
initial
begin
          rst = 1'b0 ;
 # 25 rst = 1'b1 ;
 # 50 rst = 1'b0 ;
end
. . .
endmodule
```

rst toggles at simulation time 25 and 75 ns

Testbench Example

- This is the source code for a 32-bit up counter and will be represented as the UUT in the testbench example

```
`timescale 1 ns / 1 ns
module cntr32  (    input  clk, rst, load, ce,
                    input   [ 31:0 ] d_in,
                    output reg [ 31:0 ] q ) ;

always @   ( posedge clk, posedge rst )
   begin
     if (  rst == 1'b1 )
       q <= 32'b0;
    else if ( load == 1'b1)
       q <= d_in ;
     else if  (ce == 1'b1 )
       q <= q + 1 ;
  end
endmodule
```

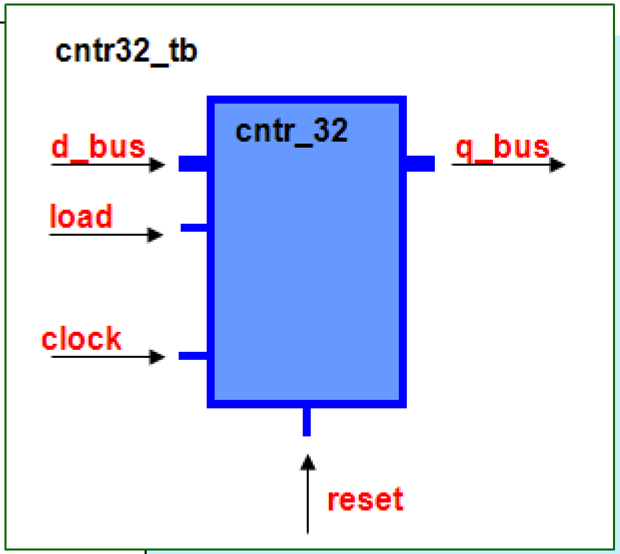


☑ Lab Marker

CNTR32 Testbench

- A simple testbench for the module **cntr32** might be constructed as follows

```
`timescale 1 ns / 1 ns
module cntr32_tb ( ) ;

reg     [ 31:0] d_bus,
wire    [ 31:0] q_bus
reg     clock, reset, load ;

cntr32 uut ( d_bus, clock, reset, load, q_bus );
 initial
     reset = 1'b1;
     #100 reset  = 1'b0 ;
    d_bus = 32'hffff0000 ;
    clock  = 1'b1;

    #25 d_bus = 32'h00000001 ;
    clock  = 1'b0;
    load = 1'b1 ;
    #50 load = 1'b0 ;

      clock  = 1'b1;
 end
```



☑ **Lab Marker**

Creating Clock Signals

- To create a free-running clock for a testbench
  – Declare the clock signal and initialize it to either 1 or 0

```
reg   clk_sig = 1'b0 ;
```

  – Create a concurrent assignment that inverts the clock signal at the appropriate interval for the intended frequency

```
always   #5  clk_sig = ~clk_sig ;
```

- If the clock will be other than a 50/50 duty cycle, use separate invert and delay statements within an *always* block

Other Clock Approaches

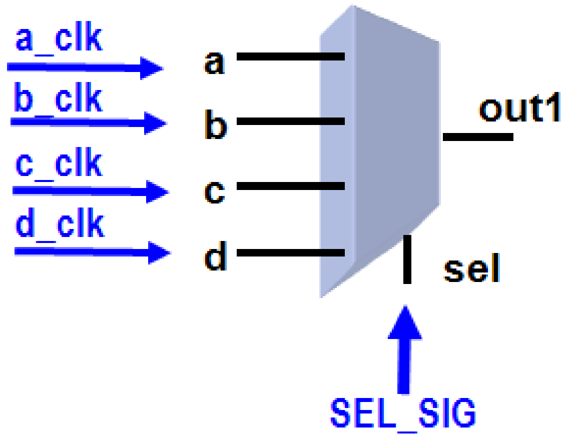
- The example on the left shows a non-50/50 duty cycle clock; the example on the right uses an *initial* block and *forever* loop construct

```
`timescale 1 ns / 1ns
`define period 10.0

module tb ( … ) ;
 reg clk_sig = 1'b0 ;

  always
   begin
   # ( `period * 0.4 ) clk_sig = 1'b1 ;
   # ( `period * 0.6 ) clk_sig = 1'b0' ;
 end
. . .
endmodule
```

```
`timescale 1 ns / 1ns
`define period 10.0

module tb ( … ) ;
 reg clk_sig = 1'b0 ;

 initial
  forever
   #( `period/2 ) clk_sig = ~ clk_sig ;
   . . .
endmodule
```

Modeling Input Stimulus

- Input stimulus can be modeled by using procedural assignment, loops, or increment statements, for example. Complete the code segments in the shaded areas according to the verification strategy outlined below



UUT 4:1 Multiplexer Verification Strategy

(1) Create four separate and distinct input clock frequencies

Use *always* for A and C_CLK and a *forever* loop for B and D_CLK

(2) Increment SEL (200 ns) to cover all possible values

1

```
reg a_clk = 1'b0 ;
always ...
```

```
reg b_clk = 1'b0 ;
initial ...
```

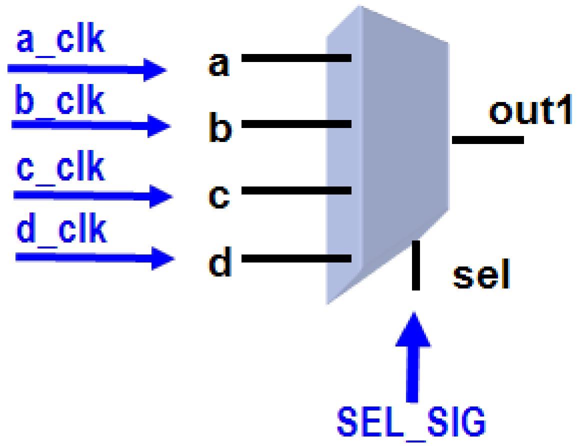
2

```
reg [1:0] sel_sig = 2'b00 ;
always ...
```

```
reg [1:0] sel_sig = 2'b00 ;
initial begin ...



 end
```

One Possible Solution

- Input stimulus can be modeled by using procedural assignment, loops, or increment statements, for example. Consider the following approaches



UUT 4:1 Multiplexer Verification Strategy
(1) Create four separate and distinct input clock frequencies
  Use *always* for A and C_CLK and a *forever* loop for B and D_CLK
(2) Increment SEL (200 ns) to cover all possible values

1

```
reg a_clk = 1'b0 ;
always #5 a_clk = ~a_clk ;
```

```
reg b_clk = 1'b0 ;
initial forever #12 b_clk = ~b_clk ;
```

2

```
reg [1:0] sel_sig = 2'b00 ;
always #200 sel_sig = sel_sig + 1 ;
```

```
reg [1:0] sel_sig = 2'b00 ;
initial begin
        #200 sel_sig = 2'b01 ;
        #200 sel_sig = 2'b10 ;
        #200 sel_sig = 2'b11 ;
 end
```

Apply Your Knowledge

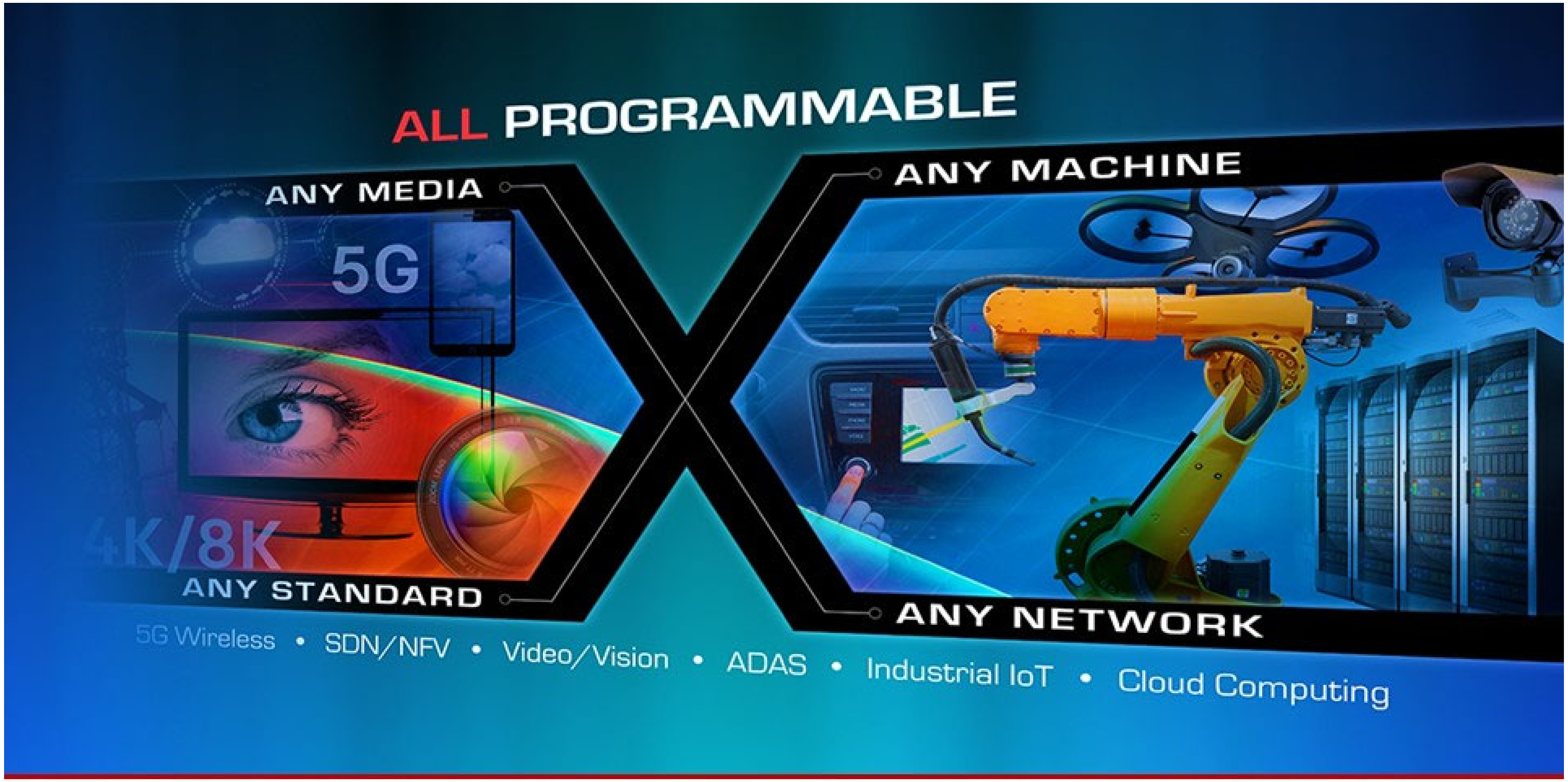
1. Write a `timescale declaration that specifies a 1-ps time unit with a 1 ps resolution

2. What is wrong the following **clk** declaration? How can this be fixed?

   reg clk;

   always

   #10 clk = ~clk;


ALL PROGRAMMABLE
ANY MEDIA
ANY MACHINE
5G
4K/8K
ANY STANDARD
ANY NETWORK
5G Wireless • SDN/NFV • Video/Vision • ADAS • Industrial IoT • Cloud Computing

XILINX ALL PROGRAMMABLE.

Verilog Operators and Expressions

2015.3

Choosing Syntax

- Verilog offers a wide choice of operators for logical operations
- From the standpoint of syntax, this leads to a variety of possibilities
- To enhance readability and module reuse, a designer should strike a balance between brevity and clarity

Objectives

After completing this module, you will be able to:

- List the types and classes of Verilog operators
- Use common Verilog operators to model a variety of functions
- Apply the rules for declaring and using signed objects within Verilog
- Apply arithmetic operators to perform standard arithmetic functions on signals and buses

Types and Classes of Verilog Operators

Operators

- Verilog gives designers a wide array of operators
- Operators add flexibility, but they are distinct with regards to operand requirements
  - Bitwise
  - Logical
  - Relational
  - Equality
  - Reduction
  - Conditional
  - Concatenation/replication
  - Shift
  - Arithmetic

Verilog ’01 Enhancements

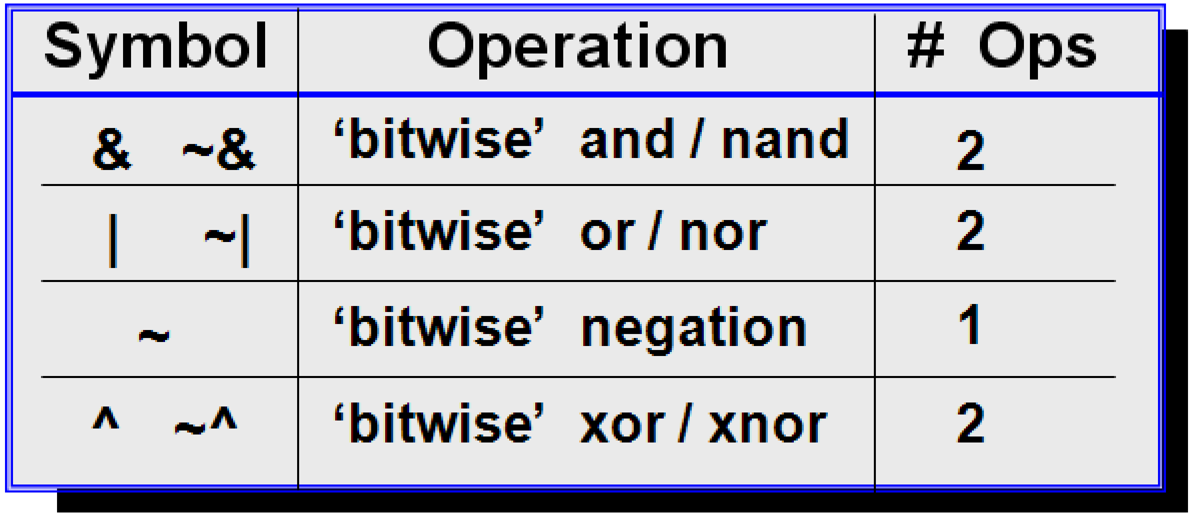
- The Verilog ’01 standard offers additional flexibility and capability
  - These will be discussed throughout this module
- In Verilog ’95, the only signed data type was integer
  - This severely restricted the ability to treat standard data-bus operations as signed
- This module discusses the changes to the language and their practical application

Using Operators

- Types and Classes of Verilog Operators
- **Using Operators**
- Signed and Unsigned Objects
- Summary

Bitwise Operators

- Bitwise operators perform standard logical operations on signals and corresponding elements within buses—bit by bit



Symbol	Operation	# Ops
& ~&	‘bitwise’ and / nand	2
| ~|	‘bitwise’ or / nor	2
~	‘bitwise’ negation	1
^ ~^	‘bitwise’ xor / xnor	2

Bitwise Operator Examples

```
// given…
```

a1= 4’b1011;	b1 = 4’b0001;	c1= 4’b111x ;	d1 = 4’b10z0;

```
assign out1 =  a1 & b1 ;    // produces    4’b0001

               b1 & d1     // produces    4’b0000

               a1  | b1     // produces    4’b1011

                  ~b1     // produces    4’b1110

               a1 ^ c1     // produces    4’b010x

               b1 | d1     // produces    4’b10x1
```

FYI: Target should have appropriate bit width to avoid truncation or zero extension

Logical Operators

- Logical operators in Verilog are abstract in nature
  - Logical operators act as connectives for Boolean-like expressions to produce a true / false result
    - Therefore, the result is always one bit

Symbol	Operation	# of Ops
&&	'Logical' and	2
||	'Logical' or	2
!	negation	1

Logical Operator Examples

// given…

a1= 4’b1011;	b1 = 4’b0000;	c1= 4’b000x ;	d1 = 4’b10z0;

```
assign out1 =  a1 && b1 ;        // produces   1’b0   (false)

               a1  ||  b1 ;      // produces   1’b1   (true)

               a1 && c1 ;        // produces   1’bx   (unknown)

               ! b1 ;            // produces   1’b1

( a1 == 4’b1011 ) && ( b1 == 4’b1111 ) ; // produces  1’b0
```

FYI: Target should be one-bit wide to avoid zero extension

Logical versus Bitwise Negation

- Use of the logical operator to test 1-bit values is a common practice
  - In this scenario, it is proper given that the conditional expression is inherently dependent on a Boolean type (1 bit) result
  - Although the end result is the same, this can potentially create confusion for new Verilog designers and can lead to the assumption that the logical operator can be applied universally
  - This could create potential problems that are difficult to debug because Verilog can automatically zero extend the assignment to match the target

```
module logical_op();
reg a, b;
initial
   begin
     a = 1'b1;
     b = 1' b1;

     if ( a && ~b)
       begin
         ....
         ....
       end
```

(a && b) results in 0 or false because *~b* = 0

```
module logical_op();
integer a, b;
initial
   begin
     a = 1'b1;
     b = 1' b1;

     if ( a && ~b)
       begin
         ....
         ....
       end
```

(a && b) results in 1 or true because *~b* = 1111 1111 1111 1111 1111 1111 1111 1110

Relational Operators

- Relational operators are used to compare operands
  - Comparisons usually control multi-branching or conditional operations

Relational Operator Examples

// given...

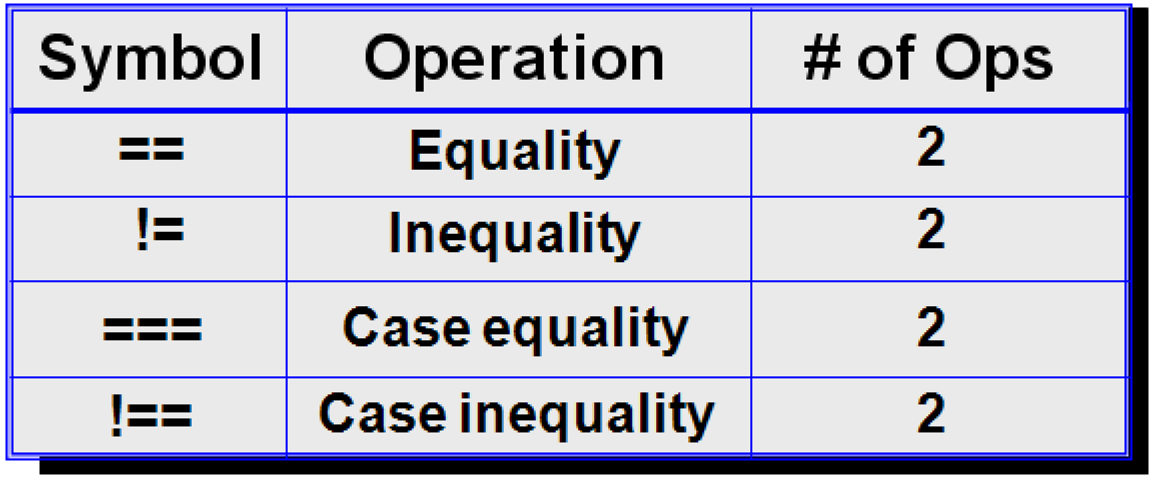
a1= 4'b1011;	b1 = 4'b0001;	c1= 4'b111x ;	d1 = 4'b10z0;

```
assign out1 = a1 < b1 ;           // produces  1'b0   (false)

              a1 > b1 ;           // produces  1'b1   (true)

              a1 >= c1 ;          // produces  1'bx   (unknown)

              b1 < d1 ;           // produces  1'bx   (unknown)
```

FYI: Any unknown 'x' or 'z' causes the result to be unknown

Equality Operators

- Equality operators perform strict comparison of operands
- Two forms of the equality operator
  - Logical equality returns logical/Boolean (1) true / (0) false
    - Or unknown if an ‘x’ or ‘z’ appears in any operand
  - Case equality considers ‘x’ and ‘z’ and returns a logical/Boolean (1) true / (0) false



Symbol	Operation	# of Ops
==	Equality	2
!=	Inequality	2
===	Case equality	2
!==	Case inequality	2

Equality Operator Examples

```
// given…
a1= 4'b1011; b1 = 4'b0001; c1= 4'b111x ; d1 = 4'b10z0; e1 = 4'b111x

assign out1 = (a1 == b1) ;  // produces  1'b0   (false)
            a1 != b1 ;      // produces  1'b1   (true)
            a1 == c1 ;      // produces  1'bx   (unknown)
            c1 === e1;      // produces  1'b1
            c1 == e1 ;      // produces  1'bx
            c1 === d1 ;     // produces  1'b0
```

FYI: Using logical equality, any unknown 'x' or 'z' causes the result to be unknown

Target should be one-bit wide to avoid zero extension

Reduction Operators

- Reduction operators perform a recursive bitwise operation starting from the right of a given vector
  - This is often used as a coding shortcut for logical operations on vectors

Symbol	Operation	# of Ops
& ~&	Reduction and / nand	1
| ~|	Reduction or / nor	1
^ ~^	Reduction xor / xnor	1

Reduction Operator Examples

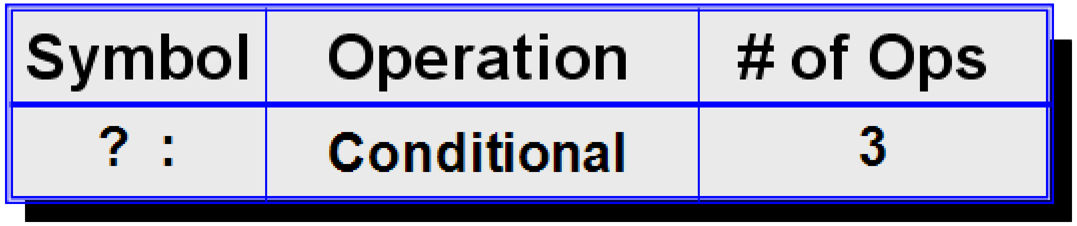
```
// given…
a1= 4'b1011; | b1 = 4'b0011; | c1= 4'b111x ; | d1 = 4'b10z0;

assign out1 =   &a1 ;        // produces    1'b0

                 |a1         // produces    1'b1

                 ^b1         // produces    1'b0

                 ^a1         // produces    1'b1

                 ^c1         // produces    1'bx

                ~^d1         // produces    1'bx
```

Target should be one-bit wide to avoid zero extension

FYI: For reduction xor/xnor, any 'x' or 'z' causes the result to be unknown

Conditional Operators

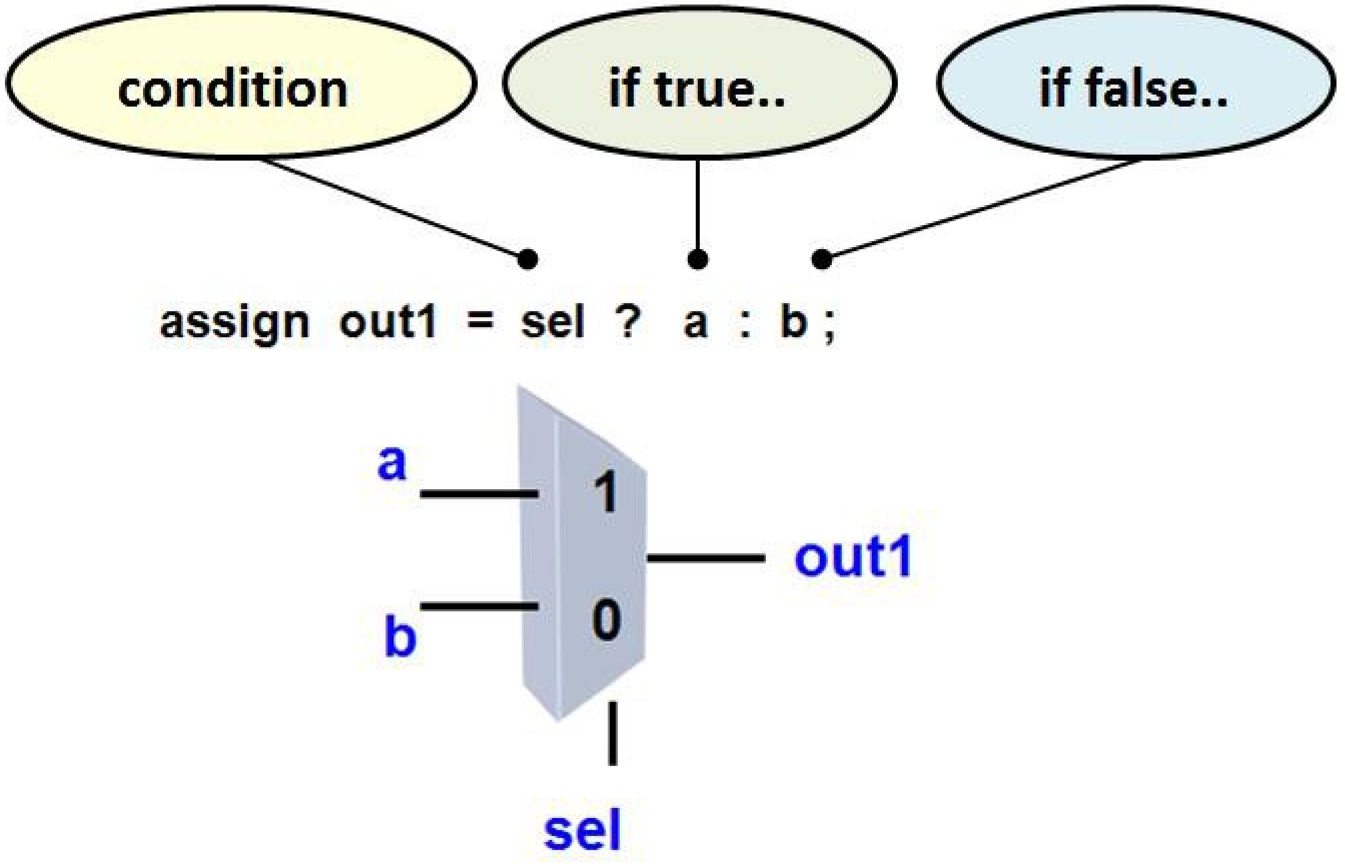
- Conditional operators offer a short-hand version of case or *if/else* statements
  - The next statement to be executed depends on the value of the condition



Symbol	Operation	# of Ops
? :	Conditional	3

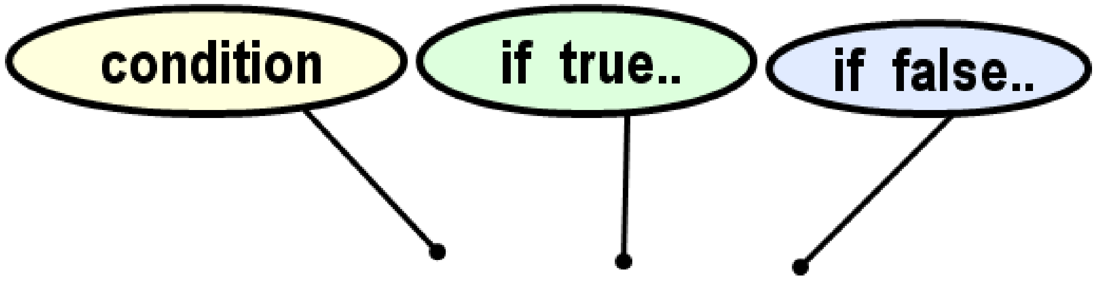
Conditional Operators

- Example one

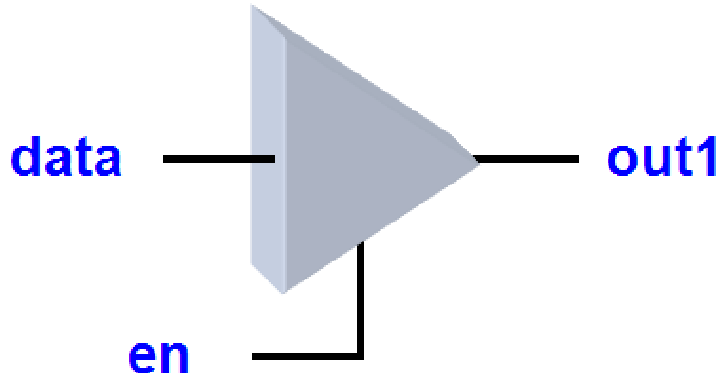


Conditional Operators

- Example two



```
assign  out1  =  en  ?   data  : 1'bz ;
```



Concatenation Operators

- Concatenation operators allow flexible grouping of signals and buses
  - The replication operator provides a short-hand method for repeated concatenations

Symbol	Operation	# of Ops
{ , }	Concatenation	2 (or more)
{ { } }	Replication	2

Concatenation Operator Examples

// given…

a1= 4'b1011;	b1 = 4'b0011;	c1= 4'b111x;	d1 = 4'b10z0;

```
assign out_bus = {a1, b1} ;     // produces   8'b1011_0011

                {a1,c1}         // produces   8'b1011_111x

                { 3{a1}}        // produces  12'b1011_1011_1011

            {{2{d1}},c1 }       // produces  12'b10z0_10z0_111x

        {{ 2{b1}}, {2{c1}}}     // produces  16'b0011_0011_111x_111x
```

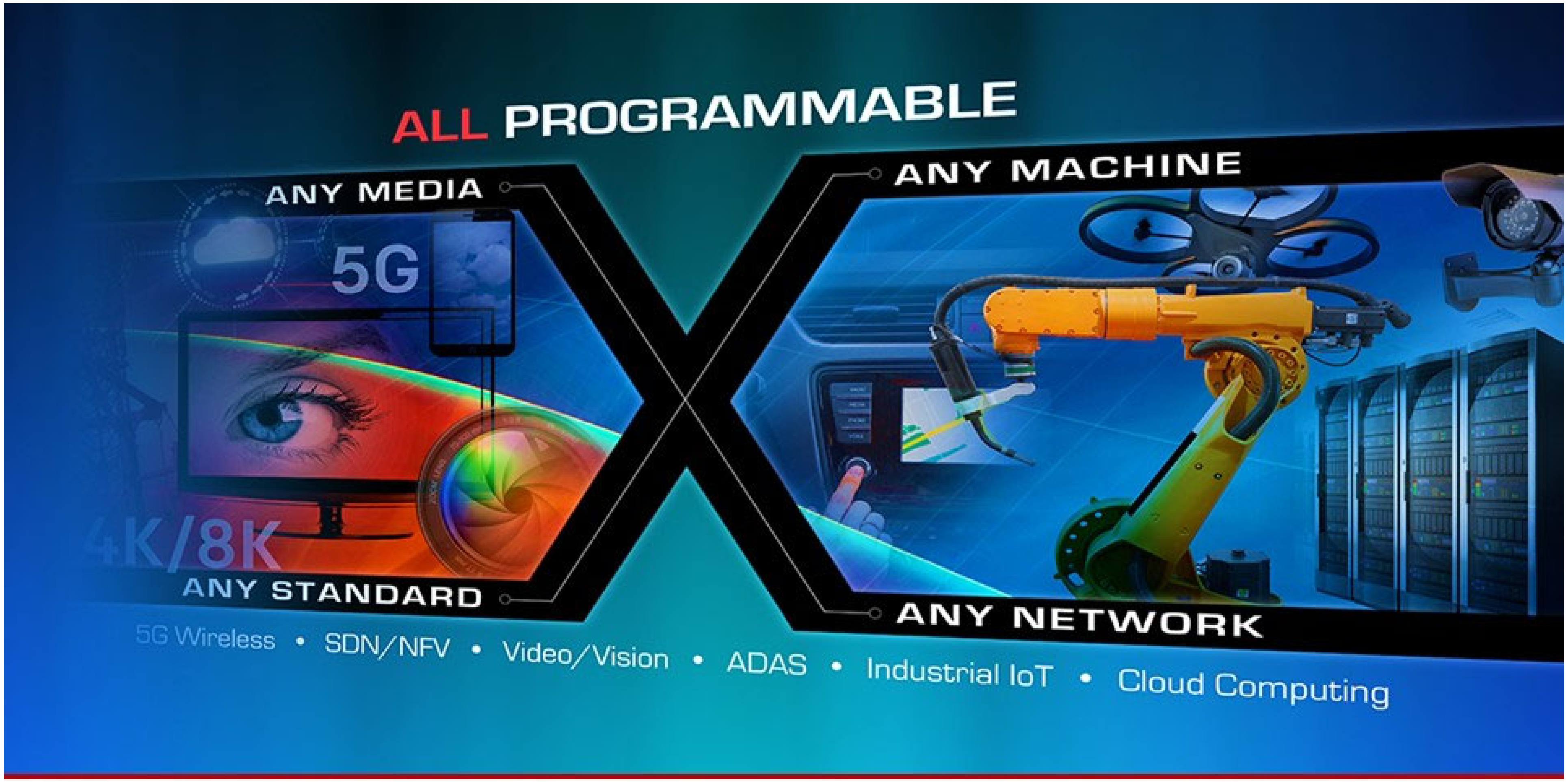
Replication constant

FYI: Target should have appropriate bit width to avoid truncation or zero extension

Shift Operators

- Shift operators allow easy manipulation of buses
- Each bus element is shifted either right or left
  - For traditional shift operators, the vacated position is zero filled
  - For an arithmetic shift right operator (Verilog ’01), the MSB is copied
  - Both arithmetic shift left and standard shift left zero fill the vacated bits

>>	Shift right	2
<<	Shift left	2
>>>	Arith. shift right	2
<<<	Arith. shift left	2


ALL PROGRAMMABLE
ANY MEDIA
ANY MACHINE
5G
4K/8K
ANY STANDARD
ANY NETWORK
5G Wireless • SDN/NFV • Video/Vision • ADAS • Industrial IoT • Cloud Computing

XILINX ALL PROGRAMMABLE™

Continuous Assign Statements

2015.3

Objectives

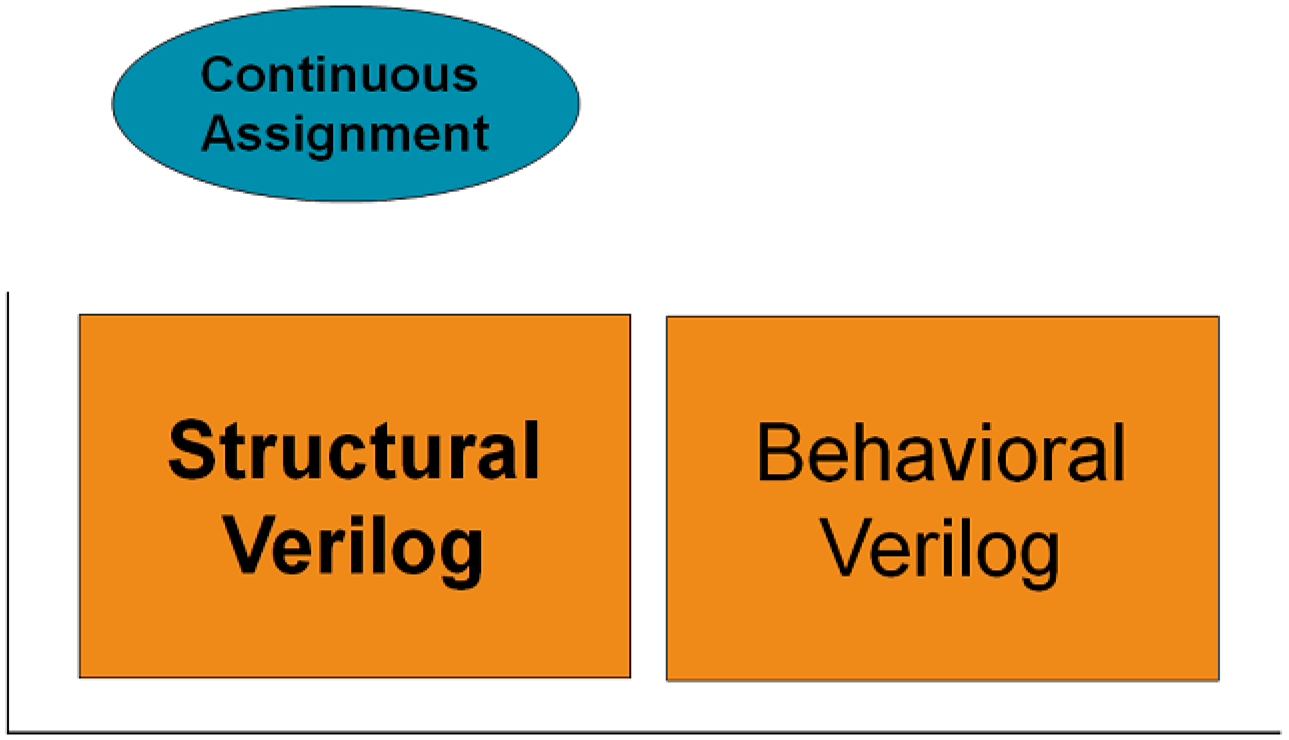
After completing this module, you will be able to:

- Model logic by using a Verilog continuous *assign* statement
- Specify delay attributes

Continuous Assignments

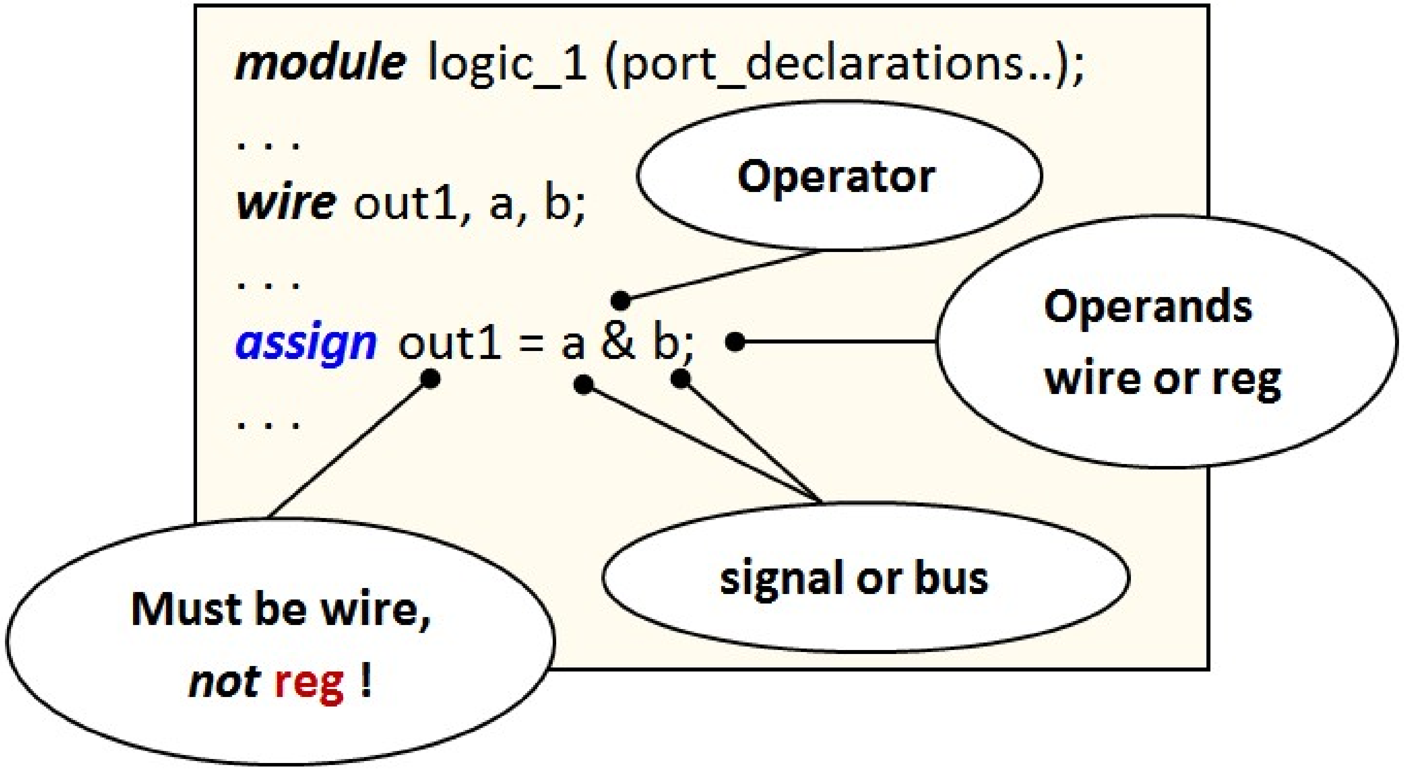
- **Continuous Assignments**
- Delay Specifications
- Summary

Structural vs. Behavioral/Procedural Coding



Continuous Assignments

- Continuous assignments are the basic construct for Structural Verilog
  - The expression can use any of the Verilog operators described earlier

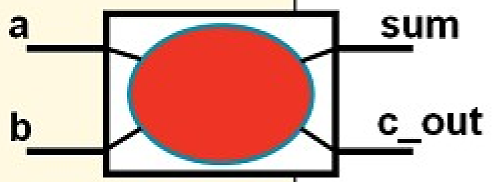


```
module logic_1 (port_declarations..);
. . .
wire out1, a, b;
. . .
assign out1 = a & b;
. . .
```

Driving Output Ports

- By default, any output port is treated as a *wire* data type; therefore, continuous assignments can be specified for that port

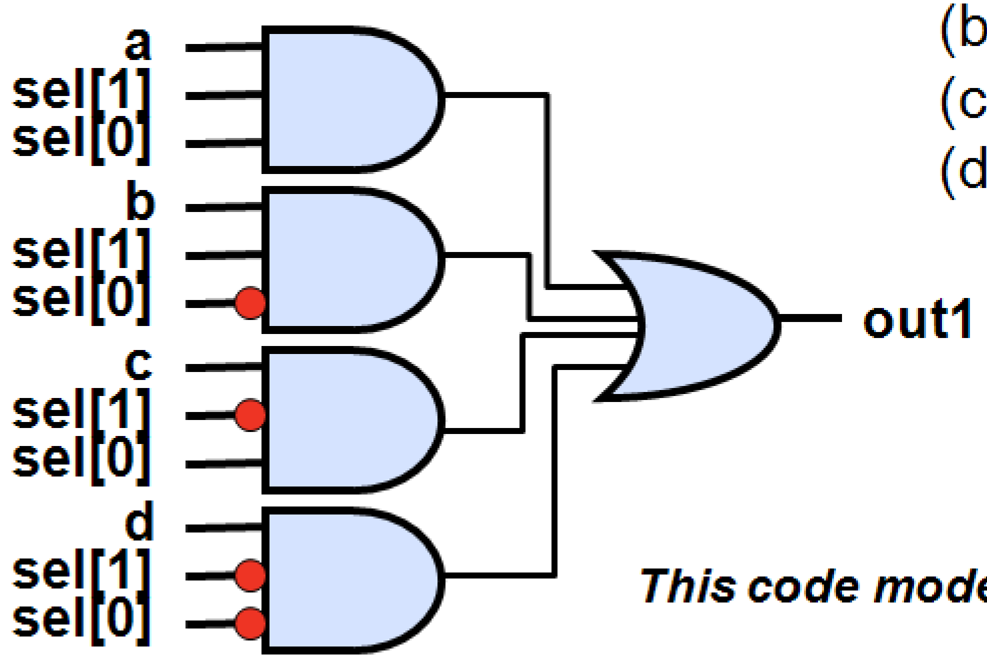
```
module  adder  ( input    [3:0] a,b,
                 output  [3:0] sum,
                 output          c_out ) ;


assign { c_out, sum }  =  a + b  ;  // data flow construct

endmodule
```



Continuous Assignments

- The continuous assignment is re-evaluated whenever any of the operands (inputs) change value

```
assign out1 =  ((a &  sel[0] &  sel[1]) |
                (b & ~sel[0] &  sel[1]) |
                (c &  sel[0] & ~sel[1]) |
                (d & ~sel[0] & ~sel[1])) ;
```



This code models a 4:1 multiplexer

Implicit Assignment Statements

- An implicit, continuous assignment offers a more concise coding method

```
module  logic_1 (port listing…) ;
. . .
wire  out1;
. . .
assign  out1 = a & b ;
```

Alternatively

```
module  logic_1 (port listing…) ;
. . .
wire  out1 = a & b ;
. . .
```

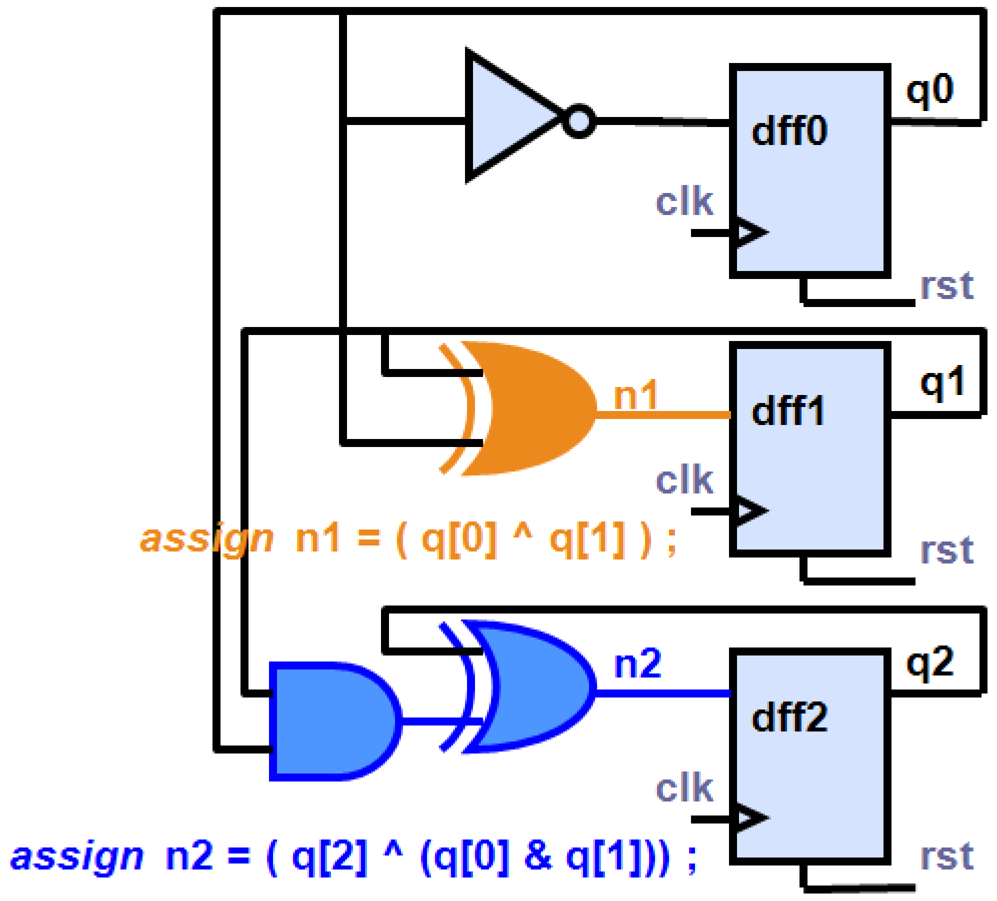
Data Flow Code and Structural Example

- This code models a 3-bit binary counter

```
module  count3  ( input  clk, rst, output  [2:0] q) ;
wire  n1, n2 ;

   assign  n1 = ( q[0] ^ q[1] ) ;
   assign  n2 = ( q[2] ^ (q[0] & q[1])) ;

   dff  dff0 (~q[0], clk, rst, q[0] ) ;
   dff  dff1 ( n1, clk, rst, q[1] ) ;
   dff  dff2 ( n2, clk, rst, q[2] ) ;

endmodule
```

```
module  dff (  input d, clk, rst,
               output reg q ) ;
. . .
```

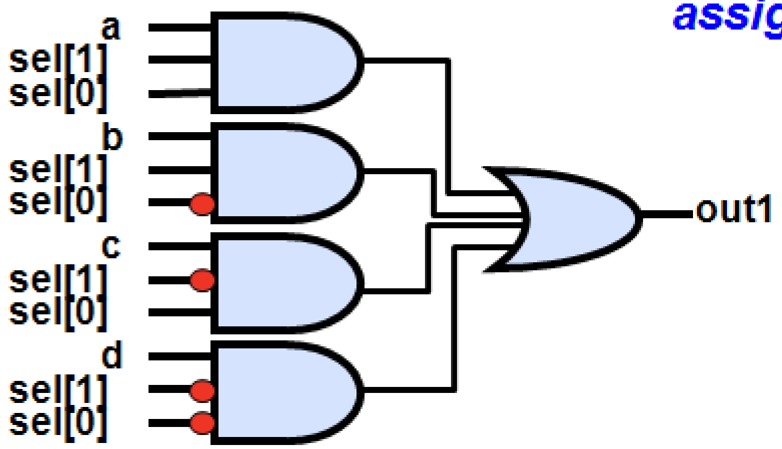
"COUNT3" Implementation



```
assign n1 = ( q[0] ^ q[1] ) ;
assign n2 = ( q[2] ^ (q[0] & q[1])) ;
```

Apply Your Knowledge

1. Using the space below, rewrite this continuous assignment using a combination of logical and equality operators to somewhat improve readability and functional intent

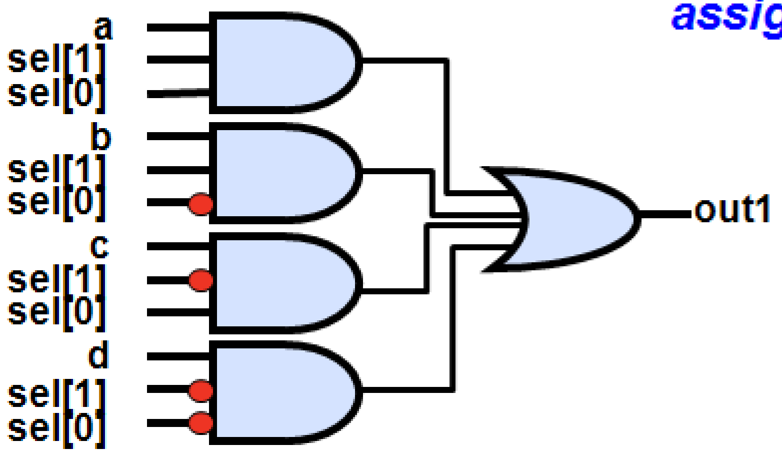


```
assign out1 =  ((a &  sel[0] &  sel[1]) |
                (b & ~sel[0] &  sel[1]) |
                (c &  sel[0] & ~sel[1]) |
                (d & ~sel[0] & ~sel[1])) ;
```

This code models a 4:1 multiplexer

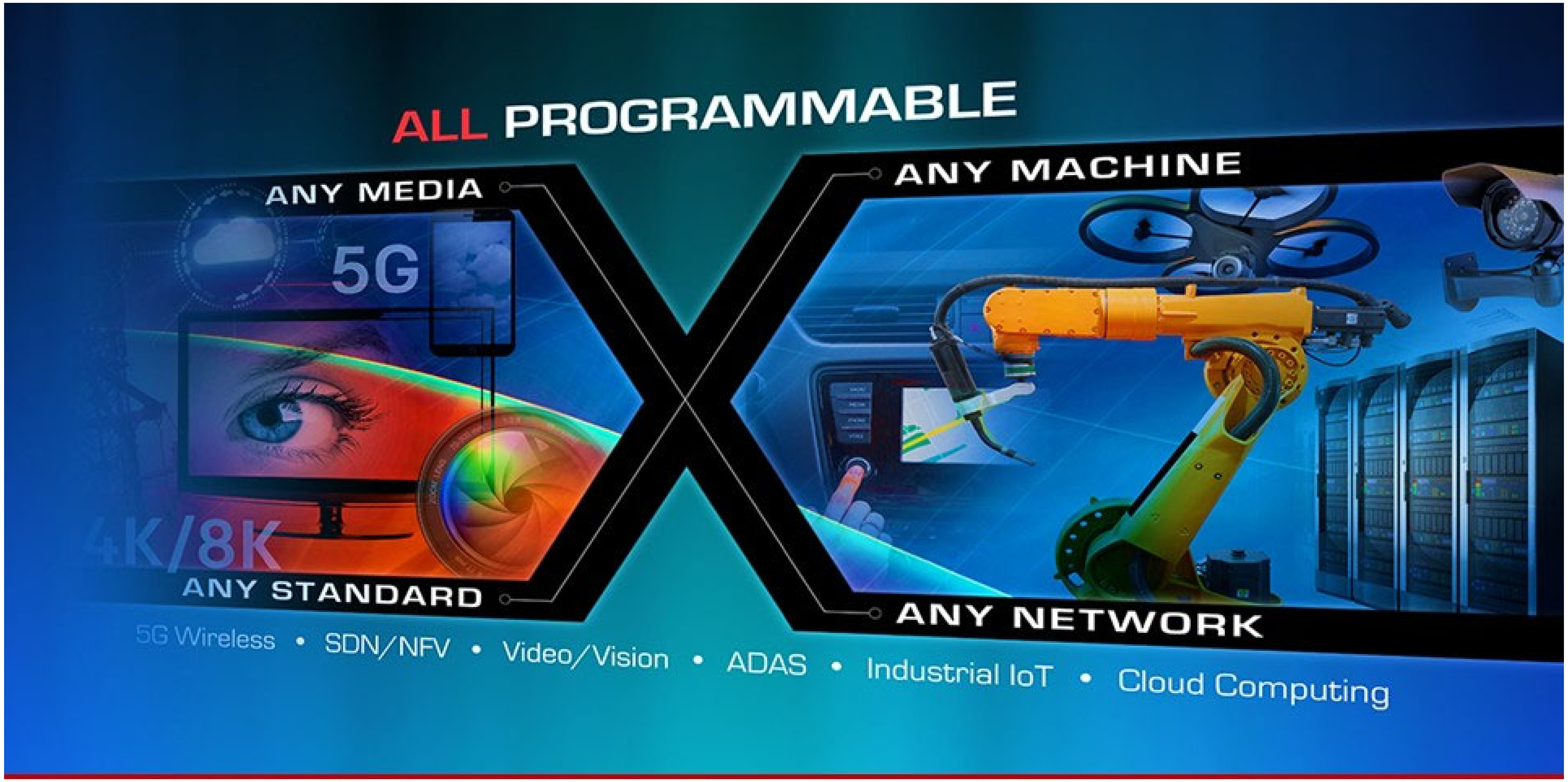
Answer

1. Using the space below, rewrite this continuous assignment using a combination of logical and equality operators to somewhat improve readability and functional intent



```
assign  out1 =   ((a &&  (sel == 2'b11) |
                  (b &&  (sel == 2'b10) |
                  (c &&  (sel == 2'b01) |
                  (d &&  (sel == 2'b00) ) ;
```

This code models a 4:1 multiplexer

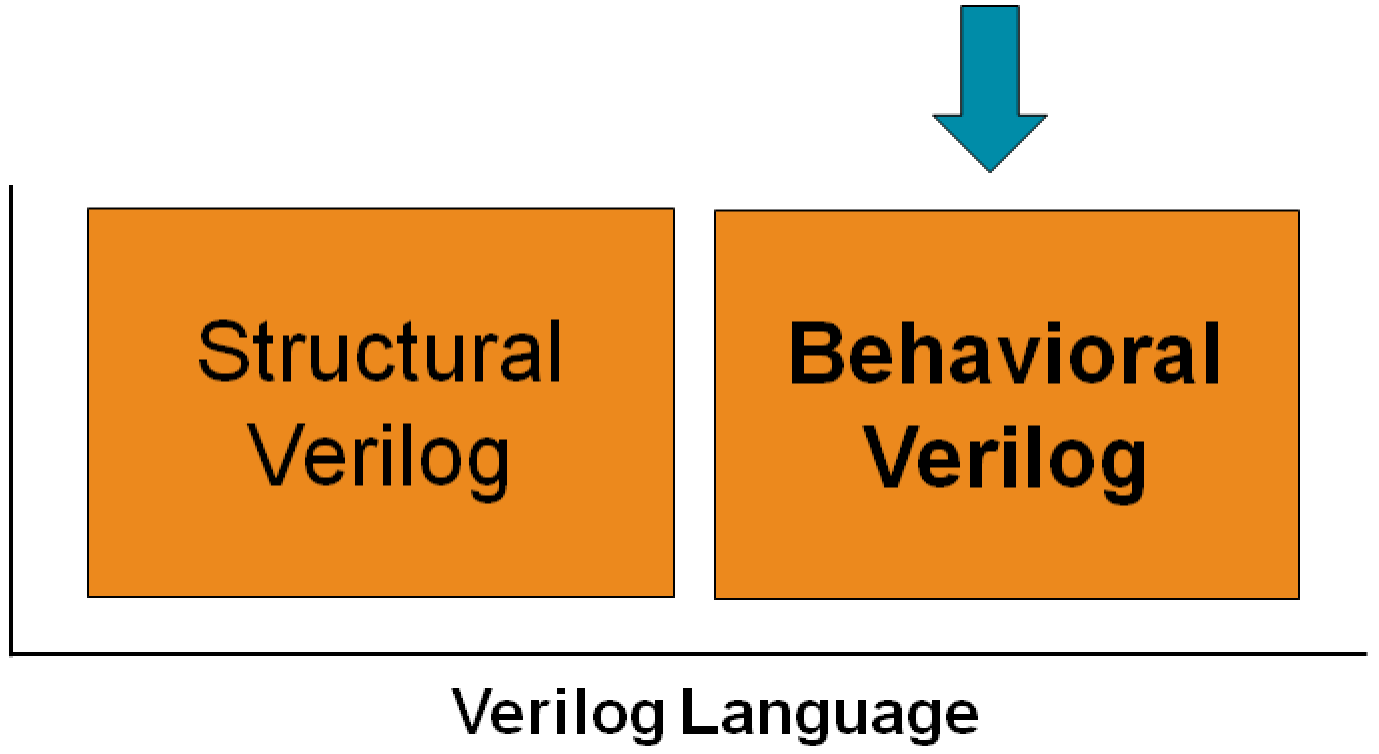

ALL PROGRAMMABLE
ANY MEDIA
ANY MACHINE
5G
4K/8K
ANY STANDARD
ANY NETWORK
5G Wireless • SDN/NFV • Video/Vision • ADAS • Industrial IoT • Cloud Computing

Verilog Procedural Statements

2015.3

Behavioral/Procedural Coding

- This module will start to present "the other half of the Verilog language"
- Can be used both for RTL and behavioral modeling



Objectives

After completing this module, you will be able to:

- Distinguish between structural and behavioral coding
- Use Verilog *initial* and *always* blocks
- Use time control in your procedural blocks
- Use Verilog blocking and non-blocking statements appropriately

Procedural Statements

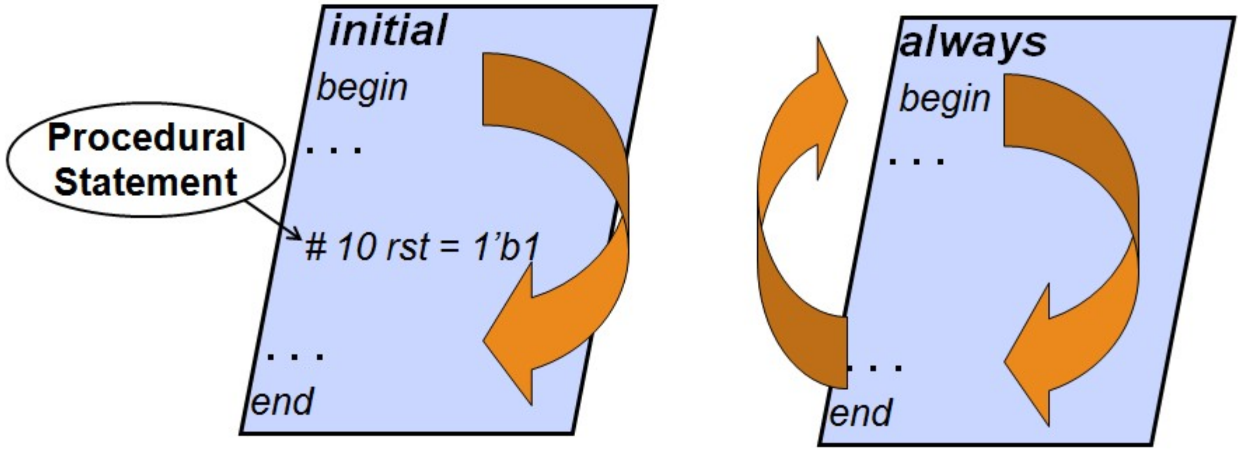
- **Procedural Statements**
- Time Control
- Blocking and Non-Blocking Assignments
- Summary

Procedures in Verilog

- A procedure is a block of code that specifies a set of operations to be performed in sequence
- A Verilog procedure is created using
  - An *initial* statement
  - An *always* statement
- There can be several procedures in a module
- Note: Statements within a procedure (between *begin* and *end*) are executed *sequentially*

Procedural Statements

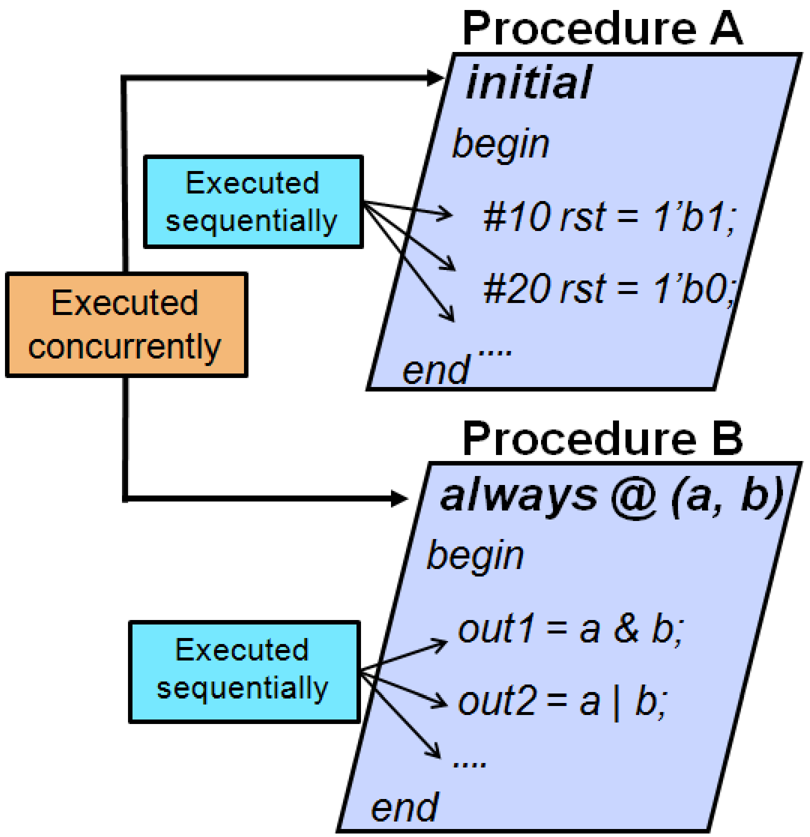
initial statement	always statement
Commonly used within the behavioral (simulation) environment	Used in both behavioral and RTL modeling
Executed only once during a simulation	Executed continually for the duration of a simulation



Statements within a procedure (between *begin* and *end*) are executed *sequentially*

Flow of Execution

- Each procedure executes *concurrently* with other procedures (unlike other languages like C) and continuous assign statements in the module
- Statements within a procedure (between *begin* and *end*) are executed *sequentially*
  - The time control constructs will be considered



Procedural Assignments

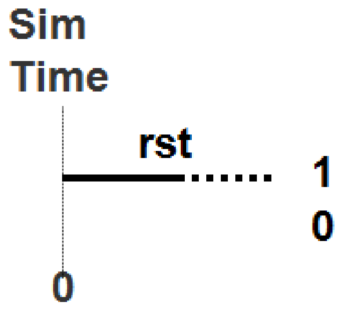
- Within the procedural block or statement, objects are updated with a procedural assignment. The assigned value will be maintained until it receives a new assignment
- Procedural assignments can only be made to variables—data types *reg*, *integer*, *real*, or *time*
  - Conversely, procedural assignments are not made to the data type *wire* because this is a class of net which is driven from a structural entity or continuous assign

```
reg [7:0] my_bus ;
. . .
initial
begin
 my_bus = 8’b11110000 ;
end
```

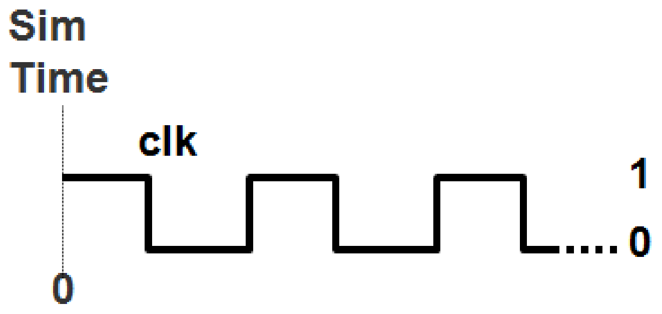
Initialization

- All initial statements are concurrent to each other, starting at simulation time 0 (zero)

```
module  dff_tb ( ) ;
reg rst;
. . .
initial
rst = 1'b1 ;
. . .
```

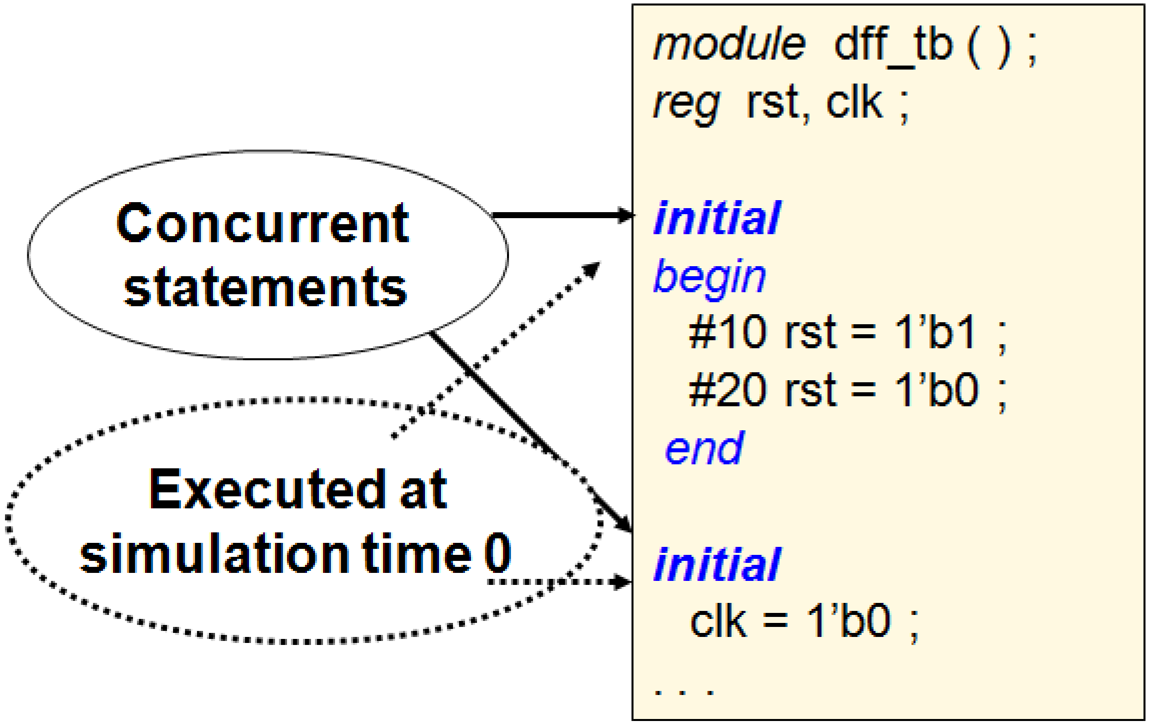


```
module  cntr_tb ( ) ;
reg clk ;
. . .
initial
begin
clk = 1'b1;
forever #10 clk = ~clk ;
end
```



initial Blocks

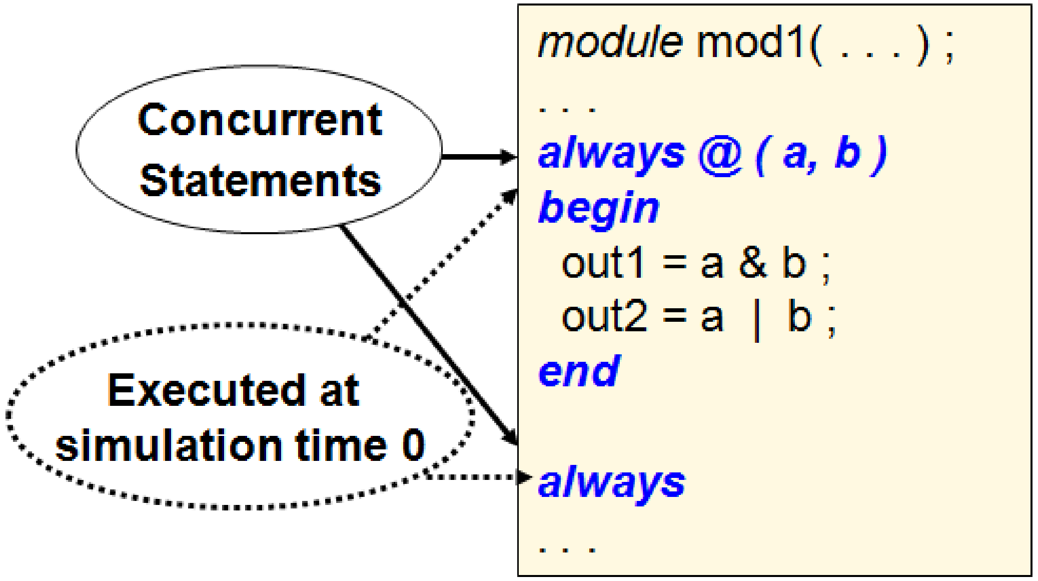
- Multiple statements following the *initial* construct must be enclosed within a block
  - In a sequential block, this is indicated with the keywords *begin* and *end*



```
module  dff_tb ( ) ;
reg  rst, clk ;

initial
begin
  #10 rst = 1'b1 ;
  #20 rst = 1'b0 ;
 end

initial
  clk = 1'b0 ;
. . .
```

always Blocks

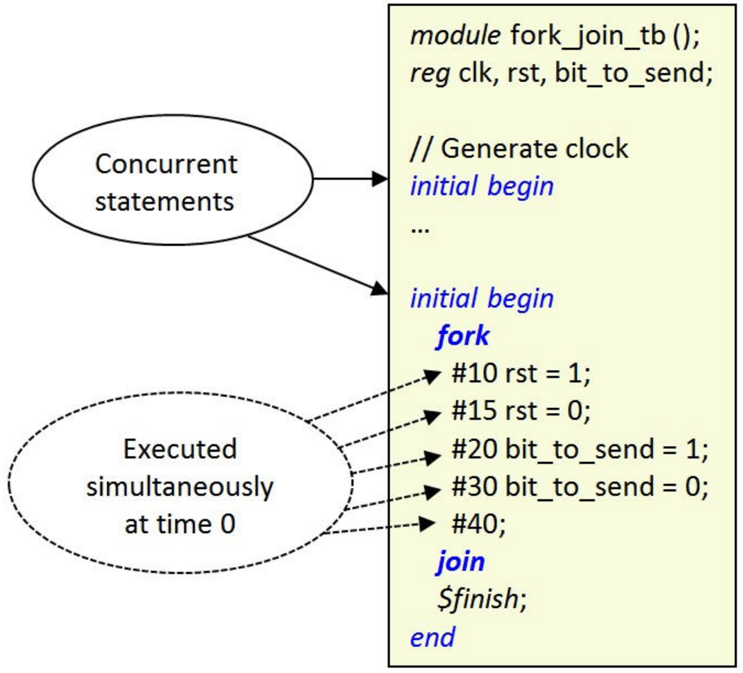
- All *always* statements are concurrent to each other, starting at simulation time 0 (zero)
- Multiple statements following the *always* construct must be enclosed within a block
  - In a sequential block, this is indicated with the keywords *begin* and *end*



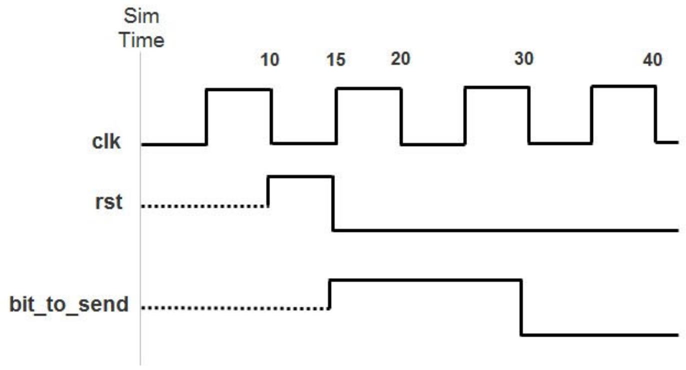
```
module mod1( . . . ) ;
. . .
always @ ( a, b )
begin
  out1 = a & b ;
  out2 = a  |  b ;
end

always
. . .
```

Initialization

- Enables each statement in the block to be executed simultaneously at the same time
- Not used in RTL code typically



```
module fork_join_tb ();
reg clk, rst, bit_to_send;

// Generate clock
initial begin
...

initial begin
  fork
   #10 rst = 1;
   #15 rst = 0;
   #20 bit_to_send = 1;
   #30 bit_to_send = 0;
   #40;
  join
  $finish;
end
```



Apply Your Knowledge

1. What is the primary difference between initial and always blocks?

2. Which of the following statements is not true?
   a) All always blocks are concurrent to each other
   b) An always block can be nested within another
   c) Multiples sequential statements require keywords begin/end
   d) All always blocks start executing at simulation time 0

Time Control

- Procedural Statements
- **Time Control**
- Blocking and Non-Blocking Assignments
- Summary

Timing Control

- Timing control provides a way to manage the execution of a process in simulation time
- Verilog provides three methods for timing control to block the execution using
  - Delay control, #
    - Mostly used in simulation environment
  - Event control, @
    - Used both in RTL and simulation environments
  - Wait statement
    - Mostly used in simulation environment

Delay Control

- Specifies the time duration between when a statement is encountered and when it is executed
- The # symbol indicates delay control
- Delay control statements are not synthesizable

```
Initial
begin
…
…
…
#5
…
…
…

end
```

Event-Based Control

- An event is defined as a change in logic value (1, 0, x, z) on a given object
- The @ symbol indicates event-based control
- When encountered in a procedure, the procedure will be blocked until the event occurs
- Can wait for multiple events
  - @(a or b) will wait for an transition on either a or b
  - Can also be written @(a,b)
- Can filter transitions using the *posedge* or *negedge* keyword
  - @(posedge a) waits for a positive edge on the signal a

```
initial
 begin
     ...
     ...
     @(a)
     ...
     ...
 end
```

Modeling Combinatorial Logic Using Event Control

- A key point when properly modeling combinatorial logic is to include all signals that are read in the procedural block in the sensitivity list. Verilog now supports three approaches to this
  - In Verilog '95 the keyword or is required as a separator
  - In Verilog '01, you can simply use a comma separated list
  - In Verilog '01, you can also effectively allow the compiler to determine the sensitivity, based on the signals read within the block

Verilog '95

```
always @ ( a or b or sel )
      if ( sel )   y = a ;
      else         y = b ;
```

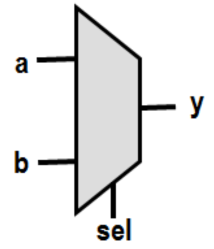
```
always @ ( a, b, sel )
      if ( sel )   y = a ;
      else         y = b ;
```

```
always @ *
      if ( sel )   y = a ;
      else         y = b ;
```

Event Control at the Top of Always Blocks

- Time controls can be placed at the beginning of a procedural block
  - After the initial/always and before the begin
- Causes the procedure to block before the execution of the first statement
- Some blocks of this structure describe the behavior of real hardware constructs
  - These blocks will be synthesizable
- The code to the right describes the behavior of a multiplexer and hence will infer a multiplexer when synthesized

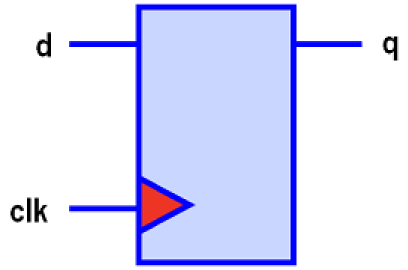
```
always @ ( a, b, sel )
 begin
     if ( sel )    y = a ;
     else          y = b ;
 end
```



Event-Control Examples

- Regular event

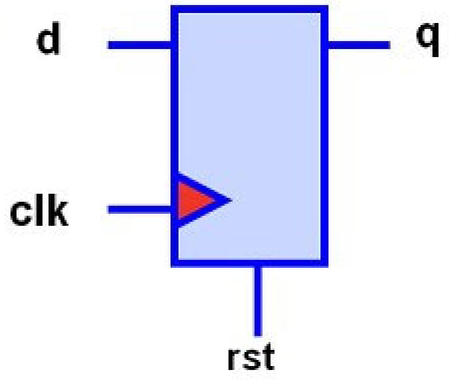
```
always @ ( posedge clk )
   q <= d ;
```



Event-Control Examples

- Modeling D-FF with active-high, asynchronous reset

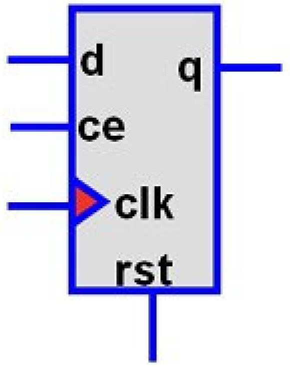
```
always @ ( posedge  clk,   posedge  rst )
  begin
     if (rst )     q <= 1'b0 ;
    else           q <= d ;
  end
```



Event-Control Examples

- Modeling D-FF with active-high asynchronous reset and chip enable

```
always @ ( posedge clk,  posedge rst )
  begin
    if ( rst )        q <= 1'b0 ;
    else if ( ce )  q <= d ;
  end
```



Modeling D-Type Flip-Flops Using Event Control

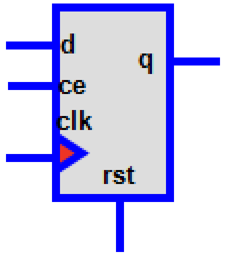
- The following examples show the code for inferring various different flip-flops

Active low async clear

```
always @ ( posedge clk,  negedge rst )
  begin
    if ( rst == 1'b0 )    q <= 1'b0 ;
    else                  q <= d ;
  end
```

Active high sync set

```
always @ ( posedge clk  )
  begin
    if ( rst== 1'b1 )    q <= 1'b1 ;
    else                 q <= d ;
  end
```



Active low async clear, active high CE

```
always @ ( posedge clk,  negedge rst )
  begin
    if ( rst == 1'b0 )           q <= 1'b0 ;
    else if ( ce == 1'b1 )       q <= d ;
  end
```

Active low sync reset, active high CE

```
always @ ( posedge clk  )
  begin
    if        ( rst == 1'b0)     q <= 1'b0 ;
    else if  ( ce ==  1'b1 )    q <= d ;
  end
```

wait Construct

- A *wait* construct is the third means by which you can control execution of a procedural block
  - The *wait* construct is not supported in synthesis, but can be useful in simulation for synchronizing procedural blocks

```
reg rst_en = 0;
always @ ( posedge clk )
  begin
  if  ( force_rst == 1'b1 )
     rst_en <= 1'b1 ;
 else
     q  <= d ;
 end
```

```
Initial
begin
...
...
wait ( rst_en )
...
...
end
```