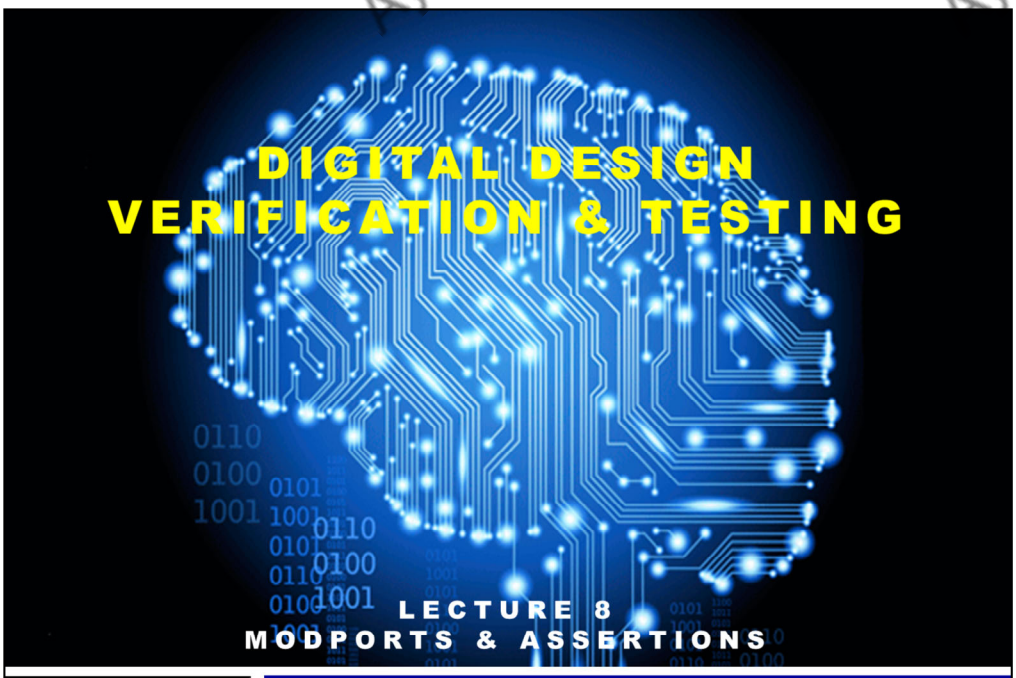
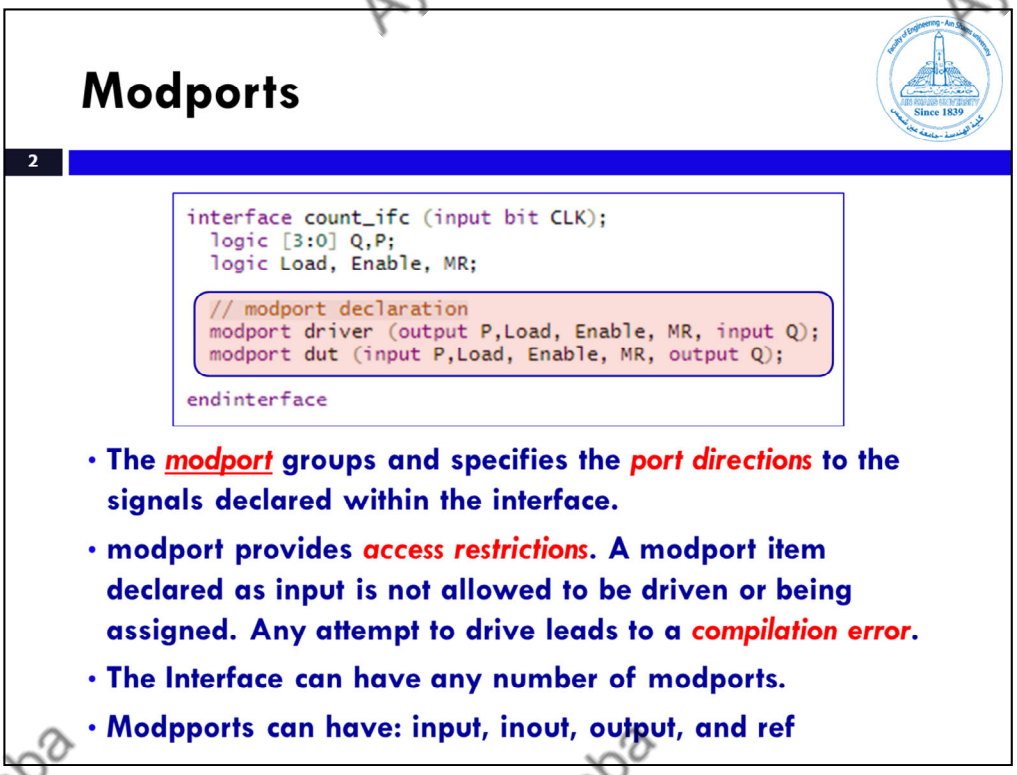


CSE 313s

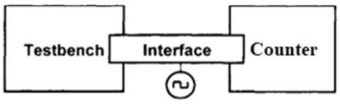
Ayman wahba



- The *modport* (module ports) groups and specifies the *port directions* to the signals declared within the interface.
- **modports** are declared inside the interface with the keyword **modport**.
- By specifying the port directions, **modport** provides *access restrictions*.
- The keyword **modport** indicates that the directions are declared (as if inside the module).
- **modport** item declared as input is not allowed to be driven or being assigned. Any attempt to drive leads to a *compilation error*.
- The Interface can have any number of modports, the signals declared in the interface can be grouped in many modports
- Modports can have, input, inout, output, and ref

Modports (Example)

3



Interface

```
1  interface count_ifc (input bit CLK);
2    logic [3:0] Q,P;
3    logic Load, Enable, MR;
4    modport driver (output P,Load, Enable, MR, input Q);
5    modport dut (input P,Load, Enable, MR, output Q);
6  endinterface
```

Testbench

```
6  module test(count_ifc x);
7    initial begin
8      x.P <= 4'b0111;
9      x.MR <= 1'b0;
10     x.Enable <= 1'b1;
11     x.Load <= 1'b0;
12     #3 x.MR <= 1'b1;
13     #6 x.MR <= 1'b0;
14     #43 x.Enable <= 1'b0;
15     #15 x.Enable <= 1'b1;
16     #16 x.Load <= 1'b1;
17     #9 x.Load <= 1'b0;
18   end
19 endmodule
```

Design

```
2  module decade_counter (count_ifc y);
3    always @(y.MR) begin
4      if (y.MR)
5        y.Q <= 4'b0000;
6    end
7
8    always @(posedge y.CLK) begin
9      if (!y.MR)
10       if (y.Load)
11         y.Q <= y.P;
12     else if(y.Enable)
13         y.Q <= (y.Q+1) % 10;
14   end
15 endmodule
```

Top module

```
18 module top;
19   bit clk;
20   always #5 clk <= ~clk;
21   count_ifc ifc(clk);
22   decade_counter u1(ifc.dut);
23   test u2(ifc.driver);
24
25   initial begin
26     $dumpfile("counter.vcd");
27     $dumpvars;
28     #200 $finish;
29   end
30 endmodule
```

- Note that when you make an instance from the DUT or the testbench, you select the convenient modport, by the following syntax:

  interface_name.modport_name

Modports (Another example)

4

```
interface intf();
  //declaring the signals
  logic [3:0] a;
  logic [3:0] b;
  logic [4:0] c;
  modport driver (output a, b, input c);
  modport dut(input a,b, output c);
endinterface
```

```
module adder(intf xyz);
  assign xyz.c = xyz.a + xyz.b;
endmodule
```

```
module top;
  intf i1();
  adder a1(i1.dut);
  tbench t1(i1.driver);
endmodule
```

```
module tbench(intf abc);
  initial
  begin
    abc.a = 6;
    abc.b = 4;
    $display("Value of a = %0d, b = %0d",abc.a,abc.b);
    #5;
    $display("Sum of a and b = %0d",abc.c);
    $finish;
  end
endmodule
```

- T

Stimulus timing

5

- The timing between the testbench and the design must be carefully orchestrated.
- Drive too late or sample too early, and your testbench is off a cycle.
- Race conditions may arise, such as when a signal is both read and written at the same time. Do you read the old value, or the one just written?
- SystemVerilog has several constructs to help you control the timing of the communication.

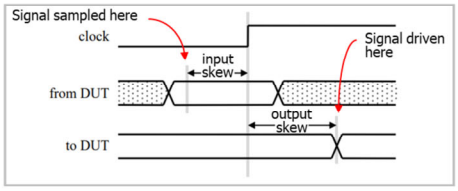
- The timing between the testbench and the design must be carefully orchestrated. You need to drive and receive the synchronous signals at the proper time in relation to the clock.
- Drive too late or sample too early, and your testbench is off a cycle.
- Even within a single time slot (for example, everything that happens at time 100 ns), mixing design and testbench events can cause a race condition, such as when a signal is both read and written at the same time. Do you read the old value, or the one just written?
- SystemVerilog has several constructs to help you control the timing of the communication.

Clocking Blocks

6

```
clocking cb @(posedge clk);
default input #10ns output #2ns;

output read,enable,addr;
input negedge data;
endclocking
```



- **Module ports and interfaces by default do not specify any timing requirements or synchronization schemes between signals.**
- **A clocking block does exactly that. It helps to specify the timing requirements between the clock and the signals.**
- **A testbench can have many clocking blocks, but only one per clock.**
- **The delay value represents a skew of how many time units away from the clock event a signal is to be sampled or driven.**

Signal directions inside a clocking block are with respect to the testbench and not the DUT.

- Module ports and interfaces by default do not specify any timing requirements or synchronization schemes between signals.
- A clocking block defined between ***clocking*** and ***endcocking*** does exactly that.
- It is a collection of signals synchronous with a particular clock and helps to specify the timing requirements between the clock and the signals.
- This would allow test writers to focus more on transactions rather than worry about when a signal will interact with respect to a clock.
- A testbench can have many clocking blocks, but only one block per clock.
- The delay_value represents a skew of how many time units away from the clock event a signal is to be sampled or driven.
- If a default skew is not specified, then all input signals will be sampled #1step and output signlas driven 0 ns after the specified event.

Controlling the timing by a clocking block (Example)

7

```
clocking ck1 @ (posedge clk);
  default input #5ns output #2ns;

  input data, valid, ready;
  output x, y;

  output negedge grant;
  input #1step addr;
endclocking
```

This is a clocking block called *ck1* that is to be clocked on the positive edge of the signal *clk*.

- **All signals in the block shall use a 5ns input skew and a 2ns output skew by default.**
- **The output *grant* is driven with the negative edge, overwriting the posedge stated in the first line.**
- **The last line contains *#1step* which overrides the skew of #5ns. Here #1step means the delay defined by the *timeprecision***

- SystemVerilog adds the clocking block that *identifies clock signals*, and *captures the timing and synchronization requirements* of the blocks being modeled.
- A clocking block assembles signals that are synchronous to a particular clock, and makes their timing explicit.
- The clocking block is a key element in a cycle-based methodology, which enables users to write testbenches at a higher level of abstraction. Simulation is faster with cycle based methodology.
- Depending on the environment, a testbench can contain one or more clocking blocks, each containing its own clock plus an arbitrary number of signals.
- If an input skew is specified then the signal is sampled at skew time units before the clock event. If output skew is specified, then output (or inout) signals are driven skew time units after the corresponding clock event. A skew must be a constant expression, and can be specified as a parameter.
- If skew is not specified, default input skew is 1 step and output skew is 0.
- Specifying a clocking block using a SystemVerilog interface can significantly reduce the amount of code needed to connect the TestBench without race condition (races are explained later).

Controlling the timing by a clocking block (case study)

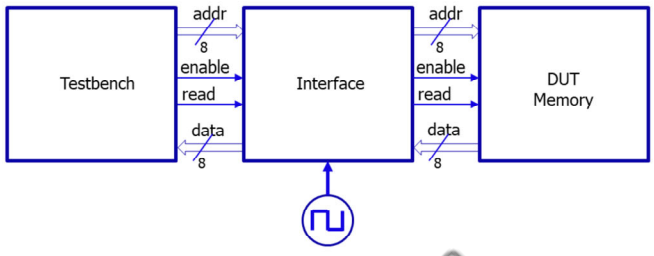
8

```
interface intf (input clk);
  logic read, enable;
  logic [7:0] addr,data;

  // clocking block for testbench
  clocking cb @(posedge clk);
    default input #10ns output #2ns;
    output read,enable,addr;
    input data;
  endclocking

  modport dut (input read,enable,addr,output data);
  // Synchronous testbench modport
  modport tb (clocking cb);
endinterface :intf
```

```
module memory(intf abc);
  logic [7:0] mem [256];
  initial begin
    foreach (mem[i])
      mem[i] = i >> 1;
  end
  always @(abc.enable,abc.read) begin
    if (abc.enable == 1 && abc.read == 1)
      abc.data = mem[abc.addr];
  end
endmodule
```



- An interface block can use a *clocking block* to specify the timing of synchronous signals relative to the clocks.
- Any signal in a clocking block is now driven or sampled synchronously, ensuring that your testbench interacts with the signals at the right time.
- If you have multiple clock domains, an interface can contain multiple clocking blocks, one per clock domain, as there is single clock expression in each block.
- Typical clock expressions are @(posedge clk) for a single edge clock and @(clk) for a DDR (double data rate) clock.
- Once you have defined a clocking block, your testbench can wait for the clocking expression with @<interface name>.cb rather than having to spell out the exact clock and edge.
- Now if you change the clock or edge in the clocking block, you do not have to change your testbench.

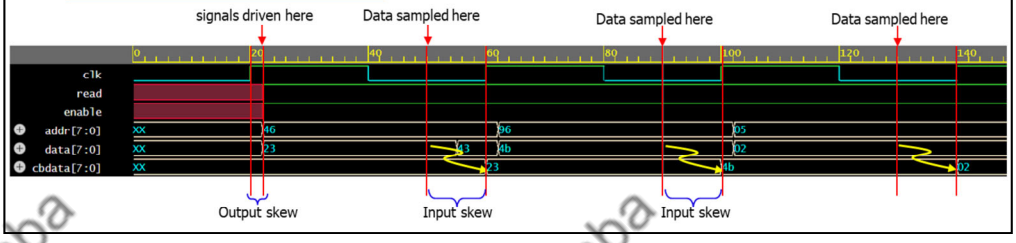
Controlling the timing by a clocking block (case study results)

9

```
15 module testbench(intf xyz);
16   logic[7:0] cbdata;
17   initial begin
18     xyz.cb.read <= 1;    // driving a synchronous signal
19     xyz.cb.enable <= 1; // driving a synchronous signal
20     xyz.cb.addr <= 70;  // driving a synchronous signal
21     #30 xyz.cb.addr <= 150;
22     #25 xyz.data <= 67; // disturbing the DUT data
23     #40 xyz.cb.addr <= 5;
24   end
25   always @(xyz.cb)
26     cbdata = xyz.cb.data;// get the sampled data
27 endmodule
```

```
26 module top;
27   bit clk = 0;
28   always #20 clk = ~clk;
29
30   intf i1(clk);
31   memory m1(i1.dut);
32   testbench t1(i1.tb);
33
34   initial begin
35     $dumpfile("uvm.vcd");
36     $dumpvars;
37     #200 $finish;
38   end
39 endmodule
```

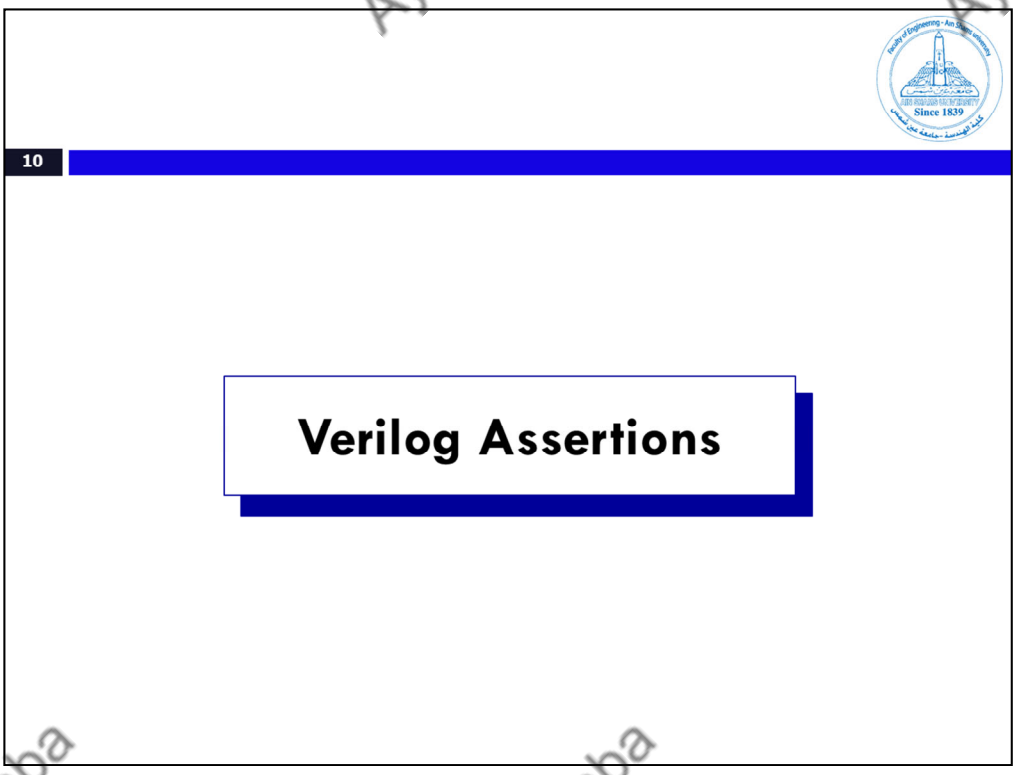
```
// clocking block for testbench
clocking cb @(posedge clk);
  default input #10ns output #2ns;
  output read,enable,addr;
  input data;
endclocking
```



- Here the test bench uses signals that are synchronized by the clocking block named cb.
- Note that the values of the testbench outputs () are assigned values at the times given the table below, however they are actually assigned 2 ns after the rising edge of the clock as specified by the clocking block.

Output name	Time of assignment	Actual driving time
read	0	22
enable	0	22
addr	0	22
addr	30	62
addr	95	102

- Also note that the data coming from the DUT is sampled 10ns before the rising clock edge. So the sampled value is not the value just before the clock edge, but the existing value 10 ns before the edge.
- The samples value appears at the rising clock edge. For example, the value appearing at time 60ns, is the DUT data value at time 50ns.



- Ver

System Verilog Assertion

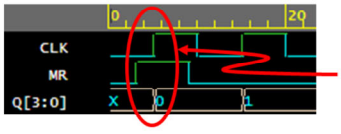
11

- Assertions are pieces of code that check the relationships between design signals, either *once* or *over a period of time*.
- Assertions are instantiated similarly to other design blocks and are active for the entire simulation.
- The most familiar assertions look for errors such as two signals that should be mutually exclusive for example
- The simulator keeps track of what assertions have triggered, and so you can gather functional coverage data on them.
- There are two types of assertions: *Immediate* & *Concurrent*

- Assertions are pieces of code that check the relationships between design signals, either *once* or over *a period of time*. You can create temporal assertions about signals in your design using SystemVerilog Assertions (SVA).
- The most familiar assertions look for errors such as two signals that should be mutually exclusive or a request that was never followed by a grant. These error checks should stop the simulation as soon as they detect a problem.
- Assertions are instantiated similarly to other design blocks and are active for the entire simulation.
- The simulator keeps track of what assertions have been triggered, and so you can gather functional coverage data on them.
- There are two types of assertions: Immediate Assertions, and Concurrent Assertions. These two types will be explained in the following slides.

Counter with an Error

12



Synchronous not Asynchronous

```
1  interface count_ifc (input bit CLK);
2    logic [3:0] Q,P;
3    logic Load, Enable, MR;
4    modport driver (output P,Load, Enable, MR, input Q);
5    modport dut (input P,Load, Enable, MR, output Q);
6  endinterface
```

Interface

```
1  module decade_counter (count_ifc y);
2  //  always @(y.MR) begin
3  //    if (y.MR)
4  //      y.Q <= 4'b0000;
5  //  end
6
7    always @(posedge y.CLK) begin
8      if (y.MR)
9        y.Q <= 4'b0000;
10     else if(y.Load)
11       y.Q <= y.P;
12     else if(y.Enable)
13         y.Q <= (y.Q+1) % 10;
14   end
15 endmodule
```

Design

```
6  module test(count_ifc x);
7    initial begin
8      x.P <= 4'b0111;
9      x.MR <= 1'b0;
10     x.Enable <= 1'b1;
11     x.Load <= 1'b0;
12     #3 x.MR <= 1'b1;
13     #6 x.MR <= 1'b0;
14     #43 x.Enable <= 1'b0;
15     #15 x.Enable <= 1'b1;
16     #16 x.Load <= 1'b1;
17     #9 x.Load <= 1'b0;
18   end
19 endmodule
```

Testbench

```
18 module top;
19   bit clk;
20   always #5 clk <= ~clk;
21   count_ifc ifc(clk);
22   decade_counter u1(ifc.dut);
23   test u2(ifc.driver);
24
25   initial begin
26     $dumpfile("counter.vcd");
27     $dumpvars;
28     #200 $finish;
29   end
30 endmodule
```

Top module

- Here, the designer mistakenly made the Master Reset Synchronous.
- As shown in the waveform, resetting the output is synchronized with the edge of the clock. The specifications states that the reset is an asynchronous reset.
- How can assertions help revealing this error?

Immediate Assertions

13

```
8  module test(count_ifc x);
9    initial begin
10     x.P <= 4'b0111;
11     x.MR <= 1'b0;
12     x.Enable = 1'b1;
13     x.Load = 1'b0;
14     #3 x.MR = 1'b1;
15
16     assertion1: assert(x.Q == 4'b0000);
17
18     #6 x.MR = 1'b0;
19     #43 x.Enable = 1'b0;
20     #15 x.Enable = 1'b1;
21     #16 x.Load = 1'b1;
22     #9 x.Load = 1'b0;
23   end
24 endmodule
```

Testbench

If the signal does not have the expected value, the simulator produces a message like this

```
"testbench.sv", 16: top.u2.assertion1: started at 3ns failed at 3ns
        Offending '(x.Q == 4'b0)'
```

- The immediate assertion statement is a test of an expression performed when the statement is executed in the procedural code.
- *Immediate* assertions are simple *non-temporal* domain assertions that are executed like statements in a procedural block.
- The assertion condition is non-temporal, which means its execution computes and reports the assertion results at the same time.
- If the expression evaluates to X, Z or 0, then it is interpreted as being false and the assertion is said to fail.
- Otherwise, the expression is interpreted as being true and the assertion is said to pass.
- In the above code we are using the assertion to test whether the output Q is reset immediately after MR is activated.
- The message says: on line 16 of the file testbench.sv, the assertion *top.u2.assertion1* started at 3 ns to check the signal *x.Q*, failed.

Customizing the Assertion Actions

14

```
8  module test(count_ifc x);
9    initial begin
10     x.P <= 4'b0111;
11     x.MR <= 1'b0;
12     x.Enable = 1'b1;
13     x.Load = 1'b0;
14     #3 x.MR = 1'b1;
15
16     assertion1: assert(x.Q == 4'b0000)
17       $display("The output value is correct");
18     else
19       $fatal("The reset is not synchronous");
20
21     #6 x.MR = 1'b0;
22     #43 x.Enable = 1'b0;
23     #15 x.Enable = 1'b1;
24     #16 x.Load = 1'b1;
25     #9 x.Load = 1'b0;
26   end
27 endmodule
```

If the assertion fails:

```
"testbench.sv", 16: top.u2.assertion1: started at 3ns failed at 3ns
        Offending '(x.Q == 4'b0)'
Fatal: "testbench.sv", 16: top.u2.assertion1: at time 3 ns
The reset is not synchronous
```

If the assertion succeeds:

```
The output value is correct
```

- An immediate assertion has optional *then* and *else* clauses.
- If you want to augment the default message, you can add your own.
- The above example shows how to create a custom error message in an immediate assertion.
- If Q does not have the expected value. you'll see an error message as shown above.
- SystemVerilog has four functions to print messages:
  - $info,
  - $warning,
  - $error
  - $fatal.
- You can use the then-clause to record when an assertion completed successfully.

Severity System Tasks

($info, $warning, $error, $fatal)

15

```
module test_module;

  initial begin
    #5 $info("Starting simulation...");
    #5 $warning("This is a warning message.");
    #5 $error("This is an error message.");
    #5 $fatal("This is a fatal error message."); // Simulation will terminate here
    #5 $info("Simulation ended."); // This will not be executed due to the previous $fatal
  end

endmodule
```

```
Info: "testbench.sv", 4: test_module: at time 5 ns
Starting simulation...
Warning: "testbench.sv", 5: test_module: at time 10 ns
This is a warning message.
Error: "testbench.sv", 6: test_module: at time 15 ns
This is an error message.
Fatal: "testbench.sv", 7: test_module: at time 20 ns
This is a fatal error message.
$finish called from file "testbench.sv", line 7.
$finish at simulation time                   20
           V C S   S i m u l a t i o n   R e p o r t
```

- In System Verilog, severity of assertion messages is classified by using four system tasks. These are *$fatal*, *$error*, *$warning* and *$info*.
- If an action block is specified, user-defined severity can be created by using these system tasks.
- Every assertion failure has an associated severity which can be specified in the fail-statement block.
- If assertion does not have a fail-statement block, then by default $error system task will be called with default message.
- The syntax of severity system tasks except $fatal is identical to $display, but for $fatal, an optional error code can be added as a first argument before the message to be displayed.
  - **$fatal:** The $fatal *ends the simulation*, that means it has an *implicit call of $finish*.
  - **$warning:** This generates a run time warning message, without terminating the simulation
  - **$error:** This generates a run time error that does not terminate the simulation.
  - **$info:** This generates information to the user and indicates that the assertion failure carries no specific severity.

Concurrent Assertions

16

- A concurrent assertion is like a small model that runs continuously, checking the values of signals for the entire simulation, only at the occurrence of a clock tick.
- The verification of a design may be specified using statements as shown in the following examples:
  - A *Request* should be followed by an *Acknowledge* occurring *no more than two clocks* after the request is asserted.

    assert property (@(posedge clk) Req |-> ##[1:2] Ack);
- Concurrent assertions can be specified in a module, interface or program block running concurrently with other statements.

- The other type of assertions is the *concurrent assertion* that you can think of as a small model that runs continuously, checking the values of signals for the entire simulation.
- Concurrent assertions are written as follows: *assert property*, unlike immediate assertions which are written as follows: *assert*
- The behaviour of a design may be specified using statements similar to these:
  - A Request should be followed by an Acknowledge occurring no more than two clocks after the request is asserted. You can check that using an assertion like this one:

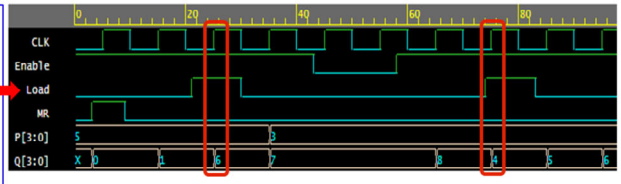
    assert property (@(posedge clk) Req |-> ##[1:2] Ack);

    We use the implication operator |-> to express some event implies some other thing. ##[1:2] means after one or two clock cycles

Concurrent Assertions

17

```
1  module decade_counter (count_ifc y);
2    always @(y.MR) begin
3      if (y.MR)
4        y.Q <= 4'b0000;
5    end
6
7    always @(posedge y.CLK) begin
8      if (!y.MR)
9        if (y.Load)
10         y.Q <= y.P+1;
11       else if (y.Enable)
12         y.Q <= (y.Q+1) % 10;
13   end
14 endmodule
```




```
8  module test(count_ifc x);
9    initial begin
10     x.P <= 4'b0101; x.MR <= 1'b0; x.Enable = 1'b1; x.Load = 1'b0;
11     #3 x.MR = 1'b1;      #6 x.MR = 1'b0;
12     #12 x.Load = 1'b1;   #9 x.Load = 1'b0;
13     #5 x.P = 4'b0011;
14     #8 x.Enable = 1'b0; #15 x.Enable = 1'b1;
15     #16 x.Load = 1'b1;   #9 x.Load = 1'b0;
16   end
17
18   assertion1: assert property (@(posedge x.CLK) x.Load |=> (x.Q == x.P));
19
20 endmodule
```

```
"testbench.sv", 18 : top.u2.assertion1: started at 25ns failed at 35ns
        Offending '(x.Q == x.P)'
"testbench.sv", 18 : top.u2.assertion1: started at 75ns failed at 85ns
        Offending '(x.Q == x.P)'
```

- Here is a simple assertion to check that whenever the load line is active, then at the very next rising edge of the clock the counter is loaded by the values found on the P inputs.
- It is clear that there is an error in the design, as the value loaded is P+1 not P, so the assertion fails.
- It is also clear that, as the assertion is a *concurrent* assertion, it is active during the *whole simulation time*, and whenever it fails an error message is issued.
- In the shown simulation the assertion fails twice. One time at 25 ns, and the second time at 75 ns.
- In the previous slide we used "|->" as an implication operator, whereas in the above example we used "|=>". So is there any difference between them ?

Types of Implication operators

18

- The left-hand side of the implication is called the *antecedent* and the right-hand side is called the *consequent*
- If the antecedent succeeds, then the consequent is evaluated.
- There are two types of implication operators:
  - Overlapped implication (|->)
  - Non-overlapped implication (|=>)
- The implication construct can be *used only with property definitions.*

- The implication is equivalent to an if-then structure.
- The left-hand side of the implication is called the “antecedent” and the right-hand side is called the “consequent.”
- The antecedent is the gating condition. If the antecedent succeeds, then the consequent is evaluated.
- The implication construct can be used only with property definitions.
- There are 2 types of implications:
  - Overlapped implication
  - Non-overlapped implication

Overlapped Implication

19

- The overlapped implication is denoted by the symbol |->.
- If there is a match on the antecedent, then the consequent expression is evaluated *in the same clock cycle*.

```
1 property p;
2   @(posedge clk) a |-> b;
3 endproperty
4
5 a: assert property(p);
```

This property checks, if signal *a* is high on a given positive clock edge, then signal *b* should also be high *on the same clock edge*.

- The overlapped implication is denoted by the symbol |->.
- If there is a match on the antecedent, then the consequent expression is evaluated *in the same clock cycle*.
- The shown property checks that, if signal “a” is high on a given positive clock edge, then signal “b” should also be high *on the same clock edge*.

Non-Overlapped Implication

20

- The non-overlapped implication is denoted by the symbol |=>.
- If there is a match on the antecedent, then the consequent expression is evaluated *in the next clock cycle*.

```
1 property p;
2   @(posedge clk) a |=> b;
3 endproperty
4
5 a: assert property(p);
```

This property checks, if signal *a* is high on a given positive clock edge, then signal *b* should also be high *on the next clock edge*.

- The non-overlapped implication is denoted by the symbol |->.
- If there is a match on the antecedent, then the consequent expression is evaluated *in the next clock cycle*.
- The shown property checks that, if signal “a” is high on a given positive clock edge, then signal “b” should also be high *on the next clock edge*.

Implication with a fixed delay on the consequent

21

- Delays for a certain number of clock cycles is expressed in implications by ## followed by the number of cycles.

```
1 property p;
2   @(posedge clk) a |-> ##2 b;
3 endproperty
4
5 a: assert property(p);
```

This property checks, if signal *a* is high on a given positive clock edge, then signal *b* should be high after *exactly 2 clock cycles*.

- If there is a match on the antecedent, then the consequent expression is evaluated *after certain number of cycles* expressed by ## followed by the number of cycles.

Timing windows in SVA checkers

22

- Delays for a certain *range* of clock cycles is expressed in implications by ## followed by a range of cycles between square brackets.

```
1 property p;
2   @(posedge clk) a |-> ##[1:4] b;
3 endproperty
4
5 a: assert property(p);
```

This property checks, if signal *a* is high on a given positive clock edge, then *within 1 to 4 clock cycles*, the signal *b* should be high. You can write the range as [0:4] if you want *b* to be high in the same cycle, or within 4 clock cycles

- If there is a match on the antecedent, then the consequent expression is evaluated *after certain range of cycles* expressed by ## followed by a range of cycles.
- In the above example, if you want b to be high in the same clock cycle, or within 4 cycles, we write the range as ##[0:4]

Infinite timing window

23

- The upper limit of the timing window specified in the right-hand side can be defined with a $ sign which implies that there is no upper bound for timing.
- This is called the *eventuality operator*. The checker will keep checking for a match until the end of the simulation.

```
1 property p;
2   @(posedge clk) a |-> ##[1:$] b;
3 endproperty
4
5 a: assert property(p);
```

This property checks, if signal *a* is high on a given positive clock edge, then signal *b* will be high *eventually* starting from the next clock cycle.

- The upper limit of the timing window specified in the right-hand side can be defined with a "$" sign which implies that there is no upper bound for timing.
- This is called the *eventuality* operator. The checker will keep checking for a match until the end of the simulation.
- The shown property checks that, if signal *a* is high on a given positive clock edge, then signal *b* will be high *eventually* starting from the next clock cycle.

Examples (1)

($rose)

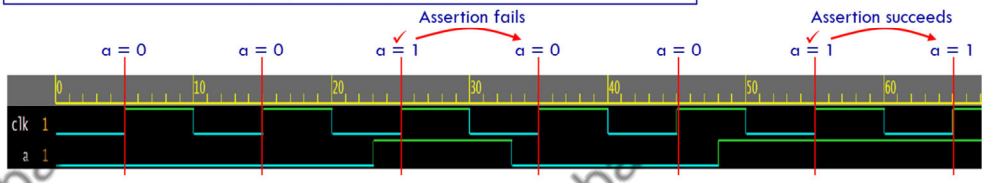
24

```
1  module test_module;
2    bit a,clk;
3
4    always #5 clk = ~clk;
5
6    initial begin
7      #23 a = 1; #10 a = 0; #15 a = 1;
8    end
9
10   property high;
11     @(posedge clk) $rose(a) |=> a;
12   endproperty
13   assert property (high)
14     else $error("a is not high after rise");
15
16 endmodule
```

- **The assertion says that if $rose(a) is true, then at the next clock edge, a should be 1.**
- **In the shown timing diagram, $rose(a) is true only at two instants:**
  - **at t = 25, and at t = 55.**

```
"testbench.sv", 13: test_module.unnamed$$_1: started at 25ns failed at 35ns
        Offending 'a'
Error: "testbench.sv", 13: test_module.unnamed$$_1: at time 35 ns
a is not high after rise
$finish called from file "design.sv", line 5.
```

Simulator output



- Here we have a property to check if a signal *b* remains low for at least 3 clock cycles.

Examples (2)

($fell, $stable)

25

```
1  module test_module;
2    bit clk, b=1, rst_n = 1;
3    always #5 clk = ~clk;
4
5    initial begin
6      #3 b = 0; #40 b = 1; #10 b = 0; #45 b = 1;
7    end
8
9    property low_for_4_cycles;
10     @(posedge clk) disable iff (!rst_n)
11     $fell(b) |=> $stable(b)[*4];
12   endproperty
13   assert property(low_for_4_cycles) else
14     $error("b is not low for at least 4 cycles");
15
16 endmodule
```

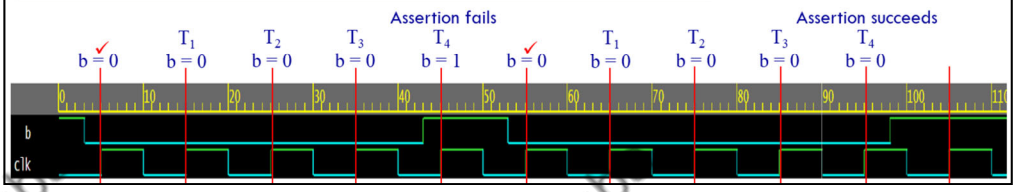
The assertion says that if $fell(b) is true, then $stable(b) should be 1 for four consecutive edges.

In the shown timing diagram, $fell(b) is true only at two instants:

- **at t = 5, and at t = 55.**

```
"testbench.sv", 13: test_module.unnamed$$_1: started at 5ns failed at 45ns
        Offending '$stable(b)'
Error: "testbench.sv", 13: test_module.unnamed$$_1: at time 45 ns
b is not low for at least 4 cycles
```

Simulator output



- When the system task $fell(b) is evaluated at certain clock edge, it compares the value of b at this clock edge with its value at the previous clock edge. If the new value is 0, and the old value is nonzero, then $fell evaluates to true.
- When the system task $stable(b) is evaluated at certain clock edge, it compares the value of b at this clock edge with its value at the previous clock edge. If the two values are the same, then $stable evaluates to true.
- $stable(b)[*4] means $stable should be true for 4 consecutive times.

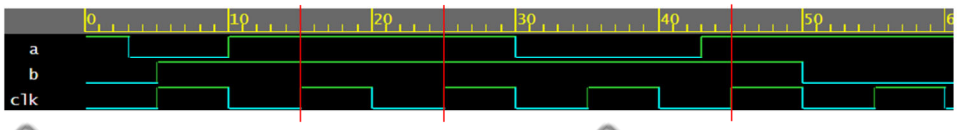
Examples (3)

26

```
1  module test_module;
2    bit clk, a=1, b=0;
3    always #5 clk = ~clk;
4
5    initial begin
6      #3 a = 0; #2 b = 1; #5 a = 1; #20 a = 0;
7      #13 a = 1;#7 b = 0;
8    end
9
10   property never_high_together;
11     @(posedge clk) !(a && b);
12   endproperty
13   assert property(never_high_together) else
14     $error("a and b are high together");
15
16 endmodule
```

```
"testbench.sv", 15: test_module.unnamed$$_1: started at 15ns failed at 15ns
        Offending '(!(a && b))'
Error: "testbench.sv", 15: test_module.unnamed$$_1: at time 15 ns
a and b are high together
"testbench.sv", 15: test_module.unnamed$$_1: started at 25ns failed at 25ns
        Offending '(!(a && b))'
Error: "testbench.sv", 15: test_module.unnamed$$_1: at time 25 ns
a and b are high together
"testbench.sv", 15: test_module.unnamed$$_1: started at 45ns failed at 45ns
        Offending '(!(a && b))'
Error: "testbench.sv", 15: test_module.unnamed$$_1: at time 45 ns
a and b are high together
```

Simulator output



- When

Examples (4)

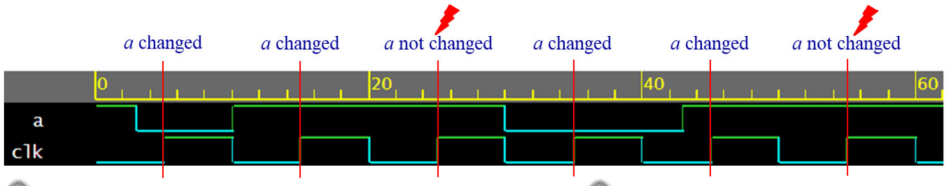
($changed)

27

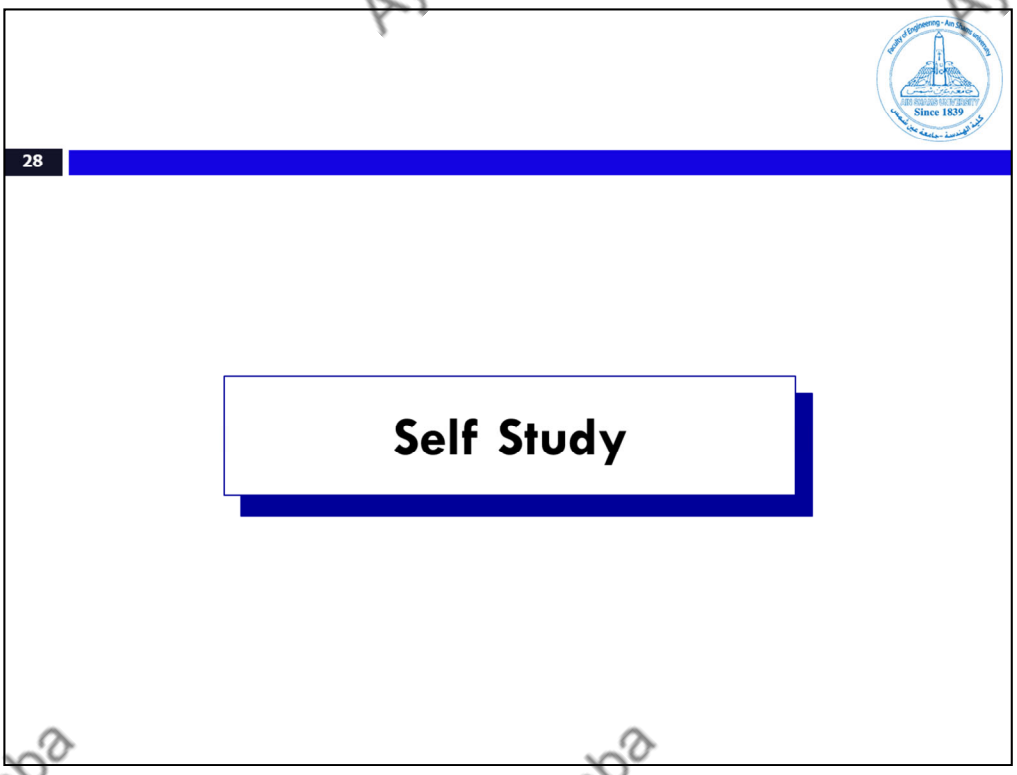
```
1  module test_module;
2    bit clk, a=1;
3    always #5 clk = ~clk;
4
5    initial begin
6      #3 a = 0; #7 a = 1; #20 a = 0; #13 a = 1;
7    end
8
9    property toggle_every_cycle;
10     @(posedge clk) $changed(a);
11   endproperty
12   assert property(toggle_every_cycle) else
13     $error("signal_g does not toggle every cycle");
14
15 endmodule
```

```
"testbench.sv", 12: test_module.unnamed$$_1: started at 25ns failed at 25ns
        Offending '$changed(a)'
Error: "testbench.sv", 12: test_module.unnamed$$_1: at time 25 ns
signal_g does not toggle every cycle
"testbench.sv", 12: test_module.unnamed$$_1: started at 55ns failed at 55ns
        Offending '$changed(a)'
Error: "testbench.sv", 12: test_module.unnamed$$_1: at time 55 ns
signal_g does not toggle every cycle
```

Simulator output



- When the system task $changed(a) is evaluated at certain clock edge, it compares the value of a at this clock edge with its value at the previous clock edge. If the the two values are different, then $changed evaluates to true.



- Ver

SVA System Functions

29

Function	Use
$rose	Returns true if the LSB of the expression changed to 1. Otherwise, it returns false.
$fell	Returns true if the LSB of the expression changed to 0. Otherwise, it returns false.
$stable	Returns true if the value of the expression did not change. Otherwise, it returns false.
$past(expression, num_cycles)	Returns value of expression from num_cycles ago
$countones	Returns the number of 1s in an expression
$onehot	Returns true if exactly one bit is 1. if no bits or more than one bit is 1, it returns false.
$onehot0	Returns true is no bits or just 1 bit in the expression is 1
$isunknown	Returns true if any bit in the expression is 'X' or 'Z'

- T

SVA Operators

30

Operator	Use
##n ##[m:n]	Delay operators - Fixed time interval and Time interval range
!, ||, &&	Boolean operators
|->	Overlapping implication
|=>	Nonoverlapping implication
not, or, and	Property operators

- T

SVA Repetition Operators

31

Operator	Use
[*n] [*m:n]	Continuous repetition operator. The expression repeats continuously for the specified range of cycles.
[->n] [->m:n]	Go to repetition operator. Indicates there's one or more delay cycles between each repetition of the expression. a[->3] is equivalent to (!a[*0:$] ##1 a) [*3]
[=n] [=m:n]	Non-consecutive implication. a[=3] is equivalent to (!a[*0:$] ##1 a ##1 !a[*0:$]) [*3]

- T