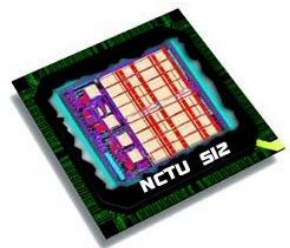


# SystemVerilog Verification

---

NYCU-EE IC LAB Autumn 2024

Lecturer: Zhen-Wei Liu



# Outline

## ✓ Section 1 Functional Coverage

- Coverpoint & Covergroup
- Specifying sample event timing
- Bin creation
- Options
- Coverage measurement

## ✓ Section 2 Assertion

- What is assertion
- Assertion types
- Sequence & Properties



# Outline

## ✓ Section 1 Functional Coverage

- Coverpoint & Covergroup
- Specifying sample event timing
- Bin creation
- Options
- Coverage measurement

## ✓ Section 2 Assertion

- What is assertion
- Assertion types
- Sequence & Properties



# Coverage

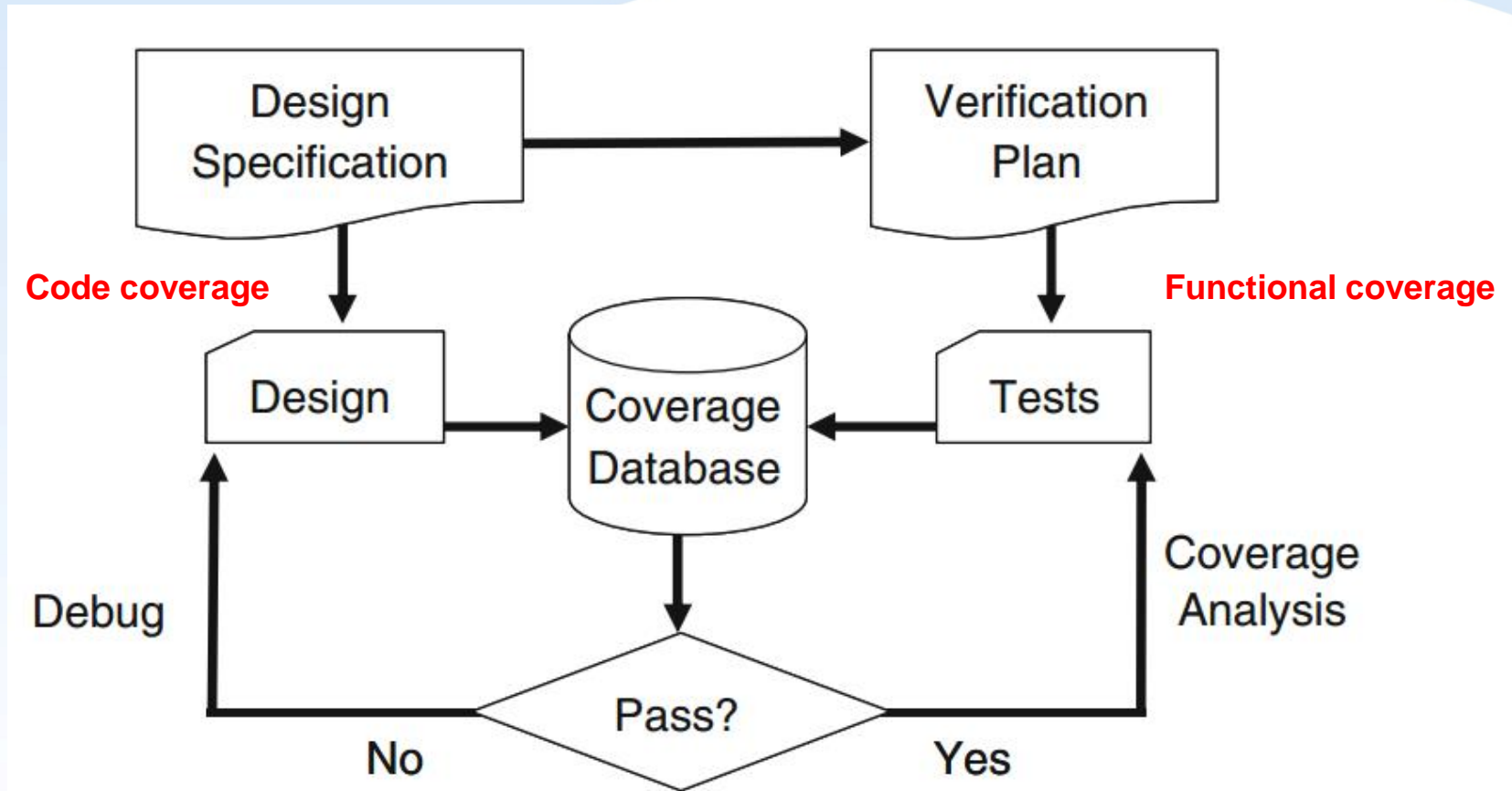
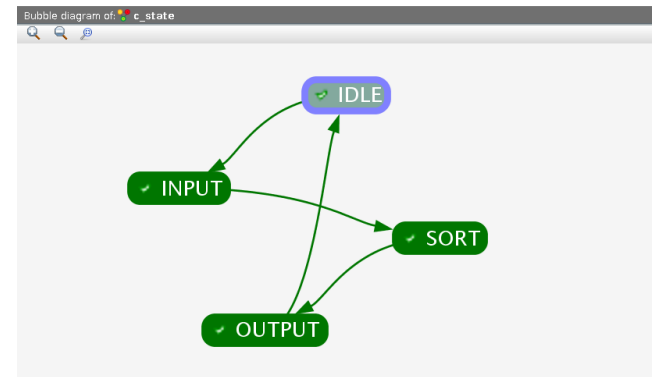


Fig. 9.2 Coverage flow

# Code Coverage

- ✓ Statement (line) coverage
- ✓ Block coverage
- ✓ Conditional/Expression coverage
- ✓ Branch/Decision coverage
- ✓ Toggle coverage
- ✓ FSM coverage



```
1 always @(posedge clock)
2   begin                                => Block 1 [always block]
3     if(x == y) begin                  => Block 2 [If block]
4       out1 = x+y;
5       out2 = x^2 + y^2;
6     else                              => Block 3 [Else block]
7       out1 = x;
8       out2 = y;
9   end
```

Hierarchy	Modules	Groups	Asserts	Statistics	Tests
* [icon]					
Name	Score	Line	Toggle	FSM	
ALU	100.00%	100.00%	100.00%	100.00%	
Control	100.00%	100.00%	100.00%	100.00%	100.00%
cpu	100.00%	100.00%	100.00%	100.00%	
ram16x8	100.00%	100.00%	100.00%	100.00%	
test	100.00%	100.00%	100.00%	100.00%	

# Functional Coverage

- ✓ Sample points are known as **cover point**
- ✓ A cover point can be an integral variable or an integral expression.
- ✓ Multiple cover points sample at the same time are placed together in a **cover group**
- ✓ A cover group can sample any visible variable directly such as program variables, signals from an interface, or any signal in the design (using a hierarchical reference).  
(see [Appendix A.](#))



# Functional Coverage Example

```
≡ CHECKER.sv 9+ ●
C: > Users > anson > Desktop > ≡ CHECKER.sv
1  module Checker(~);
2  logic [9:0] stock_price_2330;
3  logic [9:0] stock_price_2454;
4  /*
5   Suppose maximum stock_price_2330 is 1023, (10 bits)
6   however it is 255 now
7   we only care about if stock_price_2330 will go to 0
8   (go to hell)
9   Today is 2022/11/24 2330: 496, 2454: 728
10  */
11  covergroup ETF_0050 @(posedge wake_up_at_9_am);
12      Stock1 : coverpoint stock_price_2330
13      {
14          bins hell = {[0:495]};
15          bins paradise = {[496:1023]};
16      }
17      Stock2 : coverpoint stock_price_2454
18      {
19          bins hell = {[0:727]};
20          bins paradise = {[728:1023]};
21      }
22  endgroup
23
24  // remember to do this
25  ETF_0050 ETF_my_dream = new();
26
27  endmodule
```



# Functional Coverage in SystemVerilog

## ✓ Create a cover group which encapsulates:

- Group of cover points
- Bins definitions
- Coverage goal
- Defining Coverage bins sample timing
- Track progress

```
22  covergroup cg; //start
23  //...
24  //...
25  //...
26  endgroup //end
27
28  cg cg_inst = new(); //instance
```



# Specifying Sample Event Timing

- ✓ Define **sample\_event** in coverage\_group
- ✓ Valid **sample\_event\_definition**:
  - @( [specified\_edge] signals | variables )
- ✓ Bins are updated synchronously as the **sample\_event** occurs
  - Can also use **cov\_object.sample()** to update the bins

```
1  covergroup group_name (argument_list)@(sample_event);  
2      coverpoint cp1;  
3      coverpoint cp2 {...}  
4  endgroup
```

```
53  covergroup cov_grp @(negedge clk);  
54      cov_p1: coverpoint a;  
55  endgroup  
56  
57  cov_grp cov_inst = new();
```

```
46  covergroup cov_grp;  
47      cov_p1: coverpoint a;  
48  endgroup  
49  
50  cov_grp cov_inst = new();  
51  @(negedge clk) cov_inst.sample();
```



# IFF

## ✓ Event control

- Only be triggered when the expression after iff is true

```
1  @(posedge clk iff(valid));  
2  //do_something;
```

## ✓ Good for sampling

```
27  
28  // Example 1  
29  covergroup cg1 @(posedge clk iff(!reset));  
30  |    coverpoint var_1;  
31  endgroup  
32  
33  // Example 2  
34  covergroup cg2 @(posedge clk);  
35  |    coverpoint var_1 iff(!reset);  
36  endgroup
```

# How Is Coverage Information Gather

- ✓ **SystemVerilog automatically creates a number of bins for cover point.**
- ✓ **By default, NC-Verilog automatically creates default 64 bins.**
  - Values are equally distributed in each bin
    - 3-bit variable → 8 possible values → 8 auto bins will be created
    - 16-bit variable → 65536 possible values → each auto bin covers 1024 values
  - Option **auto\_bin\_max** specifies the maximum number of bins to create.
    - {option.auto\_bin\_max = your\_def }



# What is bins?

✓ **What is bins?** bins is a container for each value in the given range of possible values of a coverage point variable.

✓ **Without auto\_bin\_max:**

— Coverage is :

$$\frac{\text{\# of bins covered (have at\_least hits)}}{\text{\# of total bins}}$$

✓ **With auto\_bin\_max:**

(auto\_bin\_max limit the number of bins used in the coverage calculation)

— Coverage is :

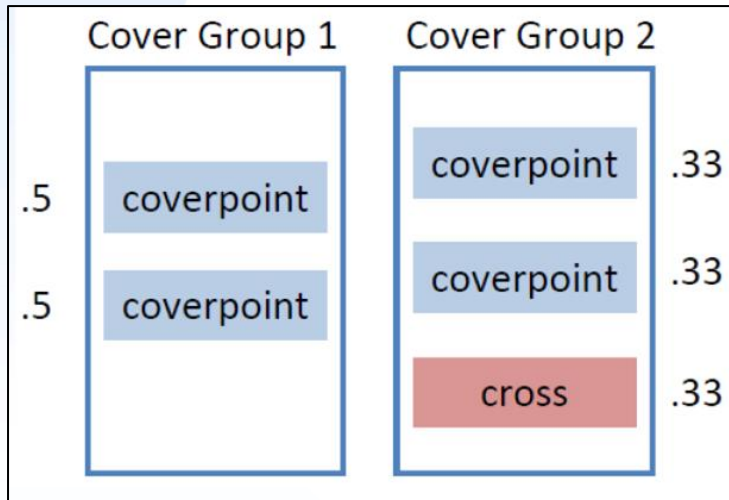
$$\frac{\text{\# of bins covered (have at\_least hits)}}{\min(\text{possible values for data type} \mid \text{auto\_bin\_max})}$$



# Coverage Measurement Example

- ✓ Each covergroup **contributes equally**
- ✓ Within covergroup, **each coverpoint/cross block contributes equally**
- ✓ Attributes change contributions

## Design



Cover Group 2  
coverpoint% x .33 +  
coverpoint% x .33 +  
cross % x .33 =  
group coverage %

Group 1 % x 0.5 +  
Group 2 % x 0.5 =  
Coverage Percent

# User-Defined Bin

- ✓ Define state bins using a **range**
- ✓ Define **transition** bins using state transitions

```
1  covergroup MyCov @(cov_event);
2  coverpoint port_number{
3
4      bins s0      = {[0: 7]};           //creates 1 state bin
5      bins s1 []   = {[8:15]};           //creates 8 state bins
6                                          //a bin array s1[8] ~ s1[15]
7      ignore_bins ignore = {16,20};      //ignore if hit
8      illegal_bins bad   = default;      //error message if hit
9
10     bins t0       = (0=>8, 9=>0);       //creates 1 transition bin
11     bins t1 []    = ([0:8]=>[8:15]);    //creates 72 transition bins 9*8=72
12     bins other_trans = default;         //all other transitions
13
14 }
15 endgroup
```

(0 => 8), (0 => 9), ... (8=>15)

# Transition Bins

```
CHECKER.sv 9+
C: > Users > anson > Desktop > CHECKER.sv
1  module Checker(~);
2  logic [9:0] stock_price_2330;
3  logic [10:0] stock_price_2454;
4  /*
5   |   2330 Highest: 688
6   |   2454 Highest: 1215 // over flow
7  */
8  covergroup ETF_0050 @(posedge wake_up_at_9_am);
9      Stock1 : coverpoint stock_price_2330
10     {
11         bins go_to_hell = (688 ==> 0);
12         bins go_to_paradise = (0 ==> 1023);
13     }
14     Stock2 : coverpoint stock_price_2454
15     {
16         bins go_to_hell = (1023 ==> 0);
17         bins go_to_paradise = (0 ==> 1023);
18     }
19 endgroup
20
21 // remember to do this
22 ETF_0050 ETF_my_dream = new();
23
24 endmodule
25
```



# Why ignore bins

- ✓ Exclude values that overlap with the explicit bins

```
coverpoint p {  
    bins exp[] = {[1:100]};  
    ignore_bins ign = {23,45,67};  
}
```

```
coverpoint p {  
    bins exp[] = {[1:22],[24:44],[46:66],[68:100]};  
}
```



# Cross Coverage Bin Creation (Automatic)

- ✓ NC-Verilog automatically creates **cross coverage bins**

```
39 logic [2:0] opa; // 0 - 7
40 logic [2:0] opb; // 0 - 7
41 covergroup cov1@(posedge clk);
42     coverpoint opa;
43     coverpoint opb;
44     cross opa, opb; // (a, b) = (0, 0), (0, 1), (0, 2), ...(7, 7)
45     // total = 8*8 = 64 => 6 bit
46 endgroup
```

- ✓ Cross bins for all combinations of the individual state

# Coverage Options

- ✓ **SystemVerilog defines a set of options.**  
Options control the behavior of the cover group, coverpoint, and cross.
- ✓ **Most of the options can be set procedurally after a cover group has been instantiated.**

Ref: <http://svref.renerta.com/sv00124.htm>



# Important Coverage Options

## ✓ **at\_least(1):**

- Minimum number of times for a bin to be hit to be considered covered

## ✓ **auto\_bin\_max(64):**

- Maximum number of bins that can be created automatically
- Each bin contains equal number of values

## ✓ **per\_instance(0):**

- Keeps track of coverage for each instance when it is set true



# Coverage Options Example

- ✓ The syntax of specifying options in the covergroup:  
`option.option_name = expression ;`

```
7  covergroup address_cov () @ (posedge ce) ;  
8      option.name          = "address_cov";  
9      option.comment       = "This is cool";  
10     option.per_instance  = 1;  
11     option.goal          = 100;  
12     ADDRESS : coverpoint addr {  
13         option.auto_bin_max = 100;  
14     }  
15     ADDRESS2 : coverpoint addr2 {  
16         option.auto_bin_max = 10;  
17     }  
18 endgroup
```

[option](#)



# Determining Coverage Progress

- ✓ `$get_coverage()` returns testbench coverage percentage as a *real* value

```
repeat (10) begin
    addr = $urandom_range(0,7);
    // Sample the covergroup
    my_cov.sample();
    #10 ;
end
// Stop the coverage collection
my_cov.stop();
// Display the coverage
$display("Instance coverage is %e", my_cov.get_coverage());
```

# Outline

## ✓ Section 1 Functional Coverage

- Coverpoint & Covergroup
- Specifying sample event timing
- Bin creation
- Options
- Coverage measurement

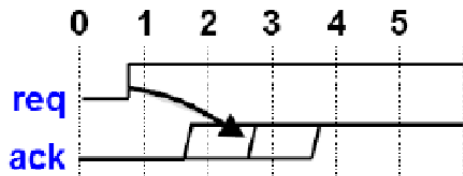
## ✓ Section 2 Assertion

- What is assertion
- Assertion types
- Sequence & Properties



# What is Assertion?

- ✓ An assertion is a design condition that you want to make sure never violates.
  - Assertion can be written in Verilog, but it's a lot of extra code



Each request must be followed by an acknowledge within 1 to 3 clock cycles



To test for a sequence of events requires several lines of Verilog code

- Difficult to write, read and maintain
- Cannot easily be turned off during reset or other don't care times

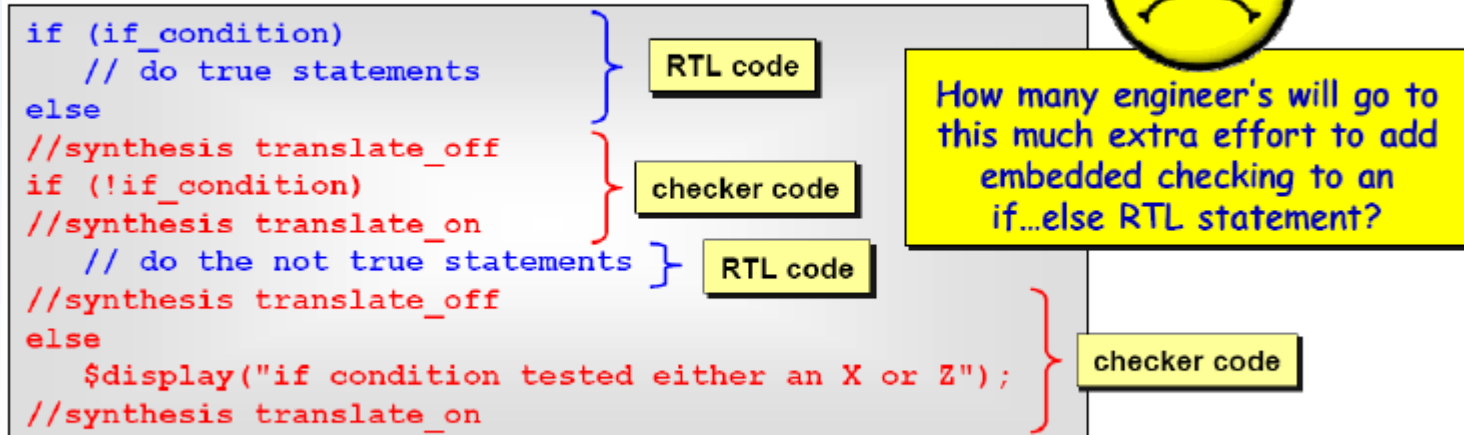
```
always @(posedge req) begin
    @(posedge clk) ; // synch to clock
    fork: watch_for_ack
        parameter N = 3;
        begin: cycle_counter
            repeat (N) @(posedge clk);
            $display("Assertion Failure", $time);
            disable check_ack;
        end // cycle_counter
        begin: check_ack
            @(posedge ack)
            $display("Assertion Success", $time);
            disable cycle_counter;
        end // check_ack
    join: watch_for_ack
end
```

4.000

# Verilog Assertion

## ✓ A checking function written in Verilog looks like RTL code

- Synthesis compiler can't distinguish the hardware model from the embedded checker code
- To hide the checker code from synthesis, need more extra effort





# SystemVerilog Assertions

## ✓ **SystemVerilog assertions have several advantages**

- Concise syntax
- Ignore by synthesis
- Can be disabled
- Can have severity level

## ✓ **Some SystemVerilog constructs have built-in assertions-like checking!**

- always\_comb / always\_ff
- Unique case / unique if ... else
- Enumerated variables
- By using this constructs, designer get the advantage of **self-checking** code without the need of assertions!



# Assertion Severity Levels

- `$fatal [ ( finish_number, "message", message_arguments ) ] ;`
  - Terminates execution of the tool
  - `finish_number` is `0`, `1` or `2`, and controls the information printed by the tool upon exit (the same levels as with `$finish`)
- `$error [ ( "message", message_arguments ) ] ;`
  - A run-time error severity; software continues execution
- `$warning [ ( "message", message_arguments ) ] ;`
  - A run-time warning; software continues execution
- `$info [ ( "message", message_arguments ) ] ;`
  - No severity; just print the message

Software tools may provide options to suppress errors or warnings or both

```
3 ReadCheck: assert (data == correct_data)
4           else $error("memory read error");
5 Igt10: assert (I > 10)
6           else $warning("I has exceeded 10");
```



# ## Cycle Delays

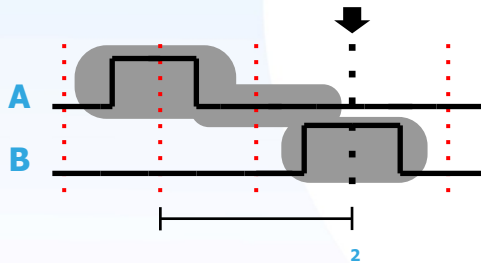
✓ ## represents a “cycle delay”

```
property p_request_grant;  
    @(posedge clock) request ##1 grant ##1 !request ##1 !grant;  
endproperty  
ap_request_grant : assert property (p_request_grant); else $fatal;
```

“(posedge clock)” is not a delay, it specifies what ## represents

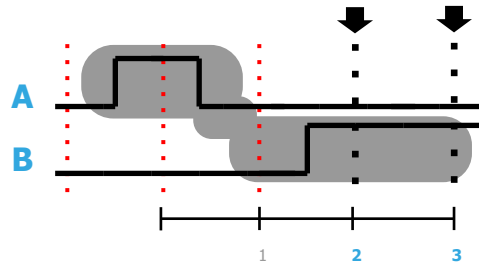
A ##2 B

"A happens then exactly 2 cycles later B happens"



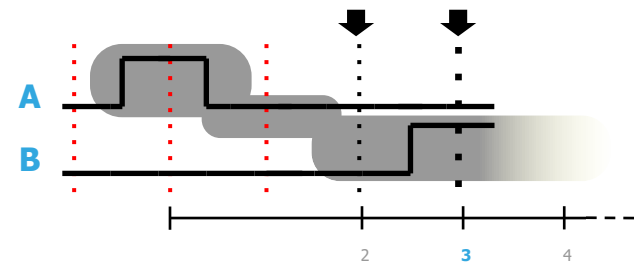
A ##[2:3] B

"A happens then 2 to 3 cycles later B happens"



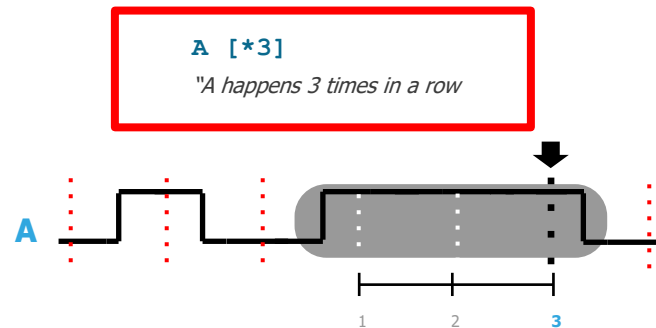
A ##[2:\$] B

"A happens then 2 or more cycles later B happens"



# Repetition operator

✓ Repetition operator **[\*N]**



# SystemVerilog Assertions

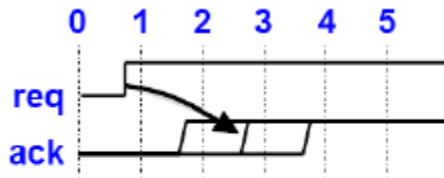
- ✓ SystemVerilog has two types of assertions.
- ✓ **Immediate assertions** test for a condition at the current time, combinational signals.

```
always @(state)
  assert ($onehot(state)) else $fatal;
```

generate a fatal error state  
variable is not a one-hot value

An immediate assertion is the same as an **if...else** statement, but with assertion controls

- ✓ **Concurrent assertions** test for a sequence of events over multiple clock cycles, sequential signals.



a complex sequence can be  
defined in very concise code

```
a_reqack: assert property (@(posedge clk) req ##[1:3] ack;) else $error;
```

One line of SVA code replaces all the Verilog code in the example three slides back!



# Immediate Assertions

- ✓ **A test of an expression when the moment the statement is executing**

[name:] assert (**expression**) [pass\_statement] [else fail\_statement]

- May be used in initial, always, tasks, and functions
- Performs a Boolean true/false test
- Evaluates the test at the instant the assert statement is executed

```
always @(negedge reset)
  a_fsm_reset: assert (state == LOAD)
    $display("FSM reset in %m passed");
  else
    $display("FSM reset in %m failed");
```

The name is optional:

- Creates a named hierarchy scope that can be displayed with %m
- Provides a way to turn off specific assertions

# Concurrent Assertions

## ✓ Test for a sequence of events spread over multiple clock cycles

[name:] assert **property** (**property\_spec**) [pass\_statement] else [fail\_statement]

- The property\_spec describes a sequence of events
- May be used in initial, always, or stand-alone

```
always @(posedge clock)
  if (State == FETCH)
    ap_req_gnt: assert property (p_req_gnt) passed_count++; else $fatal;
property p_req_gnt:
  @(posedge clock) request ##3 grant ##1 !request ##1 !grant;
endproperty: p_req_gnt
```

optional pass statement

optional fail statement

request must be true immediately, grant must be true 3 clocks cycles later, followed by request being false, and then grant being false

# Property Spec

- ✓ The argument to assert property() is a **property spec**
  - Contains the definition of a sequence of events

```
ap_Req2E: assert property (pReq2E) else $error;  
property pReq2E;  
  @(posedge clock) (request ##3 grant ##1 (qABC and qDE));  
endproperty: pReq2E
```

A property can reference and perform operations on named sequences

- A complex property can be built using sequence blocks

```
sequence qABC;  
  (a ##3 b ##1 c);  
endsequence: qABC
```

```
sequence qDE;  
  (d ##[1:4] e);  
endsequence: qDE
```

- A simple sequence can also be specify in assert

```
always @(posedge clock)  
  if (State == FETCH)  
    assert property (request ##1 grant) else $error;
```

The clock cycle can be inferred from where the assertion is called



# Implication

## ✓ Overlapped $\rightarrow$

- $S1 \rightarrow S2$ , If the sequence S1 matches, then sequence S2 must **also** matches at the same cycle

## ✓ Non-overlapped $\Rightarrow$

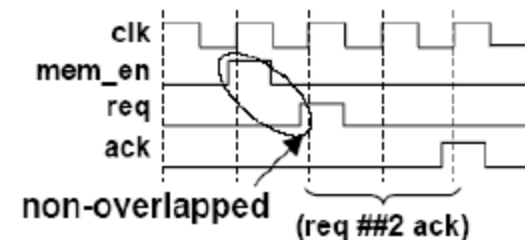
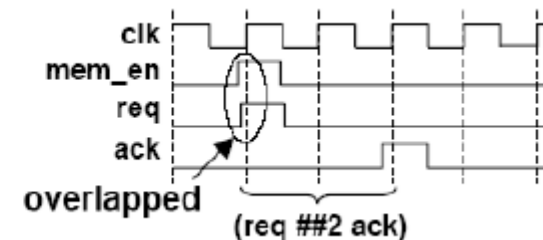
- $S1 \Rightarrow S2$ , If the sequence S1 matches, then **at the next cycle**, sequence S2 must also matches

## ✓ Preconditioned with an implication operator

- If the condition is true, sequence evaluation starts immediately ( $\rightarrow$ ) or next cycle ( $\Rightarrow$ ), otherwise it acts as if it succeeded

```
property p_req_ack;  
  @(posedge clk) mem_en  $\rightarrow$  (req ##2 ack);  
endproperty: p_req_ack
```

```
property p_req_ack;  
  @(posedge clk) mem_en  $\Rightarrow$  (req ##2 ack);  
endproperty: p_req_ack
```



# Combining Sequences

## ✓ and

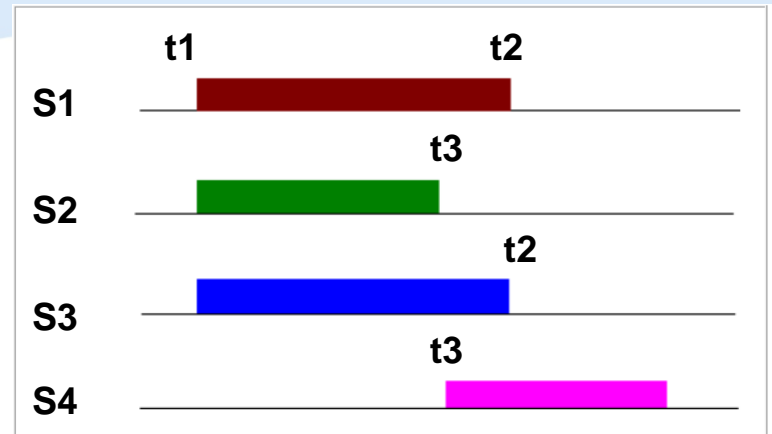
- s1 and s2, succeeds if s1 and s2 succeed. The end time is the end of the sequence that terminates last

## ✓ intersect

- s1 intersect s3, succeeds if s1 and s3 succeed and if end time of s1 is the same with the end of s3

## ✓ Or

- s1 or s4, succeeds whenever at least one of two operands s1 and s4 is evaluated to true

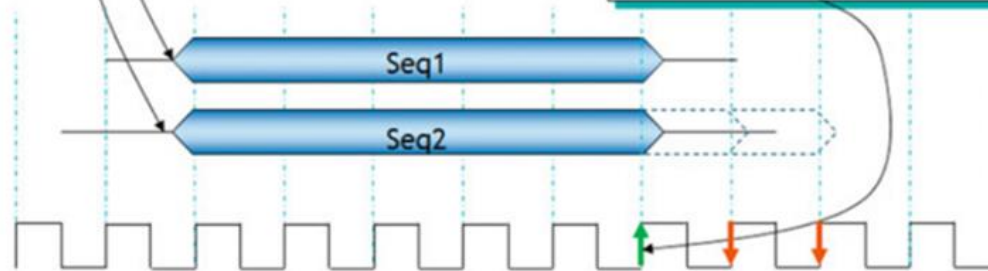


# Intersect vs And

**'intersect'**

Seq1 and Seq2 must START at the same time.

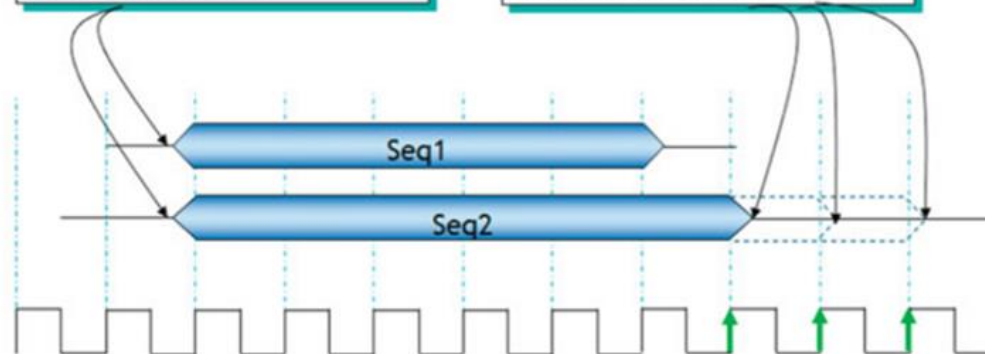
A match occurs only when both sequences end at the same time. If Seq2 ends at a different time, there won't be a match



**'and'**

Seq1 and Seq2 must start at the same time.

A match occurs whenever the longer sequence ends.



# Assertion System Functions

## ✓ \$rose

- asserts that if the variable changes from 0 to 1 between one posedge clock and the next, detect must be 1 on the following clock.

```
assert property  
    (@(posedge clk) $rose(in) ==> detect);
```

## ✓ \$fell

- asserts that if the variable changes from 1 to 0 between one posedge clock and the next, detect must be 1 on the following clock

```
assert property  
    (@(posedge clk) $fell(in) ==> detect);
```



# Assertion System Functions

## ✓ \$stable

- states that data shouldn't change while enable is 0.

```
assert property  
  (@(posedge clk) enable == 0 | => $stable(data));
```

## ✓ \$past

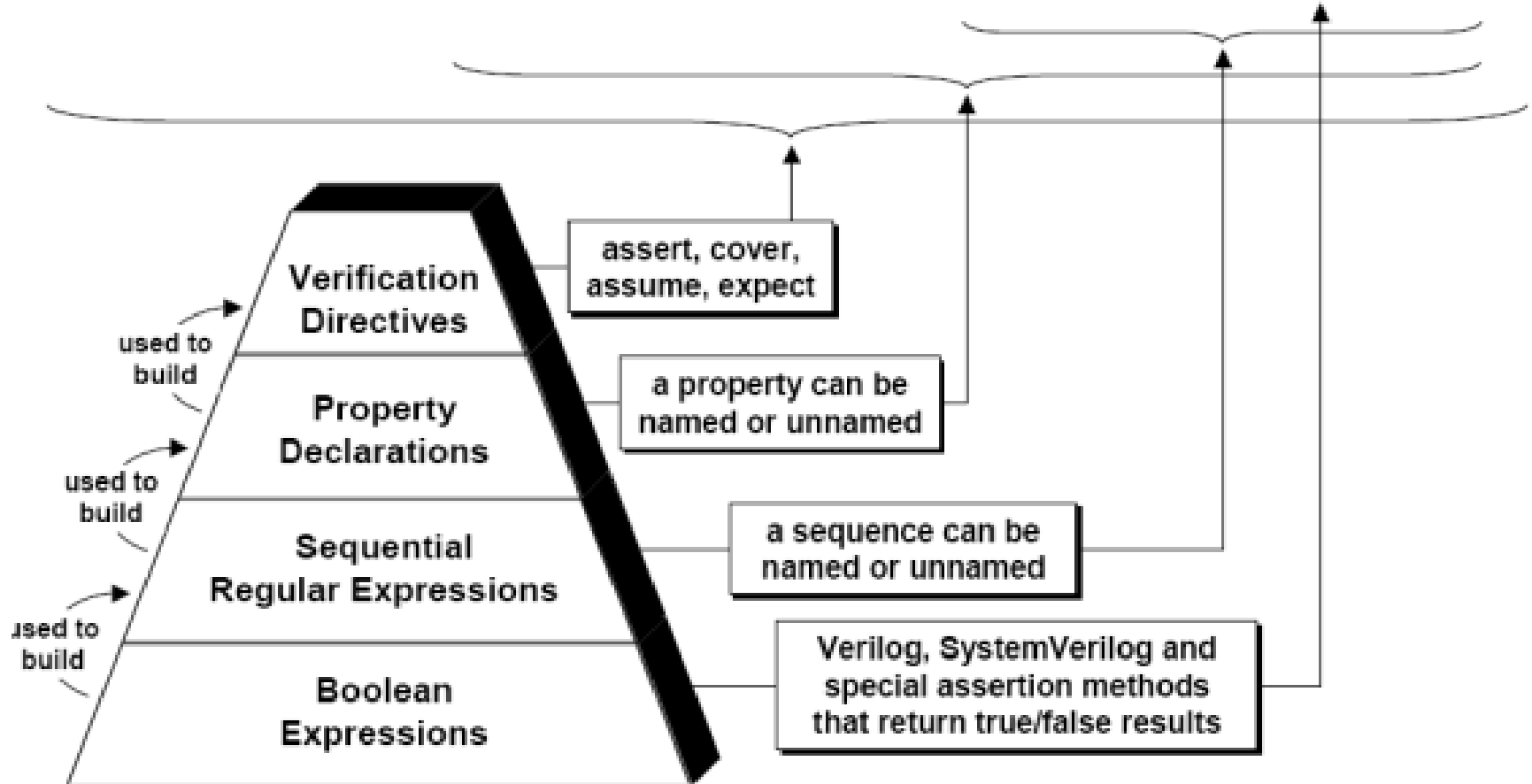
- provides the value of the signal from the previous clock cycle

```
$past(signal_name, number of clock cycles)
```

```
property p;  
  @(posedge clk) b | -> ($past(a,2) == 1);  
endproperty
```

# Assertion Building Blocks

```
assert property (@posedge clk) req |-> gnt ##1 (done && !err);
```



# Appendix A - Cover point Expression

## ✓ Using XMR (cross module reference)

- Cover\_xmr : **coverpoint** top.DUT.Submodule.bus\_address;

## ✓ Part select

- Cover\_part: **coverpoint** bus\_address[31:2];

## ✓ Expression

- Cocver\_exp: **coverpoint** (a\*b);

## ✓ Function return value

- Cover\_fun: **coverpoint** funcation\_call();

[http://www.testbench.in/CO\\_05\\_COVERPOINT\\_EXPRESSIO](http://www.testbench.in/CO_05_COVERPOINT_EXPRESSIO)  
N.html



# Automatic State Bin Creation Example

## ✓ Bin name is “auto[value\_range]”

- The value\_range are the value which triggered that bin

```
37 program automatic test(busifc.TB ifc);
38   class Transaction;
39       rand bit [31:0] data;
40       rand bit [ 2:0] port;
41   endclass
42
43   covergroup CovPort;
44       coverpoint tr.port;
45   endgroup
46
47   initial begin
48       Transaction tr;
49       CovPort ck;
50       tr = new();
51       ck = new();
52       repeat(32)begin
53           @ifc.cb;
54           assert(tr.randomize());
55           ifc.cb.port <= tr.port;
56           ifc.cb.data <= tr.data;
57           ck.sample();
58       end
59   end
60 endprogram
```

1

2

```
// Transaction to be sampled~
// Instantiate group

// wait a cycle
// Create a Transaction
// Transmit onto interface

// Gather coverage
```

Coverpoint Coverage report

CoverageGroup: CovPort

Coverpoint: tr.port

Summary

Coverage: 87.50

Goal: 100

Number of Expected auto-bins: 8

Number of User Defined Bins: 0

Number of Automatically Generated Bins: 7

Number of User Defined Transitions: 0

Automatically Generated Bins

Bin	# hits	at least
auto[1]	7	1
auto[2]	7	1
auto[3]	1	1
auto[4]	5	1
auto[5]	4	1
auto[6]	2	1
auto[7]	6	1

where is auto[0] ?





# Reference

## ✓ Website:

- <http://www.testbench.in/>
- <http://www.asic-world.com/systemverilog/tutorial.html>
- <http://www.doulos.com/knowhow/sysverilog/tutorial/assertions/>
- [Coverage](#)
- [Option](#)

## ✓ Textbook:

- “SystemVerilog for Verification: A Guide to Learning the Testbench Language Features” 3rd ed. 2012 Edition, by Chris Spear (e-book is available in NCTU library.)

