SystemVerilog Verification

NCTU-EE IC LAB SPRING 2022

Lecturer: Chih-Wei Peng

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

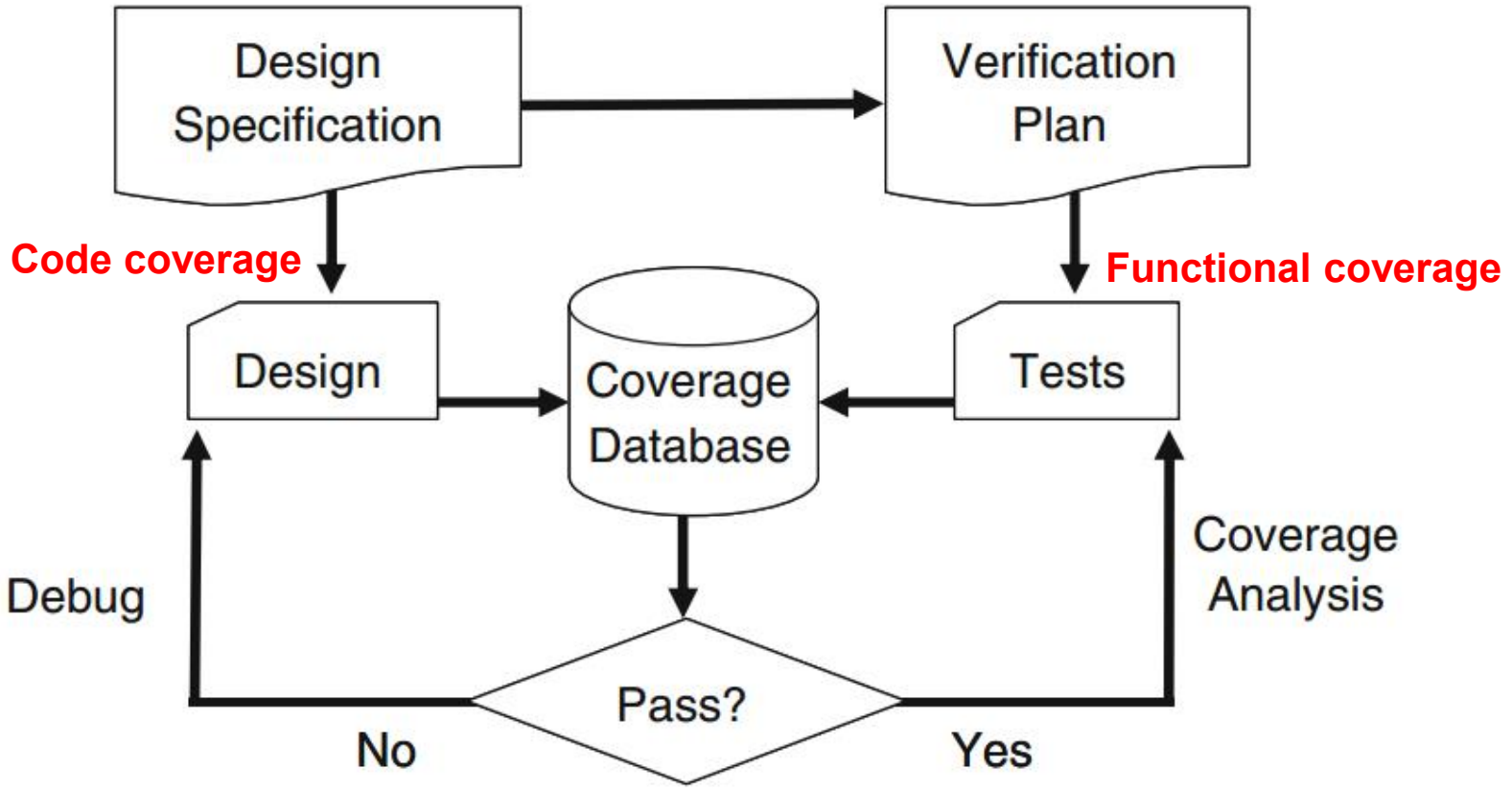
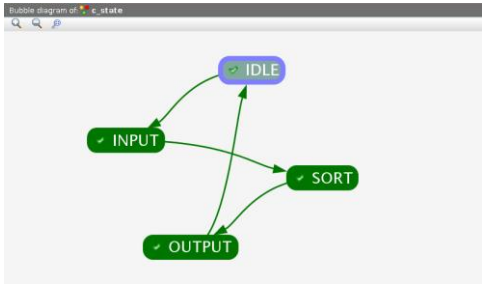


Fig. 9.2 Coverage flow

Code Coverage

- ✓ **Statement (line) coverage**
  - Lines of source code which have been tested
- ✓ **Block coverage**
  - Blocks of source code which have been tested



- ✓ **Conditional/Expression coverage**
  - Boolean sub-expressions have been tested for a true or a false value.
- ✓ **Branch/Decision coverage**
  - Branches of the control structures have been executed.
- ✓ **Toggle coverage**
  - Design activity in terms of changes in signal values.
- ✓ **FSM coverage**
  - States have been visited.

```
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
```

http://vlsi.pro/code-coverage-fundamentals/

Functional Coverage in SystemVerilog

- ✓ 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 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
```

要包的答案

Specifying Sample Event Timing

- ✓ **Define sample_event in coverage_group**
- ✓ **Valid sample_event_definition:**
  - @([specified_edge] signals | variables)

posedge or negedge

- ✓ **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
```

要抓的資訊　特定的 signal

IFF

- **Event control**
  - Only be triggered when the expression after iff is true

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

if and only if

- **Good for sampling**

```
1   //Example 1
2   covergroup cg1@(posedge clk);
3       coverpoint cp_varib iff(!reset);
4   endgroup
5
6   //Example 2
7   covergroup cg2@(posedge clk iff(!reset));
8       coverpoint cp_varib;
9   endgroup
```

(posedge clk ∧ reset == 0)

How Is Coverage Information Gather

- ✓ **SystemVerilog automatically creates a number of bins for cover point.**
- ✓ **By default, NC-Verilog automatically creates at most 64 bins.**
  - Values are equally distributed in each bin
    - 3-bit variable → 8 possible values → 8 autobins will be created
    - 16-bit variable → 65536 possible values → each bins covers 1024 values
  - Option auto_bin_max specifies the maximum number of bins to create.
    - {option.auto_bin_max = yourdef }
    - 16-bit variable → each bins covers 2048 values

2^3

2^{16}

$\frac{65536}{64}$

ex: 32

$\frac{65536}{32} = 2048$

共有65536個 value

用64 bin 去 cover

一個bin有1024個 value

What is bin?

- **What is bin?** bin is a container for each value in the given range of possible values of a coverage point variable.
- **Without auto binning:**
  - Coverage is :

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

- **With auto binning:** $2^5 = 32$ 5 bit reg ⇒ 32 bins

(auto_bin_max limit the number of bins used in the coverage calculation)

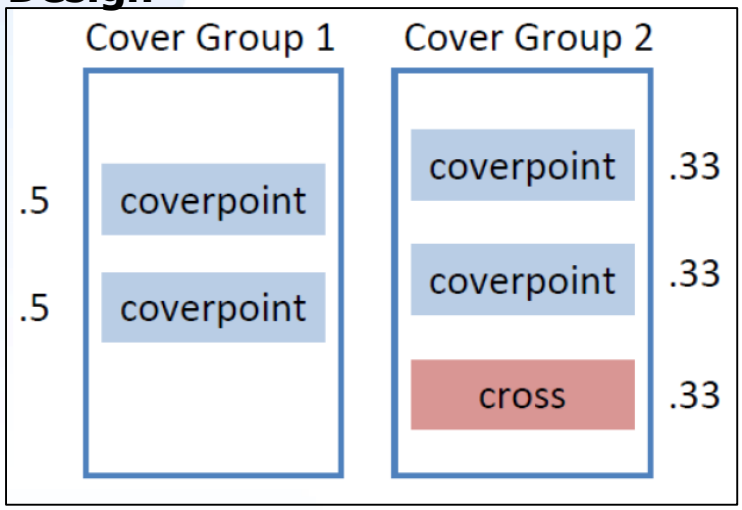
  - Coverage is :

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

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 transitions 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
12         bins other_trans = default;       //all other transitions
13
14     }
15 endgroup
```

0=>8 , 1=>8 , 2=>8 ... 7=>8

=> 共 9×8=72

;

0=>15 ... 7=>15

Cross Coverage Bin Creation (Automatic)

- ✓ SystemVerilog can automatically creates **cross coverage bins**

```
3  covergroup cov1@(posedge clk)
4      coverpoint opa;
5      coverpoint opb;
6      cross opa, opb;
7  endgroup
```

create all possible state

- ✓ 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

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

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
```

http://www.asic-world.com/systemverilog/coverage21.html#Coverage_Options

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

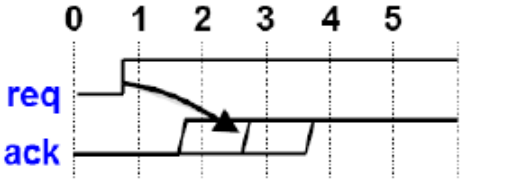
- ✓ Section 1 Functional Coverage
  - Coverpoint & Covergroup
  - Specifying sample event timing
  - Bin creation
  - Options
  - Coverage measurement
- ✓ **Section 2 Assertion**
  - What is assertion
  - Assertion types
  - Sequence & Properties

|→ 对同一个 cycle 检查

|=> 对下一个 cycle 检查

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

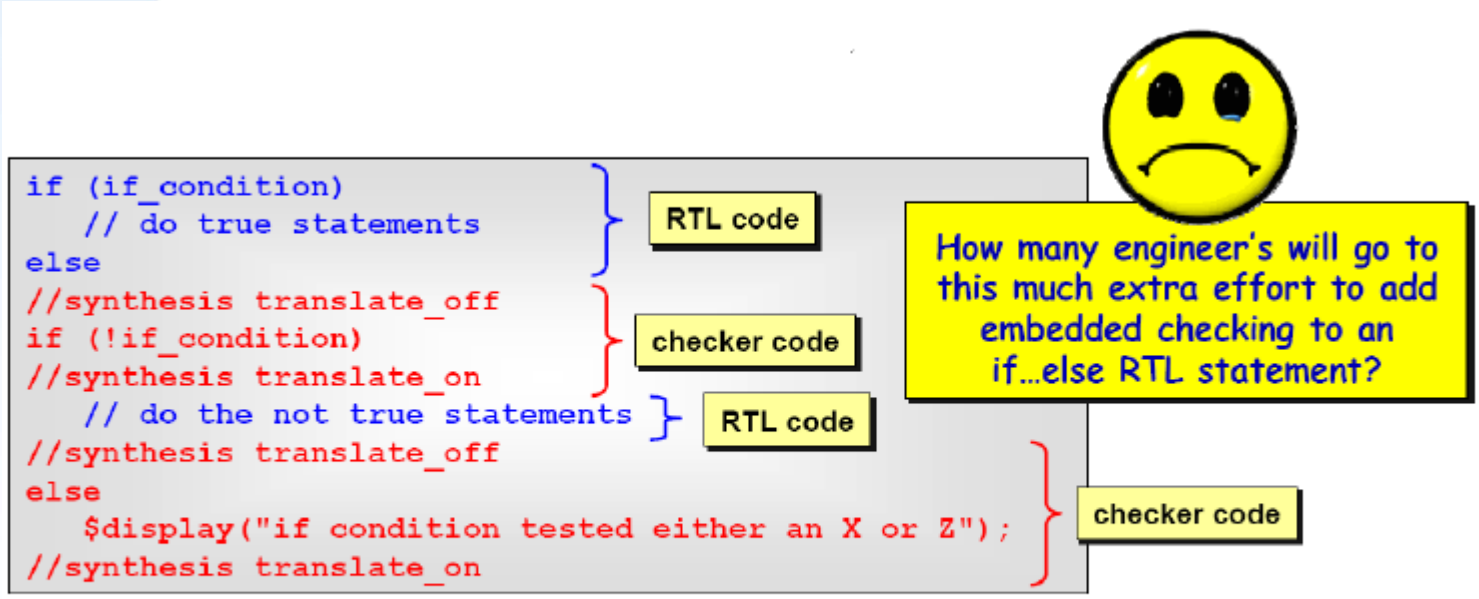
To test for a sequence of events requires several lines of Verilog code

- 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
```

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");
```

SystemVerilog Assertions

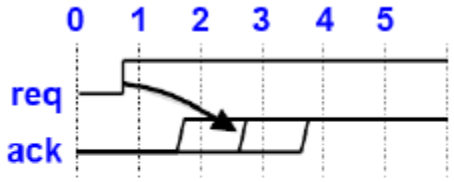
- ✓ SystemVerilog has two types of assertions.
- ✓ **Immediate assertions** test for a condition at the current time

```
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



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

1 cycle

1 cycle

how many cycle

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

Properties and Sequence

- ✓ **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

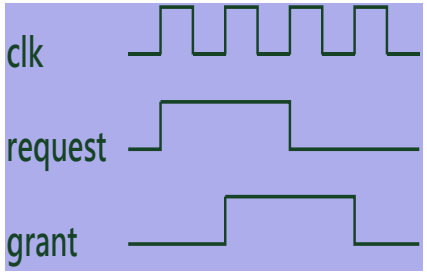
Cycle Delay

- ## represents a "cycle delay"
  - Specifies the number of clock cycles to wait until the next expression in the sequence is evaluated

```
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

- request must be followed one clock cycle later by grant
- grant must followed one clock cycle later by !request
- !request must be followed one clock cycle later by !grant



Multi-Cycle or Range Delay

- ✓ **##n specifies a fixed number of clock cycles**
  - n must be a non-negative constant expression

```
request ##3 grant;
```

After evaluating request, skip 2 clocks and then evaluate grant on the 3rd clock

- ✓ **## [min_count:max_count] specifies a range of clock cycles, also they must be non-negative constants**

```
request ##[1:3] grant;
```

After evaluating request, grant must be true between 1 and 3 clocks later

// same

This sequence would evaluate as true for:

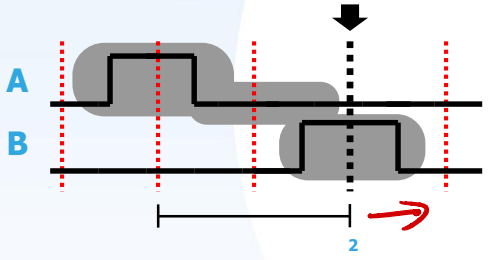
```
   (request ##1 grant);
or (request ##2 grant);
or (request ##3 grant);
```

Sequences

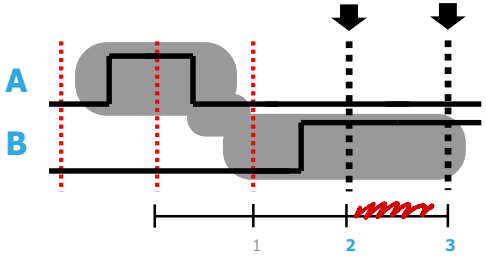
- Sometimes necessary to capture sequence of events in properties
- Most sequences are described using ## operator

all the time after 2 cycle

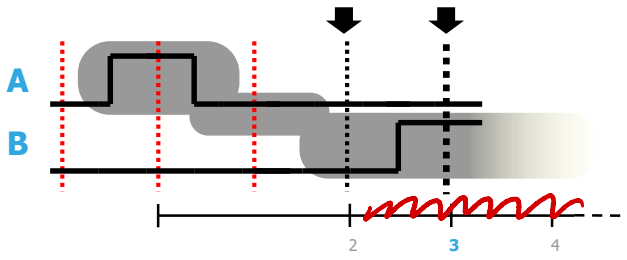
`A ##2 B`
"A happens then exactly 2 cycles later B happens"



`A ##[2:3] B`
"A happens then 2 to 3 cycles later B happens"



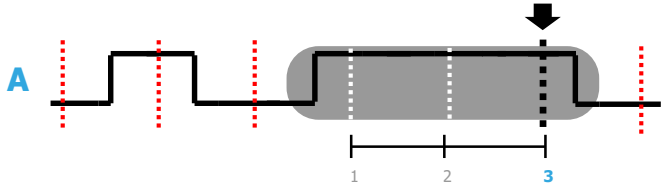
`A ##[2:$] B`
"A happens then 2 or more cycles later B happens"



Sequences

- **Repetition operator `[*N]` is also sometimes useful:**

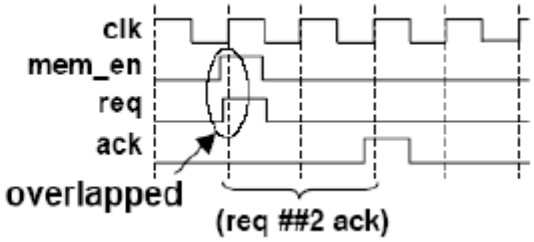
`A [*3]`
"A happens 3 times in a row



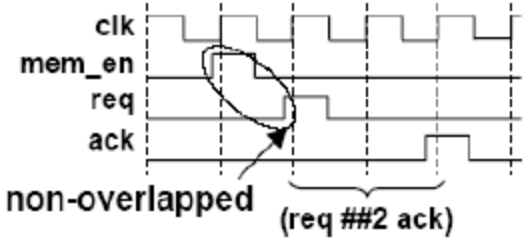
Implication

- **Overlapped |->**
  - S1 | -> S2, If the sequence S1 matches, then sequence S2 must also matches at the same cycle
- **Non-overlapped |=>**
  - S1 | => 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 (|->) or next cycle (|=>), otherwise it acts as if it succeeded

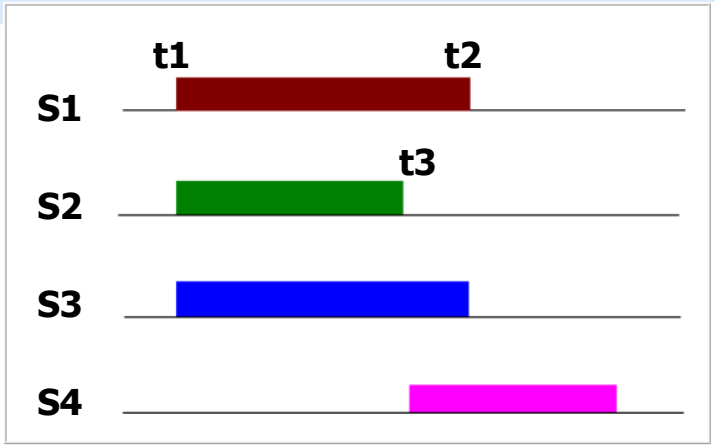
```
property p_req_ack;
 @(posedge clk) mem_en |-> (req ##2 ack);
endproperty: p_req_ack
```




```
property p_req_ack;
 @(posedge clk) mem_en |=> (req ##2 ack);
endproperty: p_req_ack
```

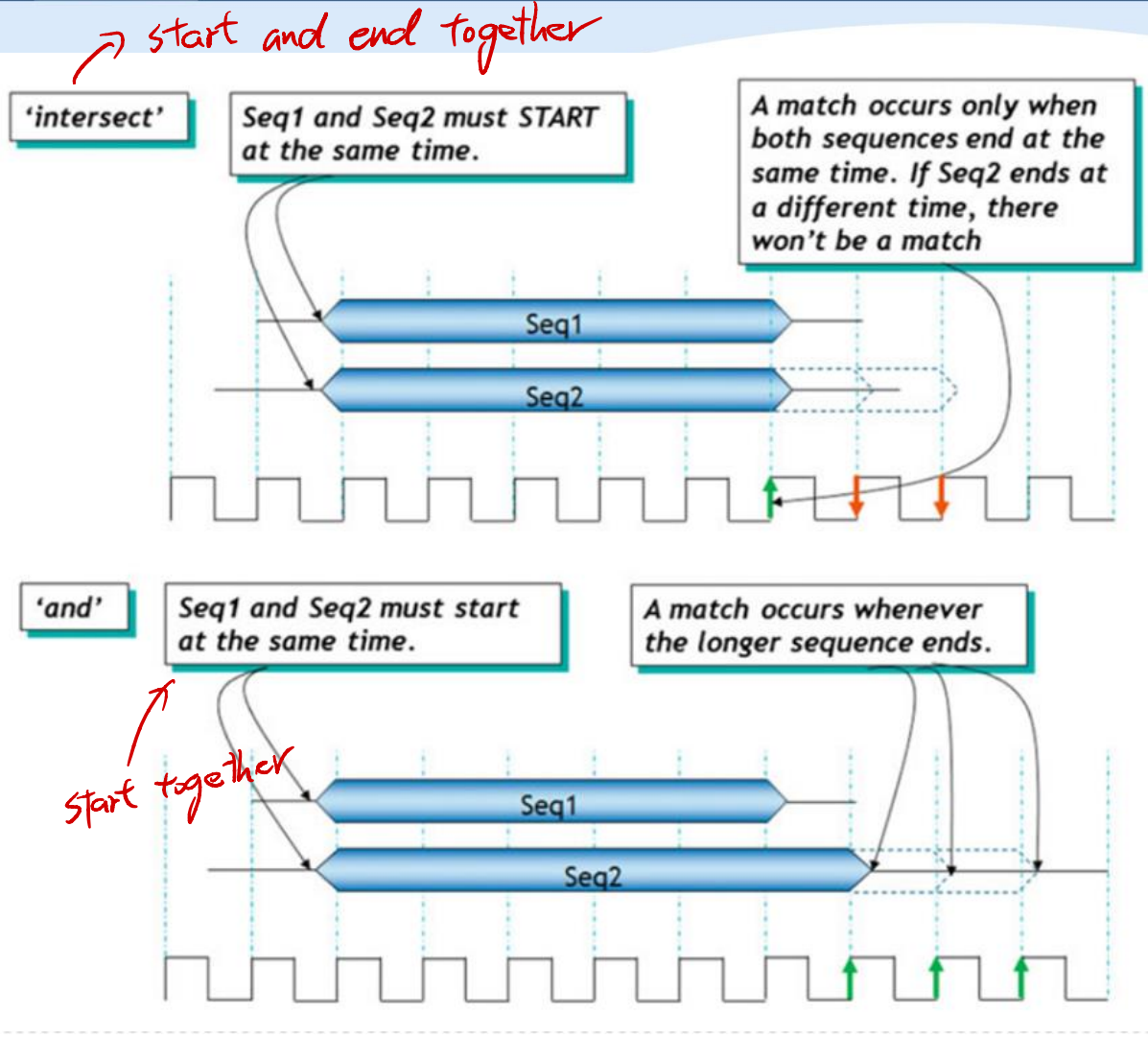


Combining Sequences



- **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

And vs Intersect



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));
```

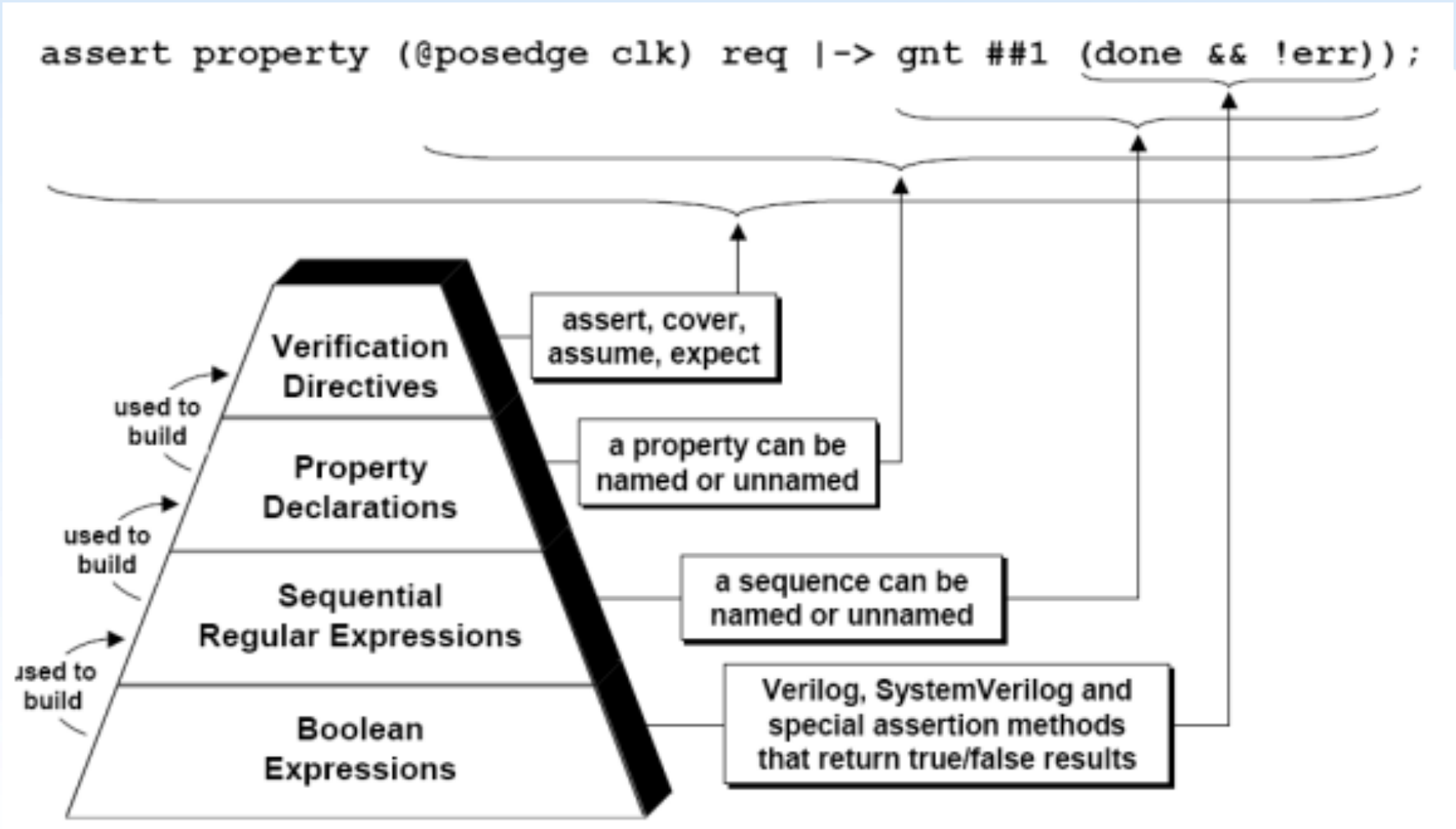
- ✓ $past
  - provides the value of the signal from the previous clock cycle

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

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

a 在 b 變 1 之前要先 1 2 cycle

Assertion Building Blocks



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();
- ✓ **Ref variable**
  - covergroup (ref int r_v) cg;
    cover_ref: coverpoint r_v;
    endgroup

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();                  // Transaction to be sampled~
51         ck = new();                  // Instantiate group
52         repeat(32)begin
53             @ifc.cb;                 // wait a cycle
54             assert(tr.randomize());  // Create a Transaction
55             ifc.cb.port <= tr.port;  // Transmit onto interface
56             ifc.cb.data <= tr.data;
57             ck.sample();             // Gather coverage
58         end
59     end
60 endprogram
```

```
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
    ================================
```

Reference

- **Website:**
  - http://www.testbench.in/
  - http://www.asic-world.com/systemverilog/tutorial.html
  - http://www.doulos.com/knowhow/sysverilog/tutorial/assertions/
- **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.)