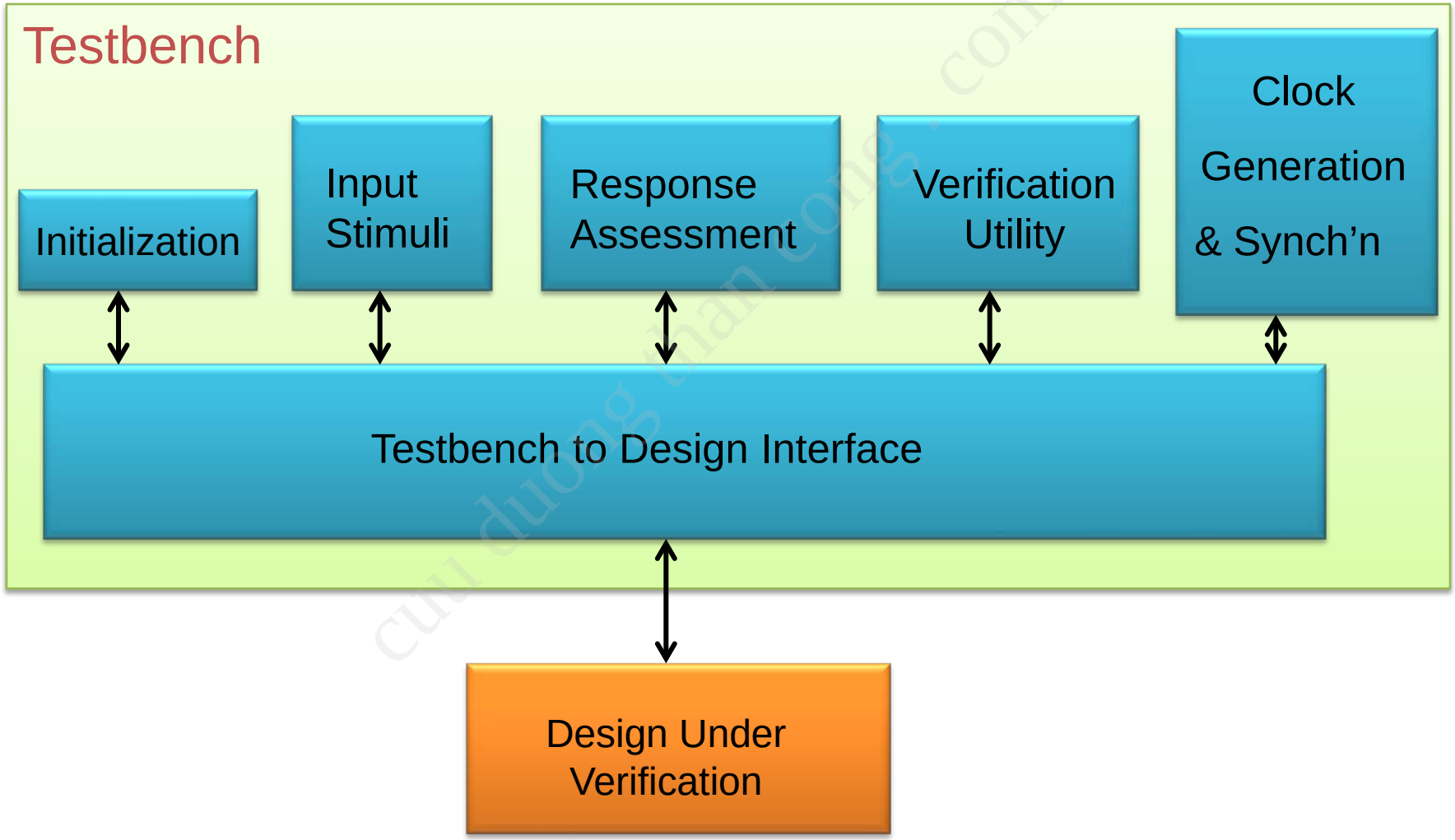
ĐẠI HỌC QUỐC GIA TP.HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA ĐIỆN-ĐIỆN TỬ
BỘ MÔN KỸ THUẬT ĐIỆN TỬ

ASIC CHIP AND IP CORE DESIGN

Chapter 4: Verification Process (Part 1)

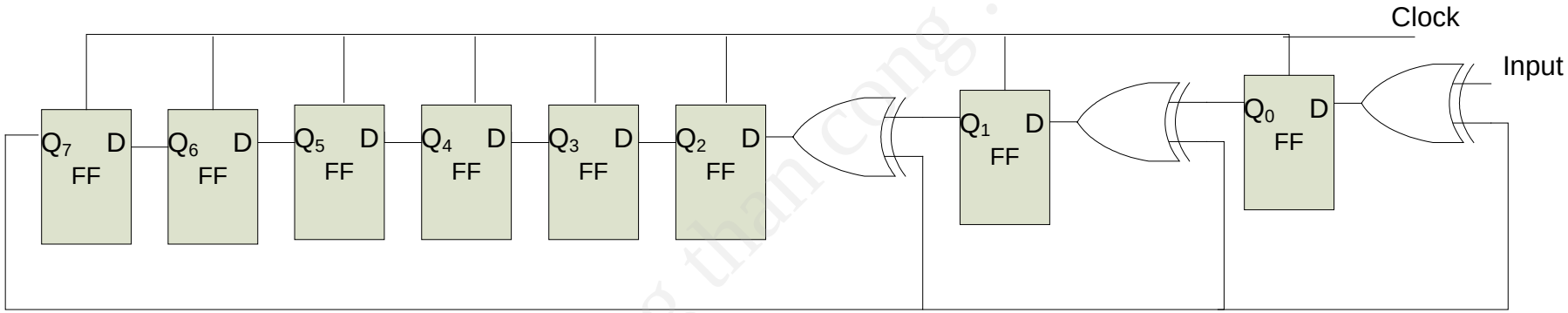
4.1 Test environment
4.2 Testbench design
4.3 Formal verification
4.4 Boundary-scan test

4.1 Test Environment



Test Environment: Examples

- Compute Remainder of CRC-8
  - Input vector % (10000111)



➢ Instantialtion:

```
CRC8 DUV (.IN(in), .CLK(clk), .Q0(q0), … , .Q7(q7));
```

Test Environment: Examples

- Description of design under verification and input stimuli:
  - Need to apply a bit stream:
  - Store bits in an array and apply the array.

```
initial i = size_of_input;

always @(posedge clk) begin
   if (i != 0) begin
      in <= input_array[i];
      i <= i - 1;
   end
end
```

➢ Response assessment:

```
remainder = input_array % 8'b10000111;
if (remainder != {q7, q6, q5, q4, q3, q2, q1,
q0})
    print_error();
```

Test Environment: Examples

– FF initialization:

```
initial begin
    DUV.Q0 = 1'b0;
    DUV.Q1 = 1'b0;
    …
    DUV.Q7 = 1'b0;
```

➢ Clock generation:

```
always clk = #1 ~clk;
```

Utilities for Test Environment

- Software tools
  - Matlab
  - ModelSim
  - VCS
- Languages
  - VHDL, Verilog, SystemC
  - Vera, Specman
  - OVM, VMM
  - C/C++

4.2 Testbench Design

- Design test cases:
  - Initialization
  - Input stimulus generation
  - Responses assessment
- Approach
  a) Initialize all inputs
  b) Generate the clock signals
  c) Send test vectors
  d) Assess the response signals

Test case: Example

- Example:
  - TC1:
    - Verify integers add/subtract
    - Input vectors chosen to cause corner cases (e.g. overflow)
  - TC2:
    - Verify Boolean operations:
    - Input vectors: certain bit patterns (e.g. 101010101, 11111111)
- Reusability:
  - Use the same testbench for multiple test cases
    - → To maximize portability, TCs must be separated from testbench,
    - e.g. read initial values from a file (that contains a TC).

a. Initialization

- Initialization:
  - Assign values to state elements (FFs, memories)
  - Although the task of circuitry (at power-on), often done in testbench.
  - Reasons:
    - Initialization circuit has not designed at the time.
    - Simulation is to run starting from a long time after power-on,
      - (e.g. simulating through initialization stage takes too long)
    - Simulation emulates an exception condition (normal operation never reaches it from its legal initial states).
  - To gain reusability, initialization code should be encapsulated inside a procedure.

a. Initialization

- Initialization @ time zero:
  - Some simulators create a transition (event) from unknown X (uninitialized) to initial value and some others don't.
    → Inconsistent results from simulator to simulator.
  - → Initialize at a positive time.
  - Even safer:
    - Initialize to X (or 'U') at time zero.
    - Then initialize to init value at a positive time.

```
task init_later;
input [N:0] value;
begin
    design.usb.xmit.Q = 1'bx;
    …
    #1;
    design.usb.xmit.Q = value[0];
    …
end
end task
```

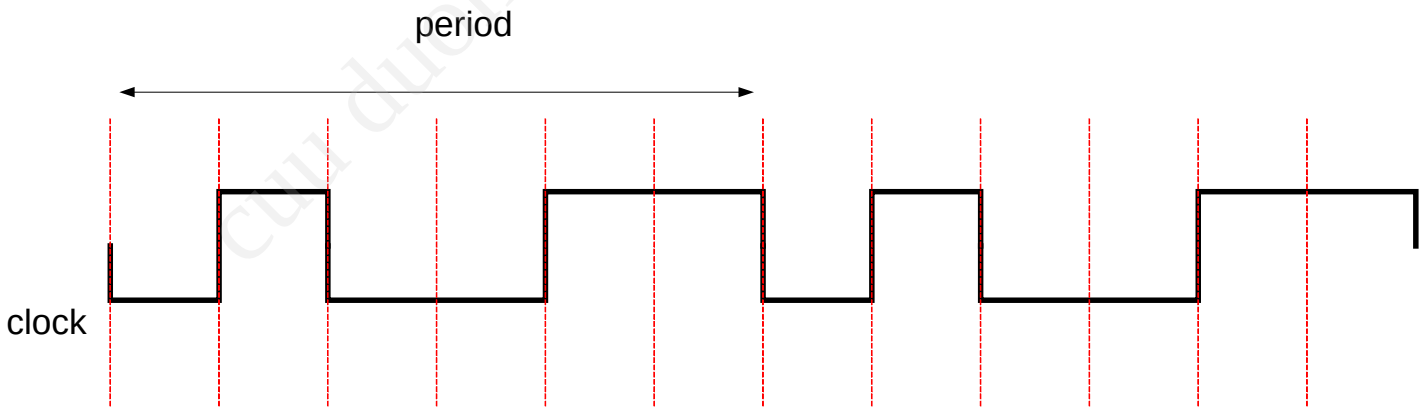
b. Clock Generation

- Explicit Method:

```
initial clock = 1'b0;
always begin
    #1 clock = 1'b1;
    #1 clock = 1'b0;
    #2 clock = 1'b1;
    #2 clock = 1'b0;
end
```

- Toggle Method:

```
initial clock = 1'b0;
always begin
    #1 clock = ~clock; // rising
    #1 clock = ~clock; // falling
    #2 clock = ~clock; // rising
    #2 clock = ~clock; // falling
end
```



b. Clock Generation

- Toggle Method:
  - Difficult to see the value of clock at a given time
    - Comments: falling/rising.
  - If left uninitialized, doesn't toggle (starts at x)
    - A potential bug.
  - Easy to change the phase or initial value.
    - Other statements kept intact.
- Time Unit and Resolution:
  - During verification, clock period/duty cycle may change
    - → Use parameters (rather than hard coding)

b. Clock Generation

- Multiple Clock Systems with a Base Clock:
  - Clock divider:

```
initial i = 0;
always @(base_clock)
begin
    i = i % N;
    if (i = 0) derived_clock = ~derived_clock;
    i = i +1;
end
```

  - Clock multiplier:
    - Note: Synchronized with the base clock.

```
always @(posedge base_clock)
begin
    repeat (2N) clock = #(period)/(2N)) ~clock;
end
```

b. Clock Generation

- Multiple Clock Systems with a Base Clock:
  - If the period of the base clock not known, Measure it:

```
initial begin
derived_clock = 1'b0; //assume starting 0
@(posedge base_clock) T1 = $realtime;
@(posedge base_clock) T2 = $realtime;
period = T2 – T1;
T1 = T2;
->start; // start generating derived_clock
end

// continuously measure base block's period
always @(start)
forever
@(posedge base_clock) begin
T2 = $realtime;
period = T2 – T1;
T1 = T2;
end

//generate derived_clock N times the freq of base_clock
always @(start)
forever derived_clock = #(period/(2N)) ~derived_clock;
```

b. Clock Generation

- If two periods are independent, don't generate one from the other.

```
initial clock1 = 1'b0;
always clock1 = #1
~clock1;

initial clock2 = 1'b0;
always clock2 = #2 ~clock;
```

Right

```
initial clock1 = 1'b0;
always clock1 = #1 ~clock1;

initial clock2 = 1'b0;
always @(negedge clock1) clock2 = #2
~clock;
```

Wrong

b. Clock Generation

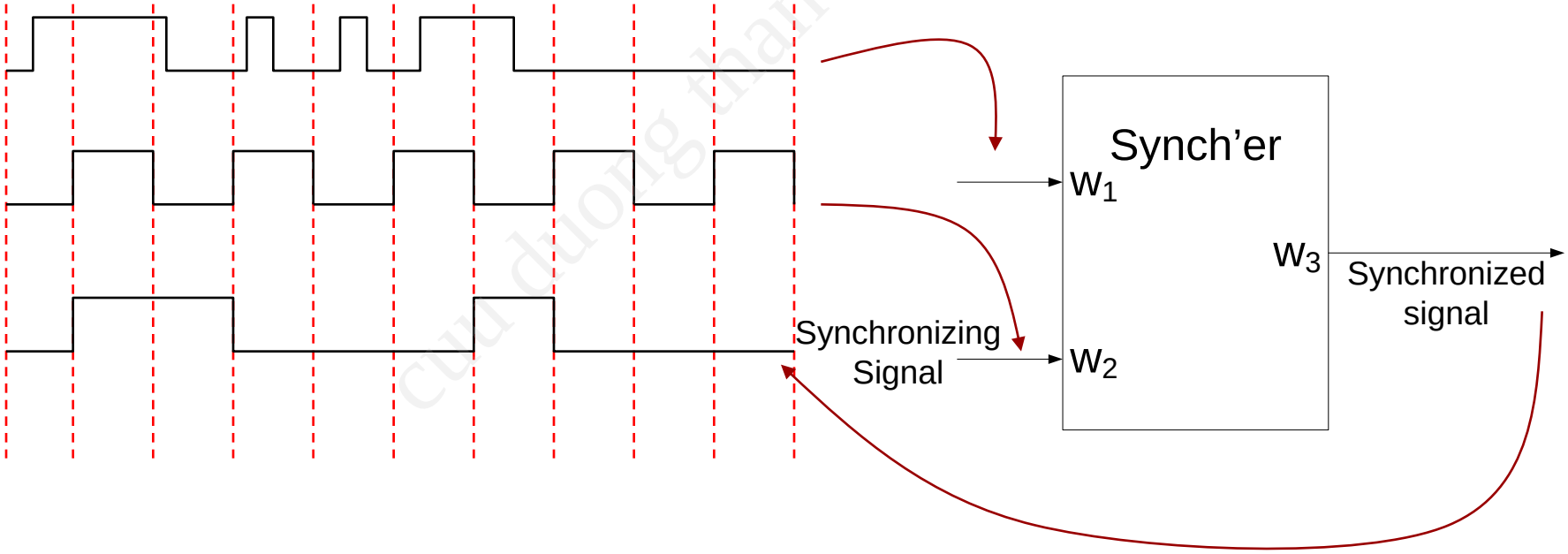
- Any simulator shows the same waveform but they are different:
  - Adding jitter to clock1 must not affect clock2

```
initial clock1 = 1'b0;
always clock1 = #1 ~clock1;

jitter = $random(seed) % RANGE;
assign clock1_jittered = (jitter) clock1;
```

b. Clock Synchronization

- When two independent waveforms arrive at the same gate, glitches may be produced:
  - → intermittent behavior.
- Independent waveforms should be synchronized before propagation



b. Clock Synchronization

- Synchronizer:
  - A latch.
  - Uses a signal (synchronizing) to trigger sampling of another to create a dependency between them.
    - → Removes uncertainty in their relative phase.

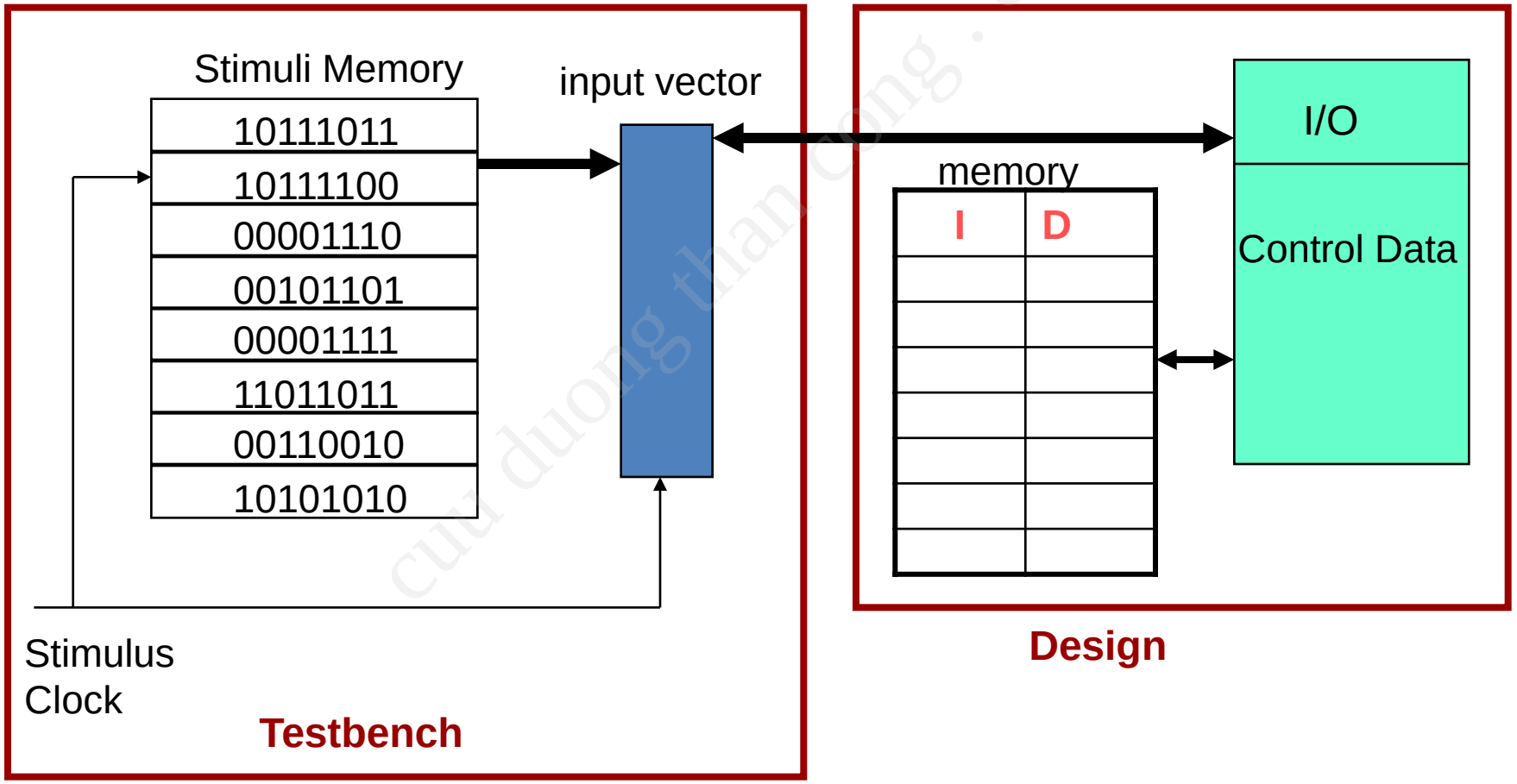
```
always @(fast_clock)
   clock_synchronized <= clock;
```

➢ Some transitions may be missed.

- → The signal with highest frequency is chosen as the synch'ing signal.

b. Clock Synchronization

- Synchronous Method:
  - Applying vectors to primary inputs synchronously.



c. Stimulus Generation

- Vectors stored in stimuli memory (read stimuli from file)
- triggered by a stimulus clock, memory is read one vector at a time.
- Stimulus clock must be synchronized with design clock.
- Encapsulate the code for applying data in a task (procedure)
  - → Stimulus application is separated from particular memory or design

- Asynchronous Method:
  - Sometimes inputs are to be applied synchronously.
    - e.g. Handshaking.

d. Response Assessment

- Two parts of response assessment:
  1. Monitoring the design nodes during simulation
  2. Comparing the node values with expected values.

- Absence of discrepancy ☺ could have many meanings:
  - Erroneous nodes were not monitored,☹
  - Bugs were not exercised by the input stimuli, ☹
  - There are bugs in the “expected” values that masks the real problem, ☹
  - The design is indeed free of bugs ☺ ☺

d. Response Assessment

- Comparison Methods:
  1. Offline (post processing)
     - Node values are dumped out to a file during simulation
     - Then the file is processed after the simulation is finished.
  2. On the fly
     - Node values are gathered during simulation and are compared with expected values.

d. Response Assessment

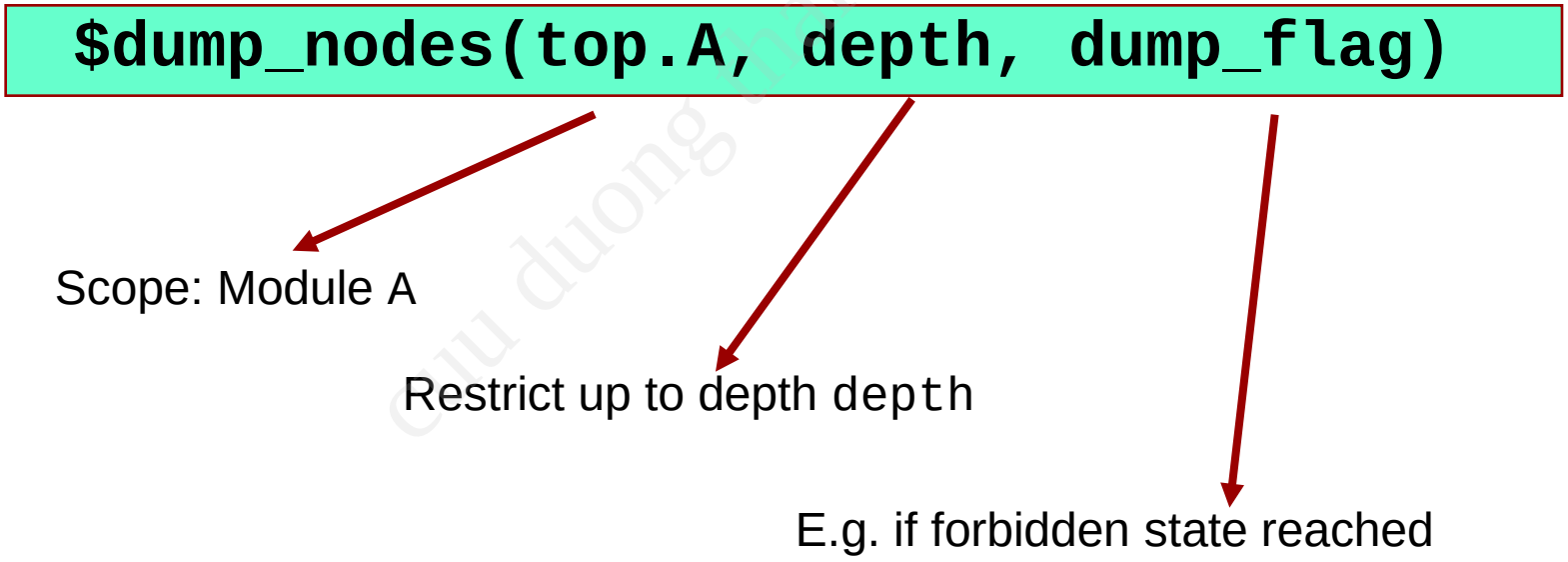
- Offline: Design State Dumping:
  - In text or VCD (Value change Dump)
    - To be viewed by a waveform viewer.
  - With dumping → simulation speed: 2x-20x decreased.
    - → If simulation performance needed, avoid dumping out all nodes.
    - Must use measures to locate bugs efficiently.

d. Response Assessment

- Measures for efficient bug locating:
  - Scope:
    - Restrict dumping to certain areas where bugs are more likely to occur.
  - Interval:
    - Turn on dumping only within a time window.
  - Depth:
    - Dump out only at a specified depth (from the top module).
  - Sampling:
    - Sample signals only when they change.
    - Sample signals with each clock: Not recommended because:
      1. Some signal values are not caught.
      2. Slow if some signals are not changed over several clocks.

d. Response Assessment

- Run-Time Restriction:
  - Dumping routines should have parameters to turn on/off dumping.



d. Response Assessment

- Golden Response:
  - Visual inspection of signal traces is suitable only
    - for a small number of signals.
    - when the user knows where to look for clues (i.e. scope is narrow).
  - For large designs, the entire set of dumped traces needs to examined.
    - → Manual inspection is not feasible.
- Common Method:
  - Compare with “golden response” automatically.
    - Unix “diff” if in text.

d. Response Assessment

- Golden Response:
  - Can be generated directly.
  - Or by a different model of the design
    - e.g. non-synthesizable higher level model or a C/C++ model.
  - If different responses →
    - bugs in design,
    - bugs in golden response.

Design testbench in Verilog

- **Initialization**

```
initial  begin
         clk = 0;
         rst = 0;
         end
```

- **Stimulus generation:**

```
#T1   rst = 1;
#T2   rst = 0;
```

- **Clock generation**

```
always #T clk = ~clk;
```

- **Response assessment**

```
if (expected != product)
begin // alert component
   $display ("ERROR: incorrect product, result = %d, …);
   $finish;
end
```

Design testbench in Verilog

Some useful commands:

- **$display**(“..”, arg2, arg3, ..); → much like printf(), displays formatted string in std output when encountered
- **$monitor**(“..”, arg2, arg3, ..); → like $display(), but .. displays string each time any of arg2, arg3, .. Changes
- **$stop**; → suspends sim when encountered
- **$finish**; → finishes sim when encountered
- **$fopen**(“filename”); → returns file descriptor (integer); then, you can use $fdisplay(fd, “..”, arg2, arg3, ..); or $fmonitor(fd, “..”, arg2, arg3, ..); to write to file
- **$fclose**(fd); → closes file
- **$random**(seed); → returns random integer; give her an integer as a seed

Design testbench in Verilog

- **$display** & **$monitor** string format

Format	Display
%d or %D	Display variable in decimal
%b or %B	Display variable in binary
%s or %S	Display string
%h or %H	Display variable in hex
%c or %C	Display ASCII character
%m or %M	Display hierarchical name
%v or %V	Display strength
%o or %O	Display variable in octal
%t or %T	Display in current time format
%e or %E	Display real number in scientific format
%f or %F	Display real number in decimal format
%g or %G	Display scientific or decimal, whichever is shorter

Class Assignment

1. Write a testbench for the following designs:
   a) Multiplexer 8 to 1
   b) Signed 8-bit Multiplier
   c) 8-bit FIFO

Requirements:

- Generate stimulus inputs
- Describe golden response for each design
- Assess the response
- Display test results
- Write a report file.