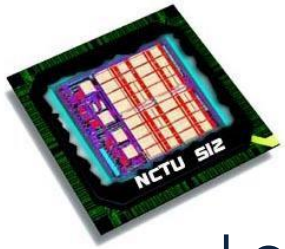
Testbench and Pattern

NYCU-EE IC LAB FALL-2023



Lecturer: Ting-Yu Chang

Outline

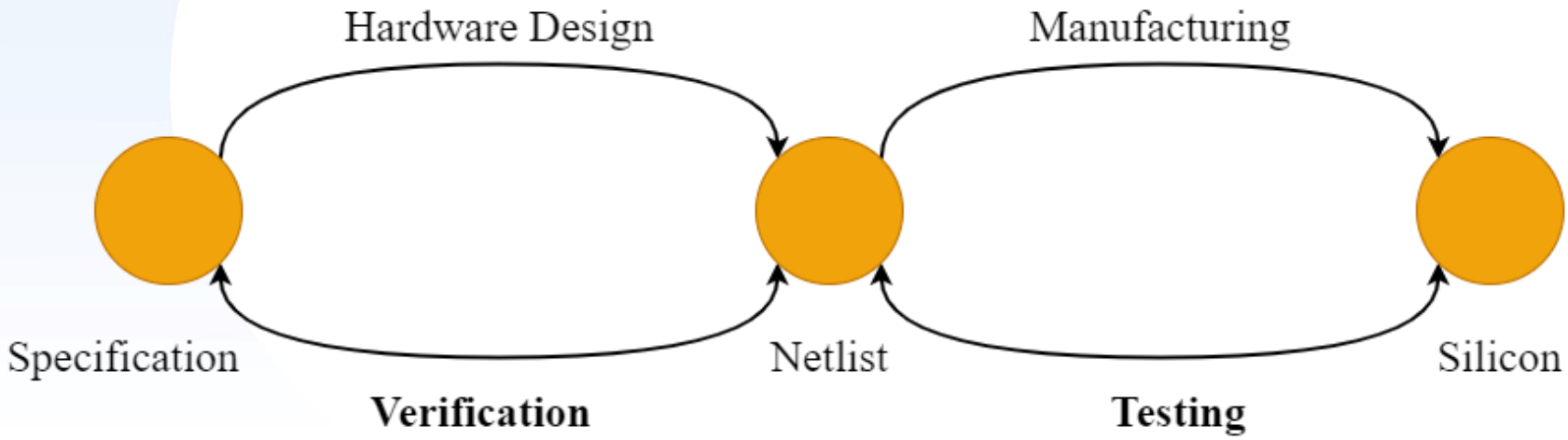
- ✓ **Section 1- Introduction to Verification**
- ✓ **Section 2- Pattern**
- ✓ **Section 3- Testbench**
- ✓ **Section 4- Environment**

Outline

- ✓ **Section 1- Introduction to Verification**
- ✓ Section 2- Pattern
- ✓ Section 3- Testbench
- ✓ Section 4- Environment

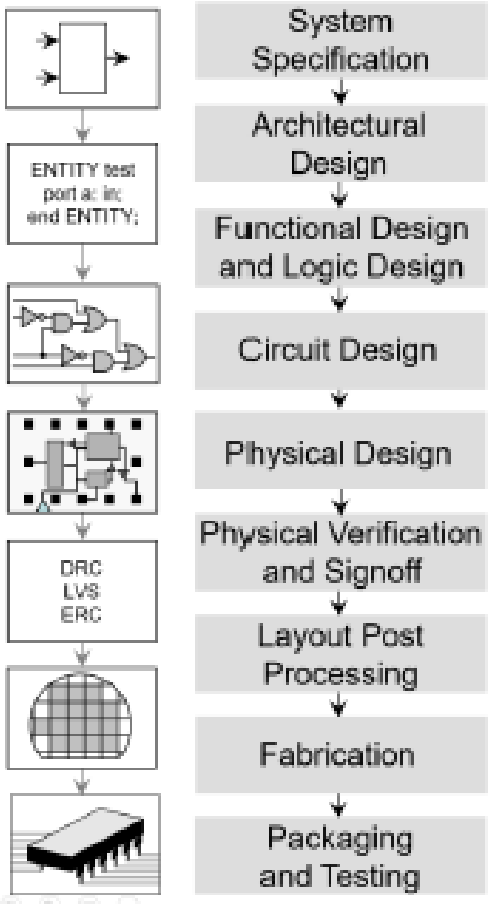
What is Verification?

- **Verification**
  - To demonstrate the functional correctness of a design
  - To make sure that you are implementing what you want
  - To ensure the result of some transformation is as expected
- **Verification vs. Testing**
  - Testing: verified the design is manufacturing correctly



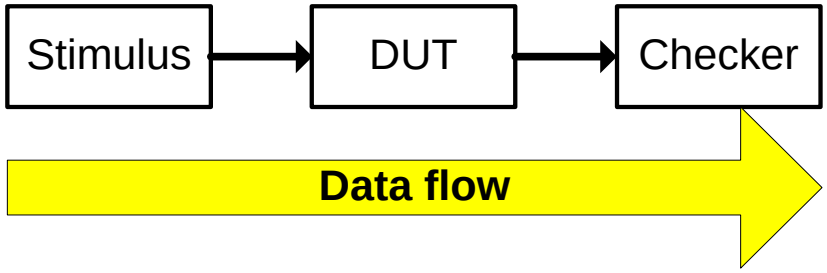
What is Verification?

- ✓ **Verification == Bug Hunting**
  - A process in which a design is verified against a given design specification before tape out
- ✓ **Verification includes:**
  - Functionality (Main goal !!)
  - Performance
  - Power
  - Security
  - Safety
- ✓ **How to perform the verification?**
  - Simulation of RTL design model (Lab03)
  - Assertions and Functional Coverage (Lab10)
  - Formal verification (Lab11)
  - Static and dynamic timing checks (Lab07)
  - Power-aware simulations (Lab08)
  - Emulation/FPGA prototyping



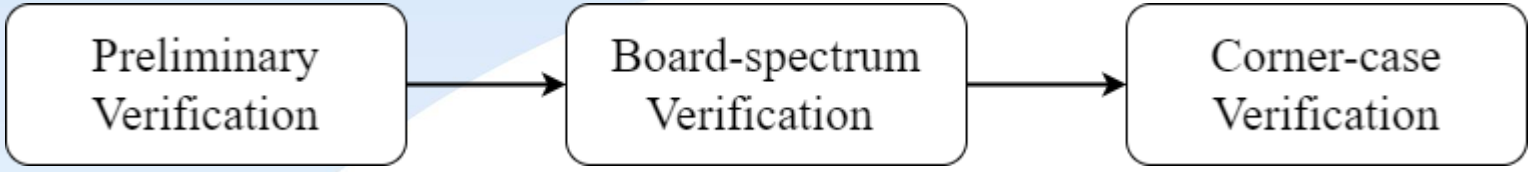
Introduction to Verification

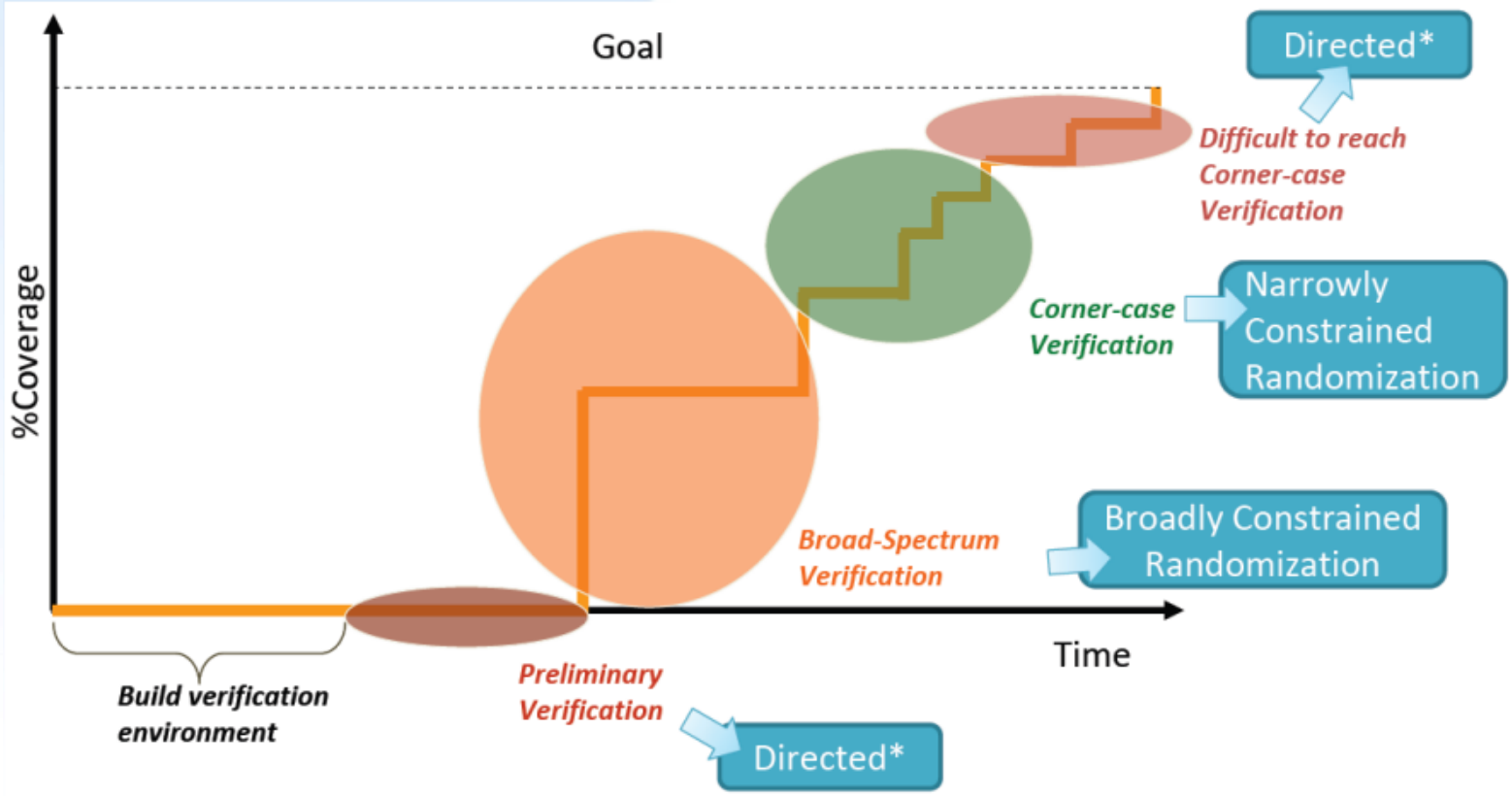
- **Stages of verification**
  - Preliminary verification -> Specification (ex. Output = 0 after reset)
  - Broad-spectrum verification -> Test pattern (ex. Random test)
  - Corner-case verification -> Special test pattern (ex. Boundary)
- **Steps of verification**
  - Generate stimulus (the inputs that drive the design)
  - Apply stimulus to DUT (Design Under Test)
  - Capture the response
  - Check for correctness
  - Measure progress against overall verification goal



Stages of verification

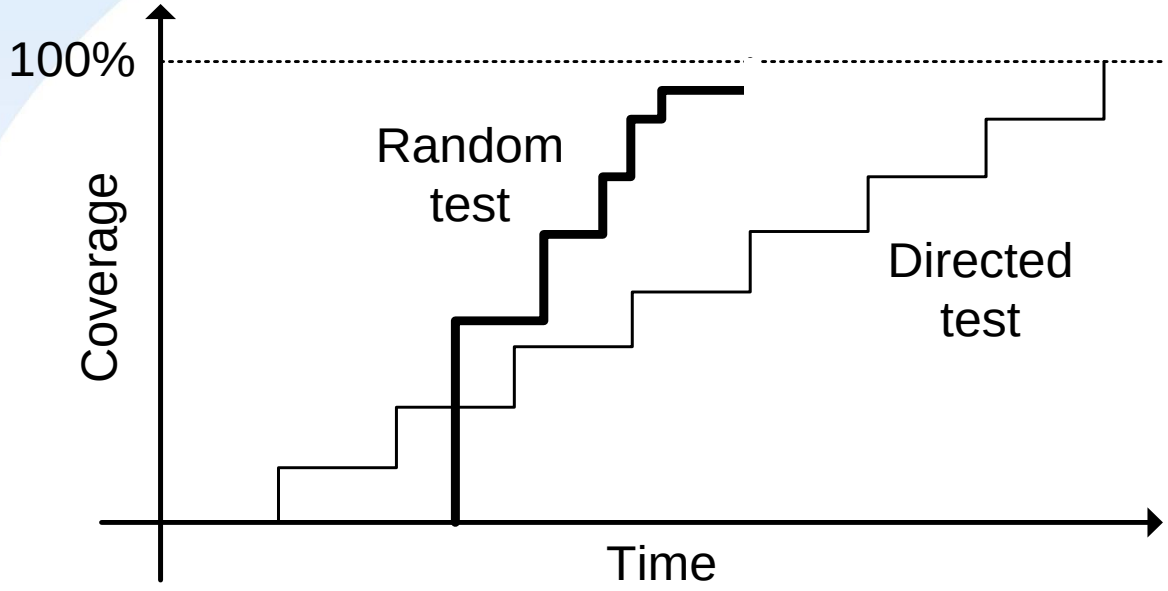
- ✓ **Which one to use?**





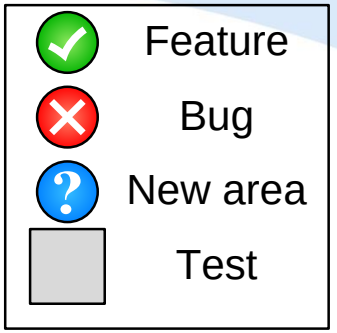
Stages of verification

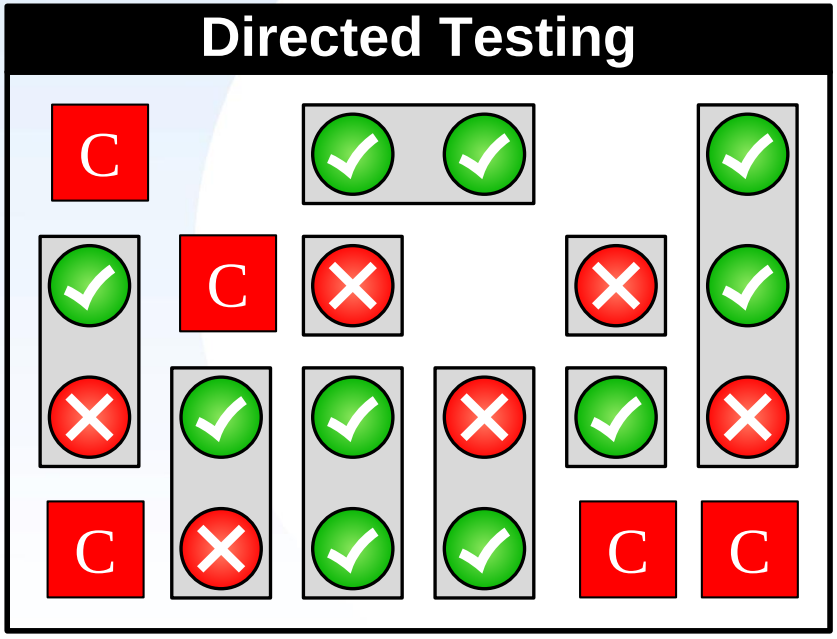
- ✓ **Time consuming vs. coverage**

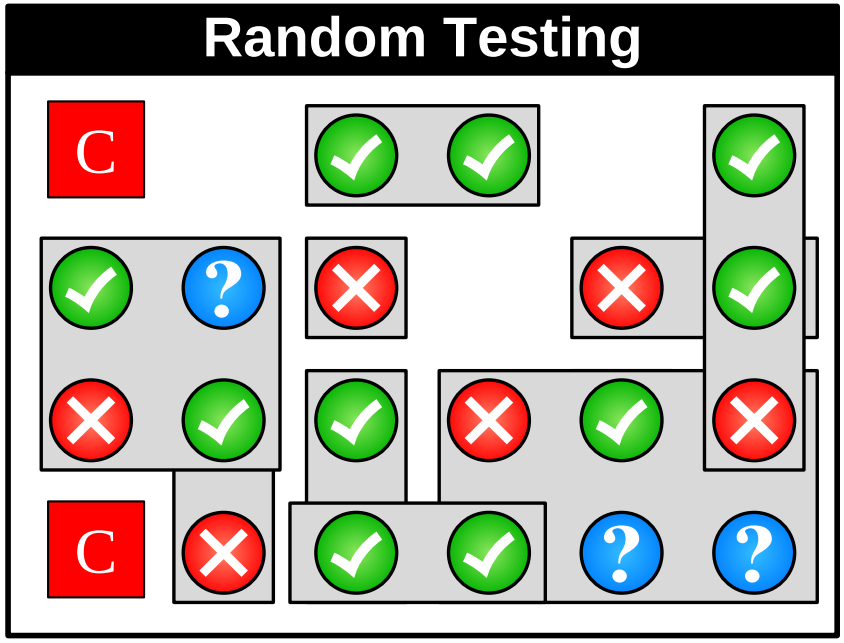


Stages of verification

- **Two kinds of strategies**
  - Directed Testing -> Check what you know
  - Random Testing -> Find what you don't know

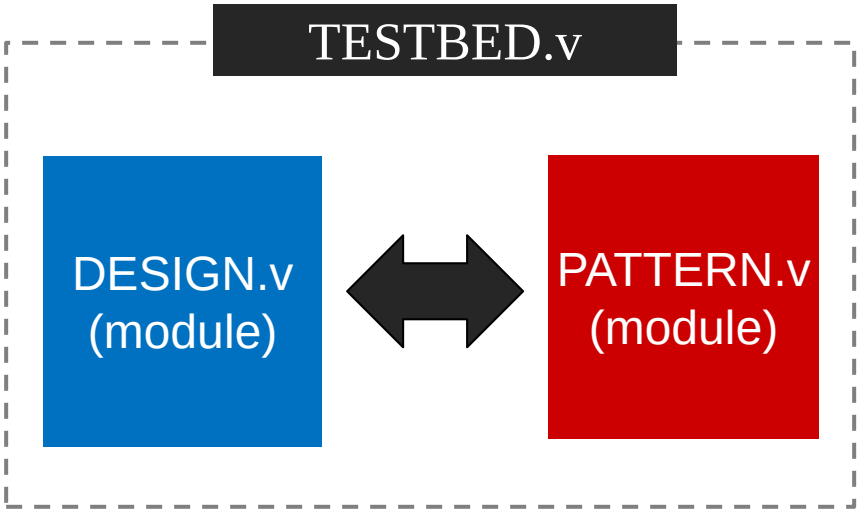






The Verilog Design Environment

- ✓ **TESTBED.v (00_TESTBED)**
  - Connecting testbench and design modules
  - Dump waveform
- ✓ **DESIGN.v (01_RTL)**
  - Design under test (DUT)
- ✓ **PATTERN.v (00_TESTBED)**
  - Pattern
  - Test program



Outline

- ✓ Section 1- Introduction to Verification
- ✓ **Section 2- Pattern**
- ✓ Section 3- Testbench
- ✓ Section 4- Environment

Pattern

✓ A simple example

```
`define CYCLE_TIME 5.0

module PATTERN (
  clk, rst_n, in_valid, a, b, out_valid, sum, overflow
);

output reg clk, rst_n, in_valid, a, b;
input out_valid, sum, overflow;

real CYCLE = `CYCLE_TIME;
parameter PATNUM = 100;
integer in_read, out_read, i_pat, i, latency, out_cnt;
reg [7:0] data_a, data_b, data_sum;
reg data_overflow;

initial clk = 0;
always #(CYCLE/2.0) clk = ~clk;

initial begin
  in_read = $fopen("../00_TESTBED/in.txt", "r");
  out_file = $fopen("../00_TESTBED/out.txt", "w");
  reset_signal_task;
  for (i_pat = 0; i_pat < PATNUM; i_pat = i_pat+1) begin
    input_task;
    wait_out_valid_task;
    check_ans_task;
    $display("PASS PATTERN NO.%4d", i_pat);
  end
  $fclose(in_read);
  $fclose(out_file);
  YOU_PASS_task;
end
```

```
task reset_signal_task; begin
  rst_n = 1'b1;
  in_valid = 1'b0;
  a = 1'bx; b = 1'bx;
  force clk = 1'b0;
  #(0.5); rst_n = 1'b0;
  #(2);
  if((out_valid !== 0)||(sum !== 0)||(overflow !== 0)) begin
    $display("Output should be 0 after RESET at %4t",$time);
    $finish;
  end
  #(10); rst_n = 1'b1;
  #(3); release clk;
end
endtask

task input_task; begin
  repeat($urandom_range(3, 5)) @(negedge clk);
  in_valid = 1'b1;
  if ($feof(in_read)) begin
    data_a = $urandom;
    data_b = $urandom;
  end else begin
    $fscanf(in_read, "%d %d", data_a, data_b);
  end
  for (i = 0; i < 8; i = i+1) begin
    a = data_a[i];
    b = data_b[i];
    @(negedge clk);
  end
  in_valid = 1'b0;
  a = 'bx;
  b = 'bx;
end endtask
```

```
task wait_out_valid_task; begin
  latency = 0;
  while(out_valid !== 1'b1) begin
    if(latency == 100) begin
      YOU_FAIL_task;
      $display("Latency are over 100 cycles  at %8t",$time);
      $finish;
    end
    latency = latency + 1;
    @(negedge clk);
  end
end endtask

task check_ans_task; begin
  {data_overflow, data_sum} = data_a + data_b;
  out_cnt = 0;
  while (out_valid === 1'b1) begin
    if (out_cnt + 1 > 8) begin
      YOU_FAIL_task;
      $display("The output count are over 8 cycles  at %8t",$time);
      $finish;
    end
    else begin
      if ((data_sum[out_cnt] !== sum) || (data_overflow !== overflow)) begin
        YOU_FAIL_task;
        $display("The output is wrong at %8t",$time);
        $finish;
      end
    end
    $fdisplay(out_file, "Data A: %0d, Data B: %0d, Sum: %0d, Overflow: %0d",
    data_a, data_b, sum, overflow);
    out_cnt = out_cnt + 1;
    @(negedge clk);
  end
  if (out_cnt < 8) begin
    YOU_FAIL_task;
    $display("The output count are less than 8 cycles  at %8t",$time);
    $finish;
  end
end endtask

task YOU_PASS_task; begin
    $display ("Congratulations!");
    $display ("Your execution cycles = %5d cycles", total_latency);
    $display ("Your clock period = %.1f ns", CYCLE);
    $display ("Total Latency = %.1f ns", total_latency*CYCLE);
    $finish;
end endtask

task YOU_FAIL_task; begin
    $display ("Fail!");
end endtask

endmodule
```

Pattern

Port declaration

```
module PATTERN (
  clk, rst_n, in_valid, a, b, out_valid, sum, overflow
);

output reg clk, rst_n, in_valid, a, b;
input out_valid, sum, overflow;
```

Data type declaration

```
real CYCLE = `CYCLE_TIME;
parameter PATNUM = 100;
integer in_read, out_read, i_pat, i, latency, out_cnt;
reg [7:0] data_a, data_b, data_sum;
reg data_overflow;
```

```
initial begin
  in_read = $fopen("../00_TESTBED/in.txt", "r");
  out_file = $fopen("../00_TESTBED/out.txt", "w");
  reset_signal_task;
  for (i_pat = 0; i_pat < PATNUM; i_pat = i_pat+1) begin
    input_task;
    wait_out_valid_task;
    check_ans_task;
    $display("PASS PATTERN NO.%4d", i_pat);
  end
  $fclose(in_read);
  $fclose(out_file);
  YOU_PASS_task;
end
```

```
task reset_signal_task; begin
  rst_n = 1'b1;
  in_valid = 1'b0;
  a = 1'bx; b = 1'bx;
  force clk = 1'b0;
  #(0.5); rst_n = 1'b0;
  #(2);
  if((out_valid !== 0)||(sum !== 0)||(overflow !== 0)) begin
    $display("Output should be 0 after RESET at %4t",$time);
    $finish;
  end
  #(10); rst_n = 1'b1;
```

Generate stimulus

```
task input_task; begin
  repeat($urandom_range(3, 5)) @(negedge clk);
  in_valid = 1'b1;
  if ($feof(in_read)) begin
    data_a = $urandom;
    data_b = $urandom;
  end else begin
    $fscanf(in_read, "%d %d", data_a, data_b);
  end
  for (i = 0; i < 8; i = i+1) begin
    a = data_a[i];
    b = data_b[i];
    @(negedge clk);
  end
  in_valid = 1'b0;
  a = 'bx;
  b = 'bx;
end endtask
```

```
task wait_out_valid_task; begin
  latency = 0;
  while(out_valid !== 1'b1) begin
    if(latency == 100) begin
      YOU_FAIL_task;
      $display("Latency are over 100 cycles  at %8t",$time);
      $finish;
```

Check result

```
task check_ans_task; begin
  {data_overflow, data_sum} = data_a + data_b;
  out_cnt = 0;
  while (out_valid === 1'b1) begin
    if (out_cnt + 1 > 8) begin
      YOU_FAIL_task;
      $display("The output count are over 8 cycles  at %8t",$time);
      $finish;
    end
    else begin
      if ((data_sum[out_cnt] !== sum) || (data_overflow !== overflow)) begin
        YOU_FAIL_task;
        $display("The output is wrong at %8t",$time);
        $finish;
      end
    end
    $fdisplay(out_file, "Data A: %0d, Data B: %0d, Sum: %0d, Overflow: %0d",
    data_a, data_b, sum, overflow);
    out_cnt = out_cnt + 1;
    @(negedge clk);
  end
  if (out_cnt < 8) begin
    YOU_FAIL_task;
    $display("The output count are less than 8 cycles  at %8t",$time);
    $finish;
  end
end endtask
```

```
task YOU_PASS_task; begin
    $display ("Congratulations!");
    $display ("Your execution cycles = %5d cycles", total_latency);
    $display ("Your clock period = %.1f ns", CYCLE);
    $display ("Total Latency = %.1f ns", total_latency*CYCLE);
    $finish;
end endtask

task YOU_FAIL_task; begin
    $display ("Fail!");
end endtask

endmodule
```

Procedural Blocks

- ✓ **Start the simulation**

```
initial begin
  in_read = $fopen("../00_TESTBED/in.txt", "r");
  out_file = $fopen("../00_TESTBED/out.txt", "w");
  reset_signal_task;
  for (i_pat = 0; i_pat < PATNUM; i_pat = i_pat+1) begin
    input_task;
    wait_out_valid_task;
    check_ans_task;
    $display("PASS PATTERN NO.%4d", i_pat);
  end
  $fclose(in_read);
  $fclose(out_file);
  YOU_PASS_task;
end
```

Procedural Blocks

- ✓ All procedural blocks will be executed concurrently.
- ✓ Initial Blocks
  - Only be executed once

```
initial
    begin
        statement…
    end
```

- ✓ Always Blocks
  - Will be executed if the condition is met

```
always@(condition_expression)
    begin
        statement…
    end
```

Procedural Blocks

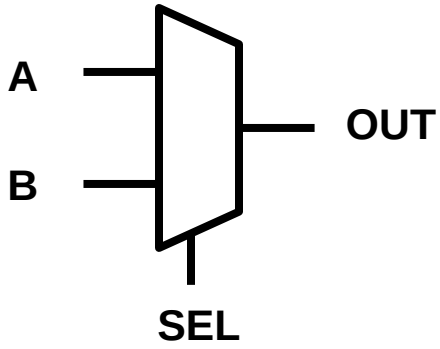
✓ A simple example

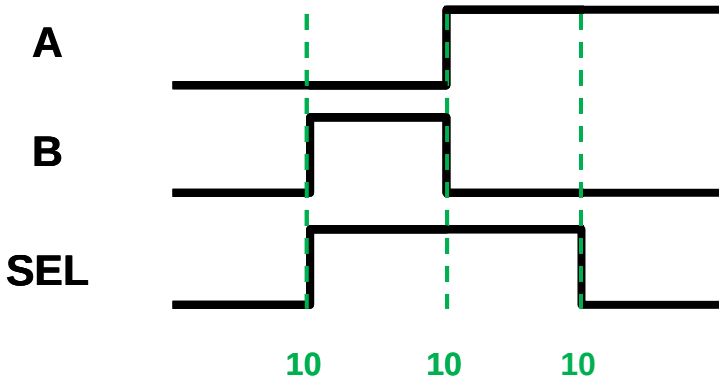
```
module Test (OUT, A, B, SEL);
output A,B,SEL;
input OUT;

initial
begin
        A=0;B=0;SEL=0;
#10     A=0;B=1;SEL=1;
#10     A=1;B=0;
#10     SEL=0;
...
#10     $finish;
end
endmodule
```

Port declaration

Delay and timing

End simulation!





➢ If simulation never stop, check …

1. No “$finish” in pattern
2. Have combinational loop in design
3. Have loop in pattern

Tasks

✓ Reset Task

```
`define CYCLE_TIME 5.0

module PATTERN (
  clk, rst_n, in_valid, a, b, out_valid, sum, overflow
);

output reg clk, rst_n, in_valid, a, b;
input out_valid, sum, overflow;

real CYCLE = `CYCLE_TIME;
parameter PATNUM = 100;
integer in_read, out_read, i_pat, i, latency, out_cnt;
reg [7:0] data_a, data_b, data_sum;
reg data_overflow;

initial clk = 0;
always #(CYCLE/2.0) clk = ~clk;

initial begin
  in_read = $fopen("../00_TESTBED/in.txt", "r");
  out_file = $fopen("../00_TESTBED/out.txt", "w");
  reset_signal_task;
  for (i_pat = 0; i_pat < PATNUM; i_pat = i_pat+1) begin
    input_task;
    wait_out_valid_task;
    check_ans_task;
    $display("PASS PATTERN NO.%4d", i_pat);
  end
  $fclose(in_read);
  $fclose(out_file);
  YOU_PASS_task;
end
```

```
task reset_signal_task; begin
  rst_n = 1'b1;
  in_valid = 1'b0;
  a = 1'bx; b = 1'bx;
  force clk = 1'b0;
  #(0.5); rst_n = 1'b0;
  #(2);
  if((out_valid !== 0)||(sum !== 0)||(overflow !== 0)) begin
    $display("Output should be 0 after RESET at %4t",$time);
    $finish;
  end
  #(10); rst_n = 1'b1;
  #(3); release clk;
end
endtask
```

- Case inequality

```
task input_task; begin
  repeat($urandom_range(3, 5)) @(negedge clk);
  in_valid = 1'b1;
  if ($feof(in_read)) begin
    data_a = $urandom;
    data_b = $urandom;
  end else begin
    $fscanf(in_read, "%d %d", data_a, data_b);
  end
  for (i = 0; i < 8; i = i+1) begin
    a = data_a[i];
    b = data_b[i];
    @(negedge clk);
  end
  in_valid = 1'b0;
  a = 'bx;
  b = 'bx;
end endtask
```

Asynchronous Reset and Clock

- **Asynchronous reset:**
  - Reset signal will reset all registers on the falling edge of reset signal.

- **Clock signal**
  - Clock signal should be forced to 0 before reset signal is given.
  - Using always procedure to produce a duty cycle 50% clock signal
  - Example:

```
reg clk, rst_n;
real CYCLE = 2.5;
initial clk = 0;
always #(CYCLE/2.0) clk = ~clk;
initial begin
   rst_n = 1'b1;
   force clk =0;
   #(0.5);  rst_n= 1'b0;
   #(10);  rst_n= 1'b1;
   #(3);  release clk;
end
```

Tasks

✓ Input data

```
`define CYCLE_TIME 5.0

module PATTERN (
  clk, rst_n, in_valid, a, b, out_valid, sum, overflow
);

output reg clk, rst_n, in_valid, a, b;
input out_valid, sum, overflow;

real CYCLE = `CYCLE_TIME;
parameter PATNUM = 100;
integer in_read, out_read, i_pat, i, latency, out_cnt;
reg [7:0] data_a, data_b, data_sum;
reg data_overflow;

initial clk = 0;
always #(CYCLE/2.0) clk = ~clk;

initial begin
  in_read = $fopen("../00_TESTBED/in.txt", "r");
  out_file = $fopen("../00_TESTBED/out.txt", "w");
  reset_signal_task;
  for (i_pat = 0; i_pat < PATNUM; i_pat = i_pat+1) begin
    input_task;
    wait_out_valid_task;
    check_ans_task;
    $display("PASS PATTERN NO.%4d", i_pat);
  end
  $fclose(in_read);
  $fclose(out_file);
  YOU_PASS_task;
end
```

```
task reset_signal_task; begin
  rst_n = 1'b1;
  in_valid = 1'b0;
  a = 1'bx; b = 1'bx;
  force clk = 1'b0;
  #(0.5); rst_n = 1'b0;
  #(2);
  if((out_valid !== 0)||(sum !== 0)||(overflow !== 0)) begin
    $display("Output should be 0 after RESET at %4t",$time);
    $finish;
  end
  #(10); rst_n = 1'b1;
  #(3); release clk;
end
endtask
```

```
task input_task; begin
  repeat($urandom_range(3, 5)) @(negedge clk);
  in_valid = 1'b1;
  if ($feof(in_read)) begin
    data_a = $urandom;
    data_b = $urandom;
  end else begin
    $fscanf(in_read, "%d %d", data_a, data_b);
  end
  for (i = 0; i < 8; i = i+1) begin
    a = data_a[i];
    b = data_b[i];
    @(negedge clk);
  end
  in_valid = 1'b0;
  a = 'bx;
  b = 'bx;
end endtask
```

- Input data change at negedge clk

Pattern

✓ Check output data

```
`define CYCLE_TIME 5.0

module PATTERN (
  clk, rst_n, in_valid, a, b, out_valid, sum, overflow
);

output reg clk, rst_n, in_valid, a, b;
input out_valid, sum, overflow;

real CYCLE = `CYCLE_TIME;
parameter PATNUM = 100;
integer in_read, out_read, i_pat, i, latency, out_cnt;
reg [7:0] data_a, data_b, data_sum;
reg data_overflow;

initial clk = 0;
always #(CYCLE/2.0) clk = ~clk;

initial begin
  in_read = $fopen("../00_TESTBED/in.txt", "r");
  out_file = $fopen("../00_TESTBED/out.txt", "w");
  reset_signal_task;
  for (i_pat = 0; i_pat < PATNUM; i_pat = i_pat+1) begin
    input_task;
    wait_out_valid_task;
    check_ans_task;
    $display("PASS PATTERN NO.%4d", i_pat);
  end
  $fclose(in_read);
  $fclose(out_file);
  YOU_PASS_task;
end
```

```
task wait_out_valid_task; begin
  latency = 0;
  while(out_valid !== 1'b1) begin
    if(latency == 100) begin
      YOU_FAIL_task;
      $display("Latency are over 100 cycles  at %8t",$time);
      $finish;
    end
    latency = latency + 1;
    @(negedge clk);
  end
end endtask
```

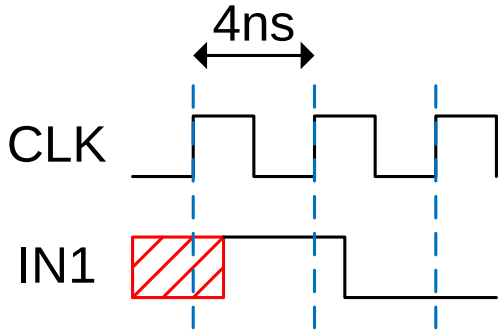
```
task check_ans_task; begin
  {data_overflow, data_sum} = data_a + data_b;
  out_cnt = 0;
  while (out_valid === 1'b1) begin
    if (out_cnt + 1 > 8) begin
      YOU_FAIL_task;
      $display("The output count are over 8 cycles  at %8t",$time);
      $finish;
    end
    else begin
      if ((data_sum[out_cnt] !== sum) || (data_overflow !== overflow)) begin
        YOU_FAIL_task;
        $display("The output is wrong at %8t",$time);
        $finish;
      end
    end
    $fdisplay(out_file, "Data A: %0d, Data B: %0d, Sum: %0d, Overflow: %0d",
    data_a, data_b, sum, overflow);
    out_cnt = out_cnt + 1;
    @(negedge clk)
  end
  if (out_cnt < 8) begin
    YOU_FAIL_task;
    $display("The output count are less than 8 cycles  at %8t",$time);
    $finish;
  end
end endtask
```

- Check output data at negedge clk

Input Delay

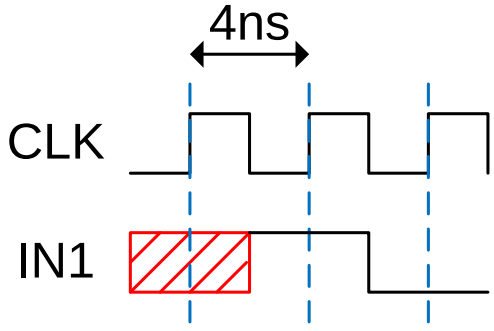
- **Consider the input interface**
  - Input signals should be synchronous to either positive clock edge or negative clock edge with specified input delays to avoid timing violation
  - Assign input delays by absolute delay value
  - Example:

```
`timescale 1ns/10ps
parameter          INDLY = 1;
bit          IN1;
initial begin
   @(posedge CLK)   #INDLY   IN1 = 1;
   @(posedge CLK)   #INDLY   IN1 = 0;
end
```



  - Assign input delays by relative delay value (relative to clock period)
  - Example:

```
`timescale 1ns/10ps
parameter  CYCLE = 4.0;
bit          IN1;
initial begin
   @(negedge CLK)  IN1 = 1;
   @(negedge CLK)  IN1 = 0;
end
```



Tasks

- ✓ **Simplified Syntax**

```
task identifier;
        parameter_declaration;
        input_decleration;
        output_decleration;
        inout_declaration;
        register_declaration;

        begin
                statement;
                ....
        end
endtask
```

Tasks

- **An example of using a task**
  - Task can take, drive and source global variables, when no local variables are used.

Global →
Local →

```
temp_in = 30;
convert(temp_in, temp_out);
$display("%d ,%d",temp_in,temp_out);

task convert;
input [7:0] temp_in;
output [7:0] temp_out;
begin
    temp_in = 20;
    temp_out = (9/5)*(temp_in+32);
end
endtask
```

```
temp_in:  30 temp_out:  52
```

Global →
Global →

```
temp_in = 30;
convert;
$display("%d,%d",temp_in,temp_out);

task convert;
begin
    temp_in = 20;
    temp_out = (9/5)*(temp_in+32);
end
endtask
```

```
temp_in:  20 temp_out:  52
```

Task and Function

- To break up a task into smaller, more manageable ones, and encapsulate reusable code, you can either divide your code into modules, or you can use tasks and functions.
- **Task**
  - A task is typically used to perform debugging operations, or to behaviorally describe hardware.
- **Function**
  - A function is typically used to perform a computation, or to represent combinational logics.

Functions

- ✓ **Simplified Syntax**

```
function type_or_range  identifier;
        parameter_declaration;
        input_decleration;
        register_declaration;

        begin
                statement;
                ....
        end
endfunction
```

Functions

- **An example of using a function**
  - Although the function cannot contain timing, you can call it from a procedural block that does.

```
…
always@(posedge CLK)
    sum = add(a,b);
…
    function [7:0] add;
        input [7:0] a;
        input [7:0] b;
        begin
            add = a + b;
        end
    endfunction
…
```

Tasks and Functions

- **Task**
  - Can have timing controls (#delay, @, wait).
  - Tasks may execute in non-zero simulation time
  - Can have port arguments (input, output, and inout) or none.
  - Does not return a value.
  - Can enable task or function.
  - Not synthesizable
  - Can call it from a procedural block
- **Function**
  - Can't have timing controls.
  - Always execute in 0 simulation time
  - Has only input arguments and no output port
  - Returns a single value through the function name.
  - Can enable function but can't enable task.
  - Synthesizable
  - Can call it from a procedural block

Generate Stimulus

```
task input_task; begin
  repeat($urandom_range(3, 5)) @(negedge clk);
  in_valid = 1'b1;
  if ($feof(in_read)) begin
    data_a = $urandom;
    data_b = $urandom;
  end else begin
    $fscanf(in_read, "%d %d", data_a, data_b);
  end
  for (i = 0; i < 8; i = i+1) begin
    a = data_a[i];
    b = data_b[i];
    @(negedge clk);
  end
  in_valid = 1'b0;
  a = 'bx;
  b = 'bx;
end endtask
```

- **Broad-spectrum verification**
- **Corner-case verification**

Generate Stimulus

- **Using Verilog random system task**
  - $random(seed);
    - Return 32-bit signed value
    - Seed is optional
  - $urandom(seed);
    - Return 32-bit unsigned value
    - Seed is optional
  - $urandom_range(int unsigned MAX, int unsigned MIN=0);
    - Return value inside range
- **Using high level language with file IO**
  - Generate random stimulus from MATLAB or Python etc. and output the stimulus into files.
  - Read the files in pattern.v

Generate Stimulus

- ✓ **A simple example**

```
integer SEED, number;
SEED = 123;
number = $random(SEED) % 7;
```

```
integer SEED, number;
SEED = 123;
number = $urandom(SEED) % 7;
```

```
integer SEED;
reg[3:0] number;
SEED = 123;
number = $random(SEED);
number = number % 7;
```

```
integer SEED;
reg[3:0] number;
SEED = 123;
number = $random(SEED) % 7;
```

```
integer SEED;
reg[3:0] number;
SEED = 123;
number = $random(SEED) % 'd7;
```

Same ?

Generate Stimulus

- **A simple example**
  - Produce random number in 0~6

May be negative

```
integer SEED, number;
SEED = 123;
number = $random(SEED) % 7;
```

signed signed signed

(signed operation)

Correct

```
integer SEED, number;
SEED = 123;
number = $urandom(SEED) % 7;
```

signed unsigned signed

(unsigned operation)

Correct

```
integer SEED;
reg[3:0] number;
SEED = 123;
number = $random(SEED);
number = number % 7;
```

unsigned → number = $random(SEED);

unsigned unsigned signed

(unsigned operation)

The value may not in range 0~6

```
integer SEED;
reg[3:0] number;
SEED = 123;
number = $random(SEED) % 7;
```

unsigned signed signed

(signed operation)

Generate Stimulus

- **A simple example**
  - Produce random number in 0~6

Correct

```
integer SEED;
reg[3:0] number;
SEED = 123;
number = $random(SEED) % `d7;
```

unsigned — signed (unsigned operation) — unsigned

The key point is to use unsigned operation!

File I/O

✓ Open file

- $fopen opens the specified file and returns a 32-bit descriptor.
  - file_descriptor = $fopen("file_name" , "type");
    - file_descriptor: bit 32 always be set (=1), remaining bits hold a small number indicating what file is opened.
    - type: "r" , open for read; "w", open for write

```
file = $fopen("test.txt","w");
file2 = $fopen("test2.txt","w");
file3 = $fopen("test3.txt","w");
```

```
10000000000000000000000000000011
10000000000000000000000000000100
10000000000000000000000000000101
```

  - multi_channel_descriptor = $fopen("file_name");
    - Multi_channel_descriptor: bit 32 always be clear (=0), bit 0 represent standard output, each remaining bit represents a single output channel.

```
file = $fopen("test.txt");
file2 = $fopen("test2.txt");
file3 = $fopen("test3.txt");
```

```
00000000000000000000000000000010
00000000000000000000000000000100
00000000000000000000000000001000
```

File I/O

- ✓ **Close file**
  - $fclose system task closes the channels specified in the multichannel descriptor
    - $fclose(<multichannel_descriptor>);
    - The $fopen task will reuse channels that have been closed

File Input

- **Read data from specific file**
  - $fgetc – reading a byte at a time
    - c = $fgetc(<descriptor>);
  - $fgets – reading a line at a time
    - i = $fgets(string, <descriptor>);
  - $fscanf – reading formatted data
    - i = $fscanf(<descriptor>,” text”, signal,signal,…);

```
opa=101,opb=1010
opa=3,opb=12
```

```
rc = $fscanf(file_input,"opa=%d,opb=%d",opa,opb);
```

  - $readmemb, $readmemh
    - $readmemb("file_name", memory_name [, start_address [, end_address]]);
    - $readmemh("file_name", memory_name [, start_address [, end_address]]);

```
$readmemh ("IN.txt",in);
$readmemh ("OUT.txt",out);
```

File Input

- ✓ **A simple example for $readmemb**

```
@002
11111111 01010101
00000000 10101010
@006
1111zzzz 00001111
```

```
reg [7:0] meme[0:7];
$readmemb("test.txt",meme);
```

```
meme[0]: xxxxxxxx
meme[1]: xxxxxxxx
meme[2]: 11111111
meme[3]: 01010101
meme[4]: 00000000
meme[5]: 10101010
meme[6]: 1111zzzz
meme[7]: 00001111
```

```
@002
11111111_01010101
00000000_10101010
@006
1111zzzz_00001111
```

```
reg [15:0] meme[0:7];
$readmemb("test.txt",meme);
```

```
meme[0]: xxxxxxxxxxxxxxxx
meme[1]: xxxxxxxxxxxxxxxx
meme[2]: 1111111101010101
meme[3]: 0000000010101010
meme[4]: xxxxxxxxxxxxxxxx
meme[5]: xxxxxxxxxxxxxxxx
meme[6]: 1111zzzz00001111
meme[7]: xxxxxxxxxxxxxxxx
```

File Input

- ✓ **A special example for $readmemb**

```
reg [7:0] meme[0:7];
$readmemb("IN.txt", meme);
```

```
1 @002
2 11111111 01010101
3 00000000 10101010
4 11110000 00001111
5 @004
6 1z1z1z1z 0z0z0z0z
```

```
meme[0]: xxxxxxxx
meme[1]: xxxxxxxx
meme[2]: 11111111
meme[3]: 01010101
meme[4]: 1z1z1z1z
meme[5]: 0z0z0z0z
meme[6]: 11110000
meme[7]: 00001111
```

```
1 @002
2 11111111 01010101
3 00000000 10101010
4 11110000 00001111
5 zzzz0000 zzzz1111
```

```
meme[0]: xxxxxxxx
meme[1]: xxxxxxxx
meme[2]: 11111111
meme[3]: 01010101
meme[4]: 00000000
meme[5]: 10101010
meme[6]: 11110000
meme[7]: 00001111
```

File Output

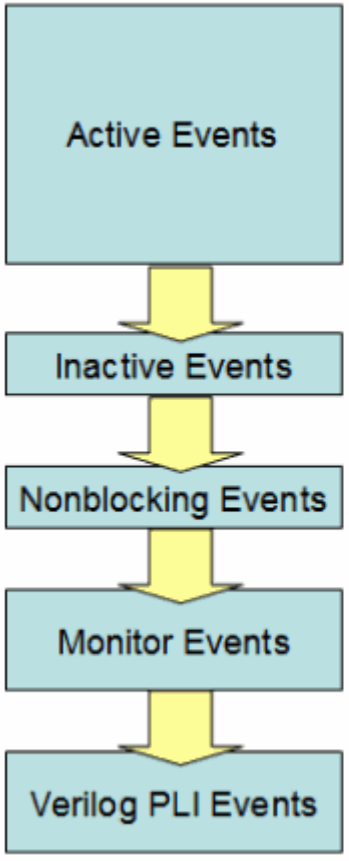
- **Display tasks that writes to specific files**
  - $fdisplay, $fwrite, $fstrobe, $fmonitor
    - $fdisplay (<descriptor>,[“format_specifiers”,] <argument_list>);
    - $fwrite (<descriptor>,[“format_specifiers”,] <argument_list>);
    - $fstrobe (<descriptor>,[“format_specifiers”,] <argument_list>);
    - $fmonitor (<descriptor>,[“format_specifiers”,] <argument_list>);

```
$fdisplay(file_output,"%d + %d = %d",opa,opb,sum);
```

	Active Events	Monitor Events
w/ newline	fdisplay	fstrobe
w/o newline	fwrite	fmonitor

Stratified Event Queue

✓ Stratified Event Queue of Verilog



Blocking assignments
Evaluate RHS of nonblocking assignments
Continuous assignments
$display command execution
$write command execution

#0 blocking assignments

Update LHS of nonblocking assignments

$monitor command execution
$strobe command execution

$fsdbDumpfile()
$fsdbDumpvars()

```
a = b + c;
assign a = b + c;
a <= b + c;
```

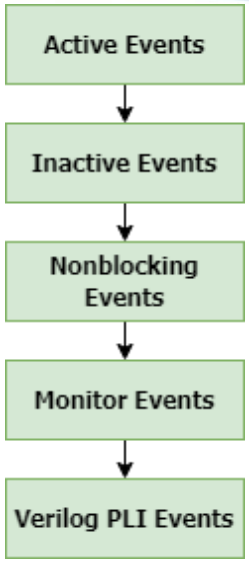
```
#0 a = b + c;
```

```
a <= b + c;
```

*PLI (Programming Language Interface)

Stratified Event Queue

```
1  module nb_schedule1;
2
3   reg a, b;
4  integer fp;
5
6  initial begin
7     fp = $fopen("log.txt","w");
8     a = 0;
9     b = 0;
10    #1;
11    a = 0;
12    b = 1;
13    a <= b;
14    b <= a;
15
16    $monitor("%0dns :\$monitor: a=%b b=%b"  , $stime, a, b);
17    $display("%0dns :\$display: a=%b b=%b"  , $stime, a, b);
18    $strobe ("%0dns :\$strobe : a=%b b=%b\n", $stime, a, b);
19    $fwrite(fp, "%0dns :\$fwrite : a=%b b=%b\n", $stime, a, b);
20 #0 $display("%0dns :#0      : a=%b b=%b"  , $stime, a, b);
21
22
23 #1 $monitor("%0dns :\$monitor: a=%b b=%b"  , $stime, a, b);
24    $display("%0dns :\$display: a=%b b=%b"  , $stime, a, b);
25    $strobe ("%0dns :\$strobe : a=%b b=%b\n", $stime, a, b);
26    $fwrite(fp, "%0dns :\$fwrite : a=%b b=%b\n", $stime, a, b);
27 #0 $display("%0dns :#0      : a=%b b=%b"  , $stime, a, b);
28
29
30    $fclose(fp);
31 end
32
33 initial begin
34   $fsdbDumpfile("nb_schedule1.fsdb");
35   $fsdbDumpvars(0, nb_schedule1);
36 end
37
38 endmodule
```



Result

```
# 1ns :$display: a=0 b=1
# 1ns :#0      : a=0 b=1
# 1ns :$monitor: a=1 b=0
# 1ns :$strobe : a=1 b=0
#
# 2ns :$display: a=1 b=0
# 2ns :#0      : a=1 b=0
# 2ns :$monitor: a=1 b=0
# 2ns :$strobe : a=1 b=0
```

File Output

- **Display tasks that writes to specific files**
  - All these four output system tasks support multiple default bases
    - $fdisplay /$fdisplayh /$fdisplayb /$fdisplayo
    - $fwrite /$fwriteh /$fwriteb /$fwriteo
    - $fstrobe /$fstrobeh /$fstrobeb /$fstrobeo
    - $fmonitor /$fmonitorh /$fmonitorb /$fmonitoro

```
reg [3:0] num;
initial begin
    num = 15;
    $fdisplay (f_out, "Num: %d", num);
    $fdisplayh(f_out, "Num: %d", num);
    $fdisplayb(f_out, "Num: %d", num);
    $fdisplayo(f_out, "Num: %d", num);
    $fdisplay (f_out, "========");
    $fdisplay (f_out, "Num: ", num);
    $fdisplayh(f_out, "Num: ", num);
    $fdisplayb(f_out, "Num: ", num);
    $fdisplayo(f_out, "Num: ", num);
end
```

```
Num: 15
Num: 15
Num: 15
Num: 15
========
Num: 15
Num: f
Num: 1111
Num: 17
```

File I/O

✓ Complete file I/O example

```
`define CYCLE_TIME 5.0

module PATTERN (
  clk, rst_n, in_valid, a, b, out_valid, sum, overflow
);

output reg clk, rst_n, in_valid, a, b;
input out_valid, sum, overflow;

real CYCLE = `CYCLE_TIME;
parameter PATNUM = 100;
integer in_read, out_read, i_pat, i, latency, out_cnt;
reg [7:0] data_a, data_b, data_sum;
reg data_overflow;
```

Open file

```
  in_read = $fopen("../00_TESTBED/in.txt", "r");
  out_file = $fopen("../00_TESTBED/out.txt", "w");
  reset_signal_task;
  for (i_pat = 0; i_pat < PATNUM; i_pat = i_pat+1) begin
    input_task;
    wait_out_valid_task;
    check_ans_task;
    $display("PASS PATTERN NO.%4d", i_pat);
  end
  $fclose(in_read);
  $fclose(out_file);
```

Close file

```
task reset_signal_task; begin
  rst_n = 1'b1;
  in_valid = 1'b0;
  a = 1'bx; b = 1'bx;
  force clk = 1'b0;
  #(0.5); rst_n = 1'b0;
  #(2);
  if((out_valid !== 0)||(sum !== 0)||(overflow !== 0)) begin
    $display("Output should be 0 after RESET at %4t",$time);
    $finish;
  end
  #(10); rst_n = 1'b1;
  #(3); release clk;
end
endtask

task input_task; begin
  repeat($urandom_range(3, 5)) @(negedge clk);
  in_valid = 1'b1;
  if ($feof(in_read)) begin
    data_a = $urandom;
    data_b = $urandom;
  end else begin
    $fscanf(in_read, "%d %d", data_a, data_b);
```

Read file

```
    @(negedge clk);
  end
  in_valid = 1'b0;
  a = 'bx;
  b = 'bx;
end endtask

task wait_out_valid_task; begin
  latency = 0;
  while(out_valid !== 1'b1) begin
    if(latency == 100) begin
      YOU_FAIL_task;
      $display("Latency are over 100 cycles  at %8t",$time);
      $finish;
    end
    latency = latency + 1;
    @(negedge clk);
  end
end endtask

task check_ans_task; begin
  {data_overflow, data_sum} = data_a + data_b;
  out_cnt = 0;
  while (out_valid === 1'b1) begin
    if (out_cnt + 1 > 8) begin
      YOU_FAIL_task;
      $display("The output count are over 8 cycles  at %8t",$time);
      $finish;
    end
    else begin
      if ((data_sum[out_cnt] !== sum) || (data_overflow !== overflow)) begin
                                                                  ",$time);
```

Write file

```
      $fdisplay(out_file, "Data A: %0d, Data B: %0d, Sum: %0d, Overflow: %0d",
      data_a, data_b, sum, overflow);
      out_cnt = out_cnt + 1;
      @(negedge clk);
    end
    if (out_cnt < 8) begin
      YOU_FAIL_task;
      $display("The output count are less than 8 cycles  at %8t",$time);
      $finish;
    end
end endtask

task YOU_PASS_task; begin
    $display ("Congratulations!");
    $display ("Your execution cycles = %5d cycles", total_latency);
    $display ("Your clock period = %.1f ns", CYCLE);
    $display ("Total Latency = %.1f ns", total_latency*CYCLE);
    $finish;
end endtask

task YOU_FAIL_task; begin
    $display ("Fail!");
end endtask

endmodule
```

Display Information

- **There are mainly four kinds of instruction to display information.**
  - $display ([“format_specifiers”,] <argument_list>);
  - $write ([“format_specifiers”,] <argument_list>);
  - $strobe ([“format_specifiers”,] <argument_list>);
  - $monitor([“format_specifiers”,] <argument_list>);

	Active Events	Monitor Events
w/ newline	display	strobe
w/o newline	write	monitor

Display Information (cont.)

- The following escape sequences are used for display special characters

\n	New line character	\"	" character
\t	Tab character	\o	A character specified in 1-3 octal digits
\\	\ character	%%	Percent character

- The following table shows the escape sequences used for format specifications

specifier	Display format	specifier	Display format
%h or %H	Hexadecimal	%m or %M	Hierarchical name
%d or %D	Decimal	%s or %S	String
%o or %O	Octal	%t or %T	Current time
%b or %B	Binary	%e or %E	real number in exponential
%c or %C	ASCII character	%f or %F	Real number in decimal
%v or %V	Net signal strength	%p or %P	Array <only for System Verilog>

Display Information

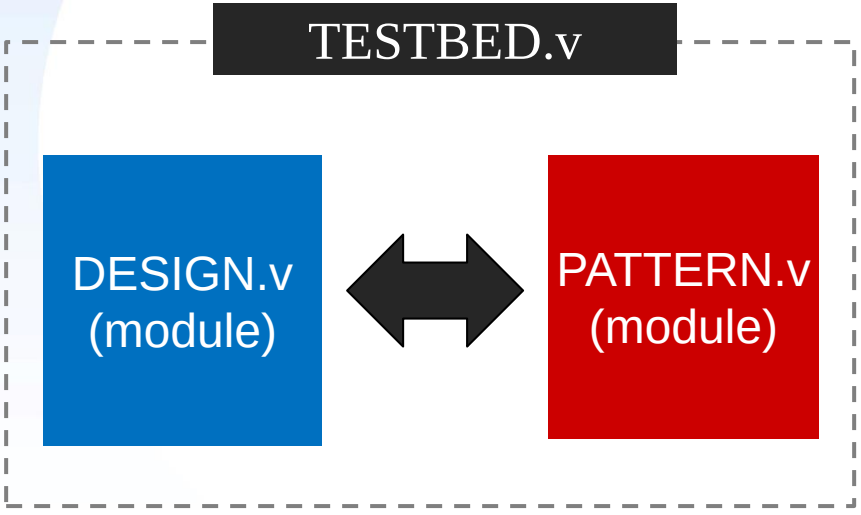
- **More detail**
  - http://verilog.renerta.com/source/vrg00013.htm
  - http://www.cnblogs.com/oomusou/archive/2011/06/25/verilog_strobe.html

Outline

- ✓ Section 1- Introduction to Verification
- ✓ Section 2- Pattern
- ✓ **Section 3- Testbench**
- ✓ Section 4- Environment

Testbench

- ✓ **Encapsulate DESIGN.v and PATTERN.v to be a top verification file**
- ✓ **Key element**
  - Timescale
  - Dump Waveform
  - Port Connection




```
`timescale 1ns/10ps

`include "PATTERN.v"
`ifdef RTL
  `include "ADD.v"
`endif
`ifdef GATE
  `include "ADD_SYN.v"
`endif

module TESTBED;

wire clk, rst_n, in_valid, a, b;
wire out_valid, sum, overflow;

initial begin
  `ifdef RTL
    $fsdbDumpfile("ADD.fsdb");
    $fsdbDumpvars(0,"+mda");
  `endif
  `ifdef GATE
    $sdf_annotate("ADD_SYN.sdf", u_ADD);
    $fsdbDumpfile("ADD_SYN.fsdb");
    $fsdbDumpvars(0,"+mda");
  `endif
end

ADD u_ADD(
    .clk(clk),
    .rst_n(rst_n),
    .in_valid(in_valid),
    .a(a),
    .b(b),
    .out_valid(out_valid),
    .sum(sum),
    .overflow(overflow)
    );

PATTERN u_PATTERN(
    .clk(clk),
    .rst_n(rst_n),
    .in_valid(in_valid),
    .a(a),
    .b(b),
    .out_valid(out_valid),
    .sum(sum),
    .overflow(overflow)
    );

endmodule
```

Timescale

- ✓ **`timescale**
  - A compiler directive
  - Specifies the unit of measurement for time and the degree of precision of the time
  - Syntax:

```
`timescale <time_unit> / <time_precision>
```

- time_unit specifies the unit of measurement for times and delays
- time_precision specifies the degree of precision
- The time_precision must be at least as precise as the time_unit
- Valid integers are 1, 10, and 100
- Valid character strings are s, ms, us, ns, ps, and fs

Timescale

✓ A simple example:

```
`timescale 1ns/100ps
module TEST;
        parameter  CYCLE = 2.5;
        reg    CLK;
        initial CLK = 1;
        always #(CYCLE/2.0) CLK = ~CLK;
endmodule
```

CYCLE/2.0 = 1.25 ns
Precision requirement: 0.01ns < 100ps
Precision loss !! The CYCLE/2.0 will become 1.3ns

‘timescale Unit/Precision	Delay	Time Delay
‘timescale 10ns/1ns	#5	50ns
‘timescale 10ns/1ns	#5.738	57ns
‘timescale 10ns/10ns	#5.5	60ns
‘timescale 10ns/100ps	#5.738	57.4ns

- Note: if you use memory in your design, you should set timescale according to the timescale specified by memory file

Timescale

✓ A simple example:

```
`timescale 1ns/100ps → 10ps (avoid precision loss)
module TEST;
        parameter  CYCLE = 2.5;
        reg    CLK;
        initial CLK = 1;
        always #(CYCLE/2.0) CLK = ~CLK;
endmodule
```

CYCLE/2.0 = 1.25 ns
Precision requirement: 0.01ns < 100ps
Precision loss !! The CYCLE/2.0 will become 1.3ns

‘timescale Unit/Precision	Delay	Time Delay
‘timescale 10ns/1ns	#5	50ns
‘timescale 10ns/1ns	#5.738	57ns
‘timescale 10ns/10ns	#5.5	60ns
‘timescale 10ns/100ps	#5.738	57.4ns

- Note: if you use memory in your design, you should set timescale according to the timescale specified by memory file

Dump Waveform

- **There are many different waveform file formats.**
  - Value Change Dump (.VCD)
    - Included in Verilog HDL IEEE Standard
  - Wave Log File (.wlf)
    - Mentor Graphics – Modelsim
  - SHM (.shm)
    - Cadence – NC Verilog / Simvision
  - VPD (.vpd)
    - Synopsys - VCS
  - Fast Signal DataBase (.fsdb)
    - Spring Soft (Merged with Synopsys) - Debussy/Verdi

Dump Waveform

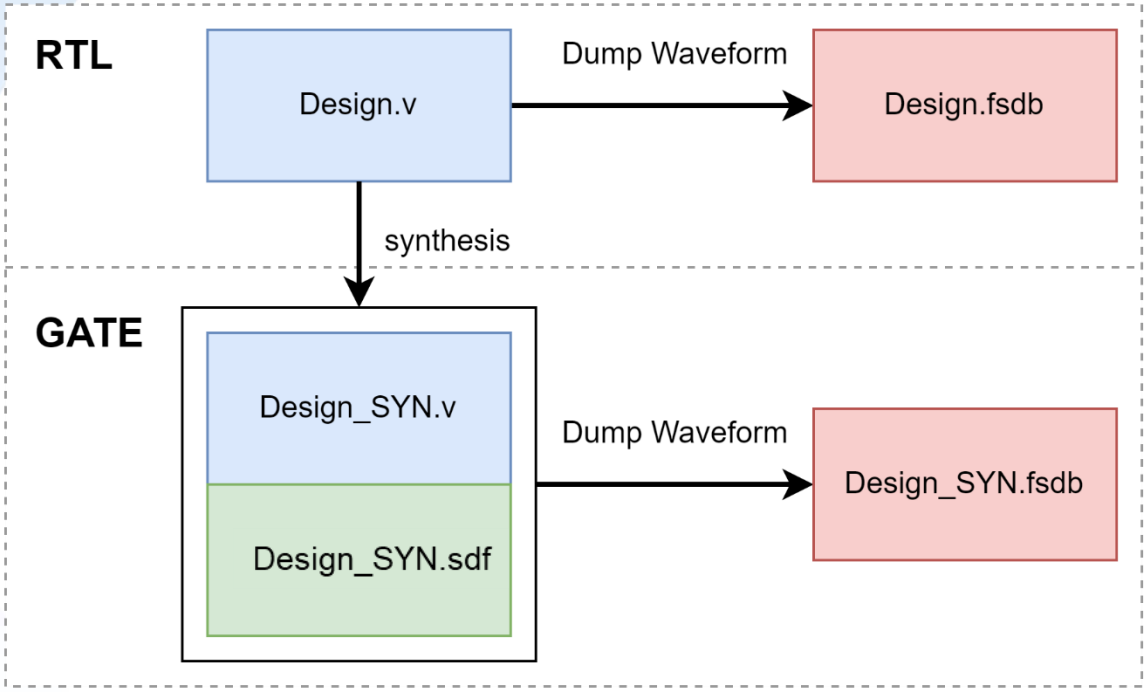
- **Command often used**
  - $fsdbDumpfile(fsdb_name[,limit_size])
    - fsdb_name: assign waveform file name
    - (Optional) limit_size: assign the limitation of file size
  - $fsdbDumpvars([depth, instance][,"option"])
    - depth: level of waveform to be dumped
    - instance: module to be dumped
    - "option": other additional specification, ex: "+mda"
  - $sdf_annotate("sdf_file"[,instance][,config_file][,log_file][,mtm_spec][,scale_factors][,scale_type])

Dump Waveform

- **Parameters of $fsdbDumpvars()**
  - Depth
    - 0: all signals in all scopes
    - 1: all signals in current scope (scope of TESTBED.v)
    - 2: all signals in the current scope and all scopes one level below
    - n: all signals in the current scope and all scopes n-1 levels below
  - Option
    - "+IO_only": only IO port signals will be dumped.
    - "+Reg_only": only reg type signals will be dumped.
    - "+all": dump all signals including the memory, MDA, packed array, structure, union and packed structure signals in all scopes specified in $fsdbDumpvars.
    - "+mda": dump all memory and MDA(multiple dimensional array) signals in all scopes specified in $fsdbDumpvars.
    - For further information, please refer http://www.eetop.cn/blog/html/55/1518355-433686.html

Dump Waveform

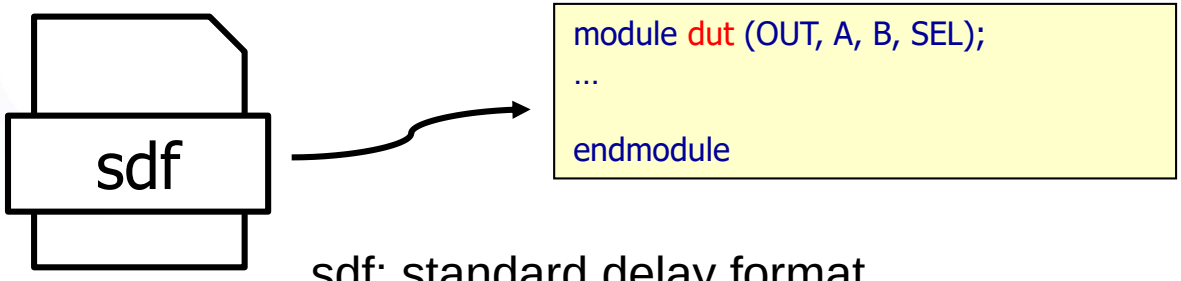
- **A simple example:**
  - Used in RTL simulation or gate-level simulation
  - Dump wave form in fsdb format for viewing in nWave
  - Include timing information in the simulation



Dump Waveform

- **A simple example:**
  - Used in RTL simulation or gate-level simulation
  - Dump wave form in fsdb format for viewing in nWave
  - Include timing information in the simulation

```
initial begin
        `ifdef RTL
                $fsdbDumpfile("Design.fsdb");
                $fsdbDumpvars(0,"+mda");
        `endif
        `ifdef GATE
                $fsdbDumpfile("Design_SYN.fsdb");
                $fsdbDumpvars(0,"+mda");
                $sdf_annotate("CORE_SYN.sdf", dut);
        `endif
end
```

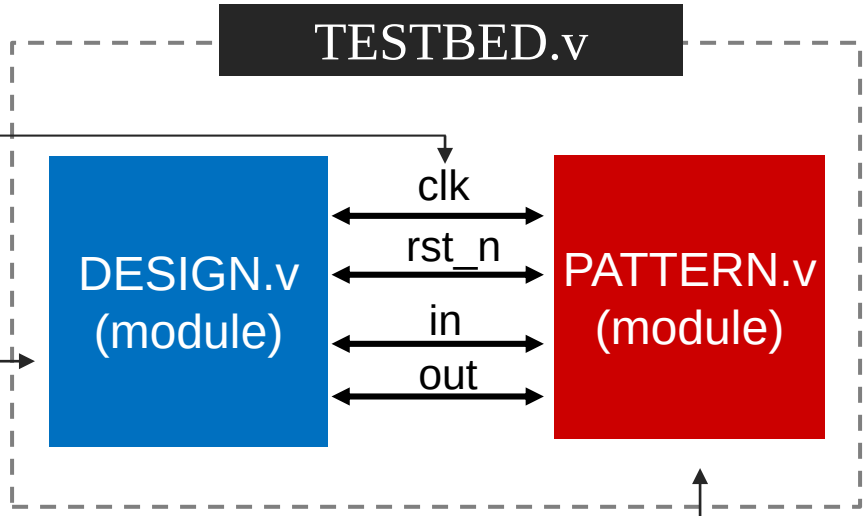


sdf: standard delay format

Port Connection

- ✓ **The input and output is reverse between design.v and pattern.v.**
- ✓ **A simple example:**

```
//input signals
wire clk, rst_n, in;
//output signals
wire out;
....
DESIGN U_DESIGN(
                .in(in)
                .rst_n(rst_n)
                .clk(clk)
                .out(out)
);
PATTERN U_PATTERN(
                .in(in)
                .rst_n(rst_n)
                .clk(clk)
                .out(out)
);
endmodule
```



Outline

- ✓ Section 1- Introduction to Verification
- ✓ Section 2- Pattern
- ✓ Section 3- Testbench
- ✓ **Section 4- Environment**

Simulation Environment

- ✓ **00_TESTBED**
  - Pattern and testbench location.
- ✓ **01_RTL**
  - RTL code functionality simulation
- ✓ **02_SYN**
  - Circuit synthesize
- ✓ **03_GATE**
  - Gate level simulation
- ✓ **04_MEM**
- ✓ **05_APR**
- ✓ **06_POST**

00_TESTBED

- **00_TESTBED:**
  - TESTBED.v
  - PATTERN.v
  - filelist.f: files to run simulation.

01_RTL

- **01_RTL:**
  - DESIGN.v
  - **01_run_vcs_rtl**: run simulation using VCS powered by Synopsys.
  - **02_irun_rtl**: run simulation using irun powered by Cadence.
  - **03_xrun_rtl**: run simulation using xrun powered by Cadence.
  - **04_verdi**: run debug tool using Verdi powered by Synopsys.
  - **05_nWave**: view waveform using nWave powered by Synopsys.
  - **08_check**: check the simulation result.
  - **09_clean_up**: clean the temporary files during simulation.
  - Link:
    - PATTERN.v & TESTBED.v
    - filelist.f

- We use VCS to simulate when demo in this semester.

02_SYN

- **02_SYN:**
  - syn.tcl
  - **01_run_dc_shell**: synthesize using Design Compiler shell
  - **02_run_design_vision**: synthesize using Design Vision (GUI version)
  - **03_read_dv_rtl**: read design (Verilog code) into Design Vision
  - **04_read_ddc**: read design – (Design Data Checkpoint (DDC) files) into Design Vision.
  - **08_check**: check the synthesis result.
  - **09_clean_up**: clean the temporary files during synthesis.
  - Generate file:
    - DESIGN_SYN.v & DESIGN_SYN.sdf

03_GATE

- **03_GATE:**
  - **01_run_vcs_rtl**: run simulation using VCS powered by Synopsys.
  - **02_irun_rtl**: run simulation using irun powered by Cadence.
  - **03_xrun_rtl**: run simulation using xrun powered by Cadence.
  - **04_verdi**: run debug tool using Verdi powered by Synopsys.
  - **05_nWave**: view waveform using nWave powered by Synopsys.
  - **08_check**: check the simulation result.
  - **09_clean_up**: clean the temporary files during simulation.
  - Link:
    - DESIGN_SYN.v & DESIGN_SYN.sdf
    - PATTERN.v & TESTBED.v
    - filelist.f

Supplemental Instructions

- ✓ **Check quota**
  - quota

```
[iclab180@ee01 ~]$ quota
Disk quotas for user iclab180 (uid 16975):
     Filesystem  blocks   quota   limit   grace   files   quota   limit   grace
    raid:/RAID2   73428  15728640 20971520            1134       0       0
```

- ✓ **Pattern encryption**
  - ncprotect -autoprotect PATTERN.v
    - For Cadence tool: irun, xrun
  - vcs +autoprotect PATTERN.v
    - Encrypt the whole content except module name.
    - For Synopsys tool: vcs
  - vcs +auto2protect PATTERN.v
    - Encrypt the whole content except module name and port declaration.
    - For Synopsys tool: vcs
  - Make sure there are no PATTERN.vp in the directory before encrypt.