Lecture 5 Testbenches

Outlines

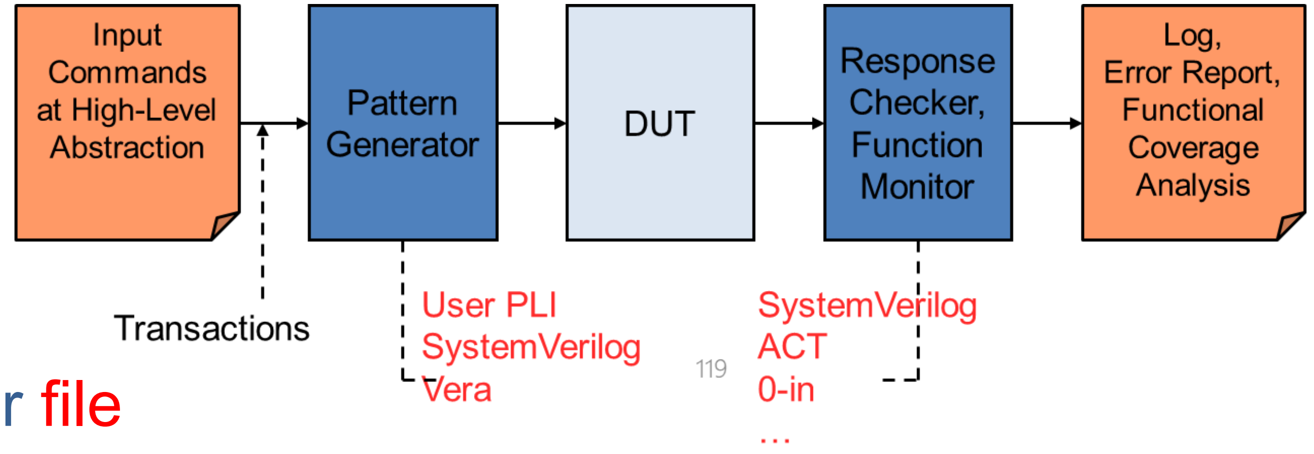
- Overview of design verification
- Testbench example
- Information display and file I/O

Overview

- HDL that tests another module: device under test (dut)

You will need to

- Instantiate DUT
- Generate **test pattern**
  - Clock and reset signals
  - Input signals by manual design or file
  - Prefer self auto or semi-auto stimulus generation
- **Check responses** if meet the golden answer
  - Compare DUT output with golden data
  - Golden data either by manual design or file
  - Prefer self automatic response checking


Input Commands at High-Level Abstraction
Pattern Generator
DUT
Response Checker, Function Monitor
Log, Error Report, Functional Coverage Analysis
Transactions
User PLI
SystemVerilog
Vera
SystemVerilog
ACT
0-in
…

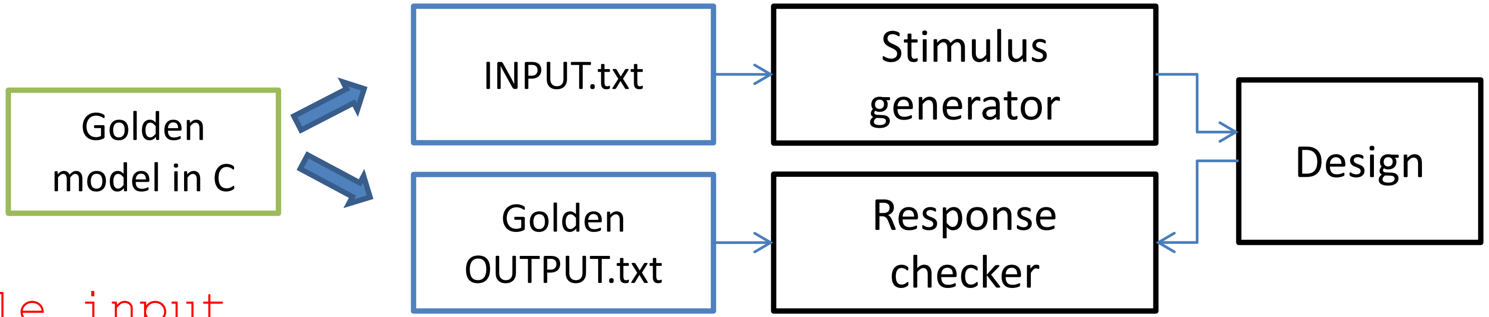
How to Start Your Own Testbench

- Start with the testbench template examples
- Basic one:

```
//    CLOCK GENERATION
//    RESET
//    INITIALIZATION
//    SEND STIMULUS &  MAIN FLOW
//    CHECK RESPONSE
//        no reset outputs
//        wrong answer
```

How to Start Your Own Testbench

- Best one for complex design
  - File I/O with behavior model
  - Example



```
initial begin  //golden file input
    file2  = $fopen("input.txt","r");
end
for(j=0;j<input_num[i];j=j+1) begin
      rc=$fscanf(file2,"%d",data_in[j]);
end
@(negedge CLK); //automatic checking
    for(k=1;k<(input_num[i]+1);k=k+1) begin
        if(OUT!==golden_ans[k]) begin
         Error = 1'b1;
         your_out[k-1] = OUT;
          $display (" Error !! OUTPUT IS WRONG!   ");
        ......................
```

TESTBENCH EXAMPLE

Testbenches Coding

- Not synthesizeable, use high level commands to facilitate this
  - initial block
  - Delay for n units of time
  - Full high-level constructs: if, while, sequential assignment.
  - Input/output: file I/O, output to display, etc.
- Test pattern
  - Manual design
  - From files
  - Output check
    - Self checking

I. A Simple Test Bench

```
`timescale 1ns/1ps
module test_bench;
// Interface to communicate with the DUT
reg a, b, clk;
wire c;
// Device under test instantiation
DUT U1 (.in1(a), .in2(b), .clk(clk), .out1(c));
initial
begin  // Test program
  test1 (); //call input pattern task
  $finish; //terminate simulation
end
initial
begin
  clk = 0;
  forever #5 clk = ~clk;
end
initial
begin // Monitor the simulation
  $dumpvars;
  $display ("clk | in1| in2 | out1 |");
  $monitor ("  %b| %b |  %b |   %b |",clk, a, b, c);
end
endmodule
```

```
module DUT (in1, in2, clk, out1);
input in1, in2;
input clk;
output reg out1;
always @(posedge clk)
out1 = in1^in2;
endmodule
```

```
task test1 ();
begin
      a = 0;   b = 0;
  #10 a = 0;   b = 1;
  #10 a = 1;   b = 1;
  #10 a = 1;   b = 0;
end
endtask
```

II. Another Testbench Example

- Write SystemVerilog code to implement the following function in hardware:

$$y = \overline{b}\,\overline{c} + a\overline{b}$$

- Name the module `sillyfunction`

II.

- Write SystemVerilog code to implement the following function in hardware:

$$y = \overline{b}\,\overline{c} + a\overline{b}$$

```
module sillyfunction(input  logic a, b, c,
                     output logic y);
  assign y = ~b & ~c | a & ~b;
endmodule
```

II.A Simple Testbench by Manual Design Pattern (手工測資)

```
module testbench1();
  logic a, b, c;
  logic y;
  // instantiate device under test
  sillyfunction dut(a, b, c, y);
  // apply inputs one at a time
  initial begin
    a = 0; b = 0; c = 0; #10;
    c = 1; #10;
    b = 1; c = 0; #10;
    c = 1; #10;
    a = 1; b = 0; c = 0; #10;
    c = 1; #10;
    b = 1; c = 0; #10;
    c = 1; #10;
    $finish; //terminate the simulation
  end
endmodule
```

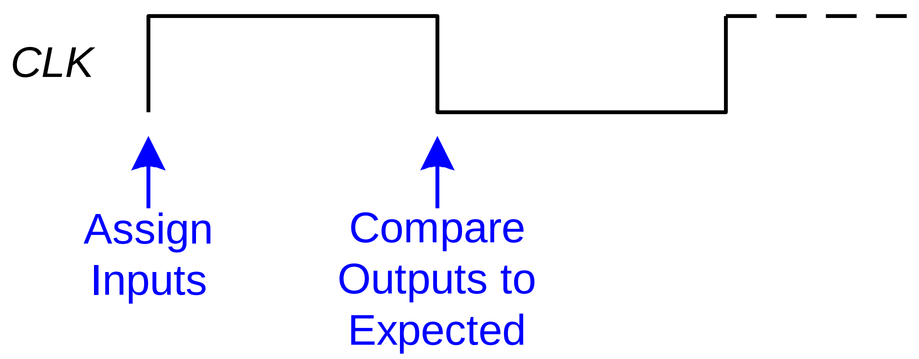
II.B Self-checking Testbench

```
module testbench2();
  logic  a, b, c;
  logic y;
  sillyfunction dut(a, b, c, y);  // instantiate dut
  initial begin // apply inputs, check results one at a time
    a = 0; b = 0; c = 0; #10;
    if (y !== 1) $display(“000 failed.”); //每個都比對一下正確答案
    c = 1; #10;
    if (y !== 0) $display("001 failed.");
    b = 1; c = 0; #10;
    if (y !== 0) $display("010 failed.");
    c = 1; #10;
    if (y !== 0) $display("011 failed.");
    a = 1; b = 0; c = 0; #10;
    if (y !== 1) $display("100 failed.");
    c = 1; #10;
    if (y !== 1) $display("101 failed.");
    b = 1; c = 0; #10;
    if (y !== 0) $display("110 failed.");
    c = 1; #10;
    if (y !== 0) $display("111 failed.");
    $finish;
  end
endmodule
```

III. Testbench with Testvectors 測資從檔案來

- Testvector file: inputs and expected outputs
- Testbench:
  1. Generate clock for assigning inputs, reading outputs
  2. Read testvectors file into array
  3. Assign inputs, expected outputs
  4. Compare outputs with expected outputs and report errors

- Testbench clock:
  - assign inputs (on rising edge)
  - compare outputs with expected outputs (on falling edge).



- Testbench clock also used as clock for synchronous sequential circuits

Testvectors File

- File: example.tv
- contains vectors of abc_yexpected

```
000_1
001_0
010_0
011_0
100_1
101_1
110_0
111_0
```

1. Generate Clock

```
module testbench3();
  logic          clk, reset;
  logic          a, b, c, yexpected;
  logic          y;
  logic [31:0] vectornum, errors;    // bookkeeping variables
  logic [3:0]  testvectors[10000:0]; // array of testvectors

  // instantiate device under test
  sillyfunction dut(a, b, c, y);

  // generate clock
  always     // no sensitivity list, so it always executes
    begin
      clk = 1; #5; clk = 0; #5;
    end
```

2. Read Testvectors into Array

```
// at start of test, load vectors and pulse reset

initial
   begin
     $readmemb("example.tv", testvectors);
     vectornum = 0; errors = 0;
     reset = 1;
     #27;
     reset = 0;
   end



// Note: $readmemh reads testvector files written in
// hexadecimal
```

3. Assign Inputs & Expected Outputs

```
// apply test vectors on rising edge of clk
always @(posedge clk)
  begin
    #1; {a, b, c, yexpected} = testvectors[vectornum];
  end
```

input

Golden data

4. Compare with Expected Outputs

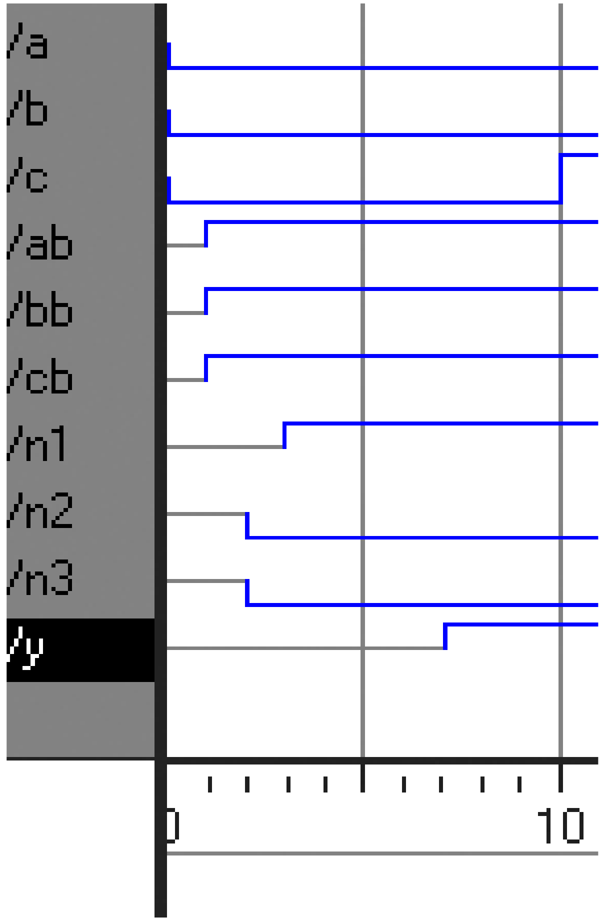
```
// check results on falling edge of clk
   always @(negedge clk)
    if (~reset) begin // skip during reset
      if (y !== yexpected) begin
        $display("Error: inputs = %b", {a, b, c});
        $display("  outputs = %b (%b expected)",y,yexpected);
        errors = errors + 1;
      end

// Note: to print in hexadecimal, use %h. For example,
//       $display("Error: inputs = %h", {a, b, c});
```

4. Compare with Expected Outputs

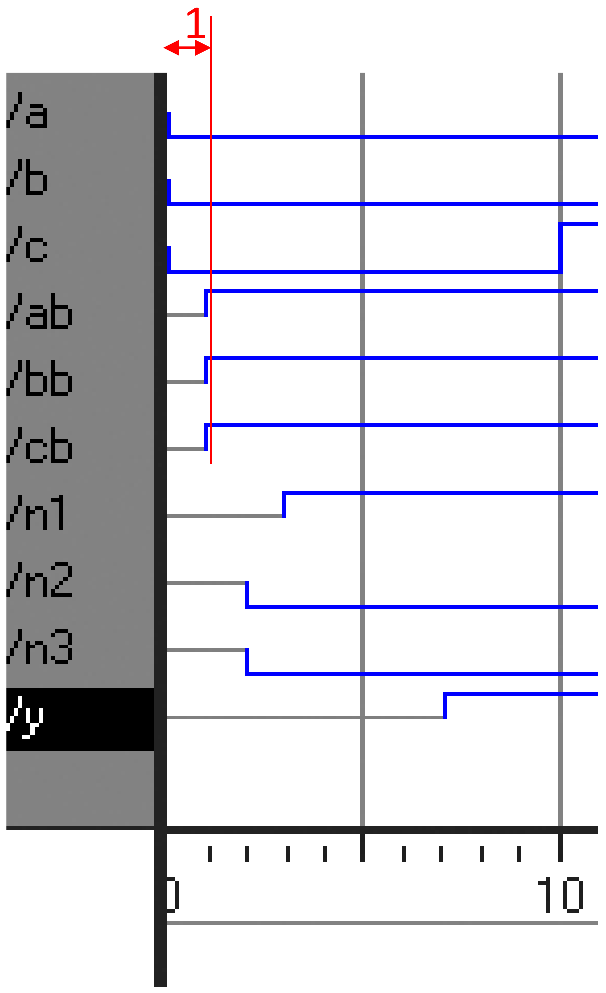
```
// increment array index and read next testvector
      vectornum = vectornum + 1;
      if (testvectors[vectornum] === 4'bx) begin
          $display("%d tests completed with %d errors",
                vectornum, errors);
        $finish;
      end
    end //end of always
endmodule


// === and !== can compare values that are 1, 0, x, or z.
```

Delays

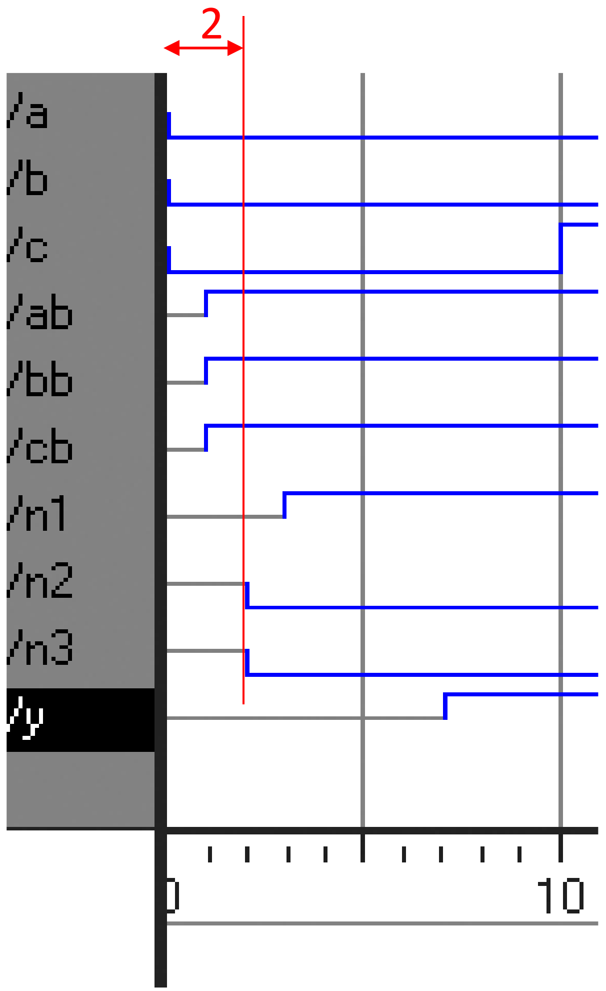
```
module example(input  logic a, b, c,
               output logic y);
  logic ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} =
                  ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```



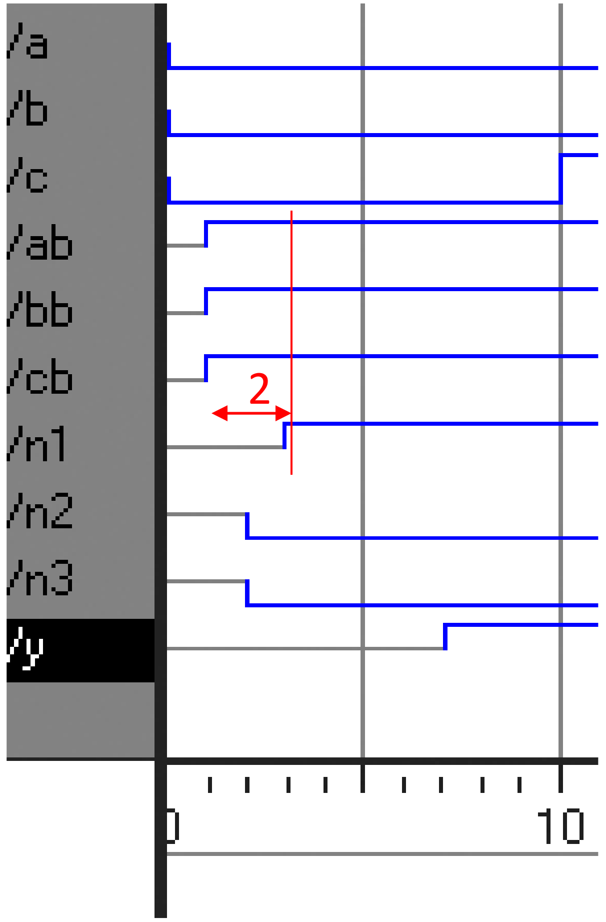
```
module example(input  logic a, b, c,
                      output logic y);
  logic ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} =
                          ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```



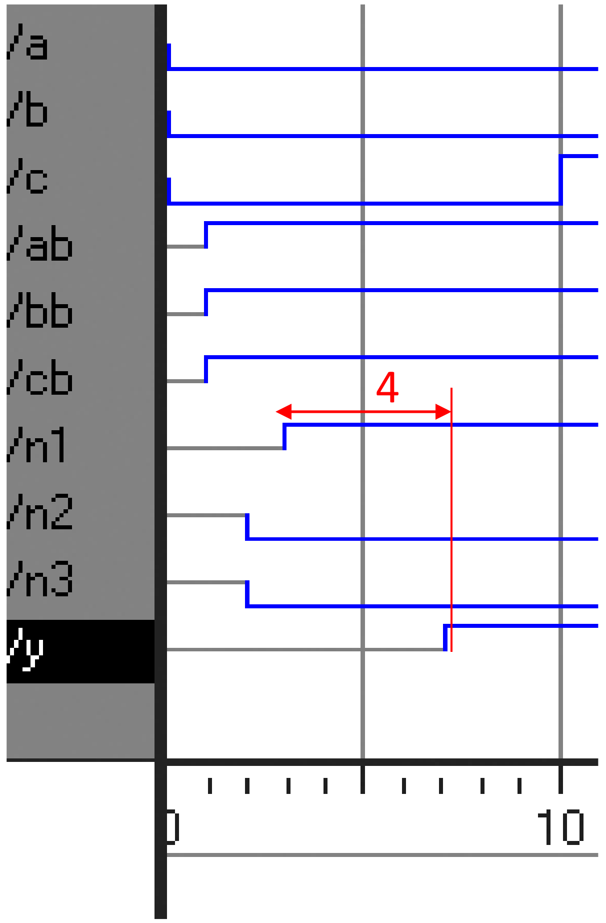
```
module example(input  logic a, b, c,
                      output logic y);
  logic ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} =
                        ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```



```
module example(input  logic a, b, c,
               output logic y);
  logic ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} =
                 ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```



```
module example(input  logic a, b, c,
               output logic y);
  logic ab, bb, cb, n1, n2, n3;
  assign #1 {ab, bb, cb} =
                ~{a, b, c};
  assign #2 n1 = ab & bb & cb;
  assign #2 n2 = a & bb & cb;
  assign #2 n3 = a & bb & c;
  assign #4 y = n1 | n2 | n3;
endmodule
```



IV. Simple testbench (SystemVerilog Version)

```
`timescale 1ns/1ps //time resolution

`include "design.v"
`include "pattern.v"

module testbench;
// inter connection wire
wire       clk_p;
wire       rst_n;
wire [7:0] data_in;
wire [7:0] data_out;
// connect test pattern
pattern U_pattern(
  .*);
//my test
design U_design(
  .*));
// dumpping waveform
initial begin
  $fsdbDumpfile("waveform.fsdb");
  $fsdbDumpvars;
end
endmodule
```

Defines the time units and simulation precision (smallest increment)

`timescale <reference_time_unit> / <time_precision>

- reference_time_unit: simulation time unit
- time_precision: unit for rounding

The precision unit must be less than or equal to the time unit

```
`ifdef RTL
     `timescale 1ns/100ps
`endif
//finer resolution for gate level
`ifdef GATE
     `timescale 1ns/10ps
`endif

//global parameters for all modules
`define CLK_PERIOD 30.0
```

```
//---------------------------------------------------
//    CLK DECLARATION
//----------------------------------------------------
logic clk;
parameter cycle = 15;

initial begin
        clk=0;
        forever #(cycle/2.0) clk = ~clk;
end
```

Another coding style

```
always    // no sensitivity list, so
          //it always executes
    begin
      clk = 1; #5; clk = 0; #5;
    end
```

Another coding style

```
module testbench;
...
logic clk = 0;
parameter cycle = 15;
always
begin
  #(cycle/2.0); // if int division => 7
  clk =~clk;
end
endmodule
```

Note. task

- A task: pack common operations into one
  - Defined in module, not synthesizable
  - Can be called by any procedural block.
  - Has n input, m output.
  - Can include timing control like wait, @, #
- Ex

```
// calling
increase(A, B);

// declaration
task increase;
  input  [4:0] b;
  output [4:0] a;
begin
  a = #5 b + 1;
end
endtask
```

Note. function

- A function: pack common **combinational operations** into one
  - Defined in module, synthesizable
  - Can be called by function but not task
  - Has n inputs, 1 output
  - Cannot include timing control.
  - Execute no delay.
- Example

```
// calling
C = increase(B);
```

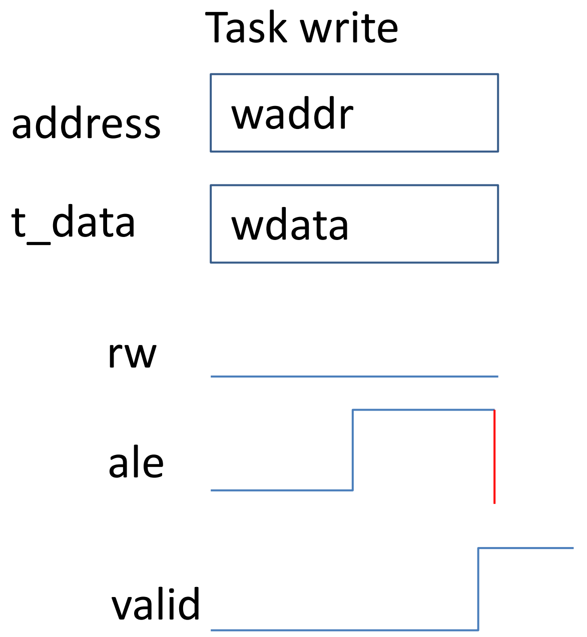
output

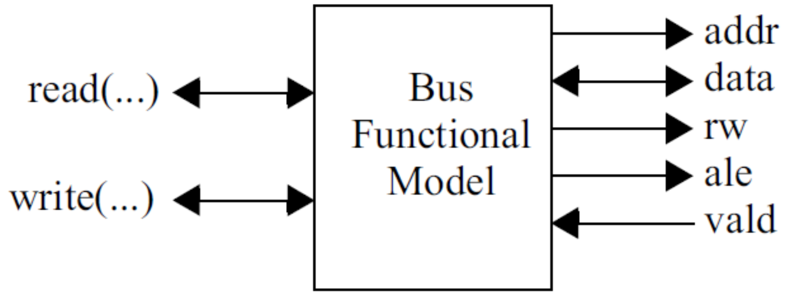
```
// declaration
function increase;
  input  [4:0] b;
begin
  increase = b + 1;
end
endfunction
```

Bus Functional Model

```
module TESTBED();
     reg  [5:0] addr;
     wire [3:0] data;
     reg  [3:0] t_data;
     reg   [3:0] rdata;// store read data
     reg rw;//0:write,1:read
     reg ale;// use bus as address or data
     wire valid;

task write;
    input [5:0] waddr;
    input [3:0] wdata;
    input valid;
    output [3:0] t_data;// task data
    output rw,  ale;
begin
    if(valid != 0)
      wait(valid==0);

    addr = waddr;
    t_data  = wdata;
    rw = 0;
    #(10); // after 1 cycle
     ale  =1;  // pattern send the reguest
    wait(valid ==1); // design ack
    ale =0;
end
endtask
```

```
task read;
    input [5:0] raddr;
    output [3:0] rdata;
    input valid;
    input [3:0] data;
    output rw,  ale;
begin
    if(valid != 0)
      wait(valid==0);

    addr = raddr;
    rw = 1;
    #(10); // after 1 cycle
     ale  =1;  // pattern send the reguest
    wait(valid ==1); // design ack
    rdata  = data;
    ale =0;
end
endtask
```





```
task write;
    input [5:0] waddr;
    input [3:0] wdata;
    input valid;
    output [3:0] t_data;// task data
    output rw,  ale;
```

```
task read;
    input [5:0] raddr;
    output [3:0] rdata;
    input valid;
    input [3:0] data;
    output rw,  ale;
```

```
Bus_module I0(addr, data, rw, ale, valid);
assign data = (!rw)?t_data:data;

initial begin
   addr = 0;
   rdata =0;
   t_data =0;
   rw =  1;
   ale =0;

    write(6'd0, 4'd3, valid, t_data, rw, ale); //write 4'd3 to address 0
    read (6'd0, rdata, valid, t_data, rw, ale);// rdata read
                                               //value 3 from
                                              //address 0

end
endmodule
```

TLDR: Task and Function

- Model reusable code
- Task
  - Behavior modeling=> allow delay or timing control.
  - Zero or more input, output, INOUT (Positional mapping)
  - Enable or disable task
  - Input values are passed into the task only once, so signals used in timing controls (such as *clk*) must not be inputs to the task
- Function
  - Pure computation, or combinational logics. => no delay or timing control
  - One output and at least one input (Positional mapping)
    - Tricky way: cascade for more output value
    - {o1,o2,o3,o4} = my_func (a,b,c,d,e);

Repeat

```
repeat (10) begin
//重複執行10次
end

repeat (10) @(posedge clk);
// equivalent to #(10*cycle);
```

INFORMATION DISPLAY AND FILE I/O

Information Display

- Display system task (once per call): display data to monitor
  - $display ,$write, and $strobe
  - Similar to printf in C
- Continuous display if signal changes (for text only era)
  - $monitor
  - Outdated with signal dump with *fsdbdumpvars for waveform*
- File I/O
  - Very primitive for Verilog-1995
  - Whole file read/write to/from memory: $readmemh, $writememh
  - $fdisplay.., very limited for file input
  - Use Verilog-2001 file I/O style (basically equal to C file I/O)

$display $monitor (text mode debug)

- Similar to printf in C
- $display (display once when executed once)
- $display automatically prints a new line to the end of its output
  - $display ([“format_specifiers”,] <argument_list>);
  - $display("Hello world")
  - $display ($time,,“%b %h %d %o”, sig1, sig2, sig3, sig4);
    - 0 6 7 8 9
- $monitor (always display if being executed)
  - $monitor ($time,,“%b %h %d %o”, sig1, sig2, sig3, sig4);
    - 0 6 7 8 9
    - 1 7 5 4 2
    - $monitoron, $monitoroff, turn on or off monitor

Display Information (cont.)

- The following escape sequences are used for display special characters, similar to C language

\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		

Default display mode: $display: decimal, $displayb:binary, $displayo: octal,$displayh: hexdecimal

Apply to $monitor, too

File I/O (Similar to C)

```
integer in,out,mon;

initial begin
  in  = $fopen("input.txt","r");
  out = $fopen("output.txt","r");
  mon = $fopen("monitor.txt","w");
end

// DUT input driver code
initial begin
    repeat (10) @ (posedge clk);
    while (!$feof(in)) begin
      @ (negedge clk);
      statusI = $fscanf(in,"%h %h\n",din[31:16],din[15:0]);
    end
```

http://www.asic-world.com/verilog/verilog2k3.html

```
    repeat (10) @ (posedge clk);
    $fclose(in);
    $fclose(out);
    $fclose(mon);
    #100 $finish;
end
// DUT output monitor and compare logic
always @ (posedge clk)
 if (valid) begin
   $fwrite(mon,"%h %h\n",dout[31:16],dout[15:0]); //no fprintf in SystemVerilog
   status0 = $fscanf(out,"%h %h\n",exp[31:16],exp[15:0]);
```

Close files

$fwrite() similar to fprintf()

$fwrite does not insert a newline at the end. $fdisplay does.

$fdisplay(file, "Hello World"); is the same as $fwrite(file, "Hello World\n");

Enhanced File I/O and String Tasks in Verilog-2001 (Similar to C)

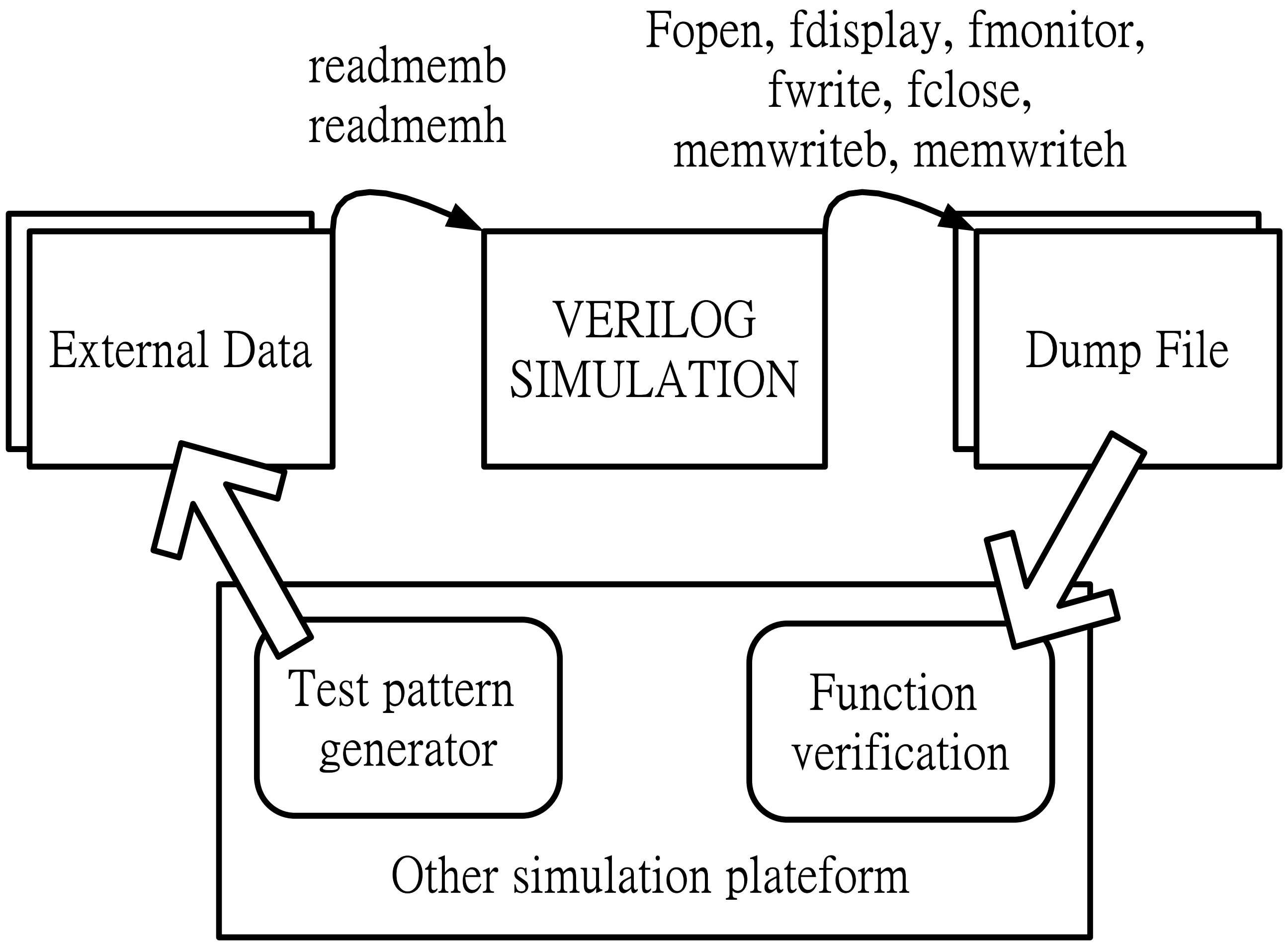
* Open up to 2^{30} files

* New built-in file I/O tasks:
$fgetc, $ungetc, $fgets
$fscanf, $fread,
$ftell, $fseek, $rewind, $fflush
$ferror

* New string manipulation tasks:
$swrite, $swriteb, $swriteh, $swriteo, $sformat
$sscanf

Verilog-2001

Verilog Specific I/O



$readmemh $readmemb read array

256 words x 8 bits ROM

rom.txt

```
//Comments are allowed
1100_1100   // This is first address i.e 8'h00
1010_1010   // This is second address i.e 8'h01
@ 55        // Jump to new address 8'h55
0101_1010   // This is address 8'h55
0110_1001   // This is address 8'h56
```

```
module romTest;
reg [7:0]  m [0:255];

reg t = 0;
wire o;

rom DUT (.i(t), .o(o));

initial $readmemb("rom.txt", m);

integer k;

initial begin
    $display("Contents of Mem after reading data file:");
    for (k=0; k<12; k=k+1) $display("%d:%h",k,m[k]);
end

endmodule
```

$writememh $writememb write array

```
module TEST;
parameter CYCLE_NEEDED = 5000;
reg [7:0]     MEMORY[0:1023];
reg    CLK;
integer       FILE1, i;
real   CYCLE;
always #(CYCLE/2) CLK=~CLK;
……
initial begin
    CYCLE = 10;
    #(CYCLE_NEEDED*CYCLE)
        $writememb (“DATA.txt”, MEMORY); //write whole memory to file
end
endmodule
```

$random

- Random number generation
- Returns as 32-bit signed integer

```
reg [31:0] rand1, rand2, rand3;
rand1 = $random;        // generates random numbers
rand2 = $random % 60;   // random numbers between -59 and 59
rand3 = {$random} % 60;// random positive values
                        // between 0 and 59
```

- $urandom_range()
  - 語法：$urandom_range(int unsigned maxval,int unsigned minval = 0);
  - 功能：傳回一個在maxval和minval之間的無符號整數

```
val = $urandom_range(7,0);
```

- **$urandom()**
  - 語法：$urandom[(int seed)];
  - 功能：產生偽亂數，每次調用時返回一個32位元的無符號偽亂數；

```
addr[32:1] = $urandom(254);//初始化生成器,獲得一個32位的亂數
addr = {$urandom,$urandom};//64位亂數
number = $urandom & 15;//4位亂數
```