Introduction to Verilog HDL

Behavioral description of logic circuits with Verilog HDL (IEEE1364)

1. Overview

Steps of Learning HDL

- Goal of this class
  - Understanding a description for logic synthesis
  - Understanding a description for simulation

- Goal of Integrated Circuit Design and Practice
  - Understanding an architecture of microprocessors with HDL including
    - BUS
    - Memory
    - Instruction set
    - Various arithmetic algorithm
    - Parallel processing and pipeline control

- Moreover…
  - Usage of intellectual properties (IPs)

References in Japanese

- 深山正幸他 「HDLによるVLSI設計」第2版, 共立出版 ISBN 4-320-12027-2
- You can find many references on web sites, for example,
  - http://cas.eedept.kobe-u.ac.jp/~arai/Verilog/chap5.html
  - https://furien.jp/columns/303/

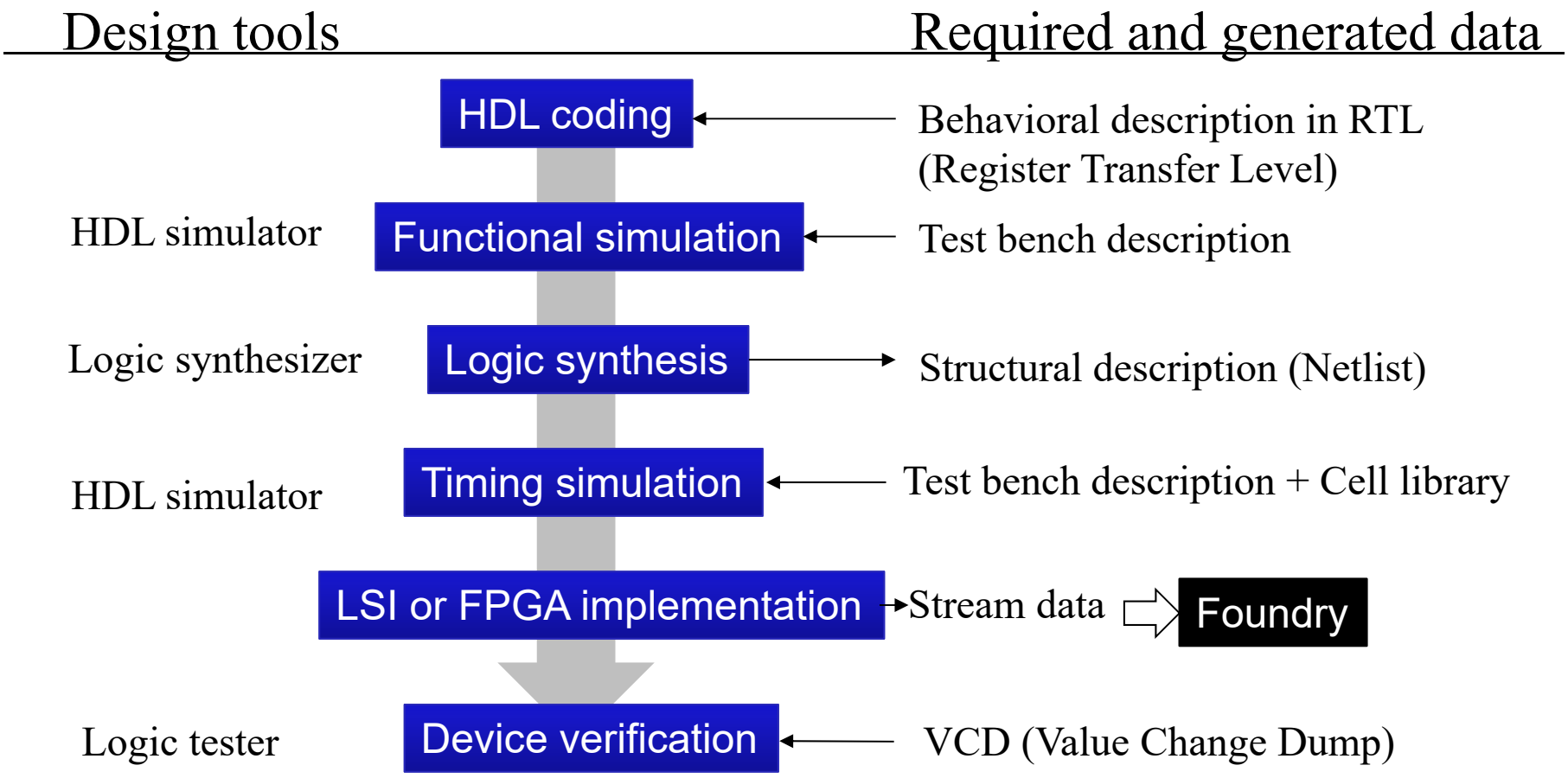
HDL Simulator

- ModelSim - Intel FPGA Edition (Lite)

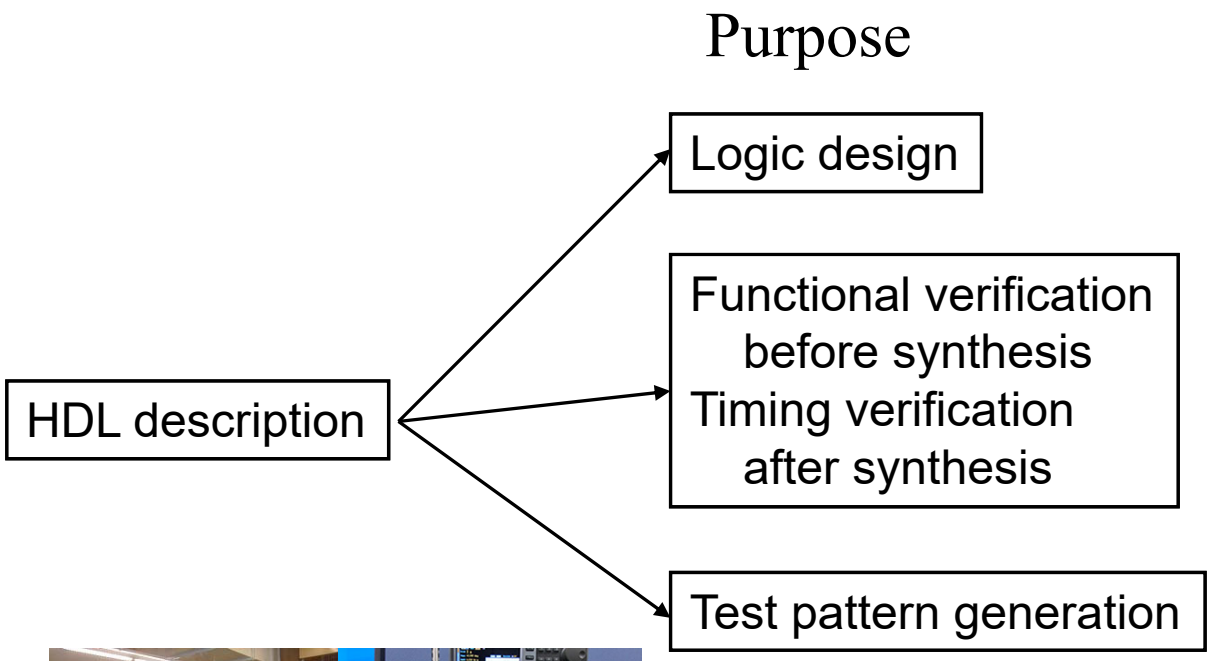
https://www.intel.co.jp/content/www/jp/ja/software/programmable/quartus-prime/model-sim.html

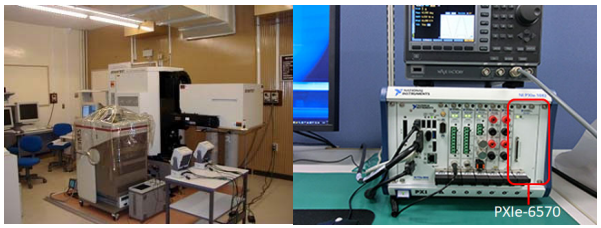
- See Appendix 1a and 1b.

LSI/FPGA design flow with HDL



The purpose of HDL description



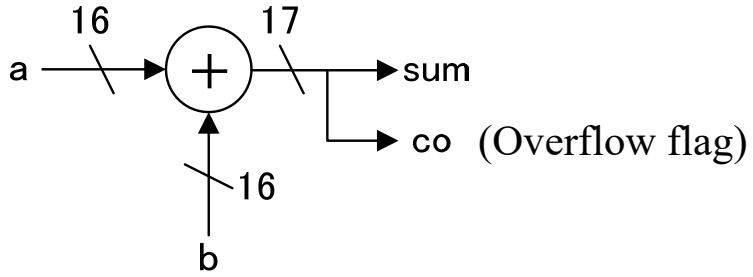


Logic tester in Kanazawa University: http://jaco.ec.t.kanazawa-u.ac.jp/edu/vlsi/ni6570/

2. HDL description in RTL

RTL: register transfer level
means the level of detail which describes the digital processing and controlling the signal flow for each clock edge. The logic circuit can be absolutely synthesize from the description.

Combinational logic with assign statement



16bit Adder

```
module adder16 (a, b, sum, co);
  // 16 bit Adder
  input    [15:0] a, b;
  output   [15:0] sum;
  output          co;

  wire     [15:0] a, b, sum;
  wire            co;

  assign  {co, sum} = a + b;

endmodule
```

Module declaration

// Comment

Port declaration
(input | output) [MSB:LSB]
Note: [MSB:LSB] is optional for 1bit port

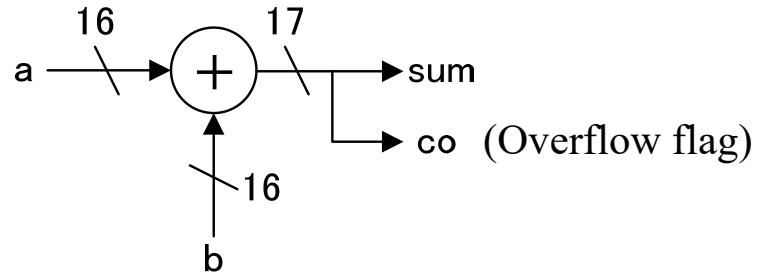
Type declaration
Note: You can omit the type declaration of wire.

An assign statement for logic value of wire
Note: The assignment statements are concurrently executed when the value of the right side is updated.
{high-order bits, low-order bits} is a concatenation operator.

No ";" after endmodule

The end of statement

Combinational logic with always statement



16bit Adder

```
module adder16 (a, b, sum, co);
   input     [15:0] a, b;
   output    [15:0] sum;
   output           co;

   wire      [15:0] a, b;
   reg       [15:0] sum;
   reg              co;

   always @(a, b) begin
      {co, sum} <= a + b;
   end
endmodule
```

Module declaration

Port declaration

Type declaration of reg (register) and wire

The always-block is started when the value in @(variables) is updated.
The multiple always-block works simultaneously.

An procedural assignment for logic value of reg.
Note: All "<=" statements in a always-block are executed when the always statement is asserted. An "<=" statement is also called non-blocking assignment.

No ";" for end and endmodule

Assignment and variable types in RTL

Statement	Type restriction	Location	Execution timing
assign a = b;	left side : wire right side : reg or wire	outside of always block	When the right side is updated.
a <= b; (non-blocking assignment)	left side: reg right side : reg or wire	inside of always block	When the always block is asserted.

Note: Normally, the register (D-FF) is synthesized by the non-blocking assignment to the reg variable, however, the always block which does not synchronize the clock does not generate the register.

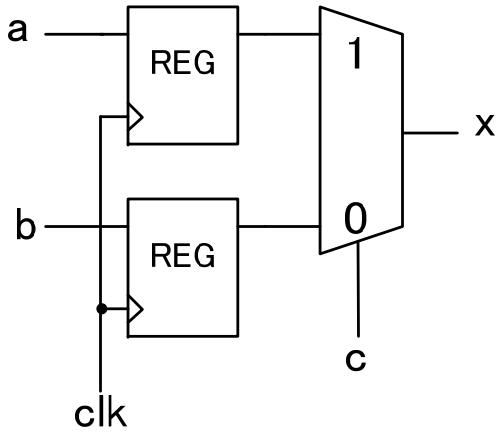
Conditional assign statement

Verilog HDL

```
assign x = (c == 1'b1) ? a : b;
```

C

```
if (c == 1) x = a;
else x =b;
```



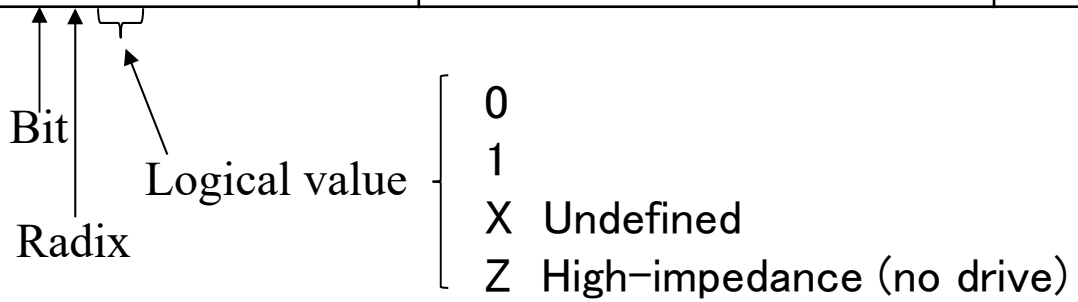
Note: The non-blocking assignment is normally used to describe the functions of the information processing. The assign statement is useful to output the value of the registers from the output port.

Variable types in RTL

Type declaration	Application	Remarks
reg a; reg [15:0] a;	Logic variable of the register	left side of <=
wire a; wire [15:0] a;	Logic variable of the wire The declaration of wire can be omitted.	left side of assign =
integer a;	Do not use the integer for the logic variables. This type is dedicated to the bit number or loop counter.	Do not assign to reg and wire.

Constant and Literal

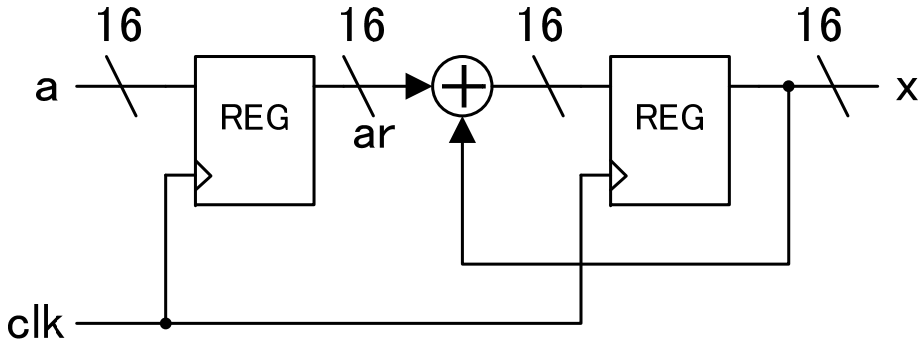
Statement	Description	Remarks
parameter a = 0;	Declaration of constant	A type of parameter depends on the initial number.
a <= 8'b00001111;	Binary number	
a <= 8'd15;	Decimal number	
a <= 8'h0f;	Hexadecimal number	



Sequential logic with always statement

16bit Accumulator

```
module integrator16 (a, clk, x);
   input            clk;
   input    [15:0] a;
   output   [15:0] x;

   wire             clk;
   wire     [15:0] a;
   reg      [15:0] ar, x;

   always @(posedge clk) begin
      ar <= a;
      x <= ar + x;
   end
endmodule
```



The always-block is started in synchronization with the edge of the clk signal. The rise edge and fall edge is specified by using the keyword "posedge" or "negedge".

An assign statement for logic value of reg. A register circuit is synthesized by the edge sensitive non-blocking assignment.
Note: All "<=" statements in a always-block are executed when the always statement is asserted.

The multiple "<=" is simultaneously executed. Thus, the value of the register "ar" in 2nd line is different from the 1st one. This is really different from the programing language See next slide.

Non-blocking assignment in always block

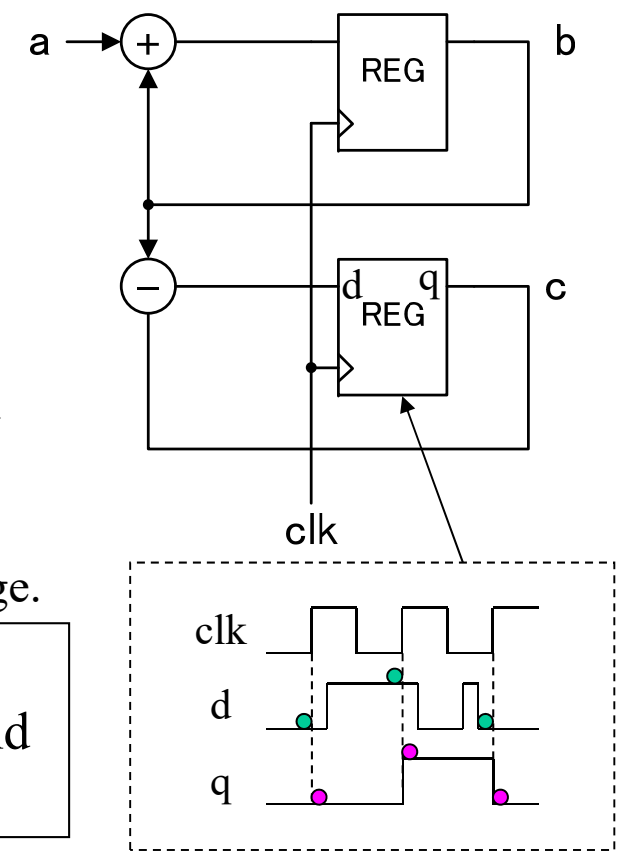
Synchronized assignment

```
always @(posedge clk) begin
    b <= a + b ;
    c <= b – c ;
end
```

after clock edge

before clock edge
(This value is calculated by last edge.)

Expected circuit for the description



The result is different from a procedural language.

If a = 1, b = 1, and c = 1 before the clock edge, b and c on the left is updated b = 2 and c = 0 after the clock edge.

Truth table with case statement

3bit Decoder

```
module dec3 (address, word);
   input     [2:0] address;
   output    [7:0] word;

   wire      [2:0] address;
   reg       [7:0] word;

   always @(address) begin
      case (address)
         3'b000 : word <= 8'b00000001;
         3'b001 : word <= 8'b00000010;
         3'b010 : word <= 8'b00000100;
         3'b011 : word <= 8'b00001000;
         3'b100 : word <= 8'b00010000;
         3'b101 : word <= 8'b00100000;
         3'b110 : word <= 8'b01000000;
         3'b111:  word <= 8'b10000000;
         default : word <= 8'bxxxxxxxx;
      endcase
   end
endmodule
```

3'b 3bit binary number
4'h0 4bit hexadecimal number

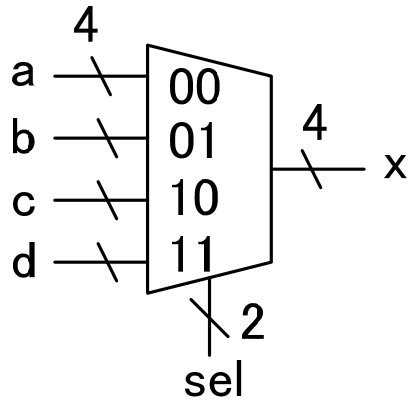
A "default" assignment is recommended in a case statement.

address	word
000	00000001
001	00000010
010	00000100
011	00001000
100	00010000
101	00100000
110	01000000
111	10000000

Conditional branch with case statement

4bit MUX

```
module mux4 (a, b, c, d, sel, x);
   input     [3:0] a, b, c, d;
   input     [1:0] sel;
   output    [3:0] x;

   wire      [3:0] a, b, c, d;
   wire      [1:0] sel;
   reg       [3:0] x;

   always @(a or b or c or d or sel) begin
      case (sel)
         2'b00 : x <= a;
         2'b01 : x <= b;
         2'b10 : x <= c;
         2'b11 : x <= d;
         default : x <= 4'bxxxx;
      endcase
   end
endmodule
```



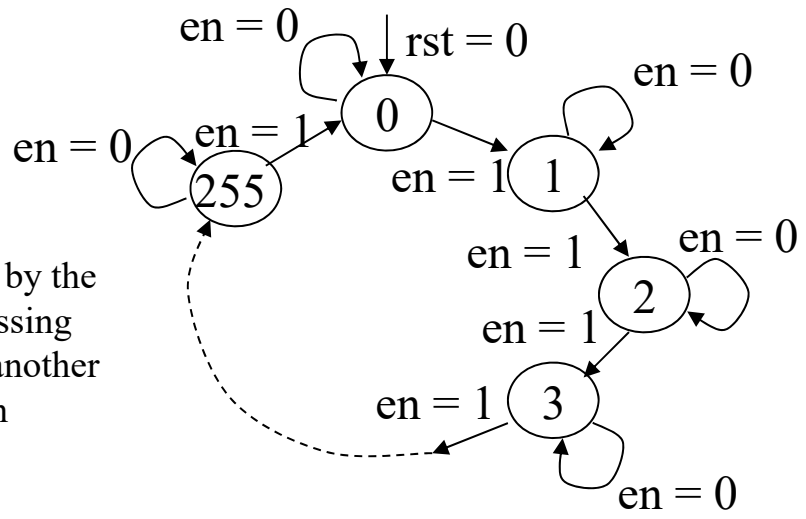
State transition with if-else statement

8bit Binary counter

```
module count8 (clk, rst, en, q, co);
   input          clk, rst, en;
   output  [7:0] q;
   output         co;

   wire           clk, rst;
   wire           co;
   reg     [7:0] q;

   always @(posedge clk) begin
      if (rst == 1'b0)
         q <= 8'b00000000;
      else
         q <= q + en;
   end

   assign  co = &q;

endmodule
```

State transition driven by the clock edge. The processing contents is written in another always-block or assign statement.

Synchronous reset



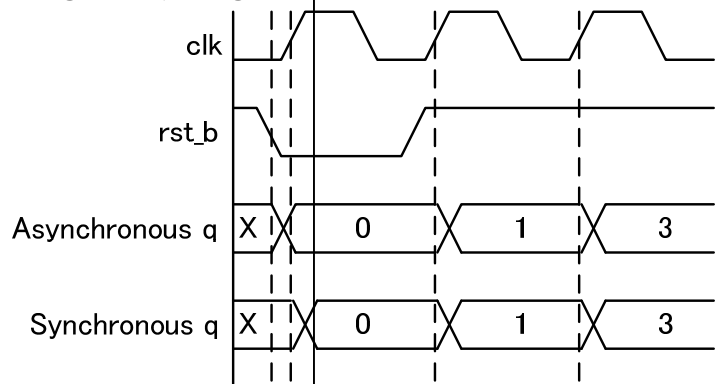
Processing content:
Unary operation "&" performs AND operation for all digit.
Thus, if q = 255, then the "1" is outputted from co port.

Asynchronous reset

4bit Johnson counter

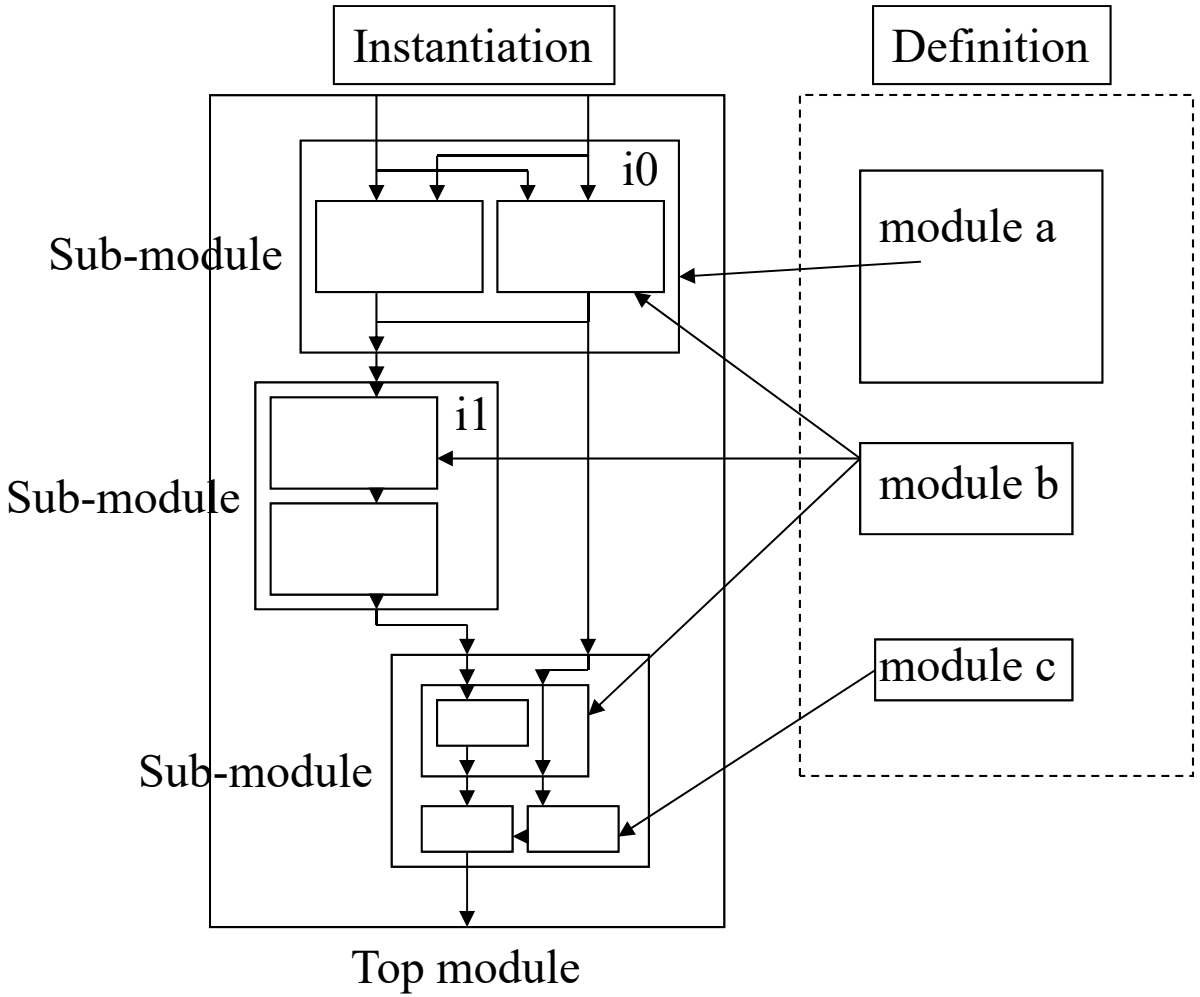
```
module jcnt4 (clk, rst_b, pr_b, q);
   input          clk, rst_b, pr_b;
   output  [3:0] q;
   reg    [3:0] q;

   always @(negedge rst_b or negedge pr_b or posedge clk) begin
      if (!rst_b) begin
         q <= 4'h0;
      end
      else if (!pr_b) begin
         q <= 4'hf;
      end
      else begin
         q <= q << 1;
         q[0] <= ~q[3];
      end
   end
endmodule
```

← Asynchronous reset



0000 → 0001 → 0011 → 0111 → 1111 → 1110 → 1100 → 1000 → 1000

Hierarchically organized description



Verilog HDL

Module definition

```
module a(in, out);
```

Instance call

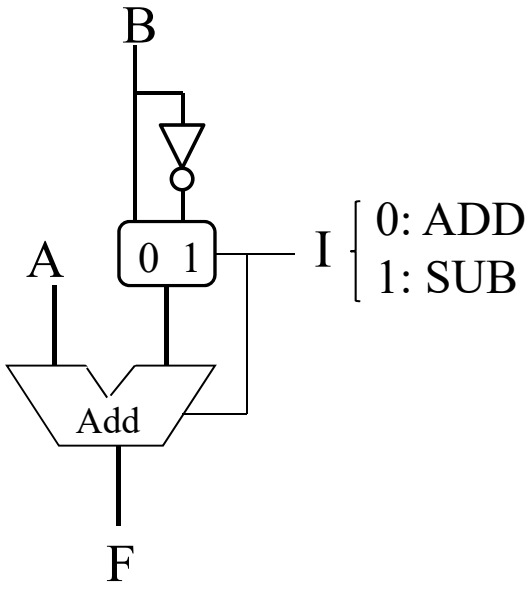
```
a i0 (.in(w1), .out(w2));
```

Port connection

Instance name

Definition name

Structured description and instantiation




```
module sub(A, B, I, F);
   input    [15:0]    A, B;
   input              I;
   output   [16:0]    F;

   wire     [15:0]    A, B;
   wire               I;
   reg      [16:0]    F;
   reg      [15:0]    CMP;

   add16  i0(.a(A), .b(CMP), .ci(I), .sum(F));

   always @(B, I) begin
      if(I == 1'b0)
         CMP <= B;
      else
         CMP <= ~B;
   end
endmodule
```

Call

```
module add16 (a, b, ci, sum);
   input    [15:0]    a, b;
   input              ci;
   output   [16:0]    sum;

   wire     [15:0]    a, b;
   wire               ci;
   reg      [16:0]    sum;

   always @(a, b, ci) begin
      sum <= a + b + ci;
   end
endmodule
```

4. HDL description of test bench

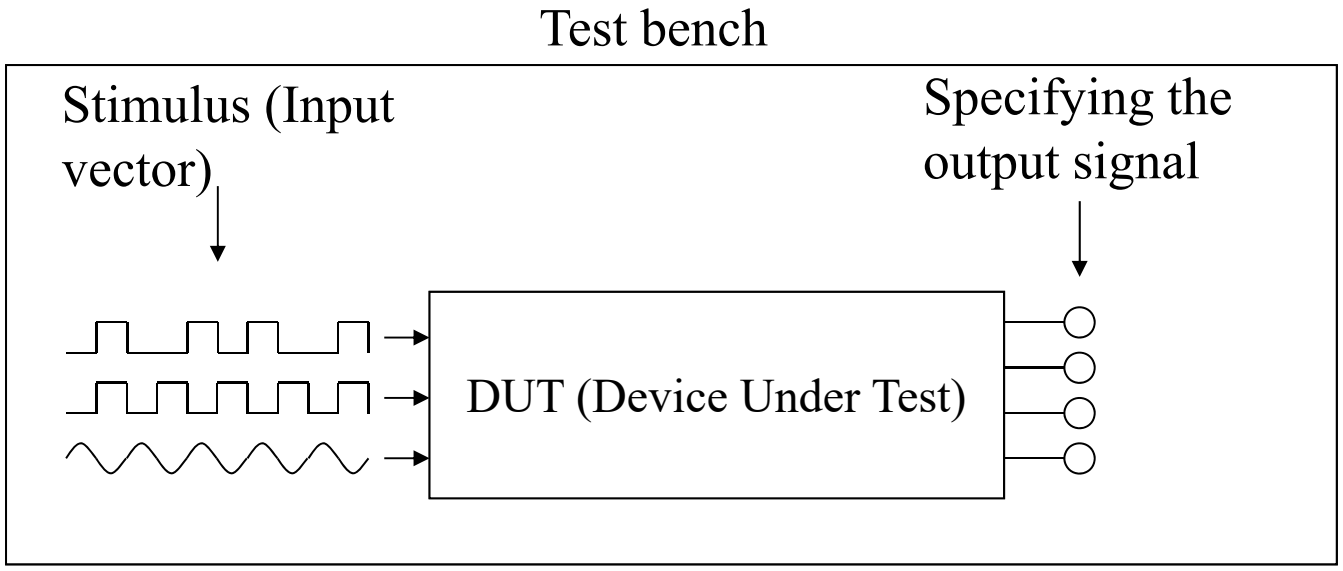
The test bench includes the DUT (Device Under Test) or UUT(Unit Under Test), the sequence of the input vector, and the directive to the simulator, which can be described by HDL code in any level of detail.

HDL Simulation

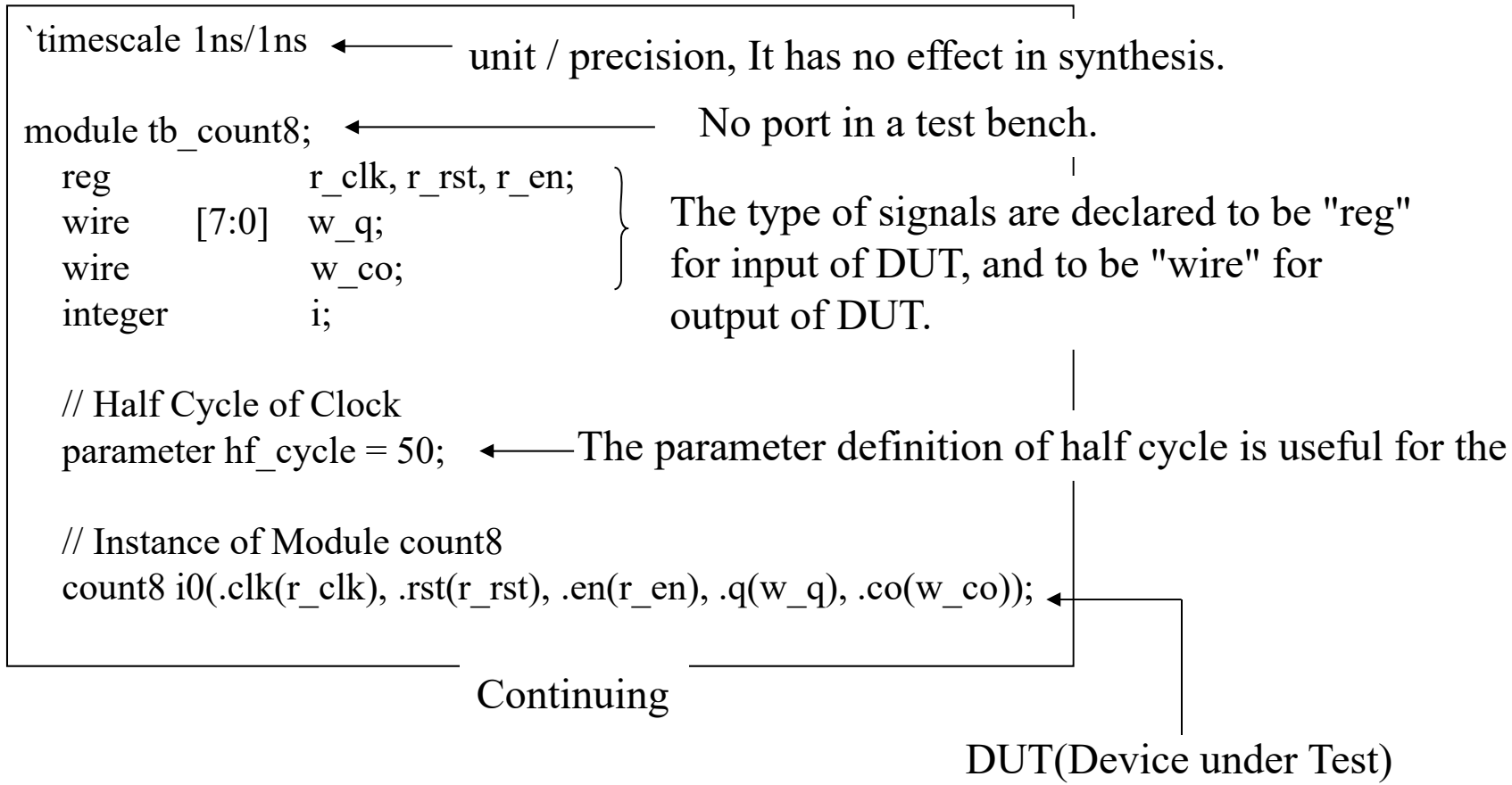
- Functional simulation
  - Logic simulation without respect to a gate propagation delay and the wiring delay.
  - A functional simulation is performed to verify the HDL description.
- Timing simulation
  - The logic simulation with considering a delay times
  - A timing simulation is performed to detect the malfunction.
  - After synthesis
    - A gate propagation delay is considered and the wiring delay is approximately estimated from a statistical data.
  - After place and route
    - A gate propagation delay and the accurate delay time of each wire are considered.

Structure of Test bench

- A test bench includes input vectors, instances of DUT, and directives to specify output signals.
- A test bench is not only an external system of DUT, but also a module in an uppermost layer. Thus, the test bench does not have any port.



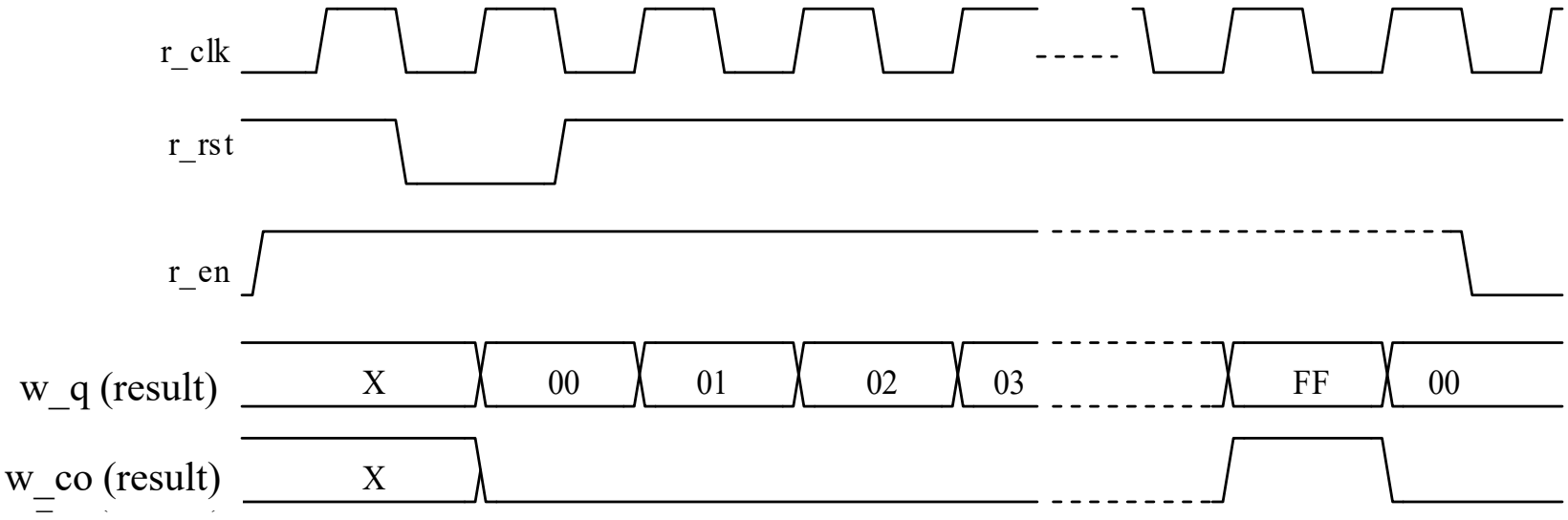
Declaration and Instantiation



Generation of test vector

```
// Test Vectors                    Waiting time
initial begin

                                   r_rst <= 1'b1;
                                   r_en <= 1'b1;
   #(2*hf_cycle)                   r_rst <= 1'b0;
   #(2*hf_cycle)                   r_rst <= 1'b1;
   #(512*hf_cycle)                 r_en <= 1'b0;
end
```

```
// Clock Generation                The simulation is started
initial begin                      from initial block.
   r_clk <= 1'b0;
   for (i = 0; i < 519; i = i + 1)
      #(hf_cycle) r_clk <= ~ r_clk;
   $finish;                        Finish of Simulation.
end
endmodule
```



"initial" block

- An "initial" statement is performed only once after starting the simulation.
  - Important: Do not use "initial" statement in a description to synthesize a circuit, because the real circuit cannot wait the initialization process to finish by oneself.
  - The initialization process of can be described in an always-block started by the external signal such as PoR (Power-on reset) or CS (chip select).

Blocking assignment in always-block

- A blocking assignment is sequentially executed in an instruction order.
- Do not use a blocking assignment for the synthesis, because the blocking assignment in the always-block generates a latch circuit and it is difficult to determine the ordering relations between the blocks.

assignment state	assign b = a; assign c = b;	c != a

Non-blocking assignment	always @(a, b) begin
	b <= a;
	c <= b;
		c != a

Blocking assignment	always @(a, b) begin
	b = a;
	c = b;
		c == a

5. Example of Lticka circuit

Learn the simple example of HDL description in RTL and test bench.

Top module

```
module LED(WIDTH, CLK, RST_B, OUT);

   input     [3:0]     WIDTH;          // Duty Ratio (0 - 15)
   input               CLK, RST_B;     // Clock, Reset
   output              OUT;            // Blinking PWM Output

   wire                dout, pout;

   // Structure of Modules
   DIV div1(.CLK(CLK), .RST_B(RST_B), .DOUT(dout));
   PWM pwm1(.WIDTH(WIDTH), .CLK(dout), .RST_B(RST_B), .POUT(pout));
   BLNK blnk1(.IN(pout), .RST_B(RST_B), .OUT(OUT));

endmodule
```

Divider

```
module DIV(CLK, RST_B, DOUT);

   parameter          NDIV = 15;      // Bit of Divider
   input              CLK, RST_B;  // Clock, Reset
   output             DOUT;           // PWM Output
   reg                DOUT;
   reg [NDIV-1:0]     div;

   always @(posedge CLK or negedge RST_B) begin
      if(!RST_B) begin
         div   <= 0;
         DOUT <= 1'b0;
      end
      else begin
         div <= div + 1;
         DOUT <= (&div) ? ~DOUT : DOUT;
      end
   end
endmodule
```

Pulse width modulator

```
module PWM(WIDTH, CLK, RST_B, POUT);

   input     [3:0]       WIDTH;         // Pulse width
   input                 CLK, RST_B;  // Clock, Reset
   output                POUT;          // PWM Output
   reg                   POUT;
   reg       [3:0]       count;

   always @(posedge CLK or negedge RST_B) begin
      if(!RST_B) begin
         POUT   <= 1'b0;
         count <= 4'b0000;
      end
      else begin
         count <= count + 4'b0001;
         POUT <= (WIDTH >= count);
      end
   end
endmodule
```

Blinking

```
module BLNK(IN, RST_B,  OUT);

   parameter Nper = 256;   // Period of blinking
   input       IN, RST_B;   // Input, Reset
   output      OUT;          // Blinking Output
   reg [8:0]   cnt;

   always @(posedge IN or negedge RST_B) begin
      if(!RST_B) begin
         cnt <= 9'b000000000;
      end
      else begin
         cnt <= cnt + 9'b000000001;
      end
   end

   assign OUT = (cnt < Nper) ? IN : 1'b0;
endmodule
```

Test bench

```
`timescale 1ns / 1ns
module tb_led;
	// Inputs
	reg [3:0]		width;
	reg			clk;
	reg			rst_b;
	// Outputs
	wire			out;

	// Half cycle of clock
	parameter hf_cycle = 1;
	parameter Nstep = 1000000000;
	parameter DUTY = 5;
	integer i, j;

	// Instantiate the Unit Under Test
	LED uut (.WIDTH(width), .CLK(clk), .RST_B(rst_b), .OUT(out));
```

Test bench (cont'd)

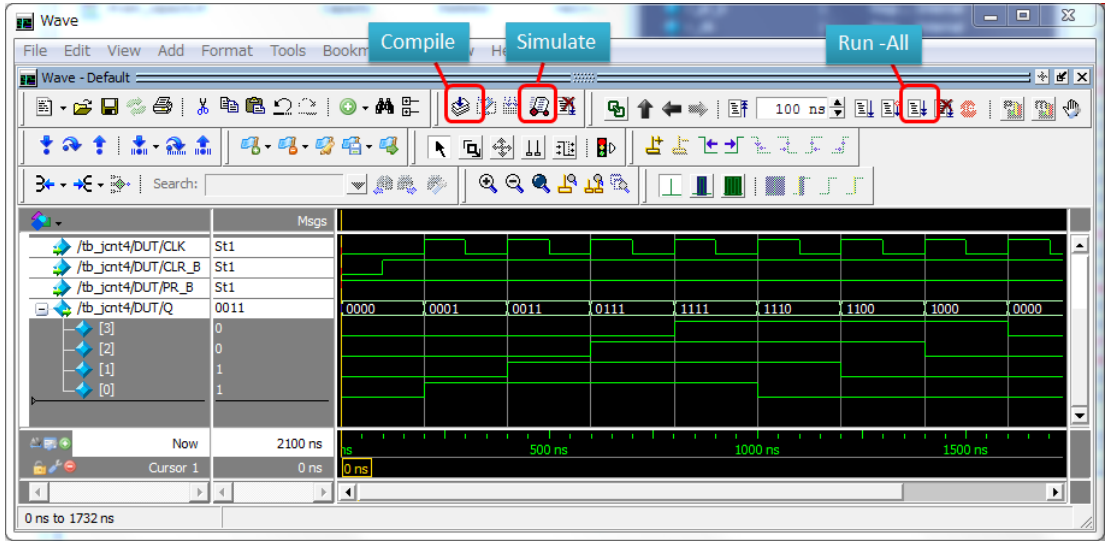
```
        initial begin
        // Initialize Inputs
        width <= 4'b0000;
        rst_b <= 1'b0;

        // Wait 100 ns for global reset
        #(2*hf_cycle) rst_b <= 1'b1;

        // pulse width modulation
        for (j = 0; j < Nstep/(64*65536); i = i + 1)
            #(64*65536*hf_cycle) width <= width + 1'b0001;
        end
        // Clock Generation
        initial begin
            clk <= 1'b0;
            for (i = 0; i < Nstep; i = i + 1)
                 #(hf_cycle) clk <= ~clk;
            $finish;
        end
endmodule
```

Summary

- A logic circuit can be synthesized from HDL code written in RTL.
- The test bench can be described by HDL code in any level of detail by using # delay time.
  - Type in RTL: wire, reg, and integer
  - Assignment in RTL: assign statement or <=
  - Synchronization with clock edge: always @(posedge or negedge)
  - Do not use blocking assignment in RTL to exclude the timing ambiguity.
- Structural description can be used for the hierarchal design.

Appendix 1a

1. Edit and save your Verilog HDL file.
2. Start Modelsim.
3. Menu - [File] - [Change Directory...], and choose your working directory in which your Verilog HDL file is saved.
4. Click the Compile button on the toolbar and choose all Verilog HDL files to be simulated.
5. Click the Compile button on the open file selection dialog.



Appendix 1b

6. Confirm the error message in the Transcript window.
7. If you find the message "Errors: 0, Warnings: 0", click Done button the file selection dialog.
8. Click the Simulation button.
9. Choose [work] - Your test bench file and click the OK button on "Start Simulation" form.
10. Choose the name of the instance of DUT in the "Default" column in the left side.
11. Choose the signal names monitored in the "Objects" column in the right side and right-click.
12. In the pop-up menu, choose "Add Wave" to add the list of the signal names.
13. The waveform window is opened with the tracks of signals.
14. Click the "Run-All" button to start the simulation.
15. The simulation result is shown in the waveform window