Verilog

- Hierarchy, Module, Port and Parameter -

May 2014

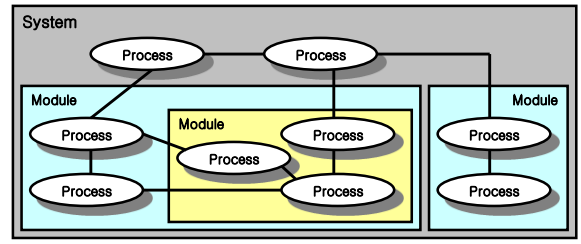
Ando KI

Contents

Verilog-HDL world

- A system consists of a set of components, which forms a hierarchical structure.
- The component is an instantiation of design entity.
- The design entity is a design unit, which is '**module**'.
- A design unit can instantiate other design units, but a design unit cannot declare other design unit.

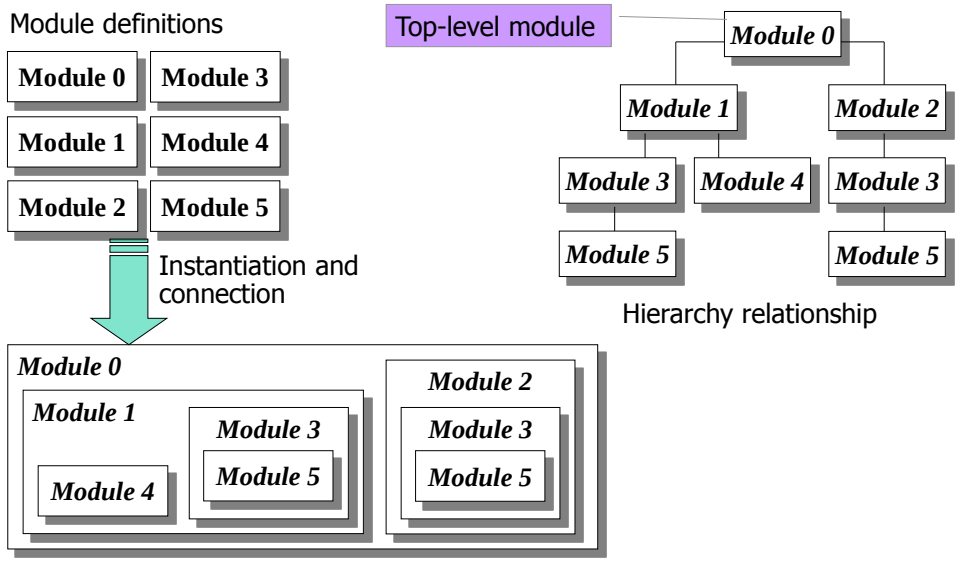
- → A module can be instantiated as many as needed.



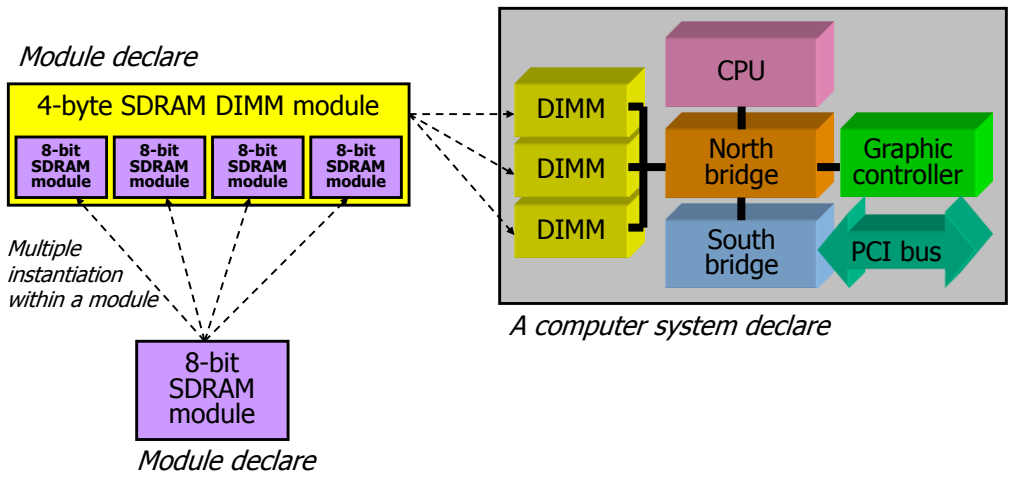
Hierarchical structure (1/2)

- Hierarchical hardware structure
  - Module can embed other modules by creating instances of its lower modules.
  - Modules are connected through ports (input, output, inout).
  - Modules communicate each other through ports.
- Each module definition stands alone.
  - The definition cannot be nested.
    - Module definition does not nest.
    - One module definition shall not contain the text of another module definition.
- A module definition nests another module by instantiating.
- Module can instantiate another module (lower-level module) into itself in order to incorporate a copy of the lower-level module.
  - Instantiation allows one module to incorporate a copy of another module into itself.
- Top-level module is included in the source text, but is not instantiated.

Hierarchical structure (2/2)

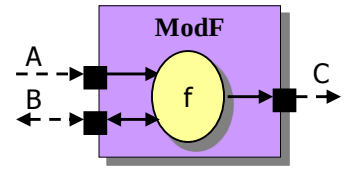


Hierarchical structure example



Module declaration

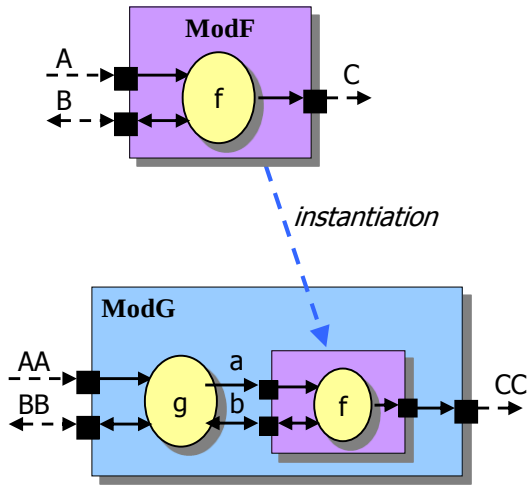
- Module is the basic design unit.
- Module must be declared. → *prepare*
- Module can be instantiated. → *use*
- A module definition shall be enclosed between the keywords 'module' and 'endmodule'.
- Module syntax




```
module module_name (list of ports);
   // in, out, inout port declarations
   // signal/wire/reg declarations
   // data variable declarations
   // sub-module instantiation and connection
   // functional blocks: initial, always, function, task
endmodule
```

```
module ModF (A, B, C);
   input          A;
   inout  [7:0] B;
   output [7:0] C;
   // declarations
   // description of 'f'
endmodule
```


Module instantiation




```
module ModG (AA, BB, CC);
   input          AA;
   inout  [7:0] BB;
   output [7:0] CC;
   wire         a;
   wire [7:0] b;
   // description of 'g'
   ModF Umodf (.A(a), .B(b), .C(CC));
endmodule
```

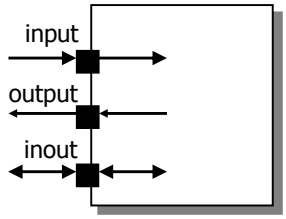
Port connection

Instance name

Module name

Port

- Ports provide a means of interconnecting a hardware description consisting of modules, primitives, and macromodules.



- Port-list
  - It follows module identifier after the keyword 'module'.
  - 'module module_name (port_list);'
  - 'port_list ::= [port [, port […]]]
- Port declaration
  - 'input'
  - 'output'
  - 'inout'
    - bidirectional

Port connection

- Connecting module instance ports by ordered list
  - Positional mapping
  - The ports expressions listed for the module instance shall be in the same order as the ports listed in the module declaration.
- Connecting module instance ports by name
  - Named mapping

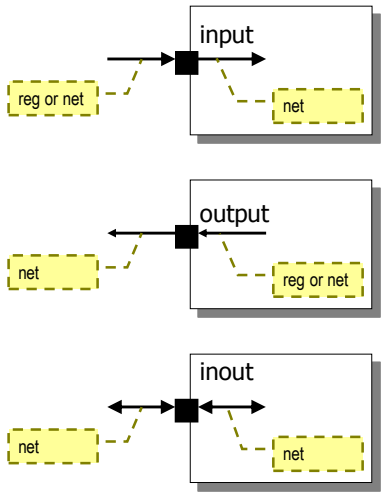
```
module topmod;
  wire [4:0] v;
  wire a,b,c,w;
  modB b1 (v[0], v[3], w, v[4]);
endmodule

module modB (wa, wb, c, d);
  inout wa, wb;
  input c, d;
  tranif1 g1 (wa, wb, cinvert);
  not #(2, 6) n1 (cinvert, int);
  and #(6, 5) g2 (int, c, d);
endmodule
```

```
module topmod;
  wire [4:0] v;
  wire a,b,c,w;
  modB b1 (.wb(v[3]),.wa(v[0]),.d(v[4]),.c(w));
  endmodule

module modB (wa, wb, c, d);
  inout wa, wb;
  input c, d;
  tranif1 g1(wa, wb, cinvert);
  not #(6, 2) n1(cinvert, int);
  and #(5, 6) g2(int, c, d);
endmodule
```

Port connection rules



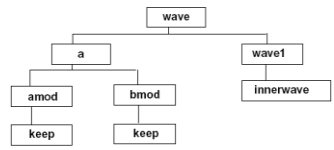
- Port connection rules
  - All input ports must be net.
  - All inout ports must be net.
  - Output port can be either net or reg.

Hierarchical names

- Every identifier in a Verilog HDL description shall have a unique hierarchical path name.

```
wave
wave.stim1
wave.stim2
wave.a
wave.a.stim1
wave.a.stim2
wave.a.amod
wave.a.amod.in
wave.a.amod.keep
wave.a.amod.keep.hold
wave.a.bmod
wave.a.bmod.in
wave.a.bmod.keep
wave.a.bmod.keep.hold
wave.wave1
wave.wave1.innerwave
wave.wave1.innerwave.hold
```

```
module wave;
  reg stim1, stim2;
  cct a(stim1, stim2); // instantiate cct
  initial begin : wave1
    #100 fork : innerwave
           reg hold;
         join
    #150 begin
           stim1 = 0;
         end
  end
endmodule
```




```
module mod (in);
  input in;
  always @ (posedge in) begin : keep
    reg hold;
    hold = in;
  end
endmodule

module cct (stim1, stim2);
  input stm1, stim2;
  // instantiate mod
  mod amod(stim1), bmod(stim2);
endmodule
```

Upward name referencing

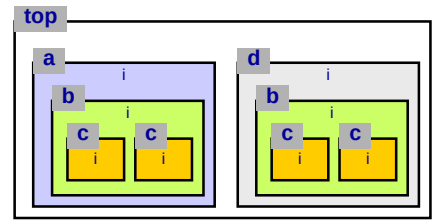
```
module top;
  a u_a();
  d u_d();
  initial begin // full path name references each copy of i
    u_a.i = 1;                u_d.i = 5;
    u_a.a_b1.i = 2;           u_d.d_b1.i = 6;
    u_a.a_b1.b_c1.i = 3; u_d.d_b1.b_c1.i = 7;
    u_a.a_b1.b_c2.i = 4; u_d.d_b1.b_c2.i = 8;
  end
endmodule

module a;
  integer i;
  b a_b1();
endmodule

module b;
  integer i;
  c b_c1();
  c b_c2();
   initial begin
     #10 b_c1.i = 2; // downward path references two copies of i
                     // a.a_b1.b_c1.i, d.d_b1.b_c1.i
     //#10 a.i = 3; d.i =3;  // upward reference - ambiguous
   end
endmodule

module c;
  integer i;
  initial begin
    i = 1; // local name references four copies of i
           // a.a_b1.b_c1.i, a.a_b1.b_c2.i,
           // d.d_b1.b_c1.i, d.d_b1.b_c2.i
    b.i = 1; // upward path references two copies of i
           // a.a_b1.i, d.d_b1.i
  end
endmodule
```

- 'code/verilog/module/upward' example

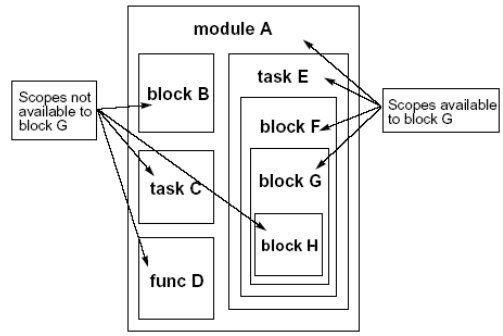



```
module d;
  integer i;
  b d_b1();
  initial begin // full path name references each copy of i
    i = 5;
    d_b1.i = 6;
    d_b1.b_c1.i = 7;
    d_b1.b_c2.i = 8;
    d.i = 9;
    d.d_b1.i = 10;
    d.d_b1.b_c1.i = 11;
    d.d_b1.b_c2.i = 12;
  end
endmodule
```

DIY

Scope rule

- The following four elements define a new scope in Verilog
  - Module
  - Task
  - Function
  - Name block
- If identifier is not found in the current scope, how to find it?
  - Scope rule determines how to find it.
  - To find a variable
    - If it is not found at the same level, search continues upward until an item by the name is found or until a module boundary is encounter.
  - To find a task, function, or name block
    - If it is not found at the same level, it continues to search higher-level modules until found.



Scope rule: example

```
`timescale 1ns/1ns
module top;
  reg [31:0] my_id;
  initial my_id = 100;
  mod_A UA();
  function [31:0] get_my_id;
    get_my_id = my_id;
  endfunction
  task put_my_id;
    $display($time,,"%m: my_id is %d", my_id);
  endtask
endmodule

module mod_A;
  reg  [31:0] mid_id;
  initial mid_id = 200;
  mod_B UB();
endmodule

module mod_B;
  reg  [31:0] local_id;
  initial begin : C
    local_id = 1;
    #100; put_id();
    #100; put_my_id();
    #100; local_id = get_my_id();
    #100; put_id();
    #100; $display($time,,"%m: mid_id  %d", mod_A.mid_id);
    #100;
    $finish;
  end
  task put_id;
    $display($time,,"%m: id is   %d", local_id);
  endtask
endmodule
```

- 'code/verilog/module/scope' example

How to find 'my_id', which is not found in 'get_my_id()' function scope.
→ search upward until module boundary

DIY

How to find 'put_id()', which is not found in 'mod_B' module scope.
→ search upward higher-level module

'mod_A' cannot directly use 'my_id' residing in 'top' module, but upward reference makes it possible.

How to find 'local_id', which is not found in 'put_id()' function scope.
→ search upward until module boundary

Parameter

- Parameters are constants, not variables.
  - Parameter represents constant since it cannot be modified at simulation time.
- Parameter can have default value.
- Parameter can be assigned by expression.
- Parameter can be modified in the module instance statement.
  - The order is important.

```
module my_memory(addr, datai, datao, rw, clk);
parameter ADD_WIDTH  = 1, DATA_WIDTH = 8;
parameter DELAY=0;
input  [ADD_WIDTH-1:0]  addr;
input  [DATA_WIDTH-1:0] datai;
output [DATA_WIDTH-1:0] datao;
input   rw;
input   clk;
//--------------------------------
parameter MEM_DEPTH = 1<<ADD_WIDTH;
reg [DATA_WIDTH-1:0] mem[0:MEM_DEPTH-1];
always @ (posedge clk) begin
  if (rw) #(DELAY) datao = mem[addr];
  else             mem[addr] = datai;
end
endmodule
//--------------------------------
module top;
...
my_memory  UmemA (addA, datA, rw, clk);
my_memory  #(3,4)   UmemB (addB, datB, rw, clk);
my_memory  #(3,4,2) UmemC (addC, datC, rw, clk);
my_memory  #(,,3)   UmemC (addD, datD, rw, clk);
...
endmodule
```

Comma-separate list is possible.

Equation is possible

Default parameter

Parameter overriding

Parameter: example (1/3)

- 'code/verilog/module/parameter' example

```
// mem_generic.v
`timescale 1ns/1ns
module mem_generic(add, datr, datw, en, we, clk, rstb);
  parameter ADD_WIDTH=8, DAT_WIDTH=8;
  parameter DELAY=0;
  //-------------------------------------------------
  input  [ADD_WIDTH-1:0] add;  wire [ADD_WIDTH-1:0] add;
  output [DAT_WIDTH-1:0] datr; reg  [DAT_WIDTH-1:0] datr;
  input  [DAT_WIDTH-1:0] datw; wire [DAT_WIDTH-1:0] datw;
  input                  en;   wire                 en;
  input                  we;   wire                 we;
  input                  clk;  wire                 clk;
  input                  rstb; wire                 rstb;
  //-------------------------------------------------
  localparam DEPTH = 1<<ADD_WIDTH;
  //-------------------------------------------------
  reg [DAT_WIDTH-1:0] mem[0:DEPTH-1];
  //-------------------------------------------------
  always @ (posedge clk or negedge rstb) begin
    if (rstb==1'b0) begin
      datr  <= {DAT_WIDTH{1'b0}};
    end else begin
      if (en==1'b1) begin
        if (we==1'b1) begin
          datr          <= #(DELAY) datw;
          mem[add] <= datw;
        end else begin
          datr     <= #(DELAY) mem[add];
        end
      end
    end
  end
endmodule
```

Generic memory model, where address width, data width and response time are specified by parameters

Width specified by parameters

Internal local parameter

Storage elements

Assignment delay specified by parameter.

Parameter: example (2/3)

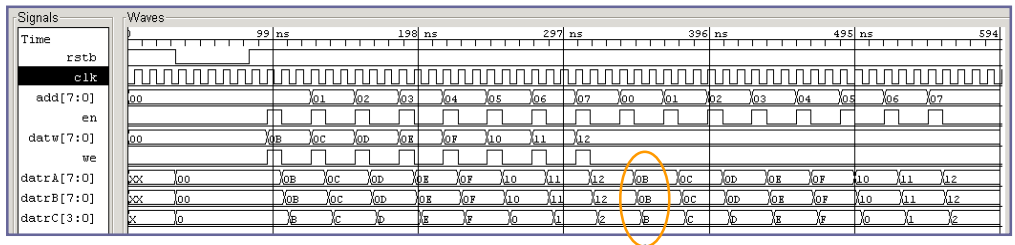
- 'code/verilog/module/parameter' example

```
// top.v
`timescale 1ns/1ns
module top;
  parameter AW=8, DW=8;
  reg  [AW-1:0] add;
  reg  [DW-1:0] datw;
  wire [DW-1:0] datrA, datrB;
  wire [3:0]    datrC;
  reg           en, we;
  reg           clk, rstb;
  integer x;
  //-------------------------------------------------
  mem_generic           UA (add, datrA, datw, en, we, clk, rstb);
  mem_generic #(8,8,2) UB (add, datrB, datw, en, we, clk, rstb);
  mem_generic #(5,4,5) UC (add[4:0], datrC[3:0], datw[3:0], en, we, clk, rstb);
  //-------------------------------------------------
  always @ (*) #5 clk <= ~clk;
  initial begin
    add  <= {AW{1'b0}};
    datw <= {DW{1'b0}};
    en   <= 1'b0; we   <= 1'b0;
    clk  <= 1'b0; rstb <= 1'b1;
    #33 rstb <= 1'b0;
    #50 rstb <= 1'b1;
  end
  //-------------------------------------------------
  .. details are not shown ..

endmodule
```

Different memory models are instantiated using parameter assignmemt.

Parameter: example (3/3)



Note the delay and bus width

DIY