95-1 Under-Graduate Project
Logic Design with Behavioral Models

Speaker: 陳圓覺
Adviser: Prof. An-Yeu Wu
Date: 2006/10/11
Outline

- Operation
- Assignment
- Blocking and non-blocking
- Appendix
Various Abstraction of Verilog HDL

- Gate level Verilog description of a full-adder

```verilog
module fadder(cout, sum, a, b, cin);
// port declaration
    output cout, sum;
    input a, b, cin;
    wire net1, net2, net3;
// Netlist description
    xor U0(sum,a,b,cin);
    and U1(net1, a, b);
    and U2(net2, b, cin);
    and U3(net3, cin, a);
    or   U4(cout, net1, net2, net3);
endmodule
```
Various Abstraction of Verilog HDL

- RTL level Verilog description of a full adder

```verilog
code
module fadder(cout, sum, a, b, cin);
// port declaration
output  cout, sum;
input    a, b, cin;
wire     cout, sum;
// RTL description
assign  sum = a^b^cin;
assign  cout = (a&b)|(b&cin)|(cin&a);
endmodule
```

Whenever \( a \) or \( b \) or \( c \) changes its logic state, evaluate \( sum \) and \( cout \) by using the equation

\[
\begin{align*}
\text{sum} &= a \oplus b \oplus c \\
\text{cout} &= ab + bc + ca
\end{align*}
\]
Various Abstraction of Verilog HDL

- Behavioral level Verilog description of a full adder

```verilog
module fadder(cout, sum, a, b, cin);
  // port declaration
  output cout, sum;
  input a, b, cin;
  reg cout, sum;
  // behavior description
  always @(a or b or cin)
    begin
      {cout,sum} = a + b + cin;
    end
endmodule
```

Diagram of a full adder circuit with inputs a, b, and cin, and outputs cout and sum.
Operators

Arithmetic Operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>%</td>
<td>Modulus</td>
</tr>
</tbody>
</table>

Examples:

A = 4'b0011; B = 4'b0100; // A and B are register vectors
D = 6; E = 4; // D and E are integers
A * B // 4'b1100
D / E // 1
A + B // 4'b0111
in1 = 4'b101x; in2 = 4'b1010;
sum = in1 + in2; // 4'bx
-10 % 3 = -1 // Take sign of the first operand
14 % -3 = 2
Operators-example (arithmetic)

module test;
    reg [3:0] A,B;
    wire [4:0] sum,diff1,diff2,neg;

    assign sum=A+B;
    assign diff1=A-B;
    assign diff2=B-A;
    assign neg=-A;

    initial
    begin
        #5 A=5;B=2;
        $display("t_sim A B A+B A-B B-A -A");
        $monitor($time,"%d%d%d%d",A,B,sum,
                   diff1,diff2,neg);
        #10 $monitoroff;
        $monitor($time,"%b%b%b%b%b",A,B,sum,
                   diff1,diff2,neg);
    endmodule

<table>
<thead>
<tr>
<th>t_sim</th>
<th>A</th>
<th>B</th>
<th>A+B</th>
<th>A-B</th>
<th>B-A</th>
<th>-A</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>2</td>
<td>7</td>
<td>3</td>
<td>29</td>
<td>27</td>
</tr>
<tr>
<td>15</td>
<td>0101</td>
<td>0010</td>
<td>00111</td>
<td>00011</td>
<td>11101</td>
<td>11011</td>
</tr>
</tbody>
</table>
Operator

- Bit-wise operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>Bitwise negation</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise and</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>Bitwise exclusive or</td>
</tr>
<tr>
<td>^, ^~</td>
<td>Bitwise exclusive nor</td>
</tr>
</tbody>
</table>

\(~(101011) = 010100\)
\((010101) \& (001100) = 000100\)
\((010101) | (001100) = 011101\)
\((010101) ^ (001100) = 011001\)
Operator

Reduction operators (1-bit result)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>Reduction and</td>
</tr>
<tr>
<td>~&amp;</td>
<td>Reduction nand</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>~</td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>Reduction exclusive or</td>
</tr>
<tr>
<td><del>^, ^</del></td>
<td>Reduction exclusive nor</td>
</tr>
</tbody>
</table>

&(10101010)=1’b0
|(10101010)=1’b1
&(10x0x0x)=1’b0
|(10x01010)=1’b1
Operator

Logical operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>Logical negation</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical and</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>==</td>
<td>Logical equality</td>
</tr>
<tr>
<td>!=</td>
<td>Logical inequality</td>
</tr>
<tr>
<td>===</td>
<td>Case equality</td>
</tr>
<tr>
<td>!==</td>
<td>Case inequality</td>
</tr>
</tbody>
</table>

=== : determines whether two words match identically on a bit-by-bit basis, including bits that have values “x” and “z”
Operator

- Shift operator
  - `>>` logical shift right
  - `<<` logical shift left

```verilog
module shift_reg(out,in);
  output [5:0] out;
  input [5:0] in;
  parameter shift=3;
  assign out=in<<shift;
endmodule
```

```
reg_in=6'b011100
reg_in<<3   100000  
reg_in>>3   000011
```
Operator

- Conditional Operator

Usage: `conditional_expression ? true_expression: false_expression;`

- The action of a conditional operator is similar to a multiplexer
Examples of Conditional Operator

//model functionality of a tristate buffer
assign addr_bus = drive_enable ? addr_out : 36’bz;

//model functionality of a 2-to-1 multiplexer
assign out = control ? in1 : in0;

Conditional operations can be nested. Each *true_expression* or *false_expression* can itself be a conditional operation
assign out = s1 ? ( s0 ? i3 : i2 ) : ( s0 ? i1 : i0 );
Operator

Conditional Operator

module MUX4_1(out,i0,i1,i2,i3,sel);
    output [3:0] out;
    input [3:0] i0,i1,i2,i3;
    input [1:0] sel;

    assign out=(sel==2'b00)?i0:
               (sel==2'b01)?i1:
               (sel==2'b10)?i2:
               (sel==2'b11)?i3:
                      4'bx;

endmodule
Operator

• Concatenation and Replication Operator

Concatenation operator in LHS

```verilog
module add_32 (co, sum, a, b, ci);

    output co;
    output [31:0] sum;
    input  [31:0] a, b;
    input   ci;

    assign #100 {co, sum} = a + b + ci;

endmodule
```

Bit replication to produce 01010101

```verilog
assign byte = {4{2'bx01}};
```

Sign Extension

```verilog
assign word = {{8{byte[7]}}, byte};
```
Expression Bit Widths

- $x ? y : z$
  - Conditional
  - Bit width = $\max(\text{width}(y), \text{width}(z))$

- $\{x, \ldots, y\}$
  - Concatenation
  - Bit width = $\text{width}(x) + \ldots + \text{width}(y)$

- $\{x,y, \ldots, z\}$
  - Replication
  - Bit width = $x \times (\text{width}(y) + \ldots + \text{width}(z))$
Expressions with Operands Containing x or z

- Arithmetic
  - If any bit is x or z, result is all x’s.
  - Divide by 0 produces all x’s.

- Relational
  - If any bit is x or z, result is x.

- Logical
  - == and != If any bit is x or z, result is x.
  - === and !== All bits including x and z values must match for equality
Outline

- Operation
- Assignment
- Blocking and non-blocking
- Appendix
Assignments

- Assignment: Drive value onto nets and registers
- There are two basic forms of assignment
  - continuous assignment, which assigns values to nets
  - procedural assignment, which assigns values to registers
- Basic form

\[ \text{ Left Hand Side} = \text{ Right Hand Side} \]

<table>
<thead>
<tr>
<th>Assignments</th>
<th>Left Hand Side</th>
</tr>
</thead>
<tbody>
<tr>
<td>Continuous Assignment</td>
<td>Net wire, tri</td>
</tr>
<tr>
<td>Procedural Assignment</td>
<td>Register reg, integer, real</td>
</tr>
</tbody>
</table>

Left Hand Side = LHS
Right Hand Side = RHS
Assignments

- **Continuous assignment**

```verilog
module holiday_1(sat, sun, weekend);
    input sat, sun; output weekend;
    assign weekend = sat | sun; // outside a procedure
endmodule
```

- **Procedural assignment**

```verilog
module holiday_2(sat, sun, weekend);
    input sat, sun; output weekend; reg weekend;
    always #1 weekend = sat | sun; // inside a procedure
endmodule
```

```verilog
module assignments
    // continuous assignments go here
always begin
    // procedural assignments go here
end
endmodule
```
Continuous Assignment

- Drive a value onto a wire, wand, wor, or tri
  - Use an explicit continuous assignment statement after declaration
  - Specify the continuous assignment statement in the same line as the declaration for a wire

- Used for datapath descriptions
- Used to model combinational circuits
Continuous Assignments

- Convenient for logical or datapath specifications

```verilog
wire [8:0] sum;
wire [7:0] a, b;
wire carryin;
assign sum = a + b + carryin;
```

Define bus widths

Continuous assignment: permanently sets the value of sum to be a+b+carryin

Recomputed when a, b, or carryin changes
Continuous Assignments

- Continuous assignments provide a way to model combinational logic

```verilog
module inv_array(out,in);
    output [31:0] out;
    input [31:0] in;
    assign out=~in;
endmodule
```

```verilog
module inv_array(out,in);
    output [31:0] out;
    input [31:0] in;
    not U1(out[0],in[0]);
    not U2(out[1],in[1]);
    ..
    not U31(out[31],in[31]);
endmodule
```
Continuous Assignments

- Examples of continuous Assignment
  ```verilog
  assign out = i1 & i2;
  // *i1* and *i2* are nets
  assign addr[15:0] = addr1[15:0] ^ addr2[15:0]
  // Continuous assign for vector nets *addr* is a 16-bit vector net
  // *addr1* and *addr2* are 16-bit vector registers
  assign {cout, sum[3:0]} = a[3:0]+b[3:0]+cin;
  // LHS is a concatenation of a scalar net and vector net
  ```

- Instead of declaring a net and then writing a continuous assignment on the net, Verilog provides a shortcut by which a continuous assignment can be placed on a net when it is declared.

```verilog
wire a;    --declare
assign a = b & c;  --assign

wire a = b & c;   --declare and assign
```
Continuous Assignment

- Avoid logic loop
  - HDL Compiler and Design Compiler will automatically open up asynchronous logic loops
  - Without disabling the combinational feedback loop, the static timing analyzer can’t resolve

- Example

```
assign a = b + a;
```

![Diagram of logic loop](image)
Procedural Assignments

Procedural assignments drive values or expressions onto registers (reg, integer, real, time)

```verilog
module adder32(sum, cout, a, b, ci);
    output [31:0] sum;
    output cout;
    input [31:0] a, b;
    input ci;
    reg [31:0] sum;
    reg cout;

    always @(a or b or ci)
        {carry, sum} = a + b + ci;
endmodule
```
Procedural Assignments

- Inside an initial or always block:

  ```
  initial
  begin
  {cout, sum} = a + b + cin;
  end
  ```

  ```
  always
  begin
  {cout, sum} = a + b + cin;
  end
  ```

- Just like in C: RHS evaluated and assigned to LHS before next statement executes.

- RHS may contain wires and regs:
  - Two possible sources for data
  - LHS must be a reg:
    - Primitives or cont. assignment may set wire values
Outline

- Operation
- Assignment
- Blocking and non-blocking
- Appendix
Procedural Assignments

- **Blocking assignment** statements are executed in the order they are specified in a sequential block.

```verilog
reg x, y, z;
reg [7:0] rega, regb;
integer count;
initial
begin
    x = 0; y = 1; z = 1;
count = 0;
rega = 8'b0; regb = rega;
#15 rega[2] = 1'b1;
#10 regb[7:5] = {x, y, z};
count = count + 1;
end
```

<table>
<thead>
<tr>
<th>time</th>
<th>statements executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x=0; y=1; z=1;</td>
</tr>
<tr>
<td></td>
<td>count=0; rega=0;</td>
</tr>
<tr>
<td></td>
<td>regb=rega=0;</td>
</tr>
<tr>
<td>15</td>
<td>rega[2]=1;</td>
</tr>
<tr>
<td>25</td>
<td>regb[7:5]={x, y, z};</td>
</tr>
<tr>
<td></td>
<td>count=count+1;</td>
</tr>
</tbody>
</table>
Procedural Assignments

Nonblocking assignment statements allow scheduling of assignment without blocking execution of the statements that follow in a sequential block. A <= operator is used to specify nonblocking assignments.

```verilog
reg x, y, z;
reg [7:0] rega, regb;
integer count;
initial
begin
  x=0; y=1; z=1;
  count=0;
  rega=8'b0; regb=rega;
  rega[2] <= #15 1'b1;
  regb[7:5] <= #10 {x, y, z};
  count <= count+1;
end
```

<table>
<thead>
<tr>
<th>Time (n)</th>
<th>Statements executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x=0; y=1; z=1; count=0; rega=0; regb=rega=0;</td>
</tr>
<tr>
<td>10</td>
<td>regb[7:5]={x, y, z};</td>
</tr>
<tr>
<td>15</td>
<td>rega[2]=1;</td>
</tr>
</tbody>
</table>
Non-blocking Can Behave Oddly

A sequence of nonblocking assignments don’t communicate

always @(posedge clk) begin
  a = 1;
  b = a;
  c = b;
end

Blocking assignment:
a = b = c = 1

always @(posedge clk) begin
  a <= 1;
  b <= a;
  c <= b;
end

Nonblocking assignment:
a = 1
b = old value of a
c = old value of b
Non-blocking Looks Like Latches

- RHS of nonblocking taken from latches
- RHS of blocking taken from wires

a = 1;
b = a;
c = b;
a <= 1;
b <= a;
c <= b;
Procedural Assignments

- Non-blocking statements to Eliminate race conditions

  // Illustration 1: blocking
  always @(posedge clk)
  begin
    a = b;
    b = a;
  end

  // Illustration 2: nonblocking
  always @(posedge clk)
  begin
    a <= b;
    b <= a;
  end

- In illustration 1, there is a race condition. In illustration 2, using nonblocking to eliminate it.
Sequential Block

- Sequential block may appear in an always or initial statement.

<table>
<thead>
<tr>
<th></th>
<th>initial</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>always</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Runs when simulation starts
- Terminates when control reaches the end
  (one time sequential activity flow)
- Good for providing stimulus (testbenches); not synthesizable

- Runs when simulation starts
- Restarts when control reaches the end
  (cycle sequential activity flow)
- Good for modeling/specifying hardware
initial and always

- Run until they encounter a delay

```verilog
initial begin
  #10 a = 1; b = 0;
  #10 a = 0; b = 1;
end
```

- or a wait for an event

```verilog
always
  begin
    wait(i); a = 0;
    wait(~i); a = 1;
  end
```
Sequential and Parallel Blocks

There are two types of blocks: **sequential** blocks and **parallel** blocks

//Illustration 1: sequential block without delay
reg x, y;
reg [1:0] z, w;
initial
    begin
        x='b0;
        y='b1;
        z={x, y};
        w={y, x};
    end

//Illustration 2: sequential blocks with delay
reg x, y;
reg [1:0] z, w;
initial
    begin
        x='b0;
        #5 y='b1;
        #10 z={x, y};
        #20 w={y, x};
    end
Sequential and Parallel Blocks

- Parallel blocks, specified by keywords fork and join, execute concurrently

```vhdl
// Parallel blocks with delay
reg x, y;
reg [1:0] z, w;
initial
fork
  x = 1'b0;
  #5 y = 1'b1;
  #10 z = {x, y};
  #20 w = {y, x};
join
```
Conditional Statements

- If and If-else statements

```
if (expression)
    statement
else
    statement
```

```
if (expression)
    statement
else if (expression)
    statement
else
    statement
```
Conditional Statements

If and If-Else Statements (cont.)

- Examples

  ```
  if (rega >= regb)
      result = 1;
  else
      result = 0;
  ```

  ```
  if (index > 0)
      if (rega > regb)
          result = rega;
      else
          result = 0;
  else
      $display("* Warning * index is equal or small than 0!");
  ```
Multiway Branching

- The nested *if-else-if* can become unwieldy if there are too many alternatives. A shortcut to achieve the same result is to use the `case` statement.

```plaintext
case (expression)
  alternative1: statement1;
  alternative2: statement2;
  alternative3: statement3;
  ...
  default: default_statement;
endcase
```

```plaintext
reg [1:0] alu_control;
... case (alu_control)
  2'b00: y=x+z;
  2'b01: y=x-z;
  2'b10: x*z;
  ...
  default: y='bx;
endcase
```
Multiway Branching

4-to-1 Multiplexer with `case` Statement

```verilog
module mux4_to_1(out,i0,i1,i2,i3,s1,s0);
//port declarations from the I/O diagram
output out;
input i0,i1,i2,i3;
input s1,s0;
reg out;

always @(s1 or s0 or i0 or i1 or i2 or i3)
    case ( {s1,s0})
        2'd0: out = i0;
        2'd1: out = i1;
        2'd2: out = i2;
        2'd3: out = i3;
        default: out = 1'bx;
    endcase

endmodule
```
Multiway Branching

- There are 2 variations of the case statement. They are denoted by keywords `casex` and `casez`
  - `casez` treats all `z` values in the case alternatives or the case expression as don’t cases. All bit positions with `z` can also be represented by `?` in that position.
  - `casex` treats all `x` and `z` values in the case item or the case expression as don’t cares.

```plaintext
reg [3:0] encoding;
integer state;
casex (encoding)
  // logic value x represents a don't care bit
  4'b1xxx: next_state=3;
  4'bx1xx: next_state=2;
  4'bxx1x: next_state=1;
  4'bxxx1: next_state=0;
  default: next_state=0;
endcase
```

`encoding=4'b10xz     next_state=3`
While Loop

The **while** loop executes until the **while**-expression becomes false.

```plaintext
initial //Illustration 1:
begin
  count=0;
while(count<128)
begin
  $display("count=%d",count);
count=count+1;
end
end
```

```plaintext
initial
begin
  reg [7:0] tempreg;
count = 0;
tempreg = reg;
while (tempreg)
begin
  if (tempreg[0]) count = count + 1;
tempreg = tempreg >> 1;
end

rega = 101;
tempreg   count
101      1
010      1
001      2
```
For Loop

- The keyword `for` is used to specify this loop. The `for` loop contain 3 parts:
  - An initial condition
  - A check to see if the terminating condition is true
  - A procedural assignment to change value of the control variable

```vhdl
//Initialize array elements
`define MAX_STAGES 32
reg [0:31] array;
integer i;

initial begin
  for(i=0;i<32;i=i+2)
    array[i]=0;
  for(i=1;i<32;i=i+2)
    array[i]=1;
end
```
Repeat Loop

The keyword `repeat` is used for this loop. The `repeat` construct executes the loop a fixed number of times.

```verilog
module multiplier(result, op_a, op_b);

    ...
    reg shift_opa, shift_opb;
    parameter size = 8;
    initial begin
        result = 0; shift_opa = op_a; shift_opb = op_b;
    repeat (size)
        begin
            #10 if (shift_opb[1])
                result = result + shift_opa;
            shift_opa = shift_opa << 1;
            shift_opb = shift_opb >> 1;
        end
    end
endmodule
```
The keyword **forever** is used to express the loop. The loop does not contain any expression and executes forever until the `$finish` task is encountered.

```verilog
//Clock generation
//Clock with period of 20 units
reg clk;

initial
begin
  clk=1'b0;
  forever #10 clk=~clk;
end

//Synchronize 2 register values
//at every positive edge of clock
reg clk;
reg x,y;

initial
  forever @(posedge clk) x=y;
```
Modeling A Flip-Flop With Always

- Very basic: an edge-sensitive flip-flop
  ```
  reg q;
  always @(posedge clk)
  q = d;
  ```

- q = d assignment runs when clock rises: exactly the behavior you expect

- Keywords:
  - `posedge` for positive edge trigger
  - `negedge` for negative edge trigger
Timing Controls

- Event-Based Timing Control
- Regular Event Control

```verilog
module test;
  reg clk;
  reg [2:0] q,d;
  always #5 clk=~clk;
  initial begin
    clk=0;
    #2 q=0;d=0;
    #2 d=2;
    @(posedge clk) q=d;
    @(negedge clk) q=1;
    q=@(posedge clk) 3;
    #10 $finish;
  end
  endmodule
```
### Event-Based Timing Control

#### Named Event Control

```verilog
event received_data;
always @(posedge clk)
begin
    if (last_data_packet)
        -> received_data;
end
always @(received_data)
    data_buf = {data_pkt[0], data_pkt[1]};
```

#### Event OR Control

```verilog
always @(rst orposedge clk or set)
begin
    if(rst) q=0;
    else if (set) q=1;
    else q=d;
end
assign _q=~q;
```
Timing Controls

- Level-Sensitive Timing Control

```plaintext
always
    wait (count_enable) #20 count = count +1 ;
// 1. If count_enable is logical 1, count=count+1
//    after 20 time unit.
// 2. If count_enable stays at 1, count will be
//    incremented every 20 time units
```
Outline

- Operation
- Assignment
- Blocking and non-blocking
- Appendix
Disabling named blocks (example: comparator)

```verilog
module comparator(a,b,a_gt_b,a_lt_b,a_eq_b);
  parameter size=2;
  input [size:1] a,b;
  output a_gt_b,a_lt_b,a_eq_b;
  reg a_gt_b,a_lt_b,a_eq_b;
  integer k;

  always @(a or b) begin: compare_loop
    for (k=size;k>0;k=k-1) begin
      if (a[k]!=b[k]) begin
        a_gt_b=a[k];
        a_lt_b=~a[k];
        a_eq_b=0;
        disable compare_loop;
      end
    end
    a_gt_b=0;
    a_lt_b=0;
    a_eq_b=1;
  end
endmodule
```

Block Disable

Disabling named blocks (example: comparator)

```verilog
module comparator(a,b,a_gt_b,a_lt_b,a_eq_b);
  parameter size=2;
  input [size:1] a,b;
  output a_gt_b,a_lt_b,a_eq_b;
  reg a_gt_b,a_lt_b,a_eq_b;
  integer k;

  always @(a or b) begin: compare_loop
    for (k=size;k>0;k=k-1) begin
      if (a[k]!=b[k]) begin
        a_gt_b=a[k];
        a_lt_b=~a[k];
        a_eq_b=0;
        disable compare_loop;
      end
    end
    a_gt_b=0;
    a_lt_b=0;
    a_eq_b=1;
  end
endmodule
```
Tasks

- Task are declared with the keyword `task` and `endtask`. It must be used if any one of the following conditions is true for the procedure:
  - There are delay, timing, or event control constructs in the procedure.
  - The procedure has zero or more than one output arguments.
  - The procedure has no input argument.
Tasks example

module operation;
...

parameter delay=10;
reg [15:0] A,B;
reg [15:0] AB_AND,AB_OR,AB_XOR;

always @(A or B)
  begin
    bitwise_oper(AB_AND,AB_OR,AB_XOR,A,B);
  end
...

task bitwise_oper;
  output [15:0] ab_and,ab_or,ab_xor;
  input [15:0] a,b;
  begin
    #delay ab_and=a&b;
    ab_or=a|b;
    ab_xor=a^b;
  end
endtask
endmodule
Functions

Functions are declared with the keywords `function` and `endfunction`. Functions are used if all of the following conditions are true for the procedure:

- There are **no delay, timing, or event control** constructs in the procedure
- The procedure **returns a single value**
- There is at least one input argument

There are some peculiarities of functions. When a function is declared, a register with name (name of function) is declared implicitly inside.

Functions **cannot invoke other tasks**. They can only invoke other functions.
Functions example

module check(number);
  input [9:0] number;
  reg correct;

  always @(number)
  begin
    correct = berger(number);
    $display("The correct of the number: %b is %b ",number,correct);
  end
...

function berger;
  input [9:0] number;
  reg [2:0] temp3;
  integer i;
  begin
    temp3 = 3'b000;
    for(i=3;i<10;i=i+1)
      temp3 = temp3 + number[i];
    temp3 = temp3 ^ 3'b111;
    if( temp3 === number[2:0])
      berger = 1'b1;
    else
      berger = 1'b0;
  end
endfunction
endumodule

module test;
  reg [9:0] num;
  check chk1(num);
  initial
    begin
      num = 10'b0110101111;
      #10 num = 10'b0111111011;
      #10 $finish;
    end
endmodule

simulation result
The correct of the number: 0110101111 is 1
The correct of the number: 0111111011 is 0
## Differences Between Tasks and Functions

### Tasks and Functions

<table>
<thead>
<tr>
<th>Functions</th>
<th>Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>A function can enable another function but not another task.</td>
<td>A task can enable other tasks and functions.</td>
</tr>
<tr>
<td>Function always execute in 0 simulation time.</td>
<td>Tasks may execute in non-zero simulation time.</td>
</tr>
<tr>
<td>Functions must not contain any delay, event, or timing control statements.</td>
<td>Tasks may contain delay, event, or timing control statements.</td>
</tr>
<tr>
<td>Functions must have <strong>at least one input</strong> argument. They can have more than one input.</td>
<td>Task may have <strong>zero or more</strong> arguments of type <strong>input</strong>, <strong>output</strong> or <strong>inout</strong>.</td>
</tr>
<tr>
<td>Functions always return a single value. They cannot have <strong>output</strong> or <strong>inout</strong> arguments.</td>
<td>Tasks do not return with a value but can pass multiple values through <strong>output</strong> and <strong>inout</strong> arguments.</td>
</tr>
</tbody>
</table>