ISTANBUL TECHNICAL UNIVERSITY

Digital System Design & Applications

for loops & parameters & generate statement

RES. ASST. FIRAT KULA & RES. ASST. SERDAR DURAN

Parameters

- Verilog allows **constants** to be defined in a module by the keyword **parameter**.
- Parameters cannot be used as variables. They are constants.
- Parameters values can be changed at **module instantiation**.

```
parameter port_id = 5;                // Defines a constant port_id
parameter cache_line_width = 256;     // Constant defines width of cache line
```

Declarations

```
module hello_world #(parameter id_num = 0)();

initial
    $display("Displaying hello_world
              id number = %d", id_num);

endmodule
```

ANSI C Style Declaration

```
module hello_word();

parameter id_num = 0;

initial
    $display("Displaying hello_word
              id number = %d", id_num );
endmodule
```

Overriding

- Parameter values can be overridden when a module is instantiated.

```
module hello_world #(parameter id_num = 0);

initial
    $display("Displaying hello_world
              id number = %d", id_num);

endmodule
```

```
module top;
// Parameter value assignment by ordered list
hello_world #(1) w1;

//Parameter value assignment by name
hello_world #(.id_num(2)) w2;

endmodule
```

Defparam

- **Defparam** statement and the hierarchical name of the instance can be used to **override parameter** values.

```
module hello_word;

parameter id_num = 0;

initial
    $display("Displaying hello_word
              id number = %d", id_num );
endmodule
```

```
module top;

//change parameter values in the instantiated modules
defparam w1.id_num = 1, w2.id_num = 2;

hello_word w1();
hello_word w2();

endmodule

/*
Output:
Displaying hello_world id number = 1
Displaying hello_world id number = 2
*/
```

Looping statements in Verilog

- **Verilog HDL includes 4 different looping statements:**

```
forever
begin
    // statement(s)
end
```

- Executes the statements within continiously

```
repeat(/*expression*/)
begin
    // statements
end
```

- Executes the statements within “value of expression” times. For example, repeat(4) would execute the statements within 4 times.

```
while(/*expression*/)
begin
    //statements
end
```

- Executes the statements within until the expression becomes false

Looping statements in Verilog

```
for(/* loop parameter initial value */; /* loop control statement*/; /* increment */)
begin
    // statements
end
```

- Verilog loops have similar syntax structure with C language
- Loop parameters might be integers, genvars, regs...
- For loop control statements, comparison operators are usually used (ex. i <= 20, i == 100)
- For incrementation rule, aritmetic operators are generally used (ex. i = i+1, i= i-1, i= 2*i)
- While coding synthesizable designs, provide extra care on what hardware would the looping statements create!

For Loop

- First usage: Repeating assignments and declarations following a repetitive pattern, within structural designs

```
assign w[0] = INPUT1;
integer i;

for(i= 0; i < 50; i=i+1)
begin
    assign w[i] = ~w[i-1] ^ INPUT2;
end
```

```
assign w[0] = INPUT1;

assign w[1] = ~w[0] ^ INPUT2;
assign w[2] = ~w[1] ^ INPUT2;
assign w[3] = ~w[2] ^ INPUT2;
assign w[4] = ~w[3] ^ INPUT2;
assign w[5] = ~w[4] ^ INPUT2;
assign w[6] = ~w[5] ^ INPUT2;
assign w[7] = ~w[6] ^ INPUT2;
assign w[8] = ~w[7] ^ INPUT2;
assign w[9] = ~w[8] ^ INPUT2;

// --- And so on...

assign w[47] = ~w[46] ^ INPUT2;
assign w[48] = ~w[47] ^ INPUT2;
assign w[49] = ~w[48] ^ INPUT2;
```

Both do the same thing...

For Loop

- Second usage: Repeating some statements in procedural blocks, very useful within testbenches

```
for(i=0;i<16;i=i+1)
begin

    {a,b,c,d} = i;
    #(wait_time);
    $write("{a,b,c,d}=%d%d%d%d => {f3,f2,f1,f0} = %d%d%d%d -- ",a,b,c,d,f3,f2,f1,f0);
    if({f3,f2,f1,f0} == correct_results[i])
        $display("TRUE");
    else
        $display("FALSE");

end
```

A testbench code piece that gives values between 0-16 to the circuit inputs, in increasing order

For Loop

```
reg [31:0] R [15:0];  // Declare 16 32-bit large regs named R

always@(posedge CLK or negedge RST)
begin

    if(!RST)
    begin
        for(i=0;i<16;i=i+1)
            R[i] <= 0;
    end
    else if(WE)
    begin
        R[ADDR] <= NVR_DIN;
    end

end
```

Same...

```
always@(posedge CLK or negedge RST)
begin

    if(!RST)
    begin
            R[0] <= 0;
            R[1] <= 0;
            R[2] <= 0;
            R[3] <= 0;
            R[4] <= 0;
            R[5] <= 0;
            R[6] <= 0;
            R[7] <= 0;
            R[8] <= 0;
            R[9] <= 0;
            R[10] <= 0;
            R[11] <= 0;
            R[12] <= 0;
            R[13] <= 0;
            R[14] <= 0;
            R[15] <= 0;
    end
    else if(WE)
    begin
        R[ADDR] <= NVR_DIN;
    end

end
```

A sequential design example, for loop is used to reset all R values when negative edge of RST signal is captured.

Generate Blocks

- Third usage: Within generate blocks

```
genvar i;

generate

// loop emploting i variable
    // generate multiple module instances

endgenerate
```

• Generate blocks are used to create multiple module instances following a repeatitive rule, or for conditional instantiations

• Within the loops inside generate block, a special data type named “genvar” is used.

Generate Blocks

- Generate statements are convenient when the same module instance is repeated.

```
module bitwise_xor( out, i0, i1 );

parameter N = 32;

output [N-1:0] out;
input [N-1:0] i0, i1;

genvar j;   // temp loop variable, used only
            // in the evaluation of the generate blocks

generate
for( j=0; j<N; j=j+1 )
    begin : xor_loop
        xor g1( out[j], i0[j], i1[j] );
    end
endgenerate
```

❖ Creating 32 XOR primitives using generate block

Generate Conditional

- A generate conditional is used for conditionally instantiation.

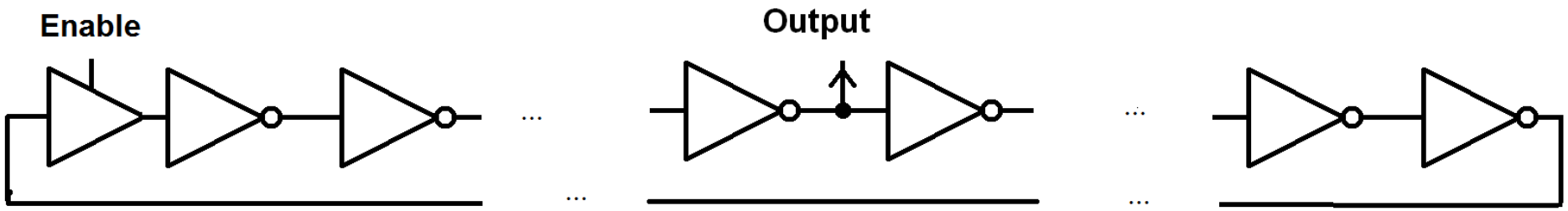
```
module multiplier( product, a0, a1 );

parameter a0_width = 8;
parameter a1_width = 8;
parameter product_width = a0_width + a1_width;

output [product_width-1:0] product;
input [a0_width-1:0] a0;
input [a1_width-1:0] a1;

// Instantiate the type of multiplier conditionally.
generate
    if( a0_width < 8) || (a1_width < 8) )
        cla_multiplier #(a0_width, a1_width) m0 (product, a0, a1);
    else
        tree_multiplier #(a0_width, a1_width) m0 (product, a0, a1);
endgenerate

endmodule
```

Generate Case

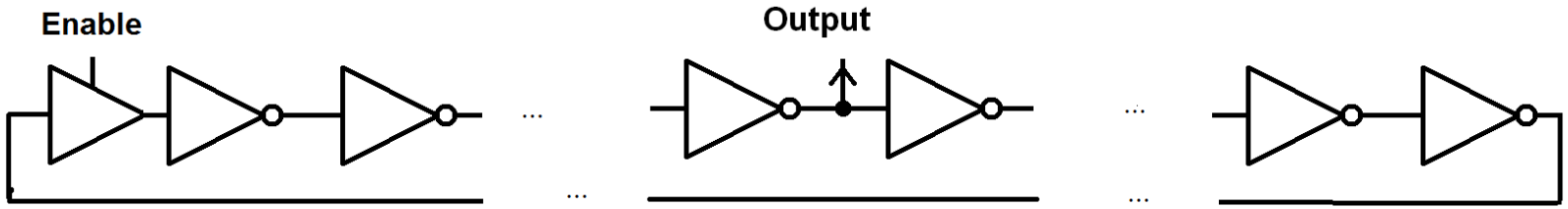
```
module adder( co, sum, a0, a1, ci );

parameter N = 4;

output co;
output [N-1:0] sum;
input [N-1:0] a0, a1;
input ci;

// Instantiate the appropriate adder based on the width of the bus.
// This is based on parameter N that can be redefined at
// instantiation time.
generate
    case(N)
        1: adder_1bit adder1( c0, sum, a0, a1, ci );
        2: adder_2bit adder2( c0, sum, a0, a1, ci );
        default: adder_cla #(N) adder3(c0, sum, a0, a1, ci);
    endcase
endgenerate

endmodule
```

Design Example: Ring Oscillator

- A **ring oscillator** is composed of an odd number of NOT gates. Its output oscillates between two voltage levels.



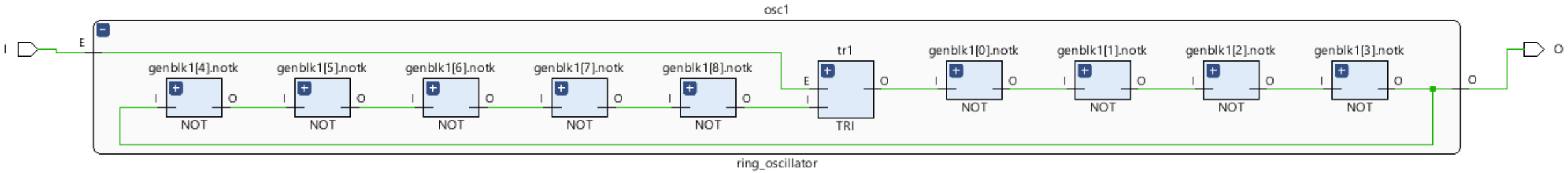
Design Example: Ring Oscillator



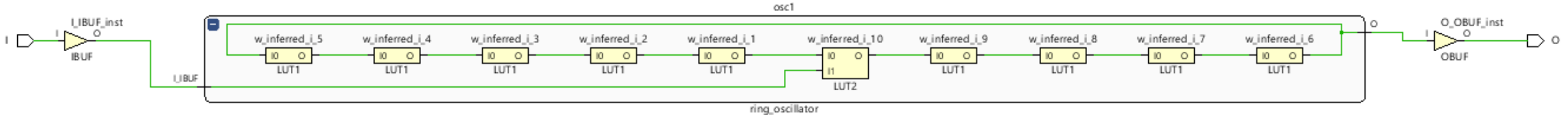

```
module ring_oscillator #( parameter size = 100) ( input E, output O );

(* dont_touch="true" *) wire [size-1:0]w;

genvar i;

generate
   for(i=0; i<size-1; i=i+1)
   begin
      NOT notk(.I(w[i]), .O(w[i+1]));
   end
endgenerate

TRI tr1(.I(w[size-1]), .E(E), .O(w[0]));

assign O = w[size/2 -1];

endmodule
```

Design Example: Ring Oscillator

RTL Schematic (size=10) :



Technology Schematic:



References

- *Verilog HDL: A Guide to Digital Design and Synthesis, Second Edition, Samir Palnitkar*
- *https://learn.sparkfun.com/tutorials/logicblocks-experiment-guide/5-ring-oscillator*