

# Introducción a SystemVerilog

Arquitectura de Computadoras 2023



# HDL to Gates

## Simulation

- Inputs applied to circuit.
- Outputs checked for correctness.
- Millions of dollars saved by debugging in simulation instead of hardware.

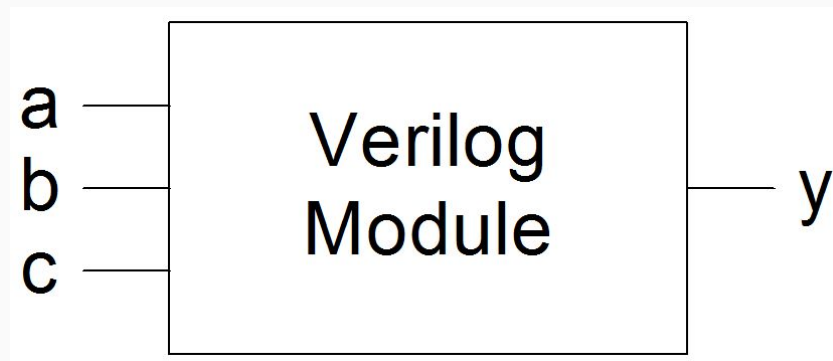
## Synthesis

- Transforms HDL code into a netlist describing the hardware (i.e., a list of gates and the wires connecting them).
- The logic synthesizer might perform optimizations to reduce the amount of hardware required.

# SystemVerilog Modules

## Two types of Modules:

- Behavioral: describe what a module does.
- Structural: describe how it is built from simpler modules.



# Behavioral SystemVerilog

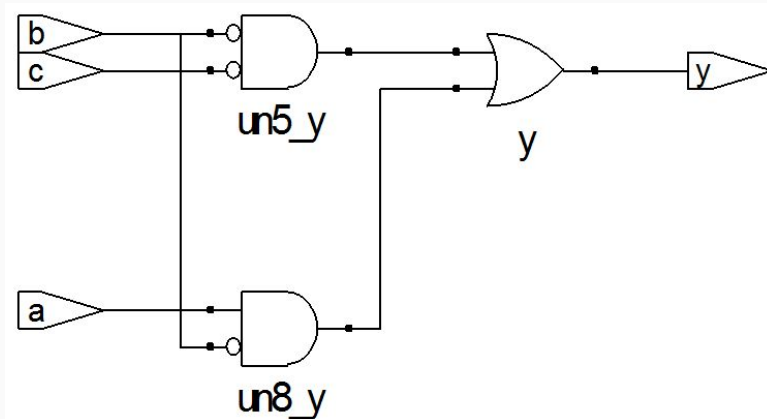
**Behavioral modeling:** describing a module in terms of the relationships between inputs and outputs.

- `module/endmodule`: required to begin/end module.
- `Module` begins with the name (`example`) and a listing of the inputs/outputs.
- `Logic` signals such as the inputs and outputs are Boolean variables (0 or 1). They may also have floating and undefined values.
- The `assign` statement describes combinational logic.

```
// Boolean function  $y = a'b'c' + ab'c' + ab'c$   
module example(input  logic a, b, c,  
               output logic y);  
    assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b & c;  
endmodule
```

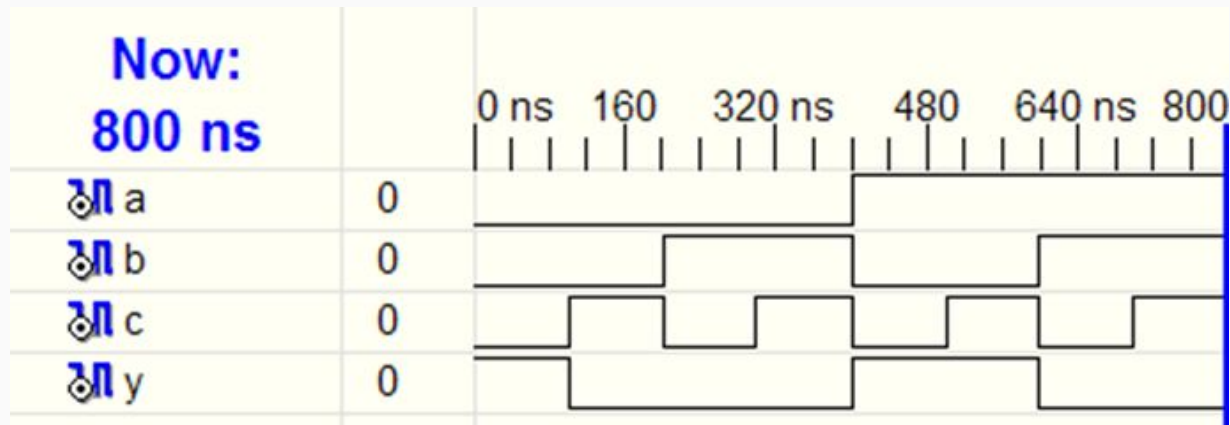
# HDL Synthesis

```
// Boolean function  $y = a'b'c' + ab'c' + ab'c$   
module example(input  logic a, b, c,  
               output logic y);  
    assign y = ~a & ~b & ~c | a & ~b & ~c | a & ~b & c;  
endmodule
```



# HDL Simulation

```
// Boolean function  $y = a'b'c' + ab'c' + ab'c$   
module example(input logic a, b, c,  
               output logic y);  
    assign y = ~a&~b&~c | a&~b&~c | a&~b&c;  
endmodule
```



a	b	c	y
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

# SystemVerilog Syntax

- **Case sensitive**

Example: reset and Reset are not the same signal.

- **No names that start with numbers**

Example: 2mux is an invalid name.

- **Whitespace ignored**

- **Comments:**

```
// single line comment  
/* multiline  
   comment */
```

# SystemVerilog Operators and precedence

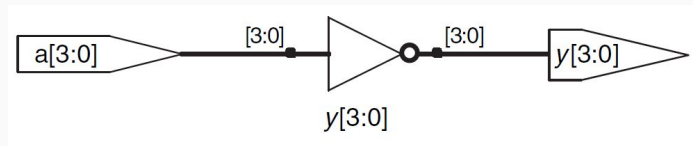
	Op	Meaning
H i g h e s t	~	NOT
	*, /, %	MUL, DIV, MOD
	+, -	PLUS, MINUS
	<<, >>	Logical Left/Right Shift
	<<<, >>>	Arithmetic Left/Right Shift
	<, <=, >, >=	Relative Comparison
	==, !=	Equality Comparison
L o w e s t	&	AND
	^	XOR
		OR
	?:	Conditional



# Bitwise Operators

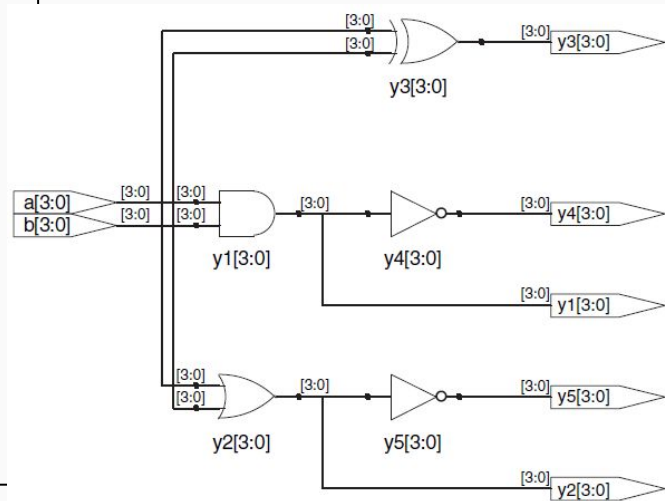
- Bitwise operators act on single-bit signals or on multi-bit busses.
- $a[3:0]$  represents a 4-bit bus. The bits, from most significant to least significant, are  $a[3]$ ,  $a[2]$ ,  $a[1]$  and  $a[0]$ . The least significant bit has the smallest bit number.

```
module inv    (input logic [3:0] a,  
               output logic [3:0] y);  
    assign y = ~a;  
endmodule
```



# Bitwise Operators

```
module gates(input  logic [3:0]  a, b,  
             output logic [3:0] y1, y2, y3, y4, y5);  
  
    /* Five different two-input logic  
       gates acting on 4 bit busses */  
  
    assign y1 = a & b; // AND  
    assign y2 = a | b; // OR  
    assign y3 = a ^ b; // XOR  
    assign y4 = ~(a & b); // NAND  
    assign y5 = ~(a | b); // NOR  
  
endmodule
```

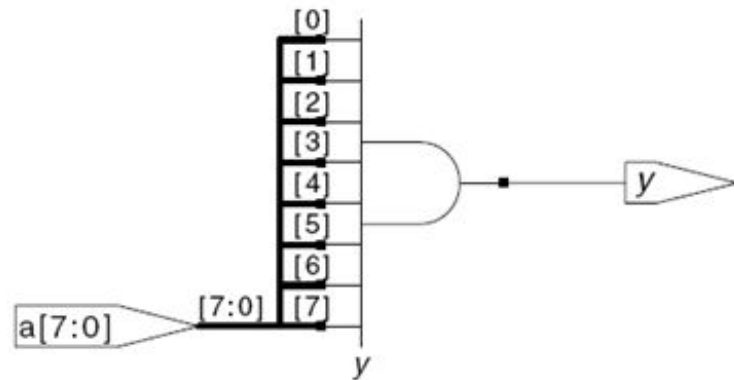


- Anytime the inputs on the right side of the `=` in a continuous assignment statement change, the output on the left side is recomputed. Thus, continuous assignment statements describe **combinational logic**.

# Reduction Operators

- Reduction operators imply a multiple-input gate acting on a single bus.

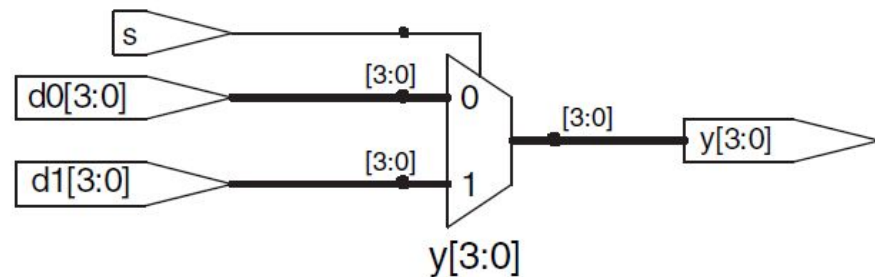
```
module and8(input  logic [7:0] a,  
            output logic  y);  
    assign y = &a;  
    // &a is much easier to write than  
    // assign y = a[7]&a[6]&a[5]&a[4]&  
    //           a[3]&a[2]&a[1]&a[0];  
endmodule
```



# Conditional Assignment

- Conditional assignments select the output from among alternatives based on an input called the condition.
- The conditional operator `? :` is called a ternary operator, because it takes three inputs

```
module mux2(input  logic [3:0] d0, d1,  
            input  logic      s,  
            output logic [3:0] y);  
  
    assign y = s ? d1 : d0;  
  
endmodule
```



# Nested conditional operators

```
// 4:1 multiplexer
module mux4 (input logic [3:0] d0, d1, d2, d3,
             input logic [1:0] s,
             output logic [3:0] y);

    assign y = s[1] ? (s[0] ? d3 : d2) : (s[0] ? d1 : d0);
endmodule
```

/\* If s[1] is 1, then the multiplexer chooses the first expression, (s[0] ? d3 : d2). This expression in turn chooses either d3 or d2 based on s[0] (y = d3 if s[0] is 1 and d2 if s[0] is 0). If s[1] is 0, then the multiplexer similarly chooses the second expression, which gives either d1 or d0 based on s[0]. \*/

# Numbers

## Format: `N'Bvalue`

- **N** = number of bits, **B** = base.
- SystemVerilog supports '**b**' for binary, '**o**' for octal, '**d**' for decimal and '**h**' for hexadecimal.
- **N'B** is optional but recommended (default is decimal).
- '**0**' and '**1**': filling a bus with all 0s or all 1s, respectively.
  
- **z**: indicate a floating value.
- **x**: indicate an invalid logic level.

# Numbers - Examples

Numbers	Bits	Base	Val	Stored
3'b101	3	2	5	101
'b11	?	2	3	000 ... 0011
8'b11	8	2	3	00000011
8'b1010_1011	8	2	171	10101011
3'd6	3	10	6	110
6'o42	6	8	34	100010
8'hAB	8	16	171	10101011
42	?	10	42	00 ... 0101010

# Bit Manipulations - Example

```
assign y = {a[2:1], {3{b[0]}}}, a[0], 6'b100_010};  
// if y is a 12-bit signal, the above statement produces:  
y = a[2] a[1] b[0] b[0] b[0] a[0] 1 0 0 0 1 0  
  
// underscores (_) are used for formatting only to make  
// it easier to read. SystemVerilog ignores them.
```

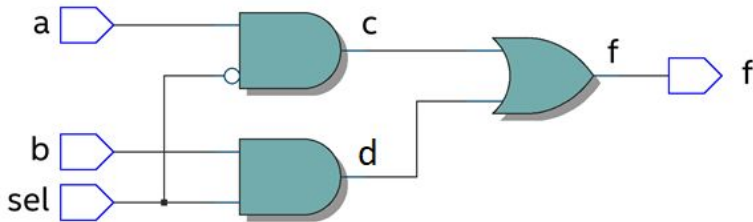
- The {} operator is used to concatenate busses.
- {3{b[0]}} indicates three copies of b[0].
- If y were wider than 12 bits, zeros would be placed in the most significant bits.



# SV Modules - Behavioral vs Structural type

```
// exampleB
```

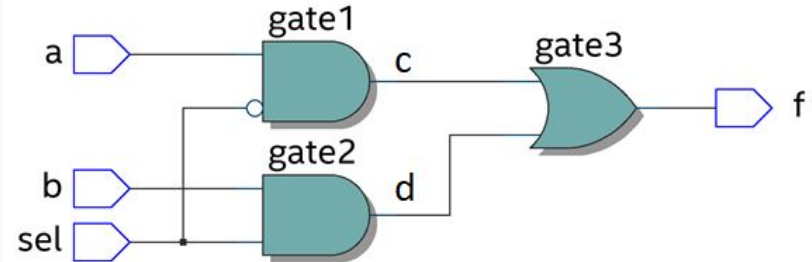
```
module exampleB
    (input logic a, b, sel,
     output logic f);
    logic c, d;
    assign c = a & (~sel);
    assign d = b & sel;
    assign f = c | d;
endmodule
```



```
// exampleS
```

```
module exampleS(input logic a, b, sel,
                output logic f);
    logic c, d, not_sel;
    not gate0(not_sel, sel);
    and gate1(c, a, not_sel);
    and gate2(d, b, sel);
    or gate3(f, c, d);
endmodule
```

\* Built-in gates  
Port order is: output, input(s)



# Structural Modeling - Hierarchy & Internal variables

Describing a module in terms of how it is composed of simpler modules.

```
module mux2(input  logic [3:0] d20, d21,  
            input  logic  s2,  
            output logic [3:0] y2);  
  
    assign y2 = s2 ? d21 : d20;  
  
endmodule
```

```
/* Assemble a 4:1 multiplexer from  
three 2:1 multiplexers */
```

```
module mux4(input logic [3:0] d0, d1, d2, d3,  
            input logic [1:0] s,  
            output logic [3:0] y);
```

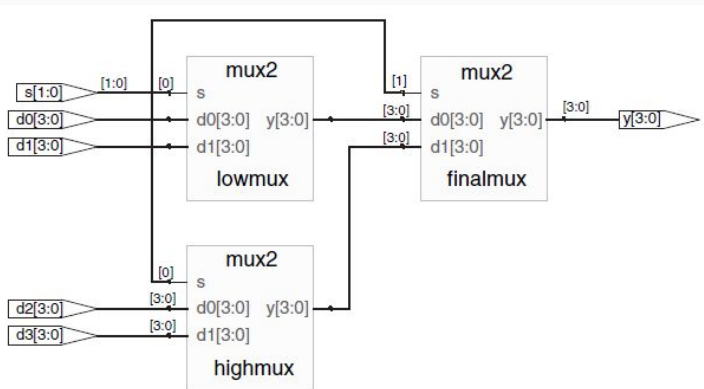
```
    logic [3:0] low, high;    // internal variables
```

```
    mux2 lowmux(d0, d1, s[0], low);    // instance of mux2
```

```
    mux2 highmux(d2, d3, s[0], high);  // instance of mux2
```

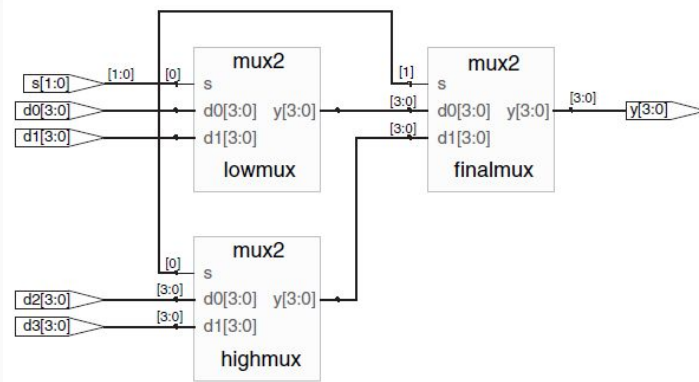
```
    mux2 finalmux(low, high, s[1], y); // instance of mux2
```

```
endmodule
```



# Structural Modeling - Hierarchy & Internal variables

```
module mux2(input  logic [3:0] d20, d21,  
            input  logic  s2,  
            output logic [3:0] y2);  
    assign y2 = s2 ? d21 : d20;  
endmodule
```



```
module mux4(input logic [3:0] d0, d1, d2, d3,  
            input logic [1:0] s,  
            output logic [3:0] y);  
  
    logic [3:0] low, high;    // internal variables  
  
    mux2 lowmux(.d20(d0), .d21(d1), .s2(s[0]), .y2(low)); // instance of mux2  
    mux2 highmux(.s2(s[0]), .y2(high), .d20(d2), .d21(d3)); // instance of mux2  
    mux2 finalmux(.y2(y), .d21(high), .d20(low), .s2(s[1])); // instance of mux2  
endmodule
```

# Accessing parts of busses

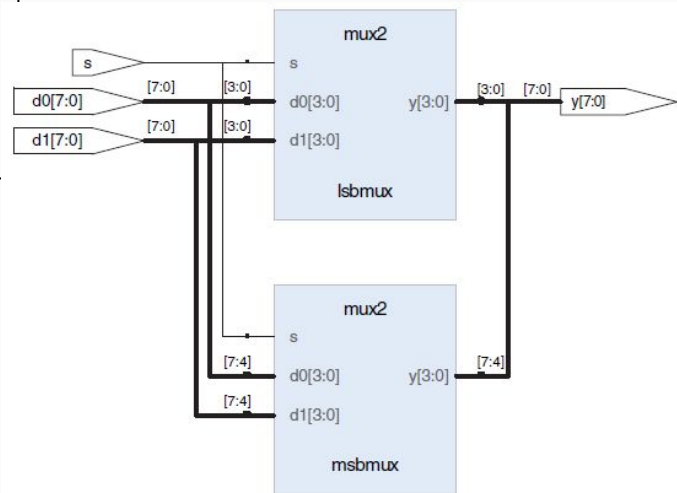
```
/* Assemble an 8-bit wide 2:1 multiplexer  
using two 4-bit 2:1 multiplexers */
```

```
module mux2_8(input logic [7:0] d0, d1,  
              input logic s,  
              output logic [7:0] y);
```

```
    mux2 lsbmux(d0[3:0], d1[3:0], s, y[3:0]);
```

```
    mux2 msbmux(d0[7:4], d1[7:4], s, y[7:4]);
```

```
endmodule
```



# *always* procedural block

In SystemVerilog `always` statements signals keep their old value until an event in the sensitivity list takes place that explicitly causes them to change.

## **General Structure:**

```
always @(sensitivity list)  
    statement;
```

Whenever the event in `sensitivity list` **occurs**, `statement` **is executed**

# *always* - Combinational Logic

To represent **combinational logic** with a general purpose always procedural block:

- The always keyword must be followed by an event control (the @ token).
- The sensitivity list of the event control cannot contain posedge or negedge qualifiers.
- The sensitivity list should include all inputs to the procedural block.
- The procedural block cannot contain any other event controls.
- All variables written to by the procedural block must be updated for all possible input conditions.
- Any variables written to by the procedural block cannot be written to by any other procedural block.

# Specialized *always* procedural block

**always\_comb**, **always\_latch** and **always\_ff**

procedural blocks indicate the design intent.

- With **always\_comb** it is not necessary to specify a sensitivity list, because software tools know that the intent is to represent combinational logic. This inferred sensitivity list includes every signal that is read by the procedural block.
  - An **always\_comb** procedural block will automatically trigger once at simulation time zero.
- If the content of a specialized procedural block does not match the rules for that type of logic, software tools can issue warning messages.

# Combinational Logic - *always\_comb*

```
module gates(input  logic [3:0] a, b,  
             output logic [3:0] y1, y2, y3, y4, y5);  
    always_comb    // need begin/end because there is  
        begin      // more than one statement in always  
            y1 = a & b;    // AND  
            y2 = a | b;    // OR  
            y3 = a ^ b;    // XOR  
            y4 = ~(a & b); // NAND  
            y5 = ~(a | b); // NOR  
        end  
endmodule
```



# *always\_comb* problem

SystemVerilog example:

```
always_comb  
if (en) y = a;
```

- Software tools can issue a warning that a latch would be required to realize the procedural block's functionality in hardware.
- The correct way to model the example as combinational logic would be to **include an else** branch so that the output 'y' would be updated for **all conditions** of 'en'.

# *always* - Sequential Logic (with Flip-Flops)

To represent **sequential logic** with a general purpose always procedural block:

- The `always` keyword must be followed by an edge-sensitive event control (the `@` token).
- All signals in the event control sensitivity list must be qualified with `posedge` or `negedge` qualifiers.
- The procedural block cannot contain any other event controls.
- Any variables written to by the procedural block cannot be written to by any other procedural block.

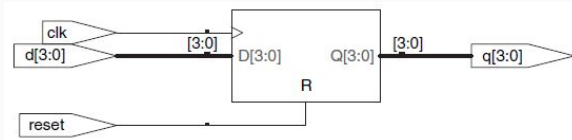
# Sequential Logic - *always\_ff*

```
always_ff @(posedge clock, negedge resetN)
    if (!resetN) q <= 0;
    else q <= d;
```

- The `always_ff` procedural block requires that every signal in the sensitivity list must be qualified with either **posedge** or **negedge**. This is a synthesis requirement for sequential logic sensitivity list.
- Making this rule a syntactical requirement helps ensure that simulation results will match synthesis results.

# Sequential Logic - Resettable register

Asynchronous reset:



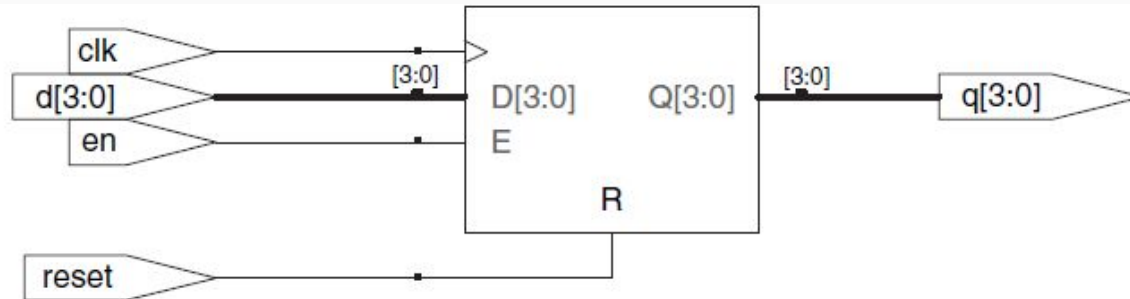
Synchronous reset:

```
module flopr(input logic clk,
             input logic reset,
             input logic [3:0] d,
             output logic [3:0] q);
    always_ff @(posedge clk, posedge reset)
        if (reset) q <= 4'b0;
        else q <= d;
endmodule
```

```
module floprsync(input logic clk,
                 input logic reset,
                 input logic [3:0] d,
                 output logic [3:0] q);
    always_ff @(posedge clk)
        if (reset) q <= 4'b0;
        else q <= d;
endmodule
```

# Sequential Logic - Resettable enabled register

```
module flopenr(input logic clk,  
               input logic reset,  
               input logic en,  
               input logic [3:0] d,  
               output logic [3:0] q);  
    always_ff @(posedge clk, posedge reset)  
        if (reset) q <= 4'b0;  
        else if (en) q <= d;  
endmodule
```

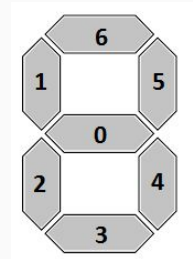


# case statement - Combinational Logic

```
module sevenseg(input  logic [3:0] data,
                output logic [6:0] segments);

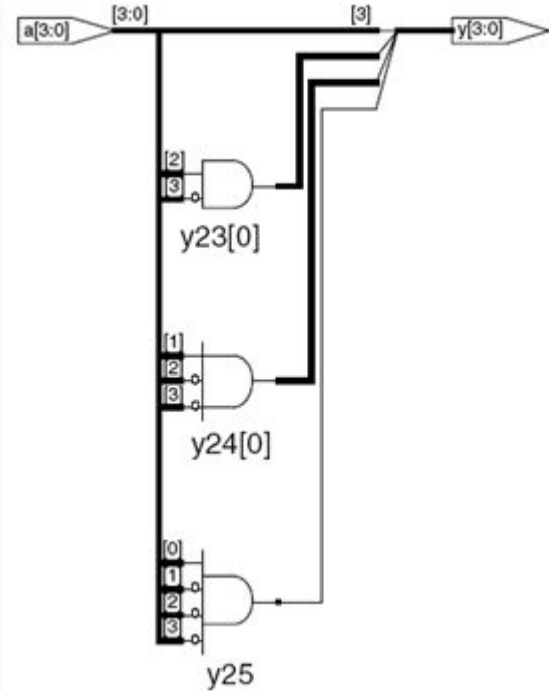
    always_comb
        case (data)
            //                654_3210
            0: segments =    7'b111_1110;
            1: segments =    7'b011_0000;
            2: segments =    7'b110_1101;
            3: segments =    7'b111_1001;
            4: segments =    7'b011_0011;
            5: segments =    7'b101_1011;
            6: segments =    7'b101_1111;
            7: segments =    7'b111_0000;
            8: segments =    7'b111_1111;
            9: segments =    7'b111_0011;
            default: segments = 7'b000_0000; // required
        endcase
endmodule
```

Case statement implies  
combinational logic only if  
all possible input  
combinations described



# casez statement - Combinational Logic

```
module priority_casez(input  logic [3:0] a,  
                     output logic [3:0] y);  
  
    always_comb  
    casez(a)  
        4'b1???: y = 4'b1000; // ?=don't care  
        4'b01??: y = 4'b0100;  
        4'b001?: y = 4'b0010;  
        4'b0001: y = 4'b0001;  
        default: y = 4'b0000;  
    endcase  
endmodule
```

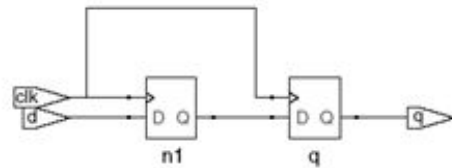


# Assignments inside “always” procedural blocks

- `<=` is nonblocking assignment: occurs simultaneously with others (for sequential logic)
- `=` is blocking assignment: occurs in order it appears in file (for combinational logic)

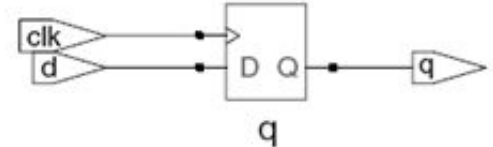
```
// Good synchronizer using
// nonblocking assignments
module syncgood(input  logic clk,
                input  logic d,
                output logic q);

    logic n1;
    always_ff @(posedge clk)
        begin
            n1 <= d; // nonblocking
            q  <= n1; // nonblocking
        end
endmodule
```



```
// Bad synchronizer using
// blocking assignments
module syncbad(input logic
               clk,
               input logic d,
               output logic q);

    logic n1;
    always_ff @(posedge clk)
        begin
            n1 = d; // blocking
            q  = n1; // blocking
        end
endmodule
```





# Parameterized Modules

2:1 mux:

```
module mux2
    #(parameter width = 8)    // name and default value
    (input  logic [width-1:0] d0, d1,
     input  logic s,
     output logic [width-1:0] y);
    assign y = s ? d1 : d0;
endmodule
```

Instance with 8-bit bus width (uses default):

```
mux2 myMux(d0, d1, s, out);
```

Instance with 12-bit bus width:

```
mux2 #(12) lowmux(d0, d1, s, out);
```

# Unpacked arrays

Basic syntax of an unpacked array declaration is:

```
<data_type> <vector_size> <array_name> <array_dimensions>
```

For example:

```
logic [7:0] table [3:0]; // array "table"  
                        // has 4 elements of 8 bit
```

- With unpacked arrays, each element of the array may be stored independently from other elements.
- Use unpacked arrays to model arrays where typically one element at a time is accessed, such as with RAMs and ROMs.

# Unpacked array initialization

- Unpacked arrays can be initialized at declaration, using a list of values enclosed between '{ and } braces for each array dimension.
- The assignment requires nested sets of braces that **exactly match the dimensions of the array**.

```
logic [31:0] ROM [0:5] = '{32'h8b020021,  
                           32'h8b000000,  
                           32'h8b000000,  
                           32'hf8008001,  
                           32'h91000461,  
                           32'h8b000000}';
```

# Unpacked array initialization examples

```
        logic [0:7] a1 [0:1023] = '{default:8'h55};  
/* Specifying a default value to initialize all the elements, or a slice, of  
an unpacked array*/
```

```
logic [3:0] a [0:3];
```

- `a = '{0,1,0,1}, '{1,0,0,0}, '{1,1,1,1}, '{0,0,1,0}};`  
    `// assign a list of values to the full array`
- `a[3] = 4'h5;`      `// assign list of values to slice of the array`
- `a[1][0] = 1'b1;`   `// assign to one element`

```
logic a1 [7:0][1023:0];    // unpacked array
```

```
logic a2 [8:1][1024:1];    // unpacked array
```

- `a2 = a1;`            `// copy an entire array`
- `a2[3] = a1[0];`      `// copy a slice of an array`

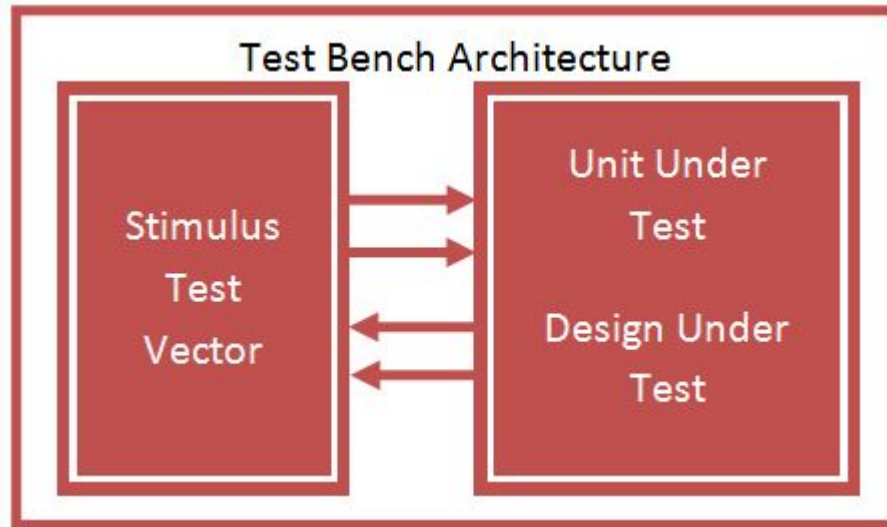
# Test benches

The synthesizable modules describe the hardware. The test bench contains code to apply inputs to a module, check whether the output results are correct, and print discrepancies between expected and actual outputs.

- HDL module that tests another module, the device under test (dut)
- Not synthesizable
- Types:
  - ➔ Simple
  - ➔ Self-checking
  - ➔ Self-checking with test vectors

# Test benches

- The testbench contains statements to apply inputs to the **DUT** and, ideally, to check that the correct outputs are produced.
- The input and desired output patterns are called **test vectors**.



# Test bench - Example

Write SystemVerilog code to implement the following function in hardware:

$$y = b'c' + ab'$$

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

# Simple Test bench

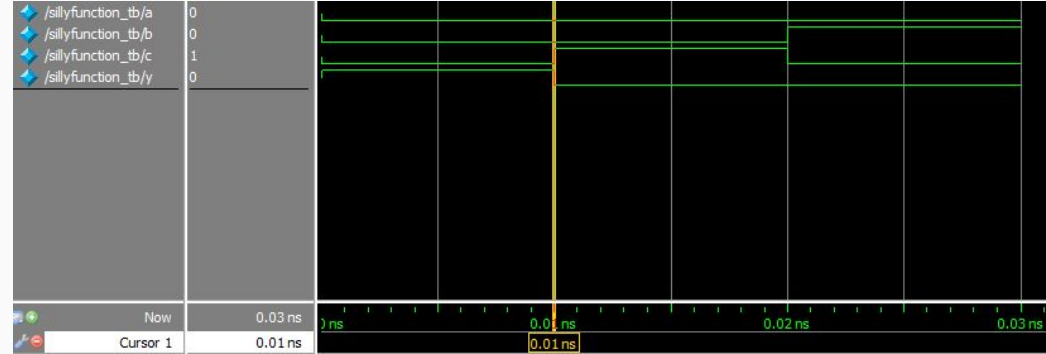
```
module testbench1 ();  
    logic a, b, c; // internal variables  
    logic y;  
    // instantiate device under test:  
    sillyfunction dut(a, b, c, y);  
    // apply inputs one at a time:  
    initial begin  
        a = 0; b = 0; c = 0; #10;  
        c = 1; #10;  
        b = 1; c = 0; #10;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



# Test bench - Delays

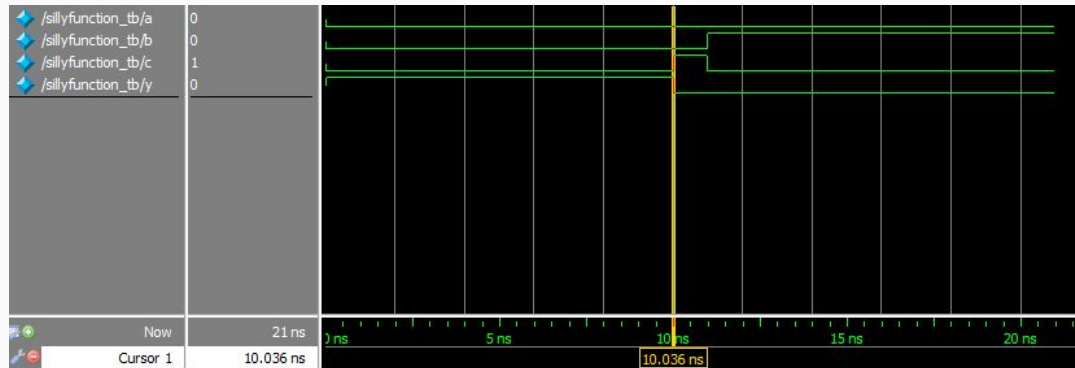
- A # symbol is used to indicate the number of units of delay.
- Default time scale unit in Quartus Prime = 1ps.

```
module sillyfunction_tb();  
  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;  
  end  
endmodule
```



# Test bench - Delays

```
module sillyfunction_tb();  
  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 ns;  
    c = 1; #1000;  
    b = 1; c = 0; #10 ns;  
  end  
endmodule
```

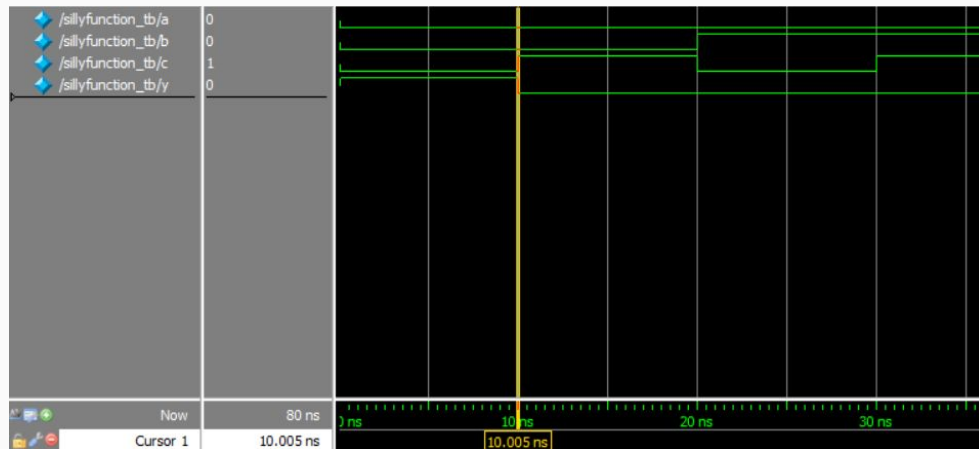


# Test bench - Delays

- 'timescale 1ns / 10ps: the software tool is instructed to use time units of 1 nanosecond, and a precision of 10 picoseconds

```
'timescale 1ns / 10ps
```

```
module sillyfunction_tb();  
  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;  
  end  
endmodule
```



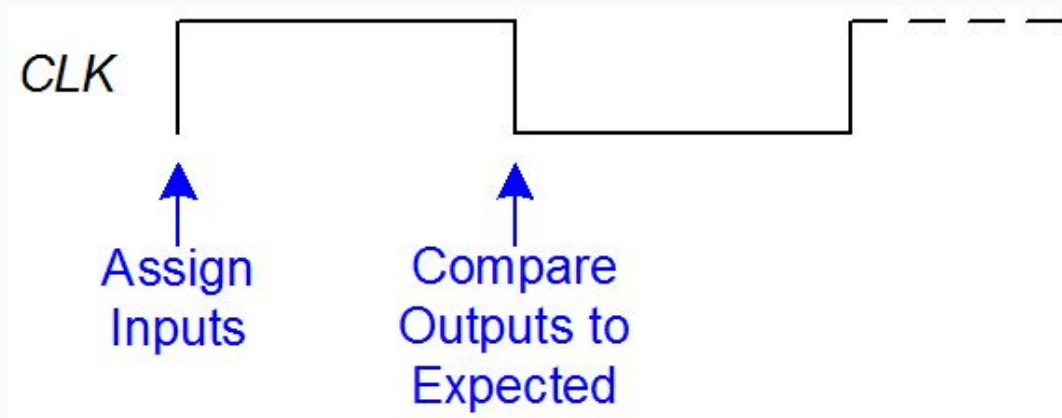
# Self-checking Test bench

```
module testbench2();  
    logic a, b, c;  
    logic y;  
    // instantiate dut  
    sillyfunction dut(a, b, c, y);  
  
    /* apply inputs and check results  
       one at a time */  
    initial begin  
        a = 0; b = 0; c = 0; #10;  
        if (y !== 1) $display("000 failed.");  
        c = 1; #10;  
        if (y !== 0) $display("001 failed.");  
        b = 1; c = 0; #10;  
        if (y !== 0) $display("010 failed.");  
        c = 1; #10;  
        ...  
    end  
endmodule
```

```
...  
    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;  
    ...  
endmodule
```

# Test bench with Test vectors (TwT)

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



- Test bench clock also used as clock for synchronous sequential circuits.

## TwT 1. Read Test vectors into Array

[illegible]

# TwT 2. Generate Clock

```
// 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

initial    // at start of test pulse reset
begin
    vectornum = 0; errors = 0; reset = 1; #27;
    reset = 0;
end

...

```

## TwT 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
```

```
...
```



# TwT 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});
```

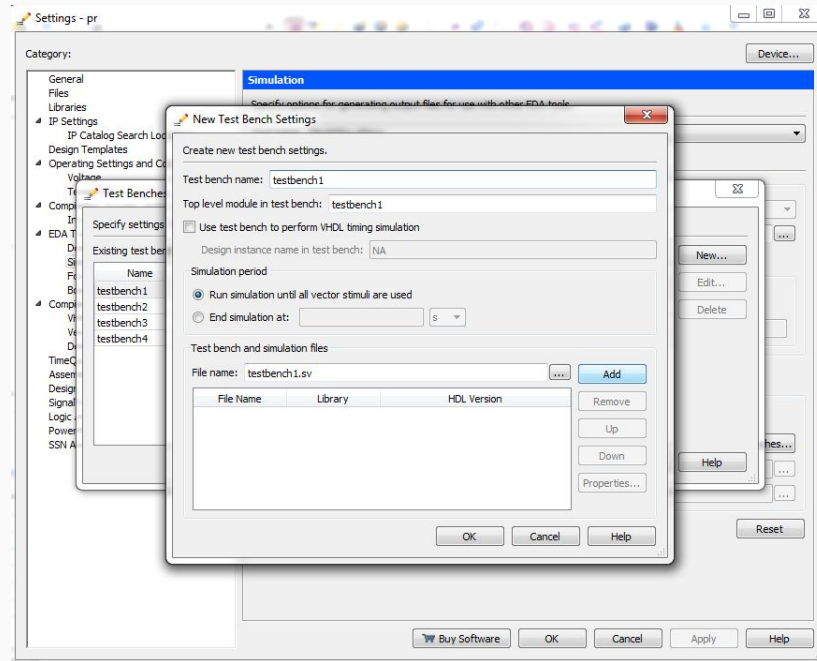
## TwT 4. Compare with Expected Outputs

```
// 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; // Usar $stop para que no se cierre ModelSim
        end
    end
endmodule




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

# Quartus test bench configuration

- Assignments > Settings > Simulation > Compile test bench
- Click on Test Benches... > New... > ...
- Add testbench file.



# sv Test bench - Quartus synthesis and simulation

- Analysis & Synthesis: 
- ~~Complete Synthesis: Processing > Start Compilation~~ 
- Simulation: Tools > Run Simulation Tool > RTL Simulation 

# Bibliografía

- S. Harris and D. Harris, “Digital Design and Computer Architecture - ARM Edition”. Elsevier, 2016.
- S. Sutherland, S. Davidmann, P. Flake and P. Moorby, “SystemVerilog for Design - A Guide to Using SystemVerilog for Hardware Design and Modeling”, Second Edition. Springer, 2006.