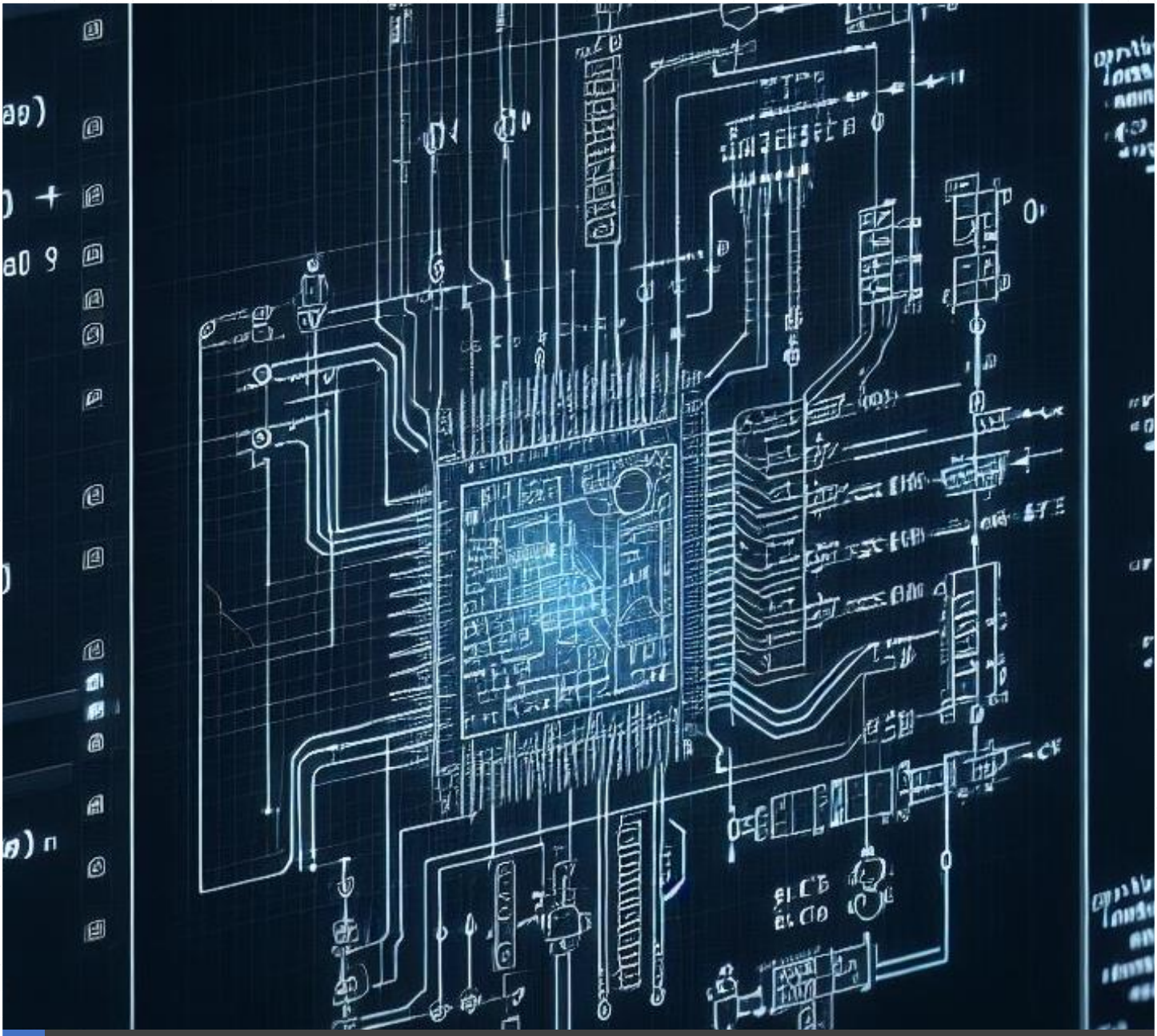


Verilog Tutorial

Part 1

Sameh Mohamed
GitHub: SamehM20
LinkedIn: Sameh Elbatsh

Contents

1. Module:

A module is a set of logic elements with inputs and outputs that do certain functions. Consider a module as a block that has a set of inputs and outputs encloses some logic.

The most common syntax of module is:


```
module [design_name] (
        [list_of_inputs],
        [list_of_outputs]
);
[inner_signals_declarations]
[instantiation_of_other_modules]
[code_segment_behavioral_or_assign]
endmodule
```

We will illustrate each part as we continue.

2. Naming Rules:

Verilog is a case sensitive language (testv ≠ Testv). Variables can be up to 1024 characters. The format can be like:

- var
- var123
- var_62
- _var

But can't start with a number nor a dollar sign '$' (as it's reserved for Verilog functions) and it can't contain a space. The following are wrong:

- var 1
- 56var
- $var

3. Commenting:

- For a single line comment use: // This is a comment.
- For a block comment use:

```
/* This is
   a block
   comment
*/
```

4. Number Format:

The default format for writing a number is in decimal format (ex: x = 5). Verilog implicitly convert its value to binary value. To express a value in a specific format, put it as in form:

[size]'[base][number]

[size]: number of bits that represents the number in binary.

[base]: the base of the numbering system:

b → binary, **d** → decimal, **h** → hexadecimal, **o** → octal.

[number]: the number represented in the [base] format.

For example:

4'**b**0010 → 0010 → 2 in decimal.

5'**b**11 → 00011 → 3 in decimal. Higher bits are padded with zero.

8'**h**0A → 0000 1010 → 10 in decimal.

8'**h**27 → 0010 0111 → 39 in decimal.

Note: '_' is ignored when it's put in a number. It is a way to make large number reading easy. For example:

1110_1101 = 11101101

11_01_1_0 = 110110

5. Inputs and Outputs Declaration:

- **input**: represents an input signal.
- **output**: represents an output signal.
- **inout**: represent a signal that can operate as an input or an output. It's usually used with Data/Address buses.

They are implicitly declared as wires. If you want another type, you should put that type after the keyword. Ex: **output reg** y.

Or declare a net with the same name with the new type. Ex:

module …….

output y

…….);

reg y;

……..

endmodule

6. Nets and Variables:

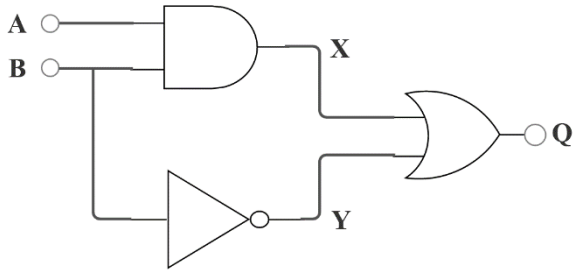
wire: it doesn't store a value and can be used outside a behavioral block.

reg: can store a value and it is used inside behavioral blocks. It may not represent a real hardware register.

tri: rarely used. Represents a tristate buffer.

7. Example:

Consider the following circuit.



A	B	X	Y	Q
0	0	0	1	1
0	1	0	0	0
1	0	0	1	1
1	1	1	0	1

It has two inputs (A, B), one output (Q), and two inner nets (X, Y). To represents these circuit signals (nets), one way is to write:

```
// Choosing the module name to be circuit_1
module circuit_1 (
        input A, B, // Declaring the inputs.
        output Q    // Declaring the outputs.

);
wire X, Y;      // Declaring the inner nets.
/*
Logic of the circuit
*/
endmodule
```

Note: Since both inputs are one bit, they can be declared with one input line. The same goes with the inner nets.

8. Data Values:

The value of the data nets can be one of four values:

0: represents logic low.

1: represents logic high.

Z or **z**: is high impedance (the net is not driving anything, dangling)

X or **x**: is unknown (the value can be 1 or 0)

9. Scaler and Vector:

Until now, we have been mentioning scaler signals.

Scaler: one bit signal.

Vector: multiple bits signal (a bus or a one-dimension array).

To declare a vector signal, [MSB : LSB] is added before the name where MSB is a number representing the most significant bit index and LSB is a number representing the least significant bit index. Ex:

input [7:0] b → ($b_7b_6b_5b_4b_3b_2b_1b_0$)

output [2:6] c → ($c_2c_3c_4c_5c_6$)

reg [6:1] x → ($x_6x_5x_4x_3x_2x_1$)

Bit-Select: selecting a single bit from a vector. Ex: for x from before

x[3] = 1; → (will assign 1 to the bit x_3)

Part-Select: selecting part of the vector. Ex: for x from before

x[4:2] = 3'b101; → (will assign 1 to the bit x_4, 0 to the bit x_3, and 1 to the bit x_2)

10. Synthesis:

Synthesis: is the operation of converting the Verilog code to a hardware block of the used library. Simply, it converts the code into logic blocks that performs that functionality. Not all Verilog codes are **synthesizable** which means that the tool won't be able to convert it to logic blocks.

11. Operators:

The are various categories of operators in Verilog:

a. Arithmetic:

'+': Addition, a + b

'-': Subtraction, a - b

'*': Multiplication, a * b

'**': Power, (a ** b) means a^b

'/': Division, a / b

'%': Modulo, a % b

N.B.: The division and the Modulo operators are not synthesizable which means you must do their logic yourself. They can, however, be used in testbenches for example.

b. Relational:

The output of the expression is either **True** (**1**) or **False** (**0**). If either of the operands is **X** or **Z**, the output will be **X** (**unknown**).

'<': Less than, a < b

'<=': Less than or equal, a <= b

'>': Greater than, a > b

'>=': Greater than or equal, a >= b

c. Equality:

The output of the expression is either **True** (**1**) or **False** (**0**).

If either of the operands is **X** or **Z**, the output will be **X** (**unknown**).

'==': Equal to, a == b
'**!**=': Not equal to, a **!**= b

For these two, even **Z** and **X** is used for matching.

'===': Equal to, a === b
'**!**==': Not equal to, a **!**== b

Example1: A = 3'**b**101; B = 3'**b**101; C = 3'**b**111

A == B → **1** (**True**) A == C → **0** (**False**)

A === B → **1** (**True**) A === C → **0** (**False**)

A **!**= B → **0** (**False**) A **!**= C → **1** (**True**)

A **!**== B → **0** (**False**) A **!**== C → **1** (**True**)

Example2: A = 3'**b**10Z; B = 3'**b**10Z; C = 3'**b**10X

A == B → **X** (**Unknown**) A == C → **X** (**Unknown**)

A === B → **1** (**True**) A === C → **0** (**False**)

A **!**= B → **X** (**Unknown**) A **!**= C → **X** (**Unknown**)

A **!**== B → **0** (**False**) A **!**== C → **1** (**True**)

d. Logical:

'**&&**': Evaluates to true if both operands are true, a **&&** b

'||': Evaluates to true if any operand is true, a || b

'**!**': Converts non-zero value to zero, and zero to one, **!**a

N.B.: Logical operators treat non-zero values as True and zero as False. For ex: !3'b101 = 0; !165 = 0. Also, in '&&' operator if one of the operands is X, the output is X, but if it is Z, it will treat it as a True value.

e. Bitwise:

These operators affect the operands bit by bit between two operands. They represent the logic gates.

'**&**': AND, a **&** b

'|': OR, a | b

'~': NOT, one operand, ~ a

'^': XOR, a ^ b

'~^' or '^~': XNOR, a ~^ b, a ^~ b

Examples: A = 4**'b**1001, B = 4**'b**0101

A **&** B = 4**'b**0001 A | B = 4**'b**1101

~ A = 4**'b**0110 A ^ B = 4**'b**1100

A ~^ B = 4**b'**0011 ← The same → A ^~ B = 4**'b**0011

Note the difference between '~' and '!'. They differ on vectors, but they are the same for scalers.

f. Reduction:

These operators affect one operand. They do their bitwise function between the vector bits. They have the same symbols as the bitwise.

'**&**': AND, **&** a

'~**&**': NAND, ~**&** a

'|': OR, | a

'~|': NOR, ~| a

'^': XOR, ^ a

'~^' or '^~': XNOR, ~^ a, ^~ a

Examples: A = 4'b1001

& A = **0** → (1 & 0 & 0 & 1)

~& A = **1** → ~ (1 & 0 & 0 & 1)

| A = **1** → (1 | 0 | 0 | 1)

~| A = **0** → ~ (1 | 0 | 0 | 1)

^ A = **0** → (1 ^ 0 ^ 0 ^ 1)

~^ A = **1** → ~ (1 ^ 0 ^ 0 ^ 1)

g. Shift:

Logical shift: '<<' Shift left A << B,

'>>' Shift right A >> B.

Padding with zeros.

Arithmetic shift: '<<<' Shift left A <<< B,

'>>>' Shift right A >>> B.

Shifting and sign extending.

Example: A = 8'b1000_1110, B = 8'b0110_1001

A << 2 = 8'b0011_1000 B << 2 = 8'b1010_0100

A <<< 2 = 8'b0011_1000 B <<< 2 = 8'b1010_0100

A >> 2 = 8'b0010_0011 B >> 2 = 8'b0001_1010

A >>> 2 = 8'b1110_0011 B >>> 2 = 8'b0001_1010

h. Other Operators:

'**?**': Conditional Operator, S **?** A **:** B, if S is True, the output equals A, else the output equals B. Ex: X = 2, Y = 5

F = (X == Y)**?** A **:** B; → F = B as X doesn't equal Y.

F = (X < Y)**?** A **:** B; → F = A as X is less than Y.

'**{}**': Concatenation, joining signals together binary, {A, B, C}. Ex:

A = 3'**b**101, B = 5'**b**11101

C = {A, B} → C = 10111101

D = {B, A} → D = 11101101

'**{n{}}**': Replication using concatenation operator where n is a positive integer, {5{A}}. Ex: X = 3'**b**110

Y = {4{X}} → Y = 110_110_110_110

12. Design Methods:

There are 3 main design methods: Dataflow, Structural, and Behavioral. Typically, any design can have a combination of them.

13. Dataflow Method:

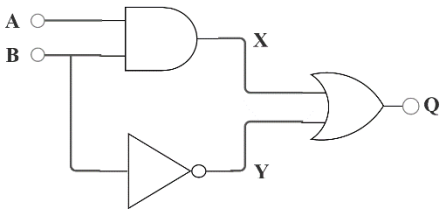
It can be done using the assign expressions or the combinational always block although the assign method is the most common.

Continuous Assignment syntax:

assign [output] = [expression_of_inputs];

For our example from before, the output can be done with one expression or divided into small expressions showing the inner nets.

```
module circuit_1 (
        input A, B,
        output Q
);
wire X, Y; // Declaring the inner nets.
assign X = A & B;
assign Y = ~ B;
assign Q = X | Y;
endmodule
```



Or

```
module circuit_1 (
        input A, B,
        output Q
);
// With one assign expression.
assign Q = (A & B) | (~ B);
endmodule
```

14. Structural Method (Hierarchal):

The design is made of blocks and each block is made from smaller blocks. The lowest level is the **Gate-Level Design**. For our example, since it is simple the top level consists of small level which, here, are gates.

To instantiate another module (block) to use within our module, the syntax is as the following:

[module_name] [instance_name] (input_and_outputs_declared_within)

For example, if we used the previous module in another design, we can instantiate it as the following:

circuit_1 c1 **(**.A(new_A_net), .B(new_B_net), .Q(new_Q_net)**)**

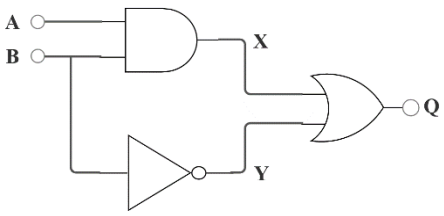
Or if we know the order of the inputs and outputs, we can enter the new nets from top design in order.

circuit_1 c2 **(**new_A_net, new_B_net, new_Q_net**)**

For our example, to make a structural design, it's done using gates. For the logic gates the first entry is the output, then the inputs are entered after it.

Ex: **and** a0 **(**Out, In1, In2, …., InN**)**

```
module circuit_1 (
        input A, B,
        output Q
);
wire X, Y; // Declaring the inner nets.
and a0 (X, A, B);
not n0 (Y, B);
nor n1 (Q, X, Y);
endmodule
```



15. Behavioral Method:

It is a way of coding by describing the logic behavior using logical statements. The main block is the **always** block followed by **@** operator that executes the always block according to the **sensitivity list**. The syntax is as the following:

- For single statement: **always @ (sensitivity_list) statement;**
- For multiple statements:

always @ (sensitivity_list) begin

statement1;

statement2;

……

end

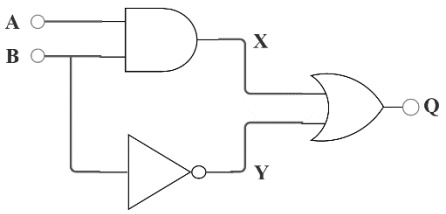
The **begin …. end** part can hold multiple statements and is treated like a single grouped statement.

The **Sensitivity List** is a list of signals that excites the always block. In other words, it is the list of signals that are inputs to the statements in the always block.

N.B.: The Left-Hand Side (LHS, Outputs) of the statements in the always block must be declared as reg.

Revisiting the example:

```
module circuit_1 (
        input A, B,
        output reg Q
);
// Or always @ (*)
always @ (A or B) begin
  Q = (A & B) | (~ B);
end
endmodule
```



N.B.: The above code doesn't represent a behavioral model. It is a sort of a Dataflow model. The following segments will offer a variety of behavioral models for this circuit.

16. Case Statement:

The **Case** statement matches the expression to one of the cases and executes its statements. If no case is matched, the **default** case is executed. It's used within a block. The syntax is as the following:

case (expression)

case_1: statement; (or begin list_of_statemtents end)

case_2, case_3: statement;

……

default: statement;

endcase

N.B.: You have to use the default case to cover all possibilities. If not all of them is covered, a latch is inferred in the synthesized design.

For the previous example, to model the design behaviorally using the **case** statement, we make use of its truth table. We combine the inputs into one vector {A, B} and use it as a selector.

A	B	X	Y	Q
0	0	0	1	1
0	1	0	0	0
1	0	0	1	1
1	1	1	0	1

There are two ways to describe it:

- By listing all possibilities:

```
module circuit_1 (
        input A, B,
        output reg Q
);
always @ (*) begin
  case({A, B})
    2'b00: Q = 1;
    2'b01: Q = 0;
    2'b10: Q = 1;
    2'b11: Q = 1;
    default: Q = 0;
  endcase
end
endmodule
```

-

Another way:

```
module circuit_1 (
        input A, B,
        output reg Q
);
always @ (*) begin
  case({A, B})
    2'b00, 2'b10, 2'b11: Q = 1;
    2'b01: Q = 0;
    default: Q = 0;
  endcase
end
endmodule
```

- By listing the least repeated output (**0**, **1**) and using the **default** case for the others:

```
module circuit_1 (
        input A, B,
        output reg Q
);
always @ (*) begin
  case({A, B})
    2'b01: Q = 0;
    default: Q = 1;
  endcase
end
endmodule
```

17. If…Else Statement:

The syntax of the **If** statement is like in programming languages. The are 3 types of it.

- Single **if** with no **else**: it may infer a latch.

```
if (expression) statement;
```

Or

```
if (expression) begin
    list_of_statements;
end
```

- Single **if** with **else**:

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

Or

```
if (expression) begin
    list_of_statements;
end else begin
    list_of_statements;
end
```

- Multiple **if** with **else** or with no **else**:

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

Or

```
if (expression) begin
    list_of_statements;
end else if begin
    list_of_statements;
end
```

For the previous example, to model the design behaviorally using the **if** statement, we make use of its truth table.

A	B	X	Y	Q
0	0	0	1	1
0	1	0	0	0
1	0	0	1	1
1	1	1	0	1

There are two ways to describe it:

- By listing all possibilities:

```
module circuit_1 (
        input A, B,
        output reg Q
);
always @ (*) begin
  if((A==0)&&(B==0)) Q = 1;
  else if((A==0)&&(B==1)) Q = 0;
  else if((A==1)&&(B==0)) Q = 1;
  else Q = 1;
end
endmodule
```

- By listing the least repeated output of (**0**, **1**) and using the **else** for the others:

```
module circuit_1 (
        input A, B,
        output reg Q
);
always @ (*) begin
  if((A==0)&&(B==1)) Q = 0;
  else Q = 1;
end
endmodule
```