The logic of a module can be described in any one (or a combination) of the following modelling styles:

• Gate-level modeling using instantiations of predefined and user-defined primitive gates.

• Dataflow modeling using continuous assignment statements with the keyword `assign`

• Behavioral modeling using procedural assignment statements with the keyword `always`
Identifiers having multiple bit widths are called vectors.

The syntax specifying a vector includes within square brackets two numbers separated with a colon.

The following Verilog statements specify two vectors:

```verilog
output [0: 3] D;
wire [7: 0] SUM;
```

The first statement declares an output vector D with four bits, 0 through 3.

The second declares a wire vector SUM with eight bits numbered 7 through 0.

(Note: The first (left-most) number (array index) listed is always the most significant bit of the vector.)

The individual bits are specified within square brackets, so D[2] specifies bit 2 of D.

It is also possible to address parts (contiguous bits) of vectors. For example, SUM[2: 0] specifies the three least significant bits of vector SUM.
Note that the keywords not and nand are written only once and do not have to be repeated for each gate, but commas must be inserted at the end of each of the gates in the series, except for the last statement, which must be terminated with a semicolon.

```verbatim
// Gate-level description of two-to-four-line decoder

module decoder_2x4_gates (D, A, B, enable);
  output [0: 3] D;
  input A, B;
  input enable;
  wire A_not, B_not, enable_not;

  not
    G1 (A_not, A),
    G2 (B_not, B),
    G3 (enable_not, enable);

  nand
    G4 (D[0], A_not, B_not, enable_not),
    G5 (D[1], A_not, B, enable_not),
    G6 (D[2], A, B_not, enable_not),
    G7 (D[3], A, B, enable_not);

endmodule
```
// Dataflow description of two-to-four-line decoder

module decoder_2x4_df (  
    output [0: 3] D,  
    input A, B, enable  
);  
assign  
D[0] = !((!A) && (!B) && (!enable)),  
D[1] = !(!A) && B && (!enable)),  
D[2] = !(A && (!B) && (!enable)),  
D[3] = !(A && B && (!enable))  
endmodule
It should be noted that a bitwise operator (e.g., \& \) and its corresponding logical operator (e.g., ! ) may produce different results, depending on their operand. If the operands are scalar the results will be identical; if the operands are vectors the result will not necessarily match. For example, !(1010) is (0101), and !(1010) is 0. A binary value is considered to be logically true if it is not 0. In general, use the bitwise operators to describe arithmetic operations and the logical operators to describe logical operations.

### Some Verilog HDL Operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
<th>Symbol</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>binary addition</td>
<td>—</td>
<td>binary subtraction</td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise AND</td>
<td>&amp;&amp;</td>
<td>logical AND</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bitwise OR</td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>bitwise XOR</td>
<td>~</td>
<td>bitwise NOT</td>
</tr>
<tr>
<td>= =</td>
<td>equality</td>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;</td>
<td>less than</td>
<td>{}</td>
<td>concatenation</td>
</tr>
<tr>
<td>&lt;</td>
<td></td>
<td>?:</td>
<td>conditional</td>
</tr>
</tbody>
</table>
The target output is the concatenation of the output carry C\_out and the four bits of Sum. Concatenation of operands is expressed within braces and a comma separating the operands. Thus, \{C\_out, Sum\} represents the five-bit result of the addition operation.

```verilog
// Dataflow description of four-bit adder

// Verilog 2001, 2005 module port syntax
module binary_adder (  
    output [3: 0] Sum,  
    output C_out,  
    input [3: 0] A, B,  
    input C_in  
);

    assign \{C\_out, Sum\} = A + B + C\_in;
endmodule
```

The continuous assignment

assign OUT = select ? A : B;

specifies the condition that OUT = A if select 1, else OUT = B if select 0.
Behavioural modelling represents digital circuits at a functional and algorithmic level. It is used mostly to describe sequential circuits, but can also be used to describe combinational circuits.

```verilog
// Behavioral description of two-to-one-line multiplexer
module mux_2x1_beh (m_out, A, B, select);
  output reg m_out;
  input A, B, select;

  always @ (A or B or select)
    if (select == 1) m_out = A;
    else m_out = B;
endmodule
```

Behavioural descriptions use the keyword always, followed by an optional event control expression and a list of procedural assignment statements.

The target output of a procedural assignment statement must be of the reg data type. Contrary to the wire data type, whereby the target output of an assignment may be continuously updated, a reg data type retains its value until a new value is assigned.
// Behavioral description of four-to-one line multiplexer

// Verilog 2001, 2005 port syntax

module mux_4x1_beh
(output reg m_out,
in_0, in_1, in_2, in_3,
in_0, in_1, in_2, in_3
);

always @ (in_0, in_1, in_2, in_3, select) // Verilog 2001, 2005 syntax
  case (select)
    2'b00: m_out = in_0;
    2'b01: m_out = in_1;
    2'b10: m_out = in_2;
    2'b11: m_out = in_3;
  endcase
endmodule
Encoder Example

- Nested IF-ELSE might lead to “priority logic”
  - Example: 4-to-2 encoder

```verbatim
always @(x)
begin : encode
if (x == 4'b0001) y = 2'b00;
else if (x == 4'b0010) y = 2'b01;
else if (x == 4'b0100) y = 2'b10;
else if (x == 4'b1000) y = 2'b11;
else y = 2'bxx;
end
```

- This style of cascaded logic may adversely affect the performance of the circuit
Encoder Example (cont.)

• To avoid “priority logic” use the case construct:

```verilog
classic @(x)
begim : encode
case (x)
  `4'b0001: y = `2'b00;
  `4'b0010: y = `2'b01;
  `4'b0100: y = `2'b10;
  `4'b1000: y = `2'b11;
  default: y = `2'bxx;
endcase
end
```

• All cases are matched in parallel