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`.

- Behavioural 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.

// 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>&amp;</td>
<td>bitwise AND</td>
</tr>
<tr>
<td>-</td>
<td>binary subtraction</td>
<td>&amp;&amp;</td>
<td>logical AND</td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise AND</td>
<td></td>
<td></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 XOR</td>
</tr>
<tr>
<td>~</td>
<td>bitwise NOT</td>
<td>^</td>
<td>bitwise NOT</td>
</tr>
<tr>
<td>=</td>
<td>equality</td>
<td>=</td>
<td>equality</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>{}</td>
<td>concatenation</td>
<td>{}</td>
<td>concatenation</td>
</tr>
<tr>
<td>?:</td>
<td>conditional</td>
<td>?:</td>
<td>conditional</td>
</tr>
</tbody>
</table>
Signed comparison in Verilog

When you write this in Verilog:

```verilog
wire [7:0] a;
wire [7:0] b;
wire less;
assign less = (a < b);
```

the comparison between a and b is unsigned, that is a and b are numbers in the range 0-255. Writing this instead:

```verilog
wire [7:0] a;
wire [7:0] b;
wire less;
assign less = ($signed(a) < $signed(b));
```

means that the comparison treats a and b as signed 8-bit numbers, which have a range of -128 to +127. Another way of writing the same thing is:

```verilog
wire signed [7:0] a;
wire signed [7:0] b;
wire less;
assign less = (a < b);
```
The question mark is known in Verilog as a conditional operator though in other programming languages it also is referred to as a ternary operator, an inline if, or a ternary if.

It is used as a short-hand way to write a conditional expression in Verilog (rather than using if/else statements). Let's look at how it is used:

```
condition ? value_if_true : value_if_false
```

Here, condition is the check that the code is performing. This condition might be things like, "Is the value in A greater than the value in B?" or "Is A=1?". Depending on if this condition evaluates to true, the first expression is chosen.

If the condition evaluates to false, the part after the colon is chosen.
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.
Verilog uses the bit length of the operands to determine how many bits to use while evaluating an expression. In the case of the addition operator, the bit length of the largest operand, including the left-hand side of an assignment, shall be used.

The tricky part of this problem is to get the carry bit right for subtract!

```
// 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
```