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:

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.

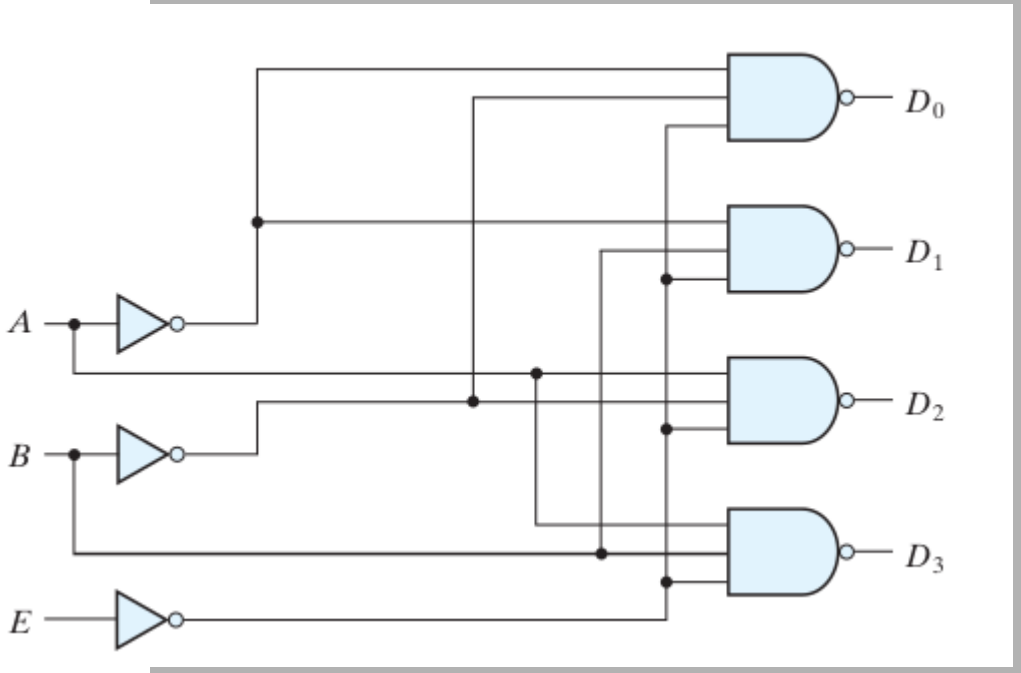
(Note : The first (left-most) number (array index) listed is always the most significant bit of the vector.)

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

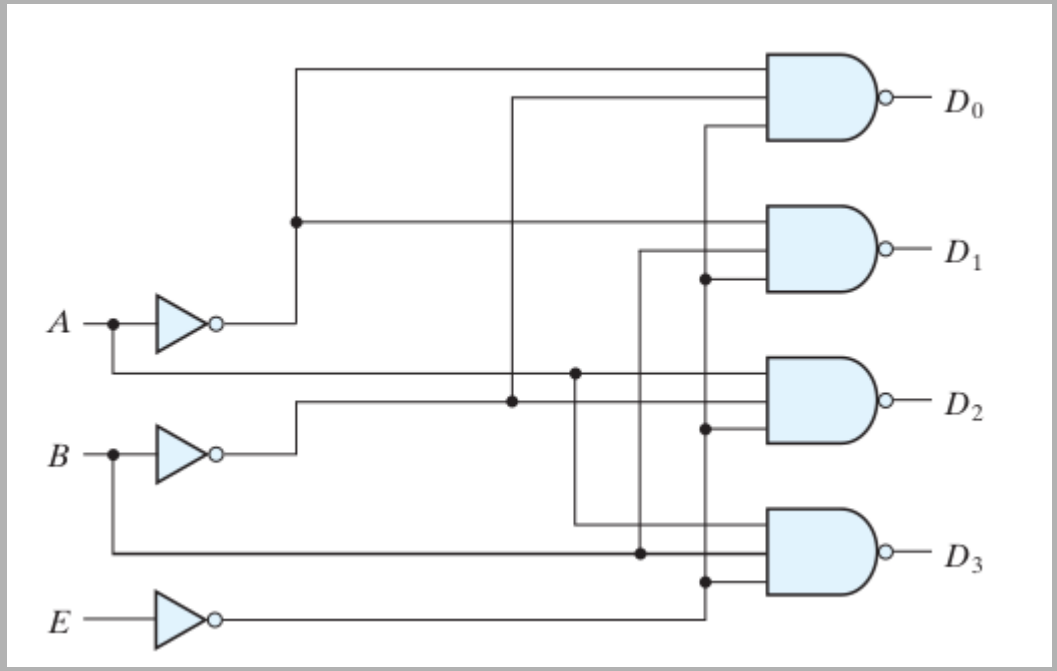
Note that the keywords not and nand
are written only once and do not have to be repeated for each gate, but commas must be inserted at the end of each of the gates in the series, except for the last statement, which must be terminated with a semicolon.

```
// 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

Symbol	Operation	Symbol	Operation
+	binary addition		
–	binary subtraction		
&	bitwise AND	&&	logical AND
|	bitwise OR	||	logical OR
^	bitwise XOR		
~	bitwise NOT	!	logical NOT
= =	equality		
>	greater than		
<	less than		
{}	concatenation		
?:	conditional		

Signed comparison in Verilog

When you write this in 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:

```
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:

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

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.

```
// 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
```

```
// Dataflow description of two-to-one-line multiplexer

module mux_2x1_df(m_out, A, B, select);
  output      m_out;
  input       A, B;
  input       select;

  assign m_out =  select ? A : B;
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

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.

```
// 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.

Single-pass behaviour

Behaviour declared by the keyword initial is called single-pass behaviour and specifies a single statement or a block statement (i.e., a list of statements enclosed by either a

begin . . . end

or a

fork . . . join

keyword pair).

A single-pass behaviour expires after the associated statement executes.

In practice, designers use single-pass behaviour primarily to prescribe stimulus signals in a test bench, never to model the behaviour of a circuit, because synthesis tools do not accept descriptions that use the initial statement.

Cyclic behaviour

The always keyword declares a cyclic behaviour.

Both types of behaviours begin executing when the simulator launches at time t = 0.

The initial behaviour expires after its statement executes; the always behaviour executes and re-executes indefinitely, until the simulation is stopped.

A module may contain an arbitrary number of initial or always behavioural statements.

They execute concurrently with respect to each other, starting at time 0, and may interact through common variables.

```
// Behavioral description of four-to-one line multiplexer

// Verilog 2001, 2005 port syntax

module mux_4x1_beh
( output reg m_out,
  input      in_0, in_1, in_2, in_3,
  input [1: 0] select
);
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
```