# Mano 4-12 ~ 4-15

HDL Models of Comb Ckts

Behavioral Modeling

Writing a Simple Testbench

Logic Simulation

*J.J. Shann*

# 4-12 HDL Models of Combinational Ckts

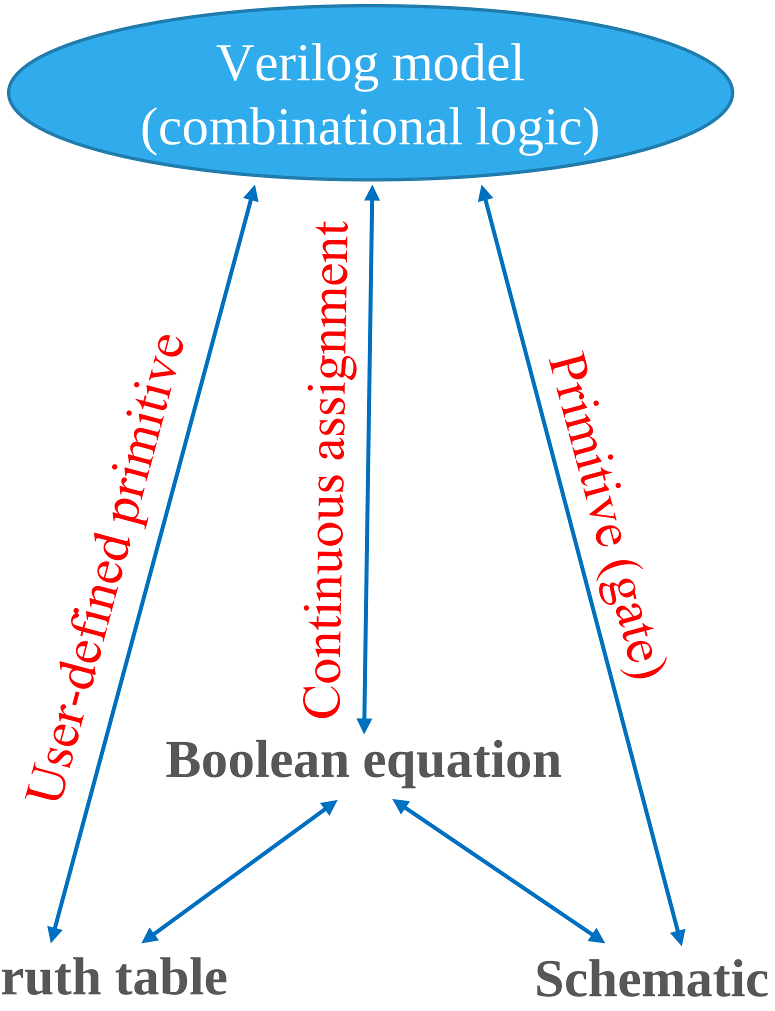
* 1 module per file (.v) w/ the same name

- Module:
  - the basic building block for modeling hardware w/ the Verilog HDL
- Modeling styles of a module:
  - ***gate-level*** modeling: use instantiations of *predefined* and *user-defined primitive gates*
    - describes a ckt by specifying its gates and how they are connected w/ each other
  - ***dataflow*** modeling: use *continuous assignment statements* w/ the keyword **assign**
    - is used mostly for describing the Boolean equations of combinational logic
  - ***behavioral*** modeling: use *procedural assignment statements* w/ the keyword **always**
    - is used to describe combinational and sequential ckts at a higher level of abstraction

# Verilog Model for Combinational Logic

Figure 4.32

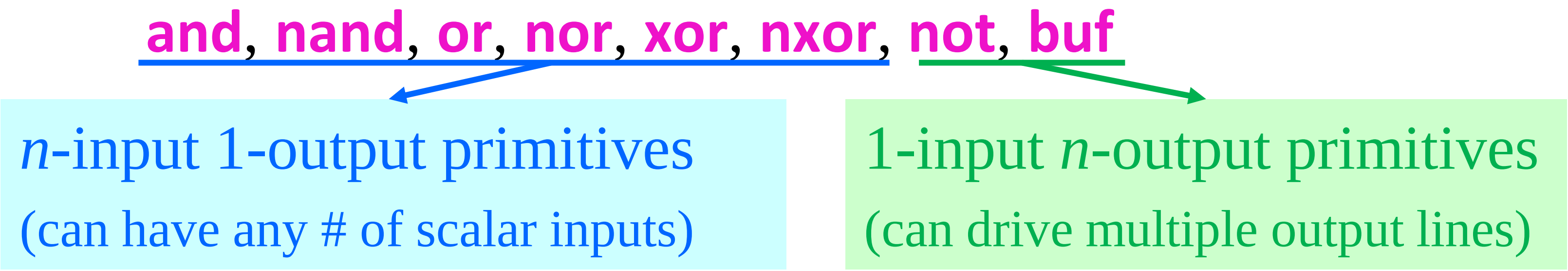
Relationship of Verilog constructs to truth tables, Boolean equations, and schematics



# A. Gate-Level Modeling

* The output of a primitive is always listed first in the port list, followed by the inputs.

- Gate-level modeling:
  - Provides a texture description of a *schematic diagram*.
    - A ckt is specified by its *logic gates* and *their interconnections*.
- Predefined primitives of gates in Verilog: 12
  - 8 digital logic primitive gates:



**and**, **nand**, **or**, **nor**, **xor**, **nxor**, **not**, **buf**

*n*-input 1-output primitives
(can have any # of scalar inputs)

1-input *n*-output primitives
(can drive multiple output lines)

  - 4 three-state type primitive gates: p.HDL-47

    **bufif1**, **bufif0**, **notif1**, **notif0**

*** The logic of each gate is based on a 4-valued system:**

**0, 1, *x* (*unknown*), *z* (*high impedance*)**

# 4-Valued System

- **0**
- **1**
- ***x***: *unknown*
  - An unknown value is assigned during simulation when the *logic value of a signal is ambiguous* (can not be determined whether its value is 0 or 1).
- ***z***: *high impedance*
  - occurs at the *output of three-state gates that are not enabled* or if *a wire is inadvertently left unconnected*.
- E.g.: 4-valued logic truth tables (p.208, Table 4.9)

# 4-Valued Logic Truth Tables

- E.g.: 4-valued logic truth tables (p.208, Table 4.9)

**Truth Table for Predefined Primitive Gates**

| **and** | 0 | 1 | **x** | **z** |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 1 | **x** | **x** |
| **x** | 0 | **x** | **x** | **x** |
| **z** | 0 | **x** | **x** | **x** |

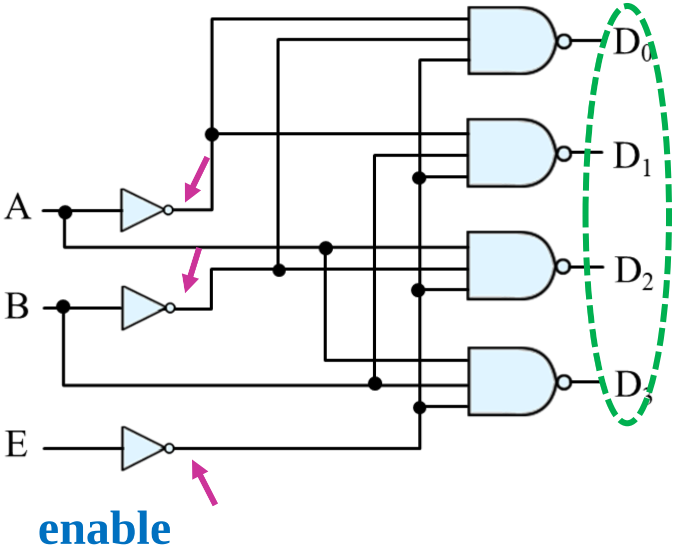
| **or** | 0 | 1 | **x** | **z** |
|---|---|---|---|---|
| 0 | 0 | 1 | **x** | **x** |
| 1 | 1 | 1 | 1 | 1 |
| **x** | **x** | 1 | **x** | **x** |
| **z** | **x** | 1 | **x** | **x** |

| **xor** | 0 | 1 | **x** | **z** |
|---|---|---|---|---|
| 0 | 0 | 1 | **x** | **x** |
| 1 | 1 | 0 | **x** | **x** |
| **x** | **x** | **x** | **x** | **x** |
| **z** | **x** | **x** | **x** | **x** |

| **not** | input | output |
|---|---|---|
| | 0 | 1 |
| | 1 | 0 |
| | **x** | **x** |
| | **z** | **x** |

# HDL Example 4.1: 2-to-4 Line Decoder

- HDL Example 4.1: 2-to-4 Line Decoder
  - *gate-level* description of Fig 4.19



```
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
```

The keyword not may be written only once for the three gates.

```
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
```

- *vector:*
  - a multiple-bit width identifier
  - e.g.s:

    **output** [0: 3] D;

    **wire** [7: 0] SUM;
  - includes within "[ ]" 2 numbers separated w/ a " : "
    - The 1st number listed is always the MSB of the vector.
    - The individual bits are specified within [ ], D[2].
    - may address parts (contiguous bits) of vectors, SUM[2: 0].
- wire: for internal connections

* The output is always listed first in the port list of a primitive, followed by the inputs.

# Design Methodologies of Digital Circuits

- 2 basic design methodologies: ***hierarchical***
  - ***bottom-up*** design: the building blocks are first identified and then combined to build the top-level block
    - E.g.:

      Half adder (HA): add two bits, 1 XOR gate + 1 AND gate

      ⇒ Full adder (FA): add three bits, 2 HAs + 1 OR gate

      ⇒ *n*-bit ripple-carry adder (RCA): add 2 *n*-bit numbers, *n* FAs
  - ***top-down*** design: the top-level block is defined and then the subblocks necessary to build the top-level block are identified
    - E.g.:

      *n*-bit ripple-carry adder (RCA) ⇒ *n* full adders (FAs)

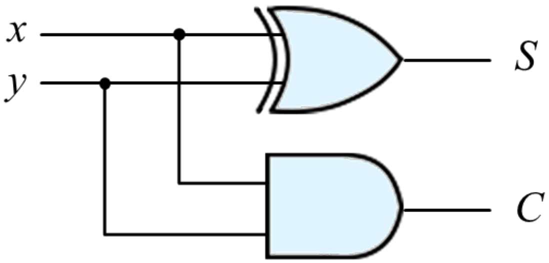
      Full adder ⇒ 2 half adders (HAs) + 1 OR gate

      Half adder ⇒ 1 XOR gate + 1 AND gate

# HDL Example 4.2: Ripple-Carry Adder

- HDL Example 4.2: 4-bit Ripple-Carry Adder
  - *gate-level* description
  - *bottom-up* hierarchical description:

    HA → FA → RCA
  - HA: Fig 4.5(b)

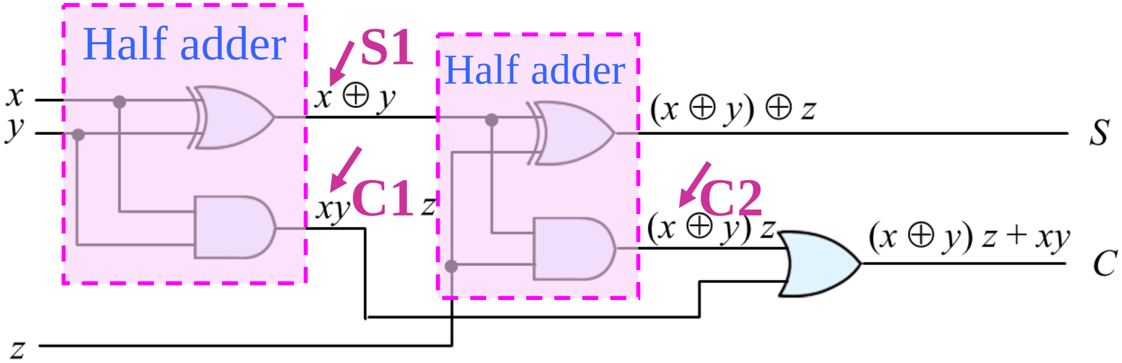


$$S = x \oplus y$$

$$C = xy$$

```
// Description of half adder

//module half_adder (S, C, x, y);
//  output    S, C;
//  input     x, y;

// Alternative Verilog 2005 syntax:

module half_adder (output S, C, input x, y);
 // Instantiate primitive gates
   xor (S, x, y);
   and (C, x, y);
endmodule
```

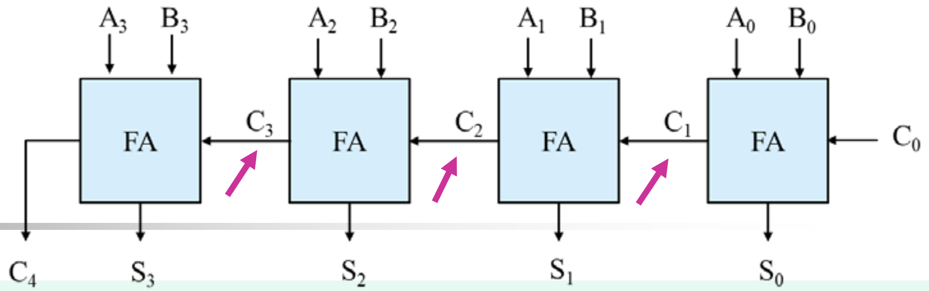
– FA: Fig 4.8



HA module

```
module half_adder (output S, C,
                   input x, y);
   xor (S, x, y);
   and (C, x, y);
endmodule
```

```
// Description of full adder
//module full_adder (S, C, x, y, z);
//  output     S, C;
//  input      x, y, z;

// alternative Verilog 2005 syntax:
module full_adder (output S, C, input x, y, z);
   wire       S1, C1, C2;

// Instantiate half adders
   half_adder HA1 (S1, C1, x, y);
   half_adder HA2 (S, C2, S1, z);
   or G1 (C, C2, C1);
endmodule
```



- 4-bit RCA: Fig 4.9

```
// Description of 4-bit adder (see Fig 4-9)
// module ripple_carry_4_bit_adder ( Sum, C4, A, B, C0);
//    output [3: 0]   Sum;
//    output          C4;
//    input    [3: 0] A, B;
//    input           C0;

// Alternative Verilog 2005 syntax:
module ripple_carry_4_bit_adder ( output [3: 0] Sum,
   output C4,   input [3:0]  A, B, input C0);
   wire         C1, C2, C3;         // Intermediate carries
// Instantiate chain of full adders
   full_adder FA0 (Sum[0], C1, A[0], B[0], C0),
              FA1 (Sum[1], C2, A[1], B[1], C1),
              FA2 (Sum[2], C3, A[2], B[2], C2),
              FA3 (Sum[3], C4, A[3], B[3], C3);
endmodule
```

FA module

```
module full_adder (output S, C,
                   input x, y, z);
  wire        S1, C1, C2;

  half_adder HA1 (S1, C1, x, y);
  half_adder HA2 (S, C2, S1, z);
  or G1 (C, C2, C1);
endmodule
```

- Module declarations **cannot** be nested.
  - A module definition (declaration) cannot be placed within another module declaration.

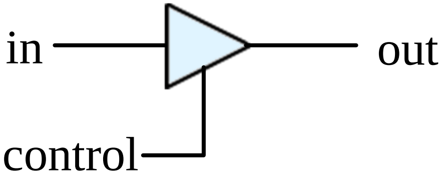
    ⇒ A module definition cannot be inserted into the text b/t the module and endmodule keywords of another module.
- Modules can be instantiated within other modules.
  - Creates a *hierarchical* decomposition of a design.

    ⇒ *structural* description

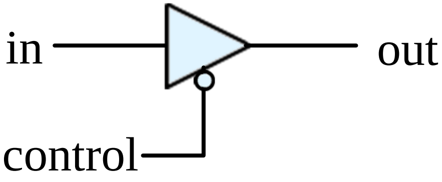
* One module per file (.v) & w/ the same name.

# B. Three-State Gates

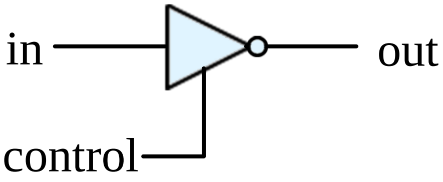
- Three-state gate:
  - has a control input that can place the gate into a *high-impedance* state (***z***).
- Types of three-state gates: 4
  - **bufif1**
  - **bufif0**
  - **notif1**
  - **notif0**



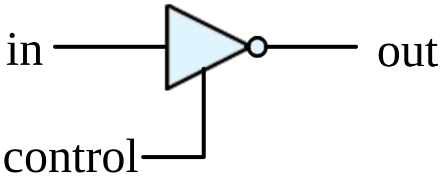
**bufif1**



**bufif0**

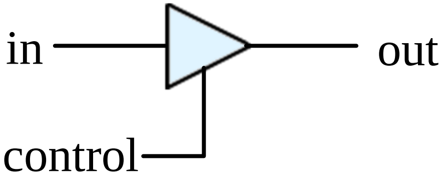


**notif1**

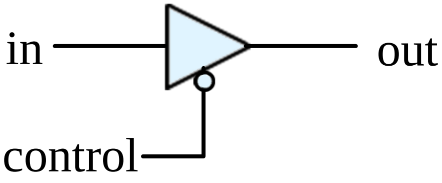


**notif0**

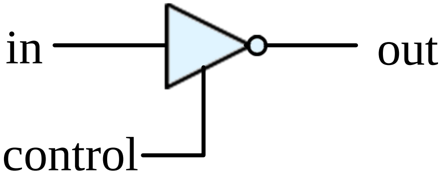
- Three-state gates:
  - **bufif1**, **bufif0**, **notif1**, **notif0**



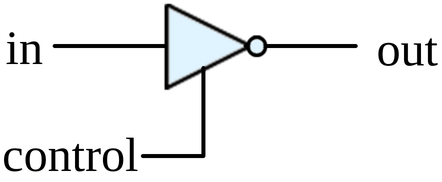
**bufif1**



**bufif0**



**notif1**



**notif0**

  - are instantiated w/ the statement

    ***gate name (output, input, control);***

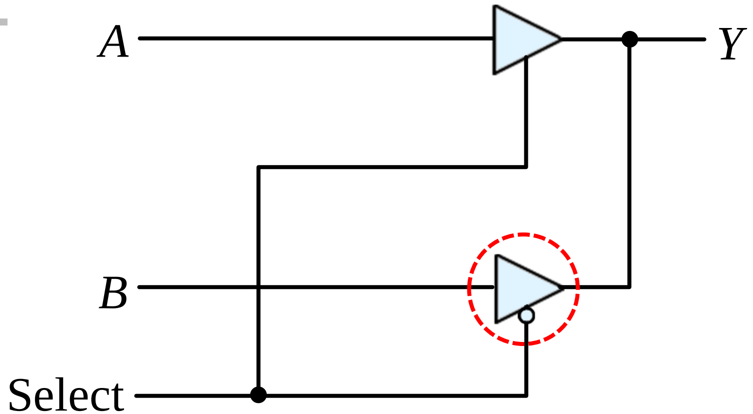
  - In simulation, the output can result in 0, 1, *x*, or *z*.
  - The outputs of 3-state gates can be connected together to form a common output line.

# HDL Example: 2-to-1 Line Multiplexer

- HDL Example: 2-to-1 Line MUX w/ three-state buffers
  - p.205, Fig 4.30
  - **tri**: for tristate
    - indicate that the output has multiple drivers
  - Types of 3-state gates:

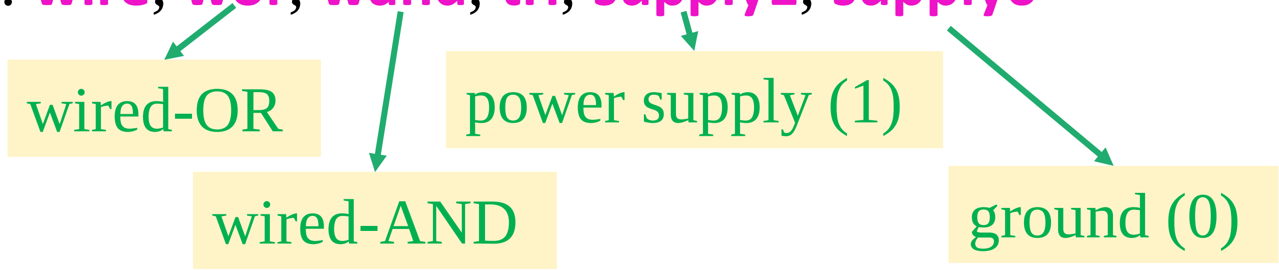
    **bufif1**, **bufif0**, **notif1**, **notif0**
  - Instantiate of a 3-state gate:

    ***gate name (output, input, control);***



```
module mux_tri (m_out, A, B, select);
   output   m_out;
   input    A, B, select;
   tri      m_out;

   bufif1 (m_out, A, select);
   bufif0 (m_out, B, select);
endmodule
```

- *nets*: a class of data types
  - represent connections b/t hardware elements
  - e.g.s: **wire**, **wor**, **wand**, **tri**, **supply1**, **supply0**



wired-OR

wired-AND

power supply (1)

ground (0)

  - If an identifier is used, but not declared, the language specifies that it will be interpreted (by default) as a **wire**.

# C. Dataflow Modeling

| | **Bitwise** | **Logical** |
|---|---|---|
| AND | & | && |
| OR | \| | \|\| |
| NOT | ~ | ! |

- Dataflow modeling:
  - uses a number of operators that act on binary operands to produce a binary result.
  - Commonly used Verilog HDL operators: p.222, Table 4.10

- Commonly used Verilog HDL operators: Table 4.10

| 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 | | |

| 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 | | |

| | Bitwise | Logical |
|---|---|---|
| AND | & | && |
| OR | \| | \|\| |
| NOT | ~ | ! |

- *bitwise* operators: operate bit by bit on a pair of vector operands to produce a vector result
  - E.g.: ~(1010) is (0101)
- *logical* operators:
  - E.g.: !(1010) is 0
- *concatenation* operator: **{}**, append multiple operands
- *conditional* operator: **?:**, acts like a multiplexer

* If the operands are ***scalar***, the results of *bitwise* and *logical* operators will be identical; if the operands are **vectors**, the result will not necessarily match.

*condition* **?** *true-expression* **:** *false-expression*

* If the operands are scalar, the results will be identical; if the operands are vectors, the result will not necessarily match.

| | **Bitwise** | **Logical** |
|---|---|---|
| AND | & | && |
| OR | \| | \|\| |
| NOT | ~ | ! |

- Example: ***A*** =1010, ***B*** = 0000
  - ***A*** has the Boolean value 1, ***B*** has Boolean value 0.
  - Results of other operations with these values:

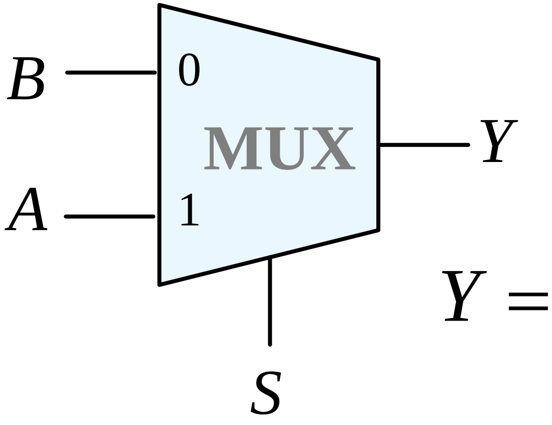
| | | |
|---|---|---|
| A & B = 0000 | // Bitwise AND | (1010) & (0000) = (0000) |
| A && B = 0 | // Logical AND | (1010) && (0000) = 0 |
| A \| B = 1010 | // Bitwise OR | (1010) \| (0000) = (1010) |
| A \|\| B = 1 | // Logical OR | (1010) \|\| (0000) = 1 |
| ~A = 0101 | // Bitwise negation | ~(1010) = (0101) |
| !A = 0 | // Logical negation | !(1010) = !(1) = 0 |
| ~B = 1111 | // Bitwise negation | ~(0000) = (1111) |
| !B = 1 | // Logical negation | !(0000) = !(0) = 1 |
| (A > B) = 1 | // is greater than | |
| (A == B) = 0 | // identity (equality) | |

# Dataflow Modeling of Comb. Logic

- Dataflow modeling of combinational logic:
  - Describes combinational ckts by their *function* rather than by their gate structure
  - uses *continuous assignments* and the keyword **assign**.
  - E.g.: a 2-to-1-line MUX w/ scalar data inputs *A* and *B*, select input *S*, and output *Y*

```
assign Y = (A && S) || (B && !S);
```

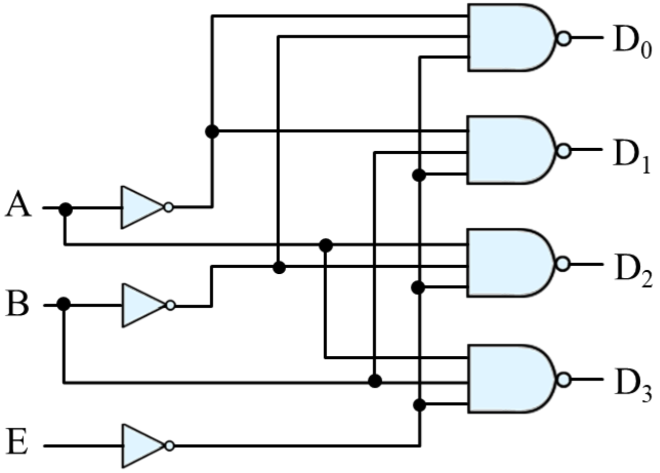
| | Bitwise | Logical |
|---|---|---|
| AND | & | && |
| OR | \| | \|\| |
| NOT | ~ | ! |



$$Y = SA + \overline{S}B$$

# HDL Example 4.3: 2-to-4 Line Decoder (Dataflow)

- HDL Example 4.3: (Dataflow)
  - *dataflow* description of Fig 4.19



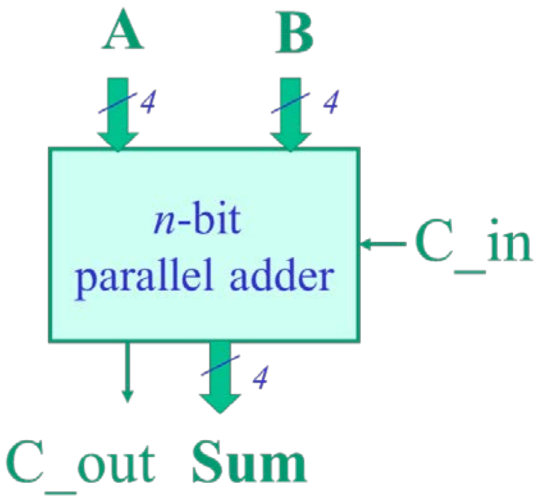
**enable**

| | Bitwise | Logical |
|---|---|---|
| AND | & | && |
| OR | \| | \|\| |
| NOT | ~ | ! |

```
//Verilog 2001, 2005 Syntax
module decoder_2x4_df (
   output [0: 3]   D,
   input           A, B,
   input           enable
);

   assign  D[0] = !((!A) && (!B) && (!enable)),
            D[1] = !((!A) && B & (!enable)),
            D[2] = !(A && (!B) && (!enable)),
            D[3] = !(A & B && (!enable));
endmodule
```

# HDL Example 4.4: 4-bit Adder (Dataflow)

- HDL Example 4.4: (Dataflow) 4-bit adder
  - *dataflow* description of 4-bit adder



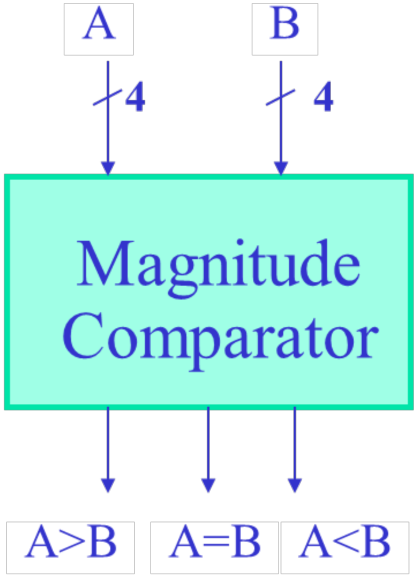
```
// Verilog 2005 module port syntax
module binary_adder (
    output  [3:0]   Sum,
    output          C_out,
    input   [3: 0]  A, B,
    input           C_in
);

  assign {C_out, Sum} = A + B + C_in;
endmodule
```

5-bit result of the addition operation

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

# HDL Example 4.5: 4-bit Comparator (Dataflow)

- HDL Example 4.5: (Dataflow) 4-bit comparator

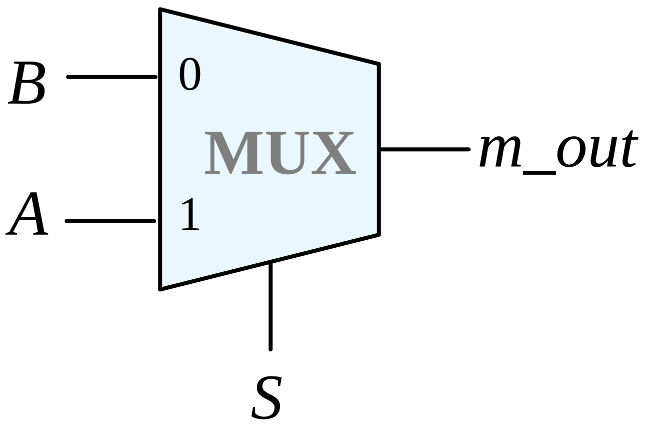


```
// Verilog 2001, 2005 syntax
module mag_compare (
   output  A_lt_B, A_eq_B, A_gt_B,
   input     [3: 0] A, B
);

   assign A_lt_B = (A < B);
   assign A_gt_B = (A > B);
   assign A_eq_B = (A == B);
endmodule
```

| 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 | | |

# HDL Example 4.6: 2-to-1 MUX (Dataflow)

- HDL Example 4.6: (Dataflow) 2-to-1 MUX



```
// Verilog 2001, 2005 syntax
module mux_2x1_df(m_out, A, B, select);
  output        m_out;
  input         A, B;
  input         select;

  assign m_out = (select)? A : B;
 endmodule
```

  - *conditional* operator: **?:**

    *condition ? true-expression : false-expression*

# D. Behavioral Modeling

* **reg** data type:
retains its value until a new value is assigned

- ***Behavioral*** modeling:
  - represents digital ckts at a *functional* and *algorithmic* level
  - is used mostly to describe *sequential* ckts, but can also be used to describe *combinational* ckts.
  - use keyword **always**, followed by an optional *event control expression* and a list of *procedural assignment statements*.

```
always @ (event control expression) begin
    //procedural assignment statements
end
```

specifies when the statements will execute

- The procedural assignment statements inside the **always** block are executed every time there is a change in *any* of the variables listed after the **@** symbol.

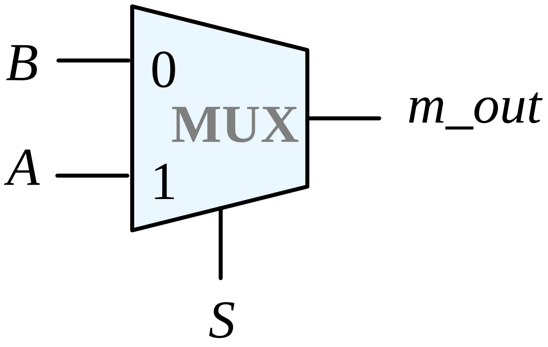
keyword **or**

* The target output of the procedural assignment statement must be **reg** data type.

- **if-else** conditional  statement
- **case** statement:  **case** … **endcase**
  - a multiway conditional branch construct
  - The case items have an implied priority because the list is evaluated from top to bottom.

# HDL Example 4.7: 2-to-1 MUX (Behavioral)

- HDL Example 4.7: (Behavioral) 2-to-1 Line MUX
  - *Behavioral* description of 2-to-1 line MUX



```
// Verilog 2001, 2005 syntax
module mux_2x1_beh(
   output  reg  m_out,
   input        A, B, select
);

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

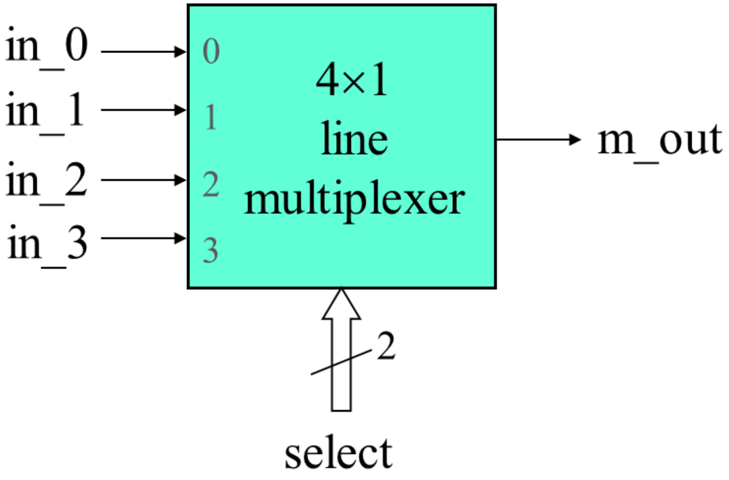
*** The target output of the procedural assignment statement inside always must be reg data type.**

*** Use *blocking assignments* (=) in always block for combinational logic.**

  - **if-else** conditional statement

# HDL Example 4.8: 4-to-1 Line MUX (Behavioral)

- HDL Example 4.7: (Behavioral) 4-to-1 Line MUX
  - Behavioral description of 4-to-1 line MUX



```
// Verilog 2001, 2005 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)
      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
```

**case items, have implied priority (top to bottom)**

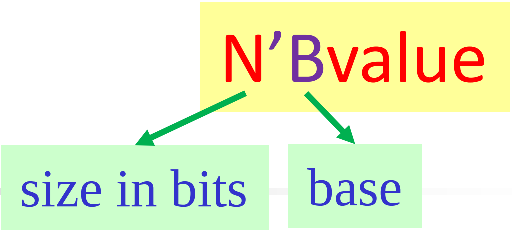
# **case** Construct

- **case** statement: **case** … **endcase**
  - a multiway conditional branch construct
  - The case items have an implied priority because the list is evaluated from top to bottom.
- **default**: the last item in the list of case items, for unlisted case items
- 2 important variations of **case** construct:
  - **casex**: don't-cares any bits of the **case** expression or the **case** item that have logic value **x** or **z**
  - **casez**: don't-cares only the logic value **z**

x: unknown

z: high-impendence

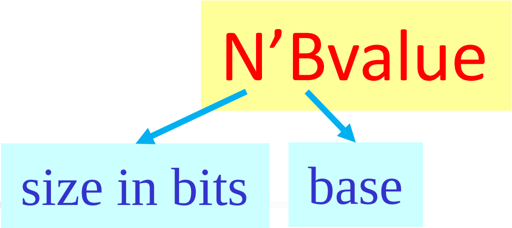
# Numbers


N'Bvalue

size in bits

base

- Numbers can be specified in *binary*, *octal*, *decimal*, or *hexadecimal* (bases 2, 8, 10, 16):
  - Underscores ( _ ) in numbers are ignored and can be helpful in breaking long numbers into more readable chunks.
- Format for declaring constants: N'Bvalue
  - N: the size in bits, leading zeros are inserted to reach this size
    - optional, but better give the size explicitly.
    - If the size is not specified, the system assumes that it is the word length of the host simulator or at least 32 bits.
  - B: the letter indicating the base
    - **'b** for binary, **'o** for octal, **'d** for decimal, and **'h** for hexadecimal
    - If the base is omitted, it defaults to decimal.
  - value: gives the value


N’Bvalue

size in bits

base

- Examples: Harris, Table 4.3, p.180

| Numbers | Bits | Base | Val | Stored |
|---|---|---|---|---|
| 3’b101 | 3 | 2 | 5 | 101 |
| ’b11 | ? | 2 | 3 | 00…………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……00101010 |

* ’0, ’1: fill a bus w/ all 0s and all 1s

# Summary

- Guidelines of modeling *combinational* logic:
  - Gate-level modeling
  - Dataflow modeling:
    - Use *continuous assignments* **assign** to model *simple combinational* logic.

```
assign y = s ? d1 : d2;
```

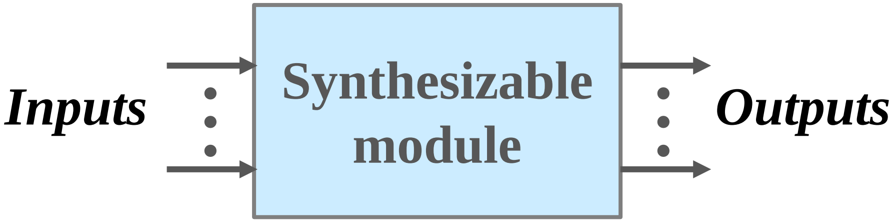
  - Behavioral modeling:
    - Use **always @ (*)** and *blocking assignments* to model more *complicated combinational* logic where the **always** statement is helpful.

```
always @ (*)
   begin
      p = a ^ b;  //blocking
      q = a & b;  //blocking
      s = p ^ cin;
      cout = q | (p & cin);
   end
```

* Do not make assignments to the same signal in more than one **always** statement or continuous assignment statement.

# E. Writing a Simple Test Bench

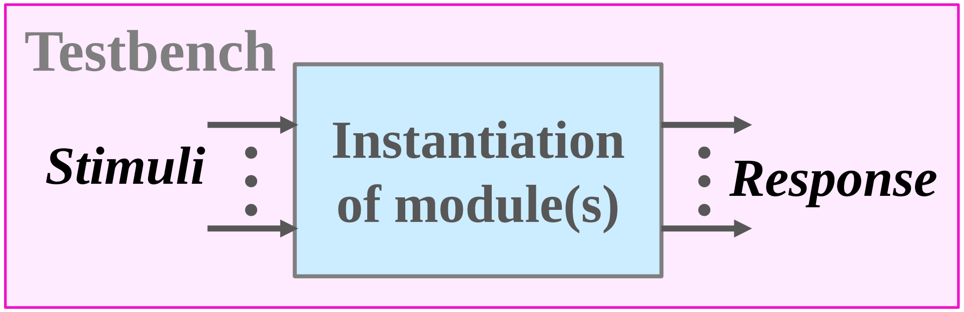
- Test bench:
  - is an HDL program used for describing and applying a *stimulus* to an HDL model of a ckt in order to test it and observe its *response* during *simulation*.
  - Write stimuli that will test a ckt thoroughly, exercising all of the operating features that are specified.
  - can be complex and lengthy and may take longer to develop than the design that is tested.
  - The results of a test are only as good as the test bench that is used to test a ckt.



***Inputs***

**Synthesizable module**

***Outputs***



**Testbench**

***Stimuli***

**Instantiation of module(s)**

***Response***

# (a) Providing Input Stimulus

- Statements used by a test bench to provide a stimulus to the ckt being tested:
  - **initial** statement: executes only once
    - starting from simulation time 0, and may continue w/ any operations that are delayed by a given number of time units, as specified by the symbol **#**.
  - **always** statement: executes repeatedly in a loop
    - specify how the associated statement is to execute (the *event control expression*)

```
always @ (event control expression) begin
      //procedural assignment statements
end
```

- Example:

| stimulus | A | B |
|---|---|---|
| (t = 0) | 0 | 0 |
| (t = 10) | 1 | 0 |
| (t = 30) | 0 | 1 |

```
Initial
   begin
       A = 0; B = 0;
       #10   A = 1;
       #20   A = 0; B = 1;
   end
```

- Example:

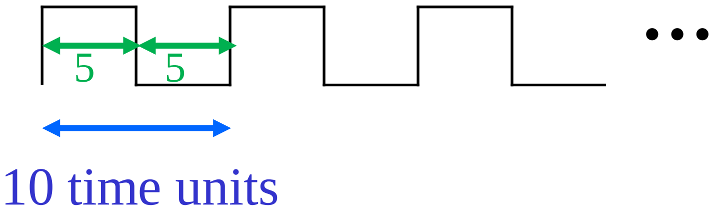
| stimulus | D |
|---|---|
| (t = 0) | 000 |
| (t = 10) | 001 |
| … | |
| (t = 70) | 111 |

```
Initial
   begin
      D = 3’b000;
      repeat (7)
      #10  D = D + 3’b001;
   end
```

  - **repeat**: specifies a looping statement

- Example:

generate periodic clock pulse (period = 10 time units)



```
always
   begin
      clock = 1;  #5;  clock = 0;  #5;
   end
```

# (b) Displaying Output Response to the Stimulus

- The response of the stimulus generated by the initial and always blocks will appear:
  - in text format as standard output &
  - as waveforms (timing diagrams) in simulators having graphical output capability
- Numerical outputs are displayed by using Verilog ***system tasks***.
- Verilog system tasks:
  - are built-in system functions that are recognized by keywords that begin with the symbol **$**.

# System Tasks for Display

- **$display**: display a one-time value of variables or strings with an end-of-line return
- **$write**: same as **$display**, but w/o going to next line
- **$monitor**: display variables whenever a value changes during a simulation run
- **$time**: display the *simulation time*
- **$finish**: terminate the simulation

- Syntax for **$display**, **$write**, and **$monitor**:

  (next page)

- Syntax for **$display**, **$write**, and **$monitor**:

  ***Task-name (format specification, argumentlist);***

  – **format specification**:
    - uses the symbol **%** to specify the radix of the numbers that are displayed : **%b**, **%d**, **%h**, **%o** (%B, %D, %H, %O)
    - may have a string enclosed in quotes (**“……”**)
    - No commas in the format specification.

  – **argument list**:
    - has commas b/t the variables.

  – E.g.s:

```
$display (“%d %b %b”, C, A, B);
$display (“time = %0d A = %b B = %b”, $time, A, B);
```

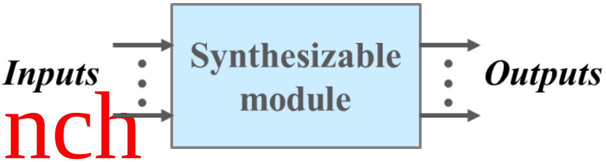
format specification argument list

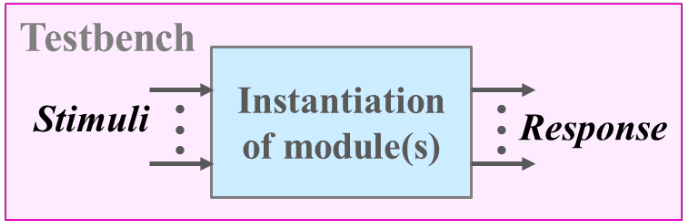
**%d**: w/ the leading spaces

**%0d**: w/o the leading spaces

* In displaying time values, use the format %0d instead of %d.
(Time is calculated as a 32-bit number.)

# (c) Testbench

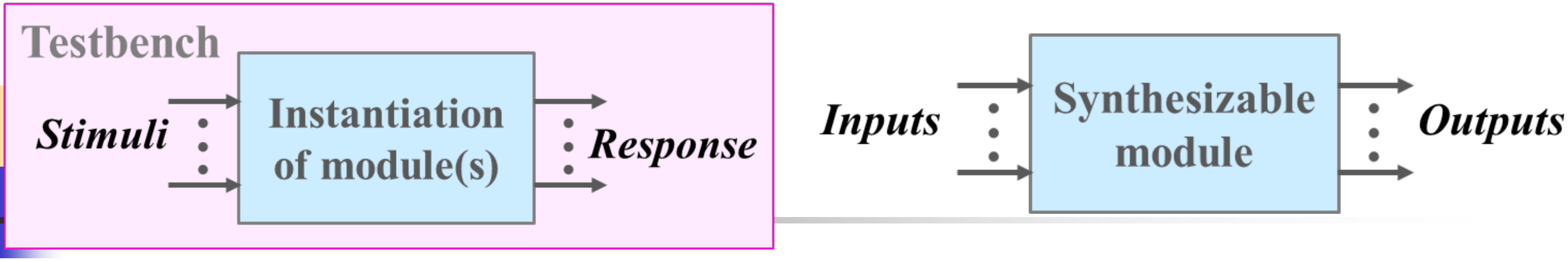




- Form of a test bench:

```
module test_module_name;      //no port list
    //Declare local reg and wire identifiers.
    //Instantiate the design module(s) under test.
    //Specify a stopwatch, using $finish to terminate the simulation.
    //Generate stimulus, using initial and always statements.
    //Display the output response (text or graphics, or both).
endmodule
```

  - has no inputs or outputs ⇒ No port list!
  - The signals that are applied as inputs to the design module for simulation are declared in the stimulus module as local **reg** data type.
  - The outputs of the design module that are displayed for testing are declared in the stimulus module as local **wire** data type.



- Interaction b/t stimulus and design modules:

**Stimulus module (testbench)**

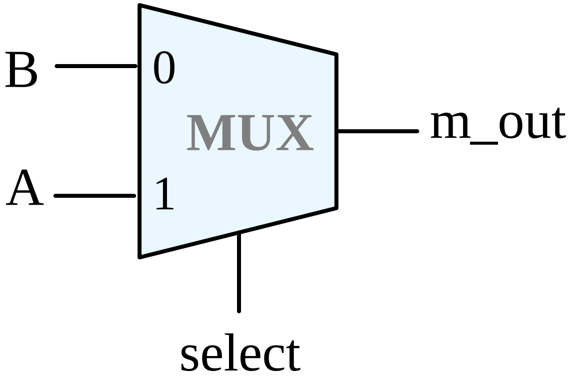
```
module t_circuit;
   reg t_A, t_B;
   wire t_C;
   parameter stop_time = 1000;

   circuit M (t_C, t_A, t_B);

// Stimulus generators for t_A and t_B
   initial # stop_time $finish;

endmodule
```

**Design module**

```
module circuit (C, A, B);
   input A, B;
   output C;

// Description of the module

endmodule
```

# HDL Example 4.9: Test Bench (2-to-1 MUX)

- HDL Example 4.9: Test bench w/ stimulus for mux_2x1_df



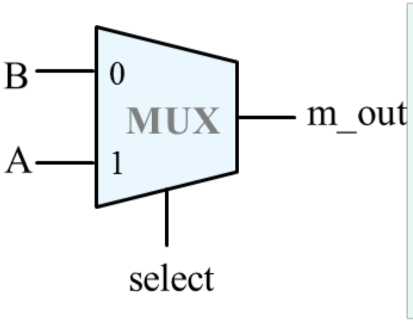
```
module mux_2x1_df(m_out, A, B, select);
   output   m_out;
   input    A, B;
   input    select;

   assign m_out = (select)? A : B;
endmodule
```

```
module  t_mux_2x1_df;
   wire        t_m_out;
   reg         t_A, t_B;
   reg         t_select;
   parameter  stop_time = 50;

   // Instantiation of circuit to be tested
   mux_2x1_df  M1 (t_m_out, t_A, t_B, t_select);

   initial # stop_time $finish;

   // Stimulus generator
   initial begin
      t_select = 1; t_A = 0; t_B = 1;
      #10    t_A = 1; t_B = 0;
      #10    t_select = 0;
      #10    t_A = 0; t_B = 1;
   end
   …
```

parameter: a keyword, defines constant

$finish: a system task, terminates the simulation

| Stimulus: | Select | A | B |
|---|---|---|---|
| (t = 0) | 1 | 0 | 1 |
| (t = 10) | 1 | 1 | 0 |
| (t = 20) | 0 | 1 | 0 |
| (t = 30) | 0 | 0 | 1 |



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

   assign m_out = (select)? A : B;
 endmodule
```

```
Simulation log:
Select = 1  A = 0  B = 1  OUT = 0  time = 0
Select = 1  A = 1  B = 0  OUT = 1  time = 10
Select = 0  A = 1  B = 0  OUT = 0  time = 20
Select = 0  A = 0  B = 1  OUT = 1  time = 30
```

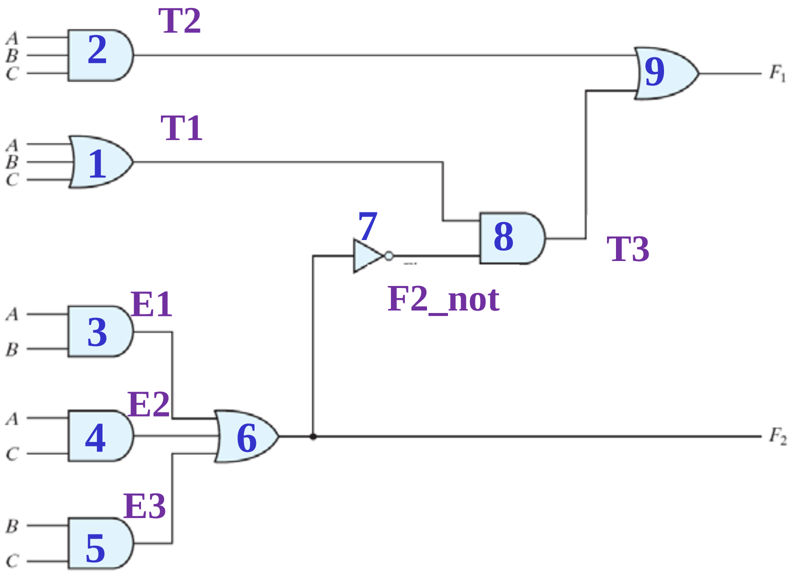
```
module  t_mux_2x1_df;
   //Declaration of local identifiers
   …
   // Instantiation of circuit to be tested
   …
   // Stimulus generator
   …

   //Response monitor
   initial begin
      $display ("     time   Select  A    B    m_out");
      $monitor ($time, "   %b      %b    %b    %b", t_select, t_A, t_B, t_m_out);
      //$monitor ("time=", $time, "select = %b A = %b B = %b m_out = %b",
      t_select, t_A, t_B, t_m_out);
   end
endmodule
```

$display:  a system task, print in the simulator window

$monitor:  a system task, displays the output caused by the given stimulus

# HDL Example 4.10: Design Module & Test Bench (1/2)

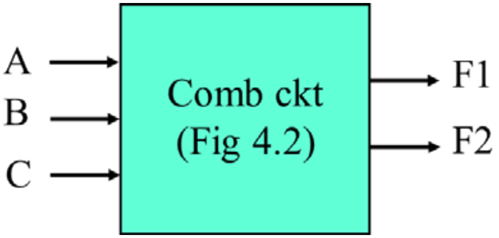
- HDL Example 4.10: Design Module & Test bench of Fig 4.2
  - *gate-level* description of ckt of Fig 4.2

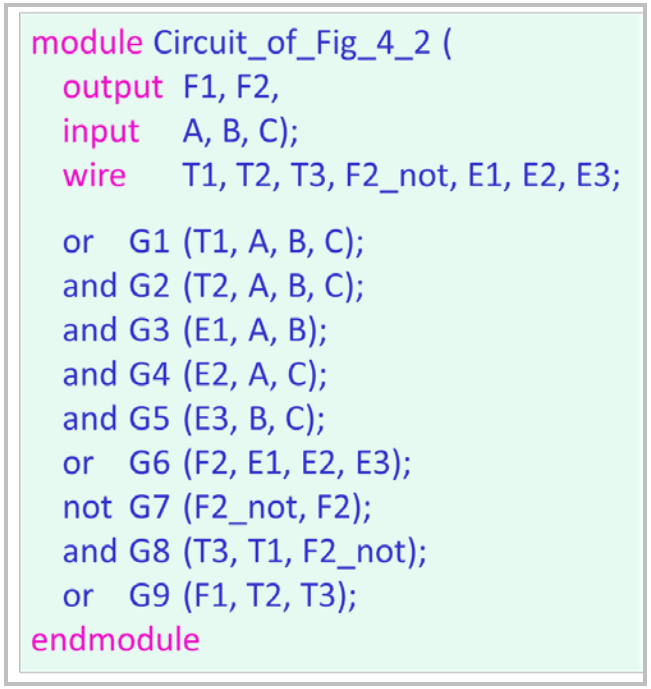


```
module Circuit_of_Fig_4_2 (
   output  F1, F2,
   input   A, B, C);
   wire    T1, T2, T3, F2_not, E1, E2, E3;

   or  G1 (T1, A, B, C);
   and G2 (T2, A, B, C);
   and G3 (E1, A, B);
   and G4 (E2, A, C);
   and G5 (E3, B, C);
   or  G6 (F2, E1, E2, E3);
   not G7 (F2_not, F2);
   and G8 (T3, T1, F2_not);
   or  G9 (F1, T2, T3);
endmodule
```

# HDL Example 4.10: Design Modul Bench (2/2)

- Stimulus to analyze the ckt of Fig 4.2





```
module Circuit_of_Fig_4_2 (
  output  F1, F2,
  input   A, B, C);
  wire    T1, T2, T3, F2_not, E1, E2, E3;

  or   G1 (T1, A, B, C);
  and G2 (T2, A, B, C);
  and G3 (E1, A, B);
  and G4 (E2, A, C);
  and G5 (E3, B, C);
  or   G6 (F2, E1, E2, E3);
  not G7 (F2_not, F2);
  and G8 (T3, T1, F2_not);
  or   G9 (F1, T2, T3);
endmodule
```

```
module t_Circuit_of_Fig_4_2;
  reg   [2: 0] D;
  wire F1, F2;
  parameter stop_time = 100;

 Circuit_of_Fig_4_2 M1 (F1, F2, D[2], D[1], D[0]);

  initial  # stop_time $finish;

  initial  begin              // Stimulus generator
     D = 3'b000;
     repeat (7) #10 D = D + 1'b1;
  end

  initial  begin              //Response monitor
     $display ("A     B     C     F1   F2");
    $monitor ("%b     %b     %b     %b    %b", D[2],
     D[1], D[0], F1, F2);
  end
endmodule
```

```
Simulation log:
ABC = 000  F1 = 0  F2 = 0
ABC = 001  F1 = 1  F2 = 0
ABC = 010  F1 = 1  F2 = 0
ABC = 011  F1 = 0  F2 = 1
ABC = 100  F1 = 1  F2 = 0
ABC = 101  F1 = 0  F2 = 1
ABC = 110  F1 = 0  F2 = 1
ABC = 111  F1 = 1  F2 = 1
```