EGC220

Digital Logic Fundamentals

Design Using Verilog

Baback Izadi

Division of Engineering Programs

bai@engr.newpaltz.edu

Basic Verilog

- Lexical Convention
- Lexical convention are close to C++.
- Comment
  - // to the end of the line.
  - /* to */ across several lines
- Keywords are lower case letter & it is **case sensitive**
- VERILOG uses 4 valued logic: 0, 1, x and z
- **Comments:** // Verilog code for AND-OR-INVERT gate

module <module_name> (<module_terminal_list>);

<module_terminal_definitions>

…

<functionality_of_module>

…

endmodule

Taste of Verilog

Module name *Module ports*

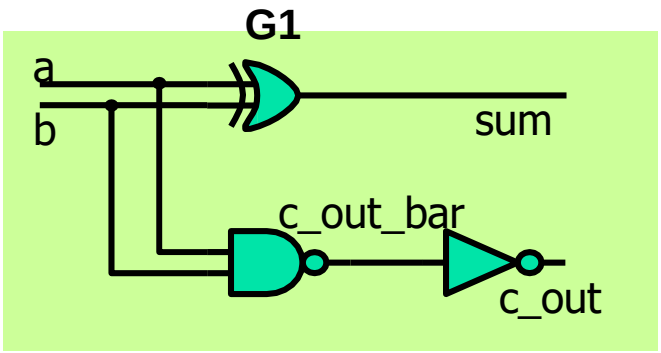
```
module Add_half ( sum, c_out, a, b );
   input      a, b;
   output     sum, c_out;
   wire       c_out_bar;

   xor (sum, a, b);
    // xor G1(sum, a, b);
   nand (c_out_bar, a, b);
   not (c_out, c_out_bar);
endmodule
```

Verilog keywords

Declaration of port modes

Declaration of internal signal

Instantiation of primitive gates



Lexical Convention

- Numbers are specified in the traditional form or below .

  <size><base format><number>

- Size: contains *decimal* digitals that specify the size of the constant in the number of bits.
- Base format: is the single character ‘ followed by one of the following characters *b(binary),d(decimal),o(octal),h(hex).*
- Number: legal digital.

Example :

- 347 -- decimal number
- 4’b101 -- 4- bit 0101_2
- 2’o12 -- 2-bit octal number
- 5’h87f7 -- 5-digit $87F7_{16}$
- 2’d83 -- 2-digit decimal
- String in double quotes

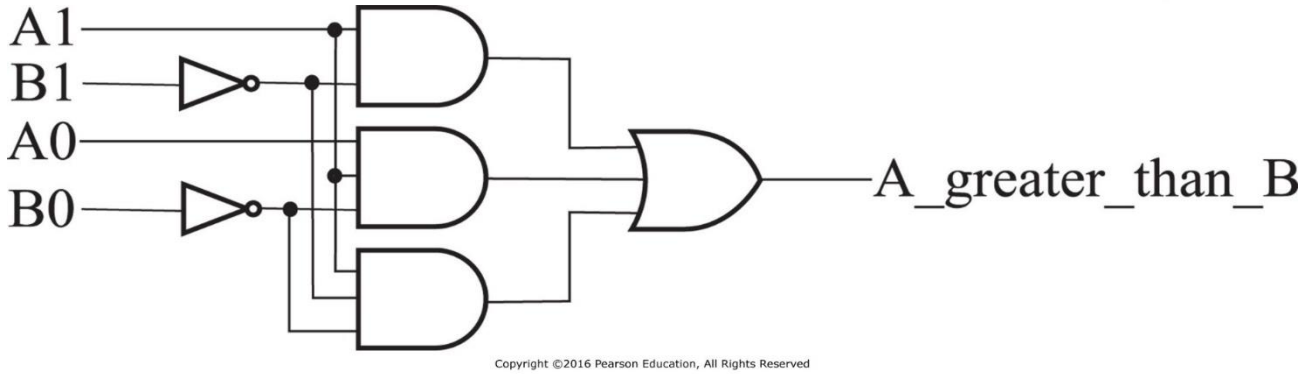
  “ this is a introduction”

Three Modeling Styles in Verilog

- Structural modeling (Gate-level)
  - Use predefined or user-defined primitive gates.
- Dataflow modeling
  - Use assignment statements (assign)
- Behavioral modeling
  - Use procedural assignment statements (always)

Structural Verilog Description of Two-Bit Greater-Than Circuit

```
// Two-bit greater-than circuit: Verilog structural model                  //  1
// See Figure 2-27 for logic diagram                                       //  2
module comparator_greater_than_structural(A, B, A_greater_than_B);         //  3
 input [1:0] A, B;                                                         //  4
 output A_greater_than_B;                                                  //  5
 wire B0_n, B1_n, and0_out, and1_out, and2_out;                            //  6
  not                                                                      //  7
   inv0(B0_n,  B[0]),  inv1(B1_n,  B[1]);                                  //  8
  and                                                                      //  9
   and0(and0_out,  A[1],  B1_n),                                           // 10
   and1(and1_out,  A[1],  A[0],  B0_n),                                    // 11
   and2(and2_out,  A[0],  B1_n,  B0_n);                                    // 12
  or                                                                       // 13
   or0(A_greater_than_B,  and0_out,  and1_out,  and2_out);                 // 14
endmodule                                                                  // 15
```



Dissection

- **Module and Port declarations**
  - Verilog-2001 syntax
    - **module** AOI (input A, B, C, D, output F);
  - Verilog-1995 syntax

    module AOI (A, B, C, D, F);

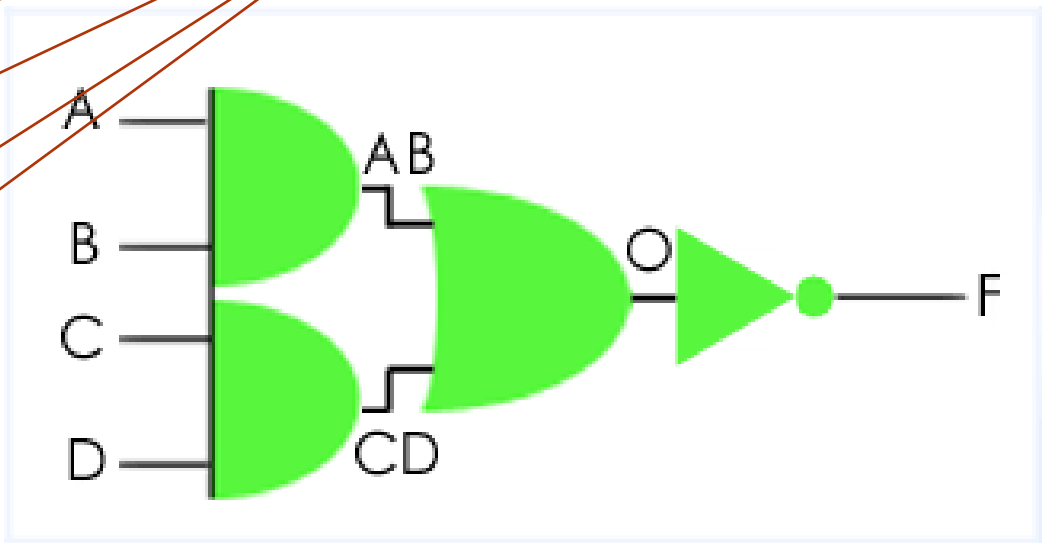
    input A, B, C, D;

    output F;
- **Wires:** Continuous assignment to an internal signal

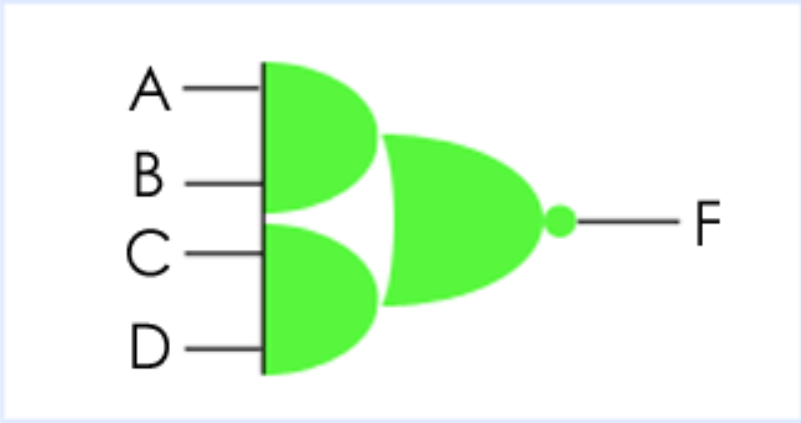
A Simple Dataflow Design

```
// Verilog code for AND-OR-INVERT gate
module AOI (input A, B, C, D, output F);
  wire F;  // the default
  wire AB, CD, O;  // necessary
   assign AB = A & B;
  assign CD = C & D;
  assign O = AB | CD;
  assign F = ~O;
endmodule
// end of Verilog code
```

Continuous Assignments



A Simple Dataflow Design




```
// Verilog code for AND-OR-INVERT gate
module AOI (input A, B, C, D, output F);
  assign F = ~((A & B) | (C & D));
endmodule
// end of Verilog code
```

'&' for AND, '|' for OR, '^' for XOR '^~' for XNOR, '&~' for NAND

Dataflow Verilog Description of Two-Bit Greater-Than Comparator

```
// Two-bit greater-than circuit: Dataflow model                    //  1
// See Figure 2-27 for logic diagram                               //  2
module comparator_greater_than_dataflow(A, B, A_greater_than_B);   //  3
 input [1:0] A, B;                                                 //  4
 output A_greater_than_B;                                          //  5
 wire B1_n, B0_n, and0_out, and1_out, and2_out;                    //  6
 assign B1_n = ~B[1];                                              //  7
 assign B0_n = ~B[0];                                              //  8
 assign and0_out = A[1] & B1_n;                                    //  9
 assign and1_out = A[1] & A[0] & B0_n;                             // 10
 assign and2_out = A[0] & B1_n & B0_n;                             // 11
 assign A_greater_than_B = and0_out | and1_out | and2_out;         // 12
endmodule                                                          // 13
```

Conditional Dataflow Verilog Description of Two-Bit Greater-Than Circuit

```
// Two-bit greater-than circuit: Conditional model                    // 1
// See Figure 2-27 for logic diagram                                  // 2
module comparator_greater_than_conditional2(A, B, A_greater_than_B);  // 3
 input [1:0] A, B;                                                    // 4
 output A_greater_than_B;                                             // 5
 assign A_greater_than_B = (A > B)? 1'b1 :                            // 6
      1'b0;                                                           // 7
endmodule                                                             // 8
```


Verilog Description of Two-Bit Greater-Than Circuit

```
// Two-bit greater-than circuit: Behavioral model                    // 1
// See Figure 2-27 for logic diagram                                 // 2
module comparator_greater_than_behavioral(A, B, A_greater_than_B);   // 3
 input [1:0] A, B;                                                   // 4
 output A_greater_than_B;                                            // 5
 assign A_greater_than_B = A > B;                                    // 6
endmodule                                                            // 7
```

A Design Hierarchy

- **Module Instances**
  - MUX_2 module contains references to each of the lower level modules

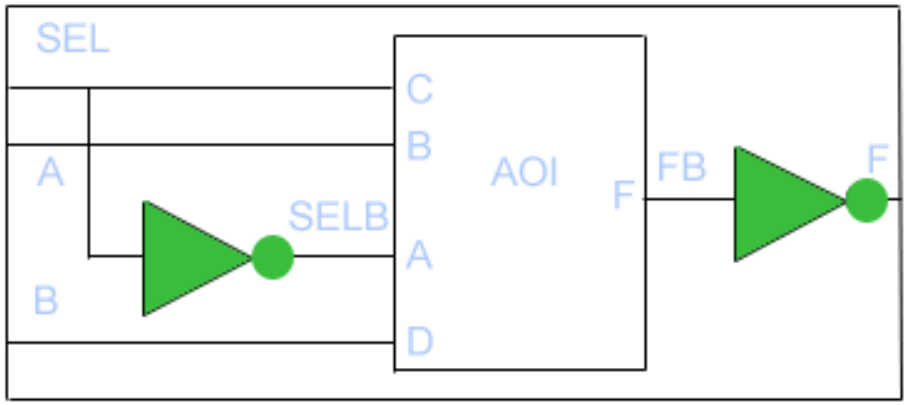
```
// Verilog code for 2-input multiplexer
module MUX2 (input SEL, A, B, output F);
// 2:1 multiplexer
// wires SELB and FB are implicit
// Module instances…
 INV G1 (SEL, SELB);
 AOI G2 (SELB, A, SEL, B, FB);
 INV G3 (.A(FB), .F(F));    // Named mapping
endmodule
// end of Verilog code
```

```
// Verilog code for 2-input multiplexer
module INV (input A, output F);   // An inverter
  assign F = ~A;
endmodule

module AOI (input A, B, C, D, output F);
  assign F = ~((A & B) | (C & D));
endmodule
```

F = (SEL)'. A + (SEL).B
SELB = (SEL)'
F=(SELB).A + (SEL).B

1. Invert SEL and get SELB
2. Use AOI and get F'
3. Invert F' and get F

MUX_2



Another Example

```
module decoder (A,B, D0,D1,D2,D3);
input A,B;
output D0,D1,D2,D3;
assign  D0 = ~A&~B;
assign  D1 = ~A&B;
assign  D2 = A&~B;
assign  D3 = A&B;
endmodule
```

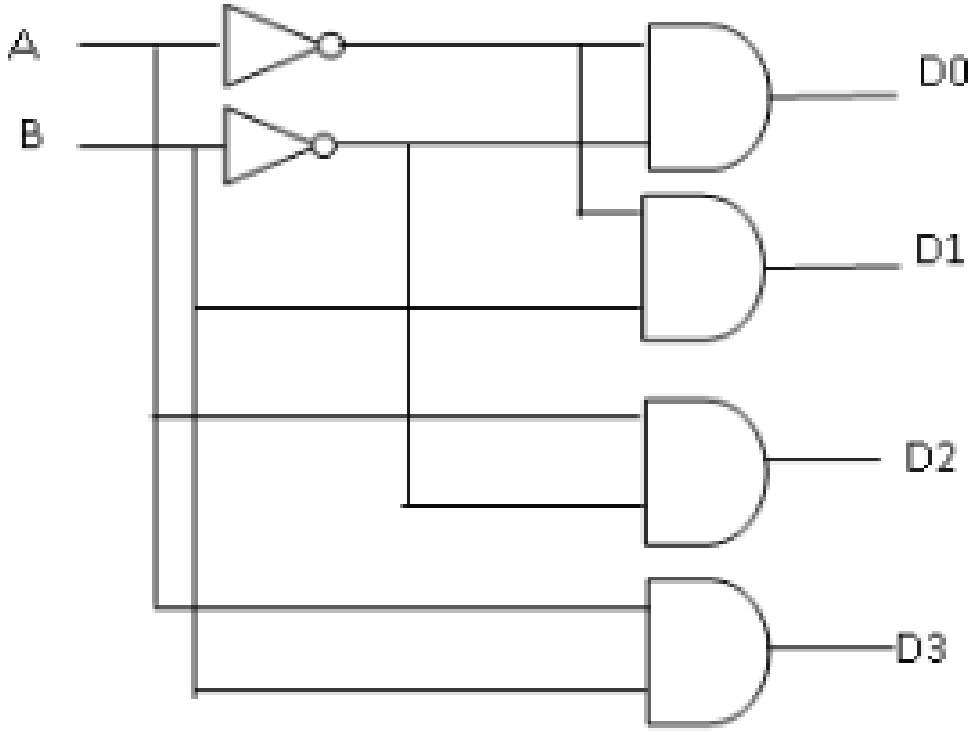
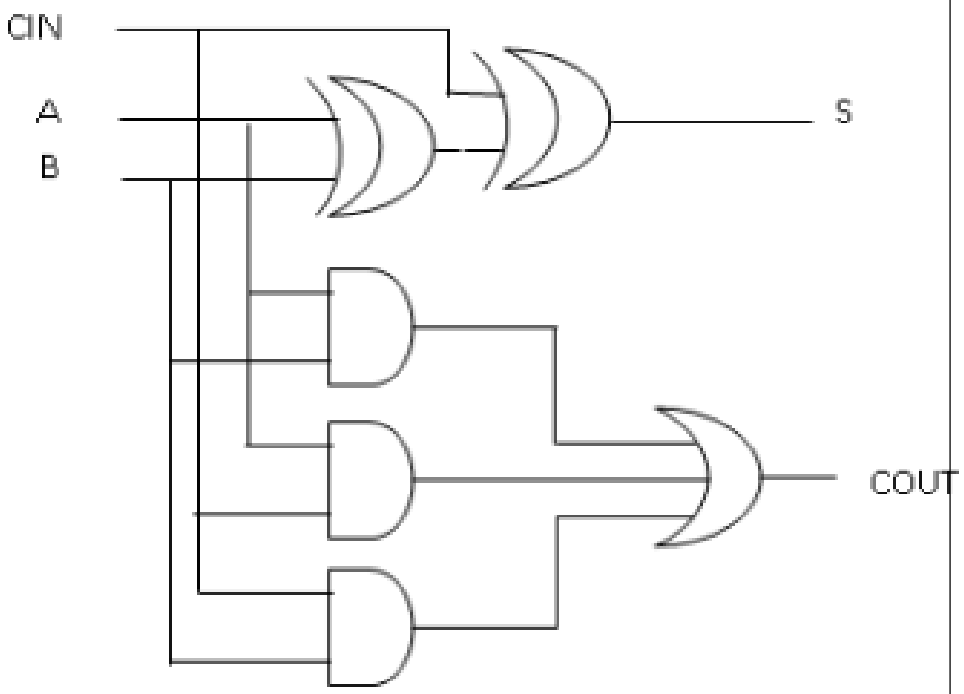


Figure 6. Logic diagram of 2-to-4 decoder

Hierarchical representation of Adder

```
module fulladder (A,B,CIN, S,COUT);
input A,B,CIN;
output S,COUT;
assign S = A ^ B ^ CIN;
assign COUT = (A &B) |(A &CIN)
| (B &CIN);
endmodule
```



Logic diagram of fulladder

```
module four_bit_adder (CIN, X3,X2,X1,X0,Y3,Y2,Y1,Y0, S3,S2,S1,S0,COUT);
input     CIN, X3, X2, X1, X0,Y3,Y2,Y1,Y0;
output    S3, S2, S1, S0, COUT;
wire      C1, C2, C3;
fulladder FA0 (X0,Y0, CIN, S0, C1);
fulladder FA1 (X1,Y1, C1, S1, C2);
fulladder FA2 (X2,Y2, C2, S2, C3);
fulladder FA3 (X3,Y3, C3, S3, COUT);
endmodule
```

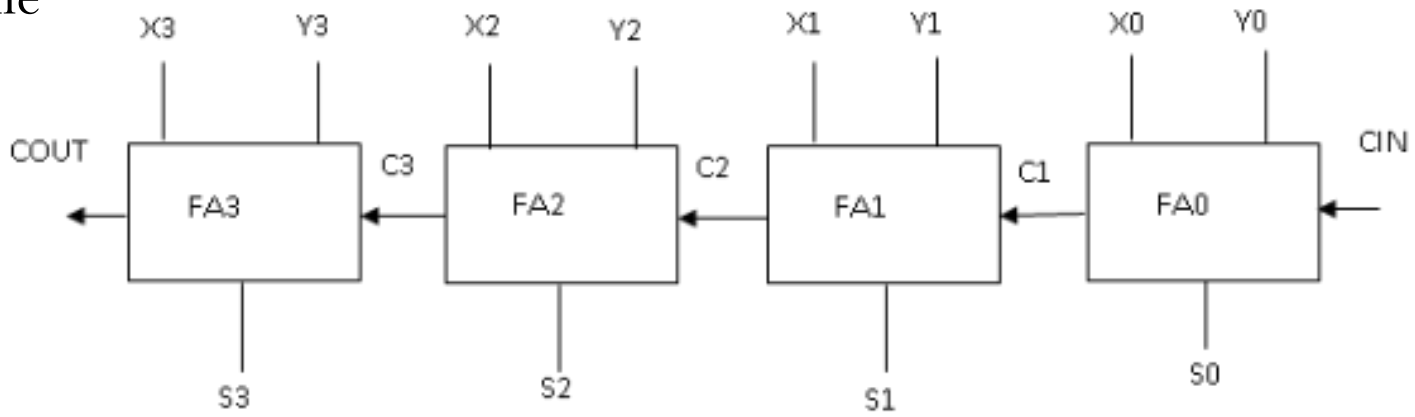


Figure 9. Four bit Full Adder

```
module adder_4 (A, B, CIN, S ,COUT);
input [3:0] A,B;
input CIN;
output [3:0] S;
output COUT;
wire  [4:0] C;
full _adder  FA0 (B(0), A(0), C(0), S(0), C(1));
full _adder  FA1 (B(1), A(1), C(1), S(1), C(2));
full _adder  FA2 (B(2), A(2), C(2), S(2), C(3));
full _adder  FA3 (B(3), A(3), C(3), S(3), C(4));
assign C(0) = CIN;
assign COUT = C(4);
endmodule
```

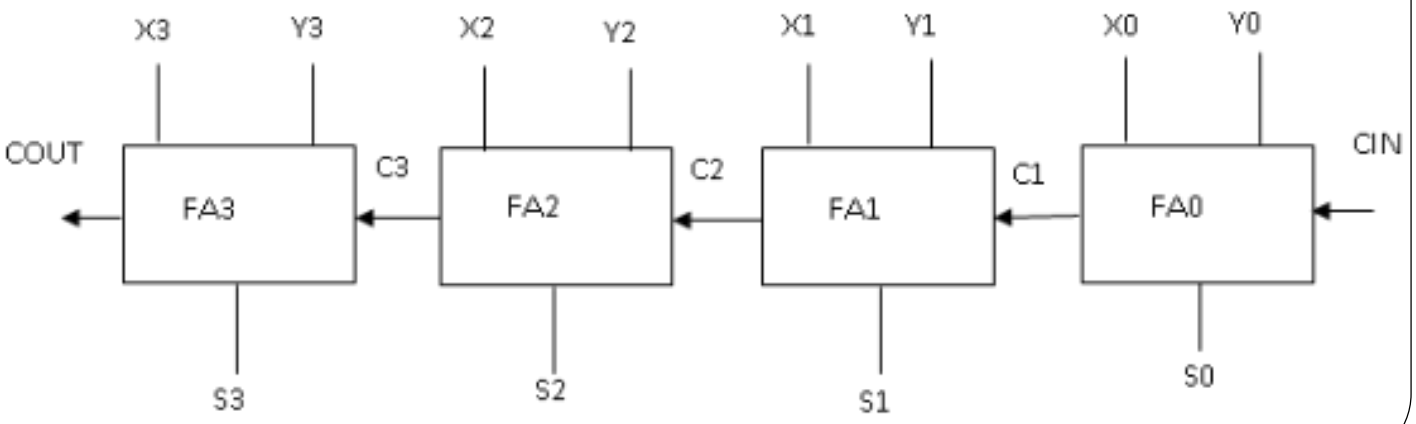


Figure 9. Four bit Full Adder

Verilog Statements

Verilog has two basic types of statements

1. Concurrent statements (combinational)

   (things are happening concurrently, ordering does not matter)

   - Gate instantiations
     - **and** (z, x, y), **or** (c, a, b), **xor** (S, x, y), etc.
   - Continuous assignments
     - **assign** Z = x & y; c = a | b; S = x ^ y

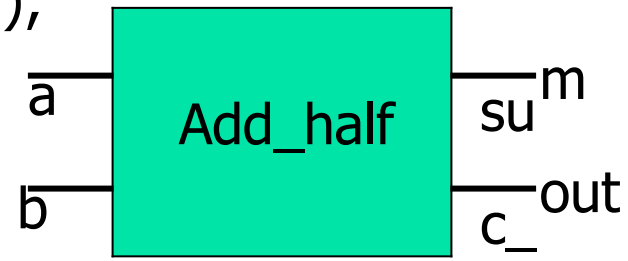
2. Procedural statements (sequential)

   (executed in the order written in the code)

   - **always** @ - executed continuously when the event is active
   - **Initial** - executed only once (used in simulation)
   - **if then else** statements

Behavioral Description

```
module Add_half ( sum, c_out, a, b );
   input      a, b;
   output     sum, c_out;
   reg sum,  c_out;
   always @ ( a or b )
      begin
         sum = a ^ b;          // Exclusive or
         c_out = a & b;        // And
      end
endmodule
```



Must be of the 'reg' type

Procedure assignment statements

Event control expression or sensitivity list

Conditional Statement

- Conditional_expression ? true_expression : false expression;

Example:

- Assign A = (B<C) ? (D+5) : (D+2);
  - if B is less than C, the value of A will be D + 5, or else A will have the value D + 2.

- An **if-else** statement is a procedural statement.

```
//Behavioral specification
module mux2to1  (w0, w1, s, F);
input wo,w1,s;
output F;
reg F;
```

```
always @ (w0,w1,s)
if (s==1) F = w1;
else F = w0;
endmodule
```

sensitivity list

```
always @ (w0,w1,s)
F = s ? w1: w2;
endmodule
```

Mux 4-to-1

```
module mux4to1 (w0, w1,w2, w3, S, F);
input w0,w1,w2,w3,[1:0] S;
output F;
reg  F;
always @ (w0,w1,w2,w3,S)
if (S==0) F = w0;
else if (S==1) F = w1;
else if (S==2) F = w2;
else F = w3;
endmodule
```

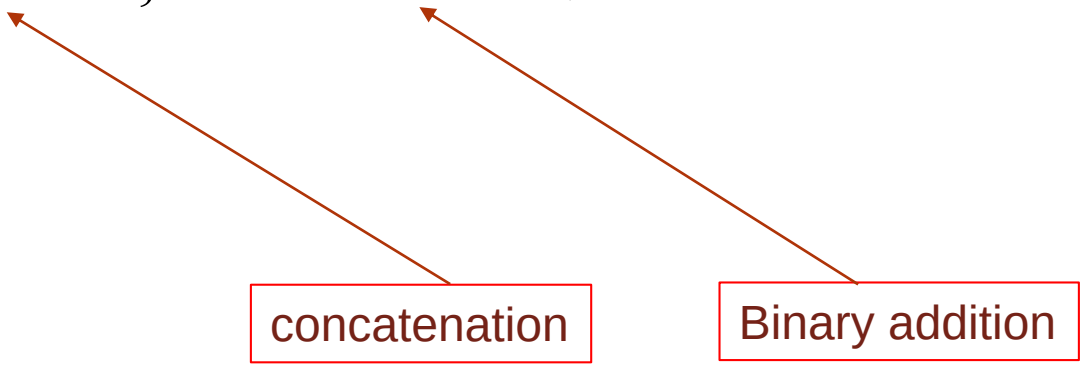
Verilog Operator	Name	Functional Group
> >= < <=	greater than greater than or equal to less than less than or equal to	relational
== !=	case equality case inequality	equality
& ^ |	bit-wise AND bit-wise XOR bit-wise OR	bit-wise bit-wise
&& ||	logical AND logical OR	logical

Another Example

```
//Dataflow description of a 4-bit comparator.
module mag_comp (A,B,ALTB,AGTB,AEQB);
input [3:0] A,B;
output ALTB,AGTB,AEQB;
assign ALTB = (A < B),
AGTB = (A > B),
AEQB = (A == B);
endmodule
```

Dataflow Modeling

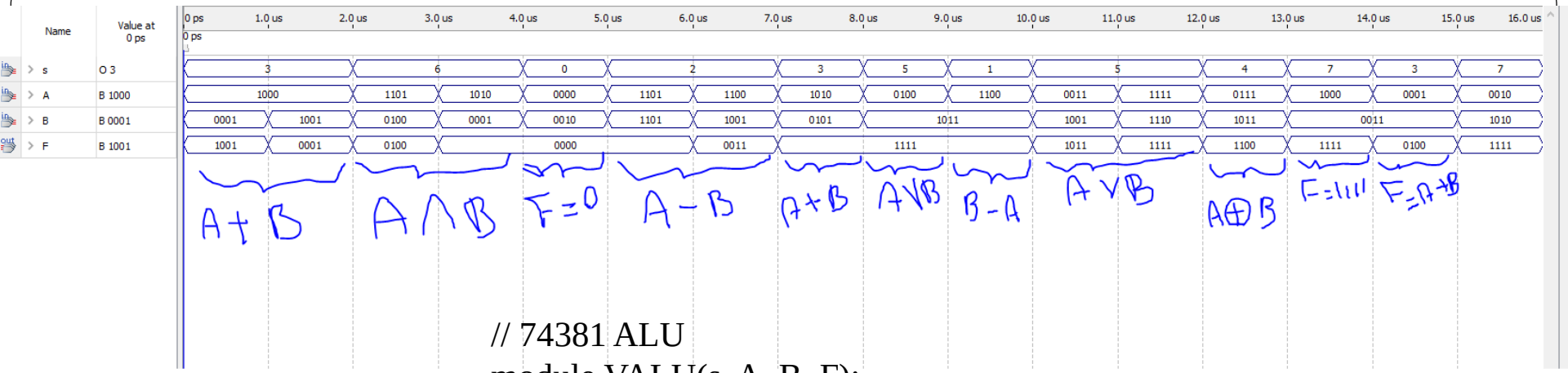
```
//Dataflow description of 4-bit adder
module binary_adder (A, B, Cin, SUM, Cout);
input [3:0] A,B;
input Cin;
output [3:0] SUM;
output Cout;
assign {Cout, SUM} = A + B + Cin;
endmodule
```



Design of an ALU using Case Statement

S	Function
0	Clear
1	B-A
2	A-B
3	A+B
4	A XOR B
5	A OR B
6	A AND B
7	Set to all 1's

```
// 74381 ALU
module alu(s, A, B, F);
input [2:0] s;
input [3:0] A, B;
output [3:0] F;
reg [3:0] F;
always @(s or A or B)
case (s)
0: F = 4'b0000;
1: F = B - A;
2: F = A - B;
3: F = A + B;
4: F = A ^ B;
5: F = A | B;
6: F = A & B;
7: F = 4'b1111;
endcase
endmodule
```




```
// 74381 ALU
module VALU(s, A, B, F);
input [2:0] s;
input [3:0] A, B;
output [3:0] F;
reg [3:0] F;
always @(s or A or B)
case (s)
0: F = 4'b0000;
1: F = B - A;
2: F = A - B;
3: F = A + B;
4: F = A ^ B;
5: F = A | B;
6: F = A & B;
7: F = 4'b1111;
endcase
endmodule
```

Blocking vs. Nonblocking Assignments

- Verilog supports two types of assignments within *always* blocks, with subtly different behaviors.
- *Blocking assignment:* evaluation and assignment are immediate

```
always @ (a or b or c)
begin
   x = a | b;          1. Evaluate a | b, assign result to x
   y = a ^ b ^ c;      2. Evaluate a^b^c, assign result to y
   z = b & ~c;         3. Evaluate b&(~c), assign result to z
end
```

- *Nonblocking assignment:* all assignments deferred until all right-hand sides have been evaluated (end of simulation timestep)

```
always @ (a or b or c)
begin
   x.<= a | b;         1. Evaluate a | b but defer assignment of x
   y.<= a ^ b ^ c;     2. Evaluate a^b^c  but defer assignment of y
   z <= b & ~c;        3. Evaluate b&(~c) but defer assignment of z
end                    4. Assign x, y, and z with their new values
```

- **Sometimes, as above, both produce the same result. Sometimes, not!**

Blocking vs. Nonblocking Assignments

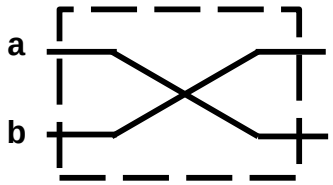
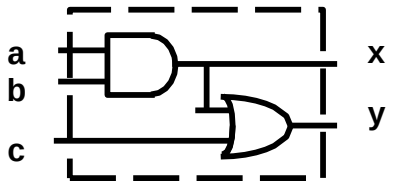
- The = token represents a blocking blocking procedural assignment
  - ✓ Evaluated and assigned in a single step
  - ✓ Execution flow within the procedure is blocked until the assignment is completed
- The <= token represents a non-blocking assignment
  - ✓ Evaluated and assigned in two steps:
    1. The right hand side is evaluated immediately
    2. The assignment to the left-hand side is postponed until other evaluations in the current time step are completed

```
//swap bytes in word
always @(posedge clk)
begin
word[15:8] = word[ 7:0];
word[ 7:0] = word[15:8];
end
```

```
//swap bytes in word
always @(posedge clk)
begin
word[15:8] <= word[ 7:0];
word[ 7:0] <= word[15:8];
end
```

Why two ways of assigning values?

Conceptual need for two kinds of assignment (in always blocks):

		
Blocking: Evaluation and assignment are immediate	~~a = b~~ ~~b = a~~	x = a & b y = x | c
Non-Blocking: Assignment is postponed until all r.h.s. evaluations are done	a <= b b <= a	~~x <= a & b~~ ~~y <= x | c~~
When to use: (only in always blocks!)	Sequential Circuits	Combinational Circuits

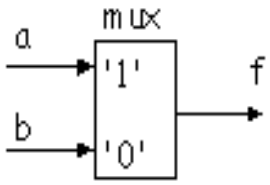
Golden Rules

- **Golden Rule 1:**
  - *To synthesize combinational logic using an always block, all inputs to the design must appear in the sensitivity list.*

- **Golden Rule 2:**
  - *To synthesize combinational logic using an always block, all variables must be assigned under all conditions.*

Golden Rules

```
reg f;
always @ (sel, a, b)
  begin :
    if (sel == 1)
      f = a;
    else
      f = b;
  end
```

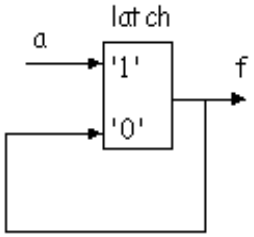


- Proper as intended

```
Reg f;
always @ (sel, a, b)
begin f = b;
    if (sel == 1)
      f = a;
  end
```

- Setting variables to default values at the start of the always block
- OK as well!

```
reg f;
always @ (sel, a)
  begin :
    if (sel == 1)
      f = a;
  end
```



- What if sel = 0?
  - Keep the current value
- Undesired functionality
  - Unintended latch
- Need to include else

Verilog Operator	Name	Functional Group
[ ]	bit-select or part-select	
( )	parenthesis	
!	logical negation	logical
~	negation	bit-wise
&	reduction AND	reduction
|	reduction OR	reduction
~&	reduction NAND	reduction
~|	reduction NOR	reduction
^	reduction XOR	reduction
~^ or ^~	reduction XNOR	reduction
+	unary (sign) plus	arithmetic
-	unary (sign) minus	arithmetic
{ }	concatenation	concatenation
{{ }}	replication	replication
*	multiply	arithmetic
/	divide	arithmetic
%	modulus	arithmetic

Verilog Operator	Name	Functional Group
+	binary plus	arithmetic
-	binary minus	arithmetic
<<	shift left	shift
>>	shift right	shift
>	greater than	relational
>=	greater than or equal	relational
<	to	relational
<=	less than	relational
	less than or equal to	
==	case equality	equality
!=	case inequality	equality
&	bit-wise AND	bit-wise
^	bit-wise XOR	bit-wise
|	bit-wise OR	bit-wise
&&	logical AND	logical
||	logical OR	logical
?:	conditional	conditional

Appendix

Arithmetic in Verilog

```
module Arithmetic (A, B,Y1,Y2,Y3,Y4,Y5);
        input [2:0] A, B;
        output [3:0]Y1;
        output [4:0]Y3;
        output [2:0]Y2,Y4,Y5;
        reg [3:0]Y1;
        reg [4:0]Y3;
        reg [2:0]Y2,Y4,Y5;
        always @(A or B)
        begin
                Y1=A+B;//addition
                Y2=A-B;//subtraction
                Y3=A*B;//multiplication
                Y4=A/B;//division
                Y5=A%B;//modulus of A divided by B
        end
endmodule
```

Sign Arithmetic in Verilog

```
module Sign (A, B,Y1,Y2,Y3);

        input [2:0] A, B;
        output [3:0]Y1,Y2,Y3;
        reg [3:0]Y1,Y2,Y3;

        always @(A or B)
        begin
                Y1=+A/-B;
                Y2=-A+-B;
                Y3=A*-B;
        end
endmodule
```

Equality and inequality Operations in Verilog

```
module Equality (A, B,Y1,Y2,Y3);
    input [2:0] A, B;
    output Y1,Y2;
    output [2:0]Y3;
    reg Y1,Y2;
    reg [2:0]Y3;
    always @(A or B)
    begin
        Y1=A==B;//Y1=1 if A equivalent to B
        Y2=A!=B;//Y2=1 if A not equivalent to B
        if (A==B)//parenthesis needed
            Y3=A;
        else
            Y3=B;
    end
endmodule
```

Logical Operations in Verilog

```
module Logical (A, B, C, D, E, F,Y);
        input [2:0] A, B, C, D, E, F;
        output Y;
        reg Y;
        always @(A or B or C or D or E or F)
        begin
                if ((A==B) && ((C>D) || !(E<F)))
                        Y=1;
                else
                        Y=0;
        end
endmodule
```

Bit-wise Operations in Verilog

```
module Bitwise (A, B,Y);
        input [6:0] A;
        input [5:0] B;
        output [6:0]Y;
        reg [6:0]Y;
        always @(A or B)
        begin
                Y[0]=A[0]&B[0]; //binary AND
                Y[1]=A[1]|B[1]; //binary OR
                Y[2]=!(A[2]&B[2]); //negated AND
                Y[3]=!(A[3]|B[3]); //negated OR
                Y[4]=A[4]^B[4]; //binary XOR
                Y[5]=A[5]~^B[5]; //binary XNOR
                Y[6]=!A[6]; //unary negation
        end
endmodule
```

. Concatenation and Replication in Verilog

- The concatenation operator "{ , }" combines (concatenates) the bits of two or more data objects. The objects may be scalar (single bit) or vectored (multiple bit). Multiple concatenations may be performed with a constant prefix and is known as replication.

```
module Concatenation (A, B,Y);
        input [2:0] A, B;
        output [14:0]Y;
        parameter C=3'b011;
        reg [14:0]Y;
        always @(A or B)
        begin
                Y={A, B, {2{C}}, 3'b110};
        end
endmodule
```

Shift Operations in Verilog

```
module Shift (A,Y1,Y2);
        input [7:0] A;
        output [7:0]Y1,Y2;
        parameter B=3; reg [7:0]Y1,Y2;
          always @(A)
        begin
                Y1=A<<B; //logical shift left
                Y2=A>>B; //logical shift right
        end
endmodule
```

Conditional Operations in Verilog

```
module Conditional (Time,Y);
        input [2:0]Time;
        output [2:0]Y;
        reg [2:0]Y;
        parameter Zero =3b'000;
        parameter TimeOut = 3b'110;
        always @(Time)
        begin
                Y=(Time!=TimeOut) ?Time +1 : Zero;
        end
endmodule
```

Reduction Operations in Verilog

```
module Reduction (A,Y1,Y2,Y3,Y4,Y5,Y6);
        input [3:0] A;
       output Y1,Y2,Y3,Y4,Y5,Y6;
       reg Y1,Y2,Y3,Y4,Y5,Y6;
        always @(A)
       begin
                Y1=&A; //reduction AND
                Y2=|A; //reduction OR
                Y3=~&A; //reduction NAND
                Y4=~|A; //reduction NOR
                Y5=^A; //reduction XOR
                Y6=~^A; //reduction XNOR
       end
endmodule
```

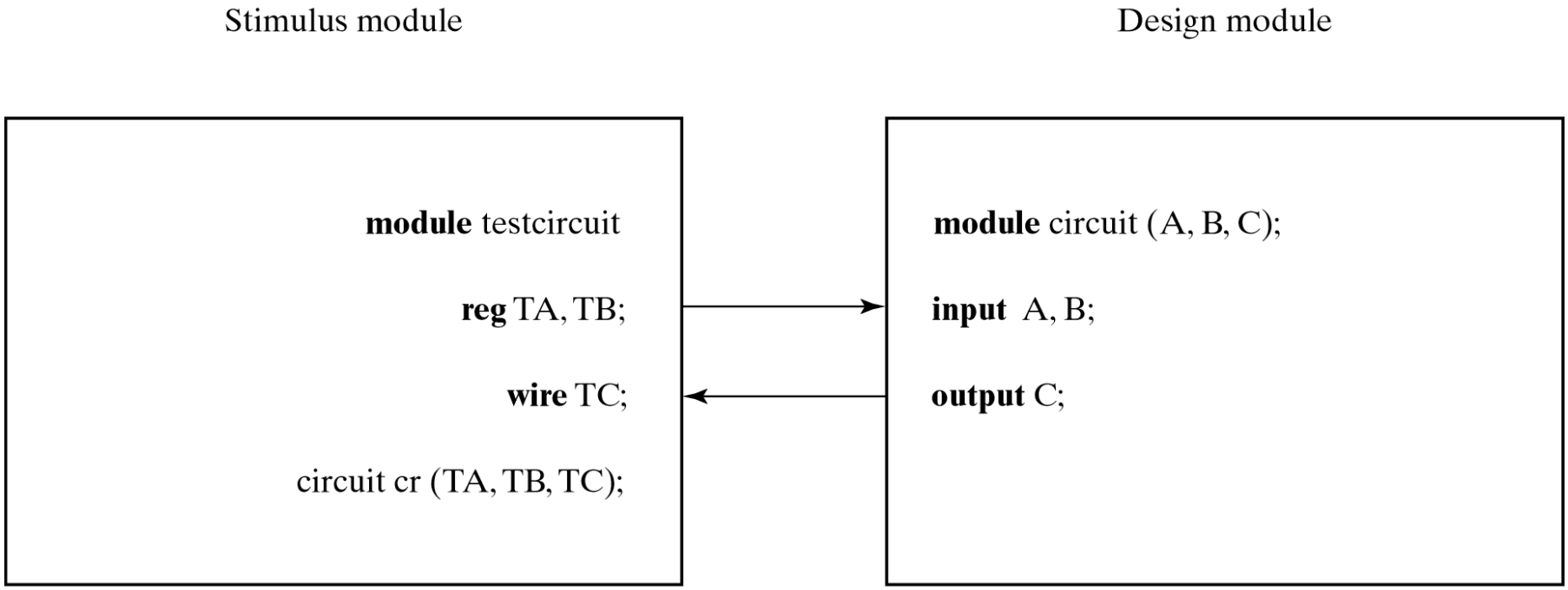
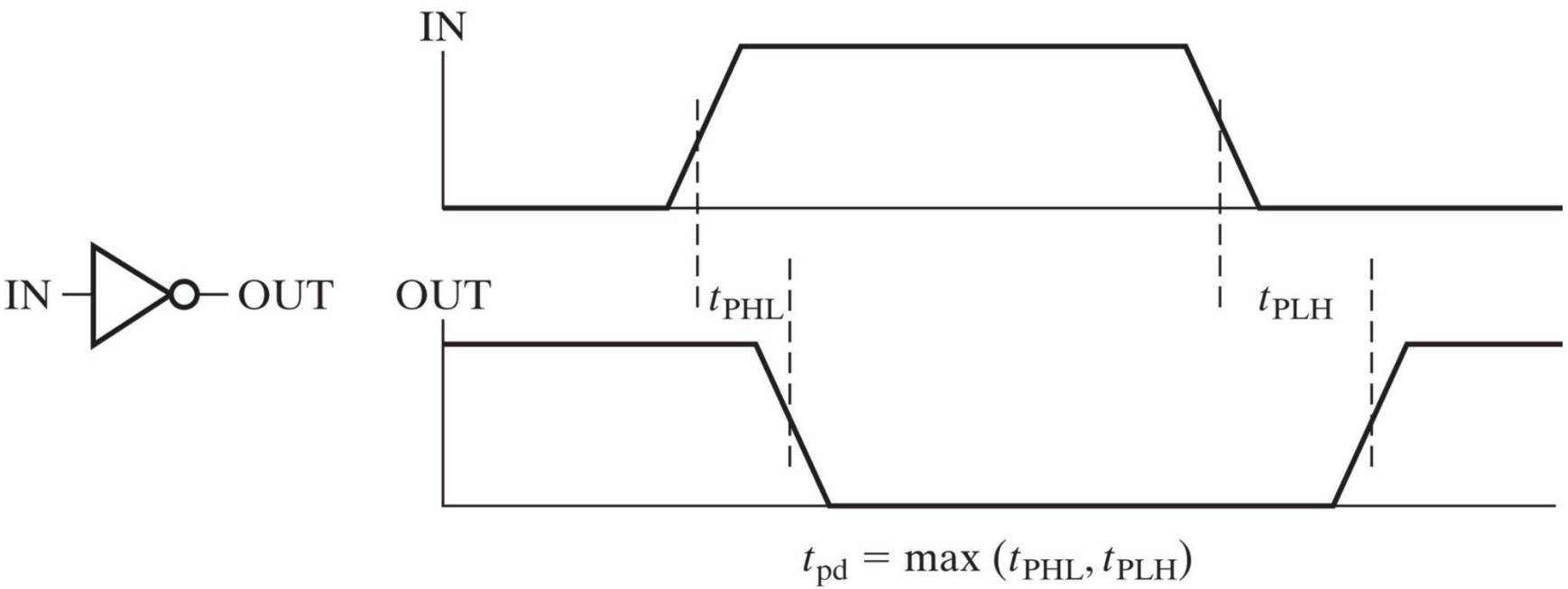


Fig. 4-33 Stimulus and Design Modules Interaction

Testbench for the Structural Model of the Two-Bit Greater-Than Comparator

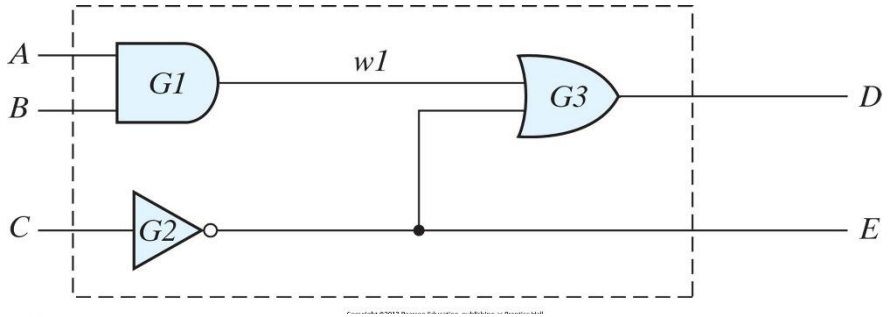
```
// Testbench for Verilog two-bit greater-than comparator          //  1
module comparator_testbench_verilog();                            //  2
 reg [1:0] A, B;                                                  //  3
 wire struct_out;                                                 //  4
 comparator_greater_than_structural U1(A, B, struct_out);         //  5
 initial                                                          //  6
 begin                                                            //  7
  A = 2'b10;                                                      //  8
  B = 2'b00;                                                      //  9
  #10;                                                            // 10
  B = 2'b01;                                                      // 11
  #10;                                                            // 12
  B = 2'b10;                                                      // 13
  #10;                                                            // 14
  B = 2'b11;                                                      // 15
 end                                                              // 16
endmodule                                                         // 17
```


Propagation Delay for an Inverter



$$t_{pd} = \max(t_{PHL}, t_{PLH})$$

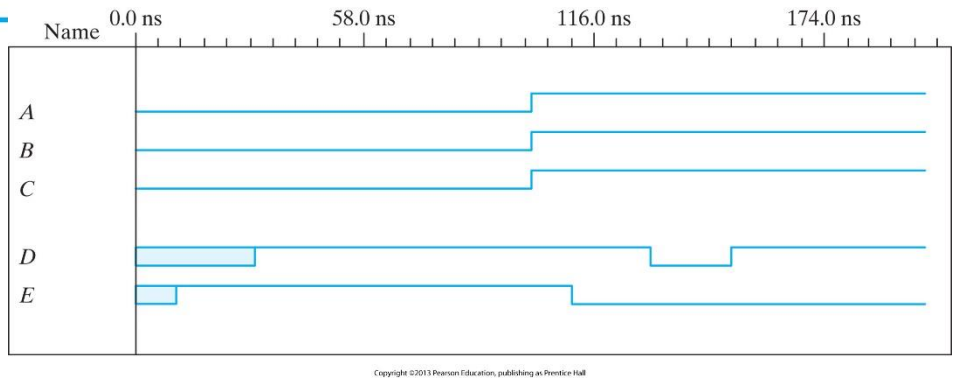
Circuit to demonstrate an HDL (Verilog)




```
Module smpl_Circuit (A, B, C, D, E)
        input A, B, C;
        output D, E;
        wire  w1;
        and # (30) G1 (w1, A, B);
        not #10 G2 (E, C);
        or #(20) G3 (D, w1, E);
        endmodule
```

Table 3.5
Output of Gates after Delay

	Time Units (ns)	Input ABC	Output E w1 D
Initial	—	0 0 0	1 0 1
Change	—	1 1 1	1 0 1
	10	1 1 1	0 0 1
	20	1 1 1	0 0 1
	30	1 1 1	0 1 0
	40	1 1 1	0 1 0
	50	1 1 1	0 1 1



Interaction between stimulus and design modules