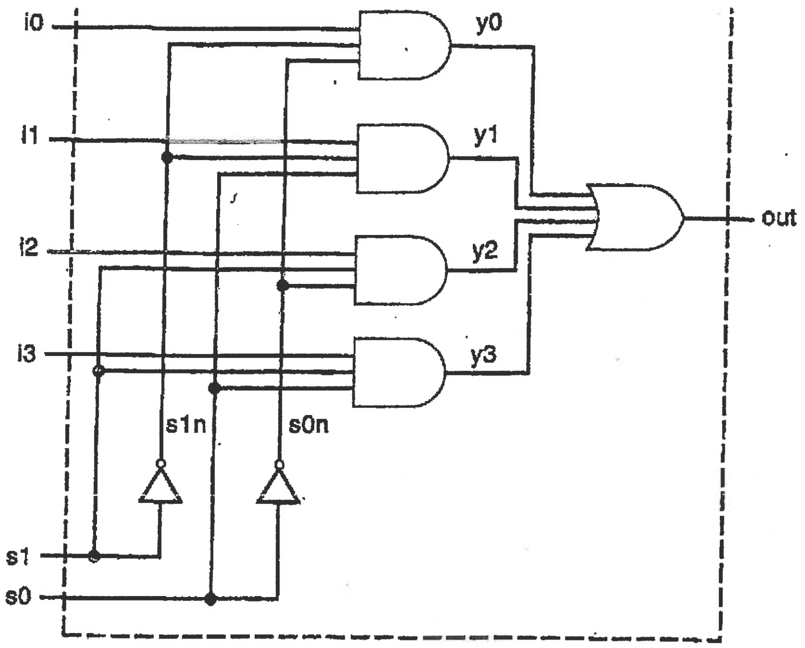


Figure 5-5 *Logic Diagram for Multiplexer*

Example 5-5 *Verilog Description of Multiplexer*

```
// Module 4-to-1 multiplexer. Port list is taken exactly from
// the I/O diagram.
module mux4_to_1 (out, i0, i1, i2, i3, s1, s0);

// Port declarations from the I/O diagram
output out;
input i0, i1, i2, i3; // data inputs
input s1, s0;         // select lines

// Internal wire declarations
wire s1n, s0n;
wire y0, y1, y2, y3;

// Gate instantiations

// Create s1n and s0n signals.
not (s1n, s1);
not (s0n, s0);

// 3-input and gates instantiated
and (y0, i0, s1n, s0n);
and (y1, i1, s1n, s0);
and (y2, i2, s1, s0n);
and (y3, i3, s1, s0);

// 4-input or gate instantiated
or (out, y0, y1, y2, y3);

endmodule
```

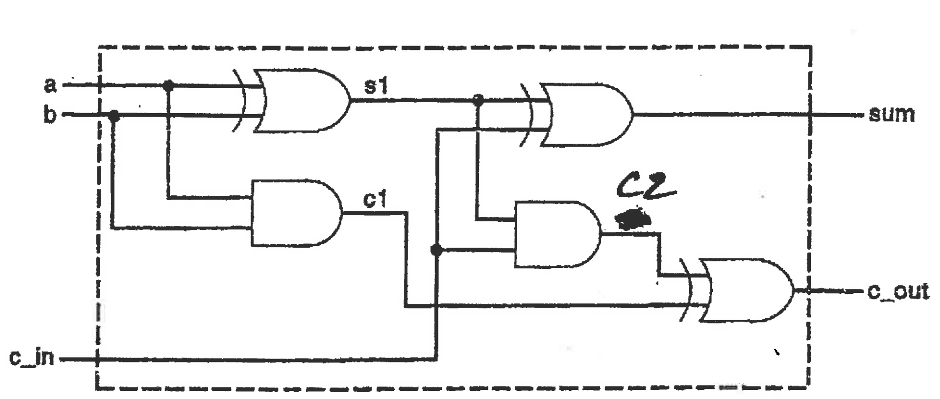


Figure 5-6 *1-bit Full Adder*

This logic diagram for the 1-bit full adder is converted to a Verilog description, shown in Example 5-7.

Example 5-7 *Verilog Description for 1-bit Full Adder*

```
// Define a 1-bit full adder
module fulladd(sum, c_out, a, b, c_in);

// I/O port declarations
output sum, c_out;
input a, b, c_in;

// Internal nets
wire s1, c1, c2;


// Instantiate logic gate primitives
xor (s1, a, b);
and (c1, a, b);

xor (sum, s1, c_in);
and (c2, s1, c_in);

xor  (c_out, c2, c1);

endmodule
```

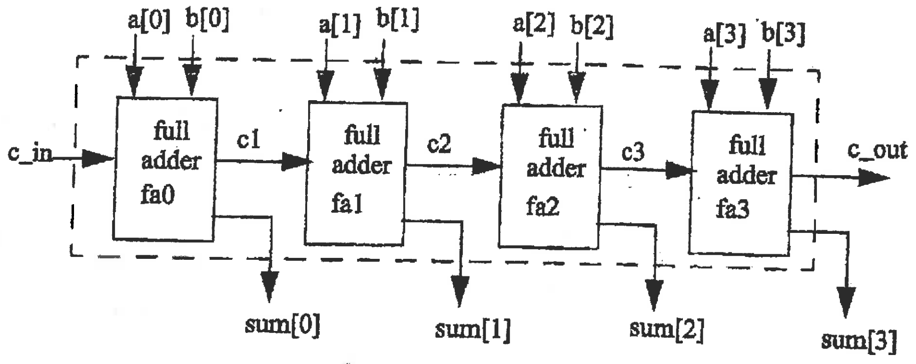


Figure 5-7 *4-bit Ripple Carry Full Adder*

Example 5-8 *Verilog Description for 4-bit Ripple Carry Full Adder*

```
// Define a 4-bit full adder
module fulladd4(sum, c_out, a, b, c_in);

// I/O port declarations
output [3:0] sum;
output c_out;
input[3:0] a, b;
input c_in;

// Internal nets
wire c1, c2, c3;

// Instantiate four 1-bit full adders.
fulladd fa0(sum[0], c1, a[0], b[0], c_in);
fulladd fa1(sum[1], c2, a[1], b[1], c1);
fulladd fa2(sum[2], c3, a[2], b[2], c2);
fulladd fa3(sum[3], c_out, a[3], b[3], c3);

endmodule
```

and #5(e, a, b);
or #4(out, e, c);

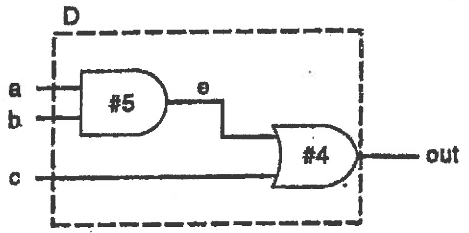


Figure 5-8 Module D

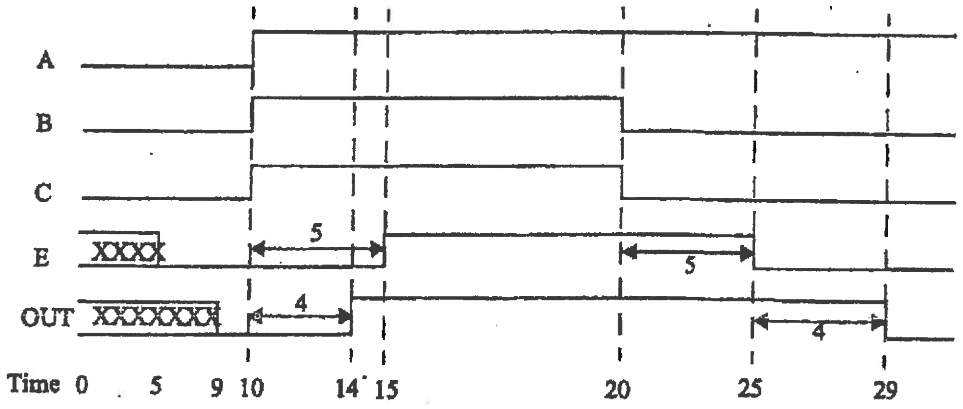


Figure 5-9 Waveforms for Delay Simulation

#delays ain't synthesizable

dataflow level of abstraction

reg variables hold values

e.g. `A = 3'b011;`

net variables don't

2 ways to give net variables a value

- drive with a module output
- with a continuous assignment statement

Continuous assignment statement

syntax: assign LHS = RHS ;

don't forget

keyword

variable identifier (for your net variable)

expression (arithmetic or logical)

notes: 1) Suppose we declare

wire [7:0] A;

A[2] $\longrightarrow$ bit select

$\underbrace{A[4:1]}_{\text{must be contiguous}}$ $\longrightarrow$ part select

2) the continuous assignment statement

<u>continuously</u>

dRives the net variable

3) LHS updates whenever the RHS changes

e.g. assign A = C&D; //A = C·D

4) actual syntax is

$\text{assign } \underbrace{[\text{strength level}]\ [\text{delay}]}_{\text{not synthesizable}}\ \text{LHS} = \text{RHS};$

Defn (expression)
a combination of operators and operands
to get some result

Defn (operand)
a data element (e.g. a net variable or constant)

Defn (operator)
anything that acts on operands

Arithmetic

```
assign A = B + C;   // A = B + C
```

the LHS size is determined by
the size of the largest operand

e.g.
```
wire [3:0] A, B, C;
wire [5:0] D;
   :
A = B + C;   // 4 bit result
D = B + C;   // 6-bit result
```

Note: if any bit on the RHS is "X", the result is "X"

Logical operators

the entire variable is considered

logic ϕ or logic 1 or X or Z

$\uparrow$

always the case if the operand $\neq 0$

(or x or z)

e.g. $A = 3,\ B = 0,\ C = -493$

$\underbrace{A}_{\text{Logic } 1} \&\& \underbrace{B}_{\text{logic } \phi} \Rightarrow \text{Logic } \phi$

$A \&\& \underbrace{C}_{\text{Logic } 1} \Rightarrow \text{Logic } 1$

6.4.5 Bitwise Operators

Bitwise operators are *negation* (~), *and*(&), *or* (|), *xor* (^), *xnor* (^~, ~^). Bitwise operators perform a bit-by-bit operation on two operands. They take each bit in one operand and perform the operation with the corresponding bit in the other operand. If one operand is shorter than the other, it will be bit-extended with zeros to match the length of the longer operand. Logic tables for the bit-by-bit computation are shown in Table 6-3. A **z** is treated as an **x** in a bitwise operation. The exception is the unary negation operator (~), which takes only one operand and operates on the bits of the single operand.

Table 6-3 Truth Tables for Bitwise Operators

bitwise and	0	1	x
0	0	0	0
1	0	1	x
x	0	x	x

bitwise or	0	1	x
0	0	1	x
1	1	1	1
x	x	1	x

bitwise xor	0	1	x
0	0	1	x
1	1	0	x
x	x	x	x

bitwise xnor	0	1	x
0	1	0	x
1	0	1	x
x	x	x	x

bitwise negation	result
0	1
1	0
x	x

Examples of bitwise operators are shown below.

```
// X = 4'b1010, Y = 4'b1101
// Z = 4'b10x1

~X      // Negation. Result is 4'b0101
X & Y   // Bitwise and. Result is 4'b1000
X | Y   // Bitwise or. Result is 4'b1111
X ^ Y   // Bitwise xor. Result is 4'b0111
```

__in C__

{ } used to group things

__e.g.__

```
for(i=0;i<10;i++)
{
  ≡
}
```

__in Verilog__ { } concantenation operater

__e.g.__

```
wire [7:0] D;

assign D[7:4] = {D[0], D[1], D[3], D[6]};
                 =1    =0    =1    =1

then D[7:4] = 4'b1011;
```

e.g.

```
assign D = {D[3:0], D[7:4]};
```

Reduction operator

e.g.

```
&y;
|y;
^y;
 :
```

} applies the bitwise operator on pairs of bits from right to left

e.g.

$y = 6'b\underbrace{0}\underbrace{1}\underbrace{1}\underbrace{0}\underbrace{1}0$

$\hat{}y = 1'b1$

<u>shift operators</u>

syntax : LHS = var $\overset{\text{shift direction}}{\overbrace{\begin{matrix}<<\\>>\end{matrix}}}$ $\underbrace{\text{decimal \#}}_{\text{\# of bit positions shifted}}$

e.g.

A = 4'b0010

A >> 1 = 4'b0001

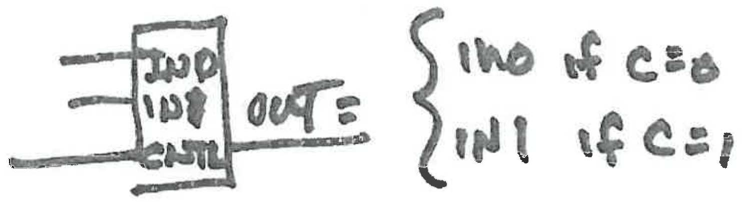
A << 3 = 4'b0000

Conditional operator

```
assign LHS = predicate ? value1 : value2;
                 ↑
            true or false

   if true    LHS = value1
   if false   LHS = value2
```

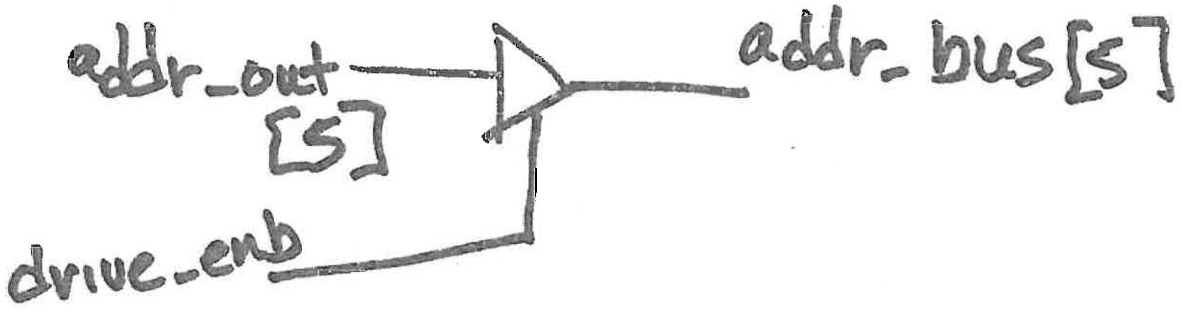
one line if-then-else statement



```
//model functionality of a tristate buffer
assign addr_bus = drive_enable ? addr_out : 36'bz;

//model functionality of a 2-to-1 mux
assign out = control ? in1 : in0;
```

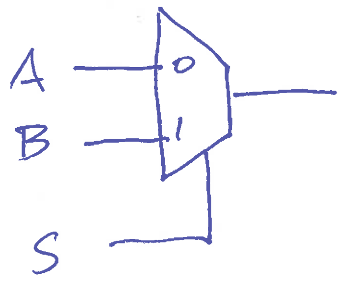
pg 105



```
assign ~~out = (A == 3) ? ( control ? x : y ) : ( control ? m : n) ;~~
```

Do on board

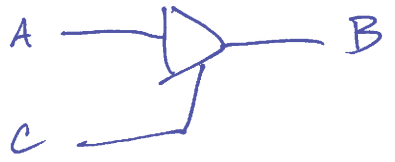
2x1 MUX



$$Y = \begin{cases} A & \text{if } S=0 \\ B & \text{if } S=1 \end{cases}$$

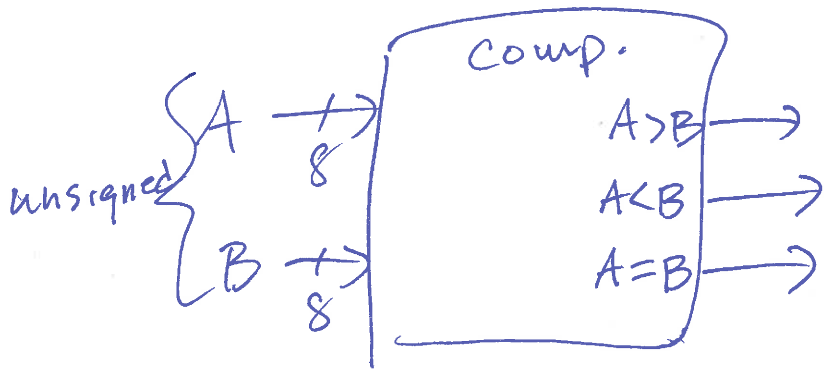
```
assign Y = S ? B : A ;
```

tri-state buffer



```
assign B = C ? A : 1'bz ;
```

8-bit comparator



```
module comp(A,B,AGTB,ALTB,AEQB)
 input [7:0] A,B;
 output AGTB,AEQB,ALTB;

    assign  AGTB = A>B;
    assign  ALTB = A<B;
    assign  AEQB = A==B;
 endmodule
```