Majority Gate with Temporary Signals

The following version of the majority gate uses some temporary "wires"

```
// Majority Logic Circuit
module maj_circ(Y, A, B, C);
   input A, B, C;
   output Y;
   wire  x1, x2, x3;  //Optional
   assign x1 = A&B;
   assign x2 = A&C;
   assign x3 = B&C;
   assign Y = x1|x2|x3;
endmodule
```


Concurrent Assignment with Ternary Select

```
assign mux_out = (select==1'b0) ? q0 : q1;
```

`if` statement is procedural (sequential)

Used inside **`begin`** – **`end`** Block

Similar to **`if`** Statement but Concurrent Version

Majority Gate with conditional statement

The following version of the majority gate uses a conditional concurrent statement:

```
// Majority Logic Circuit
module maj_circ(Y, A, B, C);
   input A, B, C;
   output Y;
     assign Y = ((A&&B)||(A&&C)||(B&&C))
          ? 1'b1   :  1'b0;
endmodule
```

You will find that there are many different ways to accomplish the same result in Verilog. There is usually no best way; just use one that you feel most comfortable with

Concurrent Versus Sequential Statements

- The statements we have looked at so far are called concurrent statements.
  - Each concurrent statement will synthesize to a block of logic.
- Another class of Verilog statements are called procedural (sequential) statements.
  - Sequential statements can ONLY appear inside of a **`always`** or an **`initial`** block.
  - An **`always`** block is considered to be a single concurrent statement
  - Can have multiple **`always`** blocks in a module
  - Usually use **`always`** blocks to describe complex combinational or sequential logic

Comments on always block model

- always statement executes at every simulator time cycle

```
always CLK = ~CLK;  //Will Loop indefinitely
```

- General Form:

always [*timing_control*] *procedural statement(s)*

- always statement with delay control

```
always #5 CLK = ~CLK;  //Waveform CLK is 10 time units
```

- always statement with event control

CL
```
always @(A or B or C) //Event on A, B, or C
```

CL
```
always @(CLK) //Event on rising-falling edge of CLK
```

Sto
```
always @(posedge CLK) //Event on rising edge of CLK
```

Sto
```
always @(negedge CLK) //Event on falling edge of CLK
```


Verilog Always Block

(a) Combinational `always` block using an `if` statement.

```
module mux2to1(s,a,b,y);
input  s,a,b;
output y;

reg y;

//use an if statement
always @(a or b or s)
begin
 if (s) y = b;
else y = a;
end

endmodule
```

(b) Combinational `always` block using Boolean operations.

```
module mux2to1(s,a,b,y);
input  s,a,b;
output y;

reg y;

//use boolean ops
always @(a or b or s)
begin
 y = (b & s)|(a & ~s);
end


endmodule
```

(c) Combinational `always` block using Boolean operations and intermediate values.

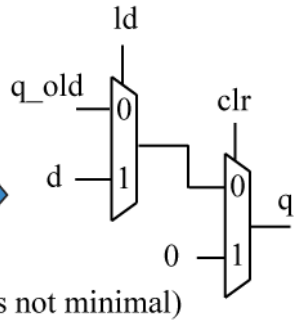
```
module mux2to1(s,a,b,y);
input  s,a,b;
output y;

reg y, na, nb;

//use intermediates
//and implicit event
//list
always @*
begin
 nb = b & s;
 na = a & ~s;
 y = na | nb;
end

endmodule
```

Precedence in Always Block

(a) `clr` takes precedence over `ld` if both are '1'

```
always @(ld or clr or d or q_old)
begin
  q = q_old;
  if (ld) q = d;
  if (clr) q = 0;
end
```

ld	clr	q
0	0	q_old
0	1	0
1	0	d
1	1	0

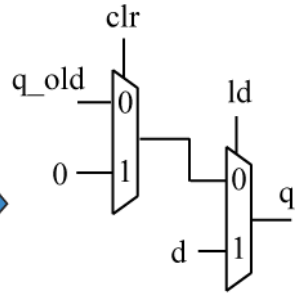


(logic is not minimal)

(b) `ld` takes precedence over `clr` if both are '1'

```
always @(ld or clr or d or q_old)
begin
  q = q_old;
  if (clr) q = 0;
  if (ld)  q = d;
end
```

ld	clr	q
0	0	q_old
0	1	0
1	0	d
1	1	d



Majority Gate using *always* block and *if* statement

```
// Majority Logic Circuit
module maj_circ(Y, A, B, C);
   input A, B, C;
   output Y;
   reg Y;
   always @(A or B or C)
     begin
       if((A==1'b1)&&(B==1'b1))
         Y = 1'b1;
       else if ((A==1'b1)&&(C==1'b1))
         Y = 1'b1;
       else if ((B==1'b1)&&(C==1'b1))
         Y = 1'b1;
       else
         Y = 1'b0;
     end
endmodule
```

Use of *if-else*

```
// Majority Logic Circuit
module maj_circ(Y, A, B, C);
   input A, B, C;
   output Y;
   reg Y;
   always @(A or B or C)
   if((A&&B)||
      (A&&C)||
      (B&&C))
      Y = 1'b1;
   else
      Y = 1'b0;
endmodule
```

Comments:
Module name is maj_circ

Used an 'else' clause to specify what the output should be if the if condition test was not true.

CAREFUL! Instead of Remembering Boolean Operator Precedence, Just Use Parentheses to Make Sure

Unassigned outputs in Always blocks

A common mistake in writing a combinational module is to leave an output unassigned. If there is a path through the block in which an output is NOT assigned a value, then that value is unassigned.

```
// Majority Logic Circuit
module bad_maj_circ(Y, A, B, C);
   input A, B, C;
   output Y;
   reg Y;
   always @(A or B or C)
   if((A&&B)||
      (A&&C)||
      (B&&C))
      Y = 1'b1;
endmodule
```

What is missing here?

Unassigned outputs in Always blocks

A common mistake in writing a combinational module is to leave an output unassigned. If there is a path through the block in which an output is NOT assigned a value, then that value is unassigned.

```
// Majority Logic Circuit
module bad_maj_circ(Y, A, B, C);
   input A, B, C;
   output Y;
   reg Y;
   always @(A or B or C)
   if((A&&B)||
      (A&&C)||
      (B&&C))
      Y = 1'b1;
endmodule
```

What if `((A&B)|(A&C)|(B&C))` equals `1'b0` ?

Comments on Previous Example

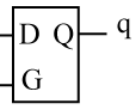
- In the previous **`always`** block, the ELSE clause was left out. If the '**`if`**' statement condition is false, then the output **`Y`** is not assigned a value.
  - In synthesis terms, this means the output **`Y`** should have a LATCH placed on it! (*INFERRED LATCH*)
  - The synthesized logic will have a latch placed on the **`Y`** output; once **`Y`** goes to a **`1'b1`**, it can NEVER return to a **`1'b0`**!!!!!
- This is probably ***the #1 student mistake*** in writing **`always`** blocks. To avoid this problem do one of the following things:
  - ALL signal outputs of the **`always`** block should have DEFAULT assignments.
  - OR, all '**`if`**' statements that affect a signal must have ELSE clauses that assign the signal a value if the '**`if`**' test is false.

Inferred Latch

2:1 Multiplexer Example

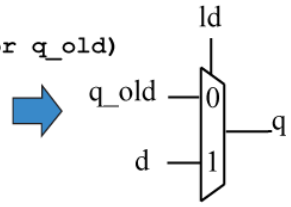
(a) Incorrect, produces an inferred latch as no assignment is made to `q` if `ld` is '0'

```
always @(ld or d)
begin
  if (ld) q = d;
end
```



(b) Correct, produces combinational logic

```
always @(ld or d or q_old)
begin
  q = q_old;
  if (ld) q = d;
end
```



Priority circuit example

```
module pri_dec(dout,y7,y6,y5,y4,y3,y2,y1);
   input y7,y6,y5,y4,y3,y2,y1;
   output  [2:0] dout;
   reg     [2:0] dout;
   always @(y7 or y6 or y5 or y4 or y3
               or y2 or y1)
     begin
      if (y7==1'b1) dout = 3'b111;
      else if (y6 == 1'b1) dout = 3'b110;
      else if (y5 == 1'b1) dout = 3'b101;
      else if (y4 == 1'b1) dout = 3'b100;
      else if (y3 == 1'b1) dout = 3'b011;
      else if (y2 == 1'b1) dout = 3'b010;
      else if (y1 == 1'b1) dout = 3'b001;
      else dout = 3'b000;
     end
endmodule
```

This priority circuit has 7 inputs;
Y7 is highest priority, Y1 is lowest priority.

Three bit output should indicate
the highest priority input that is
a '1' (ie. if Y6 ='1' , Y4 = '1', then output should be "110"). If no
input is asserted, output should be "000".

Comments on Priority Example

- The **dout** signal is a 3 bit output bus.
  - **reg [2:0] dout** describes a 3 bit bus where **dout[2]** is most significant bit, **dout[0]** is least significant bit.
  - **reg [0:2] dout** is also a 3 bit bus, but **dout[0]** is MSB, **dout[2]** is LSB. NOT RECOMMENDED!
- A bus assignment can be done in many ways:
  - **dout = 3'b110;** assigns all three bits
  - **dout[2] = 1'b1;** assigns only bit #2
  - **dout[1:0] = 2'b10;** assigns two bits of the bus.
- This module used the **else if** form of the **if** statement
  - This is called an **else if** chain.

Priority Circuit with just IF statements

```
module pri_dec(dout,y7,y6,y5,y4,y3,y2,y1);
   input y7,y6,y5,y4,y3,y2,y1;
   output  [2:0] dout;
   reg     [2:0] dout;
   always @(y7 or y6 or y5 or y4 or y3
              or y2 or y1)
     begin
      dout = 3'b000;
      if (y1==1'b1) dout = 3'b001;
      if (y2==1'b1) dout = 3'b010;
      if (y3==1'b1) dout = 3'b011;
      if (y4==1'b1) dout = 3'b100;
      if (y5==1'b1) dout = 3'b101;
      if (y6==1'b1) dout = 3'b110;
      if (y7==1'b1) dout = 3'b111;
    end
endmodule
```

By reversing the order of the assignments, we can accomplish the same as the **else if** priority chain.

In an always block, the LAST assignment to the output is what counts.

An ***attempt*** at a Priority Circuit

```
module pri_dec (dout,y7,y6,y5,y4,y3,y2,y1);
   input y7,y6,y5,y4,y3,y2,y1;
   output [2:0] dout;
       assign dout = (y1==1'b1) ? 3'b001 : 3'b000;
       assign dout = (y2==1'b1) ? 3'b010 : 3'b000;
       assign dout = (y3==1'b1) ? 3'b011 : 3'b000;
       assign dout = (y4==1'b1) ? 3'b100 : 3'b000;
       assign dout = (y5==1'b1) ? 3'b101 : 3'b000;
       assign dout = (y6==1'b1) ? 3'b110 : 3'b000;
       assign dout = (y7==1'b1) ? 3'b111 : 3'b000;
endmodule
```

Is anything wrong here?

Another attempt at a Priority Circuit (same as before-diff. syntax)

```
module pri_dec (dout,y7,y6,y5,y4,y3,y2,y1);
   input y7,y6,y5,y4,y3,y2,y1;
   output [2:0] dout;
       assign dout = (y1==1'b1) ? 3'b001 : 3'b000,
              dout = (y2==1'b1) ? 3'b010 : 3'b000,
              dout = (y3==1'b1) ? 3'b011 : 3'b000,
              dout = (y4==1'b1) ? 3'b100 : 3'b000,
              dout = (y5==1'b1) ? 3'b101 : 3'b000,
              dout = (y6==1'b1) ? 3'b110 : 3'b000,
              dout = (y7==1'b1) ? 3'b111 : 3'b000;
endmodule
```

Is anything wrong here?

Comments on "bad" Priority Circuits

- Bad Attempts for Priority Circuit
- Problems in this Description
  - Multiple Concurrent Statements Driving **`dout`** Signal
  - Causes Multiple Gate Outputs to be Tied Together
  - Creates Unknown Logic Condition on Bus
- Writer seems to think that the order of the concurrent statements makes a difference
  - The order in which you arrange concurrent statements MAKES NO DIFFERENCE. The synthesized logic will be the same.
  - Ordering of statements only makes a difference within an **`always`**(**`initial`**) block. This is why statements within such blocks are called 'sequential' statements; the logic synthesized reflects the statement ordering (only for assignments to the same output).

Priority Circuit with Concurrent Statement

No procedural (sequential) block; just one concurrent statement.

```
module pri_dec (dout,y7,y6,y5,y4,y3,y2,y1);
   input y7,y6,y5,y4,y3,y2,y1;
   output [2:0] dout;
   assign dout = (y1==1'b1) ? 3'b001 :
          ((y2==1'b1) ? 3'b010 :
          ((y3==1'b1) ? 3'b011 :
          ((y4==1'b1) ? 3'b100 :
          ((y5==1'b1) ? 3'b101 :
          ((y6==1'b1) ? 3'b110 :
          ((y7==1'b1) ? 3'b111 : 3'b000))))));
endmodule
```

4-to-1 mux with 8 bit Datapaths

```
module mux8 (dout,A,B,C,D,SEL);
     input [7:0] A,B,C,D;
     input [1:0] SEL;
     output [7:0] dout;
     assign dout = (SEL==2'b00) ? A :
                   ((SEL==2'b01) ? B :
                   ((SEL==2'b10) ? C :
                   ((SEL==2'b11) ? D : 8'hxx)));
endmodule
```


Comments on MUX example

- This is one way to write a mux, but it is not the best way. The nested ternary assignment statement is actually a ***priority*** structure.
  - A ***mux has no priority*** between inputs, just a simple selection.
  - The synthesis tool has to work harder than necessary to understand that all possible choices for **`SEL`** are specified and that no priority is necessary.
- Just want a simple selection mechanism.

Multiplexers in Verilog

(a) Four-bit 4-to-1 mux using `if-else` chain

```
module mux4to1_4bit(s,a,b,c,d,y);
input  [1:0] s;
input  [3:0] a,b,c,d;
output [3:0] y;

reg [3:0] y;

always @*
 begin
  if (s == 2'b00) y = a;
  else if (s == 2'b01) y = b;
  else if (s == 2'b10) y = c;
  else y = d;
 end

endmodule
```

2-bit binary constant

equality operator

Priority Structure

(b) Four-bit 4-to-1 mux using a `case` statement

```
module mux4to1_4bit(s,a,b,c,d,y);
input  [1:0] s;
input  [3:0] a,b,c,d;
output [3:0] y;

reg [3:0] y;

always @*
 begin
  case (s)
   2'b00 :  y = a;
   2'b01 :  y = b;
   2'b10 :  y = c;
  default:  y = d;
  endcase
 end
endmodule
```

Non-Priority Structure

Another 4-to-1 Mux using **`always`** Block

```
module mux8 (dout,A,B,C,D,SEL);
   input [7:0] A, B, C, D;
   input [1:0] SEL;
   output [7:0] dout;
   reg [7:0]dout;
   always @(SEL or A or B or C or D)
      begin
            case (SEL)
                  2'b00: dout=A;
                  2'b01: dout=B;
                  2'b10: dout=C;
                  2'b11: dout=D;
                  default: dout=8'hxx;
            endcase
         end
endmodule
```

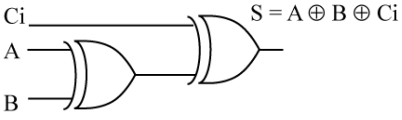
This is a concurrent always block; the sequential statement within the always block is the *case* statement.

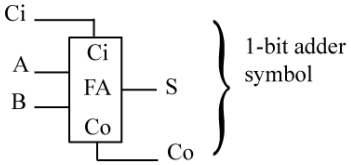
Full and Ripple Adders

A	B	Ci	S	Co
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

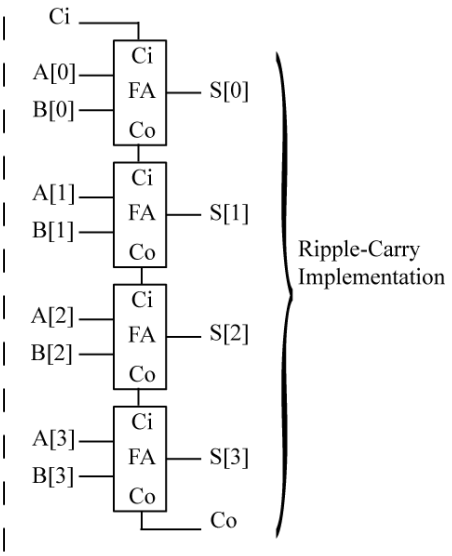
Truth Table for Full Adder
Ci - Carry In
S - Sum
Co - Carry Out

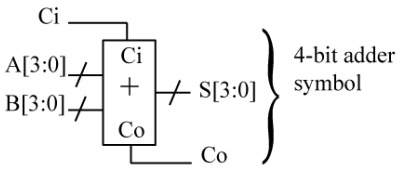
Co = Majority (A, B, Ci)
= (A ∧ B) ∨ (A ∧ Ci) ∨ (B ∧ Ci)





(a) Full Adder Truth table, Boolean equations, Symbol





(b) 4-bit adder symbol and Ripple Carry Implementation

Adders in Verilog

(a) Four-bit adder with no carry-in or carry-out

```
//4-bit adder
// no carry-in, carry-out
module add4bit ( a, b, s);

input [3:0] a,b;
output [3:0] s;

assign s = a + b;

endmodule
```

(b) Four-bit adder with carry-in, carry-out

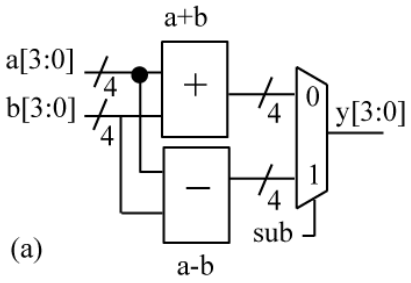
```
//4-bit adder with carry-in, carry-out
module add4bit (ci, a, b, s, co);

input ci;
input [3:0] a,b;
output [3:0] s;
output co;

wire [4:0] y;

//do 5-bit sum so that we
// have access to carry out
assign y ={1'b0,a} + {1'b0,b} + {4'b0,ci};
assign s = y[3:0]; //four-bit output
assign co = y[4]; //carry-out

endmodule
```

$$\begin{array}{r} 0a_3a_2a_1a_0 \\ +\,0b_3b_2b_1b_0 \\ +\,0\;0\;0\;0\;c_i \\ \hline y_4y_3y_2y_1y_0 \end{array}$$

{ } is the concatenation operator

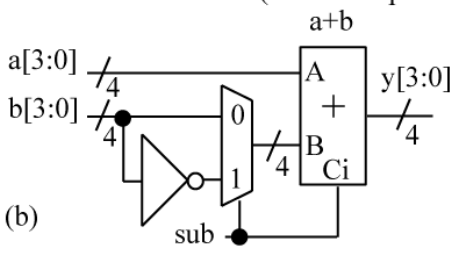
Verilog Synthesis Results

Even though these two implementations are functionally equivalent, some synthesis tools may produce a slightly more efficient implementation for (b) than (a).




```
//adder/subtractor
module addsub4bit(sub,a,b,y);

input sub;
input [3:0] a,b;
output [3:0] y;
//                sub=1  sub=0
assign y = sub ?(a-b):(a+b);

endmodule
```

a - b = a + ~b + 1 (two's complement)



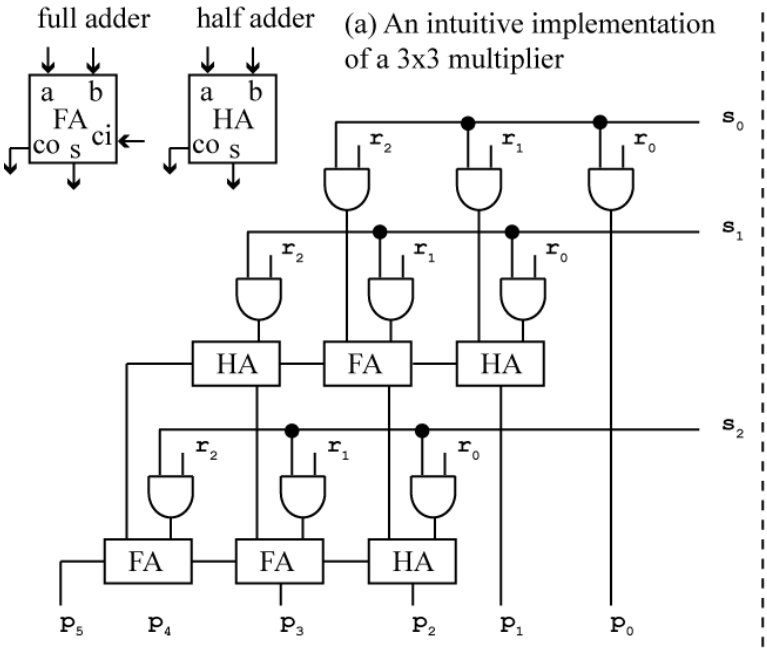

```
//adder/subtractor
module addsub4bit ( sub, a, b, y);

input sub;
input [3:0] a,b;
output [3:0] y;

wire [3:0] btmp;
//                  sub=1 sub=0
assign btmp = sub ? ~b  : b;
assign y = a + btmp + {3'b0,sub};

endmodule
```


Fixed-Point Multipliers

multiplicand		r_2	r_1	r_0	
multiplier	x	s_2	s_1	s_0	
partial product		s_0*r_2	s_0*r_1	s_0*r_0	
		s_1*r_2	s_1*r_1	s_1*r_0	
+		s_2*r_2	s_2*r_1	s_2*r_0	
	p_5	p_4	p_3	p_2	p_1	p_0	product

Binary	Decimal
1 1 1	7
x 1 0 1	x 5
1 1 1	35
0 0 0	
+ 1 1 1	
1 0 0 0 1 1 = 35	

Array Multiplier Structure



(b) A Verilog implementation of a 3x3 multiplication using the '*' operator

```
//3x3 multiplier
//using the '*' operator

module mult3x3 (a, b, y);

input [2:0] a,b;
output [5:0] y;

//do 3x3 multiply
assign y = a * b;

endmodule
```


Bit Shifting in Verilog

Concatenation Operation

a7 a6 a5 a4 a3 a2 a1 a0
si a7 a6 a5 a4 a3 a2 a1
y7 y6 y5 y4 y3 y2 y1 y0
} (a) right shift

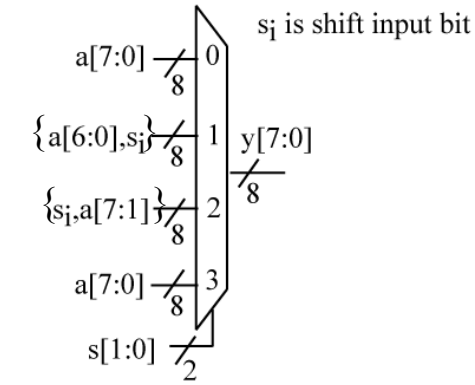
`y = {si,a[7:1]};` s_i is shift input bit

a7 a6 a5 a4 a3 a2 a1 a0
a6 a5 a4 a3 a2 a1 a0 si
y7 y6 y5 y4 y3 y2 y1 y0
} (b) left shift

`y = {a[6:0],si};` s_i is shift input bit

Bit Shifting with Multiplexers

(a) left/right shift with a multiplexer



s[0]= sleft (shift left)
s[1]= sright (shift right)

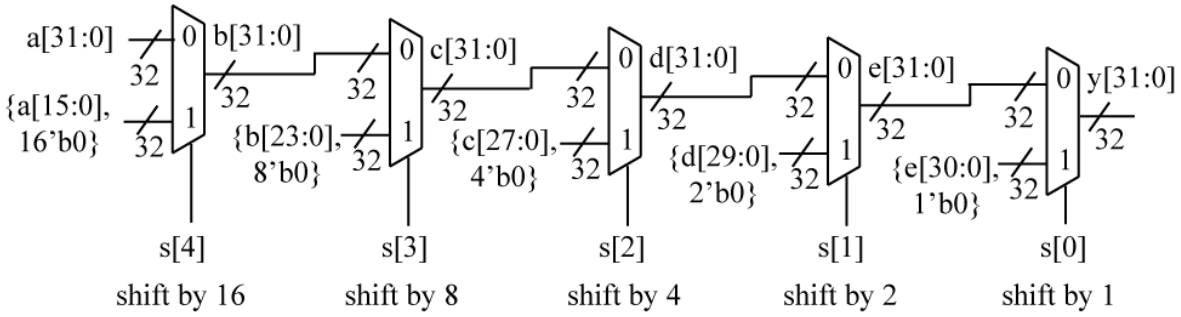
s	y
00	$y = a$
01	y is a shifted to left
10	y is a shifted to right
11	$y = a$

(b) Verilog left/right shift

```
module lrshift_8bit (si,sleft,sright,a,y);

input si,sleft,sright;
input [7:0] a;
output [7:0] y;

reg [7:0] y;
wire [1:0] s;

assign s[0] = sleft;
assign s[1] = sright;

always @(s or a) begin
 case (s)
  2'b01  : y = {a[6:0],si}; //left shift
  2'b10  : y = {si,a[7:1]}; //right shift
  default: y = a;
 endcase
end

endmodule
```


Multiplexer-based Barrel Shifter

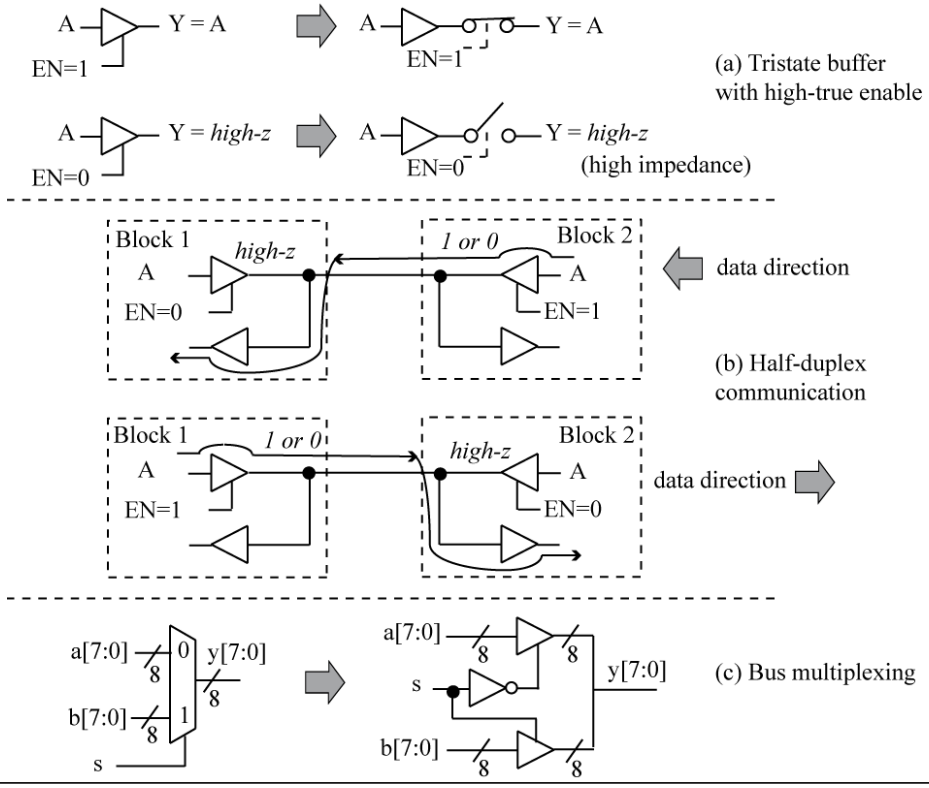
(a) Multiplexer implementation of a 32-bit left-shift barrel shifter



(b) Verilog implementation

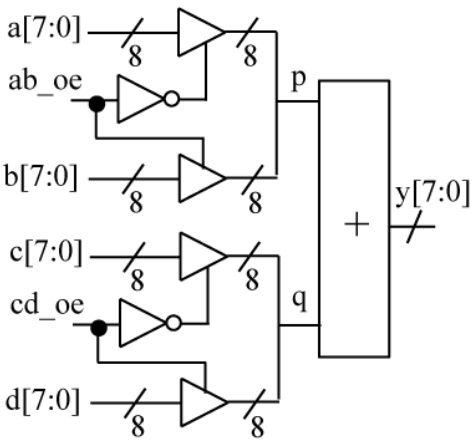
```
module bshift_32bit (s, a, y);
input [4:0] s;
input [31:0] a;
output [31:0] y;

assign y = a << s;

endmodule
```

Tri-state Buffers



Bus Multiplexing

(a) Bus multiplexing with tri-state drivers



(b) Verilog implementation

```
//tri-state buffer test
module tsb (a, b, c, d, ab_oe, cd_oe, y);

input ab_oe,cd_oe;
input [7:0] a,b,c,d;
output [7:0] y;

wire [7:0] p,q;

//              ab_oe=1 ab_oe=0
assign p = ~ab_oe ? a : 8'bzzzzzzzz;
assign p = ab_oe ? b : 8'bzzzzzzzz;
//              cd_oe=1 cd_oe=0
assign q = ~cd_oe ? c : 8'bzzzzzzzz;
assign q = cd_oe ? d : 8'bzzzzzzzz;

assign y = p + q;

endmodule
```

'z' is an allowed logic value and implies a tri-state driver for synthesis

Assignments to the same wire allowed because of tri-state drivers

Delay Modeling-Simulation

- Timescale Directive
  - `` `timescale 1ns/100ps ``
  - First number is unit of measurement for delays
  - Second number is precision (round-off increments)
- We will use Default Increments
  - # Directive for Delay Modeling
  - Timing Allows for More Accurate Modeling
  - Concept of Back Annotation
- Used ONLY FOR SIMULATION
  - Not Used for Synthesis
  - Back-annotation

Verilog Example with Delay

```
// Description of circuit with delay
module circuit_with_delay (A,B,C,x,y);
       input  A,B,C;
       output x,y;
       wire e;
       and  #(30) g1(e,A,B);
       not  #(10) g2(y,C);
       or   #(20) g3(x,e,y);
endmodule
```

Where does this come from?

	Time Units (ns)	Input ABC	Output y e x
Initial	-	000	1 0 1
Change	-	111	1 0 1
	10	111	0 0 1
	20	111	0 0 1
	30	111	0 1 0
	40	111	0 1 0
	50	111	0 1 1

Verilog Example with Delay

```
// Description of circuit with delay
module circuit_with_delay (A,B,C,x,y);
       input  A,B,C;
       output x,y;
       wire e;
       and  #(30) g1(e,A,B);
       not  #(10) g2(y,C);
       or   #(20) g3(x,e,y);
endmodule
```

t=0:

events: A:0/1 B:0/1 C:0/1

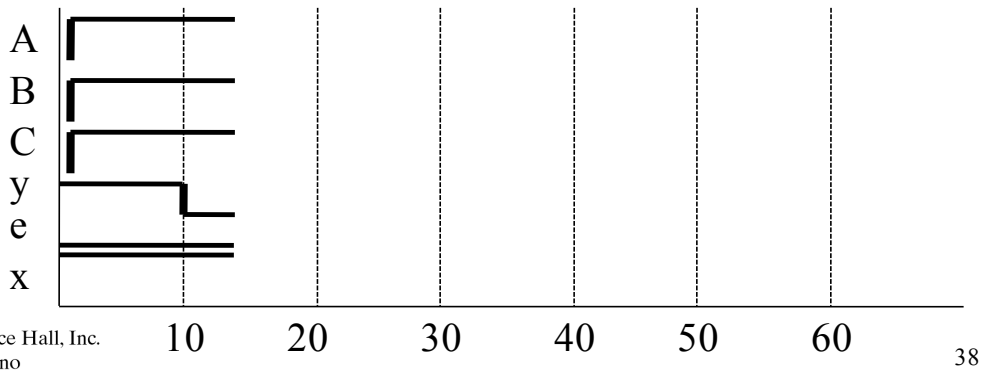
simulations: g1(AND) A=1,B=1,t=30(30)
g2(NOT) C=1,t=10(10)

Simulation Scheduled at 10

Simulation Scheduled at 30

Verilog Example (Waveform)

```
// Description of circuit with delay
module circuit_with_delay (A,B,C,x,y);
       input  A,B,C;
       output x,y;
       wire e;
       and  #(30) g1(e,A,B);
       not  #(10) g2(y,C);
       or   #(20) g3(x,e,y);
endmodule
```



Verilog Example with Delay

```
// Description of circuit with delay
module circuit_with_delay (A,B,C,x,y);
      input  A,B,C;
      output x,y;
      wire e;
      and  #(30) g1(e,A,B);
      not  #(10) g2(y,C);
      or   #(20) g3(x,e,y);
endmodule
```

t=10:

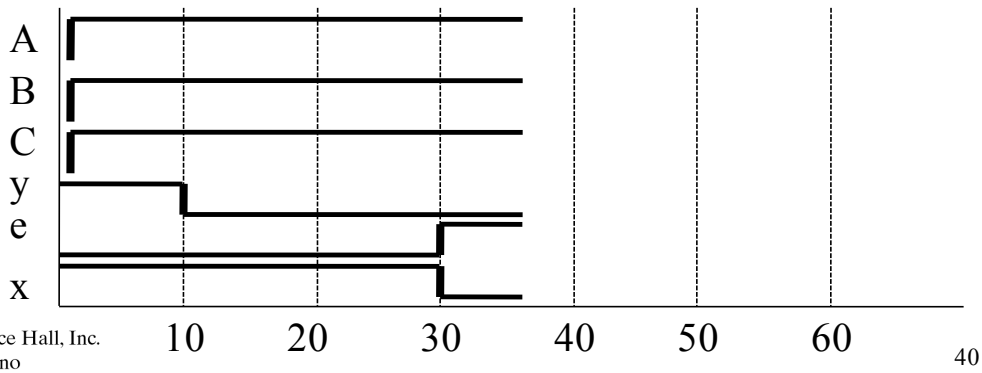
events: y:1/0

simulations: g3(OR) e=0,y=0,t=20(30)

Another Simulation Scheduled at 30

Verilog Example (Waveform)

```
// Description of circuit with delay
module circuit_with_delay (A,B,C,x,y);
      input  A,B,C;
      output x,y;
      wire e;
      and  #(30) g1(e,A,B);
      not  #(10) g2(y,C);
      or   #(20) g3(x,e,y);
endmodule
```



Verilog Example with Delay

```
// Description of circuit with delay
module circuit_with_delay (A,B,C,x,y);
      input  A,B,C;
      output x,y;
      wire e;
      and  #(30) g1(e,A,B);
      not  #(10) g2(y,C);
      or   #(20) g3(x,e,y);
endmodule
```

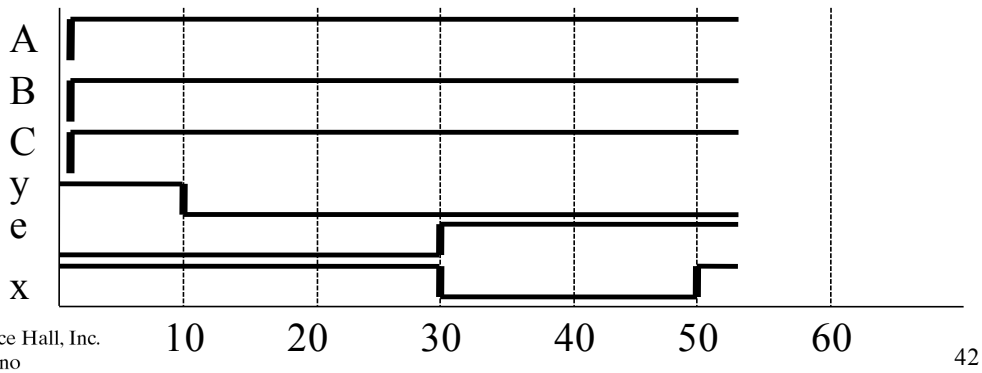
t=30:

events: e:0/1 x:1/0

simulations: g3(OR) e=1,y=0,t=20(50)

Verilog Example (Waveform)

```
// Description of circuit with delay
module circuit_with_delay (A,B,C,x,y);
      input  A,B,C;
      output x,y;
      wire e;
      and  #(30) g1(e,A,B);
      not  #(10) g2(y,C);
      or   #(20) g3(x,e,y);
endmodule
```



Verilog Example with Delay

```
// Description of circuit with delay
module circuit_with_delay (A,B,C,x,y);
      input  A,B,C;
      output x,y;
      wire e;
      and  #(30) g1(e,A,B);
      not  #(10) g2(y,C);
      or   #(20) g3(x,e,y);
endmodule
```

t=50:

events: x:0/1

simulations: NONE

Verilog Test Benches

- Only Used FOR SIMULATION
- Another Verilog Module that:
  - Provides Input
  - Allows Various Modules to be Interconnected
  - Contains Simulation Specific Commands
- Separation into Different Modules Important:
  - Can Synthesize Modules Directly
  - Can Change Abstraction of Modules
    - "Top-Level" Design
  - Allows Large Design Teams to Work Concurrently

Verilog Example with Testbench

```
// Stimulus for simple circuit
module stimcrct;
reg A, B, C;
wire x, y;
circuit_with_delay cwd(A, B, C, x, y);
initial
  begin
     A=1'b0; B=1'b0; C=1'b0;
    #100
     A=1'b1; B=1'b1; C=1'b1;
    #100    $finish;
  end
endmodule

// Description of circuit with delay
module circuit_with_delay (A,B,C,x,y);
        input  A,B,C;
        output x,y;
        wire e;
        and  #(30) g1(e,A,B);
        not  #(10) g2(y,C);
        or   #(20) g3(x,e,y);
endmodule
```


always and **initial** blocks

- Both Types of Blocks Contain Procedural (Sequential) Statements
- If More than One Procedural Statement use **begin – end**
- **Always** Block is Scheduled to Simulate/Execute at EVERY Simulation Time Cycle
- **Initial** Block is Scheduled to Simulate Only at the First Time Epoch

Verilog Simulation with Testbench

- Most Simulators Allow for Graphic Timing Diagrams to be Used
  - SynaptiCAD (software packaged with M. Mano textbook)
  - ModelTech (commercial simulator – free at www.model.com)
  - Cadence Verilog XL (commercial tool)
  - QuartusII (built-in timing simulator with Altera toolset)

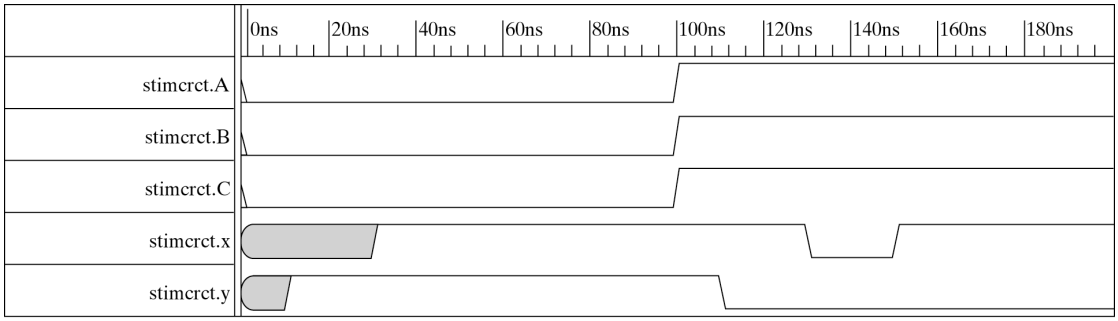


Fig. 3-38 Simulation Output of HDL Example 3-3

Model Abstractions in Verilog

- Netlist Level Models
  - Contain enough information to construct in lab
  - structural modeling
  - Commonly “Lowest” level of abstraction
  - Useful for Transfer Among CAD tools
- RTL (register transfer language) Level
  - Composed of Boolean Expressions and Registers
  - Can be Automatically Synthesized to a netlist
  - We will work mostly at this level
- Behavioral Level
  - High-level Constructs that only Describe Functionality
  - Automatic Behavioral Synthesis Tools do Exist