2

Combinational Logic Design Using Verilog HDL

2.1 Problems

2.1 Use dataflow modeling to implement the function shown below in a sum-of-products form and also in a product-of-sums form. Obtain the design module, the test bench module, and the outputs. Compare the outputs for both forms.

$$z_1(x_1, x_2, x_3, x_4) = \Sigma_m(0, 1, 6, 7, 11, 12, 13, 15))$$

$x_1x_2 \backslash x_3x_4$	0 0	0 1	1 1	1 0
0 0	1 ^{0}	1 ^{1}	0 ^{3}	0 ^{2}
0 1	0 ^{4}	0 ^{5}	1 ^{7}	1 ^{6}
1 1	1 ^{12}	1 ^{13}	1 ^{15}	0 ^{14}
1 0	0 ^{8}	0 ^{9}	1 ^{11}	0 ^{10}

z_1

$$z_1 = x_1'x_2'x_3' + x_1'x_2x_3 + x_1x_2x_3' + x_1x_3x_4$$

$$z_1(z_2) = (x_1 + x_2 + x_3')(x_1 + x_2' + x_3)(x_1' + x_3' + x_4)(x_1' + x_2 + x_3)$$

```
//dataflow to implement an sop and pos function

module log_eqtn_sop_pos (x1, x2, x3, x4, z1, z2);

//define inputs and outputs
input x1, x2, x3, x4;
output z1, z2;

//define internal nets
wire net1, net2, net3, net4,
      net5, net6, net7, net8;

//design output z1 for sum-of-products
assign   net1 = (~x1 & ~x2 & ~x3),
         net2 = (~x1 & x2 & x3),
         net3 = (x1 & x2 & ~x3),
         net4 = (x1 & x3 & x4),

         z1 = (net1 | net2 | net3 | net4);

//design output z2 for product-of-sums
assign   net5 = (x1 | x2 | ~x3),
         net6 = (x1 | ~x2 | x3),
         net7 = (~x1 | ~x3 | x4),
         net8 = (~x1 | x2 | x3),

         z2 = (net5 & net6 & net7 & net8);

endmodule
```

```
//test bench for sop and pos module
module log_eqtn_sop_pos_tb;

//inputs are reg for test bench
//outputs are wire for test bench
reg x1, x2, x3, x4;
wire z1, z2;

                     //continued on next page
```

```
//apply input vectors and display variables
initial
begin: apply_stimulus
   reg [4:0] invect;
   for (invect = 0; invect < 16; invect = invect + 1)
      begin
         {x1, x2, x3, x4} = invect [4:0];
         #10 $display ("x1 x2 x3 x4 = %b,
                        z1 = %b, z2 = %b",
                        {x1, x2, x3, x4}, z1, z2);
      end
end

//instantiate the module into the test bench
log_eqtn_sop_pos inst1 (x1, x2, x3, x4, z1, z2);

endmodule
```

```
x1 x2 x3 x4 = 0000, z1 = 1, z2 = 1
x1 x2 x3 x4 = 0001, z1 = 1, z2 = 1
x1 x2 x3 x4 = 0010, z1 = 0, z2 = 0
x1 x2 x3 x4 = 0011, z1 = 0, z2 = 0

x1 x2 x3 x4 = 0100, z1 = 0, z2 = 0
x1 x2 x3 x4 = 0101, z1 = 0, z2 = 0
x1 x2 x3 x4 = 0110, z1 = 1, z2 = 1
x1 x2 x3 x4 = 0111, z1 = 1, z2 = 1

x1 x2 x3 x4 = 1000, z1 = 0, z2 = 0
x1 x2 x3 x4 = 1001, z1 = 0, z2 = 0
x1 x2 x3 x4 = 1010, z1 = 0, z2 = 0
x1 x2 x3 x4 = 1011, z1 = 1, z2 = 1

x1 x2 x3 x4 = 1100, z1 = 1, z2 = 1
x1 x2 x3 x4 = 1101, z1 = 1, z2 = 1
x1 x2 x3 x4 = 1110, z1 = 0, z2 = 0
x1 x2 x3 x4 = 1111, z1 = 1, z2 = 1
```

2.2 A Karnaugh map is shown below using x_5 as a map-entered variable. Obtain the input equations for a nonlinear-select multiplexer using $x_1x_2 = s_1s_0$. A nonlinear-select multiplexer is a smaller multiplexer with fewer data inputs and can be effectively utilized with a corresponding reduction in machine cost. Use dataflow modeling with the **assign** statement and behavioral modeling with the **case** statement to implement the design module. Provide several combinations of the five variables $x_1x_2x_3x_4x_5$ in the test bench. Obtain the outputs and verify that they conform to the minterm entries of the Karnaugh map.

x_1x_2 \ x_3x_4	0 0	0 1	1 1	1 0
0 0	^{0} 0	^{1} x_5	^{3} 0	^{2} x_5
0 1	^{4} 0	^{5} 0	^{7} 1	^{6} 1
1 1	^{12} 0	^{13} 1	^{15} 1	^{14} x_5'
1 0	^{8} 1	^{9} 1	^{11} 0	^{10} 0

z_1

$s_1s_0 =$ 00: Data input 0 = $x_3'x_4x_5 + x_3x_4'x_5 = (x_3 \oplus x_4)x_5$

$s_1s_0 =$ 01: Data input 1 = x_3

$s_1s_0 =$ 10: Data input 2 = x_3'

$s_1s_0 =$ 11: Data input 3 = $x_4 + x_3x_5'$

```
//dataflow/behavioral nonlinear multiplexer
//using the assign and case statements

module mux_nonlinear_bh_df (sel, data, x1, x2, x3, x4,
                                x5, z1);

//define inputs and output
input [1:0] sel;
input [3:0] data;
input x1, x2, x3, x4, x5;
output z1;
                              //continued on next page
```

```
//variables are reg in always
reg z1;

assign   sel[0] = x2,
         sel[1] = x1,

         data[0] = (x3 ^ x4) & x5,
         data[1] = x3,
         data[2] = ~x3,
         data[3] = x4 | (x3 & ~x5);

always @ (sel or data)
begin
   case (sel)
      (0) : z1 = data[0];
      (1) : z1 = data[1];
      (2) : z1 = data[2];
      (3) : z1 = data[3];
      default : z1 = 1'b0;
   endcase
end

endmodule
```

```
//test bench for the nonlinear select multiplexer

module mux_nonlinear_bh_df_tb;

//inputs are reg for test bench
//outputs are wire for test bench
reg [1:0] sel;
reg [3:0] data;
reg x1, x2, x3, x4, x5;
wire z1;

//display variables
initial
$monitor ("sel = %b, x3 x4 x5 = %b, z1 = %b",
            sel, {x3, x4, x5}, z1);

                           //continued on next page
```

```
//apply input vectors
initial
begin
   #0    sel = 2'b00;   {x3, x4, x5} = 3'b000;
   #10   sel = 2'b01;   {x3, x4, x5} = 3'b001;
   #10   sel = 2'b10;   {x3, x4, x5} = 3'b010;
   #10   sel = 2'b11;   {x3, x4, x5} = 3'b011;

   #10   sel = 2'b00;   {x3, x4, x5} = 3'b011;
   #10   sel = 2'b01;   {x3, x4, x5} = 3'b101;
   #10   sel = 2'b10;   {x3, x4, x5} = 3'b111;
   #10   sel = 2'b11;   {x3, x4, x5} = 3'b101;

   #10   sel = 2'b00;   {x3, x4, x5} = 3'b101;
   #10   sel = 2'b01;   {x3, x4, x5} = 3'b000;
   #10   sel = 2'b10;   {x3, x4, x5} = 3'b111;
   #10   sel = 2'b11;   {x3, x4, x5} = 3'b100;

   #10   sel = 2'b00;   {x3, x4, x5} = 3'b100;
   #10   sel = 2'b01;   {x3, x4, x5} = 3'b011;
   #10   sel = 2'b10;   {x3, x4, x5} = 3'b000;
   #10   sel = 2'b11;   {x3, x4, x5} = 3'b101;

   #10   $stop;
end

//instantiate the module into the test bench
mux_nonlinear_bh_df inst1 (sel, data, x1, x2,
                           x3, x4, x5, z1);

endmodule
```

```
sel = 00, x3 x4 x5 = 000, z1 = 0
sel = 01, x3 x4 x5 = 001, z1 = 0
sel = 10, x3 x4 x5 = 010, z1 = 1
sel = 11, x3 x4 x5 = 011, z1 = 1

sel = 00, x3 x4 x5 = 011, z1 = 1
sel = 01, x3 x4 x5 = 101, z1 = 1
sel = 10, x3 x4 x5 = 111, z1 = 0
sel = 11, x3 x4 x5 = 101, z1 = 0

                                   //continued on next page
```

```
sel = 00, x3 x4 x5 = 101, z1 = 1
sel = 01, x3 x4 x5 = 000, z1 = 0
sel = 10, x3 x4 x5 = 111, z1 = 0
sel = 11, x3 x4 x5 = 100, z1 = 1

sel = 00, x3 x4 x5 = 100, z1 = 0
sel = 01, x3 x4 x5 = 011, z1 = 0
sel = 10, x3 x4 x5 = 000, z1 = 1
sel = 11, x3 x4 x5 = 101, z1 = 0
```

2.3 Design a structural module that will generate a high output z_1 if a 4-bit binary number $x_1x_2x_3x_4$ has a value less than or equal to four or greater than eleven. Generate a Karnaugh map and obtain the equation for z_1 in a sum-of-products form and for z_2 in a product-of-sums form. Instantiate dataflow modules for the logic gates into the structural module. Obtain the design module, the test bench module for all combinations of the inputs, and the outputs.

x_1x_2 \ x_3x_4	0 0	0 1	1 1	1 0
0 0	$\overset{0}{1}$	$\overset{1}{1}$	$\overset{3}{1}$	$\overset{2}{1}$
0 1	$\overset{4}{1}$	$\overset{5}{0}$	$\overset{7}{0}$	$\overset{6}{0}$
1 1	$\overset{12}{1}$	$\overset{13}{1}$	$\overset{15}{1}$	$\overset{14}{1}$
1 0	$\overset{8}{0}$	$\overset{9}{0}$	$\overset{11}{0}$	$\overset{10}{0}$

z_1 (z_2)

$$\begin{aligned} z_1 &= x_1'x_2' + x_1x_2 + x_2x_3'x_4' \\ &= (x_1 \oplus x_2)' + x_2x_3'x_4' \end{aligned}$$

$$z_2 = (x_1 + x_2' + x_4')\,(x_1 + x_2' + x_3')\,(x_1' + x_2)$$

```
//structural dataflow number <=4 or >11
module number_range5 (x1, x2, x3, x4, z1, z2);

input x1, x2, x3, x4;    //define inputs and output
output z1, z2;

//define internal nets
wire  net1, net2, net3, net4, net5;

//design the logic for the sum-of-products z1
xnor2_df    inst1 (x1, x2, net1);
and3_df     inst2 (x2, ~x3, ~x4, net2);
or2_df      inst3 (net1, net2, z1);

//design the logic for the product-of-sums z2
or3_df      inst4 (x1, ~x2, ~x4, net3),
            inst5 (x1, ~x2, ~x3, net4);
or2_df      inst6 (~x1, x2, net5);
and3_df     imst7 (net3, net4, net5, z2);
endmodule
```

```
//test bench for number <=4 or >11
module number_range5_tb;

reg x1, x2, x3, x4;  //inputs are reg for test bench
wire z1, z2;         //outputs are wire

//apply input vectors and display variables
initial
begin: apply_stimulus
reg [4:0] invect;
for (invect = 0; invect < 16; invect = invect + 1)
   begin
      {x1, x2, x3, x4} = invect [4:0];
      #10 $display ("x1 x2 x3 x4) = %b, z1 =%b, z2 =%b",
                  {x1, x2, x3, x4}, z1, z2);
   end
end

//instantiate the module into the test bench
number_range5 inst1 (x1, x2, x3, x4, z1, z2);

endmodule
```

```
x1 x2 x3 x4) = 0000, z1 =1, z2 =1
x1 x2 x3 x4) = 0001, z1 =1, z2 =1
x1 x2 x3 x4) = 0010, z1 =1, z2 =1
x1 x2 x3 x4) = 0011, z1 =1, z2 =1

x1 x2 x3 x4) = 0100, z1 =1, z2 =1
x1 x2 x3 x4) = 0101, z1 =0, z2 =0
x1 x2 x3 x4) = 0110, z1 =0, z2 =0
x1 x2 x3 x4) = 0111, z1 =0, z2 =0

x1 x2 x3 x4) = 1000, z1 =0, z2 =0
x1 x2 x3 x4) = 1001, z1 =0, z2 =0
x1 x2 x3 x4) = 1010, z1 =0, z2 =0
x1 x2 x3 x4) = 1011, z1 =0, z2 =0

x1 x2 x3 x4) = 1100, z1 =1, z2 =1
x1 x2 x3 x4) = 1101, z1 =1, z2 =1
x1 x2 x3 x4) = 1110, z1 =1, z2 =1
x1 x2 x3 x4) = 1111, z1 =1, z2 =1
```

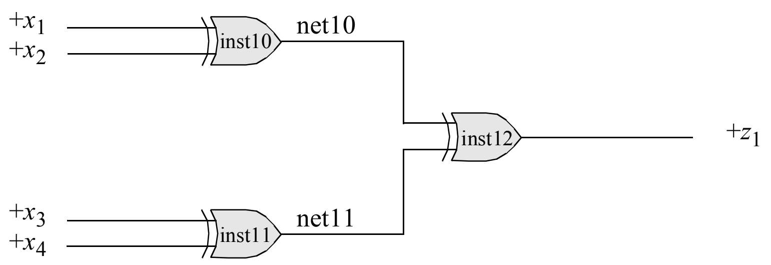
2.4 Obtain the Karnaugh map that represents the equation shown below. Then obtain the design module using built-in primitives, the test bench module, and the outputs for the equation shown below. In the same design module obtain the Verilog code for an equivalent equation using only exclusive-OR gates with logic gates that were design using dataflow modeling.

$$\begin{aligned} z_1 = {} & x_1'x_2'x_3'x_4 + x_1'x_2'x_3x_4' + x_1'x_2x_3'x_4' \\ & + x_1'x_2x_3x_4 + x_1x_2x_3'x_4 + x_1x_2x_3x_4' \\ & + x_1x_2'x_3'x_4' + x_1x_2'x_3x_4 \end{aligned}$$

$x_1x_2 \backslash x_3x_4$	0 0	0 1	1 1	1 0
0 0	^{0} 0	^{1} 1	^{3} 0	^{2} 1
0 1	^{4} 1	^{5} 0	^{7} 1	^{6} 0
1 1	^{12} 0	^{13} 1	^{15} 0	^{14} 1
1 0	^{8} 1	^{9} 0	^{11} 1	^{10} 0

z_1

$$\begin{aligned}
z_1 &= x_1'x_2'(x_3'x_4 + x_3x_4') + x_1'x_2(x_3'x_4' + x_3x_4) \\
&\quad + x_1x_2(x_3'x_4 + x_3x_4') + x_1x_2'(x_3'x_4' + x_3x_4) \\
&= x_1'x_2'(x_3 \oplus x_4) + x_1'x_2(x_3 \oplus x_4)' \\
&\quad + x_1x_2(x_3 \oplus x_4) + x_1x_2'(x_3 \oplus x_4)' \\
&= (x_1 \oplus x_2)'(x_3 \oplus x_4) + (x_1 \oplus x_2)(x_3 \oplus x_4)' \\
&= x_1 \oplus x_2 \oplus x_3 \oplus x_4
\end{aligned}$$



```
//synthesize equation using exclusive-or
module log_eqtn_xor (x1, x2, x3, x4, z1, z2);

input x1, x2, x3, x4;   //define inputs and output
output z1, z2;

//design the logic using built-in primitives
and   inst1 (net1, ~x1, ~x2, ~x3, x4),
      inst2 (net2, ~x1, ~x2, x3, ~x4),
      inst3 (net3, ~x1, x2, ~x3, ~x4),
      inst4 (net4, ~x1, x2, x3, x4),
      inst5 (net5, x1, x2, ~x3, x4),
      inst6 (net6, x1, x2, x3, ~x4),
      inst7 (net7, x1, ~x2, ~x3, ~x4),
      inst8 (net8, x1, ~x2, x3, x4);

or    inst9 (z1, net1, net2, net3, net4,
                net5, net6, net7, net8);

//design the logic using dataflow gates
xor2_df  inst10 (x1, x2, net10),
         inst11 (x3, x4, net11),
         inst12 (net10, net11, z2);

endmodule
```

```
//test bench for log_eqtn_xor
module log_eqtn_xor_tb;

//inputs are reg for test bench
//outputs are wire for test bench
reg x1, x2, x3, x4;
wire z1, z2;

initial     //apply input vectors
begin: apply_stimulus
   reg [4:0] invect;
   for (invect = 0; invect < 16; invect = invect + 1)
      begin
         {x1, x2, x3, x4} = invect [4:0];
         #10 $display ("{x1 x2 x3 x4} = %b,
                        z1 =%b, z2 = %b",
                        {x1, x2, x3, x4}, z1, z2);
         end
end

//instantiate the module into the test bench
log_eqtn_xor inst1 (x1, x2, x3, x4, z1, z2);

endmodule
```

```
{x1 x2 x3 x4} = 0000, z1 = 0, z2 = 0
{x1 x2 x3 x4} = 0001, z1 = 1, z2 = 1
{x1 x2 x3 x4} = 0010, z1 = 1, z2 = 1
{x1 x2 x3 x4} = 0011, z1 = 0, z2 = 0

{x1 x2 x3 x4} = 0100, z1 = 1, z2 = 1
{x1 x2 x3 x4} = 0101, z1 = 0, z2 = 0
{x1 x2 x3 x4} = 0110, z1 = 0, z2 = 0
{x1 x2 x3 x4} = 0111, z1 = 1, z2 = 1

{x1 x2 x3 x4} = 1000, z1 = 1, z2 = 1
{x1 x2 x3 x4} = 1001, z1 = 0, z2 = 0
{x1 x2 x3 x4} = 1010, z1 = 0, z2 = 0
{x1 x2 x3 x4} = 1011, z1 = 1, z2 = 1

{x1 x2 x3 x4} = 1100, z1 = 0, z2 = 0
{x1 x2 x3 x4} = 1101, z1 = 1, z2 = 1
{x1 x2 x3 x4} = 1110, z1 = 1, z2 = 1
{x1 x2 x3 x4} = 1111, z1 = 0, z2 = 0
```

2.5 Design an octal-to-binary code converter using logic gates that were designed using dataflow modeling. The octal-to-binary conversion table is shown below. Obtain the conversion equations, then design the dataflow module, the test bench module, and obtain the outputs.

Octal Inputs								Binary Outputs		
o[0]	*o[1]*	*o[2]*	*o[3]*	*o[4]*	*o[5]*	*o[6]*	*o[7]*	*b[2]*	*b[1]*	*b[0]*
1	0	0	0	0	0	0	0	0	0	0
0	**1**	0	0	0	0	0	0	0	0	1
0	0	**1**	0	0	0	0	0	0	1	0
0	0	0	**1**	0	0	0	0	0	1	1
0	0	0	0	**1**	0	0	0	1	0	0
0	0	0	0	0	**1**	0	0	1	0	1
0	0	0	0	0	0	**1**	0	1	1	0
0	0	0	0	0	0	0	**1**	1	1	1

$$bin[0] = oct[1] + oct[3] + 0ct[5] \mid oct[7]$$

$$bin[1] = oct[2] + oct[3] + oct[6] + oct[7]$$

$$bin[2] = oct[4] + oct[5] + oct[6] + oct[7]$$

```
//dataflow octal-to-binary code conversion

module octal_to_binary (oct, bin);

//define inputs and outputs
input [0:7] oct;
output [2:0] bin;

//design the logic for octal-to-binary using dataflow
or4_df   inst1 (oct[1], oct[3], oct[5], oct[7],
                  bin[0]),
         inst2 (oct[2], oct[3], oct[6], oct[7],
                  bin[1]),
         inst3 (oct[4], oct[5], oct[6], oct[7],
                  bin[2]);

endmodule
```

```
//test bench for octal-to-binary conversion

module octal_to_binary_tb;

//inputs are reg for test bench
//outputs are wire for test bench
reg [0:7] oct;
wire [2:0] bin;

initial     //display variables
$monitor ("octal = %b, binary = %b",
            oct [0:7], bin [2:0]);

//apply input vectors
initial
begin
   #0    oct [0:7] = 8'b1000_0000;
   #10   oct [0:7] = 8'b0100_0000;
   #10   oct [0:7] = 8'b0010_0000;
   #10   oct [0:7] = 8'b0001_0000;

   #10   oct [0:7] = 8'b0000_1000;
   #10   oct [0:7] = 8'b0000_0100;
   #10   oct [0:7] = 8'b0000_0010;
   #10   oct [0:7] = 8'b0000_0001;

   #10   $stop;
end

//instantiate the module into the test bench
octal_to_binary inst1 (oct, bin);

endmodule
```

```
      Octal [0:7} Binary [2:0]

octal = 10000000, binary = 000
octal = 01000000, binary = 001
octal = 00100000, binary = 010
octal = 00010000, binary = 011

octal = 00001000, binary = 100
octal = 00000100, binary = 101
octal = 00000010, binary = 110
octal = 00000001, binary = 111
```

2.6 Repeat Problem 2.5 using behavioral modeling with the **case** statement. Obtain the design module, the test bench module, and the outputs.

```
//behavioral octal-to-binary conversion
module octal_to_binary_case (oct, bin);

//define inputs and outputs
input [0:7] oct;
output [2:0] bin;

//variables in always are declared as reg
reg [2:0] bin;

//determine binary code
always @ (oct)
begin
   case (oct)
      8'b1000_0000 : bin = 3'b000;
      8'b0100_0000 : bin = 3'b001;
      8'b0010_0000 : bin = 3'b010;
      8'b0001_0000 : bin = 3'b011;

      8'b0000_1000 : bin = 3'b100;
      8'b0000_0100 : bin = 3'b101;
      8'b0000_0010 : bin = 3'b110;
      8'b0000_0001 : bin = 3'b111;

      default : bin = 3'b000;
   endcase
end
endmodule
```

```
//test bench for the octal-to-binary using case
module octal_to_binary_case_tb;

//inputs are reg for test bench
//outputs are wire for test bench
reg [0:7] oct;
wire [2:0] bin;

//display variables
initial
$monitor ("octal = %b, binary = %b",
            oct [0:7], bin [2:0]);
                               //continued on next page
```

```
//apply input vectors
initial
begin
   #0    oct [0:7] = 8'b1000_0000;
   #10   oct [0:7] = 8'b0100_0000;
   #10   oct [0:7] = 8'b0010_0000;
   #10   oct [0:7] = 8'b0001_0000;


   #10   oct [0:7] = 8'b0000_1000;
   #10   oct [0:7] = 8'b0000_0100;
   #10   oct [0:7] = 8'b0000_0010;
   #10   oct [0:7] = 8'b0000_0001;

   #10   $stop;
end

//instantiate the module into the test bench
octal_to_binary_case inst1 (oct, bin);

endmodule
```

```
       Octal [0:7} Binary [2:0]

octal = 10000000, binary = 000
octal = 01000000, binary = 001
octal = 00100000, binary = 010
octal = 00010000, binary = 011


octal = 00001000, binary = 100
octal = 00000100, binary = 101
octal = 00000010, binary = 110
octal = 00000001, binary = 111
```

2.7 Design a 5-input majority circuit using the dataflow continuous **assign** statement. Obtain the Karnaugh map and the equations for the sum-of-products and for the product-of-sums expressions. Obtain the design module for both the sum-of-products expression and the product-of-sums expression. Obtain the test bench module and the outputs.

$x_5 = 0$

x_1x_2 \ x_3x_4	0 0	0 1	1 1	1 0
0 0	^{0} 0	^{2} 0	^{6} 0	^{4} 0
0 1	^{8} 0	^{10} 0	^{14} 1	^{12} 0
1 1	^{24} 0	^{26} 1	^{30} 1	^{28} 1
1 0	^{16} 0	^{18} 0	^{22} 1	^{20} 0

$x_5 = 1$

x_1x_2 \ x_3x_4	0 0	0 1	1 1	1 0
0 0	^{1} 0	^{3} 0	^{7} 1	^{5} 0
0 1	^{9} 0	^{11} 1	^{15} 1	^{13} 1
1 1	^{25} 1	^{27} 1	^{31} 1	^{29} 1
1 0	^{17} 0	^{19} 1	^{23} 1	^{21} 1

$z_1(z_2)$

$$z_1 = x_1x_3x_4 + x_3x_4x_5 + x_1x_2x_5 + x_2x_4x_5 + x_2x_3x_5$$
$$+ x_1x_3x_5 + x_1x_4x_5 + x_1x_2x_4 + x_1x_2x_3 + x_1x_3x_4$$

$$z_2 = (x_3 + x_4 + x_5)\,(x_1 + x_2 + x_5)\,(x_2 + x_3 + x_4)$$
$$(x_1 + x_3 + x_5)\,(x_1 + x_4 + x_5)\,(x_2 + x_4 + x_5)$$
$$(x_1 + x_2 + x_3\ (x_1 + x_3 + x_4)\,(x_1 + x_2 + x_4)$$
$$(x_2 + x_3 + x_5)$$

```
//dataflow for 5-input majority circuit

module majority_sop_pos (x1, x2, x3, x4, x5, z1, z2);

//define inputs and outputs
input x1, x2, x3, x4, x5;
output z1, z2;

//design the sum-of-products logic using assign
assign z1 = (x2 & x3 & x4) | (x3 & x4 & x5)
            | (x1 & x2 & x5) | (x2 & x4 & x5)
            | (x2 & x3 & x5) | (x1 & x3 & x5)
            | (x1 & x4 & x5) | (x1 & x2 & x4)
            | (x1 & x2 & x3) | (x1 & x3 & x4);

                           //continued on next page
```

```
//design the product-of-sums logic using assign
assign z2 = (x3 | x4 | x5) & (x1 | x2 | x5)
            & (x2 | x3 | x4) & (x1 | x3 | x5)
            & (x1 | x4 | x5) & (x2 | x4 | x5)
            & (x1 | x2 | x3) & (x1 | x3 | x4)
            & (x1 | x2 | x4) & (x2 | x3 | x5);

endmodule
```

```
//test bench for the 5-input majority circuit

module majority_sop_pos_tb;

//inputs are reg for test bench
//outputs are wire for test bench
reg x1, x2, x3, x4, x5;
wire z1, z2;


//apply input vectors
initial
begin: apply_stimulus
   reg [5:0] invect;
   for (invect = 0; invect < 32; invect = invect + 1)
      begin
         {x1, x2, x3, x4, x5} = invect [5:0];
         #10 $display ("{x1 x32 x3 x4 x5} = %b,
                        z1 = %b, z2 = %b",
                  {x1, x2, x3, x4, x5}, z1, z2);
      end
end


//instantiate the module into the test bench
majority_sop_pos inst1 (x1, x2, x3, x4, x5, z1, z2);

endmodule
```

```
{x1 x32 x3 x4 x5} = 00000, z1 = 0, z2 = 0
{x1 x32 x3 x4 x5} = 00001, z1 = 0, z2 = 0
{x1 x32 x3 x4 x5} = 00010, z1 = 0, z2 = 0
{x1 x32 x3 x4 x5} = 00011, z1 = 0, z2 = 0

{x1 x32 x3 x4 x5} = 00100, z1 = 0, z2 = 0
{x1 x32 x3 x4 x5} = 00101, z1 = 0, z2 = 0
{x1 x32 x3 x4 x5} = 00110, z1 = 0, z2 = 0
{x1 x32 x3 x4 x5} = 00111, z1 = 1, z2 = 1

{x1 x32 x3 x4 x5} = 01000, z1 = 0, z2 = 0
{x1 x32 x3 x4 x5} = 01001, z1 = 0, z2 = 0
{x1 x32 x3 x4 x5} = 01010, z1 = 0, z2 = 0
{x1 x32 x3 x4 x5} = 01011, z1 = 1, z2 = 1

{x1 x32 x3 x4 x5} = 01100, z1 = 0, z2 = 0
{x1 x32 x3 x4 x5} = 01101, z1 = 1, z2 = 1
{x1 x32 x3 x4 x5} = 01110, z1 = 1, z2 = 1
{x1 x32 x3 x4 x5} = 01111, z1 = 1, z2 = 1

{x1 x32 x3 x4 x5} = 10000, z1 = 0, z2 = 0
{x1 x32 x3 x4 x5} = 10001, z1 = 0, z2 = 0
{x1 x32 x3 x4 x5} = 10010, z1 = 0, z2 = 0
{x1 x32 x3 x4 x5} = 10011, z1 = 1, z2 = 1

{x1 x32 x3 x4 x5} = 10100, z1 = 0, z2 = 0
{x1 x32 x3 x4 x5} = 10101, z1 = 1, z2 = 1
{x1 x32 x3 x4 x5} = 10110, z1 = 1, z2 = 1
{x1 x32 x3 x4 x5} = 10111, z1 = 1, z2 = 1

{x1 x32 x3 x4 x5} = 11000, z1 = 0, z2 = 0
{x1 x32 x3 x4 x5} = 11001, z1 = 1, z2 = 1
{x1 x32 x3 x4 x5} = 11010, z1 = 1, z2 = 1
{x1 x32 x3 x4 x5} = 11011, z1 = 1, z2 = 1

{x1 x32 x3 x4 x5} = 11100, z1 = 1, z2 = 1
{x1 x32 x3 x4 x5} = 11101, z1 = 1, z2 = 1
{x1 x32 x3 x4 x5} = 11110, z1 = 1, z2 = 1
{x1 x32 x3 x4 x5} = 11111, z1 = 1, z2 = 1
```

2.8 Given the Karnaugh map shown below, obtain the equation for output z_1 in a sum-of-products form and for output z_2 in product-of-sums form. Then obtain the design module using logic gates that were designed using dataflow modeling. Obtain the test bench module for all combinations of the five inputs. Obtain the output values for z_1 and z_2.

$x_5 = 0$

x_1x_2 \ x_3x_4	0 0	0 1	1 1	1 0
0 0	${}^{0}\ 0$	${}^{2}\ 1$	${}^{6}\ 1$	${}^{4}\ 0$
0 1	${}^{8}\ 1$	${}^{10}\ 1$	${}^{14}\ 1$	${}^{12}\ 1$
1 1	${}^{24}\ 0$	${}^{26}\ 1$	${}^{30}\ 0$	${}^{28}\ 0$
1 0	${}^{16}\ 0$	${}^{18}\ 1$	${}^{22}\ 1$	${}^{20}\ 1$

$x_5 = 1$

x_1x_2 \ x_3x_4	0 0	0 1	1 1	1 0
0 0	${}^{1}\ 0$	${}^{3}\ 1$	${}^{7}\ 1$	${}^{5}\ 0$
0 1	${}^{9}\ 1$	${}^{11}\ 1$	${}^{15}\ 1$	${}^{13}\ 1$
1 1	${}^{25}\ 1$	${}^{27}\ 1$	${}^{31}\ 0$	${}^{29}\ 0$
1 0	${}^{17}\ 1$	${}^{19}\ 1$	${}^{23}\ 1$	${}^{21}\ 0$

$z_1(z_2)$

$$z_1 = x_1'x_2 + x_3'x_4 + x_1'x_4 + x_1x_2'x_4 + x_1x_2'x_3x_5' + x_1x_3'x_5$$

$$z_2 = (x_1' + x_2' + x_3')(x_1 + x_2 + x_4)(x_1' + x_3 + x_4 + x_5)$$
$$(x_1' + x_3' + x_4 + x_5')$$

```
//dataflow sop and pos using dataflow logic gates
module dataflow_sop_pos (x1, x2, x3, x4, x5, z1, z2);

input x1, x2, x3, x4, x5;  //define inputs and outputs
output z1, z2;

//design the logic for the sum-of-products for z1
and2_df   inst1 (~x1, x2, net1),
          inst2 (~x3, x4, net2),
          inst3 (~x1, x4, net3);

and3_df   inst4 (x1, ~x2, x4, net4),
          inst5 (x1, ~x3, x5, net5);

and4_df   inst6 (x1, ~x2, x3, ~x5, net6);

or6_df    inst7 (net1, net2, net3, net4, net5, net6, z1);
                               //continued on next page
```

```
//design the logic for the product-of-sums for z2
or3_df   inst8 (~x1, ~x2, ~x3, net8),
         inst9 (x1, x2, x4, net9);

or4_df   inst10 (~x1, x3, x4, x5, net10),
         inst11 (~x1, ~x3, x4, ~x5, net11);

and4_df  inst12 (net8, net9, net10, net11, z2);

endmodule
```

```
//test bench for dataflow sop and pos

module dataflow_sop_pos_tb;

//inputs are reg for test bench
//outputs are wire for test bench
reg x1, x2, x3, x4, x5;
wire z1, z2;

//apply input vectors and display variables
initial
begin: apply_stimulus
   reg [5:0] invect;
   for (invect = 0; invect < 32; invect = invect + 1)
      begin
         {x1, x2, x3, x4, x5} = invect [5:0];
         #10 $display ("x1 x2 x3 x4 x5) = %b,
                        z1 = %b, z2 = %b",
                        {x1, x2, x3, x4, x5}, z1, z2);
      end
end

//instantiate the module into the test bench
dataflow_sop_pos inst1 (x1, x2, x3, x4, x5, z1, z2);

endmodule
```

```
x1 x2 x3 x4 x5) = 00000, z1 = 0, z2 = 0
x1 x2 x3 x4 x5) = 00001, z1 = 0, z2 = 0
x1 x2 x3 x4 x5) = 00010, z1 = 1, z2 = 1
x1 x2 x3 x4 x5) = 00011, z1 = 1, z2 = 1

x1 x2 x3 x4 x5) = 00100, z1 = 0, z2 = 0
x1 x2 x3 x4 x5) = 00101, z1 = 0, z2 = 0
x1 x2 x3 x4 x5) = 00110, z1 = 1, z2 = 1
x1 x2 x3 x4 x5) = 00111, z1 = 1, z2 = 1

x1 x2 x3 x4 x5) = 01000, z1 = 1, z2 = 1
x1 x2 x3 x4 x5) = 01001, z1 = 1, z2 = 1
x1 x2 x3 x4 x5) = 01010, z1 = 1, z2 = 1
x1 x2 x3 x4 x5) = 01011, z1 = 1, z2 = 1

x1 x2 x3 x4 x5) = 01100, z1 = 1, z2 = 1
x1 x2 x3 x4 x5) = 01101, z1 = 1, z2 = 1
x1 x2 x3 x4 x5) = 01110, z1 = 1, z2 = 1
x1 x2 x3 x4 x5) = 01111, z1 = 1, z2 = 1

x1 x2 x3 x4 x5) = 10000, z1 = 0, z2 = 0
x1 x2 x3 x4 x5) = 10001, z1 = 1, z2 = 1
x1 x2 x3 x4 x5) = 10010, z1 = 1, z2 = 1
x1 x2 x3 x4 x5) = 10011, z1 = 1, z2 = 1

x1 x2 x3 x4 x5) = 10100, z1 = 1, z2 = 1
x1 x2 x3 x4 x5) = 10101, z1 = 0, z2 = 0
x1 x2 x3 x4 x5) = 10110, z1 = 1, z2 = 1
x1 x2 x3 x4 x5) = 10111, z1 = 1, z2 = 1

x1 x2 x3 x4 x5) = 11000, z1 = 0, z2 = 0
x1 x2 x3 x4 x5) = 11001, z1 = 1, z2 = 1
x1 x2 x3 x4 x5) = 11010, z1 = 1, z2 = 1
x1 x2 x3 x4 x5) = 11011, z1 = 1, z2 = 1

x1 x2 x3 x4 x5) = 11100, z1 = 0, z2 = 0
x1 x2 x3 x4 x5) = 11101, z1 = 0, z2 = 0
x1 x2 x3 x4 x5) = 11110, z1 = 0, z2 = 0
x1 x2 x3 x4 x5) = 11111, z1 = 0, z2 = 0
```

2.9 Design a dataflow module for a full adder using logic gates that were designed using dataflow modeling. Recall that a full adder is a combinational circuit that adds two operand bits: the augend *a* and the addend *b* plus a carry-in bit *cin*. The carry-in bit represents the carry-out of the previous lower-order stage. A full adder produces two outputs: a sum bit *sum* and carry-out bit *cout*. The truth table for a full adder is shown below. Obtain the test bench module and the outputs.

a	*b*	*cin*	*cout*	*sum*
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

$$sum = a \oplus b \oplus cin$$

$$cout = cin\,(a \oplus b) + ab$$

```
//dataflow for a full adder

module full_adder_df (a, b, cin, sum, cout);

//define inputs and outputs
input a, b, cin;
output sum, cout;

//define internal nets
wire net1, net2, net3;

//design the sum for the full adder
xor3_df  inst1 (a, b, cin, sum);

//design the carry-out for the full adder
xor2_df  inst2 (a, b, net1);
and2_df  inst3 (cin, net1, net2);
and2_df  inst4 (a, b, net3);
or2_df   inst5 (net2, net3, cout);

endmodule
```

```
//test bench for full adder

module full_adder_df_tb;

//inputs are reg for test bench
//outputs are wire for test bench
reg a, b, cin;
wire sum, cout;

//apply input vectors and display outputs
initial
begin: apply_stimulus
   reg [3:0] invect;
   for (invect = 0; invect < 8; invect = invect + 1)
      begin
         {a, b, cin} = invect [3:0];
         #10 $display ("a, b, cin) = %b,
                        sum = %b, cout = %b",
                        {a, b, cin}, sum, cout);
      end
end

//instantiate the module into the test bench
full_adder_df inst1 (a, b, cin, sum, cout);

endmodule
```

```
a, b, cin) = 000, sum = 0, cout = 0
a, b, cin) = 001, sum = 1, cout = 0
a, b, cin) = 010, sum = 1, cout = 0
a, b, cin) = 011, sum = 0, cout = 1

a, b, cin) = 100, sum = 1, cout = 0
a, b, cin) = 101, sum = 0, cout = 1
a, b, cin) = 110, sum = 0, cout = 1
a, b, cin) = 111, sum = 1, cout = 1
```

2.10 Design a 4-bit comparator for two 4-bit unsigned binary operands: A [3:0] and B [3:0] using behavioral modeling. There are three outputs:

$$A < B,\ A = B,\ A > B$$

Obtain the test bench module and the outputs for 30 input vectors.

$$\begin{aligned}(A < B) = {} & a_3'b_3 + (a_3 \oplus b_3)'\,a_2'b_2 + \\ & (a_3 \oplus b_3)'(a_2 \oplus b_2)'a_1'b_1 + \\ & (a_3 \oplus b_3)'(a_2 \oplus b_2)'(a_1 \oplus b_1)'a_0'b_0\end{aligned}$$

$$(A = B) = (a_3 \oplus b_3)'(a_2 \oplus b_2)'(a_1 \oplus b_1)'(a_0 \oplus b_0)'$$

$$\begin{aligned}(A > B) = {} & a_3b_3' + (a_3 \oplus b_3)'a_2b_2' + \\ & (a_3 \oplus b_3)'(a_2 \oplus b_2)'a_1b_1' + \\ & (a_3 \oplus b_3)'(a_2 \oplus b_2)'(a_1 \oplus b_1)'a_0b_0'\end{aligned}$$

```
//behavioral for a 4-bit comparator

module comparator4_bh (a, b, a_lt_b, a_eq_b, a_gt_b);


//define inputs and outputs
input [3:0] a, b;
output a_lt_b, a_eq_b, a_gt_b;


//variables are reg in always
reg a_lt_b, a_eq_b, a_gt_b;

                              //continued on next page
```

```
//--------------------------------------------------
//determine if A is less than B
always @ (a or b)
begin
   if (~a[3] & b[3])
      a_lt_b = 1'b1;

   else if ((a[3] ^~ b[3]) & (~a[2] & b[2]))
      a_lt_b = 1'b1;

   else if ((a[3] ^~ b[3]) & (a[2] ^~ b[2])
            & (~a[1] & b[1]))
      a_lt_b = 1'b1;

   else if ((a[3] ^~ b[3]) & (a[2] ^~ b[2])
            & (a[1] ^~ b[1]) & (~a[0] & b[0]))
      a_lt_b = 1'b1;

   else a_lt_b = 1'b0;
end

//--------------------------------------------------
//determine if A is equal to B
always @ (a or b)
begin
   if ((a[3] ^~ b[3]) & (a[2] ^~ b[2]) & (a[1] ^~ b[1])
            & (a[0] ^~ b[0]))
      a_eq_b = 1'b1;

   else a_eq_b = 1'b0;
end

//--------------------------------------------------
//determine if A is greater than B
always @ (a or b)
begin
   if (a[3] & ~b[3])
      a_gt_b = 1'b1;

   else if ((a[3] ^~ b[3]) & (a[2] & ~b[2]))
      a_gt_b = 1'b1;

   else if ((a[3] ^~ b[3]) & (a[2] ^~ b[2])
               & (a[1] & ~b[1]))
      a_gt_b = 1'b1;
                              //continued on next page
```

```
else if ((a[3] ^~ b[3]) & (a[2] ^~ b[2]) & (a[1] ^~ b[1])
            & (a[0] & ~b[0]))
   a_gt_b = 1'b1;

   else a_gt_b = 1'b0;
end

endmodule
```

```
//test bench to compare two 4-bit operands A and B

module comparator4_bh_tb;

//inputs are reg for test bench
//outputs are wire for test bench
reg [3:0] a, b;
wire a_lt_b, a_eq_b, a_gt_b;;

//display inputs and outputs
initial
$monitor ("a = %b, b = %b, a_lt_b = %b, a_eq_b = %b,
               a_gt_b = %b;",
            a, b, a_lt_b, a_eq_b, a_gt_b);

//apply input vectors
initial
begin
   #0    a = 4'b0000;   b = 4'b0001;
   #10   a = 4'b0011;   b = 4'b0011;
   #10   a = 4'b1111;   b = 4'b1011;

   #10   a = 4'b0110;   b = 4'b0111;
   #10   a = 4'b0110;   b = 4'b0110;
   #10   a = 4'b1110;   b = 4'b0111;

   #10   a = 4'b0110;   b = 4'b0111;
   #10   a = 4'b1010;   b = 4'b1010;
   #10   a = 4'b1110;   b = 4'b0111;

   #10   a = 4'b1000;   b = 4'b1100;
   #10   a = 4'b1011;   b = 4'b1011;
   #10   a = 4'b1110;   b = 4'b1011;

                           //continued on next page
   #10   a = 4'b1001;   b = 4'b1100;
   #10   a = 4'b0001;   b = 4'b0001;
   #10   a = 4'b1110;   b = 4'b0011;

   #10   a = 4'b0100;   b = 4'b0111;
   #10   a = 4'b0111;   b = 4'b0111;
   #10   a = 4'b1110;   b = 4'b0110;

   #10   a = 4'b1110;   b = 4'b1111;
   #10   a = 4'b1000;   b = 4'b1000;
   #10   a = 4'b1110;   b = 4'b0101;

   #10   a = 4'b1010;   b = 4'b1100;
   #10   a = 4'b1010;   b = 4'b1010;
   #10   a = 4'b1010;   b = 4'b1001;

   #10   a = 4'b1000;   b = 4'b1100;
   #10   a = 4'b0011;   b = 4'b0011;
   #10   a = 4'b0001;   b = 4'b0000;

   #10   a = 4'b1000;   b = 4'b1001;
   #10   a = 4'b0110;   b = 4'b0110;
   #10   a = 4'b1000;   b = 4'b0111;

   #10   $stop;
end

//instantiate the module into the test bench
comparator4_bh inst1 (a, b, a_lt_b, a_eq_b, a_gt_b);

endmodule
```

```
a = 0000, b = 0001, a_lt_b = 1, a_eq_b = 0, a_gt_b = 0;
a = 0011, b = 0011, a_lt_b = 0, a_eq_b = 1, a_gt_b = 0;
a = 1111, b = 1011, a_lt_b = 0, a_eq_b = 0, a_gt_b = 1;

a = 0110, b = 0111, a_lt_b = 1, a_eq_b = 0, a_gt_b = 0;
a = 0110, b = 0110, a_lt_b = 0, a_eq_b = 1, a_gt_b = 0;
a = 1110, b = 0111, a_lt_b = 0, a_eq_b = 0, a_gt_b = 1;

a = 0110, b = 0111, a_lt_b = 1, a_eq_b = 0, a_gt_b = 0;
a = 1010, b = 1010, a_lt_b = 0, a_eq_b = 1, a_gt_b = 0;
a = 1110, b = 0111, a_lt_b = 0, a_eq_b = 0, a_gt_b = 1;

a = 1000, b = 1100, a_lt_b = 1, a_eq_b = 0, a_gt_b = 0;
a = 1011, b = 1011, a_lt_b = 0, a_eq_b = 1, a_gt_b = 0;
a = 1110, b = 1011, a_lt_b = 0, a_eq_b = 0, a_gt_b = 1;

a = 1001, b = 1100, a_lt_b = 1, a_eq_b = 0, a_gt_b = 0;
a = 0001, b = 0001, a_lt_b = 0, a_eq_b = 1, a_gt_b = 0;
a = 1110, b = 0011, a_lt_b = 0, a_eq_b = 0, a_gt_b = 1;

a = 0100, b = 0111, a_lt_b = 1, a_eq_b = 0, a_gt_b = 0;
a = 0111, b = 0111, a_lt_b = 0, a_eq_b = 1, a_gt_b = 0;
a = 1110, b = 0110, a_lt_b = 0, a_eq_b = 0, a_gt_b = 1;

a = 1110, b = 1111, a_lt_b = 1, a_eq_b = 0, a_gt_b = 0;
a = 1000, b = 1000, a_lt_b = 0, a_eq_b = 1, a_gt_b = 0;
a = 1110, b = 0101, a_lt_b = 0, a_eq_b = 0, a_gt_b = 1;

a = 1010, b = 1100, a_lt_b = 1, a_eq_b = 0, a_gt_b = 0;
a = 1010, b = 1010, a_lt_b = 0, a_eq_b = 1, a_gt_b = 0;
a = 1010, b = 1001, a_lt_b = 0, a_eq_b = 0, a_gt_b = 1;

a = 1000, b = 1100, a_lt_b = 1, a_eq_b = 0, a_gt_b = 0;
a = 0011, b = 0011, a_lt_b = 0, a_eq_b = 1, a_gt_b = 0;
a = 0001, b = 0000, a_lt_b = 0, a_eq_b = 0, a_gt_b = 1;

a = 1000, b = 1001, a_lt_b = 1, a_eq_b = 0, a_gt_b = 0;
a = 0110, b = 0110, a_lt_b = 0, a_eq_b = 1, a_gt_b = 0;
a = 1000, b = 0111, a_lt_b = 0, a_eq_b = 0, a_gt_b = 1;
```

2.11 This problem and the next two problems all design a binary-to-Gray code converter using different design techniques: this problem uses dataflow modeling with the continuous **assign** statement; Problem 2.12 uses behavioral modeling with the **always** statement; Problem 2.13 uses behavioral modeling with the

case statement. The binary-to-Gray code conversion table is shown below. Obtain the test bench module and the outputs.

Binary Code				Gray Code			
b_3	b_2	b_1	b_0	g_3	g_2	g_1	g_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	1
0	0	1	0	0	0	1	1
0	0	1	1	0	0	1	0
0	1	0	0	0	1	1	0
0	1	0	1	0	1	1	1
0	1	1	0	0	1	0	1
0	1	1	1	0	1	0	0
1	0	0	0	1	1	0	0
1	0	0	1	1	1	0	1
1	0	1	0	1	1	1	1
1	0	1	1	1	1	1	0
1	1	0	0	1	0	1	0
1	1	0	1	1	0	1	1
1	1	1	0	1	0	0	1
1	1	1	1	1	0	0	0

$$g_3 = b_3$$

$$g_2 = b_3'b_2 + b_3b_2' = b_3 \oplus b_2$$

$$g_1 = b_2'b_1 + b_2b_1' = b_2 \oplus b_1$$

$$g_0 = b_1'b_0 + b_1b_0' = b_1 \oplus b_0$$

```
//dataflow binary-to-gray using continuous assign
module bin_to_gray_assign (bin, gray);

input [3:0] bin;     //define inputs and outputs
output [3:0] gray;

assign   gray[3] = bin[3],
         gray[2] = bin[3] ^ bin[2],
         gray[1] = bin[2] ^ bin[1],
         gray[0] = bin[1] ^ bin[0];
endmodule
```

```
//test bench for the dataflow binary-to-gray
//code converter using the continuous assign statement

module bin_to_gray_assign_tb;

//inputs are reg for test bench
//outputs are wire for test bench
reg [3:0] bin;
wire [3:0] gray;

//apply input and display outputs
initial
begin: apply_stimulus
   reg [4:0] invect;
   for (invect = 0; invect < 16; invect = invect + 1)
      begin
         bin = invect [4:0];
         #10 $display ("binary = %b, gray = %b",
                        bin, gray);
      end
end

//instantiate the module into the test bench
bin_to_gray_assign inst1 (bin, gray);
endmodule
```

```
binary = 0000, gray = 0000
binary = 0001, gray = 0001
binary = 0010, gray = 0011
binary = 0011, gray = 0010

binary = 0100, gray = 0110
binary = 0101, gray = 0111
binary = 0110, gray = 0101
binary = 0111, gray = 0100

binary = 1000, gray = 1100
binary = 1001, gray = 1101
binary = 1010, gray = 1111
binary = 1011, gray = 1110

binary = 1100, gray = 1010
binary = 1101, gray = 1011
binary = 1110, gray = 1001
binary = 1111, gray = 1000
```

2.12 Repeat Problem 2.11 for the binary-to-Gray code converter using behavioral modeling with the **always** statement. Obtain the test bench module and the outputs.

```
//behavioral for behavioral binary-to-gray
//converter using the always statement
module bin_to_gray_always (bin, gray);

input [3:0] bin;     //define inputs and outputs
output [3:0] gray;

//variables are declared as reg in always
reg [3:0] gray;

//design the binary-to-gray converter
always @ (bin)
begin
   gray[3] = bin[3];
   gray[2] = bin[3] ^ bin[2];
   gray[1] = bin[2] ^ bin[1];
   gray[0] = bin[1] ^ bin[0];
end
endmodule
```

```
//test bench for the behavioral binary-to-gray
//converter using the always statement
module bin_to_gray_always_tb;

reg [3:0] bin;     //inputs are reg for test bench
wire [3:0] gray;   //outputs are wire for test bench


initial            //apply input and display outputs
begin: apply_stimulus
reg [4:0] invect;
for (invect = 0; invect < 16; invect = invect + 1)
   begin
      bin = invect [4:0];
      #10 $display ("binary = %b, gray = %b",
                     bin, gray);
   end
end

//instantiate the module into the test bench
bin_to_gray_always inst1 (bin, gray);
endmodule
```

```
binary = 0000, gray = 0000
binary = 0001, gray = 0001
binary = 0010, gray = 0011
binary = 0011, gray = 0010

binary = 0100, gray = 0110
binary = 0101, gray = 0111
binary = 0110, gray = 0101
binary = 0111, gray = 0100

binary = 1000, gray = 1100
binary = 1001, gray = 1101
binary = 1010, gray = 1111
binary = 1011, gray = 1110

binary = 1100, gray = 1010
binary = 1101, gray = 1011
binary = 1110, gray = 1001
binary = 1111, gray = 1000
```

2.13 Repeat Problem 2.11 for the binary-to-Gray code converter using behavioral modeling with the **case** statement. Obtain the test bench module and the outputs.

```
//behavioral binary-to-gray using the case statement

module bin_to_gray_case (bin, gray);

//define inputs and outputs
input [3:0] bin;
output [3:0] gray;

//variables are declared as reg in always
reg [3:0] gray;

always @ (bin)
begin
   case (bin)
      4'b0000 : gray = 4'b0000;
      4'b0001 : gray = 4'b0001;
      4'b0010 : gray = 4'b0011;
      4'b0011 : gray = 4'b0010;
                           //continued on next page
      4'b0100 : gray = 4'b0110;
      4'b0101 : gray = 4'b0111;
      4'b0110 : gray = 4'b0101;
      4'b0111 : gray = 4'b0100;

      4'b1000 : gray = 4'b1100;
      4'b1001 : gray = 4'b1101;
      4'b1010 : gray = 4'b1111;
      4'b1011 : gray = 4'b1110;

      4'b1100 : gray = 4'b1010;
      4'b1101 : gray = 4'b1011;
      4'b1110 : gray = 4'b1001;
      4'b1111 : gray = 4'b1000;

      default : gray = 4'b0000;
   endcase
end
endmodule
```

```
//test bench for the behavioral binary-to-gray
//converter using the case statement
module bin_to_gray_case_tb;

//inputs are reg for test bench
//outputs are wire for test bench
reg [3:0] bin;
wire [3:0] gray;

//apply inputs and display outputs
initial
begin: apply_stimulus
   reg [4:0] invect;
   for (invect = 0; invect < 16; invect = invect + 1)
      begin
         bin = invect [4:0];
         #10 $display ("binary = %b, gray = %b",
                           bin, gray);
      end
end

//instantiate the module into the test bench
bin_to_gray_case inst1 (bin, gray);
endmodule
```

```
binary = 0000, gray = 0000
binary = 0001, gray = 0001
binary = 0010, gray = 0011
binary = 0011, gray = 0010

binary = 0100, gray = 0110
binary = 0101, gray = 0111
binary = 0110, gray = 0101
binary = 0111, gray = 0100

binary = 1000, gray = 1100
binary = 1001, gray = 1101
binary = 1010, gray = 1111
binary = 1011, gray = 1110

binary = 1100, gray = 1010
binary = 1101, gray = 1011
binary = 1110, gray = 1001
binary = 1111, gray = 1000
```

2.14 Use structural modeling to design a 4:1 multiplexer using logic gates that were designed using dataflow modeling. Obtain the test bench module and the outputs for 16 combinations of the inputs.

```
//structural for a 4to1 multiplexer
//using dataflow logic gates
module mux_4to1_df (sel, data, z1);

//define inputs and output
input [1:0] sel;
input [3:0] data;
output z1;

//define internal nets
wire net1, net2, net3, net4;

//design the 4to1 multiplexer
and3_df  inst1 (data[0], ~sel[1], ~sel[0], net1),
         inst2 (data[1], ~sel[1], sel[0], net2),
         inst3 (data[2], sel[1], ~sel[0], net3),
         inst4 (data[3], sel[1], sel[0], net4);

or4_df   inst5 (net1, net2, net3, net4, z1);

endmodule
```

```
//test bench for the 4:1 multiplexer structural module

module mux_4to1_df_tb;

//inputs are reg for test bench
//outputs are wire for test bench
reg [1:0] sel;
reg [3:0] data;
wire z1;

initial
$monitor ("sel = %b, data = %b, z1 = %b",
            sel, data, z1);

//apply stimulus
initial
begin
   #0    sel = 2'b00;   data = 4'b0001;   //z1 = 1
   #10   sel = 2'b01;   data = 4'b1001;   //z1 = 0
   #10   sel = 2'b10;   data = 4'b1000;   //z1 = 0
   #10   sel = 2'b11;   data = 4'b1001;   //z1 = 1

   #10   sel = 2'b00;   data = 4'b0100;   //z1 = 0
   #10   sel = 2'b01;   data = 4'b1000;   //z1 = 0
   #10   sel = 2'b10;   data = 4'b1100;   //z1 = 1
   #10   sel = 2'b11;   data = 4'b1101;   //z1 = 1

   #10   sel = 2'b00;   data = 4'b0110;   //z1 = 0
   #10   sel = 2'b01;   data = 4'b0000;   //z1 = 0
   #10   sel = 2'b10;   data = 4'b1001;   //z1 = 0
   #10   sel = 2'b11;   data = 4'b0100;   //z1 = 0

   #10   sel = 2'b00;   data = 4'b0111;   //z1 = 1
   #10   sel = 2'b01;   data = 4'b0010;   //z1 = 1
   #10   sel = 2'b10;   data = 4'b1101;   //z1 = 1
   #10   sel = 2'b11;   data = 4'b1100;   //z1 = 1

   #10   $stop;
end

//instantiate the module into the test bench
mux_4to1_df inst1 (sel, data, z1);

endmodule
```

```
sel[1:0]  data[3:0]

-----------------------------
sel = 00, data = 0001, z1 = 1
sel = 01, data = 1001, z1 = 0
sel = 10, data = 1000, z1 = 0
sel = 11, data = 1001, z1 = 1

sel = 00, data = 0100, z1 = 0
sel = 01, data = 1000, z1 = 0
sel = 10, data = 1100, z1 = 1
sel = 11, data = 1101, z1 = 1

sel = 00, data = 0110, z1 = 0
sel = 01, data = 0000, z1 = 0
sel = 10, data = 1001, z1 = 0
sel = 11, data = 0100, z1 = 0

sel = 00, data = 0111, z1 = 1
sel = 01, data = 0010, z1 = 1
sel = 10, data = 1101, z1 = 1
sel = 11, data = 1100, z1 = 1
```

2.15 Design a behavioral module using the **case** statement to design an 8:1 multiplexer. Obtain the test bench module and the outputs for 20 combinations of the inputs.

```
//behavioral 8:1 multiplexer using the case statement

module mux_8to1_case_stmt (sel, data, z1);

//define the inputs and the outputs
input [2:0] sel;
input [7:0] data;
output z1;

//variables are reg in always
reg z1;

                            //continued on next page
```

```
always @ (sel or data)
begin
   case (sel)
      (0) : z1 = data[0];
      (1) : z1 = data[1];
      (2) : z1 = data[2];
      (3) : z1 = data[3];
      (4) : z1 = data[4];
      (5) : z1 = data[5];
      (6) : z1 = data[6];
      (7) : z1 = data[7];
      default: z1 = 1'b0;
   endcase
end

endmodule
```

```
//test bench for 8:1 multiplexer using case statement
module mux_8to1_case_stmt_tb;

//inputs are reg in test bench
//outputs are wire in test bench
reg [2:0] sel;
reg [7:0] data;
wire z1;

//display variables
initial
$monitor ("sel = %b, data = %b, z1 = %b",
            sel, data, z1);

//apply input vectors
initial
begin
   #0    sel = 3'b000;  data = 8'b0000_0001; //z1 = 1
   #10   sel = 3'b001;  data = 8'b0100_1011; //z1 = 1
   #10   sel = 3'b010;  data = 8'b0001_1010; //z1 = 0
   #10   sel = 3'b011;  data = 8'b0010_0101; //z1 = 0

   #10   sel = 3'b000;  data = 8'b0000_0001; //z1 = 1
   #10   sel = 3'b001;  data = 8'b0100_1011; //z1 = 1
   #10   sel = 3'b010;  data = 8'b0001_1110; //z1 = 1
   #10   sel = 3'b011;  data = 8'b0010_1101; //z1 = 1
                            //continued on next page
   #10   sel = 3'b100;  data = 8'b0000_0001; //z1 = 0
   #10   sel = 3'b101;  data = 8'b0110_1011; //z1 = 1
   #10   sel = 3'b110;  data = 8'b0001_1110; //z1 = 0
   #10   sel = 3'b111;  data = 8'b1010_1101; //z1 = 1

   #10   sel = 3'b100;  data = 8'b0000_0001; //z1 = 0
   #10   sel = 3'b101;  data = 8'b0100_1011; //z1 = 0
   #10   sel = 3'b110;  data = 8'b0001_1110; //z1 = 0
   #10   sel = 3'b111;  data = 8'b0010_1101; //z1 = 0

   #10   sel = 3'b100;  data = 8'b0001_0001; //z1 = 1
   #10   sel = 3'b101;  data = 8'b0110_1011; //z1 = 1
   #10   sel = 3'b110;  data = 8'b0111_1110; //z1 = 1
   #10   sel = 3'b111;  data = 8'b1011_1101; //z1 = 1
   #10   $stop;
end

//instantiate the module into the test bench
mux_8to1_case_stmt inst1 (sel, data, z1);
endmodule
```

```
sel = 000, data = 00000001, z1 = 1
sel = 001, data = 01001011, z1 = 1
sel = 010, data = 00011010, z1 = 0
sel = 011, data = 00100101, z1 = 0

sel = 000, data = 00000001, z1 = 1
sel = 001, data = 01001011, z1 = 1
sel = 010, data = 00011110, z1 = 1
sel = 011, data = 00101101, z1 = 1

sel = 100, data = 00000001, z1 = 0
sel = 101, data = 01101011, z1 = 1
sel = 110, data = 00011110, z1 = 0
sel = 111, data = 10101101, z1 = 1

sel = 100, data = 00000001, z1 = 0
sel = 101, data = 01001011, z1 = 0
sel = 110, data = 00011110, z1 = 0
sel = 111, data = 00101101, z1 = 0

sel = 100, data = 00010001, z1 = 1
sel = 101, data = 01101011, z1 = 1
sel = 110, data = 01111110, z1 = 1
sel = 111, data = 10111101, z1 = 1
```

2.16 Design a behavioral module that adds 5 to a variable *count* to obtain a maximum value of 100 and displays the outputs.

```
//add 5 to count to obtain a value of 100

module repeat4;

integer count;

initial
begin
   count = 0;
   repeat (21)
   begin
      $display ("count = %d", count);
      count = count + 5;
   end
end

endmodule
```

```
count = 0           count = 55
count = 5           count = 60
count = 10          count = 65
count = 15          count = 70
count = 20          count = 75
count = 25          count = 80
count = 30          count = 85
count = 35          count = 90
count = 40          count = 95
count = 45          count = 100
count = 50
```

2.17 Plot the following equation on a Karnaugh map, then change the equation to an exclusive-NOR format. Obtain the design module using built-in primitive logic gates. Then obtain the test bench module and the outputs for all combinations of the four variables.

$$z_1 = x_1'x_2'x_3'x_4' + x_1'x_2x_3'x_4 + x_1x_2x_3x_4 + x_1x_2'x_3x_4'$$

$x_1x_2 \backslash x_3x_4$	0 0	0 1	1 1	1 0
0 0	^{0} 1	^{1} 0	^{3} 0	^{2} 0
0 1	^{4} 0	^{5} 1	^{7} 0	^{6} 0
1 1	^{12} 0	^{13} 0	^{15} 1	^{14} 0
1 0	^{8} 0	^{9} 0	^{11} 0	^{10} 1

z_1

$$z_1 = x_1'x_2'x_3'x_4' + x_1'x_2x_3'x_4 + x_1x_2x_3x_4 + x_1x_2'x_3x_4'$$

$$= x_1'x_3'(x_2'x_4' + x_2x_4) + x_1x_3(x_2x_4 + x_2'x_4')$$

$$= x_1'x_3'(x_2 \oplus x_4)' + x_1x_3(x_2 \oplus x_4)'$$

$$= (x_2 \oplus x_4)'(x_1'x_3' + x_1x_3)$$

$$= (x_2 \oplus x_4)'(x_1 \oplus x_3)'$$

```
//synthesize an equation using built-in-primitives

module xnor_bip (x1, x2, x3, x4, z1);

//define inputs and output
input x1, x2, x3, x4;
output z1;

//design the equation using xnor bips
xnor  inst1 (net1, x2, x4),
      inst2 (net2, x1, x3);

and   inst3 (z1, net1, net2);

endmodule
```

```
//test bench for the bip equation
module xnor_bip_tb;

//inputs are reg for test bench
//outputs are wire for test bench
reg x1, x2, x3, x4;
wire z1;

//apply input vectors and display output
initial
begin: apply_stimulus
   reg [4:0] invect;
   for(invect = 0; invect < 16; invect = invect + 1)
      begin
         {x1, x2, x3, x4} = invect [4:0];
         #10 $display ("{x1, x2, x3, x4} = %b, z1 = %b",
                        {x1, x2, x3, x4}, z1);
      end
end

//instantiate the module into the test bench
xnor_bip inst1 (x1, x2, x3, x4, z1);

endmodule
```

```
{x1, x2, x3, x4} = 0000, z1 = 1
{x1, x2, x3, x4} = 0001, z1 = 0
{x1, x2, x3, x4} = 0010, z1 = 0
{x1, x2, x3, x4} = 0011, z1 = 0

{x1, x2, x3, x4} = 0100, z1 = 0
{x1, x2, x3, x4} = 0101, z1 = 1
{x1, x2, x3, x4} = 0110, z1 = 0
{x1, x2, x3, x4} = 0111, z1 = 0

{x1, x2, x3, x4} = 1000, z1 = 0
{x1, x2, x3, x4} = 1001, z1 = 0
{x1, x2, x3, x4} = 1010, z1 = 1
{x1, x2, x3, x4} = 1011, z1 = 0

{x1, x2, x3, x4} = 1100, z1 = 0
{x1, x2, x3, x4} = 1101, z1 = 0
{x1, x2, x3, x4} = 1110, z1 = 0
{x1, x2, x3, x4} = 1111, z1 = 1
```

2.18 Obtain a minimized equation for z_1 in a sum-of-products representation and for z_2 in a product-of-sums representation for the Karnaugh map shown below, where the outputs are $12 \le z_1(z_2) < 3$. The obtain the design module using built-in primitives, the test bench module, and the outputs.

x_1x_2 \ x_3x_4	0 0	0 1	1 1	1 0
0 0	^{0} 1	^{1} 1	^{3} 0	^{2} 1
0 1	^{4} 0	^{5} 0	^{7} 0	^{6} 0
1 1	^{12} 1	^{13} 1	^{15} 1	^{14} 1
1 0	^{8} 0	^{9} 0	^{11} 0	^{10} 0

$$z_1 = x_1x_2 + x_1'x_2'x_3' + x_1'x_2'x_4'$$

$$z_2 = (x_1 + x_2')(x_1' + x_2)(x_1 + x_3' + x_4')$$

```
//built-in primitives for number 12 <= z1 < 3
module sop_pos_bip2 (x1, x2, x3, x4, z1, z2);

//define inputs and outputs
input x1, x2, x3, x4;
output z1, z2;

//design the logic using bips for z1 in a sum-of-products
and   inst1 (net1, x1, x2),
      inst2 (net2, ~x1, ~x2, ~x3),
      inst3 (net3, ~x1, ~x2, ~x4);
or    inst4 (z1, net1, net2, net3);

//design the logic using bips for z1 in a product-of-sums
or    inst5 (net5, x1, ~x2),
      inst6 (net6, ~x1, x2),
      inst7 (net7, x1, ~x3, ~x4);
and   inst8 (z2, net5, net6, net7);

endmodule
```

```
//test bench for number 12 <= z1 < 3
module sop_pos_bip2_tb;

//inputs are reg for test bench
//outputs are wire for test bench
reg x1, x2, x3, x4;
wire z1, z2;

//apply input vectors and display variables
initial
begin: apply_stimulus
   reg [4:0] invect;
   for (invect = 0; invect < 16; invect = invect + 1)
      begin
         {x1, x2, x3, x4} = invect [4:0];
         #10 $display ("{x1 x2 x3 x4} = %b,
                        z1 = %b, z2 = %b",
                        {x1, x2, x3, x4}, z1, z2);
      end
end

//instantiate the module into the test bench
sop_pos_bip2 inst1 (x1, x2, x3, x4, z1, z2);

endmodule
```

```
{x1 x2 x3 x4} = 0000, z1 = 1, z2 = 1
{x1 x2 x3 x4} = 0001, z1 = 1, z2 = 1
{x1 x2 x3 x4} = 0010, z1 = 1, z2 = 1
{x1 x2 x3 x4} = 0011, z1 = 0, z2 = 0

{x1 x2 x3 x4} = 0100, z1 = 0, z2 = 0
{x1 x2 x3 x4} = 0101, z1 = 0, z2 = 0
{x1 x2 x3 x4} = 0110, z1 = 0, z2 = 0
{x1 x2 x3 x4} = 0111, z1 = 0, z2 = 0

{x1 x2 x3 x4} = 1000, z1 = 0, z2 = 0
{x1 x2 x3 x4} = 1001, z1 = 0, z2 = 0
{x1 x2 x3 x4} = 1010, z1 = 0, z2 = 0
{x1 x2 x3 x4} = 1011, z1 = 0, z2 = 0

{x1 x2 x3 x4} = 1100, z1 = 1, z2 = 1
{x1 x2 x3 x4} = 1101, z1 = 1, z2 = 1
{x1 x2 x3 x4} = 1110, z1 = 1, z2 = 1
{x1 x2 x3 x4} = 1111, z1 = 1, z2 = 1
```