6. [15 points] Use Shannon's expansion theorem around **a** and **b** for the function

Y = ab'cdef + a'b'c'd'e + b'c'ef' + abcde'f

so that it can be implemented using only four-variable function generators (4 variable LUT). Draw a block diagram to indicate how **Y** can be implemented using only four-variable function generators. Indicate the function realized by each four-variable function generator.

6.16 Decompose the following function using Shannon's decomposition around the variable *X6*. Do not simplify the function.

$$F = X_1'X_2X_3'X_4X_6 + X_2'X_3'X_4X_6' + X_2'X_4' + X_3X_4X_5X_6 + X_3'X_4X_6' + X_1X_3$$

Write an expression for F in terms of the decomposed functions and X_6.

6.17 Use Shannon's expansion theorem around a and b for the function

$$Y = abcde + cde'f + a'b'c'def + bcdef' + ab'cd'ef' + a'bc'de'f + abcd'e'f$$

so that it can be implemented using only 4-variable function generators. Draw a block diagram to indicate how Y can be implemented using only 4-variable function generators. Indicate the function realized by each 4-variable function generator.

6.18 Use Shannon's expansion theorem around e and f for the function

$$Y = ab'cdef + a'bc'd'e + b'c'ef' + abcde'f$$

so that it can be implemented using a minimum number of 4-variable functions. Rewrite Y to indicate how it will be implemented using 4-variable function generators and draw a block diagram. Indicate the function generated by each function generator.

6.19 **(a)** Use Shannon's expansion theorem around a for the function

$$Y = ab'cd'e + a'bc'd'e + b'c'e + abcde$$

so that it can be implemented using 4-variable functions.

(b) Use the expanded function to show how Y can be implemented using one Figure 6-3 logic block. Mark (highlight) the input signals and the activated paths on a copy of Figure 6-3.

(c) Give the contents of the three LUTs.

6.20 **(a)** If logic blocks of Figure 6-1(a) are used, how many LUTs are required to build a 4-bit adder with accumulator?

(b) If an FPGA with built-in carry-chain logic as shown in Figure 6-11 is used, how many 4-input LUTs are required?

(c) Design a 4-bit adder-subtracter with accumulator using an FPGA with carry-chain logic and 4-input LUTs. Assume a control signal *Su* which is 0 for addition and 1 for subtraction. Show the required connections on a diagram similar to that shown in Figure 6-11 and give the function realized by each LUT.

6.22 **(a)** Use Shannon's expansion theorem to expand the following function around A and then expand each sub-function around D:

$$Z = AB'CD'E'F + A'BC'D'EF' + B'C'E'F + A'BC'E'F' + ABCDE$$

6.36 Consider the Verilog code

```
module example(a,b);
input[1:0] a;
output[1:0] b;
reg[1:0] b;

always @(a)
begin
   case(a)
   0: b = 2'd3;
   1: b = 2'd2;
   2: b = 2'd1;
   3: b = 2'd1;
   endcase
end

endmodule
```

(a) Show the hardware you would obtain if you synthesize the foregoing Verilog code without any optimizations. Explain your reasoning.

(b) Show optimized hardware emphasizing minimum area. Show the steps and the reasoning by which you obtained the optimized hardware.

7.2 Convert the following decimal numbers in the IEEE single precision format: **(i)** 25.25, **(ii)** 2000.22, **(iii)** 1, **(iv)** 0, **(v)** 1000, **(vi)** 8000, **(vii)** 10^6, **(viii)** -5.4, **(ix)** 1.0×2^{-140}, **(x)** 1.5×10^9

7.3 Convert the following decimal numbers to IEEE double precision format:
(i) 25.25, **(ii)** 2000.22, **(iii)** 1, **(iv)** 0, **(v)** 1000, **(vi)** 8000, **(vii)** 10^6, **(viii)** -5.4, **(ix)** 1.0×2^{-140}, **(x)** 1.5×10^9

7.4 What do the following hex representations mean if they are in IEEE single precision format?
(i) ABABABAB, **(ii)** 45454545, **(iii)** FFFFFFFF, **(iv)** 00000000, **(v)** 11111111, **(vi)** 01010101

8.10 Hamming codes are used for error detection and correction in communication and memory systems. Error detection and correction capability is incorporated in these codes by inserting extra bits into the data word. Addition of one parity bit can detect odd number of bit flips, but no error correction is possible with one parity bit. A (7,4) Hamming code has 4 bits of data but 7 bits in total, including the 3 parity bits. It can detect two errors and correct one error. This code can be constructed as follows: If we denote data bits as $d_4d_3d_2d_1$, the encoded code word would be $d_4d_3d_2p_4d_1p_2p_1$, where p_4, p_2, and p_1 are the added parity bits. These bits must satisfy the following conditions for even parity:

$$p_4 = d_2 \text{ XOR } d_3 \text{ XOR } d_4; \quad \ldots\ldots(1)$$
$$p_2 = d_1 \text{ XOR } d_3 \text{ XOR } d_4; \quad \ldots\ldots(2)$$
$$p_1 = d_1 \text{ XOR } d_2 \text{ XOR } d_4; \quad \ldots\ldots(3)$$

When the 7 bits are received/decoded, an error syndrome $S_3S_2S_1$ is calculated as follows:

$$p_4 \text{ XOR } d_2 \text{ XOR } d_3 \text{ XOR } d_4 = S_3; \quad \ldots\ldots(4)$$
$$p_2 \text{ XOR } d_1 \text{ XOR } d_3 \text{ XOR } d_4 = S_2; \quad \ldots\ldots(5)$$
$$p_1 \text{ XOR } d_1 \text{ XOR } d_2 \text{ XOR } d_4 = S_1; \quad \ldots\ldots(6)$$

The syndrome indicates which bit is wrong. For example, if the syndrome is 110, it indicates that bit 6 from right end (i.e., d3) has flipped. If $S_3S_2S_1$ is 000, there is no error.

(a) Is there any error in the code word 0110111? If yes, which bit? What was the original data? What must be the corrected code word?

(b) How will these 6 equations get modified for odd parity? Write the 6 equations for odd parity.

(c) Write a Verilog module for error detection without using tasks. The inputs to the module are the 7-bit encoded data word and the type of parity, and the output is the syndrome. The type of parity is encoded as 0 for odd parity and 1 for even parity.

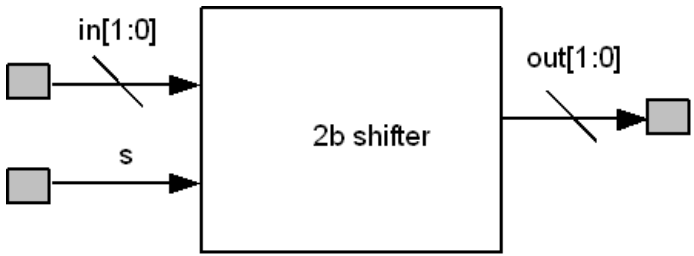
```
module  error_detector(data, PARITY, Syndrome)
{
}
```

8.24 Write a Verilog model for an N-bit comparator using an iterative circuit. In the module, use the parameter N to define the length of the input bit vectors A and B. The comparator outputs should be $EQ = 1$ if $A = B$, and $GT = 1$ if $A > B$. Use a

for loop to do the comparison on a bit-by-bit basis, starting with the high-order bits. Even though the comparison is done on a bit-by-bit basis, the final values of EQ and GT apply to A and B as a whole.

Exercise 1 – 2b shifter

- This is the same logic block as in Q2a of HW #2, but with bus notation for the inputs and outputs. This is a purely combinational logic block. The logic equations are:
  - out[0] = s' ● in[0]
  - out[1] = s' ● in[1] + s ● in[0]

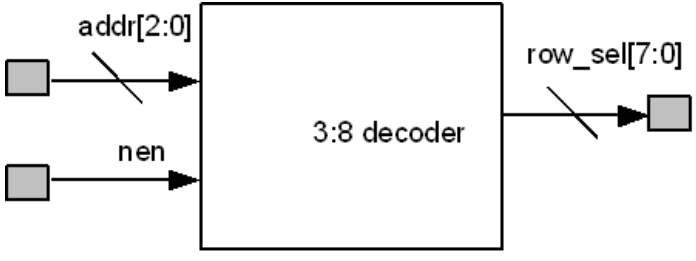


Ex 1 Solution

```
module shifter(in, out, s);
   input [1:0] in;
   input       s;

   output [1:0] out;

   wire   s;
   wire [1:0] in;
   wire   out;

   assign out[1] = (~s) & in[1] | s & in[0];
   assign out[0] = (~s) & in[0];

endmodule
```

Ex 2 – 3:8 row decoder with enable

- This decoder has inputs `addr[2:0]` and an active low enable `nen`. It drives 8 active high output lines `row_sel[7:0]`, one of which is driven when `nen` is asserted.

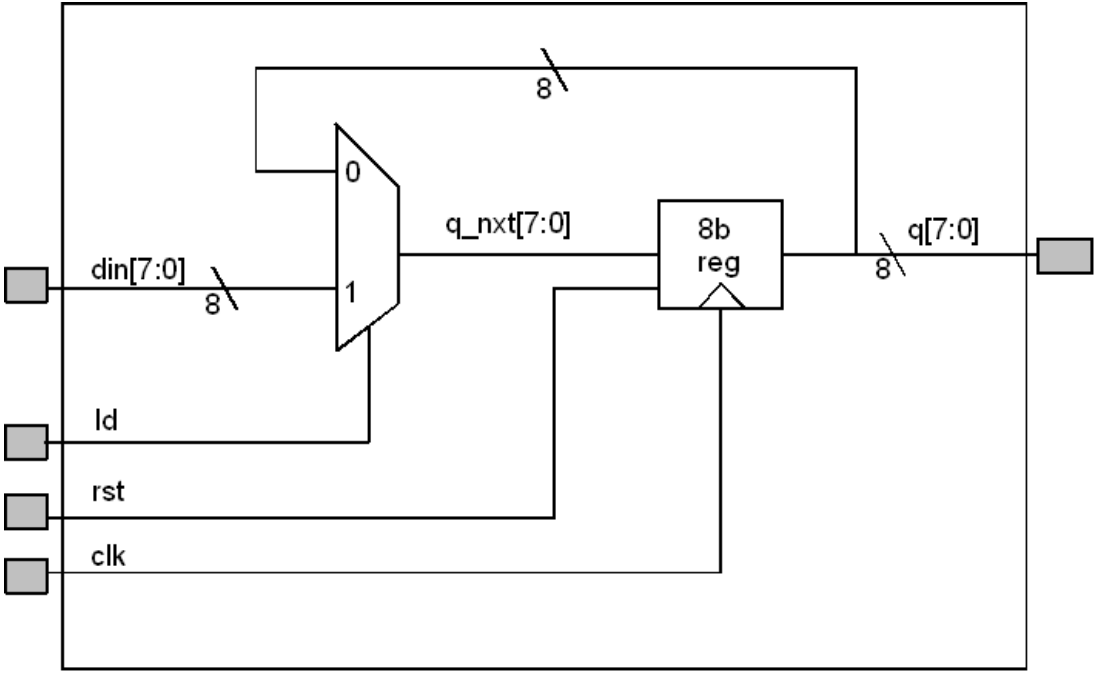


Ex 2 Solution

```
module row_decoder(row_sel, addr, nen);
   input [2:0] addr;
   input       nen;

   output [7:0] row_sel;

   wire [2:0]    addr;
   wire  nen;
   reg [7:0]     row_sel;

   // Use a case statement
   always @(addr or nen)
     begin
    if (nen)
      row_sel = 8'h0;
    else
      case (addr)
        3'h0: row_sel = 8'b0000_0001; // The _ is just for readability
        3'h1: row_sel = 8'b0000_0010;
        3'h2: row_sel = 8'b0000_0100;
        3'h3: row_sel = 8'b0000_1000;
        3'h4: row_sel = 8'b0001_0000;
        3'h5: row_sel = 8'b0010_0000;
        3'h6: row_sel = 8'b0100_0000;
        3'h7: row_sel = 8'b1000_0000;
      endcase // case(addr)
     end // always @ (addr or nen)

endmodule // row_decoder
```

Ex 3 – 8b register with load and synchronous reset

- This block implements an 8b wide register from DFFs. All flops are driven by a common clock and have a common reset `rst`. An input mux allows new data to be loaded into the register when `ld` is high; otherwise, the old data is recirculated.



Ex 3 Solution

```
module ld_reg8(din, clk, rst, ld, q);

   input [7:0] din;
   input       clk, rst, ld;
   output [7:0] q;

   wire [7:0]    din, q_nxt;
   wire   clk, rst, ld;
   reg [7:0]  q;

   //Logic on register inputs
   assign     q_nxt = ld ? din : q;

   // Update register
   always @(posedge clk)
     q <= rst ? 8'h0 : q_nxt;

endmodule // row_decoder
```