Tutorial 7 (SOLUTIONS)

NOTE THAT THERE ARE OTHER POSSIBLE SOLUTIONS TOO…

Question 1

C_2	O_1	C_1	O_0	C_0
D_in[4]	D_in[3]	D_in[2]	D_in[1]	D_in[0]

// Solution 1a – Behavioral Style of Modeling using if-else statements

```
module Dev_Encrypt (input [4:0] D_in, output reg [2:0] D_out);

  always @ (D_in)
  begin
        if  ( D_in[3] == 0 && D_in[1] == 0 )
            D_out = {D_in[4], D_in[2], D_in[0] };
        else if ( D_in[3] == 0 && D_in[1] == 1 )
            D_out = {D_in[0], D_in[4], D_in[2] };
        else if ( D_in[3] == 1 && D_in[1] == 0 )
            D_out = {D_in[2], D_in[0], D_in[4] };
        else
            D_out = { (~D_in[4]) , (~D_in[2]), (~D_in[0]) };
  end

endmodule
```

// Solution 1b – Behavioral Style of Modeling using case statements

```
module Dev_Encrypt (input [4:0] D_in, output reg [2:0] D_out);

  always @ (*)
  begin
        case ({D_in[3],D_in[1]})
           2'b00 : D_out = {D_in[4], D_in[2], D_in[0] };
           2'b01 : D_out = {D_in[0], D_in[4], D_in[2] };
           2'b10 : D_out = {D_in[2], D_in[0], D_in[4] };
           2'b11 : D_out = { ~D_in[4], ~D_in[2], ~D_in[0] };
           default : D_out = {D_in[4], D_in[2], D_in[0] };
           //the default statement is used to capture all other cases!
        endcase
  end

endmodule
```

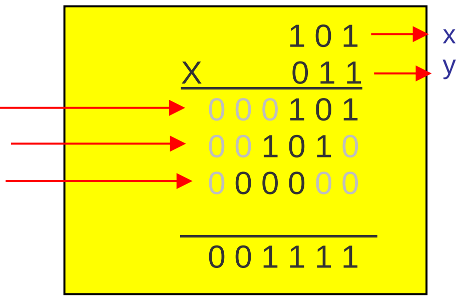
// Solution 2 – Dataflow Style of Modeling using continuous assignments

```
module Dev_Encrypt (input [4:0] D_in, output [2:0] D_out);

assign D_out =
(D_in[3]) ? ( D_in[1] ?  {~D_in[4], ~D_in[2], ~D_in[0]} :
                         {D_in[2], D_in[0], D_in[4] } )
          : ( D_in[1] ?  {D_in[0], D_in[4], D_in[2] }   :
                         {D_in[4], D_in[2], D_in[0] } );

endmodule
```

Question 2

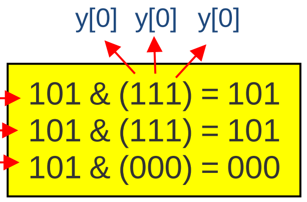
//Behavioral Style of Modeling (Using if statements to do the adding is another way to solve this!)

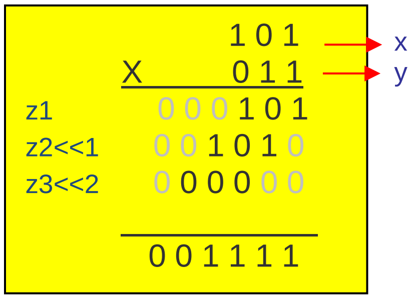
```
module mul_beh ( input [2:0] x, y, output reg [5:0] z);

reg [5:0] w1, w2, w3, temp;

    always @ ( x , y )
    begin
        w1 = {3'b000, x[2], x[1],x[0]};
        w2 = {2'b00, x[2], x[1], x[0], 1'b0};    //w1*2
        w3 = {1'b0, x[2], x[1], x[0], 2'b00};    //w1*4

        case (y)
            3'b000 : temp = 6'b000000;
            3'b001 : temp= w1;
            3'b010 : temp= w2;
            3'b011 : temp= w2+w1;
            3'b100 : temp= w3;
            3'b101 : temp= w3+w1;
            3'b110 : temp= w3+w2;
            default : temp= w3+w2+w1;
        endcase

        z<=temp;
    end

endmodule
```



// Dataflow Style of Modeling

```
module mul_d (input [2:0] x, y, output [5:0] z);

wire [5:0] z1, z2, z3;

    //Replication Operator! {n {___} }
    assign z1 = x & {3{y[0]}};
    assign z2 = x & {3{y[1]}};
    assign z3 = x & {3{y[2]}};

    assign z = z1 + (z2<<1) + (z3<<2);

endmodule
```





Question 3

```
module dff ( input D, CLK, CLR, S, output reg Q, output QB);

always @ ( negedge CLK, negedge CLR ) //Refer * Below for explanation

begin
if ( CLR == 0 )
    Q <= 0;
else
begin
    if ( S == 0 )
    Q <= 1;
   else
    Q <= D;
end
end

    assign QB = ~Q;

endmodule
```

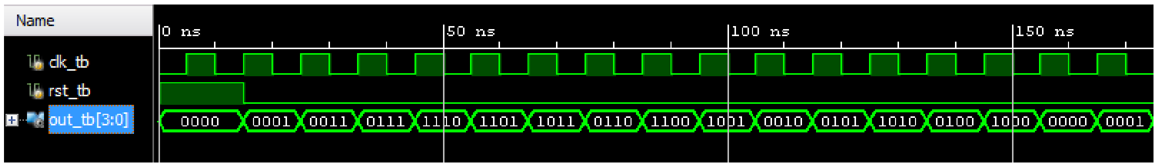
* In order to achieve an ***asynchronous reset***, the CLR signal must be included in the sensitivity list. Otherwise, the value of CLR would only be checked when there is a falling edge in the clock signal. This would implement a synchronous reset instead of asynchronous reset.

* When using posedge / negedge in the sensitivity lsit (eg. posedge clk), all signals need to specified with either posedge or negedge. This is to ensure that the design can be synthesized (mapped to an available device) and implemented on the FPGA. As such, “always@ (negedge CLK, CLR)” is not allowed.

Question 4

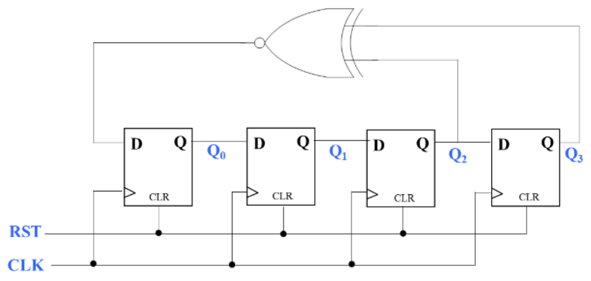
```
module lfsr ( input CLK, RST, output reg [3:0] Q);

reg [3:0] Q = 4'b0000;

always @ ( posedge CLK, posedge RST )

begin
    if ( RST )
        Q <= 0;
    else
    begin
        Q <= { Q[2:0], ~(Q[2]^Q[3]) } ; //Refer below for alternatives!
    end

end

endmodule


// Alternative 1 :
        Q <= { Q[2], Q[1], Q[0], ~(Q[2]^Q[3]) } ;

// Alternative 2 :
        Q[3] <= Q[2] ;
        Q[2] <= Q[1] ;
        Q[1] <= Q[0] ;
        Q[0] <= ~(Q[2]^Q[3]) ;

// Alternative 3 :
        Q[3:1] <= Q[2:0];
        Q[0] <= ~(Q[2]^Q[3]) ;
```

The bit sequence can be derived / simulated as follows :



```
Name           |0 ns                |50 ns                    |100 ns                   |150 ns
clk_tb
rst_tb
out_tb[3:0]    0000  0001 0011 0111 1110 1101 1011 0110 1100 1001 0010 0101 1010 0100 1000 0000 0001
```

By increasing the number of flip-flops used (eg. 12), the sequence that is generated is sufficiently random for many applications.



	Q0	Q1	Q2	Q3
Q2^Q3	0	0	0	0
1	1	0	0	0
1	1	1	0	0
1	1	1	1	0
0	0	1	1	1
1	1	0	1	1
1	1	1	0	1
0	0	1	1	0