Anika Haque, 20283

San Francisco Bay University

EE461 Verilog-HDL
Homework #6

Due day: 12/4/2024

Instruction:

a. Push answer sheets/source code to Github
b. Please follow the code style rule like programs on handout.
c. Overdue homework submission could not be accepted.
d. Takes academic honesty and integrity seriously (Zero Tolerance of Cheating & Plagiarism)

1. Write LFSR up & down count module to create 6-bits pseudo random number in Verilog module

```
`define kBitW 6
`define kBitS `kBitW-1:0

module upDownLFSR(
                    clk,
                    rst,
                    en_i,
                    upDown_i,       //upDown_i = 1 -> up; othrwise down
                    cnt_o,
                    bitStr_o        //Overflow bit
);
... ...;
Endmodule
```

MODULE

```
`timescale 1ns/1ps

module tb_upDownLFSR;

   // Inputs
   reg clk;
   reg rst;
   reg en_i;
   reg upDown_i;

   // Outputs
   wire [5:0] cnt_o;
   wire bitStr_o;

   // Instantiate the LFSR module
   upDownLFSR uut (
      .clk(clk),
```

```
        .rst(rst),
        .en_i(en_i),
        .upDown_i(upDown_i),
        .cnt_o(cnt_o),
        .bitStr_o(bitStr_o)
    );

    // Clock generation
    initial clk = 0;
    always #5 clk = ~clk;  // Adjusted to 10ns clock period

    // Test sequence
    initial begin
        // Initialize inputs
        rst = 1; en_i = 0; upDown_i = 1;
        #10 rst = 0;  // Deassert reset

        // Enable counting and count up
        en_i = 1; upDown_i = 1;
        #50;

        // Change direction to count down
        upDown_i = 0;
        #50;

        // Disable counting
        en_i = 0;
        #50;

        $finish;  // End simulation cleanly
    end

    // Monitor signals
    initial begin
        $monitor("Time: %0d, Count: %b, Overflow: %b", $time, cnt_o, bitStr_o);
    end

endmodule
```

TESTBENCH

```
`timescale 1ns/1ps

module tb_upDownLFSR;

    // Inputs
    reg clk;
    reg rst;
    reg en_i;
    reg upDown_i;

    // Outputs
    wire [5:0] cnt_o;
    wire bitStr_o;
```

```
    // Instantiate the LFSR module
    upDownLFSR uut (
        .clk(clk),
        .rst(rst),
        .en_i(en_i),
        .upDown_i(upDown_i),
        .cnt_o(cnt_o),
        .bitStr_o(bitStr_o)
    );

    // Clock generation
    initial clk = 0;
    always #5 clk = ~clk;  // Adjusted to 10ns clock period

    // Test sequence
    initial begin
        // Initialize inputs
        rst = 1; en_i = 0; upDown_i = 1;
        #10 rst = 0;  // Deassert reset

        // Enable counting and count up
        en_i = 1; upDown_i = 1;
        #50;

        // Change direction to count down
        upDown_i = 0;
        #50;

        // Disable counting
        en_i = 0;
        #50;

        $finish;  // End simulation cleanly
    end

    // Monitor signals
    initial begin
        $monitor("Time: %0d, Count: %b, Overflow: %b", $time, cnt_o, bitStr_o);
    end

endmodule
```

SIMULATION OUTPUT

testbench.sv

```
module tb_upDownLFSR;

    // Inputs
    reg clk;
    reg rst;
    reg en_i;
    reg upDown_i;

    // Outputs
    wire [5:0] cnt_o;
    wire bitStr_o;

    // Instantiate the LFSR module
    upDownLFSR uut (
        .clk(clk),
        .rst(rst),
        .en_i(en_i),
        .upDown_i(upDown_i),
        .cnt_o(cnt_o),
        .bitStr_o(bitStr_o)
    );

    // Clock generation
    initial clk = 0;
    always #5 clk = ~clk;  // Adjusted to 10ns clock period

    // Test sequence
    initial begin
        // Initialize inputs
        rst = 1; en_i = 0; upDown_i = 1;
        #10 rst = 0;  // Deassert reset

        // Enable counting and count up
        en_i = 1; upDown_i = 1;
        #50;

        // Change direction to count down
        upDown_i = 0;
        #50;

        // Disable counting
        en_i = 0;
        #50;

        $finish;  // End simulation cleanly
    end

    // Monitor signals
    initial begin
        $monitor("Time: %0d, Count: %b, Overflow: %b",
$time, cnt_o, bitStr_o);
    end

endmodule
```

design.sv

```
`define kBitW 6
`define kBitS `kBitW-1:0

module upDownLFSR(
    input clk,
    input rst,
    input en_i,       // Enable signal
    input upDown_i,   // Direction: 1 for up, 0 for down
    output reg [`kBitS] cnt_o,  // Current count value
    output reg bitStr_o         // Overflow bit
);

    reg [`kBitS] lfsr;  // LFSR register
    wire feedback;

    assign feedback = lfsr[5] ^ lfsr[3];  // Feedback taps for pseudo-randomness

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            lfsr <= 6'b1;    // Initialize LFSR to non-zero value
            cnt_o <= 6'b0;
            bitStr_o <= 1'b0;
        end else if (en_i) begin
            if (upDown_i) begin
                // Count up
                lfsr <= {lfsr[4:0], feedback};
            end else begin
                // Count down
                lfsr <= {feedback, lfsr[5:1]};
            end
            cnt_o <= lfsr;  // Output the current LFSR value
            bitStr_o <= lfsr[5];  // Overflow bit is the MSB
        end
    end
endmodule
```

Log | Share

```
[2024-12-16 12:16:24 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv  && unbuffer vvp a.out
Time: 0, Count: 000000, Overflow: 0
Time: 15, Count: 000001, Overflow: 0
Time: 25, Count: 000010, Overflow: 0
Time: 35, Count: 000100, Overflow: 0
Time: 45, Count: 001000, Overflow: 0
Time: 55, Count: 010001, Overflow: 0
Time: 65, Count: 100010, Overflow: 1
Time: 75, Count: 110001, Overflow: 1
Time: 85, Count: 111000, Overflow: 1
Time: 95, Count: 011100, Overflow: 0
Time: 105, Count: 101110, Overflow: 1
testbench.sv:47: $finish called at 160000 (1ps)
Done
```

2. Design frequency divider by 7 in Verilog module

MODULE

```
module frequencyDividerBy7(
    input clk,       // Input clock
    input rst,       // Reset signal
    output reg clk_out  // Divided clock output
);

    reg [2:0] counter;  // 3-bit counter to count up to 7

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            counter <= 3'b0;
            clk_out <= 0;
        end else begin
            if (counter == 3'b110) begin  // Counter reaches 6 (7 cycles)
                counter <= 3'b0;         // Reset counter
                clk_out <= ~clk_out;     // Toggle output clock
            end else begin
                counter <= counter + 1;  // Increment counter
            end
        end
    end
endmodule
```

TESTBENCH

```
`timescale 1ns/1ps

module tb_frequencyDividerBy7;

    // Inputs
    reg clk;
    reg rst;

    // Outputs
    wire clk_out;

    // Instantiate the Frequency Divider module
    frequencyDividerBy7 uut (
        .clk(clk),
        .rst(rst),
        .clk_out(clk_out)
    );
```

```
    // Clock generation
    initial clk = 0;
    always #5 clk = ~clk;  // 10ns clock period for the input clock

    // Test sequence
    initial begin
        // Initialize inputs
        rst = 1;  // Apply reset
        #10 rst = 0;  // Release reset after 10ns

        // Observe the output for multiple cycles
        #500;  // Allow simulation to run for 500ns

        $finish;  // End simulation
    end

    // Monitor signals
    initial begin
        $monitor("Time: %0d, Input Clock: %b, Output Clock: %b", $time, clk, clk_out);
    end

endmodule
```

SCEENSHOT

testbench.sv

```
`timescale 1ns/1ps

module tb_frequencyDividerBy7;

    // Inputs
    reg clk;
    reg rst;

    // Outputs
    wire clk_out;

    // Instantiate the Frequency Divider module
    frequencyDividerBy7 uut (
        .clk(clk),
        .rst(rst),
        .clk_out(clk_out)
    );

    // Clock generation
    initial clk = 0;
    always #5 clk = ~clk;  // 10ns clock period for the
```

design.sv

```
    input rst,       // Reset signal
    output reg clk_out  // Divided clock output
);

    reg [2:0] counter;  // 3-bit counter to count up to 7

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            counter <= 3'b0;
            clk_out <= 0;
        end else begin
            if (counter == 3'b110) begin  // Counter reaches 6 (7 cycles)
                counter <= 3'b0;          // Reset counter
                clk_out <= ~clk_out;      // Toggle output clock
            end else begin
                counter <= counter + 1;  // Increment counter
            end
        end
    end
endmodule
```

Log | Share

```
[2024-12-16 13:15:45 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv  && unbuffer vvp a.out
warning: Some design elements have no explicit time unit and/or
       : time precision. This may cause confusing timing results.
       : Affected design elements are:
       :   -- module frequencyDividerBy7 declared here: design.sv:1
Time: 0, Input Clock: 0, Output Clock: 0
Time: 5, Input Clock: 1, Output Clock: 0
Time: 10, Input Clock: 0, Output Clock: 0
Time: 15, Input Clock: 1, Output Clock: 0
Time: 20, Input Clock: 0, Output Clock: 0
Time: 25, Input Clock: 1, Output Clock: 0
Time: 30, Input Clock: 0, Output Clock: 0
Time: 35, Input Clock: 1, Output Clock: 0
Time: 40, Input Clock: 0, Output Clock: 0
Time: 45, Input Clock: 1, Output Clock: 0
Time: 50, Input Clock: 0, Output Clock: 0
Time: 55, Input Clock: 1, Output Clock: 0
Time: 60, Input Clock: 0, Output Clock: 0
Time: 65, Input Clock: 1, Output Clock: 0
Time: 70, Input Clock: 0, Output Clock: 0
Time: 75, Input Clock: 1, Output Clock: 1
Time: 80, Input Clock: 0, Output Clock: 1
Time: 85, Input Clock: 1, Output Clock: 1
Time: 90, Input Clock: 0, Output Clock: 1
Time: 95, Input Clock: 1, Output Clock: 1
Time: 100, Input Clock: 0, Output Clock: 1
Time: 105, Input Clock: 1, Output Clock: 1
Time: 110, Input Clock: 0, Output Clock: 1
Time: 115, Input Clock: 1, Output Clock: 1
Time: 120, Input Clock: 0, Output Clock: 1
Time: 125, Input Clock: 1, Output Clock: 1
Time: 130, Input Clock: 0, Output Clock: 1
Time: 135, Input Clock: 1, Output Clock: 1
Time: 140, Input Clock: 0, Output Clock: 1
Time: 145, Input Clock: 1, Output Clock: 0
Time: 150, Input Clock: 0, Output Clock: 0
Time: 155, Input Clock: 1, Output Clock: 0
Time: 160, Input Clock: 0, Output Clock: 0
Time: 165, Input Clock: 1, Output Clock: 0
```

(Output too big to put in one screenshot)

SCREENSHOT - END OF OUTPUT

testbench.sv

```
`timescale 1ns/1ps

module tb_frequencyDividerBy7;

    // Inputs
    reg clk;
    reg rst;

    // Outputs
    wire clk_out;

    // Instantiate the Frequency Divider module
    frequencyDividerBy7 uut (
        .clk(clk),
        .rst(rst),
        .clk_out(clk_out)
    );

    // Clock generation
    initial clk = 0;
    always #5 clk = ~clk;  // 10ns clock period for the input clock

    // Test sequence
    initial begin
        // Initialize inputs
        rst = 1;  // Apply reset
        #10 rst = 0;  // Release reset after 10ns
```

design.sv

```
    input rst,       // Reset signal
    output reg clk_out  // Divided clock output
);

    reg [2:0] counter;  // 3-bit counter to count up to 7

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            counter <= 3'b0;
            clk_out <= 0;
        end else begin
            if (counter == 3'b110) begin  // Counter reaches 6 (7 cycles)
                counter <= 3'b0;          // Reset counter
                clk_out <= ~clk_out;      // Toggle output clock
            end else begin
                counter <= counter + 1;  // Increment counter
            end
        end
    end
endmodule
```

Log | Share

```
Time: 350, Input Clock: 0, Output Clock: 0
Time: 355, Input Clock: 1, Output Clock: 1
Time: 360, Input Clock: 0, Output Clock: 1
Time: 365, Input Clock: 1, Output Clock: 1
Time: 370, Input Clock: 0, Output Clock: 1
Time: 375, Input Clock: 1, Output Clock: 1
Time: 380, Input Clock: 0, Output Clock: 1
Time: 385, Input Clock: 1, Output Clock: 1
Time: 390, Input Clock: 0, Output Clock: 1
Time: 395, Input Clock: 1, Output Clock: 1
Time: 400, Input Clock: 0, Output Clock: 1
Time: 405, Input Clock: 1, Output Clock: 1
Time: 410, Input Clock: 0, Output Clock: 1
Time: 415, Input Clock: 1, Output Clock: 1
Time: 420, Input Clock: 0, Output Clock: 1
Time: 425, Input Clock: 1, Output Clock: 0
Time: 430, Input Clock: 0, Output Clock: 0
Time: 435, Input Clock: 1, Output Clock: 0
Time: 440, Input Clock: 0, Output Clock: 0
Time: 445, Input Clock: 1, Output Clock: 0
Time: 450, Input Clock: 0, Output Clock: 0
Time: 455, Input Clock: 1, Output Clock: 0
Time: 460, Input Clock: 0, Output Clock: 0
Time: 465, Input Clock: 1, Output Clock: 0
Time: 470, Input Clock: 0, Output Clock: 0
Time: 475, Input Clock: 1, Output Clock: 0
Time: 480, Input Clock: 0, Output Clock: 0
Time: 485, Input Clock: 1, Output Clock: 0
Time: 490, Input Clock: 0, Output Clock: 0
Time: 495, Input Clock: 1, Output Clock: 1
Time: 500, Input Clock: 0, Output Clock: 1
Time: 505, Input Clock: 1, Output Clock: 1
testbench.sv:32: $finish called at 510000 (1ps)
Time: 510, Input Clock: 0, Output Clock: 1
Done
```

3. Design CRC-4 decoder in Verilog module showing the hardware circuit

MODULE

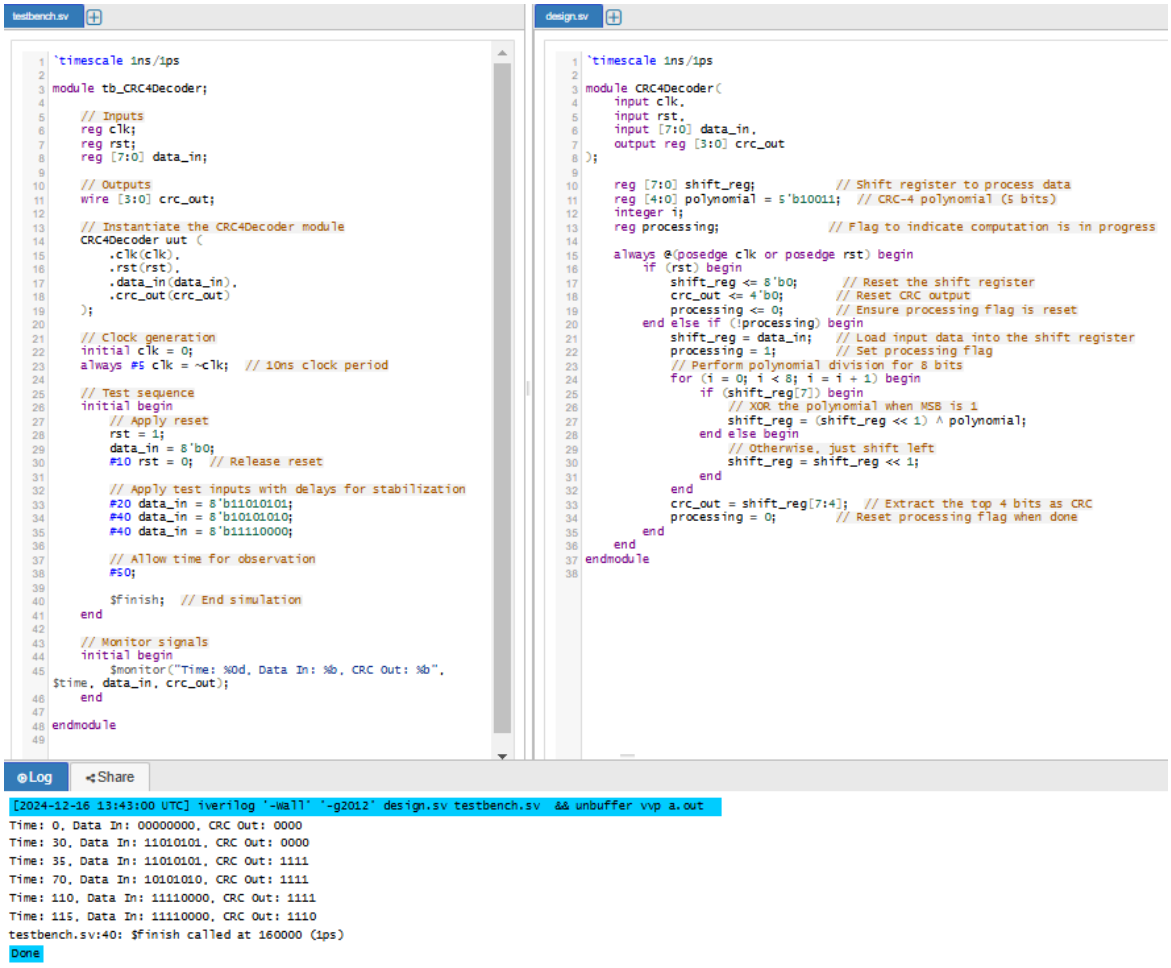
```
`timescale 1ns/1ps

module CRC4Decoder(
    input clk,
    input rst,
    input [7:0] data_in,
    output reg [3:0] crc_out
);

    reg [7:0] shift_reg;           // Shift register to process data
    reg [4:0] polynomial = 5'b10011;  // CRC-4 polynomial (5 bits)
    integer i;
    reg processing;              // Flag to indicate computation is in progress

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            shift_reg <= 8'b0;      // Reset the shift register
            crc_out <= 4'b0;       // Reset CRC output
            processing <= 0;       // Ensure processing flag is reset
        end else if (!processing) begin
            shift_reg = data_in;   // Load input data into the shift register
            processing = 1;        // Set processing flag
            // Perform polynomial division for 8 bits
            for (i = 0; i < 8; i = i + 1) begin
                if (shift_reg[7]) begin
                    // XOR the polynomial when MSB is 1
                    shift_reg = (shift_reg << 1) ^ polynomial;
                end else begin
                    // Otherwise, just shift left
                    shift_reg = shift_reg << 1;
                end
            end
            crc_out = shift_reg[7:4];  // Extract the top 4 bits as CRC
            processing = 0;        // Reset processing flag when done
        end
    end
endmodule
```

TESTBENCH

```
`timescale 1ns/1ps

module tb_CRC4Decoder;
```

```
    // Inputs
    reg clk;
    reg rst;
    reg [7:0] data_in;

    // Outputs
    wire [3:0] crc_out;

    // Instantiate the CRC4Decoder module
    CRC4Decoder uut (
        .clk(clk),
        .rst(rst),
        .data_in(data_in),
        .crc_out(crc_out)
    );

    // Clock generation
    initial clk = 0;
    always #5 clk = ~clk;  // 10ns clock period

    // Test sequence
    initial begin
        // Apply reset
        rst = 1;
        data_in = 8'b0;
        #10 rst = 0;  // Release reset

        // Apply test inputs with delays for stabilization
        #20 data_in = 8'b11010101;
        #40 data_in = 8'b10101010;
        #40 data_in = 8'b11110000;

        // Allow time for observation
        #50;

        $finish;  // End simulation
    end

    // Monitor signals
    initial begin
        $monitor("Time: %0d, Data In: %b, CRC Out: %b", $time, data_in, crc_out);
    end

endmodule
```

SCREENSHOT



testbench.sv

```
`timescale 1ns/1ps

module tb_CRC4Decoder;

    // Inputs
    reg clk;
    reg rst;
    reg [7:0] data_in;

    // Outputs
    wire [3:0] crc_out;

    // Instantiate the CRC4Decoder module
    CRC4Decoder uut (
        .clk(clk),
        .rst(rst),
        .data_in(data_in),
        .crc_out(crc_out)
    );

    // Clock generation
    initial clk = 0;
    always #5 clk = ~clk;  // 10ns clock period

    // Test sequence
    initial begin
        // Apply reset
        rst = 1;
        data_in = 8'b0;
        #10 rst = 0;  // Release reset

        // Apply test inputs with delays for stabilization
        #20 data_in = 8'b11010101;
        #40 data_in = 8'b10101010;
        #40 data_in = 8'b11110000;

        // Allow time for observation
        #50;

        $finish;  // End simulation
    end

    // Monitor signals
    initial begin
        $monitor("Time: %0d, Data In: %b, CRC Out: %b",
$time, data_in, crc_out);
    end

endmodule
```

design.sv

```
`timescale 1ns/1ps

module CRC4Decoder(
    input clk,
    input rst,
    input [7:0] data_in,
    output reg [3:0] crc_out
);

    reg [7:0] shift_reg;           // Shift register to process data
    reg [4:0] polynomial = 5'b10011;  // CRC-4 polynomial (5 bits)
    integer i;
    reg processing;               // Flag to indicate computation is in progress

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            shift_reg <= 8'b0;       // Reset the shift register
            crc_out <= 4'b0;         // Reset CRC output
            processing <= 0;         // Ensure processing flag is reset
        end else if (!processing) begin
            shift_reg = data_in;     // Load input data into the shift register
            processing = 1;          // Set processing flag
            // Perform polynomial division for 8 bits
            for (i = 0; i < 8; i = i + 1) begin
                if (shift_reg[7]) begin
                    // XOR the polynomial when MSB is 1
                    shift_reg = (shift_reg << 1) ^ polynomial;
                end else begin
                    // Otherwise, just shift left
                    shift_reg = shift_reg << 1;
                end
            end
            crc_out = shift_reg[7:4];  // Extract the top 4 bits as CRC
            processing = 0;          // Reset processing flag when done
        end
    end
endmodule
```

Log | Share

```
[2024-12-16 13:43:00 UTC] iverilog '-Wall' '-g2012' design.sv testbench.sv  && unbuffer vvp a.out
Time: 0, Data In: 00000000, CRC Out: 0000
Time: 30, Data In: 11010101, CRC Out: 0000
Time: 35, Data In: 11010101, CRC Out: 1111
Time: 70, Data In: 10101010, CRC Out: 1111
Time: 110, Data In: 11110000, CRC Out: 1111
Time: 115, Data In: 11110000, CRC Out: 1110
testbench.sv:40: $finish called at 160000 (1ps)
Done
```