Question 1

A Simple FIFO Design

This report presents my implementation and analysis of an 8-bit wide, 16-depth FIFO (First-In-First-Out) memory module.

```verilog
module FIFO (
    input clk, rst,
    input [7:0] wr_data,
    input wr_en, rd_en,
    output reg [7:0] rd_data,
    output wire empty, full
);

    reg [7:0] mem [15:0]; // 16 x 8-bit memory
    reg [4:0] pointer_wr; // Track number of stored items

    assign empty = (pointer_wr==0);
    assign full = (pointer_wr==16);

    always @(posedge clk or posedge rst) begin
        if (rst) begin
            pointer_wr <= 0;
        end else begin
            if (wr_en && !full) begin
                mem[pointer_wr] <= wr_data;
                pointer_wr <= pointer_wr + 1;
            end
            if (rd_en && !empty) begin
                rd_data <= mem[0];
                for (integer i = 0; i < 15; i = i + 1) begin
                    mem[i] <= mem[i + 1];
                end
                pointer_wr <= pointer_wr - 1;
            end
        end
    end
endmodule
```

- **Write Operation:** Data is written into mem at the location indicated by pointer_wr, and the pointer increments. Writing stops when full is asserted.
- **Read Operation:** Data is read from mem[0], and the array shifts left to fill the gap. The pointer decrements accordingly.

Testbench Analysis I developed a testbench to validate the FIFO's functionality:

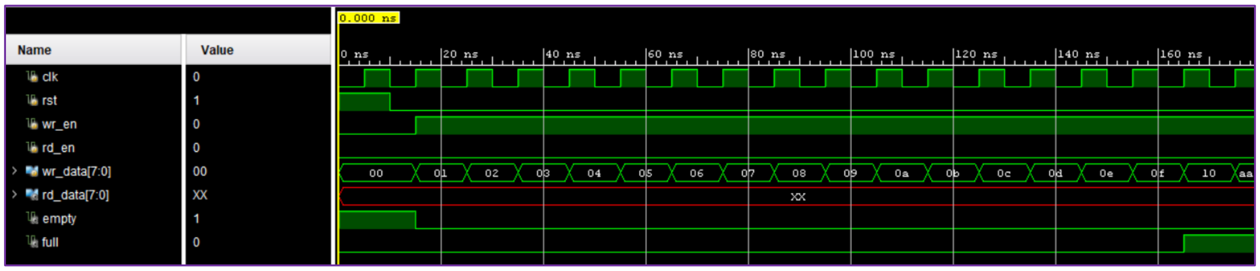
- **Reset Phase:** Ensures all signals are initialized and FIFO is empty.
- **Write Phase:** Writes 16 values into the FIFO. The full flag asserts when the memory reaches capacity.
- **Overflow Test:** Attempts to write additional data when full is active, confirming no data overwrite occurs.
- **Read Phase:** Reads all 16 values from the FIFO. The empty flag asserts when the FIFO is fully drained.
- **Underflow Test:** Attempts to read data when empty is active, confirming no invalid data is presented.

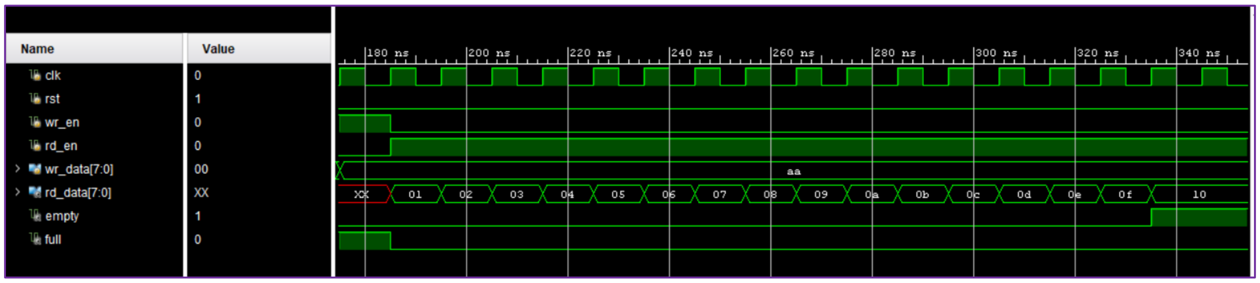
```
module tb_FIFO;
    reg clk, rst, wr_en, rd_en;
    reg [7:0] wr_data;
    wire [7:0] rd_data;
    wire empty, full;

    FIFO uut (.clk(clk), .rst(rst), .wr_data(wr_data), .wr_en(wr_en),
.rd_en(rd_en), .rd_data(rd_data), .empty(empty), .full(full));

    always #5 clk = ~clk; // 100MHz clock simulation

    initial begin
        clk = 0; rst = 1;
        wr_en = 0; rd_en = 0; wr_data = 8'b0;
        #10 rst = 0;
        $display("[INFO] Reset complete");

        // Writing data to FIFO
        repeat (16) begin
            @(posedge clk);
            wr_en = 1; wr_data = wr_data + 1;
            $display("[WRITE] Data Written: %h | Full: %b", wr_data, full);
        end

        // Attempt to write when full
        @(posedge clk);
        wr_en = 1; wr_data = 8'hAA;
        $display("[WARNING] Attempted to write when FIFO full. Data: %h | Full:
%b", wr_data, full);

        // Start reading data from FIFO
        repeat (16) begin
            @(posedge clk);
            wr_en = 0; rd_en = 1;
            $display("[READ] Data Read: %h | Empty: %b", rd_data, empty);
        end

        // Attempt to read when empty
        @(posedge clk);
        rd_en = 1;
        $display("[WARNING] Attempted to read when FIFO empty. Empty: %b", 
empty);

        #20 $stop;
    end
endmodule
```

Observations :The FIFO correctly asserts full and empty flags and handles overflow/underflow conditions.

```
[INFO] Reset complete

[WRITE] Data Written: 01 | Full: 0

[WRITE] Data Written: 02 | Full: 0

[WRITE] Data Written: 03 | Full: 0

[WRITE] Data Written: 04 | Full: 0

[WRITE] Data Written: 05 | Full: 0

[WRITE] Data Written: 06 | Full: 0

[WRITE] Data Written: 07 | Full: 0

[WRITE] Data Written: 08 | Full: 0

[WRITE] Data Written: 09 | Full: 0

[WRITE] Data Written: 0a | Full: 0

[WRITE] Data Written: 0b | Full: 0

[WRITE] Data Written: 0c | Full: 0

[WRITE] Data Written: 0d | Full: 0

[WRITE] Data Written: 0e | Full: 0
```

```
[WRITE] Data Written: 0f | Full: 0

[WRITE] Data Written: 10 | Full: 0

[WARNING] Attempted to write when FIFO full. Data: aa | Full: 1

[READ] Data Read: xx | Empty: 0

[READ] Data Read: 01 | Empty: 0

[READ] Data Read: 02 | Empty: 0

[READ] Data Read: 03 | Empty: 0

[READ] Data Read: 04 | Empty: 0

[READ] Data Read: 05 | Empty: 0

[READ] Data Read: 06 | Empty: 0

[READ] Data Read: 07 | Empty: 0

[READ] Data Read: 08 | Empty: 0

[READ] Data Read: 09 | Empty: 0

[READ] Data Read: 0a | Empty: 0

[READ] Data Read: 0b | Empty: 0

[READ] Data Read: 0c | Empty: 0

[READ] Data Read: 0d | Empty: 0

[READ] Data Read: 0e | Empty: 0

[READ] Data Read: 0f | Empty: 0

[READ] Data Read: 10 | Empty: 1

[WARNING] Attempted to read when FIFO empty. Empty: 1
```





My FIFO design meets the required functionality and behaves as expected under typical conditions.

Question 2

Using MATLAB to check the accuracy of implemented structures

Part 1

in this part of the assignment, I used MATLAB to generate sine and cosine wave data, modeled in a fixed-point format (1,16,14) to ensure compatibility with Verilog's 16-bit data structure. Here is the code:

```
%% Initial Commands
clc;
clear;


% Generating a Sine Wave with length = 1024
% Defining time
t = 0:pi/512:2*pi;
time = t(1:1024);


% Defining Sine and Cosine
sinWave = sin(time);
cosWave = cos(time);


% Fixed-point process for Sine Wave
sinFixed_temp = fi(sinWave,1,16,14);
sinFixed = sinFixed_temp * (2^14);
sin_file = dec2bin(sinFixed);


% Fixed-point process for Cosine Wave
cosFixed_temp = fi(cosWave,1,16,14);
cosFixed = cosFixed_temp * (2^14);
cos_file = dec2bin(cosFixed);


% Writing data into textfile
writematrix(sin_file);
writematrix(cos_file);
```

and here is the initial block code in Verilog:

```verilog
module sin_cos_gen (
    input clk,
    input rst,
    input wire [1:0] frequency_sel,
    output reg [15:0] output_sin,
    output reg [15:0] output_cos,
    output reg done
);
reg [15:0] sin_data [0:1023];
reg [15:0] cos_data [0:1023];

initial begin
    $readmemb("sin_file.txt", sin_data);
    $readmemb("cos_file.txt", cos_data);
end
```

part2

In this part, I designed the module sin_cos_gen that generates sine and cosine waveforms at different frequencies. The frequency of the output waveform is controlled by a 2-bit input (frequency_sel), which selects between four frequency multipliers: 1X, 2X, 4X, and 8X. The sine and cosine values are stored in two arrays, initialized from external files (sin_file.txt and cos_file.txt). The module outputs the sine and cosine values sequentially, adjusting the step size based on the selected frequency.

```verilog
module sin_cos_gen (
    input clk,
    input rst,
    input wire [1:0] frequency_sel,
    output reg [15:0] output_sin,
    output reg [15:0] output_cos,
    output reg done
);

/*
    Different cases for the input are as follows:
    frequency_sel = 0  --->   1X frequency_seluency
    frequency_sel = 1  --->   2X frequency_seluency
    frequency_sel = 2  --->   4X frequency_seluency
    frequency_sel = 3  --->   8X frequency_seluency
*/

integer i,step,counter;
```

```verilog
reg [15:0] sin_data [0:1023];
reg [15:0] cos_data [0:1023];

initial begin
    $readmemb("sin_file.txt", sin_data);
    $readmemb("cos_file.txt", cos_data);
end

always @(posedge clk) begin
    if (rst) begin
        i <= 0;
        counter <= 0;
        output_sin <= 0;
        output_cos <= 0;
        done <= 0;
        case (frequency_sel)
            2'b00: step <= 1;
            2'b01: step <= 2;
            2'b10: step <= 4;
            2'b11: step <= 8;
        endcase
    end
    else begin
        if (counter == 1024) begin
            done <= 1;
        end
        else begin
            output_sin <= sin_data[i];
            output_cos <= cos_data[i];
            counter <= counter + 1;
            if (i + step < 1024) i <= i + step;
            else i <= i + step - 1024;
        end
    end
end
endmodule
```

The sin_cos_gen module efficiently generates sine and cosine waveforms at varying frequencies using a simple step-size adjustment mechanism. This design demonstrates the use of Verilog for digital signal processing and waveform generation.

Part3

This part involved designing a Verilog testbench to generate sine and cosine waveforms at 2X and 8X frequencies using the sin_cos_gen module. The outputs were saved to .txt files and analyzed in MATLAB using FFT to verify the accuracy of the generated frequencies.

Verilog Testbench:
The testbench generated sine and cosine waveforms for two frequency settings: 2X and 8X.
Outputs were saved to 2X_sin_output.txt, 2X_cos_output.txt, 8X_sin_output.txt, and 8X_cos_output.txt.

```
`timescale 1ns/1ns

module sin_cos_gen_tb;

reg [1:0] frequency_sel;
reg clk;
reg rst;
wire [15:0] output_sin;
wire [15:0] output_cos;
wire done;


 sin_cos_gen dut (
    .frequency_sel(frequency_sel),
    .rst(rst),
    .clk(clk),
    .output_sin(output_sin),
    .output_cos(output_cos),
    .done(done)
);

integer SinFile;
integer CosFile;

always @(clk) begin
    #5 clk <= ~clk;
end

initial begin
    clk = 1;

    // Test case 1: frequency_sel = 2X
    rst = 1;
    frequency_sel = 2'b01;
    #10;
    rst = 0;
    SinFile = $fopen("2X_sin_output.txt", "w");
    CosFile = $fopen("2X_cos_output.txt", "w"); #10;

    while (dut.done == 0) begin
        $fdisplay(SinFile, "%b", output_sin);
        $fdisplay(CosFile, "%b", output_cos);
        #10;
    end

    $fclose(SinFile);
    $fclose(CosFile);

    #10;

    // Test case 2: frequency_sel = 8X
    rst = 1;
    frequency_sel = 2'b11;
    #10;
    rst = 0;
    SinFile = $fopen("8X_sin_output.txt", "w");
    CosFile = $fopen("8X_cos_output.txt", "w"); #10;

    while (dut.done == 0) begin
        $fdisplay(SinFile, "%b", output_sin);
        $fdisplay(CosFile, "%b", output_cos);
        #10;
    end

    $fclose(SinFile);
    $fclose(CosFile);

    $stop;
end

endmodule
```

MATLAB Analysis:

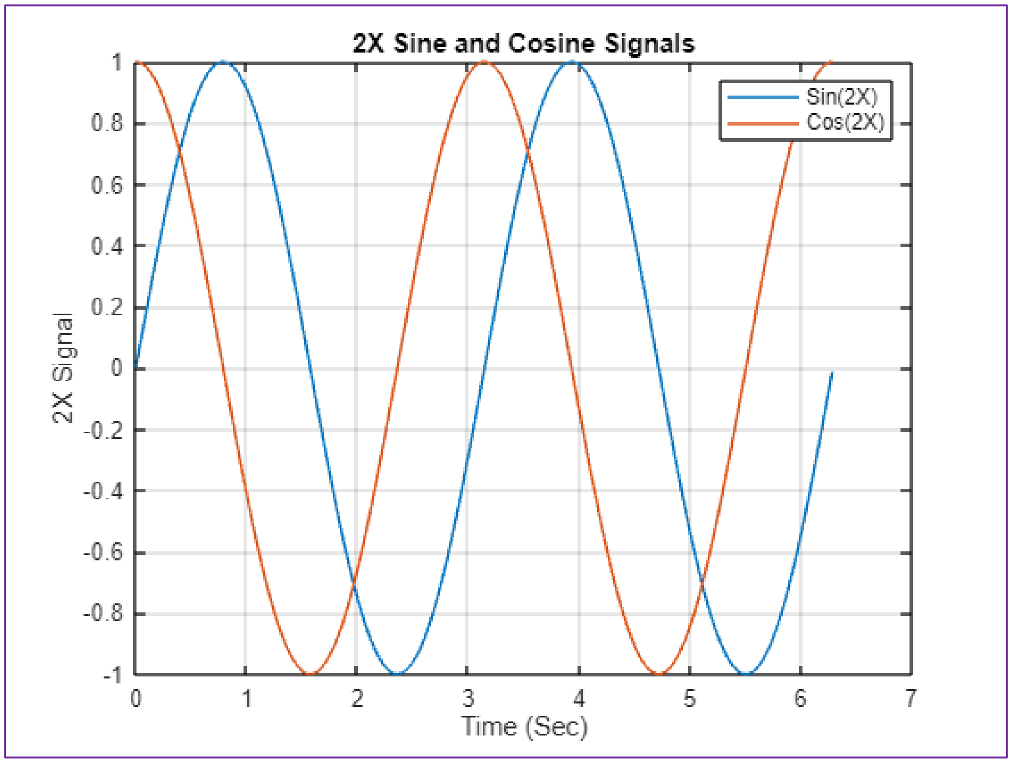
The saved waveforms were read and combined into complex signals (cos + i*sin).

FFT was applied to analyze the frequency spectrum.

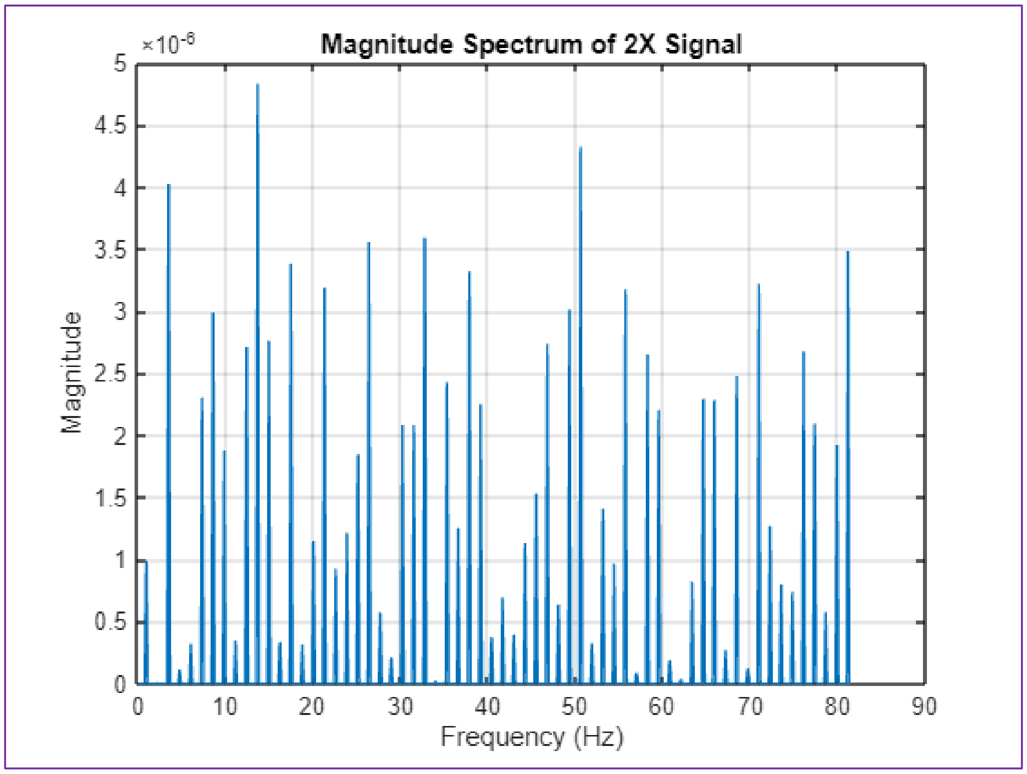
The sampling frequency was set and the frequency components were plotted.

```
%% Initialization
clc; clear;
```

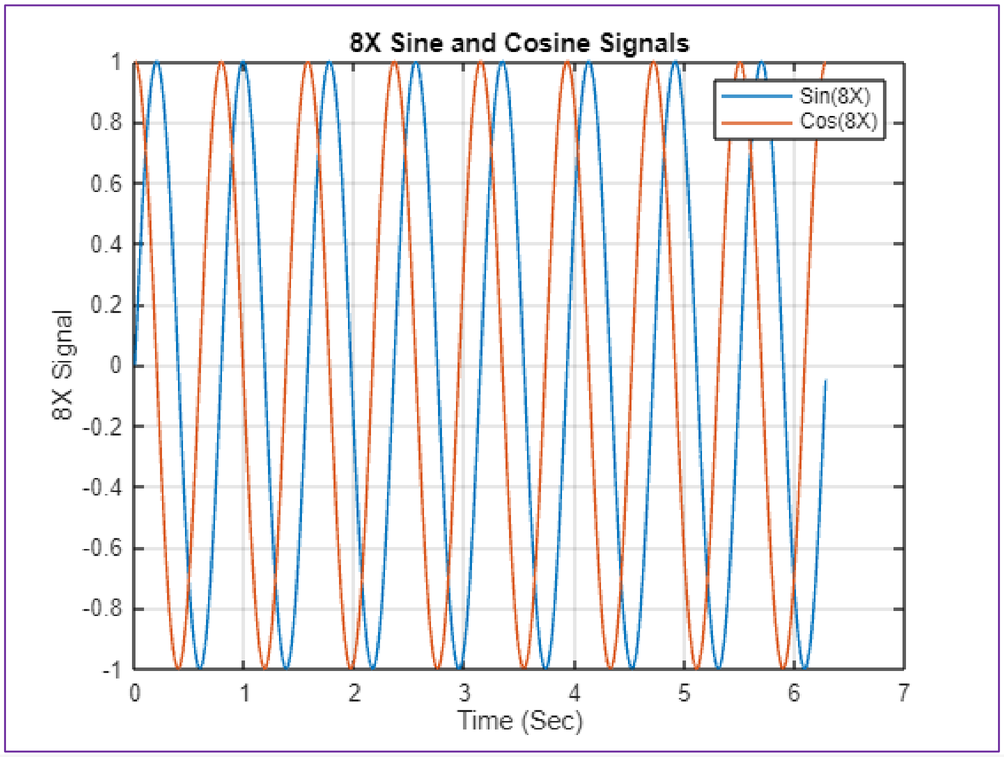
```
%% Parameters
L = 1024; % Signal length
Fs = 1024 / (2 * pi); % Sampling frequency
Freq = Fs * (0:(L/2)) / L; % Frequency range for plotting
time = linspace(0, 2 * pi, L); % Time vector

%% 2X Frequency Components
Sin2X = ReadBinaryFile('2X_sin_output.txt');
Cos2X = ReadBinaryFile('2X_cos_output.txt');

Func2 = Sin2X + 1i * Cos2X;
Func2_FFT = fft(Func2);

figure;
plot(time, Sin2X, time, Cos2X);
xlabel("Time (Sec)");
ylabel("2X Signal");
title("2X Sine and Cosine Signals");
legend("Sin(2X)", "Cos(2X)");
grid on;
```



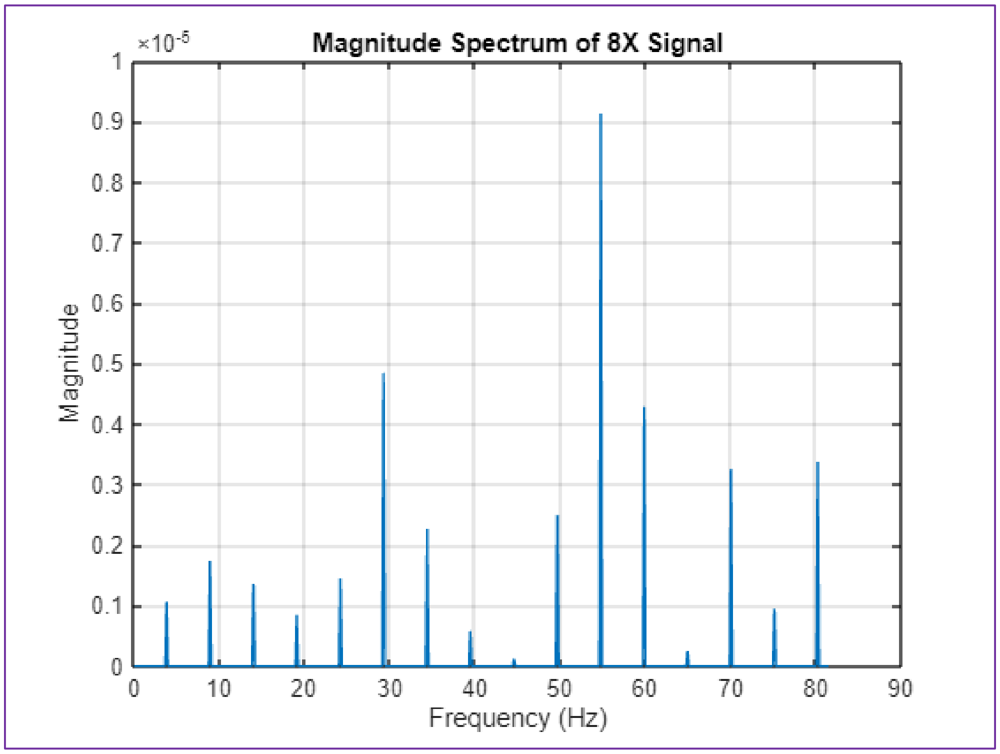
```
figure;
plot(Freq, abs(Func2_FFT(1:L/2+1)) / L);
xlabel('Frequency (Hz)');
ylabel('Magnitude');
title('Magnitude Spectrum of 2X Signal');
grid on;
```



```
%% 8X Frequency Components
Sin8X = ReadBinaryFile('8X_sin_output.txt');
Cos8X = ReadBinaryFile('8X_cos_output.txt');

Func8 = Sin8X + 1i * Cos8X;
Func8_FFT = fft(Func8);

figure;
plot(time, Sin8X, time, Cos8X);
xlabel("Time (Sec)");
ylabel("8X Signal");
title("8X Sine and Cosine Signals");
legend("Sin(8X)", "Cos(8X)");
grid on;
```



```
figure;
plot(Freq, abs(Func8_FFT(1:L/2+1)) / L);
xlabel('Frequency (Hz)');
ylabel('Magnitude');
title('Magnitude Spectrum of 8X Signal');
grid on;
```



```
%% Function for Reading Binary Files
function Out = ReadBinaryFile(filename)
    fid = fopen(filename, 'r');
    binary_data = textscan(fid, '%s');
    fclose(fid);

    decimal_data = bin2dec(binary_data{1});
    negative_indices = decimal_data > 2^(16-1) - 1;
    decimal_data(negative_indices) = decimal_data(negative_indices) - 2^16;

    Out = decimal_data / 2^14; % Adjusting for fractional bits
end
```

Results

2X Frequency:

The time-domain plots showed sine and cosine waveforms at 2X frequency.

The FFT spectrum confirmed the presence of the expected frequency component.

8X Frequency:

The time-domain plots showed sine and cosine waveforms at 8X frequency.

The FFT spectrum verified the accuracy of the 8X frequency component.

Although the Magnitude Spectrums Are a little bit confusing, but the main fundamental frequency is observed. Moreover, the plotted waveforms can simply validate the functionality of Verilog Code and MATLAB structures as they resulted in 4X and 8X frequencies.

NOTICE:

I realized too late that we should have used mem format instead of txt and that txt wouldn’t be synthesized and there really wasn't time to edit it, so I appreciate your patience.

Question 3

Introduction to the DSP-48 structure used in Xilinx chips

Part1

The 48-bit DSP (Digital Signal Processing) slice in Xilinx FPGAs is a specialized hardware block optimized for high-performance arithmetic operations, especially for signal processing, machine learning, and other computationally intensive tasks.

Key Components of the Xilinx 48-bit DSP Slice

The 48-bit DSP slice (commonly called DSP48E1 or DSP48E2 in newer Xilinx devices) is composed of several functional units:

1. Multiplier (M)
   - Performs 25-bit × 18-bit multiplication, producing a 43-bit product.
   - Efficient for implementing FIR filters, matrix multiplications, and other DSP algorithms.
2. Adder/Accumulator (P)
   - A 48-bit accumulator that supports operations like addition, subtraction, and accumulation.
   - It can directly add the multiplier output or other values from internal registers.
3. Pre-Adder
   - A two-input adder located before the multiplier, ideal for symmetric FIR filter structures.
4. Logic Unit
   - Supports AND, OR, XOR, and other bitwise operations, enhancing flexibility.
5. Pipeline Registers
   - Various pipeline stages are available to maximize performance by improving timing closure and reducing delays in high-speed designs.
6. Control Logic
   - Supports flexible data flow with features like opmode settings, carry control, and pattern detection for dynamic behavior.
7. Dynamic Mode Control
   - The DSP48 slice can be dynamically configured for different operations, such as multiply-accumulate (MAC), multiply-add (MAD), and more.

Applications

- Digital Filters
- Fast Fourier Transform (FFT)
- Image and Video Processing
- Control Systems
- Machine Learning Accelerators

Part2

To design an **18-bit complex multiplier** using DSP48 slices in Xilinx FPGAs, we can efficiently leverage the **25x18 multiplier** inside each DSP slice.

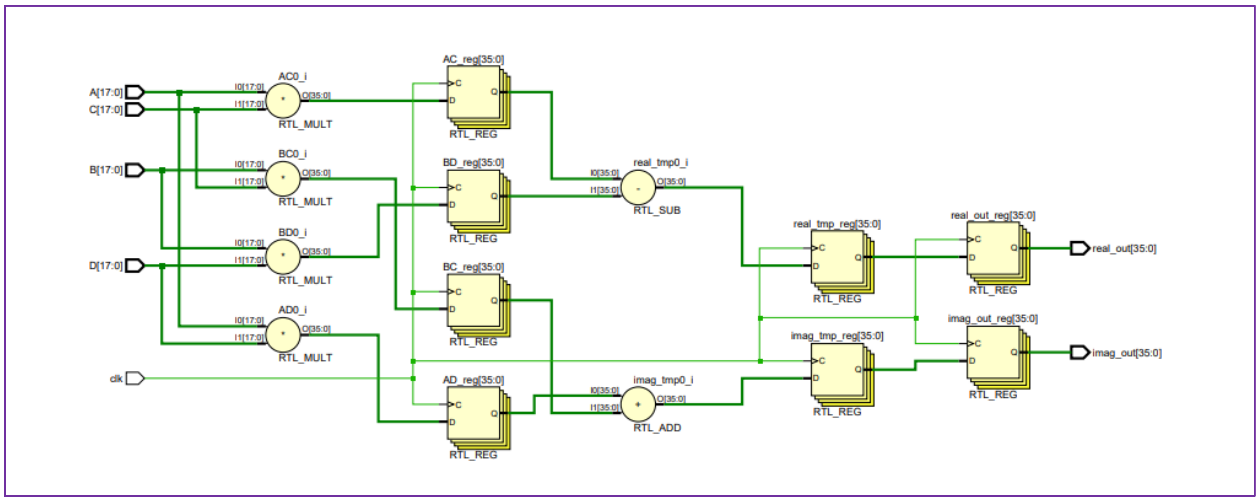
Complex Multiplier Overview

Given two complex numbers X=a+jb and Y=c+jd → X×Y=ac−bd+j(ad+bc)

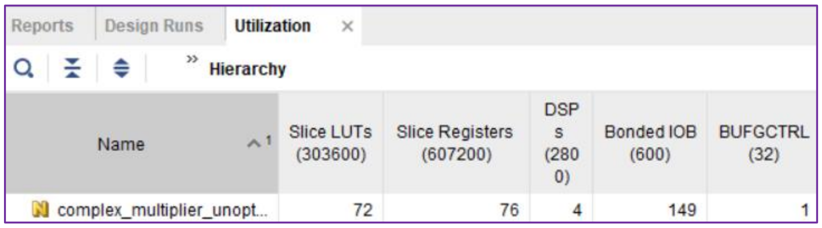
Mapping to DSP48

- Each DSP slice can perform a **25x18 multiplication**, so we will need multiple DSPs to handle the full calculation.
- Efficient pipelining can enhance timing performance during synthesis.

```verilog
module complex_multiplier (
    input  clk,
    input  [17:0] A, B, C, D,
    output reg [35:0] real_out, imag_out
);

    // First stage: Multiplication
    reg [35:0] AC, BD, AD, BC;

    always@(posedge clk) begin
        AC <= A * C;  // Real part term 1
        BD <= B * D;  // Real part term 2
        AD <= A * D;  // Imaginary part term 1
        BC <= B * C;  // Imaginary part term 2
    end

    // Second stage: Addition/Subtraction
    reg [35:0] real_tmp, imag_tmp;

    always@(posedge clk) begin
        real_tmp <= AC - BD;
        imag_tmp <= AD + BC;
    end

    // Final output
    always@(posedge clk) begin
        real_out <= real_tmp;
        imag_out <= imag_tmp;
    end
endmodule
```



RTL schematic



Reports | Design Runs | Utilization

Hierarchy

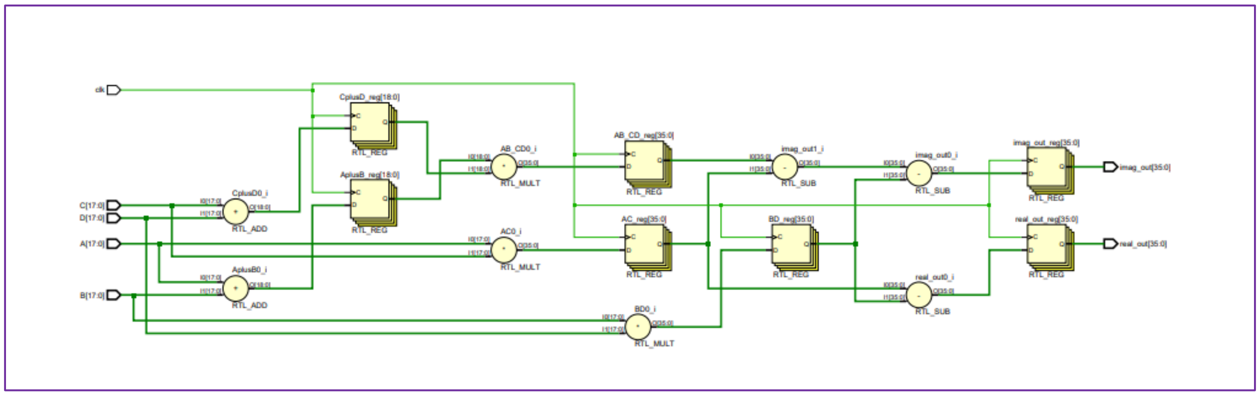
Name	Slice LUTs (303600)	Slice Registers (607200)	DSPs (2800)	Bonded IOB (600)	BUFGCTRL (32)
complex_multiplier_unopt...	72	76	4	149	1

Utilization report

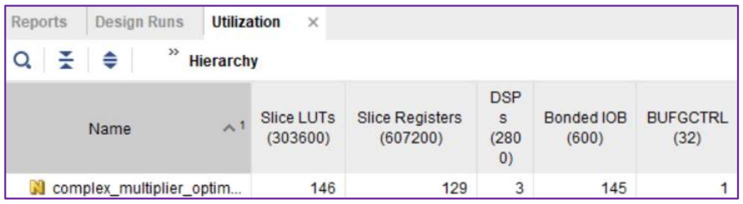
Optimized Complex Multiplication Formula

Given: X=a+jb and Y=c+jd Instead of calculating X×Y=ac−bd+j(ad+bc) We use the identity X×Y=ac−bd+j((a+b)(c+d)−ac−bd)

```
`timescale 1ns / 1ps

module complex_multiplier_optimized (
    input clk,
    input [17:0] A, B, C, D,
    output reg [35:0] real_out, imag_out
);

    // Intermediate values
    reg [35:0] AC, BD, AplusB, CplusD, AB_CD;

    // First stage: Compute the key multiplications
    always @(posedge clk) begin
        AC      <= A * C;            // 1st DSP
        BD      <= B * D;             // 2nd DSP
        AplusB  <= A + B;            // Addition in logic
        CplusD  <= C + D;            // Addition in logic
    end

    // Second stage: Combined multiplication
    always @(posedge clk) begin
        AB_CD <= AplusB * CplusD;     // 3rd DSP
    end

    // Final stage: Compute the real and imaginary outputs
    always @(posedge clk) begin
        real_out <= AC - BD;
        imag_out <= AB_CD - AC - BD;
    end

endmodule
```



RTL schematic



Name	Slice LUTs (303600)	Slice Registers (607200)	DSPs (2800)	Bonded IOB (600)	BUFGCTRL (32)
complex_multiplier_optim...	146	129	3	145	1

Utilization report

DSP Utilization Analysis

3 DSP48 slices used instead of 4 (a 25% reduction)
Efficient use of pipeline stages for improved timing
Minimal LUT usage since additions are handled efficiently in logic

Key Optimization Techniques Applied

Mathematical Identity: Reduced the number of multiplications.
Pipeline Stages: Ensured proper timing closure by introducing registers between stages.
Efficient Resource Use: Leveraged DSP48 slices specifically for multiplications while performing additions in logic.

Part3

For 24-bit inputs, the DSP usage will increase further due to the size limitation of the DSP48 slices in Xilinx FPGAs. Each DSP48 slice can handle up to an 18x25-bit multiplication in a single unit.

Understanding DSP48 Slice Limits

- A DSP48E1 or DSP48E2 slice in Xilinx FPGAs can efficiently perform a 25-bit × 18-bit multiplication.
- If either operand exceeds these limits, Vivado will automatically use additional DSP slices or LUT logic to manage the overflow.

Impact of Increasing from 18-bit to 24-bit Inputs

1. Original Design (18-bit Inputs)

- Each multiplication fits perfectly within the 25×18 multiplier inside one DSP48 slice.
- Using the optimized formula with 3 multiplications required exactly 3 DSP48 slices.

2. Modified Design (24-bit Inputs)

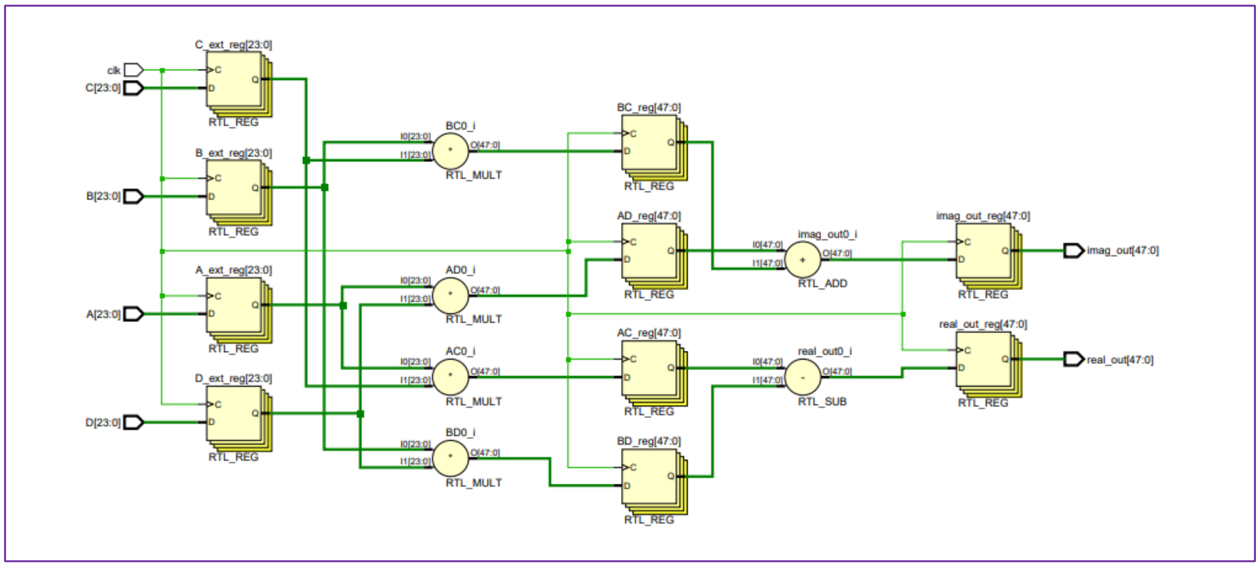
- Each 24-bit value no longer fits the 18-bit operand limit of the DSP48 multiplier.
- Vivado will split the 24-bit value into two parts:
- Each 24×24 multiplication now requires 2 DSP slices to compute the full product.

Since there are 3 multiplications in the optimized form (or 4 multiplications in the un-optimized form), we can calculate:

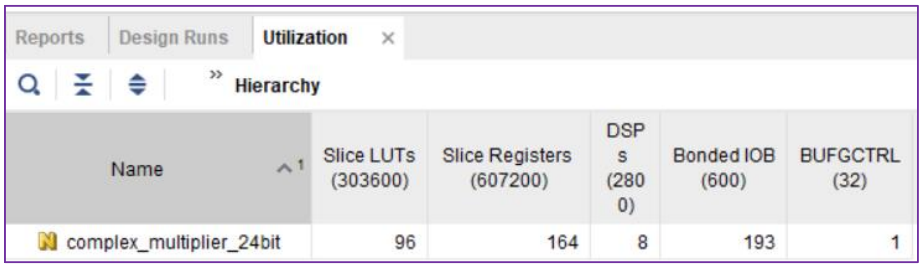
Optimized Design (3 multiplications) → 3 × 2 = 6 DSPs

Un-optimized Design (4 multiplications) → 4 × 2 = 8 DSPs

```
`timescale 1ns / 1ps

module complex_multiplier_24bit (
    input clk,
    input [23:0] A, B, C, D,
    output reg [47:0] real_out, imag_out
);

    // Intermediate values for separate multiplications
    reg [47:0] AC, BD, AD, BC;

    // Force DSP usage for all multiplications
    (* use_dsp = "yes" *) reg [23:0] A_ext, B_ext, C_ext, D_ext;

    // First stage: Compute each multiplication independently
    always @(posedge clk) begin
        A_ext <= A;
        B_ext <= B;
        C_ext <= C;
        D_ext <= D;

        AC <= A_ext * C_ext;    // DSP 1 & 2
        BD <= B_ext * D_ext;    // DSP 3 & 4
        AD <= A_ext * D_ext;    // DSP 5 & 6
        BC <= B_ext * C_ext;    // DSP 7 & 8
    end

    // Second stage: Compute final outputs
    always @(posedge clk) begin
        real_out <= AC - BD;               // Real part
        imag_out <= AD + BC;               // Imaginary part
    end

endmodule
```



RTL **schematic**



Name	Slice LUTs (303600)	Slice Registers (607200)	DSPs (2800)	Bonded IOB (600)	BUFGCTRL (32)
complex_multiplier_24bit	96	164	8	193	1

Utilization report

Part4

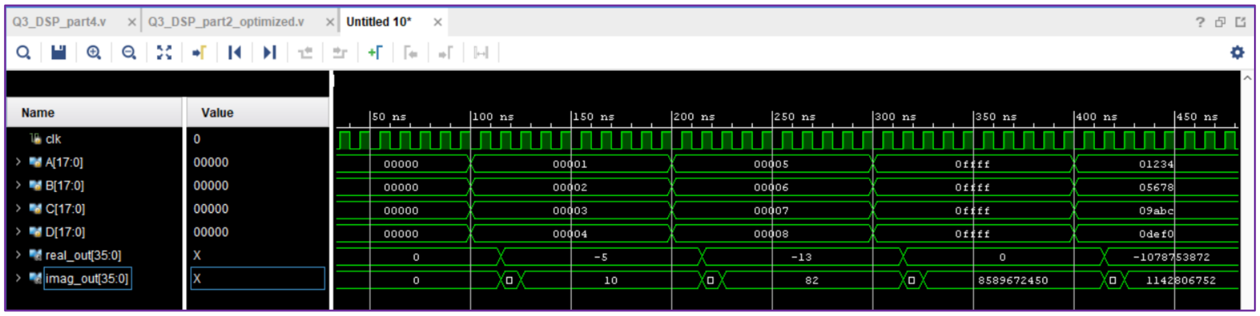
In order to make sure that the verilog codes are functional, we simulate them.

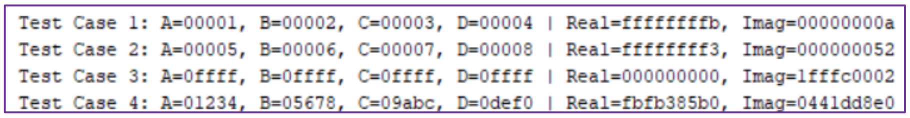
Here is the testbench:

```
`timescale 1ns / 1ps

module complex_multiplier_optimized_tb;

    // Inputs
    reg clk;
    reg [17:0] A, B, C, D;

    // Outputs
    wire [35:0] real_out, imag_out;

    // Instantiate the Unit Under Test (UUT)
    complex_multiplier_optimized uut (
        .clk(clk),
        .A(A),
        .B(B),
        .C(C),
        .D(D),
        .real_out(real_out),
        .imag_out(imag_out)
    );

    // Clock generation
    initial begin
        clk = 0;
        forever #5 clk = ~clk; // Toggle clock every 5 time units
    end

    // Test stimulus
    initial begin
        // Initialize inputs
        A = 18'h0;
        B = 18'h0;
        C = 18'h0;
        D = 18'h0;

        // Wait for global reset or initial setup
        #100;

        // Test case 1
        A = 18'h1;
        B = 18'h2;
        C = 18'h3;
        D = 18'h4;
        #100; // Wait for the outputs to stabilize
        $display("Test Case 1: A=%h, B=%h, C=%h, D=%h | Real=%h, Imag=%h",
                 A, B, C, D, real_out, imag_out);

        // Test case 2
        A = 18'h5;
        B = 18'h6;
        C = 18'h7;
        D = 18'h8;
        #100;
        $display("Test Case 2: A=%h, B=%h, C=%h, D=%h | Real=%h, Imag=%h",
                 A, B, C, D, real_out, imag_out);
```

```verilog
        // Test case 3
        A = 18'hFFFF;
        B = 18'hFFFF;
        C = 18'hFFFF;
        D = 18'hFFFF;
        #100;
        $display("Test Case 3: A=%h, B=%h, C=%h, D=%h | Real=%h, Imag=%h",
                 A, B, C, D, real_out, imag_out);

        // Test case 4
        A = 18'h1234;
        B = 18'h5678;
        C = 18'h9ABC;
        D = 18'hDEF0;
        #100;
        $display("Test Case 4: A=%h, B=%h, C=%h, D=%h | Real=%h, Imag=%h",
                 A, B, C, D, real_out, imag_out);

        // End simulation
        $stop;
    end

endmodule
```

The following is the result of simulation for different tests in testbench file:





```
Test Case 1: A=00001, B=00002, C=00003, D=00004 | Real=fffffffffb, Imag=00000000a
Test Case 2: A=00005, B=00006, C=00007, D=00008 | Real=ffffffff3, Imag=000000052
Test Case 3: A=0ffff, B=0ffff, C=0ffff, D=0ffff | Real=000000000, Imag=1fffc0002
Test Case 4: A=01234, B=05678, C=09abc, D=0def0 | Real=fbfb385b0, Imag=0441dd8e0
```

Part5

Attributes in Verilog for Synthesis and Implementation

Attributes in Verilog are special directives that provide hints to synthesis tools (like Vivado, Quartus, etc.) to control design decisions during the synthesis and implementation process. They do not affect the logic itself but influence **resource usage**, **timing**, or **optimization**.

Syntax of Attributes

In Verilog, attributes are written in the format:

```
(* attribute_name = "value" *) signal_or_module;
```

They are placed **before** the relevant element such as a wire, register, or module declaration.

Commonly Used Attributes in Vivado

1. **`(* use_dsp = "no" *)`** / **`(* use_dsp = "yes" *)`**
   - Controls whether DSP blocks are used for multiplications.
   - Example:

   ```
   (* use_dsp = "no" *) wire [37:0] result = A * B;
   ```

2. **`(* keep = "true" *)`**
   - Prevents Vivado from optimizing away specific signals or logic.
   - Useful when debugging intermediate signals.
   - Example:

   ```
   (* keep = "true" *) reg [7:0] temp_value;
   ```

3. **`(* dont_touch = "true" *)`**
   - Ensures a module or instance is preserved exactly as written, even if optimizations would normally modify it.
   - Example:

   ```
   (* dont_touch = "true" *) module my_custom_block (...);
   ```

4. **`(* max_fanout = "N" *)`**
   - Limits the number of destinations a signal can drive, helping with timing control.
   - Example:

   ```
   (* max_fanout = "10" *) wire clk_buf;
   ```

5. **`(* mark_debug = "true" *)`**

- Marks signals for observation in Vivado's **ILA (Integrated Logic Analyzer)** for on-chip debugging.
- Example:

```
(* mark_debug = "true" *) reg [15:0] data_bus;
```

6. **`(* RAM_STYLE = "block" *)`**
   - Directs the synthesis tool to infer **Block RAM (BRAM)** instead of distributed memory.
   - Example:

```
(* RAM_STYLE = "block" *) reg [7:0] memory [0:255];
```

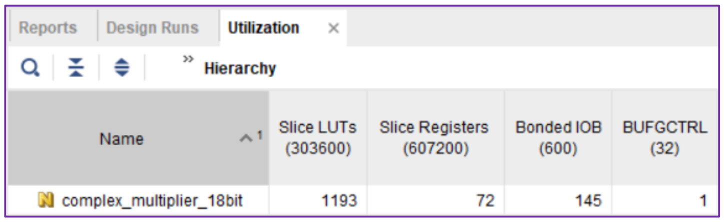
Why Use Attributes?

Control hardware resource utilization (e.g., DSPs, BRAMs, etc.).
Improve timing by limiting fanout or enforcing pipeline stages.
Preserve specific structures for debugging.
Ensure certain logic remains untouched by optimizations.

To prevent Vivado (or another synthesis tool) from inferring DSP blocks for multiplication, we can use the `(* use_dsp = "no" *)` attribute. This attribute directs the tool to implement multipliers using **LUTs** (Look-Up Tables) instead of DSP slices.

Modified code:

```
module complex_multiplier_18bit (
    input clk,
    input [17:0] A, B, C, D,
    output reg [35:0] real_out,
    output reg [35:0] imag_out
);
    // Intermediate signals for calculations
    reg [35:0] AC, BD, AB_CD;

    // Prevent DSP usage for multiplications
    (* use_dsp = "no" *) wire [35:0] A_C = A * C;
    (* use_dsp = "no" *) wire [35:0] B_D = B * D;
    (* use_dsp = "no" *) wire [35:0] ABxCD = (A + B) * (C + D);

    // Pipeline or sequential logic
    always @(posedge clk) begin
        real_out <= A_C - B_D;
        imag_out <= ABxCD - A_C - B_D;
    end
endmodule
```



Name	Slice LUTs (303600)	Slice Registers (607200)	Bonded IOB (600)	BUFGCTRL (32)
complex_multiplier_18bit	1193	72	145	1

utilization report

as shown in the utilization report no DSP was used.

Question 4

ALU design with two 8-bit inputs

The goal of this question is to design and implement an 8-bit Arithmetic Logic Unit (ALU) in Verilog. The ALU performs various arithmetic and logical operations on two 8-bit unsigned inputs (i_a and i_b) based on a 4-bit control signal (i_cont). The output (o_out) is 9 bits wide, with the 9th bit representing a carry/borrow flag for arithmetic operations or an overflow flag for multiplication.

The ALU supports the wanted operations based on the 4-bit control signal (i_cont).

For division operation it Divides i_a by i_b using a restoring division algorithm.

And for multiplization operation it Multiplies i_a and i_b. The lower 8 bits are output, and the 9th bit is the carry.

Here is the code:

```verilog
module ALU (
    input [7:0] i_a, i_b,
    input [3:0] i_cont,
    input i_clk,
    output reg [8:0] o_out
);
reg [15:0] product;
reg [7:0] quotient, remainder;
            integer i;

always @(posedge i_clk) begin
    case (i_cont)
        4'b0000: {o_out[8], o_out[7:0]} = i_a + i_b;  // Addition with carry
        4'b0001: {o_out[8], o_out[7:0]} = i_a - i_b;  // Subtraction with carry
        4'b0010: begin // Multiplication with lower 8 bits + carry
          product = i_a * i_b;
          o_out[7:0] = product[7:0];
          o_out[8] = |product[15:8];
        end
        4'b0011: begin  // Division using Restoring Division Algorithm

            quotient = 0;
            remainder = 0;
            for (i = 7; i >= 0; i = i - 1) begin
                remainder = (remainder << 1) | (i_a[i]);
                if (remainder >= i_b) begin
                    remainder = remainder - i_b;
                    quotient[i] = 1;
                end
            end
            o_out = {1'b0, quotient};  // Division result in lower 8 bits
        end
        4'b0100: o_out = i_a << 1;  // Logical left shift
        4'b0101: o_out = i_a >> 1;  // Logical right shift
        4'b0110: o_out = {i_a[6:0], i_a[7]};  // Rotate left
        4'b0111: o_out = {i_a[0], i_a[7:1]};  // Rotate right
        4'b1000: o_out = i_a & i_b;  // AND
        4'b1001: o_out = i_a | i_b;  // OR
        4'b1010: o_out = i_a ^ i_b;  // XOR
        4'b1011: o_out = {1'b0,~(i_a | i_b)};  // NOR
        4'b1100: o_out = {1'b0,~(i_a & i_b)};  // NAND
        4'b1101: o_out = {1'b0,~(i_a ^ i_b)};  // XNOR
        4'b1110: o_out = (i_a == i_b) ? 9'b1 : 9'b0;  // Equality check
        4'b1111: o_out = (i_a > i_b) ? 9'b1 : 9'b0;  // Greater than check
        default: o_out = 9'b0;  // Default case
    endcase
end

endmodule
```

A testbench is designed to verify the functionality of the ALU. The testbench performs the following steps:

Initializes the inputs (i_a, i_b, i_cont, and i_clk).

Applies different control signals to test all operations.

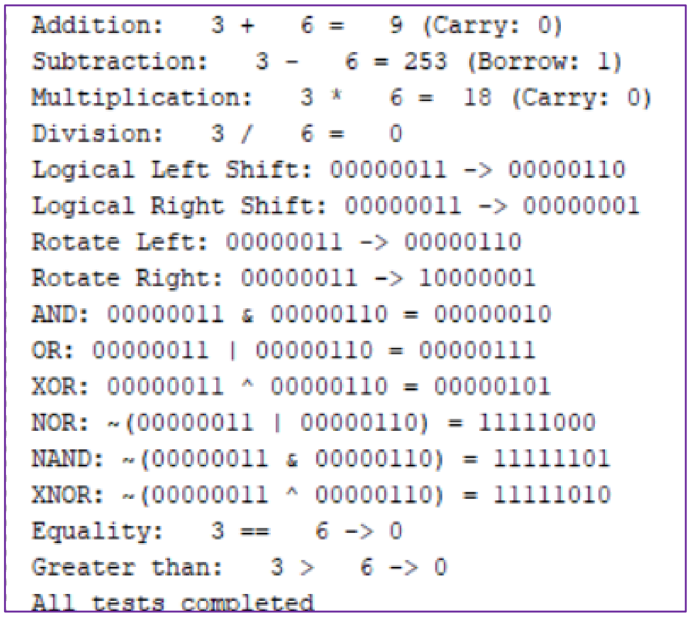
Displays the results using $display statements.

Here is the code:

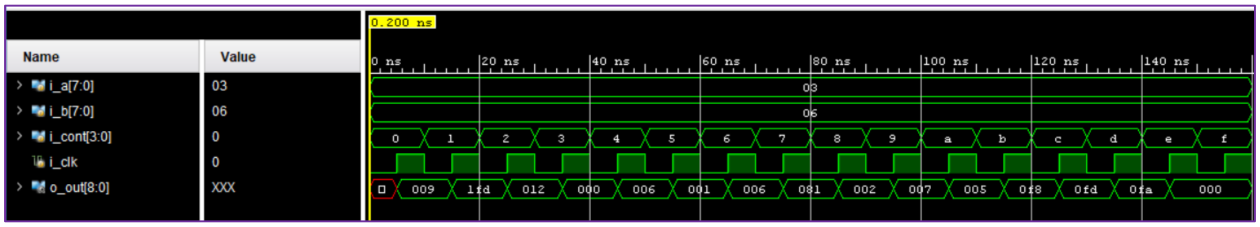
```
`timescale 1ns / 1ps

module ALU_tb;

    // Inputs
    reg [7:0] i_a, i_b;
    reg [3:0] i_cont;
    reg i_clk;

    // Output
    wire [8:0] o_out;

    // Instantiate the ALU
```

```verilog
    ALU uut (
        .i_a(i_a),
        .i_b(i_b),
        .i_cont(i_cont),
        .i_clk(i_clk),
        .o_out(o_out)
    );

    // Clock generation
    always #5 i_clk = ~i_clk;

    initial begin
        // Initialize inputs
        i_clk = 0;
        i_b = 8'b00000110;  // 6
        i_a = 8'b00000011;  // 3

        // Addition Test
        i_cont = 4'b0000;
        #10 $display("Addition: %d + %d = %d (Carry: %b)", i_a, i_b, o_out[7:0],
o_out[8]);

        // Subtraction Test
        i_cont = 4'b0001;
        #10 $display("Subtraction: %d - %d = %d (Borrow: %b)", i_a, i_b,
(o_out[7:0]), o_out[8]);


        // Multiplication Test
        i_cont = 4'b0010;
        #10 $display("Multiplication: %d * %d = %d (Carry: %b)", i_a, i_b,
o_out[7:0], o_out[8]);

        // Division Test
        i_cont = 4'b0011;
        #10 $display("Division: %d / %d = %d", i_a, i_b, o_out[7:0]);

        // Logical Left Shift Test
        i_cont = 4'b0100;
        #10 $display("Logical Left Shift: %b -> %b", i_a, o_out[7:0]);

        // Logical Right Shift Test
        i_cont = 4'b0101;
        #10 $display("Logical Right Shift: %b -> %b", i_a, o_out[7:0]);
        // Rotate Left Test
        i_cont = 4'b0110;
        #10 $display("Rotate Left: %b -> %b", i_a, o_out[7:0]);

        // Rotate Right Test
        i_cont = 4'b0111;
        #10 $display("Rotate Right: %b -> %b", i_a, o_out[7:0]);

        // AND Test
        i_cont = 4'b1000;
        #10 $display("AND: %b & %b = %b", i_a, i_b, o_out[7:0]);

        // OR Test
        i_cont = 4'b1001;
        #10 $display("OR: %b | %b = %b", i_a, i_b, o_out[7:0]);

        // XOR Test
        i_cont = 4'b1010;
        #10 $display("XOR: %b ^ %b = %b", i_a, i_b, o_out[7:0]);

        // NOR Test
        i_cont = 4'b1011;
        #10 $display("NOR: ~(%b | %b) = %b", i_a, i_b, o_out[7:0]);

        // NAND Test
        i_cont = 4'b1100;
        #10 $display("NAND: ~(%b & %b) = %b", i_a, i_b, o_out[7:0]);

        // XNOR Test
        i_cont = 4'b1101;
        #10 $display("XNOR: ~(%b ^ %b) = %b", i_a, i_b, o_out[7:0]);

        // Equality Check Test
        i_cont = 4'b1110;
        #10 $display("Equality: %d == %d -> %b", i_a, i_b, o_out[0]);

        // Greater Than Check Test
        i_cont = 4'b1111;
        #10 $display("Greater than: %d > %d -> %b", i_a, i_b, o_out[0]);

        $display("All tests completed");
        $stop;
    end
endmodule
```

for test case a=3 and b=6 we got these results:



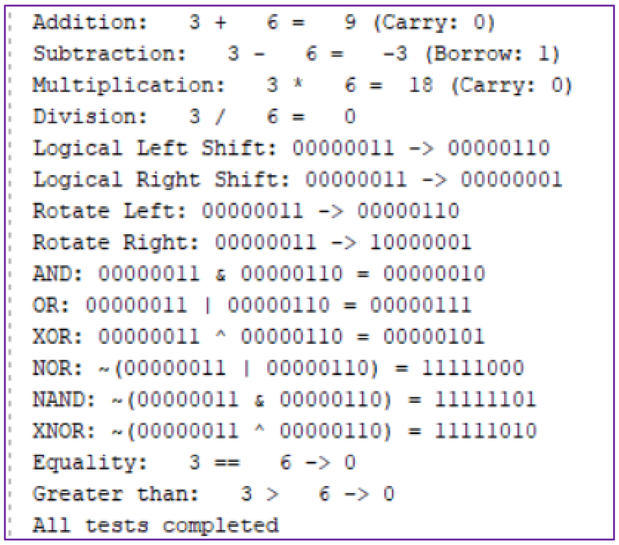
```
Addition:   3 +   6 =   9 (Carry: 0)
Subtraction:   3 -   6 = 253 (Borrow: 1)
Multiplication:   3 *   6 =  18 (Carry: 0)
Division:   3 /   6 =   0
Logical Left Shift: 00000011 -> 00000110
Logical Right Shift: 00000011 -> 00000001
Rotate Left: 00000011 -> 00000110
Rotate Right: 00000011 -> 10000001
AND: 00000011 & 00000110 = 00000010
OR: 00000011 | 00000110 = 00000111
XOR: 00000011 ^ 00000110 = 00000101
NOR: ~(00000011 | 00000110) = 11111000
NAND: ~(00000011 & 00000110) = 11111101
XNOR: ~(00000011 ^ 00000110) = 11111010
Equality:   3 ==   6 -> 0
Greater than:   3 >   6 -> 0
All tests completed
```



Name	Value
i_a[7:0]	03
i_b[7:0]	06
i_cont[3:0]	0
i_clk	0
o_out[8:0]	XXX

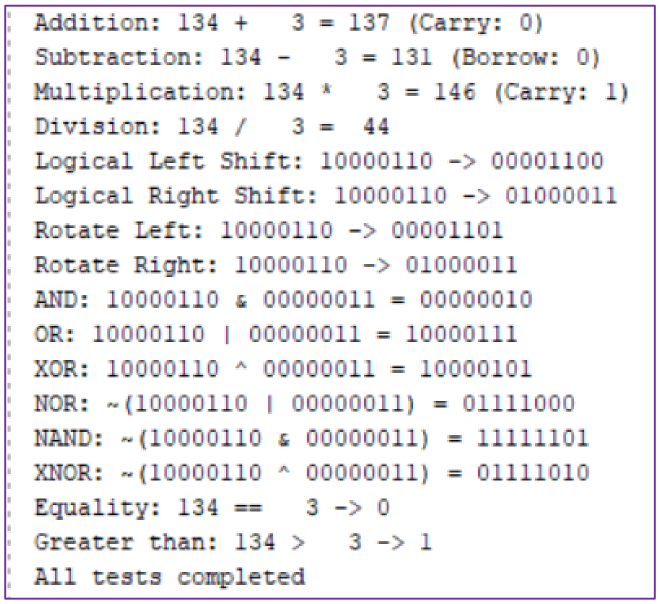
0.200 ns
0 ns
20 ns
40 ns
60 ns
80 ns
100 ns
120 ns
140 ns
03
06
0 1 2 3 4 5 6 7 8 9 a b c d e f
009 1fd 012 000 006 001 006 081 002 007 005 0f8 0fd 0fa 000

All operations worked perfectly. Borrow was also detected therefore the result of the substraction is negative so if we want to see its real decimal value we should use $signed if we do well have:

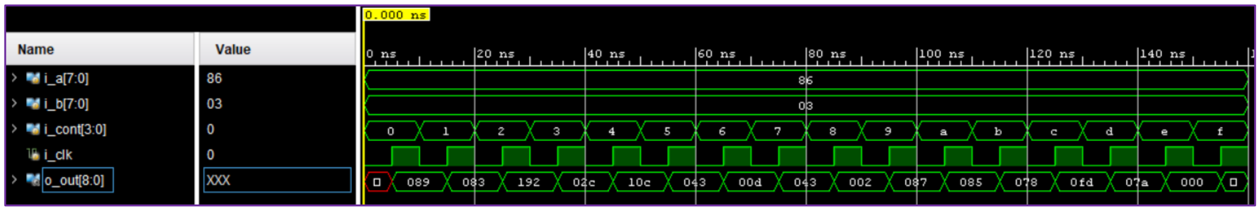


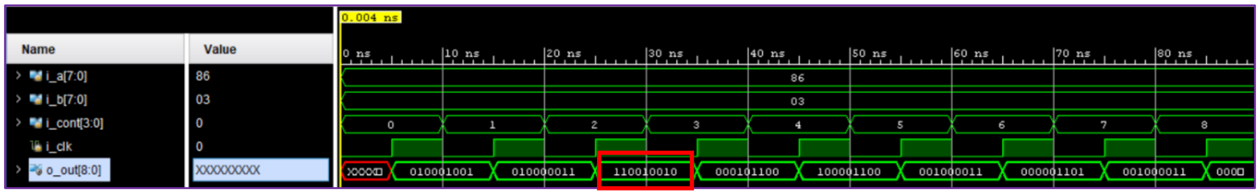
```
Addition:   3 +   6 =   9 (Carry: 0)
Subtraction:   3 -   6 =   -3 (Borrow: 1)
Multiplication:   3 *   6 =  18 (Carry: 0)
Division:   3 /   6 =   0
Logical Left Shift: 00000011 -> 00000110
Logical Right Shift: 00000011 -> 00000001
Rotate Left: 00000011 -> 00000110
Rotate Right: 00000011 -> 10000001
AND: 00000011 & 00000110 = 00000010
OR: 00000011 | 00000110 = 00000111
XOR: 00000011 ^ 00000110 = 00000101
NOR: ~(00000011 | 00000110) = 11111000
NAND: ~(00000011 & 00000110) = 11111101
XNOR: ~(00000011 ^ 00000110) = 11111010
Equality:   3 ==   6 -> 0
Greater than:   3 >   6 -> 0
All tests completed
```

To check the overflow at multiplication the next test case is a=134 and b=3



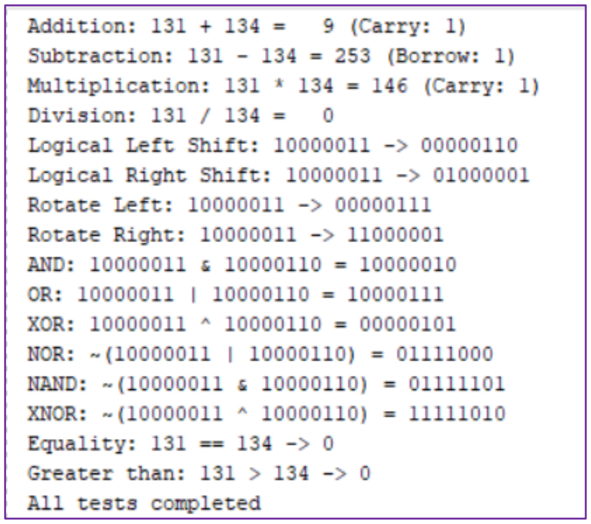
```
Addition: 134 +   3 = 137 (Carry: 0)
Subtraction: 134 -   3 = 131 (Borrow: 0)
Multiplication: 134 *   3 = 146 (Carry: 1)
Division: 134 /   3 =  44
Logical Left Shift: 10000110 -> 00001100
Logical Right Shift: 10000110 -> 01000011
Rotate Left: 10000110 -> 00001101
Rotate Right: 10000110 -> 01000011
AND: 10000110 & 00000011 = 00000010
OR: 10000110 | 00000011 = 10000111
XOR: 10000110 ^ 00000011 = 10000101
NOR: ~(10000110 | 00000011) = 01111000
NAND: ~(10000110 & 00000011) = 11111101
XNOR: ~(10000110 ^ 00000011) = 01111010
Equality: 134 ==   3 -> 0
Greater than: 134 >   3 -> 1
All tests completed
```



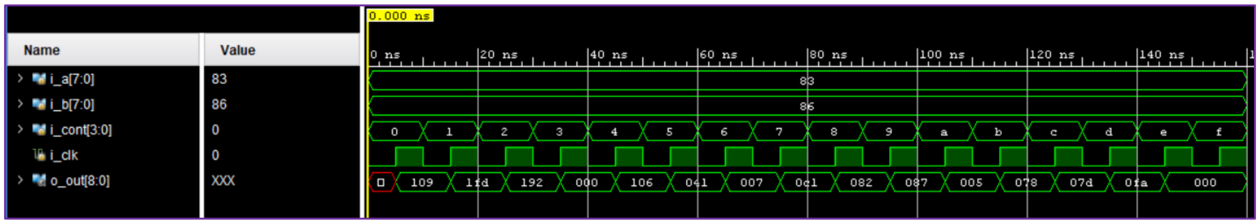


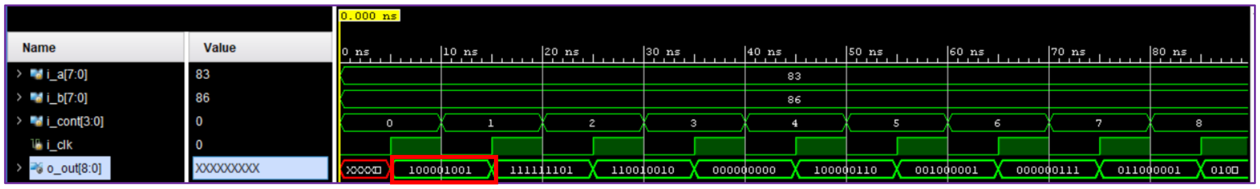
As you can see the 9th bit is set to hight but lsb byte is shown in the output.

And for the last testcase to check addition carry we go with a=131 and b=134



```
Addition: 131 + 134 =   9 (Carry: 1)
Subtraction: 131 - 134 = 253 (Borrow: 1)
Multiplication: 131 * 134 = 146 (Carry: 1)
Division: 131 / 134 =   0
Logical Left Shift: 10000011 -> 00000110
Logical Right Shift: 10000011 -> 01000001
Rotate Left: 10000011 -> 00000111
Rotate Right: 10000011 -> 11000001
AND: 10000011 & 10000110 = 10000010
OR: 10000011 | 10000110 = 10000111
XOR: 10000011 ^ 10000110 = 00000101
NOR: ~(10000011 | 10000110) = 01111000
NAND: ~(10000011 & 10000110) = 01111101
XNOR: ~(10000011 ^ 10000110) = 11111010
Equality: 131 == 134 -> 0
Greater than: 131 > 134 -> 0
All tests completed
```





As you can see carry is detected in the MSB bit of the output but the LSB byte is shown in display.

The 8-bit ALU is successfully designed and implemented in Verilog. It supports a wide range of arithmetic, logical, shift, and comparison operations. The testbench verifies the correctness of all operations, ensuring the ALU meets the design specifications. This code demonstrates the use of Verilog for digital design and highlights the importance of careful handling of arithmetic operations in hardware.

Question 5

Designing the CRC module and checking the data accuracy in a simple channel

In this part, I present the design and implementation of a Cyclic Redundancy Check (CRC) module to ensure data integrity in a simple communication channel. The CRC module is designed to detect multi-bit errors more effectively than simple parity checks. The system consists of two main components: a transmitter and a receiver. The transmitter calculates the CRC value for a 16-bit data input and appends it to the data, while the receiver verifies the integrity of the received 32-bit word.

The system uses a 16-bit CRC polynomial (represented as 16'h8007 in hexadecimal). The transmitter encodes the 16-bit data into a 32-bit word by appending the computed CRC value. The receiver decodes the 32-bit word, recalculates the CRC, and checks for errors.

Here is the code for transmitter and receiver:

```
module data_encoder(
    input wire clk,
    input wire [15:0] data_in,
    output wire [31:0] encoded_out
);

    reg [15:0] crc_val;
    reg [31:0] shift_reg;
    integer idx;
    parameter poly = 16'h8007;
    assign encoded_out = {data_in, crc_val};

    always @(posedge clk) begin
        crc_val = 16'b0;
        shift_reg = {data_in, 16'b0};

        for (idx = 0; idx < 16; idx = idx + 1) begin
            if (shift_reg[31])
                shift_reg[31:16] = shift_reg[31:16] ^ poly;

            shift_reg = shift_reg << 1;
        end

        crc_val = shift_reg[31:16];
    end

endmodule
```

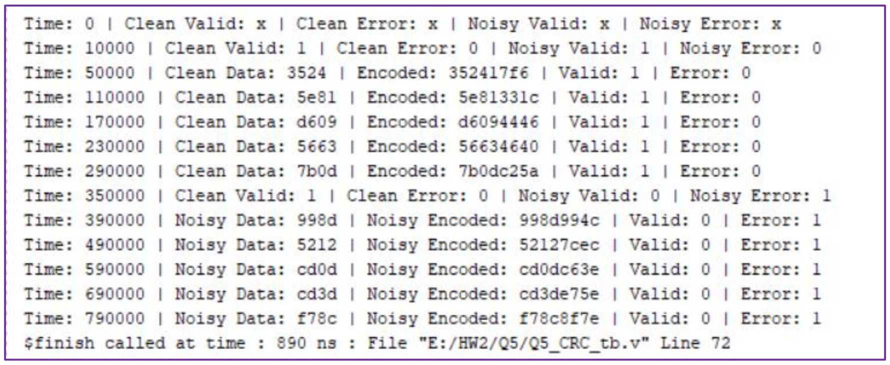
```
module data_decoder(
    input wire clk,
    input wire [31:0] data_in,
    output reg valid_flag,
    output reg error_flag
);

    reg [15:0] crc_check;
    reg [31:0] shift_reg;
    integer idx;
    parameter poly = 16'h8007;
    //assign valid_flag = (crc_check == data_in[15:0]) ? 1'b1 :1'b0 ;
    //assign error_flag = (crc_check == data_in[15:0]) ? 1'b0 :1'b1 ;

    always @(posedge clk) begin
        crc_check = 16'b0;
        shift_reg = {data_in[31:16], 16'b0};

        for (idx = 0; idx < 16; idx = idx + 1) begin
            if (shift_reg[31])
                shift_reg[31:16] = shift_reg[31:16] ^ poly;
            shift_reg = shift_reg << 1;
        end

        crc_check = shift_reg[31:16];

        if (!(crc_check == data_in[15:0]) )begin
             valid_flag <= 0;
             error_flag <= 1;
         end else begin
             valid_flag <= 1;
             error_flag <= 0;
         end
    end

endmodule
```

This part describes the testbench designed to verify the functionality of the CRC-based data integrity system. The testbench evaluates both the transmitter (data_encoder) and receiver (data_decoder) modules under two scenarios: clean data transmission and noisy data transmission. The goal is to ensure that the system correctly encodes, decodes, and detects errors in the transmitted data.

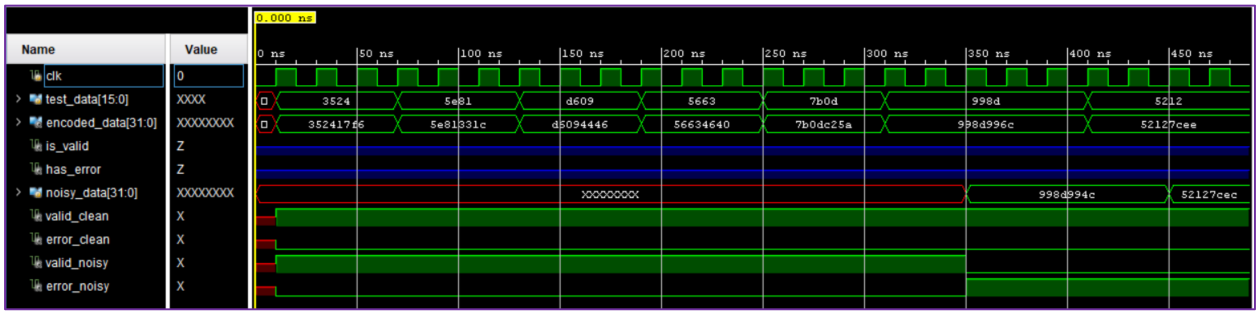
The testbench, tb_CRC_System, instantiates the following modules:
Transmitter (data_encoder):
Encodes 16-bit input data into a 32-bit word by appending a 16-bit CRC value.
Receiver (data_decoder):
Decodes the 32-bit word and checks for errors using the CRC value.
Two instances of the receiver are used: one for clean data and one for noisy data.
The testbench simulates the following scenarios:
Clean Data Transmission:
Data is transmitted without errors, and the receiver validates the data.
Noisy Data Transmission:
Errors are introduced into the encoded data, and the receiver detects the errors.

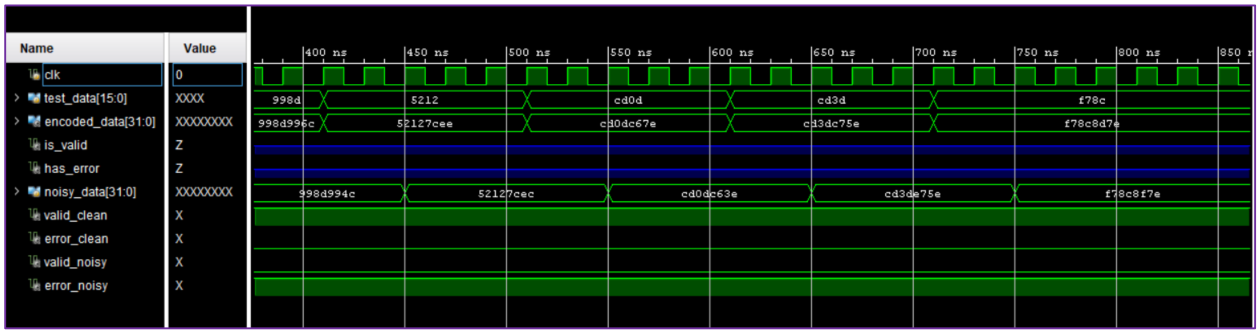
Here is the code:

```
`timescale 1ns/1ps

module tb_CRC_System();

    reg clk;
    reg [15:0] test_data;
    wire [31:0] encoded_data;
    wire is_valid, has_error;
    reg [31:0] noisy_data;
    wire valid_clean, error_clean;
    wire valid_noisy, error_noisy;


    data_encoder tx (
        .clk(clk),
        .data_in(test_data),
        .encoded_out(encoded_data)
    );


    data_decoder rx_clean (
        .clk(clk),
        .data_in(encoded_data),
        .valid_flag(valid_clean),
        .error_flag(error_clean)
    );

     data_decoder rx_noisy (
        .clk(clk),
        .data_in(noisy_data),
        .valid_flag(valid_noisy),
        .error_flag(error_noisy)
    );

    always #10 clk = ~clk;

    initial begin
        clk = 0;


        repeat (5) begin
            @(posedge clk);
            test_data = $urandom % 65536;
            @(posedge clk);
            @(posedge clk);

            $display("Time: %0t | Clean Data: %0x | Encoded: %0x | Valid: %b |
Error: %b",
                     $time, test_data, encoded_data, valid_clean, error_clean);
        end


        repeat (5) begin
            @(posedge clk);
            test_data = $urandom % 65536;

            @(posedge clk);
            @(posedge clk);


            noisy_data = encoded_data ^ (1 << ($urandom % 16));

            @(posedge clk);
            @(posedge clk);

            $display("Time: %0t | Noisy Data: %0x | Noisy Encoded: %0x | Valid:
%b | Error: %b",
                     $time, test_data, noisy_data, valid_noisy, error_noisy);
        end

        #100;
        $finish;
    end

    initial begin
        $monitor("Time: %0t | Clean Valid: %b | Clean Error: %b | Noisy Valid: %b
| Noisy Error: %b",
                     $time, valid_clean, error_clean, valid_noisy, error_noisy);
    end

endmodule
```



```
Time: 0 | Clean Valid: x | Clean Error: x | Noisy Valid: x | Noisy Error: x
Time: 10000 | Clean Valid: 1 | Clean Error: 0 | Noisy Valid: 1 | Noisy Error: 0
Time: 50000 | Clean Data: 3524 | Encoded: 352417f6 | Valid: 1 | Error: 0
Time: 110000 | Clean Data: 5e81 | Encoded: 5e81331c | Valid: 1 | Error: 0
Time: 170000 | Clean Data: d609 | Encoded: d6094446 | Valid: 1 | Error: 0
Time: 230000 | Clean Data: 5663 | Encoded: 56634640 | Valid: 1 | Error: 0
Time: 290000 | Clean Data: 7b0d | Encoded: 7b0dc25a | Valid: 1 | Error: 0
Time: 350000 | Clean Valid: 1 | Clean Error: 0 | Noisy Valid: 0 | Noisy Error: 1
Time: 390000 | Noisy Data: 998d | Noisy Encoded: 998d994c | Valid: 0 | Error: 1
Time: 490000 | Noisy Data: 5212 | Noisy Encoded: 52127cec | Valid: 0 | Error: 1
Time: 590000 | Noisy Data: cd0d | Noisy Encoded: cd0dc63e | Valid: 0 | Error: 1
Time: 690000 | Noisy Data: cd3d | Noisy Encoded: cd3de75e | Valid: 0 | Error: 1
Time: 790000 | Noisy Data: f78c | Noisy Encoded: f78c8f7e | Valid: 0 | Error: 1
$finish called at time : 890 ns : File "E:/HW2/Q5/Q5_CRC_tb.v" Line 72
```





The testbench produces the following outputs:

Clean Data Transmission:
The valid_clean signal is high, indicating that the data was successfully validated.
The error_clean signal remains low, confirming no errors were detected.

Noisy Data Transmission:
The valid_noisy signal is low, indicating that the data was invalid.
The error_noisy signal is high, confirming that errors were detected.

As we can see the error signal is set to hight when the noisy data is asserted so the modules work correctly.

CRC vs. Simple Parity for Error Detection

Simple Parity:

Adds a single parity bit to data.

Detects single-bit errors but fails for even-numbered multi-bit errors.

Limited effectiveness in noisy or high-data-rate environments.

Cyclic Redundancy Check (CRC):

Uses a polynomial-based algorithm to generate a CRC code (e.g., 16 or 32 bits).

Detects all single-bit errors, all double-bit errors, and most multi-bit and burst errors.

Highly effective in noisy channels and for long data packets.

Why CRC is Better:

Multi-Bit Error Detection: CRC detects most multi-bit errors, while parity fails for even-numbered errors.

Burst Error Detection: CRC excels at detecting contiguous bit errors, common in noisy environments.

Higher Redundancy: CRC's longer redundancy code provides stronger error detection than a single parity bit.

Flexibility: CRC polynomials can be tailored for specific error patterns.

When to Use CRC:

Noisy communication channels (e.g., wireless networks).

High-data-rate systems with increased error probability.

Critical applications requiring high data integrity (e.g., financial, medical data).

Long data packets where parity is insufficient.

Conclusion:

CRC is significantly more robust than simple parity, making it the preferred choice for modern communication systems where reliability and error detection are critical.