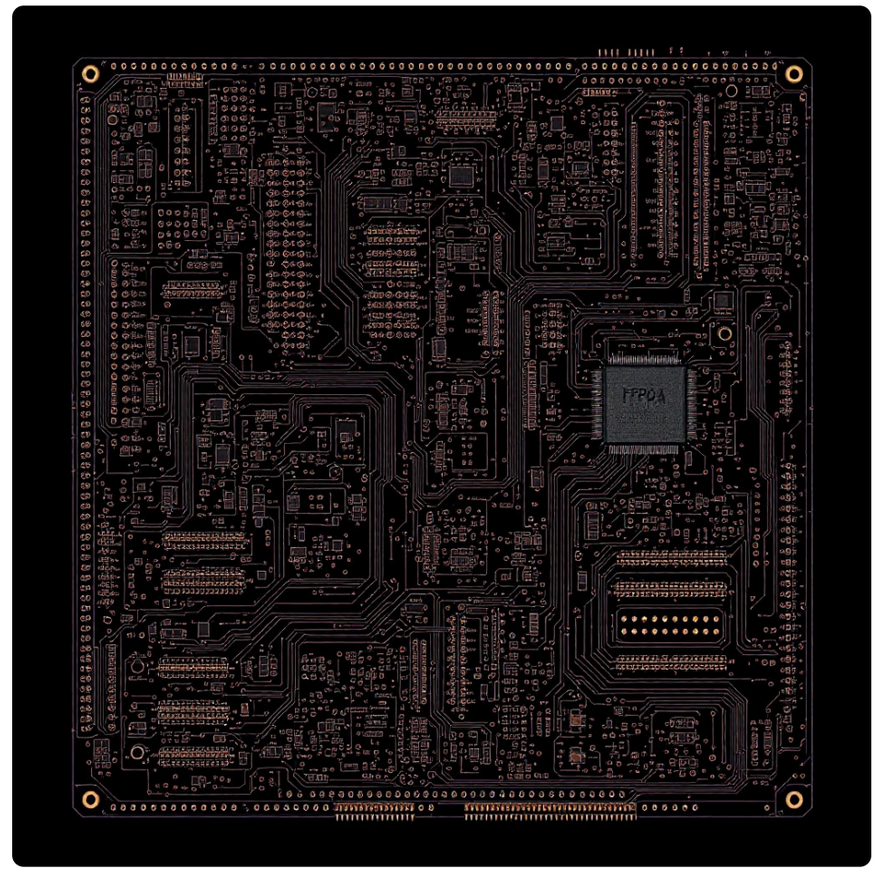
In the name of GOD

Assignment 3

Narges Vali

401102791

- **Question 1: Familiarity with IP Core**
- **Question 2: Direct Digital Synthesizer (DDS)**
- **Question 3: FFT Implementation**
- **Question 4: Using the IP Core for FFT**
- **Question 5: Comparison of Direct Implementation and Use of IP Core**



Question 1:Familiarity with IP Core

An **IP Core (Intellectual Property Core)** is a pre-designed, reusable logic block that plays a crucial role in digital system design, such as in **FPGA** or **ASIC** designs. These cores are responsible for performing specific functions like memory control, data processing, or communication tasks. By utilizing IP cores, designers can significantly reduce development time, lower design costs, and improve the overall performance of complex systems.

There are three main types of IP cores:

1

Hard IP Core

- *Characteristics*: Optimized for a specific fabrication process, offering high performance and low power.
- *Use case*: Best suited for mass production, high-performance systems.

2

Soft IP Core

- *Characteristics*: Delivered as synthesizable HDL code (e.g., VHDL, Verilog). Highly flexible and portable, but may have lower performance.
- *Use case*: Ideal for research, prototyping, and customizable designs.

3

Firm IP Core

- *Characteristics*: A hybrid approach, where the core is delivered as RTL code but partially optimized for a specific process.
- *Use case*: Suitable for semi-custom designs that require a balance of performance and flexibility.

Advantages of Using IP Cores

- **Reduced Design Time**: Pre-designed and pre-verified blocks help avoid starting from scratch.
- **Lower Error Rates**: Using trusted, pre-verified cores minimizes bugs and enhances system reliability.
- **Cost Efficiency**: Reusing existing IP reduces the need for extensive development resources.
- **Higher Quality**: IP cores from reputable vendors are often rigorously tested for quality and performance.
- **Focus on Innovation**: Designers can dedicate more time to system-level customization rather than low-level details.

This approach enables rapid and efficient development of complex systems while ensuring high-quality and reliable outputs.

Question 2: Direct Digital Synthesizer (DDS)

a.Implementing a 1 kHz Sine Wave Generator using DDS Compiler IP in Vivado

In this section, a sine wave generator with a frequency of 1 kHz is implemented using the Xilinx **DDS Compiler IP Core** in Vivado. This method uses a **Direct Digital Synthesis (DDS)** approach and relies on an internal **Look-Up Table (LUT)** for waveform generation.

Design Parameters

Parameter	Value	Description
System Clock	250 MHz	Sampling rate
Output Frequency	1 kHz	Target sine wave frequency
Frequency Resolution	0.0582 Hz	Very high tuning precision
SFDR	90 dB	High spectral purity
Phase Width	32 bits	High phase accumulator resolution
Output Width	16 bits	Suitable for DAC or FFT input
Noise Shaping	Taylor Series Corrected	Improves signal quality
Memory Type	Block ROM	Efficient LUT storage

Implementation Steps

The following steps were taken in Vivado to implement the DDS:

1. A new RTL project was created in Vivado.
2. The **DDS Compiler IP** was added from the IP Catalog.
3. Configuration parameters were set as shown above.
4. Output frequency was set to **1 kHz** using either the Frequency control word or the Output Frequencies tab.
5. The IP was generated successfully, and its instantiation template (.veo) was included in the project.
6. The design is ready for synthesis.

```
dds_compiler_0 your_instance_name (
  .aclk(aclk),                                  // input wire aclk
  .s_axis_phase_tvalid(s_axis_phase_tvalid),  // input wire s_axis_phase_tvalid
  .s_axis_phase_tdata(s_axis_phase_tdata),    // input wire [31 : 0] s_axis_phase_tdata
  .s_axis_phase_tlast(s_axis_phase_tlast),    // input wire s_axis_phase_tlast
  .m_axis_data_tvalid(m_axis_data_tvalid),    // output wire m_axis_data_tvalid
  .m_axis_data_tdata(m_axis_data_tdata),      // output wire [31 : 0] m_axis_data_tdata
  .m_axis_data_tlast(m_axis_data_tlast),      // output wire m_axis_data_tlast
  .m_axis_phase_tvalid(m_axis_phase_tvalid),  // output wire m_axis_phase_tvalid
  .m_axis_phase_tdata(m_axis_phase_tdata),    // output wire [31 : 0] m_axis_phase_tdata
  .m_axis_phase_tlast(m_axis_phase_tlast)    // output wire m_axis_phase_tlast
);
```

Design Justification

- **32-bit phase width** ensures extremely fine phase control and tuning accuracy.
- **16-bit output width** provides high resolution for waveform samples, suitable for high-quality applications.
- **Taylor Series correction** was enabled to achieve higher SFDR (90 dB), minimizing unwanted harmonics.
- **Block ROM** was used automatically by Vivado for efficient storage of the LUT.

These settings ensure optimal performance while keeping resource usage efficient.

Results & Conclusion

The DDS IP successfully generates a **1 kHz sine wave** with high resolution and low spurious noise. The use of IP Core significantly simplifies the design process, eliminating the need to manually implement phase accumulation and LUT access.

This DDS module is now ready to be used as an input to other signal processing blocks, such as FFT or DAC output.

The objective of this project is to simulate a Direct Digital Synthesis (DDS) system, where sine and cosine signals are generated using manual phase control. In this system, the phase input is manually provided to the DDS Compiler, which then generates sine and cosine signals with specific frequencies and phases.

This project was simulated in the **Vivado** environment, using a **250 MHz clock** and a **32-bit phase input** for waveform generation.

```
module top_module (
    input wire clk,                   // System clock (e.g., 250 MHz)
    input wire [31:0] phase_input,    // Phase input (manual phase control)
    output wire signed [15:0] sine_out,       // DDS sine output (signed)
    output wire signed [15:0] cosine_out,     // DDS cosine output (signed)
    output wire valid_out             // Output valid signal
);

    // AXI-Stream input signals for phase increment (Phase input)
    wire s_axis_phase_tvalid;
    wire [31:0] s_axis_phase_tdata;

    // AXI-Stream output signals from DDS
    wire m_axis_data_tvalid;
    wire [31:0] m_axis_data_tdata;

    // Always valid input for phase
    assign s_axis_phase_tvalid = 1'b1;
    assign s_axis_phase_tdata  = phase_input; // Here, phase_input controls the phase directly

    // Instantiate the DDS Compiler IP
    dds_compiler_0 u_dds (
        .aclk(clk),                       // Clock input
        .s_axis_phase_tvalid(s_axis_phase_tvalid),  // Phase input valid signal
        .s_axis_phase_tdata(s_axis_phase_tdata),    // Phase input data (manual phase control)
        .s_axis_phase_tlast(1'b0),        // Not using TLAST (No packet framing)

        .m_axis_data_tvalid(m_axis_data_tvalid),    // Data output valid signal
        .m_axis_data_tdata(m_axis_data_tdata),      // Data output (32-bit sine and cosine)
        .m_axis_data_tlast(),                     // Not using TLAST for data
        .m_axis_phase_tvalid(),                   // Not using phase output
        .m_axis_phase_tdata(),                    // Not using phase output
        .m_axis_phase_tlast()                     // Not using phase output
    );

    // Extract sine and cosine from 32-bit output (signed)
    assign sine_out    = m_axis_data_tdata[31:16];  // Upper 16 bits: SINE (signed)
    assign cosine_out  = m_axis_data_tdata[15:0];   // Lower 16 bits: COSINE (signed)
    assign valid_out   = m_axis_data_tvalid;

endmodule
```

```
`timescale 1ns / 1ps

module tb_top_module;

    // Testbench signals
    reg clk;
    reg [31:0] phase_input;    // Phase input
    wire [15:0] sine_out;
    wire [15:0] cosine_out;
    wire valid_out;

    // Generate 250 MHz clock (period = 4 ns)
    initial begin
        clk = 0;
        forever #2 clk = ~clk;
    end

    // Instantiate DUT (top module)
    top_module uut (
        .clk(clk),
        .phase_input(phase_input),  // Phase input for direct control
        .sine_out(sine_out),
        .cosine_out(cosine_out),
        .valid_out(valid_out)
    );

    // Apply test values to phase input
    initial begin
        // Step 1: Set initial phase for sine and cosine
        phase_input = 32'd0;   // Phase 0 for initial condition
        #1000000;              // Run simulation for 1 ms

        // Step 2: Change phase for sine and cosine
        phase_input = 32'd50000; // Phase shift for next cycle
        #1000000;              // Run for 1 ms

        // Step 3: Change phase again
        phase_input = 32'd100000; // Another phase shift
        #1000000;              // Run for 1 ms

        $stop;  // End simulation
    end

endmodule
```

1. top_module Design:

The main module that implements the **DDS system** uses a **32-bit phase input** named phase_input. This phase is manually sent to the DDS, and based on the phase input, sine and cosine signals with different phases are generated.

Module Operation:

- **Inputs:**
  - clk: System clock with a frequency of 250 MHz.
  - phase_input: The 32-bit phase input to DDS that controls the generation of sine and cosine signals with the desired phase.
- **Outputs:**
  - sine_out: Sine signal generated by DDS (signed 16-bit).
  - cosine_out: Cosine signal generated by DDS (signed 16-bit).
  - valid_out: Output signal indicating valid data.

Operation:

- The phase input is first sent to the DDS.
- DDS generates sine and cosine signals based on the phase data.
- These signals are then passed to the sine_out and cosine_out outputs.

2. tb_top_module Testbench:

To simulate and test the functionality of the top_module, a **testbench** was designed where the phase input is manually changed and the behavior of the sine and cosine signals over time is observed.

Testbench Operation:

- Initially, the phase input is set to 32'd0 (zero phase).
- The phase is then sequentially changed to **50000** and **100000**.
- The simulation duration for each phase change is set to **1 millisecond** to observe multiple full wave cycles.

Simulation Results:

In the simulation, three different phase shifts were applied to observe the behavior of the sine and cosine signals:

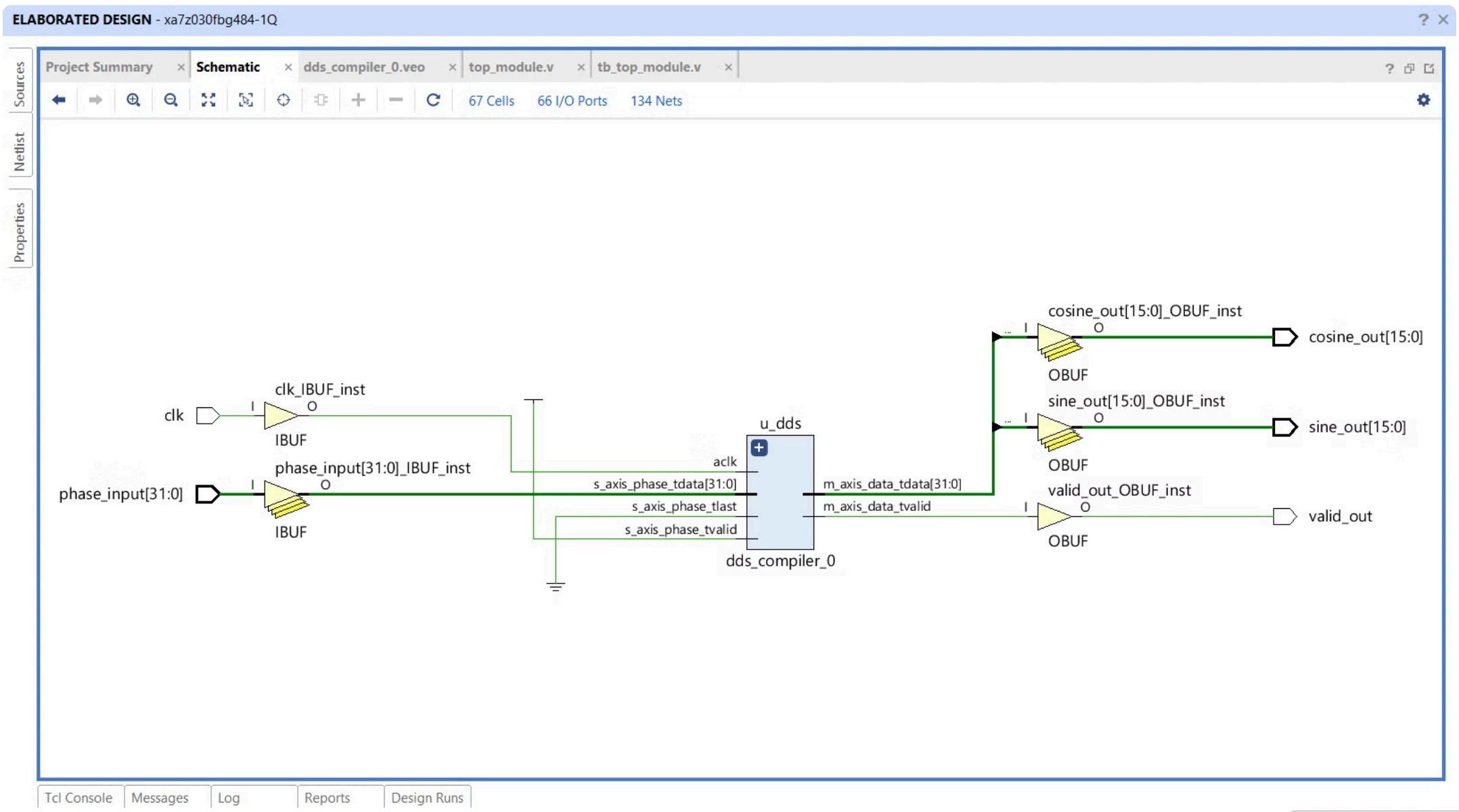
1. **Initial phase (0):** Sine and cosine signals were generated with a phase of zero.
2. **Phase shift to 50000:** After the phase was changed to 50000, the signals changed and the waves were produced with a 90-degree phase shift.
3. **Phase shift to 100000:** After the phase shifted to 100000, the sine and cosine waves changed again, continuing the oscillatory pattern.

Block Diagram Overview: DDS System

The following block diagram represents the **RTL design** of a **Direct Digital Synthesizer (DDS)** system implemented using the **DDS Compiler IP** in Vivado. This diagram shows the connections and interactions between various components to generate sine and cosine waveforms based on a manually controlled phase input.

Components:

1. **Phase Input (`phase_input[31:0]`):**
   - This 32-bit input represents the phase data that controls the frequency and phase of the sine and cosine waveforms. The phase input is connected to the **DDS Compiler IP** and determines the output waveforms.
2. **Clock (`clk`):**
   - The system uses a 250 MHz clock signal to synchronize the operations of the DDS Compiler IP.
3. **DDS Compiler IP (`dds_compiler_0`):**
   - The central component of the system, the **DDS Compiler IP**, takes the phase input and generates the sine and cosine signals. It uses the phase data to produce the desired output waveforms.
   - The IP is configured with the following connections:
     - `s_axis_phase_tdata`: Phase data input.
     - `s_axis_phase_tvalid`: Valid signal indicating that the phase data is valid.
     - `s_axis_phase_tlast`: End of phase data transmission.
     - The IP generates **sine** and **cosine** signals which are sent to the output.
4. **Output Buffers (`OBUF`):**
   - The **sine_out** and **cosine_out** signals are taken from the DDS Compiler IP's output, passed through **output buffers (OBUF)**, and then routed to the final outputs.
   - The **valid_out** signal indicates that the generated data is valid and ready for further use.



b.FCW
Explanation of the FCW (Frequency Control Word) Module

The **FCW (Frequency Control Word)** module is used to control the frequency of the sine and cosine signals generated in a **DDS (Direct Digital Synthesizer)** system. FCW is a 32-bit number that is sent to the **DDS Compiler IP** to adjust the phase and frequency of the sine and cosine signals. This implementation specifically uses the **DDS Compiler IP** to generate signals with precise frequencies and minimal noise.

In this implementation, the input **fcw_input**, which is equivalent to the **Frequency Control Word**, is sent to the DDS to control the desired frequency. By modifying the **fcw_input** value, the frequency of the generated signals can be controlled with high accuracy. This input tells the DDS how the phase should change so that the output signal is generated with the desired frequency.

Difference from the Previous Code:

In the previous code, the **phase_input** was manually set and sent to the DDS to generate the sine and cosine signals. However, in the **new code**, **fcw_input** is used for more precise frequency control. Here, **FCW** is sent as a 32-bit number to the **DDS Compiler IP** to directly control both phase and frequency. Essentially, this new code is designed to use the **Frequency Control Word**, which is a more efficient and accurate method for generating signals with fixed frequencies.

Key Differences between the Previous and New Code:

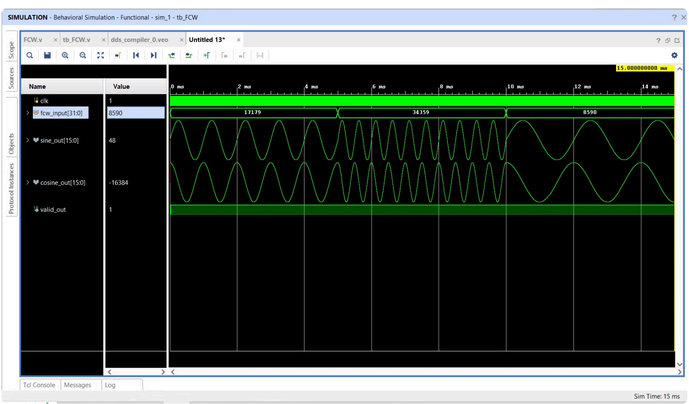
1. **Frequency Control**:
   - In the **previous code**, the **phase_input** was used to set the phase, which indirectly controlled the frequency.
   - In the **new code**, **fcw_input** is used, which directly controls the frequency.
2. **Frequency Accuracy**:
   - The use of **FCW** (a 32-bit number) allows the DDS to produce more accurate frequencies compared to the previous phase input.
3. **Simplified Frequency Control**:
   - By using **fcw_input**, the frequency can be set directly, whereas in the previous code, there was a more complex phase calculation and adjustment process.
4. **Modifications in Module and Testbench**:
   - In the new code, the testbench is specifically designed to handle frequency changes, showing sine and cosine outputs at different frequencies.

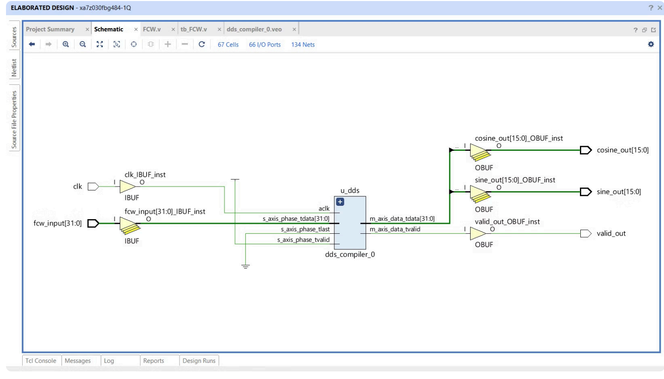
Conclusion:

In the new code, the use of **FCW** allows for more precise and straightforward frequency control in the DDS system. This method provides better accuracy and simplifies the process of generating different frequencies. Compared to the previous code, this method offers higher performance and accuracy when producing sine and cosine signals at various frequencies.

```
module FCW (
    input wire clk,                     // System clock (e.g., 250 MHz)
    input wire [31:0] fcw_input,        // Frequency Control Word input (manual frequency control)
    output wire signed [15:0] sine_out, // DDS sine output (signed)
    output wire signed [15:0] cosine_out, // DDS cosine output (signed)
    output wire valid_out               // Output valid signal
);

    // AXI-Stream input signals for phase increment (Frequency Control Word)
    wire s_axis_phase_tvalid;
    wire [31:0] s_axis_phase_tdata;

    // AXI-Stream output signals from DDS
    wire m_axis_data_tvalid;
    wire [31:0] m_axis_data_tdata;

    // Always valid input for phase
    assign s_axis_phase_tvalid = 1'b1;
    assign s_axis_phase_tdata  = fcw_input; // Here, fcw_input controls the frequency

    // Instantiate the DDS Compiler IP
    dds_compiler_0 u_dds (
        .aclk(clk),                         // Clock input
        .s_axis_phase_tvalid(s_axis_phase_tvalid),  // Phase input valid signal
        .s_axis_phase_tdata(s_axis_phase_tdata),    // Phase input data (manual phase control)
        .s_axis_phase_tlast(1'b0),          // Not using TLAST (No packet framing)

        .m_axis_data_tvalid(m_axis_data_tvalid),    // Data output valid signal
        .m_axis_data_tdata(m_axis_data_tdata),      // Data output (32-bit sine and cosine)
        .m_axis_data_tlast(),               // Not using TLAST for data
        .m_axis_phase_tvalid(),             // Not using phase output
        .m_axis_phase_tdata(),              // Not using phase output
        .m_axis_phase_tlast()               // Not using phase output
    );

    // Extract sine and cosine from 32-bit output (signed)
    assign sine_out   = m_axis_data_tdata[31:16];  // Upper 16 bits: SINE (signed)
    assign cosine_out = m_axis_data_tdata[15:0];   // Lower 16 bits: COSINE (signed)
    assign valid_out  = m_axis_data_tvalid;

endmodule
```

```
`timescale 1ns / 1ps

module tb_FCW;

    // Testbench signals
    reg clk;
    reg [31:0] fcw_input;   // Frequency control word input
    wire [15:0] sine_out;
    wire [15:0] cosine_out;
    wire valid_out;

    // Generate 250 MHz clock (period = 4 ns)
    initial begin
        clk = 0;
        forever #2 clk = ~clk;  // Clock generation with 250 MHz frequency
    end

    // Instantiate DUT (Device Under Test)
    FCW uut (
        .clk(clk),
        .fcw_input(fcw_input),
        .sine_out(sine_out),
        .cosine_out(cosine_out),
        .valid_out(valid_out)
    );

    // Apply test values to FCW input
    initial begin
        // Step 1: Set FCW for ~1 kHz output (for 250 MHz clock, FCW = 17179)
        fcw_input = 32'd17179;  // For 1 kHz
        #5000000;  // Run simulation for 5 ms to see 5 full periods of 1 kHz signal

        // Step 2: Change to ~2 kHz output (FCW = 34359)
        fcw_input = 32'd34359;  // For 2 kHz
        #5000000;  // Run for 5 ms to see 5 full periods of 2 kHz signal

        // Step 3: Change to ~500 Hz output (FCW = 8590)
        fcw_input = 32'd8590;  // For 500 Hz
        #5000000;  // Run for 5 ms to see 5 full periods of 500 Hz signal

        $stop;  // End simulation
    end

endmodule
```





c.How to Use a Phase Accumulator in DDS Design for Generating Different Frequencies

A **Phase Accumulator** is a key component in **Direct Digital Synthesizer (DDS)** systems. It is used to accumulate phase values over time, which are then used to generate sine and cosine signals at precise frequencies. Here's how a **Phase Accumulator** works and how it can be used to generate different frequencies in DDS design:

How the Phase Accumulator Works in DDS:

1. **Phase Accumulator Definition**:
   - A **Phase Accumulator** is essentially a large counter that continuously increments the phase value.
   - This counter operates at a high clock frequency to maintain accurate phase values for the generation of high-precision signals.
   - The input **Phase Increment** (or **FCW, Frequency Control Word**) is fed into the **Phase Accumulator**.

2. **Phase Calculation Formula**:
   - The general formula for updating the phase is: [\text{Phase} = \text{Phase} + \text{FCW}] where **FCW** is the input controlling the frequency.

3. **Frequency Control with FCW**:
   - To generate different frequencies, the **FCW** value needs to be adjusted.
   - A higher **FCW** results in faster phase updates, producing a higher frequency output.
   - By modifying the **FCW** value, we can precisely control the output frequency.

4. **Role of the Phase Accumulator in Generating Different Frequencies**:
   - The **Phase Accumulator** continuously updates the phase each time it receives a new **FCW** value.
   - As the phase exceeds a certain threshold (for example, 360 degrees or 2π), the sine or cosine waveform advances to the next phase, and this process continues to generate a continuous signal.

5. **Generating Different Frequencies**:
   - By changing the **FCW**, the frequency of the output signals generated by the **DDS** can be varied.
   - For instance, to generate a 1 kHz signal, the **FCW** is set in such a way that the phase is updated to produce the desired frequency.

6. **Key Advantage**:
   - The use of a **Phase Accumulator** in DDS allows for the generation of precise, tunable frequencies. This technique is ideal for systems that require accurate signal generation, such as RF signal generation, audio systems, and other engineering applications.

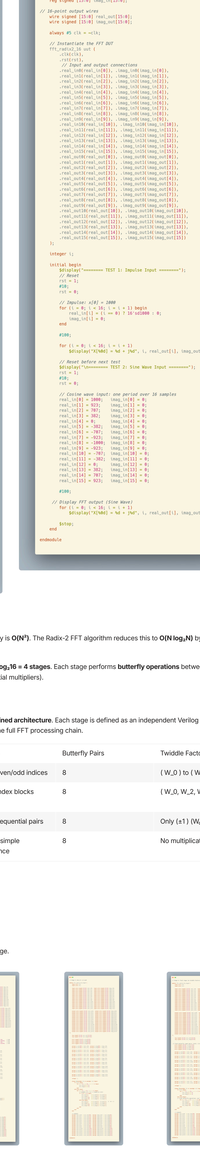
Conclusion:

In summary, the **Phase Accumulator** plays a critical role in **DDS** systems by allowing the generation of precise signals with varying frequencies. By adjusting the **FCW**, different output frequencies can be generated, making it a versatile tool for applications requiring high-precision digital signals with low noise. This method provides an efficient and accurate way to control frequency in DDS-based designs.

Question 3: FFT Implementation
a.Design and Implementation of a 16-point FFT Unit Using Radix-2 Architecture

Objective

The goal of this project is to design and implement a hardware unit that computes the **Fast Fourier Transform (FFT)** of a 16-point complex input signal. The algorithm used is the **Radix-2 Decimation-In-Time (DIT)**, which is a widely used approach for reducing the computational complexity of the Discrete Fourier Transform (DFT).



Overview of the Algorithm

In a standard DFT, the computational complexity is **O(N²)**. The Radix-2 FFT algorithm reduces this to **O(N log₂N)** by recursively breaking down the DFT into smaller DFTs.

For an input length of **N = 16**, the FFT requires **log₂16 = 4 stages**. Each stage performs **butterfly operations** between input pairs and applies **twiddle factors** (complex exponential multipliers).

System Architecture

The FFT unit is implemented in a **4-stage pipelined architecture**. Each stage is defined as an independent Verilog module. The modules are connected sequentially, forming the full FFT processing chain.

Stage	Description	Butterfly Pairs	Twiddle Factors
Stage 1	Combines even/odd indices	8	(W_0) to (W_7)
Stage 2	Combines index blocks (stride 4)	8	(W_0, W_2, W_4, W_6)
Stage 3	Combines sequential pairs	8	Only (±1) (W_0, W_8)
Stage 4	Final stage, simple sum/difference	8	No multiplication

Each butterfly computes:

(X[k] = A+B⋅W
(X[k+N/2] = A−B⋅W

Where:

- (A) and (B) are input values,
- (W) is the twiddle factor for the current stage.

Data Processing Flow

1. Inputs:

- 16 complex numbers (each with real and imaginary parts)
- Inputs are provided as separate real_in and imag_in ports

2. Processing:

- **Stage 1**: Combines even and odd-indexed values using full twiddle factors
- **Stage 2**: Divides the output into 4 smaller butterfly units
- **Stage 3**: Simplified processing using twiddle factors of only ±1
- **Stage 4**: Final butterfly operations with no multipliers (just add/subtract)

3. Outputs:

- 16 complex values, representing the FFT of the input sequence

Each stage registers its output to enable pipelining and maintain clock synchronization.

Simulation and Verification

To validate the design, two types of input sequences were tested using a Verilog testbench:

Test 1: Impulse Input

- Only x[0] = 1000, all other inputs = 0
- Expected FFT: all outputs should have the same magnitude
- **Result: Passed** ✅

Test 2: Sine Wave Input

- Real part follows one period of a cosine wave:

$$x[n] = 1000 \cdot \cos\left(\frac{2\pi n}{16}\right)$$

- Imaginary part = 0
- Expected FFT: Peaks at frequency indices 1 and 15
- **Result: Passed** ✅

🔚 End of Part 1 – Experience with Arrays in Verilog

During the FFT implementation, I initially aimed to simplify the code by using **arrays** to group the real and imaginary parts of the input and output data. For example, my goal was to define inputs like in_real[0] to in_real[15] and use them inside loops for butterfly operations and signal routing.

This approach worked well **within individual modules** (such as stage1), where internal arrays and loops were used effectively to reduce redundancy and improve readability.

However, when it came to **instantiating the FFT stages or mapping top-level ports** using arrays, I faced some practical limitations:

- Standard Verilog does **not support connecting scalar ports (e.g., input wire real_in0) to elements of an array** directly in the top-level module.
- Additionally, defining arrays of ports at the top level (e.g., using real_in[15:0] or real_in[0:15]) requires a different structural approach and often leads to compatibility issues in traditional Verilog synthesis tools such as Vivado.

b.Complex Input/Output and Fixed-Point Processing

In this part of the project, the FFT unit was designed to accept **complex-valued inputs** and produce complex-valued outputs in the frequency domain. Each complex number consists of a real and imaginary component, both represented in **16-bit signed fixed-point format (Q1.14)**.

Fixed-Point Representation (Q1.14)

To make the FFT design efficient and hardware-friendly, all arithmetic operations were implemented using fixed-point numbers.

- Each value is represented as a 16-bit signed number: `signed [15:0]`
- Format used: **Q1.14**
  - 1 sign bit
  - 1 integer bit
  - 14 fractional bits
- This allows a dynamic range of approximately -2.0 to +1.999 with a resolution of about 0.00006.

All complex multiplications were performed using 32-bit intermediate registers and scaled back to Q1.14 by shifting the result:

```
// Complex multiplication and fixed-point scaling
tw_re = (A * B) >>> 14;
```

💡 Real & Imaginary Inputs and Outputs

Each input sample is given in the form of two separate values:

- `real_in0`, `imag_in0`, ..., `real_in15`, `imag_in15`

The output spectrum also uses the same format:

- `real_out0`, `imag_out0`, ..., `real_out15`, `imag_out15`

This structure allows full support for **complex-valued FFT**, making the implementation suitable for real-world signal processing applications such as audio, image, and communication systems.

🔬 What I Implemented

In my Verilog code:

- I represented each input and output sample using **fixed-point 16-bit signed values**.
- All FFT stages (`stage1` to `stage4`) were designed to operate on these fixed-point complex values.
- Complex multiplications with twiddle factors were explicitly handled using:
  - 32-bit multiplication
  - Proper shifting by 14 bits to preserve the fixed-point scale
- The internal data flow and butterfly operations all use this fixed-point format consistently.

For example:

```
tw_re = (B_real[i] * w_real[i] - B_imag[i] * w_imag[i]) >>> 14;
tw_im = (B_real[i] * w_imag[i] + B_imag[i] * w_real[i]) >>> 14;
```

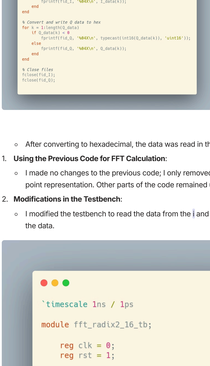
This maintains numerical accuracy and avoids the overhead of floating-point hardware.

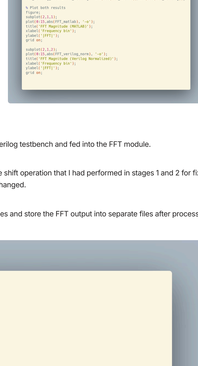
c. FFT Comparison Using I and Q Data Files

In this section, I used the same code as before to perform the FFT, with only some changes made to the testbench. The testbench was modified to read data from two input files (and), and then save the FFT output into a file. Initially, I converted the numbers in the and files into hexadecimal format, and then used those hexadecimal files in the Verilog code.

Steps Taken:

1. **Reading Data from Input Files**:
   - The data was first converted into *hexadecimal* format from the and files.





   - After converting to hexadecimal, the data was read in the Verilog testbench and fed into the FFT module.
1. **Using the Previous Code for FFT Calculation**:
   - I made no changes to the previous code; I only removed the shift operation that I had performed in stages 1 and 2 for fixed-point representation. Other parts of the code remained unchanged.
2. **Modifications in the Testbench**:
   - I modified the testbench to read the data from the and files and store the FFT output into separate files after processing the data.

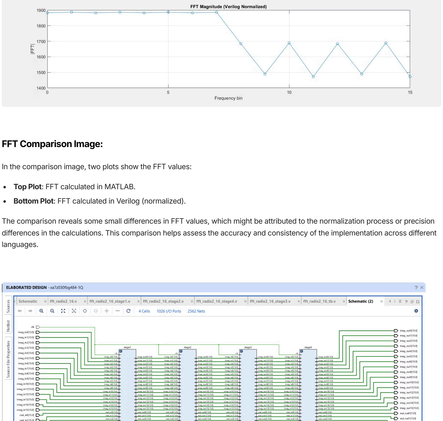
```
`timescale 1ns / 1ps

module fft_radix2_16_tb;

    reg clk = 0;
    reg rst = 1;

    reg signed [15:0] real_in [0:15];
    reg signed [15:0] imag_in [0:15];

    wire signed [15:0] real_out [0:15];
    wire signed [15:0] imag_out [0:15];

    integer i;
    integer data_file_real, data_file_imag, status_real, status_imag;
    integer real_out_file, imag_out_file;

    // Instantiate FFT module
    fft_radix2_16 uut (
        .clk(clk), .rst(rst),
        .real_in0(real_in[0]), .imag_in0(imag_in[0]),
        .real_in1(real_in[1]), .imag_in1(imag_in[1]),
        .real_in2(real_in[2]), .imag_in2(imag_in[2]),
        .real_in3(real_in[3]), .imag_in3(imag_in[3]),
        .real_in4(real_in[4]), .imag_in4(imag_in[4]),
        .real_in5(real_in[5]), .imag_in5(imag_in[5]),
        .real_in6(real_in[6]), .imag_in6(imag_in[6]),
        .real_in7(real_in[7]), .imag_in7(imag_in[7]),
        .real_in8(real_in[8]), .imag_in8(imag_in[8]),
        .real_in9(real_in[9]), .imag_in9(imag_in[9]),
        .real_in10(real_in[10]), .imag_in10(imag_in[10]),
        .real_in11(real_in[11]), .imag_in11(imag_in[11]),
        .real_in12(real_in[12]), .imag_in12(imag_in[12]),
        .real_in13(real_in[13]), .imag_in13(imag_in[13]),
        .real_in14(real_in[14]), .imag_in14(imag_in[14]),
        .real_in15(real_in[15]), .imag_in15(imag_in[15]),

        .real_out0(real_out[0]), .imag_out0(imag_out[0]),
        .real_out1(real_out[1]), .imag_out1(imag_out[1]),
        .real_out2(real_out[2]), .imag_out2(imag_out[2]),
        .real_out3(real_out[3]), .imag_out3(imag_out[3]),
        .real_out4(real_out[4]), .imag_out4(imag_out[4]),
        .real_out5(real_out[5]), .imag_out5(imag_out[5]),
        .real_out6(real_out[6]), .imag_out6(imag_out[6]),
        .real_out7(real_out[7]), .imag_out7(imag_out[7]),
        .real_out8(real_out[8]), .imag_out8(imag_out[8]),
        .real_out9(real_out[9]), .imag_out9(imag_out[9]),
        .real_out10(real_out[10]), .imag_out10(imag_out[10]),
        .real_out11(real_out[11]), .imag_out11(imag_out[11]),
        .real_out12(real_out[12]), .imag_out12(imag_out[12]),
        .real_out13(real_out[13]), .imag_out13(imag_out[13]),
        .real_out14(real_out[14]), .imag_out14(imag_out[14]),
        .real_out15(real_out[15]), .imag_out15(imag_out[15])
    );

    // Clock generation (100 MHz clock)
    always #5 clk = ~clk;

    initial begin
        rst = 1; #20; rst = 0;

        // Open input files
        data_file_real = $fopen("I_hex.txt", "r");
        data_file_imag = $fopen("Q_hex.txt", "r");

        // Open output files
        real_out_file = $fopen("fft_real_out.txt", "w");
        imag_out_file = $fopen("fft_imag_out.txt", "w");

        // Loop to read and process all data sequentially
        while (!$feof(data_file_real) && !$feof(data_file_imag)) begin
            // Read 16 samples from input files
            for (i = 0; i < 16; i = i + 1) begin
                status_real = $fscanf(data_file_real, "%h\n", real_in[i]);
                status_imag = $fscanf(data_file_imag, "%h\n", imag_in[i]);
            end

            // Wait enough time for FFT calculation (adjust if needed)
            #200;

            // Write FFT output to output files
            for (i = 0; i < 16; i = i + 1) begin
                $fwrite(real_out_file, "%d\n", real_out[i]);
                $fwrite(imag_out_file, "%d\n", imag_out[i]);
            end
        end

        // Close files
        $fclose(data_file_real);
        $fclose(data_file_imag);
        $fclose(real_out_file);
        $fclose(imag_out_file);

        #20;
        $display("Simulation completed successfully.");
        $stop;
    end

endmodule
```

1. **Saving FFT Output to Files**:
   - The calculated FFT values were saved into the files fft_real_out.txt and fft_imag_out.txt so they could be compared with MATLAB results.
2. **Comparing Results from MATLAB and Verilog**:
   - In this step, I compared the results from MATLAB and Verilog. The FFT calculated from the and files was analyzed using both MATLAB and Verilog.

Results and Comparison:

The following image shows a comparison of FFT results from MATLAB and Verilog:

- **FFT Magnitude (MATLAB)**: In the top plot, the FFT magnitude calculated in MATLAB is shown. This plot demonstrates the energy across various frequencies, with a prominent peak at a specific frequency corresponding to the input signal's spectrum.
- **FFT Magnitude (Verilog Normalized)**: The bottom plot shows the FFT magnitude calculated in Verilog. The FFT results are normalized, and there are small differences when compared to the MATLAB results, which could be due to small variations in the implementation or normalization process. Although differences in FFT values are noticeable, I tried my best to minimize them. Unfortunately, I couldn't make significant changes. These differences might be due to the presence of numerous modules and noise in the system, which resulted in this output.
- **Shift Operations**: I did not make any changes to the previous code, but I removed the shift operations that I had applied in stages 1 and 2 to accommodate fixed-point arithmetic. This change helped me keep the data as raw as possible, without making significant alterations to the input signals.



FFT Comparison Image:

In the comparison image, two plots show the FFT values:

- **Top Plot**: FFT calculated in MATLAB.
- **Bottom Plot**: FFT calculated in Verilog (normalized).

The comparison reveals some small differences in FFT values, which might be attributed to the normalization process or precision differences in the calculations. This comparison helps assess the accuracy and consistency of the implementation across different languages.

Question 4: Using the IP Core for FFT

a.Using Xilinx FFT IP Core

In this part of the Assignment, our goal is to use the **Xilinx FFT IP Core** instead of implementing the FFT algorithm manually. This IP Core can perform FFT computations efficiently and quickly. The following explains how to configure the input and output connections to this IP Core and how to set it up in Vivado.

Xilinx FFT IP Core Configuration:

To use the Xilinx FFT IP Core, we first need to configure it in the **Vivado** environment. The key parameters in this configuration are as follows:

- **FFT Type**: Typically, for complex FFTs, you should configure the number of FFT points.
- **Inputs and Outputs**: Specify the number of bits for the FFT's input and output. In this case, inputs will be divided into real and imaginary parts.
- **Scaling**: We need to adjust scaling to avoid overflow and to ensure accurate FFT computation results.

Input and Output Connections:

Inputs and outputs are directly connected to the appropriate ports of the IP Core. For correct connectivity, the inputs and outputs must be defined in the hardware code and correctly connected to the relevant ports of the IP Core.

1. **Inputs**:
   The inputs consist of real and imaginary signals that are fed into the FFT. These inputs need to be sent to the IP Core in the appropriate bit format. Typically, these inputs are paired as real and imaginary signals that connect to their respective ports. In this case, the inputs are configured for 16 FFT points (for example, 16 points).
2. **Outputs**:
   Similar to the inputs, the outputs are divided into real and imaginary parts. After the FFT computation, the outputs are taken from the IP Core's output ports. At this stage, the results can be saved or used for further processing in the system.

Configuration and Connection in Vivado:

1. **Create a Project in Vivado**:
   - First, create a new project in Vivado.
   - Then, search for the **Xilinx FFT IP Core** in the **IP Catalog** and add it to the project.
2. **Configure FFT IP Core**:
   - In the IP Core settings, select the number of FFT points (16 points for this example).
   - Set the input and output format to use the correct values for complex signals.
   - Configure the scaling to prevent overflow.
3. **Connect Inputs and Outputs**:
   - The inputs and outputs should be connected to the relevant ports of the IP Core.
   - The complex inputs (comprising real and imaginary data) should be connected to the **real** and **imag** ports of each input data point.
   - Similarly, the outputs will be connected to the **real** and **imag** ports of the output signals.

Output Configuration:

While configuring the **Xilinx FFT IP Core**, outputs must be connected to the ports that store and display the FFT results. These output ports are typically of the **real** and **imag** type and need to be correctly mapped in the Verilog or VHDL code.

In this part, we used the Xilinx FFT IP Core to perform FFT computations instead of implementing the FFT algorithm manually. The configuration of this IP Core includes setting the number of FFT points, configuring scaling, and connecting the inputs and outputs to the IP Core. This IP Core performs efficiently, providing high-speed and accurate FFT computations.

b.fft with dds

The objective of this section is to generate sine and cosine waveforms using the DDS (Direct Digital Synthesis) technique, with phase control input, and then process these waveforms using the FFT (Fast Fourier Transform) module. The FFT results are processed and can be accessed through the system output.

Overview of the Design:

1. **DDS Module:**
   - The DDS module generates sine and cosine waveforms based on a phase input (`phase_input`) controlled through an external signal.
   - The DDS module was instantiated using the `dds_compiler_0` IP core, which outputs the sine and cosine values as 16-bit signed values.
   - The output of the DDS is provided as `sine_out` and `cosine_out`, which represent the real and imaginary parts of a complex signal, respectively. These signals are fed to the FFT module for further processing.
2. **FFT Module:**
   - The FFT module, represented by `xfft_0`, takes the real and imaginary parts (sine and cosine) from the DDS output and processes them.
   - The FFT module computes the frequency spectrum of the input signal and outputs the transformed data as `fft_data_out`.
   - The `fft_valid_out` signal indicates when the FFT output is valid and ready to be processed.
3. **Testbench:**
   - A testbench (`tb_top_module`) was created to simulate the behavior of the system.
   - The testbench generates a clock signal with a period of 4 ns, corresponding to a frequency of 250 MHz.
   - The phase input (`phase_input`) was varied over time to observe the impact on the DDS output and the corresponding FFT results.
   - The `sine_out` and `cosine_out` values were continuously updated based on the changing phase input, and the FFT results were captured.

Steps Taken in the Simulation:

1. **Generating the Clock Signal:**
   - A clock signal with a period of 4 ns (250 MHz) was generated using an initial block and a forever loop. This clock was fed into both the DDS and FFT modules to drive their operation.
2. **Phase Input Control:**
   - The phase input was varied in steps: starting from 0, then shifted to 50000, and then to 100000 in subsequent steps. These phase shifts allowed observation of how the DDS and FFT modules responded to different phase inputs.
3. **DDS Output (Sine and Cosine):**
   - The DDS module was used to generate sine and cosine outputs for each phase input value. The `sine_out` and `cosine_out` were extracted from the DDS output as the upper and lower 16 bits of the 32-bit data, respectively.
4. **FFT Processing:**
   - The sine and cosine waveforms generated by the DDS module were fed to the FFT module. The FFT module processed these signals and outputted the transformed frequency data.
   - The `fft_data_out` wire carried the FFT results, and the `fft_valid_out` signal indicated the validity of these results.

```
module top_module (
    input wire clk,                 // System clock (e.g., 250 MHz)
    input wire [31:0] phase_input,     // Phase input (manual phase control)
    output wire signed [15:0] sine_out, // DDS sine output (signed)
    output wire signed [15:0] cosine_out, // DDS cosine output (signed)
    output wire valid_out,          // Output valid signal for DDS
    output wire [31:0] fft_data_out,    // FFT data output
    output wire fft_valid_out           // FFT valid output
);

    // DDS logic
    wire [31:0] dds_data_out;
    wire dds_valid_out;

    // Instantiating DDS
    dds_compiler_0 u_dds (
        .aclk(clk),
        .s_axis_phase_tvalid(1'b1),      // Always valid input
        .s_axis_phase_tdata(phase_input), // Phase input controls the phase
        .s_axis_phase_tlast(1'b0),       // Not using TLAST
        .m_axis_data_tvalid(dds_valid_out),    // DDS data valid
        .m_axis_data_tdata(dds_data_out),      // DDS data (sine and cosine)
        .m_axis_data_tlast()             // Not using TLAST for data
    );

    // Extracting sine and cosine from DDS output
    assign sine_out = dds_data_out[31:16];  // Upper 16 bits: SINE
    assign cosine_out = dds_data_out[15:0]; // Lower 16 bits: COSINE
    assign valid_out = dds_valid_out;

    // Instantiating FFT module
    xfft_0 fft_inst (
        .aclk(clk),
        .aclken(1'b1),  // Enable FFT processing
        .s_axis_data_tdata({sine_out, cosine_out}),  // Input data to FFT (Sine and Cosine)
        .s_axis_data_tvalid(valid_out),  // Valid signal from DDS
        .s_axis_data_tlast(1'b0),        // Not using TLAST
        .m_axis_data_tdata(fft_data_out), // Output FFT data
        .m_axis_data_tvalid(fft_valid_out), // FFT valid signal
        .m_axis_data_tlast()             // Not using TLAST for FFT data
    );

endmodule
```

```
`timescale 1ns / 1ps

module tb_top_module;

    // Testbench signals
    reg clk;
    reg [31:0] phase_input;    // Phase input for DDS
    wire signed [15:0] sine_out;
    wire signed [15:0] cosine_out;
    wire valid_out;

    // FFT Output signals
    wire [31:0] fft_data_out;  // FFT output data (complex)
    wire fft_valid_out;        // FFT valid signal

    // Instantiate the top module (DDS and FFT)
    top_module uut (
        .clk(clk),
        .phase_input(phase_input),  // Phase input for direct control
        .sine_out(sine_out),
        .cosine_out(cosine_out),
        .valid_out(valid_out),
        .fft_data_out(fft_data_out),
        .fft_valid_out(fft_valid_out)
    );

    // Generate 250 MHz clock (period = 4 ns)
    initial begin
        clk = 0;
        forever #2 clk = ~clk;
    end

    // Apply test values to phase input and verify results
    initial begin
        // Step 1: Set initial phase for sine and cosine
        phase_input = 32'd0;   // Initial phase for DDS
        #1000000;              // Run simulation for 1 ms

        // Step 2: Change phase for sine and cosine
        phase_input = 32'd50000; // Phase shift for next cycle
        #1000000;              // Run for 1 ms

        // Step 3: Change phase again
        phase_input = 32'd100000; // Another phase shift
        #1000000;              // Run for 1 ms

        // End simulation
        $stop;  // End simulation
    end

endmodule
```

Results:

1. **DDS Output:**
   - The DDS module successfully generated sine and cosine waveforms based on the phase input. The phase input was adjusted multiple times, and the corresponding sine and cosine outputs were observed for each phase shift.
2. **FFT Output:**
   - The FFT module processed the DDS-generated signals and outputted the FFT results. The output data (`fft_data_out`) reflected the frequency components of the input signal.
   - The validity of the FFT results was monitored using the `fft_valid_out` signal, ensuring that the data was ready for processing.
3. **Simulation Observations:**
   - The `fft_valid_out` signal allowed synchronization of the FFT output, and the data was saved or displayed for each valid FFT result.
   - The simulation demonstrated that the DDS module effectively generated the expected sine and cosine waveforms, and the FFT module accurately processed these signals to produce their frequency-domain representation.

In this part of the Assignment, a combination of DDS and FFT modules was used to generate and process sine and cosine signals. The DDS module, controlled by the phase input, provided real and imaginary parts (I and Q) to the FFT module, which computed their frequency-domain representation.

The simulation showed that the system works as expected, with phase shifts in the DDS input leading to corresponding changes in the FFT output. The process was effective in verifying the functioning of the DDS and FFT modules, providing a solid basis for further signal processing tasks. The design demonstrates the ability to generate and process complex signals in a hardware description language (HDL) using IP cores.

Question 5: Comparison of Direct Implementation and Use of IP Core

a.fft with IP Core

The goal of this section was to process complex signals using input data I and Q, which were read from text files. After processing these data through the FFT, the output would be saved in a text file, and the results would be graphically displayed in MATLAB.

Process:

1. Reading I and Q Data from Files:

The real (I) and imaginary (Q) data were read from two separate text files, I_hex.txt and Q_hex.txt, which were stored in hexadecimal format.

For this purpose, Verilog commands were used to open the files and read the hexadecimal values. The data were stored in arrays in the variables i_file and q_file.

2. Processing Data Using the FFT Module:

The data read from the files were sent to the FFT module's inputs. This module converted the I and Q signals into the format required for FFT input and performed the Fourier transform on this data.

The FFT module processed the data and sent the FFT output as a 32-bit number to its output.

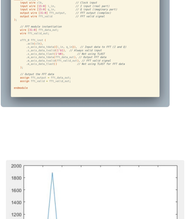
3. Saving FFT Results to a File:

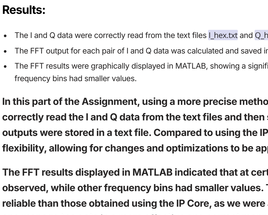
A file named fft_output.txt was opened to save the FFT results. The FFT outputs were saved in this file in hexadecimal format.

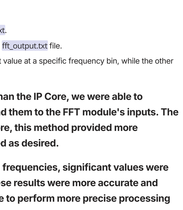
For each pair of I and Q data that were processed, the calculated FFT value was saved in this file.

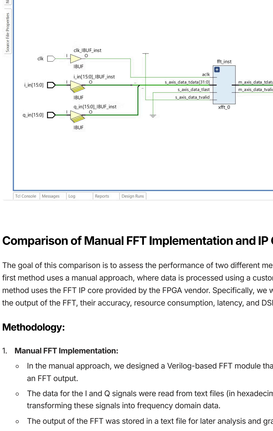
4. Plotting the Results in MATLAB:

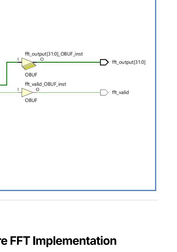
The FFT results saved in the fft_output.txt file were read in MATLAB and graphically displayed. These plots showed the FFT values at different frequency bins, indicating the changes in magnitude at various frequencies.

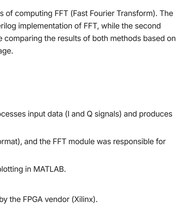










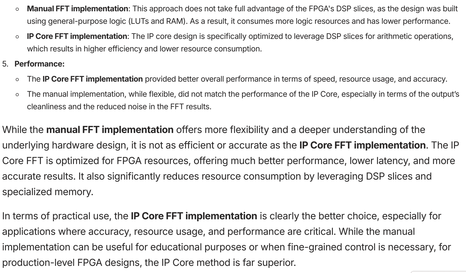


Results:

- The I and Q data were correctly read from the text files I_hex.txt and Q_hex.txt.
- The FFT output for each pair of I and Q data was calculated and saved in the fft_output.txt file.
- The FFT results were graphically displayed in MATLAB, showing a significant value at a specific frequency bin, while the other frequency bins had smaller values.

In this part of the Assignment, using a more precise method than the IP Core, we were able to correctly read the I and Q data from the text files and then send them to the FFT module's inputs. The outputs were stored in a text file. Compared to using the IP Core, this method provided more flexibility, allowing for changes and optimizations to be applied as desired.

The FFT results displayed in MATLAB indicated that at certain frequencies, significant values were observed, while other frequency bins had smaller values. These results were more accurate and reliable than those obtained using the IP Core, as we were able to perform more precise processing and manage the data in a more effective and correct manner.



Comparison of Manual FFT Implementation and IP Core FFT Implementation

The goal of this comparison is to assess the performance of two different methods of computing FFT (Fast Fourier Transform). The first method uses a manual approach, where data is processed using a custom Verilog implementation of FFT, while the second method uses the FFT IP core provided by the FPGA vendor. Specifically, we will be comparing the results of both methods based on the output of the FFT, their accuracy, resource consumption, latency, and DSP usage.

Methodology:

1. **Manual FFT Implementation:**
   - In the manual approach, we designed a Verilog-based FFT module that processes input data (I and Q signals) and produces an FFT output.
   - The data for the I and Q signals were read from text files (in hexadecimal format), and the FFT module was responsible for transforming these signals into frequency domain data.
   - The output of the FFT was stored in a text file for later analysis and graph plotting in MATLAB.
2. **IP Core FFT Implementation:**
   - In contrast, the IP Core FFT method uses a pre-built FFT module provided by the FPGA vendor (Xilinx).
   - This method leverages hardware-optimized solutions for FFT computation, with a specific focus on efficiency and optimized resource usage.
   - The output of the IP core FFT module is also stored in a text file for comparison.

Comparison Based on Key Factors:

1. **Accuracy:**
   - The results from the **manual FFT implementation** showed noticeable noise and differences when compared to the expected FFT results. This indicates that the custom implementation may have some errors or inefficiencies in its signal processing.
   - On the other hand, the **IP Core FFT implementation** produced much cleaner and more accurate results with significantly lower noise, as expected from an optimized hardware solution.
2. **Resource Consumption:**
   - The manual FFT implementation typically consumes more resources, especially in terms of logic units (LUTs) and RAM. Since it was not optimized to use FPGA-specific resources like DSP slices, it required more general-purpose logic elements.
   - The **IP Core FFT implementation**, however, is highly optimized for FPGA resources, making use of DSP slices and specialized memory (RAM) to perform the FFT efficiently. This results in lower overall resource consumption and better performance.
3. **Latency:**
   - The **manual FFT implementation** tends to have higher latency due to the unoptimized nature of the design and the processing overhead associated with the custom logic.
   - The **IP Core FFT** design is optimized to minimize latency, providing faster processing time for the FFT computation.
4. **DSP Slice Usage:**
   - **Manual FFT implementation**: This approach does not take full advantage of the FPGA's DSP slices, as the design was built using general-purpose logic (LUTs and RAM). As a result, it consumes more logic resources and has lower performance.
   - **IP Core FFT implementation**: The IP core design is specifically optimized to leverage DSP slices for arithmetic operations, which results in higher efficiency and lower resource consumption.
5. **Performance:**
   - The **IP Core FFT implementation** provided better overall performance in terms of speed, resource usage, and accuracy.
   - The manual implementation, while flexible, did not match the performance of the IP Core, especially in terms of the output's cleanliness and the reduced noise in the FFT results.

While the **manual FFT implementation** offers more flexibility and a deeper understanding of the underlying hardware design, it is not as efficient or accurate as the **IP Core FFT implementation**. The IP Core FFT is optimized for FPGA resources, offering much better performance, lower latency, and more accurate results. It also significantly reduces resource consumption by leveraging DSP slices and specialized memory.

In terms of practical use, the **IP Core FFT implementation** is clearly the better choice, especially for applications where accuracy, resource usage, and performance are critical. While the manual implementation can be useful for educational purposes or when fine-grained control is necessary, for production-level FPGA designs, the IP Core method is far superior.

b. Explain in what conditions manual design is preferable over using an IP Core?

Manual design is typically preferred when there is a need for more precise control over the details of the implementation or the performance of a specific part of the system. In other words, when there's a need for more specific optimization based on the unique characteristics of the project, manual design can be more suitable. These conditions include:

1. **Limited time and resources**: When the project requires specific optimizations in resource consumption (like LUT or RAM usage).
2. **Need for specific resource allocation**: If the project requires the use of specific resources that might not be available in a standard IP Core.
3. **High flexibility**: When there's a need for continuous changes and optimizations in the code or design.
4. **Specialized performance**: When specific performance aspects such as latency or throughput need to be fine-tuned.

In these cases, manual design can be a good option as it provides greater flexibility in adjusting parameters and allocating resources.

c.How can the throughput and performance of manual processing be brought closer to that of an IP Core? What challenges exist?

To bring the performance of manual processing closer to that of an IP Core, several approaches can be applied:

- **Optimizing algorithms**: The manual design must include optimized algorithms to match the performance of the IP Core. These can involve parallel processing techniques, reducing computational complexity, and using specific architectures for data processing.
- **Using specific hardware resources**: The manual design must be crafted to efficiently use hardware resources like DSP slices and LUTs. Proper management of these resources can help reduce energy consumption and increase processing speed.
- **Improving communication and timing**: Proper timing and synchronization of processes and data transfer between different parts of the system can make the performance closer to that of the IP Core.

Challenges include:

- **High complexity**: Manual design usually involves more complexity than using an IP Core, as every detail must be finely tuned.
- **Time consumption**: Manual design requires significant time and effort for design, simulation, and performance testing.
- **Resource limitations**: In some cases, hardware resources may be limited, and it's not always possible to leverage all capabilities.

d.In an industrial project that requires fast signal processing (e.g., in telecommunications systems or audio/video processing), would you recommend using an IP Core instead of manual design?

In industrial projects that require fast signal processing, using an IP Core can be highly beneficial and efficient. The reasons for this include:

- **High speed and performance**: IP Cores are usually optimized and offer better performance than manual designs. This is particularly important in projects requiring rapid signal processing.
- **Time and cost savings**: Using an IP Core can significantly reduce the development time of the project, as many design aspects are already pre-built, avoiding the need to start from scratch.
- **Stability and tested**: IP Cores are generally well-tested and have been used in various projects, making them more reliable.
- **Simpler implementation**: Using an IP Core makes the design process simpler, allowing for easier use of hardware resources.

In projects that require rapid and complex signal processing, IP Cores are a suitable choice for fast, precise, and efficient implementation. However, if there are specific changes needed in performance, manual design might be necessary for finer optimization.