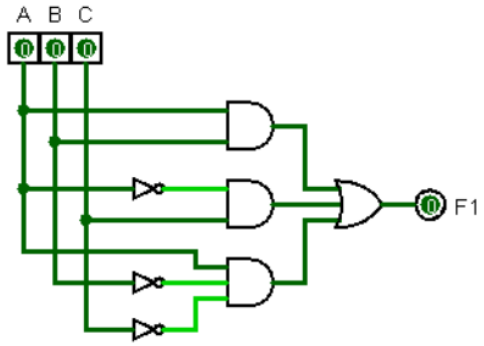
Designing a 4-bit Adder-Subtractor using Vivado IP Integrator

Part I. (Pre-Lab)

In this part of the lab, students will design a digital circuit (Circuit 1) using **behavioral VHDL**, based on the example discussed in the **Introduction to VHDL** lecture. For this task, students are required to:

1. Write a VHDL program to implement the circuit using behavioral modeling.
2. Develop a simple testbench to verify the functionality of the circuit through simulation.

The goal is to ensure that the circuit operates as expected.



$$F_1(A,B,C) = A.B + A'.C + A.B'.C'$$

Circuit 1

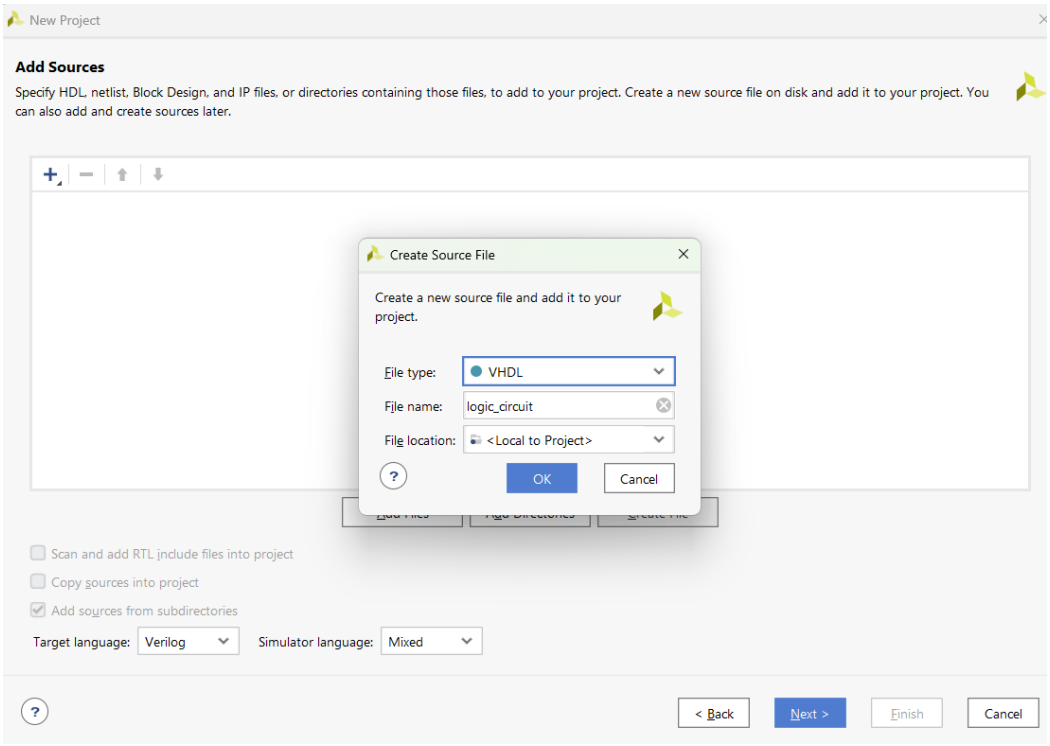
Required Steps: Guide to Design and Simulation

Step 1: Create new Project

In Vivado, Create New project and name the project Lab_2_Part1. Select RTL.

Step 2: Add logic_circuit source

In Add Sources, choose Create File and name the file ***logic_circuit***. At the bottom choose the target language **VHDL**.



Step 3: Write the VHDL Code for the logic_circuit

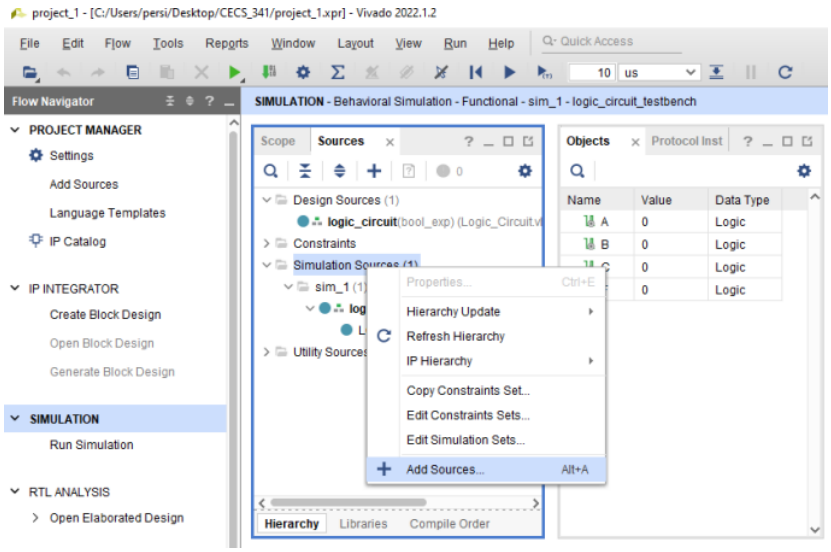
After successfully creating the new project, start writing the source code to implement the given digital circuit in VHDL. You may refer to Slides 35-52 from the Introduction to VHDL lecture for guidance. However, it's a good opportunity to practice writing the code on your own, as slides are mainly for the purpose of conceptual understanding.

Hint: The circuit involves the following logical operations:

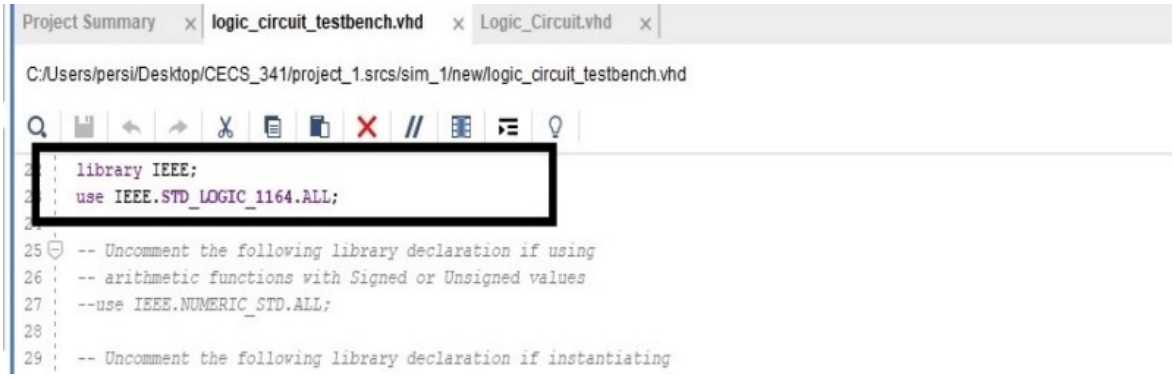
- **AND (•)**
- **OR (+)**
- **NOT (')**

Step 4: Create a New Simulation Source and Testbench

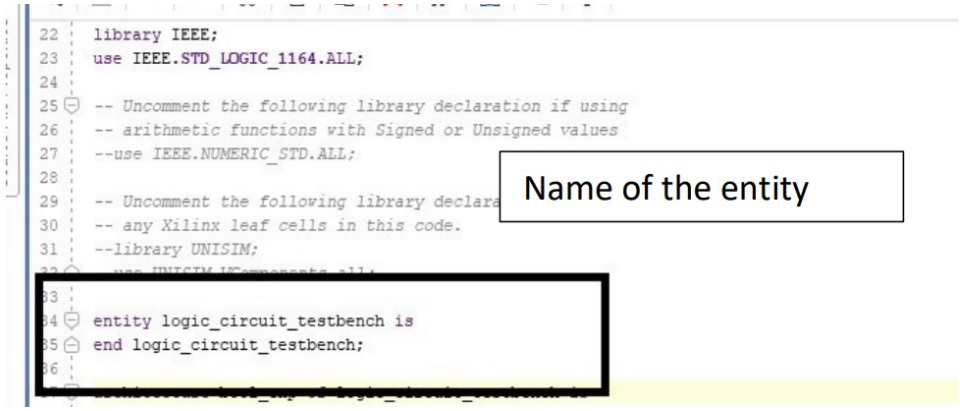
Now, create a new simulation source file and name it `logic_circuit_testbench`. Select **VHDL** as the target language. This file will be used to write the testbench for your logic circuit.



- Ensure that the appropriate libraries are included to support your simulation.



- Since a testbench does not need to interact with external ports, it does not require any port definitions. Ensure the entity is `logic_circuit_testbench` named and then close it.



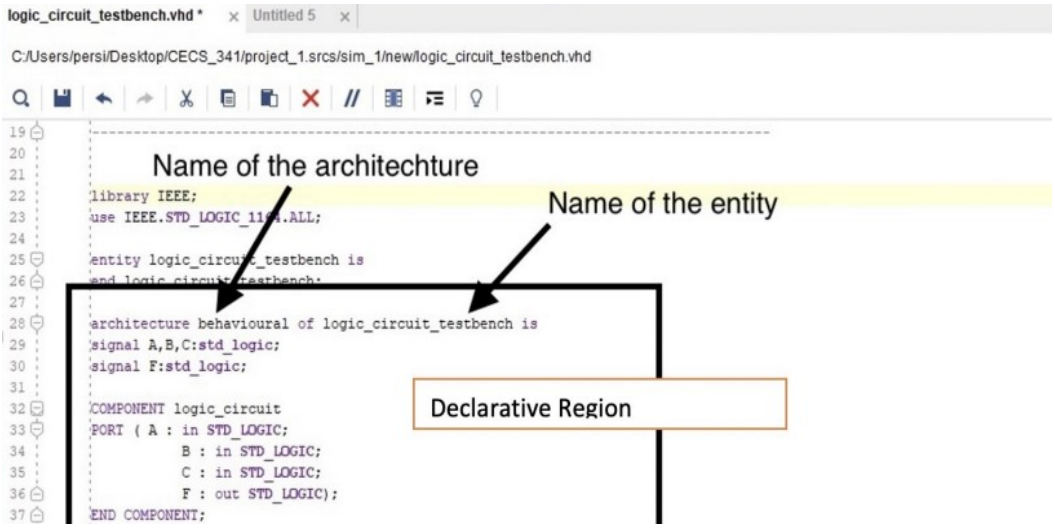
Create the Architecture:

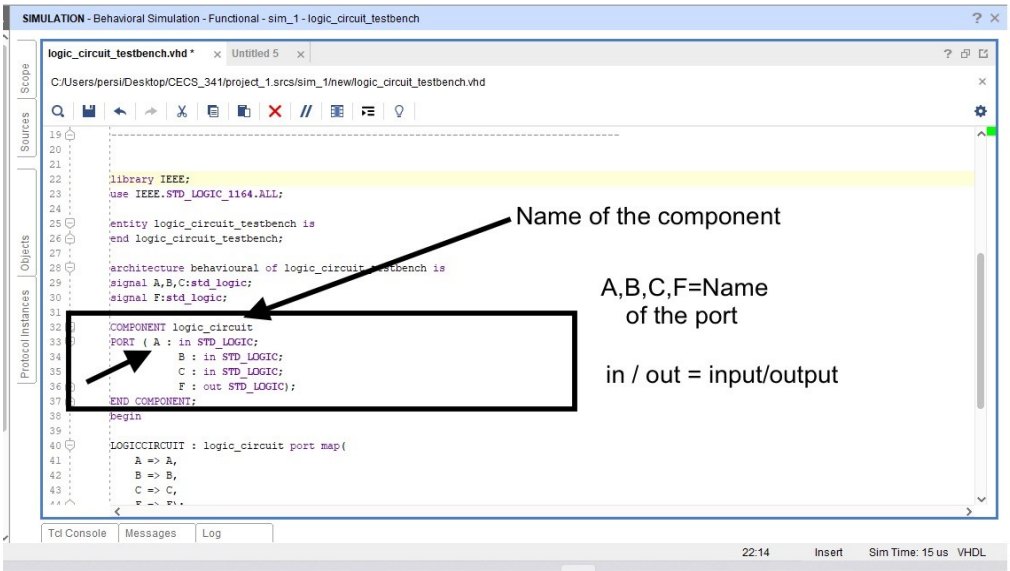
The ARCHITECTURE statement is composed of two parts:

1. Declarative region: Where you can define signals, components, ports, etc.
2. Architectural Body: Contain the actual logic circuit that you're testing and its ports.

1. Declarative Region:

- Define the internal signals needed to test the circuit (A, B, C, and F).
- Declare the component.

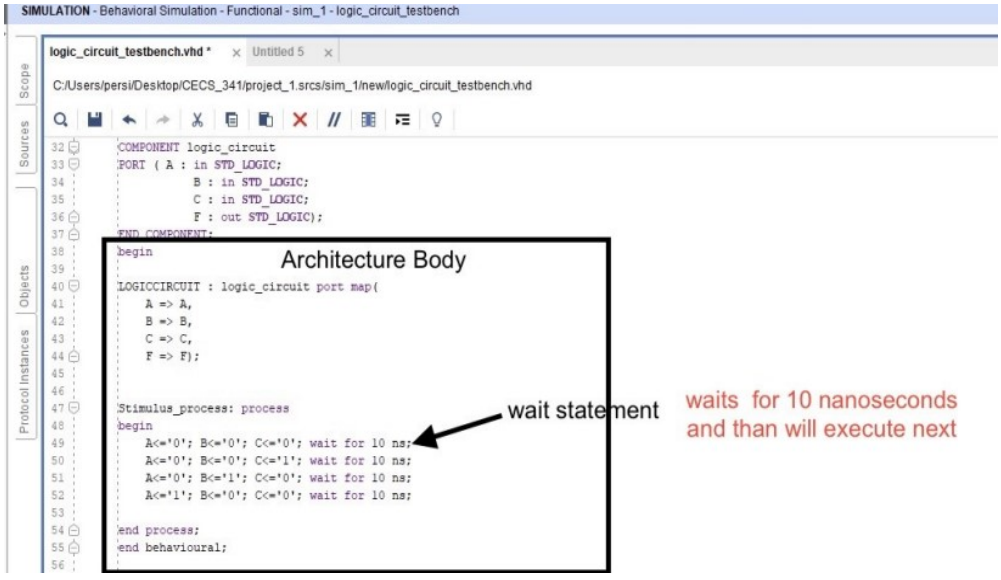




2. Architecture body:

- Use `begin` and `end` keywords followed by the architecture name to begin and end architecture body.
- Instantiate the component (in your logic circuit entity in `logic_circuit.vhd`) by mapping the internal signals (`A`, `B`, `C`, and `F`) to the corresponding ports of the component.
    - Example: `A => A`
    - **Left-hand side:** The left-hand side of `=>` refers to the **port** of the component being instantiated (in this case, the input port `A` of the `logic_circuit.vhd` component).
    - **Right-hand side:** The right-hand side of `=>` (also `A`) refers to the **signal** declared earlier in the architecture of the testbench that you want to connect to that port.
- Create the Stimulus Process (Test Vectors):
    - Write a **process block** to provide input stimulus for testing. This process should apply different test cases (combinations of `A`, `B`, and `C`) and observe the resulting output `F` after each set of inputs.
    - Use the **wait** statement to introduce a delay between each input set, allowing the simulation to capture changes.

      Timeout_clause ::= for time_expression
      wait for 10 ns;

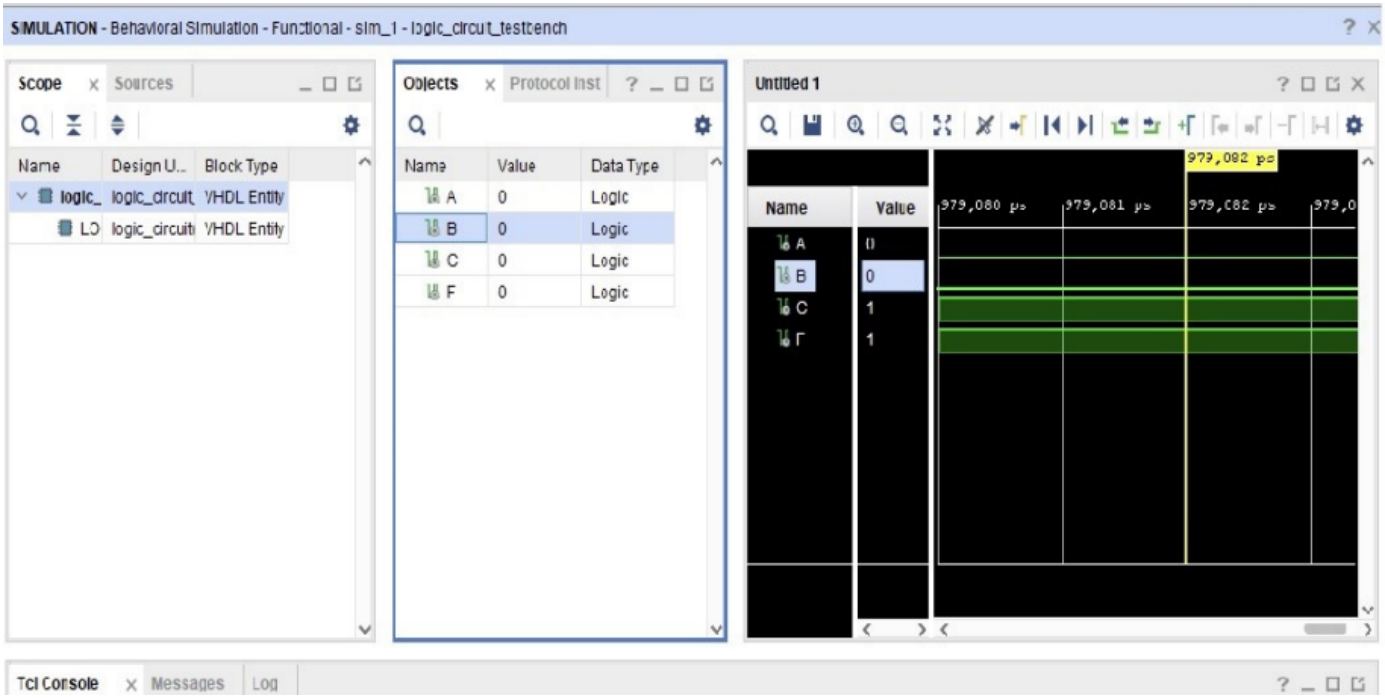


Try testing the following test vectors:

```
A<='0'; B <='0';C<='0';        -- Expected output  F(A,B,C) = 0
A<='0'; B <='0';C<='1';        -- Expected output  F(A,B,C) = 1
A<='0'; B <='1';C<='0';        -- Expected output  F(A,B,C) = 0
A<='0'; B <='1';C<='1';        -- Expected output  F(A,B,C) = 1
A<='1'; B <='0';C<='0';        -- Expected output  F(A,B,C) = 1
A<='1'; B <='0';C<='1';        -- Expected output  F(A,B,C) = 0
A<='1'; B <='1';C<='0';        -- Expected output  F(A,B,C) = 1
A<='1'; B <='1';C<='1';        -- Expected output  F(A,B,C) = 1
```

Step 5: Run Simulation

Now run the behavioral simulation by clicking on “Run Simulation” in the Flow Navigation section and observe the waveform result.



Part II. 4-bit Adder/Subtractor

In this part of the lab, you will extend the concepts learned from **Lab 1**, the **pre-lab tutorial**, and the **Intro to VHDL Lecture** to design a 4-bit adder/subtractor circuit. As we discussed in Lab 1, you need to create 4 full adders, and then use them to construct the larger design (4-bit adder-subtractor) using Vivado IP Integrator. Lastly, you need to develop a testbench to verify your design using simulation results and generated waveforms.

Hints:

- A **4-bit adder/subtractor** can be constructed by combining **4 full adders** (one for each bit) and **4 XOR gates** that allow the circuit to perform both addition and subtraction.
- When performing subtraction, an XOR gate is used to invert the second operand (input) and the circuit adds a "1" to the least significant bit (LSB) to implement two's complement subtraction.

Step 1: Create the Full Adder Components:

Before building the 4-bit adder/subtractor, you need to construct the full adder. You can use the full adder code provided in Lab 1 for this design.

Step 2: Construct the 4-bit Adder/Subtractor

Now that you have your 1-bit full adder component, you can extend it to create the **4-bit adder/subtractor** as follows:

1. Instantiate four full adders and connect them together.
    - Each full adder will handle a single bit of the operation (i.e., one for bit 0, one for bit 1, etc.).
    - The carry-out of each full adder should connect to the carry-in of the next full adder.
2. Incorporate XOR gates for Subtraction:
    - You can add an XOR gate in the IP integrator tool by clicking on the plus (+) sign on top of the IP integrator window and selecting **Utility Vector Logic**. Or you can define the xor gate in VHDL and add it to your design as a block. Or

- Each bit of B will pass through an XOR gate, and the output of the XOR gate will feed into the corresponding full adder.
- When the control signal SUB is '0', the circuit performs addition. When SUB is '1', the XOR gates invert the bits of B, effectively converting the operation to subtraction using two's complement.

3. Block diagram overview:
   - Input A (4 bits): Directly connects to the full adders.
   - Input B (4 bits): Passes through XOR gates before reaching the full adders.
   - Control signal SUB: Controls whether the circuit performs addition (`SUB = 0`) or subtraction (`SUB = 1`).
   - Carry in (C0): The initial carry-in for the first full adder should be SUB, which allows for correct two's complement subtraction.

Note: For a 4-bit adder/subtractor, the result can sometimes have an extra carry-out bit after the final (most significant bit) addition, especially when performing addition with large values or subtraction that involves borrowing.

You can choose to either:

1. Display it as a separate bit `Cout`.
2. Challenge yourself by handling the carry-out bit and making the result a 5-bit number. This will account for overflow in addition or borrowing in subtraction.

Step 3: Create a Testbench

Once your 4-bit adder/subtractor design is complete, you need to develop a **testbench** to verify its functionality. The testbench will simulate the behavior of your circuit by applying different inputs and checking the outputs. Follow instructions from the pre-lab to create your testbench. Ensure that:

1. Test addition by setting `SUB = 0` and applying different values to A and B.
2. Test subtraction by setting `SUB = 1` and applying different values to A and B.
3. Use **wait** statements in the testbench to allow sufficient time for the circuit to compute each output before applying the next test case.
4. Test the following operations, feel free to add additional test cases.

Table 1. Test vectors and expected results

Binary Operation	Expected Result (4-bit)	Expected Carry-Out	Expected Result (HEX)
0000 + 0000			
0000 - 0000			
1111 + 1111			
1111 - 1111			
1100 + 0101			
1100 - 0101			
0111 + 0110			
0111 - 0110			
1111 + 0001			
0000 - 0001			

Step 4: Run the Simulation

- Run the behavioral simulation in Vivado to generate waveforms.
- Observe the outputs for different input combinations and verify that the adder/subtractor is working as expected. Compare waveform results with your hand calculations in Table I.

Lab Deliverables

Submission Instructions:

Submit a zipped file named **Lab2_Group#Number.zip**. The file should include the following:

1. **Pre-lab (Part I)** folder containing all project files for the tested circuit, including any Xilinx-generated files.
2. **Lab 2 (Part II)** project folder for the 4-bit adder-subtractor, including all design files and Xilinx-generated files.
3. A **comprehensive lab report** that clearly explains the steps you followed to develop your design for both Part I and Part II. Please follow the report template provided on Canvas to prepare your final report.
4. Be sure to include the snapshots of your design, waveform, and the completed Table 1 in your report (for Lab 2, Part II).

Only **one submission per group** is required. One student from each group can procced with submission of the lab. Ensure the lab report includes the names of all group members.

Submission due date: Friday, November 21, 11:59 PM.