Experiment 1: Digital system design with VHDL

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | ZYH | **Class** | AI1 |
| **Date** | 2024.11.20 | **Score** |  |
| **Examiner** |  |  |  |

**I. Experiment procedure**

**1.1 Design a 1-bit adder**

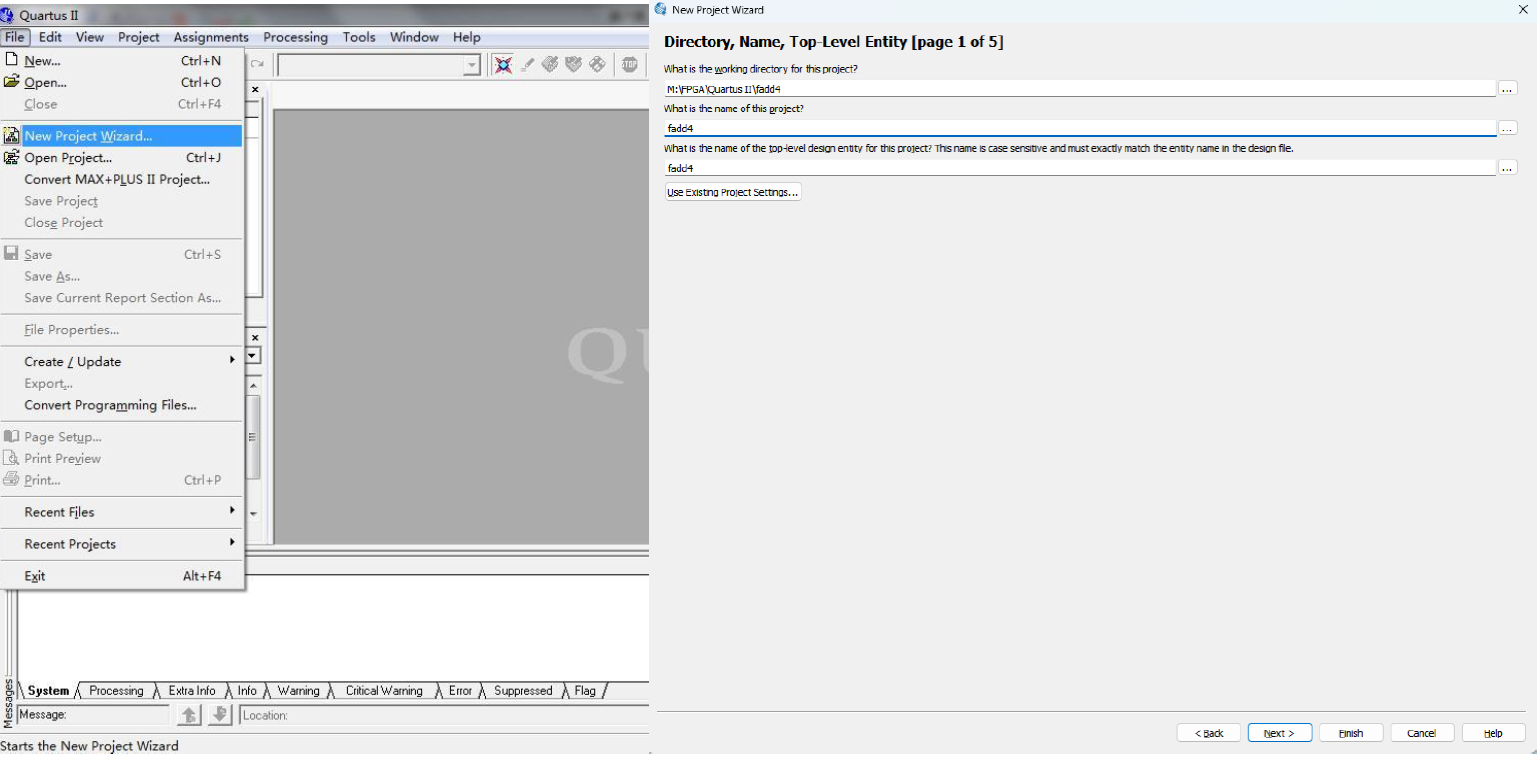


Figure1: Create the project in the Quartus II

To begin, open Quartus II and go to the File menu, then select New Project Wizard.... This will launch a step-by-step wizard to help you set up a new project for your design. On the first page of the wizard, you need to specify the working directory where all project files will be saved, such as M:\FPGA\Quartus II\fadd4 in this case. Next, enter the name of the project, which will also become the folder name under the specified directory, and input the name of the top-level design entity, which should match the name of the entity in your VHDL file exactly, as it is case-sensitive. Once all the required fields are completed, click the Next > button to proceed to the next step of the wizard, where you will continue configuring your project settings.

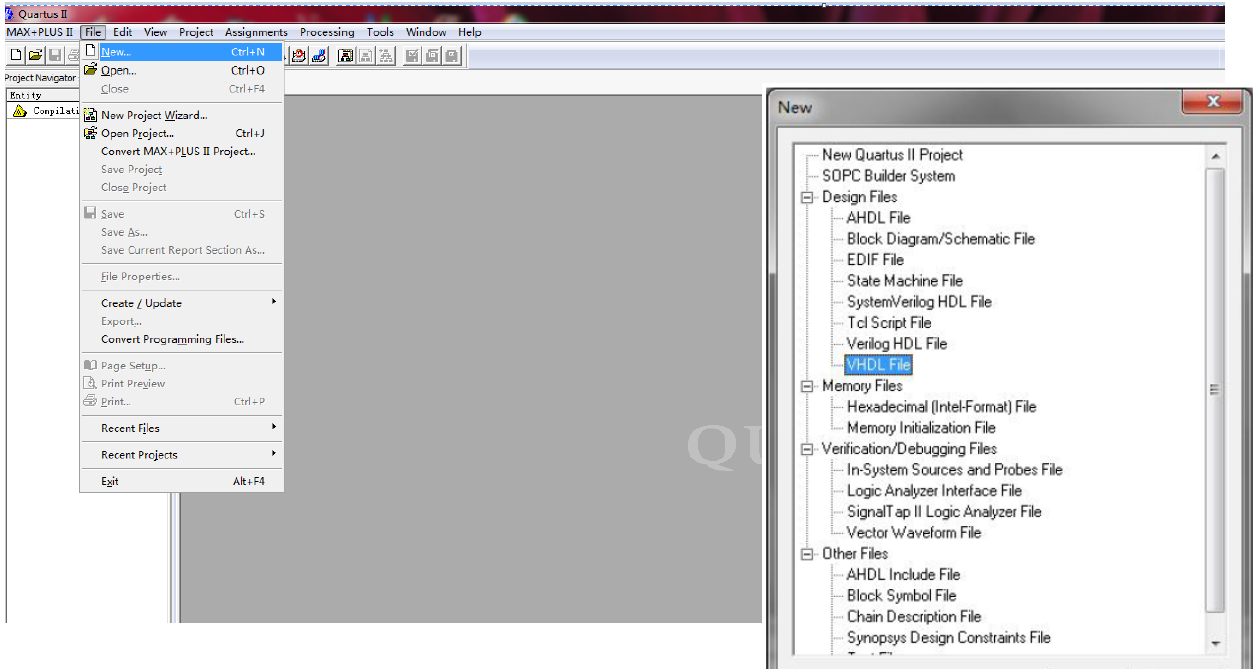


Figure2: Create VHDL file in the Quartus II

In this step, to create a new VHDL file in Quartus II, navigate to the File menu and select New. In the dialog box that appears, under the "Design Files" category, choose VHDL File and click OK. This will open a new editor window where you can write your VHDL code.

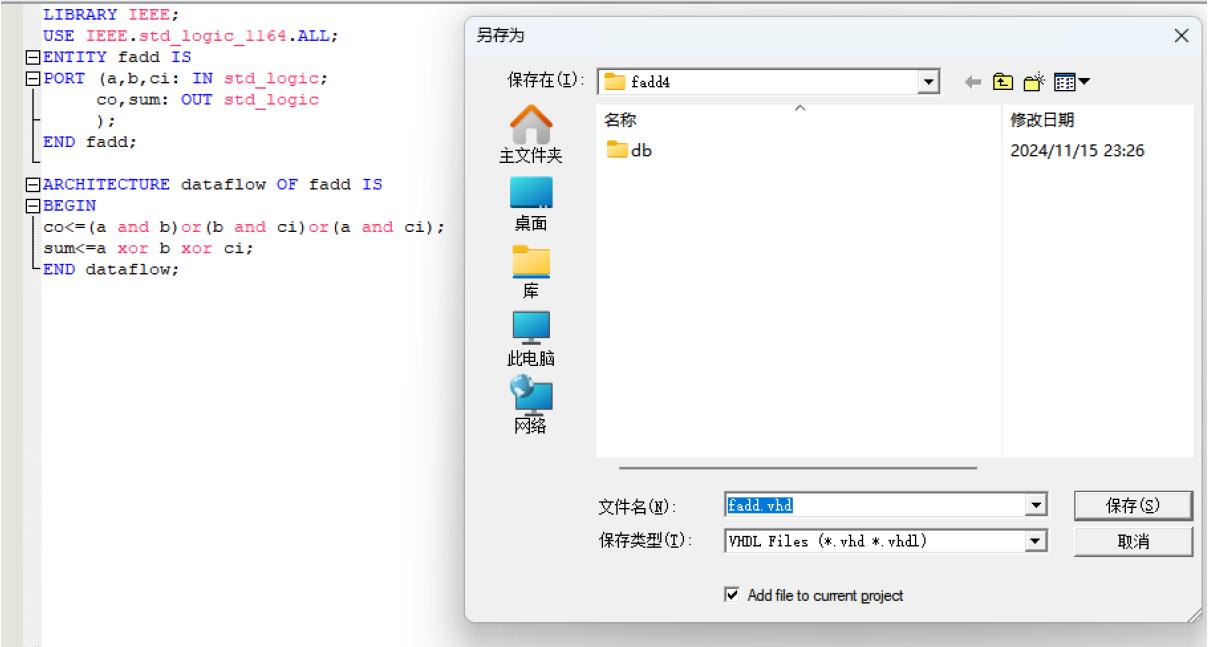


Figure3: Create entity in VHDL file and save as .vhd file

In this step, you write your VHDL code in the editor window, as shown in the left part of the image, defining the ENTITY and ARCHITECTURE sections. Once the code is completed, save the file with the .vhd extension. It is recommended to name the file after the ENTITY name (e.g., fadd.vhd) to maintain consistency and improve the readability of the project. Use the Save As dialog to specify the file name and location, ensuring it is saved within the project's working directory.

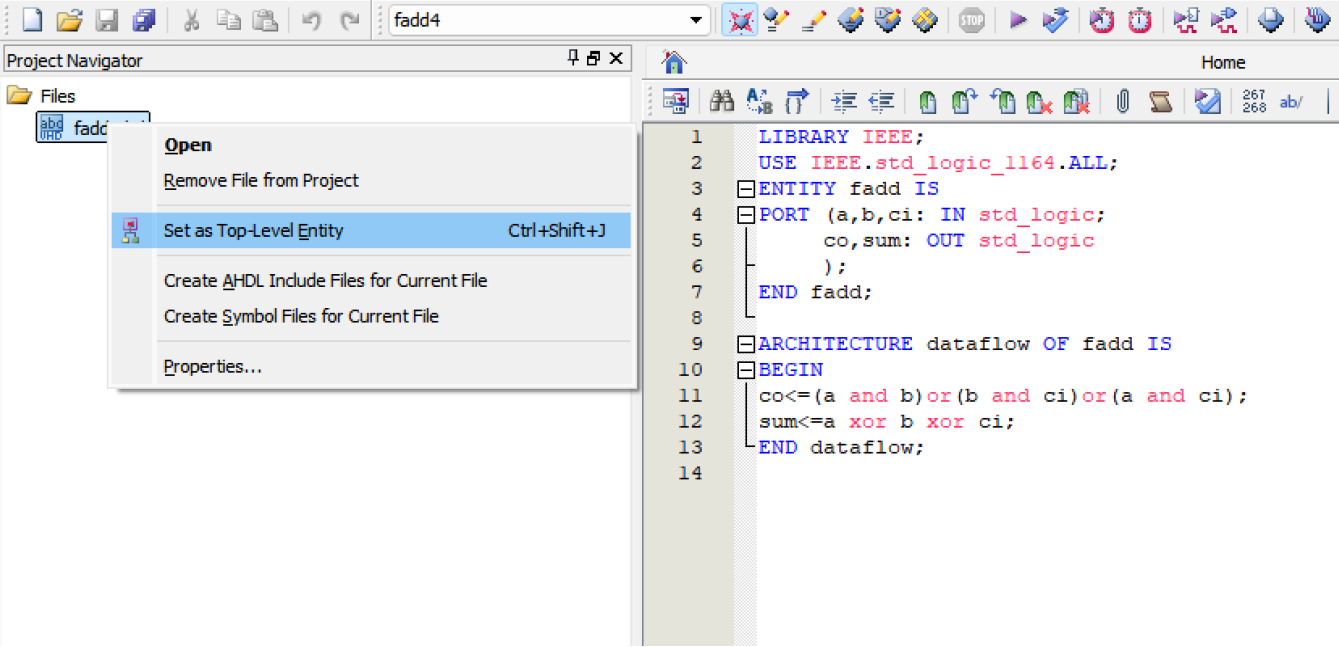


Figure4: Compile VHDL file and set as top-level entity

In this step, after writing and saving your VHDL file, you need to set it as the top-level entity to ensure the compiler recognizes it as the primary module of the project. Right-click on the file in the Project Navigator and select "Set as Top-Level Entity". Once set, click the "Start Compilation" button in the toolbar to compile the design. This process checks for syntax errors and generates the necessary files for simulation and implementation.

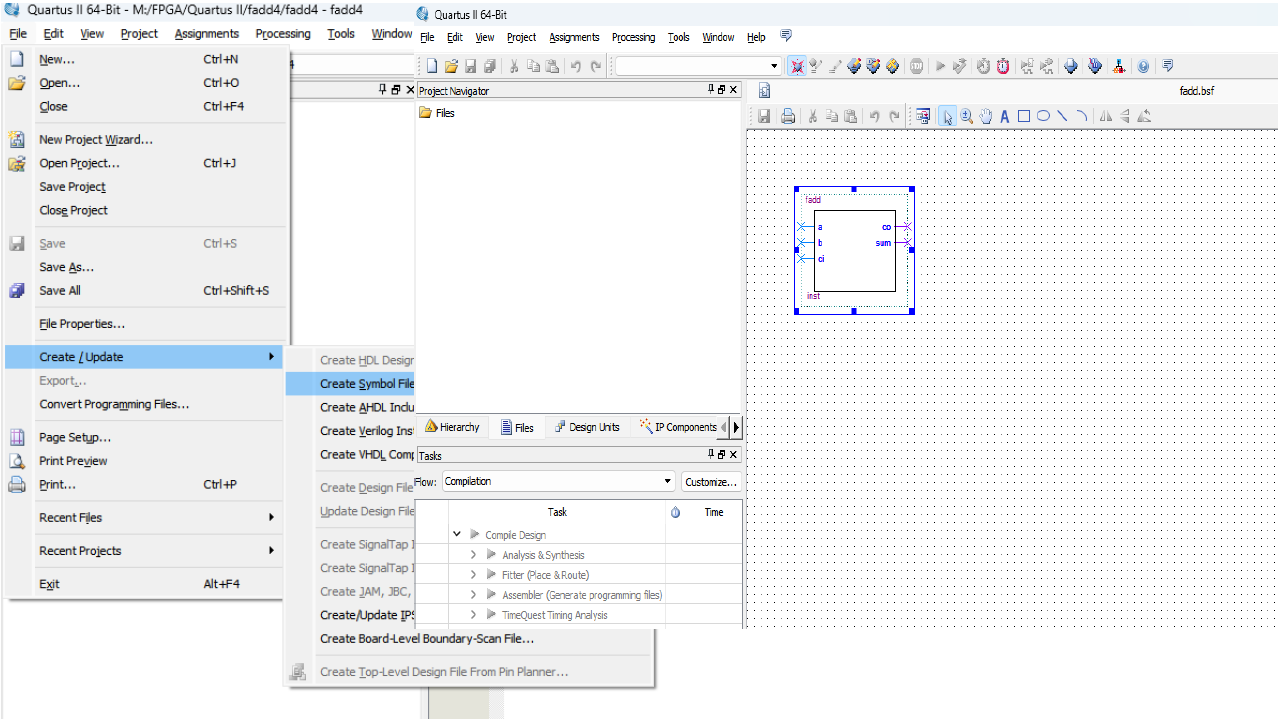


Figure5: Create symbol file for the entity and connect the port

To create a symbolic file in Quartus II, go to the File menu, select Create/Update, and then choose Create Symbol Files for Current File. This generates a Block Symbol File (BSF) for your design, which can be used in schematic designs. Once created, the symbolic representation of your VHDL file will appear, showing input and output ports as a block. This step is useful for integrating your design into higher-level schematic diagrams.

**1.2 Design an example of a 4-bit adder using component.**

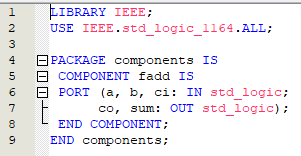


Figure6: Create component module for the 4-bit full adder

This code snippet demonstrates the creation of a component module for a 4-bit full adder. Within a package named components, a component fadd is declared. The component specifies the same interface as the full adder entity, including input ports a, b, and ci of type std\_logic and output ports co and sum of type std\_logic. This component declaration allows the full adder to be instantiated within other modules, enabling modular and reusable design practices.

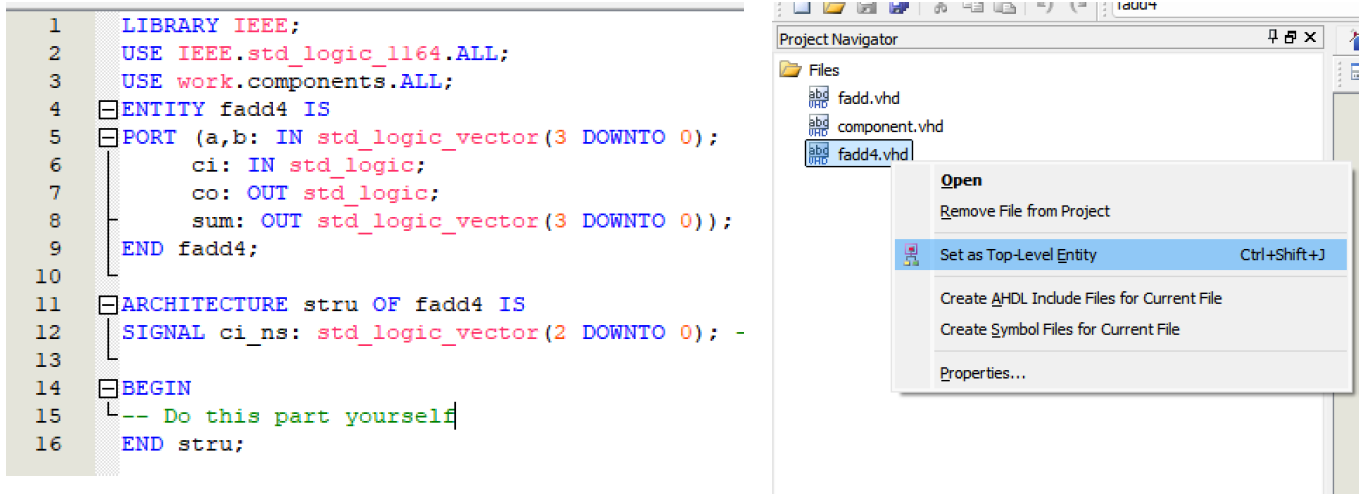


Figure7: Create architecture for the 4-bit full adder

This step demonstrates the design of a 4-bit adder using the previously created fadd component. A new VHDL file, fadd4.vhd, is created, defining an entity named fadd4 with input ports a and b (4-bit vectors) and ci (carry-in), and output ports co (carry-out) and sum (4-bit sum vector). The architecture stru includes signals for internal carry connections (ci\_ns) and uses component instantiation to define the behavior. After writing the code, set fadd4.vhd as the top-level entity by right-clicking it in the Project Navigator and selecting "Set as Top-Level Entity". Then, compile the project to verify the design.

**1.3 Perform the waveform simulation Using Quartus land Modelsim.**

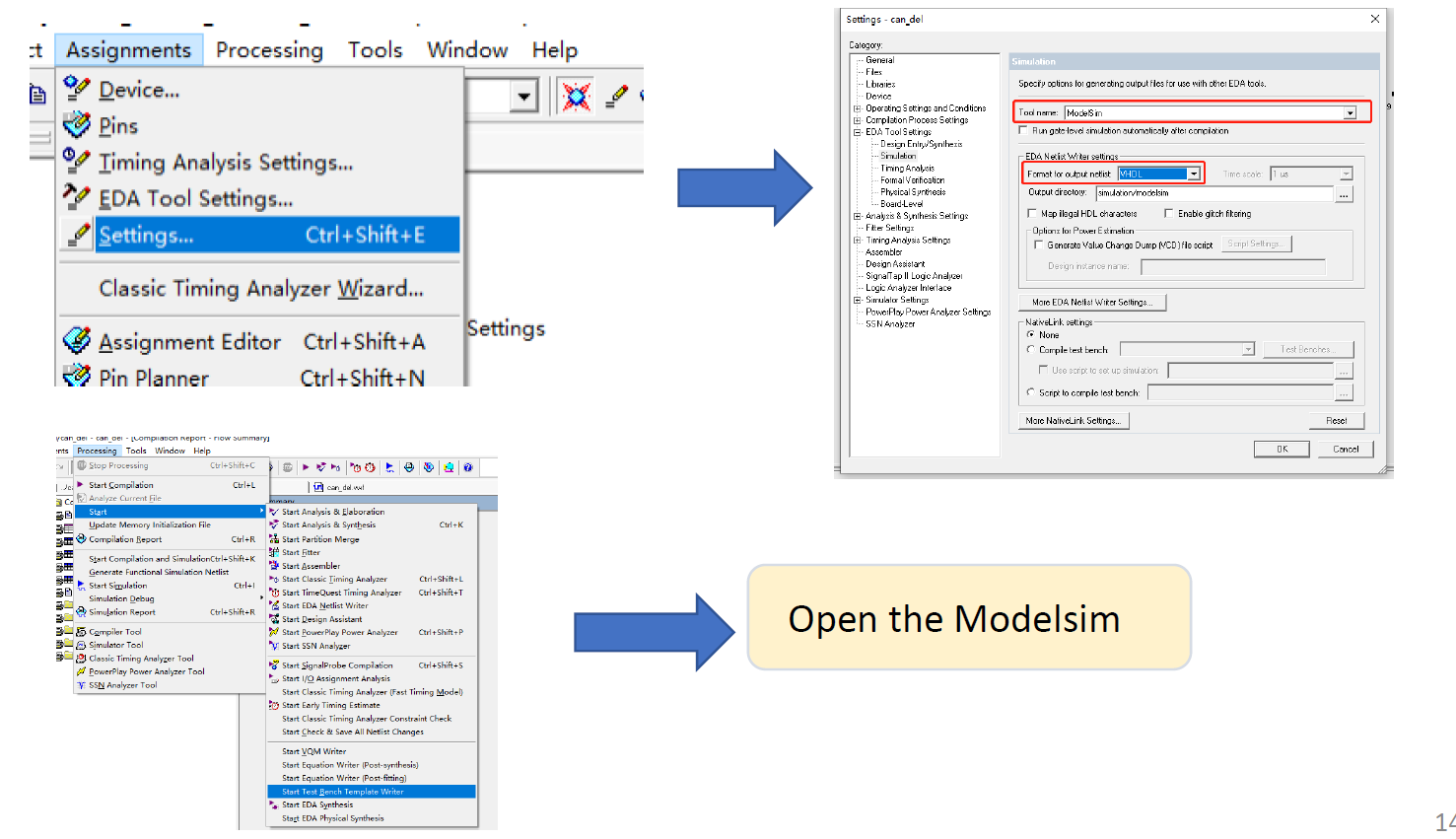


Figure8: Generate related test bench template file

To begin the simulation process in Quartus II, go to the Assignments menu and select Settings. In the Settings dialog box, navigate to the "EDA Tool Settings" section and choose Simulation. Set the tool name to ModelSim and ensure the format for output netlist is set to VHDL. Specify the output directory for simulation-related files and click OK to save the settings. After this, go to the Processing menu and select Start Compilation to ensure your design is compiled and ready for simulation. Finally, open ModelSim to simulate the generated files and verify the functionality of your design.

**1.4 Run the simulation and record the results of Modelsim.**

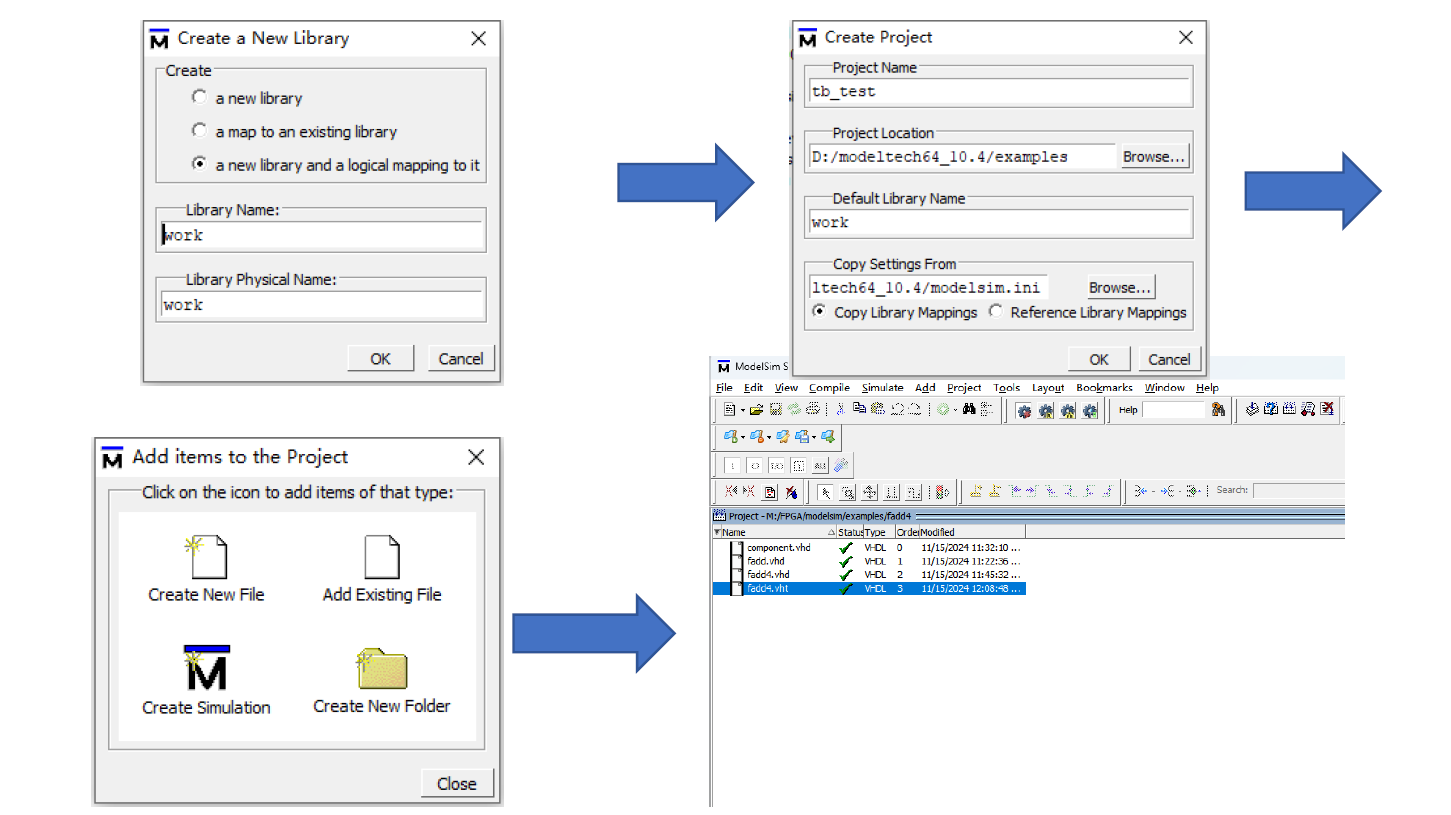


Figure9: Create project in Modelsim project and add VHDL files

The second part of the simulation process involves using ModelSim. First, create a new library by choosing "a new library and a logical mapping to it," naming the library (e.g., work), and clicking OK. Then, create a new project by specifying the project name (e.g., tb\_test), the project location, and the default library name as work. Click OK to proceed. Once the project is created, add files to the project by selecting "Add Existing File" and importing the necessary VHDL files (e.g., fadd.vhd, fadd4.vhd, and the testbench file). The files are added to the workspace, where they can be compiled and used for simulation.

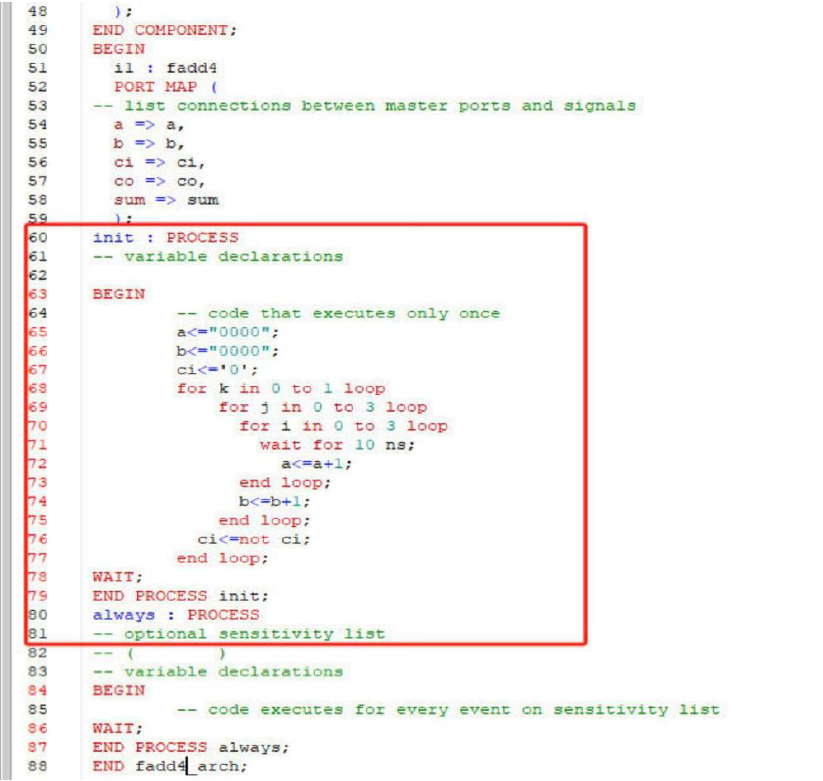


Figure10: Modify fadd4.vht

The testbench includes the necessary library declarations (ieee.std\_logic\_1164 and ieee.std\_logic\_unsigned) and defines the entity fadd4\_vhd\_tst with its architecture. Inside the architecture, signals are declared to mimic the inputs and outputs of the 4-bit adder. A component for fadd4 is instantiated, with its ports mapped to these signals. The testbench includes two processes: init and always. The init process initializes inputs a, b, and ci to 0000 and performs nested loops to increment these values, introducing delays using wait for 10 ns. This simulates various input combinations for functional testing. The always process is an optional placeholder for further sensitivity-based actions. The testbench verifies the behavior of the adder under different input conditions, ensuring correctness through simulation in ModelSim.

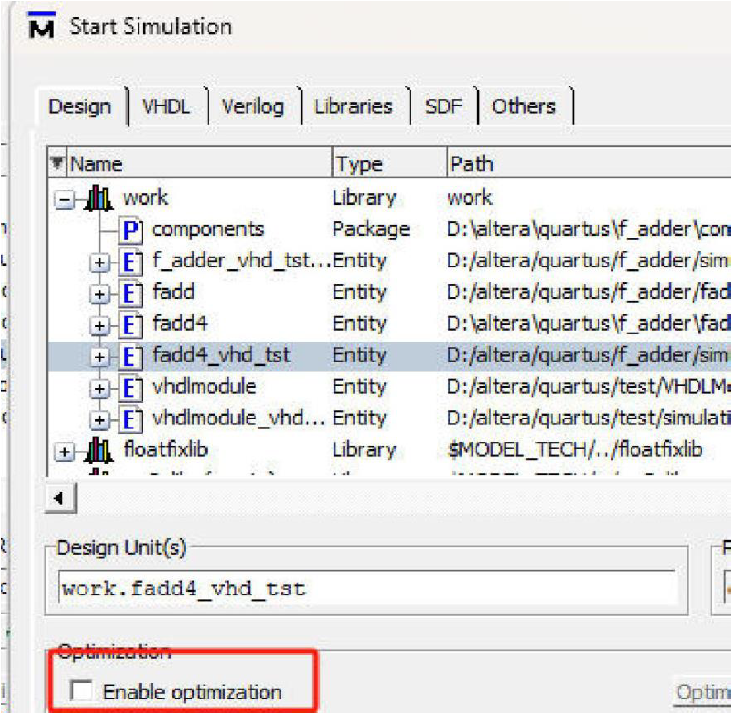
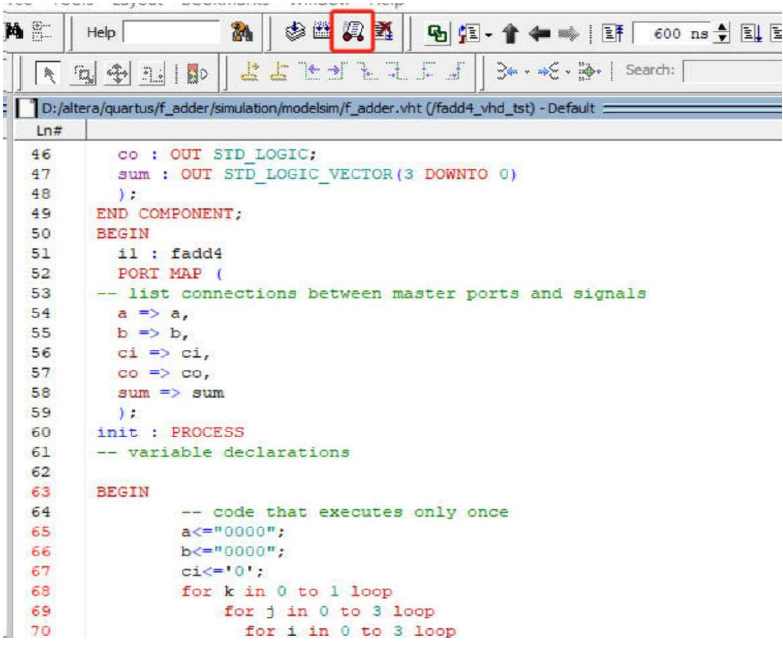


Figure11: Start simulation for the .vht file

To start the simulation for the .vht testbench file in ModelSim, open the Start Simulation window by clicking the simulation icon in the toolbar. In the Start Simulation dialog box, navigate to the work library and select the fadd4\_vhd\_tst entity under the design hierarchy. Ensure that the Enable optimization option is unchecked, as shown in the figure, to avoid issues during the debugging process. Click OK to load the simulation, after which you can view the signals and run the simulation to verify the behavior of the 4-bit adder. This step ensures that the testbench executes correctly, and you can monitor the output waveforms to validate the design's functionality.

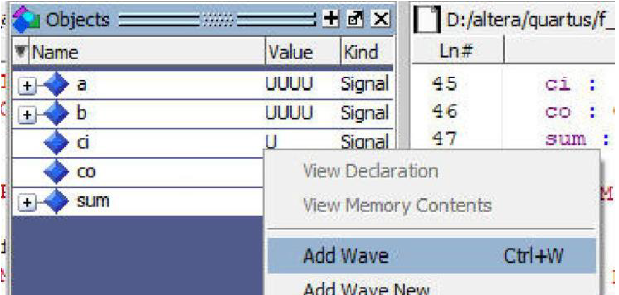
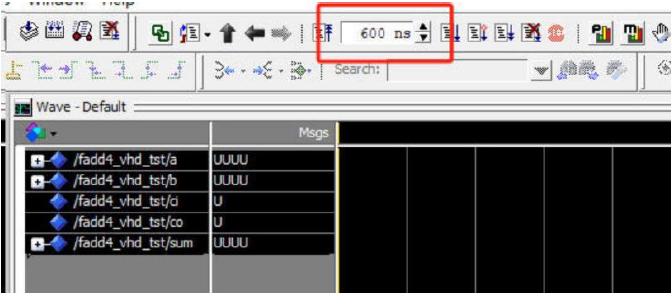
 

Figure12: Add notes and set time for the simulation

To set up and run the simulation, first, add the signals to the waveform viewer. In the Objects window, right-click on the signals such as a, b, ci, co, and sum, and select Add Wave. This action adds the selected signals to the Wave - Default window for visualization. After adding the signals, specify the simulation run time by entering a value (e.g., 600 ns) in the simulation toolbar, as shown in the figure. This defines the duration for which the simulation will execute. Once everything is set up, you can run the simulation to observe the signal transitions and verify the functionality of your design.



Figure13: Simulation results

The simulation results show the waveform of the 4-bit adder testbench, with signals a, b, ci, co, and sum displayed. The inputs a and b increment sequentially in binary, while the carry-in ci toggles as specified in the testbench. The outputs co (carry-out) and sum represent the result of the addition operation. The results validate the functionality of the 4-bit adder, demonstrating correct binary addition with carry propagation. The waveform aligns with expected behavior, confirming the design's correctness.

**II. Experimental record**

**2.1 Create entity “fadd” for the 1-bit adder**

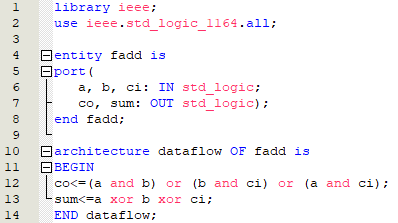


Figure14: code for entity “fadd”

This code defines the entity and architecture for a 1-bit full adder named fadd. The entity declaration specifies the inputs and outputs: a, b, and ci are 1-bit input signals of type std\_logic, while co (carry-out) and sum (sum output) are 1-bit outputs of type std\_logic. The architecture dataflow implements the logic of the full adder using Boolean equations.

The carry-out (co) is calculated as the OR of three terms: (a AND b), (b AND ci), and (a AND ci). This represents the cases where a carry is generated from the addition. The sum output (sum) is calculated as the XOR of the three inputs: a XOR b XOR ci, which corresponds to the sum logic of a full adder. This concise logic ensures efficient implementation and correct behavior of the full adder.

**2.2 Create component for the 4-bit full adder**

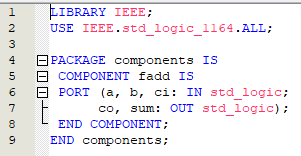


Figure15: code for component “fadd”

This code defines a package named components that includes a component declaration for fadd. The fadd component represents a 1-bit full adder, and its interface matches the entity fadd defined earlier. The PORT declaration specifies three inputs (a, b, and ci) of type std\_logic and two outputs (co and sum) also of type std\_logic. By including this component within a package, it allows the fadd design to be instantiated in other modules or architectures, enabling modular design and reusability. This approach is essential for hierarchical designs where the fadd logic can be used as a building block for larger circuits, such as a multi-bit adder.

**2.3 Create architecture for the 4-bit full adder**

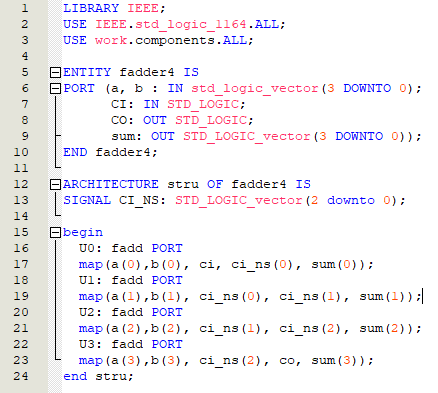


Figure16: code for architecture of “fadder4” 4-bit full adder

This code implements a 4-bit ripple carry adder using the fadd component. The ENTITY declaration, named fadder4, defines the interface with two 4-bit input vectors a and b, a single-bit carry-in CI, a 4-bit output vector sum, and a single-bit carry-out CO. The ARCHITECTURE named stru defines the internal structure of the 4-bit adder.

The architecture includes a signal CI\_NS, which is a 3-bit vector to propagate the internal carry bits between the 1-bit adders. Four instances of the fadd component are used (U0, U1, U2, U3), each representing a 1-bit full adder. The PORT MAP statements connect the inputs and outputs of each instance to the corresponding bits of a, b, sum, CI, CO, and the intermediate carry signals.

1. U0: The least significant bit addition, using a(0), b(0), and the external CI as inputs, and generating sum(0) and the first carry CI\_NS(0).
2. U1: Adds the next significant bits a(1) and b(1) along with CI\_NS(0) as the carry-in, producing sum(1) and CI\_NS(1).
3. U2: Adds a(2) and b(2) with CI\_NS(1) as carry-in, generating sum(2) and CI\_NS(2).
4. U3: The most significant bit addition, taking a(3), b(3), and CI\_NS(2) as inputs, producing sum(3) and the final carry-out CO.

**2.4 Modify the generated test bench file of 4-bit full adder**

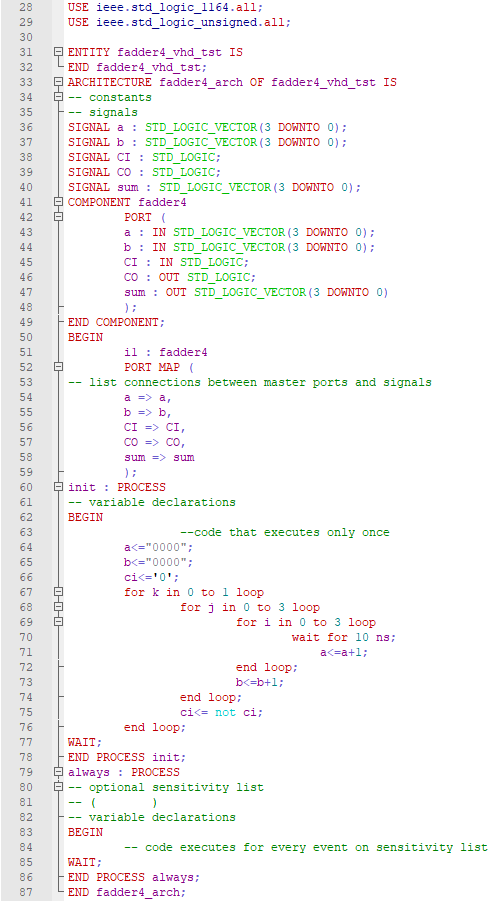


Figure17: code for architecture of the test bench file

In the PROCESS block labeled init, the testbench generates a sequence of inputs for testing the 4-bit adder. The process begins by initializing the input signals a, b, and ci to 0000, 0000, and 0 respectively. A nested loop structure is used to iterate through various combinations of these inputs:

1. Outer loop (k loop): This loop runs for 2 iterations, representing the two states of the ci (carry-in) signal (0 and 1).
2. Middle loop (j loop): This loop runs for 4 iterations, representing the possible values of the 4-bit vector b.
3. Inner loop (i loop): This loop also runs for 4 iterations, representing the possible values of the 4-bit vector a.

Inside the innermost loop, a wait for 10 ns statement introduces a delay between input changes, simulating the propagation time for the adder. After each delay, the value of a is incremented by 1 (a <= a + 1) to test the next input combination. Once all combinations for a are tested, the b signal is incremented (b <= b + 1), and the loop continues. At the end of each middle loop iteration, the ci signal toggles (ci <= not ci) to test the adder's behavior with and without carry-in.

This systematic approach ensures that all possible input combinations are tested, covering the entire input space for a 4-bit adder. It verifies the correctness and robustness of the design under various scenarios.

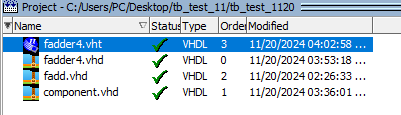


Figure18: All files compiled well after modification

**2.5 Final results of the simulation**

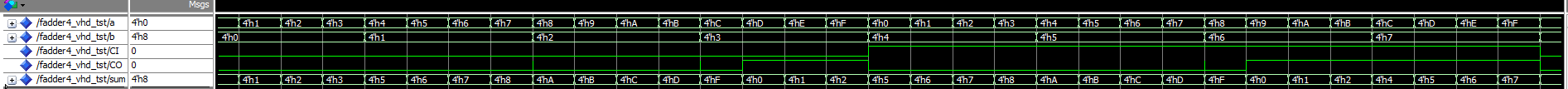


Figure19: Results of the simulation

This simulation waveform demonstrates the behavior of a 4-bit full adder, highlighting how the inputs a, b, and the carry-in CI influence the outputs sum and the carry-out CO. The 4-bit full adder is designed to take two 4-bit binary numbers and a single-bit carry-in as inputs, producing a 4-bit sum and a single-bit carry-out. In the simulation, the inputs a and b increment sequentially, covering all possible combinations, while CI toggles at specific intervals to test the adder's response to different carry-in scenarios. A key feature of the 4-bit full adder is its ability to handle carry propagation, where the carry-out CO of a lower bit addition serves as the carry-in for the next higher bit. This is evident in the waveform, as CO becomes active when the combined sum of a, b, and CI exceeds the 4-bit limit, such as when a and b are both at their maximum values (0xF + 0xF). The sum output in the waveform aligns with the expected results of binary addition. When CI is zero, the sum matches a + b, and when CI is one, the sum reflects a + b + 1. The behavior remains consistent and accurate across all input combinations, demonstrating the adder’s correct functionality. Furthermore, the simulation effectively tests boundary conditions, such as when both inputs are at their maximum values with CI = 1, ensuring that the design correctly handles overflow by producing a valid carry-out while the sum wraps within 4 bits. The simulation results confirm that the 4-bit full adder is functioning as intended, accurately performing addition, handling carry propagation, and managing edge cases reliably.

**III. Analysis and discussion**

The implementation and simulation of the 4-bit full adder in VHDL provide a comprehensive workflow that showcases the design process, modularity, and validation of digital systems. The project begins with the creation of individual components and progresses through architectural assembly, testbench design, and final simulation to verify functionality. The workflow is both methodical and iterative, allowing for error detection and resolution at each step.

The design starts with the definition of a 1-bit full adder entity, fadd, which forms the core building block of the 4-bit adder. The entity's inputs and outputs are clearly defined, and its functionality is implemented using Boolean equations to handle the sum and carry logic. This modular design enables reusability, as the fadd component is instantiated multiple times in the 4-bit adder architecture. A package declaration further encapsulates the component, facilitating its inclusion in higher-level designs. The hierarchical approach adopted in this project underscores the importance of modularity in digital design, where smaller, well-defined components are combined to build more complex systems.

The 4-bit adder, fadder4, utilizes the fadd component in a structured architecture. Intermediate signals, such as CI\_NS, are introduced to handle carry propagation between bits. A FOR ... GENERATE loop with conditional IF statements ensures that the appropriate logic is applied to each bit of the adder. The least significant bit uses the external carry-in directly, the intermediate bits propagate carry using CI\_NS, and the most significant bit outputs the final carry-out. However, an error in the initial implementation of the MSB logic resulted in the carry-out CO being undefined, as the carry-in was hardcoded to '0' rather than correctly linked to the carry propagation signal CI\_NS(2). This issue highlights the importance of carefully managing signal connections in multi-bit designs.

The testbench for the 4-bit adder, fadder4\_vhd\_tst, systematically verifies the design's functionality. Signals are initialized, and a nested loop structure generates all possible input combinations for the adder. The use of delays between input changes ensures realistic timing behavior, and the carry-in signal toggles to test the adder under various conditions. The testbench also instantiates the fadder4 component, mapping its ports to internal signals for simulation. This approach provides comprehensive coverage of the input space, verifying the adder's behavior across all possible scenarios. The testbench serves as a crucial step in the design process, allowing errors to be identified and corrected before hardware implementation.

The simulation results reveal both the strengths and weaknesses of the initial design. While the sum outputs align with expectations, the undefined carry-out CO for the MSB exposed a flaw in the implementation. By modifying the MSB logic to correctly connect CI\_NS(2) to the ci input of the final adder stage, the issue was resolved, and the design produced valid results. The corrected simulation confirmed the accurate operation of the 4-bit adder, including proper carry propagation and handling of edge cases.

Overall, this project demonstrates the complete process of designing, implementing, and validating a digital system in VHDL. The modular design approach, combined with systematic testing, ensures reliability and scalability. The error encountered in the carry-out logic underscores the iterative nature of digital design, where simulation plays a critical role in identifying and addressing issues. Through this process, the final design achieves its intended functionality, showcasing the robustness and flexibility of hierarchical, component-based digital design.

**VI. Other attachments**

During the experiment, we developed a code on our own attempting to accomplish the task.

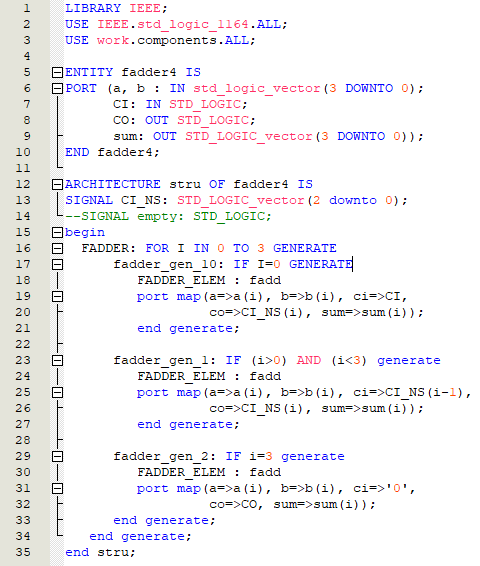


Figure20: Our own code design by using IF GENERATE statement

This code implements a 4-bit adder design using VHDL's GENERATE statement to instantiate the fadd component for each bit of the addition.

**Entity Declaration:**

The ENTITY section defines the interface for the 4-bit adder, including two 4-bit input vectors a and b, a single-bit carry-in CI, a 4-bit sum output vector sum, and a single-bit carry-out CO.

**Architecture Structure:**

The ARCHITECTURE section declares a signal CI\_NS, which holds the carry-out values between the bits (3 intermediate carry signals for the 1st, 2nd, and 3rd bits).

A FOR ... GENERATE loop is used to instantiate the fadd component for each bit of the adder, creating the logic for each individual bit.

**IF GENERATE Statement:**

1. For I = 0 (fadder\_gen\_10): The least significant bit (LSB) adder directly uses the external CI as the carry-in and computes the outputs sum(0) and CI\_NS(0) for the next bit.
2. For I > 0 and I < 3 (fadder\_gen\_1): The intermediate bits use the carry-out from the previous bit (CI\_NS(i-1)) as their carry-in to compute their sum(i) and propagate the carry-out to CI\_NS(i).
3. For I = 3 (fadder\_gen\_2): The most significant bit (MSB) adder incorrectly sets the carry-in to '0' rather than connecting to CI\_NS(2). This is supposed to calculate the final sum(3) and propagate the carry-out to the external CO.

**Identified Problem:**

In the current implementation, the MSB adder (I = 3) does not correctly connect the carry-out from the third bit (CI\_NS(2)) as its carry-in. Instead, the carry-in is hardcoded to '0'. This causes the calculation for the MSB to be incorrect, resulting in an invalid value for the carry-out CO.

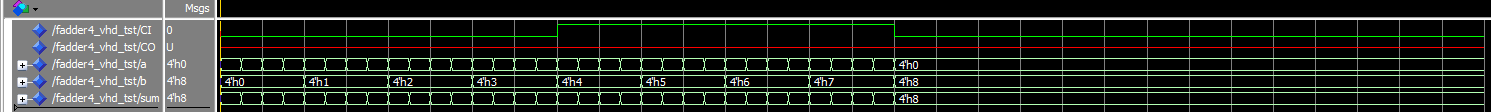


Figure21: The wrong simulation result

From the simulation results, the carry-out signal CO is displayed in red, indicating it is uninitialized or left in an undefined state (U). This confirms the issue in the original code, where the most significant bit's carry-out CO was not properly connected to the carry-out logic of the adder.

As discussed earlier, the problem lies in the fadder\_gen\_2 block of the code, where the carry-in (ci) for the most significant bit is hardcoded to '0', and CO is not correctly derived from the carry propagation. Specifically, CO should be connected to the carry-out of the last instantiated adder (CI\_NS(2)), but this connection is missing.

To resolve this issue, the carry-out CO for the most significant bit should be updated to reflect the correct propagation logic. The corrected code snippet should ensure that the CI\_NS(2) (carry-out of the second bit) serves as the carry-in to the third bit and calculates the final carry-out CO accurately, as the figure below:

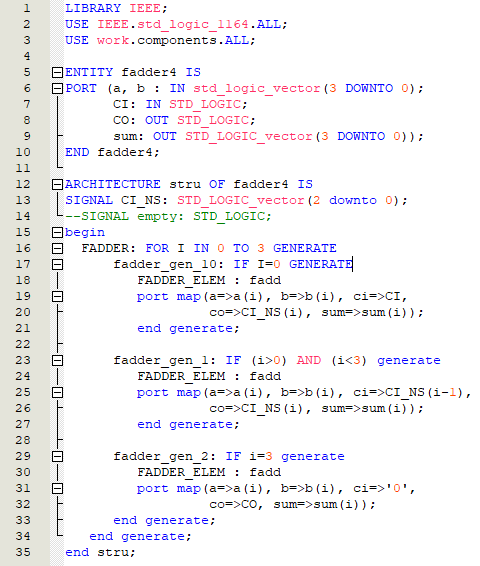


Figure22: Our modified code

And finally the simulation result was correct as figure below:

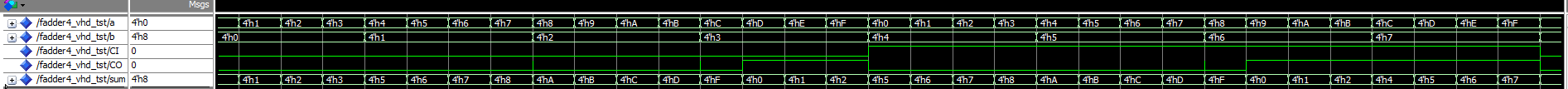


Figure23: The correct result after the code being modified