Simulation and Implementation of a One-bit (N-bits) Full Adder on FPGA

Wang Zhuoyang[1]

[1] *12112907, Department of Electrical and Electronic Engineering, SUSTech. Email: glverfer@outlook.com*

Abstract

This is the second lab report for the Digital System Designing course, which offers an overview of the principle and implementation of the **One-bit Full Adder**. During the pre-lab exercises, we derived the expected timing diagram for the signals under assumed conditions. Following this, we implemented the full adder in VHDL and simulated it to verify the correctness of the results.

In addition, this report documents the fundamental workflow of developing VHDL projects using Vivado. By utilizing the full adder module as the Unit Under Test (UUT), we investigate the differences between the five **simulation** types in Vivado. This analysis is conducted to enhance our understanding of the working principles of Electronic Design Automation softwares for FPGA programming.

Based on this work, we implemented an entity for an **N-bit Ripple Adder** and programmed the device with N = 4.

Keywords: Full Adder; Simulation; VHDL; Vivado

Contents

Digital System Designing Laboratory Reports (2023)

Corresponding author
Wang Zhuoyang

1 Introduction

About the Full Adder

A 1-bit full adder is a digital circuit that performs the addition of three one-bit inputs: A, B, and Cin (carry-in), and produces two one-bit outputs: S (sum) and Cout (carry-out). It allows for the addition of three inputs instead of just two, which is what a half adder does.

The structure of a 1-bit full adder typically consists of two stages: a first stage that computes the sum bit and a second stage that computes the carry-out bit. The first stage is implemented using two XOR gates, while the second stage is implemented using two AND gates and one OR gate. And the logic diagram is as shown in Figure 1.

Additionally, to facilitate analysis, we have tagged the intermediate signals as s_1, s_2, and s_3 to display their values in progress.

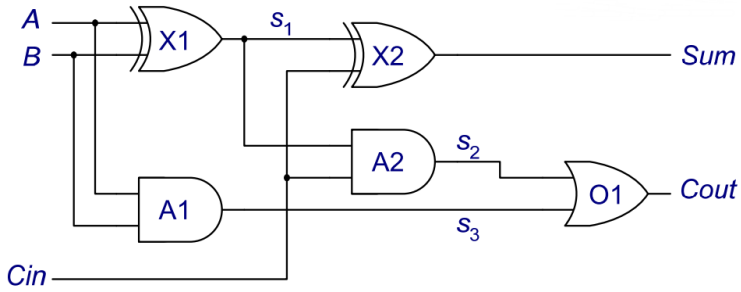

Figure 1. The logic diagram for a full adder.

Objective

The objective of this report can be divided into two main aspects.

One objective is to construct a 1-bit full adder using HDL and conduct logic and timing analysis on it, which is intended to deepen our understanding of the structure and principles of this circuit.

However, as the full adder is a fundamental circuit that we have already learned in a previous course, the more crucial objective of this exercise is to become familiar with the entire workflow of Vivado. By constructing a full adder as an example, we can obtain a comprehensive understanding of the software's architecture and its compilation procedure. In this lab, we will focus on the simulation process and investigate the disparities between the results of various simulation types, as well as their underlying causes.

Based on our previous work, we implemented an entity for an **N-bit Ripple Adder** and programmed the device with N = 4 to extend our understanding of the topic.

Disclaimer

While part of this article has been language-edited by ChatGPT (In this way, Figure 2), **it is in no way related to** the specific content of the text, such as the analysis, implementation, and other topics which are discussed in the article. This statement is hereby declared.

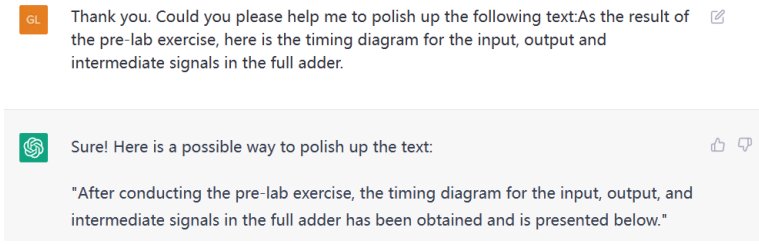

Figure 2. An example for how this report use GhatGPT.

2 Pre-lab Exercises

Below is the timing diagram for the input, output, and intermediate signals in the full adder, which was obtained as a result of the pre-lab exercise. This exercise provided the input signals and required us to manually simulate the behavior of the intermediate and output signals under the condition that the gate propagation delay is 10 ns.

The definitions of s_1, s_2, and s_3 are illustrated in Figure 1. And the entire simulation process was conducted for a duration of 100 ns.

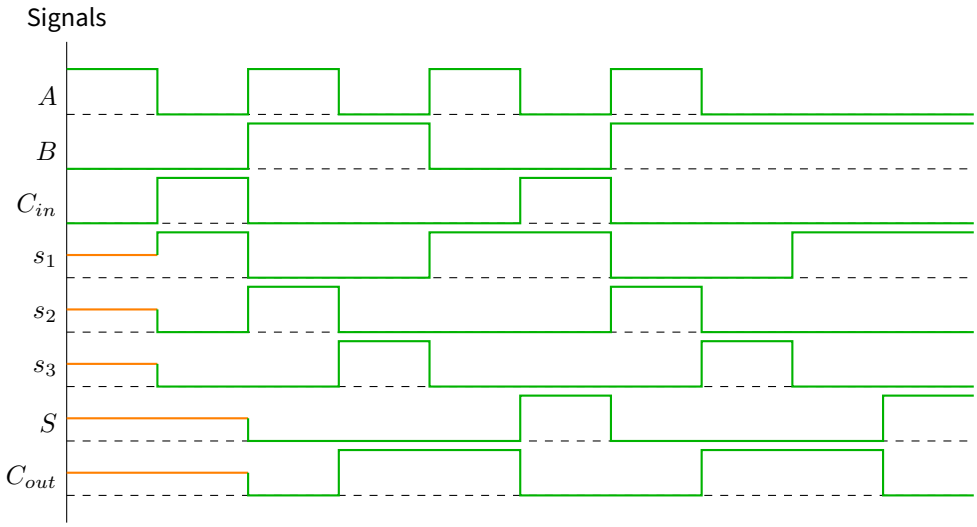

Figure 3. The timing diagram for the case that the propagation delay of each gate is 10 ns. (0 - 100 ns)

Due to the 10 ns propagation delay we have set, we were able to observe changes in the intermediate and output signals between 80 ns and 100 ns, despite the input signals no longer changing after 80 ns. Additionally, uninitialized values can also be observed at the beginning of the intermediate and output signals due to the same reasons.

We think it is unnecessary to conduct further analysis on this matter. The similar diagram can be viewed in the following text, which shows the result of the Behavioral Simulation in Vivado.

3 Source Code Implementation

3.1 Entity “full_adder_1bit”

Gate-level Architecture

There are many ways to construct a full adder. First, it can be described using gate circuits as the basic units. The logical diagram is shown in Figure 1.

```
architecture GateLevel of full_adder_1bit is
    signal s1, s2, s3: STD_LOGIC;
    constant gate_propagation_delay: time := 10 ns;
begin
    s1 <= (A xor B) after gate_propagation_delay;
    s2 <= (Cin and s1) after gate_propagation_delay;
    s3 <= (A and B) after gate_propagation_delay;
    S <= (s1 xor Cin) after gate_propagation_delay;
    Cout <= (s2 or s3) after gate_propagation_delay;
end GateLevel;
```

Listing 1. Gate level description.

We set the propagation delay of each gate to 10 ns in order to simulate a more realistic scenario (although it is an exaggerated value). However, for a clearer visualization of the simulation results, we set the propagation delay to 0 ns in subsequent simulations.

Here are some other remarks, like we can pin out the intermediate signals, which would be talked about in the following texts.

Module-level Architecture

We can also construct a full adder using two half adders and an OR gate. The structural view of this architecture is shown in Figure 4. The code for this architecture is the same as that presented in the slides, and is therefore omitted here.

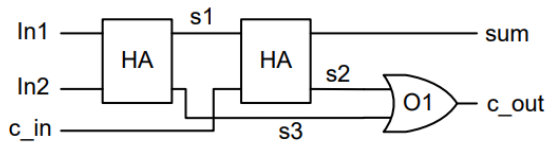

Figure 4. Module-level description of a full adder.

Behavioral Architecture

We could also implement the full adder using numeric operations provided by VHDL libraries such as ‘IEEE.NUMERIC_STD’. We may turn the signals into integers or other appropriate types to do the addition operation, and then pick out the LSB and the Sub-LSB as S and Cout. This can also be used to quickly construct a n-bit adder.

However, this approach may not be as instructive for our learning purposes and can be set aside for now. Also, the code is omitted.

Look-up-table Architecture

We can even program the truth table directly into the one-bit full adder module. For the same reason, we deprecated it.

3.2 Testbench “full_adder_1bit_tb”

In the testbench, our main tasks are to register and instantiate the full adder entity, and to generate the input signals. Part of the VHDL code is shown in Code 2.

```vhdl
architecture Behavioral of full_adder_1bit_tb is
    component full_adder_1bit is
        ...
    end component full_adder_1bit;
    ...
begin
    UUT: full_adder_1bit port map (A => A_tb, B => B_tb, Cin => Cin_tb, S =>
     S_tb, Cout => Cout_tb, S1out => s1_tb, S2out => s2_tb, S3out => s3_tb);

    A_tb <= '1', '0' after 10 ns, '1' after 20 ns, '0' after 30 ns, '1'
    after 40 ns, '0' after 50 ns, '1' after 60 ns, '0' after 70 ns;
    B_tb <= '0', '1' after 20 ns, '0' after 40 ns, '1' after 60 ns;
    Cin_tb <= '0', '1' after 10 ns, '0' after 20 ns, '1' after 50 ns, '0'
    after 60 ns;
end Behavioral;
```

Listing 2. The testbench for the 1-bit full adder entity.

The input signal we generated here is the same as the values of A, B and Cin in the section Pre-lab Exercises.

4 Simulation And Analysis

Pin Out the Intermediate Signals

During the development of a digital circuit, intermediate signals are often used to represent the values at specific points within the circuit. These signals are helpful in understanding the internal workings of the circuit, as they allow us to track the progression of values between inputs and outputs.

To display the values of the intermediate signals during simulation, we can add three additional outputs to the design. One way to achieve this is to modify the code as shown below (Code 3).

```vhdl
entity full_adder_1bit is
    Port (
        ...
        S1out, S2out, S3out: out STD_LOGIC
    );
end full_adder_1bit;

architecture GateLevel of full_adder_1bit is
    ...
begin
    ...
    S1out <= s1;
    S2out <= s2;
    S3out <= s3;
end GateLevel;
```

Listing 3. Pin out the intermediate signals.

4.1 Behavioral Simulation

The simulation result of the Behavioral Simulation is shown in Figure 5.

The timing diagram of the Behavioral Simulation matches the result of the pre-lab exercise, showing an apparent propagation delay of 10 ns. This delay is evident because the Behavioral Simulation is based solely on the VHDL code and does not take into account the specific circuits and devices. Therefore, characteristics that can only be observed in the code are apparent in this simulation type. However, there are some features that cannot be accurately represented

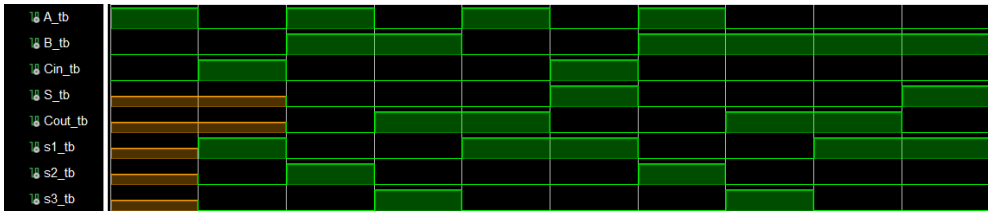

Figure 5. The result of the Behavioral Simulation

by circuits, which can result in differences between the Behavioral Simulation and other types of simulations.

4.2 Post-Synthesis Functional Simulation

The simulation result of the Post-Synthesis Functional Simulation is shown in Figure 6.

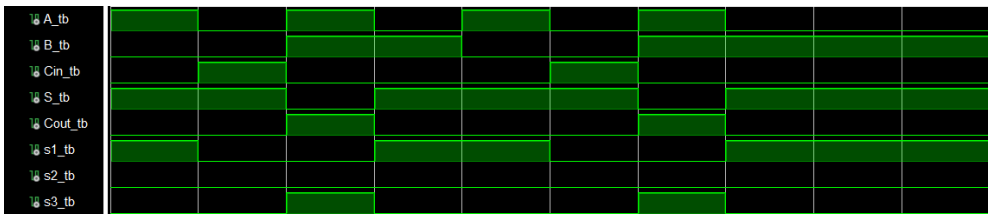

Figure 6. The result of the Post-Synthesis Functional Simulation

The reason for this difference in propagation delays between Behavioral Simulation and Post-Synthesis Functional Simulation is that the time values we set in Behavioral Simulation are only assumed values, while Vivado ignores those values and uses default propagation delay values for gates during Post-Synthesis Simulation. And, we think, these default values are set to zero in Functional Simulation.

The ‘after’ statements in the testbench are still functional, which we believe is reasonable. The testbench often uses keywords that are difficult to synthesize, but they are used only for testing purposes. Therefore, regardless of the simulation type that is running, we assume that the testbench is run in the Behavioral Simulation type.

4.3 Post-Synthesis Timing Simulation

The simulation result of the Post-Synthesis Timing Simulation is shown in Figure 7.

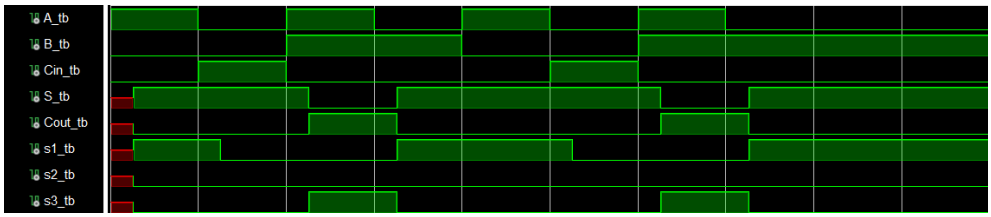

Figure 7. The result of the Post-Synthesis Timing Simulation

In this figure, there are two notable points. The first one is the timing characteristic, where we can observe the simulator adding default propagation delays to the logic gates. The delay is shorter than the 10 ns that we set before and depends on the built-in configurations. Unlike the Functional Simulation before, in the Timing Simulation, the delays are taken into consideration, which is the major difference between the two simulation types.

The second point to note in this figure is the red region ahead of the timing sequence. This comes from the uncertainty introduced by the propagation delay. In Behavioral Simulation, these parts are colored orange and denoted as 'U', which stands for "Uninitialized". However, in Post-Synthesis Simulations, the simulator attempts to model realistic situations, where signals must be asserted to a certain value instead of 'U'. As a result, 'X' (the red color) is used to represent "Uncertain", to differentiate it from other states.

4.4 Post-Implementation Functional Simulation

The simulation result of the Post-Implementation Functional Simulation is shown in Figure 8.

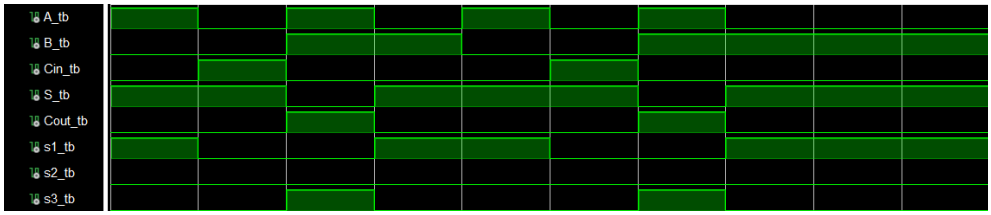

Figure 8. The result of the Post-Implementation Functional Simulation

The Post-Implementation Simulation is based on the results of the place-and-route process, which means that the unit under test (UUT) can be treated as nearly real circuits in the FPGA. And similarly, the "Functional" means ignoring the time delays.

This simulation type is similar to the Post-Synthesis Functional Simulation. Except that the simulator optimizes the schematics, the outputs of these two simulation types are expected to be the same.

4.5 Post-Implementation Timing Simulation

The simulation result of the Post-Implementation Timing Simulation is shown in Figure 9.

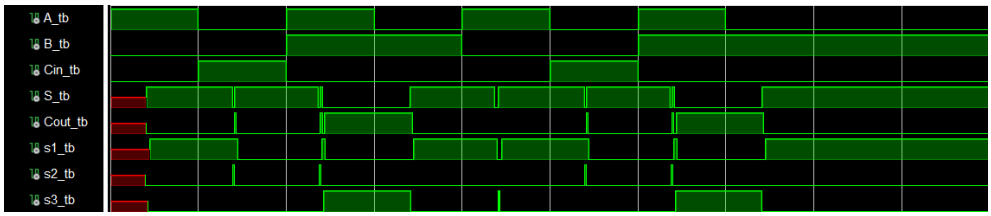

Figure 9. The result of the Post-Implementation Timing Simulation

The timing diagram shown in the Post-Implementation Simulation represents the final result of the entire simulation process. It displays the red regions that were mentioned earlier, indicating uncertainty due to propagation delay. Notably, the delays of different gates seem to vary, which better reflects the actual circuit behavior. These timing characteristics are a result of the configurations provided by the manufacturers of the chips.

Due to the varying delays of gates and interconnections, there may be spikes in this diagram, which could cause "hazard" phenomena. Overall, the Post-Implementation Simulation treats the UUT nearly as a real circuit in an FPGA, making it a valuable tool for verifying the functionality of the design.

5 Thinking And Extension

Note: ChatGPT was consulted regarding the usage of some characteristics in VHDL such as "generic" in this section.

5.1 Architecture Selection

Sometimes we want to preserve the different architectures of the same module, and VHDL also provides the way to name the architectures. So we can build the full adder entity in different ways in a single source file.

We can use sentences to choose different architectures like:

```
for UUT: full_adder_1bit use entity work.full_adder_1bit(Structure);
```

It is important to note that the top modules cannot be multi-architectural due to several reasons. These reasons will be explained in the following text.

5.2 N-bits Ripple Adder Implementation And Simulation

Since a full adder has already been constructed, it can be cascaded to obtain a multi-bit ripple adder. To achieve this, there are two main challenges to overcome. The first is how to define a module that depends on a given parameter N, and the second is how to generate and connect an uncertain number of full adders.

The first challenge can be addressed by adding a “generic” declaration to the entity. This allows for the definition of generic parameters that can be used in the “Port” declaration.

The second challenge can be addressed by using the “generate” syntax and preparing a vector of signals to preserve the intermediate connections. This is necessary because we have not found a method to call the labels that are auto-generated by the “generate” keyword.

The code is shown in Code 4.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity full_adder_nbits is
    generic (
        bits: POSITIVE := 8 -- Bits number as a parameter
    );
    port (
        A, B: in STD_LOGIC_VECTOR(bits - 1 downto 0);
        S: out STD_LOGIC_VECTOR(bits - 1 downto 0);
        Cin: in STD_LOGIC;
        Cout: out STD_LOGIC
    );
end full_adder_nbits;

architecture Ripple of full_adder_nbits is
    component full_adder_1bit is
        Port (
            A, B, Cin: in STD_LOGIC;
            S, Cout: out STD_LOGIC
        );
    end component full_adder_1bit;

    -- Preserve the intermediated carry-in/outs
    signal Cmid: STD_LOGIC_VECTOR(bits downto 0);
begin
    Cmid(0) <= Cin;
    -- Use a loop to generate the same modules
    NBITS_ADDER: for i in 0 to bits - 1 generate
        ONEBIT_FULL_ADDER: full_adder_1bit
            port map (
                A => A(i), B => B(i), S => S(i),
                Cin => Cmid(i),
                Cout => Cmid(i + 1)
            );
```

```
    end generate;
    Cout <= Cmid(bits);
end Ripple;
```

Listing 4. The VHDL code for a n-bits ripple adder. (n is a parameter)

We can write a testbench for the n-bits adder. Two “for ... loop” structures are used to generate numbers from 0 - 8 respectively for the input signals A and B (Code 5).

```
...
    UUT: full_adder_nbits
        generic map (bits => bits)
        port map (A => A_tb, B => B_tb, S => S_tb, Cin => Cin_tb, Cout =>
    Cout_tb);

    Cin_tb <= '0';
    process is
    begin
        for i in 0 to 8 loop
            A_tb <= STD_LOGIC_VECTOR(to_unsigned(i, bits));
            for j in 0 to 8 loop
                B_tb <= STD_LOGIC_VECTOR(to_unsigned(j, bits));
                wait for 10 ns;
            end loop;
        end loop;
        wait;
    end process;
...
```

Listing 5. Part of the VHDL code for the n-bits adder testbench.

We set $bits$ to 4 as an example and ran the Behavioral Simulation. The result is shown in Figure 10.

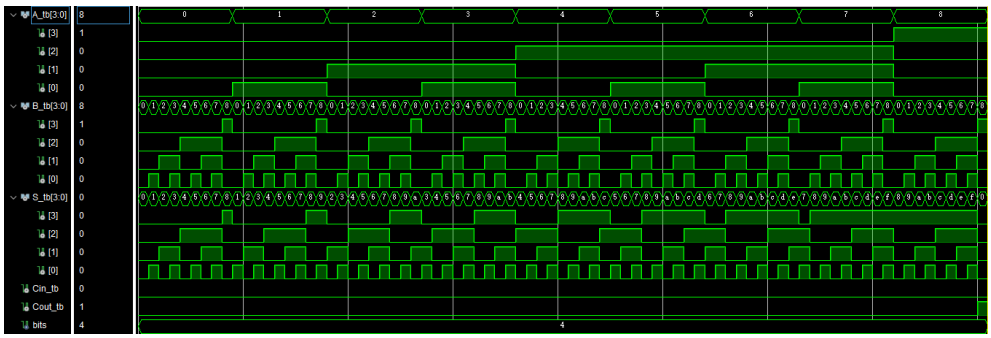

Figure 10. The result of the Behavioral Simulation for 4-bits ripple adder.

5.3 Program the Device

Top Module

The top module of a design can be thought of as the “main function” and plays a crucial role in the hierarchy system of Vivado. By right-clicking on an entity and selecting "Set as Top", we can designate which module is to be programmed into the device and directly connected to the chip’s input/output ports.

Therefore, it is essential to ensure that the final module is totally confirmed. Attempting to make the top module multi-architectural would result in an error when writing the bitstream or possibly earlier in the design process.

Constraint Files

Constraint files are written to regulate the voltage standards and the package pins mapping before writing bitstream. So in fact we can define these files after implement the design. For example, if we want to connect a signal to a concrete pin on the chip, we should define this information in the .xdc files.

Perhaps there is something wrong with my computer and Vivado, the constraint files created by me can never been detected by Vivado. So I always use the “I/O Ports” tool window to create the constraints. That is convenient.

It seems that constraints can also contains some other things, like those which are related to timing characteristics. We may learn about it in the following classes.

Program Device

Then we can eventually program our FPGA board (Figure 11).

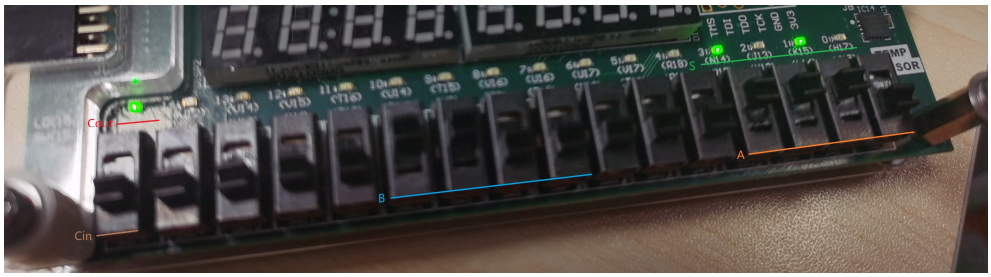

Figure 11. The 4-bits adder on board. I/O ports are shown in the figure.

We connected the switches and the leds with the inputs and outputs of the 4-bits ripple adder. The I/O ports are tagged in the figure.

6 Conclusion

In summary, this experiment reviewed the structure of adders and attempted to use VHDL to write adders with different architectures. We then conducted a complete simulation to compare the similarities and differences between different simulation types, and analyzed the reasons behind them. Finally, we used a full adder to construct an n-bit adder and actually ran a four-bit adder on a development board.