EE2016: Microprocessers Lab - Assignment 1 Report

Aadarsh Ramachandiran (EE23B001), Aditya Kartik (EE23B003), Rohita Datta (EE23B065)

August 14, 2024

Abstract

This is a detailed report on the Verilog assignment on implementing a half-adder, a full-adder and a four-bit ripple-carry adder in Xilinx Vivado. It contains the details of the solution including code, observation and comments on the programming and debugging experience.

1 Introduction

This assignment involves (i) simulating a half-adder using Xilinx Vivado (ii) realizing the half-adder on the FPGA board (iii) extending the half-adder design to a full-adder, simulating the same and testing the design on the FPGA board and (iv) designing a 4-bit ripple-carry adder in Verilog and implementing on the FPGA board. The half-adder is the simplest circuit for an arithmetic addition of two single bit numbers. The full-adder extends the functionality to addition of 3 single bit numbers or two numbers with a carry input. The four-bit ripple-carry adder can add two 4-bit numbers by using a combination of four full-adder circuits in series with carry propagation across them, hence the name "ripple".

2 Design

2.1 Half-adder

2.1.1 Description

Figure 1 shows the circuit diagram for a half-adder. There are two single bit inputs A and B and two single bit outputs, the corresponding sum and carry.

$$SUM = A \oplus B$$

$$CARRY = A \cdot B$$

The next section shows the code implemented for the half-adder circuit in Verilog. The inputs are defined as a and b and the outputs sum and carry as s and c respectively. We write a "dataflow" model for this which basically means that we describe the combinational circuit elements by their function rather than their gate structure. If we choose to use the buit-in xor and and modules within the half-adder module we achieve a "gate-level" model. It is a matter of style and does not affect the results of the experiment.

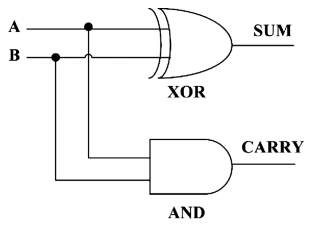

Figure 1: Half-adder circuit diagram (Leonardo Banchi, 2018 - source)

2.1.2 Code

Half-adder module:

```
// This code describes a dataflow model of a half adder logic circuit
module half_adder(a, b, s, c);
input a;
input b;
output s;
output c;
assign s = a ^ b;
assign c = a & b;
endmodule
```

We also write a testbench to simulate the circuit in software. This allows us to test its veracity before implementing it in hardware. In this file, we define the inputs and outputs, instantiate the half-adder module and vary the inputs a and b sequentially (in the initial block) and monitor the inputs and corresponding outputs.

Half-adder testbench:

```
// This code describes a testbench for testing
// and simulating the half adder circuit model
`timescale 1ns/1ps
module tb_half_adder;
reg a, b;
wire s, c;
half_adder h1(a, b, s, c);
initial
begin
  $dumpfile("test_half_adder.vcd");
  $dumpvars(1, tb_half_adder);
  a = 1'b0;
  b = 1'b1;
  # 20
  a = 1'b0;
  b = 1'b0;
  # 20
  a = 1'b1;
  b = 1'b0;
  # 20
  a = 1'b1;
  b = 1'b1;
  # 30 $finish;
end
initial $monitor($time, " a = %b, b = %b, c = %b, s = %b", a, b, c, s);
endmodule
```

2.2 Full-adder

2.2.1 Description

Figure 2 shows the circuit diagram for a full-adder implemented using two half adders. There are three inputs, A, B and Cin, and two outputs S and Cout, the resulting sum and carry.

$$\mathbf{S = A \oplus B \oplus C_{in}}$$

$$S_1 = A \oplus B$$

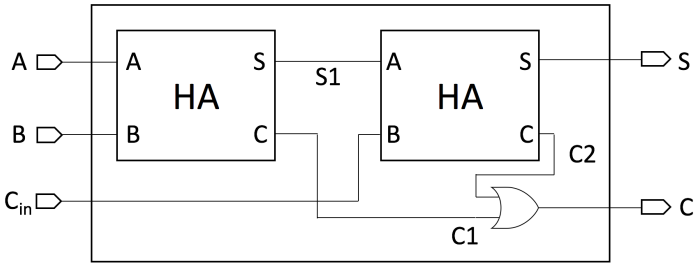

Figure 2: Full-adder implementation using Half-adder (source)

Sum output of first half adder, input to second

$$S = S_1 \oplus C_{in}$$

$$\mathbf{C_{out}} = \mathbf{A} \cdot \mathbf{B} + (\mathbf{A} \oplus \mathbf{B}) \cdot \mathbf{C_{in}}$$

$$C_1 = A \cdot B$$

Carry output of first half adder

$$C_2 = (A \oplus B) \cdot C_{in}$$

Carry output of second half adder

$$C_{out} = C_1 + C_2$$

Carry output of full adder

2.2.2 Code

Full adder Module:

```
//This code describes a gate-level implementation of a Full adder,along
//with using abstraction as we use instantiations of the half adder module
module full_adder(a, b, cin, s, cout);

input a, b, cin;
wire c1, s1, c2;
output cout,s;

half_adder ha1(a, b, s1, c1);
half_adder ha2(s1, cin, s, c2);
or(cout, c1, c2);

endmodule
```

We also write a testbench to simulate this in software. We vary the inputs x, y, z simultaneously and monitor how the outputs s and cout change.

2.2.3 Testbench:

Full Adder Testbench:

```
// This code describes a testbench for testing and simulating the full adder circuit mod
`timescale 1ns/1ps
module tb_full_adder;
reg a, b, c;
wire s, c;
```

```
full_adder f1(a, b, cin, s, cout);

initial
begin
    $dumpfile("test_full_adder.vcd");
    $dumpvars(1, tb_full_adder);
    a = 1'b0;
    b = 1'b0;
    cin = 1'b0;
    #20
    a = 1'b0;
    b = 1'b0;
    cin = 1'b1;
    #20
    a = 1'b0;
    b = 1'b1;
    cin = 1'b0;
    #20
    a = 1'b0;
    b = 1'b1;
    cin = 1'b1;
    #20
    a = 1'b1;
    b = 1'b0;
    cin = 1'b0;
    #20
    a = 1'b1;
    b = 1'b0;
    cin = 1'b1;
    #20
    a = 1'b1;
    b = 1'b1;
    cin = 1'b0;
    #20
    a = 1'b1;
    b = 1'b1;
    cin = 1'b1;
    # 30 $finish;
end
initial $monitor($time, "a=%b, b=%b, cin=%b, cout=%b, s=%b", a, b, cin, cout
endmodule
```

2.3 Four-bit Ripple-Carry Adder

2.3.1 Description

Figure 3 shows the circuit diagram for a four-bit ripple-carry adder. There are two four bit inputs A and B, a one bit carry input, C_{in}, a four bit output sum, S, and a one bit output carry, C_{out}. The i^{th} full adder takes inputs A_{i-1}, B_{i-1} and C_{i-2} and gives outputs S_{i-1} and C_{i-1} which are governed by the following formulae.

$$S_i = A_i \oplus B_i \oplus C_i$$

$$C_i = A.B + (A \oplus B).C_{in}$$

The next section shows the code implemented for the four-bit ripple-carry adder circuit in Verilog. The inputs are defined as i1 and i2 and the outputs sum and carry as o and cout respectively. We create instances of the full adder module from the previous section, to create the ripple carry adder

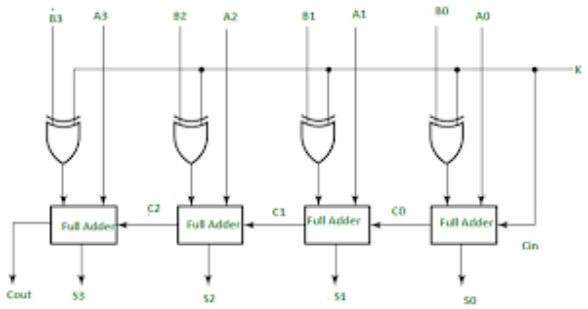

Figure 3: Four-bit adder circuit (source)

circuit. In the circuit, C_{in} is taken to be 0 and the rest of the inputs are fed in according to the users choice. The first full adder adds the least significant bits of the inputs and outputs their sum and carry. This carry is used as the input carry in the next full adder along with the next two bits from the inputs and this continues in series. This way four-bit numbers are added using single-bit full adders.

2.3.2 Code

Four bit ripple carry adder module

```
// This code describes a model for four-bit ripple-carry adder using four full adders

module ripple_carry_adder(i1, i2, cin, o, cout);

input [3:0] i1, i2;
input cin;
wire c1, c2, c3;
output[3:0] o;
output cout;

full_adder fa1(i1[0], i2[0], cin, o[0], c1);
full_adder fa2(i1[1], i2[1], c1, o[1], c2);
full_adder fa3(i1[2], i2[2], c2, o[2], c3);
full_adder fa4(i1[3], i2[3], c3, o[3], cout);

endmodule
```

We also write a testbench to simulate this. We vary the inputs and monitor how the outputs change.

Four bit ripple carry adder testbench

```
// This code describes a testbench for testing and simulating
// the four-bit ripple-carry adder circuit module
`timescale 1ns/1ps
module tb_ripple_carry_adder;
reg[3:0] i1, i2;
reg cin;
wire[3:0] o;
wire cout;
ripple_carry_adder rca(i1, i2, cin, o, cout);

initial
begin
    $dumpfile("test_ripple_carry_adder.vcd");
    $dumpvars(1, tb_ripple_carry_adder);
    cin = 1'b0;
```

```
    i1 = 4'b0000;
    i2 = 4'b0000;
    #20
    i1 = 4'b0010;
    i2 = 4'b0101;
    #20
    i1 = 4'b0111;
    i2 = 4'b1000;
    #20
    i1 = 4'b0110;
    i2 = 4'b1000;

    # 30 $finish;

end
initial $monitor($time, "-i1-=-%b,-i2-=-%b,-cin-=-%b,-cout-=-%b,-o-=-%b-", i1, i2, cin,

endmodule
```

3 Implementation

Once the Verilog code for the circuits are done, we proceed to simulate them, one circuit at a time, in Xilinx and realize the results on an FPGA board. We create a new project in Vivado and add the required Verilog files (module and testbench) through the "Add sources" option.
Note: It must be ensured that the module is declared as "top-level" in the hierarchy of files. Vivado displays errors, otherwise. Also, the Verilog files of all relevant modules in the experiment must be included. (For example, the half-adder file must be included when simulating the full-adder since the latter uses half-adder in its circuit)

3.1 Simulating in Xilinx

After adding the verilog files, we simulate them. Vivado shows the outputs and pops open the waveform veiwer to display the waveforms of various signals. The waveforms for all the three circuits can be viewed in section 6.

3.2 FPGA Realization

Once the outputs are analyzed and the circuit is verified to be correct, we proceed to implement it on the hardware. We write a Xilinx Design Constraints (.xdc) file to configure the input and output ports to be used in the FPGA board.
Code for the.xdc file

```
# Switches
set_property -dict { PACKAGE_PIN L5    IOSTANDARD LVCMOS33 } [get_ports { sw[0] }];#LSB
set_property -dict { PACKAGE_PIN L4    IOSTANDARD LVCMOS33 } [get_ports { sw[1] }];
set_property -dict { PACKAGE_PIN M4    IOSTANDARD LVCMOS33 } [get_ports { sw[2] }];
set_property -dict { PACKAGE_PIN M2    IOSTANDARD LVCMOS33 } [get_ports { sw[3] }];
set_property -dict { PACKAGE_PIN M1    IOSTANDARD LVCMOS33 } [get_ports { sw[4] }];
set_property -dict { PACKAGE_PIN N3    IOSTANDARD LVCMOS33 } [get_ports { sw[5] }];
set_property -dict { PACKAGE_PIN N2    IOSTANDARD LVCMOS33 } [get_ports { sw[6] }];
set_property -dict { PACKAGE_PIN N1    IOSTANDARD LVCMOS33 } [get_ports { sw[7] }];
set_property -dict { PACKAGE_PIN P1    IOSTANDARD LVCMOS33 } [get_ports { sw[8] }];

# LEDs
set_property -dict { PACKAGE_PIN J3    IOSTANDARD LVCMOS33 } [get_ports { led[0] }];#LSB
```

```
set_property -dict { PACKAGE_PIN H3    IOSTANDARD LVCMOS33 } [get_ports { led[1] }];
set_property -dict { PACKAGE_PIN J1    IOSTANDARD LVCMOS33 } [get_ports { led[2] }];
set_property -dict { PACKAGE_PIN K1    IOSTANDARD LVCMOS33 } [get_ports { led[3] }];
set_property -dict { PACKAGE_PIN L3    IOSTANDARD LVCMOS33 } [get_ports { led[4] }];
```

In this, we can configure switches and LEDs on the FPGA board to be used as inputs and outputs, respectively. This can be done simply by changing sw[0] to i1[0], led[0] to o[0], etc. Then we connect the FPGA board to the computer and generate the bitstream by clicking "Generate Bitstream" in the Xilinx menu. Then, we program the device by clicking "Program device". The circuit will then be successfully implemented in the FPGA board.

4 Observation

We used the sliding switches in the FPGA board for inputs and corresponding LEDs for the output. For any given input, corresponding LEDs glow to give the correct output. With this, we realise half-adder, full-adder and four-bit ripple-carry adder.

5 Conclusion

The experiment successfully demonstrated the design, simulation and implementation of digital adders using Verilog and FPGA. This provided a deeper understanding how digital arithmetic circuits work.

6 Appendices

The verilog codes used in this assignment are available in here
The waveforms for various circuits can be seen below:

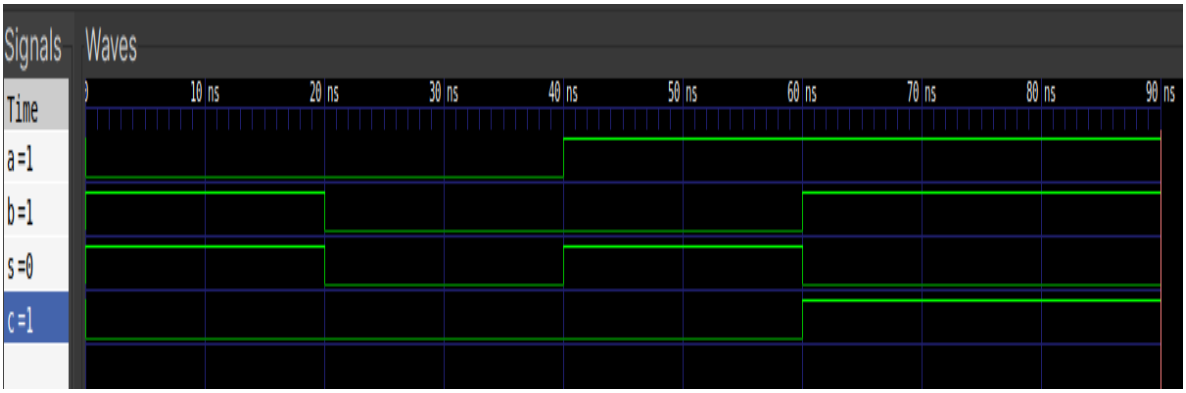

Figure 4: Half-adder waveform

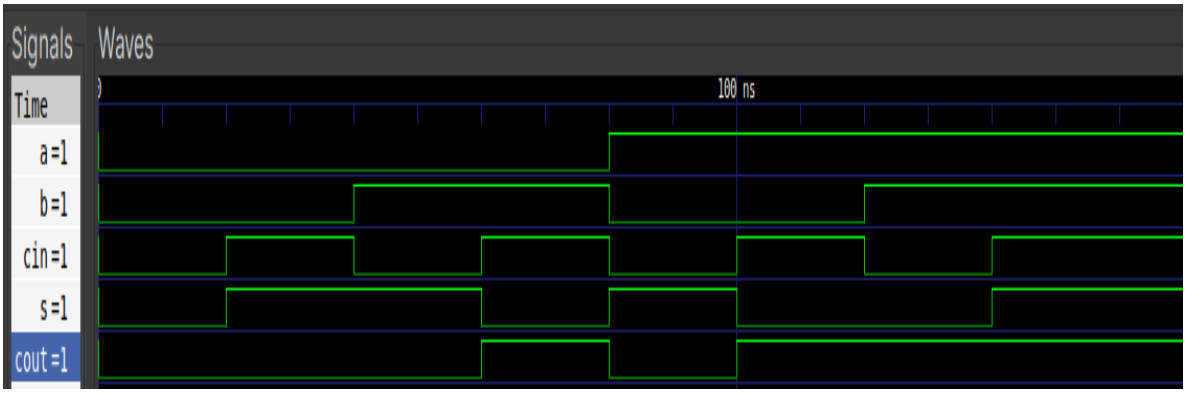

Figure 5: Full-adder waveform

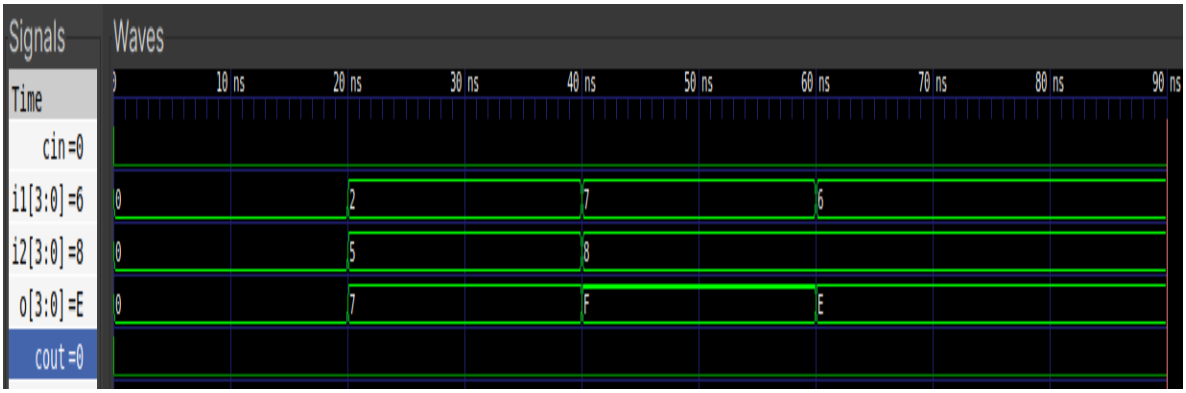

Figure 6: Four-bit ripple-carry adder waveform