# Lab 2 - Multiplexers
# Hardware Design - ECE 311

January 12, 2021

Partners: Jinhan Zhang
Kevin Jiang

**Abstract**

The lab aims to get us familiar with the Verilog language and different types of modeling: Gate-level, Dataflow, and Behavioral. The purpose of this lab is to use the three different types of modeling to design a 2-1 multiplexer in Vivado and to use them to implement a 3-1 multiplexer. The different models were tested by running simulations with given testbench codes. We verified that the models function properly by generating a bitstream and downloading it to the ZedBoard. Finally, we used what we learned about modelling to implement two different types of 4-bit adders.

## Introduction

The first three parts of the lab are to design a two bit wide 2-1 multiplexer using gate-level modeling, dataflow modeling, and behavioral modeling, respectively. For all three models we need to synthesize the design, implement the design, generate the bitstream, and download it into the ZedBoard to verify the functionality. For the dataflow modeling multiplexer, we need to run a simulation on a provided testbench before we synthesize and implement the design. The fourth part of the lab is to design a 3-1 multiplexer using the 2-1 multiplexers we designed in the first part of the lab. The original instruction was to design a 3-2 multiplexer, but we design a 3-1 multiplexer instead because it makes more sense. The fifth and sixth parts of the lab are to model 4-bit adders using dataflow modeling. The first adder is a Ripple Carry Adder, and the second adder is a Carry Look Ahead Adder. For both adders, we synthesized and implemented our design, and generated a bit-stream to download to the ZedBoard to verify the functionality of the model.

# Procedure

## Part 1

Figure 1 shows the circuit of the implemented 2-1 multiplexer using gate-level modeling. The circuit takes in 2 two bits input and there is a one bit address to select from. When the address is at 0, the output of the circuit is the same as input 1. When the address bit is 1, the output is the same as input 2. The design uses 4 AND gates and 2 OR gates which each bit of the input is AND with the address and then results of different input go into two different OR gates to generate the two bits of the output.

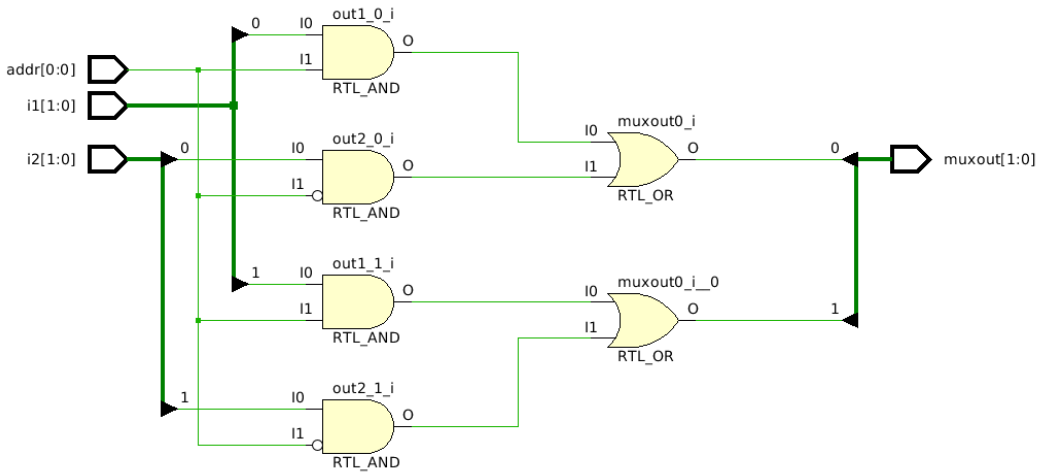


Figure 1: Design Schematics of 2-1 Multiplexer using gate-level modeling

### Synthesis Results

Figure 2 shows the synthesize schematic of the design.

### Implementation Results

Project Summary is shown in Figure 3.
The implemented schematics is shown in Figure 4.
LUTs: 1% of LUTs is utilized.
IO: 4% of IO is utilized.

### Summary and Discussion

| Experiment | Simulation | Theoretical |
|---|---|---|
| The experiment was a success. We were able to select from two different inputs by changing the address bit. | The simulation matched the expected output. Success. | From this experiment, we learned how to model a MUX by using logic gates. |

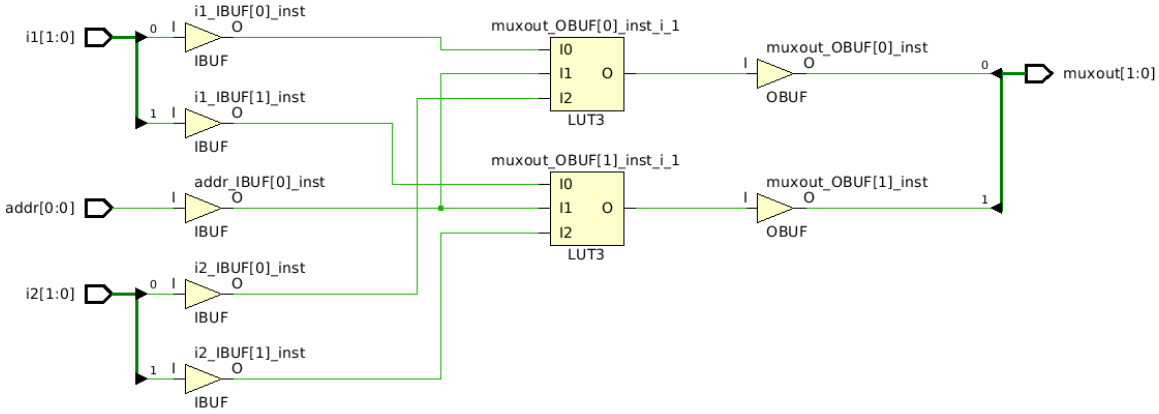


Figure 2: Design Schematics of 2-1 Multiplexer using gate-level modeling

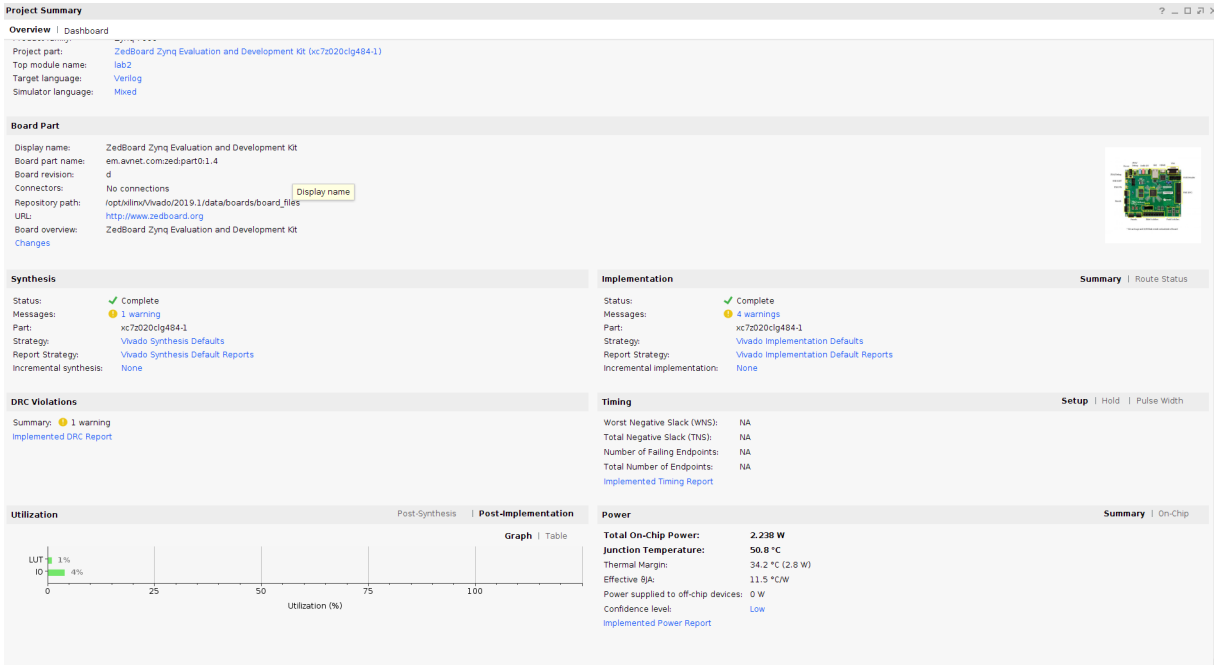


Figure 3: Project Summary of 2-1 Multiplexer using gate-level modeling

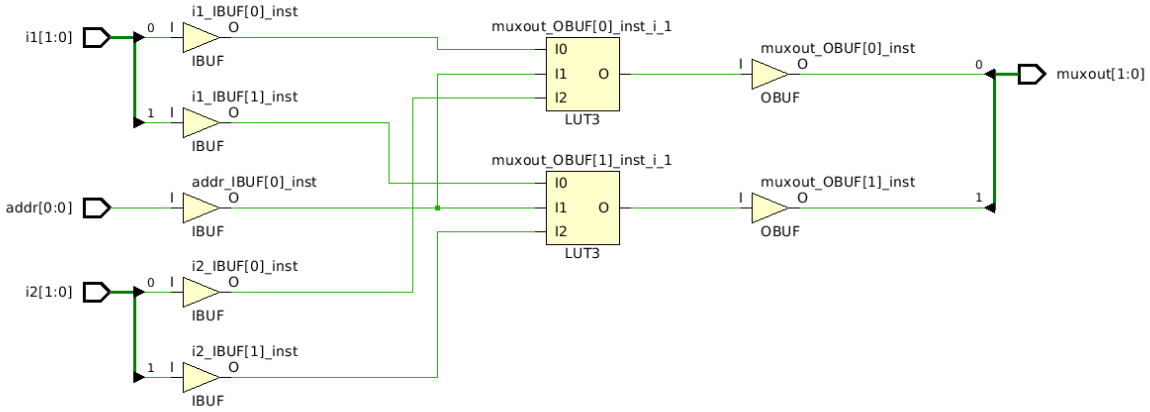


Figure 4: Implemented Schematics of 2-1 Multiplexer using gate-level modeling

## Part 2

Figure 5 shows the circuit of the implemented 2-1 multiplexer using dataflow modeling. The circuit function is the same as Part 1. The design schematics uses two 2-1 MUX gate that process the two bits of the inputs.

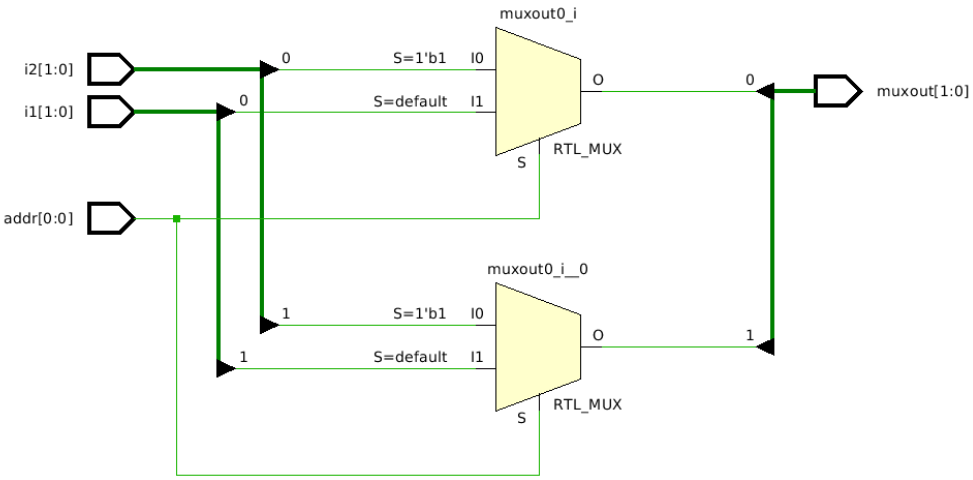


Figure 5: Design Schematics of 2-1 Multiplexer using dataflow modeling

### Simulation Results

Figure 6 shows the behavioral simulation of the 2-1 multiplexer. The simulation results match what you would expect from a typical 2 bit wide 2 to 1 multiplexer.

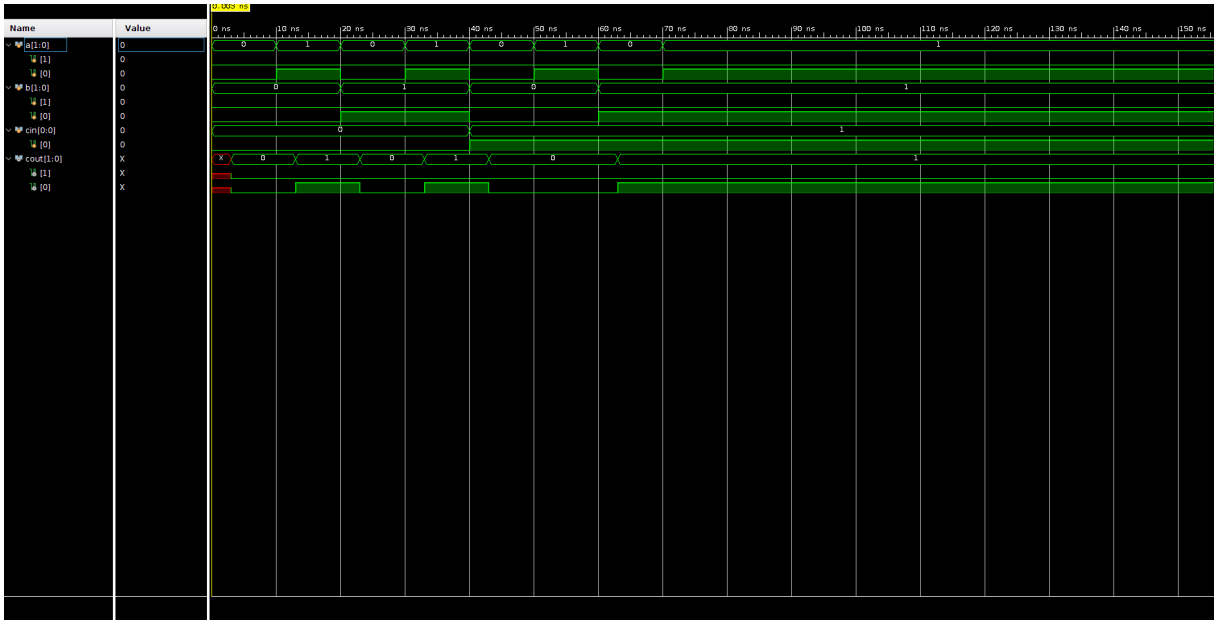


Figure 6: Behavioral Simulation of 2-1 Multiplexer using dataflow modeling

### Synthesis Results

Figure 7 shows the synthesize schematic of the design.

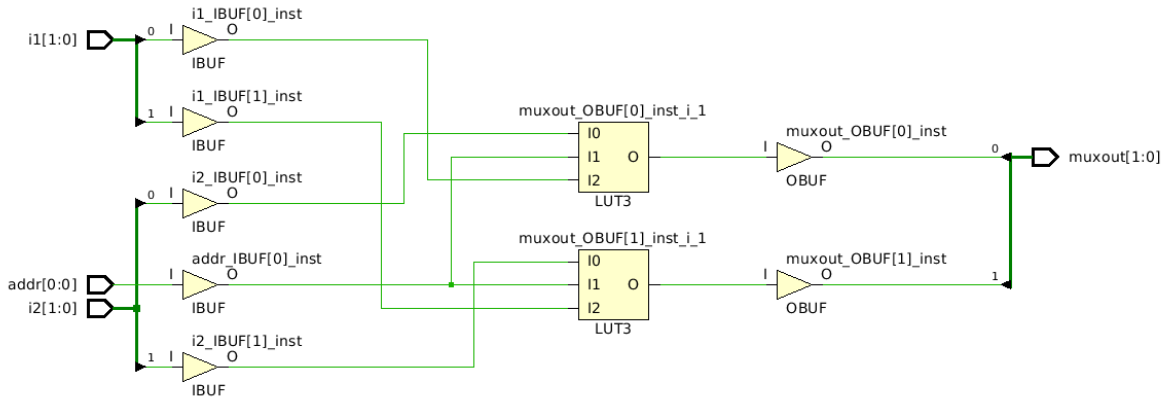


Figure 7: Design Schematics of 2-1 Multiplexer using dataflow modeling

### Implementation Results

Project Summary is shown in Figure 8.
The implemented schematics is shown in Figure 9.
LUTs: 1% of LUTs is utilized.
IO: 4% of IO is utilized.

### Summary and Discussion

| Experiment | Simulation | Theoretical |
|---|---|---|
| The experiment was a success. We were able to select from two different inputs by changing the address bit. | Simulation success. | From this experiment, we learned how to model a MUX by using dataflow modeling. |

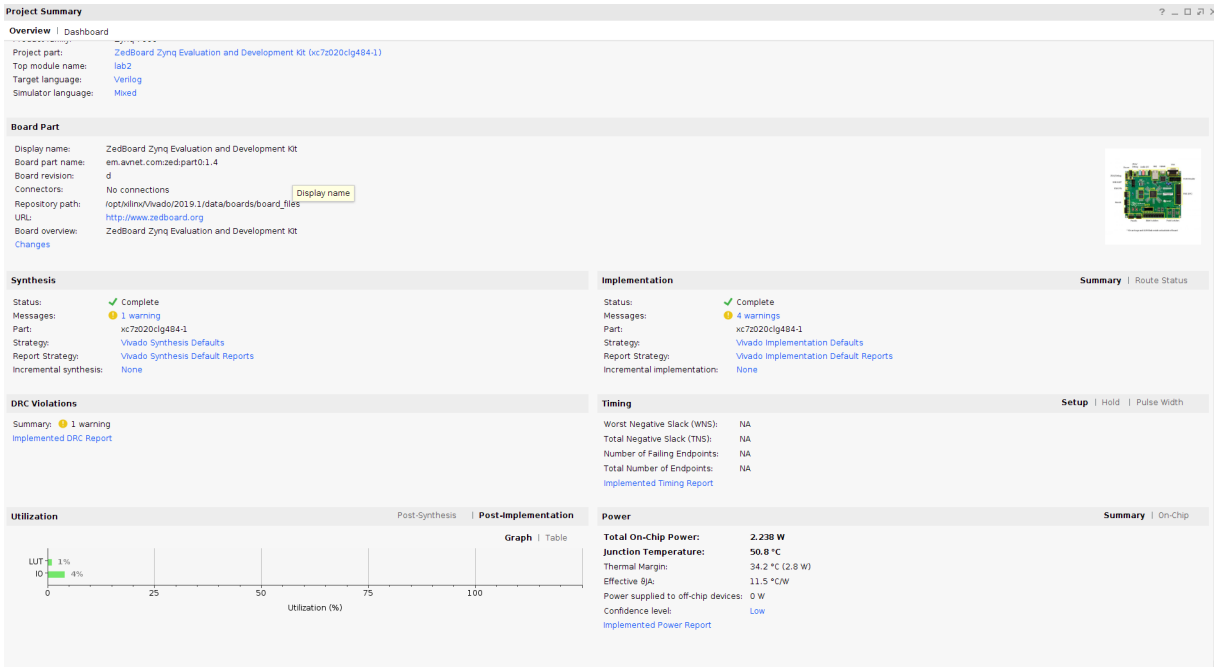


Figure 8: Project Summary of 2-1 Multiplexer using gate-level modeling

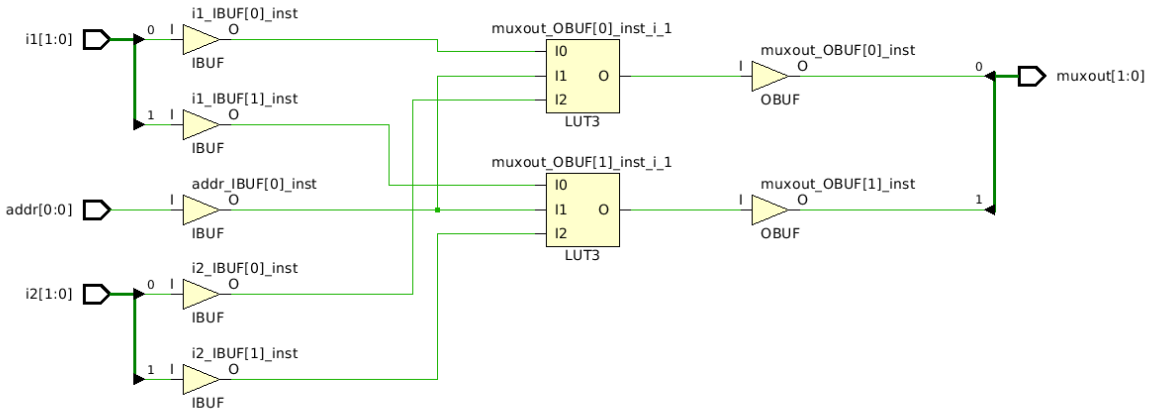


Figure 9: Implemented Schematics of 2-1 Multiplexer using gate-level modeling

## Part 3

Figure 10 shows the circuit of the implemented 2-1 multiplexer using behavior modeling. The circuit function is the same as Part 1.

### Synthesis Results

Figure 11 shows the synthesize schematic of the design.

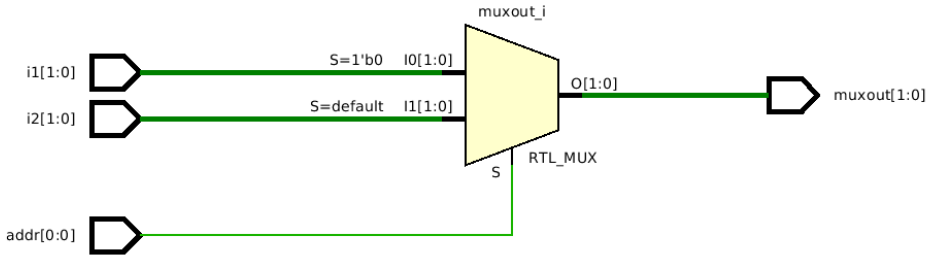


Figure 10: Design Schematics of 2-1 Multiplexer using behavior modeling

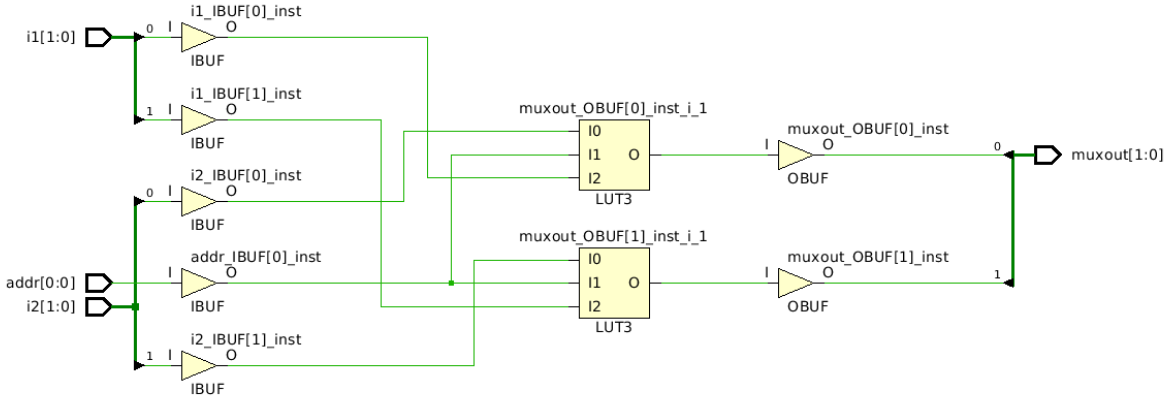


Figure 11: Design Schematics of 2-1 Multiplexer using behavor modeling

**Implementation Results**

Project Summary is shown in Figure 12.
The implemented schematics is shown in Figure 13.
LUTs: 1% of LUTs is utilized.
IO: 4% of IO is utilized.

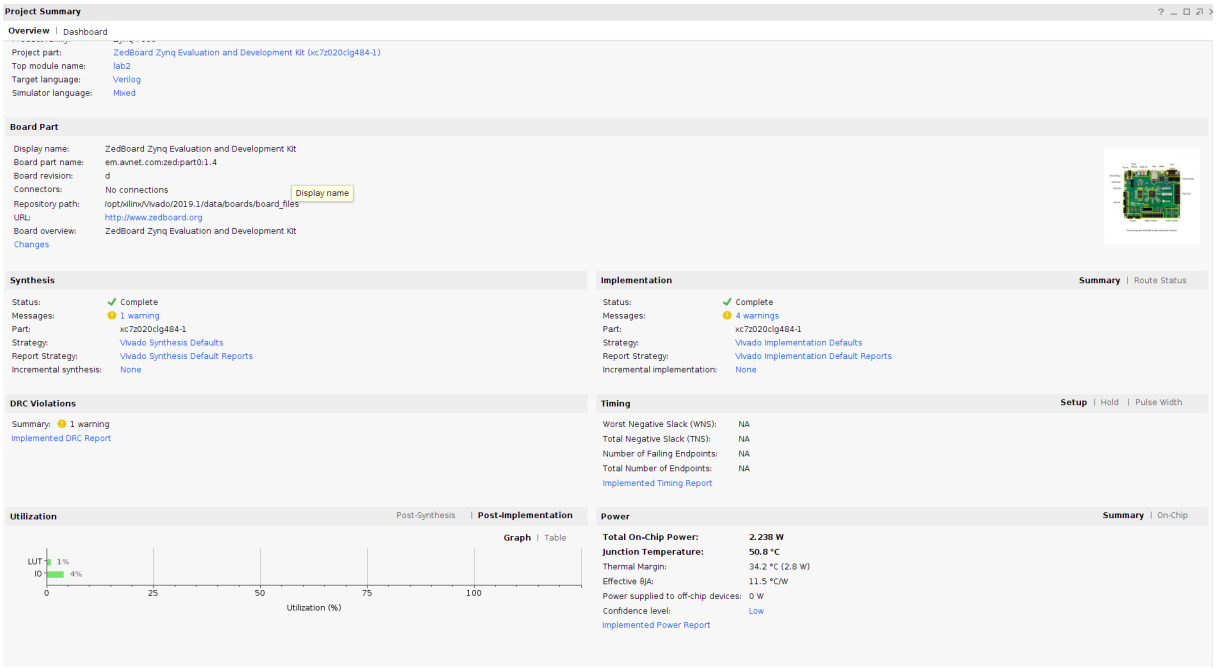


Figure 12: Project Summary of 2-1 Multiplexer using gate-level modeling

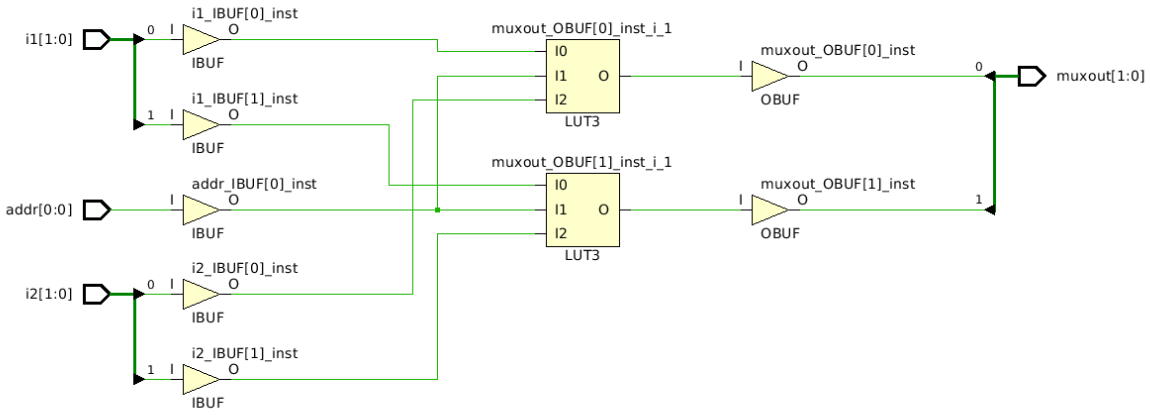


Figure 13: Implemented Schematics of 2-1 Multiplexer using gate-level modeling

## Summary and Discussion

| Experiment | Simulation | Theoretical |
|---|---|---|
| The experiment was a success. We were able to select from two different inputs by changing the address bit. | No simulation required. | From this experiment, we learned how to model a MUX by using behavioral modeling. |

## Part 4

Figure 14 shows the circuit of the implemented 2-1 multiplexer using behavior modeling. The circuit function is the same as Part 1.

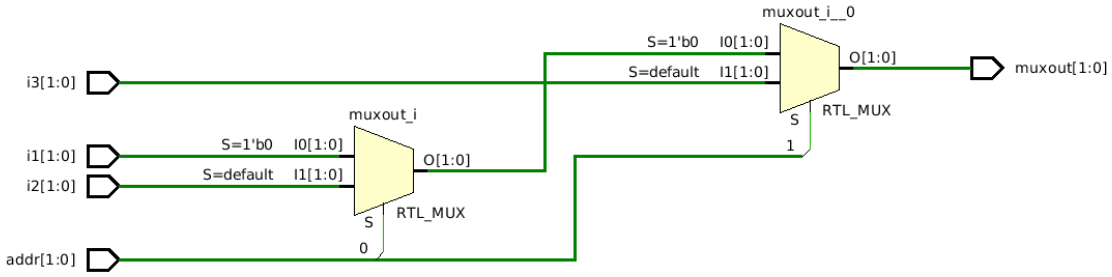


Figure 14: Design Schematics of 2-1 Multiplexer using behavior modeling

### Synthesis Results

Figure 15 shows the synthesize schematic of the design.

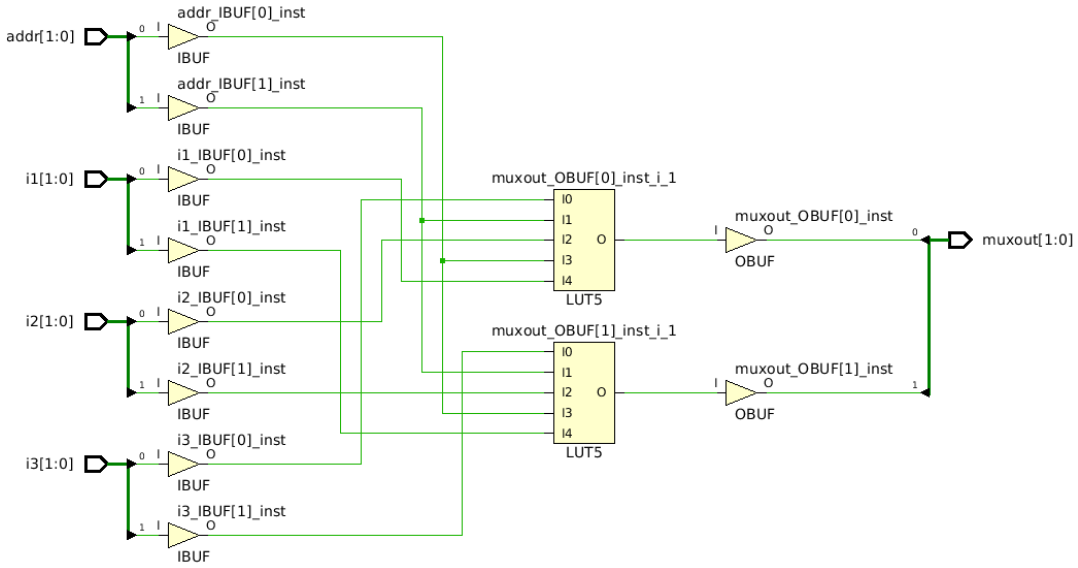


Figure 15: Design Schematics of 2-1 Multiplexer using behavor modeling

### Implementation Results

Project Summary is shown in Figure 16.
The implemented schematics is shown in Figure 17.
LUTs: 1% of LUTs is utilized.
IO: 5% of IO is utilized.

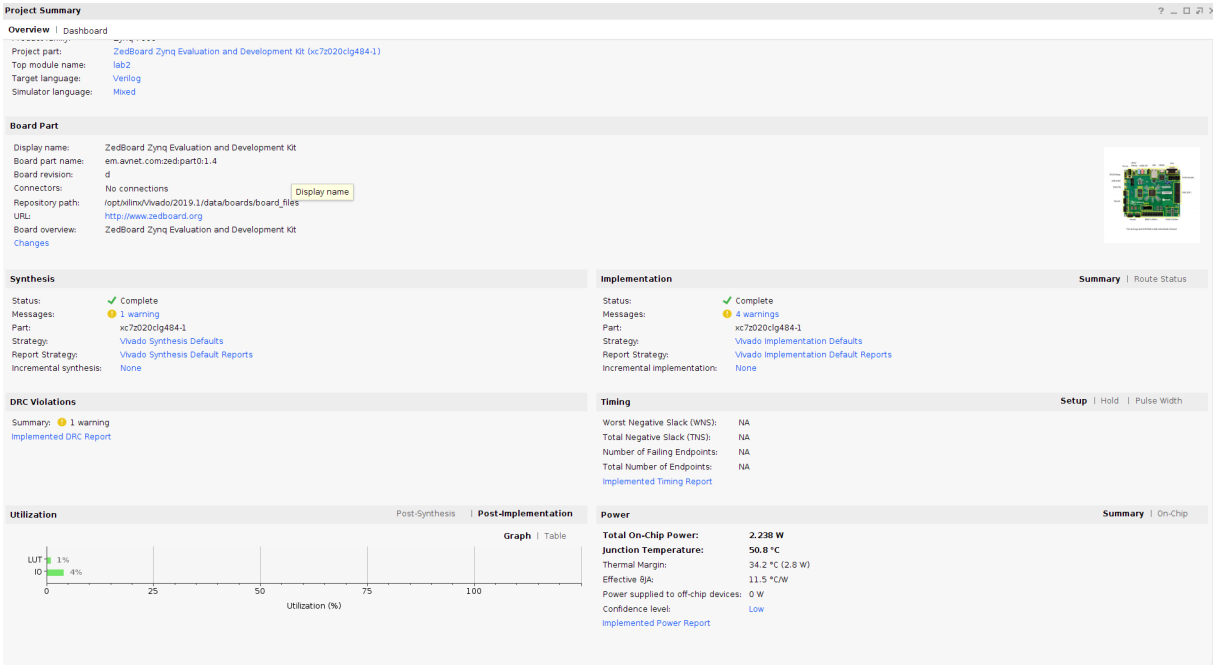


Figure 16: Project Summary of 2-1 Multiplexer using gate-level modeling

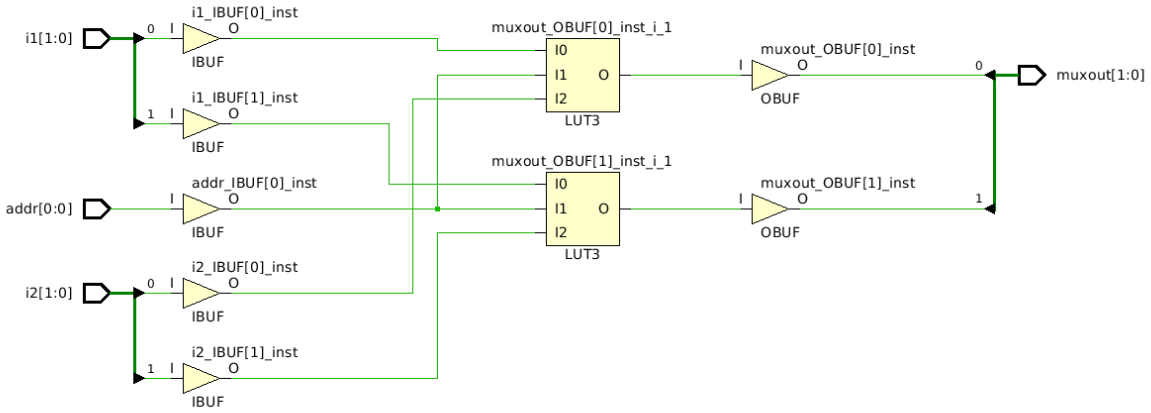


Figure 17: Implemented Schematics of 2-1 Multiplexer using gate-level modeling

**Summary and Discussion**

| Experiment | Simulation | Theoretical |
|---|---|---|
| The experiment was a success. We were able to select from two different inputs by changing the address bit. | No simulation required. | From this experiment, we learned how to combine two 2:1 MUX into a 3:1 MUX using behavioral modeling |

# 1 Part 5

In Part 5 of this lab, we implement a Ripple Carry Adder using dataflow modeling. The functional block diagram and truth table is shown in Figure 18. We modeled and tested our Ripple Carry Adder according to the truth table shown in Figure 18. The Ripple Carry Adder is a 4-bit adder, and the design schematic is shown in Figure 19.

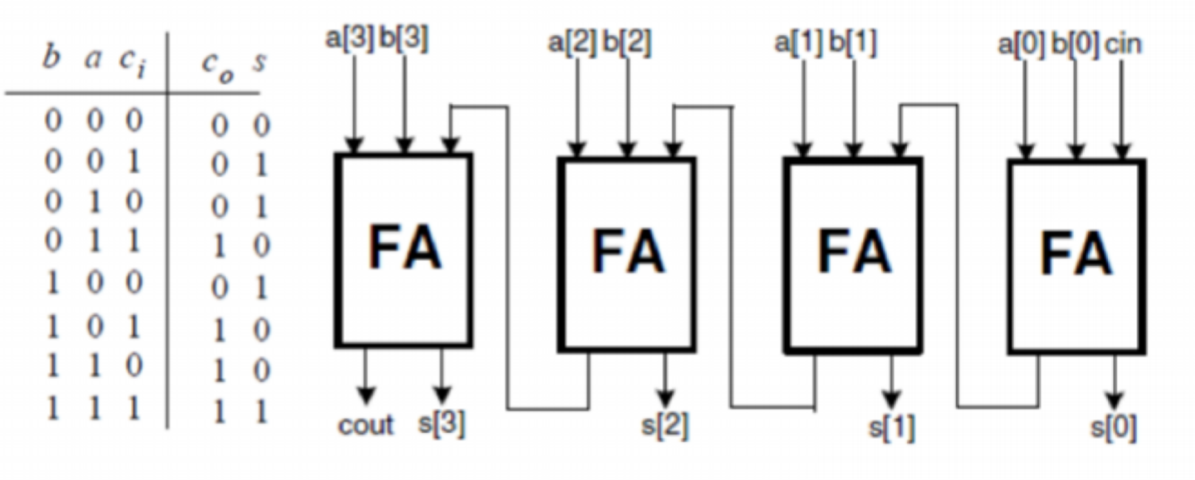


Figure 18: Functional Block Diagram and Truth Table for Ripple Carry Adder

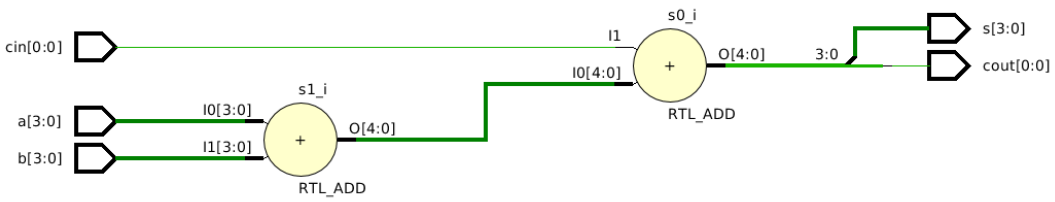


Figure 19: Design Schematics of 4-bit Ripple Carry Adder

### Simulation Results

Figure 20 shows the behavioral simulation of the 4-bit Ripple Carry Adder. As you can see, the simulation results match the expected results in the truth table.

### Synthesis Results

Figure 21 shows the synthesized schematic of the design.

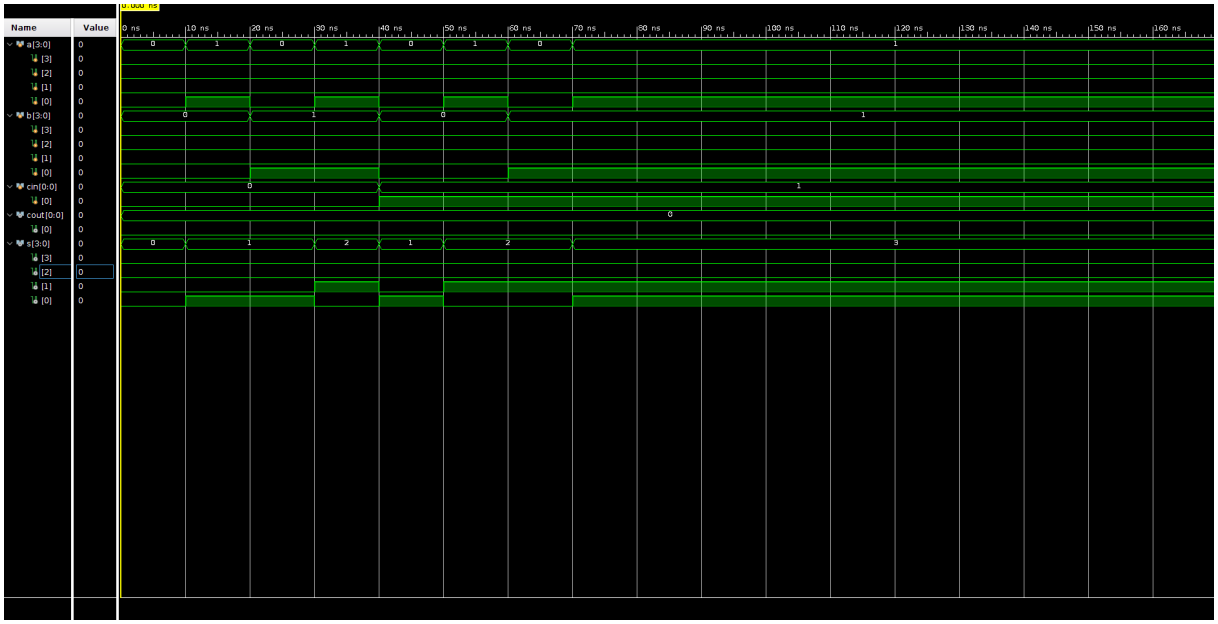


Figure 20: Behavioral Simulation of 4-bit Ripple Carry Adder

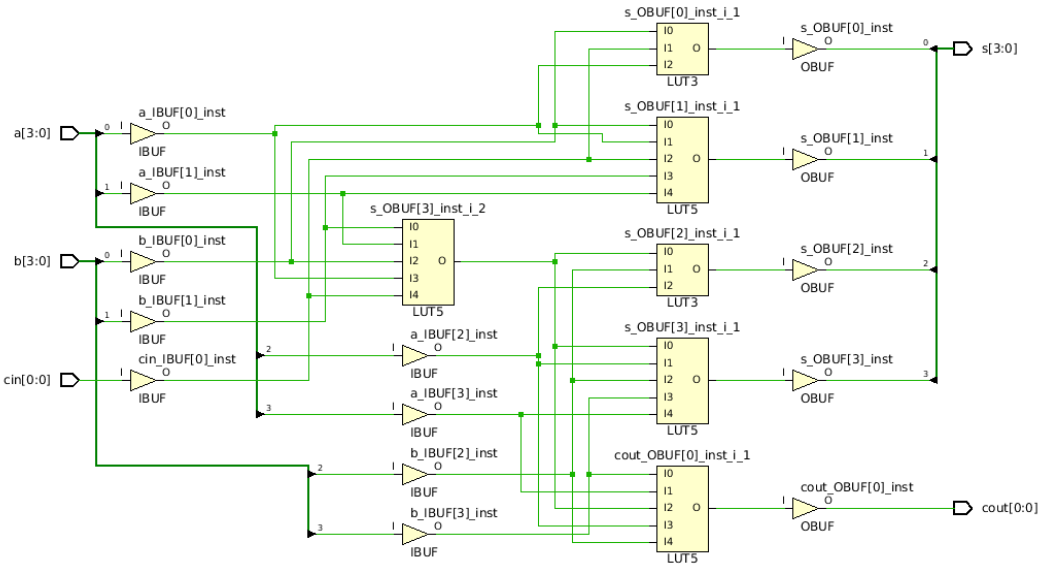


Figure 21: Synthesized Schematics of 4-bit Ripple Carry Adder

**Implementation Results**

Figure 22 shows the Project Summary.
Figure 23 shows the Implemented Schematics.
LUTs: 7% of LUTs are utilized.
IO: 1% of IO is utilized.

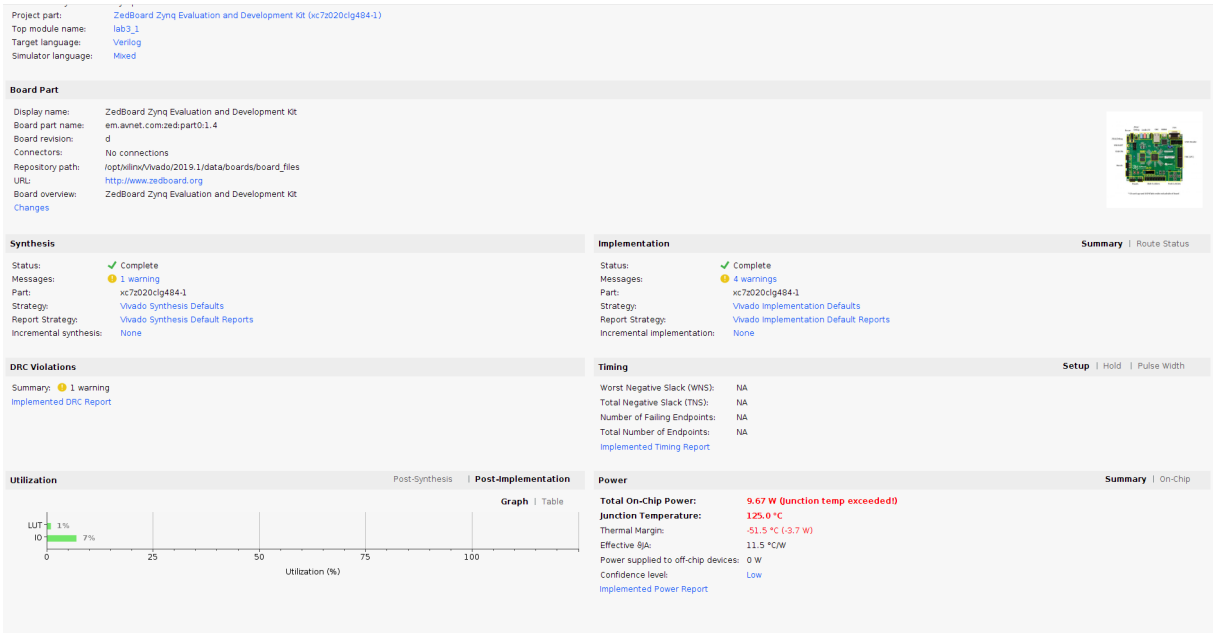


Figure 22: Project Summary of 4-bit Ripple Carry Adder

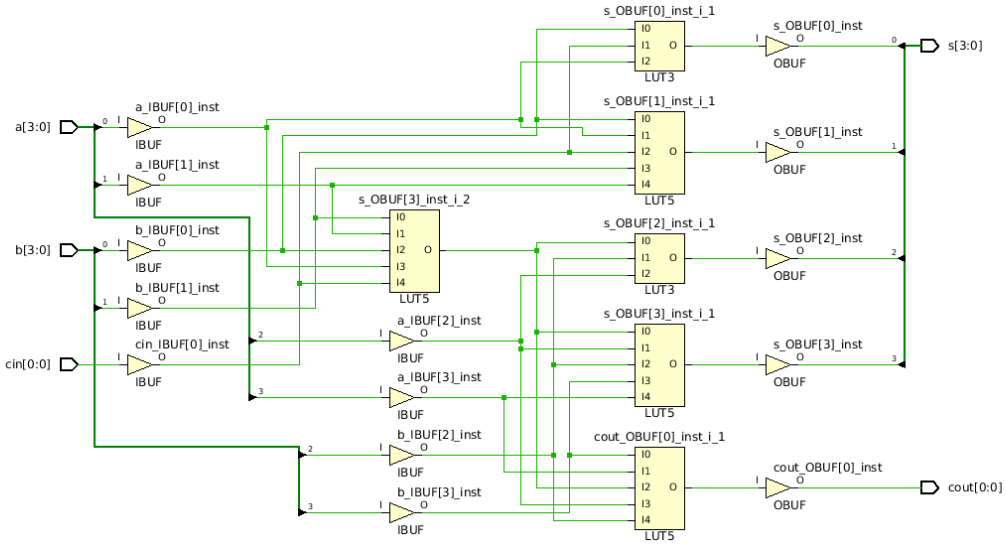


Figure 23: Implemented Schematics of 4-bit Ripple Carry Adder

### Summary and Discussion

In this section, summarize the results for this part of the experiment through tabulation when possible.

| Experiment | Simulation | Theoretical |
|---|---|---|
| The experiment was a success. The sum and carry out bits all followed the truth table when we tested our model on the Zedboard. | The simulation matched the expected output. Success. | We learned that we could concatenate bits and add them. This allowed us to write concise modules. |

## 2 Part 6

In Part 6 of this lab, we implement a Carry Look Ahead Adder using dataflow modeling. The functional block diagram is shown in Figure 24. We modeled and tested our Carry Look Ahead Adder according to the functional block diagram shown in Figure 24. The design schematic is shown in Figure 25.

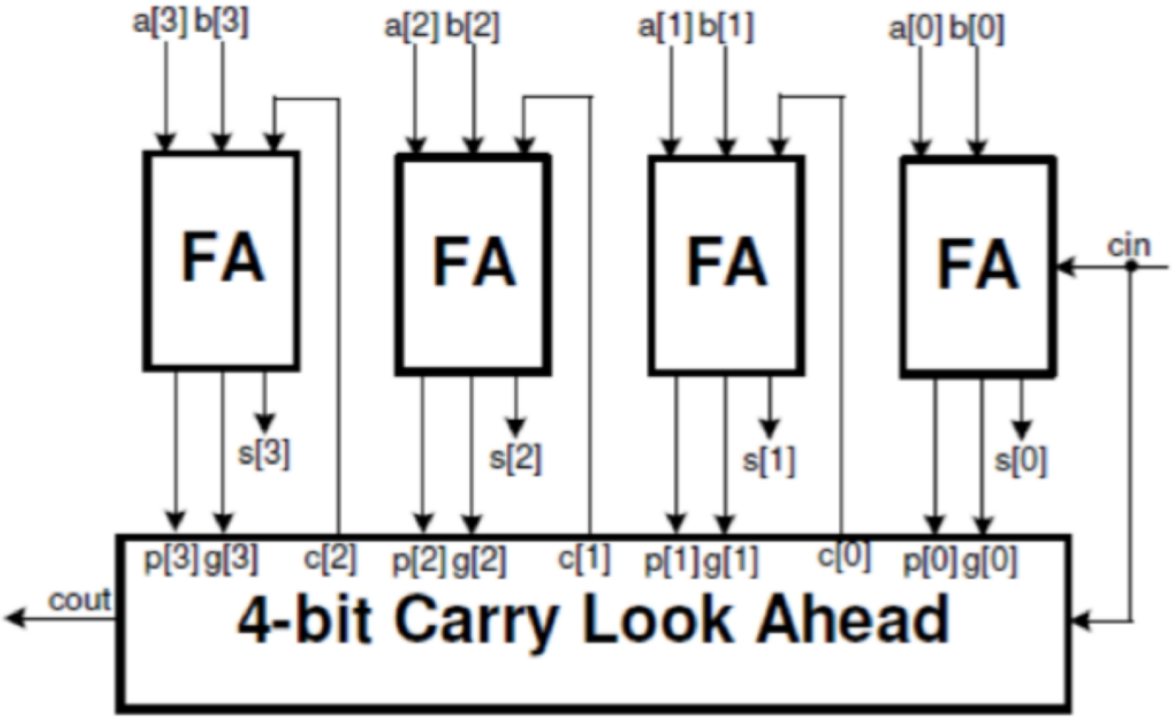


Figure 24: Functional Block Diagram for Carry Look Ahead Adder

### Synthesis Results

Figure 26 shows the synthesized schematic of the design.

### Implementation Results

Show the project summary and comment on the estimated LUTs and IOs that are used and utilization.
Project Summary is shown in Figure 27.
Figure 28 shows the implemented schematics LUTs:7% of LUTs are utilized.
IO: 1% of IO is utilized.

### Summary and Discussion

In this section, summarize the results for this part of the experiment through tabulation when possible.

| Experiment | Simulation | Theoretical |
|---|---|---|
| The experiment was a success. The sum and carry out bits all followed the truth table when we tested our model on the Zedboard. | No simulation required. | We learned that we could concatenate bits and add them. This allowed us to write concise modules. |

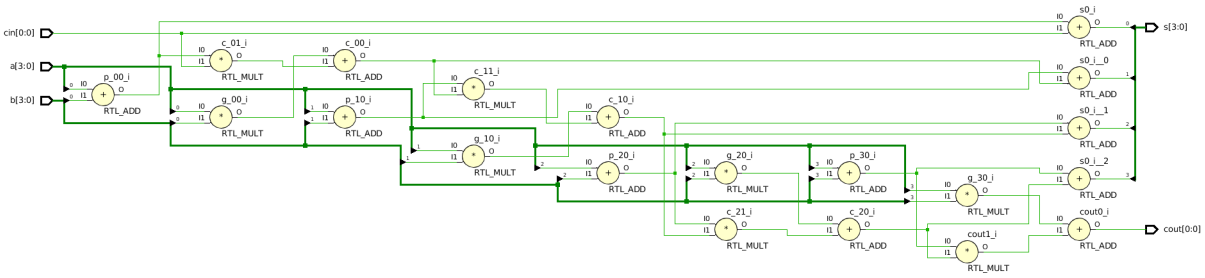


Figure 25: Design Schematics of Carry Look Ahead Adder

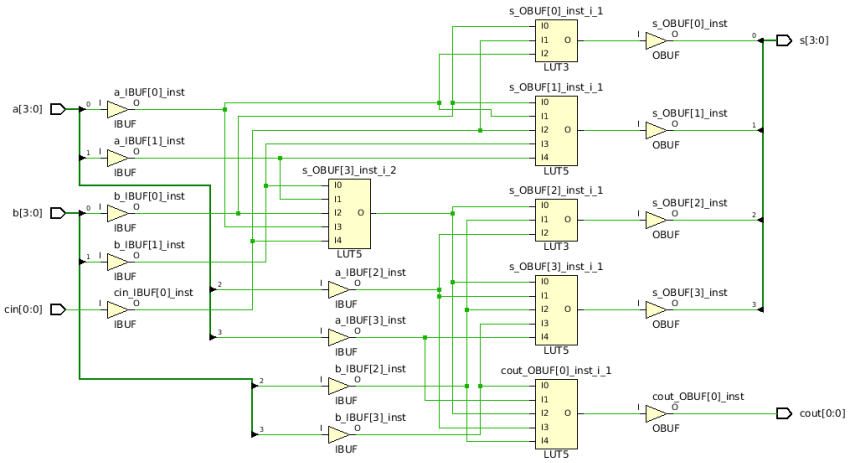


Figure 26: Design Schematics of Carry Look Ahead Adder

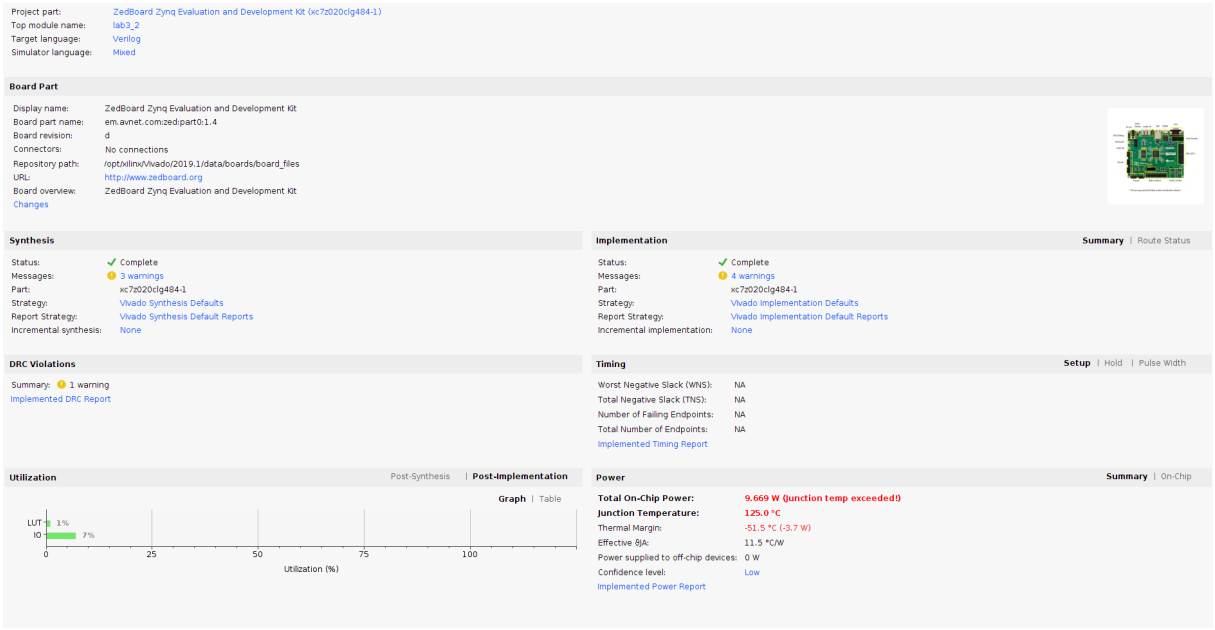


Figure 27: Project Summary of Carry Look Ahead Adder

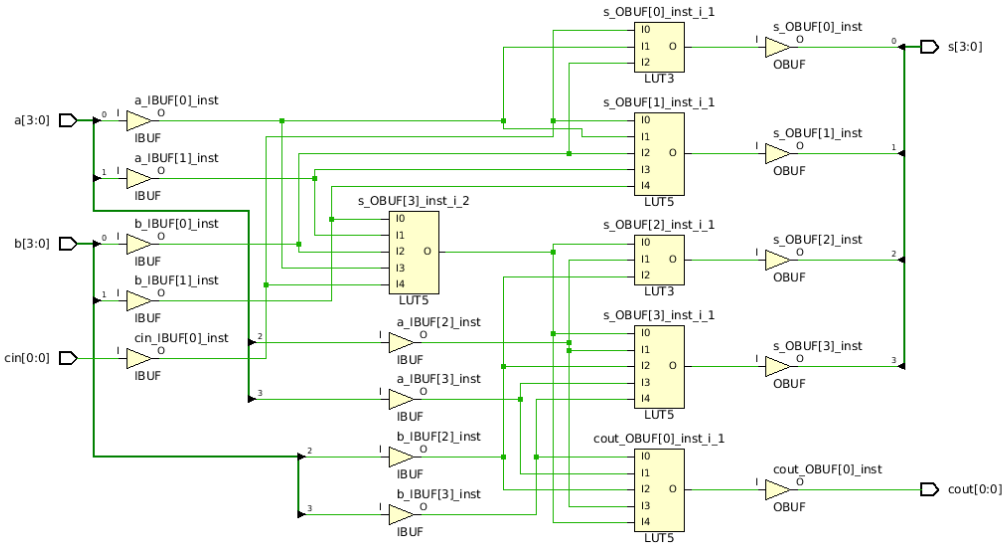


Figure 28: Implemented Schematics of Carry Look Ahead Adder

## Overall Conclusions

In this lab, we learned the basics skills of Verilog, such as writing our own modules, writing testhbench codes to test our modules, simutlating, synthesizing, implementing, and generating bitstreams for our module. We learned how to design module using three different types of modeling: Gate-level, Dataflow, and Behavioral. These skills and models will helps us implement more complex functions later in this course.

## Appendix A Part 1 Code

```
1  `timescale 1ns / 1ps
2
3
4
5  module lab2(
6
7      i1, i2, addr, muxout
8      );
9
10     input wire [1:0]i1;
11     input wire [1:0]i2;
12     input wire [0:0]addr;
13     wire [0:0]notaddr;
14     wire [1:0]out1;
15     wire [1:0]out2;
16     output wire [1:0]muxout;
17
18     not(notaddr, addr);
19     and(out1[0], i1[0], addr);
20     and(out1[1], i1[1], addr);
21     and(out2[0], i2[0], notaddr);
22     and(out2[1], i2[1], notaddr);
23     or(muxout[0], out1[0], out2[0]);
24     or(muxout[1], out1[1], out2[1]);
25
26
27 endmodule
28
```

Figure 29: Design Source code for Part 1

## Appendix B Part 2 Code

```
1  `timescale 1ns / 1ps
2
3  module lab2_2(
4      i1, i2, addr, muxout
5      );
6      input wire [1:0]i1;
7      input wire [1:0]i2;
8      input wire [0:0]addr;
9      output wire [1:0]muxout;
10         assign #3 muxout[0]=addr? i2[0]:i1[0];
11         assign #3 muxout[1]=addr? i2[1]:i1[1];
12 endmodule
13
```

Figure 30: Design Source code for Part 2

# Appendix C Part 3 Code

```
1  `timescale 1ns / 1ps
2
3
4  module lab2_3(
5      i1,
6      i2,
7      addr,
8      muxout
9      );
10         //what are the input ports.
11     input wire [1:0] i1;
12     input wire [1:0] i2;
13     input wire [0:0] addr;
14     //What are the output ports.
15     output wire [1:0] muxout;
16     //Internal variables.
17     reg [1:0] muxout;
18
19     //Always block - the statements inside this block are executed when the given sensitivity list
20     //is satidfied. for example in this case the block is executed when any changes occur in the three signals
21     //named 'Data_in_0','Data_in_1' or 'sel'.
22     always @(i1,i2,addr)
23     begin
24         if(addr == 0)
25             begin
26                 muxout[0] = i1[0];  //when select signal to the mux is low
27                 muxout[1] = i1[1];  //when select signal to the mux is low
28             end
29         else
30             begin
31                 muxout[0] = i2[0];  //when select signal to the mux is high
32                 muxout[1] = i2[1];  //when select signal to the mux is high
33             end
34     end
35
36 endmodule
37
```

Figure 31: Design Source code for Part 3

# Appendix D Part 4 Code

```
1  `timescale 1ns / 1ps
2
3  module lab2_4(
4      i1,
5      i2,
6      i3,
7      addr,
8      muxout
9      );
10     input wire [1:0] i1;
11     input wire [1:0] i2;
12     input wire [1:0] i3;
13     input wire [1:0] addr;
14     //What are the output ports.
15     output wire [1:0] muxout;
16     //Internal variables.
17     reg [1:0] muxout;
18
19     //Always block - the statements inside this block are executed when the given sensitivity list
20     //is satidfied. for example in this case the block is executed when any changes occur in the three signals
21     //named 'Data_in_0','Data_in_1' or 'sel'.
22     always @(i1,i2, i3, addr)
23     begin
24     if(addr[1] == 0)
25         begin
26             if(addr[0] == 0)
27                 begin
28                     muxout[0] = i1[0];  //when select signal to the mux is low
29                     muxout[1] = i1[1];  //when select signal to the mux is low
30                 end
31             else
32                 begin
33                     muxout[0] = i2[0];  //when select signal to the mux is low
34                     muxout[1] = i2[1];  //when select signal to the mux is low
35                 end
36         end
37     else
38         begin
39             if(addr == 0)
40                 begin
41                     muxout[0] = i3[0];  //when select signal to the mux is low
42                     muxout[1] = i3[1];  //when select signal to the mux is low
43                 end
44             else
45                 begin
46                     muxout[0] = i3[0];  //when select signal to the mux is low
47                     muxout[1] = i3[1];  //when select signal to the mux is low
48                 end
49         end
50     end
51 endmodule
```

Figure 32: Design Source code for Part 4

## Appendix E Part 5 Code

```
1  `timescale 1ns / 1ps
2
3
4  module lab3_1(
5      a, b, cin, s, cout
6      );
7      input wire [3:0]a;
8      input wire [3:0]b;
9      input wire [0:0]cin;
10     output wire [3:0]s;
11     output wire [0:0]cout;
12         assign {cout,s} = a + b + cin;
13 endmodule
14
```

Figure 33: Design Source code for Part 5

## Appendix F Part 6 Code

```
1  `timescale 1ns / 1ps
2
3
4  module lab3_2(
5      a, b, cin, s, cout,
6      );
7      input wire [3:0]a;
8      input wire [3:0]b;
9      input wire [0:0]cin;
10     wire [3:0]p;
11     wire [3:0]g;
12     output wire [3:0]s;
13     wire [2:0]c;
14     output wire [0:0]cout;
15         assign {p[0]} = a[0] + b[0];
16         assign {p[1]} = a[1] + b[1];
17         assign {p[2]} = a[2] + b[2];
18         assign {p[3]} = a[3] + b[3];
19         assign {g[0]} = a[0] * b[0];
20         assign {g[1]} = a[1] * b[1];
21         assign {g[2]} = a[2] * b[2];
22         assign {g[3]} = a[3] * b[3];
23         assign {c[0]} = g[0] + (p[0] * cin);
24         assign {c[1]} = g[1] + (p[1] * c[0]);
25         assign {c[2]} = g[2] + (p[2] * c[1]);
26         assign {cout} = g[3] + (p[3] * c[2]);
27         assign {s[0]} = p[0] + cin;
28         assign {s[1]} = p[1] + c[0];
29         assign {s[2]} = p[2] + c[1];
30         assign {s[3]} = p[3] + c[2];
31 endmodule
32
```

Figure 34: Design Source code for Part 6

# Appendix G Testbench Code for Part 2

```
1  `timescale 1ns / 1ps
2
3  module lab2_2test;
4      reg [1:0]a;
5      reg [1:0]b;
6      reg [0:0]cin;
7      wire [1:0]cout;
8      lab2_2 DUT (.i1(a), .i2(b), .addr(cin), .muxout(cout));
9      initial
10         begin
11         a = 0; b = 0; cin = 0;
12         #10 a = 1;
13         #10 b = 1; a = 0;
14         #10 a = 1;
15         #10 cin = 1; a = 0; b = 0;
16         #10 a = 1;
17         #10 b = 1; a = 0;
18         #10 a = 1;
19         #10;
20         end
21 endmodule
```

Figure 35: Simulation Source code for Part 2

# Appendix H Testbench Code for Parts 5 and 6

```
1  `timescale 1ns / 1ps
2
3  module lab3_1test;
4      reg [3:0]a;
5      reg [3:0]b;
6      reg [0:0]cin;
7      wire [0:0]cout;
8      wire [3:0]s;
9      lab3_1 DUT (.a(a), .b(b), .cin(cin), .cout(cout), .s(s));
10     initial
11     begin
12     a = 0; b = 0; cin = 0;
13     #10 a = 1;
14     #10 b = 1; a = 0;
15     #10 a = 1;
16     #10 cin = 1; a = 0; b = 0;
17     #10 a = 1;
18     #10 b = 1; a = 0;
19     #10 a = 1;
20     #10;
21     end
22 endmodule
23
```

Figure 36: Simulation Source code for Parts 5 and 6

# Appendix I Constraint File for Parts 1 to 4

```
# Assign inputs/outputs to actual pins on the FPGA
set_property PACKAGE_PIN M15 [get_ports {i1[0]}]
set_property PACKAGE_PIN H17 [get_ports {i1[1]}]
set_property PACKAGE_PIN H18 [get_ports {i2[0]}]
set_property PACKAGE_PIN H19 [get_ports {i2[1]}]
set_property PACKAGE_PIN F21 [get_ports {addr[0]}]
set_property PACKAGE_PIN U14 [get_ports {muxout[0]}]
set_property PACKAGE_PIN U19 [get_ports {muxout[1]}]
# Define voltage levels (3.3 for LEDs and 1.8 for Switches)
set_property IOSTANDARD LVCMOS18 [get_ports {i1[0]}]
set_property IOSTANDARD LVCMOS18 [get_ports {i1[1]}]
set_property IOSTANDARD LVCMOS18 [get_ports {i2[0]}]
set_property IOSTANDARD LVCMOS18 [get_ports {i2[1]}]
set_property IOSTANDARD LVCMOS18 [get_ports {addr[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {muxout[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {muxout[1]}]
```

Figure 37: Constraint File for Parts 1 to 4

# Appendix J Constraint File for Parts 5 to 6

```
# Assign inputs/outputs to actual pins on the FPGA
set_property PACKAGE_PIN M15 [get_ports {a[0]}]
set_property PACKAGE_PIN H17 [get_ports {a[1]}]
set_property PACKAGE_PIN H18 [get_ports {a[2]}]
set_property PACKAGE_PIN H19 [get_ports {a[3]}]
set_property PACKAGE_PIN F21 [get_ports {b[0]}]
set_property PACKAGE_PIN H22 [get_ports {b[1]}]
set_property PACKAGE_PIN G22 [get_ports {b[2]}]
set_property PACKAGE_PIN F22 [get_ports {b[3]}]
set_property PACKAGE_PIN R16 [get_ports {cin[0]}]
set_property PACKAGE_PIN U14 [get_ports {s[0]}]
set_property PACKAGE_PIN U19 [get_ports {s[1]}]
set_property PACKAGE_PIN W22 [get_ports {s[2]}]
set_property PACKAGE_PIN V22 [get_ports {s[3]}]
set_property PACKAGE_PIN U21 [get_ports {cout[0]}]
# Define voltage levels (3.3 for LEDs and 1.8 for Switches)
set_property IOSTANDARD LVCMOS18 [get_ports {a[0]}]
set_property IOSTANDARD LVCMOS18 [get_ports {a[1]}]
set_property IOSTANDARD LVCMOS18 [get_ports {a[2]}]
set_property IOSTANDARD LVCMOS18 [get_ports {a[3]}]
set_property IOSTANDARD LVCMOS18 [get_ports {b[0]}]
set_property IOSTANDARD LVCMOS18 [get_ports {b[1]}]
set_property IOSTANDARD LVCMOS18 [get_ports {b[2]}]
set_property IOSTANDARD LVCMOS18 [get_ports {b[3]}]
set_property IOSTANDARD LVCMOS18 [get_ports {cin[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {s[0]}]
set_property IOSTANDARD LVCMOS33 [get_ports {s[1]}]
set_property IOSTANDARD LVCMOS33 [get_ports {s[2]}]
set_property IOSTANDARD LVCMOS33 [get_ports {s[3]}]
set_property IOSTANDARD LVCMOS33 [get_ports {cout[0]}]
```

Figure 38: Constraint File for Parts 5 to 6