

# Lab Report 2

- Nathaniel Fargo
- Sept. 25, 2024
- ECE 3700

## Introduction

In this lab different adder circuits are implemented and tested. Adding in base two involves handling small to large amounts of digits in a cascading manner, as the next most significant bit always depends on many of the least significant bits that come before it. Because of the repetitive nature, circuits can take simple adders: 1, 2 or more bit adders, and string them together in a pattern. Alternatively, more complex designs can be developed that save time by taking into account more bits at each step of the way.

## Part 1: Ripple Adder

The first assignment is to design a 4-bit ripple carry adder. We first started with a full 1-bit adder and then strung those together to create a 4-bit adder. We took the “Cout” of each adder and inputted it into the “Cin” of the next adder. Below is the code for both adders.

```

1  module Full_Adder (
2      input a,
3      input b,
4      input cin,
5      output sum,
6      output cout
7  );
8
9      assign sum = a ^ b ^ cin;
10     assign cout = a & b | b & cin | cin & a;
11
12 endmodule
13

```

This type of adder works by looking at each pair of two bits and determining the output. If both are one, a bit is carried to the addition of the next pair. For example  $11 + 11$  would end up carrying one bit to the second bit, which would be  $1 + 1 + 1$ , which would carry one bit to the overflow and set the second bit as 1. Thus an xor is used to determine whether the current bit

```

1  module Four_Bit_Adder (
2      input [3:0] a,
3      input [3:0] b,
4      input cin,
5      output [3:0] out,
6      output cout
7  );
8
9      wire [4:0] c;
10     assign c[0] = cin; // initial carry
11
12     genvar i;
13     generate
14         for (i = 0; i < 4; i = i + 1) begin : adders
15             Full_Adder adder_inst (
16                 .a(a[i]),
17                 .b(b[i]),
18                 .cin(c[i]),
19                 .sum(out[i]),
20                 .cout(c[i+1])
21             );
22         end
23     endgenerate

```

should stay at one based on whether an odd number of bits present are being added. From these conclusions we can derive:

$$sum = a \oplus b \oplus c_{in}$$

$$c_{out} = a \& b | b \& c_{in} | c_{in} \& a$$

Again, “cout” is high if a bit is carried to the next place, so we check if at least two of the inputs, (a, b, or cin) are high.

Next, a simulation of this code is run and check that it adds numbers correctly. Below is provided the code used to test it and the corresponding output. The script tested each possible input case, but only a handful of the total 512 ( $16^3 \times 2$ ) cases are shown below for conciseness. All cases that were tested were validated to be correct.

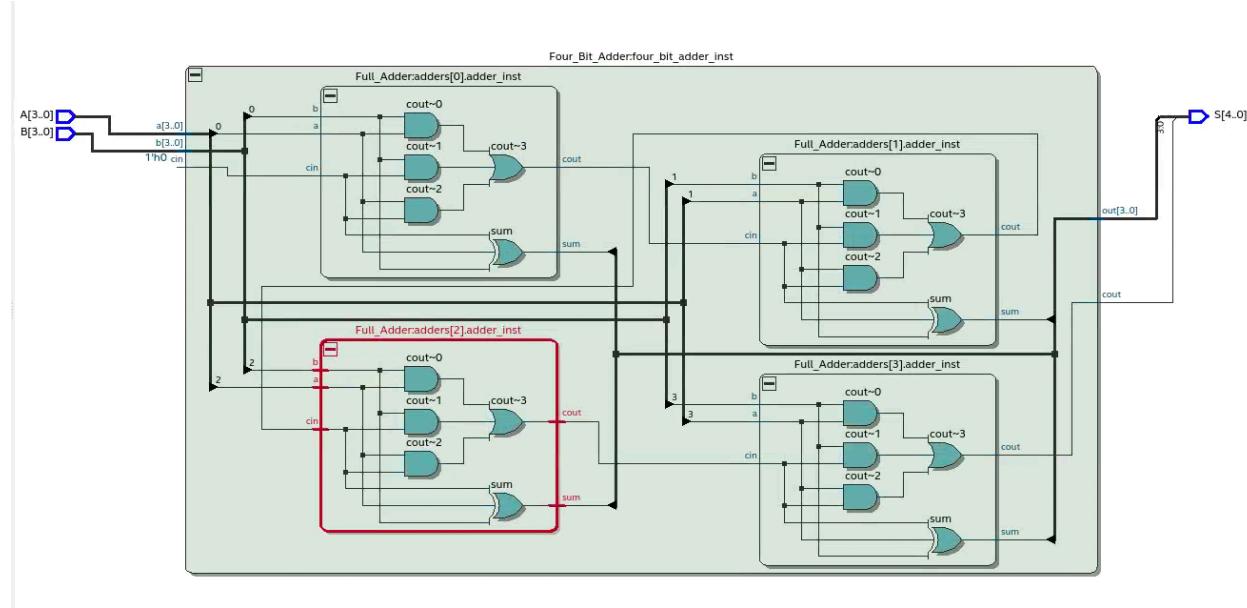
```

1  module tb_Four_Bit_Adder;
2
3    reg [3:0] a, b;
4    reg cin;
5    wire [3:0] out;
6    wire cout;
7
8    /*
9     * Four_Bit_Adder uut (
10      .a(a),
11      .b(b),
12      .cin(cin),
13      .out(out),
14      .cout(cout)
15    );
16   */
17
18   Look_Ahead_Four_Bit_Adder uut2 (
19     .a(a),
20     .b(b),
21     .cin(cin),
22     .out(out),
23     .cout(cout)
24   );
25
26
27   integer n, i, j;
28   initial begin
29     for (n = 0; n < 2; n = n + 1) begin
30       cin = n;
31       for (i = 0; i < 16; i = i + 1) begin
32         a = i;
33         for (j = 0; j < 16; j = j + 1) begin
34           b = j;
35           #10;
36           $display(" ");
37           $display("cin is %d", cin);
38           $display("a is %d", a);
39           $display("b is %d", b);
40           $display("sum is %d", out);
41           $display("cout is %d", cout);
42         end
43       end
44     end
45   end
46
47 endmodule

```

Sum is 0  
# cout is 0  
#  
# cin is 1  
# a is 6  
# b is 2  
# sum is 9  
# cout is 0  
#  
# cin is 1  
# a is 6  
# b is 3  
# sum is 10  
# cout is 0  
#

Next we look at the space and timing taken up by this circuit. First the RTL schematic is shown below.



We can see it takes up a acceptable amount of space, only five gates per bit, but the carry outputs loop backwards to the carry inputs, meaning a time delay of upwards of 8 gates is associated with this circuit.

Next we look at the reports generated to quantify the amount of time and logic elements used. These reports found that **8/49760** logic elements were used.

Looking at the timing of this design, we see that the maximum rise time and fall time are 9.750 and 9.760 respectively. We can compare this later to the look-ahead adder type.

	Input Port	Output Port	RR	RF	FR	FF	▲
1	B[1]	S[4]	9.750			9.760	
2	B[1]	S[3]	9.588	9.469	9.824	9.670	
3	A[0]	S[4]	9.600			9.599	
4	B[0]	S[4]	9.496			9.511	
5	A[0]	S[3]	9.438	9.319	9.663	9.509	
6	B[0]	S[3]	9.334	9.215	9.575	9.421	

## Part 2: Look-Ahead Adder

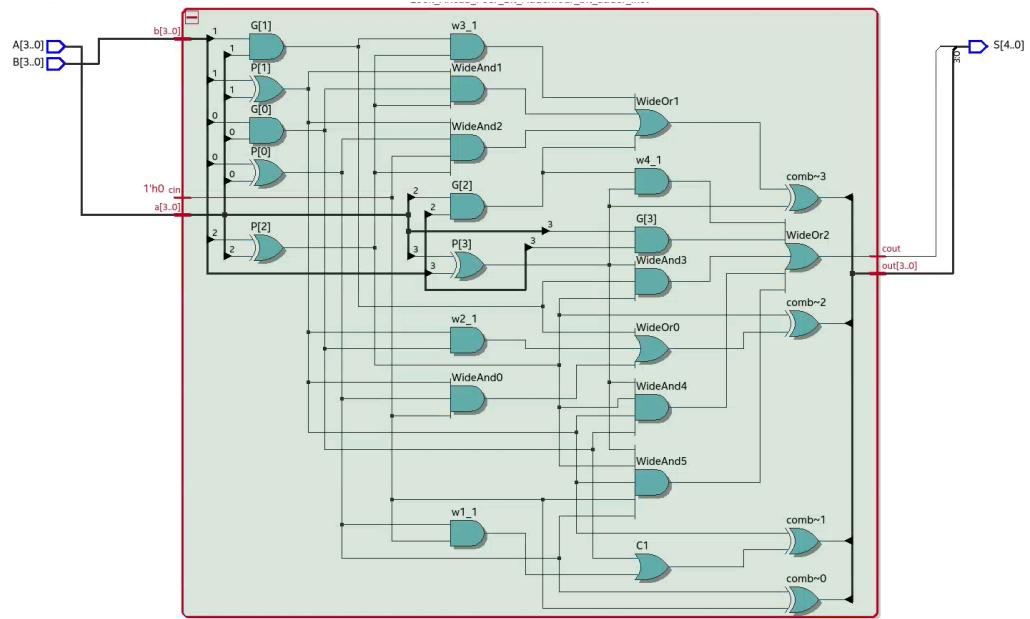
The look ahead adder uses a generate-propagate structure to run faster. This helps remove the time it takes to propagate all the carry signals since the outputs are prepared ahead of time. Here's the verilog code below that executes a 4-bit adder. This code was written structurally to ensure that Quartus implemented it correctly. The formulas for each carry are listed in the comments and executed as shown using a series of wires. At the end these carry and the propagate signals are combined to form the final summation, sending the MSB to "cout" as overflow, like before.

```

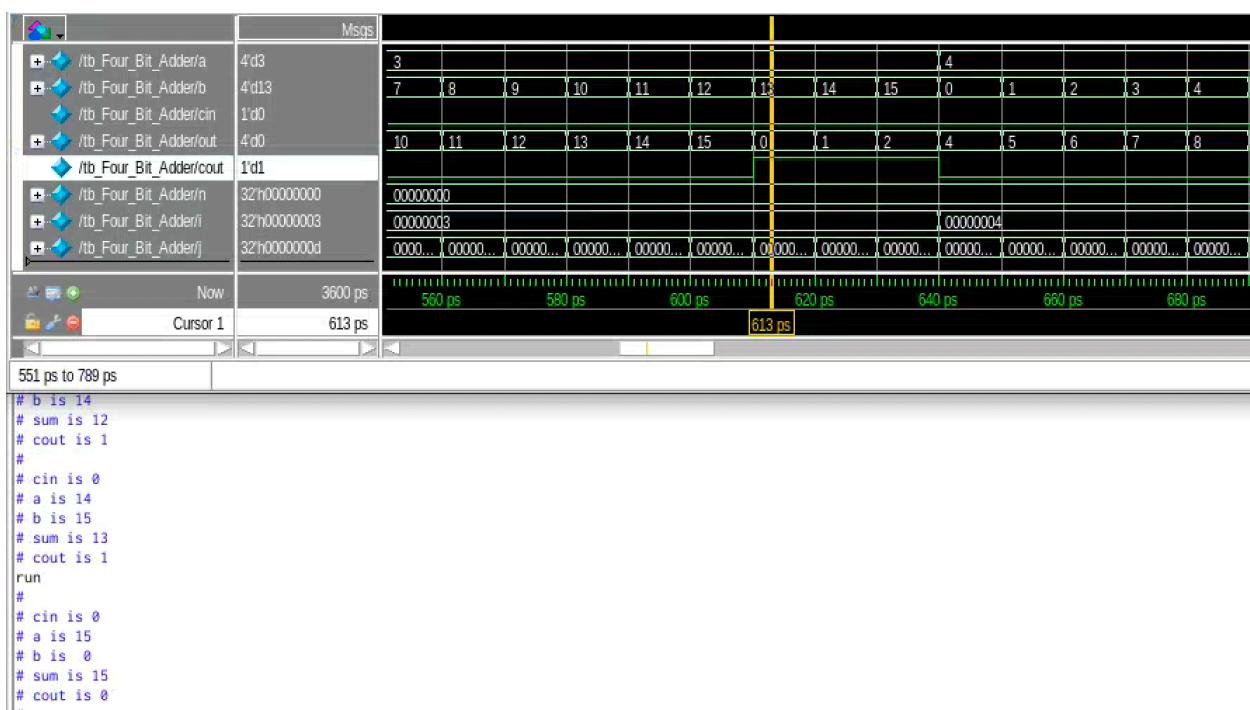
1  module Look_Ahead_Four_Bit_Adder (
2      input [3:0] a,
3      input [3:0] b,
4      input cin,
5      output [3:0] out,
6      output cout
7  );
8
9      wire [3:0] P; // Propagate signals
10     wire [3:0] G; // Generate signals
11     wire C1, C2, C3; // Intermediate carry signals
12
13     // Instantiate XOR gates for Propagate signals: P = A XOR B (may or may not carry)
14     xor (P[0], a[0], b[0]);
15     xor (P[1], a[1], b[1]);
16     xor (P[2], a[2], b[2]);
17     xor (P[3], a[3], b[3]);
18
19     // Instantiate AND gates for Generate signals: G = A AND B (will carry)
20     and (G[0], a[0], b[0]);
21     and (G[1], a[1], b[1]);
22     and (G[2], a[2], b[2]);
23     and (G[3], a[3], b[3]);
24
25     // C1 = G[0] + (P[0] * Cin)
26     wire w1_1;
27     and (w1_1, P[0], cin);
28     or (C1, G[0], w1_1);
29
30     // C2 = G[1] + (P[1] * G[0]) + (P[1] * P[0] * Cin)
31     wire w2_1, w2_2;
32     and (w2_1, P[1], G[0]);
33     and (w2_2, P[1], P[0], cin);
34     or (C2, G[1], w2_1, w2_2);
35
36     // C3 = G[2] + (P[2] * G[1]) + (P[2] * P[1] * G[0]) + (P[2] * P[1] * P[0] * Cin)
37     wire w3_1, w3_2, w3_3;
38     and (w3_1, P[2], G[1]);
39     and (w3_2, P[2], P[1], G[0]);
40     and (w3_3, P[2], P[1], P[0], cin);
41     or (C3, G[2], w3_1, w3_2, w3_3);
42
43     // Cout = G[3] + (P[3] * G[2]) + (P[3] * P[2] * G[1]) + (P[3] * P[2] * P[1] * G[0]) + (P[3] * P[2] * P[1] * P[0] * Cin)
44     // so big
45     wire w4_1, w4_2, w4_3, w4_4;
46     and (w4_1, P[3], G[2]);
47     and (w4_2, P[3], P[2], G[1]);
48     and (w4_3, P[3], P[2], P[1], G[0]);
49     and (w4_4, P[3], P[2], P[1], P[0], cin);
50     or (cout, G[3], w4_1, w4_2, w4_3, w4_4);
51
52     // Sum calculation: Sum = P XOR C (for each bit)
53     xor (out[0], P[0], cin);
54     xor (out[1], P[1], C1);
55     xor (out[2], P[2], C2);
56     xor (out[3], P[3], C3);
57
58 endmodule

```

The RTL version of this is provided below. Because it was written in structural verilog, it is not hard to verify that the internal logic matches the schematic quite well.



The simulation for this test was identical to the one for the ripple carry save that the Look\_Ahead\_Four\_Bit\_Adder module was instantiated instead of the ripple carry adder. This simulation was run, and the outputs of the console were identically correct. Below is provided a waveform of these signals changing along with the console output. At the cursor one can observe  $3 \text{ (a)} + 13 \text{ (b)} + 0 \text{ (cin)} = 0 \text{ (out)} + 1 \text{ (cout)} * 16$ . This is the expected behavior.



Next, looking at the reports, we observe here that **10/49760** elements were used, two more than the ripple version. However, looking at the timing report we find that this version operates slightly faster, at about **9.65** instead of **9.75**. This is a clear example of area vs timing constraints opposing each other.

	Input Port	Output Port	RR	RF	FR	FF
1	B[0]	S[4]	9.649			9.662
2	A[0]	S[4]	9.634			9.659
3	B[1]	S[4]	9.464			9.471
4	B[3]	S[3]	9.206	8.998	9.332	9.124
5	B[0]	S[3]	9.129	8.958	9.315	9.069
6	B[0]	S[2]	9.127	8.950	9.295	9.115

We also flashed this program to the FPGA to verify that it works in real life, which it did.

## Part 3: Eight Bit Adder

For this final adder two of the look-ahead adders were connected via ripple-carry to form a full eight bit adder. The verilog code and simulation results are shown below.

```

1  module Eight_Bit_Adder (
2      input [7:0] a,
3      input [7:0] b,
4      input cin,
5      output [7:0] out,
6      output cout
7  );
8
9      wire carry;
10
11     Look_Ahead_Four_Bit_Adder adder1 (
12         .a(a[3:0]),
13         .b(b[3:0]),
14         .cin(cin),
15         .out(out[3:0]),
16         .cout(carry)
17     );
18
19     Look_Ahead_Four_Bit_Adder adder2 (
20         .a(a[7:4]),
21         .b(b[7:4]),
22         .cin(carry),
23         .out(out[7:4]),
24         .cout(cout)
25     );
26
27 endmodule

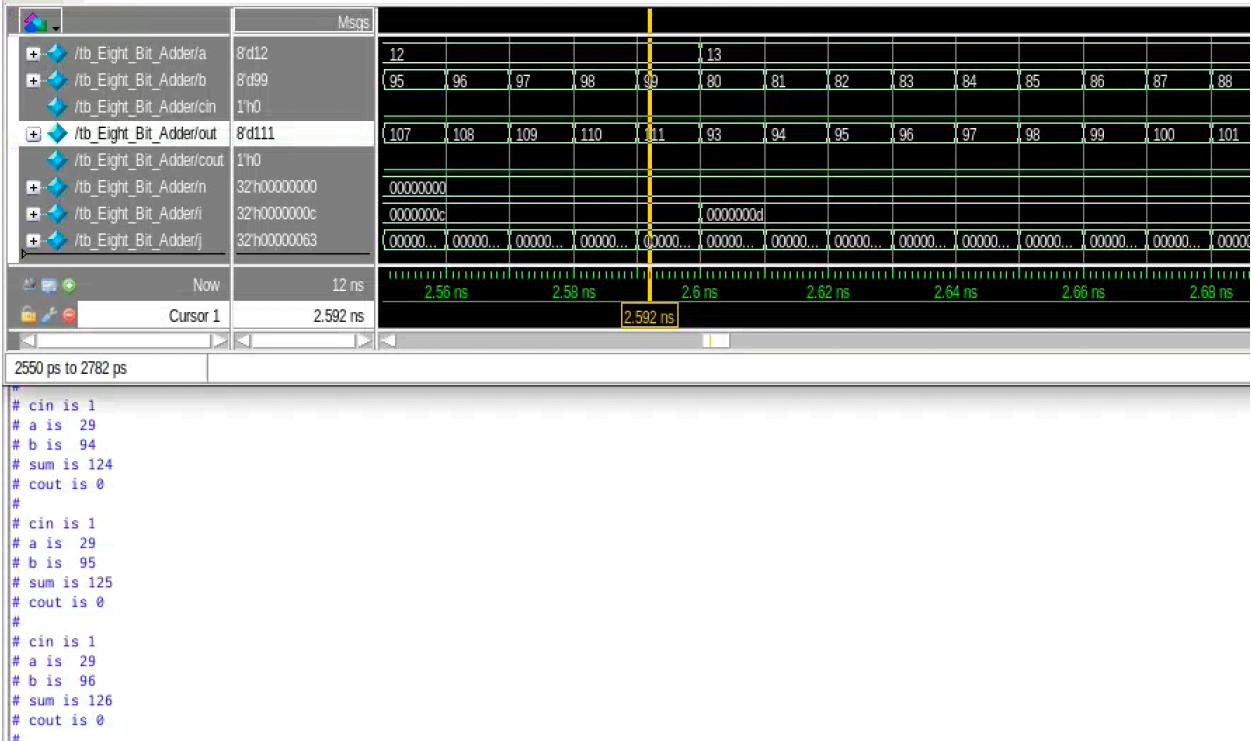
```

The simulation for this adder is very similar to the last one but tests a select range of samples rather than all 131072 ( $2^{17}$ ) possibilities. Provided below are both the code and the simulations results from this section.

```

1  module tb_Eight_Bit_Adder;
2
3      reg [7:0] a, b;
4      reg cin;
5      wire [7:0] out;
6      wire cout;
7
8      Eight_Bit_Adder uut (
9          .a(a),
10         .b(b),
11         .cin(cin),
12         .out(out),
13         .cout(cout)
14     );
15
16     integer n, i, j;
17     initial begin
18         for (n = 0; n < 2; n = n + 1) begin
19             cin = n;
20             for (i = 0; i < 30; i = i + 1) begin
21                 a = i;
22                 for (j = 80; j < 100; j = j + 1) begin
23                     b = j;
24                     #10;
25                     $display(" ");
26                     $display("cin is %d", cin);
27                     $display("a is %d", a);
28                     $display("b is %d", b);
29                     $display("sum is %d", out);
30                     $display("cout is %d", cout);
31                 end
32             end
33         end
34     end
35
36 endmodule

```



## Conclusion

From this lab we have looked at various adder implementations. Each other these adder types have their own pros and cons, and can even been combined together to balance the trade-offs each type has. More specifically, we looked at the delay between the inputs and outputs of these adders, as well as how many logic elements were used. Overall a lower delay usually required more logic elements, but the change in both time and logic elements was small for the 4-bit adder. These trends likely scale up and can make a big impact in energy critical, size critical, or timing critical applications.