Lab Report 2

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

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.

```
module Full_Adder (
    input a,
    input b,
    input cin,
    output sum,
    output cout
);

    assign sum = a ^ b ^ cin;
    assign cout = a & b | b & cin | cin & a;

endmodule
```

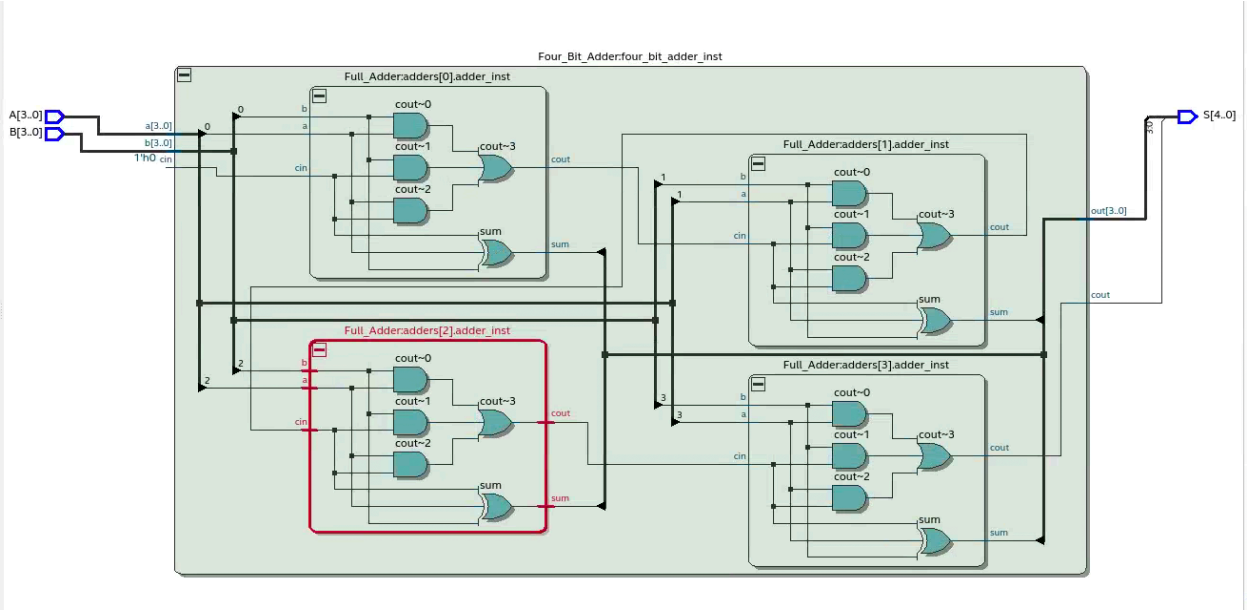
```
module Four_Bit_Adder (
    input [3:0] a,
    input [3:0] b,
    input cin,
    output [3:0] out,
    output cout
);

    wire [4:0] c;
    assign c[0] = cin; // initial carry

    genvar i;
    generate
        for (i = 0; i < 4; i = i + 1) begin : adders
            Full_Adder adder_inst (
                .a(a[i]),
                .b(b[i]),
                .cin(c[i]),
                .sum(out[i]),
                .cout(c[i+1])
            );
        end
    endgenerate
```

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 should stay at one based on whether an odd number of bits present are being added.

```
#
# cin is 1
# a is  7
# b is  6
# sum is 14
# cout is 0
#
# cin is 1
# a is  7
# b is  7
# sum is 15
# cout is 0
#
# cin is 1
# a is  7
# b is  8
# sum is  0
# cout is 1
#
```

Next, a simulation of this code is run and check that it adds numbers correctly. All results appear to be correct.

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 but the carry outputs loop backwards to the carry inputs.

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.

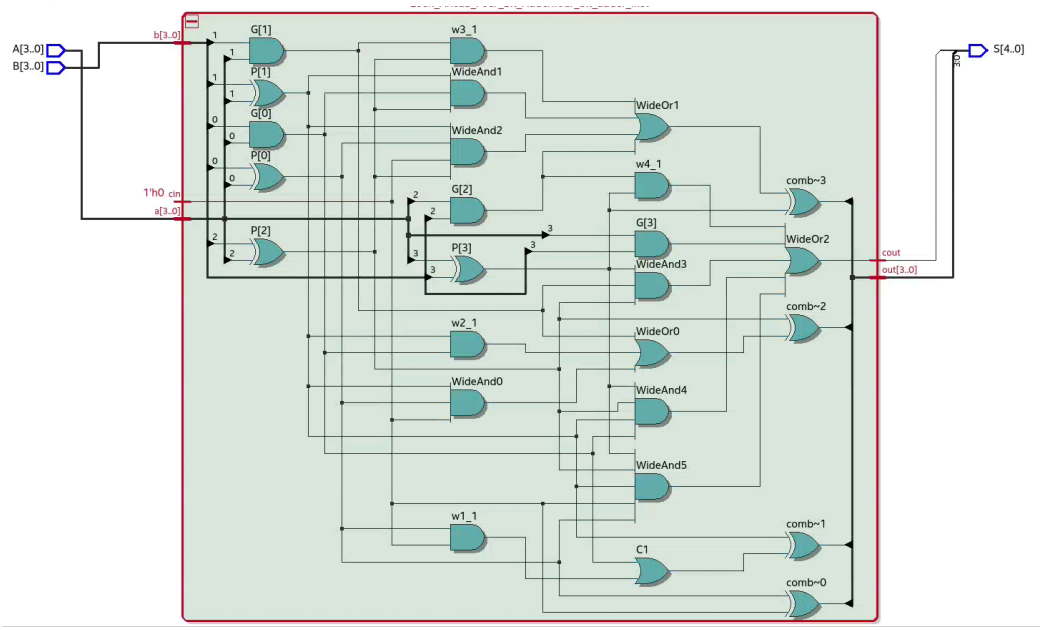
	Input Port	Output Port	RR	RF	FR	FF
1	B[1]	S[4]	9.750			9.760
2	B[1]	S[3]	9.588	9.469	9.824	9.670
3	A[0]	S[4]	9.600			9.599
4	B[0]	S[4]	9.496			9.511
5	A[0]	S[3]	9.438	9.319	9.663	9.509
6	B[0]	S[3]	9.334	9.215	9.575	9.421

Part 2: Look-Ahead Adder

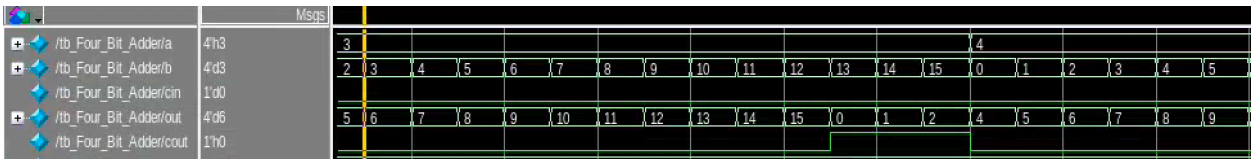
The look ahead adder uses a generate-propagate structure to run faster. This helps remove the time it takes to propagate all the carry signals since the outputs are prepared ahead of time. Here's the verilog code below that executes a 4-bit adder. This code was written structurally to ensure that Quartus implemented it correctly. The formulas for each carry are listed in the commends 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.

```
module Look_Ahead_Four_Bit_Adder (
    input [3:0] a,
    input [3:0] b,
    input cin,
    output [3:0] out,
    output cout
);

    wire [3:0] P; // Propagate signals
    wire [3:0] G; // Generate signals
    wire C1, C2, C3; // Intermediate carry signals

    // Instantiate XOR gates for Propagate signals: P = A XOR B (may or may not carry)
    xor (P[0], a[0], b[0]);
    xor (P[1], a[1], b[1]);
    xor (P[2], a[2], b[2]);
    xor (P[3], a[3], b[3]);

    // Instantiate AND gates for Generate signals: G = A AND B (will carry)
    and (G[0], a[0], b[0]);
    and (G[1], a[1], b[1]);
    and (G[2], a[2], b[2]);
    and (G[3], a[3], b[3]);

    // C1 = G[0] + (P[0] * Cin)
    wire w1_1;
    and (w1_1, P[0], cin);
    or (C1, G[0], w1_1);

    // C2 = G[1] + (P[1] * G[0]) + (P[1] * P[0] * Cin)
    wire w2_1, w2_2;
    and (w2_1, P[1], G[0]);
    and (w2_2, P[1], P[0], cin);
    or (C2, G[1], w2_1, w2_2);

    // C3 = G[2] + (P[2] * G[1]) + (P[2] * P[1] * G[0]) + (P[2] * P[1] * P[0] * Cin)
    wire w3_1, w3_2, w3_3;
    and (w3_1, P[2], G[1]);
    and (w3_2, P[2], P[1], G[0]);
    and (w3_3, P[2], P[1], P[0], cin);
    or (C3, G[2], w3_1, w3_2, w3_3);

    // 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)
    // so big
    wire w4_1, w4_2, w4_3, w4_4;
    and (w4_1, P[3], G[2]);
    and (w4_2, P[3], P[2], G[1]);
    and (w4_3, P[3], P[2], P[1], G[0]);
    and (w4_4, P[3], P[2], P[1], P[0], cin);
    or (cout, G[3], w4_1, w4_2, w4_3, w4_4);

    // Sum calculation: Sum = P XOR C (for each bit)
    xor (out[0], P[0], cin);
    xor (out[1], P[1], C1);
    xor (out[2], P[2], C2);
    xor (out[3], P[3], C3);

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.



A simulation of this module was also run, and the outputs of the console were identical. Below is provided a waveform of these signals changing. At the cursor one can observe 3 + 3 + 0 = 6 very clearly, and once it overflows to 13 + 3 = 16 the result is 0 for out, and a cout of 1 to represent the 16.



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.

```
module Eight_Bit_Adder (
    input [7:0] a,
    input [7:0] b,
    input cin,
    output [7:0] out,
    output cout
    );

    wire carry;

    Look_Ahead_Four_Bit_Adder adder1 (
        .a(a[3:0]),
        .b(b[3:0]),
        .cin(cin),
        .out(out[3:0]),
        .cout(carry)
    );

    Look_Ahead_Four_Bit_Adder adder2 (
        .a(a[7:4]),
        .b(b[7:4]),
        .cin(carry),
        .out(out[7:4]),
        .cout(cout)
    );

endmodule
```

Here's some output from the simulation, which shows that the eight bit adder is working correctly.

```
"
# cin is 1
# a is  23
# b is  90
# sum is 114
# cout is 0
#
# cin is 1
# a is  23
# b is  91
# sum is 115
# cout is 0
#
# cin is 1
# a is  23
# b is  92
# sum is 116
# cout is 0
#
# cin is 1
# a is  23
# b is  93
# sum is 117
# cout is 0
"
```

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.