Lab 1 Report

Objective:

The objective of this lab was to learn how to instantiate in Verilog to create a 4-bit binary adder subtractor.

Methodology:

To create the 4-bit adder subtractor, I started by creating smaller adders to then use in the final 4-bit adder subtractor (fig. 1). I started with making a 1-bit half adder. This was done by simply inputting the two input bits (A & B) into an XOR gate to get the sum, and an AND gate to get the carry.

I then used two of these half adders, instantiating them in Verilog, to create a 1-bit full adder (fig. 3). This was done by having A & B input into the first half adder, with this half adder outputting a sum and a carry. The second half adder then receives the sum from the first half adder and a third input, carry in (C), as its inputs. The carry output from the second half adder is inputted into an OR gate with the carry output from the first half adder. The output of this OR gate was then the final carry output of the full adder, and the sum output from the second half adder was the final sum output of the full adder.

I then used four of these 1-bit adders to create a 4-bit adder (fig. 5). The 4-bit adder has 3 inputs overall: a 4-bit bus A, another 4-bit bus B, and a 1-bit carry in, C. A and B represent the two 4-bit numbers to add. The first 1-bit adder takes in C, and the least significant bits of A and B. The sum output of this adder is then the least significant bit of the final sum output. The carry output of the first adder is inputted into the carry input of the second adder, along with the next significant bits of A and B. This second adder outputs the second least significant bit of the sum output, and the carry output continues to the next adder. This continues until it reaches the fourth adder, which will output the most significant bit of the sum output, giving you the entire 4-bit sum. The carry output of the fourth 1-bit adder is then outputted as the carry out for the full 4-bit adder.

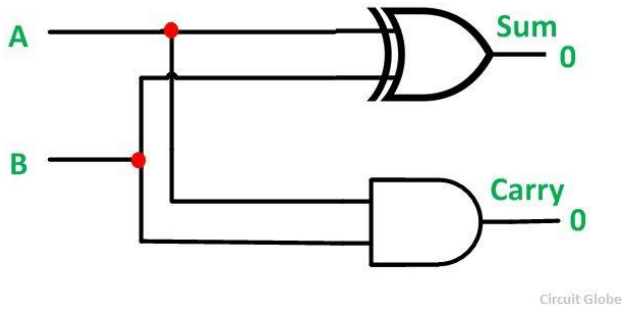
My next step was to use the 4-bit adder in my 4-bit adder subtractor design, but I was not able to do this because the adder subtractor needed access to two of the carry's in my 4-bit adder design, but the adder only outputted one carry, the final one. So instead I determined that the easiest way to design the adder subtractor would be to just use four of the 1-bit adders, similar to the design of the 4-bit adder, but with some additional gates. Since the adder subtractor needs to subtract as well as add, the third input (M instead of C) is used to determine if the numbers should be added or subtracted. So if M is 1, it is a subtraction, and if it is 0, it is an addition. To implement this, I XORed each bit of the B input with M, so that if M is 1, the bits of B will be flipped, and inputted the results of these into each 1-bit adder instead of just B. M is also inputted as the Carry in of the first 1-bit adder, so if it is 1 you add 1 to the numbers. This effectively turns the B number into its 2's complement, allowing you to add the

Alexander Salmi
U04201274

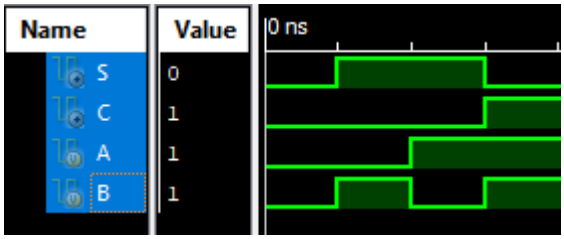
two new numbers to subtract B from A. The rest of the design is the same as with the 4-bit adder, with each 1-bit adder outputting a bit of the final sum, and the final Carry out being the final carry out of the adder subtractor. However, the adder subtractor also has a third output, V. If V is 1, there is overflow. So to represent the output V, I simply put the final carry and the carry between the third and fourth 1-bit adders into an XOR gate, with the output of this gate being the final output V for the adder subtractor. (fig. 7)

Observations:

Overall, all of the designs worked as intended. The 1-bit half adder (fig. 1) correctly added the two bits, noting if there is a carry or not. This is shown in the timing diagram below (fig. 2).

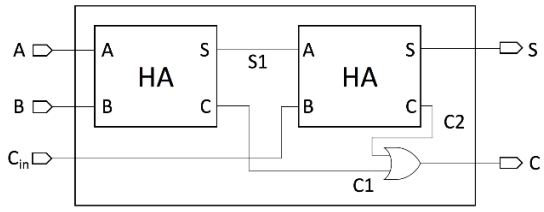


(Figure 1: https://circuitglobe.com/half-adder-and-full-adder-circuit.html)

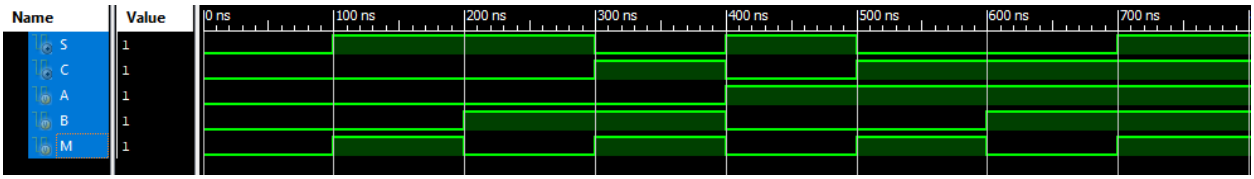


(Figure 2)

The 1-bit full adder (fig. 3) correctly added the two bits, also taking the carry in into account, and outputting both the correct sum and carry out. This is shown in the timing diagram below (fig. 4).



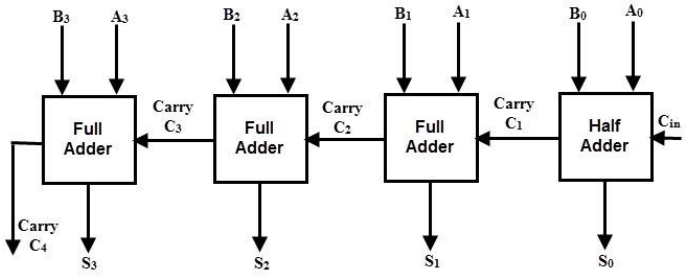
(Figure 3: https://techyelf.blogspot.com/2017/09/vhdl-code-for-full-adder-using-half.html)



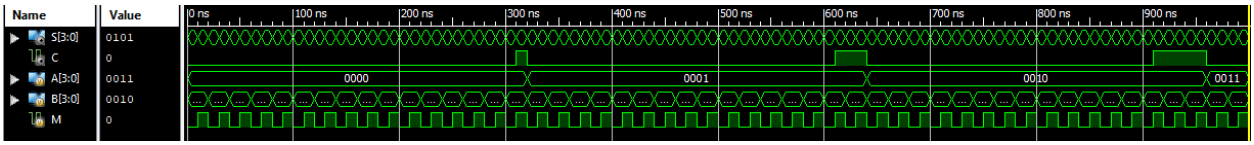
(Figure 4)

The 4-bit adder (fig. 5) also worked correctly, adding all four bits, taking the carry in into account, and outputting the correct carry out. This is shown in the diagram below (fig. 6).

Alexander Salmi
U04201274

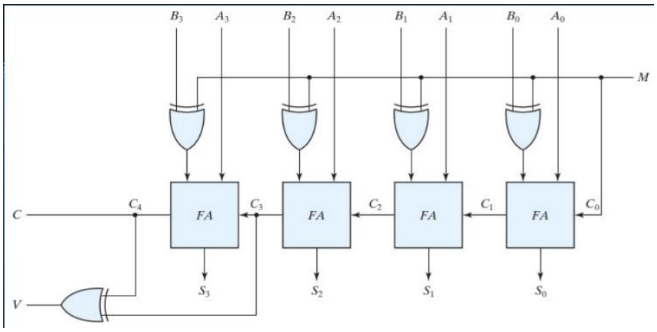


(Figure 5: https://www.electronicshub.org/binary-adder-and-subtractor/)

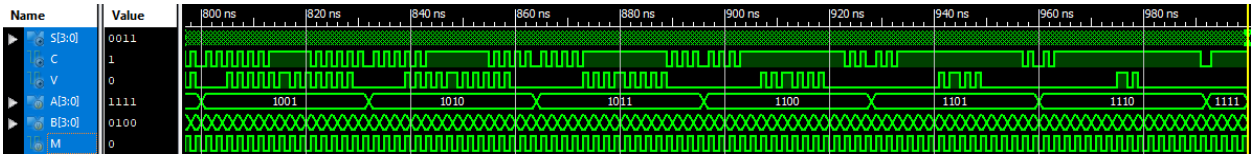


(Figure 6)

Finally, the adder subtractor (fig. 7) also worked as desired. While it is hard to see in the timing diagram (fig. 8) due to the large amount of test instances. When the input M was 1, the design subtracted B from A successfully, and added them when M was 0. Whenever there was overflow, meaning that the resulting sum could not be displayed in 4 bits, the V output was 1, and it was 0 in all other instances.



(Figure 7: https://electronics.stackexchange.com/questions/267966/designing-a-4-bit-adder-subtractor-circuit)



(Figure 8)

Conclusion:

Overall, this was a very successful lab. Not only did I learn how to design adder schematics for both 1 and 4 bits, I also gained a better understanding of carry's, overflow, and how to represent 2's complement in a schematic. Furthermore, I also started becoming more familiarized with Verilog, and how to correctly implement and test a circuit design.

Alexander Salmi
U04201274

Code:

1-Bit Half Adder

Schematic:

```
module OneBitHalfAdder(
   input A,
   input B,
   output S,
   output C
   );

          xor g1(S, A, B);
          and g2(C, A, B);


endmodule
```

Implementation:

```
module OneBitHalfAdderTB;

          // Inputs
          reg A;
          reg B;

          // Outputs
          wire S;
          wire C;

          // Instantiate the Unit Under Test (UUT)
          OneBitHalfAdder uut (
                    .A(A),
                    .B(B),
                    .S(S),
                    .C(C)
          );

          initial begin
                    // Initialize Inputs
                    A = 0; B = 0;

                    // Wait 100 ns for global reset to finish
                    #100; A = 0; B = 1;
                    #100; A = 1; B = 0;
                    #100; A = 1; B = 1;

                    // Add stimulus here

          end

endmodule
```

EC311
Lab 1

Alexander Salmi
U04201274

1-Bit Full Adder

Schematic:

```
module OneBitFullAdder(
  input A,
  input B,
  input M,
  output S,
  output C
  );

        wire w1, w2, w3;

        OneBitHalfAdder g1(.S(w1),.C(w2),.A(A),.B(B));
        OneBitHalfAdder g2(.S(S),.C(w3),.A(w1),.B(M));
        or g3(C, w3, w2);


endmodule
```

Implementation:

```
module OneBitFullAdderTB;

        // Inputs
        reg A;
        reg B;
        reg M;

        // Outputs
        wire S;
        wire C;

        // Instantiate the Unit Under Test (UUT)
        OneBitFullAdder uut (
                .A(A),
                .B(B),
                .M(M),
                .S(S),
                .C(C)
        );

        initial begin
                // Initialize Inputs
                A = 0; B = 0; M = 0;

                // Wait 100 ns for global reset to finish
                #100; A = 0; B = 0; M = 1;
                #100; A = 0; B = 1; M = 0;
                #100; A = 0; B = 1; M = 1;
                #100; A = 1; B = 0; M = 0;
                #100; A = 1; B = 0; M = 1;
                #100; A = 1; B = 1; M = 0;
                #100; A = 1; B = 1; M = 1;

                // Add stimulus here

        end

endmodule
```

EC311
Lab 1

Alexander Salmi
U04201274

4-Bit Adder

Schematic:

```
module FourBitAdder(
  input [3:0] A,
  input [3:0] B,
  input M,
  output [3:0] S,
  output C
  );

        wire w1, w2, w3;

        OneBitFullAdder g1(.S(S[0]),.C(w1),.A(A[0]),.B(B[0]),.M(M));
        OneBitFullAdder g2(.S(S[1]),.C(w2),.A(A[1]),.B(B[1]),.M(w1));
        OneBitFullAdder g3(.S(S[2]),.C(w3),.A(A[2]),.B(B[2]),.M(w2));
        OneBitFullAdder g4(.S(S[3]),.C(C),.A(A[3]),.B(B[3]),.M(w3));


endmodule
```

Implementation:

```
module FourBitAdderTB;

        // Inputs
        reg [3:0] A;
        reg [3:0] B;
        reg M;

        // Outputs
        wire [3:0] S;
        wire C;

        // Instantiate the Unit Under Test (UUT)
        FourBitAdder uut (
                .A(A),
                .B(B),
                .M(M),
                .S(S),
                .C(C)
        );

        initial begin
                // Initialize Inputs
                A = 0;
                B = 0;
                M = 0;

                // Wait 100 ns for global reset to finish
                #100;

                // Add stimulus here
                #5190 $stop;

        end

        always begin
                #10 {A, B, M} = {A, B, M} + 1'b1;
        end

endmodule
```

EC311
Lab 1

Alexander Salmi
U04201274

4-Bit Adder Subtractor

Schematic:

```
module FourBitAddSub(
   input [3:0] A,
   input [3:0] B,
   input M,
   output [3:0] S,
   output C,
   output V
   );

        wire w1, w2, w3, Bw0, Bw1, Bw2, Bw3;

        xor g1(Bw0, B[0], M);
        xor g2(Bw1, B[1], M);
        xor g3(Bw2, B[2], M);
        xor g4(Bw3, B[3], M);

        OneBitFullAdder g5(.S(S[0]),.C(w1),.A(A[0]),.B(Bw0),.M(M));
        OneBitFullAdder g6(.S(S[1]),.C(w2),.A(A[1]),.B(Bw1),.M(w1));
        OneBitFullAdder g7(.S(S[2]),.C(w3),.A(A[2]),.B(Bw2),.M(w2));
        OneBitFullAdder g8(.S(S[3]),.C(C),.A(A[3]),.B(Bw3),.M(w3));

        xor g9(V, w3, C);


endmodule
```

Implementation:

```
module FourBitAddSubTB;

        // Inputs
        reg [3:0] A;
        reg [3:0] B;
        reg M;

        // Outputs
        wire [3:0] S;
        wire C;
        wire V;

        // Instantiate the Unit Under Test (UUT)
        FourBitAddSub uut (
                .A(A),
                .B(B),
                .M(M),
                .S(S),
                .C(C),
                .V(V)
        );

        initial begin
                // Initialize Inputs
                A = 0;
                B = 0;
                M = 0;

                // Wait 100 ns for global reset to finish
                #100;

                // Add stimulus here
                #1000 $stop;

        end

        always begin
                #1 {A, B, M} = {A, B, M} + 1'b1;
        end

endmodule
```