EC311 Alexander Salmi
Lab 1 U04201274

# Lab 2 Report

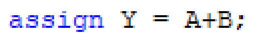
**Objective:**

The objective of this lab was to familiarize ourselves with writing behavioral Verilog code and to design a 4-bit Arithmetic-Logic Unit, that can perform addition, multiplication, concatenation, and shifting.

**Methodology:**

To create the 4-bit ALU, we first had to go through and create all the separate components of the ALU. This means creating a design for each of the ALU's functions: an adder, a multiplier, a concatenator, and a shifter. We also had to make a 4:1 multiplexer to combine all the other designs and only output the desired result. These were all then combined to create the full 4-bit ALU.

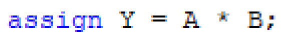
Starting simple seemed like a reasonable approach, so I started with designing the adder. At first, I did not fully grasp how to write behavioral Verilog code, so I mistakenly started designing the adder in structural Verilog at first. After I realized my mistake, and behavioral Verilog was explained to me a bit further, I began doing it correctly, and realized that the code for the adder would be very simple. All that was needed to be done was to assign the output to the sum of the two inputs using the addition syntax (Fig. 1). Since the final design was to be a 4-bit ALU, the two inputs to the adder were 4-bits, however the output was eight bits, even though this many bits is never necessary for the sum of two 4-bit decimal numbers. The output was set as eight bits since some of the other functions of the ALU would require eight bits as an output, such as the concatenator or the shifter. So to ensure that all the functions of the ALU would have the same sized output, and the multiplexer would be able to take them all in as inputs seamlessly, all function designs were set at two 4-bit inputs and one 8-bit output.



```
assign Y = A+B;
```

(Figure 1)

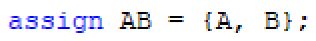
Moving on to the multiplier design, I quickly realized this would also be a simple function to code. Much like the adder, all that needed to be done was to assign the output to the product of the two inputs using the multiplication syntax (Fig. 2). Much like the adder, the two inputs were set to 4-bit and the output set to 8-bit.



```
assign Y = A * B;
```

(Figure 2)

The next function to design was the concatenation function. I expected this one to be more of a challenge than the previous two, as I was not sure if there was a concatenation function in Verilog, and I started thinking of ways to perform the concatenation with boolean operations. However, after some quick research, I found that there was a simple way to design this function as well. All that needed to be done was to assign the output to the result of the built in concatenation function (Fig. 3).



```
assign AB = {A, B};
```

(Figure 3)

The final function design of the shifter turned out to be a little more complicated, but still relatively easy. Since the function’s purpose was to shift the bits of input A to the left by a number of bits determined by input B, I had to start by extending the 4-bit input A to 8-bits, so that it could be shifted all the way. To do this, I simply concatenated four zeros to the front of A, using the concatenation function. The to perform the shifting itself, I found out that there is also a built in function for this. Basically, to shift a number A to the left by B bits, you put to less than symbols in-between A and B (Fig. 4). I set the output Y to the result of this operation, and it was finished.

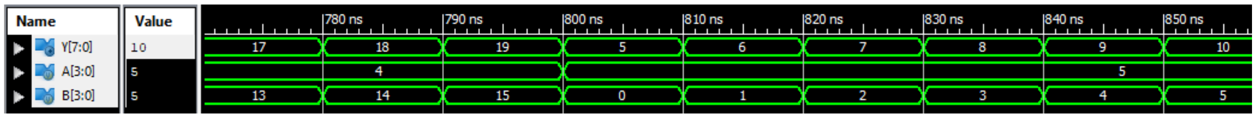
```
assign Y = y_temp << B;
```
(Figure 4)

Next, I needed to design the multiplexer to combine all of these functions and output only one of their outputs, based on the input S chosen by the user. S is a 2-bit input, so that the user can input one of four numbers to choose one of the four functions. If S is 00, the result of the concatenation function should be outputted. If S is 01, the addition function is used. For 10 and 11, the results of left shift and multiplication are outputted respectively. The multiplexer also has four other inputs, all eight bits, which represent the outputs of the four functions. The multiplexer also has an 8-bit output, which is assigned to the result of whichever function the user chooses. To implement this, I simple did a switch case for the input variable S, assigning the output Y to the corrects corresponding input A, B, C, or D, depending of the value of S.

Finally, I combined all of these designs into the 4-bit ALU. Since I had to instantiate the other designs I had made, the code for the ALU would have to be structural Verilog instead of behavioral. The ALU has two 4-bit inputs A and B, which are sent straight to the inputs of each of the four functions as their two inputs. There is also a 2-bit input S which is sent straight to the S input of the multiplexer. The results of each of the functions are then sent to one of the four inputs of the multiplexer, so that each multiplexer input is the output of one of the functions. The only output of the ALU is an 8-bit one, and sends out the output of the function that the user requested with the S-input.

**Observations:**

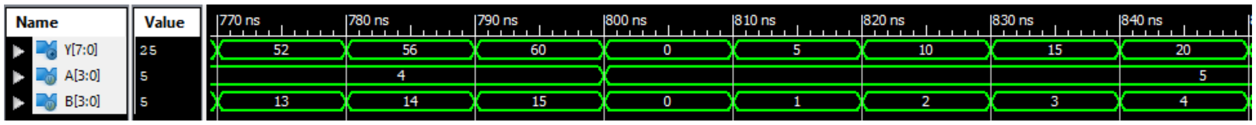
Overall, all of the functions and ALU worked as intended. The adder successfully added the two input bits and converted the sum into and 8-bit output. The timing diagram below (Fig. 5) shows the inputs/outputs as decimal numbers to make it more readable, but as it shows, the addition is done correctly.



(Figure 5)

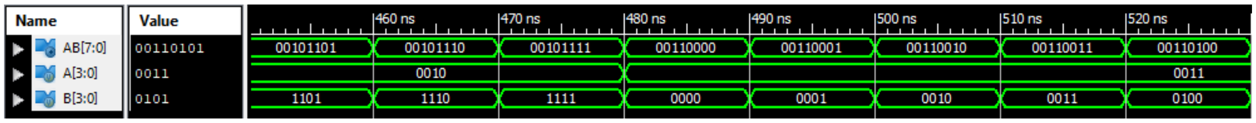
The multiplier correctly multiplies the two inputs to output the 8-bit product. This timing diagram (Fig. 6) is also shown in decimal for clarity.

Alexander Salmi
U04201274

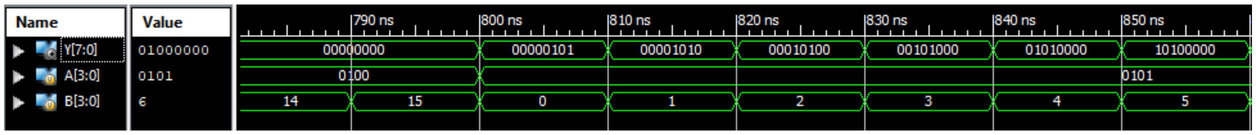


(Figure 6)

The shifter and concatenator also work correctly, as shown in the timing diagrams below (Figure 7 and 8).

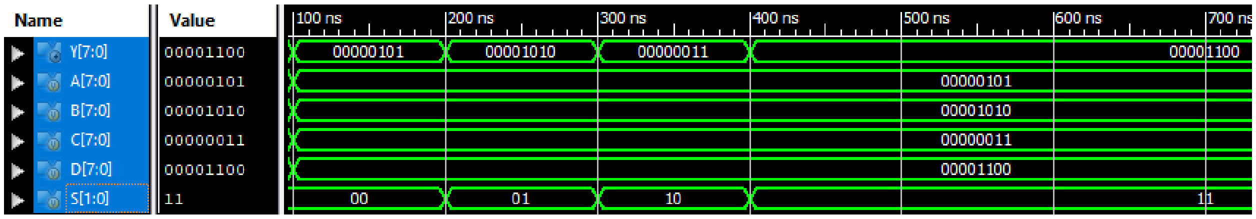


(Figure 7)

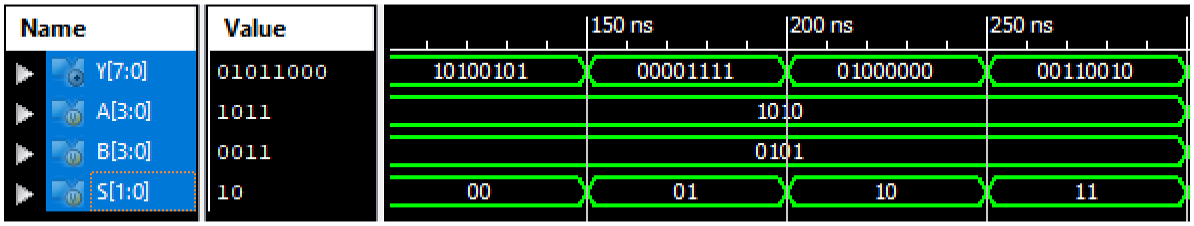


(Figure 8)

The timing diagrams below (Fig. 9 and 10) show the correct function of the multiplexer and ALU, but since they both have a lot of inputs, the diagrams only show what happens when you change the value of S, with constant values for the rest of the inputs. The multiplexer chooses the correct output based on the value of S, and the ALU both performs the correct function and outputs the correct value based on the value of S.



(Figure 9)



(figure 10)

**Conclusion:**

Overall, this was a very successful lab. I became a lot more familiar with writing behavioral Verilog, as I was quite unsure of how to do this before the lab, but am now very comfortable with it. I also learned a lot more about how multiplexers work, and what an ALU is, which will be very valuable information moving forward.

**Code:**

*Adder Behavioral Verilog:*

```
`timescale 1ns / 1ps

module Adder(
   input [3:0] A,
   input [3:0] B,
   output [7:0] Y
   );

             assign Y = A+B;


endmodule
```

---

*Adder Test Bench:*

```
`timescale 1ns / 1ps

module Adder_TB;

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

            // Outputs
            wire [7:0] Y;

            // Instantiate the Unit Under Test (UUT)
            Adder uut (
                        .A(A),
                        .B(B),
                        .Y(Y)
            );

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

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

```
		// Add stimulus here
		#3000 $stop;
	end

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

endmodule
```

---

*Multiplier Behavioral Verilog:*

```
`timescale 1ns / 1ps

module Multiplier(
    input [3:0] A,
    input [3:0] B,
    output [7:0] Y
    );

	assign Y = A * B;

endmodule
```

---

*Multiplier Test bench:*

```
`timescale 1ns / 1ps

module Multiplier_TB;

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

	// Outputs
	wire [7:0] Y;

	// Instantiate the Unit Under Test (UUT)
	Multiplier uut (
		.A(A),
		.B(B),
		.Y(Y)
	);

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

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

		// Add stimulus here
		#3000 $stop;
	end

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

endmodule
```

---

*Concatenator Behavioral Verilog:*

```
`timescale 1ns / 1ps

module Concatenator(
   input [3:0] A,
   input [3:0] B,
   output [7:0] AB
   );

	assign AB = {A, B};

endmodule
```

---

*Concatenator Test Bench:*

```
`timescale 1ns / 1ps

module Concatenator_TB;

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

	// Outputs
	wire [7:0] AB;

	// Instantiate the Unit Under Test (UUT)
	Concatenator uut (
		.A(A),
		.B(B),
		.AB(AB)
	);

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

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

		// Add stimulus here
		#3000 $stop;
	end

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

endmodule
```

*Shifter Behavioral Verilog:*

```
`timescale 1ns / 1ps

module Shifter(
   input [3:0] A,
   input [3:0] B,
   output [7:0] Y
   );
	 wire [7:0] y_temp;

	assign y_temp = {4'b0000, A};
	assign Y = y_temp << B;

endmodule
```

---

*Shifter Test Bench:*

```
`timescale 1ns / 1ps

module Shifter_TB;

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

	// Outputs
	wire [7:0] Y;

	// Instantiate the Unit Under Test (UUT)
	Shifter uut (
		.A(A),
		.B(B),
		.Y(Y)
	);

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

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

Alexander Salmi
U04201274

```
		// Add stimulus here
		#3000 $stop;
	end

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

endmodule
```

---

*Multiplexer Behavioral Verilog:*

```
`timescale 1ns / 1ps

module Multiplexer(
    input [7:0] A,
    input [7:0] B,
    input [7:0] C,
    input [7:0] D,
    input [1:0] S,
    output reg [7:0] Y
    );

	 always@(*)begin
		 case(S)
			2'b00: Y = A;
			2'b01: Y = B;
			2'b10: Y = C;
			2'b11: Y = D;
		endcase
	end

endmodule
```

---

*Multiplexer Test Bench:*

```
`timescale 1ns / 1ps

module Multiplexer_TB;

	// Inputs
	reg [7:0] A;
	reg [7:0] B;
	reg [7:0] C;
	reg [7:0] D;
	reg [1:0] S;

	// Outputs
	wire [7:0] Y;

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

	initial begin
		// Initialize Inputs
		A = 0;
		B = 0;
		C = 0;
		D = 0;
		S = 0;

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

		// Add stimulus here
		A = 4'b0101;
		B = 4'b1010;
		C = 4'b0011;
		D = 4'b1100;

		S = 2'b00;
```

```
                #100 S = 2'b01;
                #100 S = 2'b10;
                #100 S = 2'b11;

        end

endmodule
```

---

*alu_module Structural Verilog:*

```
`timescale 1ns / 1ps

module alu_module(
   input [3:0] A,
   input [3:0] B,
   input [1:0] S,
   output [7:0] Y
   );

        wire [7:0] wAdd, wMult, wCon, wShift;

        Adder g1(.Y(wAdd), .A(A), .B(B));
        Multiplier g2(.Y(wMult), .A(A), .B(B));
        Concatenator g3(.AB(wCon), .A(A), .B(B));
        Shifter g4(.Y(wShift), .A(A), .B(B));

        Multiplexer g5(.Y(Y), .S(S), .A(wCon), .B(wAdd), .C(wShift), .D(wMult));

endmodule
```

---

*alu_module Test Bench:*

```
`timescale 1ns / 1ps


module alu_module_TB;

	// Inputs
	reg [3:0] A;
	reg [3:0] B;
	reg [1:0] S;

	// Outputs
	wire [7:0] Y;

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

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

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

		// Add stimulus here
		A = 4'b1010;
		B = 4'b0101;

		S = 2'b00;
		#50 S = 2'b01;
		#50 S = 2'b10;
		#50 S = 2'b11;

		#50
		A = 4'b1011;
		B = 4'b0011;
		S = 2'b00;
		#50 S = 2'b01;
		#50 S = 2'b10;
		#50 S = 2'b11;

		#50
		A = 4'b1001;
		B = 4'b0110;

		S = 2'b00;
		#50 S = 2'b01;
		#50 S = 2'b10;
		#50 S = 2'b11;
	end

endmodule
```