ALU Design Report

CIE 239

By: Marwan Bassem, Mohammed Mahmoud Ibrahim, Youssef Allam

Presented to: Dr. Mohamed Samir Eid

23/12/2023

Objective

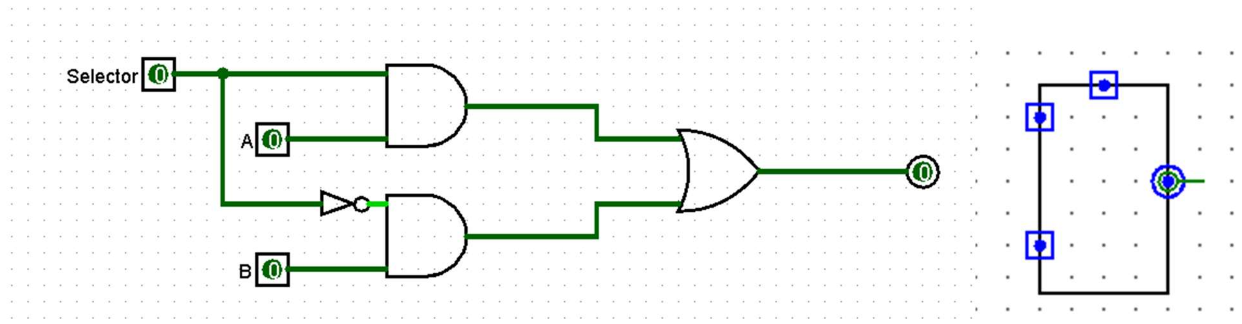
The idea behind this project is to develop an Arithmetic and Logic Unit (ALU). An ALU is a unit that does a set of logical and arithmetic operations. The operation being done by the alu is controlled by a set of selection lines that decide which operation is to be done. The design of our ALU first started by designing the basic combinational circuit elements that will be needed such as multiplexer and adders which will be discussed in detail later in this report. Once these components were selected they were then designed and simulated on Logisim. Using the concept of modularity each component was designed and then used in other components to simplify the result. Once the final design was simulated and tested on Logisim, we began implementing the circuit in HDL code using system Verilog on ModelSim. On model sim we first designed modules for each of the basic gates we will need. After that we designed modules for all the combinational circuits we needed using the concept of structural coding which depends on implementing our entire circuit using only logic gates instead of conditional statements. Finally, the ALU was constructed in a module in ModelSim and tested using a testbench.

Phase 1 (Designing Basic Combinational Elements on Logisim)

The basic logic gates from the built in Logisim library are *and*, *or*, *xor* and *inverters.* All remaining combinational logic was designed using these gates. In this section of the report, we will focus on showing the design for each of these components.

2:1 Multiplexer

A multiplexer is a combinational circuit that takes three inputs. One of which is denoted as a selector and controls which of the other 2 is transferred by the multiplexer.

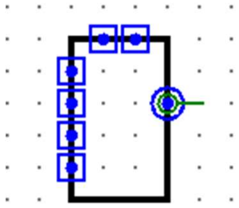


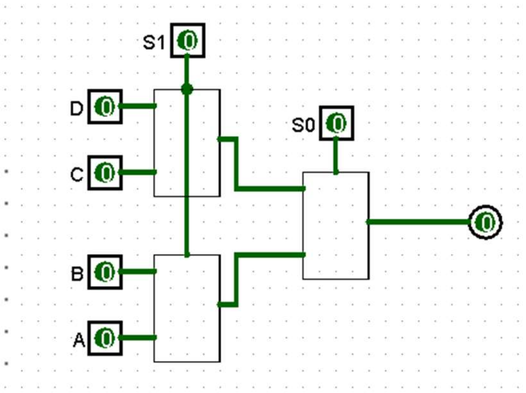
In the above design for the multiplexer the value of A is output when S=1 and the Value of B is output when S=0.

4:1 Multiplexer

A 4:1 multiplexer is the same concept as a 2:1 multiplexer however it has two bits of selection inputs allowing for 4 inputs to be selected from. In this design the values are as follows

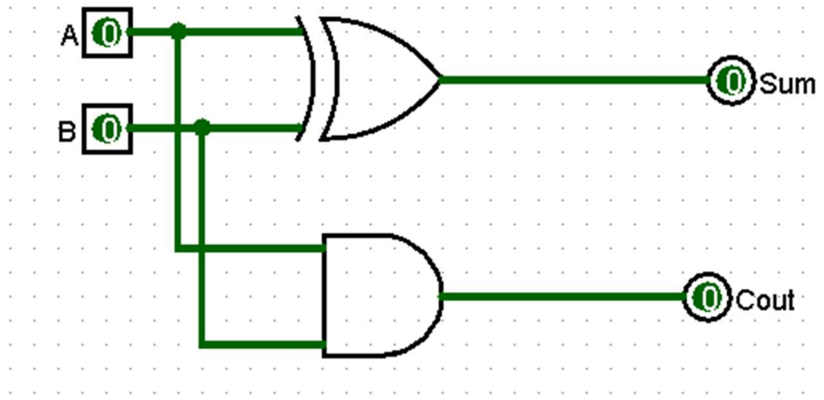
Selector	Output
00	A
10	B
01	C
11	D

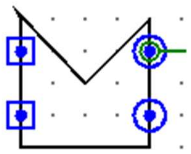




Half Adder

A half adder is a combinational circuit that takes 2 inputs of 1 bit and adds them together giving us the same and the remainder (to be carried to the next MSB bit). This function can be conducted by an XOR gate and an AND gate.





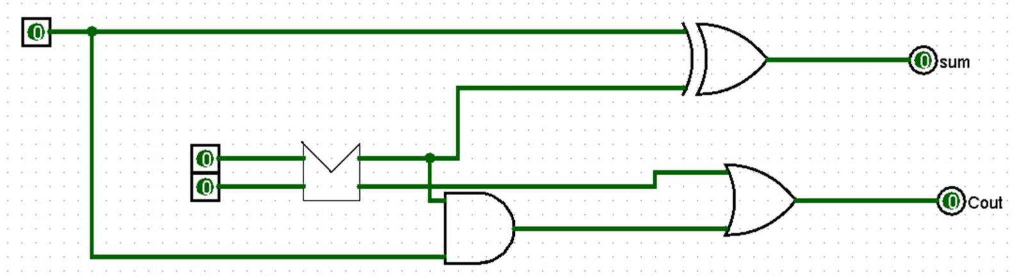
The outputs for this combinational circuit are as follows.

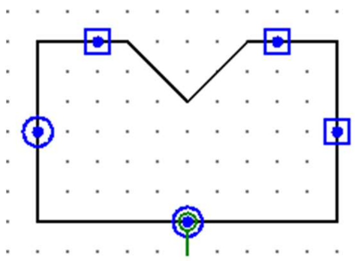
These outputs make sense because when you add 1 to 1 in binary that gives 0 in the current bit and 1 in the next bit to be carried (similar to long addition).

A	B	Sum	Cout
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Full Adder

The half adder design, however, has a minor issue that it has no place for the input from the carry out to be carried into. For that we developed the full adder using a half adder and some logic gates.





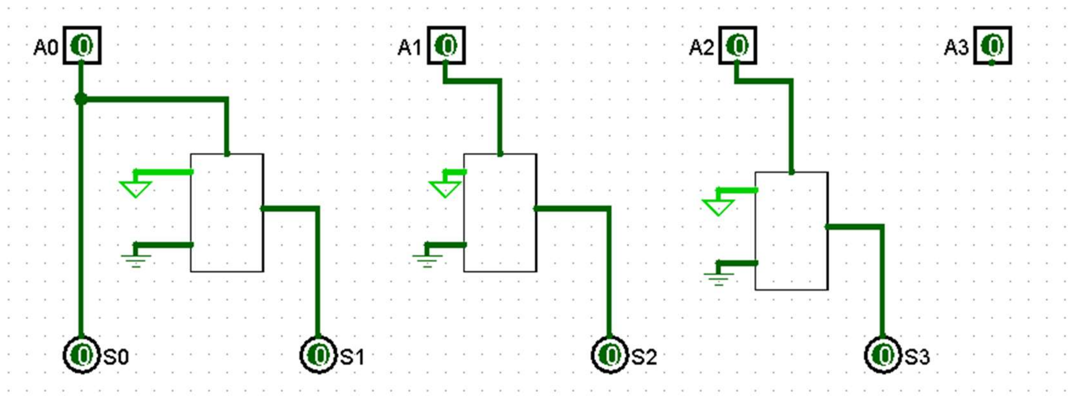
The outputs for the full adder are as follows:

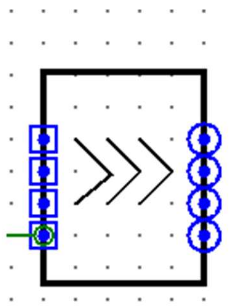
Tracing the truth table shows that the sum is equal to 1 when there are one or three true bits and Cout is equal to 1 when there are two or more true bits.

A	B	Cin	Sum	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Arithmetic Right Shifter

Shifting is the process of moving bits. There are three main types of shifting (arithmetic, logical, and rotational). It also happens in either direction. In this ALU we implemented arithmetic right shifting. Which means moving bits to the right and replacing the moved bits with the most significant bit.





As is shown by this design the shifter moves all bits to the next bit. The first bit stays the same as it was before shifting. The last bit is discarded since it was replaced by the bit before it.

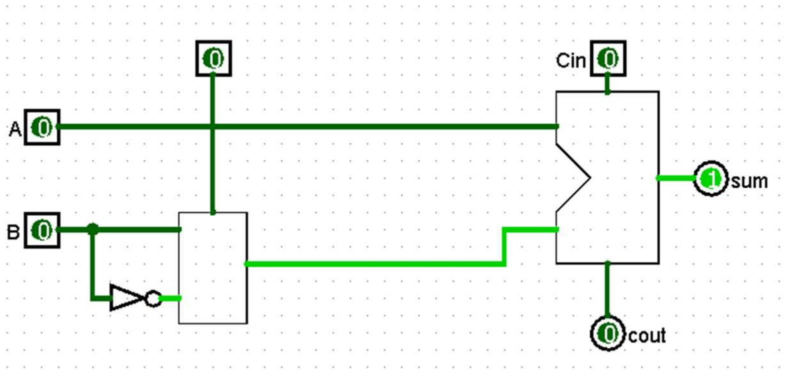
Phase 2 (Designing the ALU on Logisim)

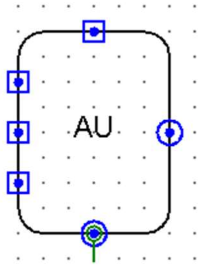
Before designing the ALU we decided it was wiser to first implement the Arithmetic Unit individually and the logic Unit individually. Each first for one bit then each for four bits.

Arithmetic Unit (1 bit)

The operations we were focused on in the case of 2 inputs A and B were A+B, A+B’, A+B, A+B+1.

The design for the one Bit arithmetic Unit was designed to be controlled by 1 Input Selection to a multiplexer and a CIN input to a full adder. As shown by the circuit the following operations are done in the following conditions.

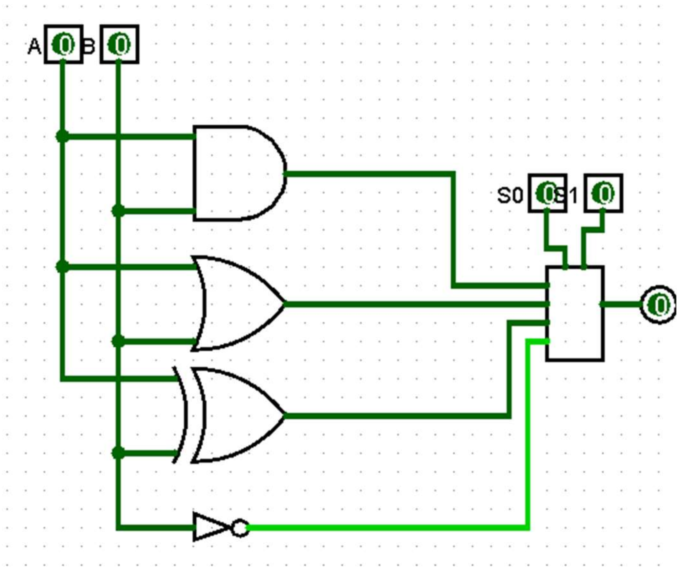


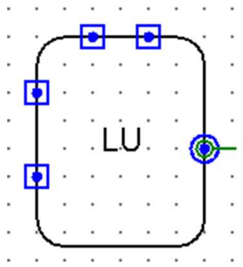


Logic Unit (1 Bit)

The design for the logic unit was very similar to the arithmetic unit however less complicated due to not having to deal with carrying in and out. The outputs can be summarized as follows.

S0	S1	Operation
0	0	AB
0	1	A or B
1	0	A xor B
1	1	B’

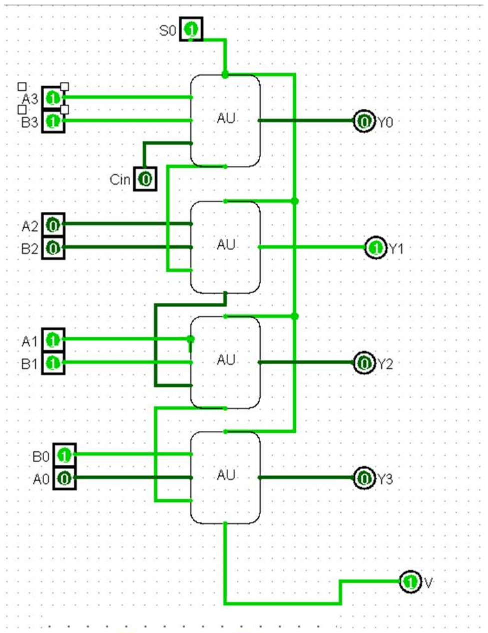




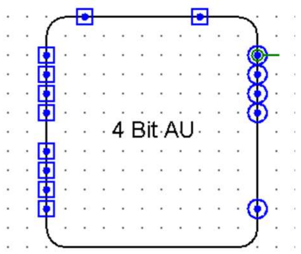
4 Bit Arithmetic Unit

Once the design for the 1-bit Arithmetic Unit was completed all what was left was to generalize for four bits. This circuit is four of the 1 Bit arithmetic unit all with the same selection line. However, Cin is given to the first AU and then the Cin for each consequent AU is the cout of the previous AU.

The Only new output in this circuit is the V output which signals one if the carry out of the last AU is not 0 then that means that an overflow occurred indicating that 4 bits wasn't enough to perform the requested operation on the specified values.

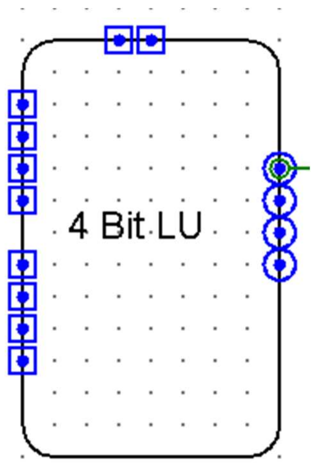


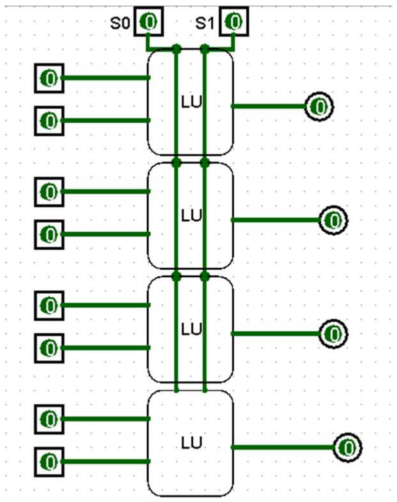
S0	CIN	Operation
0	0	A+B'
0	1	A+B'+1 (A-B)
1	0	A+B
1	1	A+B+1



4 Bit Logic Unit

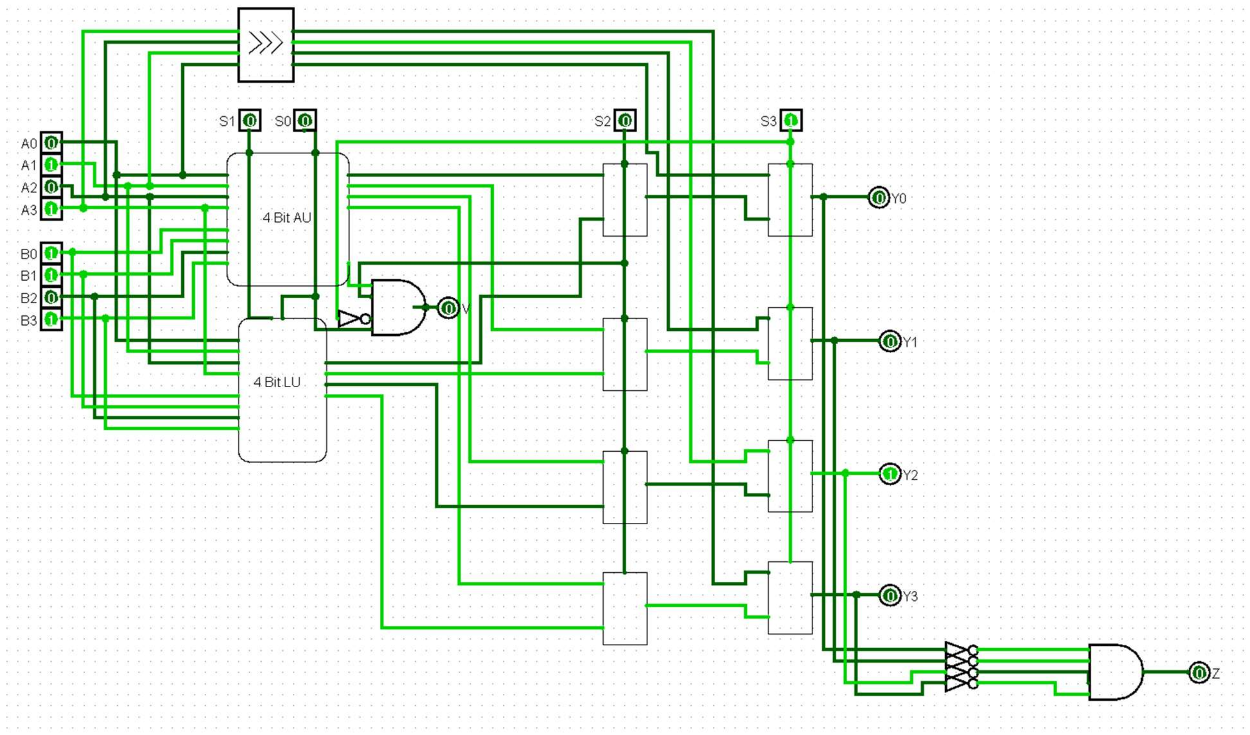
Same logic as the 4 Bit Arithmetic unit was applied. However, this time the 2 selection lines were passed to all of the logic units.





General 4 Bit ALU

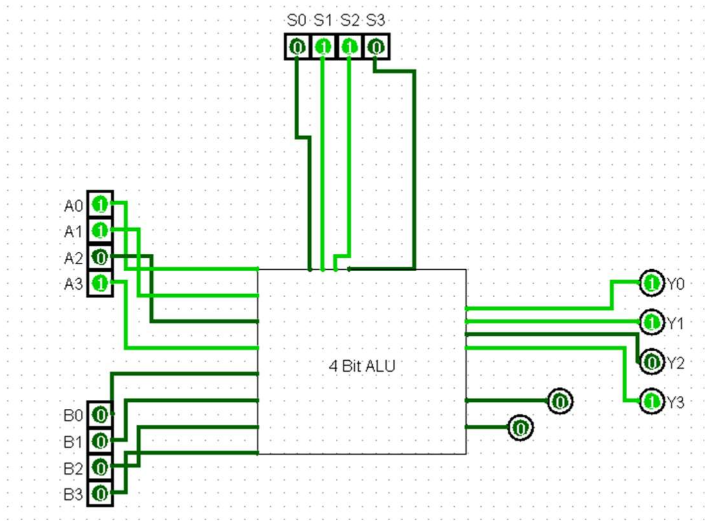
The previous components were prepared to build the final version of the ALU.



There are 2 4-Bit inputs. Two selection lines at the first stage of the circuit (The Cin input has been called S1 and is now considered a selection line for the sake of simplicity). This means that both the AU and LU perform their respective operations, however the two outputs are passed onto a 2:1 Multiplexer with a third selection line. That selection line decides which of the outputs (Arithmetic or logic) should be transmitted. However, none of these operations include shifting yet so a shifter was added outside both units and the values of A were passed onto it. A second set of multiplexers with a fourth selection line was added to allow the user to choose which value to transmit (shifting or the output from previous multiplexer). All four final outputs were inverted and passed through a four bit AND gate giving output Z which is one when all the values of y are zero. Finally, and 4-Bit AND gate was added after the V output of the 4 Bit AU and was given the value of V, S0, S2, and S3'. This ensured that the V did not give any positive output except if the circuits were in the arithmetic display mode since it is not possible for overflow to occur in the case of logic operations. The operations being done by the circuit are summarized in the following table.

S0	S1	S2	S3	Operation
0	0	0	0	AB
0	0	1	0	A+B'
0	1	0	0	A xor B
0	1	1	0	A - B

1	0	0	0	A or B
1	0	1	0	A+B
1	1	0	0	B’
1	1	1	0	A+B+1
X	X	X	1	ASR (A)



Phase 3(Logisim Testing)

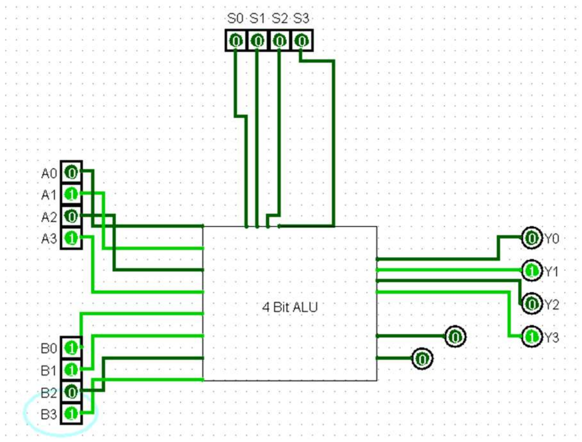
Before beginning implementation on system Verilog, we tested our simulated circuit on Logisim.

We used sample Numbers A=0101_2 and B=1101_2

AND operation:

Expected Value: 0101_2

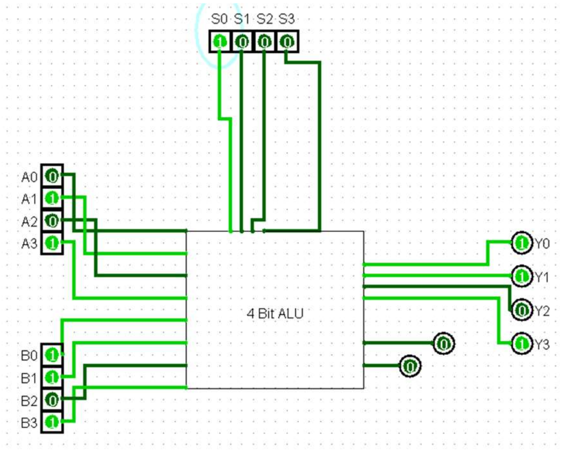
Experimental Value: 0101_2



OR Operation

Expected Value: 1101_2

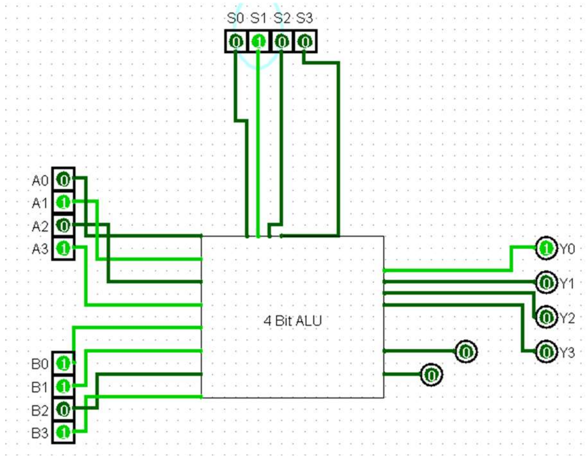
Experimental Value: 1101_2



XOR Operation

Expected Value: 1000_2

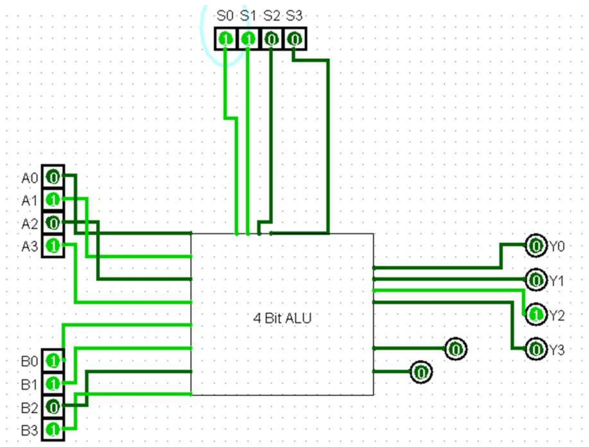
Experimental Value: 1000_2



Inverting Operation

Expected Value: 0010_2

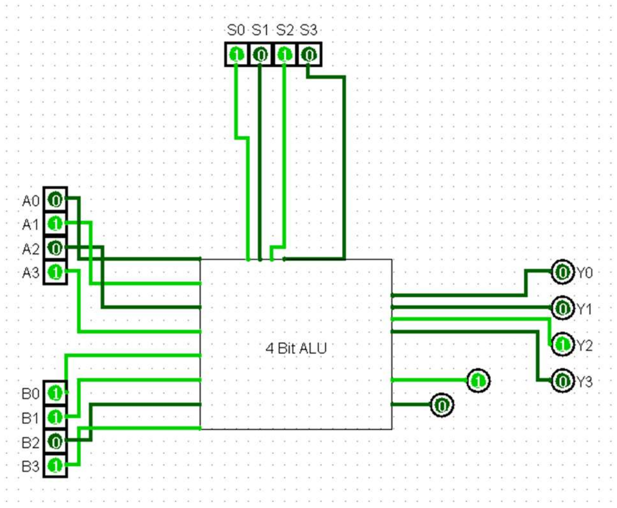
Experimental Value: 0010_2



Addition (A+B)

Expected Value: 0010_2 (with overflow)

Experimental Value: 0010_2 (with overflow)

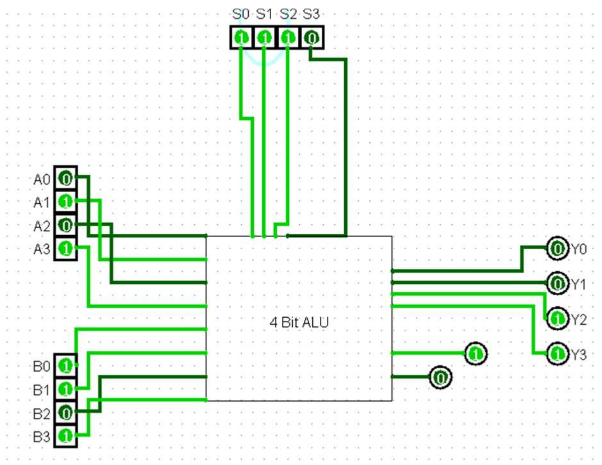


Addition (A+B+1)

Expected Value: 0011_2 (with overflow)

Experimental Value: 0011_2 (with overflow)

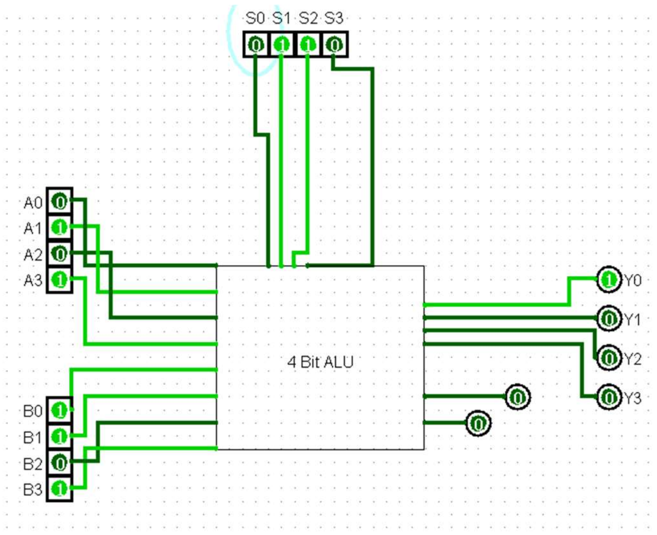
(The increment function is a special case of this function in which A=0000)



Addition (A+B'+1=A-B)

Expected Value:1000_2

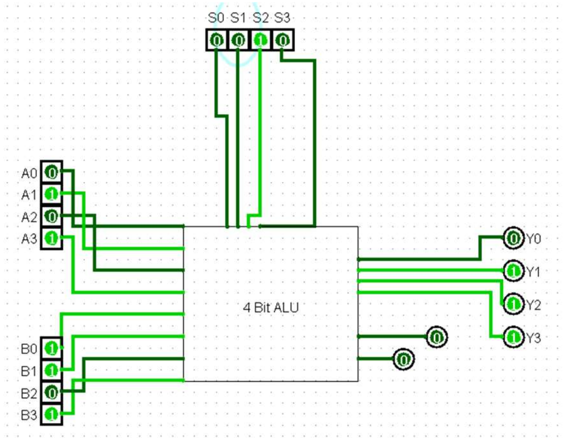
Experimental Value: 1000_2



Addition (A+B'=A-B-1)

Expected Value: 0111_2

Experimental Value: 0111_2

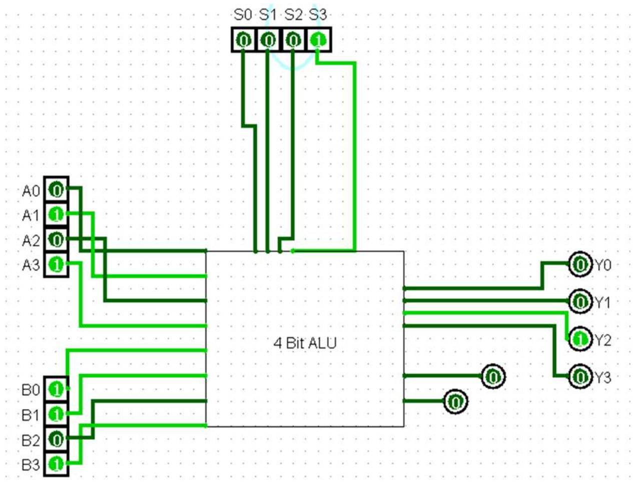


Arithmetic Right Shifting

Expected Value: 0010_2

Experimental Value: 0010_2

Note: ASR is called whenever is S3 equals one regardless of the other selectors



Phase 4 (System Verilog Design)

After the circuits were completed on Logisim it was time code on system Verilog. Our code is attached in a file along with the report however it went as followed. We first designed the basic gates we needed (**AND**, **OR XOR**). Once that was completed we designed the **2:1 Multiplexer** using the AND, OR, and XOR modules we designed. Using the 2:1 Multiplexer module we designed a **4:1 Multiplexer** module. Next we designed a **half adder** module in the same way it was shown in the design above using the AND and XOR gate modules we previously designed. Next we developed a **Full adder** module using the same gates and the half adder module. We also designed the **Right Arithmetic Shifter** using multiple instances of the 2:1 Multiplexer module.

Next a **One-bit Arithmetic Unit** module was designed using the full adder module and a 2:1 Multiplexer. Once that was completed we designed a **One-bit Logic Unit** using logic gates and a 4:1 multiplexer. Next we generalized for 4 bits by creating a **4-bit Arithmetic Unit** and a **4 Bit Logic Unit** by calling 4 instances of the One-bit Arithmetic Unit and the One-bit Logic Unit respectively. In the addition the Cin of each adder was the cout of the previous one until the last Adder passed its cout to V to detect if there is overflow.

Finally, the **4-Bit ALU** was designed by calling one instance of the 4-Bit Arithmetic Unit and one instance of the 4-Bit Logic Unit as well as one instance of the shifter module. Then a series of multiplexer instances was created (two for each output bit) first to select between the arithmetic and logic unit then select between that output and the shifter output.

<u>**Codes:**</u>

The coding segment was putting our previous design in modules and calling instances of these modules in other modules as shown below.

Two-Bit AND gate:

```
module two_and(input a, input b, output y);

assign y=a&b;

endmodule
```

Two-Bit OR gate:

```
module two_or(input a, input b, output y);

assign y=a|b;

endmodule
```

Three-Bit AND gate:

```
module three_and(input a, input b, input c, output y);

assign y=a&b&c;

endmodule
```

Four-Bit AND gate

```
module four_and(input a, input b, input c,input  d, output y);

assign y=a&b&c&d;

endmodule
```

Two-Bit XOR gate

```
module two_xor(input a, input b, output y);

assign y=a^b;

endmodule
```

Three-Bit XOR gate

```
module three_xor(input a, input b, input c, output y);
```

```
assign y=a^b^c;

endmodule
```

2:1 Multiplexer

```
module two_mux(input a, input b, input select, output y);

logic f;

logic s;

two_and first(select, a,f);

two_and second(~select, b,s);

two_or fin(f,s,y);

endmodule
```

4:1 Multiplexer

```
module four_mux(input a,input b, input c, input d, input s0,input s1, output y);

logic f;

logic s;

two_mux first(d,c,s1,f);

two_mux second(b,a,s1,s);

two_mux fin(f,s,s0,y);

endmodule
```

Half Adder

```
module half_adder(input a, input b, output y, output cout);

two_xor sum(a,b,y);

two_and carry(a,b,cout);

endmodule
```

Full Adder

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

logic halfsum;

logic halfcout;

logic extracout;

half_adder half(a,b,halfsum,halfcout);
```

```
two_xor finsum(halfsum,cin,sum);

two_and prefin(cin,halfsum,extracout);

two_or fincout(extracout,halfcout,cout);

endmodule
```

Arithmetic Right Shifter

```
module asr(input a0, input a1, input a2, input a3, output y0, output y1, output y2, output y3);

assign y0=a0;

two_mux first(1,0,a0,y1);

two_mux second(1,0,a1,y2);

two_mux third(1,0,a2,y3);

endmodule
```

One Bit Arithmetic Unit

```
module one_au(input a, input b, input s0, input s1,output sum, output cout);

logic two;

two_mux first(b,~b,s0,two);

full_adder f(a,two,s1,sum,cout);

endmodule
```

One Bit Logic Unit

```
module one_lu(input a, input b, input s0, input s1, output y);

logic addr;

logic orr;

logic xorr;

logic invr;

two_and aa(a,b,addr);

two_or bb(a,b,orr);

two_xor cc(a,b,xorr);

assign invr = ~b;

four_mux result(addr,orr,xorr,invr,s0,s1,y);

endmodule
```

4-Bit Arithmetic Unit

```
module four_au(input a0,a1,a2,a3,b0,b1,b2,b3,input s0,input s1,output y0,y1,y2,y3, output v);

logic c1;

logic c2;

logic c3;

one_au aa(a3,b3,s0,s1,y3,c1);

one_au bb(a2,b2,s0,c1,y2,c2);

one_au cc(a1,b1,s0,c2,y1,c3);

one_au dd(a0,b0,s0,c3,y0,v);

endmodule
```

4-Bit Logic Unit

```
module four_lu(input a0,a1,a2,a3, input b0,b1,b2,b3, input s0,s1,output y0,y1,y2,y3);

one_lu aa(a0,b0,s0,s1,y0);

one_lu bb(a1,b1,s0,s1,y1);

one_lu cc(a2,b2,s0,s1,y2);

one_lu dd(a3,b3,s0,s1,y3);

endmodule
```

ALU Final Design

```
module alu(input a0,a1,a2,a3, input b0,b1,b2,b3, input s0,s1,s2,s3, output y0,y1,y2,y3, output z,v);

logic l0,l1,l2,l3;

logic u0,u1,u2,u3;

logic v_temp;

logic al0,al1,al2,al3;

logic sh0,sh1,sh2,sh3;

asr gg(a0,a1,a2,a3,sh0,sh1,sh2,sh3);

four_au aa(a0,a1,a2,a3,b0,b1,b2,b3,s0,s1,u0,u1,u2,u3,v_temp);

four_lu bb(a0,a1,a2,a3,b0,b1,b2,b3,s0,s1,l0,l1,l2,l3);

two_mux cc(u0,l0,s2,al0);

two_mux dd(u1,l1,s2,al1);
two_mux ee(u2,l2,s2,al2);

two_mux ff(u3,l3,s2,al3);

two_mux fin1(sh0,al0,s3,y0);

two_mux fin2(sh1,al1,s3,y1);

two_mux fin3(sh2,al2,s3,y2);

two_mux fin4(sh3,al3,s3,y3);

four_and fin5(~y0,~y1,~y2,~y3,z);

four_and fin6(s0,v_temp,s2,~s3,v);

endmodule
```

Phase 5 (testing)

Finally, it was time to test the code using ModelSim wave forms. To ease the testing process a testbench was created. The code for the test bench can be found below.

TestBench

```
module alu_DUT();

logic a0=0; logic a1=1; logic a2=0; logic a3=1;

logic b0=1; logic b1=1;logic b2=0;logic b3=1;

logic s0; logic s1; logic s2;logic s3;

reg y0,y1,y2,y3; reg z; reg v;

initial

begin

s0=0;s1=0;s2=0;s3=0;#100; ///AND //Check

s0=1;s1=0;s2=0;s3=0;#100; //XOR // Check

s0=0;s1=1;s2=0;s3=0;#100; //OR  // Check

s0=1;s1=1;s2=0;s3=0;#100; //INV // Check

s0=1;s1=0;s2=1;s3=0;#100; //A+B //Check

s0=1;s1=1;s2=1;s3=0;#100; //A+B+1 //Check

s0=0;s1=0;s2=1;s3=0;#100; //A+B' (A-B-1) //Check

s0=0;s1=1;s2=1;s3=0;#100; //A+B'+1 (A-B) //Check
```

```
s0=0;s1=0;s2=0;s3=1#100; //ASR //Check

a0=0;a1=0;a2=0;a3=0;s0=1;s1=1;s2=1;s3=0;#100; //B+1 // Check

end

alu f(a0,a1,a2,a3,b0,b1,b2,b3,s0,s1,s2,s3,y0,y1,y2,y3,z,v);

endmodule
```

This piece of code was given four-bit values for A:0101 and B:1101 and then the values of the 4 selections were cycled through and an instance of the ALU was called to perform the respective operation.

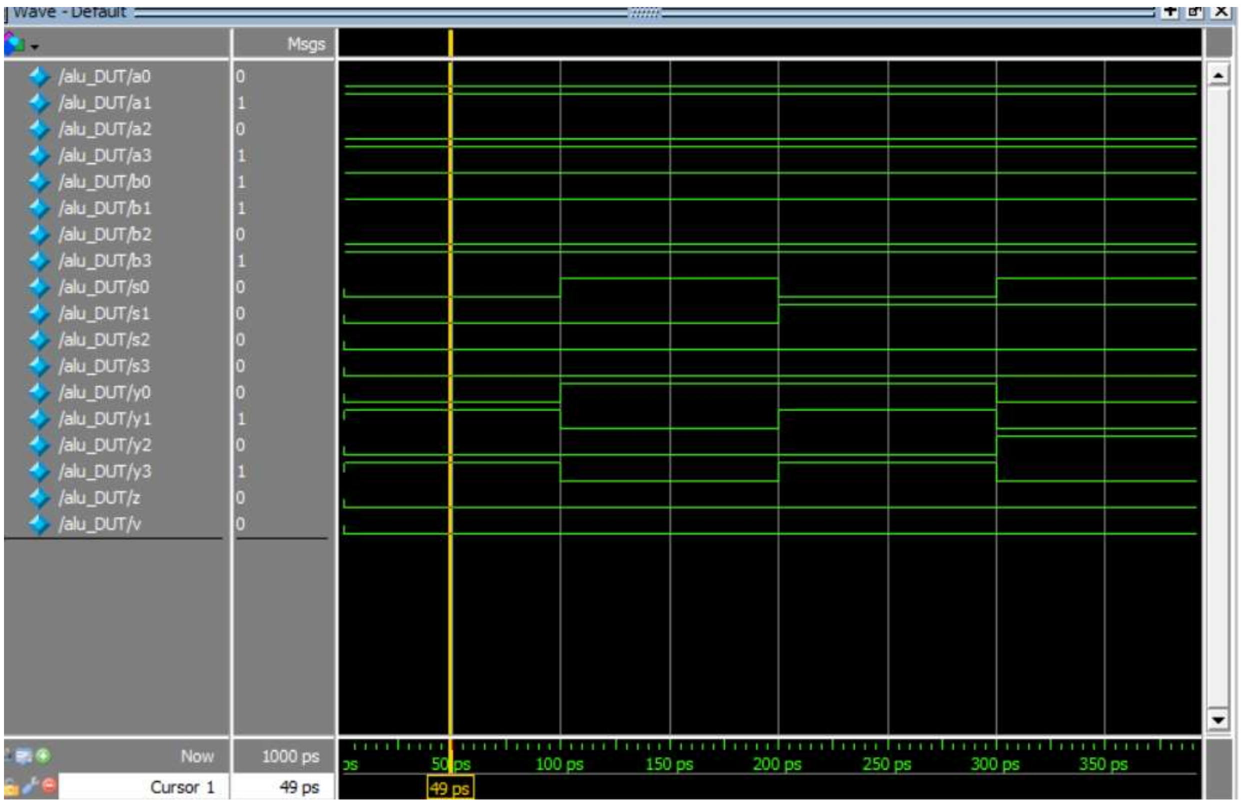
Waveform

According to the values given in the testbench code we should expect the following operations to be performed with the following order when simulating the testbench.

Output #	Operation	Value
1	AND	0101
2	XOR	1000
3	OR	1101
4	Invert	0010
5	A+B	0010 (V=1)
6	A+B+1	0011 (V=1)
7	A+B’	0111
8	A+B’+1	1000
9	ASR	0010
10	B+1	1110

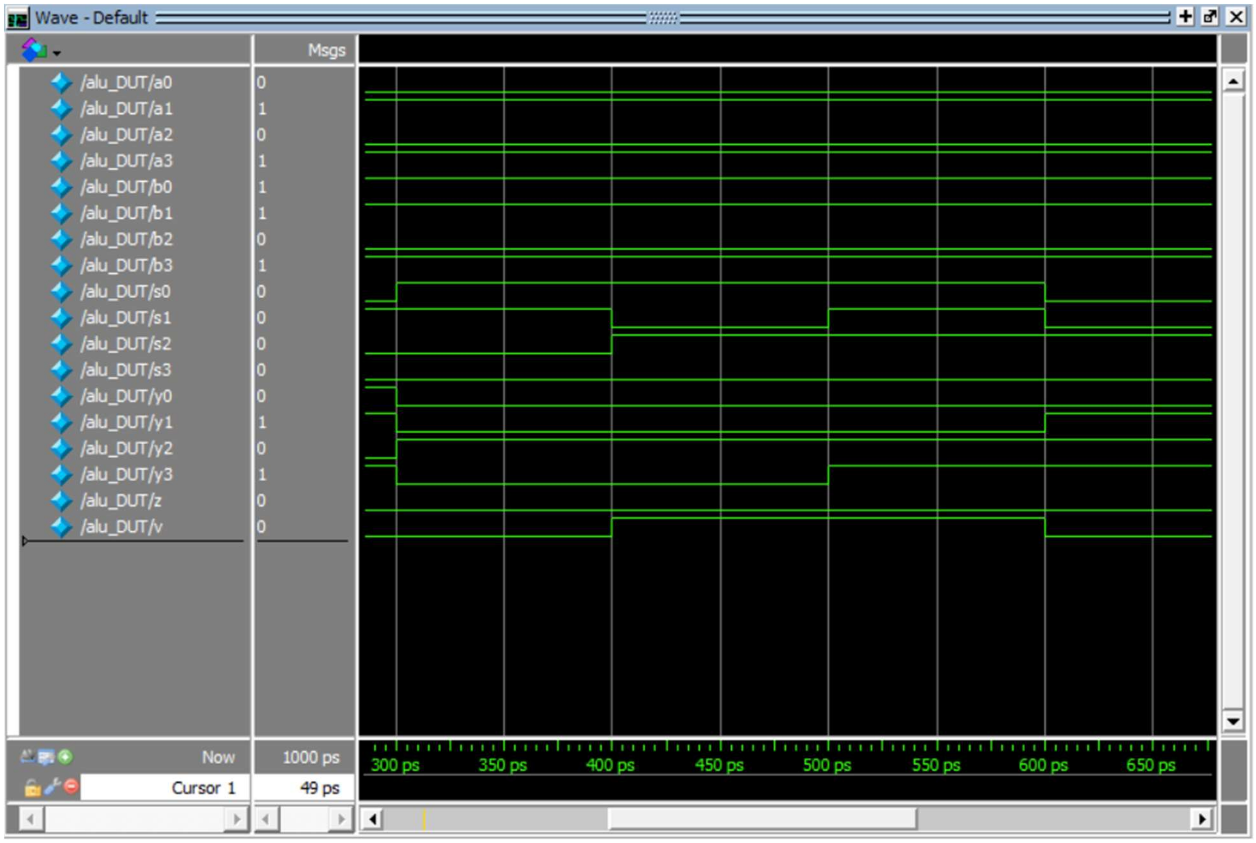
Note: The values in the above table are the same as the values in phase Logisim simulations and the same as the values achieved by hand calculations.

The waveforms below show the results of the system Verilog wave form simulation.



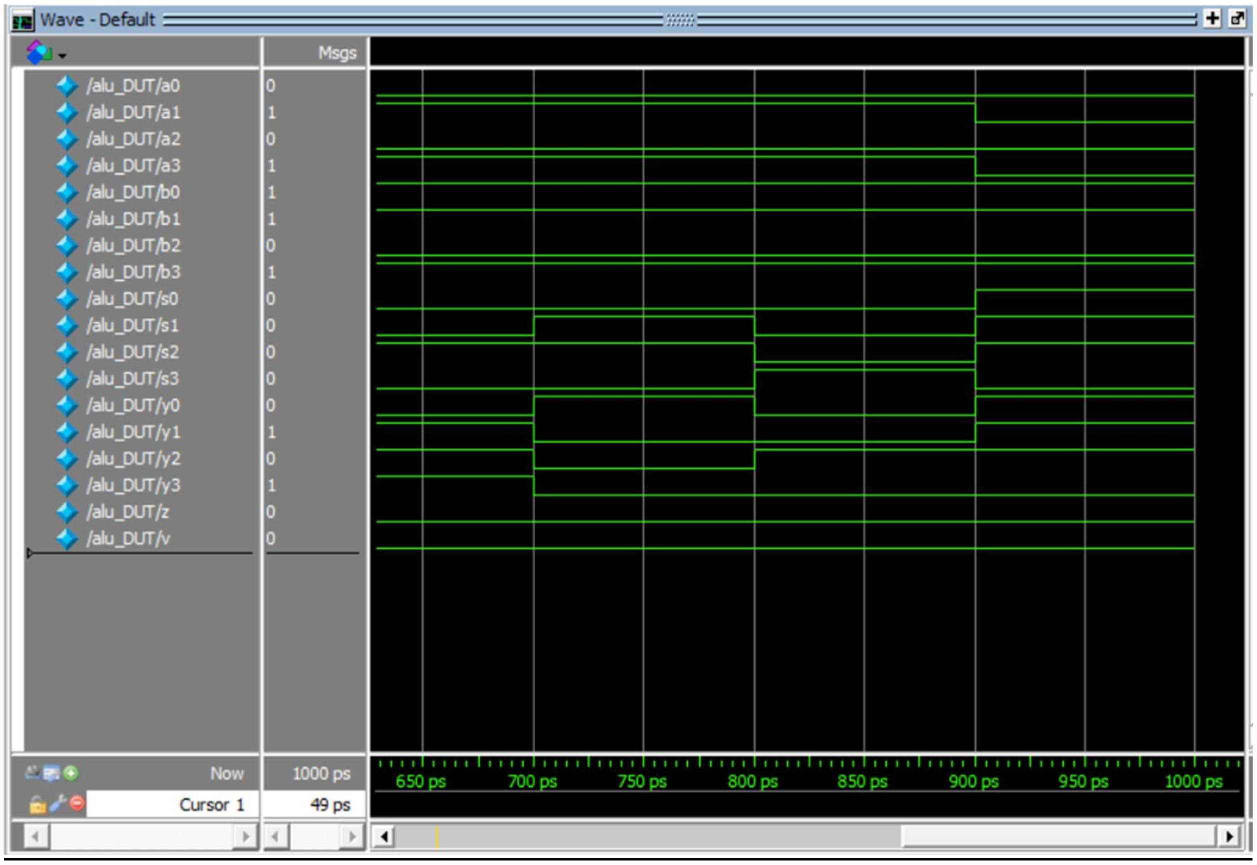
```
/alu_DUT/a0   0
/alu_DUT/a1   1
/alu_DUT/a2   0
/alu_DUT/a3   1
/alu_DUT/b0   1
/alu_DUT/b1   1
/alu_DUT/b2   0
/alu_DUT/b3   1
/alu_DUT/s0   0
/alu_DUT/s1   0
/alu_DUT/s2   0
/alu_DUT/s3   0
/alu_DUT/y0   0
/alu_DUT/y1   1
/alu_DUT/y2   0
/alu_DUT/y3   1
/alu_DUT/z    0
/alu_DUT/v    0

Now        1000 ps
Cursor 1     49 ps
```



```
/alu_DUT/a0   0
/alu_DUT/a1   1
/alu_DUT/a2   0
/alu_DUT/a3   1
/alu_DUT/b0   1
/alu_DUT/b1   1
/alu_DUT/b2   0
/alu_DUT/b3   1
/alu_DUT/s0   0
/alu_DUT/s1   0
/alu_DUT/s2   0
/alu_DUT/s3   0
/alu_DUT/y0   0
/alu_DUT/y1   1
/alu_DUT/y2   0
/alu_DUT/y3   1
/alu_DUT/z    0
/alu_DUT/v    0

Now        1000 ps
Cursor 1     49 ps
```



	Msgs
/alu_DUT/a0	0
/alu_DUT/a1	1
/alu_DUT/a2	0
/alu_DUT/a3	1
/alu_DUT/b0	1
/alu_DUT/b1	1
/alu_DUT/b2	0
/alu_DUT/b3	1
/alu_DUT/s0	0
/alu_DUT/s1	0
/alu_DUT/s2	0
/alu_DUT/s3	0
/alu_DUT/y0	0
/alu_DUT/y1	1
/alu_DUT/y2	0
/alu_DUT/y3	1
/alu_DUT/z	0
/alu_DUT/v	0
Now	1000 ps
Cursor 1	49 ps

Close inspection of the waveform shows that the outputs of the waveform values y0, y1, y2, y3 were exactly as we expected in the above table. Additionally, the system Verilog codes will be uploaded along with the report for further testing if needed.