**ALU Design Report**

**CIE 239**

By: Marwan Bassem, Mohammed Mahmoud Ibrahim, Youssef Allam

Presented to: Dr. Mohamed Samir Eid

23/12/2023

**Objective**

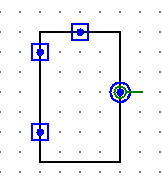
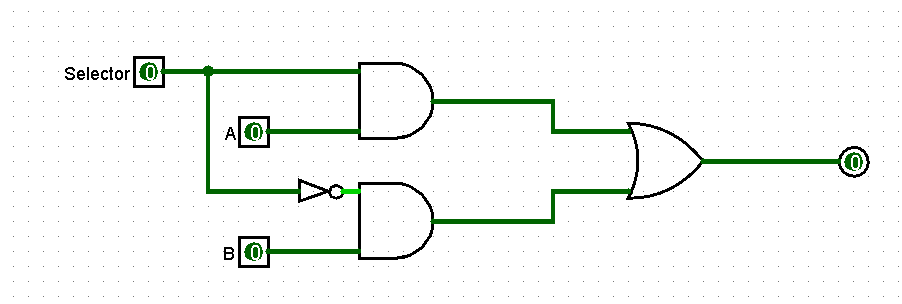
The idea behind this project is to develop an Arithmetic and Logic Unit (ALU). An ALU is a unit that does a set of logical and arithmetic operations. The operation being done by the alu is controlled by a set of selection lines that decide which operation is to be done. The design of our ALU first started 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)**

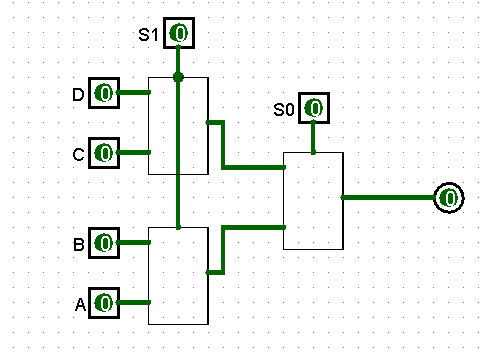
The basic logic gates from the built in Logisim library are *and*, *or*, *xor* and *inverters.* All remaining combinational logic was designed using these gates. In this section of the report, we will focus on showing the design for each of these components.

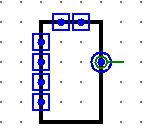
**2:1 Multiplexer**

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**

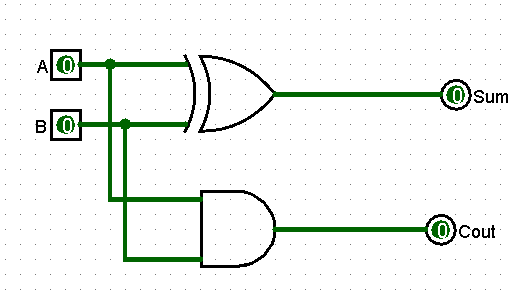
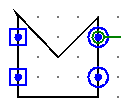
A 4:1 multiplexer is the same concept as a 2:1 multiplexer however it has two bits of selection inputs allowing for 4 inputs to be selected from. In this design the values are as follows

|  |  |
| --- | --- |
| 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.

|  |  |  |  |
| --- | --- | --- | --- |
| A | B | Sum | Cout |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

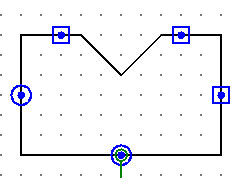
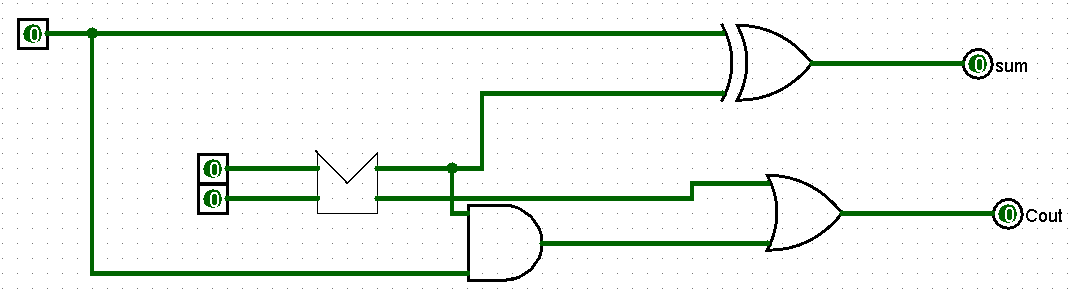


The outputs for this combinational circuit are as follows.

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).

**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.



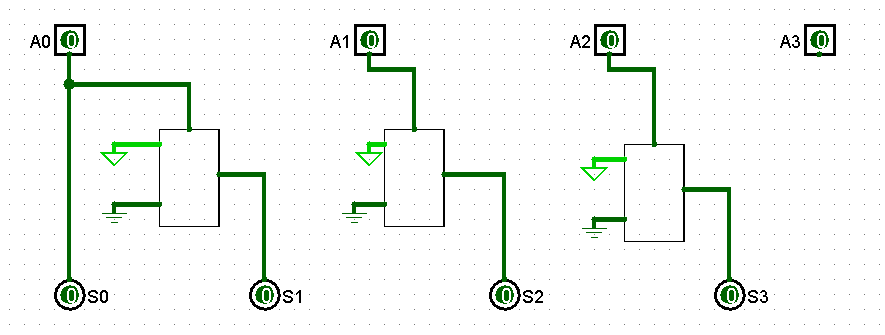
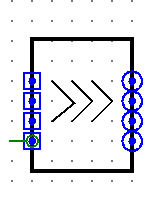
The outputs for the full adder are as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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 |

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.

**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.

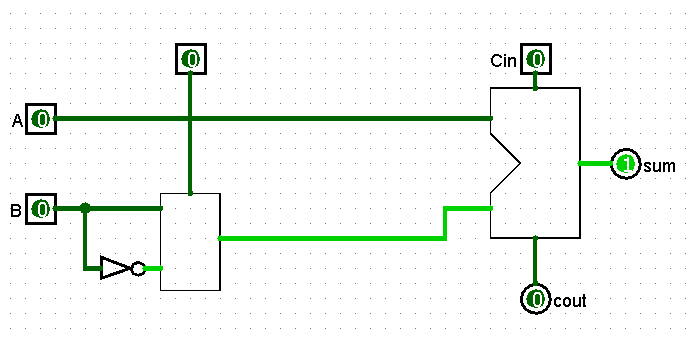


As is shown by this design the shifter moves all bits to the next bit. The first bit stays the same as it was before shifting. The last bit is discarded since it was replaced by the bit before it.

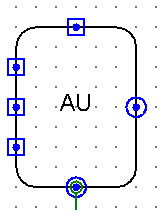
**Phase 2 (Designing the ALU on Logisim)**

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.

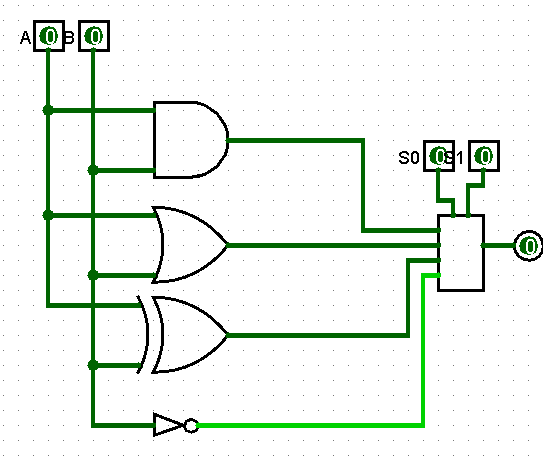
The design for the one Bit arithmetic Unit was designed to be controlled by 1 Input Selection to a multiplexer and a CIN input to a full adder. As shown by the circuit the following operations are done in the following conditions.

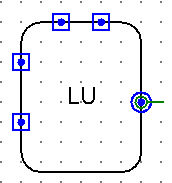


**Logic Unit (1 Bit)**

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**

A screenshot of a computer

Description automatically generatedOnce 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.

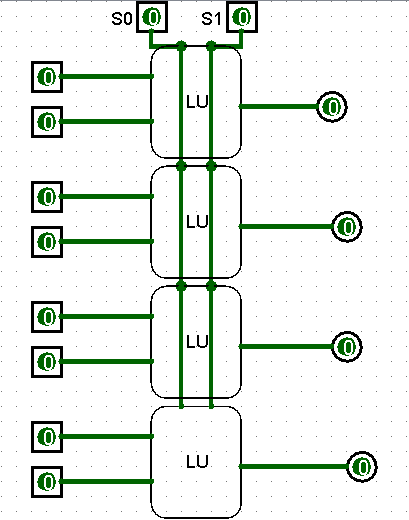
The Only new output in this circuit is the V output which signals one if the carry out of the last AU is not 0 then that means that an overflow occurred indicating that 4 bits wasn’t enough to perform the requested operation on the specified values.

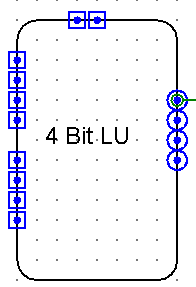
|  |  |  |
| --- | --- | --- |
| S0 | CIN | Operation |
| 0 | 0 | A+B’ |
| 0 | 1 | A+B’+1 (A-B) |
| 1 | 0 | A+B |
| 1 | 1 | A+B+1 |

A grid with a square and a line with dots and a green arrow

Description automatically generated with medium confidence

**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**

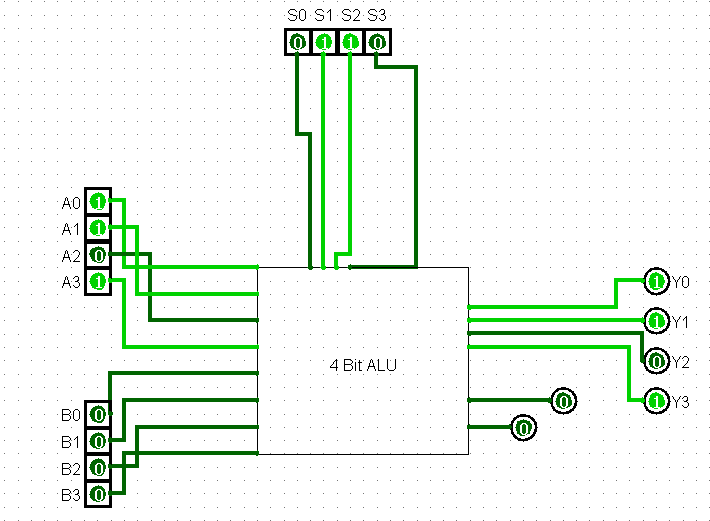
The previous components were prepared to build the final version of the ALU.

A computer screen shot of a diagram

Description automatically generated

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=01012 and B=1101­­2

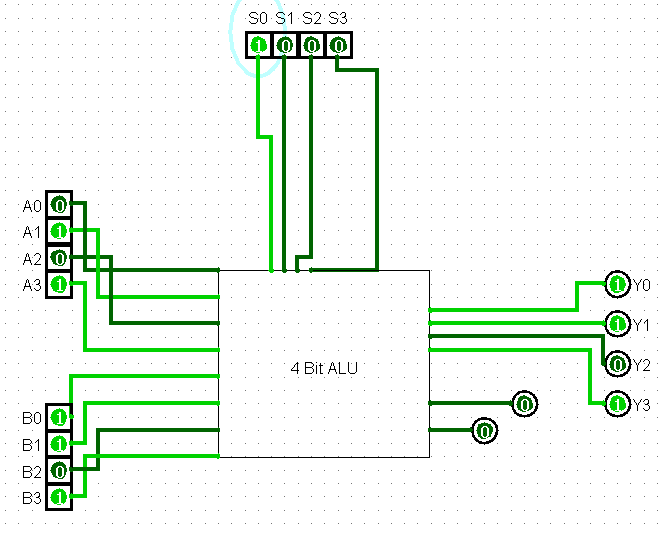
**AND operation:**

A computer circuit diagram with green lines

Description automatically generated with medium confidenceExpected Value: 01012

Experimental Value: 01012

**OR Operation**

Expected Value: 11012

Experimental Value: 11012

**XOR Operation**

Expected Value: 10002

A computer diagram of a circuit board

Description automatically generatedExperimental Value: 1000­­2

**Inverting Operation**

A computer circuit board with green lines

Description automatically generatedExpected Value: 00102

Experimental Value: 00102

**Addition (A+B)**

A computer diagram of a circuit board

Description automatically generatedExpected Value: 00102 (with overflow)

Experimental Value: 00102 (with overflow)

**Addition (A+B+1)**

A computer screen shot of a computer

Description automatically generatedExpected Value: 00112 (with overflow)

Experimental Value: 00112 (with overflow)

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

**Addition (A+B’+1=A-B)**

Expected Value:10002

A computer circuit board with many green lines

Description automatically generatedExperimental Value: 10002

**Addition (A+B’=A-B-1)**

A computer circuit diagram with green lines

Description automatically generated with medium confidenceExpected Value: 01112

Experimental Value: 01112

**Arithmetic Right Shifting**

A computer screen shot of a circuit board

Description automatically generatedExpected Value: 00102

Experimental Value: 00102

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.

**Codes:**

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.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**A screenshot of a computer

Description automatically generated**

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.