Lab 1 ALU + Register File

Group1

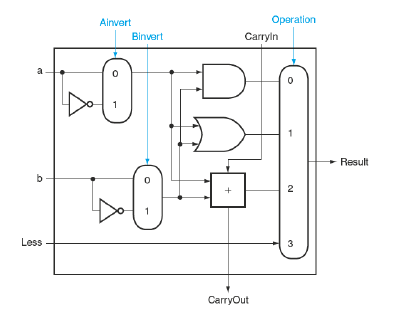
# Introduction

# This lab includes three parts: Part I: One-bit ALU An one-bit ALU will be implemented by AND gate, OR gate, NOT gate, 2-1 MUX, 3-1 MUX, and a Full Adder to perform one of several operations based on the control signals. Part II: 16-bit ALU A 16-bit ALU is implemented to perform all the operations listed in the lab manual. Part III: Register File A register file with 32 16-bit registers, two read ports, and a single write port is built. The register file is able to perform two concurrent read operations and a write operation.

# Descriptions

## Part I: One-bit ALU

In this lab, we need to build the following 1 bit ALU in Xilinx ISE:



This seems to be an easy job. However, there is one constraint: we are only allowed to use structural code but not behavioral code. This means that, for example, we need to use bitwise AND and or to implement if statements.

The implementation of this lab is not too difficult. First, we use combinational logic to generate a&b, a+b, and a|b. Then, we use a multiplexer to select the output based on signal sel.

The following code snippet shows our multiplexer implementation.

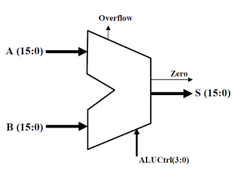
|  |
| --- |
| module mux3(in0, in1, in2,sel, out)**;** input in0, in1,in2**;** input sel**;** wire [1:0] sel**;** output out**;** assign out = (sel[1]**&**~sel[0]**&**in2) **|**(~sel[0]**&**~sel[1]**&**in0)**|**(sel[0]**&**~sel[1]**&**in1)**;** endmodule |

## Part II: 16-Bit ALU

In this section, we are asked to design and build a 16-bit asynchronous ALU using the tools provided by the Xilinx ISE. Our goal is to implement an ALU that can perform all the operations listed in the table below:

|  |
| --- |
| Ctrl Description  0000 Subtraction  0001 Addition  0010 Bitwise OR  0011 Bitwise AND  0100 Decrement  0101 Increment  0110 Invert  1100 Arithmetic Shift Left  1110 Arithmetic Shift Right  1000 Logical Shift Left  1010 Logical Shift Right  1001 Set on Less than or Equal |

The following figure shows the high level diagram of the ALU:



Similarly, in this section, we are only allowed to use structural logic.

The implementation is similar to 1 Bit ALU. First we use combinational logic to generate the results, and use two multiplexers to select the output for S and Overflow.

The following code snippet shows how we call each module (fulladder, bitwiseOR, ...) and multiplex the results.

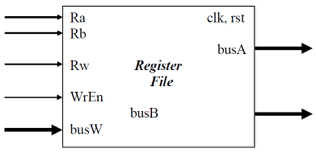
|  |
| --- |
| fulladder ADDITION(**.**r1(a), **.**r2(b), **.**ci(0), **.**co(co\_addition), **.**result(addition), **.**overflow\_adder(overflow\_addition)); **//** need to multiplex overflow fulladder SUBTRACTION(**.**r1(a), **.**r2(**~**b), **.**ci(1), **.**co(co\_subtraction), **.**result(subtraction), overflow\_adder(overflow\_subtraction));  bitwiseOR BITWISEOR(**.**a(a), **.**b(b), **.**result(OR)); bitwiseAND BITWISEAND(**.**a(a), **.**b(b), **.**result(AND)); fulladder DECREMENT(**.**r1(a), **.**r2(**~**1), **.**ci(1), **.**co(co\_dec), **.**result(decrement), **.**overflow\_adder(overflow\_dec)); fulladder INCREMENT(**.**r1(a), **.**r2(1), **.**ci(0), **.**co(co\_inc), **.**result(increment), **.**overflow\_adder(overflow\_inc)); assign invert **=** **~**a; shift\_left ASL(**.**a(a), **.**b(b), **.**overflow(overflow\_asl), **.**result(asl));  shift\_left LSL(**.**a(a), **.**b(b), **.**overflow(overflow\_lsl), **.**result(lsl));  ASR ASR(a, b, asr); LSR LSR(a, b, lsr); slte SLTE(a, b, slt); fullMux16to1 RESULTMUX(**.**i0(subtraction), **.**i1(addition), **.**i2(OR), **.**i3(AND), **.**i4(decrement), **.**i5(increment),  **.**i6(invert), **.**i7(0), **.**i11(0), **.**i13(0), **.**i15(0), **.**i12(asl), **.**i14(asr), **.**i8(lsl), **.**i10(lsr) ,**.**i9(slt), **.**sel(ALUCtrl), **.**s(s)); bitMux16to1 OVERFLOWMUX(**.**i0(overflow\_subtraction), **.**i1(overflow\_addition), **.**i4(overflow\_dec), **.**i5(overflow\_inc), **.**i12(overflow\_asl), **.**i8(overflow\_lsl), **.**sel(ALUCtrl), **.**s(overflow), **.**i7(0), **.**i11(0), **.**i13(0), **.**i15(0), **.**i2(0), **.**i3(0), **.**i6(0), **.**i9(0), **.**i10(0),**.**i14(0)); |

## Part III: Register File

We are asked to build a register file with 32 16-bit registers, two read ports, and a single write port. The register file should be able to support two concurrent read operations and one write operation. The top level semantic is as follows:

* The signals busA, busB, and busW are 16-bits wide, while the Ra, Rb, and Rw are 5-bits wide.
* Ra, Rb, and Rw represent the address of the register that is being read or written to. The address will be a 5-bit number.
* BusW is the data that we are writing.
* The register file also requires clock and reset pins.
* You may use either structural or behavioral Verilog to implement the register file.
* In case of concurrent read/write operations to the same register, the output must reflect the new value.
* The reset signal is a synchronous reset which has higher priority than all write signals. If reset is asserted at the rising edge of the clock, all of the registers are set to all 0’s.

Below is the high level diagram of the register file:



Unlike the previous section, we are allowed to use behavioral logic here.

The implementation is a bit more complicated. First, at posedge clk, we check if rst is asserted. If so, we set all registers, busA, and busB to 0.

If rst is not asserted

# Answer to Questions

1）What is the difference between structural and behavioral Verilog? Please provide an example of a structural and behavioral implementation of a multiplexer.

Structural Verilog uses components such as an basic digital logic gates and mux to design the functions. It is similar to connect the circuits.

Behavioral Verilog uses to describe the function of a design in an algorithmic manner.

Structural Mux:

module mux2(in0, in1, sel, out);

input in0, in1, sel;

output out;

assign out = (~sel & in0)| (sel & in1);

endmodule

Behavior Mux:

module mux2(in0, in1, sel, out);

input in0, in1, sel;

output out;

always @\*

If (sel==0) begin

out=in0;

end

if (sel==1) begin

out=in1;

end

endmodule

2) What is the difference between an asynchronous and synchronous Multiplexer? Please provide a brief explanation on how you could implement both using behavioral Verilog.

Asynchronous Multiplexer will output the right result whenever the input signals change. Synchronous Multiplexer will output the result until next state is triggered.

In order to implement both using behavioral Verilog, we only need to change the sensitivity list in the always statement.

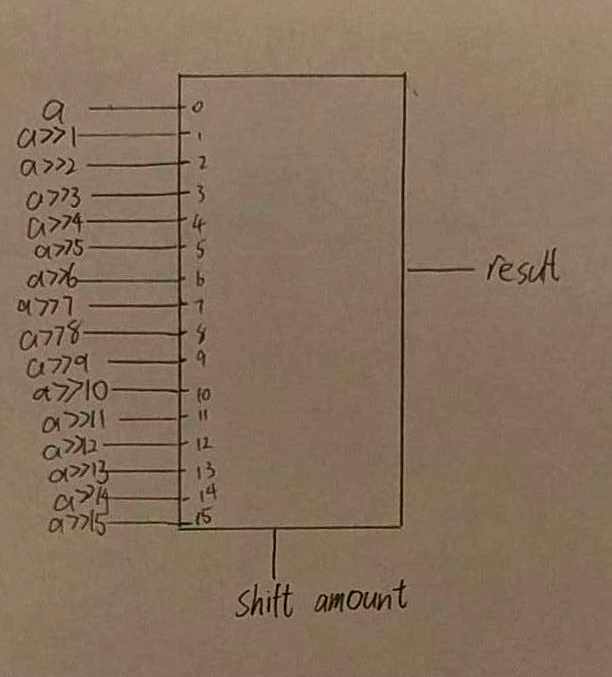
Async: always @\*

Sync: always @(posedge clk)

3) What is the difference between an arithmetic and logical shifter?  
Left-shifting operations are exactly same for arithmetic and logical shifters. They both move 0 to the least significant bit.  
For right-shifting operations, logical shifter will move 0 to the most significant bit while arithmetic shifter will replicate the sign bit to fill bit positions.

4) Assuming that you did NOT use Behavioral Verilog to implement an arithmetic shifter, how could you design one from scratch? Please include a simple diagram.

We can a mux which is connected to all possible shifted registers(inputs) and shift\_amount(select).



# Simulation

Subtraction:

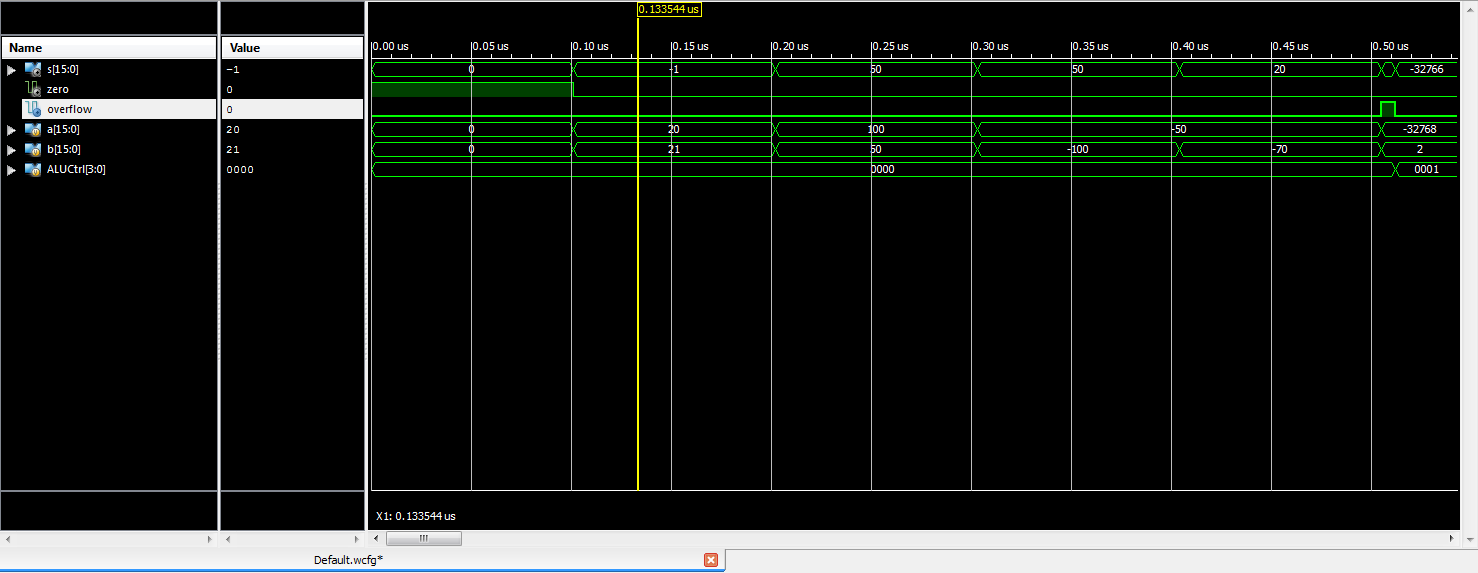
a = 20, b = 21, s = -1, overflow = 0

a = 100, b = 50, s = 50, overflow = 0

a = -50, b = -100, s = 50, overflow = 0

a = -50, b = -70, s = 20, overflow = 0

a = -32768, b = 2, s = 32766, overflow = 1



Addition:

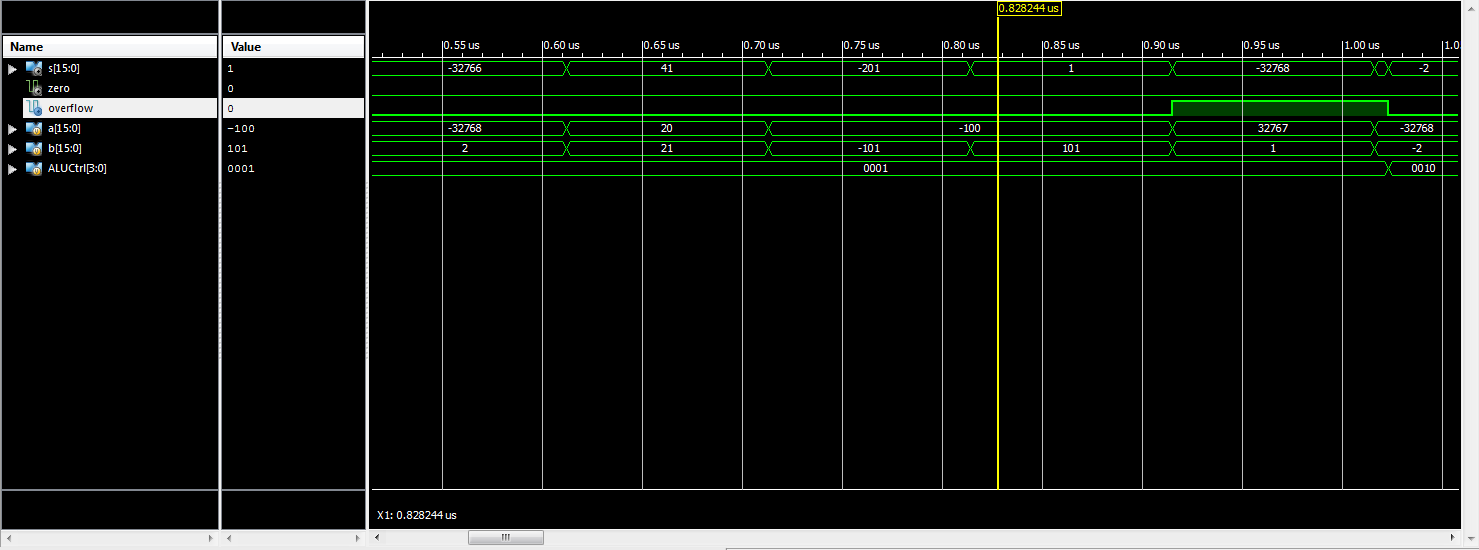
a = 20, b = 21, s = 41, overflow = 0

a = -100, b = -101, s = -201, overflow = 0

a = -100, b = 101, s = 1, overflow = 0

a = 32767, b = 1, s = -32768, overflow = 1

a = -32768, b = -2, s = 32766, overflow = 1



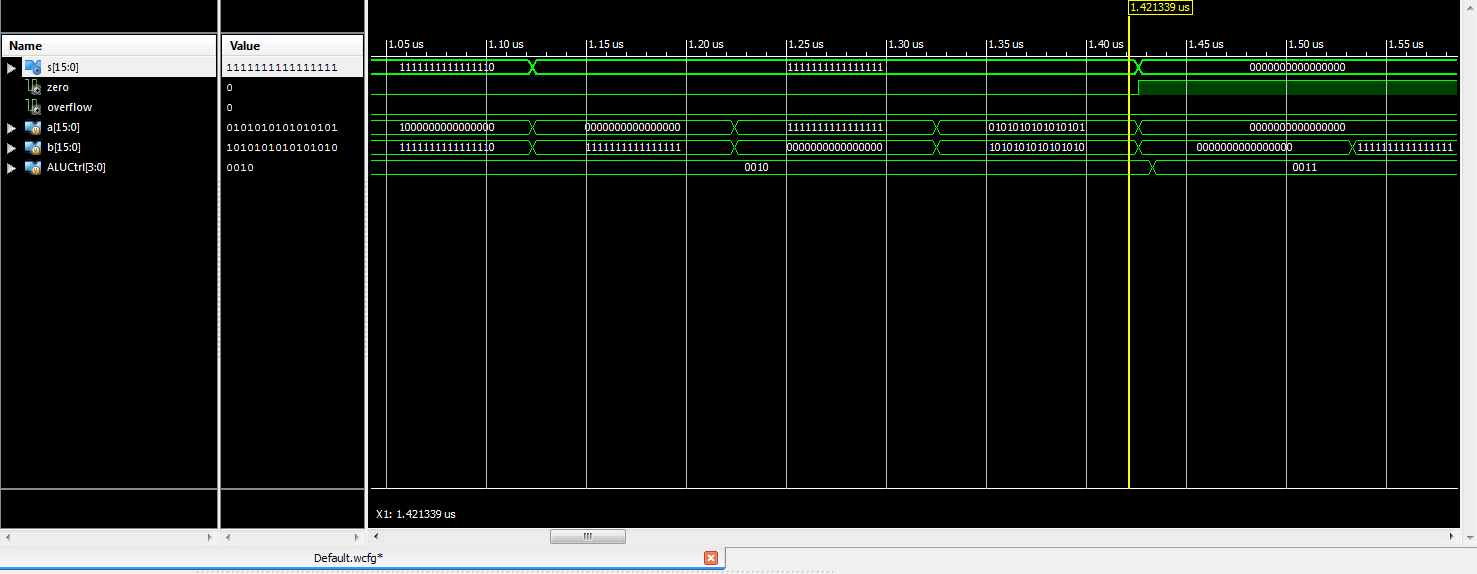
OR:

a = 0000000000000000, b = 1111111111111111, s = 1111111111111111

a = 1111111111111111, b = 0000000000000000, s = 1111111111111111

a = 0101010101010101, b = 1010101010101010, s = 1111111111111111

a = 0000000000000000, b = 0000000000000000, s = 0000000000000000



AND:

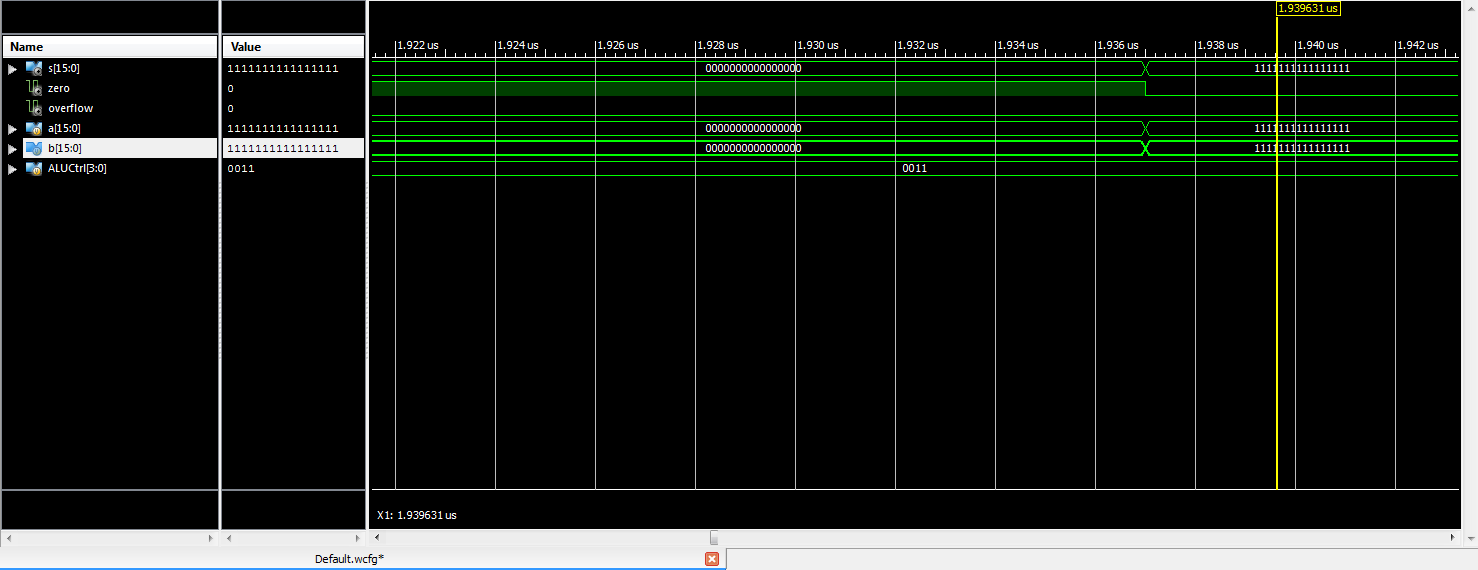
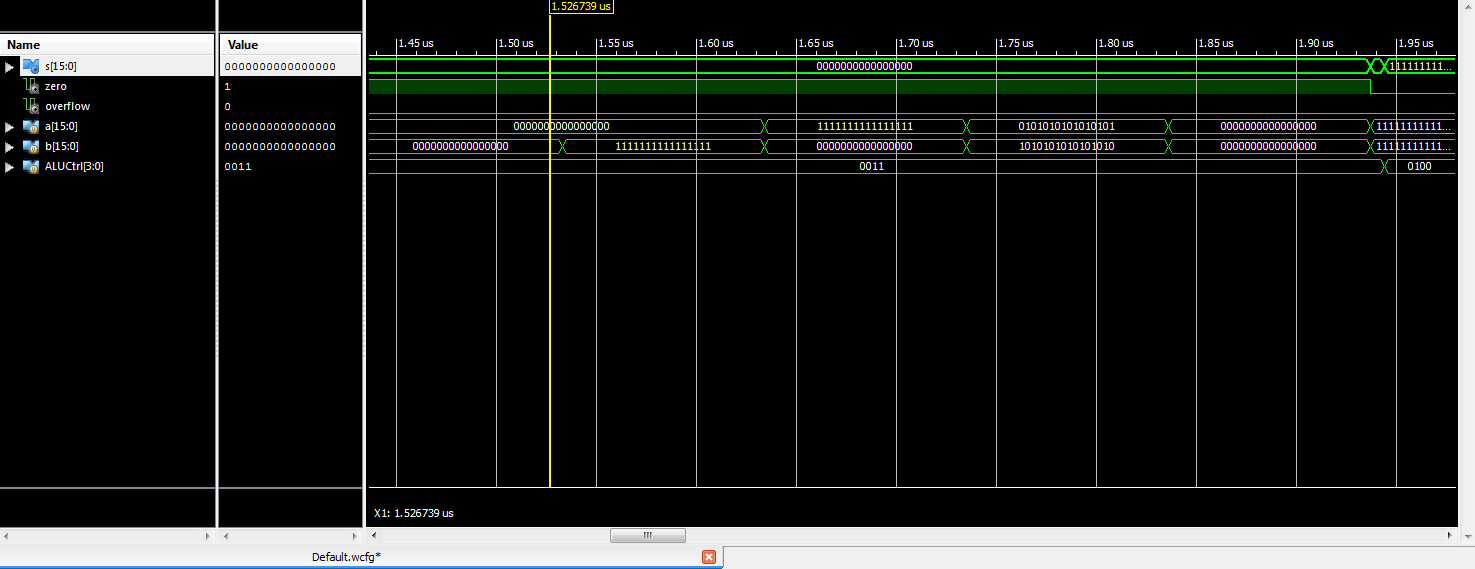
a = 0000000000000000, b = 1111111111111111, s = 0000000000000000

a = 1111111111111111, b = 0000000000000000, s = 0000000000000000

a = 0101010101010101, b = 1010101010101010, s = 0000000000000000

a = 0000000000000000, b = 0000000000000000, s = 0000000000000000

a = 1111111111111111, b = 1111111111111111, s = 1111111111111111



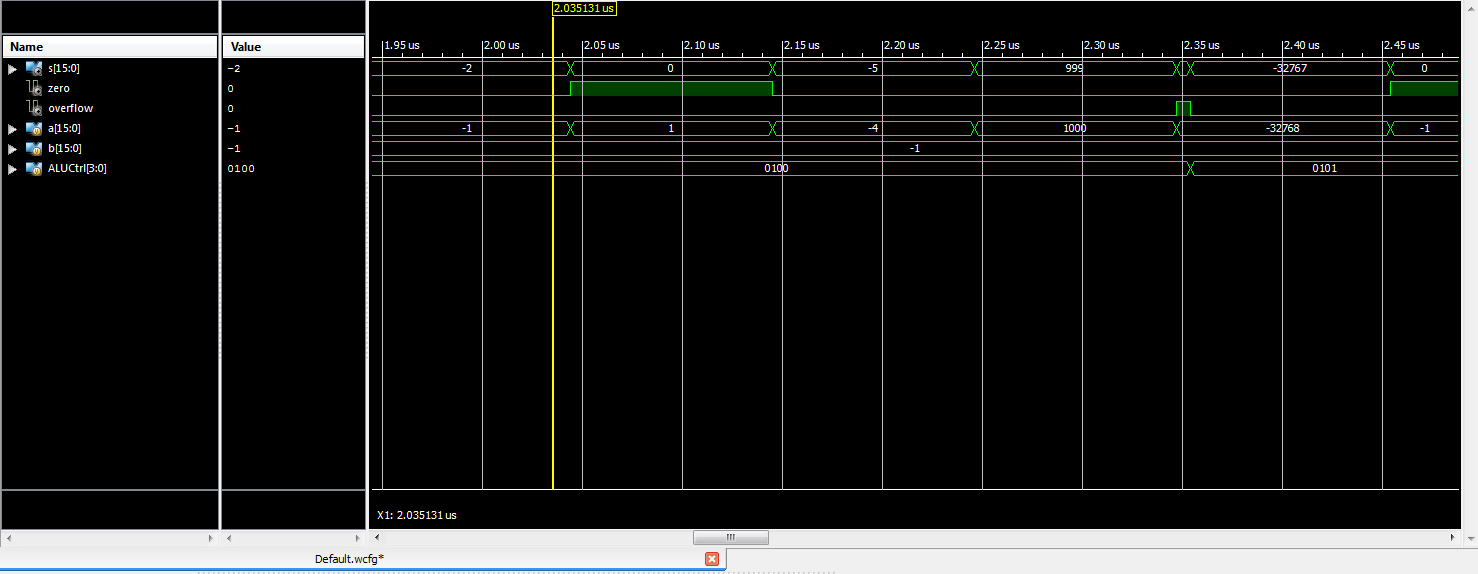
Decrement:

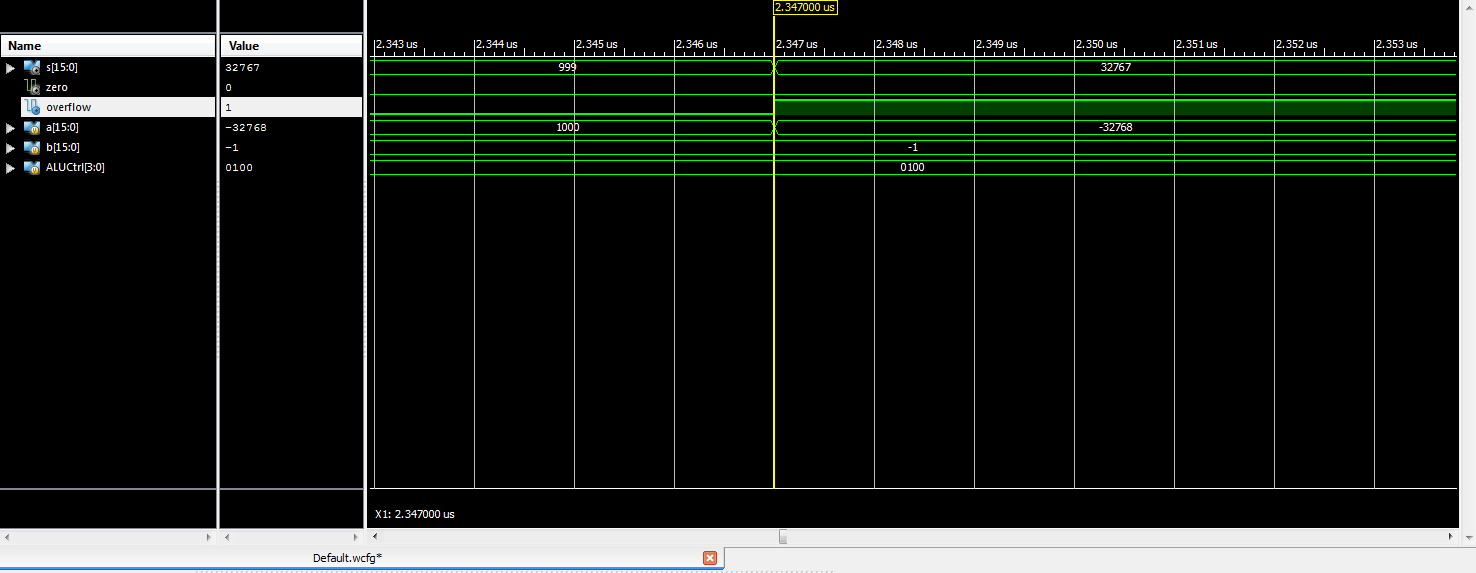
a = 1, s = 0, overflow = 0

a = -4, s = -5, overflow = 0

a = 1000, s = 999, overflow = 0

a = -32768, s = 32767, overflow = 1





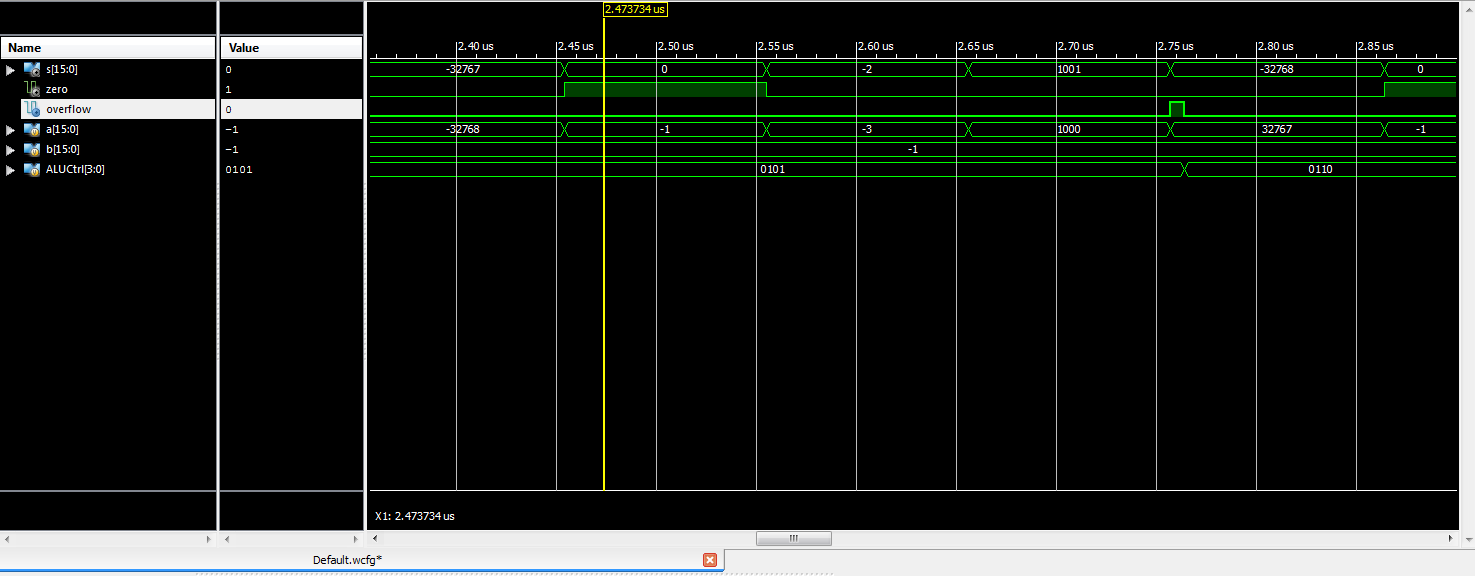
Increment:

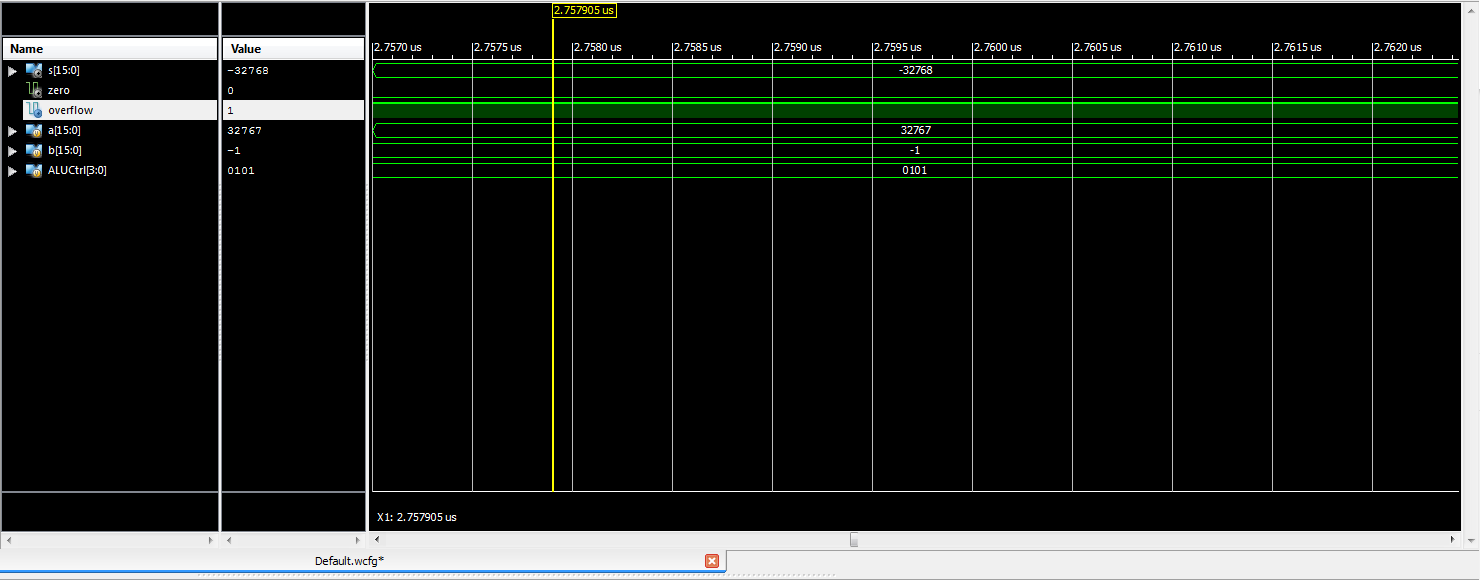
a = -1, s = 0, overflow = 0

a = -3, s = -2, overflow = 0

a = 1000, s = 1001, overflow = 0

a = 32767, s = -32768, overflow = 1



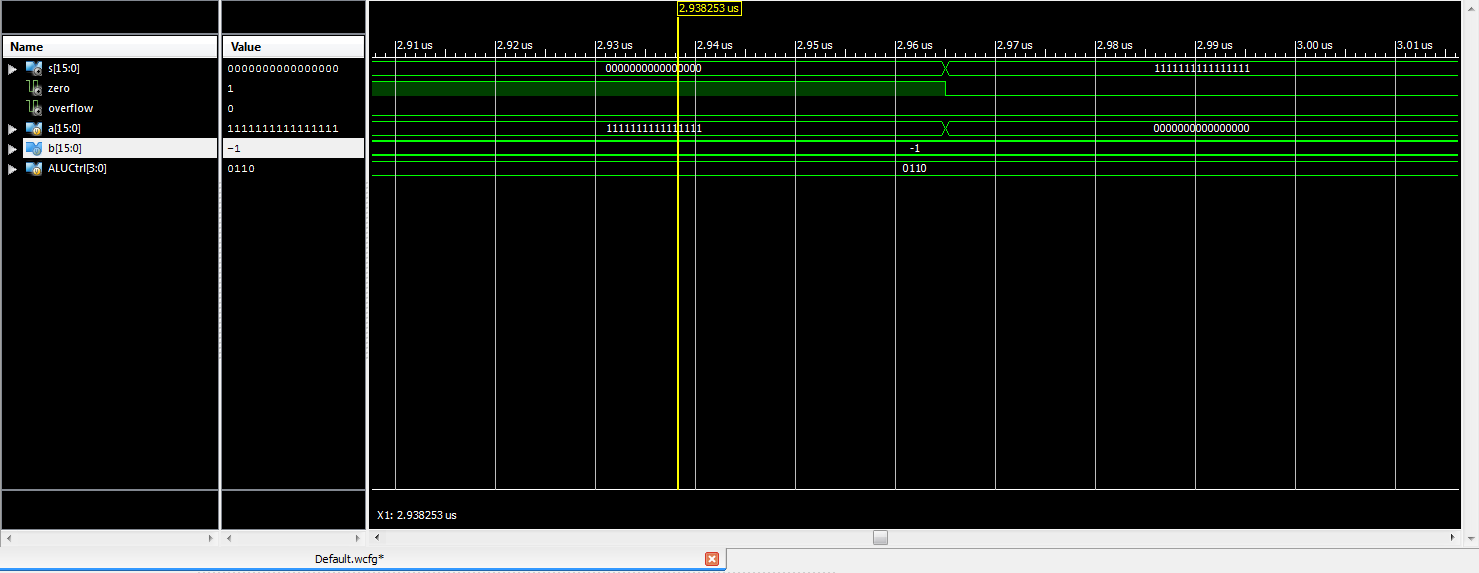


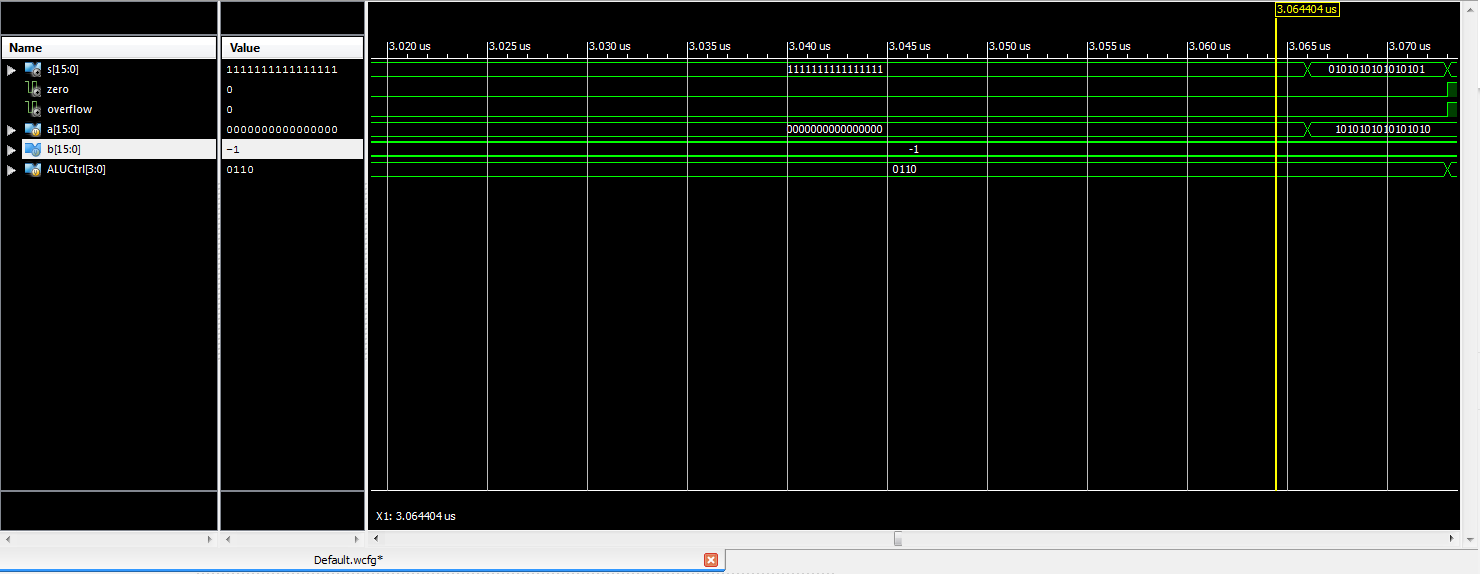
Invert:

a = 1111111111111111, s = 0000000000000000

a = 0000000000000000, s = 1111111111111111

a = 1010101010101010, s = 0101010101010101





Arithmetic Shift Left:

a = 0000000000000001, b = 3, s = 0000000000001000, overflow = 0

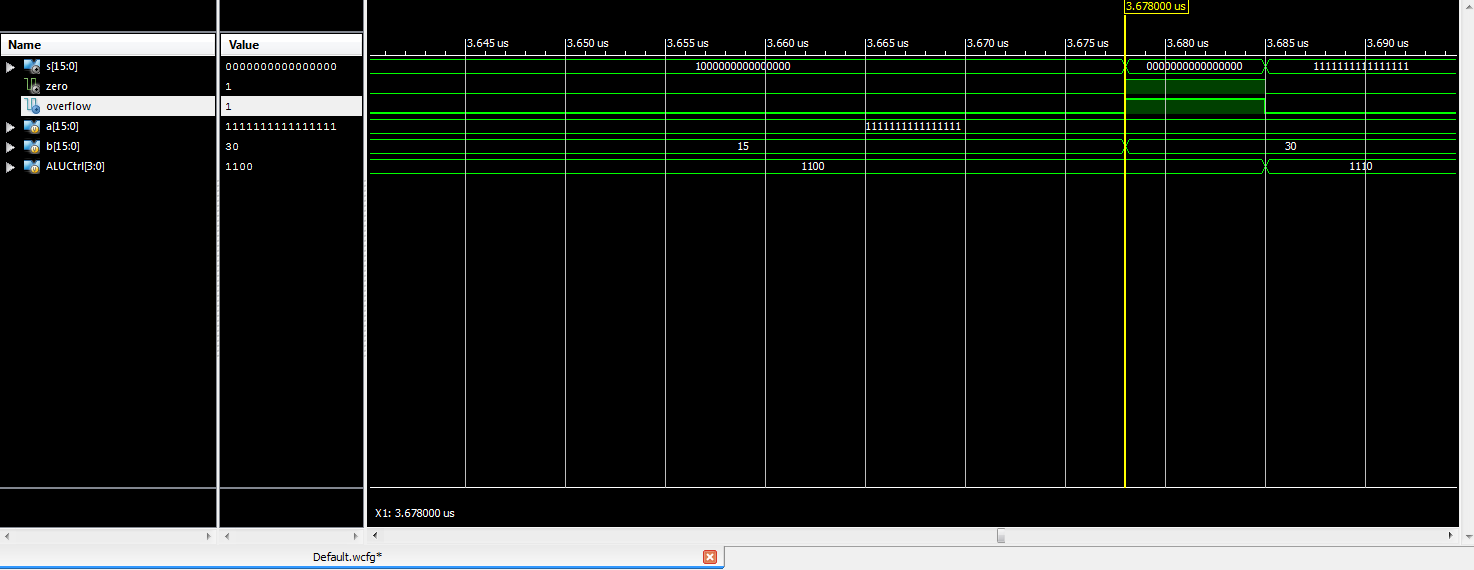
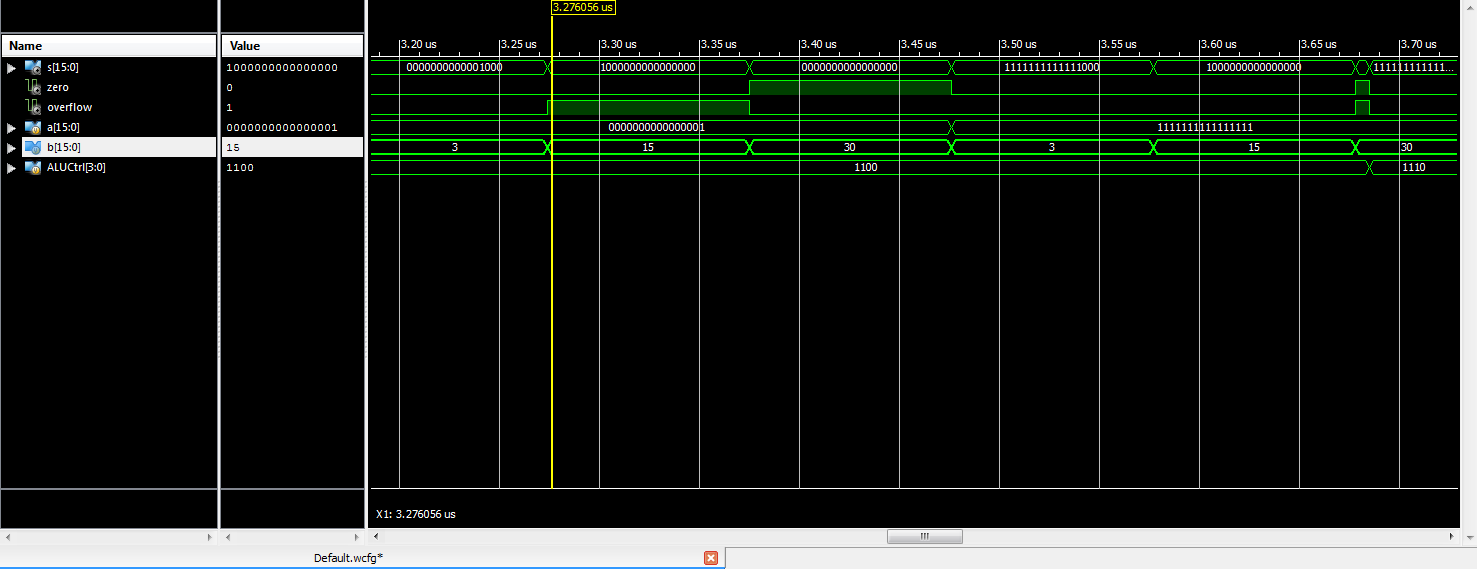
a = 0000000000000001, b = 15, s = 1000000000000000, overflow = 1

a = 0000000000000001, b = 30, s = 0000000000000000, overflow = 0

a = 1111111111111111, b = 3, s = 1111111111111000, overflow = 0

a = 1111111111111111, b = 15, s = 1000000000000000, overflow = 0

a = 1111111111111111, b = 30, s = 0000000000000000, overflow = 1



Arithmetic Shift Right:

a = 1000000000000000, b = 3, s = 1111000000000000, overflow = 0

a = 1000000000000000, b = 15, s = 1111111111111111, overflow = 0

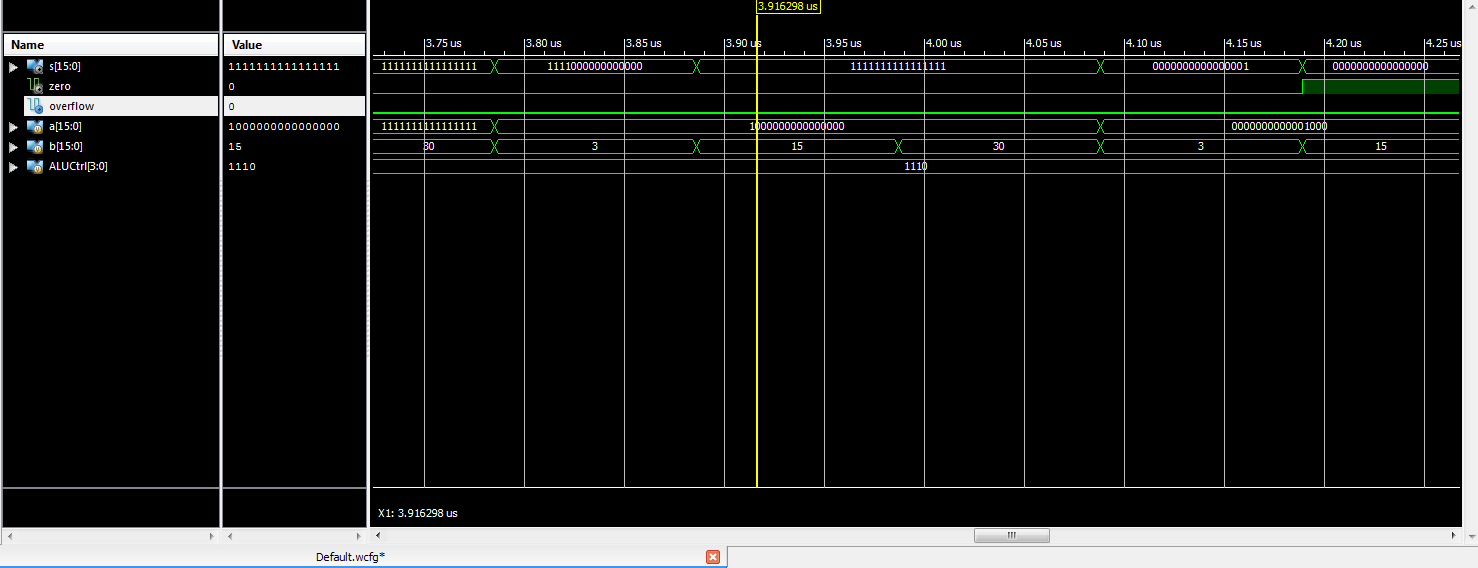
a = 1000000000000000, b = 30, s = 1

11111111111111, overflow = 0

a = 0000000000001000, b = 3, s = 0000000000000001, overflow = 0

a = 0000000000001000, b = 15, s = 0000000000000000, overflow = 0

a = 0000000000001000, b = 30, s = 0000000000000000, overflow = 0



Logical Shift Left:

a = 0000000000000001, b = 3, s = 0000000000001000

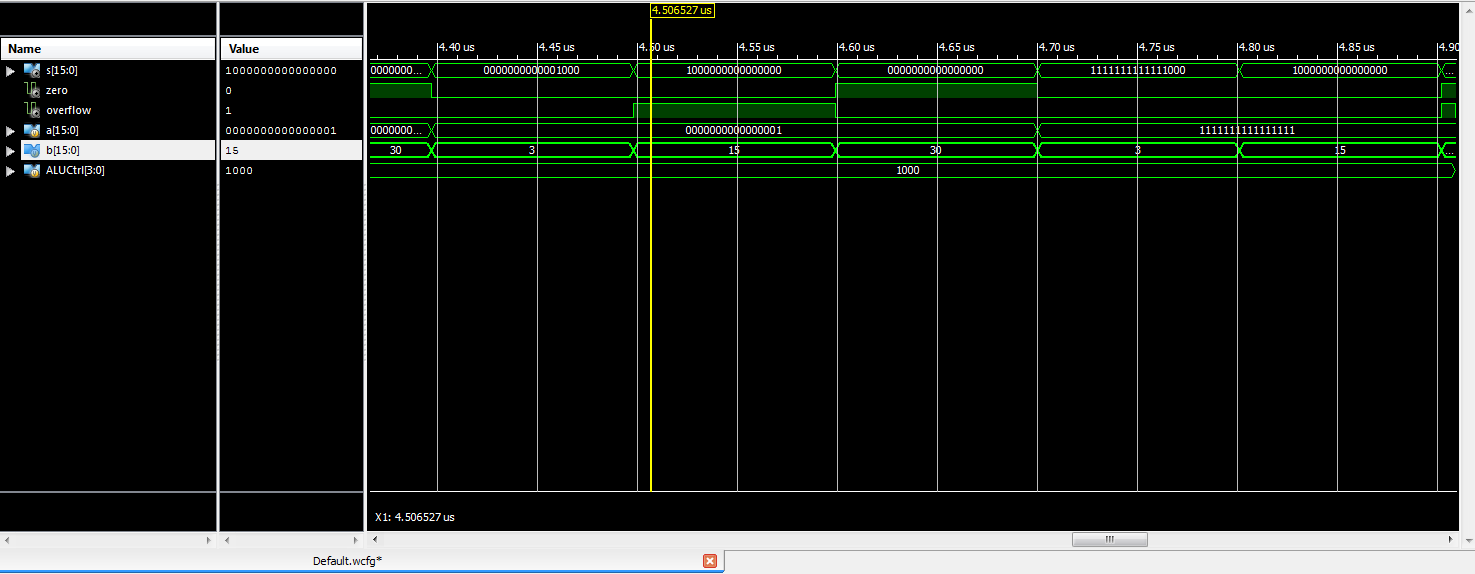
a = 0000000000000001, b = 15, s = 1000000000000000

a = 0000000000000001, b = 30, s = 0000000000000000

a = 1111111111111111, b = 3, s = 1111111111111000

a = 1111111111111111, b = 15, s = 1000000000000000

a = 1111111111111111, b = 30, s = 0000000000000000



Logical Shift Right:

a = 1000000000000000, b = 3, s = 0001000000000000

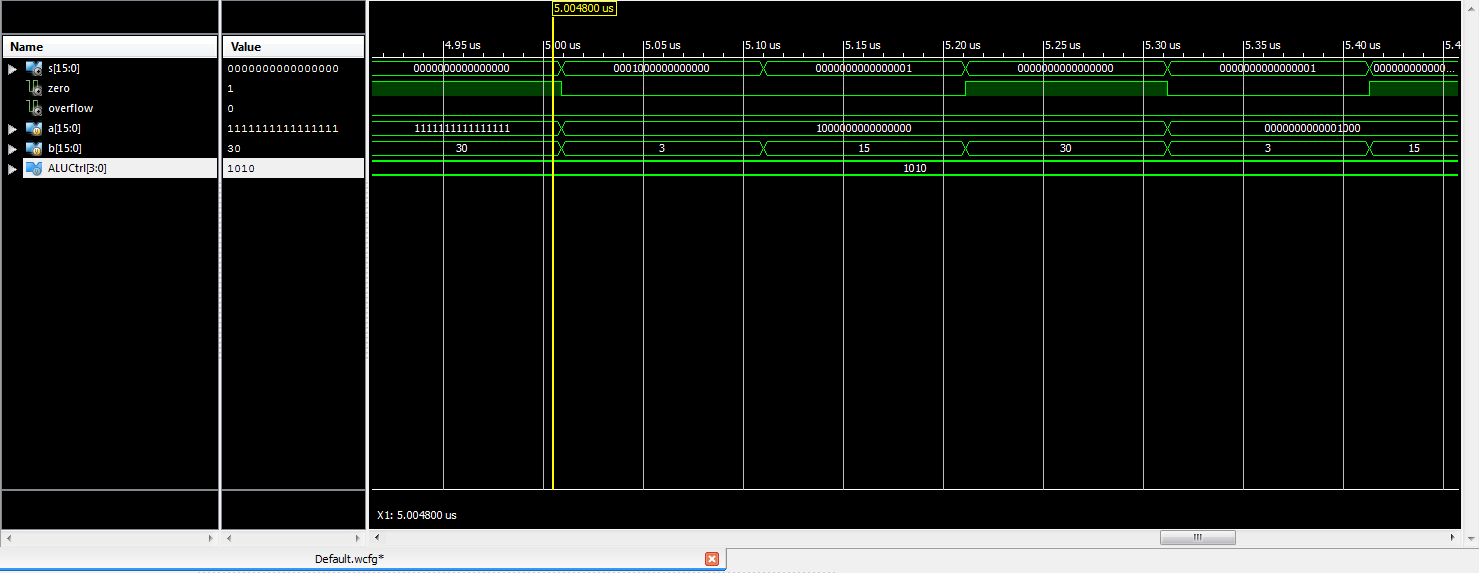
a = 1000000000000000, b = 15, s = 0000000000000001

a = 1000000000000000, b = 30, s = 0000000000000000

a = 0000000000001000, b = 3, s = 0000000000000001

a = 0000000000001000, b = 15, s = 0000000000000000

a = 0000000000001000, b = 30, s = 0000000000000000



Set on Less Than or Equal:

a = 2000, b = 3000, s = 1

a = 3000, b = 2000, s = 0

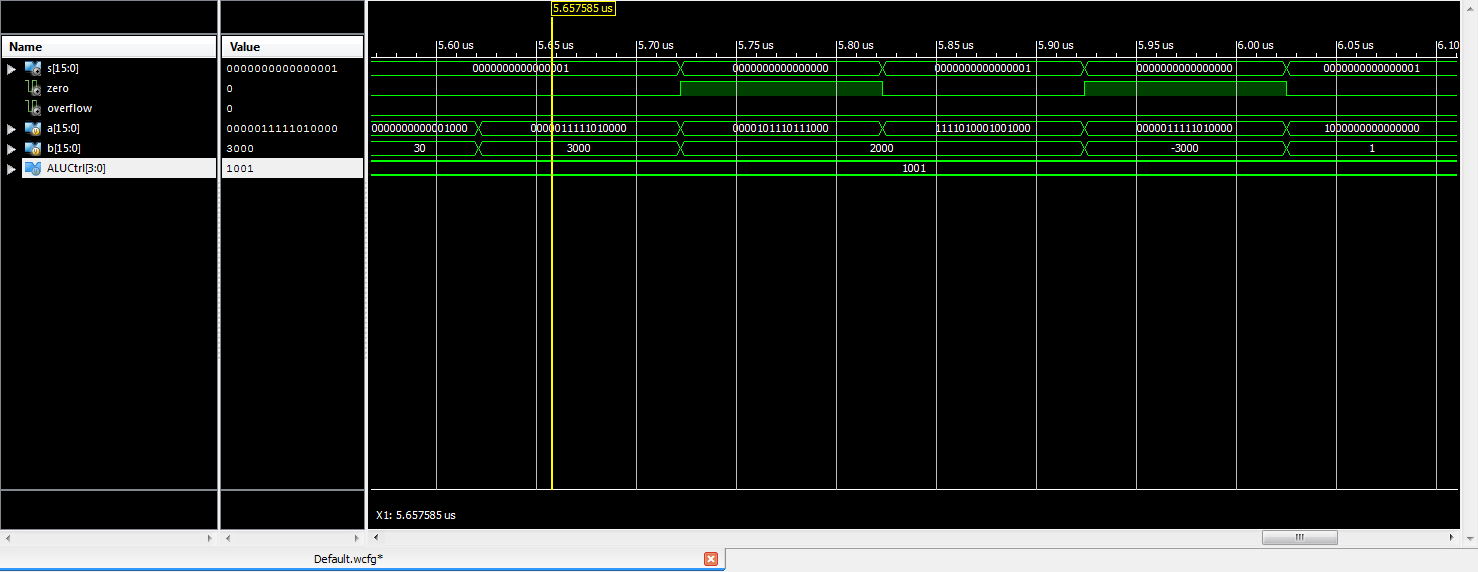
a = -3000, b = 2000, s = 1

a = 2000, b = -3000, s = 0

a = -32768, b = 1, s = 1

a = 1, b = 1, s = 1

a = -32768, b = -1, s = 1



# Encountered Problem

1. 16-bit 16-1 MUX  
   In part II, the output S depends on the control signal. Thus, we plan to use a 16-1 MUX to determine the output. We first implemented a 1-bit 16-1 MUX using the components provided and then realized that the alternative results are 16-bit long, therefore they cannot be simply passed to the MUX we have. Therefore, we built a 16-bit 16-1 MUX by parallelizing 16 1-bit MUXs, one for each bit.
2. Verilog loop condition  
   We plan to use for loop to implement a shifting operation. But, we later realized that a variable cannot be used as a condition in Verilog. So, we changed our design to repeat statements and it works well.