**CS M152B Lab 1: ALU and Register File**

**By: Chris Baker and Rodrigo Gonzalez**

**UID: 105180929, 905161154**

**TA: Ghazal Ershadi**

1. **Introduction**
   1. **Overview**

The Arithmetic Logic Unit (ALU) is an essential component in computers which performs the arithmetic and bitwise operations that are essential to computing. In our lab we are tasked with designing an ALU which can perform Subtraction, Addition, Decrement, Increment, Invert, Arithmetic Shifts, Logical Shifts, and Gate Level logic on the inputs passed into the ALU. The ALU provides much more functionality than the surface of a given operation which can be seen in different Instruction Set Architectures (ISA’s) implementations of various ISA specific instructions. For example, the use of shifting and addition make it possible to jump or branch to different instructions from the base pointer's current position in a program’s instruction list memory.

The ability of the ALU to handle the computations necessary in an ISA allows us to work with a Register File which holds various general purpose and preset registers which hold meaningful data for these instructions and operations. Based on the input of signals to a register file, which are driven from the program instructions or feedback from ALU and other units like the Memory Unit, we can read, write, or update the value of registers, buses, and even some other memory locations.

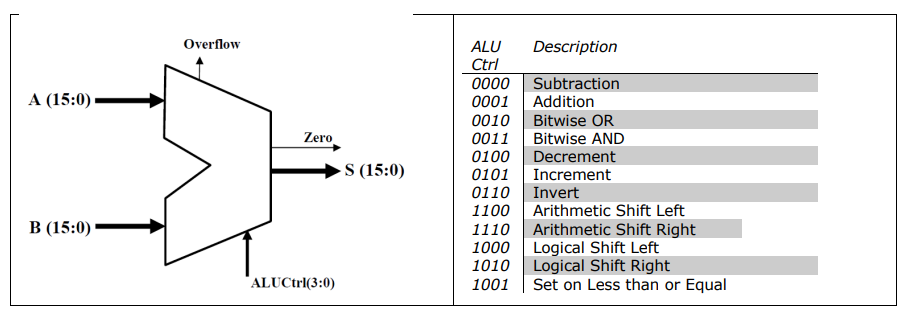
Through the use of Gate Level Logic and the creation of Digital Components such as Muxes, Adders, and Shifters, we will be able to connect the modules in such a way that we can perform the operations required in our 16-Bit ALU as well as our Register File to emulate the functions of general computer architecture common in PC’s.

* 1. **Requirements**

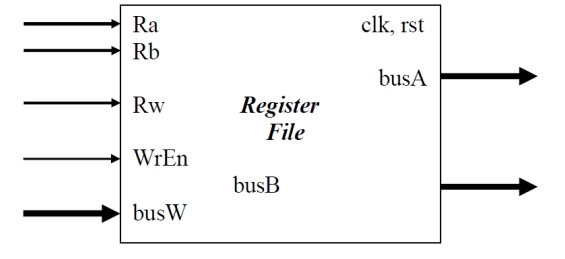
The requirements for this lab are split up into three parts. First, design a 1 bit ALU that supports 3 to 4 instructions and is designed entirely in structural Verilog. No demo is needed for this part, however a testbench that tests all inputs and operations is required. Then, we must design a 16 bit asynchronous ALU that supports 12 different instructions. To complete the lab we design a Register File that supports concurrent read and write operations and contains 32 16-bit registers.

In our 1-Bit ALU task we were not required to come up with Gate Level Operations that were already provided such as the operations for AND, OR, and NOT gates. However, it was necessary to implement a 2-1 Multiplexer. We were also instructed that the 2-1 Multiplexer may be a Behavioral Model where we could branch into creating further 4-1 Multiplexers and even 16-1 Multiplexers hierarchically. The last required task for designing the 1-Bit ALU was to design a testbench to perform different rudimentary operations as instructed by our TA including addition, subtraction, and gate logic.

In designing the 16-Bit ALU it has been noted that the output of signal Zero should be 1 whenever the output S is 0. Whenever S is not 0, then Zero should result in 0 to designate its namesake result. Arithmetic operations should yield an Overflow signal of 1 when a sign change occurs from the original expected number. Further, when performing a shift, the input A is shifted by the amount B. Lastly, the Invert must yield the proper sign change value of the input. We are required to use structural coding for the 16-Bit ALU, but it has been authorized that we may use Behavioral Verilog in the shifter, however our team completed this whole component through structural means anyways. A picture of the ALU design goal as well as the operations supported is shown below.



The Register File must contain 32 16-bit registers with 2 read ports and 1 write port. Further, this unit must be able to perform two concurrent read operations and a write operation. The busA, busB, and busW are 16 bits while the Ra, Rb, and Rw registers are 5 bits wide and contain the memory address for what is being read or written to. The buses represent the data in the buffer that is being read or written. The Register File must also contain a clock, reset, and write enable signal. As with most applications, the reset signal will have the highest priority and can trigger the resetting of all registers to 0 on a rising clock edge. Should we encounter a concurrent read and write to the same register, then we must reflect the new value. Lastly, for this component we were given authority to implement the code through either behavioral or structural means. To better depict the information above, we have provided an image of this component below.



1. **Design and Implementation**
   1. **1-Bit ALU**

In order to create our 1-Bit ALU we needed to first design the Muxes. A multiplexer

takes n inputs and selects 1 output. Generally a Mux is configured to select the first input when select is 0, the second input when select is 1 and so on incrementing the select signal corresponding to the next input. Thus, in order to design a 2:1 Multiplexer we need the select signal to choose input A, the first one, when select is 0. Further, we need the select signal to choose input B when select is 1. We can depict our logic gate designs using Boolean Algebra where this expression is . To be explicit, the + symbol represents an OR gate, two characters next to each other are inputs to an AND gate and the line above a character represents a signal that was passed through an inverter.

Since our 1-Bit ALU needs to pick from 4 operations we need the select signal to be bits wide. So, this required us to extend the 2:1 Mux to a 4:1 Mux. We accomplished this by creating a new module called mux\_4\_1 where the function of this module can be represented as Mux(A,B,C,D, S[1:0]) = OUT. We created wires m1 and m2 which would hold the output of a first level of 2:1 Muxes consisting of mux1 and mux2. Mux 1 is represented as the function mux1(A,B,S[0]) = m1 and mux 2 as mux2(C,D,S[0]) = m2. Their final results are passed to a second level mux called mux 3 represented as mux3(m1,m2,S[1]) = OUT. In essence we used the first level muxes to select 1 input as the output from the initial inputs and pass that in a wire to mux3 in the second level. Mux 3 then just selects one of the two passed inputs on wires m1 and m2 by using the second bit of information on S[1] to make the final selection from two inputs to 1 output. The second level of the 4:1 mux is a 2:1 mux written as .

Now that we had the ability to select from 4 operations we were able to create the 1-Bit ALU. ALUCtrl was a signal that determined which operation would be used {AND, OR, Add, Invert}. To design AND and OR we used the gate level logic functions built into vivado which appear as function\_name(output\_wire, input\_wire1, input\_wire2). So we used the and(and\_out, A, B) and OR(or\_out, A, B) gate functions to store the results into the respective output wires for those gate operations. The Add operation required us to implement a new module coded as such:

module addbit(

input cin,

input a,

input b,

output sum,

output cout

);

wire x1, a1, a2;

xor(x1, a, b);

xor(sum, x1, cin);

and(a1, a, b);

and(a2, x1, cin);

or(cout, a1, a2);

endmodule

We designed this code based on the Full Adder Boolean Algebra expression that yields the result of adding inputs A,B, and Cin. The resulting formula is and . Cout holds the overflow result which would occur anytime we have a value that surpasses 1 from addition since binary values of 1 bit can only represent 0 or 1 for one binary digit. We asked the TA if we needed to do subtraction, but she said it was redundant and these operations were fine. We also only set overflow here in the 1-bit add module since this was a step toward designing the 16 bit ALU where we handled overflow for each different module. We later solved the overflow for different modules by creating wires in modules that did not need overflow and kept them at 0 which could be replicated here.

Lastly, the Invert signal was another gate function that took an input A and returned inverted A through the gate function not(invA, A). Our 1-Bit ALU connected the inputs to each gate level operation as well as to the inner modules called. In order to return the results we used our 4:1 mux:

mux\_4\_1 newmux(

.A(and\_out),

.B(or\_out),

.C(add),

.D(invA),

.S(ALUCtrl),

.OUT(S)

);

* 1. **16-Bit ALU**

Our task for this section was to design and built a 16-bit asynchronous ALU including the operations of {Subtraction, Addition, OR, AND, Decrement, Increment, Invert, ASL, ASR, LSL, LSR, SLTE} shown in the first diagram on this report. Since we have 12 operations we need the select signal to be bits wide. Unlike our 1 bit ALU with 4 operations and 1 output as well as overflow for addbit, we now need an output for a zero flag also. We also noticed that the operations are not necessarily incremented by 1 after each other, so we must pay attention to the signal values and not just increment when designing them. Since we wanted to first ensure that the modules for eac operation were working correctly we decided to code those first and test them robustly before connecting them to a 16:1 Mux.

Our first task was to create a subtractor, however this would be like performing addition with some extra checks involved, so we instead handled 16 bit addition first. We looked to implement a ripple carry adder that utilized our 1 bit adder. The inputs of our adder\_16bit module are A and B and the outputs are sum and overflow. Overflow is 1 bit wide while A,B, and sum are 16 bits wide. To accomplish this we created 15 wires {C0 … C14} that would hold the sum result from each addbit for bits [14:0]. The 15th bit is used for determining the sign of the number. The very first addbit for bit position 0 has cin set to 0 and performs addition using A[0] and B[0] and puts outputs into sum[0] and cout[0]. The next addbit for bit position 1 performs similar logic except the carryin is no longer 0, rather it uses the previous carry out as the carry in value as such:

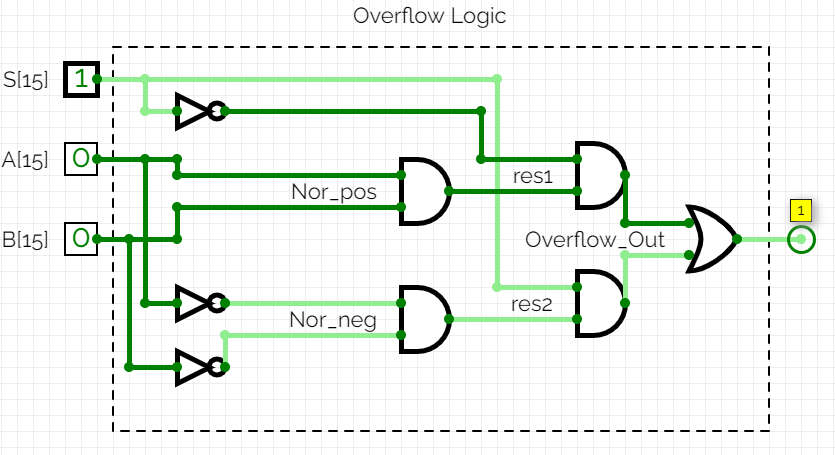
addbit s0 (.cin(0), .a(A[0]), .b(B[0]), .sum(sum[0]), .cout(C0));

addbit s1 (.cin(C0), .a(A[1]), .b(B[1]), .sum(sum[1]), .cout(C1));

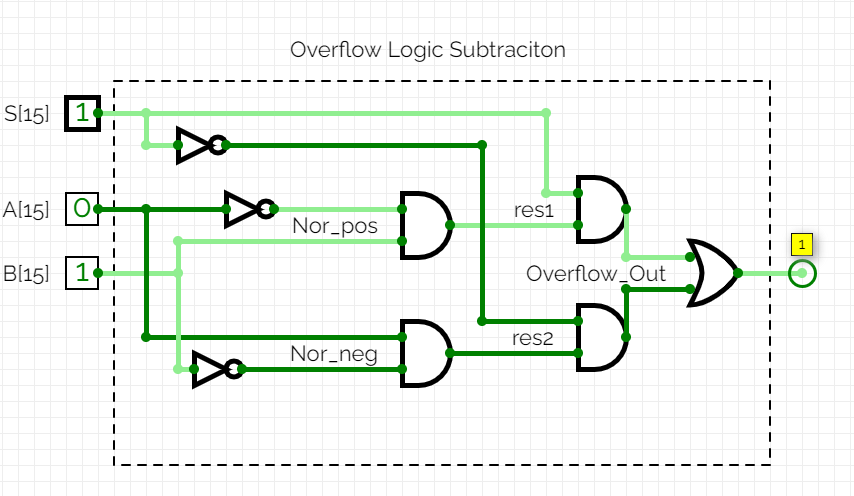
…

addbit s15 (.cin(C14), .a(A[15]), .b(B[15]), .sum(sum[15]), .cout(of\_res));

This essentially handled Addition, but did not take into account the overflow. Overflow occurs when there is a sign change between the inputs and the output. One such case is adding two positive numbers, but getting a negative result. In order to handle this we designed a system of gates that would compare for A and B being positive against the result being negative as well as A and B being negative against the result being positive. If either case was true that we encountered a sign change, then there is overflow. We created and attached wires as were needed to emulate the system below which shows one possible configuration of values for the MSB in A, B, and S inputs.

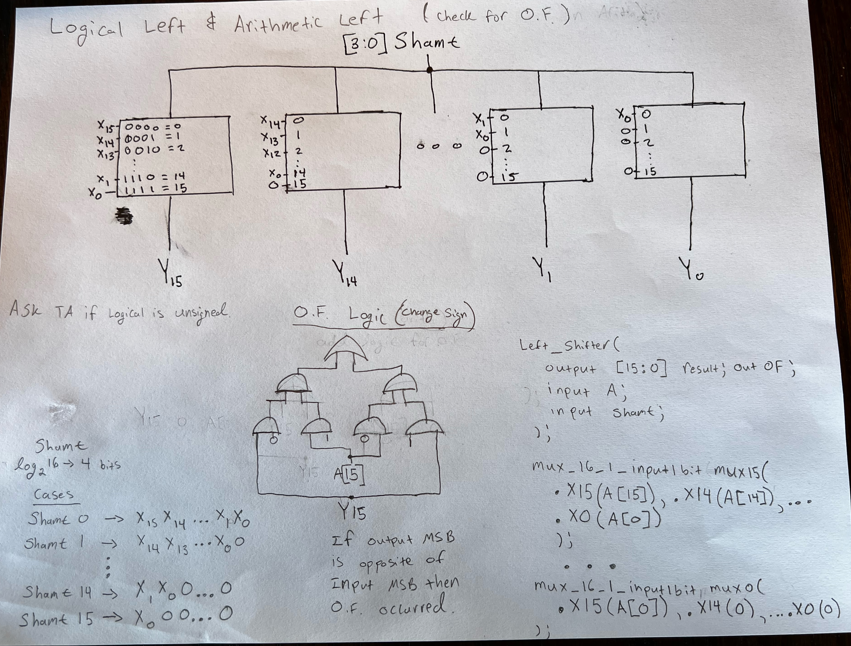


Designing the subtractor was not much different. Addition works on negative numbers which we saw when using our testbench. So, in order to subtract a value we need to just change the second input to a negative value and pass both input A and negative input B into our addition module. To get the opposite sign value for B we can utilize the formula to change the sign of a 2’s Complement number. This formula is x = ~x + 1. We implemented this formula by assigning each notB wire n to each n bit of the input B. We then passed in A=1 and B = notB into our 16 bit addition module to return the properly inverted value of B with the same magnitude, but different sign. Note we also needed a temporary wire to sit in the overflow output as a junk unused value. We then ran our second addition module with inputs of A and inverted B which returned the correct sum and another temporary overflow result. We handled our overflow by using the same name conventions, but swapping some of the wire inputs around. The reason is that we are now looking for underflow since we have one number changing sign to subtract from another number. So, if the inputs were different signs, and the result is a changed sign of A then overflow should be set. In layman's terms, If A is positive and B is negative, then the result would be A - (-)B = A + B which is a higher positive value. Likewise if A is negative then A - B should result in a more negative value. However, if this does not occur in either case, then we trigger overflow. This happens as a result of limited bits and carry over in computing. Our diagram below shows how we solve overflow for subtraction (showing one possible case).

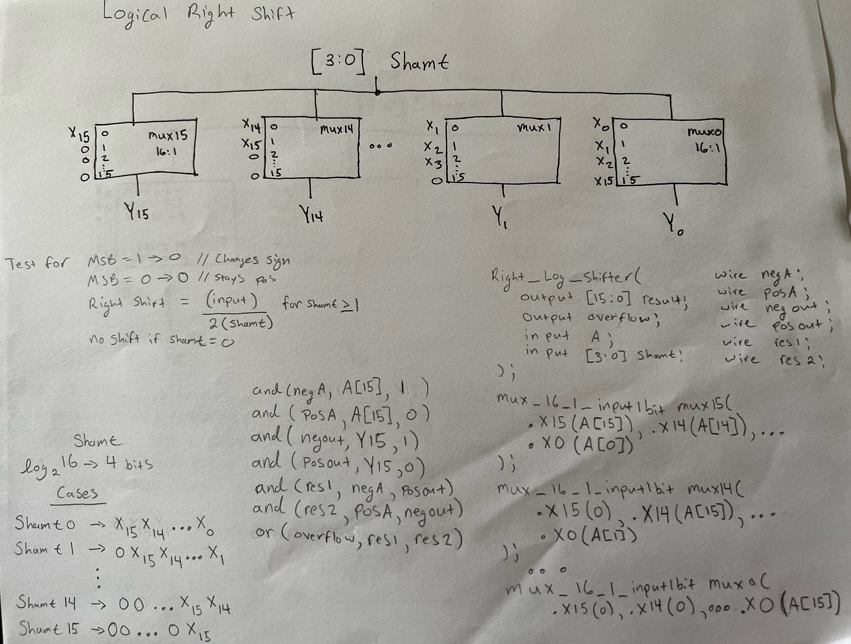


For bitwise OR we simply ran every bit of A and B through the or() operation in vivado. We performed the same methodology by running every bit of A and B through the and() operation to handle returning the AND operation result for the ALU. Decrement was achieved by using a subtractor and passing in a constant 1 as the B input. The overflow logic in place is more than sufficient to determine the overflow for the decrement operation, so the desired results were obtained. Increment was performed by using the addition module and passing a constant 1 into the B input. Again overflow within the addition module is already well designed to handle the case of adding 1 and determining if overflow occurred. For invert we implemented the same logic used in subtraction, however we now realized that we wrote out the inversion explicitly twice. This was a result of coding the addition and subtraction modules first, but if we did code invert before finishing subtraction, then we could have used that module within our subtraction module.

The shifting was designed by implementing 16 16:1 muxes. We created the 16:1mux from using 5 4:1 mux modules. This was done in a similar fashion to how we described using 3 2:1 mux modules to create the 4:1 mux. To summarize this process, 4 of the 4:1 muxes select 1 of the 4 inputs using the least two significant bits of the select signal. Then those output wires are sent to the last 4:1 mux which will select the final output using the remaining two bits of the select signal (the 2 MSB’s ). Since we are able to now select 1 input from a total of 16 inputs, then we are able to appropriately write an output value for each bit of a 16 bit number based on the 4 bit select signal. We do this by using 16 16:1 muxes where each 16:1 mux appropriately selects the input value based on the shift amount for that bit position. Logical Left and Arithmetic Left shifts essentially perform the same shift logic and were designed by hand with this being our initial design goals which were updated or modified as needed as we wrote the code. Notice that cases are listed at the bottom left, code for implementation is at the bottom right, system design is at the top, and general overflow system and ideas are in the middle.



We implemented similar designs for the Logical Right Shift as shown below. Notice that instead of x0 bit position eventually shifting all the way through from right to left (as well as the neighbors to the side of it) we now have x15 bit position shifting from left to right (with its neighbors). The cases again are on the bottom left, overflow logic in the middle, initial code ideas on the bottom right, and system design up top.



I lost the design schematic for our Arithmetic Right shift, but the main difference in this design from the Logical Right Shift photo above is that we kept the x15 bit as we shifted rightward. This is because we want to preserve the sign value by maintaining the MSB while continually dividing by 2 as we shift rightward. Thus, case shamt0 = x15,x14,...x0, case shamt1 = x15,x15,x14,...x1 , case shamt15 = x15,x15,...x15. Below is an example of the code for mux0 which holds the result for the LSB.

// Returns the shifted bit output for the LSB

mux\_16\_1\_input1bit mux0(

.X0(A[0]), .X1(A[1]), .X2(A[2]), .X3(A[3]),

.X4(A[4]), .X5(A[5]), .X6(A[6]), .X7(A[7]),

.X8(A[8]), .X9(A[9]), .X10(A[10]), .X11(A[11]),

.X12(A[12]), .X13(A[13]), .X14(A[14]), .X15(A[15]),

.S(shamt[3:0]), .OUT(Y[0])

);

…

// Returns the shifted bit output for the MSB

mux\_16\_1\_input1bit mux15(

.X0(A[15]), .X1(A[15]), .X2(A[15]), .X3(A[15]),

.X4(A[15]), .X5(A[15]), .X6(A[15]), .X7(A[15]),

.X8(A[15]), .X9(A[15]), .X10(A[15]), .X11(A[15]),

.X12(A[15]), .X13(A[15]), .X14(A[15]), .X15(A[15]),

.S(shamt[3:0]), .OUT(Y[15])

);

Notice in Arithmetic Shift Right that mux0 holds the MSB bit which will be A[0] for no shift, but A[15] if shifted 15 times. Contrastingly, mux15 is always A[15] no matter the amount of the shift because we are right shifting and preserving/passing along the MSB. We used similar overflow logic as with earlier here, but generally this is not encountered since the A[15] input is copied throughout the sequence so there is no sign change.

The Last Design part of the 16 bit ALU was the Set on Less Than or Equal (SLTE). We implemented this design by storing the xor result of A[15] and B[15] into an xor\_sign wire. Then we used the subtractor to store the sum of inputs A and B into a wire called set. The subtractor told us if the result was positive or negative which would let us know if A was greater than or equal to B. Then we used our 16 bit AND module on inputs set[15] and 1 to see if the result was negative and stored this in a wire called sub\_set. We used another 16 bit AND module on inputs A[15] and 1 to see if the original A input was negative and stored this into wire xor\_set. We then were able to pass the inputs into a 2:1 mux to generate the correct output as shown below:

mux\_2\_1\_16bit slt(

.A(sub\_set),

.B(xor\_set),

.S(xor\_sign),

.D(out)

);

Our 16-Bit ALU had the following inputs and outputs in its module:

module alu\_16bit(

input [15:0] A,

input [15:0] B,

input [3:0] ALUCtrl,

output [15:0] S,

output Overflow,

output Zero

);

We designed our top module such that the ALU would pass inputs A and B to every operation module. Each module was then instantiated and would then output its output for S and for Overflow. We then called an instance of our 16:1 16 bit mux module (alderaans\_last\_uncle) and ran the results through our wires and used ALUCtrl as the select and S as our Output. We repeated this by calling another mux (of\_mux) to input our overflow results. Separately, there is a zero output from which we made a z wire that for each bit held the nor(S[n], 0). This guarantees that the S signal with 0 evaluates to 1 when both are equal to 0. Then we take the AND of every z wire bit value to make sure the whole 16 bit S is zero and store the result into the zero wire for output such that Zero is set to 1 , when all S bits are zero. Thus, a 16:1 mux returned the results in S, another 16:1 mux returned the overflow results to Overflow, and separate gate logic on S using nors and an and gate returned our zero output.

* 1. **Register File**

The Register File needed to hold three 5 bit wide inputs for Ra, Rb, and Rw. There is also an input for clk, rst, WrEn, and a 16 bit wide busW. The last ports are two 16-bit output registers for busA and busB. We also needed to include 32 16-bit registers in the register file. In order to build our registers we create 32 16-bit registers we made an array of these registers:

reg [15:0] reg\_array [31:0];

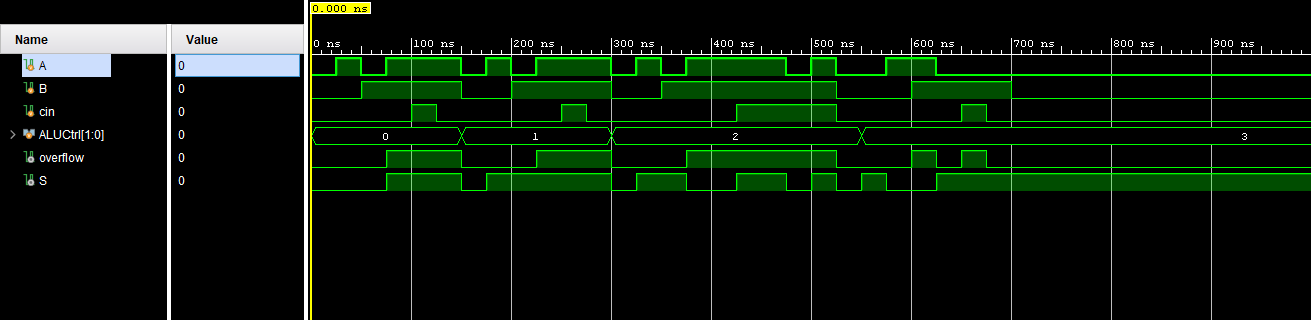
To implement the functionality of our register file we created an always block that triggered on positive clock edges. We first performed a check for the reset signal. If the reset signal occurred, then we would set every register in the register array to 0. We also would set the busA and busB to 0. We set the reset signal as the first check in our code to ensure that this signal, if caught, would have the highest priority over other signals and occur first.

Next we created a conditional to check for Write Enable. If this signal was high then we would assign the register in our register array with address Rw to hold the value of busW. Here we performed another check to see if the address in Ra was the same as Rw and if it was we would set busA to busW. This handled the case of concurrent read and write operations where we would hold the new value. Else we would assign the value in the register array at address Ra into bus A. We performed another check to do the same comparison between busB and busW to handle concurrent reads and writes for our b inputs and bus. If write enable was not set, then we just update busA with the value in the register array at address Ra and busB with the value in the register array at address Rb.

1. **Simulation Waveforms**
   1. **1 Bit ALU**

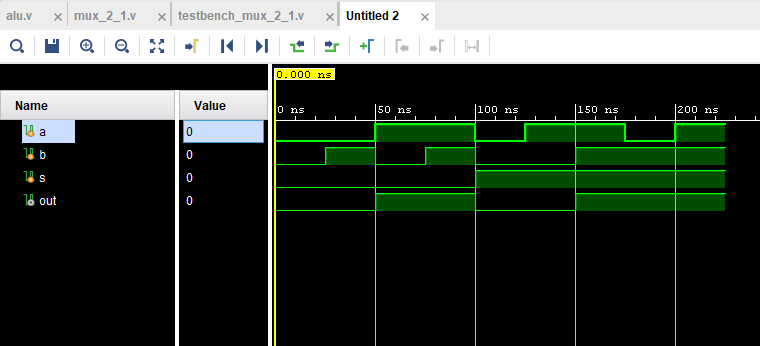
The following 1 bit ALU demonstrates add bit and overflow. When shown to the TA we were approved in not needing to demonstrate subtraction for this since the results were redundant. An OR would select either bit if set and an AND would result in 1 if both inputs were 1. Our TA approved that showing the following operations per the control signal [1:0] were sufficient {AND, OR, Add, Invert}.

For **control signal 0 (AND)** we see at around 75ns both A and B inputs are set to 1, which results in the output S being set to 1. The other signals are redundant and were approved to be left in the ALU to be used for the add operation and invert operation. We could easily assign overflow to a 0 wire for the AND and OR case as discussed with the TA, but she allowed for us to just show this in the 16 bit ALU. For **control signal 1 (OR)** we see that whenever either input is set to 1, then the resulting output S is 1 such as at time points 175ns and 200 ns as we toggled A on and Off while turning on B. Also notice how initially at 150ns, both inputs were low and the result was 0 for the OR output. For **control signal 2 (Add)** the sum is only set when just one of the following inputs is set XOR(Cin, a,b) as well as when all of the inputs are set (1,1,1). We see this occur at time points 325ns, 350ns when just A and B are set and at 425ns when all inputs are set. Notice all other configurations result in a 0 sum output. Further, we have a carry out whenever two inputs are set or when all three inputs are set shown from 325ns to 520ns. Lastly when the **control signal is 3 (Invert)** the result is the flipped value of the input A. At 550ns when A is 0, the result is 1. Also when A goes low at 625ns we see the result of S set to 1for the remainder of the signal, regardless of the second input’s value.

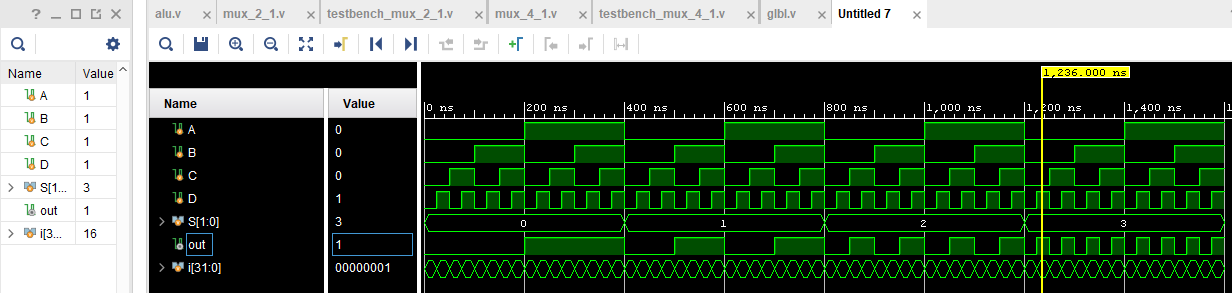
****

* 1. **Muxes {2:1, 4:1, 16:1, 16:1 16 bit output}**

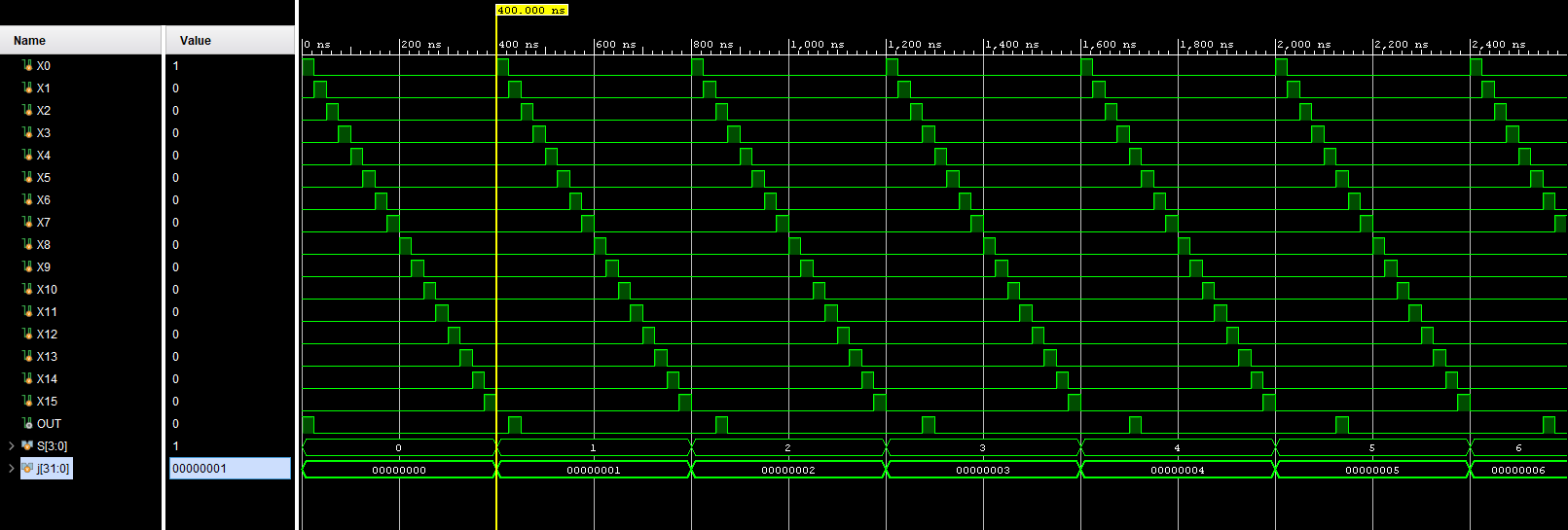
The 2:1 mux has three inputs which takes a,b,s and outputs the selected value. As mentioned in the design section the output is given by the formula: . We see that the output is 0 until 50ns. Then since A is 1 and S is 0 our formula yields 1 as the output until 100ns. We see the output go to 0 as A goes to 0 and S goes to 1 until 150ns. Then we see output go to 1 as we set the three inputs to 1 which results in the B and S portion yielding our 1 output since A is a dc value when the right side constantly yields true.

****

The below photo demonstrates the 4:1 mux which is based on 2:1 muxes which yielded comprehensive correct results earlier. We verified our design by writing a loop that checked various selection values while also setting different input values over 200ns intervals. Shown in the diagram below we have a select value of 3 and a D input value of 1. So, the output has selected the 3rd input as the result which is the value of D, so the output is 1 at 1,236ns. Further, we notice that select = 00 is the only time output A is selected which occurs from 200ns to 400ns. Also that B is selected for select = 01 which we see only results in the output being set from 500ns to 600ns and 700ns to 800ns. This trend continued for the other selected values corresponding to S = 10 and S = 11 corresponding to inputs C and D only selected at specific time intervals where the output is shown below.

****

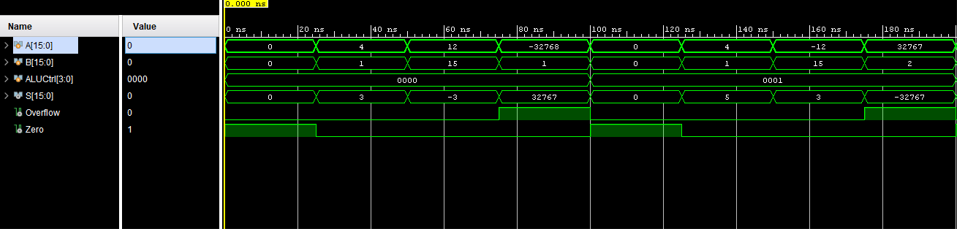
Very much similar to the above photo we again designed a 16:1 mux that used 4:1 muxes as its foundation. We again also used a loop to simply show that the select signal increments every 200ns, but only during the interval where select is set to a value does it pull the input value corresponding to that selected input. This functioned correctly for every selected value and is used several times throughout our ALU.

****

* 1. **ALU**

**Tests are written as such (A,B) = Out**

* + 1. **Subtraction: 0000, Addition: 0001**

****

Test 1: (0,0) = 0 , Also demonstrates zero flag is working correctly, Zero = 1

Test 2: (4,1) = 4-1 = 3, demonstrates correct positive result of sub

Test 3: (12,15) = 12-15 = -3, demonstrates correct negative result of sub

Test 4: (-32768,1) = -32768-1 = -32769 != 32767, shows overflow flag set since the result realistically should be more negative, but our result is a large positive.

**Subtraction Behaved Correctly**

Test 1: (0,0) = 0, demonstrates zero flag works in multiple modules

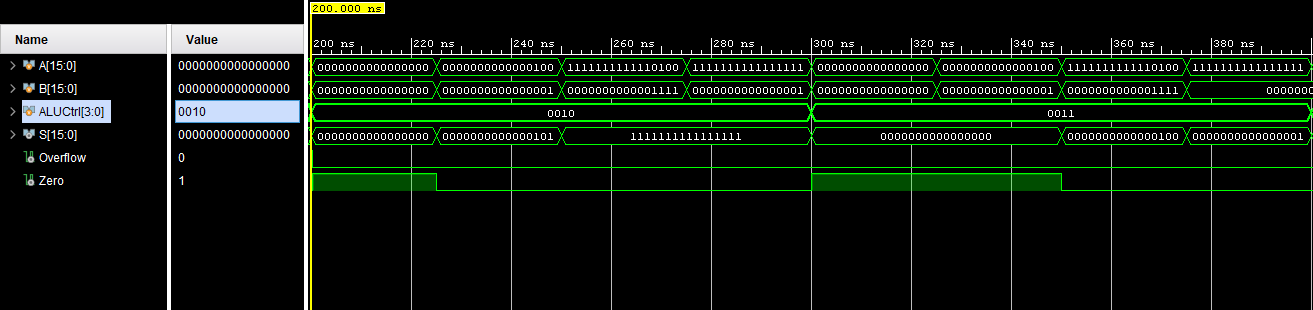
Test 2: (4,1) = 4+1 = 5, positive results in bit limits are correct

Test 3: (-12, 15) = -12 + 15 = 3, handles negative plus positive correctly

Test 4: (32767, 2) = 32767 + 2 = 32769 != -32767, demonstrates overflow. This result should be more positive, but instead becomes a large negative since it exceeds bounds.

**Addition Behaved Correctly**

* + 1. **OR: 0010, AND: 0011**

****

Test 1: (0|0) = 0 , zero flag works properly again, correct result

Test 2: (4|1) = 0101 = 5, results in any set bits being set in result

Test 3: (111…10100|00…01111) = 11…1 = -1, sets every bit to 1 even where a 0 and 1 occur in the same bit position.

Test 4: (11…1|00…01) = 11…1 = -1= Again sets all bit positions to 1 and enhances the verified attributes shown in case 3. OR is a setter method and by ORing two inputs we effectively set values in the result for any set input bit.

**OR Behaved Correctly**

Test 1: (0&0) = 0, demonstrates zero flag works for again

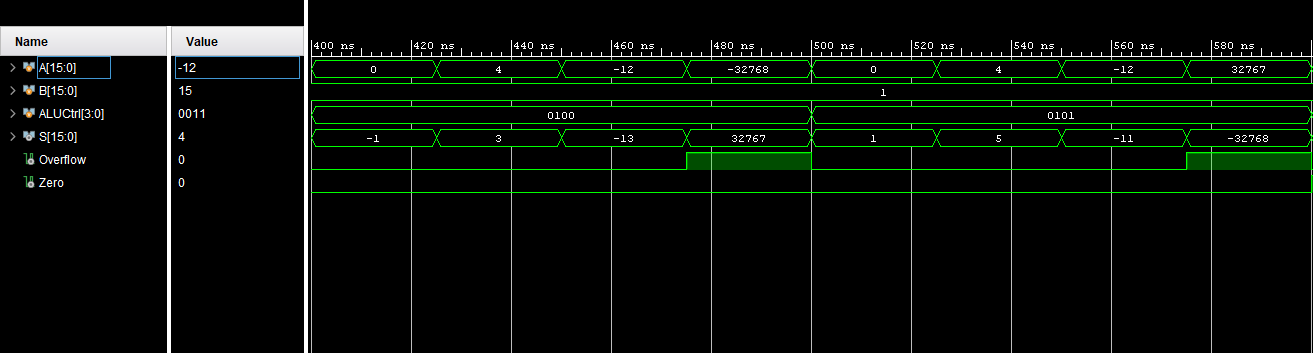
Test 2: (4&1) = 0, demonstrates zero flag after operation

Test 3: (11…10100&00…01111) = 0100 = 4, Shows that only the single set bit in both inputs is passed to output.

Test 4: (11…1&00…01) = -1&1 = 1, no need to show overflow since we did not pass bounds and this is the expected output. AND is an extraction method that eliminates bits who are not set in the respective bit positions in both inputs.

**AND Behaved Correctly**

* + 1. **Decrement: 0100, Increment: 0101**

****

Test 1: (0) = 0 -1 = -1 , successfully decremented base case

Test 2: (4) = 4-1 = 3, correct value for positive

Test 3: (-12) = -12-1 = -13, demonstrates correct result for a negative within bounds of bit range

Test 4: (-32768) = -32768-1 = -32769 != 32767, the realistic mathematical result is not achieved due to overflow. Instead of getting a more negative number we passed over the bit range bounds and yielded a positive number. Thus, overflow is set.

**Decrement Behaves Correctly**

Test 1: (0) = 0+1= 1, correct result base case

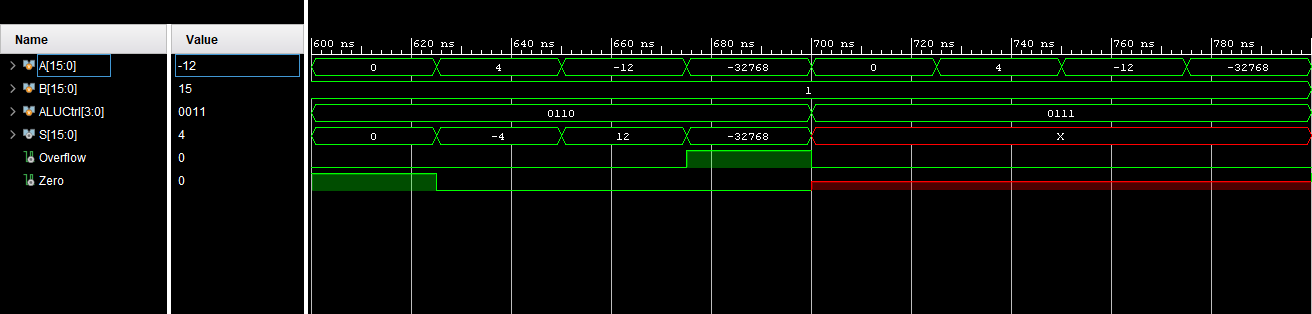
Test 2: (4) = 4+1 = 5, correct positive result within bit range

Test 3: (-12) = -12 + 1 = -11, correct increment on negative number

Test 4: (32767) = 32767 + 1 = 32768 != -32768, demonstrates overflow. This result should be more positive, but instead becomes a large negative since it exceeds bounds. Overflow is set.

**Increment Behaves Correctly**

* + 1. **Invert: 0110 , DC: 0111**

****

Test 1: (0) = ~0+1 = 0, correct base case

Test 2: (4) = ~4+1 = -4, demonstrates correct inversion for positive

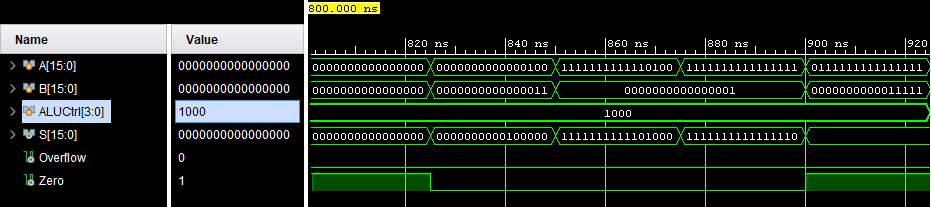
Test 3: (-12) = ~(-12)+1 = 12, demonstrates correct inversion for negative

Test 4: (-32768) = ~(-32768)+1 = 32769 != -32768, If we were to mathematically invert this value then it would become positive. However, the range of negative numbers in 2s complement is 1 index larger than positive numbers, so the most negative number (100…0) overflows when we negate it (011…1) and then add 1 , we get (100…0) back.

**Inversion Behaves Correctly**

Tests for Control = 0111 are dc.

* + 1. **Logical Shift Left: 1000 , SLTE: 1001**

****

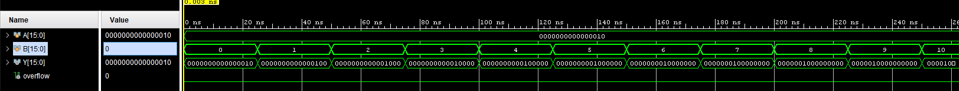
Test 1: (0,0) = 0<<1 = 0, correct base case , zero flag set

Test 2: (4,3) = 4<<3 = 32, correct value demonstrated for positive

Test 3: (-12,1) = (-12)<<1 = -24, demonstrates correct left shifting for negative

Test 4: (-1,1) = -2, demonstrates correct left shifting for another negative

Test 5: (32767, 32) = 0, demonstrates shifting out every value past the 16 bit range for the output.

****

Test 6: (2,0) = 2

Test 7: (2,1) = 4

Test 8: (2,2) = 8

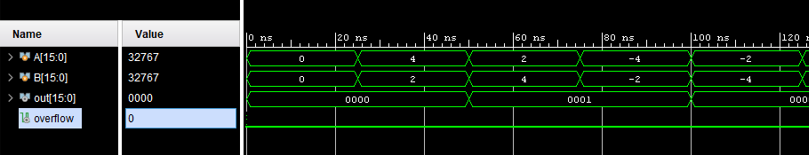
Test 9: (2,3) = 16

Test 10: (2,4) = 32

…

**Logical Shift Left Behaved Correctly**

* + 1. **SLTE: 1001**

****

Test 1: (0<=0) = 0, correct

Test 2: (4<=2) = 0, correct value demonstrated for positive

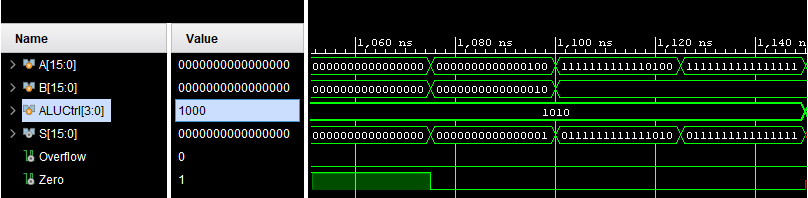
Test 3: (2<=4) = 1, correct a less then b

Test 4: (-4<=-2) = -2, correct a < b

Test 5: (-2<= 4) = 0, correct a < b

**SLTE Behaved Correctly**

* + 1. **Logical Shift Right: 1010, DC: 1011**

****

Test 1: (0>>0) = 0, correct, zero flag correct

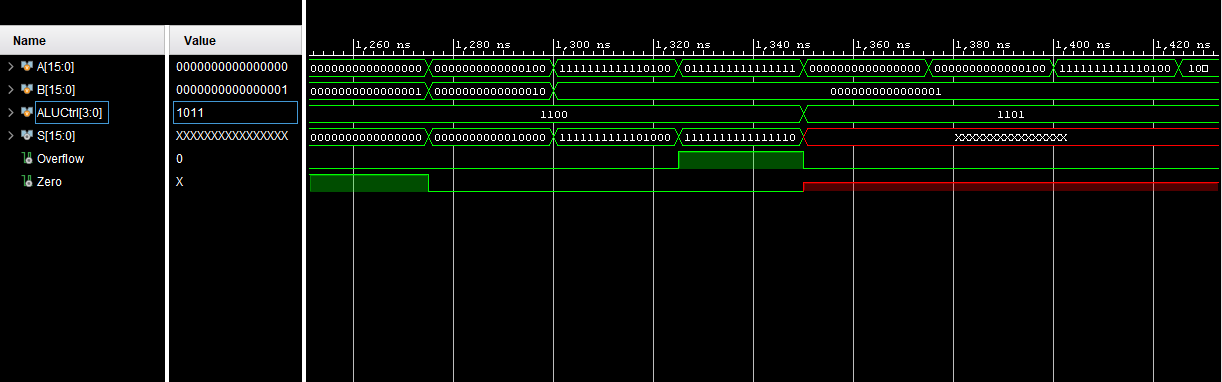
Test 2: (4>>2) = 1, correct value demonstrated for positive

Test 3: (-12>>1) = 32762, correct result arbitrary small negative to positive

Test 4: (-1>>1) = 32767, correct result small negative to most positive

**Logical Right Shift Behaved Correctly**

* + 1. **Arithmetic Shift Left: 1100 , DC: 1101**

****

Test 1: (0,1) = 0 , 0 left shifted by 1 is still 0. Effectively we are multiplying by 2 each shift. The Zero flag is appropriately set.

Test 2: (4,2) = 0…0100<<<0…0010 = 10000 = 16 , we took 4 and multiplied it by 2 twice. This results in 16 from left shifting. Correct value.

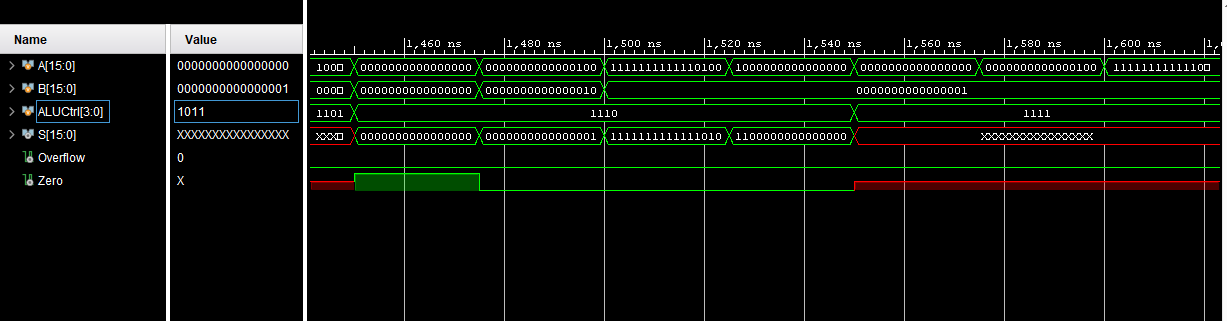
Test 3: (11…10100,00…01) = -12 <<<1 = 11…101000 = -24, we notice that there are now three zeros on the right bits of the output when there were only 2 zeros on the input A’s rightmost side. Thus, shifting left one added a zero on the right and multiplied by 2. Correct.

Test 4: (32767,1) = 32767<<<1 = 65534 != -2, shows overflow flag set since the result realistically should be twice the value of input A, but this exceeds 16 bit range for two’s complement numbers and becomes negative.

**Arithmetic Left Shift Behaved Correctly**

Test for Control = 1101 are DC values

* + 1. **Arithmetic Shift Right: 1110 ,DC: 1111**

****

Test 1: (0,0) = 0 >>> 0 = 0 , Shows again zero flag working

Test 2: (4,2) = 4>>>2 = 1, Correctly divides 4 by 2 twice, so 4/4 = 1 through shifting right

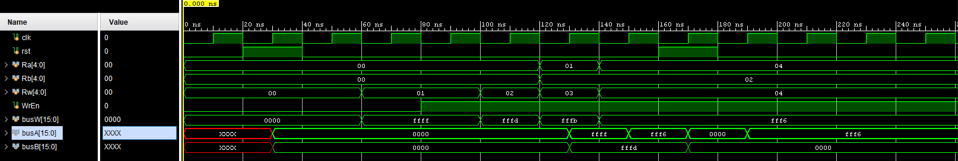
Test 3: (11…10100,1) = -12>>>1 = 11…1010 = -6, shows that -12 was divided by 2 and is now -6

Test 4: (-32768,1) = -32768>>>1 = -16,384, Correctly shifts the most negative number to the right by 1 and copies the MSB since the value remained negative

**Arithmetic Shift Right Behaved Correctly**

Test for Control = 1111 is DC values

* + 1. **Register File**

****

Test 1: 30ns: Upon reaching the positive edge of the clock and reset signal we rest the registers and values to 0 as prompted in the assignment

Test 2: 60ns: We see the Rw signal set to 1 and the busW has ffff in the buffer.

Test 3: 80ns: We see the Write Enable Signal is turned on, which allows us to start writing to register. We also see that over the next 20ns that the write register 1 will get the bus data of ffff and the register 2 will get the bus data of fffd

Test 4: 120ns: Beginning the read from Register 1 and 2 while writing to register 3: Ra is set to 1, Rb is still 2, Rw is set to 3. busW is set to -5. We see values updated 10ns later on the positive clock edge.

Test 4: 130ns: We see the busW is now holding fffb and the Rw is 03 so this register should be updated with that value. Since Rb is on register 2 we also see the busB is holding the value fffd as expected. The readings both occurred and we see an updated write at the end of the clock interval here.

Test 5: 140ns: We change the Ra to 4 and Rw to 4 as well as busW to -10. We are reading and writing to the same register which is register 4. Within the next 20ns we see that register 4 busA holds value ffff, while we have fff6 in Rw, and update Ra to fff6 within the clock cycle time frame. This effectively performs a concurrent read and write to the same register.

Test 6: 170ns: Showing that reset does reset values to 0 and takes priority again.

Test 7: 180ns: At the next clock rising edge we update the register for Ra to hold the value in the bus which is still fff6.

1. **Conclusion**
   1. **Lab Questions**
      1. **The difference between structural and behavioral Verilog**

Structural verilog describes functionality through connections. An example of this is the primitive gate level technology that handles Gate Level operations or modules on inputs and can yield the resulting output from those specific hardware elements. When coding using this setup the engineer is specifically telling the program how the connections and elements should be arranged since it is written explicitly.

Behavioral verilog describes functionality by behavior instead of structure. In more detail this is based on Input and Output responses and can generally include the ability of conditionals and non structural elements to yield such results. An engineer designing in this method is likely to see many optimizations and generally a different picture then what they would expect when evaluating the design synthesis for their code. This is because the code written is implicitly creating Muxes, Flip Flops, and other elements, but not stating exactly which connections should interact with one another aside from just being an input or output to some VHDL generated unit. In essence the system engine creates the structure based on the code provided.

A Behavioral example of a 2-1 multiplexer would have 3 input registers and one output. The select input value could be modeled as the condition for a if else statement where if select is set, then the input corresponding to select would be chosen. If select is not set then the first input not corresponding to select is chosen. The output would be assigned to be the result in the if case and the else case since only one will be chosen depending on select. This yields a design that would be system generated during RTL Synthesis based on our implicit coding of connections.

A Structural example would be like how we performed most of our code for this assignment. We would set up wires that represent the three inputs and 1 output of the 2-1 Multiplexer. However, to implement this mux we need to use the primitive gate technology available in VHDL. We created 4 more wires for this case to help us in achieving the correct result. We used the not(s\_not, S) gate hardware to create an inverted Select signal. We used the and(a1, A, s\_not) gate hardware to take in our first input (A) and the newly created inverted Select. We then used the and(b1, S, B) gate hardware to take in the original Select signal with our second input (B). Lastly we ran the output wires, a1 and b1, through the or(D, a1, b1) gate hardware to result in the output going through our final wire D. So, with a couple and gates, an inverter, and an or gate we were able to explicitly map out the connections and wire inputs and outputs to specific hardware technology to create a 2-1 multiplexer.

* + 1. **Difference between asynchronous and synchronous Mux**

Synchronous Multiplexers have an assigned fixed time slot to each connected device regardless of whether data is transmitted. Asynchronous Multiplexers have flexible time slots and are assigned to connected devices when they have data that is ready to be sent. Synchronous Muxes can suffer from the dead time spent whenever a time slot is not used, thus utilization is low and assigned time slots can be inefficient. Asynchronous intends to assign the time slots as needed to avoid making this an issue.

* + 1. **Difference between arithmetic and logical shifter**

The difference between the arithmetic and logical shifter mainly occurs in the right shift. When left shifting each bit is moved over n times to the left and the lowest n bits are filled with 0’s. This is the same for both arithmetic shift and logical shift. When dealing with signed numbers then left shifting can result in changing the sign of the number if the MSB is replaced with a 0 when it was originally 1 (Negative to Positive) or when a 1 in the MSB is replaced with a 0 (Positive to Negative). If the numbers are unsigned then the sign bit (MSB) is used to give the positive values roughly twice the range in exchange for no negative values.

When right shifting however there is an opportunity to fill with 1’s instead of 0’s. A logical right shift pads n 0’s on the left as we shift n times to the right. This could result in a negative becoming a positive if the MSB was 1, but we change it to zero while working with signed numbers two’s complement. If it is unsigned numbers, then the sign is traded for a higher possible range again even though we do not utilize it in this case by dividing through shifting by n 2’s. However, an arithmetic right shift aims to preserve the sign of the original number. This is done by copying the sign bit as the padded value which is padded from the left to the right n times when shifting. The result of an arithmetic right shift is a sign-preserved value that has been divided by 2^n where n is the shift value.

In summary, these two methods differ by sign preservation in the right shift and the bit that is copied into place and another copy shifted. In the left shift the behavior is identical.

* + 1. **Design of arithmetic shifter**

We completed the shifter using structural coding instead of behavioral. If we used behavioral coding it could have been simpler by utilizing the shifts operations or conditionals within our shift modules to handle overflow cases, shift amounts, and value setting. However, we constructed 16-1 Multiplexers where one module had a 1 bit output and the other had a 16 bit output. Through careful analysis, designing, selection, and coding we were able to handle each shifter structurally. Please note that there were some changes made to the code after demoing with the TA to implement overflow, and to handle shifts larger than 16 as shown in our code. These results were verified with the TA several times during week 4, Spring 2022.

Please see our images shown in the design section for further reference. However, to briefly summarize our designs here, we created 16 16:1 muxes where each mux handled selecting the correct original bit that would take the place of the final bit in the shifted output. For arithmetic left shifting the MSB code looked like this:

// Returns the shifted bit output for the MSB **(Note for ASL)**

mux\_16\_1\_input1bit mux15(

.X0(A[15]), .X1(A[14]), .X2(A[13]), .X3(A[12]),

.X4(A[11]), .X5(A[10]), .X6(A[9]), .X7(A[8]),

.X8(A[7]), .X9(A[6]), .X10(A[5]), .X11(A[4]),

.X12(A[3]), .X13(A[2]), .X14(A[1]), .X15(A[0]),

.S(shamt[3:0]), .OUT(Y[15])

);

…

// Returns the shifted bit output for the LSB

mux\_16\_1\_input1bit mux0(

.X0(A[0]), .X1(0), .X2(0), .X3(0),

.X4(0), .X5(0), .X6(0), .X7(0),

.X8(0), .X9(0), .X10(0), .X11(0),

.X12(0), .X13(0), .X14(0), .X15(0),

.S(shamt[3:0]), .OUT(Y[0])

);

We conceptualize that the Xs are wires that are selecting the mux inputs from x0 = 0 to x15 = 15. If we do not shift, then select is 0, which chooses X0, which holds the A[15] value. This is the original MSB bit unchanged. We pass this as the output to Y[15], our new shifted (in this case unshifted) value. However, if we shift 15 times we select X15 which holds A[0]. This results in the MSB for Y[15] being the LSB of the Original A input. By the time we reached mux0 which handled the LSB we already knew the pattern was that anything more than 1 shift would result in a 0 taking this bit position for the output. We just had to correctly attach wires to the correct inputs as designed above as we walked case by case in our drawings shown in the design section.

The Right shift was different in that we shifted from the left to right and effectively would copy the MSB as we shifted. So, whereas above we decreased the number of 0’s that would be in the mux that selected for LSB’s as we moved to MSB’s (more options of a shifted bit, instead of 0 value), in ASR we decrease the amount of A[15]’s we see in a mux as we move from mux15 to mux0. This is because the mux15 for the 15th bit is guaranteed to select A[15] , but depending on shifts right, a later mux may have the option of choosing A[14], or some other value. Take ASR mux13 for example. If the shift amount is greater than 2, then we know the 13th bit will be A[15].

// Returns the shifted bit output for the MSB

mux\_16\_1\_input1bit mux15(

.X0(A[15]), .X1(A[15]), .X2(A[15]), .X3(A[15]),

.X4(A[15]), .X5(A[15]), .X6(A[15]), .X7(A[15]),

.X8(A[15]), .X9(A[15]), .X10(A[15]), .X11(A[15]),

.X12(A[15]), .X13(A[15]), .X14(A[15]), .X15(A[15]),

.S(shamt[3:0]), .OUT(Y[15])

);

mux\_16\_1\_input1bit mux13(

.X0(A[13]), .X1(A[14]), .X2(A[15]), .X3(A[15]),

.X4(A[15]), .X5(A[15]), .X6(A[15]), .X7(A[15]),

.X8(A[15]), .X9(A[15]), .X10(A[15]), .X11(A[15]),

.X12(A[15]), .X13(A[15]), .X14(A[15]), .X15(A[15]),

.S(shamt[3:0]), .OUT(Y[13])

);

Note that we did include overflow logic by checking for sign switches as such in ASR:

// Overflow Detection

// Not signals

not(negA, A[15]);

not(negY, Y[15]);

//First Level ANDs

and(res1, A[15], negY);

and(res2, negA, Y[15]);

// Overflow Result for pos to neg or neg to pos case

or(overflow, res1, res2);

And we also had to account for the fact that a left shift could fully clear the bit field with all 0’s if the shift amount was greater than 15. So we implemented a slt module that would check if shift was less than 15.We used a final 16 bit mux to select 0 for the outputs if slt was 0 and to select Y for the output if slt result was 1, so we knew it wouldn't clear the buffer.

* 1. **Challenges**

Designing the 1 bit ALU and creating the first muxes was not very difficult, but required a bit of time to revisit some earlier concepts in logic design. This is likely due to having a couple of quarters between the first logic design class at UCLA that taught the basic modules and construction of them in various systems. Once we understood those and reviewed design material we were able to build a 4:1 mux from 2:1 muxes, and then build the 16:1 mux from the 4:1 muxes.

The next hurdle was designing the 1 bit ALU which asked us to use different inputs and operations. One issue was that we looked at the 16 bit ALU for inspiration and attempted subtraction for a while. We were just confused attempting this since it is generally addition, but inverting the second number. We discussed how 1 would differentiate from -1 using only 1 bit, but eventually we confirmed with the professor that subtracting with 1 bit values was unnecessary and beyond the scope of the task. We then proceeded to do AND, OR, Addition, and Inversion which was approved by the professor.

When beginning the 16 bit ALU we accelerated through the addition module and subtraction module. We started to question how we would do shifting, but decided to think about it on our own for another class session or two. The next class we handled most of the gate level tasks like AND, OR, Decrement, Increment, and Invert. We still had to finish up coding and testing on a couple of these, but were also still investigating how to perform shifting in our spare time. We had read a couple documents about shifting and eventually it started to click, but we wanted to design it by hand and verify the logic case by case to make sure it was correct before coding. This likely took us longer and was harder than for some of our other classmates since we did it completely Behaviorally. After drawing a schematic and analyzing the trends of a constant 0 shifting along bit positions in some cases, or neighboring bits in others, or the MSB in the case of ASR; we finally understood the shifter implementation, had some general code, and some logic to detect overflow as shown in our drawings in the Design section of this report.

We had not yet handled the case of clearing the bit field though for when B was greater than 15. The professor pointed this out during one of the demos for which we adjusted to code to do a SLTE comparison and set the resulting bits to 0 if the comparison came out to 0 and set the resulting bits to the shifted output of our Y 16 bit wire if the comparison evaluated to 1 (shift amount is less then or equal to 15).

Lastly we noticed that not all overflow cases were the same and that some things generally do not overflow. However, we also had overflow as a necessary output in each module. So, we created wires for zero and overflow and made sure that any module that did not need a overflow check was set to 0 and the results of overflow from appropriate modules with specific overflow logic code differing in some modules would then pass into a overflow 16:1 mux to correctly choose the right output depending on the Select signal. The zeros wire also checked for zero in each output for which we took the nor of each S bit with 0 and then passed into an and gate so that we could return 1, when S is 0.

* 1. **Contributions**

Both members were very involved throughout this whole lab. Chris and Rodrigo both spent a lot of time revisiting old logic designs and discussing how to proceed with each module. Rodrigo coded most of the Gatewise operations as well as the adder and subtractor. Chris worked outside of the classroom on the shifters to create the schematics shown above in the Design section which included initial code and overflow logic to go along with the 4 shifter modules. Both members worked on the remaining operations together. Chris gave a lot of input on how the testbenches should be designed to try and show various cases such as edge testing, triggering zero, and triggering overflow, as well as testing intermediary values to ensure the robustness of the designs. Rodrigo contributed a lot to the register file coding and worked with Chris on the testbench to accomplish testing the concurrent reads and writes. There were a couple of class sessions in this project where both team members were happy to stay a little longer to finish up something so we could make sure a module worked properly and was saved before going home. Even though the partners are rotating reports and generally focus on one or two tasks, we are always discussing the questions, debugging, and coding together. We thought this lab was a very extensive one that took quite a while to complete, but was very much a refresher for logic design and challenged the thought process extensively. It was a long, but positive experience for both partners.