Alberto Diaz - UID: 604967268

Evan He - UID: 705619333

Brandon Truong - UID: 705326387

Lab 1 Report

**Introduction**

In this lab, we were tasked with creating a 16-bit ALU using different forms of writing Verilog code. For the majority of the lab, we wrote our code using structural Verilog in which we implemented our ALU’s functionality by describing how our module is composed of simpler modules and basic logic gates. In some of the simpler modules, such as our 2-to-1 mux, we used behavioral Verilog which resembles more traditional programming that uses procedural statements such as “if” statements and “always” blocks. Our design was heavily inspired by our experience in CSM151B (Introduction to Computer Architecture) and the slides that were provided from that class.

We also implemented a Register File which is how processors keep memory locally. We decided to implement the RF using behavioral Verilog because we were more accustomed to using this design paradigm.

**Design Overview**

*two-to-one Mux:* Because the 2-to-1 mux is essential to the rest of the lab, we will discuss its design first. We were allowed to use behavioral Verilog to implement this, so we were able to use a simple “if” statement to implement it. Below is the entire module.

//Behavioral design using always block to handle. When sel is high, pick b, otherwise pick a

module two\_one\_mux\_behavorial(

input sel,

input a,

input b,

output reg x

);

always @\*

if(sel)

x = b;

else

x = a;

endmodule

We also wrote a 2-to-1 mux using structural Verilog:

//Structural design using gates to handle. When sel is high, pick b, otherwise pick a

module two\_one\_mux(

input sel,

input a,

input b,

output x

);

//logic is (not\_sel and a) or (sel and b), which when sel is 1, outputs b, otherwise outputs a

wire a\_res;

wire not\_sel;

not(not\_sel, sel);

and(a\_res, not\_sel, a);

wire b\_res;

and(b\_res, sel, b);

or(x, b\_res, a\_res);

endmodule

*one-bit ALU*: As described in the lab manual, we need to implement an AND gate, OR gate, NOT gate, 2-1 Mux, 3-1 Mux, and a Full Adder.

The AND, OR, and NOT gate are very simple to implement since Verilog already has these functions built-in. Generally, they take the form of and(*output, input1, input2*).

For the 2-1 Mux and 3-1 mux, we used the structural two-to-one mux that we implemented above. For a 3-1 mux, we need a 2-bit selector such that the first bit is used as the select bit on a 2-to-1 mux that feeds its output to a second 2-1 mux. On the second 2-1 mux, we used the second bit of the selector as the select bit. Generally, this is the pattern used for n-to-1 mux where n>2.

The addition operation has three inputs: a, b, and cin; and 2 outputs: sum and cout. To build it structurally, we first wrote out the truth table, then implemented it using logic gates. The full code can be found below:

//Module that adds two one-bits, along with cin (carry in), outputting cout (carry out) and sum (value of (cin+a+b)%2)

module addbit(

input cin,

input a,

input b,

output cout,

output sum

);

//Sum logic

wire a\_xor\_b;

xor(a\_xor\_b, a, b);

xor(sum, a\_xor\_b, cin);

//Carry out logic

wire a\_and\_b;

and(a\_and\_b, a, b);

wire and\_xor;

and(and\_xor, cin, a\_xor\_b);

or(cout, a\_and\_b, and\_xor);

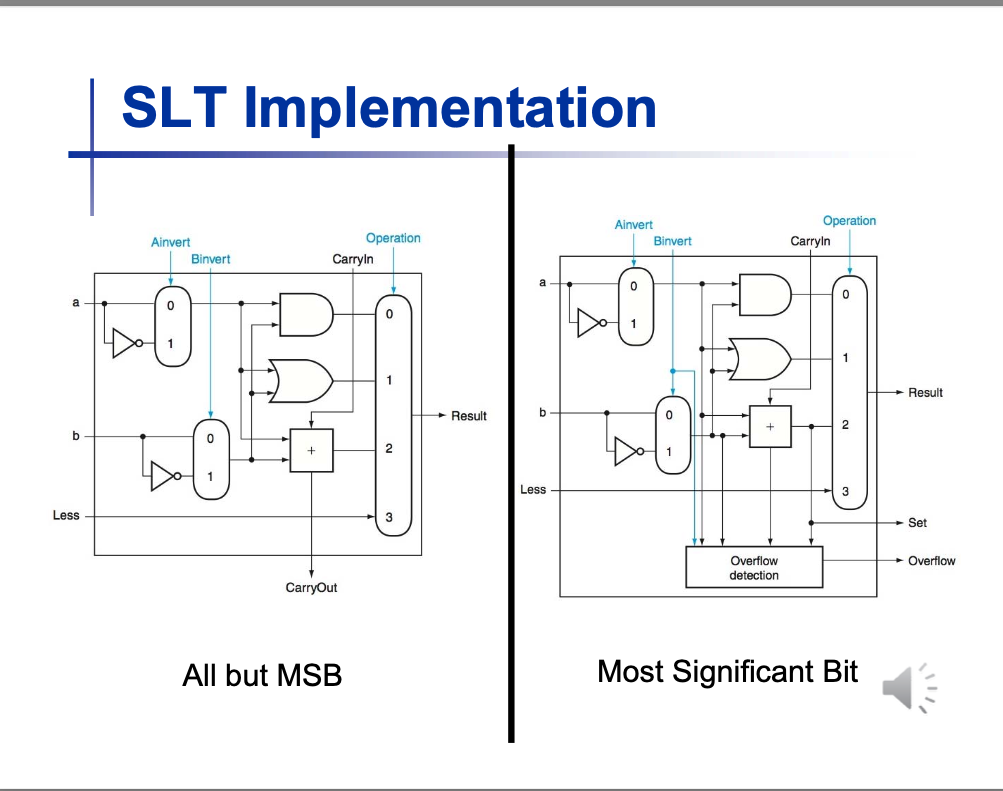
endmodule

The above module can also be used for subtraction so long as we flip one of the inputs by using a ‘not’ gate. Ideally, when an operation code is given, there will be a controller that flips input ‘b’ when the subtraction operation is chosen.

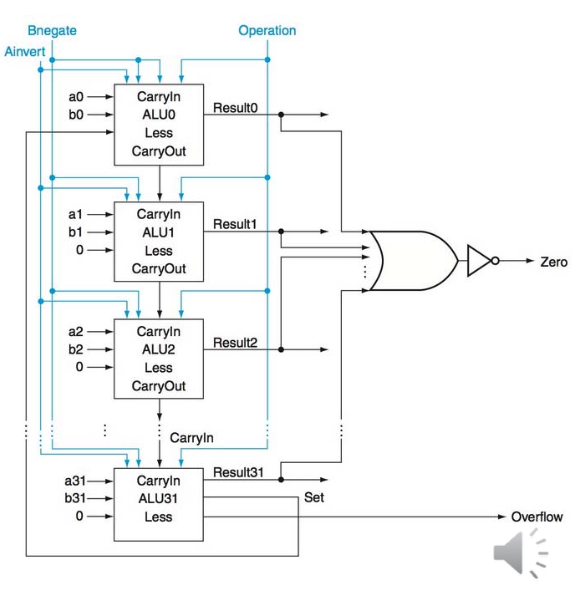
We chose the final output using a larger Mux to select which operation’s output should be given.

*16-bit ALU*

We based our implementation off of the MIPS implementation discussed in CSM151B taught by Professor Reinmann. We essentially chained together 16 one-bit ALUs to build the whole module. We also needed to create a Most Significant Bit implementation of the one-bit ALU to account for overflow and the “SLT or equal” operation. The following is a screenshot of one of the lecture slides in CSM151B:



Chained together, this is what our 16-bit ALU should look like (ours only goes up to result15):



We of course needed to make some changes to the design. For one, we needed to implement “SLT or equal” rather than just SLT. This meant needing to add extra logic to check if two inputs are equal. This was also the case for adding increment, decrement, invert, and four shifters. To deal with all of these additions, we implemented an ALU controller which is described below.

As with the one-bit ALU, we also needed a multi-level mux to decide which operation to output.

*ALU\_Control*

With so many operations needing to be supported, we noticed that we didn’t necessarily need to modify the one-bit ALU. Instead, we added a newer module called ALU\_control to cover some of the more specific cases. By outputting a unique string of bits, we are able to check whether we need to modify one of the inputs or do extra work such as checking equality of inputs and shifting.

The following is a list of these control strings and a brief description of what they mean:

// (flip a?, flip b?, op?, inc/dec?)

// if inc/dec then change ‘b’ to 1

// if flip a but not flip b then change ‘b’ to 0

// if flip a and b, do a shift.

localparam SUB = 5'b01100; // flip ‘b’ and perform addition

localparam ADD = 5'b00100; // flip neither and perform addition

localparam OR = 5'b00010; // flip neither and perform OR

localparam AND = 5'b00000; // flip neither and perform AND

localparam DEC = 5'b01101; // change ‘b’ to 1 and flip it, perform ADD

localparam INC = 5'b00101; // change ‘b’ to 1 and perform ADD

localparam INV = 5'b10100; // flip ‘a’, change ‘b’ to 0, and perform ADD

localparam ASL = 5'b11000; // shift, arithmetic left

localparam ASR = 5'b11010; // shift, arithmetic right

localparam LSL = 5'b11100; // shift, logical left

localparam LSR = 5'b11110; // shift, logical right

localparam SLT = 5'b01110; // flip ‘b’, do ADD, take SLT

The first two bits are used to instruct the ALU to flip ‘a’ or ‘b’ inputs. Typically we don’t want to flip ‘a’ except to perform an invert, so the flip ‘a’ bit is also used to check if we want to do a shift. The third and fourth bit are used as opcodes that are given to the one-bit ALU chain. The last bit is used to check for increment or decrement operations, in which case we want to tell our top module to modify ‘b’ to be equal to 1. This module is implemented using a series of 2-1 muxes that are specially designed to handle 5-bit inputs.

*Shifters*

Using the control bits from above, we can call a shifter module to perform any type of shift. We are quite familiar with string manipulation from our intro to CS classes, so we decided to implement this using behavioral Verilog. The following is the entire module for our shifters:

/\* Behavioral module that implements the corresponding shifts in 16-bit format, shifting a by b amount. Since left shifts are the same, we handle both cases together. Right shift arithmetic will use the most significant value to replace the other bits, and logical will use 0 to replace.\*/

module shifter(

input[15:0] a,

input[15:0] b,

input[1:0] shift\_control, // 00 arithmeitc left shift, 01 logical left shift, 10 arithmetic right shift, 11 logical right shfit

output reg[15:0] x

);

integer i;

integer j;

always @\*

begin

if (b > 0)

case (shift\_control)

2'b00, // arithmetic left

2'b10: // logical left

begin

for (i = 15; i >= b; i = i - 1)

x[i] = a[i - b];

for (j = b - 1; j >= 0; j = j - 1)

x[j] = 0;

end

2'b01: // arithmetic right

begin

for (i = b; i <= 15; i = i + 1)

x[i - b] = a[i];

for (j = 15 - b + 1; j <= 15; j = j + 1)

x[j] = a[15];

end

2'b11: // logical right

begin

for (i = b; i <= 15; i = i + 1)

x[i - b] = a[i];

for (j = 15 - b + 1; j <= 15; j = j + 1)

x[j] = 0;

end

endcase

end

endmodule

*Register File*

The RF was relatively easy to implement using behavioral Verilog especially because the behavior was given in the spec. The more interesting part about this module was the need to use blocking assignment rather than non-blocking. This is because our RF is required to reflect new register values when there are concurrent reads and writes, so we want our assignments to happen in specific order. The following is our entire module for the RF:

/\* Structural module that reads and writes to 32 16-bit registers. Reset will reset all registers (prioritizing reset over write enable), otherwise, it will write busW to the register Rw, and read registers Ra and Rb into busA and busB respectively \*/

module register\_file(

input clk,

input rst,

input[4:0] Ra,

input[4:0] Rb,

input[4:0] Rw,

input WrEn,

input[15:0] busW,

output reg[15:0] busA,

output reg[15:0] busB

);

reg[15:0] registers [31:0];

integer i;

always @ (posedge clk)

begin

if (rst)

//reset all registers

for (i = 0; i < 32; i = i + 1)

registers[i] = 0;

else if (WrEn)

//write to register specified by Rw

registers[Rw] = busW;

//outputs are linked to corresponding registers

busA = registers[Ra];

busB = registers[Rb];

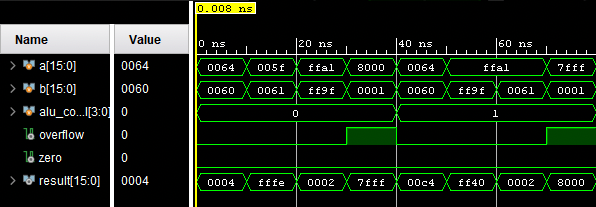
end

endmodule

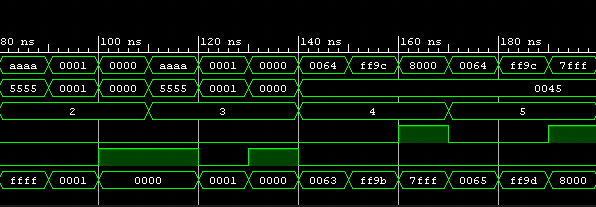
**Results**

Figure 1 displays our results for the 16-bit ALU. We include at least three examples of each operation: an arbitrary example, an edge case with large input, and a case that should result in overflow (if applicable).

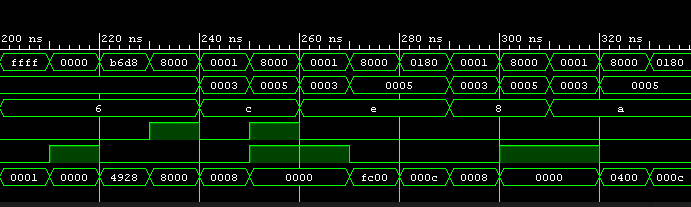
**Figure 1:** Results for 16-bit ALU with at least three examples for each operation:



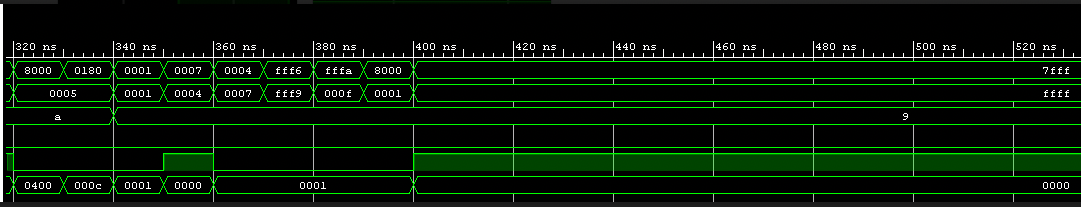
(a) Results for subtraction and addition operations



(b) Results for Bitwise OR, Bitwise AND, Increment, and Decrement operations



(c) results for invert, arithmetic shift left, arithmetic shift right, logical shift left, and logical shift right.



(d) results for “set on less than or equal to”

**Problems and Solutions**

One issue that arose was how to deal with the “zero” output. At first, we implemented our 16-bit ALU by modeling it after the MIPS ALU that includes the “set less” operation. When we tried to extend our ALU to have “set less than or equal to”, we attempted to use the “zero” output that was set by a subtraction operation. If “zero” was set to high, we would output 1 on the “set less than or equal to” operation. Doing this however created a circular dependency between “SLT or equal to” and “zero” which caused problems for the simulation. To fix this issue, we included extra logic specifically to check if two inputs are equal so that our “SLT or equal” does not depend on “zero” anymore.

Another interesting problem was building an ALU controller from the given select bits. For the most part, this ALU controller kept getting modified as some operations were similar to others. For example, our “SLT or equal” operation used the output of a subtraction operation to see if one input was less than the other. In cases where we are executing an “SLT or equal” operation however, we do not want to overflow if the subtraction operation overflows, so we had to use an extra control bit in the ALU control to clearly differentiate between “SLT or equal” and subtraction. Similar solutions were needed to implement the increment and decrement operations to change ‘b’ to 1 or -1, and to fix some issues with shifting.

**Answers to Lab Questions**

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

In structural Verilog, the programmer’s main objective is to describe the functionality of their module using smaller modules or basic primitives like logic gates and transistors. In behavioral Verilog, the programmer may use procedural statements that are common in other programming languages. These procedural statements may include “if” statements, or even “always” blocks just to name a few examples.

The following is our behavioral implementation of a 2-to-1 mux:

//Behavioral design using always block to handle. When sel is high, pick b, otherwise pick a

module two\_one\_mux\_behavorial(

input sel,

input a,

input b,

output reg x

);

always @\*

if(sel)

x = b;

else

x = a;

endmodule

Structural implementation of a 2-to-1 mux:

//Structural design using gates to handle. When sel is high, pick b, otherwise

//pick a

module two\_one\_mux(

input sel,

input a,

input b,

output x

);

//logic is (not\_sel and a) or (sel and b), which when sel is 1,

//outputs b, otherwise outputs a

wire a\_res;

wire not\_sel;

not(not\_sel, sel);

and(a\_res, not\_sel, a);

wire b\_res;

and(b\_res, sel, b);

or(x, b\_res, a\_res);

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.*

The main difference between asynchronous and synchronous mux is that asynchronous will calculate a result at all times whereas synchronous will rely on an external clock to determine when an output should be calculated.

The behavioral multiplexer above is an asynchronous mux because it does not rely on an external clock. To implement as a synchronous mux, we would need to add a clock input and change the “always @\*” to instead be “always @ (posedge clock)” or whichever part of the clock is desirable for the application.

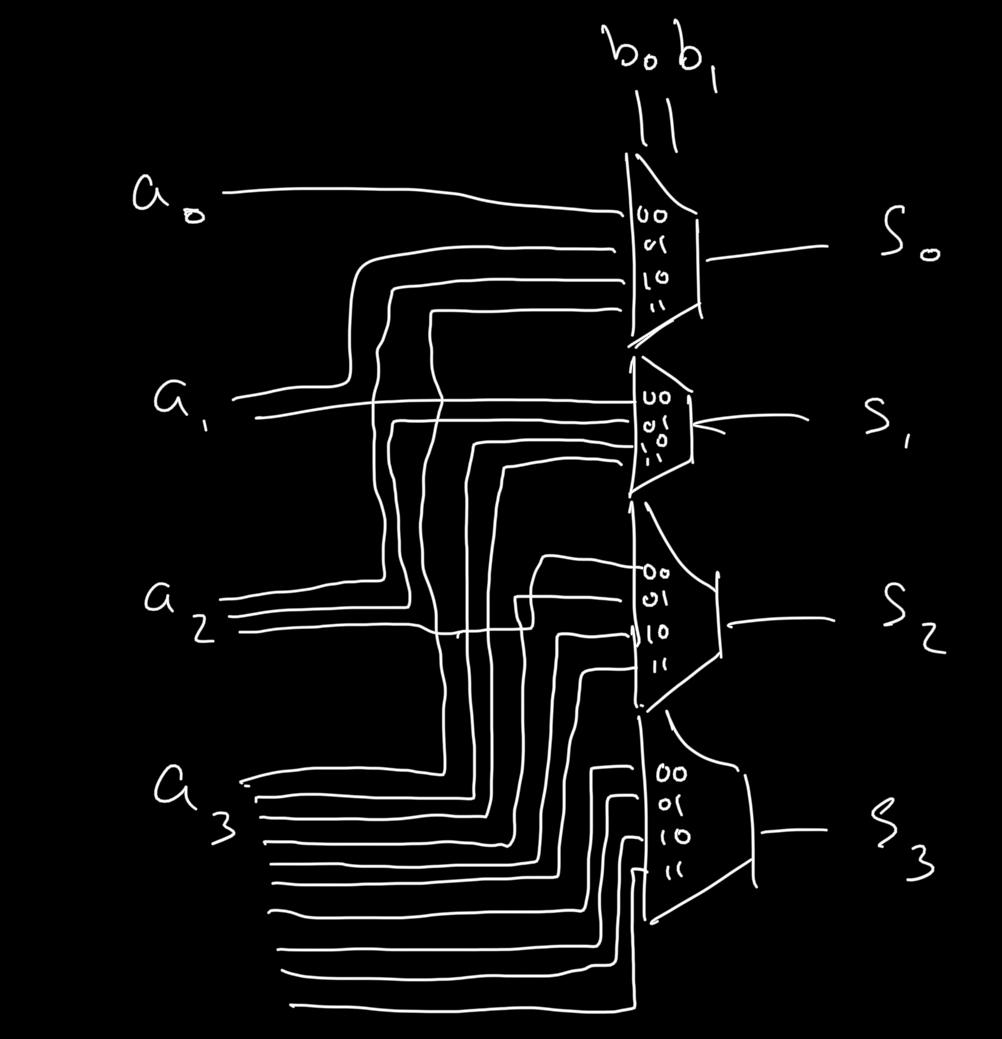
*3) What is the difference between an arithmetic and logical shifter?*

Simply put, an arithmetic shifter treats an input as a signed integer, meaning that it preserves the most significant bit. A logical shifter on the other hand, simply treats the input as a string of bits and fills in with 0’s. An arithmetic shifter is typically more complex because of the need to preserve the sign.

*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.*

Our behavioral implementations of all shifters is above. In those implementations, we use our experience in string manipulation to shift the bits accordingly.

A structural implementation would use multiple multiplexers to decide which input bit gets assigned to which output bits. Figure 2 is a simple diagram showcasing how arithmetic shifting works for a 4-bit input.



**Figure 2:** Simple diagram of arithmetic shifter using multiplexers on a 4-bit input. The 2-to-1 muxes required to implement 4-to-1 mux are abstracted away.

**Contributions**

Alberto - Contributed equally to the design, debugging, and simulation of this lab. Provided documentation from previous UCLA CS courses to inspire ALU design. Typed up answers to lab questions.

Evan - Contributed equally to the design and simulation of this lab. Split most of the manual labor of typing in our code with Brandon.

Brandon - Contributed equally to the design and simulation of this lab. Split most of the manual labor of typing in our code with Evan.