Comp Arch: Lab 1
10/1/2018
Vicky McDermott
Minju Kang
Allison Basore

Comp Arch: Lab 1

For this lab, we built an Arithmetic Logic Unit (ALU). We did this by building a bitslice module which is able to perform the specified operations including ADD, SUB, XOR, SLT, AND, NAND, NOR, and OR and then instantiating 32 of these. Each bitslice sets the bit value for a given bit and the carryout value of one bitslice is input into the next bit. We determine which operation to perform on each bit using a combination of a lookup table and a multiplexer. The command is input into the lookup table and then a 2-bit address for the multiplexer is output as well as a value for invert.

Simulation

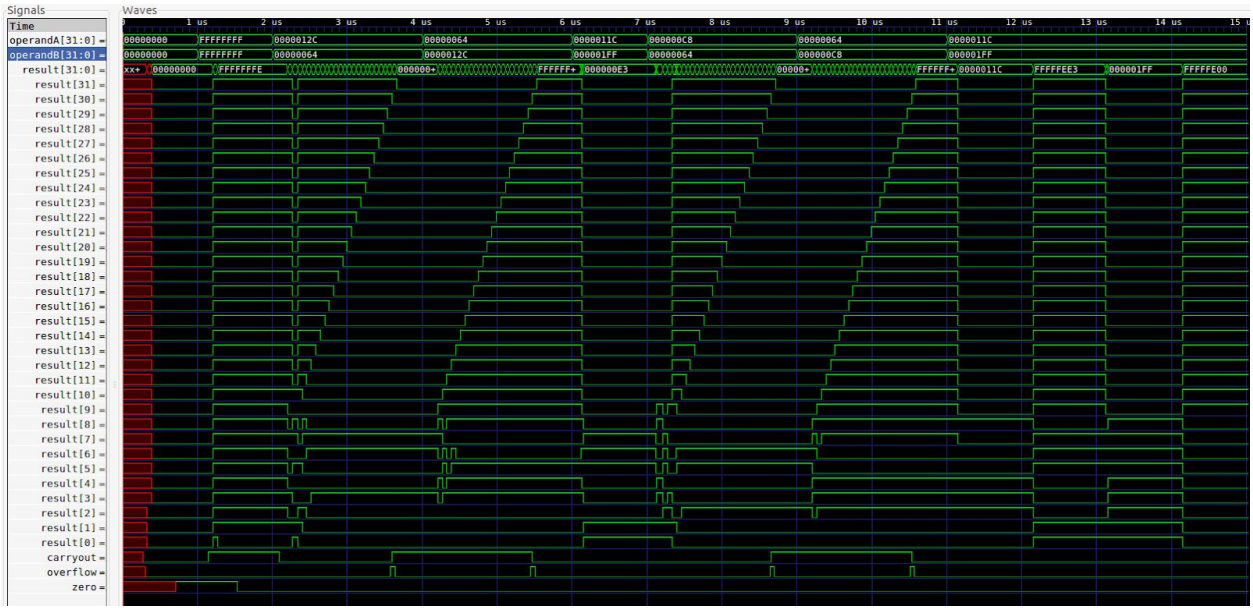

Figure 1: Waveform Output

We can clearly see the propagation delays of our ALU in the waveforms in Figure 1 above. The value of result[0] is set before the values of result[1], result[2], etc. that follow it. This is due to the setup of our bitslice module which calculates one bit at a time and moves from bit 0 (result[0]) to the most significant bit (result[31]) going through several gates at each step along the way.

Circuit Overview

Each Bitslice Has:

<Adder & Subtractor>

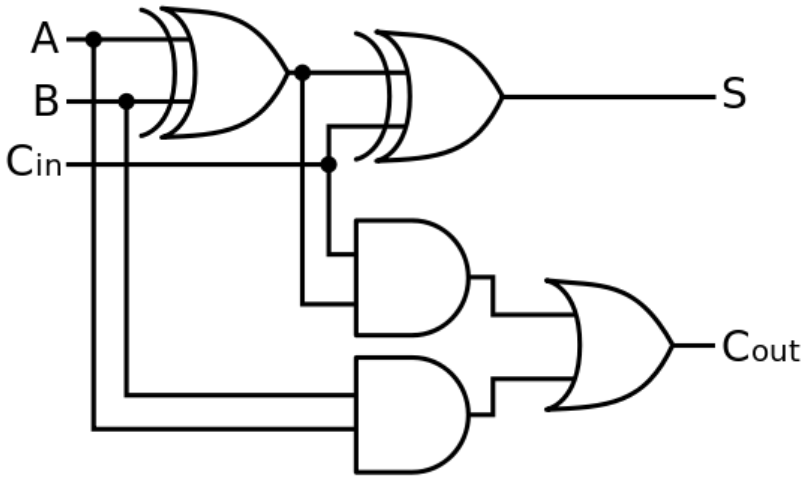

Figure 2: Full Adder diagram

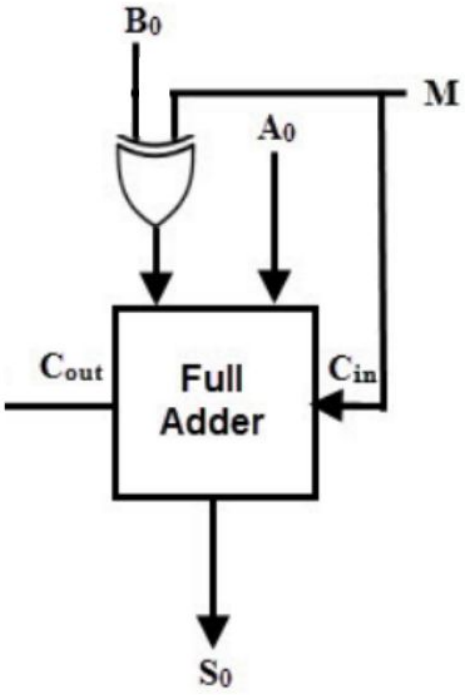

Figure 3: 1 bit Adder and Subtractor integrated diagram

Comp Arch: Lab 1
10/1/2018
Vicky McDermott
Minju Kang
Allison Basore

For the full adder, we used the same design as we used before in Lab 0. Then, we built the subtractor by attaching XOR gates as one of the inputs to the adder. In that way, we could treat the subtraction as "adding two negative numbers".

<Set Less Than>

For the set less than operation, we just checked the most significant bit's value of the result of the subtractor. If A < B, A - B < 0. Therefore, the most significant bit will be 1 since the result of the subtractor is signed. If A > B, A - B > 0 and the most significant bit of the subtractor's signed result will be 0. Therefore, we subtracted B from A and set the result[31] to be true if the most significant bit was 1 and false otherwise.

<Boolean Operation>

- AND
- NAND
- OR
- NOR
- XOR

For the boolean operations above, we simply passed two inputs to corresponding gates for and, or, and xor. We used xor on the value of invert (set from the lookup table) and the result of and/or to get the value for nand/nor. We then passed the values for and/nand, or/nor, xor, and subtraction/addition/slt to a multiplexer which determined which of these results to use based on the input command.

Then, we connected 32-bit slices:

First, we used a LUT (Look Up Table) to determine what arithmetic operation we want to perform. The LUT outputs a value for address 0 and address 1 which correspond to a given operation and a value for invert which indicates whether or not to invert the result. A value of (1,1) indicates that that the ALU should perform the xor operation. A value of (0,1) indicates that the ALU should perform the and operation (or nand if invert is true). A value of (1,0) indicates that the ALU should perform the or operation (or nor if invert is true). Lastly, a value of (0,0) indicates that the ALU should perform addition (or subtraction and set less than if invert is true). Then, the appropriate 4 values (inverted if invert is true and the original value otherwise) are input into a 4 input multiplexer with the address 0 and address 1 values described above to output the appropriate result for a given bit.

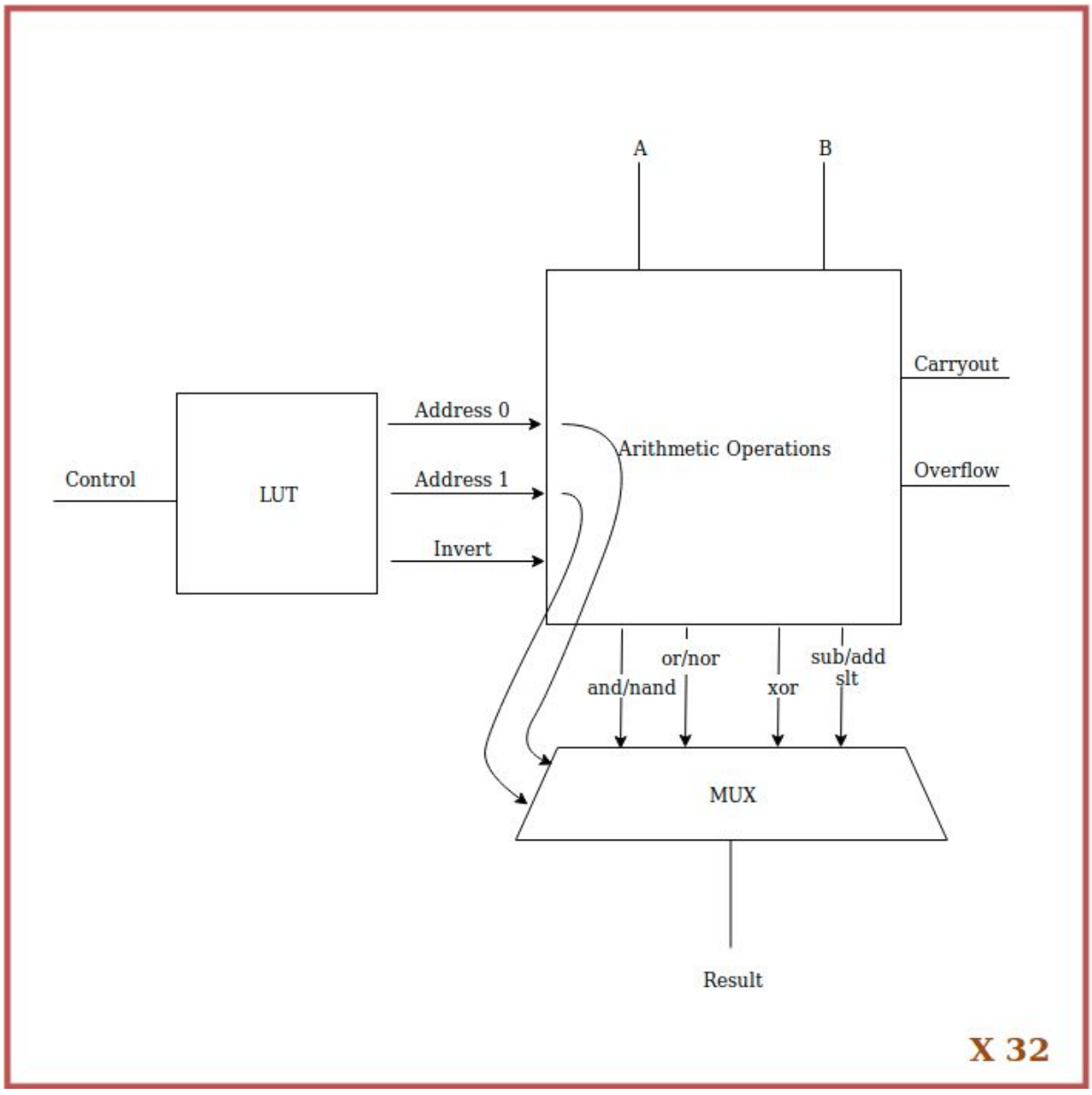

Figure 4: System Diagram of ALU

Operation	Result	Sets flags?	ALU Control
ADD	`R=A+B`	Yes	`b000`
SUB	`R=A-B`	Yes	`b001`
XOR	`R=A^B`	No	`b010`
SLT	`R=(A<B)?1:0`	No	`b011`
AND	`R=A&B`	No	`b100`
NAND	`R=~(A&B)`	No	`b101`
NOR	`R=~(A|B)`	No	`b110`
OR	`R=A|B`	No	`b111`

Figure 5: Control Signal Table with the corresponding arithmetic operation

Test Results

We made several tests to find potential errors in our bit slices and overall ALU. We tested the bitslice nearly exhaustively with each of the right functions for a= 1 and 0, b = 1 and 0, and carryIn = 1 and 0. These bitslice tests gave us lots of insight into problems with our bitslice.

On the first attempt, the following function worked correctly: and, or, xor, nor, nand, add. At first, subtract was not producing the correct carryOut value. We found that this was because we had mixed up a variable with the logic gates and were not always inverting the value of b when we should have been.

When we moved on to the overall ALU, which was comprised of 32 bitslices, we also had the following functionalities working on the first attempt (after a few indexing bugs): and, or, xor, nor, nand, add. The SLT function was not working at first because we were not inverting b when the SLT command was used. The subtract functionality did not work at first because we didn't have sufficient delay before we checked the result of the test cases. The operation worked with a single bitslice but didn't with 32 bit ALU module. We realized that this was because the propagation delay for the calculation the ALU was performing was longer than amount of delay we had included in our test bench before checking the result. As a result, we had incorrect checked values for several of the most significant digits (result[31], result[30], etc.) since we were checking the result before it had even been set. We fixed this error by adding a longer delay before checking the result of the ALU operation.

For the ALU, we did not test exhaustively since there are far too many test cases to create in our timeframe. Instead, we tested several specific cases for each of the functionalities. Below is a table of the tests we ran and the reasoning behind them. We wanted to test our basic functionality for the and, or, xor,

nand, nor gates because they did not deal with overflow and carryout. For add and subtract, we made more advanced tests to try multiple cases.

A	B	Result	Function	Reason
32'd0	32'd0	32'd0	ADD	A basic test of the add, carryout, and overflow result
32`b111….111	32`b111….111	32`b111….110	ADD	An advanced test of add, carryout, and overflow
32'd300	32'd100	32'd200	SUB	A basic test of the sub, carryout, and overflow result
32'd100	32'd300	32'd200	SUB	An advanced test of sub, carryout, and overflow
32'b100011100	32'b111111111	32'b011100011	XOR	Basic functionality
32'd200	32'd100	32'b1	SLT	Basic functionality
b100011100	b111111111	32'b100011100	AND	Basic functionality
32'b100011100	32'b111111111	32'b11011100011	NAND	Basic functionality
32'b100011100	32'b111111111	32'b111111111	OR	Basic functionality
32'b100011100	32'b111111111	32'b1...1000000000	NOR	Basic functionality

Comp Arch: Lab 1
10/1/2018
Vicky McDermott
Minju Kang
Allison Basore

Gate Delays and Timing Analysis

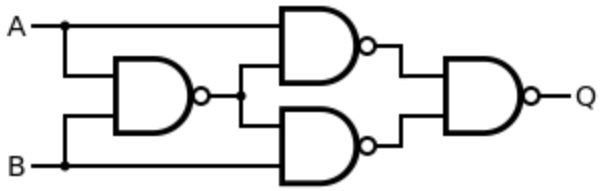

Figure 6: XOR gate implementation with using 4 NAND gates

Since gates such as AND, OR, XOR are not base gates, (they have hidden inverter gates) we had to treat these gates differently than the base gates NAND, NOR, and NOT. AND and OR gates have inverters attached to NAND and NOR gates. Therefore, they have the propagation delay of 30 time units (20 from a 2 input NAND or 2 input NOR and 10 from NOT). The XOR gate can be implemented as figure 6 with the propagation delay of 60. The largest amount of gates you can have in figure 6 is 3, therefore you can get 60 units of propagation delay after multiplying 3 by 20, which is the propagation delay of a 2 input NAND gate. When we add up all of the gates that the inputs in our ALU travel through, we get a total worst case propagation delay of 30,930 microseconds. Because we did not do our operations as separate modules and instead did them all as part of one bitslice, the worst case propagation delay will be the same for all operations. The gates in the general ALU portion (performed only once) account for 530 microseconds and each bitslice contributes a delay of 950 microseconds.

Work Plan Reflection

We originally planned to be done with our ALU implementation by Wednesday, but we ran into many different bugs along the way that slowed down our process. Therefore, we were not able to completely finish our ALU implementation until Thursday afternoon. In general, our work plan was a good way for us to organize our tasks and get the work done in a timely manner, even if it was not quite as early as we had originally planned.

Conclusion

We believe the ALU subsystem design is ready to be included in the CPU only if several flaws are handled. Currently, the operation SLT does not yield error or corrected value for overflowed values. The ALU unit doesn't think it is wrong, therefore it gives the value without any error but still, with overflow = 1, the result is not reliable. The bitslice submodule has been nearly exhaustively tested. The ALU has also been selectively tested. The tests have all passed with the correct expected results that we calculated manually. Therefore, we concluded that what we built can be called Arithmetic Logic Unit since our circuit functions as same as conventional ALU.