Comp Arch Lab 1: Arithmetic Logic Unit

Zarin Bhuiyan, David Zhu, Bonnie Ishiguro

October 6, 2016

Implementation

We implemented the ADD operation the same way we implemented it for our full adder. We re-purposed our original adder into an adder-subtractor by creating an "isSub" flag. We set "isSub" to 0 or 1 based on the operation's 3-bit command, using an 8-bit multiplexer. If "isSub" is true, we add the complement of operandB to operandA instead. We use this "isSub" flag later on to add 1 to the least significant bit of our 32-bit result by passing it in as the first carryin. The inputs to the adder-subtractor are: operandA, operandB, the carryin, and "ifsub", and the outputs are: result, carryout, zero, and overflow.

We implemented AND, NAND, OR, NOR, and XOR with Verilog's built-in logic gates and two inputs, operandA and operandB.

We soon realized that since we implemented a bitslice design for our ALU, we needed to postpone the SLT calculation until after all 32 bit operations were complete. We therefore replaced our final result output with a preliminary result called internalResult that we could manipulate before sending to output. We created an "isSLT" flag and used a 2-input multiplexer to choose between returning either the most significant bit and 31 "0"s if the flag is set to true, or the original calculated result otherwise. Although we had originally designed our ALU without a look up table, we ended up incorporating one into our design so that we could reuse the subtraction module to implement SLT.

To calculate our zero flag, we passed our calculated result into a 32 bit NOR. To do this, we implemented four layers of recursive for loops to OR the 32 bits. Since the result of this OR returns 0 if the result is comprised of 32 "0" bits and 1 otherwise, we set the "zero" flag to be the NOT of this result. We implemented "zero" and "SLT" last, and we found it very difficult to incorporate them into our pre-existing mux and bitslice design. In retrospect, we wish we had designed these components earlier, which would hopefully have led us to implement a more efficient ALU.

One of the major design issues with our ALU implementation was the choice of using an 8-input multiplexer as opposed to additional flags from the initial lookup table. Because of this implementation, we had to interpret the 3 bit ‘command‘ signal at multiple places. Each of our ALU bitslices took in the ‘command‘ signal and had to determine what to do then.

This caused a cascading issue where addition and subtraction can no longer be combined into one module. Instead, we duplicated them, and passed in a hard flag for ifsub. After that, we had to include additional 8 input multiplexers just to get carryout and overflows to be from the correct ‘command‘ channels. These duplications were definitely not space efficient as opposed to if we took more advantage of the starting lookup table flags.

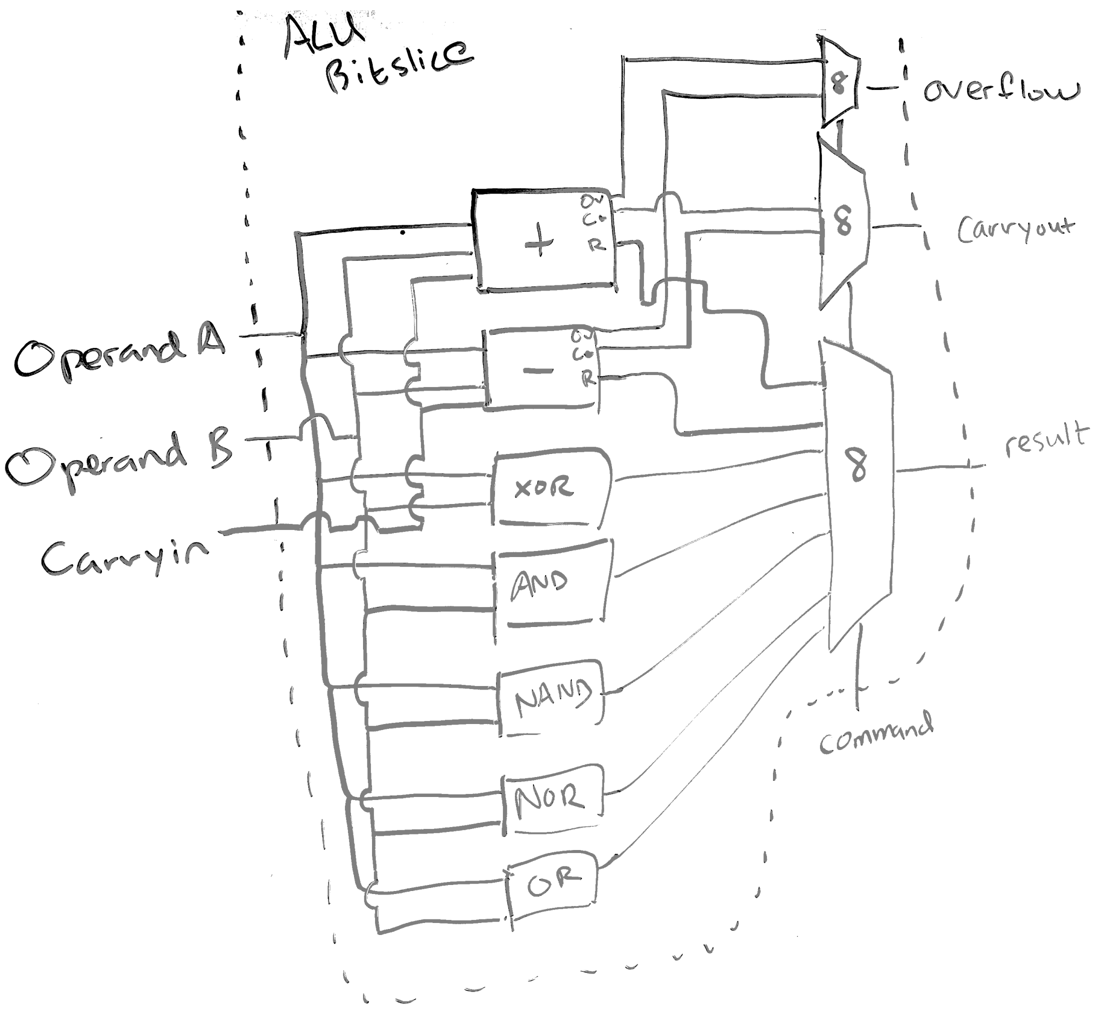

Figure 1: This figure shows how our ALU bit slices work. There are 7 operations that are done on three inputs: operandA, operandB, and carryin. The operations output the result, carryout, and the overflow in each case.

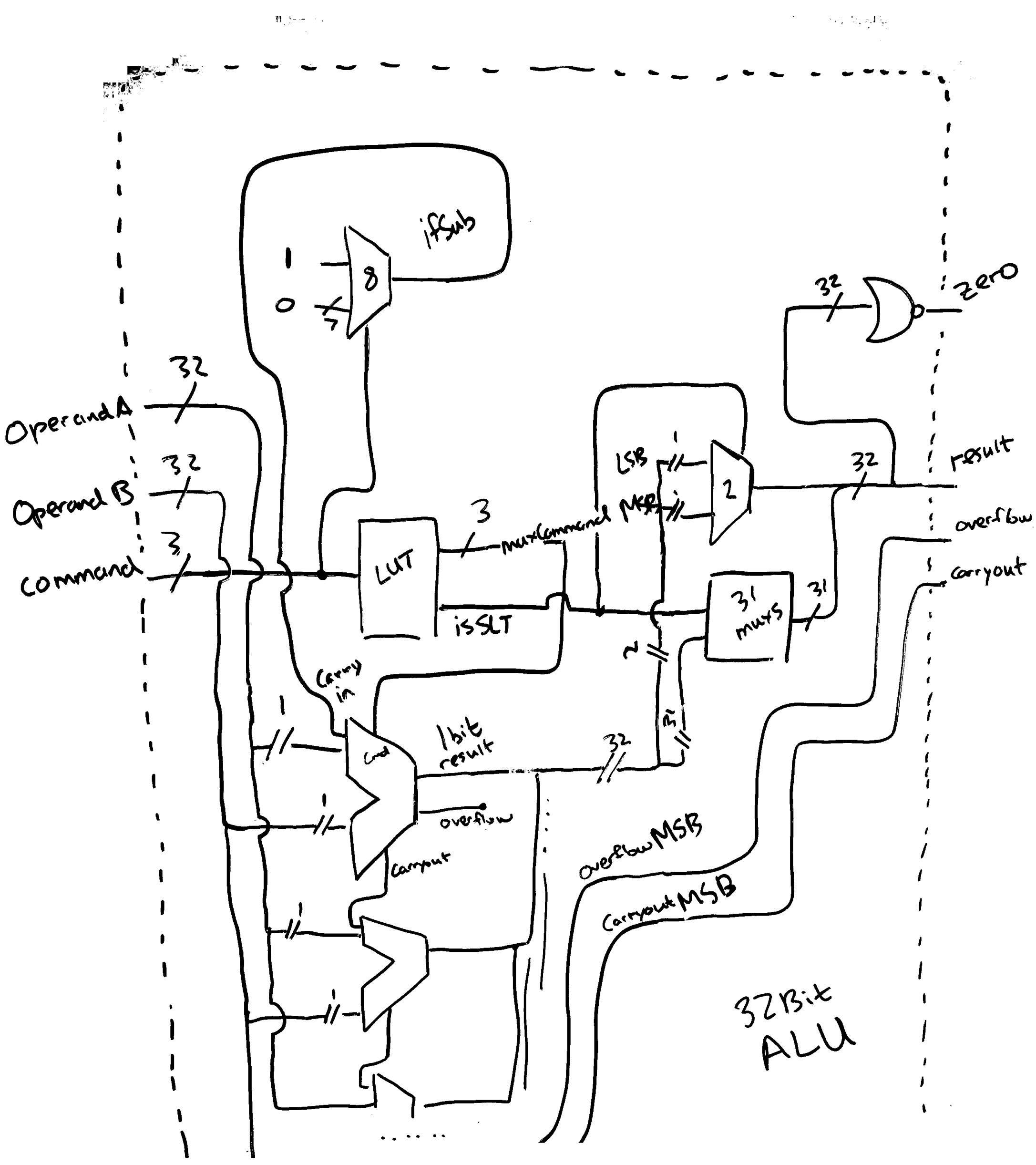

Figure 2: This figure shows how the individual bit slices are hooked up within the ALU. Outside the ALU, we show our LUT, how the adder-subtractor chooses its operation, how the SLT is calculated, and how the zero flag is calculated.

Test Results

The table below displays the tests that we chose for our test bench for all of our operations.

To validate our ADD function, we chose to test it with two positive operands, two negative operands, a negative operand and a positive operand, an addition with a carryout and overflow, and an addition that sums to zero. Our original tests led us to find that we had flipped the 32 bits of the result that we fed into the output.

opName	A	B	Result	Carryout	Zero	Overflow
ADD	32'd1	32'd2	3	0	0	0
ADD	32'hFF	32'hF9	hF8	0	0	0
ADD	32'hFF	32'h02	101	0	0	0
ADD	32'h80000001	32'h80000001	h80000002	1	0	1
ADD	32'b0	32'b0	0	0	1	0
SUB	32'd3	32'd2	1	1	0	0
SUB	32'hFFFFFFFE	32'd3	FFFFFFFB	1	0	0
SUB	32'd3	32'd3	0	1	1	0
SUB	32'hFFFFFFFC	32'hFFFFFFFC	0	1	1	0
SUB	32'h80000001	32'h00000001	h80000000	1	0	0
SUB	32'b1	32'b1	0	1	1	0
XOR	32'b0	32'b0	0	0	1	0
XOR	32'b0	32'b1	1	0	0	0
XOR	32'b1	32'b0	1	0	0	0
XOR	32'b1	32'b1	0	0	1	0
SLT	32'd2	32'd3	1	0	0	0
SLT	32'd3	32'd2	0	1	1	0
SLT	2'hFF	32'h01	0	1	1	0
SLT	32'h01	2'hFF	1	0	0	0
AND	32'd0	32'd0	0	0	1	0
AND	32'd0	32'd1	0	0	1	0
AND	32'd1	32'd0	0	0	1	0
AND	32'd1	32'd1	1	0	0	0
NAND	32'h0	32'h0	1	0	0	0
NAND	32'h0	32'hFFFFFFFF	1	0	0	0
NAND	32'hFFFFFFFF	32'h0	1	0	0	0
NAND	32'hFFFFFFFF	32'hFFFFFFFF	0	0	1	0
NOR	32'h0	32'h0	1	0	0	0
NOR	32'h0	32'hFFFFFFFF	0	0	1	0
NOR	32'hFFFFFFFF	32'h0	0	0	1	0
NOR	32'hFFFFFFFF	32'hFFFFFFFF	0	0	1	0
OR	32'b0	32'b0	0	0	1	0
OR	32'b0	32'b1	1	0	0	0
OR	32'b1	32'b0	1	0	0	0
OR	32'b1	32'b1	1	0	0	0

Because we were getting confused by our 32-bit test file results for the ADD and SUB operations, we wrote a separate exhaustive test bench for them to catch any flaws. We realized from this that the order of the inputs of our ALU bit slice multiplexer ”aluSliceNDice” was incorrect.

To validate our SUB function, we tested it with combinations of both positive and negative operands. Neither the ”3 - 3” nor the ”-5 - (-5)” test originally returned ”1” for our zero flag, which alerted us to a flaw in our design. We originally calculated the ”zero” flag by first ”OR”ing all of the result’s 32 bits, returning a 1 if any bit was 1 and returning 0 otherwise. We ”NOR”ed this result and the carryout, but this produced the misleading result of ”0” for our ”zero” flag for the subtractions listed above. We switched our implementation to ”NOT” the result instead, disregarding the carryout.

We ran into no trouble with XOR, OR, AND, NAND, and NOR during testing and exhaustively tested each of these functions.

Timing Analysis

Through simulation using GTKWave, it was observed that the subtraction operation had the worst case propagation delay when compared to the other operations as we expected. The worst delay for each of the operations are shown below:

Table 1: Worst case delay for each operation.

ADD	85,460 ns
SUB	86,780 ns
XOR	660 ns
SLT	1980 ns
AND	1983 ns
NAND	320 ns
NOR	320 ns
OR	330 ns

We had originally hypothesized that the SUB operation would have the worst delay because it would require the ADD operation, the inverting of all the components, and an extra NOT gate.

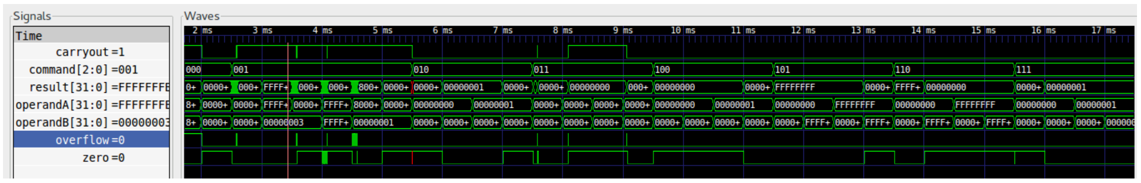

Figure 3: This waveform shows each of the operations for our 32-bit ALU and their propagation delays.

Work Plan Reflection

We planned to spend 1.5 hours working on creating test benches, and we ended up spending about 2 hours on this part of the lab. Designing and implementing the control logic for each function took about 5 hours, as compared to our goal of 3 hours. It is difficult to say exactly how long it took us to test our operations, since we were doing so each step of the way during implementation. We ran into various flaws in our design during this process and took much longer than the 1 hour that we originally allocated to testing. We took an hour to create a block diagram for a single bit and a diagram for how the bit slices fit together, which was close to our 30 minute goal. Capturing the worst propagation delay for each of our operations took an hour, which is what we had originally predicted. In retrospect, we could have allocated more time to designing and implementing the control logic. We particularly wish we had chosen to implement an LUT-based design instead of a mux-based design from the start, and spending more time understanding the individual operations within the ALU may have led us to a smarter design.