CompArch: Lab 1

Brenna Manning, William Saulnier, Ziyu (Selina) Wang

10-14-15

Abstract

We have created an ALU, a subset of the standard MIPS ALU, and a processor component. Using Verilog, we wrote an ALU that could perform 8 operations: addition, subtraction, xor, slt, and, nand, or, and nor. We are confident that our ALU design is complete, correct, and ready to be included in the CPU. Details about our ALU is documented in the following sections.

1 Implementation

To implement our 32 bit ALU, we used a look up table that would take in a command and all of the operations our ALU could perform, and would output which operation to set as the result. It would also output nesssary control flags to modify the inputs accordingly. After determining which operation to perform, the ALU would calculate the result of that operation using one of five modules: AND, OR, ADD, XOR, SLT, and output the result as well as flags for cases of carryout, overflow, and a zero result, when appropriate.

Following the path from the inputs, both enter an inverter block which, based on the control signals from the look up table (LUT), would invert one, both, or none of the input operands. The operands, after passing through the inversion block, enter all five of our operand boxes. Those outputs are then passed into a five input mux which, using the control signal from our LUT, decides which module's output to pass through to our result pins. Lastly, a final check is made to set the carryout, overflow, and zero flag pins. In the case that we care about the carryout or overflow flag, an and gate performs that final check for the proper instruction. For the zero flag, the 32 bits of the result are passed through an OR gate together to see if any one of them is set high, and if none of them are set high, then the zero flag is set high.

The only other unique factor in our implementation is that our adder accepts the invert b flag from the LUT. This is so we can add 1 to our inverted result to make the b input a proper 2's complement. We don't need to perform this operation in our other modules because they do not rely on 2's complement.

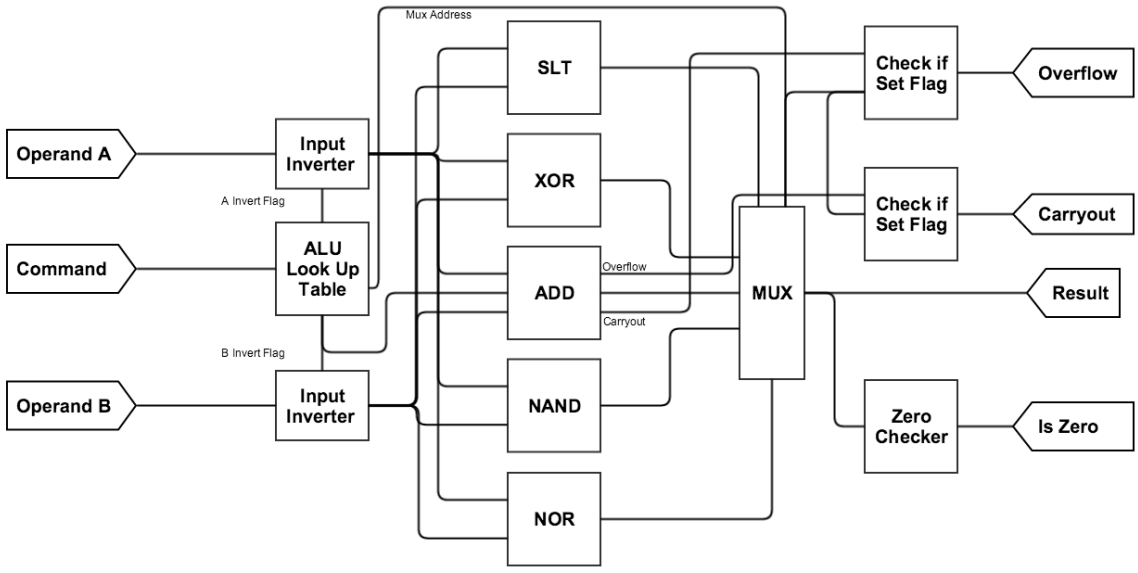

Figure 1: *Abstract Top-Level Diagram*

2 Test Results

Our test bench was designed to succinctly test each of the modules specifically, ensuring proper function as well as verifying correct flag setting behavior. A short test bench was selected for each opperation with the goal of isolating problems in a quick and succinct way. All of our operands were selected via a random generator.

For the adder and subtractor, our test bench includes a set of 6 mathamatical operations each. For both adding and subtracting, we test operating on two positive numbers, two negative numbers, a positive and negative number, and a negative and positive number. Two additional test cases were select such that they would cause overflow; two appropriately large (or small) negative and positive numbers for the adder, and a positive and very small negative number plus a negative and a very large positive number for the subtractor. These tests together verify proper operation of the bitwise addition operations through their outputs, as well as the proper setting of the carryout and overflow flags.

For the SLT, a similar system of the adder and subtractor test benches was used. This time, we are expecting our flags to remain 0 except for the zero flag. This test allows us to verify the correct operation of SLT as well as our zero checking functionality.

For the boolean operations, a pair of random values were chosen to verify that their behavior is functional. Only two were used, as additional testing became redundent as the operands selected provide a mostly comprehensive analysis of every bitwise operation.

Our test benches were instrumental in ensuring that our SLT was setting each of the 31 bits to 0 when it is initialized. Without that operation, the SLT's outputs and the flags were being indeterminatly set. With a quick code change, the code worked properly. Otherwise, the testing was great from the get go, and we were able to quickly assess if our ALU was behaving properly. Additionally, at the very end, we noticed that our zero flag checking was incorrect. This was because our delays were set too short. By extending our delays we were able to ensure correct calculation every time.

3 Timing Analysis

We calculated the worst case propagation delay for each of the operations of the ALU. The gate delays for gates used in ALU operations were included at the beginning of the verilog file. Taking into account that the basic logic gates that make up all the other gates are NAND, NOR, and NOT, we created the other gates out of these three to see how many basic gates they are actually made of. Since NOT, NOR, and NAND are the basic gates, they are each actually only one gate in the worst case propagation. Using the standard 10 units per input gate delays, this means NOT, NOR, and NAND, each have a delay of 10 units per input per bit. Our ALU is a 32bit ALU. This means that the single input NOT gate has a delay of 320, and in the case of two 32 bit inputs, NOR, and NAND, would each have a delay of 640.

We created AND and OR gates by inverting the inputs to NOR and NAND gates. The worst case propagation in the case of an AND gate is a NOT gate followed by a NOR gate. Likewise, the worst case propagation in the case of an OR gate is a NOT gate followed by a NAND gate. The AND and OR gates each must have delay values equal to the combined delays of the gates they are made up of. We already determined that a 32 bit NAND or NOR gate with 2 inputs has a delay of 640, and a single input 32 bit NOT gate has a delay of 320. Therefore, in a 32 bit AND gate consisting of a NAND gate with two 32 bit inputs and a NOT gate with one 32 bit input, the delay must be 960. The same goes for a 32 bit OR gate.

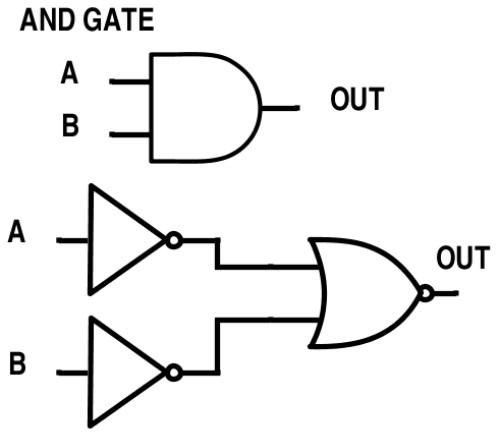

Figure 2: *AND gate in basic gates: Worst case propagation = 2 gates: 1 NOT and 1 NOR*

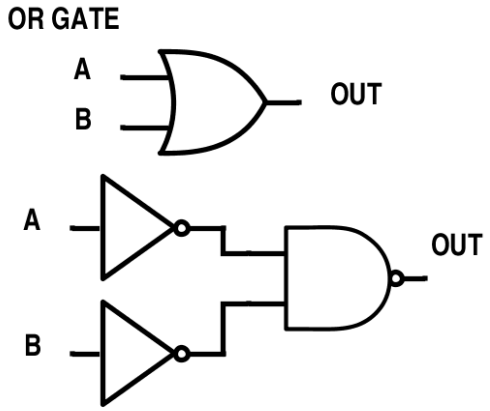

Figure 3: *OR gate in basic gates: Worst case propagation = 2 gates: 1 NOT and 1 NAND.*

Finally, we constructed XOR out of basic gates. XOR can be made using four NAND gates, with a worst case propagation passing through 3 gates. This means that in the worst case propagation, an XOR gate is effectively 3 NAND gates, the delay of a 32 bit XOR gate with two inputs is equal to triple the delay of a 32bit NAND gate with two inputs. $3 * 640 = 1920$ so the delay of a 2 input 32 bit XOR gate is 1920.

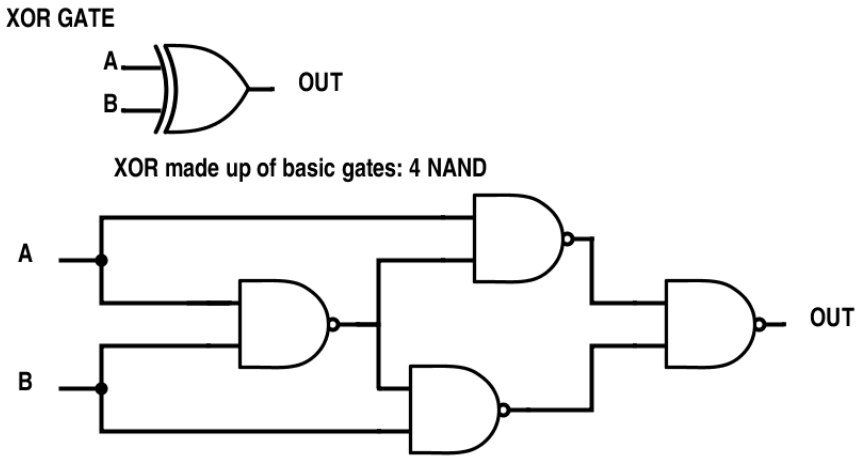

Figure 4: *XOR in basic gates: Worst case propagation = 3 NAND gates.*

Consider that each of the 32 bit ALU's Operations take in two 32 bit inputs:

Operation	Basic Gates/Bit	Total Gates(32bit)	Delay(32bit 2 Inputs/Op.)
NOR	1	32	640
NAND	1	32	640
OR	2	64	960
AND	2	64	960
XOR	3	96	1920
ADD	7	224	3870
SUB	8	256	4190
SLT	11	352	4250

All other operations in our ALU are made up of these gates. The worst case propagation delay through the 32 bit adder passes through three gates: an XOR gate, an AND gate, and an OR gate. From this we can determine that the delay of the 32bit adder/subtractor is equal to the sum of the delays of XOR, AND, and OR. $1920 + 960 + 960 = 3840$, then an additional delay of 30 is added to this value to account for the XOR at the end of the adder checking for overflow, so the total delay of the adder is equal to 3870.

The delay for the 32 bit subtractor will be nearly the same as the adder, except it contains an additional NOT gate before the adder/subtractor module. As calculated previously, this adds an additional delay of 320, so the delay of the subtractor is equal to 4190.

The worst case propagation delay for the 32 bit SLT, Set Less Than module, is a subtractor followed by a 1 bit XOR gate. The subtractor has a delay of 4190 and the XOR has a delay of 60, so the 32 bit SLT has a total delay of 4250.

4 Work Plan Reflection

Overall, we spent more time than we had expected to on this Lab. Some things, such as the structural modules for the simple gates, took far less time than we had accouonted for. Others, such as the SLT module, took a bit longer to figure out than we had planned. The Set Less than module took longer than expected because it took more time than we had accounted for to figure out how to determine if the result of the subtractor was negative.

Also, in our original work plan, we had not specifically accouonted for the time we would spend

on the Multiplexer Module of our verilog, where the command was chosen. We spent hours on this, but had not included it in our initial work plan.

We also spent more far more time working on the day we had planned to simply finish the write-up than we had intended to. We discovered some things about our prior work that were problematic through the process of writing about them, and then needed to correct these mistakes. This led to a lot of time spent debugging issues with the verilog and changing how gate delays were being calculated. One of these problems was that we had forgotten to add 1 to one of the operands after inverting it to make it negative in our subtraction. Another problem was that the delays were not correct, so the SLT output flags were inaccurate. The night we intended to spend only a brief amount of time reviewing what we had written, we ended up spending 3-4 hours fixing problems we had not caught before.