Computer Architecture Lab 1

Shane Skikne — Meg McCauley — Ryan Eggert

October 14, 2015

Implementation

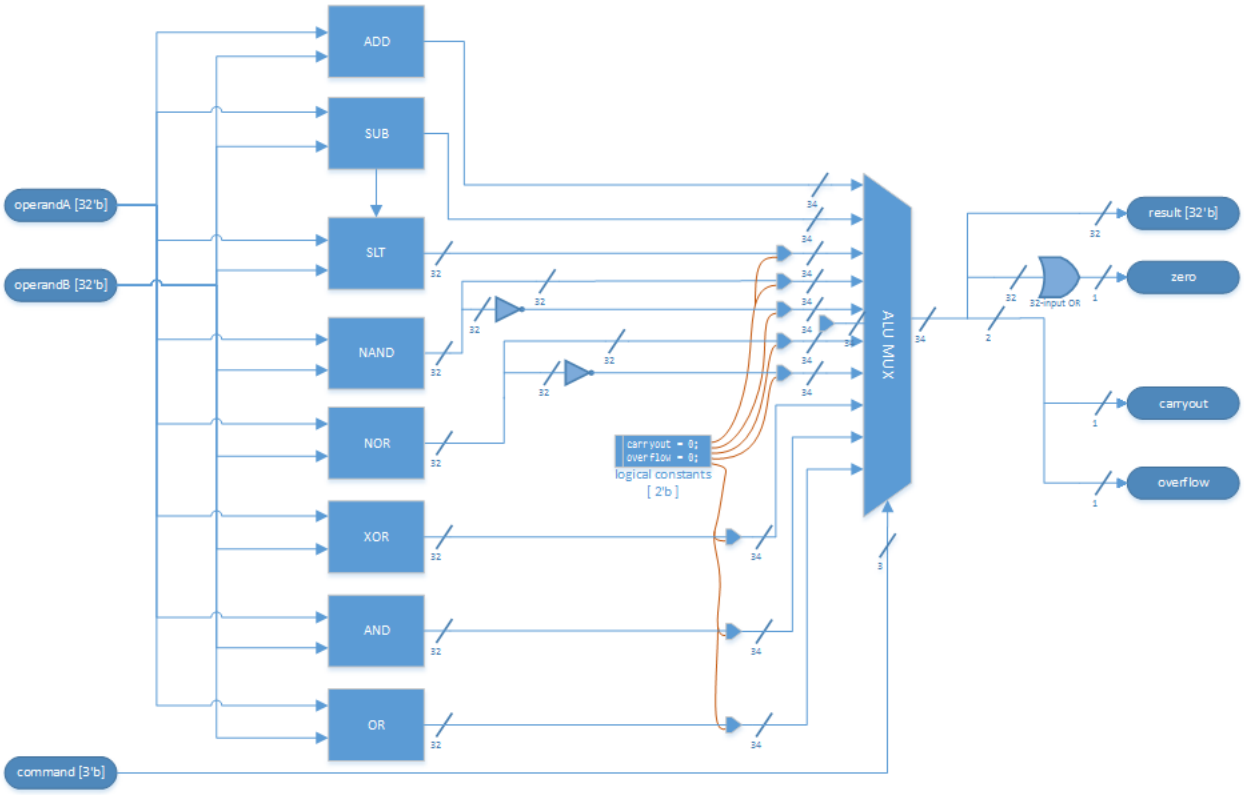

Figure 1: A schematic of the general architecture of our ALU.

As shown in 1, we did not use a bitslice architecture for our ALU. Instead we have eight semi-independent modules operating generally in parallel. Those eight modules may be grouped into two larger units. The AND, NAND, OR, NOR, and XOR modules have very similar architectures and may be considered part of a logical subunit. The ADD, SUB, and SLT modules can share structures and functions and may be considered part of a arithmetic subunit.

Implementing the logical subunit was quite simple, because each bit operation is independent so we're just doing the same operation 32 times. To change the overall operation, we just switched the gates being used. The only part that could have been challenging was writing the code correctly 32 times without an error, but this was avoided by using a python script to create all of our duplicate code. At this point, we also weren't sure whether or not to only use the basic gates or all gates so we implemented all the logical operations

with only the basic gates. While this made things slightly more complicated, that logical subunit was still generally straightforward.

Testbenches

ADD

The 32-bit adder testbench is meant to check that all the bits in the adder are interacting correctly. First, simple one-bit addition behavior is verified. Next, the carryout bit is tested. The ability of the adder to correctly propagate carried digits is tested next. Finally, it is checked that the adder correctly handles positive and negative overflow situations. To check that each adder input is wired similarly, nearly every set of test values is run twice, swapping adder inputs.

Our first testbench failure occurred while testing our 32-bit adder. (See Figure 2 on page 2.) In our output, there were Z's which indicated that a wire was not properly attached. Upon further investigation, we determined that one of our variable names was not consistent throughout the code and so the wire wasn't getting hooked up. We had referred to the overflow as 'ovf' in the definition but then 'overflow' in the structural verilog. Once we changed the 'ovf' in the definition to match the 'overflow,' we fixed the first failure in our testbenches.

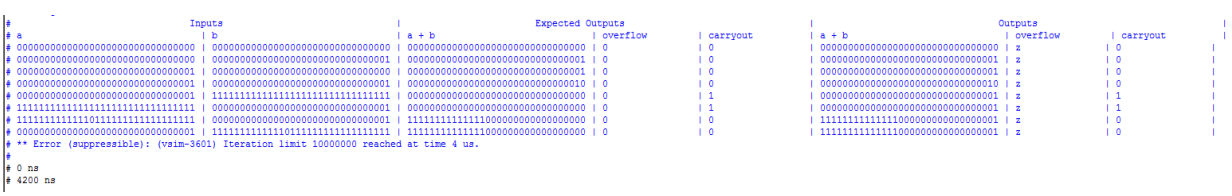

Figure 2: Our first testbench failure caused by wires not being hooked up due to inconsistent variable names

The next failure occurred right after we fixed the first one. (See Figure 3 on page 2.) This error caused the last four test cases to have an incorrect a+b values. This was due to the carry in value of the first adder being set to adder1_cout instead of carryin_null. After this was fixed, our testbenches for the 32 bit adder were successful.

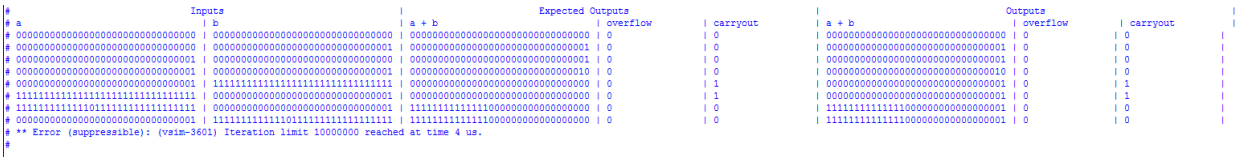

Figure 3: Our second testbench failure caused by the wrong carry in values being set for the first adder

Logical Subunit

The modules in the logical subunit all use similar test cases. First, the module is run through four one-bit test cases–this verifies that one bit of the 32-bit output is behaving correctly. Since all bits are independent, theoretically, one bit working means the Verilog is correct and each bit should work. We then test 'consistent' cases (one input all zeros, the other input all ones; both inputs all ones) to verify consistent behavior across all bits (since these are bitwise operations, each bit should be independent of neighboring bits). To further check

Table 1: Test cases for each module in the logical subunit

a	b
b00000000000000000000000000000000	b00000000000000000000000000000000
b00000000000000000000000000000000	b00000000000000000000000000000001
b00000000000000000000000000000001	b00000000000000000000000000000000
b00000000000000000000000000000001	b00000000000000000000000000000001
b11111111111111111111111111111111	b11111111111111111111111111111111
b00000000000000000000000000000000	b11111111111111111111111111111111
b10101010101010101010101010101010	b01010101010101010101010101010101
b00000000000000000000000000000011	b00000000000000000000000000000100
b00110100111110110101111000111000	b01011011000111110001100010010011
b00000000000000000000000000000011	b00000000000011000000000000000011

for erroneous intra-bit interactions and verify functionality, there are four more tests. A set of 'interlaced' inputs verifies consistent behavior when presented with `a[n]` = ¬`b[n]`. The remaining tests serve to check various combinations of ones and zeros across both inputs. The same set of test cases are used for all logical modules [AND, OR, XOR, NAND].

Since the logical subunit was generally straight forward, our test bench didn't uncover any major bugs in our implementation. At one point the entire output was not 0's or 1's, because we hadn't named variables correctly, but that was discovered and corrected quickly. the only significant bug discovered was actually an incorrect output in our testbench, not from our code.

0.1 Central ALU Logic

The central ALU logic has four main responsibilities–

1. Provide the correct inputs and outputs to each module
2. Using the three-bit `command` input, select (multiplexer) the output of the correct module
3. Unpack the 34-bit multiplexer output to the correct ALU outputs
4. Raise the `zero` flag if necessary

Table 2 shows the tests we run on the ALU as a whole. Because each module is independently tested and verified, these tests are meant to verify that all modules are connected, that the multiplexer chooses the correct module, that `result`, `carryout`, and `overflow` are correctly interpreted from the multiplexer's output, and that the zero flag is raised appropriately. We perform three additions to check addition functionality, the zero flag, and then the carryout and overflow outputs. Each module is individually checked to verify its connectivity. The SLT module is tested twice, again to check that the zero flag is being raised correctly.

Timing Analysis

In the table below, we calculated and simulated the worst case propagation delay for each operation. For the everything except ADD, all the computing can happen simultaneously so the worst case delay for 32 bits is the same as the worst case delay for 1 bit. So, except ADD, the worst case delay is simply the delay of using that gate once. ADD is unique, because each addition is affected by the previous addition. The worst case delay of adding just once is 120ns. The worst case for adding 32 bits isn't 120ns multiplied by 32, though, because half of the worst case delay of a single addition can be done simultaneously. The operation then has to wait for the previous addition's carry out, which accounts for the other half of the delay. Therefore, the worst case delay is actually 60ns multiplied by 32. To get the overflow, we then use XOR which adds 60ns on top.

Table 2: ALU Central Logic/Whole ALU test cases

INPUTS			OUTPUTS			
command	operandA	operandB	result	carryout	overflow	zero
‘ADD	b00000000000000000000000000000001	b00000000000000000000000000000001	b00000000000000000000000000000010	0	0	0
‘ADD	b11111111111111111111111111111111	b00000000000000000000000000000001	b00000000000000000000000000000000	0	0	1
‘ADD	b11111111111111111111111111111011	b10000000000000000000000000000000	b01111111111111111111111111111011	1	1	0
‘AND	b00011111000001100011100000100001	b00111111111111110000111000010001	b00011111000001100000100000000001	0	0	0
‘OR	b00011111000001100011100000100001	b00111111111111110000111000010001	b00111111111111111001111000110001	0	0	0
‘NOR	b00011111000001100011100000100001	b00111111111111110000111000010001	b11000000000000001100000111001110	0	0	0
‘NAND	b00011111000001100011100000100001	b00111111111111110000111000010001	b11100000111110011111011111111110	0	0	0
‘XOR	b00011111000001100011100000100001	b00111111111111110000111000010001	b00100000111110010011011000110000	0	0	0
‘SUB	b00000000000000000000000000000100	b00000000000000000000000000000011	b00000000000000000000000000000001	0	0	0
‘SLT	b00000000000000000000000000000001	b00000100000000000000000000000001	b11111111111111111111111111111111	0	0	0
‘SLT	b00100000000000000000000000000001	b00000000000000000000000000000001	b00000000000000000000000000000000	0	0	1

Operation	Calculated (ns)	Simulated (ns)
ADD	1980	1980
XOR	60	60
AND	30	30
NAND	20	20
NOR	20	20
OR	30	30

Work Plan Reflection

The work plan called for the testbenches and central ALU logic to be written by one person, the ADD/-SUB/SLT modules written by a second person, and the logical modules written by a third person.

Writing testbenches without modules to test them against was somewhat more difficult than anticipated. With no modules to run the testbenches against, the first set of testbenches usually had syntactical errors or omitted components (e.g did not include a newly-specified input or output). As a result, the testbenches took longer than expected–much of this time has been spent revising and fixing the first testbenches. Contrastingly, the central ALU logic took very little time to write and test. After taking the time to draw out a system diagram [Figure 1], writing and testing the associated verilog took very little time. In the end, the testbench-writing took more hours than anticipated and the ALU logic took less time than anticipated; net, this took approximately 2-3 hours longer than the 6 hours budgeted.

Additionally, while it specified the writing of certain components, the workplan did not account for time spent writing up the lab.

Finally, the team's travels certainly interfered both in our ability to work and to communicate. These communication and working issues definitely affected our ability to complete our work plan. Hopefully, in the future, that will no longer be an issue. We will set hard in-team deadlines to make sure we don't fall behind.