EE25: Computer Organization — Remy Ren
Tufts University, Fall 2023 — Lab 2 Report

Using two late tokens on this assignment.

Lab Intro and Objectives

The main objective of this lab is to build and test several starter components via VHDL to help us in our CPU design. These starter components are primarily focused on arithmetic and memory, such as an ADD (64-bit adder), ALU (arithmetic logic unit), registers, and a memory testbench.

Brief Component Design Descriptions

1. ADD: For this component, it takes in two 64-bit STD_LOGIC_VECTORs and adds them together. These can be signed or unsigned, the adder does not care. In implementing the component, I chose to use structural modeling. This was due to it utilizing a 1-bit full-adder I made. The powerful VHDL generate statement allows me to initialize 64 of these full-bit adders via for loop, then connect the corresponding carries, ultimately allowing for a guaranteed proper result.

   This utilizes structural modeling as it required a full-adder component.

2. ALU: The ALU component contains two 64-bit inputs, an operation selector input (4-bits), a 64-bit result output, as well as a zero and overflow flag. There are 6 total modes implemented, those being: bitwise AND of the two inputs, bitwise OR of the two inputs, addition of the two inputs using ADD, twos complement subtraction of the inputs, passthrough of the second input, and a bitwise NOR of the inputs. The overflow output flag will only change if the ALU is in add or subtract modes. The zero flag will change whenever result is zero.

   This utilizes structural modeling as it requires multiple instances of the ADD component in twos-complement subtraction.

3. alu_control: This component takes in a two-bit input for the ALU operation code, then an 11-bit general operation code. Using dataflow modeling, this component is then designed to output the correct 4-bit ALU mode depending on a combination of the ALUOp and general operation code bits.

4. registers: This component acts as the memory, allowing for reads and stored writes. Reads can be done concurrently, however the component is made with behavioral modeling as writes must be on the negative clock edge. There are a total of 32 accessible addresses, with each address storing 64-bits. Two reads can be done at the same time as there are two different input lines for addresses. Only one write can be done at a time. Writes are done by specifying the address of the register and the corresponding data to go with it.

5. dmem_testbench can be found in the next section. It is just an exercise in understanding how to read and write to memory.

A Brief Explainer on My Testbenches

All testbenches are modeled using structural as they require their to be tested components in their architecture. Should any test fail, its corresponding error message should be reported, typically with the contents that show what was asserted.

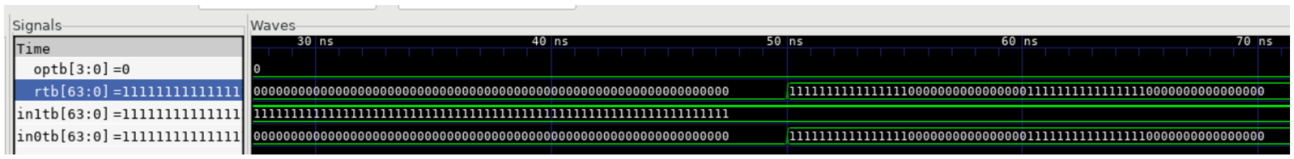
In the following results and waveform sections, you will see these specific tests:

1. ALU: AND/OR operations in binary, add/subtract operations in hex, and working zero/overflow flags.

2. Registers: A read using both read lines, a write, and a demo showing XZR cannot be written to.

3. DMEM Testbench: Two reads, two writes, and inbetween there may be some debug messaging.

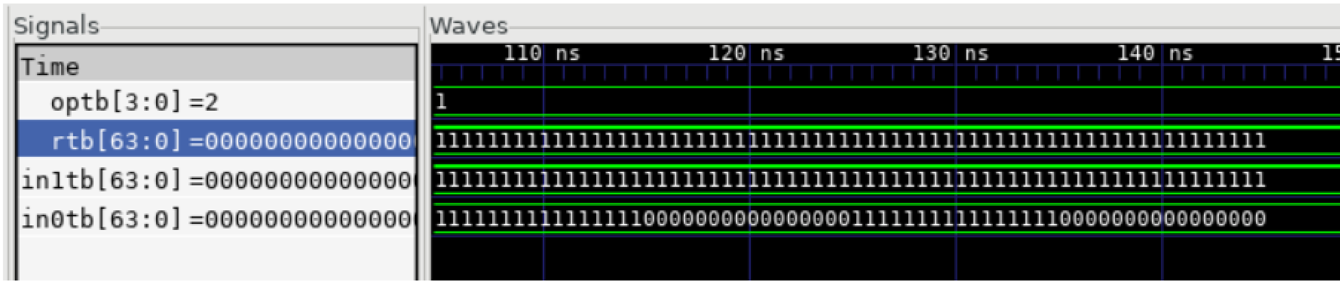
Results and Waveforms



```
Time                          30 ns          40 ns          50 ns          60 ns          70 ns
optb[3:0] =0                  0
rtb[63:0] =11111111111111     0000000000000000000000000000000000000000000000000000000000000000   1111111111111111000000000000000011111111111111110000000000000000
in1tb[63:0] =11111111111111   1111111111111111111111111111111111111111111111111111111111111111
in0tb[63:0] =11111111111111   0000000000000000000000000000000000000000000000000000000000000000   1111111111111111000000000000000011111111111111110000000000000000
```

Figure 1: ALU AND Operations in Binary

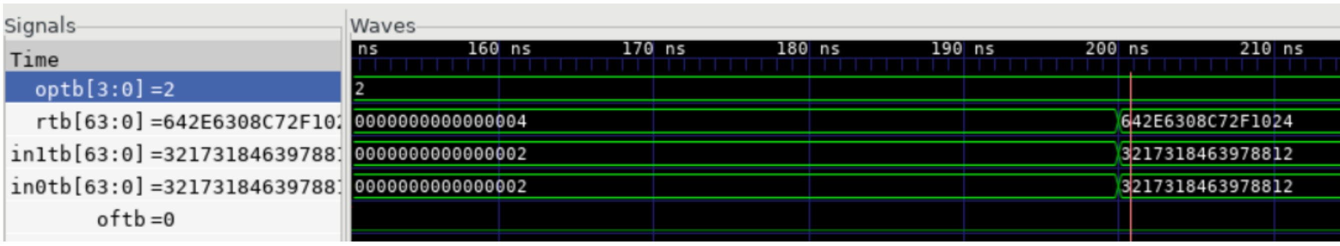
In **Figure 1**, we see a test for AND operations on the ALU, displayed in binary. Our inputs are in1tb and in2tb, and rtb is our output. The first test has in0tb with all zeros and in1tb with all ones. We see that the AND operation is correct as rtb is all zeros. In the second test, in0tb has changed to a pattern of ones and zeros, and we see that the result follows in0tb as in1tb is still correctly all ones.



```
Signals                          Waves
Time                                  110 ns          120 ns          130 ns          140 ns
   optb[3:0] =2                  1
   rtb[63:0] =00000000000000     1111111111111111111111111111111111111111111111111111111111111111
in1tb[63:0] =00000000000000      1111111111111111111111111111111111111111111111111111111111111111
in0tb[63:0] =00000000000000      1111111111111111000000000000000011111111111111110000000000000000
```

Figure 2: ALU OR Operation in Binary

In **Figure 2**, we see that the OR operation works properly as the result matches an input of all ones.



```
Signals                          Waves
Time                             ns     160 ns     170 ns     180 ns     190 ns     200 ns     210 ns
   optb[3:0] =2                  2
   rtb[63:0] =642E6308C72F10     0000000000000004                                  642E6308C72F1024
in1tb[63:0] =32173184639788      0000000000000002                                  3217318463978812
in0tb[63:0] =32173184639788      0000000000000002                                  3217318463978812
         oftb =0
```

Figure 3: ALU ADD Operations in Hex

In **Figure 3**, we see the working ADD operation in HEX. The first is simply $2+2=4$, the second is also addition of two identical values, with the result correctly being double the input amount.

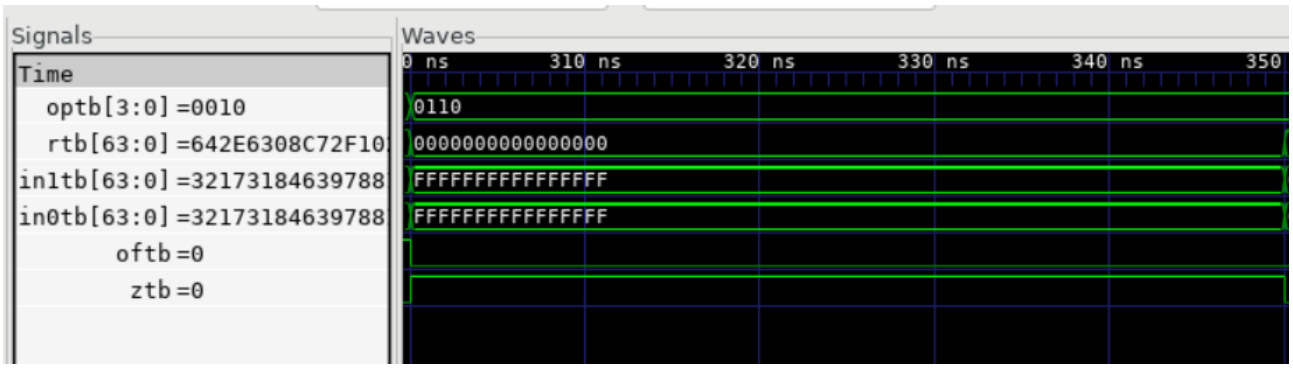


Figure 4: ALU SUB Operation in Hex and Zero/Overflow Flag

Figure 4 correctly demos subtraction (now with the operation code in binary, sorry earlier!), where identical values get a zero result. We also see that the overflow flag (oftb) is properly zero and the zero flag (ztb) is active!

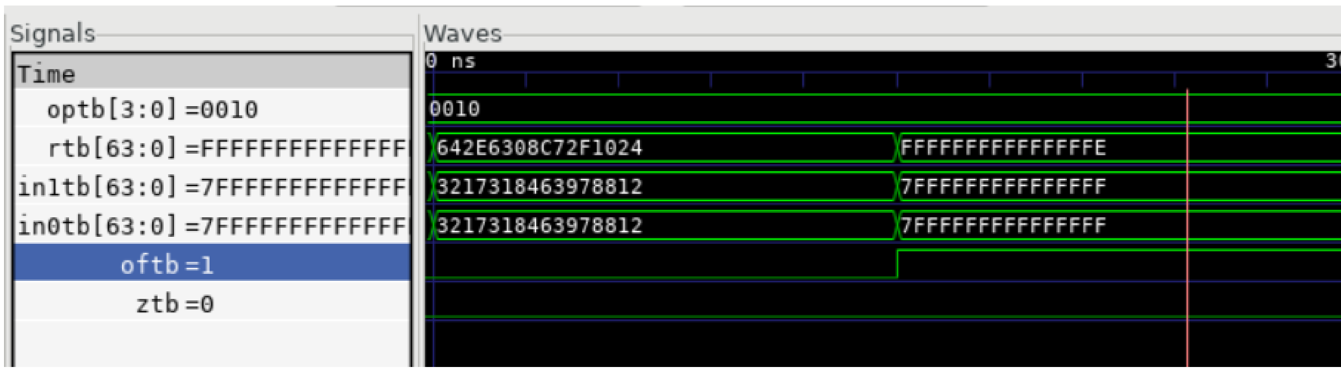


Figure 5: ALU Overflow Flag for HIGH

Figure 5 demos the overflow flag being set from an ADD operation. As both inputs are $x7FFFFFFFFFFFFFFF$ which is positive, the result is negative so the overflow flag properly set high.

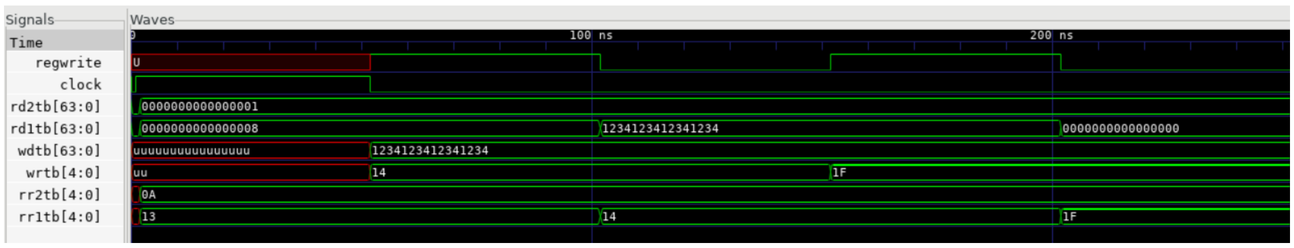


Figure 6: Register Tests

Figure 6 depicts all the tests done for registers. There is some initial clock checking. Afterwards one read occurs on both lines, where rr1tb and rr2tb (register to read) allow rd1tb and rd2tb (register data) to properly read in data at X19 and X10. Both retrieve proper values of hex 8 and 1. Note that the reads happen regardless of clock edge.

Then, a write is done, where wrtb (write to register) has data written to X20, and afterwards a read is done to verify that the write has taken place. Both show their expected outcome, and note the negative clock edge resulting in the write.

Finally, a write is done to XZR. This is at 150ns, using the same write data as the previous test. We then read XZR at 200ns, which shows that it correctly did not change and remained all zeros.

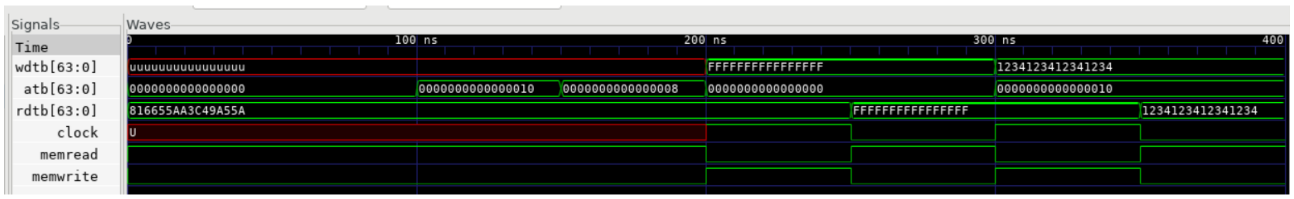


Figure 7: DMEM Testbench

In **Figure 7**, a variety of test are done on the dmem_testbench. The first few reads were done to help me figure out the interval spacings in memory, as dmem had 8 byte "chunks" that were identical. Afterwards, I write all ones (shows up as Fs in HEX) to the base address of all zeros. I then read to check it was correct, which it was. Finally I preform a write then read with data that has 1234 repeating in HEX, which worked out fine.