**EECE 2323 Digital Logic Design Lab Report**

Lab 5 Adding Data Memory to the Datapath

| Student Name:  Ousmane Toure & Kaitlyn O’Flaherty | Section #: 1 |
| --- | --- |
| Instructor Name: Dr. Aboelela  Lab TA Name: Keshav Bharadwaj Vaidyanathan, Srinidi Somasundaram Subbulakshmi, Yuchao Su | Date: 08/04/22 |

1. **Background & Purpose**:

In this experiment, we implemented data memory into our processor datapath utilizing the Register file with a zero register and Arithmetic Logic Unit that were created in Labs 4 and 3, respectively. The processor is a RISC, so the values are loaded from memory to register and stored to memory from register. The results of the ALU will be stored in the register before they are stored in the memory. The ALU is used to calculate the memory address but not the data itself when used in load and store instructions. Our goal was to implement the data memory into the processor datapath using the ALU, zero register, register file, and multiple muxes. From our previous labs, our ALU performs all necessary operations to generate new values and needs a register file to load and store these values in memory. The results were then displayed utilizing the VIO Dashboard on Vivado and the LEDs on the PYNQ. The data memory is much larger and slower, but can hold many more locations than the register file created in lab 4. The goal of this lab was to utilize verilog code in vivado to implement data memory into our design with the register file and ALU functions and virtually test it before physically implementing it. Completing this lab enhances our confidence and proficiency in vivado and computational architecture.

This lab is important to the scientific community because each processor needs to be able to store values in its memory. This allows the processor to quickly access information, and without memory a computer would not be able to function properly.

1. **Pre-Lab Response:**

**Test Sequence**

| Register | Value (binary) | Value (signed decimal) |
| --- | --- | --- |
| 0 | 00000010 | 2 |
| 1 | 11111011 | -5 |
| 2 | 00010101 | 21 |
| 3 | 00010111 | 23 |

**Table 1: The expected values stored in each register in Test Sequence**

Functionally:

1. reset

sets all values in the registers to 0

2. store values 2 and 4 to Mem[0] and Mem[1]

sets the value of memory address 0 to equal 2 and value of memory address 1 equal to 4

3. Load Mem[0] to Reg[0]

The value of register 0 is now equal to the value of memory address 0 (2).

4. Load ~Mem[1] to Reg[1]

the value in memory address 2 (4) is bitwise inverted (now = -5). The value of

register 1 is now equal to the new value (-5).

5. Write the value 8’h15 to Reg[2]

The value of register 2 is changed to 8’h15 (21).

6. store the value of (Reg[0] + Reg[2]) to Reg[3]

The value of register 0 (2) and register 2 (21) are added. Register 3 is now equal

to their sum (23)

7. store Reg[2] and Reg[3] to Mem[2] and Mem [3]

The values of memory addresses 2 and 3 are equal to the values of registers 2

and 3, respectively (21, 23).

8. reset

The values of the register are now all set to 0.

**2.2 - Create Your Own sequence**

Sequence:

1. reset
2. store 1 to memory 1 and 2 to memory 2
3. load memory 1 to reg 1, and memory 2 to reg 2
4. add reg 1 reg 2 store in memory 3
5. save memory 3 as register 3
6. send register 3 to memory
7. clear reg 3 2 and 1
8. end

\*ALU operations not tested: bitwise OR, bitwise AND, bitwise inverse, arithmetic shift right, logical shift left, beq, bne.

| Register | Value(binary) | Value (decimal) |
| --- | --- | --- |
| 1 | 00000001 | 1 |
| 2 | 00000010 | 2 |
| 3 | 00000011 | 3 |

**Table 2: The expected values stored in each register in My Own Sequence**

Functionally:

1. Reset
   1. sets all values in the registers to 0
2. store 1 to memory 1 and 2 to memory 2
   1. sets the value of memory address 1 to equal 1 and value of memory address 2 equal to 2
3. load memory 1 to reg 1, and memory 2 to reg 2
   1. the value in Mem[1] is now the value of Reg[1], is equal to 1. The value in Mem[2] is now the value of Reg[2], is equal to 2.
4. add reg 1 reg 2 store in memory 3
   1. The value of Reg[1] is added to the value of Reg[2], values 1+2. The result of the operation is stored in Mem[3], value 3
5. save memory 3 as register 3
   1. The value in Mem[3] is saved to Reg[3], value 3.
6. send register 3 to memory
   1. Value in reg[3] is stored in Mem[3], value 3.
7. clear reg 3 2 and 1
   1. reg[3], reg[2], reg[1] are all set to 0.
8. end

**Screenshots of results found in Appendix A.**

1. **Summary of Design Implementation**
   1. **Results and Analysis:**

When conducting this experiment, we tested the functionality of data memory in our processors datapath, scaling our design up from Lab 4 and 3. We loaded various values from memory to register by utilizing the ALU. The ALU calculated the memory address in which the data was to be stored, as well as completing various operations on the data. We successfully implemented the data memory into the processor path and were able to test its functionality utilizing the Virtual Input Output dashboard. A datapath was created and data could be sent to and from both the register file and the data memory, as seen in the screenshots below. The pictures of the VIO show us clearing the register file, loading values 2 and 1 to their respective registers, doing an ALU operation of addition between the registers, storing the sum into register 3, verifying the value was in the correct register, and finally clearing out previous values. All values from the VIO dashboard were consistent with the prelab example, highlighting our processor datapath was in fact correct. You could face many errors when conducting this lab. For example, when creating your virtual input output, if you didn’t correctly instantiate your values in your top file, Vivado will leave you with constants with no values. Furthermore, accidentally saving a port in the VIO for clock would cause you to have an extra empty port because VIO already adds clock.

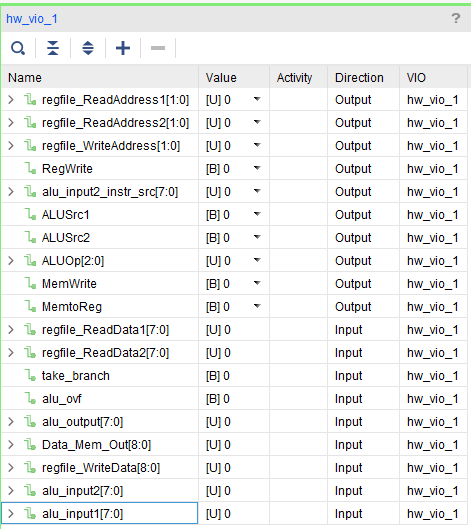
* 1. **Conclusion & Recommendations:**

Based on our results, we can conclude that Lab 5 consists of implementing data memory into our processor datapath from Labs 4 and 3, scaling our processor into a more sophisticated design. By utilizing the ALU we were able to send instructions to be read by the processor, allowing us to test important arithmetic operations such as addition, bitwise inversion, etc. These tools are crucial for a Central Processing Unit, and gives it the ability to perform varying tasks of different degrees. Completing this lab showed the difference between data memory and registers, especially highlighting the path at which data is stored in registers. Furthermore, it highlighted the importance of separating instructions and data, particularly putting emphasis on the different memory locations each uses according to MIPS

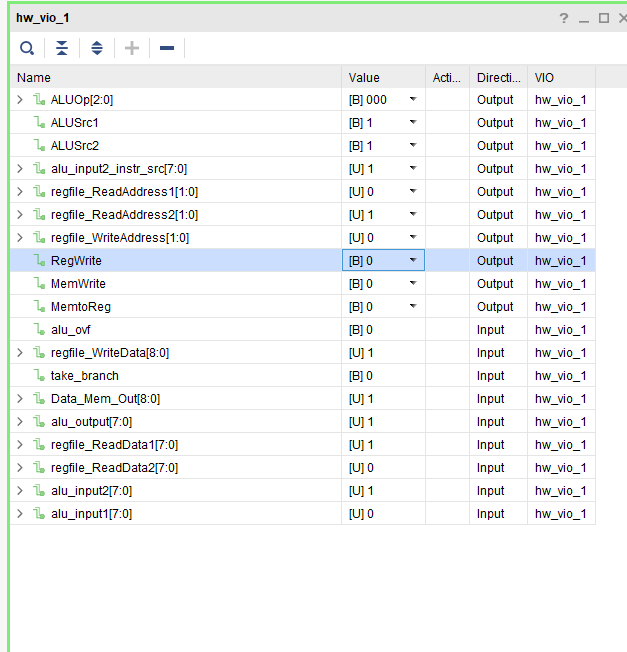
Recommendations going forward would be to give one instruction set that students should copy. Furthermore, I’d recommend telling students to put the regwrite, memwrite, and memtoreg inputs as buttons in the VIO, as many students had issues with the register continuously writing.

**Appendix A: Design Program Files (Verilog modules, testbenches, etc)**

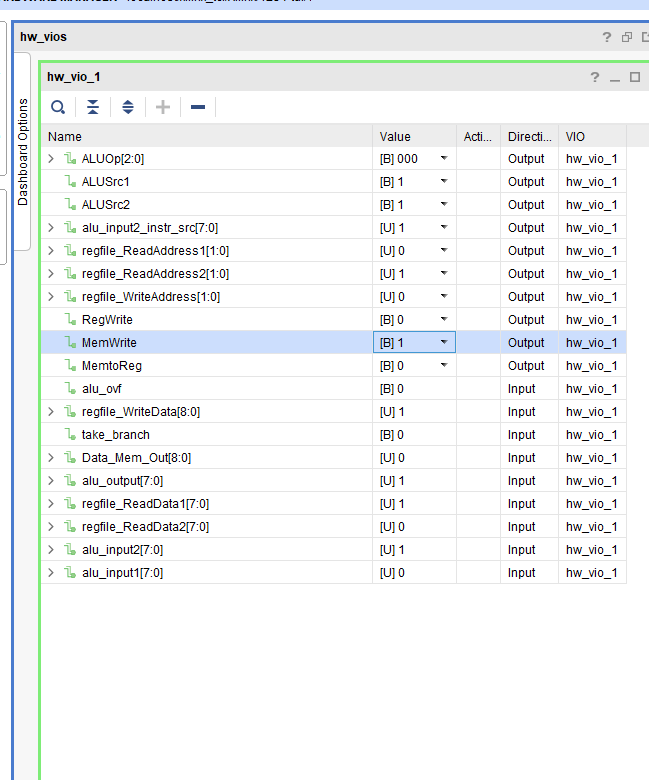
**Sequence Testing Screenshots:**



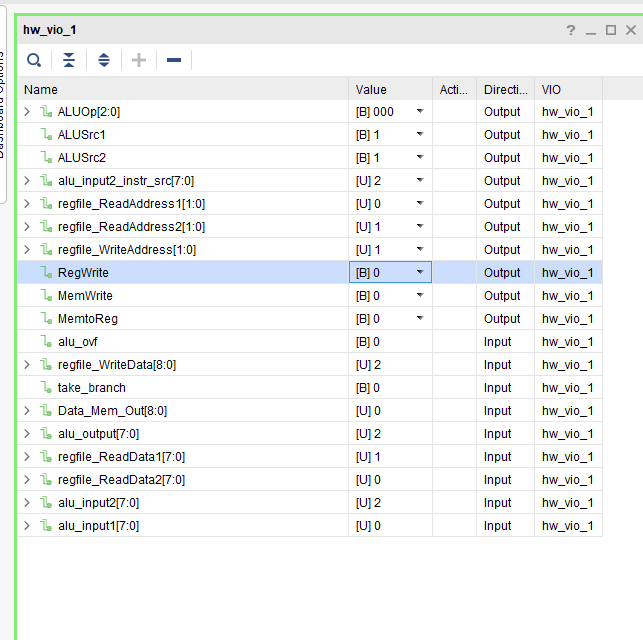
**Reset**



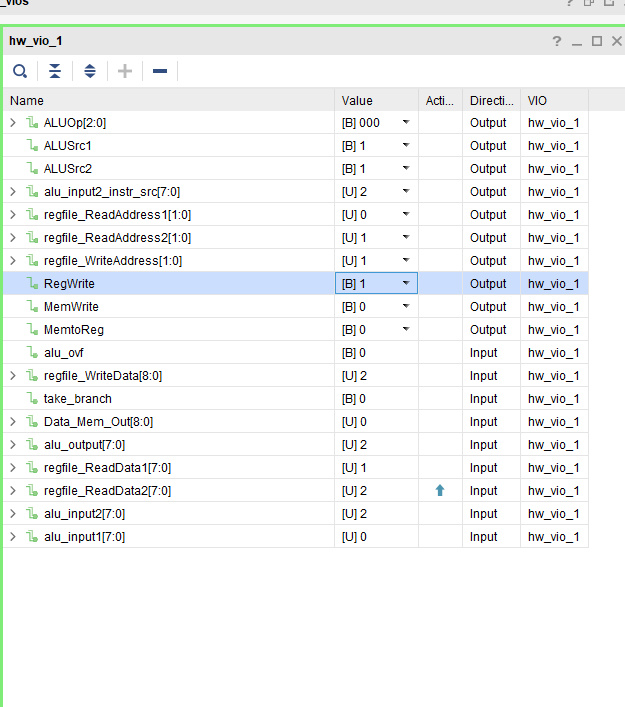
**writing 1 to Reg[1]**

****

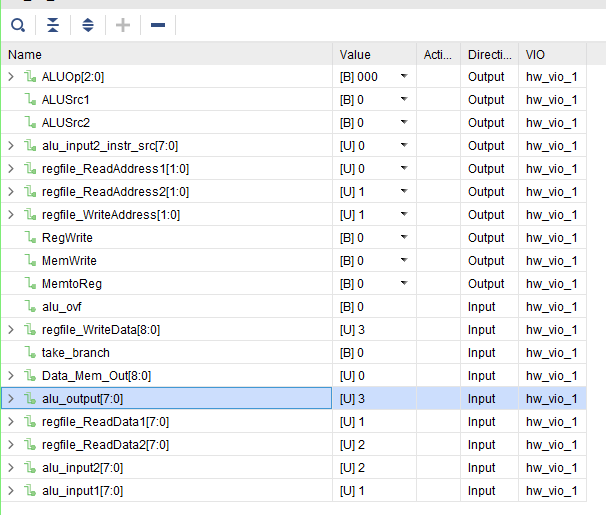
**verifying value in Reg[1] is equal to 1**

****

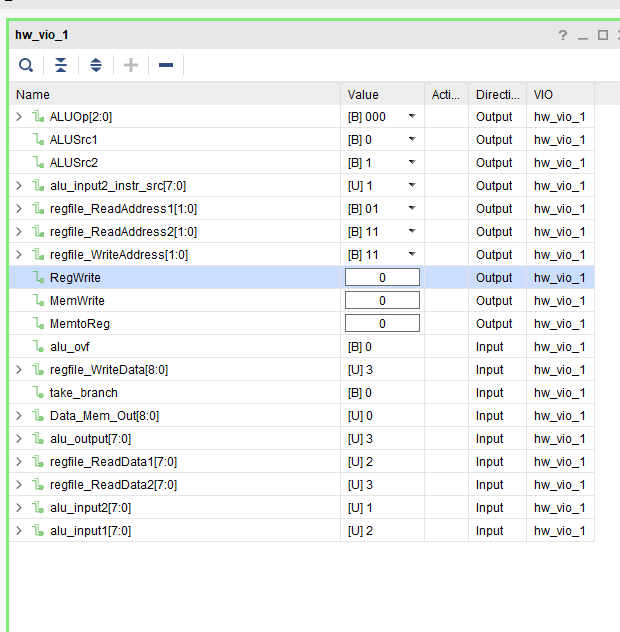
**writing 2 to Reg[2]**

****

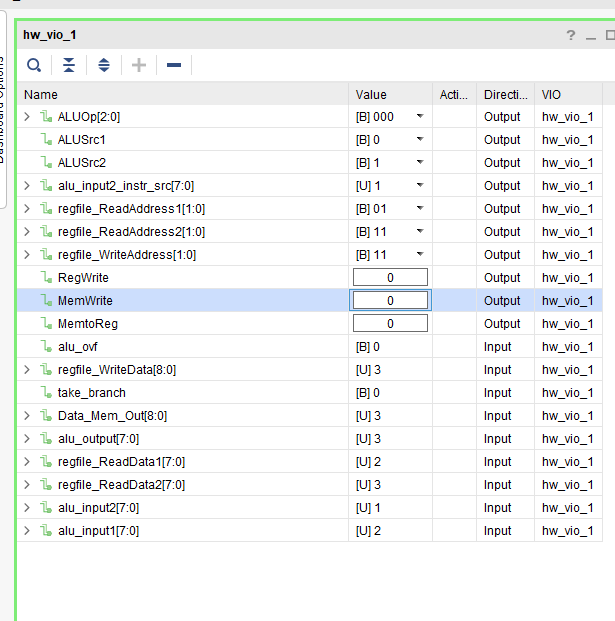
**verifying value in Reg[2] is equal to 2**

****

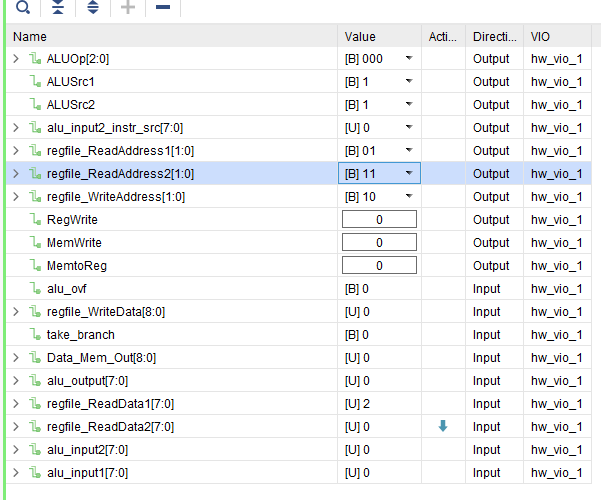
**Doing add operation (ALUOp = 000) and storing result in Reg[3]**

****

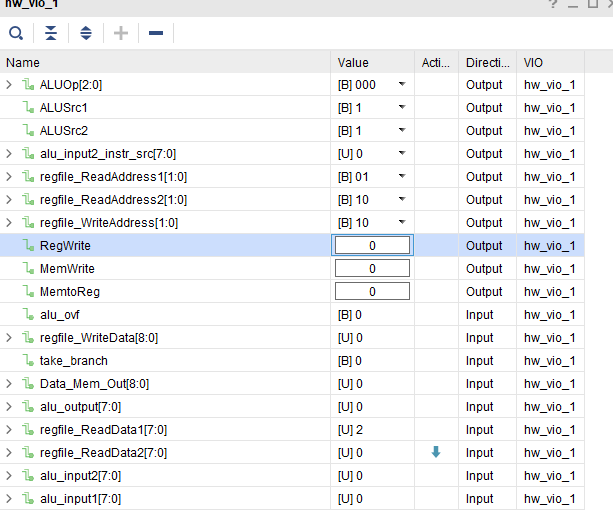
**Reading Reg[3] and seeing value is indeed 1 + 2.**

****

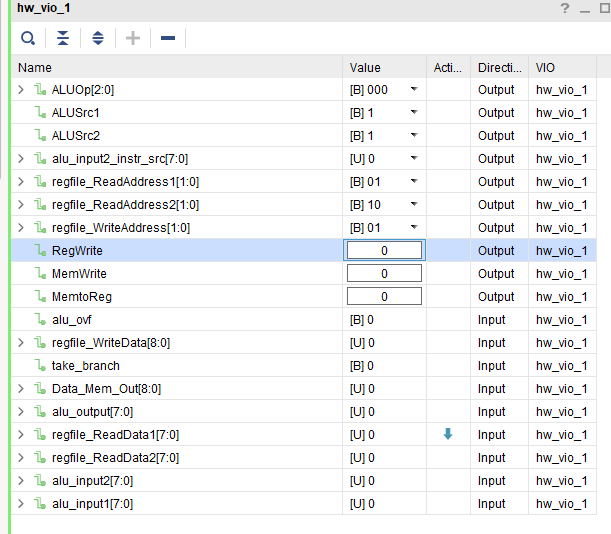
**storing Reg[3] to Mem[3]**

****

**clear Reg[3]**

****

**clear Reg[2]**

****

**clear Reg[1]**