CMPE 260 Laboratory Exercise 4
Execute Stage

Glenn Vodra
Performed: March 14, 2023
Submitted: March 28, 2023

Lab Section: 3
Instructor Sutradhar
TA: Ian Chasse
Cole Johnson
Robbie Riviere
Jonathan Russo
Colin Vo

Lecture Section: 1
Professor Cliver

By submitting this report, you attest that you neither have given nor have received any assistance (including writing, collecting data, plotting figures, tables or graphs, or using previous student reports as a reference), and you further, acknowledge that giving or receiving such assistance will result in a failing grade for this course.

Abstract

The purpose of this exercise was to design the execute stage of a MIPS processor. This design featured one main component, the Arithmetic Logic Unit, as well as some control logic. The ALU design takes previous functionality and adds additional pieces. The first of these is a ripple carry adder subtractor. This is capable of performing one of the two operations on the two input values based on a few control signals. The second piece of functionality added to the ALU is the multiplier. This design is capable of taking two values (each of which is half the size of the normal signals) and performing multiplication. This updated ALU, in addition to the minor control logic in this stage, was tested with several test benches, at various levels in the design. This exercise was successful because the execution stage of this processor design was capable of reading control logic and performing ALU operations.

Design Methodology

The first important step for understanding the design of the execute stage was to look at a high-level, abstracted view of the ALU and full execute stage. The first of these the ALU features two values, used as inputs. It also has a control line, that is used to control a MUX, determining what ALU operation is performed. In addition, there is a single output line that carries the output of the ALU to the next part of the design. This is shown in figure one.

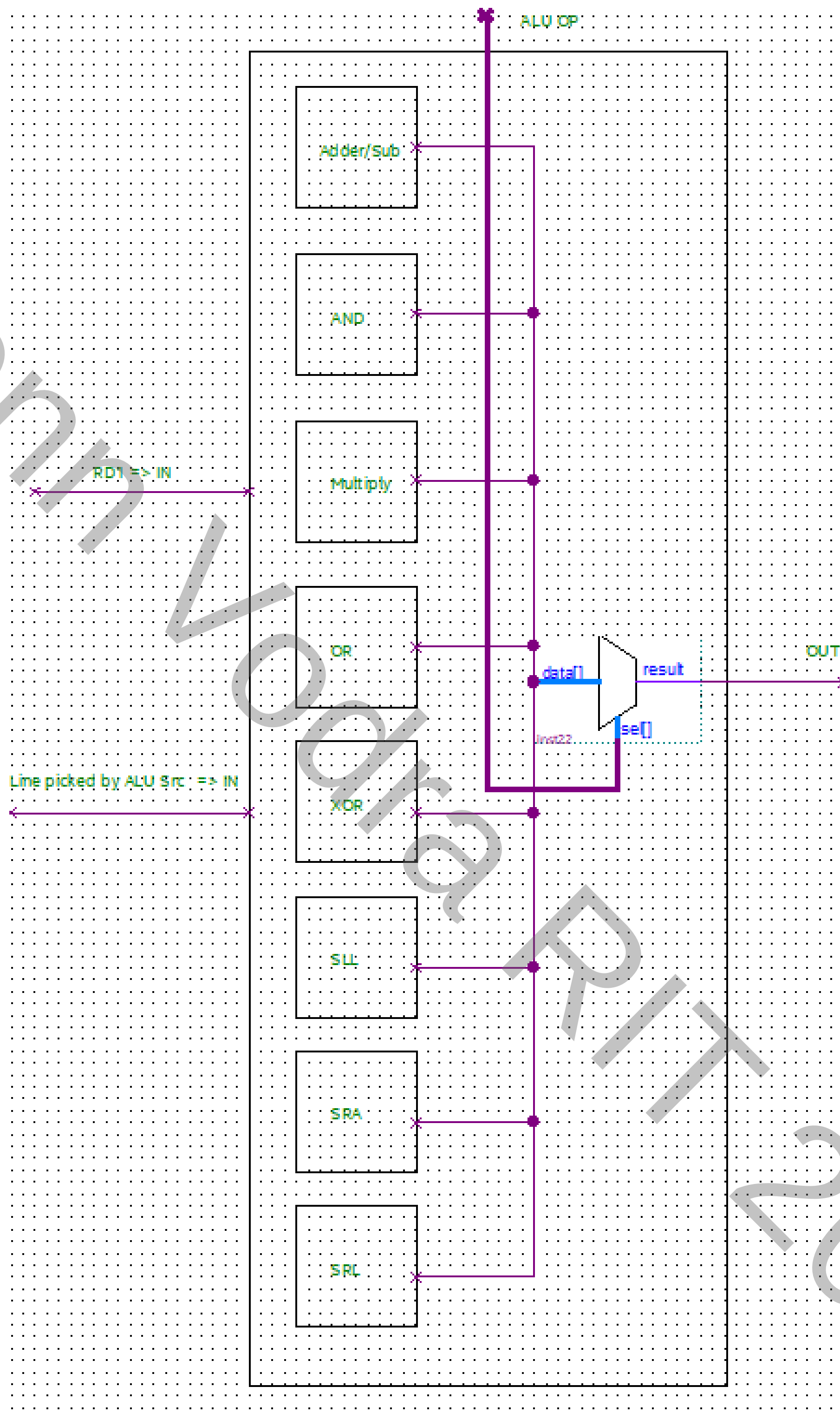

Figure 1: *ALU Block Diagram*

In the above figure, the first input will always be the value read from a register. The second ALU operand is the result of a MUX that selects between the value of a second register, and a thirty-two-bit immediate value. It is also important to explain that the output signal of the ALU is determined by the control signal ALUOP, which is four bits. The values of ALUOP will be explained below in the context of the ALUControl signal.

The second design that was elaborated on for this exercise was the execute stage as a whole. The execute stage is a block that holds all the control logic and the ALU shown in figure one. The execute stage is shown in figure two.

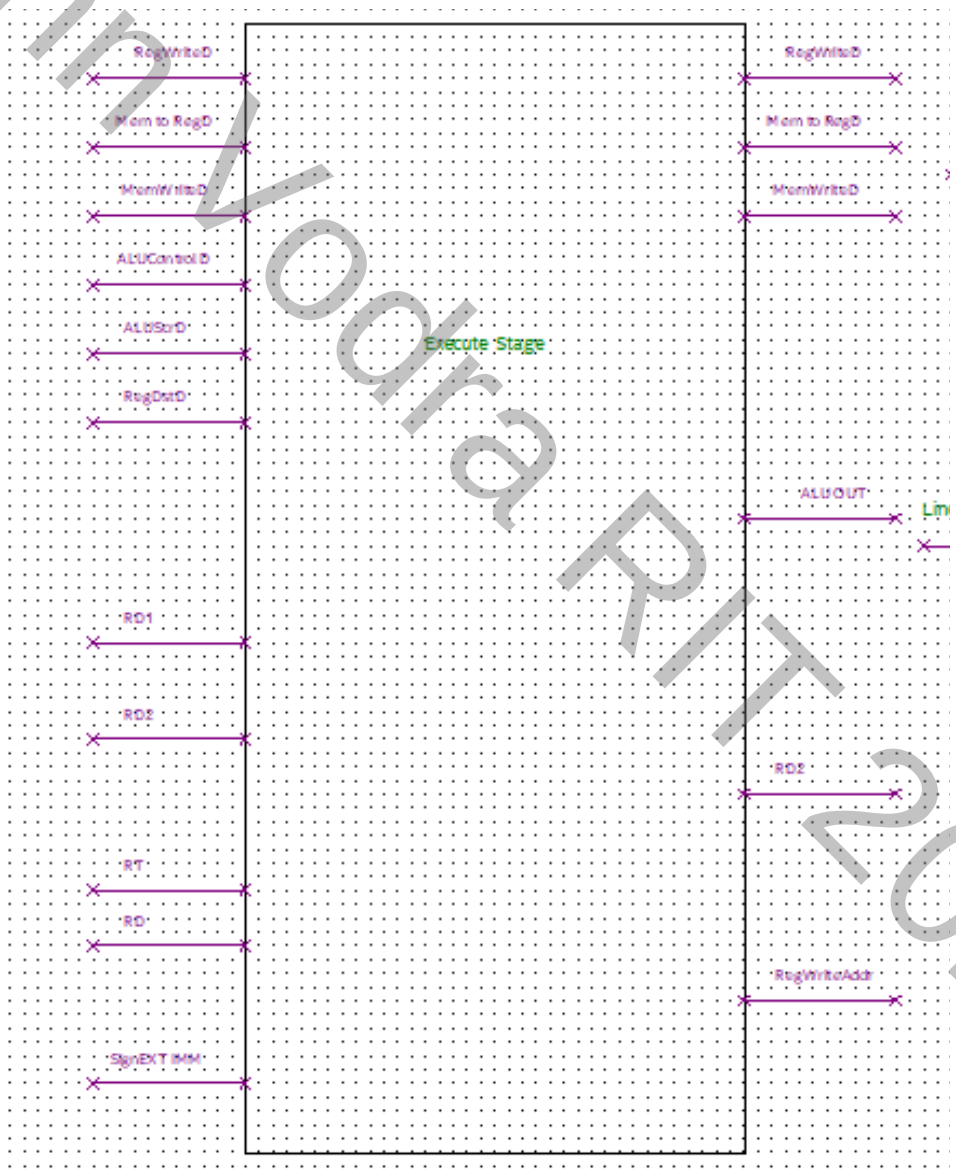

Figure 2: *Execute Stage Block Diagram*

The top six signals shown on the right of figure two, are all control logic signals. These are RegWrite, MemToReg, MemWriteD, ALUControl, ALUSrc, and RegDest. The last three of these signals are used to control the logic in the execute stage, while the top three signals are passed-through to the next stage of the MIP processor design. RD1, and 2, are data coming from the register reads in the previous stage. Also from the previous stage, are RT, RD, and SignEXTImm. RD2 is passed-through, while RT and RD are chosen based on the control logic. ALU result is also assigned from the output of the ALU.

To implement the ALU, the following assignment was used to select what ALU result was sent to the output of the execute stage. The possible options for ALUControl, are shown in the following table, table one.

Table 1: *ALUControl options*

Function	ALUControl
Addition/Store-Load	0100
AND	1010
Multiplication	0110
OR	1000
XOR	1011
SLL	1100
SRA	1110
SRL	1101
Subtraction	0101

While all sixteen options aren't used by the output ALU, this list of operations, still provides plenty of mathematical and logical operations. For this exercise support for Addition, Subtraction, and Multiplication shown in Table one, was added. To reduce the design size, the Add and Sub functionality was handled by a single system. This system takes the last bit of ALUControl and uses it as a smaller control signal. If this bit is low normal addition is performed. If the bit is high, then the second operand is inverted, and this signal is used as a carry-in, into the first full adder. This design utilizes two's complements to implement both of these operations, with the same hardware.

For the design of the multiplier, a carry-save, the design was used. This design utilizes a combination of AND logical and operations, as well as additional. The first row of this design is all logical AND. The second row onward is a combination of AND and addition, taking some signals from the previous row to complete the computation. The output of the first adder of each row, as well as the result of the first AND in the very first row, became part of the product result. The rest of the product is found by taking the output of each adder on the final row, which is the same row as the length of the factors being multiplied.

Results and Analysis

To demonstrate the proper functionality of the ALU several simulations were performed. Below is the ALU behavioral simulation. It is very important to note that only the new functionality of addition, subtraction, and multiplication are clearly shown in the figure. In this simulation, the other ALU operations were tested, however, they are not pictured, as they have been previously explained. This simulation is shown in figure three.

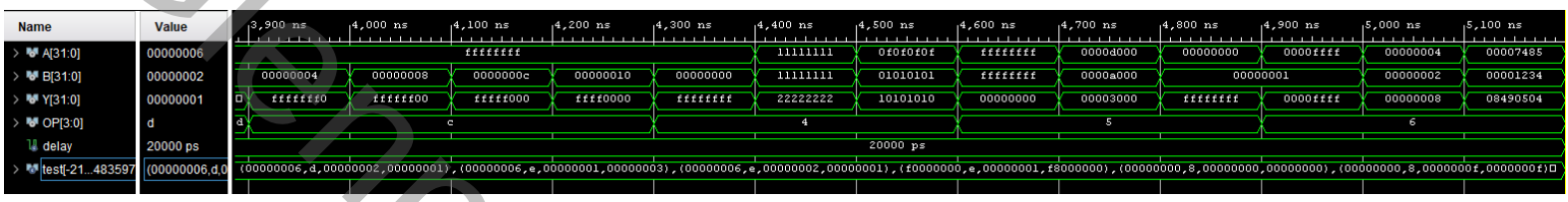

Figure 3: *ALU Behavioral Simulation*

In figure three, the test cases from 4,300ns to 5,200ns are test cases that were added specifically added to test the functionality of addition, subtraction, and multiplication. This is further shown by the OP signal, which is set to the values four, five, and six during this time. The following figure, figure four, shows the design was successfully run to completion, with no errors, even though the whole waveform isn't pictured in figure three.

```
restart
INFO: [Simtcl 6-17] Simulation restarted
run 5200 ns
Error: DONE
Time: 5200 ns  Iteration: 0  Process: /aluTB/data_proc
```

Figure 4: *TCL Output*

To further show the proper functionality of the design a test case was selected from figure three. The following test shows the result of an addition between two hex values. Figure five, shows "0x0F0F0F0F" + "0x01010101".

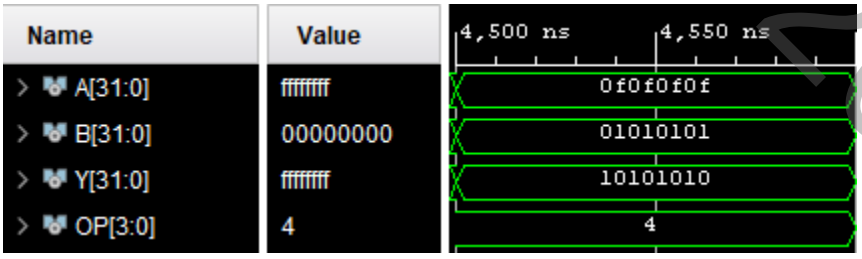

Figure 5: *Addition Test*

In figure five, the ALU OP is set to four, "0100", which is addition. In this test, the ripple carry functionality is shown as each hex value of "0xF" overflowed into the following digit. The result of this addition is "0x10101010", which is the value shown in figure five's Y (output), value.

In addition to a behavioral simulation, a post-implementation timing simulation was performed to ensure the proper functionally was achieved. The following figure, figure six, shows the result of this simulation. As before, the figure will only picture the newly added design functionality.

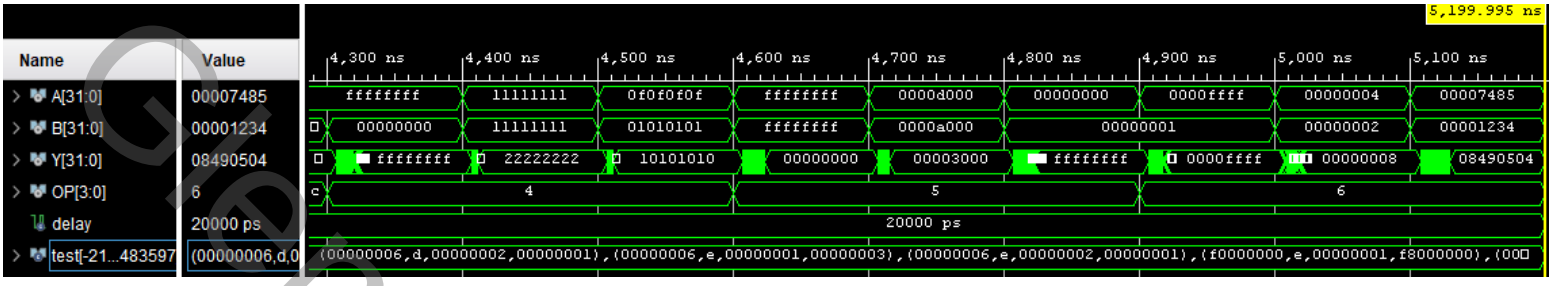

Figure 6: *Post-Implementation Timing Simulation*

This simulation is identical to figure three, with the exception of the added synthesized delay. Once again ALU OP for addition, subtraction, and multiplication are shown. These are still, four, “0100”, five, “0101”, and six, “0110”, respectively. The simulation also produced a TCL output that was identical to that of figure four. To ensure that the functionality is still correct, a test case was also examined closer, from figure six. This test case is a subtraction. The subtraction utilizes the same hardware as the addition, so it is important to show that the control logic is correctly implemented. Figure seven shows the result of a subtraction of “0xA”, from “0xD” in the fourth hex position.

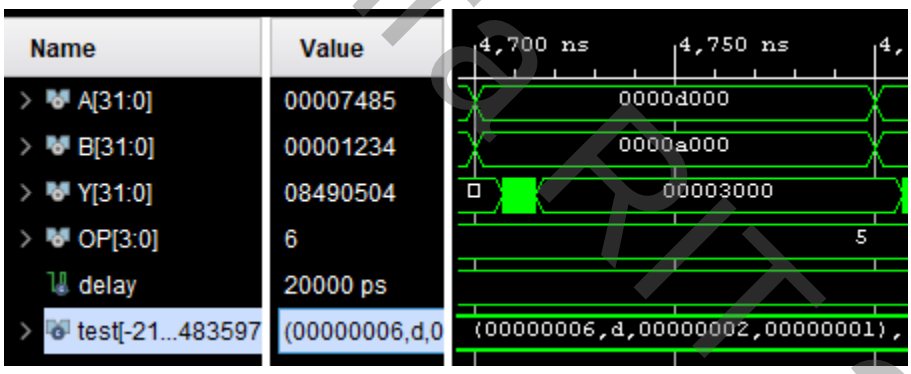

Figure 7: *Subtraction Test*

The result of this subtraction is “0x3”. It is important that while this subtraction is happening, the leading and trailing zeros are unaffected by this subtraction. If the value being subtracted was larger than “0xA”, then a negative number would be produced as the output. In this test case, however, this does not happen, and the result “0x00003000” is returned in the signal Y.

The arithmetic logic unit wasn’t the only part of the execute stage that was tested. The entire execute stage, made from control logic, the ALU, and pass-through signals were tested in the two simulation methods the ALU was tested. The following figure, figure eight, shows the result of a behavioral simulation of a fully completed execute stage.

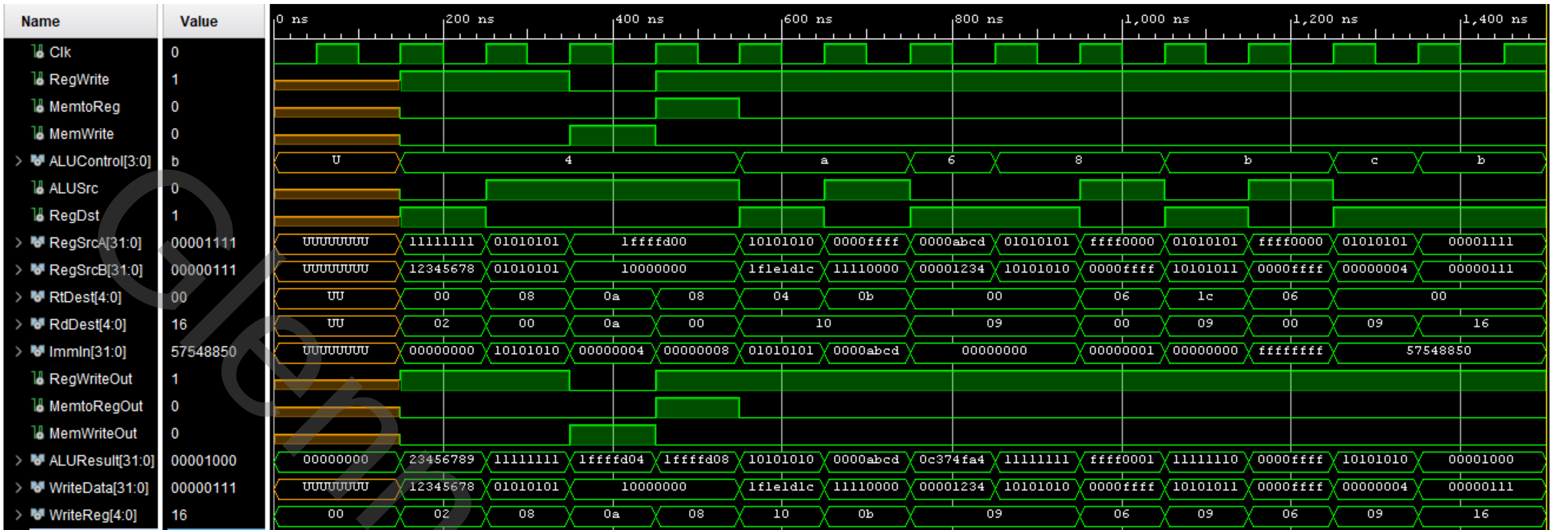

Figure 8: *Full Execute Stage Behavioral Simulation*

This behavioral simulation tested thirteen different test cases. These test cases include all of the ALU operations listed in table one (with the exception of SRA and SRL), as well as load-store, and a few I-type variants of the ALU operations. Some of the shifting instructions were ignored as they just involve selecting a different mux output, and this is properly working with the operations being used. In addition, these shifting functions were previously tested. In figure eight, all signals are toggled or changed at some point. Similar to the ALU testing, specific test cases will be highlighted to show the proper functionality of the design.

A test case that shows the proper functionality of many aspects of the design is the LOAD instruction. This test selects an immediate value from the end of an instruction and adds it to the value in the first register. This test also relies on control bits like MemtoReg to be passed through reliably, for future stages of the design. This execution is in figure nine.

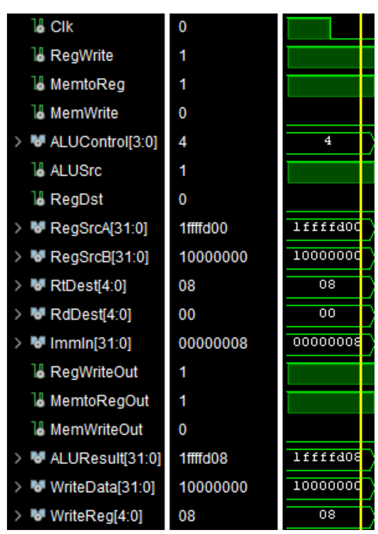

Figure 9: *Load Instruction*

In this test case three control signals, RegWrite, MemtoReg, and MemWrite need to be passed through, and they are, successfully. ALU Control/OP is four, which as described above is the add operation. An addition is needed to calculate the memory address to be read from. ALUSrc is '1' as it needs to select the immediate value. RegScrA and B are values from registers. Only RegScrA is needed as this is an I-type instruction. RT and RD are selected between; RT is picked for an I-type with the control signal RegDst set to '0'. The correct memory address "0x1FFFFD08" is calculated, and some additional data that is not needed for this operation is passed along as an output.

This execute stage was simulated again with the same test cases, however, this time it was run with a post-implementation timing simulation. This is shown in figure ten.

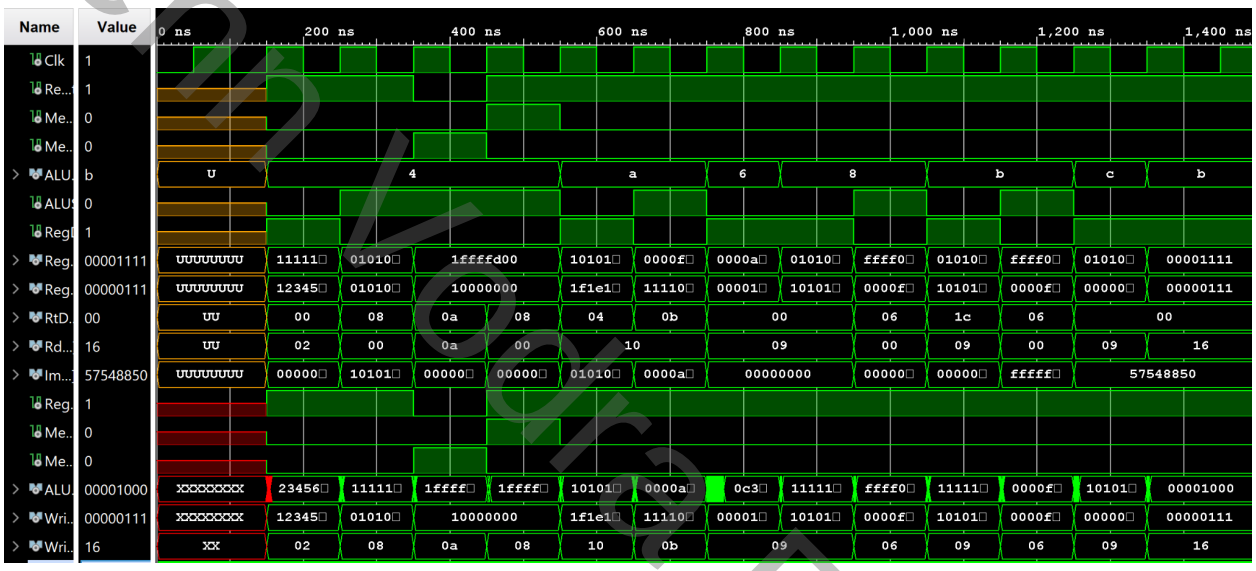

Figure 10: *Full Execute Stage Post-Implementation Timing Simulation*

This simulation behaves as expected. Initial test bench values are left undeclared, as the initial state of the processor will be unknown when the first instruction reaches the execute stage. Looking at one final test case this time to show that the execute stage worked after being simulated in post-implementation mode, figure eleven shows a multiplication instruction.

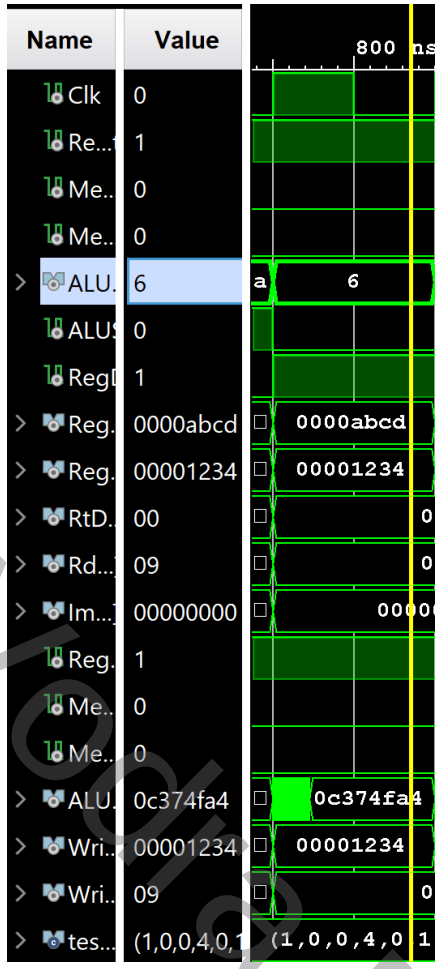

Figure 11: *Multiply Instruction*

For this instruction the most important thing to draw attention to, is the computation. The multiplication of “0xABCD” and “0x1234” is “0x0C374FA4”. This is the correct output, showing that the multiplication executed from the execute stage level was successful.

Conclusion

This exercise was successful because an execute stage was successfully designed and tested. There weren’t many components required for this exercise except the ALU, however, this part alone contained a lot of combinational logic. Even more specifically, the multiplier introduced a lot of propagation delay into the design. It is important to have balanced stages when pipelining a processor so that there isn’t wasted time in a particular stage. While this specifically wasn’t looked at for this exercise, it is important to keep it in mind. This execution stage performed as expected, completing ALU operations, selecting a few inputs and pass-through signal values, and just simply passing through values.