CMPE 260 Laboratory Exercise 4

Execute Stage

Andrei Tumbar
Performed: March 16th
Submitted: March 30th

Lab Section: 1
Instructor: Moskal
TA: Jacob Meyerson
Dennis Lam

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.

Your Signature: _____

Abstract

In this laboratory exercise, execute stage and the full ALU were implemented. The ALU was missing two vital operations: `MUL` and `ADD`. A ripple carry adder as well as an unsigned integer multiplier were implmented to complete these tasks. The execute stage was modeled to perform ALU functionality given inputs from the decode stage. This exercise was successful as it properly implemented the ALU subsystems as well as the execute stage and tested their functionality using test benches.

Design Methodology

ALU Block diagram

The ALU in this exercise is an extension of the ALU implemented in the first exercise. Here, the `ADD`, `SUB`, and `MUL` operations are implemented. A 4-select mux will choose between the 9 different operations made available by the ALU.
The `ADD` and `SUB` were implemented with the same hardware. `SUB` simply passed a subtraction operation to the ripple-carry adder implementation. The ripple-carry adder is a series of full adders where the carry-out from the previous full adder will feed into the carry-in of the next full adder. When subtraction was performed, a `1` was passed to the first full adder and all of the bits of the second operand were flipped to get the two's complement form of the number.
To implement the `MUL` operation, partial products placed into a table of bits. Each row would shift over product by one bit. Using this table, a product was calculated by summing up all of the rows using the ripple carry adder construct. Once added, the output would be the product. Inputs into the `MUL` circuit were half the size in bits that their output so that the output could not overflow.

A diagram was created to illustrate the functionality of the full ALU.

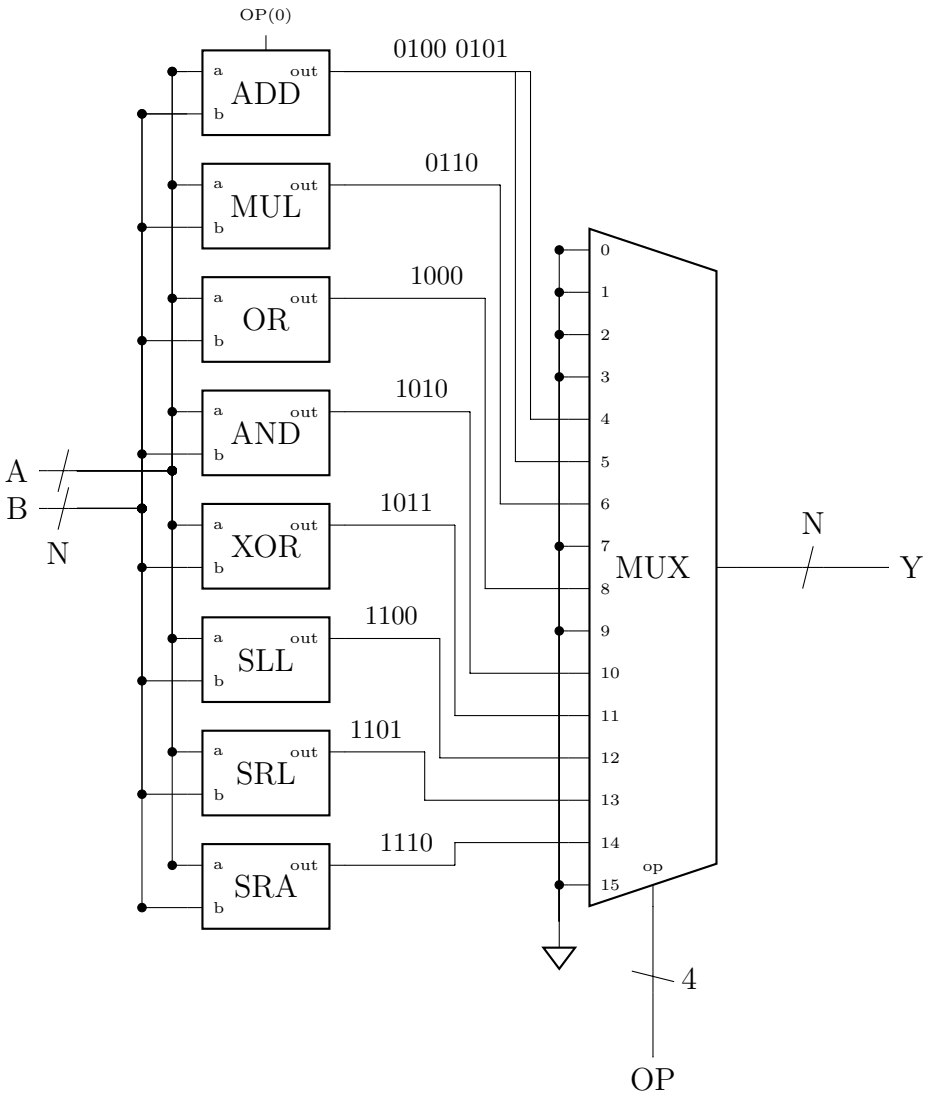

Figure 1: Layout of the N-bit ALU with nine operations

Figure 1 shows the functionality of the nine operation ALU. Notice that there is no explicit `SUB` component. The `SUB` operation is handled by the `ADD` operation by also passing a one bit op-code to the adder circuit. When the op-code is `1`, the operation being performed will be subtraction and vis-versa. Using this functionality the ALU operand `0101` will output the result from the subtraction operation while `0100` will output the addition operation's result. This is done to eliminate any redundant hardware that would be produced from creating two separate adders.

Execute Stage

The purpose of the execute stage is to consume inputs from the decode stage and perform ALU functionality. Many signals from the decode stage are unused and are simply output unchanged. These signals are known as "passthroughs". The execute stage will feed inputs into the ALU and output the ALU result. It will also select between the destination registers `RtDest` and `RdDest` given a control signal `RegDst`.

A block diagram was created to illustrate the functionality of the execute stage.

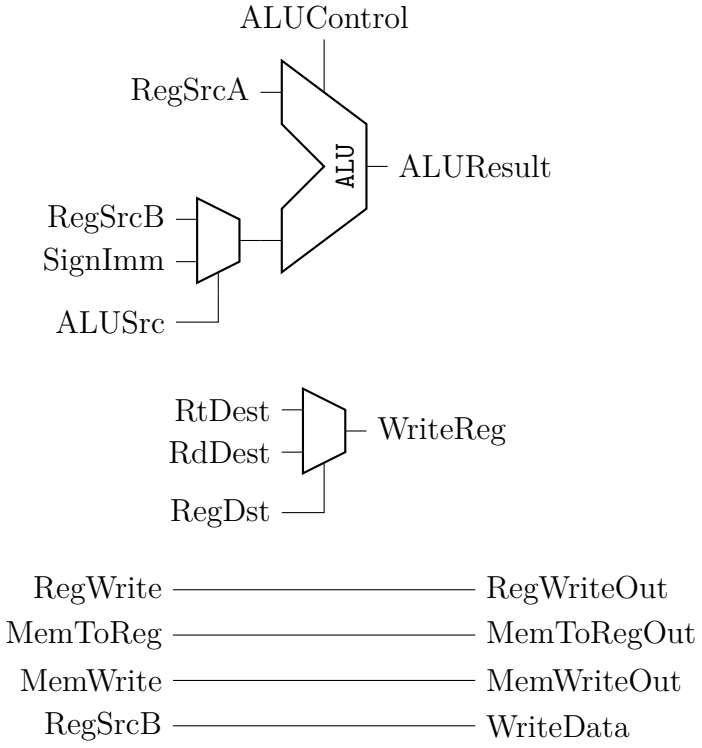

Figure 2: Block diagram of the Execute Stage

Figure 2 shows the general functionality of the execute state. Most of the heavy lifting in this stage is handled by the ALU. This diagram will simply select between two different ALU inputs as well between two different destination registers to write to. The bottom four signals are simply passthrough signals.

Results & Analysis

ALU

To test the ALU and the Execute Stage, a testbench for each one was written. The same input combination was passed to the ALU for each operation. A separate ALU was instantiated in the test bench to test all of the operations in parallel. Expected outputs of the ALU were defined for each operation and asserted against to verify their outputs. A table of the inputs and expected outputs is shown.

Table 1: Expected outputs of integer operations

A	B	ADD	SUB	MUL
0xA	0x2	0xC	0x8	0x14
0xABCDEF	0x4	0xABCDEF16	0xABCDEF0E	0x0003BC48
0xABCDEF	0x8	0xABCDEF1A	0xABCDEF0A	0x00077890
0xAB	0x7	0xB2	0xA4	0x4AD

Table 2: Expected outputs of logical operations

A	B	OR	AND	XOR
0xA	0x2	0xA	0x2	0x8
0xABCDEF	0x4	0xABCDEF16	0x0	0xABCDEF16
0xABCDEF	0x8	0xABCDEF1A	0x0	0xABCDEF1A
0xAB	0x7	0xAF	0x3	0xAC

Table 3: Expected outputs of shift operations

A	B	SLL	SRA	SRL
0xA	0x2	0x28	0x2	0x2
0xABCDEF	0x4	0xBCDEF120	0xFABCDEF1	0x0ABCDEF1
0xABCDEF	0x8	0xCDEF1200	0xFFABCDEF	0x00ABCDEF
0xAB	0x7	0x5580	0x1	0x1

The ALU test cases are shown in Tables 1, 2, and 3. All expected outputs are checked for each ALU operations and testcases and will trip an assertion failure in the testbench if they are to fail. A waveform was generated using this input data.

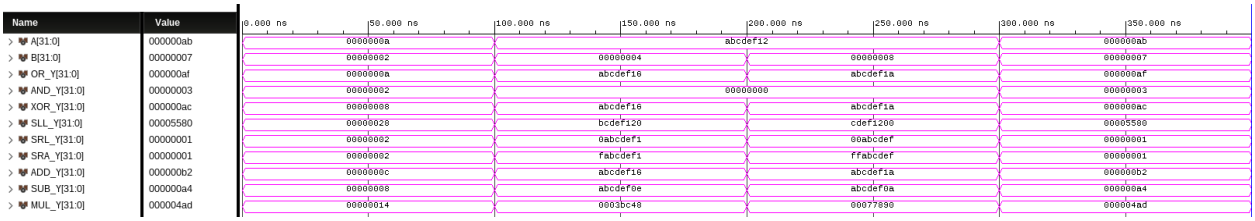

Figure 3: Full ALU behavioural waveform

Figure 3 shows the same 4 input cases for the each operations and the expected outputs in the tables above. The ALU is correctly implemented as there are no discrepencies between the expected outputs and the real outputs. When a working behavioural waveform was generated, a post implementation waveform was also generated to show the ALU implementation with timing taken into account.

Figure 4: Full ALU post-implementation timing waveform

Figure 4 shows the timing and delays of each operation given four sets of inputs. It is important to note here that the multiplication circuit has a significant delay because the circuit is made up of a chain of ripple-carry adders which in turn are a chain of full adders. This complexity will cause a large delay in calculating multiplication outputs.

Execute Stage

To test the execute stage, one simply needs to test a small subset of the operations of the ALU as well as the execute stage's select signals. A table is shown to illustrate the nature of the execute stage's test bench.

Table 4: Inputs and expected outputs of Execute Stage

RegSrcA	RegSrcB	SignImm	ALUSrc	ALUOp	ALUResult
0x0	0x3	-	0	ADD	0x3
0x2	-	0xffffffff	1	ADD	0x1
0xFF	-	0x8	1	SLL	0xFF00
0xABCD	0xFFFF	-	0	MUL	0xabcc5433

Table 4 shows the inputs and expected output of the execute stage. All of the passthrough

output signals shown in Figure 2 are automatically tested by passing an alternating signals to the inputs and verifying the outputs are the same. Notice how SignImm and RegSrcB are marked with – or dont-care when the ALUSrc is not selecting them as an input to the ALU. This test only runs a subset of the valid ALU operations because its purpose is to test only the Execute stage. Waveforms were generated for the testcases outlines in Table 4.

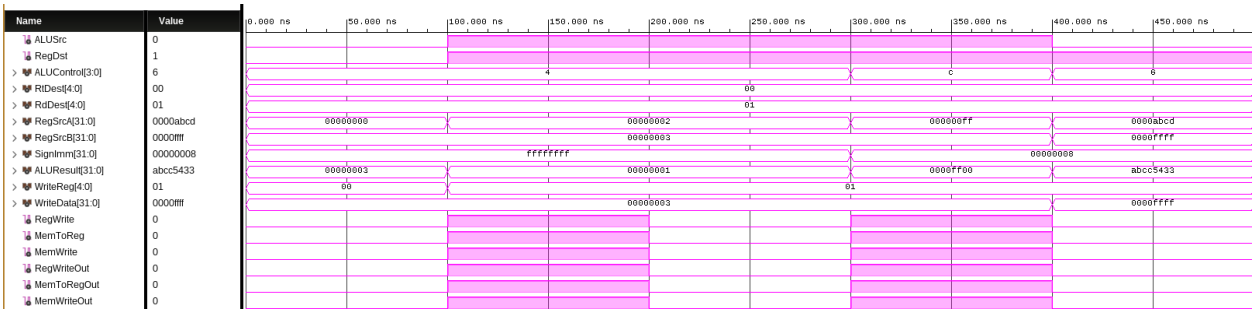

Figure 5: Full Execute Stage behavioural waveform

A post-implementation timing simulation was also run for the execute stage to show the delays associated with this component.

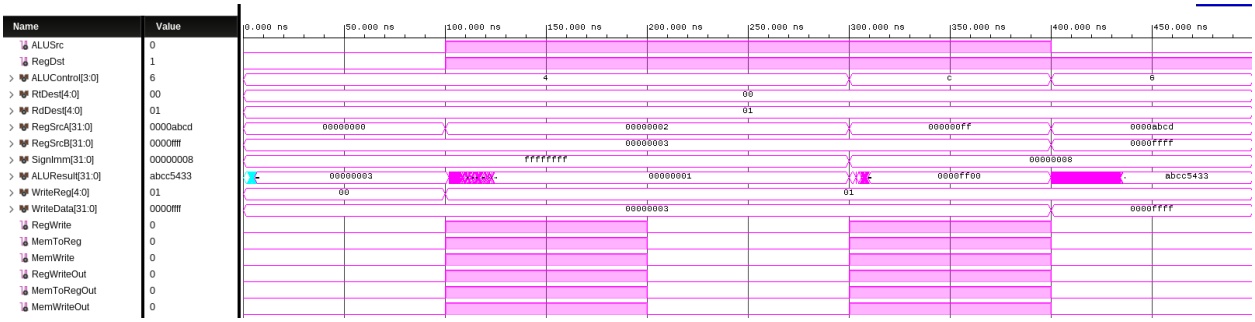

Figure 6: Full Execute Stage post-implementation timing waveform

The delays shown in Figure 6 are slightly longer than the ones shown in Figure 5 because there is extra logic to select between the ALU inputs. Because of this long delay seen in the execute stage, the clock period may not be shorter than about 30ns and the execute stage should be placed in its own clock cycle in the pipeline. This is because any shorter clock period will result in passing an unstable value through the pipeline for `MUL` operations.

Conclusion

This laboratory exercise the complete ALU along with the MIPS execute stage were implemented. To finish the ALU from exercise 1, a ripple carry adder and an unsigned integer multipier circuit were written. The execute stage provided some logic to select inputs into the ALU as well as passthrough signals given from the decode stage. This exercise was successful as it properly implementation the ALU and Execute stage with a proper corpus of tests written for each component.

Demo results

Part1

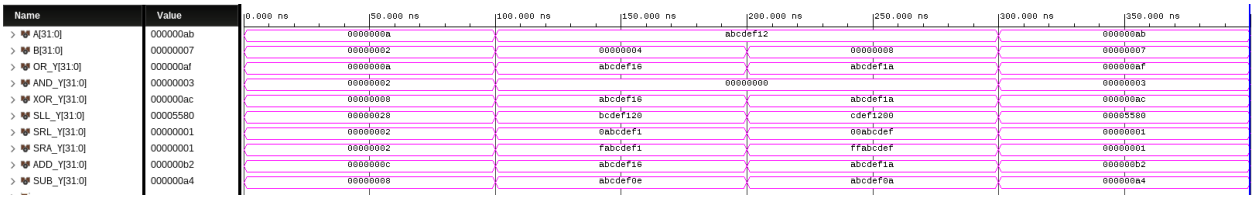

Figure 7: Behavioural simulation of ALU (without MUL)

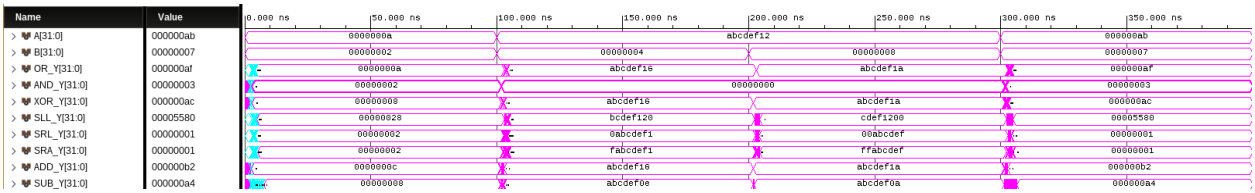

Figure 8: Post-synthesis timing simulation of ALU

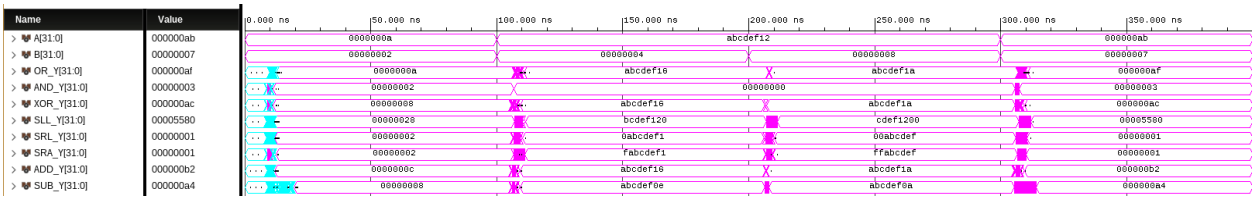

Figure 9: Post-implementation timing simulation of ALU

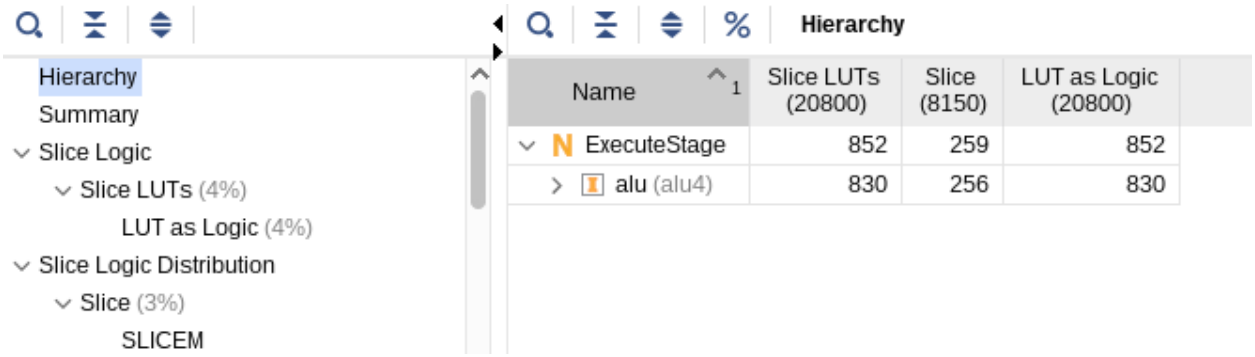

Figure 10: Utilization report on Baysis 3 board

Part2

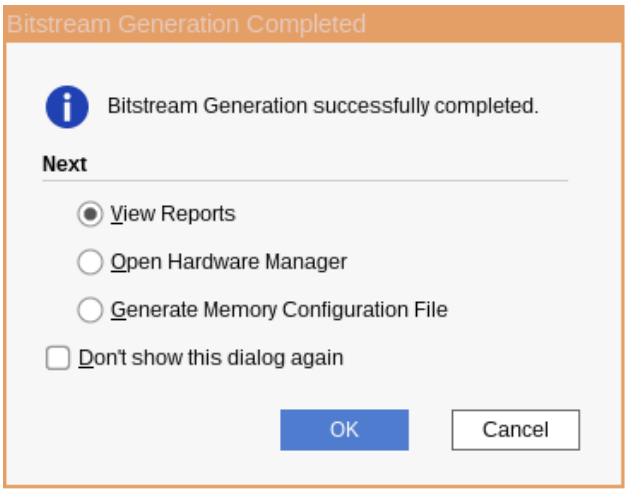

Figure 11: Bitstream successfully generated for 4-bit ALU

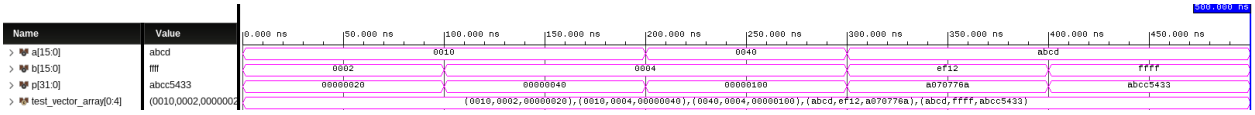

Figure 12: Behavioral simulation of multiply

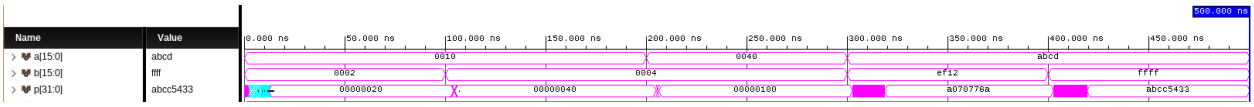

Figure 13: Post-synthesis timing simulation of multiply

Figure 14: Post-implementation timing simulation of multiply

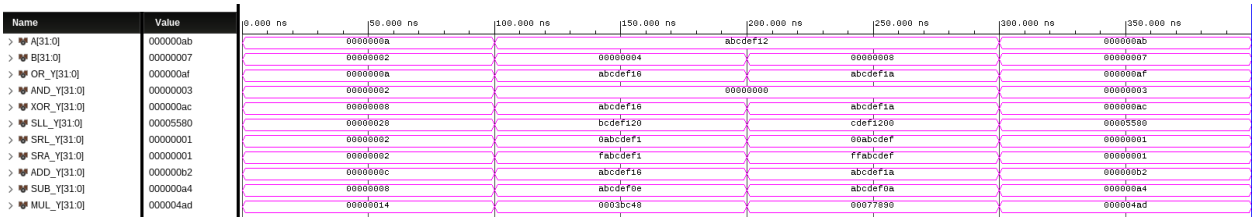

Figure 15: Full ALU behavioral simulation

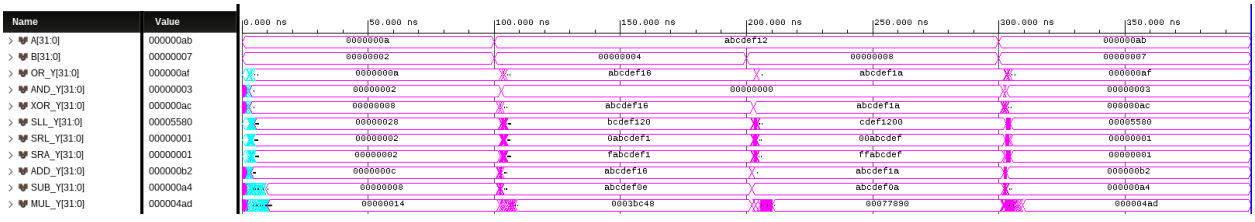

Figure 16: Post-synthesis timing simulation of full ALU

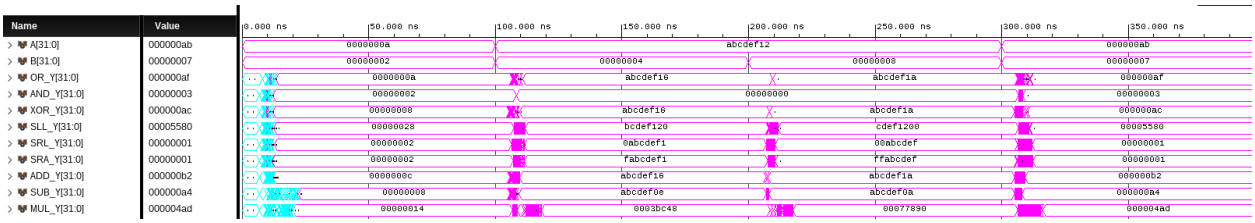

Figure 17: Post-implementation timing simulation of full ALU

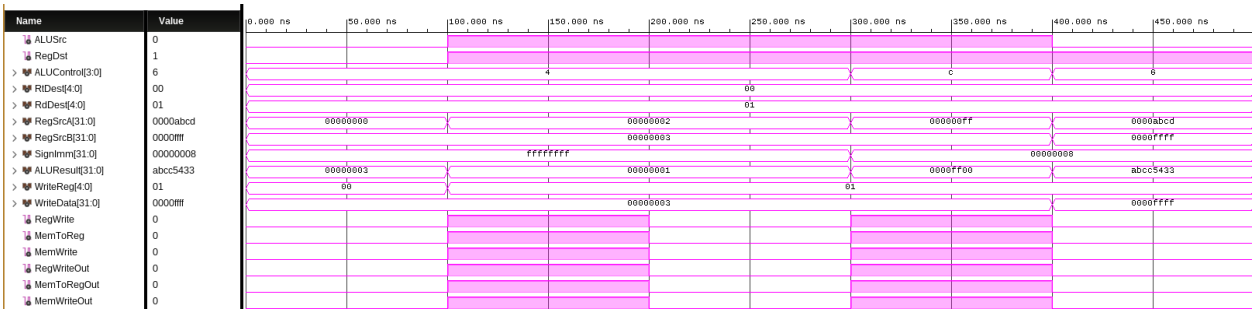

Figure 18: Complete Execute Stage behavioral simulation

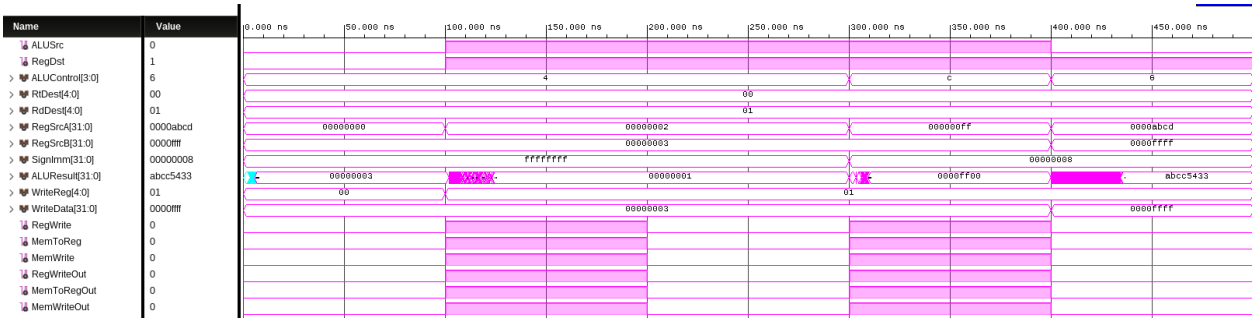

Figure 19: Complete Execute Stage post-implementation timing simulation