CMPE 260 Laboratory Exercise 3

Instruction Fetch and Decode Stages

Andrei Tumbar
Performed: March 2nd
Submitted: March 16th

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, the instruction fetch and decode stages were implemented for the MIPS architecture. The instruction fetch stage will read from instruction memory at the program counter (PC). The instruction decode stage is meant to take in an instruction and generate control signals as well as data inputs to the ALU and register file. This stage will drive the actions of the execute stage later on. The exercise was successful as the test-benches created to test the output of the fetch and decode stages found no errors in the implementation.

Design Methodology

Instruction Memory

Instruction Memory is a large read only block of memory built to hold instructions. The memory is byte addressable with an address width of 28-bits. This implementation of instruction memory can hold 1024 bytes. Any address above the maximum address of 1023 will output zeroes.

A read operation at an address will read an entire word (4 bytes). The memory was initialized in with bytes in the order that the test-bench expected them.

Instruction Fetch

The instruction fetch stage read from instruction memory at the program counter. The program counter is an address internal to the instruction fetch stage that will increment by 4 on every clock. The 4 byte increment is done so that the next instruction can be read.

Instruction Decode

The purpose of the instruction decode is to make sense of a certain instruction and generate certain control signals. Each control signal is sent to a later stage in the architecture to select various functionality.
The instruction decode stage holds various components inside of it. The register file is instantiated and controlled by the instruction decode stage. The control unit is implemented to simplify the work done by the decode stage and will generate various control signals given an Opcode and function.

Table 1 is shown to describe output signals and their purpose is shown.

Table 1: Instruction decode signals and their descriptions

Signal	Description
`RegWrite`	Set if writing to registers
`MemToReg`	Set if reading from memory
`MemWrite`	Set if writing to memory
`ALUControl`	4-bit ALU opcode
`ALU`	Set if second ALU operator is coming from immediate
`RegDst`	Determines which register to write to
`RD1`	Read data from address 1 in register file
`RD2`	Read data from address 2 in register file
`RtDest`	Register Rt in instruction (used to propagate destination)
`RdDest`	Register Rd in instruction
`ImmOut`	Sign extended immediate

The control unit implements most of the MIPS instructions including all of the R-type instructions and most of the I-type instructions. The missing I-type instructions are the non-word memory instructions (`lb`, `lh`, `sb` etc.).
One of the signals not fully explained in the table, `RegDst`, will select between `RtDest` and `RdDest` to write data to when the instruction has finished. The reason `RtDest` and `RdDest` are required to be outputed is because the data is not ready yet (instruction hasn't executed) and therefore we need to propagate the potential outputs to a later stage and select between the data down the line.
Another thing to note here is that the register file is falling-edge triggered. This means that the outputs `RD1` and `RD2` will not update until the falling edge where-as the other output signals are updated on the rising edge when the instruction comes in from the fetch stage.
The sign-extended immediate is used when performing I-type instructions. All I-type instructions will take in a 16-bit immediate signed number. This number needs to be extended to a 32-bit number to input into the ALU. To sign-extend, if the most significant bit is `1`, the extended bits should also be `1`.

Results & Analysis

Fetch stage

To test the implementation of the fetch stage, a test bench was written to check the output of the instruction fetch after every clock. The instruction memory is initialized with a sequence of 8 words followed by zeroes. The test-bench will reset the program counter asynchronously to start from `PC=0`.
The following words were initialized in order in instruction memory

Table 2: Instruction fetch test cases

Address	Value
0x00	0x11111111
0x04	0x22222222
0x08	0x1f2e3d4c
0x0C	0xaaaaaaaa
0x10	0xbbbbbbbb
0x14	0xcccccccc
0x18	0xdddddddd
0x1C	0xeeeeeeee

Behavioural, post-synthing timing, and post-implementation timing waveforms were generated.

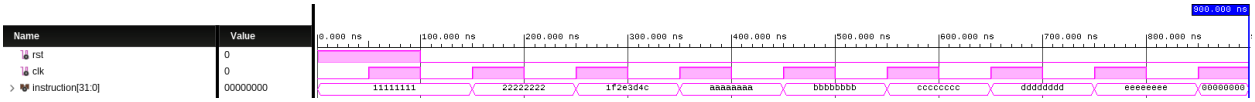

Figure 1: Decode stage behavioural simulation

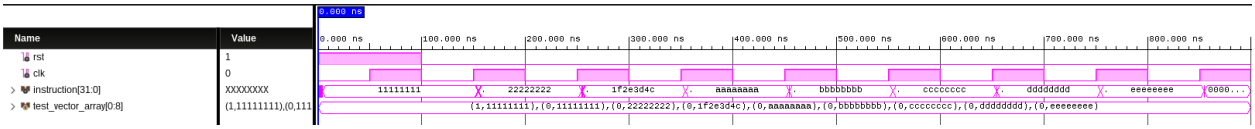

Figure 2: Decode stage post-synthensis timing simulation

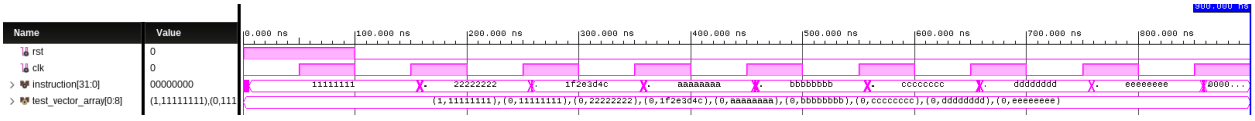

Figure 3: Decode stage post-implementation timing simulation

Figures 1, 2, and 3 correctly show the expected outputs noted in Table 2. These results are automicatally confirmed by the assertions placed in the test bench loop to verify results.
It is important to note that the timing delays associated with the Baysis-3 board were small enough to fit inside a single 100ns clock cycle.
The first instruction will stay active for two clock periods because the active high reset will keep the program counter at zero. This means that it is reading the first instructions twice which is expected.

Decode stage

The decode stage test bench involves choosing a set of instructions and verifying the output of the corresponding control signals.
Because the decode stage also takes in inputs for writing data to the registers. This data is output from a previous instruction and is passed to the decode stage via following stages.
The following instructions were passed to the decode testbench. The set of expected control signals is also noted.

Table 3: Instruction fetch test cases

Instruction	RegWrite	MemToReg	MemWrite	ALUControl	ALUSrc	RegDst
NOOP	1	0	0	1100	0	1
add	1	0	0	0100	0	1
addi	1	0	0	0100	1	0
ori	1	0	0	1000	1	0
sw	0	0	1	0100	1	0

The instruction without the operands are shown in Table 3. This is because the operands are not related to the expected control signal values shown in the table.
Using this input data, behavioural, post-synthesis timing, and post-implementation timing simulation waveforms were generated.
Assertions were placed inside the testing loop to verify that the output signals from the circuit were correct.

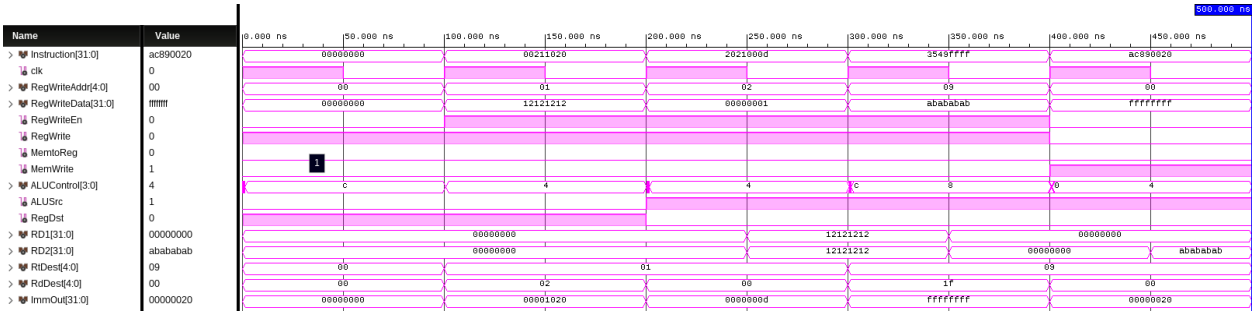

Figure 4: Behvioural simulation for decode stage

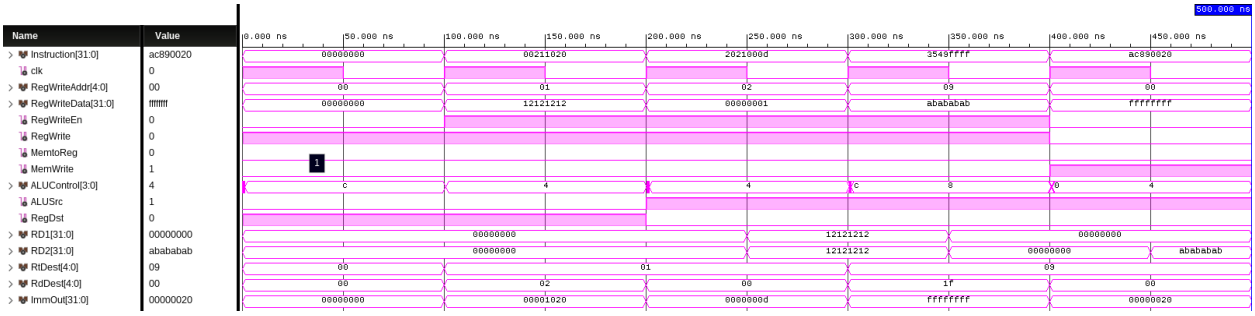

Figure 5: Post-synthesis timing simulation for decode stage

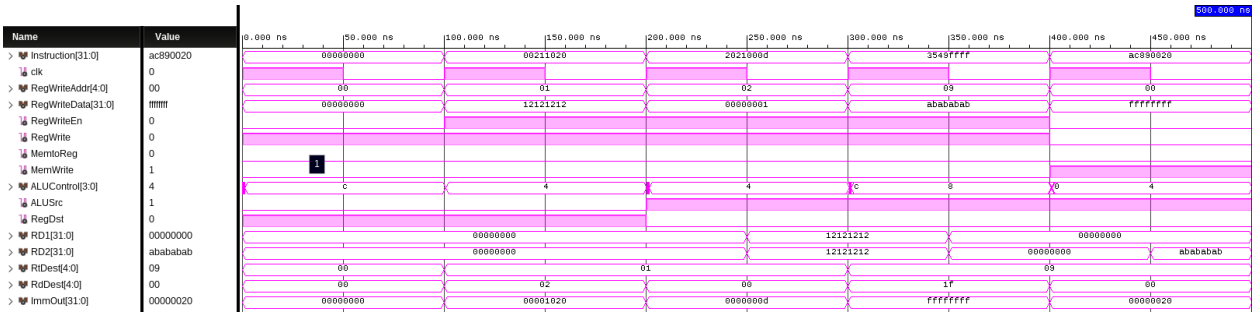

Figure 6: Post-implementation timing simulation for decode stage

Looking at the simulation results, the output signals are as expected and shown in Table 3. There is more being tested in this test-bench than depicted in the table. Read data from the register file is also asserted. All registers in the register file are initialized to zero by default. When certain instructions are executed with write data inputs, register data is overwritten. The final two instructions (`ori` and `sw`) will sequentially write to the `$t1` register followed by a read to verify its outputs. Because none of the assertions were tripped, the simulation resulted in success indicating a working implementation of the decode stage.
The sign extend functionality was also tested on the `ori` instruction. The full instruction was `ori $a0 $a1 0xffff`. The `0xffff` or `-1` would need to be sign-extended into `0xffffffff` when shown in 32-bit `ImmOut` signal. Looking at the second to last testcase, the `ImmOut` signal has the correct value.

Conclusion

This laboratory exercise introduced the concept of the instruction memory and implemented the instruction fetch and decode stages. and implemented a generically sized register file. The instruction fetch wired an internal program counter to keep track of the address of the current instruction and incremented the PC after every clock. The purpose of the decode stage was to break down the input instruction into control signals used by the execute stage later in the architecture. This exercise was successful as the design was fully tested and all issues found in testbenches were eliminated.

Demo results

Fetch

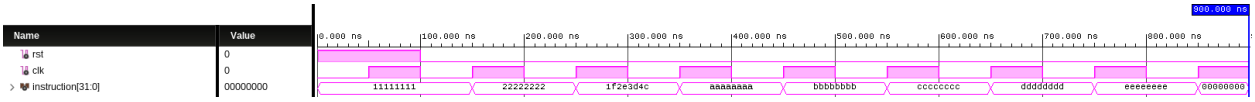

Figure 7: Behavioural simulation of Fetch

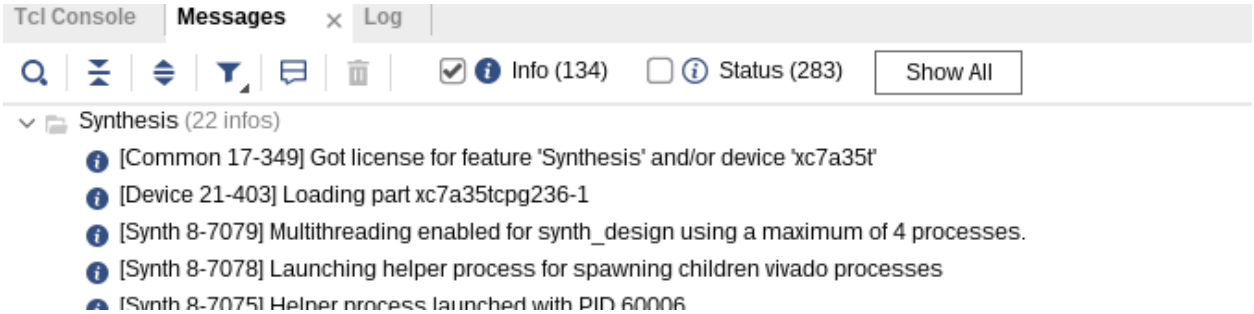

Figure 8: No warnings generated during synthesis

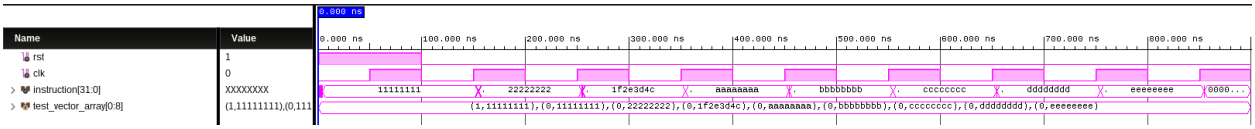

Figure 9: Post-synthesis timing simulation of fetch

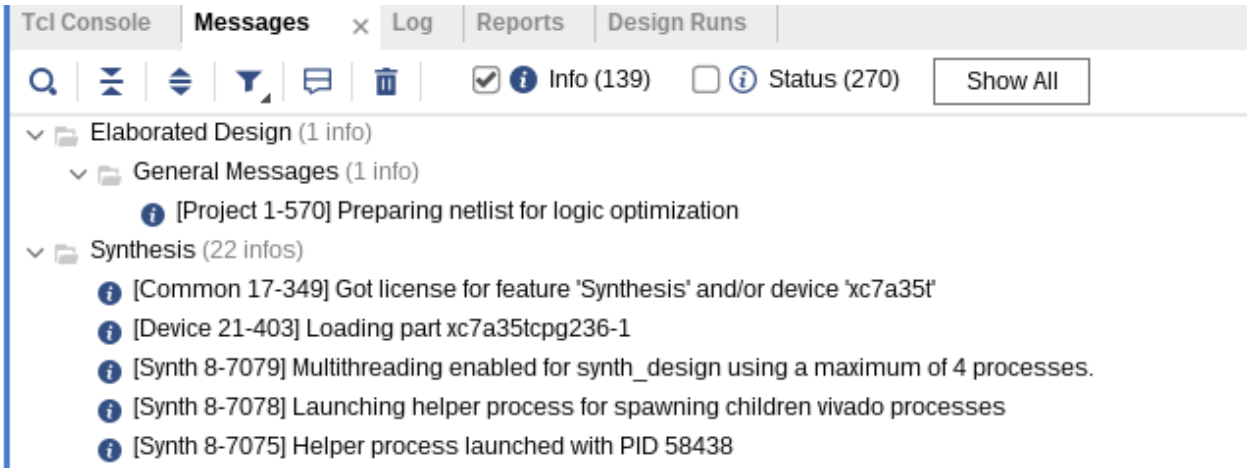

Figure 10: No warnings generated during implementation

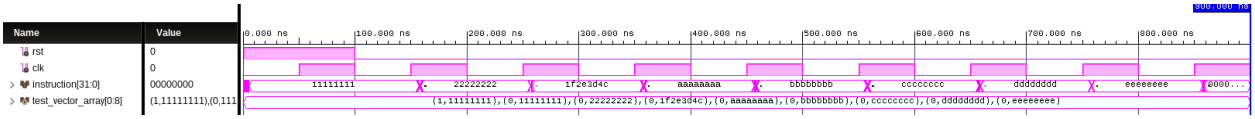

Figure 11: Post-implementation timing simulation of fetch

Decode

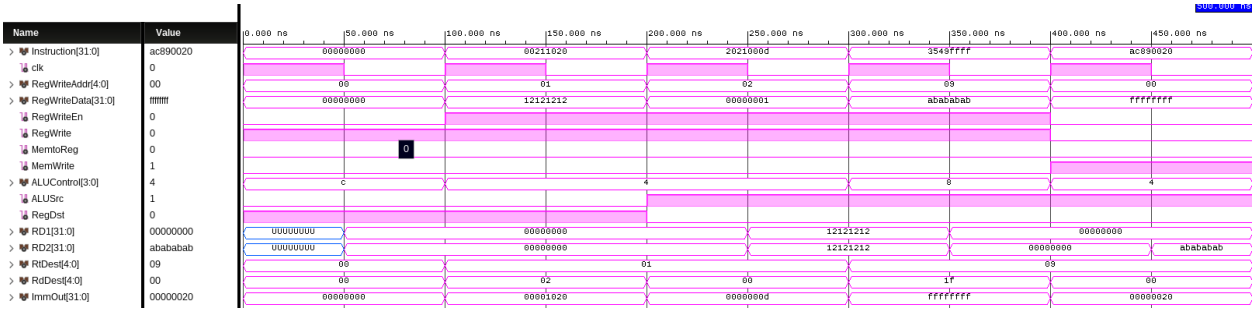

Figure 12: Behavioural simulation of decode

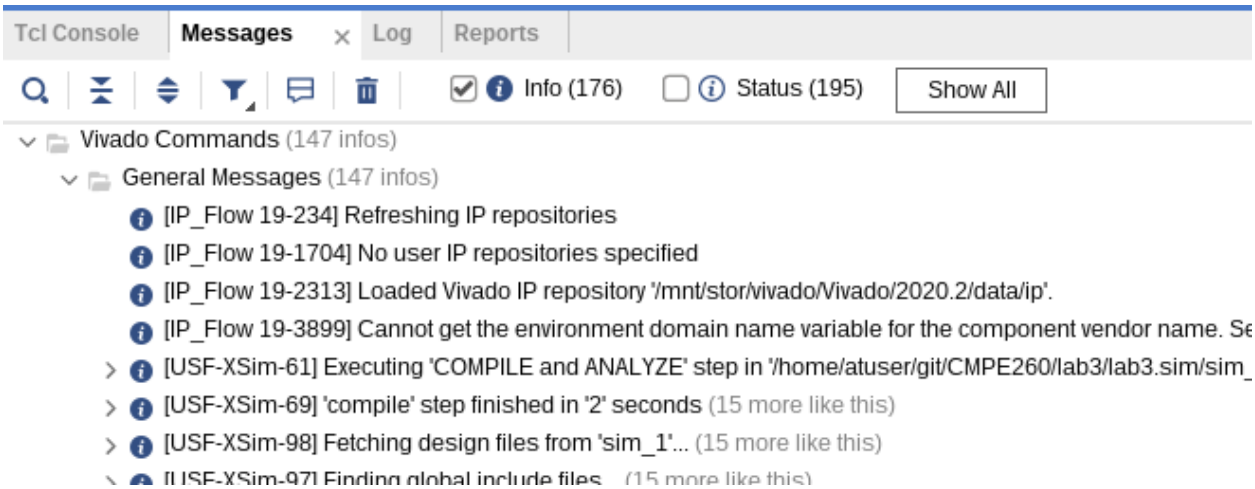

Figure 13: No warnings generated during synthesis

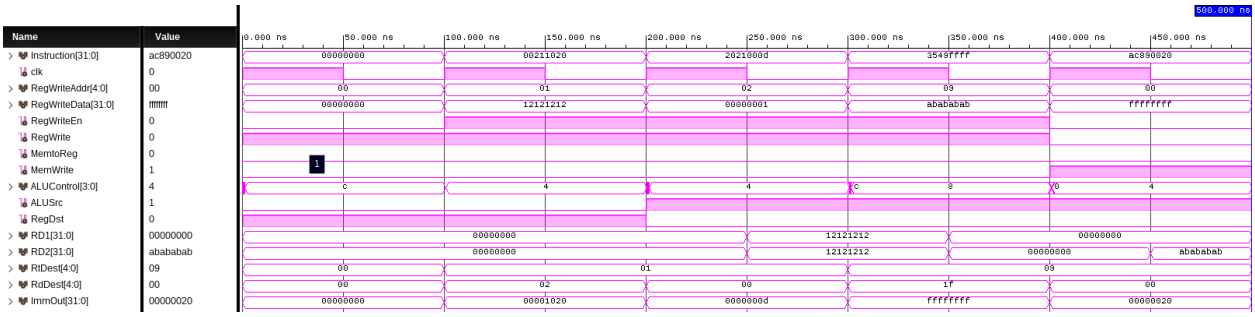

Figure 14: Post-synthesis timing simulation of decode

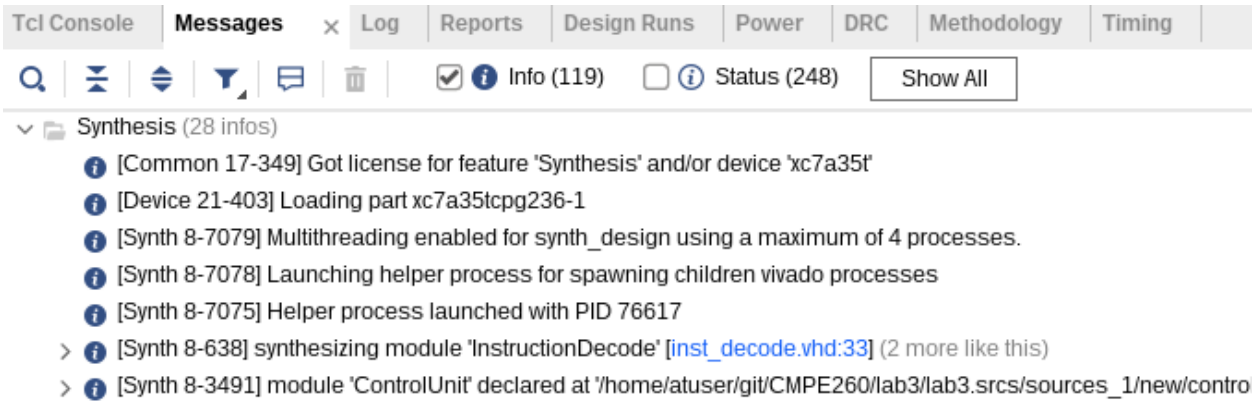

Figure 15: No warnings generated during implementation

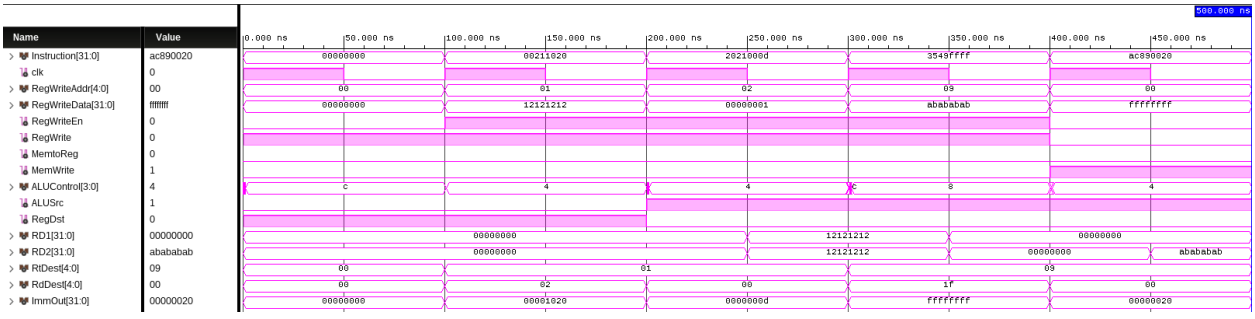

Figure 16: Post-implementation timing simulation of decode