Project 1: In-order MIPS Pipeline

Arjun Vinod, Tianrui Hu

January 24, 2022

1 Introduction

In this project, we build on the design of an in-order MIPS pipelined processor found in H&H, and implement it in Verilog – compiled and simulated using ModelSim.

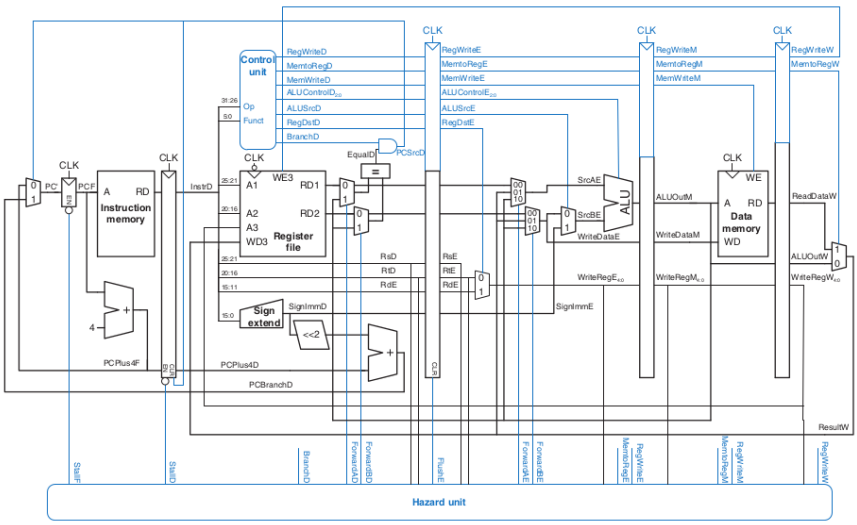


Figure 1: H&H MIPS Pipelined Processor Schematic

We will now discuss the design modifications made to each stage to satisfy the requirements of this project.

2 Design Specification

2.1 Fetch

In the fetch stage, the processor fetches and outputs the instruction from instruction memory that corresponds to the program counter value stored in a register: `instrF`. At each clock cycle, the register is updated with the next program counter value. The other output of this stage is `pcplus4F`, which is computed by adding the current `pc` by 4 using an adder.

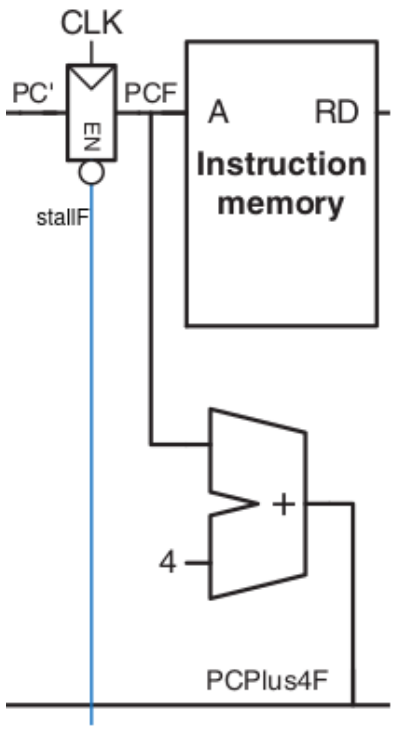


Figure 2: Pipeline Fetch Stage

2.2 Decode

In the decode stage, we had to make some modifications that will later support the implementation of the multiplier. To support the unsigned multiplication operation, we need to pass the zero-extended immediate to the execute stage, `UnsgnImmD`. In addition, we need another bit for `MemtoReg`, or `WBSrc` as labelled in our schematic, to be able to select `hi/lo` from the multiplier or the output from the ALU in the execute stage. A `MultStartD` signal indicates that the

decoded instruction initiates a multiplication and the `MultSgnD` signal indicates whether or not the multiplication needs to be signed (0: unsigned, 1:signed).

To support the jump instructions, we also add `jumpD` as a control signal that also gets wired to the writeback stage to choose the `jumpDstD` immediate for the next cycle's program counter. The `jumpD` signal is also wired to the clear of the pipeline register for the decode stage, so when it's high, the instruction fetched in this cycle is flushed.

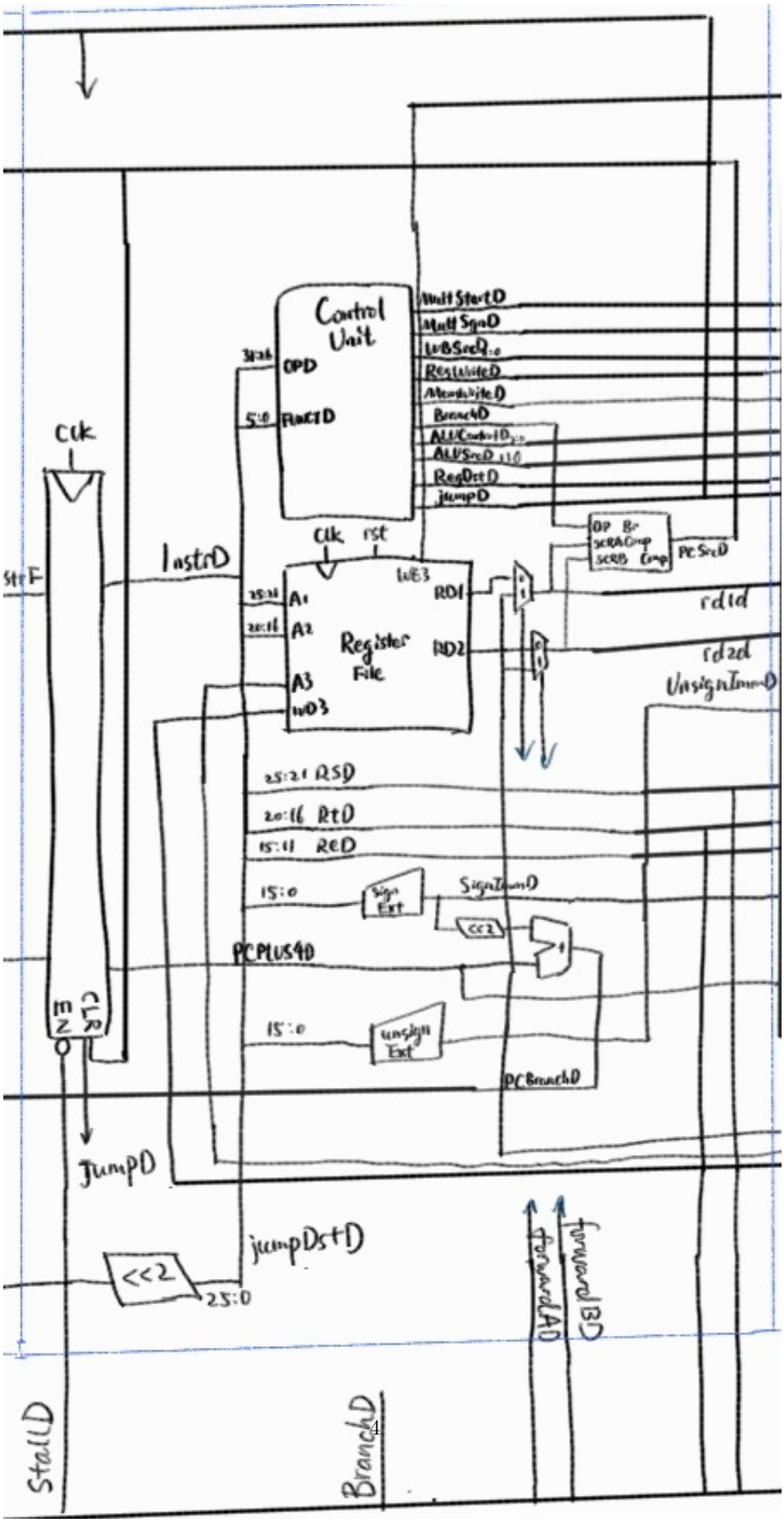


Figure 3: Pipeline Decode Stage

2.3 Execute

Compared to the H&H design, the execute stage has the brunt of the modifications.

2.3.1 ALU

The ALU originally only implemented 4 functions: add/subtract, and, or, and set less than. We utilize the remaining 3 unused control sequences to implement xor, xnor, and $\ll 16$. The first two are built-in operators in verilog. We implemented a hard-coded 16-bit left shift to support the LUI operation.

The inputs into the ALU are multiplexed between the inputs to the ALU as specified in the H&H design and a pair of outputs from the multiplier. The output of the ALU is also fed into the multiplier.

2.3.2 Multiplier

This design implements a shift and add multiplier as described in P&H [1] [2]. By using a shift and add register, we can reuse the 32-bit ALU's adder in the schematic instead of implementing a 64-bit bit adder that's otherwise necessary in a standard multiplier implementation.

The `mult_status` flag is an internal flag that is high during multiplication and low otherwise. For 32 cycles, the multiplier performs the shift and add algorithm while the decode and fetch stages are stalled. During this operation, noops are sent down the pipeline so that we can be sure nothing happens during the execution of the multiplier.

The second two bits of `WBSrcE` is used to select the output of the stage, relabeled as `ALUMultOut`. This output multiplexes `hi` and `lo` from the multiplier, and the output directly from the ALU.

[1] Computer Organization and Design RISC-V Edition: FIGURE 3.5 Refined version of the multiplication hardware

[2] Binary Multiply - Repeated Shift and Add

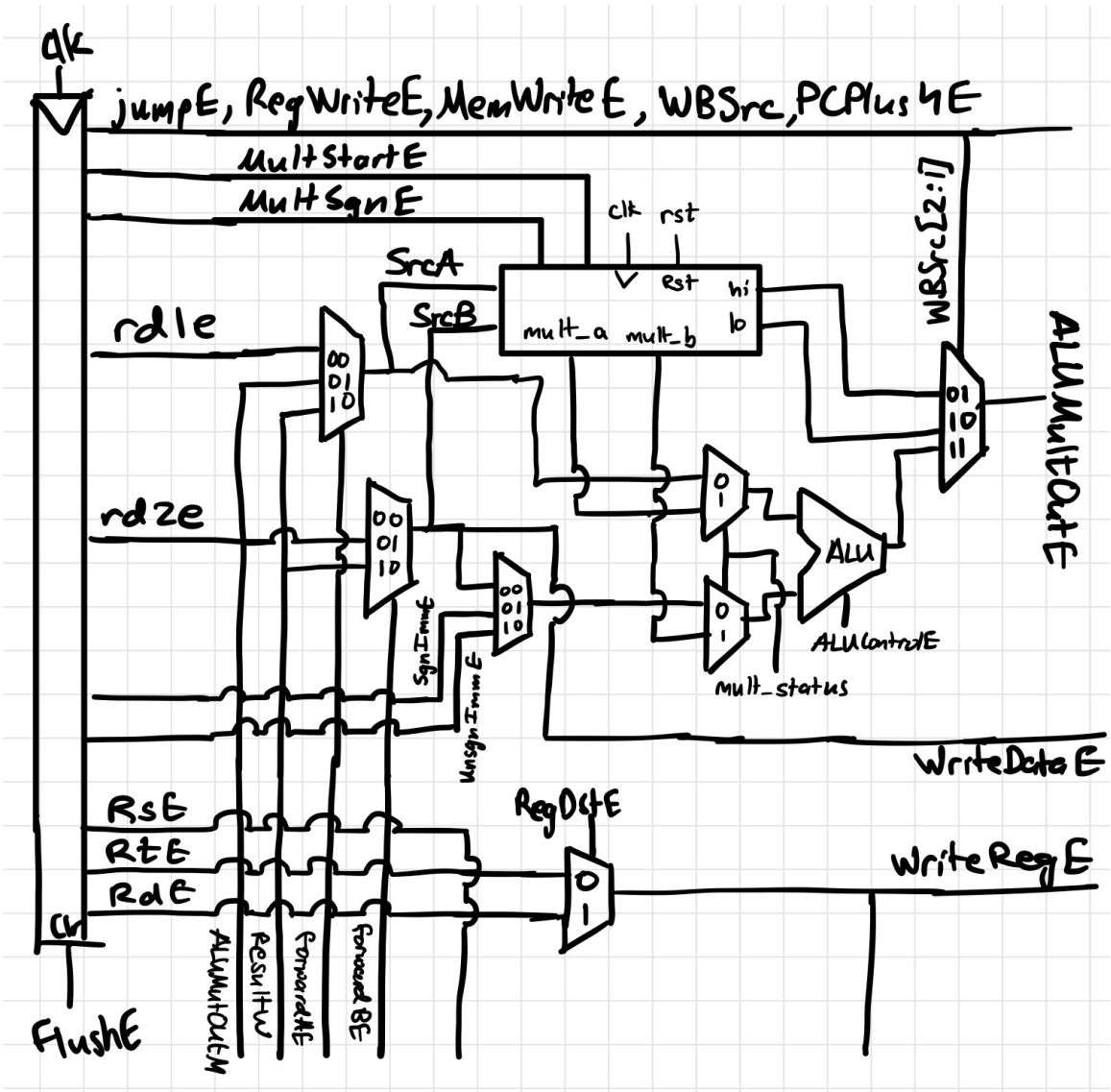


Figure 4: Pipeline Execute Stage

[3]

[3]Note that signals going down out of the image or coming up from the bottom of the figure for the execution stage are connections with the hazard unit.

2.4 Memory

The memory stage is largely the same as the original H&H design except for the addition of an adder to compute `PCPlus8M`. This is used to support the `jal` instruction so that the correct return address can be written to `$ra`.

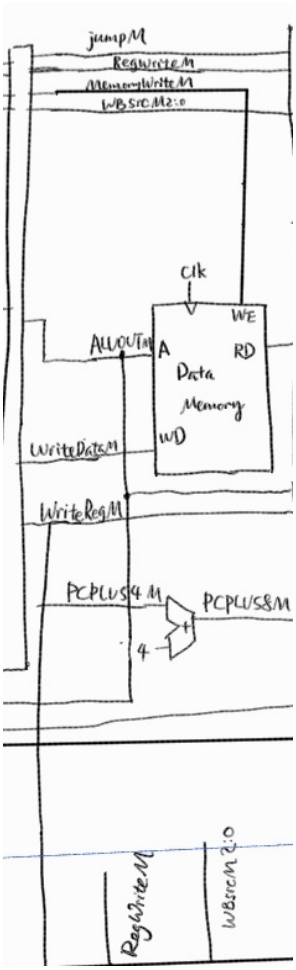


Figure 5: Pipeline Memory Stage

2.5 Writeback

The writeback stage is modified to support the jump instruction. The second two bits of `WBSrcW` is used to select between memory read out from data memory, `ALUMultOutW`, and `PCPlus8W`. The output of the first multiplexer is labeled `ResultW`. This output is wired to the decode stage if data is written to a register. It's also wired to the execute stage in the event of a data forward.

The `WriteRegW` flag is multiplexed between the `WriteReg` flag from the previous stage and the integer 31. The control signal of this multiplexer is `jumpW`, so if this signal is high, the register to write into is hardcoded into *ra*. The `jal` instruction is performed if `jumpW` is high and the `WBSrcW` bits multiplex `PCPlus8W` to `ResultW`.

Two multiplexers in the writeback stage pipe the value for the next program counter value. If decode stage indicates that a branch is successful with the `pcsrcD` flag, then `pc` is set to `pcbranchD`, which is computed in the decode stage. If the branch is unsuccessful, then `pc` depends on the other control flag, `jumpD`. If the jump signal is high, then the new program counter is computed by concatenating the immediate from the the jump instruction to the right of the first 4 bits of the current program counter. If the jump signal is low, then the next program counter is `PCPlus4F`.

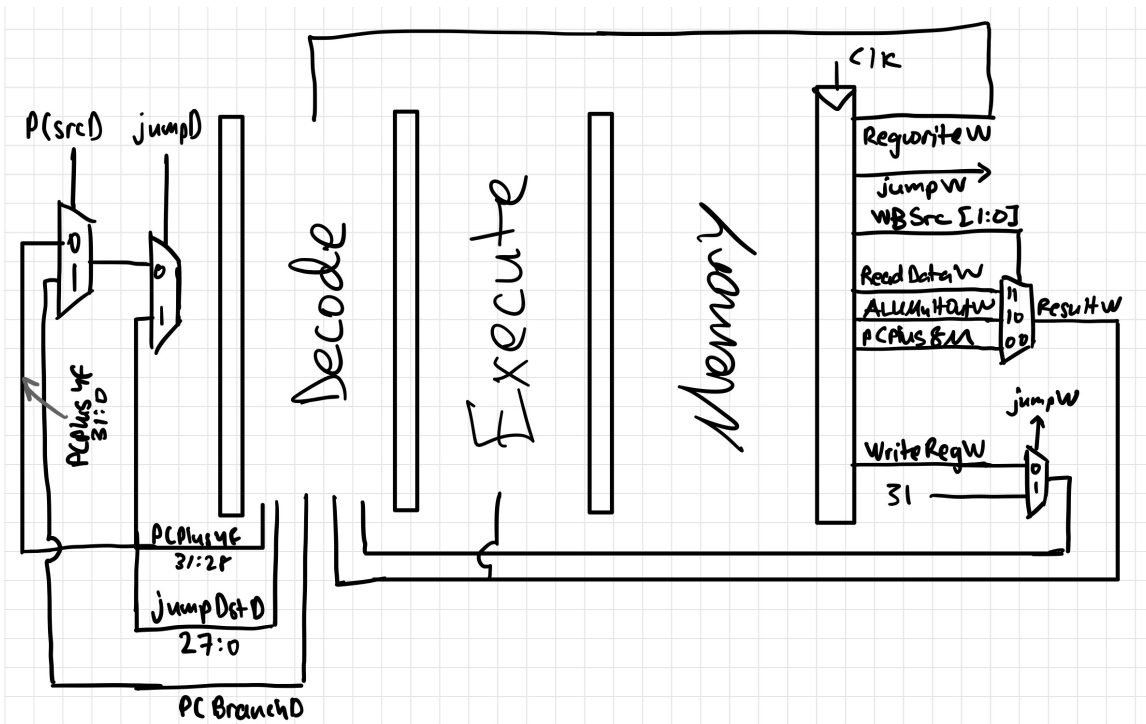


Figure 6: Pipeline Writeback Stage

2.6 Hazard Unit

The hazard unit is a completely combinational module that takes inputs from each stage and outputs the flags used in each stage asynchronously. We used a series of case-statements to determine each output flag for forwarding, stalling,

and flushing.

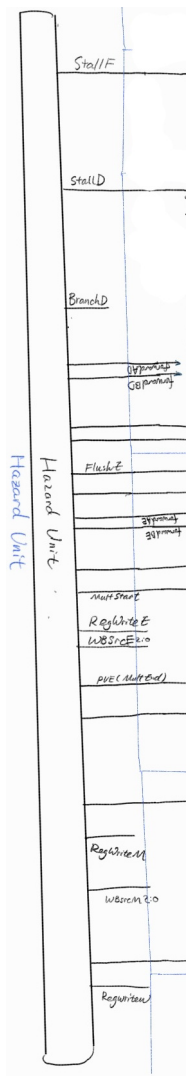


Figure 7: Pipeline Hazard Module

3 Testing

3.1 Unit Testing

Many of the components were extensively tested in previous labs, so we did not deem it necessary to spend time further testing them. Notable modules include

the instruction memory, register file, ALU, and data memory. In addition, being short on time and losing a group member, unit testing for each module and stage was largely limited to testing basic functionality. We prioritized the error-prone situations such as stalls, flushes, and the multiplication operation.

Instead of torture-testing the individual components, we aimed to put the modules together and get the pipeline working before extensive testing. We made up for the low bar set for unit testing by designing a final test program that we thought would push the processor with a perfect storm of hazards, stalls, flushes, forwarding, branching, jumping, and multiplication.

3.2 Stalls & Flushes

In the event of a stall and flush, each stage needs to respond properly to ensure the pipeline doesn't get stuck. The stall turned out to be relatively easy to implement by just checking at every rising edge of stall is high, and if it is, not updating the registers. To flush, we simply checked for a flush signal at the rising edge and passed a noop down the pipeline if the signal was high.

3.3 Multiplier

The most crucial component to test robust functionality for was the multiplier. Since it's a serial operation that takes multiple cycles, there are many opportunities for mistakes in timing that can translate to data not being ready when it's needed in the final pipeline. This was made easier by the fact that we're implementing a shift adder multiplier and relied on an ALU we were confident already worked. To test the multiplier, we inputted several binary values of various lengths.

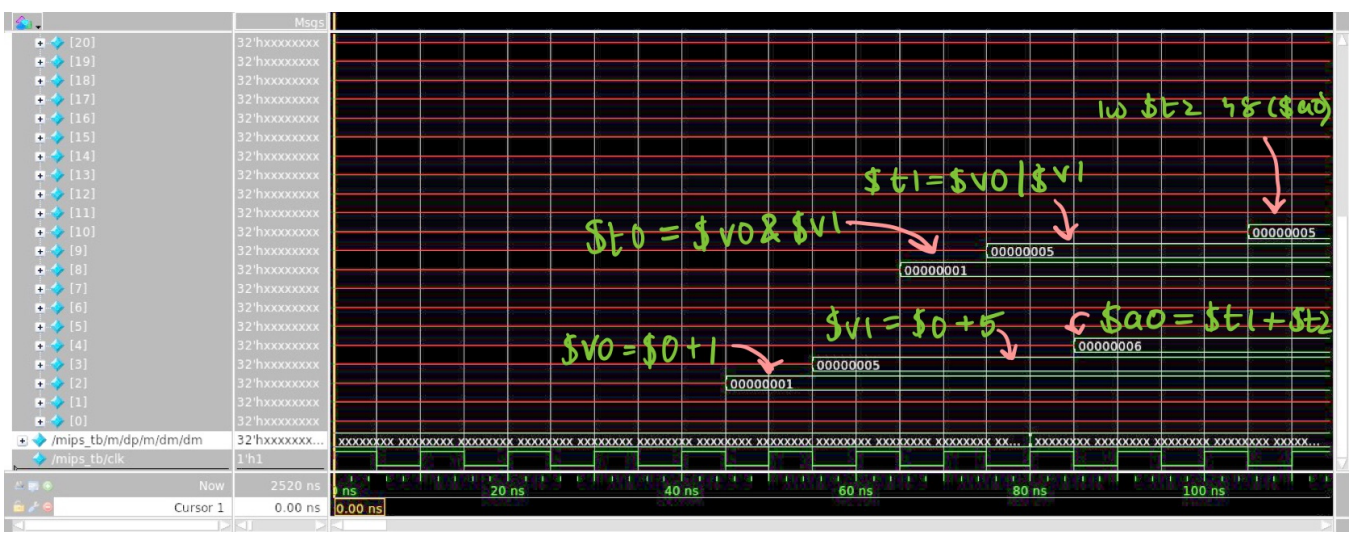
3.4 Final Project Manager

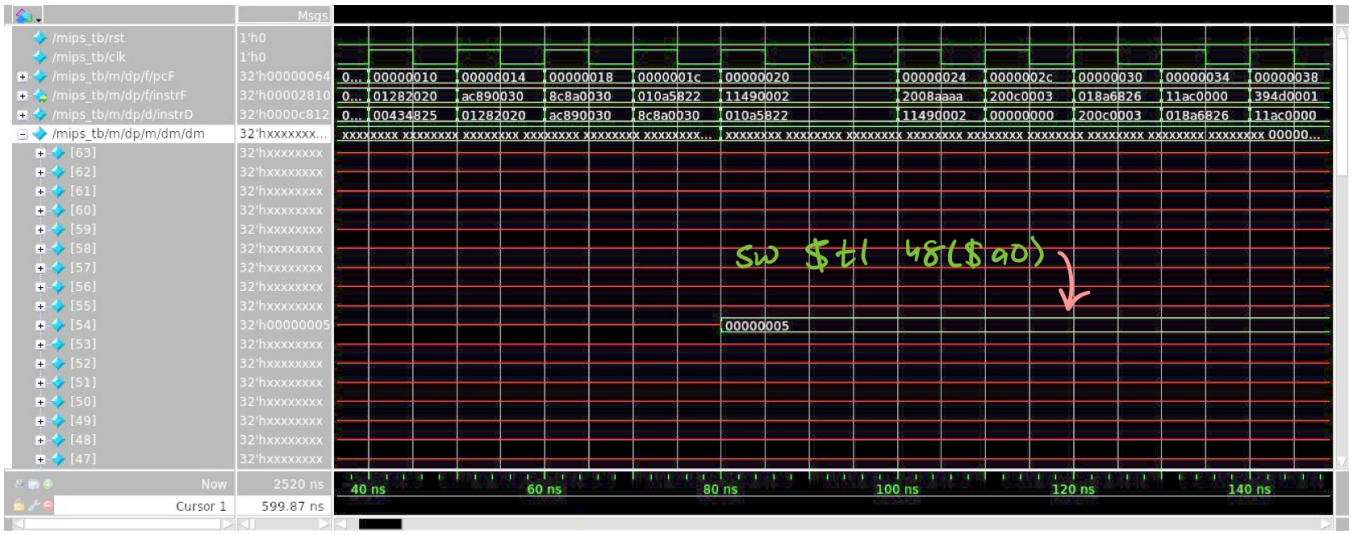
The following program was designed to thoroughly test the functionality of all the instructions. Since the vector multiplication tests signed multiplication, our program only checks for unsigned multiplication.

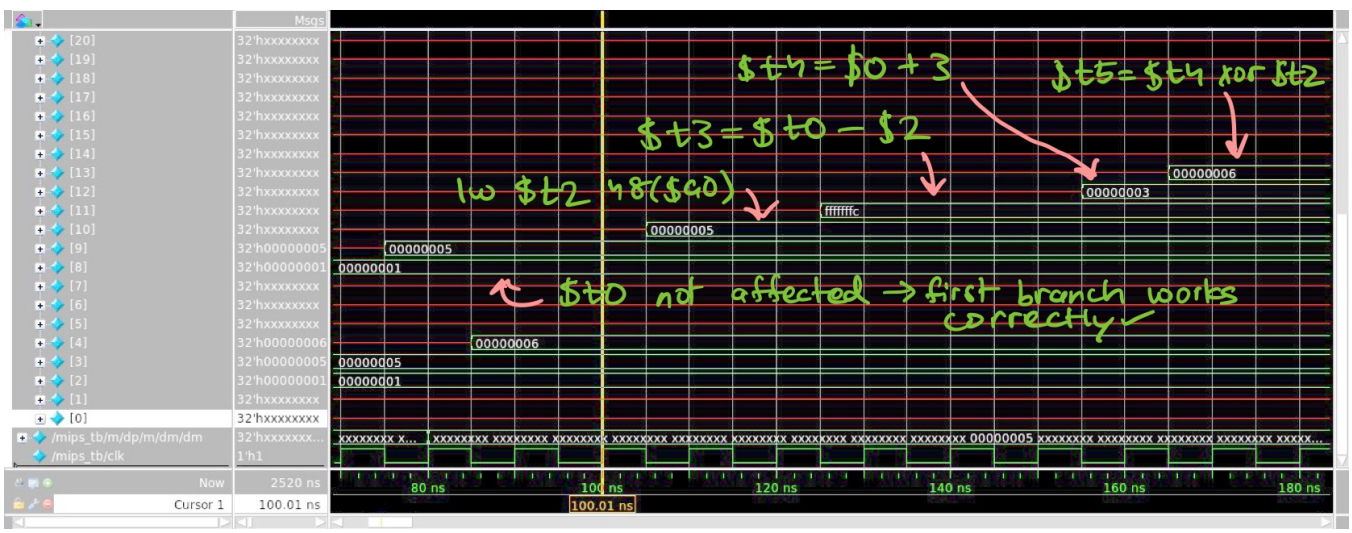
```
addi $v0,$0,1  #v0 = 1
ori $v1,$0,5   #v1 = 5
and $t0,$v0,$v1 #t0 = 1
or $t1,$v0,$v1 #t1 = 5
add $a0,$t1,$t0 #a0 = 6
sw $t1, 48($a0)
lw $t2, 48($a0) #t2 = 5
sub $t3, $t0, $t2 #t3 = -4
beq $t2,$t1,2 #branch
addi $t0,$0,0xAAAA  #Should not perform
addi $t0,$0,0xFFFF #Should not perform
addi $t4,$0,3  #t4 = 3
```

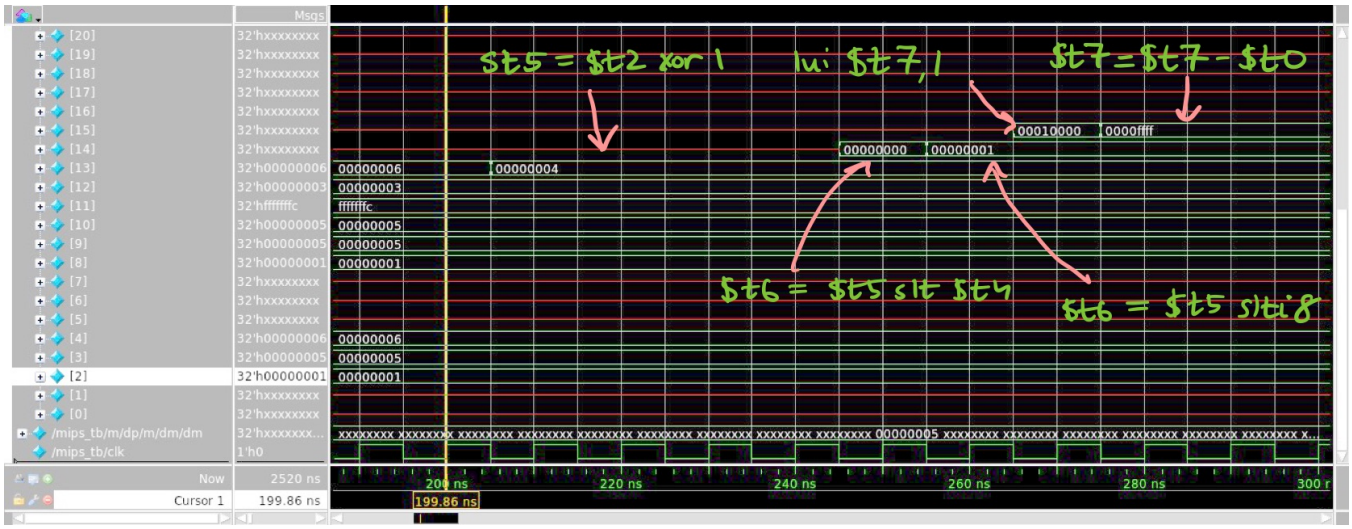
```
xor $t5,$t4,$t2 #t5 = 6
beq $t5,$t4,0 #Should not perform
xori $t5,$t2,1 #t5 = 4
bne $t5,$t4,0 #branch
addi $t0,$0,0xAAAA #Should not perform
slt $t6,$t5,$t4 #t6 = 0
slti $t6,$t5,8 #t6 = 1
lui $t7,1 #t7 = 10000 HEX
sub $t7,$t7,$t0 #t7 = FFFF HEX
andi $t8,$t7,4 #t8 = 4
lui $t9, 8 #t9 = 80000 HEX
multu $t7,$t9 #mult = 7FFF80000 HEX
mflo $t9 #t9 = FFF80000 HEX
mfhi $a1 #a1 = 7
addi $a1,$a1,1 #a1=8
sw $t1, 52($a0)
xnor $t5,$t4,$t2 #t5 = fffffff9
JAL 0x100025
```

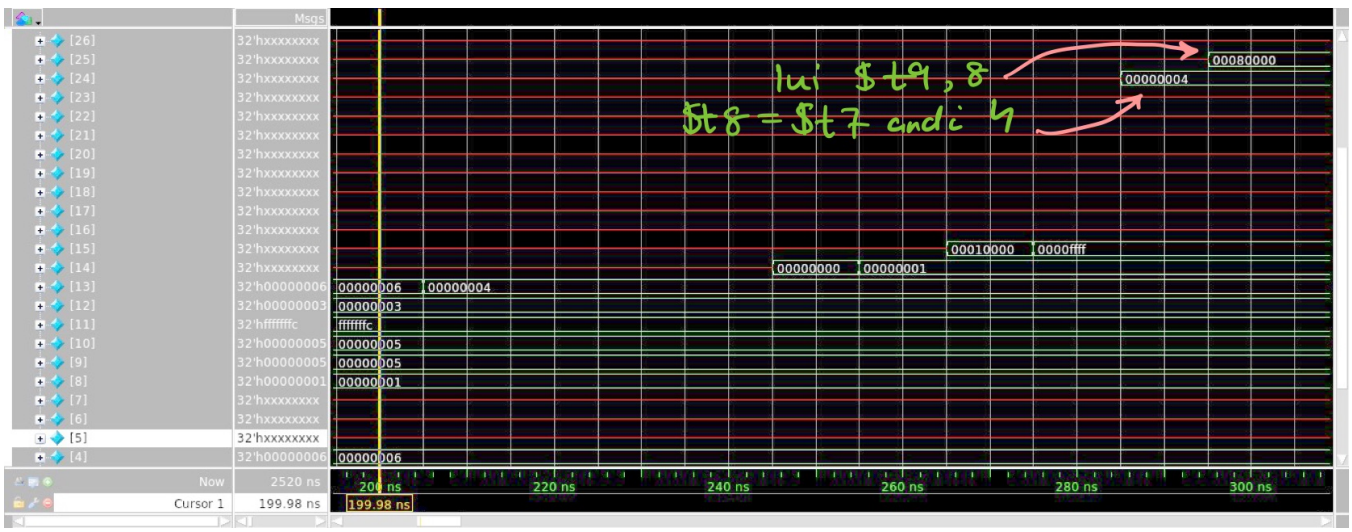
As shown in the program, there are many cases of read after write hazards, load-use hazards, and branches. This, combined with the multiplication, convinces us that our design is robust if it can pass this test. Using ModelSim, we'll simulate our design and check the data saved in the data memory and register file to verify correctness. The simulation waveform of the register file is shown in the images below, annotated with the corresponding instructions.

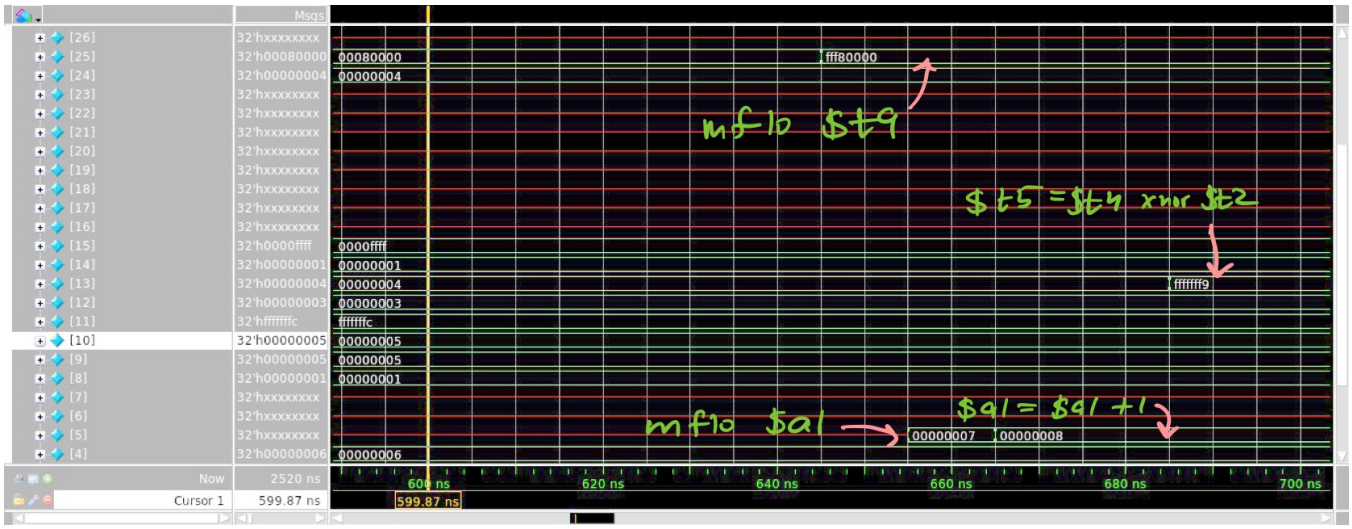


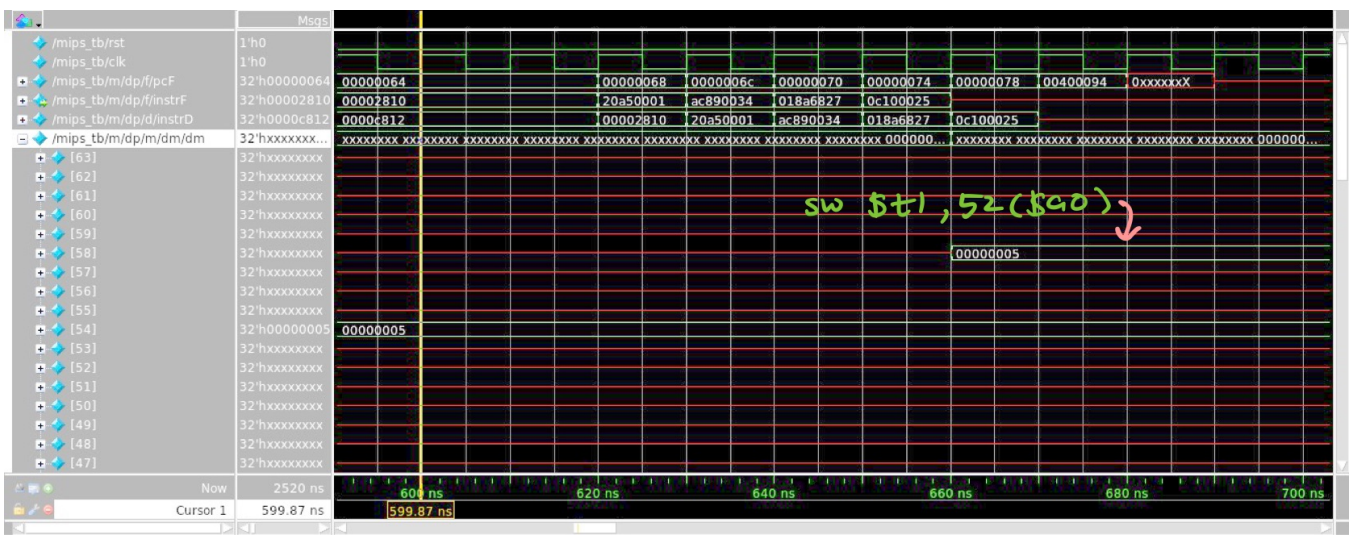

sw $t1 48($a0)
00000005


$t4 = $0 + 3
$t5 = $t4 xor $t2
$t3 = $t0 − $2
lw $t2 48($a0)
00000006
00000003
fffffffc
00000005
00000005
00000001
$t0 not affected → first branch works correctly ✓
00000006
00000005
00000001


$t5 = $t2 xor 1
lui $t7,1
$t7=$t7-$t0
$t6 = $t5 slt $t4
$t6 = $t5 slti 8
00010000
0000ffff
00000000
00000001
00000006
00000004
00000003
fffffffc
00000005
00000005
00000001
00000006
00000005
00000001
Now
2520 ns
Cursor 1
199.86 ns
200 ns
220 ns
240 ns
260 ns
280 ns


lui $t9, 8
$t8 = $t7 and 4
00080000
00000004
00010000
0000ffff
00000000
00000001
00000006
00000004
00000003
fffffffc
00000005
00000005
00000001
00000006
Now
2520 ns
Cursor 1
199.98 ns
200 ns
220 ns
240 ns
260 ns
280 ns
300 ns





3.5 Dot Product

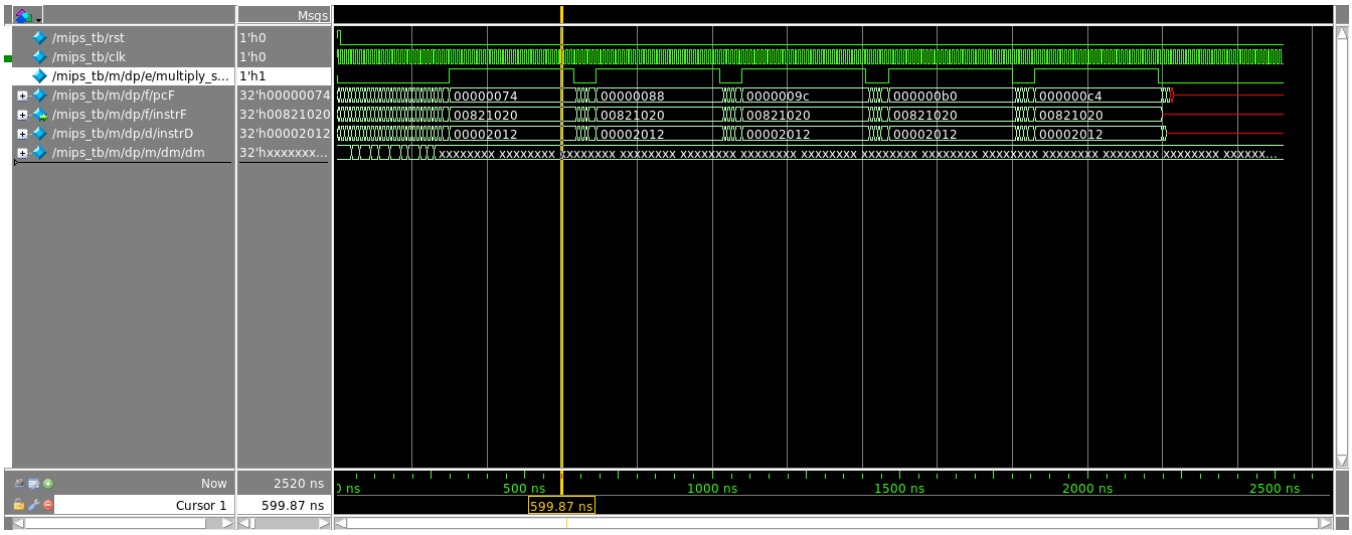
Now we'll test our processor by evaluating a dot product of the two vectors given in the handout. Below is the program to accomplish this

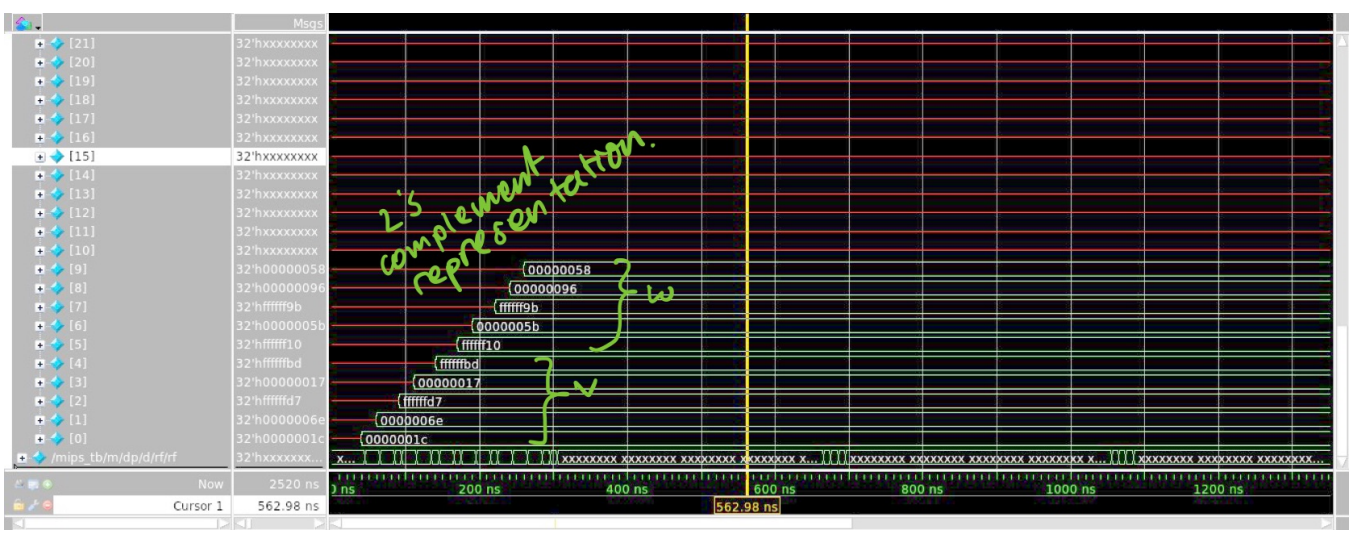
```
ADDI $v0 $zero 0x1C
SW   $v0 0x0($zero)
ADDI $v0 $zero 0x6E
SW   $v0 0x1($zero)
LUI  $v0 0xFFFF
```

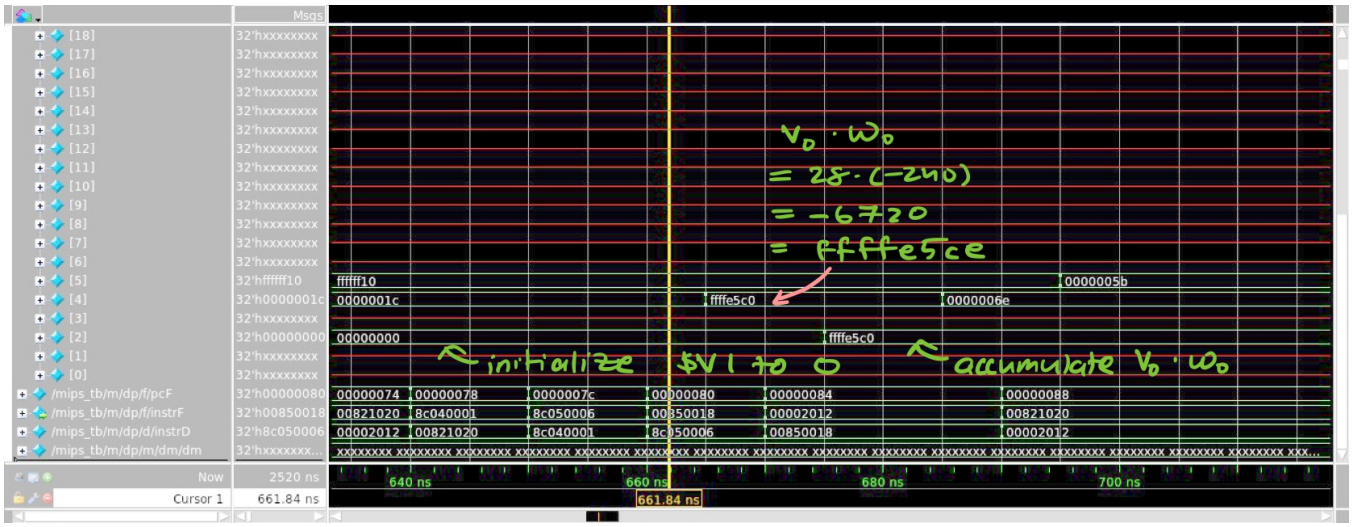
```
ADDI $v0 $zero 0xFFD7
SW   $v0 0x2($zero)
ADDI $v0 $zero 0x17
SW   $v0 0x3($zero)
LUI  $v0 0xFFFF
ADDI $v0 $zero 0xFFBD
SW   $v0 0x4($zero)
LUI  $v0 0xFFFF
ADDI $v0 $zero 0xFF10
SW   $v0 0x5($zero)
ADDI $v0 $zero 0x5B
SW   $v0 0x6($zero)
LUI  $v0 0xFFFF
ADDI $v0 $zero 0xFF9B
SW   $v0 0x7($zero)
ADDI $v0 $zero 0x96
SW   $v0 0x8($zero)
ADDI $v0 $zero 0x58
SW   $v0 0x9($zero)

ADDI $v0 $zero 0x0
LW $a0 0x0($zero)
LW $a1 0x5($zero)
MULT $a0 $a1
MFLO $a0
ADD $v0 $a0 $v0
LW $a0 0x1($zero)
LW $a1 0x6($zero)
MULT $a0 $a1
MFLO $a0
ADD $v0 $a0 $v0
LW $a0 0x2($zero)
LW $a1 0x7($zero)
MULT $a0 $a1
MFLO $a0
ADD $v0 $a0 $v0
LW $a0 0x3($zero)
LW $a1 0x8($zero)
MULT $a0 $a1
MFLO $a0
ADD $v0 $a0 $v0
LW $a0 0x4($zero)
LW $a1 0x9($zero)
MULT $a0 $a1
MFLO $a0
ADD $v0 $a0 $v0
```

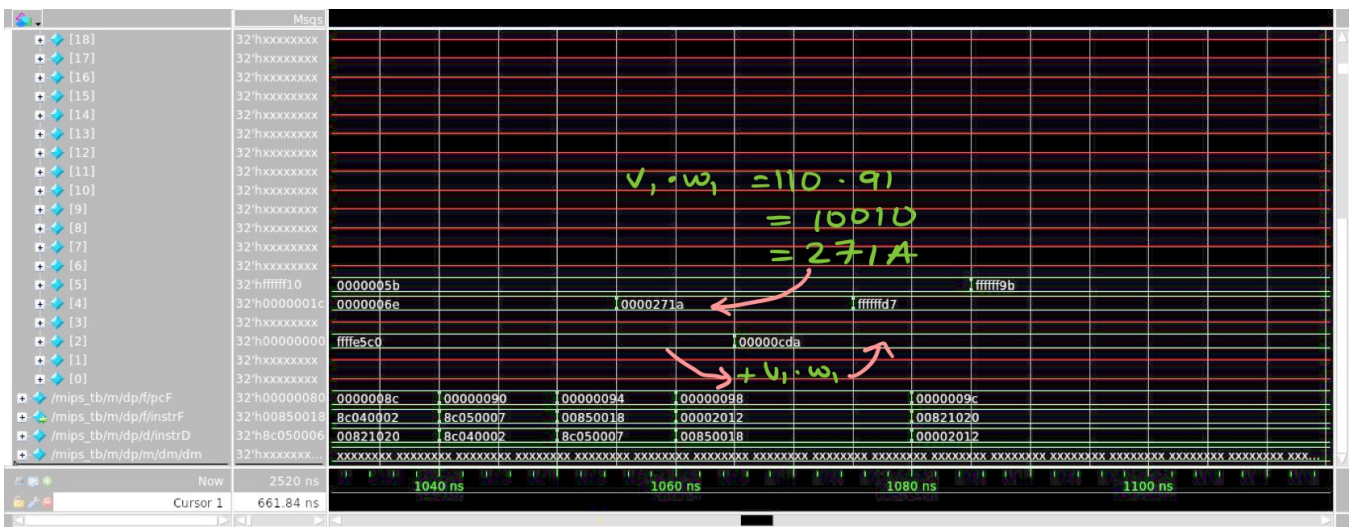
This program first loads each of the values in the vectors in data memory. Then it pulls each pair of values that need to be multiplied, and keeps a cumulative sum in register `$v0`. Again, we will examine the waveform to evaluate the correctness of the dot product and take a closer look at each step.

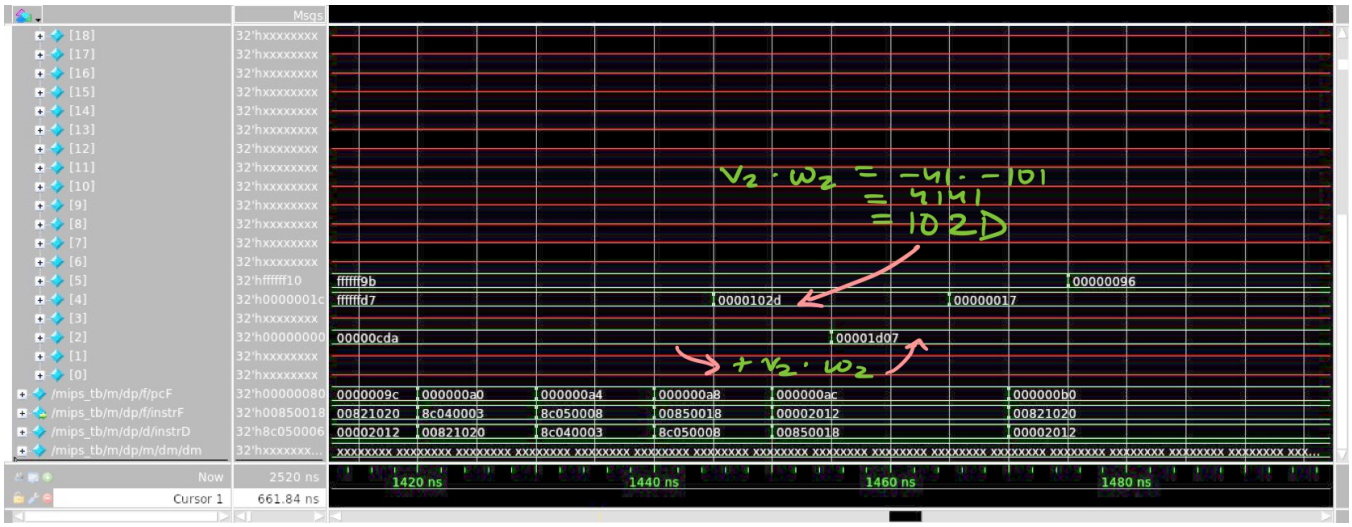
By zooming out of the waveform window, we can see that the `multiply_status` signal is high five times. This tells us that the processor is in fact multiplying five times, which is a sanity check before proceeding. The expected result is 0d4985=0x1369.



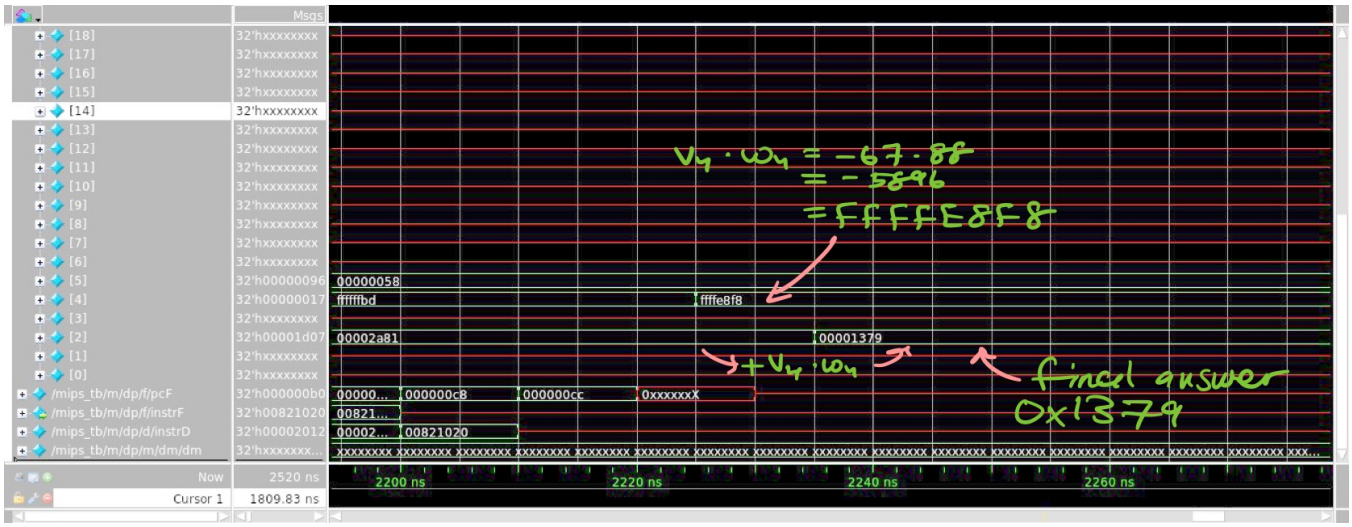



$v_0 \cdot w_0$
$= 28 \cdot (-240)$
$= -6720$
$=$ ffffe5ce
initialize $v1 to 0
accumulate $v_0 \cdot w_0$


$v_1 \cdot w_1 = 110 \cdot 91$
$= 10010$
$=$ 271A
$+ v_1 \cdot w_1$


Msgs
[18] 32'hxxxxxxxx
[17] 32'hxxxxxxxx
[16] 32'hxxxxxxxx
[15] 32'hxxxxxxxx
[14] 32'hxxxxxxxx
[13] 32'hxxxxxxxx
[12] 32'hxxxxxxxx
[11] 32'hxxxxxxxx
[10] 32'hxxxxxxxx
[9] 32'hxxxxxxxx
[8] 32'hxxxxxxxx
[7] 32'hxxxxxxxx
[6] 32'hxxxxxxxx
[5] 32'hffffff10 ffffff9b 00000096
[4] 32'h0000001c ffffffd7 0000102d 00000017
[3] 32'hxxxxxxxx
[2] 32'h00000000 00000cda 00001d07
[1] 32'hxxxxxxxx
[0] 32'hxxxxxxxx
/mips_tb/m/dp/f/pcF 32'h00000080 0000009c 000000a0 000000a4 000000a8 000000ac 000000b0
/mips_tb/m/dp/f/instrF 32'h00850018 00821020 8c040003 8c050008 00850018 00002012 00821020
/mips_tb/m/dp/d/instrD 32'h8c050006 00002012 00821020 8c040003 8c050008 00850018 00002012
/mips_tb/m/dp/m/dm/dm 32'hxxxxxxx...
Now 2520 ns
Cursor 1 661.84 ns
1420 ns
1440 ns
1460 ns
1480 ns
V2 · W2 = -41 · -101
= 4141
= 102D
+ V2 · W2

Msgs
[18] 32'hxxxxxxxx
[17] 32'hxxxxxxxx
[16] 32'hxxxxxxxx
[15] 32'hxxxxxxxx
[14] 32'hxxxxxxxx
[13] 32'hxxxxxxxx
[12] 32'hxxxxxxxx
[11] 32'hxxxxxxxx
[10] 32'hxxxxxxxx
[9] 32'hxxxxxxxx
[8] 32'hxxxxxxxx
[7] 32'hxxxxxxxx
[6] 32'hxxxxxxxx
[5] 32'h00000096 00000096 00000058
[4] 32'h00000017 00000017 00000d7a ffffffbd
[3] 32'hxxxxxxxx
[2] 32'h00001d07 00001d07 00002a81
[1] 32'hxxxxxxxx
[0] 32'hxxxxxxxx
/mips_tb/m/dp/f/pcF 32'h000000b0 000000... 000000b4 000000b8 000000bc 000000c0 000000c4
/mips_tb/m/dp/f/instrF 32'h00821020 00821... 8c040004 8c050009 00850018 00002012 00821020
/mips_tb/m/dp/d/instrD 32'h00002012 00002... 00821020 8c040004 8c050009 00850018 00002012
/mips_tb/m/dp/m/dm/dm 32'hxxxxxxx...
Now 2520 ns
Cursor 1 1809.83 ns
1809.83 ns
1820 ns
1840 ns
1860 ns
1880 ns
V3 · W3
= 23 · 150
= 3450
= D7A
+ V3 · W3



Using the integer variable in the for loop of the testbench, we can find the number of cycles taken for the multiplication by subtracting the value when it started multiplying from when it finished. This comes out to. The first instruction of the multiplication process is when the `$v0` register is cleared. This instruction's hex is 20020000 and is fetched on cycle 22. The final answer is available to read at register `$v0` on cycle 221. This means that the multiplication takes 199 cycles in total. The multiplication process requires 26 instructions so the total CPI is $\frac{199 \text{ cycles}}{26 \text{ instructions}} \approx 7.65 \frac{\text{cycles}}{\text{instruction}}$.

4 Conclusion

This project was extremely challenging, but an extremely valuable learning experience for a large Verilog project. The largest project done for a class before this was the multi-cycle MIPS processor done in ECE 154A, which was comparatively simple since the pipeline has five independently operating stages. Debugging the timing issues were the most challenging because it was difficult to track down the source of the issue with this many moving parts. This is where I learned how to use wires and registers together to correctly time the movement of data. In the past, I'd avoid mixing the two constructs because they often led to frustrating results because of the mix of asynchronous and synchronous components. However, once I learned how to use them correctly, it was easy to update registers correctly at each clock rising edge, or not do so when there's a stall/flush.

Although we thought less-rigorous unit testing would come at a price of a frustrating debugging experience, it's clear that it was the right decision to prioritize getting the modules together into a pipeline before polishing it for the edge cases. Because we had a pipeline to work with, even if it wasn't complete, it allowed us to more quickly find bugs in each stage in simpler cases before

moving to complicated cases. It was less time consuming to make changes to the code since we'd be able to change the individual modules and see how that change reflects in the pipeline.

We also learned the value of planning before committing to code. Putting together a schematic made it easy to split the work while staying on the same page. Despite my partner not being able to help for much past the independent components, our planning in advance made it easier to pick up his work without having to read every line of code. In total, the project took 25-35 hours.