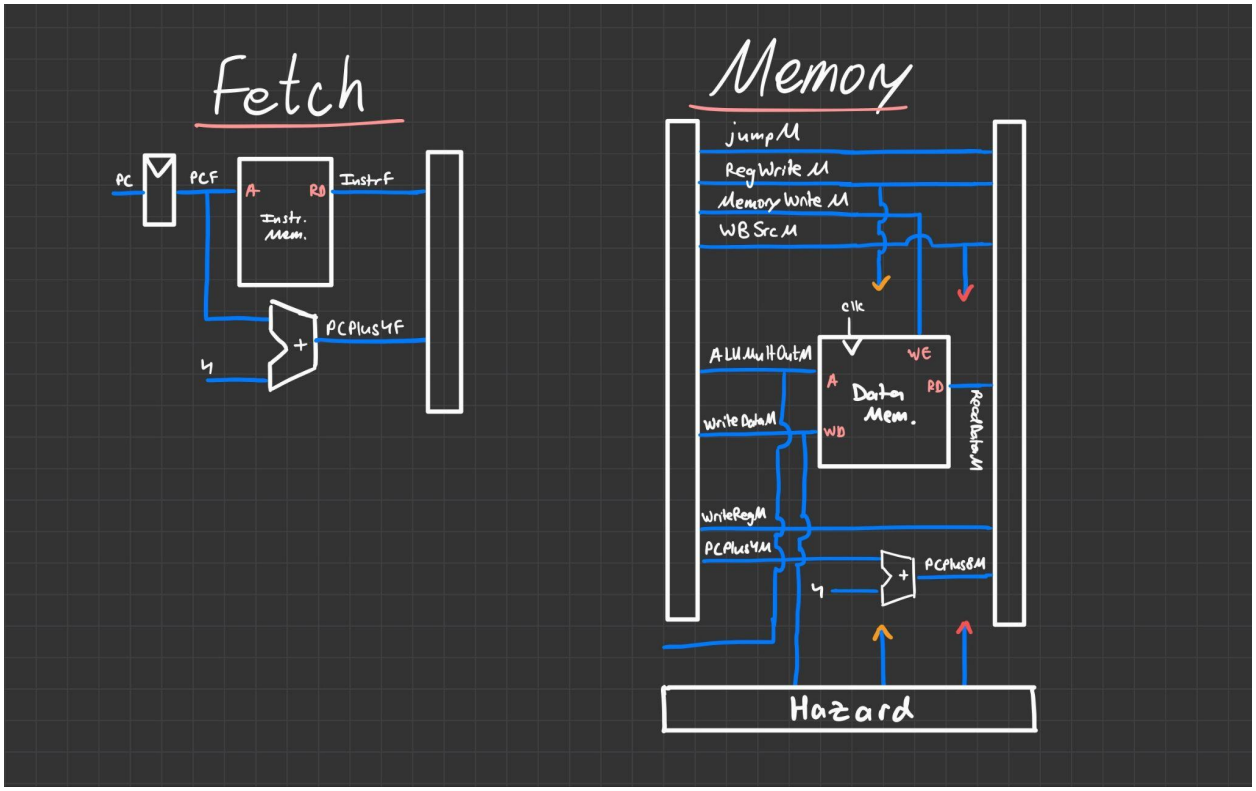
Project 2: Memory Hierarchy
Brain Lee, Arjun Vinod
February 9, 2022

Introduction

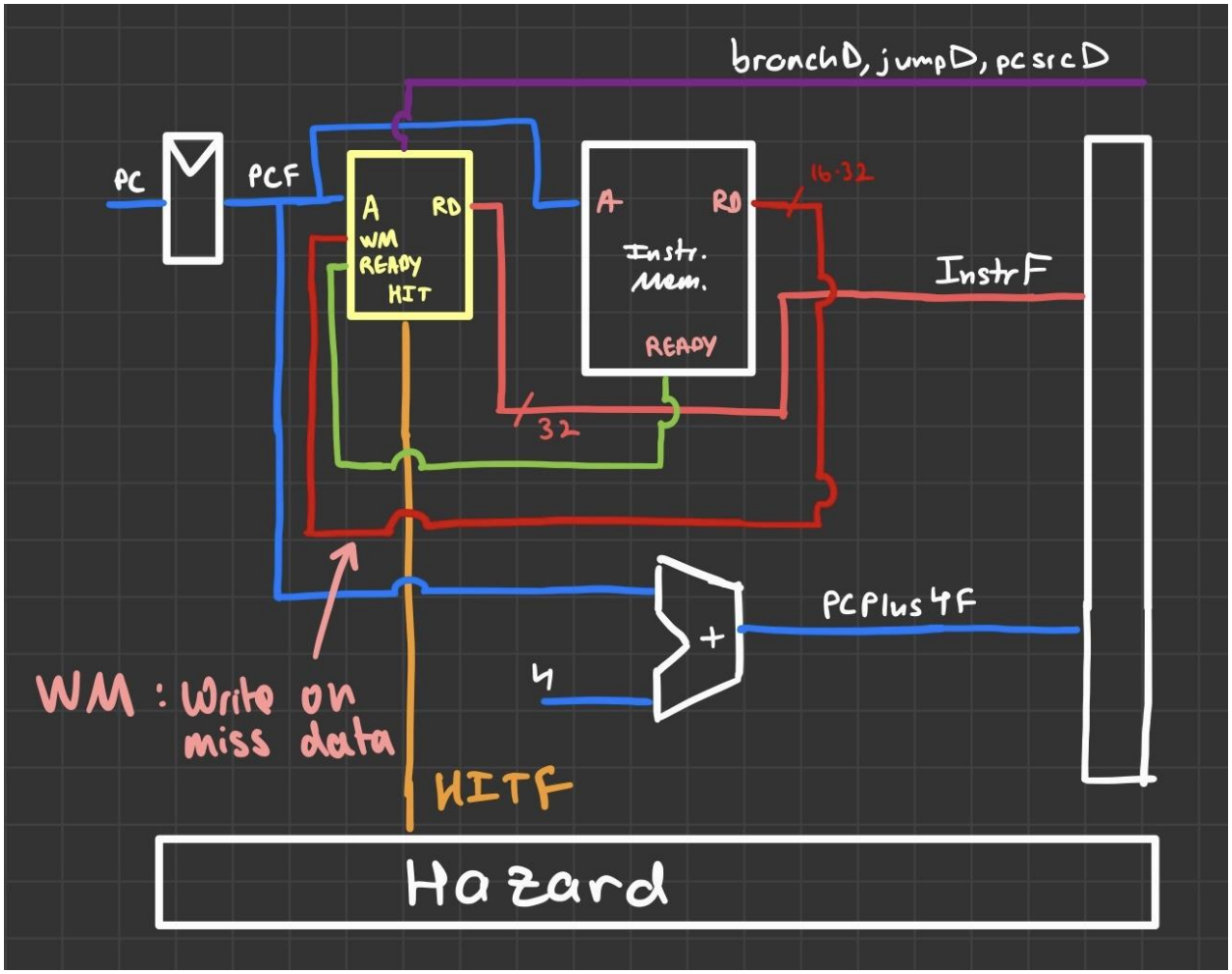
In this project, we revise our pipelined microprocessor from the previous lab to accommodate for slower memories by building a cache system new memory module that uses delays. This report covers the design, implementation, and testing of our new pipelined microprocessor.

Design Specification

For reference, below is the design of the original microprocessor without the caches:



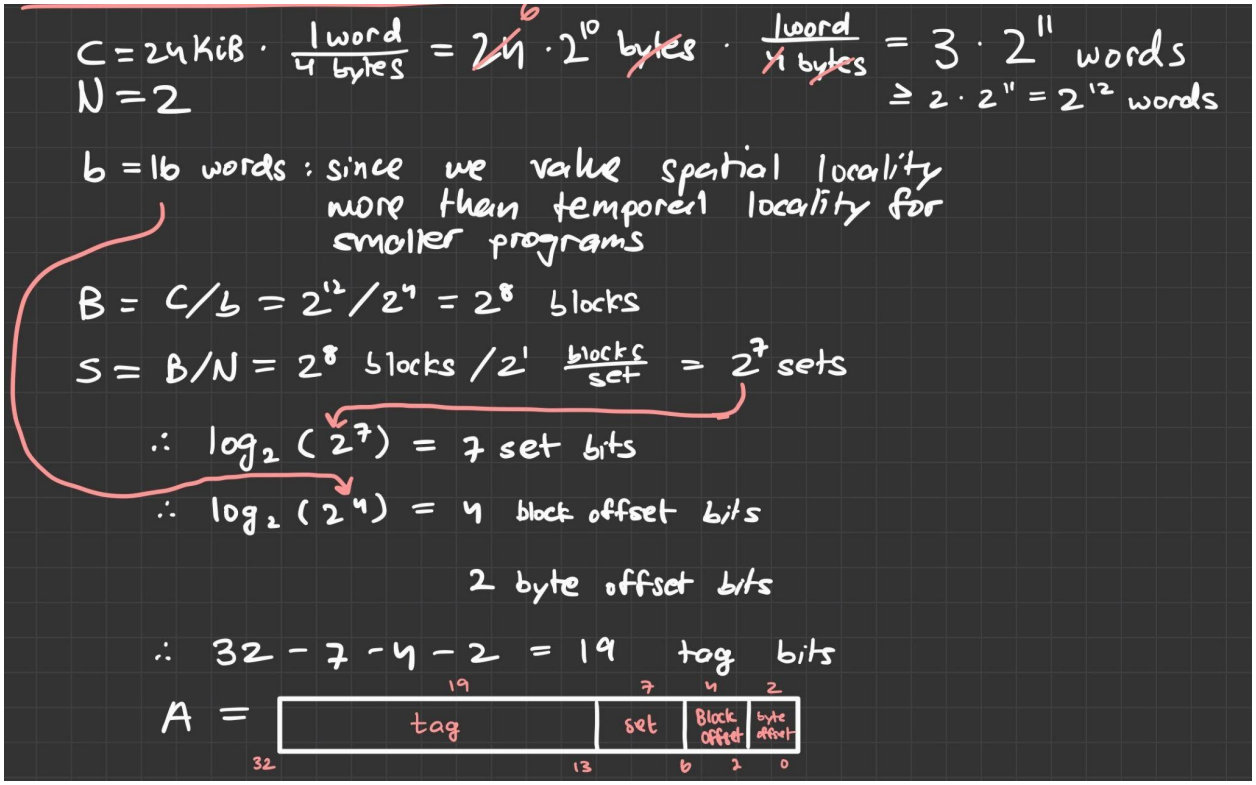
Instruction Cache (~24 KiB)



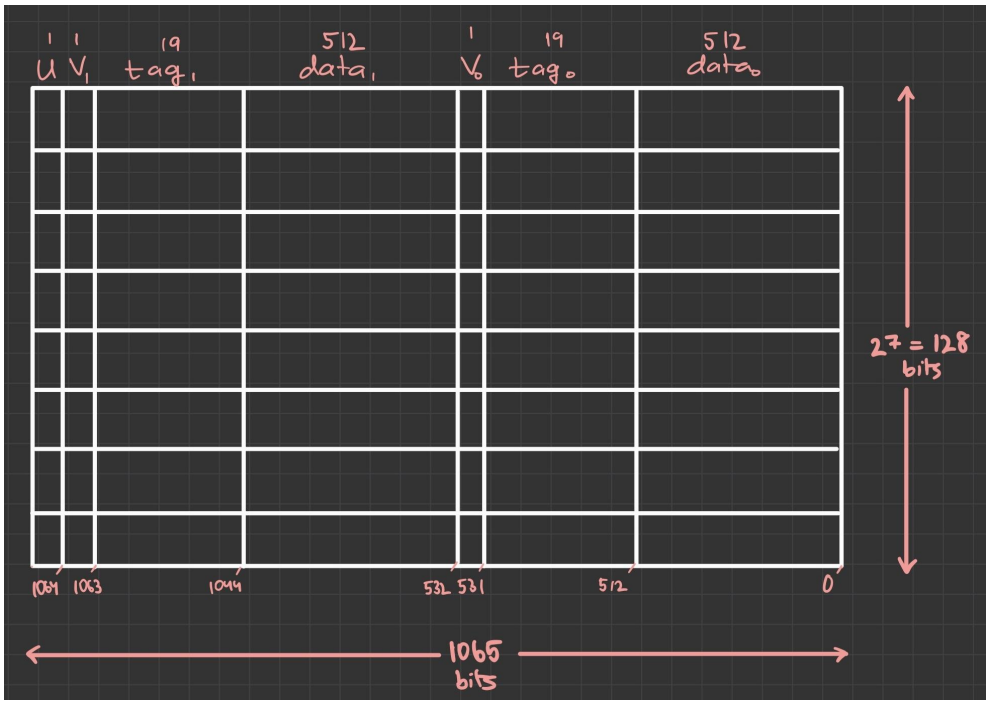
The instruction cache module now precedes the instruction memory module in the fetch stage. The inputs of this module include: the program counter, data read out from instruction memory, the ready signal from instruction memory, and branch/jump signals from the decode stage. The branch and jump signals from the decode stage do not play a role in the functionality of the cache, but are kept for debugging purposes in the future with more complicated branching behaviors. The outputs of the module include a hit signal sent to the hazard module and the data read out from the cache.

We made some assumptions and educated guesses in the process of determining the parameters of the cache. We allocated the majority of the cache budget to instruction cache because we expect more control flow (branches and jumps) in our small MIPS programs compared to data memory accesses. We decided to choose an N=2 set associative cache to strike a balance between implementation complexity and avoiding a “ping-pong” effect of overwriting necessary cached memory in a direct-mapped cache. We also chose a block size of 16 because, again, for a small MIPS program that branches often (such as a matrix multiply), we will value storing contiguous blocks of instructions in cache if we repeatedly branch back to the same

position in the program. Using these parameters, we computed the number of blocks and sets to find the indices of the address used to map addresses to a position in cache.

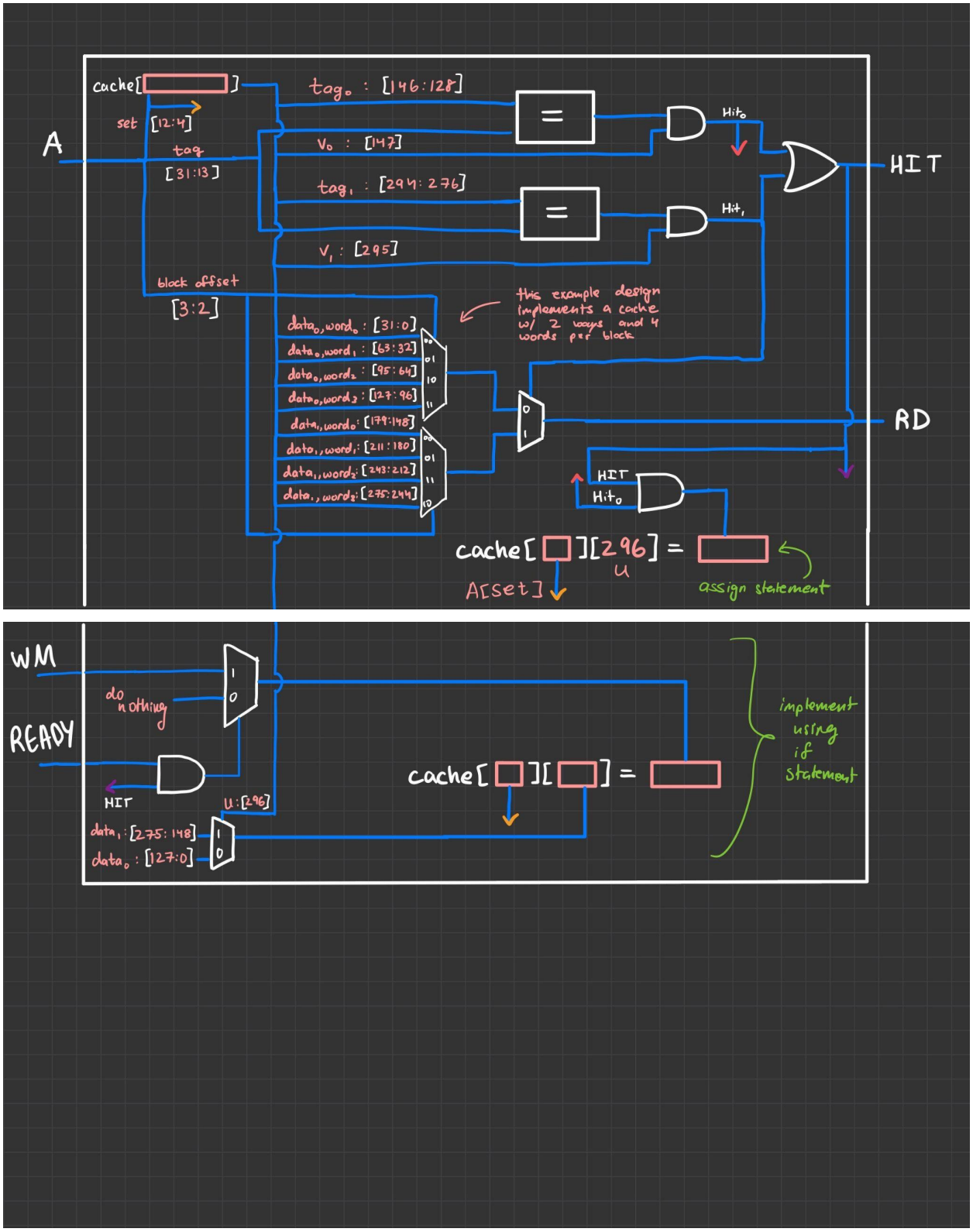


We then determined the structure of the cache register internal to the cache module.



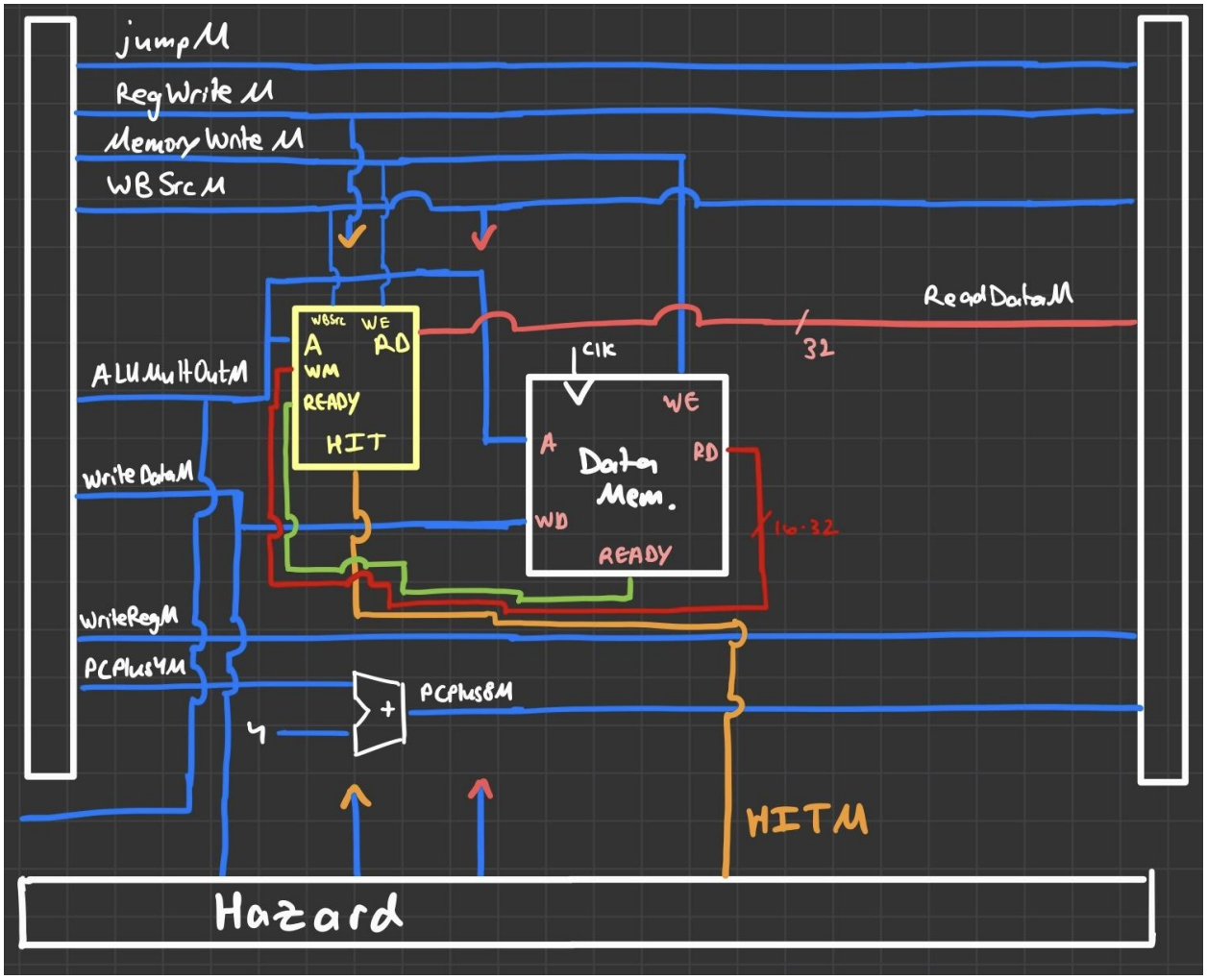
Finally, we sketched a schematic of the cache module before implementing it in verilog. Note that the schematic only shows 4 words per block, and this is because we changed the block size from 4 to 16 after experimenting with our initial design of 4 blocks. This general idea

represented in the schematic stays the same despite the values of the indices being invalid after our changes to the design:



Lastly, we modified the instruction memory to output 16 words of memory at a time to match the block size of 16 in our cache.

Data Cache (~8 KiB)



$C = 8\,\text{KiB} \cdot \frac{1\,\text{word}}{4\,\text{bytes}} = 8 \cdot 2^{10}\,\text{bytes} \cdot \frac{1\,\text{word}}{4\,\text{bytes}} = 2^{11}\,\text{words}$

$N = 2$

$b = 16$ words : since we value spatial locality more than temporal locality for smaller programs

$B = C/b = 2^{11}/2^4 = 2^7$ blocks

$S = B/N = 2^7\,\text{blocks} / 2^1\,\frac{\text{blocks}}{\text{set}} = 2^6$ sets

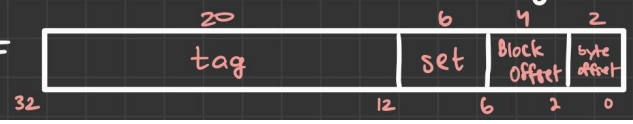
$\therefore \log_2(2^6) = 6$ set bits

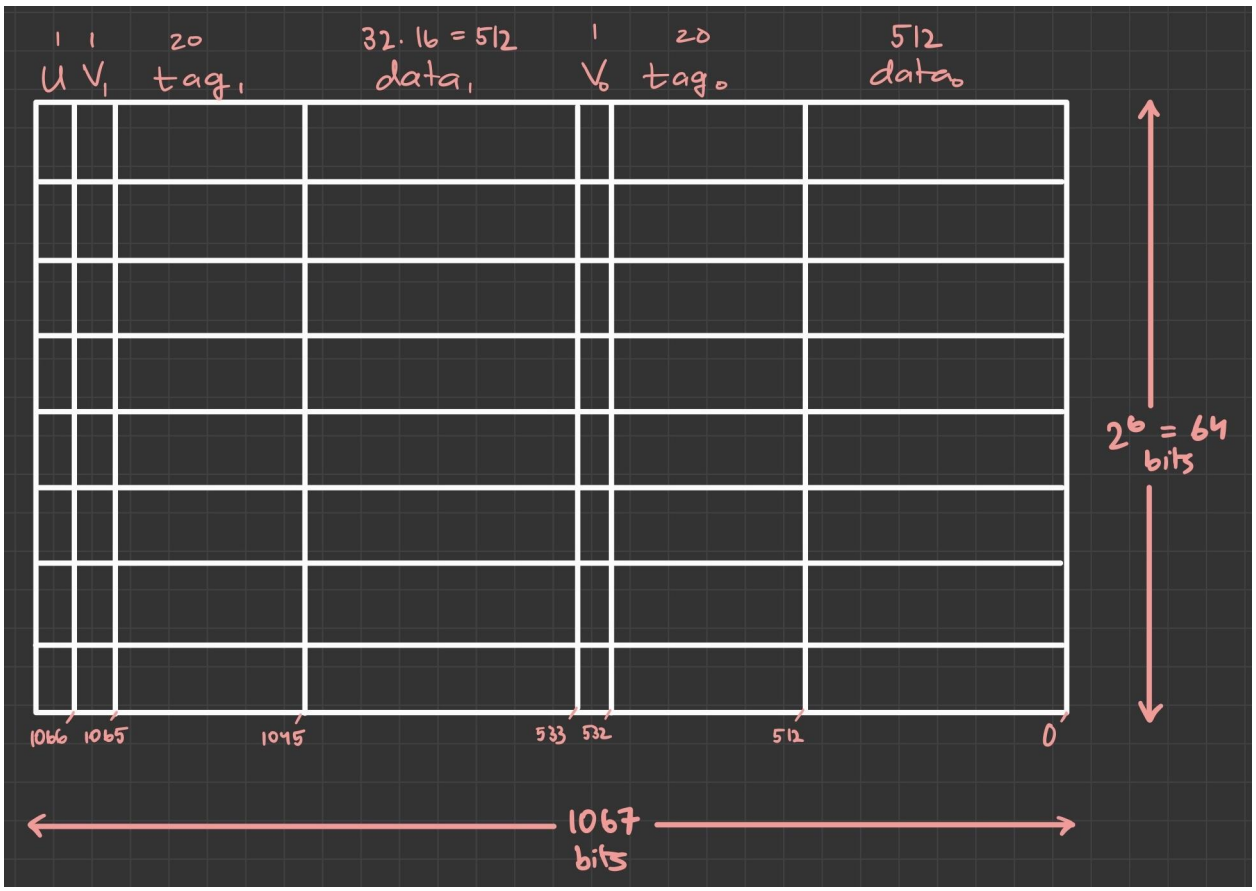
$\therefore \log_2(2^4) = 4$ block offset bits

2 byte offset bits

$\therefore 32 - 6 - 4 - 2 = 20$ tag bits

A =





Using the same process as with the instruction memory, we implemented the instruction memory. The only difference is the write behavior of the data cache. We also input a write enable and memtoreg signal to check if there i a write to data memory. If there is, we also update the value in cache if that address is a hit. Otherwise, we do nothing since the data is not in cache until it is fetched from memory.

Other Changes to the Pipeline

Although the cache modules were novel to the pipeline, a large portion of our time was spent debugging the pipeline and making changes to other modules, like the hazard unit, to behave correctly in light of the 20-cycle penalty of memory reads. Most notably, we split the behavior of a branch in the hazard module into two cases: a branch during an instruction read hit and miss. These two cases behave differently because if the next instruction should not be executed due to a branch, we should not unnecessarily make a 20-cycle memory read. So even if the instruction is a miss and if the decode stage determines that there is a branch, the hazard module should not stall the pipeline until the cache signals the data is ready from memory and is now a hit.

Testing

Data Cache/Memory

As the data cache and the instruction cache were the only new modules for this lab assignment, we did not feel the need to unit-test the modules created in the previous lab and only created testbenches for the cache systems. The data cache and instruction cache are functionally the same with only a few differences in the parameters of their internal caches. We found it adequate to write a simple test program that tests that data is fetched from memory to cache on a miss and that it is read from cache if it is a miss. The cache uses a write-through policy so we also had to test the write behavior and ensure that the LRU policy is implemented correctly by writing data to the same set with a different tag.

Project 1 Grader

Our previous grader program from project 1 intensively tested all the possible hazards and expected functionality of the pipeline without a cache. As a sanity check, we ran this program with our new pipeline and compared results to ensure that none of the existing functionality had been compromised. By simulating and comparing the waveforms to our previous report, we verified that the behavior of the pipeline is still correct.

Matrix Multiplication

The matrix multiply program multiplies a 3x3 matrix (M) with a 3x1 vector (v) to produce a resultant 3x1 vector (Mv). The program can be broken down into the following stages:

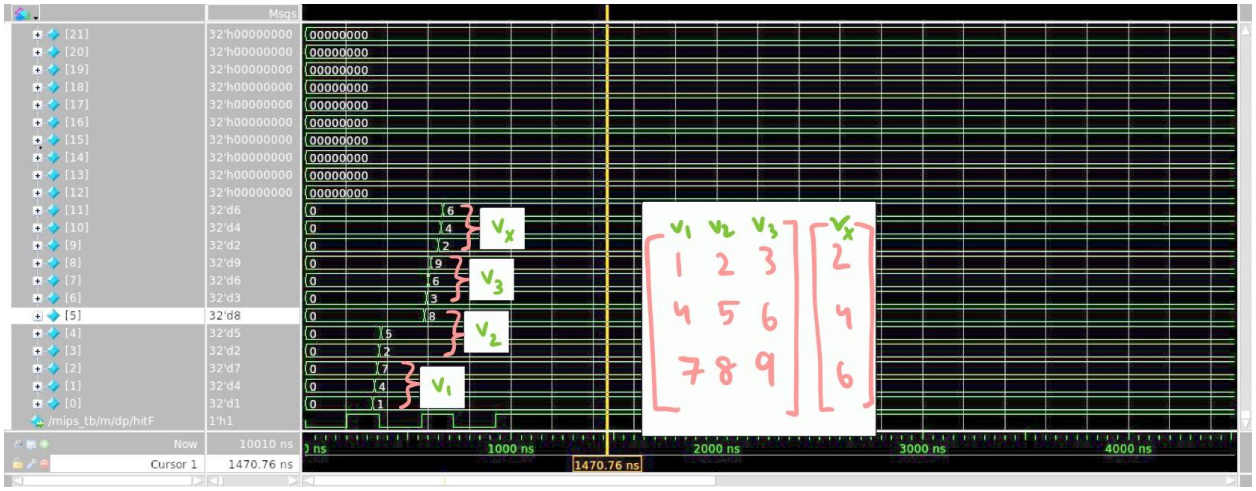
1. Load 3x1 matrix (M) values into data memory in column-major order
2. Load 3x1 vector (v) values into data memory
3. Clear $a0, $a1, and $a2 registers to store the result (Mv)
4. Define $t9 as the address offset to the first element of the matrix (M)
5. Define $t7 as the address offset to the first element of the vector (v)
6. Define $t8 as the address offset to the last element of the matrix (M) + 1
7. Load the data at a 0x04, 0x08, and 0x0C offset to $t9 from data memory into three registers. This is one column of the matrix $[M_{1,i}\ M_{2,i}\ M_{3,i}]$.
8. Load the data at a 0x00 offset to $t7 into a register. This is one value of the vector (v_i).
9. Multiply each value of the matrix column by the value of the vector
10. Accumulate each resulting product in $a0, $a1, and $a2
11. Add 0x0C to $t9
12. Add 0x04 to $t7
13. If $t9 != $t8, branch to step (7)

The assembly instructions for the matrix multiply are shown below.

```
# Load matrix to data memory (column-major)
# [1 2 3]
# [4 5 6]
# [7 8 9]
addi $t0 $0 0x1
addi $t1 $0 0x4
addi $t2 $0 0x7
addi $t3 $0 0x2
addi $t4 $0 0x5
addi $t5 $0 0x8
addi $t6 $0 0x3
addi $t7 $0 0x6
addi $t8 $0 0x9
SW   $t0 0x00($0)
SW   $t1 0x04($0)
SW   $t2 0x08($0)
SW   $t3 0x0c($0)
SW   $t4 0x10($0)
SW   $t5 0x14($0)
SW   $t6 0x18($0)
SW   $t7 0x1c($0)
```

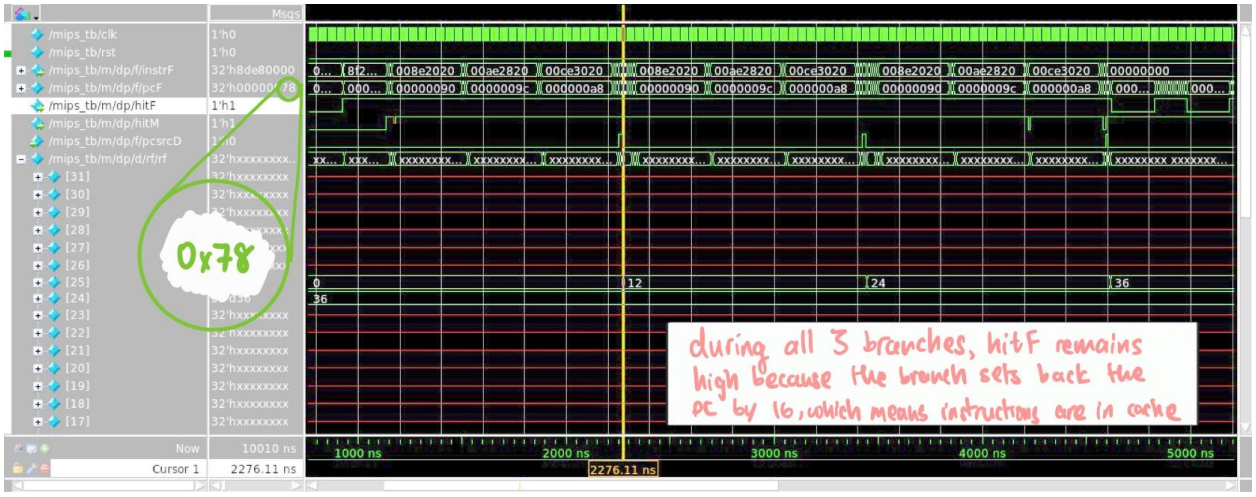
```
SW   $t8 0x20($0)

# Load vector to data memory
# [2 4 6]
addi $t0 $0 0x2
addi $t1 $0 0x4
addi $t2 $0 0x6
SW   $t0 0x24($0)
SW   $t1 0x28($0)
SW   $t2 0x2c($0)

# Store results of each cell in [a0 a1 a2]
addi $a0 $0 0x0
addi $a1 $0 0x0
addi $a2 $0 0x0
addi $t9 $0 0x0  # Address offset for matrix
addi $t8 $0 0x24 # Used to branch/loop later
addi $t7 $0 0x24 # Address offset for vector

# Load vector and matrix elements from memory
LW   $t0 0x00($t7) # vector element v(i)
LW   $t1 0x00($t9) # matrix element m(1,i)
LW   $t2 0x04($t9) # matrix element m(2,i)
LW   $t3 0x08($t9) # matrix element m(3,i)
# Multiply and accumulate the partial sums
MULT $t0 $t1
MFLO $t6
ADD  $a0 $a0 $t6
MULT $t0 $t2
MFLO $t6
ADD  $a1 $a1 $t6
MULT $t0 $t3
MFLO $t6
ADD  $a2 $a2 $t6
# Add to address offset of matrix and vector
# to use next column and cell respectively
ADDI $t7 $t7 0x04
ADDI $t9 $t9 0x0C
# Loop up to loading data from memory with
# new offset 3 times to complete computation
BNE  $t9 $t8 FFF0 # -16 2's complement
```

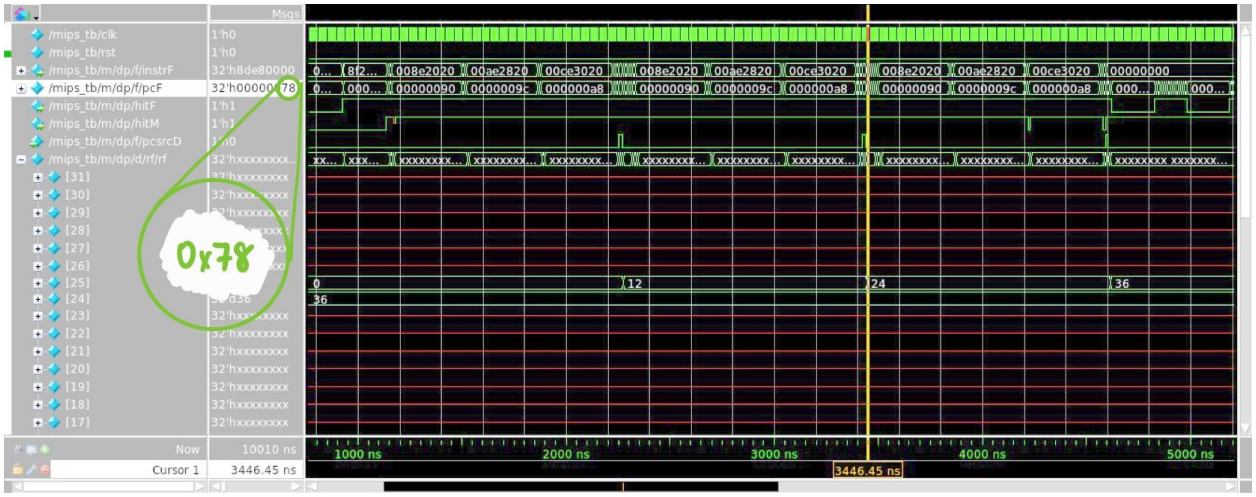
After running this program, we captured important information from the waveforms to verify the correctness of the algorithm and our design. As mentioned before, the values of the matrix and vector are first stored in data memory. We see that there is a 20 cycle gap after loading the value 5, and this is because we have reached the end of the 16 cached instructions and must fetch the next 16 before we can progress with the program.

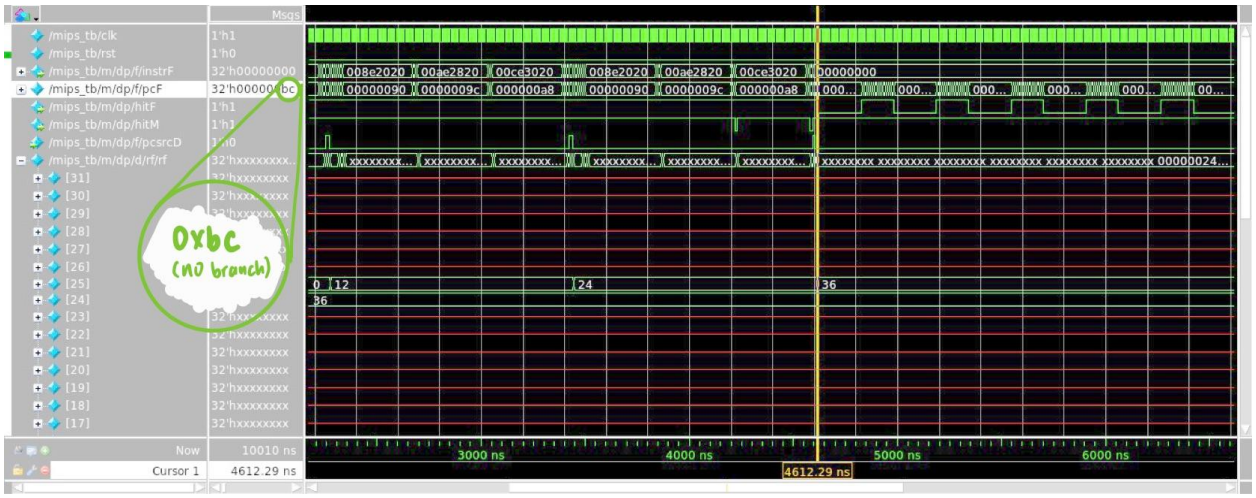


Now we will look at the branching behavior of our program, which ensures the correctness of our hazard handling, since each branch will happen at the 16th instruction of the fetched block of instructions. This is a strong test for our design because if our design did not determine the branch decision in time, we would waste 20 cycles fetching instructions we do not need after the last iteration of the loop. We can see where the program encounters the branch by observing the pcsrcD signal in the waveform. For both instances **pcsrcD** is high, **hitF** is also high. This means that for all three iterations of the do while loop, the instructions were cached and there were no delays due to instruction cache misses. And for the final branch decision, hitF goes low and there is a 20 cycle delay to fetch the next instruction, which is the end of the program.

We can also be certain the forwarding logic of our design still works because in each iteration of the do while loop, there are load instructions followed by multiplication and add instructions that do not have gaps in between.

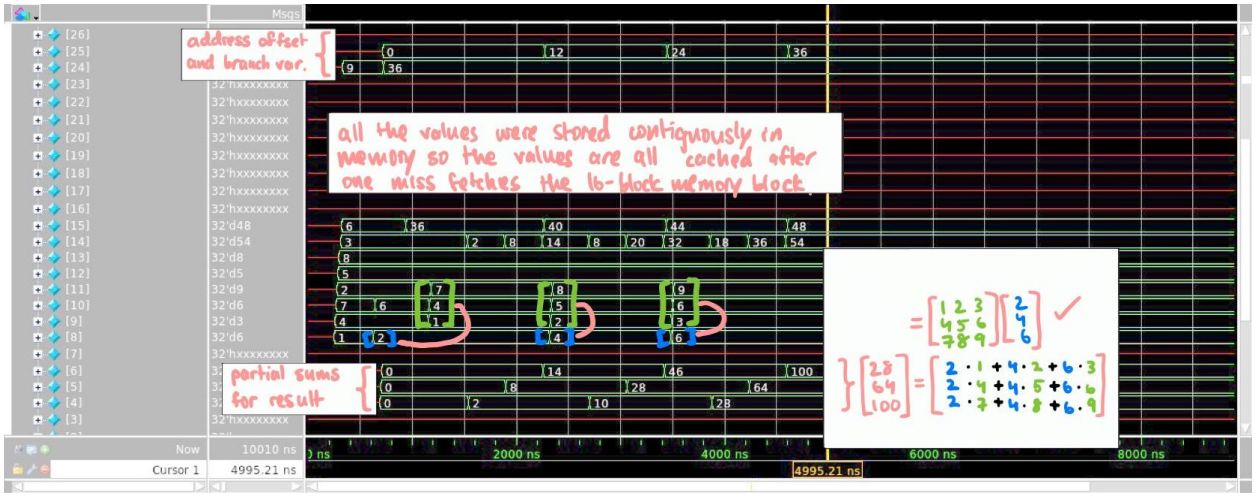






Lastly, we will look at the register file to follow the computations. We can see that when the values are first read from data memory and stored in registers, there is one 20-cycle delay before the first value is fetched. None of the following reads require reading from the data memory because the first 16-word block of memory was cached. This includes every value we need, so it is expected that there are no delays due to data memory reads.

At each iteration of the do while loop, we can see the values of the matrix column read from the cache and the first element of the vector is read from cache. These values are multiplied and accumulated in the $a0, $a1, and $a2 registers as expected to produce a result vector of [23 64 100]. This is the correct answer we verified with manual calculations. The 32-cycle delay in between each stage of reading new values from cache is due to the delay from the multiplier.



Conclusion

As a whole, project 2 was challenging and time-consuming to complete, since it took approximately 25-30 hours to complete. Naturally, because we did not have to build this pipelined microprocessor from scratch, the design phase was not nearly as involved as the previous project, this project had a decent amount of hazard and cycle delay issues to take into account when implementing the cache systems. One thing we noticed early on is the importance of large block sizes for both the data and instruction memory with the former being critical for performance in repeatedly branching programs like ours for matrix multiplication and the latter enabling values referenced sequentially and in parallel. Overall, there were fewer modules and testbenches to design compared to the first project, but handling hazards and making sure the matrix multiplication was executed correctly caused this project to take just as long.

A write-allocate policy fetches data from memory to the cache on a write miss. This would be beneficial because a variable that is being updated is more likely to be used soon, and if we can fetch it before it is needed, we can save time by having it ready in cache before it is needed. And to minimize the effect of the stalls on writes, a buffer would allow the pipeline to continue processing instructions while data is being written to memory and fetched to cache. Associativity is another factor that can affect the performance of the processor, but it is hard to gauge without empirical evidence such as via benchmarking. Since our caches have a relatively small number of sets (64 and 128 for data and instruction respectively), we may run into more collisions. More collisions would lead to more changes in the cache, which means we are not taking advantage of temporal locality as much as we can. This weakness makes it easy for our

design to encounter a program that renders the cache useless because each memory address maps to the same set. For example, if four memory addresses have the same mapping to a set with two ways. At every miss, one block of data will be evicted, so the processor may end up cycling between these four memory locations and having to stall for a memory read at every attempt to read from cache.

Being a new team, we learned about the power of planning and communication. Before writing any new piece of code for this project, we drew out diagrams and calculated the space needed for both the data and instruction caches. This allowed us to minimize the amount of testing and debugging that would have been done later. Joining a new code base is an incredibly daunting task because many decisions are made in the process of implementing a design in code that are not immediately clear to a fresh pair of eyes. This contributes significantly to the time taken to reach an equilibrium in productivity, but it provides an opportunity to get a fresh perspective on a problem you solved before. For Arjun, explaining how the code fits together was a valuable refresher before moving on to building on it for project 2.