Ray Chang
Tim Kim
ECE 154B

Project 2: Memory Hierarchy

Introduction

In the project, a 2-way set associative cache was designed and implemented for data memory, while a direct mapped cache was designed and implemented for instruction memory. The data cache acts as a smaller and faster memory which sits in front of the data memory in the pipeline for quicker accesses. Within the 32 KB area budget, around 11 KB were allocated for both the data and instruction caches, leading to a total of 22 KB, which meets the area budget requirement. Both data and instruction memories were meticulously designed and implemented in order to allow for the most efficient access time with the least amount of space used.

Block-Level Top Schematic

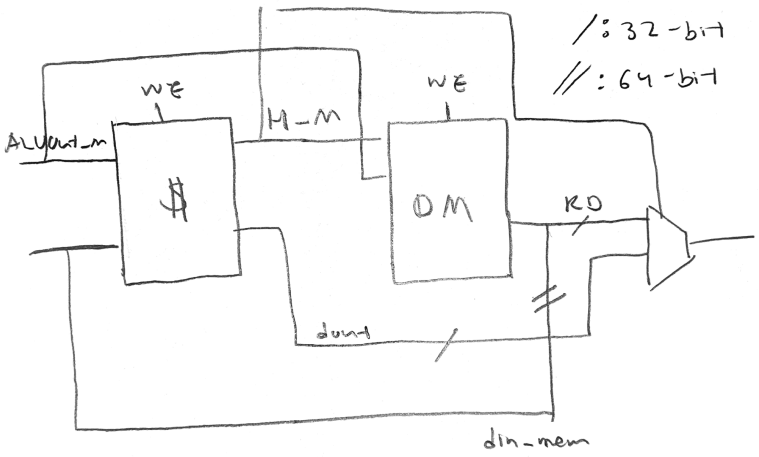


Figure 1. Schematic of Complete Data Memory Structure

The schematic above represents the complete data memory structure that is implemented in the MIPS pipeline processor. The data structure consists of a cache which lies in front of the data memory, and it is accessed before the data main memory is reached. If a hit happens, then the cache outputs the word data, and the 32 bit 2-input multiplexer outputs that data. On the other hand if a miss happens, then the main memory is accessed. After 20 cycles of runtime, a 32-bit word is outputted to the multiplexer. There is a feedback signal from the main memory to the cache which allows for the cache to hold the most recently used data. The feedback signal is a 64 bit signal as it contains the data of two words, and this implementation allows for spatial locality in the cache. It is important to note that the hit/miss signal controls the behavior of the result multiplexer; if the signal is 1, then the cache data is used, and if the signal is 0, then the data from main memory is used.

Ray Chang
Tim Kim
ECE 154B

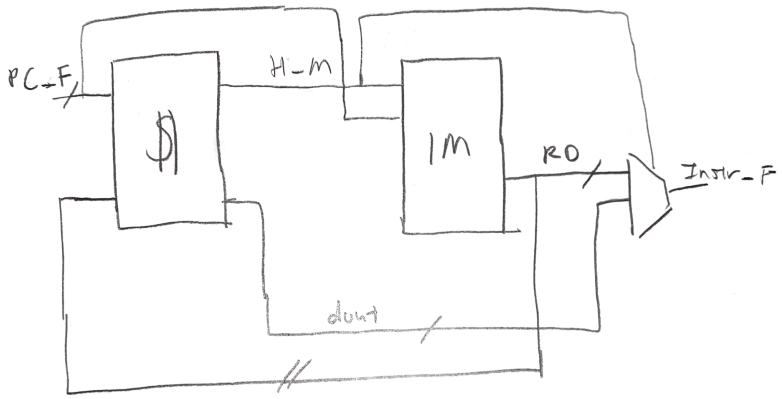


Figure 2. Schematic of Complete Instruction Memory Structure

The schematic diagram above represents the complete structure of the instruction memory which is used in the MIPS pipeline processor. Such an implementation allows for an efficient system which only accepts reads from the data memory. No instructions are written to the system; a memory file with all relevant instructions is initially loaded, and instructions are outputted based on the program counter. The instruction memory system acts similarly to the data memory system, except it does not take a write data signal from an external source.

Design Methodology

The data cache was implemented as a 2-way associative cache with a block size of 64 bits. The block size of 64 bits allows for two words to be spatially allocated to a single block, and such a design choice allows for a faster memory access time and a lower miss rate. Each set of the cache has two ways, and the memory system utilizes a LRU (least recently used) bit in order to know which block would be replaced given that both ways are filled. A system with two ways allows for a reduction in conflicts as a set can hold multiple words at a time, and there is a reduced chance that relevant data is accidentally lost if another word is mapped to the same address. In order to meet the 32 KiB area budget, the data cache is allocated 512 sets, each containing 2 blocks that consist of two 32-bit words.

The instruction cache was implemented as a direct-mapped cache with a block size of 64 bits. The design is fairly similar to that of the data cache, except the instruction cache only has one way to contain all instructions of the processor. Such a design choice was made as there were not enough instructions to fill the cache sets entirely, so there was no need for two ways as there was no chance for conflicts in the system. However, the design makes use of spatial locality, as the cache loads 2 words at a time after a miss.

Modules Designed

1. Data Cache
2. Instruction Cache

Ray Chang
Tim Kim
ECE 154B

3. Data Memory (revised)
4. Instruction Memory (revised)
5. Hazard Unit (revised)

Questions

Row major has a lower miss rate as row major stores words row after row, and matrix vector calculation multiplies each row of the matrix with the vector, so memory access is easier when blocks contain words that are in the same row. Miss rate would decrease if block size is bigger (or equivalent to the number of columns in the matrix). Column major has a higher miss rate because it stores words column after column, so when performing the calculation, the processor must access data that is not contiguous (must jump several memory addresses), and the cache block does not hold words that are in the same row.

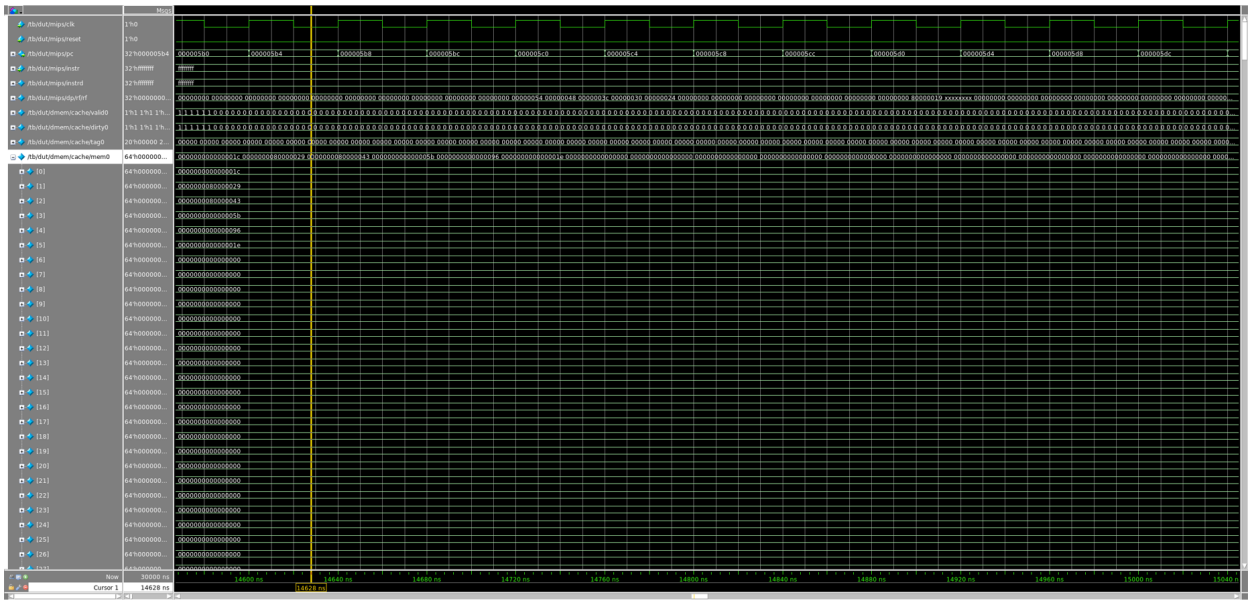
Unfortunately, due to minor errors in both the data and instruction memory systems, a fully successful matrix vector multiplication program could not be run. Although exact values cannot be stated, we do predict that the miss rate of the row major program is lower than that of the column major program due to the reasons stated above.

Area for the entire cache?

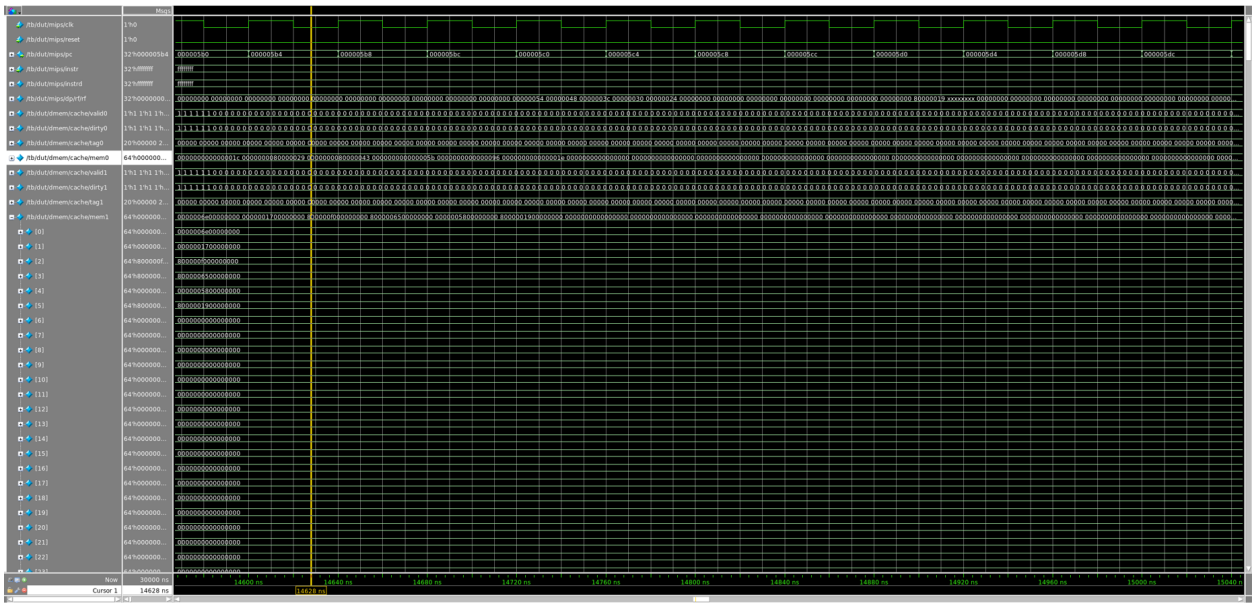
The approximated total area of the both caches implemented is $0.09\ m^2$.

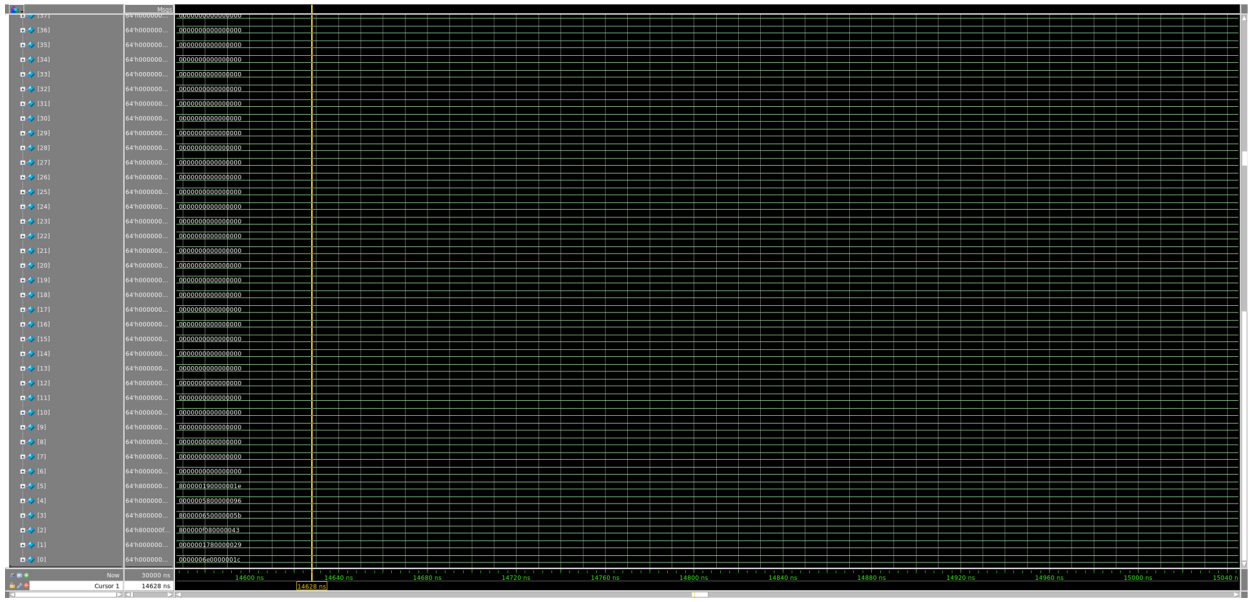
$$total\ area\ =\ 22KiB\ \times\ (\frac{1024\ bytes}{1\ KiB})(\frac{8\ bits}{1\ byte})(\frac{0.499\ \mu m^2}{1\ bit})\ =\ 0.09\ m^2$$

Row major vs column major

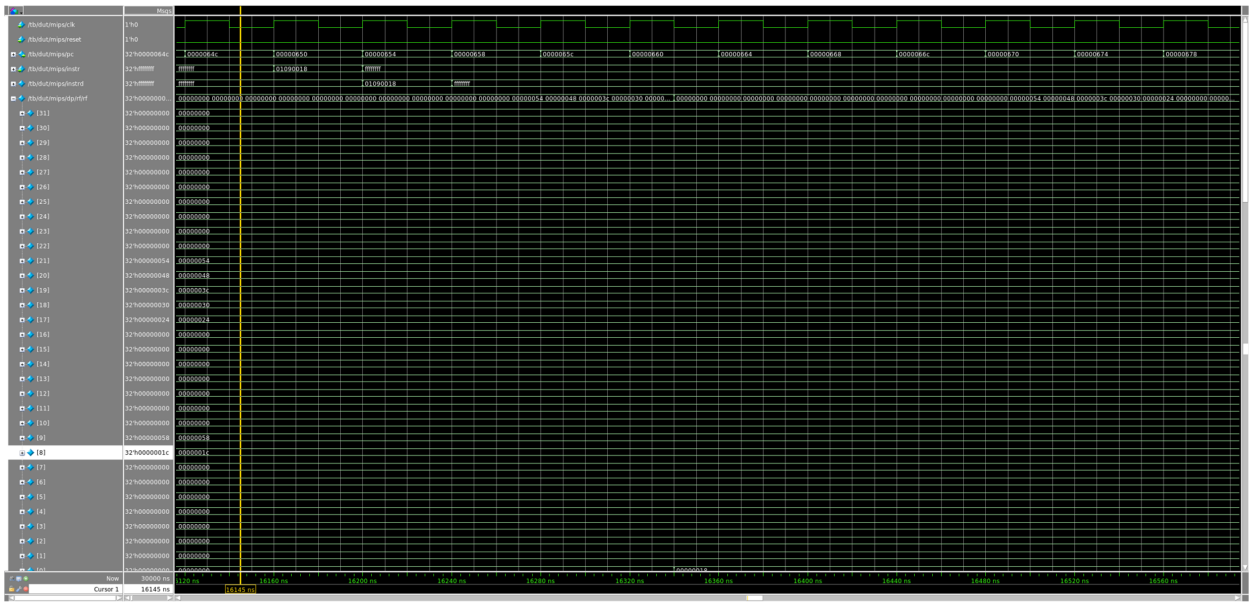


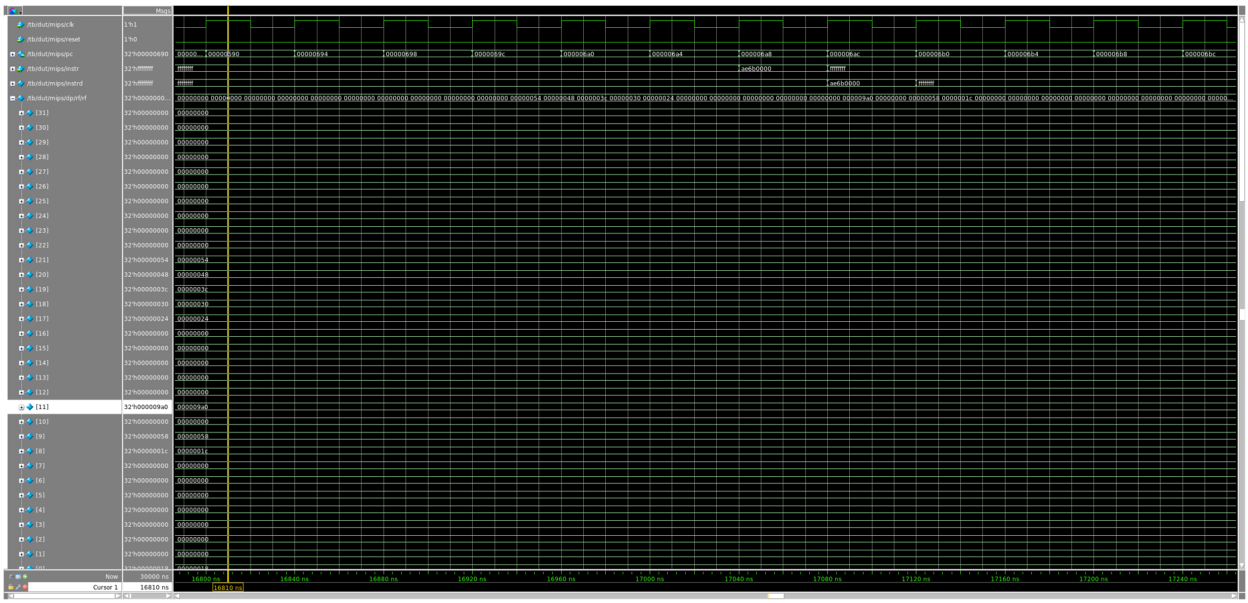
Ray Chang
Tim Kim
ECE 154B



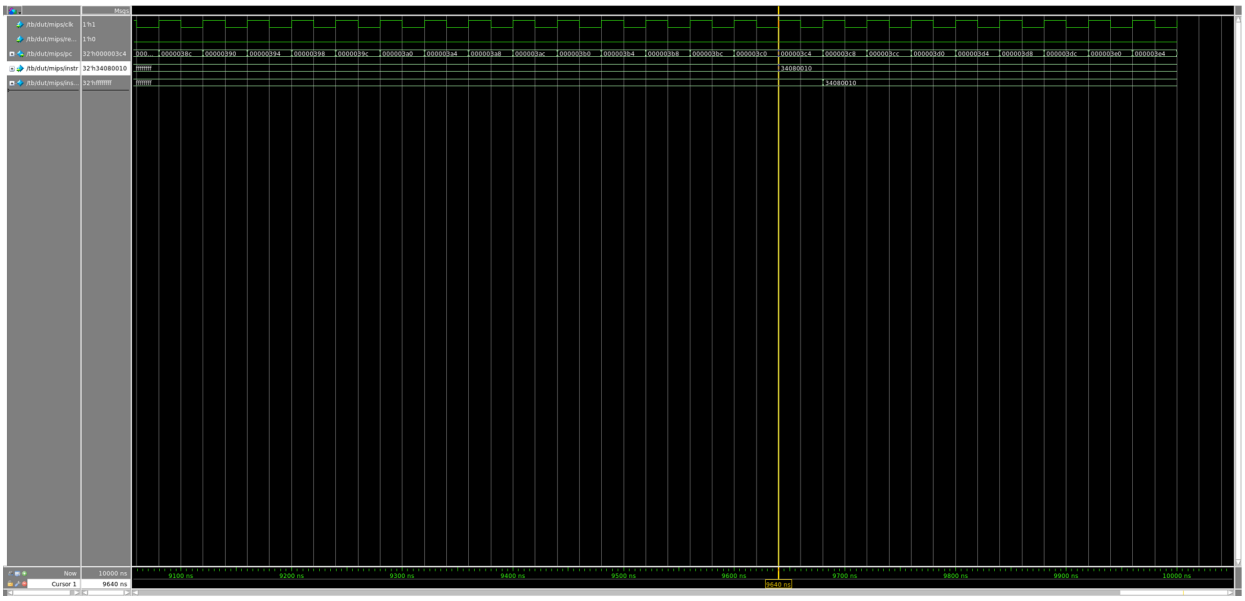


Ray Chang
Tim Kim
ECE 154B





Ray Chang
Tim Kim
ECE 154B



In dcache1.png, the waveform shows some of the values of the matrix and vector being stored into the LSB of each set in mem0, which is one of the ways of the data cache. Also, dcache2.png shows the remaining values being stored into the MSB of each set in mem1, which is the other way of the data cache. The dcache3.png shows values being stored into the first 6 rows of 64 bits of the data memory. The dcache4.png shows the multiplicand and multiplier of the first multiplication operation being loaded into the register file, and dcache4.png shows the correct product in 2's complement being loaded into the register file. The program intends to continue multiplications and additions to complete the matrix vector multiplication, but after this point the pipeline is unsuccessful in performing the remaining instructions. We believe there is an error with the read miss loading data from the wrong address into the cache, but we were unable to resolve this issue. Lastly, icache1.png shows the instruction cache loading data into the pipeline, but we saw an issue with our implementation where the program counter did not stall so that instructions would be loaded into the pipeline in the wrong order.

Conclusion

For this project we attempted to implement an instruction and data memory system that each involved a cache and main memory. In one version of the pipeline with only the data cache implemented, we were able to successfully perform store operations with sufficient nops in the program by adding to the hazard module. This allowed us to follow the write-through and no-write-allocate policies. We then saw two load operations with the correct output, followed by a successful multiplication, but the remaining output was incorrect. For the other version of our pipeline with only the instruction cache implemented, we saw instructions being loaded into the pipeline, but many instructions were skipped as we were unable to take care of the hazard

introduced by the 20 cycle delay of the main memory. We believe that once this hazard is resolved it can be combined with our data cache to produce an improved pipeline processor.