Project 5 Report

By Nick Vaughn

Problem:

The beta processor from previous designs did not keep a cache of accessed entries in memory. This can make retrieving data longer than necessary. To fix this, a one-word block cache must be implemented to keep record of the memory accesses used between read and write operations. The cache must have either a write-back or write-through policy and must have a cache size between 8-256 words for addresses of 32-bit size.

Design Approach:

To solve the problem, I chose to implement a direct mapped cache of 128 words. The reason I chose a direct mapped cache is because the design was more simplistic to implement conceptually in hardware, over other designs such as set-associative. It has an index that makes searching for relevant tags that exist in the cache efficient. Full associative would have been more complex to implement in hardware and possibly less efficient with speed, because it requires looking at every entry inside of the cache. I chose to use 128 words for the size, just ensure there were enough spaces for unique tags without overusing space. The memory is Byte aligned. Since the addresses are 32 bits wide, this gives me 7 bits used for the index [9:2] and 23 bits for the tags [31:10]. Valid bits are set when data is written to the cache. If data is read from memory that isn't inside of the cache, it will be added to an appropriate entry.

Analysis:

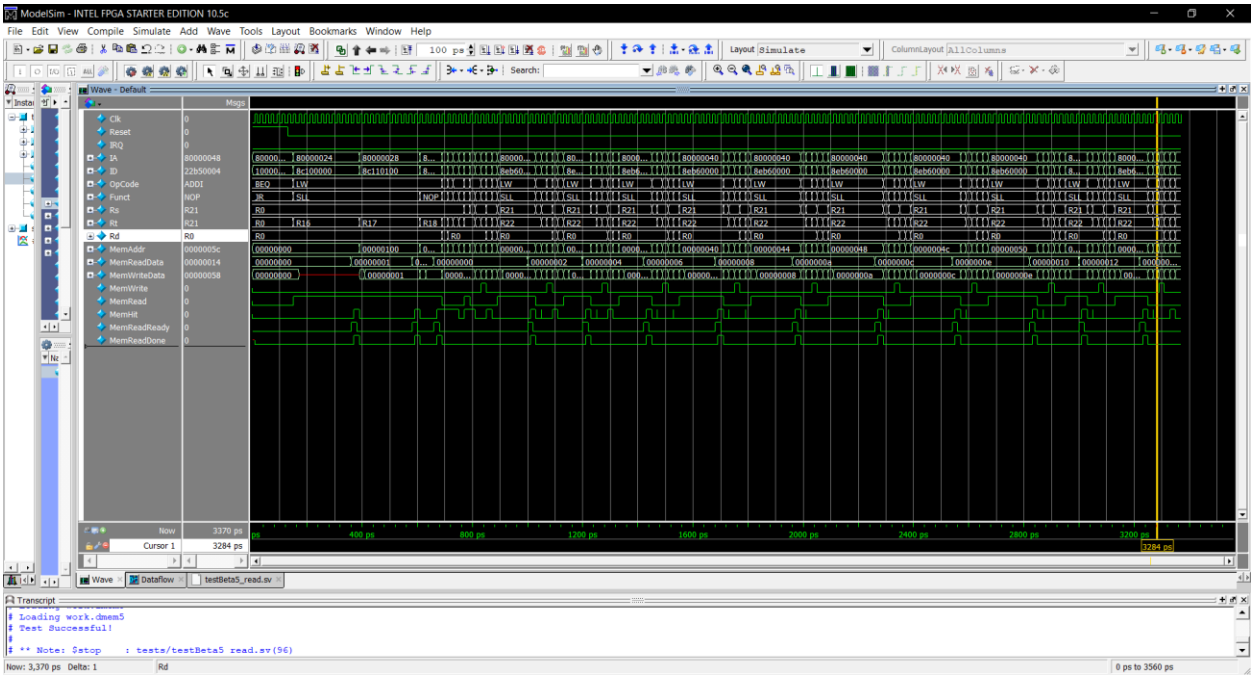

Figure 1 – Read only waveform

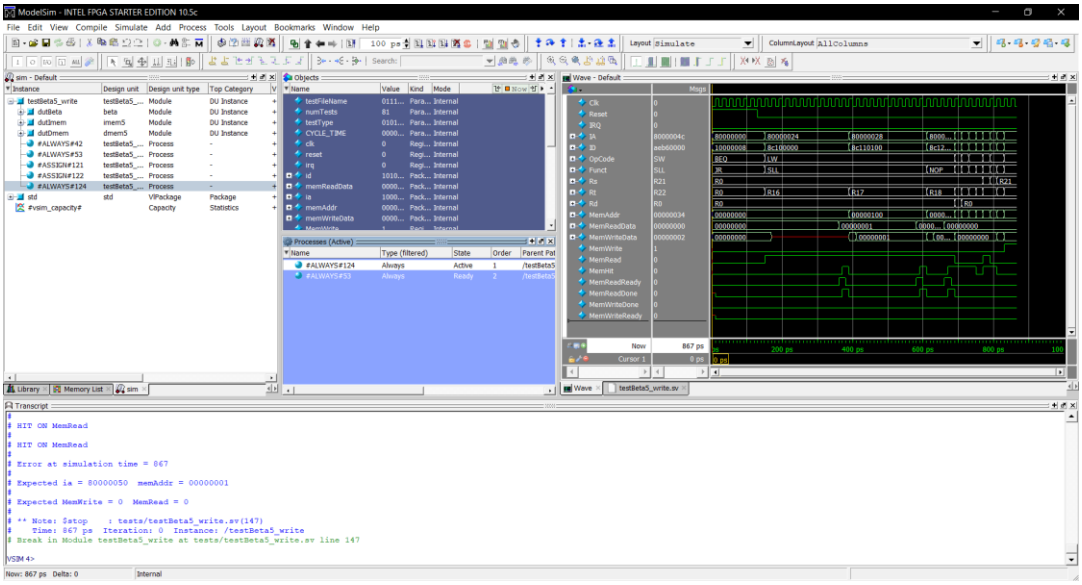

Figure 2 – Unsuccessful Write Through Waveform

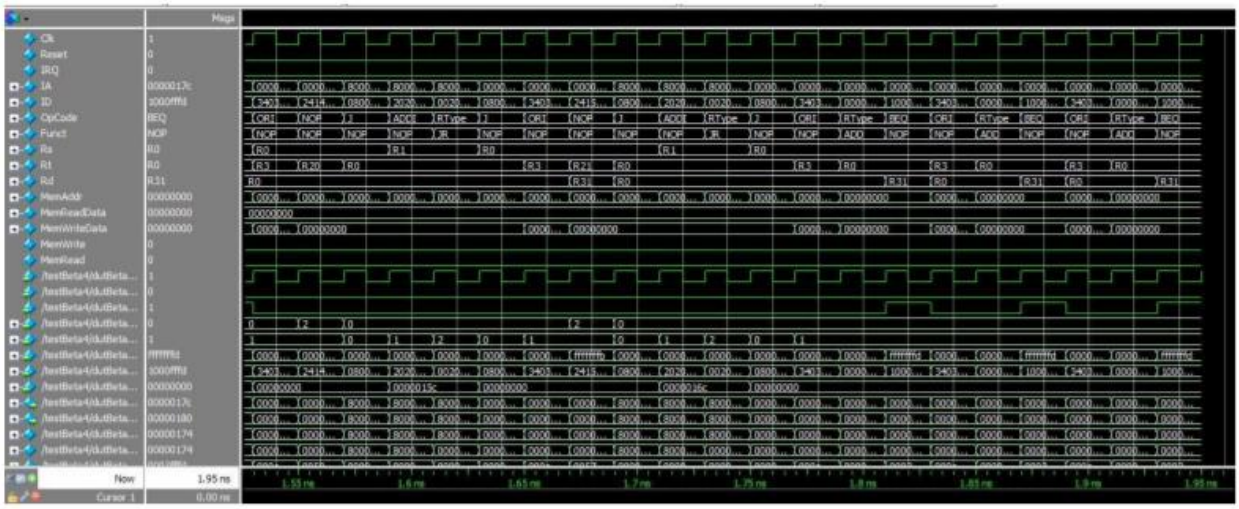

Figure 3 – Processor without the Cache

For the Analysis I will only be using the results of the Cache in figure 1 reading, because it was the only successful operation I was able to implement fully. The Miss Rate of the cache was 11/12 of the LW instructions as can be seen in figure 1. This makes sense, because having 7 bits for the index allows a lot of unique entries to the cache. At least more than a smaller cache size would. This also leaves a 23 bit tag (32 bits – 2 byte align bits – 7 index bits), which would most likely give us a lot of replacement inside of the cache when using an index. Since it is a direct map cache, it doesn't hold memory of previous addresses as much as a set associative cache would. This is the sacrifice I made for simplistic design schemes.

CPI:

For this data I visually analyzed how many cycles and instructions were inside the simulation and calculated the CPI. Cycles/instructions. Using figure 1 and 3. Note that period for a clock cycle is 20 ps. Simulation with cache had 3370 ps and the simulation without cache had 1950 ps.

CPI without cache:

1950/20 = 97.5 cycles

CPI = 97.5/ 92 = 1.06 cycles/instruction

CPI with Cache:

3370/20 = 168.5 cycles

CPI = 168.5/75 = 2.25 cycles/instruction

These results make sense because our cache had a lot of misses inside of it, causing it to stall. Plus the simulation without the cache was mostly show casing different type of arithmetic instructions more than LW or SW.

Total size:

Entries * (block_size+tag_size++valid)

128* (2^0*32+(32-(7+0+2)+1) = 7168 bits

Conclusion:

To get better miss results I would have implemented a 4-set associative cache. This way I would be able to store more than one value at a unique address location and increase my chances of getting a hit for a tag with the 128-word size. More hits would also increase my CPI overall. Also, a more organized structure would have allowed me to modularize my writes better, and perhaps get the functionality implemented easier.

Although using set associative would increase the amount of tags that are associated with one index, it would have also increased the bits used for the tag. The more bits that are inside of a tag, the more unique the entries are. This may have influence over the miss rate. If I were to commit to the set associative technique I would increase the amount

words my cache size has to the maximum 256 words. These two design changes would most likely increase my hits. The downside to this is the amount of hardware that must be implemented to use this design. My direct cache did not require the use of an LRU policy for each entry in an index. This gave me a huge convenience as a designer and saved me hardware space. The LRU algorithm would require tracking which entry in every index was used last, to decide which one to replace on a miss.

If I had enough time to design it, I would ditch my current design and implement the set associative design I mentioned above. Because it would give me better results for the CPI, and more efficient hit ratios.