Ellis Chen
05/05/2021
Project 5: Beta + Memory Report
Collaborators: Nathan Chanachai, Kevin Acosta, Cordero Forgach

Cache Design

My cache will be direct-mapped with 8 entries. Each entry in my cache will only contain a block with one word, so 8 words in total. My memory address will be byte-aligned, which means out of the 32 bits that come from the address, only bits 31 to 2 are used to search through the cache. My cache will be broken up into 3 sections: the 1st section of my cache is one bit long and will contain the valid bit, the 2nd section of my cache will hold 27 bits for my tag, and the 3rd section of my cache will contain 32 bits that hold the data. The width of each entry in my cache should be 60 bits wide. The proposed cache structure is shown in table 1. In terms of replacement policy, there really isn't because direct-mapped is only "one-way" in that the entry is determined by the index bits that come from the memory address.

The breakup of the memory address should provide the index bits and tag bits, which will be used to find the corresponding data in the cache. Again, my memory address is byte-aligned, so bits 1 and 0 are ignored. My cache will have 8 entries, so that means I need to reserve 3 bits for the index. Bits 4 to 2 of the memory address will be the index bits. The rest of the memory address, bits 31 to 5 are used for the tag bits.

For the reading functionality, the plan is to use the index bits from each new memory address and map them to the index bits of the cache, from there the tag bits from both the memory address and cache entry are compared with each other. Also, the valid bit is checked alongside the tag bits. If both the valid bits match and the valid bit in the cache is high, then a cache hit occurs. When a cache hit occurs, the 32 bits of data in that entry are read out into register block. If the tag bits do not match or the valid bit is not high, then a cache miss occurs and then the processor needs to stall until the cache can be updated from memory and the correct data is read out. If a miss occurs on a read, the program counter will stall and any write functionality to the memory will be momentarily be disabled. During the stall, the valid bit of the cache will have to be set high, the tag bits of the cache will be updated by bits 31 to 5 from the memory address, and the 32 data bits in the cache will be updated by data coming from memory.

My cache will use a write-through approach because it seemed easier to update the cache and the memory at the same time rather than the less synchronous approach of the write-back approach. I believe the process of writing to the cache is similar to the read and stall logic of the read functionality. I am sure there will be some additional logic or rewiring between ports in the beta processor to keep the memory up to date with the cache and vice versa.

Index (3 bits)	Valid (1 bit)	Tag (27 bits)	Data (32 bits)
000			
001			
010			
011			
100			
101			
110			
111			

Table 1: Proposed Direct-Mapped Cache Structure of 8 entries and 60 bits in width

Cache Read

The way that I instantiated the cache structure is similar to the procedure for creating the memory for the register block. Initially, the tag bits and index bits from the memory address are assigned to as separate logic from the cache structure. This way, I could compare the bits from the memory address to the bits from the cache entry to see if the tag bits matched and the valid bit was high. I found that any type of logic that reads from the cache would be better implemented in an always combinational block. I found that the logic that writes to the cache would be better implemented in an always flip flop on the positive edge of the clock. When I tried writing the valid bit, the tag bits, and data bits to the cache using combinational logic, the data was never properly written to the cache, which caused problems in later instructions.

The stall logic was quite tricky, as it needed a solid understanding of how the MemReadReady and the MemReadDone signals play their role in the stalling of the processor. After trial and error, I found that on a read miss, the processor must not stall while MemReadReady is 0 and MemReadDone is 1, since this basically means that the cache is done reading out the data and the beta processor is done waiting to receive the data. All other combinations of MemReadReady and MemReadDone should result in a stall. I noticed that the first load word instruction lasted from 150 ps to 690 ps of simulation time. Since each clock cycle seems to last 20 ps and the interval measured was 540 ps, that would mean that the first load word instruction that also indicates a miss in the cache lasted for 27 clock cycles. When the load word instruction had a cache hit, the processor did not stall, so the instruction only lasted 1 clock cycle.

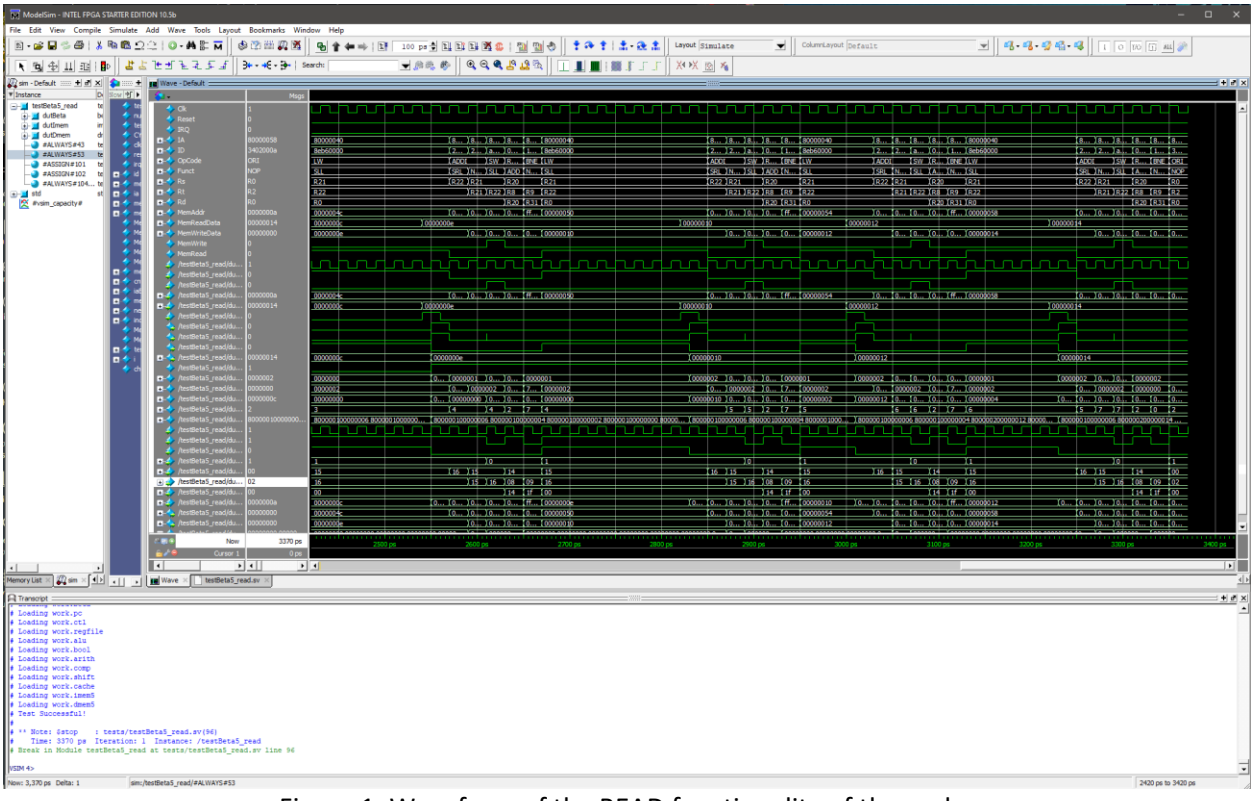

Figure 1: Waveform of the READ functionality of the cache

Cache Write

The logic for writing to the cache and the stalling logic for the write-through approach is similar to the logic of the read functionality. The only differences between the logics is the is that the write-through approach uses the memWriteData input from beta rather than the memReadData input to update the cache data bits. Also, I learned that the stall logic does not require checking if there was a hit or miss in the cache entry for write-through. During my debugging period, I found that the stall logic was working properly, but the cache and memory did not match, which meant that I had to make adjustments where the cache would feed out the write data to the memory in beta, and likewise the memory in beta had to feed into the cache to update it, this allowed for a mutual flow of data between the two. Also, a couple things were changed in terms of wiring in the beta module. The first change was replacing the second input to the ALU with the cache write out data. The second change was to rewire rbdata as an input for the cache write out data. These modifications to the sv files allowed the testbench to complete, but what raises questions is why there are various simulation times for the sw instruction. Some sw instructions lasted 5 clock cycles, some lasted 7 clock cycles, and some lasted 11 clock cycles. I understand that there is going to be some latency when writing to the cache, but this behavior raises more questions than answers, whether this is just a design flaw or the duration of the stall is dependent on the what type of data is being updated in the cache and memory. Maybe it just takes longer in some cases to write through to memory due to locality.

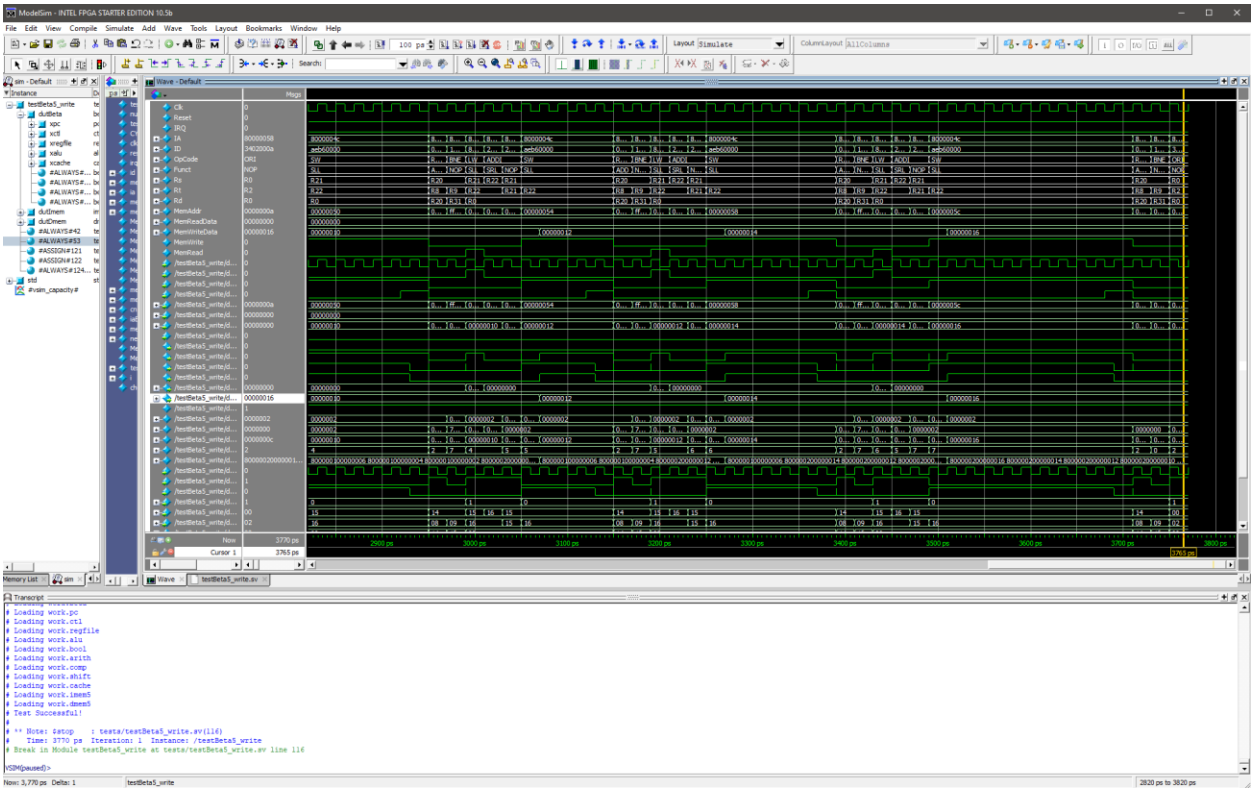

Figure 2: Waveform of the WRITE THROUGH functionality of the cache

Analysis of Cache Behavior

1. What is the miss rate of the cache?
   For the Read functionality of my cache, there were a total of 12 lw instructions, each one of those instructions were a hit in the cache except for only one. 1 out of 12 instructions that were simulated by the testbench resulted in a cache miss.

   For the Write functionality of the cache, there were a total of 11 sw instructions, every one of those instructions resulted in a cache miss initially for one clock cycle and then a cache hit for the rest of the duration. It seems that the miss rate is 100% for the sw instructions.

   Combining the sw and lw instructions, that would be a total of 25 instructions. Overall, 12 of those instructions in total means that the miss rate would be 48%.

	Load word	Store word
Hit	11	0
Miss	1	11
Total	12	11
Miss Rate (Miss/Total)	1/12 ≈ 8.33%	11/11 = 100%
LW/SW Total Miss Rate	12/25 ≈ 48%	

Table 2: Miss rates of the direct-mapped cache

2. What is the CPI of the Beta without the cache?
   The CPI of the beta without the cache is 1. The ia signals run synchronously to the clock, so for each rise in the clock, we perform one instruction.

$$\frac{1 \text{ cycle}}{1 \text{ instruciton}} = 1 \text{ CPI}$$

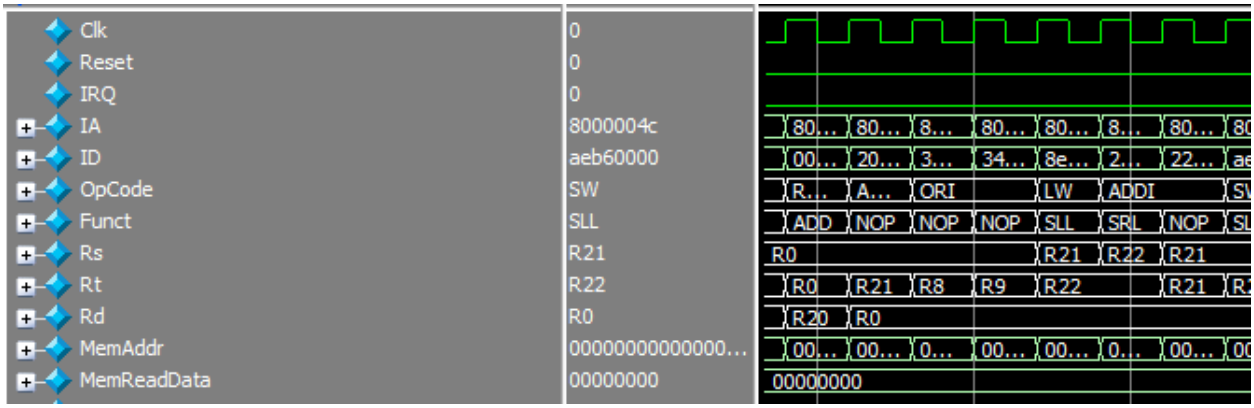

Figure 3: Waveform showing that most instructions without the cache will last one cycle per instruction

3. What is the CPI of the Beta with the cache?
   For the lw instruction, during a miss, it takes 27 cycles to complete the read from the cache. But ia changes at a maximum every 12 cycles and there are three times that ia changes, so the CPI for a read miss would be 27 cycles divided by 3 instruction or 9 CPI. During a hit, it only takes 1 cycle to complete the read, which is also the same for ia, which means the CPI would be 1 CPI.

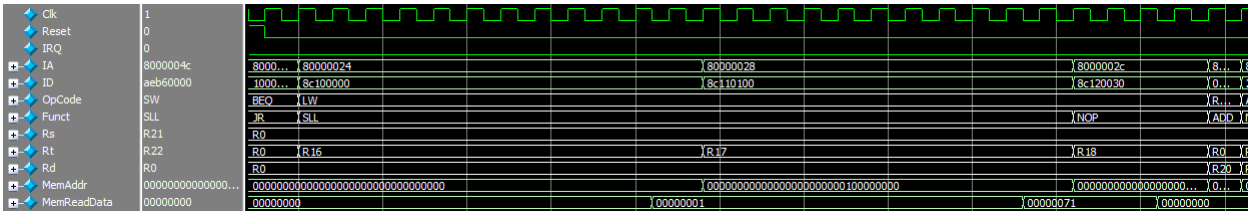

Figure 4: Waveform showing the amount of clock cycles needed for LW instruction to complete vs the number of instructions

The sw instruction had a few variabilities in its simulation time. It could take 5 to 7 to 11 clock cycles, but ia changed at the same time as the sw instruction every time. So, the worst-case CPI that I found for the cache write was 11 CPI, and the best-case I could find was 5 CPI. I am unsure why there exists a wide range of sw instruction times, this may be attributed to a design flaw of my caching system or some unknown factors within the testbenches are at work.

4. What is the total size of the cache?

$$\text{byte aligned}$$

$$32 \text{ bit addresses}$$

$$8 \text{ words} \rightarrow 2^3 \text{ words}$$

$$\text{block size} = 1 \text{ word} \rightarrow \text{m} = 0$$

$$\frac{2^3 \text{ words}}{2^0 \frac{\text{words}}{\text{block}}} = 2^3 \text{ blocks} \rightarrow \text{n} = 3$$

$$\text{tag size} = 32 - (\text{n} + \text{m} + 2) \Longrightarrow 32 - (3 + 0 + 2) = 27 \text{ bits}$$

$$\text{total cache size} = 2^3(1 * 32 + 27 + 1) = 480 \text{ bits}$$

The total size of my 8-entry direct-mapped cache is 480 bits

Modification

If I wanted to reduce the miss rate of my cache, I would most likely start off with increasing the number of entries in the cache. By doing this, there are more options for where the memory address maps to in the cache. If that does little to decrease the miss rate, then I would increase the block size. As long as I do not make the block size too large, then I can take advantage of spatial locality. Despite increasing the number of entries and increasing the block size, there is a tradeoff in terms of the number of tag bits we can use from the memory address. Greater number of entries and block sizes take up more bits from the memory address, leaving fewer tag bits left. Actually, this might not necessarily be a disadvantage, if the number of tag bits are reduced, the tags are more likely to match. If I wanted to make an even more drastic change to my design, I could increase the level of associativity. The greater the associativity, the less likely there will be a conflict between memory access that have the same block or index offsets. If I designed a fully associative cache with LRU policy instead, there would be less misses that could be caused by addresses with the same index or block offsets. The drawback of designing a fully associative cache however is the performance in terms of speed and cost of building one. If I had a choice between fully associative vs direct-mapped, I would choose direct-mapped because it is faster and the logic is less complex and less costly, even if direct-mapped generally results in a greater miss rate. In conclusion, I would keep the direct-mapped approach but increase the number of entries and block size.