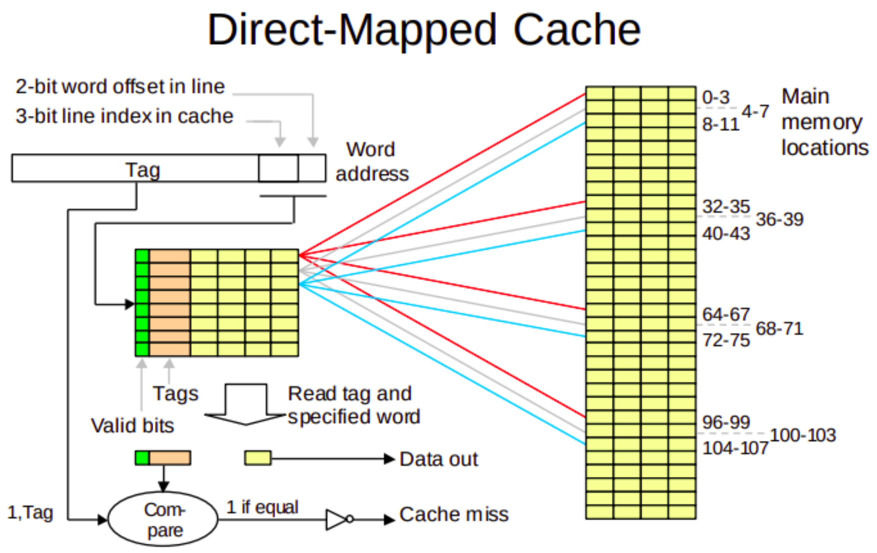
CME 433
Lab 5 Report
Tyrel Kostyk, tck290, 11216033

Lab 5 – Caching Part 3: Direct-Mapped and Set-Associative Cache

Part 1 – Multi-line Direct Mapped Cache

1. Please refer to the diagram of a direct-mapped cache provided in class as a reference:



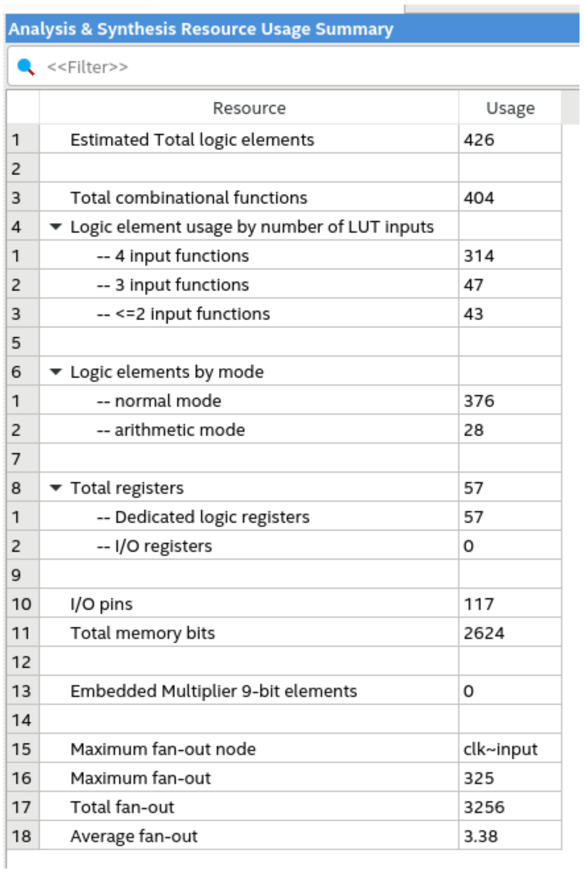
There are some key differences between the Direct-Mapped cache above and the one we implemented in Part 1 of this lab. First, we only had 4 lines in our cache instead of 8 shown above, and therefor also only needed 2 bits to represent the line number, not 3. Further, our word offset required 3 bits instead of 2, as our cache's lines had 8 words each, not 4.

So overall: compared to the diagram above, our cache had half the amount of lines to store, but also had twice as many words per line, so it ended up being the same amount of words in the cache.

2. Initially, (let's assume) the cache will be empty, **so the first cache request is definitely a miss.** Lets say it was address 6. The direct-mapped cache loads in the respective line into the cache (which in this case would hold words 0-7), filling the first line of the cache with the 8 words of data. Next, address 4 is requested – **it's a hit!** Memory is provided to the instruction decoder much faster than usual. Next, address 11 is requested **– it's a miss!** So now the second line of the cache fills up with words 8-15. Finally, let's say a request to memory address 40 is requested, and the *tag* for memory address 32 over laps with the *tag* for memory address 0 – **that's a cache conflict!** The cache will load in words 32-39, overwriting words 0-7 in the process. And the process continues!

3. See the following two photos showing the resource utilization of the single_cache vs the multi_cache. As you can see, the multi_cache uses significantly less combinational functions and logic elements. Additionally, the multi_cache is slightly faster than the singe_cache implementation.

Resource Utilization of the Single Line Cache

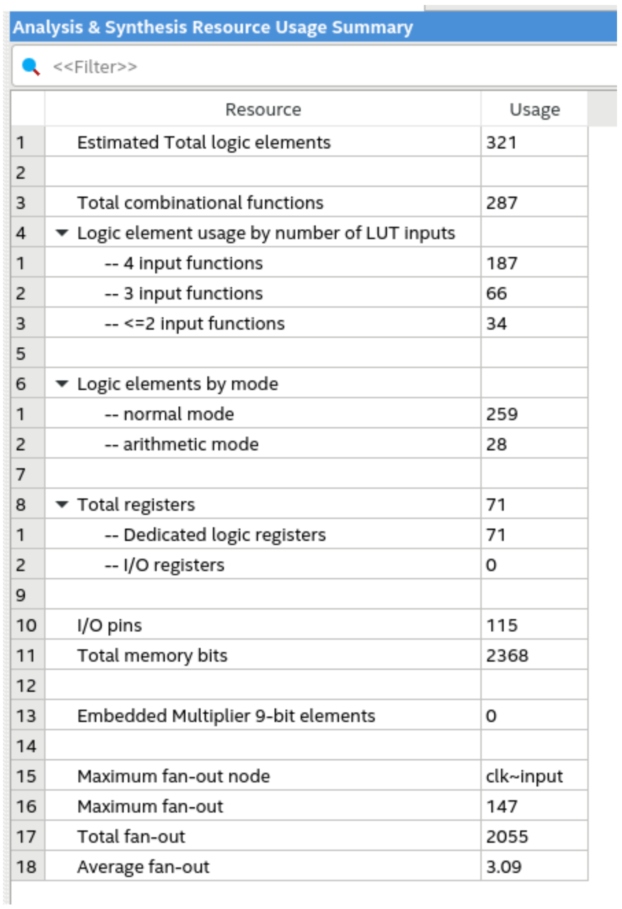


Analysis & Synthesis Resource Usage Summary

<<Filter>>

	Resource	Usage
1	Estimated Total logic elements	426
2		
3	Total combinational functions	404
4	Logic element usage by number of LUT inputs	
1	-- 4 input functions	314
2	-- 3 input functions	47
3	-- <=2 input functions	43
5		
6	Logic elements by mode	
1	-- normal mode	376
2	-- arithmetic mode	28
7		
8	Total registers	57
1	-- Dedicated logic registers	57
2	-- I/O registers	0
9		
10	I/O pins	117
11	Total memory bits	2624
12		
13	Embedded Multiplier 9-bit elements	0
14		
15	Maximum fan-out node	clk~input
16	Maximum fan-out	325
17	Total fan-out	3256
18	Average fan-out	3.38

Resource Utilization of the Multi-Line Cache

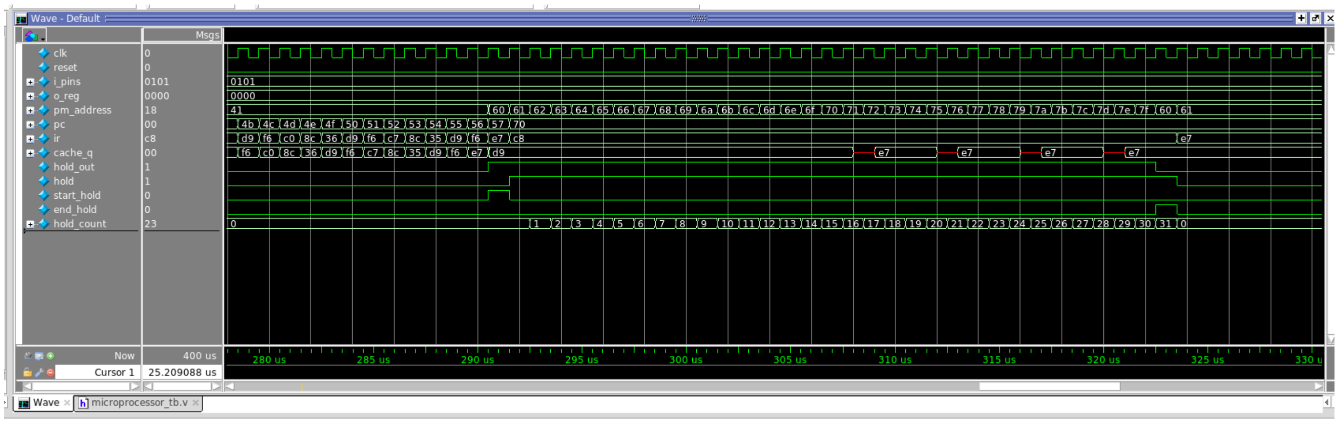


Analysis & Synthesis Resource Usage Summary

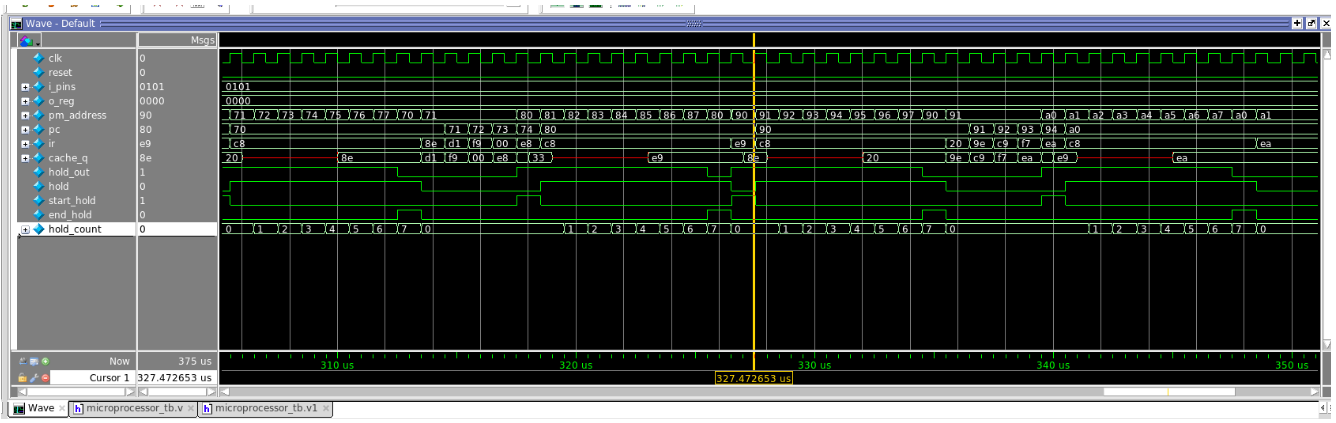
<<Filter>>

	Resource	Usage
1	Estimated Total logic elements	321
2		
3	Total combinational functions	287
4	Logic element usage by number of LUT inputs	
1	-- 4 input functions	187
2	-- 3 input functions	66
3	-- <=2 input functions	34
5		
6	Logic elements by mode	
1	-- normal mode	259
2	-- arithmetic mode	28
7		
8	Total registers	71
1	-- Dedicated logic registers	71
2	-- I/O registers	0
9		
10	I/O pins	115
11	Total memory bits	2368
12		
13	Embedded Multiplier 9-bit elements	0
14		
15	Maximum fan-out node	clk~input
16	Maximum fan-out	147
17	Total fan-out	2055
18	Average fan-out	3.09

4. See waveforms before



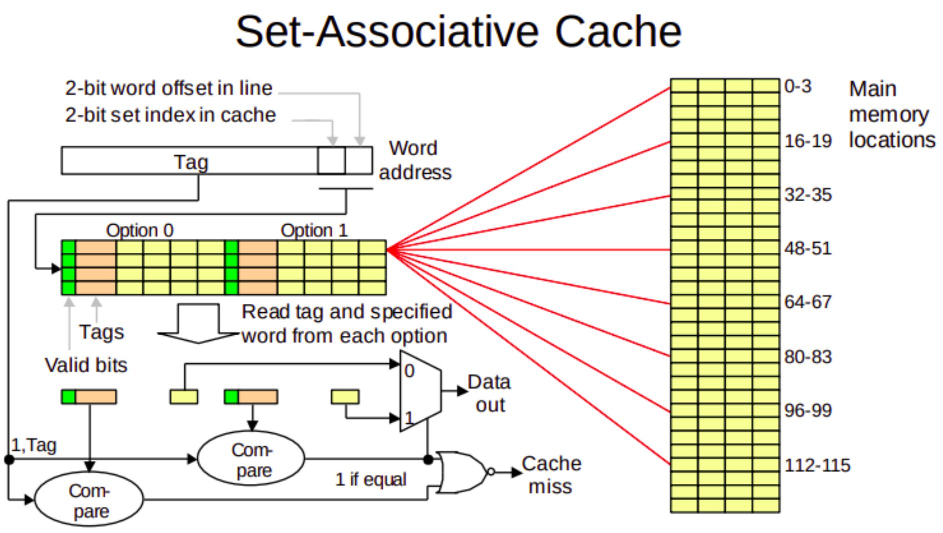
End of waveform for single_line_cache (cache_q is the data output from the cache)



End of waveform for multi_line_cache (cache_q is the data output from the cache)

Part 2 – Set-Associative Cache

1. Please refer to the diagram of a direct-mapped cache provided in class as a reference:

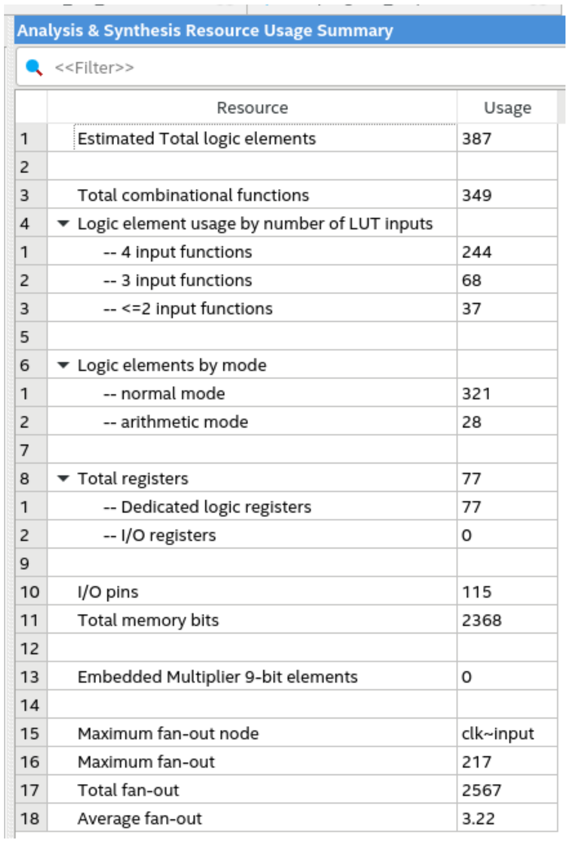


Just like in part 1, there are some slight differences in the diagram compared to our lab implementation: instead of 4 lines with 4 words each (per group), our design had 2 lines with 8 words (per group). So again same amount of words.

2. We implemented our set-associative cache using the Least-Recently Used (LRU) replacement scheme. First, with an empty cache – **a miss occurs,** and then the cache pulls in that line from main memory and then places it into either of the sets. **Another miss occurs,** but this time the cache places it in the empty set, but the same line as the last line that was fetched (let’s assume the memory addresses lined up like that). **A cache hit will occur** when the tag bits of the request match any of the tags of the cache lines in a set. **A cache miss occurs** when the tag bits don’t match any of the tags of the cache lines in the set. At this point, the LRU scheme fetches and places the new line in what it deems to be the “oldest” line.

3. See the following image of the Resource Utilization of the set_assoc_cache implementation.

As you can see, set_assoc_cache uses more combinational functions and logic elements than multi_cache, but uses less than single_cache. set_assoc_cache uses the same number of memory bits as multi cache, which is less than the amount used by single cache. set_assoc_cache is also the slowest of the 3 implementations.

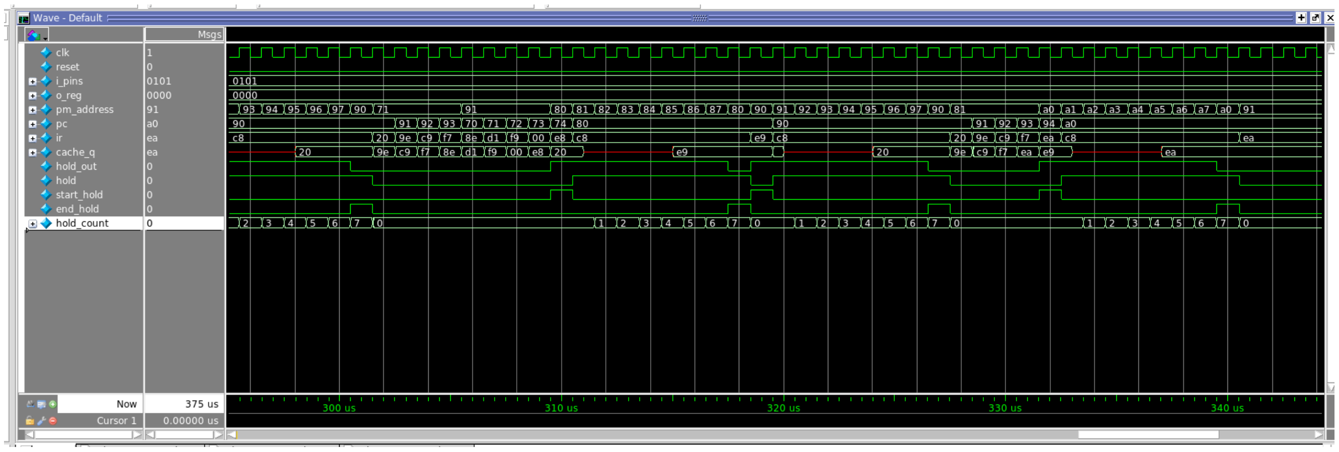


Analysis & Synthesis Resource Usage Summary

<<Filter>>

	Resource	Usage
1	Estimated Total logic elements	387
2		
3	Total combinational functions	349
4	Logic element usage by number of LUT inputs	
1	-- 4 input functions	244
2	-- 3 input functions	68
3	-- <=2 input functions	37
5		
6	Logic elements by mode	
1	-- normal mode	321
2	-- arithmetic mode	28
7		
8	Total registers	77
1	-- Dedicated logic registers	77
2	-- I/O registers	0
9		
10	I/O pins	115
11	Total memory bits	2368
12		
13	Embedded Multiplier 9-bit elements	0
14		
15	Maximum fan-out node	clk~input
16	Maximum fan-out	217
17	Total fan-out	2567
18	Average fan-out	3.22

4. See waveform below



End of waveform for set_assoc_cache (cache_q is the data output from the cache)

5.
- *tagID* represents the the requested memory address; it's what gets evaluated to check for a hit, miss, or conflict. The line offset is used to select which line (and therefor, which tag) to compare it to. It is updated every time start_hold is asserted.

- *valid* is used to determine whether a line has been written with data or not, and is an array (with 4 values), each holding a single bit. Gets updated whenever end_hold is asserted.

- *currdentry* is used to find the current entry for the cache.

- *lastused* represents the last used value in the cache.