Lab 3 Report: Cache

Group 71:
Yibin Xu (yx623)
Yanwen Zhu (yz2949)

Section 1: Introduction

This lab focuses on enhancing a pipelined processor model by incorporating both a data cache (D-cache) and an instruction cache (I-cache). "Through this lab, we gain hands-on experience in implementing the cache management strategies discussed in class, particularly in distinguishing between Instruction (I-Cache) and Data Caches (D-Cache). The baseline design involves implementing a 2kB, directly mapped, write-allocate cache with 64-byte blocks, ensuring parallel tag and data access, and handling back-to-back read hits efficiently. The cache must also write back all dirty blocks upon a flush signal and indicate completion with a flush_done signal. The alternative design expands on this by developing a 4kB, 2-way set associative, write-allocate cache, also with 64-byte blocks and using an LRU replacement policy.

Section 2: Baseline Design

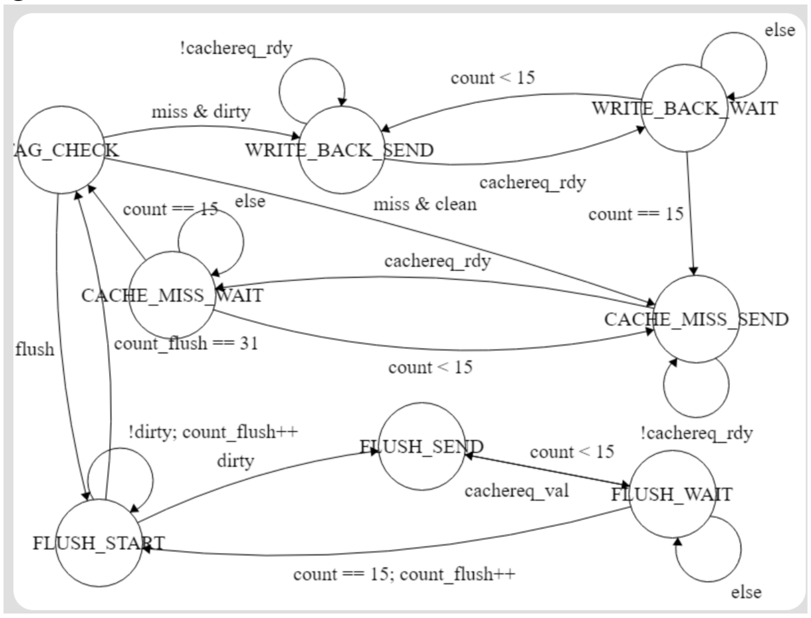
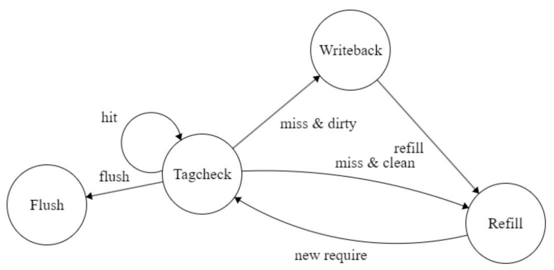


Figure 1 Finite State Machine for Baseline Design
(some information may not be contained in the figure due to limited page space)

In our baseline design, we initially identified eight states in Figure 1. However, to enhance clarity and simplify management, we have consolidated these into four main stages in Figure 2, each distinctly separated for easier oversight: Tagcheck, Refill, Writeback, and Flush stages.



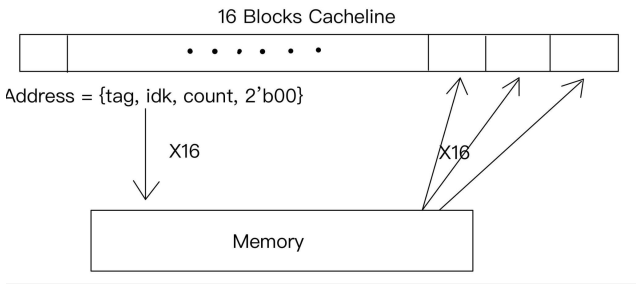


Figure 2 Four Main Stages In Baseline Design Figure 3 Data Transfer Between Cache And Memory

Due to the lab's requirement for consistent cycle output, the Tag Check state employs combinational logic to determine a hit or miss. In the event of a hit, the system remains in the current stage, outputs the corresponding response message, and awaits the next cache request. Conversely, in the case of a miss, the subsequent stage—either Writeback or Refill—is determined based on whether the condition is identified as clean or dirty. The Writeback and Refill stages share a similar structure. The Writeback stage is responsible for writing dirty cache lines back to the memory, while the Refill stage handles reading the target address from memory into the cache. Both the Writeback and Refill stages cycle between the Sending and Waiting stages, repeating this process 16 times. This is necessary because the baseline's size is 16 * 32 bits, and the memory can only process 32 bits of data per cycle. Therefore, to send a complete cache line, it is essential to loop through this process 16 times. The system transitions to the Flush stage only upon receiving a flush signal. This stage functions similarly to the Writeback stage, requiring 16 loops to completely flush a cache line. Additionally, it involves checking all 32 cache lines in the entity, and only those marked as 'dirty' are flushed.

As previously discussed, for each cache request message (cache_req_msg), it is necessary to cycle through the send and wait stages 16 times. During each iteration, the count is incremented by one. This count serves a dual purpose: it not only provides the requisite address but also tracks the number of loops completed.

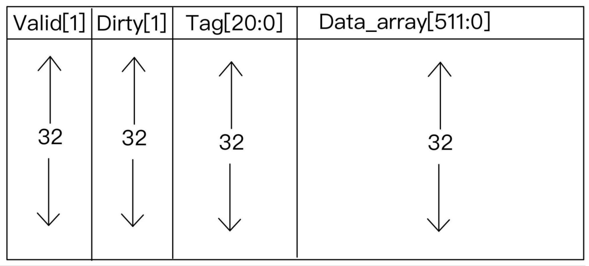


Figure 4 Data Structure for Cache

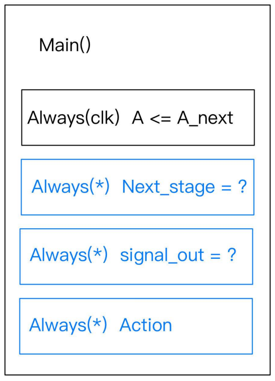


Figure 5 Structure for Baseline FSM

The foundational architecture of the design is structured around a Finite State Machine (FSM) framework. This framework is integrated with a singular sequential logic block and three distinct combinational logic blocks. The primary combinational logic block is tasked with determining the subsequent stages in the process. The second block is dedicated to ascertaining the output signals emanating from the cache modules. Lastly, the third block is designed to make decisions regarding the functional responses of the system.

Section 3. Alternative Design

The proposed alternative design introduces a 4kB, 2-way set associative cache, featuring 64-byte blocks and employing a Least Recently Used (LRU) replacement strategy. In addition to the baseline design, this configuration adds an extra data structure. This includes a [31:0] LRU component to monitor the most recently accessed set in the cache line. Additionally, a [31:0] SET element is integrated to specify which set of the data structure is to be utilized. At the onset of the Tag_check stage, the system first determines whether there is a cache miss or hit. In the event of a hit, it identifies which specific set has been accessed. Subsequently, the function focuses exclusively on the set that experienced the hit. Conversely, in the case of a miss, the system consults the LRU (Least Recently Used) table to pinpoint the last used set. It then proceeds to work on this particular set of data.
For instance, consider a scenario with three input miss requests. If the data array is initially empty, the default action is to select set zero as the primary data set and accordingly set the LRU (Least Recently Used) value to 0. During the second miss request, given that the LRU is still at 0, Set1 is chosen for use, as it is the least recently utilized set. Upon encountering the third miss, with the LRU now indicating 1, the system opts to revert to using set 0."

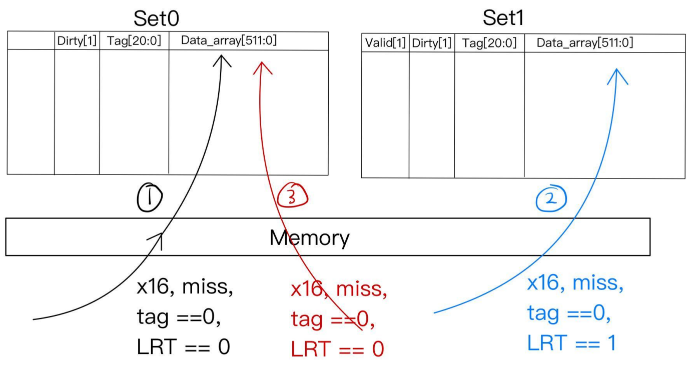


Figure 6 Example of three miss requires

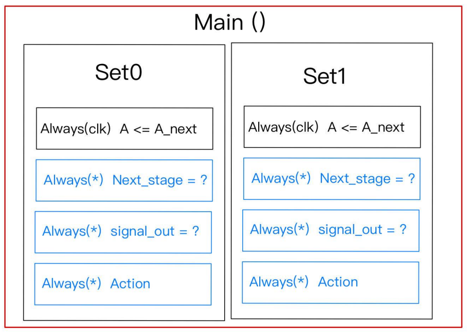


Figure 7 Structure for Alternative FSM

Before the cache processes the data, the function selects a Set to use for the current input instruction. This selection is based on whether there's a hit or it's determined by the Least Recently Used (LRU) algorithm

Testing Strategy – Describe the testing framework provided for testing your design. Describe the overall testing strategy. Be specific and explain why you used a specific testing strategy. Explain, at a high level, the kind of directed test cases implemented and why those were chosen. Consider including a table with a test case summary or some kind of quantitative summary of the number of test cases that are passing. We recommend starting this section with a short paragraph that provides an overview of your strategy for testing (so how all of the testing fits together). Then you might have one paragraph for each kind of testing. Each paragraph starts with the "why" (why that kind of testing) and then goes on to the "what" (what did you actually test using that kind of testing). Then you can end with a paragraph that pulls it all together and tries to make a compelling case for why you believe your design is functionally correct. Do not include the actual test code itself; your lab report should be at a higher level. If you include waveforms line traces, then you must annotate them so that the reader can understand what they mean (i.e., what corner case does the waveform or line trace illustrate?). For lab 4, you will also need to discuss your software testing strategy in addition to your hardware testing strategy. Remember to provide a balanced discussion between how you tested your design and why you chose that testing strategy and test cases.

Section 4: Testing Strategy
In this test, we employed both unit testbench and assembly code to test the baseline design and the alternate design. Our expected testing strategy is to ensure that all state machines operate as specified and that internal signals function correctly. Similarly, functional-level tests should run correctly. Initially, functional-level tests encompass read operations, where we verify that the cache accurately retrieves data for read operations, and write operations, ensuring proper handling of write operations, including the updating or eviction of cache lines as expected. Besides, we also verify cache flush operations to ensure correct behavior during system shutdown or context switching. Furthermore, our testing strategy extends to performance testing, where we measure the number of cycles under diverse workloads to evaluate the cache's efficiency. Stress testing is another crucial aspect, subjecting the cache subsystem to stressful conditions such as high traffic or rapid changes in data access patterns to evaluate its robustness and ability to handle extreme scenarios.

Testcase	Pass	Testcase	Pass	Testcase	Pass	Testcase	Pass	Testcase	Pass
add.asm	33/33	andi.asm	39/39	bge.asm	5/5	bltu.asm	5/5	bypass.asm	2/2
addi.asm	38/38	auipc.asm	11/11	bgeu.asm	5/5	bne.asm	5/5	jal.asm	10/10
and.asm	29/29	beq.asm	5/5	blt.asm	5/5	Branch.asm	25/25	jar.asm	5/5
Jump.asm	7/7	lui.asm	11/11	lw_2.asm	5/5	lw.asm	9/9	matrix.asm	5/5
mul.asm	9/9	or.asm	12/12	ori.asm	11/11	Memory.asm	3/3	RI.asm	12/12
RR.asm	12/12	sll.asm	12/12	slli.asm	15/15	slt.asm	11/11	slti.asm	15/15
sltiu.asm	15/15	sltu.asm	15/15	sra.asm	12/12	srai.asm	15/15	srl.asm	12/12
srli.asm	15/15	sub.asm	12/12	sw_2.asm	5/5	sw.asm	9/9		
Testcase				Pass	Testcase				Pass

ub_read_hit_clean.asm	5/5	ub_read_hit_dirty.asm	4/4
ub_read_hit_dirty.asm	4/4	ub_read_miss_refill_and_eviction.asm	1/1
ub_read_miss_refill_no_eviction.asm	4/4	ubench_vvadd_unopt.asm	2/2
ubench_vvadd_opt.asm	2/2	ub_write_hit_clean.asm	5/5
ub_write_hit_dirty.asm	4/4	ub_write_miss_refill_and_eviction.asm	4/4
ub_write_miss_refill_no_eviction.asm	5/5		

Table 1 Asm test case

The first is Unit test. The purpose of the unit test is to ensure that all state machines can execute and jump normally. For read/write hit paths for clean lines, only the TAG_CHECK stage is executed. For the test of the write operation, we input the specified memreq_msg to the DUT and assert the memreq_val signal, and then detect whether the corresponding data_array and tag_array are successfully written. The test of the read operation is similar to the test of the write operation, and checks whether memresp_msg.data meets the requirements. The test for read/write hit paths for clean lines is similar to the test for read/write hit path for clean lines. Next is the test of read/write miss with refill and no eviction. For the read operation, first check whether the state machine enters the CACHE_MISS_SEND stage, then after receiving cache_req_rdy, the state machine enters the CACHE_MISS_WAIT stage, and then the state machine enters the CACHE_MISS_SEND stage again until the counter count reaches 15. After completing the refill process, it enters the TAG_CHECK state and completes the read operation. The write operation is similar, except that after the counter reaches 15, the TAG_CHECK stage is entered for the write operation. Next is the test of Write/Read miss with refill and eviction. Here, it repeats and alternates 16 times to enter the WRITE_BACK_SEND state and enter WRITE_BACK_WAIT to write the cache with dirty bit 1 back to memory. Here, it tests whether the cache_req_msg is correct each time, and then enters the refill state, similar to the test for read/write miss with refill and no eviction.

Section 5: Evaluation

In comparison to the baseline, an alternate design typically yields superior hit rates and mitigates miss penalties. This improvement is attributed to the availability of two choices for block placement within each set, offering an additional location to locate the necessary block and consequently enhancing throughput. Conversely, direct mapping may experience heightened conflict misses due to its one-to-one mapping, particularly when multiple memory blocks are assigned to the same cache line. This can result in increased conflicts and potentially elevated miss penalties, leading to a reduction in throughput. The baseline boasts simplicity in implementation, requires minimal hardware, and exhibits lower energy consumption. Conversely, the alternate design, while more adaptable and capable of accommodating greater concurrent access, entails a more intricate implementation that may incur higher hardware costs and greater power consumption.

Test Function	ChcheI	ChcheD	ChcheAll	ChcheNone	ChcheAll2
Ubench_vvadd_opt	2282	10272	9860	2728	2346
Ubench_vvadd_unopt	3418	32894	31742	4612	3482
Matrix	1110	1178	1314	962	N/A
Iinstrction Chche Type	base	bypass	base	bypass	Alt
Data Chche Type	bypass	Alt	Alt	bypass	bypass

Cycle time:

The alternative design should reduce the cycle time overall. However, when the memory access delay (max_delay) is reduced to zero, an intriguing observation emerges: the baseline design actually requires less computation time than the alternative design. This is because a zero memory access time implies the absence of any miss penalties. For example, the flush_done needs to take both sets of data, instead of just one set of data. Consequently, the more complex structure of the alternative design results in longer processing times for the function. However if we increase the memory access delay, the performance of alternative design will increase, due to its bigger memory capacity.

Area:
The alternative design consumes more than twice the resources and area compared to the baseline design. This increase is attributed to the duplication of the data structure into two sets. Additionally, the inclusion of LRU (Least Recently Used) and SET data arrays further expands its resource usage. These arrays are essential for tracking and identifying the least used set, adding complexity and size to the overall design.

Energy:
The alternative design is anticipated to be more energy-efficient than the baseline design. This efficiency is attributed to the doubling of the data structure, which significantly increases the hit rate of cache requests. As a result, the processor expends less energy because it reduces the need for redundant waiting and minimizes the frequency of memory requests

Even though we were not able to complete the lab entirely, here is the function that works:

- We pass all of our own asm code for cacheI.
- We pass all of our own asm code for cacheD, cacheAll, except we have memory differences in ubench_vvadd_opt.asm and ubench_vvadd_unopt.asm.

Section 6: Conclusion
In conclusion, the alternative design is anticipated to outperform the baseline in terms of processing time, primarily due to its enlarged data structure. This expanded structure allows for storing only two distinct data sets per tag in the best-case scenario. Unlike the baseline design, the alternative consistently achieves hits, paving the way for an almost 100% hit rate. This significant increase in efficiency underscores the superiority of the alternative design for processing tasks. As we have mentioned before, the bigger memory latency should ley better performance for the alternative design compared with the baseline design. In conclusion, the baseline design has simplicity in implementation, requires minimal hardware, and exhibits lower energy consumption, but it has high conflict misses, leading to a reduction in throughput. The alternative design has high hit rates, less miss penalties and high throughput. The choice between these two designs depends on the trade-off analysis based on the specific need. When less hardware complexity, low power consumption and low cost are needed, I think the baseline design is preferred. But when lower conflict misses and more flexibility are needed, I think alternate design is preferred.

Section 7 Work Distribution
Yibin Xu work on the Baseline and Alternative designs. Yanwen Zhu help with the Baseline design and focused on crafting unit tests. Both work together for the lab report.