Project 4

- Pipelined MIPS with Cache -

Prepared by

Lee Chang Yoon

32183641@dankook.ac.kr

Left Days: 5
June 20, 2022

Table of Contents

1. Introduction

Cache memories are small, fast SRAM-based memories that are managed automatically in hardware. These memories hold frequently accessed blocks of main memory (*Bryant & O'Halloran, 15-213 (18-213): Introduction to computer systems (ICS) 2010*). In the Von-Neumann computer, memory access operation is very frequent because all of the instructions and data are stored in the main memory, and memory access is much slower than accessing the register. This latency of memory access generates a stall in the pipeline. To resolve this problem, modern CPUs use this cache memory.

In this report, we will mainly discuss the cache operation, implementation, and evaluation in the pipelined microarchitecture. The works presented in this report are the last part of a large project designed to implement and optimize the microarchitecture that uses MIPS ISA. We will add the concept of the cache into the pipelined microarchitecture to enhance the performance of the microarchitecture.

The first step in this project is to specify the requirements for the cache structure. Second, we will move on to the concepts for the cache: memory hierarchy, memory locality, performance, types of the cache, cache replacement policies, and write-back policies. Third, we will state the program definition for our cache implementation. Fourth, we will describe how we implemented the cache with various cache replacements and write-back policies. Then, there will be some results for each execution of the binary program by the cache MIPS simulator. At the end of this report, we will evaluate the cache MIPS simulator with the performance comparison, and the memory latency based on some assumptions.

2. Requirements

Index	Requirement
1	Understand cache structure and analysis of cache performance. a) Program should produce the correct output. b) Compare the cache hit/miss and average memory access time (AMAT).
2	Before execution, the binary file is loaded into the memory. If PC is 0xFFFF:FFFF, it completes execution, and halts. a) Initial value of register RA is 0xFFFF:FFFF. b) Initial value of register SP is 0x100:0000. c) Initial value of other registers are 0x0000:0000.
3	Example assumptions: a) Cache line size is 64 bytes. Set-associativity can be various (Direct-mapped, 2- way, 4-way, 8-way, etc.), configured before the execution. Cache size (data store size) can also be configured before execution (64 bytes, 128 bytes, 256 bytes). b) The cache should implement a proper replacement algorithm. c) The cache should implement a proper write policy. d) Cache access latency is one CPU clock cycle time; memory access latency is 1000 CPU clock cycle time.
4	The simulator prints out the statistics from the execution at the end of the program execution. a) total number of cycles of execution. b) number of memory (load/store) operations. c) number of register operations. d) number of branches (total/taken). e) cache hit/miss (cold miss or conflict miss).
5	Program completion/terminal condition: a) At the end of the execution, we need to print out the calculated value. b) If PC moves to 0xFFFF:FFFF, the program prints out the result and terminates. c) The final return value is stored in V0 (or r2) register.

Figure 1 - Requirement Specification

Figure 1 shows the requirements for a cache MIPS simulator. The implementations for these requirements will be described in detail afterward.

3. Concepts

Before we get into the description of the cache MIPS simulator, we will briefly discuss the concepts that are mainly used in the implementation: memory hierarchy, memory locality, performance, types of the cache, cache replacement policies, and write-back policies.

3-1. Memory Hierarchy

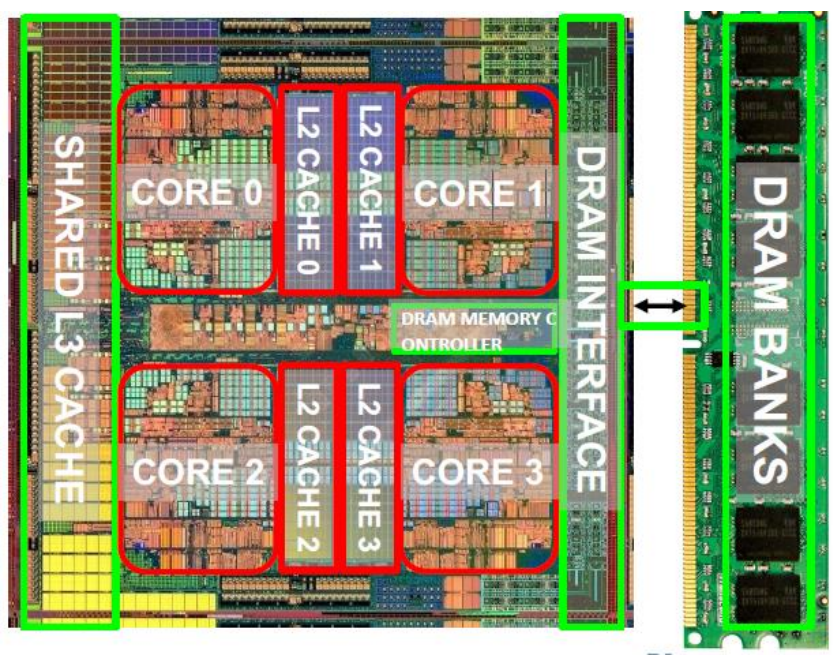


Figure 2 - Memory Hierarchy (*Yoo,, 13-memory_intro*)

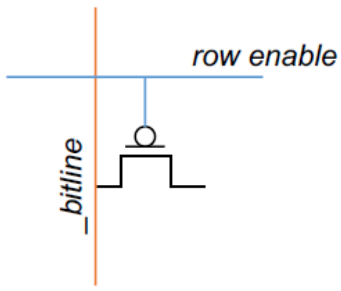


Figure 3 - DRAM (*Yoo, 13-memory_intro*)

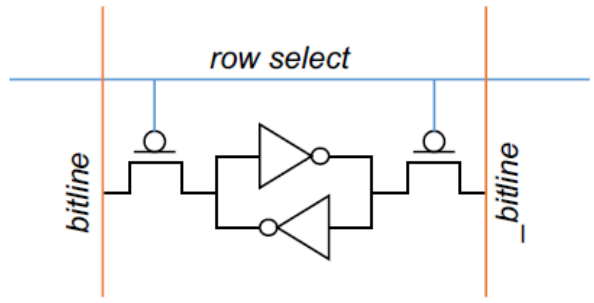


Figure 4 – SRAM (*Yoo, 13-memory_intro*)

Memory can construct by dynamic random-access memory (DRAM) or static random-access memory (SRAM). DRAM has consisted of multiple capacitors that store value with the charged state. If the capacitor is charged, the value is 1, and if not, the value is 0. Capacitor leaks through the RC path. Therefore, the DRAM cell loses the charge over time, and to avoid this situation, the DRAM cell needs to be refreshed by charging the state of the cells. SRAM has consisted of multiple two-crossed coupled inverters that store a single bit. The Feedback path in the following component enables the stored value to persist in the cell. SRAM needs 4 transistors for storing the value, and 2 transistors for accessing the value. DRAM shows the characteristic of slower access, higher density, and lower cost. It also requires the refresh of the cells and the manufacture of putting capacitor and logic together. However, SRAM shows the characteristics of faster access, lower density, and higher cost. It does not require an additional refresh of the cell, and its manufacture is compatible with the logic process.

The problem is that the ideal memory requires the opposite characteristics of both two memory types. If we want to expand the size of the memory, the accessing time gets slower. If we want faster memory accessing time, it requires a higher cost of money. To achieve both using the larger size and faster accessing time memory, the concept of memory hierarchy has come out.

The idea of the memory hierarchy is to have multiple levels of storage (progressively bigger and slower as the levels are farther from the processor) and ensure most of the data the processor needs is kept in the memory with faster accessing time (*Yoo, 13-memory_intro*). Figure 2 shows the memory levels with the main memory, cache, and registers in the CPU. L3 cache is the shared cache among the multiple cores, L2 cache interacts with the L1 cache in the single core. L1 cache is divided into instruction and data cache, and they are mainly accessed by single core.

3-2. Memory Locality

The locality is the concept of one's recent past referencing history can be a very good predictor of the near future. Locality can be divided into two main types: temporal locality, and spatial locality. Temporal locality means that if one just did something, it is very likely that one will do the same thing again soon. Spatial Locality means that if one did something, it is very likely that one will do something similar or be related to it.

Now, let's apply the concept of the locality to memory. Typical programs have a lot of locality in the memory references, and this locality can be also divided into the temporal and spatial locality. In the aspect of temporal locality, a program tends to reference the same memory location many times and all within a small window of time. In the aspect of spatial locality, a program tends to reference a cluster of memory locations at a time, such operations as instruction memory references and array structure references.

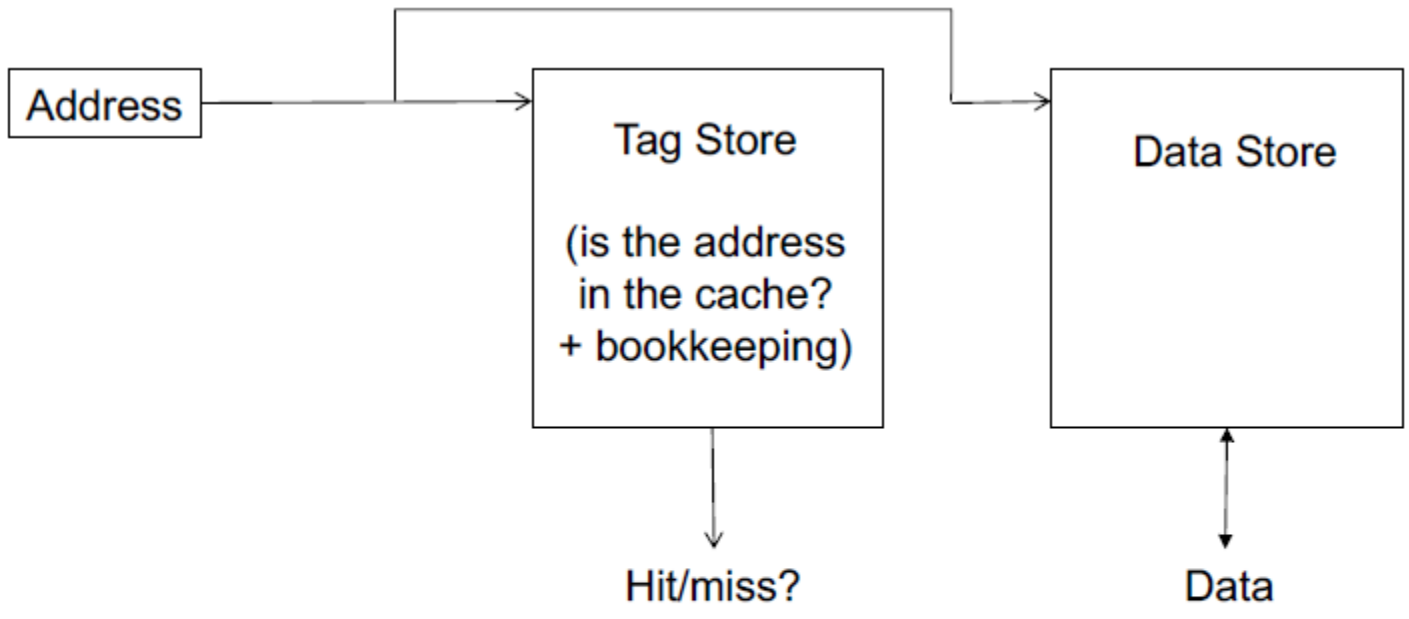
The locality explained above is the main idea of the cache. Cache stores recently accessed data that is automatically managed in fast memory. The cache contains both principles of temporal and spatial locality. To achieve both localities, the cache logically divides the memory into the equal size blocks and fetches to cache the accessed block in its entirety.

3-3. Hierarchical Latency Analysis

Let's denote that the given memory hierarchy level i has a technology-intrinsic access time of t_i, and the perceived access time T_i is longer than t_i. Except for the outer-most hierarchy, when we look for a given address, there is a chance (hit-rate h_i) for hit, which has the accessing time of t_i. Also, there is a chance (miss-rate m_i) for miss, which has accessing time of $t_i + T_i$. According to these accessing time, we can get the recursive latency equation which is $T_i = t_i + m_i * T_i$. The goal of the following equation is to desire T_1 within allowed cost. To achieve the following goal, we should keep the miss-rate m_i and T_{i+1} low for the faster lower hierarchies.

The hierarchical latency analysis which is explained above will be used in section 8, evaluation of the cache with different structures. By using the following analysis, we can measure the performance of each cache structure.

3-4. Cache



- Cache hit rate = (# hits) / (# hits + # misses) = (# hits) / (# accesses)
- Average memory access time (AMAT)
  = (hit-rate * hit-latency) + (miss-rate * miss-latency)
- Aside: *Can reducing AMAT reduce performance?*

Figure 5 - Cache Abstraction and Metrics (*Yoo, 13-memory_intro*)

A cache generically means any structure that memorizes frequently used results to avoid repeating the long latency operations required to reproduce the results from scratch. Most commonly, an automatically managed memory hierarchy is based on the SRAM, to avoid the repeated cost of paying for DRAM access latency (*Yoo, 13-memory_intro*).

Figure 5 shows the structure of simple cache abstraction. The cache has consisted of a tag store and a data store. Tag store contains the information that can be used to check whether the given address is repetitively referenced or not, which is a cache hit and cache miss. In the data store, it stores the data in the unit of the data block, which is also called a cache line. These blocks are generated by dividing the memory logically into cache blocks that map to locations in the cache. When the data is referenced, it searches data from the cache first. If the data is in the cache, the cache is hit, and the CPU use cached data instead of accessing the memory. However, if the data is not in the cache, then it brings the block from the lower memory into the cache.

The addressing of the cache is operated in the following sequence. Each block, cache line, maps to a location in the cache, which is determined by the index bits in the address. By using the given index, we can index the cache into the tag and data stores, check the valid bit in the tag store, and compare the tag bit in the address with the stored tag in the tag store. If they are the same, then it is the cache hit.

There are some important design decisions in the cache: placement, replacement, management, writing policies, and separation of instruction and data memory. These topics will be discussed in the later sections.

3-4-1. Direct Mapped Cache

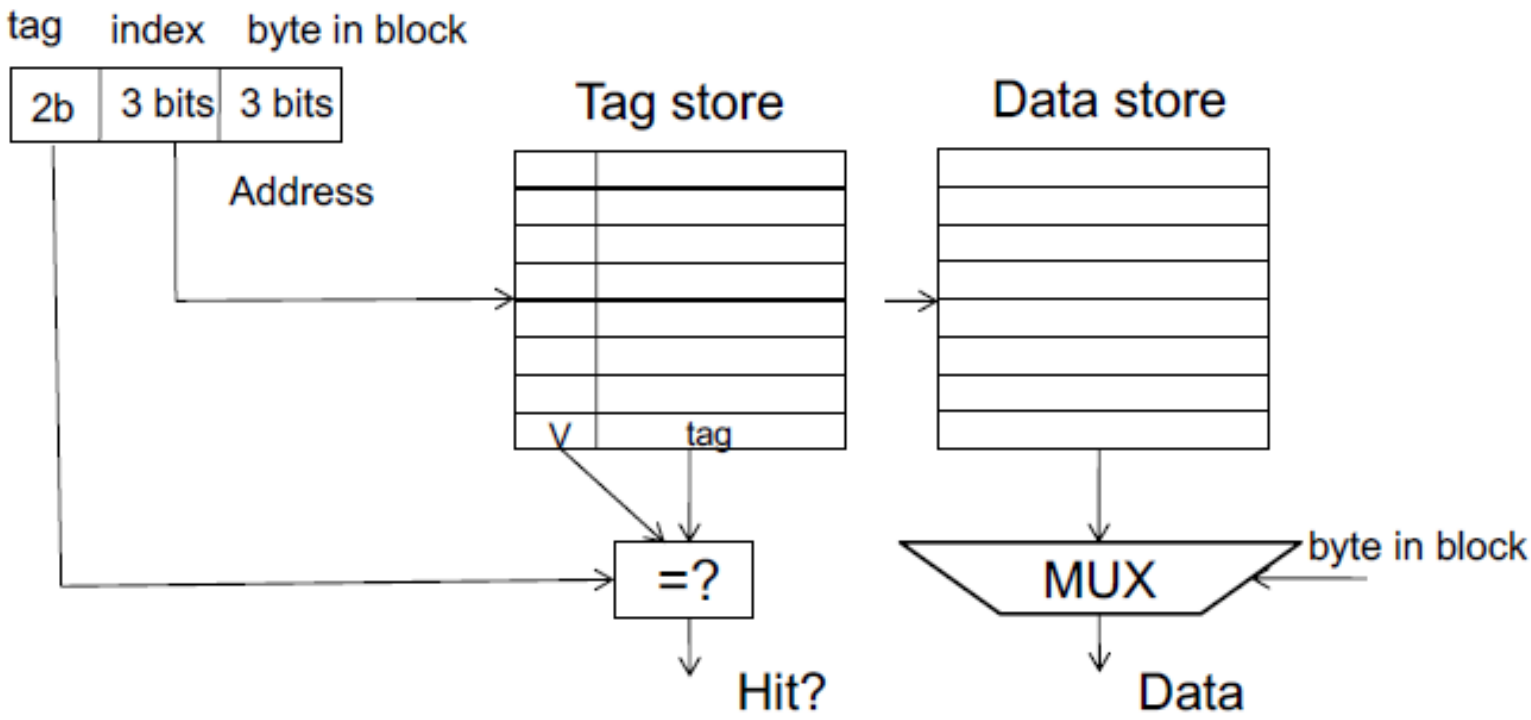


Figure 6 - Direct Mapped Cache (*Yoo, 13-memory_intro*)

The first structure for the cache is a direct-mapped cache. In a direct-mapped cache, two blocks in the memory that are mapped to the same index in the cache cannot be presented in the cache at the same time, which means that the only one index is matched with the one entry.

To implement this method, we divide the address into three parts: tag, index, and byte in the block (offset). The CPU accesses the tag store by using the index, checking the valid bit, and comparing the tag with the stored tag. If tags are the same, it uses the data that is placed in the data block with the distance of the offset.

As we can check from the operation of the direct-mapped cache, the CPU uses the index bits from the address, so it needs only one cache access to check if there is required data or not in the cache. It seems like the following method is efficient due to the single access for searching the data in the cache, however, it has its limitation. This method can lead the hit rate to 0% if more than one block is accessed in an interleaved manned map to the same index. If we assume the addresses A and B have the same index bits from the different tag bits, and the reference sequence is A, B, A, B, …, all of the cache access will occur the cache miss. Therefore, we need to find out a better way to resolve this limitation of direct-mapped cache.

3-4-2. Set Associative Cache

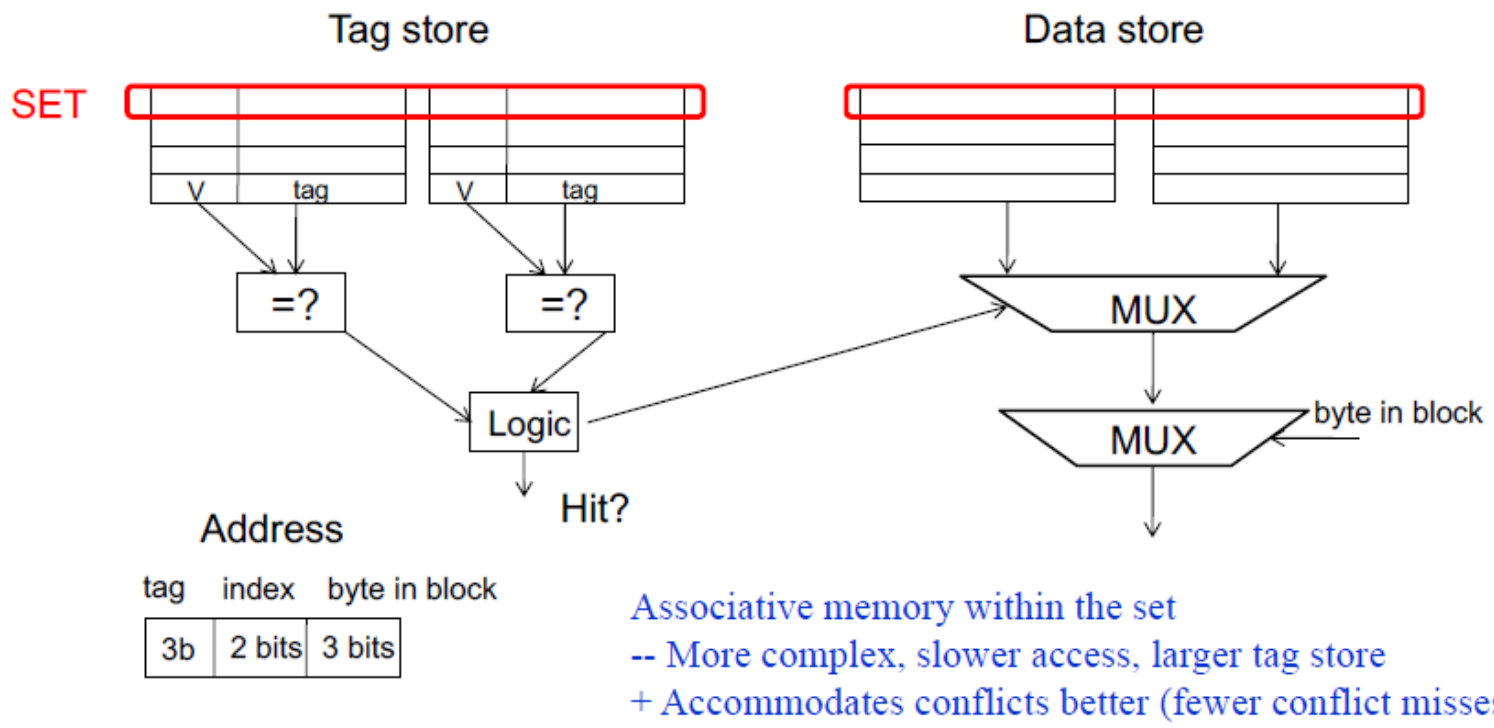


Figure 7 - Set Associative Cache (*Yoo, 13-memory_intro*)

The second cache structure is the set-associative cache, which is also called an N-way set-associative cache. An N-way-set associative cache reduces the conflicts, which was the limitation of the direct-mapped cache, by providing N blocks in each set where data mapping to that set might be found, and hence, a direct-mapped cache is another name for a one-way set associative cache (*Harris & Harris, 2012*).

To implement the following cache, we divided the address into three parts: tag, index, and bytes in block (offset). By using the given index, we search the set of ways in the following index. Each wat consists of a data block, valid bit, and tag bits. The cache reads blocks from both ways in the selected set and checks the tags and valid bits for a hit. If a hit occurs in one of the ways, a multiplexer selects data from the following way.

Set associative cache generally has lower miss rates than the direct-mapped cache of the same cache capacity because it has fewer conflicts. However, the set-associative cache is usually slower and more expensive to build due to the output multiplexer and additional comparators. It also raises the question of which way to replace when all of the ways are full. In the later sections, we will describe the methods to resolve the following situation.

3-4-3. Fully Associative Cache

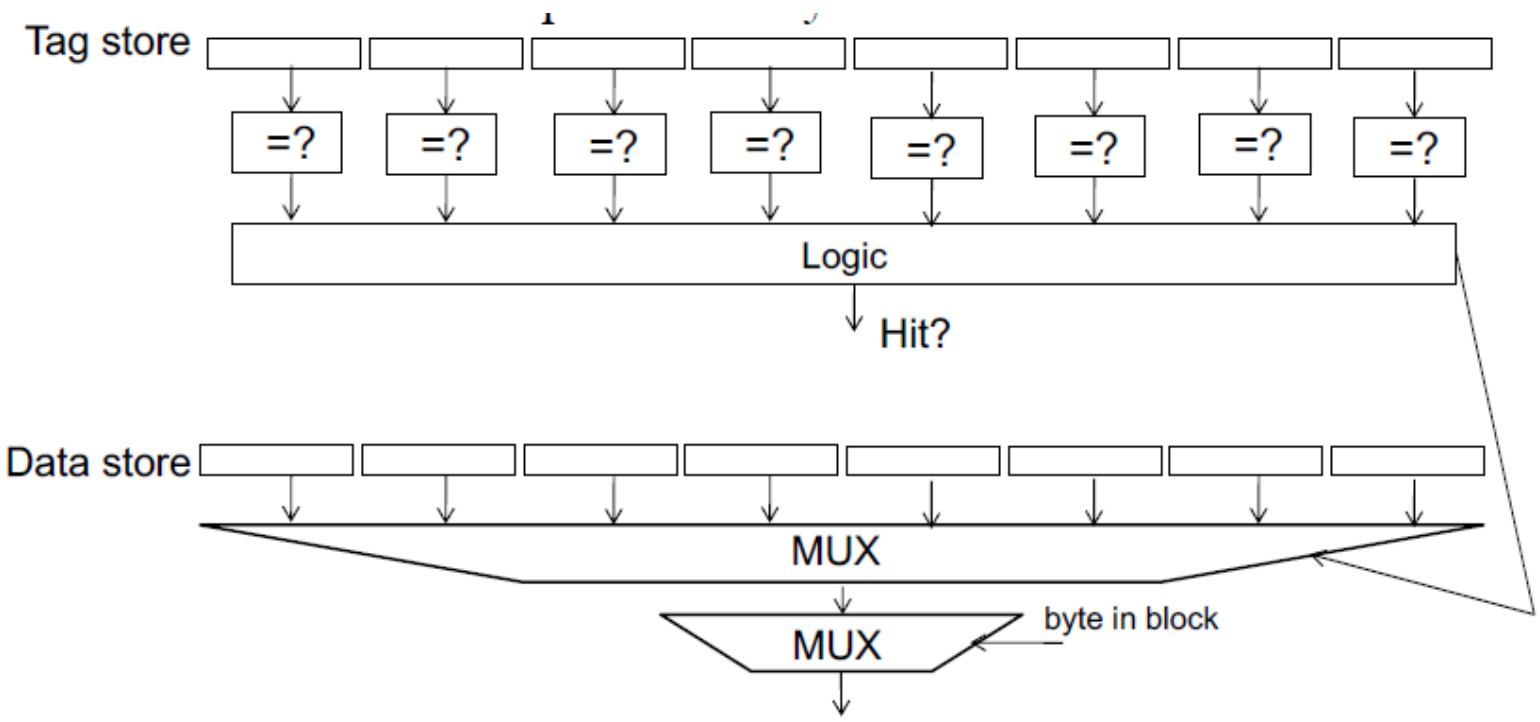


Figure 8 - Fully Associative Cache (*Yoo, 13-memory_intro*)

The third structure is a fully associative cache. A fully associative cache contains a single set with B ways, where B is the number of the data block. A memory address can map to a block in any of these ways. A fully associative cache is another name for a B-way set-associative cache with one set (*Harris & Harris, 2012*).

Figure 8 shows the fully associative cache with 8 blocks. Upon a data request, eight tag comparisons must be made because the data could be in any block in the data store. As same as the following operation, the 8-to-1 multiplexer chooses the proper data if a cache hit occurs.

A fully associative cache tends to have the fewest conflict misses for a given cache capacity. However, due to the more additional tag comparisons than the N-set-associative cache, they are best suited to relatively small caches. Without the following additional hardware cost and tradeoffs, a fully associative cache is the most ideal cache.

3-5. Replacement Policy

The cache memory is a resource that does not need to be explicitly managed by the user, it is automatically managed. However, the cache itself is managed by a set of cache replacement policies, which is also known as cache algorithm, that determine which data is stored in the cache during the program execution (*Schmidt et al., 2018*). As mentioned in the previous sections, due to the concept of the memory hierarchy, the dataset that the program is currently working on can easily exceed the cache capacity for many applications. To resolve this situation, a cache algorithm is required to address questions of which data should be loaded from the memory and where to store them, and if the cache is already full, which data should be evicted.

The caching algorithm aims at optimizing the hit ratio, which is (cache hit) / (cache hit + cache miss). In the case of a direct-mapped cache, it does not need the cache algorithm because it just evicts the cache line if the cache misses. However, in the case of set-associative and fully associative cache, they need the cache algorithm to choose which cache line to be changed. Various cache algorithms will be presented in later sections.

3-5-1. Random

Radom replacement randomly selects the next cache line in a set to replace. The selection algorithm uses a nonsequential incrementing victim counter. In a random replacement algorithm, the controller increments the victim counter by randomly selecting an increment value and adding this value to the victim counter. When the victim counter reaches a maximum value, it resets the value into the base value (*Sloss et al., 2009*).

3-5-2. Simple Queued Based Policies (FIFO)

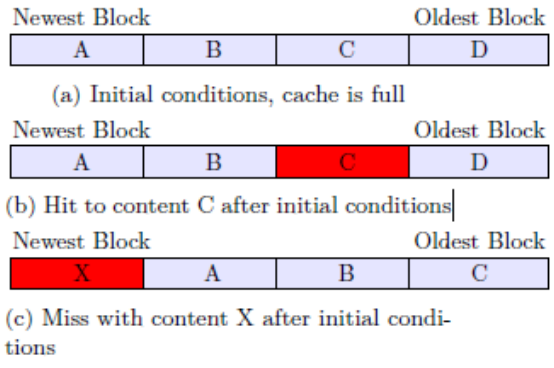


Figure 9 - FIFO Algorithm (*Logo: To the web site of Uppsala University*)

FIFO is the abbreviation of First-In, First-Out. After all the blocks are occupied, in the case of a new block must be inserted. This policy takes the first block inserted and replaces it with the new one. If the cache hits, the cache memory does not change its state. For this reason of that after taking the oldest referenced cache from the end of the array and storing it in the front of the array, we have to use a circular queue, which is also known as deque, to implement the following algorithm. Figure 9 shows the example of operation for cache replacement algorithm that uses 4 block FIFO algorithm.

3-5-3. Least Recently Used (LRU)

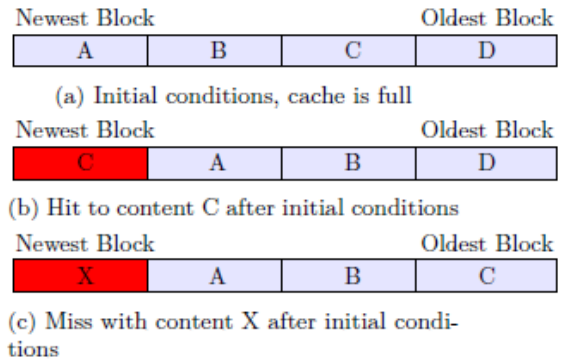


Figure 10 - LRU Algorithm (*Logo: To the web site of Uppsala University*)

LRU stands for Least Recently Used and the idea of this policy is to replace the blocks which are used less in the cache memory. After the initial starting point, where all the blocks are occupied, where a new block must be accessed and it is not presented in the cache memory, PRU will put this new data block in the position of the oldest block without a cache hit. In the case of a cache hit, the hit block is moved to the position of the newest data block.

The main difference between FIFO and LRU algorithms is the operation after a cache hit. FIFO does not change the order of the data blocks in the cache. However, LRU puts the hit block into the newest position. Figure 10 shows the operation of the LRU algorithm that uses 4 data blocks.

3-5-4. Least Frequently Used (LFU)

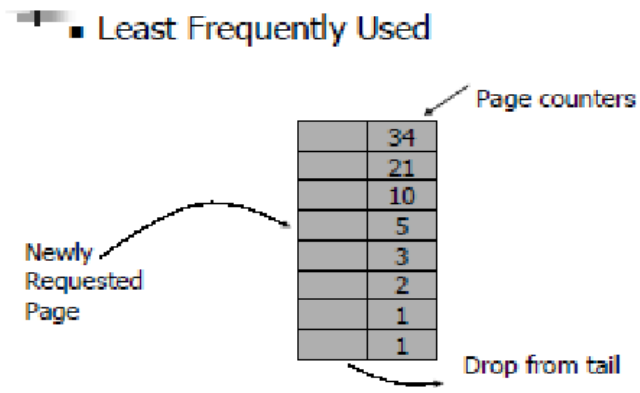
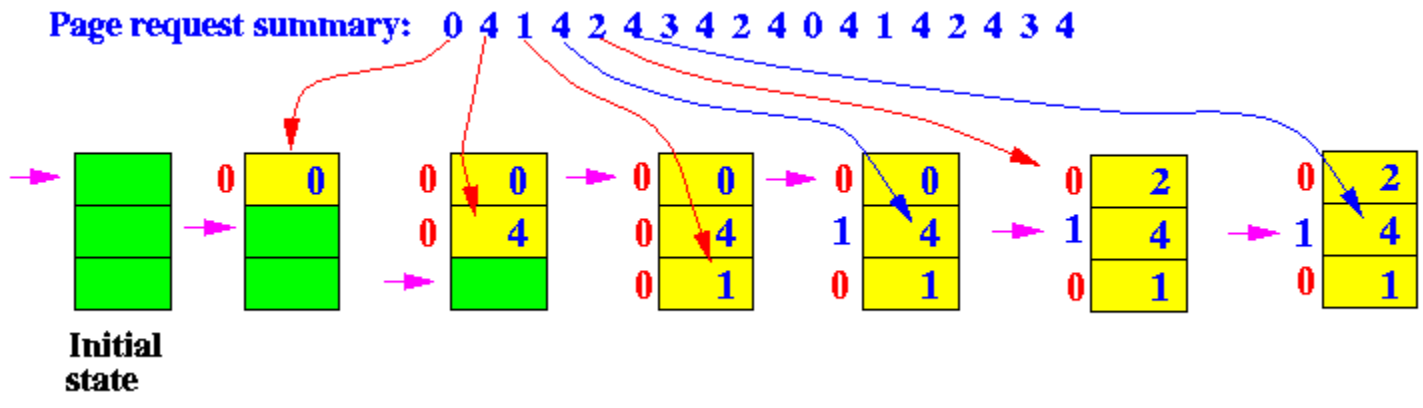


Figure 11 - LFU Algorithm

LFU is the abbreviation of Least Frequently Used. It is the cache algorithm where less frequently used data block are evicted first among the other data blocks. This algorithm can be used for scenarios that depend on few cache objects that will be used repeatedly for every request.

To implement the following cache algorithm, we need to add the counter that stores the number of references to the data block. With the following counter, LFU algorithm works as same as LRU algorithm, by having only difference of the benchmark that uses to select the cache blocks that should be evicted or updated.

3-5-5. Second Chance Algorithm (SCA)



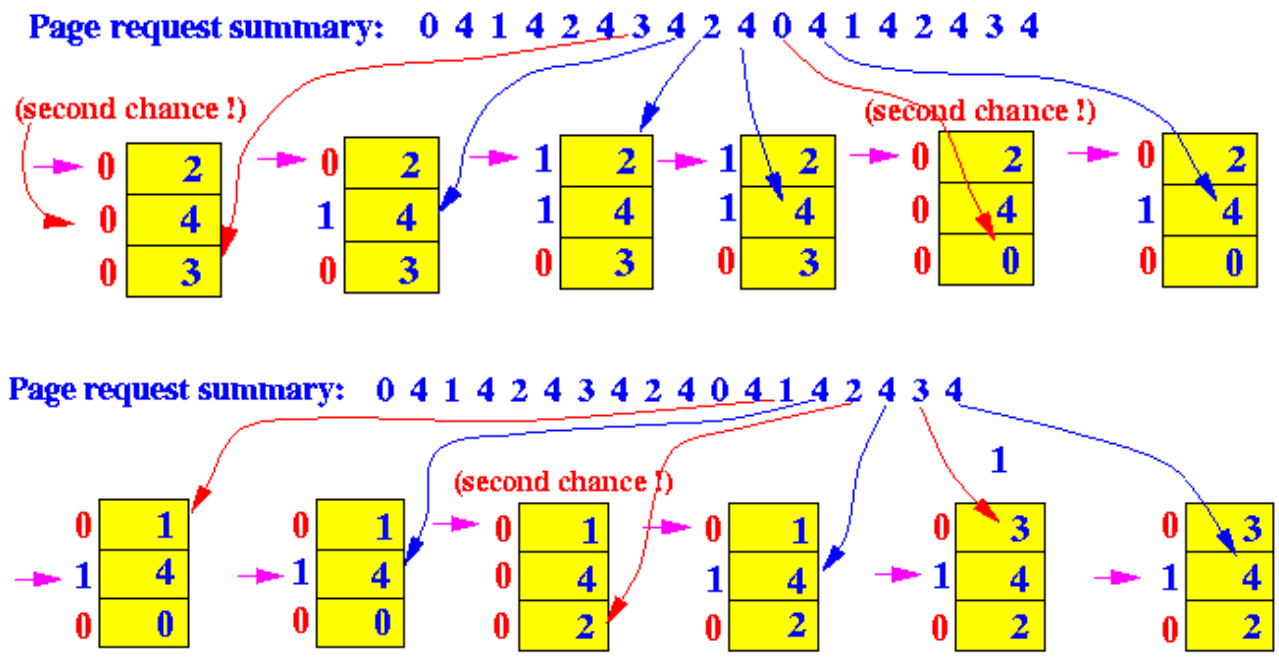


Figure 12 - Second Chance Algorithm (*The second chance page replacement policy*)

Implementing the LRU algorithm for cache is hard due to practical reasons such as additional const on the counter that records the timestamp of each cache line and linear search for picking up the oldest cache line etc. Therefore, to implement the cache algorithm similarly to the LRU algorithm without using the counter, we can replace it with the second chance algorithm.

In the second chance cache replacement policy, the candidate data blocks for removal are considered in a round-robin matter, and a data block that has been accessed between the consecutive considerations will not be replaced (*The second chance page replacement policy*). The data block which is replaced is the one considered in a round-robin matter with the benchmark of second chance bit.

To implement the following algorithm, we must add the additional bit, which is the second chance bit, into each cache line. At each time the cache hit happens, it will set the second chance bit into 1, which will give the data block a second chance. If a new data block is read from the memory into the cache the second chance bit of the following cache line is 0. When it needs the cache line removal, it should look in a round-robin manner in the cache lines. If the second chance bit is 1, reset its second chance bit set to 0. If the second chance bit is 0, replace the page in that memory frame. Figure 12 shows the following operation of the second chance algorithm.

3-5-6. Enhanced Second Chance Algorithm (ESCA)

Enhanced Second Chance Bit	State
00	Neither recently used not modified – best block to replace.
01	Not recently used but modified – not as good we need to swap out a block, but still better than used block.
10	Recently used but unmodified.
11	Recently used and modified – the worst page to replace.

Figure 13 - Enhanced Second Chance Algorithm

The enhanced second chance algorithm is an algorithm that uses the reference bit and modifies a bit to find the victim block. It operates in the same way as the second chance algorithm but uses different pattern bits. Unlike the second chance algorithm, the enhanced second chance algorithm uses 2 bits, which consist of reference bit and modification a bit. Figure 13 shows the state of the following data block on each bit of the pattern bits. According to the table in Figure 13, the data block is only evicted when the pattern bit is 00. Except 00, every bit will be decreased in the value of 1 if the data block with that pattern is the one that should be evicted. The advantage of the following algorithm is that this can search for the victim with high accuracy. The disadvantage of this algorithm is that the problem in search can occur due to the repeatable search on the circular queue.

3-6. Write Policy

In this section, we will mainly discuss what happens when the CPU writes data to a register and then call a store instruction. If the following data is already in the cache, there will three possible operations to execute. The first is no-write. This operation writes invalidates the cache and goes directly to the memory. The second is write-through. In write-through operation, it writes on the memory and the cache. The third is write-back. In this operation, the CPU writes the data only in the cache and the cache writes the data block into the main memory later, when the following data block is evicted. In the later sections we will mainly discuss the cache write policies of write-through and write-back policies.

3-6-1. Write Through

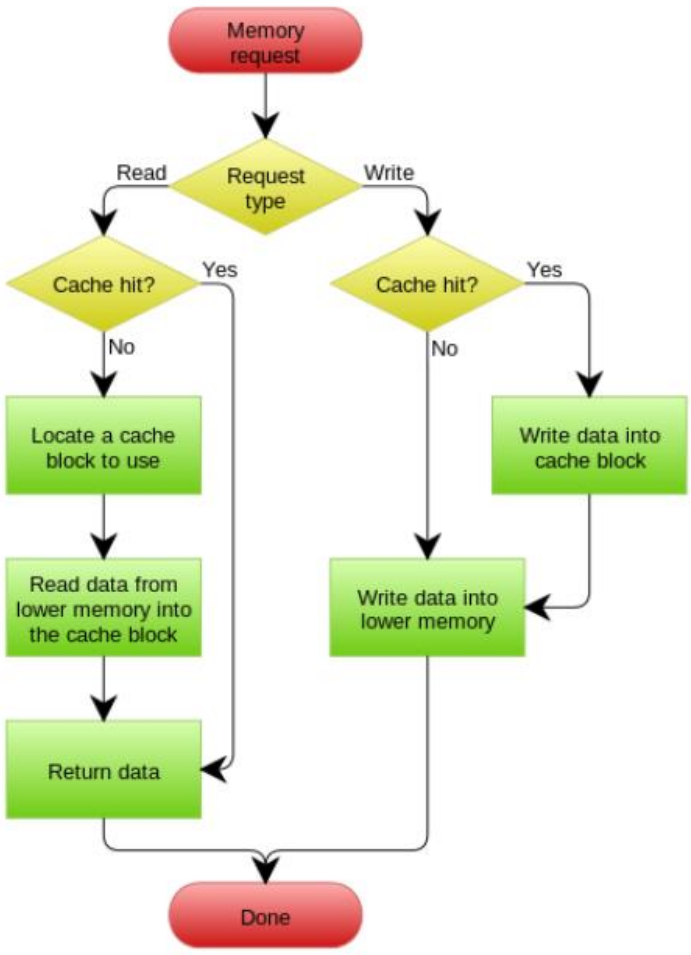


Figure 14 - Flow Chart of Write Through Policy

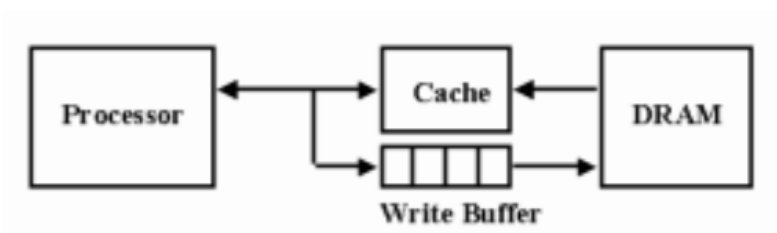


Figure 15 - Write Buffer in Write Through Policy (*Jose Nelson Amaral homepage*)

Write through policy is the operation that stores the data into the memory when some value is written in the cache. This means, that if the data write operation is executed, the following data is written in both cache and the memory. Figure 14 shows the sequence of write-through policies as a flow chart.

The advantage of the following policy is that it is stable because following policy can maintain the cache consistency. The disadvantage of writing through policy is slow throughput. Due to the slower accessing and processing time in the memory than in the cache, this operation increases the stalling time of the CPU, to wait until the memory operation is done.

To resolve the following disadvantage, we use a write buffer with the write-through policy. Write buffer decrease the waiting time by avoiding the direct write operation command from the CPU. Write buffer can be also used for write-back policy. Figure 15 shows the sequence of how the write buffer operates in the write-through policy.

3-6-2. Write Back

Figure 16 - Flow Chart of Write Back Policy

When a cache controller uses a write-back policy, it writes to valid cache data memory and not to the memory. Consequently, valid cache lines and the memory may contain different data. The cache line holds the most recent data, and the memory contains older data, which has not been updated.

Caches configured as write-back caches must use one or more dirty bits in the cache line status tag block. When a cache controller in writeback writes a value to the cache, it sets the dirty bit as true. If the cores accessed the cache line at a later time, it knows by the state of the dirty bit that the cache line contains the data which is not in the memory. If the cache controller evicts a dirty cache line, it is automatically written out to the memory. The controller does this operation to prevent the loss of critical information held in the cache and not in the memory. The sequence of the following policy operation is presented in Figure 16 as a flow chart.

One performance advantage in cache with the write-back policy is that it has over a cache with write-through cache in the frequent use of temporary local variables by a subroutine. These variables are transient in nature and never really need to be written to the memory. An example of one of these transient variables is a local variable that overflows onto a cached stack because there are not enough registers in the register file to hold the variable.

4. Pipelined Microarchitecture with Cache

In this section, we will mainly discuss the ideas and methods to implement our cache MIPS simulator. With pipelined microarchitecture and cache, the cache needs to be tightly integrated into the pipeline. In the ideal case, the access operation must happen in a single cycle so that dependent operations will not stall. However, to satisfy the high frequency of the pipeline, there is the limitation of increasing the size of the cache. Therefore, the cache should be implemented by considering the following tradeoffs. Our implementation of the cache MIPS simulator includes these concepts, and it is implemented by following concepts and program definitions.

4-1. Implemented Cache

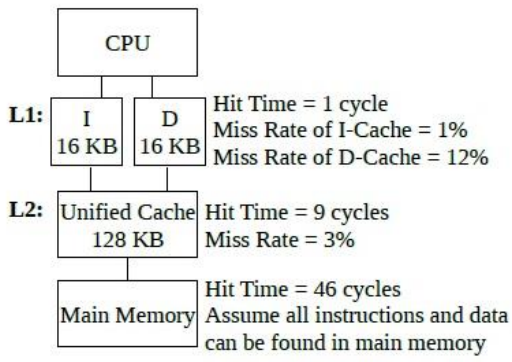


Figure 17 - Multiple Caches (*Project2, CHANGYOON*)

The cache that we have implemented is the L1 cache. L1 cache is the closest cache from the processor. Due to the processing speed of each cycle in pipelined microarchitecture, the L1 cache is separated into an instruction cache ($I) and a data cache ($D). Figure 17 shows the cache hierarchy of each level of cache. Instruction cache deals with the memory text area and data cache deals with all data without text area (*Project2, CHANGYOON*).

Modern processors these days have a cache with L1, L2, and L3, which is presented in Figure 2. As mentioned before, the L1 cache operates with the processor, the L2 cache operates between the L1 and L3 cache, and the L3 cache operates with the main memory. In the ideal case, we must implement the whole cache to check if they are operating in the right way. However, in this project, because the size of the execution of test programs is not that large, we do not need to separate that many caches to check the operations of the cache. As a result, we only implemented the L1 cache and main memory.

4-2. Program Definition

Before implementing the cache MISP simulator in the physical schema, we will state the program definitions that will be used in the real implementation.

- Global Variables

Variables	Data Type	Definition
CACHE	char	Cache Index: 1. FAC, 2. DMC, 3. SAC
WRITEBACK	int	Write Policy Index: 0. Write Through, 1. Write Back
REPLACEMENT	char	Replacement Policy Index: 0. FIFO, 1. Radom, 2. LRU, 3. LFU, 4. SCA, 5.ESCA
MAXWAY	determined	Number of the ways
MAXSIZE	determined	Size of the data block in the data store
MAXLINE	determined	Number of the cache line
FAC_Deque	int	Deque (circular queue) that is used in the FIFO based cache algorithms
FAC_L1_Data	FAC_Line*	Fully associative L1 data cache.
FAC_L1_Inst	FAC_Line*	Fully associative L1 instruction cache.
DMC_L1_Data	DMC_Line*	Direct mapped L1 data cache
DMC_L1_Inst	DMC_Line*	Directed mapped L1 instruction cache
SAC_Deque	Int	Deque (circular queue) that is used in the FIFO based cache algorithms
SAC_L1_Data	SAC_Line**	Set associative L1 data cache
SAC_L1_Inst	SACLine**	Set associative L1 instruction cache
cache_hit	int	Counter of cache hit operation
cache_miss	int	Counter of cache miss operation
cache_total	int	Counter of total cache operation

- Modules and Functions

Modules	Functions	Definition
DEQ	Deque_Cycle	Pop the index at the end of the deque
FAC	FAC_Init	Initialize the fully associative cache
	FAC_L1_Inst_Operation	Instruction fully associative cache operation
	FAC_L1_Data_Operation	Fully associative data cache operation
	FAC_Read	Read data from the fully associative data cache
	FAC_Write	Write the data into the fully associative data cache
	FAC_FIFO	First in first out algorithm
	FAC_Random	Random algorithm
	FAC_LRU	Least frequently used algorithm
	FAC_LFU	Least frequently used algorithm
	FAC_SCA	Second chance algorithm
	FAC_ESCA	Enhanced second chance algorithm
DMC	DMC_Init	Initialize the direct mapped cache
	DMC_L1_Inst_Operation	Instruction direct mapped cache operation
	DMC_L1_Data_Operation	Direct mapped data cache operation
	DMC_Read	Read data from the direct mapped data cache
	DMC_Write	Write the data into the direct mapped data cache
SAC	SAC_Init	Initialize the set associative cache
	SAC_L1_Inst_Operation	Instruction set associative cache operation
	SAC_L1_Data_Operation	Set associative data cache operation
	SAC_Read	Read data from the set associative data cache
	SAC_Write	Write the data into the set associative data cache
	SAC_FIFO	First in first out algorithm
	SAC_Random	Random algorithm
	SAC_LRU	Least frequently used algorithm
	SAC_LFU	Least frequently used algorithm
	SAC_SCA	Second chance algorithm
	SAC_ESCA	Enhanced second chance algorithm

5. Implementation

5-1. Memory (MEM)

```
57  /* copy the memory from the main memory to the cache */
58  void Memory_to_Cache(uint32_t* Cache, uint32_t Address) {
59      Address >>= 6;
60      Address <<= 6;
61      uint32_t result = 0;
62      for (int i = 0; i < MAXSIZE; i++) {
63          for (int j = 0; j < (32 / 8); j++) {
64              result <<= 8;
65              result |= Memory[Address + (i * 4) + j];
66          }
67          Cache[i] = result;
68      }
69  }
```

Figure 18 - Memory to Cache

```
71  /* copy the memory from the cache to the main memory */
72  void Cache_to_Memory(uint32_t* Cache, uint32_t Address) {
73      Address >>= 6;
74      Address <<= 6;
75      memory_access++;
76      for (int i = 0; i < MAXSIZE; i++) {
77          uint32_t WriteData = Cache[i];
78          for (int j = (32 / 8) - 1; j >= 0; j--) {
79              Memory[Address + (i * 4) + j] = WriteData & 0xff;
80              WriteData = WriteData >> 8;
81          }
82      }
83  }
```

Figure 19 - Cache to Memory

The figures above show the functions that are used in the operation between the cache and the memory. Memory_to_Cache() function writes the data block from the memory into the cache by using the first 26 bits of the address. Cache_to_Memory() function writes the data block from the cache into the memory by using the first 26 bits of the address. These functions are used to process the operation between the cache and the memory.

5-2. Direct Mapped Cache (DMC)

```
43      if (Cache[index].tag == tag) {
44          cache_hit++;
45          data = Cache[index].data[offset >> 2];
46      }
47
48      else {
49          cache_miss++;
50          uint32_t address = (Cache[index].tag << 10) + (index << 6);
51
52          Cache[index].tag = tag;
53          Cache[index].valid = true;
54
55          if (WRITEBACK) {
56              if (Cache[index].dirty) Cache_to_Memory(Cache[index].data, address);
57              Cache[index].dirty = false;
58          }
59
60          Memory_to_Cache(Cache[index].data, Address);
61          data = Cache[index].data[offset >> 2];
62      }
63      return data;
64  }
```

Figure 20 - DMC Read

```
75      if (Cache[index].tag == tag) {
76          cache_hit++;
77          Cache[index].dirty = true;
78          Cache[index].data[offset >> 2] = value;
79
80          if (WRITEBACK) {
81              Cache[index].dirty = true;
82          }
83          else {
84              Memory_Access(Address, size: 32, WriteData: value, MemRead: 0, MemWrite: 1);
85          }
86      }
87
88      else {
89          cache_miss++;
90          uint32_t address = (Cache[index].tag << 10) + (index << 6);
91
92          Cache[index].tag = tag;
93          Cache[index].valid = true;
94
95          if (WRITEBACK) {
96              if (Cache[index].dirty) Cache_to_Memory(Cache[index].data, address);
97              Memory_to_Cache(Cache[index].data, Address);
98              Cache[index].data[offset >> 2] = value;
99              Cache[index].dirty = true;
100         }
101         else {
102             Memory_Access(Address, size: 32, WriteData: value, MemRead: 0, MemWrite: 1);
103             Memory_to_Cache(Cache[index].data, Address);
104         }
105     }
106 }
```

Figure 21 - DMC Write

Based on the concept that presents the operation of the directed mapped cache in sections 3-4-1, we implemented the direct-mapped cache functions that is presented in Figures 20 and 21. Figures 20 and 21 show the function that operates the read and write operation of the direct-mapped cache. They decode the address into three parts, which are tag, index, and offset, and check if the cache is hit or miss. After determining whether the cache is hit or missing, it operates in the same process which is explained in section 3-4-1. By implementing the following code and executing them for data reference, we can simulate the cache MIPS simulator with the direct-mapped cache.

5-3. Set Associative Cache (SAC)

```
56      if (Cache[index][way].tag == tag) {
57          cache_hit += 1;
58          isUpdated = true;
59          Cache[index][way].frequency += 1;
60          Cache[index][way].timestamp = cycle;
61          data = Cache[index][way].data[offset >> 2];
62          break;
63      }
64  }
65
66  if (!isUpdated && isEmpty) {
67      cache_miss += 1;
68      isUpdated = true;
69      Cache[index][EmptyIdx].tag = tag;
70      Cache[index][EmptyIdx].valid = true;
71      Cache[index][EmptyIdx].frequency = 0;
72      Cache[index][EmptyIdx].timestamp = cycle;
73      Memory_to_Cache(Cache[index][EmptyIdx].data, Address);
74      data = Cache[index][EmptyIdx].data[offset >> 2];
75  }
76
77  if (!isUpdated && !isEmpty) {
78      int way = 0;
79      cache_miss += 1;
80      switch (REPLACEMENT)
81      {
82      case '0': way = SAC_FIFO(index); break;
83      case '1': way = SAC_Random(); break;
84      case '2': way = SAC_LRU(Cache, index); break;
85      case '3': way = SAC_LFU(Cache, index); break;
86      case '4': way = SAC_SCA(Cache, index); break;
87      case '5': way = SAC_ESCA(Cache, index); break;
88      default:
89          printf( _Format: "Error: wrong replacement policy selection.\n");
90          exit( _Code: 0);
91      }
92      uint32_t address = (Cache[index][way].tag << 8) + (index << 6);
93
94      Cache[index][way].tag = tag;
95      Cache[index][way].valid = true;
96      Cache[index][way].frequency = 0;
97      Cache[index][way].timestamp = cycle;
98
99      if (WRITEBACK) {
100         if (Cache[index][way].dirty) Cache_to_Memory(Cache[index][way].data, address);
101         Cache[index][way].dirty = false;
102     }
103     Memory_to_Cache(Cache[index][way].data, Address);
104     data = Cache[index][way].data[offset >> 2];
105 }
106 return data;
107 }
```

Figure 22 - SAC Read

```
119 for (int way = 0; way < MAXWAY; way++) {
120     if (!Cache[index][way].valid && !isEmpty) {
121         EmptyIdx = way;
122         isEmpty = true;
123         continue;
124     }
125
126     if (Cache[index][way].tag == tag) {
127         cache_hit += 1;
128         isUpdated = true;
129         Cache[index][way].frequency += 1;
130         Cache[index][way].timestamp = cycle;
131         Cache[index][way].data[offset >> 2] = value;
132
133         if (WRITEBACK) {
134             Cache[index][way].dirty = true;
135         }
136         else {
137             Memory_Access(Address, size: 32, WriteData: value, MemRead: 0, MemWrite: 1);
138         }
139         break;
140     }
141 }
142
143 if (!isUpdated && isEmpty) {
144     cache_miss += 1;
145     isUpdated = true;
146     Cache[index][EmptyIdx].tag = tag;
147     Cache[index][EmptyIdx].valid = true;
148     Cache[index][EmptyIdx].frequency = 0;
149     Cache[index][EmptyIdx].timestamp = cycle;
150
151     if (WRITEBACK) {
152         Memory_to_Cache(Cache[index][EmptyIdx].data, Address);
153         Cache[index][EmptyIdx].data[offset >> 2] = value;
154         Cache[index][EmptyIdx].dirty = true;
155     }
156     else {
157         Memory_Access(Address, size: 32, WriteData: value, MemRead: 0, MemWrite: 1);
158         Memory_to_Cache(Cache[index][EmptyIdx].data, Address);
159     }
160 }
161
162 if (!isUpdated && !isEmpty) {
163     int way = 0;
164     cache_miss += 1;
165     switch (REPLACEMENT)
166     {
167     case '0': way = SAC_FIFO(index); break;
168     case '1': way = SAC_Random(); break;
169     case '2': way = SAC_LRU(Cache, index); break;
170     case '3': way = SAC_LFU(Cache, index); break;
171     case '4': way = SAC_SCA(Cache, index); break;
172     case '5': way = SAC_ESCA(Cache, index); break;
173     default:
174         printf( _Format: "Error: wrong replacement policy selection.\n");
175         exit( _Code: 0);
176     }
177     uint32_t address = (Cache[index][way].tag << 8) + (index << 6);
178
179     Cache[index][way].tag = tag;
180     Cache[index][way].valid = true;
181     Cache[index][way].frequency = 0;
182     Cache[index][way].timestamp = cycle;
183
184     if (WRITEBACK) {
185         if (Cache[index][way].dirty) Cache_to_Memory(Cache[index][way].data, address);
186         Memory_to_Cache(Cache[index][way].data, Address);
187         Cache[index][way].data[offset >> 2] = value;
188         Cache[index][way].dirty = true;
189     }
190     else {
191         Memory_Access(Address, size: 32, WriteData: value, MemRead: 0, MemWrite: 1);
192         Memory_to_Cache(Cache[index][way].data, Address);
193     }
194 }
```

Figure 23 - SAC Write

According to the concept that presents the operation of the set associative cache in section 3-4-2, we implemented the set associative cache functions that are presented in Figure 22 and 23. Figure 22 and 23 shows the function that operates the read and write operation of set associative cache. They decode the address into three parts, which are tag, index, and offset, and check if the cache hit or miss. After determining the cache hit or miss, it operates in the same process which is explained in section 3-4-2. By implementing the following code and executing them for data reference, we can simulate the cache MIPS simulator with the set associative cache.

5-4. Fully Associative Cache (FAC)

```
for (int i = 0; i < MAXLINE * MAXWAY; i++) {
    if (!Cache[i].valid) {
        cache_miss += 1;
        isUpdated = true;
        Cache[i].tag = tag;
        Cache[i].valid = true;
        Cache[i].frequency = 0;
        Cache[i].timestamp = true;
        Memory_to_Cache(Cache[i].data, Address);
        data = Cache[i].data[offset >> 2];
        break;
    }

    if (Cache[i].tag == tag) {
        cache_hit += 1;
        isUpdated = true;
        Cache[i].frequency += 1;
        Cache[i].timestamp = cycle;
        data = Cache[i].data[offset >> 2];
        break;
    }
}

if (!isUpdated) {
    uint32_t Index = 0;
    cache_miss += 1;
    switch (REPLACEMENT)
    {
    case '0': Index = FAC_FIFO();  break;
    case '1': Index = FAC_Random(); break;
    case '2': Index = FAC_LRU(Cache); break;
    case '3': Index = FAC_LFU(Cache); break;
    case '4': Index = FAC_SCA(Cache); break;
    case '5': Index = FAC_ESCA(Cache); break;
    default:
        printf(_Format: "Error: wrong replacement policy selection.\n");
        exit(_Code: 0);
    }
    uint32_t address = Cache[Index].tag << 6;

    Cache[Index].tag = tag;
    Cache[Index].valid = true;
    Cache[Index].frequency = 0;
    Cache[Index].timestamp = cycle;

    if (WRITEBACK) {
        if (Cache[Index].dirty) Cache_to_Memory(Cache[Index].data, address);
        Cache[Index].dirty = false;
    }
    Memory_to_Cache(Cache[Index].data, Address);
    data = Cache[Index].data[offset >> 2];
}
return data;
}
```

Figure 24 - FAC Read

```
for (int i = 0; i < MAXLINE * MAXWAY; i++) {
    if (!Cache[i].valid) {
        cache_miss += 1;
        isUpdated = true;
        Cache[i].tag = tag;
        Cache[i].valid = true;
        Cache[i].frequency = 0;
        Cache[i].timestamp = cycle;

        if (WRITEBACK) {
            Memory_to_Cache(Cache[i].data, Address);
            Cache[i].data[offset >> 2] = value;
            Cache[i].dirty = true;
        }
        else {
            Memory_Access(Address, size: 32, WriteData: value, MemRead: 0, MemWrite: 1);
            Memory_to_Cache(Cache[i].data, Address);
        }

        break;
    }

    if (Cache[i].tag == tag) {
        cache_hit += 1;
        isUpdated = true;
        Cache[i].frequency += 1;
        Cache[i].timestamp = cycle;
        Cache[i].data[offset >> 2] = value;

        if (WRITEBACK) {
            Cache[i].dirty = true;
        }
        else {
            Memory_Access(Address, size: 32, WriteData: value, MemRead: 0, MemWrite: 1);
        }
        break;
    }
}

if (!isUpdated) {
    uint32_t Index = 0;
    cache_miss += 1;
    switch (REPLACEMENT)
    {
    case '0': Index = FAC_FIFO();  break;
    case '1': Index = FAC_Random(); break;
    case '2': Index = FAC_LRU(Cache); break;
    case '3': Index = FAC_LFU(Cache); break;
    case '4': Index = FAC_SCA(Cache); break;
    case '5': Index = FAC_ESCA(Cache); break;
    default:
        printf(_Format: "Error: wrong replacement policy selection.\n");
        exit(_Code: 0);
    }
    uint32_t address = Cache[Index].tag << 6;

    Cache[Index].tag = tag;
    Cache[Index].valid = true;
    Cache[Index].frequency = 0;
    Cache[Index].timestamp = cycle;

    if (WRITEBACK) {
        if (Cache[Index].dirty) Cache_to_Memory(Cache[Index].data, address);
        Memory_to_Cache(Cache[Index].data, Address);
        Cache[Index].data[offset >> 2] = value;
        Cache[Index].dirty = true;
    }
    else {
        Memory_Access(Address, size: 32, WriteData: value, MemRead: 0, MemWrite: 1);
        Memory_to_Cache(Cache[Index].data, Address);
    }
}
}
```

Figure 25 - FAC Write

Based on the concept that presents the operation of the fully associative cache in sections 3-4-3, we implemented the fully associative cache functions that are presented in Figures 24 and 25. Figures 24 and 25 show the function that operates the read and write operation of a fully associative cache. Unlike the previous implementations of the caches, the following functions decode the address into two parts, which are tag and offset. The concept of a fully associative cache checks the tag by using many comparators to check whether the cache is hit or miss. However, because we are using C language that executes in the linear sequence, we implemented the process of checking the cache hit or miss by linear searching. Also, to get the ideal results that are the same as the concept, we counted only one cache access operation for the following cache operation. After determining the cache hit or miss, it operates in the same process which is explained in section 3-4-3. By implementing the following code and executing them for data reference, we can simulate the cache MIPS simulator with the fully associative cache.

5-5. Replacement Policies

In this section, we will mainly discuss the implementation of the cache replacement policies presented in section 3. The following algorithms are used in the full and set-associative cache for cache line replacement. The figures shown in the later sections are the functions that operate with the set-associative cache functions. The following cache replacement algorithms are also implemented for fully associative cache with different structures due to the different structures with the set-associative cache.

5-5.1. Random

```
197     /* random algorithm */
198     int SAC_Random() {
199         return rand() % MAXWAY;
200     }
```

Figure 26 - Radom Algorithm

Random algorithm literally means that it randomly generates the index of the cache line that should be evicted in the next sequence. To implement the following algorithms, we used the random function, which is already implemented in the basic library in the C language.

5-5.2. Simple Queue Based Policy (FIFO)

```
3     /* pop the index at the end of the deque */
4     void Deque_Cycle(int* Deque, int size) {
5         int temp = Deque[size - 1];
6         for (int i = size - 1; i > 0; i--) Deque[i] = Deque[i - 1];
7         Deque[0] = temp;
8     }
```

Figure 27 - Cycle Operation of Deque

```
236     /* first in first out algorithm */
237     int SAC_FIFO(uint16_t index) {
238         cache_total++;
239         int FIFOIdx = SAC_Deque[index][MAXWAY - 1];
240         Deque_Cycle(SAC_Deque[index], size: MAXWAY);
241         return FIFOIdx;
242     }
```

Figure 28 - FIFO Algorithm

The word FIFO stands for First-In, First-Out. To implement the data structure that follows the FIFO characteristics, we must use a queue. Also, in the cache operation, the popped element should be pushed to the front of the queue, which means it should be operated circularly. Therefore, we used deque, which is also known as a circular queue, to implement the basic FIFO algorithm.

In an ideal situation, we must implement the deque with the linked list and its operations with functions. However, in the cache operation, the only used deque operation is circular pop, which pops the element and pushes it into the front of the deque. Therefore, we implemented the deque in a single array, and built the following operations by shifting each array element to the next index. After shifting the elements, we swapped the value of index 0 to the data that we popped in the former operation.

Figure 28 shows the function of the FIFO algorithm that operates with the set-associative cache. SAC_FIFO() function uses the presented function, Deque_Cycle() to operate the FIFO algorithm. By implementing the following code and executing them for data reference, we can simulate the cache MIPS simulator with a FIFO algorithm.

5-5.3. Least Recently Used (LRU)

```
202     /* least recently used algorithm */
203     int SAC_LRU(SACLine Cache[][MAXWAY], uint16_t index) {
204         int current;
205         int LRUIdx = index;
206         int timestamp = Cache[index][0].timestamp;
207
208         for (int way = 0; way < MAXWAY; way++) {
209             cache_total++;
210             current = Cache[index][way].timestamp;
211             if (timestamp > current) {
212                 LRUIdx = way;
213                 timestamp = current;
214             }
215         }
216         return LRUIdx;
217     }
```

Figure 29 - LRU Algorithm

The word LRU stands for Least Recently Used. To implement the LRU algorithm, we need an additional counter that counts the cache referenced time, which is the timestamp. The data that the timestamp stores can vary. We have tried storing it in real-time in milliseconds and nanoseconds, but it showed some problems. If the conflicted cache reference happens in a shorter time than the millisecond or nanosecond, the non-ideal cache line can be evicted. To resolve the following problem, we decided to store the number of cycles in the timestamp.

Figure 29 shows the function of the LRU algorithm that operates with the set-associative cache. SAC_LRU() function searches the oldest referenced cache line using the timestamp, and after picking the right index, it returns it.

5-5.4. Least Frequently Used (LFU)

```
219     /* least frequently used algorithm */
220     int SAC_LFU(SACLine Cache[][MAXWAY], uint16_t index) {
221         int current;
222         int LFUIdx = index;
223         int frequency = Cache[index][0].frequency;
224
225         for (int way = 0; way < MAXWAY; way++) {
226             cache_total++;
227             current = Cache[index][way].frequency;
228             if (frequency > current) {
229                 LFUIdx = way;
230                 frequency = current;
231             }
232         }
233         return LFUIdx;
234     }
```

Figure 30 - LFU Algorithm

The word LFU stands for Least Frequently Used. To implement the LFU algorithm, we need an additional counter that counts the number of how many caches have referenced, which is the frequency. The data that the frequency stores is the count of the reference.

Figure 30 shows the function of the LFU algorithm that operates with the set-associative cache. SAC_LFU() function searches the least referenced cache line using the frequency, and after picking the right index, it returns it.

5-5.5. Second Chance Algorithm (SCA)

```
244     /* second chance algorithm */
245     int SAC_SCA(SACLine Cache[][MAXWAY], uint16_t index) {
246         int SCAIdx = SAC_Deque[index][MAXWAY - 1];
247         while (Cache[index][SCAIdx].sca) {
248             cache_total++;
249             Cache[index][SCAIdx].sca = 0;
250             Deque_Cycle(SAC_Deque[index], size: MAXWAY);
251             SCAIdx = SAC_Deque[index][MAXWAY - 1];
252         }
253         cache_total++;
254         Cache[index][SCAIdx].sca = 1;
255         Deque_Cycle(SAC_Deque[index], size: MAXWAY);
256         return SCAIdx;
257     }
```

Figure 31 - Second Chance Algorithm

The second chance algorithm is the extended version of the LRU algorithm that uses the concept of FIFO. While it searches for the next evicted index in the circular sequence, it checks whether the chosen cache line is the oldest or not by using the second chance bit which is stored in the tag store. After referencing the second chance bit, it operates in the process that is presented in section 3-5-5.

The implementation of the second chance bit also uses the deque for the circular search operation. Figure 31 shows the function of the second chance algorithm that operates with a set-associative cache. SAC_SCA() function finds out the properly evicted index due to the second chance algorithm and returns the following index. By implementing the following code and executing them for data reference, we can simulate the cache MIPS simulator with a second chance algorithm.

5-5.6. Enhanced Second Chance Algorithm (ESCA)

```
259     /* enhanced second chance algorithm */
260     int SAC_ESCA(SACLine Cache[][MAXWAY], uint16_t index) {
261         int ESCAIdx = SAC_Deque[index][MAXWAY - 1];
262         while (Cache[index][ESCAIdx].esca) {
263             cache_total++;
264             Cache[index][ESCAIdx].esca -= 1;
265             Deque_Cycle(SAC_Deque[index], size: MAXWAY);
266             ESCAIdx = SAC_Deque[index][MAXWAY - 1];
267         }
268         cache_total++;
269         Cache[index][ESCAIdx].esca = 0b11;
270         Deque_Cycle(SAC_Deque[index], size: MAXWAY);
271         return ESCAIdx;
272     }
```

Figure 32 - Enhanced Second Chance Algorithm

The enhanced second chance algorithm is the extended version of the second chance algorithm. While the second chance algorithm uses a single bit for the second chance bit, the enhanced second chance algorithm uses two bits as a second chance bit. The first bit of the second chance bit is reference bit, and other is modification bit. After referencing the second chance bit, it operates in the process that is presented in section 3-5-6.

The implementation of the enhanced second chance bit also uses the deque for the circular search operation. Figure 32 shows the function of the enhanced second chance algorithm that operates with a set-associative cache. SAC_ESCA() function finds out the properly evicted index due to the enhanced second chance algorithm and returns the following index. By implementing the following code and executing them for data reference, we can simulate the cache MIPS simulator with the enhanced second chance algorithm.

5-6. Write Policies

In this section, we will mainly discuss the implementation of the written policies presented in section 3. The written policies presented in section 3 are used for all types of caches. The figures shown in the later sections are the functions that operate with the set-associative cache functions. The following written policies are also implemented for a fully associative and direct-mapped cache.

5-5-1. Write Through

```
56      if (Cache[index][way].tag == tag) {
57          cache_hit += 1;
58          isUpdated = true;
59          Cache[index][way].frequency += 1;
60          Cache[index][way].timestamp = cycle;
61          data = Cache[index][way].data[offset >> 2];
62          break;
63      }
64  }
```

```
133         if (WRITEBACK) {
134             Cache[index][way].dirty = true;
135         }
136         else {
137             Memory_Access(Address, size: 32, WriteData: value, MemRead: 0, MemWrite: 1);
138         }
139         break;
```

Figure 33 - Read and Write Operation when Cache Hit

```
66      if (!isUpdated && isEmpty) {
67          cache_miss += 1;
68          isUpdated = true;
69          Cache[index][EmptyIdx].tag = tag;
70          Cache[index][EmptyIdx].valid = true;
71          Cache[index][EmptyIdx].frequency = 0;
72          Cache[index][EmptyIdx].timestamp = cycle;
73          Memory_to_Cache(Cache[index][EmptyIdx].data, Address);
74          data = Cache[index][EmptyIdx].data[offset >> 2];
75      }
```

```
151         if (WRITEBACK) {
152             Memory_to_Cache(Cache[index][EmptyIdx].data, Address);
153             Cache[index][EmptyIdx].data[offset >> 2] = value;
154             Cache[index][EmptyIdx].dirty = true;
155         }
156         else {
157             Memory_Access(Address, size: 32, WriteData: value, MemRead: 0, MemWrite: 1);
158             Memory_to_Cache(Cache[index][EmptyIdx].data, Address);
159         }
160     }
```

Figure 34 - Read and Write Operation when Cache Line is Empty

```
99      if (WRITEBACK) {
100       if (Cache[index][way].dirty) Cache_to_Memory(Cache[index][way].data, address);
101       Cache[index][way].dirty = false;
102     }
103     Memory_to_Cache(Cache[index][way].data, Address);
104     data = Cache[index][way].data[offset >> 2];
105   }
106   return data;
107 }
```

```
184     if (WRITEBACK) {
185       if (Cache[index][way].dirty) Cache_to_Memory(Cache[index][way].data, address);
186       Memory_to_Cache(Cache[index][way].data, Address);
187       Cache[index][way].data[offset >> 2] = value;
188       Cache[index][way].dirty = true;
189     }
190     else {
191       Memory_Access(Address, size: 32, WriteData: value, MemRead: 0, MemWrite: 1);
192       Memory_to_Cache(Cache[index][way].data, Address);
193     }
```

Figure 35 - Read and Write Operation when Cache Miss

Figures 33, 34, and 35 show the implemented codes for write-through policy. The codes with red rectangular are the implementations. For each read and writes operation, follows the sequence of the process presented in section 3-6-1. Due to the direct memory access operation for cache miss in the write-through policy, we used Memroy_Access() function that is already implemented in Project 3.

5-5-2. Write Back

```
56      if (Cache[index][way].tag == tag) {
57        cache_hit += 1;
58        isUpdated = true;
59        Cache[index][way].frequency += 1;
60        Cache[index][way].timestamp = cycle;
61        data = Cache[index][way].data[offset >> 2];
62        break;
63      }
64    }
```

```
133       if (WRITEBACK) {
134         Cache[index][way].dirty = true;
135       }
136       else {
137         Memory_Access(Address, size: 32, WriteData: value, MemRead: 0, MemWrite: 1);
138       }
139       break;
```

Figure 36 - Read and Write when Cache Hit

```
66    if (!isUpdated && isEmpty) {
67      cache_miss += 1;
68      isUpdated = true;
69      Cache[index][EmptyIdx].tag = tag;
70      Cache[index][EmptyIdx].valid = true;
71      Cache[index][EmptyIdx].frequency = 0;
72      Cache[index][EmptyIdx].timestamp = cycle;
73      Memory_to_Cache(Cache[index][EmptyIdx].data, Address);
74      data = Cache[index][EmptyIdx].data[offset >> 2];
75    }
```

```
151     if (WRITEBACK) {
152         Memory_to_Cache(Cache[index][EmptyIdx].data, Address);
153         Cache[index][EmptyIdx].data[offset >> 2] = value;
154         Cache[index][EmptyIdx].dirty = true;
155     }
156     else {
157         Memory_Access(Address, size: 32, WriteData: value, MemRead: 0, MemWrite: 1);
158         Memory_to_Cache(Cache[index][EmptyIdx].data, Address);
159     }
160 }
```

Figure 37 - Read and Write Operation when Cache Line is Empty

```
 99     if (WRITEBACK) {
100         if (Cache[index][way].dirty) Cache_to_Memory(Cache[index][way].data, address);
101         Cache[index][way].dirty = false;
102     }
103     Memory_to_Cache(Cache[index][way].data, Address);
104     data = Cache[index][way].data[offset >> 2];
105 }
106 return data;
107 }
```

```
184     if (WRITEBACK) {
185         if (Cache[index][way].dirty) Cache_to_Memory(Cache[index][way].data, address);
186         Memory_to_Cache(Cache[index][way].data, Address);
187         Cache[index][way].data[offset >> 2] = value;
188         Cache[index][way].dirty = true;
189     }
190     else {
191         Memory_Access(Address, size: 32, WriteData: value, MemRead: 0, MemWrite: 1);
192         Memory_to_Cache(Cache[index][way].data, Address);
193     }
```

Figure 38 - Read and Write Operation when Cache Miss

Figures 36, 37, and 38 show the implemented codes for the write-back policy. The codes with red rectangular are the implementations. For each read and writes operation, follows the sequence of the process presented in section 3-6-2. Due to the operation in the write-back policy, we added the dirty bit into each cache line. By using this dirty bit, the function can determine whether to write the data from the cache to memory or not. By implementing the following code and executing them for data reference, we can simulate the cache MIPS simulator with the write-back policy.

6. Build Environment

Following build environments are required to execute the cache MIPS simulator:

- Build Environments:
  1. Linux environment – Vi editor, GCC complier
  2. Program is built by using the Makefile.

- Make Command:
  1. $make cpu -> build the execution program
  2. $make clean -> clean all the object files that builds main

- Execution Command
  1. ./cpu test_prog/{$program} 1-x-x x -> Execute the program with Fully Associative Cache.
  2. ./cpu test_prog/{$program} 2-x-x x -> Execute the program with Direct Mapped Cache.
  3. ./cpu test_prog/{$program} 3-x-x x -> Execute the program with Set Associative Cache.
  4. ./cpu test_prog/{$program} x-0-x x -> Execute the program with Write Through Policy.
  5. ./cpu test_prog/{$program} x-1-x x -> Execute the program with Write Back Policy.
  6. ./cpu test_prog/{$program} x-x-0 x -> Execute the program with FIFO Algorithm.
  7. ./cpu test_prog/{$program} x-x-1 x -> Execute the program with Random Algorithm.
  8. ./cpu test_prog/{$program} x-x-2 x -> Execute the program with LRU Algorithm.
  9. ./cpu test_prog/{$program} x-x-3 x -> Execute the program with LFU Algorithm.
  10. ./cpu test_prog/{$program} x-x-4 x -> Execute the program with Second Chance Algorithm.
  11. ./cpu test_prog/{$program} x-x-5 x -> Execute the program with Enhanced Second Chance Algorithm.

7. Results

Program	Cache	Write	Replace	Memory	Cache Hit	Cache Miss	Cache Ratio	Cache Total
input4.bin	Fully Associative Cache	Write Through	FIFO	1026130	30423693	67548	99.778	30558661
			Random	1026130	30423532	67709	99.778	30491241
			LRU	1026130	30425532	65709	99.785	34688425
			LFU	1026130	29297572	1193669	96.085	106877865
			SCA	1026130	30423693	67548	99.778	30626017
			ESCA	1026130	30423693	67548	99.778	30760742
		Write Back	FIFO	1831	30423693	67548	99.778	30558661
			Random	1829	30423479	67762	99.778	30491241
			LRU	826	30425532	65709	99.785	34688425
			LFU	655	29297572	1193669	96.085	106877865
			SCA	1831	30423693	67548	99.778	30626017
			ESCA	1831	30423693	67548	99.778	30760742
	Direct Mapped Cache	Write Through	FIFO	1026130	29970022	521219	98.291	30491241
			Random	1026130	29970022	521219	98.291	30491241
			LRU	1026130	29970022	521219	98.291	30491241
			LFU	1026130	29970022	521219	98.291	30491241
			SCA	1026130	29970022	521219	98.291	30491241
			ESCA	1026130	29970022	521219	98.291	30491241
		Write Back	FIFO	172396	29970022	521219	98.291	30491241
			Random	172396	29970022	521219	98.291	30491241
			LRU	172396	29970022	521219	98.291	30491241
			LFU	172396	29970022	521219	98.291	30491241
			SCA	172396	29970022	521219	98.291	30491241
			ESCA	172396	29970022	521219	98.291	30491241
	Set Associative Cache	Write Through	FIFO	1026130	30417118	74123	99.757	30565332
			Random	1026130	30413845	77396	99.746	30491241
			LRU	1026130	30235526	255715	99.161	31513973
			LFU	1026130	29986986	504255	98.346	32508133
			SCA	1026130	30417118	74123	99.757	30639403
			ESCA	1026130	30417118	74123	99.757	30787539
		Write Back	FIFO	5137	30417118	74123	99.757	30565332
			Random	5427	30413817	77424	99.746	30491241
			LRU	770	30235526	255715	99.161	31513973
			LFU	747	29986986	504255	98.346	32508133
			SCA	5137	30417118	74123	99.757	30639403
			ESCA	5137	30417118	74123	99.757	30787539

Figure 39 - Result of Program Execution by Cache MIPS Simulator

Figure 39 shows the table for execution result of the input4.bin program. In the following format, each result for cache simulator with a different implementation of methods, write policies, replacement policies, and cache types, are stored as a CSV file in each folder. There are two CSV files in the result folder: result.csv and cache.csv files. result.csv file stores the whole result of the execution of programs, and the cache.csv file stores the information that is used as a performance indicator. To generate a CSV file that contains the results, we must follow the following process: First, if we execute the cache MIPS simulator, we can get the result.txt file. Second, open the Data_Purification.ipnyb and upload the result.txt on it, and then execute all the programs. As a result, we can get the following CSV files that contain the results of our cache MIPS simulator.

8. Evaluating the Pipelined Microarchitecture

8-1. Analysis

In short, we have built a cache MIPS simulator with the implementations above. In this section, we will analyze the cache performance in multiple ways, types of cache, writing policies, and cache replacement algorithms, by using the average memory access time (AMAT) formula.

8-2. Performance Analysis

- Cache hit rate = (# hits) / (# hits + # misses) = (# hits) / (# accesses)
- Average memory access time (AMAT)
  = (hit-rate * hit-latency) + (miss-rate * miss-latency)

Figure 40 - Average Memory Access Time (AMAT) (Yoo, 13-memory_intro)

Figure 40 shows the formula for average memory access time (AMAT) that is used in evaluating the performance of the cache. By using the following formula, we can compare the performances of different caches.

Let us denote the simple assumption that cache hit latency takes the time of T, which is the CPU's single clock time, and miss latency takes 1000 times than the cache hit latency, which is 1000 * T. According to the result table presented in section 7, the result for the execution of input4.bin program by using 4-way, set-associative cache with write-back policy, and enhanced second chance algorithm shows 5137 times of memory access, 30417118 times of cache hit, and 99.757% of the cache hit ratio. If there is no cache implementation, 7116606 times the memory access operation happens (Project 3, CHANGYOON). The AMAT for the execution of cache and non-cache operations is as below.

- Non-Cache Operation: (0 * t) + {(1 – 0) * 1000 * t} = 1000 * t
- Cache Operation: (0.99757 * t) + {(1 – 0.99757) * 1000 * t} = 3.42757 * t

As a result, the cache operation shows 333 times faster average memory accessing time than the non-cache operation, which means that the efficiency of the microarchitecture will be increased when we use the cache operation. Also, to enhance the performance of the microarchitecture with the cache, it is important to increase the cache-hit rate, so that we can decrease the average memory access time. The analysis for comparison of the results is based on this concept. In short, the performance comparison for later sections will be mainly compared by the cache-hit ratio.

8-2-1.Comparison on Ways

Program	4 Way	8 Way	16 Way
simple.bin	83.333	83.333	83.333
simple2.bin	87.5	87.5	87.2685
simple3.bin	99.847	99.847	99.16106
simple4.bin	97.512	97.512	97.64172
gcd.bin	98.58364	98.64467	98.57861
fib.bin	99.746	99.746	99.68794
input4.bin	98.95844	99.04089	99.14333

Figure 41 - Comparison Table for Multiple Ways

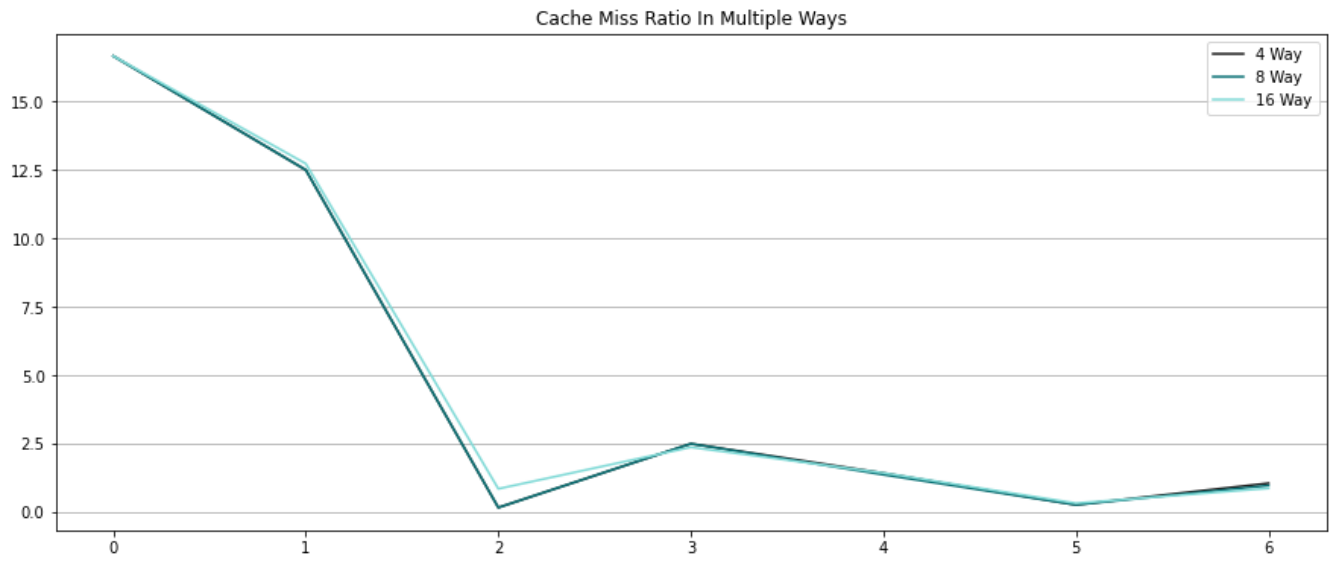


Figure 42 - Comparison Graph for Multiple Ways

Figures 41 and 42 are the table and graph results of the cache MIPS simulator in different set ways. As the number of ways is increased, the cache-miss ratio decreases. In short, due to the decrement of the cache-miss ratio, average memory access time (AMAT) decreases, and this led to the higher performance of the microarchitecture.

8-2-2.Comparison on Types of Cache

Program	Fully Associative	Direct Mapped	Set Associative
simple.bin	83.333	83.333	83.333
simple2.bin	87.5	87.5	87.5
simple3.bin	99.847	99.847	99.847
simple4.bin	97.512	97.512	97.512
gcd.bin	98.701	98.532	98.51791667
fib.bin	99.746	99.746	99.746
input4.bin	99.16366667	98.291	99.42066667

Figure 43 - Comparison Table for Multiple Types of Cache

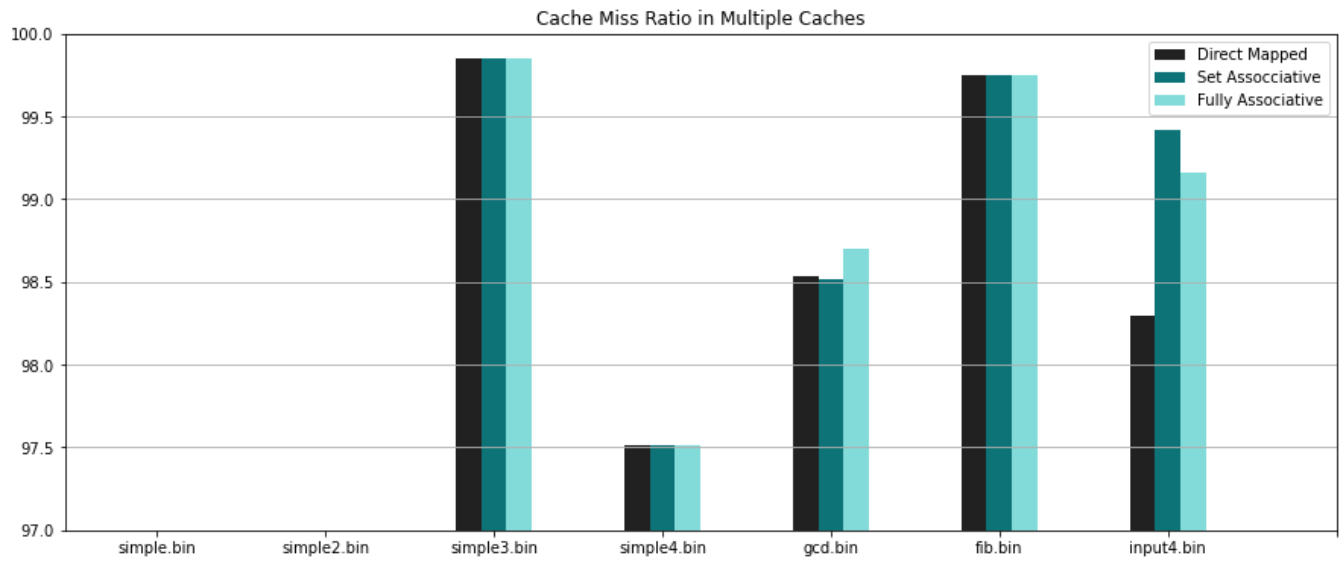


Figure 44 - Comparison Graph for Multiple Types of Cache

As we mentioned in section 3, the ideal performance of types of the cache will have the inequality of Fully Associative Cache > Set Associative Cache > Direct Mapped Cache

Now, let us check Figures 43 and 44, which are the table and graph for result comparison. As we can check from the figures, except for the input4.bin program, the fully associative cache shows the highest cache-hit ratio among all the programs. However, the input4.bin program set-associative cache shows the highest cache-hit ratio.

8-2-3.Comparison on Cache Write Policy

Program	Write Through	Write Back
simple.bin	1	0
simple2.bin	2	0
simple3.bin	206	34.77777778
simple4.bin	41	0
gcd.bin	153	4
fib.bin	494	0
input4.bin	1026130	59196.33333

Figure 45 - Comparison Table for Multiple Cache Write Policy

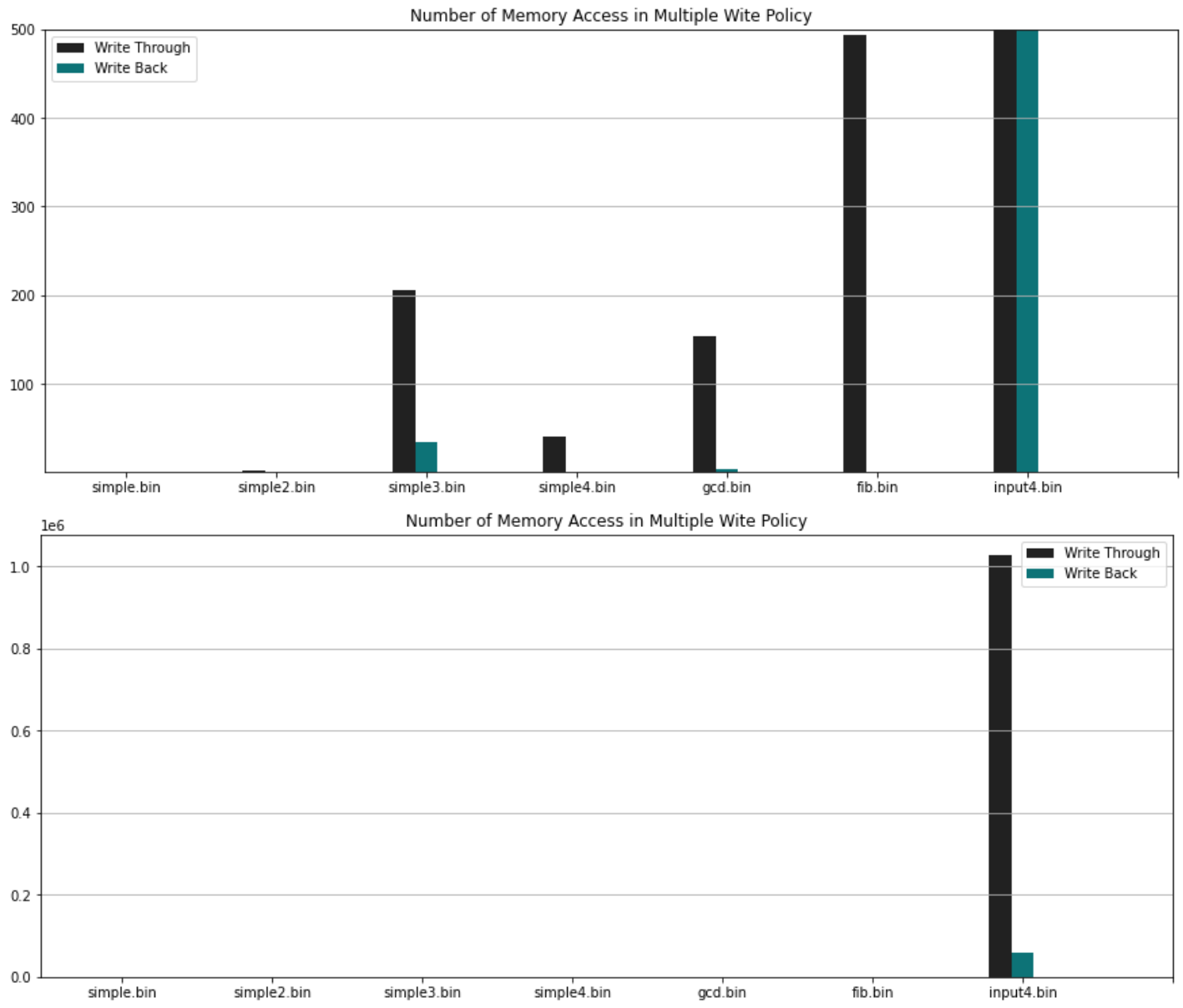


Figure 46 - Comparison Graph for Multiple Cache Write Policy

Figures 44, 45, and 46 show the comparison of the results for the cache MIPS simulator on different cache writing policies. The number presented in the table in Figure 44 is the number of memory access operations. We can check that the write-back policy shows a significantly lower number of operations that accesses memory compared to the memory through policy. In the aspect of the average memory access time, the cache operation with a write-back policy will show the lover AMAT time rather than the cache operation with a write-through policy. As a result, a cache operation with a write-back policy shows much higher cache performance than the cache operation with a write-through policy.

8-2-4.Comparison on Cache Replacement Policy

Program	FIFO	Random	LRU	LFU	SCA	ESCA
simple.bin	83.333	83.333	83.333	83.333	83.333	83.333
simple2.bin	87.5	87.5	87.5	87.5	87.5	87.5
simple3.bin	99.847	99.847	99.847	99.847	99.847	99.847
simple4.bin	97.512	97.512	97.512	97.512	97.512	97.512
gcd.bin	98.58833333	98.56016667	98.58833333	98.58833333	98.58833333	98.58833333
fib.bin	99.746	99.746	99.746	99.746	99.746	99.746
input4.bin	99.27533333	99.27166667	99.079	97.574	99.27533333	99.27533333

Figure 47 - Comparison Table for Multiple Cache Replacement Policy

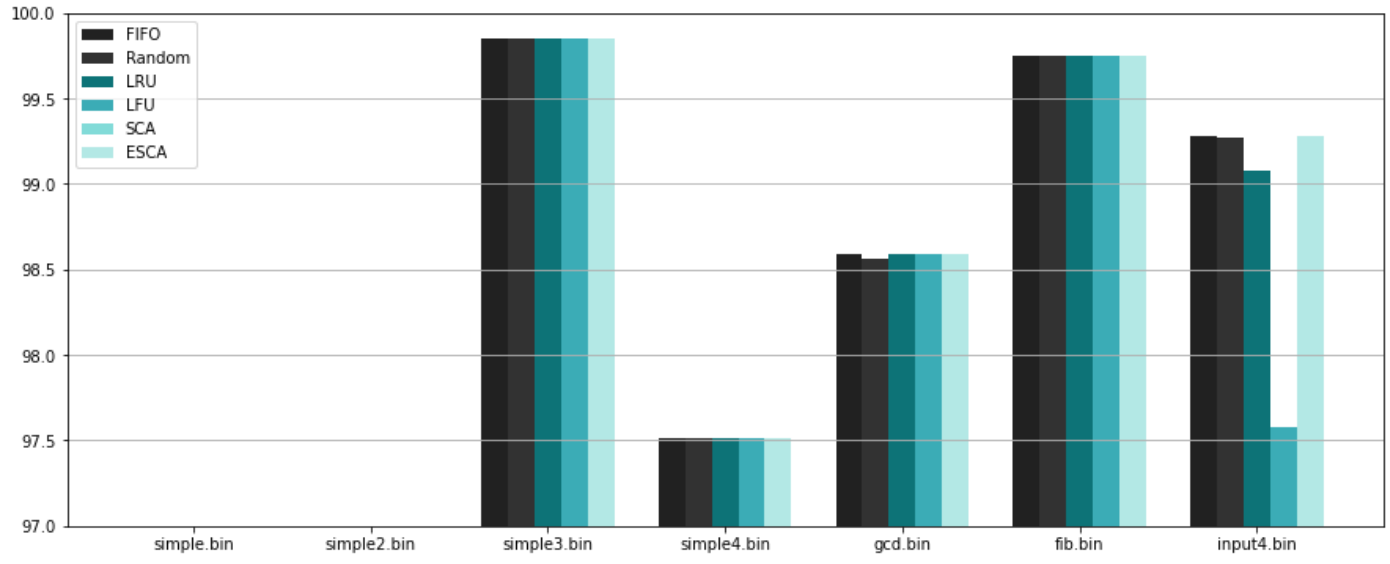


Figure 48 - Comparison Graph for Multiple Cache Replacement Policy

Figures 47 and 48 show the table and graph for comparison of results generated by the cache MIPS simulator with different cache replacement algorithms. For simple.bin, simple2.bin, simple3.bin, and simple4.bin, because they have fewer cache accessing operations and the same number of cold-misses of the cache, they show the same cache-hit ratio as a result. The derivative cache-hit ratio is shown on the other programs, which are gcd.bin, fib.bin, and input4.bin. In gcd.bin and fib.bin programs, they show a difference in the cache-hit ratio in the range of ~0.5%. However, in the input4.bin program, FIFO, Random, and enhanced second chance algorithm show the best cache-hit ratio compared to the second chance and LFU algorithms.

9. Conclusion

From sections 1 to 8, we have shown the concepts that are used to implement our pipelined MIPS simulator with cache: memory hierarchy, memory locality, hierarchical latency analysis, cache with different types, cache replacement policies, and cache write policies. By applying these concepts, we implemented a real cache MIPS simulator that follows the structure and operation of L1 cache. Then, we evaluated our simulator. By applying the useful concepts to the cache, we optimized our pipelined microarchitecture by decreasing the average memory access time (AMAT). Due to these optimizations, we enhanced the performance of the processor from the former microarchitecture, which is pipelined microarchitecture without the cache operation. We have shown the results of the program execution which are generated by cache MIPS simulator and had comparisons for the result that is generated by the cache MIPS simulator with different ways, types of the cache, write policies, and cache replacement algorithms. From this comparison, we found out some of the cache structure that generates the higher performance due to the higher cache-hit ratio. Also, pipelined microarchitecture with cache operation shows higher performance compared to pipelined microarchitecture without the cache operation.

10. Citations

- Bryant, R., & O'Halloran, D. (2010, September 23). 15-213 (18-213): Introduction to computer systems (ICS). 15-213 (18-213): Introduction to Computer Systems. Retrieved June 18, 2022, from https://www.cs.cmu.edu/afs/cs/academic/class/15213-f10/www/

- *CS 3410: Computer System Organization and programming*. CS 3410 - Spring 2019 - Computer System Organization and Programming. (n.d.). Retrieved June 18, 2022, from https://www.cs.cornell.edu/courses/cs3410/2019sp/

- Harris, D. M., & Harris, S. L. (2012). *Digital Design and computer architecture*. Elsevier/Morgan Kaufmann.

- Jose Nelson Amaral homepage. (n.d.). Retrieved June 18, 2022, from https://webdocs.cs.ualberta.ca/~amaral/

- Lee, C. (2022). (rep.). Project2. Seoul: LEE CHANGYOON.

- *Logo: To the web site of Uppsala University*. Simple search. (n.d.). Retrieved June 18, 2022, from https://uu.diva-portal.org/

- Schmidt, B., Gonzalez-Dominguez, J., Hundt, C., & Schlarb, M. (2018). *Parallel Programming: Concepts and Practice*. Morgan Kaufmann Publishers, an imprint of Elsevier.

- Sloss, A. N., Rayfield, J., Symes, D., & Wright, C. (2009). *Arm System Developer's Guide: Designing and Optimizing System Software*. Morgan Kaufmann/Elsevier.

- *The second chance page replacement policy*. CS355 Sylabus. (n.d.). Retrieved June 18, 2022, from http://www.mathcs.emory.edu/~cheung/Courses/355/Syllabus/9-virtual-mem/SC-replace.html

- Virtual workshop. Cornell Virtual Workshop: Memory Hierarchy. (n.d.). Retrieved June 18, 2022, from https://cvw.cac.cornell.edu/codeopt/memhier

- Yoo, S. H. (n.d.). 13-memory_intro.