# Cache Controller with Write-through and Write-around Policies

(Verilog Project)

**Presented by:**
Noura Medhat Shawky

# 1. Cache Memory

Cache memory is a chip-based computer component that makes retrieving data from the computer's memory more efficient. It acts as a temporary storage area that the computer's processor can retrieve data from easily.

## 1.1 Why do we need cache memory? [1]

Cache memory is important because it improves the efficiency of data retrieval. It stores program instructions and data that are used repeatedly in the operation of programs or information that the CPU is likely to need next. The computer processor can access this information more quickly from the cache than from the main memory. Fast access to these instructions increases the overall speed of the program.

## 1.2 Multi-level Caches [2]

Multilevel Caches is one of the techniques to improve Cache Performance by reducing the "MISS PENALTY". Miss Penalty refers to the extra time required to bring the data into cache from the Main memory whenever there is a "miss" in the cache.

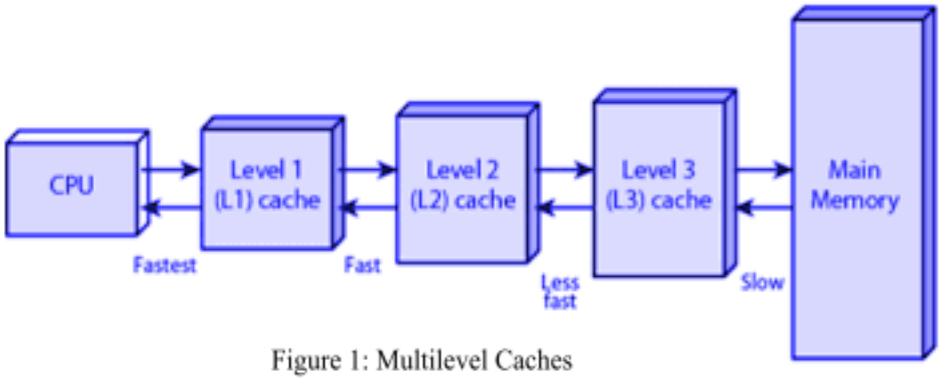


Figure 1: Multilevel Caches

## 1.3 How the processor interacts with the cache memory

When the processor needs to read or write a location in the main memory, it first checks for a corresponding entry in the cache.

- If the processor finds that the memory location is in the cache, a Cache Hit has occurred and data is read from the cache.
- If the processor does not find the memory location in the cache, a cache miss has occurred. For a cache miss, the cache allocates a new

entry and copies in data from the main memory, then the request is fulfilled from the contents of the cache.

- The performance of cache memory is frequently measured in terms of a quantity called Hit ratio**.**

## 2. Cache Mapping

There are 3 techniques for cache mapping: Direct Mapping, Associative Mapping, and Set Associative Mapping. I am going to cover the Direct Mapping technique only.

### 2.1 Direct Mapping

The simplest technique, known as direct mapping, maps each block of main memory into only one possible cache line. or In Direct mapping, assign each memory block to a specific line in the cache. If a line is previously taken up by a memory block when a new block needs to be loaded, the old block is trashed. An address space is split into two parts: index field and a tag field. The cache is used to store the tag field whereas the rest is stored in the main memory. Direct mapping`s performance is directly proportional to the Hit ratio.

$i = j \, modulo \, m$

Where:

$i$ is the cache line number

$j$ is the main memory block number

$m$ is the number of lines in the cache

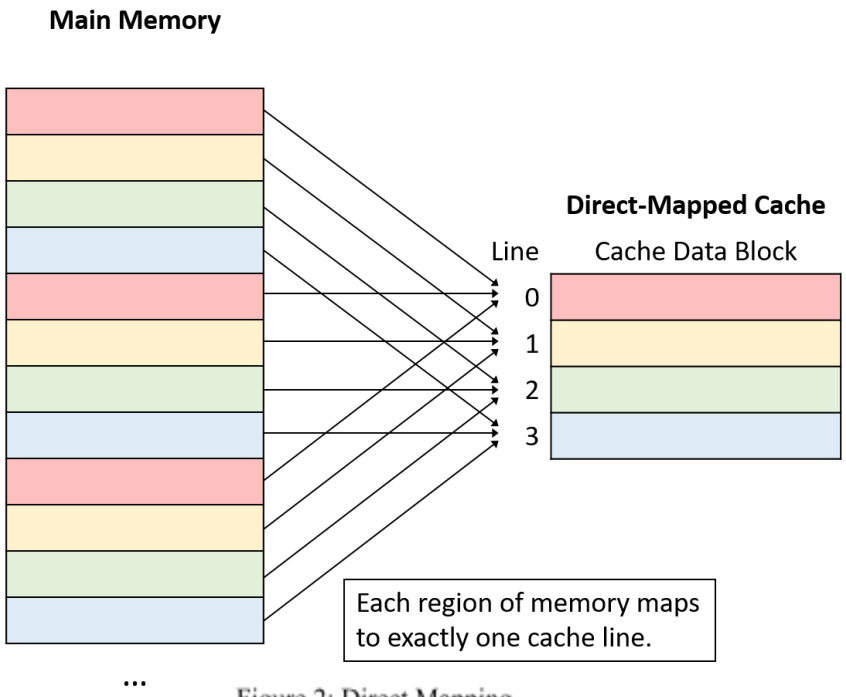


Figure 2: Direct Mapping

# 3. Caching Strategies

## 3.1 Write-Through

We can update the value in the cache and avoid expensive main memory access. But this results in Inconsistent Data Problem. As both cache and main memory have different data, it will cause problems in two or more devices sharing the main memory (as in a multiprocessor system).
This is where **Write Through** comes into the picture.

In write-through, data is simultaneously updated to cache and memory. This process is simpler and more reliable. This is used when there are no frequent writes to the cache(The number of write operations is less).
It helps in data recovery (In case of a power outage or system failure). A data write will experience latency (delay) as we have to write to two locations (both Memory and Cache). It Solves the inconsistency problem.

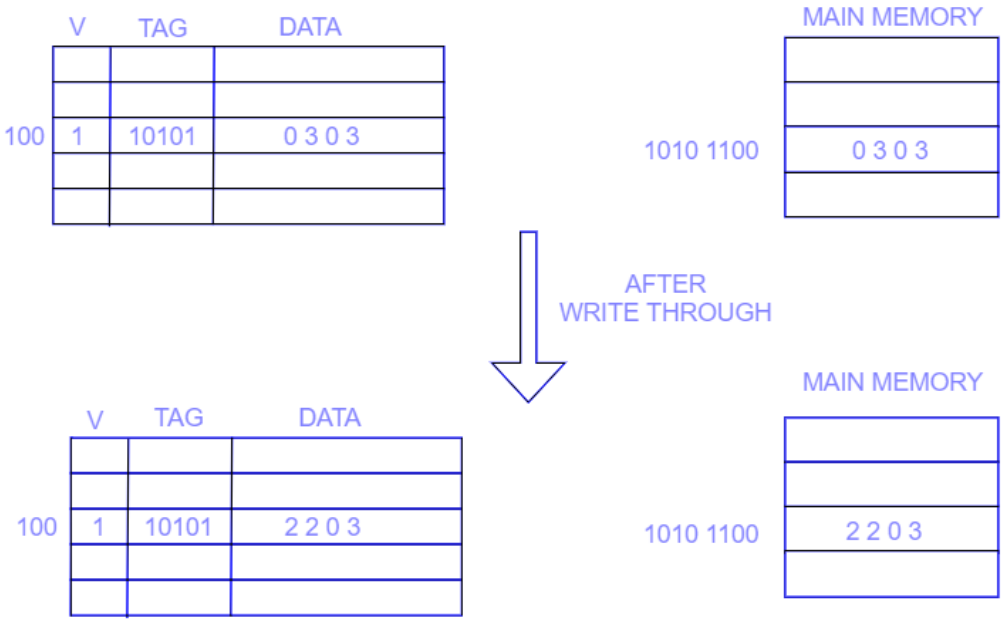


Figure 3: Write Through

## 3.2 Write-Around

Using the write-around policy, data is written only to the data memory without writing to the cache.

# 4. Cache Controller with Write-Through and Write-Around Policies

In this project, we will work on implementing a simple caching system for the RISC-V processor. For simplicity, we will integrate the caching system with the single-cycle implementation. Additionally, we assume the following:

- Only data memory will be cached. The instruction memory will not be affected.
- We will have only one level of caching.
- The main memory module is assumed to have a capacity of 4 Kbytes (word addressable using 10 bits)
- Main memory access (for read or write) takes 4 clock cycles
- The data cache geometry is (512, 16, 1). This means that the total cache capacity is 512

bytes, that each cache block is 16 bytes, and that the cache uses direct mapping.

- The cache uses write-through and write-around policies for write hit and write miss handling.
- LW instructions will only stall the processor in case of a miss.

## 4.1 Cache Design

In this project, the cache is required to be 512B, and each word is 4B. So, in total we have 128 words, and therefore we need a 7 bit address to access any word in the cache.

The line in the cache consists of 16 bytes, 4 words, which means that we have 32 lines in the cache. The lines in it can be illustrated as follows:

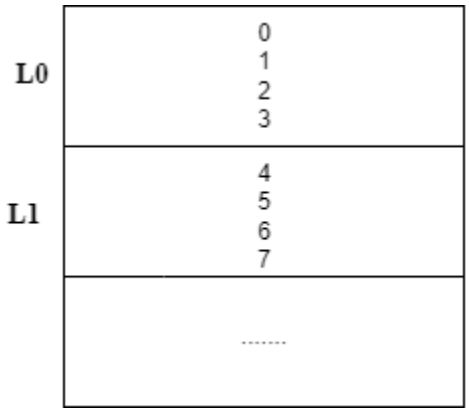


Figure 4: Lines in Cache Memory

## 4.2 Data Memory Design

The data memory is required to be 4KB, and each word is 4B. So, in total we have 1024 words, and therefore we need a 10 bit address to access any word in the data memory. This means we have 256 blocks in the data memory.

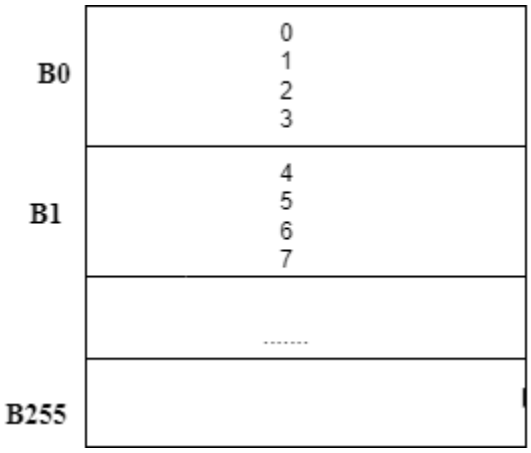


Figure 5: Blocks in Data Memory

## 4.3 Physical Address

As explained, the cache requires only a 7 bit address and the data memory requires a 10 bit address. So, how to map this physical address into a logical one?

- First, each block in the data memory, or each line in the cache memory, consists of 4 words. So, to determine which word we want to access, we need 2 bits.
- Second, the blocks in the data memory are mapped to 32 lines in the cache. So, to select which line we are talking about, we need 5 bits.
- Third, different blocks in the data memory can be mapped to the same line in the cache. So, the last 3 bits are used for this task, these bits are known as tag bits.


Figure 6: Physical Address

To determine whether the data in the cache are valid or not, we will have a valid bit for each address.

0: Not valid

1: Valid

Finally, the cache can be illustrated as follows:

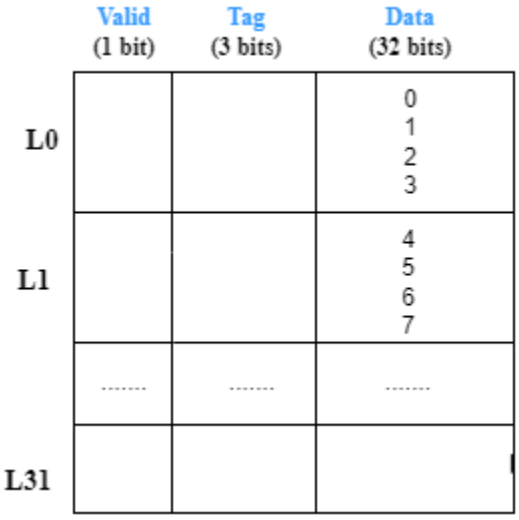


Figure 7: The Final Internal Architecture of the Cache

## 4.4 FSM of the Cache Controller

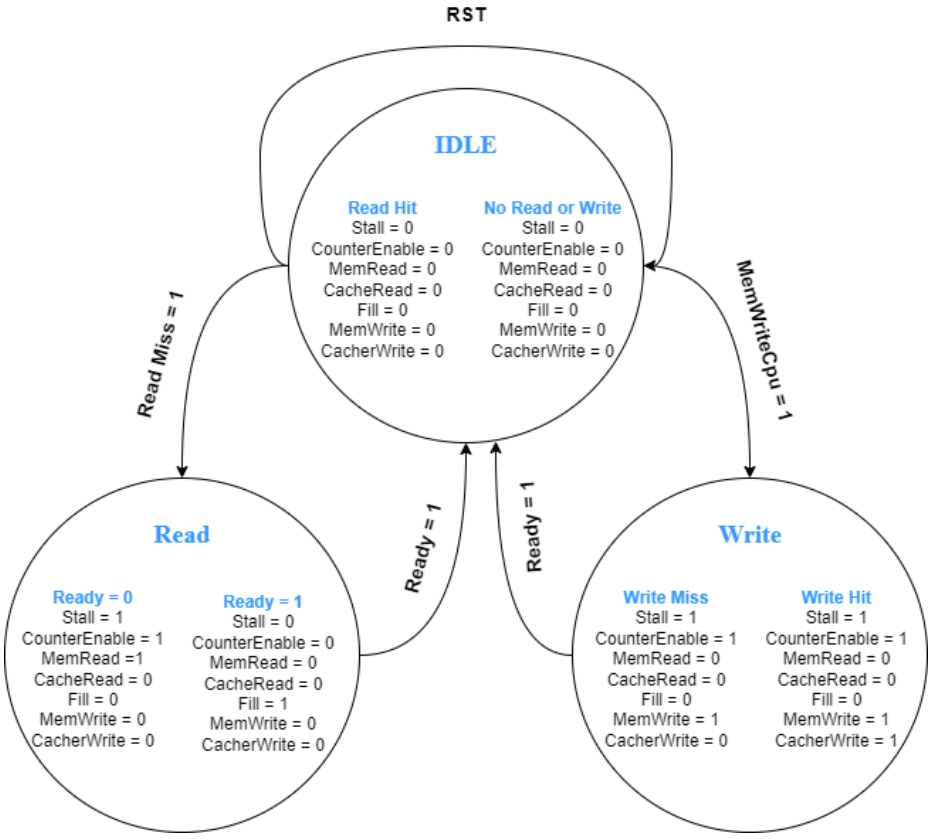


Figure 8: Cache Controller FSM

Here, we have 4 scenarios below:

- The processor requests a read operation (executing a LW instruction) and the cache controller decides that it is a hit. In this case, there is no stall necessary and the data is read from the cache module.
- The processor requests a read operation (again executing a LW instruction) and the cache controller decides that it is a miss. In this case, the stall signal is asserted and the data is read from the data memory module which provides 1 block (16 bytes or 128 bits) of data to the data cache. When this data is available, the data memory module asserts a ready signal that the cache controller uses to ask the data cache to fill the corresponding block with the data coming from the memory and to deassert the stall signal.
- The processor requests a write operation (executing a SW instruction) and the cache controller decides that it is a hit. In this case, the word to be stored has to be written both in the cache memory and in the data

memory (due to the write-through policy). So the cache controller asserts the stall signal until the memory confirms that it finished writing via its ready signal. Simultaneously the cache controller asks the cache memory to update the value.

- The processor requests a write operation (again executing a SW instruction) and the cache controller decides that it is a miss. In this case, the word to be stored is written in the data memory only (due to the write-around policy); however, in this case too, the cache controller asserts the stall signal until the memory finishes the storing

## 4.5 Required Blocks

From the previous specifications, the required blocks are: Cache Memory, Data Memory, Counter, and Control Unit.

### 4.5.1 Cache Memory Block

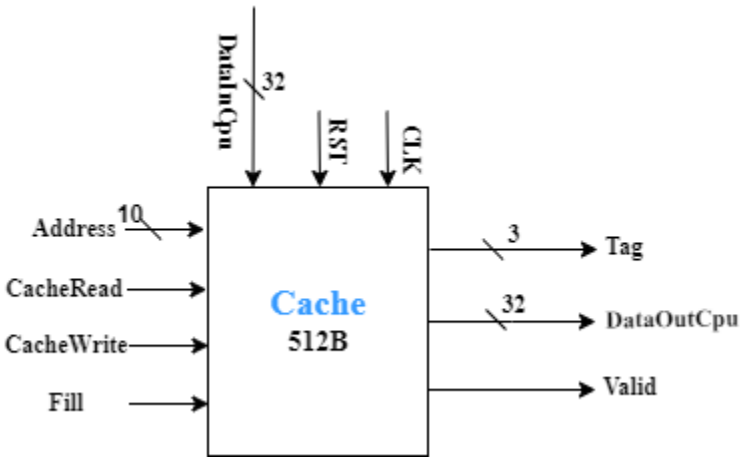


Figure 9: Cache Memory Block

- **CacheRead:** This signal is asserted in case the controller decides that it is a read hit. If asserted, then a word is read from the cache as an output on the **DataOutCpu** port.

- **Fill:** This signal is asserted in case the controller decides that it is a read miss. If asserted, then a block, 4 words, is written in the cache and a word is read from the cache as an output on the DataOutCpu port.

- **CacheWrite:** This signal is asserted in case the controller decides that it is a write hit. If asserted, the data in the given address is updated to have the value of the **DataInCpu** port.
- **Tag:** A 3-bit port, has the value of the tag bits of the given address.
- **Valid:** A single bit output, to determine the state of the data in the cache memory.

### 4.5.2 Data Memory Block

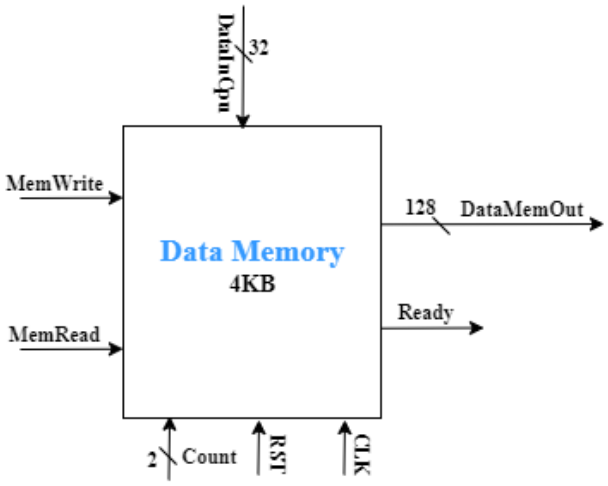


Figure 10: Data Memory Block

- **MemWrite:** This signal is asserted in case of write miss or write hit.
- **MemRead:** This signal is asserted in case of read miss.
- **Ready:** This signal is asserted after 4 clock cycle, in case of this is a read miss, write miss or write hit.
- **Count:** 2-bit input port, used to determine when to assert the ready signal.
- **DataMemOut:** 128-bit output port, used to pass a complete block from the data memory to the cache, in case of the occurrence of read miss.

### 4.5.3 Counter

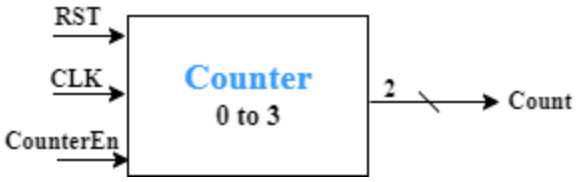


Figure 11: Counter

The counter is mainly used to determine the end of the following: read miss, write miss, and write hit. It is used to count from 0 to 3, one count per clock cycle. So, reaching 3 means that 4 clock cycles have passed.

- **CounterEn:** This signal is asserted by the controller, in case of read miss, write hit, or write miss.

### 4.5.4 Cache Controller

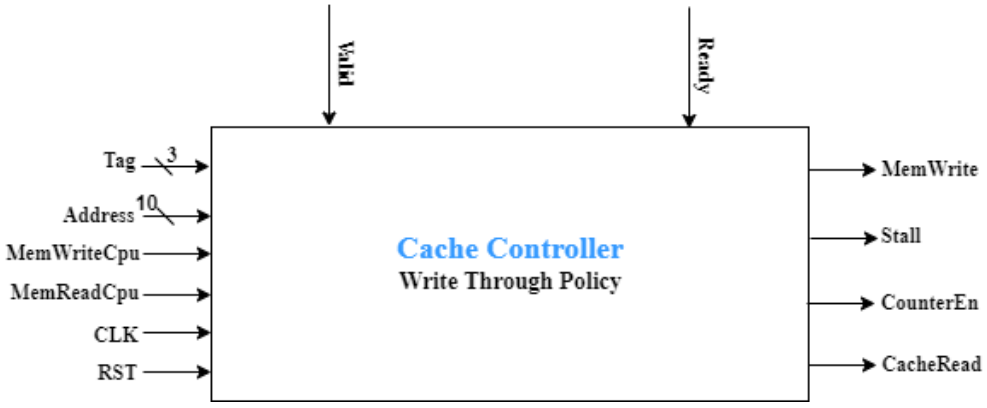


Figure 12: Cache Controller Block

This block was designed and implemented to imply the function of the mentioned FSM. The cache controller encapsulates the array of tags and valid bits and uses the index and tag parts of the requested memory address to decide whether there is a hit or a miss. It is also responsible for generating the stall control signal in addition to controlling both the cache module and the memory .

We can now explain how the 4 mentioned scenarios can be covered:

- **Scenario One - Read hit:** If the tag bits were equal to the last 3 bits in the address, the valid bit equal 1, and the MemReadCpu signal was asserted as an indication of LW instruction, then the CacheRead signal will be asserted to read a single word for the given address from the cache.
- **Scenario Two - Read miss:** If the tag bits weren't equal to the last 3 bits in the address or the valid bit equal 0, and the MemWriteCpu signal was asserted as an indication of LW instruction, then the Stall signal, the CounterEn signal, and the MemRead signal will be asserted to read a single block from the data memory for 4 clock cycles. After that, and when the Ready signal is high, the Stall signal, the CounterEn signal, and the MemRead signal will be de-asserted, and the Fill signal will be asserted, in order to make the cache read this block and output the needed data to the CPU again.
- **Scenario Three - Write hit:** If the tag bits were equal to the last 3 bits in the address, the valid bit equal 1, and the MemWriteCpu signal was asserted as an indication of SW instruction, then the MemWrite signal, the Stall signal will be asserted to update a single word for the given address in the data memory, for 4 clock cycles. After that, and when the Ready signal is high, the Stall signal, the CounterEn signal, and the MemWrite signal will be de-asserted, and the CacheWrite signal will be asserted, in order to update the same word in the cache memory, due to the write-through policy.
- **Scenario Four - Write miss:** If the tag bits weren't equal to the last 3 bits in the address, the valid bit equal 0, and the MemWriteCpu signal was asserted as an indication of SW instruction, then the MemWrite signal, the Stall signal will be asserted to update a single word for the given address in the data memory, for 4 clock cycles. After that, and when the Ready signal is high, the Stall signal, the CounterEn signal, and the MemWrite signal will be de-asserted, and nothing will happen in the cache, due to the write-around policy.

### 4.5.5 Top Design Architecture

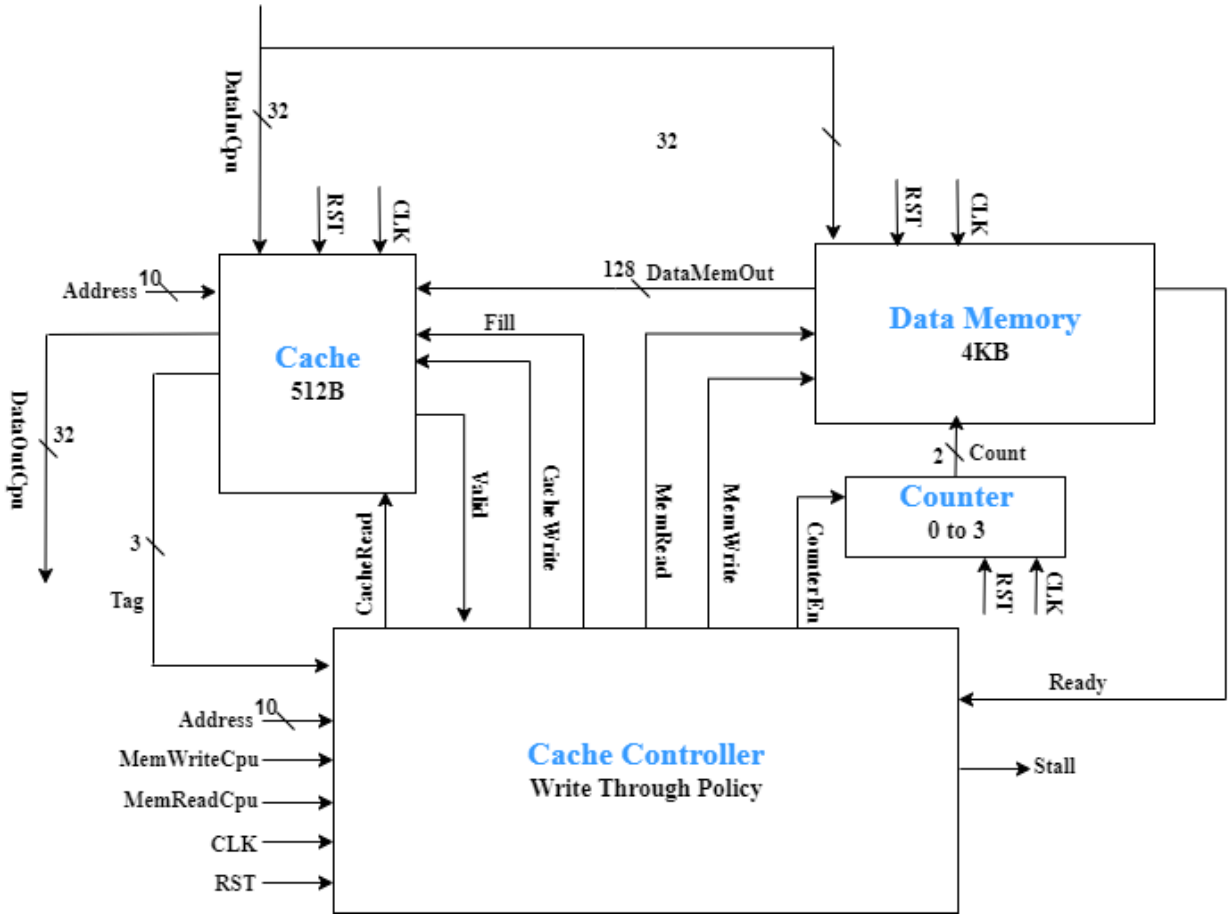


Figure 13: Top Design Architecture

The mentioned 3 blocks were integrated into one top architecture.

## 4.6 The Integration with a Single-Cycle RISC Processor

The previous cache module was integrated with a Single-Cycle RISC processor.

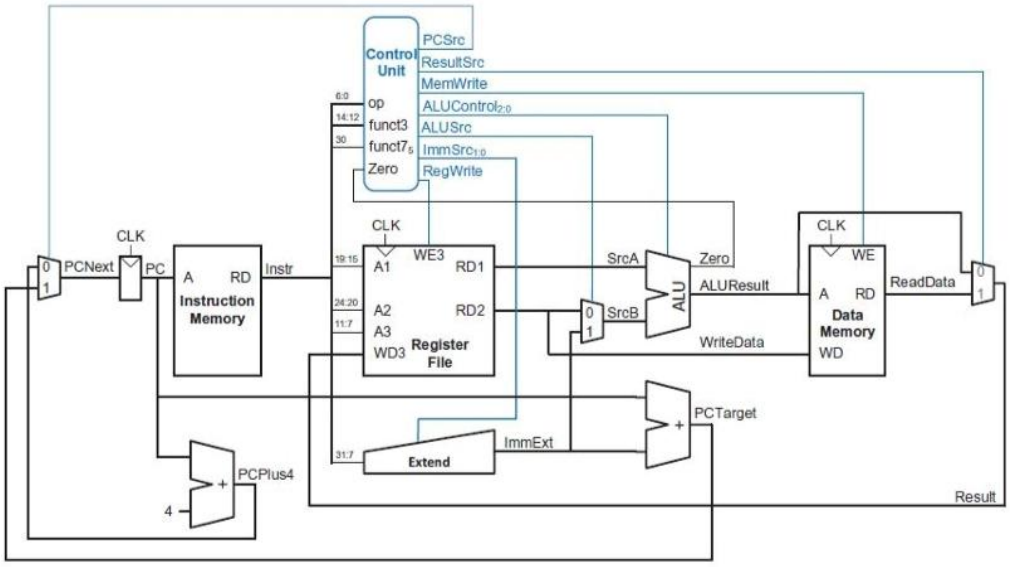


Figure 14: Single-Cycle RISC-V Architecture without the Cache Module

The implemented Single-Cycle RISC-V from **"Digital Design and Computer Architecture, RISC-V Edition"** reference, doesn't have a MemRead control signal, and the output address from the ALU is 32bits, while the required address by the cache module is 10 bits.

- **MemReadCpu Control Signal:** To solve this one, a control signal, MemReadCpu, was added to the control unit of the RISC-V. This signal is asserted in case of LW instruction.
- **Cache Module Address:** To make the integrated system operates properly, we are going to connect only the least significant 10 bits of the ALUResult to the cache module address.
- **Stall Control Signal:** To temporarily stop the processor when needed, the Stall signal is going to act as an active low enable to the program counter.

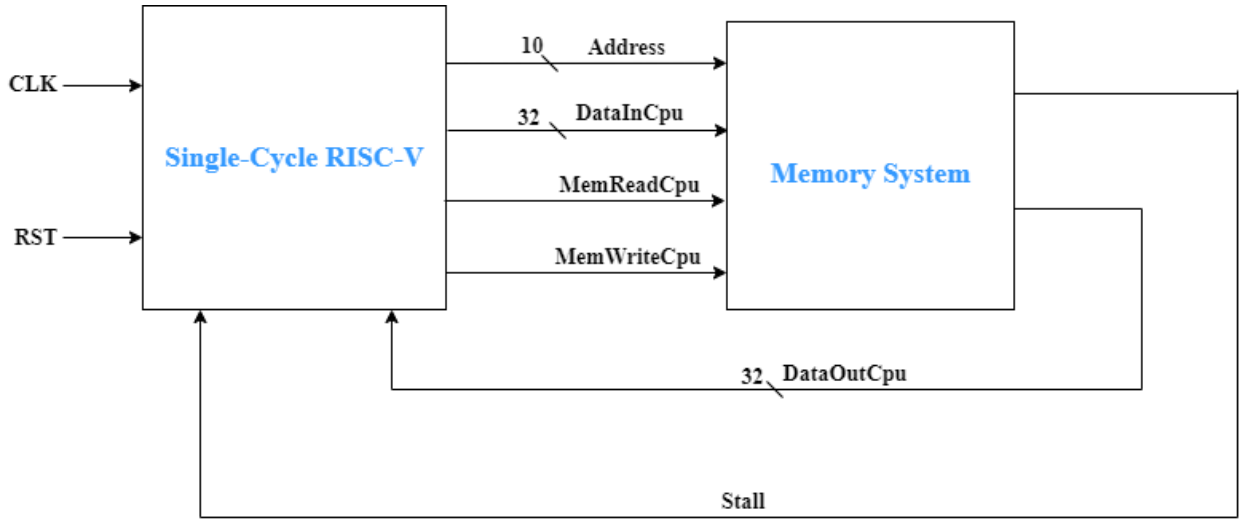


Figure 15: Single-Cycle RISC-V Architecture with the Cache Memory System

## 4.7 Testing

### 4.7.1 Cache Module Testing

To test the memory system, we need to cover the 4 mentioned scenarios. I have applied 6 test cases:

1. Reset TC
2. Write Miss TC
3. Read Miss TC
4. Write Hit TC
5. Read Hit TC
6. Read Hit TC, to test that a complete block was read from the data memory.
7. Read Miss TC: For Data Memory alignment
8. Read Hit TC: For Cache Memory alignment

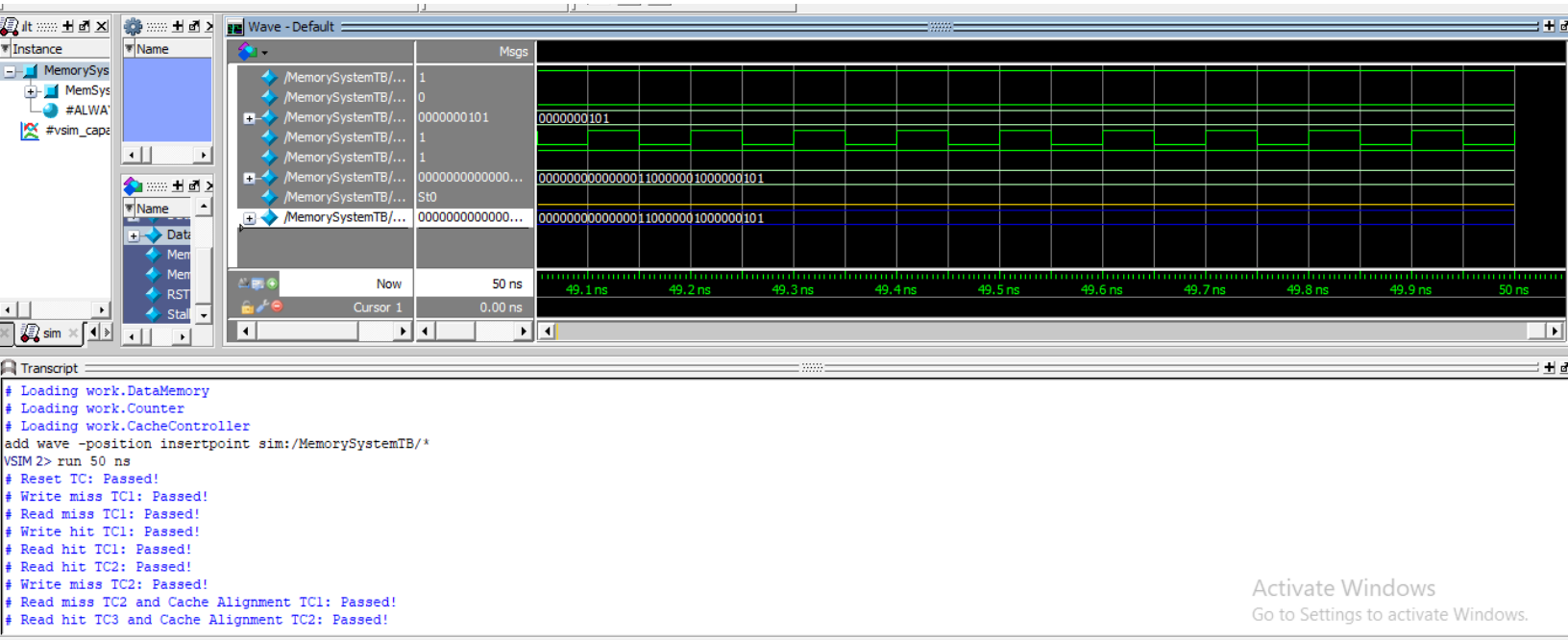


Figure 16: Memory System Testbench Results

### 4.7.2 Single-Cycle RISC-V with the Cache Module Testing

The whole processor was tested with the Fibonacci Series program, and everything was as it should be.

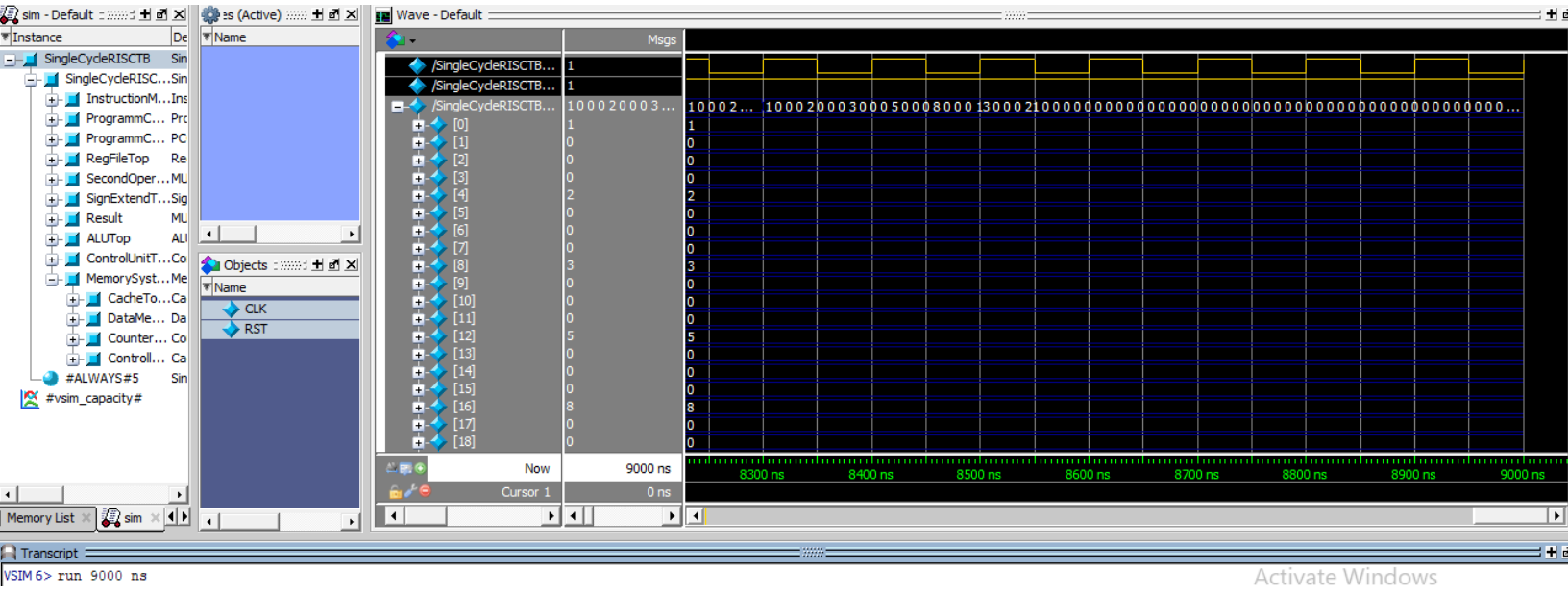


Figure 17: Integrated System Testing