

**Management unit for the cache memory control**

Student: Pătran Raluca-Maria

Group: 30433

**Contents**

Introduction . . . . . . . . . . . . . . . . . . 3

Objectives . . . . . . . . . . . . . . . . . . . 5

Theoretical considerations . . . . . . . . . . . . . 6

The CPU . . . . . . . . . . . . . . . . . . . . . 6

The cache memory . . . . . . . . . . . . . . . . 8

The cache controller . . . . . . . . . . . . . . 11

The RAM . . . . . . . . . . . . . . . . . . . . . 12

How caching works . . . . . . . . . . . . . . . . 13

Implementation . . . . . . . . . . . . . . . . . . 14

Testing . . . . . . . . . . . . . . . . . . . . . . 33

Conclusions. . . . . . . . . . . . . . . . . . . . 36

References. . . . . . . . . . . . . . . . . . . . . 37

Introduction

Random-access memory (**RAM**) is a type of computer memory that can be accessed randomly: any byte of memory can be accessed without touching the preceding bytes. It is considered to be the main memory of the computer and can be found in computers, smartphones, printers etc.

There are two types of RAM:

1. **DRAM** (Dynamic Random Access Memory)

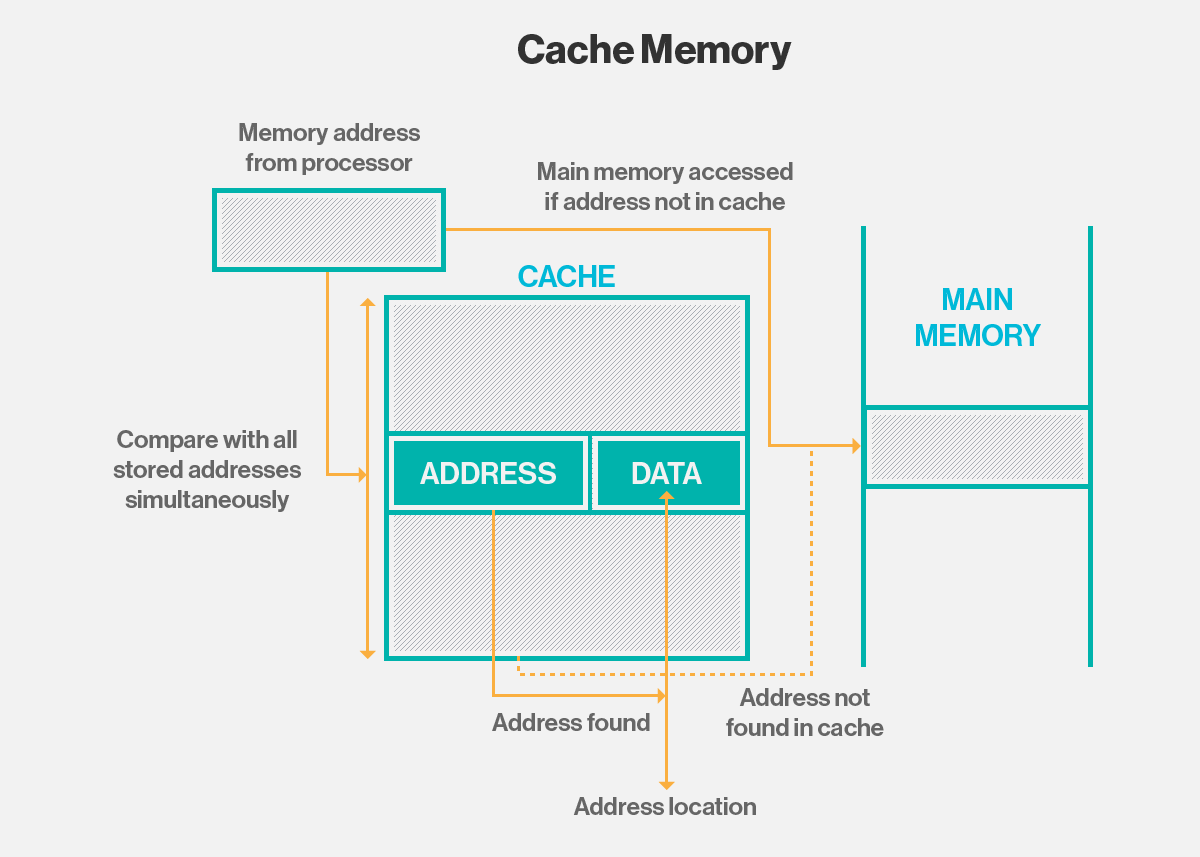
2. **SRAM** (Static Random Access Memory)

Dynamic Random Access Memory is used for the main memory and must be refreshed constantly or it will lose its contents. Static Random Access Memory is used for system cache and because of this it does not need to be refreshed. Both DRAM and SRAM are volatile, meaning that they lose their contents when the power is turned off.

The cache memory, known as the Central Processing Unit (**CPU**) memory, is a SRAM. It is integrated directly into the CPU chip, or placed on a separate chip that has a bus interconnect with the CPU. Its role is to store program instructions and data that are used repeatedly in the operations of programs or information the CPU is likely to need next. The computer processor can access this information quickly from the cache rather than having to get it from computer's main memory. As the microprocessor processed data, it looks first in the cache memory. We also have the word “cashing” which is the process of storing data in a cache.

2

Below we can see how the cache memory works.



Suppose we have a file in the main memory. That file will be stored as blocks in RAM. If the CPU wants to make use of that file or open it, first it will “ask” that cache if it has it. The cache obviously does not have it so the CPU will go and “ask” the RAM for the file. Here, the CPU finds the file and grabs a block and stores it. Now, the cache controller comes in. It “sees” that the CPU is trying to make use of a certain file so it goes into the RAM and grabs the rest of the blocks that make the file and stores them in cache. When the CPU will “ask” the cache again if it has the file, now the cache will be able to give the CPU the entire file. And because this is happening, the CPU does not need to go and look in the RAM for the file.

RAM

1

CPU

Cache

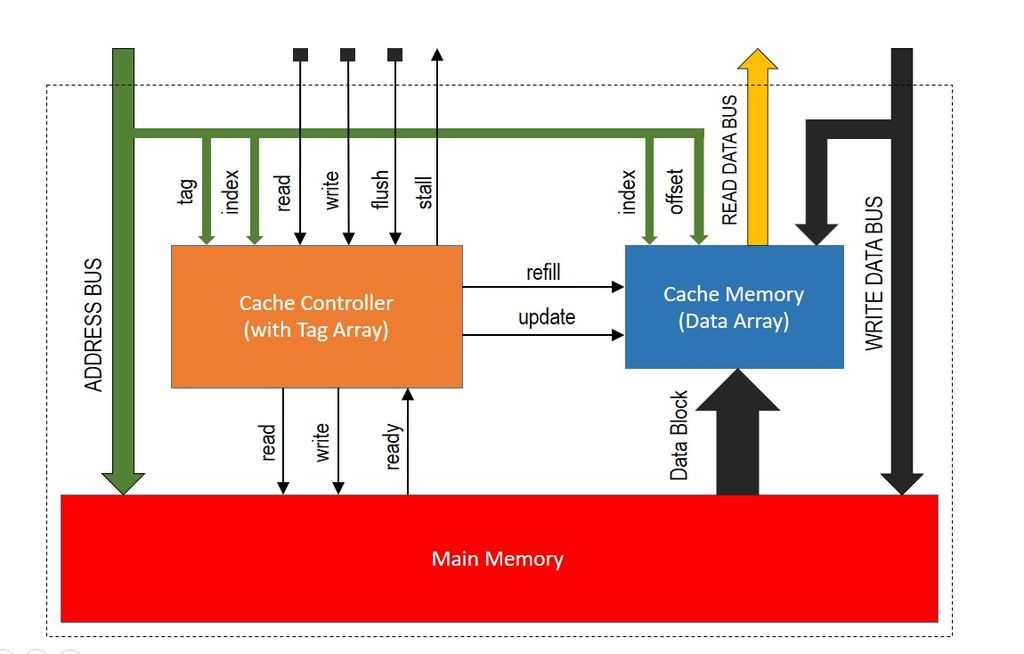
Cache controller

2

Objectives

The purpose of this project is to implement and design a cache memory in Xilinx. The board should display the address at which certain data is stored or the other way around: the data which is stored at a certain address. The entered data should be given from the buttons/switches and with the help of a switch we will be able to see on the 7 segment display the specific data or the address at which the data is stored (1 for the address and 0 for the data).

The scheme for the cache controller with cache memory and RAM memory will be the one presented below:



Theoretical considerations

**1.** The central processing unit (CPU)

The central processing unit (CPU) is the unit which performs most of the processing inside a computer. To control instructions and data flow to and from other parts of the computer, the CPU relies heavily on a chipset, which is a group of microchips located on the motherboard.

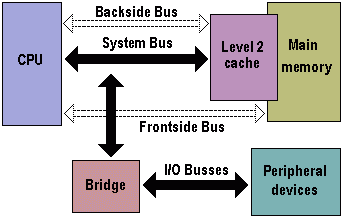
The CPU has two components:

* **Control Unit**: extracts instructions from memory and decodes and executes them
* **Arithmetic Logic Unit** (ALU): handles arithmetic and logical operations

To function properly, the CPU relies on the system clock, memory, secondary storage, and data and address buses.

This term is also known as a central processor, microprocessor or chip. Additionally, the CPU has an internal bus for communication with the internal cache memory, called the backside bus. The fundamental operation of most CPUs, regardless of the physical form they take, is to execute a sequence of stored [instructions](https://en.wikipedia.org/wiki/Instruction_(computing)) that is called a program. The instructions to be executed are kept in some kind of [computer memory](https://en.wikipedia.org/wiki/Memory_(computers)). Nearly all CPUs follow the fetch, decode and execute steps in their operation, which are collectively known as the [instruction cycle](https://en.wikipedia.org/wiki/Instruction_cycle). The main bus for data transfer to and from the CPU, memory, chipset, and AGP socket is called the front-side bus.

Below we can see the dual independent bus (DIB) architecture that allows the processor to use both buses simultaneously, and also confers other advantages:

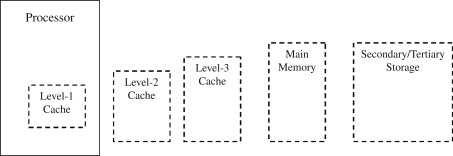
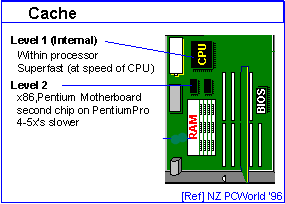


Most high-end microprocessors (in desktop, laptop, server computers) have a memory management unit, translating logical addresses into physical random access memory (RAM) addresses, providing [memory protection](https://en.wikipedia.org/wiki/Memory_protection) and [paging](https://en.wikipedia.org/wiki/Paging) abilities, useful for [virtual memory](https://en.wikipedia.org/wiki/Virtual_memory). Simpler processors, especially [microcontrollers](https://en.wikipedia.org/wiki/Microcontroller), usually don't include a memory management unit (MMU).

A CPU cache is a [hardware cache](https://en.wikipedia.org/wiki/Hardware_cache) used by the central processing unit (CPU) of a [computer](https://en.wikipedia.org/wiki/Computer) to reduce the average cost (time or energy) to access [data](https://en.wikipedia.org/wiki/Data_(computing)) from the [main memory](https://en.wikipedia.org/wiki/Main_memory). A cache is a smaller, faster memory, closer to a [processor core](https://en.wikipedia.org/wiki/Processor_core), which stores copies of the data from frequently used main [memory locations](https://en.wikipedia.org/wiki/Memory_location). Most CPUs have different independent caches, including [instruction](https://en.wikipedia.org/wiki/Instruction_cache) and [data caches](https://en.wikipedia.org/wiki/Data_cache), where the data cache is usually organized as a hierarchy of more cache levels (L1, L2, L3, L4, etc.).

**2.** The cache memory

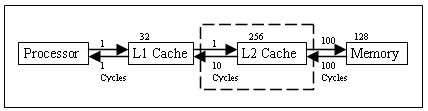
A level 1 cache (L1 cache) is a memory cache that is directly built into the microprocessor, which is used for storing the microprocessor’s recently accessed information, thus it is also called the primary cache. It is also referred to as the internal cache or system cache. L1 cache is the fastest cache memory, since it is already built within the chip with a zero wait-state interface. However, it has limited size. In my project I am going to use an additional 2-level cache. A level 2 cache (L2 cache) is a CPU cache memory that is located outside and separate from the microprocessor chip core, although, it is found on the same processor chip package. It is also called the secondary cache or external cache. It is outside of the core, the capacity can be increased and it is still faster than the main memory. Its main goal is to provide the necessary stored information to the processor without any interruptions or any delays or wait-states. It also helps in reducing the access time of data, especially in certain events wherein that specific data was already accessed before, so it doesn’t have to be loaded again. It has more capacity than the level 1 cache.



Above we can see exactly where L1 and L2 cache are placed

Both of these depend on the CPU. There are CPUs which have no cache at all, there are CPUs which have the L1 cache on die and the L2 cache on a separate die on the same chip or even on a separate chip, or there are CPUs which have both L1 and L2 cache on the same die as the CPU core. L1 has a smaller memory capacity than L2. Also, L1 can be accessed faster than L2. Moreover, L2 is accessed only if the requested data is not found in L1. Therefore, L1 has a very little delay compared to L2. Because L1 is implemented using static random access memory (SRAM) and L2 is implemented using dynamic random access memory (DRAM), L1 does not need refreshing, while L2 needs to be refreshed. If the caches are strictly inclusive, all data in L1 can be found in L2 as well. However, if the caches are exclusive, same data will not be available in both L1 and L2.

Consider the idealized memory hierarchy depicted below:



Simplified L2 cache model

We can clearly observe that the data the CPU is searching for if not found in the memory, is first looked up in L1 Cache and if it cannot find it there, it keeps on looking in the level 2 cache.

Data is transferred between memory and cache in blocks of fixed size, called cache lines or cache blocks. When a cache line is copied from memory into the cache, a cache entry is created. The cache entry will include the copied data as well as the requested memory location (called a tag).

When the processor needs to read or write a location in memory, it first checks for a corresponding entry in the cache. The cache checks for the contents of the requested memory location in any cache lines that might contain that address. If the processor finds that the memory location is in the cache, a cache hit has occurred. However, if the processor does not find the memory location in the cache, a cache miss has occurred. In the case of a cache hit, the processor immediately reads or writes the data in the cache line. For a cache miss, the cache allocates a new entry and copies data from main memory, then the request is fulfilled from the contents of the cache.

Cache row entries usually have the following structure:

Flag bits

Data block

Tag

The data block (cache line) contains the actual data fetched from the main memory. The tag contains (part of) the address of the actual data fetched from the main memory. An instruction cache requires only one flag bit per cache row entry: a valid bit. The valid bit indicates whether or not a cache block has been loaded with valid data. The "size" of the cache is the amount of main memory data it can hold. This size can be calculated as the number of bytes stored in each data block times the number of blocks stored in the cache. An effective memory address which goes along with the cache line (memory block) is split ([MSB](https://en.wikipedia.org/wiki/Most_significant_bit) to [LSB](https://en.wikipedia.org/wiki/Least_significant_bit)) into the tag, the index and the block offset.

The index describes which cache set that the data has been put in. The index length is {\displaystyle \lceil \log \_{2}(s)\rceil }log2(s) bits for s cache sets. The block offset specifies the desired data within the stored data block within the cache row.

Block offset

Index

Tag

**3.** The cache controller

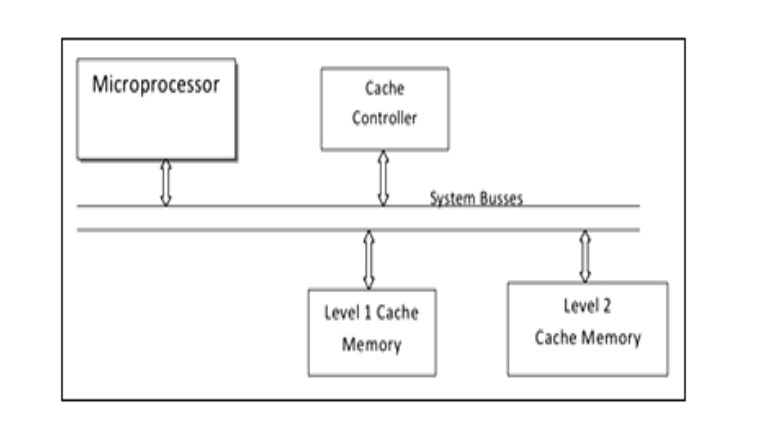
Cache controller is a device which is used to control the transfer of data between the processor, main memory and the cache memory. It is largely invisible to the program and It detects cache misses and controls sending and receiving the cells. It automatically writes code or data from main memory into the cache. The cache controller contains a status register, which can be read by the processor. It takes read and write memory requests from the core and performs the necessary actions to the cache memory or the external memory. When it receives a request from the core, it must check to see whether the requested address is to be found in the cache. This is known as a cache look-up. It does this by comparing a subset of the address bits of the request with tag values associated with lines in the cache. If there is a match, known as a hit, and the line is marked valid, then the read or write occurs using the cache memory. When the core requests instructions or data from a particular address, but there is no match with the cache tags, or the tag is not valid, a cache miss results and the request must be passed to the next level of the memory hierarchy, an L2 cache, or external memory. It takes the tag and the index for the cache memory in order to know where to write the data. The cache controller designed here consists of four operations: fetching address from the processor, read cache and main memory, write main memory and cache and providing the required data to the processor.

**4.** The random access memory (RAM)

RAM (Random Access Memory) is the hardware in a computing device where the operating system ([OS](https://whatis.techtarget.com/definition/operating-system-OS)), application programs and data in current use are kept so they can be quickly reached by the device's [processor](https://whatis.techtarget.com/definition/processor). RAM is the main memory in a computer, and it is much faster to read from and write to than other kinds of storage, such as a hard disk drive ([HDD](https://searchstorage.techtarget.com/definition/hard-disk-drive)), solid-state drive ([SSD](https://searchstorage.techtarget.com/definition/SSD-solid-state-drive)) or optical drive. It is the physical [hardware](https://www.lifewire.com/computer-hardware-2625895) inside a computer that temporarily stores data, serving as the computer's "working" memory. The term random access as applied to RAM comes from the fact that any storage location, also known as any memory address, can be accessed directly. To find a specific cell, the RAM controller sends the column and row address down a thin electrical line etched into the chip. Each row and column in a RAM array has its own address line. Any data that's read flows back on a separate data line. RAM is physically small and stored in [microchips](https://whatis.techtarget.com/definition/microchip). It's also small in terms of the amount of data it can hold. Static RAM is mainly used in small amounts as [cache memory](https://searchstorage.techtarget.com/definition/cache-memory) inside a computer's processor. It contains memory cells and in order to be useful, memory cells must be readable and writeable. Within the RAM device, multiplexing and demultiplexing circuitry is used to select memory cells. Typically, a RAM device has a set of address lines A0... An, and for each combination of bits that may be applied to these lines, a set of memory cells are activated. Due to this addressing, RAM devices virtually always have a memory capacity that is a power of two.

**5.** How caching works

CPU caches are small pools of memory that store information the CPU is most likely to need next. Which information is loaded into cache depends on sophisticated algorithms and certain assumptions about programming code. The CPU has the level 1 cache integrated. This small cache has a dual purpose, having both an instruction cache and a data cache. The instruction cache deals with the operations the CPU has to perform, and the data cache holds the information on which the process has to be done. Next, there is the L2 cache. The L2 is slower and holds more information than the L1.  While the program is running, the CPU looks for the information it needs to run, starting in the L1 cache and working backwards from there. If the CPU finds the needed information, it’s called a cache hit. If it cannot find the information it needs, it is a cache miss, and the computer has to go looking somewhere else to find the information it needs. This is where the L2 cache comes into play — while it’s slower, it’s also much larger. Some processors use an inclusive cache design (meaning data stored in the L1 cache is also duplicated in the L2 cache) while others are exclusive (meaning the two caches never share data). Latency is an important factor in the efficiency of a computer. Latency is the time needed for a piece of information to be retrieved. The L1 cache is the fastest, and therefore it has the lowest latency. When a cache miss occurs, latency increases as the computer must keep searching in different caches to find the information it needs.



**6**. Direct Mapping

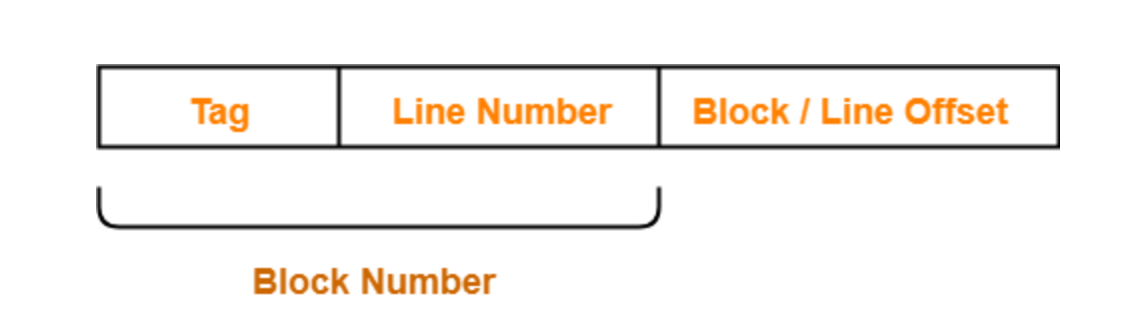
In a direct mapped cache structure, the cache is organized into multiple sets with a single cache line per set. Based on the address of the memory block, it can only occupy a single cache line. The cache can be framed as a (n\*1) column matrix.

In direct mapping,

* A particular block of main memory can map to only one particular line of the cache.
* The line number of cache to which a particular block can map is given by:

|  |
| --- |
| **Cache line number**  **= ( Main Memory Block Address ) Modulo (Number of lines in Cache)** |

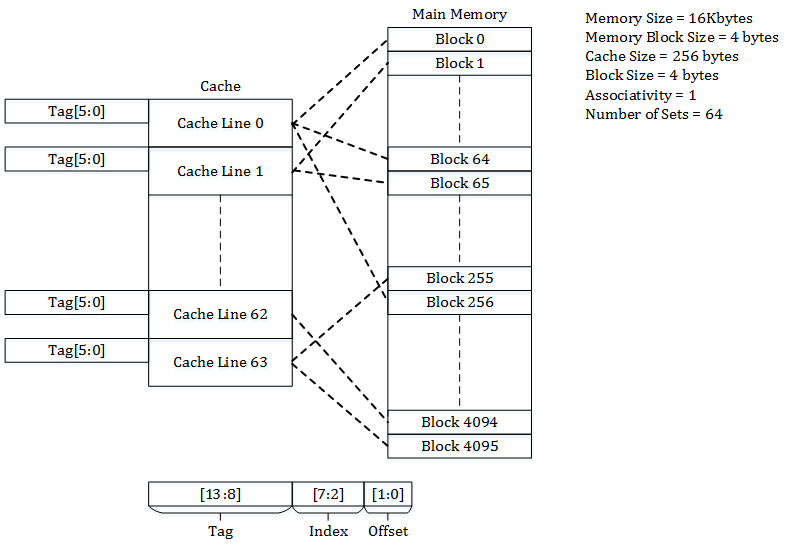
**Division of Physical Address**



In direct mapping, the physical address is divided as:

After CPU generates a memory request,

* The line number field of the address is used to access the particular line of the cache.
* The tag field of the CPU address is then compared with the tag of the line.
* If the two tags match, a cache hit occurs and the desired word is found in the cache.
* If the two tags do not match, a cache miss occurs.
* In case of a cache miss, the required word has to be brought from the main memory.
* It is then stored in the cache together with the new tag replacing the previous one.



Consider Main memory of 16 Kilobytes, which is organized as 4-byte blocks and Cache of 256 bytes with block size of 4 bytes. Since each cache block is of size 4 bytes, the total number of sets in the cache is 256/4, which equals 64 sets. The incoming address to the cache is divided into bits for [Offset](https://en.wikipedia.org/wiki/CPU_cache" \l "Cache_entry_structure" \o "CPU cache), [Index](https://en.wikipedia.org/wiki/CPU_cache#Cache_entry_structure) and [Tag](https://en.wikipedia.org/wiki/CPU_cache" \l "Cache_entry_structure" \o "CPU cache). Offset corresponds to the bits used to determine the byte to be accessed from the cache line. In the example, the offset bits are 2 which are used to address the 4 bytes of the cache line. Index corresponds to bits used to determine the set of the Cache. In the example, the index bits are 6 which are used to address the 64 sets of the cache. Tag corresponds to the remaining bits. In the example, the tag bits are 6 (14 – (6+2)), which are stored in tag field to match the address on cache request. Address 0x0000(tag - 00\_0000, index – 00\_0000, offset – 00) maps to block 0 of the memory and occupies the set 0 of the cache. Address 0x0004(tag - 00\_0000, index – 00\_0001, offset – 00) maps to block 1 of the memory and occupies the set 1 of the cache. Similarly, address 0x00FF(tag – 00\_0000, index – 11\_1111, offset – 11) maps to block 63 of the memory and occupies the set 63 of the cache. Address 0x0100(tag – 00\_0001, index – 00\_0000, offset – 00) maps to block 64 of the memory and occupies the set 0 of the cache.

Implementation

First, I will begin with more of a simple component, ad that would be the RAM memory. We are going to have a 256 x 8 single port RAM that has 1 kB. That means we will have 8-bit input data to be written to RAM at the provided input address “input\_address” when write enable “write\_enable” will be ‘1’ logic. The black box for the RAM memory is:

32

clk data\_out

reset data\_ready

read

write

RAM

8

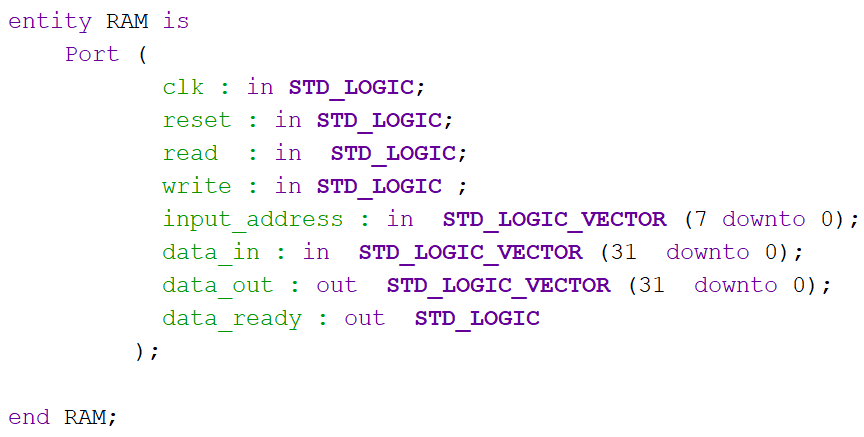
input\_address

32

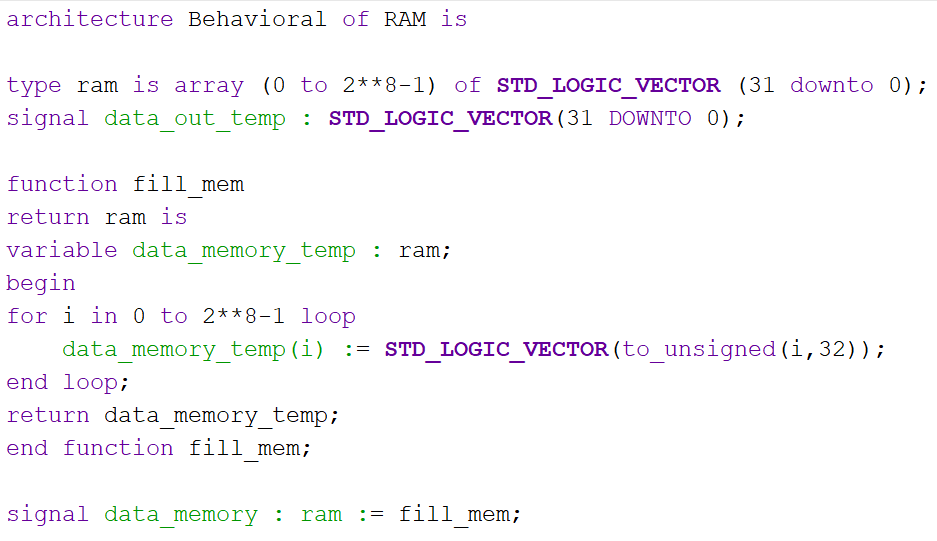
data\_in

* “clk” is the clock signal for sequentially writing data to the single-port RAM;
* “reset” is for resetting everything back to zero; it can be either ‘0’ or ‘1’ and it is active low asynchronous reset;
* “read” is used for when we are able to read from the RAM; it can be either ‘1’ or ‘0’ logic;
* “write” is used for when we are able to write into the RAM block; it can be either ‘1’ or ‘0’ logic;
* “input\_address” is 8-bit input address from RAM at which the provided data will be written;
* “data\_in” 32-bit it will be the data input for write operation;
* “data\_out” 32-bit it will be the data output for read operation;
* “data\_ready” signal for announcing the processor that the data has been written or the data can be read; it can be either ‘1’ or ‘0’ logic;

First, I declared the RAM entity which contains all the input and output signals presented before:



Next, I declared some signals that will help me later in the processes, plus a function which will be explained below:



As we can see, I have a type called “ram” which is a 2-D array type for the RAM with “locations” from 0 to 255 that can hold values of 32-bit length.

The function called “fill\_mem” is a function that uses the type “ram” declared before to fill the data memory, declared as a variable “data\_memory\_temp” with default values of 32-bit length from locations 0 to 255 and then return it.

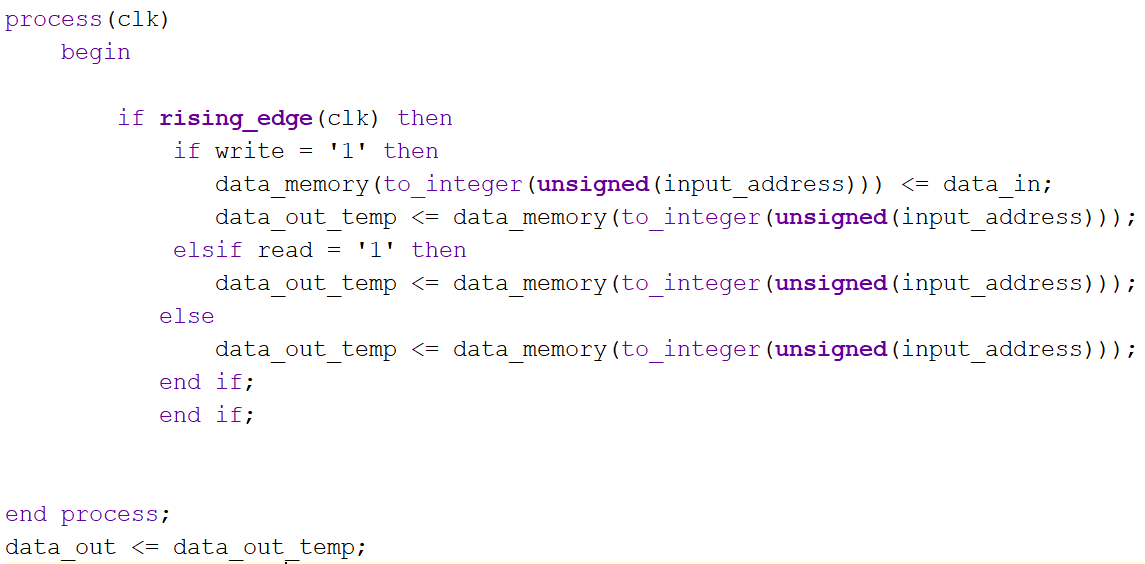
In addition, we also have a signal “data\_memory” which will hold the values returned by calling the function “fill\_mem”.

Next, we begin with a process in which I am going to process the clock (“clk”) signal.

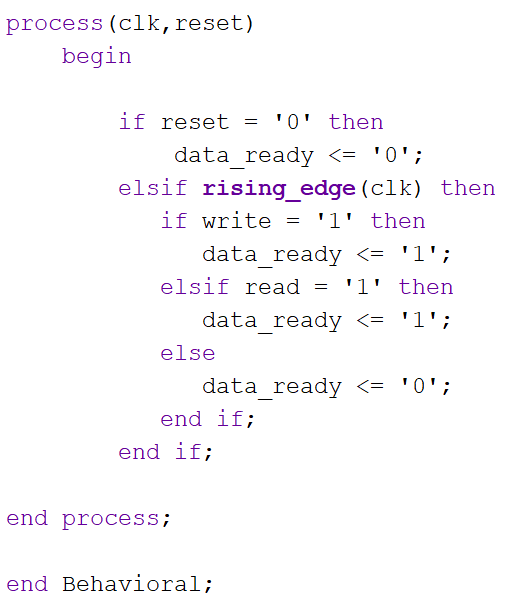
If the clock signal is equal to ‘1’ logic ( “rising\_edge (clk)“ ) and if write is also equal to ‘1’ logic ( “write = ‘1’ “, which means that we are allowed to write into the RAM’s memory ) then we will write the data which “data\_in” holds into the “data\_memory” at the address we chose ( “input\_address” ) and this is considered to be a synchronous write. After this, the signal “data\_out\_temp” will hold the value of the data we wanted to write into the RAM, which we consider it to be a synchronous read.

If read signal is equal to ‘1’ logic ( “read =’1’ “ ) then we will be able to read from the RAM, and the output data will be displayed.

If neither write, nor read signal is equal to ‘1’ logic then we will again have a synchronous read and the output data will be equal to the input data. We close both “ifs” and we end the process. After that, the output “data\_out” will hold the value of the “data\_out\_temp”.



Finally, we have a second process which involves the clock and the reset signal. The reset signal will be an active low asynchronous reset. That means when the reset signal will be equal to ‘0’ logic, the “data\_ready” signal will be equal to ‘0’ logic, meaning that “we” cannot “tell” the processor that it can read/write from/to the RAM memory. However, if the reset signal is equal to ‘1’ logic and the clock signal is equal to ‘1’ logic and also the write signal is equal to ‘1’ logic, that means the data is written and we can acknowledge the processor. If instead of the write signal, the read signal is equal to ‘1’ logic, that means that the data is ready to be read and we can acknowledge the processor. Any other combination will not activate the “data\_ready” signal and it will remain ‘0’ logic. The process is ended and so is the architecture.



Next, I will implement a synchronous read/write multi-banked interleaved main memory system which consists of four memory banks of type RAM (implemented earlier) of 1 kB each. So we will have a main memory of 4 kB.

128

clk data\_out

reset data\_ready

Main\_

Memory

rd

wr

10

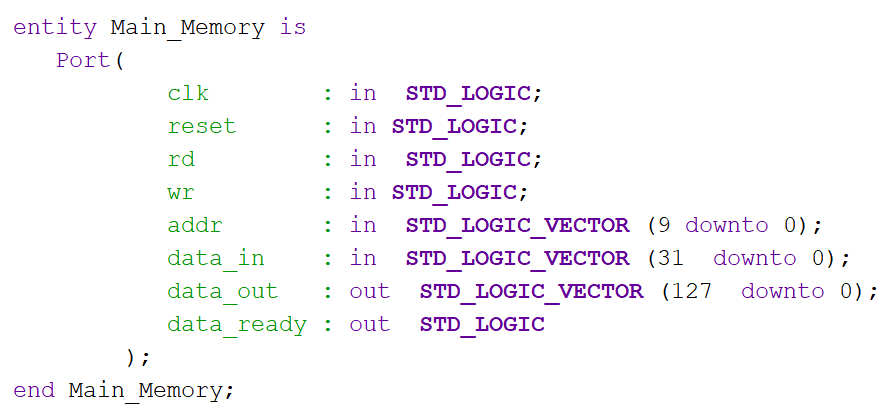
addr

32

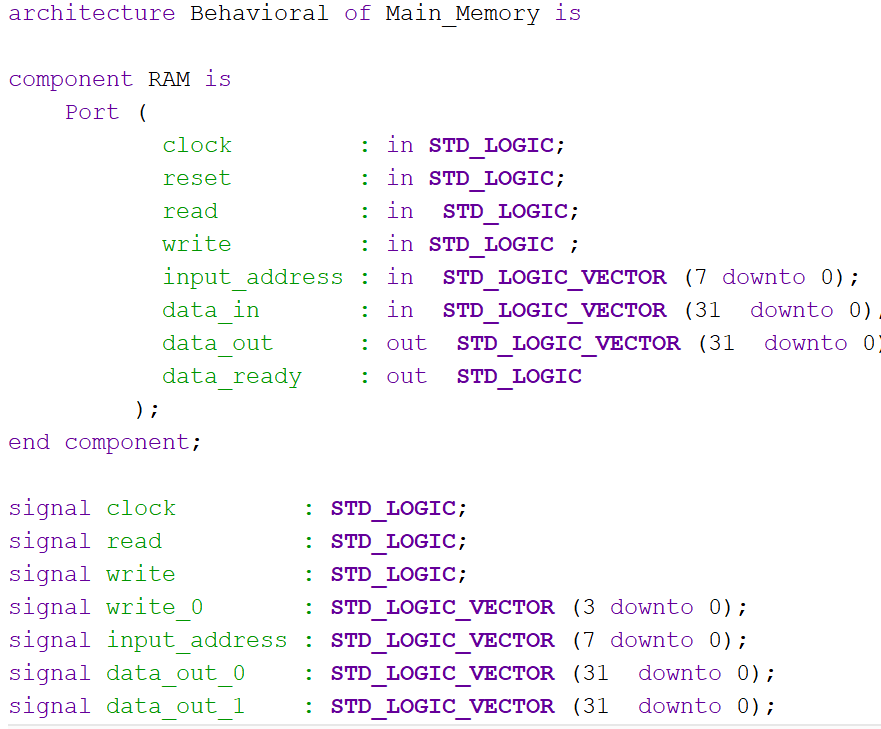
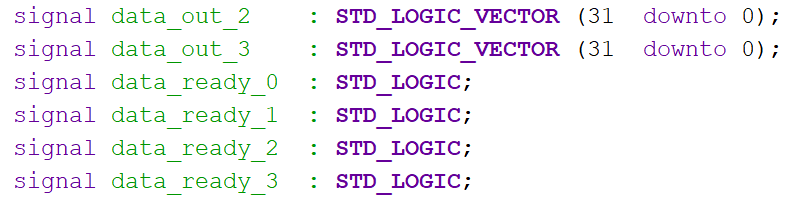
data\_in

* “clk” is the clock signal;
* “reset” is the reset signal;
* “rd” is the signal which tells us that the data is going to be read from the main memory
* “wr” is the signal which tells us that the data is going to be written in the main memory
* “addr” is an address on 10 bits which consists of tag(4 bits) + index(4 bits) + offset(2 bits); the tag and the index specify the exact position of where in the cache should be the data written and the offset selects the memory bank (one of the four RAMs) in which the data is going to be read;
* “data\_in” is the data that is going to be written in the main memory;
* “data\_out” lets us acknowledge the end of the data processing;

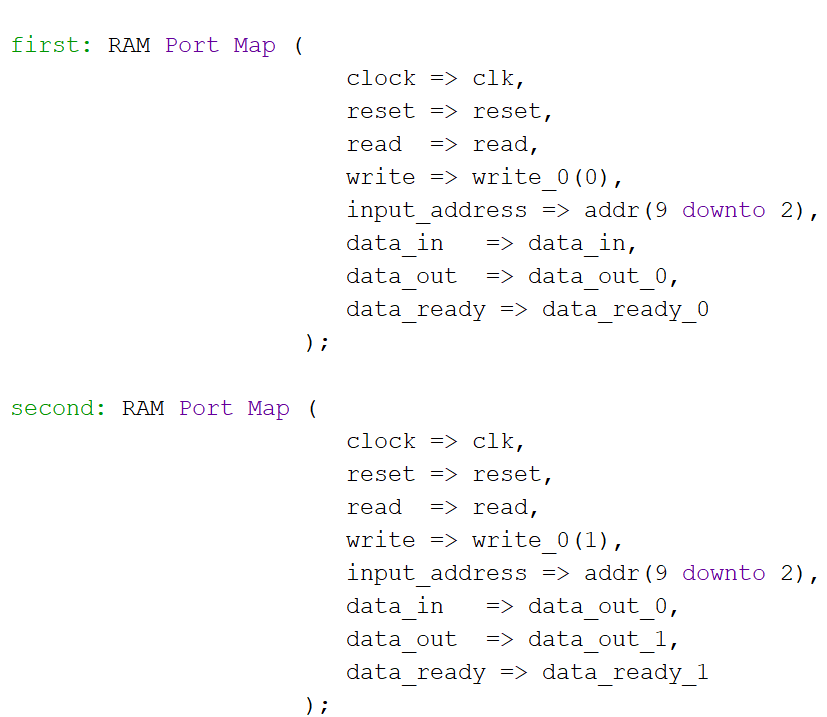
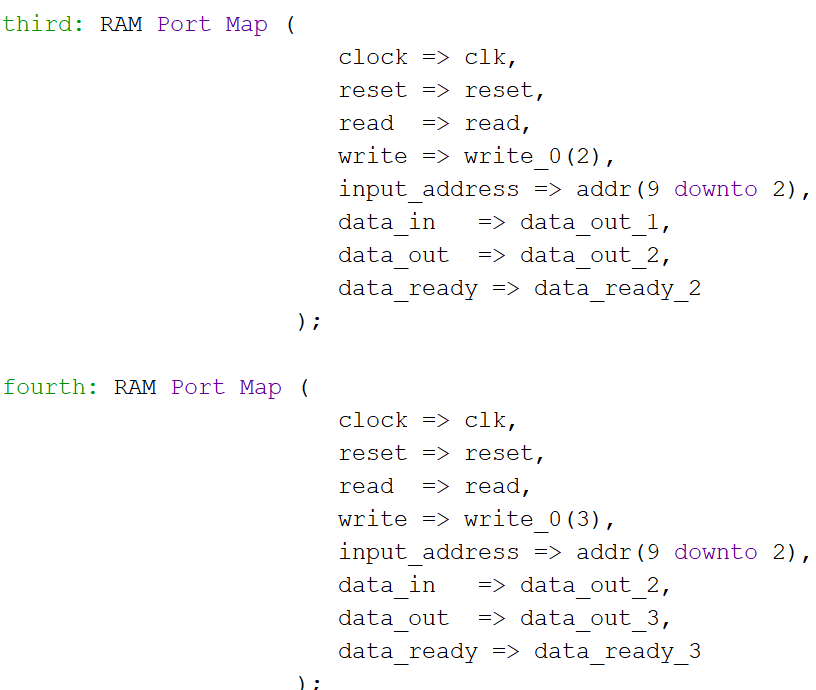
First, I declared the Main\_Memory entity which contains all the input and output signals presented before:



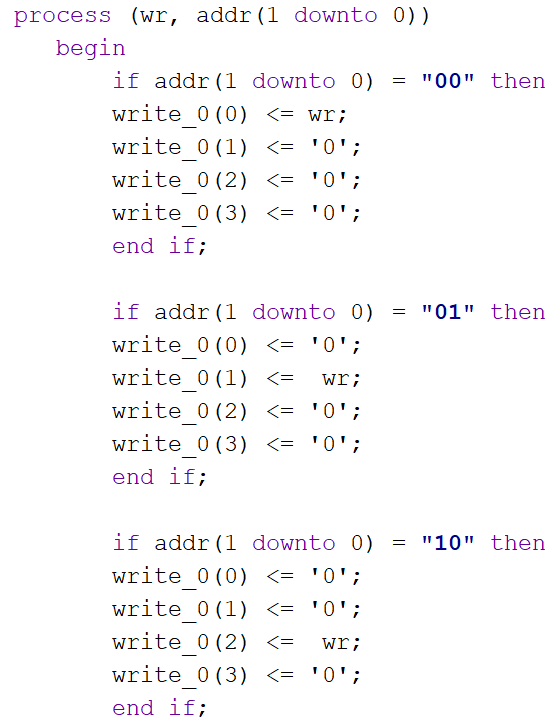
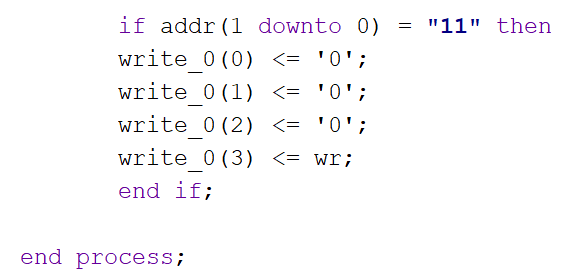
Next, I declared some signals that will help me later in the processes, plus the RAM component:



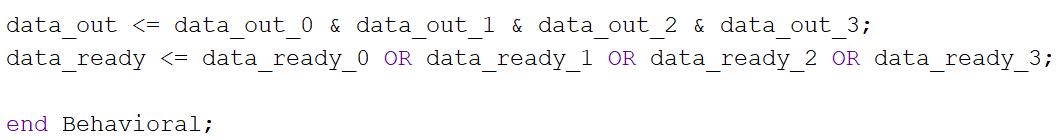
After doing this, I started to link the four RAM banks together with the help of the signals declared above to form a single main memory:



After this, I created a process which consists of the “wr” signal and the last two bits of the “addr” signal with the help of which we can choose in which one of the four memory banks to write data through the “data\_in” signal. I also transmit the value of the “wr” signal to the corresponding write of each memory bank:



Finally, I assign the values for the output signals “data\_out” and “data\_ready”. The “data\_ready” signal will be an “OR” between all the “data\_ready\_0” array, of each memory bank because if one data has been written in one of the memory banks that means that the data is ready. The “data\_out” signal will be an “AND” between all the outputs of each memory bank. In this way we are combining words from the memory banks:



Next, I am going to implement the cache memory, memory in which the data from the main memory is copied and requested by the CPU.

Cache\_Memory

Clock

32

Refill Read\_data

Update

4

Index

4

Offset

128

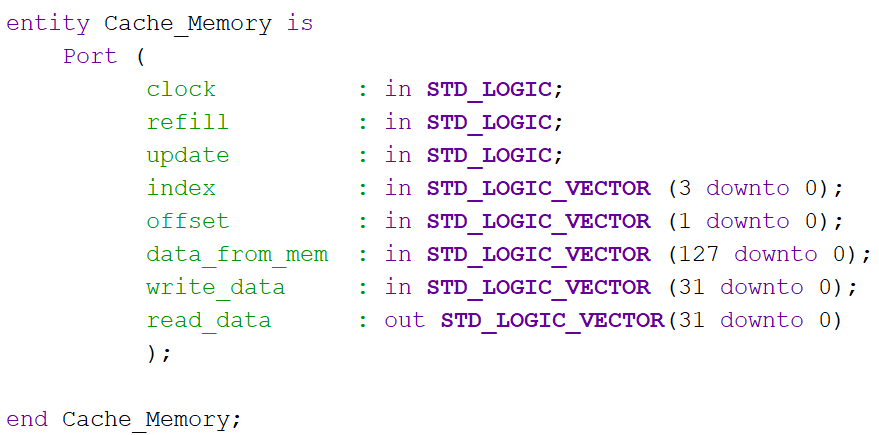
Data\_from\_mem

32

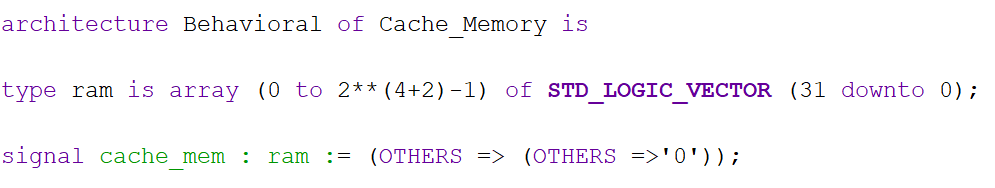
Write\_data

* “clock” is the clock signal;
* “refill” signal is for when a miss occurs (refill = ‘1’) and for that we will refill the old data in the cache with new data from memory;
* “update” signal is for when a hit occurs (update = ‘1’); it updates the cache using data from processor;
* “index” signal represents the index of each cache line; it is used for selecting the index from cache;
* “offset” signal represents the offset selection;
* “data\_from\_mem” signal represents the data from the main memory;
* “write\_data” signal represents the data which is going to be written from the processor;
* “read\_data” signal represents the data that has been read and it will be transmitted to the processor;

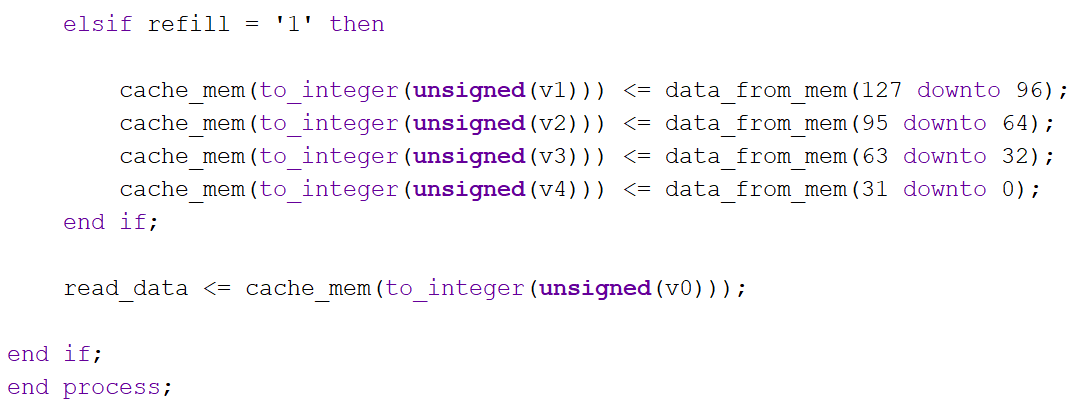
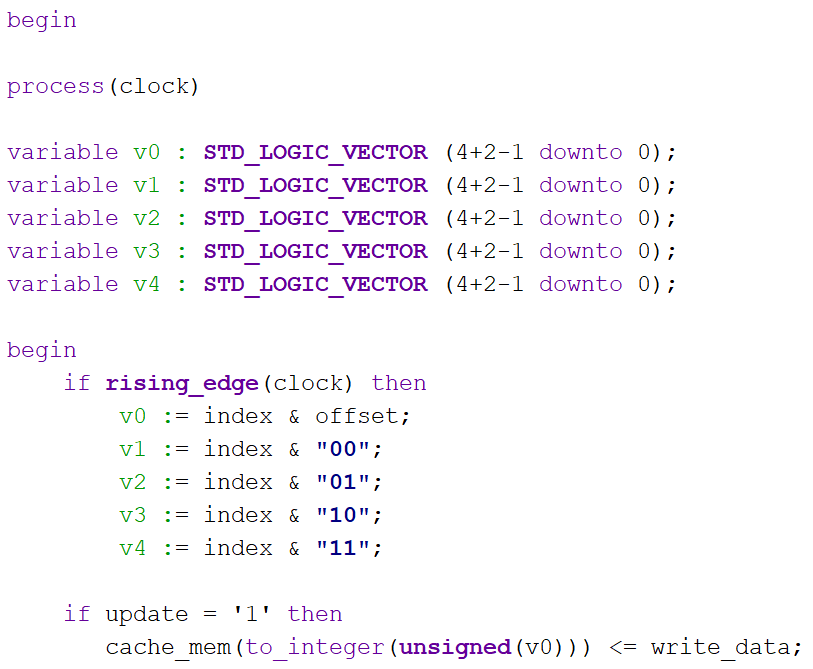
First, I declared the Cache\_Memory entity which contains all the input and output signals presented before:



Next, I declared some a signal of type ram that will help me later and initialized it with zero:



After doing this, I declared four variables, each one on six bits which will help me later to determine the locations from the cache for updating/refilling certain data:



As we can see above, the first variable “v0” is composed by the index and offset. This is because a hit occurred, and we need to update the cache data and for that we need to go to a certain index and offset in the cache memory. The other variables “v1”,”v2”,”v3”,”v4” are composed of index and 2 bits which represent the bank memory in which the data needs to be refilled. For “v1” we have “00” which means that we extract the data from main memory from the first bank memory which contains the memory blocks from 127 to 96 (32). For “v2” we have “01” which means that we extract the data from main memory from the first bank memory which contains the memory blocks from 95 to 64 (32). For “v3” we have “10” which means that we extract the data from main memory from the first bank memory which contains the memory blocks from 63 to 32 (32). For “v4” we have “11” which means that we extract the data from main memory from the first bank memory which contains the memory blocks from 31 to 0 (32).

The “read\_data” signal means that the data from cache is read, and that is always available.

Next, I am going to implement the cache controller, which is in charge of controlling when to write to cache or when to read from the cache or main memory. It is a state machine with 6 states: Idle, Write\_Miss\_Hit, Read\_Miss, Refill\_State, Stall\_State and Wait\_State.

clock stall

code refill

reset update

**Cache\_**

**Controller**

enable read\_from\_mem

4

tag write\_to\_mem

4

index

read

write

ready

flush

* „clock” signal is the global clock
* „code” signal is for choosing if hit or miss
* „reset” is for resetting to the idle state
* „enable” is for goind to the current state to the nex tone when a button is pressed
* „tag” is the signal for the tag
* „index” is the signal for de index
* „read” is the read signal
* „write” is the write signal
* „ready” is ready signal from main memory
* „flush” is signal for invalidating the cache line
* „stall” signal is going to be on a led as output
* „refill” signal is for refilling the cache lines
* „update” signal is for updating the cache
* „read\_from\_mem„ signal is going to be on a led as output
* „write\_to\_mem” signal is going to be on a led as output

Code=’0’ or ‘1’

Code=’0’

Write\_

Miss\_Hit

Read\_Miss

Idle

Refill

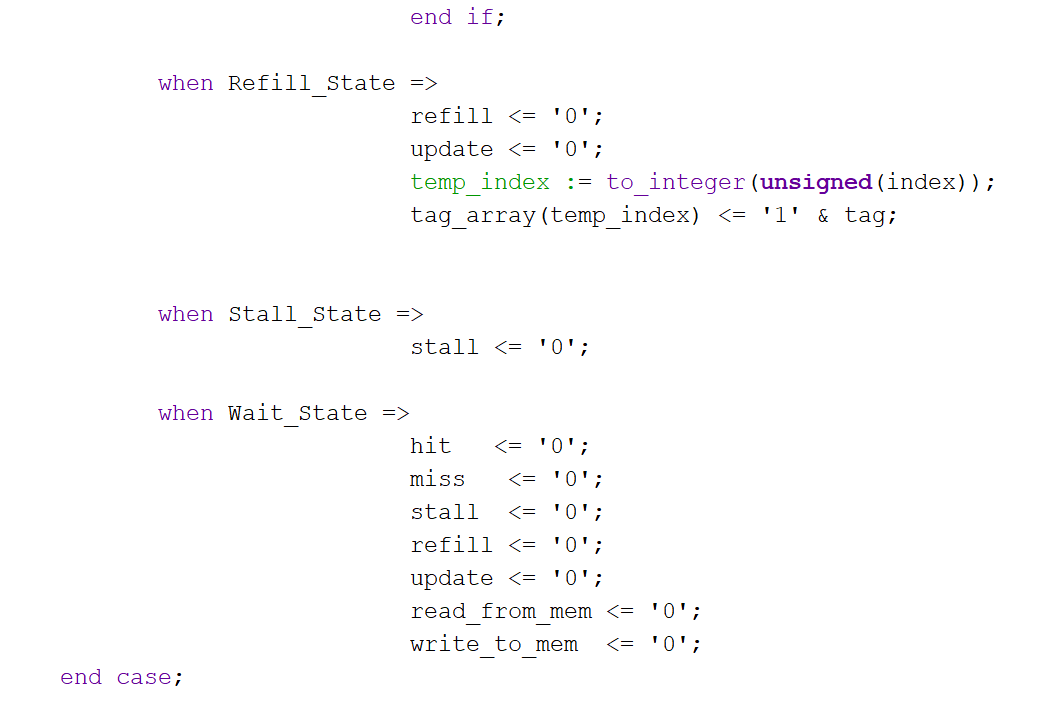
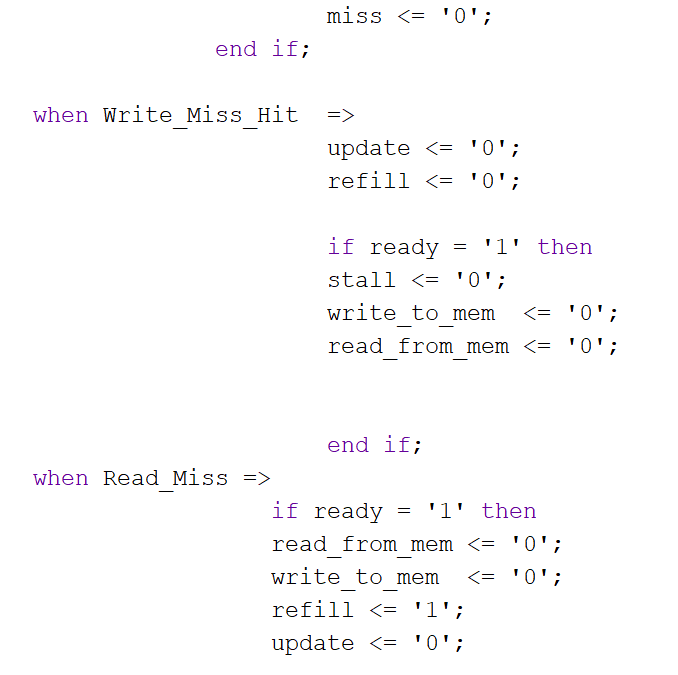
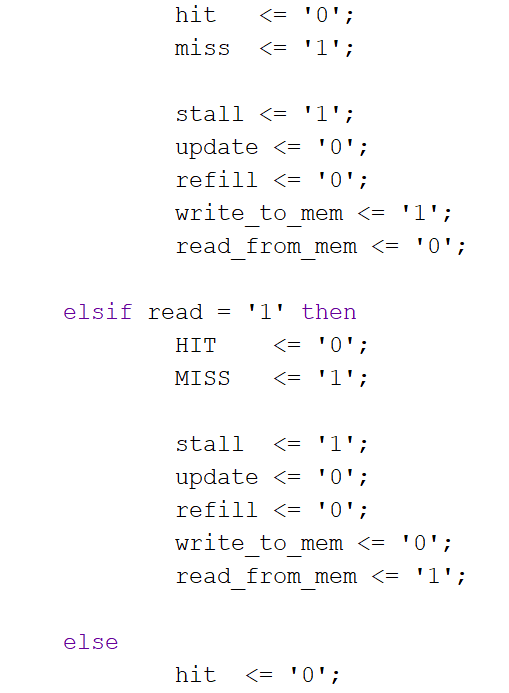
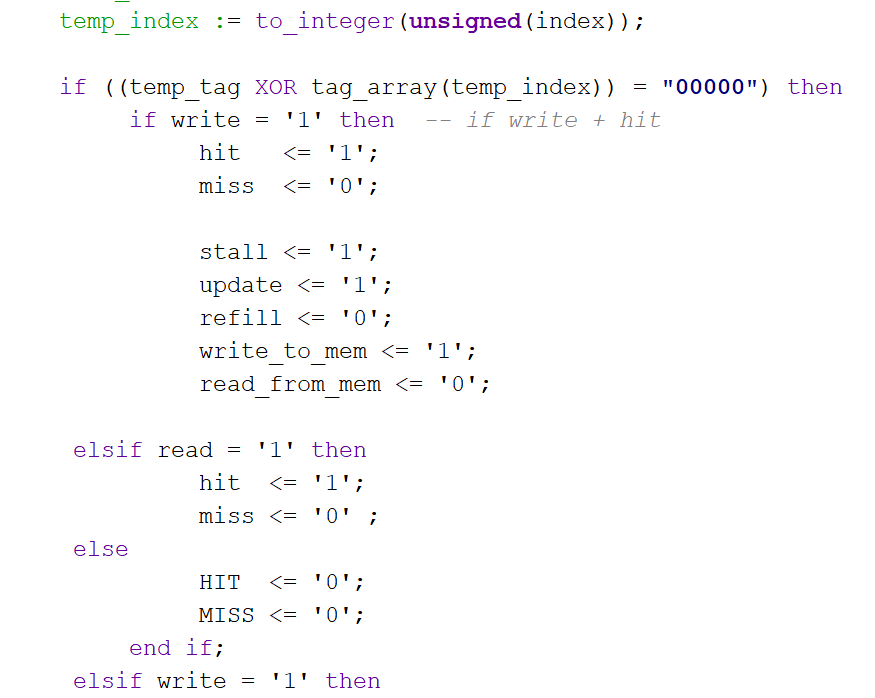
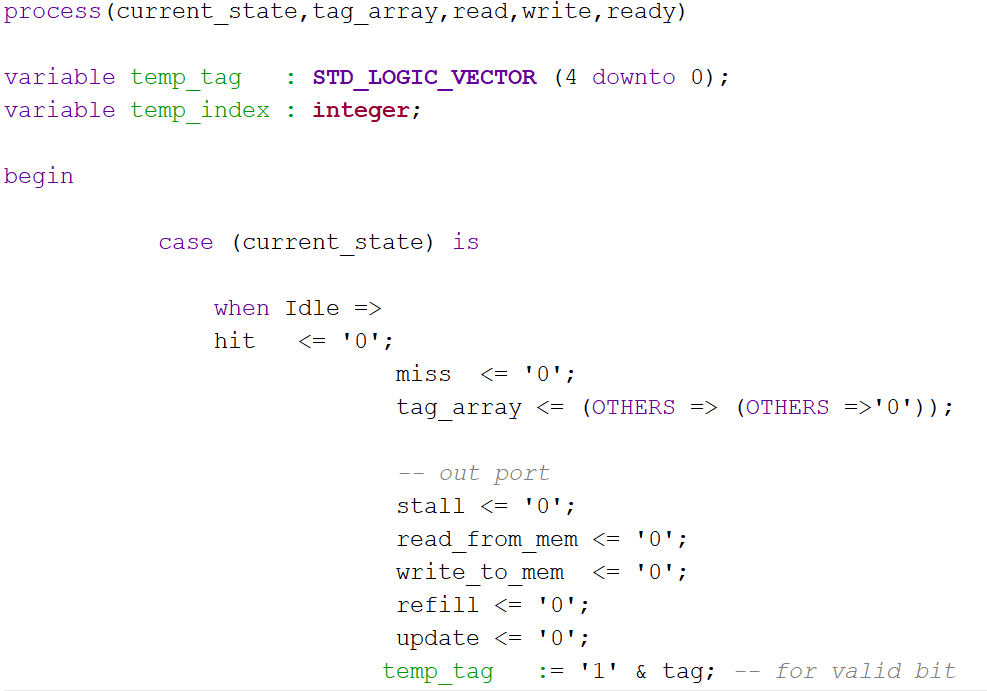
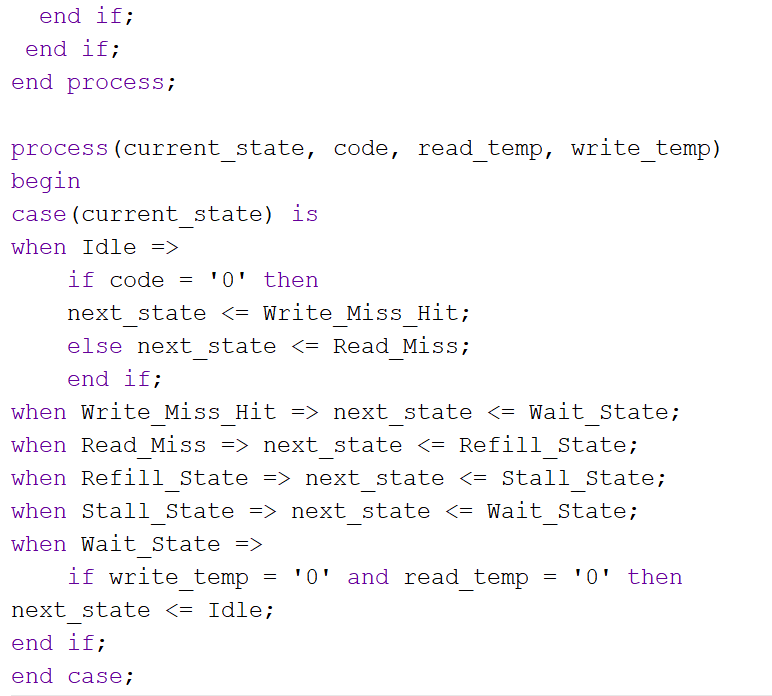
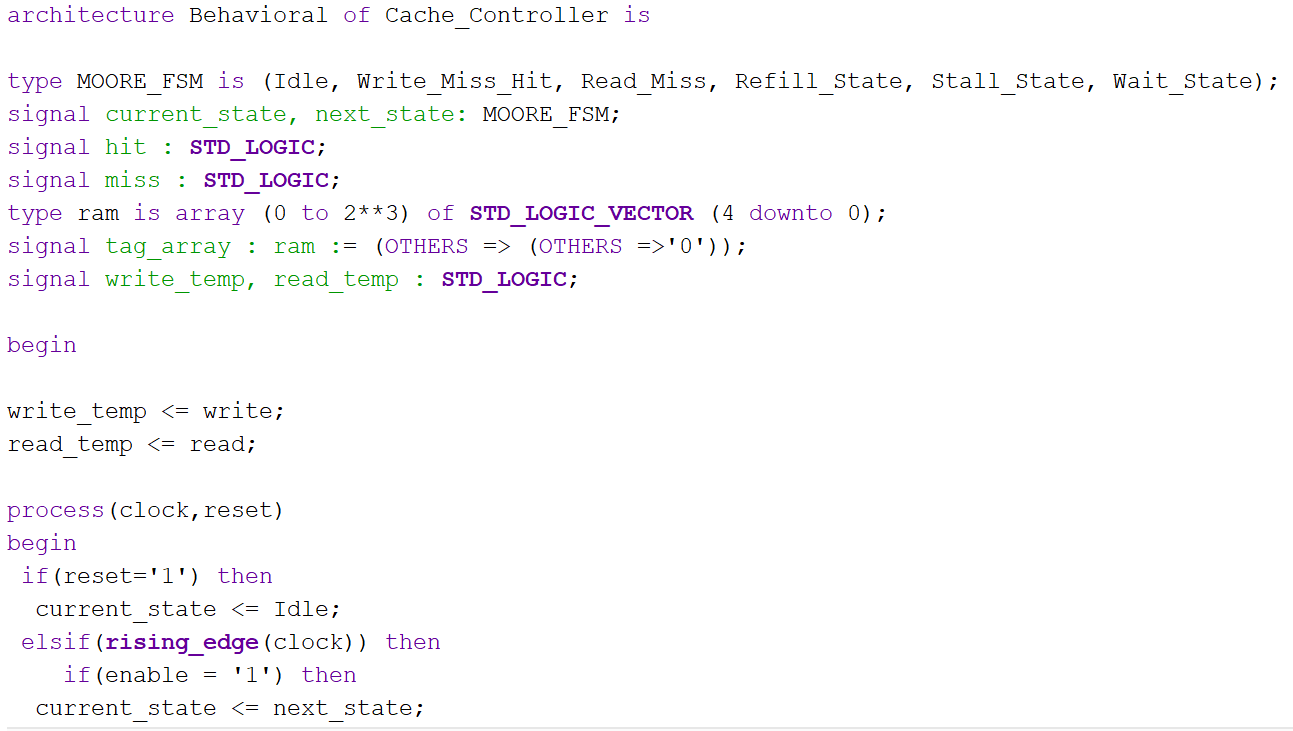
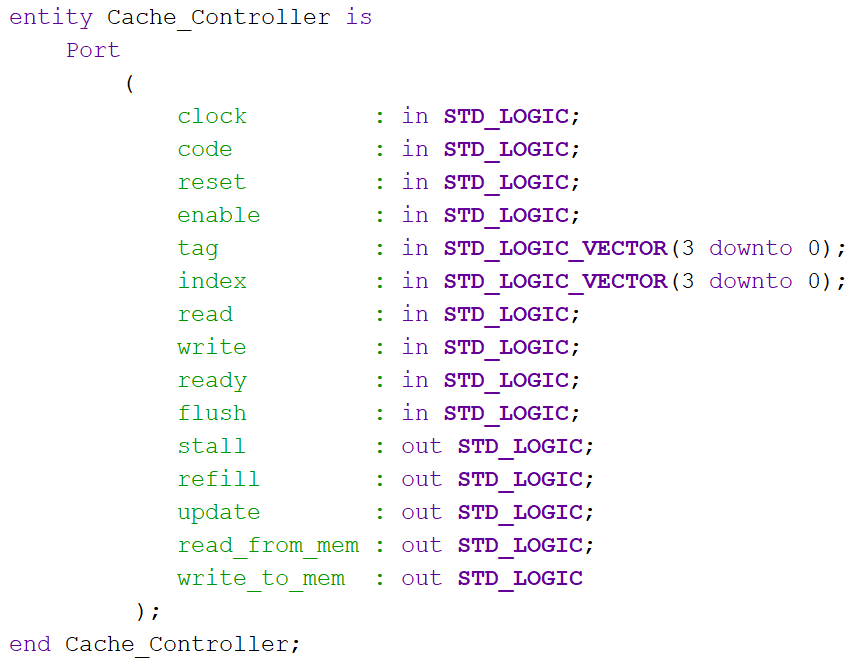
Wait\_

State

Stall\_

State

In the Idle state, all the internal and output signals are set to 0. In the same state, we verify if the valid bit is set to ‘1’ logic and if the tags match. If they do, we can obtain a write miss or a write hit or read miss or read hit. If we obtain a write hit we stall the machine because we need to access the main memory to show our initial data and then we update the cache memory and go to the next state which is Write\_Miss\_Hit. There, we wait until the „ready” signal is 1 which means that the data has been written to the memory and on the output we will have the new data written to that specific address. The next state is a global Wait\_State where if the signals read and write have the value 0, we go to the Idle state. If we have a write miss then we stall the machine, access the main memory and write the data given as input. After this, w ego to the Write\_Hit\_Miss state and then to the Wat\_State and back to the Idle state. If we have a read hit then we just read the data from the cache and display it and do nothing. If we have a read miss then we stall the machine because we need to access the main memory using the signal „read\_from\_mem” and then we got o the Read\_Miss state. Here we refill the cache line with the data from the main memory and then we go to the Refilll\_State. Here the data will be displayed on the output and w ego to the Stall\_State where the „stall” signal will become 0 and then to the global Wait\_State and back to the Idle state.



Next, I am going to implement the top module which contains all the components described above. They are connected and form the management unit for the cache memory control. Also, the signals for the led are set and the signals which will use the switches as input are set.

16

clock led

7

5

btn cat

4

16

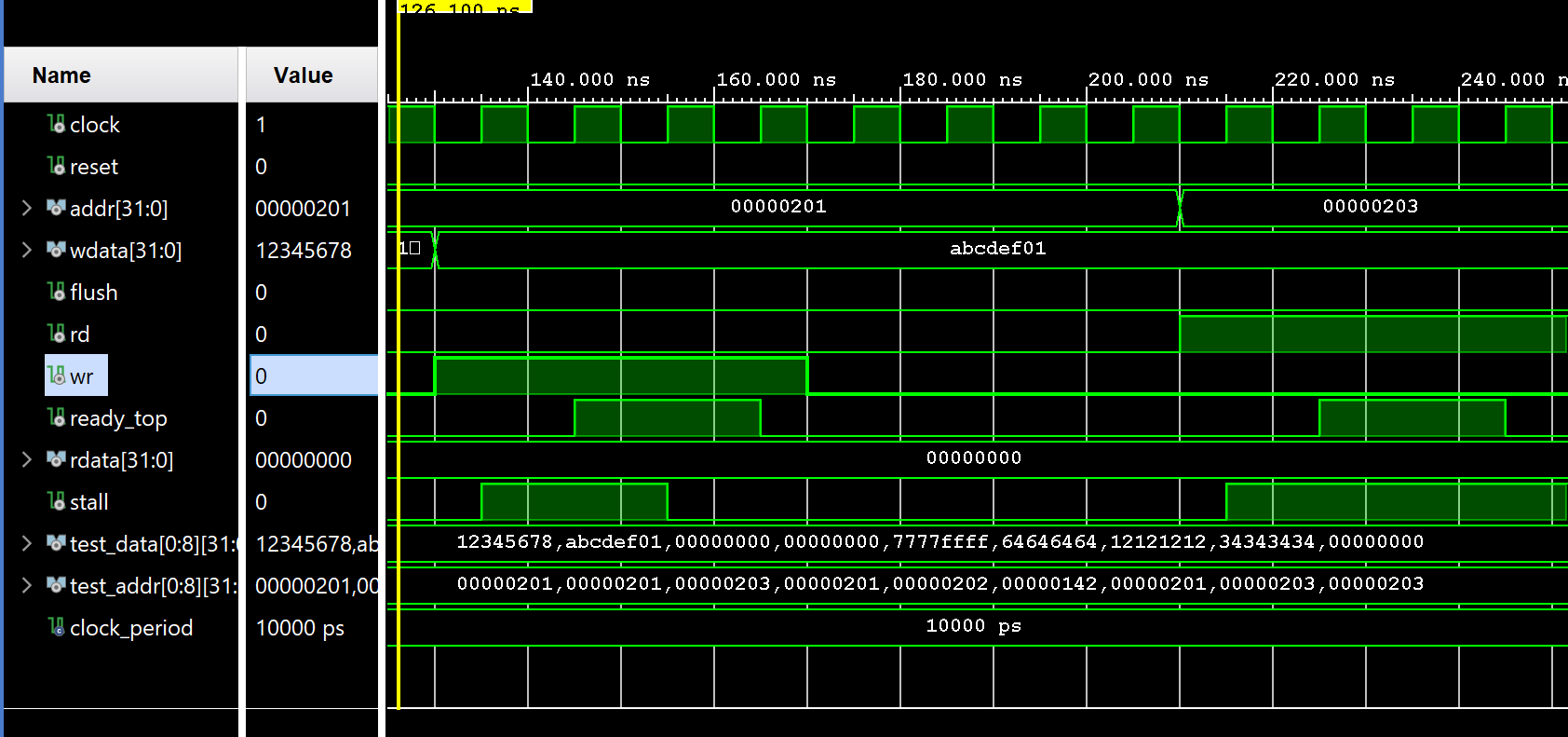
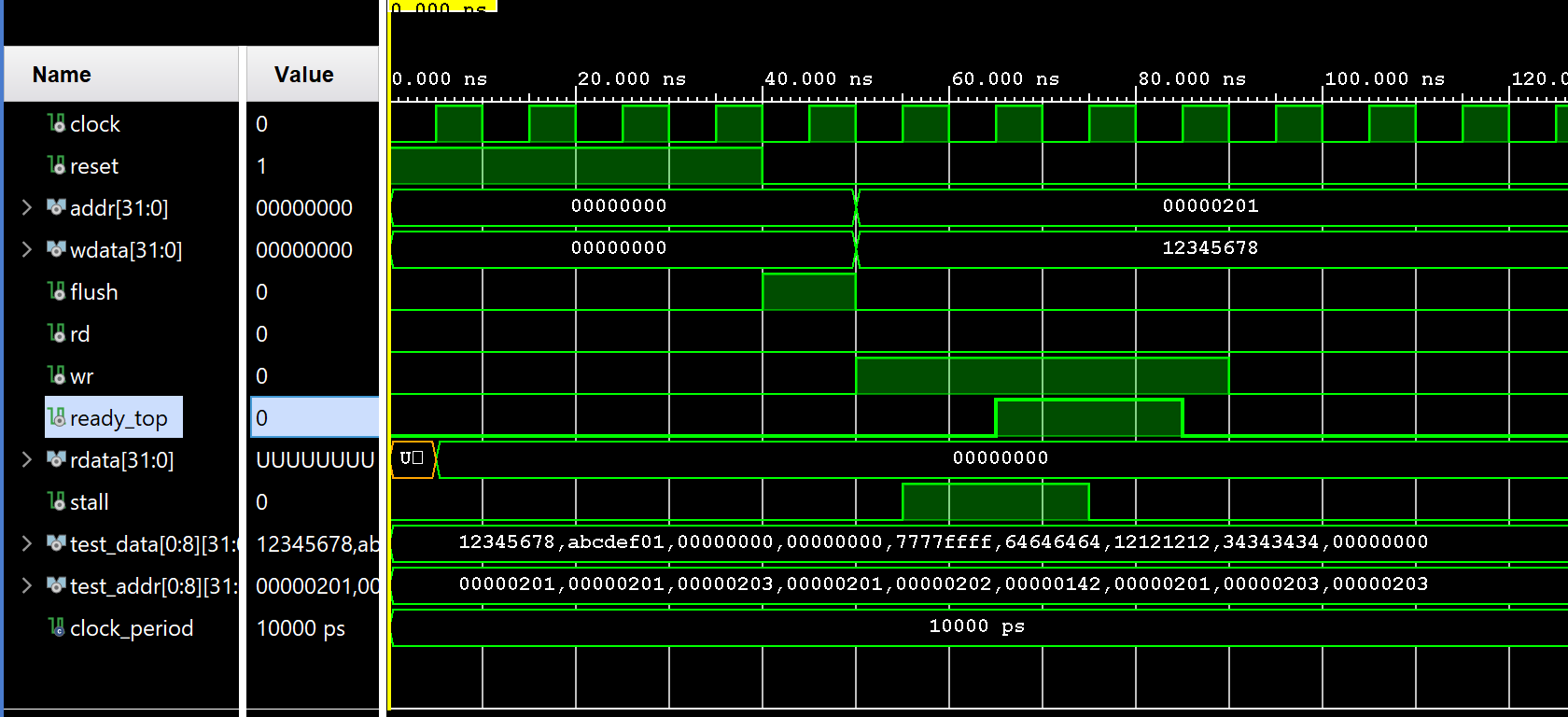
TOP

sw an

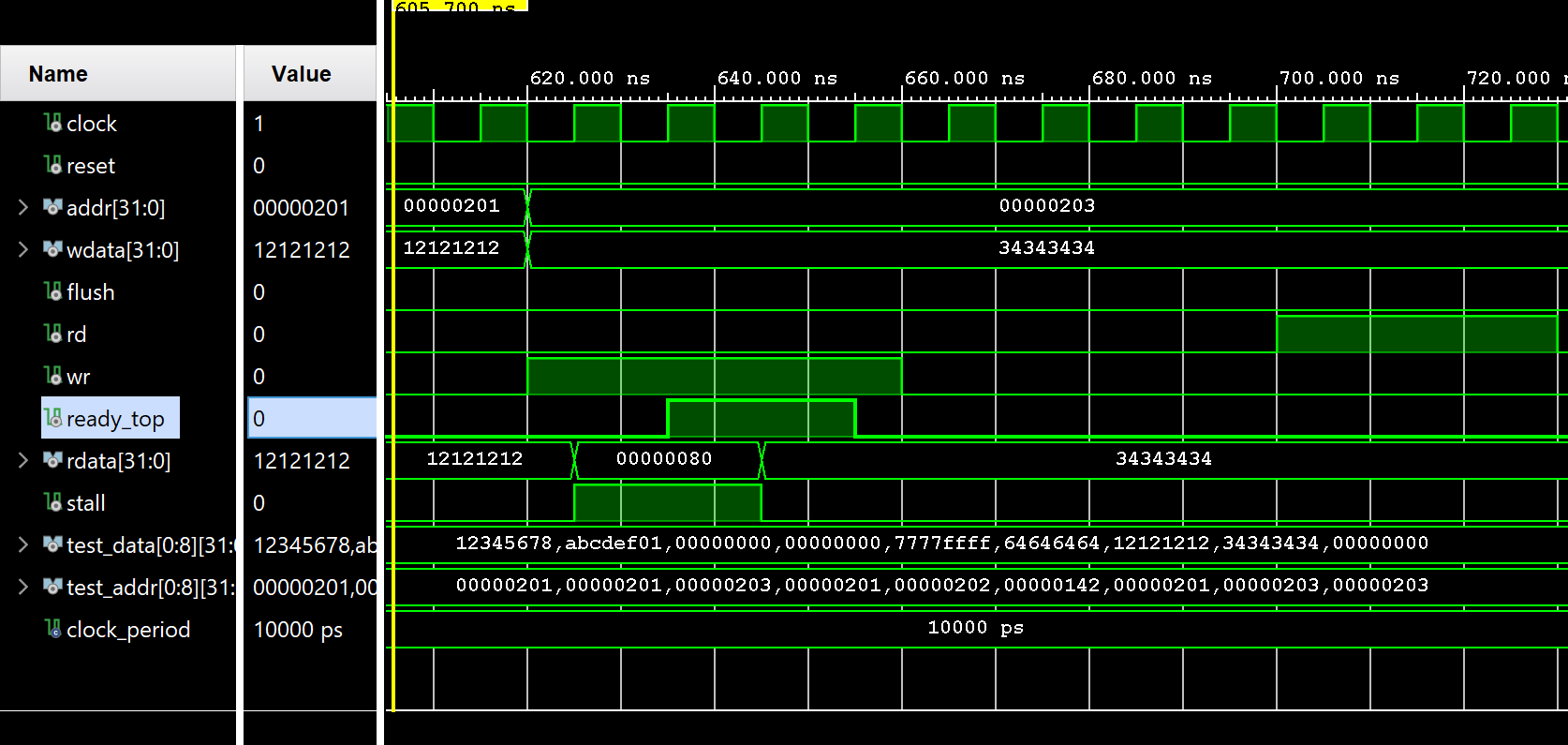
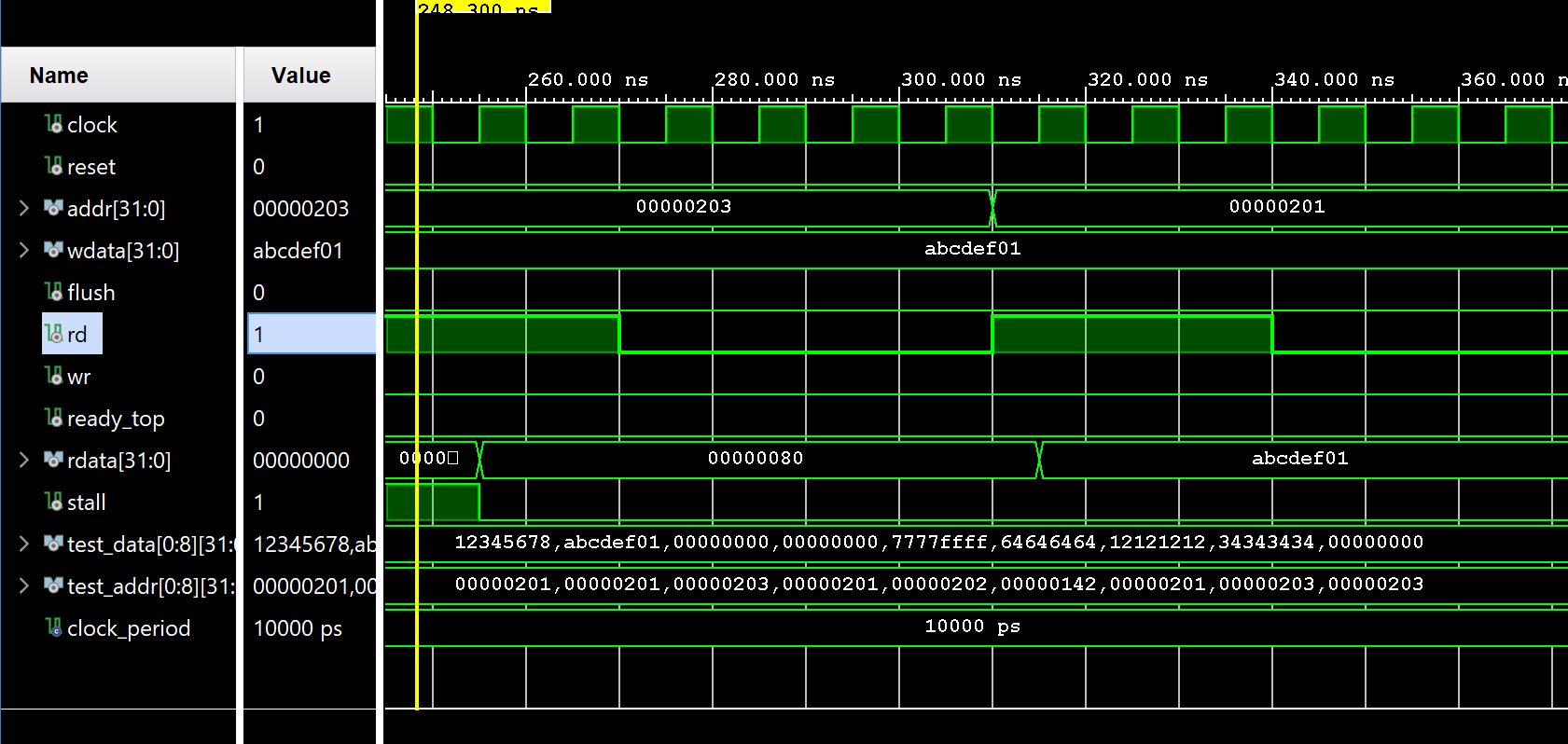
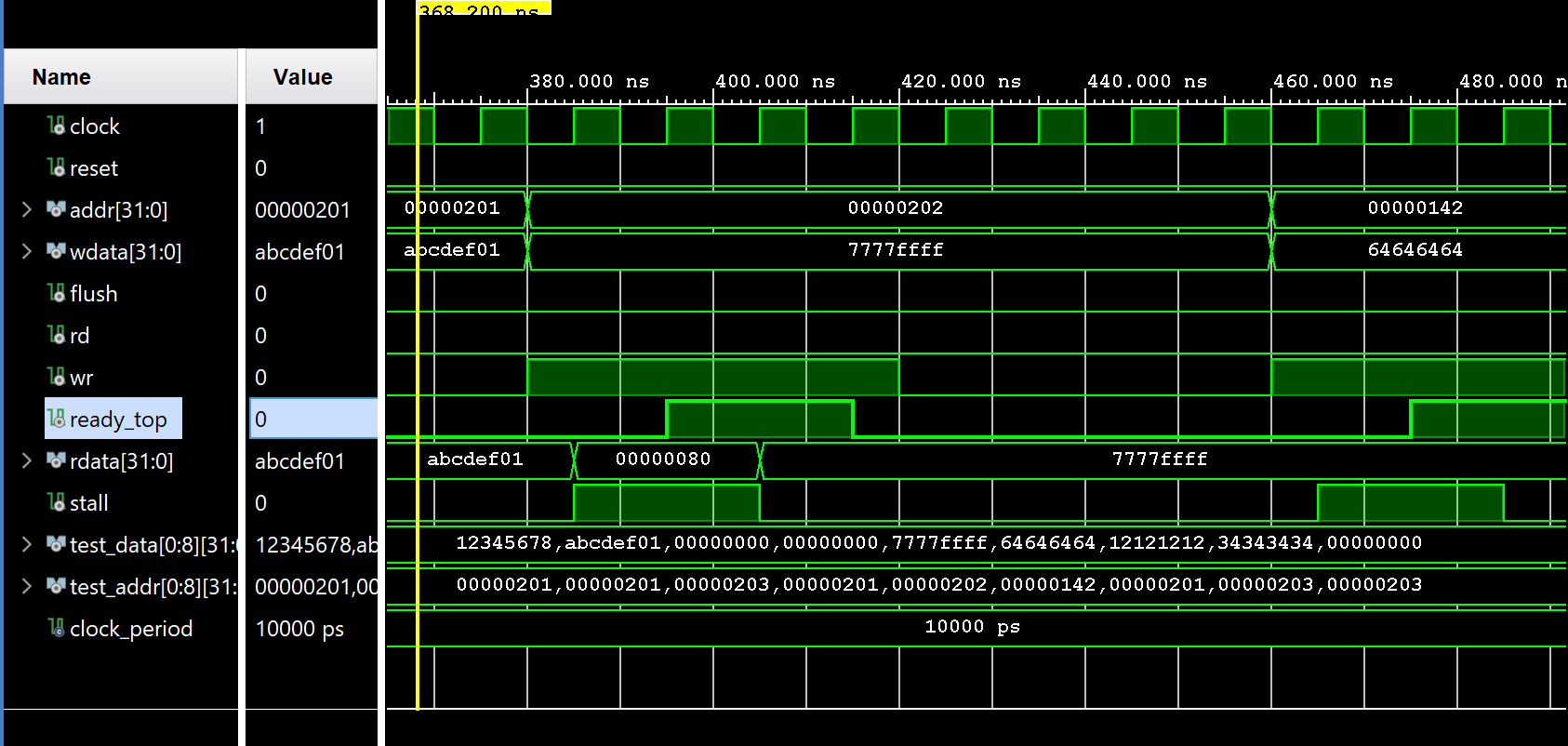
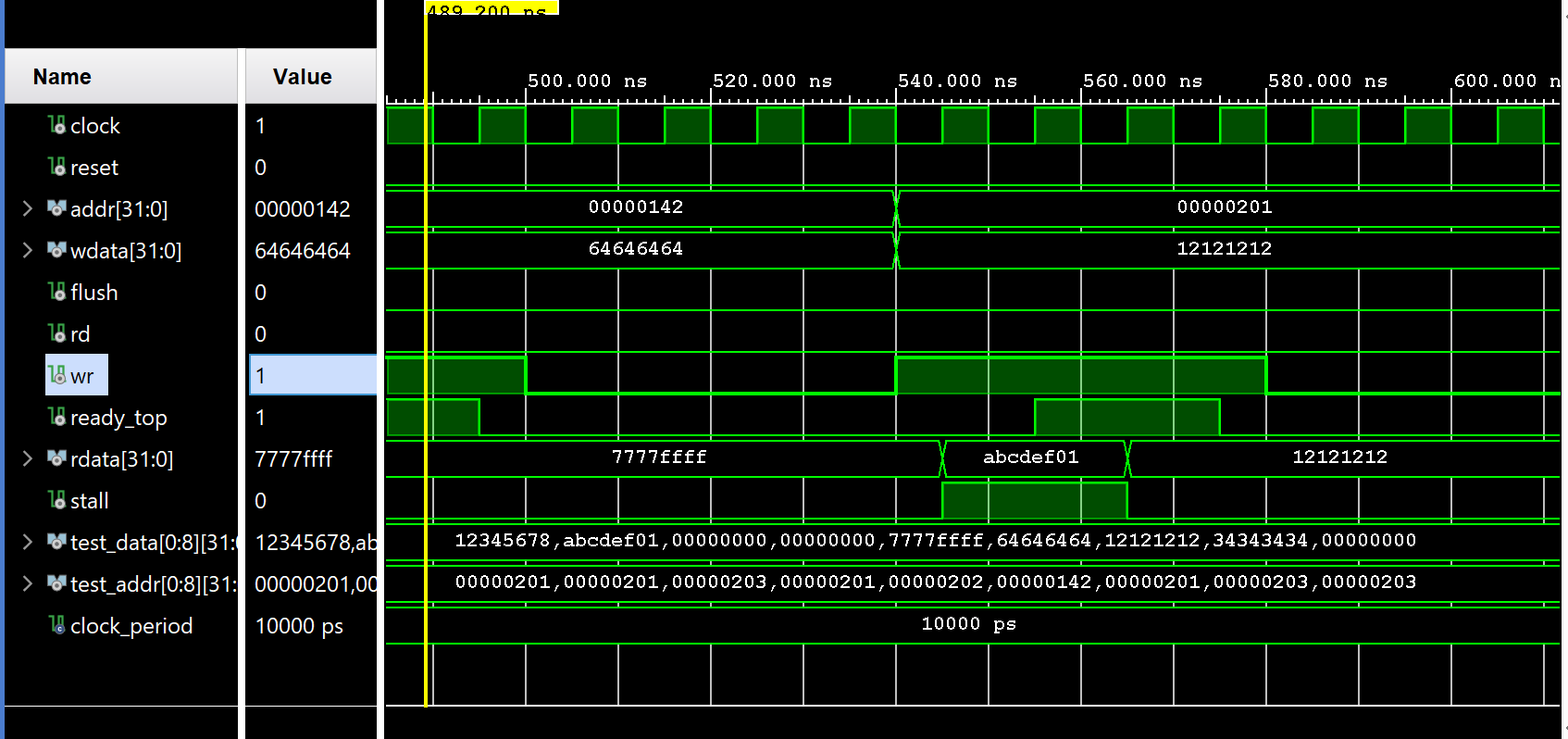
Also, the Top module also contains a monopulse generator and a SSD which will display the read address from the cache memory. It will display the last 16 bits is sw(15) = ‘0’ and it will display the first 16 bits if sw(15)= ‘1’.

* “clock” is the global clock signal
* “btn” is the button that will be pressed in order to go from the current state to the next one
* “sw” is the input signal for the switches. Sw(0) represents the “reset” signal, sw(1) represents the “read” signal, sw(2) represents the “write” signal, sw(3) represents the “flush” signal, s(4) represents the “code” signal, sw(14 downto 5) are used for introducing a specific address. All of the mentioned signals are active high.
* “led” is the output for the leds. Led(0) represents the “stall signal”, led(1) represents the “enable” signal, led(2) represents the “update\_cache” signal, led(3) represents the “refill\_cache” signal, led(4) represents the “read\_from\_mem” signal, led(5) represents the “write\_to\_mem” signal, led(6) represents the “data\_ready” signal.

Testing



In these two pictures from above we can see the example of what happens when a write miss occurs by looking at how the signals respond.



In the pictures above we can see that our cache controller is working, by testing all the possible cases : write miss, read miss, read hit and write hit.

Conclusions

A cache controller has been designed for a direct-mapped cache and it has been found to work successfully with the given inputs. In this little project we learned the idea of how our processor inside our PC/laptop works with the data. In the real life, these operations are done extremely fast.

References

* <https://searchstorage.techtarget.com/definition/cache-memory>
* <https://en.wikipedia.org/wiki/Memory_management_unit>
* <https://docs.magento.com/m2/ce/user_guide/system/cache-management.html>
* <https://en.wikipedia.org/wiki/Random-access_memory>
* <https://en.wikipedia.org/wiki/Central_processing_unit>
* <https://www.techopedia.com/definition/2851/central-processing-unit-cpu>
* <https://en.wikipedia.org/wiki/CPU_cache#Implementation>
* <https://www.hardwaresecrets.com/how-the-cache-memory-works/>
* <https://www.extremetech.com/extreme/188776-how-l1-and-l2-cpu-caches-work-and-why-theyre-an-essential-part-of-modern-chips>
* <https://courses.cs.washington.edu/courses/cse378/09wi/lectures/lec15.pdf>