MINISTERUL EDUCAȚIEI NAȚIONALE

UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

FACULTATEA DE AUTOMATICĂ ŞI CALCULATOARE

DEPARTAMENTUL CALCULATOARE

# Control Unit for a Cache Memory

LUCRARE DE LABORATOR

Student: **Andra Cristina SIGHIARTAU**

**2019**

# Table of Contents

# 1.Introduction

A cache memory, also called a CPU memory, is a high-speed SRAM (static random access memory that can be accessed more quickly by the computer than a regular RAM (random access memory). It is generally integrated directly into the CPU chip (Level 1 cache) or it is interconnected with it through a bus (Level 2 cache). A cache memory's purpose is to store data or program instructions which are used repeatedly by programs or the CPU. By accessing the cache, the computer processor does not have to get data from the computer's main memory, which saves a considerable amount of time. Thus, a cache memory is responsible for speeding up computer operations and processing.

Most programs use only a few of a computer's resources after having been opened and operated for a long time because the frequently used instructions are cached.

The history of the cache memory started as early as the 1980s. Mainframes used an early version of cache memories, but once early PCs started to appear, processor performance started to increase much faster than memory performance, so memory became a bottleneck, slowing systems down. The first cache memory for a PC was external to the CPU, placed on a separate chip along with its memory cache controller. The amount of available memory cache varied depending on the motherboard model of the PC and typical values for that time were between 16KB to 128KB.

With 486 processors, Intel added 8KB of memory to the CPU as a Level 1 memory. Level 2 memories went up to 256KB until Pentium doubled the amount to 512KB and split the internal cache memory into two parts: one for instructions and another one for data. These external memories were accessed at a much slower clock speed than the rate at which the processor ran, slowing the performance of the system.

In 1995 Intel introduced the P6 architecture, which incorporated a Level 2 cache memory into the CPU. This meant that the cache memory ran at the same clock speed as the processor, increasing performance considerably. This architecture is used to this day: both Level 1 and Level 2 cache memories are located inside the CPU, running at the CPU internal clock rate. [1]

# 2.Objectives

The objective of this project is to simulate a Level 2 cache memory control unit whose role is to send information to the cache memory and check if the desired information is stored in the cache and can be retrieved quickly or if it has to be retrieved from the main memory.

The control unit will be programmed in VHDL using the Vivado ISE provided by Xilinx. The simulation will be performed using a Basys3 board.

The cache memory will be pre-filled with instructions and input data will be provided through switches on the board.

For a read request, if there is a cache hit, the data and tag will be showed on the display. If a cache miss occurs, the block to be placed in the cache is displayed on the board. Leds will be used to signal a cache hit or miss.

For a write request, data read from the switched will be displayed on the board.

# 3.Theory section

A cache memory is used to reduce the average memory access time. There are multiple steps involved in accessing data from the cache memory:

1. A request is made by the CPU
2. The cache memory is checked for data
3. If data is found in the cache, it is returned to the CPU. This is called a cache hit.
4. If data is not found in the cache, the data will be returned from the main memory. This is called a cache miss. When a cache miss occurs, the memory allocates a new entry and copies in data from the main memory, and then the request is fulfilled from the contents of the cache.[2]

The performance of a cache memory is frequently measured in terms of hit ratios. The formula through which the hit ratio is computed is the following:

Hit ratio = hit / (hit + miss) = number of hits/total accesses [3]

When a system writes data to cache, it must at some point write that data to the backing store as well. The timing of this write is controlled by what is known as the write policy. There are two basic writing approaches:

- Write-through: write is done synchronously both to the cache and to the backing store.
- Write-back (also called write-behind): initially, writing is done only to the cache. The write to the backing store is postponed until the modified content is about to be replaced by another cache block.

Since no data is returned to the requester on write operations, a decision needs to be made on write misses, whether or not data would be loaded into the cache. This is defined by these two approaches:

- Write allocate (also called fetch on write): data at the missed-write location is loaded to cache, followed by a write-hit operation. In this approach, write misses are similar to read misses.
- No-write allocate (also called write-no-allocate or write around): data at the missed-write location is not loaded to cache, and is written directly to the backing store. In this approach, data is loaded into the cache on read misses only.

Both write-through and write-back policies can use either of these write-miss policies, but usually they are paired in this way:

- A write-back cache uses write allocate, hoping for subsequent writes (or even reads) to the same location, which is now cached.
- A write-through cache uses no-write allocate. Here, subsequent writes have no advantage, as they still need to be written directly to the backing store.[4]

The replacement policy decides where in the cache a copy of a particular entry of main memory will go. If the replacement policy is free to choose any entry in the cache to hold the copy, the cache is called fully associative. At the other extreme, if each entry in main memory can go in just one place in the cache, the cache is direct mapped. [5]

The memory that is implemented for this project is a direct mapped cache. Therefore, it has one block in each set, so the number of sets in the main memory equals to the number of blocks in the cache. In other words, 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. An address in block 1 of the main memory maps to set 1 of the cache, and so forth until an address in block B-1 of main memory maps to block B-1 of the cache. There are no more blocks of the cache, so the mapping wraps around, in such a way that block B of the main memory maps to block 0 of the cache. [6]

Cache row entries usually have the following structure:

| | | |
|---|---|---|
| index | tag | data |

The data block (cache line) contains the actual data retrieved from the main memory.

The tag contains the address of the actual data fetched from the main memory.

The index is address of the data in the cache memory.

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 multiplied by the number of blocks stored in the cache.

The figure below shows the correspondence between the main memory and the cache memory. [4]

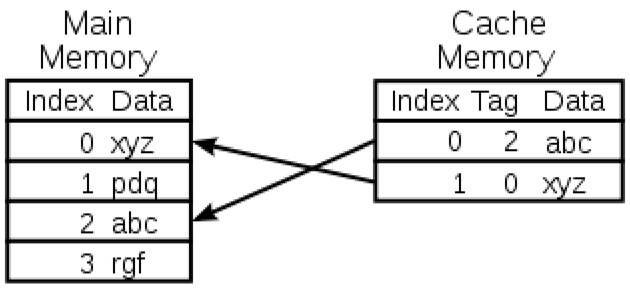


Figure 3.1 Cache to Memory Correspondence

## Placing a block in the cache [7]

The tag is determined by the index bits derived from the address of the memory block.

The memory block is placed in the set identified and the tag is stored in the tag field associated with the set.

If the cache line is previously occupied, then the new data replaces the memory block in the cache.

## Searching a word in the cache [7]

The tag is identified by the index bits of the address.

The tag bits derived from the memory block address are compared with the tag bits associated with the set. If the tag matches, then there is a cache hit and the cache block is returned to the processor. If this is not the case then there is a cache miss and the memory block is fetched from the main memory.

There are multiple advantages to a direct mapped cache memory:

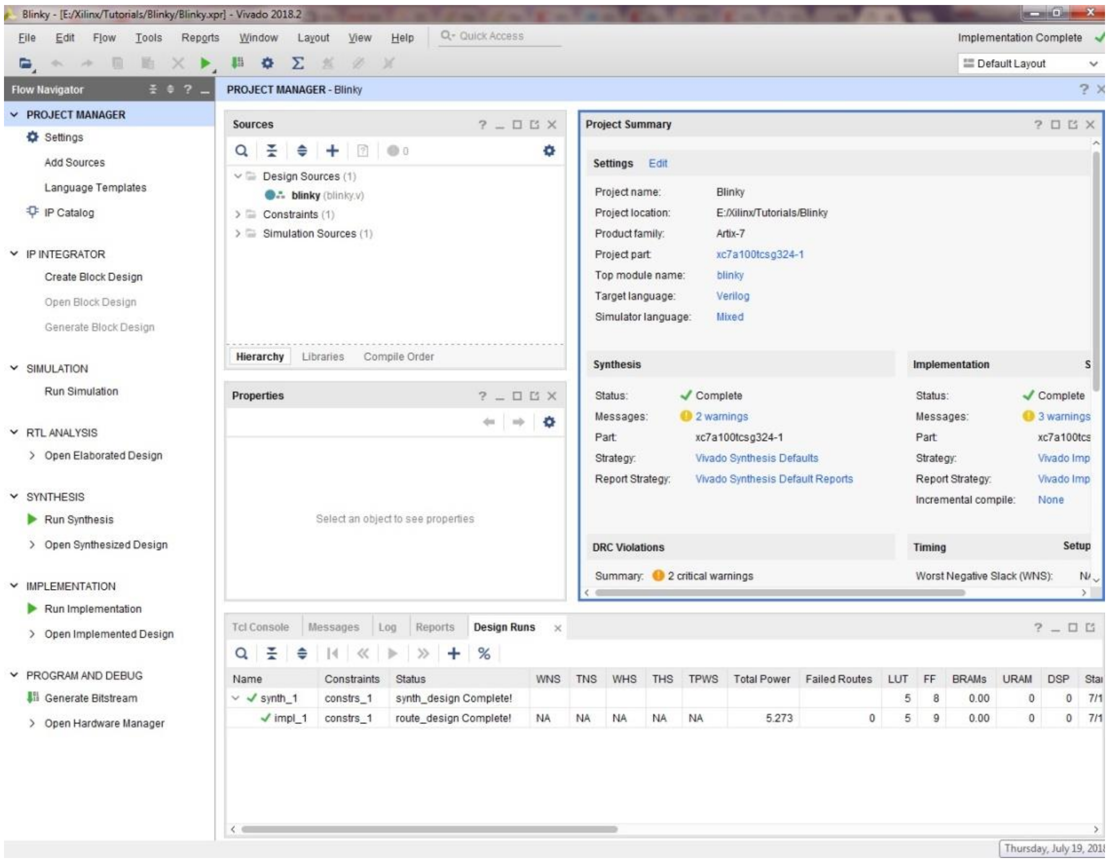
- This policy is power efficient as it avoids the search through all the cache lines.
- The placement policy and the replacement policy are simple.
- It requires cheap hardware as only one tag needs to be checked at a time. [7]

However, there is one major disadvantage: a lower cache hit rate, as there is only one cache line available in a set. Every time a new memory is referenced to the same set, the cache line is replaced, which causes conflict miss. [7]

The cache memory controller will be implemented using a Basys3 board. As mentioned in the previous chapter, the seven segment display will be used to show the contents of the memory (the data). For a read request, if the data is contained in the cache, the leds on the board will be turned on. Otherwise, they will be turned off and the display will show the data that is to be retrieved from the main memory. For a write request, data will be read from the switches and sent to both the cache and the main memories to be replaced in the appropriate block.

The switches on the board will be used to input the request sent to the cache memory.

The code for the cache memory controller is written in VHDL in the Vivado ISE. Vivado also allows for simulations in order to test the code before programming the board. The results of the simulations will be presented in chapters to follow. The Vivado interface is presented in the figure below. In order to create a project, the board specifications have to be given. The code is written across multiple files that are used as modules. Before the code is programmed on the board, Vivado performs the synthesis and implementation phases, where it checks for errors in both the code and the constraints file. If no errors occur, the bitstream file is generated. This file is programmed on the connected board.



Settings Edit

| | |
|---|---|
| Project name: | Blinky |
| Project location: | E:/Xilinx/Tutorials/Blinky |
| Product family: | Artix-7 |
| Project part: | xc7a100tcsg324-1 |
| Top module name: | blinky |
| Target language: | Verilog |
| Simulator language: | Mixed |

Synthesis

| | |
|---|---|
| Status: | ✓ Complete |
| Messages: | 2 warnings |
| Part: | xc7a100tcsg324-1 |
| Strategy: | Vivado Synthesis Defaults |
| Report Strategy: | Vivado Synthesis Default Reports |

Implementation

| | |
|---|---|
| Status: | ✓ Complete |
| Messages: | 3 warnings |
| Part: | xc7a100tcs |
| Strategy: | Vivado Imp |
| Report Strategy: | Vivado Imp |
| Incremental compile: | None |

DRC Violations

Summary: 2 critical warnings

Timing Setup

Worst Negative Slack (WNS): N/

| Name | Constraints | Status | WNS | TNS | WHS | THS | TPWS | Total Power | Failed Routes | LUT | FF | BRAMs | URAM | DSP | Sta |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ✓ synth_1 | constrs_1 | synth_design Complete! | | | | | | | | 5 | 8 | 0.00 | 0 | 0 | 7/1 |
| ✓ impl_1 | constrs_1 | route_design Complete! | NA | NA | NA | NA | NA | 5.273 | 0 | 5 | 9 | 0.00 | 0 | 0 | 7/1 |

Figure 3.2 Vivado Interface

# 4. Implementation and Design

The cache memory implemented for this project has a capacity of 32B, while the main memory has a capacity of 128B. One block of data has 2B. The main memory is byte-addressable, the data bus is 8 bits wide and the address bus is 7 bits wide.

The number of blocks in the main memory is:

$$no.of\ blocks = \frac{main\ memory\ size}{block\ size} = \frac{128B}{2B} = 64\ blocks = 2^6 blocks$$

$$=> 6\ bits\ for\ block\ indexing\ in\ the\ main\ memory$$

The number of lines in the cache memory is:

$$no.of\ lines = \frac{cache\ memory\ size}{block\ size} = \frac{32B}{2B} = 16\ lines = 2^2 lines$$

$$=> 4\ bits\ for\ cache\ indexing$$

The main memory's size is 128B = $2^7$ B, so 7 bits are needed for addressing the main memory.

The cache memory's size is 32B = $2^5$ B, so 5 bits are needed for addressing the cache memory.

A block's size is of 2B, so 1 bit will be needed for addressing the block offset.

The cache controller sends a request to the cache memory. The request has the following structures, all equivalent to each other:

| | |
|---|---|
| block index<br>6 bits | block offset<br>1 bit |

| | |
|---|---|
| tag<br>2 bits | cache address<br>5 bits |

| | | |
|---|---|---|
| tag<br>2 bits | cache line number<br>4 bits | block offset<br>1 bit |

The tag index size is the difference between the number of bits needed for block indexing in the main memory and the number of bits needed for cache indexing.

The cache controller can handle two types of requests: a read request or a write request. The type of request is specified via a switch on the board, namely switch 15. If its value is ‘0’, the request is a read request; otherwise it is a write request.

The Vivado project is made up of three main components: the main memory, the cache memory and the cache memory controller, as seen in Figure 4.1.

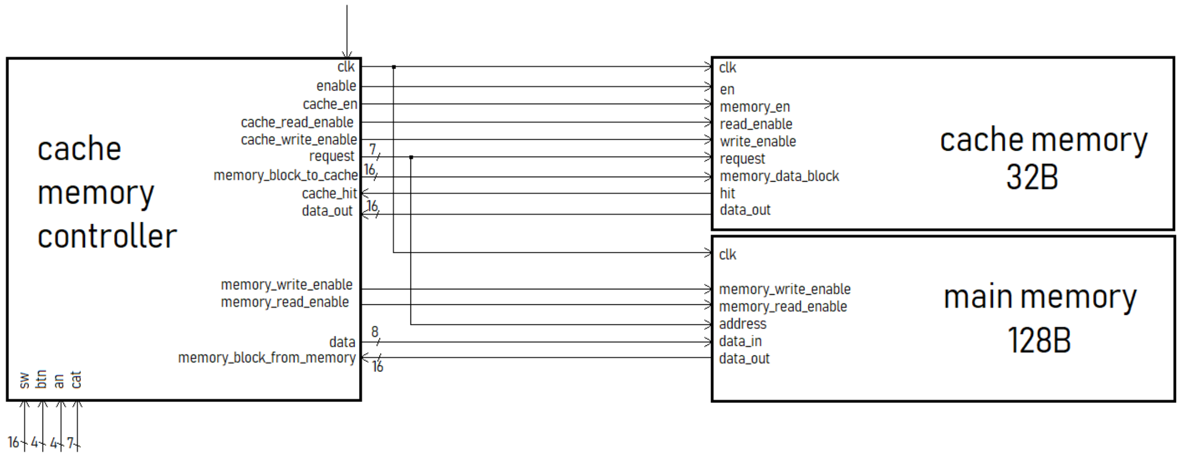


Figure 4.1 Block Schematic

## 4.1 The Main Memory

The main memory module has:

- four inputs: the clock (clk, 1 bit), the address (7 bits), and the memory read and write enable signals(memory_read_enable, memory_write_enable, 1 bit)
- one output: the data (data_out, 16 bits)

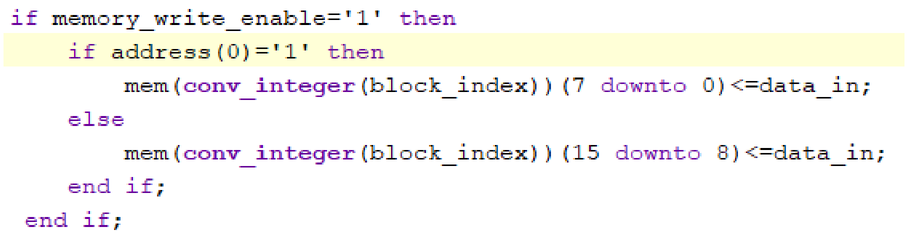
Pre-defined values are saved for certain positions in the main memory, in order to be able to test the cache memory.

For a read request, the module simply returns the data that is located at the specified block index. The block index address represents bits 6 to 1 of the input address. However, the data is returned only if the memory_read_enable signal is set to ‘1’. Otherwise, there is no need for data to be read.

```
if memory_read_enable='1' then
    data_out<=mem(conv_integer(block_index));
end if;
```

Figure 4.2 Main Memory Read Request

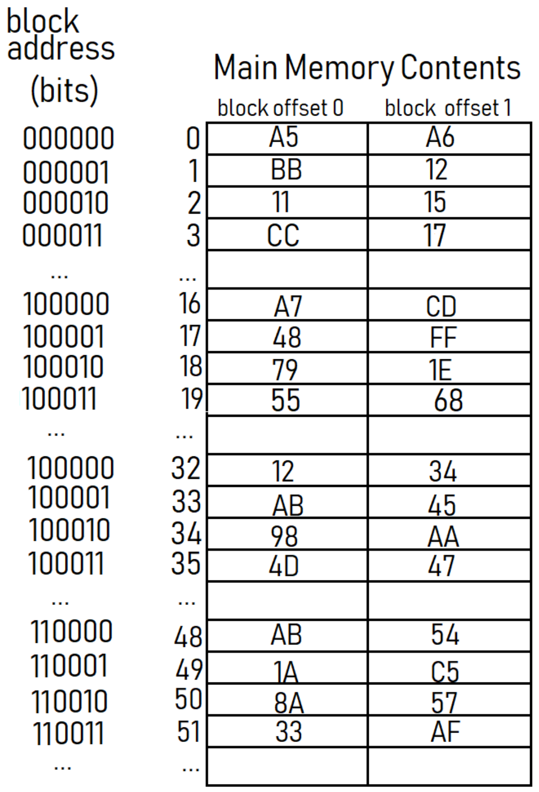
For a write request, the data_in is placed in the memory at the specified address. Depending on whether the block offset is 0 or 1 (the least significant bit of the address vector), the byte sent from the controller is placed at address 15 to 8 of the block (for a block offset equal to 0) or 7 to 0 (for a block offset equal to 1).



```
if memory_write_enable='1' then
    if address(0)='1' then
        mem(conv_integer(block_index))(7 downto 0)<=data_in;
    else
        mem(conv_integer(block_index))(15 downto 8)<=data_in;
    end if;
 end if;
```

Figure 4.1 Main Memory Write Request

The structure and content of the main memory is the following:



| block address (bits) | | Main Memory Contents | |
|---|---|---|---|
| | | block offset 0 | block offset 1 |
| 000000 | 0 | A5 | A6 |
| 000001 | 1 | BB | 12 |
| 000010 | 2 | 11 | 15 |
| 000011 | 3 | CC | 17 |
| ... | ... | | |
| 100000 | 16 | A7 | CD |
| 100001 | 17 | 48 | FF |
| 100010 | 18 | 79 | 1E |
| 100011 | 19 | 55 | 68 |
| ... | ... | | |
| 100000 | 32 | 12 | 34 |
| 100001 | 33 | AB | 45 |
| 100010 | 34 | 98 | AA |
| 100011 | 35 | 4D | 47 |
| ... | ... | | |
| 110000 | 48 | AB | 54 |
| 110001 | 49 | 1A | C5 |
| 110010 | 50 | 8A | 57 |
| 110011 | 51 | 33 | AF |
| ... | ... | | |

Figure 4.2 Main Memory

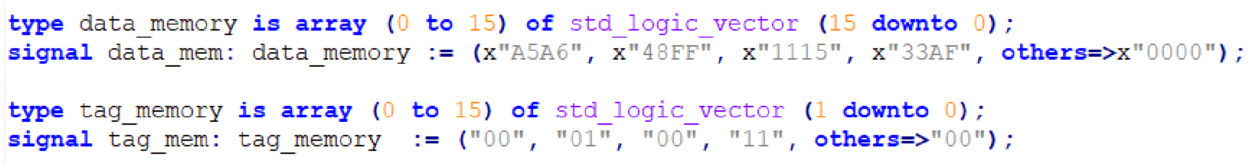
## 4.2 The Cache Memory

The cache memory module has:

- seven inputs: the clock(clk, 1 bit), two enable signals(en and memory_en, 1 bit), the read and write enable signals (read_enable, write_enable, 1 bit) and the request from the CPU (request, 7 bits)
- two outputs: the signal for hit(hit, 1 bit) and the returned data (data_out, 16 bits)

The cache memory is made up of the tag memory and the data memory. The size of the cache memory is of 32B with each line containing a block of 2B, so 4 bits are needed for addressing both the data memory and the tag memory. They are represented in the VHDL module as two arrays of 16 elements.

The tag memory is made of vectors of 2 bits, because the tag index is only 2 bits long, while the data memory is made of vectors of 16 bits, equivalent to a block of 2B.



```vhdl
type data_memory is array (0 to 15) of std_logic_vector (15 downto 0);
signal data_mem: data_memory := (x"A5A6", x"48FF", x"1115", x"33AF", others=>x"0000");

type tag_memory is array (0 to 15) of std_logic_vector (1 downto 0);
signal tag_mem: tag_memory  := ("00", "01", "00", "11", others=>"00");
```

Figure 4.5 Data Memory and Tag Memory of the Cache

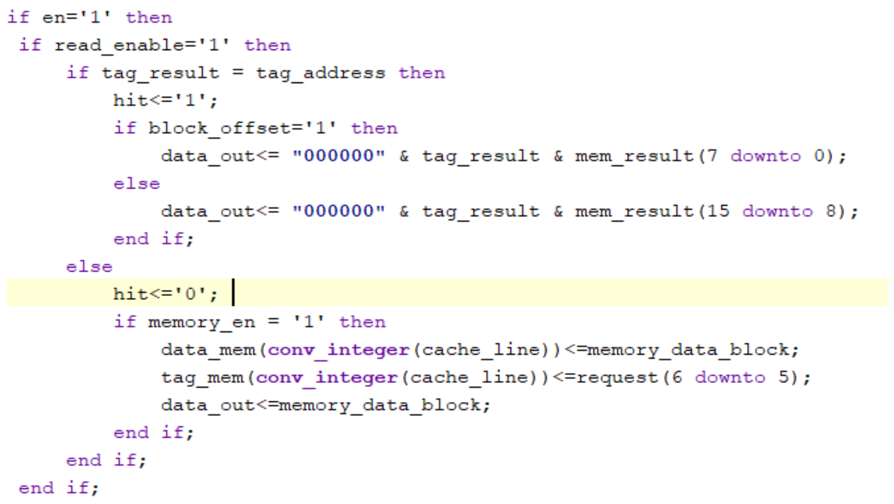
The signals used in the cache memory are:

- tag_address: 2-bit vector, holds the tag address, which is the most significant bits of the request
- cache_line:4-bit vector, holds the number of the cache line, which is bits 4 to 1 of the request
- block_offset: 1 bit, the least significant bit of the request
- tag_result: 2-bit vector, output of the tag memory which takes as address the cache line
- mem_result: 16-bit vector, output of the data memory which takes as address the cache line

The en signal is connected to the enable in the cache controller, which represents a button on the board. On each press of the button a request is sent from the controller and the cache responds.

The read_enable and write_enable signals represent a read or write request from the controller.

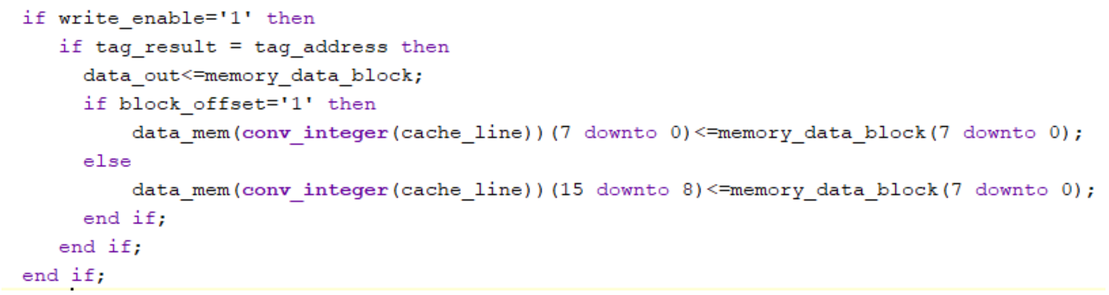
For a read request, the cache will respond with a hit or a miss. If the tag specified in the request is the same as the one located at the specified address line in the tag memory, then a cache hit occurs and the hit signal is set to ‘1’. Then, the block offset will be checked. If its value is ‘0’, then the data to be retrieved is situated in the data vector at positions 15 down to 8. Otherwise, it is situated at positions 7 downto 0. The retrieved data will be displayed on latter 2 digits of the 7 segment display, while the first two digits will hold the tag value. If the tags compared do not match, the hit signal is set to ‘0’ and a cache miss occurs. This means the cache memory has to wait for the memory_en signal from the controller to switch to ‘1’, which means data was retrieved from the main memory and is ready to be placed into the cache. This data will also be displayed on the board for a cache miss. The replacement process will be completed in three presses of the button, as data has to be read from the main memory and only then transmitted to the cache.



```
if en='1' then
 if read_enable='1' then
     if tag_result = tag_address then
         hit<='1';
         if block_offset='1' then
             data_out<= "000000" & tag_result & mem_result(7 downto 0);
         else
             data_out<= "000000" & tag_result & mem_result(15 downto 8);
         end if;
     else
         hit<='0';
         if memory_en = '1' then
             data_mem(conv_integer(cache_line))<=memory_data_block;
             tag_mem(conv_integer(cache_line))<=request(6 downto 5);
             data_out<=memory_data_block;
         end if;
     end if;
 end if;
```

Figure 4.3 Cache Memory Read Request Handling

For a write request, the data read from the switches is simply placed in the data memory at the correct offset if the block is present in the cache memory.



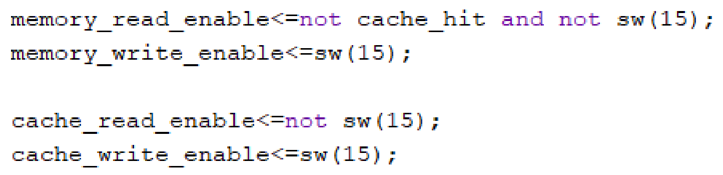
```
if write_enable='1' then
   if tag_result = tag_address then
     data_out<=memory_data_block;
     if block_offset='1' then
         data_mem(conv_integer(cache_line))(7 downto 0)<=memory_data_block(7 downto 0);
     else
         data_mem(conv_integer(cache_line))(15 downto 8)<=memory_data_block(7 downto 0);
     end if;
   end if;
end if;
```

Figure 4.7 Cache Memory Write Request Handling

## 4.3 The Cache Memory Controller

The Cache Memory Controller is the module that reads the data and the request and sends them to the cache or main memory. It has three inputs: the clock, the switches (sw signal, 16 bits) and the buttons (btn, 4 bits) of the board. The switches are used to input a request, the request type and the data to be read. Only a button is used; when pressed, the request and data are read and the response shows up on the display and leds.

The 15th switch on the board is used to signal whether the request sent to the cache is a read or write one. The signals are computed as following:



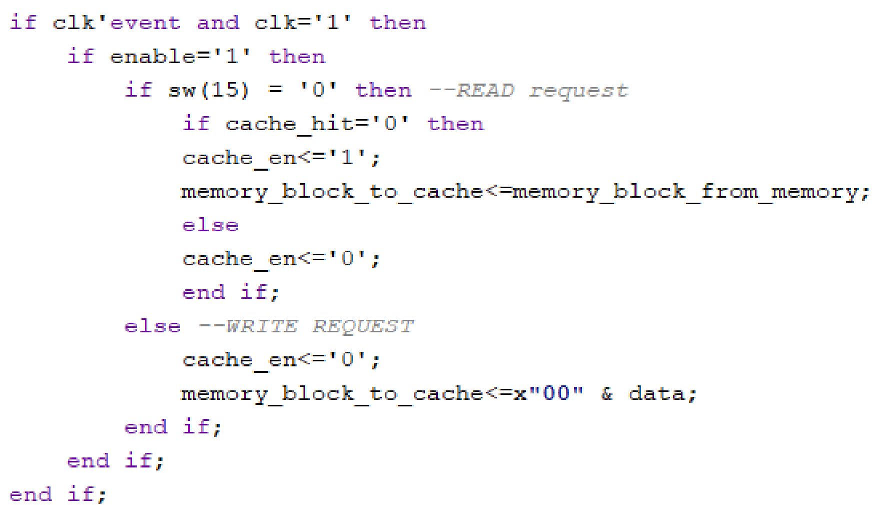
```
memory_read_enable<=not cache_hit and not sw(15);
memory_write_enable<=sw(15);

cache_read_enable<=not sw(15);
cache_write_enable<=sw(15);
```

Figure 4.8 Cache Controller Signals

When the switch is set to ‘0’, the request is a read. Therefore, the cache_read_enable will be set to ‘1’. If there is a cache miss, the memory_read_enable will also switch to ‘1’.Otherwise, for a write request, i.e. a switch set to ‘1’, both the cache_write_enable and memory_write_enable will be set to ‘1’.

The handling of a request is presented in the figure below. For a read request, the controller sends it to the cache memory and checks if a cache hit or a cache miss occurred. If there was a cache hit, the data and the tag are simply displayed on the board. Otherwise, the data block has to be retrieved from the main memory (cache_en signal becomes ‘1’), the cache_write_enable signal becomes ‘1’ and data is sent and written to the cache.

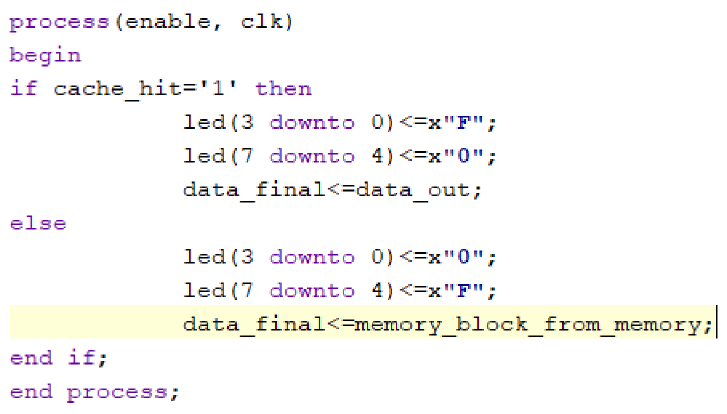
For a write request, a byte is read from the switches and sent to both the main memory and the cache memory after the corresponding write signals have been enabled. The block is modified according to the offset in the main memory and in the cache memory as well, if present there (the tags match).



```vhdl
if clk'event and clk='1' then
    if enable='1' then
        if sw(15) = '0' then --READ request
            if cache_hit='0' then
            cache_en<='1';
            memory_block_to_cache<=memory_block_from_memory;
            else
            cache_en<='0';
            end if;
        else --WRITE REQUEST
            cache_en<='0';
            memory_block_to_cache<=x"00" & data;
        end if;
    end if;
end if;
```

Figure 4.9 Cache Controller Request Handling

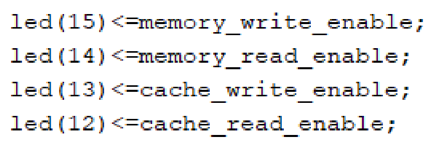
The seven segment display on the board will show different values depending on the request and cache response. For a read request, if a cache hit occurs, the tag and byte will be displayed and the last 4 leds on the board will be switched on. If a cache hit occurs, leds 7 to 4 will be turned on and the data displayed will represent the memory block that will be placed in the cache. For a write request, the leds will behave the same as for a read request and the displayed data will represent the byte read from the switches.



```vhdl
process(enable, clk)
begin
if cache_hit='1' then
            led(3 downto 0)<=x"F";
            led(7 downto 4)<=x"0";
            data_final<=data_out;
else
            led(3 downto 0)<=x"0";
            led(7 downto 4)<=x"F";
            data_final<=memory_block_from_memory;
end if;
end process;
```

Figure 4.10 Cache Controller Display

The read and write enable signals for the cache and main memories are also shown on the leds:



```vhdl
led(15)<=memory_write_enable;
led(14)<=memory_read_enable;
led(13)<=cache_write_enable;
led(12)<=cache_read_enable;
```

Figure 4.11 Cache Controlled Led Signals

# 5.Experimental Results

## 5.1 User Manual

The Basys3 board is shown in the figure below:

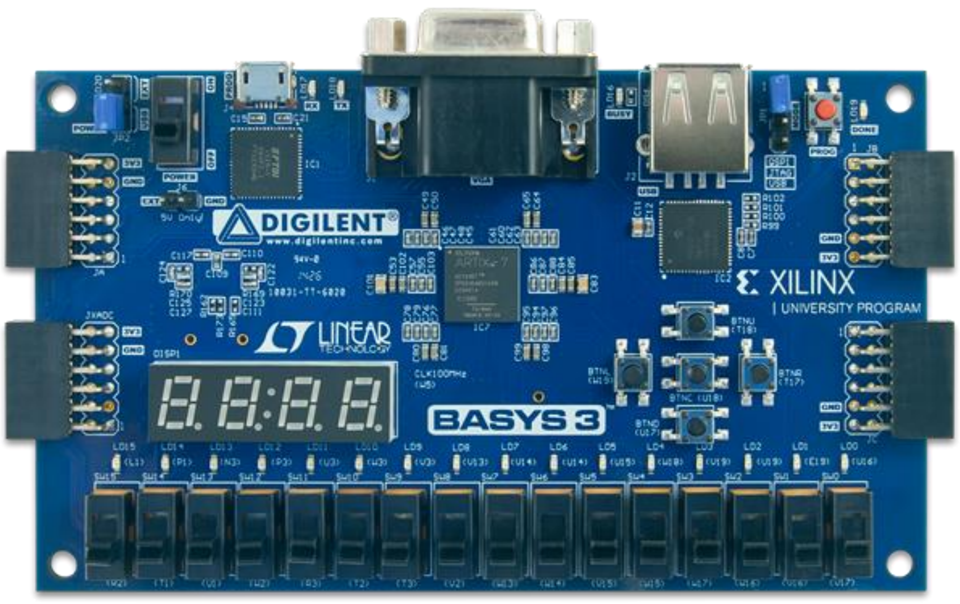


Figure 5.1 Basys3 board

### 5.1.1. Switches

The first switch (switch 15) is used to specify the type of request: ‘0’ for read, ‘1’ for write.

Switches 14 to 7 are used for data input in case of a write request. In case of a read request, their value does not matter.

Switches 6 to 0 are used for the request (read or write). For the main memory, the first 5 bits (switches 6 to 1) represent the block index, while the last bit (switch 0) represents the block offset. For the cache memory, the first two bits of the request (switches 6 and 5) represent the tag, the next 5 bits represent the cache line (switches 4 to 1) and the last bit represents the block offset (switch 0).

### 5.1.2 Leds

Leds display the following information:

Led 15: main memory write enable

Led 14: main memory read enable

Led 13: cache memory write enable

Led 12: cache memory read enable

Leds 7-4: turn on when a cache miss occurs in a read or write request

Leds 3-0: turn on when a cache hit occurs in a read or write request

### 5.1.3 How to use

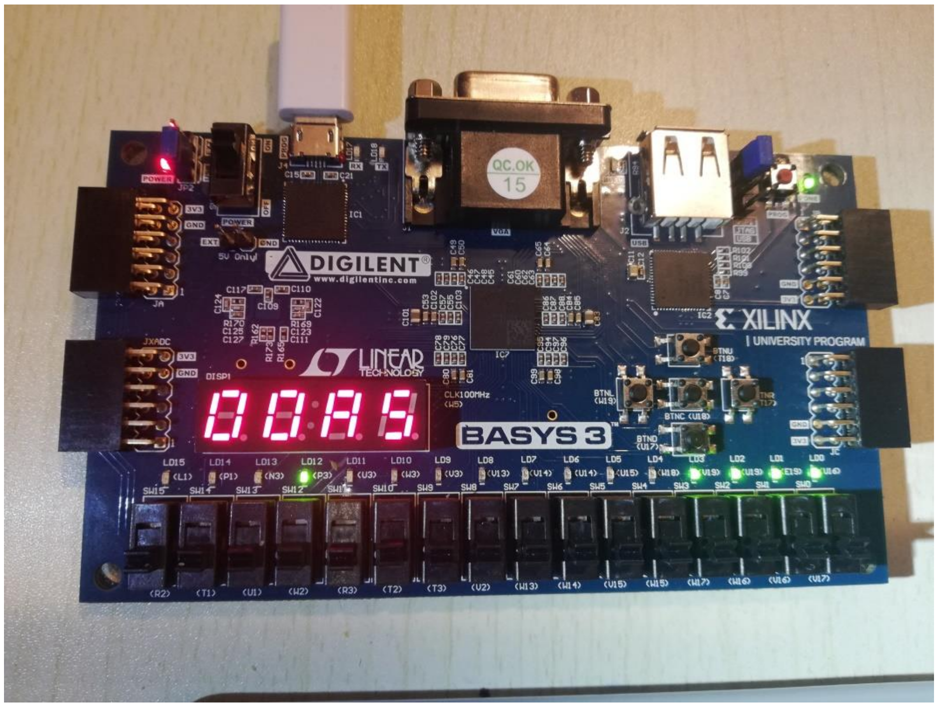
After the type of request and the input is selected, the middle button is pressed.

For a read request and a cache hit, the display shows the tag on the first two digits and the requested byte on the last two digits. For a cache miss, at the next press of the button, the memory block fetched from the main memory is displayed and at the following press of the button the block has been placed in the cache memory and a cache hit occurs. The fact that the button is pressed multiple times in a cache miss procedure is also used to demonstrate that reading from the main memory is costlier than reading from the cache memory.

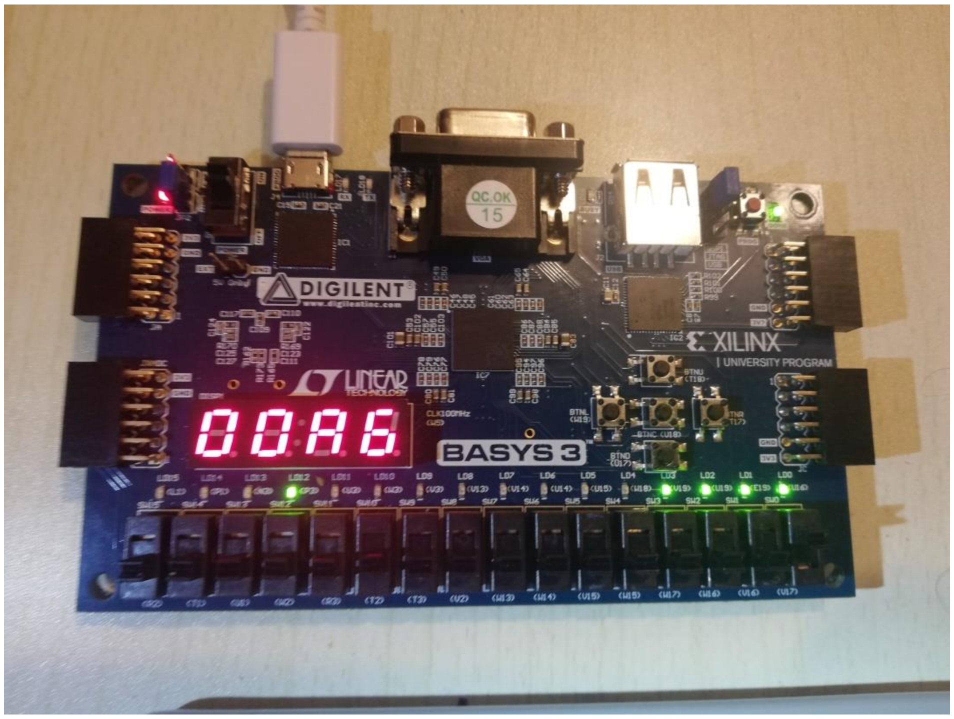
For a write request, at the press of the button, the display will show the byte read from the switches. The controller sends the data to both of the memories. If the block is present in the cache, it will be replaced there as well as in the main memory.

## 5.2. Demo

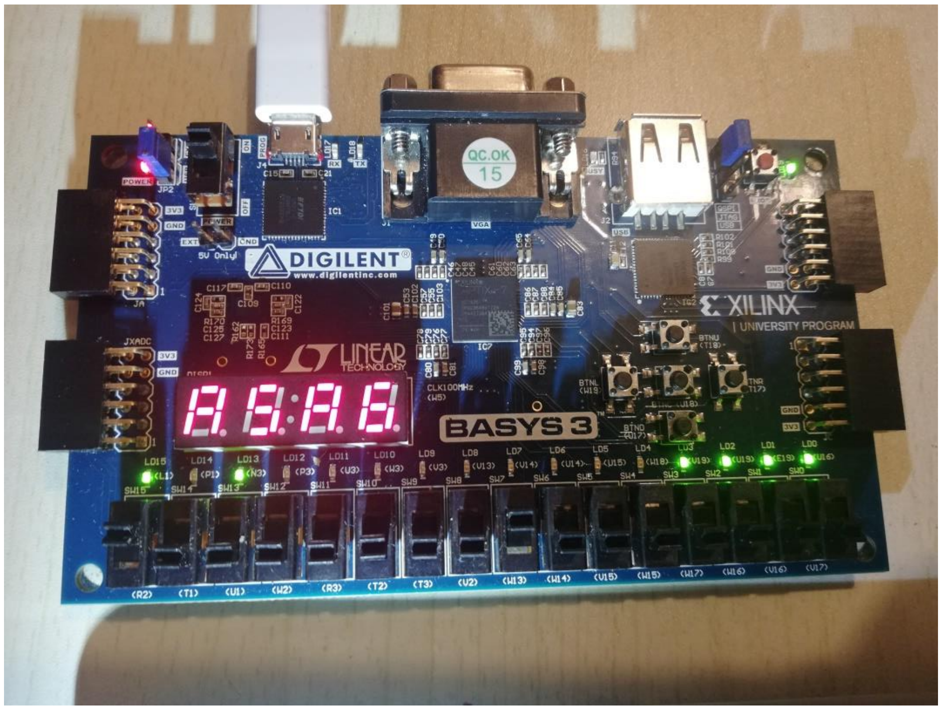
The board is programmed. The request is a read request, “0000000” and the cache_read_enable signal is ‘1’. After pressing the button, leds 3 to 0 will turn on to signal a cache hit, and the display will show “00A5”. We can observe that the tag of the request matched the tag of the line in the cache memory at line 00000 and offset 0.



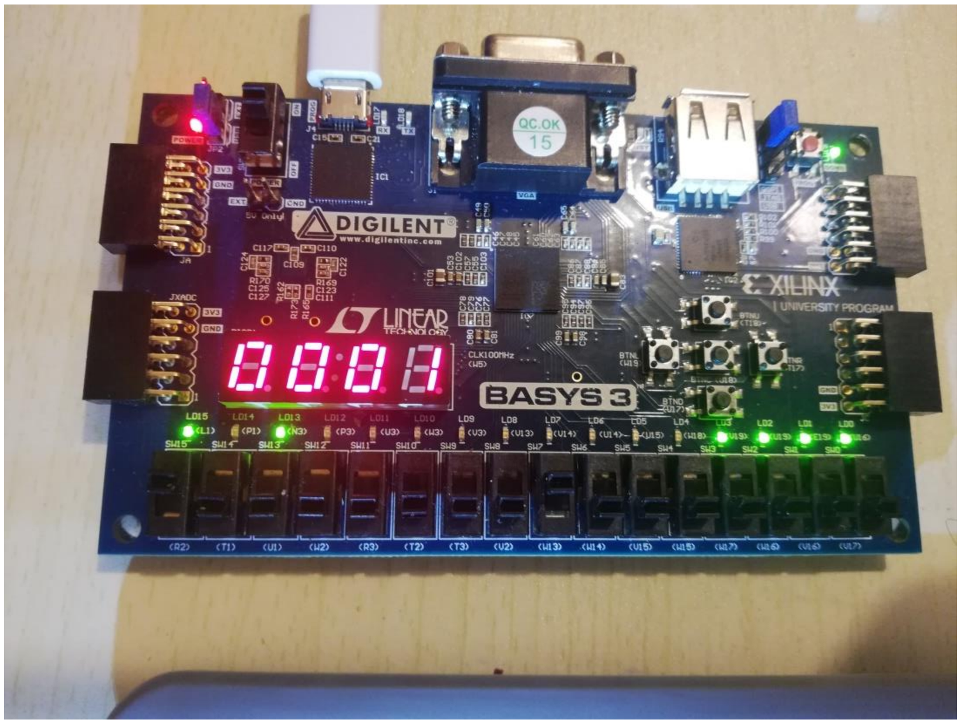
Changing the last switch to ‘1’ and pressing the button will cause the display to show the byte situated at offset 1 of the same cache line: A6. The request is still a cache hit, as the tag has not changed.



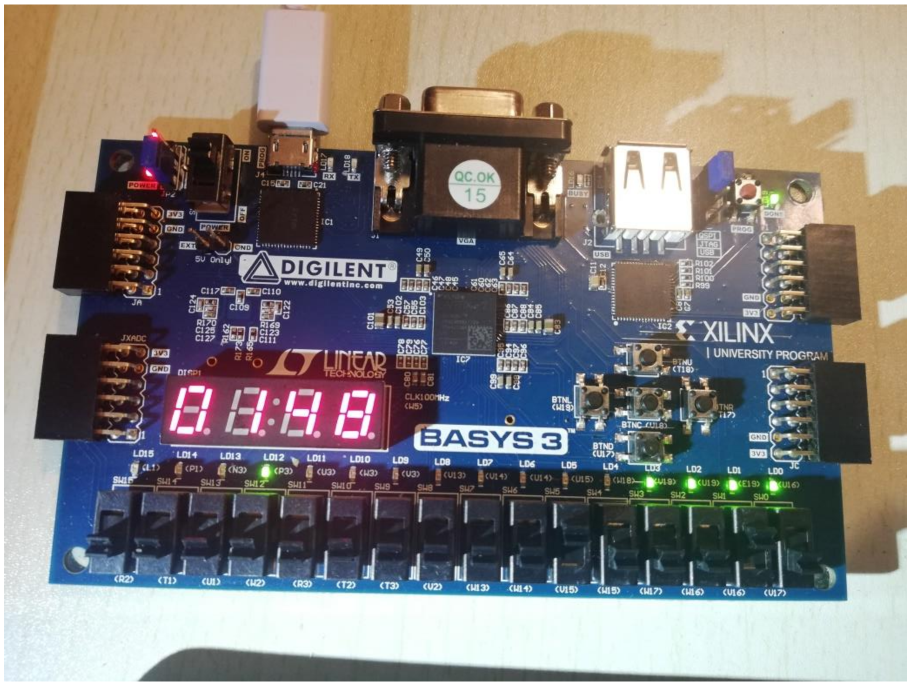
Next, a write request is executed. The display will start by displaying the data block that will be changed: A5A6, the first block of the main memory that is also present in the cache memory. The data that we want to place at address 0 and offset 0 of the main memory is “01”, as shown in the figure below: switched 14 to 7 are set to “00000001”. The cache_write_enable and memory_write_enable signals become ‘1’.



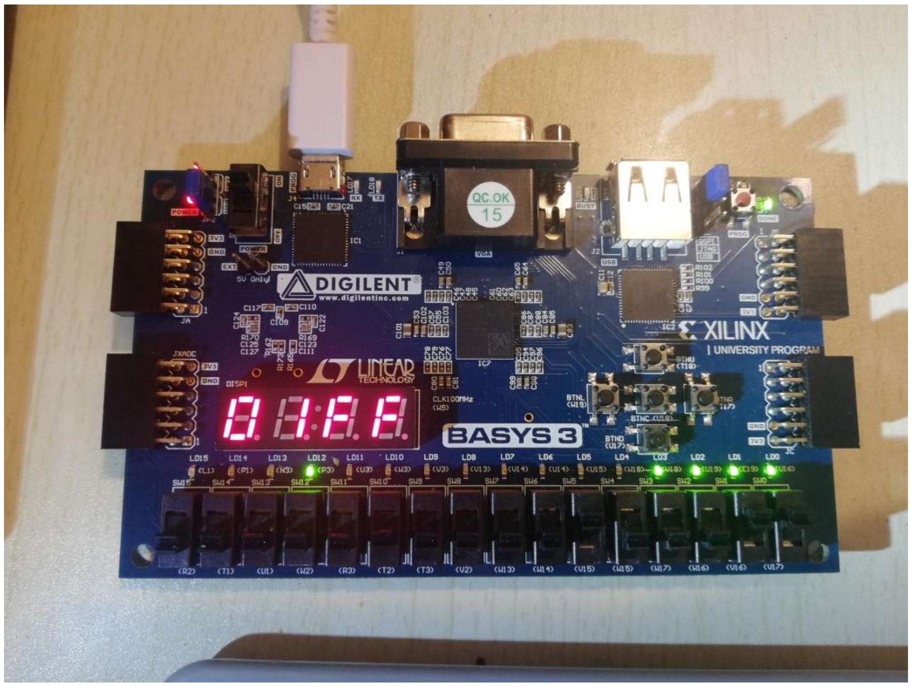
At the next press of the button, the byte has been written to both the cache and the main memory, and its value is displayed on the board, in this case 0001.



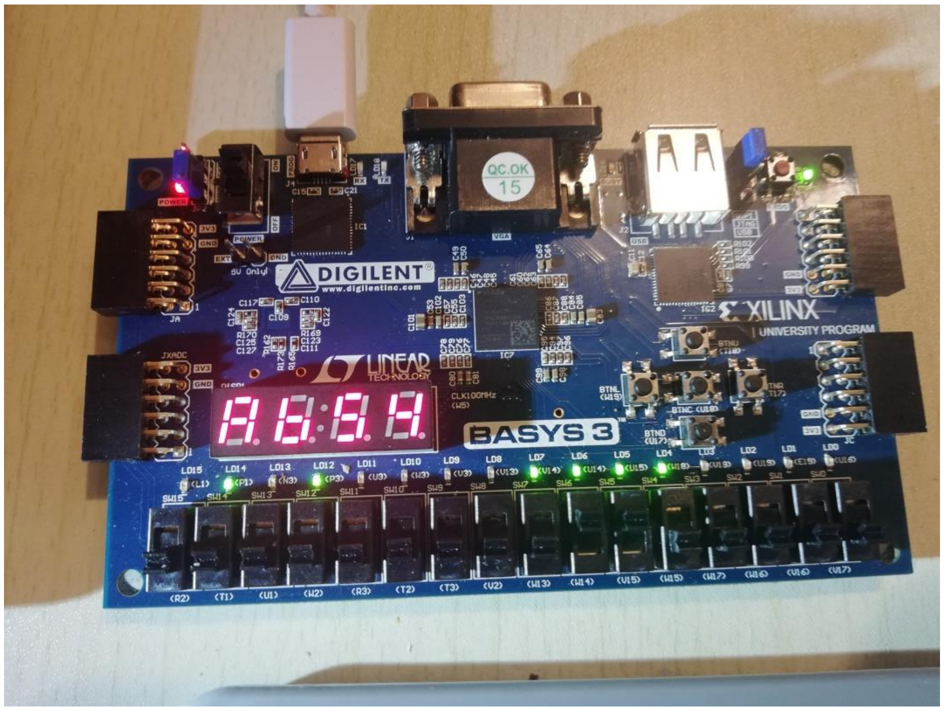
Another read request for the first line in the cache (line 00001), offset 0 and tag 01 (request 01000010) will result in a cache hit. The byte read from the cache is “48”.



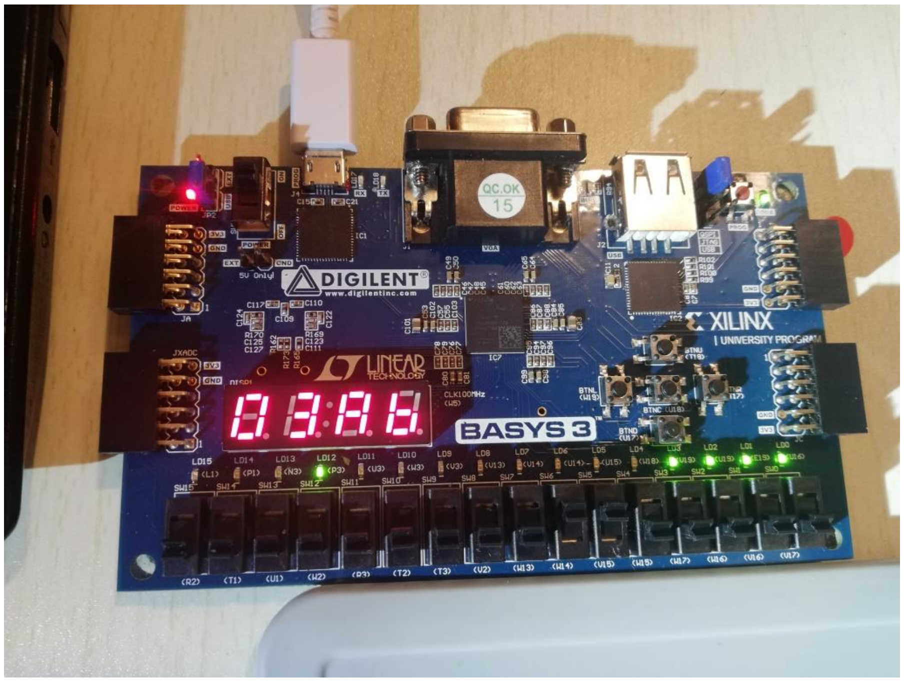
The other byte in the block is “FF”, same request but an offset of 1, still a cache hit:

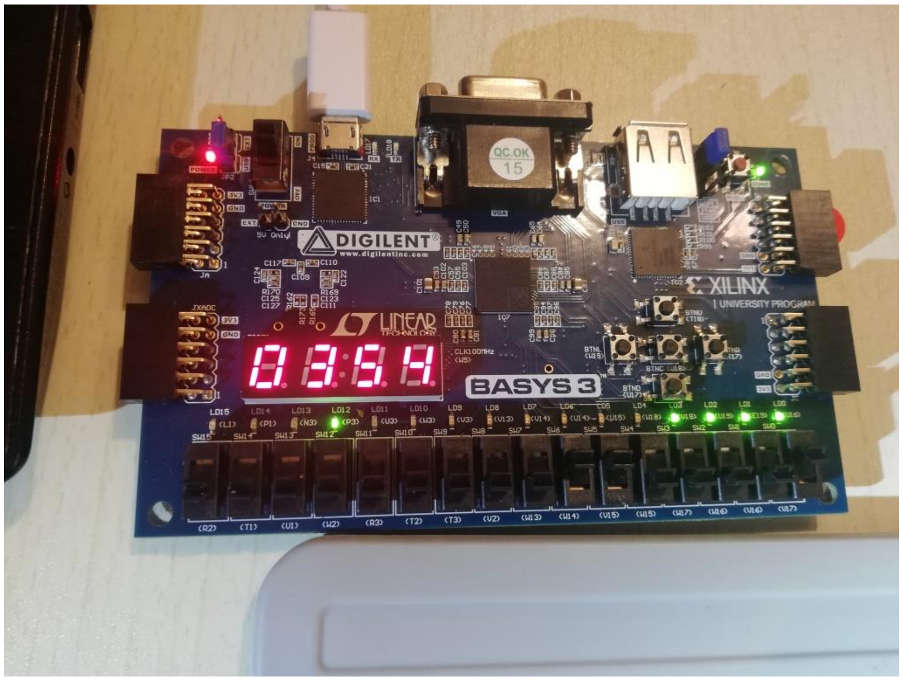


Another read request, of “1100000” should result in a cache miss: the tag at line 0 in the cache is “00”, while the tag that we need is “11”. Indeed, for such a request, the board displays the block that will be written at the next press of a button and the leds will signal a cache miss. The memory_read_enable signal is turned on, because the block has to be read from the main memory:

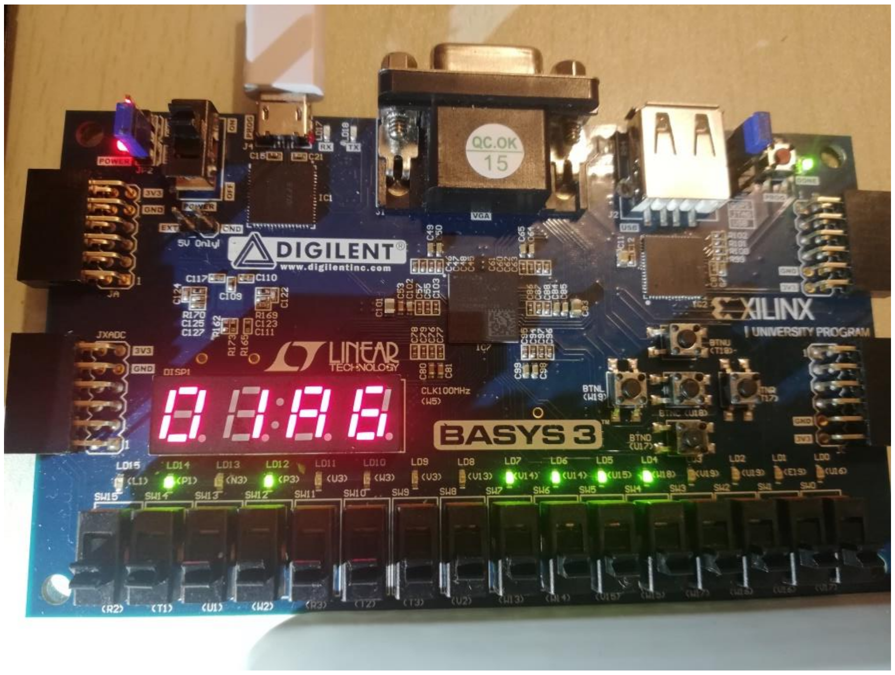


After the cache has been written, the same request will result in a cache hit for offset 0 and 1:

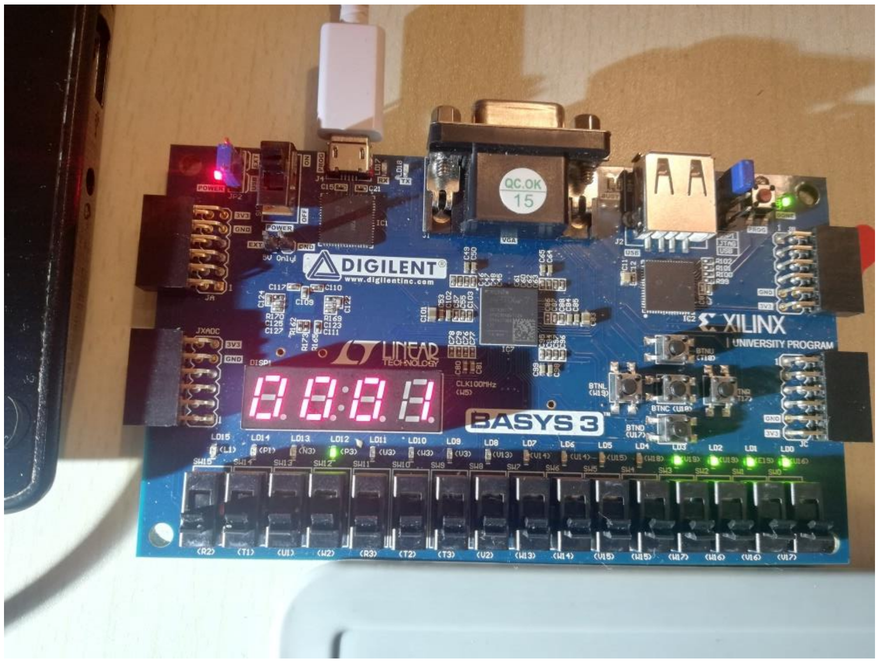
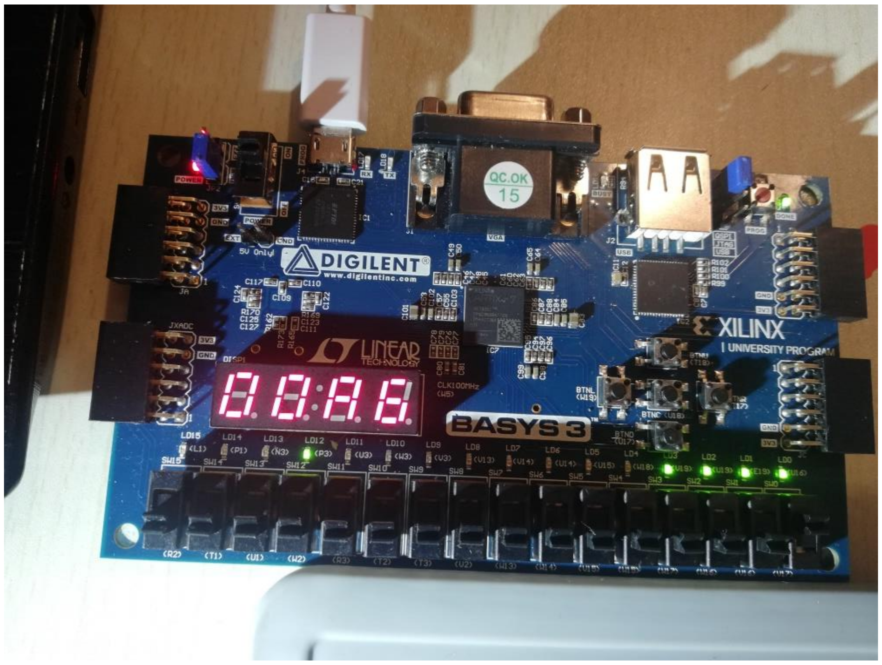




In order to check if the write operation performed earlier has been carried through correctly, a read request of “0000000” is performed. The board should signal a cache miss and display the block read from memory, which is no longer “A5A6”, but “01A6”:



After the cache miss, the block is placed in the memory and the request is sent again, resulting in a cache hit:

# 6.Conclusion

This project has lead to a better understanding of not only a cache memory and its interaction with the controller and main memory, but also the VHDL programming language and the Basys3 board.

Through this project, several concepts related to a cache memory have been studied, such as levels of the cache, write policies and cache miss handling.

## 7. References and Bibliography

[1] https://searchstorage.techtarget.com/definition/cache-memory

[2] https://computersciencewiki.org/index.php/Cache_memory

[3] https://www.geeksforgeeks.org/cache-memory-in-computer-organization/

[4] https://en.wikipedia.org/wiki/Cache_(computing)

[5] https://en.wikipedia.org/wiki/CPU_cache

[6] https://www.sciencedirect.com/topics/computer-science/direct-mapped-cache

[7] https://en.wikipedia.org/wiki/Cache_placement_policies