Project Report

Claudio César Claros Olivares
Paulo Roberto Loma Marconi
Walter Abel Claros Olivares

July 16, 2017

1 Introduction

Over the recent years, the CPU has evolved very quickly achieving higher and higher processing speeds. However, the memory, which is the element that provides the CPU with the data to be processed, has not followed the same path. In one hand, most of the problems with the regular memories lie on the access time since it takes too many clocks cycles to retrieve or store data for the CPU, which is a waste of its processing speed. In the other hand, faster memories that can meet the speed requirements of actual CPUs are very expensive which makes them impractical to be utilized when it comes to large storage capacities. In that sense, the concept of memory hierarchy has been introduced in order to deal with these difficulties.

The principle behind memory hierarchy resides in the use of small, affordable yet fast memories, known as cache memories, whose access times are very short compared to the access time of regular memories. These cache memories work as interfaces between the CPU and the main memory which try to maximize the data transfer speed while avoiding accessing the slow, large, main memory. By doing this, the overall performance of the CPU is significantly boosted.

In this project, the goal was to *design, simulate and implement a Direct-mapping cache memory system in VHDL and compare it to a reference system without cache memory to verify the performance enhancement.* With this purpose in mind, we performed modifications on the reference system in both structure and functionality, considering that the benchmarking application is a *matrix multiplication that shows the impact of the target cache memory scheme in the performance of the architecture.* In the following sections, the design, the implementation and the simulation of the cached system are described.

2 Design

2.1 Reference system modifications

The original system provides nine possible operations (Table 1) which allow us to execute simple processes. Nonetheless, performing a matrix multiplication requires an advanced set of functions that provides us with more flexibility to carry out complex procedures. Besides, we have to overcome the addressing limitation since the original system has an 8-bit address bus and the instructions were designed considering that the memory addresses were not going to exceed the $2^8 - 1$ possible directions. The reference system must have a 12-bit-wide address bus, so all the instructions that used direct addressing are now useless. That is why, we came up with six new instructions: multiplication (mult), indexed-memory read (mov5), jump if not equal (jne), increment (inc), register-to-register copy (mov6) and indexed-memory out (outRF).

Original Instruction Set		
Symbolic	**Operation Code**	**HDL**
mov1	'0000' = 0x0	RF[r_n]←mem[direct]
mov2	'0001' = 0x1	mem[direct]←RF[r_n]
mov3	'0010' = 0x2	mem[RF[r_n]]←RF[r_m]
mov4	'0011' = 0x3	RF[r_n]←imm
add	'0100' = 0x4	RF[r_n]←RF[r_n]+RF[r_m]
subt	'0101' = 0x5	RF[r_n]←RF[r_n]−RF[r_m]
jz	'0110' = 0x6	if RF[r_n]= 0:PC←imm
readm	'0111' = 0x7	out←mem[direct]
halt	'1111' = 0xF	do nothing

Instruction Format (16 bits)			
Operation Code (4 bits)	RF[r_n] (4 bits)	RF[r_m] (4 bits)	RF[r_o] (4 bits)
		immediate constant (8 bits)	
	First operand	Second operand	Third operand

Table 1: Original system's instruction set and instruction format

2.1.1 Multiplication

The multiplication instruction considers that the data bus is 16-bit wide. In consequence, the operands must be 8-bit long to meet the CPU's data dimension. This consideration brings up a limitation related to the operands maximum value which is $2^8 - 1 = 255$, meaning that the maximum possible product equals $255 \cdot 255 = 65025$. Taking into account that the data bus has 16 bits, this last result does not exceed the maximum allowed by the CPU registers (65535 or 0xFFFF). This instruction uses the two first operands to execute the mathematical operation and the result is stored into the first operand specified in the instruction format. The description of the multiplication instruction is as follows:

Symbolic	Operation Code	HDL
mult	'1000' = 0x8	RF[r_n]←RF[r_n]∗RF[r_m]

The procedure to perform the multiplication is shown in Fig. 1 along with the code.

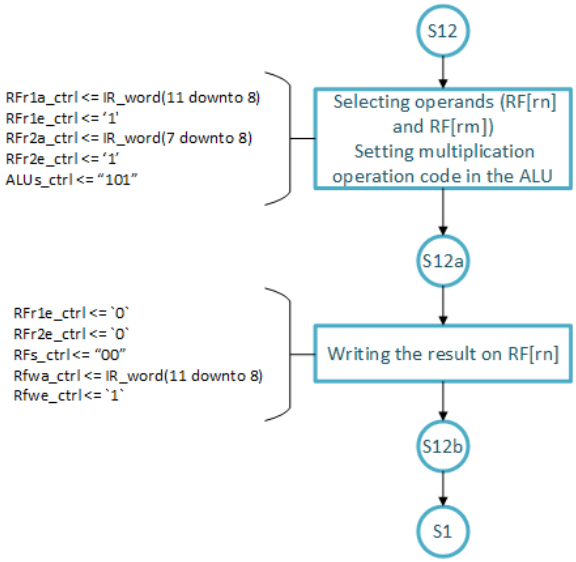


Figure 1: Multiplication state diagram

```
when S12 => -- RF[rn] <- RF[rn] * RF[rm]
  RFr1a_ctrl <= IR_word(11 downto 8);
  RFr1e_ctrl <= '1';
  RFr2a_ctrl <= IR_word(7 downto 4);
  RFr2e_ctrl <= '1';
  ALUs_ctrl <= "101";
  state <= S12a;
when S12a =>
  RFr1e_ctrl <= '0';
  RFr2e_ctrl <= '0';
  RFs_ctrl <= "00";
  RFwa_ctrl <= IR_word(11 downto 8);
  RFwe_ctrl <= '1';
  state <= S12b;
when S12b =>
  state <= S1;
```

Listing 1: Multiplication state vhdl

2.1.2 Indexed-memory read

In order to read stored data from the memory, we can count on the instruction *mov1*. Nevertheless, this operation presents two problems. First, we need to specify directly in the instruction the address of the stored element to be read, which means that recursiveness gets too complicated when it comes to several mathematical operations. Second, the limit for addressable locations given by this instruction is 255 or 0xFF, which is a very reduced portion of the reference system's memory. Therefore, it was imperative that a new instruction be designed that can solve these setbacks. By indexing, we can use the CPU registers to set the addresses of the elements that we want to process. In doing so, we can now address $2^{16} - 1 = 65535$ possible locations, which meet the requirements of the reference system. Moreover, the fact that the address can be expressed within the content of a register enables us to use this instruction inside a loop more easily since it can modified after each iteration. The description of the indexed-memory read instruction is as follows:

Symbolic	Operation Code	HDL
mov5	'1001' = 0x9	RF[r_m]←mem[RF[r_n]]

The procedure to perform the indexed-memory read is shown in Fig. 2 along with the code.

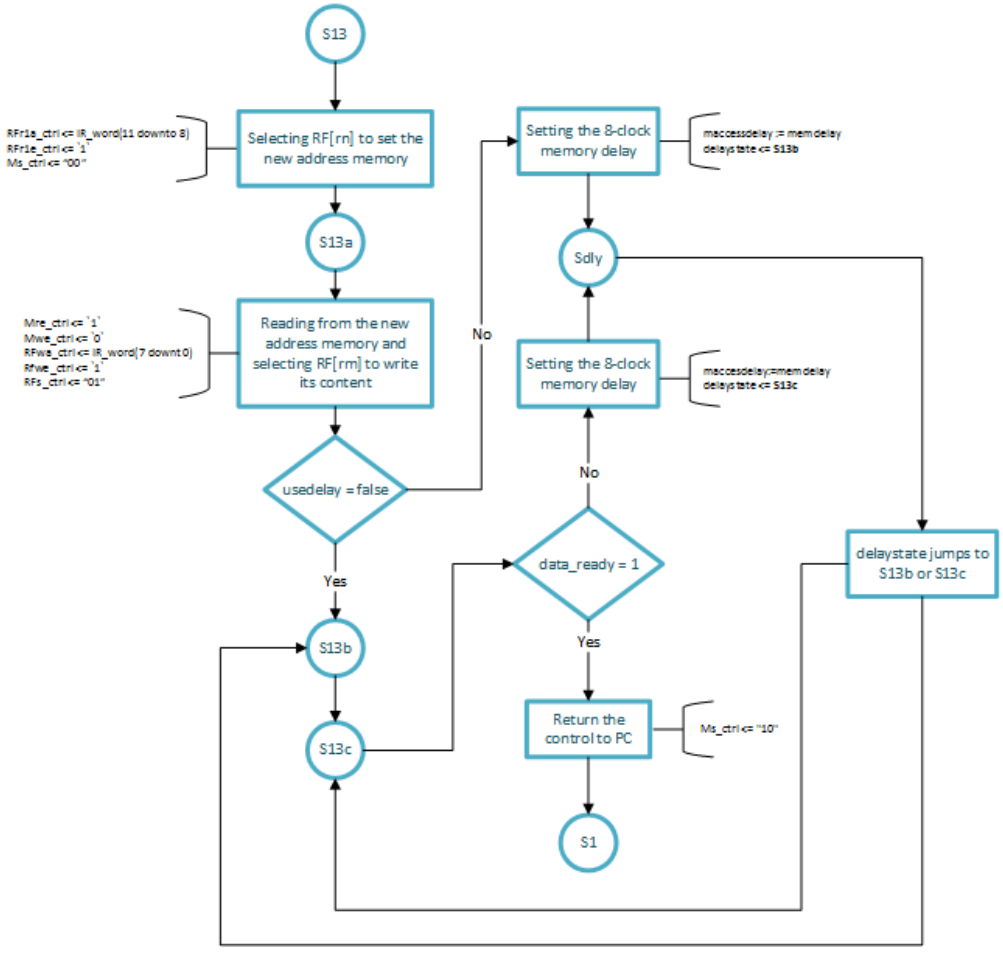


Figure 2: Indexed memory read state diagram

```
when S13 => -- RF[rm] <- mem[RF[rn]]
  RFr1a_ctrl <= IR_word(11 downto 8);
  RFr1e_ctrl <= '1';
  Ms_ctrl <= "00"; -- fetching content of RF[rn] for address memory
  state <= S13a;
when S13a =>
  Mre_ctrl <= '1';
  Mwe_ctrl <= '0';
  RFwa_ctrl <= IR_word(7 downto 4); -- RF[rm]
  RFwe_ctrl <= '1';
  RFs_ctrl <= "01"; -- save mem_data
  if usedelay = false then
    state <= S13b;
  else
    maccesdelay := memdelay;
    delaystate <= S13b;
    state <= Sdly;
  end if;
when S13b =>
  state <= S13c;
when S13c =>
  if data_ready = '1' then
    Ms_ctrl <= "10";-- return
    state <= S1;
  elsif data_ready = '0' then
    maccesdelay := memdelay;
    delaystate <= S13c;
```

```
    state <= Sdly;
  end if;
```

Listing 2: Indexed memory read state vhdl

2.1.3 Jump if not equal

The operation *jz* from the original set allows us to control the program flow and implement loops; however, the destination operand is defined by the immediate constant, which is 8-bit long and let us branch only within the first 255 memory locations. In the light of this issue, we dealt with it by setting up a new instruction called *jump if not equal*. The *jump-if-not-equal* operation uses the three registers involved in the instruction format. The first two operands are utilized to verify the condition, which checks if their content are not the same. If the condition yields true then the Program Counter is set to the content of the third register, otherwise the Program Counter behaves regularly. The description of the jump-if-not-equal instruction is as follows:

Symbolic	Operation Code	HDL
jne	'1010' = 0xA	if RF[r_n]≠RF[r_m]:PC←RF[r_o]

The procedure to perform the jump-if-not-equal operation is shown in Fig. 3 along with the code.

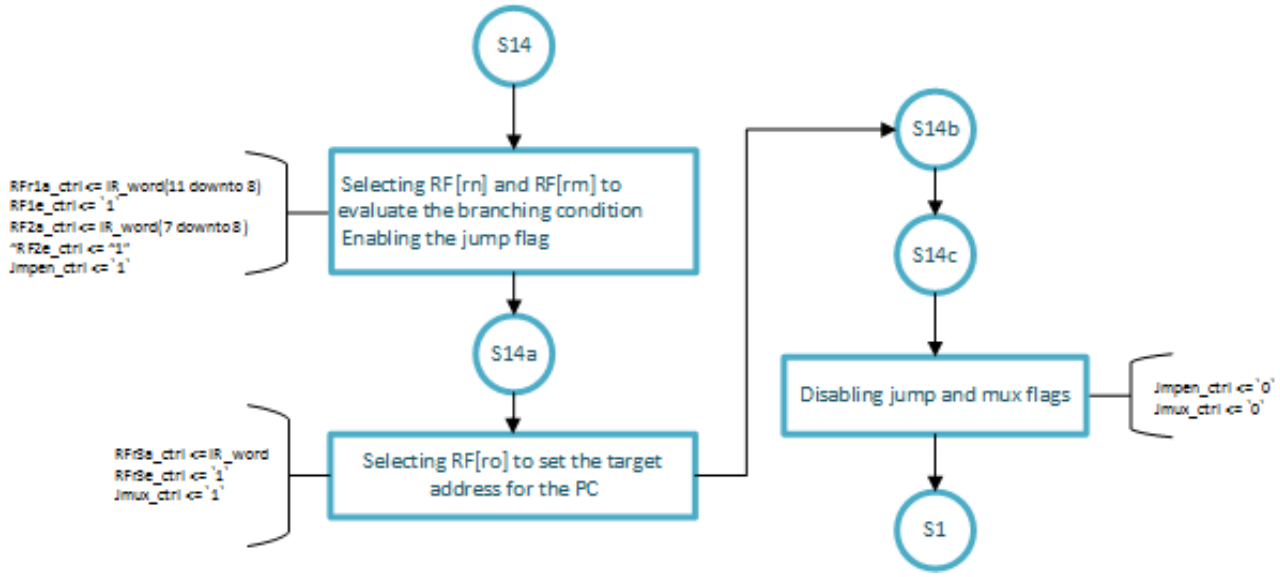


Figure 3: Jump if not equal state diagram

```
when S14 =>  -- jne, RF[rn] ~= RF[rm] : PC[RF[ro]]
  RFr1a_ctrl <= IR_word(11 downto 8);
  RFr1e_ctrl <= '1';
  RFr2a_ctrl <= IR_word(7 downto 4);
  RFr2e_ctrl <= '1';
  jmpen_ctrl <= '1';
  state <= S14a;
when S14a =>
  RFr3a_ctrl <= IR_word(3 downto 0);
  RFr3e_ctrl <= '1';
  jmux_ctrl <= '1';
  state <= S14b;
when S14b =>
```

```
  state <= S14c;
when S14c =>
  jmpen_ctrl <= '0';
  jmux_ctrl <= '0';
  state <= S1;
```

Listing 3: Jump if not equal state vhdl

2.1.4 Increment

With the original instruction set, two operations has to be executed in order to achieve unitary increments: first, one register has to be set to one and, second, execute the addition instruction with this last register; this makes the code more extensive and less intuitive. Besides, inside a loop, unitary increments are very common. In that sense, we designed the increment instruction which increases by a unit the value of the register specified in the first operand of the instruction format. The description of the increment operation is as follows:

Symbolic	Operation Code	HDL
inc	'1011' = 0xB	RF[r_n]←RF[r_n]+1

The procedure to perform the increment instruction is shown in Fig. 4 along with the code.

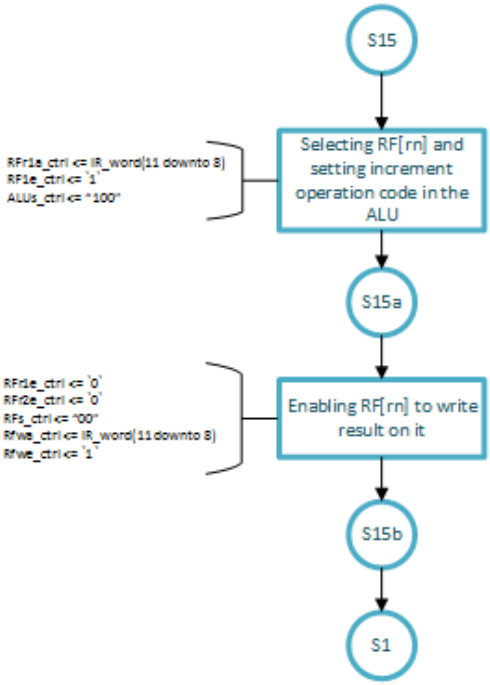


Figure 4: Increment state diagram

```
when S15 => -- inc, RF[rn] <- RF[rn] + 1
  RFr1a_ctrl <= IR_word(11 downto 8);
  RFr1e_ctrl <= '1';
  ALUs_ctrl <= "100";
  state <= S15a;
when S15a =>
  RFr1e_ctrl <= '0';
  RFr2e_ctrl <= '0';
  RFs_ctrl <= "00";
  RFwa_ctrl <= IR_word(11 downto 8);
```

```
  RFwe_ctrl <= '1';
  state <= S15b;
when S15b =>
  state <= S1;
```

Listing 4: Increment state vhdl

2.1.5 Register-to-register move

In general terms, it is more efficient to keep the data that has to be processed inside the CPU registers until every calculation that needs it is done. In this way, there is no need to reach the memory, which could take many clock cycles from the CPU. For this reason, it is useful to copy the content from one register to another temporarily, so it can be accessed rapidly without the need to store it to and then retrieve it from the memory. The data movement from one register to another, for our instruction format, copies the content from the second operand to the content of the first operand. The description of the register-to-register move operation is as follows:

Symbolic	Operation Code	HDL
mov6	'1100' = 0xC	RF[r_n]←RF[r_m]

The procedure to perform the register-to-register move instruction is shown in Fig. 5 along with the code.

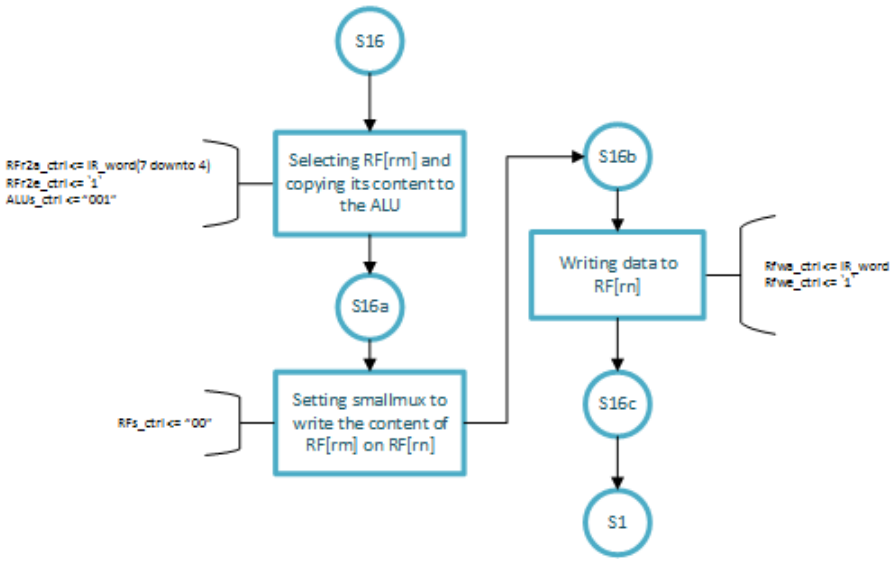


Figure 5: Register-to-register move state diagram

```
when S16 => -- RF[rn] <- RF[rm]
  RFr2a_ctrl <= IR_word(7 downto 4);
  RFr2e_ctrl <= '1';
  ALUs_ctrl <= "001";
  state <= S16a;
when S16a =>
  RFs_ctrl <= "00";
  state <= S16b;
when S16b =>
  RFwa_ctrl <= IR_word(11 downto 8);
  RFwe_ctrl <= '1';
  state <= S16c;
when S16c =>
```

```
14    state <= S1;
```

Listing 5: Register-to-register move state vhdl

2.1.6 Indexed memory out

The instruction in charge of displaying results (readm) presents the same problem of the instructions that use the immediate constant as part of their execution: the lack of bits to address the whole memory. Taking into consideration this caveat, we designed a instruction that reads the content from the memory indexed by the content of the register specified in the first operand. he description of the register-to-register move operation is as follows:

Symbolic	Operation Code	HDL
outRF	'1101' = 0xD	out←RF[r_n]

The procedure to perform the register-to-register move instruction is shown in Fig. 6 along with the code.

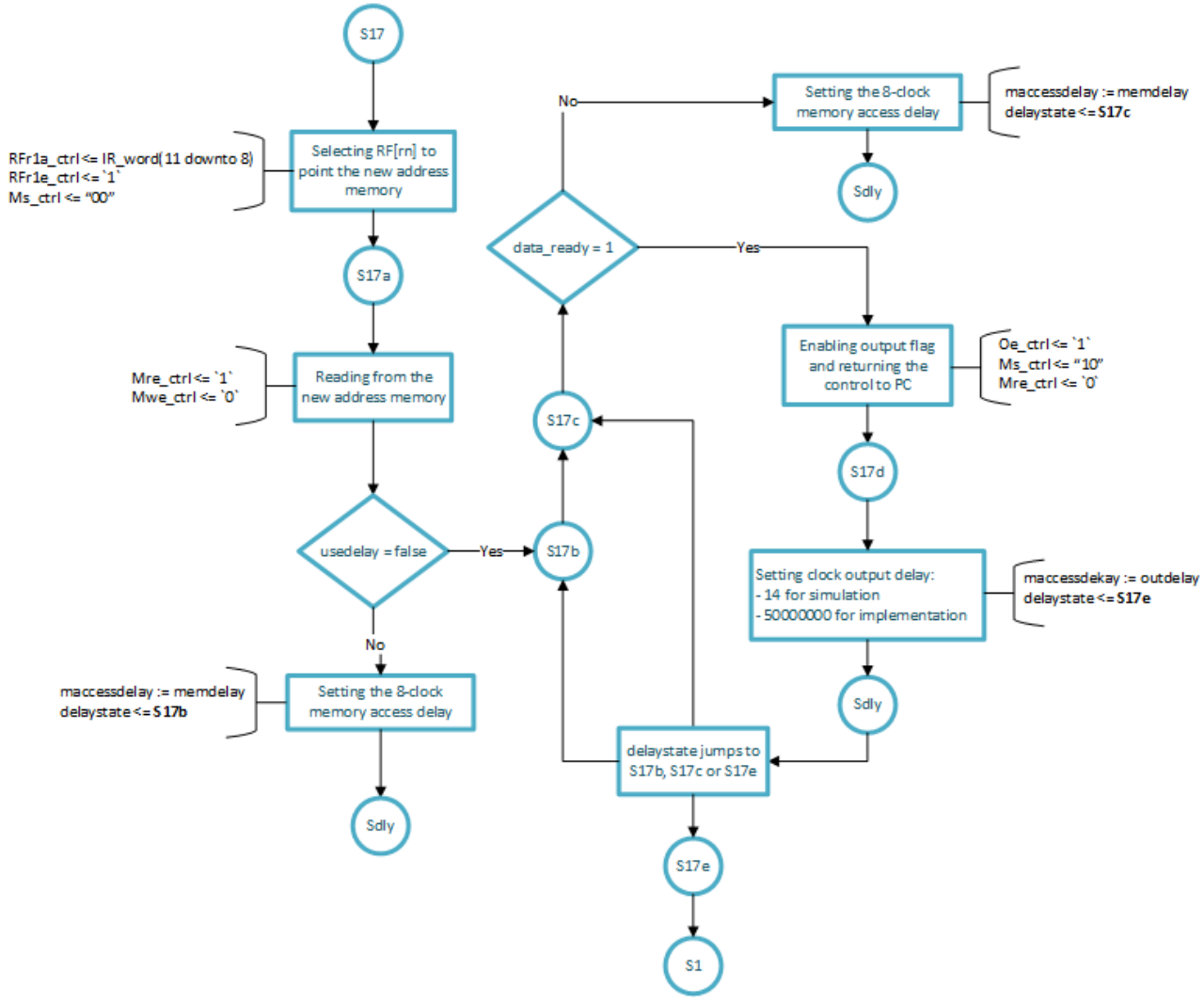


Figure 6: Indexed memory out state diagram

```
when S17 => -- output <- mem[RF[rn]]
  RFr1a_ctrl <= IR_word(11 downto 8);
  RFr1e_ctrl <= '1';
  Ms_ctrl <= "00"; -- selecting content of RFr1 for address memory
  state <= S17a;
when S17a =>
  Mre_ctrl <= '1'; -- read memory
  Mwe_ctrl <= '0';
  if usedelay = false then
    state <= S17b;
  else
    maccesdelay := memdelay;
    delaystate <= S17b;
    state <= Sdly;
  end if;
when S17b =>
  state <= S17c;
when S17c =>
  if data_ready = '1' then
    oe_ctrl <= '1';
    Ms_ctrl <= "10"; -- return
    Mre_ctrl <= '0';
    state <= S17d;
  elsif data_ready = '0' then
    maccesdelay:=memdelay;
    delaystate <= S17c;
    state <= Sdly;
  end if;
when S17d =>
  maccesdelay:=outdelay;
  delaystate <= S17e;
  state <= Sdly;
when S17e =>
  state <= S1;
```

Listing 6: Indexed memory out state vhdl

2.1.7 Reference system's instruction set

After the implementation of the instructions described before, a new instruction set was created which is shown in Table 2.

2.2 Cache implementation

2.2.1 Memory organization

The reference main memory size is 4 kbytes of 16 bits words (address width of 12 [1] bits) while the enhanced system requires that the memory be a two-level memory with a slow (main) memory size of 4kbytes with 64-bit lines and a total of 1024 lines and fast (cache) memory with 32 words of 16 bits organized in 8 lines of 4 words per line. Taking into account these requirements, the organization of the enhanced memory in contrast with the organization of the reference memory are shown in Fig. 7.

2.2.2 Data transfer

In order to execute a program or application, the CPU sends directions to the enhanced memory asking for data words to process and, hopefully, that information is in the cache memory which is very fast and can satisfy the CPU's processing speed needs. However, if the information is not inside the cache, then some mechanism must

[1] Review subsection 3.7 (Unexpected compilation error) to know the reason why we changed from 12-bit address to 11-bit address.

Reference Instruction Set		
Symbolic	**Operation Code**	**HDL**
mov1	'0000' = 0x0	RF[r_n]←mem[direct]
mov2	'0001' = 0x1	mem[direct]←RF[r_n]
mov3	'0010' = 0x2	mem[RF[r_n]]←RF[r_m]
mov4	'0011' = 0x3	RF[r_n]←imm
add	'0100' = 0x4	RF[r_n]←RF[r_n]+RF[r_m]
subt	'0101' = 0x5	RF[r_n]←RF[r_n]−RF[r_m]
jz	'0110' = 0x6	if RF[r_n]= 0:PC←imm
readm	'0111' = 0x7	out←mem[direct]
mult	'1000' = 0x8	RF[r_n]←RF[r_n]∗RF[r_m]
mov5	'1001' = 0x9	RF[r_m]←mem[RF[r_n]]
jne	'1010' = 0xA	if RF[r_n]≠RF[r_m]:PC←RF[r_o]
inc	'1011' = 0xB	RF[r_n]←RF[r_n]+1
mov6	'1100' = 0xC	RF[r_n]←RF[r_m]
outRF	'1101' = 0xD	out←RF[r_n]
halt	'1111' = 0xF	do nothing

Table 2: Reference system's instruction set

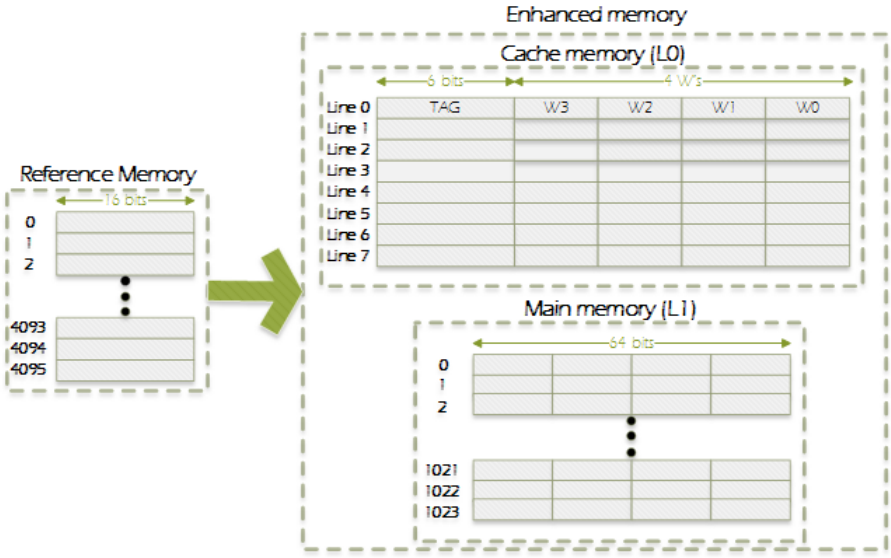


Figure 7: Memory organization of the reference and enhanced systems

be executed so that the required data gets to the CPU. This mechanism is the cache controller and its main purpose is make sure that the data required be inside the cache. Therefore, the cache controller is in charge of the data transfer between the enhanced memory and the CPU.

When the CPU asks for data that is not found in the cache memory, the cache controller retrieves the information from the main memory, but the transfer between the cache and the main memory is done by blocks. The reason to do that is based on the principle of locality, which assumes that most probably the next data word that will be required is inside the block that was transfered to the cache. In this way, there is no need to reach the main memory slowing down the program execution. Fig. 8 shows the transfers performed in the system.

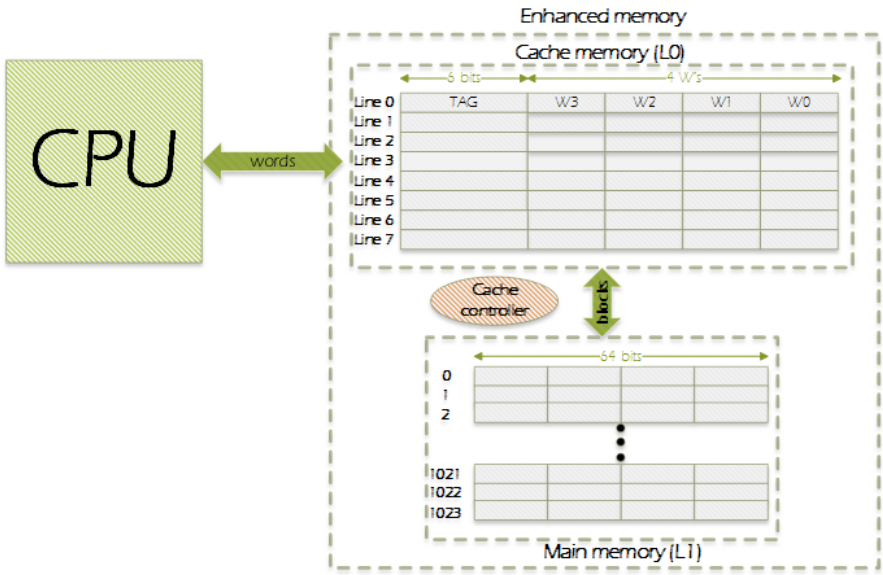


Figure 8: Data transfer diagram

2.2.3 Cache controller

Determining if the required data word is in the cache is, as well, one of functions that the cache controller must perform and it is achieved by analyzing the address direction set by the CPU. The first operation is to point to one of the eight lines of the cache using the line field (bits 3 to 5) extracted from the address. After that, the tag fields from the line and from the address (bits 6 to 12) are compared in order to know if they are the same. If they are equal then it is a hit and the data word in the line that is going to be transfered is selected by the word field in the address (bits 1 to 2). Otherwise, it is a miss, which means that the block in the selected line has to be moved to the main memory and a new block with the required data from the main memory has to take this position. This procedures takes eight clock cycles for purposes of simulation. Finally, once the data is already in the cache, the needed word is chosen with the word field and then it is sent to the CPU. Fig. 9 illustrates how the cache establishes if the required information is or is not in the cache memory and Fig. 10 and Fig. 11 shows the complete operation of the cache controller.

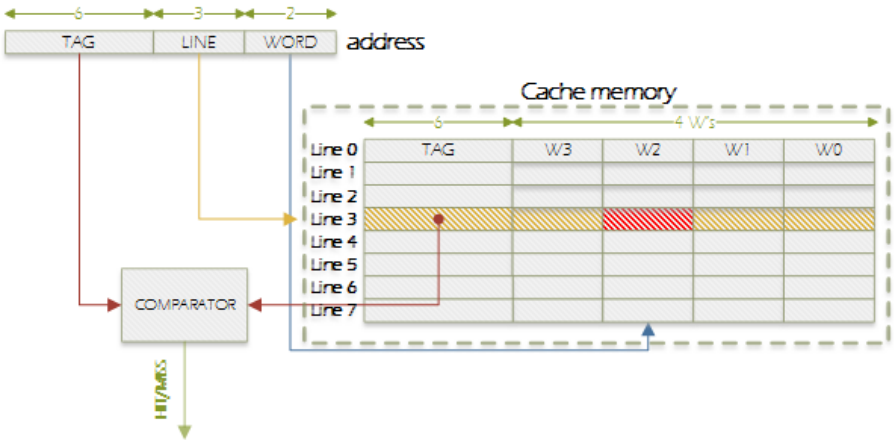


Figure 9: Cache controller operation

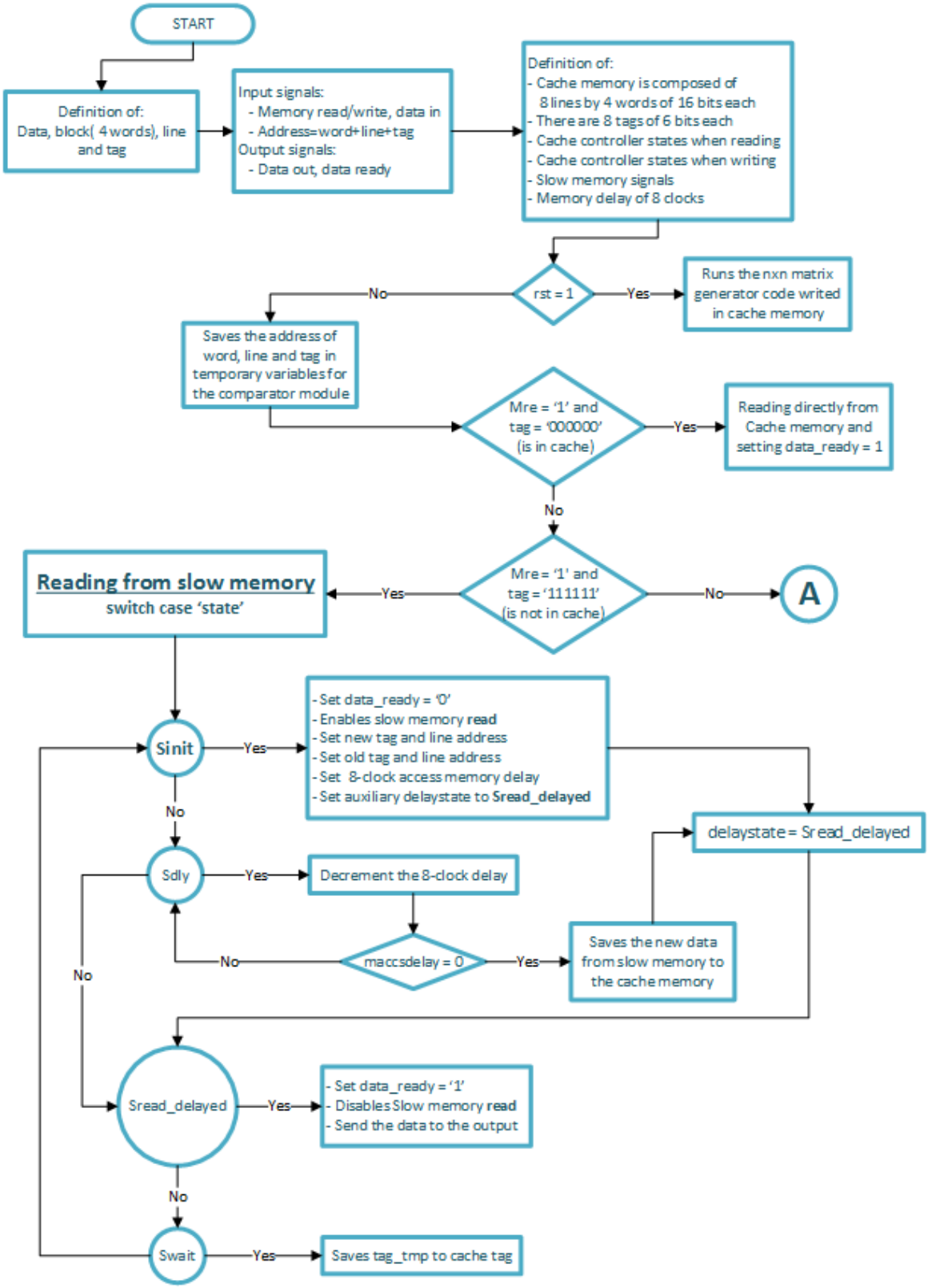


Figure 10: Cache controller initial and reading operation

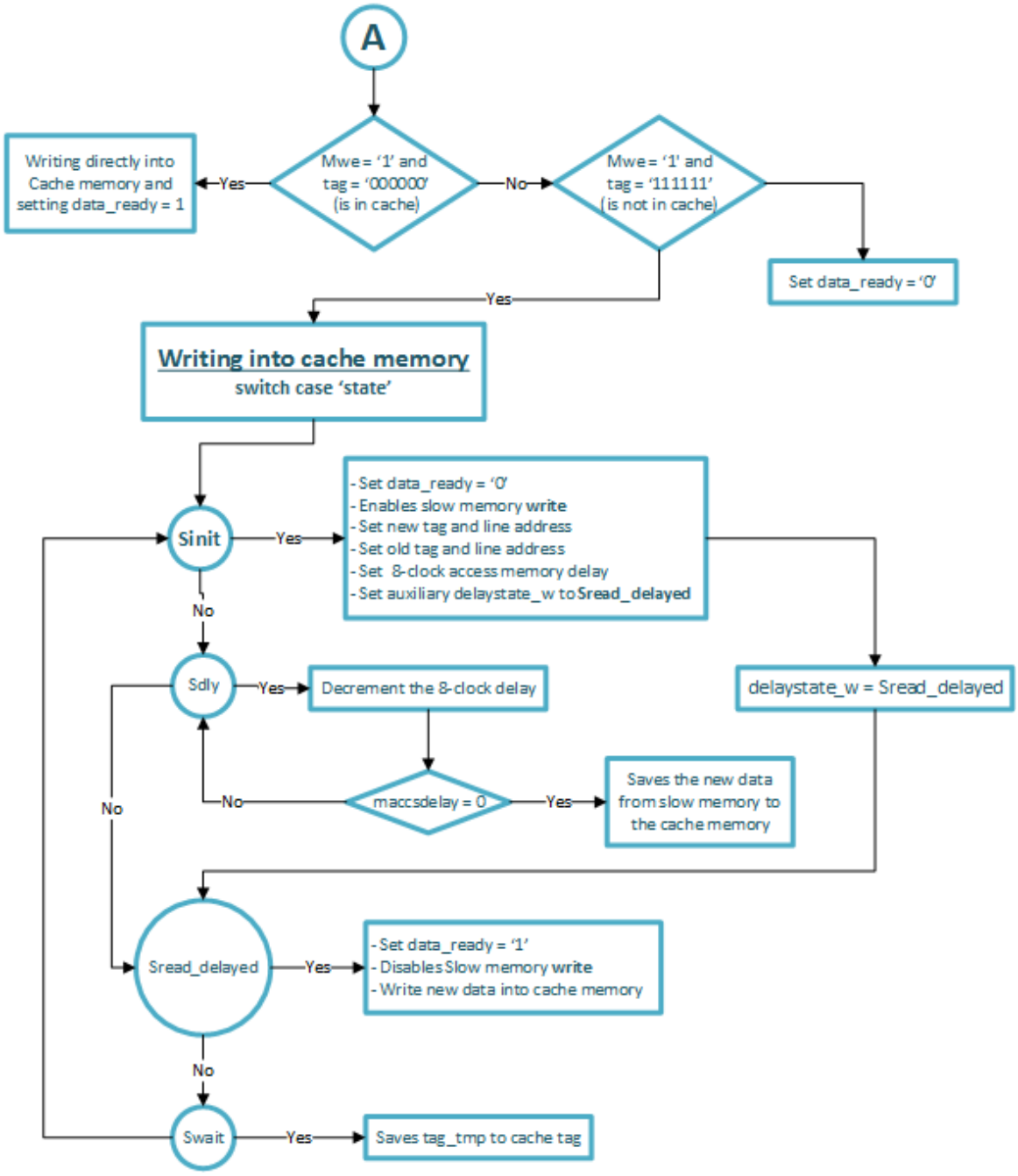


Figure 11: Cache controller writing operation

```
process(clock, rst, Mwe, Mre, address, data_in, state)
  variable word_tmp: std_logic_vector((block_size)-1 downto 0);
  variable line_tmp:std_logic_vector((line_size)-1 downto 0);
  variable tag_tmp: std_logic_vector((tag_size)-1 downto 0);
  variable maccesdelay: integer;
  begin
  if rst='1' then
    --  A&B nxn matrix generator (vhdl)

    data_out <= ZERO;
    state <= Sinit;
    state_W <= Sinit;
```

```
13    elsif (clock'event and clock = '1') then
14      word_tmp := address(block_size-1 downto 0);
15      line_tmp := address((block_size+line_size)-1 downto block_size);
16      tag_tmp := address((block_size+line_size+tag_size)-1 downto (block_size+line_size));
17
18      -- Reading directly from Cache memory
19      if (Mre ='1' and Mwe ='0' and tag_tmp = cache_tag(conv_integer(line_tmp))) then
20        data_ready <= '1';
21        data_out <= tmp_ram(conv_integer(line_tmp),conv_integer(word_tmp));
22
23      -- Reading from slow memory
24      elsif (Mre ='1' and Mwe ='0' and tag_tmp /= cache_tag(conv_integer(line_tmp))) then
25        case state is
26          when Sinit =>
27            data_ready <= '0';
28            SM_rw_enable <= '1';
29            new_address((tag_size+line_size)-1 downto (line_size)) <= tag_tmp;
30            new_address((line_size-1) downto 0) <= line_tmp;
31            old_address((tag_size+line_size)-1 downto (line_size)) <= cache_tag(conv_integer(
    line_tmp));
32            old_address((line_size-1) downto 0) <= line_tmp;
33            old_data <= tmp_ram(conv_integer(line_tmp),0) & tmp_ram(conv_integer(line_tmp),1) &
    tmp_ram(conv_integer(line_tmp),2) & tmp_ram(conv_integer(line_tmp),3);
34            maccesdelay:=memdelay;
35            delaystate <= Sread_delayed;
36            state <= Sdly;
37          when Sdly =>                -- Delay State
38            maccesdelay := maccesdelay-1;
39            if maccesdelay = 0 then
40              tmp_ram(conv_integer(line_tmp),0) <= new_data((data_width*(2**block_size))-1
    downto (data_width*(2**block_size-1)));
41              tmp_ram(conv_integer(line_tmp),1) <= new_data((data_width*(2**block_size-1))-1
    downto (data_width*(2**block_size-2)));
42              tmp_ram(conv_integer(line_tmp),2) <= new_data((data_width*(2**block_size-2))-1
    downto (data_width*(2**block_size-3)));
43              tmp_ram(conv_integer(line_tmp),3) <= new_data((data_width*(2**block_size-3))-1
    downto 0);
44              state <= delaystate;
45            else
46              state <= Sdly ;
47            end if;
48          when Sread_delayed =>
49            data_ready <= '1';
50            SM_rw_enable <= '0';
51            data_out <= tmp_ram(conv_integer(line_tmp),conv_integer(word_tmp));
52            state <= Swait;
53          when Swait =>
54            cache_tag(conv_integer(line_tmp)) <= tag_tmp;
55            state <= Sinit;
56        end case;
57
58      -- Writing into cache memory
59      elsif (Mwe ='1' and Mre = '0' and tag_tmp = cache_tag(conv_integer(line_tmp))) then
60        data_ready <= '1';
61        tmp_ram(conv_integer(line_tmp),conv_integer(word_tmp)) <= data_in;
62      elsif (Mwe ='1' and Mre = '0' and tag_tmp /= cache_tag(conv_integer(line_tmp))) then
63        case state_w is
64          when Sinit =>
65            data_ready <= '0';
66            SM_rw_enable <= '1';
67            new_address((tag_size+line_size)-1 downto (line_size)) <= tag_tmp;
68            new_address((line_size-1) downto 0) <= line_tmp;
69            old_address((tag_size+line_size)-1 downto (line_size)) <= cache_tag(conv_integer(
    line_tmp));
70            old_address((line_size-1) downto 0) <= line_tmp;
71            old_data <= tmp_ram(conv_integer(line_tmp),0) & tmp_ram(conv_integer(line_tmp),1) &
    tmp_ram(conv_integer(line_tmp),2) & tmp_ram(conv_integer(line_tmp),3);
72            maccesdelay:=memdelay;
```

```
73            delaystate_w <= Sread_delayed;
74            state_w <= Sdly;
75          when Sdly =>                  -- Delay State
76            maccesdelay := maccesdelay-1;
77            if maccesdelay = 0 then
78              tmp_ram(conv_integer(line_tmp),0) <= new_data((data_width*(2**block_size))-1
    downto (data_width*(2**block_size-1)));
79              tmp_ram(conv_integer(line_tmp),1) <= new_data((data_width*(2**block_size-1))-1
    downto (data_width*(2**block_size-2)));
80              tmp_ram(conv_integer(line_tmp),2) <= new_data((data_width*(2**block_size-2))-1
    downto (data_width*(2**block_size-3)));
81              tmp_ram(conv_integer(line_tmp),3) <= new_data((data_width*(2**block_size-3))-1
    downto 0);
82              state_w <= delaystate_w;
83            else
84              state_w <= Sdly ;
85            end if;
86          when Sread_delayed =>
87            data_ready <= '1';
88            SM_rw_enable <= '0';
89            tmp_ram(conv_integer(line_tmp),conv_integer(word_tmp)) <= data_in;
90            state_w <= Swait;
91          when Swait =>
92            cache_tag(conv_integer(line_tmp)) <= tag_tmp;
93            state_w <= Sinit;
94        end case;
95      else
96        data_ready <= '0';
97      end if;
98    end if;
99  end process;
100 Unit: memory_slow port map(clock,rst,SM_rw_enable,old_address,new_address,old_data,new_data);
```

Listing 7: Cache controller vhdl

2.3 Program design

2.3.1 Matrix generator

The dimension of the two matrices that we have to operate are 10x10, which means that every matrix has 100 elements. Therefore, we can set every single element manually or we can generate them automatically. Since it takes two instructions to store an element, we will need 400 instructions to store all the matrices elements. This last procedure is too tedious. Instead, we generated the matrices element with a short algorithm that produces a first matrix with even numbers and a second matrix with odd numbers interleaved with zeros so that the results do not exceed the maximum register value (65535). The matrices are the following:

$$A = \begin{bmatrix} 0 & 0 & 2 & 0 & 4 & \dots \\ 10 & 0 & 12 & 0 & 14 & \dots \\ 20 & 0 & 22 & 0 & 24 & \dots \\ 30 & 0 & 32 & 0 & 34 & \dots \\ 40 & 0 & 42 & 0 & 44 & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}_{10x10} , B = \begin{bmatrix} 1 & 0 & 3 & 0 & 5 & \dots \\ 11 & 0 & 13 & 0 & 15 & \dots \\ 21 & 0 & 23 & 0 & 25 & \dots \\ 31 & 0 & 33 & 0 & 35 & \dots \\ 41 & 0 & 43 & 0 & 45 & \dots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{bmatrix}_{10x10}$$

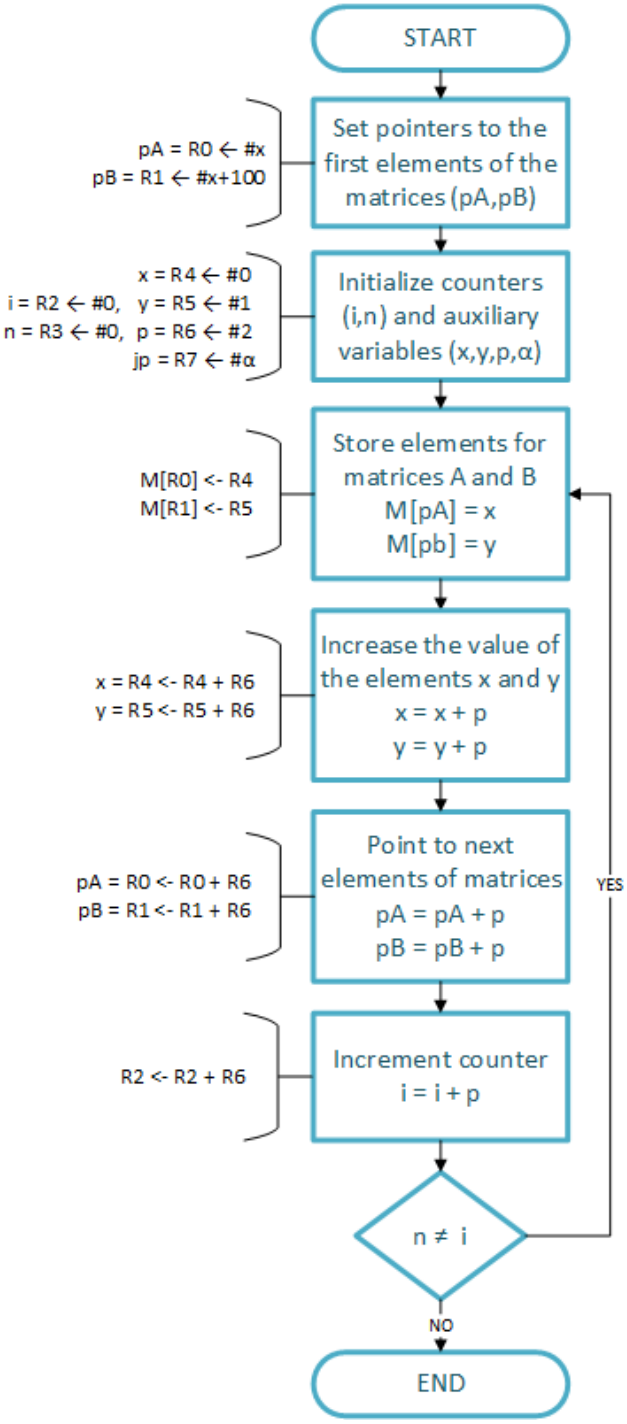


Figure 12: Matrix generation algorithm

```
--  ------------------------------------------------
--  -- A&B nxn matrix generator
--  0 => x"3064" & x"31C8" & x"3200" & x"3364",
--  -- 0:  R0 <- #100   :pA = 100
--  -- 1:  R1 <- #200   :pB = 200
--  -- 2:  R2 <- #0   :i = 0
--  -- 3:  R3 <- #100   :n = 100, total matrix elements
--  1 => x"3400" & x"3501" & x"3602" & x"3708",
```

```
--  -- 4:  R4 <- #0   :x = 0
--  -- 5:  R5 <- #1   :y = 1
--  -- 6:  R6 <- #2   :p = 2
--  -- 7:  R7 <- #8(jump position)
--  2 => x"2040" & x"2150" & x"4460" & x"4560",
--  -- 8:  M[R0] <- R4  :M[pA] = x
--  -- 9:  M[R1] <- R5  :M[pb] = y
--  -- 10: R4 <- R4 + R6  :x = x + p
--  -- 11: R5 <- R5 + R6  :y = y + p
--  3 => x"4060" & x"4160" & x"4260" & x"A237",
--  -- 12: R0 <- R0 + R6  :pA = pA + p
--  -- 13: R1 <- R1 + R6  :pB = pB + p
--  -- 14: R2 <- R2 + R6  :i = i + p
--  -- 15: R2~=R3:PC<-[R7]  goto([R7]) if R2~=R3
--
-----------------------------------------------
-- A&B nxn matrix generator
-- tag 0              -- line 0, words 0-3
cache_tag(0) <= "000000"; tmp_ram(0,0) <= x"3064";tmp_ram(0,1) <= x"31C8";tmp_ram(0,2) <= x"
    3200";tmp_ram(0,3) <= x"3364";
-- tag 1              -- line 1, words 4-7
cache_tag(1) <= "000000"; tmp_ram(1,0) <= x"3400";tmp_ram(1,1) <= x"3501";tmp_ram(1,2) <= x"
    3602";tmp_ram(1,3) <= x"3708";
-- tag 2              -- line 2, words 8-11
cache_tag(2) <= "000000"; tmp_ram(2,0) <= x"2040";tmp_ram(2,1) <= x"2150";tmp_ram(2,2) <= x"
    4460";tmp_ram(2,3) <= x"4560";
-- tag 3              -- line 3, words 12-15
cache_tag(3) <= "000000"; tmp_ram(3,0) <= x"4060";tmp_ram(3,1) <= x"4160";tmp_ram(3,2) <= x"
    4260";tmp_ram(3,3) <= x"A237";
-- tag 4              -- line 4, words 16-19
cache_tag(4) <= "111111";
-- tag 5              -- line 5, words 20-23
cache_tag(5) <= "111111";
-- tag 6              -- line 6, words 24-27
cache_tag(6) <= "111111";
-- tag 7              -- line 7, words 28-31
cache_tag(7) <= "111111";
```

Listing 8: Matrix generator vhdl

2.3.2 Matrix multiplication

Considering that the two matrices are stored consecutively, the following algorithm performs the matrix multiplication:

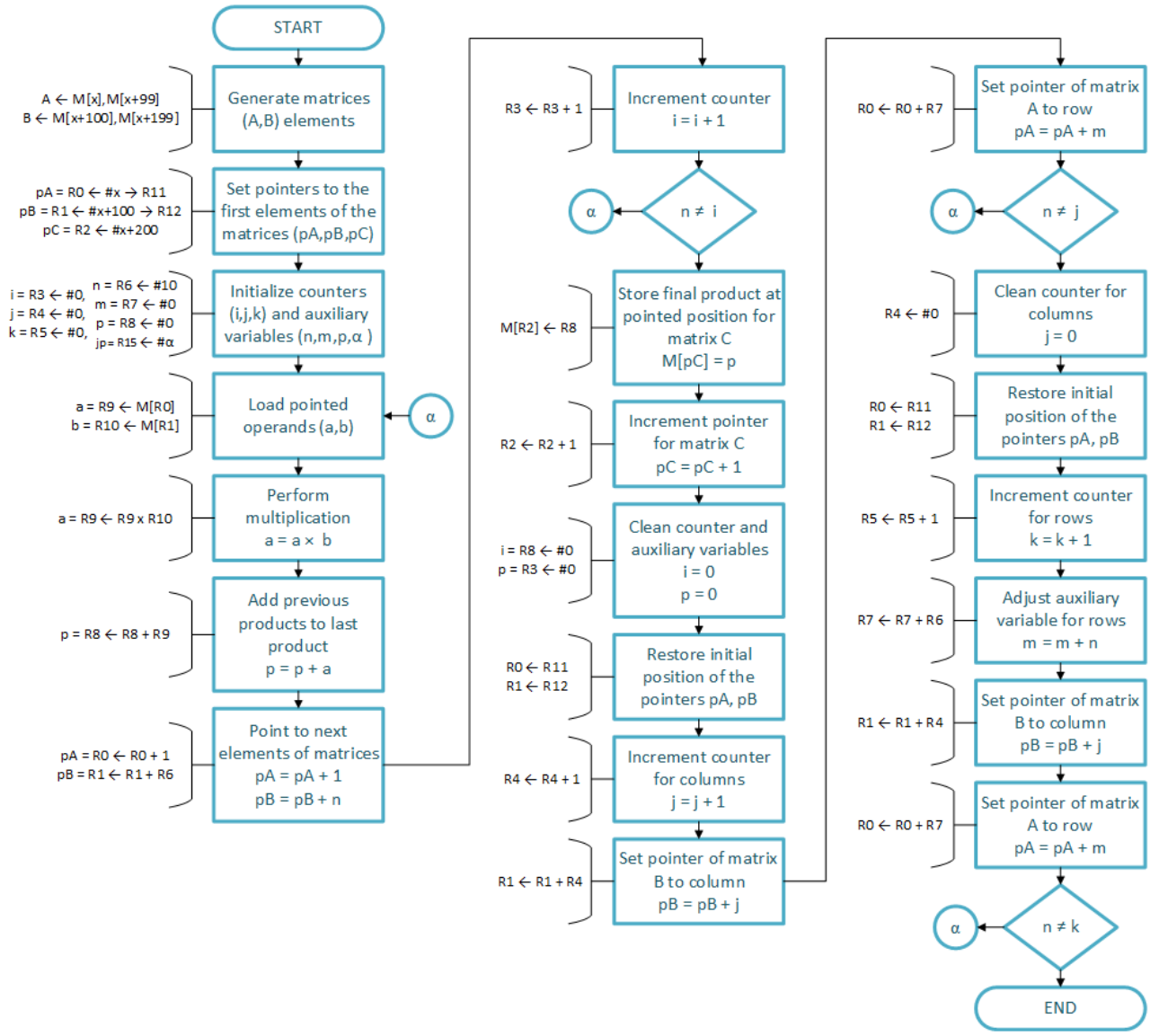


Figure 13: Multiplication algorithm

```
-- Matrix multiplication A*B=C
4 => x"3064" & x"3B64" & x"31C8" & x"3CC8",
-- 16: R0  <- #x=100  :pA
-- 17: R11 <- #x=100
-- 18: R1  <- #x=200  :pB
-- 19: R12 <- #x=200
5 => x"C210" & x"4200" & x"3300" & x"3400",
-- 20: R2 <- R1   :#200
-- 21: R2 <- R2+R0  :#200+100=300 :pC
-- 22: R3 <- #0   :i=0
-- 23: R4 <- #0   :j=0
6 => x"3500" & x"360A" & x"3700" & x"3800",
-- 24: R5 <- #0   :k=0
-- 25: R6 <- #10    :n=10, column number
-- 26: R7 <- #0   :m=0
-- 27: R8 <- #0   :p=0
7 => x"3F1D" & x"9090" & x"91A0" & x"89A0",
-- 28: R15 <- #29     :jp (#alpha)
```

```
-- 29: R9  <- M[R0]   :a
-- 30: R10 <- M[R1]   :b
-- 31: R9 <- R9*R10   :a=a*b
8 => x"4890" & x"B000" & x"4160" & x"B300",
-- 32: R8 <- R8+R9    :p=p+a
-- 33: R0 <- R0+1   :pA=pA+1
-- 34: R1 <- R1+R6  :pB=pB+n
-- 35: R3 <- R3+1   :i=i+1
9 => x"A63F" & x"2280" & x"B200" & x"3800",
-- 36: R6~=R3:PC<-[R15]  goto([R15]) if R6~=R3  :n~=i
-- 37: M[R2] <- R8  :M[pC]=p, store final product
-- 38: R2 <- R2+1   :pC=pC+1
-- 39: R8 <- #0   :i=0
10 => x"3300" & x"C0B0" & x"C1C0" & x"B400",
-- 40: R3 <- #0   :p=0
-- 41: R0 <- R11    :Restore init pos of pA
-- 42: R1 <- R12    :Restore init pos of pB
-- 43: R4 <- R4+1   :j=j+1
11 => x"4140" & x"4070" & x"A64F" & x"3400",
-- 44: R1 <- R1+R4  :pB=pB+j
-- 45: R0 <- R0+R7  :pA=pA+m
-- 46: R6~=R4:PC<-[R15]  goto([R15]) if R6~=R4  :n~=j
-- 47: R4 <- #0   :j=0
12 => x"C0B0" & x"C1C0" & x"B500" & x"4760",
-- 48: R0 <- R11    :Restore init pos of pA
-- 49: R1 <- R12    :Restore init pos of pB
-- 50: R5 <- R5+1   :k=k+1, inc row counter
-- 51: R7 <- R7+R6  :m=m+n
13 => x"4140" & x"4070" & x"A65F" & x"3200",
-- 52: R1 <- R1+R4  :pB=pB+j
-- 53: R0 <- R0+R7  :pA=pA+m
-- 54: R6~=R5:PC<-[R15]  goto([R15]) if R6~=R5  :n~=k
-- 55: R2 <- #0     :i=0

-- Output matrix C
14 => x"3364" & x"343D" & x"3564" & x"30C8",
-- 56: R3 <- #100     :n=100, total matrix elements
-- 57: R4 <- #61(jump position)
-- 58: R5 <- #100
-- 59: R0 <- #200
15 => x"4050" & x"D000" & x"B000" & x"B200",
-- 60: R0 <- R0+R5    :#300 pA=300
-- 61: output<- M[R0]
-- 62: R0 <- R0+1     :pA=pA+1
-- 63: R2 <- R2+1     :i=i+1
16 => x"A234" & x"F000" & x"0000" & x"0000",
-- 64: R2~=R3:PC<-[R4]  goto([R4]) if R2~=R3
-- 65: halt
```

Listing 9: Matrix multiplication vhdl

3 Results

3.1 Matlab code

In order to verify the simulation and implementation results, the following code in `matlab` allows us to view the 100 elements of the resultant matrix $A * B = C$, Fig. 14.

```
clear all; clc; close all;
n=10; % nxn, n value of matrix
x=0; y=1;
for i=1:n
  for j=1:n
    if mod(j, 2) == 0   % j is even
```

```
      A(i,j)=0;
      B(i,j)=0;
    else                    % j is odd
      A(i,j)=x;
      x=x+2;
      B(i,j)=y;
      y=y+2;
    end
  end
end
A
B
C=A*B
```

Listing 10: Matlab multiplication.

```
A =

     0     0     2     0     4     0     6     0     8     0
    10     0    12     0    14     0    16     0    18     0
    20     0    22     0    24     0    26     0    28     0
    30     0    32     0    34     0    36     0    38     0
    40     0    42     0    44     0    46     0    48     0
    50     0    52     0    54     0    56     0    58     0
    60     0    62     0    64     0    66     0    68     0
    70     0    72     0    74     0    76     0    78     0
    80     0    82     0    84     0    86     0    88     0
    90     0    92     0    94     0    96     0    98     0


B =

     1     0     3     0     5     0     7     0     9     0
    11     0    13     0    15     0    17     0    19     0
    21     0    23     0    25     0    27     0    29     0
    31     0    33     0    35     0    37     0    39     0
    41     0    43     0    45     0    47     0    49     0
    51     0    53     0    55     0    57     0    59     0
    61     0    63     0    65     0    67     0    69     0
    71     0    73     0    75     0    77     0    79     0
    81     0    83     0    85     0    87     0    89     0
    91     0    93     0    95     0    97     0    99     0


C =

        1220           0        1260           0        1300           0        1340           0        1380           0
        3270           0        3410           0        3550           0        3690           0        3830           0
        5320           0        5560           0        5800           0        6040           0        6280           0
        7370           0        7710           0        8050           0        8390           0        8730           0
        9420           0        9860           0       10300           0       10740           0       11180           0
       11470           0       12010           0       12550           0       13090           0       13630           0
       13520           0       14160           0       14800           0       15440           0       16080           0
       15570           0       16310           0       17050           0       17790           0       18530           0
       17620           0       18460           0       19300           0       20140           0       20980           0
       19670           0       20610           0       21550           0       22490           0       23430           0
```

Figure 14: Matlab multiplication.

3.2 Compilation instructions

Because we are using the same project to test the upgraded performance of the direct mapped cache system vs the use of simple memory system, we need to change some parameters to compile both cases.

Cache and simple memory simulation/implementation:

1. Comment/uncomment in `SimpleCompArch.vhd` (line `85-86`) as the following code.

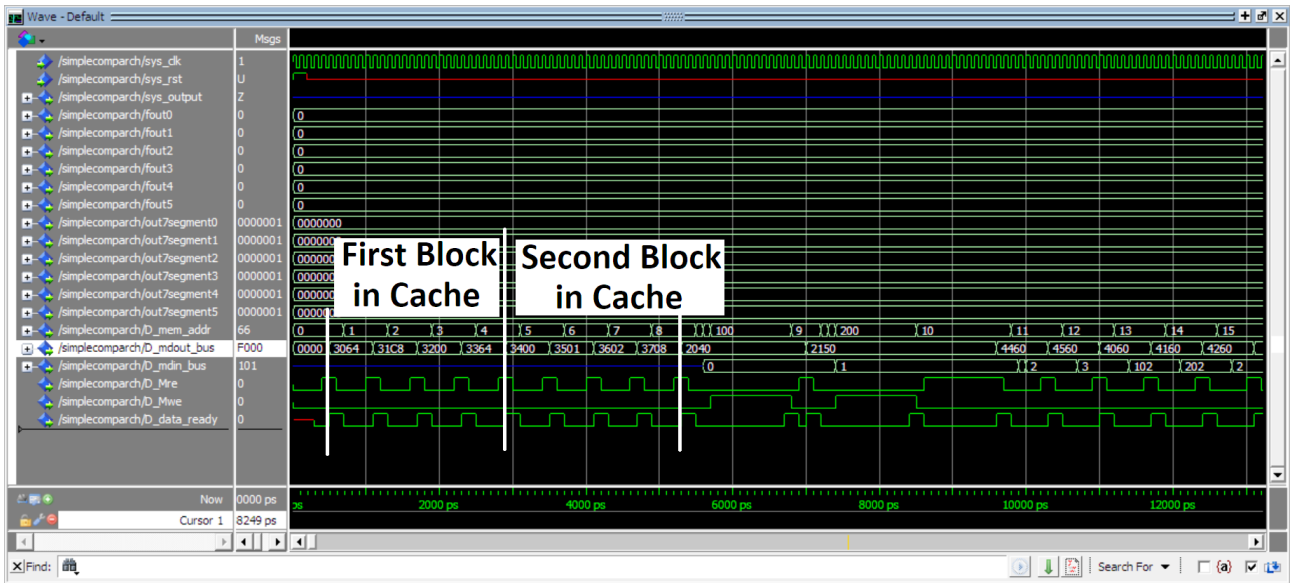
```
Unit2: memory port map(sys_clk,sys_rst,Mre,Mwe,mem_addr,mdin_bus,mdout_bus,data_ready);
--Unit5: memory_simple port map(sys_clk,sys_rst,Mre,Mwe,mem_addr,mdin_bus,mdout_bus,
  data_ready);
```

2. Edit `controller.vhd` (line `52-53`) with the following values,

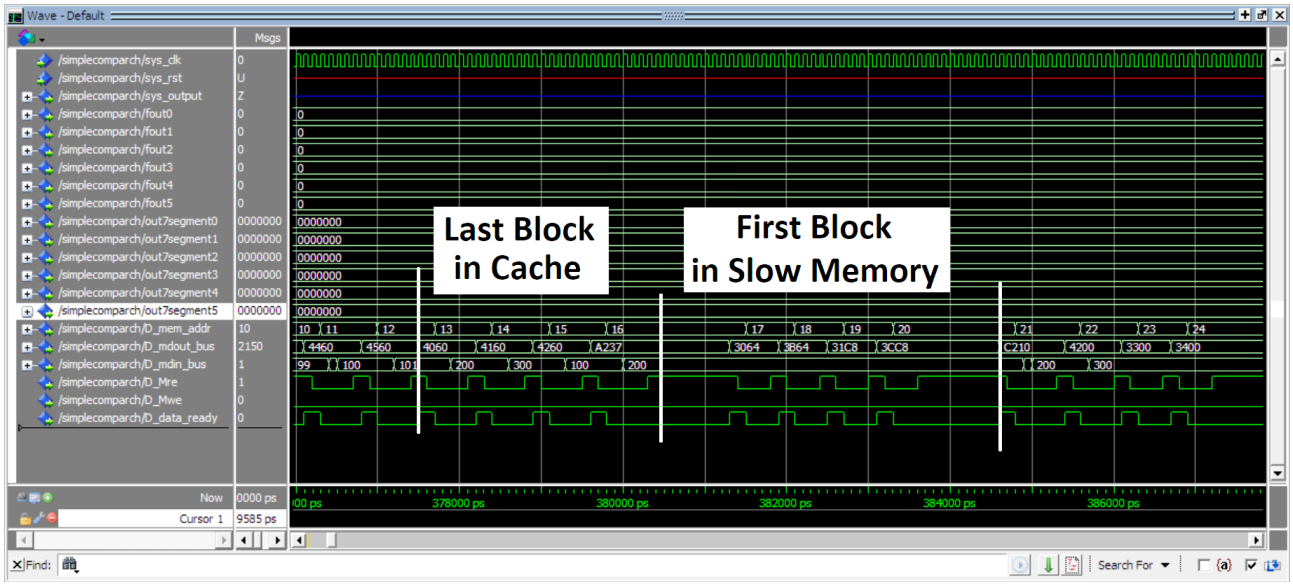
```
1   constant outdelay: integer := 50000000; -- 50000000 = 1sec for implementation, 14 for
      simulation
2   signal usedelay: boolean := false;     -- false for cache memory, true for memory_slow
```

3.3 Cache memory simulation

Fig. 15a shows the initialization of the whole process, at the same, it can be seen the execution of the First and Second Block in Cache memory, Fig. 15b shows the execution of the Last Block in Cache and the First Block in Slow Memory, during that time the cache controller enters in execution.



(a) Initialization of simulation.

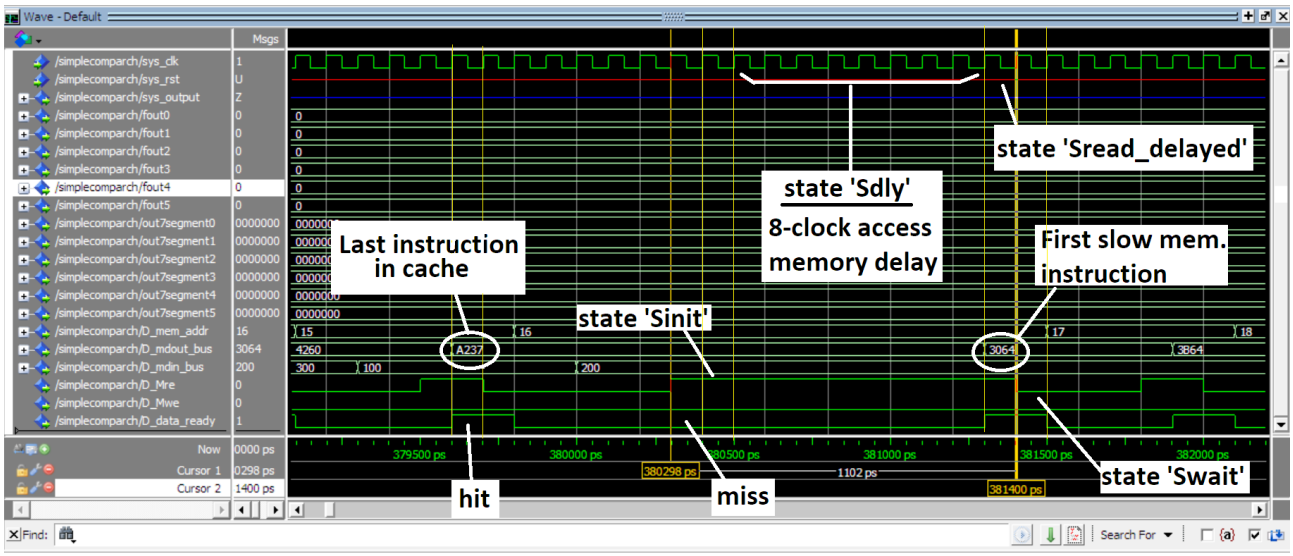


(b) Last Block executed in Cache memory and first Block executed in Slow Memory.

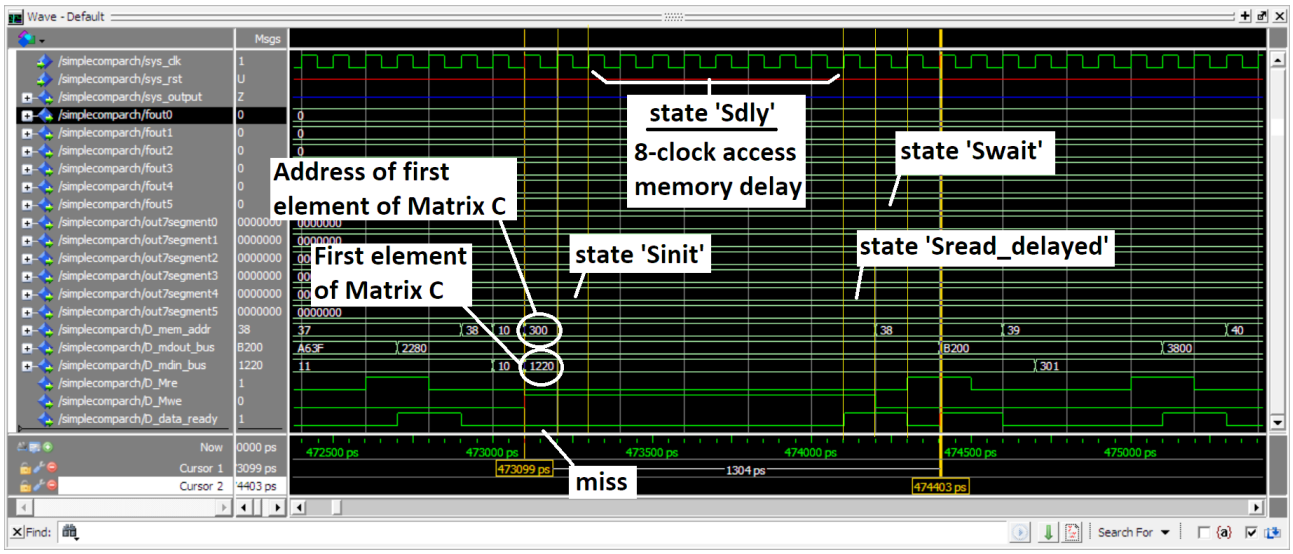
Figure 15: Cache memory simulation.

3.4 Cache controller execution

Fig. 16a represents in detail the states of **Reading from slow memory** process, and Fig. 16b describes the **Writing into cache memory** process.



(a) Reading from slow memory.

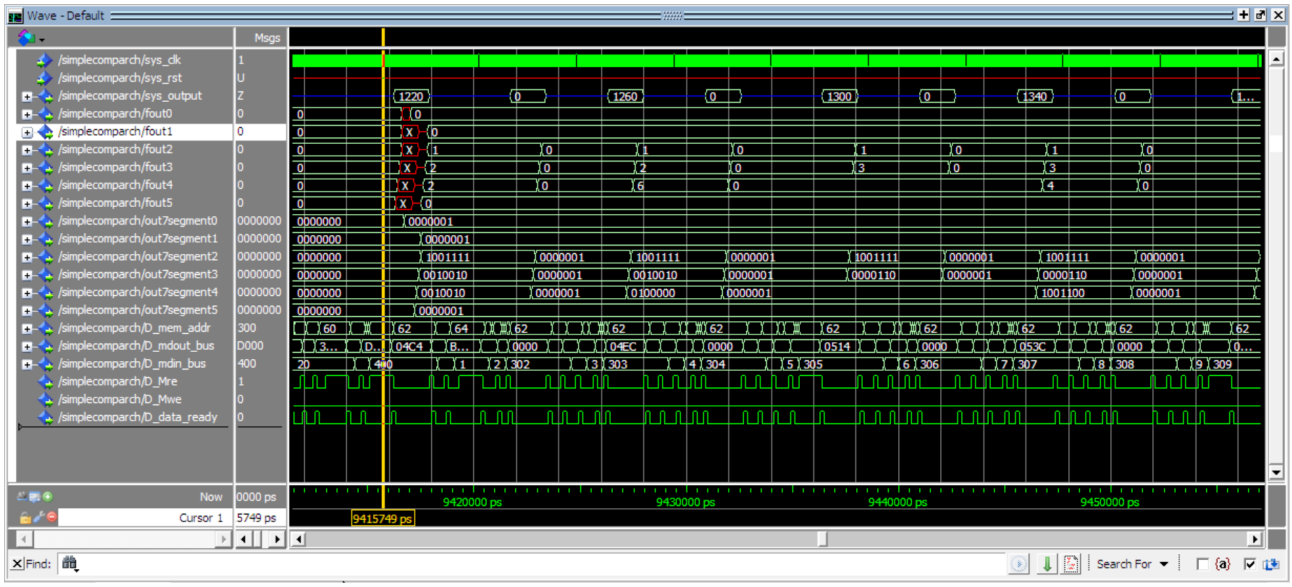


(b) Writing into cache memory.

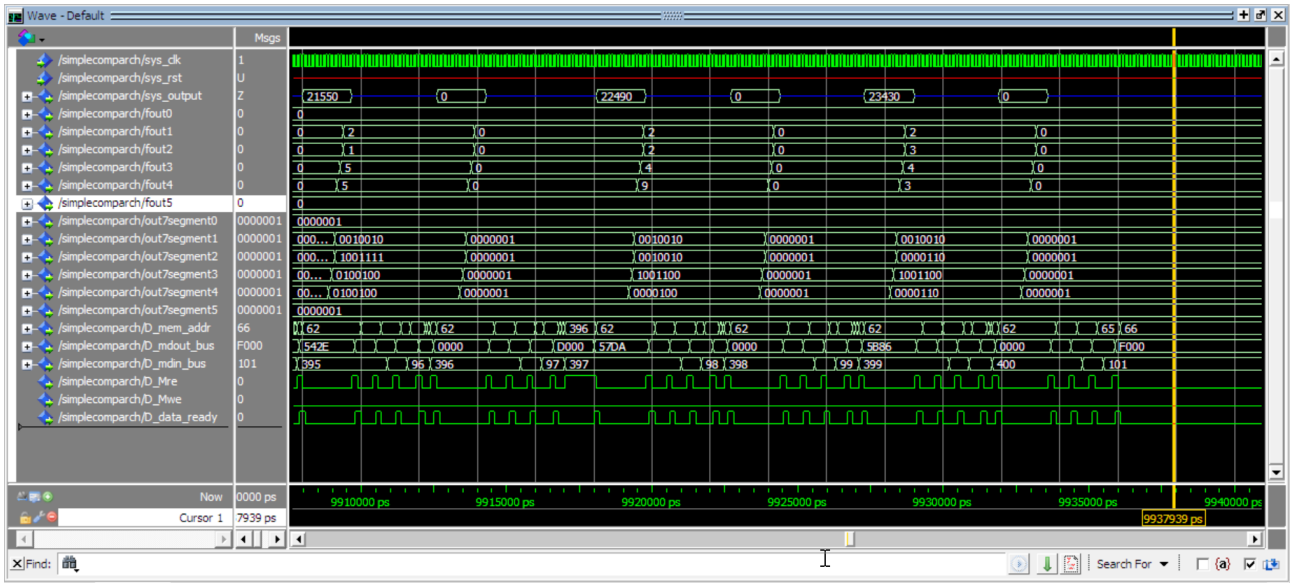
Figure 16: Cache controller execution.

3.5 Cache memory timing

Fig. 17b shows that after approx. **9.41 ns** the multiplication process ends and the first element of resultant Matrix C is displayed, and finally, Fig. 17c displays approx. **9.93 ns** was used to complete all the **multiplication and displaying process**.



(a) First element of resultant Matrix C to be displayed in the 7-segment displays.

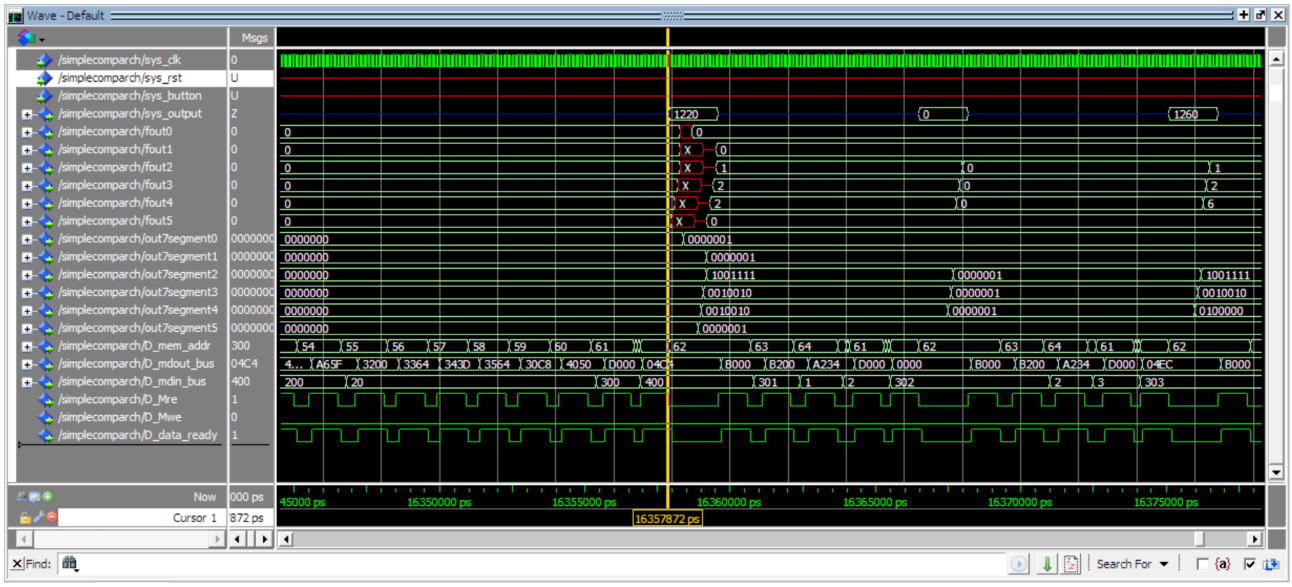


(b) Last instruction and last displayed element.

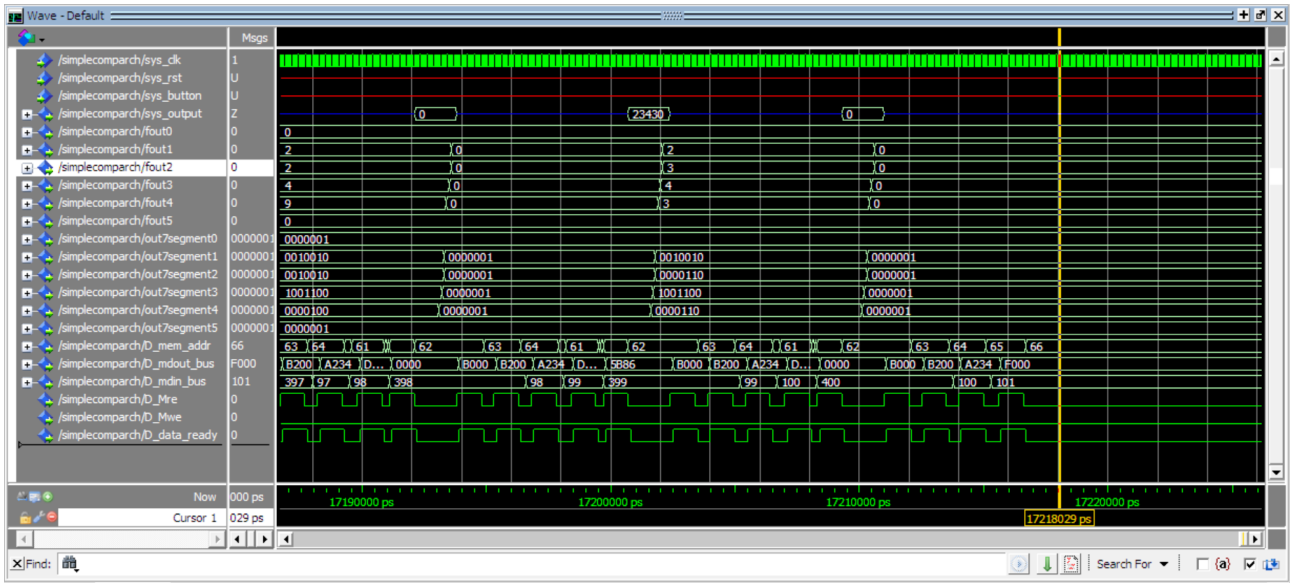
Figure 17: Cache memory simulation.

3.6 Simple memory timing

Fig. 18b shows that after approx. **16.36 ns** the multiplication process ends and the first element of resultant Matrix C is displayed, and finally, Fig. 18c displays approx. **17.22 ns** was used to complete all the **multiplication and displaying process**.



(a) First element of resultant Matrix C to be displayed in the 7-segment displays.

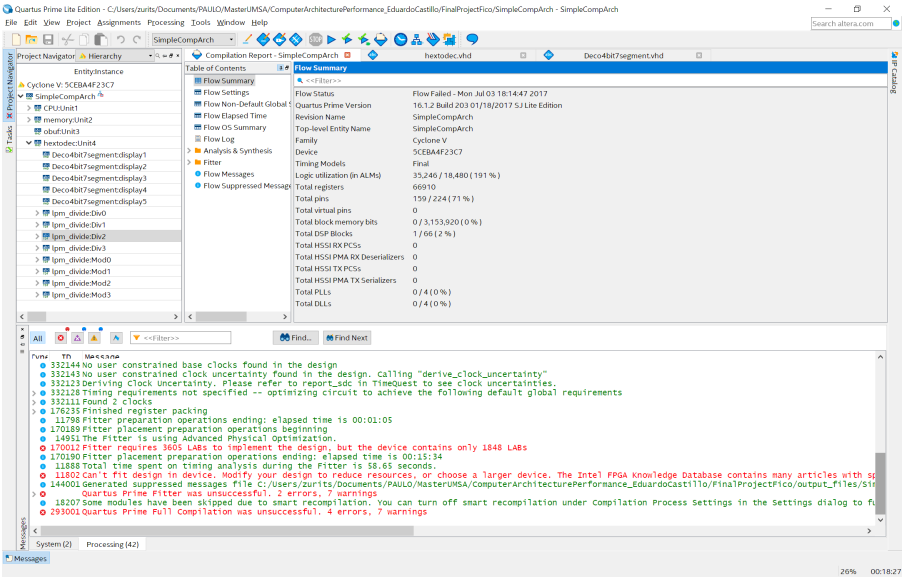


(b) Last instruction and last displayed element.

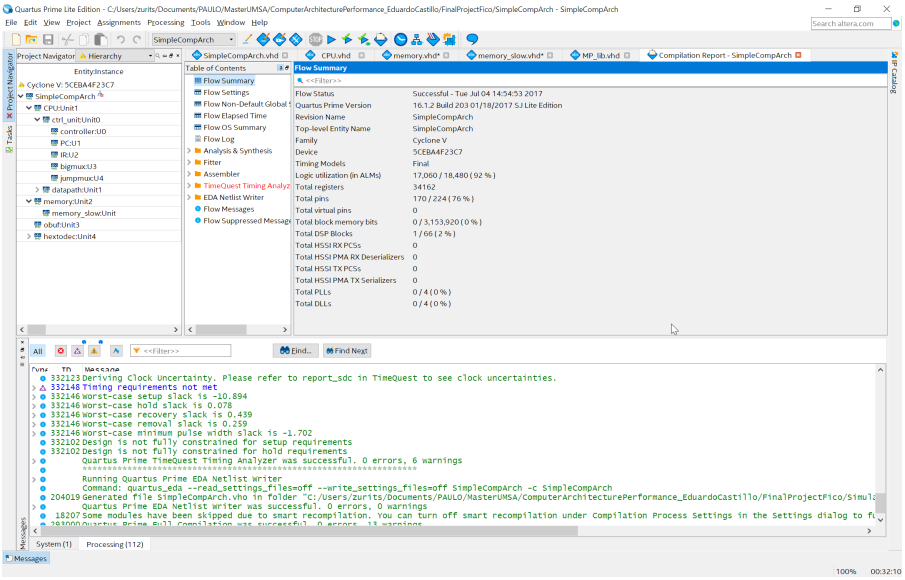
Figure 18: Simple memory simulation.

3.7 Unexpected compilation error

During the compilation process an unexpected error occurred, `Quartus` displayed an over-sized `Logic utilization (in ALMs` of 191% that didn't allow us to fit the project into the `DE0-CV` board, Fig. 19a, so, fitting `Logic utilization` at 92% we reduce the 12-bit address to 11 bits by reducing the `tag size` from 7 bits to 6 bits.



(a) 12-bit address compilation.



(b) 11-bit address compilation.

Figure 19: Unexpected compilation error

4 Conclusions

The design, simulation and implementation of Direct-mapping cache memory system in VHDL was accomplished and compared it to a reference system without cache memory (simple memory) which is resumed in table 3, that reflects about 50% of increased performance and can be visualized in Fig. 17b and Fig. 18.

Simple memory (ns)	Direct mapped cache memory (ns)
17.22	9.93

Table 3: Timing simulation results.

For simulation and implementation purpose, section 3.2 established the parameters to compile correctly in order to visualize the results for `ModelSim` and for the `DE0-CV` board.

Finally, a video was edited to visualize the 100 elements of the resultant Matrix C where you can find attached to this .pdf or in the following link `https://youtu.be/Zvd96RGTEjk`.