Fully Associative Mapped Cache Memory System

David Murphy, Brent Simmons, Marc-Andre Couturier

University of New Brunswick Fredericton

ECE 3242

Winter 2016

Table of Contents

Abstract

The scope of this document is to describe in a concise way, the design, implementation and benchmarking of two systems. The first system, reference, is partially provided by the professor of the course. This system needs to be modified to meet the reference system requirements. The second system, enhanced, must implement a fully associative mapped cache to the reference. The two architectures must then be benchmarked to provide concrete evidence of: both architectures behaving as expected and to quantify the performance difference between the two architectures.

Problem Statement

We were tasked to design, simulate and implement a fully associative cache memory system in VHDL using Altera Quartus II and to compare it to a reference system without cache memory to verify the performance enhancement.

Reference System

This system is meant to be the main building block of the cached system as well as a benchmark to assess the improvement achieved by the cached system.

Specifications

i. Main memory size should be 4kbytes of 16 bits words (address width of 12 bits). See appendix for detailed steps.

ii. The speed of memory module is controlled by the memory clock, which has to be changed to 1/8 of the CPU clock.

iii. System modification will require changes to the microprocessor module (instantiate the new memory module and new address width) and to the controller module (to deal with the new memory timing).

Design

The m9k memory was implemented as indicated in the project manual provided. We have used the ".mif" file format to program the contents of the memory (See appendix – Matrix_addition.mif). A for loop was used in the top level entity to reduce the memory clock to $1/8^{th}$ of the system clock. Delays had to be introduced in the controller as to assure memory was read and write properly (See appendix – controller.vhd). That was necessary because of the assumption that the original systems' memory would read/write in only 1 clock cycle. However, the m9k memory needs 1 extra clock cycle to operate properly. The system was then tested and proper behavior was observed.

The m9k memory does operate at $1/8^{th}$ the speed of the system clock, but it is always running. This means that the memory access time has the potential to take a variable number of clock cycles, but this kind of memory access is the same for both the reference and enhanced system. In the code for memory access, a memory ready flag is set to observe when the memory has completed reading and writing. This memory ready flag waits for two rising edges as the m9k memory takes this long to process the command. Therefore, the time it takes to access memory depends on when the read and write flags are triggered. Since the memory ready flag could be triggered at any time relative to the $1/8^{th}$ memory clock, the range of cycles memory access can take is between 9 and 15 cycles per access. This is true for both the reference and enhanced systems.

Note. The range of cycles memory access can take is assumed to effect both systems the same as we are simulating the exact same code on the systems. The number of cycles is assumed to average to the same memory access time in both systems.

Matrix addition

The matrix addition program implements the m9k memory module with the required operations to perform a 5 x 5 matrix addition. Two new operational codes were implemented. These were implemented to provide the necessary functionality to do matrix addition on the system. The two operational codes are code 8 and 9. (See appendix – MP_lib.vhd).

1. Op-Code 0x8: It is called mov5 and will read the memory location specified in a register and write the information read into a register. The pseudo code for the operation looks like this: RF[r2] <= mem[RF[r1]].
2. Op-Code 0x9: It is called jz2 and will jump to a memory address regardless of the value of RF[1].

The organization of the instruction set is shown as :

OP CODE	r1	r2	Not used

Enhanced System

This architecture was implemented using the reference system as a base. It is to be tested using the same benchmark that was used to test the reference system. The cache system is expected to provide faster execution time compared to the reference system.

Specifications

i. Main memory size should be 4kbytes of 16 bits words (address width of 12 bits).
ii. Memory should be implemented with M9K units as before.
iii. Cache size should be 32 words of 16 bits (8 lines of 4 words).
iv. Mapping scheme: fully associative.
v. Writing scheme: write back.
vi. CPU will not access main memory directly but through the cache.

Design

Cache. The cache block was implemented in three parts: the cache controller, SRAM and TRAM (See appendix – TRAM.vhd, SRAM.vhd, and cache_controller.vhd). The cache controller coordinated all behaviors exhibited by the cache block. The TRAM holds the values of the tags' of each line present in cache. The SRAM holds the cached data. The general structure described above is shown in Figure 1.

Cache controller. The cache controller is the interface between the CPU and its memory. All memory accesses are handled by the cache controller. When memory access is needed, the controller in the CPU waits and triggers a flag for memory to be accessed. The cache controller processes the request and then triggers a flag that memory is ready to be used by the CPU.

The cache controller has finite state machine. This is the way it differentiates between the incoming read/write signals and whether it can access cache or main memory. The following is a description of the state machine implemented in the cache controller. When a memory access request is made, the state machine starts. If memory is not needed, the cache controller resets to state S0.

State S0 uses the top ten bits, the tag, to read from the TRAM tag table to check if the tag is there. A cache hit flag is set based upon finding the tag in TRAM; a cache hit is logical '1' and a cache miss is logical '0'.

State S1 decides what memory to access (cache or main) and what the operation is (read or write). This is based off cache hit flag and the read/write enable operations.

Cache Hit. If the cache hit flag is set logical '1', then the tag is found in cache. The proper read/write operations are set and the cache memory is accessed. The state goes to S2. State S2 is a wait state that allows for the operation to be performed on the cache memory and, importantly, signals that memory is ready for the CPU to use. S2 goes to S0 to restart the process again.

Cache Miss. If the cache hit flag is set to logical '0', then the tag is not in cache. First, the tag to be replaced is checked to see if it is 'dirty'; this check is to see if any of the data contained in the tag has been written to/altered. If it has, this has to be written back to main memory first. If not, main memory can be accessed normally.

Main Memory Read. On a cache miss and the tag is not 'dirty', this means that a block of main memory needs to be brought into cache memory. This triggers a read operation and the state is called S_mem1. There are two wait states that allow for the memory to be read called S_mem1b and S_mem1c. Then state S_mem2 which writes the data obtained from memory into the cache. This state then goes to S0 to repeat the process. A cache hit should be triggered at this time.

Main Memory Write. On a cache miss and the tag is 'dirty', this means that a block of main memory needs to first written to memory and then the new data needs to be brought into cache memory. The state main_write_state is entered. This waits until for main memory and then passes to main_write_stateb which allows for the finishing of the write operation. This state then goes to S0 to repeat the process. A main memory read should be triggered at this time.

Replacement policy. A First In, First Out policy (FIFO) is used when replacement is necessary. This policy comes into effect when the tag that the memory access needs in not in TRAM i.e. a cache miss. The cache controller keeps track of the index of the cache line to be replaced. It replaces both the tag in the TRAM table and the corresponding cache line in SRAM.

Optimizations. A different replacement policy could have been used. The FIFO policy was chosen as it was the simplest to implement. Other algorithms that incorporate the 'dirty' bit or how often specific tags are reference could be used as well.

The way the write back is implemented could be improved upon as well. If it executed in parallel with other instructions, this would cut down on the time it takes to access main memory within the cache.

A possible enhancement of both systems could be achieved through the use of an enable on the m9k memory. This would allow for a constant instead of the variable access time.

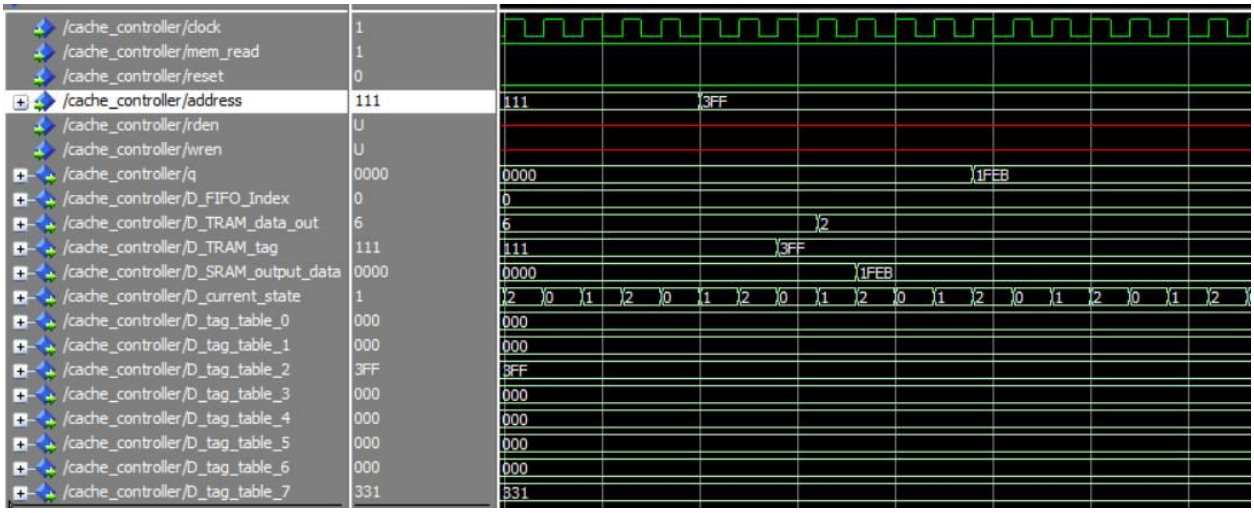


Figure 1. A cache read shown by the address line containing the data. The TRAM tag then gets the data, a cache hit is triggered, and the corresponding cache line is read from SRAM. This is then output back to the CPU.

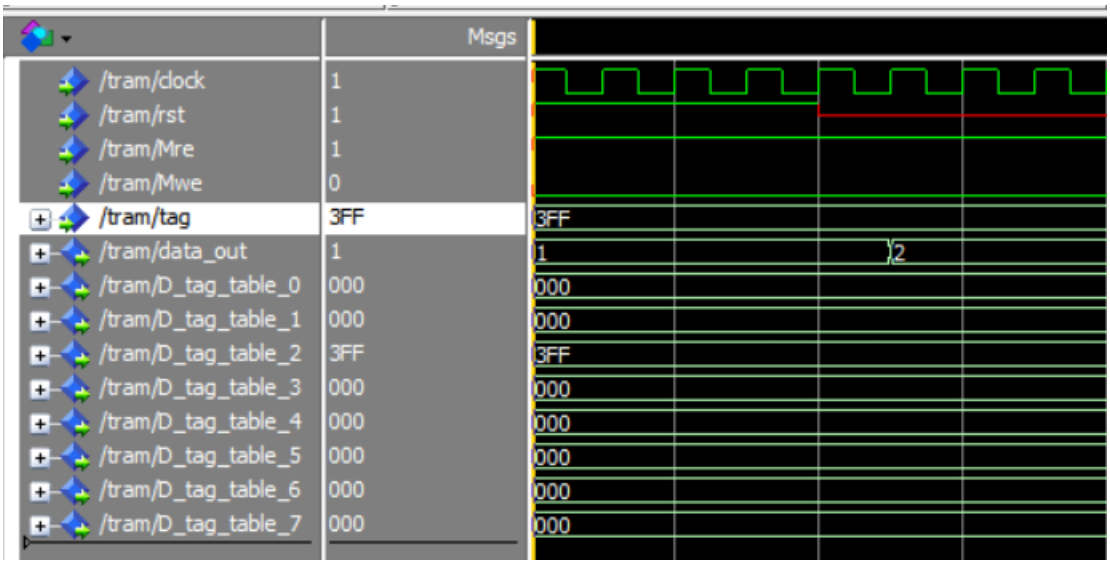


Figure 2. This is showing how TRAM reads the index of the tag to data out. Shown is 3FF being looked for in the tag table. When the reset signal is not high, the table table reads the index in one clock cycle.

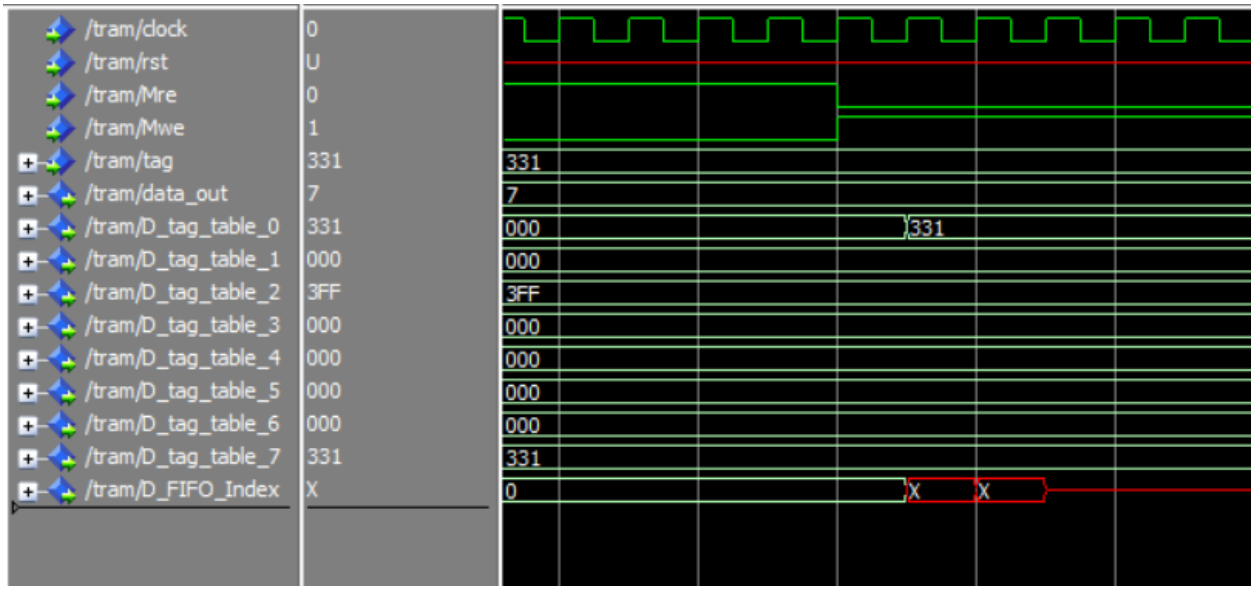


Figure 3. This is showing how TRAM writes the tag into the table. Shown is 321 being written into the tag table at the index of data out. When the write signal is enable, it only takes one clock cycle.

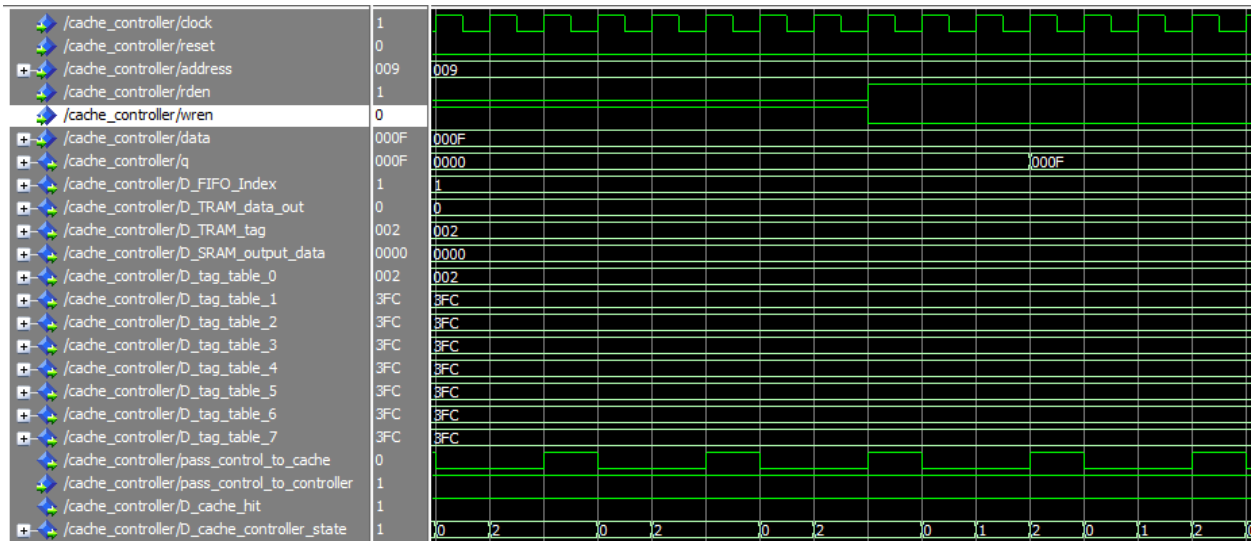


Figure 4. Showing a cache write then read. The value 0x000F was written to tag 0x002. This was then read out on the ‘q’ output line.

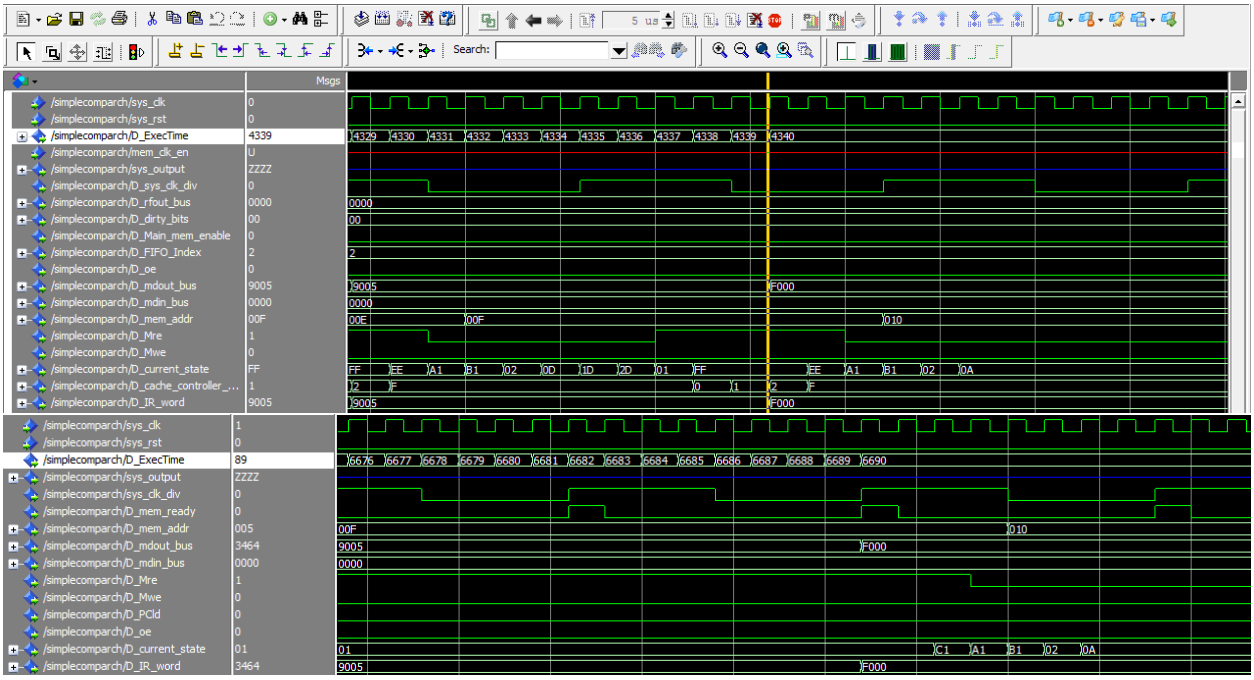
Results Section

After running the benchmark program on both systems it was found that the enhanced system performed 54% better than the reference system. The reference system took 6690 cycles to complete and enhanced system took 4340 cycles.

The enhanced system performed better because the cache access time, 3 cycles, is significantly faster than the m9k access time, 9 to 15 cycles. The total access time of the enhanced system was larger than the cache access time because of the penalties incurred in a cache miss and write back time.

Figure 5. Enhanced system execution time running the 5 x 5 benchmark matrix addition. The total clock cycles for execution was 4340 cycles of the system clock.

Figure 6. Reference system execution time running the 5 x 5 benchmark matrix addition. The total clock cycles for execution was 6690 cycles of the system clock.



Conclusion

The enhanced system outperformed the reference system. It was proven to be faster when executing the benchmark program. There are some optimizations that could be performed on the system though.

The strength is that the cache system is that it loads memory in blocks and stores recently used instructions in the cache, therefore increasing the performance of the system. It also uses a modular structure for the cache.

The weaknesses of the cache system are that we are using a FIFO replacement algorithm which is not the optimal replacement algorithm that could have been implemented. FIFO was chosen for its simplicity. Another weakness is the memory access time not being constant because of the different clock speeds used. It may be rectified but it was not deemed critical as it is implemented the same in both systems.

A different replacement policy could have been used. The FIFO policy was chosen as it was the simplest to implement. Other algorithms that incorporate the ‘dirty’ bit or how often specific tags are reference could be used as well.

The way the write back is implemented could be improved upon as well. If it executed in parallel with other instructions, this would cut down on the time it takes to access main memory within the cache.

A possible enhancement of both systems could be achieved through the use of an enable on the m9k memory. This would allow for a constant instead of the variable access time.

There is also a wasted cycle in the controller.vhd. This is due to the memory ready tag being set and then the next state is set. This could be optimized by using a custom wait cycle for each memory access operation.

References

[1] Intel. (n.d.). Cache Architecture. Retrieved February 26, 2016, from *http://download.intel.com/design/intarch/papers/cache6.pdf*

[2] Stallings, W. (2006). *Computer organization and architecture: Designing for performance* (8th ed.). Upper Saddle River, NJ: Pearson Prentice Hall.

Appendix A

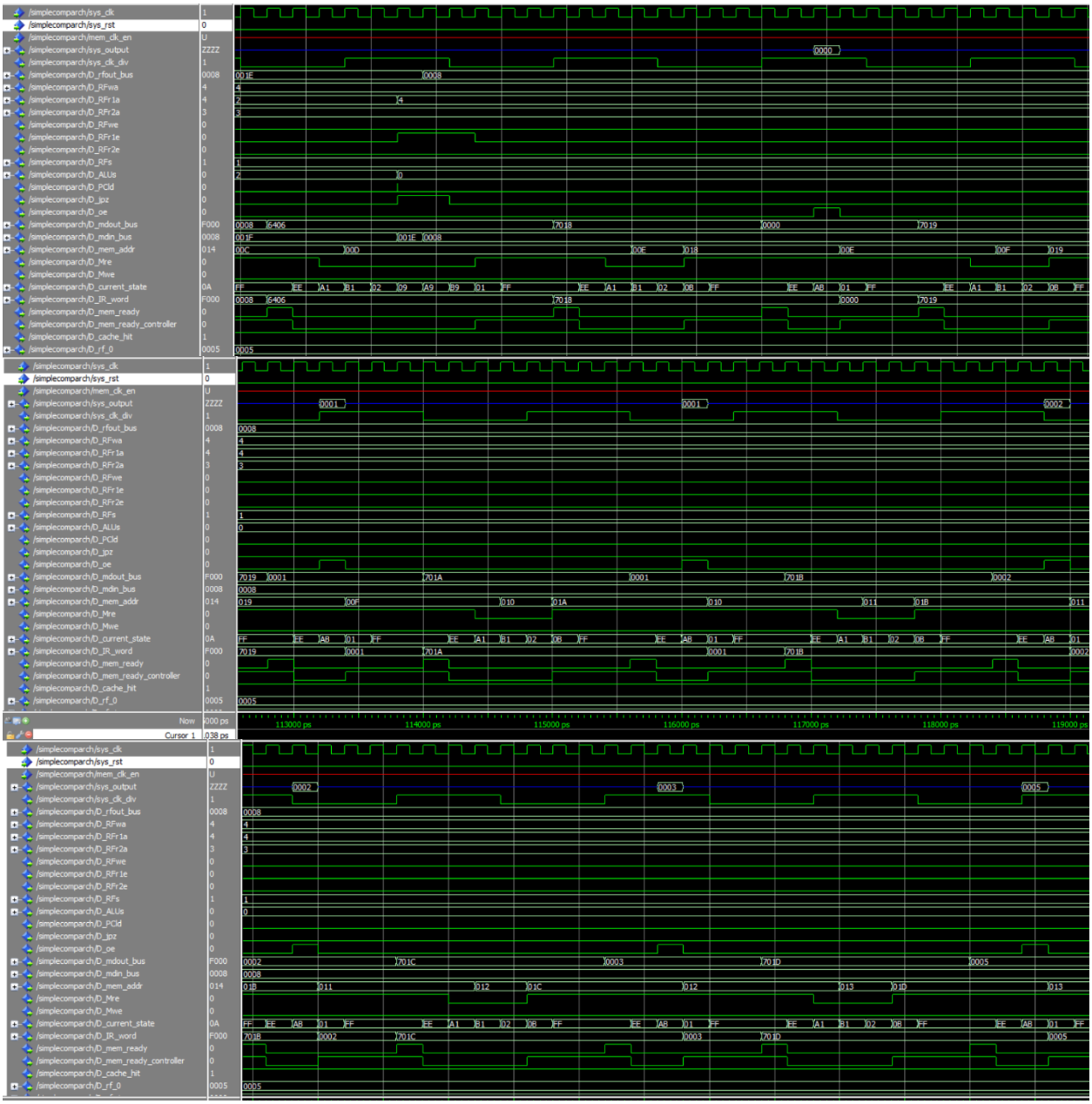


Figure 7. The Fibonacci Series being executed by cache memory. It outputs the first seven results.

Code

```
--------------------------------------------------------------------------------
-- Simple Microprocessor Design (ESD Book Chapter 3)
-- Copyright 2001 Weijun Zhang
--
-- Controller (control logic plus state register)
-- VHDL FSM modeling
-- controller.vhd
--------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.MP_lib.all;

entity controller is
port(   clock:      in std_logic;
    pass_control_to_controller  :in std_logic;
    rst:        in std_logic;
    IR_word:    in std_logic_vector(15 downto 0);
    RFs_ctrl:   out std_logic_vector(1 downto 0);
    RFwa_ctrl:  out std_logic_vector(3 downto 0);
    RFr1a_ctrl: out std_logic_vector(3 downto 0);
    RFr2a_ctrl: out std_logic_vector(3 downto 0);
    RFwe_ctrl:  out std_logic;
    RFr1e_ctrl: out std_logic;
    RFr2e_ctrl: out std_logic;
    ALUs_ctrl:  out std_logic_vector(1 downto 0);
    jmpen_ctrl: out std_logic;
    PCinc_ctrl: out std_logic;
    PCclr_ctrl: out std_logic;
    IRld_ctrl:  out std_logic;
    Ms_ctrl:    out std_logic_vector(1 downto 0);
    Mre_ctrl:   out std_logic;
    Mwe_ctrl:   out std_logic;
    oe_ctrl:    out std_logic;
    current_state : out std_logic_vector(7 downto 0);
    mem_ready_controller:   out std_logic;
    jmpen_ctrl2 : out std_logic
);
end controller;

architecture fsm of controller is

  type state_type is (  S0,S1,S1a,S1b,S2,S3,S3a,S3b,S4,S4a,S4b,S5,S5a,S5b,

S6,S6a,S7,S7a,S7b,S8,S8a,S8b,S9,S9a,S9b,S10,S11,S11a,S12,S12a,S12b,S13,S13a,S
13b,WAIT_STATE);
  signal state: state_type;
    signal next_state: state_type;
    signal count : integer:=0;
begin
  process(clock, rst, IR_word)
    variable OPCODE: std_logic_vector(3 downto 0);
```

```
begin
  if rst='1' then
  Ms_ctrl <= "10";
  PCclr_ctrl <= '1';          -- Reset State
  PCinc_ctrl <= '0';
  IRld_ctrl <= '0';
  RFs_ctrl <= "00";
  Rfwe_ctrl <= '0';
  Mre_ctrl <= '0';
  Mwe_ctrl <= '0';
  jmpen_ctrl <= '0';
  oe_ctrl <= '0';
  state <= S0;

  elsif (clock'event and clock='1') then
  case state is
    when S0 =>    PCclr_ctrl <= '0';  -- Reset State
          current_state <= x"00";
          state <= S1;

    when S1 =>    PCinc_ctrl <= '0';
          current_state <= x"01";
          IRld_ctrl <= '1'; -- Fetch Instruction
          Mre_ctrl <= '1';
          RFwe_ctrl <= '0';
          RFr1e_ctrl <= '0';
          RFr2e_ctrl <= '0';
          Ms_ctrl <= "10";
          Mwe_ctrl <= '0';
          jmpen_ctrl <= '0';
          jmpen_ctrl2 <= '0';
          oe_ctrl <= '0';
          next_state <= S1a;

          pass_control_to_cache <= '1';
          state <= WAIT_STATE;
      when S1a =>
          current_state <= x"A1";
        IRld_ctrl <= '0';
        PCinc_ctrl <= '1';
        Mre_ctrl <= '0';
          state <= S1b;           -- Fetch end ...
    when S1b => PCinc_ctrl <= '0';
          current_state <= x"B1";
         state <= S2;

    when S2 =>
              current_state <= x"02";
              OPCODE := IR_word(15 downto 12);
            case OPCODE is
              when mov1 =>    state <= S3;
              when mov2 =>    state <= S4;
              when mov3 =>    state <= S5;
              when mov4 =>    state <= S6;
              when add =>     state <= S7;
              when subt =>    state <= S8;
              when jz =>      state <= S9;
```

```
            when halt =>    state <= S10;
            when readm =>   state <= S11;
             when mov5 =>   state <= S12;
             when jz2 =>   state <= S13;
            when others =>  state <= S1;
            end case;

when S3 =>
      current_state <= x"03";
      RFwa_ctrl <= IR_word(11 downto 8);
      RFs_ctrl <= "01";  -- RF[rn] <= mem[direct]
      Ms_ctrl <= "01";
      Mre_ctrl <= '1';
      Mwe_ctrl <= '0';
      next_state <= S3a;

      pass_control_to_cache <= '1';
      state <= WAIT_STATE;
when S3a =>
      current_state <= x"A3";
      RFwe_ctrl <= '1';
    Mre_ctrl <= '0';
      state <= S3b;
when S3b =>
      current_state <= x"B3";
      RFwe_ctrl <= '0';
      state <= S1;

when S4 =>
      current_state <= x"04";
      RFr1a_ctrl <= IR_word(11 downto 8);
      RFr1e_ctrl <= '1'; -- mem[direct] <= RF[rn]
      Ms_ctrl <= "01";
      ALUs_ctrl <= "00";
      IRld_ctrl <= '0';
      state <= S4a;           -- read value from RF
when S4a =>
      current_state <= x"A4";
      Mre_ctrl <= '0';
      Mwe_ctrl <= '1';
      next_state <= S4b;

      pass_control_to_cache <= '1';
      state <= WAIT_STATE;
when S4b =>
      current_state <= x"B4";
      Ms_ctrl <= "10";
      Mwe_ctrl <= '0';
      state <= S1;

when S5 =>
      current_state <= x"05";
      RFr1a_ctrl <= IR_word(11 downto 8);
      RFr1e_ctrl <= '1'; -- mem[RF[rn]] <= RF[rm]
      Ms_ctrl <= "00";
      ALUs_ctrl <= "01";
      RFr2a_ctrl <= IR_word(7 downto 4);
```

```
      RFr2e_ctrl <= '1'; -- set addr.& data
      state <= S5a;
when S5a =>
      current_state <= x"A5";
      Mre_ctrl <= '0';
      Mwe_ctrl <= '1'; -- write into memory
      next_state <= S5b;

      pass_control_to_cache <= '1';
      state <= WAIT_STATE;
when S5b =>
      current_state <= x"B5";
      Ms_ctrl <= "10";-- return
      Mwe_ctrl <= '0';
      state <= S1;

when S6 =>
      current_state <= x"06";
      RFwa_ctrl <= IR_word(11 downto 8);
      RFwe_ctrl <= '1'; -- RF[rn] <= imm.
      RFs_ctrl <= "10";
      IRld_ctrl <= '0';
      state <= S6a;
when S6a =>
      current_state <= x"A6";
      state <= S1;

when S7 =>
      current_state <= x"07";
      RFr1a_ctrl <= IR_word(11 downto 8);
      RFr1e_ctrl <= '1'; -- RF[rn] <= RF[rn] + RF[rm]
      RFr2e_ctrl <= '1';
      RFr2a_ctrl <= IR_word(7 downto 4);
      ALUs_ctrl <= "10";
      state <= S7a;
when S7a =>
      current_state <= x"A7";
      RFr1e_ctrl <= '0';
      RFr2e_ctrl <= '0';
      RFs_ctrl <= "00";
      RFwa_ctrl <= IR_word(11 downto 8);
      RFwe_ctrl <= '1';
      state <= S7b;
when S7b =>
      current_state <= x"B7";
      state <= S1;

when S8 =>
      current_state <= x"08";
      RFr1a_ctrl <= IR_word(11 downto 8);
      RFr1e_ctrl <= '1'; -- RF[rn] <= RF[rn] - RF[rm]
      RFr2a_ctrl <= IR_word(7 downto 4);
      RFr2e_ctrl <= '1';
      ALUs_ctrl <= "11";
      state <= S8a;
when S8a =>
      current_state <= x"A8";
```

```
      RFr1e_ctrl <= '0';
      RFr2e_ctrl <= '0';
      RFs_ctrl <= "00";
      RFwa_ctrl <= IR_word(11 downto 8);
      RFwe_ctrl <= '1';
      state <= S8b;
when S8b =>
      current_state <= x"B8";
      state <= S1;
when S9 =>
      current_state <= x"09";
      jmpen_ctrl <= '1';
      RFr1a_ctrl <= IR_word(11 downto 8);
      RFr1e_ctrl <= '1'; -- jz if R[rn] = 0
      ALUs_ctrl <= "00";
      state <= S9a;
when S9a =>
      current_state <= x"A9";
      state <= S9b;
when S9b =>
      current_state <= x"B9";
      jmpen_ctrl <= '0';
    state <= S1;
when S10 =>
      current_state <= x"0A";
      state <= S10; -- halt

when S11 =>
      current_state <= x"0B";
      Ms_ctrl <= "01";
      Mre_ctrl <= '1'; -- read memory
      Mwe_ctrl <= '0';
      next_state <= S11a;

      pass_control_to_cache <= '1';
      state <= WAIT_STATE;
when S11a =>
      current_state <= x"AB";
      oe_ctrl <= '1';
      state <= S1;

  -- this should do : R2 <= mem[RF[r1]] (inverse of MOV3)
  -- copied mov3 code as a starting point
  -- does not work( 10/03/2016 4:45pm )
  -- updated and tested : new works (13/03/2016)

  when S12 =>
      current_state <= x"0C";
      RFr1a_ctrl <= IR_word(11 downto 8);
      Ms_ctrl <= "00";
      Mre_ctrl <= '1';
      RFwe_ctrl <= '0';
      RFr1e_ctrl <= '1';
      RFr2e_ctrl <= '0';
      RFs_ctrl <= "01";
      Mwe_ctrl <= '0';
      next_state <= S12a;
```

```
        pass_control_to_cache <= '1';
        state <= WAIT_STATE;
        --state<=S12a;

  when S12a =>
  current_state <= x"AC";
        Mre_ctrl <= '0';
        RFs_ctrl <= "01";
        RFwa_ctrl <= IR_word(7 downto 4);
        RFwe_ctrl <= '1';
        state<=S12b;


  when S12b =>
        current_state <= x"BC";
        Ms_ctrl <= "10";-- return
        Mwe_ctrl <= '0';
        state <= S1;

when S13 =>
        current_state <= x"0D";
        jmpen_ctrl2 <= '1';
        jmpen_ctrl <= '0';
        RFr1a_ctrl <= IR_word(11 downto 8);
        RFr1e_ctrl <= '1'; -- jz  R[rn]
        ALUs_ctrl <= "00";
        state <= S13a;
  when S13a =>
        current_state <= x"1D";
        state <= S13b;
  when S13b =>
        current_state <= x"2D";
        jmpen_ctrl2 <= '0';
        jmpen_ctrl <= '0';
      state <= S1;


    -- A
    --can SAVE one clock cycle with customized
    -- states for each wait state.
    -- i.e. execute the next_state when count = 1
    --      -> no wasted clock cycles
    when WAIT_STATE =>
        current_state <= x"FF";

        if (pass_control_to_controller = '1') then
            current_state <= x"EE";
            state <= next_state;
            pass_control_to_cache <= '0';
            --A1
            --current_state <= x"A1";
        end if;

  when others =>
end case;
end if;
```

```
  end process;
end fsm;
```

```
---- Cache Controller
--
---- Replacement policy : write back
---- Architecture        : Look through architecture
--
---- Has an input signals from the cpu
---- Has an output signals to the system (where appropriate to communicate to
the main memory)
---- Has two port access to SRAM and TRAM**********
--
---- TRAM: where the tag of the cached lines are found
---- SRAM: cached memory
--
---- needs to do :
----        1. takes in address from cpu and checks if tag is in it.
----        2. HIT  : tag is in TRAM -> respond to cpu request without
starting main memory access.
----            MISS    : Cache passes the bus cycle onto system bus
----                      -Main memory responds to cpu request ( to the cache
controller)
----                      -CC takes info from data line and saves it in SRAM
and TRAM.
--
--
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_unsigned.all;
use work.MP_lib.all;

ENTITY cache_controller IS
    PORT
    (
        pass_control_to_controller : IN STD_LOGIC;
        address : IN STD_LOGIC_VECTOR (11 DOWNTO 0);
        reset       : IN STD_LOGIC;
        clken       : IN STD_LOGIC  := '1';
        clock       : IN STD_LOGIC; --deleted := '1';
        D_sys_clk_div : OUT std_logic;
        D_MAIN_mem_enable : OUT std_logic;
        data        : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
        rden        : IN STD_LOGIC  := '1';
        wren        : IN STD_LOGIC ;
        q           : OUT STD_LOGIC_VECTOR (15 DOWNTO 0);

        D_FIFO_Index : out std_logic_vector(2 downto 0);

        D_TRAM_data_out : out std_logic_vector(2 downto 0);
        D_TRAM_tag : out std_logic_vector(9 downto 0);
        D_SRAM_output_data : out STD_LOGIC_VECTOR (15 DOWNTO 0);
```

```
        D_cache_controller_state : out std_logic_vector(3 downto 0);

        D_tag_table_0 : out std_logic_vector(9 downto 0);
        D_tag_table_1 : out std_logic_vector(9 downto 0);
        D_tag_table_2 : out std_logic_vector(9 downto 0);
        D_tag_table_3 : out std_logic_vector(9 downto 0);
        D_tag_table_4 : out std_logic_vector(9 downto 0);
        D_tag_table_5 : out std_logic_vector(9 downto 0);
        D_tag_table_6 : out std_logic_vector(9 downto 0);
        D_tag_table_7 : out std_logic_vector(9 downto 0);

        D_cache : out cache_type;

        D_mem_data_out : out std_logic_vector(63 downto 0);
        D_mem_read : out std_logic;

        pass_control_to_cache   : out std_logic;

        D_cache_hit : out std_logic;

        D_dirty_bit : out std_logic_vector(7 downto 0);

        D_cache_controller_mem_address : out std_logic_vector(9 downto 0)

    );
END cache_controller;

architecture fsm of cache_controller is

type state_type is ( S0,S1,S2, S_MEM1, S_mem1b, S_mem1c, S_MEM2,
MAIN_WRITE_STATE, main_write_state_b);
  signal state: state_type;
signal TRAM_read : std_logic;
signal TRAM_write : std_logic;
signal TRAM_tag : std_logic_vector(9 downto 0);
signal TRAM_data_out : std_logic_vector(2 downto 0);
signal SRAM_read  : std_logic;
signal SRAM_write  : std_logic;
signal SRAM_word  : std_logic_vector(1 downto 0);
signal SRAM_output_data : STD_LOGIC_VECTOR (15 DOWNTO 0);

signal cache_controller_state : std_logic_vector(3 downto 0);

signal MAIN_read  :  std_logic;
signal MAIN_write : std_logic;

signal MAIN_output_data : STD_LOGIC_VECTOR (63 DOWNTO 0);
signal MAIN_input_data : STD_LOGIC_VECTOR (63 DOWNTO 0);

signal cache_hit  : std_logic;
signal write_to_word : std_logic;
signal write_to_block : std_logic;

signal main_mem_address : std_logic_vector(9 downto 0);

-- The location of the next write to TRAM.
signal FIFO_Index : integer := 0;
```

```
-- Dirty bits
signal dirty_bits : std_logic_vector(7 downto 0);

-- The cache that is managed in SRAM
signal cache : cache_type;

-- The TRAM tag table
signal tag_table : tag_type;

signal sys_clk_div      : std_logic;
signal MAIN_mem_enable : std_logic;

signal count : integer := 0;

signal main_mem_ready : std_logic;

begin

process (clock, reset, address)
begin
    SRAM_word <= address(1 downto 0);
    TRAM_tag <= address(11 downto 2);
    D_cache_controller_mem_address <= main_mem_address;
    if reset='1' then
        TRAM_read  <= '0';
        TRAM_write <= '0';
        TRAM_tag <= address(11 downto 2);
        state <= S0;
        write_to_word <= '0';
        write_to_block <= '0';
        FIFO_Index <= 0;
        MAIN_mem_enable <= '0';

    elsif pass_control_to_controller = '0' then
        cache_controller_state <= x"F";
        state <= S0;

   elsif (clock'event and clock='1' and pass_control_to_controller = '1')
then
        case state is
            when S0 =>
                cache_controller_state <= x"0";
                pass_control_to_cache <= '0';

                main_mem_address <= address(11 downto 2);
                MAIN_read <= '0';

                -- Clear SRAM write;
                SRAM_write <= '0';
                write_to_word <= '0';
                write_to_block <= '0';

                --read from tag_table in TRAM
                TRAM_read  <= '1';
                TRAM_write <= '0';
                state <= S1;
```

```
                MAIN_mem_enable <= '0';

            --delay to account for writing to memory
            -- with instruction mov2

            when S1 =>

                --CHECK cache miss or hit
                if (cache_hit = '1') then
                --on cache HIT

                    TRAM_read  <= '0';
                    TRAM_write <= '0';
                    --read
                    if(rden = '1' and wren = '0') then
                        cache_controller_state <= x"1";
                        SRAM_read  <= '1';
                        SRAM_write <= '0';
                        SRAM_word  <= address(1 downto 0);

                    elsif(rden = '0' and wren = '1') then
                        cache_controller_state <= x"2";
                        SRAM_read  <= '0';
                        SRAM_write <= '1';
                        write_to_word <= '1';
                        write_to_block <= '0';
                        SRAM_word  <= address(1 downto 0);
                        dirty_bits(conv_integer(TRAM_data_out)) <= '1';
                    end if;

                    state <= S2;
                --end HIT
                else
                    --cache MISS

                    MAIN_mem_enable <= '1';

                    -- To write back
                    -- (We need to add a 'dirty' bit to the indexes of the
tag)
                    -- if (dirty = '1')
                    --  Get old tag from FIFO_Index
                    --  Get old data from SRAM
                    --  Write SRAM data to TRAM's tag address in Main memory
                    --  Read new tag (address) from memory
                    -- else
                    --      Read new tag (address) from memory

                    -- To optimize:
                    -- (create a new process that operates on the
'write_back_flag')
                    -- Read new tag (address) from memory
                    -- pass back control to 'controller'
                    -- while this is happening,
                    --  'cache-controller': write back to memory.
```

```
                    --  have a 'write_back_flag' in S0 that says when
'memory' is not writing


                    --WRITE to MAIN memory on cache miss and dirty bit set.
                    if(dirty_bits(FIFO_Index) = '1') then
                        -- This is the memory address of the data being
written back from
                        -- the cache.
                        cache_controller_state <= x"4";
                        main_mem_address <= tag_table(FIFO_Index);
                        MAIN_input_data <= cache(FIFO_Index)(0) &
cache(FIFO_Index)(1) & cache(FIFO_Index)(2) & cache(FIFO_Index)(3);
                        MAIN_write <= '1';
                        MAIN_read <= '0';
                        dirty_bits(FIFO_Index) <= '0';

                        state <= MAIN_WRITE_STATE;

                    else
                        cache_controller_state <= x"5";
                        --READ from MAIN memory on cache miss
                        MAIN_read <= '1'; -- read memory
                        MAIN_write <= '0';
                        -- Write to TRAM;
                        TRAM_write <= '1';
                        TRAM_read <= '0';

                        dirty_bits(FIFO_Index) <= '0';

                        state <= S_MEM1;
                    end if;

                --end MISS
                end if;

            when S2 =>
                cache_controller_state <= x"2";

                --shut off read
                SRAM_read  <= '0';
                SRAM_write <= '0';

                pass_control_to_cache <= '1';
                state <= S0;

            when S_MEM1 =>
                cache_controller_state <= x"6";
                -- Clear TRAM controls;
                TRAM_write <= '0';
                TRAM_read <= '0';

                if(main_mem_ready = '0') then
                    state <= S_MEM1;
                else
                    cache_controller_state <= x"7";
                    state <= S_mem1c;
```

```
        end if;

    when S_mem1c =>
        if(main_mem_ready = '0') then
            state <= S_mem1c;
        else
            cache_controller_state <= x"3";
            state <= S_mem1b;
        end if;

    when S_mem1b =>
        cache_controller_state <= x"8";
        if(main_mem_ready = '0') then
            state <= S_mem1b;
        else
            cache_controller_state <= x"C";
            -- Increment the FIFO Index after a write
            if (FIFO_Index = 7) then
                FIFO_Index <= 0;
            else
                FIFO_Index <= FIFO_Index + 1;
            end if;
            state <= S_MEM2;
        end if;

    when S_MEM2 =>
        cache_controller_state <= x"D";

        --Write to SRAM;
        SRAM_write <= '1';
        SRAM_read <= '0';
        write_to_word <= '0';
        write_to_block <= '1';

        state <= S0;

    when MAIN_WRITE_STATE =>
        cache_controller_state <= x"A";
        if(main_mem_ready = '0') then
            state <= MAIN_WRITE_STATE;
        else
            cache_controller_state <= x"B";
            state <= main_write_state_b;
        end if;

    when main_write_state_b =>
        cache_controller_state <= x"C";
        if(main_mem_ready = '0') then
            state <= main_write_state_b;
        else
            cache_controller_state <= x"D";
            MAIN_write <= '0';
            MAIN_read <= '0';
            state <= S0;
        end if;

    when others =>
```

```
        end case;
    end if;

end process;

--process (clock, MAIN_mem_enable) begin
--  if (MAIN_mem_enable = '0') then
--      count <= 0;
--      sys_clk_div <= '0';
--  elsif (rising_edge(clock) and MAIN_mem_enable = '1') then
--          count <= count + 1;
--          if (count = 3) then
--              sys_clk_div <= NOT sys_clk_div;
--              count <= 0;
--              if (sys_clk_div = '0') then
--                  main_mem_ready <= '1';
--              end if;
--          end if;
--  end if;
--end process;
process (clock, reset) begin
    if (reset = '1') then
        count <= 0;
        sys_clk_div <= '0';
    elsif (rising_edge(clock)) then
        main_mem_ready <= '0';
        count <= count + 1;
        if (count = 3) then
            sys_clk_div <= NOT sys_clk_div;
            count <= 0;
            if (sys_clk_div = '0') then
                main_mem_ready <= '1';
            end if;
        end if;
    end if;
end process;

Unit1: memory_4KB port map(
    main_mem_address,
    '1',
    sys_clk_div,
    MAIN_input_data,
    MAIN_read,
    MAIN_write,
    MAIN_output_data);

Unit2: TRAM port map(
        clock,
        reset,
        --TRAM_read,
        '1',    -- forcing to 1 to always read TRAM tag from address line
        TRAM_write,
        TRAM_tag,
        TRAM_data_out,
        cache_hit,
        FIFO_Index,
        D_FIFO_Index,
```

```
        tag_table
        );

Unit3: SRAM port map(
        clock,
        reset,
        SRAM_read,
        SRAM_write,
        SRAM_word,
        TRAM_data_out,
        data,
        q,
        MAIN_output_data,
        write_to_word,
        write_to_block,
        cache
        );

        D_TRAM_data_out <= TRAM_data_out;
        D_SRAM_output_data <= SRAM_output_data;
        D_TRAM_tag <= TRAM_tag;
        D_cache_controller_state <= cache_controller_state;
        D_cache_hit <= cache_hit;
        D_mem_data_out <= MAIN_output_data;
        D_mem_read <= MAIN_read;


        D_cache <= cache;
            D_tag_table_0 <= tag_table(0);
    D_tag_table_1 <= tag_table(1);
    D_tag_table_2 <= tag_table(2);
    D_tag_table_3 <= tag_table(3);
    D_tag_table_4 <= tag_table(4);
    D_tag_table_5 <= tag_table(5);
    D_tag_table_6 <= tag_table(6);
    D_tag_table_7 <= tag_table(7);

    D_MAIN_mem_enable <= MAIN_MEm_enable;
    D_sys_clk_div <= sys_clk_div;
end fsm;




-- TRAM

-- Memory where the tag of each line is stored

-- Has a two way port to the Cache Controller (read and write) **********

-- 8 tag length (to match number of lines in cache)

-------------------------------------------------------------
-- Simple Computer Architecture
--
-- sram 256*16
```

```
-- 8 bit address; 16 bit data
-- sram.vhd
----------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
use work.MP_lib.all;

entity tram is
port (
        clock       :   in std_logic;
        rst     :   in std_logic;
        TRAM_read       :   in std_logic;
        TRAM_write      :   in std_logic;
        tag         :   in std_logic_vector(9 downto 0);
        data_out :  out std_logic_vector(2 downto 0);

        cache_hit : buffer std_logic;

        FIFO_Index : in integer;

        D_FIFO_Index : out std_logic_vector(2 downto 0);

        tag_table : buffer tag_type
);
end tram;

architecture behv of tram    is

begin
    write: process(clock, rst, TRAM_read, tag)
    begin
        if rst='1' then
            tag_table <= (

--              0 => "0000000000",
--              1 => "0000000001",
--              2 => "0000000010",
--              3 => "0000000011",
--              4 => "0000000100",
--              5 => "0000000101",
--              6 => "0000000110",
--              7 => "0000000111",
                others => "1111111100"
            );
        elsif (clock'event and clock = '1') then
                --if (TRAM_write ='1' and TRAM_read = '0') then
                if (TRAM_write ='1') then
                    tag_table(FIFO_Index) <= tag;
                end if;
        end if;
    end process;

   read: process(clock, rst, TRAM_write, tag)
```

```
    begin
        if rst='1' then
            data_out <= "001";
        else
            if (clock'event and clock = '1') then
                cache_hit <= '0';
                for index in 0 to 7 loop
                    if tag_table(index) = tag then
                        data_out <= std_logic_vector(to_unsigned(index,
data_out'length));
                        cache_hit <= '1';
                    end if;
                end loop;
            end if;
        end if;
    end process;

    D_FIFO_Index <= std_logic_vector(to_unsigned(FIFO_Index,
D_FIFO_Index'length));


end behv;
```

```
----------------------------------------------------------
-- Simple Computer Architecture
--
-- sram 256*16
-- 8 bit address; 16 bit data
-- sram.vhd
----------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.MP_lib.all;

entity sram is
port (
        clock    :   in std_logic;
        rst      :   in std_logic;
        Mre      :   in std_logic;
        Mwe      :   in std_logic;
        word     :   in std_logic_vector(1 downto 0);
        tag      :   in std_logic_vector(2 downto 0);
        data_in  :   in std_logic_vector(15 downto 0);
        data_out:    out std_logic_vector(15 downto 0);
        mem_data_in : in std_logic_vector(63 downto 0);
        write_to_word : in std_logic;
        write_to_block : in std_logic;
        cache       : buffer cache_type
);
end sram;
```

```
architecture behv of sram    is

begin
    write: process(clock, rst, Mre, tag, word, data_in)
    begin
        if rst='1' then
            cache(0)<= (
                0 => x"1010",
                others => x"0001");
            cache(1)<= (
                others => x"0001");
            cache(2)<= (
                others => x"0001");
            cache(3)<= (
                others => x"0001");
            cache(4)<= (
                others => x"0001");
            cache(5)<= (
                others => x"0001");
            cache(6)<= (
                others => x"0001");
            cache(7)<= (
                others => x"0001");


--          cache(0) <= ( -- 0d
--              0 => x"3000",
--              1 => x"3101",
--              2 => x"321A",
--              3 => x"3301",others => x"0000");
--          cache(1) <= ( -- 4d
--              0 => x"1018",
--              1 => x"1119",
--              2 => x"111F",
--              3 => x"4100",others => x"0000");
--          cache(2) <= ( -- 8d
--              0 => x"001F",
--              1 => x"2210",
--              2 => x"4230",
--              3 => x"041E",others => x"0000");
--          cache(3) <= ( -- 12d
--              0 => x"6406",
--              1 => x"7018",
--              2 => x"7019",
--              3 => x"701A",others => x"0000");
--          cache(4) <= ( -- 16d
--              0 => x"701B",
--              1 => x"701C",
--              2 => x"701D",
--              3 => x"F000",others => x"0000");
--          cache(5) <= ( -- 20d
--              0 => x"0000",
--              1 => x"0000",
--              2 => x"0000",
--              3 => x"0000",others => x"0000");
--          cache(6) <= ( -- 24d
```

```
--               0 => x"0000",
--               1 => x"0000",
--               2 => x"0000",
--               3 => x"0000",others => x"0000");
--           cache(7) <= ( -- 28d
--               0 => x"0000",
--               1 => x"0000",
--               2 => x"0000",
--               3 => x"0000",others => x"0000");
        else
            if (clock'event and clock = '1') then
                if (Mwe ='1' and Mre = '0' and write_to_word = '1' and
write_to_block = '0') then
                    cache(conv_integer(tag))(conv_integer(word)) <= data_in;
                elsif (Mwe ='1' and Mre = '0' and write_to_word = '0' and
write_to_block = '1') then
                    --
                    cache(conv_integer(tag))(0) <= mem_data_in(63 downto 48);
                    cache(conv_integer(tag))(1) <= mem_data_in(47 downto 32);
                    cache(conv_integer(tag))(2) <= mem_data_in(31 downto 16);
                    cache(conv_integer(tag))(3) <= mem_data_in(15 downto 0);
                end if;
            end if;
        end if;
    end process;

    read: process(clock, rst, Mwe, tag, word)
    begin
        if rst='1' then
            data_out <= ZERO;
        else
            if (clock'event and clock = '1') then
                if (Mre ='1' and Mwe ='0') then
                    data_out <= cache(conv_integer(tag))(conv_integer(word));
                end if;
            end if;
        end if;
    end process;

end behv;
```

```
-- Library for Microprocessor example
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;

package MP_lib is

type ram_type is array (0 to 255) of
                std_logic_vector(15 downto 0);

type rf_type is array (0 to 15) of
        std_logic_vector(15 downto 0);
```

```
type tag_type is array (7 downto 0) of std_logic_vector(9 downto 0);

type cache_line is array (3 downto 0) of std_logic_vector(15 downto 0);
type cache_type is array (7 downto 0) of cache_line;

constant ZERO : std_logic_vector(15 downto 0) := "0000000000000000";
constant HIRES : std_logic_vector(15 downto 0) := "ZZZZZZZZZZZZZZZZ";
constant mov1 : std_logic_vector(3 downto 0) := "0000";
constant mov2 : std_logic_vector(3 downto 0) := "0001";
constant mov3 : std_logic_vector(3 downto 0) := "0010";
constant mov4 : std_logic_vector(3 downto 0) := "0011";
constant add  : std_logic_vector(3 downto 0) := "0100";
constant subt : std_logic_vector(3 downto 0) := "0101";
constant jz   : std_logic_vector(3 downto 0) := "0110";
constant halt  : std_logic_vector(3 downto 0) := "1111";
constant readm  : std_logic_vector(3 downto 0) := "0111";
constant jz2    :  std_logic_vector(3 downto 0) := "1001";
constant mov5 : std_logic_vector(3 downto 0) := "1000"; --new op code to move
mem[Reg1]-> Reg2


component CPU is
port (
        cpu_clk                 : in std_logic;
        mem_ready :  in std_logic;
        cpu_rst                 : in std_logic;
        mdout_bus               : in std_logic_vector(15 downto 0);
        mdin_bus                    : out std_logic_vector(15 downto 0);
        mem_addr                    : out std_logic_vector(7 downto 0);
        Mre_s                       : out std_logic;
        Mwe_s                       : out std_logic;
        oe_s                        : out std_logic;
        current_state: out std_logic_vector(7 downto 0);
        IR_word                 : out std_logic_vector(15 downto 0);
        tmp_rf                  : out rf_type;
        mem_ready_controller :  out std_logic;
        -- Debug variables: output to upper level for simulation purpose only
        D_rfout_bus: out std_logic_vector(15 downto 0);
        D_RFwa_s, D_RFr1a_s, D_RFr2a_s: out std_logic_vector(3 downto 0);
        D_RFwe_s, D_RFr1e_s, D_RFr2e_s: out std_logic;
        D_RFs_s, D_ALUs_s: out std_logic_vector(1 downto 0);
        D_PCld_s, D_jpz_s: out std_logic
        -- end debug variables
);
end component;

component alu is
port (
        num_A:  in std_logic_vector(15 downto 0);
        num_B:  in std_logic_vector(15 downto 0);
        jpsign: in std_logic;
        ALUs:   in std_logic_vector(1 downto 0);
        ALUz:   out std_logic;
        ALUout: out std_logic_vector(15 downto 0);
        jpsign2: in std_logic
);
```

```
end component;

component bigmux is
port(
    Ia:     in std_logic_vector(15 downto 0);
    Ib:     in std_logic_vector(15 downto 0);
    Ic: in std_logic_vector(15 downto 0);
    Id: in std_logic_vector(15 downto 0);
    Option: in std_logic_vector(1 downto 0);
    Muxout: out std_logic_vector(15 downto 0)
);
end component;

component controller is
port(
    clock:      in std_logic;
    pass_control_to_controller:  in std_logic;
    rst:        in std_logic;
    IR_word:    in std_logic_vector(15 downto 0);
    RFs_ctrl:   out std_logic_vector(1 downto 0);
    RFwa_ctrl:  out std_logic_vector(3 downto 0);
    RFr1a_ctrl: out std_logic_vector(3 downto 0);
    RFr2a_ctrl: out std_logic_vector(3 downto 0);
    RFwe_ctrl:  out std_logic;
    RFr1e_ctrl: out std_logic;
    RFr2e_ctrl: out std_logic;
    ALUs_ctrl:  out std_logic_vector(1 downto 0);
    jmpen_ctrl: out std_logic;
    PCinc_ctrl: out std_logic;
    PCclr_ctrl: out std_logic;
    IRld_ctrl:  out std_logic;
    Ms_ctrl:    out std_logic_vector(1 downto 0);
    Mre_ctrl:   out std_logic;
    Mwe_ctrl:   out std_logic;
    oe_ctrl:    out std_logic;
    current_state: out std_logic_vector(7 downto 0);
    mem_ready_controller:   out std_logic;
    jmpen_ctrl2:    out std_logic
);
end component;

component IR is
port(
    IRin:     in std_logic_vector(15 downto 0);
    IRld:     in std_logic;
    dir_addr: out std_logic_vector(15 downto 0);
    IRout:    out std_logic_vector(15 downto 0)
);
end component;

component memory_4KB is
    PORT
    (
        address : IN STD_LOGIC_VECTOR (9 DOWNTO 0);
        clken       : IN STD_LOGIC  := '1';
        clock       : IN STD_LOGIC  := '1';
        data        : IN STD_LOGIC_VECTOR (63 DOWNTO 0);
```

```
        rden        : IN STD_LOGIC;
        wren        : IN STD_LOGIC ;
        q       : OUT STD_LOGIC_VECTOR (63 DOWNTO 0)
    );
end component;
component cache_controller is
    PORT
    (
        pass_control_to_cache : IN STD_LOGIC;
        address : IN STD_LOGIC_VECTOR (11 DOWNTO 0);
        reset       : IN STD_LOGIC;
        clken       : IN STD_LOGIC  := '1';
        clock       : IN STD_LOGIC;
        D_sys_clk_div : OUT std_logic;
        D_main_mem_enable : out std_LOGIC;
        data        : IN STD_LOGIC_VECTOR (15 DOWNTO 0);
        rden        : IN STD_LOGIC  := '1';
        wren        : IN STD_LOGIC ;
        q           : OUT STD_LOGIC_VECTOR (15 DOWNTO 0);
        D_FIFO_Index : out std_logic_vector(2 downto 0);
        mem_ready    : OUT std_logic;
        D_cache_hit : OUT std_logic;
        D_TRAM_tag : out std_logic_vector(9 downto 0);
        D_tag_table_0 : out std_logic_vector(9 downto 0);
        D_tag_table_1 : out std_logic_vector(9 downto 0);
        D_tag_table_2 : out std_logic_vector(9 downto 0);
        D_tag_table_3 : out std_logic_vector(9 downto 0);
        D_tag_table_4 : out std_logic_vector(9 downto 0);
        D_tag_table_5 : out std_logic_vector(9 downto 0);
        D_tag_table_6 : out std_logic_vector(9 downto 0);
        D_tag_table_7 : out std_logic_vector(9 downto 0);

        D_cache : out cache_type;

        D_cache_controller_state : out std_logic_vector(3 downto 0);
        D_dirty_bit : out std_logic_vector(7 downto 0);
        D_cache_controller_mem_address : out std_logic_vector(9 downto 0)
    );
end component;
component TRAM is
port (
        clock       :   in std_logic;
        rst     :   in std_logic;
        TRAM_read       :   in std_logic;
        TRAM_write      :   in std_logic;
        tag         :   in std_logic_vector(9 downto 0);
        data_out :  out std_logic_vector(2 downto 0);

        cache_hit : out std_logic;
        FIFO_Index: in integer;

        D_FIFO_Index : out std_logic_vector(2 downto 0);

        tag_table : buffer tag_type
);
end component;
```

```
component SRAM is
port (
        clock   :   in std_logic;
        rst     :   in std_logic;
        Mre     :   in std_logic;
        Mwe     :   in std_logic;
        word    :   in std_logic_vector(1 downto 0);
        tag     :   in std_logic_vector(2 downto 0);
        data_in :   in std_logic_vector(15 downto 0);
        data_out:   out std_logic_vector(15 downto 0);
        mem_data_in : in std_logic_vector(63 downto 0);
        write_to_word : in std_logic;
        write_to_block : in std_logic;
        cache       : buffer cache_type
);
end component;

component obuf is
port(
    O_en:       in std_logic;
    obuf_in:    in std_logic_vector(15 downto 0);
    obuf_out:   out std_logic_vector(15 downto 0)
);
end component;

component PC is
port(
    clock:  in std_logic;
    PCld:   in std_logic;
    PCinc:  in std_logic;
    PCclr:  in std_logic;
    PCin:   in std_logic_vector(15 downto 0);
    PCout:  out std_logic_vector(15 downto 0)
);
end component;

component reg_file is
port (
    clock   :   in std_logic;
    rst :   in std_logic;
    RFwe    :   in std_logic;
    RFr1e   :   in std_logic;
    RFr2e   :   in std_logic;
    RFwa    :   in std_logic_vector(3 downto 0);
    RFr1a   :   in std_logic_vector(3 downto 0);
    RFr2a   :   in std_logic_vector(3 downto 0);
    RFw :   in std_logic_vector(15 downto 0);
    RFr1    :   out std_logic_vector(15 downto 0);
    RFr2    :   out std_logic_vector(15 downto 0);
    debug_tmp_rf : out rf_type
);
end component;

component smallmux is
port(
    I0:     in std_logic_vector(15 downto 0);
    I1:     in std_logic_vector(15 downto 0);
```

```
    I2:     in std_logic_vector(15 downto 0);
    Sel:    in std_logic_vector(1 downto 0);
    O:      out std_logic_vector(15 downto 0)
    );
end component;

component ctrl_unit is
port(
    clock_cu:   in  std_logic;
    mem_ready:   in std_logic;
    rst_cu:     in  std_logic;
    PCld_cu:    in  std_logic;
    mdata_out:  in  std_logic_vector(15 downto 0);
    dpdata_out: in  std_logic_vector(15 downto 0);
    maddr_in:   out     std_logic_vector(15 downto 0);
    immdata:    out     std_logic_vector(15 downto 0);
    RFs_cu:     out std_logic_vector(1 downto 0);
    RFwa_cu:    out std_logic_vector(3 downto 0);
    RFr1a_cu:   out std_logic_vector(3 downto 0);
    RFr2a_cu:   out std_logic_vector(3 downto 0);
    RFwe_cu:    out std_logic;
    RFr1e_cu:   out std_logic;
    RFr2e_cu:   out std_logic;
    jpen_cu:    out     std_logic;
    ALUs_cu:    out std_logic_vector(1 downto 0);
    Mre_cu:     out     std_logic;
    Mwe_cu:     out     std_logic;
    oe_cu:      out     std_logic;
    current_state: out std_logic_vector(7 downto 0);
    IR_word                 : out std_logic_vector(15 downto 0);
    mem_ready_controller:   out std_logic;
    jpen_cu2:   out     std_logic
);
end component;

component datapath is
port(
    clock_dp:   in  std_logic;
    rst_dp:     in  std_logic;
    imm_data:   in  std_logic_vector(15 downto 0);
    mem_data:   in  std_logic_vector(15 downto 0);
    RFs_dp:     in  std_logic_vector(1 downto 0);
    RFwa_dp:    in  std_logic_vector(3 downto 0);
    RFr1a_dp:   in  std_logic_vector(3 downto 0);
    RFr2a_dp:   in  std_logic_vector(3 downto 0);
    RFwe_dp:    in  std_logic;
    RFr1e_dp:   in  std_logic;
    RFr2e_dp:   in  std_logic;
    jp_en:      in  std_logic;
    ALUs_dp:    in  std_logic_vector(1 downto 0);
    ALUz_dp:    out     std_logic;
    RF1out_dp:  out     std_logic_vector(15 downto 0);
    ALUout_dp:  out     std_logic_vector(15 downto 0);
    tmp_rf : out rf_type;
    jp_en2:     in  std_logic
);
end component;
```

```
end MP_lib;



package body MP_lib is

    -- Procedure Body (optional)

end MP_lib;




-- Addition of two 5x5 matrices


-- matrix addition. result found in memory location mem[70]..mem[94]
--      (starts adding from top left corner then goes top-down following the
columns)


WIDTH=16;
DEPTH=4096;

ADDRESS_RADIX=UNS;
DATA_RADIX=HEX;

CONTENT
BEGIN
        -- space for program mem[0..49]

0 :   3019;     -- R0 = 25   (Start-up value of program counter)
1 :   3132;     -- R1 = 50   (position of first element of matrix a)
2 :   324B;     -- R2 = 75   (position of first element of matrix b)
3 :   3301;     -- R3 = 1    (constant 1)
4 :   3464;     -- R4 = 100 (position of first element of matrix c (result
matrix))

--5:   112D;        -- M[45] <- R0 (M[45] is the position of matrix 0
pointer)             M[45]=50
--6:   1242E;       -- M[46] <- R1 (M[46] is the position of matrix 1
pointer)             M[46]=75
--7:   102F;        -- M[45] <- R0 (M[47] is the program counter, loop exits
at 25 runs)    M[47]=0

--loop beginning
    5:  8160;       -- R6    <- M[R1] (matrix 0 current element copied in R6)
    6:  8270;       -- R7    <- M[R2] (matrix 0 current element copied in R7)
    7:  4678;       -- R6    <- R6 + R7
    8:  2460;       -- M[R4] <- R6 (R4 is matrix c pointer location)


    --increment pointers
    9:   5030;      -- decrement program counter
```

```
    10:  4131;      -- increment matrix a pointer
    11:  4232;      -- increment matrix b pointer
    12:  4434;      -- increment matrix c pointer
    13:  600E;      -- jump to mem[14]]if register 0 = 0
    14:  9005;      -- jump back to instruction 5
--loop end


15:  F000;      -- HALT

[16..49]  :   0;


--matrix a (columns created from left to right)(items in columnss are created
top to bottom)

--column 0

50 :  0001;         -- Mem[50] = 0001
51 :  0001;         -- Mem[51] = 0001
52 :  0001;         -- Mem[52] = 0001
53 :  0001;         -- Mem[53] = 0001
54 :  0001;         -- Mem[54] = 0001

--column 1

55 :  0001;         -- Mem[55] = 0001
56 :  0001;         -- Mem[56] = 0001
57 :  0001;         -- Mem[57] = 0001
58 :  0001;         -- Mem[58] = 0001
59 :  0001;         -- Mem[59] = 0001

--column 2

60 :  0001;         -- Mem[60] = 0001
61 :  0001;         -- Mem[61] = 0001
62 :  0001;         -- Mem[62] = 0001
63 :  0001;         -- Mem[63] = 0001
64 :  0001;         -- Mem[64] = 0001
--column 3

65 :  0001;         -- Mem[65] = 0001
66 :  0001;         -- Mem[66] = 0001
67 :  0001;         -- Mem[67] = 0001
68 :  0001;         -- Mem[68] = 0001
69 :  0001;         -- Mem[69] = 0001

--column 4

70 :  0001;         -- Mem[70] = 0001
71 :  0001;         -- Mem[71] = 0001
72 :  0001;         -- Mem[72] = 0001
73 :  0001;         -- Mem[73] = 0001
74 :  0001;         -- Mem[74] = 0001
```

```
--matrix b (columns created from left to right)(items in colums are created
top to bottom)

--column 0

75:   0001;          -- Mem[75] = 0001
76 :  0001;          -- Mem[76] = 0001
77 :  0001;          -- Mem[77] = 0001
78 :  0001;          -- Mem[78] = 0001
79 :  0001;          -- Mem[79] = 0001

--column 1

80 :  0001;          -- Mem[80] = 0001
81 :  0001;          -- Mem[81] = 0001
82 :  0001;          -- Mem[82] = 0001
83 :  0001;          -- Mem[83] = 0001
84 :  0001;          -- Mem[84] = 0001

--column 2

85 :  0001;          -- Mem[85] = 0001
86 :  0001;          -- Mem[86] = 0001
87 :  0001;          -- Mem[87] = 0001
88 :  0001;          -- Mem[88] = 0001
89 :  0001;          -- Mem[89] = 0001

--column 3

90 :  0001;          -- Mem[90] = 0001
91 :  0001;          -- Mem[91] = 0001
92 :  0001;          -- Mem[92] = 0001
93 :  0001;          -- Mem[93] = 0001
94 :  0001;          -- Mem[94] = 0001

--column 4

95 :  0001;          -- Mem[95] = 0001
96 :  0001;          -- Mem[96] = 0001
97 :  0001;          -- Mem[97] = 0001
98 :  0001;          -- Mem[98] = 0001
99 :  0001;          -- Mem[99] = 0001

    [100..4095]  :   0;
END;
```

Figures

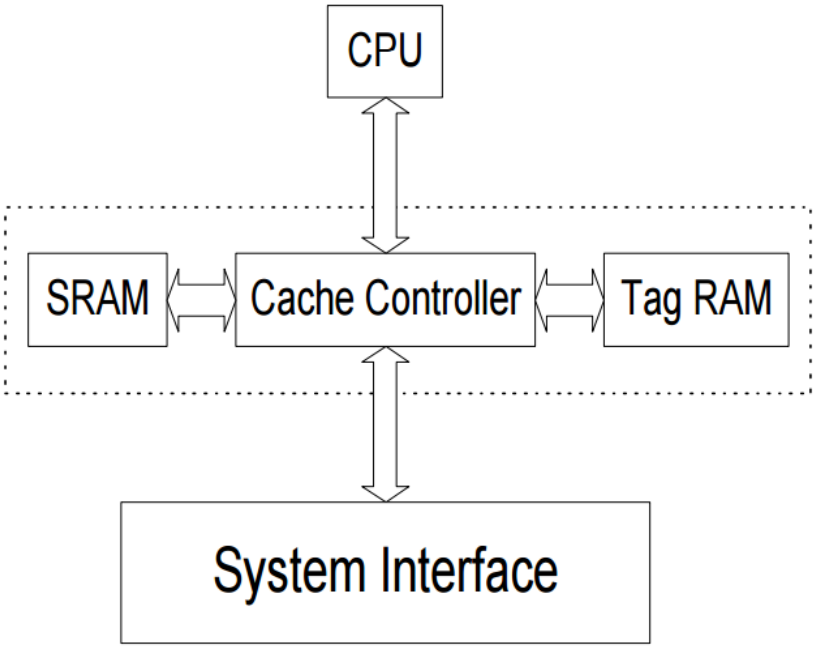


Figure 8. Look through cache structure [1]

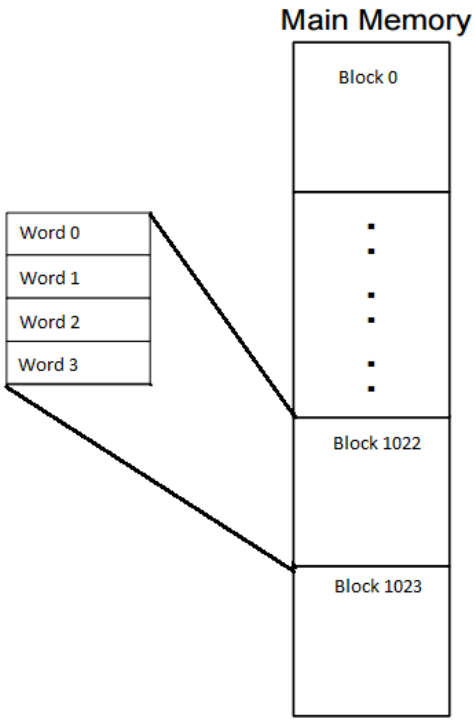


Figure 9. Block structure of main memory and cache memory [1]

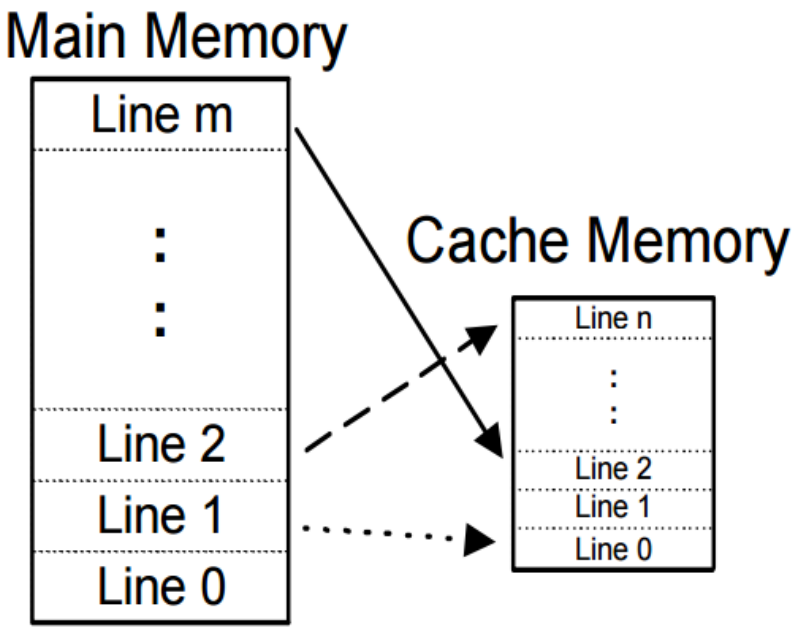


Figure 10. Full associative cache line mapping [1]