Department of Electrical, Computer
& Biomedical Engineering
Faculty of Engineering & Architectural Science

COE758 – Digital Systems Engineering

LAB REPORT

Semester/Year:	Fall 2024
Lab Number:	Project 1

Instructor:	Dr. Lev Kirischian
Section No:	042
Submission Date:	10/20/2024
Due Date:	10/17/2024

Student Name	Student ID	Signature
Carlo Dumlao	501018239	C.D

By signing above you attest that you have contributed to this written lab report and confirm that all work you have contributed to this lab report is your own work. Any suspicion of copying or plagiarism in this work will result in an investigation of Academic Misconduct and may result in a 0 on the work, an F in the course, or possibly more severe penalties, as well as a Disciplinary Notice on your academic record under the Student Code of Academic Conduct, which can be found online at: www.ryerson.ca/senate/current/pol60.pdf.

Table of Contents

Abstract

This project is intended to introduce the functionality and behavior of caching – in particular, how it fetches and writes a requested data from CPU in the case of a cache hit, and how it transfers a block between two different memories including SRAM and SDRAM to resolve a cache miss. To accomplish this, the state and process diagram are firstly constructed to realize the four different functionalities (or cases) of the cache: hit and CPUwr, hit and not CPUwr, miss and dirty, miss and not dirty. Afterwards, the block diagram (or sub-components) of the cache controller system is created, which is then implemented via a VHDL code within the Xilinx ISE CAD environment. With the provided CPU generator and a coded SDRAM, the correctness of the designed cache controller is simulated under the Xilinx Spartan-3E FPGA. A quantity analysis is finally conducted to determine its performance for each of the mentioned cases. The result shows that the cache itself provides a faster communication link between the CPU and a slower high-volume memory (SDRAM), ideally. However, if the cache is subject to higher miss rate, it will significantly reduce the performance of a device – may be much worse than when the CPU directly communicates with the main memory, without the cache system.

Introduction

With the ever-growing complexity of applications and softwares, it is no doubt that memory size is a crucial part in the development of modern technologies. Of course, one can directly satisfy those memory-hungry apps by implementing additional cells within a relatively inexpensive memory, for instance, an SDRAM. However, there is a trade-off: when the volume of the main memory is increased, its undesired impedance will also increase. Put simply, the performance of the device will be hindered as the communication between the CPU and the SDRAM now experiences a longer propagation delay. To even realize and arrive at an effective solution for this performance issue, one must firstly understand the concept of caching.

A cache is usually composed of SRAM storage, and in terms of instantly reading/writing data, it reigns over the SDRAM. Its memory volume is only limited, however – but by taking advantage of temporal and spatial locality, it can be effectively utilized to improve the overall memory speed and data rate. The cache SRAM is therefore strictly reserved for elements that are likely to be used by the CPU in the near future.This collection of elements is grouped into blocks. Each block has its own status bits including index, tag, valid, and dirty bits. Every time a CPU requests data, the cache controller firstly checks those status bits then decides whether the desired data can be directly accessed in the fast memory SRAM, or whether transfer of block from (or to) the SDRAM is required.

The objective of this project is to explore the varying behavior of the designed cache controller, including a cache hit and cache miss; and how this behavior helps achieve the “bridge” and a faster communication link between the CPU and main memory, as well as some of its tradeoffs. Furthermore, it is to become knowledgeable of measuring the performance of a cache such as hit/miss determination time, block replacement time, hit time, and miss penalty time.

System Specifications

There are 4 possible actions that can be taken by the cache controller. Based on the CPU command and the stored identifiers for a particular block, the controller will choose and execute the appropriate action, which is described below:

1. Hit and CPUwr: if the requested data is in the cache, then simply override that data.
2. Hit and not CPUwr: if the requested data is in the cache, then directly fetch it to the CPU.
3. Miss and not Dirty: if the requested data does not exist in the cache and the block to be replaced has not been modified (ie. not Dirty), then do block transfer from SDRAM to SRAM to obtain the missing data. Finally, read or write that data in the cache.
4. Miss and Dirty: if the requested data does not exist in the cache and the block to be replaced has been modified (ie. Dirty), update the SDRAM to reflect the modified block, then do block transfer from SDRAM to SRAM to obtain the missing data. Finally, read or write that data in the cache.

Device Description / Design

Symbol

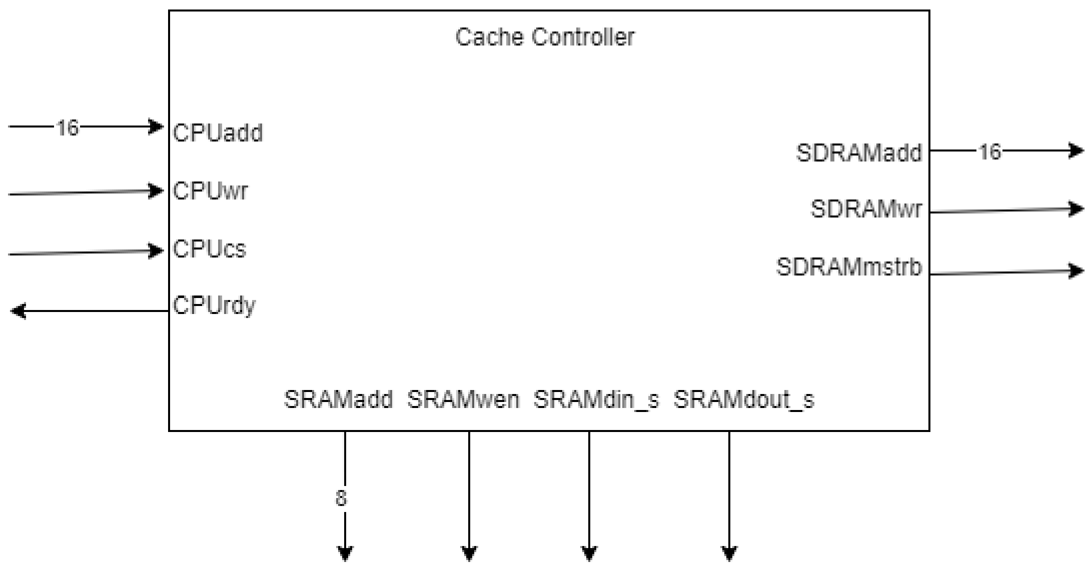

Figure 1: Symbol of the cache controller.

Table 1: Description of the input and output of the symbol.

Input and Output	Description
CPUadd	Address of the requested data from CPU
CPUwr	Indicates if the CPU is requesting a write operation
CPUcs	Triggers the cache controller; run the cache sequence of states
CPUrdy	Notifies the CPU that the cache has completed its execution
SDRAMadd	SDRAM address
SDRAMwr	SDRAM write operation
SDRAMmstrb	SDRAM memory strobe
SRAMadd	SRAM address
SRAMwen	SRAM write operation
SRAMdin_s	Selector for the 2:1 MUX that is connected to the SRAM din
SRAMdout_s	Selector for the 1:2 DEMUX that is connected to the SRAM dout

Block Diagrams

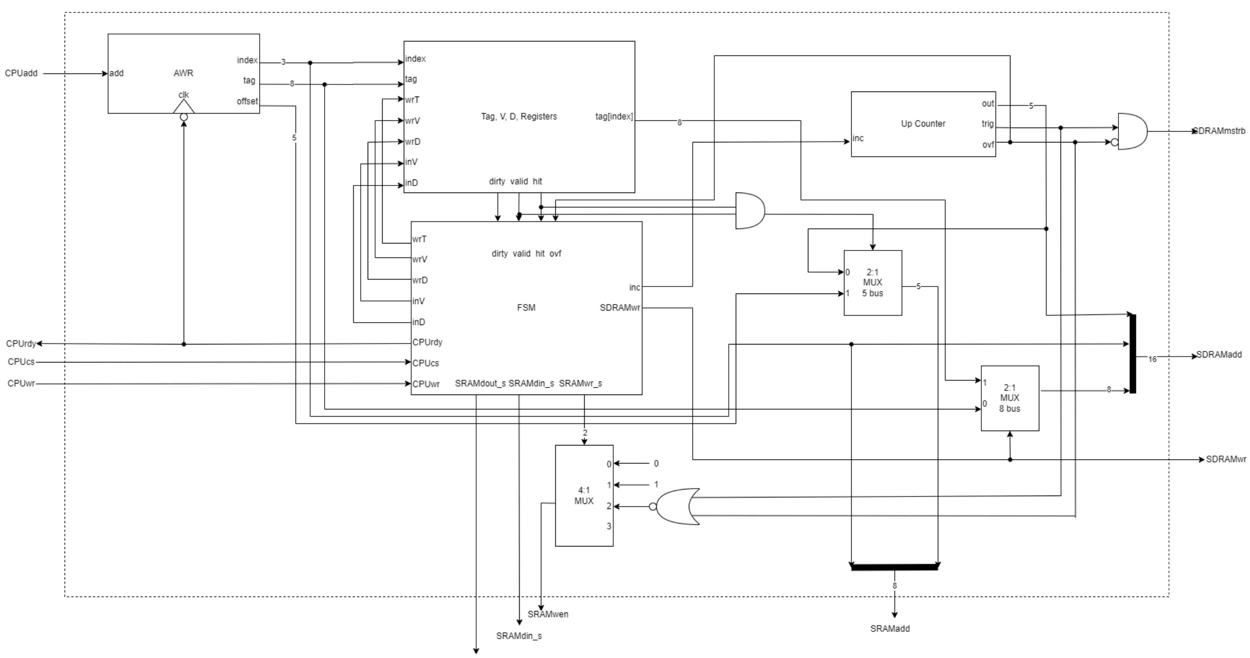

Figure 2: Block diagram of the cache controller.

Table 2: Description of the sub-components.

Sub-component	Description
AWR	• This is an address word register. Once the FSM starts running or becomes busy (CPUrdy = 0), AWR will latch the incoming address of the CPU and immediately split it into 3 signals – that is, tag, index, and offset.
FSM	• The state machine of the cache controller. Refer to Figure X.
Tag, V, D Registers	• Stores the tag, valid and dirty bit of a block • Notifies the FSM whether the requested data is a hit or miss, whether the selected block is dirty, and whether it is invalid.
Up Counter	• Crucial for transferring a block between the SRAM and SDRAM: ○ Generates an incrementing value from “00000” to “11111” for the SRAMadd and SDRAMadd ○ Generates a periodic signal which has a period of 3 clock cycles to synchronize the reading and writing between the two memories ○ Notifies the FSM whether it has reached an overflow
4:1 MUX	• Controls the writing operation of the SRAM ○ 0: Read ○ 1: Write a word ○ 2: Write a block
2:1 MUX 5 bus	• Determines the address of the SRAM: ○ 0: if it is miss or not valid, let the least significant 5 bits of the SRAMadd to increment from “00000” to “11111”, generated by the Up Counter. ○ 1: Otherwise, let that 5 bits be the CPU offset.
2:1 MUX 8 bus	• Determines the address of the SDRAM: ○ 0: for Case 3 (ie. SDRAMwr = 0), let the most significant byte of the SDRAMadd be the CPU tag ○ 1: for Case 4 (ie. SDRAMwr = 1), let that byte be the tag of the selected block to be replaced.

State Diagram

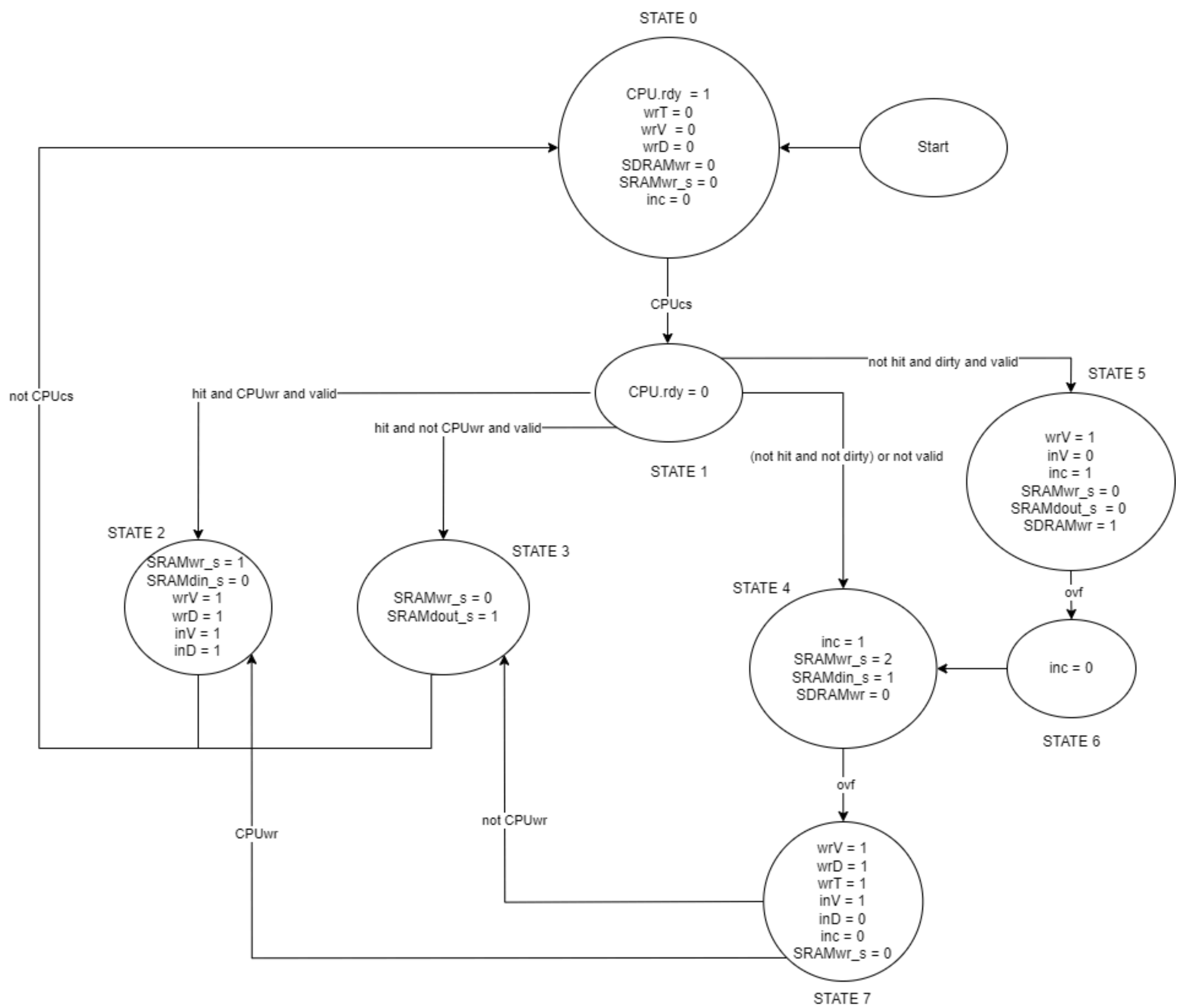

Figure 3: State diagram of the cache controller.

Table 3: Description of the states.

State	Description
0	● Notify the CPU that it is ready for operation ● Turn off all writing operations, including the Tag, V, D Registers, SRAM, SDRAM, and Up Counter
1	● Notify the CPU that it is busy ● Consequently, the AWR will latch the CPU address; and Tag, V, D Registers will determine if the requested data is a hit or miss.
2	● Replace a word in the SRAM with the CPUdout

3	• Fetch the requested data in the SRAM for the CPUdin
4	• Block transfer from SDRAM to SRAM: ○ Trigger the Up Counter ○ Do reading operation in the SDRAM; do writing operation in the SRAM
5	• Block transfer from SRAM to SDRAM: ○ Trigger the Up Counter ○ Do reading operation in the SRAM; do writing operation in the SDRAM
6	• Before proceeding Case 3, reset the Up Counter
7	• Update the status bits in the Tag, V, D Registers accordingly

Process Diagram

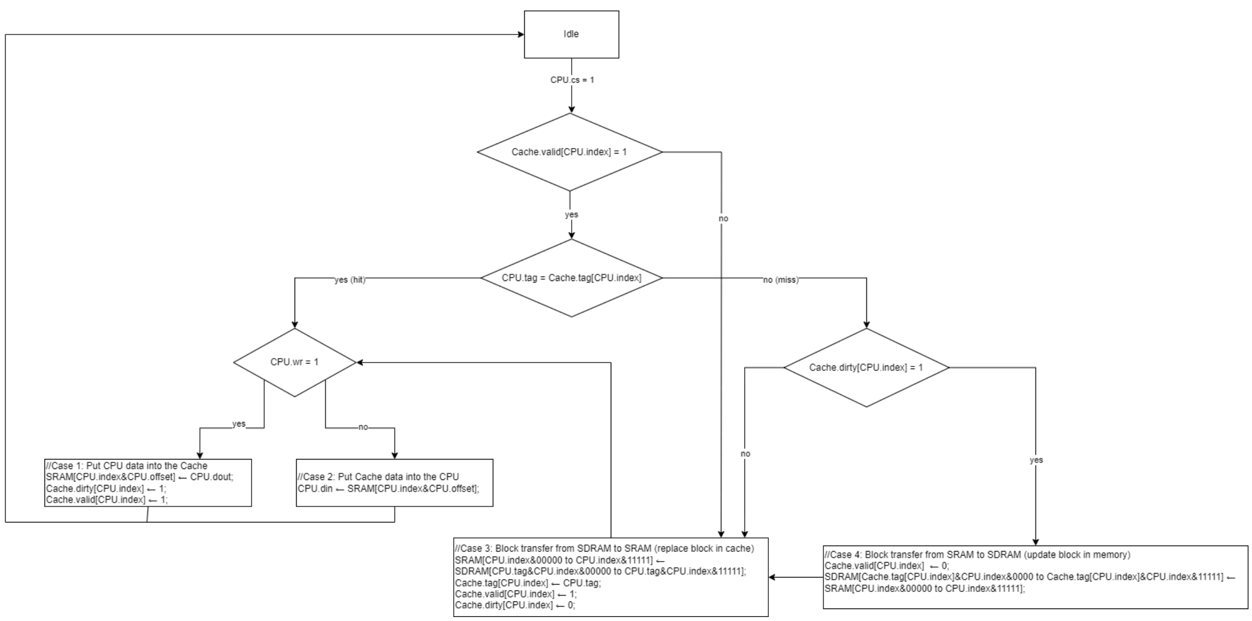

Figure 4: Process diagram of cache controller.

Results

Note: The following ChipScope timing diagrams are based on the provided CPU_gen.

Case 1: hit and CPUwr

Pay attention to the value of the CPUdout between 211ns and 221ns (green and blue markers) in Figure 5. When the requested data is a hit and the CPU wants to do a writing operation with CPUdout = 0xBB, the CPUin which is currently connected to the output of SRAM transitions to 0xBB just right after the cache becomes busy. This indicates that the CPUdout has been successfully saved/stored in the SRAM.

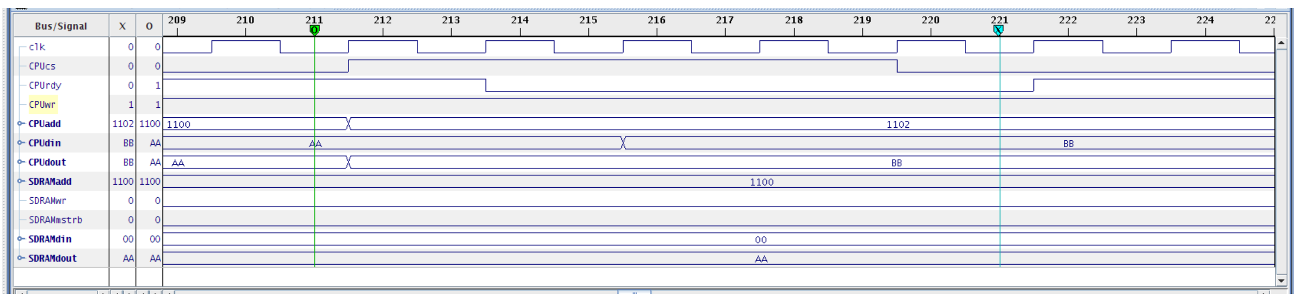

Figure 5: ChipScope diagram for case 1.

Case 2: hit and not CPUwr

In FIgure 6, the CPU wishes to read a word at the address 0x1100, and a cache hit has occurred. Notice that at 231 – when the CPUrdy becomes low – the cache correctly fetches the data 0xAA at the specified address into the data input of the CPU (CPUdin),

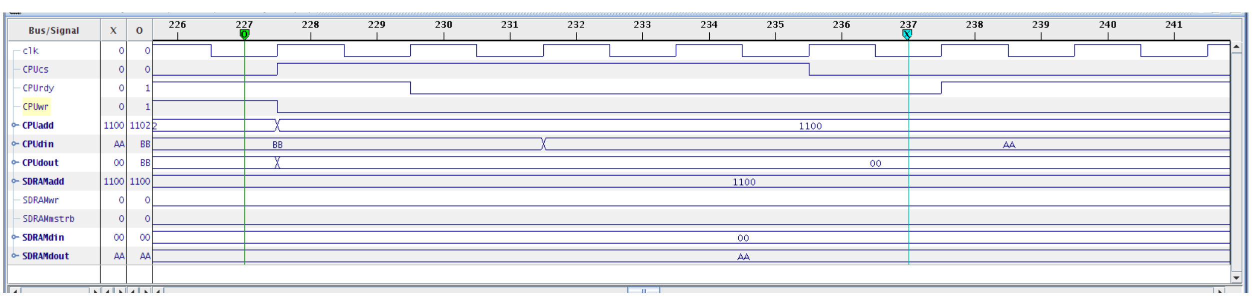

Figure 6: ChipScope diagram for case 2.

Case 3: miss and not dirty

The diagram below showcases a scenario when the CPU wants to write a 0xAA at address 0x1100, and it results in a cache miss and the selected block is not dirty. Take a closer look at inputs and outputs of the SDRAM in Figure 7, specifically its address, memory strobe, and data out. The most significant byte of SDRAMadd is the same as the upper byte of the CPUadd; and for every 3 clock cycles, that address continues to increment from 0x1100 to 0x1111 to fetch the SDRAM block of 32 words . It can also be seen that before the cache fetches the next word in the SDRAM, the SDRAMmstrb is triggered to initiate the reading operation. By observing the stream of data in SDRAMdout, the data at 0x1100 and 0x1102 are 0xAA and 0xBB, respectively, as expected by the CPU_gen. Notice that once the block transfer has been completed at time 200, the CPUdin has been updated to 0xAA, indicating that the CPUdout has been successfully written in the SRAM.

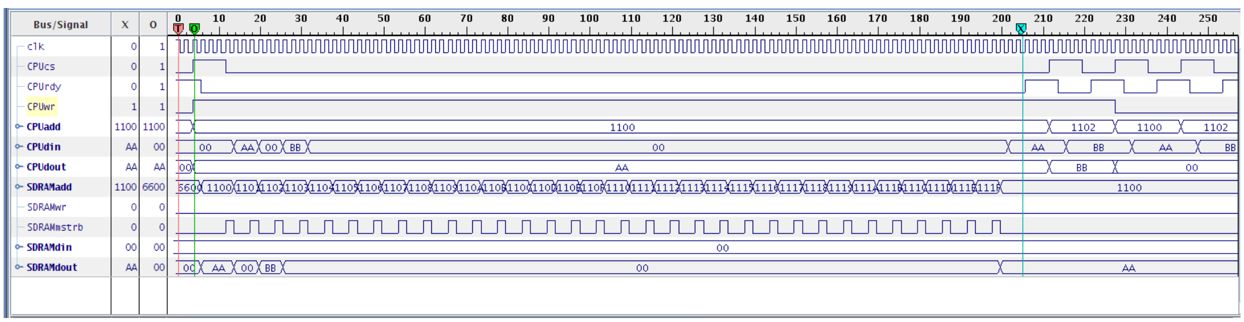

Figure 7: ChipScope diagram for case 3.

Case 4: miss and dirty

The figures below presents Case 4, that is, the CPU wants to read the data at address 0x6606 and it results in cache miss and the selected block is dirty. Pay attention to the SDRAMadd and SDRAMwr. Notice that in Figure 9, initially, the most significant byte of SDRAMadd is the same as the tag (ie. 0x55) of the selected block and the SDRAM is in write operation, demonstrating the block transfer from SRAM to SDRAM. Furthermore, at time 205 – when the block in SDRAM now reflects the modified/dirty block – the upper byte of SDRAMadd is the same as upper byte of CPUadd and SDRAMwr is low, indicating that there is block transfer from SDRAM to SRAM. In Figure 8, at time 400 after the block has been transferred, the data input of the CPU received a value of 0x00, as expected considering the CPU_gen made no modifications or write operations at address 0x6606.

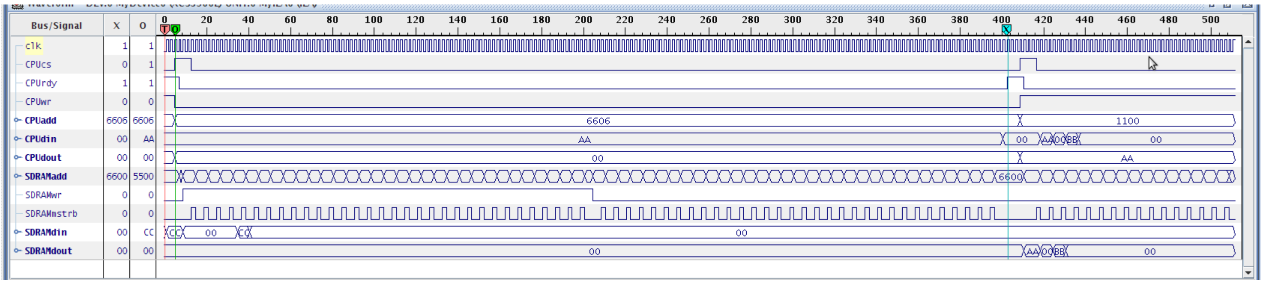

Figure 8: ChipScope diagram for case 4.

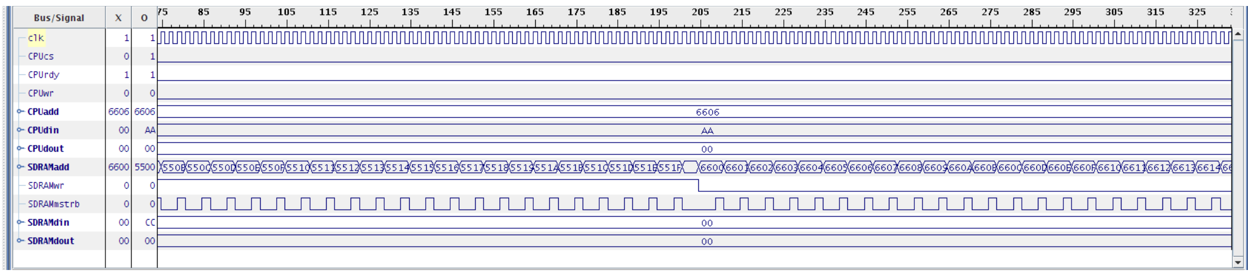

Figure 9: Zoomed in version of Figure 8 showcasing the varying value of SRAMadd.

Performance

In regards to analyzing the cache performance parameters, note that:

- A word transfer between SRAM and SDRAM requires 3 cycles. The first cycle is for calculating the address to be fetched in the block, second cycle is to wait for the SRAM to output the correct data (for Case 4), and the third cycle is to trigger the SDRAMmstrb when the SRAM output data has been stabilized.
- The Tag, V, D Registers is an asynchronous component. That is, once that AWR has been latched, the hit/miss will immediately be determined.
- A single clock signal triggers the FSM to transition from one state to another

Table 4: Performance of the cache for each case.

N	Cache performance parameter	Time (clock cycle)
1	Hit/Miss determination time	(# of cycles to transition state 0 → 1) = 1
2	Data access time	(# of cycles to transition state 0 → 1 → 2) = 2
3	Block replacement time	(# of words in block) x (# of cycles for word transfer) = 32 words x 3 cycles = 96
4	Hit time (Case 1 and 2)	(# of cycles to transition state 0 → 1 → 2 → 0) = 3

5	Miss penalty for Case 3 (when D-bit = 0)	(# of cycles to transition state 0 → 1 → 4 → 7 → 2 → 0) + (Block Replacement Time) = 5 + 96 = 101
6	Miss penalty for Case 4 (when D-bit = 1)	(# of cycles to transition state 0 → 1 → 5 → 6 → 4 → 7 → 2 → 0) + 2x(Block Replacement Time) = 7 + 2 x 96 = 199

Conclusion

In terms of performance, the best-case scenario of the designed cache is when the requested data already exists in the SRAM, that is, a cache hit. As shown in Table 4, the data can be retrieved or written after 3 cycles – put it simply, the CPU, whose CPUcs is asserted for 4 clock cycles, only needs to wait a single cycle before the cache becomes ready for operation again. On the other hand, the worst case scenario is a cache miss: the requested data is not stored in the SRAM. This scenario always requires the cache to initially do a block transfer between the cache and the main memory (SDRAM), in order to modify or fetch the requested data. Hence, when the selected block is not dirty, the cache needs 101 cycles; but if it is dirty, the performance is reduced by two-fold, requiring 199 cycles. As a result, if the cache occasionally experiences the latter scenario – for instance, the initial boot up of an application or program – the CPU needs to wait a longer time than usual before it can proceed to the next instruction. Such implementation of a cache is therefore undesirable when hard-time systems are concerned, since the time required for a block transfer may inevitably result in missed deadlines of processes.

Appendix

Cache + CPU + SDRAM

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;


entity test is
   Port ( clk : in  STD_LOGIC);
end test;

architecture Behavioral of test is
		component CPU_gen
			Port (
			clk				: in  STD_LOGIC;
    rst				: in  STD_LOGIC;
    trig				: in  STD_LOGIC;
			-- Interface to the Cache Controller.
    Address	: out  STD_LOGIC_VECTOR (15 downto 0);
    wr_rd	: out  STD_LOGIC;
    cs				: out  STD_LOGIC;
    DOut				: out  STD_LOGIC_VECTOR (7 downto 0)
			);
		end component;

		component cache_control
			Port(		CPUadd : in STD_LOGIC_VECTOR (15 downto 0);
						CPUwr : in STD_LOGIC;
						CPUcs : in STD_LOGIC;
						clk : in STD_LOGIC;
						mux_i0 : in STD_LOGIC_VECTOR (7 downto 0);
						mux_i1 : in STD_LOGIC_VECTOR (7 downto 0);
						dmux_o0 : out STD_LOGIC_VECTOR (7 downto 0);
						dmux_o1 : out STD_LOGIC_VECTOR (7 downto 0);
						CPUrs : out STD_LOGIC;
						SDRAMwr : out STD_LOGIC;
						SDRAMadd : out STD_LOGIC_VECTOR(15 downto 0);
						SDRAMmstrb : out STD_LOGIC);
		end component;

		component sdram
			Port ( add : in  STD_LOGIC_VECTOR (15 downto 0);
						 wr : in  STD_LOGIC;
						 mstrb : in  STD_LOGIC;
						 din : in  STD_LOGIC_VECTOR (7 downto 0);
						 dout : out  STD_LOGIC_VECTOR (7 downto 0));
		end component;

		component icon
			PORT (
				CONTROL0 : INOUT STD_LOGIC_VECTOR(35 DOWNTO 0)
			);
		end component;

		component ila
			PORT (
				CONTROL : INOUT STD_LOGIC_VECTOR(35 DOWNTO 0);
				CLK : IN STD_LOGIC;
				DATA : IN STD_LOGIC_VECTOR(255 DOWNTO 0);
				TRIG0 : IN STD_LOGIC_VECTOR(7 DOWNTO 0)
			);
		end component;

		signal control0 : std_logic_vector(35 downto 0);
		signal ila_data : std_logic_vector(255 downto 0);
		signal trig0 : std_logic_vector(7 downto 0);

		signal CPUadd : std_logic_vector (15 downto 0);
		signal CPUwr : std_logic;
		signal CPUcs : std_logic;
		signal CPUrdy : std_logic;
		signal CPUdout : std_logic_vector (7 downto 0);
```

```
        signal CPUdin : std_logic_vector (7 downto 0);

        signal SDRAMadd : std_logic_vector (15 downto 0);
        signal SDRAMwr : std_logic;
        signal SDRAMmstrb : std_logic;
        signal SDRAMdin : std_logic_vector (7 downto 0);
        signal SDRAMdout : std_logic_vector (7 downto 0);

        signal i : std_logic_vector(1 downto 0) := "00";

begin

        sys_icon : icon port map (
                CONTROL0 => control0
        );

        sys_ila : ila port map (
                CONTROL => control0,
                CLK => clk,
                DATA => ila_data,
                TRIG0 => trig0
        );

        sys_cpu : CPU_gen port map(
                clk => i(0),
  rst => '0',
  trig => CPUrdy,
  Address  => CPUadd,
  wr_rd => CPUwr,
  cs => CPUcs,
  DOut => CPUdout
        );

        sys_cache : cache_control port map(
                CPUadd => CPUadd,
                CPUwr => CPUwr,
                CPUcs => CPUcs,
                clk => i(0),
                mux_i0 => CPUdout,
                mux_i1 => SDRAMdout,
                dmux_o0 => SDRAMdin,
                dmux_o1 => CPUdin,
                CPUrs => CPUrdy,
                SDRAMwr => SDRAMwr,
                SDRAMadd => SDRAMadd,
                SDRAMmstrb => SDRAMmstrb
        );

        sys_sdram : sdram port map (
                add => SDRAMadd,
                wr => SDRAMwr,
                mstrb => SDRAMmstrb,
                din => SDRAMdin,
                dout => SDRAMdout
        );

        process(clk)
        begin
                if(clk'Event and clk = '1') then
                        i <= i + '1';
                end if;
        end process;

                ila_data(0) <= i(0);
                ila_data(16 downto 1) <= SDRAMadd;
                ila_data(17) <= SDRAMwr;
                ila_data(18) <= SDRAMmstrb;
                ila_data(26 downto 19) <= SDRAMdin;
                ila_data(34 downto 27) <= SDRAMdout;
                ila_data(50 downto 35) <= CPUadd;
                ila_data(51) <= CPUwr;
                ila_data(52) <= CPUcs ;
                ila_data(60 downto 53) <= CPUdout;
                ila_data(68 downto 61) <= CPUdin;
                ila_data(69) <= CPUrdy;
                ila_data(255 downto 70) <= (others => '0');


end Behavioral;
```

Cache: Cache Controller + SRAM

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity cache_control is
	Port(			CPUadd : in STD_LOGIC_VECTOR (15 downto 0);
				CPUwr : in STD_LOGIC;
				CPUcs : in STD_LOGIC;
				clk : in STD_LOGIC;
				mux_i0 : in STD_LOGIC_VECTOR (7 downto 0);
				mux_i1 : in STD_LOGIC_VECTOR (7 downto 0);
				dmux_o0 : out STD_LOGIC_VECTOR (7 downto 0);
				dmux_o1 : out STD_LOGIC_VECTOR (7 downto 0);
				CPUrs : out STD_LOGIC;
				SDRAMwr : out STD_LOGIC;
				SDRAMadd : out STD_LOGIC_VECTOR(15 downto 0);
				SDRAMmstrb : out STD_LOGIC);
end cache_control;

architecture Behavioral of cache_control is
	signal CPUindex : STD_LOGIC_VECTOR (2 downto 0);
	signal CPUtag : STD_LOGIC_VECTOR (7 downto 0);
	signal CPUoffset : STD_LOGIC_VECTOR (4 downto 0);

	signal UCinc : STD_LOGIC;
	signal UCout : STD_LOGIC_VECTOR (4 downto 0);
	signal UCbit0 : STD_LOGIC;
	signal UCovf : STD_LOGIC;

	signal SRAMwrs: STD_LOGIC_VECTOR (1 downto 0);
	signal SRAMdout : STD_LOGIC;
	signal SRAMdin : STD_LOGIC;
	signal SRAMwen : STD_LOGIC;
	signal SRAMadd : STD_LOGIC_VECTOR(7 downto 0);
	signal SRAMin : STD_LOGIC_VECTOR(7 downto 0);
	signal SRAMout : STD_LOGIC_VECTOR(7 downto 0);

	signal SDRAM_wr : STD_LOGIC;

	signal MUX5bus_out : STD_LOGIC_VECTOR (4 downto 0);
	signal MUX8bus_out : STD_LOGIC_VECTOR (7 downto 0);

	signal TVDwrT : STD_LOGIC;
	signal TVDwrV : STD_LOGIC;
	signal TVDwrD : STD_LOGIC;
	signal TVDinD : STD_LOGIC;
	signal TVDinV : STD_LOGIC;
	signal TVDtag_out : STD_LOGIC_VECTOR (7 downto 0);
	signal TVDhit : STD_LOGIC;
	signal TVDdirty : STD_LOGIC;
	signal TVDvalid : STD_LOGIC;

	signal MUX5bus_select : STD_LOGIC;
	signal rs : STD_LOGIC;

	component awr
	Port ( addr : in  STD_LOGIC_VECTOR (15 downto 0);
		clk : in STD_LOGIC;
	index : out  STD_LOGIC_VECTOR (2 downto 0);
	tag : out  STD_LOGIC_VECTOR (7 downto 0);
    offset : out  STD_LOGIC_VECTOR (4 downto 0));
	end component;

	component fsm
	Port ( CPUwr : in  STD_LOGIC;
    CPUcs : in  STD_LOGIC;
    UCovf : in  STD_LOGIC;
    TVDhit : in  STD_LOGIC;
	  clk: in  STD_LOGIC;
	  TVDdirty: in  STD_LOGIC;
    CPUrs : out  STD_LOGIC;
    SRAMdout : out  STD_LOGIC;
    SRAMdin : out  STD_LOGIC;
    SRAMwrs : out  STD_LOGIC_VECTOR (1 downto 0);
    SDRAMwr : out  STD_LOGIC;
    UCinc : out  STD_LOGIC;
    TVDwrT : out  STD_LOGIC;
    TVDwrV : out  STD_LOGIC;
    TVDwrD : out  STD_LOGIC;
```

```
        TVDinV : out  STD_LOGIC;
        TVDinD : out  STD_LOGIC;
              TVDvalid : in STD_LOGIC);
            end component;

            component mux2to1_8bus
                    Port ( s : in  STD_LOGIC;
        i0 : in  STD_LOGIC_VECTOR (7 downto 0);
        i1 : in  STD_LOGIC_VECTOR (7 downto 0);
        o : out  STD_LOGIC_VECTOR (7 downto 0));
            end component;

            component mux2to1
                    Port ( s : in  STD_LOGIC;
        i0 : in  STD_LOGIC_VECTOR (4 downto 0);
        i1 : in  STD_LOGIC_VECTOR (4 downto 0);
        o : out  STD_LOGIC_VECTOR (4 downto 0));
            end component;

            component muxforSRAMwen
                    Port ( s : in  STD_LOGIC_VECTOR (1 downto 0);
            i1 : in  STD_LOGIC;
                    i2 : in  STD_LOGIC;
            o : out  STD_LOGIC);
            end component;

            component counter
                    Port ( inc : in  STD_LOGIC;
                    clk : in STD_LOGIC;
            output : out  STD_LOGIC_VECTOR (4 downto 0);
            bit0 : out  STD_LOGIC;
            overflow : out  STD_LOGIC);
            end component;

            component sram
                    Port ( clk : in  STD_LOGIC;
            wr : in  STD_LOGIC;
            din : in  STD_LOGIC_VECTOR (7 downto 0);
            dout : out  STD_LOGIC_VECTOR (7 downto 0);
            add : in  STD_LOGIC_VECTOR (7 downto 0));
            end component;

            component demu1to2_8bus
                    Port ( i : in  STD_LOGIC_VECTOR (7 downto 0);
                    o0 : out  STD_LOGIC_VECTOR (7 downto 0);
                    o1 : out  STD_LOGIC_VECTOR (7 downto 0);
                    s : in  STD_LOGIC);
            end component;

            component tagvd
                    Port (                  index : in  STD_LOGIC_VECTOR (2 downto 0);
                                            tag_in : in  STD_LOGIC_VECTOR (7 downto 0);
                                            tag_wr : in  STD_LOGIC;
                                            v_in : in  STD_LOGIC;
                                            d_in : in  STD_LOGIC;
                                            v_wr : in  STD_LOGIC;
                                            d_wr : in  STD_LOGIC;
                                            tag_out : out  STD_LOGIC_VECTOR (7 downto 0);
                                            hit : out  STD_LOGIC;
                                            valid : out STD_LOGIC;
                                            dirty : out STD_LOGIC
                                            );
            end component;

begin


            sys_awr : awr port map(    addr => CPUadd,
                                       clk => rs,
                                       index => CPUindex,
                                       tag => CPUtag,
                                       offset => CPUoffset);

            sys_tvd : tagvd port map(   index => CPUindex,
                                               tag_in => CPUtag,
                                               tag_wr => TVDwrT,
                                               v_in => TVDinV,
                                               d_in => TVDinD,
                                               v_wr => TVDwrV,
                                               d_wr => TVDwrD,
                                               tag_out => TVDtag_out,
```

```
                                        hit => TVDhit,
                                        valid => TVDvalid,
                                        dirty => TVDdirty
                                        );

        sys_fsm : fsm port map( CPUwr => CPUwr,
                                CPUcs => CPUcs,
                                UCovf => UCovf,
                                TVDhit => TVDhit,
                                clk => clk,
                                TVDdirty => TVDdirty,
                                CPUrs => rs,
                                SRAMdout => SRAMdout,
                                SRAMdin => SRAMdin,
                                SRAMwrs => SRAMwrs,
                                SDRAMwr => SDRAM_wr,
                                UCinc => UCinc,
                                TVDwrT => TVDwrT,
                                TVDwrV => TVDwrV,
                                TVDwrD => TVDwrD,
                                TVDinV => TVDinV,
                                TVDinD => TVDinD,
                                TVDvalid => TVDvalid
                                );

        sys_counter : counter port map( inc => UCinc,
                                        clk => clk,
                                        output => UCout,
                                        bit0 => UCbit0,
                                        overflow => UCovf);

        sys_mux2to1_8bus : mux2to1_8bus port map(s => SDRAM_wr,
                                                i0 => CPUtag,
                                                i1 => TVDtag_out,
                                                o => MUX8bus_out);


        MUX5bus_select <= TVDhit and TVDvalid;
        sys_mux2to1_5bus : mux2to1 port map(    s => MUX5bus_select,
                                                i0 => UCout,
                                                i1 => CPUoffset,
                                                o => MUX5bus_out);


        sys_muxforSRAMwen : muxforSRAMwen port map(  s => SRAMwrs,
                                                        i1 => UCovf,
                                                        i2 => UCbit0,
                                                        o => SRAMwen);
                                                        SRAMadd(4 downto 0) <= MUX5bus_out;
                                                        SRAMadd(7 downto 5) <= CPUindex;
                                                        SDRAMmstrb <= UCbit0 and not(UCovf);
                                                        SDRAMwr <= SDRAM_wr;
                                                        SDRAMadd(4 downto 0) <= UCout;
                                                        SDRAMadd(7 downto 5) <= CPUindex;
                                                        SDRAMadd(15 downto 8) <= MUX8bus_out;
                                                        CPUrs <= rs;

        sys2_mux2to1_8bus : mux2to1_8bus port map(    s => SRAMdin,
                                                        i0 => mux_i0,
                                                        i1 => mux_i1,
                                                        o => SRAMin);


        sys_sram : sram PORT MAP (      clk => clk,
                                        wr => SRAMwen,
                                        din => SRAMin,
                                        dout => SRAMout,
                                        add => SRAMadd);

        sys_demux1to2_8bus : demu1to2_8bus port map(  i => SRAMout,
                                                        o0 => dmux_o0,
                                                        o1 => dmux_o1,
                                                        s => SRAMdout);

end Behavioral;
```

AWR

```
library IEEE;
```

```
use IEEE.STD_LOGIC_1164.ALL;

entity awr is
    Port ( addr : in  STD_LOGIC_VECTOR (15 downto 0);
                                                clk : in STD_LOGIC;
           index : out  STD_LOGIC_VECTOR (2 downto 0);
           tag : out  STD_LOGIC_VECTOR (7 downto 0);
           offset : out  STD_LOGIC_VECTOR (4 downto 0));
end awr;

architecture Behavioral of awr is
                signal temp : STD_LOGIC;
begin
                temp <= not clk;
                process(temp)
                begin
                                if(temp'Event and temp = '1') then
                                                offset <= addr(4 downto 0);
                                                index <= addr(7 downto 5);
                                                tag <= addr(15 downto 8);
                                end if;
                end process;

end Behavioral;
```

Tag, V, D Registers

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;


entity tagvd is
    Port ( index : in  STD_LOGIC_VECTOR (2 downto 0);
           tag_in : in  STD_LOGIC_VECTOR (7 downto 0);
           tag_wr : in  STD_LOGIC;
           v_in : in  STD_LOGIC;
           d_in : in  STD_LOGIC;
           v_wr : in  STD_LOGIC;
           d_wr : in  STD_LOGIC;
           tag_out : out  STD_LOGIC_VECTOR (7 downto 0);
           hit : out  STD_LOGIC;
                                                valid : out STD_LOGIC;
                                                dirty : out STD_LOGIC
                                               );
end tagvd;

architecture Behavioral of tagvd is
                type memory is array (0 to 7) of std_logic_vector(7 downto 0);
                type mybit is array (0 to 7) of std_logic;
                signal tag_reg : memory := (others => (others => '0'));
                signal v_bit : mybit := (others => '0');
                signal d_bit : mybit := (others => '0');
begin

                process(index, tag_in, tag_wr)
                begin

                                if(tag_wr = '1' or tag_in = tag_reg(to_integer(unsigned(index)))) then
                                                hit <= '1';
                                else
                                                hit <= '0';
                                end if;

                                if(tag_wr = '1') then
                                                tag_reg(to_integer(unsigned(index))) <= tag_in;
                                end if;

                end process;

                process(index, v_wr, v_in)
                begin
                                if(v_wr = '1') then
                                                v_bit(to_integer(unsigned(index))) <= v_in;
                                end if;
                end process;

                process(index, d_wr, d_in)
```

```
        begin
                if(d_wr = '1') then
                        d_bit(to_integer(unsigned(index))) <= d_in;
                end if;

        end process;

        tag_out <= tag_reg(to_integer(unsigned(index)));
        dirty <= d_bit(to_integer(unsigned(index)));
        valid <= v_bit(to_integer(unsigned(index)));

end Behavioral;
```

FSM

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;

entity fsm is
   Port ( CPUwr : in  STD_LOGIC;
          CPUcs : in  STD_LOGIC;
          UCovf : in  STD_LOGIC;
          TVDhit : in  STD_LOGIC;
                          clk: in  STD_LOGIC;
                          TVDdirty: in  STD_LOGIC;
          CPUrs : out  STD_LOGIC;
          SRAMdout : out  STD_LOGIC;
          SRAMdin : out  STD_LOGIC;
          SRAMwrs : out  STD_LOGIC_VECTOR (1 downto 0);
          SDRAMwr : out  STD_LOGIC;
          UCinc : out  STD_LOGIC;
          TVDwrT : out  STD_LOGIC;
          TVDwrV : out  STD_LOGIC;
          TVDwrD : out  STD_LOGIC;
          TVDinV : out  STD_LOGIC;
          TVDinD : out  STD_LOGIC;
                          TVDvalid : in STD_LOGIC
                          );
end fsm;

architecture Behavioral of fsm is
        signal state : integer := 8;
begin
        process(clk)
        begin
                if(clk'Event and clk = '1') then
                        if(state = 0) then
                                if(CPUcs = '1') then
                                        state <= 1;
                                end if;

                        elsif(state = 1) then
                                if(TVDvalid = '0' or (TVDhit = '0' and TVDdirty = '0')) then
                                        state <= 4;
                                elsif(TVDhit = '1' and CPUwr = '1') then
                                        state <= 2;
                                elsif(TVDhit = '1' and CPUwr = '0') then
                                        state <= 3;
                                elsif(TVDhit = '0' and TVDdirty = '1') then
                                        state <= 5;
                                end if;

                        elsif(state = 2) then
                                if(CPUcs = '0') then
                                        state <= 0;
                                end if;

                        elsif(state = 3) then

                                if(CPUcs = '0') then
                                        state <= 0;
                                end if;

                        elsif(state = 4) then

                                if(UCovf = '1') then
                                        state <= 6;
```

```
                        end if;

                elsif(state = 5) then

                        if(UCovf = '1') then
                                state <= 7;
                        end if;

                elsif(state = 6) then

                        if(CPUwr = '1') then
                                state <= 2;
                        else
                                state <= 3;
                        end if;

                elsif(state = 7) then
                        state <= 4;

                elsif(state = 8) then
                        state <= 9;
                elsif(state = 9) then
                        state <= 0;
                end if;
        end if;
end process;

process(state)
begin
        if(state = 0) then
                        CPUrs  <= '1';
                        TVDwrT <= '0';
                        TVDwrV  <= '0';
                        TVDwrD <= '0';
                        SDRAMwr <= '0';
                        SRAMwrs <= "00";
                        UCinc <= '0';

        elsif(state = 1) then
                        CPUrs <= '0';

        elsif(state = 2) then
                        SRAMwrs <= "01";
                        SRAMdin <= '0';
                        TVDwrV <= '1';
                        TVDwrD <= '1';
                        TVDinV <= '1';
                        TVDinD <= '1';

        elsif(state = 3) then
                        SRAMwrs <= "00";
                        SRAMdout <= '1';

        elsif(state = 4) then
                        UCinc <= '1';
                        SRAMwrs <= "10";
                        SRAMdin <= '1';
                        SDRAMwr <= '0';

        elsif(state = 5) then
                        TVDwrV <= '1';
                        TVDinV <= '0';
                        UCinc <= '1';
                        SRAMwrs <= "00";
                        SRAMdout  <= '0';
                        SDRAMwr <= '1';

        elsif(state = 6) then
                        TVDwrV <= '1';
                        TVDwrD <= '1';
                        TVDwrT <= '1';
                        TVDinV <= '1';
                        TVDinD <= '0';
                        UCinc <= '0';
                        SRAMwrs <= "00";

  elsif(state = 7) then
                        UCinc <= '0';

        elsif(state = 8 or state = 9) then
                        CPUrs <= '0';
```

```
                end if;
        end process;

end Behavioral;
```

Up Counter

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity counter is
   Port ( inc : in  STD_LOGIC;
                                        clk : in STD_LOGIC;
          output : out  STD_LOGIC_VECTOR (4 downto 0);
          bit0 : out  STD_LOGIC;
          overflow : out  STD_LOGIC);
end counter;

architecture Behavioral of counter is
        signal counter:std_logic_vector(6 downto 0) := (others => '0');
        signal ovf:std_logic := '0';
begin
        process(clk, inc)
        begin
                if(inc = '0') then
                        counter <= (others => '0');
                        ovf <= '0';
                elsif(clk'Event and clk = '1') then
                        if(inc = '1') then
                                if(counter  = "1111111") then
                                        ovf <= '1';
                                end if;

                                if(counter(1 downto 0) = "01") then
                                        counter(1 downto 0) <= "11";
                                else
                                  counter <= counter + '1';
                                end if;
                        end if;
                end if;
        end process;


        output <= counter(6 downto 2);
        bit0 <= counter(1);
        overflow <= ovf;

end Behavioral;
```

2:1 MUX 8 bus

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity mux2to1_8bus is
   Port ( s : in  STD_LOGIC;
          i0 : in  STD_LOGIC_VECTOR (7 downto 0);
          i1 : in  STD_LOGIC_VECTOR (7 downto 0);
          o : out  STD_LOGIC_VECTOR (7 downto 0));
end mux2to1_8bus;

architecture Behavioral of mux2to1_8bus is

begin
        process(s, i0, i1)
        begin
                if(s = '1') then
                        o <= i1;
                else
                        o <= i0;
```

```
        end if;
    end process;


end Behavioral;
```

2:1 MUX 5 bus

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity mux2to1 is
   Port ( s : in  STD_LOGIC;
          i0 : in  STD_LOGIC_VECTOR (4 downto 0);
          i1 : in  STD_LOGIC_VECTOR (4 downto 0);
          o : out  STD_LOGIC_VECTOR (4 downto 0));
end mux2to1;

architecture Behavioral of mux2to1 is

begin

    process(s, i0, i1)
    begin
        if(s = '1') then
            o <= i1;
        else
            o <= i0;
        end if;
    end process;


end Behavioral;
```

4:1 MUX

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity muxforSRAMwen is
   Port ( s : in  STD_LOGIC_VECTOR (1 downto 0);
          i1 : in  STD_LOGIC;
                        i2 : in  STD_LOGIC;
          o : out  STD_LOGIC);
end muxforSRAMwen;

architecture Behavioral of muxforSRAMwen is

begin
    process(s, i1, i2)
    begin
        if(s = "00") then
            o <= '0';
        elsif(s = "01") then
            o <= '1';
        elsif(s = "10") then
            o <= not(i1 or i2);
        end if;
    end process;


end Behavioral;
```

SRAM

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
use ieee.numeric_std.all;

entity sram is
   Port ( clk : in  STD_LOGIC;
          wr : in  STD_LOGIC;
          din : in  STD_LOGIC_VECTOR (7 downto 0);
          dout : out  STD_LOGIC_VECTOR (7 downto 0);
          add : in  STD_LOGIC_VECTOR (7 downto 0));
end sram;

architecture Behavioral of sram is
		type memory is array (0 to 255) of std_logic_vector(7 downto 0);
		signal my_memory : memory := (others => (others => '0'));
begin
		process(clk)
		begin
			if(clk'Event and clk = '1') then
				if(wr = '1') then
					my_memory(to_integer(unsigned(add))) <= din;
				else
					dout <= my_memory(to_integer(unsigned(add)));
				end if;
			end if;
		end process;


end Behavioral;
```

1:2 DEMUX 8 bus

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity demu1to2_8bus is
   Port ( i : in  STD_LOGIC_VECTOR (7 downto 0);
          o0 : out  STD_LOGIC_VECTOR (7 downto 0);
          o1 : out  STD_LOGIC_VECTOR (7 downto 0);
          s : in  STD_LOGIC);
end demu1to2_8bus;

architecture Behavioral of demu1to2_8bus is

begin
		process(s, i)
		begin
			if(s = '1') then
				o1 <= i;
			else
				o0 <= i;
			end if;
		end process;


end Behavioral;
```

SDRAM

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.numeric_std.all;


entity sdram is
   Port ( add : in  STD_LOGIC_VECTOR (15 downto 0);
          wr : in  STD_LOGIC;
          mstrb : in  STD_LOGIC;
          din : in  STD_LOGIC_VECTOR (7 downto 0);
          dout : out  STD_LOGIC_VECTOR (7 downto 0));
end sdram;

architecture Behavioral of sdram is
```

```
		type memory is array (0 to 128) of std_logic_vector(7 downto 0);
		signal mem : memory := (others => (others => '0'));

begin
		process(add, wr)
		begin
				if(wr = '0') then
						case add(15 downto 5) is
								when "00010001000" => dout <= mem(to_integer(unsigned(add(4 downto 0))));
								when "00110011010" => dout <= mem(to_integer(unsigned(add(4 downto 0)+"100000")));
								when "01000100010" => dout <= mem(to_integer(unsigned(add(4 downto 0)+"1000000")));
								when "01100110000" => dout <= mem(to_integer(unsigned(add(4 downto 0)+"1100000")));
								when others => dout <= (others => '0');
						end case;
				end if;
		end process;

		process(mstrb)
		begin
				if(mstrb'Event and mstrb = '1' and wr = '1') then
						case add(15 downto 5) is
								when "00010001000" => mem(to_integer(unsigned(add(4 downto 0)))) <= din;
								when "00110011010" => mem(to_integer(unsigned(add(4 downto 0)+"100000"))) <= din;
								when "01000100010" => mem(to_integer(unsigned(add(4 downto 0)+"1000000"))) <= din;
								when "01100110000" => mem(to_integer(unsigned(add(4 downto 0)+"1100000"))) <= din;
								when others => mem(128) <= (others => '0');
						end case;
				end if;
		end process;

end Behavioral;
```