Group 11, L02
Anish Paramsothy, paramsa, 400300097
Maryem Abdullatif, abdulm41, 400297192

Lab 4: SDRAM Controller Interface

MECHTRON 3TB4 POSTLAB, McMaster University

Date Submitted: March 21st, 2025

Due Date: March 24th, 2025

Contents

1 Introduction

Lab 4 Introduced us to the concept of software/hardware codesign, as we were tasked to create an SOPC system, use a SDRAM chip and analyze the output of our C code through a Signal Tap Logic Analyzer.

2 Part 1 - Pre-lab

2.1 Module Creation

After recovering the `lab4.qar`, we needed to update the modules so they could work alongside the system and code that was created.

We needed to first update the `SDRAM_Controller.v` module to include correct output assignments.

```
/*****************************************************************************
*                                                                            *
* Module:       SDRAM_Controller                                              *
* Description:                                                               *
*      This module is used for the sram controller for MT3TB4 Lab 4           *
*                                                                            *
*****************************************************************************/
`timescale 1ns / 1ps

module SDRAM_Controller (  //the ports sequence follows the sequence of wires in the diagram in the lab manual
        //signals through Avalon Interface:
        input       clock,
        input       reset_n,
        input       chipselect,
        input       write_n,
        input       read_n,
        input  [1:0]  byteenable_n,
        input  [24:0] address,   //2's power of 25 is  33,554,432 that is 32M.  (16 bits word, that makes capable for 64MBytes)
        input  [15:0] write_data,
        output [15:0] read_data,
        output      wait_request,
        output      data_validation,

        //signals between Conroller and SDRAM:

        inout  [15:0] DRAM_DQ,
        output [12:0] DRAM_ADDR,
        output [1:0]  DRAM_BA,
    //  output      DRAM_CLK,
        output      DRAM_CKE,
        output      DRAM_LDQM,
        output      DRAM_UDQM,
        output      DRAM_WE_N,
        output      DRAM_CAS_N,
        output      DRAM_RAS_N,
        output      DRAM_CS_N

);

        wire        clock_wire/*synthesis keep*/;
        wire        reset_n_wire/*synthesis keep*/;
        wire        chipselect_wire/*synthesis keep*/;
        wire        write_n_wire/*synthesis keep*/;
        wire        read_n_wire/*synthesis keep*/;
        wire  [1:0]   byteenable_n_wire/*synthesis keep*/;
```

```
        wire  [24:0]  address_wire/*synthesis keep*/;   //2's power of 25 is 33,554,432 that is 32M.  (16 bits word, that makes capable for 64MBytes)
        wire  [15:0]  write_data_wire/*synthesis keep*/;
        wire        Wait_request_wire/*synthesis keep*/;
        wire        data_validation_wire/*synthesis keep*/;


        reg [15:0]  read_data_reg/*synthesis preserve*/;
        //wire  [15:0]  read_data_wire/*synthesis keep*/;

        //signals between Conroller and SDRAM:

        wire  [15:0]  DRAM_DQ_wire/*synthesis keep*/;
        wire  [12:0]  DRAM_ADDR_wire/*synthesis keep*/;
        wire  [1:0]   DRAM_BA_wire/*synthesis keep*/;

        wire        DRAM_CKE_wire/*synthesis keep*/;

        wire  [1:0]   DRAM_DQM_wire/*synthesis keep*/;    //higher bit for UDQM , lower bit for LDQM
        wire        DRAM_WE_N_wire/*synthesis keep*/;
        wire        DRAM_CAS_N_wire/*synthesis keep*/;
        wire        DRAM_RAS_N_wire/*synthesis keep*/;
        wire        DRAM_CS_N_wire/*synthesis keep*/;

        wire [15:0]   m_data_wire;
        wire        output_enable_wire;

        DE1_SoC_QSYS_sdram  my_sdram (
                            // inputs:
                            .az_addr(address_wire),
                            .az_be_n(byteenable_n_wire),
                            .az_cs(chipselect_wire),
                            .az_data(write_data_wire),
                            .az_rd_n(read_n_wire),
                            .az_wr_n(write_n_wire),
                            .clk(clock_wire),
                            .reset_n(reset_n_wire),

                            // outputs:
                            //.za_data(read_data_wire),    //can not get read_data from here.
                            .za_valid(data_validation_wire),
                            .za_waitrequest(Wait_request_wire),
                            .zs_addr(DRAM_ADDR_wire),
                            .zs_ba(DRAM_BA_wire),
                            .zs_cas_n(DRAM_CAS_N_wire),
                            .zs_cke(DRAM_CKE_wire),
                            .zs_cs_n(DRAM_CS_N_wire),
                            //.zs_dq(DRAM_DQ_wire),
                            .zs_dqm(DRAM_DQM_wire),
                            .zs_ras_n(DRAM_RAS_N_wire),
                            .zs_we_n(DRAM_WE_N_wire),
                                        .output_enable(output_enable_wire),
                                        .internal_m_data(m_data_wire)
                            );

    //=============Make connections =====================

        assign      DRAM_DQ=output_enable_wire?m_data_wire:{16{1'bz}} ;

        //this way works  , get data from za_data port does not works.
        always @(posedge clock or negedge reset_n)   //must be synchronize with the clock!
        begin
            if (reset_n == 0)
```

```
                read_data_reg <= 0;
            else
                read_data_reg <= DRAM_DQ;
        end

        assign      clock_wire=clock;
        assign      reset_n_wire=reset_n;
        assign      chipselect_wire=chipselect;
        assign      write_n_wire=write_n;
        assign      read_n_wire=read_n;
        assign      byteenable_n_wire=byteenable_n;
        assign      address_wire=address;   //2's power of 25 is  33,554,432 that is 32M.  (16 bits word, that makes capable for 64MBytes)
        assign      write_data_wire=write_data;


        //output signals

        assign      read_data=read_data_reg;
        //assign      read_data=read_data_wire;   //can not get read_data from the za_data port

        assign      wait_request=Wait_request_wire;
        assign      data_validation=data_validation_wire;

        //signals between Conroller and SDRAM:

        assign      DRAM_UDQM=DRAM_DQM_wire[1];     //higher bit for UDQM , lower bit for LDQM
        assign      DRAM_LDQM=DRAM_DQM_wire[0];

    //  ==========Make  more  necessary connections================

        assign      DRAM_CKE=DRAM_CKE_wire;
        assign      DRAM_ADDR=DRAM_ADDR_wire;
        assign      DRAM_BA=DRAM_BA_wire;
        assign      DRAM_WE_N=DRAM_WE_N_wire;
        assign      DRAM_CAS_N=DRAM_CAS_N_wire;
        assign      DRAM_RAS_N=DRAM_RAS_N_wire;
        assign      DRAM_CS_N=DRAM_CS_N_wire;

endmodule
```

The SDRAM Circuit is also shown:

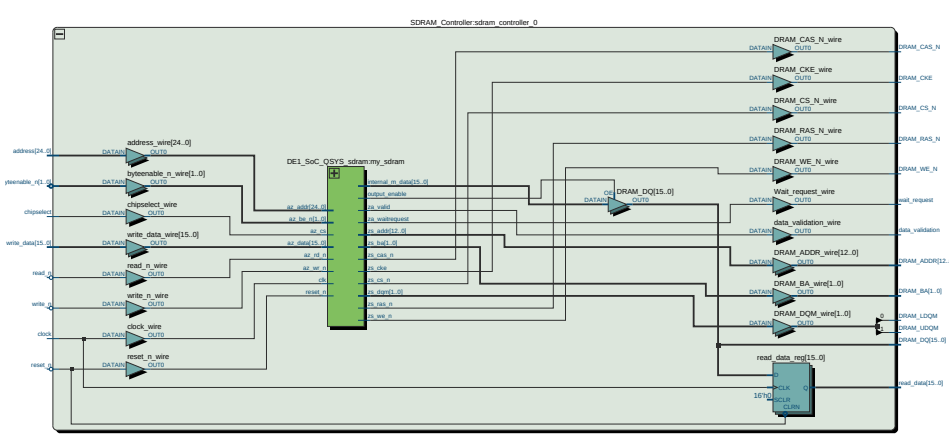
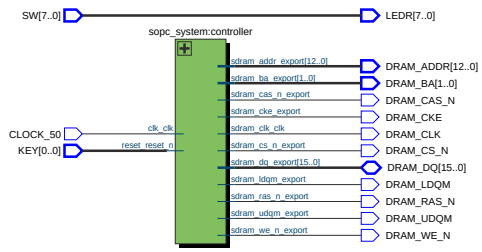


Figure 1: SDRAM Controller Circuit

Then we needed to instantiate the SOPC system in the top-level module.

```
/*****************************************************************************
*                                                                            *
* Module:       Lab4                                                        *
* Description:                                                               *
*      This module is the top level module of MT3TB4 Lab 4                   *
*                                                                            *
*****************************************************************************/

module lab4 (
input       CLOCK_50,
input    [0:0] KEY,
input    [7:0] SW,                //for reset
output   [7:0] LEDR,

// Bidirectionals
inout    [15:0]    DRAM_DQ,

// Outputs

output     [12:0]  DRAM_ADDR,
output     [1:0]   DRAM_BA,
output           DRAM_LDQM,  //data mask; when it is low, the DQ is valid for reading and writing.
output           DRAM_UDQM,
output           DRAM_RAS_N,
output           DRAM_CAS_N,
output           DRAM_CLK,
output           DRAM_CKE,
output           DRAM_WE_N,
output           DRAM_CS_N

);

// Internal Wires

assign LEDR=SW;

//Instantiate your sopc_system module generated by Platform Designer.


sopc_system  controller (
        // example ports
        .clk_clk(CLOCK_50),              //            clk.clk
        .reset_reset_n(KEY[0]),         //          reset.reset_n
        .sdram_addr_export(DRAM_ADDR),
        .sdram_ba_export(DRAM_BA),
        .sdram_cas_n_export(DRAM_CAS_N),
        .sdram_cke_export(DRAM_CKE),
        .sdram_clk_clk(DRAM_CLK),
        .sdram_cs_n_export(DRAM_CS_N),
        .sdram_dq_export(DRAM_DQ),
        .sdram_ldqm_export(DRAM_LDQM),
        .sdram_ras_n_export(DRAM_RAS_N),
        .sdram_udqm_export(DRAM_UDQM),
        .sdram_we_n_export(DRAM_WE_N)
        // more ports

    );

endmodule
```

The Top level SOPC circuit can be seen below:



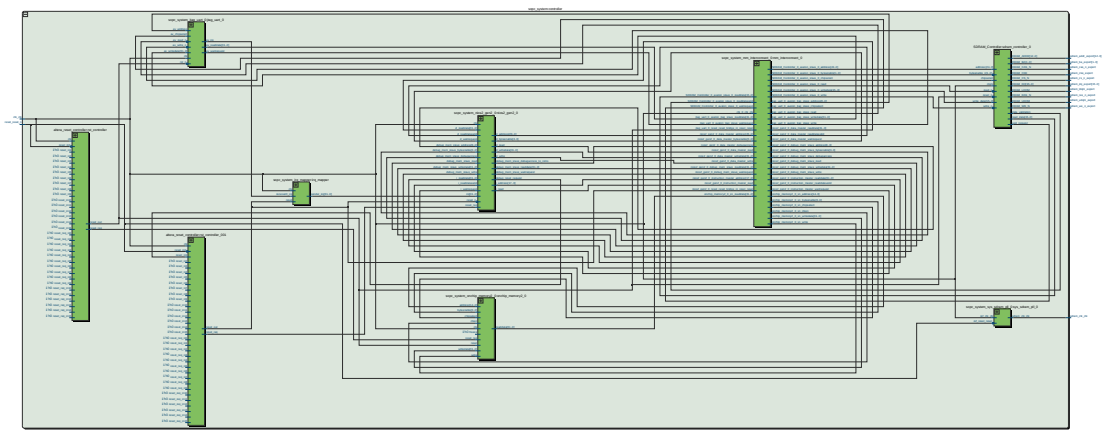


Figure 2: SOPC RTL

After completing the modules, we compiled the code to ensure there were no errors.

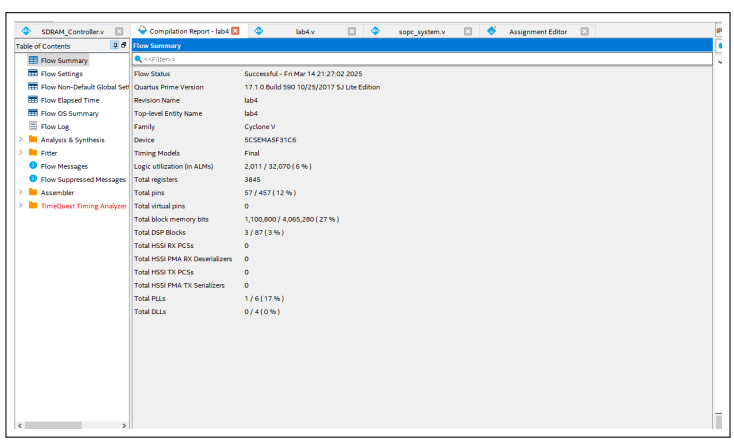


Figure 3: Compilation Report

The final portion was developing the C code for writing and reading `char`, `short`, and `int`, ensuring that the value written is the same as the one that is read.

```
#include "system.h"
#include <stdio.h>

```

```c
#define BASE SDRAM_CONTROLLER_0_BASE

#define MAXNUM_WORDS   SDRAM_CONTROLLER_0_SPAN/2
//For the SDRAM on DE1-SoC, the size is 64MB=67108864 Byte (SDRAM_SPAN)

int main()
{
    printf("Hello from MT3TB4 Group 11!\n");

    int i;
    int char_err_num=0, short_err_num=0, int_err_num=0;

    char * char_ptr;

    char aChar;

    short *short_ptr;
    short aShort;

    int *int_ptr;
    int aInt;

    int charsize, shortsize, intsize;

    charsize=sizeof(aChar);
    shortsize=sizeof(aShort);
    intsize=sizeof(aInt);
    printf("the sizeof char, short, int are: %d, %d, %d\n", charsize, shortsize, intsize);

        //----------------TEST CHAR-----------------------------------

        printf("\n writing chars.....\n");
        for (i=0; i<MAXNUM_WORDS*2; i++) {
                *(char*)(BASE+i)=i%128; // to be safe, use 128 rather than 256
        }

        printf("\n testing chars.....\n");
        for (i=0; i<MAXNUM_WORDS*2; i++) {
            if (* (char*)(BASE+i)!=i%128){  // or .....(char)i,   if not i%128
                    char_err_num++;
                }

        }
            printf("Testing  Char: the total numbers of error is : %i\n" ,char_err_num);




        //----------------TEST SHORT----------------------------------

            printf(" \n writing short......\n");
            for (i=0; i<MAXNUM_WORDS; i++) {
            //for (i=0; i<32767; i++) {
                *(short*)(BASE+i*2)=i%32767; // short, uses two bytes

            }

            printf(" \n testing short......\n");
            for (i=0; i<MAXNUM_WORDS; i++) {
            //for (i=0; i<32767; i++) {
                if(*(short*)(BASE+i*2)!=i%32767) {
                    short_err_num++;
                } // short, uses two bytes

```

```c
            }



            printf(" \n testing short......\n");
    //
    //
    //
            printf("Testing Short: the total numbers of error is : %i\n" , short_err_num);

        //----------------TEST INT    ----------------------------------
            printf(" \n writing integer......\n");

            for (i=0; i<MAXNUM_WORDS/2; i++) {
                *(int*)(BASE+i*4)=i; // int, use 4 bytes

            }

            printf(" \n testing integer......\n");
            for (i=0; i<MAXNUM_WORDS/2; i++) {
                if(*(int*)(BASE+i*4)!=i) {
                    int_err_num++;
                } // int, use 4 bytes

            }


            printf("Testing Integer: the total numbers of error is : %i\n" , int_err_num);
            //
    //
    //
            printf("Testing Completed.\n");
    return 0;
}
```

The final output of the C code is shown below:

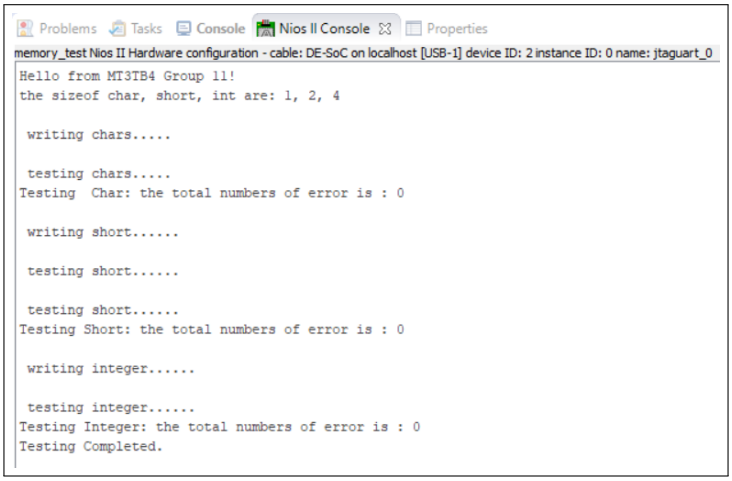


Figure 4: Software Output in NIOS II

The code is able to succesfully write `chars`, `shorts`, and `ints` into memory, and correctly read them back in the same order, which signifies that there is no error in the writing or reading process.

We have also provided a few screenshots of the Signal Tap Analyzer window during each of the processes.

Figure 5: Testing `char`

Figure 6: Writing `short`

Figure 7: Writing `int`

3 Conclusion

3.1 Post-lab Questions

Q1: Using screen shots taken in the "Develop Software" part of the lab explain the following:

a) Why does it appear that writing takes less clock cycles than reading?

The reason that writing takes less clock cycles than reading is that it requires less steps to complete the process. For writing, data is already fetched and the operation just includes placing it into a cache/register. For reading, the data needs to be fetched/retrieved from a specific location, meaning it requires this extra layer of access and can thus take more clock cycles.

b) While reading or writing a char value, e.g. 5, why does the SDRAM_DQ show values such as 0x0505 instead of 0x0005?

We know that a char is represented in 8 bits, while the hexadecimal 0x0000 represents 16 bits, since each digit that makes up the hexadecimal represents 4 bits. Since the 8-bit char occupies half of the space in the hexadecimal number, it means that two chars can fit into the 16-bit word. Since there is

a possibility for two chars to exist in the same word, when a 8-bit char is fed in the memory controller, it may replicate the value in both bytes of the 16-bit word. This wil lead to 0x0505.

c) While reading or writing two consecutive char values, e.g. 5 and 6 using memory locations 0x0000 and 0x0001, why does the SDRAM_ADDR show the same 0x0000 value for both locations?

The reason the same 0x0000 value is shown for both locations (0x0000 an 0x0001) is because a word address is assumed to be represented in two bytes, meaning that the first two bytes would share the same word address.

Q2: What steps does the Avalon Interconnect take to write a 32-bit integer into the 16-bit SDRAM memory?

The first step is to map the memory location in SDRAM by retrieving an address to write to. It will then split the integer into 2, writing the MSB into memory, shifting the SDRAM address to the next one, and then writing the other 16 bits into memory.

As mentioend in the *Avalon Switch Fabric* document, a wider master results in multiple slave-read transfers to sequential addresses in the slaves address space (essentially splitting the master into small parts to be accessible by the slave).

Q3: Open the compilation report in Quartus, and report the following numbers:

- **Total number of logic elements used by your circuit:** 2,011/32,070
- **Total number of memory bits used by your circuit:** 1,100,800/4,065,280
- **Total number of pins:** 57/457
- **The maximum number of logic elements that can fit on the FPGA used:** 32,070

Q4: Considering just the maximum number of logic elements on the FPGA, approximately how many SOPC systems like the one you built could fit on the FPGA?

$$\frac{32,070}{2,011} = 15.94 \approx 15$$

You would be able to fit approximately 15 SOPC systems if we base this on logic elements used.

Q5: Considering just the maximum amount of memory available in the FPGA, approximately how many SOPC systems like the one you built could fit on the FPGA?

$$\frac{4,065,280}{1,100,800} = 3.69 \approx 3$$

You would be able to fit approximately 3 SOPC systems if we are looking at memory available.

Q6: Considering just the maximum number of pins on the FPGA, how many SOPC systems like the one you built could fit on the FPGA?

$$\frac{457}{57} = 8.02 \approx 8$$

You would be able to fit approximately 8 SOPC systems if we are focused on pin count.

3.2 Wrap-up

Overall the Lab was an interesting introduction to the concept of software/hardware codesign. Since this concept is very important, I would hope that this would be covered in greater detail prior to completing the lab to help with understanding more.