The City College of New York

Computer Organization Lab

CSC 34300

(Spring 2024)

LAB 3

Integration ADD/SUM LPM module with SRAM LPM module.

Mahir Asef

Instructor: Albi Arapi

Date: 04.03.2024

Table of Contents

Objective:

The purpose of this lab experiment is to introduce ourselves to working with LPM modules in Quartus. By the end of this experiment, we will be able to import and implement add/sub lmp as well as a sram lpm and able to see how the addsub lpm adds two numbers and how sram writes a number in memory and how memory reads it. Ultimately, our goal is to write two numbers in sram memory and store them in two different addresses from where the memory reads the numbers and feeds it into lpm module to add them together. The result of the addition will then be stored back to a specific address of sram memory.

Functionality and Specifications:

Sram LPM:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY altera_mf;
USE altera_mf.altera_mf_components.all;

ENTITY asef_sram IS
   PORT
   (
      address     : IN STD_LOGIC_VECTOR (7 DOWNTO 0);
      clock       : IN STD_LOGIC  := '1';
      data        : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
      wren        : IN STD_LOGIC ;
      q     : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
   );
END asef_sram;

ARCHITECTURE SYN OF asef_sram IS

   SIGNAL sub_wire0  : STD_LOGIC_VECTOR (31 DOWNTO 0);

BEGIN
   q    <= sub_wire0(31 DOWNTO 0);

   altsyncram_component : altsyncram
   GENERIC MAP (
      clock_enable_input_a => "BYPASS",
      clock_enable_output_a => "BYPASS",
      init_file => "init.mif",
      intended_device_family => "Cyclone V",
      lpm_hint => "ENABLE_RUNTIME_MOD=NO",
      lpm_type => "altsyncram",
      numwords_a => 256,
      operation_mode => "SINGLE_PORT",
      outdata_aclr_a => "NONE",
      outdata_reg_a => "CLOCK0",
      power_up_uninitialized => "FALSE",
      read_during_write_mode_port_a => "NEW_DATA_NO_NBE_READ",
      widthad_a => 8,
      width_a => 32,
      width_byteena_a => 1
   )
   PORT MAP (
      address_a => address,
      clock0 => clock,
      data_a => data,
      wren_a => wren,
      q_a => sub_wire0
   );


END SYN;
```

The above is VHDL code for the Sram module. In our experiment, this sram module will take two input data initiated from a MIF file with address 1 and 2 and write it on memory when wren is high with the clock at rising edge and q will read the data from memory for further operations.

32 Bit 1:2 Demux:

```
library ieee;
use ieee.std_logic_1164.all;

entity asef_1x2_demux_32bit is
    port (
        input : in  std_logic_vector(31 downto 0); -- 32-bit input
        sel   : in  std_logic;                     -- Selection signal
        output1 : out std_logic_vector(31 downto 0); -- Output 1
        output2 : out std_logic_vector(31 downto 0)  -- Output 2
    );
end asef_1x2_demux_32bit;

architecture Behavioral of asef_1x2_demux_32bit is
begin
    process (input, sel)
    begin
        if sel = '0' then
            output1 <= input;
            output2 <= (others => '0'); -- Set output 2 to all zeroes
        elsif sel = '1' then
            output1 <= (others => '0'); -- Set output 1 to all zeroes
            output2 <= input;
        else
            output1 <= (others => '0'); -- Default: Set both outputs to all zeroes
            output2 <= (others => '0');
        end if;
    end process;
end Behavioral;

```

The 1:2 demultiplexer in our design will be fed the data that's read by sram as its input signal. With the select input being 0, the data at memory address 1 will be the 1st output of the demux and data at address 2 will be the 2nd output of the demux when select input is high.

Non-Shift-Register:

```
library ieee;
use ieee.std_logic_1164.all;

entity asef_non_shift_register is
    port (
        clk   : in  std_logic;
        load  : in  std_logic;
        data  : in  std_logic_vector(31 downto 0);
        q     : out std_logic_vector(31 downto 0)
    );
end asef_non_shift_register;

architecture Behavioral of asef_non_shift_register is
    signal reg_data : std_logic_vector(31 downto 0);
begin
    process(clk)
    begin
        if rising_edge(clk) then
            if load = '1' then
                reg_data <= data;
            end if;
        end if;
    end process;

    q <= reg_data;
end Behavioral;

```

Since we will not be able to store the data sitting at output1 and output2 of demux to add sub lpm for further operation, we created this non shift register. The non shift register only stores the values and feeds the values into the add sub module when both data are ready. The reason we need to store data is because of the process in sram which will write and read the data at address 1 first so this data may have to wait until sram also writes andreads the data from memory address 2 and then feed it through demux. Once both data are ready, this register takes them to storage for further operation.

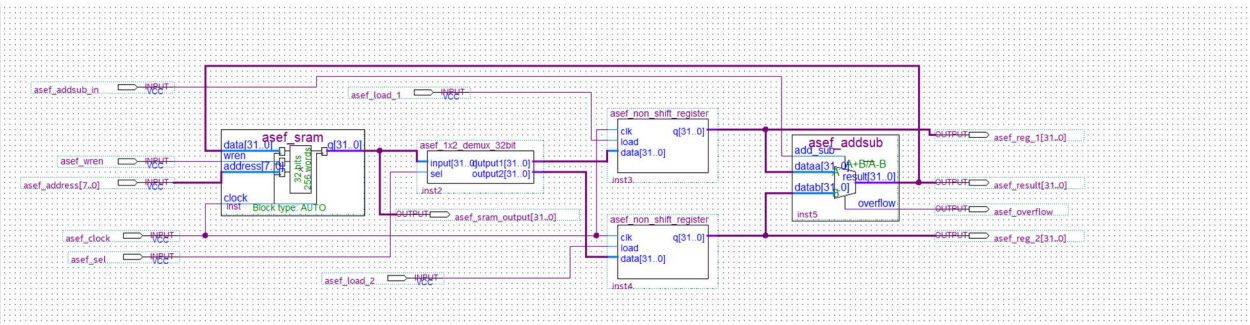
Add/Sub LPM:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY lpm;
USE lpm.all;

ENTITY asef_addsub IS
    PORT
    (
       add_sub     : IN STD_LOGIC ;
       dataa    : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
       datab    : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
       overflow    : OUT STD_LOGIC ;
       result      : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
    );
END asef_addsub;


ARCHITECTURE SYN OF asef_addsub IS

    SIGNAL sub_wire0  : STD_LOGIC ;
    SIGNAL sub_wire1  : STD_LOGIC_VECTOR (31 DOWNTO 0);


    COMPONENT lpm_add_sub
    GENERIC (
       lpm_direction     : STRING;
       lpm_hint    : STRING;
       lpm_representation      : STRING;
       lpm_type    : STRING;
       lpm_width      : NATURAL
    );
    PORT (
          add_sub  : IN STD_LOGIC ;
          dataa : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
          datab : IN STD_LOGIC_VECTOR (31 DOWNTO 0);
          overflow : OUT STD_LOGIC ;
          result   : OUT STD_LOGIC_VECTOR (31 DOWNTO 0)
    );
    END COMPONENT;

BEGIN
    overflow    <= sub_wire0;
    result    <= sub_wire1(31 DOWNTO 0);

    LPM_ADD_SUB_component : LPM_ADD_SUB
    GENERIC MAP (
       lpm_direction => "UNUSED",
       lpm_hint => "ONE_INPUT_IS_CONSTANT=NO,CIN_USED=NO",
       lpm_representation => "UNSIGNED",
       lpm_type => "LPM_ADD_SUB",
       lpm_width => 32
    )
    PORT MAP (
       add_sub => add_sub,
       dataa => dataa,
       datab => datab,
       overflow => sub_wire0,
       result => sub_wire1
    );


END SYN;
```

This is the final step of our design where w e create a add/sub module that takes the two inputs store in the register and with the operation input being high, it adds the two numbers and outputs the result of addition as well as if any overflow ocurred.

Complete Design:

Add/Sub SRAM Circuit:



Add/Sub SRAM VHDL:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

LIBRARY work;

ENTITY asef_addsub_sram IS
    PORT
    (
        asef_addsub_in :  IN  STD_LOGIC;
        asef_wren :  IN  STD_LOGIC;
        asef_clock :  IN  STD_LOGIC;
        asef_sel :  IN  STD_LOGIC;
        asef_load_2 :  IN  STD_LOGIC;
        asef_load_1 :  IN  STD_LOGIC;
        asef_address :  IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
        asef_overflow :  OUT  STD_LOGIC;
        asef_reg_1 :  OUT  STD_LOGIC_VECTOR(31 DOWNTO 0);
        asef_reg_2 :  OUT  STD_LOGIC_VECTOR(31 DOWNTO 0);
        asef_result :  OUT  STD_LOGIC_VECTOR(31 DOWNTO 0);
        asef_sram_output :  OUT  STD_LOGIC_VECTOR(31 DOWNTO 0)
    );
END asef_addsub_sram;

ARCHITECTURE bdf_type OF asef_addsub_sram IS

COMPONENT asef_sram
    PORT(wren : IN STD_LOGIC;
         clock : IN STD_LOGIC;
         address : IN STD_LOGIC_VECTOR(7 DOWNTO 0);
         data : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
         q : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
    );
END COMPONENT;

COMPONENT asef_1x2_demux_32bit
    PORT(sel : IN STD_LOGIC;
         input : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
         output1 : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
         output2 : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
    );
END COMPONENT;

COMPONENT asef_non_shift_register
    PORT(clk : IN STD_LOGIC;
         load : IN STD_LOGIC;
         data : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
         q : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
    );
END COMPONENT;

COMPONENT asef_addsub
    PORT(add_sub : IN STD_LOGIC;
         dataa : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
         datab : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
         overflow : OUT STD_LOGIC;
         result : OUT STD_LOGIC_VECTOR(31 DOWNTO 0)
    );
END COMPONENT;

SIGNAL   SYNTHESIZED_WIRE_0 :  STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL   SYNTHESIZED_WIRE_1 :  STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL   SYNTHESIZED_WIRE_2 :  STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL   SYNTHESIZED_WIRE_3 :  STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL   SYNTHESIZED_WIRE_4 :  STD_LOGIC_VECTOR(31 DOWNTO 0);
SIGNAL   SYNTHESIZED_WIRE_5 :  STD_LOGIC_VECTOR(31 DOWNTO 0);


BEGIN
asef_reg_1 <= SYNTHESIZED_WIRE_4;
asef_reg_2 <= SYNTHESIZED_WIRE_5;
asef_result <= SYNTHESIZED_WIRE_0;
asef_sram_output <= SYNTHESIZED_WIRE_1;


b2v_inst : asef_sram
PORT MAP(wren => asef_wren,
         clock => asef_clock,
         address => asef_address,
         data => SYNTHESIZED_WIRE_0,
         q => SYNTHESIZED_WIRE_1);


b2v_inst2 : asef_1x2_demux_32bit
PORT MAP(sel => asef_sel,
         input => SYNTHESIZED_WIRE_1,
         output1 => SYNTHESIZED_WIRE_2,
         output2 => SYNTHESIZED_WIRE_3);


b2v_inst3 : asef_non_shift_register
PORT MAP(clk => asef_clock,
         load => asef_load_1,
         data => SYNTHESIZED_WIRE_2,
         q => SYNTHESIZED_WIRE_4);


b2v_inst4 : asef_non_shift_register
PORT MAP(clk => asef_clock,
         load => asef_load_2,
         data => SYNTHESIZED_WIRE_3,
         q => SYNTHESIZED_WIRE_5);


b2v_inst5 : asef_addsub
PORT MAP(add_sub => asef_addsub_in,
         dataa => SYNTHESIZED_WIRE_4,
         datab => SYNTHESIZED_WIRE_5,
         overflow => asef_overflow,
         result => SYNTHESIZED_WIRE_0);

END bdf_type;
```

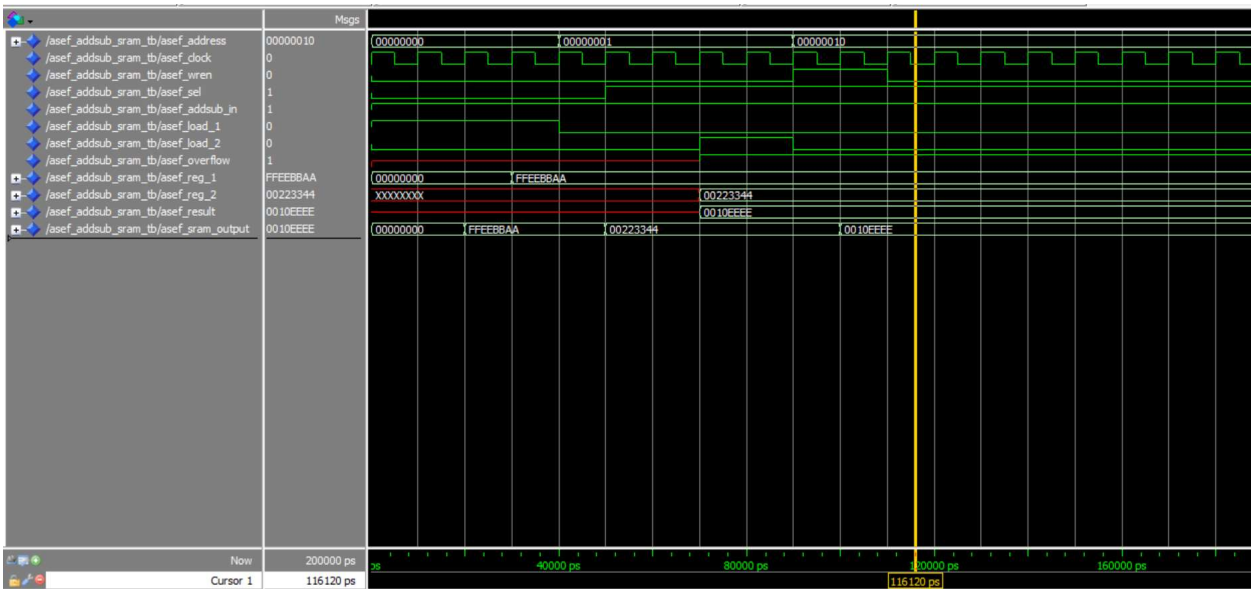
Testbench Code:

```
library ieee;
use ieee.std_logic_1164.all;

entity asef_addsub_sram_tb is
end entity asef_addsub_sram_tb;



architecture TbArchitecture of asef_addsub_sram_tb is
   signal asef_address: std_logic_vector(7 downto 0);
   signal asef_clock :std_logic := '0';
   signal asef_wren, asef_sel, asef_addsub_in, asef_load_1, asef_load_2, asef_overflow : std_logic;
   signal asef_reg_1, asef_reg_2, asef_result, asef_sram_output: std_logic_vector(31 downto 0);

   component asef_addsub_sram is
         PORT
   (
      asef_addsub_in :  IN  STD_LOGIC;
      asef_wren :  IN  STD_LOGIC;
      asef_clock :  IN  STD_LOGIC;
      asef_sel :  IN  STD_LOGIC;
      asef_load_2 :  IN  STD_LOGIC;
      asef_load_1 :  IN  STD_LOGIC;
      asef_address :  IN  STD_LOGIC_VECTOR(7 DOWNTO 0);
      asef_overflow :  OUT  STD_LOGIC;
      asef_reg_1 :  OUT  STD_LOGIC_VECTOR(31 DOWNTO 0);
      asef_reg_2 :  OUT  STD_LOGIC_VECTOR(31 DOWNTO 0);
      asef_result :  OUT  STD_LOGIC_VECTOR(31 DOWNTO 0);
      asef_sram_output :  OUT  STD_LOGIC_VECTOR(31 DOWNTO 0)
   );
   end component  asef_addsub_sram;


   begin
      DUT: asef_addsub_sram port map(
         asef_wren => asef_wren,
         asef_addsub_in => asef_addsub_in,
         asef_sel => asef_sel,
         asef_clock => asef_clock,
         asef_load_1 => asef_load_1,
         asef_load_2 => asef_load_2,
         asef_address => asef_address,
         asef_sram_output => asef_sram_output,
         asef_overflow => asef_overflow,
         asef_reg_1 => asef_reg_1,
         asef_reg_2 => asef_reg_2,
         asef_result => asef_result
      );


      -- clk
PROCESS
    BEGIN
      asef_clock <= NOT asef_clock;
      WAIT FOR 5 ns;
    END PROCESS;

  stimulus: process
        begin
        asef_wren <= '0';
        asef_addsub_in <= '1';

        asef_sel <= '0';
        asef_load_1 <= '1';
        asef_load_2 <= '0';
        asef_address <= "00000000";
        wait for 40ns;
        asef_address <= "00000001";


        asef_load_1 <= '0';
        wait for 10ns;
        asef_sel <= '1';
        wait for 20ns;
        -- reg_load_in_1 <= '0';
        asef_load_2 <= '1';


        wait for 20ns;


        asef_load_2 <= '0';
        asef_load_1 <= '0';
        asef_address <= "00000010";
        asef_wren <= '1';
        wait for 20ns;
        asef_wren <= '0';

        wait;
        end process;
end architecture TbArchitecture;
```

The above is VHDL testbench code that tests our full circuit design. Here, we initialze a mif file with two numbers. 1st number 0xFFEEBBAA is stored at address 00000000 and 2nd number 0x00223344 is stored at address 00000001. The sram module writes the 1st number in memory and allows the output to read the number from memory and thus 0xFFEEBBAA goes as input to 32 bit demux and when select is low, demux outputs this number on output 1 from where the number gets stored in the non shift register. Then sram writes and reads the number 0x00223344 that gets fed into the input of demux where the demux sends the number at output 2 when the select input is high. Then the number gets stored in non shift register and when both numbers r ready in the register, they are put through add/sub lpm for addition with the operand input being high for addition. Once the

addition is done the result output shows 0x0010EEEE with overflow signal being high which means the addition causes an overflow which is the correct result. At the final step, the result gets stored back to sram at memory address 3 (0000010).

Simulation:



This is the modelsim simulation of our circuit design. As we can see the 1st data gets ready first from address 00000000 and then it waits for 2nd data to come through from 00000001. When both data are available, two numbers get added and the resulting number is a correct addition that is stored at address 00000010 which is the 3rd address in the sram memory.

Conclusion:

We have reached the end of this laboratory experiment. Throughout this experimen, we were able to correctly define the functionality of add/sub module and sram odule as well as understanding how they store data or operates on data. Then we have started building our circuit that will take two inputs from sram moudle and perfom addition on the add/sub module and store the result of operation back to sram. As shown above, we were able successfully create a sram that feeds data into a 32 bit demux that we have built and 32 bit demux sends the data to a register for storage which we have also programmed. At the end we were successfully able to connect the add/sub moudle with rest of our design and as we can see, we were successfully able to perform addition between two numbers and then also found out how to store the result back to the memory of sram.

We have tehn moved onto modelsim to simulate our design for verification. Our modelsim compiled all the vhdl files with no errors and successfully created waveforms from which we were able to see that our circuit indeed performs as intended by writing and reading data from memory and then perform addtion to store the result back on memory. Overall, this has been a very successfully laboratory work that helped us learn a great deal about the built in libraray of Quartus and how they can be helpful in our won designs.