Assignment 5 - Phase I

Name - Zubair Nadaph
Ankit Taskar
Group no. - 4

Due Date - 11/16/2017

Objective

This lab is based on developing a multiplier code in VHDL which multiplies two 4-bit numbers and produces an eight bit output. The code was then burnt onto Cyclone IV FPGA DE2-115 Development kit. This project requires basic understanding of VHDL. This lab allows an individual to understand the basic arithmetic addition of ALU. It also allows to understand the parallel and serial shift registers. So, the basic objective of the lab was to generate an 8-bit output using ALU, shift registers and accumulator[1]. The keys and switches for the FPGA need to be debounced.

List of Components Used

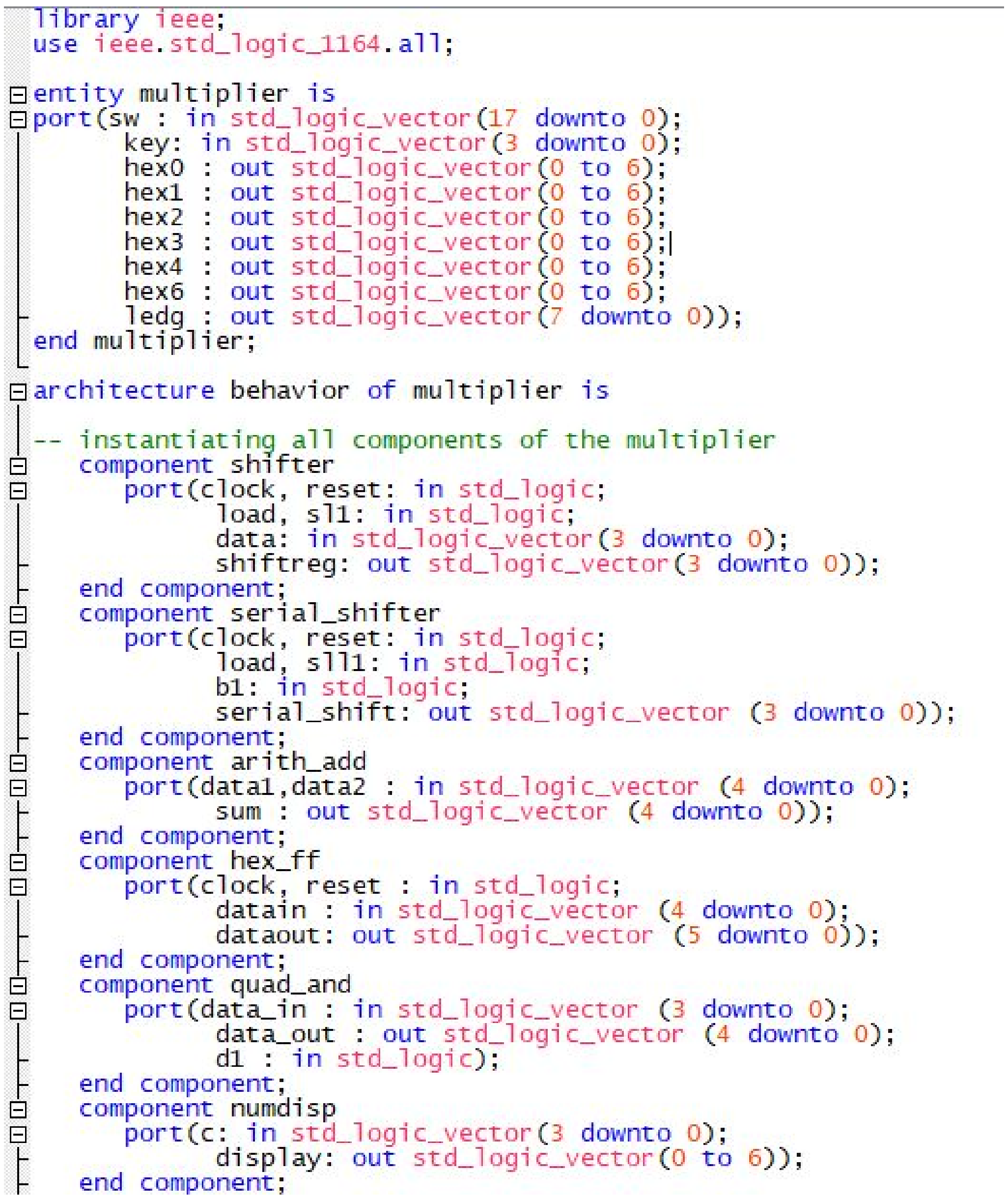
- DE2 115 Development Board
- Quartus II CAD Tool
- DE2 115 Parallel Cable for FPGA
- Board Power Supply
- DE2 115 user manual
- Tutorials

Experimental Approach

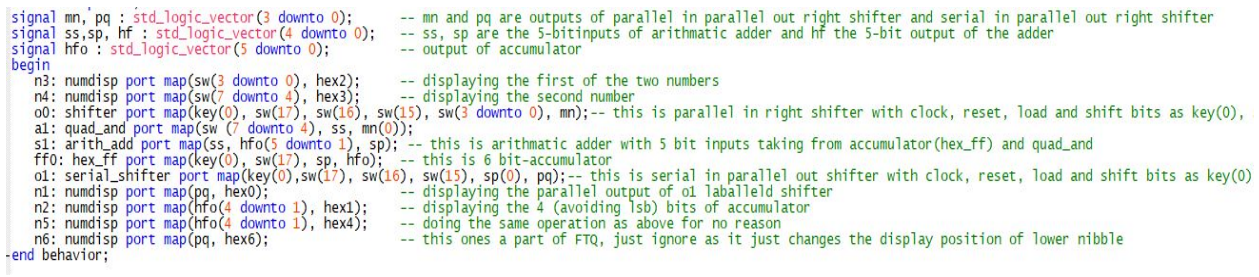
The main task of this lab to create an 8-bit multiplier using VHDl and display it on the development kit. Code for individual components were created and linked together using a portmap function. The code is given below for reference. After the code was burned onto the kit, the input was given using the switch. The inputs were displayed on HEX0 and HEX1, while the outputs were displayed on HEX7 and HEX 6. The components were designed exactly the way the lab3 was done i.e.multiplication of two numbers of higher order is done by multiplying lsb of one of the operand with the other operand, successively adding the results, collecting the lsb of successive addition and then right shift the to update the result[1]. For this, the code has a component “4 bit universal right shift register”[1]. The LSB from this shift register is right shifted and the respective bit is given to component of 4 input 5 output AND gates; the other inputs to the AND gates is the other 4-bit input and the output is given to a Arithmetic adder component, which is although quiet similar to ALU but differ only in the sense that it does not need to have mode input[1]. Also, this adder component adds two 5 bit numbers and gives 5 bit output (to allow record the carry). The output of this adder was thus given as an input to the accumulator component (to have addition in synchronism with clock)[1]. The output of the accumulator is provided back to the ALU[1]. This output acts as the second set of input[1]. The LSB of the ALU is given to serial 8-bit right shift register component[1]. In this way with every clock, the LSB of successive addition is stored in the 8-bit shift register (the previously stored number is right shifted)[1]. The upper nibble of the final result comprises of carry and first 3 outputs of ALU respectively, whereas for the lower nibble the 4 bits are taken from 8 bit serial shift (parallel out) register in reverse order And these outputs are provided to 7-segment display through BCD. The clock is provided using the Key(0). This was implemented using VHDL Code.

VHDL Code

-- Code for Combining all the components of multiplier



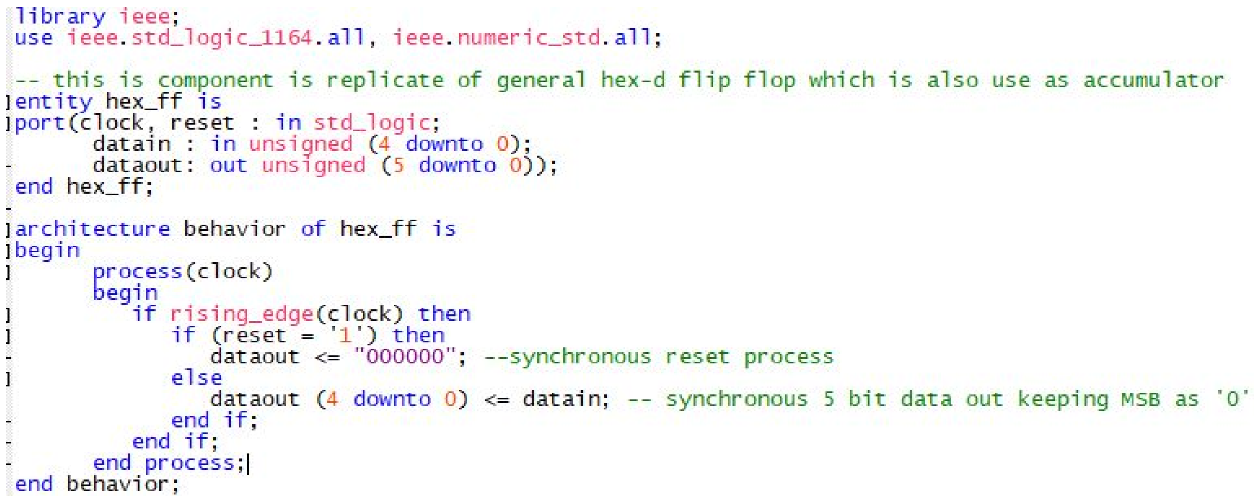
```
library ieee;
use ieee.std_logic_1164.all;

entity multiplier is
port(sw : in std_logic_vector(17 downto 0);
      key: in std_logic_vector(3 downto 0);
      hex0 : out std_logic_vector(0 to 6);
      hex1 : out std_logic_vector(0 to 6);
      hex2 : out std_logic_vector(0 to 6);
      hex3 : out std_logic_vector(0 to 6);
      hex4 : out std_logic_vector(0 to 6);
      hex6 : out std_logic_vector(0 to 6);
      ledg : out std_logic_vector(7 downto 0));
end multiplier;

architecture behavior of multiplier is

-- instantiating all components of the multiplier
   component shifter
      port(clock, reset: in std_logic;
            load, sl1: in std_logic;
            data: in std_logic_vector(3 downto 0);
            shiftreg: out std_logic_vector(3 downto 0));
   end component;
   component serial_shifter
      port(clock, reset: in std_logic;
            load, sl1l: in std_logic;
            b1: in std_logic;
            serial_shift: out std_logic_vector (3 downto 0));
   end component;
   component arith_add
      port(data1,data2 : in std_logic_vector (4 downto 0);
            sum : out std_logic_vector (4 downto 0));
   end component;
   component hex_ff
      port(clock, reset : in std_logic;
            datain : in std_logic_vector (4 downto 0);
            dataout: out std_logic_vector (5 downto 0));
   end component;
   component quad_and
      port(data_in : in std_logic_vector (3 downto 0);
            data_out : out std_logic_vector (4 downto 0);
            d1 : in std_logic);
   end component;
   component numdisp
      port(c: in std_logic_vector(3 downto 0);
            display: out std_logic_vector(0 to 6));
   end component;
```



```vhdl
signal mn, pq : std_logic_vector(3 downto 0);       -- mn and pq are outputs of parallel in parallel out right shifter and serial in parallel out right shifter
signal ss,sp, hf : std_logic_vector(4 downto 0);    -- ss, sp are the 5-bitinputs of arithmatic adder and hf the 5-bit output of the adder
signal hfo : std_logic_vector(5 downto 0);          -- output of accumulator
begin
   n3: numdisp port map(sw(3 downto 0), hex2);      -- displaying the first of the two numbers
   n4: numdisp port map(sw(7 downto 4), hex3);      -- displaying the second number
   o0: shifter port map(key(0), sw(17), sw(16), sw(15), sw(3 downto 0), mn);-- this is parallel in right shifter with clock, reset, load and shift bits as key(0),
   a1: quad_and port map(sw (7 downto 4), ss, mn(0));
   s1: arith_add port map(ss, hfo(5 downto 1), sp); -- this is arithmatic adder with 5 bit inputs taking from accumulator(hex_ff) and quad_and
   ff0: hex_ff port map(key(0), sw(17), sp, hfo);  -- this is 6 bit-accumulator
   o1: serial_shifter port map(key(0),sw(17), sw(16), sw(15), sp(0), pq);-- this is serial in parallel out shifter with clock, reset, load and shift bits as key(0)
   n1: numdisp port map(pq, hex0);                  -- displaying the parallel output of o1 laballeld shifter
   n2: numdisp port map(hfo(4 downto 1), hex1);     -- displaying the 4 (avoiding lsb) bits of accumulator
   n5: numdisp port map(hfo(4 downto 1), hex4);     -- doing the same operation as above for no reason
   n6: numdisp port map(pq, hex6);                  -- this ones a part of FTQ, just ignore as it just changes the display position of lower nibble
end behavior;
```

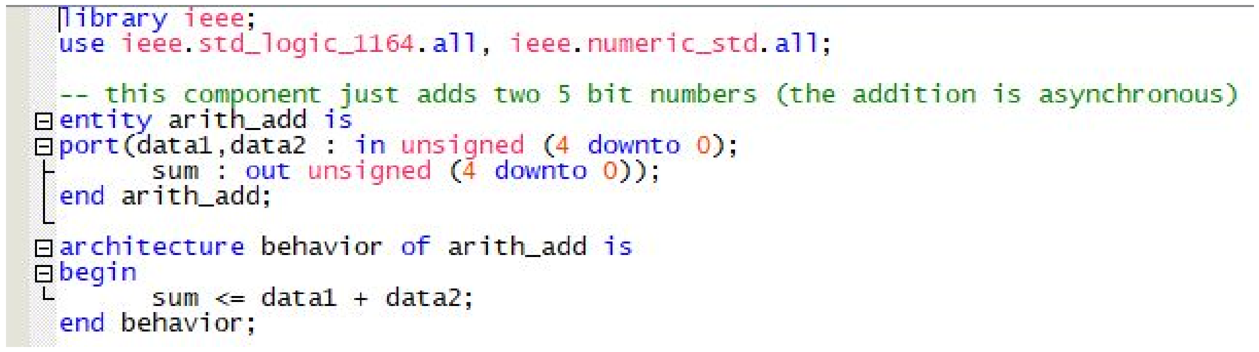
Fig.1 main component

-- Code for Hex Flip Flop.



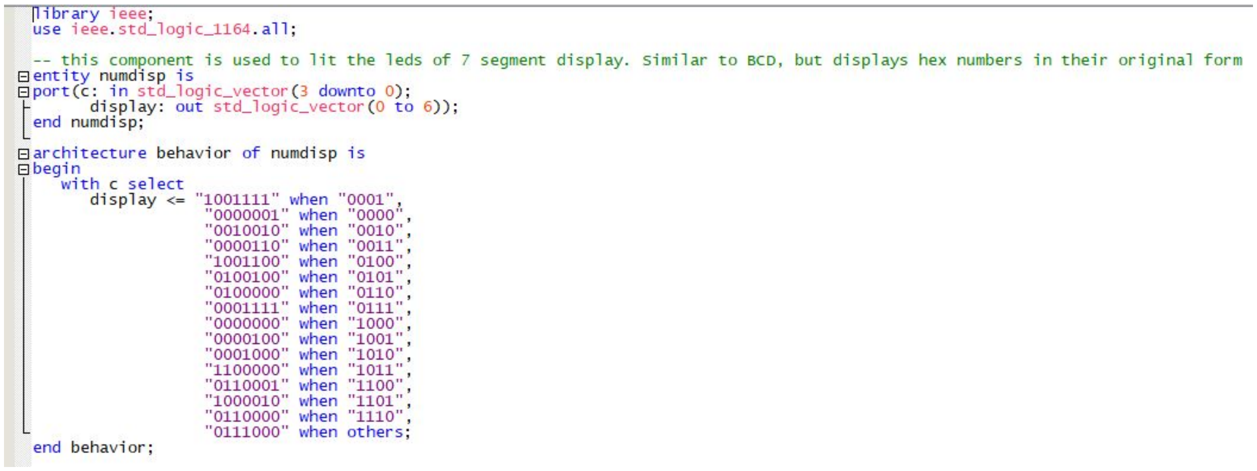
```vhdl
library ieee;
use ieee.std_logic_1164.all, ieee.numeric_std.all;

-- this is component is replicate of general hex-d flip flop which is also use as accumulator
entity hex_ff is
port(clock, reset : in std_logic;
      datain : in unsigned (4 downto 0);
      dataout: out unsigned (5 downto 0));
end hex_ff;

architecture behavior of hex_ff is
begin
      process(clock)
      begin
         if rising_edge(clock) then
            if (reset = '1') then
               dataout <= "000000"; --synchronous reset process
            else
               dataout (4 downto 0) <= datain; -- synchronous 5 bit data out keeping MSB as '0'
            end if;
         end if;
      end process;
end behavior;
```

--Code for Arithmetic addition



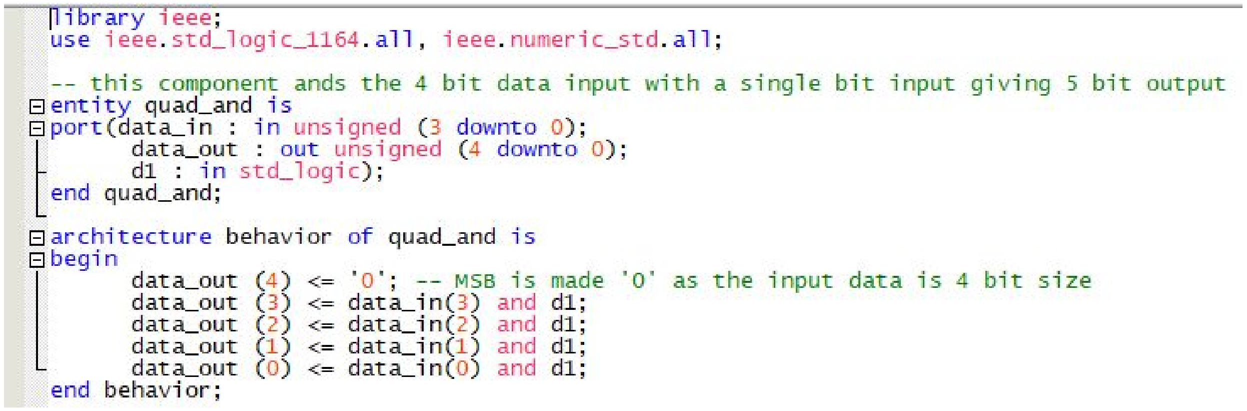
```vhdl
library ieee;
use ieee.std_logic_1164.all, ieee.numeric_std.all;

-- this component just adds two 5 bit numbers (the addition is asynchronous)
entity arith_add is
port(data1,data2 : in unsigned (4 downto 0);
      sum : out unsigned (4 downto 0));
end arith_add;

architecture behavior of arith_add is
begin
      sum <= data1 + data2;
end behavior;
```

--Code for Number Display



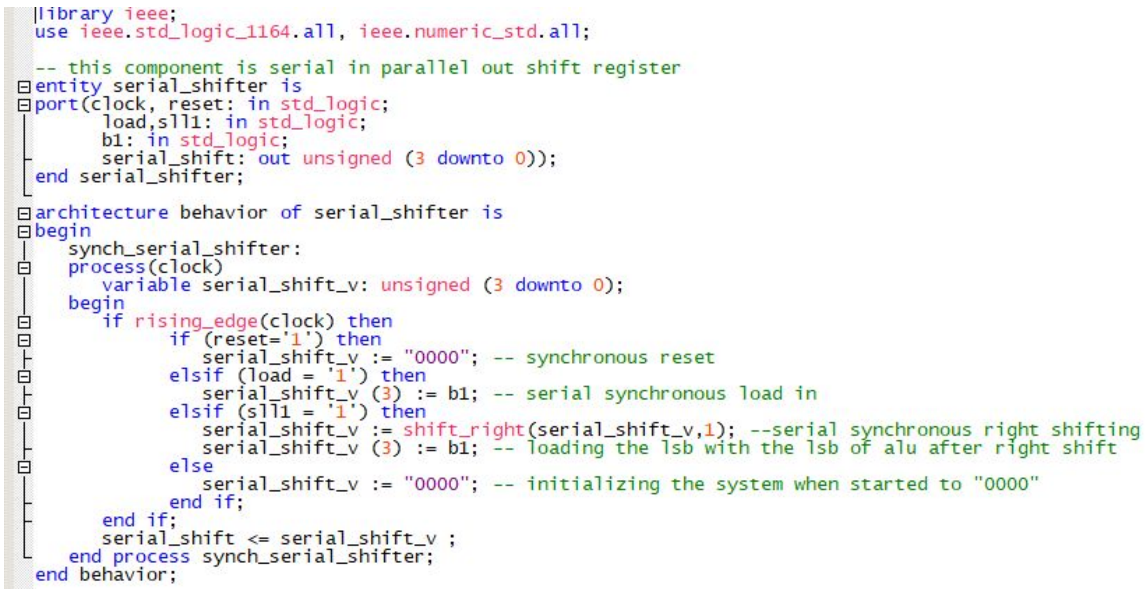
```
library ieee;
use ieee.std_logic_1164.all;

-- this component is used to lit the leds of 7 segment display. Similar to BCD, but displays hex numbers in their original form
entity numdisp is
port(c: in std_logic_vector(3 downto 0);
        display: out std_logic_vector(0 to 6));
end numdisp;

architecture behavior of numdisp is
begin
    with c select
        display <= "1001111" when "0001",
                    "0000001" when "0000",
                    "0010010" when "0010",
                    "0000110" when "0011",
                    "1001100" when "0100",
                    "0100100" when "0101",
                    "0100000" when "0110",
                    "0001111" when "0111",
                    "0000000" when "1000",
                    "0000100" when "1001",
                    "0001000" when "1010",
                    "1100000" when "1011",
                    "0110001" when "1100",
                    "1000010" when "1101",
                    "0110000" when "1110",
                    "0111000" when others;
end behavior;
```

-- Code for Quad AND



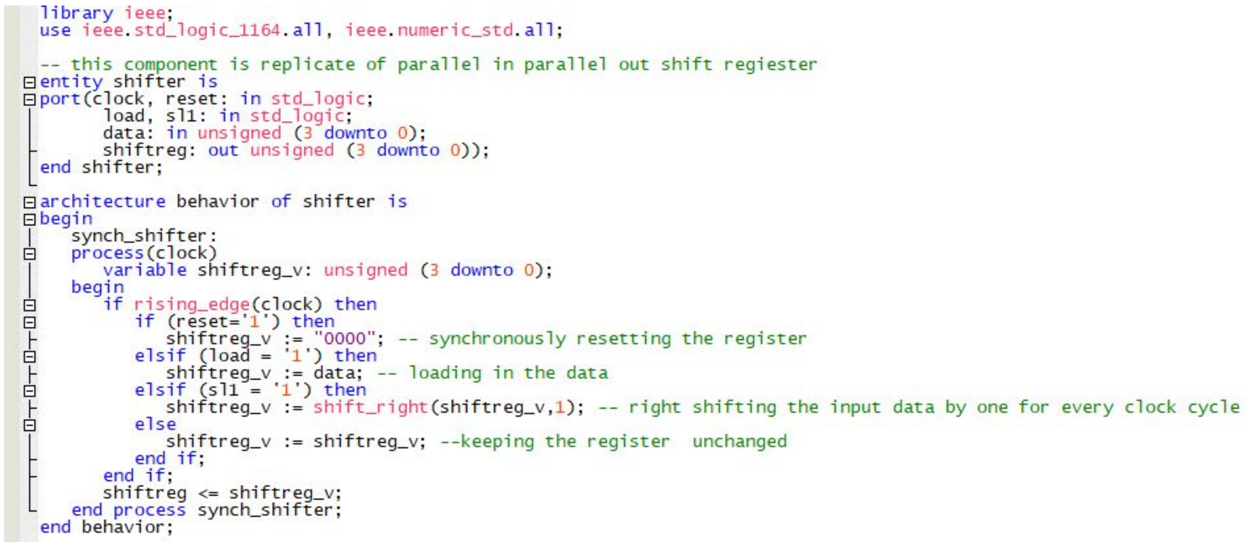
```
library ieee;
use ieee.std_logic_1164.all, ieee.numeric_std.all;

-- this component ands the 4 bit data input with a single bit input giving 5 bit output
entity quad_and is
port(data_in : in unsigned (3 downto 0);
        data_out : out unsigned (4 downto 0);
        d1 : in std_logic);
end quad_and;

architecture behavior of quad_and is
begin
        data_out (4) <= '0'; -- MSB is made '0' as the input data is 4 bit size
        data_out (3) <= data_in(3) and d1;
        data_out (2) <= data_in(2) and d1;
        data_out (1) <= data_in(1) and d1;
        data_out (0) <= data_in(0) and d1;
end behavior;
```

--Code for Serial Shifter

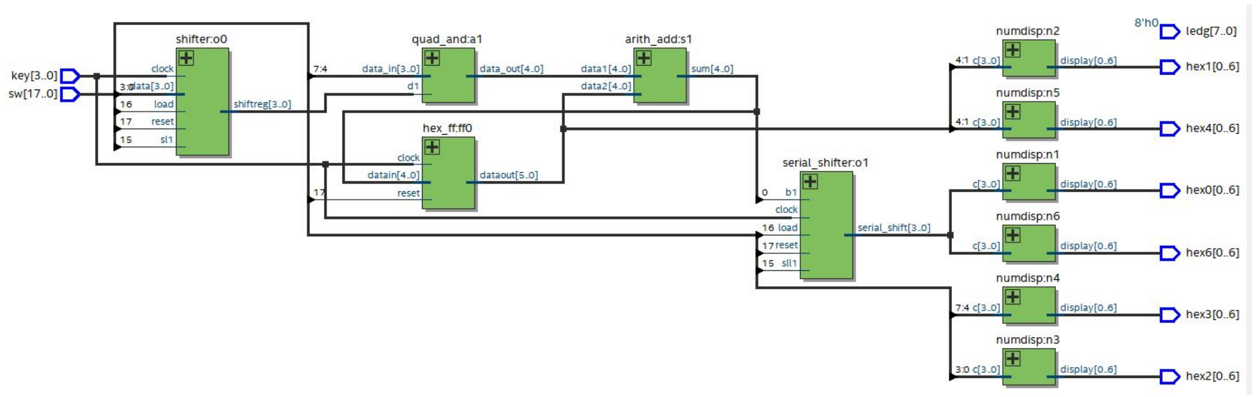


```
library ieee;
use ieee.std_logic_1164.all, ieee.numeric_std.all;

-- this component is serial in parallel out shift register
entity serial_shifter is
port(clock, reset: in std_logic;
        load,sll1: in std_logic;
        b1: in std_logic;
        serial_shift: out unsigned (3 downto 0));
end serial_shifter;

architecture behavior of serial_shifter is
begin
    synch_serial_shifter:
    process(clock)
        variable serial_shift_v: unsigned (3 downto 0);
    begin
        if rising_edge(clock) then
            if (reset='1') then
                serial_shift_v := "0000"; -- synchronous reset
            elsif (load = '1') then
                serial_shift_v (3) := b1; -- serial synchronous load in
            elsif (sll1 = '1') then
                serial_shift_v := shift_right(serial_shift_v,1); --serial synchronous right shifting
                serial_shift_v (3) := b1; -- loading the lsb with the lsb of alu after right shift
            else
                serial_shift_v := "0000"; -- initializing the system when started to "0000"
            end if;
        end if;
        serial_shift <= serial_shift_v ;
    end process synch_serial_shifter;
end behavior;
```

--Code for Shifter


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

-- this component is replicate of parallel in parallel out shift regiester
entity shifter is
port(clock, reset: in std_logic;
        load, sl1: in std_logic;
        data: in unsigned (3 downto 0);
        shiftreg: out unsigned (3 downto 0));
end shifter;

architecture behavior of shifter is
begin
    synch_shifter:
    process(clock)
        variable shiftreg_v: unsigned (3 downto 0);
    begin
        if rising_edge(clock) then
            if (reset='1') then
                shiftreg_v := "0000"; -- synchronously resetting the register
            elsif (load = '1') then
                shiftreg_v := data; -- loading in the data
            elsif (sl1 = '1') then
                shiftreg_v := shift_right(shiftreg_v,1); -- right shifting the input data by one for every clock cycle
            else
                shiftreg_v := shiftreg_v; --keeping the register  unchanged
            end if;
        end if;
        shiftreg <= shiftreg_v;
    end process synch_shifter;
end behavior;
```

Results



While working with the FPGA board, it was observed that the design had to be changed a lot from the actual circuit that was followed in lab3, the prime reason behind the changes was to keep the component concise doing the function with minimum possible control inputs. Few of these simplicities have been mentioned under experimental approach(arithmetic adder) section. The result was observed in 5 clock cycles and not in 4, the reason to this seems to be because of shifter components, which required to get loaded initially and then shift then after the initial load takes additional pulse. The keys and switches needed to be debounced as there were instances where it was observed that more than pulses were passed. Although debounce was not done for this lab, but for phase II of the project has debounced clock in order to avoid error in any operation.

Conclusion

The code developed was able to multiply two 4-bit number and display the 8-bit output. To check this, a variety of inputs were tried and the correct output was obtained. To complete this operation 5 clock cycles were needed.

FTQs

Zubair Nadaph

Move the lower nibble to display HEX6

This has been shown in fig.1 with the instruction:

N: numdisp port map(pq, hex6);

Here numdisp is the component that take the 4 bit input and lits the 7 segment display accordingly to display the given 4 bit #, also ‘pq’ is the output of accumulator.

Ankit Taskar

Change the clock input to be from key0 to Key 2.

For this all the Key(0) which was originally assigned to the clock was changed to Key(2).
Eg : ff0 : hex_ff port map (**key(0)**,sw(17),sp,hfo); was replaced to
ff0 : hex_ff port map (**key(2)**,sw(17),sp,hfo);

References

[1] ADSD Lab 3 Report
[2] Final Project Phase 1 handout.