Experiment 4 - Accelerator and Wrappers

Student Name : Shahzad Momayez , Mohammad Amanluo
Student ID : 810100272 , 810100084

Abstract — This document is a student report to experiment #4 of Digital Logic Laboratory course at ECE Department, University of Tehran. The goal of this experiment is to design an accelerator and wrapper and experiment how accelerators can increase the performance of our work.

Keywords — Accelerator and Wrappers

I. Introduction

Chip is an integrated circuit that integrates multiple components including digital, analog , hardware and software programs all in a single chip. The main core of an SoC is a processor that handles different computational tasks within the system. In addition to the processor, the system includes a memory, Input/Output ports and accelerators.

The main task work of an SoC(system on chip) is a processor that handles different computational tasks within the system. In addition to the processor, the system includes a memory, Input/ Output ports and accelerators.
Accelerator does a single task with a high frequency which is contrary to CPU. To increase the speed of an SOC, hardware accelerators are usually embedded in the system
The processor will dispute some of its tasks to the hardware accelerator and during this time the accelerator performs several of the same or different operations and store the result values in a memory. The CPU will access these results when it finishes its tasks. The focus of this experiment is on Accelerators and how to integrate them in an SOC. Figure 1 shows the block diagram of a typical Embedded system including a processor, an accelerator and a memory.

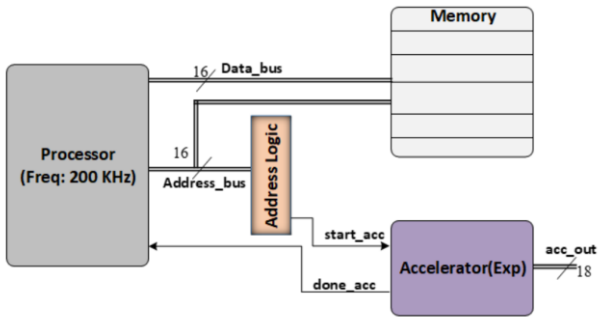

Figure 1: Block diagram of a typical integrated circuit

II. Design

Any components that is in communication with CPU talks to CPU via signals ”start” and ”done”. The embedded system shown in this figure works as follows: When the CPU needs to compute an exponential value, because of the higher estimation speed of accelerator it asks the exponential hardware accelerator to complete this task. In this way the CPU can complete other software tasks in parallel with the accelerator. Before starting the computation, the CPU should send a set of data from memory to the accelerator. This data will be stored in a buffer inside the accelerator. When transferring is finished, CPU initiates the accelerator for an N round exponential estimation. CPU uses its address bus for initiating a component. By decoding, the address bus through an address logic as shown in figure 1, accelerator will have its ”start” signal issued when needed. For simplicity in this experiment you will implement the whole CPU and address logic inside the test bench and when implementing on an FPGA, you will feed ”start” through board switches.

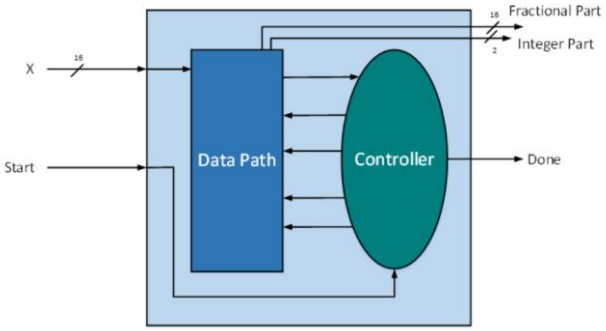

Figure 2: Block diagram of exponential accelerator

In this experiment we implement 3 main parts:

- Exponential Accelerator Engine
- Exponential Accelerator Wrapper
- Implementing Accelerator on FPGA

III. Exponential Engine

this module receives a 16-bit input ”x” and generates an 16-bit output ”Fractionalpart” and 2-bit ”Integerpart”.
The accelerator starts working with a complete pulse on signal ”start” and when the computation is completed signal ”done” will be sent to the processor to acknowledge it.
Here are some of the formulas we used in this experiment:

$$e^{x_i} = 2^{u_i}.e^{v_i}$$

$$e^{x_i} = e^{v_i} << u_i \qquad u_i = \left\lfloor z_i.log_2 e \right\rfloor$$

$$e^{x_i} = \begin{cases} e^{v_i} << u_i & \\ e^{2*v_i} << u_i & \text{if } v_i < 1/2^{(n-1)} \\ \ldots & \text{if } u_i < x_i < u_i + 1 \\ e^{2^{(n-1)}*v_i} << u_i & \end{cases}$$

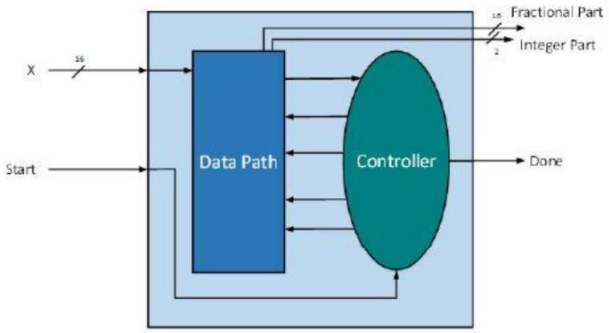

Fig3. Block diagram of exponential accelerator

Here are the Verilog of the project :

```
module exponential(input clk,rst,start, input [15:0] x,
                   output done, output[1:0] intpart, output [15:0]fracpart);

        wire co,zx,initx,ldx,zt,initt,ldt,zr,initr,ldr,zc,ldc,enc,s;

        controller control(clk,rst,start,co,done, zx,initx,ldx,
                           zt,initt,ldt, zr,initr,ldr, zc,ldc,enc, s);

        datapath dP(clk,rst, zx,initx,ldx, zt,initt,ldt,
                    zr,initr,ldr, zc,ldc,enc, s,x,co,{intpart,fracpart});

endmodule
```

```
`timescale 1 ns/ 1 ns
module expTB;
        reg clk, rst, start;
        reg [15:0] x;
        wire done;
        wire [1:0] intpart;
        wire [15:0]fracpart;

        exponential expEng(clk ,rst , start, x, done, intpart, fracpart);

        always #5 clk = ~clk;
        initial begin
                rst = 1'b1;
                clk = 1'b0;
                start = 1'b0;
                #10 start = 1'b1;
                #10 x = 0;
                #5 rst = 1'b0;
                #40 start = 1'b0;
                #10000;
                #50 x = 16'b0100000000000000;
                #1000 start = 1'b1;
                #20 start = 1'b0;
                #10000;
                #50 x = 16'h8000;
                #1000 start = 1'b1;
                #20 start = 1'b0;
                #30000 $stop;
        end

endmodule
```

Fig 4. Verilog of exponential engine

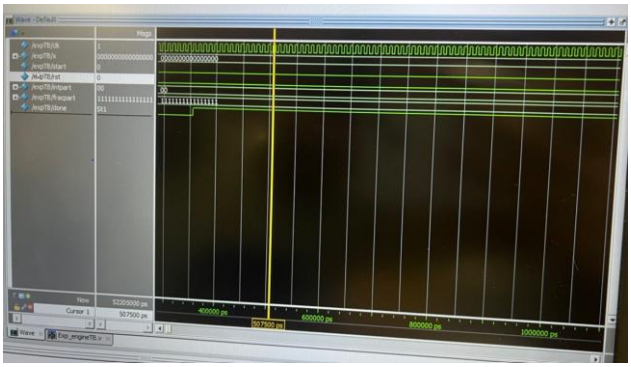

Fig5. X=00
e^0=1 ~00.1111111111111111

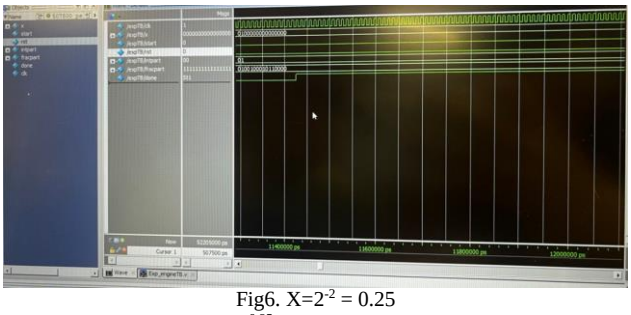

Fig6. $X=2^{-2}$ = 0.25
$e^{0.25}$ =1.28402541

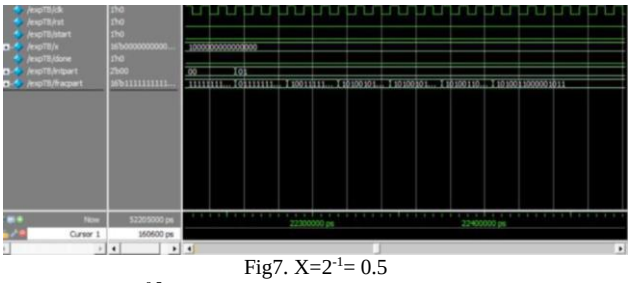

Fig7. $X=2^{-1}$= 0.5
$e^{0.5}$=1.648721270 ~01.10100110000001011

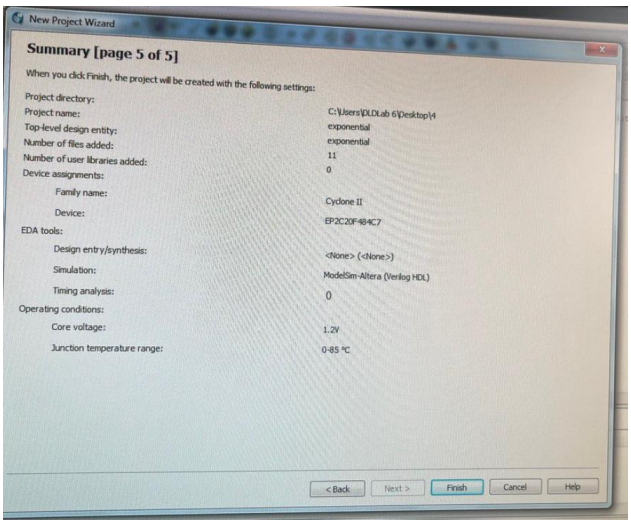

Fig8. summery

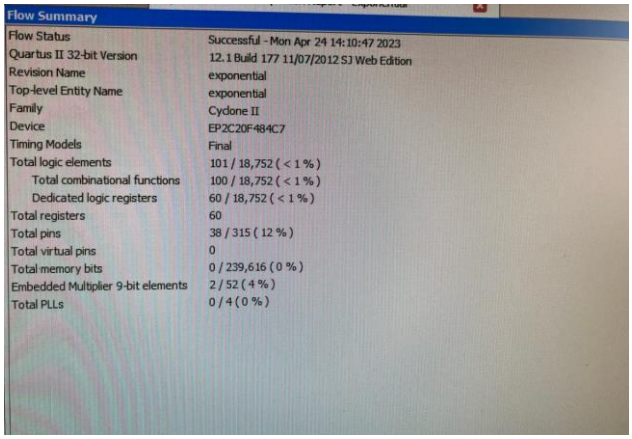

Fig9. Flow summary

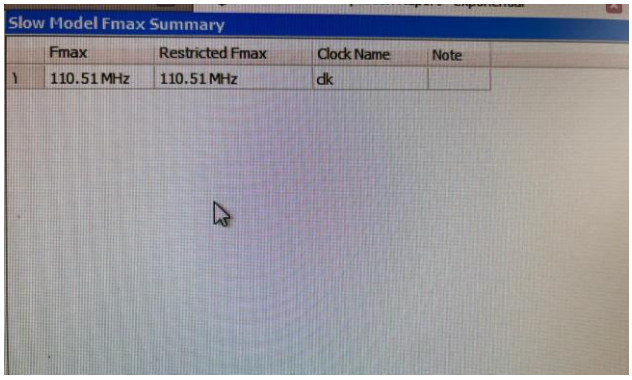

Fig 10 . max frequency of module

The figure above shows the synthesis result in Quartus II Software. It shows that the maximum frequency that this module can operate is 110.51 MHz

IV. Exponential Accelerator Wrapper

We know that the accelerator works with a higher frequency than the processor, for the handshaking signals of ”start” and ”done” the accelerator have to wait for the processor to send and receive these signals with its low frequency. This imposes some timing overhead to the accelerator and hence performance reduction. In order to use this free time, the accelerator can calculate multiple exponential values. Each input value is split into an integer number zi and a fractional number vi as below:

$$xi = zi + vi$$

$$e^{xi} = e^{zi} . e^{vi}$$

we calculate the exponential part by shifting to the left. the wrapper receives single input in the form of a fractional value, vi , and an integer value ui and a start signal from processor.

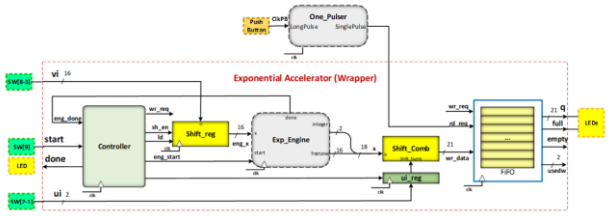

Figure 11: Wrapper for exponential accelerator

In order to shift the output of exponential engine, a combinational shifter is required. The input of this shifter is the ui value that is provided outside the wrapper. To store the value of ui , a register called uireg is used.
When an exponential value is calculated then it should be stored in a FIFO so when the CPU finishes it’s work it can retrieve all the results. When all calculations are finished the controller sends a done signal on the wrapper output.

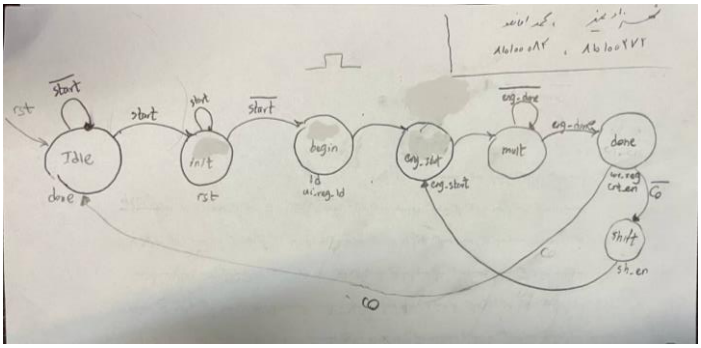

Fig12. controller

```
`define Idle 3'b000
`define Init 3'b001
`define Begin 3'b010
`define Eng 3'b011
`define Mult 3'b100
`define Done 3'b101
`define Shift 3'b110

module counter_ctrl (input clk,rst,zero,ld,enb, input [2:0] in ,output reg [3:0] out, output  co);
        always@(posedge clk, posedge rst)begin
                if(rst == 1'b1)
                        out <= 0;
                else
                        if(zero == 1'b1)
                                out <= 0;
                        else if(ld == 1'b1)
                                out <= in;
                        else if(enb == 1'b1)
                                out <= out + 1'b1;
        end
        assign co = (out >= 4'b1000)? 1'b1: 1'b0;
endmodule
```

```
module cu(input start,clk,rst,eng_done,
output reg done,reset,ld,ui_reg_ld,eng_start,wr_reg,cnt_en,sh_en);
        reg[2:0] ps,ns;
        reg[1:0] count ;
        reg co;
        always@(posedge clk , posedge rst)begin
                if(rst) begin
                        ps<=`Idle;

                end
                else
                        ps<=ns;
        end

        always@(ps ,start,eng_done,co)begin
                case(ps)
                        `Idle: ns = start?`Init:`Idle;
                        `Init: ns = start?`Init:`Begin;
                        `Begin: ns= `Eng;
                        `Eng: ns = `Mult;
                        `Mult : ns = eng_done?`Done:`Mult;
                        `Done: ns = co?`Idle:`Shift;
                        `Shift : ns = `Eng;
                        default ns=`Idle;
                endcase
        end

        always@(ps)begin
                {done,reset,ld,ui_reg_ld,eng_start,wr_reg,cnt_en,sh_en } = 8'b00000000;
                case(ps)
                        `Idle : done = 1;
                        `Init : reset = 1;
                        `Begin : begin ld = 1; ui_reg_ld=1 ; end
                        `Eng : eng_start=1;
                        `Done : begin wr_reg=1;cnt_en=1; end
                        `Shift : sh_en =1;
                endcase
        end
        always@(posedge cnt_en , posedge rst)begin
                if(rst){co ,count} = 3'b000 ;
                else {co ,count} = count +1;
        end

endmodule
```

Fig13. Controller verilog

```
`timescale 1 ns/ 1 ns
module cntrlerTB;
        reg start,rst,eng_done;
        reg clk = 0;
        wire done,reset,ld,ui_reg_ld,eng_start,wr_reg,cnt_en,sh_en;

        cu c(start,clk,rst,eng_done,done,reset,ld,ui_reg_ld,eng_start,wr_reg,cnt_en,sh_en);

        always #1 clk = ~clk;
        initial begin
                #5 rst = 1'b1;
                #5 rst = 1'b0;
                #5 start = 1'b1;
                #5 start = 1'b0;
                #5 eng_done = 1;
                #30 $stop;
        end

endmodule
```

Fig14. Controller testbench

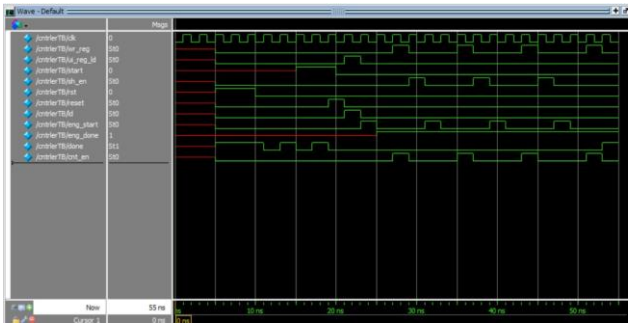

Fig15 . wave form

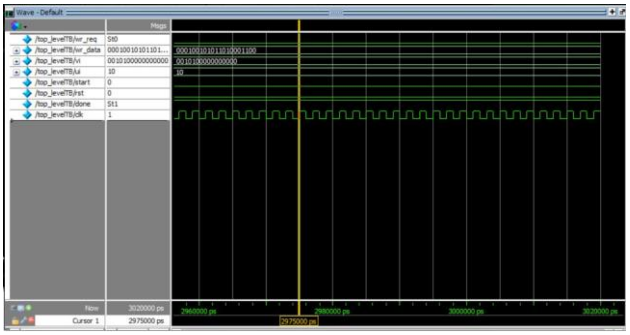

Fig16. ui= 10 =(2)d
vi=0010100000000000=0.03125+0.12500=0.15625
expected : $4*e^{(0.15625)}$= 4.67647378468
achived: 100.101011010001100 = 4.677734375

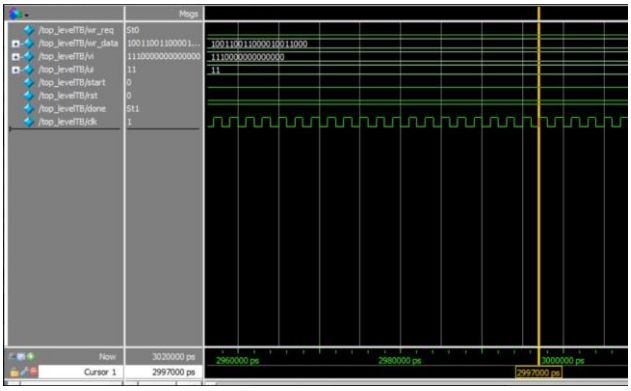

Fig17. ui= 11 =(3)d
vi=1110000000000000=0.5+0.25+0.12500=0.875
expected : $8*e^{(0.875)}$= 19.1910023517
achived: 10011.0011000010011000 = 19.1904296875

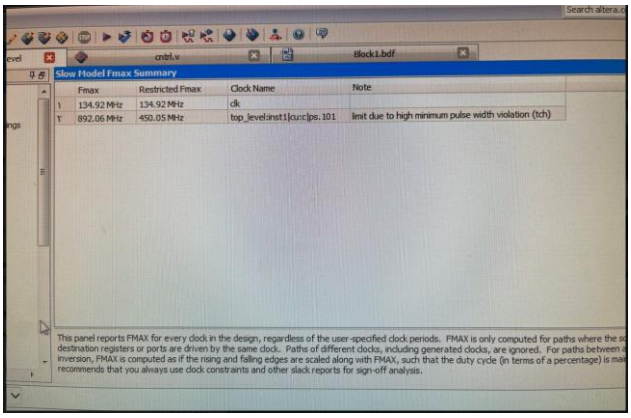

Fig 18 . frequency is shown in the picture above

V. Implementing Accelerator on FPGA

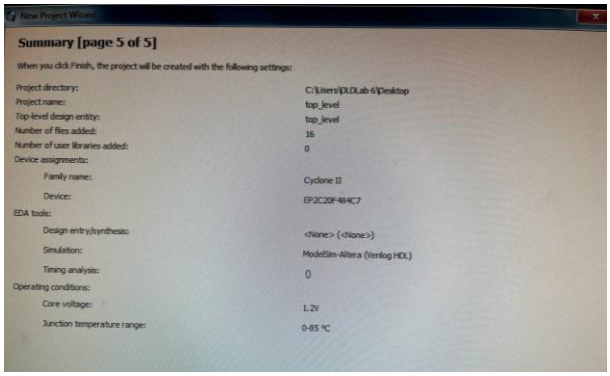

Fig 20.summary

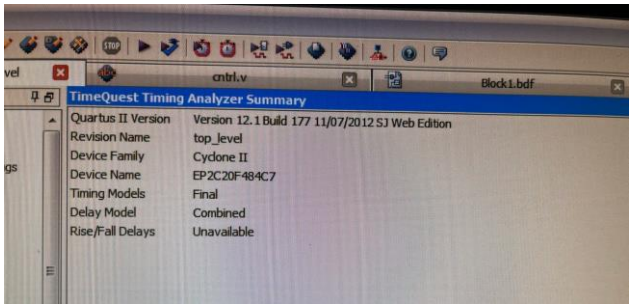

Fig 21. Time Quest

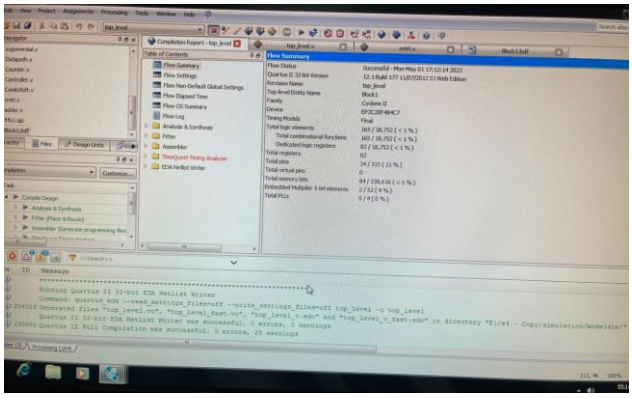
Fig 22. Flow summary

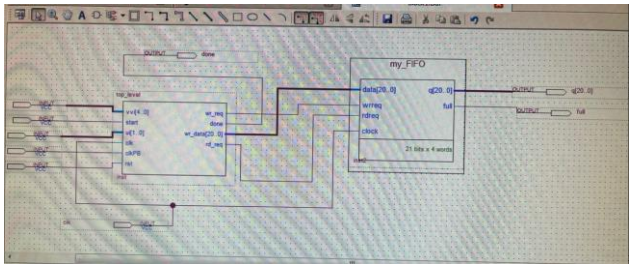

Fig23 . scheme we implemented in Quartus II

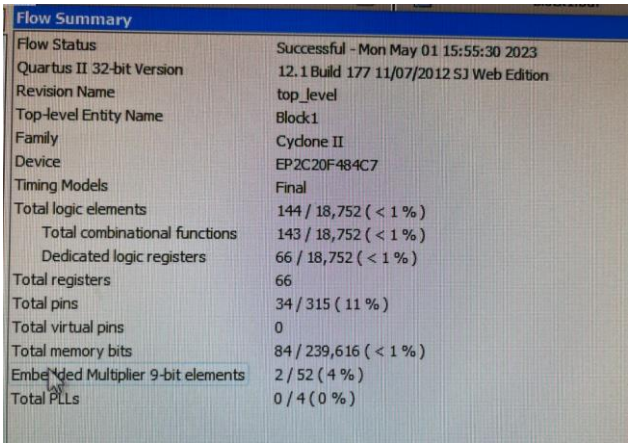

Fig 24 . Flow summary

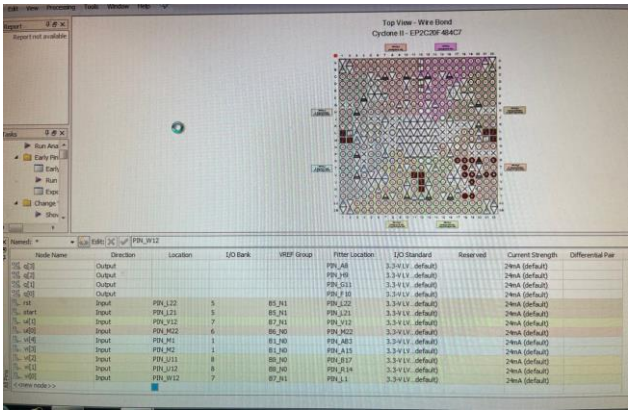
Fig 25 .pins implementation

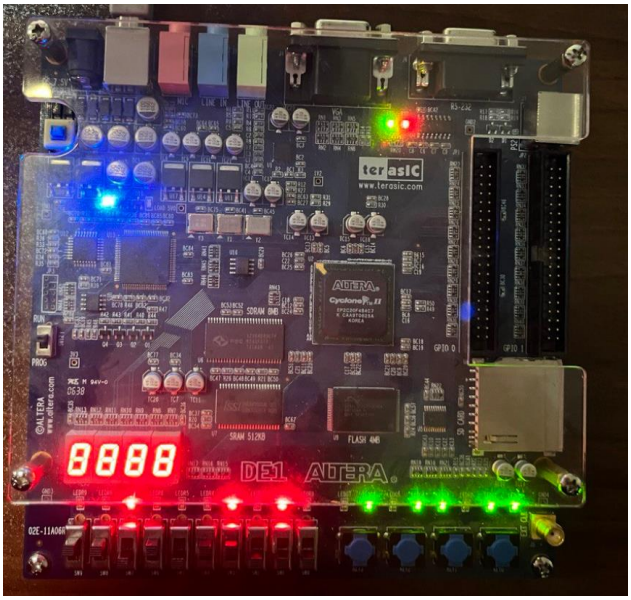

Fig 26.

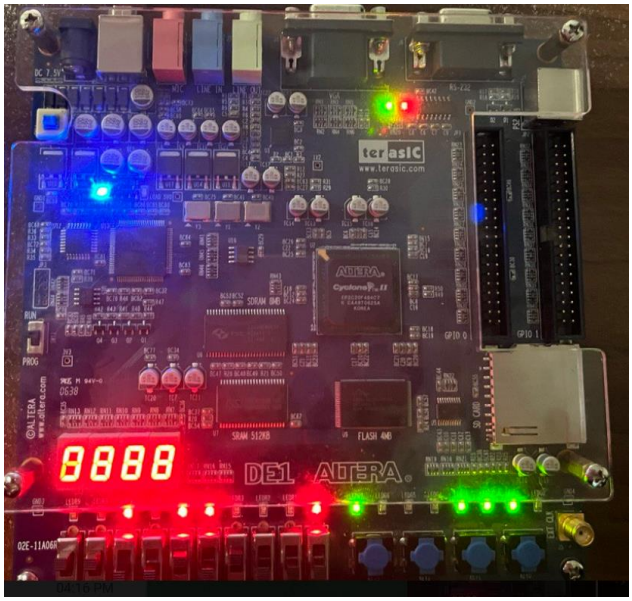

Fig 27.

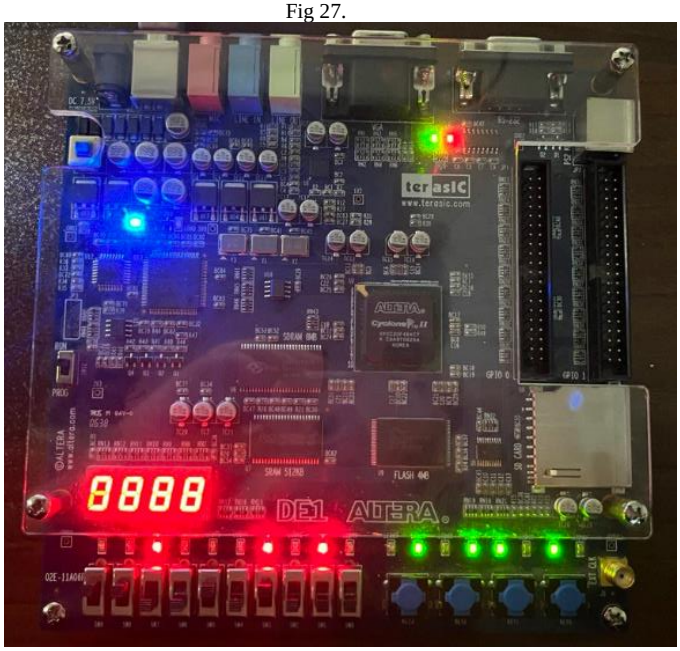

Fig28.

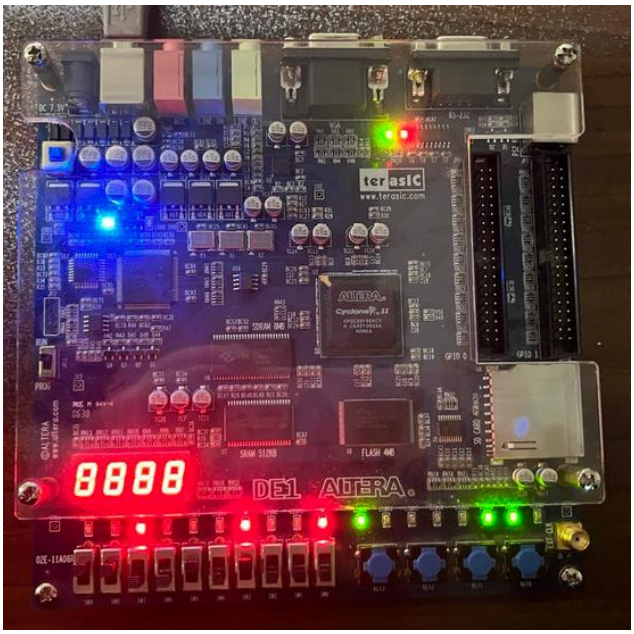

Fig29. FPGA implementation

VI.CONCLUSIONS

All in all, we designed these acclator and wrapper and we implemented our work on FPGA.

ACKNOWLEDGMENT

This report was prepared and developed by Shahzad Momayez(SID:810100272) and Mohammad Amanlou(SID:810100084), bachelor students of Computer engineering at University of Tehran, under the supervision of Professor Zain Navabi.