Experiment 4 – Accelerators and Wrappers

Parsa Sattari - 810199436 / Iman Rasouli - 810199425

Abstract *— In this experiment, we designed a wrapper for an accelerator, which is essential for the communication between the accelerator and the CPU of the SoC. To be able to see the results separately, a few simplifications were made.*

Keywords— Accelerator, Wrapper, SoC, FIFO, Buffer

I. EXPERIMENT

A. Exponential Engine

First, we examine the accuracy of the exponential engine by the following testbench:

```
`timescale 1ns/1ns
module Accelarator_TB();

  reg clk = 1'b0, rst = 1'b0, start = 1'b0;
  reg [15:0] x = 16'b1111_1111_1111_1111; //1
  wire done;
  wire [1:0] intpart;
  wire [15:0] fracpart;
  exponential CUT(clk, rst, start, x, done, intpart, fracpart);

  always #5 clk = ~clk;
  initial begin
  #11 rst = 1'b1;
  #10 rst = 1'b0;
  #5 start = 1'b1;
  #10 start = 1'b0;
  #500 x = 16'b1011110101110000; //0.74
  #5 start = 1'b1;
  #10 start = 1'b0;
  #500 x = 16'b0011001100110011; //0.2
  #5 start = 1'b1;
  #10 start = 1'b0;
  #500 $stop;
  end
endmodule
```

Fig 1. Exponential Engine Testbench

We check 3 values: $e^1, e^{0.74}, e^{0.2}$. We have "*intpart*" and "*fracpart*" that final result will have the form below:

$$\{intpart.fracpart\}$$

I. $e^1 = 2.7182818285$

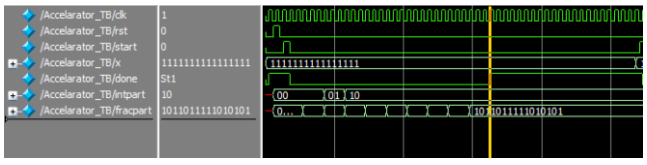

Fig 2. Exponential Engine_ Calculating exp (1)

intpart = 2, fracpart = 0.7180938720 → 2.718093872

The result is almost correct.

II. $e^{0.74} = 2.0959355145$

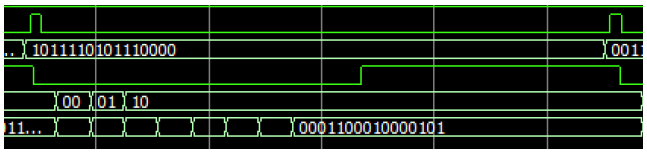

Fig 3. Exponential Engine_ Calculating exp (0.74)

→ 2.0957794189453125

III. $e^{0.2} = 1.2214027582$

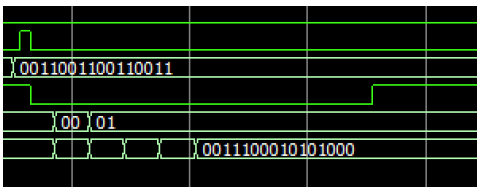

Fig 4. Exponential Engine_ Calculating exp (0.2)

→ 1.2213134765625

After checking the accuracy, we synthesize the exponential engine in Quartus.

Flow Summary	
Flow Status	Successful - Tue Jan 10 23:02:52 2023
Quartus II 64-Bit Version	13.0.1 Build 232 06/12/2013 SP 1 SJ Web Edition
Revision Name	exponential
Top-level Entity Name	exponential
Family	Cyclone II
Device	EP2C20F484C7
Timing Models	Final
Total logic elements	101 / 18,752 (< 1 %)
Total combinational functions	100 / 18,752 (< 1 %)
Dedicated logic registers	60 / 18,752 (< 1 %)
Total registers	60
Total pins	38 / 315 (12 %)
Total virtual pins	0
Total memory bits	0 / 239,616 (0 %)
Embedded Multiplier 9-bit elements	2 / 52 (4 %)
Total PLLs	0 / 4 (0 %)

Fig 5. Synthesis Result

Slow Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name	Note
1	112.13 MHz	112.13 MHz	clk	

Fig 6. Maximum Frequency of Accelerator

B. Exponential Accelerator Wrapper

Since the accelerator data will be accessed before and after completing CPU task, the data has to be stored in memory elements in the accelerator wrapper when CPU is busy with other works. To simulate memory map communication between the CPU and accelerator, a buffer is required. We use a 1-Port ROM for this purpose.

As it mentioned in manual the wrapper needs a controller with proper states and a counter to access the ROM's data.

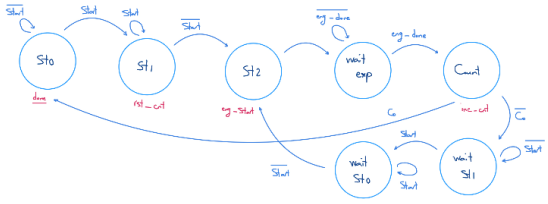

Fig 7. Wrapper Controller State Diagram

```
module Wrapper_Controller(input clk, rst, start, Co, eng_done, output reg done, inc_count, rst_count, eng_start);
  reg [2:0] ps, ns;
  parameter [2:0]
  St0 = 0, St1 = 1, St2 = 2, wait_exp = 3, cnt = 4, wait_St1 = 5, wait_St0 = 6;

  always@(clk,start,Co)begin
    ns = St0;
    case(ps)
      St0: ns = (start) ? St1 : St0;
      St1: ns = (start) ? St1 : St2;
      St2: ns = wait_exp;
      wait_exp: ns = (eng_done) ? cnt : wait_exp;
      cnt: ns = (Co) ? St0 : wait_St1;
      wait_St1: ns = (start) ? wait_St0 : wait_St1;
      wait_St0: ns = (start) ? wait_St0 : St2;
    endcase
  end

  always@(ps)begin
    {done, inc_count, rst_count, eng_start} = 4'b0000;
    case(ps)
      St0: done = 1'b1;
      St1: rst_count = 1'b1;
      St2: eng_start = 1'b1;
      wait_exp:;
      cnt: inc_count = 1'b1;
      wait_St1:;
      wait_St0:;
    endcase
  end

  always@(posedge clk, posedge rst)begin
    if(rst)
      ps <= St0;
    else
      ps <= ns;
    end
endmodule
```

Fig 8. Wrapper Controller Verilog Description

As it shown in state diagram above, "done" signal will be asserted after 5 calculations and after every single calculation, "*eng_done*" is asserted.

As we have controller, buffer and exponential engine we will have the wrapper.

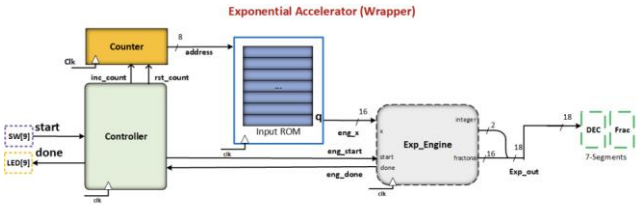

Fig 9. Wrapper (include components) [1]

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

    wire done, eng_done, Co, eng_start, inc_count, rst_count;
    wire [7:0] address;
    wire [15:0] eng_x;

    exponential exp(clk, rst, eng_start, eng_x, eng_done, intpart, fracpart);
    Wrapper_Controller controller(clk, rst, start, Co, eng_done, done, inc_count, rst_count, eng_start);
    Wrapper_Counter count(clk, rst_count, inc_count, address, Co);
    ROM rom(clk, address, eng_x);

endmodule
```

Fig 10. Wrapper Verilog Description (wiring components)

And we have the testbench below to examine the accuracy of wrapper

```
`timescale 1ns/1ns
module Wrapper_TB();

  reg clk = 1'b0, rst = 1'b0, start = 1'b0;
  wire done;
  wire [1:0] intpart;
  wire [15:0] fracpart;

  wrapper CUT(clk, rst, start, done, intpart, fracpart);

  always #5 clk = ~clk;
  initial begin
  #11 rst = 1'b1;
  #10 rst = 1'b0;
  #5 start = 1'b1;
  #10 start = 1'b0;
  #500;
  #5 start = 1'b1;
  #10 start = 1'b0;
  #500;
  #5 start = 1'b1;
  #10 start = 1'b0;
  #500;
  #5 start = 1'b1;
  #10 start = 1'b0;
  #500;
  #5 start = 1'b1;
  #10 start = 1'b0;
  #1000 $stop;
  end
endmodule
```

Fig 11. Wrapper Testbench

As it shown in testbench because we don't have a FIFO in the output, we need 5 complete "*start*" pulse to have the calculations.

Using data in previous part (Exponential Engine) we will have the result:

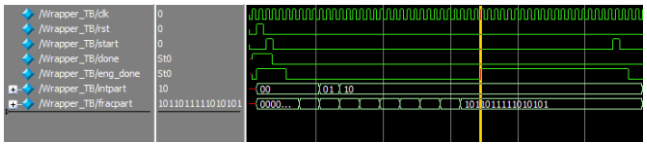

Fig 12. Wrapper Output (exp (1))

Note: We add "*eng_done*" signal to see when every single calculation will be done.

The first ROM's data (address = 8'b00000000) is 1 which result will be $e^1 = 2.71828$. And the wrapper output is approximately same.

We have the accuracy for rest of the ROM's data.

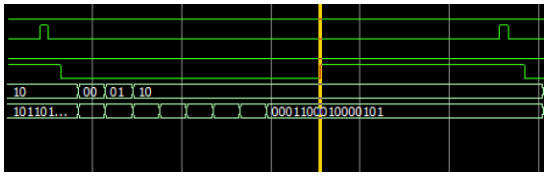

Fig 13. Wrapper Output (exp (0.74))

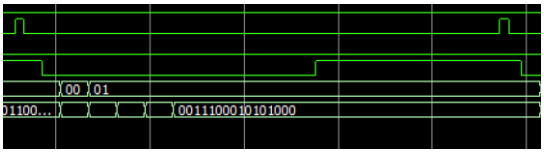

Fig 14. Wrapper Output (exp (0.2))

And we have two more "*x*" input: x = 0.66 & x = 1

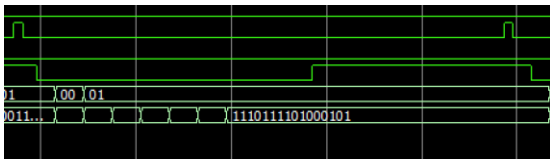

Fig 15. Wrapper Output (exp (0.66))

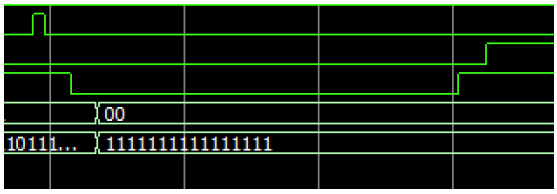

Fig 16. Wrapper Output (exp (1))

As can be seen from the waveform, "*done*" signal is asserted after 5 calculations.

C. Implementing Accelerator on FPGA

Using aforementioned Verilog descriptions, which are Exp_Engine and Wrapper_Controller, and the description below, we build symbols. Besides, we use LPM library for the ROM.

```
module Wrapper_Counter (input clk, rst_count, inc_count, output reg [7:0] address, output Co);
    always@(posedge clk, posedge rst_count)begin
        if(rst_count == 1'b1)
            address <= 0;
        else
          if(inc_count == 1'b1)
            address <= address + 1'b1;
    end
    assign Co = (address == 8'b00000100) ? 1'b1 : 1'b0;
endmodule
```

Fig 17. Wrapper_Counter Verilog description

To show the results on the FPGA, we also need a converter that enables us to use the Seven-Segment display. For this purpose, we use the description below.

```
module Seven_Segment(input[3:0] count, output reg [6:0] SSD);
    always@(count)begin
        case(count)
            4'b0000: SSD = 7'b1000000;
            4'b0001: SSD = 7'b1111001;
            4'b0010: SSD = 7'b0100100;
            4'b0011: SSD = 7'b0110000;
            4'b0100: SSD = 7'b0011001;
            4'b0101: SSD = 7'b0010010;
            4'b0110: SSD = 7'b0000010;
            4'b0111: SSD = 7'b1111000;
            4'b1000: SSD = 7'b0000000;
            4'b1001: SSD = 7'b0010000;
            4'b1010: SSD = 7'b0001000;
            4'b1011: SSD = 7'b0000011;
            4'b1100: SSD = 7'b1000110;
            4'b1101: SSD = 7'b0100001;
            4'b1110: SSD = 7'b0000110;
            4'b1111: SSD = 7'b0001110;
        endcase
    end
endmodule
```

Fig 18. SSD Verilog description

We use the first ssd for the int part. To show the fraction part, we convert the first four bits of it to hex and assign them to the second ssd. The same thing is done for the next eight bits of the fraction part.

Now by wiring the elements in a proper way, our design will be completed.

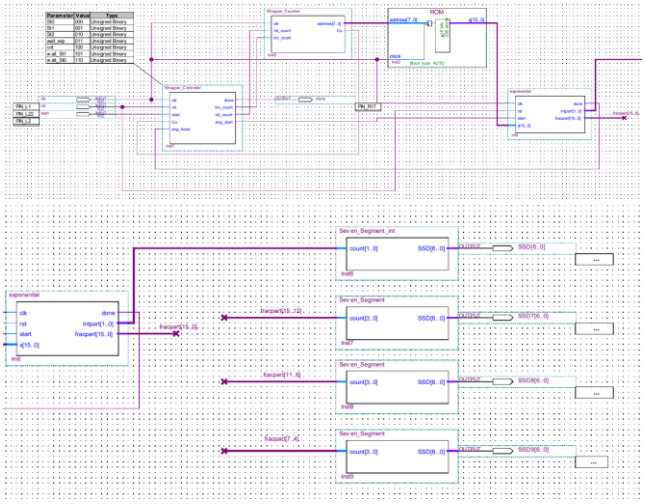

Fig 19. Total design

Eventually, pins are assigned as mentioned in the manual and the result can be seen below:

After resetting the FPGA, the result sets to zero. (Fig. 19)

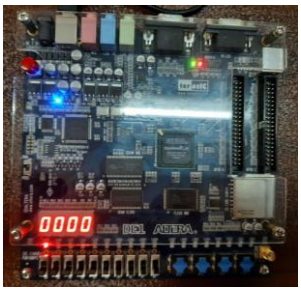

Fig 20. Resetting the FPGA

According to the data that was in the .mif file

```
WIDTH=16;
DEPTH=256;
ADDRESS_RADIX=HEX;
DATA_RADIX=BIN;
CONTENT BEGIN
00 : 1111111111111111;                  -- memory address : data
01 : 1011110101110000;
02 : 0011001100110011;
03 : 1010100011110101;
04 : 0000000000000000;

END;
```

Fig 21. .mif file

The first value must be:

$$e^1 = 2.7182 = 2.1011\ 0111\ 1101 = 2.B7D$$

That can be seen on the FPGA.

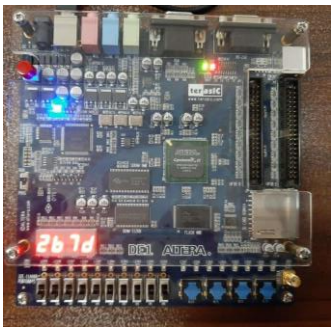

Fig 22. First output

The calculations for the next four values can also be seen below.

$$e^{0.74} = 2.09593 = 2.0001\ 1000\ 1000 = 2.188$$

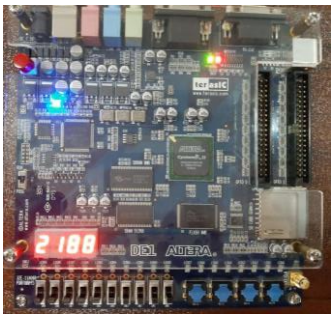

Fig 23. Second output

$$e^{0.2} = 1.2214 = 1.0011\ 1000\ 1010 = 1.38A$$

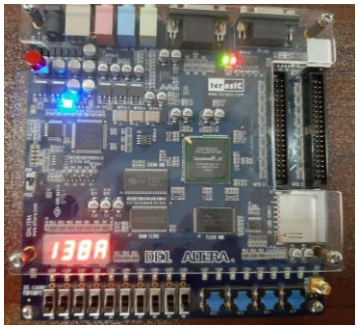

Fig 24. Third output

$$e^{0.66} = 1.9347 = 1.1110\ 1111\ 0100 = 1.EF4$$

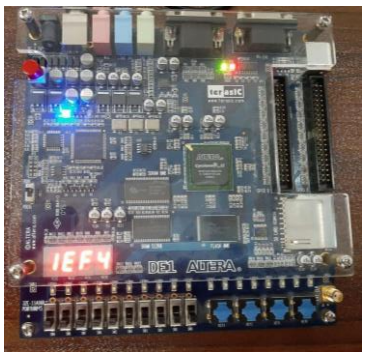

Fig 25. Fourth output

$$e^{0} = 0.9999 = 0.1111\ 1111\ 1111 = 0.FFF$$

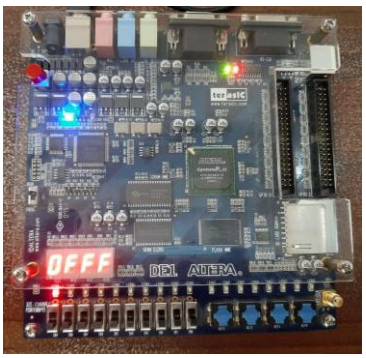

Fig 26. Fifth output

Finally, after the fifth cycle, the "*done*" signal will be asserted (one of the Red LEDs on FPGA).

II. Conclusions

In every SoC, there are components such as Accelerators which need a wrapper to work properly. This wrapper can be implemented by designing a controller and some other minor parts.

III. References

[1] Katayoon Basharkhah and Zahra Jahanpeim and Zain Navabi, *Digital Logic Laboratory*, University of Tehran, Fall 1401.