Experiment 4 - Accelerator and Wrappers

Amir Naddaf Fahmideh,
810101540,
Mostafa Kermani nia,
810101575

***Abstract*— This is the report of experiment 4 of Digital Logic Laboratory course at ECE Department, University of Tehran. In this experiment we are going to design a wrapper and accelerator and see how it effects in a good way in performance of our work.**

***Keywords*— Wrapper, Accelerator, Exponential Engine, SoC, Handshaking, FIFO, Processor**

I. Introduction

A System on Chip (SoC) is an integrated circuit that combines multiple components, including a processor, memory, I/O ports, and accelerators, all on a single chip (see Figure 1). The main processor handles various computational tasks, while accelerators are specialized units designed to execute specific tasks quickly and efficiently.

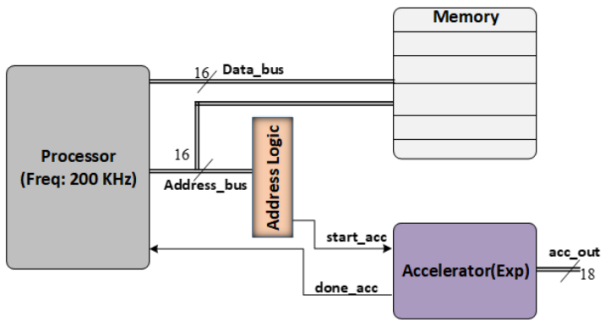


Fig. 1 Block diagram of a typical integrated circuit

Key Concepts:

- Processor: The central unit handling diverse tasks.
- Accelerators: Dedicated units for specific tasks, enabling high-speed operation due to simpler, focused datapaths.
- Handshaking: Communication protocol between components, using signals like "start" and "done".

Working of an Embedded System with an Accelerator:

- The CPU delegates specific tasks, such as computing exponential values, to the accelerator (see Figure 1).
- The accelerator performs these tasks faster than the CPU could, allowing the CPU to handle other tasks simultaneously.
- Data transfer: The CPU sends necessary data from memory to the accelerator's buffer before computation begins.
  The CPU starts the accelerator via an address logic that decodes the address bus and triggers the "start" signal (see Figure 1).

Experiment Goals:

- Understand the concept of SoC and accelerators.
- Learn handshaking mechanisms in SoC.
- Implement and integrate an accelerator within an SoC.

Components to Implement:

1. Exponential Accelerator Engine: The core unit performing exponential calculations.
2. Exponential Accelerator Wrapper: Interfaces the engine with the rest of the system.
3. FPGA Implementation: Testing the accelerator on an FPGA board, using board switches to control the "start" signal.

This experiment focuses on integrating an exponential accelerator into a SoC, demonstrating how specialized hardware can enhance system performance by offloading specific tasks from the CPU.

II. Simple Example of Exponential Engine

1. Simulation outcomes

The simulation results for x = 0.25 which in binary is 0100000000000000 is shown in fig 2. The value of e^0.25 is 1.284025 which in binary is 01.0100100010110101111 and it is very close to our simulation output. (nearly the same). The output in binary is 01.01001000010110000 which is 1.2839 in decimal and it is very close to the actual value we calculated above.

The simulation results for x = 0.5 which in binary is 1000000000000000 is shown in fig 3. The value of e^0.5 is 1.6487 which in binary is 01.10100110000100 and it is very close to our simulation output. (nearly the same). The output in binary is 01.1010011000001011 which is 1.6486 in decimal and it is very close to the actual value we calculated above.

The simulation results for x = 0.75 which in binary is 1100000000000000 is shown in fig 4. The value of e^0.75 is

2.117 which in binary is 10.000111011111001 and it is very close to our simulation output. (nearly the same). The output in binary is 10.0001110111101011 which is 2.116 in decimal and it is very close to the actual value we calculated above.

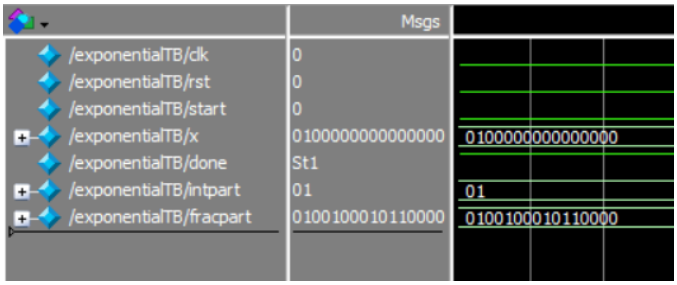


Fig. 2 Simulation results for x = 0.25

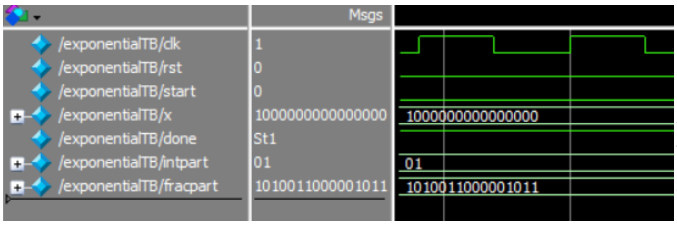


Fig. 3 Simulation results for x = 0.5

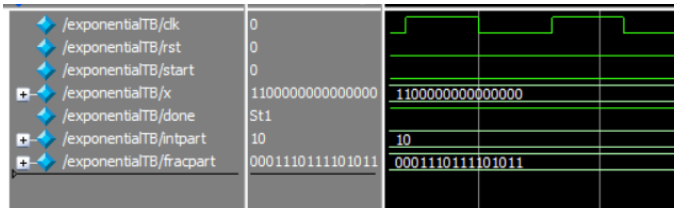


Fig. 4 Simulation results for x = 0.75

2. *FPGA resources*

The resources of FPGA for the Exponential Engine is shown in fig 5.

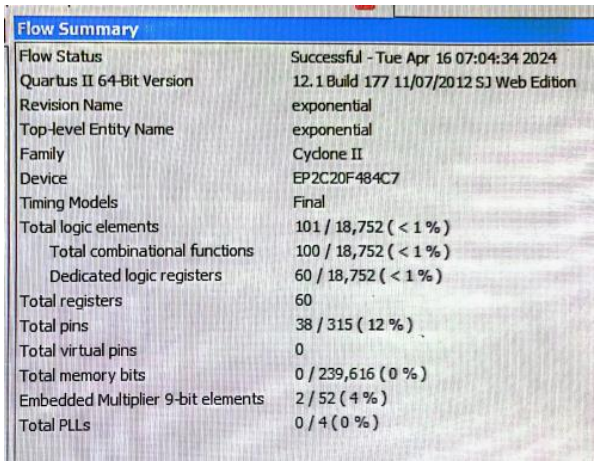


Fig. 5 FPGA resources for the Exponential Engine

3. *Frequency*

The frequency of exponential engine is shown in fig 6.

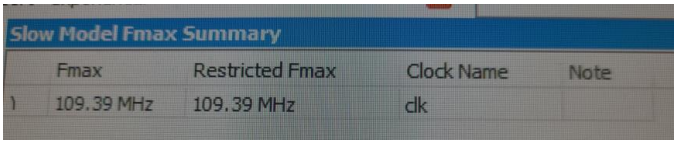


Fig. 6 Frequency of exponential

III. Exponential Accelerator Wrapper Simulation

1. *Explanation*

Because our exponential engine has higher frequency than the processor, we need a wrapper to do the handshaking between them. The goal of this wrapper (fig 7) is to calculate 4 values ($e^{v_i} << u_i$, $e^{2v_i} << u_i$, $e^{4v_i} << u_i$, $e^{8v_i} << u_i$). So It has a shift register which will be initialized by v_i and then after calculating each one of the above values, It shifts v_i to creat $2v_i$, $4v_i$ and $8v_i$. The output of this shift register is going to be the input of the exponential engine to calculate the $\exp(v_i)$ (or $\exp(2v_i)$, $\exp(4v_i)$, $\exp(8v_i)$). The wrapper has a shift register which contains u_i and the output of this register and the output of exponential engine goes into a combinational shift register which shifts the output of the exponential engine, u_i times we would calculate $e^{v_i} << u_i$, $e^{2v_i} << u_i$, $e^{4v_i} << u_i$, $e^{8v_i} << u_i$ values. So it uses the exponential engine with the output of the v_i shift register and each time it shifts the result with the combinational shift register and then it shifts v_i with the shift register and does the same again. (we will explain about the controller part in next part)

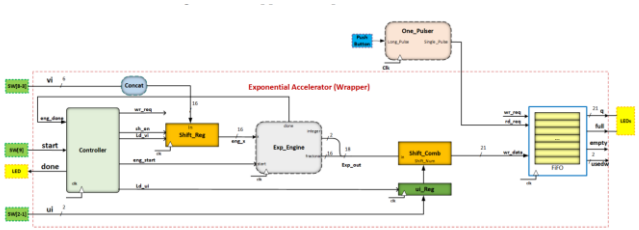
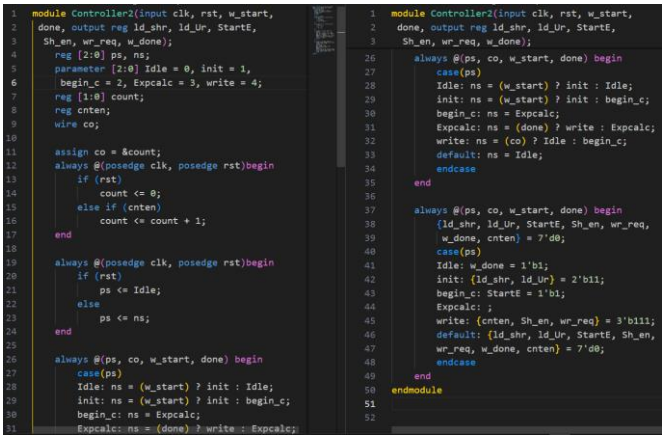


Fig. 7 Wrapper for exponential accelerator

2. *Controller representation*

The code of controller is shown in fig 8 and the state machine is shown in fig 9.



```
module Controller2(input clk, rst, w_start,
 done, output reg ld_shr, ld_Ur, StartE,
  Sh_en, wr_req, w_done);
    reg [2:0] ps, ns;
    parameter [2:0] Idle = 0, init = 1,
     begin_c = 2, Expcalc = 3, write = 4;
    reg [1:0] count;
    reg cnten;
    wire co;

    assign co = &count;
    always @(posedge clk, posedge rst)begin
        if (rst)
            count <= 0;
        else if (cnten)
            count <= count + 1;
    end

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

    always @(ps, co, w_start, done) begin
        case(ps)
        Idle: ns = (w_start) ? init : Idle;
        init: ns = (w_start) ? init : begin_c;
        begin_c: ns = Expcalc;
        Expcalc: ns = (done) ? write : Expcalc;
```

```
module Controller2(input clk, rst, w_start,
 done, output reg ld_shr, ld_Ur, StartE,
  Sh_en, wr_req, w_done);
    always @(ps, co, w_start, done) begin
        case(ps)
        Idle: ns = (w_start) ? init : Idle;
        init: ns = (w_start) ? init : begin_c;
        begin_c: ns = Expcalc;
        Expcalc: ns = (done) ? write : Expcalc;
        write: ns = (co) ? Idle : begin_c;
        default: ns = Idle;
        endcase
    end

    always @(ps, co, w_start, done) begin
        {ld_shr, ld_Ur, StartE, Sh_en, wr_req,
         w_done, cnten} = 7'd0;
        case(ps)
        Idle: w_done = 1'b1;
        init: {ld_shr, ld_Ur} = 2'b11;
        begin_c: StartE = 1'b1;
        Expcalc: ;
        write: {cnten, Sh_en, wr_req} = 3'b111;
        default: {ld_shr, ld_Ur, StartE, Sh_en,
        wr_req, w_done, cnten} = 7'd0;
        endcase
    end
endmodule
```

Fig. 8 code of controller

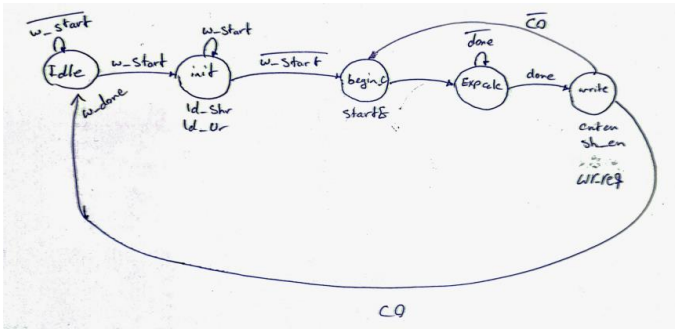


Fig. 9 state machine

As you can see in the state machine of controller, when the wrapper gets a complete pulse on w_start input, It begins to work. It will load the input in its registers in init state and then after the w_start becomes zero again (wrapper gets the complete pulse of w_start) it begins to work. In the begin state the startE signal becomes one which means the exponential engine is ready to work. Then the exponential engine works until we get the done signal from it. Controller has a 2bit counter which its carry out is showing if the wrapper is calculated the 4 values or not. In the write state the v_i shift register shifts the v_i for the next calculation and wr_req will be sent to FIFO to write the calculated data in it. This loop is going on until we calculate the mentioned 4 values and then the carry out becomes one and we go to Idle state and issue the w_done signal to say the wrapper work is done and waiting for the next complete pulse to run the wrapper again.

3. *Simulation outcomes for v_i examples*

For v_i = 11111 and u_i = 00 the expected values and the binary of them are:

- e^0.121093 = 1.12872988 (fig 10)

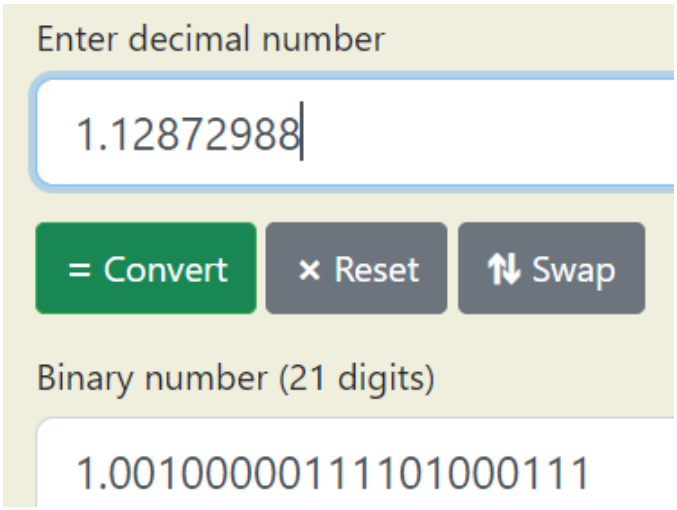


Fig. 10 binary representation of 1.12872988

- e^0.24184 = 1.2735904 (fig 11)

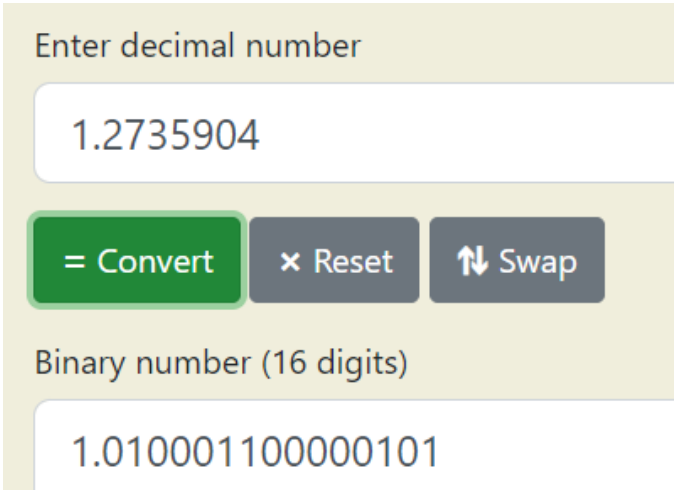


Fig. 11 binary representation of 1.2735904

- e^0.4843 = 1.62303848 (fig 12)

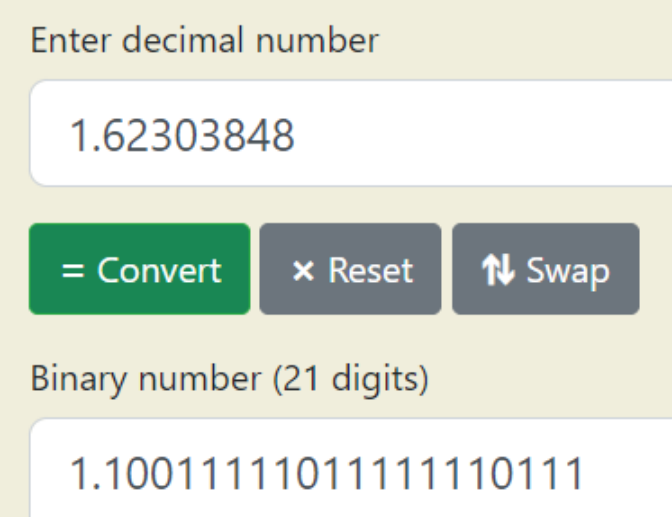


Fig. 12 binary representation of 1.62303848

- e^0.9687 = 2.63451736 (fig 13)

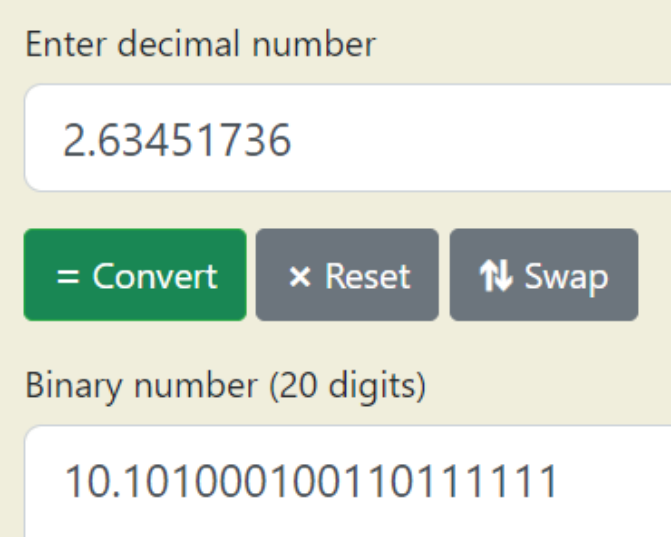


Fig. 13 binary representation of 2.63451736

And the simulation results are:

- e^0.121093 = 00001.001000001110000
- e^0.24184 = 00001.0100011000100001
  (figs 14,15)

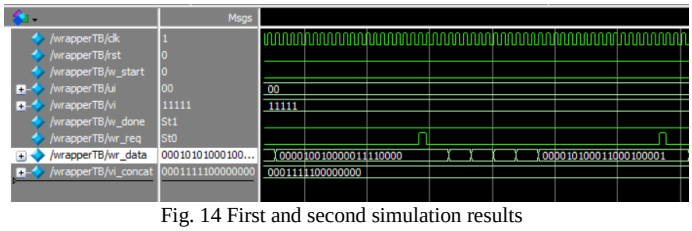


Fig. 14 First and second simulation results

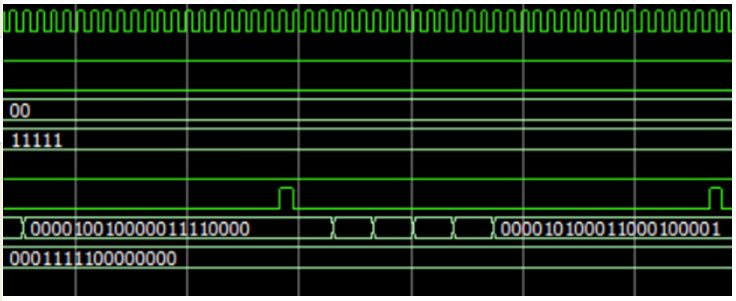


Fig. 14 First and second simulation results (more zoom)

- e^0.4843 = 00001.100111111000000
- e^0.9687 = 00010.1010001001101111
  (figs 15, 16)

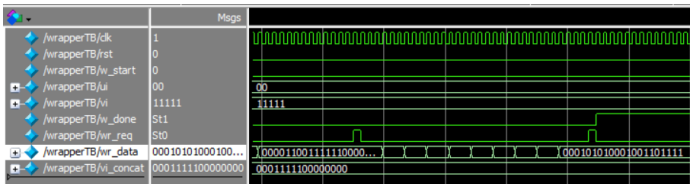


Fig. 15 Third and fourth simulation results

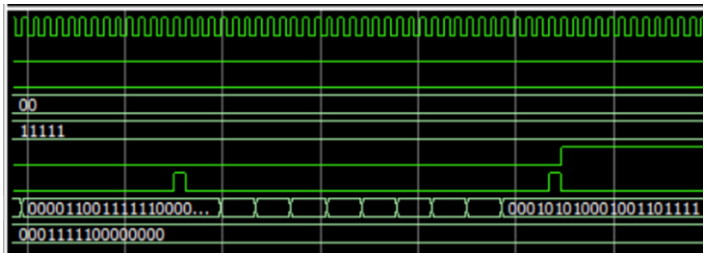


Fig. 16 Third and fourth simulation results (more zoom)

Which are the close enough to the actual values.

For v_i = 10101 and u_i = 00 the expected values and the binary of them are:

- e^0.08203125 = 1.08203125 (fig 17)

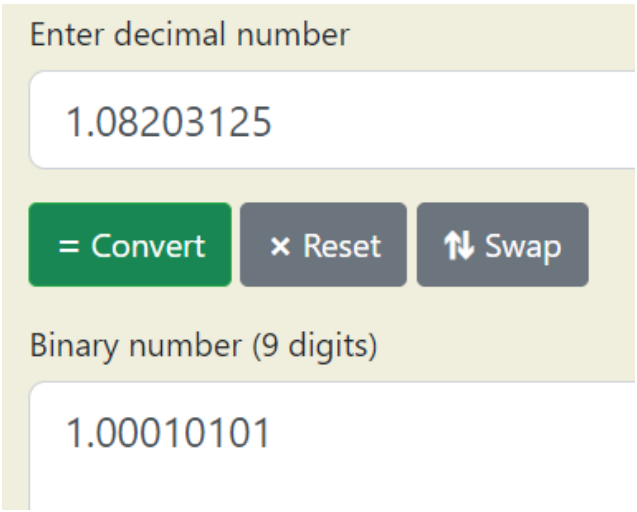


Fig. 17 binary representation of 1.08203125

- e^0.1640625 = 1.17828796 (fig 18)

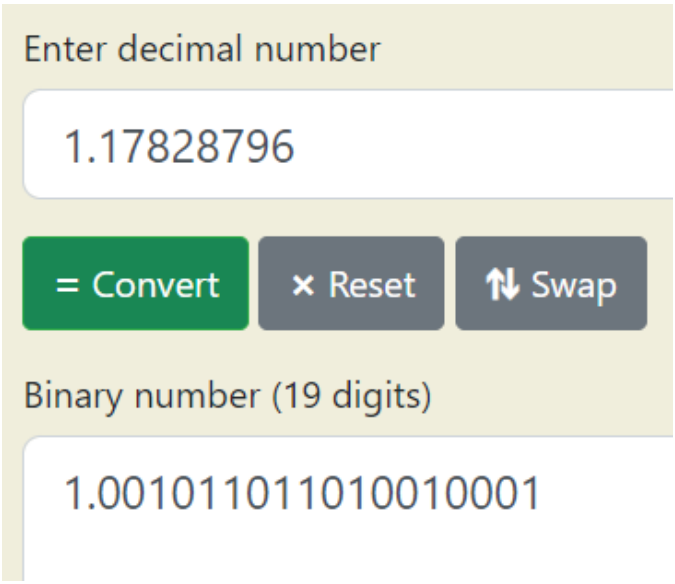


Fig. 18 binary representation of 1.17828796

- e^0.328125 = 1.38836251 (fig 19)

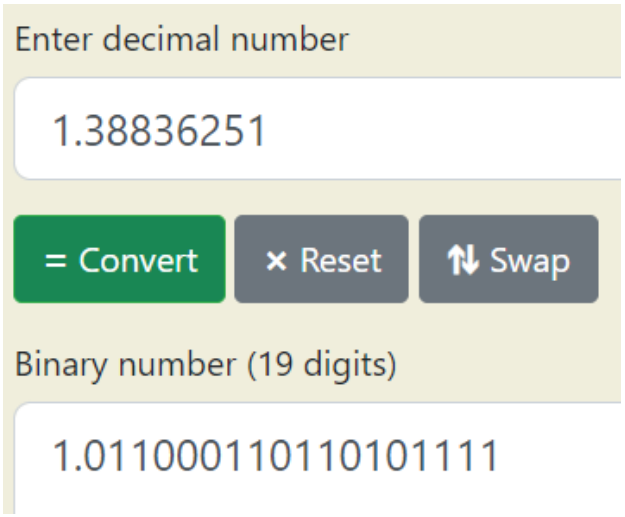


Fig. 19 binary representation of 1.38836251

- e^0.65625 = 1.92755045 (fig 20)

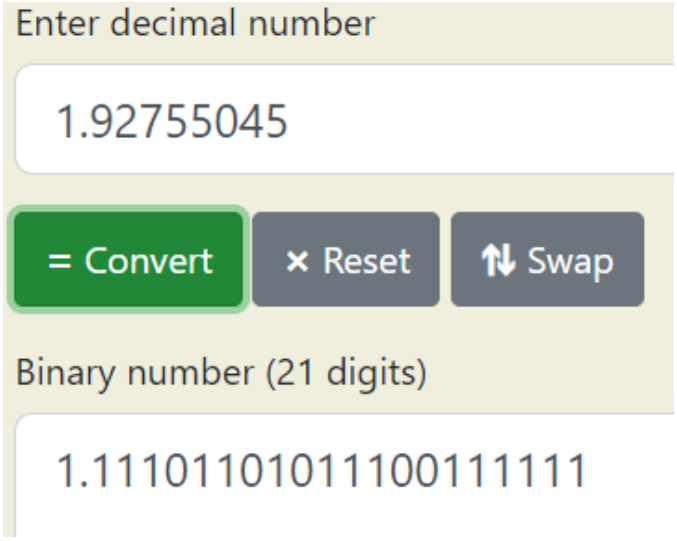


Fig. 20 binary representation of 1.92755045

And the simulation results are:

- e^0. 08203125 = 00001.0001010111011110
- e^0.1640625 = 00001.001011010011110
  (figs 21,22)

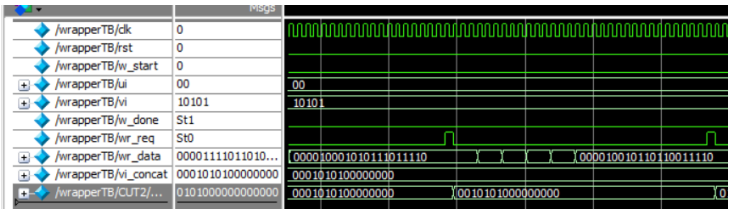


Fig. 21 First and second simulation results

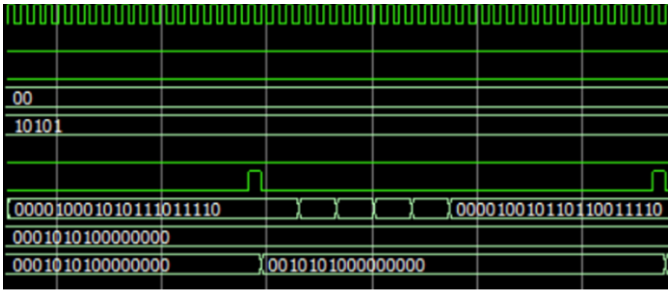


Fig. 22 First and second simulation results (more zoom)

- e^0.328125 = 00001.0110001101100101
- e^0.65625 = 00001.1110110101101101
  (figs 23,24)

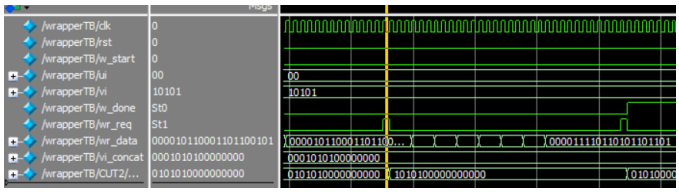


Fig. 23 Third and fourth simulation results

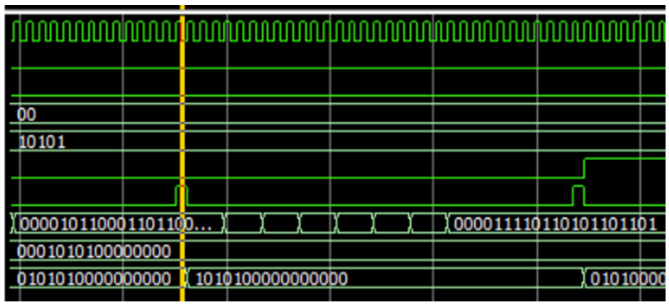


Fig. 24 Third and fourth simulation results (more zoom)

Which are the close enough to the actual values

4. Simulation outcomes for u_i examples

For v_i = 11111 and u_i = 00 the expected values and the binary of them are:

- e^0.121093 = 1.12872988 (fig 10)
- e^0.24184 = 1.2735904 (fig 11)
- e^0.4843 = 1.62303848 (fig 12)
- e^0.9687 = 2.63451736 (fig 13)

And the simulation results are:

- e^0.121093 = 00001.001000001110000
- e^0.24184 = 00001.0100011000100001
  (figs 14,15)
- e^0.4843 = 00001.100111111000000
- e^0.9687 = 00010.1010001001101111
  (figs 15, 16)

Which are the close enough to the actual values.

For v_i = 11111 and u_i = 01 the expected values and the binary of them are:

- 2*e^0.121093 = 2.25745976 (fig 25)

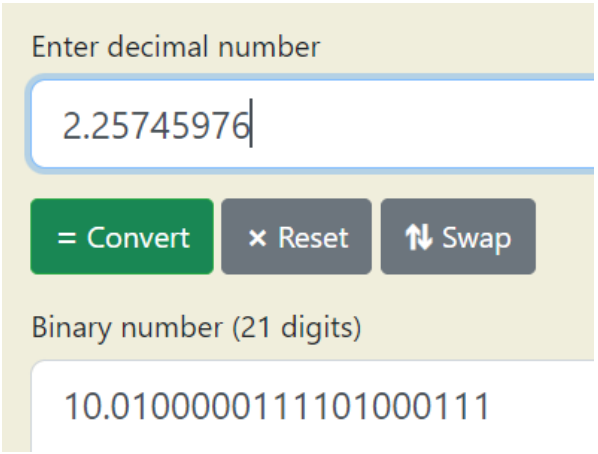


Fig. 25 binary representation of 2.25745976

- 2*e^0.24184 = 2.5471808 (fig 26)

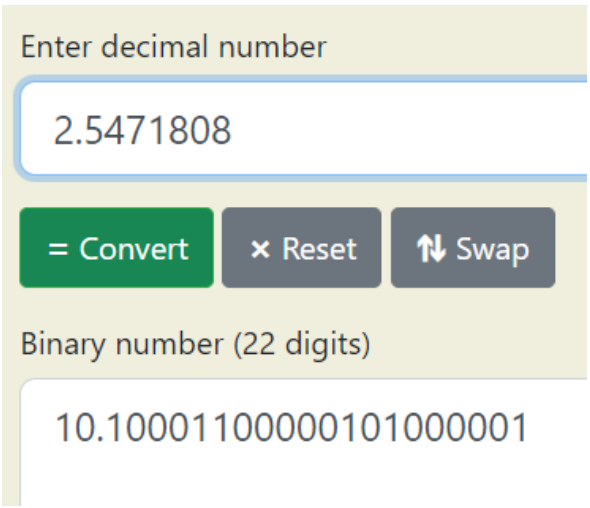


Fig. 26 binary representation of 2.5471808

- 2*e^0.4843 = 3.24607696 (fig 27)

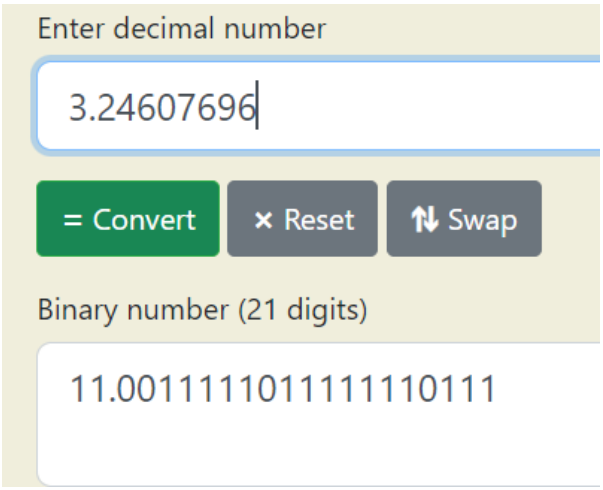


Fig. 27 binary representation of 3.24607696

- 2*e^0.9687 = 5.26903472 (fig 28)

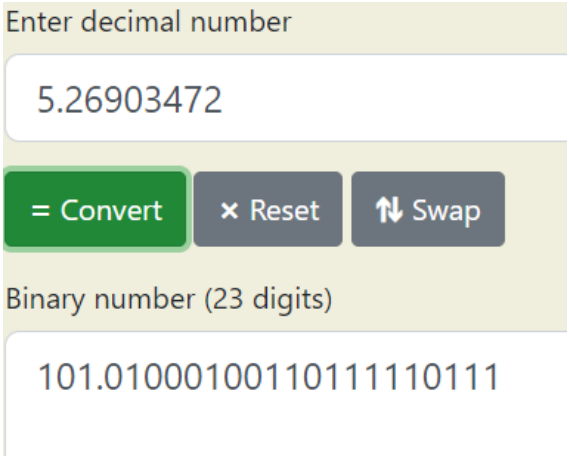


Fig. 28 binary representation of 5.26903472

And the simulation results are:

- 2*e^0.121093 = 00010.010000011100000
- 2*e^0.24184 = 00010.1000110001000010 (Fig 29)

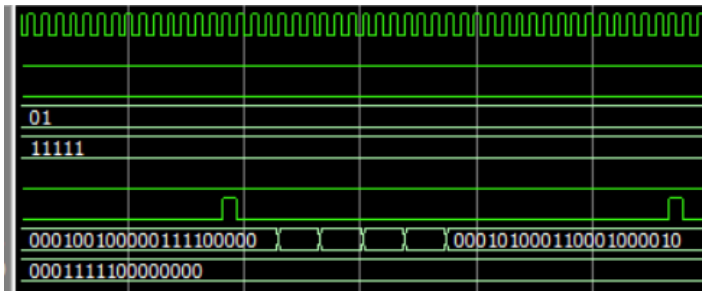


Fig. 29 First and second simulation results

- 2*e^0.4843 = 00011.001111110000000
- 2*e^0.9687 = 00101.0100010011011110 (Fig 30)

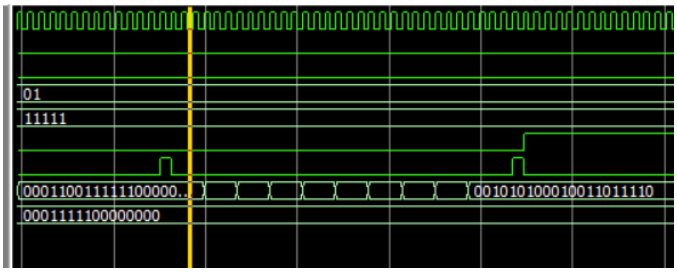


Fig. 30 Third and fourth simulation results

Which are the close enough to the actual values

IV. Exponential Accelerator Wrapper Implementation

1. *FPGA resources*

Report the resources of FPGA for Exponential Accelerator Wrapper is shown in fig 31.

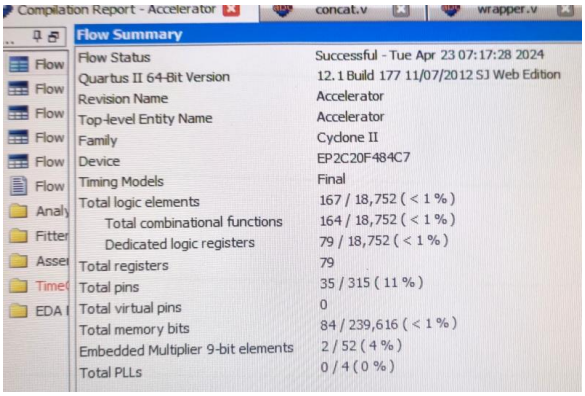


Fig. 30 resources of FPGA for Exponential Accelerator Wrapper

2. *Schematic in Quartus*

The schematic of the design in Quartus is shown in fig 31

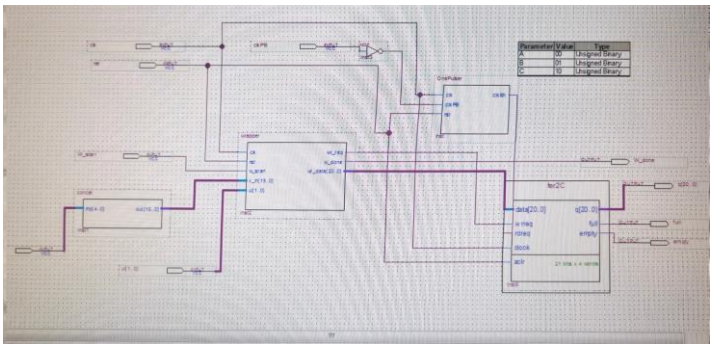


Fig. 31 The schematic of the design in Quartus

3. *FIFO values for vi=11111 and ui=00*

The 10 MSB is shown by red LEDs on the board so for v_i = 11111 and the u_i = 00 the values are (the first value is in binary and the second one is in Decimal) (Note that LEDG1 is the done signal of exponential engine and the LEDG4 is the Full signal of FIFO).:

e^0.121093 = 1.12872988
FPGA first output: 00001.00100 = 1.125
(Fig 32)

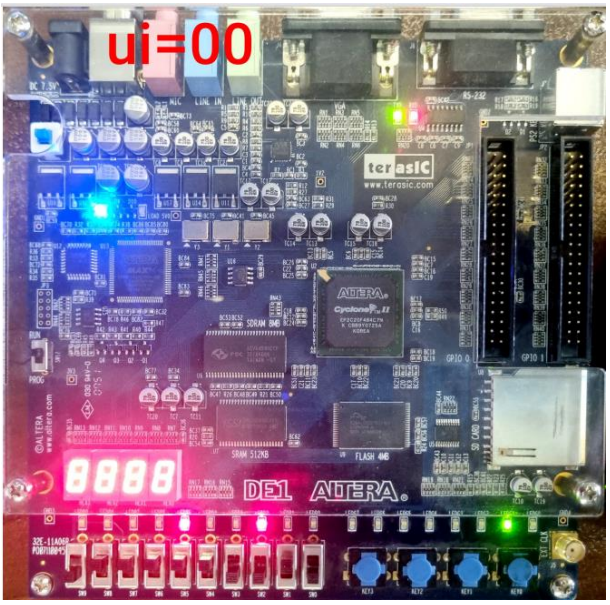


Fig. 32 The first output for vi=11111 and ui=00

e^0.24184 = 1.2735904
FPGA second output: 00001.01000 = 1.25
(Fig 33)

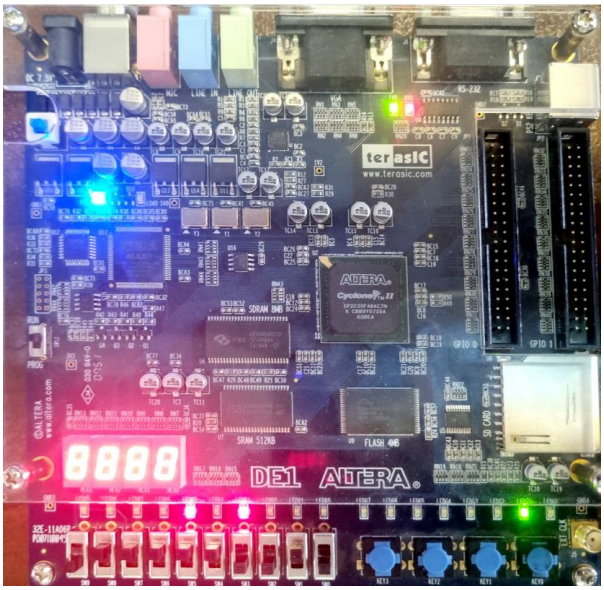


Fig. 33 The second output for vi=11111 and ui=00

e^0.4843 = 1.62303848
FPGA third output: 00001.10011 = 1.59375
(Fig 34)

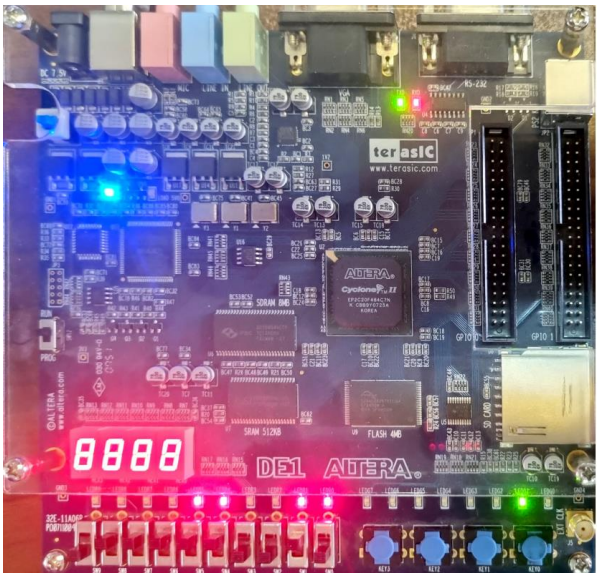


Fig. 34 The third output for vi=11111 and ui=00

e^0.9687 = 2.63451736
FPGA fourth output: 00010.10100 = 2.625
(Fig 35)

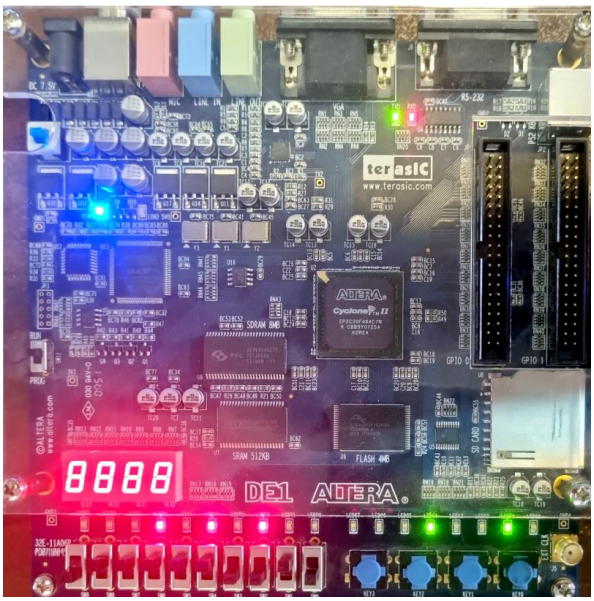


Fig. 35 The fourth output for vi=11111 and ui=00

Which are close to actual values considering we are just using the 10 MSB of the output

4. FIFO values for vi=11111 and ui=01

The 10 MSB is shown by red LEDs on the board so for v_i = 11111 and the u_i = 01 the values are (the first value is in binary and the second one is in Decimal) (Note that LEDG1 is the done signal of exponential engine and the LEDG4 is the Full signal of FIFO).:

2*e^0.121093 = 2.25745976
FPGA first output: 00010.01000 = 2.25
(Fig 36)

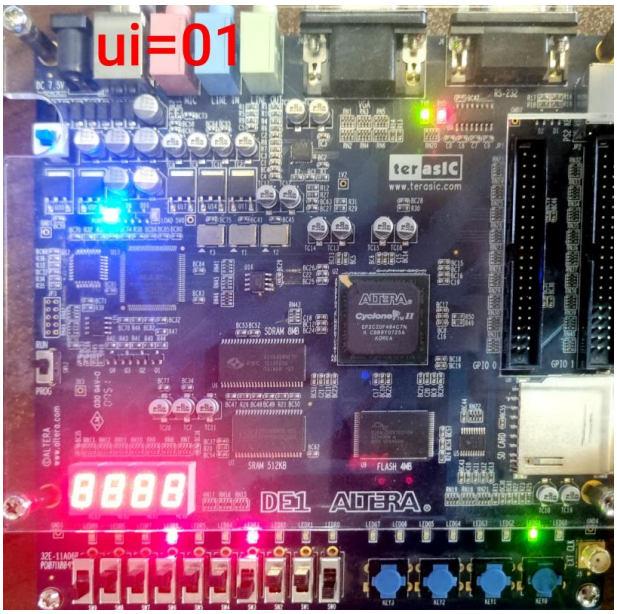


Fig. 36 The first output for vi=11111 and ui=01

2*e^0.24184 = 2.5471808
FPGA second output: 00010.10001 = 2.53125
(Fig 37)

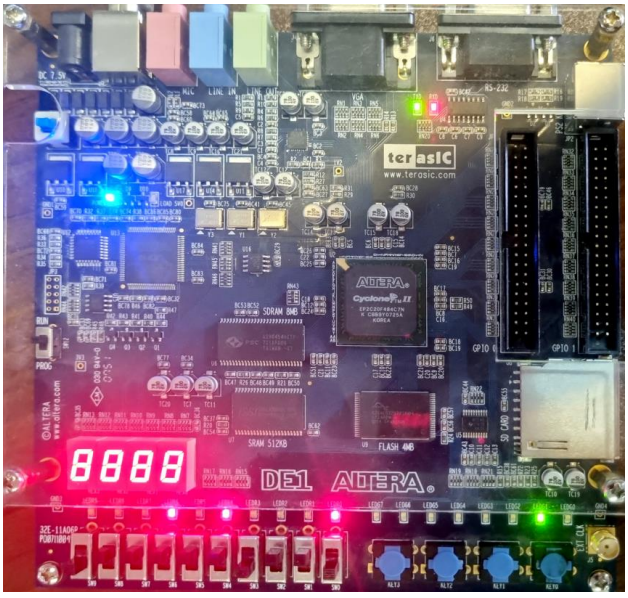


Fig. 37 The second output for vi=11111 and ui=01

2*e^0.4843 = 3.24607696
FPGA third output: 00011.00111 = 3.21875
(Fig 38)

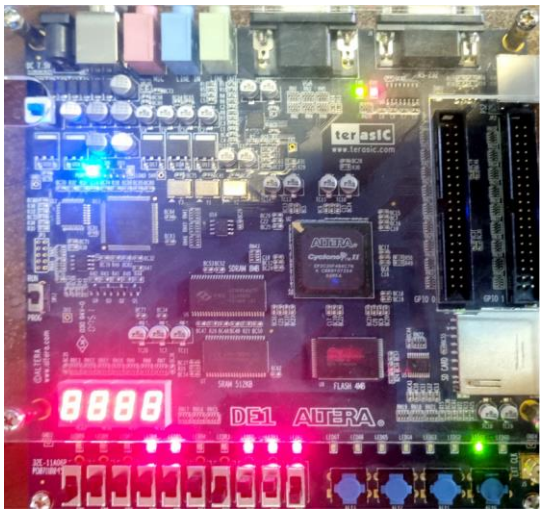


Fig. 38 The third output for vi=11111 and ui=01

2*e^0.9687 = 5.26903472
FPGA fourth output: 00101.01000 = 5.25
(Fig 39)

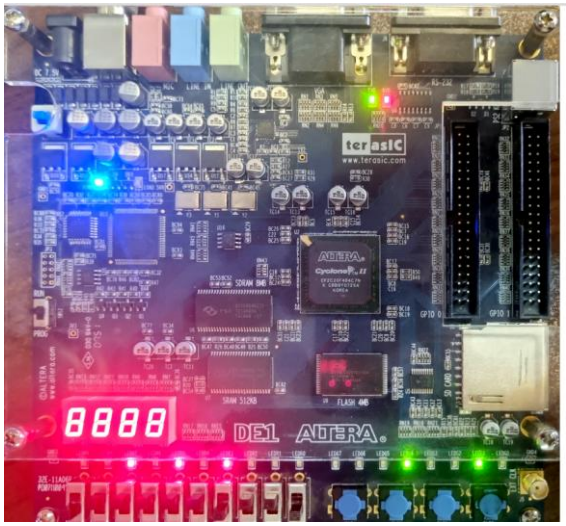


Fig. 39 The fourth output for vi=11111 and ui=01

Which are close to actual values considering we are just using the 10 MSB of the output.

5. Differences

As we explained before, the goal of this structure is to calculate $e^{v_i} << u_i$. So in the first example u_i is 0 so we are

calculating e^{v_i}. In the second example u_i is 1 so we are calculating $e^{v_i} << 1$ so as you can see on the FPGA the calculated values are shifted once from the last part results. In other word the results in second example are the results of the first example shifted once.

V. Conclusions

In this experiment we learned the usage of wrapper and the reason of using accelerator which is to prevent the processor to do some tasks and do other things. We also learned how to implement the accelerator and we test our implementation on FPGA and validate our outputs.

References

[1] K. Basharkhah “Experiment 4 Digital Logic Laboratory,” under supervision of Professor Z. Navabi, University of Tehran, Spring 1403