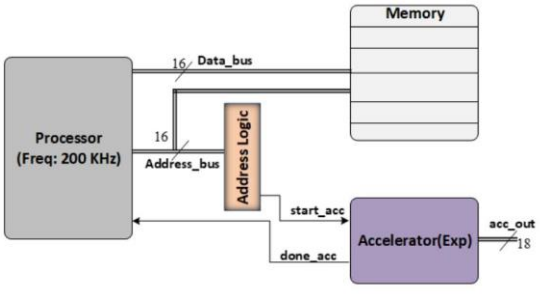
DLD LAB – Experiment #4 - Accelerator and Wrappers

Ali Ghahari 810100201 Mohammad Sadeghi 810100175

Introduction

System on Chip is an integrated circuit that integrates multiple components. The main core of a SoC is a processor that handles different computational tasks. In addition to the processor, the system includes memory, Input/output ports, and accelerators. Accelerators are dedicated computation units that usually execute one specific task. This single task needs a smaller and less complicated datapath which leads to a high frequency. However, CPUs in which millions of operations must be executed within a fixed time interval have low frequency of operation. To increase the speed of a SOC, hardware accelerators are usually embedded in the system. The processor will dispute some of its tasks to the hardware accelerator. During this time, the accelerator performs several of the same or different operations and stores 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 into a SOC

Figure 1: Block diagram of a typical integrated circuit

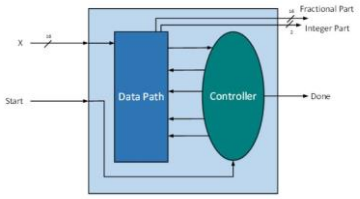


When the CPU needs to compute an exponential value, because of the higher estimation speed of the accelerator it asks the exponential hardware accelerator to complete this task. In this way, the CPU can complete other software tasks in parallel. Before starting the computation, the CPU should send a data from memory to the accelerator. This data will be stored in a buffer inside the accelerator. When transferring is finished, the 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, the accelerator will have its *start* signal issued when needed.

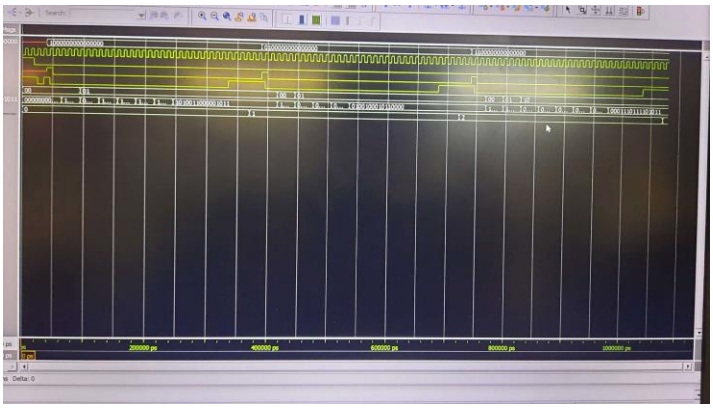
Exponential Engine

The accelerator that we are going to use is an exponential circuit. As Figure 2 shows, this module receives a 16-bit input *x* and generates a 16-bit output *Fractionalpart* and 2-bit *Integerpart*. Remember that the x value fits between zero and one. 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 aware it.

Figure 2: Block diagram of exponential accelerator

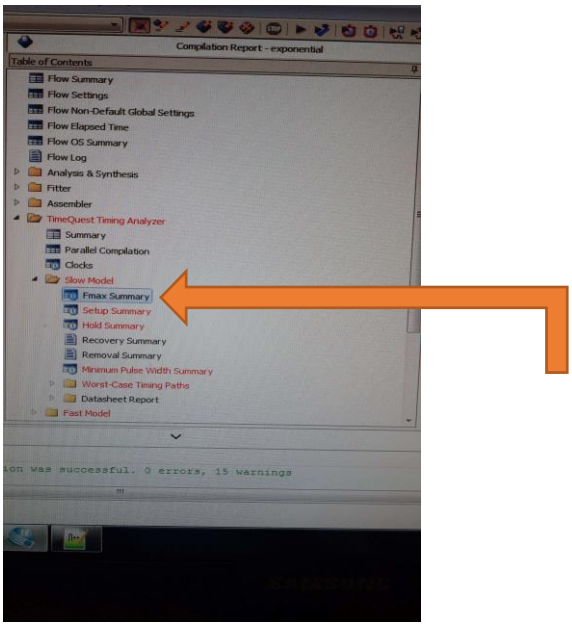


For clock generation for this module we need to be aware of the maximum frequency of this accelerator. First we running Modelsim simulation of module, write a testbench for this design with three different values for input *x*.

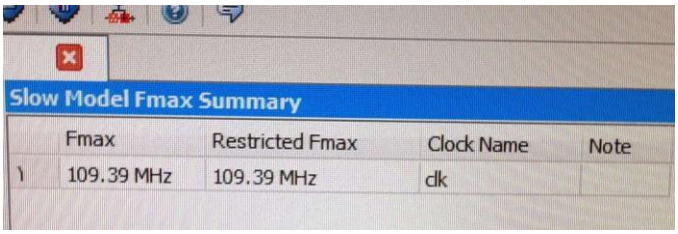


As you can see we have three value for x and in three times done signal become 1.

In next step we Synthesize this design in Quartus II Software. After synthesizing the design, we can find out the maximum frequency of this accelerator by referring to the Timing Analyzer reports in the Quartus synthesis tool:



As in this part we can find maximum frequency of accelerator:



Slow Model Fmax Summary

	Fmax	Restricted Fmax	Clock Name	Note
1	109.39 MHz	109.39 MHz	clk	

Exponential Accelerator Wrapper

One of the applications that makes use of such multi-value exponential calculation is an activation function in Deep Neural Networks (DNN).

$$f(x_i) = e^{x_i} \{i = 1, 2, \dots, N\}$$

Multiple of these exponential values can be calculated with one accelerator instead of using one accelerator unit for each x_i. To reduce the required hardware resources for the softmax exponential function, a mathematical transformation would be useful. Each input value is split into an integer number z_i and a fractional number v_i as below:

$$x_i = z_i + v_i$$

$$e^{x_i} = e^{z_i} * e^{v_i}$$

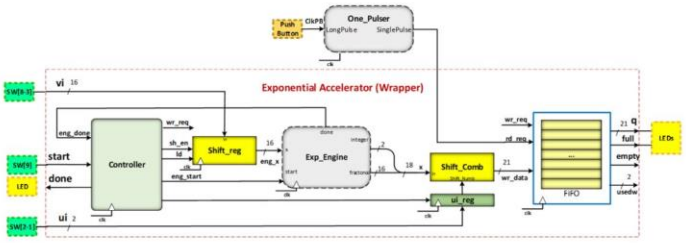
The second segment of this equation can be easily implemented with the exponential engine. To reduce the complexity of the hardware implementation of this equation, we use the base number 2 to take place of the base number e and the exponential function will be changed to:

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

$$e^{x_i} = e^{v_i} \ll u_i$$

As can be seen, this equation can be implemented with a shifter that shifts e^(v_i) for u_i times. We are going to apply this transformation to the exponential in a wrapper around the exponential engine.Some other tasks are also included in the wrapper that are explained below:

Figure 4: Wrapper for exponential accelerator



- Referring to Figure 4, the wrapper receives single input in the form of a fractional value v_i , and an integer value u_i and a *start* signal from the processor.

- Considering the input values are within u_i and u_i+1 we can calculate n number of exponential in this range as follows:

$$e^{x_i} = \begin{cases} e^{v_i} \ll u_i \\ e^{2*v_i} \ll u_i \quad if\ v_i < 1/2^{n-1} \\ \dots \\ e^{2^{n-1}*v_i} \ll u_i \quad if\ u_i < x_i < u_i + 1 \end{cases}$$

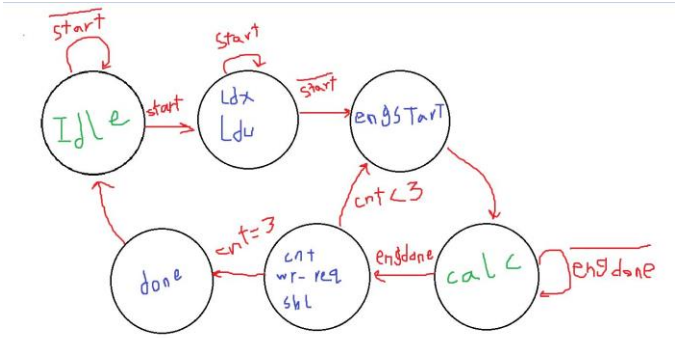
In this experiment n=4. Based on these assumptions, 4 exponential values can be calculated. Four different values can be generated with the shift register unit by first registering the value of v_i and then shifting its value one bit to the left for each exponential calculation.

- The controller is responsible for generating the *load* and *shift enable* signals for the shift register, the *start* signal for the exponential engine, and the *load* signal for the ui-register. The exponential engine should start each calculation when the previous one is completely done. For this purpose *engdone* is fed to the controller and when done is asserted the controller generates a complete pulse on *engstart*. At the same time, the correct value of x should appear on the corresponding input of the exponential engine. For each exponential value estimation, the controller issues the *wrreq* signal for writing data to the FIFO. When all calculations are finished the controller sends a *done* signal on the wrapper output.

- For shifting the output of the exponential engine, a combinational shifter is required. The input of this shifter is the u_i value that is provided outside the wrapper. To store the value of u_i , a register called ui-reg is used.

- When an exponential value is calculated then it should be stored in a FIFO so that when the CPU finishes its work it can retrieve all the results. As shown in the Figure 4, the FIFO has a *writereq* input and a *write data* for writing into the buffer and a *readreq* for reading the outputs. Since we are limited to four calculations in this experiment the FIFO size would be four as well.

Here is state diagram for module controller and Huffman code for its verilog:




```
always @(*) begin
    ns = 3'b0;
    case (ps)
        IDLE: ns = start ? WAIT : IDLE;
        WAIT: ns = start ? WAIT : START;
        START: ns = CALC;
        CALC: ns = eng_done ? WRITE : CALC;
        WRITE: ns = count < 2'b11 ? START : DONE;
        DONE: ns = IDLE;
        default: ns = IDLE;
    endcase
end

always @(posedge clk, posedge rst) begin
    if(rst) count = 2'b0;
    else if(cnt) count = count + 1;
end

assign ldu = ps == WAIT;
assign ldx = ps == WAIT;
assign wr_req = ps == WRITE;
assign shl = ps == WRITE;
assign eng_start = ps == START;
assign done = ps == DONE;
assign cnt = ps == WRITE;
endmodule
```

In the next page you can see Verilog code for wrapper shown in figure 4 includes 5 component:

1-controller 2-combinational shift

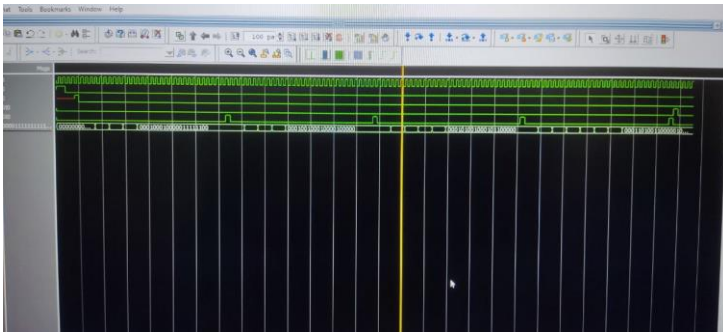
3-Ui register 4-exponential unit 5-shift register

```
module Wrapper(clk, rst, start, v, u, done, wr_req, wr_data);
input clk, rst, start;
input[4:0] v;
input[1:0] u;
output done, wr_req;
output[20:0] wr_data;

wire ldx, ldu, shl, eng_start, eng_done;
wire[15:0] shreg_out, frac;
wire[1:0] int, ureg_out;

Controller controller(.clk(clk), .rst(rst), .wr_req(wr_req), .ldx(ldx), .ldu(ldu), .shl(shl), .eng_start(eng_start), .done(done),
ShiftReg sh_reg(.clk(clk), .rst(rst), .ld(ldx), .sh(shl), .data({3'b0, v, 8'b0}), .out(shreg_out));
Reg u_reg(.clk(clk), .rst(rst), .ld(ldu), .data(u), .out(ureg_out));
exponential exp(.clk(clk), .rst(rst), .start(eng_start), ._x(shreg_out), .done(eng_done), .intpart(int), .fracpart(frac));
CombShift comb_shift(.frac(frac), .int(int), .sh(ureg_out), .out(wr_data));
endmodule
```

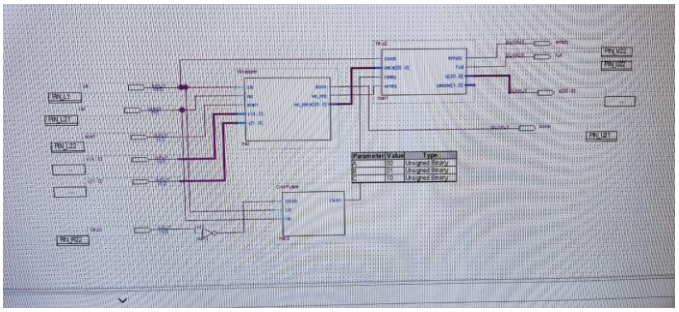
And this is our testbench result:



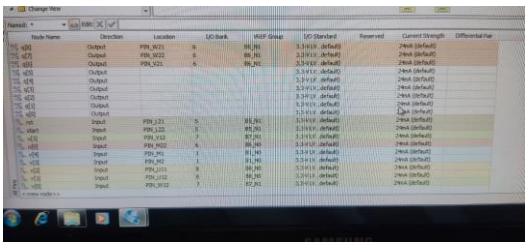
FPGA design

At last part we simulate this module with same input as Modelsim and see result with LED on board

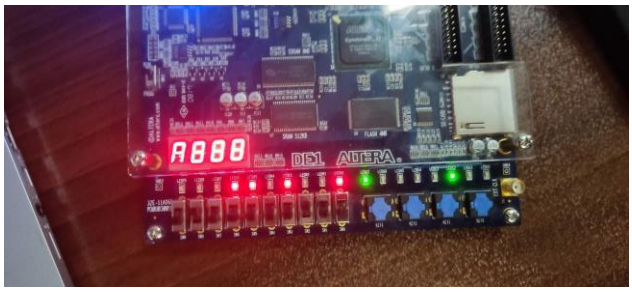
First we build symbol of each component of wrapper and connect internal bus together:

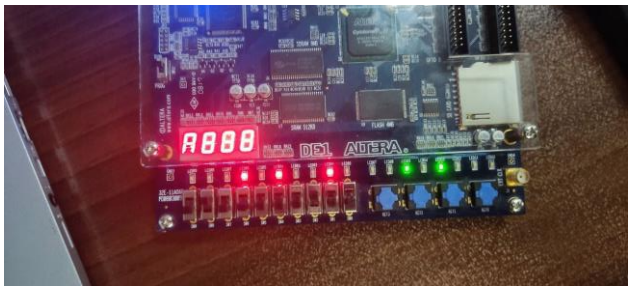


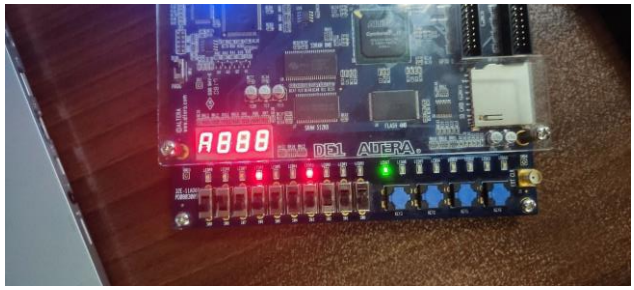
And in next part we assign switch pins to our inputs and LED pins to done and FIFO full signal and other remained LEDs to MSB of our output:

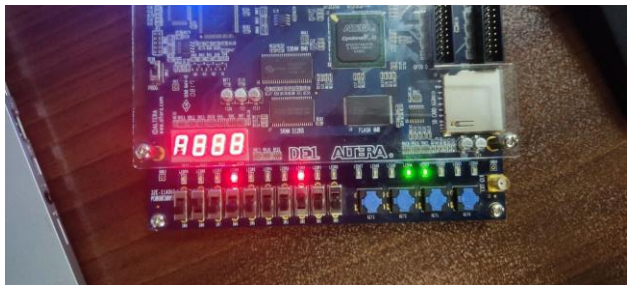


And you can see our output on board LED is same as wave we got in Modelsim:









And LEDs for stack *full* and *done* signal be 1: