CPRE 465 FINAL PROJECT

Harith Mohd., Akash Pradhan, Joyal Babu

Akash Pradhan – Streamlining synthesis process

M. Harith Arsyad – Verilog Code, Optimizations, and Synthesis

Joyal Babu – Report writing

5th December 2022

1. Problem Description

The problem that the USDA is facing is that they want to investigate the relationship between the temperature of a plant and the protein level present in the plant. In order to do this, they need to monitor the temperature of the plants remotely, using temperature sensors and IoT motes. Our task is to design a circuit that can solve this problem for the USDA, allowing them to monitor the temperature of the plants and investigate the relationship between temperature and protein levels. This will require a combination of digital circuit design and algorithm design in order to implement a solution that meets the USDA's requirements. We will

1. Start by using an IoT mote to collect temperature readings from the temperature sensor. The mote should be able to communicate with the temperature sensor and collect the readings at regular intervals.

2. Next, design a circuit that can receive the temperature readings from the mote and store them in a memory device, such as a shift register or a FIFO buffer. The circuit should be able to store a large number of readings, so that we can calculate the average and standard deviation over a long period of time.

3. Once the readings are stored in the memory device, we can use a digital circuit to calculate the average and standard deviation of the readings. This can be done by implementing a simple algorithm that adds up all the readings and divides the result by the number of readings to calculate the average. The standard deviation can be calculated by taking the square root of the sum of the squared differences between each reading and the average, divided by the number of readings.

4. Finally, the circuit should be able to display the average and standard deviation of the temperature readings on a display device, such as an LCD screen. This will allow the USDA to monitor the temperature of the plants and investigate the relationship between temperature and protein levels.

In detail regarding the design of the digital circuit the objective is to find the moving average and the moving standard deviation of the last fourteen readings from a temperature sensor.

Moving average is a calculation to analyze the data points by creating a series of averages of different subsets of the full data set. Here we are taking the temperature value the sensor gives as the average temperatures the sensor detects and we are calculating the average temperature of the last 14 inputs from the temperature sensor. The temperature average is given by

$$T_{avg} = \frac{1}{n-1} \sum_{i=1}^{n} T_i$$

The standard deviation can be written as

$$\sigma = \sqrt{\frac{1}{n-1} \sum_{i=1}^{n} (T_i - T_{avg})^2} = \sqrt{\frac{1}{n-1} \left(\sum_{i=1}^{n} T_i^2 \right) - T_{avg}^2}$$

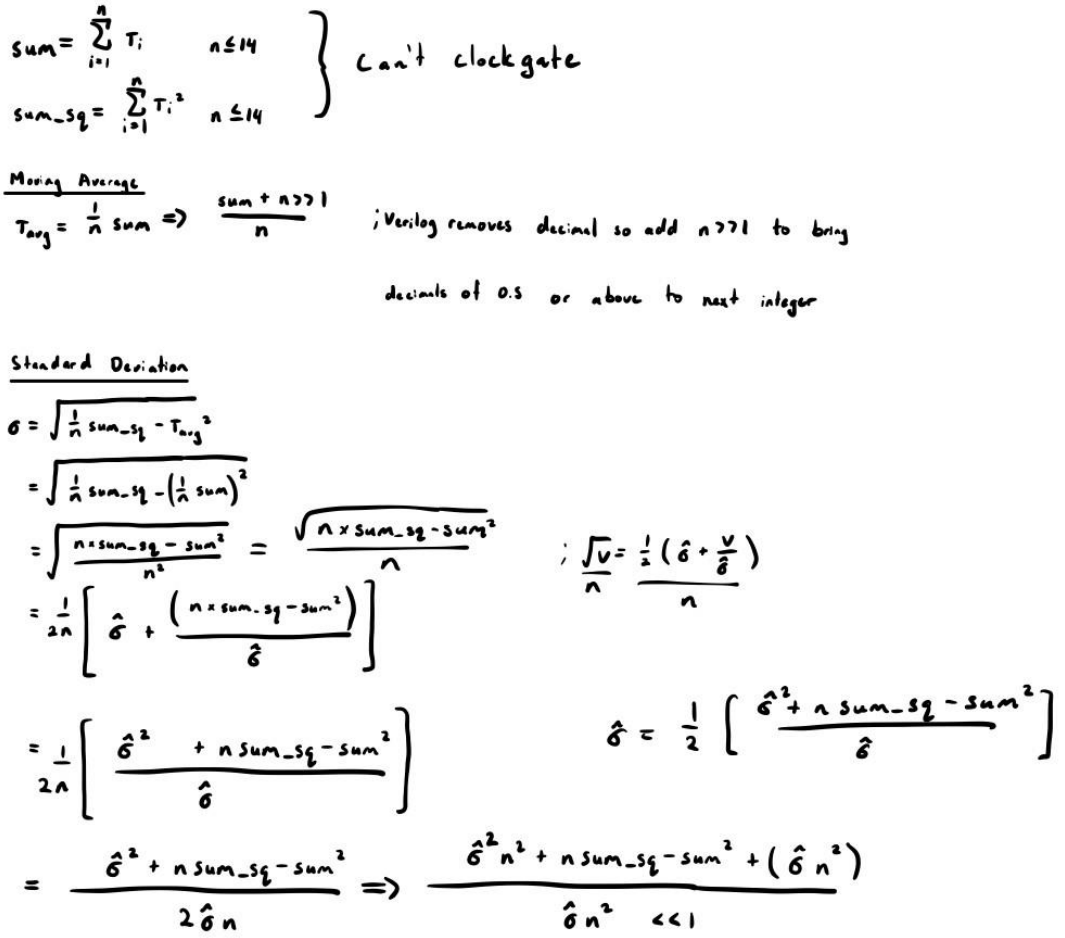
hence since we want the moving average for the last 14 temperature readings if n,14 we take the average and standard deviation of all n readings.

The design specification given are

When RESET is set to 1 all previous temperature readings are forgotten. The temperatures are taken one at a time and the circuit takes in new temperature when the SAMPLE is set to 1. The circuit then takes in one new temperature reading TN and the mode of operation MODE at the

next clock edge. The next few clock cycles are used to calculate the moving average and the moving standard deviation. After the that the circuit will change DONE to 1 and output will be given which can the moving average(if MODE =0) or the moving standard deviation (if MODE =1) given as AVG/SD.

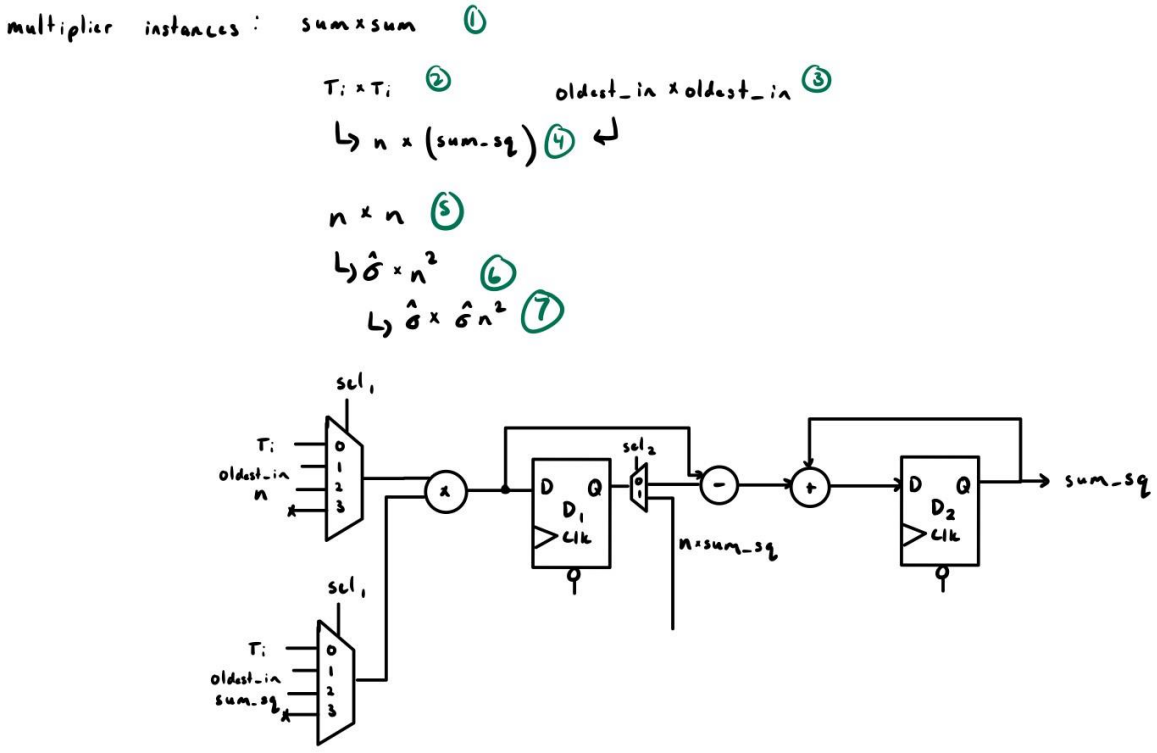
Square root is not easy to be implemented via hardware hence we use the Babylonian method.



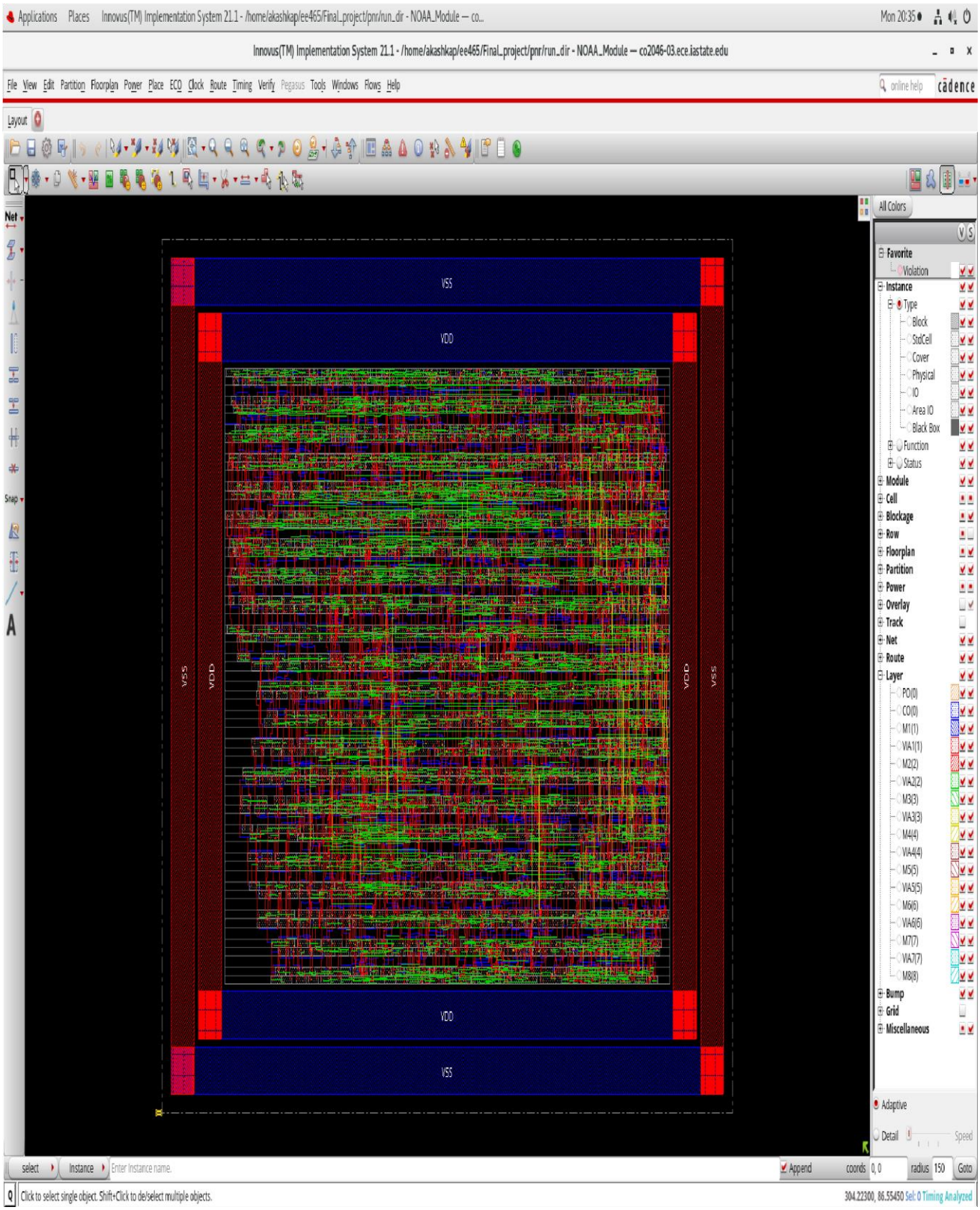
2. Brief Description of Team Approach to this Problem

In this project we needed to preserve previous values of temperatures and their running averages. We started with a big memory of registers for storing input temperatures and used accumulator for calculating the sum of temperatures and square of temperatures. These used a lot of adders and multipliers. Then we did division to get the averages and square root using the Netwon-Raphson approximation method. This gave us uneven number of cycles for

calculation of running averages and standard deviation. The problem was that we were doing multiple things in sequential manner without maintaining proper cycle accuracy and that is when we changed our design. Next we started with scratch design on paper and made a flow diagram where we found what could be optimized in the process to calculate the required things. We were able to save on multiplication and division hardware by proper insertion of buffers and using the techniques learned in previous labs. We had the following diagrams in our rough sketches which shows our design



Schematics and layouts after optimization :


Applications Places Innovus(TM) Implementation System 21.1 - /home/akashkap/ee465/Final_project/pnr/run_dir - NOAA_Module – co...
Mon 20:35
Innovus(TM) Implementation System 21.1 - /home/akashkap/ee465/Final_project/pnr/run_dir - NOAA_Module – co2046-03.ece.iastate.edu
File View Edit Partition Floorplan Power Place ECO Clock Route Timing Verify Pegasus Tools Windows Flows Help
online help
cadence
Layout
Net
Snap
VSS
VDD
VSS
VDD
VDD
VSS
VDD
VSS
All Colors
Favorite
Violation
Instance
Type
Block
StdCell
Cover
Physical
IO
Area IO
Black Box
Function
Status
Module
Cell
Blockage
Row
Floorplan
Partition
Power
Overlay
Track
Net
Route
Layer
PO(0)
CO(0)
M1(1)
VIA1(1)
M2(2)
VIA2(2)
M3(3)
VIA3(3)
M4(4)
VIA4(4)
M5(5)
VIA5(5)
M6(6)
VIA6(6)
M7(7)
VIA7(7)
M8(8)
Bump
Grid
Miscellaneous
Adaptive
Detail
Speed
select
Instance
Enter Instance name.
Append
coords 0, 0
radius 150
Goto
Click to select single object. Shift+Click to de/select multiple objects.
304.22300, 86.55450 Sel: 0 Timing Analyzed

Timing, Area and Power Reports (by RTL Compiler and Innovus)

	Before Optimization	After Optimization
Clock Period(ns)	4.2	4.2
Slack(ns)	-3.923	0
Area(μm^2)	62051.760	11157.48
Average power consumption(mW)	12.761	2.372

	Before optimization	After optimization
Clock period(ns)	4.2	4.2
Slack(ns)	-3.923	0
Area(μm^2)	62051.760	11157.48
Average power consumption(mW)	4.30942	2.372
Average Energy consumption per temperature reading(W)	33.28*10^-12	4.9831*10^-12
Latency	4	4
Throughput	One every 8.12ns, 0.123*10^9	0.476*10^9, one every 2.1 ns

4. Verilog Codes of design

This is the final code after optimization:

```
module NOAA_Module (
input CLK,
input RESET,
input MODE,
input [11:0] TN,
output SAMPLE, output
DONE, output reg [11:0]
AVG_SD);
```

//Flag wires and regs

wire newData_flag1; //flags for every new data into the process. Waits for the process to be done first then flags for the next set to be processed

reg newData_flag2, newData_flag3, newData_flag4; //registers to delay newData flag to 2, 3, ... cycles

//Clock Components

wire newData_clk1, newData_clk2, newData_clk3; //clocks for every new input to the process. Each clock line adds 1 cycle delay after every new input wire t_avg_clk2; //clock-gating clock for MOVING AVERAGE. 2 cycles after new input to process

wire sd_clk1, sd_clk2, sd_clk3, sd_clk4; //clock-gating clock for STANDARD DEVIATION. 1, 2, 3, ... cycle delay wire mult_clk;

//Multiplier Wires and select wire [31:0]

mult1_w1, mult1_w2; wire [31:0]

mult2_w1, mult2_w2; wire [39:0]

mult1_w3, mult2_w3, mult3_w3;

reg [1:0] mult_sel;

//Division Wires wire [31:0]

div_w1, div_w2; wire

[31:0] div_result;

//Continuos Process Components

reg [15:0] sum; reg [31:0]

sum_sq; reg [31:0] Ti_sq; reg

[31:0] OD_sq; wire [11:0]

oldest_data; reg mode_delay,

```
mode_delay2; wire [3:0]
counter_n;

//MOVING AVERAGE Components
wire [11:0] t_avg;

//STANDARD DEVIATION Components
reg [31:0] sd_n_sq, sd_n, sum_2;
reg [39:0] sum_sq_n; wire [15:0]
sd_result;
reg [15:0] sd_temp1;

shift_reg x1(oldest_data, counter_n, newData_flag1, CLK, RESET, TN);

//Processes running every clock cycle
always @(posedge CLK, posedge RESET)
begin
  if (RESET) begin     mode_delay
<= 0;     mode_delay2 <= 0;   end
else begin     mode_delay <=
MODE;     mode_delay2 <=
mode_delay;   end
end

//Processes that need to run every time there is a new input (1st cycle)
//ie. sum(TI), TI^2, OD^2 always @(posedge
newData_clk1, posedge RESET) begin
```

```
 if (RESET) begin
sum <= 0;
   Ti_sq <= 0;    OD_sq <= 0;   end
else begin    sum <= sum + TN -
oldest_data;    Ti_sq <=
mult1_w3;    OD_sq <=
mult2_w3;   end
end

//Processes that need to run every time there is a new input (2nd cycle)
//ie. sum_sq always @(posedge newData_clk2, posedge
RESET) begin
 if (RESET)
sum_sq <= 0;
 else
   sum_sq <= sum_sq + Ti_sq - OD_sq;
end

//MOVING AVERAGE
assign t_avg = (sum+(counter_n>>1))/counter_n;

//STANDARD DEVIATION //2nd Cycle
always @(posedge sd_clk2, posedge
RESET) begin
 if (RESET) begin
sum_2 <= 0;    sd_n <=
0;   end   else begin
```

```
sum_2 <= mult1_w3;
sd_n <= mult2_w3;
end
end

//3rd Cycle always @(posedge sd_clk3,
posedge RESET) begin
 if (RESET) begin
sd_n_sq <= 0;
   sum_sq_n <= 0;   end
else begin     sd_n_sq <=
mult1_w3;     sum_sq_n <=
mult2_w3;   end
end

//4th Cycle always @(posedge sd_clk4,
posedge RESET) begin
 if (RESET) begin
sd_temp1 <= 1024;   end
else begin     sd_temp1
<= sd_result;   end
end
//Final Result assign sd_result = (sd_n_sq + sum_sq_n - sum_2 +
sd_n)/(sd_n<<1);

//Mupltipier Select Logic always @(posedge
mult_clk, posedge RESET) begin
```

```
 if (RESET) begin
mult_sel <= 0;   end
else begin
   if (mult_sel == 2)
mult_sel <= 0;
   else
     mult_sel <= mult_sel + 1;

 end
end

assign mult_clk = newData_clk1 | newData_clk2 | newData_clk3;


always @(posedge DONE, posedge RESET) begin
 if (RESET)
AVG_SD <= 0;
 else
   AVG_SD <= div_result;
end
//Delay new input flag always
@(negedge CLK, posedge RESET) begin
 if (RESET) begin     newData_flag2
<= 0;     newData_flag3 <= 0;
newData_flag4 <= 0;   end else
begin     newData_flag2 <=
newData_flag1;     newData_flag3
```

```
<= newData_flag2;

newData_flag4 <= newData_flag3;

end

end


//DONE and SAMPLE logic assign

DONE = t_avg_clk2 | sd_clk4;

assign SAMPLE = newData_clk1;


//Multiplier 1 logic

assign mult1_w1 = (mult_sel == 0) ? TN : ( (mult_sel == 1) ? sum : ((mult_sel == 2) ? sd_temp1 :
0)); assign mult1_w2 = (mult_sel == 0) ? TN : ( (mult_sel == 1) ? sum : ((mult_sel == 2) ? sd_n :

0));

assign mult1_w3 = mult1_w1 * mult1_w2;


//Multiplier 2 logic

assign mult2_w1 = (mult_sel == 0) ? oldest_data : ( (mult_sel == 1) ? sd_temp1 : ((mult_sel == 2)
? counter_n : 0));

assign mult2_w2 = (mult_sel == 0) ? oldest_data : ( (mult_sel == 1) ? mult3_w3 : ((mult_sel ==
2)
? sum_sq : 0));

assign mult2_w3 = mult2_w1 * mult2_w2;


//Multiplier 3 logic

assign mult3_w3 = counter_n * counter_n;
```

```
//Division Logic assign div_w1 = mode_delay ? (sd_n_sq + sum_sq_n - sum_2 + sd_n) :
(sum+(counter_n>>1)); assign div_w2 = mode_delay ? (sd_n<<1) : counter_n; assign
div_result = div_w1/div_w2;

//create clock at every new process delayed by a cycle each assign
newData_clk1 = CLK & newData_flag1; //At cycle 1 of new data
assign newData_clk2 = CLK & newData_flag2; //At cycle 2 of new
data assign newData_clk3 = CLK & newData_flag3; //At cycle 3 of
new data assign newData_clk4 = CLK & newData_flag4; //At cycle 3
of new data

//clock-gating clock for MOVING AVERAGE. n cycle delay after new input to process
assign t_avg_clk2 = newData_clk2 & ~MODE; //at cycle 2

//clock-gating clock for STANDARD DEVIATION. n cycle delay after new input to
process assign sd_clk1 = newData_clk1 & MODE; //at cycle 1 assign sd_clk2 =
newData_clk2 & MODE; //at cycle 2 assign sd_clk3 = newData_clk3 & MODE; //at
cycle 3 assign sd_clk4 = newData_clk4 & mode_delay2; //at cycle 4

endmodule



// This is linear queue / FIFO
// The queue length 14 // The
data width is also 12 bits
module shift_reg( output [11:0]
        DATAOUT, output reg [3:0]
```

```
        counter_n, output reg
        newData_flag1,
        input clock,
        input reset,
        input [11:0] DATAIN);

 wire newData_clk1;   reg [11:0] memory [13:0]; // the stack is 12 bit wide
and 14 locations in size

 integer i;

 always @(posedge newData_clk1, posedge reset)
 begin
  if (reset)
begin
     for (i = 0; i < 14; i = i + 1) begin
                memory[i] <= 0;
        end
        counter_n <= 0;
   end
else
begin
     memory[0] <= DATAIN;
     for (i = 0; i < 13; i = i + 1) begin
                memory[i + 1] <= memory [i];
        end
        if (counter_n < 14)
```

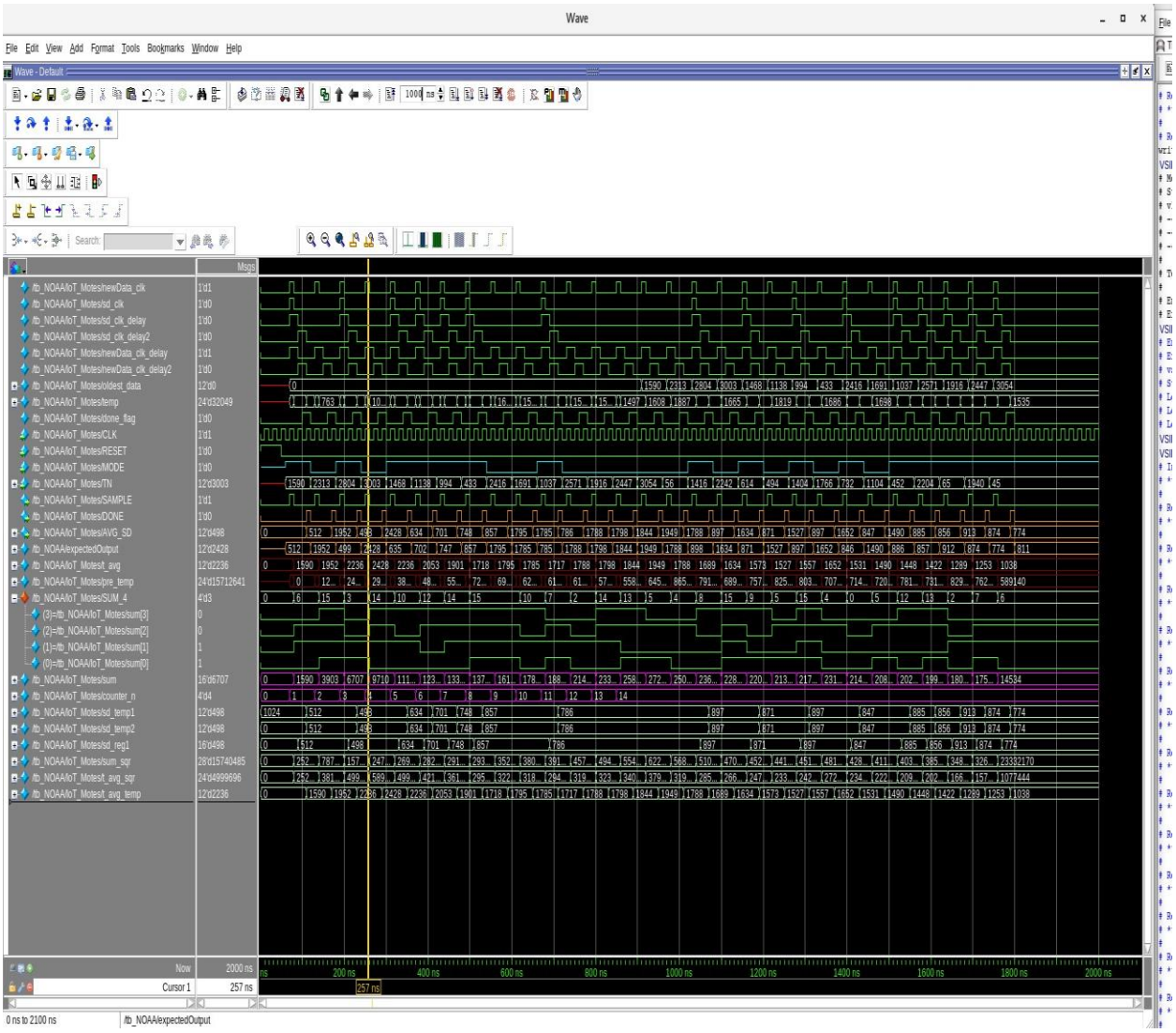
```
            counter_n <= counter_n + 1;
     end
  end

always @(negedge clock, posedge reset)
begin
  if (reset) begin
    newData_flag1 <= 0;
end else begin
    if (memory[0] == DATAIN)
            newData_flag1 <= 0;
    else
      newData_flag1 <= 1;
end
end

assign newData_clk1 = clock & newData_flag1;
assign DATAOUT = memory[13];

endmodule
```

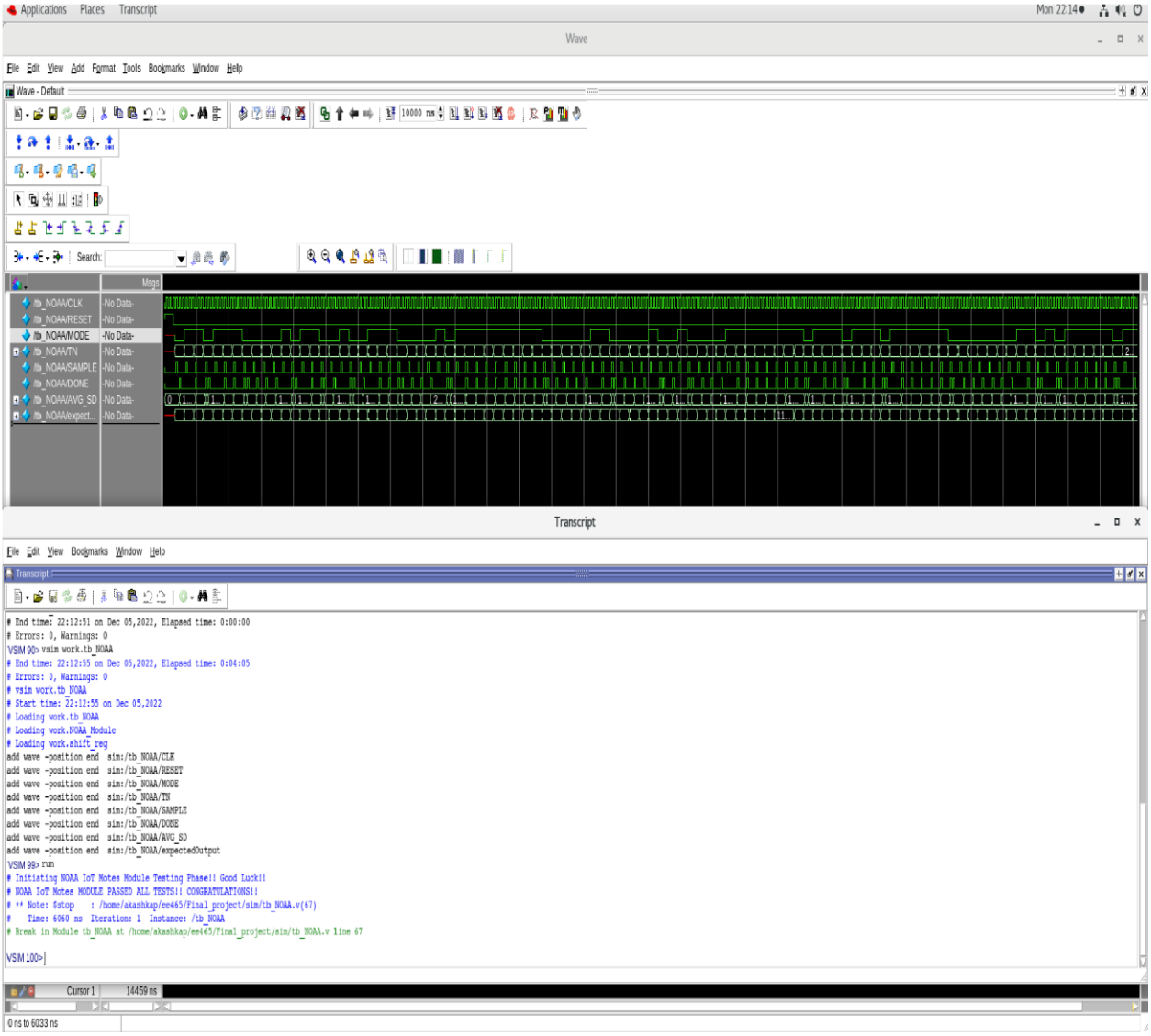
5. Test results



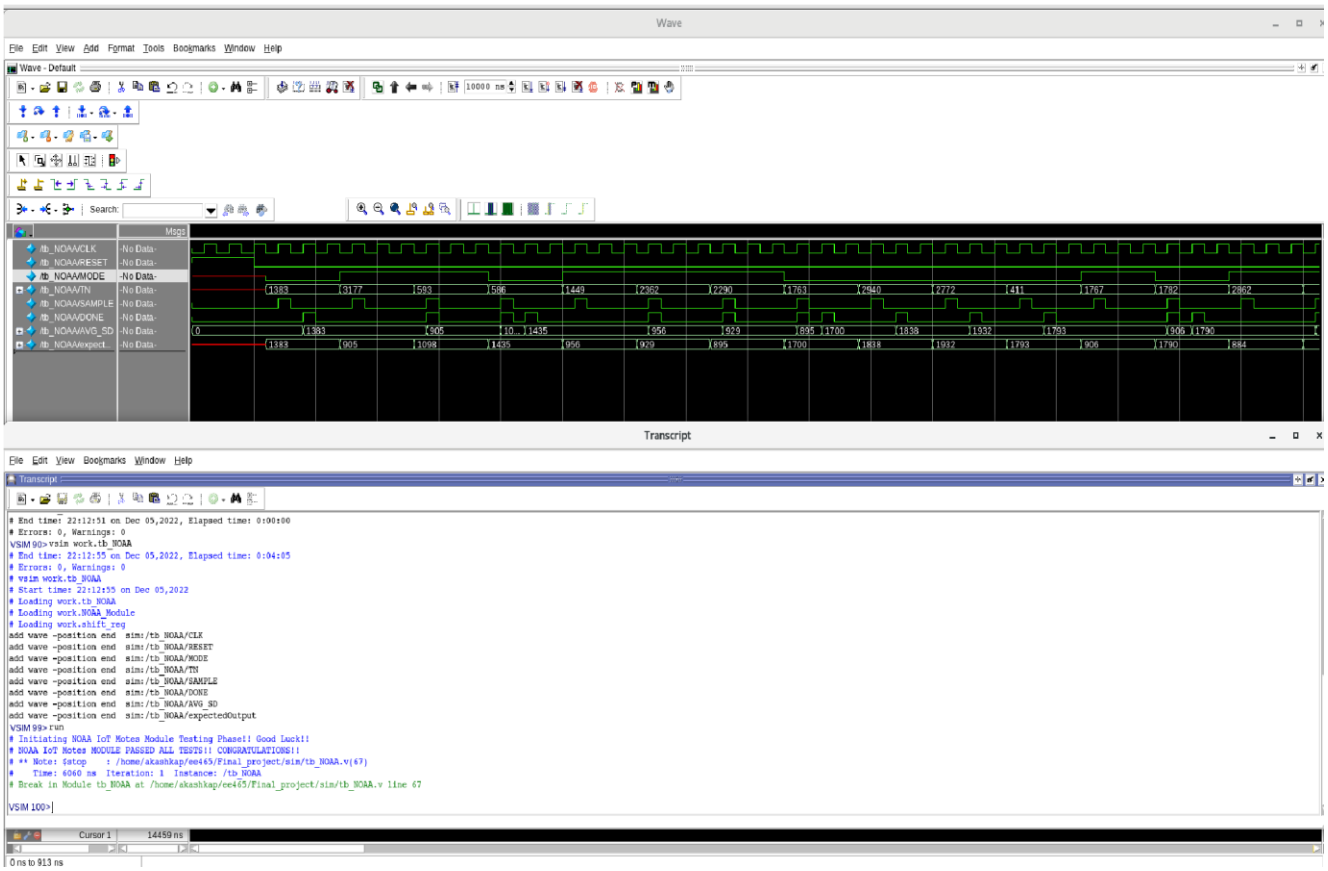
This is the simulation with initial code with failing test scenarios due to rounding errors and mismatch between cycle for different outputs

Test benches and results

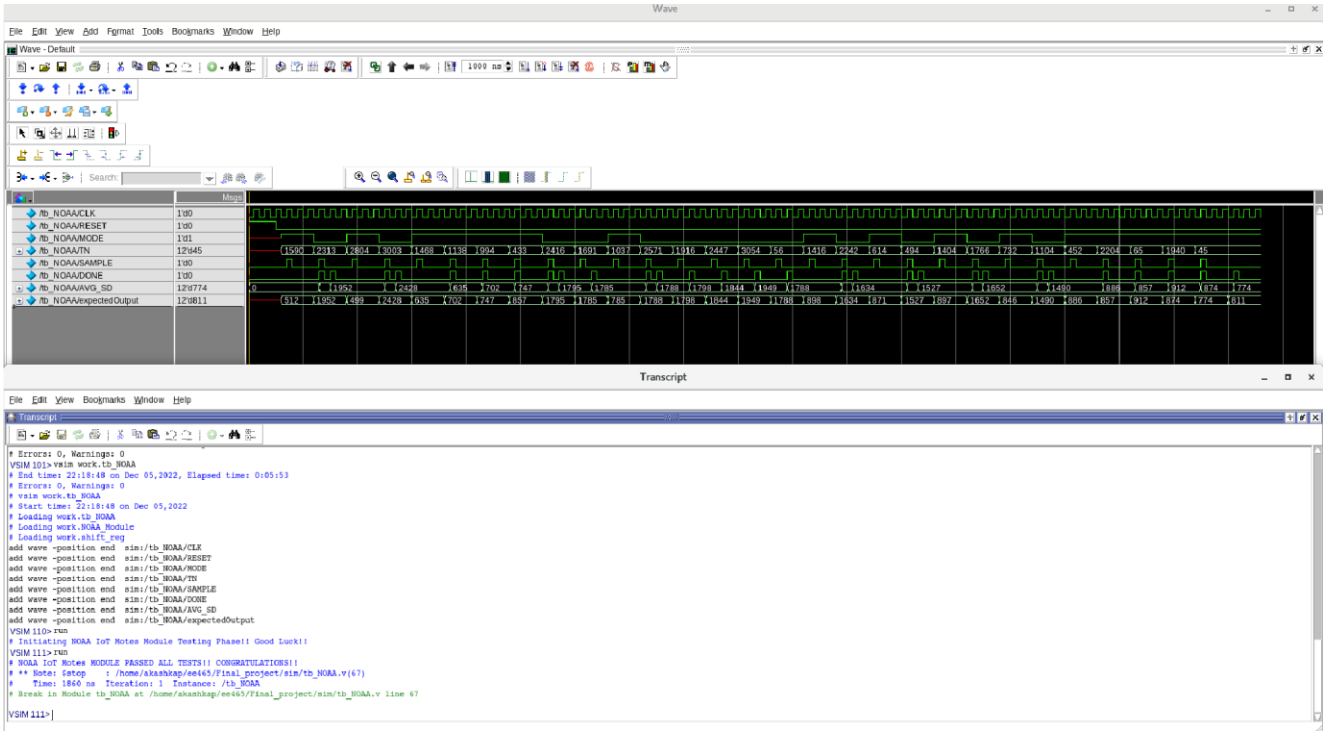
Below shows the successful running of our design with provided test-benches.



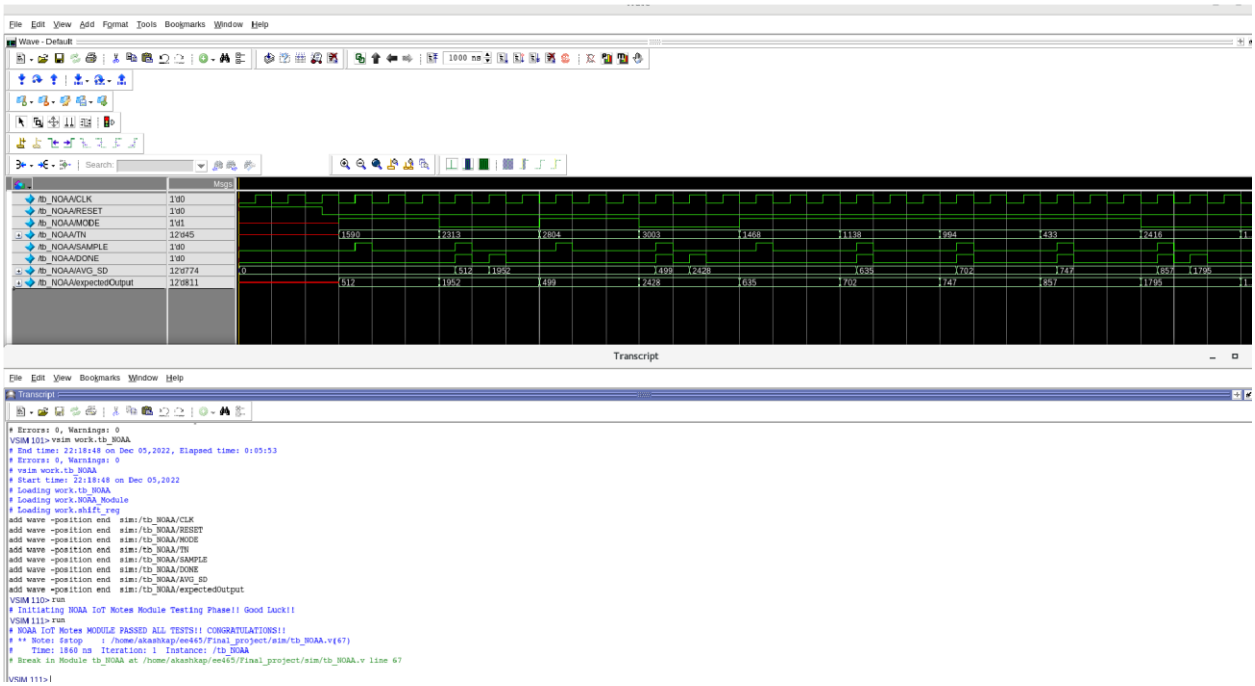
Using the 100 values, final simulation run



Zoomed screenshot of the above run.



Using the 30 value data.



Zoomed version of above simulation.

6. Conclusions and Discussion

The learning experience of solving a real world problem using the techniques we learned in the class and lab was an eye opener. The hardest part of the project was verilog coding and optimizing our code to obtain required results(basically incorporating division towards the end in the code). Suggestions would be to use better tools like Cadence NCSIM and Synopsys Verdi for simulation part(or even Xilinx Vivado is better than ModelSim) and different project descriptions should be given to different groups to enhance creativity.