To demonstrate our knowledge of digital ASIC design, we were tasked with creating a circuit which calculates the moving average or standard deviation of a dataset.

Moving Average and Standard Deviation ASIC

Final Project – EE 465

Reece Dodge

Introduction

Digital ASICs (Application-Specific Integrated Circuits) can play a crucial role in processing temperature data in various applications. Digital ASICs can process temperature data in real-time, enabling timely responses to temperature changes. This is essential in applications where rapid adjustments or interventions based on temperature variations are required.

ASICs can be integrated with temperature sensors on the same chip, forming a complete solution for temperature monitoring and control. This integration reduces the need for external components, making the system more compact and efficient. For this project, we are focusing strictly on the data processing portion of a temperature monitoring system.

Design Specifications

OVERVIEW

"The United States Department of Agriculture (USDA) has discovered that the temperature of a plant affects the protein level present in the plant. They want to investigate more about the relationship, hence the need to monitor the temperature of the plants remotely.

They come to see you, 'a certified digital circuit engineer'. They want you to design a circuit that takes the temperature readings from a temperature sensor connected to IoT motes placed on one of the plants and calculate the average and the standard deviation of the readings.

The problem is to find the moving average and the moving standard deviation of the last 14 temperature readings from the temperature sensor."

Modifications and Compromises

Output Validity

My circuit does not output an average or standard deviation unless the FIFO sampling register is full. This allows me to use the same circuitry for each mean/standard deviation calculation, rather than implementing the hardware necessary to perform several divisions with different divisors. For example: if there are only 13 samples, the average would need to be computed by dividing by 13.

In an application such as this, there is little value to calculating the average/standard deviation of a dataset with fewer than 14 samples. Realistically, temperature sensors are capable of outputting streams of data at high frequencies and could benefit from continuous sampling. To illustrate my point: With a hypothetical frequency of 100 MHz, the FIFO register would be full after 140 nanoseconds. If the circuit is sampling for 1 minute, the additional circuitry would become irrelevant after 0.00000023% of runtime. I do not think this is acceptable, so I'm restricting my computations to a complete dataset.

FIFO Depth Increased to 16

Dividing by 16 involves shifting the binary representation of a number four positions to the right, which is equivalent to dividing by 2^4 (or 16 in decimal). By increasing the depth of my FIFO register to 16, I can avoid all but one division by a non-power-of-2. This change led to the largest overall improvement to the efficiency of my design.

Continuous Sampling and Output

My circuit is designed to take in a continuous stream of data, where the signal to capture a new sample is always high. This is made possible by my implementation of pipelining. An output is produced every clock cycle regardless of mode.

My Design

Initial Schematic

Below is a general outline for how I wanted to implement the circuit. I wanted two major pipeline stages: one responsible for computing variance and one for estimating the square root.

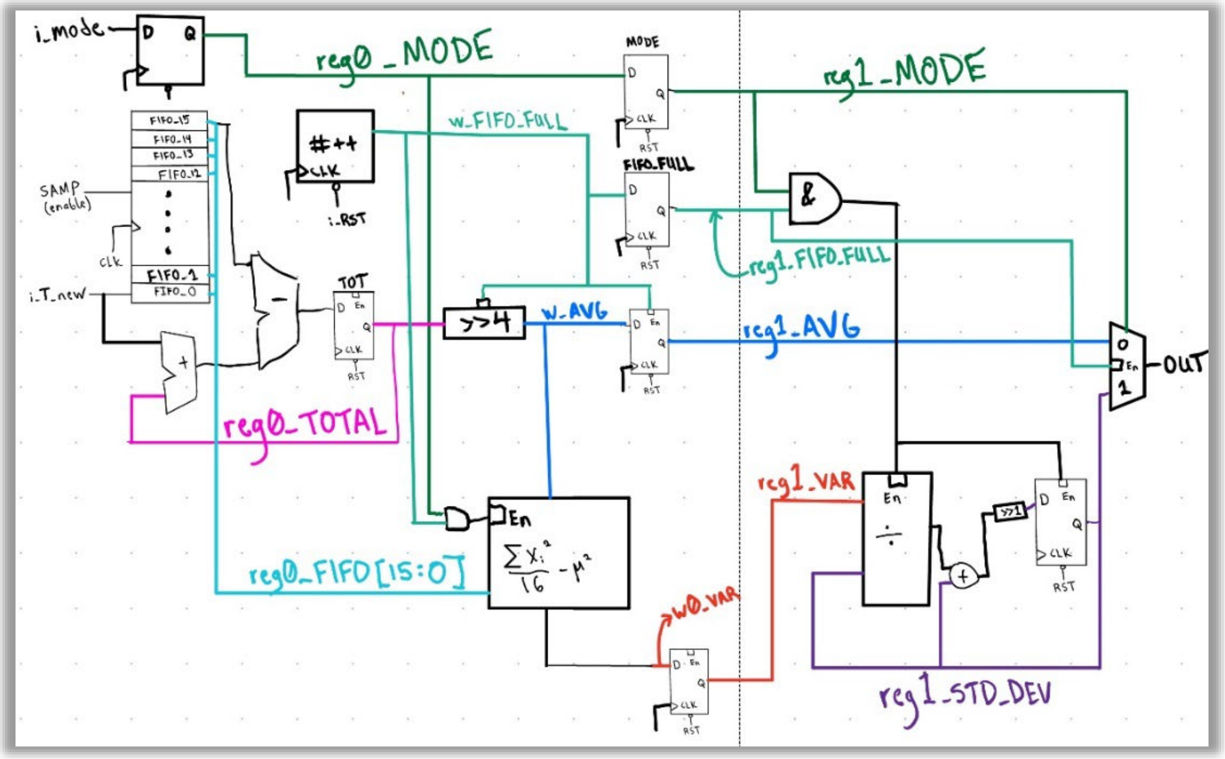
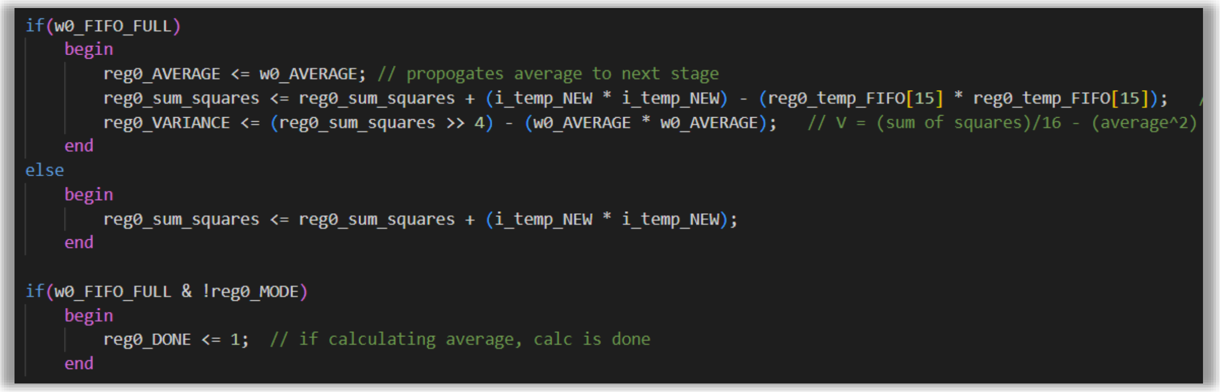


Figure 1 - Initial Schematic

Here is the formula I used to calculate variance →

$$V = \frac{\sum {X_i}^2}{16} - \mu^2$$

Below is the snippet of my code which represents the variance calculation.

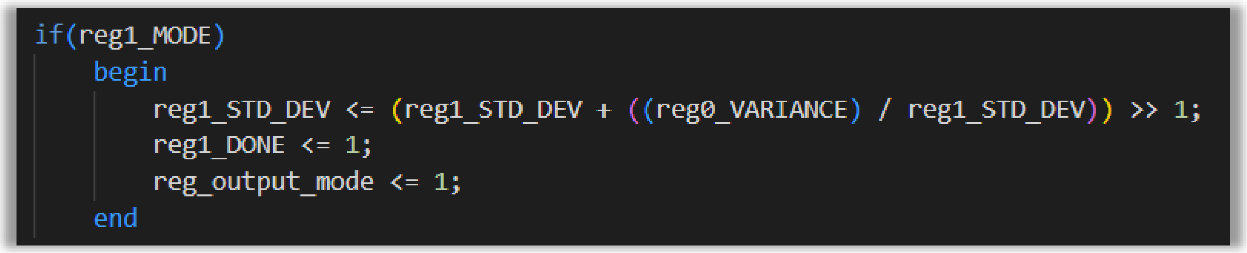


```
if(w0_FIFO_FULL)
    begin
        reg0_AVERAGE <= w0_AVERAGE; // propogates average to next stage
        reg0_sum_squares <= reg0_sum_squares + (i_temp_NEW * i_temp_NEW) - (reg0_temp_FIFO[15] * reg0_temp_FIFO[15]);   /
        reg0_VARIANCE <= (reg0_sum_squares >> 4) - (w0_AVERAGE * w0_AVERAGE);   // V = (sum of squares)/16 - (average^2)
    end
else
    begin
        reg0_sum_squares <= reg0_sum_squares + (i_temp_NEW * i_temp_NEW);
    end

if(w0_FIFO_FULL & !reg0_MODE)
    begin
        reg0_DONE <= 1;  // if calculating average, calc is done
    end
```

Figure 2 - Verilog Variance Calculation

Only one division by a non-power-of-two is present in my design. The division is present in the second major pipeline stage where this formula is used to approximate the square root of variance → Below is the formula represented in my code:

$$\sqrt{V} \approx \frac{1}{2}(\sigma + \frac{V}{\sigma})$$



```
if(reg1_MODE)
    begin
        reg1_STD_DEV <= (reg1_STD_DEV + ((reg0_VARIANCE) / reg1_STD_DEV)) >> 1;
        reg1_DONE <= 1;
        reg_output_mode <= 1;
    end
```

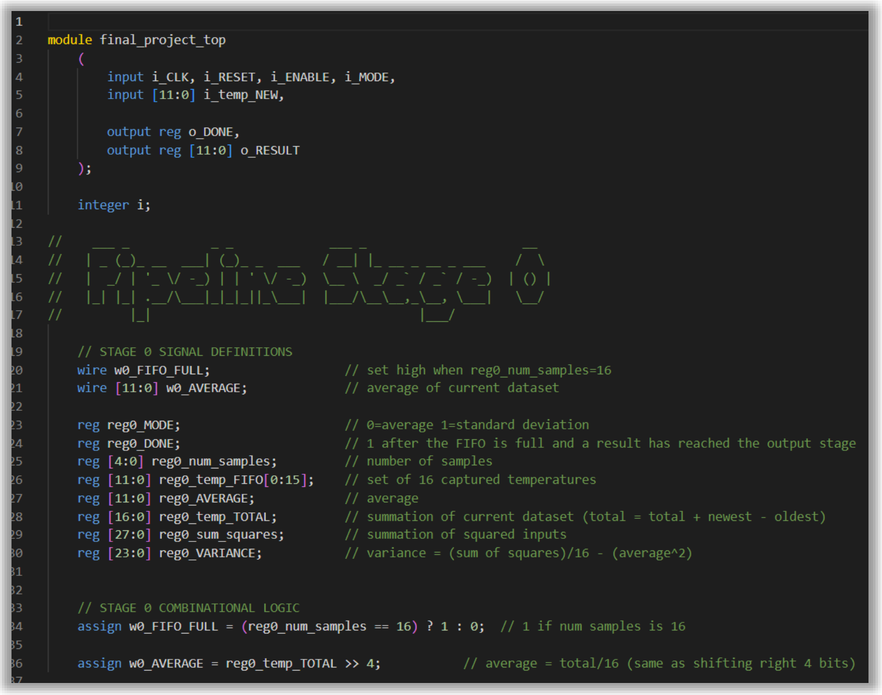
Figure 3 - Verilog Square Root Calculation

Pipelining

My design uses a 3-stage pipeline that outputs a result on every rising clock edge. As a result, the output is reflected 3 clock edges after the input is captured. Stage 0 is responsible for average and variance calculations, stage 1 is responsible for square root estimation, and the final stage is responsible for producing the output.

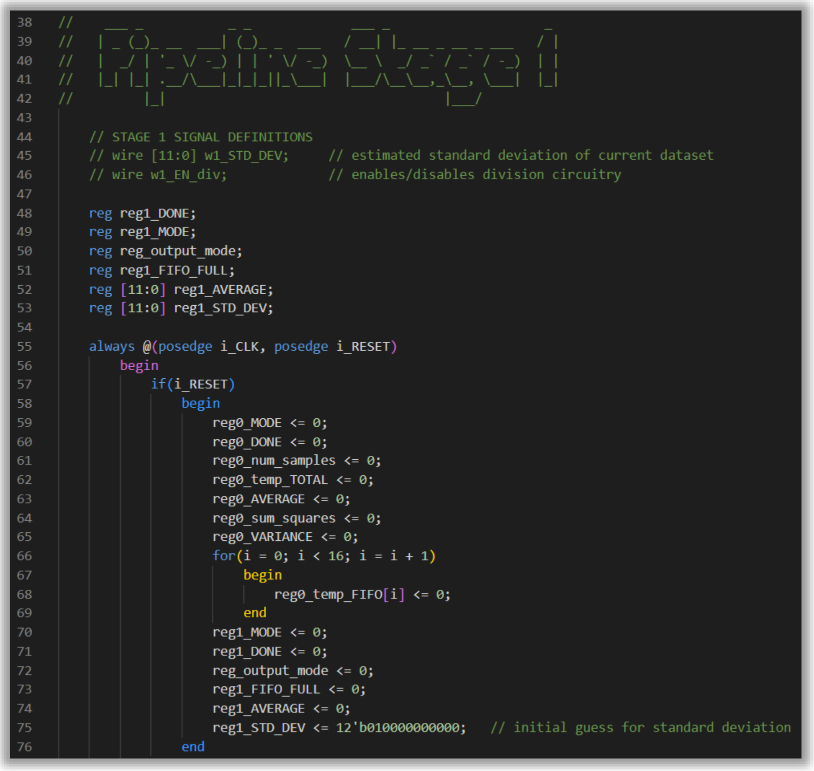
Below is my entire final project verilog code, loosely separated by stages:

Verilog



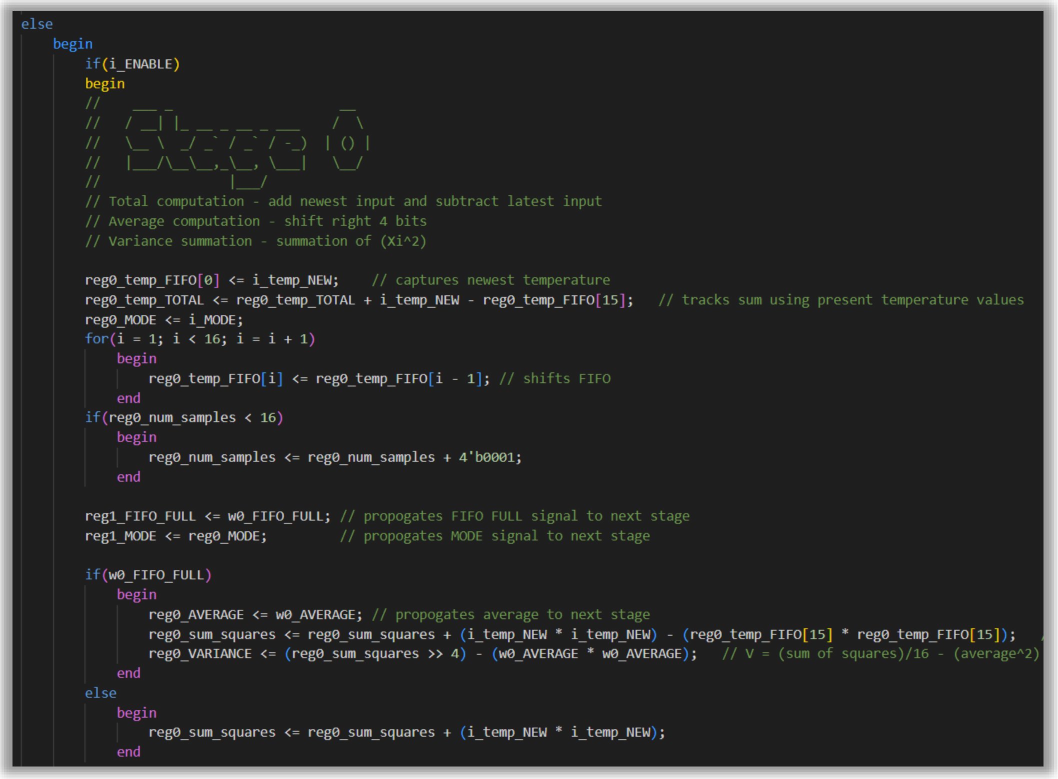
```verilog

module final_project_top
    (
        input i_CLK, i_RESET, i_ENABLE, i_MODE,
        input [11:0] i_temp_NEW,

        output reg o_DONE,
        output reg [11:0] o_RESULT
    );

    integer i;

//   ___ _           _ _              ___ _                    __
//  | _ (_)_ __  ___| (_)_ _  ___   / __| |_ __ _ __ _ ___   /  \
//  |  _/ | '_ \/ -_) | | ' \/ -_)  \__ \  _/ _` / _` / -_) | () |
//  |_| |_| .__/\___|_|_|_||_\___|  |___/\__\__,_\__, \___|  \__/
//        |_|                                    |___/

    // STAGE 0 SIGNAL DEFINITIONS
    wire w0_FIFO_FULL;                  // set high when reg0_num_samples=16
    wire [11:0] w0_AVERAGE;             // average of current dataset

    reg reg0_MODE;                      // 0=average 1=standard deviation
    reg reg0_DONE;                      // 1 after the FIFO is full and a result has reached the output stage
    reg [4:0] reg0_num_samples;         // number of samples
    reg [11:0] reg0_temp_FIFO[0:15];    // set of 16 captured temperatures
    reg [11:0] reg0_AVERAGE;            // average
    reg [16:0] reg0_temp_TOTAL;         // summation of current dataset (total = total + newest - oldest)
    reg [27:0] reg0_sum_squares;        // summation of squared inputs
    reg [23:0] reg0_VARIANCE;           // variance = (sum of squares)/16 - (average^2)


    // STAGE 0 COMBINATIONAL LOGIC
    assign w0_FIFO_FULL = (reg0_num_samples == 16) ? 1 : 0;  // 1 if num samples is 16

    assign w0_AVERAGE = reg0_temp_TOTAL >> 4;           // average = total/16 (same as shifting right 4 bits)
```

Figure 4 – Verilog – Top level module declaration, pipeline stage 0 wires, registers, and combinational logic



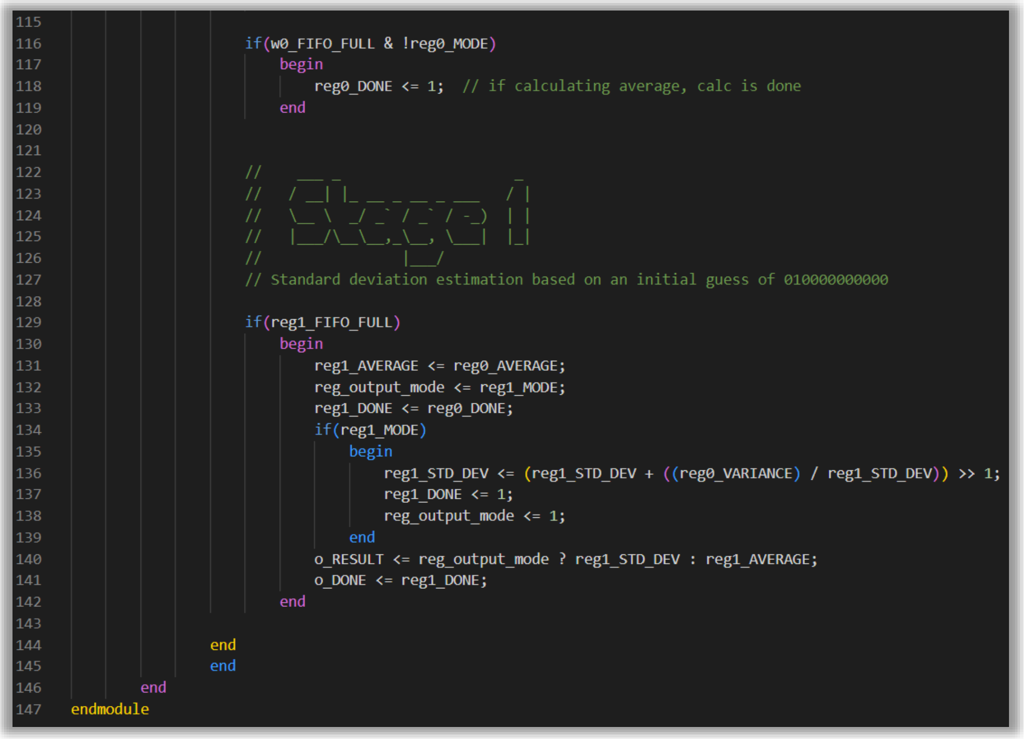
```verilog
//   ___ _           _ _              ___ _                    _
//  | _ (_)_ __  ___| (_)_ _  ___   / __| |_ __ _ __ _ ___   / |
//  |  _/ | '_ \/ -_) | | ' \/ -_)  \__ \  _/ _` / _` / -_)  | |
//  |_| |_| .__/\___|_|_|_||_\___|  |___/\__\__,_\__, \___|  |_|
//        |_|                                    |___/

    // STAGE 1 SIGNAL DEFINITIONS
    // wire [11:0] w1_STD_DEV;      // estimated standard deviation of current dataset
    // wire w1_EN_div;              // enables/disables division circuitry

    reg reg1_DONE;
    reg reg1_MODE;
    reg reg_output_mode;
    reg reg1_FIFO_FULL;
    reg [11:0] reg1_AVERAGE;
    reg [11:0] reg1_STD_DEV;

    always @(posedge i_CLK, posedge i_RESET)
        begin
            if(i_RESET)
                begin
                    reg0_MODE <= 0;
                    reg0_DONE <= 0;
                    reg0_num_samples <= 0;
                    reg0_temp_TOTAL <= 0;
                    reg0_AVERAGE <= 0;
                    reg0_sum_squares <= 0;
                    reg0_VARIANCE <= 0;
                    for(i = 0; i < 16; i = i + 1)
                        begin
                            reg0_temp_FIFO[i] <= 0;
                        end
                    reg1_MODE <= 0;
                    reg1_DONE <= 0;
                    reg_output_mode <= 0;
                    reg1_FIFO_FULL <= 0;
                    reg1_AVERAGE <= 0;
                    reg1_STD_DEV <= 12'b010000000000;   // initial guess for standard deviation
                end
```

Figure 5 - Verilog – Pipeline stage 1 wires and registers, reset block



```verilog
else
    begin
        if(i_ENABLE)
        begin
        //    ___ _                  __
        //   / __| |_ __ _ __ _ ___  /  \
        //   \__ \  _/ _` / _` / -_)| () |
        //   |___/\__\__,_\__, \___| \__/
        //                |___/
        // Total computation - add newest input and subtract latest input
        // Average computation - shift right 4 bits
        // Variance summation - summation of (Xi^2)

        reg0_temp_FIFO[0] <= i_temp_NEW;    // captures newest temperature
        reg0_temp_TOTAL <= reg0_temp_TOTAL + i_temp_NEW - reg0_temp_FIFO[15];   // tracks sum using present temperature values
        reg0_MODE <= i_MODE;
        for(i = 1; i < 16; i = i + 1)
            begin
                reg0_temp_FIFO[i] <= reg0_temp_FIFO[i - 1]; // shifts FIFO
            end
        if(reg0_num_samples < 16)
            begin
                reg0_num_samples <= reg0_num_samples + 4'b0001;
            end

        reg1_FIFO_FULL <= w0_FIFO_FULL; // propogates FIFO FULL signal to next stage
        reg1_MODE <= reg0_MODE;         // propogates MODE signal to next stage

        if(w0_FIFO_FULL)
            begin
                reg0_AVERAGE <= w0_AVERAGE; // propogates average to next stage
                reg0_sum_squares <= reg0_sum_squares + (i_temp_NEW * i_temp_NEW) - (reg0_temp_FIFO[15] * reg0_temp_FIFO[15]);
                reg0_VARIANCE <= (reg0_sum_squares >> 4) - (w0_AVERAGE * w0_AVERAGE);   // V = (sum of squares)/16 - (average^2)
            end
        else
            begin
                reg0_sum_squares <= reg0_sum_squares + (i_temp_NEW * i_temp_NEW);
            end
```

Figure 6 - Verilog – stage 0 sequential logic



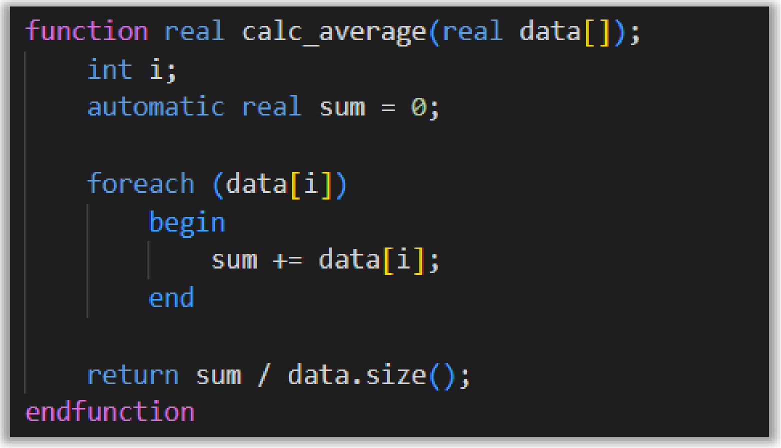
```verilog
                if(w0_FIFO_FULL & !reg0_MODE)
                    begin
                        reg0_DONE <= 1;  // if calculating average, calc is done
                    end


                //    ___ _                  _
                //   / __| |_ __ _ __ _ ___  / |
                //   \__ \  _/ _` / _` / -_) | |
                //   |___/\__\__,_\__, \___| |_|
                //                |___/
                // Standard deviation estimation based on an initial guess of 010000000000

                if(reg1_FIFO_FULL)
                    begin
                        reg1_AVERAGE <= reg0_AVERAGE;
                        reg_output_mode <= reg1_MODE;
                        reg1_DONE <= reg0_DONE;
                        if(reg1_MODE)
                            begin
                                reg1_STD_DEV <= (reg1_STD_DEV + ((reg0_VARIANCE) / reg1_STD_DEV)) >> 1;
                                reg1_DONE <= 1;
                                reg_output_mode <= 1;
                            end
                        o_RESULT <= reg_output_mode ? reg1_STD_DEV : reg1_AVERAGE;
                        o_DONE <= reg1_DONE;
                    end

            end
            end
        end
endmodule
```

Figure 7 - Verilog – Stage 1 sequential logic and output stage

Custom Test Bench

Since my design strays from the design specifications, I decided to build my own test bench. Retrofitting the provided test bench would've taken me as long as it did to build a custom one. Plus, writing my test bench in system verilog gives me greater flexibility, such as the "$random" function.

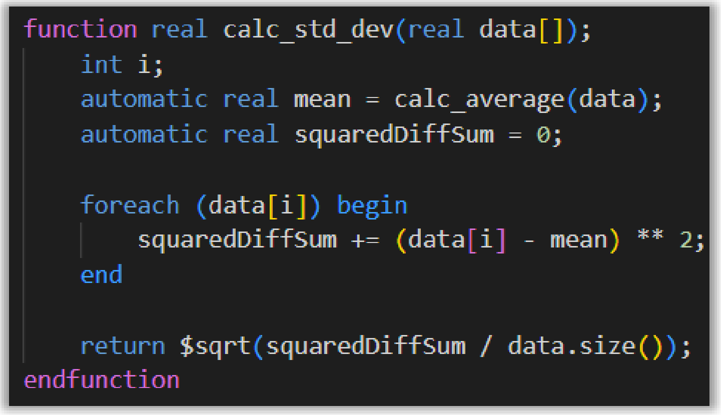
First, I wrote 2 system verilog functions to mathematically calculate average and standard deviation. Here are the two functions:



```
function real calc_average(real data[]);
    int i;
    automatic real sum = 0;

    foreach (data[i])
        begin
            sum += data[i];
        end

    return sum / data.size();
endfunction
```

Figure 8 - Mathematical mean function

This function is used in the standard deviation calculation. I used the "$sqrt" function rather than the estimation function found in the actual design, so my standard deviation output values are more likely to differ from the mathematical test bench output.

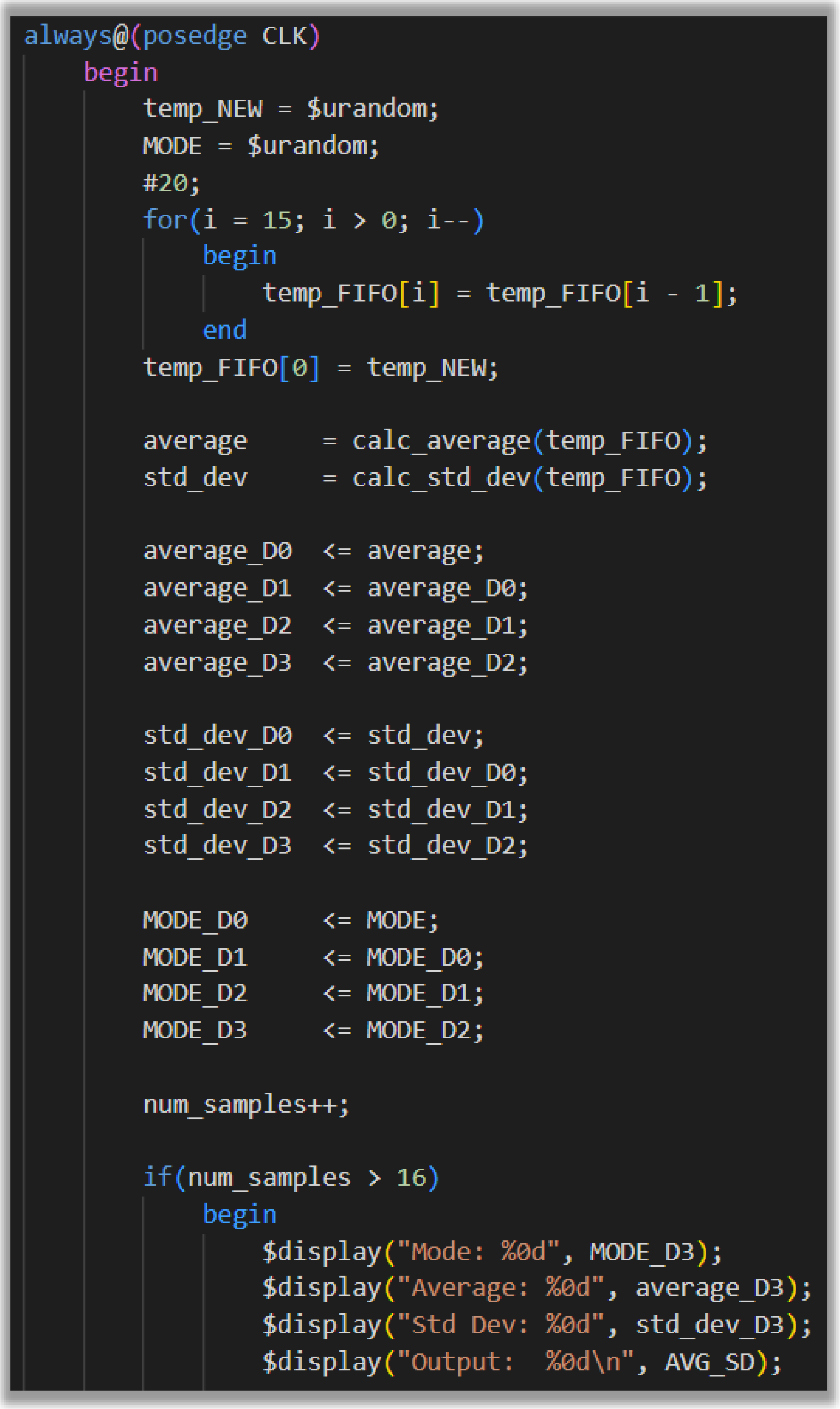


```
function real calc_std_dev(real data[]);
    int i;
    automatic real mean = calc_average(data);
    automatic real squaredDiffSum = 0;

    foreach (data[i]) begin
        squaredDiffSum += (data[i] - mean) ** 2;
    end

    return $sqrt(squaredDiffSum / data.size());
endfunction
```

Figure 9 - Mathematical standard deviation function

Next, I wrote a test bench structure to randomly generate test data and mirror the data into a FIFO register. The calculated mean/standard deviation are produced, and 3 clock cycles of delay are added to align the calculations with the output of my circuit.

Here is the test data generation structure:

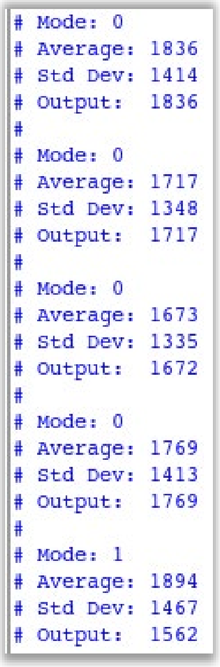


```
always@(posedge CLK)
    begin
        temp_NEW = $urandom;
        MODE = $urandom;
        #20;
        for(i = 15; i > 0; i--)
            begin
                temp_FIFO[i] = temp_FIFO[i - 1];
            end
        temp_FIFO[0] = temp_NEW;

        average     = calc_average(temp_FIFO);
        std_dev     = calc_std_dev(temp_FIFO);

        average_D0  <= average;
        average_D1  <= average_D0;
        average_D2  <= average_D1;
        average_D3  <= average_D2;

        std_dev_D0  <= std_dev;
        std_dev_D1  <= std_dev_D0;
        std_dev_D2  <= std_dev_D1;
        std_dev_D3  <= std_dev_D2;

        MODE_D0     <= MODE;
        MODE_D1     <= MODE_D0;
        MODE_D2     <= MODE_D1;
        MODE_D3     <= MODE_D2;

        num_samples++;

        if(num_samples > 16)
            begin
                $display("Mode: %0d", MODE_D3);
                $display("Average: %0d", average_D3);
                $display("Std Dev: %0d", std_dev_D3);
                $display("Output:  %0d\n", AVG_SD);
```

Figure 10 - Test data generation

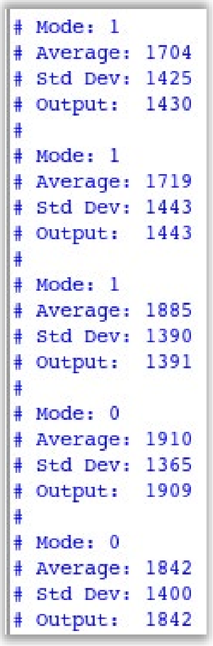
Simulation Output

Below are screenshots of Modelsim output logs demonstrating proper calculations and simulation output waveforms:



```
# Mode: 0
# Average: 1836
# Std Dev: 1414
# Output:  1836
#
# Mode: 0
# Average: 1717
# Std Dev: 1348
# Output:  1717
#
# Mode: 0
# Average: 1673
# Std Dev: 1335
# Output:  1672
#
# Mode: 0
# Average: 1769
# Std Dev: 1413
# Output:  1769
#
# Mode: 1
# Average: 1894
# Std Dev: 1467
# Output:  1562
```

Figure 11 - Simulation output log



```
# Mode: 1
# Average: 1704
# Std Dev: 1425
# Output:  1430
#
# Mode: 1
# Average: 1719
# Std Dev: 1443
# Output:  1443
#
# Mode: 1
# Average: 1885
# Std Dev: 1390
# Output:  1391
#
# Mode: 0
# Average: 1910
# Std Dev: 1365
# Output:  1909
#
# Mode: 0
# Average: 1842
# Std Dev: 1400
# Output:  1842
```

Figure 12 - Simulation output log II

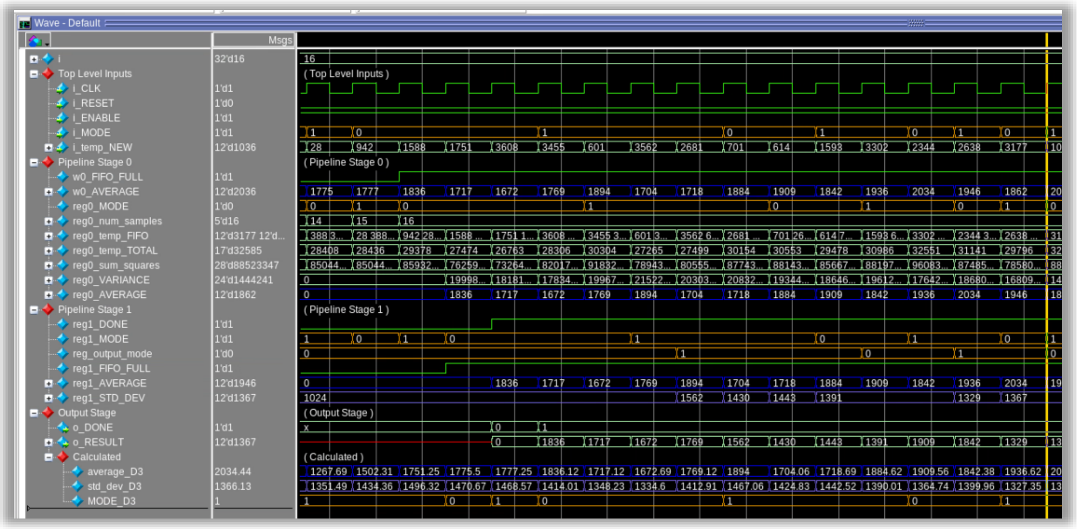


Figure 13 - Simulation output waveform

The differences between calculated standard deviation and my output standard deviation can be attributed to my use of a square root approximation. The standard deviation output is more accurate if other standard deviation calculations have occurred recently. Multiple consecutive standard deviations will always produce the most accurate outputs.

One inherent advantage of my circuit is **gated division**. Since the division is only performed when a standard deviation is calculated, the circuit will naturally consume less power.

Genus Synthesis

Once I got my verilog working correctly, I set out to find the minimum clock period of the circuit before retiming optimization. Since my design is already pipelined, implied division circuitry is a significantly larger bottleneck than data register timing adjustments, so I do not anticipate retiming to have a major impact.

Below is a tabular comparison of my design before and after retiming, including a percentage change for each parameter in question:

Parameter	No Retiming	Retimed	% Change
Min. Clock Period	4100	4000	-2%
Max Frequency (MHz)	243.902439	250	3%
Area (µm^2)	14833.4	15458.4	4%
Power (mW)	1.49798	1.65693	11%
Slack (ps)	0	0	0%
Gates	4702	4878	4%
Registers	320	331	3%

Table 1 - Retiming synthesis comparison

As you can see, retiming only led to a 2% reduction in clock period. However, retiming enabled me to reach a nice, round maximum frequency of 250 MHz

Below is the synthesized schematic from Genus:

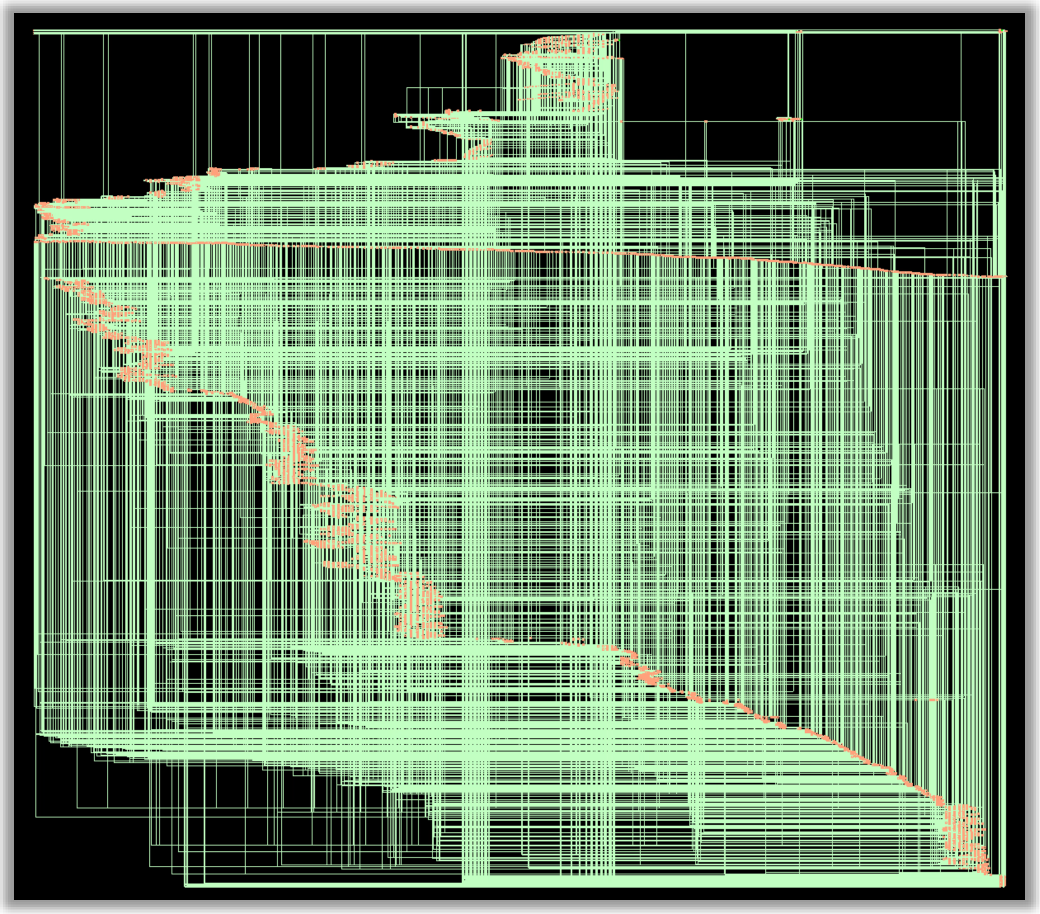


Figure 14 - Synthesized schematic

Layout – Innovus Placement and Routing

I ran the placement and routing tool flow similarly to lab 6. Here are the results:

Parameter	Pre-Optimization	Post-Optimization
Area (μm^2)	17929.8	18106.56
Power (mW)	5.84815984	5.8733316
Slack (ps)	-0.706	-0.645
Instances	5036	5080

The Innovus post-route optimization appears to have increased area but saved slack. Below is the design fully placed and routed:

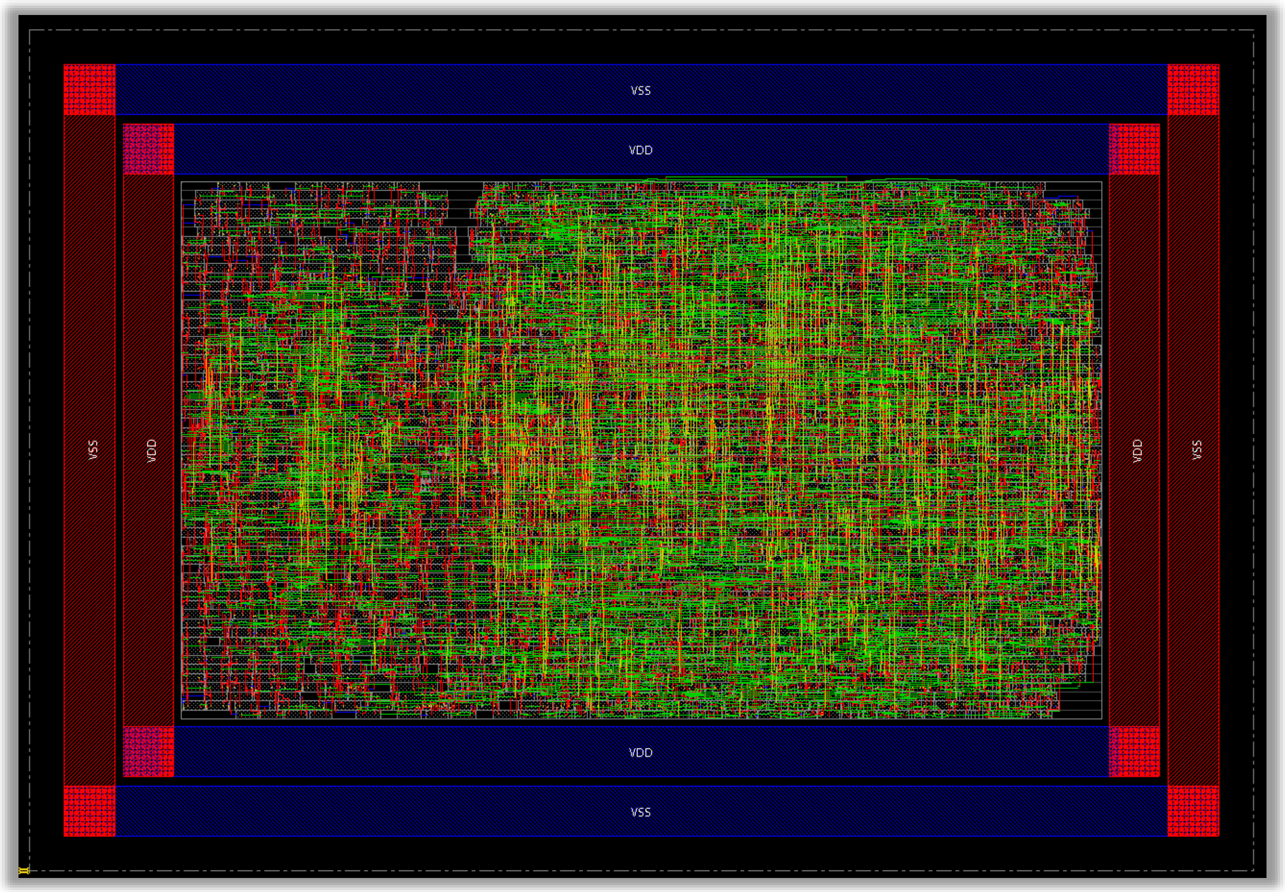


Figure 15 - Synthesized layout

Conclusion

I believe my changes to the design specifications were intelligent and justified. Reducing the division hardware to a single instance gives me a massive advantage in terms of delay, area, and power alike. Also, throughput is significantly higher compared to a circuit without pipelining.

If I were to change anything about the design, I would adjust my pipelining implementation to involve 4 stages total:

- 0 – Input Capture and average calculation
- 1 – Variance calculation
- 2 – Square root estimation
- 3 – Output selection

This would allow me to be confident that my circuit is always operating on the correct dataset. I often had to double/triple check that my circuit was both 1: operating on the correct data and 2: producing accurate calculations. It is also bad practice to perform calculations on input wires if you cannot be certain the data is staying the same for the necessary amount of time for accuracy. Additionally, I would introduce circuitry to account for the remainder of the single division.

One other change I could possibly implement with more time is **data forwarding**. Since averages are available immediately after the input is captured, I could tune the circuit to output average values sooner. This approach would require highly robust forwarding logic to guarantee standard deviation calculations later in the pipeline aren't overwritten.