# IoT Environmental Monitoring

## Final Project EE465

**Team**

**Professor: Chris Chu**
**Organization: Iowa State University**

---

**Team Members**

**Ahmed Abuhjar**
**aabuhjar@iastate.edu**

**Zach Bumstead**
**zrbum@iastate.edu**

Revised: 12/03/2018

# Introduction

Our final project is to create a circuit that helps getting the average and the standard deviation for the samples that are relayed from the sensors in the IoT network. Specifically, we had to design a circuit that takes several samples of temperature inputs and then have the circuit do more than one arithmetic operations to determine the average and the standard deviation for the last ten samples. The Verilog code for the design is shown below. We first had to implement the code that satisfies the requirement using Verilog. The circuit will periodically take samples to temperature measurement. After the result has been confirmed to be satisfactory, we then synthesized the circuit using RTL method. After synthesizing the circuit, we optimized our design to make it small in size and more power efficient by reducing the number of clock cycles the system can operate with. This project was a group work, and we have cooperated to have the project meet all the requirements. We organized and implemented the code and synthesized and optimized the corresponding logic circuit.

# Design

## Design Overview

The system takes four inputs, *RESET*, *MODE*, *TN*, and *CLK*. *RESET* is used to restart the sampling and computations. *MODE* is used to determine whether the average or standard deviation is to be calculated. *TN* is the input data coming from the temperature sensor. This data will be used to compute average or standard deviation. The last input is the clock signal. Table 1 on the next page summarizes the ports that were just mentioned. It also describes their sizes and how they are used.

| Port | Type | Description |
|---|---|---|
| **Reset** | 1 bit Input | When reset is set to high, the circuit will be initialized, and it will start taking samples from the beginning when the reset turns to low. |
| **TN** | 12 bits Input | TN corresponds to temperature readings from the sensors to the circuit. 12 bits of inputs is adequate to take a wide range of temperature readings. |
| **CLK** | 1 bit Input | Clock period is set to be 6 ns |
| **MODE** | 1 Input | If mode is high, the circuit will calculate the standard deviation. If the mode is low, the circuit will calculate the average. |
| **SAMPLE** | 1 bit Output | This bit is set to high when the circuit is ready to take the next readings. In our design. Sample is set to one every clock cycle which will make the circuit power efficient. |
| **DONE** | 1 bit Output | This bit is set to high when the average or the standard deviation output is calculated. |
| **AVG/SD** | 12 bit Output | This is a 12 bit output and it will be either the average or the standard deviation depending in the input MODE |

Table 1: Summary of ports

The three outputs of the system are *SAMPLE*, *DONE*, and *AVG/SD*. *SAMPLE* is set to high when the system is reading in samples. Our system is always sampling as soon as *RESET* is set to low, so *SAMPLE* will be high always in this case. *DONE* is set to high each time an average or standard deviation is computed. When *DONE* is high, the test bench compares *AVG_SD* with the expected output. Lastly, *AVG/SD* represents the average or standard deviation that is computed. When *DONE* is high, the test bench compares *AVG_SD* with the expected output.

For this project, we used Verilog to build our design. The code itself can be divided into two sections. The first section is concerned with reading the inputs from *TN* and computing the variance of the last ten signals. The second section is concerned with computing either the standard deviation or the average, depending on whether *MODE* is high or low.

For the first section, the sampling of the various inputs is separated into two blocks. The first block is used to load the first ten samples. The second block loads a new input from *TN* after the first ten have been sampled already. This is done by shifting all of the loaded samples to an adjacent register and loading the new sample into the open register. For example, the design uses ten registers, *r1* through *r10,* to hold the previous ten samples. After the first ten samples are loaded into these registers, the second block of code in this section will shift the samples down a register and load the new sample into *r10.* The contents of *r2* will be shifted into *r1*, *r3* will be shifted into *r2*, and so on. This method of sampling allows the system to always perform calculations based on the last ten samples recorded.

The second section computes the average and the standard deviation, depending on the status of the *MODE* input. If *MODE* is set to high, then the standard deviation will be computed using the variance that was calculated whenever a new sample has been loaded to a register. If *MODE* is set to low, then the average of the last ten samples will be computed. The computed results will be sent to the output, *AVG_SD*.

## Calculating *AVG_SD*

The computation of *AVG_SD* depends on the status of *MODE*. If *MODE* is low, then *AVG_SD* will be equal to the average of the last ten samples. The formula for computing the average is not difficult. To do so, the summation of the previous ten samples is divided by the number of samples. If the total number of samples is less than ten, then the summation will be divided by the number of samples as stated previously. If more than ten samples have been recorded, then each subsequent average will be computed by dividing the average by ten.

If *MODE* is set to high, then *AVG_SD* will be equal to the standard deviation, given the variance and previous standard deviation. A formula for calculating standard deviation was given to us in the project manual. The equation can be seen in Figure 1 below.

$$\sigma = \sqrt{\frac{1}{10}\sum_{i=n-9}^{n}\left(T_i - T_{avg}\right)^2} = \sqrt{\frac{1}{10}\left(\sum_{i=n-9}^{n} {T_i}^2\right) - {T_{avg}}^2}$$

Where:

$T_i =$ the temperature reading of sample $i$.
$T_{avg} =$ the calculated average for the last ten samples
$n =$ the order number of the samples

Figure 1: Equation for standard deviation

Unfortunately, computing the square root in our design could be difficult. The project manual describes a method of estimating the square root of a number. The method involves an equation that requires the variance and previous standard deviation. The equation given to us to calculate the square root of the variance to find standard deviation, and this is shown below in Figure 2.

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

Where:

$V$ = The variance of the last ten samples. This variance depends on the average of the last ten samples as well. As the average changes, the variance will change.

$\hat{\sigma}$ = The guess of the standard deviation. This value is set to the moving standard deviation when the MODE bit is set to high. At the beginning, this value is set to be roughly 32 in decimal (12'b010000000000).

Figure 2: Estimation of standard deviation

From previous knowledge of statistics, we recognized that the square root of the variance is equal to the standard deviation. By looking at Figure 1, it can be observed that variance is equal to the expression within the square root. We can use this expression to compute the variance required by the equation in Figure 2.

Besides variance, we also need to keep track of previous standard deviations. The project manual specifies that the initial value of the previous standard deviation is 1024. Therefore, we initialized the variable that we used to hold the previous standard deviation to 1024. Afterwards, when later standard deviations are being calculated, the previous standard deviation variable is set equal to the current standard deviation. The current standard deviation is calculated first, and the previous standard deviation is calculated second. This ensures that the previous standard deviation does not change until after the current standard deviation is computed.

By looking at the equations in Figures 1 and 2, you can see that multiple division operations occur for each equation. We noticed that performing multiple division operations within one line of Verilog code does not lead to accurate results. Therefore, we had to find a way to calculate standard deviation and variance using only one division operation in each equation. Through some algebraic manipulation, we were able to factor out common denominators in each equation so that only one division operation is performed in each equation. The resulting equations can be seen in Figures 3 below. Figure 3a shows the modified equation for calculating variance, and Figure 3b shows the modified equation for calculation the standard deviation.

## Rounding

Once we have simulated our design, we had some issues with the results as they were not rounded correctly. One way to round off our results correctly was to shift, divide. For example, in our case, to calculate average, we first shifted the original summation for all the numbers that we need to take average for. We decided to shift the sum by 2 bits to the left. Then, we divided this shifted summation by the corresponding number of samples (i.e. 10 samples or less) to get the initial result. We then determined if we need to add 1 to the resulted average by looking at the second bit from the least significant bit of the average.

## Delayed Signals

In order to get our code to work with the provided test benches, we had to delay some input signals. Specifically, we had to delay the *RESET* and *MODE* inputs. First, we focused on *RESET* because the system starts taking samples immediately one clock cycle after *RESET* is set to low. Otherwise, the code tries to place the first sample into the register *r4*. This register is supposed to store the fourth sample. By delaying *RESET* by two clock cycles, we were able to ensure that all samples were stored in the correct registers. Similarly, the system would read *MODE* incorrectly, resulting in strange values for the averages and standard deviations. To fix this issue, we delayed the MODE signal by one clock cycle. Doing this ensures that the correct mode is being read for each subsequent clock cycle.

### Other Outputs

Besides *AVG_SD*, the other outputs of the system are *SAMPLE* and *DONE*. Since our code is always sampling when the *RESET* is set to low, *SAMPLE* will be set to high as long as *RESET* is low. When *RESET* is set to high, the system resets, and *SAMPLE* is set to low until *RESET* is set to high again. When *DONE* is high, the test bench compares *AVG_SD* with the expected output. Since the system produces a new value for *AVG_SD* every other clock cycle, *DONE* will be set to high every other clock cycle as well.

# Simulations

This section of the final report goes in depth of the operation of the code. This will be done by showing various simulation waveforms from Modelsim. Figure 3a and 3b show when the code begins accepting inputs from *TN*. When *SAMPLE* is high, the system begins loading the first ten input from *TN*. This is done one at a time until all ten registers have been filled. *SAMPLE* is set to high after the *RESET* input is set low and delayed twice. The count register is also included in the waveform. It helps show that the first ten *TN* inputs are being loaded into the registers. Figure **Xa** below shows how the delayed *RESET* input affect the *SAMPLE* output. The figure also shows samples being loaded into registers one at a time.

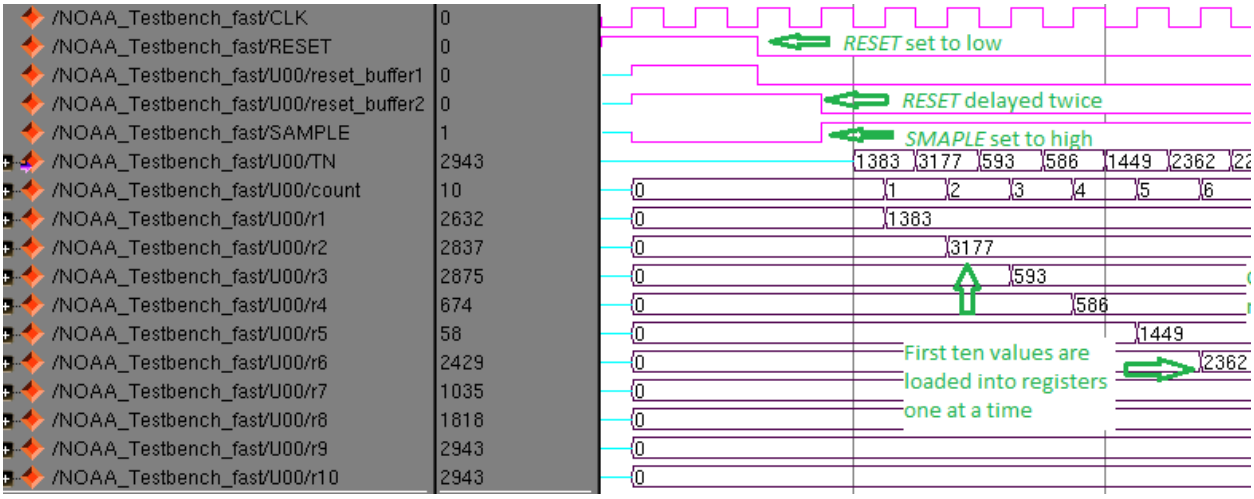


Figure 3a: First ten samples loaded into registers, *RESET*, and *SAMPLE*

Figure 3b below is a continuation of the simulation waveform in Figure 3a. It shows the count register increase as the first ten samples are loaded. It also points out the input values coming in from *TN*.

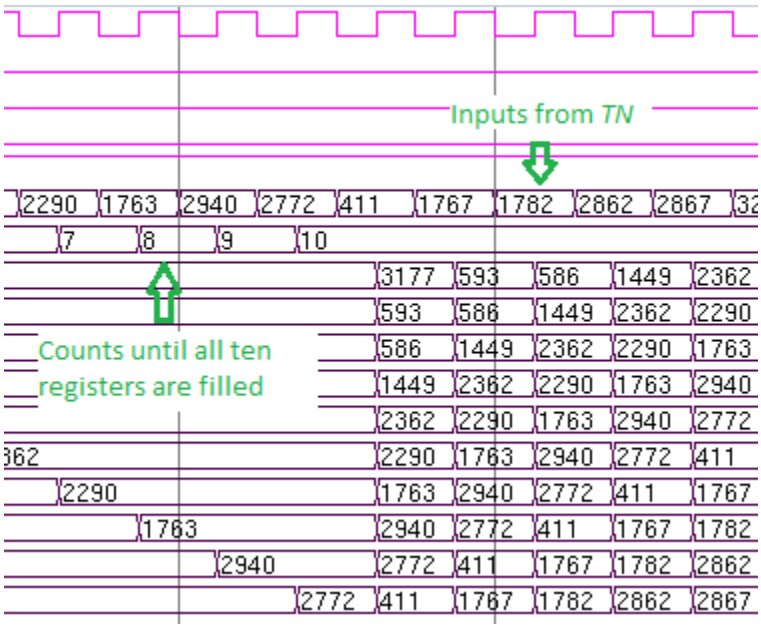


Figure 3b: *TN* inputs and count register increasing

The system wants to calculate the average or standard deviation of the last ten samples at any given time. To allow the system to do so, we shifted the previously loaded samples down a register and loaded the newest sample into the tenth register. Figure 4 below illustrates this.

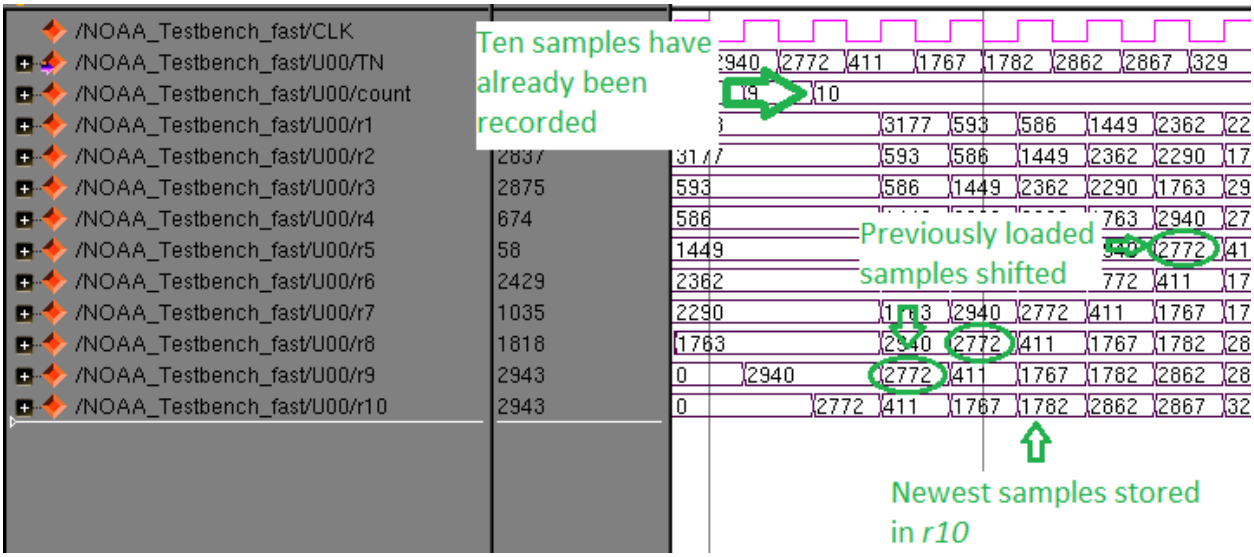


Figure 4: Samples shifted when new samples are introduced

Next, the system looks at *MODE* to either calculate the average or standard deviation. Keep in mind that our code requires *MODE* to be delayed by one clock cycle in order to function properly. You can also see that it takes one clock cycle for *AVG_SD* to be calculated after its corresponding input from *TN* is introduced to the system. This can be seen in Figure 5 below.

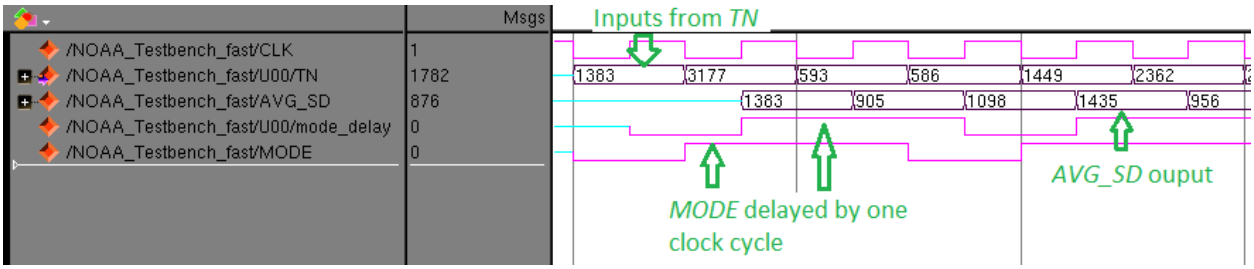


Figure 5: *AVG_SD* calculated based on value of *MODE*

Calculating the average is simple, as it only requires the previous samples and the newest sample. However, calculating standard deviation is a little trickier. Calculating standard deviation requires a more complex series of operations than when calculating the average. Since calculating standard deviation involves multiple division operations, we had to break up the overall calculation of the standard deviation into a few steps. We already mentioned how standard deviation and variance is calculated, so here we will focus on how we used variance and the previously computed standard deviation to get the new standard deviation. First, we had to perform some algebraic manipulation on the equation in Figure 2 in order to write it in such way that only uses one division operation. The modified equation can be seen in Figure 6 below. In the equation, $\sqrt{V}$ represents the new standard deviation, $\hat{\sigma}$ represents the old standard deviation, and $V$ is the variance.

$$\sqrt{V} = \frac{\hat{\sigma}^2 + V}{2\hat{\sigma}}$$

Figure 6: Modified equation used to calculate standard deviation

So, to calculate standard deviation, we only need the previous standard deviation and the variance. This equation also has only one division, so it will provide reliable results. Figure 7 below illustrates how calculating *AVG_SD* uses the variance and previous standard deviation. It takes one clock cycle to calculate *AVG_SD* because one clock cycle is spent computing variance and the previous standard deviation.

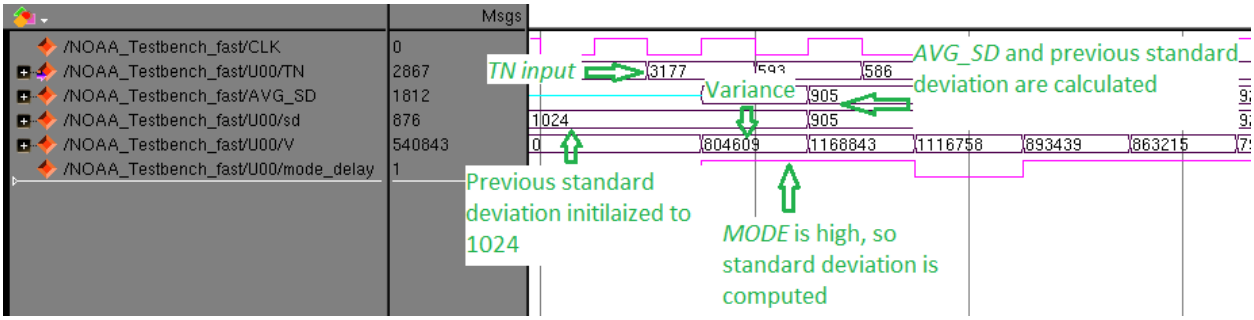


Figure 7: *AVG_SD* calculated based on variance and previous standard deviation

Figure 8 on the next page shows the output when we simulated our code with the provided test bench. These results were observed using the provided text file with 30 inputs. The results in the following figure are not the same as the results in the other figures in this section. For the waveforms in the other figures, the text file with 100 inputs was used. Here, we decided to show the results of the 30 input text file because it can fit on one page, unlike the results from the 100 input text file. The results are as expected, but it cuts off the final result. We are not concerned about this because it says that our module passes all tests. You will also notice that there are five lines of blank data in the figure below. This is because we had *DONE* alternate between high and low every clock cycle. The test bench compares the computed *AVG_SD* with the expected value whenever *DONE* is high. For this reason, empty data is compared to empty data for a few clock cycles until actual values are computed and compared to expected values.

```
# Initiating NOAA IoT Motes Module Testing Phase!! Good Luck!!
# Reset: x TN:     x MODE: x DONE:           1 AVG_OR_SD:     x expectedOutput:     x
# Reset: x TN:     x MODE: x DONE:           1 AVG_OR_SD:     x expectedOutput:     x
# Reset: x TN:     x MODE: x DONE:           1 AVG_OR_SD:     x expectedOutput:     x
# Reset: x TN:     x MODE: x DONE:           1 AVG_OR_SD:     x expectedOutput:     x
# Reset: x TN:     x MODE: x DONE:           1 AVG_OR_SD:     x expectedOutput:     x
# Reset: x TN:     x MODE: x DONE:           1 AVG_OR_SD:     x expectedOutput:     x
# Reset: 0 TN:  1590 MODE: 1 DONE:           1 AVG_OR_SD:   512 expectedOutput:   512
# Reset: 0 TN:  2313 MODE: 0 DONE:           1 AVG_OR_SD:  1952 expectedOutput:  1952
# Reset: 0 TN:  2804 MODE: 1 DONE:           1 AVG_OR_SD:   499 expectedOutput:   499
# Reset: 0 TN:  3003 MODE: 0 DONE:           1 AVG_OR_SD:  2428 expectedOutput:  2428
# Reset: 0 TN:  1468 MODE: 1 DONE:           1 AVG_OR_SD:   635 expectedOutput:   635
# Reset: 0 TN:  1138 MODE: 1 DONE:           1 AVG_OR_SD:   702 expectedOutput:   702
# Reset: 0 TN:   994 MODE: 1 DONE:           1 AVG_OR_SD:   747 expectedOutput:   747
# Reset: 0 TN:   433 MODE: 1 DONE:           1 AVG_OR_SD:   857 expectedOutput:   857
# Reset: 0 TN:  2416 MODE: 0 DONE:           1 AVG_OR_SD:  1795 expectedOutput:  1795
# Reset: 0 TN:  1691 MODE: 0 DONE:           1 AVG_OR_SD:  1785 expectedOutput:  1785
# Reset: 0 TN:  1037 MODE: 1 DONE:           1 AVG_OR_SD:   820 expectedOutput:   820
# Reset: 0 TN:  2571 MODE: 0 DONE:           1 AVG_OR_SD:  1756 expectedOutput:  1756
# Reset: 0 TN:  1916 MODE: 0 DONE:           1 AVG_OR_SD:  1667 expectedOutput:  1667
# Reset: 0 TN:  2447 MODE: 0 DONE:           1 AVG_OR_SD:  1611 expectedOutput:  1611
# Reset: 0 TN:  3054 MODE: 0 DONE:           1 AVG_OR_SD:  1770 expectedOutput:  1770
# Reset: 0 TN:    56 MODE: 0 DONE:           1 AVG_OR_SD:  1662 expectedOutput:  1662
# Reset: 0 TN:  1416 MODE: 1 DONE:           1 AVG_OR_SD:   931 expectedOutput:   931
# Reset: 0 TN:  2242 MODE: 0 DONE:           1 AVG_OR_SD:  1885 expectedOutput:  1885
# Reset: 0 TN:   614 MODE: 1 DONE:           1 AVG_OR_SD:   889 expectedOutput:   889
# Reset: 0 TN:   494 MODE: 0 DONE:           1 AVG_OR_SD:  1585 expectedOutput:  1585
# Reset: 0 TN:  1404 MODE: 1 DONE:           1 AVG_OR_SD:   947 expectedOutput:   947
# Reset: 0 TN:  1766 MODE: 0 DONE:           1 AVG_OR_SD:  1541 expectedOutput:  1541
# Reset: 0 TN:   732 MODE: 1 DONE:           1 AVG_OR_SD:   915 expectedOutput:   915
# Reset: 0 TN:  1104 MODE: 0 DONE:           1 AVG_OR_SD:  1288 expectedOutput:  1288
# Reset: 0 TN:   452 MODE: 1 DONE:           1 AVG_OR_SD:   684 expectedOutput:   684
# Reset: 0 TN:  2204 MODE: 1 DONE:           1 AVG_OR_SD:   643 expectedOutput:   643
# Reset: 0 TN:    65 MODE: 1 DONE:           1 AVG_OR_SD:   733 expectedOutput:   733
# Reset: 0 TN:  1940 MODE: 1 DONE:           1 AVG_OR_SD:   686 expectedOutput:   686
# Reset: 0 TN:    45 MODE: 1 DONE:           1 AVG_OR_SD:   744 expectedOutput:   744
# NOAA IoT Motes MODULE PASSED ALL TESTS!! CONGRATULATIONS!!
```

Figure 8: Results from simulation with the 30 input text file

# Schematic and Layout

Once we verified that our code works as we intended it to, we synthesized it in the RTL Compiler to generate the schematic. We then turned the schematic into a layout using Encounter.

## Schematic

This section will discuss the generation of the circuit using the RTL compiler. Figure 9 below shows the overall schematic view.

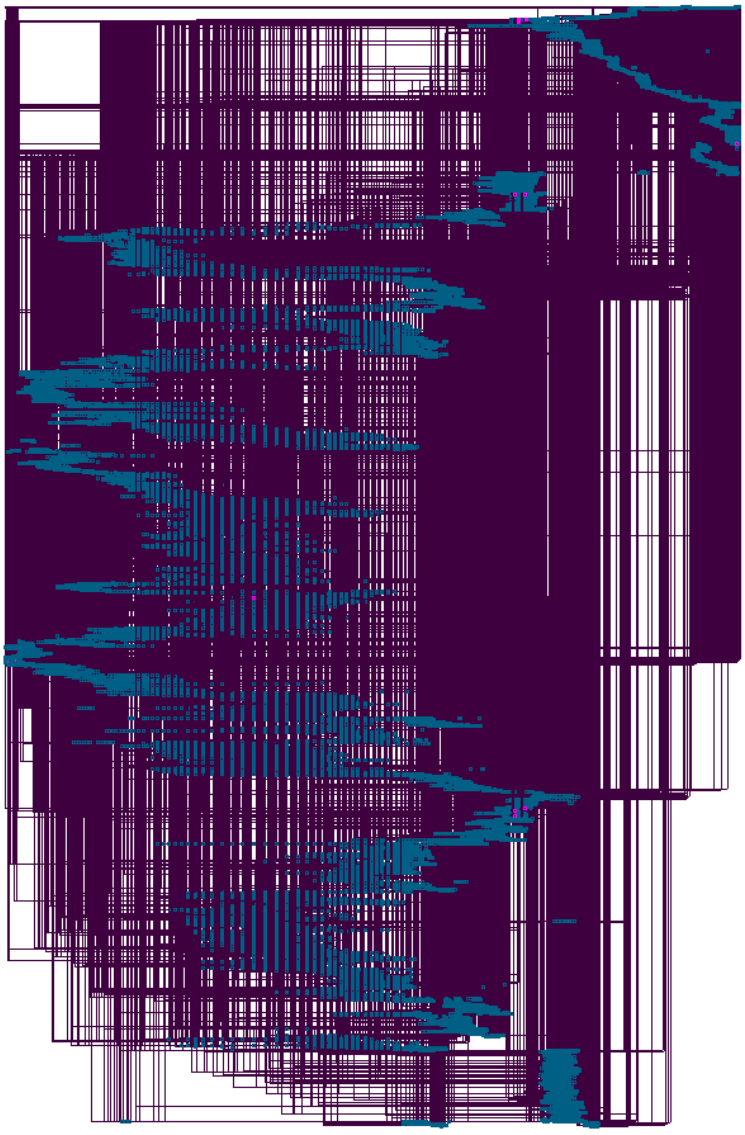

Figure 9: Schematic generated by the RTL compiler

Once we generated the schematic, we were able to report timing, area, and power. Using these three reports, we were able to find the slack, average power consumption, latency, and throughput. Table 2 below summarizes this data.

| | |
|---|---|
| Clock period (ns) | 7.5 |
| Slack (ns) | 0 |
| Area ($\mu m^2$) | 73822 |
| Average power consumption (mW) | 0.0002875 |
| Latency (clock cycles) | 2 |
| Throughput | 50000000 |

Table 2: Summary of Design post-RTL synthesis

The average power consumption was calculated by dividing the total reported power by the total number of cells. The total power was 8.3966 mW, and the total number of cells was 29,210. The latency is 2 because that is how many clock cycles the system requires to generate the output after *SAMPLE* is set to high.

In order to optimize our design, we wanted to decrease the energy consumption per temperature reading. By decreasing the clock cycle time, we will be able to decrease energy consumption. Awe changed the clock period to 6 ns and used the RTL compiler again to generate the schematic. Once again, we recorded data from the reports and placed it in Table 3 below.

| | |
|---|---|
| Clock period (ns) | 6 |
| Slack (ns) | 0 |
| Area ($\mu m^2$) | 97468 |
| Average power consumption (mW) | 0.000547 |
| Latency (clock cycles) | 2 |
| Throughput | 50000000 |

Table 3: Summary of data from optimized schematic

When comparing Tables 2 and 3, you can see that the area and average power consumption both increase. Although average power consumption increased, we are not concerned because we optimized energy consumption per temperature reading. Average power consumption depends on the number of cells and the total power. Both of these factors decreased, but number of cells decreased at a higher rate, so average power increases as seen in Table 3 above.

## Layout

To create the layout, we used Encounter. Figure 10 below shows the Encounter layout.

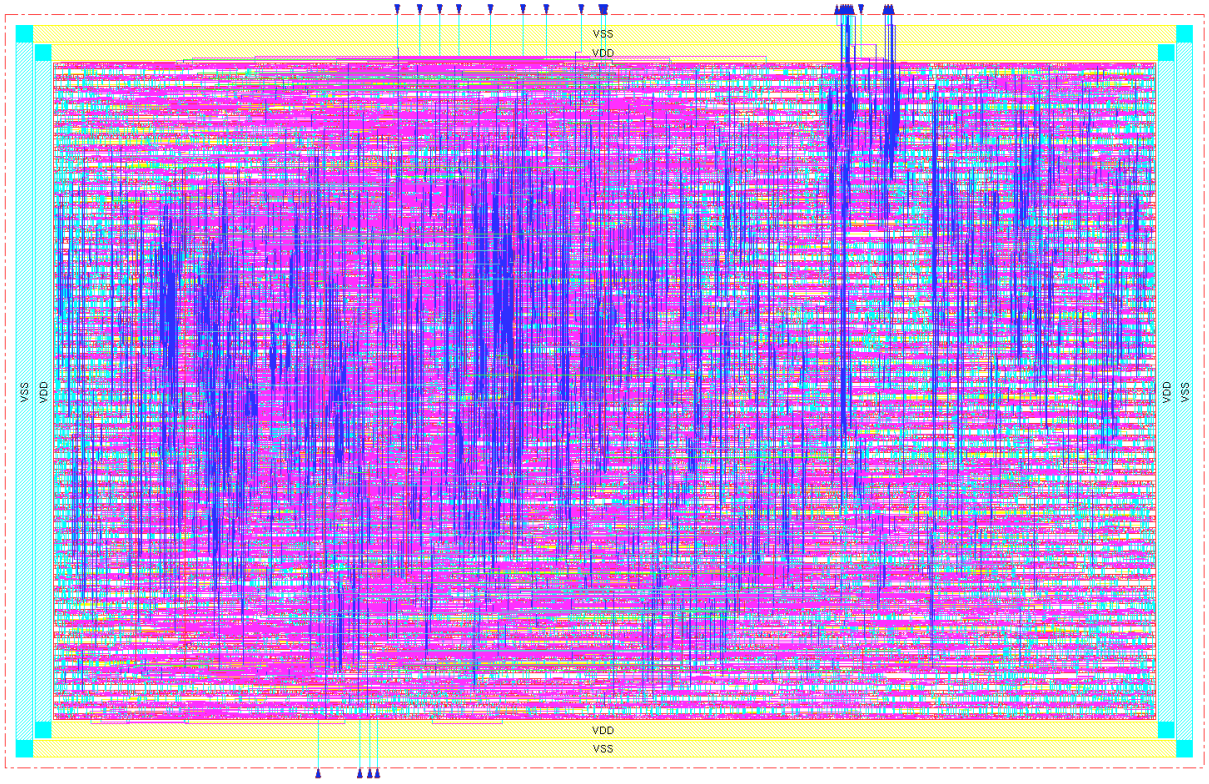


Figure 10: Encounter Layout

We also reported timing, area, and power as we did with the schematic. The summary of our results is in Table 4 below.

| | |
|---|---|
| Slack (ns) | 0 |
| Area ($\mu m^2$) | 73822 |
| Latency (clock cycles) | 2 |
| Throughput | 50000000 |

Table 4: Summary of data from Encounter layout

## Conclusion

The project was very interesting to work on since we were able to design a circuit that can be useful in many industries. Throughout the process we ran into several problems that took us a while to find a solution for. However, once we overcame each problem we learned a way to solve it for future reference. The project helped us understand much more about Verilog coding and synthesized circuits and about digital VLSI in the working environment. As soon as we started the analyzing process, we found that there were many problems that were never clarified in the introduction. To finish the given task, we developed our own understanding and assumptions of the problem and ways to solve it. Ultimately, our design in a very good degree satisfied all the required functions. Some uncertainties were resulted for the average and the standard deviation at the beginning, and this was primarily due to the fact that several division was operated to the numbers and the inputs (Samples), which had caused a little bit of uncertainty on the output. However, we have overcome this issue by rounding off the numbers using the method elaborated above. For the synthesized circuit, our Verilog simulation in the time domain was identical to the theoretical result. Overall, we really enjoyed this project and are quickly realizing what it is like to be an electrical engineer.