April 2009

Temperature Estimation Using Ring Oscillators

Gregory D. Pierre-Louis
Worcester Polytechnic Institute

Justin Wells
Worcester Polytechnic Institute

Project Number: BS2-0902

TEMPERATURE ESTIMATION USING RING OSCILLATORS

A Major Qualifying Project Report

submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

By

_____

Grégory Pierre-Louis

_____

Justin Wells

_____

Professor Berk Sunar, Advisor

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF EQUATIONS

ACKNOWLEDGEMENTS

This team would like to thank Prof. Berk Sunar for his direction and advising. We would also like to thank Mr. Brendon Chetwynd, Mr. Evan Custodio, and Mr. Gerardo Orlando for supplying us with our workspace and equipment and for tending to our requests. We thank Ms. Robyn Colopy and Mr. Jatin Chopra for their VHDL and MATLAB guidance. We would like to acknowledge Mr. David Houlette for answering our questions about Xilinx and FPGA's. Lastly, we would like to thank all of the employees of GD in Needham for providing us with a friendly work environment.

ABSTRACT

Partnering with C-4 Systems of General Dynamics in Needham, MA, this Worcester Polytechnic Institute Major Qualifying Project team explored the idea of designing a completely digital temperature sensor on field-programmable gate arrays. The goals of this MQP were (1) to find consistency between our research and results, and (2) to design a sensor capable of outputting a range of 0-70°C, a resolution of 0.1°C/count, and an error of ±1°C. Since propagation delay is dependent on temperature, we designed a ring oscillator out of logical inverters and counted the number of set clock periods to measure the length of the oscillator's total delay. We implemented our design and determined its measuring capability to be 20-70°C with an average resolution of ~0.13°C/count and an error of ±2.75°C.

INTRODUCTION

Partnering with C-4 Systems of General Dynamics in Needham, MA, this Worcester Polytechnic Institute (WPI) Major Qualifying Project (MQP) team explored the idea of measuring temperature on field-programmable gate arrays (FPGAs) without using any on-board temperature sensors. Power losses can occur in the form of heat because of manufacturing imperfections and more than often lead to errors in the functionality of the chip or unexpected results due to the increase in temperature. Overheating is becoming a serious concern as more densely packed transistors within the chips are generating more heat per unit area. The advantages of using a digital sensor over an analog sensor are that less chip size and power is used, there is no permanent use of any of the FPGA elements, and it will be flexible enough so any FPGA may use with minor adjustments.

The demand for small sized, high accuracy chips has grown over time, and with that, the demand for low consumption smart temperature sensors has also grown. Also, demand for these sensors to consume less power has grown as the desire to prevent internal overheating and also improve battery life has grown[1].

Developers of smart sensors have become more concerned with increasing power consumption when increasing resolution, accuracy, range, or die size of a chip. Consumers on the other hand have become more concerned with the decreasing resolution, accuracy, range, or die size when buying cheaper sensors. Seemingly, the higher the power consumption, the more expensive the sensor become. Along with increased power consumption, cost of manufacturing FPGAs with on-board sensors also increases [1].

Much research has been done to turn away from analog-to-digital converters (ADC's) which occupy much of the chip size. Recent technological focus has been geared towards

completely digital sensors with use of a time-to-digital converter (TDC) to produce the digital output. In the literature we found one particular digital sensor that consumed 8.42μW of power with a range of 0-75°C, a resolution of less than 0.1°C, and measurement error of ±1.5°C and [1].

Many of the designs make use of a time domain temperature sensor, a timing reference, and a TDC. The time domain temperature sensor is a thermally sensitive digital oscillator or pulse generator which is based on the fact that the propagation delay of logic gates increases with time. The timing reference is simply the on-board clock and the TDC being a simple counter. The number of oscillations or pulses that the counter counts varies with temperature. With this count, we can create a series of temperature curves and create an approximate count-to-temperature conversion table. From this table, as opposed to outputting a count, one should be able to create a look-up table (LUT) to directly output the die temperature.

The goal of this MQP was two tiered: (1) to verify the theoretical design with practical experiments, and (2) to design a sensor capable of outputting a temperature with a range of 0-70°C – the typical commercial range, a resolution of 0.1°C/count, and an error of ±1°C. The sensor would be integrated into the chip for temperature monitoring and has the potential to serve as a way to minimize the risk of overheating and damaging the FPGA.

BACKGROUND

In this chapter, we discuss many of the issues we encountered in the course of our MQP, such as the general concept of FPGAs and how temperature relates to propagation delay.

Previous Work

Although we did much research in digital sensors, this particular piece contained much of the information we would use to base our design. The article describes a project similar to ours, so our initial steps in the MQP were to attempt to duplicate their project as much as we could and analyze temperature in a similar way [1].

FPGAs provide a great platform for this implementation. FPGAs yield high performance at relatively low cost, consume a relatively small area, and most importantly are reprogrammable. In the past, smart temperature sensors have been primarily constructed using analog-to-digital converters (ADCs). ADCs account for a large chip area, high power consumption, and often come with a limited temperature range. In this project, a time-to-digital converter (TDC) was used in place of the analog-to-digital converter. The TDC was combined with a pulse generator to produce a digital output. The chip size was reduced to nearly one tenth of that of the typical ADC. Furthermore, the power consumption was reduced to as low as 10 microwatts.

A simple counter would suffice as a TDC, as long as the pulse generator created a wide enough pulse. Using a reference clock of 100 MHz, the width would need to be tens of nanoseconds for an accurate reading.

Using the equations from [1] shown below, it was easier for the group to analyze the relationship between propagation delay and temperature.

$$T_D = \frac{(L/W)C_L}{\mu C_{OX}(V_{DD} - V_T)} \ln\left(\frac{1.5V_{DD} - 2V_T}{0.5V_{DD}}\right)$$

Equation 1 – Propagation Delay for the NOT gate

Mobility of holes and electrons, μ, is the relation of the speed of holes and electrons to an applied electric field. The mobility μ is expressed in Equation 2.

$$\mu = \mu_0 \left(\frac{T}{T_0}\right)^{km}, \qquad km = -1.2 \sim -2.0$$

Equation 2 – Mobility

The voltage across the insulating layer and the substrate of the transistor is called the threshold voltage, V_T. This is affected by temperature as seen in Equation 3.

$$V_T(T) = V_T(T_0) + \alpha(T - T_0), \qquad \alpha = -0.5 \sim -3.0\,\text{mV/°K}$$

Equation 3 – Threshold Voltage Temperature Dependency

Looking at these equations, one can see that as temperature increases, both mobility and threshold voltage decrease. In the situation in which V_{DD} is much higher than the threshold voltage (V_T), it is seen that the higher the temperature is, the longer the propagation delay. This allowed for an easier creation of a delay line using a series of *N* inverters given the desired length, which would need to be quite long to get a reading for the desired temperature range.

FPGA's

Field-programmable gate arrays fall under a class of devices that are labeled "reconfigurable" or "reprogrammable" hardware. FPGAs are able to perform nearly any digital function they are programmed to realize. Application-specific integrated circuits (ASICs) are non-programmable and typically have one use. ASICs perform better than FPGAs but are much more expensive due to the customization capabilities. FPGAs are useful because they can be reprogrammed and reused respective to the task at hand for a cheaper cost with slight compromise in performance. Undesired functionality of the device can be fixed on field and non-recurring engineering costs are reduced [2].

Structure

FPGAs contain configurable logic blocks (CLBs) adjacent to two-dimensional arrays of wires. These wires are similar to roads on a map, running east to west and north to south. Each of these vertical and horizontal wires forms switches; and the intersection of these switches collectively create a programmable switch matrix (PSM). At a gate-level design, small sub-circuits are partitioned from the design to be assigned to a CLB. Connections are then established between the CLBs via PSMs. The hardware blocks around the outskirts of the chip are used to drive signals on and off the chip. These blocks are called input/outputs blocks (IOBs). It can then be said FPGAs are comprised of mainly two types of building blocks: CLBs and IOBs [2, 3].

CLBs are basically comprised of a RAM modules, which are referred to as a lookup tables (LUTs), D-flip-flops, and multiplexers to bypass the D-flip-flop. A LUT can perform any logic gate of four inputs and quite recently even six. Depending on the manufacturer, the blocks can be made to be more complex for efficiency and functionality by duplicating or adding more components. "The propagation delay through a LUT is independent of the function

implemented," [4]. Thus, meaning that no matter what the function is, the propagation delay of the LUT is the same as the delay of all functions. Furthermore, LUTs are truth tables. A truth table is a predefined list of outputs generated by every combination of inputs [2, 3].

Design Process

There are several ways to model a design into a CAD tool which will synthesize and produce a configuration bitmap to be downloaded into an FPGA. One way to enter a design is schematic capture. Hardware description languages (HDLs) are a great alternative to enter a design. The two most popular HDLs are Verilog and VHDL. This team will focus mainly on VHDL. The sequence of activities that is encompassed in FPGA implementation is as follows [2]:

1. *HDL Model* – This step refers to the actual coding process that proceeds interpreting the initial problem description which may be through state diagrams, truth tables, or a simple English description.
2. *Behavioral Simulation* – The coding must be simulated before synthesized on the FPGA to ensure that its functionality is properly implemented.
3. *Synthesis* – The design is then synthesized to a library of gates, flip-flips, latches, and perhaps other primitive components
4. *Functional Simulation* – Using the physical properties of the FPGA, timing information is acquired to simulate preliminary performance estimates.
5. *Place and Route* – The design is then mapped and placed; each primitive component is assigned to a specific CLB on the FPGA. The connections are then routed via PSMs. Accurate, more so then the functional simulation, timing information can be

obtained – this is often referred to as verification. The configuration bits or bit stream are then produced.

6. *Programming* – The bit stream is loaded into the chip to be configured to implement the design.

Propagation Delay vs. Temperature

Propagation delay is the amount of time it takes for a logic gate to switch to the correct output after any of its inputs have changed. Delay of transitioning from a logic 1 to a logic 0 may be different from the delay transitioning from a logic 0 to a logic 1. Also, depending on the quality of the components of the gates manufactured, propagation delay may vary. Furthermore, propagation delay may attribute to glitches in digital components as logic blocks may not change faster than others resulting in an undesired output. For the primitive gates, p-channel transistors are used to control the sourcing current. In contrast, n-channel transistors are used to control the sinking current. P-channel transistors when in series or parallel can be combined into an equivalent p-channel transistor. The same follows for an equivalent n-channel transistor. Usually, the p-channel transistor is a pMOS transistor and the n-channel channel is an nMOS transistor [1].

Ring Oscillator

A ring oscillator is a chain of an odd number of NOT gates whose output is connected back to the input. The output of the chain would change when the previous output is inverted due to the odd number of inverters after the gates' propagation delays. When sufficiently many gates are used it is possible to generate a near square wave pattern. An even number of gates would not

cause an oscillation since two or any other even number of inversions of a signal would cause an unchanged output. A schematic of a basic 3 inverter ring oscillator can be seen in Figure 1.

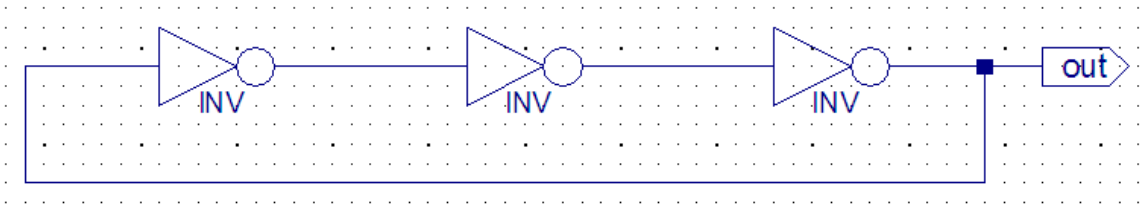

Figure 1 – A 3-Inverter Ring Oscillator

The frequency of the oscillator is shown below in Equation 4 with N being the number of gates and τ being the propagation delay of one gate, assuming that the delays of all the gates used are the same.

$$f = \frac{1}{2N\tau}$$

Equation 4 – Approximate Frequency of a Ring Oscillator

As discussed in the previous Section and captured in Equation 4 higher temperatures result in larger propagation delayshigher temperatures result in smaller frequencies. If a method could be devised to count the amount of times the oscillator fluctuates, one could theoretically relate the count to temperature.

TESTING TOOLS

Many programs and software tools are required for interfacing, simulation, testing, and eventually programming of a development board. We used these tools, along with some Xilinx tools, as stepping stones to a final temperature sensor design.

FPGA Development Board

Upon starting our project with General Dynamics, we were provided with a Virtex-5 ML505 development board – specifically with the XC5VLX50T device with a speed of -1 and the FF1136 package. The package provided many features, most of which we would not require during our design. The board came equipped with a Virtex 5 FPGA, which would eventually become our programming device to measure temperature.

JTAG

We used the JTAG interface when trying to connecting our board to the computer and establishing a method of sending and receiving information was one of the first steps in our design process. The JTAG chain starts at the connector and runs through the series of PROMs, CPLD, ACE controller, an FPGA board and an expansion card if necessary. Our main use would be programming and accessing the FPGA using the JTAG connection. However, it also allowed for us to incorporate the analysis tool Chipscope which we will discuss later in this section.

System Monitor

Along with the many functions available on the Virtex-5 platform, the FPGA comes with a block known as System Monitor [5]. System Monitor is located at the center of the die, and is developed around a 10-bit, 200-kSPS Analog-to-Digital Converter (ADC). Using other on-chip sensors, the ADC can measure operating characteristics such as on-chip power supply voltages and die temperatures. There are also 16 user-selectable external inputs, which allows monitoring

of the external environment of the chip and entire board. The system also allows for the user to set constraints on voltage and temperature, as well as external inputs, to set off alarms based on parameters going either above or below the desired constraints. The System Monitor also is able to implement averaging of desired output values, either 16, 64, or 256 times.

The System Monitor does however have an error of ±4°C. Noticing on the chart that 1 LSB (least significant bit) correlated to around .49 °C, and that a count of 0 correlated to -273 °C, we created the equation below to relate our count to an appropriate output temperature.

$$Temperature = Count * 0.49 - 273, \text{ where Temperature is expressed in °C}$$

Equation 5 – System Monitor Count-to-Temperature Equation

We multiplied our count by the .49 °C for the LSB, then subtracted the 273 °C representing a count of zero to create out temperature output in MATLAB. We used the System Monitor as the reliable source of temperature considering other such measurement sensors and devices are less accommodating for automation.

Applications

Xilinx ISE 10.1

Xilinx ISE 10.1 was our main programming tool used throughout the project. The software allows the user to develop, simulate, and eventually program in a variety of methods. Our main use of the Xilinx ISE was for VHDL coding as well as simulation and schematic development. The Xilinx tool is designed to maximize connectivity and programming speed between the computer and the Virtex 5 FPGA. Xilinx ISE 10.1 allowed for us to easily write and

simulate VHDL for our project, and connect easily and program to our FPGA via the JTAG chain [3].

Our programming language throughout this project was VHDL. The program was created by the United States Department of Defense to assist in the development of high-speed integrated circuits. The language was changed in 1987, then again in 1993, and updates have been taking place ever since, all adhering to IEEE standards. VHDL is a commonly used programming language for FPGAs.

ChipScope

ChipScope is a software tool that comes with the Xilinx ISE 10.1 package, and is essentially like having an oscilloscope that analyzes cores designed onto the chip. The tool analyzes logic and allows the user to view any internal FPGA signal or node, which includes embedded processors. The signals are analyzed via the JTAG chain and analyzed through the ChipScope Pro Logic Analyzer tool. The ChipScope tool was another method used to monitor on-chip temperature. We used ChipScope for debugging, initial informal testing, and verification of our implementation of the System Monitor [6].

METHODOLOGY, TESTING, AND ANALYSIS

Ring Oscillator vs. Pulse Generator

We attempted to design a sensor using the generator, with little success. In the initial steps of our project we had discussed the creation of a ring oscillator, which would serve as a delay line to allow for the measurement of temperature. Our ring oscillator design was going to be a simple series of inverters; however the Xilinx tool removes inverters from a design during synthesis, so this added a new challenge to the task. We found a coding method that would allow for the synthesis of inverters, and after trying both a schematic and VHDL approach to the design, we decided that the coding was much more practical, as with the large number of inverters being used the program was finding it difficult to handle the design from a schematic standpoint.

After creating our ring oscillator, we soon ran into another issue. During the Xilinx testing phase, we realized our design experienced some degree of metastablility. Metastability occurs when an output is neither 0 nor 1 at a specific time, and is essentially fluctuates in a random way between the two stable states. It can be compared to a ball rolling up a hill, with 0 being at the bottom of the hill, and 1 being on the other side. However, the ball does not have enough momentum to get all the way over the hill, and is stuck at the top. That is essentially the idea of metastability. We noticed this problem in our testbench waveformsWe overcame metastability by simply inserting two groups of 2 flip-flops into our design.

While we have chosen to design our circuit with the ring oscillator as opposed to the pulse generator, the pulse generator is also a useful option for this design since it has less jitter. It

might have actually made the design easier, however we felt more comfortable with the oscillator circuit.

Initial Testing and Design

During the first stages of our MQP when we determined that a ring oscillator would be an appropriate delay circuit, the question that needed to be answered was how many inverters we would need within the oscillator. In [1], the team was able to develop a circuit with a resolution near .1 degree Celsius per count with 140 inverters.

When we began testing we found that we were only receiving a reasonable resolution with an extremely large number of inverters, around 4000. We knew this was not practical for a design, and also took too much time to simulate and implement using Xilinx. Also with such a high number of inverters, the chip temperature would be raised due to self-heating, which was a side-effect we were hoping to avoid. In order to help reduce the number of inverters, we created a cyclic counter. The counter counts the number of times the oscillator completes a full cycle. This number can be changed within our code to increase or decrease the resolution. We then output the total number of clock ticks that our master counter sees between the given numbers of cycles. The cycle counter dramatically helped us reduce the number of inverters in our circuit.

Informally, we seemed to have reached our desired resolution of 0.1ºC/count with multiple inverter and count combinations, as shown in Table 1. According to Equation 4, assuming a propagation delay of 0.238ns [4], the frequency of an oscillator with only 21 inverters is 100.04MHz. Since this is faster than our 100MHz master clock, it would cause synchronization issues and our sampling would not be accurate. The fewer inverters in the ring, the higher the oscillation frequency. We chose not to test rings with less than 21 inverters.

Inverters	Cycles	Count (16 bit)	Resolution
421	256	8100	.7 deg/count
421	512	16,200	.4 deg/count
421	1000	33,400	.16 deg/count
41	1000	4,300	1 deg/count
41	4096	16,700	.49 deg/count
41	8,192	31,500	.33 deg/count
100	10000	**OVERFLOW**	
41	16,000+	**OVERFLOW**	
51	10000	41,000	.16 deg/count
51	12345	49,100	.12 deg/count
51	13579	56,200	.086 deg/count
25	15000	33,500	.15 deg/count
4200	16	900	

Table 1 – Initial Inverter-Cycle Results

Our overflow problems were easily fixed when we began to use a 32-bit counter rather than 16-bit. From this test, we determined our optimal number of inverters to be 51 since it seemly gave us an appropriate resolution for our application and as one can realize from Table 1, the higher the count, the better the resolution. We did our formal testing using 51 inverters and a cycle count of 16,384 yielding an outputted count of ~70000.

Temperature Chamber Testing

We were granted access to a laboratory equipped with a temperature chamber capable of providing a stable temperature within our desired range (0-70ºC). The user is able to input the

desired temperature, and the machine will either heat up or cool down until that point is reached. We placed our board inside, and ran our JTAG and RS232 cables out to a computer.

We decided that in our first test we would sweep through an input range on the chamber of -15 to 70°C. During our first half a dozen tests we saw unexpected results. We determined that we had lowered the temperature of the board too much and neither our program nor the system monitor was behaving properly. Once we reached an input temperature of around 15-20 °C, both the system monitor and our design began running smoothly. From -15 to 55 °C we let our design rest for 2 minutes to allow the chamber to heat up to the desired input. From 60-70 °C we let the design rest for 3 minutes as the hotter the desired temperature, the longer the chamber took to heat up. As previously mentioned, from 20-70°C our design appeared to operating with the least error, so in the ensuing tests we did not set the temperature any lower than 0°C.

In our next test we ran our design continuously. We allowed the temperature sensor to cool to 0°C, and then set it for 70 and ran our design. The sensor took 16 minutes to heat up to 70°C, at which point we stopped running our design. We repeated this test going from 70°C down to 0, which took around 16 minutes as well. Both produced similar and accurate results.

In our final test of the day we let the chamber again settle at 0°C, and then every minute increased the temperature by 5°C until it reached 70, with our design again continuously running in the background.

In these 4 tests we were able to see excellent results, matching perfectly to our expectations. We recorded well over 1000 data points for analysis.

Analysis

When comparing the System Monitor temperature output to our count, results show that there is indeed a direct proportionality between temperature and propagation delay. Although we

expected a linear relationship, we found that there was some exponential curving and also relatively more noise located at the lower values of our count. We call this effect "eyebrowing" since the features of our curve are similar to that of a human eyebrow as seen below.

Count-to-Temperature Conversion

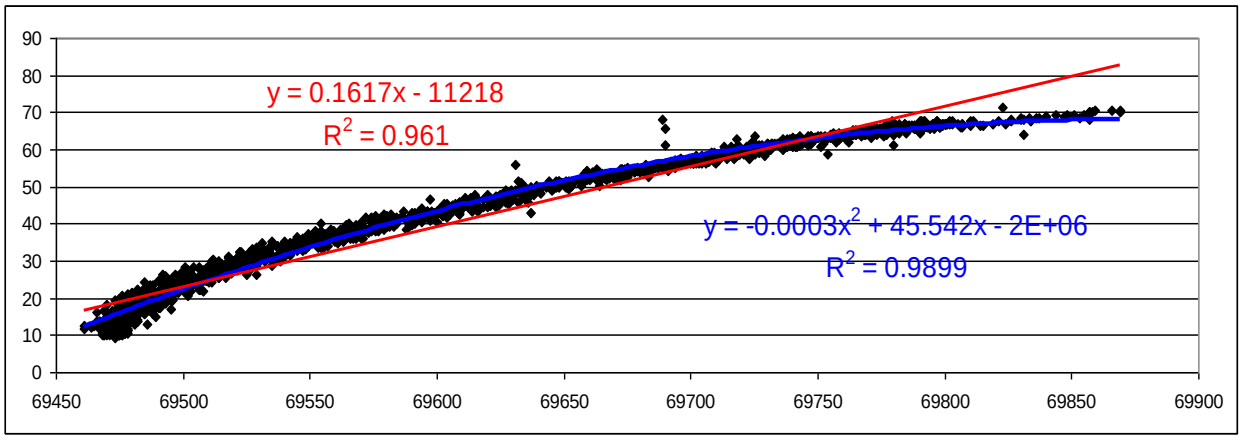

Figure 2 – Measured Temperature vs. Count, "Eyebrowing" Effect

In Figure 2, the x-axis represents our outputted count and the y-axis represents the System Monitor temperature output. Each data point represents one of our counts corresponding to temperature. The red trend line is a one-termed line with a determination coefficient (R^2) of 0.961 while the blue trend line is a two-termed line with a R^2 value of 0.9899. The determination coefficient is used to observe the variance of data. In this context, the closer R^2 is to 1, the more likely it is to fit the trend line.

If we were to use the linear approximation, we would expect, as seen in its equation of the line, a total resolution of 0.16°C/count. Below is a chart expressing the count range, and the resolution and R^2 value of one-termed trend lines generated between 10°C margins.

Temperature Range (°C)	Count Range	Resolution (°C/count)	R^2
<20	69461-69496	0.2852	0.4490
20-29	69475-69538	0.1880	0.7309
30-39	69512-69590	0.1467	0.8296
40-49	69554-69647	0.1295	0.8350
50-59	69631-69754	0.1107	0.9024
>59	69689-69869	0.0689	0.8173

Table 2 – Resolution Table

Looking at the table above, we notice that as temperature increases our resolution and accuracy also increase. Looking back, our goal was to design a sensor capable of outputting a resolution of .1°C/count. Our current design is able to output such a resolution, but only in the in the range 20-70°C. Below is a plot of our range.

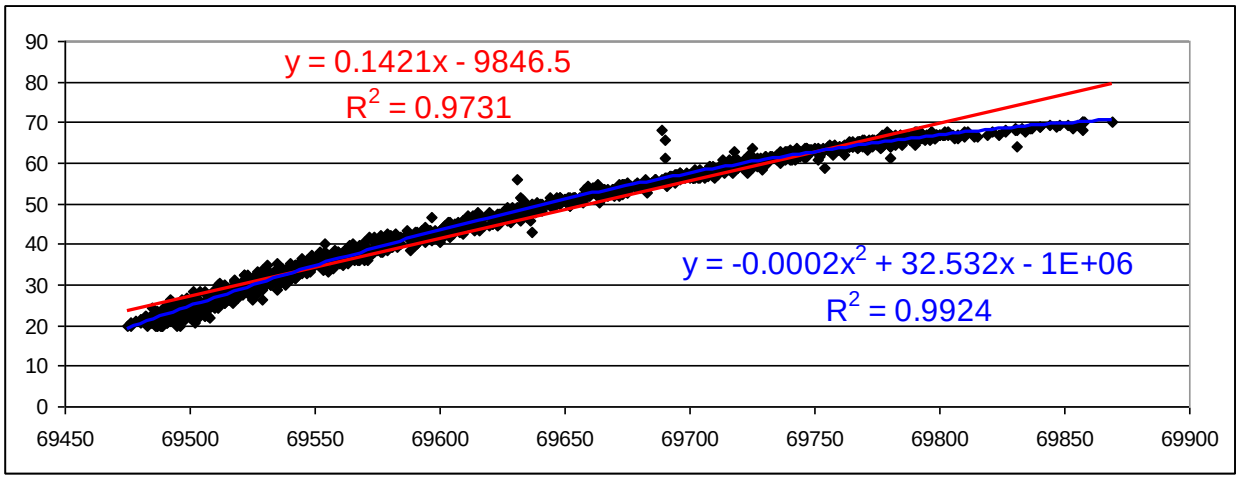

Figure 3 – Measured Temperature vs. Count, 20-70°C

In the figure above, we notice that the eyebrowing effect is somewhat less dramatic and both the resolution of the one-term trend line and the R^2 of both trend lines have improved. Below is a plot expressing the calibrated count generated by subtracting 69420 from our outputted count.

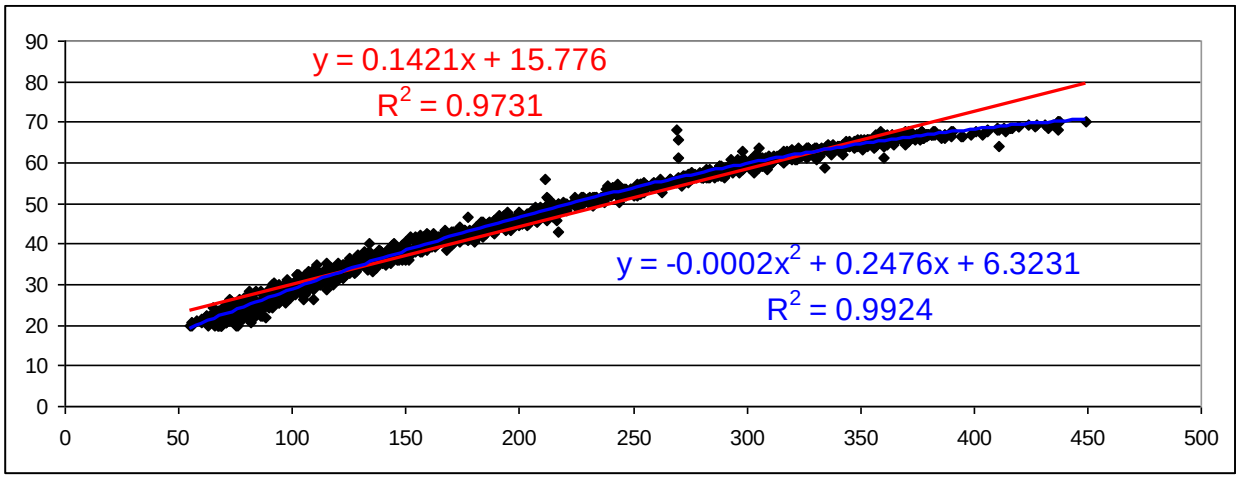

Figure 4 – Measured Temperature vs. Calibrated Count, 20-70°C

Above, we see that the magnitude of the intercepts used in the line equations have dramatically decreased. Also, the quadratic approximation seems to be a near fit of the recorded curve. If we attempted to use the linear approximation, we would find that calibrated counts above 350 would become increasing inaccurate. Below is the equation that expresses our count-to-temperature conversion, CTC, using the quadratic approximation equation.

$$CTC = -0.000234*(N-69420)^2 + 0.2476*(N-69420) + 6.3231, \text{ where N is the count}$$

Equation 6 – Count-To-Temperature Conversion Equation

Using our calibrated recorded counts as the independent variable, we find that the resulting calculated temperature vs. count curve closely resemble that of our measured temperature vs. count curve as seen below.

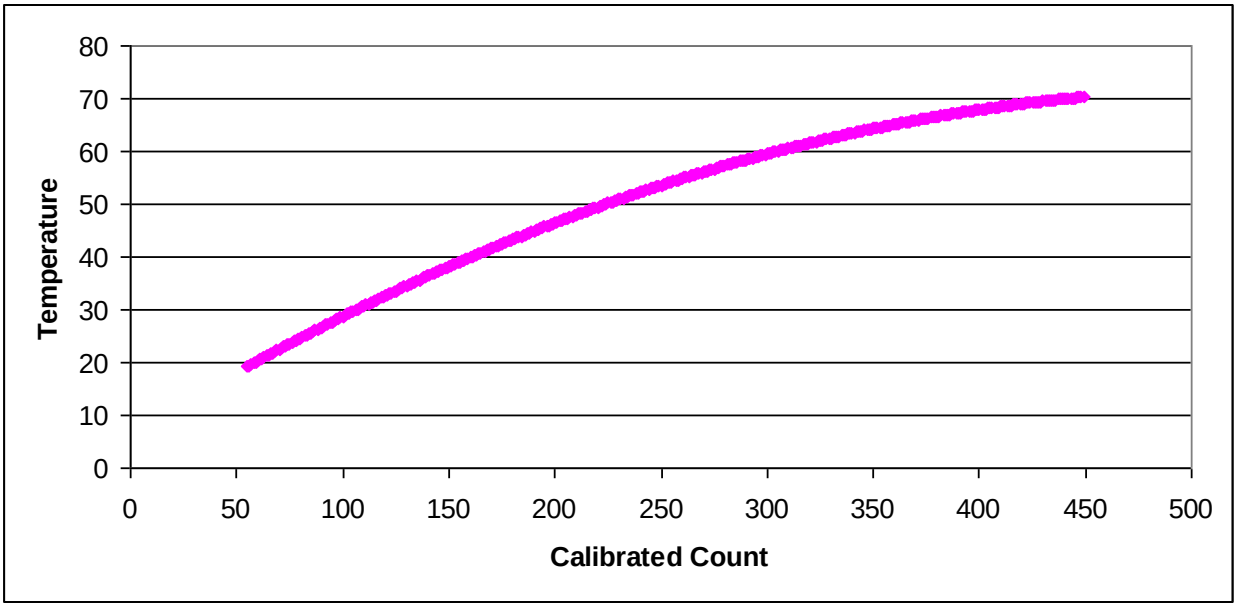

Figure 5 – Calculated Temperature vs. Calibrated Count, 20-70°C (using Equation 6)

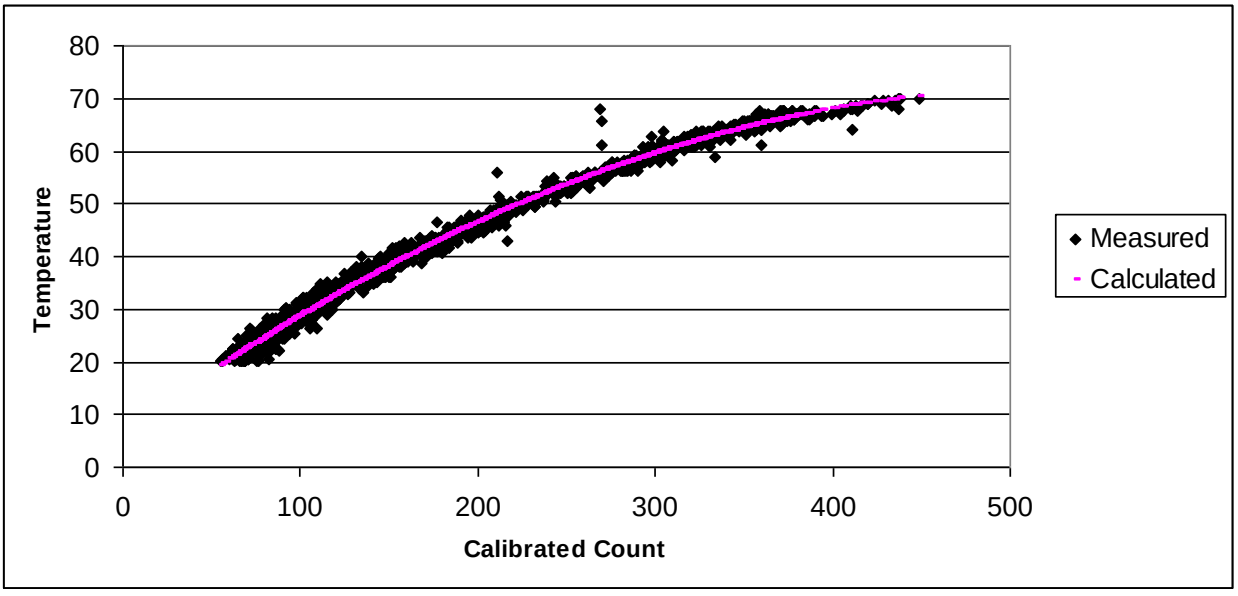

Figure 6 – Measured and Calculated Temperature vs. Calibrated Count, 20-70°C

Sweeping through calibrated counts of 55, our lowest recorded calibrated count for 20°C, and up to 449, our highest recorded calibrated count at 70°C, we find that the difference in the

resulting temperatures between adjacent counts slowly decreased as the count increased. We also find that with the calculated values, we have the worst case resolution of ~0.22°C/count at the lowest recorded count at 20°C, and we have the best case resolution of ~0.039°C/count at the highest recorded count at 70°C. Averaging all of the differences, we in addition find that the mean resolution of our calculated temperature is ~0.13°C/count.

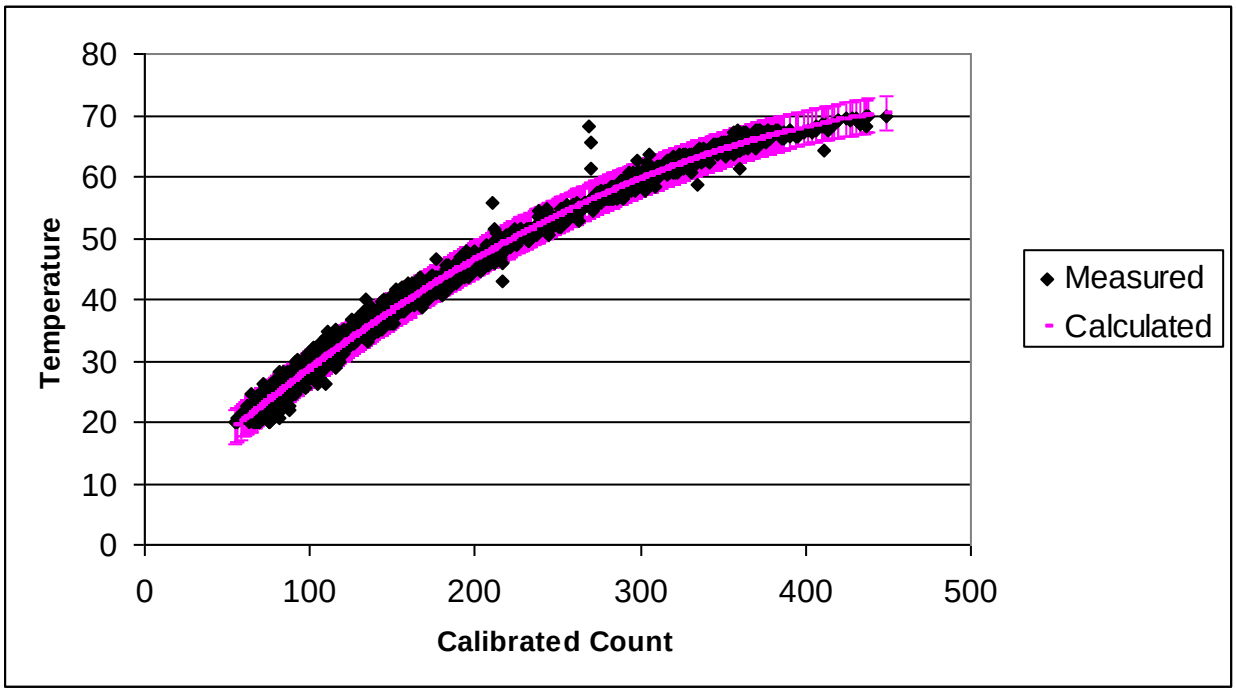

Figure 7 – Measured and Calculated Temperature vs. Calibrated Count, 20-70°C, with ±2σ Error

Our calculated curve generates a max error of ±12°C when referencing our recordings. However, we calculated that the 95% of recorded values fall within ±2.75°C of our calculated temperature. We found this by calculating the mean and standard deviation σ of the error between our measured and calculated temperature values. While the mean of our error is 0.945°C, σ turned out to be 0.905°C.

Realistically, this team would prefer to use the same amount of output bits as the System Monitor. This is plausible since the temperature would output at the most 3 digits which would only require 10 bits. The truth table in which count, calibrated count, actual temperature, and output temperature is located in Appendix C.

Ring Oscillator Delay

Empirically, there must be a way to calculate the propagation delay through the inverters used in the ring oscillator by using the count and our design specifications. Recall that our design basically works by counting the number of clock ticks between the oscillation of a pulse coming from a 51-inverter ring oscillator. Knowing that we use a circulation counter of 2^{14} and a master clock of 100MHz, we can determine the propagation delay with respect to the count. The equation below expresses mean delay through one inverter.

$$\tau_{P\,individual} = \mathrm{T}(\frac{N}{2C}) * \frac{1}{K},$$

Equation 7 – Mean Delay through One Inverter, $\tau_{Pindividual}$

In Equation 7 T is the period the master clock, N is the outputted count, C is the circulation count, and K is the number of inverters. According to this equation, an outputted count of 70000 indicates a delay of 0.419ns on average for every inverter, $\tau_{Pindividual}$. Further investigation is called on since we postulated that the delay through each inverter delay should be ~ 0.238ns. A simpler equation is expressed below to represent the delay through the ring of inverters.

$$\tau_{P\,ring} = \mathrm{T}(\frac{N}{2C}),$$

Equation 8 – Delay through the Ring, τ_{Pring}

With a count of 70000 (N), we find that the approximate average delay of one circulation τ_{Pring} is 21.36ns. Considering that the master clock allows a clock tick of 10ns (T) and the foundation upon which our count is based on supports one instance of a signal propagating through the ring oscillator, we find that accurately approximating the delay is difficult and imprecise. The delay found in Equation 8 is timed using the 100MHz clock. According to Equation 8, 51 inverters signify a delay of approximately 21.36ns. Each inverter delay should be ~0.238ns, thus 51 inverters should have yielded a delay of ~12.14ns – nearly 9ns of error.

An obvious solution would be to reference a faster clock if accessible. Another solution to this problem is redesigning the ring oscillator to be larger than 51 inverters. For example, 511 inverters would nearly increase the delay 10 times. Theoretically, increasing the ring delay would inversely decrease the error of counting. A number of 511 inverters would result in a delay of ~121.6ns. Using 10ns grades, miscounts – both under and over counts – will have less of an impact. Not only would the accuracy increase, but the resolution would as well. Also, average individual inverter delays would be more predictable. However, an immediate disadvantage to this is the expanded ring would require more chip space. Furthermore, the ring will consume more power.

CONCLUSION

Drawing from the analyses we formulated through our results, we found that there is indeed a proportional correlation between rise in propagation delay and rise in temperature and thus succeeded in completing goal (1). Using this property we designed a digital temperature sensor by counting the number of clock periods between the cycles of a ring oscillator comprised of inverters.

However, we were slightly unsuccessful in completing goal (1) which expressed the desire to create a sensor with a resolution of 0.1°C/count, a range of 0-70°C, and an error of ±1°C. Instead, our results show our sensor has a range of 20-70°C, an average resolution of ~0.13°C/count, and an error of ±2.75°C. When comparing the results from [1], whose sensor had a range of 0-75°C, a resolution of less than 0.1°C, and measurement error of ±1.5°C, our sensor is outmatched.

Nonetheless, we formulated a conversion equation and thus made a CTC table (Appendix C) that could potentially be made into an LUT to be implemented in our design. Our design also uses 1% of the available resources on the FPGA as seen in Figure 8.

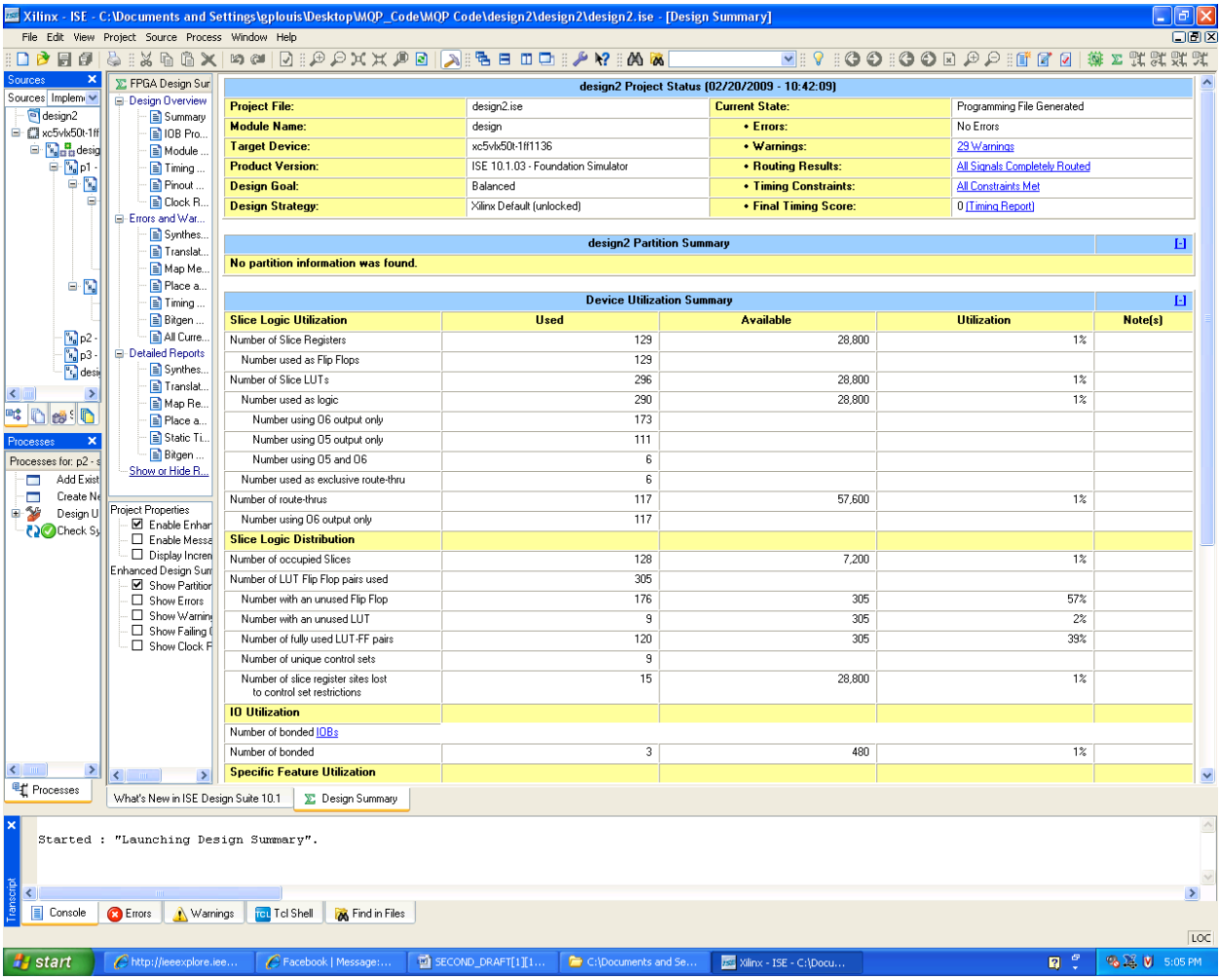

Figure 8 – Xilinx Device Utilization Summary Screen Shot

FUTURE RECOMMENDATIONS

Pulse Generator

Initially we intended to use a pulse generator in place of our ring oscillator/delay line. The pulse generator would create a temperature proportional width which we would measure and eventually output a temperature reading. In fact, for nearly half of our project we were under the impression that the pulse generator would play a key role in our design, only to find out later it would be an aspect soon forgotten. When we ran into difficulties implementing our design using the pulse generator even from a simulation standpoint, we knew that we could run into some serious problems later. We made the decision to do away with the pulse generator and create our ring oscillator using a given number of inverters. As we had put in many hours developing our pulse generator, this was quite the daunting task to simply remove it completely from our project. The idea behind the pulse generator seemed brilliant, as developed in [1]. As a team, along with our advisors, we agreed that a pulse generator would be a great design, however was not practical for our capabilities and the time constraints of our project. We believe that with more time and research, the pulse generator would be a great replacement of the ring oscillator and could even provide a better design with a more accurate resolution and output.

Temperature Chamber

During the course of our project our ultimate goal from the start was to test our design in a temperature chamber provided by General Dynamics. The chamber allowed for us to vary the temperature (0-75°C), corresponding to the restraints of our chip and board. The chamber was extremely useful in that it allowed for us to control the precise temperature and measure over a period of time at that given temperature. Unfortunately, we were only provided limited access to

the temperature chamber. We could only utilize the chamber to our benefit once our design was completely developed, which took nearly 5 weeks to complete. With only a 2 week window left, the chamber was only available for us to use for one full day. We took complete advantage and performed numerous tests, acquiring nearly 1500 data points. However, more time with the chamber would have allowed us to obtain more accurate results, and given us a chance to fine-tune the design. While 1500 data points is a good amount, we could have used a lot more to truly understand where our design was working successfully and also expose its flaws. With the need to provide written analysis of our testing with only one week remaining, combined with its limited availability, we were forced to settle for our one trip to the temperature chamber.

We also discussed the possibility of using multiple temperature measurement techniques. We had discussed the possibility of a portable temperature chamber, as well as a laser temperature control Another issue we encountered while using the temperature chamber was issues with both our design and the System Monitor at low temperature. The System Monitor operates at a temperature range of -40 to 125°C, and our chip and board itself can safely operate at a range of 0 to 70°C. In our first test, we set the temperature chamber at -15°C. The System Monitor was approximately 10°C off until we increased the chamber temperature to 15°C, at which point it began to output the correct temperature. Our design count was also not responding well to the changes in the temperature chamber until we reached around 15-20°C, at which point it began showing appropriate changes in count based on temperature. These issues could have occurred due to a number of issues. We suspected that when lowering the temperature chamber to -15°C in an attempt to see a reading of absolute 0 on the System Monitor, we exceeded the chip's temperature range and it took a while to return to an operating state. While the System Monitor User Guide states it will operate at temperatures as low as -40°C, the chip and board

itself cannot. Once the chip realized a constant temperature of over 0°C, our design as well as the System Monitor began outputting properly. We believe that improvements on resolution and accuracy would have been our next steps if we had been given more time for this project; however with our restrictions we feel our resolution was adequate.

Placement of the Ring Oscillator

Early in the project we had we considered the idea of directly implementing configurable logic blocks (CLB's). While we investigated the process, it was clearly not plausible within the timeframe of this project. There is a Xilinx product called PlanAhead that allows users to directly choose the CLB's desired for their design. Basically, if one wished to place our ring oscillator in any corner of the chip, one may use this application to physically drag the signal instantiations into the CLB's in the floor plan on the chip. Figure 9 shows a screenshot of the 51 inverters being moved from the center of the chip towards the top, left corner.

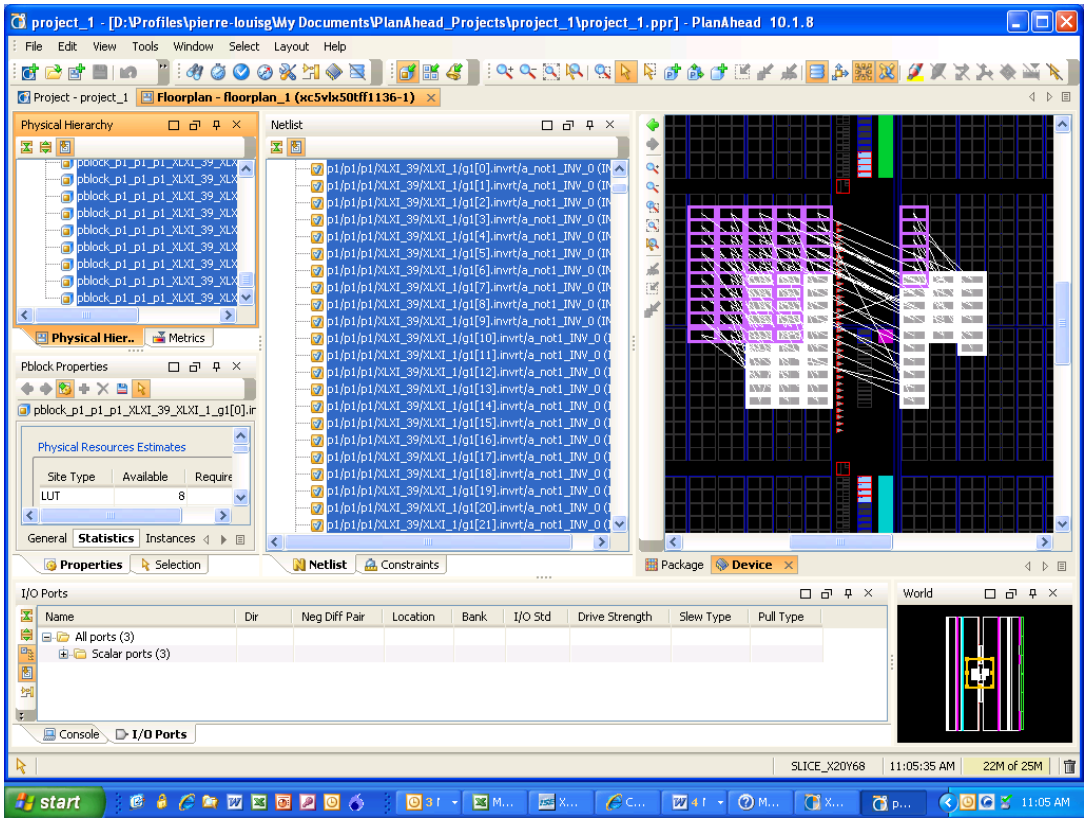

Figure 9 – PlanAhead Screenshot

After the inverters are moved, the application can then create a constraints file with the proper signals moved to their correct locations.

REFERENCES

[1] Poki Chen, Chun-Yan Chu, Mon-Chau Shie, Zi-Fan Zheng, and Zhi-Yuan Zheng, "A Fully Digital Time-Domain Smart Temperature Sensor Realized With 140 FPGA Logic Elements," *IEEE Transactions On Circuits And Systems—I: Regular Papers*, vol. 54, no. 12, pp. 2661-2668, Dec. 2007.

[2] Sudhakar Yalamanchili, *Introductory VHDL: From Simulation to Synthesis*, Upper Saddle River, NJ: Prentice-Hall, Inc., 2001.

[3] Dr. Zainalabedin Navabi, *VHDL: Modular Design and Synthesis of Cores and Systems*, New York, NY: McGraw-Hil, 2007.

[4] "Virtex-5 User Guide," 2009, Xilinx, Feb. 2009, <http://www.xilinx.com/support/documentation/user_guides/ug190.pdf>

[5] "Virtex-5 System Monitor: User Guide," 2009, Xilinx, Feb. 2009, <http://www.xilinx.com/support/documentation/user_guides/ug192.pdf>

[6] "ChipScope Pro Software and Cores User Guide", 2009, Xilinx, Mar. 2009, <http://www.xilinx.com/support/documentation/sw_manuals/chipscope_pro_sw_cores_9_1i_ug029.pdf>

[7] "MATLAB - The Language of Technical Computing," MathWorks, Mar. 2009, <http://www.mathworks.com/products/matlab/>

APPENDICES

Appendix A : MATLAB Code

Get Data

```
clear


%all_my_files = {};
%save('file_list', 'all_my_files');
load('file_list');
%mkdir('data');

while(1)
    a = datestr(clock, 'mm_dd__HH_MM_SS');


    d = instrfind('Port', 'COM1');

    if length(d) ~= 0
        fclose(d);
        delete(d);
    end

    s = serial('COM1', 'InputBufferSize',3, 'BaudRate', 19200, ...
        'Timeout', 500000, 'ReadAsyncMode', 'continuous');

    fopen(s)

    readasync(s)

    out = fread(s,3,'uint8');



    %recieve data
    COUNT1 =  out(1)
%      *2^24+out(3)*2^16+out(2)*2^8+out(1)
    SYSMON_Temp_DegreesCelsius = (out(3)*2^8+out(2))*.49 - 273


    filename = ['data/', a]; %data is folder

    save(filename, 'COUNT1','SYSMON_Temp_DegreesCelsius'); %out is your
recieved data

    all_my_files = [all_my_files; {filename}];
    save('file_list', 'all_my_files');
```

```
    file_count = length(all_my_files);

    pause(1);

end
```

Load Data (Create Spreadsheet)

```
    load ('file_list');
    file_count = length(all_my_files);
    all_data = zeros(file_count, 2);

    for i=1:file_count
       one_file=char(all_my_files(i));
       load(one_file);
       all_data(i,1) = COUNT1;
       all_data(i,2) = SYSMON_Temp_DegreesCelsius;
    end

    all_data
```

Appendix B : FPGA Code (VHDL and Schematics)

Xilinx ISE Project Properties

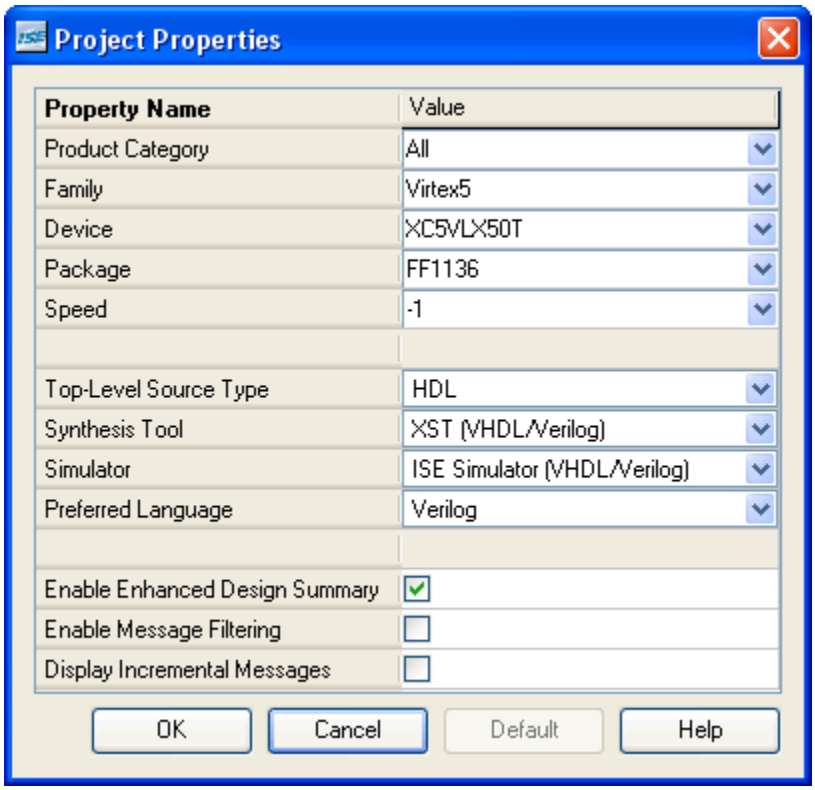

Figure 10 – ISE Project Properties Selection

Inverter

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity inverter is
    Port ( a     : in  STD_LOGIC;
           a_not : out  STD_LOGIC );
end inverter;

architecture Behavioral of inverter is

begin
       a_not <= not(a); --inverter
end Behavioral;
```

Delayline

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
entity delayline is
    Port ( sgnl         : in STD_LOGIC;
           delayed_sgnl : out STD_LOGIC );
end delayline;

architecture Behavioral of delayline is

component inverter
      port(a     : in  std_logic;
        a_not : out std_logic );
end component;

signal avector : STD_LOGIC_VECTOR (0 to 51); -- 51 inverters

attribute keep : integer;                              --used so the synthesizer
attribute keep of avector: signal is 1;  --doesn't throw away inverters

begin
      avector(0) <= sgnl; --signal goes through first inverter

      g1: for i in 0 to 50 generate --up to 50 because of the indexing
            invrt: invertor port map ( avector(i), avector(i+1) ); --move signals
through inverters
      end generate g1;

      delayed_sgnl <= avector(51); --signal comes out last inverter

end Behavioral;
```

Ring Oscillator

Figure 11 – Ring Oscillator

Fixed Ring Oscillator

This component is the synchronized version of the ring oscillator component.

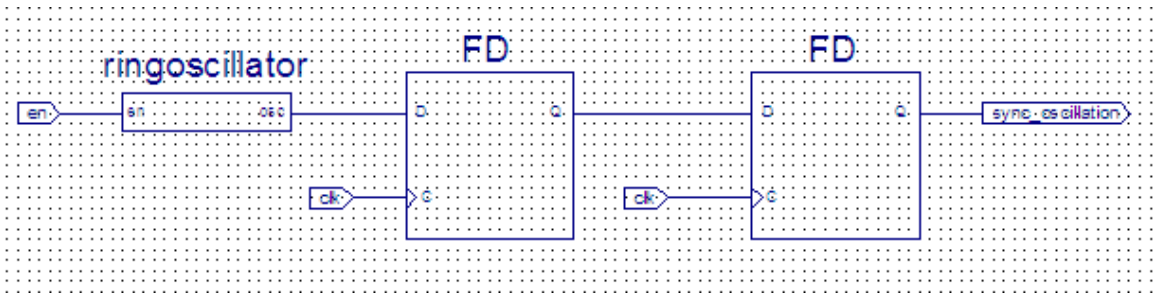

Figure 12 – Fixed Ring Oscillator

Counting State Machine

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity countingstatemachine is
    Port ( osc    : in  STD_LOGIC;
           clk    : in  STD_LOGIC;
           reset  : in  STD_LOGIC;
                    send   : in  STD_LOGIC;
                    sample : out  STD_LOGIC_VECTOR (31 downto 0));
end countingstatemachine;

architecture Behavioral of countingstatemachine is

signal tempcount : STD_LOGIC_VECTOR (31 downto 0) := (others => '0');
signal tempsample : STD_LOGIC_VECTOR (31 downto 0);
signal cyclecount : STD_LOGIC_VECTOR (14 downto 0) := (others => '0');
signal osc_prime : STD_LOGIC;

begin

       process (reset, clk, cyclecount, osc, osc_prime, send) begin

              if (reset = '1') then -- when reset button is pressed, then reset
                     osc_prime <= '0';
                     tempcount <= (others => '0');
                     tempsample <= (others => '0');
                     cyclecount <= (others => '0');
              elsif clk'event and clk = '1' then -- the clk is rising
                     osc_prime <= osc; -- set osc_prime to osc on the next clk cycle
                     if (cyclecount = "100000000000000") then -- if oscillator cycles
2^14 times
                            if send = '0' then --and getting
                                   tempsample <= tempcount; --then send highest count
                            else
                                   tempsample <= tempsample; --if sending then keep
                            end if;
                            tempcount <= (others => '0');  --reset count
                            cyclecount <= (others => '0'); --reset cycle count
                     else --when not equal to 2^14 cycles
                            if osc_prime = '0' and osc = '1' then -- osc prime is
rising
                                   cyclecount <= cyclecount + 1; --increment cycle count
                            else
                                   cyclecount <= cyclecount; --if not rising, keep
                            end if;
```

```
                        tempcount <= tempcount + 1; --increment count
                        tempsample <= tempsample; --keep
                end if;
        else --if clk not rising and not reseting, then keep everything
                tempcount <= tempcount;
                tempsample <= tempsample;
                cyclecount <= cyclecount;
                osc_prime <= osc_prime;
        end if;

    end process;

    sample <= tempsample;

end Behavioral;
```

Clock Convert

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

--  This code generates the clock signal
--  5208 Hz for a baud rate of 19200 from a single
--  100 MHz clock provided by the Virtex 5 board
--  Code from WPI ECE Course 3801 01.26.2009

entity Clk_Convrt is
    Port ( Clk_in              : in std_logic;
           Reset               : in std_logic;
           Clk_outStateMachine : out std_logic;
                    Clk_outrs232        : out std_logic );
end Clk_Convrt;

architecture Behavioral of Clk_Convrt is

   signal tmp_clkstate : std_logic:='0'; --temp signal for Clk_outStateMachine
      signal tmp_clkrs : std_logic:='0'; --temp signal for Clk_outrs232

begin

      Clk_outStateMachine <= tmp_clkstate;
      Clk_outrs232 <= tmp_clkrs;

      process(Reset,Clk_in)

      variable counterstate:integer range 0 TO 50_000_000; -- 1Hz
      variable counterrs:integer range 0 TO 2604; --~19.2kHz

      begin

            if Reset = '1' then --if reset then reset
                  counterstate := 0;
                  counterrs := 0;
            elsif Clk_in'event and Clk_in = '1' then --if clock rising
                  counterstate := counterstate+1; --counts++
                  counterrs := counterrs+1;
                  if counterstate = 50_000_000 then
                        tmp_clkstate <= not tmp_clkstate;
                        counterstate := 0;
                  end if;
```

```
                    if counterrs = 2604 then
                          tmp_clkrs <= not tmp_clkrs;
                          counterrs := 0;
                    end if;
             end if;
      end process;
end Behavioral;
```

RS232 State Machine

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity rs232statemachine is
    Port ( DB     : in  STD_LOGIC_VECTOR (47 downto 0);
                     clk    : in STD_LOGIC;
                     reset  : in STD_LOGIC;
                     send   : in STD_LOGIC;
                     button : in STD_LOGIC;
                     bitout : out STD_LOGIC );
end rs232statemachine;

architecture Behavioral of rs232statemachine is

type statetype is (idle1, start, db0, db1, db2, db3, db4, db5, db6, db7, stop, idle2);
--there will be 12 states
signal state : statetype;
signal index : integer range 0 TO 48 := 0; --array index

begin
      process (clk, reset) begin
            if (reset = '1') then --if reset = 1 then reset
                  state <= idle1;
                  index <= 0;
            elsif (clk'event and clk = '1') then -- if clk is rising
                  case (state) is
                        when idle1 =>  -- if not sending then stay in this state
                              if (button = '1' and send = '1') then
                                    state <= start;
                              else
                                    state <= idle1;
                              end if;
                        when start =>
                              state <= db0;
                        when db0 =>
                              state <= db1;
                        when db1 =>
                              state <= db2;
                        when db2 =>
                              state <= db3;
                        when db3 =>
                              state <= db4;
                        when db4 =>
                              state <= db5;
                        when db5 =>
                              state <= db6;
                        when db6 =>
                              state <= db7;
                        when db7 =>
                              state <= stop;
```

```
                        when stop =>
                                if not(index = 40) then --send more bits if not
complete
                                        state <= start;
                                        index <= index + 8;
                                else
                                        index <= 0;
                                        state <= idle2;
                                end if;
                        when idle2 => --stay in idle if done sending but not
getting
                                if (send = '1') then
                                        state <= idle2;
                                else
                                        state <= idle1;
                                end if;
                        when others =>
                                null;
                end case;
        end if;
    end process;

    with state select
        bitout <= '1' when idle1,
                          '0' when start,
                          DB(index) when db0,
                          DB(index + 1) when db1,
                          DB(index + 2) when db2,
                          DB(index + 3) when db3,
                          DB(index + 4) when db4,
                          DB(index + 5) when db5,
                          DB(index + 6) when db6,
                          DB(index + 7) when db7,
                          '1' when stop,
                          '1' when idle2,
                          '1' when others;
end Behavioral;
```

System Monitor Setup

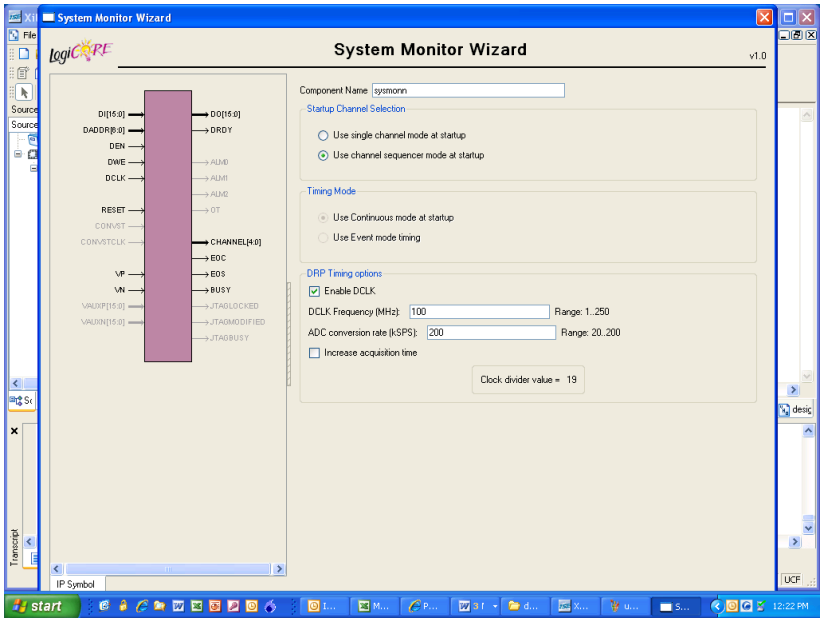

Figure 13 – System Monitor Wizard Setup Pg 1

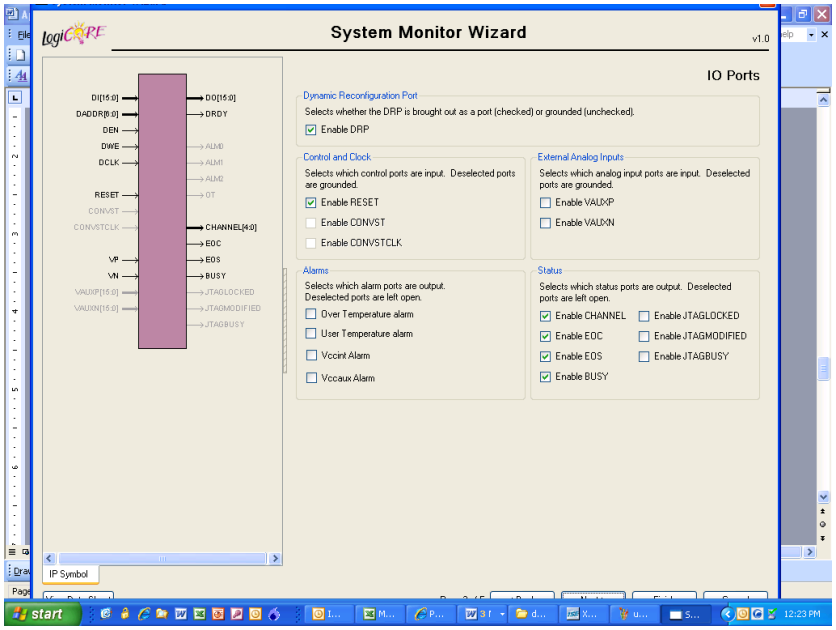

Figure 14 – System Monitor Wizard Setup Pg 2

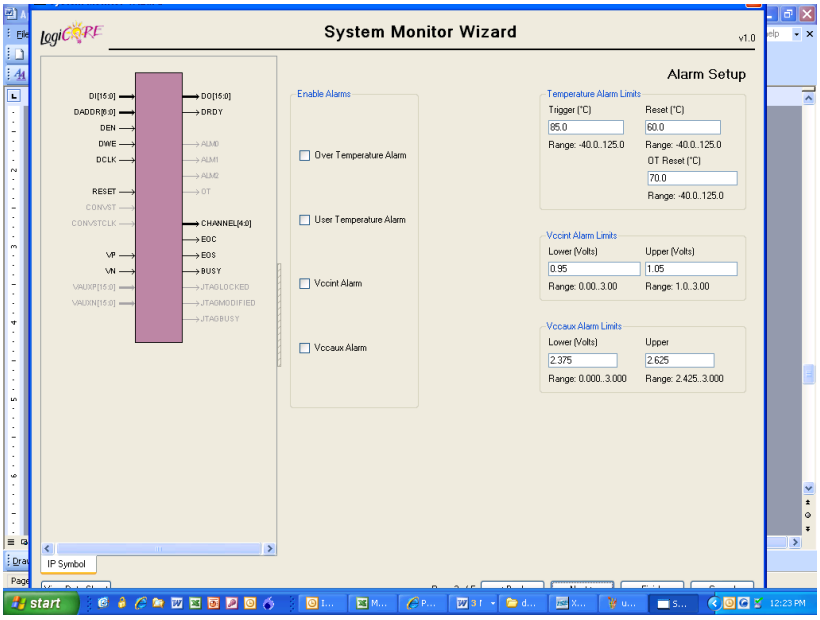

Figure 15 – System Monitor Wizard Setup Pg 3

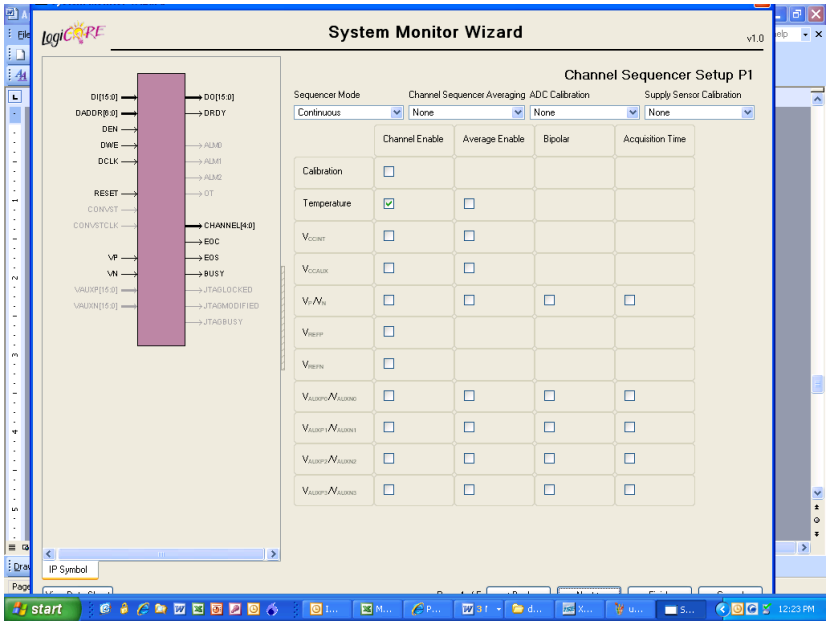

Figure 16 – System Monitor Wizard Setup Pg 4

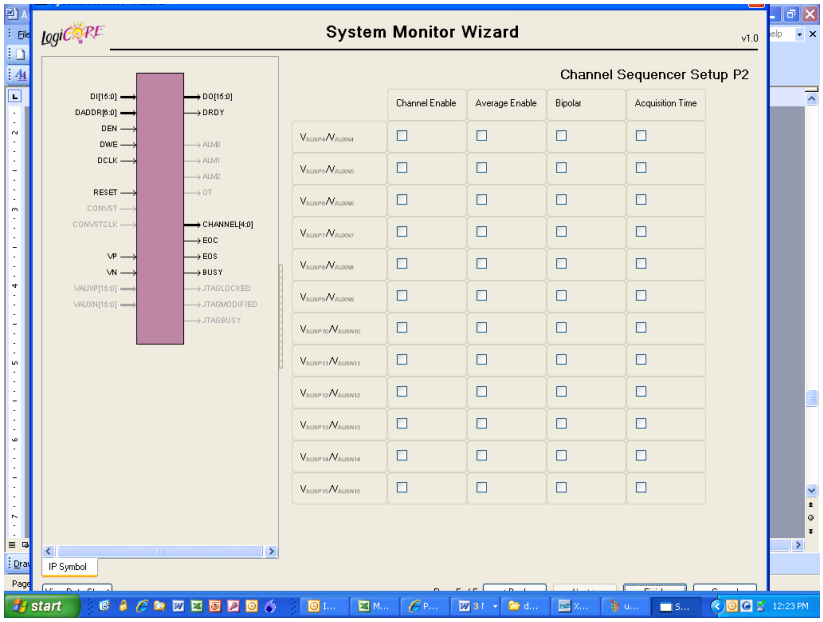

Figure 17 – System Monitor Wizard Setup Pg 5

RS232

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

library UNISIM;
use UNISIM.VComponents.all;

entity rs232 is
    Port ( sample : in  STD_LOGIC_VECTOR (31 downto 0);
                       clk    : in STD_LOGIC;
                       reset  : in STD_LOGIC;
                       send   : in STD_LOGIC;
                       button : in STD_LOGIC;
                       bitout : out STD_LOGIC );
end rs232;

architecture Behavioral of rs232 is

component Clk_Convrt is
    Port ( Clk_in              : in std_logic;
           Reset               : in std_logic;
           Clk_outStateMachine : out std_logic;
                       Clk_outrs232        : out std_logic );
end component;

component rs232statemachine is
    Port ( DB     : in  STD_LOGIC_VECTOR (47 downto 0);
                       clk    : in STD_LOGIC;
                       reset  : in STD_LOGIC;
                       send   : in STD_LOGIC;
                       button : in STD_LOGIC;
                       bitout : out STD_LOGIC );
end component;
```

```
component sysmonn is
    Port ( DCLK_IN     : in  STD_LOGIC;
                       DWE_IN      : in STD_LOGIC;
                       DEN_IN      : in STD_LOGIC;
                       DADDR_IN    : in STD_LOGIC_VECTOR (6 downto 0);
                       DI_IN       : in STD_LOGIC_VECTOR (15 downto 0);
                       RESET_IN    : in STD_LOGIC;
                       DO_OUT      : out STD_LOGIC_VECTOR (15 downto 0);
                       CHANNEL_OUT : out STD_LOGIC_VECTOR (4 downto 0);
                       BUSY_OUT    : out STD_LOGIC;
           EOS_OUT     : out STD_LOGIC;

                       DRDY_OUT: out STD_LOGIC; --unused
           EOC_OUT: out STD_LOGIC --unused
                       );
end component;


signal tempclk : STD_LOGIC;
signal dummy : STD_LOGIC; --dummy signal
signal tempDO : STD_LOGIC_VECTOR (15 downto 0);
signal temperatures : STD_LOGIC_VECTOR (47 downto 0);
signal channel : STD_LOGIC_VECTOR (4 downto 0);
signal daddr : STD_LOGIC_VECTOR (6 downto 0);
signal en : STD_LOGIC;

begin

       p1: Clk_Convrt port map ( clk, '0', dummy, tempclk );
       temperatures (47 downto 0) <= "000000" & tempDO(15 downto 6) & sample;
       daddr <= "00" & channel;
       p2: rs232statemachine port map ( temperatures, tempclk, reset, send, button,
bitout );
       mysys: sysmonn port map (clk, '0', en, daddr, "0000000000000000", '0', tempDO,
channel, en);
end Behavioral;
```

Design Implementing the RS232

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

---- Uncomment the following library declaration if instantiating
---- any Xilinx primitives in this code.
--library UNISIM;
--use UNISIM.VComponents.all;

entity designWithRS232 is
    Port ( clk    : in  STD_LOGIC;
           reset  : in  STD_LOGIC;
                      send   : in STD_LOGIC;
                      button : in STD_LOGIC;
                      bitout : out  STD_LOGIC);
end designWithRS232;

architecture Behavioral of designWithRS232 is

component counter is
    Port ( clk    : in  STD_LOGIC;
           reset  : in  STD_LOGIC;
```

```
                        send   : in  STD_LOGIC;
                        sample : out  STD_LOGIC_VECTOR (31 downto 0));
end component;

component rs232 is
        Port ( sample : in  STD_LOGIC_VECTOR (31 downto 0);
                        clk    : in STD_LOGIC;
                        reset  : in STD_LOGIC;
                        send   : in STD_LOGIC;
                        button : in STD_LOGIC;
                        bitout : out STD_LOGIC);
end component;

signal tempsample : STD_LOGIC_VECTOR (31 downto 0);

begin

       p1: counter port map ( clk, reset, send, tempsample );
       p2: rs232 port map ( tempsample, clk, reset, send, button, bitout );

end Behavioral;
```

Send/Get State Machine

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity sendgetstatemachine is --sends either 0 (get) or 1 (send) everyother clock
cycle
    Port ( clk  : in  STD_LOGIC;
           send : out  STD_LOGIC );
end sendgetstatemachine;

architecture Behavioral of sendgetstatemachine is

signal tempsend : STD_LOGIC := '0'; --temp send signal

begin

       process (clk) begin

       if (clk'event and clk = '1') then --if clk rising, send = not send
              tempsend <= not(tempsend);
       else
              tempsend <= tempsend; --else keep
       end if;

       send <= tempsend;

       end process;
end Behavioral;
```

Top Level Design Module

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
```

```
entity design is
    Port ( clk    : in  STD_LOGIC;
           reset  : in  STD_LOGIC;
           bitout : out  STD_LOGIC );
end design;

architecture Behavioral of design is

component designWithRS232 is
    Port ( clk    : in  STD_LOGIC;
           reset  : in  STD_LOGIC;
                      send   : in STD_LOGIC;
                      button : in STD_LOGIC;
                      bitout : out  STD_LOGIC );
end component;

component sendgetstatemachine is
    Port ( clk  : in  STD_LOGIC;
           send : out  STD_LOGIC );
end component;

component Clk_Convrt is
    Port ( Clk_in              : in std_logic;
           Reset               : in std_logic;
           Clk_outStateMachine : out std_logic;
                      Clk_outrs232        : out std_logic);
end component;

signal tempsend : STD_LOGIC;
signal tempclk : STD_LOGIC;
signal dummy : STD_LOGIC; --dummy signal

begin

       p1: designWithRS232 port map ( clk, reset, tempsend, '1', bitout );
       p2: sendgetstatemachine port map ( tempclk, tempsend );
       p3: Clk_Convrt port map ( clk, '0', tempclk, dummy );

end Behavioral;
```

Appendix C: CTC Table

Count	Calibrated Count	Calc Temp
69475	55	19.2
69476	56	19.5
69477	57	19.7
69478	58	19.9
69479	59	20.1
69480	60	20.3
69481	61	20.6
69482	62	20.8
69483	63	21.0
69484	64	21.2
69485	65	21.4
69486	66	21.6
69487	67	21.9
69488	68	22.1
69489	69	22.3
69490	70	22.5
69491	71	22.7
69492	72	22.9
69493	73	23.2
69494	74	23.4
69495	75	23.6
69496	76	23.8
69497	77	24.0
69498	78	24.2
69499	79	24.4
69500	80	24.6
69501	81	24.8
69502	82	25.1
69503	83	25.3
69504	84	25.5
69505	85	25.7
69506	86	25.9
69507	87	26.1
69508	88	26.3
69509	89	26.5
69510	90	26.7
69511	91	26.9
69512	92	27.1
69513	93	27.3
69514	94	27.5
69515	95	27.7
69516	96	27.9
69517	97	28.1

69518	98	28.3
69519	99	28.5
69520	100	28.7
69521	101	28.9
69522	102	29.1
69523	103	29.3
69524	104	29.5
69525	105	29.7
69526	106	29.9
69527	107	30.1
69528	108	30.3
69529	109	30.5
69530	110	30.7
69531	111	30.9
69532	112	31.1
69533	113	31.3
69534	114	31.5
69535	115	31.7
69536	116	31.9
69537	117	32.1
69538	118	32.3
69539	119	32.5
69540	120	32.7
69541	121	32.9
69542	122	33.0
69543	123	33.2
69544	124	33.4
69545	125	33.6
69546	126	33.8
69547	127	34.0
69548	128	34.2
69549	129	34.4
69550	130	34.6
69551	131	34.7
69552	132	34.9
69553	133	35.1
69554	134	35.3
69555	135	35.5
69556	136	35.7
69557	137	35.9
69558	138	36.0
69559	139	36.2
69560	140	36.4
69561	141	36.6
69562	142	36.8
69563	143	36.9
69564	144	37.1

69565	145	37.3
69566	146	37.5
69567	147	37.7
69568	148	37.8
69569	149	38.0
69570	150	38.2
69571	151	38.4
69572	152	38.6
69573	153	38.7
69574	154	38.9
69575	155	39.1
69576	156	39.3
69577	157	39.4
69578	158	39.6
69579	159	39.8
69580	160	39.9
69581	161	40.1
69582	162	40.3
69583	163	40.5
69584	164	40.6
69585	165	40.8
69586	166	41.0
69587	167	41.1
69588	168	41.3
69589	169	41.5
69590	170	41.7
69591	171	41.8
69592	172	42.0
69593	173	42.2
69594	174	42.3
69595	175	42.5
69596	176	42.7
69597	177	42.8
69598	178	43.0
69599	179	43.1
69600	180	43.3
69601	181	43.5
69602	182	43.6
69603	183	43.8
69604	184	44.0
69605	185	44.1
69606	186	44.3
69607	187	44.4
69608	188	44.6
69609	189	44.8
69610	190	44.9
69611	191	45.1

69612	192	45.2
69613	193	45.4
69614	194	45.6
69615	195	45.7
69616	196	45.9
69617	197	46.0
69618	198	46.2
69619	199	46.3
69620	200	46.5
69621	201	46.6
69622	202	46.8
69623	203	46.9
69624	204	47.1
69625	205	47.2
69626	206	47.4
69627	207	47.5
69628	208	47.7
69629	209	47.9
69630	210	48.0
69631	211	48.1
69632	212	48.3
69633	213	48.4
69634	214	48.6
69635	215	48.7
69636	216	48.9
69637	217	49.0
69638	218	49.2
69639	219	49.3
69640	220	49.5
69641	221	49.6
69642	222	49.8
69643	223	49.9
69644	224	50.0
69645	225	50.2
69646	226	50.3
69647	227	50.5
69648	228	50.6
69649	229	50.8
69650	230	50.9
69651	231	51.0
69652	232	51.2
69653	233	51.3
69654	234	51.4
69655	235	51.6
69656	236	51.7
69657	237	51.9
69658	238	52.0

69659	239	52.1
69660	240	52.3
69661	241	52.4
69662	242	52.5
69663	243	52.7
69664	244	52.8
69665	245	52.9
69666	246	53.1
69667	247	53.2
69668	248	53.3
69669	249	53.5
69670	250	53.6
69671	251	53.7
69672	252	53.9
69673	253	54.0
69674	254	54.1
69675	255	54.2
69676	256	54.4
69677	257	54.5
69678	258	54.6
69679	259	54.8
69680	260	54.9
69681	261	55.0
69682	262	55.1
69683	263	55.3
69684	264	55.4
69685	265	55.5
69686	266	55.6
69687	267	55.8
69688	268	55.9
69689	269	56.0
69690	270	56.1
69691	271	56.2
69692	272	56.4
69693	273	56.5
69694	274	56.6
69695	275	56.7
69696	276	56.8
69697	277	57.0
69698	278	57.1
69699	279	57.2
69700	280	57.3
69701	281	57.4
69702	282	57.5
69703	283	57.7
69704	284	57.8
69705	285	57.9

69706	286	58.0
69707	287	58.1
69708	288	58.2
69709	289	58.3
69710	290	58.4
69711	291	58.6
69712	292	58.7
69713	293	58.8
69714	294	58.9
69715	295	59.0
69716	296	59.1
69717	297	59.2
69718	298	59.3
69719	299	59.4
69720	300	59.5
69721	301	59.7
69722	302	59.8
69723	303	59.9
69724	304	60.0
69725	305	60.1
69726	306	60.2
69727	307	60.3
69728	308	60.4
69729	309	60.5
69730	310	60.6
69731	311	60.7
69732	312	60.8
69733	313	60.9
69734	314	61.0
69735	315	61.1
69736	316	61.2
69737	317	61.3
69738	318	61.4
69739	319	61.5
69740	320	61.6
69741	321	61.7
69742	322	61.8
69743	323	61.9
69744	324	62.0
69745	325	62.1
69746	326	62.2
69747	327	62.3
69748	328	62.4
69749	329	62.5
69750	330	62.5
69751	331	62.6
69752	332	62.7

69753	333	62.8
69754	334	62.9
69755	335	63.0
69756	336	63.1
69757	337	63.2
69758	338	63.3
69759	339	63.4
69760	340	63.5
69761	341	63.5
69762	342	63.6
69763	343	63.7
69764	344	63.8
69765	345	63.9
69766	346	64.0
69767	347	64.1
69768	348	64.1
69769	349	64.2
69770	350	64.3
69771	351	64.4
69772	352	64.5
69773	353	64.6
69774	354	64.6
69775	355	64.7
69776	356	64.8
69777	357	64.9
69778	358	65.0
69779	359	65.1
69780	360	65.1
69781	361	65.2
69782	362	65.3
69783	363	65.4
69784	364	65.4
69785	365	65.5
69786	366	65.6
69787	367	65.7
69788	368	65.8
69789	369	65.8
69790	370	65.9
69791	371	66.0
69792	372	66.0
69793	373	66.1
69794	374	66.2
69795	375	66.3
69796	376	66.3
69797	377	66.4
69798	378	66.5
69799	379	66.6

69800	380	66.6
69801	381	66.7
69802	382	66.8
69803	383	66.8
69804	384	66.9
69805	385	67.0
69806	386	67.0
69807	387	67.1
69808	388	67.2
69809	389	67.2
69810	390	67.3
69811	391	67.4
69812	392	67.4
69813	393	67.5
69814	394	67.6
69815	395	67.6
69816	396	67.7
69817	397	67.7
69818	398	67.8
69819	399	67.9
69820	400	67.9
69821	401	68.0
69822	402	68.0
69823	403	68.1
69824	404	68.2
69825	405	68.2
69826	406	68.3
69827	407	68.3
69828	408	68.4
69829	409	68.4
69830	410	68.5
69831	411	68.6
69832	412	68.6
69833	413	68.7
69834	414	68.7
69835	415	68.8
69836	416	68.8
69837	417	68.9
69838	418	68.9
69839	419	69.0
69840	420	69.0
69841	421	69.1
69842	422	69.1
69843	423	69.2
69844	424	69.2
69845	425	69.3
69846	426	69.3

69847	427	69.4
69848	428	69.4
69849	429	69.5
69850	430	69.5
69851	431	69.6
69852	432	69.6
69853	433	69.7
69854	434	69.7
69855	435	69.8
69856	436	69.8
69857	437	69.8
69858	438	69.9
69859	439	69.9
69860	440	70.0
69861	441	70.0
69862	442	70.0
69863	443	70.1
69864	444	70.1
69865	445	70.2
69866	446	70.2
69867	447	70.2
69868	448	70.3
69869	449	70.3