EGR 4830 – Senior Project

The Rule of Three: Triple Modular Redundancy

Student: Landon Page
Advisor: Menglai Yin
Computer Engineering, B.S.
Dec. 3, 2021

The Rule of Three: Triple Modular Redundancy (TMR)

Abstract:

Triple Modular Redundancy is an effective method of redundant computing to give computer systems fault tolerance. Triple Modular Redundancy is commonly used in space missions and other applications where risk of computer or component failure is considerably high in order to mitigate risk of complete system failure. A fault tolerant system usually involves 3 single-board computers and an FPGA board. For my senior project, I built a TMR system using 3 Raspberry Pi Zeros, 3 Temperature sensors and a Digilent Nexys-4 A7 FPGA board. The system is asynchronous, however after multiple rounds of testing the system, I've found that lack of synchronization in the system has had little impact on the performance. TMR systems are designed to be able to tolerate up to one computer fault in the system, and after multiple rounds of testing with my system I'm confident in saying that it meets that specification.

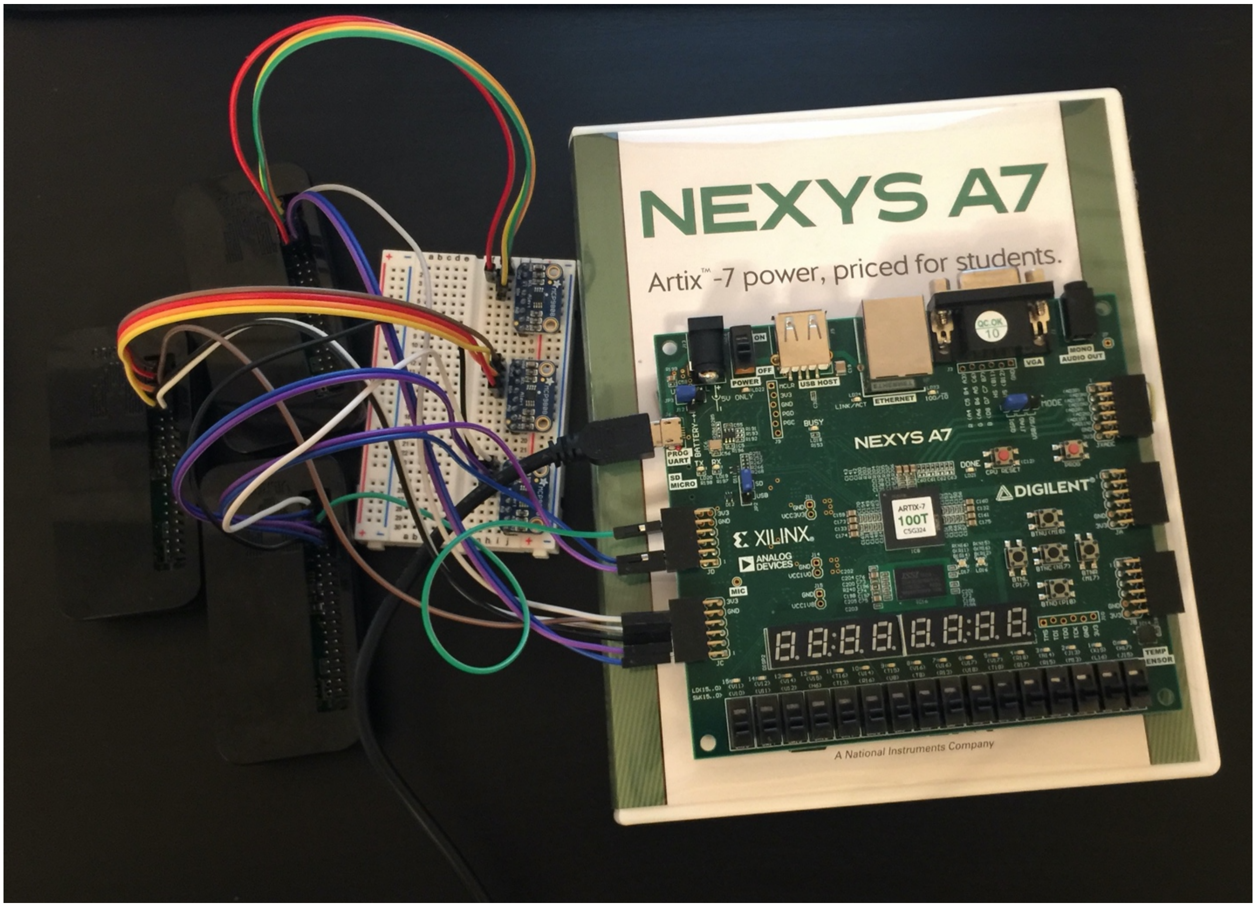
Table of Contents:

1. Introduction & Objective

Modular Redundancy is the idea of using multiple computers in parallel for a specific task such that if one of the computers malfunctions or ceases operation, the other computers can continue system operation. There are many different applications of this used in several different industries from the automotive industry to space exploration. One particular application I had in mind was TMR systems used for satellites and deep space exploration missions. During the summer of 2021, I performed a Software Engineering Internship at the NASA Jet Propulsion Laboratory where I work on the Europa Lander Project. Europa Lander is a mission concept still in development by NASA where a TMR system will be used. This internship gave me a lot of insight into what Triple Modular Redundancy is and how it works. For my senior project, I decided to build a system similar to what NASA plans to use on said mission.

Say NASA or some other organization engaged in space exploration has a deep space mission where they choose to employ a TMR system over a conventional single-computer setup. Obviously, there are some drawbacks to this approach. For one, they are obviously more expensive, since one needs not only three computers instead of one, but also three sets of all the scientific equipment attached to each computer to carry out the data collection required for the mission. One would also need a voting module or actuator, which is a processing unit that takes in the outputs from each of the computers and compares them to determine how valid the data is. All of these extra pieces of equipment add up in price. The main benefit here is that on a dangerous mission like this, a TMR can continue to work and provide data to mission control even if some components fail. So, say in this deep space mission, one of the computers fails in some way. It or its scientific equipment has been damaged by cosmic radiation and now either produces incorrect output or no output at all. The data from the computers is fed into the voting module and the voting module determines what the correct output should be based on the inputs. Hence, one would still have data integrity in this scenario.

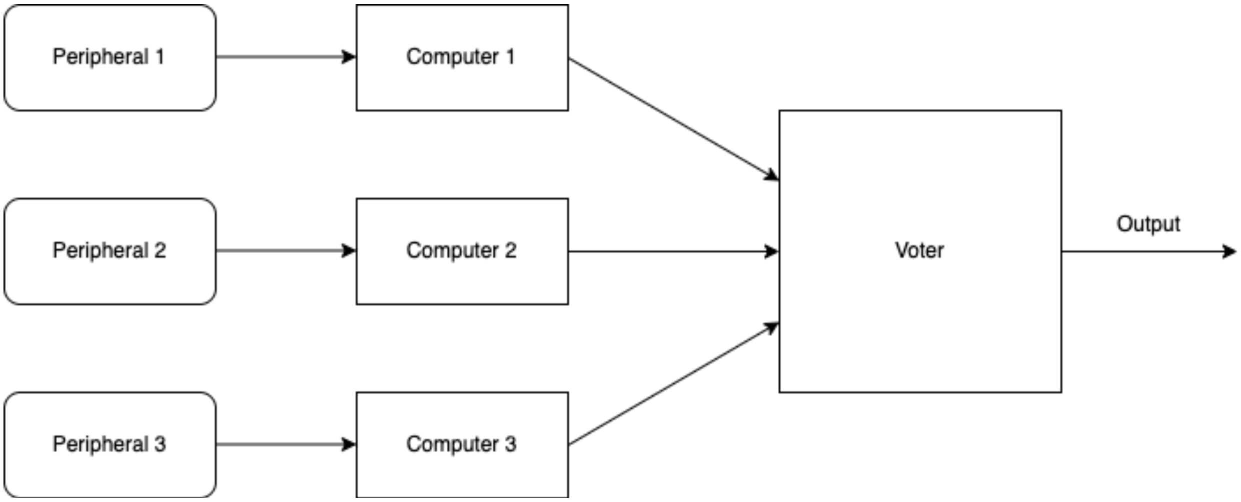
One might also ask, “Well, you’ve reduced chance of mission failure by using a TMR but isn’t the voting module or actuator a single point of failure which could cause the whole system to fail if it fails?” and this would be a valid concern. However, in most deep space missions that use a system like this, the actuator used is usually what is called a ‘radiation-hardened’ or ‘radiation-tolerant’ FPGA board or microcontroller. By using something like this, the risk of failure is reduced to a minimal level, since the actuator is treated to handle the expected levels of radiation for the mission. Hence, for my senior project I decided to use a Digilent Artix-7 FPGA board for my actuator, and three Raspberry Pi Zero computers with temperature sensors as my computers and scientific equipment. A picture of my hardware setup can be seen below:


NEXYS A7
Artix™-7 power, priced for students.
NEXYS A7
XILINX
ANALOG DEVICES
DIGILENT
A National Instruments Company

The three Raspberry Pis are on the left, the MCP9808 Temperature Sensors are in the middle, and the FPGA board is on the right. This project is designed to mimic space satellite missions, and how a TMR would be effectively used in them.

2. Design Standards of a Triple Modular Redundancy System

In this section, we will detail the typical characteristics and features of a TMR system. Every system is unique in many ways, as a TMR system is not something readily available to buy at a local RadioShack or Best Buy. With that being said, the most basic TMR systems need three computers or processing units, and then a voter or actuator module to process the output from said computers and determine what the correct output should be. Then TMR systems usually have peripherals for scientific research to record things like atmospheric data or something else. Each computer in a TMR system will need its own set of peripherals, otherwise the system will have the same chance of failure as a single computer system. And of course, each computer/set of peripherals will need their own power supply as well. See the standard layout of a TMR system below:



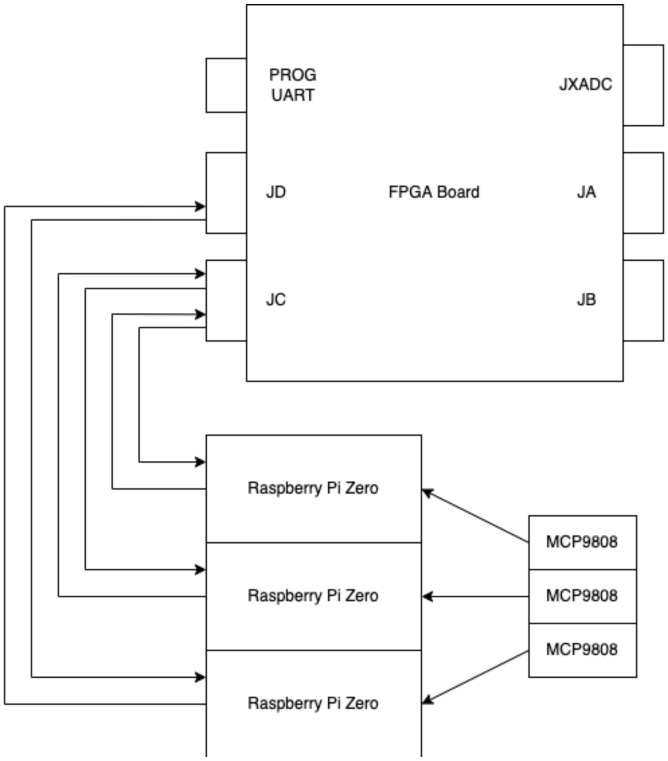
In our case, the temperature sensors are the peripherals, the Raspberry Pis are the computers, and the FPGA board is the voter.

As for the standard features of a TMR system, there are several, and those that are present in a system largely come down to the custom software on the computers and the actuator. The most basic feature is voting, which can be used to mask faults if a single computer in the TMR is malfunctioning. For example, if a TMR system has barometer peripherals and the real pressure in the room is 1 atmosphere, and one of the computers in the TMR outputs 0 atmospheres, then the voter in the system should detect this and output 1 atmosphere. This is called Majority Vote, and in modularly redundant computer systems, it is the minimum number of computers required to output correct data for the TMR to work successfully. Hence, a TMR with voting can tolerate up to one computer/peripheral failure and still work successfully, because there are three computers present (2 out of 3 majority). There are other features such as computer synchronization (ensuring that all computers send output at the same time, and the actuator receives and compares data sent at the same time), fault correction (not only detecting when a computer sends incorrect output but including countermeasures to correct said computer when this happens), and many more.

There are many different configurations and features that a TMR system can have, but ultimately, so long as the TMR system has the capability of functioning normally when one computer or peripheral fails in some way, it is working as intended.

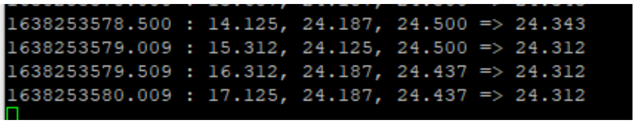
3. Asynchronous TMR Voter System

Keeping in mind all of the typical TMR system characteristics and features we discussed in the previous section, I will now detail the setup and features incorporated into our TMR system. Firstly, for the hardware, I used three Raspberry Pi Zero computers with Adafruit MCP9808 Temperature Sensor peripherals, and a Digilent Artix-7 FPGA board as the actuator. I chose the Raspberry Pi Zero because it was one of the cheapest and most widely used single-board computers that I could find, which meant that there would be a lot of documentation I could look through in case I encountered problems. I used the MCP9808 because it was also cheap and fairly commonly used, which made it easy to integrate into the system. For the FPGA board, I used the Artix-7 because I had already used it in ECE 3300 and ECE 4305 and was already familiar with Digilent products and Xilinx software. The hardware was fairly easy to set up, and said setup can be seen in the schematic below:



As shown in the schematic, each temperature sensor is hooked up to an individual Raspberry Pi computer and then each of those are hooked up to the FPGA board actuator. Each of the Raspberry Pi computers also have tx and rx wires to and from the FPGA board for UART communication. It's not shown in the schematic, but each Raspberry Pi is hooked up to its own power source (standard micro-USB wall plugs), and the FPGA board is powered and programmed by its USB cable hooked up to the host PC.

Next, I'll detail the major features and limitations of the system. Each of the computers has a temperature sensor peripheral, and they measure the temperature of their surroundings roughly every 500 milliseconds or 0.5 seconds. They then send that information to the FPGA board actuator, in a UART communication configuration where each byte represents an ASCII value which is converted to text by the actuator and displayed over UART on the host PC. The actuator takes the converted data and compares the received temperatures from the three computers to see if one temperature can be considered a numerical outlier. For example, if all three temperatures lie within 0.5 degrees C of each other, then the actuator simply takes the average of the three and outputs that. If one outlier exists, but the other two are close to each other, then the actuator takes the average of those two and ignores the outlier. If all three temperatures are far apart, then the system takes the average of all three and outputs that, since this is a basic TMR voter system and there would be no way of knowing which computer is correct. The system also outputs the individual temperatures to give the user a sense of how each computer and peripheral is behaving. For example, the output from the actuator looks like this: “current epoch time (in seconds): temperature 1, temperature 2, temperature 3 => actuator temperature.” See a picture of what the actual output looks like below:



```
1638253578.500 : 14.125, 24.187, 24.500 => 24.343
1638253579.009 : 15.312, 24.125, 24.500 => 24.312
1638253579.509 : 16.312, 24.187, 24.437 => 24.312
1638253580.009 : 17.125, 24.187, 24.437 => 24.312
```

The time on the left is the current epoch time, which is the number of seconds that have elapsed since January 1, 1970. The three temperatures to the right of that represent the output from each of the individual computers, and then the rightmost temperature is the result of the TMR or majority vote. In this example, the 2nd and the 3rd temperatures are far closer to each other than the 1st, which implies the 1st is an outlier, hence it is not considered. This way, if more than one computer or peripheral fails, the user can see and analyze what happened for each computer at a specific time and make a more educated judgement on what the correct temperature should be.

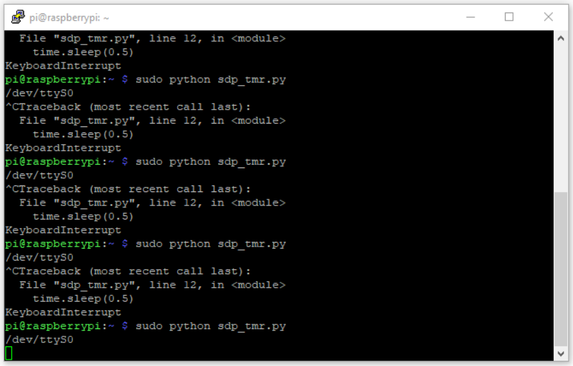
4. Microblaze MCS FPGA Microcontroller System Integration

During the early stages of development, I tried using a pure Verilog approach for the FPGA actuator. This meant that I could only use hardware description language to try to program the FPGA board to do what I needed it to do as the actuator. I had been using code references from the excellent book “FPGA Prototyping by Verilog Examples” by Pong Chu up to this point but resynthesizing code and reprogramming the board each time I made changes to the code made this extremely time-consuming and difficult. Around this time, in my ECE 4305 class, we were using another textbook by the same author (FPGA Prototyping by SystemVerilog Examples, see it in the Work Cited section). I had found that the FPGA board could be used like a microcontroller, and that my textbook had instructions on how to do this. This is called the Xilinx MicroBlaze MCS SoC, and it is an IP available from Xilinx to use the FPGA as a microcontroller. In the class, we did a few labs that utilized this setup, and said setup allowed for the FPGA board to be used as a standard microcontroller, with UART capabilities available in SystemVerilog code. This greatly accelerated development time since I only had to compile the SystemVerilog code for the hardware once, then after that it was purely building software to run on the microcontroller. I also had quite a bit of experience with working with microcontrollers in the past, particularly in my experience as an engineer on UMBRA’s 2020-2021 Singularity Rocket Team.

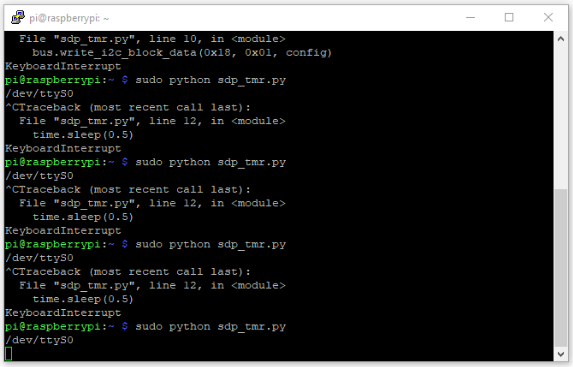
After I setup the microcontroller system on the FPGA board, I then implemented the TMR system software. The final build is a single-threaded program that receives and compares the outputs of each of the Raspberry Pi computers. Due to issues with simultaneous serial read and write for the computers, and the complications of trying to make a multi-threaded program on a microcontroller with no access to standard programming libraries, I made the FPGA actuator asynchronous. Aside from that though, the microcontroller setup on the FPGA made it quite easy and straightforward to implement the software for the TMR. The program simply constantly checks for incoming UART data from each of the computers, compares the difference in measured temperature from each of them, and then outputs a result based on the criterion discussed in the previous section.

5. Testing Results and Discussion

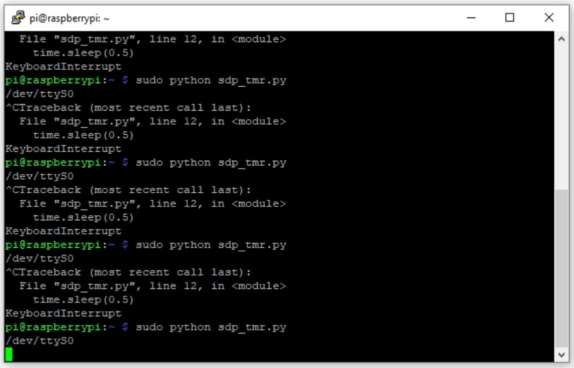
I finished the final build of the project in early November, after resolving all the previously discussed issues. An example of what the environment looks like can be seen below:



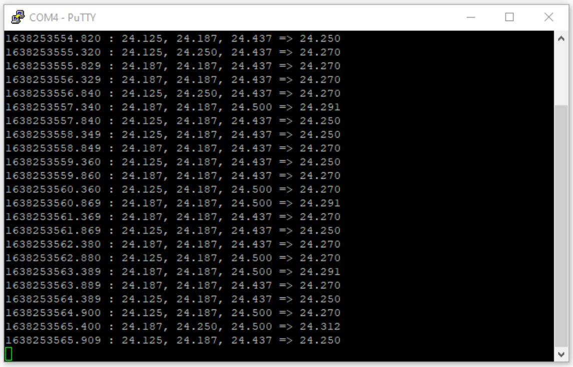
```
  File "sdp_tmr.py", line 12, in <module>
    time.sleep(0.5)
KeyboardInterrupt
pi@raspberrypi:~ $ sudo python sdp_tmr.py
/dev/ttyS0
^CTraceback (most recent call last):
  File "sdp_tmr.py", line 12, in <module>
    time.sleep(0.5)
KeyboardInterrupt
pi@raspberrypi:~ $ sudo python sdp_tmr.py
/dev/ttyS0
^CTraceback (most recent call last):
  File "sdp_tmr.py", line 12, in <module>
    time.sleep(0.5)
KeyboardInterrupt
pi@raspberrypi:~ $ sudo python sdp_tmr.py
/dev/ttyS0
^CTraceback (most recent call last):
  File "sdp_tmr.py", line 12, in <module>
    time.sleep(0.5)
KeyboardInterrupt
pi@raspberrypi:~ $ sudo python sdp_tmr.py
/dev/ttyS0
```



```
  File "sdp_tmr.py", line 10, in <module>
    bus.write_i2c_block_data(0x18, 0x01, config)
KeyboardInterrupt
pi@raspberrypi:~ $ sudo python sdp_tmr.py
/dev/ttyS0
^CTraceback (most recent call last):
  File "sdp_tmr.py", line 12, in <module>
    time.sleep(0.5)
KeyboardInterrupt
pi@raspberrypi:~ $ sudo python sdp_tmr.py
/dev/ttyS0
^CTraceback (most recent call last):
  File "sdp_tmr.py", line 12, in <module>
    time.sleep(0.5)
KeyboardInterrupt
pi@raspberrypi:~ $ sudo python sdp_tmr.py
/dev/ttyS0
^CTraceback (most recent call last):
  File "sdp_tmr.py", line 12, in <module>
    time.sleep(0.5)
KeyboardInterrupt
pi@raspberrypi:~ $ sudo python sdp_tmr.py
/dev/ttyS0
```



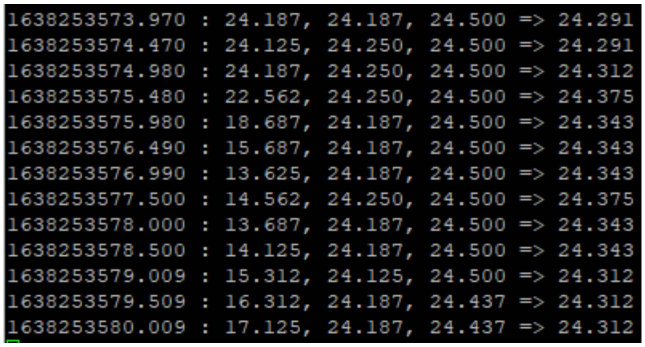
```
  File "sdp_tmr.py", line 12, in <module>
    time.sleep(0.5)
KeyboardInterrupt
pi@raspberrypi:~ $ sudo python sdp_tmr.py
/dev/ttyS0
^CTraceback (most recent call last):
  File "sdp_tmr.py", line 12, in <module>
    time.sleep(0.5)
KeyboardInterrupt
pi@raspberrypi:~ $ sudo python sdp_tmr.py
/dev/ttyS0
^CTraceback (most recent call last):
  File "sdp_tmr.py", line 12, in <module>
    time.sleep(0.5)
KeyboardInterrupt
pi@raspberrypi:~ $ sudo python sdp_tmr.py
/dev/ttyS0
^CTraceback (most recent call last):
  File "sdp_tmr.py", line 12, in <module>
    time.sleep(0.5)
KeyboardInterrupt
pi@raspberrypi:~ $ sudo python sdp_tmr.py
/dev/ttyS0
```



```
1638253554.820 : 24.125, 24.187, 24.437 => 24.250
1638253555.320 : 24.125, 24.250, 24.437 => 24.270
1638253555.829 : 24.187, 24.187, 24.437 => 24.270
1638253556.329 : 24.187, 24.187, 24.437 => 24.270
1638253556.840 : 24.125, 24.250, 24.437 => 24.270
1638253557.340 : 24.187, 24.187, 24.500 => 24.291
1638253557.840 : 24.125, 24.187, 24.437 => 24.250
1638253558.349 : 24.125, 24.187, 24.437 => 24.250
1638253558.849 : 24.187, 24.187, 24.437 => 24.270
1638253559.360 : 24.125, 24.187, 24.437 => 24.250
1638253559.860 : 24.187, 24.187, 24.437 => 24.270
1638253560.360 : 24.125, 24.187, 24.500 => 24.270
1638253560.869 : 24.187, 24.187, 24.500 => 24.291
1638253561.369 : 24.187, 24.187, 24.437 => 24.270
1638253561.869 : 24.125, 24.187, 24.437 => 24.250
1638253562.380 : 24.187, 24.187, 24.437 => 24.270
1638253562.880 : 24.125, 24.187, 24.500 => 24.270
1638253563.389 : 24.187, 24.187, 24.500 => 24.291
1638253563.889 : 24.187, 24.187, 24.437 => 24.270
1638253564.389 : 24.125, 24.187, 24.437 => 24.250
1638253564.900 : 24.125, 24.187, 24.500 => 24.270
1638253565.400 : 24.187, 24.250, 24.500 => 24.312
1638253565.909 : 24.125, 24.187, 24.437 => 24.250
```

As can be seen, I used three PuTTY terminals to communicate over SSH with my PC to my Raspberry Pis, and another PuTTY terminal to receive serial data from my FPGA board. For the software running on the Raspberry Pis, I wrote a simple program that establishes serial communication and sends temperature data every 500 milliseconds. The program formats the data into a string prefixed by the character ‘z’, so that the software running on the FPGA board will know where the start of the string is since data is sent byte by byte. Since there is no synchronization, the program is quite simple. It simply reads the temperature and performs a serial write operation.

I performed some testing on the TMR using a can of compressed air to lower the temperature of individual temperature sensors. As expected, when I lower the temperature of one sensor significantly, the actuator ignores it, and averages the other two.



```
1638253573.970 : 24.187, 24.187, 24.500 => 24.291
1638253574.470 : 24.125, 24.250, 24.500 => 24.291
1638253574.980 : 24.187, 24.250, 24.500 => 24.312
1638253575.480 : 22.562, 24.250, 24.500 => 24.375
1638253575.980 : 18.687, 24.187, 24.500 => 24.343
1638253576.490 : 15.687, 24.187, 24.500 => 24.343
1638253576.990 : 13.625, 24.187, 24.500 => 24.343
1638253577.500 : 14.562, 24.250, 24.500 => 24.375
1638253578.000 : 13.687, 24.187, 24.500 => 24.343
1638253578.500 : 14.125, 24.187, 24.500 => 24.343
1638253579.009 : 15.312, 24.125, 24.500 => 24.312
1638253579.509 : 16.312, 24.187, 24.437 => 24.312
1638253580.009 : 17.125, 24.187, 24.437 => 24.312
```

If I lower more than one, then the average will no longer be correct. This is fine because as stated in previous sections, a TMR system is designed to tolerate up to one failure in the voter system. To tolerate more faults, the modularly redundant system would need at least 5 computers, and such a system could tolerate up to two faults or failures. Aside from that, the system works to specification, as shown in the images above.

6. Conclusion

Overall, I learned quite a bit during this project. I chose this project because I wanted to do something that was a good combination of Computer Science and Electrical Engineering. During my internship at NASA JPL over the summer of 2021, I worked on the Europa Lander Project which involved a TMR system which generated my interest in TMRs in general. My TMR system, while it does not have features like synchronization or fault correction, it is capable of tolerating a single fault, as it should. There aren’t many projects on the internet that involve TMRs or single-board computers working with FPGA, which is why I thought it would be interesting and challenging. The incorporation of sensors and an FPGA board, as well as single-board computers made this a project where I had to use nearly all of the skills I had developed in the course of my Computer Engineering degree. The TMR system I built is a faithful approximation of TMR systems used in satellites and space missions.

Work Cited

A. Eddin, *ece_4305*, California State Polytechnic University-Pomona, GitHub Repository, 2021. Accessed: Nov. 30, 2021. [Online]. Available: https://github.com/aseddin/ece_4305

B. O'Connell, "Achieving fault tolerance via robust partitioning and N-Modular Redundancy," M.S. thesis, Dept. of Aeronautics and Astronautics, M.I.T., Cambridge, MA, U.S.A., 2007. [Online]. Available: https://dspace.mit.edu/handle/1721.1/46573

DcubeTechVentures, *MCP9808*, GitHub Repository, 2016. Accessed: Nov. 30, 2021. [Online]. Available: https://github.com/DcubeTechVentures/MCP9808/blob/master/Python/MCP9808.py

F. Kastensmidt, "On the Optimal Design of Triple Modular Redundancy Logic for SRAM-based FPGAs," Design Automation and Test in Europe Conference and Exhibition, Munich, Germany, Mar. 7 – 11, 2005, 1530-1591/05.

P. Chu, *FPGA Prototyping by SystemVerilog Examples: Xilinx MicroBlaze MCS SoC*,
New Jersey: Wiley, 2018.