Designing a Microcontroller Using RISC-V in Triple Modular Redundancy for NASA'S Caution & Warning System

Authors & Designers:

Jonah Boe,

Spencer Cheney,

Chris Porter,

Kyle Tolliver

Index

List of Figures

Acronyms

CWS- Caution and Warning System

EMU- Extravehicular Mobility Unit

FIFO- First In First Out

FMC- FPGA Mezzanine Card

FPGA- Field Programmable Gate Array

HPC- High Pin Count

IP- Intellectual Property

LPC- Low Pin Count

LVDS- Low Voltage Differential Signaling

NASA- National Aeronautics and Space Administration

PLSS- Portable Life Support System

RISC- Reduced Instruction Set Computer

TMR- Triple Modular Redundancy

xEMU- Exploratory EMU

Mission Statement

Create and test a microcontroller using TMR with various configurations of the RISC-V open source instruction set on Microsemi's PolarFire radiation tolerant FPGA. This is to test the possibility of using RISC-V architecture in space, specifically the possibility of implementing this system in the CWS in NASA's newest space suit, the xEMU.

Design Specifications

Required Specifications

- Use VHDL, not Verilog/System Verilog
- Multiple configurations of a RISC-V core (at least 3)
- Create a program to benchmark and test RISC-V cores
- Microcontroller architecture with the following features:
    - 10 LVDS Full-Duplex UARTs
    - TMR
    - GPIO pins
    - SPI
    - I2C
- PCB that connects to the PolarFire's FMC HPC with the following features:
    - LCD screen
    - 3-position toggle switch
    - Ports for Full-Duplex LVDS UART using the provided FIFOs
    - LEDs and Switches for testing
    - ADC

Optional Specifications

- Program to test all microcontroller and PCB features
- Interface with program over FPGA's micro USB port
- Microcontroller architecture:
    - PWM signals
    - Normal UART
    - UART over FPGA's micro USB port
    - GPIO interrupts
- PCB:
    - Pressure and humidity sensor
    - Heartrate sensor
    - Accelerometer
    - 12v, 5v, and 3.3v ports
    - SPI, I2C, UART ports

Project Report

Microcontroller Design

There are two main parts to this project the Libero module design and the Softconsole functions used to test and operate each of the Libero modules.

Last semester's team left us with most of a working design for Libero. We edited the design to allow for a working Full Duplex UART and LVDS UART that transmits a differential signal but does not receive.

We did the most work with SoftConsole. We added modules to operate the LVDS and Full-Duplex UART, check status of 10 GPIO switches, we started creating functions to output text to the LCD screen and to control the operation through the large switch on the PCB, and debugged the code that wasn't working correctly.

LVDS Full-Duplex and Normal UARTs

The LVDS Full-Duplex UART was set up using 3 IPs, two of them are used to create an LVDS signal and the other is used to receive and transmit UART signals. The PF_IOD_GENERIC_TX_C0_0 module turns the signal into an LVDS signal but in order to operate it we have to have clocks at different frequencies and one clock that is 90 degress out of phase. We included the PF_CCC_C0 to generate these clock signals As for the Full-Duplex UART, all that needed to change from last semester was the settings of the UART_apb module so it is configured correctly for Full-Duplex. Using these modules, we were able to transmit a working differential UART signal that ranges between 0 volts for low and 2 volts for high.

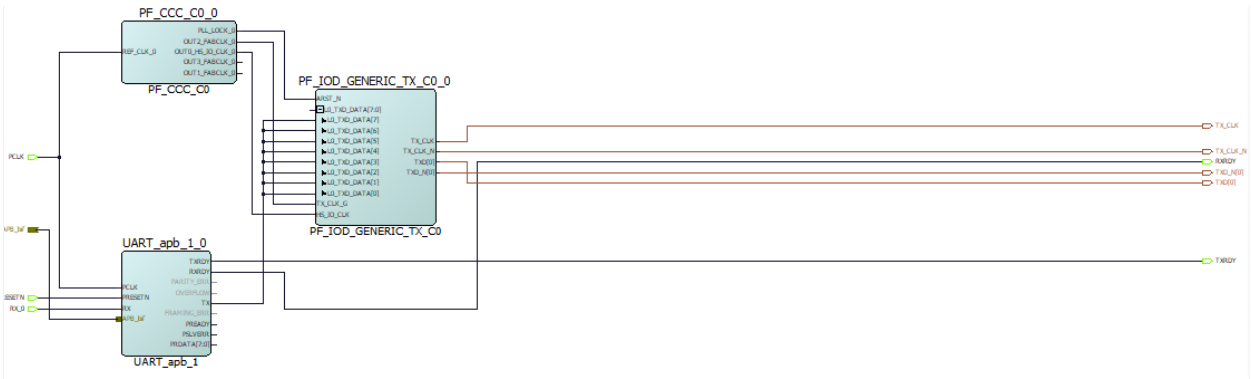

Figure 1: *Implementation of a single LVDS UART Module*

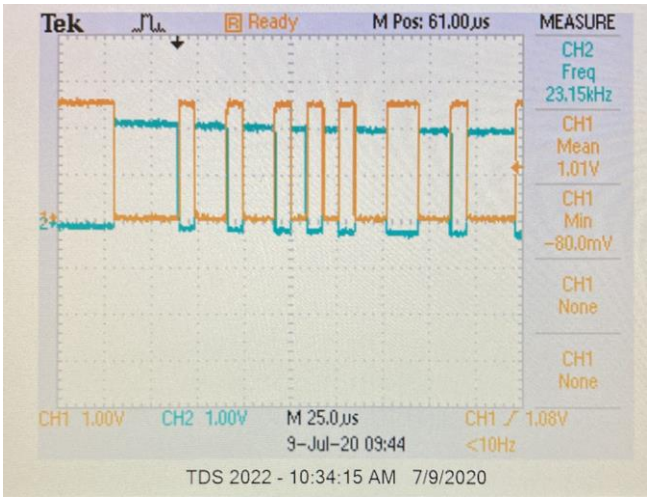

Figure 2: *Transmitting Hello Moon using LVDS Module*

The biggest problem we encountered with the UART is timing. In the SoftConsole code in order to program the UART to operate at a specific baud rate they use a clock divider equation (Baud Value = System Clock Frequency / ((16 * Baud Rate) - 1)) that becomes inaccurate when the baud rate gets to 1 MHz. The Libero LVDS module is very confusing and isn't set up to work well with UART. Instead of taking in a single signal and turning one signal into an LVDS signal it reads from an array of signals and transmits them one at a time. To combat the issue we set the have the UART Tx connect to each of the input pins of the LVDS module since this is an array of 8 signals this makes the LVDS UART signal 8 times slower than it would normally be. We were only able to get the LVDS module to output a differential signal that ranges from 0 to 2 volts. All the Libero documentations says that this is the module needed for LVDS documentation, but we couldn't figure out how to configure the settings to output an LVDS signal. If you try to fix this in the future looking at the low power settings of the LVDS module and the MIPI module might be able to resolve this issue.

Figure 3 & 4: *Left - Full Duplex Transmitting 0x31 at 1Mhz; Right - What was received, showing inaccuracy of Baud Value equation*

SPI

The SPI module is an IP that was implemented when following the tutorial. It has been changed to include five extra SPI select pins to be used for a couple SPI devices on the PCB, and for some external SPI connections.

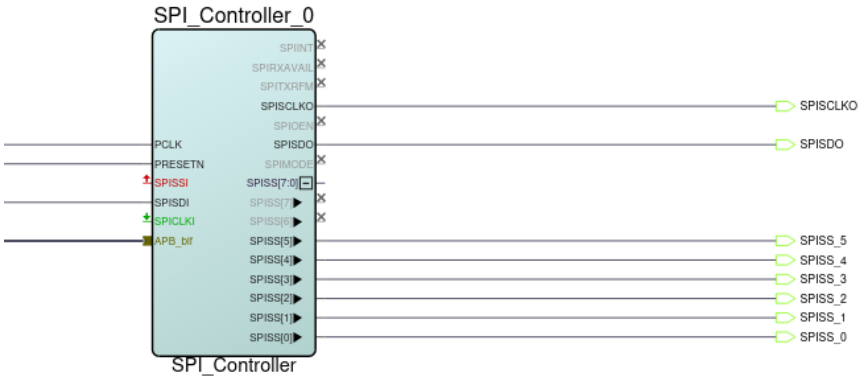

Figure 5: ***Smart Design implementation of the SPI module***

"SSISS[0]" in the figure above connects to the FPGA's SPI FLASH, which is what is flashed with the program for the processor to run. Limited testing was done for the SPI, but "SPISCLKO" would activate when sending data over SPI.

I2C

The I2C communication capabilities have been added into CPU design using a CoreI2C module. This is connected to the processor via the Advanced Peripheral Bus (APB). This module currently uses a single I2C channel set to Full Master RX/TX mode.

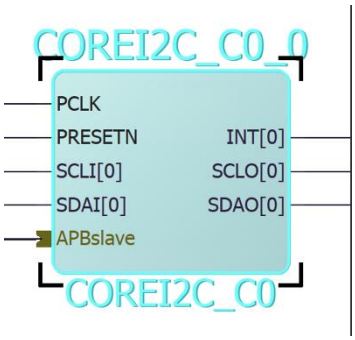

Figure 6: ***CORE I2C IP Module added to the project***

Because this module has a separate line for input and output of the serial clock and the serial data, bi-directional buffers are used to combine them. This allows for bi-directional communication using only 2 pins.

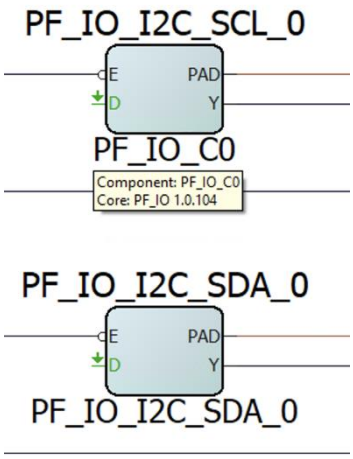

Figure 7: **Bi-directional buffers combining inputs and output lines of I2C SDA and SCL**

This I2C implementation was able to be tested to a small degree. It was verified to produce output on the SDA and SCL pins that it was assigned to. This is the extent that was able to be tested before the quarantine went into effect, so more work will probably be needed to properly configure the I2C module to communicate with the sensors.

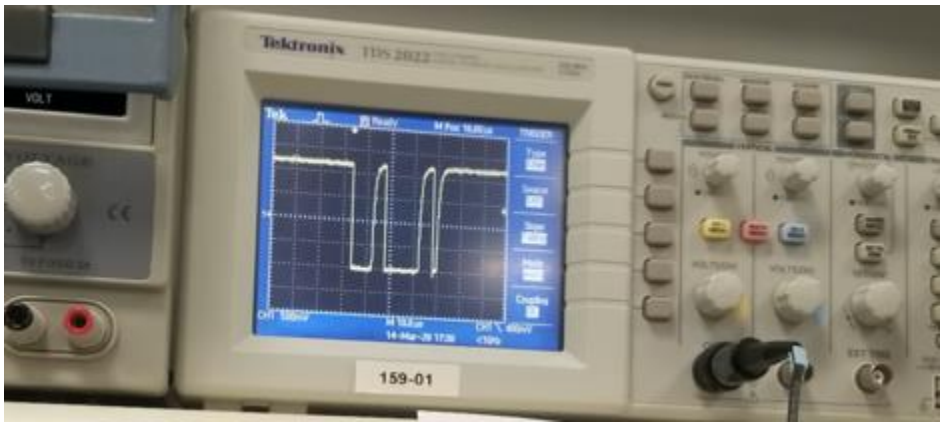

Figure 8: **I2C SDA output viewed on an oscilloscope**

GPIO

The GPIO pins initially came from the tutorial also, but have been modified to include 32 GPIO ports, both input and output (so they can be configured in software), and are able to send interrupts signals to the ISR.

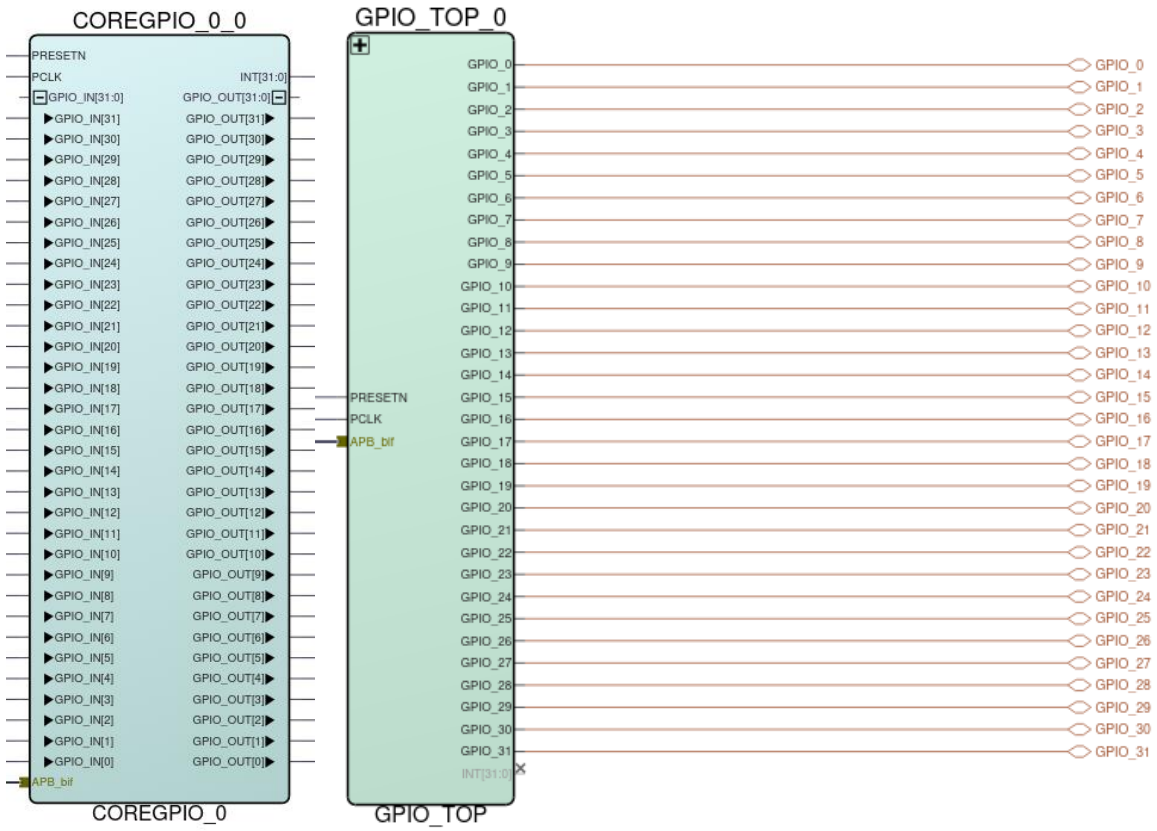

Figure 9: ***CORE GPIO IP Module (left) and GPIO Top Module (right)***

The GPIO pins were set to both input and output and the interrupt settings were set in the settings of the Core GPIO IP module. To give them the ability to be both an input and an output, a bi-directional buffer had to be added to each GPIO port (this buffer is also used for the I2C IP). To make things look cleaner in the top Smart Design, all of this was implemented in a sub-module call GPIO_TOP.

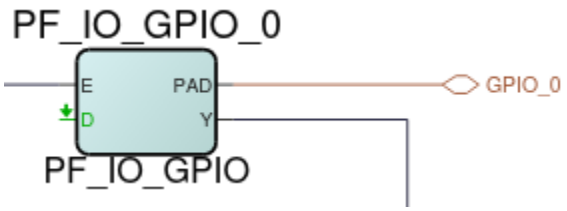

Figure 10: ***Bi-directional Buffer IP***

Not all aspects of the GPIO ports were tested. The tutorial had them set as output only pins with no interrupts attached. This worked in Software, but no testing has been done to see if the changes listed still work as expected.

PCB Assembly

Assembling the PCB was rather straight forward. The only major hiccup was the proximity of the VGA connectors to one another. The sockets would not all be accessible at the same time for connection given their proximity to one another and the dimensions of a standard VGA cable connector. For this reason, it was decided to use VGA connectors with both 90-degree and 180-degree connections, staggering them so no two of the same were side by side.

The other issue we encountered in assembling the board was that due to COVID-19 and world health conditions, it would not be viable to outsource the assembly of the PCB.

The PCB was entirely assembled and tested by the team. After all issues had been resolved, we connected the PCB to the PolarFire FPGA PCB and were able to proceed with the software tests.

Drivers and UI Design

Once a RISC-V core is built and loaded onto the FPGA, we can use SoftConsole, a software package to program softcores, to run our custom drivers and software on our cores. This is how we set up the interface with the PCB and how we planned on benchmarking the processors.

We used the user interface that was set up by last semesters team and added functionality to allow the FPGA to communicate over UART, check the status of GPIO pins, we started on using it to create an interface to work with the LCD screen, and debugged and fixed many of the issues used to communicate with sensors.

The LCD functions still need to be debugged. Something with the SPI communication or the actual LCD function isn't working so no text can be outputted to the screen.

The GPIO section of Libero still needs to be looked at. The switches in the PCB are going to a positive and negative part of the same pin and the I/O part of Libero needs to be looked at because now it is reading one switch and counting it for two.

We were not able to start writing functions in SoftConsole to communicate with the Heart Rate, Accelerometer, and Barometer sensor or the ADC on the PCB. There still may be some IPs that need to added to Libero in order to communicate with them.

Conclusion

The main goal of this semester was to get all communication, GPIO, LCD, and sensor functions created and to get one processor working with all the requirements. While we weren't able to complete all of the requirements. We were able to get to a point when with some debugging you should be able to fix all the issues and create the functions needed to get this project working.

Even with the set backs encountered with COVID-19 we were able to nearly complete all the requirements for a single core. All that is necessary is to debug and resolve the errors with the LVDS module, add in a LVDS module for receiving data, finish the function to communicate with the LCD screen, and fully test the SPI and I2C modules. Once each of these issues are fix for the core that is alright set up it should be easy to adjust the project to work with multiple types of cores.