CALIFORNIA STATE UNIVERSITY, NORTHRIDGE

Real Time Measurements of Biosignals on Development Board

A graduate project submitted in partial fulfillment of the

requirements for the degree of Master of Science in Computer Engineering

By

Luis Rivera

December 2022

The graduate project of Luis Rivera is approved:

Dr. David Hawkins Date

Dr. Xiaojun Geng Date

Dr. Ramin Roosta, Chair Date

California State University Northridge

ACKNOWLEDGEMENTS

Words cannot express my gratitude to my professor and chair of my committee Dr. Ramin Roosta for his invaluable experience, patience, and feedback. I also could not have undertaken this journey without the help of Dr. David Hawkins, who generously provided knowledge and expertise. Additionally, this endeavor would not have possible without the generous support from Dr. Xiaojun Geng.

Finally, I want to thank my family and friends, for their belief and unconditional support that has kept my spirit and motivation high.

Table of Contents

Signature Page ..................................................................................................................ii Acknowledgements ..........................................................................................................iii

List of Figures ..................................................................................................................vii

List of Tables ...................................................................................................................ix Abstract ............................................................................................................................x

[1. Theory 1](#_Toc119580966)

[1.1 Motivation 1](#_Toc119580967)

[1.2 Analysis 1](#_Toc119580968)

[1.2.1. Electrocardiography 2](#_Toc119580972)

[1.2.2. Galvanic Skin Response 3](#_Toc119580973)

[1.2.3. Temperature 4](#_Toc119580974)

[1.3 Stress Detection 6](#_Toc119580975)

[1.4 OLED Display 7](#_Toc119580976)

[1.5 SoC 7](#_Toc119580977)

[1.6 Zynq Architecture 9](#_Toc119580978)

[1.7 AXI Interface 12](#_Toc119580986)

[2. Methodology 14](#_Toc119580987)

[2.1 Sensors & Communication Protocols 14](#_Toc119580989)

[2.1.1. Pmod AD2 14](#_Toc119580993)

[2.1.2. Pmod TMP3 16](#_Toc119580994)

[2.1.3. Pmod TC1 17](#_Toc119580995)

[2.1.4. Pmod OLED 18](#_Toc119580996)

[2.2 Hardware Setup 19](#_Toc119580997)

[2.3 Software Flow 20](#_Toc119580998)

[2.3.1. Heart Rate 20](#_Toc119581004)

[2.3.2. GSR 21](#_Toc119581005)

[2.3.3. Temperature 22](#_Toc119581006)

[2.3.4. OLED Display 23](#_Toc119581015)

[3. Implementation 25](#_Toc119581016)

[3.1 SoC Design 25](#_Toc119581018)

[3.2 Constraints 28](#_Toc119581019)

[4. Verification 32](#_Toc119581020)

[4.1 MATLAB 32](#_Toc119581022)

[4.2 Testing Heart Rate 36](#_Toc119581023)

[4.3 Testing GSR 36](#_Toc119581024)

[4.4 Testing Temperature 37](#_Toc119581025)

[4.5 Testing OLED Display 37](#_Toc119581026)

[4.6 Results 38](#_Toc119581035)

[4.7 Earlier Prototype 39](#_Toc119581036)

[5. Future Work 41](#_Toc119581037)

[5.1 Bluetooth 41](#_Toc119581039)

[5.2 WIFI 41](#_Toc119581040)

[5.3 BPM Algorithm 41](#_Toc119581041)

[5.3.1. Hardware Filter 41](#_Toc119581048)

[5.3.2. Circular Buffer 43](#_Toc119581049)

[5.4 Machine Learning 43](#_Toc119581050)

[5.5 Prototype Glove 44](#_Toc119581051)

[6. Conclusion 45](#_Toc119581052)

[Bibliography 46](#_Toc119581053)

[Appendix A: Vitis Source Code 47](#_Toc119581054)

[Appendix B: Vivado Block Design Constraints 54](#_Toc119581055)

[Appendix C: Sensors w/ OLED Display Vivado Block Design 56](#_Toc119581056)

[Appendix D: Sensors Vivado Block Design 57](#_Toc119581057)

[Appendix E: Sensors w/ stress detection Vivado Block Design 58](#_Toc119581058)

[Appendix F: GitHub Source Code 59](#_Toc119581059)

List of Figures

[Figure 1. Pulse Sensor Theory 2](#_Toc119577579)

[Figure 2. ECG Signal 3](#_Toc119577580)

[Figure 3. GSR Circuit Diagram 4](#_Toc119577581)

[Figure 4. K-type Thermocouple Concept 5](#_Toc119577582)

[Figure 5. K-Type Thermocouple Circuit 6](#_Toc119577583)

[Figure 6. Conceptual Zynq SoC 8](#_Toc119577584)

[Figure 7. Conceptual Configurable Logic Block 9](#_Toc119577585)

[Figure 8. Zynq SoC 10](#_Toc119577586)

[Figure 9. Zynq SoC Architecture 11](#_Toc119577587)

[Figure 10. Zynq PL Architecture 12](#_Toc119577588)

[Figure 11. AXI Architecture Design 13](#_Toc119577589)

[Figure 12. AD7991 Functional Block Diagram 15](#_Toc119577590)

[Figure 13. I2C Timing Diagram 16](#_Toc119577591)

[Figure 14. I2C Connections Diagram 16](#_Toc119577592)

[Figure 15. TCN75A Typical Application Circuit 17](#_Toc119577593)

[Figure 16. MAX31855 Block Diagram 17](#_Toc119577594)

[Figure 17. SPI Connections Diagram 18](#_Toc119577595)

[Figure 18. SSD1306 Block Diagram 19](#_Toc119577596)

[Figure 19. Pulse Sensor using Pmod AD2 Flowchart 21](#_Toc119577597)

[Figure 20. GSR Sensor using Pmod AD2 Flowchart 22](#_Toc119577598)

[Figure 21. Pmod TC1 Flowchart 23](#_Toc119577599)

[Figure 22. Simplified block design 25](#_Toc119577600)

[Figure 23. Sensors with OLED display block design 28](#_Toc119577601)

[Figure 24. Pmod Connectors 29](#_Toc119577602)

[Figure 25. ADC Data from Pulse Sensor Example 1 32](#_Toc119577603)

[Figure 26. Histogram 1 34](#_Toc119577604)

[Figure 27. ADC Data from Pulse Sensor Example 2 34](#_Toc119577605)

[Figure 28. Histogram 2 34](#_Toc119577606)

[Figure 29. First 30 seconds of ADC Data from Pulse Sensor Example 1 35](#_Toc119577607)

[Figure 30. Power Spectrum of ADC Data from Pulse Sensor Example 1 35](#_Toc119577608)

[Figure 31. GSR Voltage and Resistance Table 37](#_Toc119577609)

[Figure 32. Image of project in action 38](#_Toc119577610)

[Figure 33. Live Demo with Sensors 39](#_Toc119577611)

[Figure 34. Sensors with stress detection module 40](#_Toc119577612)

[Figure 35. Simplified block design 40](#_Toc119577613)

[Figure 36. Oscilloscope Reading of Pulse Sensor 42](#_Toc119577614)

List of Tables

[Table 1. Constraints for Pmod AD2 30](#_Toc119581060)

[Table 2. Constraints for Pmod TC1 30](#_Toc119581061)

[Table 3. Constraints for Pmod OLED 30](#_Toc119581062)

Abstract

Real Time Measurements of Biosignals on Development Board

By

Luis Rivera

Master of Science in Computer Engineering

This graduate project implements real time analysis on multiple sensors connected to a Digilent Zybo Z7-20 development board, reference can be found here Figure 33. The development board used Digilent Pmod’s to interface with sensors that connect to a human for measuring biosignals in real time. Once the development board is connected to the sensors (heart rate, galvanic skin response and temperature), it will perform processing on the received data from the sensors and calculate corresponding beats per minute (BPM), skin conductance and temperature of subject. The outcome of this project is a prototype wearable device that can measure vital signals of the user and identify stress using a stress detection algorithm.

# Theory

# Motivation

For many years, electronic devices have been used to diagnose patients and analysis of biometric data, required expensive equipment that only medical practitioners can afford. With advancements in technology, biometric sensors and processors have significantly reduced in size and area, to the extent that we can now wear these devices on our wrists. By nature, people have become curious to monitor their own health, from the comfort of home with such wearable devices.

As the rise in technology increases, a trend for wearable devices that can be incorporated in everyday life are in demand. Being able to determine one’s biometric data such as BPM, skin conductance and temperature can be utilized to help understand one’s medical or emotional state. For example, smartwatches from companies such as Apple, Fossil, Fitbit and others, record user’s vitals such as BPM, sleeping patterns, level of oxygen in blood, GPS tracking for exercise and other features. Wearable medical or fitness devices can help people improve health and fitness goals.

# Analysis

Understanding the sensor capabilities and how they operate will be useful knowledge for implementation and integration. Knowing how the sensor functions, will give the necessary background in determining the logic and filtering needed to calculate each respective measurement.



# Electrocardiography

The pulse sensor used was the [Sen-11574](https://www.sparkfun.com/products/11574) that works by shining a green LED light on the finger and measuring the amount of light reflected with a photosensor; which can be translated to measuring the change in flow of blood in each heartbeat. The analog signal generated from the photosensor usually is small and noisy, therefore an RC filter and Op Amp are used. The RC filter is used to remove noise while the Op Amp is used to amplify the few millivolts output, before being sampled by the analog-to-digital converter (ADC).

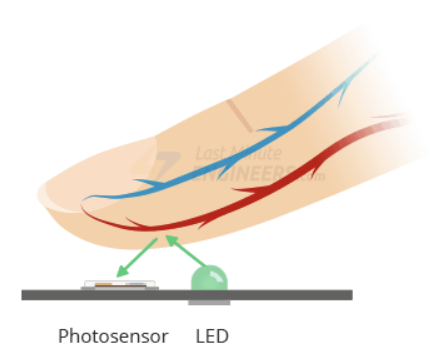


Figure 1. Pulse Sensor Theory

After the whole process to measure the amount of light reflected, filter and amplify the signal, this signal can be referred to as an electrocardiogram (ECG). For this project’s purpose, understanding when the ECG signal is increasing, is sufficient to measure BPM. The QRS complex is composed of the Q, R, and S wave, illustrated by Figure 2. The increasing part of the QRS complex happens to be from the Q to R wave. By detecting the increasing part of the ECG signal, we can measure a person’s BPM. Many different approaches exist for measuring BPM with an ECG signal but the Q to R wave is one of the simplest and most effective methods to understand and implement.

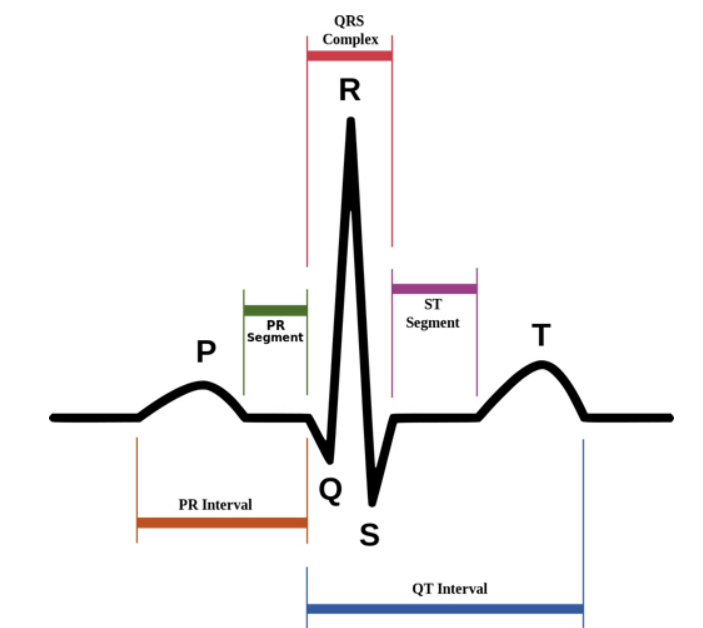


Figure 2. ECG Signal

# Galvanic Skin Response

The [Grove-GSR](https://wiki.seeedstudio.com/Grove-GSR_Sensor/) was used for the galvanic skin response (GSR) sensor that measures electrical conductance of the skin, this can be referred to as sweat gland activity or emotional arousal. This electrodermal activity is strongly related to moods such as happy, sad, afraid, nervous or others that can be related to stress. The GSR sensor works by using two electrodes that apply a small constant voltage usually 0.5V and measuring the variations in voltage through the electrodes, seen in Figure 3. Changes in the voltage are related to sweat gland activity or moods.

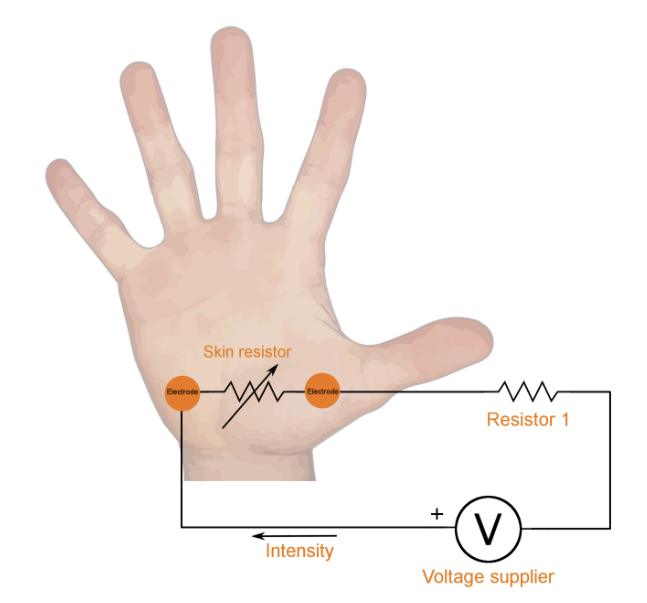


Figure 3. GSR Circuit Diagram

When placing the two electrodes of the GSR sensor on a hand like Figure 3, the circuit closes as your hand behaves as a resistor. The output of the GSR sensor is a voltage that will be converted to resistance to determine skin conductance. Conductance is inversely proportional to Ohms and calculating conductance can approximate for sweat gland activity. This is assuming the subject does not have sweaty hands in the beginning or in an environment that causes the fingers to be moist, therefore fingers should be wiped dry before experimentation. Measuring GSR has been shown to be an extremely accurate method to determine emotional stress.

# Temperature

A [Pmod TMP3](https://digilent.com/shop/pmod-tmp3-digital-temperature-sensor/) was used an ambient temperature sensor for measuring the environment temperature that can be extra parameter for the project but note, this is not an accurate approach since this assumes the person is a similar temperature to the environment. The sensor works by measuring their own die temperature that is a voltage across a bandgap and uses the difference in the base-emitter voltage () of a transistor (BJT) to relate voltage to temperature. A Sigma-Delta ADC is used to convert () to a digital word that relates to the transistor temperature. The output of this sensor is a digital signal encoded with the temperature. After integration, the decision was made to upgrade the Pmod TMP3 for a temperature sensor with contact leads or that is more appropriate for body temperature or holding.

The alternative was a [Pmod TC1](https://digilent.com/shop/pmod-tc1-k-type-thermocouple-module-with-wire/) that is a K-type thermocouple with wide temperature range -73°C to 482°C and has accuracy of 2°C. The output of thermocouples, range from B, E, J, K, N, R, S and T type but typically output a small voltage in the mV range. Thermocouples J, K and T type are the most common and at room temperature, their voltage varies from 52µV/°C, 41 µV/°C, and 41 µV/°C while other less-common types are even smaller. This sensor works by measuring the cold junction that consists of two different electrical conductors that form a thermal junction. As the temperature changes at the junction, the voltage measured can be translated to a corresponding temperature, illustrated in Figure 4.

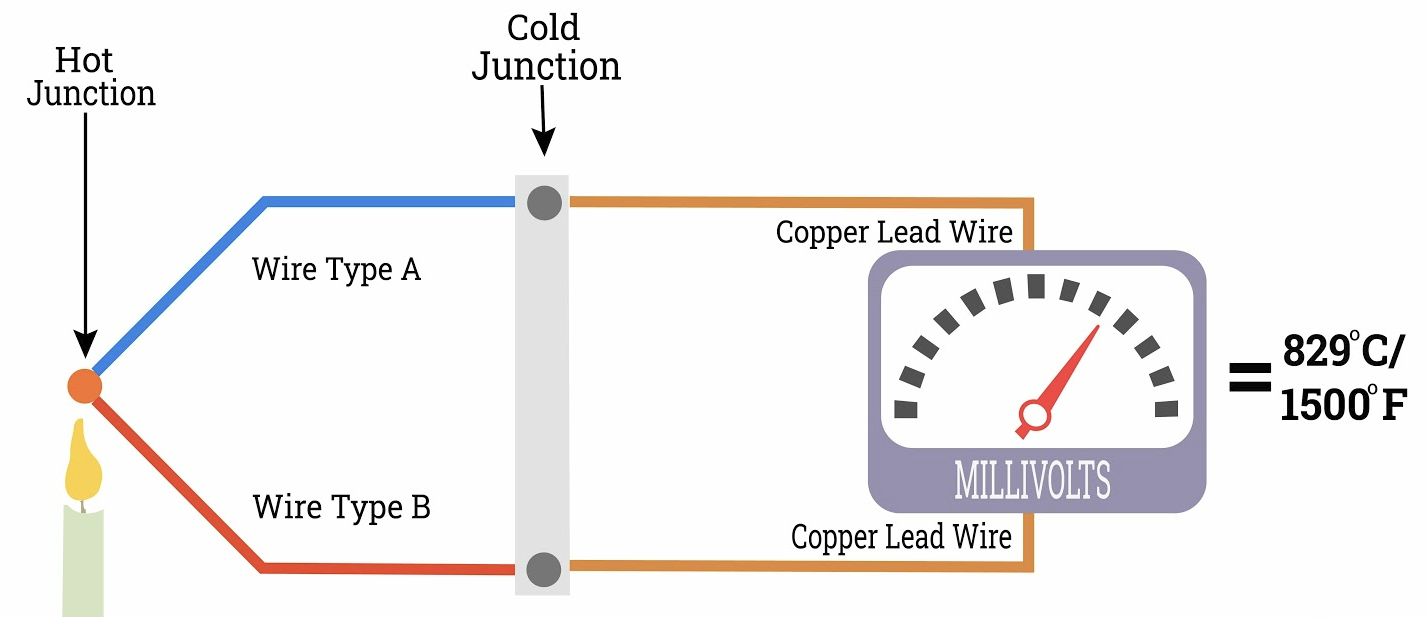


Figure 4. K-type Thermocouple Concept

For an external device such as the development board, there needs to be an amplifier, signal conditioning and cold junction compensation sensor to interface with thermocouple, like Figure 5. There are several modules such as the Pmod TC1 that are equipped with a temperature sensor to measure the reference junction of the k-type thermocouple, amplifier to increase voltage and meet input requirements of FPGA system and an ADC.

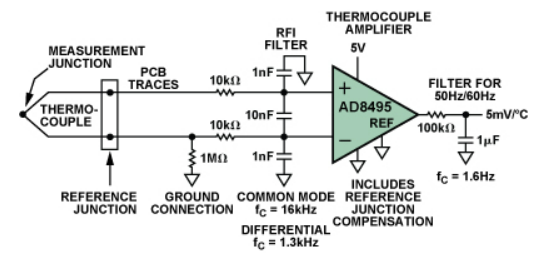


Figure 5. K-Type Thermocouple Circuit

# Stress Detection

With all the sensors used for this project such as heart rate, GSR and temperature, a stress detection algorithm can be used to perform stress analysis. By observing the sensor outputs, if the heart rate is 110-130 BPM, then it can be perceived as you are exercising, normally have a fast heartbeat, fever, or other conditions. Now if the temperature is high like above 100ºF or 37.78°C, then maybe the subject is in a really warm environment, ill, intense physical activity or other factors. With the two results, it’s not bad judgement to say the subject is stressed since BPM and temperature measurements are higher than normal. The stress detection algorithm will receive the measurements from the sensors as inputs and will output a status of stressed or not.

Each person has a unique emotional response, so it would be difficult to interpret the same set of sensor readings on two different people as corresponding to the same emotional state. To accommodate the unique features of each user, testing different scenarios on the subject for different emotional states are needed. The relationship between input and output can be stored in memory to recall what values output a stressed result, this can be thought of as the training mode. After training on a few subjects, a real time application can be implemented using the training mode results to help predict if the subject is stressed with more accurate results. For a more modern approach, a machine learning algorithm can be implemented to learn from previous subjects and still learn from new patients by storing data in memory.

# OLED Display

A [Pmod OLED](https://digilent.com/reference/pmod/pmodoled/start) display was used for the project, the measurements from the sensors can be displayed on the OLED. Results are displayed on the Vitis GUI terminal but to make the embedded system more user friendly, results will be printed on OLED display.

# SoC

A [System on Chip](https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html) (SoC) combines the software programmability of ARM processors with the hardware programmability of a [Field Programmable Gate Array](https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html) (FPGA). An FPGA is a reprogrammable logic semiconductor device containing a matrix of configurable logic blocks (CLBs) that are interconnected with wires. CLBs allows the device to implement combinational and sequential logic using Flip-Flops, Look-up Tables, and Multiplexers. FPGAs can be programmed using Hardware Description Language (HDL) such as VHDL, Verilog and System Verilog while ARM processors and connected peripherals can be programmed using C/C++.

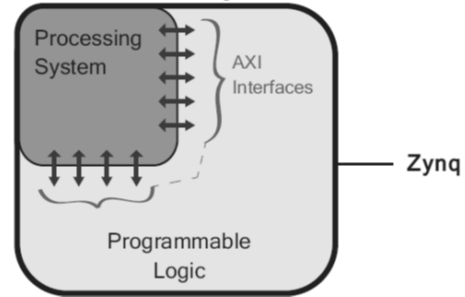


Figure 6. Conceptual Zynq SoC

CLBs are essential to the FPGA as they contain the design logic and can be broken down to two main structures called slices. Each slice consists of LUTs, FFs, multiplexers and depending on the FPGA family, the elements may vary. If the slice contains memory such as RAM or LUTs are used as memory, it can be referred to as SLICEM since it has the capability of behaving like distributed memory. If no memory, more than likely it is a SLICEL which has exclusive logic functionality. Key distinction between the two slices is one has memory capability while the other doesn’t.

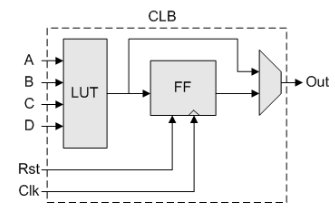


Figure 7. Conceptual Configurable Logic Block

# Zynq Architecture

The Digilent Zybo Z7-20 development board contains a Xilinx Zynq-7000 device is equipped with a dual-core ARM Cortex-A9 processor and an FPGA. The general architecture of a Zynq device consists of two parts: the Processing System (PS) and Programmable Logic (PL). A Zynq device is a SoC since PS is the processor and PL is the FPGA fabric, illustrated in Figure 6. PS is better used for dynamic tasks, general purpose sequential tasks, operating system, and complicated logic controls assuming the processor can keep up, while the PL is good for static parallel tasks, peripheral controls, and intensive data computations.

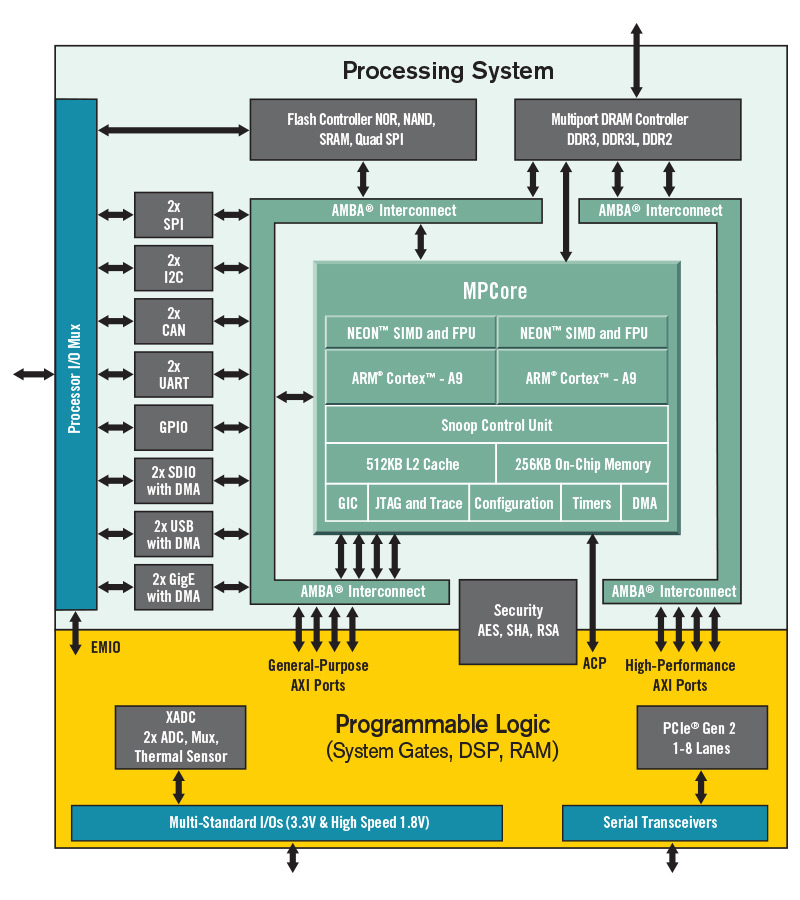


Figure 8. Zynq SoC

The PS includes ARM Cortex-A9, peripheral interfaces, cache memory, memory interfaces, clock generation, interconnects and more. The PS can communicate directly with external interfaces such as sensors via Multiplexed Input/Output (MIO) that provides 54 pins of flexible connectivity. Extended MIO (EMIO) connects the PS to the fabric and can be used to connect to external devices through the FPGA PL pins. The I/O peripherals on the PS include communication protocols such as SPI, I2C, CAN, UART, SD, USB, GigE and General-Purpose Input/Output (GPIO) to control buttons, switches and LEDs.

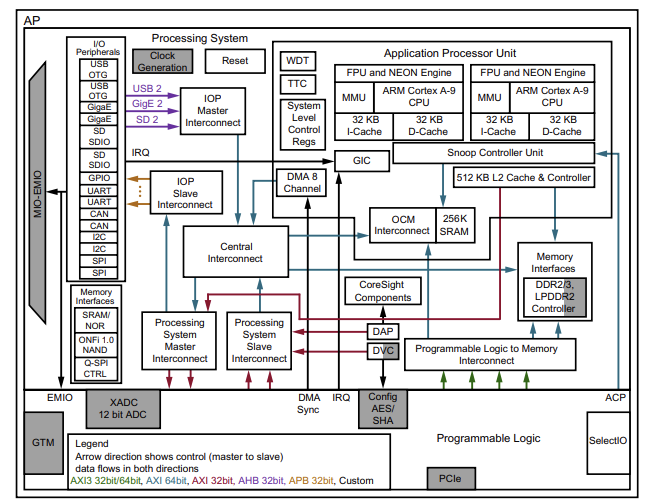


Figure 9. Zynq SoC Architecture

The PL primarily consists of general-purpose FPGA logic fabric that is composed of components like Block RAM (BRAM) for embedded memory, DSP48E1 slice for high-speed arithmetic operations, PCIe general purpose serial interconnect and XADC for analog-to-digital signal conversion. The Input/Output blocks are GPIO interfaces permitting up to 3.3V that contain a pad, to provide a physical connection to the outside world for a single input or output signal.

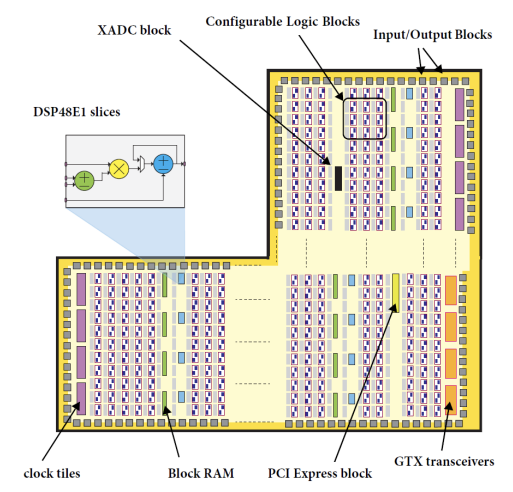


Figure 10. Zynq PL Architecture



# AXI Interface

To integrate the PS and PL, [Advanced eXtensible Interface](https://www.xilinx.com/products/intellectual-property/axi.html#details) (AXI) buses are used to connect PS to other blocks on the PL. This protocol enhances the performance and utilization of the interconnect when used by multiple masters, allowing better bandwidth and low latency. There are three types of AXI bus protocols: AXI4, AXI4-Lite, and AXI4-Stream. AXI4 is a memory-mapped link, which an address is supplied followed by a data transfer of up to 256 data words; while AXI4-Lite is a simpler version that supplies an address, and a single data word can be transferred. AXI4-Stream is the only nonmemory-mapped link that supports burst transfers of any size.

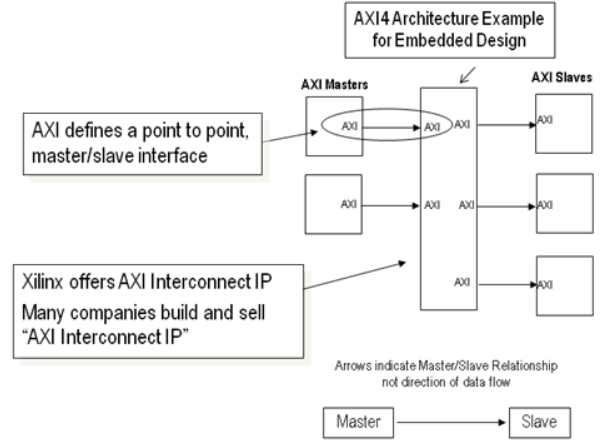


Figure 11. AXI Architecture Design

This makes SoC designs very versatile and powerful as they can perform well for several applications ranging from Aerospace & Defense, ASIC Prototyping, Automotive, High-Performance Computing and Data Storage, Machine Learning & Vision, and Medical among other. The most important aspect of the Digilent Zybo Z7-20 development board is that it can be programmed to desired applications and as the design evolves or complexes, board can be reprogrammed to meet new requirements.

# Methodology



# Sensors & Communication Protocols

For this project, the sensors utilized required an overall of two different communication protocols such as Inter-Integrated Circuit (I2C), and Serial Peripheral Interconnect (SPI) but to print on the Vitis terminal, Universal Asynchronous Receiver Transmitter (UART) was used. For any external device to communicate with any sensor, both devices must be using the same protocol.



# Pmod AD2

The heart rate and galvanic skin response sensor both have an analog output. To communicate or interface with either sensor, an ADC must be used to interpret the output voltage and convert to a digital signal so the FPGA can process. The development board has an onboard ADC peripheral known as the “XADC”; however, the input range is limited to 1V. The typical output of heart rate and GSR sensor is 1-1.5V, therefore if the XADC is to be utilized, the output of both sensors must be attenuated using a voltage divider. This required additional circuitry, therefore a Pmod AD2 module was purchased since the analog input range is 0 to power supply input () and meets requirements. Note, range is from 2.7V to 5.5V.

The [Pmod AD2](https://digilent.com/reference/pmod/pmodad2/start) uses the [AD7991](https://www.analog.com/media/en/technical-documentation/data-sheets/AD7991_7995_7999.pdf?_ga=2.94889308.1924887543.1667439250-829788110.1643500797) IC chip that can read up to four channels sequentially with up to 12-bit resolution, found in Figure 12. Therefore, the development board must use I2C to send/receive data from the Pmod AD2, which is connected to the sensors. I2C is a serial communication protocol that transfers data, bit by bit sequentially, while only using two wires to transmit/receive data (SDA) and another for the clock (SCL), hence synchronous device. Note, SDA and SCL must be connected to pull up resistors because I2C devices are open drain requiring an initial state of logic high on the bus, otherwise the bus does not work.

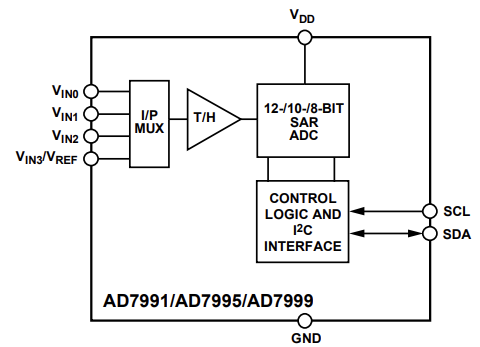


Figure 12. AD7991 Functional Block Diagram

I2C can be broken down into start/stop conditions, 7-bit slave address, read/write bit, ACK/NACK bit and the data bits, illustrated in Figure 13. The start condition is used to signal the start of a data transfer, while the stop condition is for the end of transmission, which is initiated by the master device. I2C allows for multiple masters and slaves and some devices can behave as both master and slave, illustrated in Figure 14. To distinguish between devices, a 7-bit or 10-bit address is needed at the start of each data transfer to identify the slave in communication. A read/write bit is used to determine if the slave is reading from or writing to the master. When a byte of data is being transferred, there is an ack/nack bit that ensures each byte is successfully transferred and no corruption of data. Lastly, the last byte is the data bits to be transferred to the receiving device.

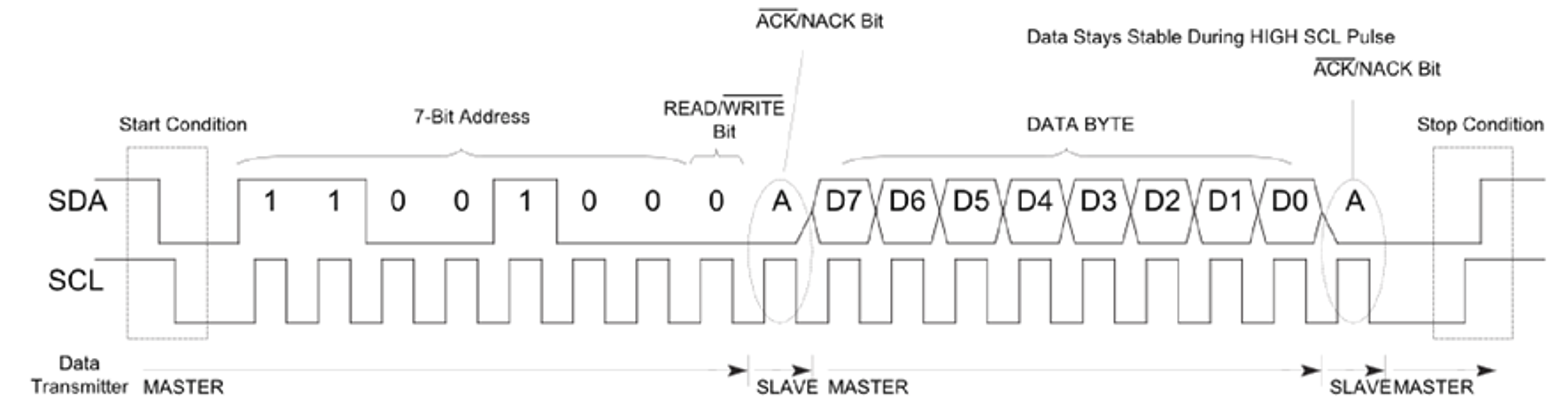


Figure 13. I2C Timing Diagram

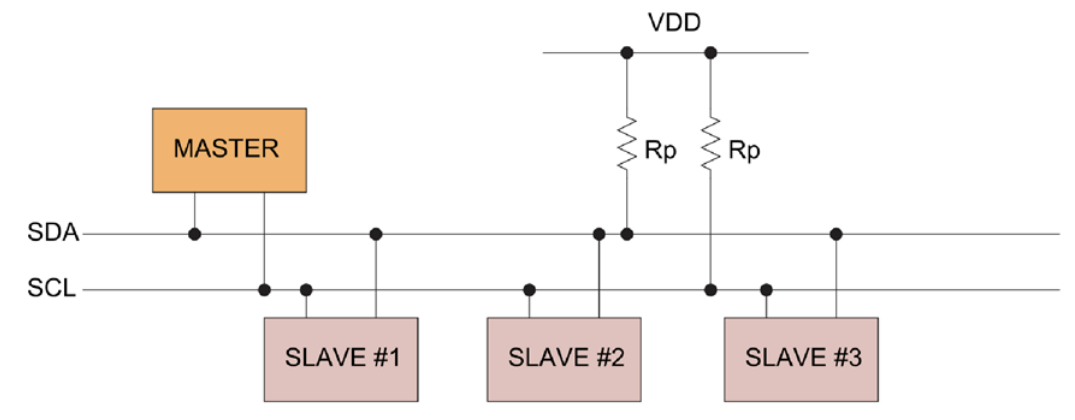


Figure 14. I2C Connections Diagram

# Pmod TMP3

The [Pmod TMP3](https://digilent.com/shop/pmod-tmp3-digital-temperature-sensor/) uses the [TCN75A](https://ww1.microchip.com/downloads/aemDocuments/documents/OTH/ProductDocuments/DataSheets/21935D.pdf) IC chip that outputs a signed 12-bit digital signal via I2C, illustrated in Figure 15. For the project, a sensor that is more appropriate for body temperature is needed, therefore not much will be discussed about this Pmod, and communication protocol is same as Pmod AD2.

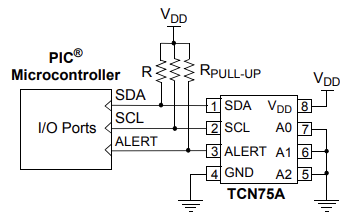


Figure 15. TCN75A Typical Application Circuit

# Pmod TC1

The [Pmod TC1](https://digilent.com/shop/pmod-tc1-k-type-thermocouple-module-with-wire/) uses the [MAX31855](https://datasheets.maximintegrated.com/en/ds/MAX31855.pdf) IC chip to perform cold junction compensation, ADC, and outputs a signed 14-bit digital signal via SPI, illustrated in Figure 16. The data is transmitted from the ADC to a digital controller that is SPI compatible.

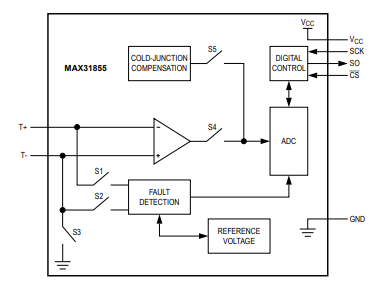


Figure 16. MAX31855 Block Diagram

SPI can be connected to multiple slaves if they each have unique slave select signals and the devices tri-state their outputs when not selected; furthermore, operation is controlled by 3 inputs SCLK, MOSI, CS# and output MISO, illustrated in Figure 17. This protocol is synchronous, therefore there is a clock signal (SCLK) and chip select (CS) is used to enable device for reading/writing. Transmission of data from master to slave is on the Master Output Slave Input (MOSI) bus while sending data from slave to master is on Master Input Slave Output (MISO). The master is denoted by the controller device that drives the SCLK, CS and MOSI pins. Note, SPI is the fastest communication protocol compared to UART and I2C.

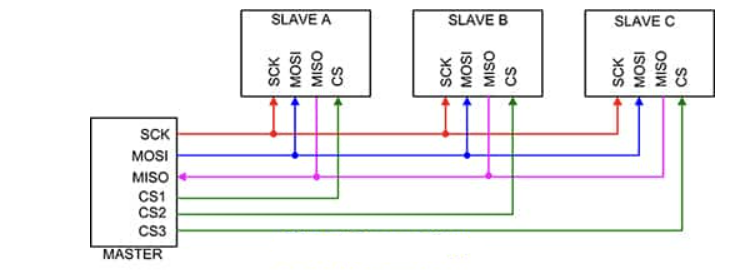


Figure 17. SPI Connections Diagram

SPI Timing Diagram and Explanation.

# Pmod OLED

A wearable device is to be prototyped and since Bluetooth or WIFI were not used, an OLED display is used to print the sensor measurements. The [Pmod OLED](https://digilent.com/reference/pmod/pmodoled/start) uses an IC [SSD1306](https://cdn-shop.adafruit.com/datasheets/SSD1306.pdf?_ga=2.261924301.1924887543.1667439250-829788110.1643500797) display controller. The OLED display is an SPI peripheral that uses a display controller chip. The embedded display only supports SPI write and has a variety of features such as turn on individual pixels, print predefined characters or load bitmaps onto the screen.

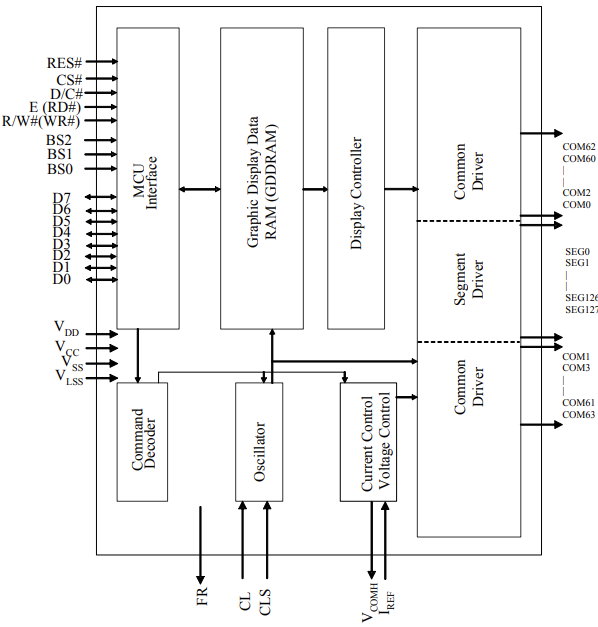


Figure 18. SSD1306 Block Diagram

# Hardware Setup

After having a general idea of how the sensors work and what communication protocols are needed, the FPGA can be configured. Like mentioned before, the heart rate and GSR sensor will be connected to a Pmod AD2 that uses I2C while the temperature uses SPI and connects directly to the FPGA Pmod connectors. For the FPGA to interface with the sensors, I2C and SPI must be enabled. Various approaches can be implemented as each peripheral can be enabled on the processing system (PS) or programmable logic (PL) of the FPGA. Enabling the communication peripherals will allow the FPGA to read data from the sensors and once the data is retrieved, the processing can start.

# Software Flow

For the “fit bit product”, the algorithm implemented different configurations for each sensor, read the data and used signal processing to clean up the signal.

Each Pmod requires different initializations, configurations, registers, and signal processing. Flow charts will be used to explain the software implementation and details.



# Heart Rate

For the BPM algorithm, the Pmod ADC samples used signal processing to isolate the pulses and detect peaks or rising edges more effectively, the flowchart can be found here Figure 19. Before initiating any read or write operations, the Pmod must be initialized and configured. After initializing and configuring Pmod AD2 to read from channel 1 (, which is connected to pulse sensor), next step is to read from the device and store conversion into a variable or array. Note, the output result will be 16-bits but only the LSB 12-bits are relevant, therefore variable with ADC conversion will be masked with 0xFFF.

To filter the ECG signal, a moving average filter was used that behaves as a lowpass filter. The high frequency content is of interest so by using the moving average result, it can be used as a threshold to not accept lesser values. In an ECG signal, the pulse or peak is the highest value or point in the signal. By looking at the digital value or voltage representation, the moving average can easily be used in a conditional statement for filtering by comparing the ADC conversion with moving average. During implementation, it was noticed that simply detecting rising edges it was not efficient enough.

To determine BPM,

For further signal conditioning, by comparing the ADC conversion with moving average, another joint condition of checking previous ADC conversion to be below the threshold is applied. This ensures that the sample before detecting the pulse, is below the moving average threshold, which validates the pulse and helps not detecting pulses in the same period.

After detecting two rising edges or pulses, BPM can be calculated by dividing 60 with the elapsed time between those two pulses. This can further be improved by adding another conditional logic to ensure that the elapsed time is greater than 0.4 seconds since any elapsed time less than 0.4 will output a high BPM > 150. Using the histogram function from MATLAB, helped selecting a reasonable offset for adding to the moving average, for a more effective threshold.

Diagram

Description automatically generated

Figure 19. Pulse Sensor using Pmod AD2 Flowchart

# GSR

Work in progress! output.

Graphical user interface, table, Word

Description automatically generated with medium confidence

Figure 20. GSR Sensor using Pmod AD2 Flowchart

# Temperature

The software implementation for temperature is relatively easy since it’s plug and play. For using the Pmod TC1, there is three steps: creating instance of Pmod device, initializing Pmod TC1, and computing Celsius and Fahrenheit measurement.

Diagram, text

Description automatically generated

Figure 21. Pmod TC1 Flowchart



# OLED Display

The software implementation.

Table

Description automatically generated with low confidence

Figure 22. Pmod OLED Flowchart

# Implementation



# SoC Design

To configure the SoC, a block design can be created to implement desired PS and PL blocks. All IPs will be implemented in the PL such as SPI and I2C Master to communicate with external sensors Pmod AD2, TC1 and OLED display. A Zynq PS is used, and the enabled peripherals can be connected to an AXI Interconnect that allows communication between PS-PL.

The Pmod IP blocks are custom IPs designed by Digilent with specific configurations for each specific Pmod. Pmod IP’s such as PmodTC\_v1\_0 and PmodOLED\_v1\_0 are simply SPI controllers and PmodAD2\_v1\_0 is an I2C controller, illustrated in Figure 23 . Similar design can be implemented using two SPI controllers and a single I2C controller.

Diagram

Description automatically generated

Figure 23. Simplified block design

Six logic blocks are used for this system in which two are generated by the development environment and the others need to be enabled from the IP catalog, illustrated in Figure 24. By using the block and connection automation tool, blocks will be generated to connect the design and create ports such as DDR and Fixed\_IO that connect directly to the PS. The blocks generated are the Processor System Reset and AXI Peripherals Interconnect.

1. **Zynq Processing System**

This is the instanton of the PS that must be included in every project for the ARM processor to run the application. The PS block has configuration options that can only be set at the hardware level such as enabling interfaces, defining clock frequency, setting interrupts, managing MIO configurations, and many more. For this project, the following configurations were done:

* Clock configuration: The CPU clock is automatically set to 667MHz along with a default PL fabric clock at 50MHz. The PL fabric clock sets the PS and PL to the same clock frequency.
* MIO Configuration: UART1 interface enabled with IO property set to MIO. This allows an output port to be defined for this block design which can stream data.
* PS-PL Configuration: UART1 should be set to a baud rate of 115,200.

1. **Processor System Reset**

This block is automatically generated by adding the PS to a hardware design which provides resets to the entire system such as peripherals, interconnects and the processor. All connections for this block are automated using block and connection automation tool.

1. **AXI Peripherals Connect**

Generated when a PS and PL block are not connected in a hardware design. This block allows peripherals to be connected using AXI interconnect which provides communication between PS-PL. Connections are automated as well.

1. **Pmod AD2**

The Pmod AD2 needs to be enabled from the IP catalog for usage which will perform ADC conversions. Recall, the Pmod AD2 has four channels with up-to 12-bit resolution and reads from the channels sequentially. The interface output port of the Pmod must be made into an external connection so the IP block can be mapped to the FPGA pins that connect to the physical Pmod AD2.

After integrating and testing the system, it was noticed that when using the Pmod AD2 on the heart rate sensor alone, the BPM calculations were spot on. When two sensors such as heart rate and GSR are connected to the Pmod AD2, the device begins to read from the channels sequentially. This creates a small delay that can corrupt the BPM calculations because it’s almost like the sample rate is halved due to the sequential reads and the UART can only print so fast.

One way to resolve this is by increasing the UART baud rate to 115,200 so it can print the measurement much quicker and can move onto the next sequential read sooner. Therefore, the baud rate for the UART needs to be set to 115,200 otherwise the BPM calculation will be skewed.

1. **Pmod OLED**

The OLED display needs to be enabled from the IP catalog to be included in the design and will allow writing to the display such as printing the measurements, text, or an image. The interface output port needs to be made into an external connection allowing the pins to be mapped to the FPGA pins.

1. **Pmod TC1**

Just like previous Pmod’s, this IP block can be enabled from the IP catalog. The Pmod TC1 is a read-only device meaning that the Zynq (master) can only read from the module. Unlike the previous Pmod’s, only four ports of the Pmod interface output will be made into an external connection such as Pmod\_out\_pin1\_o, Pmod\_out\_pin2\_o, Pmod\_out\_pin3\_o and Pmod\_out\_pin4\_o. I renamed the external ports as CS, MOSI, MISO, and SCK (pin1-4) to map the FPGA pins easier.

Diagram, schematic

Description automatically generated

Figure 24. Sensors with OLED display block design

# Constraints

After creating HDL wrapper of block design and successfully synthesizing design, the external output interfaces and ports can be mapped creating a constraint file or configuring the I/O ports from IO planning layout. I synthesized the design and then used the IO planning layout since the GUI and constraint file can sometimes have bugs like not setting the pins to the correct Pmod port.

The Zybo Z7-20 has six Pmod ports with a variety of capabilities such as high speed, standard, MIO and XADC. For the current hardware setup, only Pmod ports JB-JC are needed, and they are high speed ports. These ports allow for maximum switching speeds by routing the data signals as impedance matched differential signals.

The Zybo Z7-20 pins follow the convention on Figure[x] and the interface output ports must be mapped to corresponding Pmod ports to interface with sensors and modules correctly. When mapping Pmod or I/O ports, sometimes additional hardware setup is needed. For example, using an I2C peripheral requires pull-up resistors on the SDA and SCK wire to keep the digital output at a defined logic state, no ambiguity.

With peripherals like Pmod TMP3, come with jumper pins that allow you to enable pull-up or pull-down resistors, other modules provide an actual resistor ranging from 4.7k-10k, and other cases, they don’t provide you the resistor. Note, the IO planning layout displays all options for configuring the I/O ports. In the I/O port settings, there are options like setting the I/O standard, drive strength, slew type, and pull type among others.



Figure 25. Pmod Connectors

Pmod ports JB-JD were utilized, and JB was mapped to Pmod AD2, JC to Pmod TC1, and JD to Pmod OLED. For the Pmod AD2, the pull type options can be used to enable pull-up or pull-down resistors since the Pmod does not have pullup resistors. The I/O standard for all ports should be set to LVCMOS33, meaning the pin is now 3.3V tolerant. After all that, the tedious part is left which is to map the ports to a designated FPGA Pmod pin. The Pmod ports are GPIO’s that can be used as either input or output or bidirectional ports. Note, constraints are used to map the Zynq PL I/O pins to the FPGA Pmod pins.

Table 1. Constraints for Pmod AD2

|  |  |  |
| --- | --- | --- |
| Block Diagram Port: Pmod\_out\_0 | Package pin | Pmod pin |
| Pmod\_out\_0\_pin1 | V8 | JB1 |
| Pmod\_out\_0\_pin2 | W8 | JB2 |
| Pmod\_out\_0\_pin3 | U7 | JB3 |
| Pmod\_out\_0\_pin4 | V7 | JB4 |
| Pmod\_out\_0\_pin7 | Y7 | JB7 |
| Pmod\_out\_0\_pin8 | Y6 | JB8 |
| Pmod\_out\_0\_pin9 | V6 | JB9 |
| Pmod\_out\_0\_pin10 | W6 | JB10 |

Table 2. Constraints for Pmod TC1

|  |  |  |
| --- | --- | --- |
| Block Diagram Port: | Package pin | Pmod pin |
| CS | V15 | JC1 |
| MOSI | W16 | JC2 |
| MISO | T11 | JC3 |
| SCK | T10 | JC4 |

Table 3. Constraints for Pmod OLED

|  |  |  |
| --- | --- | --- |
| Block Diagram Port: Pmod\_out\_1 | Package pin | Pmod pin |
| Pmod\_out\_0\_pin1 | T14 | JD1 |
| Pmod\_out\_0\_pin2 | T15 | JD2 |
| Pmod\_out\_0\_pin3 | P14 | JD3 |
| Pmod\_out\_0\_pin4 | R14 | JD4 |
| Pmod\_out\_0\_pin7 | U14 | JD7 |
| Pmod\_out\_0\_pin8 | U15 | JD8 |
| Pmod\_out\_0\_pin9 | V17 | JD9 |
| Pmod\_out\_0\_pin10 | V18 | JD10 |

# Verification



# MATLAB

MATLAB was used to read the text file and plot the data but shortly after, it was also used to verify the algorithm by using a histogram. The very first samples plotted, had peaks saturating or flat near the peak pulse. The typical ECG signal does not have a flat peak or pulse but nonetheless, by zooming in, it is very apparent that the signal indeed is an ECG signal with flat peaks. The flat peaks could be happening for a multitude of reasons such as not probing correctly, light is interfering with photosensor, cheap sensor, algorithm has bugs or other factors. Despite the flat peaks, is the Vitis algorithm calculating a reasonable BPM using the clipped data?

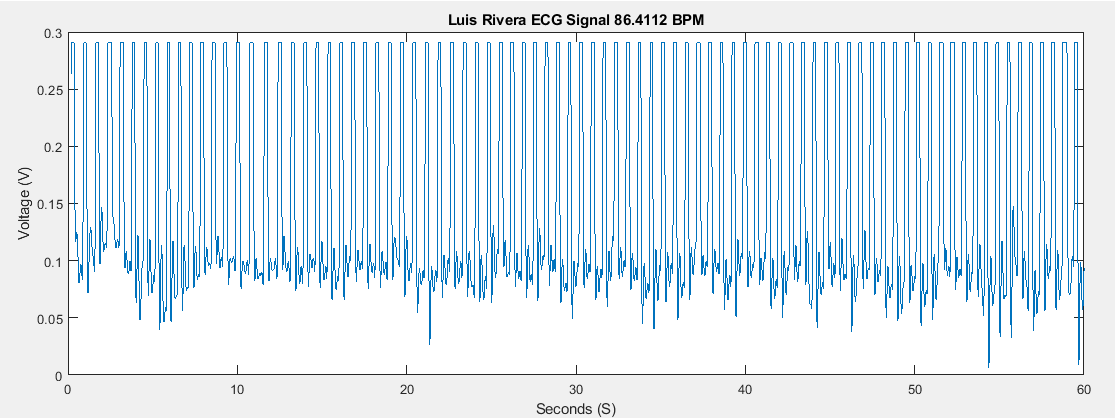


Figure 26. ADC Data from Pulse Sensor Example 1

MATLAB has useful built-in functions like findpeaks() that can simplify calculating BPM. The findpeaks() function finds the local peaks in a data vector and can return the index of each peak. Since the x-axis contains the time information in seconds, any two neighboring peaks are sufficient to calculate a BPM. By using the diff() function, all elements in a data vector can be subtracted (x(1)-x(0)). Using the indexes of each peak and dividing by the difference of all elements, results in a BPM calculation. The average can be taken of all BPM calculation to determine the average BPM in that 1-minute duration. Comparing the results from MATLAB vs Vitis BPM, they are nearly identical every run and this is even with clipped data.

Can we really trust the MATLAB algorithm and Vitis algorithm? Another set of verification is an empirical observation by using the plotted ADC data. If we look at the first 10 seconds and measure the pulses in this 10 second interval, if there are 15 pulses, it can easily be determined that the expected pulses in a minute, would be (15 pulses per 10 seconds) x (6) is equivalent to 1 minute of pulses or in other words, 90 BPM. By using this empirical method, the Vitis and MATLAB algorithm are spot on.

Last method of verification was using a histogram plot that was a nice visualization of the data and helped determine a reasonable threshold for filtering false peaks or bad data. The histogram reveals all the voltages and their corresponding occurrences, which helped determining the sweet spot for calculating BPM. From the figure below, the pink represents the mean, green represents sweet spot and red is the max value. By using the histogram, it is very apparent that to calculate a reasonable BPM, the sweet spot must be within the mean and max (mean < sweet spot < max).

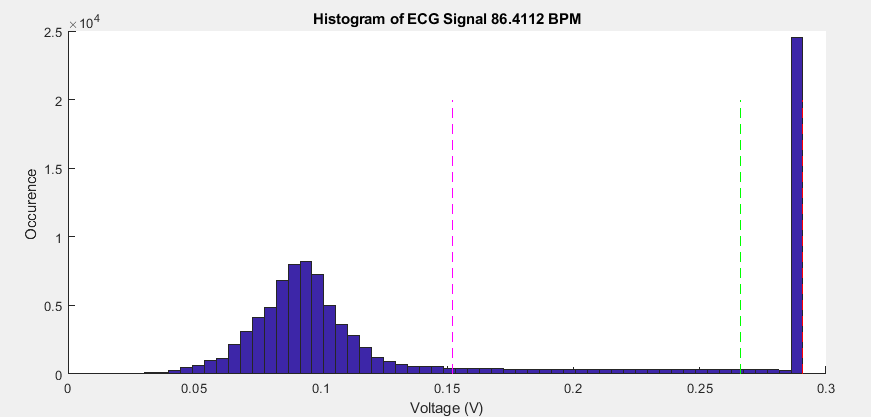


Figure 27. Histogram 1

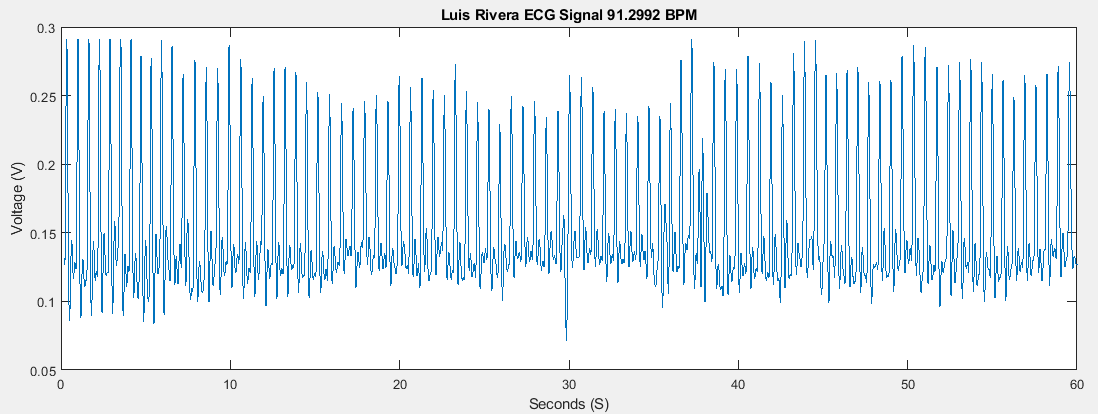


Figure 28. ADC Data from Pulse Sensor Example 2

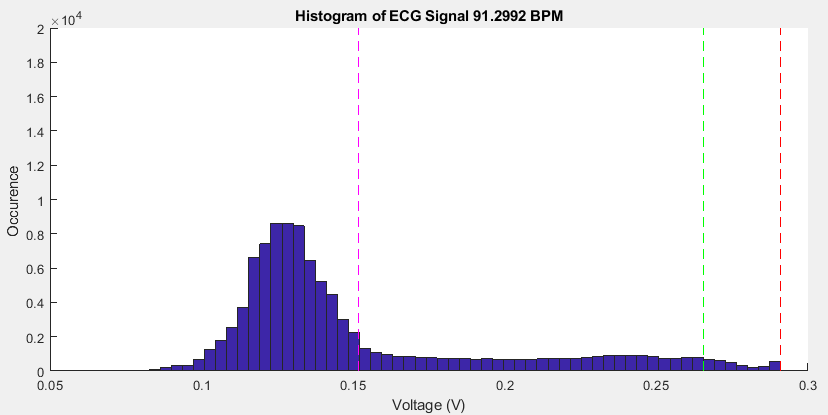


Figure 29. Histogram 2

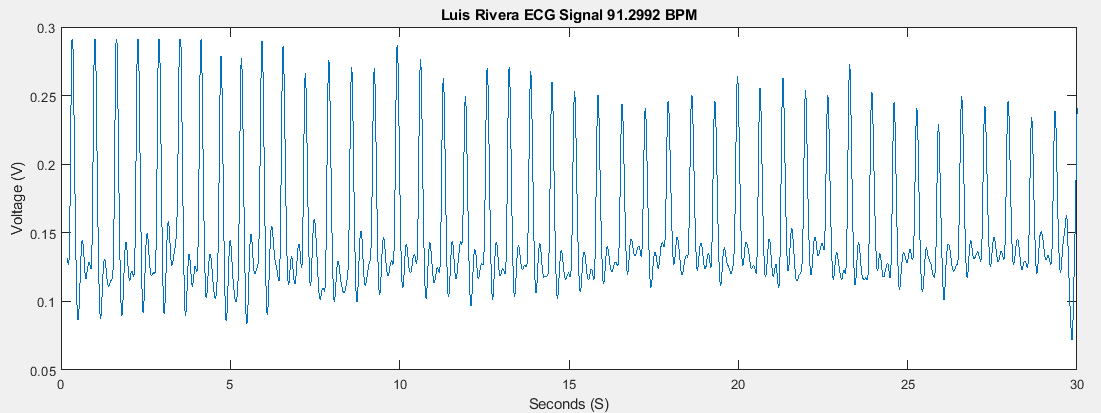


Figure 30. First 30 seconds of ADC Data from Pulse Sensor Example 1

The power spectrum displayed in the figure below plots the signal power against the frequencies. The spectrum can be calculated by performing the DFT or FFT of the analog signal and then squaring the signal over time for the magnitude. The use of this graph helps correlate the noise associated with the signal and decide the appropriate sampling rate. Note that FFT is very similar to DFT and usually is a preferred operation since it requires less resources hence the computation is less intensive and more efficient.

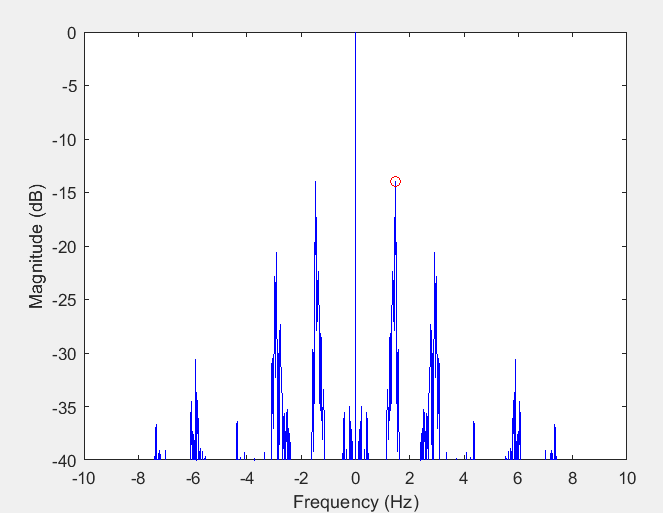


Figure 31. Power Spectrum of ADC Data from Pulse Sensor Example 1

# Testing Heart Rate

Different testing approaches were done on each sensor and then tested as an integrated system. Initially the algorithm was tested for reasonable measurements and after thoroughly testing the algorithms, reference sensors were used to verify each measurement. MATLAB was used to verify the heart rate and GSR algorithms.

Initially the BPM algorithm was fixed for collecting 1 minute of data and after observing the BPM calculations and not having a clue whether the sensor is outputting correctly, the data was printed in such a manner that it can be saved onto a text file so the raw ECG signal can be plotted on MATLAB. This allows us to probe the signal and observe what the signal is doing.

# Testing GSR

MATLAB was used to calculate the resistance from a given voltage range using a table from previous research. For verifying GSR results, the voltage from the GSR sensor is mapped to a interpolated resistance and then conductance can easily be calculated by applying the inverse. If the resistance from the output voltage is within range of our table, we can assume it is working. For further verification, an external reference sensor was used to compare measurements and results are within 5-15% range.

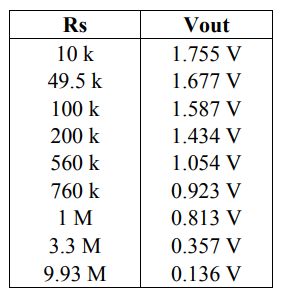


Figure 32. GSR Voltage and Resistance Table

# Testing Temperature

An external reference sensor such as a non-contact thermometer was used to verify the temperature measurements. The Pmod TMP3 and TC1 were both tested for a few scenarios such as warm, ambient, and cold environments. For testing warm, a heated towel was wrapped around the sensor, while testing for cold required to small Ziploc ice bags. The ambient environment is simply the room temperature. In all scenarios, the temperature adjusted appropriately, and output was nearly identical to thermometer measurement.

# Testing OLED Display

Testing the OLED is the simplest because once you can write to the display, that confirms functionality.



Figure 33. Image of project in action



# Results

Each sensor individually was tested before integration and results for each sensor are reasonable and within 1-5% range of the external reference sensor. If all sensors work individually, then integrating should not give a problem. All sensors even after integration still had accurate measurement but it was noticed that the BPM was a bit more inconsistent than before. During integrating & testing, the problem of the Pmod AD2 reading sequentially from two channels, explained why the integration portion was having an inconsistent BPM. One method to resolve this was increasing the UART baud rate which in turn made the BPM calculation more consistent like before. The results for the project demonstrate that the system works and more importantly, a wearable device can be designed.

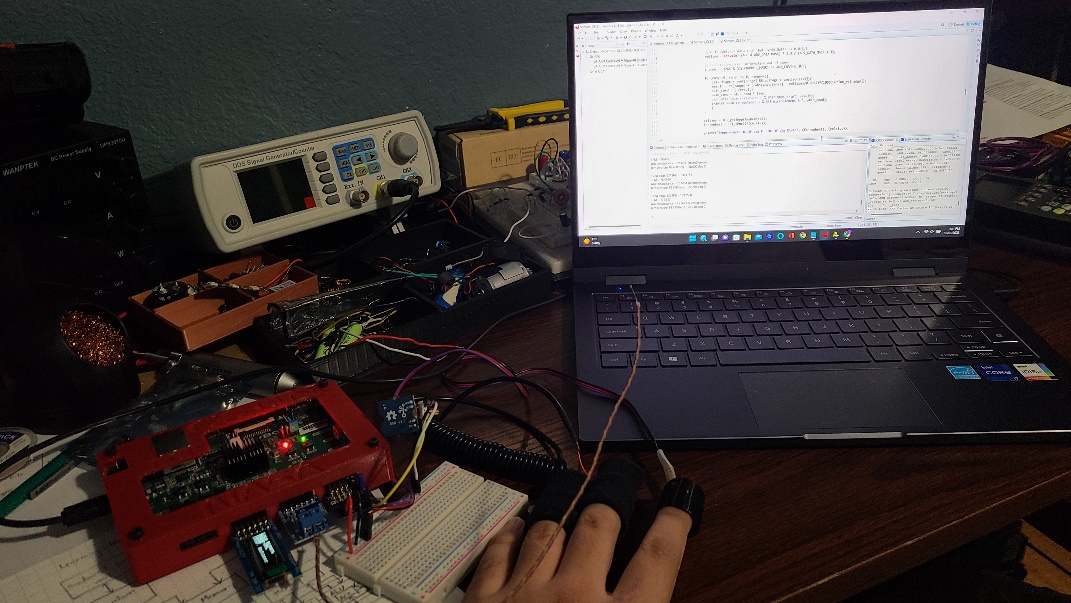


Figure 34. Live Demo with Sensors

# Earlier Prototype

A proof-of-concept implementation was created for integrating a stress detection RTL module to the block design and further understand how to create custom IP block logic and interface with other PS and PL peripherals. Creating custom IP’s and interfacing with other PS and PL peripherals requires an understanding of handshakes, and communication protocols. IO ports on a custom IP block can’t be connected unless they are AXI compatible, using the same communication protocol or some form of handshake is utilized.

If using handshakes, AXI stream or communication protocols, this needs to be implemented in the source code. The stress detection module was not compatible therefore GPIOs were used to drive and poll the signals. For high-speed processing, GPIO method should be avoided since GPIOs are slow and not a standard method for streaming lots of data.

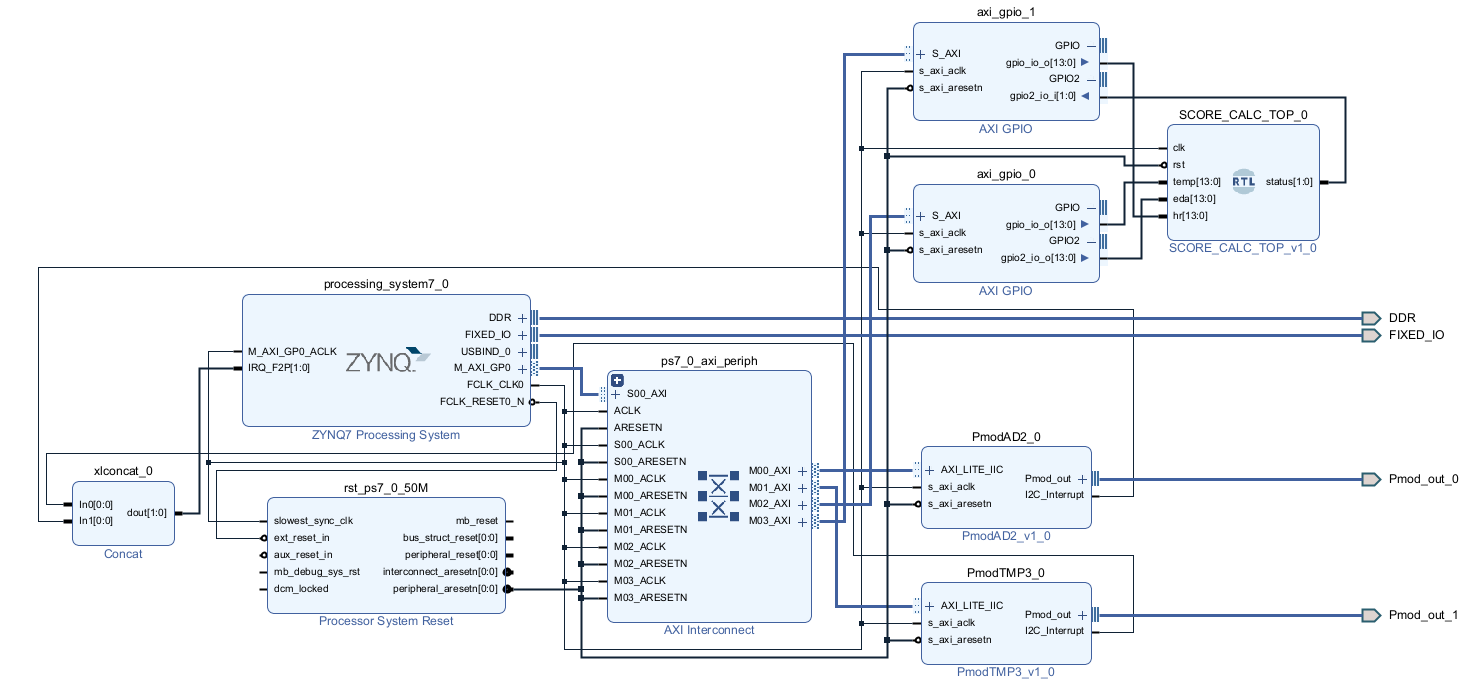


Figure 35. Sensors with stress detection module

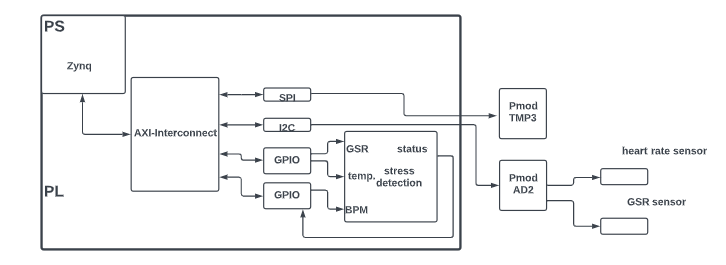


Figure 36. Simplified block design

# Future Work



# Bluetooth

Adding an OLED display to the project, revealed another set of ideas for developing a IoT system. Just like the measurements can be sent from the FPGA to an OLED display, similarly Bluetooth can be added to send the data to a nearby Bluetooth device. Bluetooth devices tend to be low-power IoT devices since they require less energy to transmit a signal, however transmitting and receiving devices need to be within proximity.

# WIFI

Another popular choice for creating an IoT device is WIFI as it can offer similar capabilities like Bluetooth. Unlike Bluetooth, WIFI can send data to a device without being nearby if there is a WIFI access point. WIFI is incredibly much faster than Bluetooth, better security, longer usable range but has a higher power consumption.

# BPM Algorithm

A few different optimizations can be implemented for BPM algorithm on Vitis such as using a hardware filter instead of the moving average filter software implementation and circular buffer for holding the data in the window. If a hardware filter is implemented, no circular buffer will be needed and the Vitis code will be reduced allowing for faster execution time.



# Hardware Filter

For the BPM algorithm, the digital signal processing was done on the software side. The output from the heart rate sensor was not detailed enough to calculate an effective BPM, therefore filtering was needed to eliminate the discrepancies. For example, using an oscilloscope to probe my ECG signal from the heart rate sensor entails that my pulse is roughly 1-2Hz. My ECG pulse corresponds to a BPM ranging from 60-120 but more importantly, what determines the pulse or peak is happening faster than 2Hz. The pulse is active for a duration of roughly 300ms that translates to a frequency of 3.3Hz. For this signal, the high frequency content is the pulse therefore to clean the signal, a high pass filter can be used to block or attenuate low frequencies.

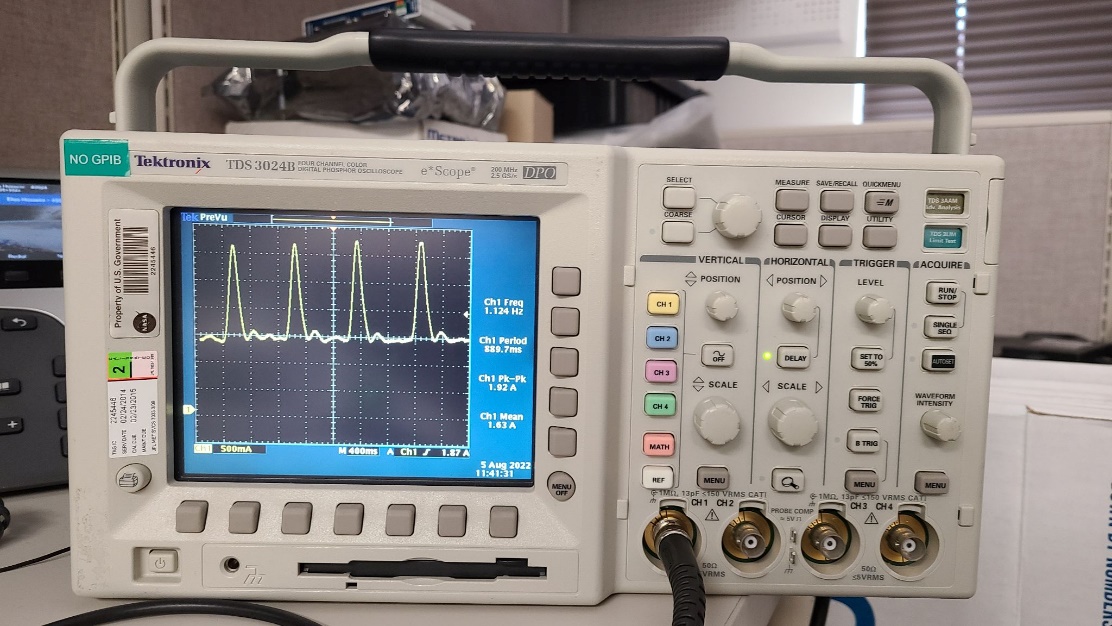


Figure 37. Oscilloscope Reading of Pulse Sensor

The software implementation filtered the signal by applying a moving average filter, which behaves like a low pass filter. By knowing the moving average of the ECG signal, you can filter the signal to not include any pulses less than the moving average. This implementation can be improved by using FPGA resources such as digital filters. The FPGA has DSP slices that allow high speed arithmetic or filtering. By understanding the ECG pulse and frequency of interest, an FIR or IIR filter can be deployed to attenuate the signal.

We are interested in in the high frequency content; therefore, a high pass filter can be used but the same functionality can be achieved using a low pass and simple arithmetic operation. For example, if the low pass filter is used on the ECG signal the filtered output can be subtracted from the original ECG signal and the high frequency content will be remaining.

# Circular Buffer

After experimenting, it was noticed that the moving average filter could be optimized by using a circular buffer. The filter needed an array or window to store the raw signal from the Pmod AD2 to calculate a continuous moving average. To calculate a new moving average, the array holding the current elements needs to be shifted by one element every calculation. Since only one data value changes per moving average calculation, the window should be shifted by 1 to create a new space for the incoming signal. This process can be very slow and in fact, operates at O(n) since n operations are needed to shift all elements. A circular buffer behaves like a FIFO by releasing oldest elements first and can a be single or fixed size. This can enhance the performance of the algorithm as each moving average calculation won’t require the data be shuffled around but instead just update head or tail pointer. If data is added, the head pointer advances while the tail pointer advances when data is consumed. If you reach the end of the buffer, the pointers wrap around.

# Machine Learning

A stress detection template on GitHub with suitable parameters was used to create a custom IP block with stress detection logic. This allowed for easy integration to the system, but improvements can be made. The detection module can be improved to a machine learning algorithm by adding memory and a feedback loop to learn from previous data.

# Prototype Glove

One of the goals of the project was to develop a wearable device using the sensors, like an Apple smartwatch. Not enough time was spent designing a wearable device, but a Velcro glove could be a simple protype, nothing flashy. The sensors are contact type, which allows for them to be embedded or sewed onto the Velcro.

# Conclusion

By analyzing the results from the project, the Digilent Zybo Z7-20 development board demonstrates to be a suitable option for wearable devices. The development board can be connected to multiple sensors and perform signal processing for accurate measurements of biometric data. While the size may not be ideal for commercial wearable devices, this approach provides insight of SoC design capabilities. Based on these conclusions, engineers should consider a modular development board, research other sensors that can be used, integrate stress analysis, and optimize algorithm among others.

The project was a challenge but served its purpose in learning fundamentals of enabling peripherals via PS or PL, configuring IP’s using Vivado or writing to specific registers using Vitis, creating custom IP by using RTL code as a block, using global timer to benchmark execution time of code, using Pmod ‘s, implementing communication protocols, integrating a system that can read from multiple sensors, reading datasheets for details like timing diagrams that explain how to read and write to module or registers that control peripherals and the bits needed to enable features for that peripheral, and many more but the general idea is that Vivado and FPGAs is now part of my toolbox.

# Bibliography

1. "What is an FPGA? Field Programmable Gate Array - Xilinx." <https://www.xilinx.com/products/silicon-devices/fpga/what-is-an-fpga.html>.
2. "FPGA – Configurable Logic Block – Digilent Blog." 16 Jun. 2021, <https://digilent.com/blog/fpga-configurable-logic-block/>.
3. "Configurable Logic Block (CLB) - HardwareBee Semipedia." 04 Aug. 2021, <https://hardwarebee.com/WikiBee/configurable-logic-block-clb/>.
4. "Design Hubs - Xilinx." <https://www.xilinx.com/support/documentation-navigation/design-hubs.html>.

# Appendix A: Vitis Source Code

**#include** <stdio.h>

**#include** <stdint.h>

**#include** "PmodAD2.h"

**#include** "PmodTC1.h"

**#include** "PmodOLED.h"

**#include** "sleep.h"

**#include** "platform.h"

**#include** "xparameters.h"

**#include** "xil\_io.h"

**#include** "xil\_printf.h"

**#define** global\_timer\_clk 333000000 //global timer clock frequency 333MHz

**#define** global\_time\_base 0xF8F00200

**#define** size 10000000 //increase size for continous data or dont use array

**#define** window\_size 700 //window size of 50 elements

//SCL low and high registers for changing I2C speed

**#define** XIIC\_THIGH\_REG\_OFFSET 0x13C

**#define** XIIC\_TLOW\_REG\_OFFSET 0x140

//Pulse Sensor

PmodAD2 myDevice;

u8 channel;

**double** voltage;

u16 volt;

**static** u16 conv[size] = {'\0'};

**static** u16 ADC = 0;

//Temperature Sensor

PmodTC1 myDevice2;

//Pmod OLED Display

PmodOLED myDevice3;

// To change between PmodOLED and OnBoardOLED is to change Orientation

**const** u8 orientation = 0x0; // Set up for Normal PmodOLED(false) vs normal

//Onboard OLED(true)

**const** u8 invert = 0x0; // true = whitebackground/black letters

// false = black background /white letters

//Timer function

**static** uint64\_t timestamp[size] = {'\0'};

**static** **double** tmp[size] = {'\0'};

**static** **double** dummy[size] = {'\0'};

**static** **double** dummy2[size] = {'\0'};

**static** **double** t\_diff[size] = {'\0'};

//Moving Average

uint32\_t valid\_index[size] = {'\0'}; //window size

uint32\_t array[window\_size] = {'\0'}; //window size

uint32\_t mov\_avg\_array[size] = {'\0'}; //array for storing moving avg

uint32\_t sum = 0;

uint32\_t mov\_avg = 0;

uint32\_t cnt = 0;

**int** x = 0;

**int** i =0;

**double** sum\_time = 0;

**double** avg\_BPM = 0;

**double** avg\_BPM2 = 0;

**double** BPM = 0;

**int** y = 0;

**int** j = 0;

**int** range = 0;

//change some variables to local

//DDR memory base address 0x200

//#define ddr\_base 0x00000200

uint64\_t **time\_elapsed**(); //allows function to be used globally

**int** **main**()

{

init\_platform();

//GSR parameters

**double** vout[] = {1.755, 1.677, 1.587, 1.434, 1.054, .923, .813, .357, .135};

**double** rs [] = {10000, 49500, 100000, 200000, 560000, 760000, 1000000, 3300000, 9930000};

**double** inc\_rs[] = {0.1632, 0.1808, 0.2106, 0.3053, 0.4920, 0.7031, 1.6255, 9.6244};

**double** result = 0;

**double** skin\_cond = 0;

//temperature

**double** celsius, fahrenheit;

print("Hello World\n\r");

print("Successfully ran Hello World application");

//initialize PMOD AD2

AD2\_begin(&myDevice, XPAR\_PMODAD2\_0\_AXI\_LITE\_IIC\_BASEADDR, AD2\_IIC\_ADDR);

//Change I2C operating frequency

// Old SCL parameters

**int** t\_high = XIic\_ReadReg(XPAR\_PMODAD2\_0\_AXI\_LITE\_IIC\_BASEADDR, XIIC\_THIGH\_REG\_OFFSET);

**int** t\_low = XIic\_ReadReg(XPAR\_PMODAD2\_0\_AXI\_LITE\_IIC\_BASEADDR, XIIC\_TLOW\_REG\_OFFSET);

**printf**("T\_high = %d, T\_low = %d\n", t\_high, t\_low);

// New SCL parameters

t\_high = 63;

t\_low = 63;

XIic\_WriteReg(XPAR\_PMODAD2\_0\_AXI\_LITE\_IIC\_BASEADDR, XIIC\_THIGH\_REG\_OFFSET, t\_high);

XIic\_WriteReg(XPAR\_PMODAD2\_0\_AXI\_LITE\_IIC\_BASEADDR, XIIC\_TLOW\_REG\_OFFSET, t\_low);

//initialize Pmod TC1

TC1\_begin(&myDevice2, XPAR\_PMODTC1\_0\_AXI\_LITE\_SPI\_BASEADDR);

//initialize Pmod OLED

OLED\_Begin(&myDevice3, XPAR\_PMODOLED\_0\_AXI\_LITE\_GPIO\_BASEADDR,XPAR\_PMODOLED\_0\_AXI\_LITE\_SPI\_BASEADDR, orientation, invert);

//AD2\_WriteConfig(&myDevice, AD2\_CONFIG\_CH0);

//Turn on channel 0 & 1, pin V1 & V2

AD2\_WriteConfig(&myDevice, AD2\_CONFIG\_CH0 | AD2\_CONFIG\_CH1);

uint64\_t start\_time = time\_elapsed(); //get start time

**while**(1) { //forever loop

AD2\_ReadConv(&myDevice, &ADC);

conv[i] = ADC;

timestamp[i]= time\_elapsed(); //get current time

// Pull channel read information out of conv

channel = (ADC & AD2\_CHANNEL\_MASK) >> AD2\_CHANNEL\_BIT;

//do only if channel 0 = Pin V1

**if** (channel == 0) {

// Keep 12-bit result, masking or bit select

conv[i] &= 0xFFF;

//calculate first moving average

sum += conv[i];

insert(conv[i], cnt);

cnt++;

**if**(cnt >= window\_size) {//window size of 700 and collecting 20,000 samples (5.98\*2 = 12 seconds)

mov\_avg = (sum - array[window\_size-1])/(window\_size-1);

mov\_avg\_array[i] = mov\_avg;

sum -= array[0];

erase(1); //erase(0)

//moving average scaler 1.35-1.75

//if(conv[i] > (mov\_avg\*1.45) && conv[i-1] < (mov\_avg\*1.45)){ //detecting rising edges

**if**(conv[i] > (mov\_avg\*1.75) && conv[i-1] < (mov\_avg\*1.75)){

tmp[i] = (**double**) ((timestamp[i]-timestamp[0])/333000000.0); //returns time measurement in seconds

//timestamp[i]-timestamp[0] never exceeds a 52-bit number (156 days) 2^52/(333e6\*86400) , 86400 = 24\*60\*60

valid\_index[j] = i; //store valid timestamp index

**if**((tmp[(valid\_index[j])] - tmp[(valid\_index[j-1])] > .4) && j >= 1) {

BPM = 60/(tmp[(valid\_index[j])] - tmp[(valid\_index[j-1])]);

//neglect bad BPM

**if** (BPM > 40 && BPM < 160) {

**printf**("rising edge[%d]= %.04lf \n", i, tmp[i]-tmp[i-1]);

**printf**("BPM = %.04lf \n", i, BPM);

//GSR Reading, channel 1 = Pin V2

AD2\_ReadConv(&myDevice, &ADC);

// Keep 12-bit result, masking or bit select

ADC &= 0xFFF;

//Scale captured data such that 0x000:0xFFF => 0.0:3.3

voltage = (**double**) (ADC & AD2\_DATA\_MASK) \* 3.3 / (AD2\_DATA\_MASK + 1);

// Pull channel read information out of conv

channel = (ADC & AD2\_CHANNEL\_MASK) >> AD2\_CHANNEL\_BIT;

**for**(range=0; range <= 8; range++){

**if**(voltage < vout[range] && voltage > vout[range+1]){

result = rs[range] + (fabs(vout[range] - voltage)/0.0032)\*(10000.0\*inc\_rs[range]);

skin\_cond = (1/result);

skin\_cond = skin\_cond \* 10e6;

//printf("skin resistance = %.04lf Ohms \r\n", result);

**printf**("skin conductance = %.04f microSiemens \n", skin\_cond);

}

}

celsius = TC1\_getTemp(&myDevice2);

fahrenheit = TC1\_tempC2F(celsius);

**printf**("Temperature: %0.3f deg F %0.3f deg C\n\r", (fahrenheit), (celsius));

**int** irow, ib, i;

u8 \*pat;

**char** c;

//while (1) {

// xil\_printf("entering loop\r\n");

// Choosing Fill pattern 0

pat = OLED\_GetStdPattern(0);

OLED\_SetFillPattern(&myDevice3, pat);

// Turn automatic updating off

OLED\_SetCharUpdate(&myDevice3, 0);

// Draw a rectangle over writing then slide the rectangle down slowly

// displaying all writing

**for** (irow = 0; irow < OledRowMax; irow++) {

OLED\_ClearBuffer(&myDevice3);

OLED\_SetCursor(&myDevice3, 0, 0);

OLED\_PutString(&myDevice3, "Luis Rivera\r\n");

OLED\_SetCursor(&myDevice3, 0, 1);

OLED\_PutString(&myDevice3, "NASA-JPL");

OLED\_SetCursor(&myDevice3, 0, 2);

OLED\_PutString(&myDevice3, "CSUN");

OLED\_SetCursor(&myDevice3, 0, 3);

OLED\_PutString(&myDevice3, "Comp. Engineer");

OLED\_MoveTo(&myDevice3, 0, irow);

OLED\_FillRect(&myDevice3, 127, 31);

OLED\_MoveTo(&myDevice3, 0, irow);

OLED\_LineTo(&myDevice3, 127, irow);

OLED\_Update(&myDevice3);

}

}

}

j++;

}

}

i++;

}

}

cleanup\_platform();

**return** 0;

}

**void** **insert** (uint32\_t val, uint32\_t cond) {

uint16\_t empty = {'\0'};

//when array is completely empty

**if**(cond<window\_size){//only search for empty elements when window size is being filled

**for**(**int** i = 0; i < window\_size; i++){

**if**(array[i] == empty) {

array[i] = val;//store adc value

**return**;

}

}

}

**else**{

array[window\_size-1] = val;//if window is filled (window size - 1), store adc value at most recent sample

}

**return**;

}

**void** **erase**(){

**for**(**int** i = 0; i < window\_size; i++){

**if**(i < window\_size) {

array[i] = array[i+1]; //shift contents by 1

}

}

array[window\_size-1] = 0; //empty most recent sample

**return**;

}

uint64\_t **time\_elapsed**() {

/\* local variable declaration \*/

//1. Read the upper 32-bit timer counter register.

uint32\_t upper\_bits = Xil\_In32(global\_time\_base + 4);

//2. Read the lower 32-bit timer counter register.

uint32\_t lower\_bits = Xil\_In32(global\_time\_base);

//3. Read the upper 32-bit timer counter register again. If the value is different to the 32-bit upper value

uint32\_t tmp\_bits = Xil\_In32(global\_time\_base + 4);

**if**(tmp\_bits == upper\_bits) { //global timer value is valid

uint64\_t i = (uint64\_t) upper\_bits << 32 | lower\_bits; //returns 64 bit result

//printf("%" PRIu64 "\n", i);

**return** i;

}

//read previously, go to the previous step. Otherwise the 64-bit timer counter value is correct.

}

# Appendix B: Vivado Block Design Constraints

#Pmod JB --> Pmod AD2

set\_property PACKAGE\_PIN V8 [get\_ports Pmod\_out\_0\_pin1\_io]

set\_property PACKAGE\_PIN W8 [get\_ports Pmod\_out\_0\_pin2\_io]

set\_property PACKAGE\_PIN U7 [get\_ports Pmod\_out\_0\_pin3\_io]

set\_property PACKAGE\_PIN V7 [get\_ports Pmod\_out\_0\_pin4\_io]

set\_property PACKAGE\_PIN Y7 [get\_ports Pmod\_out\_0\_pin7\_io]

set\_property PACKAGE\_PIN Y6 [get\_ports Pmod\_out\_0\_pin8\_io]

set\_property PACKAGE\_PIN V6 [get\_ports Pmod\_out\_0\_pin9\_io]

set\_property PACKAGE\_PIN W6 [get\_ports Pmod\_out\_0\_pin10\_io]

set\_property IOSTANDARD LVCMOS33 [get\_ports Pmod\_out\_0\_pin1\_io]

set\_property IOSTANDARD LVCMOS33 [get\_ports Pmod\_out\_0\_pin2\_io]

set\_property IOSTANDARD LVCMOS33 [get\_ports Pmod\_out\_0\_pin3\_io]

set\_property IOSTANDARD LVCMOS33 [get\_ports Pmod\_out\_0\_pin4\_io]

set\_property IOSTANDARD LVCMOS33 [get\_ports Pmod\_out\_0\_pin7\_io]

set\_property IOSTANDARD LVCMOS33 [get\_ports Pmod\_out\_0\_pin8\_io]

set\_property IOSTANDARD LVCMOS33 [get\_ports Pmod\_out\_0\_pin9\_io]

set\_property IOSTANDARD LVCMOS33 [get\_ports Pmod\_out\_0\_pin10\_io]

set\_property SLEW SLOW [get\_ports Pmod\_out\_0\_pin3\_io]

set\_property PULLUP true [get\_ports Pmod\_out\_0\_pin3\_io]

set\_property PULLUP true [get\_ports Pmod\_out\_0\_pin4\_io]

# Pmod JC --> Pmod TC1

set\_property IOSTANDARD LVCMOS33 [get\_ports CS]

set\_property IOSTANDARD LVCMOS33 [get\_ports MISO]

set\_property IOSTANDARD LVCMOS33 [get\_ports MOSI]

set\_property IOSTANDARD LVCMOS33 [get\_ports SCK]

set\_property PACKAGE\_PIN V15 [get\_ports CS]

set\_property PACKAGE\_PIN W15 [get\_ports MOSI]

set\_property PACKAGE\_PIN T11 [get\_ports MISO]

set\_property PACKAGE\_PIN T10 [get\_ports SCK]

#Pmod JD --> Pmod OLED

set\_property PACKAGE\_PIN T14 [get\_ports Pmod\_out\_1\_pin1\_io]

set\_property PACKAGE\_PIN T15 [get\_ports Pmod\_out\_1\_pin2\_io]

set\_property PACKAGE\_PIN P14 [get\_ports Pmod\_out\_1\_pin3\_io]

set\_property PACKAGE\_PIN R14 [get\_ports Pmod\_out\_1\_pin4\_io]

set\_property PACKAGE\_PIN U14 [get\_ports Pmod\_out\_1\_pin7\_io]

set\_property PACKAGE\_PIN U15 [get\_ports Pmod\_out\_1\_pin8\_io]

set\_property PACKAGE\_PIN V17 [get\_ports Pmod\_out\_1\_pin9\_io]

set\_property PACKAGE\_PIN V18 [get\_ports Pmod\_out\_1\_pin10\_io]

set\_property IOSTANDARD LVCMOS33 [get\_ports Pmod\_out\_1\_pin1\_io]

set\_property IOSTANDARD LVCMOS33 [get\_ports Pmod\_out\_1\_pin2\_io]

set\_property IOSTANDARD LVCMOS33 [get\_ports Pmod\_out\_1\_pin3\_io]

set\_property IOSTANDARD LVCMOS33 [get\_ports Pmod\_out\_1\_pin4\_io]

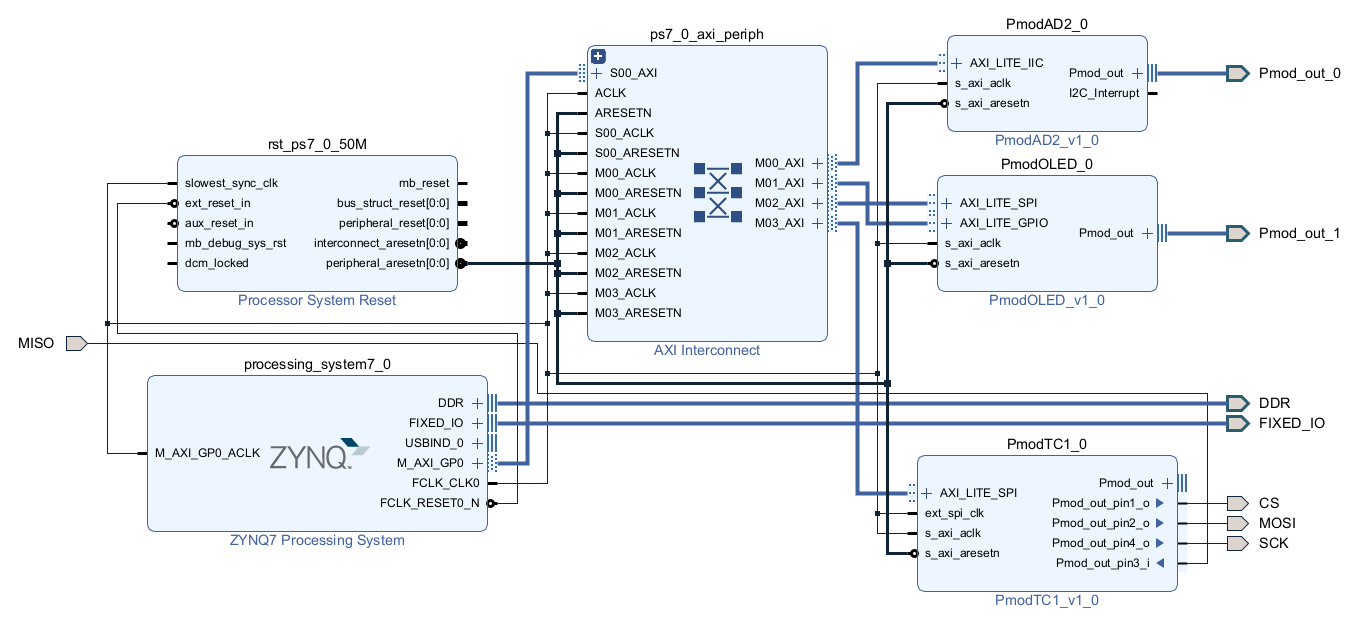
set\_property IOSTANDARD LVCMOS33 [get\_ports Pmod\_out\_1\_pin7\_io]

set\_property IOSTANDARD LVCMOS33 [get\_ports Pmod\_out\_1\_pin8\_io]

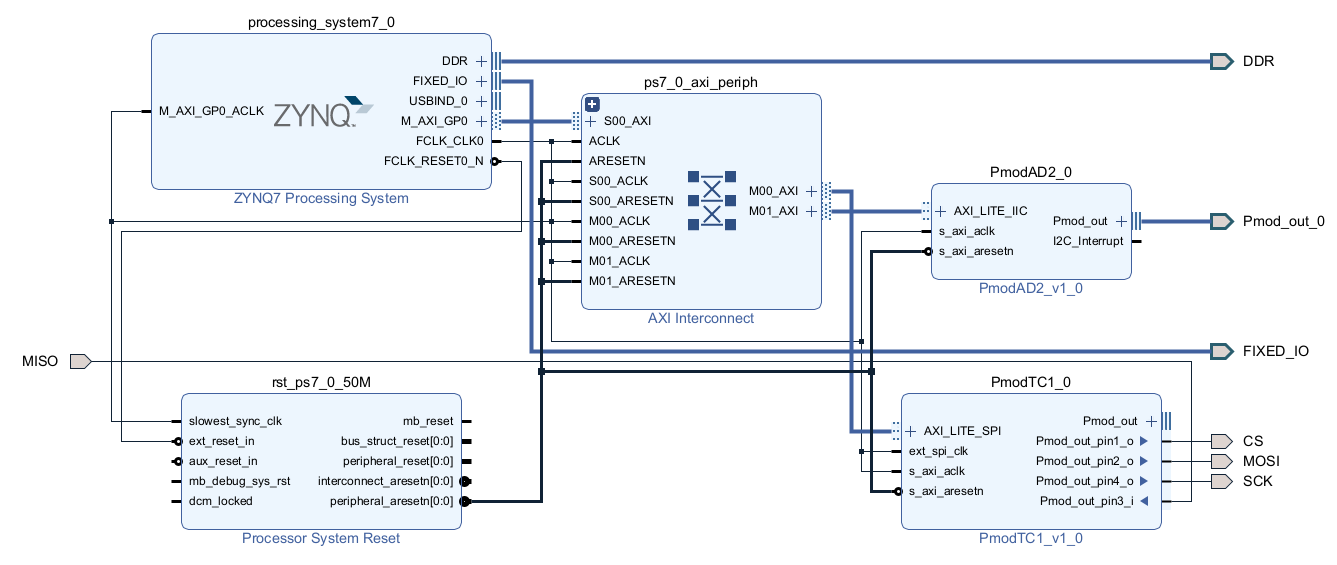
set\_property IOSTANDARD LVCMOS33 [get\_ports Pmod\_out\_1\_pin9\_io]

set\_property IOSTANDARD LVCMOS33 [get\_ports Pmod\_out\_1\_pin10\_io]

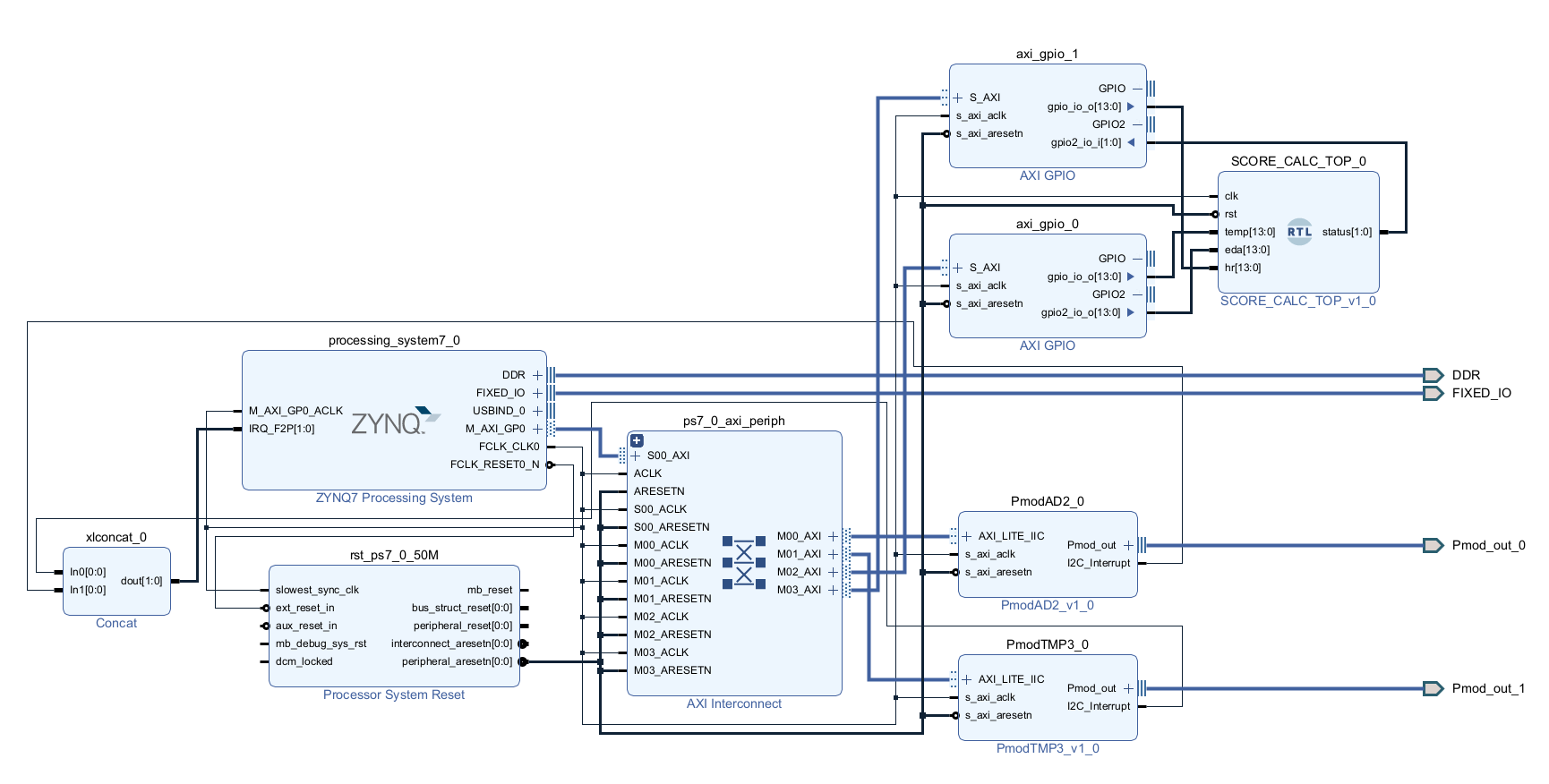
# Appendix C: Sensors w/ OLED Display Vivado Block Design

****

# Appendix D: Sensors Vivado Block Design



# Appendix E: Sensors w/ stress detection Vivado Block Design

****

# Appendix F: GitHub Source Code

[GitHub Source Code](https://github.com/CSUN-Masters-Project/Zybo-Z720)