Department of Computer Science
Technical University of Cluj-Napoca

Sensor data processing using AXI4-Stream protocol on Basys3

SCS project activity

Name: Katalin Simon
Group: Computer Science, 30434
Email: skata.univ@gmail.com

Teaching Assistant: Anca Hangan
Anca.Hangan@cs.utcluj.ro

Contents

Chapter 1

Introduction

1.1 Project Proposal

1.1.1 Motivation

In modern times **health is of essence**. Usually, people tend to say things such as "I regret not traveling more", "I regret not telling him/her that I love him her/more often" and so on. But one sentence that always catches my ears is: "I wish I had taken better care of myself".

Good health is the foundation upon our every aspiration rests. It determines how we live, how we work, and how we experience the world. Yet, in an age of constant motion and digital distraction, we tend to notice our well-being only when something goes wrong.

Modern medicine is trying to shift this tendency. With the growing presence of data and technology, we now have the means to observe ourselves continuously, to understand patterns before they become problems, and to take informed action. Data has become the language of health, and learning to interpret it might be one of the most valuable pursuits of our time.

On the other hand, prevention through fitness became one of the most beloved method for preserving and maintaining good physical health. Through regular exercise, balanced nutrition, and conscious daily choices, many health issues can be avoided before they even emerge. Fitness tracking and real-time monitoring allow individuals to connect effort with outcome — transforming raw numbers into motivation, and awareness into lasting habit. The fusion of personal discipline and technological insight opens the path toward a future where maintaining health is not reactive, but proactive.

Most existing wearable devices rely heavily on microcontroller-based processing, which is limited in speed and scalability. Hardware-level data processing using FPGAs can provide real-time responsiveness and higher throughput, making it suitable for applications that demand fast and reliable physiological signal analysis.

1.1.2 Project Objective

The objective of this project is to design and implement a real-time heart rate signal processing system using an FPGA and an Arduino board acting as an analogue-to-digital converter (ADC) bridge. The system will receive alanlogue pulse data from a heart rate sensor, convert it to a digital signal via Arduino, and transmit it to the FPGA through a serial UART link.

The FPGA will process the incoming data stream using the AXI4-Stream protocol, per-

forming filtering, peak detection, and heart rate computation. The computed result will be displayed in real time on the FPGA's 7-segment display.

1.1.3 System Overview

Hardware Components

- Heart Rate Sensor: Provides an alanlogue voltage corresponding to pulse intensity. (MAX30100);
- Arduino Uno/Mega: Acting as ADC bridge, transmitting data to FPGA via UART;
- FPGA Board (Basys 3 – Xilinx Artix-7): Processes serial data stream and outputs the calculated heart rate;
- Wires and connections

Data Flow

- The heart rate sensor outputs an alanlogue voltage waveform.
- The Arduino samples and digitizes this signal.
- The digitized samples are sent serially to the FPGA.
- The FPGA implements real-time filtering, peak detection, and BPM calculation using AXI4-Stream modules.
- The result is displayed on the 7-segment display of the Basys board.

1.1.4 Functional and Non-functional Requirements

Functional requirements

- **Data Acquisition** The system shall acquire alanlogue heart rate data from the sensor through Arduino's ADC. The Arduino shall sample the data at a fixed rate (e.g., 100 Hz).
- **Data Transmission** The Arduino shall transmit digitized samples to the FPGA via UART communication. The FPGA shall correctly receive and synchronize the incoming serial data stream.
- **Signal Processing** The FPGA shall implement filtering (e.g., moving average) to reduce noise, calculate data trends (increasing/decreasing tencency), moving averages. The FPGA shall detect peaks in the signal corresponding to heartbeats. The system shall compute the heart rate in beats per minute (BPM) using time intervals between peaks.
- **Display Output** The computed BPM value and other data shall be shown on the FPGA's 7-segment display depending on which button was last pressed.
- **Error Handling** The system shall handle missing or invalid data samples gracefully. In case of signal loss, the display shall indicate an error or pause state.

Non-Functional requirements

- **Performance** The system shall process incoming data in real time with minimal latency.

- **Reliability** The system shall maintain consistent data transfer without loss or corruption during normal operation. The hardware modules shall recover automatically from UART synchronization errors.
- **Scalability** The AXI4-Stream-based architecture shall allow adding new processing blocks (e.g., variance, filtering, anomaly detection) without major redesign.
- **Usability** The display output shall be easily readable and continuously updated. The design shall require minimal user configuration to operate.
- **Maintainability** The VHDL design shall use modular coding practices with clear documentation. System parameters (sampling rate, filter size) shall be easily adjustable in code.

1.1.5 Expected Results

The system should be able to display the heart rate values on the 7-segment display in continuous way in real-time. This relies on accurate filtering and heartbeat detection from the raw alanlogue signal. This project aims to demonstrate of the benefits of FPGA-based parallel processing in biomedical data handling providing table, low-latency system operation.

1.2 Project Plan

This chapter aims to outline the proposed timeline for project development, using a **Gantt chart**, key element in project management. The structure of the chart timeline will follow this semester's structure, having 13 weeks for development and 1 week for delivery.

This chart lists the tasks to be performed on the vertical axis, and time intervals on the horizontal axis. The width of the horizontal bars in the graph shows the duration of each activity. One bar unit is equivalent to one week.

This approach not only helps in keeping track of current project progress, but it is also helpful tool used in planning, calculating and allocating time for certain tasks, and then, after project delivery, it plays a crucial role in project evaluation. The image below illustrates the project plan with current progress. In order to view progress in real time, please access this link. For paper based lecture of this documentation please do not hesitate to contact me regarding access to this resource.

ACTIVITY	PLAN START	PLAN DURATION	ACTUAL START	ACTUAL DURATION	PERCENT COMPLETE
Choosing project	2	1	2	1	100%
Overeaf outline	2	1	3	1	100%
Bibliographic research	2	2	3	2	50%
Proj. proposal+Intro.	2	1	3	1	100%
Bibliographic sinthesys	2	2	4	1	50%
Project plan creation	3	1	3	1	100%
Revision and adjustments	4	10	4	10	0%
Analysis	4	2	4	2	0%
Initial Design Development	4	2	4	2	0%
Refined Design Development	6	2	6	2	0%
Implementation	6	4	6	4	0%
Testing and Simulation	10	2	10	2	0%
Results and Conclusions	11	1	11	1	0%
Documentation	1	13	3	13	15%
Final refinements	12	2	12	2	0%
Project delivery	14	1	14	1	0%

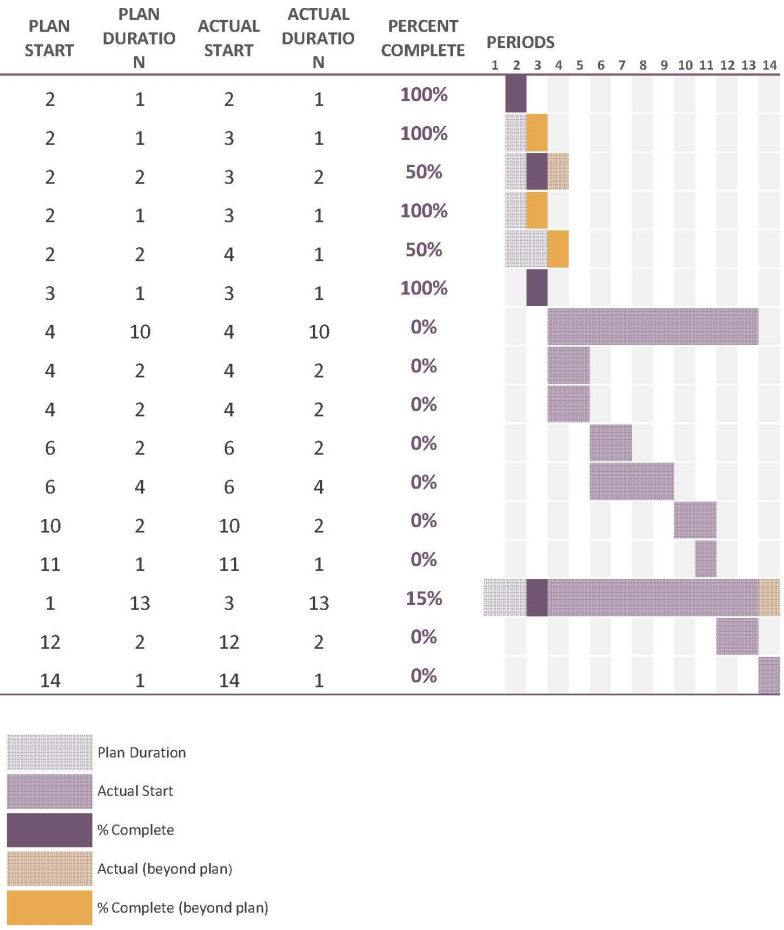

Figure 1.1: Project plan using Gantt chart

Chapter 2

Bibliographic research

The AXI protocol, part of ARM's AMBA 4 specification, defines a standardized interface for high-performance communication between modules in a system-on-chip. AXI4-Stream, a variant of AXI, is specifically designed for continuous, unidirectional data streams, making it ideal for real-time signal processing applications. Unlike memory-mapped AXI, AXI4-Stream does not use addresses, allowing data to flow efficiently from one processing block to another with minimal overhead. Data transfer is controlled using handshake signals: TVALID indicates valid data, TREADY signals readiness to accept data, and TLAST marks the end of a packet or frame. The protocol supports pipelining and backpressure, enabling modular, low-latency designs that can scale to multiple processing stages. AXI4-Stream is widely used in FPGA designs for sensor data processing, video/audio pipelines, and DSP applications, providing a flexible framework for connecting hardware modules. Its modularity and reliability make it well-suited for a heart-rate monitoring project, where streaming sensor data must be filtered and analyzed in real time.[2]

Gao et al. present in their paper an embedded image processing system on the ZYNQ-7000 SoC that combines high-speed data processing with efficient data transmission integrating the AXI4-Stream bus to provide a low-latency solution. Computationally intensive tasks are implemented in the FPGA's programmable logic (PL) for parallel processing, while the processor system (PS) handles flexible operations. High-Level Synthesis (HLS) was used to generate IP cores in PL, accelerating feature extraction and coordinate calculation. The AXI4-Stream bus was used for the data transfer, enabling fast streaming of image and coordinate data while simplifying the circuit. Experiments showed that this approach achieved 15.6× higher processing speed compared to implementation on the ARM processor alone. [4]

Medical applications and research also adopts FPGA-based platforms fos signal aquisition and processing, providing a flexible alternative to expensive commercial systems. Velarte at al. use a Cortex-M1 softcore processor on a Xilinx Artix-7 FPGA that enables modular hardware blocks for signal acquisition, conditioning, processing, and storage. The RHD2000 IC ensures low-noise, high-fidelity biopotential recording, with real-time visualization through a GUI. Notably, the system does not use AXI4-Stream; instead, it relies on custom FPGA interfacing for data transfer and processing. Tests demonstrate robust sampling rates and reliable performance with minor deviations from commercial systems. The platform illustrates how FPGA designs can provide low-latency, customizable signal processing for biomedical research in a cost-effective and adaptable way.[9]

Singh's and their fellow researchers' study demonstrates FPGA-based hardware acceleration for linear prediction analysis in speech coding, addressing the high computational cost of

processing large audio streams. The system uses a hardware/software co-design with an AXI4-Stream interface to efficiently transfer data between modules. Optimizations at the algorithmic level (autocorrelation and Levinson–Durbin) enable high-speed processing and reduced resource usage. Implementation on a Zynq Zybo FPGA achieves over 99% speed improvement and a 60% reduction in resources, highlighting the benefits of FPGA acceleration for real-time signal processing. The work illustrates how AXI4-Stream can support low-latency, high-throughput data streaming in embedded systems.[7]

Rohith's and Ram's thesis explores FPGA-based implementation of image processing algorithms for autonomous driving, addressing the high computational demand of real-time lane detection and tracking. The design integrates three modules—Edge Detector, Hough Transform, and Kalman Filter—optimized for FPGA to achieve fast processing and compact system architecture. FPGAs allow rapid prototyping and efficient synthesis compared to traditional approaches, enabling testing and optimization of complex algorithms in a short time. The study highlights the advantages of FPGA architectures for real-time, parallel data processing in applications requiring low-latency performance.[6]

Qianyu and Bing's paper proposes an FPGA-based hardware-software co-simulation framework for mixed-signal (AMS) system-on-chip designs to overcome efficiency bottlenecks of traditional software simulations. The architecture combines an ARM processor (PS) and FPGA logic (PL), where the PS handles alanlogue circuit equations and the PL accelerates digital logic at RTL level. Data transfer between PS and PL is achieved through an AXI4 interface, supporting high-bandwidth streaming (5.6 GB/s). The design incorporates optimized multipliers, time integrators, and dual-step synchronization to maintain high precision (quantization error ¡0.5 LSB, sync error ¡0.1%). Implementation on a Xilinx ZCU102 platform achieves 12.4× speedup, low power (1.2 W), and high accuracy (RMSE reduced from 3.2% to 1.8%). This study demonstrates the effectiveness of FPGA-based streaming and parallel processing for accelerating complex mixed-signal simulations.[5]

Campanini and Torsello's thesis presents an FPGA-based embedded system for real-time stress level estimation, using ECG and EDA signals. The design employs a heterogeneous Zynq™ 7000 SoC, combining a dual-core CPU with an FPGA hardware accelerator developed in High-Level Synthesis (HLS). The accelerator performs signal processing and feature extraction, while the CPU runs a classification algorithm (XGBoost) on Petalinux. Data transfer between the accelerator and CPU is handled via AXI4-Stream, enabling efficient, low-latency streaming of physiological data. The system was validated against simulation models, showing accurate classification and modularity for future extensions. This study demonstrates the benefits of FPGA-based streaming and parallel processing for real-time physiological signal analysis and affective computing applications.[3]

Chapter 3

Analysis

3.1 Problem Domain

This section focuses on "what" the project needs to do: understanding the problem, identifying requirements and defining what the project needs to achieve.

The current problem is: **how do we operate our MAX30100 Pulse Oximeter in such way, that the streamed derived data would show up on our basys board's seven-segment display, providing useful information to the user.**

The data output from the pulse oximeter is alanlogue, its key properties include amplitude (maximum signal strength), frequency (rate of change), and phase (position in the cycle). They are susceptible to noise and distortion, which degrades signal quality, but can provide a more accurate representation of a natural signal compared to digital signals.

The problem becomes more complex, when the Basys3's technical sheet is examined, because it only accepts digital signals as input on its Pmod ports. In this case, there are 2 solutions:

- Physically creating an alanlogue to digital converter or purchasing one - time and money intensive solution;
- Using a microcontroller as an ADC bridge between the sensor and the FPGA board communicationg via UART (Arduino's RX-TX pins and Basys3's Pmod ports)- fast and money intensive solution (I already own a both boards).

Another aspect that needs to be taken into consideration, that Mega 2560 board's digital output pins produce 5V output, but the Basys3 board takes 3.3V. This calls for the usage of a voltage divider as follows.

This concludes the additional hardware components needed in order to make this project work. Each of these components impose a communication protocol explained in the Design chapter.

- MAX30100 Pulse oximeter - Atmel Atmega 2560 development Board: I^2C;
- Atmel Atmega 2560 development Board - Basys3 : UART.

Upon successful connection of the components and establishing information flow, the soon-to-be-processed data receives the spotlight.

3.2 Data processing

The type of data that gets transmitted to the FPGA, that is 16 bit binary encoded digital values of the IR sensor measurements. For this project no RED sensor data was used. This

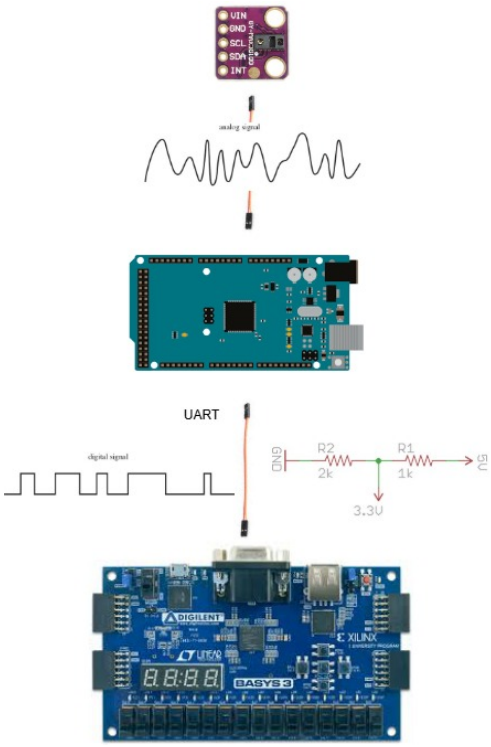

Figure 3.1: Microcontroller as ADC bridge with voltage divider - Data Flow diagram.

means that there will be no SpO_2 (blood oxygen level saturation) measurement data. This (IR) metric alone allows us to derive the following information from the recorded sensor data:

- Beats per minute (i.e. BPM)
- Average BPM (with a 1 minute reference period)
- Heart Rate Variability (HRV)
- Stress estimation
- RR (R-to-R interval - time between two consecutive heartbeats, measured in seconds (or milliseconds))

It is important to note that the sensor sampling frequency is programmable from the interface it has with the microcontroller. Programmable values are in the range of 50-100 Hz[1]. In this project we would like to receive the most possible data, such we set the sampling rate to the highest possible value.

Another aspect to note is the nature of the IR signal:

- Contains a DC baseline (slow drift - caused by sensor contact, ambient light, finger pressure, movement - justifies filtering it out)
- Contains AC pulsatile component (heart beat)
- Contains random noise (requires to be removed)
- Contains motion induced noise

Thus, correct interpretation of IR signal requires filtering and data processing. The entire data processing will take place on the FPGA board. In order to be able to detect heart beats, one should look for peaks in the data.

In order to be able to handle the high frequency of data while ensuring data integrity, and maintaining low overhead and low-latency response time justifies the usage AXI4-Stream compilant modules while designing the system. Another advantage is the modularity of compilant blocks and the built-in flow control ensures that no data is lost. Additionally, this eliminates the need of designing FIFOs, buffers or custom handshake logic. It also provides the possibility of easy integration with future extensions.

In order to achieve this the designed sistem must fulfill the following non-functional requirements:

- System latecny should be $<$ 1 beat detection period and FPGA clock frequency should be able to match the sampling rate or be able to run at higher frequency.
- Frequency of false-peak detections shall be $< 5\%$.
- UART communication must tolerate occasional dropped bytes - might corrupt samples.
- All filters shall be modular AXI4-Stream blocks.
- The system must fit within Basys3 resources.
- Display must refresh at least every second.

Chapter 4

Design

This section focuses on "how" the project will achieve the goal and aims to provide the necessary steps for a successful implementation.

4.1 Heart Rate Detection

Pulse Oximeter measure oxygen saturation. Before we learn the principles of how pulse Oximeter work, we need to have an understanding of what oxygen saturation is. Oxygen enters the lungs and then is passed on into blood. The blood carries the oxygen to the various organs in our body. The main way oxygen is carried in our blood is by means of hemoglobin. The hemoglobin without oxygen we will call de oxygenated hemoglobin. The hemoglobin with oxygen, we will call oxygenated hemoglobin.

A pulse oximeter detects a pulse by using two different wavelengths of light and a sensor to distinguish pulsating arterial blood from other tissues. It works by shining red and infrared light through a body part like a fingertip and measuring the amount of light absorbed. Because oxygenated and deoxygenated hemoglobin absorb light differently, the device can calculate oxygen saturation and also detect the pulsatile changes in arterial blood flow, which indicates the pulse rate.

As blood flows through the arteries with each heartbeat, the volume of blood pulsates, which changes the amount of light that is absorbed. Venous blood and other tissues like skin, bone, and muscle have a relatively constant volume and do not contribute to the pulsatile signal. **By comparing the amount of light absorbed during the pulsatile phase (systole) versus the non-pulsatile phase (diastole), the oximeter can isolate the arterial blood flow. The pulsatile component is used to calculate the heart rate.**[8]

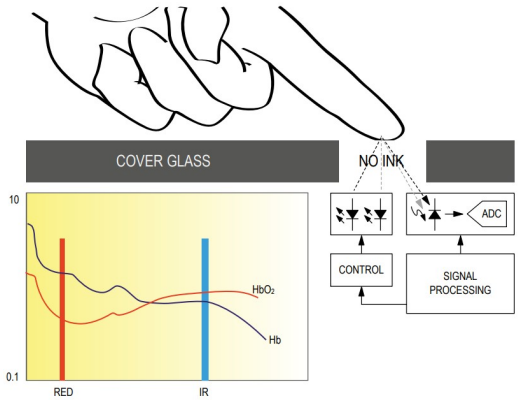

Figure 4.1: MAX30100 internal structure.

4.2 Communication Protocols

4.2.1 I^2C Communication

The Inter-Integrated Circuit (I^2C) protocol is a **synchronous, serial communication bus commonly used to connect sensors to microcontrollers.** It operates over two wires:

- SCL (Serial Clock) – generated by the master
- SDA (Serial Data) – bidirectional data line

In this project, the Arduino acts as the I^2C master, while the MAX30100 PPG sensor acts as the I^2C slave. The master controls all communication by generating the clock and initiating data transfers.

Each I^2C transaction begins with a START condition, followed by the 7-bit slave address of the MAX30100 and a read/write bit. The addressed device acknowledges by pulling SDA low (ACK). After this, registers inside the sensor can be written to or read from. The MAX30102 continuously stores sampled IR values in internal FIFO registers; the Arduino periodically reads these FIFO registers using I^2C read operations. When all required data is transferred, the master sends a STOP condition, releasing the bus.[10]

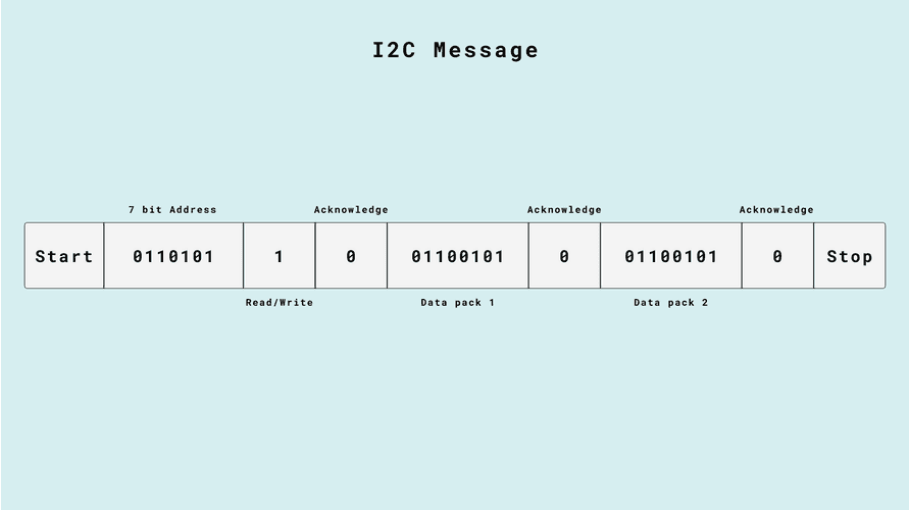

Figure 4.2: Sample I^2C message.
https://docs.arduino.cc/learn/communication/wire/

I^2C is well-suited for this project because it:

- requires only two wires;
- allows the microcontroller to configure sampling rate, LED current, and FIFO updates;
- provides reliable, low-latency access to the IR data stream needed for real-time FPGA processing;
- would support adding multiple sensors on the same bus, the project could be extended with other sensor data.

Once the Arduino retrieves the IR samples over I^2C, it forwards them over UART to the FPGA, where they enter the AXI4-Stream processing pipeline.

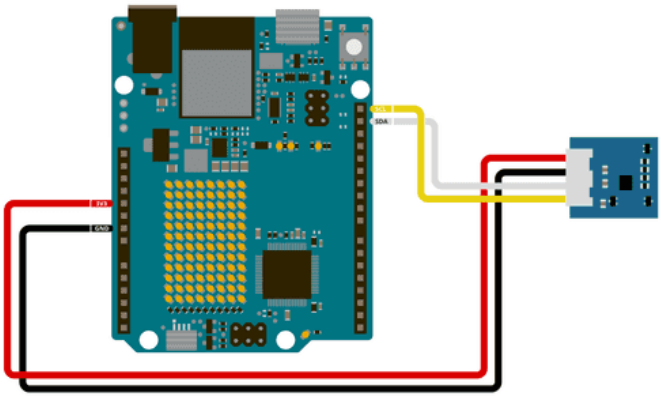

Figure 4.3: Sample I²C communication between sensor and Arduino board.
`https://docs.arduino.cc/learn/communication/wire/`

4.2.2 UART Communication

The Universal Asynchronous Receiver/Transmitter (UART) protocol is a simple, **asynchronous, serial communication method** commonly used to transfer digital data between two devices. Unlike synchronous protocols, UART does not require a clock line; instead, both devices must operate at the same baud rate (bits per second), such as 9600 or 115200 baud.

UART frames data into packets consisting of:

- 1 start bit (pulls the line low to mark the beginning)
- 8 data bits (least significant bit first, typical for sensor values)
- optional parity bit (not used in this project)
- 1 or more stop bits (line is high)

In this project, the Arduino acts as the UART transmitter, sending each sampled IR value to the FPGA, which acts as the UART receiver. The FPGA must detect the start bit, sample the data bits at the correct clock intervals, and reconstruct the transmitted 16-bit IR sample.

UART is well suited for this project because it requires only TX → RX and GND, it supports continuous real-time data streaming, it can handle the 50–100 Hz sample rate of the MAX30100 , the FPGA can convert UART bytes directly into an AXI4-Stream sample stream, forming the input to the filter pipeline.

The downside of UART is that noise or timing mismatch can produce corrupted samples, so this project includes a noise/artefact filter in the FPGA pipeline to reject unrealistic values and ensure stable peak detection.

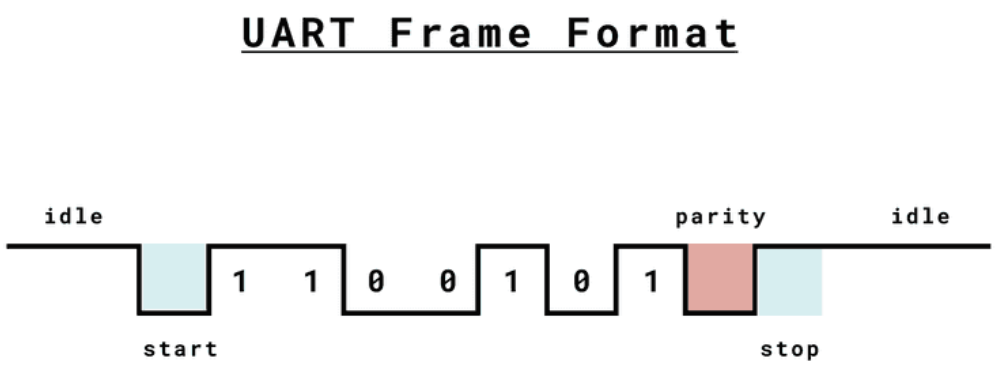

Figure 4.4: UART Frame format with sample data.
`https://docs.arduino.cc/learn/communication/uart/`

4.2.3 AXI4 - ARM AMBA protocol family

AXI4-Stream (AXIS) is a lightweight, high-speed, unidirectional data-streaming protocol from the ARM AMBA 4 specification. Unlike memory-mapped AXI, AXI4-Stream does not use addresses; instead, it delivers a continuous stream of data samples from one hardware module to another. This makes it ideal for real-time signal-processing pipelines, such as the heart-rate filtering chain used in this project.

AXI4-Stream uses very few signals, the most important being:

- TVALID – sender asserts when data on TDATA is valid
- TREADY – receiver asserts when it can accept data
- TDATA – the payload (16-bit IR sample)
- TLAST – optional marker for end-of-frame

A transfer occurs only when TVALID = 1 and TREADY = 1, providing built-in flow control (or backpressure). If a downstream module is busy or not ready, it lowers TREADY, and the upstream module automatically waits. No data is lost.

The four rules every AXI4-Stream-Compilant module must follow are:

- Data transfer occurs ONLY when TVALID = 1 AND TREADY = 1 - *ensures lossless, perfectly synchronized data movement*
- TDATA (and all sideband signals) must remain stable while TVALID = 1 and TREADY = 0 - *prevents corrupted samples and ensures deterministic flow control*
- A master must NEVER depend on TREADY before asserting TVALID - *ensures AXI4-Stream remains push-based, not pull-based*
- A slave may deassert TREADY at any time (backpressure allowed) - *provides flow control and protects the downstream modules from overflow*

Rules regarding master modules are stricter then the ones regarding slave modules.

Data transfer occurs in 3 cases: valid before ready handshake, ready before valid handshake and valid with ready handshake. [2]

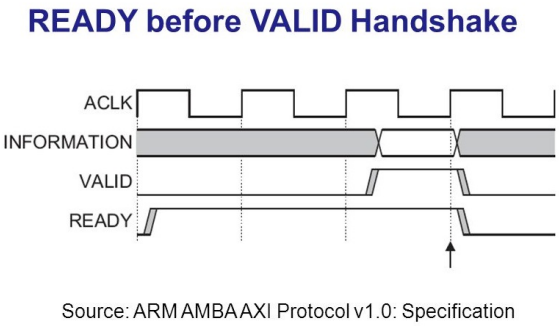

Figure 4.5: Ready before Valid handshake.

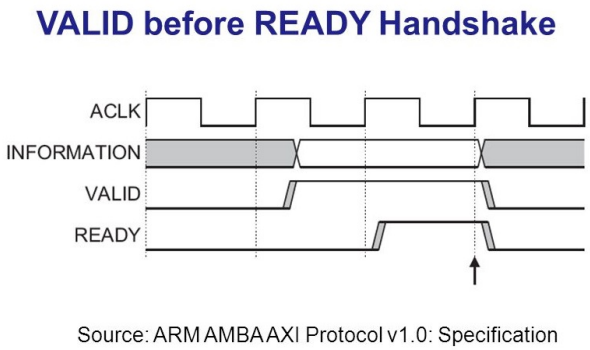

Figure 4.6: Valid before Ready handshake.

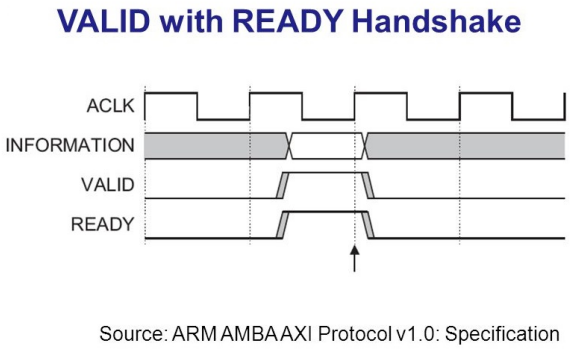

Figure 4.7: Valid with Ready handshake.

4.2.4 Voltage divider

The mathematical formula describing the case in which we have one input voltage and one output voltage can be calculated as follows:

$$V\_out = V\_in \cdot \frac{R_2}{R_1 + R_2}$$

Using this formula, one can calculate the magnitude of the necessary resistors in order to filter (high-pass) the Arduino's digital output pin to provide suitable input for the FPGA board, that is, 3.3 Volts.

Solving the equation for R2, with input, desired output voltages and R1 given - one might own multiple units of 1k Ohm resistors due to it's popularity and low price - R2 results: 1941.18 2k Ohm. This rounding would allow one to use 3 pieces of 1k resistors in order to achieve the goal. Further implementation will be conducted using the voltage divider described above.

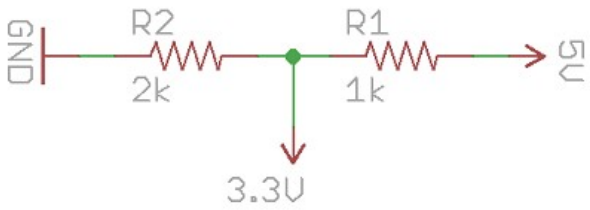

Figure 4.8: Voltage Divider circuit to convert 5V output into 3.3V input.
`https://randomnerdtutorials.com/how-to-level-shift-5v-to-3-3v/`

4.3 FPGA Design

Note that all modules named **axis_*** do have the valid and ready signals, and are AXI4-Stream compilant. These signals won't be individually listed in case of each module, only in this disclaimer.

Another aspect to note is, that the modules are presented in the order in which the pipeline is structured. As in the figure below.

debouncer

This module aims to ensure that one button push performed by the user is in fact triggering one enable signal for the system. This module is crucial for selection of the data we would like to visualize and triggering system resets.

uart_bit_sampler

This module implements a UART receiver (RX) bit sampler that converts a serial input signal into parallel 8-bit data.

It first synchronizes the asynchronous rx line to the system clock, then, using a baud-rate–derived counter, it detects the start bit, waits half a bit period to sample in the middle of the bit, and proceeds to sample 8 data bits (LSB first) at precise bit intervals. After the stop bit, it transfers the assembled byte to data_out and generates a one-clock-cycle pulse on data_ready to signal that a complete byte has been received.

It transforms a 115200-baud serial stream into clean, clocked 8-bit parallel data synchronized to a 100 MHz system clock.

uart_word_assembler_axis

This module converts incoming UART bytes into AXI4-Stream 16-bit words and properly handshakes them to downstream logic by instantiating the uart_word_assembler component that groups two received 8-bit UART bytes into one 16-bit word. When a complete 16-bit word

is ready (word_valid_int), the module latches it into an internal register (data_reg) — but only if it is new and the previous word has already been transmitted.

It then presents this word on the AXI4-Stream master interface:

- m_axis_tdata carries the 16-bit word
- m_axis_tvalid is asserted when a word is available
- It waits for m_axis_tready before clearing the valid flag and allowing the next word to load

In short, this block acts as a bridge between UART reception and AXI4-Stream, buffering each 16-bit word and ensuring proper AXI handshake timing so no data is lost.

uart_word_assembler

This module assembles two consecutive 8-bit UART bytes into a single 16-bit word.

It operates as a simple two-state finite state machine:

- In WAIT_HI state, it waits for rx_ready = '1' and stores the incoming byte as the high byte.
- In WAIT_LO state, it waits for the next byte, stores it as the low byte, concatenates the two bytes (hi_byte & rx_byte) into a 16-bit word, and asserts word_valid for one clock cycle.

After producing the 16-bit word, it returns to WAIT_HI to begin assembling the next word.

This block converts a stream of UART bytes into aligned 16-bit parallel words and generates a valid pulse whenever a complete word is formed and forwards it to the uart_word_assembler_axis, that serves as the axis wrapper.

axis_saturator

clamps each incoming data sample to a predefined range (MIN_VAL to MAX_VAL).

axis_dc_filter - not used, but possible improvement

The purpose of this module is to remove baseline drift and extract the AC component of the IR signal, such that one is able to observe the pulsations - for accurate peak detection. The baseline is caused by finger pressure, skin characteristics, ambient light. This module implements a high-pass filter using the formula:

$$y[n] = x[n] - moving\_average(x[n])$$

In order to implement the module, a simple ALU and a running sum with Shift registers is necessary. The moving average/running sum module is used also in the case of the next module.

axis_moving_avg

It aims to reduce high-frequency noise. In fact, it only computes the average of the input stream over a window of width N. N is a and exponent of 2, the division is performed by shifting with the exponent.

$$y[n] = \frac{x[n] + x[n-1] + \ldots + x[n-N+1]}{N}$$

axis_peak_detector

This is one of the central elements of the system, it detects heart beats (local maximum of the signal values).

It is given a stream of samples, arriving as AXI4-Stream, and it should detect local maximum as already mentioned (peaks), output a pulse when a peak occurs, optionally a peak value and peak index (sample counter).

This would be done by a digital synthesizable algorithm, with detection logic as follows:

$$if(x[n-2] < x[n-1]) \ \wedge \ (x[n] < x[n-1]) \ \wedge \ (x[n-1] > threshold) \quad \Rightarrow \quad peakAt : x[n-1]$$

Meaning that a sample x[n] is a peak if:

- Slope was positive:
$$x[n] > x[n-1]$$
- Then becomes negative:
$$x[n] < x[n-1]$$
- Amplitude is above noise threshold:
$$x[n] > T_{peak}$$

axis_rr_interval

This module measure time between consecutive peaks. it receives the peak_detected signal and the system clock and provides the RR interval in milliseconds, by counting the clock cycles between peaks.

Needs a counter that is peak_detected enabled, and needs to make the correct value to time conversion depending on the FPGA's internal clock configuration. Very short intervals below RR_MIN are treated as false peaks and ignored. The output is buffered with more significant AXI backpressure on the branch that computest stress and HRV.

$$RR = (t_n^{peak} - t_{n-1}^{peak}) \cdot T_{clk}$$

This module will serve the basys of:

- BPM
- average BPM
- HRV
- stress estimation
- minimum and maximum average BPM

axis_bpm_instant

This is another central element of our sytem that converts RR interval into instantaneous BPM. As input it takes the RR interval and returns the BPM value (integer) based on the following formula:

$$BPM = \frac{60}{RR}$$

This module's output data would make it possible to display real-time heart rate on the FPGA's seven-segment display and it only entails a division - supported by **ieee.numeric_std.all**.

axis_bpm_avg

This module implements an AXI4-Stream sliding average filter for BPM values. It stores the last WINDOW_SIZE BPM samples, maintains a running sum, and computes their average each time a new sample is accepted. Once the window is full, it continuously outputs the updated average while respecting AXI handshaking and backpressure.

axis_hrv_calculator

Computing the Heart rate variability is in fact the computation of RMSSD (root-mean-square of successive RR differences). This takes as input a buffer of last N RR intervals, based on which it returns the HRV score.

$$\text{RMSSD} = \sqrt{\frac{1}{N-1}\sum_{i=1}^{N-1}(RR_i - RR_{i+1})^2}$$

This module provides an advanced health metric (mimicking Garmin-like stress data). In order to be able to implement this module, the following are needed:

- RR Interval Buffer (Shift Register - that stores last N RR intervals)
- Displacement Calculator (successive values)

  $$d[n] = RR[n] - RR[n-1]$$

  Can be easily implemented: diff <= signed(RR_n) - signed(RR_prev);
- Square & Sum Module This module would compute diff^2 and accumulate the sum over the window.

  squared <= diff * diff;

  sum <= sum + squared;
- Divide & Square Root Module This module would take as input the accumulated sum and provide RMSSD value (integer) by performing a fixed-point division the calculating the square root (using a simple iterative integer sqrt by function defined as follows:

Listing 4.1: Division and square root function in VHDL

```
function isqrt(n : unsigned) return unsigned is
variable x0  : unsigned(n'length-1 downto 0) := (others => '0');
variable x1  : unsigned(n'length-1 downto 0) := (others => '0');
variable tmp : unsigned(n'length-1 downto 0);
begin
for i in reverse 0 to n'length-1 loop
tmp := x1;
tmp(i) := '1';
if tmp * tmp <= n then
x1 := tmp;
end if;
end loop;
return x1;
end function;
```

stress_estimator

This module estimates a stress level based on averaged heart rate (BPM) and heart rate variability (RMSSD). It applies a simple combinational heuristic: higher BPM and lower HRV correspond to higher stress. The output is a 2-bit encoded stress level (00=low, 01=mild, 10=medium, 11=high).

min_tracker

This module tracks the minimum value of a streaming signal once it has been enabled. After enable is asserted (sticky), it compares each valid input sample to the current minimum and updates the stored value if a smaller one is found. It outputs the current minimum and asserts minimum_valid once at least one valid sample has been captured.

max_tracker

This module tracks the maximum value of a streaming signal after it has been enabled. Once enable is asserted (sticky), each valid input sample is compared to the stored maximum and replaces it if it is larger. The current maximum is continuously output, and maximum_valid is asserted once at least one valid sample has been processed.

uart_word_to_bcd

This module is used to convert numeric BPM into BCD digits. On the inputs it receives data from the filters implemented, after processing it provides four 4+1-bit BCD digits ready to decode the display. The extra bit comes from the need to encode more than 16 characters

ssd_controller

This module will act as a multiplexer that decides what data is shown on the FPGA's display, depending on the current state of data processing and selected data channel.

seven_segment_display

This module will perform the actual configuration of FPGA's seven-segment display, by turning the leds on based on the data input.

4.3.1 Arduino design

The Arduino Design module is a sipmle one, only performing conversion from analog to digital the forwarding all data buffered byte by byte through the UART interface towards the FPGA board. Data collection is done on the default I2C communication interface of the Arduino Mega 2560 board, with MAX30100's public library provided modified raw data collection example code, available online.

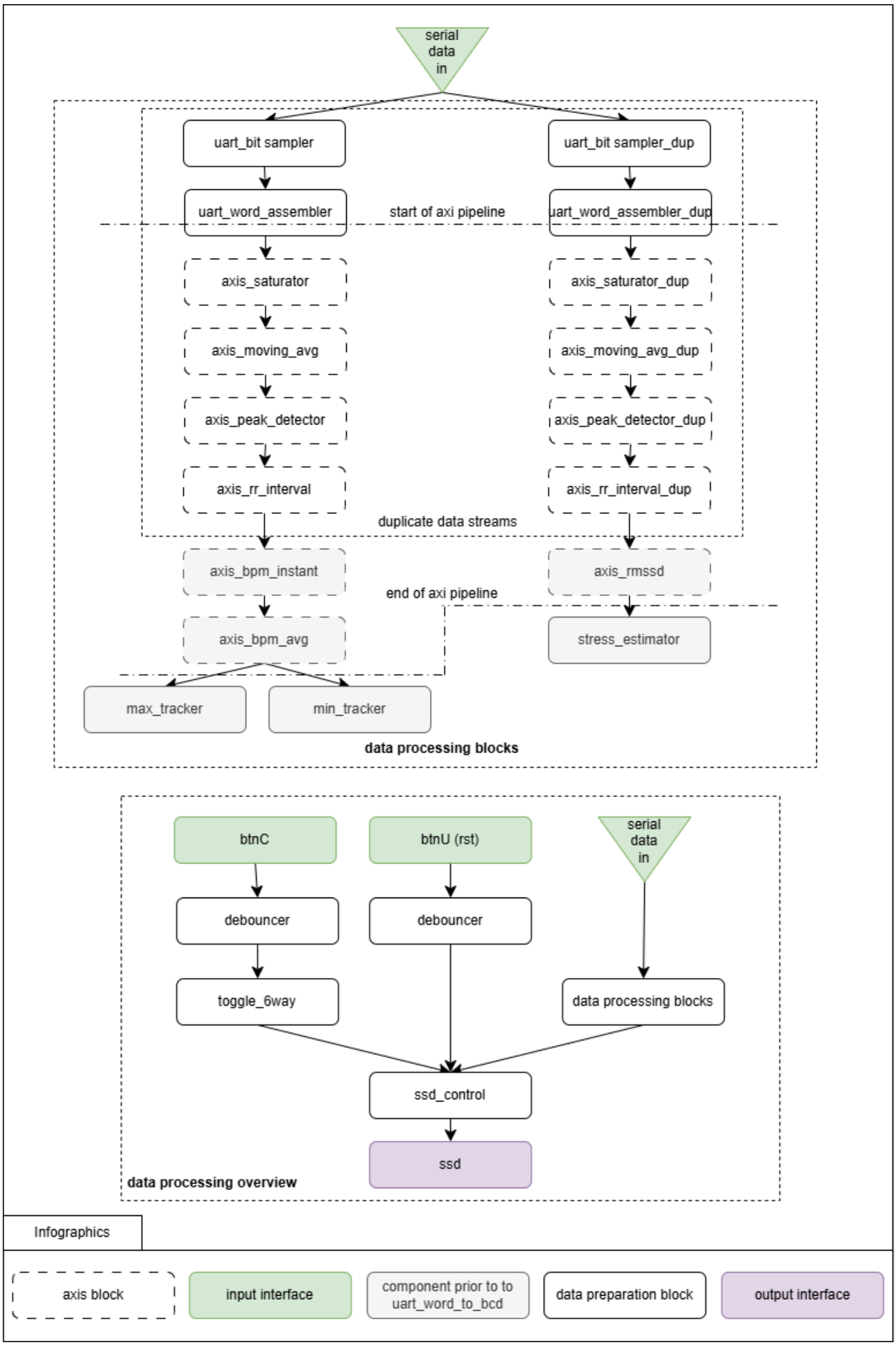

Figure 4.9: FPGA data pipeline

Chapter 5

Implementation

5.1 Overview

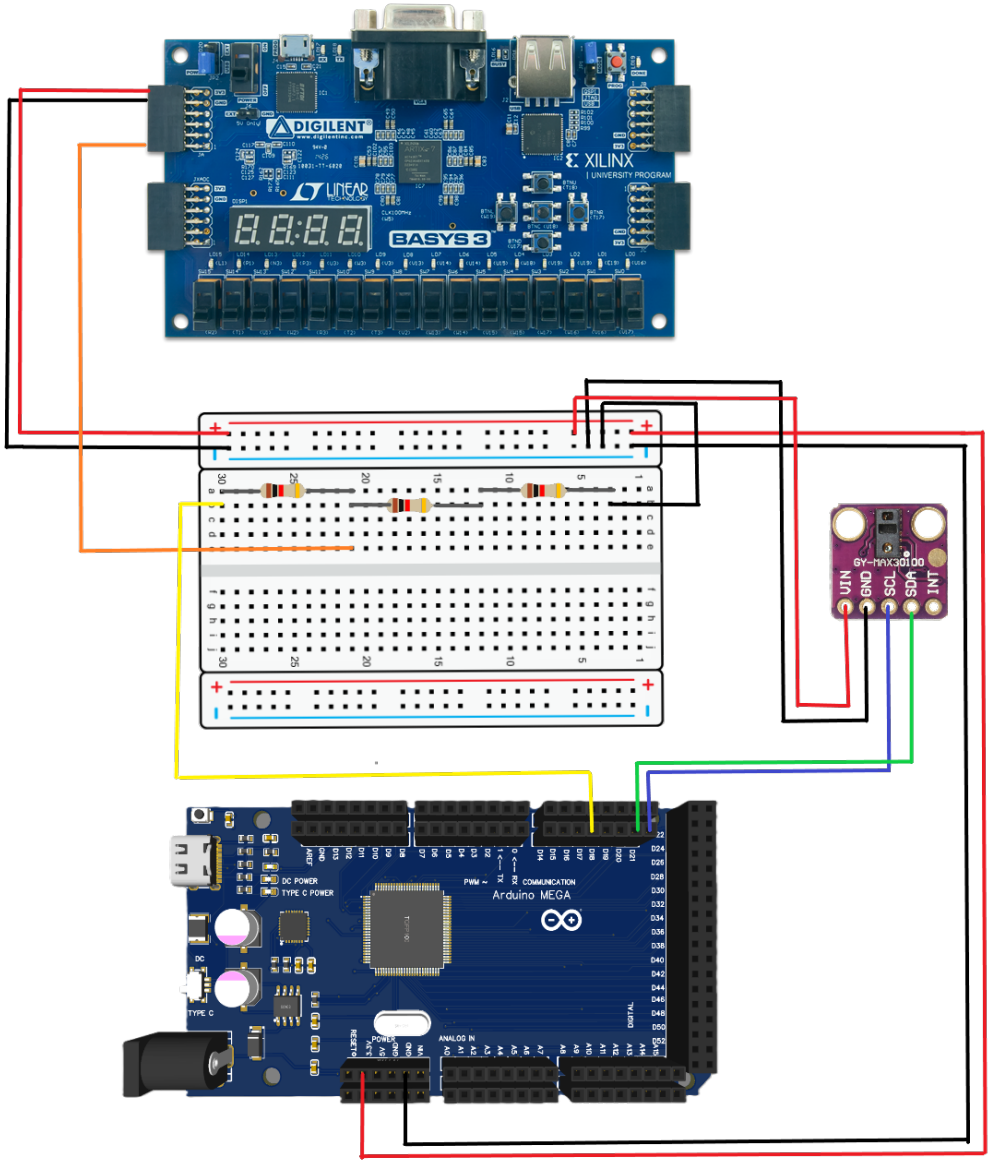

Figure 5.1: Planned system setup

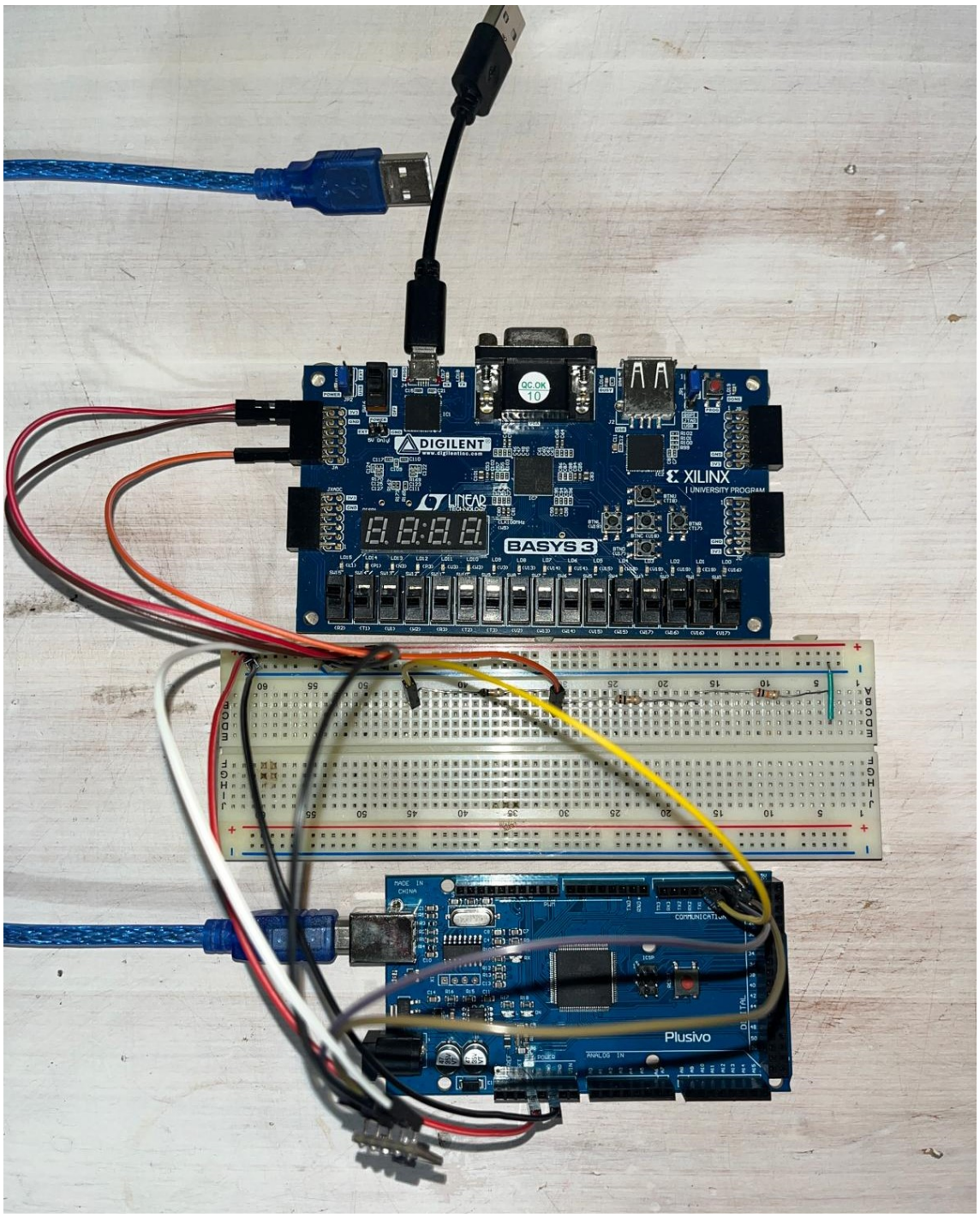

Figure 5.2: Physical system setup

5.2 Arduino configuration and implementation

The Arduino IDE needs to be set up for the exact type of board and correct port in order to be able to establish connection and be able to communicate with the host PC.

The basis of this design component is the Example provided by the Sensor manufacturer in the library files that is used to access Raw data transferred by the sensor. This data is forwarded on the above described UART serial interface to the FPGA byte by byte using The built-in Serial1 function and the Mega 2560 compatible board's serial interface.

Listing 5.1: MAX30100 sensor interfacing on Arduino

```
/*
Arduino-MAX30100 oximetry / heart rate integrated sensor library
Copyright (C) 2016  OXullo Intersecans <x@brainrapers.org>

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>.
*/

// The example shows how to retrieve raw values from the sensor
// experimenting with the most relevant configuration parameters.
// Use the "Serial Plotter" app from arduino IDE 1.6.7+ to plot the output

#include <Wire.h>
#include "MAX30100.h"

// Sampling is tightly related to the dynamic range of the ADC.
// refer to the datasheet for further info
#define SAMPLING_RATE           MAX30100_SAMPRATE_100HZ

// The LEDs currents must be set to a level that avoids clipping and maximises the
// dynamic range
#define IR_LED_CURRENT          MAX30100_LED_CURR_4_4MA
#define RED_LED_CURRENT         MAX30100_LED_CURR_4_4MA
#define MAX30100_LED_CURR_0MA   0x00



// The pulse width of the LEDs driving determines the resolution of
// the ADC (which is a Sigma-Delta).
// set HIGHRES_MODE to true only when setting PULSE_WIDTH to
// MAX30100_SPC_PW_1600US_16BITS
// chose to set it to 13bit res, 200ns resolution so data would fit between 0-9999
//for raw ir data display
#define PULSE_WIDTH         MAX30100_SPC_PW_200US_13BITS
#define HIGHRES_MODE        true


// Instantiate a MAX30100 sensor class
MAX30100 sensor;

unsigned long lastSend = 0;
const unsigned long SEND_INTERVAL = 10; //10 miliseconds
```

```
53
54
55  void setup()
56  {
57    Serial.begin(115200);     // USB serial (for debugging)
58    Serial1.begin(115200);    // UART to FPGA (TX1 = Pin 18)
59
60    Serial.print("Initializing MAX30100..");
61
62    // Initialize the sensor
63    // Failures are generally due to an improper I2C wiring, missing power supply
64    // or wrong target chip
65    if (!sensor.begin()) {
66      Serial.println("FAILED");
67      for(;;);
68    } else {
69      Serial.println("SUCCESS");
70    }
71
72    // Set up the wanted parameters
73    sensor.setMode(MAX30100_MODE_SPO2_HR);
74    //sensor.setLedsCurrent(IR_LED_CURRENT, MAX30100_LED_CURR_0MA);
75    sensor.setLedsCurrent(IR_LED_CURRENT, MAX30100_LED_CURR_0MA);
76    sensor.setLedsPulseWidth(PULSE_WIDTH);
77    sensor.setSamplingRate(SAMPLING_RATE);
78    sensor.setHighresModeEnabled(HIGHRES_MODE);
79  }
80
81  void loop()
82  {
83    uint16_t ir, red;
84    sensor.update();
85
86    while (sensor.getRawValues(&ir, &red)) {
87
88      unsigned long now = millis();
89      if (now - lastSend >= SEND_INTERVAL) {
90        lastSend = now;
91
92        //Serial.print(ir, HEX);
93        ir &= 0x1FFF;   // fit 13-bit range
94
95        uint8_t ir_hi = (uint8_t)(ir >> 8);
96        uint8_t ir_lo = (uint8_t)(ir & 0xFF);
97
98        Serial.print(ir);
99        Serial.println(", ");
100       Serial1.write(ir_hi);
101       Serial1.write(ir_lo);
102     }
103   }
104 }
```

5.3 FPGA Implementation

Below there is the final RTL diagram of the system described in the previous chapters.

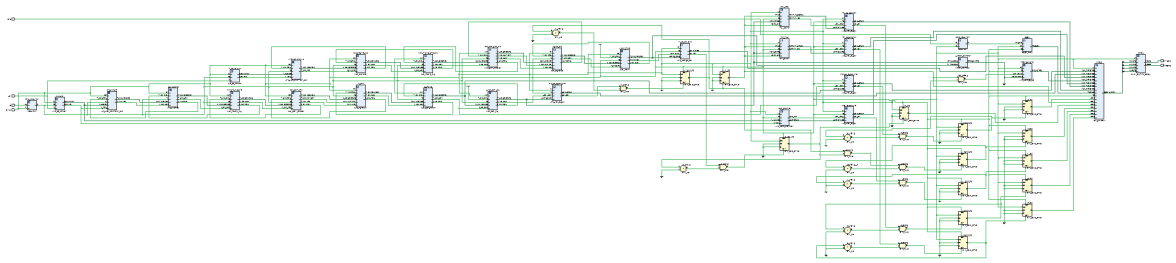

Figure 5.3: RTL Schematic overview

The image below shows the entry point of serial data coming through the UART interface. Note, that there are two separate streams of these starting with the very first serial data processing component: uart_bit_sampler. Another notable component is uart_word_assembler_axis. This is the starting point of out AXI4-Stream pipeline.

This section will provide detailed explanation of component implementation where it seems necessary, and where it is needed.

5.3.1 uart_bit_sampler component

As previously presented, this module receives an asynchronous UART RX signal and reconstructs an 8-bit byte using a clocked finite state machine (FSM). The key timing goal is to sample each UART bit in the middle of its bit period, where the signal is most stable and least affected by edge jitter.

After detecting a start bit falling edge, it waits HALF_TICKS to align to the middle of the start bit, then samples each data bit every BAUD_TICKS cycles, which hits the middle of each data bit slot. After one stop bit time, it asserts data_ready for one clock cycle and outputs the 8-bit word.

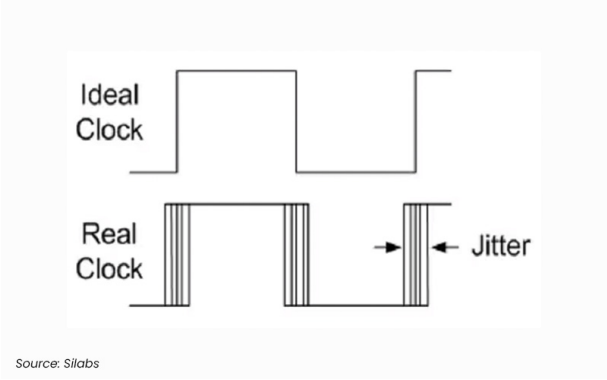

Figure 5.4: Clock Jitter in Electronics
`https://genetroncorp.com/blog/understanding-clock-jitter-causes-and-effects-in-electronics/`

BAUD_TICKS = number of FPGA clock cycles per UART bit

$$BAUD\_TICKS = \frac{CLK\_FREQ}{BAUD\_RATE}$$

It is rounded to the nearest integer using + 0.5.
HALF_TICKS = BAUD_TICKS / 2 (used to move to the center of the start bit)
Example at 100 MHz and 115200 baud:

$$BAUD\_TICKS = \frac{100,000,000}{115200} = 868.0(5) \approx 869$$

$One\ \ bit\ \ time \approx 869\ \ clock\ \ cycles \approx 8.69\ \ s$
$Half\ \ bit\ \ time \approx 434\ \ cycles \approx 4.34\ \ s$
Then rx data is synchronized for metastability protection through passing it through a 2-flip-flop synchronizer:
rx_sync1 ⇐ rx rx_sync2 ⇐ rx_sync1 rx_s ⇐ rx_sync2
This reduces metastability risk and ensures the FSM only sees RX transitions aligned to clk. The tradeoff is a small latency (1–2 clock cycles), which is negligible relative to a UART bit period.
The FSM has four states: IDLE ⇒ START ⇒ DATA ⇒ STOP.

- IDLE: detect the start bit edge
- START: move to the middle of the start bit
- DATA: sample 8 data bits, one per bit period
- STOP: wait one bit time, then output the byte content...

Condition: Arduino uses default UART data format(SERIAL_8N1: 1 start no parity 2 stop bit).

Listing 5.2: uart_bit_sampler.vhd

```
content...library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

library work;
use work.common_pkg.all;

entity uart_bit_sampler is
generic (
  CLK_FREQ  : integer := C_CLK_FREQ;
  BAUD_RATE : integer := C_BAUD_RATE
);
port (
  clk        : in  std_logic;
  rst        : in  std_logic;
  rx         : in  std_logic;

  data_out   : out std_logic_vector(7 downto 0);
  data_ready : out std_logic
);
end entity;

architecture Behavioral of uart_bit_sampler is

constant BAUD_TICKS        : integer := integer(real(CLK_FREQ) /
real(BAUD_RATE) + 0.5);
```

```
27  constant BAUD_TICKS_MINUS1 : integer := BAUD_TICKS - 1;
28  constant HALF_TICKS        : integer := BAUD_TICKS / 2;
29
30  type state_t is (IDLE, START, DATA, STOP);
31  signal state      : state_t := IDLE;
32
33  signal bit_index  : integer range 0 to 7 := 0;
34  signal baud_count : integer := 0;
35  signal shift_reg  : std_logic_vector(7 downto 0) := (others => '0');
36
37  -- rx synchronizer
38  signal rx_sync1, rx_sync2, rx_s : std_logic := '1';
39
40  begin
41
42  process(clk)
43  begin
44    if rising_edge(clk) then
45      rx_sync1 <= rx;
46      rx_sync2 <= rx_sync1;
47    end if;
48  end process;
49
50  rx_s <= rx_sync2;
51
52  --FSM
53  process(clk)
54  begin
55    if rising_edge(clk) then
56    data_ready <= '0';
57
58      if rst = '1' then
59        state      <= IDLE;
60        bit_index  <= 0;
61        baud_count <= 0;
62
63      else
64        case state is
65
66        when IDLE =>
67        --wait for start bit falling edge
68        if rx_s = '0' then
69          baud_count <= 0;
70          state      <= START;
71        end if;
72
73        when START =>
74        -- wait HALF_TICKS to get to the middle of the start bit
75        if baud_count = HALF_TICKS then
76          baud_count <= 0;
77          bit_index  <= 0;
78          state      <= DATA;
79        else
```

```
80            baud_count <= baud_count + 1;
81          end if;
82
83          when DATA =>
84          -- wait one full bit period between bit samples
85          if baud_count = BAUD_TICKS_MINUS1 then
86            baud_count <= 0;
87
88          -- Sample the current data bit in the middle of its slot
89          shift_reg(bit_index) <= rx_s;  -- LSB first
90
91          if bit_index = 7 then
92            state <= STOP;
93            else
94            bit_index <= bit_index + 1;
95          end if;
96
97          else
98            baud_count <= baud_count + 1;
99          end if;
100
101         when STOP =>
102         -- wait one more bit time for stop bit then output the byte
103         if baud_count = BAUD_TICKS_MINUS1 then
104           data_out   <= shift_reg;
105           data_ready <= '1';
106           state      <= IDLE;
107           baud_count <= 0;
108         else
109           baud_count <= baud_count + 1;
110         end if;
111
112         end case;
113       end if;
114     end if;
115   end process;
116
117   end Behavioral;
118
```

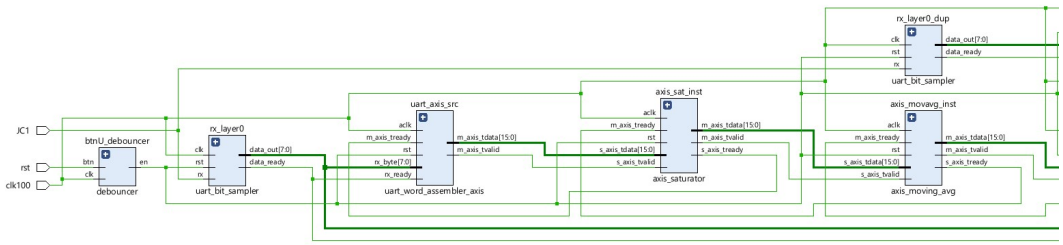

Figure 5.5: Start of FPGA data pipeline implementation

5.3.2 uart_word_assembler_axis component

This component is a AXI4-Stream wrapper, meaning that it wraps the 16-bit word output into an AXI4-Stream master interface, acting as a bridge between the UART receive logic and the AXI streaming pipeline.

The component below provides this one with a word and a valid signal: word16_int and word_valid_int the 1-cycle pulse when the word becomes ready. Because AXI4-Stream requires that data and valid remain stable until the receiver accepts them, the wrapper adds a one-word buffer:

- data_reg: stores the word to be transmitted to AXI (word16_int from uart_word_assembler)
- have_data: acts as a "buffer full" flag and directly drives m_axis_tvalid (based on word_valid_int from uart_word_assembler)

Timing and handshake behavior: when new word is received (it is signaled by word_valid_int becoming '1'), the wrapper tries to latch the word into data_reg, but only if the buffer is empty (have_data='0').

Listing 5.3: uart_word_assembler.vhd

```
1  if word_valid_int='1' and have_data='0' then
2    data_reg  <= word16_int;
3    have_data <= '1';
4  end if;
```

This means the wrapper can hold exactly one pending word at a time, that is waiting to be consumed by the AXI pipeline. Once have_data='1', the wrapper asserts m_axis_tvalid='1' and holds m_axis_tdata stable. The downstream module accepts the word when it asserts m_axis_tready='1'. On that clock edge, the wrapper drops have_data and **the AXI handshake takes place**.

Listing 5.4: uart_word_assembler.vhd

```
1  if have_data='1' and m_axis_tready='1' then
2    have_data <= '0';
3  end if;
```

Note the following:

- Backpressure support: If the downstream pipeline deasserts m_axis_tready, the wrapper keeps m_axis_tvalid high and holds m_axis_tdata constant until acceptance.
- Throughput limitation: Because the wrapper only has a one-word buffer, if word_valid_int had produced a new word while have_data='1', that new word is not stored (it would be effectively missed). In practice, this is okay since UART produces words much slower than the speed at which the AXI pipeline consumes them. An alternative solution could be adding FIFO modules in-between components.

The wrapper prevents repeatedly latching the same value if the upstream word output remains stable while word_valid_int pulses, by checking whether word16_int equals word16_prev.

Listing 5.5: uart_word_assembler_axis.vhd

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
```

```
library work;
use work.common_pkg.all;

entity uart_word_assembler_axis is
Port (
aclk        : in  STD_LOGIC;
rst     : in  STD_LOGIC;  -- active low normally, made it active high

--from uart bit sampler
rx_ready    : in  STD_LOGIC;
rx_byte     : in  STD_LOGIC_VECTOR(7 downto 0);

--axi4 stream master interface
m_axis_tdata  : out STD_LOGIC_VECTOR(15 downto 0);
m_axis_tvalid : out STD_LOGIC;
m_axis_tready : in  STD_LOGIC
);
end uart_word_assembler_axis;

architecture Behavioral of uart_word_assembler_axis is
component uart_word_assembler is
Port (
  clk        : in  STD_LOGIC;
  rst        : in  STD_LOGIC;
  rx_ready   : in  STD_LOGIC;
  rx_byte    : in  STD_LOGIC_VECTOR (7 downto 0);
  word16     : out STD_LOGIC_VECTOR (15 downto 0);
  word_valid : out STD_LOGIC
);
end component;

signal word16_int, word16_prev, data_reg      : STD_LOGIC_VECTOR(15 downto 0) :=
                                                (others => '0');
signal word_valid_int, have_data : STD_LOGIC := '0';

begin

u_word_asm: uart_word_assembler
  port map (
    clk        => aclk,
    rst        => rst,
    rx_ready   => rx_ready,
    rx_byte    => rx_byte,
    word16     => word16_int,
    word_valid => word_valid_int
);

process(aclk)
begin
  if rising_edge(aclk) then
    if rst = '1' then
      data_reg  <= (others => '0');
      have_data <= '0';
```

```
58      else
59        -- if we receive a new 16-bit word from the assembler, latch it
60        if (word_valid_int = '1') and
61          (word16_int /= word16_prev) and
62          (have_data = '0') then
63
64          data_reg    <= word16_int;
65          word16_prev <= word16_int;
66          have_data   <= '1';
67        end if;
68
69        -- if we currently have data and the AXI sink accepted it,
70        -- drop have_data to allow next word to be loaded.
71        if (have_data = '1') and (m_axis_tready = '1') then
72          have_data <= '0';
73        end if;
74      end if;
75    end if;
76  end process;
77
78  m_axis_tdata  <= data_reg;
79  m_axis_tvalid <= have_data;
80
81  end Behavioral;
82
```

5.3.3 uart_word_assembler component

This component in instantiated by uart_word_assembler_axis. It converts the UART byte stream into 16-bit words. It operates as a 2-state FSM, with state, one that waits for the first byte from the word (high byte - WAIT_HI), and one that waits for the second byte (low byte - WAIT_LO).

Timing behavior:

- In WAIT_HI, when rx_ready='1', it latches rx_byte into hi_byte and moves to WAIT_LO.
- In WAIT_LO, when rx_ready='1', it latches rx_byte as the low byte and assembles the 16-bit word:

$$word16 = hi\_byte \ \ \& \ \ lo\_byte$$

It then asserts word_valid for one clock cycle and returns to WAIT_HI.

Listing 5.6: uart_word_assembler.vhd

```
1   library IEEE;
2   use IEEE.STD_LOGIC_1164.ALL;
3
4   entity uart_word_assembler is
5   Port (  clk : in STD_LOGIC;
6       rst : in STD_LOGIC;
7       rx_ready : in STD_LOGIC;
8       rx_byte : in STD_LOGIC_VECTOR (7 downto 0);
```

```
9       word16 : out STD_LOGIC_VECTOR (15 downto 0);
10      word_valid : out STD_LOGIC);
11  end uart_word_assembler;
12
13  architecture Behavioral of uart_word_assembler is
14
15  signal hi_byte   : std_logic_vector(7 downto 0) := (others => '0');
16  signal lo_byte   : std_logic_vector(7 downto 0) := (others => '0');
17  signal word16_inter    : std_logic_vector(15 downto 0) := (others => '0');
18
19  type rx_state_t is (WAIT_HI, WAIT_LO);
20  signal rx_state  : rx_state_t := WAIT_HI;
21  signal word_valid_inter : std_logic := '0';
22
23  begin
24
25    process(clk)
26    begin
27    if rising_edge(clk) then
28      if rst = '1' then
29        rx_state        <= WAIT_HI;
30        hi_byte         <= (others => '0');
31        lo_byte         <= (others => '0');
32        word16_inter    <= (others => '0');
33        word_valid_inter<= '0';
34
35      else
36        case rx_state is
37
38        when WAIT_HI =>
39        if rx_ready = '1' then
40          hi_byte         <= rx_byte;
41          rx_state        <= WAIT_LO;
42          -- starting new word -> no longer valid
43          word_valid_inter<= '0';
44        end if;
45
46        when WAIT_LO =>
47        if rx_ready = '1' then
48          lo_byte         <= rx_byte;
49          word16_inter    <= hi_byte & rx_byte;
50          word_valid_inter<= '1';
51          rx_state        <= WAIT_HI;
52        end if;
53
54        end case;
55      end if;
56    end if;
57    end process;
58
59    word16     <= word16_inter;
60    word_valid <= word_valid_inter;
61  end Behavioral;
```

```
62
63
```

5.3.4 ssd_conroller component

The ssd_controller module selects what is shown on the 4-digit seven-segment display based on the current mode (0...6). It also implements a short mode-dependent animation when the system starts up or when the user changes the mode. This helps the user identify what metric is currently selected (HR, Avg HR, HRV, etc.).

The output digits_out is a 20-bit bus containing 4 digits, where each digit is encoded as a 5-bit "character" value (from common_pkg, e.g., CH_H, CH_R, CH_BLANK, etc.). The downstream ssd driver time-multiplexes these four digits onto the physical display.

In case data of interest is not ready, meaning, that is no valid sample generated yet, the display will show "PRoc".

Listing 5.7: ssd_conroller.vhd

```
1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use IEEE.NUMERIC_STD.ALL;
4
5  library work;
6  use work.common_pkg.all;
7
8  entity ssd_controller is
9  Port (
10   clk          : in  std_logic;
11   rst          : in  std_logic;
12   mode         : in  std_logic_vector(3 downto 0);
13
14   digits_mode0 : in  std_logic_vector(15 downto 0); --hr
15   digits_mode1 : in  std_logic_vector(15 downto 0); --average hr
16   digits_mode2 : in  std_logic_vector(15 downto 0); --hrv
17   digits_mode3 : in  std_logic_vector(15 downto 0); --stress
18   digits_mode4 : in  std_logic_vector(15 downto 0); --raw ir
19   digits_mode5 : in  std_logic_vector(15 downto 0); --max hr
20   digits_mode6 : in  std_logic_vector(15 downto 0); --min hr
21
22   valid0 : in  std_logic; --hr
23   valid1 : in  std_logic; --avg hr
24   valid2 : in  std_logic; --hrv
25   valid3 : in  std_logic; --stress
26   valid4 : in  std_logic; --raw ir
27   valid5 : in  std_logic; --min hr
28   valid6 : in  std_logic; --max hr
29
30   digits_out   : out std_logic_vector(19 downto 0)
31 );
32 end ssd_controller;
33
34 architecture Behavioral of ssd_controller is
35
36 --animation state + show state
```

```
37  type state_t is (ANIM, SHOW);
38  signal state_reg, state_next : state_t;
39
40  signal curr_mode_reg, curr_mode_next : std_logic_vector(3 downto 0);
41  -- Animation timer: 0.5 s at 100 MHz
42  -- 0.5 s * 100e6 =  50,000,000 cycles
43  -- 1.0 s * 100e6 = 100,000,000 cycles
44  constant ANIM_TICKS : unsigned(26 downto 0) :=
45                                  to_unsigned(C_ANIMATION_DURATION - 1, 27);
46  signal   anim_counter : unsigned(26 downto 0) := (others => '0');
47  signal   anim_done    : std_logic;
48
49  begin
50
51  process(clk)
52  begin
53    if rising_edge(clk) then
54      if rst = '1' then
55        state_reg     <= ANIM; -- show animation at startup
56        curr_mode_reg <= "0000";
57      else
58        state_reg     <= state_next;
59        curr_mode_reg <= curr_mode_next;
60      end if;
61    end if;
62    end process;
63
64    -- animation timer: counts only in ANIM
65    process(clk)
66    begin
67    if rising_edge(clk) then
68      if rst = '1' then
69        anim_counter <= (others => '0');
70      else
71        if state_reg = ANIM then
72          if anim_counter = ANIM_TICKS then
73            anim_counter <= (others => '0');
74          else
75            anim_counter <= anim_counter + 1;
76          end if;
77        else
78          anim_counter <= (others => '0');
79        end if;
80      end if;
81    end if;
82    end process;
83
84    anim_done <= '1' when (state_reg = ANIM and anim_counter = ANIM_TICKS)
85    else '0';
86
87    --next-state+mode logic
88    process(state_reg, mode, curr_mode_reg, anim_done)
89    begin
```

```
90        state_next     <= state_reg;
91        curr_mode_next <= curr_mode_reg;
92
93        case state_reg is
94
95        when ANIM =>
96          curr_mode_next <= mode;
97
98          if anim_done = '1' then
99            state_next <= SHOW;
100         end if;
101
102       -- normal display; if mode changes, go back to ANIM
103       when SHOW =>
104         if mode /= curr_mode_reg then
105           -- new mode selected -> start animation for that mode
106           state_next     <= ANIM;
107           curr_mode_next <= mode;
108         end if;
109
110       end case;
111     end process;
112
113     -- output logic: animation frame or selected mode digits
114     process(state_reg, curr_mode_reg,
115     digits_mode0, digits_mode1, digits_mode2, digits_mode3)
116     begin
117       digits_out <= (others => '0');
118
119       case state_reg is
120
121         when ANIM =>
122           case curr_mode_reg is
123             --Hr (blank)(blank)
124             when "0000"   => digits_out <= CH_H & CH_R & CH_BLANK & CH_BLANK;
125             --A_Hr
126             when "0001"   => digits_out <= CH_A & CH_UNDERSCORE & CH_H & CH_R;
127             --hrv_
128             when "0010"   => digits_out <= CH_H & CH_R & CH_U & CH_UNDERSCORE;
129             --Stress
130             when "0011"   => digits_out <= CH_S & CH_T & CH_R & CH_S;
131             --r_ir
132             when "0100"   => digits_out <= CH_R & CH_UNDERSCORE & CH_I & CH_R
133             --MAX;
134             when "0101"   => digits_out <= CH_N & CH_N & CH_A & CH_H;
135             --MIN
136             when "0110"   => digits_out <= CH_N & CH_N & CH_I & CH_N;
137             --ERR
138             when others   => digits_out <= CH_E & CH_R & CH_R & CH_BLANK;
139           end case;
140
141         when SHOW =>
142           --now output data from the corresponding mode
```

```
143             case curr_mode_reg is
144               when "0000"   =>
145               if valid0 = '1' then
146                 digits_out <= '0' & digits_mode0(15 downto 12) & '0'
147                 & digits_mode0(11 downto 8) & '0' & digits_mode0(7 downto 4)
148                 & '0' & digits_mode0(3 downto 0);
149               end if;
150               when "0001"   =>
151               if valid1 = '1' then
152                 digits_out <= '0' & digits_mode1(15 downto 12) & '0'
153                  & digits_mode1(11 downto 8) & '0' & digits_mode1(7 downto 4)
154                  & '0' & digits_mode1(3 downto 0);
155               else
156                 digits_out <= CH_p & CH_R & CH_o & CH_c;
157               end if;
158               when "0010"   =>
159               if valid2 = '1' then
160                 digits_out <= '0' & digits_mode2(15 downto 12) & '0'
161                  & digits_mode2(11 downto 8) & '0' & digits_mode2(7 downto 4)
162                   & '0' & digits_mode2(3 downto 0);
163               else
164                 digits_out <= CH_p & CH_R & CH_o & CH_c;
165               end if;
166               when "0011"   =>
167               if valid3 = '1' then
168                 digits_out <= '0' & digits_mode3(15 downto 12) & '0'
169                 & digits_mode3(11 downto 8) & '0' & digits_mode3(7 downto 4)
170                  & '0' & digits_mode3(3 downto 0);
171               else
172                 digits_out <= CH_p & CH_R & CH_o & CH_c;
173               end if;
174               when "0100"   =>  --note that there is no else!
175               if valid4 = '1' then
176                 digits_out <= '0' & digits_mode4(15 downto 12) & '0'
177                 & digits_mode4(11 downto 8) & '0' & digits_mode4(7 downto 4)
178                  & '0' & digits_mode4(3 downto 0);
179               end if;
180               when "0101"   =>
181                 if valid5 = '1' then
182                 digits_out <= '0' & digits_mode5(15 downto 12) & '0'
183                  & digits_mode5(11 downto 8) & '0' & digits_mode5(7 downto 4)
184                   & '0' & digits_mode5(3 downto 0);
185               else
186                 digits_out <= CH_p & CH_R & CH_o & CH_c;
187               end if;
188               when "0110"   =>
189               if valid6 = '1' then
190                 digits_out <= '0' & digits_mode6(15 downto 12) & '0'
191                  & digits_mode6(11 downto 8) & '0' & digits_mode6(7 downto 4)
192                   & '0' & digits_mode6(3 downto 0);
193               else
194                 digits_out <= CH_p & CH_R & CH_o & CH_c;
195               end if;
```

```
196
197             when others    => digits_out <= CH_E & CH_R & CH_R & CH_BLANK;
198           end case;
199
200       end case;
201   end process;
202
203   end Behavioral;
204
```

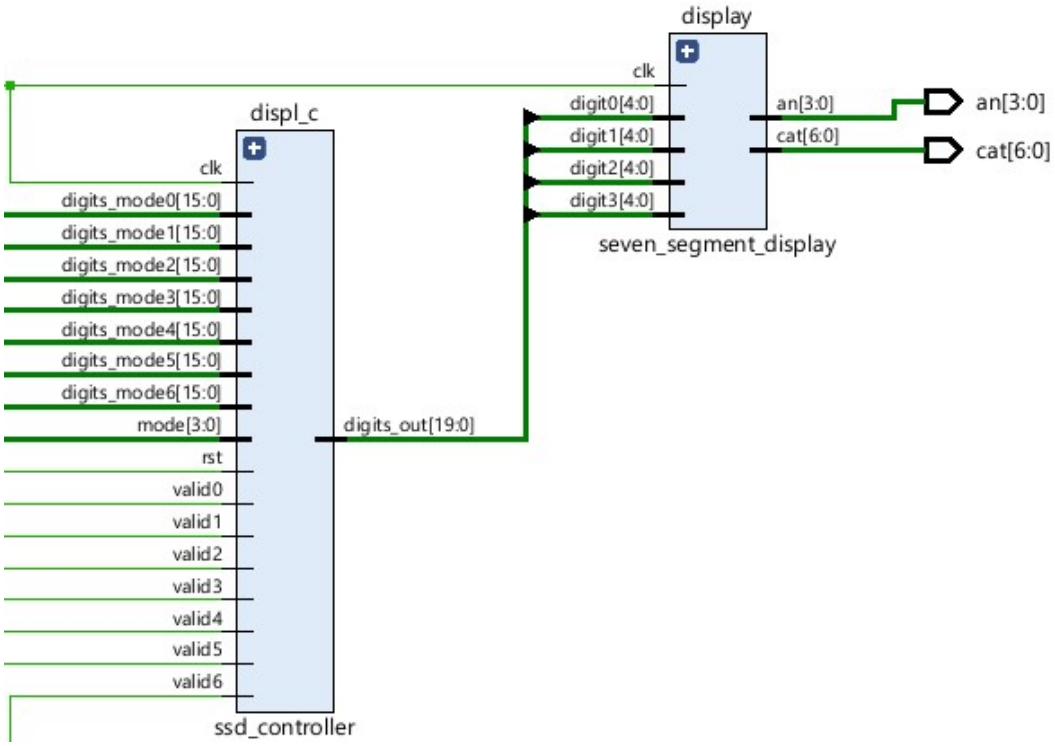

Figure 5.6: Display Modules

5.3.5 package: common_pkg

The package below is used as an entry point for system calibration, storing functions and constants.

Listing 5.8: common_pkg.vhd

```
1   library IEEE;
2   use IEEE.STD_LOGIC_1164.ALL;
3   use IEEE.NUMERIC_STD.ALL;
4
5   package common_pkg is
6
7   constant C_SAMPLE_RATE   : integer :=                    100;
8   constant C_DATA_WIDTH    : integer :=                    16;
9   constant C_IR_MIN        : integer :=                    1000;
10  constant C_IR_MAX        : integer :=                    8191;
11  constant C_CLK_FREQ      : integer :=                    100_000_000;    --100 MHz
12  constant C_BAUD_RATE     : integer :=                    115_200;         -- match
```

```
 --Serial1.begin(...)
constant C_ANIMATION_DURATION : integer :=              100_000_000; --1 second

constant C_AXIS_DATA_WIDTH : integer :=                 16;

constant C_AXIS_MOV_AVG_WINDOW_SIZE : integer :=        32; --MIST BE POWER OF TWO
constant C_AXIS_MOV_AVG_SUM_WIDTH : integer :=          22; --enough bits for sum
constant C_AXIS_MOV_AVG_SHIFT_BITS : integer :=         5;  --
-- LOG2(C_AXIS_MOV_AVG_WINDOW_SIZE)

constant C_AXIS_PEAK_DET_THRESHOLD : integer :=         1000;

constant C_AXIS_RR_INTERVAL_COUNTER_WIDTH : integer :=  16;
constant C_AXIS_RR_INTERVAL_RR_WIDTH : integer :=       16;
constant C_AXIS_RR_INTERVAL_RR_MIN : integer :=         80;

constant C_AXIS_BPM_WIDTH : integer :=                  16;
constant C_AXIS_AVG_BPM_WINDOW_SIZE: integer :=         16;
constant C_AXIS_RMSSD_WINDOW_SIZE: integer :=           16;
constant C_AXIS_RMSSD_WIDTH: integer :=                 16;

constant C_BPM_MEDIUM: integer :=   70;
constant C_BPM_HIGH: integer :=     90;
constant C_HRV_LOW: integer :=      30;
constant C_HRV_MEDIUM: integer :=   40;



--characters for ssd_display
constant CH_BLANK       : std_logic_vector(4 downto 0) := "11011";
constant CH_UNDERSCORE  : std_logic_vector(4 downto 0) := "10010";
constant CH_A           : std_logic_vector(4 downto 0) := "01010";
constant CH_U           : std_logic_vector(4 downto 0) := "10100";
constant CH_H           : std_logic_vector(4 downto 0) := "10000";
constant CH_R           : std_logic_vector(4 downto 0) := "10001";
constant CH_E           : std_logic_vector(4 downto 0) := "01110";
constant CH_S           : std_logic_vector(4 downto 0) := "00101";
constant CH_T           : std_logic_vector(4 downto 0) := "10101";
constant CH_I           : std_logic_vector(4 downto 0) := "10011";
constant CH_P           : std_logic_vector(4 downto 0) := "11000";
constant CH_o           : std_logic_vector(4 downto 0) := "11001";
constant CH_c           : std_logic_vector(4 downto 0) := "11010";
constant CH_N           : std_logic_vector(4 downto 0) := "11100";

function isqrt(n : unsigned) return unsigned;

end package;

package body common_pkg is

function isqrt(n : unsigned) return unsigned is
  variable x0  : unsigned(n'length-1 downto 0) := (others => '0');
  variable x1  : unsigned(n'length-1 downto 0) := (others => '0');
```

```
66      variable tmp : unsigned(n'length-1 downto 0);
67    begin
68    -- Binary restoring iteration
69      for i in n'length-1 downto 0 loop
70        tmp := x1;
71        tmp(i) := '1';
72        if tmp * tmp <= n then
73          x1 := tmp;
74        end if;
75      end loop;
76      return x1;
77    end function;
78
79    end package body;
80
81    content...
```

Chapter 6

Testing and validation

To verify the correctness of the digital signal-processing chain independently of the hardware environment, a mixed hardware–software testing methodology was adopted using **signal capture and simulation-based verification**. A custom Python application was developed to capture the raw sensor output transmitted by the Arduino over a serial interface. The recorded CSV dataset was subsequently used to simulate selected FPGA components in a controlled environment.

Advantages of This Testing Approach:

- Reproducibility: The same dataset could be replayed multiple times.
- Isolation of hardware variables: Sensor noise and UART timing effects could be separated from FPGA logic issues.
- Controlled debugging: Edge cases (e.g., signal spikes, flat signals, threshold crossings) could be examined precisely.
- Validation of mathematical correctness: Output waveforms from simulation could be compared directly with expected results.
- This hybrid methodology significantly accelerated debugging and ensured functional correctness of the signal-processing pipeline before full hardware integration.

The figure below shows how raw ir signal is reconstructed from the UART input on JC1 port. Thus figure shows an early stage of the simulation, when no data is valid except instant BPM and the raw digital input. It also illustrates the speed differences between UART and system clock (uppermost digital signal). Mode is set to 4, dedicated for output of raw ir values.

Another notable that the yellow marker marks the point where the first valid rx sample is created at $\approx$ 40 001 us. Until that point the display shows "r_ir", but after that it changes to (supposedly) 6258. In later simulations an animation delay was introduced to make sure, that the user observes the data on the output without the possibility of confusion. That delay is exactly 1 second. In this example the signal digits_out is 31448, and that reduces to the following:

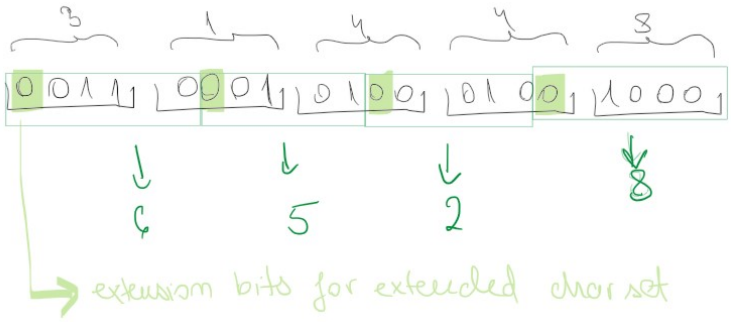

Figure 6.1: Digits out conversion to actual digits

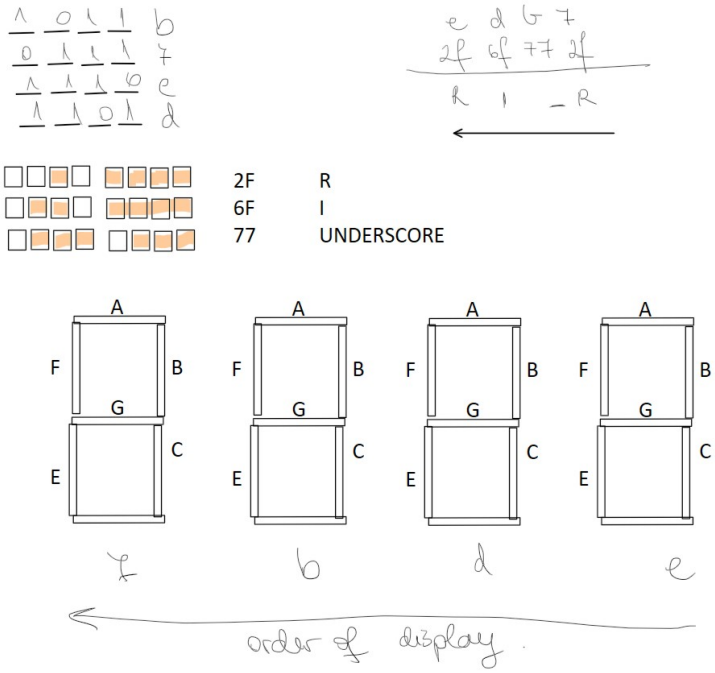

Figure 6.2: Raw ir values

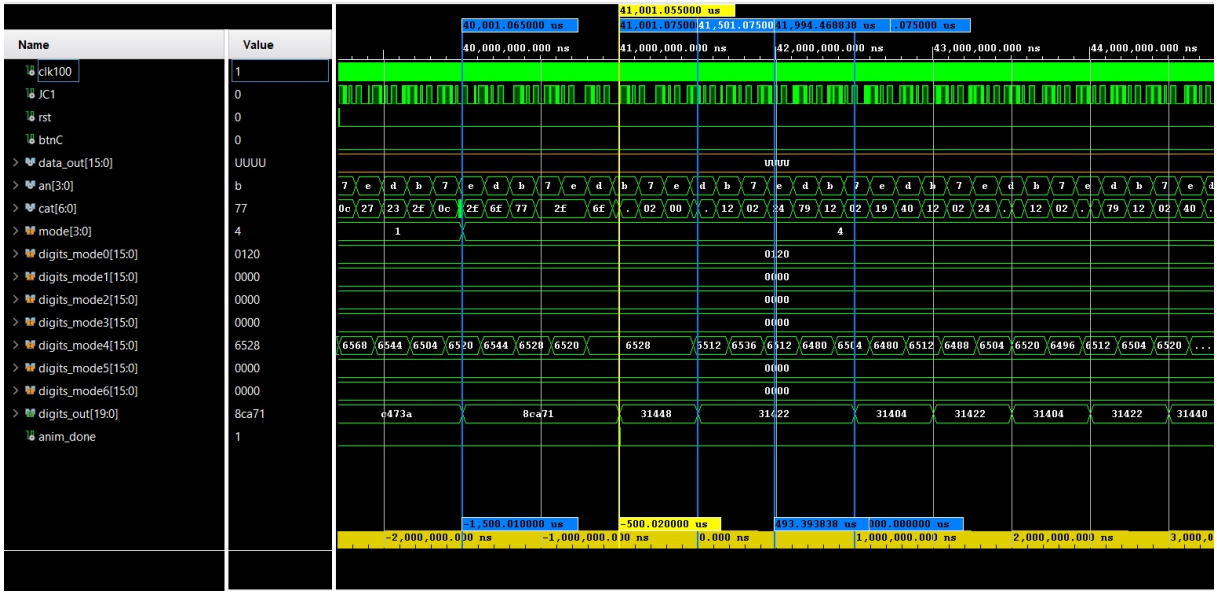

Figure 6.3: Raw ir values

Below you will find an actual overview of the whole system simulation. It can be observed exactly when the valid signals emerge, starting with v4 for valid raw ir sensor data. That is the first sampled word that came through the UART interface as it got assembled by the uart_word_assemble_axis component, entering the AXI4 pipeline. This event happens around 1.07 ms after system power-up.

Around 33.8 ms the first instant heart rate gets computed. This means that the latency of the AXI4 pipeline from start-up to the axi_bpm_inst is around 32 ms. For computing the average heart rate the latency of the AXI4-Stream only is around 423 ms, a significant stretch compared to instant computations. The HRV gets computed at 443.88 ms. Thus, the hrv branch stacks a latency around 19 ms compared to the bpm branch.

This would mean seeing results after 0.443 seconds from power-up. That is the theoretical processing time, in practice this number is significantly higher, much likely due to the software implementation of the Arduino side data processing imposed delays.

Notice, that once there is a valid average heart rate and heart rate variability, the stress gets computed once with the latest valid between the two of the previously mentioned.

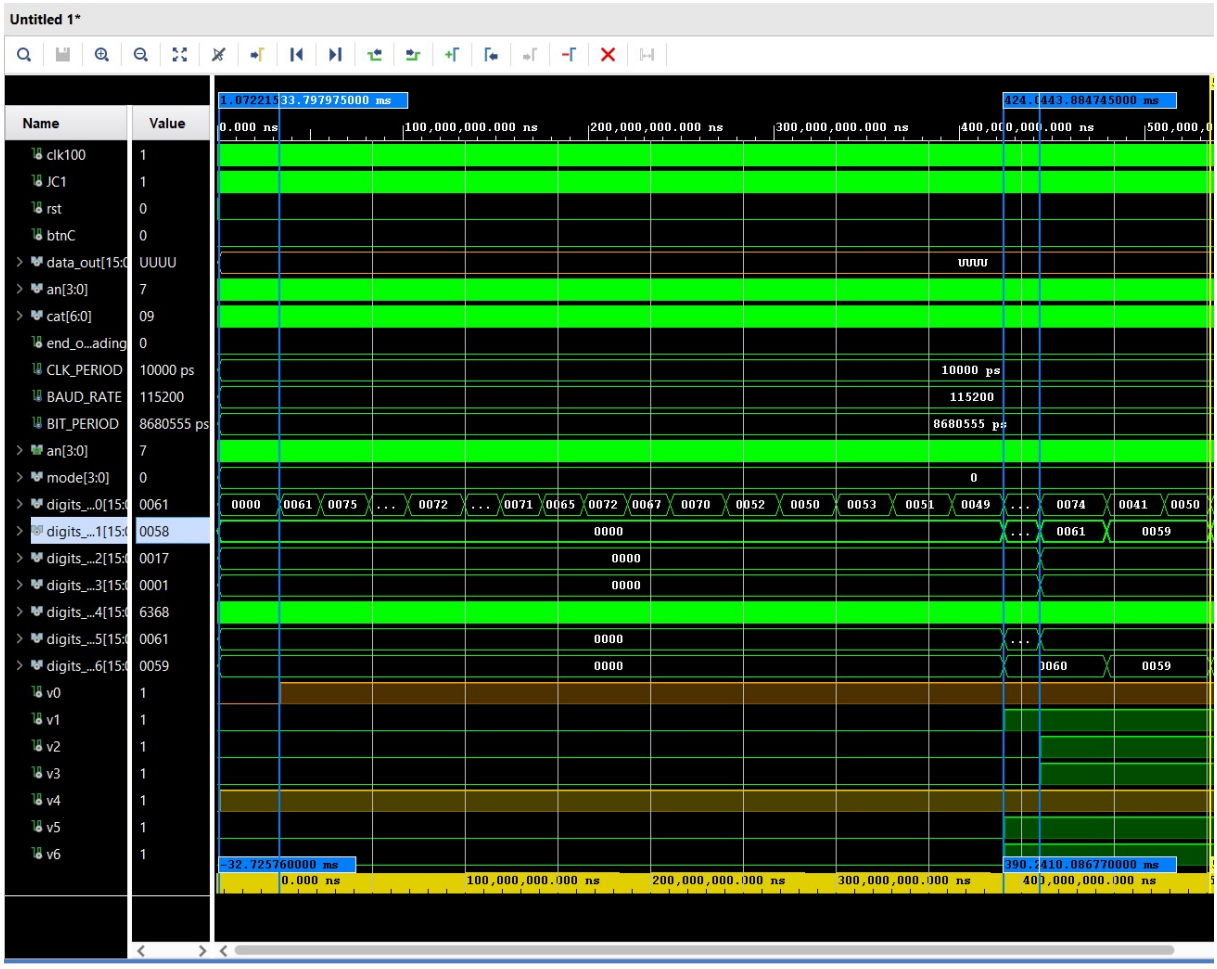

Figure 6.4: System simulation overview

Next we will observe how the average heart rate gets computed, the sliding window adder operates on a window of 16 values. To check it's correctness, we replicate the computations in excel, using averaging, and rounding. The green border marks the first valid average bpm. Note that it uses the current average heart rate for the current computation as well - indexed 16-th. Due to visibility issues, values from the simulation are scattered across two figures. Observe the signals in magenta.

Figure 6.5: Average BPM for validation figure 1

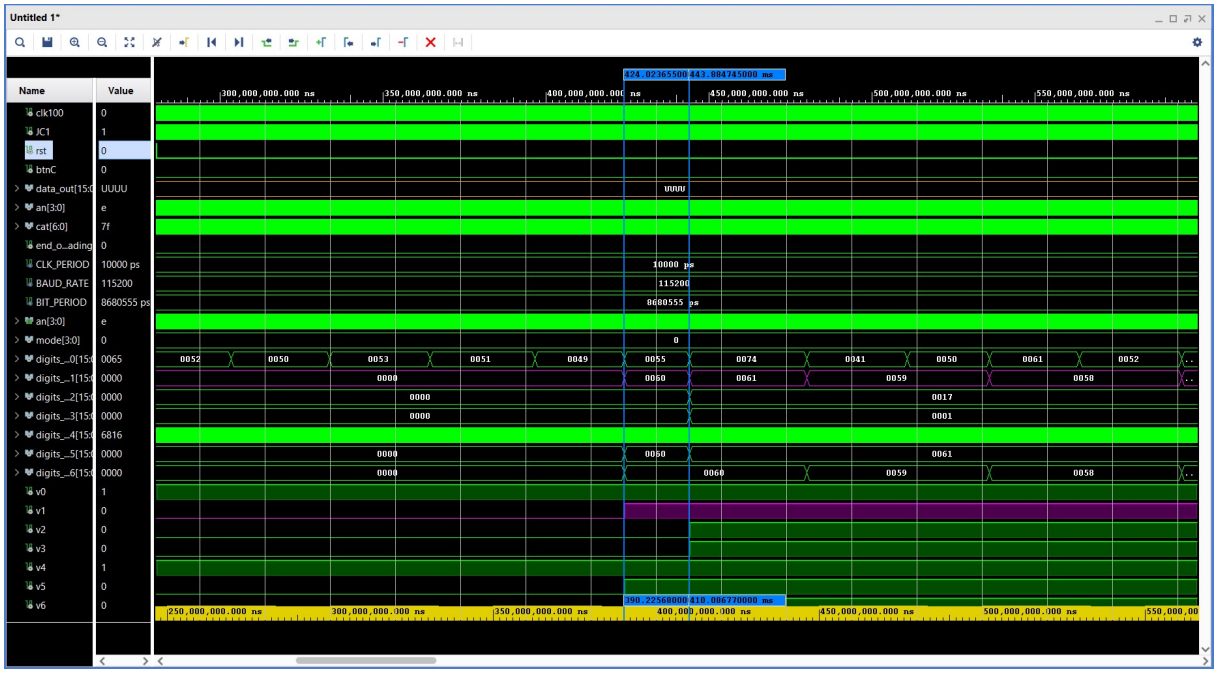

Figure 6.6: Average BPM for validation figure 2

Observe the computation of minimum and maximum average heart rates, note the pink signals.

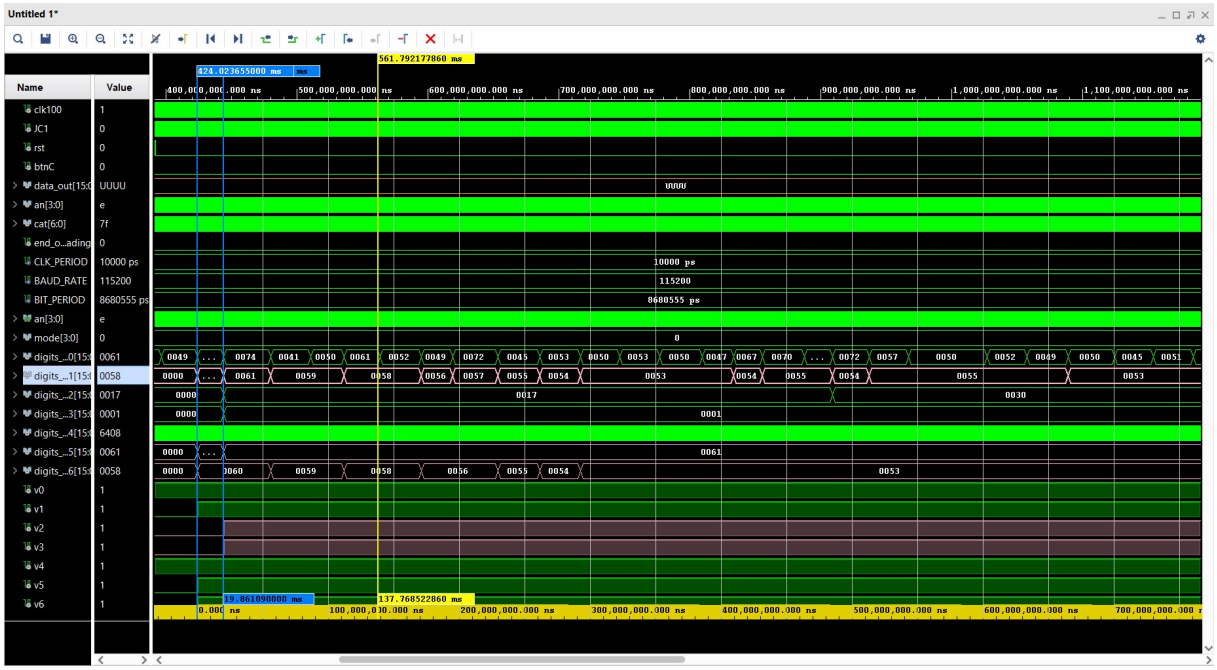

Figure 6.7: Minimum and maximum average heart rate validation

Listing 6.1: Testbench: tb_top_level_module2.vhd

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.MATH_REAL.ALL;
use STD.TEXTIO.ALL;

entity tb_top_module2 is
end tb_top_module2;

architecture sim of tb_top_module2 is

constant CLK_PERIOD : time := 10 ns;          -- 100 MHz
constant BAUD_RATE  : integer := 115200;
constant BIT_PERIOD : time := 1 sec / BAUD_RATE;  -- ~8.68 us

signal clk100   : std_logic := '0';
signal JC1      : std_logic := '1';  -- UART idle high
signal rst      : std_logic := '0';
signal btnC     : std_logic := '0';
signal data_out : std_logic_vector(15 downto 0);
signal an       : std_logic_vector(3 downto 0);
signal cat      : std_logic_vector(6 downto 0);


-- Seeds for uniform() RNGenerator
shared variable seed1 : positive := 12;
shared variable seed2 : positive := 543;

```

```
29
30  signal end_of_reading : std_logic := '0';
31  procedure send_uart_byte(
32  signal line        : out std_logic;
33  constant data      : std_logic_vector(7 downto 0);
34  constant bit_period: time
35  ) is
36  begin
37  -- idle high
38  line <= '1';
39  wait for bit_period;
40
41  -- start bit
42  line <= '0';
43  wait for bit_period;
44
45  -- 8 data bits, LSB first
46  for i in 0 to 7 loop
47  line <= data(i);
48  wait for bit_period;
49  end loop;
50
51  -- stop bit
52  line <= '1';
53  wait for bit_period;
54  end procedure;
55
56
57  begin
58
59  clk_proc : process
60  begin
61  clk100 <= '0';
62  wait for CLK_PERIOD/2;
63  clk100 <= '1';
64  wait for CLK_PERIOD/2;
65  end process;
66
67  dut : entity work.top_level_module
68  port map (
69  clk100       => clk100,
70  JC1          => JC1,
71  rst          => rst,
72  btnC         => btnC,
73  an           => an,
74  cat          => cat
75  );
76
77  stim : process
78  file sensors_data : text open read_mode is "<INPUT YOUT ABSOLUTE PATH To FILE>";
79  variable in_line  : line;
80  variable ppg_val  : integer;
81  variable ppg_slv  : std_logic_vector(15 downto 0);
```

```
  variable lo_byte, hi_byte : std_logic_vector(7 downto 0);
  begin
  -- reset pulse
  rst <= '1';
  JC1 <= '1';
  wait for CLK_PERIOD;
  rst <= '0';

  wait for 100 * BIT_PERIOD;

  if not endfile(sensors_data) then
  readline(sensors_data, in_line);
  -- no read() here, it's just the header text
  end if;

  -- read until EOF
  while not endfile(sensors_data) loop
  -- Get next line from file
  readline(sensors_data, in_line);

  -- parses integer from the line and assumes each line is just a number
  read(in_line, ppg_val);

  ppg_slv := std_logic_vector(to_unsigned(ppg_val, 16));

  hi_byte := ppg_slv(15 downto 8);
  lo_byte := ppg_slv(7 downto 0);

  -- send over UART
  send_uart_byte(JC1, hi_byte, BIT_PERIOD);
  wait for 2 * BIT_PERIOD;
  send_uart_byte(JC1, lo_byte, BIT_PERIOD);

  wait for 2 * BIT_PERIOD;
  end loop;

  wait for 10 * CLK_PERIOD;

  report "End of simulation" severity note;
  wait;
  end process;


  end sim;

```

For full project code and other resources, insights please visit: https://github.com/K4t4sztrof4/FPGA_Portfolio/tree/main/Sensor_data_proc_w_AXI4-Stream.

Bibliography

[1] Analog Devices. *MAX30100 Pulse Oximeter and Heart-Rate Sensor IC*, 2014. Datasheet.

[2] ARM Ltd. AMBA AXI Protocol Specification, Rev. 4.0. `https://developer.arm.com/documentation/ihi0022/latest`, 2010. Accessed: 2025-10-22.

[3] Nicolò Campanini and Salvatore Torsell. Design and optimization of an fpga-based system for real-time human stress level assessment. Master's thesis, Politecnico di Milano, 2023. Accessed: 2025-10-22.

[4] Hui Gao, Houde Dai, and Yadan Zeng. High-speed image processing and data transmission based on vivado hls and axi4-stream interface. In *2018 IEEE International Conference on Information and Automation (ICIA)*, pages 575–579, 2018.

[5] Qianyu Guan and Bing Yang. Design and implementation of fpga-based efficient simulation platform for mixed-signal systems. In *2025 2nd International Conference on Digital Media, Communication and Information Systems (DMCIS)*, pages 01–07, 2025.

[6] Rohith Raj Ram Prakash. Fpga-based implementation of lane detection and tracking algorithms for autonomous driving. KTH, School of Electrical Engineering and Computer Science (EECS). Available at `https://www.diva-portal.org/smash/get/diva2:1429726/FULLTEXT01.pdf`, 2020. Accessed: 2025-10-22.

[7] D. Singh and R. Chandel. Fpga-based hardware-accelerated design of linear prediction analysis for real-time speech signal. *Arabian Journal for Science and Engineering*, 48:14927–14941, 2023.

[8] K. D. Torp, P. Modi, E. J. Pollard, et al. *Pulse Oximetry*. StatPearls Publishing, 2025. Updated 2023 Jul 30.

[9] Antonio Velarte, Antonio Castel, Aranzazu Otin, and Esther Pueyo. A softcore processor-based system for electrophysiological signal acquisition. Available at SSRN: `https://ssrn.com/abstract=5063470` or `http://dx.doi.org/10.2139/ssrn.5063470`, 2023. Accessed: 2025-10-22.

[10] Nicholas Zambetti, Karl Söderby, and Jacob Hylén. Inter-integrated circuit (i2c) protocol. Arduino Docs platform.