Circuit for processing signals

Student: Popan Razvan-Calin

Lab Professor: Dragos Bogdan Lazea

Course Professor: Gheorghe Sebestyen

Date: 21/10/2024

Contents

1 Introduction **3**
- 1.1 Context . . . 3
- 1.2 Objectives . . . 3

2 Bibliographic Research **4**
- 2.1 AXI4-Stream Protocol . . . 4
- 2.2 Low-Pass Filter in Digital Signal Processing . . . 4
- 2.3 Kalman Filter . . . 6

3 Analysis **7**
- 3.1 Project Proposal . . . 7
- 3.2 Project Analysis & Design . . . 7
  - 3.2.1 Photoresistor.Illuminance.Luminous Flux . . . 7
  - 3.2.2 Microcontroller . . . 7
  - 3.2.3 Field Programmable Gate Array . . . 8
  - 3.2.4 Kalman Filter . . . 8
  - 3.2.5 Kalman using AXI4-Stream Communication Protocol . . . 10

4 Implementation **11**
- 4.1 AXI4-Stream Basic Operation Modules . . . 11
  - 4.1.1 AXI4-Stream Adder/Subtractor . . . 11
  - 4.1.2 AXI4-Stream Multiplier . . . 12
  - 4.1.3 AXI4-Stream Remainder Module . . . 13
  - 4.1.4 AXI4-Stream Vector Addition/Subtraction . . . 14
  - 4.1.5 AXI4-Stream Matrix Addition/Subtraction . . . 15
  - 4.1.6 AXI4-Stream Matrix Multiplication . . . 15
  - 4.1.7 AXI4-Stream Matrix Multiplication with Column Vector 15
  - 4.1.8 AXI4-Stream Matrix Multiplication with Row Vector . . . 16
  - 4.1.9 AXI4-Stream Dot Product . . . 16
  - 4.1.10 AXI4-Stream Column Vector Multiplication with Row Vector . . . 17
  - 4.1.11 AXI4-Stream Vector Data FIFO with initial Value . . . 18
  - 4.1.12 AXI4-Stream Matrix Data FIFO with initial Value . . . 19
- 4.2 Kalman Filter VHDL Implementation . . . 20
  - 4.2.1 Predict State . . . 20
  - 4.2.2 Estimate State . . . 21
  - 4.2.3 Computer Error Covariance State . . . 22
  - 4.2.4 Update State . . . 22
- 4.3 UART Communication . . . 23
- 4.4 Top Level . . . 25
- 4.5 Photoresistor Circuit Implementation & Arduino Code . . . 27

5 Testing and Validation **29**
5.1 Arduino Code and Photoresistor Testing 29
5.2 AXI4-Stream Components Testing 30
5.2.1 AXI4-Stream Vector Addition/Subtraction Testing 30
5.2.2 AXI4-Stream Dot Product Testing 30
5.2.3 AXI4-Stream Matrix Addition/Subtraction Testing 31
5.2.4 AXI4-Stream Matrix Multiplication Testing 32
5.2.5 AXI4-Stream Matrix Column Vector Multiplication Testing 32
5.3 UART Testbench 33
5.4 UART Communication Testbench 34
5.5 Kalman Filter Testing & Results 36

1 Introduction

1.1 Context

The aim of this project is to design a circuit for calculating real-time statistics of incoming data from sensors connected to a microcontroller. Microcontrollers are well-suited for managing multiple sensor connections, while Field Programmable Gate Arrays (FPGAs) excel at processing the incoming data streams. FPGAs operate at a higher clock rate than microcontrollers and can process multiple data streams simultaneously. This capability is particularly useful for applications that integrate with machine learning algorithms, require real-time processing of sensor data, or need rapid result generation.

1.2 Objectives

The objective of this project is to design a circuit in VHDL capable of filtering data using a Kalman filter. The data, received from a microcontroller originates from sensors connected to the microcontroller. These sensors may include ultrasound, temperature, pressure, humidity, or light sensors. To achieve this objective, the following steps are required:

1. Implement and test the AXI4-Stream communication protocol.
2. Develop and test a filter for sensor data.
3. Connect a sensor to the microcontroller, send data to the FPGA
4. Connect a sensor to the microcontroller and verify sensor data handling.
5. Integrate all components to achieve the project's overall objective.

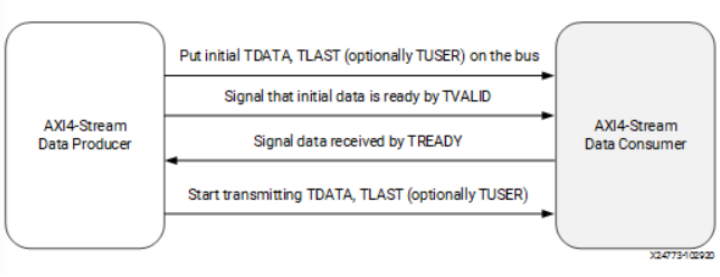
2 Bibliographic Research

2.1 AXI4-Stream Protocol

The AXI4-Stream protocol is used for transmitting unidirectional data of arbitrary width, following a consumer-producer pattern. In this communication, two entities are involved: a receiver (or consumer), which receives the data, and a sender (or producer), which sends the data.

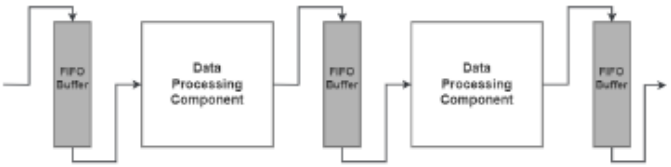
Data transfer initiates when the sender asserts the `TVALID` signal, indicating valid data on the bus. The receiver then responds with the `TREADY` signal once it is ready to accept data. When the producer asserts `TVALID`, it begins transmitting data via the `TDATA` signal, and uses the `TLAST` signal to indicate the last byte of data in the transfer. The consumer continues to read the `TDATA` signal until it detects the `TLAST` signal, which marks the end of the data stream.

The protocol also includes optional features that enhance functionality. For instance, the `TKEEP` and `TSTRB` signals allow for the multiplexing of data positions and the data itself on the `TDATA` signal. Additionally, the `TID` and `TDST` signals provide stream routing capabilities, with `TID` representing the stream identifier and `TDST` representing the stream destination identifier [1].



[1]

Figure 1: AXI4-Stream protocol



[5]

Figure 2: AXI4-Stream Communication between different modules

2.2 Low-Pass Filter in Digital Signal Processing

A low-pass filter is a fundamental component in digital signal processing (DSP) used to allow signals with frequencies lower than a specified cutoff frequency to

pass through while attenuating signals with higher frequencies. This filtering process is crucial in applications where high-frequency noise or interference needs to be removed from a signal, leaving the desired low-frequency components intact.

In digital implementations, a low-pass filter can be designed using various techniques, such as:

- **Finite Impulse Response (FIR) Filters**: FIR filters are characterized by their finite duration response to an impulse input, meaning the output signal eventually returns to zero. FIR low-pass filters are inherently stable and can be designed to have a linear phase response, making them ideal for preserving the waveform shape of the filtered signal.
- **Infinite Impulse Response (IIR) Filters**: IIR filters have a recursive structure, meaning they use previous output values in their calculations, resulting in an infinite-duration response to an impulse. While IIR filters can achieve sharper cutoff characteristics with fewer coefficients compared to FIR filters, they may introduce phase distortion and are potentially less stable.

Low-pass filters are widely used in applications such as:

- *Audio Processing*: To remove high-frequency noise from audio signals, enhancing clarity.
- *Image Processing*: For smoothing images by removing high-frequency details.
- *Data Communication*: For limiting bandwidth to reduce interference in transmitted signals.

Digital low-pass filters can be implemented using software algorithms on processors or hardware circuits, such as Field Programmable Gate Arrays (FPGAs).The equation of the first-order low-pass filter is:

$$\hat{x}_k = \alpha\, \hat{x}_{k-1} + (1 - \alpha)\, \vec{x}_{k-1} \tag{1}$$

Where $\hat{x}_k$ represents the current estimate at time k, $\hat{x}_{k-1}$ represents the previous estimate and $\vec{x}_k$ is the value read by the sensor at time k. $\alpha \in [0, 1]$ is a free variable which can be tuned in order to change how the final signal will look like. By giving α a value closer to 1, the resulting signal will take in consideration the previous estimates rather than the value given by the sensor, and the curve will be smoother, otherwise if the value is closer to 0 the pervious estimates won't be taken into consideration and the signal will look almost like the value of the signal given by the sensor (with noise).

2.3 Kalman Filter

The Kalman Filter tries to solve the problem of tuning the α parameter in the low-pass filter by calculating the α value at each step by splitting the process into 4 steps:

- *Prediction step*: predict the estimate state and the covariance matrix based on the previous estimate state and covariance matrix
- *Estimation step*: Compute the Kalman Gain(α) and the estimate state
- *Compute Error Covariance step*:Compute the covariance matrix
- *Update step*: update the current estimate state with the new one and the current covariance matrix with the calculated one

3 Analysis

3.1 Project Proposal

The final circuit will contain the following:

- A photoresistor
- A microcontroller used to send the data of the sensor to the FPGA
- A FPGA where the Kalman filter will be implemented and the different components of the Kalman filter will comunicate with eachother through the AXI4-Stream Protocol
- A 7-segment display where the estimate of the measurement will be displayed

3.2 Project Analysis & Design

3.2.1 Photoresistor.Illuminance.Luminous Flux

The sensor that will be used will be a photoresistor, it decreases the resistance with the increase of luminosity on the sensitive part of the photoresistor. With the value read from the photoresistor we can calculate the luminous flux (perceived power of visible light, measured in lumen (lm)) and the illuminance (total luminous flux incident on a area per unit of area,measured in lux).

$$E_v = K(\frac{1}{R_{LDR}})^n(\text{lux}) \tag{2}$$

$$\phi_v = E_v \times A(\text{lm}) \tag{3}$$

Where ϕ_v is the luminous flux, E_v is the illuminance, A is the area and R_{LDR} is the resistance of the photoresistor.

3.2.2 Microcontroller

The microcontroller used will be an Arduino microcontroller which is cheap and has the necessary components for this project. The photoresistor will be connected to an analog pin on the Arduino board, in the program the, since the microcontroller uses a 10 bit ADC we will have to convert the voltage in the range $[0, 5]$V, then the voltage has to converted into resistance using the formula:

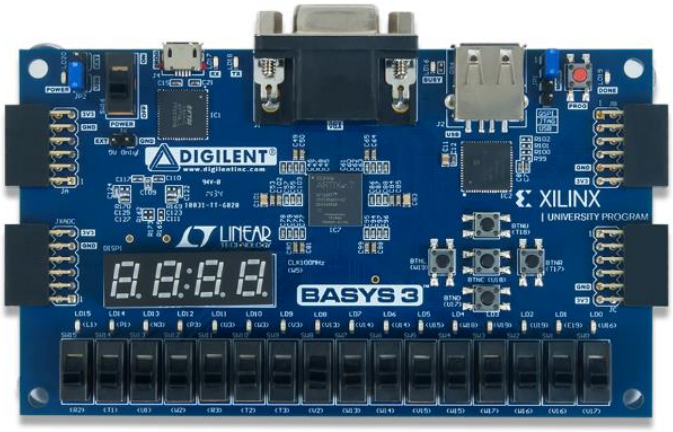
$$R_{LDR} = \frac{R \times (5 - V_{LDR})}{1023}(\Omega) \tag{4}$$

R is the resistance of the resistor connected in series with the photoresistor (10k Ω).Using this value and equation (2), and setting the K and n values based on the photoresistor, we can calculate the illuminance and luminous flux. For the sake of simplicity we assume that $n = 1$ and $K = 500 \times 1000$ (in the FPGA). The R_{LDR} value will be sent to the FPGA for filtering.

3.2.3 Field Programmable Gate Array

The FPGA that will be used in this project will be a Basys 3 Artix-7 that has the following features:

- 4-digit 7-segment display to display the results of the filter
- USB-UART Bridge
- 33,280 logic cells in 5200 slices (each slice contains four 6-input LUTs and 8 flip-flops)
- 1,800 Kbits of fast block RAM
- Internal clock speeds exceeding 450 MHz



[10]

Figure 3: Basys 3 Artix-7

The Basys 3 will be used to implement the Kalman Filter, using VHDL (VHSIC Hardware Description Language) and to implement the AXI4-Stream protocol between the stages of the Kalman Filter.

3.2.4 Kalman Filter

As described in the Bibliographic research section, the Kalman filter is like a low-pass filter, which has the equation (1), but the α value is computed at each new measurement received. To understand how the Kalman filter works, first we have to clarify some notations used.

K_k – Kalman Gain of size nxm (5)

$\hat{x}_k$ – estimate state,which is a column vector of size n-what other features we want to find (6)

P_k – estimate covariance matrix of size nxn (7)

$\hat{x}_k^-$ – predicted estimate state, column vector of size n (8)

P_k^- – predicted covariance matrix of size nxn (9)

z_k – measurement, which is a column vector of size m-how many features (10)

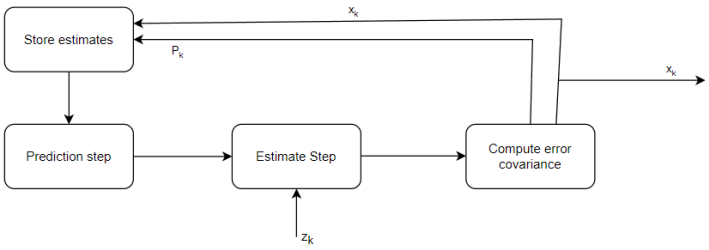
A – a matrix related to the best estimate of a linear model,of size nxn (11)

Q – related to the noise of the process,of size nxn (12)

R – related to the noise of the measurement of size mxm (13)

H – related to how the measurements is related to the state,of size mxn (14)

By splitting the Kalman process into more steps we can increase the throughput and make the filter process signals faster. The Kalman filter will act as a pipeline where each stage is one of the stages in the Kalman Process. The stages



[2]

Figure 4: Kalman filter Process

in the kalman filter and the computations done at each step.

- *Prediction step*: here we calculate the predicted estimate state and the predicted error covariance matrix using the formulas:
  $\hat{x}_k^- = A \times \hat{x}_{k-1}$
  $P_k^- = A \times P_{k-1} \times A^T + Q$
- *Estimation Step*: in this step the Kalman Gain and the estimate state are computed
  $K_k = P_k^- \times H^T \times (H \times P_k^- \times H^T + R)^{-1}$
  $\hat{x}_k = \hat{x}_k^- + K_k \times (z_k - H \times \hat{x}_k^-)$
- *Compute error covariance*: $P_k = P_k^- - K_k \times H \times P_k^-$

- *Update the new estimates*: $\hat{x}_{k+1} \leftarrow \hat{x}_k$, $P_{k+1} \leftarrow P_k$

The A,H,Q,R represent the system model matrices. For the Kalman filter a linear state model is assumed and that the estimate state $x_k \sim \mathcal{N}(\hat{x}_k, P_k)$:
$x_{k+1} = A \times x_k + w_k$
$z_{k+1} = H \times x_k + v_k$
Where w_k is the process noise and v_k is the measurement noise, and both are assumed to belong to a Gaussian Distribution. Q and R are diagonal matrices related the noise of the process and the noise of the measurement, where the elements on the diagonal are the variance of the noises.

$$Q = \begin{bmatrix} \sigma_1^2 & 0 & \dots & 0 \\ 0 & \sigma_2^2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \sigma_n^2 \end{bmatrix}, w_k = \begin{bmatrix} w_k^1 \\ w_k^2 \\ \vdots \\ w_n^1 \end{bmatrix}, R = \begin{bmatrix} \sigma_1^2 & 0 & \dots & 0 \\ 0 & \sigma_2^2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \sigma_m^2 \end{bmatrix}, v_k = \begin{bmatrix} v_k^1 \\ v_k^2 \\ \vdots \\ v_n^1 \end{bmatrix}$$

The Kalman filter has to have the values $\hat{x}_0$,P_0,A,H,Q,R initialized this is done if there is more knowledge about the process or done by trial and error.

3.2.5 Kalman using AXI4-Stream Communication Protocol

The AXI4-Stream protocol can be used for communication between different modules, this improves the thorughput of the circuit since we can split the kalman filter into different modules, each communicating with the next module by sending the data into the FIFO buffer, and the module will accept data when it can by sending the TREADY signal.

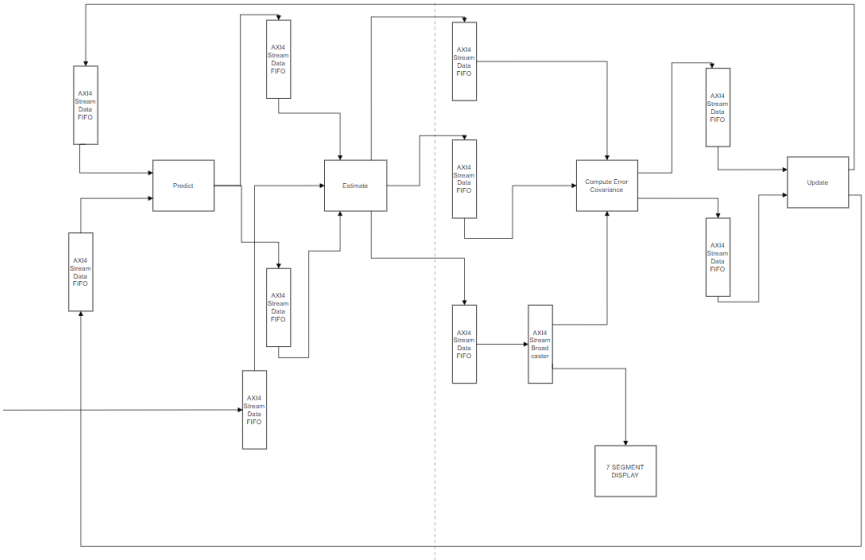


Figure 5: Kalman Filter Modules communicating using AXI4-Stream Protocol

4 Implementation

4.1 AXI4-Stream Basic Operation Modules

In the following section will be presented the basic modules that will be used to implement the steps of the Kalman Filter. The modules are: AXI4-Stream Add/Subtract on 64 bits, AXI4-Stream Multiply on 64 bits, and AXI4-Stream Remainder on 64 bits. With these 3 blocks, we will implement the modules to compute operations on vector and matrices, using a structural design approach to improve scalability (if needed) and efficiency, since operations are done in parallel rather than in a procedural manner (using a process). The operations on matrices that will be needed for the Kalman Filter implementation will be: dot product of two vectors, product between a column vector and a row vector, product between a matrix and a column vector, product between a row vector and a matrix, product between two matrices, addition/subtraction for vectors and addition/subtraction for matrices.

4.1.1 AXI4-Stream Adder/Subtractor

The entity declaration of the Adder/Subtracter module, it follows the AXI4-Stream design approach since each signal coming from the "slave" has a TVALID and TDATA signal, and they receive the TREADY signal form this module. Also the "master" receives the processed data and the valid signal, and this module gets the ready signal from the master and transitions form the read to write state when the inputs are valid, and form th write to read state when the "master" is ready to accept the processed data.

```
1       library IEEE;
2   use IEEE.STD_LOGIC_1164.ALL;
3   use IEEE.numeric_std.all;
4
5   entity AXI4_Adder_Subtracter is
6   port(
7       aclk: in std_logic;
8       aresetn: in std_logic;
9       s_axis_a_tdata:in std_logic_vector(63 downto 0);
10      s_axis_b_tdata:in std_logic_vector(63 downto 0);
11      s_axis_op_tdata:in std_logic;
12      s_axis_a_tready:out std_logic;
13      s_axis_b_tready:out std_logic;
14      s_axis_op_tready:out std_logic;
15      s_axis_a_tvalid:in std_logic;
16      s_axis_b_tvalid:in std_logic;
17      s_axis_op_tvalid:in std_logic;
18      m_axis_res_tdata: out std_logic_vector(63 downto 0);
19      m_axis_res_tready: in std_logic;
20      m_axis_res_tvalid: out std_logic
21  );
22  end AXI4_Adder_Subtracter;
```

In the code below we can see the module followin the AXI4-Stream design specification, with the op signal choosing which operation to be performed.

```
architecture Behavioral of AXI4_Adder_Subtracter is

type state_type is (READ,WRITE);
signal state: state_type := READ;

signal res_valid : STD_LOGIC := '0';
signal result : STD_LOGIC_VECTOR (63 downto 0) := (others => '0');
signal a_ready, b_ready, op_ready : STD_LOGIC := '0';
signal internal_ready, external_ready, inputs_valid : STD_LOGIC := '0';

begin

s_axis_a_tready <= external_ready;
s_axis_b_tready <= external_ready;
s_axis_op_tready <= external_ready;

internal_ready <= '1' when state = READ else '0';
inputs_valid <= s_axis_a_tvalid and s_axis_b_tvalid and s_axis_op_tvalid;
external_ready <= internal_ready and inputs_valid;

m_axis_res_tvalid <= '1' when state = WRITE else '0';
m_axis_res_tdata <= result;

process(aclk)
begin
    if aclk'event and aclk='1' then
          case state is
            when READ=>
                if external_ready = '1' and inputs_valid='1' then
                    if s_axis_op_tdata='1' then
                        result <= std_logic_vector(signed(s_axis_a_tdata) - signed(s_axis_b_tdata));
                    elsif s_axis_op_tdata='0' then
                        result <= std_logic_vector(signed(s_axis_a_tdata) + signed(s_axis_b_tdata));
                    end if;
                    state <= WRITE;
                end if;
            when WRITE=>
                if m_axis_res_tready = '1' then
                    state<=READ;
                end if;
          end case;
     end if;
end process;
end Behavioral;
```

4.1.2 AXI4-Stream Multiplier

The code below represents the AXI4-Stream odule of multiplying two 64 bits numbers, and the result is also on 64 bit. This is done in orther to make the implementation easier and since the values to do not pass the 64bit signed maximum value in our case.

```
    library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;

entity AXI4_Multiplier is
port(
    aclk: in std_logic;
    aresetn: in std_logic;
    s_axis_a_tdata:in std_logic_vector(63 downto 0);
    s_axis_b_tdata:in std_logic_vector(63 downto 0);
    s_axis_a_tready:out std_logic;
    s_axis_b_tready:out std_logic;
    s_axis_a_tvalid:in std_logic;
    s_axis_b_tvalid:in std_logic;
    m_axis_res_tdata: out std_logic_vector(63 downto 0);
    m_axis_res_tready: in std_logic;
    m_axis_res_tvalid: out std_logic
);
end AXI4_Multiplier;
```

```
architecture Behavioral of AXI4_Multiplier is

type state_type is (READ,WRITE);
signal state: state_type := READ;

signal res_valid : STD_LOGIC := '0';
signal result : STD_LOGIC_VECTOR (127 downto 0) := (others => '0');
signal a_ready, b_ready, op_ready : STD_LOGIC := '0';
signal internal_ready, external_ready, inputs_valid : STD_LOGIC := '0';

begin

s_axis_a_tready <= external_ready;
s_axis_b_tready <= external_ready;

internal_ready <= '1' when state = READ else '0';
inputs_valid <= s_axis_a_tvalid and s_axis_b_tvalid;
external_ready <= internal_ready and inputs_valid;

m_axis_res_tvalid <= '1' when state = WRITE else '0';
m_axis_res_tdata <= result(63 downto 0);

process(aclk)
begin
    if aclk'event and aclk='1' then
          case state is
            when READ=>
                if external_ready = '1' and inputs_valid='1' then
                    result <=std_logic_vector(signed(s_axis_a_tdata) * signed(s_axis_b_tdata));
                    state <= WRITE;
                end if;
            when WRITE=>
                if m_axis_res_tready = '1' then
                    state<=READ;
                end if;
          end case;
     end if;
end process;

end Behavioral;
```

4.1.3 AXI4-Stream Remainder Module

The remainder operaion will be needed when we compute the Kalman Gain, to divide each element of the vector with β

```
    library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.ALL;

entity AXI4_Remainder is
port(
    aclk: in std_logic;
    areset: in std_logic;
    s_axis_a_tdata:in std_logic_vector(63 downto 0);
    s_axis_b_tdata:in std_logic_vector(63 downto 0);
    s_axis_a_tready:out std_logic;
    s_axis_b_tready:out std_logic;
    s_axis_a_tvalid:in std_logic;
    s_axis_b_tvalid:in std_logic;
    m_axis_res_tdata: out std_logic_vector(63 downto 0);
    m_axis_res_tready: in std_logic;
    m_axis_res_tvalid: out std_logic
);
end AXI4_Remainder;
```

The code of the AXI4-Stream Remainder Module.

```
architecture Behavioral of AXI4_Remainder is

type state_type is (READ,WRITE);
signal state: state_type := READ;

signal res_valid : STD_LOGIC := '0';
signal result : STD_LOGIC_VECTOR (63 downto 0) := (others => '0');
signal a_ready, b_ready, op_ready : STD_LOGIC := '0';
signal internal_ready, external_ready, inputs_valid : STD_LOGIC := '0';

begin

s_axis_a_tready <= external_ready;
s_axis_b_tready <= external_ready;

internal_ready <= '1' when state = READ else '0';
inputs_valid <= s_axis_a_tvalid and s_axis_b_tvalid;
external_ready <= internal_ready and inputs_valid;

m_axis_res_tvalid <= '1' when state = WRITE else '0';
m_axis_res_tdata <= result;

process(aclk)
begin
    if aclk'event and aclk='1' then
          case state is
            when READ=>
                if external_ready = '1' and inputs_valid='1' then
                    result <=std_logic_vector(signed(s_axis_a_tdata) rem signed(s_axis_b_tdata));
                    state <= WRITE;
                end if;
            when WRITE=>
                if m_axis_res_tready = '1' then
                    state<=READ;
                end if;
          end case;
     end if;
end process;
end Behavioral;
```

4.1.4 AXI4-Stream Vector Addition/Subtraction

This module computes the addition/subtraction of two vectors using two AXI4-Stream Addition/Subtraction modules:

```
    entity AXI4_Vect_Add_Subtract is
   Port (
    aclk: in std_logic;
    aresetn: in std_logic;
    s_axis_vect0_tdata: in std_logic_vector(127 downto 0);
    s_axis_vect1_tdata: in std_logic_vector(127 downto 0);
    s_axis_op_tdata: in std_logic;
    s_axis_vect0_tready: out std_logic;
    s_axis_vect1_tready: out std_logic;
    s_axis_op_tready: out std_logic;
    s_axis_vect0_tvalid: in std_logic;
    s_axis_vect1_tvalid: in std_logic;
    s_axis_op_tvalid: in std_logic;
    m_axis_vect_tdata: out std_logic_vector(127 downto 0);
    m_axis_vect_tready: in std_logic;
    m_axis_vect_tvalid: out std_logic
  );
end AXI4_Vect_Add_Subtract;
```

The code follows the formula

$$\begin{bmatrix} x_0 \\ x_1 \end{bmatrix} \pm \begin{bmatrix} y_0 \\ y_1 \end{bmatrix} = \begin{bmatrix} x_0 \pm y_0 \\ x_1 \pm y_1 \end{bmatrix} \tag{15}$$

4.1.5 AXI4-Stream Matrix Addition/Subtraction

This module computes the addition/subtraction of two matrices using four AXI4-Stream Addition/Subtraction modules:

```
    entity AXI4_Matrix_Add_Subtract is
  Port (
    aclk: in std_logic;
    aresetn: in std_logic;
    s_axis_matrix0_tdata: in std_logic_vector(255 downto 0);
    s_axis_matrix1_tdata: in std_logic_vector(255 downto 0);
    s_axis_op_tdata: in std_logic;
    s_axis_matrix0_tready: out std_logic;
    s_axis_matrix1_tready: out std_logic;
    s_axis_op_tready: out std_logic;
    s_axis_matrix0_tvalid: in std_logic;
    s_axis_matrix1_tvalid: in std_logic;
    s_axis_op_tvalid:in std_logic;
    m_axis_matrix_tdata: out std_logic_vector(255 downto 0);
    m_axis_matrix_tready: in std_logic;
    m_axis_matrix_tvalid: out std_logic
  );
end AXI4_Matrix_Add_Subtract;
```

The code follows the formula:

$$\begin{bmatrix} x_0 & x_1 \\ x_2 & x_3 \end{bmatrix} \pm \begin{bmatrix} y_0 & y_1 \\ y_2 & y_3 \end{bmatrix} = \begin{bmatrix} x_0 \pm y_0 & x_1 \pm y_1 \\ x_2 \pm y_2 & x_3 \pm y_3 \end{bmatrix} \tag{16}$$

4.1.6 AXI4-Stream Matrix Multiplication

This module computes the multiplication of two matrices using eight AXI4-Stream Multipliers and four AXI4-Stream Addition/Subtraction modules:

```
    entity AXI4_Mult_Matrix_Matrix is
  Port (
    aclk: in std_logic;
    aresetn: in std_logic;
    s_axis_matrix0_tdata: in std_logic_vector(255 downto 0);
    s_axis_matrix1_tdata: in std_logic_vector(255 downto 0);
    s_axis_matrix0_tready: out std_logic;
    s_axis_matrix1_tready: out std_logic;
    s_axis_matrix0_tvalid: in std_logic;
    s_axis_matrix1_tvalid: in std_logic;
    m_axis_matrix_tdata: out std_logic_vector(255 downto 0);
    m_axis_matrix_tready: in std_logic;
    m_axis_matrix_tvalid: out std_logic
  );
end AXI4_Mult_Matrix_Matrix;
```

The code follows the formula:

$$\begin{bmatrix} x_0 & x_1 \\ x_2 & x_3 \end{bmatrix} \times \begin{bmatrix} y_0 & y_1 \\ y_2 & y_3 \end{bmatrix} = \begin{bmatrix} x_0 \times y_0 + x_1 \times y_2 & x_0 \times y_1 + x_1 \times y_3 \\ x_2 \times y_0 + x_3 \times y_2 & x_2 \times y_1 + x_3 \times y_3 \end{bmatrix} \tag{17}$$

4.1.7 AXI4-Stream Matrix Multiplication with Column Vector

This module computes the multiplication of a matrix and a column vector using four AXI4-Stream Multipliers and two AXI4-Stream Addition/Subtraction modules. The code follows the formula:

$$\begin{bmatrix} x_0 & x_1 \\ x_2 & x_3 \end{bmatrix} \times \begin{bmatrix} y_0 \\ y_1 \end{bmatrix} = \begin{bmatrix} x_0 \times y_0 + x_1 \times y_1 \\ x_2 \times y_0 + x_3 \times y_1 \end{bmatrix} \tag{18}$$

```
    entity AXI4_Mult_Matrix_ColumnVector is
  Port (
    aclk: in std_logic;
    aresetn: in std_logic;
    s_axis_vect_tdata: in std_logic_vector(127 downto 0);
    s_axis_matrix_tdata: in std_logic_vector(255 downto 0);
    s_axis_vect_tready: out std_logic;
    s_axis_matrix_tready: out std_logic;
    s_axis_vect_tvalid: in std_logic;
    s_axis_matrix_tvalid: in std_logic;
    m_axis_vect_tdata: out std_logic_vector(127 downto 0);
    m_axis_vect_tready: in std_logic;
    m_axis_vect_tvalid: out std_logic
  );
end AXI4_Mult_Matrix_ColumnVector;
```

4.1.8 AXI4-Stream Matrix Multiplication with Row Vector

This module computes the multiplication of row vector and a matrix using four AXI4-Stream Multipliers and two AXI4-Stream Addition/Subtraction modules:

```
    entity AXI4_Mult_RowVector_Matrix is
  Port (
    aclk: in std_logic;
    aresetn: in std_logic;
    s_axis_vect_tdata: in std_logic_vector(127 downto 0);
    s_axis_matrix_tdata: in std_logic_vector(255 downto 0);
    s_axis_vect_tready: out std_logic;
    s_axis_matrix_tready: out std_logic;
    s_axis_vect_tvalid: in std_logic;
    s_axis_matrix_tvalid: in std_logic;
    m_axis_vect_tdata: out std_logic_vector(127 downto 0);
    m_axis_vect_tready: in std_logic;
    m_axis_vect_tvalid: out std_logic
  );
end AXI4_Mult_RowVector_Matrix;
```

The code follows the formula:

$$\begin{bmatrix} x_0 & x_1 \end{bmatrix} \times \begin{bmatrix} y_0 & y_1 \\ y_2 & y_3 \end{bmatrix} = \begin{bmatrix} x_0 \times y_0 + x_1 \times y_2 & x_0 \times y_1 + x_1 \times y_3 \end{bmatrix} \tag{19}$$

4.1.9 AXI4-Stream Dot Product

This module computes the dot product between two vectors using two AXI4-Stream Multipliers and one AXI4-Stream Addition/Subtraction module:

```
    entity AXI4_Mult_Vect_Vect is
  Port (
    aclk: in std_logic;
    aresetn: in std_logic;
    s_axis_vect0_tdata: in std_logic_vector(127 downto 0);
    s_axis_vect1_tdata: in std_logic_vector(127 downto 0);
    s_axis_vect0_tready: out std_logic;
    s_axis_vect1_tready: out std_logic;
    s_axis_vect0_tvalid: in std_logic;
    s_axis_vect1_tvalid: in std_logic;
    m_axis_res_tdata: out std_logic_vector(63 downto 0);
    m_axis_res_tready: in std_logic;
    m_axis_res_tvalid: out std_logic
  );
end AXI4_Mult_Vect_Vect;
```

The code follows the formulas:

$$\begin{bmatrix} x_0 & x_1 \end{bmatrix} \times \begin{bmatrix} y_0 & y_1 \end{bmatrix} = x_0 \times y_0 + x_1 \times y_1 \tag{20}$$

$$\begin{bmatrix} x_0 \\ x_1 \end{bmatrix} \times \begin{bmatrix} y_0 \\ y_1 \end{bmatrix} = x_0 \times y_0 + x_1 \times y_1 \tag{21}$$

4.1.10 AXI4-Stream Column Vector Multiplication with Row Vector

This module computes the a column vector and a row vector using four AXI4-Stream Multipliers modules:

```
    entity AXI4_Mult_ColVect_RowVect is
  Port (
     aclk: in std_logic;
    aresetn: in std_logic;
    s_axis_vect0_tdata: in std_logic_vector(127 downto 0);
    s_axis_vect1_tdata: in std_logic_vector(127 downto 0);
    s_axis_vect0_tready: out std_logic;
    s_axis_vect1_tready: out std_logic;
    s_axis_vect0_tvalid: in std_logic;
    s_axis_vect1_tvalid: in std_logic;
    m_axis_matrix_tdata: out std_logic_vector(255 downto 0);
    m_axis_matrix_tready: in std_logic;
    m_axis_matrix_tvalid: out  std_logic
  );
end AXI4_Mult_ColVect_RowVect;
```

The code follows the formulas:

$$\begin{bmatrix} x_0 \\ x_1 \end{bmatrix} \times \begin{bmatrix} y_0 & y_1 \end{bmatrix} = \begin{bmatrix} x_0 \times y_0 & x_0 \times y_1 \\ x_1 \times y_0 & x_1 \times y_1 \end{bmatrix} \tag{22}$$

4.1.11 AXI4-Stream Vector Data FIFO with initial Value

This module is used for initializing the $\hat{x_0}$ estimate

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

entity AXI4_FIFO_Vect_With_InitialValue is
  PORT (
    s_axis_aresetn : IN STD_LOGIC;
    s_axis_aclk : IN STD_LOGIC;
    s_axis_tvalid : IN STD_LOGIC;
    s_axis_tready : OUT STD_LOGIC;
    s_axis_tdata : IN STD_LOGIC_VECTOR(127 DOWNTO 0);
    m_axis_tvalid : OUT STD_LOGIC;
    m_axis_tready : IN STD_LOGIC;
    m_axis_tdata : OUT STD_LOGIC_VECTOR(127 DOWNTO 0)
  );
end AXI4_FIFO_Vect_With_InitialValue;

architecture Behavioral of AXI4_FIFO_Vect_With_InitialValue is

COMPONENT AXI4_Vect_FIFO
  PORT (
    s_axis_aresetn : IN STD_LOGIC;
    s_axis_aclk : IN STD_LOGIC;
    s_axis_tvalid : IN STD_LOGIC;
    s_axis_tready : OUT STD_LOGIC;
    s_axis_tdata : IN STD_LOGIC_VECTOR(127 DOWNTO 0);
    m_axis_tvalid : OUT STD_LOGIC;
    m_axis_tready : IN STD_LOGIC;
    m_axis_tdata : OUT STD_LOGIC_VECTOR(127 DOWNTO 0)
  );
END COMPONENT;

signal sig_axis_tvalid,sig_axis_tready :  STD_LOGIC:='0';
signal sig_axis_tdata :  STD_LOGIC_VECTOR(127 DOWNTO 0):=(others=>'0');

begin

fifo : AXI4_Vect_FIFO PORT MAP
(
    s_axis_aresetn => s_axis_aresetn,
    s_axis_aclk => s_axis_aclk,
    s_axis_tvalid => sig_axis_tvalid,
    s_axis_tready => sig_axis_tready,
    s_axis_tdata => sig_axis_tdata,
    m_axis_tvalid => m_axis_tvalid,
    m_axis_tready => m_axis_tready,
    m_axis_tdata => m_axis_tdata
  );

s_axis_tready<=sig_axis_tready;

process(s_axis_aclk)
variable first:std_logic:='0';
begin
    if s_axis_aclk'event and s_axis_aclk='1' then
        if s_axis_aresetn='0' then
            first:='0';
            sig_axis_tvalid<='0';
            sig_axis_tdata<=(others =>'0');
        elsif s_axis_aresetn='1' and first='0' and sig_axis_tready='1' then
            sig_axis_tvalid<='1';
            sig_axis_tdata<=(others =>'0');
            first:='1';
        elsif first='1' then
            sig_axis_tvalid<=s_axis_tvalid;
            sig_axis_tdata <=s_axis_tdata;
        end if;
    end if;
end process;
end Behavioral;
```

4.1.12 AXI4-Stream Matrix Data FIFO with initial Value

This module is used for initializing the P_0 covariance matrix:

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

entity AXI4_FIFO_Matrix_With_InitialValue is
  PORT (
    s_axis_aresetn : IN STD_LOGIC;
    s_axis_aclk : IN STD_LOGIC;
    s_axis_tvalid : IN STD_LOGIC;
    s_axis_tready : OUT STD_LOGIC;
    s_axis_tdata : IN STD_LOGIC_VECTOR(255 DOWNTO 0);
    m_axis_tvalid : OUT STD_LOGIC;
    m_axis_tready : IN STD_LOGIC;
    m_axis_tdata : OUT STD_LOGIC_VECTOR(255 DOWNTO 0)
  );
end AXI4_FIFO_Matrix_With_InitialValue;

architecture Behavioral of AXI4_FIFO_Matrix_With_InitialValue is

COMPONENT AXI4_Matrix_FIFO
  PORT (
    s_axis_aresetn : IN STD_LOGIC;
    s_axis_aclk : IN STD_LOGIC;
    s_axis_tvalid : IN STD_LOGIC;
    s_axis_tready : OUT STD_LOGIC;
    s_axis_tdata : IN STD_LOGIC_VECTOR(255 DOWNTO 0);
    m_axis_tvalid : OUT STD_LOGIC;
    m_axis_tready : IN STD_LOGIC;
    m_axis_tdata : OUT STD_LOGIC_VECTOR(255 DOWNTO 0)
  );
END COMPONENT;

signal sig_axis_tvalid,sig_axis_tready :  STD_LOGIC:='0';
signal sig_axis_tdata :  STD_LOGIC_VECTOR(255 DOWNTO 0):=(others=>'0');

begin

fifo : AXI4_Matrix_FIFO PORT MAP
(
    s_axis_aresetn => s_axis_aresetn,
    s_axis_aclk => s_axis_aclk,
    s_axis_tvalid => sig_axis_tvalid,
    s_axis_tready => sig_axis_tready,
    s_axis_tdata => sig_axis_tdata,
    m_axis_tvalid => m_axis_tvalid,
    m_axis_tready => m_axis_tready,
    m_axis_tdata => m_axis_tdata
  );

s_axis_tready<=sig_axis_tready;

process(s_axis_aclk)
variable first:std_logic:='0';
begin
    if s_axis_aclk'event and s_axis_aclk='1' then
        if s_axis_aresetn='0' then
            first:='0';
            sig_axis_tvalid<='0';
            sig_axis_tdata<=(others =>'0');
        elsif s_axis_aresetn='1' and first='0' and sig_axis_tready='1' then
            sig_axis_tvalid<='1';
            sig_axis_tdata<=(others =>'0');
            first:='1';
        elsif first='1' then
            sig_axis_tvalid<=s_axis_tvalid;
            sig_axis_tdata <=s_axis_tdata;
        end if;
    end if;
end process;
end Behavioral;
```

4.2 Kalman Filter VHDL Implementation

The ports of the Kalman filter are the clk, which has a frequency of 100MHz on the basys3 FPGA, a reset in case we want to reset the filter, z_measured which is the value measured by the sensor:

```
entity KalmanFilter is
 Port (
  aclk: in std_logic;
  aresetn: in std_logic;
  s_axis_measurement_tdata:in std_logic_vector(63 downto 0);
  s_axis_measurement_tready:out std_logic;
  s_axis_measurement_tvalid:in std_logic;
  m_axis_estimate_tdata:out std_logic_vector(127 downto 0);
  m_axis_estimate_tready:in std_logic;
  m_axis_estimate_tvalid:out std_logic
  );
end KalmanFilter;
```

The variables for the Kalman Filter will be declared as:

- The $\hat{x}_k^-$ since in our case is a 2×1 column vector is two numbers of 64bit length.
- The P_k^-,A,Q are 2×2 matrices so it is considered as 4 numbers of 64bit length.
- The R matrix is a 1×1 matrix so it is only a 64bit length number.
- The H matrix is a 1×2 matrix so it becomes a row vector of two numbers of 64bit length.
- The K_k matrix is 2×1 matrix so it becomes a column vector of two numbers of 64bit lenght.

4.2.1 Predict State

In the predict step we want to calculate the $\hat{x}_k^-$ and P_k^- values. They are calculated as follows

$$\hat{x}_k^- = \begin{bmatrix} \hat{x}_{0k} \\ \hat{x}_{1k} \end{bmatrix} = \begin{bmatrix} a_0 & a_1 \\ a_2 & a_3 \end{bmatrix} \times \begin{bmatrix} \hat{x}_{0(k-1)} \\ \hat{x}_{1(k-1)} \end{bmatrix} = \begin{bmatrix} a_0 * \hat{x}_{0(k-1)} + a_1 * \hat{x}_{1(k-1)} \\ a_2 * \hat{x}_{0(k-1)} + a_3 * \hat{x}_{1(k-1)} \end{bmatrix} \tag{23}$$

$$P_k^- = \begin{bmatrix} p_{0k}^- & p_{1k}^- \\ p_{2k}^- & p_{3k}^- \end{bmatrix} = \begin{bmatrix} a_0 & a_1 \\ a_2 & a_3 \end{bmatrix} \times \begin{bmatrix} p_{0(k-1)}^- & p_{1(k-1)}^- \\ p_{2(k-1)}^- & p_{3(k-1)}^- \end{bmatrix} \times \begin{bmatrix} a_0 & a_2 \\ a_1 & a_3 \end{bmatrix} + \begin{bmatrix} q_0 & q_2 \\ q_1 & q_3 \end{bmatrix} => \tag{24}$$

$$p_{0k} = a_0^2 * p_{0(k-1)}^- + a_1 * a_0 * (p_{1(k-1)}^- + p_{2(k-1)}^-) + a_1^2 * p_{3(k-1)}^- + q_0$$
$$p_{1k} = a_2 * (a_0 * p_{0(k-1)}^- + a_1 * p_{2(k-1)}^-) + a_3 * (a_0 * p_{1(k-1)}^- + a_1 * p_{3(k-1)}^-)$$
$$p_{2k} = a_0 * (a_2 * p_{0(k-1)}^- + a_3 * p_{2(k-1)}^-) + a_1 * (a_2 * p_{1(k-1)}^- + a_3 * p_{3(k-1)}^-)$$
$$p_{3k} = a_2^2 * p_{0(k-1)}^- + a_3 * a_2 * (p_{1(k-1)}^- + p_{2(k-1)}^-) + a_3^2 * p_{3(k-1)}^-$$

Using these formulas we can make the filter more efficient by assigning directly to each element in the matrix it's corresponding value. After the computations are done we can move to the **Estimate** state. The design of the Predict Step looks as follows:

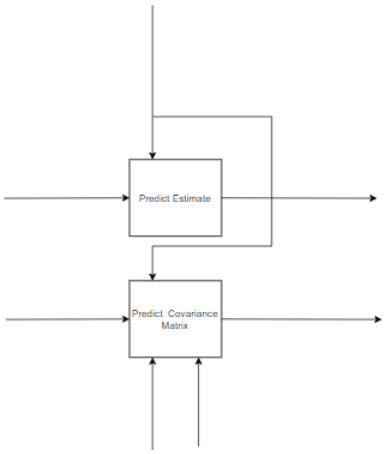


Figure 6: Predict Step

4.2.2 Estimate State

In the Estimate State we compute first the Kalman gain which acts as the α in the Low-Pass filter and helps to filter out the noise, then we compute the estimate, the output of the Kalman filter which estimates the values we want to obtain from the sensor.

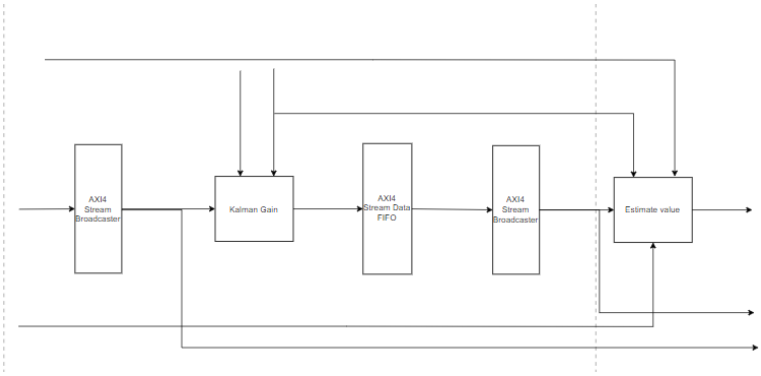


Figure 7: Estimate Step

Kalman Gain computation:

$$K_k = \begin{bmatrix} K_{0k} \\ K_{1k} \end{bmatrix} = \begin{bmatrix} p_{0k}^- & p_{1k}^- \\ p_{2k}^- & p_{3k}^- \end{bmatrix} \times \begin{bmatrix} h_0 \\ h_1 \end{bmatrix} \times (\begin{bmatrix} h_0 & h_1 \end{bmatrix} \times \begin{bmatrix} p_{0k}^- & p_{1k}^- \\ p_{2k}^- & p_{3k}^- \end{bmatrix} \times \begin{bmatrix} h_0 \\ h_1 \end{bmatrix} + R)^{-1} => \tag{25}$$

$K_{0k} = (p_{0k}^- * h_0 + p_{1k}^- * h_1)/\beta$ and
$K_{1k} = (p_{2k}^- * h_0 + p_{3k}^- * h_1)/\beta$, where $\beta = (h_0^2 * p_{0k}^- + h_0 * h_1 * (p_{1k}^- + p_{2k}^-) + h_1^2 * p_{3k}^- + R$

Then the estimate becomes

$$\hat{x}_k = \begin{bmatrix} \hat{x}_{0k} \\ \hat{x}_{1k} \end{bmatrix} = \begin{bmatrix} \hat{x}_{0k}^- \\ \hat{x}_{1k}^- \end{bmatrix} + \begin{bmatrix} K_{0k} \\ K_{1k} \end{bmatrix} \times (z - \begin{bmatrix} h_0 & h_1 \end{bmatrix} \times \begin{bmatrix} \hat{x}_{0k}^- \\ \hat{x}_{1k}^- \end{bmatrix}) = \tag{26}$$

$$\begin{bmatrix} \hat{x}_{0k}^- + K_{0k} * (z - h_0 * \hat{x}_{0k}^- - h_1 * \hat{x}_{1k}^-) \\ \hat{x}_{1k}^- + K_{1k} * (z - h_0 * \hat{x}_{0k}^- - h_1 * \hat{x}_{1k}^-) \end{bmatrix} \tag{27}$$

4.2.3 Computer Error Covariance State

After finding the kalman gain we can compute the error covariance matrix

$$P_k = \begin{bmatrix} p_{0k} & p_{1k} \\ p_{2k} & p_{3k} \end{bmatrix} = \begin{bmatrix} p_{0k}^- & p_{1k}^- \\ p_{2k}^- & p_{3k}^- \end{bmatrix} - \begin{bmatrix} K_{0k} \\ K_{1k} \end{bmatrix} \times \begin{bmatrix} h_0 & h_1 \end{bmatrix} \times \begin{bmatrix} p_{0k}^- & p_{1k}^- \\ p_{2k}^- & p_{3k}^- \end{bmatrix} => \tag{28}$$

$$= \begin{bmatrix} p_{0k}^- - K_{0k} * h_0 * p_{0k}^- - K_{0k} * h_1 * p_{2k}^- & p_{1k}^- - K_{0k} * h_0 * p_{1k}^- - K_{0k} * h_1 * p_{3k}^- \\ p_{2k}^- - K_{1k} * h_0 * p_{0k}^- - K_{1k} * h_1 * p_{2k}^- & p_{3k}^- - K_{1k} * h_0 * p_{1k}^- - K_{1k} * h_1 * p_{3k}^- \end{bmatrix} \tag{29}$$

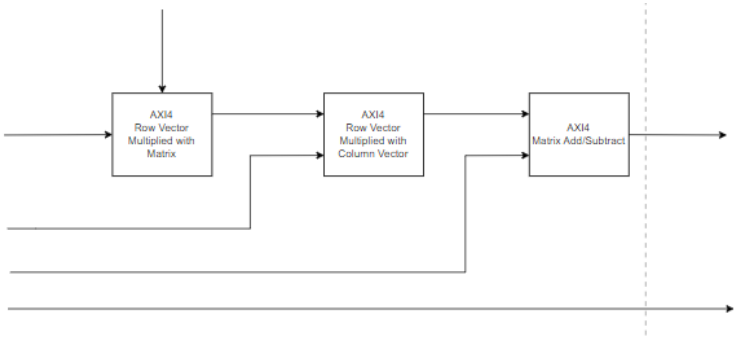


Figure 8: Compute Covariance Matrix Step

4.2.4 Update State

In this step we simply give the calculated estimate and covariance error matrix to the next step $\hat{x}_{k+1} \leftarrow \hat{x}_k$, $P_{k+1} \leftarrow P_k$ This update is the done with the help of the two FIFO's at the beginning that are initialized first.

4.3 UART Communication

Since we want to send the data from a MCU to an FPGA we have to use a communication protocol to be able to send data. The UART protocol suits this application since it is easy to implement in the MCU but for the FPGA it might not be easy, but it is not as hard as other communication protocols to be implemented

```
entity UART is
  Port (
    clk:in std_logic;
    reset:in std_logic;
    tx_start:in std_logic;
    data_in: in  std_logic_vector (7 downto 0);
    data_out: out std_logic_vector (7 downto 0);
    rx: in  std_logic;
    tx: out std_logic
  );
end UART;
```

This is the UART module of the FPGA and it consists of two main components: the UART receiver (RX) and the UART Transmitter (TX).

```
entity UART_RX is
  Port (
    clk:in std_logic;
    reset:in std_logic;
    rx_data_in:in std_logic;
    rx_data_out:out std_logic_vector(7 downto 0)
  );
end UART_RX;
```

```
entity UART_TX is
 Port (
    clk:in std_logic;
    reset:in std_logic;
    tx_start:in std_logic;
    tx_data_in:in std_logic_vector(7 downto 0);
    tx_data_out:out std_logic
  );
end UART_TX;
```

UART communication protocol uses a baud rate generator. The FPGA has a 100MHz frequency and the baud rate chosen is 115200 then the TX component will have the baud clock set every 864 (10^8/115200) clk cycles and the receiver will have the baud clock set every 54 (10^8/115200/16) clock cycles ,oversampling the data to put the capture point at the middle of the receiving bit.

```
entity UART_Baud_Rate_Generator is
    Generic ( baud_rate: std_logic_vector(15 downto 0):=x"28B0");
    Port (
        reset : in std_logic;
        clk: in std_logic;
        baud_clk:out std_logic
     );
end UART_Baud_Rate_Generator;

process(clk)
variable cnt:std_logic_vector(15 downto 0):=baud_rate - 1;
begin
    if clk'event and clk='1' then
        if reset = '1' then
            cnt:=baud_rate - 1;
            baud_clk<='0';
        else
            if cnt = x"000" then
                cnt:=baud_rate - 1;
                baud_clk<='1';
            else
                cnt:=cnt - 1;
                baud_clk<='0';
            end if;
        end if;
     end if;
end process;
```

Another component which both the RX and the TX part have are their own FSM which are used to receive/send data. Both FSM have 4 states :Idle, Start, Data, and Stop. At each step they almost do the same thing : in Idle state they wait for the Start bit 0 to go to the Start State, in Start State they prepare to receive the data, in the DATA state they receive/transmit the data frame and in the Stop state they stop after receiving/transmitting the data frame.

```
entity UART_RX_FSM is
  Port (
    clk : in std_logic;
    baud_clk:in std_logic;
    reset : in std_logic;
    rx_data_in : in std_logic;
    rx_data_out : out std_logic_vector(7 downto 0)
  );
end UART_RX_FSM;
```

```
entity UART_TX_FSM is
  Port (
    clk:in std_logic;
    reset:in std_logic;
    start_detected:in std_logic;
    data_index:in std_logic_vector(2 downto 0);
    stored_data:in std_logic_vector(7 downto 0);
    baud_clk:in std_logic;
    start_reset:out std_logic;
    data_index_reset:out std_logic;
    tx_data_out:out std_logic
   );
end UART_TX_FSM;
```

The TX part has two more components: the start detection component and the data index counter component. The start detection component's role is to secure transmitting data by capturing when the txstart signal changes.

```
entity UART_TX_Start_Detector is
  Port (
    clk:in std_logic;
    reset:in std_logic;
    stored_data:out std_logic_vector(7 downto 0);
    tx_data_in:in std_logic_vector(7 downto 0);
    start_reset:in std_logic;
    tx_start:in std_logic;
    in_start_detected:in std_logic;
    out_start_detected:out std_logic
   );
end UART_TX_Start_Detector;
```

The data index counter component is a simple 0 to 7 counter that keeps track of the current bit we are sending from the data frame.

```
entity UART_TX_data_index is
    Port (
        clk : in std_logic;
        reset : in std_logic;
        data_index_reset : in std_logic;
        baud_clk : in std_logic;
        data_index : out std_logic_vector(2 downto 0)
     );
end UART_TX_data_index;
```

4.4 Top Level

The top level of the design is the following:

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

entity TopLevel is
   Port (
        clk: in std_logic;
        sw : in std_logic_vector(15 downto 0);
        btn : in std_logic_vector(4 downto 0);
        an: out std_logic_vector(3 downto 0);
        cat: out std_logic_vector(6 downto 0);
        JA1: in std_logic;
        JA2: out std_logic
    );
end TopLevel;

architecture Behavioral of TopLevel is

component KalmanFilter is
 Port (
  aclk: in std_logic;
  aresetn: in std_logic;
  s_axis_measurement_tdata:in std_logic_vector(63 downto 0);
  s_axis_measurement_tready:out std_logic;
  s_axis_measurement_tvalid:in std_logic;
  m_axis_estimate_tdata:out std_logic_vector(127 downto 0);
  m_axis_estimate_tready:in std_logic;
  m_axis_estimate_tvalid:out std_logic
  );
end component KalmanFilter;

component UART is
  Port (
    clk:in std_logic;
    reset:in std_logic;
    tx_start:in std_logic;
    data_in: in  std_logic_vector (7 downto 0);
    data_out: out std_logic_vector (7 downto 0);
    rx_data_valid: out std_logic;
    rx: in  std_logic;
    tx: out std_logic
  );
end component UART;

component ShiftRegister64Bit is
  Port (
    clk : in std_logic;
    rx_data_valid:in std_logic;
    reset : in std_logic;
    rx_data_out : in std_logic_vector(7 downto 0);
    measurement: out std_logic_vector(63 downto 0);
    measurement_valid: out std_logic;
    measurement_ready: in std_logic
  );
end component ShiftRegister64Bit;

component debouncer is
  Port ( clk : in std_logic;
         btn : in std_logic;
         en : out std_logic );
end component debouncer;

component SevenSegDisplay is
    Port (
        clk:in std_logic;
        digit0: in std_logic_vector(3 downto 0);
        digit1: in std_logic_vector(3 downto 0);
        digit2: in std_logic_vector(3 downto 0);
        digit3: in std_logic_vector(3 downto 0);
        anodes: out std_logic_vector(3 downto 0);
        cathodes: out std_logic_vector(6 downto 0)
    );
end  component SevenSegDisplay;
```

```
signal estimate_tready : std_logic :='1';
signal reset_sig, estimate_tvalid : std_logic:='0';
signal rx_data_valid,measurement_valid,measurement_ready : std_logic:='0';
signal areset_sig : std_logic:='1';

signal data_out :std_logic_vector(7 downto 0) := (others => '0');

signal measurement : std_logic_vector(63 downto 0) := (others => '0');
signal estimate : std_logic_vector(127 downto 0) := (others => '0');
signal estimateSSD : std_logic_vector(127 downto 0) := (others => '0');

signal digit: std_logic_vector(15 downto 0):= (others => '0');

begin

debouncer_reset:debouncer port map
(
    clk => clk,
    btn => btn(0),
    en =>reset_sig
);

areset_sig <= not reset_sig;

UART_module:UART port map(
    clk => clk,
    reset => reset_sig,
    tx_start => '0',
    data_in => x"00",
    data_out =>data_out,
    rx_data_valid => rx_data_valid,
    rx => JA1,
    tx => JA2
);

ShiftReg: ShiftRegister64Bit port map(
    clk => clk,
    rx_data_valid => rx_data_valid,
    reset => reset_sig,
    rx_data_out => data_out,
    measurement => measurement,
    measurement_valid => measurement_valid,
    measurement_ready => measurement_ready
);

KalmanFilterUnit:KalmanFilter port map(
  aclk => clk,
  aresetn =>areset_sig,
  s_axis_measurement_tdata => measurement,
  s_axis_measurement_tready => measurement_ready,
  s_axis_measurement_tvalid => measurement_valid,
  m_axis_estimate_tdata =>  estimate,
  m_axis_estimate_tready => estimate_tready,
  m_axis_estimate_tvalid => estimate_tvalid
);

with sw(0) select digit <= estimateSSD(15 downto 0) when '0',
                           estimateSSD(79 downto 64) when '1',
                           (others =>'0') when others;


SSD:SevenSegDisplay port map(
        clk => clk,
        digit0 =>digit(3 downto 0),
        digit1 =>digit(7 downto 4),
        digit2 =>digit(11 downto 8),
        digit3 =>digit(15 downto 12),
        anodes => an,
        cathodes => cat
);

end Behavioral;
```

4.5 Photoresistor Circuit Implementation & Arduino Code

The circuit for reading the voltage after the photoresistor is:

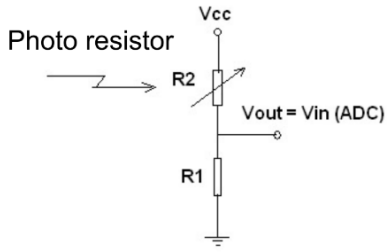


Figure 9: Photoresistor Circuit

Where $R_1 = 200\Omega$. In order to convert the read voltage into resistance we use the fact that the $V_{CC} = 5V$ and the ADC resolution 10 bits, it follows that the voltage is $V_{out} = \frac{V_{in}*5}{1024}$. Since the photoresistor takes values from $1k\Omega$ to $350M\Omega$ then the resoluiton of the photoresistor will be $\frac{5V}{350M\Omega - 1k\Omega}$ which is approximately 0.0000000142857551. With the previously mentioned formulas we can cmpute the resistance on the MCU:

```
const int photoresistor_pin = 34;
const float sensor_resolution = 0.0000000142857551f; //
    range for photoresistor is 350.000.000 ohm(light) - 1000
    ohm(dark)
int value=0;

void setup() {
  pinMode(photoresistor_pin,INPUT);
  Serial.begin(115200);
  Serial2.begin(115200);
}

void loop() {

  value = analogRead(photoresistor_pin);
  float vout = ((float)value*3.3)/1024;
  float LDR_resistance = vout /sensor_resolution;
  uint64_t int_LDR = (uint64_t) LDR_resistance;
  Serial.println(int_LDR);
  Serial2.print(int_LDR);
  delay(1000);
}
```

The values will be printed in a CSV using the CoolTerm software, which connects to the Serial0 of the MCU, instead of printing the values in the serial monitor they are directly written in the CSV.

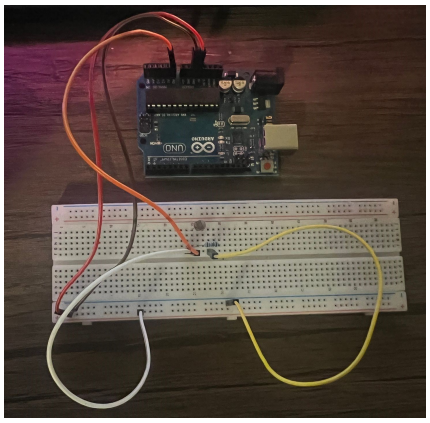


Figure 10: Photoresistor and Arduino Circuit

5 Testing and Validation

5.1 Arduino Code and Photoresistor Testing

The testing of the circuit the the Arduino code was made using the CoolTerm software. The following values were received from the MCU:

```
65624812.00
82372816.00
182860816.00
244042272.00
278221856.00
300438592.00
319237376.00
338719744.00
343846688.00
342821280.00
319579168.00
279930848.00
235839168.00
193456480.00
112109056.00
69042776.00
66308404.00
65966608.00
66991996.00
68359184.00
61523264.00
60839672.00
84423592.00
81005632.00
66650200.00
40331916.00
64599428.00
19140570.00
2392571.25
1367183.62
683591.81
341795.90
0.00
0.00
68700976.00
69042776.00
68700976.00
69384568.00
69042776.00
69042776.00
69042776.00
69042776.00
69042776.00
69384568.00
68700976.00
68700976.00
69384568.00
68700976.00
```

Figure 11: Photoresistor Values

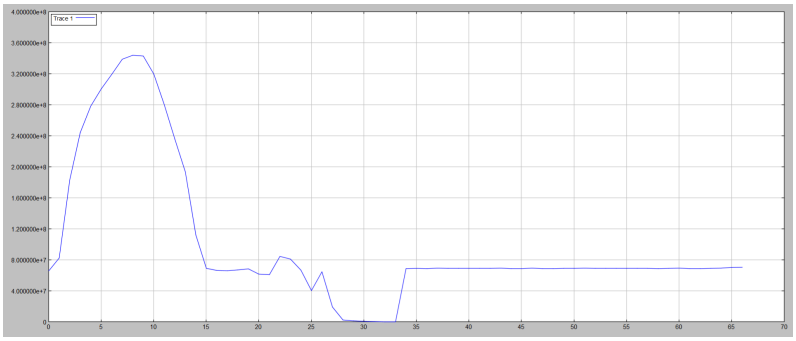


Figure 12: Photoresistor Values Chart

The values were obtained by putting the photoresistor in different conditions, in a dimly lit room, using the flashlight from the phone and covering the photoresistor. Also the light from Arduino board influences the value of the photoresistor's resistance.

5.2 AXI4-Stream Components Testing

5.2.1 AXI4-Stream Vector Addition/Subtraction Testing

The vectors chosen for this testbench were two equal vector $\begin{bmatrix}1\\1\end{bmatrix}$ which were added and then subtracted. The expected values are $\begin{bmatrix}2\\2\end{bmatrix}$ or addition and $\begin{bmatrix}0\\0\end{bmatrix}$ for subtraction.

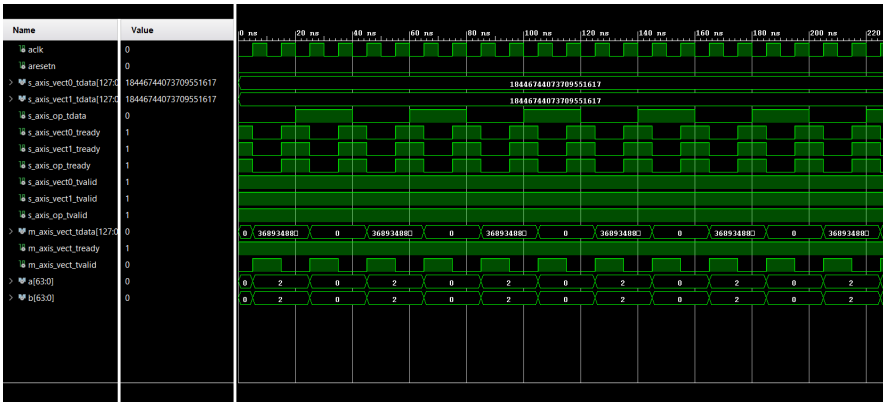


Figure 13: Vector Addition/Subtraction Testbench Results

5.2.2 AXI4-Stream Dot Product Testing

The chosen dot products for this testbench are : $\begin{bmatrix}1\\1\end{bmatrix} \times \begin{bmatrix}1\\1\end{bmatrix} = 2$

$\begin{bmatrix}2\\2\end{bmatrix} \times \begin{bmatrix}3\\4\end{bmatrix} = 14$, $\begin{bmatrix}5\\2\end{bmatrix} \times \begin{bmatrix}-1\\2\end{bmatrix} = -1$

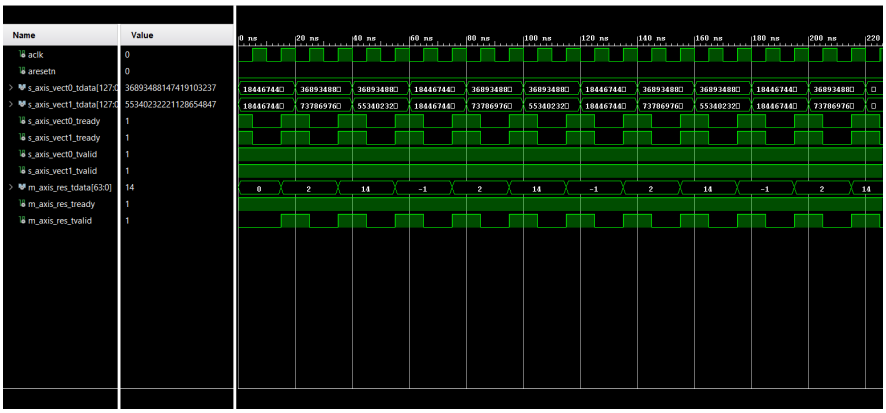


Figure 14: Dot Product Testbench Results

5.2.3 AXI4-Stream Matrix Addition/Subtraction Testing

The chosen matrices for addition in this testbench are:
$\begin{bmatrix}1 & 1\\1 & 1\end{bmatrix} + \begin{bmatrix}1 & 1\\1 & 1\end{bmatrix} = \begin{bmatrix}2 & 2\\2 & 2\end{bmatrix}$, $\begin{bmatrix}1 & 1\\1 & 1\end{bmatrix} - \begin{bmatrix}1 & 1\\1 & 1\end{bmatrix} = \begin{bmatrix}0 & 0\\0 & 0\end{bmatrix}$, $\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix} + \begin{bmatrix}0 & 1\\2 & 3\end{bmatrix} = \begin{bmatrix}1 & 3\\5 & 7\end{bmatrix}$, $\begin{bmatrix}-1 & 1\\1 & 1\end{bmatrix} - \begin{bmatrix}1 & 1\\1 & 1\end{bmatrix} = \begin{bmatrix}-2 & 0\\0 & 0\end{bmatrix}$,

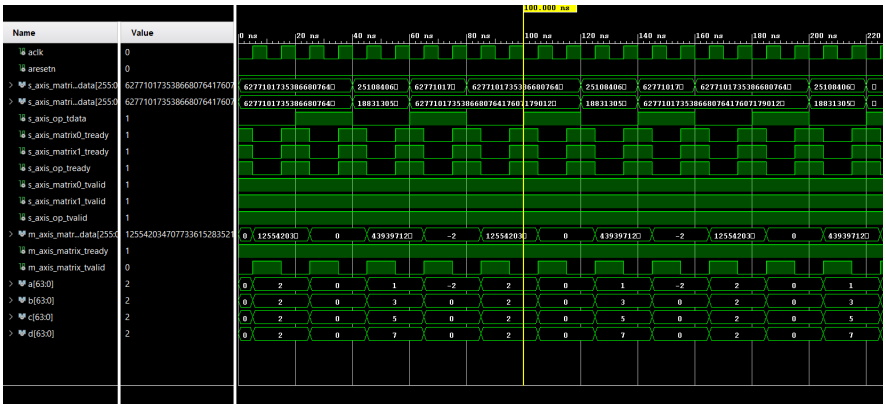


Figure 15: Matrix Addition/Subtraction Testbench Results

5.2.4 AXI4-Stream Matrix Multiplication Testing

The chosen matrices for addition in this testbench are:
$\begin{bmatrix}1 & 1\\1 & 1\end{bmatrix} \times \begin{bmatrix}1 & 1\\1 & 1\end{bmatrix} = \begin{bmatrix}2 & 2\\2 & 2\end{bmatrix}$, $\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix} \times \begin{bmatrix}2 & 2\\2 & 2\end{bmatrix} = \begin{bmatrix}2 & 2\\2 & 2\end{bmatrix}$

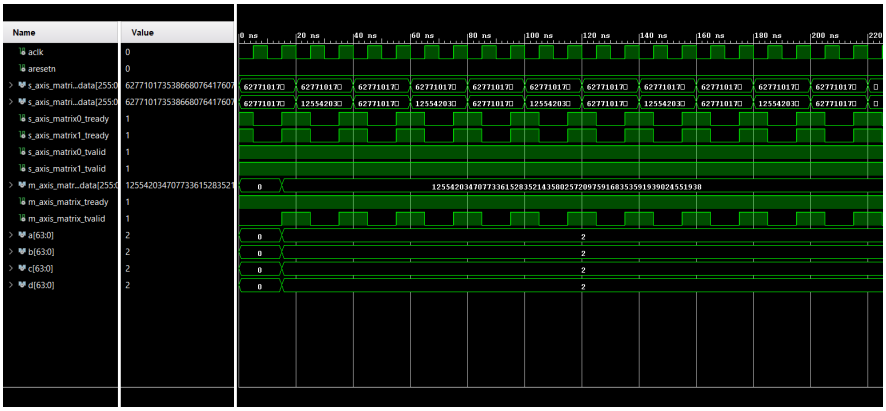


Figure 16: Matrix Multiplication Testbench Results

5.2.5 AXI4-Stream Matrix Column Vector Multiplication Testing

The chose matrices and vectors for the operations are:
$\begin{bmatrix}1 & 1\\1 & 1\end{bmatrix} \times \begin{bmatrix}1\\1\end{bmatrix} = \begin{bmatrix}2\\2\end{bmatrix}$, $\begin{bmatrix}15 & 14\\13 & 12\end{bmatrix} \times \begin{bmatrix}0\\0\end{bmatrix} = \begin{bmatrix}0\\0\end{bmatrix}$, $\begin{bmatrix}1 & 0\\0 & 1\end{bmatrix} \times \begin{bmatrix}1\\1\end{bmatrix} = \begin{bmatrix}1\\1\end{bmatrix}$

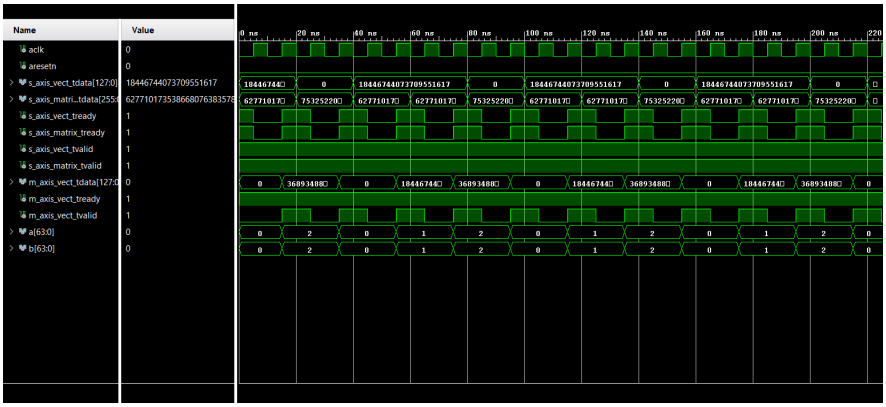


Figure 17: Matrix Multiplication With Column Vector Testbench Results

5.3 UART Testbench

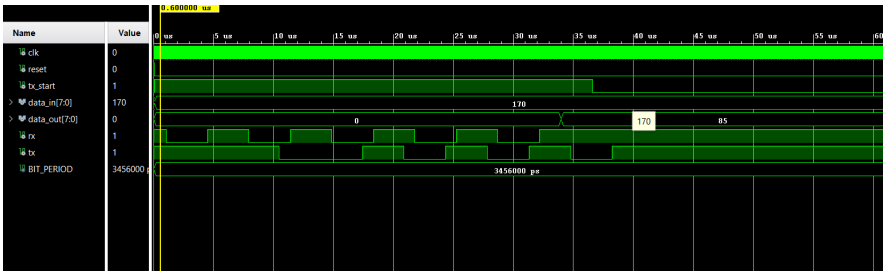


Figure 18: UART Testbench

From the image we can follow the communication protocol and how does the UART module work, the **rx** and **tx** signals are on logic level '1' at first then the data is received or sent. Both signals are first brought to logic '0' to signal that the device should start receiving data. After 8 bits have been read/sent, the signal go back to logic '1' to signal that the data transmission has ended.The code for the testbench of the UART module is:

```
1       library IEEE;
2   use IEEE.STD_LOGIC_1164.ALL;
3
4   entity tb_UART is
5   --  Port ( );
6   end tb_UART;
7
8   architecture Behavioral of tb_UART is
9
10  component UART is
11    Port (
12      clk:in std_logic;
13      reset:in std_logic;
14      tx_start:in std_logic;
15      data_in: in  std_logic_vector (7 downto 0);
16      data_out: out std_logic_vector (7 downto 0);
17      rx: in  std_logic;
18      tx: out std_logic
19    );
20  end component UART;
21
22      signal clk: std_logic := '0';
23      signal reset: std_logic := '0';
24      signal tx_start: std_logic := '0';
25      signal data_in: std_logic_vector(7 downto 0) := (others => '0');
26      signal data_out: std_logic_vector(7 downto 0);
27      signal rx: std_logic := '1';
28      signal tx: std_logic := '1';
29
30
31      constant BIT_PERIOD   : time := 54*4*16 ns;
32
33  begin
34
35      clk <= not clk after 2ns;
36      uarttb: UART
37          Port map (
38              clk => clk,
39              reset => reset,
40              tx_start => tx_start,
41              data_in => data_in,
42              data_out => data_out,
43              rx => rx,
44              tx => tx
45          );
46
47
48      process
49      begin
50
51          reset <= '1';
```

```
        wait for 50 ns;
        reset <= '0';

        tx_start <= '1';
        data_in <= "10101010";

        wait for 1 us;

        rx <= '0'; -- Start bit
        wait for BIT_PERIOD;
        rx <= '1'; -- Data bit 0
        wait for BIT_PERIOD;
        rx <= '0'; -- Data bit 1
        wait for BIT_PERIOD;
        rx <= '1'; -- Data bit 2
        wait for BIT_PERIOD;
        rx <= '0'; -- Data bit 3
        wait for BIT_PERIOD;
        rx <= '1'; -- Data bit 4
        wait for BIT_PERIOD;
        rx <= '0'; -- Data bit 5
        wait for BIT_PERIOD;
        rx <= '1'; -- Data bit 6
        wait for BIT_PERIOD;
        rx <= '0'; -- Data bit 7
        wait for BIT_PERIOD;
        rx <= '1'; -- Stop bit
        wait for BIT_PERIOD;

        wait for 1 us;
        tx_start <= '0';

        wait;
    end process;
end Behavioral;
```

5.4 UART Communication Testbench

For this testbench we simulate two devices which communicate via the UART protocol using the built UART module.The modules were connected by putting the **rx** port of one module to the **tx** port of the other module and the same was done for the second module.

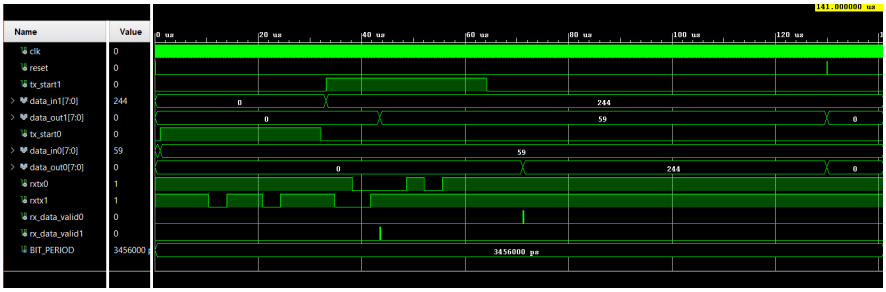


Figure 19: UART Communication Testbench

Following the simulation we can see that the modules transmit and receive the correct data that was given as an input to both of them on the **data_in** signal, and the output is correctly set back to the default 0 when the **reset** signal is asserted.

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

entity tb_UART_Communication is
--  Port ( );
end tb_UART_Communication;

architecture Behavioral of tb_UART_Communication is

component UART is
  Port (
    clk:in std_logic;
    reset:in std_logic;
    tx_start:in std_logic;
    data_in: in  std_logic_vector (7 downto 0);
    data_out: out std_logic_vector (7 downto 0);
    rx_data_valid: out std_logic;
    rx: in  std_logic;
    tx: out std_logic
  );
end component UART;

    signal clk: std_logic := '0';
    signal reset: std_logic := '0';


    signal tx_start1: std_logic := '0';
    signal data_in1: std_logic_vector(7 downto 0) := (others => '0');
    signal data_out1: std_logic_vector(7 downto 0);


    signal tx_start0: std_logic := '0';
    signal data_in0: std_logic_vector(7 downto 0) := (others => '0');
    signal data_out0: std_logic_vector(7 downto 0);
    signal rxtx0: std_logic := '1';
    signal rxtx1: std_logic := '1';

    signal rx_data_valid0: std_logic := '0';
    signal rx_data_valid1: std_logic := '0';

    constant BIT_PERIOD   : time := 54*4*16 ns;

begin


    clk <= not clk after 2ns;
    uarttb0: UART
        Port map (
            clk => clk,
            reset => reset,
            tx_start => tx_start0,
            data_in => data_in0,
            data_out => data_out0,
            rx_data_valid => rx_data_valid0,
            rx => rxtx0,
            tx => rxtx1
        );

    clk <= not clk after 2ns;
    uarttb1: UART
        Port map (
            clk => clk,
            reset => reset,
            tx_start => tx_start1,
            data_in => data_in1,
            data_out => data_out1,
            rx_data_valid => rx_data_valid1,
            rx => rxtx1,
            tx => rxtx0
        );

    process
    begin

        reset <= '1';
        wait for 50 ns;
        reset <= '0';

        wait for 1 us;
        data_in0<= x"3B";
        tx_start0 <= '1';

        wait for BIT_PERIOD*9;
        tx_start0 <='0';

        wait for 1us;
        data_in1<= x"F4";
        tx_start1 <= '1';

        wait for BIT_PERIOD*9;
```

```
        tx_start1 <='0';

        wait for BIT_PERIOD*19;
        reset <= '1';
        wait for 20ns;
        reset <= '0';
        wait;
    end process;

end Behavioral;
```

5.5 Kalman Filter Testing & Results

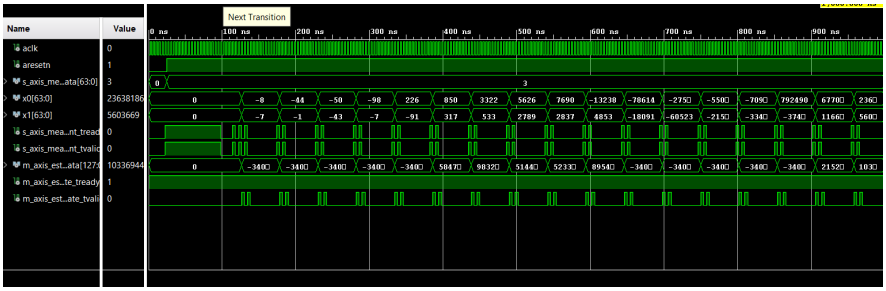


Figure 20: Kalman Filter Testbench

$\hat{x_0} = \begin{bmatrix} -5 \\ 2 \end{bmatrix}, P_0 = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} A = \begin{bmatrix} 3 & 3 \\ 1 & -1 \end{bmatrix}, Q = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}, A^T = \begin{bmatrix} 3 & 1 \\ 3 & -1 \end{bmatrix}, H = \begin{bmatrix} -2 & 2 \end{bmatrix}, R = -107$

For testing I chose the measurement $z = 3$, and the following computations are performed:

I.

1.Predict:

$\hat{x_2^-} = \begin{bmatrix} 3 & 3 \\ 1 & -1 \end{bmatrix} \times \begin{bmatrix} -8 \\ 7 \end{bmatrix} = \begin{bmatrix} -9 \\ 7 \end{bmatrix}$

$, P_1^- = \begin{bmatrix} 3 & 3 \\ 1 & -1 \end{bmatrix} \times \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \times \begin{bmatrix} 3 & 1 \\ 3 & -1 \end{bmatrix} + \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} = \begin{bmatrix} 38 & 0 \\ 0 & 2 \end{bmatrix}$

2.Estimate:

$K_1 = \begin{bmatrix} 38 & 0 \\ 0 & 2 \end{bmatrix} \times \begin{bmatrix} -2 \\ 2 \end{bmatrix} \times (\begin{bmatrix} -2 & 2 \end{bmatrix} \times \begin{bmatrix} 38 & 0 \\ 0 & 2 \end{bmatrix} \times \begin{bmatrix} -2 \\ 2 \end{bmatrix} - 107)^{-1} = \begin{bmatrix} -1 \\ 0 \end{bmatrix}$

$\hat{x_1} = \begin{bmatrix} -9 \\ 7 \end{bmatrix} + \begin{bmatrix} -1 \\ 0 \end{bmatrix} \times (3 - \begin{bmatrix} -2 & 2 \end{bmatrix} \times \begin{bmatrix} -9 \\ 7 \end{bmatrix}) = \begin{bmatrix} -8 \\ 7 \end{bmatrix}$ the first value of the estimate in the testbench.

3.Compute Covariance:

$P_1 = \begin{bmatrix} 38 & 0 \\ 0 & 2 \end{bmatrix} - \begin{bmatrix} -1 \\ 0 \end{bmatrix} \times \begin{bmatrix} -2 & 2 \end{bmatrix} \times \begin{bmatrix} 38 & 0 \\ 0 & 2 \end{bmatrix} = \begin{bmatrix} -38 & 4 \\ 0 & 2 \end{bmatrix}$

II.

1.Predict:

$\hat{x_1^-} = \begin{bmatrix} 3 & 3 \\ 1 & -1 \end{bmatrix} \times \begin{bmatrix} -8 \\ -7 \end{bmatrix} = \begin{bmatrix} -45 \\ -1 \end{bmatrix}$

$, P_1^- = \begin{bmatrix} 3 & 3 \\ 1 & -1 \end{bmatrix} \times \begin{bmatrix} -38 & 4 \\ 0 & 2 \end{bmatrix} \times \begin{bmatrix} 3 & 1 \\ 3 & -1 \end{bmatrix} + \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix} = \begin{bmatrix} -286 & -132 \\ -108 & -38 \end{bmatrix}$

2.Estimate:

$K_1 = \begin{bmatrix} -286 & -132 \\ -108 & -38 \end{bmatrix} \times \begin{bmatrix} -2 \\ 2 \end{bmatrix} \times (\begin{bmatrix} -2 & 2 \end{bmatrix} \times \begin{bmatrix} -286 & -132 \\ -108 & -38 \end{bmatrix} \times \begin{bmatrix} -2 \\ 2 \end{bmatrix} - 107)^{-1} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$

$\hat{x_1} = \begin{bmatrix} -45 \\ -1 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \end{bmatrix} \times (3 - \begin{bmatrix} -2 & 2 \end{bmatrix} \times \begin{bmatrix} -45 \\ -1 \end{bmatrix}) = \begin{bmatrix} -45 \\ -1 \end{bmatrix}$ the first value of the estimate in the testbench.

3.Compute Covariance:

$P_1 = \begin{bmatrix} -286 & -132 \\ -108 & -38 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \end{bmatrix} \times \begin{bmatrix} -2 & 2 \end{bmatrix} \times \begin{bmatrix} -286 & -132 \\ -108 & -38 \end{bmatrix} = \begin{bmatrix} -286 & -132 \\ -108 & -38 \end{bmatrix}$

References

[1] AMD, *Vitis High-Level Synthesis User Guide (UG1399) – AXI4 Stream Interfaces* [Online]. Available: https://docs.amd.com/r/en-US/ug1399-vitis-hls/AXI4-Stream-Interfaces. Accessed: Oct. 2024.

[2] N. A. Thacker and A. J. Lacey, *Tutorial: The Kalman Filter*, Last updated: Dec. 1, 1998. [Online]. Available: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=c631533d3095f0385dee2774675300a02c6cf6b7. Accessed: Oct. 2024.

[3] Wikipedia, *Kalman filter* [Online]. Available: https://en.wikipedia.org/wiki/Kalman_filter. Accessed: Oct. 2024.

[4] R. Bracamontes Del Toro, "How to interface FPGAs to microcontrollers," *Atmel*, Jul. 30, 2008. [Online]. Available: https://www.eetimes.com/how-to-interface-fpgas-to-microcontrollers/. Accessed: Oct. 2024.

[5] Structure of Computer Systems Laboratory, *AXI4-Stream Communication Protocol. Introduction* [Online]. Accessed: Oct. 2024.

[6] Structure of Computer Systems Laboratory, *AXI4-Stream Communication Protocol. AXI4-Stream Compliant Modules* [Online]. Accessed: Oct. 2024.

[7] G. Chen and L. Guo, "The FPGA Implementation of Kalman Filter," Department of Electronic Science and Technology, University of Science & Technology of China. [Online]. Available: https://web.cecs.pdx.edu/~mperkows/CLASS_VHDL_99/S2016/04.KALMAN_FILTER_FADDEEV_ALG/016.%20Chen%20and%20Guo%20=%20Kalman%20Filter%20Faddeev.pdf. Accessed: Oct. 2024.

[8] Luminous flux [Online]. Accessed: Nov. 2024. https://en.wikipedia.org/wiki/Luminous_flux

[9] Illuminance [Online]. Accessed: Nov. 2024. https://en.wikipedia.org/wiki/Illuminance

[10] Basys 3 Artix-7 FPGA Trainer Board: Recommended for Introductory Users [Online]. Accessed: Nov. 2024. https://digilent.com/shop/basys-3-artix-7-fpga-trainer-board-recommended-for-introductory-users/

[11] Calculating luminosity from LDR. Accessed: Nov. 2024. https://www.aranacorp.com/en/luminosity-measurement-with-a-photoresistor/

[12] UART Communication Protocol on Basys3. Accessed: Dec. 2024 https://www.hackster.io/alexey-sudbin/uart-interface-in-vhdl-for-basys3-board-eef170