EEE102-02 Project Report:

Finger Tracking Ring

Kaan Ermertcan - 22202823

31.12.2023

YouTube Presentation Video:

https://youtu.be/DLpPyW4ItyI

Introduction:

In this project, I designed a ring that tracks finger movements and moves the cursor of a PC towards the point user's finger is pointing at. Besides, it can simulate mouse clicks when the push button on the ring is pressed.

Design Specifications:

Parts List: Basys 3, BMX160 9-axis sensor board, push button, 10K resistor.

Basys 3 communicates with the sensor via i2c protocol. It configures the sensor to the required configuration after power-up. Then, it periodically reads the gyroscope data (angular speed in 3 axes) and time of the data sample and calculates how much it has been rotated in degrees compared to the reference direction. Reference direction can be set by pressing center button on Basys3. Calculated angle is displayed on seven segment display and displayed axis can be selected by the rightmost switch. Angle data is periodically transmitted over UART to the PC. Finally, a Python script reads the data from UART and maps the angles in a predefined angle range in Y and Z axis of the sensor to the cursor's absolute position on the screen. Average of the angle data for the two axes is calculated and sent over UART and used for data redundancy checking. The state of the external push button is read from one of the ports, sent via UART and mouse clicks are applied according to push button's state.

Methodology:

My design runs at 100 MHz clock of Basys 3. Module diagram of the design can be seen in the results section. A module named sensor_handler is created to acquire and process sensor data. The first task was to configure the sensor before the data could be acquired. For this purpose, an i2c

communication must be established with the sensor. To save time, a component handling the i2c specifications is acquired from the web (Digikey, 2021). I2c is configured to run at 100 KHz. The usage of the component is studied. The datasheet of BMX160 is also studied to determine how the sensor is configured (Bosch, 2019). Summary of the configuration procedure is listed below:

Perform a soft reset by writing to command register.
Write the gyroscope configuration to gyro_conf register.
Set the gyroscope range by writing to gyro_range register.
Write the accelerometer configuration to acc_conf register.
Set the accelerometer range by writing to acc_range register.
Set power mode of the accelerometer to normal by writing to command register.
Set power mode of the gyroscope to normal by writing to command register.
Set power mode of the magnetometer to normal by writing to command register.

Table 1.1: Configuration procedure.

For each step, a write operation is performed. An example transaction for one write can be seen in the figure below:

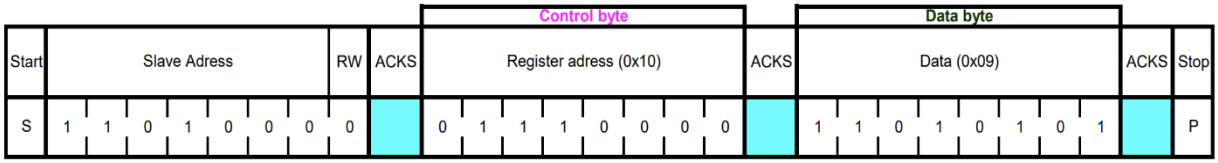

Figure 1.1: An example write operation (Bosch, 2019).

Here, serial data and start, stop, acknowledge bits are handled by i2c component. As we are communicating with a single device, the address is directly connected to the i2c component. Thus, for each step in the configuration process the designed module enables the i2c component to initiate a write transaction, latch the register address to the i2c component and after the control byte is sent (indicated by busy signal from i2c component going low), latch the data byte. After the data byte is sent, i2c component is disabled to end the transaction. Finally, before the next configuration step there is a 1 second wait for the command to be performed by the sensor.

The configuration procedure is run once in the setup state which is the initial state on power-up. The gyroscope range is set as 125 degrees per second, it is the smallest range for increased accuracy. Motion is expected to be slow enough to not exceed the range. Accelerometer range is set as 2g for similar reasons. Output data rate, or frequency of data sampling of the sensor is set as maximum, 1600 Hz, to increase the accuracy and responsiveness of the angle data.

After the sensor is configured, the module should periodically read the data from the sensor. The waiting period between reads is performed by waiting in the hold state for a defined number of clock cycles. After the wait period is finished, it goes into the read state, in which the data is read from the sensor.

To read from the sensor, the register address to be read must be written first. After the write operation is finished (indicated by busy going low), i2c component should be switched to read mode and

the data can be taken when the busy signal goes low, indicating one byte of data is read. The sensor continues to send the data in the subsequent register addresses until i2c component is disabled, thus the full range of addresses can be read subsequently without writing address byte again. An example read transaction can be seen below:

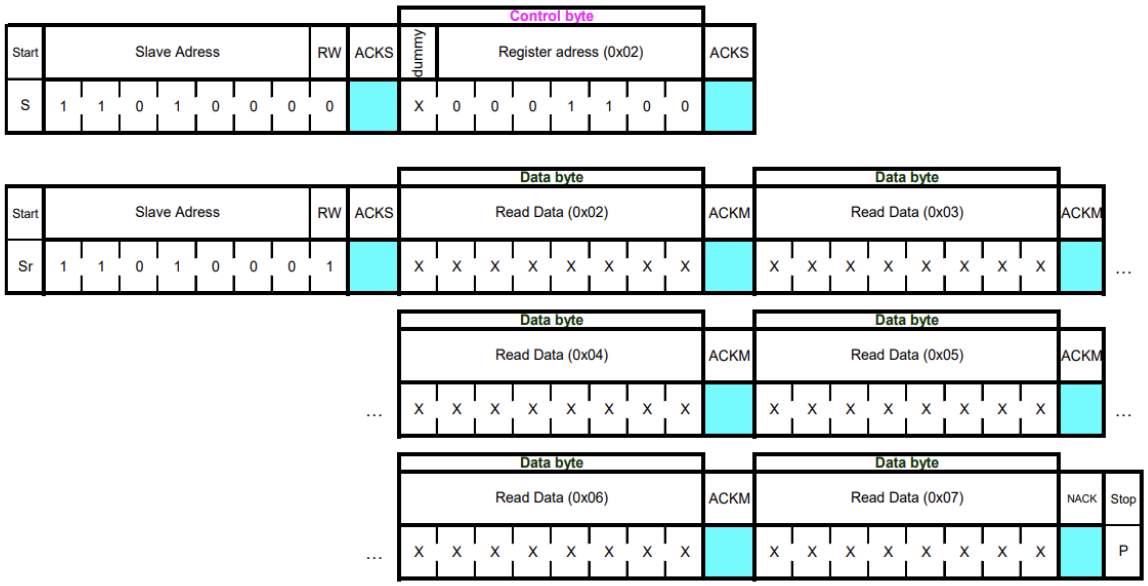

Figure 1.1: An example read operation (Bosch, 2019).

In the read state, the register address for gyroscope data is written and gyroscope data, accelerometer data and sensor time (time when the data is sampled) is read throughout the subsequent registers. Data for each axis is 2 bytes and for the sensor time is 3 bytes. Thus, the first byte(s) is kept in data signal until it is concatenated with subsequent byte and multiplied with a multiplier to get the data in a meaningful unit. This is necessary as the sensor data for each axis is a 16-bit signed integer where the range set in the configuration corresponds to the complete range of signed numbers that can be represented with 16 bits. As it is not easy to deal with floating point numbers, multiplier is multiplied by 10^6. After multiplication the number is divided again by 10^3 to be able to fit it to 24 bits. After sensor time is read transmission is ended. Sensor time is a counter with a clock period of 39µs. Thus, to determine the change in angle since the last received data, sensor time difference is multiplied by 39µs and data for each axis. These angle changes are summed continuously to determine the angle change since last reference reset. Then, the design goes to the hold state again until the next read.

A module named transmitter is designed to send the angle data to the PC. The transmitter sends the data over UART via FT2232HQ chip on the Basys3. Component required for handling UART transmission specifications is borrowed from the internet (Merrick, 2023). Its baud rate is set as 1 MHz to achieve fast data flow and because 100 MHz is an integer multiple of 1 MHz for exact clock division. The designed module takes the data and their average from sensor_handler via the main module. The transmitter is made up of two states, hold and transmit, similar to sensor_handler. It switches between these two states periodically and transmits the data each time it goes through the transmit state. In transmit state, tx_count keeps how many bytes of data is transmitted. Data byte going to the uart component is switched depending on tx_count. A total of 7 bytes is transferred where Y axis angle, Z axis angle, average of Y and Z angles are each 2 bytes and mouse click is 1 byte. If the uart component is

on idle indicated by the active and done signals, the data byte is latched onto uart component and transmission begins. When the byte is transmitted tx_count is increased. If the last byte has been transmitted, tx_count is reset, and state becomes hold.

A short python script is used to receive the UART transmission, move the cursor and perform left mouse clicks. UART transmissions can get corrupted because the clock is not synchronized between two ends of the transmission. To prevent unwanted cursor movements caused by data corruption, a redundancy check is implemented. After each transmission, the average of the axis data is compared with the sent average to discard the data if they are not equal. If the data passes the check, angles between +20 and -20 are mapped to the screen area and cursor's absolute position is modified.

Finally, the main module connects all modules and inputs & outputs. It routes the angle data to the seven-segment display depending on the switch position. Sevensegment module is borrowed from my lab 5 design.

Results:

A simplified version of circuit diagram can be found below:

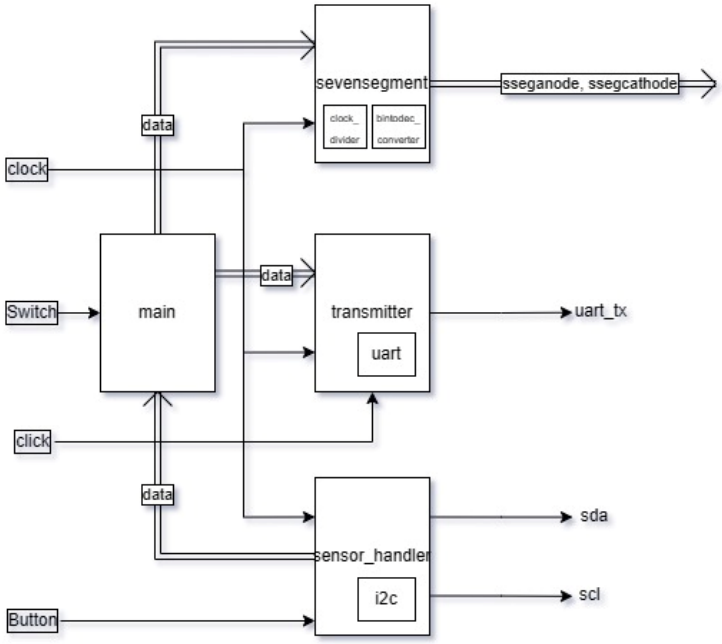

Figure 2.1: Circuit diagram.

RTL schematics of the modules can be found below:

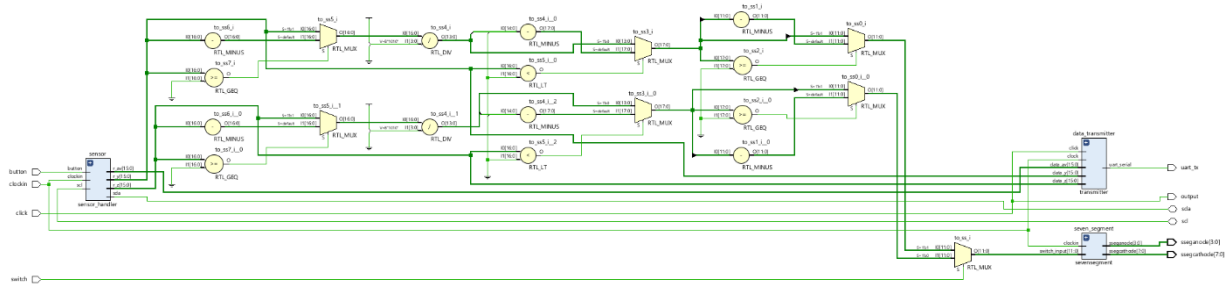

Figure 2.2: RTL of the main module.

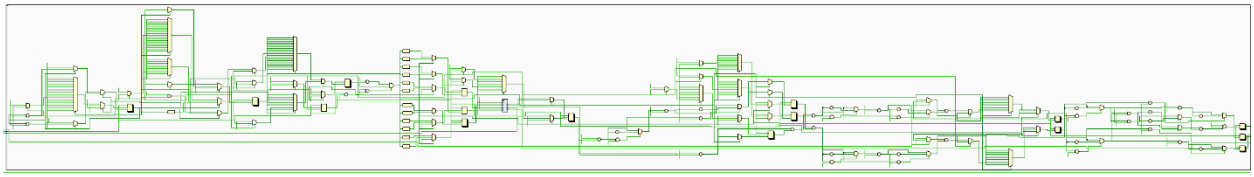

Figure 2.3: RTL of the sensor_handler module.

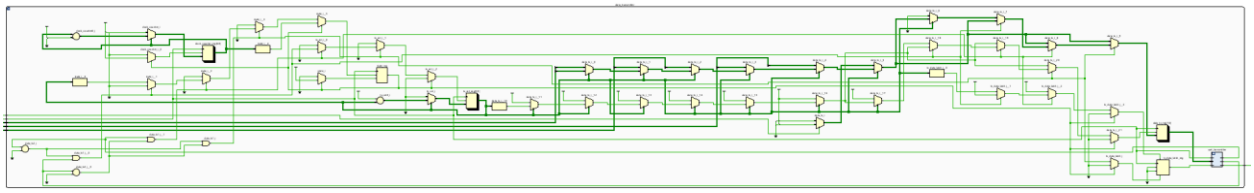

Figure 2.4: RTL of the transmitter module.

Photos of my ring and circuit design while they are in operation can be found below:

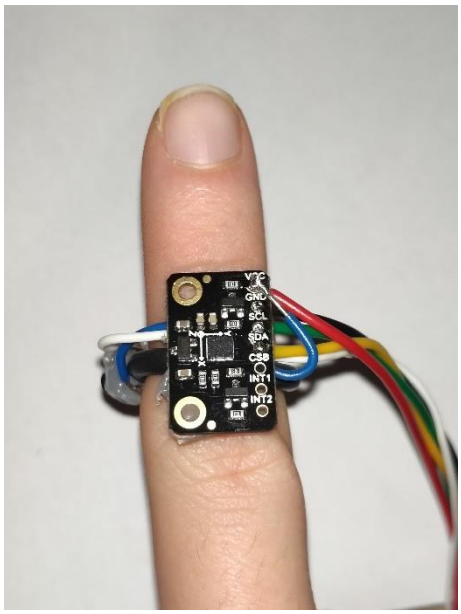

Figure 2.3: Ring design.

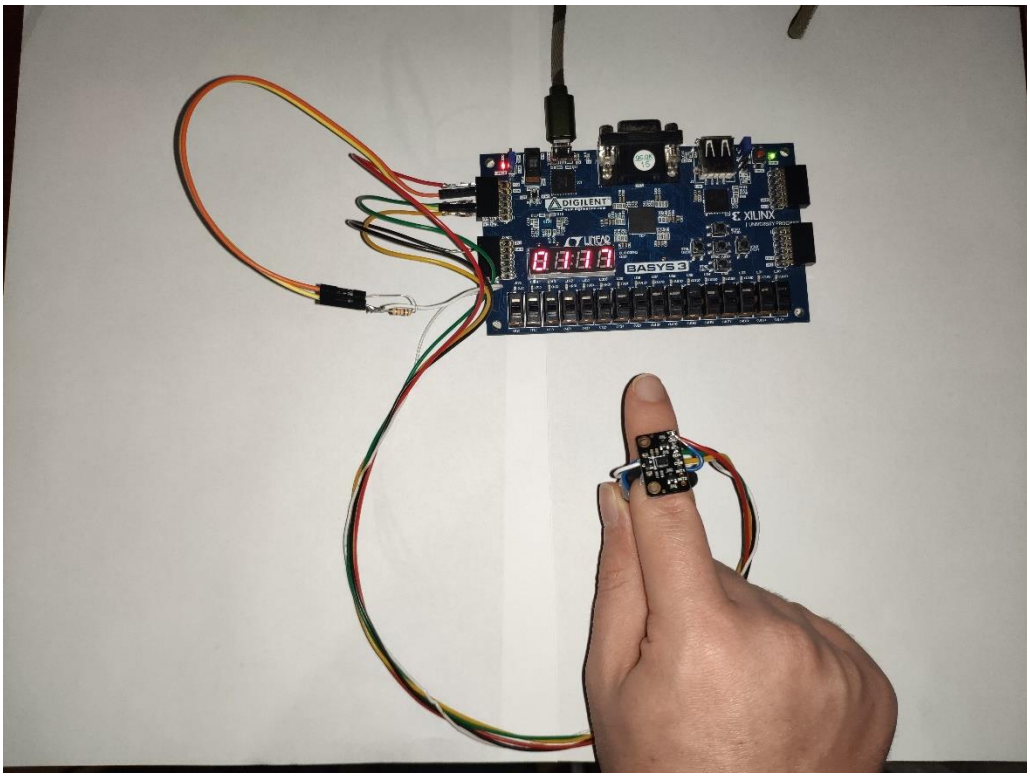

Figure 2.3: Working circuit.

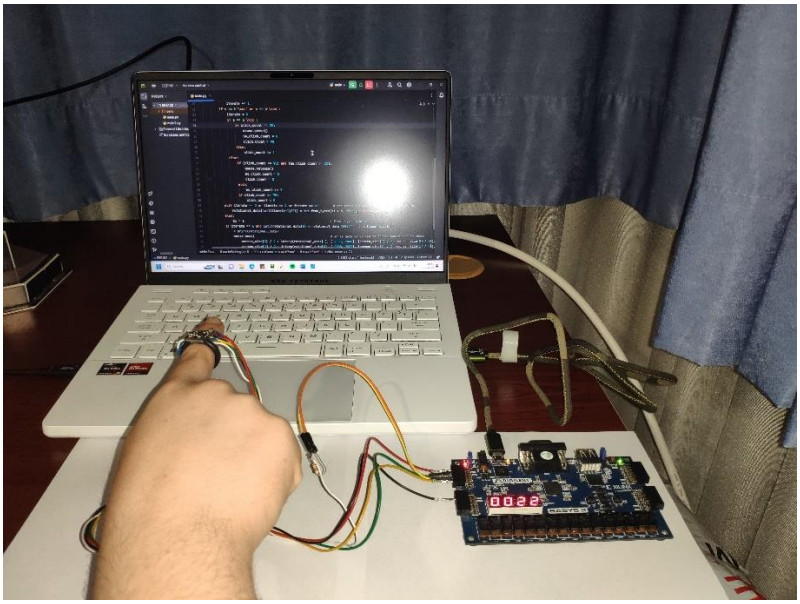

Figure 2.4: Circuit in operation.

As can be seen in the pictures, cables connecting the sensor to the Basys3 are specially made for this project. They can be soldered on one end and the other end has jumper connection. The push button with a 10 K pull-down resistor and the sensor are connected to Pmod ports on Basys3. More photos demonstrating the cursor's position at different finger directions are below.

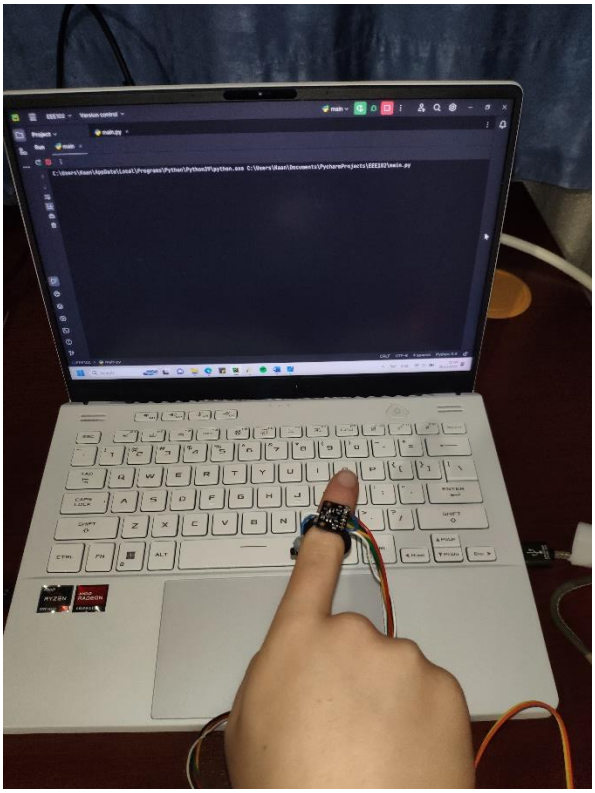

Figure 2.5: Moving cursor to the right.

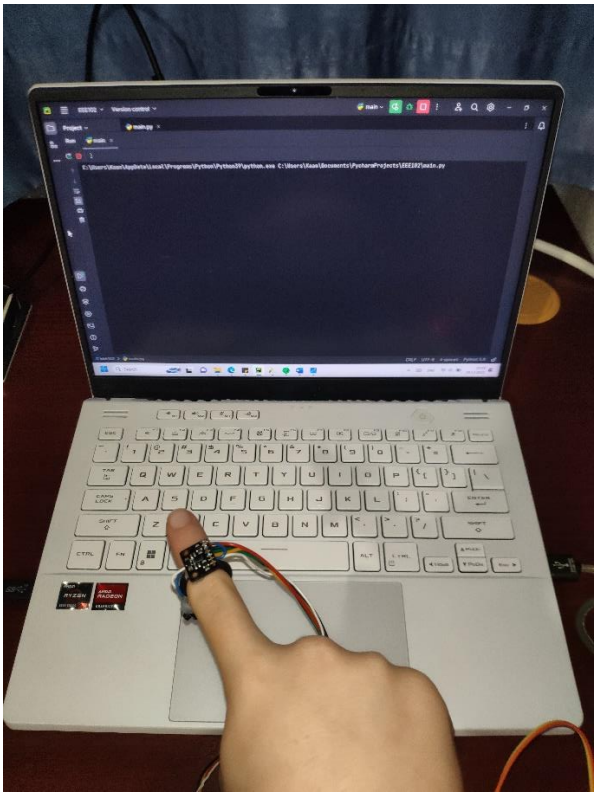

Figure 2.6: Moving cursor to the left.

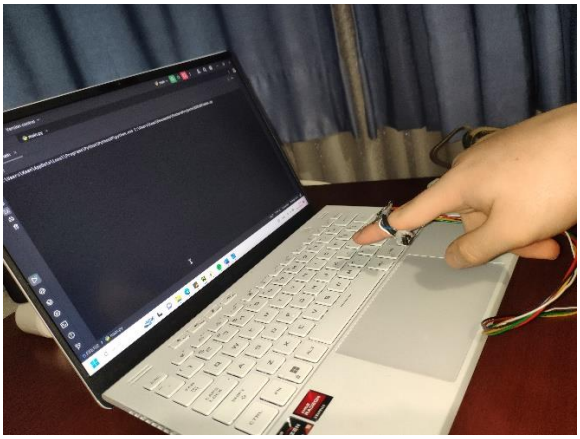

Figure 2.7: Moving cursor to the bottom.

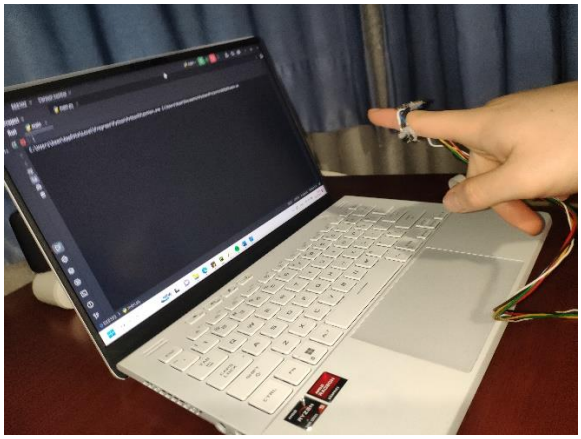

Figure 2.8: Moving cursor to the top.

Finally, the pictures below demonstrate functionality with the mouse click.

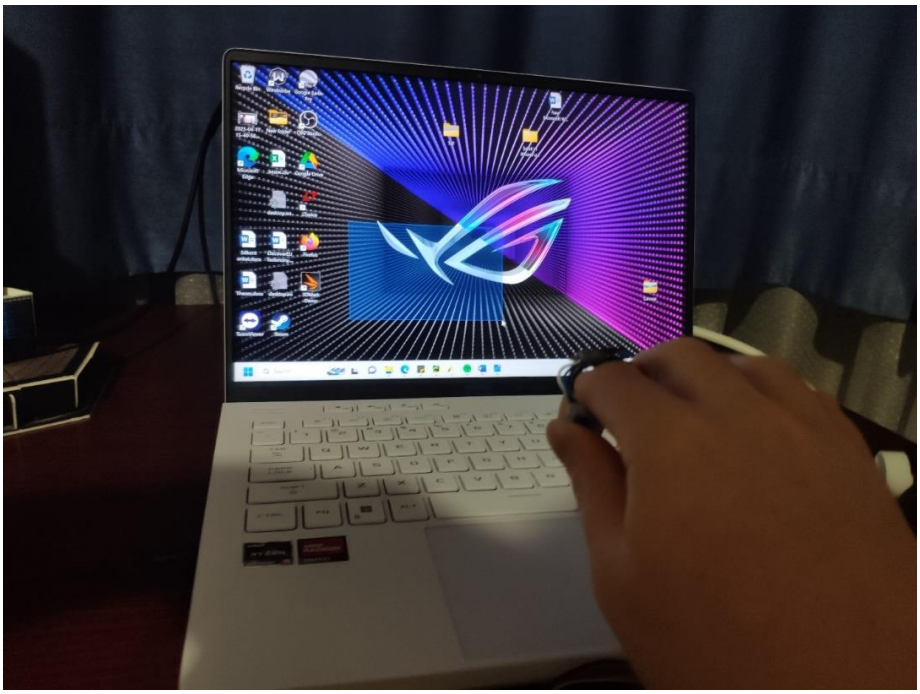

Figure 2.9: Dragging and selecting an area.

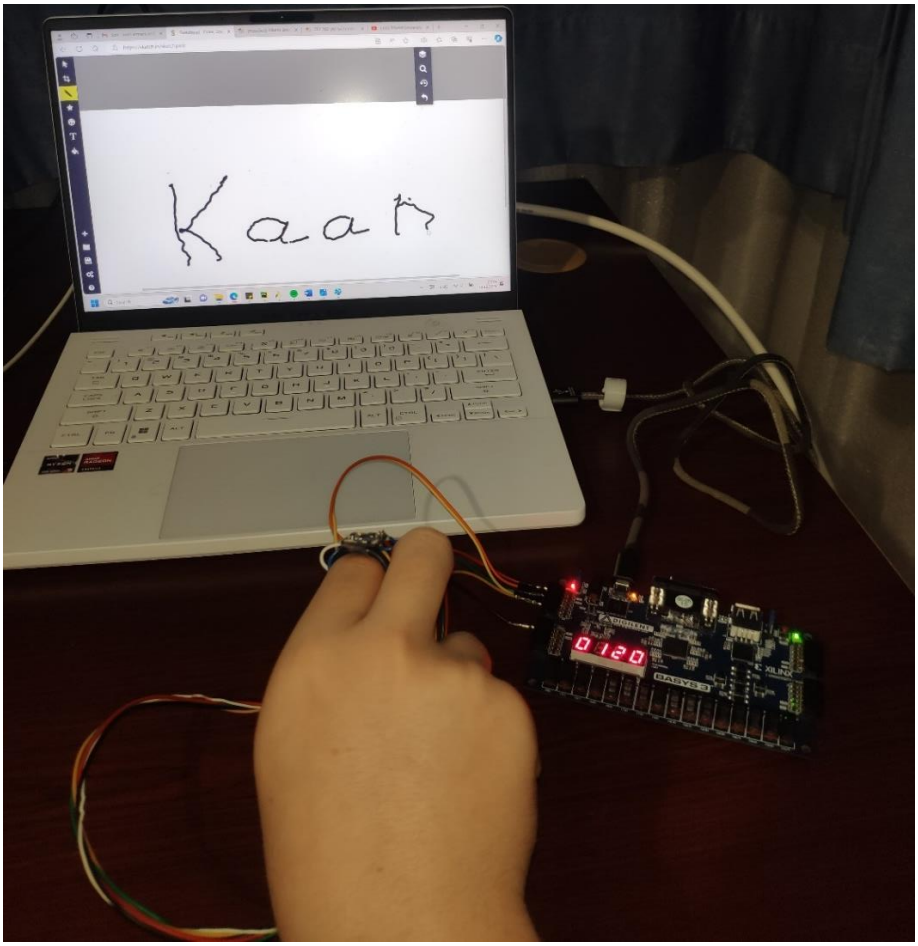

Figure 2.10: Writing my name.

Accurate finger tracking is successfully achieved with my design. With this project, a PC can be used without a mouse or a flat surface. Although design specifications are met, there is still room for improvement. The most significant shortcomings of the design are being susceptible to finger vibrations and cursor drifting away from the reference direction sometime after setting the reference direction. The amount of drift depends on all movements since reference reset thus, exact amount of drift cannot be specified. This issue can be solved by acquiring magnetometer and accelerometer data and then, correcting the drift by using the directions of gravitational acceleration and north pole as reference.

Conclusion:

I successfully designed a ring that tracks finger movements to move the cursor of a PC and perform left clicks by pressing the push button using Basys 3 and VHDL. Python is only used for receiving data over UART and sending mouse events. Acquisition, processing and transmission of the data is done on Basys3 with VHDL. I utilized VHDL skills that we learned throughout the semester. Finite state machines, registers and bus architecture are some topics we learned in the lectures and related to this project. In addition to course topics, I learned about communication protocols like I2C and UART when working on this project. By using an FPGA for this project, I was able to communicate with the sensor, process the sensor data, and transmit the data to the PC at the same time as FPGAs can be designed to run many operations in parallel.

References:

Bosch Sensortec GMBH. (2019, January). BMX 160 Data Sheet. https://img.dfrobot.com.cn/wiki/none/40e914cf5839ec7f4d10675f34f8f78c.pdf

I2C master (VHDL). Digikey. (2021, March 17). https://forum.digikey.com/t/i2c-master-vhdl/12797

Merrick, R. (2023, September 28). *UART in VHDL and Verilog for an FPGA*. Nandland. https://nandland.com/uart-serial-port-module/

Appendices:

Code 1: main.vhd

```
----------------------------------------------------------------------------------
--EEE102 Project
--Kaan Ermertcan
----------------------------------------------------------------------------------

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


entity main is
   Port ( clockin : in STD_LOGIC;                      -- 100 MHz Clock of Basys3
          sda   : INOUT  STD_LOGIC;                    -- Data pin for I2C
          scl   : INOUT  STD_LOGIC;                    -- Clock pin for I2C
          sseganode : out STD_LOGIC_VECTOR (3 downto 0);      -- Anodes of Seven Segment Display
          ssegcathode : out STD_LOGIC_VECTOR (7 downto 0);    -- Cathodes of Seven Segment Display
          switch: in std_logic;                        -- Sliding Switch on Basys 3
          uart_tx: out std_logic;                      -- Uart Tx pin going to FT2232HQ
          click: in std_logic;                         -- Button for mouse click
          button: in std_logic);                       -- Center button for reference reset
end main;


architecture Behavioral of main is

component sevensegment is
   Port ( switch_input : in std_logic_vector (11 downto 0);   -- Data to be displayed
          sseganode : out STD_LOGIC_VECTOR (3 downto 0);      -- Anodes of Seven Segment Display
          ssegcathode : out STD_LOGIC_VECTOR (7 downto 0);    -- Cathodes of Seven Segment Display
          clockin : in std_logic);                     -- Clock
end component;

component transmitter is
   Port ( data_z : in STD_LOGIC_VECTOR (15 downto 0);        -- Referenced angle data for Z axis to be transmitted via UART
          data_y : in STD_LOGIC_VECTOR (15 downto 0);        -- Referenced angle data for Y axis to be transmitted via UART
          data_av : in STD_LOGIC_VECTOR (15 downto 0);       -- Average of angles for Z and Y axis for redundancy check
          uart_serial : out STD_LOGIC;                       -- UART serial output
          click : in STD_LOGIC;                              -- Input for mouse click
          clock : in STD_LOGIC);                             -- Clock
end component;

component sensor_handler is
   Port ( clockin : in STD_LOGIC;                      -- Clock
          sda   : INOUT  STD_LOGIC;                    -- Data pin for I2C
          scl   : INOUT  STD_LOGIC;                    -- Clock pin for I2C
          button: in std_logic;                        -- Input for reference Reset
          r_av, r_y, r_z: out std_logic_vector(15 downto 0));  -- Referenced angle data for Y and Z axis and their average
(degrees*100)
end component;
```

```
signal to_ss: std_logic_vector(11 downto 0);                    -- Data signal going to seven segment display
signal r_av, r_y, r_z: std_logic_vector(15 downto 0):= (others => '0'); -- Referenced angle data for Y and Z axis and their average
(degrees*100)


begin
seven_segment: sevensegment Port Map(clockin => clockin, sseganode => sseganode, ssegcathode => ssegcathode,
switch_input => to_ss); -- Component handling Seven segment display
data_transmitter: transmitter Port map(clock => clockin, data_y => r_y, data_z => r_z, data_av => r_av, click => click, uart_serial
=> uart_tx);     -- Component handling transmission of angle data over UART
sensor: sensor_handler Port map(clockin => clockin, sda => sda, scl => scl, button => button, r_av => r_av, r_y => r_y, r_z =>
r_z); -- Component handling communication with the sensor and processing of sensor data

with switch select
to_ss <= std_logic_vector(to_unsigned(abs(to_integer(signed(r_y))/10),12)) when '1',        -- to see angle data for Y axis on seven
segment (degrees*10)
       std_logic_vector(to_unsigned(abs(to_integer(signed(r_z))/10),12)) when '0',        -- to see angle data for Z axis on seven
segment (degrees*10)
       (others => '0')                                    when others;

end Behavioral;
```

Code 2: sensor_handler.vhd

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

entity sensor_handler is
   Port ( clockin : in STD_LOGIC;                        -- Clock
          sda   : INOUT  STD_LOGIC;                      -- Data pin for I2C
          scl   : INOUT  STD_LOGIC;                      -- Clock pin for I2C
          button: in std_logic;                          -- Button for angle reference reset
          r_av, r_y, r_z: out std_logic_vector(15 downto 0)    -- Referenced angle data for Y and Z axis and their average
(degrees*100)
        );
end sensor_handler;

architecture Behavioral of sensor_handler is

Component i2c_master is
  Port(
    clk       : IN     STD_LOGIC;                    --system clock
    reset_n   : IN     STD_LOGIC;                    --active low reset
    ena       : IN     STD_LOGIC;                    --latch in command
    addr      : IN     STD_LOGIC_VECTOR(6 DOWNTO 0); --address of target slave
    rw        : IN     STD_LOGIC;                    --'0' is write, '1' is read
    data_wr   : IN     STD_LOGIC_VECTOR(7 DOWNTO 0); --data to write to slave
    busy      : OUT    STD_LOGIC;                    --indicates transaction in progress
    data_rd   : OUT    STD_LOGIC_VECTOR(7 DOWNTO 0); --data read from slave
    ack_error : BUFFER STD_LOGIC;                    --flag if improper acknowledge from slave
    sda       : INOUT  STD_LOGIC;                    --serial data output of i2c bus
```

```
    scl      : INOUT  STD_LOGIC);                  --serial clock output of i2c bus
END component;


signal enable, rw, busy, error: std_logic;                        -- to be connected to i2c_master
signal address: std_logic_vector(6 downto 0) := "1101000";         -- I2C Address of the sensor
signal data_r, data_w: std_logic_vector(7 downto 0);               -- data_r: Data read, data_w: Data to be written
type state_type is(hold, read, setup);                            -- setup: configures the sensor, read: reads the data from the sensor,
hold: waits for defined nb of clock cycles
signal state: state_type:= setup;
signal busy_prev: std_logic := '0';                               -- Busy signal at the previous rising edge
signal data: std_logic_vector(15 downto 0) := (others => '0');      -- For keeping the first byte(s) of data if the data is more than
one byte until all bytes are read
signal gyr_x, gyr_y, gyr_z: std_logic_vector(23 downto 0) := (others => '0'); -- Gyroscope data (degrees/s *1000)

begin

IIC: i2c_master Port map (clk => clockin, reset_n => '1', ena => enable, addr => address, rw => rw,     -- Component handling I2C
transmission
data_wr => data_w, busy => busy, data_rd => data_r, ack_error => error, sda => sda, scl => scl);

process(clockin, button)

variable dir_x, dir_y, dir_z: integer:= 0;                        -- Keeps the referanced angle data (degrees*1000)
variable busy_cnt: INTEGER range 0 to 23:= 0;                     -- Keeps the transmission index
variable clock_counter: INTEGER range 0 to 100000000:= 100000000;         -- Keeps the clock count
variable stime: integer range 0 to 654311385:= 0;                 -- Keeps the time of data
variable last_stime: integer range 0 to 654311385:= 0;            -- Keeps the previous time of data

constant cmd_addr: std_logic_vector(7 downto 0) := "01111110";            -- Address of command register
constant softreset_cmd: std_logic_vector(7 downto 0) := "10110110";       -- Soft Reset Command
constant acc_pwr_normal_cmd: std_logic_vector(7 downto 0) := "00010001";    -- Accelerometer Normal Power Mode
Command
constant gyr_pwr_normal_cmd: std_logic_vector(7 downto 0) := "00010101";    -- Gyroscope Normal Power Mode Command
constant mag_pwr_normal_cmd: std_logic_vector(7 downto 0) := "00011001";    -- Magnetometer Normal Power Mode
Command
constant gyr_conf: std_logic_vector(7 downto 0) := "00101100";            -- Gyroscope Configuration (odr=1600Hz)
constant gyr_range: std_logic_vector(7 downto 0) := "00000100";           -- Gyroscope Range = 125dps
constant gyr_multiplier_125dps: integer := 3811; -- *10^-6                -- Multiplier for gyroscope data for 125 dps range
(sensor_data * gyr_multiplier_125dps = dps*10^6)
constant gyr_conf_addr: std_logic_vector(7 downto 0) := "01000010";       -- Address of gyroscope configuration register
constant gyr_range_addr: std_logic_vector(7 downto 0) := "01000011";      -- Address of gyroscope range register
constant gyr_data_addr: std_logic_vector(7 downto 0) := "00001100";       -- Address of gyroscope data register
constant acc_conf_addr: std_logic_vector(7 downto 0) := "01000000";       -- Address of accelerometer configuration register
constant acc_range_addr: std_logic_vector(7 downto 0) := "01000001";      -- Address of accelerometer range register
constant acc_conf: std_logic_vector(7 downto 0) := "00101100";            -- Accelerometer configuration
constant acc_range: std_logic_vector(7 downto 0) := "00000011";           -- Accelerometer Range = 2G

begin
if rising_edge(clockin) then

r_y <= std_logic_vector(to_signed(dir_y/10,16));                  -- Referenced angle data for Y and Z axis and their average
(degrees*100)
r_z <= std_logic_vector(to_signed(dir_z/10,16));
```

```
r_av <= std_logic_vector(to_signed((dir_y/10 + dir_z/10)/2, 16));

if button = '1' then                                    -- Reset the reference angle
dir_x := 0;
dir_y := 0;
dir_z := 0;
end if;

case state is
WHEN setup =>
   busy_prev <= busy;                        -- capture the value of the previous i2c busy signal
   IF(busy_prev = '0' AND busy = '1') THEN        -- i2c busy just went high
      busy_cnt := busy_cnt + 1;                -- counts the times busy has gone from low to high during transaction
   END IF;
   CASE busy_cnt IS                          -- busy_cnt keeps track of which command we are on
   WHEN 0 =>                                 -- no command latched in yet
      enable <= '1';                         -- initiate the transaction
      rw <= '0';                             -- command is a write
      data_w <= cmd_addr;                    -- command adress to be written
   WHEN 1 =>                                 -- 1st busy high: command 1 latched, okay to issue command 2
      data_w <= softreset_cmd;                -- Command to perform soft reset of the sensor
   WHEN 2 | 5 | 8 | 11 | 14 | 17 | 20 =>         -- Wait for required delay between commands
      enable <= '0';                         -- End transaction
      IF(busy = '0') THEN                    -- Wait until the acknowledge bit
      if clock_counter = 0 then
      busy_cnt := busy_cnt + 1;              -- Using the same variable to track the waiting states
      clock_counter := 100000000;              -- Wait for 1 second
      else
      clock_counter := clock_counter - 1;
      end if;
      end if;
   WHEN 3 =>
      enable <= '1';                         -- initiate the transaction
      data_w <= cmd_addr;                    -- Command register to be written
   WHEN 4 =>
      data_w <= acc_pwr_normal_cmd;             -- Command to set Acc power to normal
   WHEN 6 =>
      enable <= '1';                         -- initiate the transaction
      data_w <= gyr_conf_addr;                -- gyro configuration register to be written
   WHEN 7 =>
      data_w <= gyr_conf;                    -- Gyroscope configuration is set
   WHEN 9 =>
      enable <= '1';                         -- initiate the transaction
      data_w <= gyr_range_addr;                -- Gyro range register to be written
   WHEN 10 =>
      data_w <= gyr_range;                    -- Set gyro range
   WHEN 12 =>
      enable <= '1';                         -- initiate the transaction
      data_w <= acc_conf_addr;                -- Acc configuration register to be written
   WHEN 13 =>
      data_w <= acc_conf;                    -- Accelerometer configuration is set
   WHEN 15 =>
      enable <= '1';                         -- initiate the transaction
      data_w <= acc_range_addr;                -- Acc range register to be written
```

```
   WHEN 16 =>
      data_w <= acc_range;                    -- Accelerometer range is set
   WHEN 18 =>
      enable <= '1';                          -- initiate the transaction
      data_w <= cmd_addr;                     -- command register to be written
   WHEN 19 =>
      data_w <= gyr_pwr_normal_cmd;           -- Set Gyroscope power mode to normal
   WHEN 21 =>
      enable <= '1';                          -- initiate the transaction
      data_w <= cmd_addr;                     -- command register to be written
   WHEN 22 =>
      data_w <= mag_pwr_normal_cmd;           -- Set Magnetometer power mode to normal
   WHEN 23 =>
      enable <= '0';                          -- End transaction
      IF(busy = '0') THEN
         busy_cnt := 0;                       -- Reset busy_cnt
         state <= hold;                       -- Go to hold state
      end if;
   WHEN OTHERS => NULL;
   end case;


WHEN read =>
   busy_prev <= busy;                         -- capture the value of the previous i2c busy signal
   IF(busy_prev = '0' AND busy = '1') THEN    -- i2c busy just went high
      busy_cnt := busy_cnt + 1;               -- counts the times busy has gone from low to high during transaction
   END IF;
   case busy_cnt is
   WHEN 0 =>                                  -- no command latched in yet
      enable <= '1';                          -- initiate the transaction
      rw <= '0';                              -- command is a write
      data_w <= gyr_data_addr;                -- Gyroscope data register address to be read
   WHEN 1 =>
      rw <= '1';                              -- Switch to read
   WHEN 2 | 4 | 6 | 14 =>
      IF(busy = '0') THEN                     -- Wait until the acknowledge bit
      data (7 downto 0) <= data_r;            -- First byte to be kept for one clock cycle
      end if;
   WHEN 3 =>
      IF(busy = '0') THEN                     -- Convert X axis gyro data(full range (+-125 dps) corresponding to 16-bit signed number)
to degrees/s *1000
      gyr_x <= std_logic_vector(to_signed((gyr_multiplier_125dps*to_integer(signed(data_r&data(7 downto 0)))/1000), 24));
      end if;
   WHEN 5 =>
      IF(busy = '0') THEN                     -- Convert Y axis gyro data(full range (+-125 dps) corresponding to 16-bit signed number)
to degrees/s *1000
      gyr_y <= std_logic_vector(to_signed((gyr_multiplier_125dps*to_integer(signed(data_r&data(7 downto 0)))/1000), 24));
      end if;
   WHEN 7 =>
      IF(busy = '0') THEN                     -- Convert Z axis gyro data(full range (+-125dps) corresponding to 16-bit signed number) to
degrees/s *1000
      gyr_z <= std_logic_vector(to_signed((gyr_multiplier_125dps*to_integer(signed(data_r&data(7 downto 0)))/1000), 24));
      end if;
   WHEN 15 =>
```

```
      IF(busy = '0') THEN                    -- Keep second byte of sensor time
      data (15 downto 8) <= data_r;
      end if;
   WHEN 16 =>
      enable <= '0';                         -- End transaction
      IF(busy = '0') THEN
      last_stime := stime;                   -- Keep time of the previous data
      stime:= (39*to_integer(unsigned(data_r&data)));     -- calculate sensor time (us)
      dir_x := dir_x + ((stime-last_stime)*to_integer(signed(gyr_x))/1000000);    -- calculate change in rotation of x axis
      dir_y := dir_y + ((stime-last_stime)*to_integer(signed(gyr_y))/1000000);    -- calculate change in rotation of y axis
      dir_z := dir_z + ((stime-last_stime)*to_integer(signed(gyr_z))/1000000);    -- calculate change in rotation of z axis
      busy_cnt := 0;
      state <= hold;                                      -- Hold for defined nb of clock cycles
      end if;
   WHEN others => NULL;
   end case;


When hold =>                                  -- Do nothing for a specified nb of clock cycles and go to read after that
   if clock_counter = 0 then
      clock_counter := 100000000;
      state <= read;
      else
      clock_counter := clock_counter - 1000;
      end if;
end case;

end if;
end process;

end Behavioral;
```

Code 3: transmitter.vhd

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

entity transmitter is
   Port ( data_z : in STD_LOGIC_VECTOR (15 downto 0);        -- Referenced angle data for Z axis to be transmitted via UART
          data_y : in STD_LOGIC_VECTOR (15 downto 0);        -- Referenced angle data for Y axis to be transmitted via UART
          data_av : in STD_LOGIC_VECTOR (15 downto 0);       -- Average of angles for Z and Y axis for redundancy check
          uart_serial : out STD_LOGIC;                       -- UART serial output
          click : in STD_LOGIC;                              -- Input for mouse click
          clock : in STD_LOGIC);                             -- Clock
end transmitter;

architecture Behavioral of transmitter is
type state_type is(hold, transmit);                          -- Hold: Holds for defined nb of clock cycles, Transmit: transmits the data over
UART
signal state: state_type:= hold;
signal data_tx: std_logic_vector(7 downto 0);                -- A byte of data to be transmitted on the next transmission
signal tx_data_latch: std_logic;                             -- For latching a byte of data to UART component and starting transmission
```

```
signal tx_active: std_logic;                          -- Signal of UART components state: 1 -> Transmission in progress 0 -> Waiting for
transmission start signal
signal tx_done: std_logic;                            -- Signal coming from UART component signifying copmletion of transmission: 1 -
> Transmission is completed
component uart is
generic (
   g_CLKS_PER_BIT : integer := 100     -- 100MHz / 1 MHz       -- Clock / UART Baud rate
   );
port (
   Clk      : in  std_logic;                          -- Clock
   TX_DV    : in  std_logic;                          -- For latching a byte of data to UART component and starting transmission
   TX_Byte  : in  std_logic_vector(7 downto 0);            -- A byte of data to be transmitted on the next transmission
   TX_Active : out std_logic;                          -- Signal of UART components state: 1 -> Transmission in progress 0 -> Waiting
for transmission start signal
   TX_Serial : out std_logic;                          -- UART serial output
   TX_Done  : out std_logic                            -- Signal coming from UART component signifying copmletion of transmission: 1 -
> Transmission is completed
   );
end component;
begin
uart_transmitter: uart Port Map(Clk => clock, TX_DV => tx_data_latch, TX_Byte => data_tx, TX_Active => tx_active, TX_Serial =>
uart_serial,TX_Done => tx_done); -- Component for handling uart transmission of a byte of data

process(clock)
variable tx_cnt: INTEGER range 0 to 6:= 0;               -- Keeping the index of transmitted data
variable clock_counter: INTEGER range 0 to 100:= 100;      -- Keeping the clock count for hold state
begin
if rising_edge(clock) then
if state = transmit then
   if tx_cnt = 0 then                          -- Select which data to be transmitted according to tx_cnt
      if click = '1' then
         data_tx <= "11001100";                  -- Transmit mouse click
      elsif click = '0' then
         data_tx <= "10101010";
      end if;
   elsif tx_cnt = 1 then
      data_tx <= data_y(15 downto 8);              -- Transmit MSB of Y axis data
   elsif tx_cnt = 2 then
      data_tx <= data_y(7 downto 0);               -- Transmit LSB of Y axis data
   elsif tx_cnt = 3 then
      data_tx <= data_z(15 downto 8);              -- Transmit MSB of Z axis data
   elsif tx_cnt = 4 then
      data_tx <= data_z(7 downto 0);               -- Transmit LSB of Z axis data
   elsif tx_cnt = 5 then
      data_tx <= data_av(15 downto 8);             -- Transmit MSB of data average
   elsif tx_cnt = 6 then
      data_tx <= data_av(7 downto 0);              -- Transmit LSB of data average
   end if;

   if tx_active = '0' and tx_done = '0' then          -- start transmission
      tx_data_latch <= '1';
   elsif tx_active = '1' and tx_done = '0' then       -- transmission in progress
      tx_data_latch <= '0';
   elsif tx_active = '0' and tx_done = '1' then       -- Transmission completed
```

```
      if tx_cnt = 6 then
         tx_data_latch <= '0';
         data_tx <= (others => '0');                -- end transmission
         tx_data_latch <= '0';
         tx_cnt:= 0;
         state <= hold;                          -- Hold
      else
         tx_cnt := tx_cnt + 1;                   -- Continue with the next transmission
      end if;
   else
      tx_data_latch <= '0';
   end if;

elsif state = hold then                          -- Hold for 100 clock cycles
   if clock_counter = 0 then
      clock_counter := 100;
      state <= transmit;                         -- Transmit again
   else
      clock_counter := clock_counter - 1;
   end if;
end if;
end if;
end process;
end Behavioral;
```

Code 4: main.py (Python Code)

```
import serial
import mouse
from numpy import interp

ser = serial.Serial('COM4', 1000000, timeout=0)  # Open serial port
s1 = ser.read()                          # s1 keeps first byte, s keeps second byte
s = ser.read()
iterate = 6                              # keeps the byte index
rotational_data = [0, 0, 0]                  # keeps the last received angle data
screen_size = [1280, 800]                  # size of the screen space
click_count = 0
no_click_count = 0
while True:
   if ser.inWaiting() > 0:                  # If data is received
      s = ser.read()                    # read a byte
      if iterate == 6:                 # cycle through data index
         iterate = 0
      else:
         iterate += 1
      if s == b'\xcc' or s == b'\xaa':
         iterate = 0
         if s == b'\xcc':
            if click_count == 10:
               mouse.press()
               no_click_count = 0
               click_count = 90
```

```
            else:
                click_count += 1
        else:
            if (click_count >= 90) and (no_click_count >= 10):
                mouse.release()
                no_click_count = 0
                click_count = 0
            else:
                no_click_count += 1
            if click_count <= 90:
                click_count = 0
    elif iterate == 2 or iterate == 4 or iterate == 6:       # two bytes are received and converted to int
        rotational_data[int((iterate-1)/2)] = int.from_bytes(s1 + s, "big", signed=True)
    else:
        s1 = s                                  # first byte is kept
    if iterate == 6 and int((rotational_data[0] + rotational_data[1])/2) == rotational_data[2]:
        # print(rotational_data)
        mouse.move(                             # angle data is mapped to screen and cursor is moved
            screen_size[0] / 2 + interp(rotational_data[1], [-2000, 2000], [screen_size[0] / 2, -screen_size[0] / 2]),
            screen_size[1] / 2 + interp(rotational_data[0], [-2000, 2000], [screen_size[1] / 2, -screen_size[1] / 2]),
            absolute=True, duration=0)
```