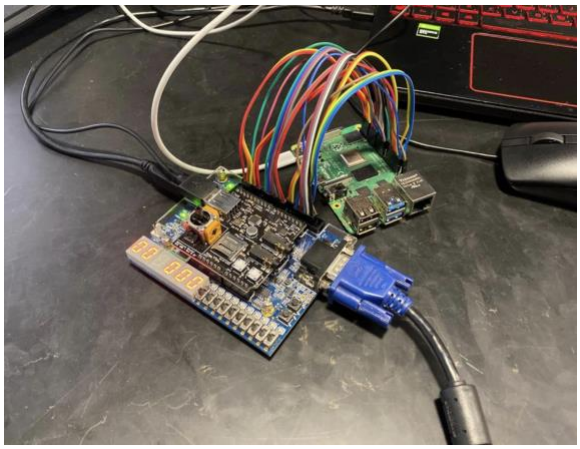
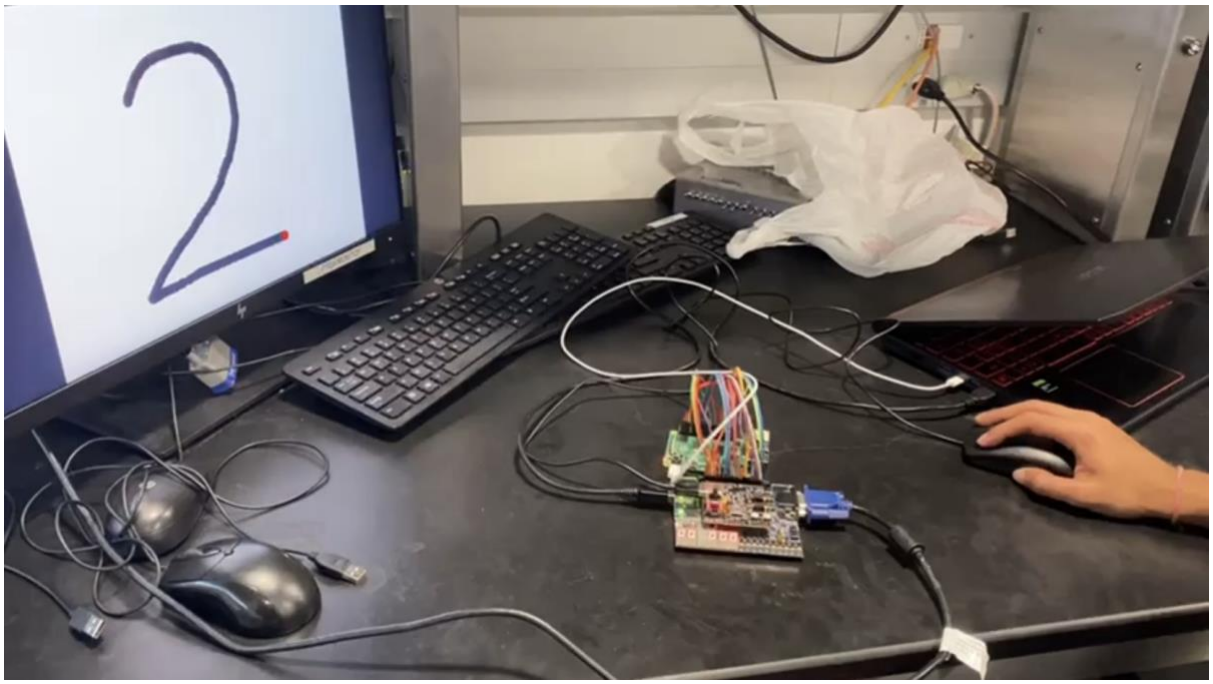


# ECE 385 FINAL PROJECT REPORT

## *HANDWRITING DETECTOR (FPGA + RASPBERRY PI)*

Shubham Gupta (sg49) & Devul Nahar (danahar2)

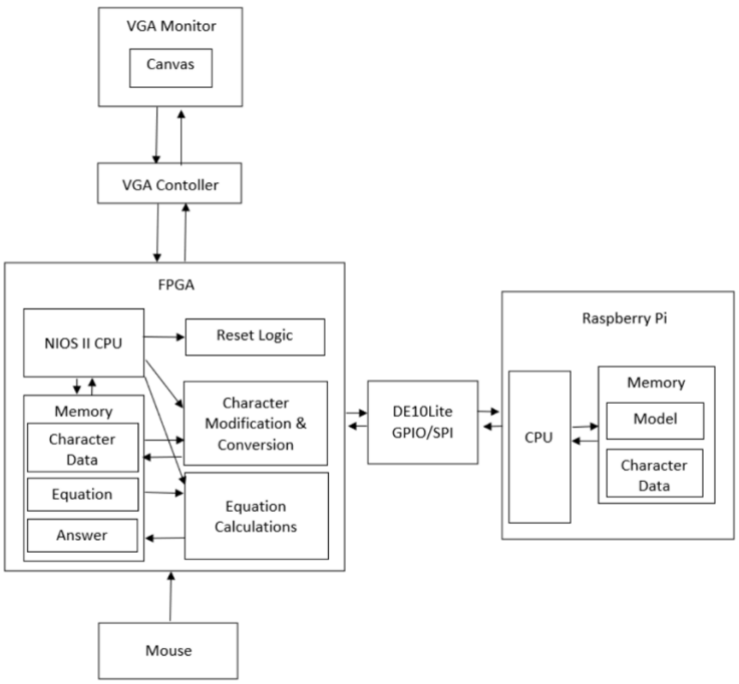
May 11, 2022



## Introduction:

The purpose was to design and build a project which can recognize human handwriting consisting of digits from 0 to 9 and a few operators like +, -, x, and /. By doing so, we can solve simple mathematical expressions like “9 + 3”. The FPGA would be responsible for taking user input and displaying output on the VGA display. Meanwhile, our Rasberry Pi 4b would be responsible for handwriting interpretation via the TensorFlow library for Python. The Rasberry Pi would be the slave and FPGA is the master device, we will use SPI to communicate between the two. The FPGA’s OCM would store CNN model as well as the array of pixels between 0 and 1. The user will draw on the VGA display via mouse. The Raspberry Pi and the FPGA talk to each other via GPIO pins and an FSM is maintained to keep track of states in both machines.

## Block Diagram:




## Platform Designer:

This is the tool we used to generate our IP modules and link the inputs and outputs together. We have instantiated the on-chip memory of the FPGA and the NIOS II initially. This is the heart of our design. Then we have added PIOs which are parallel input outputs for our peripherals. Now, to store our C code we then add a SDRAM. Since SDRAM is physically away from the chip to account for the delay of 1ns in the clock cycle we have added a PLL. C code is written to interact with mouse via MAX2431E chip and we use OCM as VRAM.

| Components | Description |
|---|---|
| Clk_0 | Used to generate clock for the FPGA, SDRM, as well as the MAX3421E. |

| | |
|---|---|
| Nios2_gen2_0 | The CPU layer that behaves like slc3 but is more robust. This CPU controls all functionality within the machine. It is responsible for parsing and acting on instructions that it fetches from memory. |
| Sdram | SDRAM is the external memory chip connected to the FPGA board. The NIOS II is stored here instead of the onchip memory. Instructions as well as data is fetched from here. |
| Sdram_pll | Since sdram is further away from the onchip memory and because it requires a clock signal, the PLL handles the timing offsets that are caused due to board layout. This component is very important as the SDRAM requires precise timings due to the fact that it is made up of capacitors. |
| Jtag_uart_0 | JTAG UART allows serial character communications streams between a host PC and the FPGA. In addition, the JTAG UART core is also connected to the Interrupt controller as transferring text over the console is slow and we don't want to slow down the CPU. |
| Timer_0 | This block handles the timer for the USB Port. The timer is essential to the USB driver in order to keep track any time-out that the USB needs. |
| Spi_0 | SPI is known as the Serial Port Interface, which allows us to read and write to the register on the MAX3421E chip. |
| Key | Key is the input to FPGA. It specifies the key entered on the keyboard. Key is 2 bits to specify the 4 different relevant keys the user can enter (WSAD). |

| | |
|---|---|
| Keycode | The 2 bits generated by the Key block is received by the keycode, which generates a 8 bit ASCII code. This ascii code is then used to change the direction of the ball on the screen. |
| Onchip_memory2_0 | Onchip memory is a small memory block that resides within the FPGA chip. Since onchip memory is very close to the actual FPGA, it is faster access times. We don't use this in this lab, but we could for future labs. |
| Sysid_qsys_0 | Used to check the system ID so that it can maintain the compatibility between software and hardware. It functions by giving a serial number which the software cross checks with the software. |
| Hex_digits_pio | The PIO address for Hex displays. A set of addresses are assigned for this PIO, so that we can use them in software. |
| leds_pio | The PIO address for LED displays. A set of addresses are assigned for this PIO, so that we can use them in software. |
| Usb_gpx | Connects the circuit to the USB port, which is connected to the keyboard. |
| Usb_rst | Provides the resetting signal from the USB port. |
| Usb_irq | Handles the interrupt controller for the USB port. |

This is the same as lab 6.2, we now use Keycode to send mouse button data.

## **Design Resources and Statistics:**

| | |
|---|---|
| LUT | 3426 |
| DSP | 10 |
| Memory (BRAM) | 405,504 |
| Flip-Flop | 2,585 |
| Frequency | 83.25 Mhz |
| Static Power | 96.57 mW |
| Dynamic Power | 67.16 mW |
| Total Power | 190.32 mW |

**Desciptions of Modules:**

**Module**: VGA_controller.sv

**Inputs**: Clk, Reset

**Outputs**: hs, vs, pixel_clk, blank, sync, [9:0] DrawX, [9:0] DrawY

**Description**: This module handles the generation of the VGA signals. One of its key's features are to make a 25 Mhz clock from the 50 Mhz clock. This is done by using a always_ff block. Then this module keeps track of the horizontal and vertical pixel coordinates. It also resets them if they reach the end of horizontal and vertical pixel bounds. Another function is to generate the horizontal and vertical pixel sync pulse. Lastly, we use if else block to generate an active low blank signal.

**Purpose**: It is the key component that generates the VGA signals. It provides us the DrawX and DrawY cordinates which are extremely important to draw anything on the screen. This module also generates a 25 MHz pixel clock. It provides info for the blanking interval as well, so that we do not draw anything.

---

**Module**: HexDriver.sv

**Inputs**: [3:0] In0

**Outputs**: [6:0] Out0

**Description**: We use a UNIQUE switch case to map the binary values to the hexadecimal display. Given to us from previous labs.

**Purpose**: It maps the binary values to the hexadecimal LED display. Used for displaying values stored in registers. For user interface on the FPGA.

**Module**: Color_Mapper.sv

**Inputs**: [9:0] BallX, [9:0] BallY, [9:0] DrawX, [9:0] DrawY, [9:0] Ball_size

**Outputs**: [7:0] Red, [7:0] Green, [7:0] Blue

**Description**: This module generates the appropriate color values (Red, Green, Blue) based on the DrawX and DrawY along with BallX and BallY. We use if else blocks to check if the ball is supposed to be drawn on the given DrawX and DrawY coordinates, based on ball size, BallY, and BallX inputs. If the ball is not supposed to be drawn, then we simply draw the background.

**Purpose**: It is primarily used to generate the color (RGB) for the pixel depending on X and Y location of the pixel.

---

**Module**: Cursor.sv

**Inputs**: Reset, frame_clk, [7:0] keycode

**Outputs**: [9:0] BallX, [9:0] BallY, [9:0] BallS

**Description**: This is the module that generates the X and Y coordinates of the cursor. We used always_ff to calculate new positions of the ball based on the boundary conditons and update the motion in X and Y direction. Followed by updating the X and Y coordinates based on motion X and Y, along with step X and Y. It is dependent on the X and Y displacement given by keycode. We have also taken care of the edge cases where the keycode is continuously pressed and the ball should not escape the screen.

**Purpose**: Its primary purpose is to generate the new position (X and Y) for the cursor based on the keycode (mouse displacement) and boundary conditions.

---

**Module**: Canvas.sv

**Inputs**: [9:0] BallX, BallY, DrawX, DrawY, Ball_size, mouseButton, blank, pixel_clk, clk, [18:0] addressFromPi, SW

**Outputs**: [7:0] Red, Green, Blue, dataToPi, restartToPi, charDoneToPi, solveToPi

**Description**: Module that reads and writes to OCM initialized VRAM based on control signals from the Raspberry Pi/ switches.

**Purpose**: Provide the functionality of whiteboard to write on the screen with mouse. Supports drawing, eraser, and a complete reset.

---

**Module**: final_project_soc.v

**Inputs**: clk_clk, [1:0] key_external_connection_export, reset_reset_n, [15:0]

sdram_wire_dq, spi0_MISO, usb_gpx_export, usb_irq_export, [18:0] addressFromPi, answerReadyFromPi

**Outputs**: sdram_clk_clk, [12:0] sdram_wire_addr, [1:0] sdram_wire_ba, sdram_wire_cas_n, sdram_wire_cke, sdram_wire_cs_n, [1:0] sdram_wire_dqm, sdram_wire_ras_n sdram_wire_we_n, spi0_MOSI, spi0_SCLK, spi0_SS_n, usb_rst_export, dataToPi, restartToPi, charDoneToPi, solveToPi

**Description:** We have instantiated the following IPs in the platform designer which make up this module. This top-level module is so that we can connect inputs and outputs of these IPs. The IPs include: NIOS ll/e, OCM, PIOs, JTAG UART, SDRAM, PPL for SDRAM, SPI, TIMER, and USB INTERRUPTS. NIOS ll/e is our processor. JTAG UART is for debugging purposes that we use to debug the NIOS ll. SDRAM is

the primary memory where our compiled code is stored as instructions. PLL is to accommodate the physical time offset due to the path difference between the SDRAM and the OCM. SPI is the IP we used to communicate between the USB shield and NIOS ll. USB interrupts are so that we can send interrupt requests whenever mouse sends X and Y displacement. PIOs are used to display the content to HexDrivers as memory mapped I/O. Same goes for SWs and LEDS.

**Purpose**: Our top-level SOC.

**SystemVerilog Code Description:**

**Finalproject.sv**

```
vga_controller myVGA (.Clk(MAX10_CLK1_50),
                      .Reset(Reset_h),
                      .hs(VGA_HS),
                      .vs(VGA_VS),
                      .pixel_clk(VGA_Clk),
                      .blank(blank),
                      .sync(sync),
                      .DrawX(drawxsig),
                      .DrawY(drawysig));
```

Initializing VGA controller here.

```
cursor myCursor(.Reset(Reset_h),
                .frame_clk(VGA_VS),
                .keycode(keycode),
                .BallX(ballxsig),
                .BallY(ballysig),
                .BallS(ballsizesig),
                .mouseX(mousex),
                .mouseY(mousey),
                .mouseButton(mousebutton));
```

Initializing the Cursor module here.

```
canvas myCanvas(.BallX(ballxsig),
                .BallY(ballysig),
                .mouseButton(mousebutton),
                .DrawX(drawxsig),
                .DrawY(drawysig),
                .Ball_size(ballsizesig),
                .Red(Red),
                .Green(Green),
                .Blue(Blue),
                .blank(blank),
                .pixel_clk(VGA_Clk),
                .clk(MAX10_CLK1_50),
                .addressFromPi(addressFromPi),
                .dataToPi(dataToPi),
                .restartToPi(restartToPi),
                .charDoneToPi(charDoneToPi),
                .solveToPi(solveToPi),
                .LEDR(LEDR),
                .SW(SW));
```

Initializing the canvas here.

**Canvas.sv:**

```
logic cursor_on, freezeCursor;

int DistX, DistY, Size;
 assign DistX = DrawX - BallX;
assign DistY = DrawY - BallY;
assign Size = Ball_size;

always_comb
begin:cursor_on_proc
    if ( ( DistX*DistX + DistY*DistY) <= (Size * Size) && SW[0] == 0)
        cursor_on = 1'b1;
    else
        cursor_on = 1'b0;
end
```

To decide if the cursor should be ON/OFF.

```
logic [18:0] addrA, addrB;
logic writeEnA, outputA, outputB, setDataA;

assign addrA = DrawX + (DrawY * 640);

assign addrB = addressFromPi;
assign dataToPi = outputB;

logic my_restartToPi, my_charDoneToPi, my_solveToPi;

assign restartToPi = SW[2];
assign charDoneToPi = my_charDoneToPi;
assign solveToPi = my_solveToPi;

assign LEDR = SW;

logic submit, next, clearScreen;

assign submit = SW[1];
assign next = SW[0];
```

Here we calculate the VRAM address and initialize the switches to control signals.

```
always_ff @ (posedge pixel_clk) begin
    if(next == 1 && submit == 0 && addressFromPi != 19'h4afff) begin
        freezeCursor = 1;
        my_charDoneToPi = 1;
        clearScreen = 0;
    end else if(next == 1 && submit == 0 && addressFromPi == 19'h4afff) begin
        my_charDoneToPi = 0;
    end else if(next == 1 && submit == 0 && addressFromPi == 19'h7ffff) begin
        freezeCursor = 0;
        clearScreen = 1;
    end else if (next == 0 && submit == 1 && addressFromPi == 19'h7ffff) begin
        my_solveToPi = 1;
    end else if (next == 0 && submit == 0) begin
        my_solveToPi = 0;
    end
end
```

Implementing the FSM between Raspberry pi and the FGPA based on control signals.

```
VRAM myVRAM (
    .address_a(addrA), // addra - for reading and writing to the white board
    .address_b(addrB), // addrb - for Pi to read data
    .clock(clk),
    .data_a(setDataA),
    .data_b(0),
    .wren_a(writeEnA),
    .wren_b(0),
    .q_a(outputA),
    .q_b(outputB)
);
```

VRAM initialization on the OCM. This is an IP module. It is true dual port RAM with 512000 words and 1-bit as word size for each pixel. 0 if white to be printed, 1 for black.

```
always_comb begin
    if((mouseButton == 2 || clearScreen == 1) && freezeCursor == 0) begin
        writeEnA = 1;
        setDataA = 0;
    end else begin
        if(mouseButton == 1 && cursor_on == 1 && DrawX < 640 && DrawY < 480 && DrawX >= 0 && DrawY >= 0) begin
            writeEnA = 1;
            setDataA = 1;
        end else if(mouseButton == 4 && cursor_on == 1) begin
            writeEnA = 1;
            setDataA = 0;
        end else begin
            writeEnA = 0;
            setDataA = 0;
        end
    end
end
```

These are the conditions for when to update the VRAM.

```
always_ff @ (posedge pixel_clk) begin
    if(blank == 0) begin
        // BLANKING
        Red = 8'h00;
        Green = 8'h00;
        Blue = 8'h00;
    end else begin
        // NON BLANKING
        if (cursor_on == 1'b1) begin
            // drawing cursor
            Red = 8'hff;
            Green = 8'h00;
            Blue = 8'h00;
        end else if (cursor_on == 1'b0) begin
            //drawing background from memory
            if(outputA == 0) begin
                Red = 8'hff;
                Green = 8'hff;
                Blue = 8'hff;
            end else begin
                Red = 8'h00;
                Green = 8'h00;
                Blue = 8'h00;
            end
        end
    end
end
```

Logic to produce VGA signals based on cursor and VRAM.

**Description of Software on Raspberry Pi:**

```
'import libraries'
import numpy as np
import pandas as pd
from tensorflow import keras
from PIL import Image
from itertools import groupby
import RPi.GPIO as GPIO
import time
import subprocess
```

We first import all the relevant libraries that run the neural network in the raspberry pi.

---

```
'Pin Definitions'
outputPin1 = 2
outputPin2 = 3
outputPin3 = 4
outputPin4 = 17
outputPin5 = 27
outputPin6 = 22
outputPin7 = 10
outputPin8 = 9
outputPin9 = 11
outputPin10 = 5
outputPin11 = 6
outputPin12 = 13
outputPin13 = 19
outputPin14 = 26
outputPin15 = 18
outputPin16 = 23
outputPin17 = 24
outputPin18 = 25
outputPin19 = 8
samplingInputPin = 21
restartInputPin = 12
charDoneInputPin = 16
solveInputPin = 20
raspReadyOutputPin = 7
```

The raspberry pi has several GPIO (general purpose input/output) pins. In total the raspberry pi has 26 GPIO pins, out of which we have used 24 GPIO pins:

- 19 output wires – used to send a 19-bit address requesting pixel data from the FPGA.
- raspReadyOutputPin – used to tell FPGA that it is ready with the output.
- samplingInputPin – pixel data sent from the FPGA at a particular memory address.
- restartInputPin – restart signal from FPGA telling raspberry pi to reset data.
- charDoneInputPin – signal from FPGA telling the raspberry pi that the character is drawn and that it can now do its process to calculate the number.
- solveInputPin – signal from FPGA telling the raspberry pi that all the characters have been entered and that it should calculate the equation.

```
def record_data():
    length = 307200
    arr = [0] * length

    for i in range (length):
        output_to_fpga(i)
        arr[i] = GPIO.input(samplingInputPin)

    return arr
```

In this function we create a 1D array recording all the pixel data coming from the FPGA. There is a total of 640*480 = 307200 pixels on the canvas in which the person is drawing. Therefore, we first initialize an array of that size with all 0s. We then request a particular address from the FPGA using the output_to_fpga function and store the value into the array coming from the samplingInputPin.

---

```
def output_to_fpga(num):
    bin_string = bin(num)[2:].zfill(19)
    bin_string = bin_string[::-1]
    GPIO.output(outputPin19, int(bin_string[18]))
    GPIO.output(outputPin18, int(bin_string[17]))
    GPIO.output(outputPin17, int(bin_string[16]))
    GPIO.output(outputPin16, int(bin_string[15]))
    GPIO.output(outputPin15, int(bin_string[14]))
    GPIO.output(outputPin14, int(bin_string[13]))
    GPIO.output(outputPin13, int(bin_string[12]))
    GPIO.output(outputPin12, int(bin_string[11]))
    GPIO.output(outputPin11, int(bin_string[10]))
    GPIO.output(outputPin10, int(bin_string[9]))
    GPIO.output(outputPin9, int(bin_string[8]))
    GPIO.output(outputPin8, int(bin_string[7]))
    GPIO.output(outputPin7, int(bin_string[6]))
    GPIO.output(outputPin6, int(bin_string[5]))
    GPIO.output(outputPin5, int(bin_string[4]))
    GPIO.output(outputPin4, int(bin_string[3]))
    GPIO.output(outputPin3, int(bin_string[2]))
    GPIO.output(outputPin2, int(bin_string[1]))
    GPIO.output(outputPin1, int(bin_string[0]))
```

This function takes a number as an input, converts it into binary, reverses the order (because of how the wires are mapped between the FPGA and the raspberry pi), and finally sends each bit of the binary number to the FPGA through the wires.

---

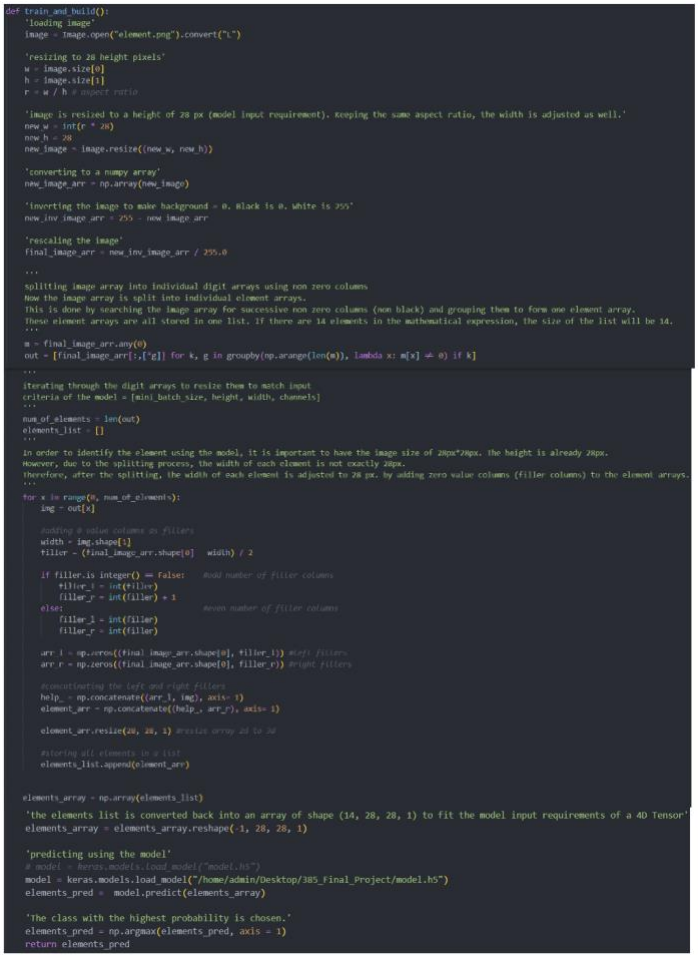
```python
def create_image(arr, w, h):
    arr = np.reshape(arr, (h, w))
    data = np.zeros((h, w, 3), dtype=np.uint8)

    for i in range (h):
        for j in range (w):
            if (arr[i][j] == 0):
                data[i][j] = [255, 255, 255]
            else:
                data[i][j] = [0, 0, 0]

    img = Image.fromarray(data, 'RGB')
    img.save('element.png')
```

This function takes the 1D array created by the record_data function and makes an image out of it. It does this by first making the 1D array into 3D. Where the first dimension is height, second dimension is the width, and the third dimension is the RGB values of that pixel. This 3D array is made into an image through python's PIL library.

---



```
def math_expression_generator(arr):
    op = {
                10,    # = "/"
                11,    # = "+"
                12,    # = "-"
                13     # = "*"
                  }

    'creating a list separating all elements'
    'm_exp would become [9, 8], 10, [7, 6], 13, [5, 4], 11, [3, 2], 12, [1, 0]] ⟶ 98 / 76 * 54 + 32 - 10 = 91.63'
    m_exp = []
    temp = []
    for item in arr:
        if item not in op:
            temp.append(item)
        else:
            m_exp.append(temp)
            m_exp.append(item)
            temp = []
    if temp:
        m_exp.append(temp)

    'converting the elements to numbers and operators'
    'so things like [9, 8] in m_exp are converted to the number 98'
    i = 0
    num = 0
   for item in m_exp:
        if type(item) == list:
            if not item:
                m_exp[i] = ""
                i = i + 1
            else:
                num_len = len(item)
                for digit in item:
                    num_len = num_len - 1
                    num = num + ((10 ** num_len) * digit)
                m_exp[i] = str(num)
                num = 0
                i = i + 1
        else:
            m_exp[i] = str(item)
            m_exp[i] = m_exp[i].replace("10","/")
            m_exp[i] = m_exp[i].replace("11","+")
            m_exp[i] = m_exp[i].replace("12","-")
            m_exp[i] = m_exp[i].replace("13","*")

            i = i + 1


    'joining the list of strings to create the mathematical expression'
    separator = ' '
    m_exp_str = separator.join(m_exp)

    return (m_exp_str)
```

This function is responsible for two main things. Firstly, it turns the 640*480 image into an array of 28*28 images each representing a character drawn on the canvas. It does this by resize the image, and then recognize where an element ends by the process of segmentation. Segmentation works by recognizing where all black pixels in a column change to white

pixels, and then chopping up the picture at the point where the character ends. Secondly, the function sends each 28*28 image in the array through a neural network that recognizes what element resides in each image. To see the entire process discussed in depth, please take a look at the comments in the image above.

---

This function is responsible for taking the array of predicted elements and turning it into a string equation. For instance, if the person wrote “1” and “2” on the canvas, then this function will recognize it as “12”. Furthermore, it turns all the operators which were numbers into symbols. Finally, it takes all of this information and then turns it into one comprehensive equation. This function is only called when the person is done drawing the full equation and wants to calculate the final answer. This means that theoretically, the person can draw as big of an equation as required.

---

```
m_exp_str = ""
GPIO.output(raspReadyOutputPin, 0)
output_to_fpga(0)

while True:
    try:
        if (GPIO.input(restartInputPin) == 1):
            m_exp_str = ""
            GPIO.output(raspReadyOutputPin, 0)
            output_to_fpga(0)
```

This is the main script that is run at the start of the while loop. Since there are multiple parts to this, the code is divided into several parts and explained. Firstly, to indicate to the FPGA, that the raspberry pi is booted and is ready to do its task, the FPGA sends 0 through the 19 wires to the FPGA and sets to output of the raspReadyOutputPin to 0. 0s through the 19 wires act work like a flag that tells the FPGA to unfreeze the cursor and allow the person to draw.

```
elif (GPIO.input(charDoneInputPin) == 1):
    output_to_fpga(0)
    GPIO.output(raspReadyOutputPin, 0)
    'recording data from fpga'
    data = record_data()

    'create image from data'
    create_image(data, 640, 480)

    'Building and Training CNN Model'
    predictions = train_and_build()

    'creating the mathematical expression'
    predicted = math_expression_generator(predictions).replace(" ", "")
    m_exp_str += predicted

    while(1):
        if (GPIO.input(charDoneInputPin) == 0): break

    print("Char Recognition Done")
    print(m_exp_str)

    is_int = False

    if (predicted[0] == '-'):
        is_int = predicted[1:].isdigit()
    else:
        is_int = predicted.isdigit()

    if(is_int == True):
        num = int(predicted)
        if(num < 0):
            num = -num
            output_to_fpga(num)
            GPIO.output(outputPin19, 1)
        else:
            output_to_fpga(num)
    else:
        output_to_fpga(0)

    GPIO.output(raspReadyOutputPin, 1)
    time.sleep(1)
    GPIO.output(raspReadyOutputPin, 0)
    output_to_fpga(0)
```

Once this is done, an infinite while loop is run, that constantly checks if the FPGA is trying to communicate with the raspberry pi. For instance, if the restart button has been clicked on the FPGA, then the raspberry pi also clears all of its content.

When the user is done with typing the character, then this condition is entered. This is where all the functions that take care of recording the data from the FPGA, creating its image, recognizing the elements in the image, and creating a mathematical equation in the form of a string is implemented. All of these functions have been discussed above. Once it has gotten the string, it decides whether the element was a positive number, negative number, or an operator. If it is an integer, then it turns it into a positive number and sets the $19^{th}$ bit of output to 1 implying that a negative number is being sent over. If it is positive, then it just

sends the output without setting the 19th bit to 1. Notice that the raspReadyOutputPin has been set high, meaning that the raspberry pi just sent the output to the FPGA for it to display. Once the answer has been sent, it sends all output pins to 0 again implying that the raspberry pi is done and is now waiting for the FPGA to do its work again. There is a 1 second delay in between to handle time synchronization issues that occurred due to the fact FPGA took a while to clear its screen and be ready again. By doing this we successfully implemented our custom protocol to communicate between the FPGA and the raspberry pi. It is also important to note that the string recognized here is added to the overall mathematical equation. Essentially, each time the person presses character done; the overall equation gets longer. Lastly notice that there is a while loop inside the if statement. This while loop enforces the program to pause until the turns button down asking the raspberry pi to do calculations.

```
    elif (GPIO.input(solveInputPin) == 1 and m_exp_str != ""):
        'calculating the mathematical expression using eval()'
        answer = eval(m_exp_str)    #evaluating the answer
        time.sleep(1)
        print(answer)
        answer = round(answer)

        m_exp_str = ""

        if(answer < 0):
            answer = -answer
            output_to_fpga(answer)
            GPIO.output(outputPin19, 1)
        else:
            output_to_fpga(answer)

        GPIO.output(raspReadyOutputPin, 1)
        time.sleep(0.1)

        while(1):
            if (GPIO.input(solveInputPin) == 0): break
```

When the person wants to solve the equation, it pushes the solve button which makes the raspberry pi program come to this if statement. Inside here, the final answer is calculated using the python's eval function (which considers the order of operations). As discussed above, negative answer is made positive and sent over the wires with the 19th bit set to 1.

```
    except Exception:
        m_exp_str = ""
        GPIO.output(raspReadyOutputPin, 1)
        output_to_fpga(int('7FFFF', 16))
        print("Expression cannot be predicted")
```

Also notice that error handling has been implemented, meaning that if there is any error at any point, then instead of killing the script, the user is notified that there is an error by sending ‘7FFF’ to the FPGA. If there is ever an error, the user can simply restart the system by pressing the restart button.

---

```
#Preparing the dataset
np.random.seed(2)

'load the dataset'
dataset = pd.read_csv("dataset.csv")

"creating label"
y = dataset["label"]

"dropping label"
X = dataset.drop(labels = ["label"], axis = 1)

"deleting dataset to reduce memory usage"
del dataset

'overview of dataset'
g = sns.countplot(y)
y.value_counts()

'Grayscale normalization to reduce the effect of illumination differences.'
X = X / 255.0

'''
reshaping the dataset to fit standard of a 4D tensor of shape [mini-batch size, height = 28px, width = 28px, channels = 1 due to grayscale].
-1 for reshape means unknown (python will figure out this dimension itself by looking at the number of elements left) Batch size in this case
is going to be the number of images. 28x28 is the x and y values of the image. We are essentially reshaping 1D array of pixel values ranging from
0 to 255 into 2D array of size 28x28'
'''
X = X.values.reshape(-1,28,28,1)

'categorical conversion of label. 10 numbers and 4 operations. Converts it to hot bit vectors'
y = to_categorical(y, num_classes = 14)

'90% Training and 10% Validation split'
random_seed = 2
X_train, X_test, y_train, y_tes = train_test_split(X, y, test_size = 0.1 , random_state = random_seed, stratify = y)
```

Now, I will discuss how the neural network is made. This function reads the dataset.csv file which contains the raw data of 500,000 images of characters and operators. It then standardizes the data and creates X and y arrays. X contains all the data of the images, and y contains all the labels of those images. Once this is done, data is split into 90%-10% testing-evaluation ratio. This means that the neural network is trained on the training set and tested on the evaluation set.

---


This function creates the model using the X and y training sets. To be specific a 12-layer convolutional neural network is created and trained using data. How good the model is judged based on how accurate the model is. Once the model is made, it is saved as model.h5 file which is later accessed by the train_and_build function as explained above. The specifics of what this code is doing can be understood by looking at the comments.

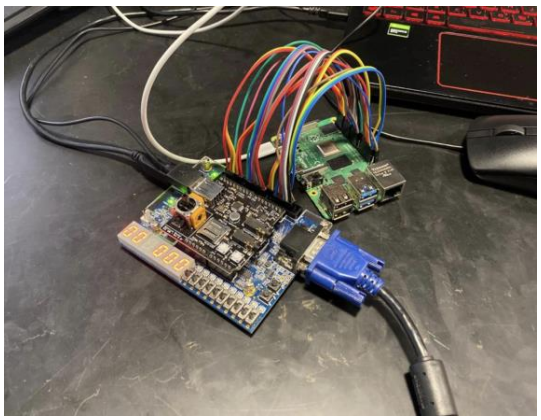
**ModelSim:**

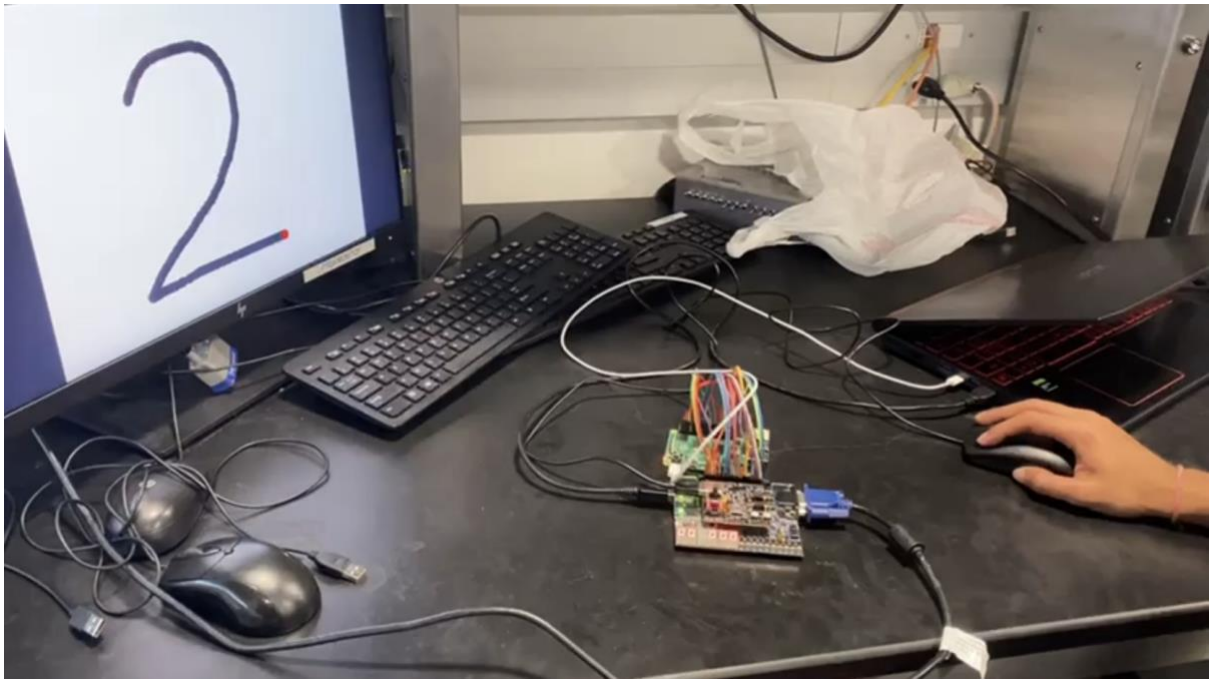
Due to the nature of our project, it was not feasible to run ModelSim to check all 480x640 bits of data. Checking 320,000 pixels was done by checksum function in lab 6.2. Anyways we were able to replicate our VRAM to raspberry pi and hence did not require modelsim.

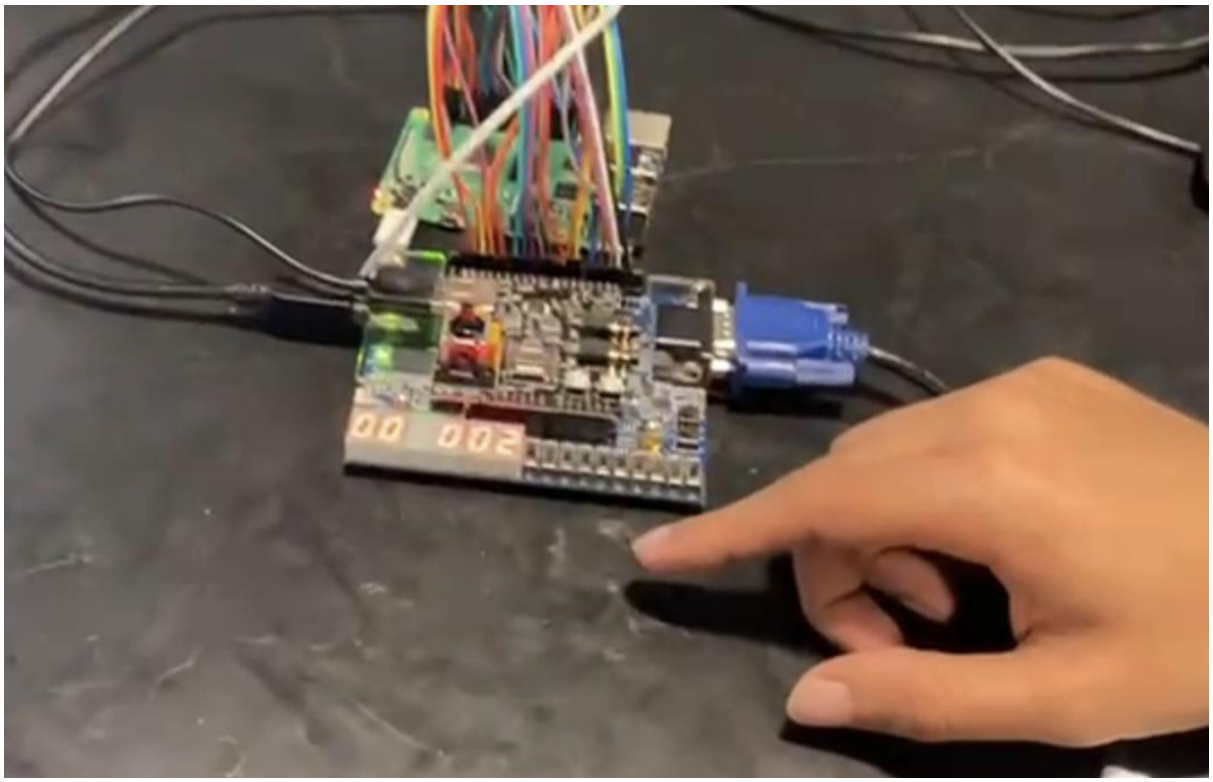
**Conclusion:**

We had lots of fun making the project. In this project we learnt about micro-controllers such as Raspberry Pi and how to use GPIO pins. Though we tried to implement SPI protocol, we were not very successful and instead implemented memory mapped I/O to transfer data between the FPGA and Raspberry Pi. The AI script was auto run-on boot. It would be appreciated if we could have more resources on SPI protocol.

**Some Images:**


2


00 002