

# Touchscreen Digit Recognition System

ECE 540 Final Project  
Team Presentation

# Project Introduction

- Goal: Create an alphanumeric character recognition system using Raspberry Pi5 + touch screen + FPGA.
- Touch drawn digit on capacitive touchscreen
- RPi5 captures and pre-processes the input
- FPGA receives image and runs a 2D convolutional neural network that is trained on the EMNIST dataset to identify alphanumeric characters
- Due to time limits, we did not complete the full pipeline but each stage has varying levels of success

# System Architecture Overview

- Touchscreen (FT6336U) connected to RPi5 via I<sup>2</sup>C
- RPi5 uses Python/C to capture touch screen data and convert it to a 28x28 pixel image
- Image sent to FPGA over custom GPIO protocol pixel by pixel
- FPGA receives and buffers the image, prints the results to the UART terminal (using `ee_printf`) from the ML model


# OpenCores I<sup>2</sup>C Master Integration (FPGA)

- Open-source i2c\_master\_top.v with Wishbone wrapper
- Successfully connected to FT6336U via SDA/SCL
- Observed 0x03 or 0x0C consistently (no real touch data)
- Tried pull-up resistors and manual reset – no change
- Likely issues: bus timing, FSM bugs, wrong command sequence

# RPi5 I<sup>2</sup>C Communication with FT6336U

- Used smbus2 in Python to access FT6336U over I<sup>2</sup>C
- Touch count and coordinates captured reliably
- PIL used to render touch image as grayscale
- Image resized to 28x28 pixels and prepared for transfer

## Raspberry Pi 5 → Touchscreen (FT6336U & ST7796S)

Touchscreen Signal	RPi GPIO	Protocol	Notes	
CTP_SCL (I <sup>2</sup> C clock)	GPIO3 (SCL1)	I <sup>2</sup> C		
CTP_SDA (I <sup>2</sup> C data)	GPIO2 (SDA1)	I <sup>2</sup> C		
CTP_RST	GPIO17	Output	Use to reset the touchscreen on startup	
LCD_SCLK (SPI)	GPIO11 (SPI0_SCLK)	SPI	ST7796S LCD driver	
LCD_MOSI	GPIO10 (SPI0_MOSI)	SPI		
LCD_CS	GPIO8 (SPI0_CE0_N)	SPI		
LCD_DC	GPIO25	Output	Data/Command select	
LCD_RST	GPIO27	Output	Reset pin for LCD panel	
LCD_BL	GPIO18 (PWM0)	PWM	Backlight brightness (optional)	

# Custom GPIO Protocol (RPi5 → FPGA)

- 8-bit parallel pixel data via GPIOs JB1 - JB4, JB7 - JB10
- JA1 = data\_valid
- JA2 = data\_ack (FPGA → RPi5)
- JA3 = end\_of\_image
- Pi sends pixel + control flags, FPGA latches data
- JA2 toggle confirms FPGA handshake
- Pixel buffer stored in gpio0 peripheral (0x80001400)
- We saw communication between FPGA, RPi5 and Touchscreen, but we cannot access the transferred information

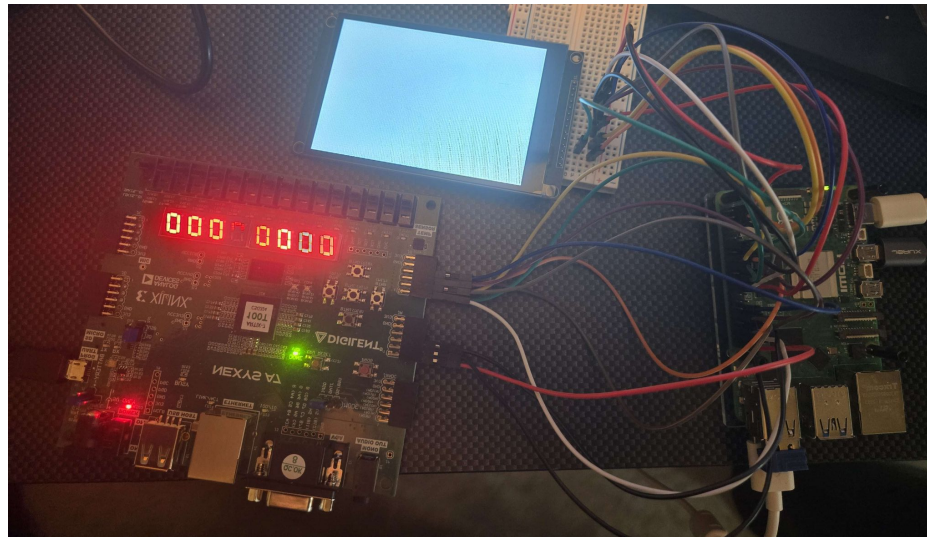
Raspberry Pi 5 → FPGA (Nexys A7 PMOD GPIO Protocol)				
RPi5 GPIO	Protocol Bit	Wire Color	FPGA PMOD Pin	PMOD Header
GPIO26	Bit 0	White	JB1	PMOD JB
GPIO16	Bit 1	Grey	JB2	PMOD JB
GPIO6	Bit 2	Purple	JB3	PMOD JB
GPIO5	Bit 3	Blue	JB4	PMOD JB
GPIO24	Bit 4	Green	JB7	PMOD JB
GPIO23	Bit 5	Yellow	JB8	PMOD JB
GPIO22	Bit 6	Orange	JB9	PMOD JB
GPIO12	Bit 7	Brown	JB10	PMOD JB (PWM, set correctly in software)
GPIO21	Bit 8 (data_valid)	Red	JA1	PMOD JA
GPIO20	Bit 9 (data_ack)	Black	JA2	PMOD JA
GPIO13	Bit 10 (end_of_image)	Black	JA3	PMOD JA

# FSM + Pixel Buffer HDL

- Inside the FPGA, we implemented an FSM with three main states: IDLE, RECEIVE, and WAIT\_ACK.
  - IDLE: Waits for the first pixel.
  - RECEIVE: Latches the pixel data into a local 784-byte register array (pixel\_buffer[0:783]).
  - WAIT\_ACK: Waits for the Pi to acknowledge before returning to RECEIVE.
- Each pixel is written sequentially to a memory-mapped region inside the gpio0 peripheral.
- Once all 784 bytes have been sent from the RPi5, the end\_of\_image flag is set to tell it when to stop

# Hardware Setup + Results

- We learned a lot about the difficulties of I2C, even when using verified cores for the specific bus we are using
- We learned how to build interfaces between an FPGA, RPi5 and peripherals.
- We achieved communication between the touchscreen, RPi5 and FPGA, but it was garbled data when it got to the FPGA





# Alternative Hardware Setup

FPGA PMOD PINS -> TOUCHSCREEN PINS

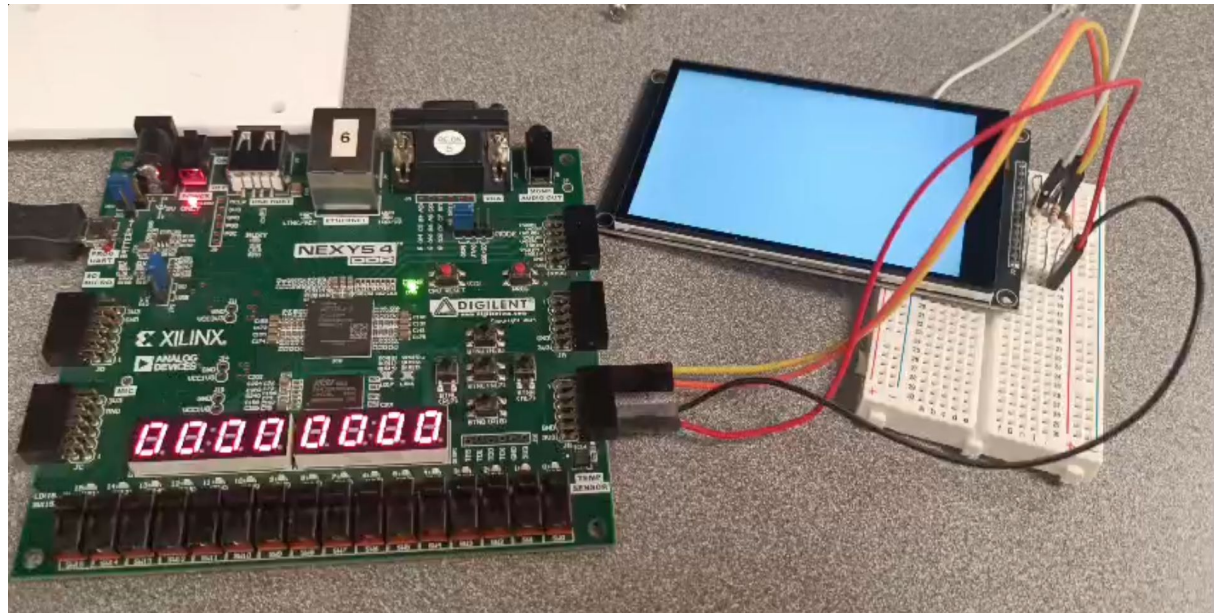
- JB[6]- VCC
- JB[5] - GND

I2C

- JB[1] - CTP\_SCL
- JB[2] - CTP\_SDA
- JB[10] - CTP\_RST

SPI

- JB[3] - LCD\_SCK
- JB[4] - LCD\_MOSI
- JB[7] - LCD\_CS
- JB[8] - LCD\_DC
- JB[9] - LCD\_RST



# SPI LCD Display Integration

- Modified SPI core for LCD timing requirements
- Added delay states for LCD initialization
- Connected to LCD display for command/data transmission.
- Ensured proper WISHBONE bus interfacing.
- Verified data flow via FIFO buffers

Why It Might Have Failed:

- FIFO or bus handshaking issues in the SPI-WISHBONE integration.
- Clocking mismatch with the touch screen's SPI settings.
- Timing delays not properly calibrated for LCD command readiness—delay may be too short/long or misaligned.

# Software Setup

- Raspberry Pi reads touch data over I<sup>2</sup>C and draws the pixels onto the screen using SPI
- Touch screen data formatted into 28x28 image with anti-aliasing
- Image data is sent to the FPGA via a custom GPIO protocol
- Image array fed to 2D convolutional neural network trained on the EMNIST dataset to identify alphanumeric characters.
- Model returns the determined character and confidence
- Model results are printed to the console.

# General CNN Structure

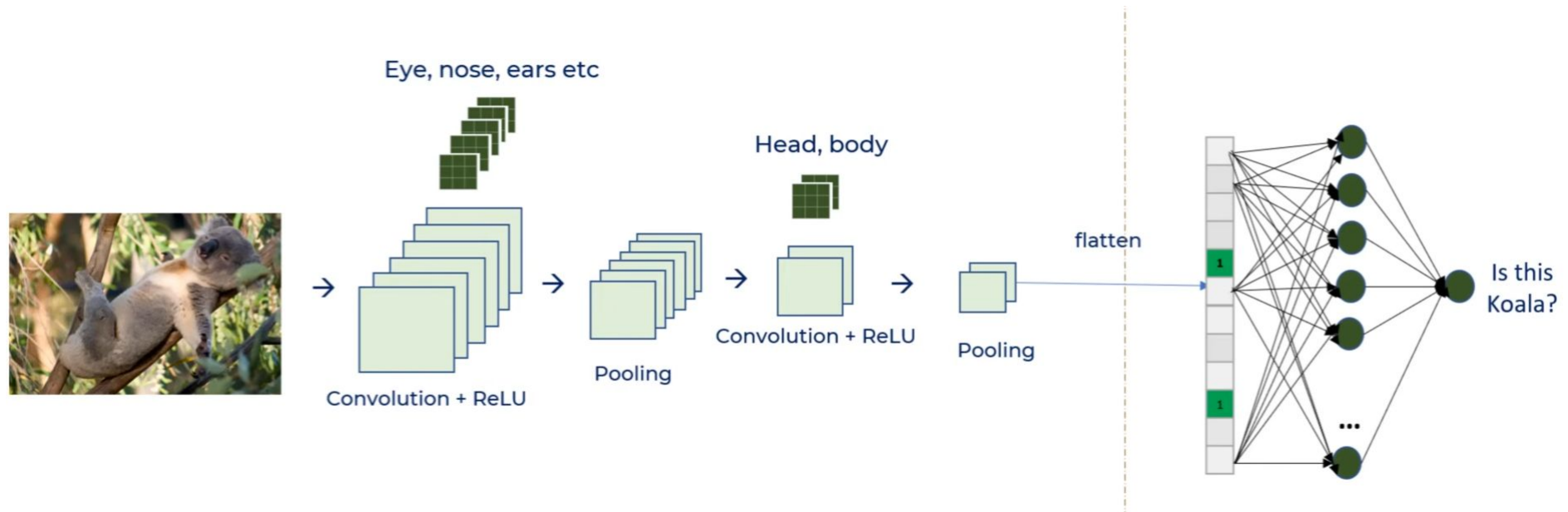


Figure from codebasics, "Simple explanation of convolutional neural network...", YouTube (2020)

# Model Architecture

## First and Second Section

- Conv2D: 32 Filters, Kernel size of 3, L2 Regularization
- Relu Activation
- Batch Normalization
- 25% dropout on second section

## Third and Fourth Section

- Conv2D: 64 Filters, Kernel size of 3, L2 Regularization
- Relu Activation
- Batch Normalization
- 25% dropout

## Fifth Section

- Flattening
- Dense Layer 256 outputs
- Relu Activation
- Batch Normalization
- 25% dropout

## Fifth Section

- Dense Layer
- Softmax Activation

# Model Optimization

Optimizer: Stochastic gradient descent (SGD)

- Learning Rate: 0.01
- Momentum: 0.9

Loss Function: Cross Entropy

# Model Results

- The prototype model made with python worked well with good accuracy considering the data set. An example training run can be seen below
- The model was able to be reconstructed in C without the aid of any libraries, however, due to differences in preparation of data batch normalization currently zeros out the weights. This was quite problematic as the weights of the C model were imported from the training done by the python model

```
882/882 ————— 0s 43ms/step - accuracy: 0.9126 - loss: 0.2802
Epoch 20: val_accuracy did not improve from 0.90032
882/882 ————— 40s 45ms/step - accuracy: 0.9126 - loss: 0.2802 - val_accuracy: 0.8981 - val_loss: 0.3452 - learning_rate: 0.0025
Restoring model weights from the end of the best epoch: 19.
147/147 - 2s - 10ms/step - accuracy: 0.9003 - loss: 0.3420
Final Test:
    Accuracy: 90.03%
    Loss: 0.3420
Saving model info
Saved: emnist_model.json, emnist_weights.npz, label_names.json
```

# Conclusion & Future Work

- Touchscreen and RPi5 I<sup>2</sup>C code functional
- GPIO protocol between RPi5 and FPGA verified but not finished
- I<sup>2</sup>C OpenCores RTL connected but failed to provide valid data
- Prototype CNN in python works but data isn't quite prepared correctly for the final model made using C
- Future work:
  - Retry I<sup>2</sup>C with clean state - Using OpenCore I2C or LiteX
  - Retry SPI with clean state - Using OpenCore SPI
  - Add VGA connector to print the information on a monitor
  - Integrate digit recognition on RISC-V core, in hardware
  - Complete port of all Python models to C/C++



# Team Member Contributions

- **Phil Nevins** - FPGA HDL, RPi5 Python/C, FPGA->RPi5 Communication Protocol HDL, Debugging
- **Jeremiah Vandagriff** - Machine Learning Models and FPGA->Pi Communication Protocol Design
- **Aaditya Shah** - FPGA Hardware/HDL -> I2C Protocol Debugging, SPI
- **Sai Charan Reddy Balpunuri** - FPGA Hardware/HDL -> I2C Protocol debugging, SPI