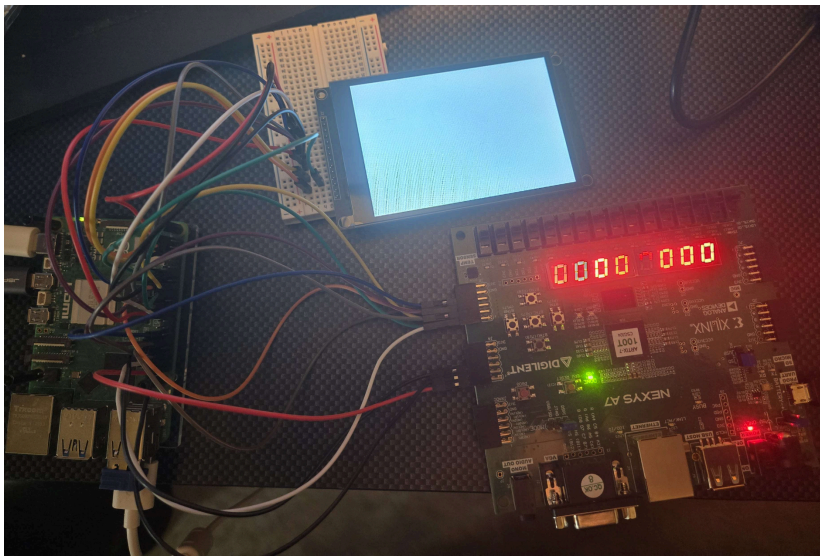
Touchscreen Digit Recognition System



ECE 540 SOC Design Final Project
https://github.com/Dawgburt/ECE540-FinalProj

Phil Nevins
p.nevins971@gmail.com

Jeremiah Vandagrift
jcv3@pdx.edu

Aaditya Shah
aadityas@pdx.edu

Sai Charan Reddy Balpunuri
saicha@pdx.edu

Project Overview

This project aimed to build a complete embedded system that enables a user to draw an alphanumeric character on a capacitive touchscreen, capture that image using a Raspberry Pi 5 (RPi5), convert the drawing to a 28x28 pixel grayscale frame, and transmit it to an FPGA for further processing. The final goal was to create a complete digit recognition pipeline on hardware, but due to time constraints, we were unable to complete the full system integration. However, significant progress was made in both hardware and software development.

System Architecture

Touchscreen to RPi5 Communication

The touchscreen used is a 4.0-inch capacitive touchscreen with the FT6336U touch controller, connected to the RPi5 using I²C (SDA/SCL). The display component is controlled over SPI, although display output was not used in this phase of the project.

I²C Pins:
- SDA: GPIO2 (Pin 3)
- SCL: GPIO3 (Pin 5)

SPI Pins (Display - not used - stretch goal):
- MOSI: GPIO10
- SCLK: GPIO11
- CS: GPIO8
- DC: GPIO25

Python Script on RPi5

A custom Python script using smbus2, PIL, and libgpiod handles the following tasks:
1. Read touch data from FT6336U via I²C.
2. Tracks touch gestures and captures the drawn image.
3. Converts the captured gesture into a 28x28 grayscale image using PIL.
4. Transmits the image pixel-by-pixel over GPIO using a custom parallel protocol.

GPIO Transmission Protocol

We designed a custom 8-bit parallel GPIO protocol where each GPIO line corresponds to a bit of the grayscale pixel. A data_ack line from the FPGA (JA[2]) provides handshaking.

Signal Assignments:
- Data Lines (connected from Pi to JB1–JB10 on the FPGA)
- Control Lines:
  - data_valid (output from Pi to FPGA): marks valid pixel
  - end_of_image (output from Pi to FPGA): marks last pixel

- data_ack (input from FPGA): used to signal ready-for-next

Observed Behavior:
- JA[2] (data_ack) toggled in response to data_valid from the Pi, confirming handshaking worked.
- All 784 pixels (28x28) are being sent, but FPGA output does not show correct behavior—possibly due to timing, FSM bugs, or incorrect Wishbone register mapping.

FPGA Hardware Design

I²C Core Attempts

- Tried integrating the **OpenCores I²C master** (i2c_master_top.v) with a custom wrapper.
- Verified the bus transactions through UART and internal signal debugging.
- **Partial Success**: The core returned fixed values (0x03 or 0x0C), regardless of touch activity.
- GPIO INT pin was pulled up with a 10k resistor, reset line manually toggled.

LiteX I²C Core Attempt

- Attempted integrating the **LiteX I²C core** for Wishbone.
- Integration was not completed due to limited documentation and build complexities.
- Future work may include a standalone I²C-to-Wishbone test bench to validate functionality.

Challenges Encountered

- The **FPGA FSM** correctly handshakes with the RPi5, but output logic or framebuffer reading is not functional.
- The **OpenCores I²C core** produced consistent but incorrect values—possibly due to bus timing issues or invalid FT6336U command sequences.

GPIO Transmission Protocol

To facilitate high-speed communication between the Raspberry Pi 5 and the FPGA, we designed a custom 8-bit parallel GPIO protocol with handshaking. This protocol enables the Pi to stream a full 28x28 grayscale image (784 bytes) to the FPGA one pixel at a time.

FPGA Pin	RPi GPIO	Function	Bit
JB1	GPIO26	Pixel Bit 0 (White)	0
JB2	GPIO16	Pixel Bit 1 (Gray)	1
JB3	GPIO6	Pixel Bit 2 (Purple)	2
JB4	GPIO5	Pixel Bit 3 (Blue)	3
JB7	GPIO24	Pixel Bit 4 (Green)	4
JB8	GPIO23	Pixel Bit 5 (Yellow)	5
JB9	GPIO22	Pixel Bit 6 (Orange)	6
JB10	GPIO12	Pixel Bit 7 (Brown)	7
JA1	GPIO21	`data_valid`	
JA2	GPIO20	`data_ack` (FPGA → Pi)	
JA3	GPIO13	`end_of_image`	

Transmission Protocol:

1. RPi5 sets pixel bits (JB1–JB10) with an 8-bit grayscale value.
2. It sets the end_of_image flag (JA3) high only on the last pixel.
3. The FPGA monitors these signals and enters a RECEIVE state when data is valid.
4. FPGA latches the pixel byte into an internal buffer indexed by a 10-bit counter.
5. Once latched, the FPGA raises data_ack (JA2) to inform the Pi that the pixel was received.
6. The Pi waits for data_ack, then proceeds to the next pixel.

FSM and Pixel Buffer Implementation

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.

GPIO-0 Peripheral Integration

We reuse the existing gpio0 peripheral (originally used for switch-to-LED mapping) and modified it to expose 784 bytes of internal pixel buffer memory to the SoC address space:

- Base Address: 0x80001400
- Access Width: 32-bit, but internally adapted to 8-bit per pixel for framebuffer.
- Use Case: Software can poll this region to verify incoming data, or eventually pass it to an ML model.

The pixel buffer is memory-mapped, so the CPU core (Veerwolf/RISC-V) can read the pixel array for future processing or debugging.

SPI Controller Modification and LCD Integration

Overview

To interface a touchscreen display over SPI with our RISC-V SoC on the Nexys A7 FPGA, we adopted and extended the OpenCores simple_spi.v controller. Our modifications aimed to accommodate timing-sensitive peripherals such as an LCD display requiring command/data sequencing and initialization delays.

Tasks Performed

1. ***Modified SPI Core***
    - Used an open-source SPI controller and adapted it for our system.
    - Implemented parameterizable slave-select width and added LCD-specific control parameters like LCD_COMMAND_DELAY.
2. ***Integrated SPI Core with WISHBONE Bus***
    - Connected the SPI controller to the system's WISHBONE bus interface.
    - Ensured correct decoding of address, data, and control signals for write/read operations.
3. ***Inserted Custom Timing Control***
    - Introduced a configurable delay counter inside the SPI state machine.
    - Ensured delay cycles between command and data phases as required by the LCD.
    - Held CS active and paused SPI transmission for the LCD initialization sequence using a FSM wait state driven by the lcd_delay_active signal.
4. ***WISHBONE Bus Integration***
    - Interfaced SPI with the WISHBONE bus for system-wide communication.
    - Verified proper bus handshaking using cyc_i, stb_i, ack_o, and address decoding.
5. ***FIFO-Based Data Buffering***
    - Verified bidirectional data flow using write (wfifo) and read (rfifo) FIFOs.

- FIFO read/write signals (wfre, rfwe) were tested with known patterns.
- Ensured FIFO flags (full, empty) were correctly read via the SPSR status register.

6. ***Connected LCD Display***
   - Transmitted commands and data over SPI to the LCD module.
   - Used logic analyzers/simulation to confirm proper SPI waveform generation (MOSI, SCK, SS) matched the LCD's protocol requirements.

Software Design

FPGA Software

The software running on the FPGA was originally planned to directly interact with the touch screen but we pivoted the design so that it would receive the screen data through a Raspberry Pi 5 as an intermediary. The FPGA program takes that image data and feeds it to a convolutional neural network that identifies the character drawn and returns it to the main portion of the program. The program would then print the results out to the console. A python version of the convolutional neural network was originally developed to test and train the architecture and the weights of the trained network were exported for the C program. This removed the need to implement training in the C program that was running on the FPGA. The particular architecture implemented for the machine learning model was as follows:

First and Section 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 on second section

Firth Section

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

Sixth Section

- Dense Layer 47 outputs
- Softmax Activation

The model was optimized using stochastic gradient descent (SGD) with a learning rate of 0.01 and momentum of 0.9

RPi5 Touch Screen Data Acquisition and Transmission

To interface with the capacitive touchscreen (FT6336U controller), we developed both **Python** and **C** programs on the Raspberry Pi 5. These scripts handled real-time touch detection via I²C, rendered the gesture into a grayscale image, and transmitted the data to the FPGA using a custom GPIO protocol.

I²C Communication (Python)

- The FT6336U touchscreen was accessed using the **smbus2** library over I²C.
- We implemented functions to:
    - Read the **touch count** register.
    - Extract **X and Y coordinates** of up to 2 touch points.
    - Track continuous movement to construct a gesture path.
- A reset sequence (manual or GPIO-based) and a 10kΩ pull-up on the **INT** line were added for stable communication.
- The gesture was drawn on a **Pillow (PIL) Image canvas**, updated in real time.

Image Formatting

- Once the user finished drawing (determined by no-touch timeout), the gesture image was:
    - Converted to grayscale ('L' mode).
    - **Resized to 28×28 pixels** using PIL.Image.resize().
    - Thresholded and normalized for transmission.

This format matches the input shape expected by MNIST-based digit recognition systems, allowing compatibility with future ML stages.

GPIO Data Transmission Protocol

We implemented a **custom 8-bit parallel GPIO protocol** to transmit the 784-byte (28×28) image from the Pi to the FPGA. This was done using:

- The **python-libgpiod** library in Python.
- A direct GPIO register access method in C (planned but not fully deployed).

Transmission Workflow

- Each grayscale pixel (0–255) was encoded on 8 GPIO lines connected to the FPGA (JB1–JB10).

- A data_valid flag (GPIO21 → JA1) was asserted to indicate valid data.

- On the final pixel, end_of_image (GPIO13 → JA3) was also asserted.

- The Pi waited for the data_ack response (GPIO20 ← JA2) from the FPGA before sending the next pixel.

This continued until all 784 pixels were sent.

The protocol ensured reliable, synchronized transmission of image data without requiring a clock or serial encoding scheme.

Outcome

- The touchscreen-to-Pi I²C communication was **fully functional**.
- The full 28x28 pixel image was **successfully generated** on the Pi.
- Handshaking over GPIO was verified, as the Pi detected FPGA data_ack in real time.
- Image reception by the FPGA was initiated but not fully debugged for VGA display.

Future Work

Restart I²C Integration

- Rebuild the OpenCores I²C master from scratch.
- Re-attempt LiteX I²C integration.
- Explore other verified open-source I²C Wishbone peripherals.

VGA Output Testing

- Verify framebuffer writes from custom GPIO protocol.
- Map 28x28 grayscale frame to visual VGA output for debug.

Add Image Processing

- Design PL layer to process the 28x28 image for digit recognition.

System Timing Analysis

- Ensure FSM, memory buffering, and VGA access are properly synchronized.

Conclusion

While the full system was not completed, major subsystems—namely touchscreen I²C communication, image rendering, and GPIO transmission—are functional. The FPGA design is largely complete, with working FSM and protocol response, but touch screen output and internal frame buffer visualization remain to be debugged.

This project forms a solid foundation for a hardware-accelerated digit recognition system and offers many future avenues for development and refinement.

Appendix

HW Pinout

RPI5 -> Touch Screen

🖱️ Raspberry Pi 5 → Touchscreen (FT6336U & ST7796S)

Touchscreen Signal	RPi GPIO	Protocol	Notes
CTP_SCL (I²C clock)	GPIO3 (SCL1)	I²C	
CTP_SDA (I²C data)	GPIO2 (SDA1)	I²C	
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)

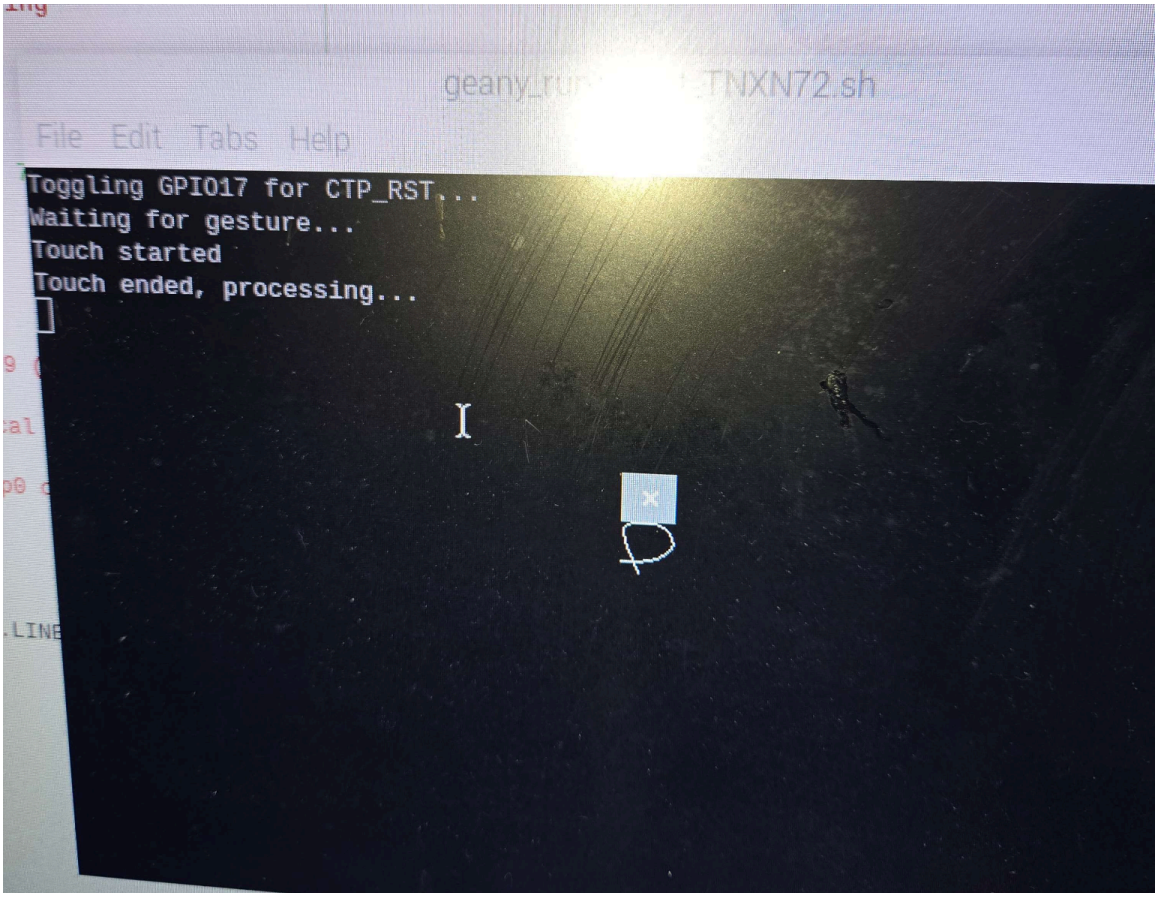
RPI5 -> FPGA

🔌 Raspberry Pi 5 → FPGA (Nexys A7 PMOD GPIO Protocol)

RPi5 GPIO	Protocol Bit	Wire Color	FPGA PMOD Pin	PMOD Hea
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

Screenshots

RPI5 Image Capture



Contributions

Phil Nevins - I worked on the system architecture for the FPGA. I tried to integrate the Open Core I2C, which I got to print a value when reading but it was garbage - it would be NOTHING if it wasn't connected in some way, so that is a good sign but nothing further came of this.. I tried LiteX i2c core, which did not work. I tried writing my own I2C core, which did not work at all. I worked on the GPIO handshaking protocol with Jeremiah, and wrote the HDL to support it. I wrote the RPI5 C / Python code (python was easier) to allow touchscreen data collection.

Jeremiah Vandagrift - Using python, I programmed an initial 2D convolutional network trained to recognize the handwritten alphanumeric characters using the EMNIST data set. I wrote several helper scripts for testing and exporting the trained weights from the python model to a C header file for use in a C version of the model. Reconstructed the prototype model in C using various approaches. Ultimately, I was not able to get the C model fully functional. I believe the architecture was correct and that the issues mostly came from a difference in the way the image data was being prepared or improper importing of weights. After the first batch normalization all of the weights would become zero as they were negative before normalization. In addition to writing the machine learning model, Phil and I worked together to create a communication protocol to send image data from the Raspberry Pi to the FPGA using the GPIO pins.

Aaditya Shah - In this project, I worked on implementing helper functions to enable I^2C communication between the FPGA and the FT6336U touch controller, along with SPI communication for rendering display output. Worked on the OpenCore I2C and SPI. Despite setting up the hardware interface, the system didn’t function as intended. I conducted thorough debugging of the I^2C protocol, checked signal connections, and tested multiple register configurations. Even after extensive testing, the hardware failed to respond correctly, indicating deeper integration challenges. This process significantly enhanced my understanding of hardware-software interaction and peripheral-level debugging.

Sai Charan Reddy Balpunuri - I focused on integrating the SPI controller to interface with a capacitive touchscreen LCD. I modified the SPI core to add custom timing delays required for LCD command transmission and ensured correct WISHBONE bus handshaking. I verified the internal data flow through FIFO buffers and monitored key SPI signals like MOSI, SCK, and SS. Although the display did not respond as expected, the groundwork for SPI-based communication and LCD timing control was successfully implemented and debugged.