Hardware Design Final Project

Application of Handwritten Text Recognition – The Best IDE

Team 13

111062109 Ting-Yao Huang, 111062209 Bo-Yi Mao

Table of Contents

§1 Introduction 1
- §1.1 Motivation 1
- §1.2 Project Overview 1

§2 Graphics Rendering 3
- §2.1 User Inputs 3
- §2.2 Utilization of Distributed Memory 3
- §2.3 Canvas Input 4
- §2.4 Pixel Generator 6

§3 Handwriting Recognizer 8
- §3.1 Train Data 8
- §3.2 CNN Model Structure 9
- §3.3 Parameter Format 10
- §3.4 Parameter Storage 11
- §3.5 Feature Map Buffer 12
- §3.6 CNN Implementation 15
- §3.7 Testing Model Correctness 21

§4 Text Data Communication 22
- §4.1 Text Editor 22
- §4.2 USB-UART Communication 23
- §4.3 Connect to Python Terminal 26

§5 Conclusion 27
- §5.1 Project Demonstration 27
- §5.2 Resource Utilization 28
- §5.3 Takeaways 29
- §5.4 Possible Improvements 30

§A Appendix 31
- §A.1 Contribution 31
- §A.2 The Bresenham's Line Algorithm 31
- §A.3 References 32

§1 Introduction

§1.1 Motivation

We stumbled upon a video[1] on YouTube about making Microsoft Paint a functional "IDE (integrated development environment)" with a neural network–based handwriting recognizer. However, its usage still needs to be improved, as we must save the paint file before running the code.

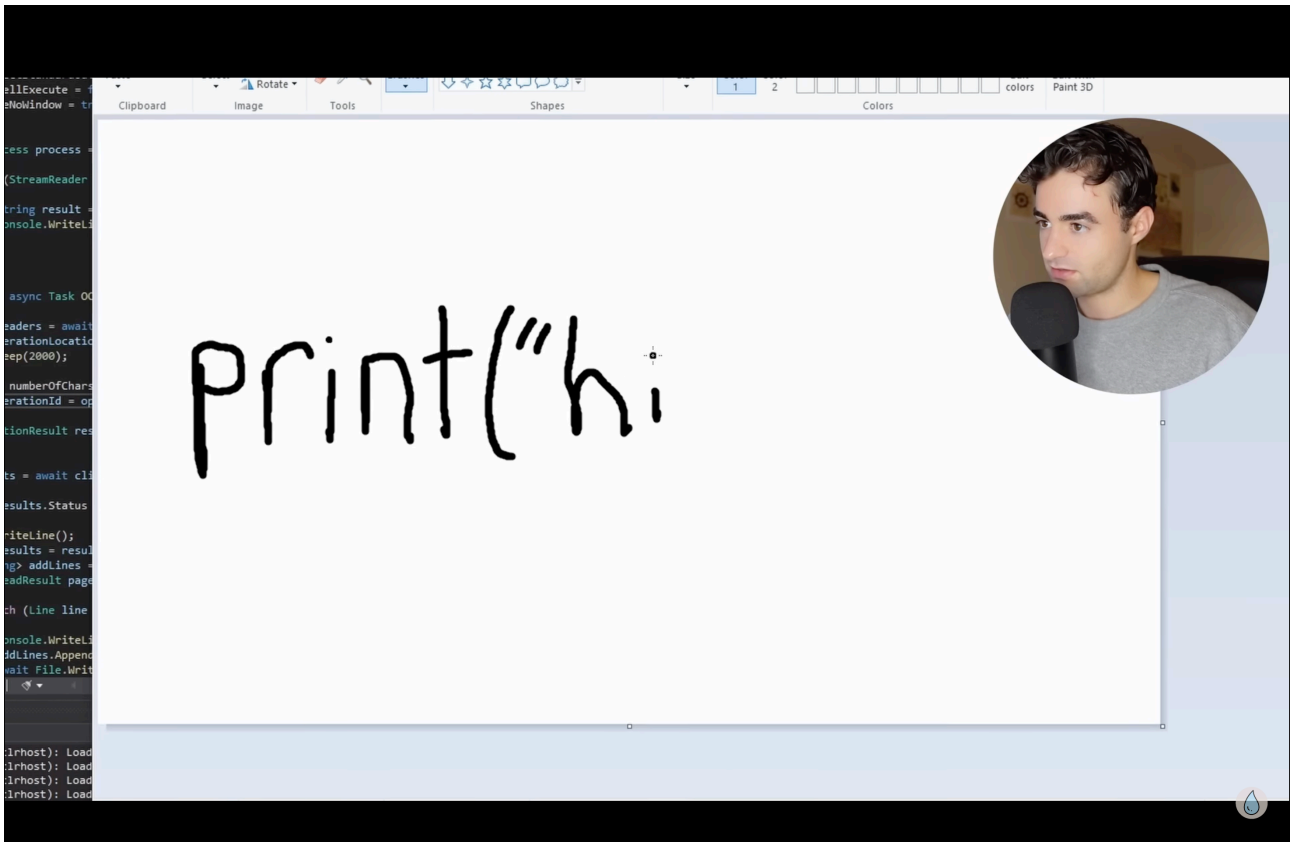

Fig 1.1 – Screenshot from "Why Microsoft Paint is the best IDE for programming"

We aim to implement a deep learning model in our final project while making the whole thing enjoyable. In addition, we want to improve the entire user experience better than the IDE from the video by actually "integrating" various useful functionalities into our version of The Best IDE.

§1.2 Project Overview

The project mainly contains three parts: canvas with mouse input and proper display, convolutional neural network (CNN) for handwriting recognizer, and text data communication with a Python server.

---

[1] Byte of Michael. (2021, March 30). Why Microsoft Paint is the best IDE for programming [Video]. YouTube. https://www.youtube.com/watch?v=JKxVEuy2d6k

Fig 1.2 demonstrates the overall structure of the project, as well as the data flow between the modules. External I/Os are represented as parallelograms, whereas the storage modules are drawn with an additional cross-line. All the modules are labeled with their actual functionality at the upper half and their module name in the Verilog code at the bottom half.

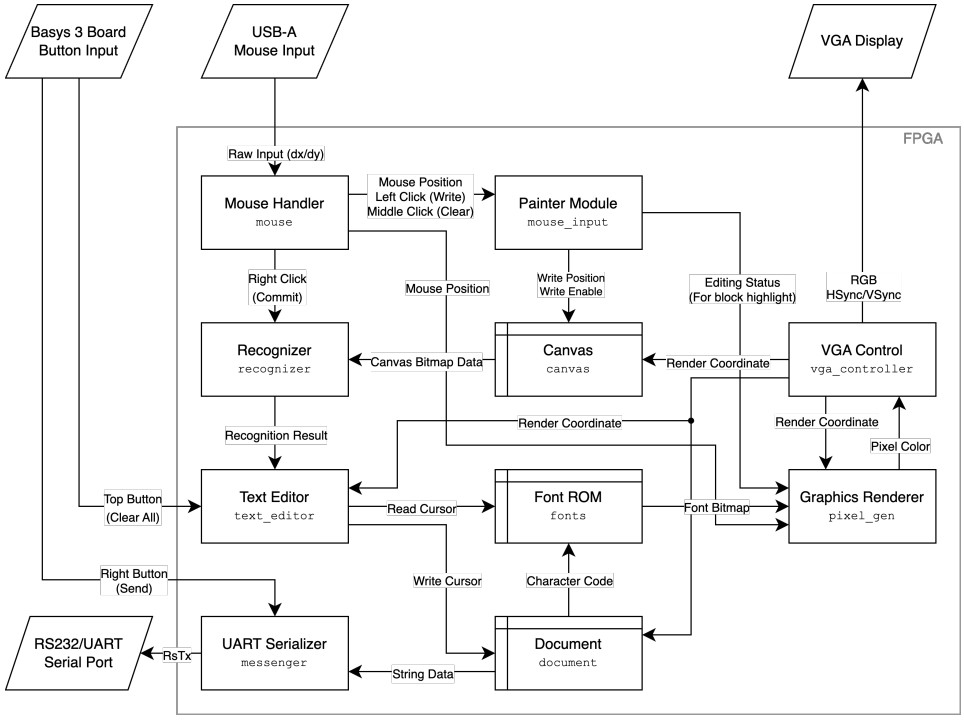

Fig 1.2 – Abstract module structure of the whole project

The detailed block diagram for the hardware implementation is shown in **Fig 1.3**, and each of these modules will be explained in the following sections. For each module, the green signals represent input signals, while the blue signals represent output signals. Also, the two-way port for the PS/2 interface is marked red.

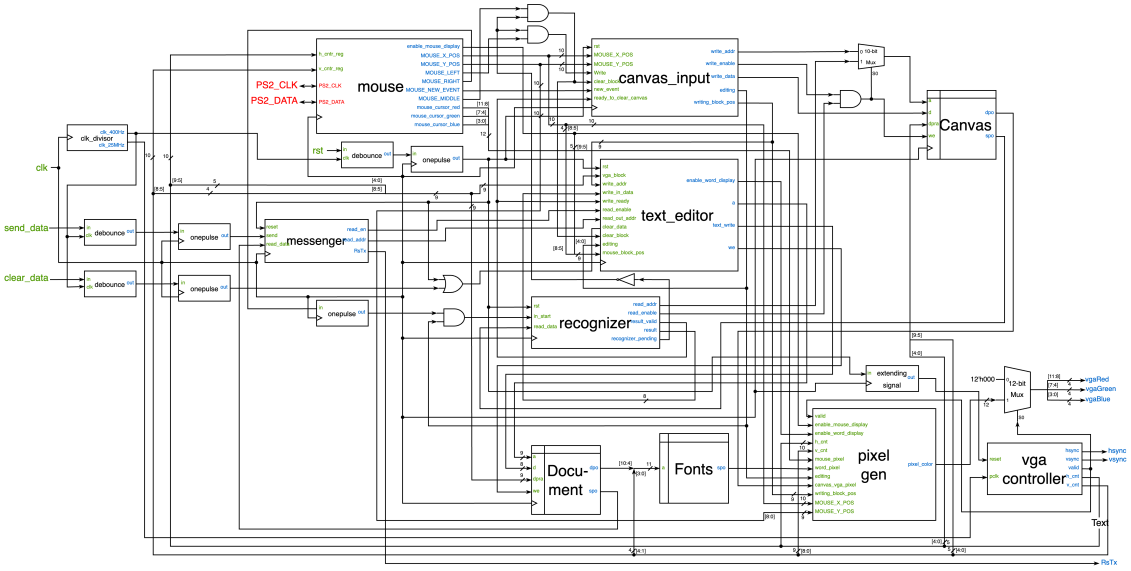

Fig 1.3 – The top-level block diagram of the whole project

§2 Graphics Rendering

§2.1 User Inputs

Fig 2.1 shows the primary input for user control. Users can write on any block they want to start the writing process. Once the block enters the editing state, the user must complete the block drawing and send it to the recognizer through the commit button or clear the so-far written progress to switch to other blocks. After completing the "coding," the user should press the send button to send the document data to the Python server. The user can clear the document via the top button on the FPGA board anytime.

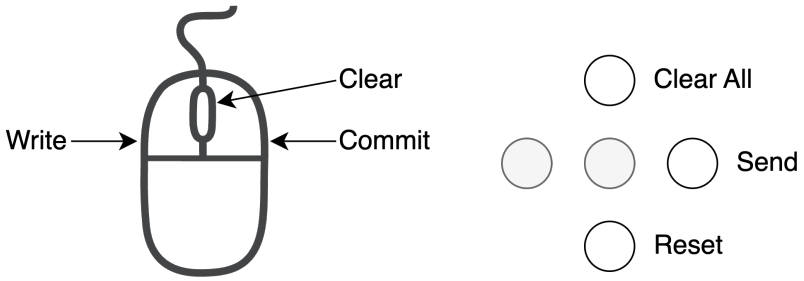

Fig 2.1 – Overall input methods

§2.2 Utilization of Distributed Memory

There are three distributed memory blocks used for the graphics rendering, shown in **Table 2.1**.

	Canvas	Fonts	Document
Type	Dual Port RAM	Single Port ROM	Dual Port RAM
Width	1	16	8
Depth	1024 (32×32)	1536 (16×96)	512 (16×32)
Stored Content	Handwriting drawn by user	Bitmap information of text display	So-far written & committed characters

Table 2.1 – Usage of distributed memory blocks

"Canvas" is designed as a 1×1024 block. With this design, we can **easily address the data with the vertical and horizontal coordinates** of the mouse position and write the pixels one bit at a time. We use the `canvas_input` module to control one RAM port, while the second port is reserved for the `recognizer` module.

"Fonts" is a 16×(16x96) block, storing the font bitmap for all 96 (including space and character 127) characters. Instead of using a 32×32 font, our implementation could reduce the memory usage to one-fourth of the original. Also, even with the 16x16 font, it is enough to show the character clearly to the user.

We can address the correct font pixel by obtaining **the character code and the block coordinate.** The block coordinate could be easily calculated by right-shifting the VGA's vertical and horizontal coordinates. As for the character code, we've encoded it as the **ASCII code with bias 32** since the first 32 ASCII characters are all control characters, which we're not considering.

Fig 2.2 – All printable ASCII characters displayed on VGA screen

"Document" is designed as an 8×512 block. For each memory entry, it stores the biased 8-bit ASCII code, and the display coordinate addresses each block. In our screen, there are actually only 20×15 blocks. However, the depth is padded to 32x16 since we'd like to address the block only by concatenating the vertical and horizontal block positions instead of multiplying a constant, which will consume more logic resources. **The `text_editor` module controls one of the two ports of the RAM, while the other port is reserved for `display`.**

There are several advantages to using the DRAM (Distributed RAM) instead of the BRAM (Block RAM). We mainly use DRAM here because it could implement **asynchronous read and write**, which is more convenient for display since we do not allow time cycle delays to get the correct pixel color. The critical path must go through"Document" and "Fonts" to get the right pixel color. If we implement these two blocks as BRAM blocks instead, we would get a two-cycle delay. Also, since the data size is not too large for a DRAM to handle, it would be great to save these resources for the parameter storage of the recognizer.

§2.3 Canvas Input

The primary input method for "Canvas" is drawing through the mouse. We used the mouse code provided in the previous labs to implement the mouse input. After some observation, we discovered that the mouse position outputted by the module would **be refreshed in a certain period**, approximately 8 to 16 cycles. Also, each time the mouse data is refreshed, the `MOUSE_NEW_EVENT` signal will be raised for one cycle. However, our main problem is that the position provided is **discrete**, i.e., we couldn't just "draw" the positions from the output module. We have to fill up the gap between

the two mouse positions. To achieve this, we used the well-known **Bresenham's line algorithm** to implement it. (Refer to §A.2)

To implement Bresenham's line algorithm, we use an FSM to replace the for-loop. The simplified state transition diagram is shown in **Fig 2.3**. During the `WRITE` state, the drawing positions will be updated every cycle. The axis with more significant differences will be the step direction, and with sequential calculations, the algorithm will tell us the corresponding draw position. The state will return to the WAIT state after the draw position reaches the end position (the latest mouse position).

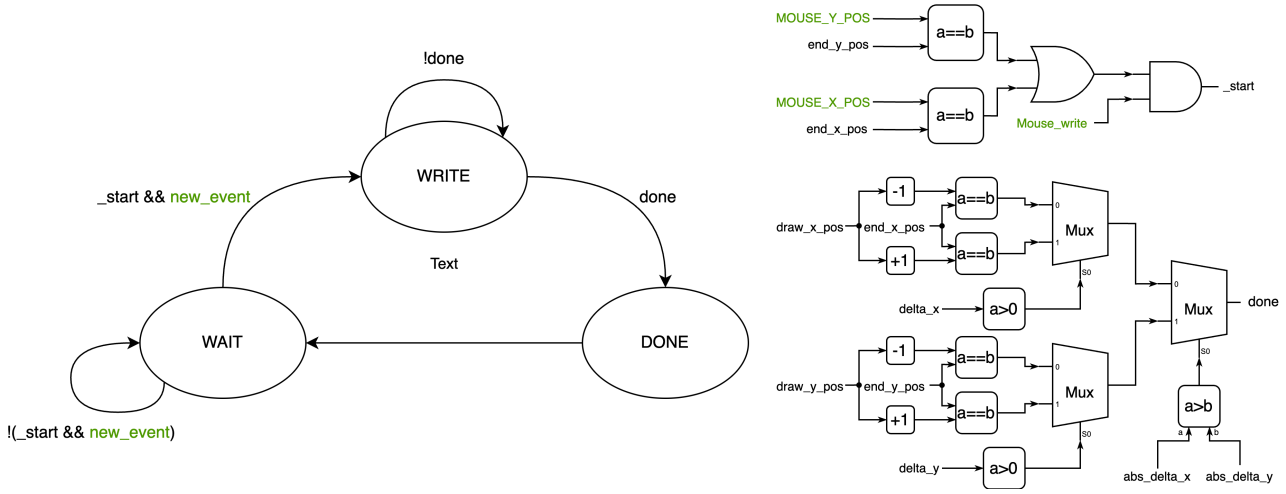

Fig 2.3 – Simplified state transition diagram for Bresenham's line algorithm

As mentioned above, the `canvas_input` module is the main module that controls one port of the "Canvas." The block diagram is shown in **Fig 2.4**. Since it controls the main port of the "Canvas," we must deal with the reset mode. When the `rst`, `ready_to_clear_canvas` (raised when done recognizing data), or `clear_block` is asserted, it will trigger the **reset address generator** shown on the left side of the diagram. The address generator counts from 1 to 1023 and back to 0 each clock cycle. Since the DRAM is asynchronous, the address 0 will be reset as soon as any reset signals are triggered. For regular mouse input, we have to check the block of the writing position provided by the Bresenham's module is equivalent to the block address so far in editing. If not, we should ignore the writing since it's out of bounds. Note that **resetting the block has a higher priority** than writing the block through mouse input.

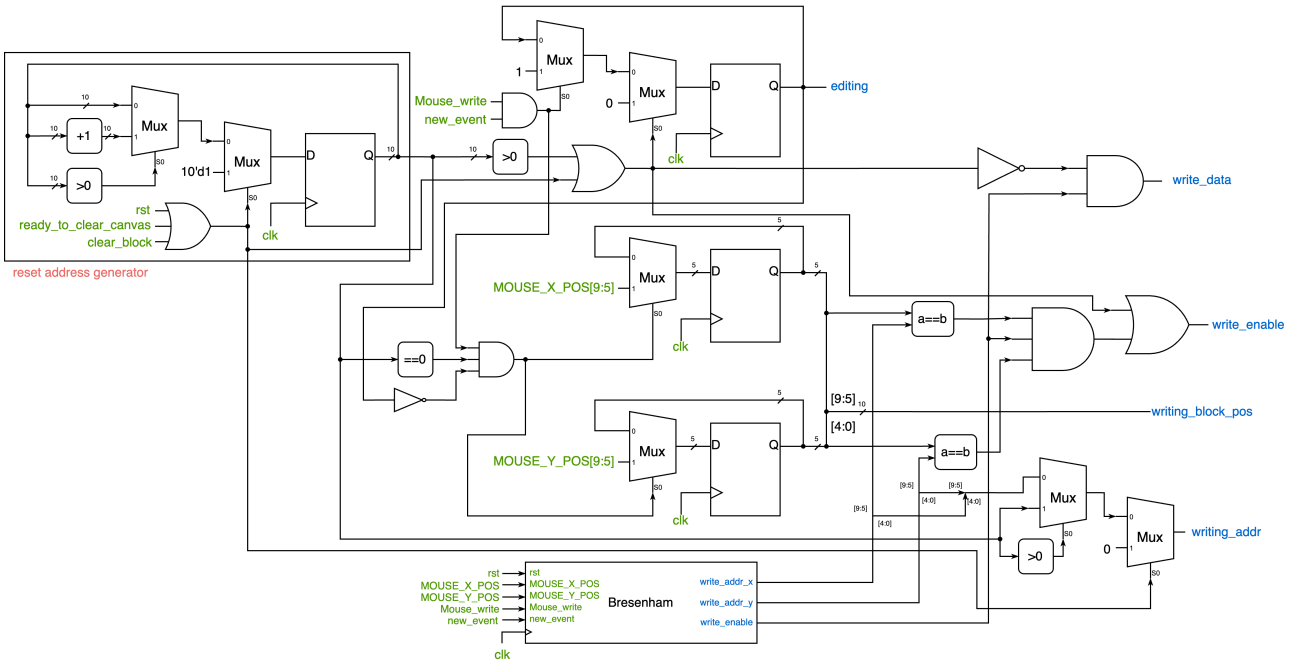

Fig 2.4 – Block diagram of the `canvas_input` module

§2.4 Pixel Generator

The `pixel_gen` module provides the correct pixel color for the VGA to display. The abstract module structure and the block diagram are shown in Fig **2.5** and **2.6**, respectively. In the first stage, the pixel generator will decide which data source to pick. If the position overlaps the mouse position, the mouse pixel will be fetched and illuminated. Then, the "Canvas" data will be taken if the block is editing. If the block isn't editing, it will fetch the Fonts block if the block is editing; otherwise, it shows the default colors. In the second stage, it determined whether to illuminate the margin colors. If the position is on the margin of any blocks, i.e., the last five bits of position is 0 or 31, the VGA should illuminate margin colors. To improve the user interface, we designed that when none of the blocks are being edited, the margin of the mouse's block position will illuminate blue. If a block is being edited, the margin of the block will be orange. This design allows the user to **trace the block in editing mode easily**. The whole structure is a combinational circuit to avoid clock delays during the request from VGA to data ready.

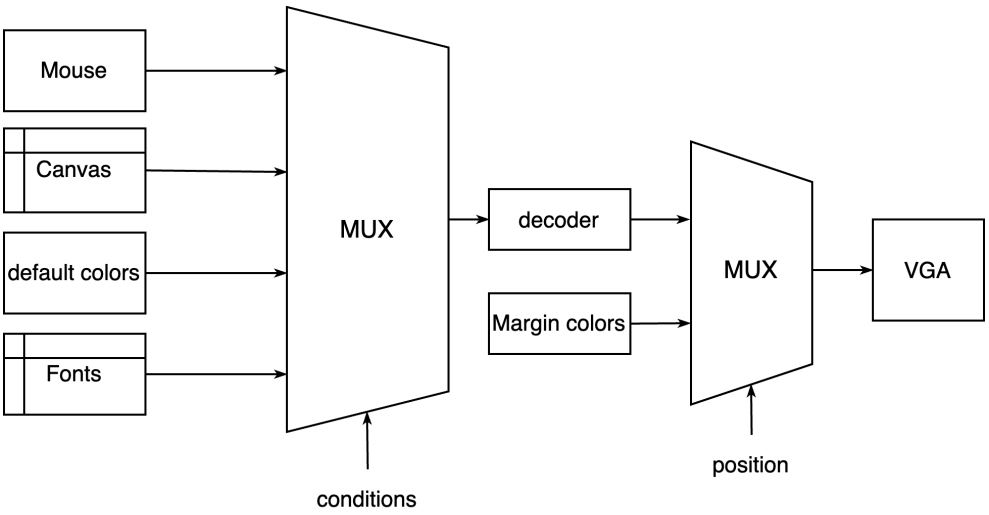

Fig 2.5 – Abstract module structure of `pixel_gen`

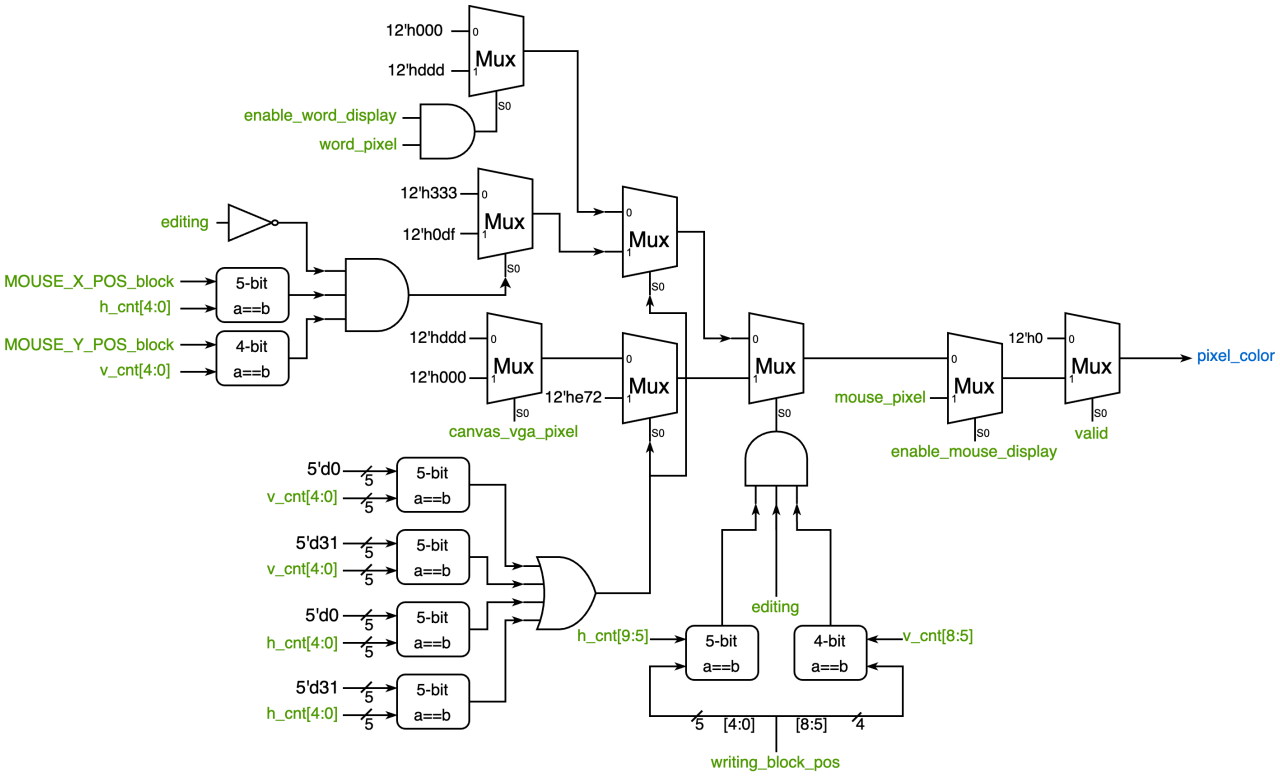

Fig 2.6 – Block diagram of `pixel_gen`

§3 Handwriting Recognizer

§3.1 Train Data

Most train data is obtained from a database[2] containing roughly 50000 images of 64x64 grayscale handwritten ASCII printable characters. In other words, it covers most characters from code point 33 (`'!'`) to 126(`'~'`).

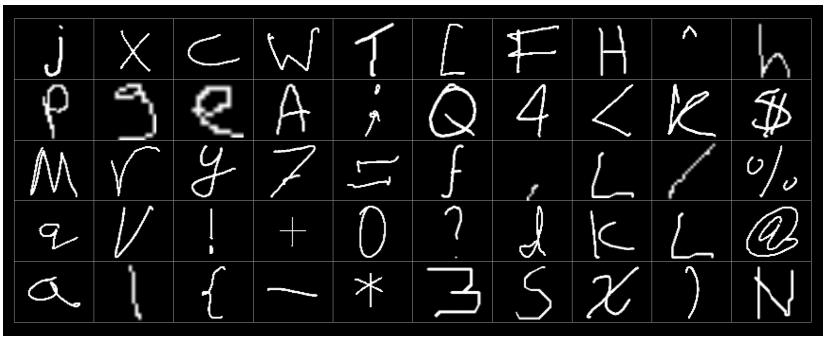

Fig 3.1 – A subset of our training data

However, it was not until we started training the model that we noticed the dataset didn't have data for characters 32 (`' '`) and 92 (`'\\'`). The quality of the database is also questionable, with some images categorized into the wrong class or even hardly recognizable. Therefore, we built a simple Python painter app (**Fig 3.2**) to draw the images ourselves. Each time, it randomly picks a character for the user to draw, with the weight of each class being chosen assigned to the reciprocal of the number of existing data in the class.

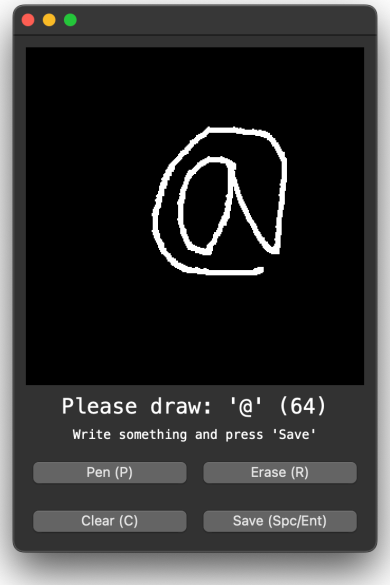

Fig 3.2 – Painter app for generating custom train data

---

[2] J. Sueiras, Handwritting characters database, (2016), GitHub repository, https://github.com/sueiras/handwritting_characters_database

During training, the Keras image augmentation layers are also utilized. By repeating the whole database and applying random zoom or translation, we could obtain an illusion of much larger train data, thus vastly improving the model's performance in recognizing the wider variety of inputs.

§3.2 CNN Model Structure

The best currently-known model structure for recognizing and categorizing images is the convolutional neural network (CNN) structure, which features one or more convolutional layers at the top of the layers. In contrast with the fully connected layers (i.e., dense layers), convolutional layers take advantage of the spatial locality information of the image pixels. With the assumption that "nearby pixels must be closely related" and vice versa, convolutional layers can significantly reduce the number of neuron connections while having better training performance than the dense layers.

During the design phase of the model, we referred a lot to a paper about implementing CNN on an FPGA board,[3] which utilized lots of techniques to make the hardware implementation easier. After some adjustments, we finally devised the model visualized in **Fig 3.3** and used some of the techniques presented in the mentioned paper.

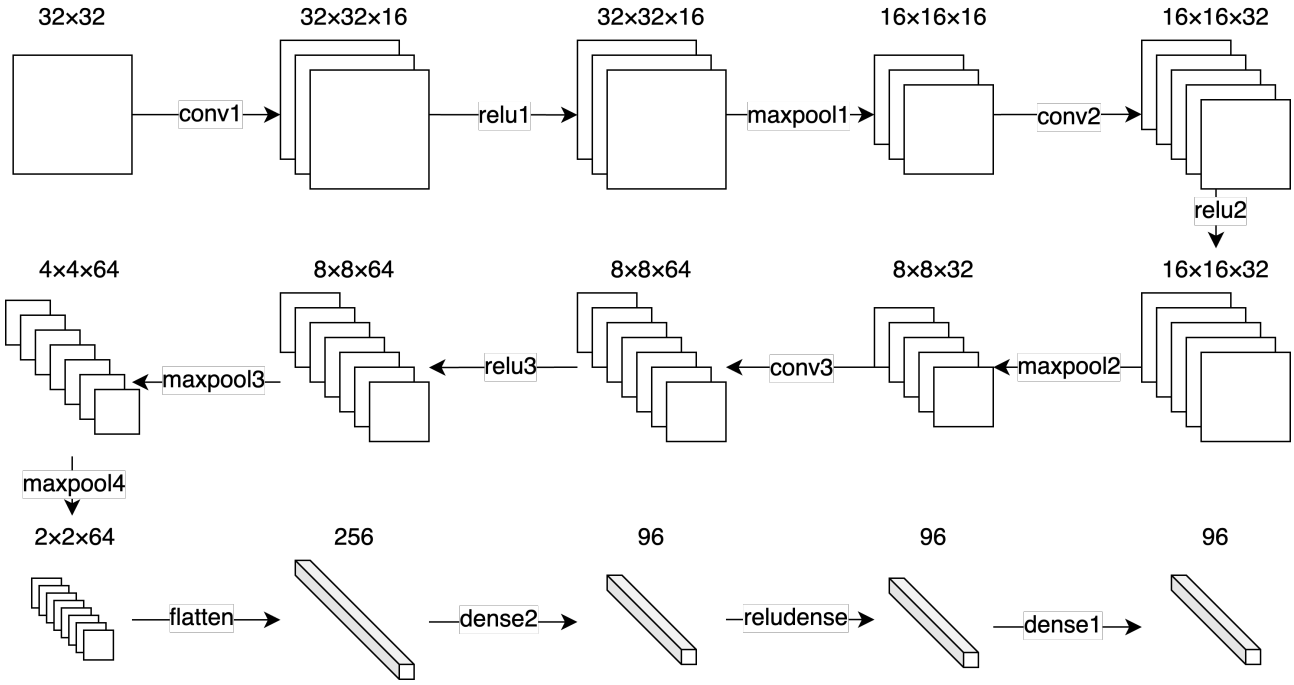

Fig 3.3 – The overall structure of the CNN model

All convolutional layers use same-padding, making output having the same height and width as input

[3] Yan, F., Zhang, Z., Liu, Y., & Liu, J. (2022). Design of Convolutional Neural Network Processor Based on FPGA Resource Multiplexing Architecture. Sensors (Basel, Switzerland), 22(16), 5967. https://doi.org/10.3390/s22165967

The size of the layers is carefully optimized such that the prediction accuracy is high enough while still not using too much computational resources. Also, a more **hardware-friendly version of the activation function (ReLU) and pooling layer (MaxPool)** are utilized to make the model even more efficient in hardware implementation. Also, the layers are put together in a regular pattern, allowing time-multiplexing of computational resources to be utilized.

Most layers have a **size of integral power of 2**, which would be handy when implementing the addressing of parameters and internal data. However, limited by the memory capacity of Basys 3 and the output size, we decided to make the last two dense layers have a size of 96.

Even though the train data is 64x64, the **input shape is downscaled to 32x32**. Indeed, 64x64 input provides a better resolution for prediction, but it has more cons than pros. To fit enough characters in the 640x480 VGA screen and make the whole thing accessible as a text editor, a smaller canvas size is needed. Also, larger input data implies more parameters at the dense layers and longer processing time, which would become unacceptable for a 64x64 input.

After training, the model provided achieved an accuracy of 90.6%. Even though the performance could be further improved by adding more neurons, optimizing the layer structure, or tidying up the train data, such accuracy is quite enough for practical use.

§3.3 Parameter Format

In a neural network, aside from the model structure itself, tens of thousands of "parameters" are also multiplied and added to the input in each layer. However, due to the limited resources on the FPGA board, some optimizations have to be done for the parameter.

Under the black box of a neural network, all the computations are just some real numbers multiplied and added together, and these real numbers are often represented with floating point numbers. However, floating point arithmetics is known to be time- and resource-consuming on hardware for a more accurate and flexible representation of real numbers, and our hardware implementation couldn't have afforded these heavy computational works. Therefore, another easier-to-work-with version of real number representation is utilized instead, that is, the fixed point representation.

Instead of scientific notations, fixed point representation scales, biases, and rounds the real number, such that it could be represented as a 2's complement signed integer. Our implementation stores the parameters as a **16-bit fixed-point number**, with one sign bit, one integer bit, and 14 mantissa bits. Such representation could express a range of numbers **between -2 to 2** and achieve a precision down to the 4th decimal place. We can derive the conversion formula from any real numbers (within the valid range) to fixed point format as $F = \text{int}(x \times 2^{14} + 0.5)$, where F is the

fixed point representation parsed as a 16-bit signed integer and x is the original real number. Adding 0.5 effectively rounds instead of floors the number.

The number of mantissa bits could have been anything other than 14. Nevertheless, such number is chosen according to the distribution of the parameter values, shown in **Fig 3.4**. Actually, some parameters would fall out of the range if no parameter constraints were added during the training phase. But these numbers are few enough for the model to operate nicely with the parameter constraint added.

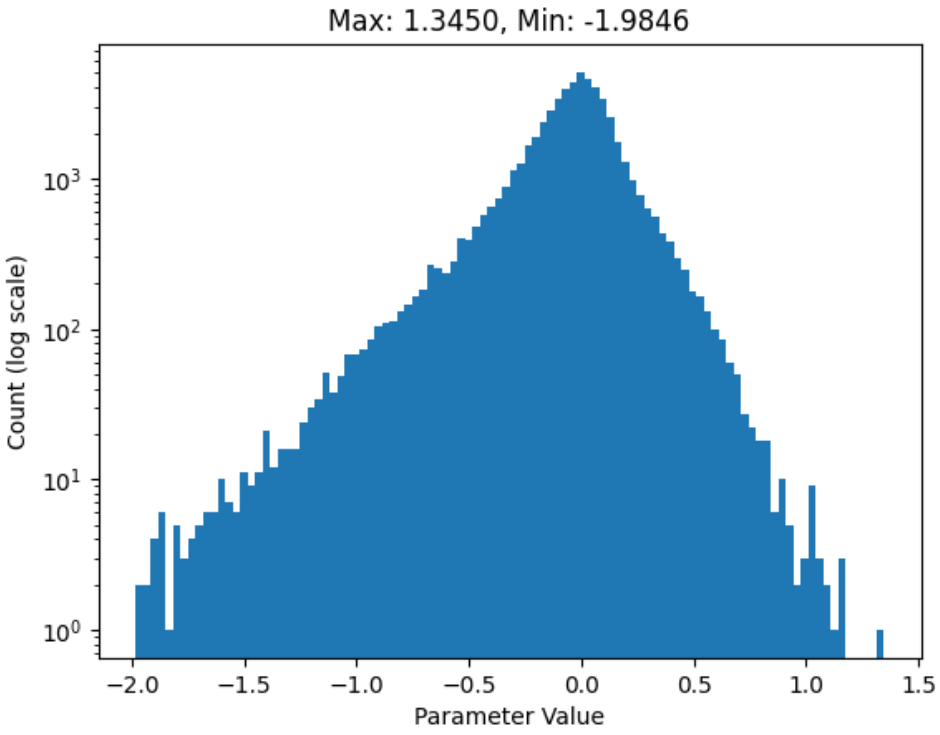

Fig 3.4 – Distribution of parameter values

Not only do the parameters need a storage format, though. There should also be a way to properly store all the intermediate values between each layer. Investigating the intermediate values by inserting some train data, we know these values range between −80 and 40. To simplify things, the number of mantissa bits is kept at 14, and the integer bits are extended to 7 bits to accommodate the larger value range, giving us a total of 22 bits.

Sacrificing the flexibility, the arithmetic operation on **fixed point numbers is much simpler to implement on hardware**. We can directly use the integer addition and multiplication algorithm with little pain – but not zero, since we still have to **shift back the multiplied product by 14 bits** to obtain the correct value. But for the rest of the operations, everything is easier with the fixed point format.

§3.4 Parameter Storage

Laid down the storage format of the parameters, it's time to put them into the FPGA. Here, we separated the memory by the parameter type (weights/bias) and the usage

of the parameters (convolution/dense). The detailed usage of each memory block is listed in **Tab 3.1**. The weights take up most of the memory usage, so they're put into BRAMs provided by the Artix-7 chip. As for the bias, they're relatively small and, therefore, could fit into smaller DRAMs. Notice that the weights for convolutional layers are accessed per kernel, which has 3*3=9 weights each.

Memory Block		Memory Type	Data Width	Memory Depth	
Convolution	Weights	BRAM (ROM)	16*9=144	1*16 + 16*32 + 32*64	2576
	Bias	DRAM (ROM)	16	16 + 32 + 64	112
Dense	Weights	BRAM (ROM)	16	256*96 + 96*96	33792
	Bias	DRAM (ROM)	16	96 + 96	192

Tab 3.1 – The memory usage for each parameter memory

Besides, we also need a way to address these data. To make the Verilog code maintainable while ensuring that the resource usage is within the limit, the parameters are addressed with two layers of conversion.

The memory wrapper, whose abstract structure is shown in **Fig 3.5**, provides a straightforward interface for accessing the correct data only with the input/output channel of the currently processing layer and the FSM state that contains the information of which layer is currently being processed.

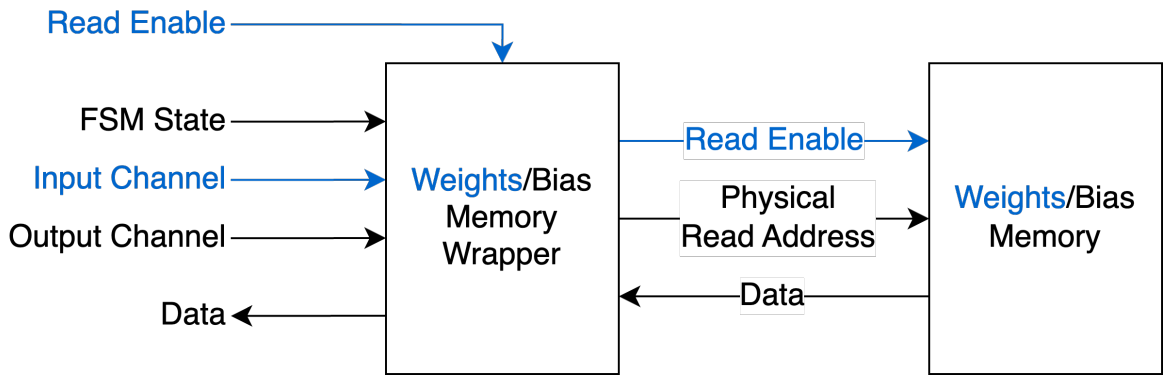

Fig 3.5 – Two-layer addressing of parameter memory
Items marked blue only appears in weights memory.

The implementation details of the address conversion will be skipped in this document. The key point is, though, that these values are **concatenated bitwise and added with an offset** to generate the corresponding address. (Except for the layer `dense1`, which multiplies the desired output channel by 96.) This addressing method greatly reduces the consumption of computational resources.

§3.5 Feature Map Buffer

The intermediate values between each layer also require some storage memory, which we'll call the "feature map buffers." In the implementation, we placed two

feature map buffers that exchange their values repeatedly. The two feature map buffers are reused for each layer, and the details will be discussed in §3.6.1. The memory usage of the feature map buffers is shown in **Tab 3.2**.

Memory Block	Memory Type	Write Width	Read Width	Memory Depth
Feature Map Buffer (Convolution)	BRAM (Simple Dual Port RAM)	22	22	4096
Feature Map Buffer (Pooling)	BRAM (Simple Dual Port RAM)	22	44	16384

Tab 3.2 – The memory usage for feature map buffers

Note that in **Tab 3.2**, the read width of the pooling feature map buffer is 44 instead of 22. Such design decision is made according to the mechanism of the 2x2 max-pooling layer. From **Fig 3.6**, we know that each layer output is produced from 4 values in the input data. Conveniently, the four values are actually two pairs of **consecutive data in memory**, as long as we address the data by its Y coordinate and then its X coordinate. With that, we can reduce the memory access for each output from 4 accesses down to 2 accesses.

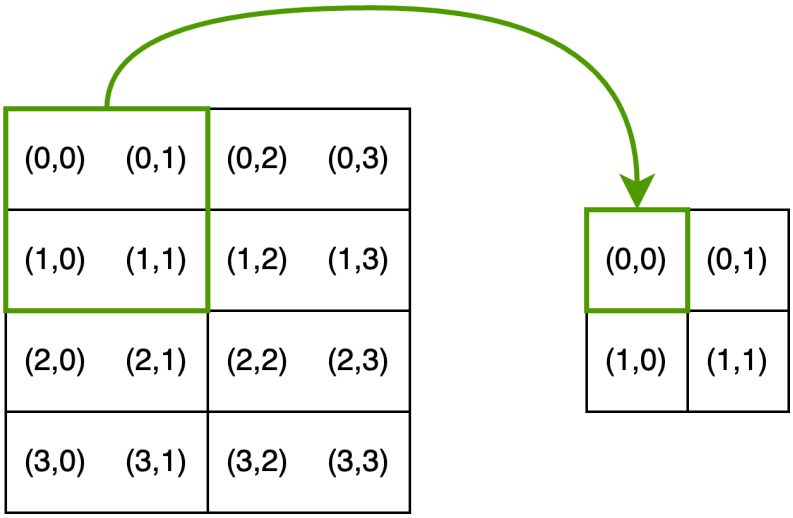

Fig 3.6 – Visualization of 2x2 max-pooling

Similar to the parameter storage, feature map buffers are also addressed with the mentioned two-layer addressing technique, but some additional inputs are required for complete addressing. These inputs include the X/Y coordinate on a bitmap, as well as "read shift" and "read up/down" for the convolutional and pooling feature map buffer correspondingly. Also, since we're starting to write some data into the memory, the write port and read port are both presented to the recognizer module.

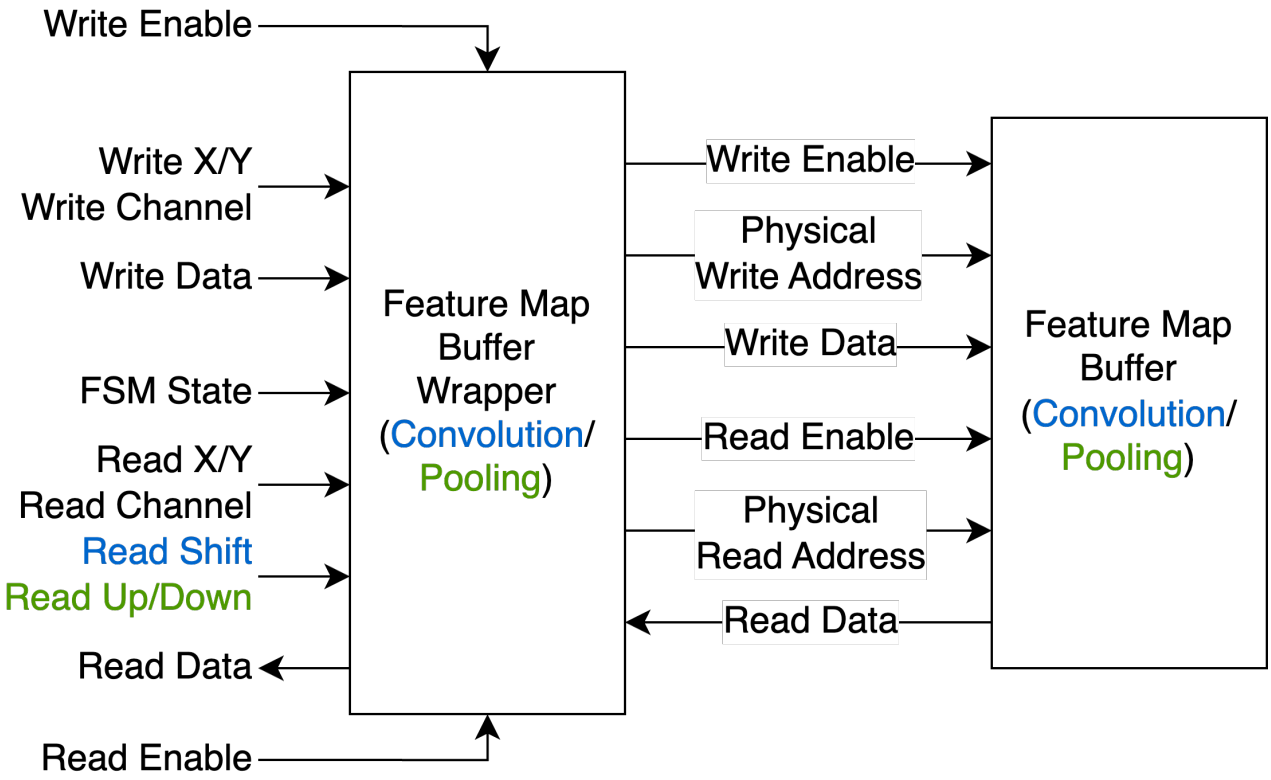

Fig 3.7 – Two-layer addressing of feature map buffers
Item marked blue only appears in convolutional feature map buffer.
Item marked green only appears in pooling feature map buffer.

"Read up/down" is implemented according to the mechanism shown in **Fig 3.6**. The **read addressing of the feature buffers is based on the output coordinates**, but it needs to read two different memory locations to compute the correct result. By adding the read-up/down input, we can easily access the two pairs of values.

"Read shift" is related to the calculation of convolutional layers. A 3x3 convolutional kernel multiplies the 3x3 nearby pixels by the corresponding weights and then sums them up as the output. The read shift input is required to simplify the access of nearby pixels and implement the same-padding feature of the convolutional layer. As shown in **Fig 3.8**, the read shift is projected to the 2-dimensional offset from the target coordinate from 0 to 8.

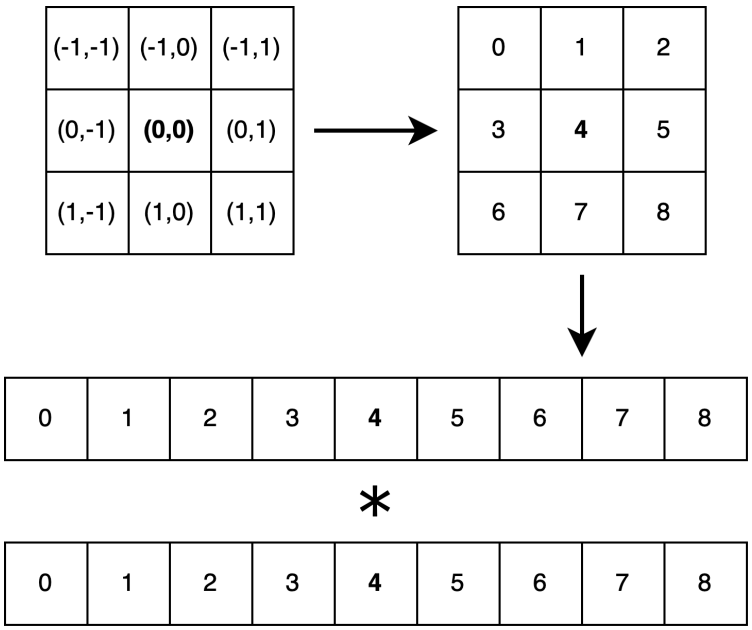

Fig 3.8 – Indexing of offsets (shifts) for convolution

§3.6 CNN Implementation

§3.6.1 Overall Structure

The CNN contains eight layers containing millions of calculations, which must be distributed over time. The bottommost label in **Fig 3.9** is the **state name in the recognizer's FSM**, and the label on top is the layer name in our model shown in **Fig 3.3**. The time diagram shows the **ping–pong data transmission between the two feature map buffers**. Each data transmission is processed through a convolution unit or a ReLU/max–pooling unit. With the symmetric design of the model, we can simplify the process as follows: During convolutional stages, the convolutional feature map buffer sends data to the convolutional unit, the output of which is sent to the pooling feature map buffer; During pooling stages, the data in pooling feature map buffer is processed by the ReLU/max–pooling unit, and the output is sent to the convolutional feature map buffer.

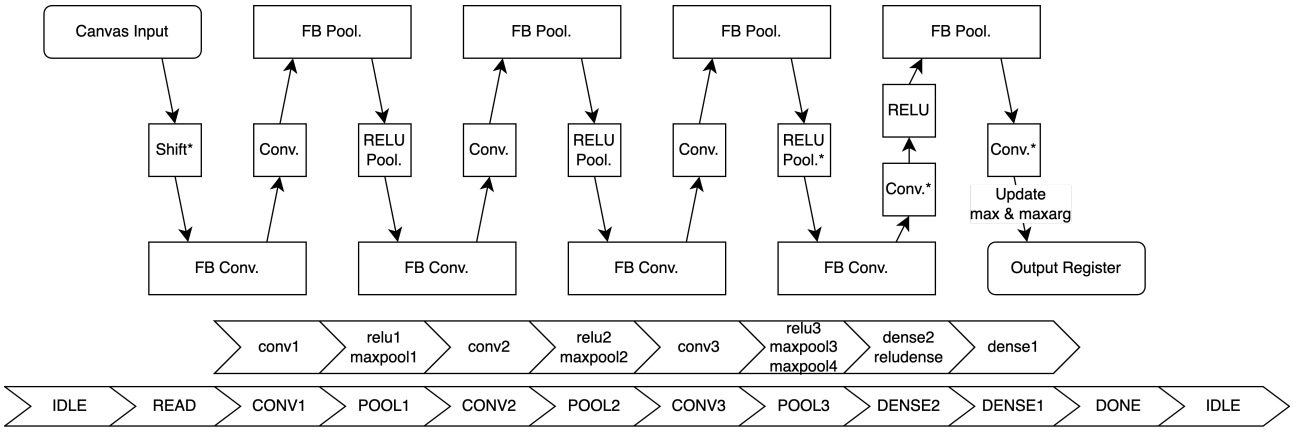

Fig 3.9 – Time diagram of the whole CNN algorithm

In the `recognizer_core` module, there are two counters, counter and subcounter, that form a **two-level for-loop for each layer**. Once `subcounter` reaches its maximum value according to the current state, it will reset to 0, and `counter` is incremented by 1. If both counters reached maximum value, the FSM would transition to the next state, i.e., the next layer. The maximum value of the counters in each state is tabulated in **Tab 3.3**. The maximum value for `counter` is closely tied to the number of outputs in each layer, whereas the maximum value for `subcounter` is related to the operation of the layer.

	CONV1	POOL1	CONV2	POOL2	CONV3	POOL3	DENSE2	DENSE1
`counter`	16383	4095	131071	2047	131071	255	95	95
`subcounter`	11	4	11	4	11	17	259	99

Tab 3.3 – The maximum value of `counter` and `subcounter` according to FSM state

However, there are some exceptions to the symmetric structure. At `POOL3`, it does a 4x4 max pooling instead of 2x2; therefore, it requires additional hardware. The details are shown in **Fig 3.14**. At the two dense layers, the convolution unit is reused in a slightly different way. The details will be discussed in **Fig 3.15** and **Fig 3.16**.

§3.6.2 Convolutional Layers

The convolution operation on a 3-dimensional tensor (width, height, and channel) could be expressed as follows.

$$x_{ijc}^{n} = \sum_{d=0}^{D} (\sum_{s=0}^{8} y_{sd}^{n-1} w_{s}^{n}) + b_{c}^{n}$$

Here, x_{ijc}^{n} is the output tensor of the convolutional layer, i, j is the X/Y coordinate, c is the output channel, D is the number of input channels, s is the shift indexing described in **Fig 3.8**, w_{s}^{n} is the weight of the layer according to the shift index, b_{c}^{n} is the bias of the layer according to the output channel, and y_{sd}^{n-1} is the input from the previous layer.

Complicated as it seems, by expanding the nine multiplied terms to a vector, it's just a **sum of dot products** of kernels and inputs in different channels. With that in mind, the sequential vector multiplication unit could be designed as shown in **Fig 3.10**. It comprises a **fixed-point multiplier** at the left and an **accumulator** at the right. Such organization is also utilized in the matrix multiplication in the dense layers, which is basically just more vector multiplications.

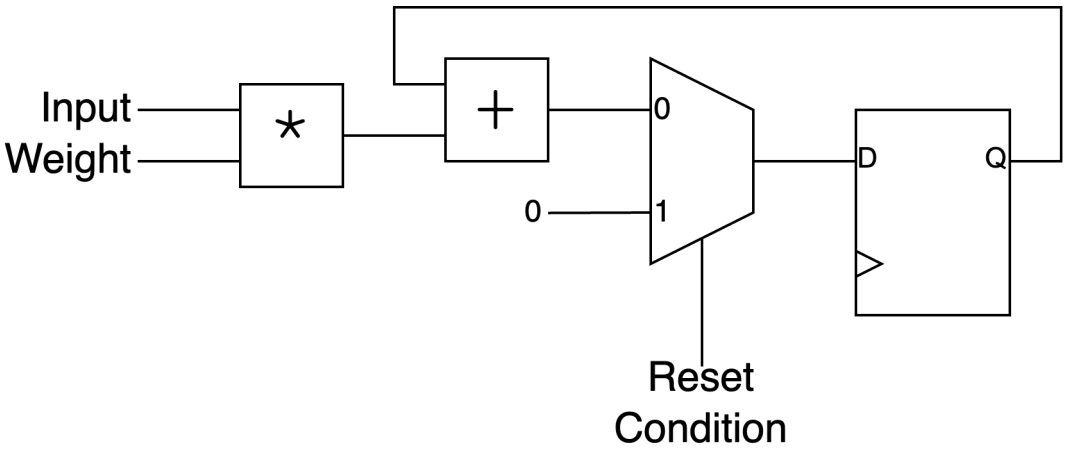

Fig 3.10 – Block diagram of convolution/vector multiplication unit

At the beginning of every convolution, or vector multiplication, the accumulator is reset to 0. During the calculation stage, the weights and inputs from the previous layer are sent pair by pair to the multiplier for each cycle until all the multiplied products are added to the accumulator. The result of the accumulator is then added by the bias (not shown in **Fig 3.10**) and stored in the pooling feature map buffer.

A simplified waveform of the convolutional layers is shown in Fig 3.11, where we ignore the fact that input could have more than one channel. However, it's still clear enough to demonstrate the accumulating process mentioned above. Note that the output of the accumulator (`acc`) is actually **delayed by two clock cycles**. This is because fetching data from the BRAMs would have taken too much time, with little time remaining for the multiplication, causing a timing violation. By **adding some registers in between and separating the works into multiple cycles**, the "pipelining" technique could significantly reduce the critical path length within a clock cycle, maintaining a well-behaved sequential behavior.

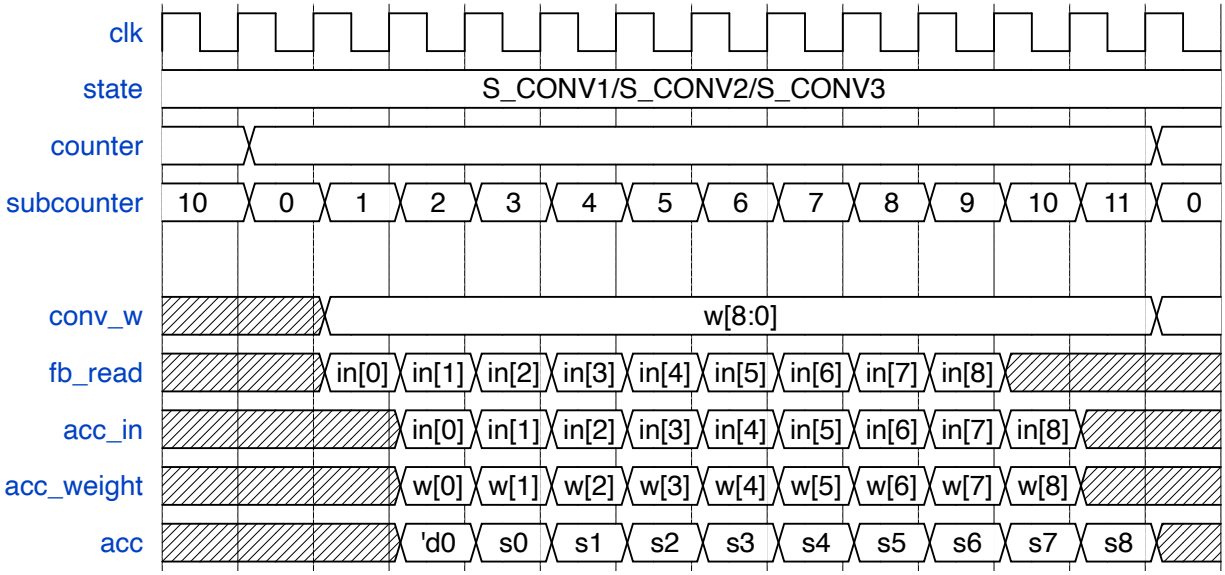

Fig 3.11 – Simplified waveform of the convolutional layers

§3.6.3 Pooling/Activation Layers

The ReLU function is used as the activation function of our neural network, which can be expressed as follows.

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

Since our data is stored as a signed binary integer, we can easily decide the sign of the number from the MSB and obtain the negated value with the 2's complement.

Max-pooling and ReLU activation are done together in the same pooling phase to reduce the processing time. The data flow in the pooling stages from the pooling feature map buffer to the convolutional feature map buffer is shown in **Fig 3.12**. Similar to the convolution unit, the pipeline registers (`max_in` and `relu_reg`) are added to comply with the time constraint, producing the waveform shown in **Fig 3.13**.

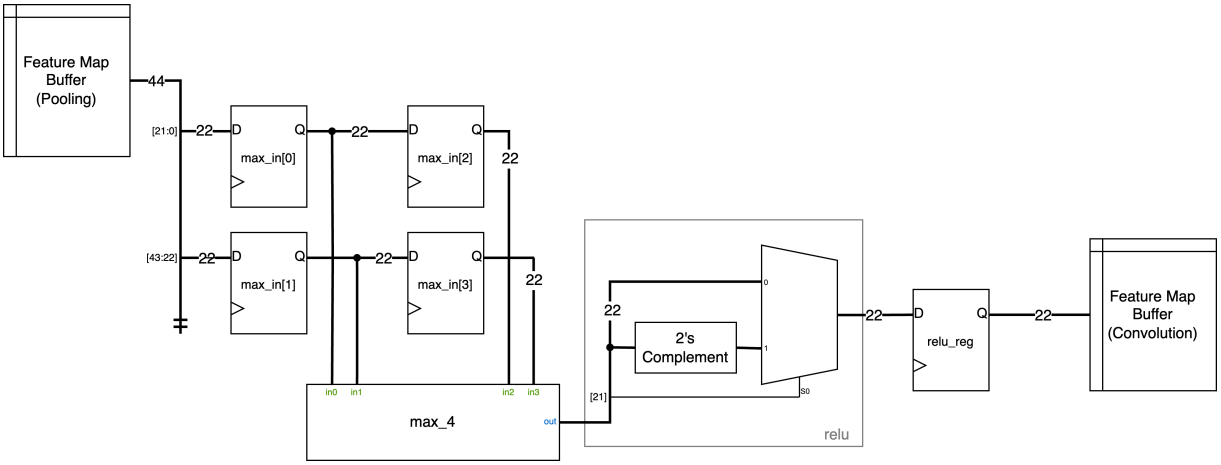

Fig 3.12 – Block diagram of the pooling unit

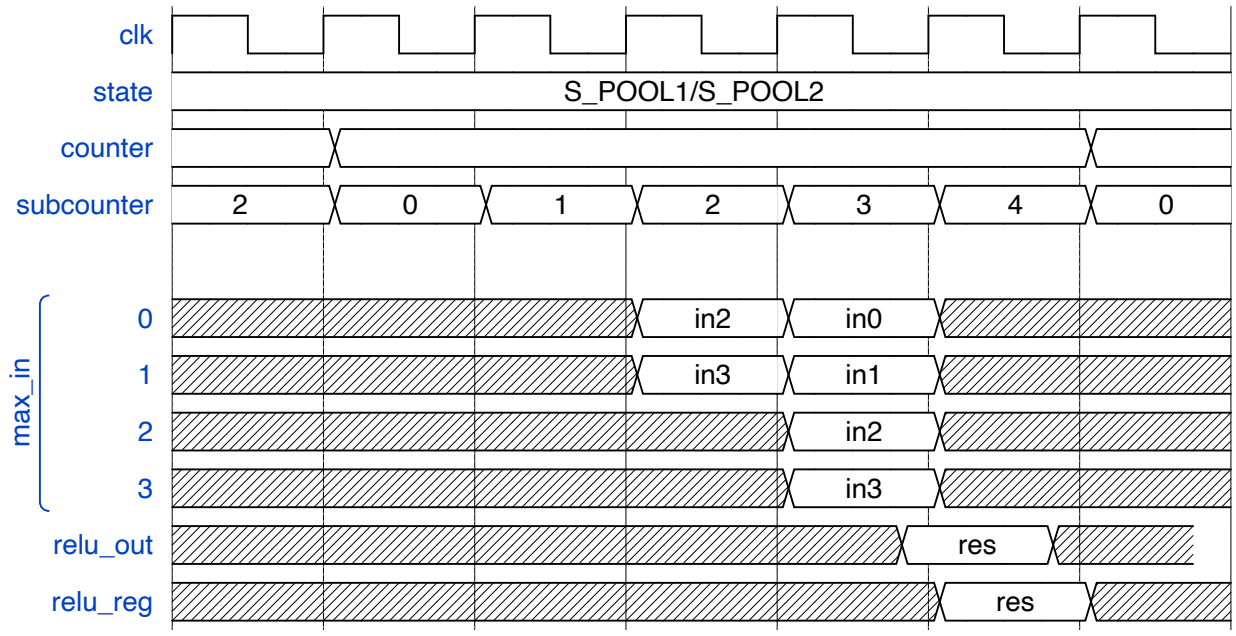

Fig 3.13 – Waveform of pooling layers `POOL1` and `POOL2`

However, for `POOL3`, instead of a 2x2 max-pooling, we're doing a 4x4 pooling here. Extending from the 2x2 pooling, every time a "sub-pooling" is complete, the **`max_out2` register compares it with its own value and updates itself with the larger one**. The initial value of `max_out2` in each computation cycle is set to 0 since the output of the 2x2 pooling has also gone through the ReLU function and, therefore, must be greater or equal to zero. One clock cycle after the four "sub-pooling" cycles, we can obtain the correct max value in a group of 4x4 pixels.

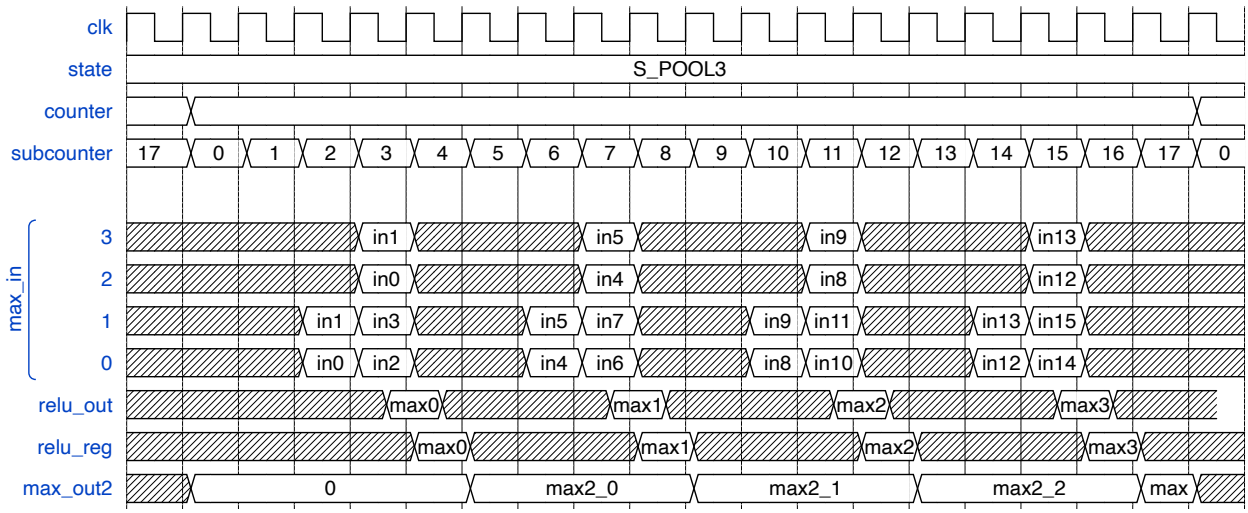

Fig 3.14 – Waveform of the 4x4 pooling layer `POOL3`

§3.6.4 Dense Layers

The output of a dense layer could be expressed as follows.

$$x_j^n = (\sum_{i=0}^{N} y_i^{n-1} w_{ij}^n) + b_j^n$$

Here, x_j^n is the output vector of the convolutional layer, N is the size of the input vector y^{n-1}, w_{ij}^n is the weight of the layer according to the input index i and output index j, and b_c^n is the bias of the layer according to the output index. This is just another dot product calculation, and it can be done with the same hardware shown in **Fig 3.10**. In other words, a dense layer could be treated as a special case of **1-dimensional convolution along a 2-D tensor**. Therefore, we could obtain a similar waveform for the dense layer, shown in **Fig 3.15**.

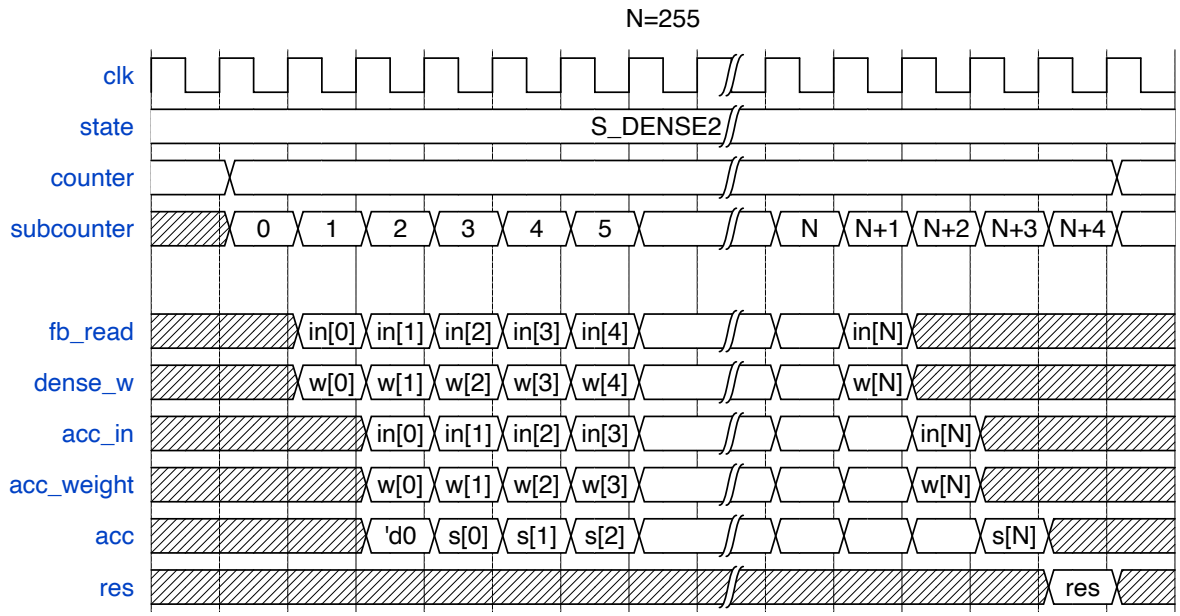

Fig 3.15 – Waveform of the dense layer `DENSE2`

For the final layer `DENSE1`, aside from the "convolution" process, we also maintain a max value along with its index `maxarg`. Since this is the last layer of the whole network, we don't have to store the result anywhere for later use, and the entire process could be seen as another "**second-level max-pooling" similar to the `POOL3` layer**. The waveform is shown in **Fig 3.16**.

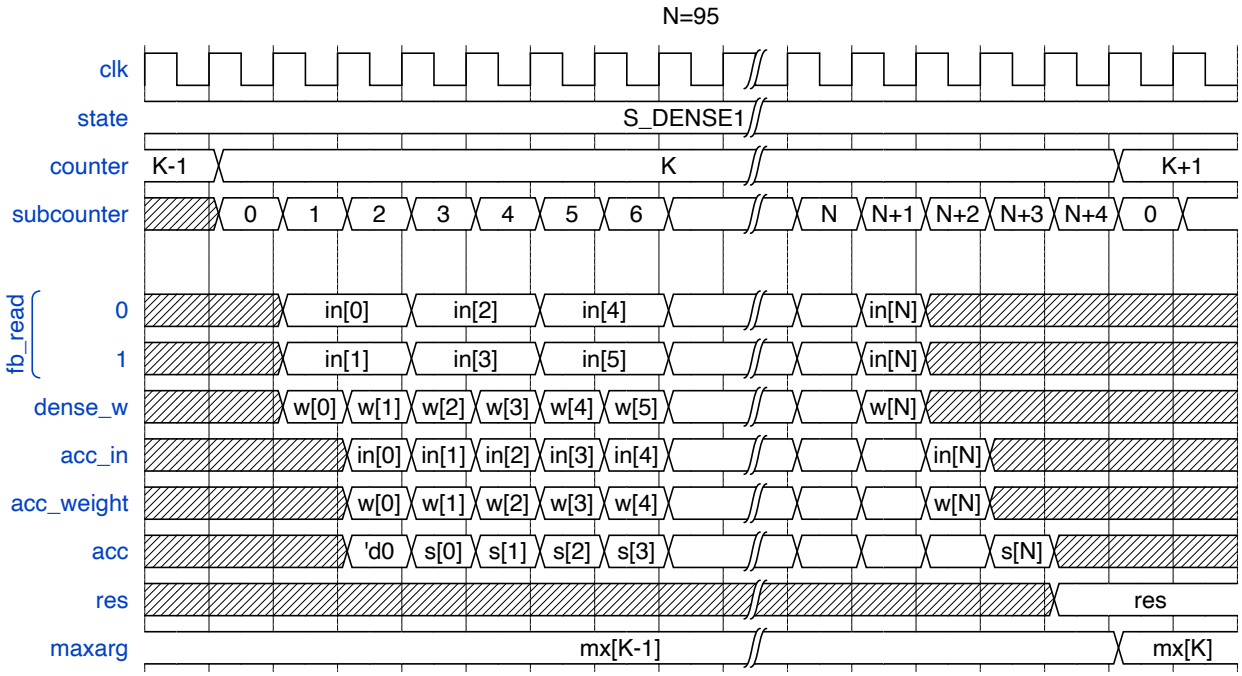

Fig 3.16 – Waveform of the dense layer `DENSE1`

§3.7 Testing Model Correctness

Instead of implementing the whole network at once and praying for it to work, we can build it one layer at a time and examine the correctness with a simulation. With the simulation tool provided by Vivado, not only can we view the waveform of various signals, but the content of the BRAMs is also **exposed via the "Objects" window**. As shown in **Fig 3.17**, after finding the "`native_mem_module`" scope in the "Scopes" window and searching for "memory," all the contents are conveniently grouped in the `memory` item.

With this information, we could compare the calculation result of the Verilog code with the original result from Python. We could even obtain the rounding errors from the fixed point arithmetics, which is about 0.01 at maximum. Such error is negligible in practice, and the correctness of the model is not affected much by the errors, thanks to the **redundancy nature of the neural network and its capability of dealing with noises**.

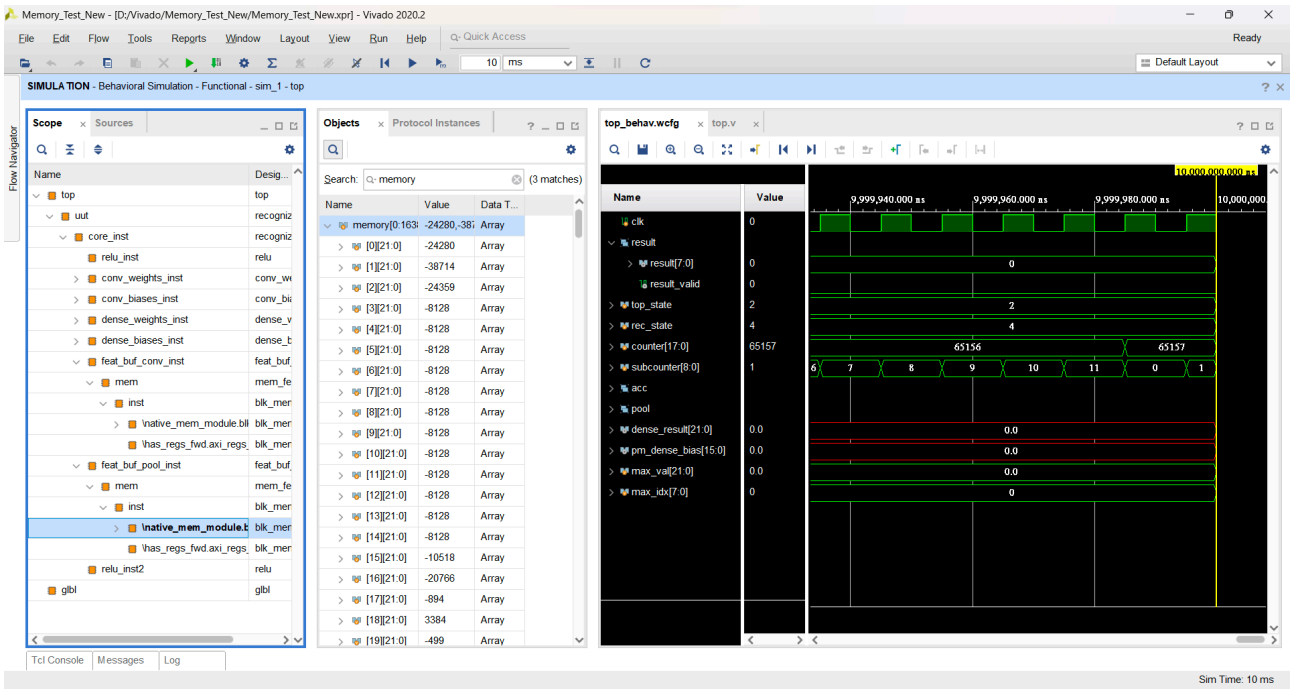

Fig 3.17 – Viewing the content of pooling feature map buffer via "Objects" window

§4 Text Data Communication

§4.1 Text Editor

The text editor is the main module that controls one port of the "Document" memory block, whose block diagram is shown in **Fig 4.1**. The primary input method is from the recognizer. When the recognizer has completed recognizing the "Canvas," the result ASCII number (biased by 32) will be outputted. The text editor will pick up the result and write the data to the "Document." The primary output method through this module is the UART module. When the user presses the send button, the UART will request the data from the "Document." The text editor will catch these requests and output the corresponding data to the UART. We decided to let these two I/O methods share the same port because both of these occur rarely compared to displaying the data. It is almost **impossible to do both UART read and user write simultaneously** since usually UART will be triggered when the user has done all the writing. Also, the time for sending the data is almost negligible compared to the time the user spends writing a character.

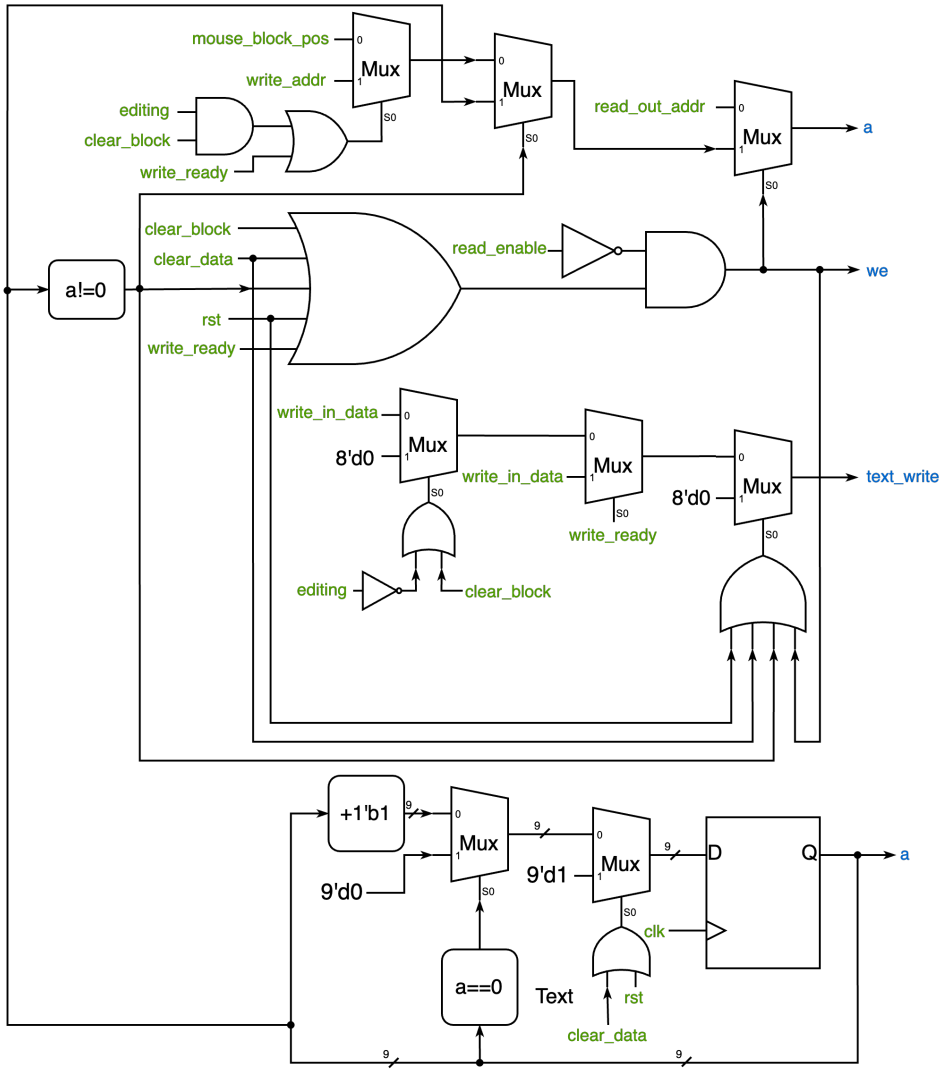

Fig 4.1 – Block diagram of `text_editor` module

§4.2 USB–UART Communication

To send the text data stored in "Document" to the host computer, we must utilize the RS–232/UART interface provided by Basys 3.

RS–232 is a serial communication protocol, meaning that data is sent **one bit at a time over a single wire**. It is typically asynchronous, meaning the data transmission timing is not synchronized between the transferor and the receiver. Instead, a **start bit 0 and a stop bit 1** are used to frame each data byte. The byte order is not specified in the RS–232 protocol, but we've implemented it so that the **LSB is sent first**. As for the data transfer rate, a unit "baud" describes how many bits could be sent within a second. In our implementation, the **baud rate is set to 230400**. The protocol also supports parity bits and different stop bit lengths, but we'll not go into detail in this document.

For the example waveform in **Fig 4.2**, the wire RsTx transmits a byte according to a clock signal `baudclk`. Counting from right to left, we know that it's sending a byte `0b10111010` here. On the other hand, the receiver knows a byte is coming from the start bit 0 and could adjust its baud clock to the correct phase for synchronizing the read process.

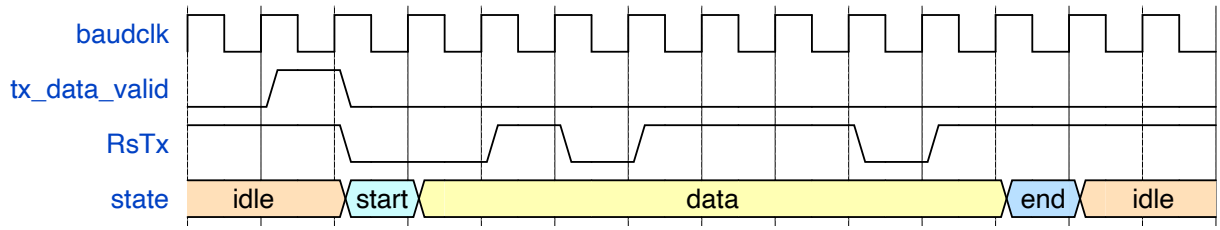

Fig 4.2 – Example waveform of the RS–232/UART protocol

Fig 4.3 shows the block diagram of the main UART communication module. With a central baud clock generator that produces a 230400Hz clock signal, the `uart_tx` and `uart_rx` modules could synchronize their data transfer/receive. The details for these two modules will be discussed shortly after. In the implementation, uart_rx is omitted since we're not **receiving any data** from the host computer. Nevertheless, it's still good to know its mechanism.

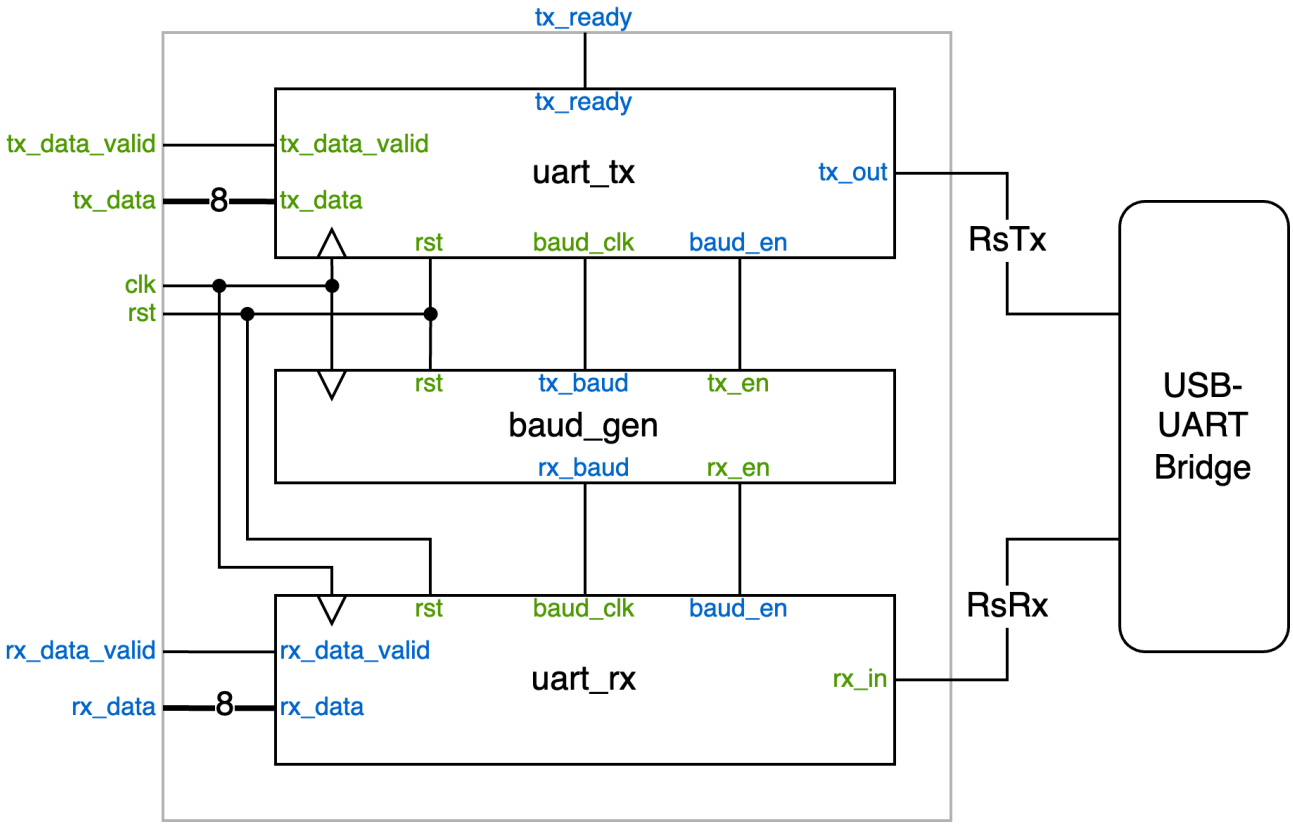

Fig 4.3 – Block diagram of `uart_top`

In actual implementation, `RsRx` related hardwares are omitted.

The `uart_tx` module accepts an 8-bit input `tx_data` and will send it through the `RsTx` wire. When it is at the IDLE state, `tx_ready` is asserted to acknowledge the top module that it is ready to send something. Once the input `tx_data_ready` is asserted, the module will transition to the SEND state and push the 8-bit `tx_data` to a shift register `tx_shift_reg` along with the start and stop bit. During the SEND state, at every positive edge of `baud_clk`, the module shifts the value in `tx_shift_reg` one by one, from LSB to MSB, until all ten bits are shifted out and sent through the `tx_out` wire. After that, it returns to its IDLE state, waiting for the next byte to send.

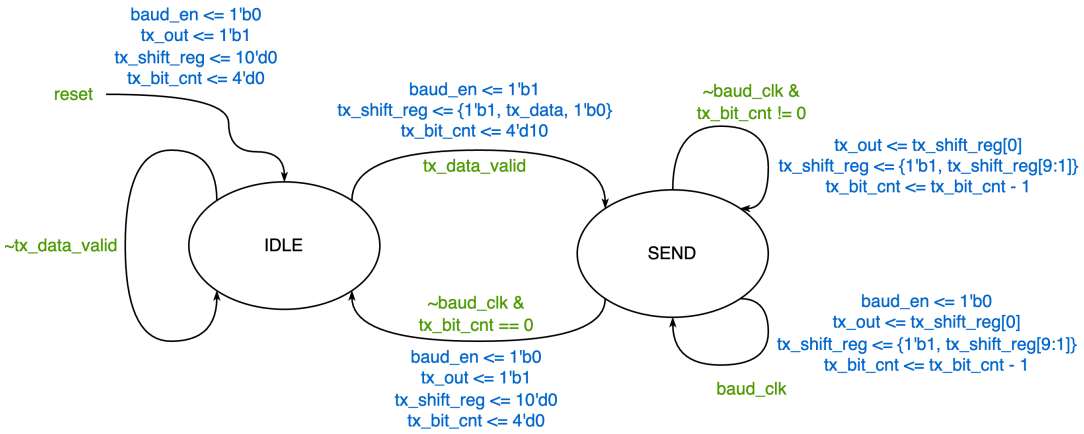

Fig 4.4 – State diagram of `uart_tx`

Similarly, the `uart_tx` module receives external data from `rx_in` and exposes the parsed byte data through `rx_data`. When a negative edge of `rx_in` is detected, the module goes from IDLE state to RECV state and enables the baud clock `baud_clk` at the same time. For each positive edge of the baud clock, it reads the data from rx_in and shifts it into a shift register `rx_shift_reg`. After ten bit-shifts, `rx_shift_reg` should contain the 8 data bits and the two frame bits. Then, we can extract the data bits and store them in `rx_data`.

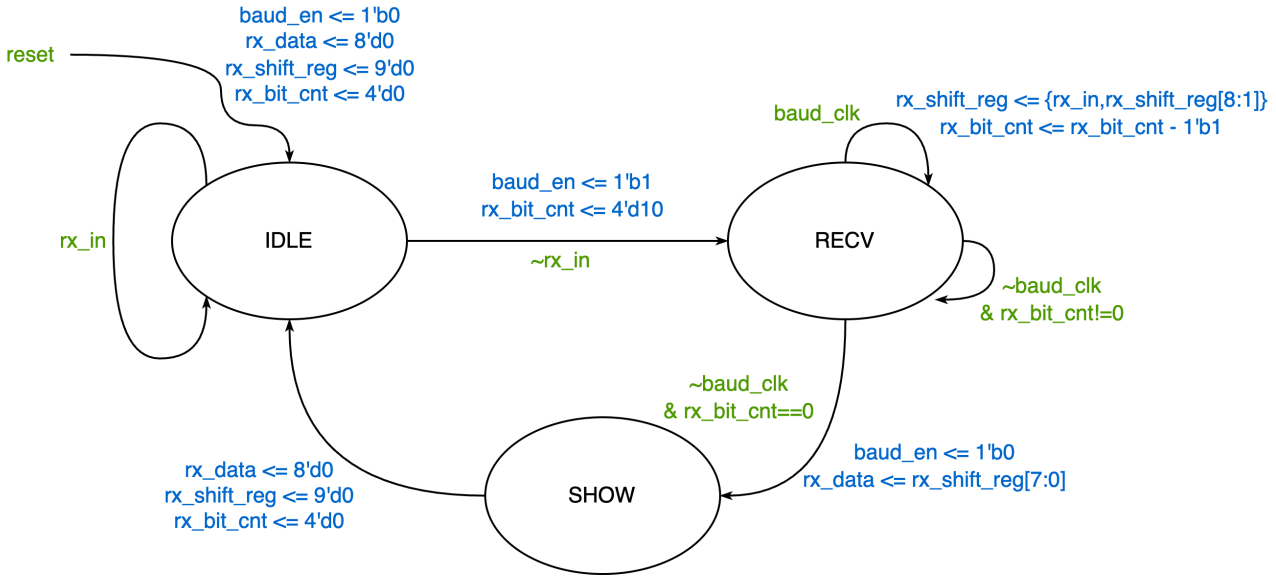

Fig 4.5 – State diagram of `uart_rx`

The whole UART communication module is wrapped by the messenger module, which handles the byte-level data transfer protocol (in contrast to the bit-level protocol of RS-232/UART.)

We define a data packet to have the following fields: 1 **start byte `0xCC`**, 300 data bytes of ASCII characters, and one **end byte `0xDD`**. The special frame bytes (start byte and end byte) are chosen to be **greater than 127**, the maximum value of the regular ASCII characters. `messenger` will soon start a transmission cycle when the user decides to send the current data stored in FPGA to the host computer, as shown in the state diagram in **Fig 4.6**.

Initially, `messenger` will start a send-byte request by asserting the `tx_data_valid` signal at `uart_tx`, sending the ACK signal, or the start byte `0xCC`. After that, the positive edge of `tx_ready` acknowledges `messenger` that it's ready to send the next byte. Here at READ state, two counters, `read_x` and `read_y`, are used to address the data in "Document," which are stored as a biased ASCII value. Therefore, we must add the bias 32 back before giving it to the `uart_tx` module. Similarly, we should wait for `tx_ready` to be asserted before sending the next byte. Finally, an EOF signal is sent as an end byte `0xDD`.

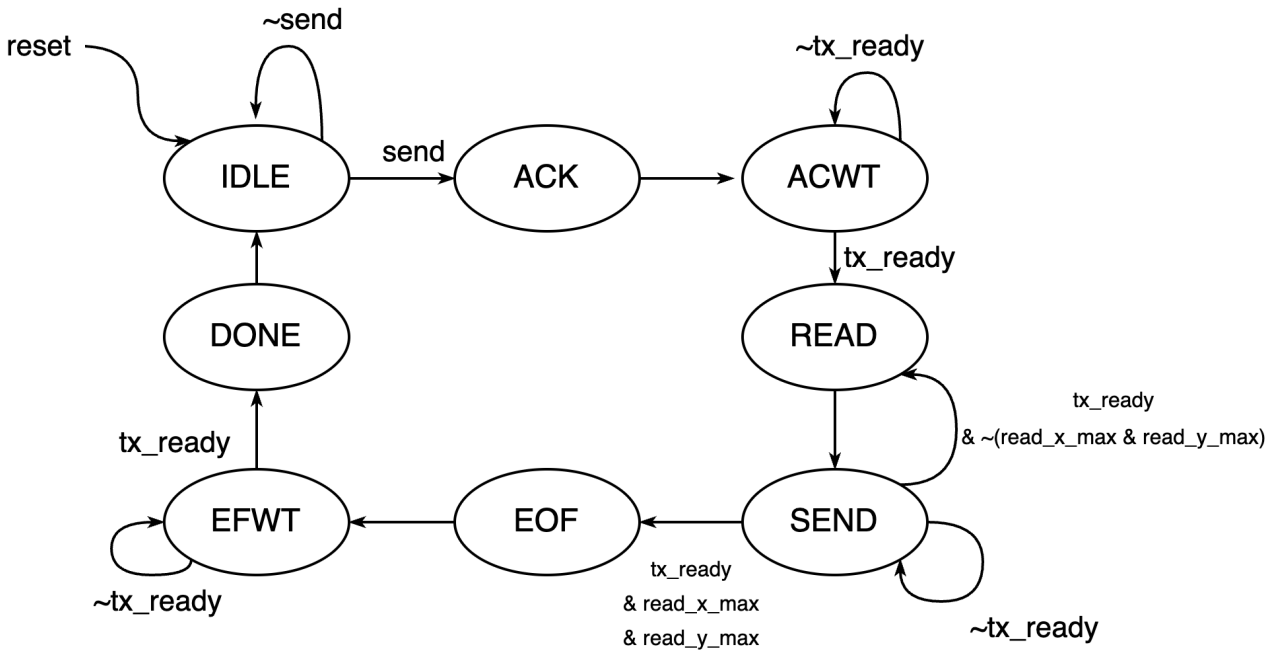

Fig 4.6 – State diagram of `messenger`

§4.3 Connect to Python Terminal

At the host computer, we should receive and parse the data sent from the FPGA via the UART interface. Thankfully, a Python module, **PySerial**, can read and decode the transmitted data. By configuring the correct port name, baud rate (230400), byte size (8), stop bits (1), and parity (none), it will automatically parse the input from the USB cable according to the protocol, and allow us to read data from a data stream. The details of the receiving process are already covered in §4.2.

In the previous section, we've regulated the data protocol, with each data packet having 300 data bytes and two frame bytes. Since for each read from the stream, we're **not sure if all of a packet has arrived**, we need to **queue these data until an end byte is acknowledged**.

Once we receive the end byte, the 300 characters will be split into 15 lines, each containing 20 characters. Before actually executing the code, the empty lines are dropped, and all the space characters after each line are also trimmed. Then, the interpreter first checks if the input code is an expression (something like 1+2) by the built-in `eval` function in Python. If nothing goes wrong, we know it is an expression, and the evaluated output is displayed in the terminal. Otherwise, it might still be a valid statement (something like `a=1`), and it's executed with the built-in `exec` function. If anything goes wrong, we know it must be the user's fault for inputting an invalid Python code.

Note that for the execution of the code to work as intended, a **scope object should be provided** to both `eval` and `exec`. This makes each code submission share the same execution environment. We can even add some pre-defined aliases by adding

some entries to the initial scope. Here, we added aliases for `print` (`P`) and `range` (`R`), both of which are frequently used functions in Python.

The Python app provides syntax highlighting (supported by the pygments module) and some colored text (supported by the termcolor module), which makes the whole thing look better. It could also detect available ports and even directly decide the port if only one port is open. However, if there are multiple available ports, the user is responsible for inputting the correct port name in the terminal

§5 Conclusion

§5.1 Project Demonstration

To demonstrate, we wrote a code that outputs the Fibonacci sequence. The result is shown in **Fig 5.1**. All data is successfully recognized and sent to the Python shell through UART. During the testing, the recognizer sometimes outputs incorrect results. For example, when we write the letter "o," the recognizer could think that the data is "0" or the capital letter "O." Since the Python interpreter is case-sensitive, the incorrectness may cause the code to fail to execute or interpret incorrectly. However, this kind of typo also occurs in the traditional IDEs, so we consider such case something the user can easily solve by rewriting the character.

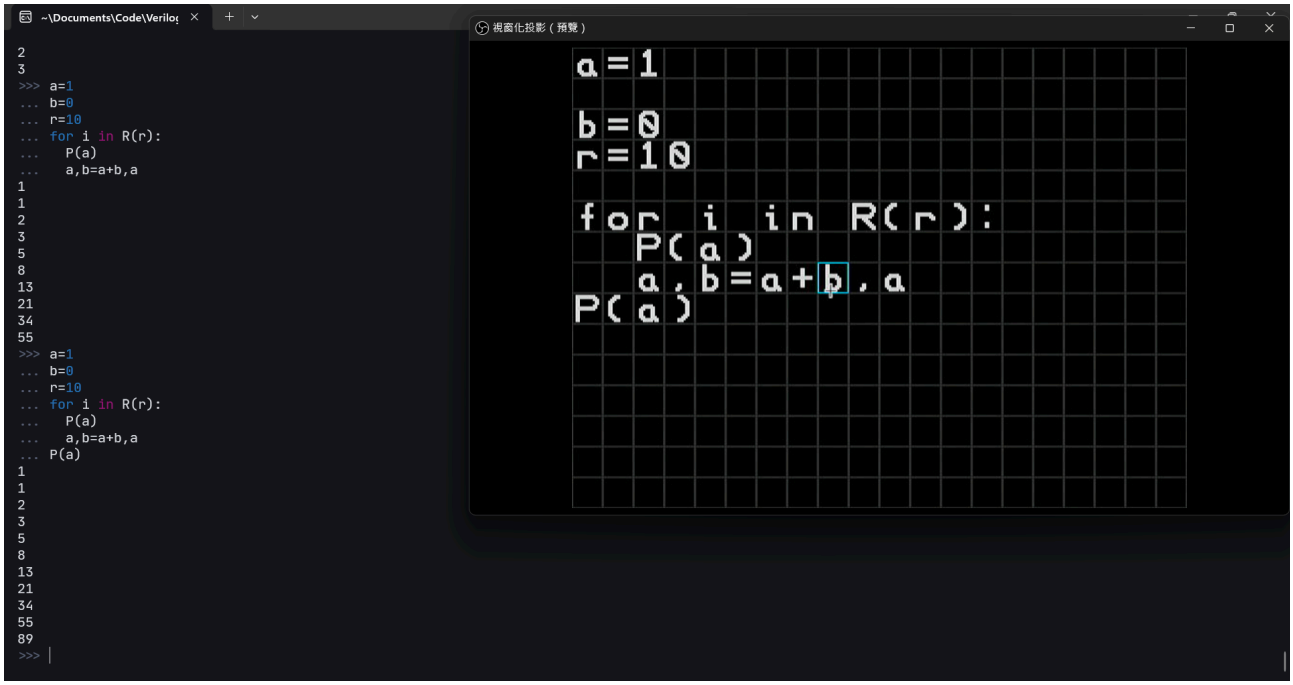

Fig 5.1 – Screenshot of project demonstration

The GitHub repository for the Verilog codes is Dogeon188/BestIDE, and the GitHub repository for the train data generation and the training of the CNN model is Dogeon188/HandwritingRecognizeUnipen.

§5.2 Resource Utilization

Fig 5.2 shows the resource usage in the whole project, respectively. "Canvas," "Document," and "Fonts" all use DRAM, which will be implemented with LUTs (Look Up Tables). We can observe that "Canvas" and "Document," which are "Dual port RAM" blocks, are implemented mainly as "LUT as Memory," and "Fonts," which is a "ROM" block, is implemented as "LUT as Logic." Block RAMs are mainly used for saving the parameters used in the recognizer. We have used about 80% of the block RAMs on the FPGA board.

In the recognizer, we only used one DSP as a multiplier since we didn't parallelize the calculations, making the whole recognizing process take about 30 milliseconds. If we want to add parallelism, the BRAMs should be separated, and more complicated addressing should be implemented. This may use up all the available BRAMs and threaten the timing constraint from the increased address calculation time. However, these can still be improved in future works.

Name	Slice LUTs (20800)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Slice (8150)	LUT as Logic (20800)	LUT as Memory (9600)	Block RAM Tile (50)	DSPs (90)	Bonded IOB (106)	BUFGCTRL (32)
top	2638	1049	229	50	841	2478	160	41	1	21	1
canvas (Canvas)	40	0	18	0	11	8	32	0	0	0	0
canvas_input_inst (canvas_input)	262	111	0	0	98	262	0	0	0	0	0
clk_wiz_0_inst (clock_divisor)	1	18	0	0	5	1	0	0	0	0	0
debounce_clear_data (debounce)	3	4	0	0	4	3	0	0	0	0	0
debounce_rst (debounce_0)	2	4	0	0	3	2	0	0	0	0	0
debounce_send (debounce_1)	1	4	0	0	1	1	0	0	0	0	0
document (Document)	145	0	64	0	46	17	128	0	0	0	0
extending_signal_rst (extending_signal)	2	3	0	0	1	2	0	0	0	0	0
fonts (Fonts)	387	0	90	29	114	387	0	0	0	0	0
messenger_inst (messenger)	40	38	0	0	18	40	0	0	0	0	0
mouse_ctrl_inst (mouse)	636	273	0	0	234	636	0	0	0	0	0
onepulse_clear_data (onepulse_2)	0	1	0	0	1	0	0	0	0	0	0
onepulse_MOUSE_RIGHT (onepulse)	1	1	0	0	2	1	0	0	0	0	0
onepulse_rst (onepulse_3)	0	1	0	0	1	0	0	0	0	0	0
onepulse_send (onepulse_4)	0	1	0	0	1	0	0	0	0	0	0
recognizer_inst (recognizer)	908	259	15	0	342	908	0	41	1	0	0
core_inst (recognizer_core)	867	240	15	0	321	867	0	41	1	0	0
conv_biases_inst (conv_biases)	76	0	15	0	24	76	0	0	0	0	0
conv_weights_inst (conv_weights)	228	4	0	0	89	228	0	12	0	0	0
dense_biases_inst (dense_biases)	83	0	0	0	27	83	0	0	0	0	0
dense_weights_inst (dense_weights	81	12	0	0	36	81	0	16.5	0	0	0
feat_buf_conv_inst (feat_buf_conv)	85	1	0	0	37	85	0	2.5	0	0	0
feat_buf_pool_inst (feat_buf_pool)	107	4	0	0	45	107	0	10	0	0	0
max_inst (max_4)	99	0	0	0	30	99	0	0	0	0	0
text_editor_inst (text_editor)	102	309	42	21	114	102	0	0	0	0	0

Fig 5.2 – Resource usage of the project

Fig 5.3 shows the timing summary for the whole project. Most of the critical paths occur in the recognizer module, and that's why the pipeline registers are introduced in §3.6. With the help of pipelining, we can suppress the critical path length under 10 nanoseconds, maintaining a positive WNS.

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.502 ns	Worst Hold Slack (WHS):	0.031 ns	Worst Pulse Width Slack (WPWS):	3.750 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	4411	Total Number of Endpoints:	4411	Total Number of Endpoints:	1251

All user specified timing constraints are met.

Fig 5.3 – Timing summary of the project

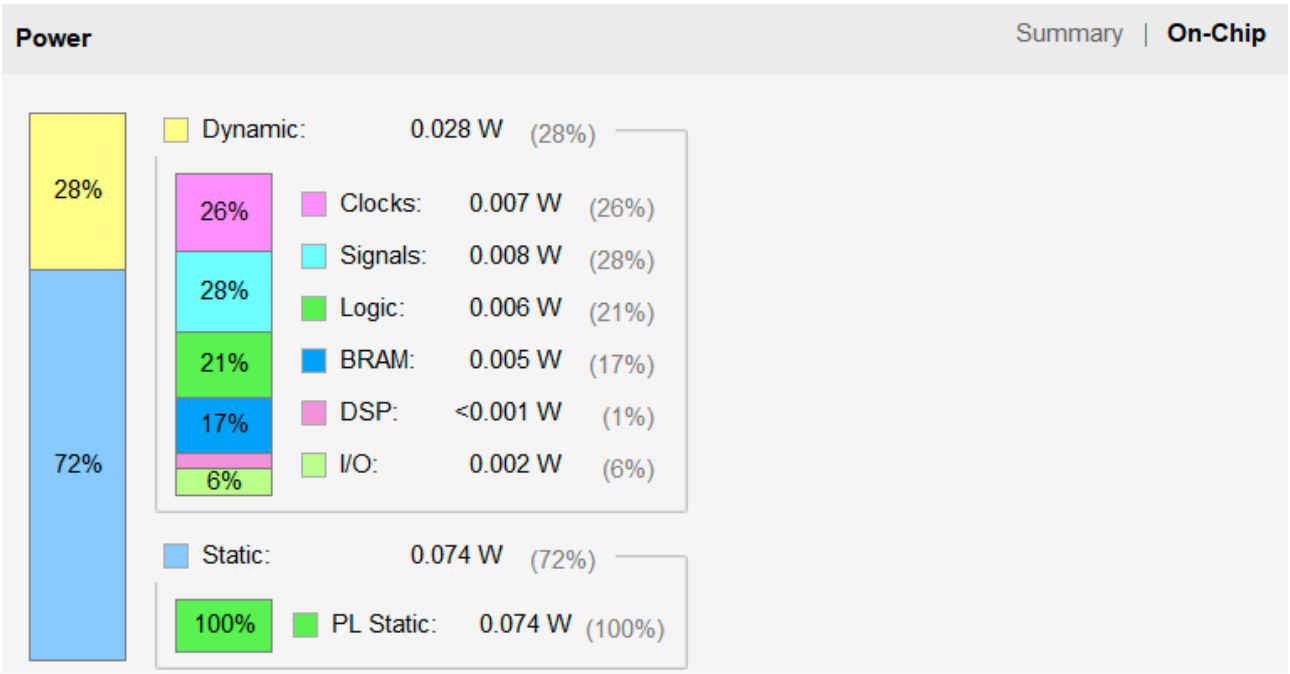

Fig 5.4 – Power analysis of the project

§5.3 Takeaways

- Timing analysis is important for a large hardware design project.
- Fixed-point representation makes it easier to implement real number arithmetics.
- DRAM can be used asynchronously, which is useful for non-delayable circuits.
- Bresenham's line algorithm is a perfect method to draw lines on grid-based displays.
- Simulating the Verilog code is easier for debugging than running on actual device.
- For loops in pseudo-codes can be implemented by FSMs in hardware designs.

§5.4 Possible Improvements

- Add a scrolling feature to increase the maximum line length to at least 32 characters.

When we write Python code, more than a maximum of 20 characters per line is required sometimes. Also, "Document" is extended to 32 characters per line to make addressing easier. If we can use the wasted space, the maximum number of characters per line can be increased. However, we can't change the size of the screen. The best solution is to use the scroll wheel on the mouse to extend the horizontal view. If we define a scroll shift as one block, most modules only need minor modifications.

- Add automatic port confirmation protocol.

It's possible to add a **handshaking protocol** between a host computer and the FPGA board via the RS-232/UART interface. With something similar to a TCP protocol, we could automatically select the correct port without having the user manually input the port name.

- Add fallback class for the recognizer.

Some common image-categorizing neural network implementations have a fallback class that contains anything that **couldn't be categorized into the intended classes**. Our 96 classes include a redundant class (95) that could have been used as a fallback class. That said, we didn't train the model with a fallback class anyway, which is a possible improvement for our project.

- Better addressing at feature map buffers.

As stated in §5.2, the calculation of recognizer is not optimized in execution time. By increasing the parallelism, lower execution time is nothing but impossible. However, the most significant bottleneck is the memory access, not the arithmetic operations. We've made a slight improvement by reading two entries from the pooling feature map buffer simultaneously. Still, it could have been better if we addressed the nearby 2x2 entries consecutively in memory.

As for the convolutional layers, we haven't thought of a better solution other than just reading one by one, but there should be a better way to address them so that multiple entries can be read at once. With the cooperation of weight storage, which reads nine entries at once, the performance could be further increased.

- Taking advantage of the two ports on BRAM.

The referenced paper mentioned in §3.2 provided a way to increase parallelism by calculating multiple outputs at once. This is a much more friendly approach compared to the addressing of feature map buffers. Furthermore, if we could separate the BRAMs into smaller chunks, more ports could be utilized, which implies higher parallelism.

§A Appendix

§A.1 Contribution

黃庭曜

- Graphics rendering and painter module.
- Bresenham's line algorithm implementation.
- Text editor module.

毛柏毅

- UART communication module.
- Python terminal server.
- Handwriting recognizer CNN model and hardware module.

§A.2 The Bresenham's Line Algorithm

Bresenham's line algorithm is a tool to draw lines on a grid-based display. Our project has sufficient clock cycles for us to update our canvas. With this advantage, we implement Bresenham's line algorithm with an FSM.

To draw lines, we would have to choose the step direction. The step direction should choose the one with larger differences to make the line look continuous. In the module, we used the signal `abs_delta_x` and `abs_delta_y` to determine the direction. When the step direction is the one with larger differences, all the lines we're drawing have a slope (m) less than or equal to 1.

$$0 < m = \frac{\min(|\Delta x|, |\Delta y|)}{\max(|\Delta x|, |\Delta y|)} \leq 1$$

Without loss of generality, let's assume $\Delta x \geq \Delta y, \Delta x > 0, \Delta y > 0$.

When we step once, the y position will be increased by m.

$$x = x_0 + 1 \quad y = y_0 + \frac{\Delta y}{\Delta x} = y_0 + m$$

Since m is less than or equal to 1, the y position will only stay at its original value or be incremented by 1. We round the y position to the nearest integer to select the closer block. Then, we can turn the math expression to the following equation.

$$x = x_0 + 1 \quad y = \begin{cases} y_0 + 1 & \text{if } m \geq 1/2 \\ y_0 & \text{if } m < 1/2 \end{cases}$$

For the second step, we can have the following math expression.

$$x = x_0 + 2 \quad y = \begin{cases} y_0 + 2 & \text{if } 2 \times m \geq \frac{3}{2} \\ y_0 + 1 & \text{if } \frac{3}{2} > 2 \times m \geq \frac{1}{2} \\ y_0 & \text{if } 2 \times m < 1/2 \end{cases}$$

The case expression goes forever; however, with observation, we can use a parameter called D to save the so-far accumulated m. In other words, D will be added with m for each cycle. The expression can be simplified to the following expression.

$$x = x_0 + 1 \quad y = \begin{cases} y_0 + 1 & \text{if } D \geq \frac{1}{2} \\ y_0 & \text{if } D < \frac{1}{2} \end{cases}, \quad x = x_0 + 2 \quad y = \begin{cases} y_0 + 2 & \text{if } D \geq \frac{3}{2} \\ y_0 + 1 & \text{if } \frac{3}{2} > D \geq \frac{1}{2} \\ y_0 & \text{if } D < 1/2 \end{cases}$$

Since the value of the slope is less than or equal to 1, if y is incremented, we can subtract D by 1. Then, we can reuse the expression on the left-hand side.

$$x_n = x_p + 1 \quad y_n = \begin{cases} y_p + 1 & \text{if } D_p \geq \frac{1}{2} \\ y_p & \text{if } D_p < \frac{1}{2} \end{cases} \quad D_n = \begin{cases} D_p + m & \text{if } D_p \geq \frac{1}{2} \\ D_p + m - 1 & \text{if } D_p < \frac{1}{2} \end{cases}$$

Note that since we implemented it as an FSM, the value of D should be initialized by the value slope.

Instead of doing floating point calculations, finding a method that only involves integers is better.

During the whole process, Δx and Δy is fixed, we can try to multiply all Ds by Δx.

$$x_n = x_p + 1 \quad y_n = \begin{cases} y_p + 1 & \text{if } D_p \geq \frac{1}{2} \times \Delta x \\ y_p & \text{if } D_p < \frac{1}{2} \times \Delta x \end{cases} \quad D_n = \begin{cases} D_p + \Delta y & \text{if } D_p \geq \frac{1}{2} \times \Delta x \\ D_p + \Delta y - \Delta x & \text{if } D_p < \frac{1}{2} \times \Delta x \end{cases}$$

Multiply D by 2 again to remove all $\frac{1}{2}$ terms.

$$x_n = x_p + 1 \quad y_n = \begin{cases} y_p + 1 & \text{if } D_p \geq \Delta x \\ y_p & \text{if } D_p < \Delta x \end{cases} \quad D_n = \begin{cases} D_p + 2 \times \Delta y & \text{if } D_p \geq \Delta x \\ D_p + 2 \times (\Delta y - \Delta x) & \text{if } D_p < \Delta x \end{cases}$$

Then, we can get the perfect expression for drawing the lines. Note that the "×2" terms can be turned to a left shift when implementing on hardware. Then, all we need for the drawing algorithm are adders, counters, comparators, multiplexers, and D-flip flops.

§A.3 References

- Byte of Michael. (2021, March 30). Why Microsoft Paint is the best IDE for programming [Video]. YouTube. https://www.youtube.com/watch?v=JKxVEuy2d6k
- J. Sueiras, Handwritting characters database, (2016), GitHub repository, https://github.com/sueiras/handwritting_characters_database
- Yan, F., Zhang, Z., Liu, Y., & Liu, J. (2022). Design of Convolutional Neural Network Processor Based on FPGA Resource Multiplexing Architecture. Sensors (Basel, Switzerland), 22(16), 5967. https://doi.org/10.3390/s22165967
- Bresenham's line algorithm, Wikipedia, https://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm