ECE532: Digital Systems Design Final Group Project Report April 7 2025

Real Time Digit Recognition over HDMI Video

Group 28:

Yiannis Cunning
Christine Lim
Muhammad Ozair
Mia Qiu

1.0 Overview

The motivation of the project stemmed from the growing relevance of embedded AI and edge computing, particularly for real-time classification tasks on resource-constrained hardware. The goal was to implement a complete end-to-end system that performs digit recognition on live HDMI input using a hardware-accelerated K-means classifier, with visual feedback provided on both an HDMI monitor and an OLED display.

Throughout the duration of the project, the main objectives were to implement a real-time handwritten digit recognition on FPGA, maintain a responsive and interactive visual user interface, and stay within hardware constraints of the Nexys Video board – particularly the BRAM.

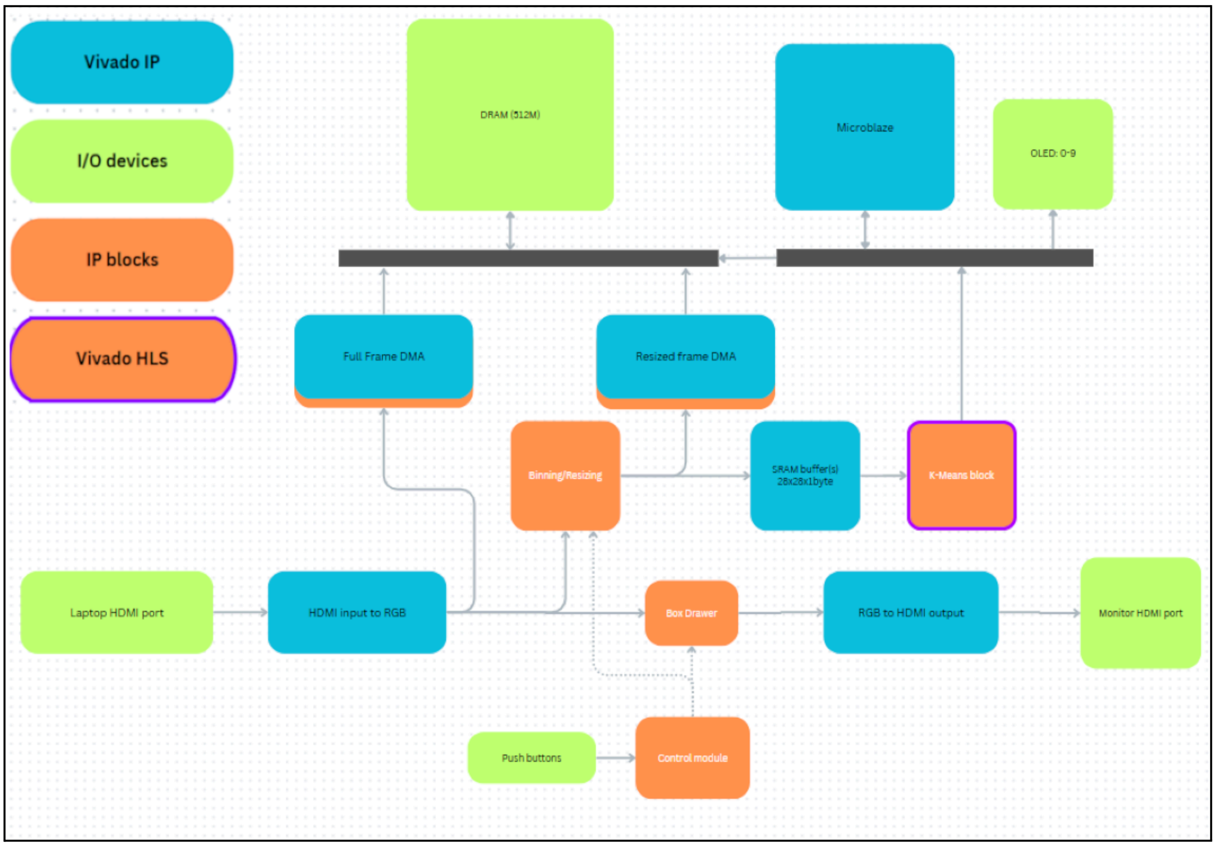


Figure 1. High-level system architecture for real-time digit recognition.

Figure 1. shows a high-level overview block diagram of the system. It first accepts a live video feed via HDMI from a laptop or other source. It then passes through the following processing stages:

1. HDMI to RGB Conversion: Converts HDMI signal to a raw RGB stream.
2. Box Drawer and Controller Module: Overlays a movable bounding box on screen and manages user inputs via push buttons.
3. Binning/Resizing Block: Crops the selected sub-image and resizes it to 28x28 using a multi-stage pipelined custom IP.
4. Resizer to K-Means Bridge: Stores the resized frame into SRAM and triggers the classifier.

5. K-Means Block: Classifies the image into one of 10 digits based on pre-trained centroids using Euclidean distance.
6. Output Displays:
    a. OLED Display: Shows the recognized digit and optionally the 28x28 image.
    b. RGB to HDMI Output: Passes the original video stream through with bounding box overlays.

The following are the IPs used and developed:

- HDMI IPs: Digilent HDMI input/output IPs for RGB conversion and passthrough.
- Custom Resizing IP: Created from scratch using Verilog to convert high-resolution sub-images to 28x28 grayscale images.
- Custom K-Means Classifier IP: Synthesized from C++ to Verilog using Vivado HLS, optimized to calculate distances to pre-trained centroids in hardware.
- Resizer to K-Means Bridge: A custom FSM that controls dataflow and synchronization between modules.
- OLED Driver: Based on Digilent's OLED PMOD IP, modified to receive input from classification and image resizing blocks.
- Software IP: Runs on MicroBlaze, used for configuring hardware blocks, user interaction, and toggling modes between software/hardware classification.

The most resource-intensive component of the system was the K-Means classification block, which required careful optimization to meet timing and BRAM constraints. Initial experiments with more complex classifiers like KNN highlighted significant limitations in BRAM and synthesis timing, reinforcing the need for a more efficient solution. Throughout the design, testbenches were used to validate functionality across modules, but verifying real-time data streams remained challenging. Synchronizing module interactions without storing full image frames required precise control logic and iterative hardware testing.

2.0 Outcome Results.

2.1 Features Comparisons

2.1.1 Waveshare video vs HDMI input

One of the initial objectives of the project was to utilize the Waveshare video module to provide input video streams to the FPGA board. However, the user manual and datasheet for the module were poorly documented, which would have significantly increased the time required for testing and peripheral integration. Additionally, the raw video stream output from the Waveshare module required a considerable amount of image pre-processing before it could be fed into the image processing IP blocks. This added layers of complexity exceeded the course scope and introduced a higher risk of not meeting the final deadline. As a result, the plan was revised to use an HDMI input source instead, allowing bounding boxes to be captured and displayed directly on a monitor.

2.1.2 Image Recognition Algorithm

Classifier type	Accuracy on	Storage	Computational Requirement

	MNIST	requirement	(per inference)
Full KNN	97.05%	45 MiB	47M MACs
Reduced KNN	81.74%	784 KiB	802K MACs
K means	82.05%	7.66 KiB	7.86K MACs

The original project plan proposed the use of a K-Nearest Neighbors (KNN) due to their high accuracy rates on the MNIST dataset. The target accuracy was set at a minimum of 90%, a benchmark that could be achieved by K-Nearest Neighbors (KNN) using relatively simple neural network frameworks. However, KNNs typically require substantial computational resources, particularly in terms of multipliers and memory for convolutional layers. Even with a reduced resource version of the KNN framework, the K-means algorithm was able to achieve a similar accuracy rate, with much lower resource requirements. For the final implementation, we opted to use the K-means algorithm as the image processing block. achieved an accuracy of approximately 80% on the training NMIST dataset.

2.1.3 Performance Monitor using UART vs OLED
Since the UART port was allocated for CPU communication, we decided to use the OLED display as a performance monitor instead of the originally planned UART interface. This also allowed for the feature of displaying the live sub-image.

2.1.4 Dataflow
For the HDMI input and output, attempting to store a 60fps signal in DRAM became impractical, as the bandwidth limitations of DDR3 memory could not meet the required throughput. To address this, we opted to stream video data directly from the HDMI input to the HDMI output. This architecture allows for higher bandwidth and thus higher resolution and framerates.

2.2 Final Performance Measurement
To evaluate the feasibility and performance gains of our design, we compared and calculated the speedup and latency of each IP block against a fully software-based implementation running on the MicroBlaze processor. As expected, all IP blocks implemented in hardware demonstrated significant acceleration, achieving up to a 99% speedup compared to the software benchmark, as summarized below.

Function	**SW Latency (on microblaze)**	**HW Latency**	**Speedup**
K-Means	15.7ms	235.4 us / 4.09 us (optimized)	99%

Update OLED with text / 28x28 image	11.9ms / 15.8ms	—	—
Resizing	10x10 input = 78.9ms 100x100 input = 1.86 s 600x600 input 56.26 s … 1280x720 = 2min 22 s	1.1 us (from after last digit input) / 16.6 ms	99%
Box control	(infinite/blocking)	(none/very small)	100%
Box drawing	8.3 ms	(none/very small)	100%

2.3 Future Improvement

One area for potential improvement in our project is enhancing the classification accuracy of the image processing IP block. One approach is to increase the number of centroids (i.e., more than one) and parallelize the distance calculations within the K-means algorithm. This would allow more refined clustering and faster computation. Additionally, applying image preprocessing techniques, such as deskewing, centering, and artifact removal, could help reduce edge noise and improve input quality. Exploring alternative distance metrics or feature extraction methods may also contribute to improved accuracy.

These types of improvements would be easily implemented as we have lots of left over FPGA resources and have a lot of extra slack time in terms of inference rate.

To further accelerate the system, implementing a ping-pong buffering mechanism could increase throughput, thereby supporting higher frame rates for HDMI input and output.

For future extensions, incorporating a more advanced algorithm could enable support for multi-character recognition and general non-digit character classification.

2.4 Alternative approach to the Original Design

If we were to start over, we would place greater emphasis on accurately estimating the resource requirements for the FPGA, including memory, processing power, and bandwidth, to ensure the system operates efficiently at scale. A more thorough analysis of the DDR3 bandwidth would help prevent limited bandwidth issues when handling high data rates. Additionally, we would establish more specific milestones and allocate dedicated time for testing and optimization. Regular checkpoints throughout the project would facilitate timely delivery, while also allowing us to make necessary adjustments in response to unforeseen challenges.

2.5 Suggestions on Next Steps

If someone were to take over the project, we would recommend a thorough review of the current design, code, and hardware setup to gain a comprehensive understanding of the project's status. As previously mentioned, there are opportunities for performance enhancement in the image recognition custom IP block. We suggest experimenting with different feature extraction methods, increasing the number of centroids in the K-means algorithm, or exploring alternative machine learning models to improve accuracy and efficiency. One possible existing architecture that may be able to fit within these requirements is the LeNet CNN.

3.0 Project Schedule

The original project milestones can be found in Appendix C. The main differences between these goals and our actual executed results are summarized below.

3.1 Comparison

Original Milestone	**Completed Milestone**
Milestone #1	
● Research Neural classifiers ● Show waveshare Vivado block diagram ● Research into the resizing algorithm	● Researched Neural classifiers ● Waveshare block diagram shown ● Average resizing algorithm suggested as baseline
Comments: All goals on schedule.	
Milestone #2	
● Show waveshare images saved to DRAM ● Implement SW neural classifier ● Show RTL diagram of resizing algorithm and start writing the Verilog	● Waveshare testbench had issues - not fully verified. ● SW neural classifier implemented with accuracy determined successfully. ● HDMI test output started on Zed board ● Preliminary Verilog code made for resizing & grayscale module ● Luminance algorithm decided for RGB to grayscale conversion in hardware
Comments: Waveshare testbench was not working fully, putting us slightly behind in this regard. HDMI output was started early as we had the bandwidth. Resizing Verilog code was made but was not functional. Software character recognition was showing good results in software.	
Milestone #3	
● Show complete simulated testbench	● HDMI input and output and frame

of resizing module ● Show a completed RTL diagram of the neural classifier ● Show a Vivado project diagram of HDMI output	capture was shown on Nexys video. ● The static resizing module and testbench was written but the simulation waveform was showing some issues. ● KNN was run with Vivado HLS, testbench of KNN block started on.
Comments: HDMI passthrough was shown, putting us ahead of schedule in this regard. Also this was where the video was switched to a streaming interface as there were DRAM bandwidth limitations. The static resizing block was written along with a testbench, however there were some issues with it. KNN was on schedule, now using HLS.	
Milestone #4	
● Show a complete verified simulation of neural classifier ● Show HDMI input and output working	● Variable resizing module completed and testbench was shown to be working ● Extra feature OLED started on ● New feature of the HW box controlling module started on. ● KNN testbench developed ● Switched to K-Means due to HW limitations, SW redone and HLS redone
Comments: Some of the I/O features were ahead of schedule allowing us to start work on new blocks such as the OLED and HW box controlling module. The Neural classifier had to be reworked as there were some HW limitations that came up. The testbench for the KMeans was being finalized. This puts us ahead of some components and behind on others.	
Milestone #5	
● Combined testbench of Resizing and inference ● Video I/O ● RTL simulation of UART output.	● OLED shown to be working ● K-means testbench showing good results ● KMeans integrated with the rest of the system ● Controlling box module integrated ● Combined testbench shown
Comments: At this point we had successfully caught up to our goals and were ahead of schedule. We also have been integrating new features we had not planned on. Verification was going well for resizing block + K-Means block + controlling module and OLED I/O.	
Milestone #6	
● Show all parts working together ● Implement stretch goal of convolutional layers if possible	● All hardware components were successfully integrated. ● Additional features were added to all of the components. ● Implement additional features to the controlling box module

Comments: In this milestone, we are ahead of schedule still and have extra time to implement new features. We did not choose to implement any sort of convolutional layers as this would not be easy to integrate with the current algorithms.

3.2 Discussion

As can be seen with the comparison of original and completed tasks, the team had many different changes to our plan. We discovered setbacks, most importantly with the waveshare camera as the input and DRAM bandwidth constraints, so the team pivoted quickly from a video streaming input approach to HDMI input and by leveraging resources such as Vivado HLS for developing the neural classifier. These changes were due to technical challenges and new ideas that were brought forward. Overall, different components each had unforeseen challenges which caused them to take different amounts of time than planned. Some components were completed far ahead of schedule and some were far behind.

By milestone 3, Hardware limitations prompted the team to switch from KNN to K-Means which required reworking both the software and HLS implementation. This shift allowed for smoother integration in later milestones.

We managed to respond to these challenges effectively and by milestone 5 had reached our minimum Viable product with all core modules integrated giving us time for improvements and additional features. By milestone 6, we were able to achieve multiple unplanned stretch goals and improve our project by having complete system integration and stability with the optional features being functional and integrated as well. It enhanced the user interaction experience such as adding more modes to the box control module.

Overall, the team managed to remain on and ahead of schedule despite early technical hurdles, showing strong adaptability and effective planning throughout the project lifecycle.

4.0 Description of the Blocks

In this section, we will go over the individual components of the design. In total the design can be split into 8 functional components. These are highlighted below on the top level diagram below. These components may each consist of a combination of packaged IP, RTL modules, imported IP and/or Vivado IP and each one implements a core function of our project.

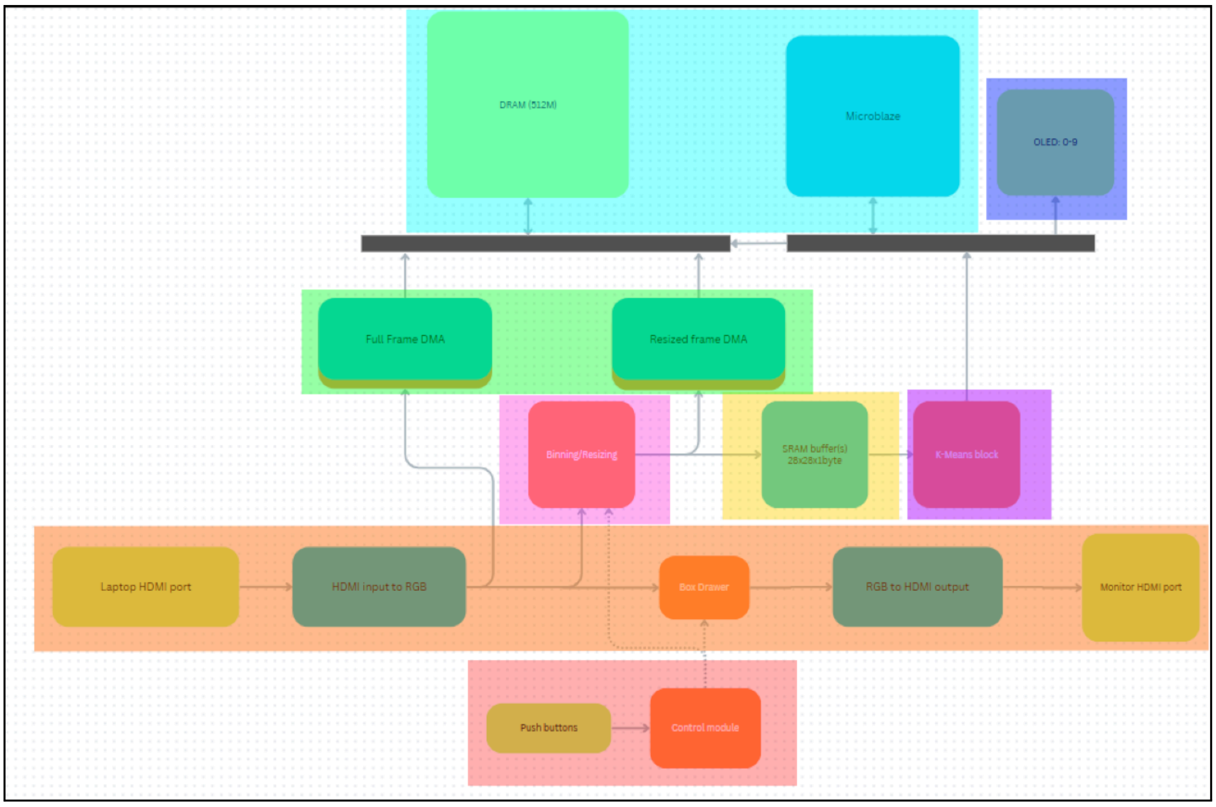


Figure 2: Separate components of the overall design.

Legend:

1. K-Means (Purple)
2. Resizing block (pink)
3. Box Controller Module (red)
4. HDMI passthrough (orange)
5. Resizer to K-Means bridge (yellow)
6. Frame DMAs (green)
7. OLED (dark blue)
8. Software (Light blue)

4.1 - K-Means Block

Source code:

- “Knn” folder (github repo [2])

Algorithm

The K-means algorithm is an unsupervised clustering method typically used to group data points based on similarity. While not originally designed for classification, we adapted K-means for digit classification by using 10 centroids to represent the digits from 0 to 9. The training of these centroids was performed using the MNIST dataset, with the algorithm clustering similar digits together. This allowed K-means to classify images by assigning them to the nearest centroid, corresponding to one of the 10 digits.

Instead of computing distances to every sample in the dataset (as in traditional KNN, which would require storing all 60 000 MNIST training images), we predefine a single centroid for each digit class. These centroids are 784 dimensional vectors (28 x 28 grayscale image pixels), representing the average digit for each class. This centroid-based version is significantly more resource-efficient for FPGA implementation, as it eliminates the need to store the entire training dataset and reduces the number of distance computations to just 10 per classification – resulting in faster inference and smaller memory usage.

The input is an image represented as a 784 element array of unsigned bytes (grayscale values from 0 to 255). It then calculates the Euclidean distance between the input image and the 10 class centroids. The output is the class label with the minimum distance to the test image.

Mathematically, for a given input image vector $\vec{x}$ and centroids $\vec{c}_0$, $\vec{c}_1$, $\vec{c}_9$, the classification result is:

$$\text{argmin} k \sum i = 1^{784} (x_i - c_{k,i})^2$$

Implementation

The classification system is implemented in C++ for synthesis with Vivado HLS. It uses a nearest-centroid classifier that computes the squared Euclidean distance between a test image and precomputed class centroids. The classifier is designed for low-latency hardware execution using HLS-specific directives to maximize pipelining and minimize initiation intervals.

The classify_image function takes a grayscale image of size IMAGE_SIZE (28×28 = 784 pixels) as input and returns the predicted class label. For each class c (0 to 9), it computes the squared Euclidean distance between the input image and the class's sole centroid (stored in centroids[c][0]). This is done by iterating through all pixels, computing the difference, squaring it, and accumulating the result into a dist variable. The smallest distance among all classes is tracked, and the corresponding class is returned as the final classification result.

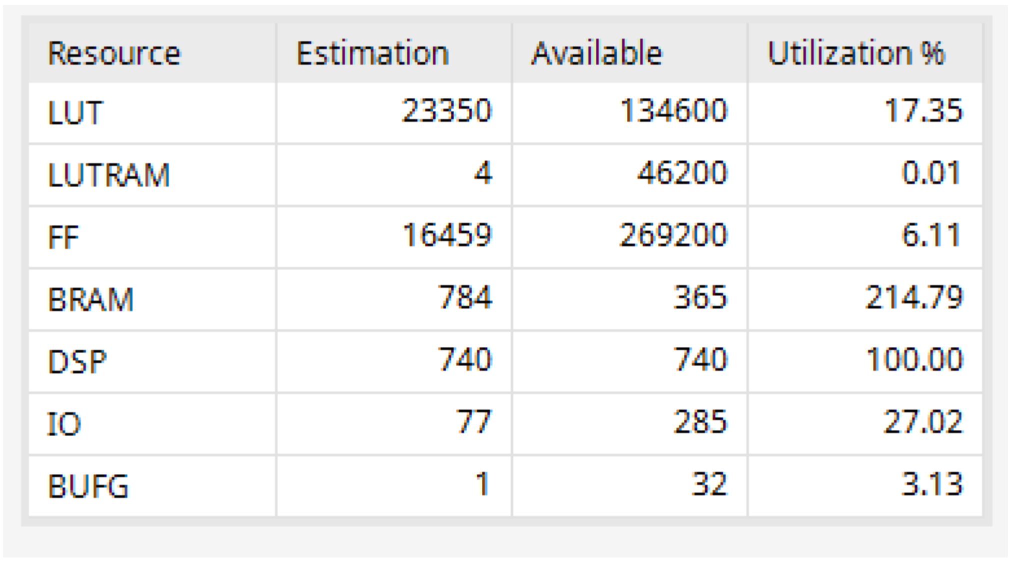
To ensure efficient hardware synthesis, loop pipelining and tripcount directives are applied:
• #pragma HLS pipeline II=1 is used to pipeline the pixel and class loops with an initiation interval of 1, allowing new iterations to start every clock cycle.
• #pragma HLS loop_tripcount provides the synthesis tool with bounds information for better performance estimation and optimization.

Finally, the knn_top function serves as the top-level HLS synthesis wrapper, making the classifier accessible for integration into an FPGA design. It simply calls classify_image and stores the predicted class in the provided result pointer.

This architecture achieves a balance between classification accuracy and hardware efficiency by avoiding full KNN comparisons with all training images and instead using a centroid-based method, which is much faster and more resource-efficient on FPGAs.

The module has been synthesized from C++ into Verilog using Vivado HLS, then packaged into an IP core using Vivado. This system meets the Vivado HLS timing constraints.

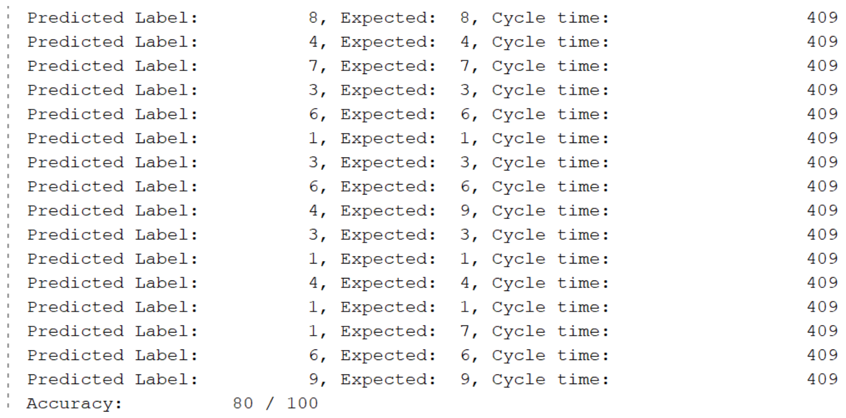
Utilization
The post implementation results for this block are shown below.



Resource	Estimation	Available	Utilization %
LUT	23350	134600	17.35
LUTRAM	4	46200	0.01
FF	16459	269200	6.11
BRAM	784	365	214.79
DSP	740	740	100.00
IO	77	285	27.02
BUFG	1	32	3.13

Figure 3: Post implementation K-Means utilization results

Testing
To verify the functionality of the K-means IP blocks, we implemented a testbench in SystemVerilog to evaluate the accuracy using the MNIST dataset. The results showed an accuracy of 80% on 200 test cases, which closely aligns with the expected accuracy observed in the software version of our algorithm, using the same trained centroids.



```
Predicted Label:          8, Expected:  8, Cycle time:              409
Predicted Label:          4, Expected:  4, Cycle time:              409
Predicted Label:          7, Expected:  7, Cycle time:              409
Predicted Label:          3, Expected:  3, Cycle time:              409
Predicted Label:          6, Expected:  6, Cycle time:              409
Predicted Label:          1, Expected:  1, Cycle time:              409
Predicted Label:          3, Expected:  3, Cycle time:              409
Predicted Label:          6, Expected:  6, Cycle time:              409
Predicted Label:          4, Expected:  9, Cycle time:              409
Predicted Label:          3, Expected:  3, Cycle time:              409
Predicted Label:          1, Expected:  1, Cycle time:              409
Predicted Label:          4, Expected:  4, Cycle time:              409
Predicted Label:          1, Expected:  1, Cycle time:              409
Predicted Label:          1, Expected:  7, Cycle time:              409
Predicted Label:          6, Expected:  6, Cycle time:              409
Predicted Label:          9, Expected:  9, Cycle time:              409
Accuracy:          80 / 100
```

Figure 4: K-Means testbench output

4.2 - Resizing block

Source code:

- “dynamic_resizing_ip” (github repo [2], Vivado project, Associated RTL files within)
- Includes: “tb_top.v”, “constant_control.v”, “hor_avg_filter.v”, “luminance_filter.v”, “normalizer.v”, “sub_image_filter.v”, “vert_avg_*.v”

Overview

The original goal of the resizing block was to convert a fixed 640x480p, 15fps video stream into a fixed 28x28 image. As the video input was changed from the waveshare camera to HDMI in, so did the resolution of the video stream. Another big change that was made since the conception of this block was the addition of a variable resizing mechanism. This was added to improve on the practical functionality of the design. It is not very useful to classify the whole screen as a single digit. Instead running it on a selected portion of the screen would allow for more flexibility.

This block takes in a 720x1280p at 60fps RGB video stream as well as a sub-image bounding box in the form of top left = (x1, y1) and bottom right = (x2, y2). This block will then produce the desired 28x28p black and white image for any given aspect ratio.

The output of this block can then be captured in a SRAM bank for later processing.

In addition to these constraints, the design of this block tried to minimize usage of FPGA resources and tried to minimize processing latency.

This block was implemented as a fully packaged custom IP with the following main interface ports

- RGB input - data, valid, hsync, vsync
- RGB output
- Bounding box - x1/y1/x2/y2
- Update box signal

An example waveform of the inputs and outputs can be seen below.

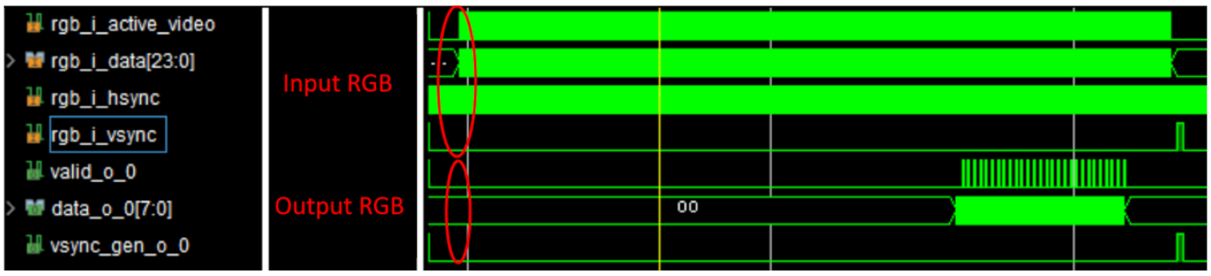


Figure 5: RGB input and output pixel example.

Next in the section we will go over the algorithm used to implement the resizing, the implementation details of the algorithm in hardware, the utilization results and finally the testing procedure used.

Algorithm - Linear weighted average

The first function of this block is to convert RGB values into black and white values, or intensity values. This can be easily done with a weighted sum of the RGB component values. The weights of the components are different as the perception of some colors is darker than others.

$$\text{Luminance} = 0.299*R + 0.587*G + 0.114*B$$

Figure 6: Luminance calculation for the resizing block.

When resampling an image there are many methods to use, we ended up with an algorithm that computes the weighted average of groups of pixels.

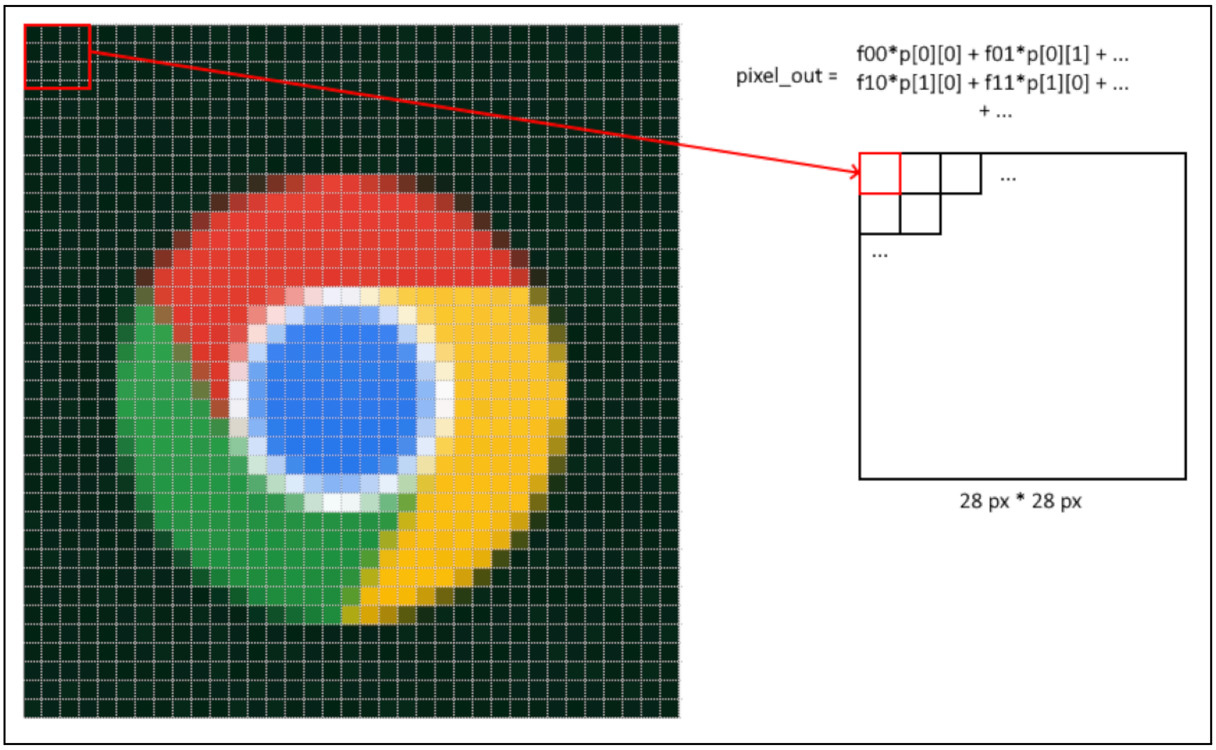

pixel_out = f00*p[0][0] + f01*p[0][1] + ...
f10*p[1][0] + f11*p[1][0] + ...
+ ...
...
...
28 px * 28 px

Figure 7: Example of downsampling through addition of weighted averages.

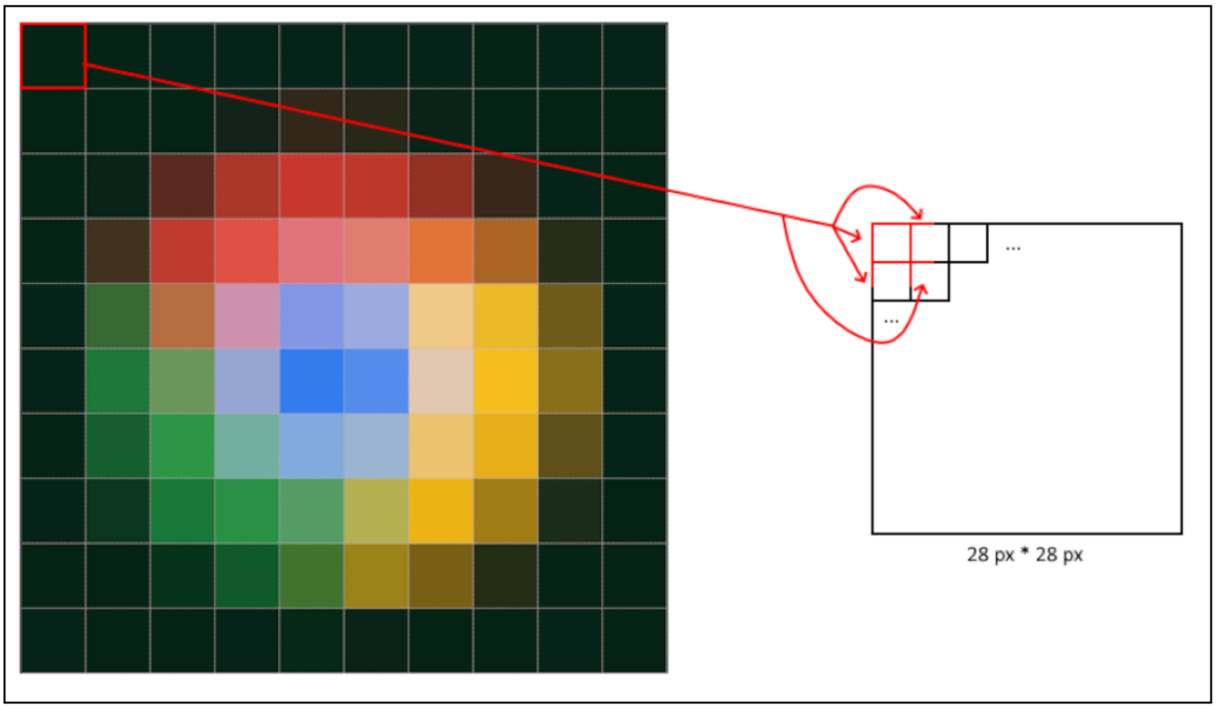


Figure 8: Example of upsampling through weighted averages.

The mathematical formulation of this algorithm can be seen below. (PV = Pixel Value). The areas are representative of a normalized coordinate system where both are fit to have the same width and height.

$$PV_{i,j,out} = \left(\sum_{input\ k,\ n} PV_{k,n,in} * \frac{Area\ input\ pixel \cap Area\ output\ pixel}{Area\ output\ pixel} \right)$$

This formulation is generalized for upsampling, downsampling, and same-size sampling.

One key implementation detail that will have to be considered is the precision of the resampling. This is because you will get decimal values when converting the coordinates of different images. If these are not handled correctly the errors can easily add up and values will lose information from rounding stages.

Implementation

In the implementation of this block, the data is processed in a streaming fashion as the input is provided in a streaming fashion. Values are kept for as little time as possible to reduce the storage/utilization overhead. For each piece of data that comes in, it provides its ‘effect’ on the output and then is discarded.

One property that is of note is that in a RGB stream, horizontal pixel values will be very close to each other while vertical values will be further away in time. This makes it difficult to produce an average pixel value over a horizontal and vertical range at the same time as you will have to store more values at once.

In order to take advantage of this property, horizontal and vertical averaging was separated. First the image will be resized to be 28x(input subimage height) then through a separate stage it will be resized to the final 28x28.

This whole process can be broken down into a 7 stage pipeline. This is shown below. (2 inter-stage FIFOs not shown)
Note that the following details have been annotated on the figure:

- Pixel input/output data type of each stage (including bit width)
- RGB image size being passed/streamed
- sub-image specific coefficients that are needed by each stage.
- Main resource requirements of each stage (in red)

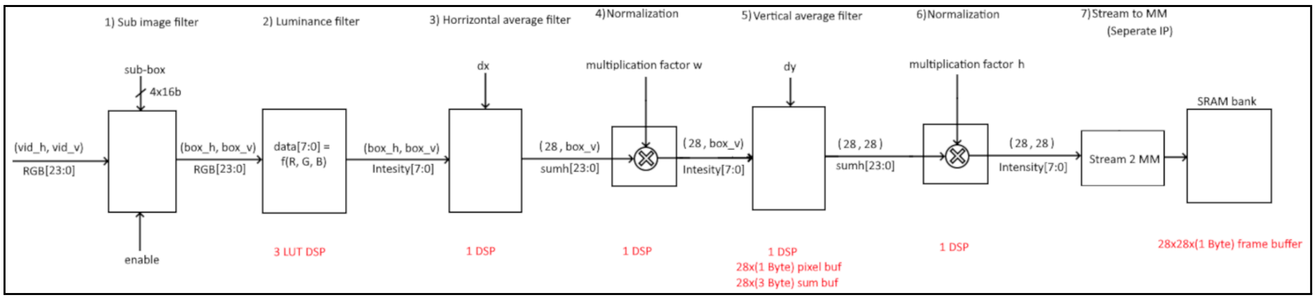


Figure 9: 7 Stage pipeline for image resizing.

The 7 Stages are listed below.

1. Sub Image filter - Only pass pixels that lie within the desired range
2. Luminance filter - Convert RGB values into 8 bit intensity values
3. Horizontal average filter - Average across the columns of the sumimage
4. Horizontal Normalization - Normalize the horizontal values
5. Vertical average filter - Average across the rows of the sumimage
6. Vertical Normalization - Normalize the vertical values
7. Convert the streamed values in writes to a SRAM block (not part of this IP)

In order to process a frame, as noted previously, there are certain constant coefficients needed for some blocks. These coefficients are functions of the input box coordinates and size.

1. dx, dy - normalized increments in destination pixel domain
    - = subimage_width/28, subimage_height/28
2. factor_w, factor_h - normalization coefficients
    - = 28/subimage_width, 28/subimage_heigh

These coefficients are calculated in parallel to the pipeline so that the pipeline does not have to stop when they are updated. When they are updated by the user, they will simply take effect at the start of the next frame. This logic is shown in the figure below.

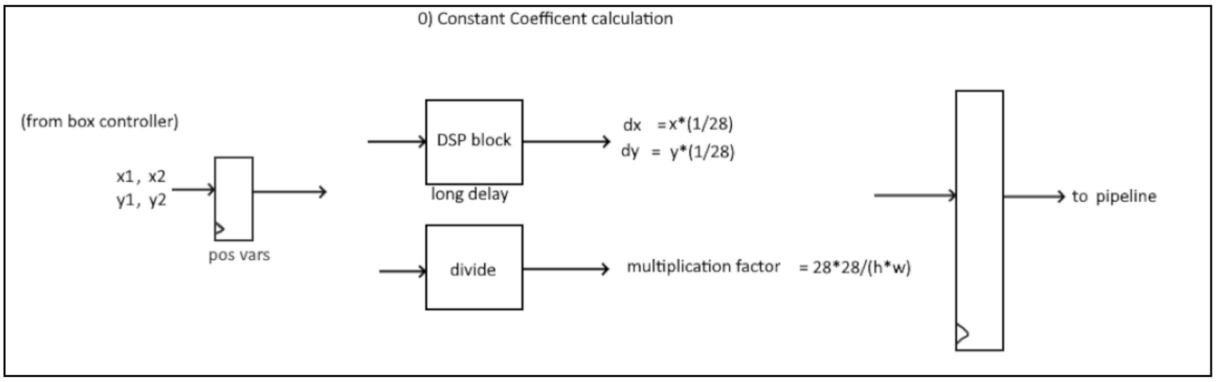


Figure 10: Constant Coefficient Calculation

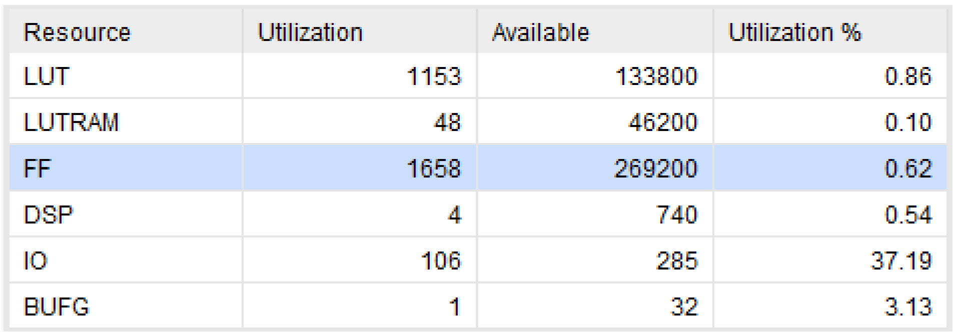
Detailed block diagrams for the horizontal average filter and vertical average filter can be found in appendix A.

Bit widths and precision of value was of large importance in this design as fixed point values are used not floating point. Fixed point values were used to make the computations easier and quicker. All rounding stages were analysed to make sure they did not lose too much information.

All buffers and FIFOs used in this design were marked as free to be optimized by Vivado in terms of the implementation to be used (BRAMs vs LUTS vs FFs). Some of the DSP blocks were chosen to be specifically implemented by FFs instead of DSP units. This was for units that were multiplying small bit widths or did multiplication with a constant. This is because for these cases the LUT implementation can be optimized to very few gates more easily making using a whole DSP unit a waste.

Utilization

The post implementation results for this block are shown below.



Resource	Utilization	Available	Utilization %
LUT	1153	133800	0.86
LUTRAM	48	46200	0.10
FF	1658	269200	0.62
DSP	4	740	0.54
IO	106	285	37.19
BUFG	1	32	3.13

Figure 11: Vivado Post-implementation Utilization results for Resizing IP.

As can be seen above, there are only 48 LUTRAMs being used and 1658 FFs. This roughly matches the estimates shown in the pipeline diagram after the 2 inter-stage FIFOs are included. These results are quite good as no full row buffer was used.

Testing

In order to ensure correctness of this block, a custom testbench was created for this IP only. This testbench reads a 720x1280p image from a memory file and generates a RGB video stream to be provided to the DUT. At the same time, it reads a test vector file with a list of x1/y1/x2/y2 coordinates to be provided to the DUT on each subsequent frame.

The output of this module was captured into a group of hex files for each sub-box input. These outputs could then be compared to the same image being resized by just a paint program. Some of the results of this testing is shown in Appendix B.

Conversion between .png files and .hex files was done with a simple python script.

4.3 - Box Controller Module

Source files:

- “box_controller” (github repo [2], Vivado project, Associated RTL files within)
- Includes: “controlling_module.v”, “controlling_module_2_mode.v”, “tb_controlling_module.v”

Overview

This IP block was developed from scratch, it was not sourced from external IP libraries. The box control manages the position and dimensions of the **dynamic bounding box overlay** which is used to select regions of interest (handwritten digits on the display). This module enables and supports real time interaction through the buttons and switches on the FPGA board which essentially allows the user to **move** or **resize** the bounding box, with a diverse set of control modes to enhance the user experience.

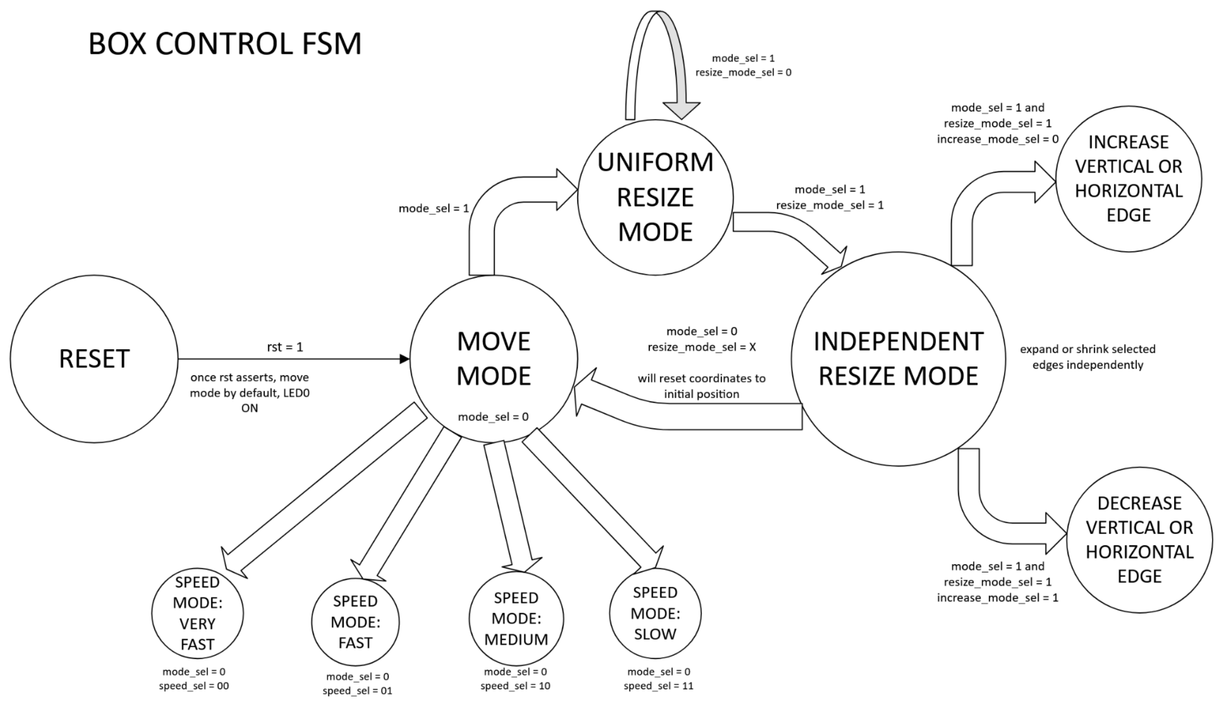


Figure 12: Box control FSM diagram.

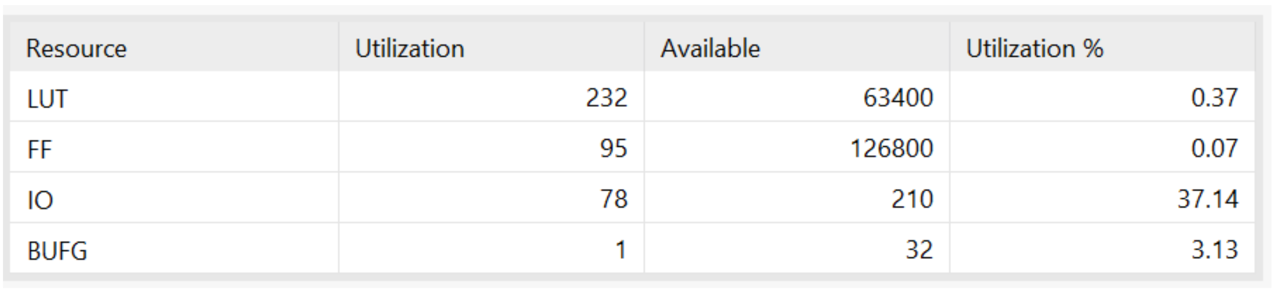
1. Movement mode: Supports all 4 box direction translations (up, down, left, right).
2. Resizing mode
    a. Uniform: Scale the box symmetrically
    b. Independent: Increase/decrease any of the edges separately
3. Speed mode: 4 movement speed settings
    a. Fast
    b. Regular
    c. Slow
    d. Super slow
4. LED outputs: Indicate which mode currently in

Algorithm & Implementation

The algorithm that was used was rising edge and time sensitive detection (delay-based movement control) functionalities to it. By using a single cycle delay register, this ensured each button press is only registered once per press and eliminated the possibility of debouncing. Each button's state was stored in a delay register, and the movement/resizing occurred when a new press was detected (button ANDed with the ~delay). A 24 bit move counter was implemented to have control over the timing. This was to ensure the movement will occur at controlled intervals, which prevents excessive updates to the bounding box and allows 4 configurable speed modes.

Utilization

There was a low usage of LUT & FF usage which confirms that the logic implemented was efficient. The usage of I/O was expected to be high due to multiple control states. Overall, the design for the box control IP was highly optimized and did not over exceed the demand of using valuable and significant FPGA resources.



Resource	Utilization	Available	Utilization %
LUT	232	63400	0.37
FF	95	126800	0.07
IO	78	210	37.14
BUFG	1	32	3.13

Figure 13: Vivado Post-implementation Utilization results for Box Controller IP.

Testing

This IP was simulated by implementing a testbench in Verilog to verify the accurateness of the movement and resizing in different selection modes. The button presses were simulated to ensure the rising edge detection algorithm that was used was working correctly. The real time delay (~50, 100, 150, 200 in ms) were emulated and observed in live testing which matched the expectation. The LED indicators were inspected to ensure they were ON/OFF in the correct mode. Most importantly, the output coordinates of the box were thoroughly inspected to ensure it was moving and adjusting accurately in all positions, and that it wasn't going out of bound (height and width limits).

4.4 - HDMI I/O and Box Drawing

Source files:

- "rgb_boxer.v" (github repo [2], hdmi_passthrw prj, RTL file)
- "rgb_splitter.v" (github repo [2], hdmi_passthrw prj, RTL file)
- "rgb_debugger.v" (github repo [2], hdmi_passthrw prj, RTL debugging file)
- "tb_dviout.v" (github repo [2], hdmi_passthrw prj, RTL testbench file)
- "DVI to RGB Video Decoder (sink) 2.0" (Digilent IP [1])
- "RGB to DVI Video Encoder (source) 1.4" (Digilent IP [1])

Overview

The main backbone of the datapath in this design is the pixel path between HDMI input and output. This is shown in the figure below.

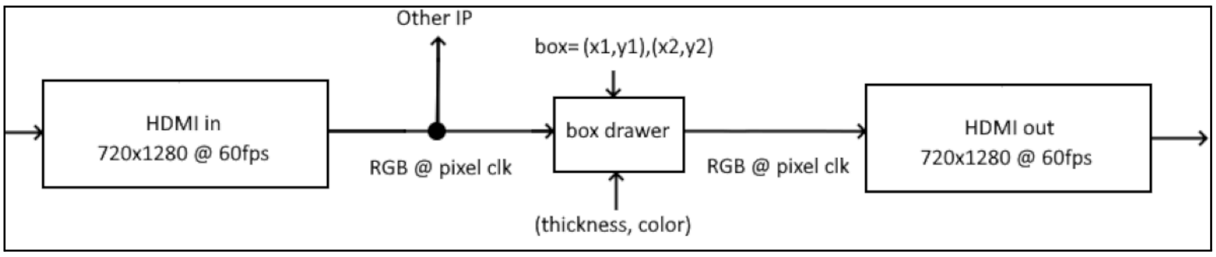


Figure 14: HDMI RGB backbone datapath.

As can be seen, the only block on this path is the box drawer and all other modules are to the side where the RGB stream is diverted in a read-only fashion.

This component is implemented so that the user can see what area of the screen they are selecting.

Algorithm

The algorithm for the box drawer is very simple, allowing it to minimally affect the data stream. This block takes in box coordinates, thickness of the box, and desired block color. It will then use the hsync and vsync signals to keep track of the coordinates of the current pixel value being passed through. If the coordinates are within the range of the box outline being drawn it will override the pixel value to have a value of ‘color’.

Implementation

The HDMI receiver and transmitter subsystems were taken from digilents IP repository [1]. This converts the HDMI data stream into a RGB video stream (pixel data, valid, hsync, vsync) and visa-versa for the output. The Box drawer was written in RTL.

Multiple overrides and resets were added that are connected to switches and to the microblaze GPIO. This made debugging easier.

Testing

In order to test that the HDMI blocks were working properly, a RGB debugging module was written in Verilog. This module simply collected statistics on the input stream such as horizontal size, vertical size and total number of pixels received. It was then easy to mark all of these as debugging nets and make sure everything about the input stream was as expected.

This made it easy to debug the input stream using an ILA without yet having an output set up to view the actual results. Also, this made it easy to verify that the pixel data was in the expected form before it was diverted into the resizing module. This helped to inform the creation of testbenches for other modules.

A simple testbench was created to test the box drawing capabilities by providing a fake input video stream. The drawing module actions could then be verified visually.

Once the HDMI output was connected, it was easy to verify that the HDMI I/O and box drawer was working as intended on the FPGA.

4.5 - Resizer to K-Means Bridge

Source files:

- “knn_bridge.v” (github repo [2], hdmi_passthrw prj, rtl file)
- “tb_knn_2.v” (github repo [2], hdmi_passthrw prj, rtl testbench file)

Overview

This helper IP was needed to bridge the gap between the two main IP blocks in this design. The output of the resizer module is in a video stream format, and the k-Means block expects to be able to read pixel values from SRAM.

This block implements stage 7 of the resizing pipeline as shown earlier.

The main features of this design is coordinating when to start and stop the K-Means block and synchronizing and converting stream data to be memory mapped. This block can be enabled/disabled by the Microblaze processor.

Implementation

This module was implemented as a FSM with 3 main states.

1. IDLE state
2. LOAD state
3. INFER state

The LOAD state makes sure the frame is aligned and writes the bytes sequentially to memory.

The INFER state triggers the K-Means block and waits until it is done. Once it is it will try and trigger the next load stage.

If the K-Means/Inference block took a longer time to infer, more time could be saved by implementing this block with a ping pong buffer to save on time.

Testing

For this testbench, an input stream was provided that mimicked the output of the resizing block. The data for this input could be provided by a hex file (converted from a png by a python script). The waveform could then be verified visually. The output bits of the K-means block should represent the recognized digit from the image.

4.6 - RGB Video to DMA

Source files:

- “binned_to_axis.v” (github repo [2], hdmi_passthrw prj, RTL file)
- “rgb_to_stream.v” (github repo [2], hdmi_passthrw prj, RTL file)
- “tb_top.v” (github repo [2], hdmi_passthrw prj, RTL testbench file)
- AXI DMA, FIFO (vivado IP)

Overview

In order to facilitate better performance monitoring and debugging, 2 extra modules were added to this design. One was tied to the input HDMI stream (allowing access to full

720x1280p frames by software) and one was tied to the output RGB stream of the resizing block (allowing access to 28x28p resampled frames by software).

The software could then decide dynamically how much of the processing pipeline it wants to implement.

In order to implement this block, two key features needed to be implemented. First is allowing for software control on when to save a frame via a GPIO signal/AXI access. Next is synchronization of the RGB stream and conversion into an AXI stream.

Implementation

This was implemented in both cases with a Verilog module that handled synchronization and conversion to AXIS. This was then fed into a Vivado IP FIFO to cross clock domains from the pixel clock to the CPU clock. A regular Vivado DMA controller can then be used to write the frames to the DRAM.

4.7 - OLED

Source files:

- Pmod_OLEDrgb_v1.0 (Digilent IP [2])

Overview

The purpose of this block is to be able to provide real time results in a visual way to the user. Additionally, this can provide feedback to the user on what the resampled image looks like. The live video stream coming out of the resizing block is copied to the OLED as a bitmap. Also, the detected digit output of the k-Means block is copied to the OLED. In the ideal case this output would be updated at the full 60fps but this is only possible in some software modes. Additionally, the output of this screen may be 1 frame off of the current HDMI screen. Of course both of these imperfections are undetectable by a human user.

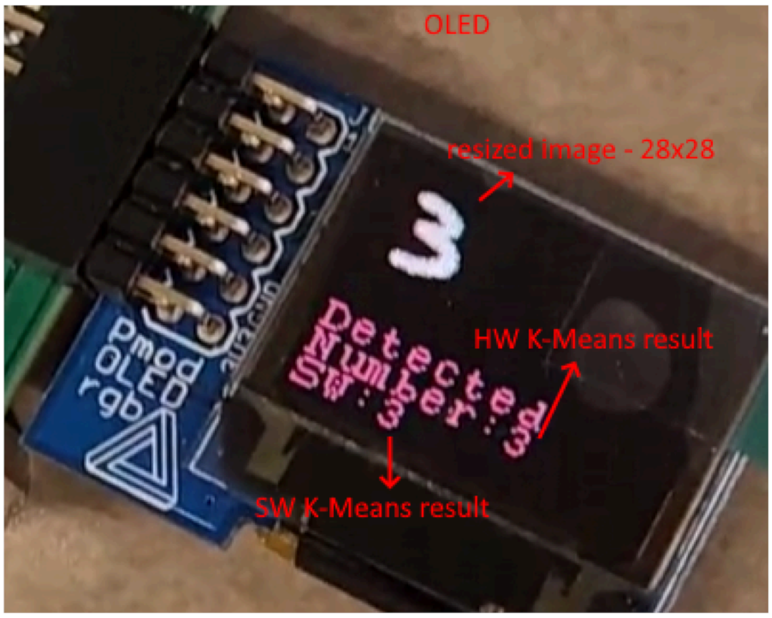


Figure 15: Output of the OLED screen.

Implementation

The OLED is mainly controlled by the diligent IP for this PMOD [1]. The Software can use the drivers for this IP to push images and text to the screen. The driver does these updates through the AXI bus. In order to collect the resized frame, one of the DMAs mentioned before is activated with a GPIO signal. Next it is copied over to a separate OLED frame buffer and changed into the correct format. Notably, the resized frame is in a 8-bit intensity format and the OLED uses 12-bit RGB values. In order to make this conversion the 4 MSBs of the intensity values are copied to be the R, G, and B values of the output buffer. This results in some loss of information but is mostly unnoticeable in most use cases. The K-Means results are simply polled through a GPIO bus and updated to the OLED as a ASCII character.

4.8 - Software

Source files:

- Microblaze soft processor (Vivado IP)

Overview

The software of this design acted as the controller module to decide what blocks are active. The K-Means algorithm was additionally implemented in software. Also the image resizing was implemented in software. Thus through software configuration, the following modes can be enabled.

1. SW resize, SW classification, and output to OLED
    - 1.62 fps (only at low sized sub images)
2. HW resize, SW classification, and output to OLED
    - 19.81 fps
3. HW resize, HW classification, and output to OLED
    - 29.72 fps
4. HW resize, HW classification, only value pushed to OLED (60fps)
    - Full 60fps

By testing each of these modes we can easily see the effect of our hardware acceleration. Also, by using the timer module, we were able to profile the algorithms used on the microblaze and provide the speedup results shown earlier.

Implementation

The software utilizes the drivers for each of the Viviado IPs used including the following.

- 2x AXI DMA interface drivers
- AXI timer driver
- Digilent OLED driver

Additionally there are many different AXI GPIO signals that can be used to configure the custom IP blocks.
Otherwise, the software generally just acts in an endless loop of capturing results and displaying them on the OLED screen. This process is shown abstractly below.

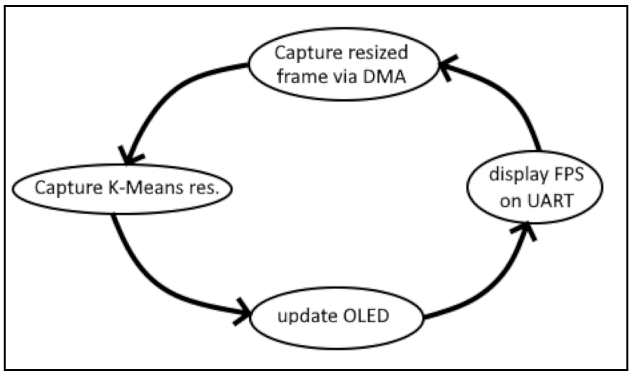


Figure 16: Software control loop

5.0 Description of Your Design Tree

The github repository for all project files can be found at this link - https://github.com/yiannis-cunning/ECE532_Project_KMeans.

In the repository there are four main folders.

1. Documentation
    a. Final report document
    b. Final demo presentation PDF
    c. Assorted figures/diagrams
2. Hardware
    a. hdmi_passthrw Vivado project (top level Vivado project)
        ■ Main project
    b. dynamic_resizing_ip + repo
        ■ Resizer IP and source code
    c. k-Means
        ■ K-Means IP and source code
    d. Box controller
        ■ Box control IP and source code
3. Scripts
    a. Png <-> hex file conversions
4. Software
    a. Main c file (helloworld.c)
    b. Knn code c file (knn.c)

Please see the “Description of Blocks” section of this document to find relation between source files and design components. Source RTL files can be found under hardware->”Vivado Project”->.srcs.

6.0 Tips and Tricks

One tip that is suggested for this course when working with the Nexys FPGAs is to overutilize ILAs. It is very easy to mark almost all nets as debug nets and set the sample depth to very deep. As long as you have the excess resources available in your design this can be an easy and effective way to debug your design after implementation.

7.0 Video

Please see the below linked youtube video for a demonstration of the final testbench.
https://youtu.be/L9q9B49reb8

References

[1] Github, https://github.com/Digilent/vivado-library/tree/master
[2] Github, https://github.com/yiannis-cunning/ECE532_Project_KMeans
[3] Youtube demo, https://youtu.be/L9q9B49reb8

Appendix A

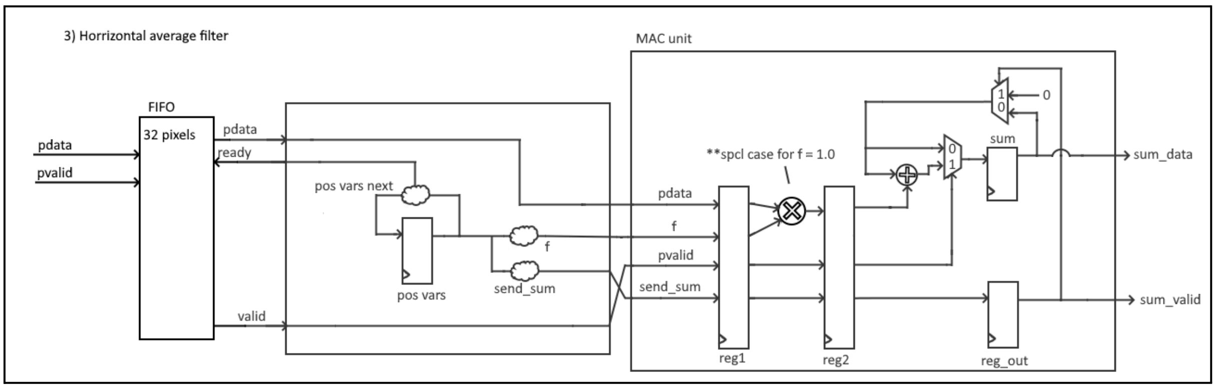


Figure 17: Horizontal average filter detailed block diagram for stage 3 of the resizing pipeline.

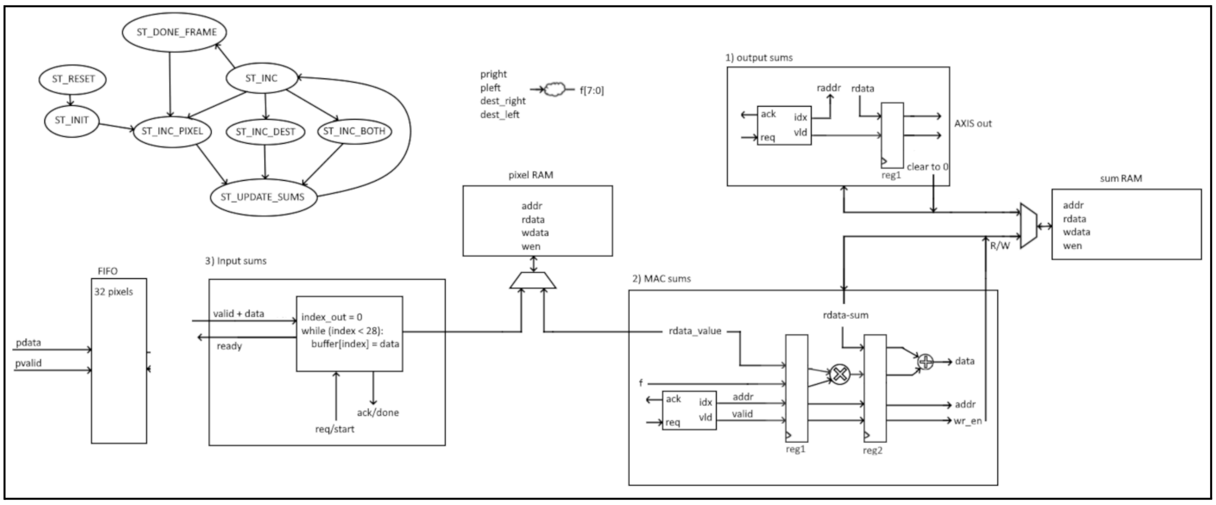


Figure 18: Vertical average filter detailed block diagram for stage 5 of the resizing pipeline.

Appendix B - Testbench Results For Resizing Block

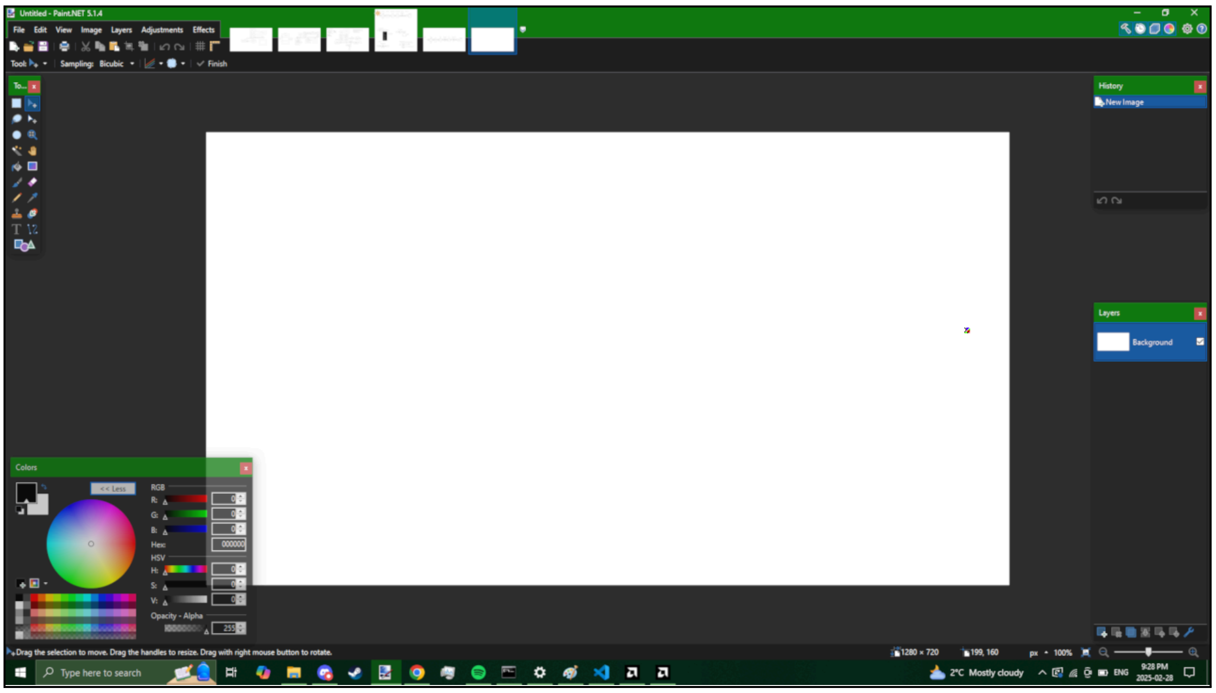
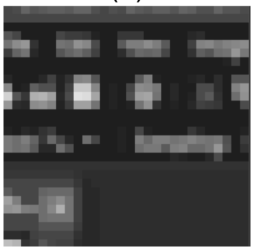
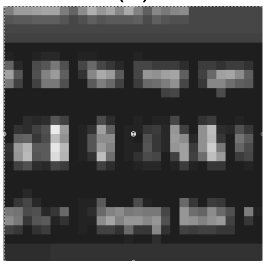
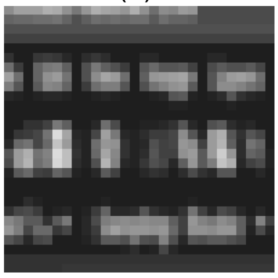
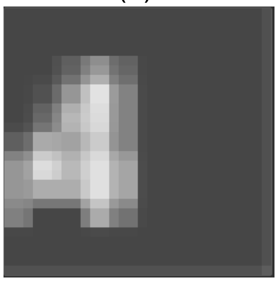
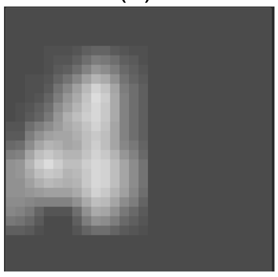
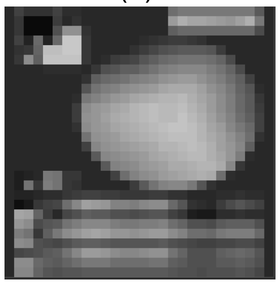
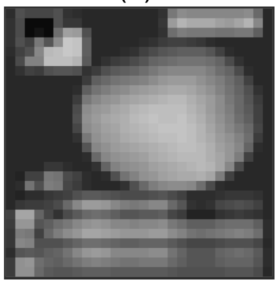

Untitled - Paint.NET 5.1.4
File Edit View Image Layers Adjustments Effects
Sampling: Bicubic
Finish
History
New Image
Layers
Background
Colors
Drag the selection to move. Drag the handles to resize. Drag with right mouse button to rotate.
1280 × 720
Type here to search
2°C Mostly cloudy

Figure 19: Input image (720x1280)

Testbench Output	Paint.net Output	Testbench Output	Paint.net Output
(0) 	(0)	(1) 	(1) 
(2) 	(2) 	(3) 	(3) 
(4)	(4)	(5)	(5)

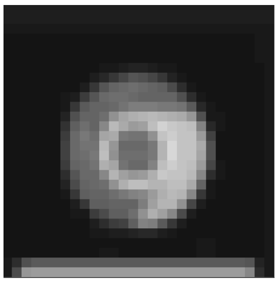
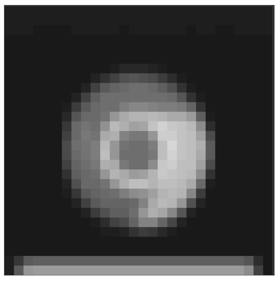
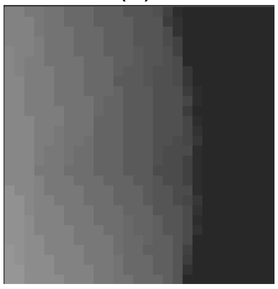
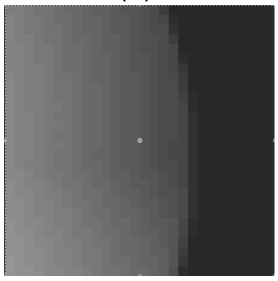
			
(6) 	(6) 		

Table: Output of Resizing model testbench vs ideal paint.net resizing.

Appendix C - Original Project Milestones 1-6

Milestone #1

- Research **Neural Classifier** to be used for feature (2.a) - KNN, SVM.
  - This should be feasibly implemented in RTL.
  - This may need SRAM storage for the weights. This should not be excessively large.
  - This should be pre-trainable.
- Show a **Vivado block diagram of (1)**.
  - Add in IP parts
- Show **what binning algorithm** we are going to use. Linear? Bi-cubic? Max value?
  - Quick overview of the re-sampling method

Milestone #2

- Show testbench with feature (1) working. **Saving camera feed to DRAM**.
  - Prove images are saved to DRAM.
- Implement the **Neural classifier in python**. Show working with 28x28 Black/white input.
  - Implement without python libraries if possible - show each calculation going on.
  - Get a pre-trained model or train our own.
- Show a complete RTL - gate level - diagram of **binning algorithm.**
  - How can a streaming interface be used to cumulatively produce the output?
  - What is the estimated storage requirement?

Milestone #3

- Show **complete simulated testbench** with the feature (2.b) working. Binning/re-sampling input image.
    - Test bench should act as the DMA controller - providing image data in the form of successive RGB pixels.
    - Output should be visually reviewed to prove binning is effective.
- Show a complete RTL - gate level - **diagram of the Neural classifier**.
    - How is input data read and pipelined?
    - Can this meet the timing requirement of 15 inferences per second?
- Show Vivado block diagram for (3). HDMI output.
    - Show IP blocks used in a file.

Milestone #4

- Show a **complete verified simulation** of the neural classifier.
    - Use the same test cases used with the python implementation.
    - Use the python implementation to debug if necessary.
- Show a **FPGA testbench with feature (3) working** - HDMI output.
    - Show a blue screen/red screen/white screen.

Milestone #5

- Show a **combined simulation** of Binning/resizing and inference.
- Show a **testbench with microblaze generated data and show an inference happen.**
    - Microblaze should create the frame data, start the ML hardware and wait for the result.
- Show a **testbench with Video in going to video output**.
    - This should pass through the camera signal.
- Show **RTL code and simulation for UART controller.**
    - Show successful printing to the UART.

Milestone #6

- **Show all parts working together.**
- Implement parallel **Convolutional layers** if possible.