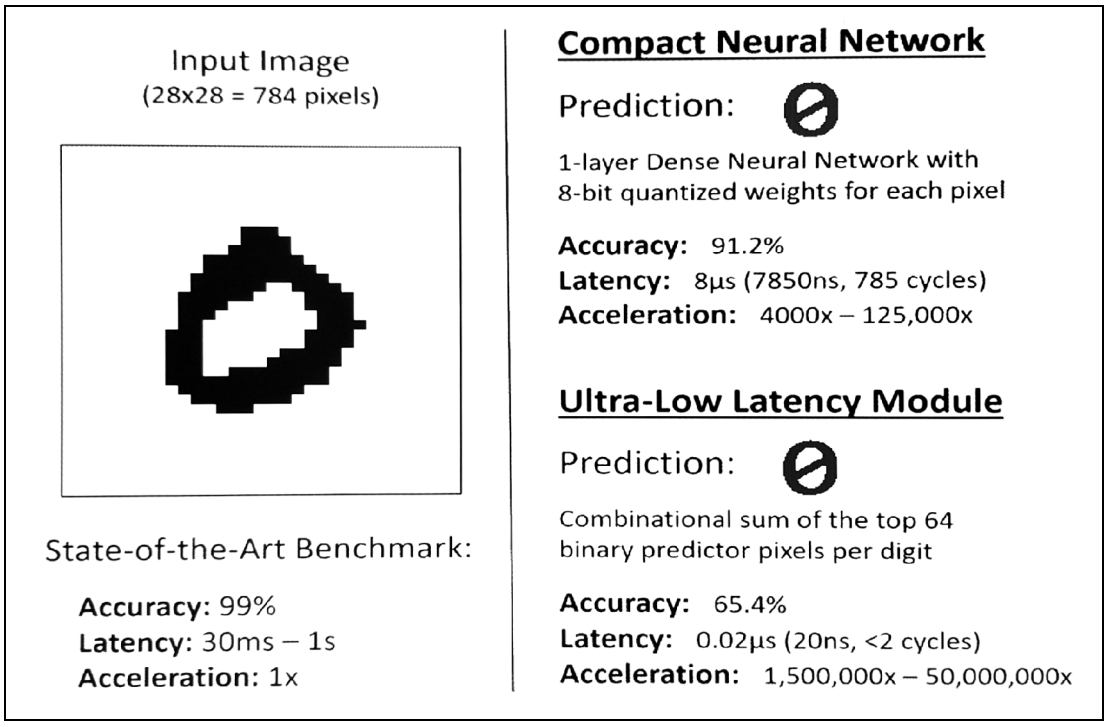
FPGA-based Artificial Intelligence:

A Low-Latency FPGA MNIST Digit Classifier



Syamantak Payra

6.111 Fall 2020

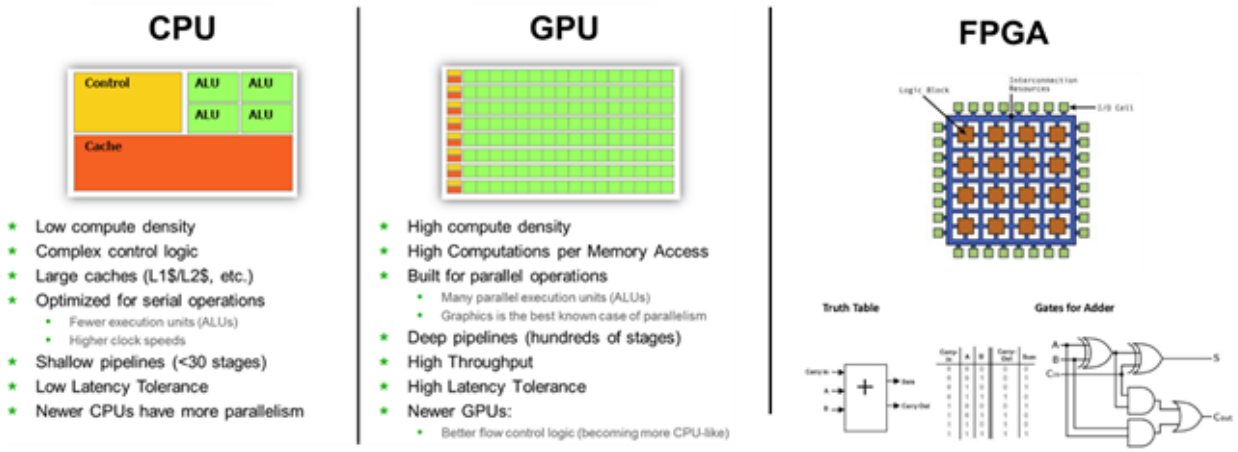
Table of Contents

1 Abstract

While GPUs are still the predominant computing architecture leveraged for machine learning / artificial intelligence (ML/AI), recent research and development efforts have explored the use of FPGAs as hardware accelerators for specific elements of neural network construction and evaluation. However, this acceleration is enabled merely by the more efficient multiply/add calculations and lower memory overhead possible on FPGAs: the rest of the ML model architecture remains the same. I propose the development and implementation of task-specific binarized neural networks to embed machine learning models into predetermined sequences of logic gates, as a mechanism to enable near-zero-latency computation using very little power for a wide variety of post-training neural network models in field applications. Through cyclic dimension-compression, node weights in conventional neural network representations can be iteratively converted into sequences of operations corresponding to n-bit lookup tables and logic within the FPGA silicon. As this minimizes the amount of arithmetic that must be performed (such as adding and multiplication), far fewer clock cycles are required to compute the results of a machine learning model compared to conventional calculations. Multiple layers can be implemented either within individual FPGA chips, or as series FPGAs that receive and process the output data from the chip acting as the previous 'layer'. Convolutions can similarly be accommodated as connections within FPGA modules. As models are 'frozen' into the hardware, instruction-based computation can be eliminated, memory transfers can be minimized, and signal propagation through sequential binary logic gates becomes the only contributor to latency of model computation. This new protocol for embedding neural networks into FPGAs has the potential to unlock low-power, low-latency computation of neural networks in a variety of edge computing applications, including minimalistic neural compute architectures and system-on-chip designs as well as one-dimensional neuromorphic computation pipelines that mimic biological neuronal pathways, facilitating rapid throughput and transformation of data and enabling instantaneous digital decision-making capabilities.

2 Introduction / Background

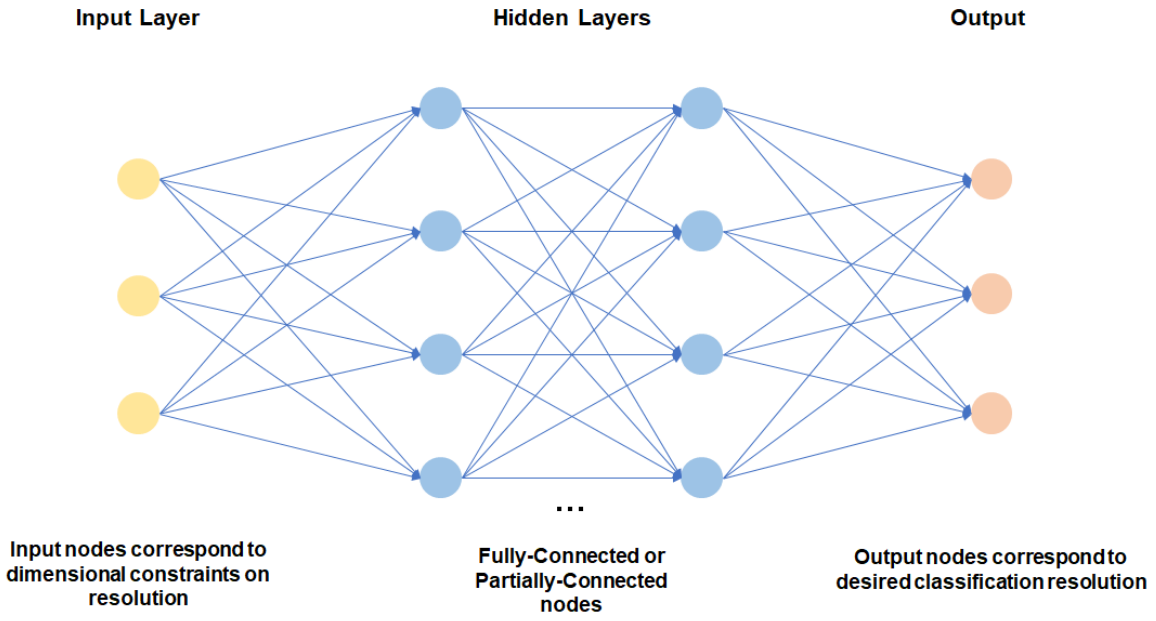
In recent years, artificial intelligence and the use of machine learning have progressively become more ubiquitous throughout everyday technologies. As a result, this has necessitated large-scale shifts in computer architecture utilization for both training and running machine learning models. Graphics processing units (GPUs) are still the predominant computing architecture leveraged for machine learning / artificial intelligence (ML/AI) model training, and tensor core units are now being utilized for cloud-based machine-learning inference models. While these platforms have successfully enabled large-scale high-performance machine learning, a main limitation of transferring neural networks to edge-computing infrastructures is the computation-intensive and memory-intensive nature of conventional models. On end-user (client) devices, model evaluation is often conducted on conventional central processing units (CPUs) or dedicated neural compute units, such as tensor cores. However, low-power platforms such as wearables and Internet of Things (IoT) devices often have low-speed CPUs, small memory capacity, or power constraints that prohibit most forms of intensive computation.



Comparison: Central Processing Units (CPUs) vs. Graphics Processing Units (GPUs) vs. Field-Programmable Gate Arrays (FPGAs). (source)

Some research and development efforts in past years have explored the use of FPGAs as hardware accelerators for specific elements of neural network construction

and evaluation. FPGAs, or field-programmable gate arrays, are specialized microchips with logic blocks that can be dynamically reconfigured to adjust to different computational tasks. While the majority of this research has been focused on utilizing FPGAs to accelerate high-performance, high-load machine learning computation, FPGAs have nonetheless shown significant capacity to decrease power consumption, latency, and computational power requirements. In order to create a viable system architecture for machine learning computation on low-power platforms, this paper demonstrates the utilization of neuromorphic logic circuitry designed on FPGAs in conjunction with binary-flattened representations of neural networks, as a mechanism to enable near-zero-latency computation using very little power for a wide variety of post-training neural network models in field applications.

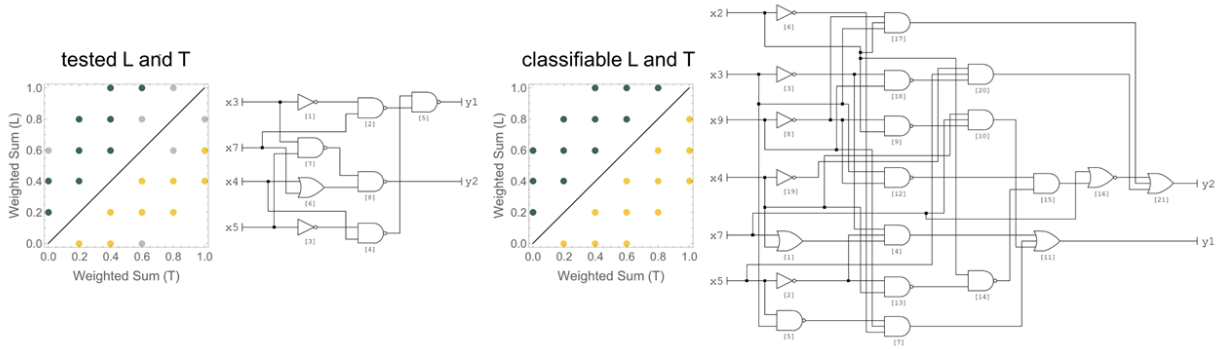


Neural Network: Nodes and layers in a conventional architecture

In this characteristic diagram of a generic convolutional neural network, each arrow represents a set of weights entering an activation function. Through cyclic dimension-compression, node weights in conventional neural network representations can be iteratively converted into n-dimension logic operations corresponding to n-bit lookup tables and binary logic gates within the FPGA silicon. By compressing

conventional arithmetic operations into binary logic, this minimizes the amount of calculations that must be performed (such as adding and multiplication).

This can be enabled through a variety of pruning and quantization strategies [Han, 2016], including: (i) Dimension-reduction then binarization (top-down approach), (ii) expansion and binarization (sideways approach), (iii) recursive binarization from layers to model (inside-out approach), and (iv) recursive binarization from model to layers (outside-in approach). As a result, significantly fewer clock cycles are required [Brelet, 1999] to compute the results of a machine learning model on a minimal binary representation hosted on FPGA silicon.



Binarized Neural Networks: weight reproduction as sequences of logic gates. [Cherry & Qian, 2018]

Uniquely, the complexity of such representation-flattening algorithms have been modulated to achieve different degrees of model resolution or to accommodate multidimensional input data [Li, 2016]. An increase in classification accuracy is achieved with an increase in model complexity; in this case, by increasing the number of logic gates devoted to a classification task. This allows higher-resolution input acceptance and greater differentiation / classification capability, noise rejection, and task accuracy. As models are 'frozen' into the hardware, instruction-based computation is eliminated, memory transfers can be minimized, and signal propagation through sequential binary logic gates is the only contributor to latency of model computation.

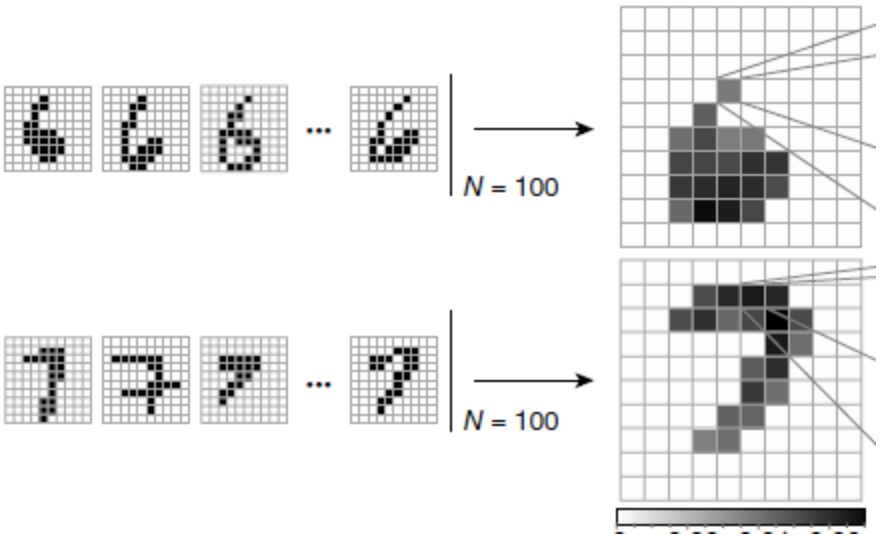
Through a series of silicon-based digital I/O channels, raw data of finite resolution can be input into the FPGA for computation and release. The protocol allows multiple layers to be implemented either within individual FPGA chips, or as series FPGAs that receive and process the output data from the chip acting as the previous 'layer'. In this way, software-abstracted computation is minimized by delegating functionalities to purely hardware-based implementations. Convolutions can similarly be accommodated as connections within individual FPGAs and between discrete FPGA chips. Since the time-complexity of an algorithm that is implemented completely in sequential logic blocks can be represented by $O(1) = L_i \sum_{i=0}^{n}(g_i)$ where g is a single logic block, n is the number of logic blocks within an FPGA and L is the latency of a single gate [He, 2014], this hardware and software protocol collective enables neural network model computation to be flattened from O(n^2) time to O(1) time, a significant improvement to both latency and computability.

3 Approach / Process

Conventional computation of neural networks involves arithmetic at each stage of the neural network. This work aims to demonstrate low-latency digit classification on an FPGA in two different ways: first, with a "binarized" network (binary logic gates producing a direct determination), and second, with an accelerated "conventional" network, with computed neural layers embedded into the FPGA fabric.

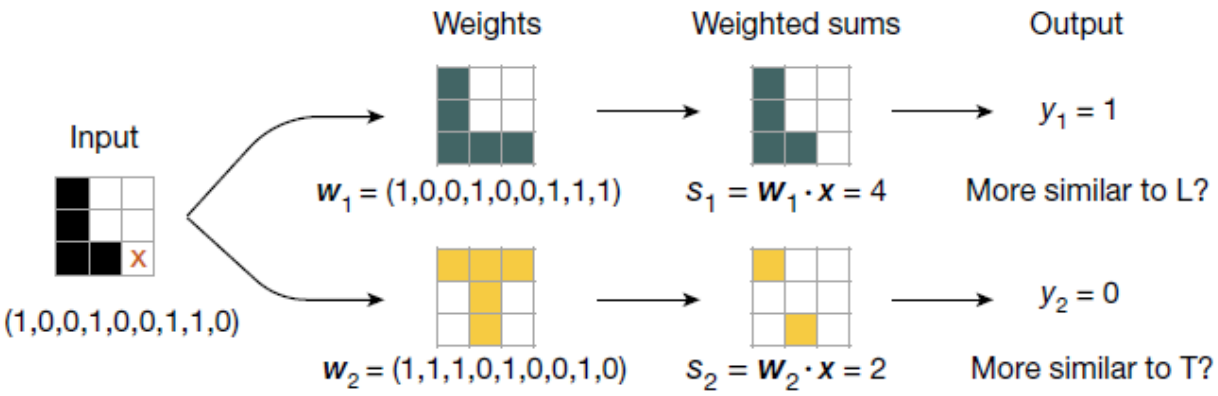
The algorithms will be tested on the MNIST dataset, a canonical machine learning dataset of 60,000 handwritten numbers. The dataset, which contains 28x28 8-bit (0-255) grayscale images, has been pre-processed to scale and centralize the digits within the area of the image. Since 1999, when LeCun et al. created the MNIST dataset from US Postal Service handwritten zip codes, there have been many networks that achieve high levels of accuracy on the dataset. However, the goal of this work is to explore the performance envelope possible if neural networks are embedded into FPGA fabric. Specifically, the tradeoff between accuracy and speed will be compared between implementations with varying degrees of complexity and resource-intensiveness.

From literature, we know that MNIST pixel weights can be determined by digit by overlaying a training set of images and calculating the aggregate proportion of pixel overlap. This principle can be utilized for neural networks of varying complexities, whether single-layer or convolutional.



Heatmap: Condensed (9x9) MNIST images

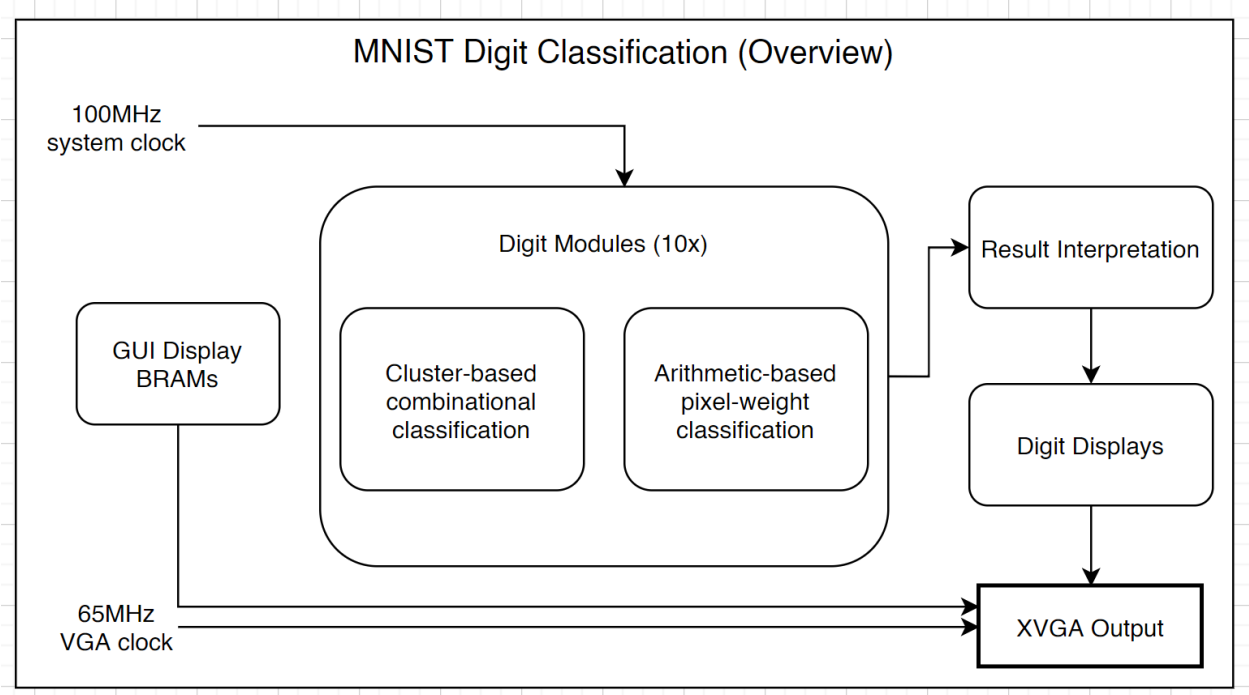
We can also observe that pixel values can be utilized in simple cases to directly differentiate between inputs. For example, in this image, a rudimentary "L" and T" can be differentiated based on only 4 logic gates (first column + col 2 row 3). Although additional logic gates will be necessary to differentiate between more digits and higher-resolution images, this strategy can be utilized for a "binary" representation where iterative computation of the values in each layer is not necessary.



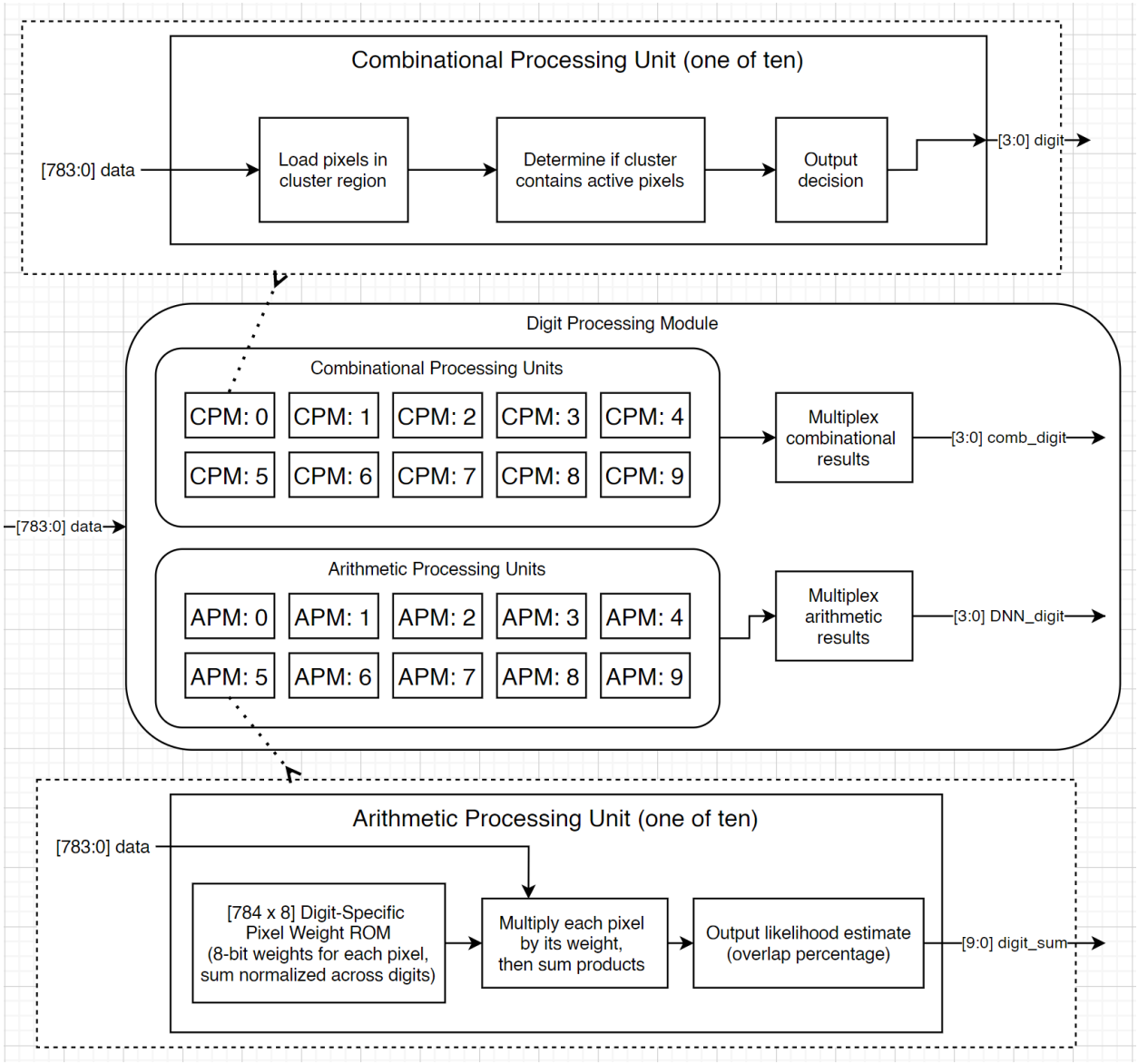
Binarized Neural Network: Differentiating between 'L' and 'T' [Cherry & Qian, 2018]

In this system, images will be transferred to the Nexys4 DDR FPGA, where digit modules will utilize a flattened and computational representation to create independent predictions of the input digit. Based on this, the display module will show the predicted

digits on a VGA output GUI, along with the latency that was required for each determination.



System Diagram: High-Level Overview

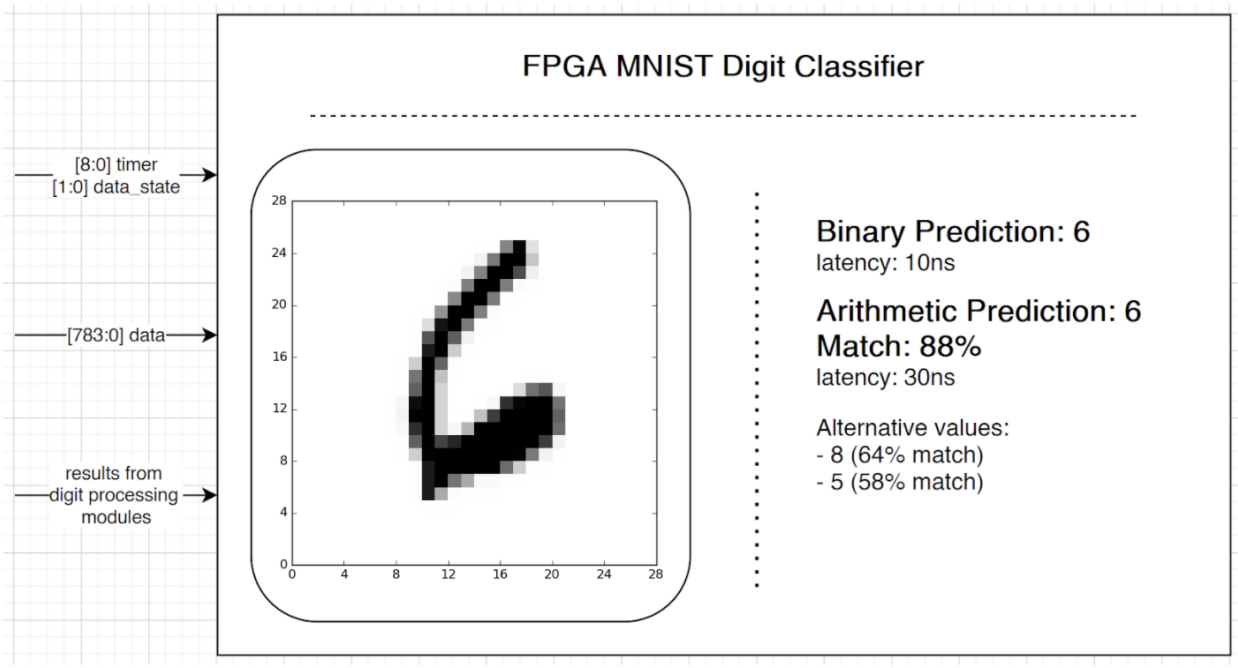


System Diagram: Digit-Processing Module

Any 28x28 pixel (784 binary pixels) test image is sent through two pathways. In the top pathway, the pixel inputs are decomposed and entered into combinational logic that can determine a digit without clock-restriction. In the bottom pathway, a sum of pixel weights is calculated for each digit, and then used to compare between digit modules and select a final digit value.

The display module collects the binary decision, as well as the probability matches from the sequential network logic, and compiles it together with the image

input into a display readout that can give insight into the underlying mechanism. In this sample screen, the input image on the left is successfully detected as a '6', along with alternative options and relative match likelihood. In addition, the classification latency is compared between the binary network and the computed network.

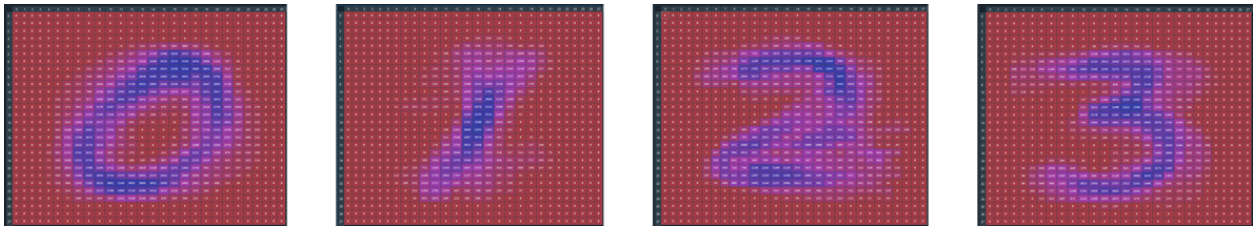


Sample GUI: Digit Classifier System

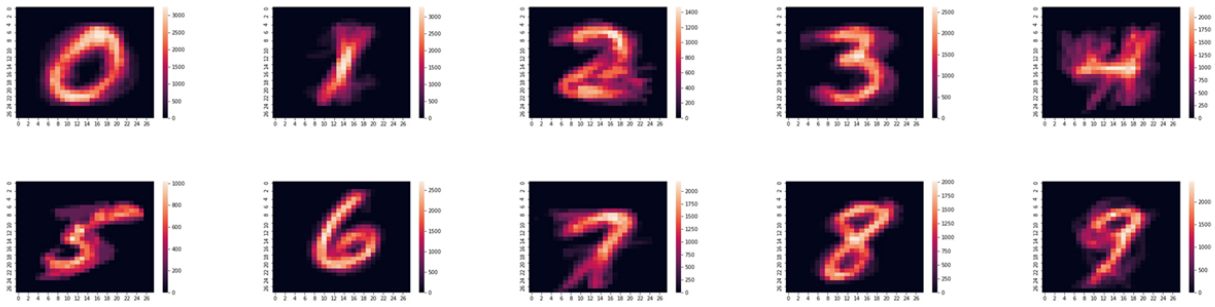
4 Combinational Module / Binarized Neural Network

To create the combinational module / binarized neural network, I took the path of recreating most direct representation of the data, which is the observed frequencies of pixels in the original data. To start with, I created a Python script to download the MNIST data (MNISTconverter.py), and a script to convert the data into heatmaps (DigitHeatmaps.py). As I worked, I added Python helper functions that I needed into this second file. These functions can be found throughout the code, which has been added to the Appendix section.

As visible in these heatmaps, even small samples of 100 images show clear overlap areas. However, it is possible to see significant amounts of variance between samples in small datasets. For example, there are some vertical and slanted 1s, there is a lot of variation in the lower bar of 2s, and 4s and 5s are somewhat obscured in the different ways they can be drawn.

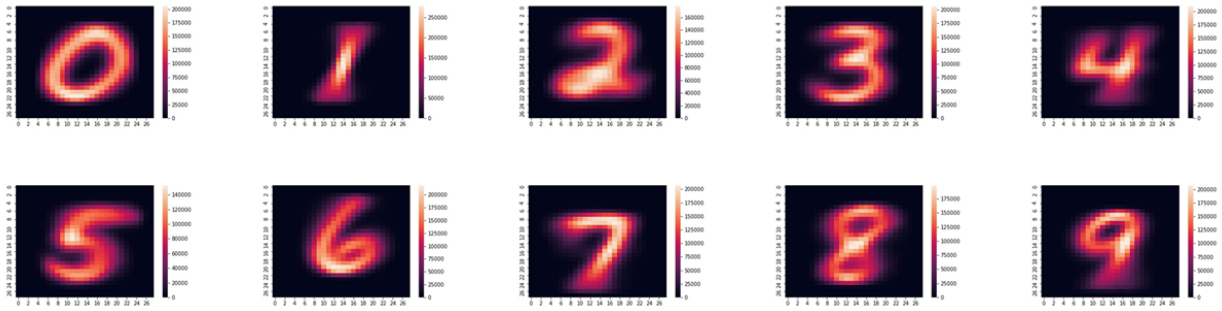


Heatmap: 100 training images

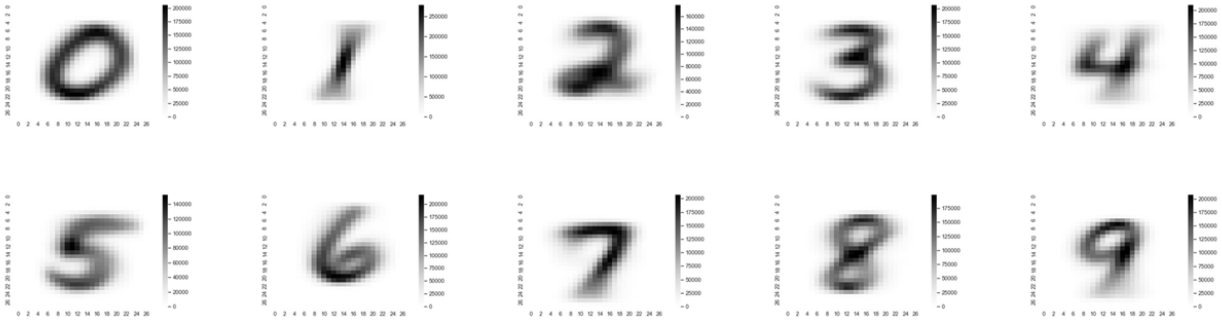


Heatmap: 100 training images

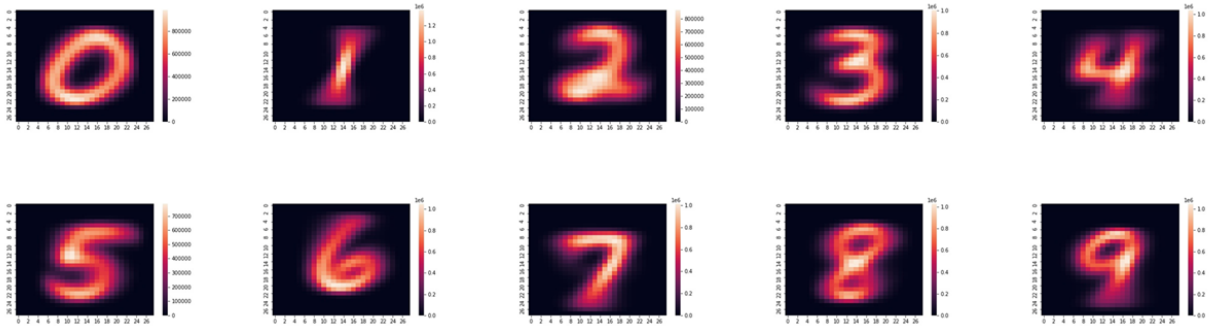
As we increase the number of images displayed, we can see those artifacts smoothed out, in these heatmaps with 10,000 and then 50,000 training images. In the grayscale images, we can start to get a better understanding of how we may be able to utilize these images to create binarized representations: from the 'strongest' pixels in the diagram.



Heatmap: 10,000 training images



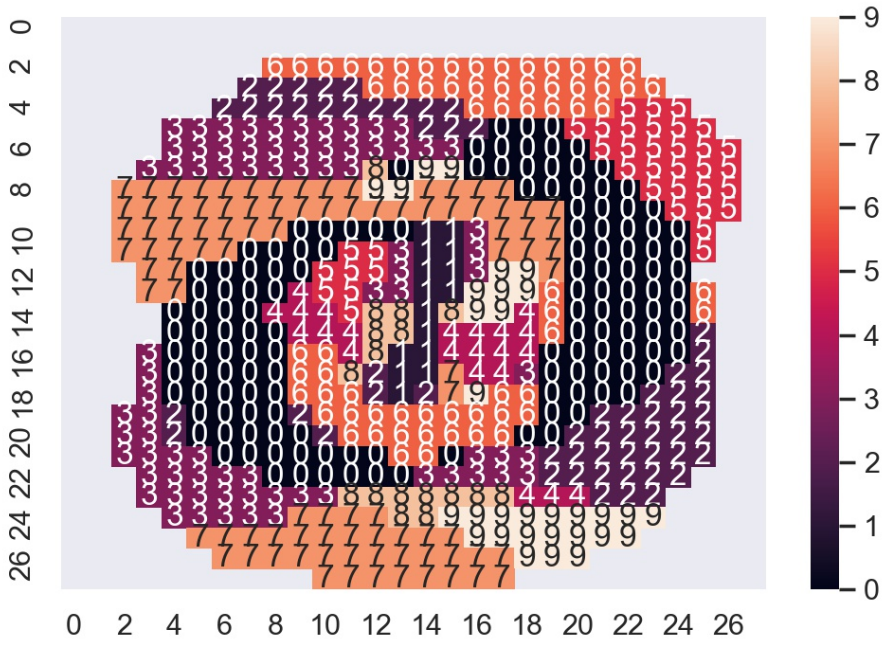
Heatmap (grayscale): 10,000 training images



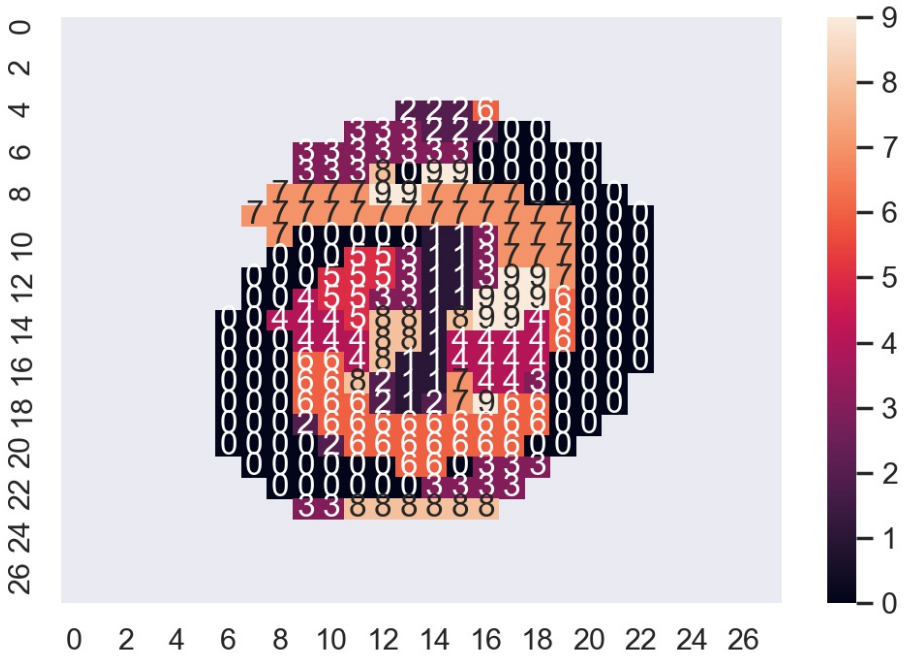
Heatmap: 50,000 training images

Heatmap (grayscale): 50,000 training images

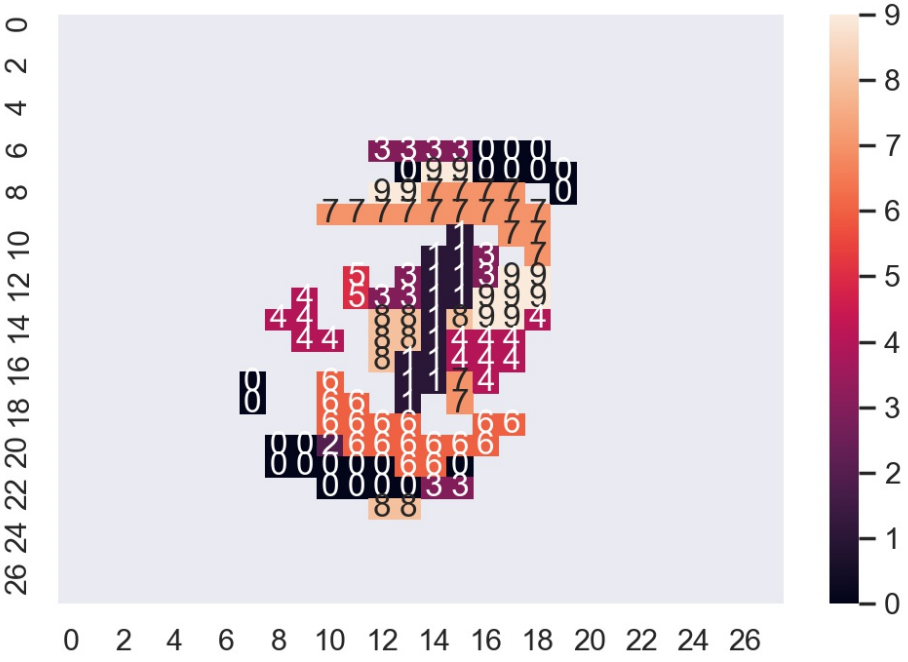
Based on the idea that the 'strongest' pixels can be used to positively identify specific digits, I created pixel maps that showed the digit with the highest value at that pixel location. Following are some plots showing the highest-valued digit. In the first plot, you can see that some digits are over-represented. For example, '4' has far fewer active pixels than 0 or 7. To try to combat this, I started thresholding the values, to reduce the number of total pixels available on the map.



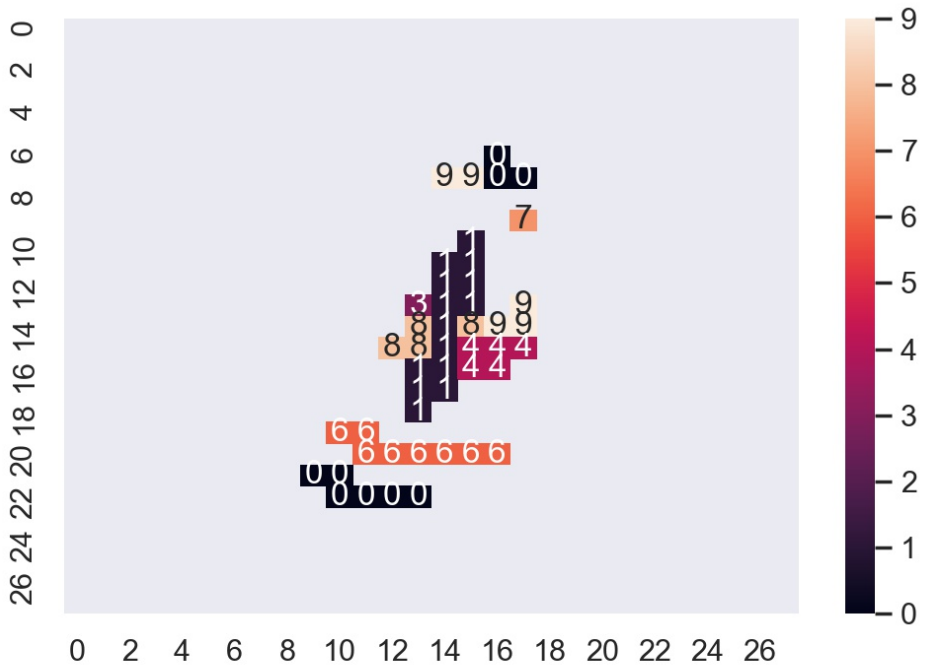
Pixel map: highest-valued digit per pixel (0 threshold)



Pixel map: highest-valued digit per pixel (100 threshold)

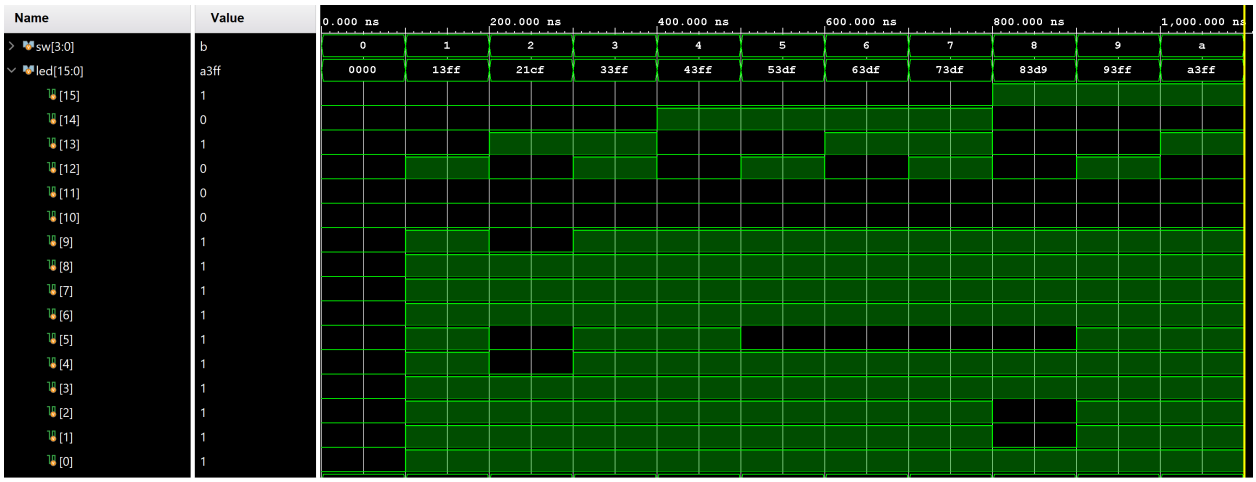


Pixel map: highest-valued digit per pixel (170 threshold)



Pixel map: highest-valued digit per pixel (190 threshold)

However, this thresholding strategy does not effectively create an accurate classification system. Even using a threshold of 100 (before the pixel dropout becomes too high to successfully visually distinguish digits, as in the 170 and 190 thresholds), an additional problem was that this combinational module indicated purely whether or not the highest-value-pixels were present in a test image; as a result, multiple



Simulation results: Pixel map module output

Even by selecting only the digit with the largest number of activated pixels, the inherent issues with this method caused the the classification accuracy to only reach a maximum of 0.28, which is quite low.

```
test_binarymaxes(4999)
0.2806561312262453
```

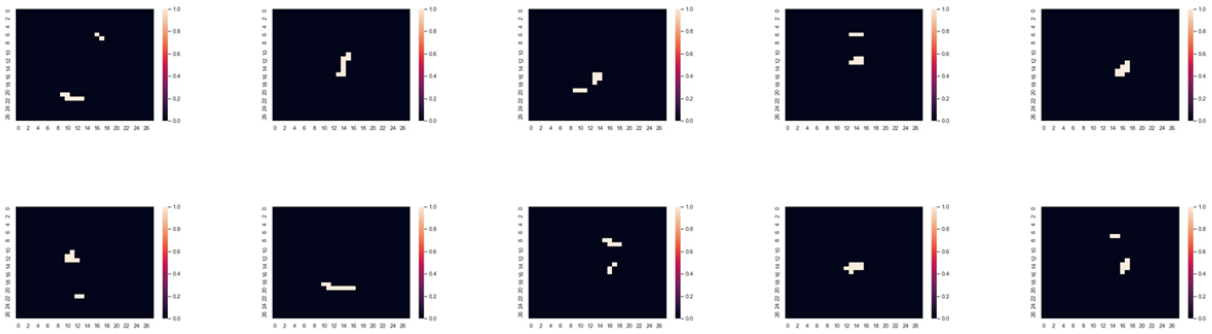
Model results: Digit pixel map maximum-value-pixels method

Instead, I went back to the heatmaps to try to figure out an alternative method. This time, I kept the principle of 'thresholding' to try to understand what phenomena could be isolated if low-occurrence (outlier) pixels could be removed from the equation and essentially ignored.

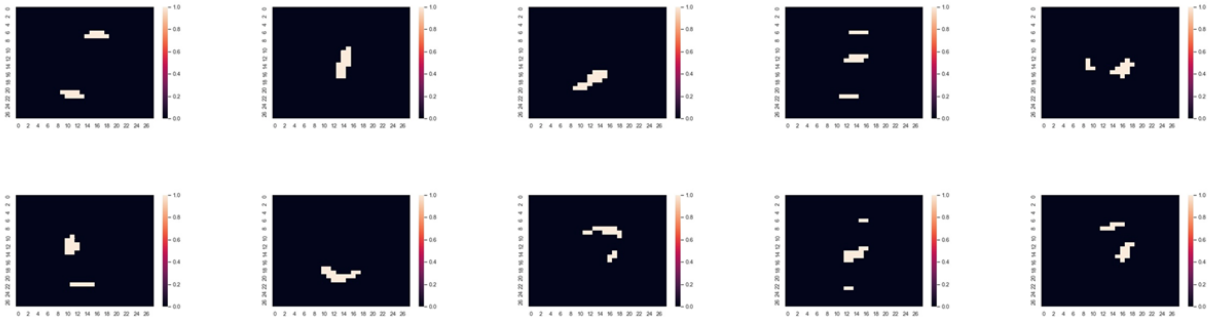
Heatmap (grayscale): 50,000 training images, pixels with value >128 (/255)

This time, I took the approach of identifying the top N pixels per digit, irrespective of how their absolute value compared with the pixel-values of other digits at that location. This was effectively a cross-digit decomposition of the previous heatmaps, with more visually representative information.

As expected, low values of N do not yield much visually identifiable information:

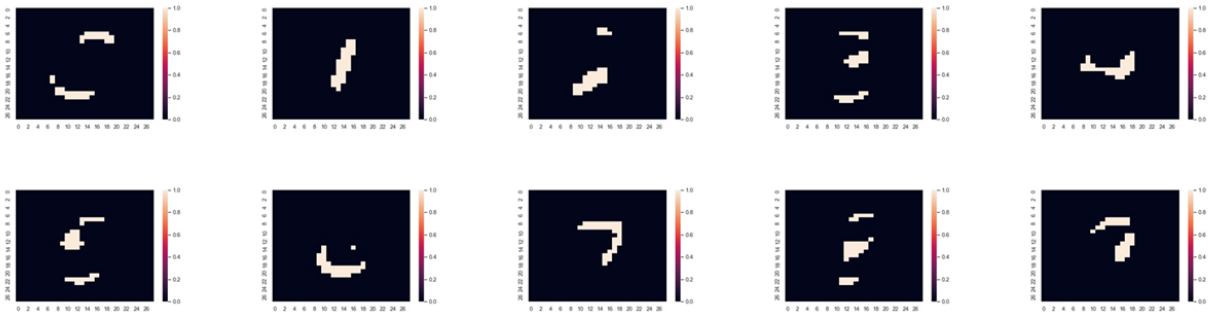


Binary reference: top 8 pixels per digit

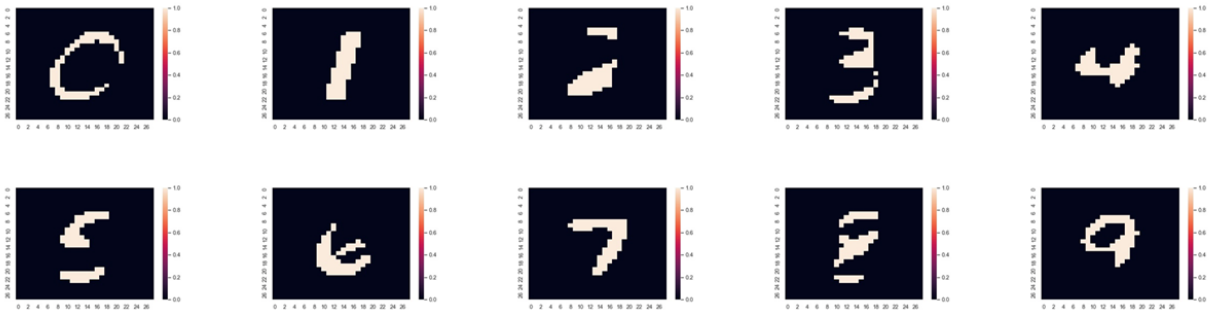


Binary reference: top 16 pixels per digit

As we get towards the top 32 and top 64 pixels, we start to be able to identify individual digits.

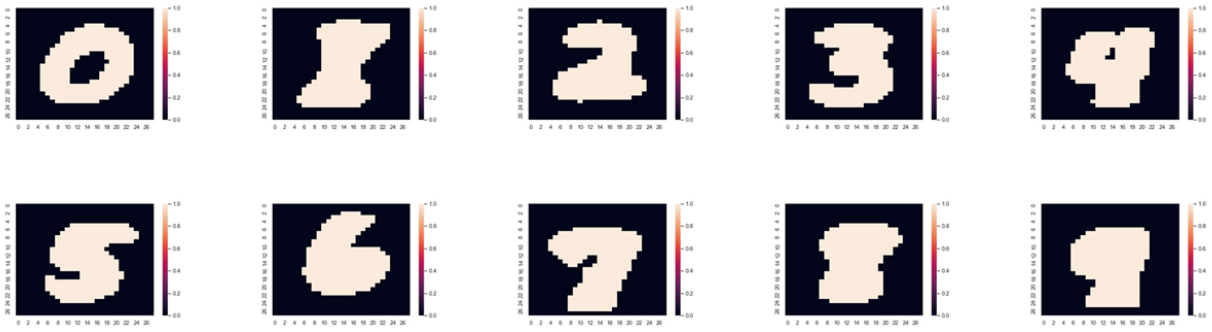


Binary reference: top 32 pixels per digit

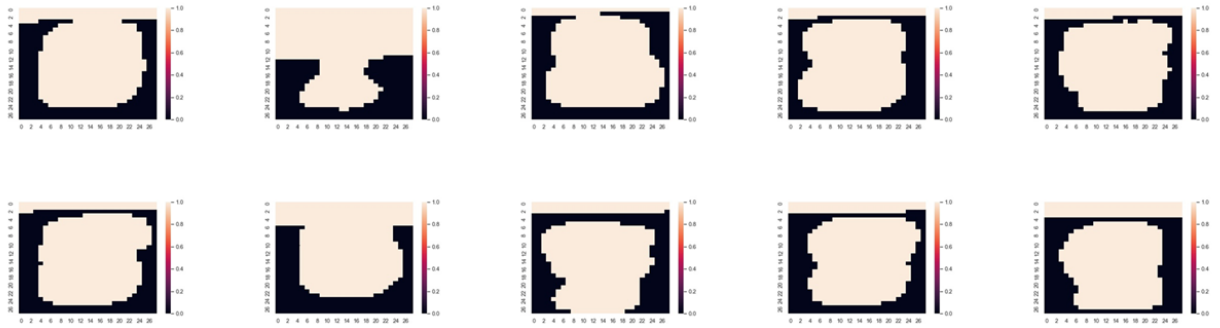


Binary reference: top 64 pixels per digit

Interestingly, increasing the pixel count from there creates a "fuzzy" representation of the digits, which can help create a buffer zone for some of the outlier / oddly-drawn images, but as the pixel count increases significantly, the maps get so fuzzy that digits are no longer distinguishable.



Binary reference: top 256 pixels per digit



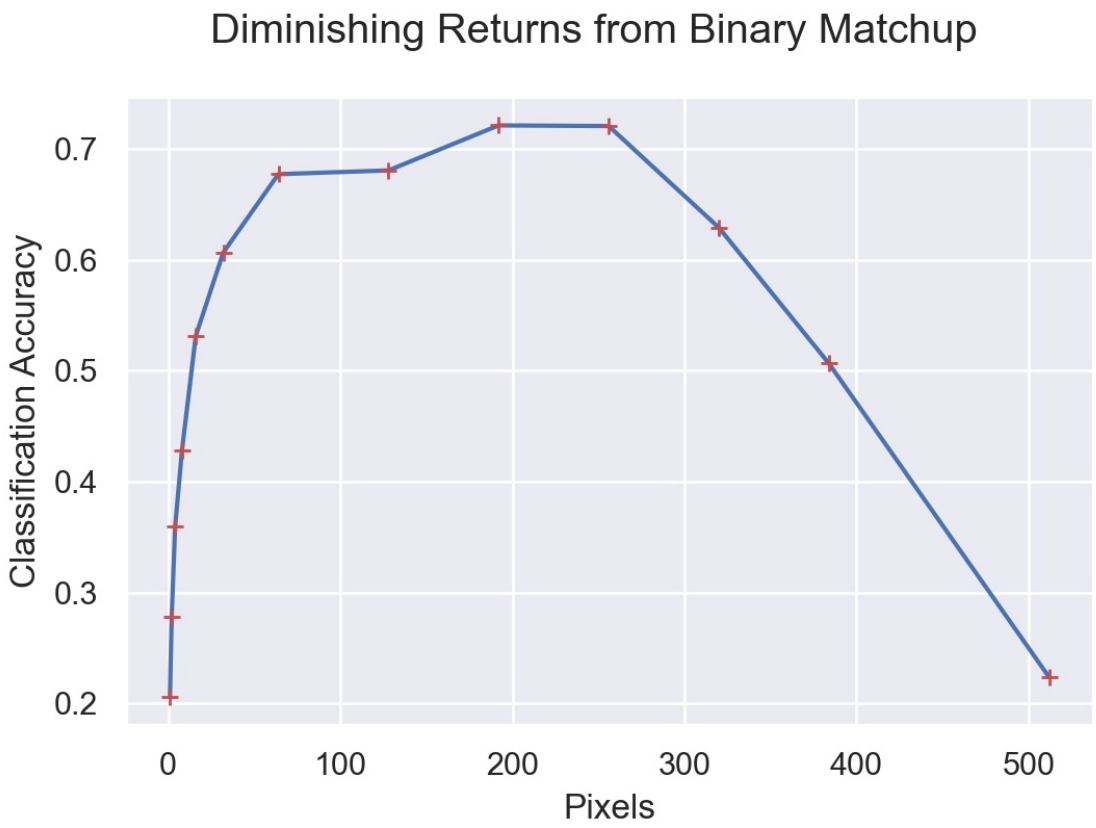
Binary reference: top 512 pixels per digit

To quantify this phenomenon, I made a Python script that would compare the classification accuracy across different values of N when identifying the presence of the top N pixels.

Number of Pixels	Classification Accuracy
1	0.20624124824964993
2	0.2534506901380276
4	0.30686137227445487
8	0.4388877775555111
16	0.5341068213642729
32	0.5947189437887578
64	0.6543308661732347
128	0.6789357871574315
192	0.712742548509702
256	0.7089417883576715
320	0.6149229845969194
384	0.496499299859972
512	0.22864572914582917

Table: Classification Accuracy vs top-N pixel reference

When visualized, we can see a clear pattern of diminishing returns past a certain point, which corresponds to the progressively "fuzzier" images we saw in the heatmaps.



Plot: Classification Accuracy vs top-N pixel reference

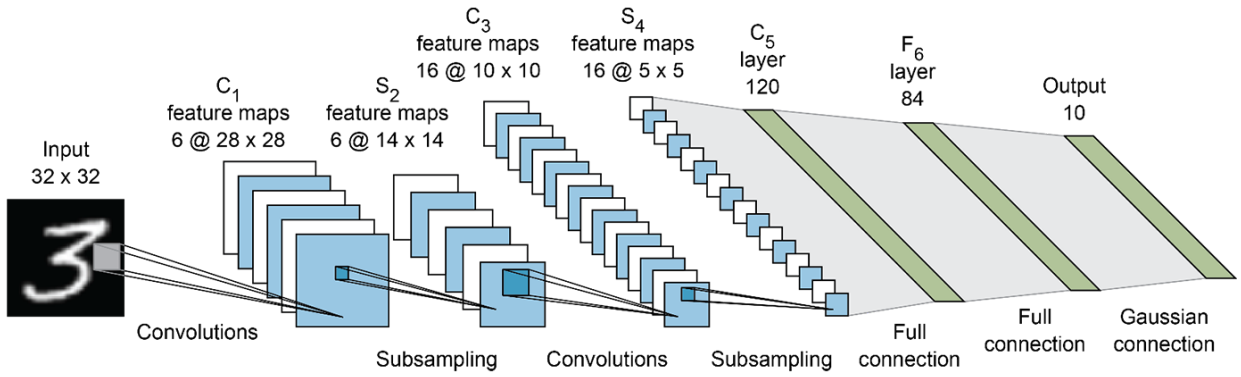
I wanted to take advantage of the most efficient value regime possible, which I took to be the corner of this performance curve: the smallest possible value of N (which would allow a reduction in the number of binary operations) that still had a positive slope. In this case, I selected 64 pixels as the top-N-pixels threshold to continue with for the combinational module.

The code for this module can be found below in the section for digit_top.sv. I used a Python script to automate writing some of the longer lines - I didn't have to actually sit down and type out 784 pixel sums by hand, thankfully. What the sum_64_digit module does is that it takes a 784-bit-wide incoming image and compares it against ten reference blocks, 784-bit-wide variables with 64 binary 1s corresponding to the locations of the top 64 pixels for each digit. There are then ten instances of the tallier module, which perform 10 simultaneous sums of 784 bitwise ANDs between the reference block and the test block. The sum_64_digit module then compares these

sums, and the digit with the largest sum is selected as the correct prediction. As verified using Python, the aggregate accuracy of this model using the top 64 bits is 65.4%.

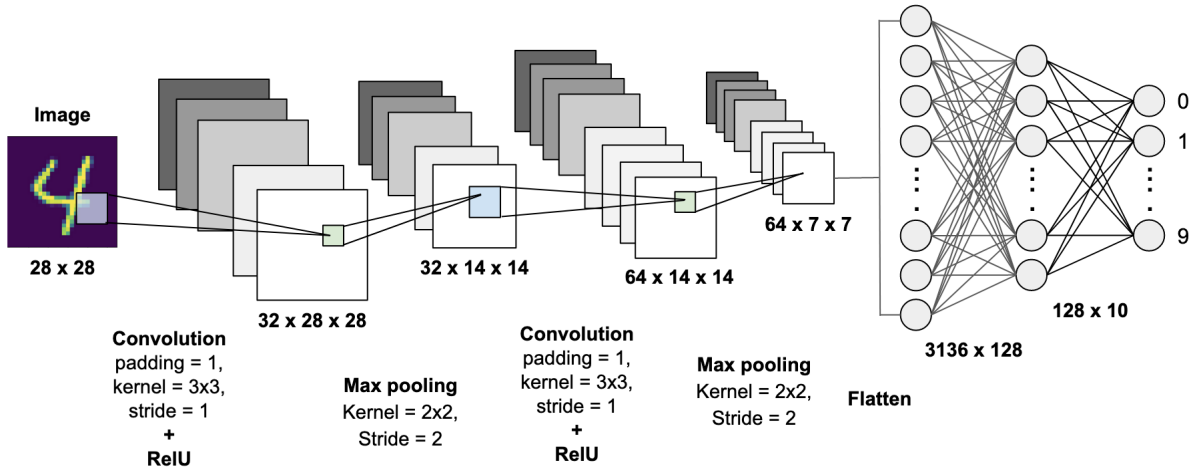
5 Arithmetic Module / Dense Neural Network

For the arithmetic-based module, I wanted to identify a method to approximate the structure of a conventional neural network, while still minimizing the amount of computation necessary onboard the FPGA. As my starting point, I took the classical neural network LeNet, developed by LeCun et al. as one of the original MNIST-processing convolutional neural networks (CNNs). This structure is unique not only because of its relatively compact nature (~6-7 layers) and relative accuracy (>90%), but for the many ways it has been adapted and modified to improve performance. A few examples follow:



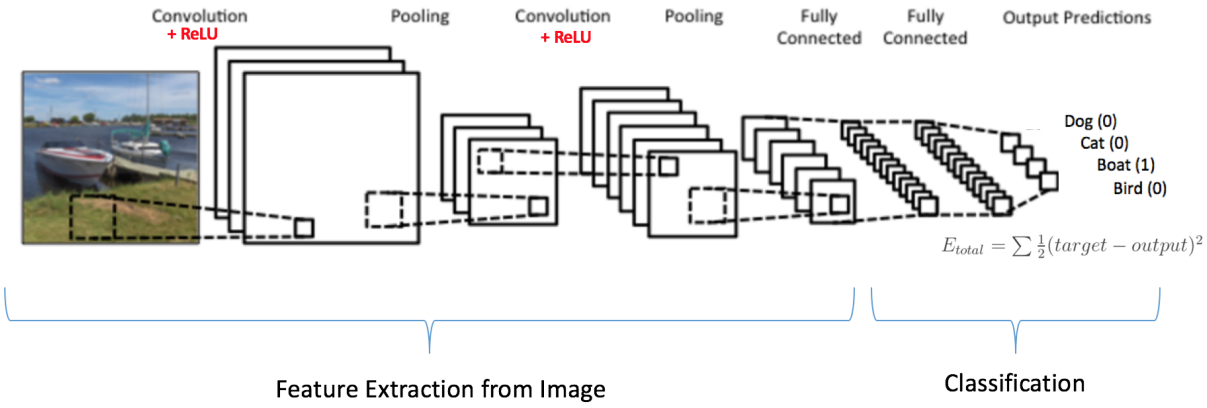
LeNet-5: One implementation (source)

In this example, the input image has been zero-padded (2 pixel border on all sides around the image).



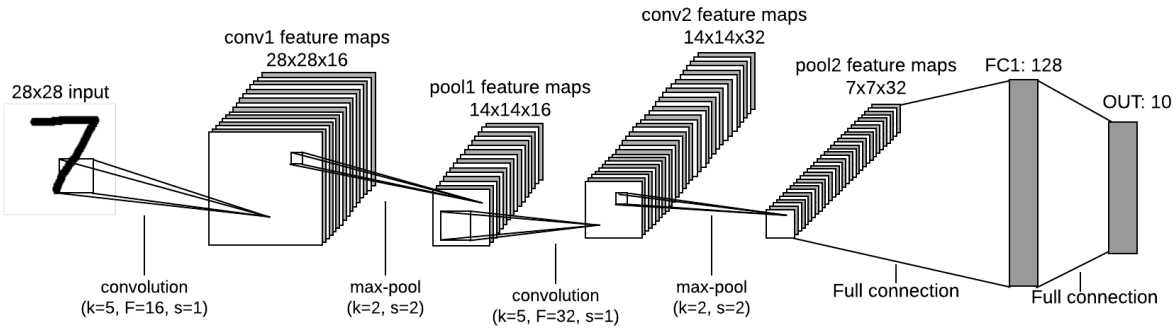
LeNet-5: Another implementation (source)

In this example, the image is not zero-padded. However, the second convolutional layer is very large, and the first fully-connected layer is also very large.



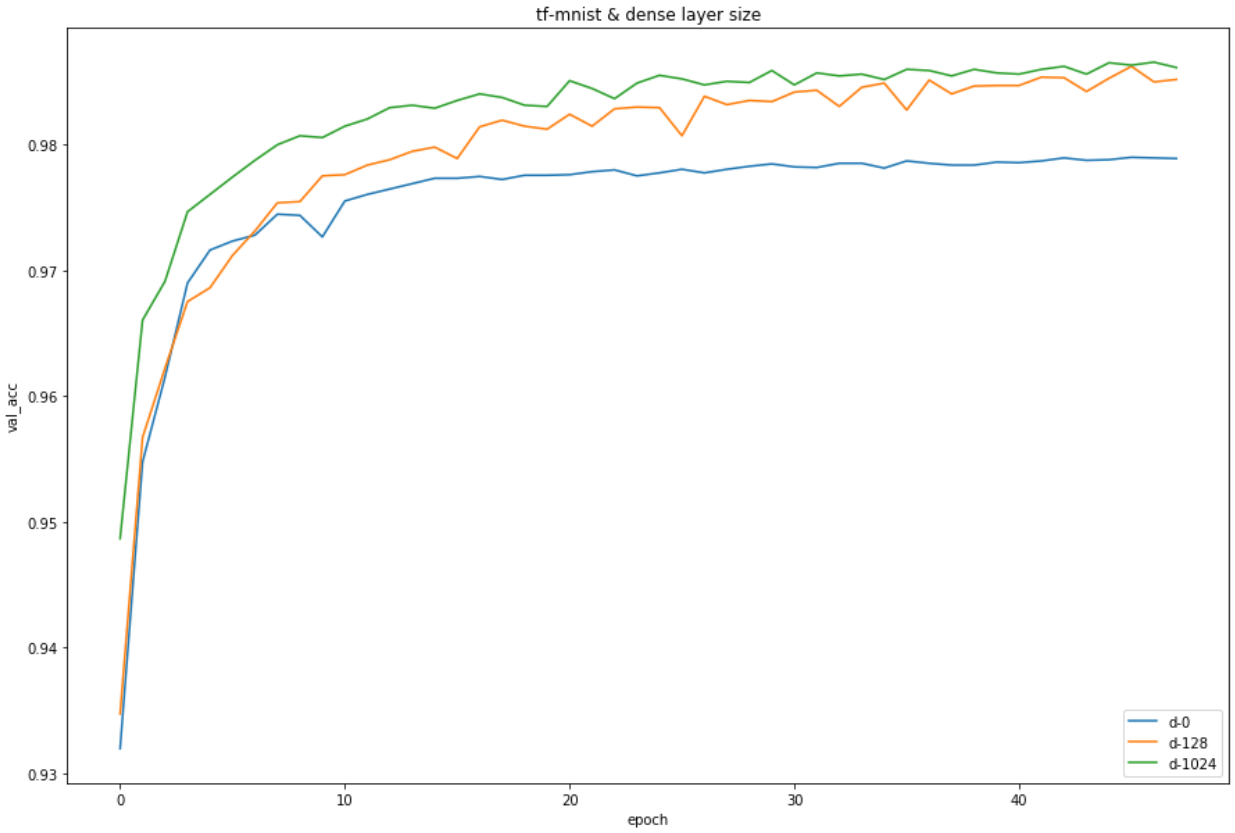
LeNet-5: Use beyond MNIST (source)

Interestingly, this architecture has also found use beyond MNIST digit classification, and can also perform rudimentary image-comprehension tasks due to the convolutional layers, which help identify different visual features (like color contrast, edges, and angles).



LeNet-5: Condensing the fully-connected layers (source)

Based on literature that confirmed that the LeNet structure can be condensed successfully without significant impact to performance, I started to condense and cut out different portions of the network to evaluate performance.

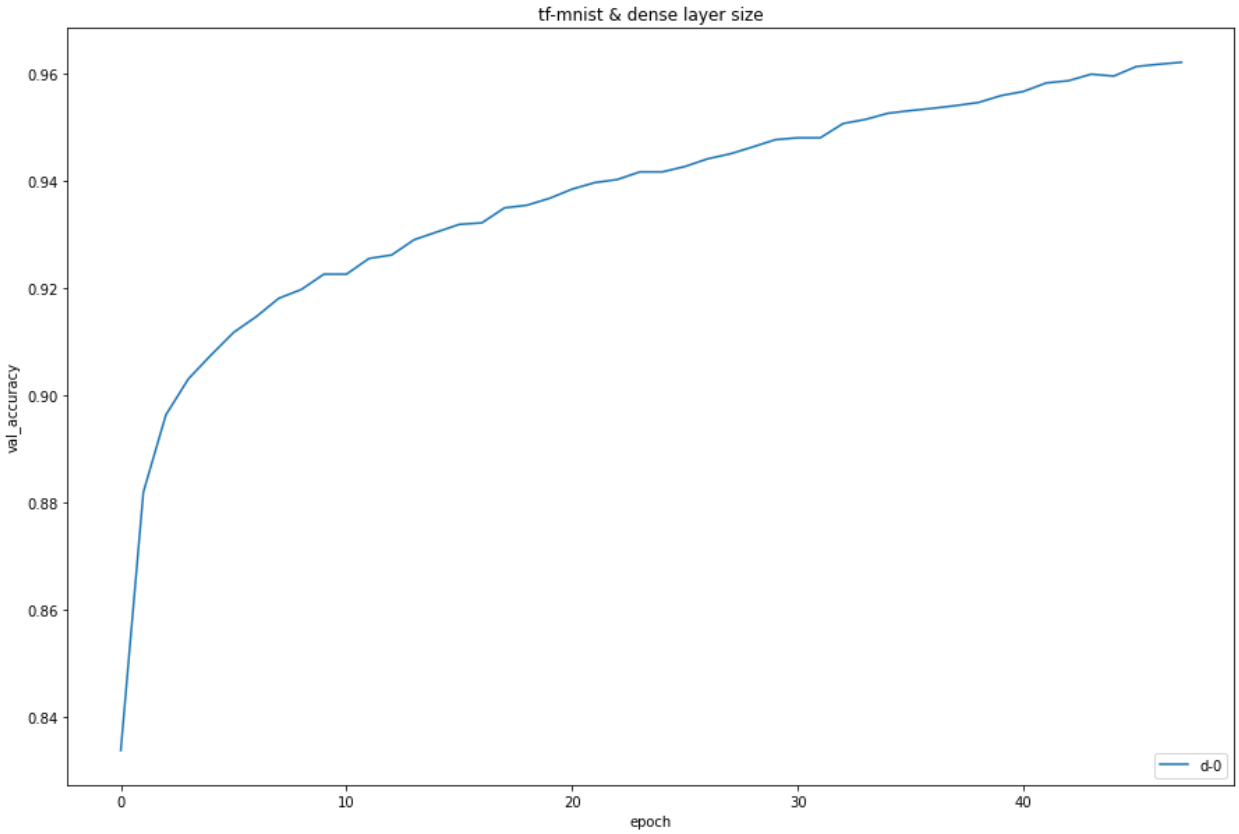


LeNet-5: Performance with different dense-layer sizes (source)

```
Epoch 47/48
13999/13999 [==============================] - 36s 3ms/step - loss: 0.1791 - accuracy: 0.9480 - val_loss: 0.1939 - val_accuracy: 0.9432
Epoch 48/48
13999/13999 [==============================] - 33s 2ms/step - loss: 0.1760 - accuracy: 0.9483 - val_loss: 0.1910 - val_accuracy: 0.9427
```

Readout: Running the original implementation

Running the original implementation, you can see the starting accuracy range of around 94%. This also took around 35 seconds to run per epoch, resulting in a total training time of 21 minutes. While this wasn't bad by any means, I knew I could get much better performance since I do have a discrete GPU in my system. I installed the tensorflow-gpu and Nvidia CUDA packages, and immediately saw my training time chopped down to 8 seconds per epoch. A nearly 5x improvement, going from an i7-9750H to a Quadro T2000.



Accuracy during training: Intermediary model (One convolutional layer + one dense layer)

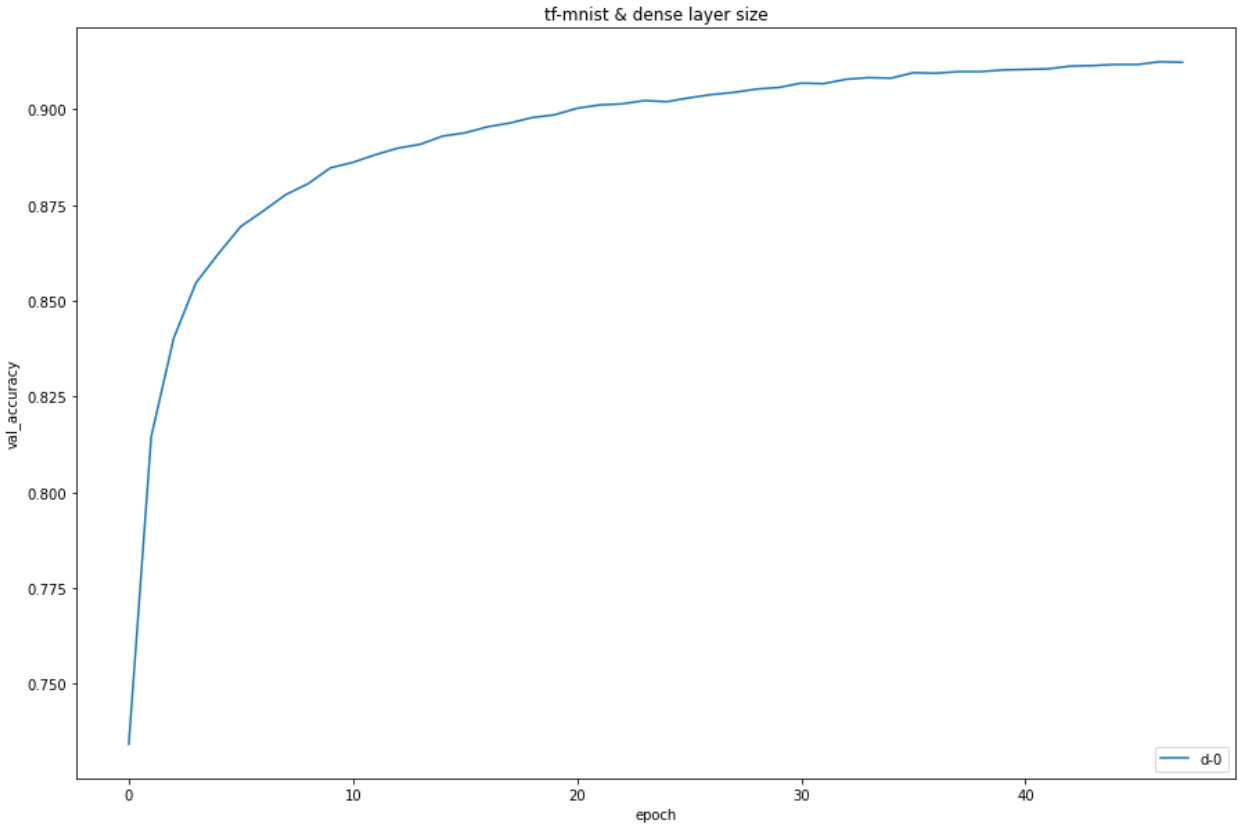
```
Epoch 48/48
55998/55998 [==============================] - 8s
143us/step - loss: 0.1601 - accuracy: 0.9550 -
val_loss: 0.1429 - val_accuracy: 0.9620
```

Readout: Intermediary model testing (one convolutional layer + one dense layer)

This neural net setup performed well, but I was trying to figure out how to implement a convolutional system on the FPGA fabric. While I found some literature on ways this had been done, they did involve fairly nontrivial logic overhead, and I wanted to explore ways to squeeze performance out of more minimal implementations. From [Chen et al, 2018], I saw that dense neural networks often rivaled convolutional neural networks at relatively simple tasks, like MNIST digit classification, and so I tried cutting out the convolutional layers entirely and keeping only a fully-connected layer.

```
Epoch 48/48
62998/62998 [==============================] - 6s
92us/step - loss: 0.4873 - accuracy: 0.8644 -
val_loss: 0.4277 - val_accuracy: 0.8847
```

Readout: Intermediary model testing (max-pooling + one dense layer)



Accuracy during training: Final model testing (one dense layer)

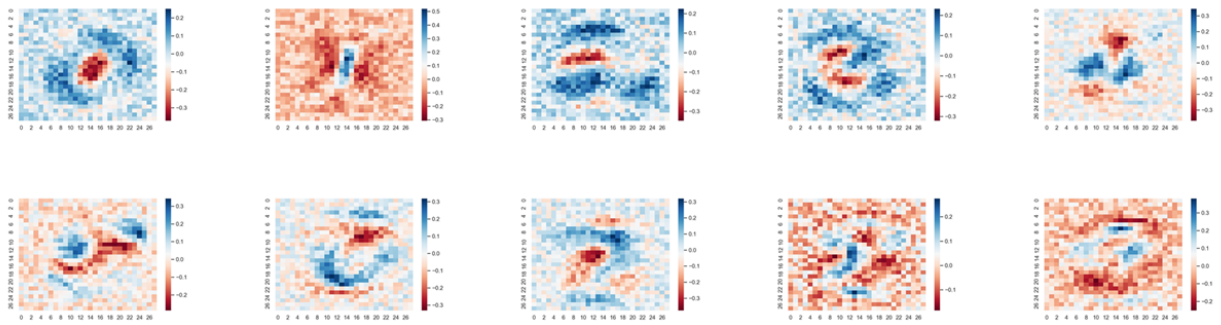
```
Epoch 48/48
62998/62998 [==============================] - 7s
107us/step - loss: 0.3804 - accuracy: 0.8939 -
val_loss: 0.3223 - val_accuracy: 0.9123
```

Readout: Final model testing (one fully-connected layer)

After cutting out all components except for one fully-connected layer, I was still able to achieve 91.2% accuracy on a 90/10 train/test split of the 60,000 images. This was a really promising result, because that meant I could utilize a single-stage implementation, which would allow me to minimize the amount of inter-block

interconnections necessary on the FPGA, as opposed to having multiple fully-connected layers, in which a separate submodule for each layer would have to have inputs from every single output connection in the previous layer. Note here that "val_accuracy" stands for "validation accuracy", on the defined 'test' dataset. This is the accuracy value we are interpreting, as opposed to the run "accuracy" measured during this epoch of the algorithm training.

As displayed in the figure below, the 1-layer dense neural network operates on 10 series of 784 weights, one series per digit and one weight per pixel, which denotes whether that particular pixel is a positive or negative predictor of that particular digit, and to what extent. For example, the strong red circle in the middle of the 0 denotes that if any pixels are activated in that region, then the negative weights will be added and the probability that the given image is a 0 will decrease correspondingly. Similarly, the strong center blue region in the 1 denotes that if any pixels are activated in that vertical rectangle, it will increase the probability that the input data represents a 1.



Heatmap: Digit Weights (32-bit floating point)

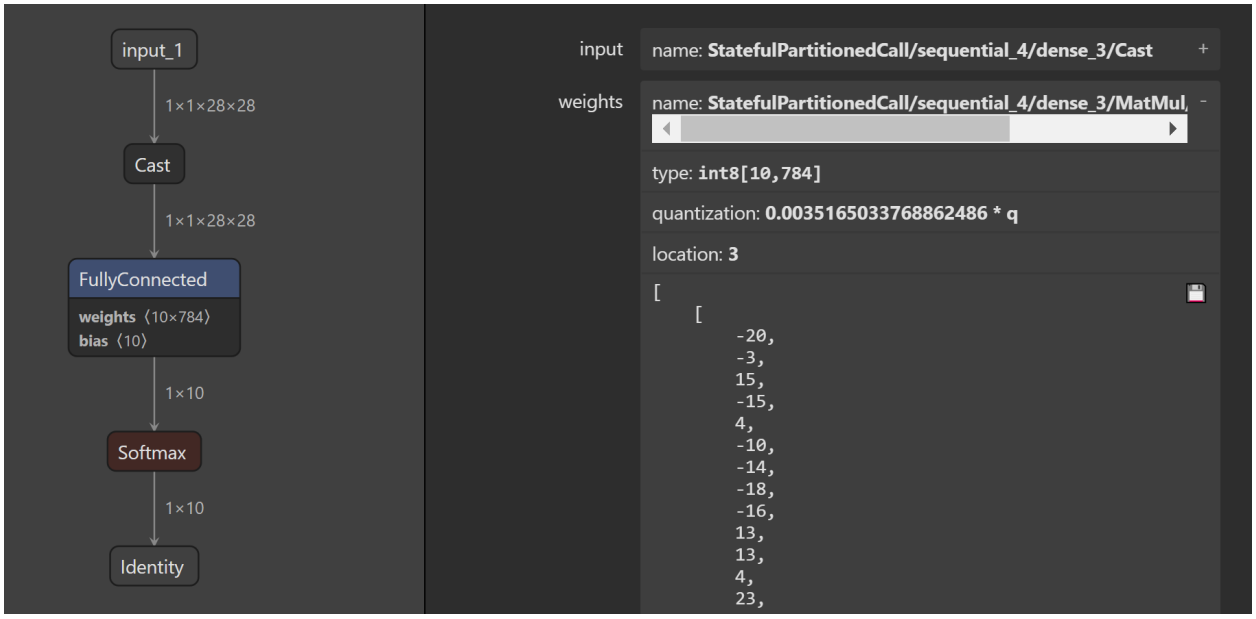
The final hurdle before implementing this algorithm on the FPGA was quantizing weights. The quandary of whether or not to use floating-point math is something that has been often questioned and studied. Recent industry use has advocated for the use of lower-precision weights as a way to accelerate large-scale machine learning computation; Google themselves claimed that using binary 16-bit floating-point numbers instead of normal 32-bit decimals was the "secret to high performance on Cloud TPUs" (source). Because I was already using binary digits to represent the MNIST

images in the FPGA, I didn't have to worry about multiplication - I would already be multiplying the weights by 1s and 0s, not greyscale integers. What left was the arithmetic involved in summing the weights for a particular image in the layer of neurons, and floating-point arithmetic is time- and hardware-intensive. To simplify the arithmetic necessary on the FPGA, I decided to quantize the weights from 32-bit floating-point decimals to 8-bit integers. I could do this in Python or Matlab, but instead used a new function in Tensorflow 2.0 that does this. After installing tensorflow 2.0, I also found that it is marginally faster at training, too (5s 87us vs 7s 107us in the last epoch)

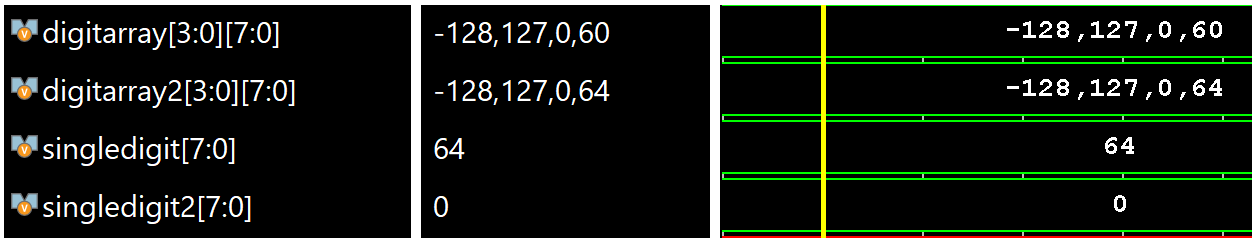
```
Epoch 48/48
62998/62998 [==============================] - 5s 87us/
sample - loss: 0.3792 - accuracy: 0.8956 - val_loss:
0.3211 - val_accuracy: 0.9116
```

Readout: Final model testing, repeated using tensorflow 2.0

After the model was quantized, I used a web tool to extract a numpy dictionary of the quantized weights from the compressed .tflite model (ready for mobile applications). Instead of porting the algorithm to a phone like the makers at TensorFlow intended, I hopped over to Vivado and used the weights to populate 784 x 8-bit weight arrays for each of the ten digits.



Visualization: Network parameters displayed using netron.app

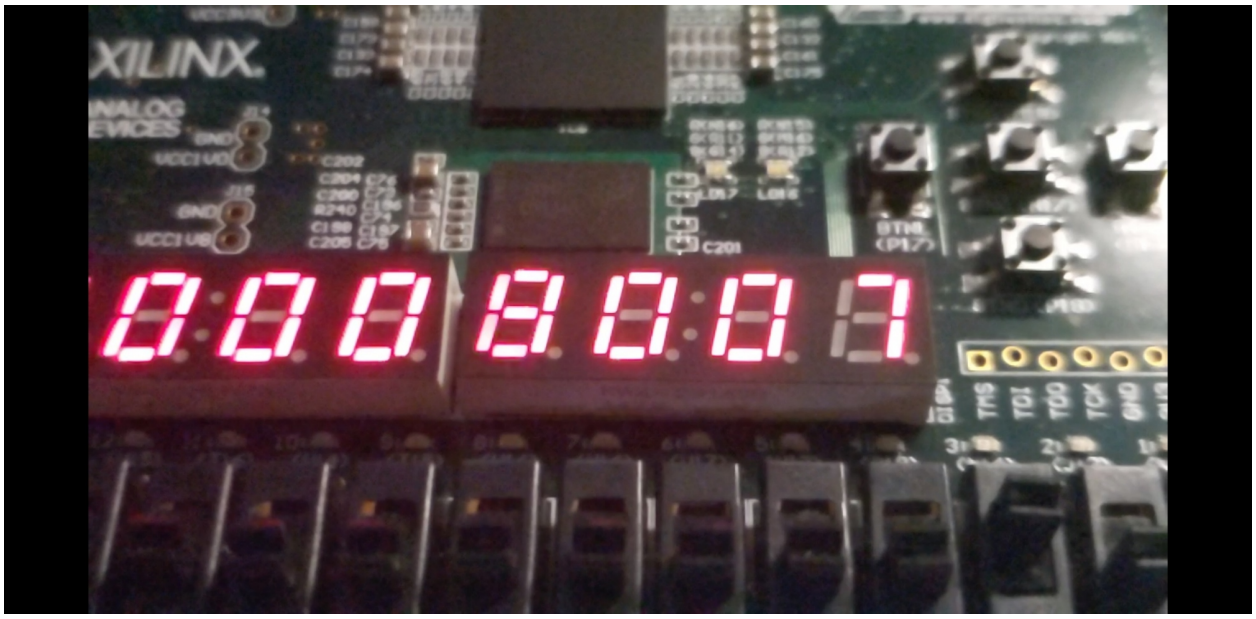


Simulation Readout: Quantized weights entered into 8-bit Verilog arrays

In the original algorithm, the final output-generating function was softmax, which will generate a probability value for each digit output based on its relative value compared to the other outputs. While there have been implementations of softmax-like functions, as in [Kouretas and Paliouras, 2020], they involve even more logic overhead and clock cycles. Instead, I decided to just use the raw value of the sums, since the result would be the same; the only impact would be not having a human-readable probability percentage, but that can be added in later. Using the weights, I was able to implement a similar decision-making strategy as in the combinational module: sum up the values, and pick the largest one. Because the weights are eight-bit integers, a sufficiently large variable is enough to hold the sum, and the addition operation can be done with the primitive addition operation, and arithmetic IP units do not have to be instantiated.

This combinational setup doesn't use tallier modules like the combinational method, however, since the values are continuously summed each clock cycle in a separate variable for each possible digit candidate. This entire setup, visible in densenet.sv, is able to generate a prediction output in 784 clock cycles. For simplicity and robustness, I have added a buffer period to ensure that my output has always stabilized. The always_ff loop operates similar to a finite state machine, tallying the sums at the index denoted by the current state, repeating these sums for 784 cycles, and then resetting the sum variables and state to 0, allowing for a new prediction to be made.

That final note was something that I hadn't gotten right on the first try. Initially, I had the arithmetic module executing only on a button-press, but for some reason, after the first time I pressed the button, it just wouldn't display the correct value. I thought it might be a debounce issue, so then I removed the button-press and had it trigger on an input value state change. Still, the incorrect results. Then I removed any input triggers and created the state machine to continuously loop through. My combinational module was working fine, but on the LED display I used for debugging, the output from the arithmetic module was flickering continuously, as if the output was cycling through 0-9 instead of settling on one prediction. I tried many different ways to debug this.

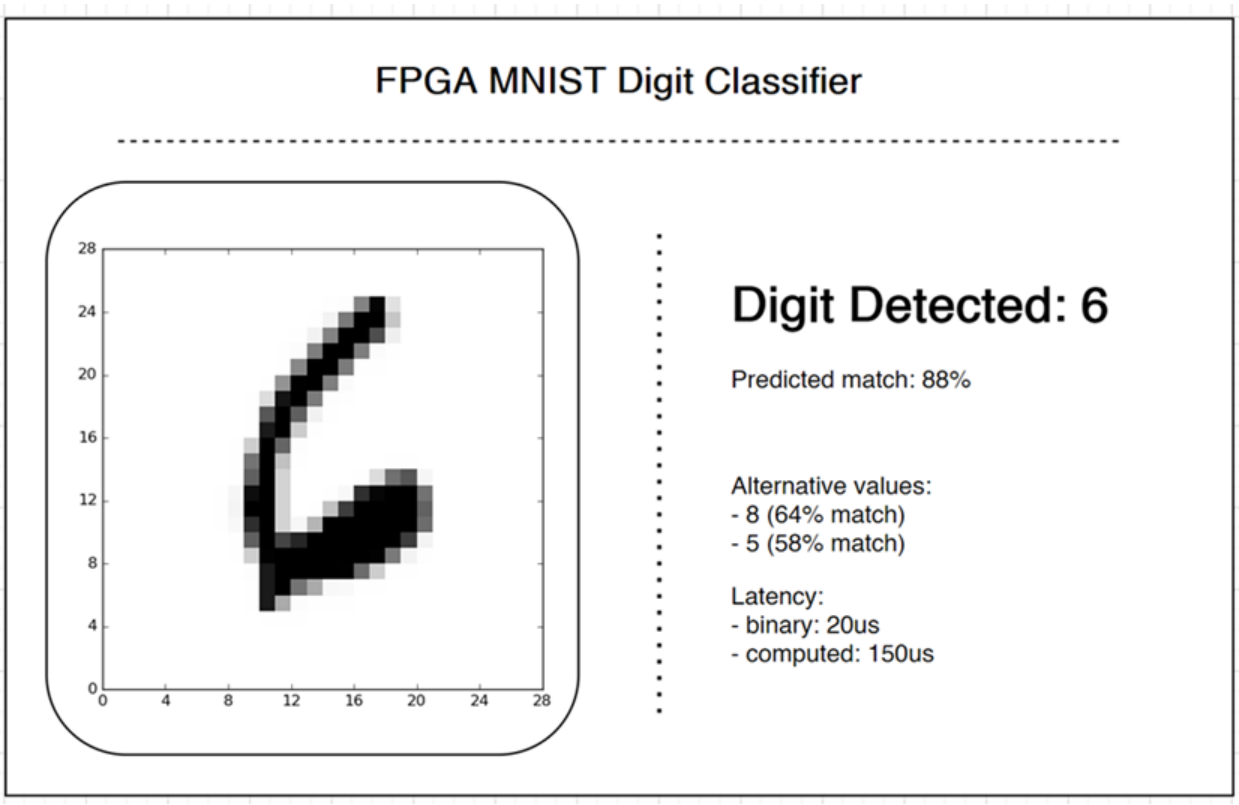


Flickering Output: Arithmetic Output is the 4th digit from the right (1-indexed)

In this photograph (actually a screenshot of a slow-motion video I took to try to ascertain whether all values were being looped through, or just some), the result looks like some combination of 3, 8, and any number of additional results. I was perplexed and mildly annoyed but it was 3AM or so and so I went to bed, but as I tried to fall asleep, I was running through the code in my mind to try to mentally debug. (This is generally a good strategy - either you discover the issue, or you fall asleep. Sometimes, however, it causes frustration and prevents sleep, in which case it is not a good strategy). Luckily, the first case occurred and while I was still lucid, I realized that I wasn't actually resetting the sum variables after the 784 cycles completed - I was only resetting the state. This was having the effect of causing the sums to continuously keep incrementing, and after variables looped past 0, different digits ended up having the largest values - which caused many different digits to be activated on the LED, one after another. I got out of bed, implemented the variable-emptying, it worked, and I went back to bed happy. Despite going to bed late, this was possibly a pareto-optimal outcome, because it resulted in more restful sleep than I would have had otherwise.

6 VGA GUI Module

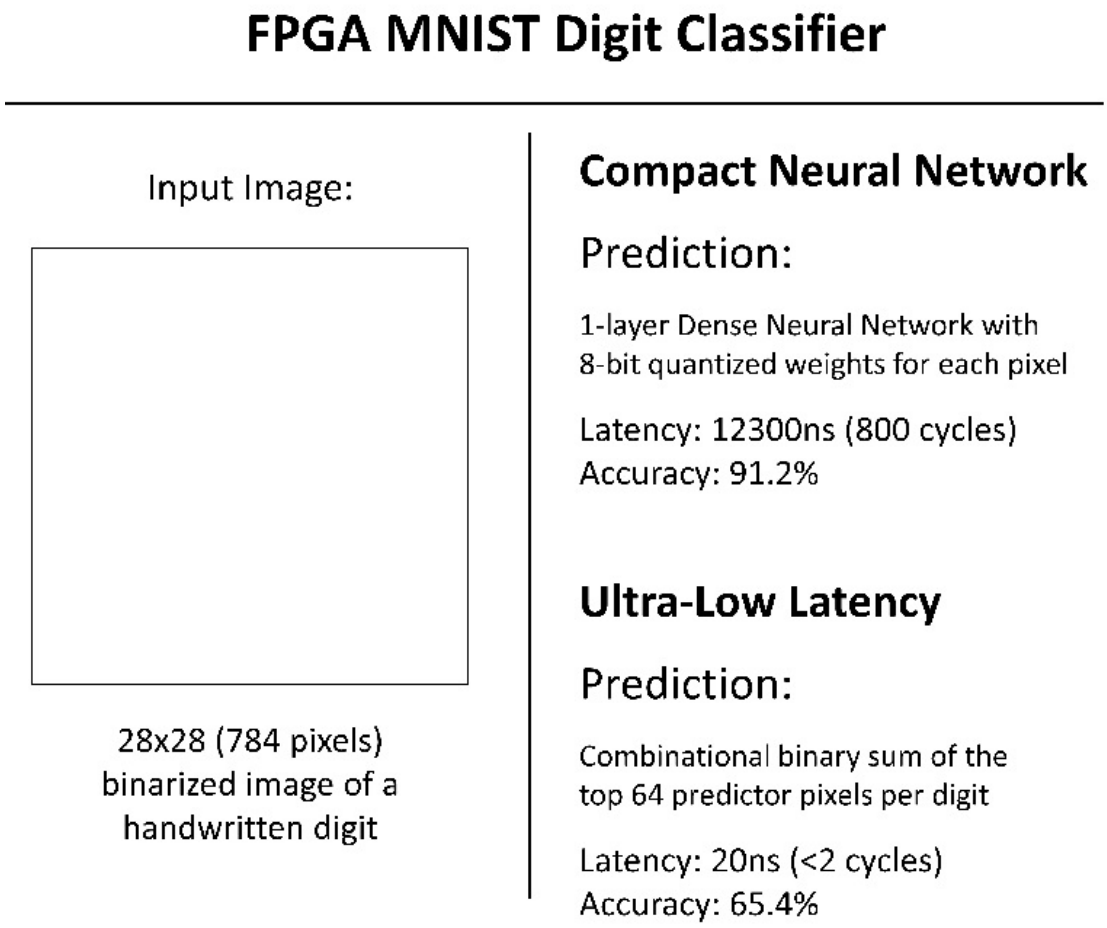
With the output from the arithmetic module sorted out, I was able to move on to the GUI module. This was my original idea for a GUI, and I had put off trying to figure out how to actually display a GUI on-screen. Rather than trying to position a bunch of text manually using the FPGA, I decided to just present the GUI background using an image, and have dynamic components for the three changing entities: the input image, the combinational prediction, and the neural-net prediction.



GUI: Draft 1 (presentation version)

This is the first draft I created for a GUI. I made this as a PowerPoint slide fit to 1024x768 resolution, exported it to an image, and converted it to a COE. I quickly ran into the realization that the BRAM limitations on the chip would not be able to support that large of an image. If I had converted it to be binary instead of 8-bit, then it would possibly fit, but I kept the 8-bit format for ease of iteration using the provided python

scripts for COE generation, as I ended up tweaking the GUI and having to regenerate and reload the image file 5 or 6 times.

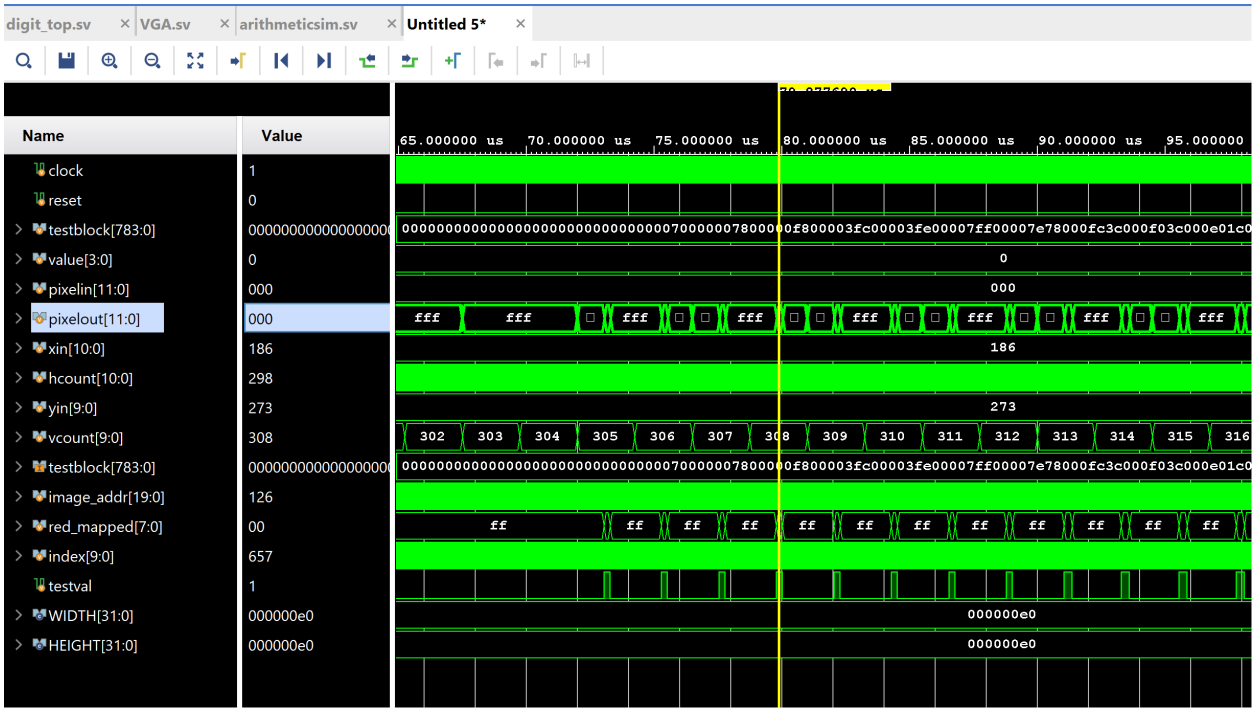


GUI: Draft 2 (initial implementation, without explanatory info)

To implement the image display, I leveraged the existing image-display framework from the Pong Lab (Lab 3). In fact, my current display module is still named pong_game, and the GUI is named deathstar_test, as an homage to the death star image module that happily traversed the screen without a care as to the true identity of various handwritten digits floating through the cosmos.

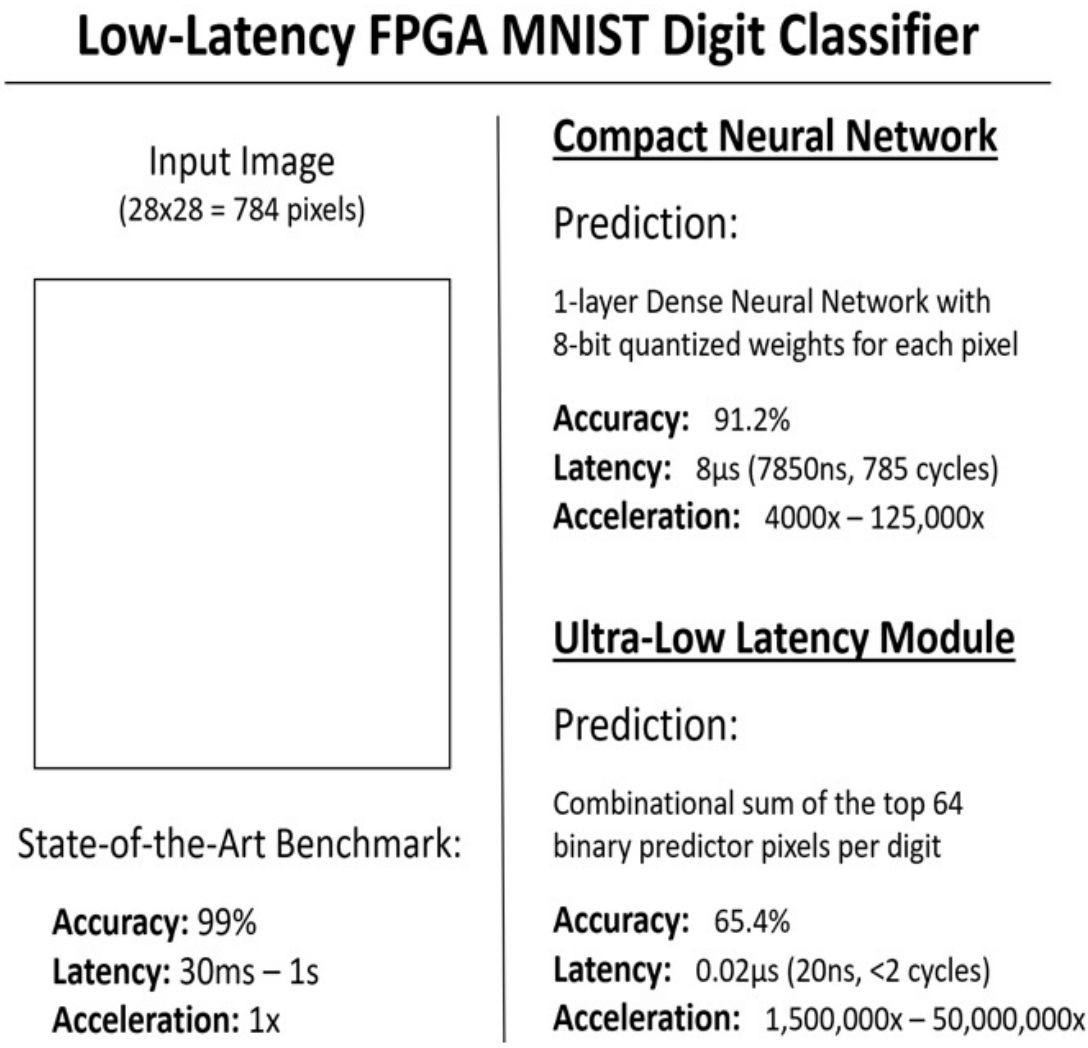
To fit the GUI background in BRAM, I reduced the scale and size of the image, and settled on 761 x 641, which, at 3900 Kbits, would leave another 20% of the BRAM free. This proved fortuitous, as I ended up with exactly 99% BRAM usage at the end.

Before then, however, I ran into an issue where the GUI wasn't displaying at all, and then as a huge blurry streak. I created a simulation testbench to mock up the VGA module, and was able to work out the issue: a size mismatch between the COE and the image file, which was causing over-reads and nothing was displaying right as a result.



Simulation readout: functional VGA module

In my final design for the GUI background, I made two main adjustments: first, I added explanatory information including the benchmark information, help put the modules in context. Second, I realized that on my 1920 x 1080 monitor, auto-scaling for a 1024 x 768 image resulted in displaying (1080/768 * 1024) = 1440 images across a 1920-pixel region, resulting in a display that was 33% stretched out horizontally. To compensate for this, I squashed the GUI image by 15% horizontally, which made the distortion less noticeable.

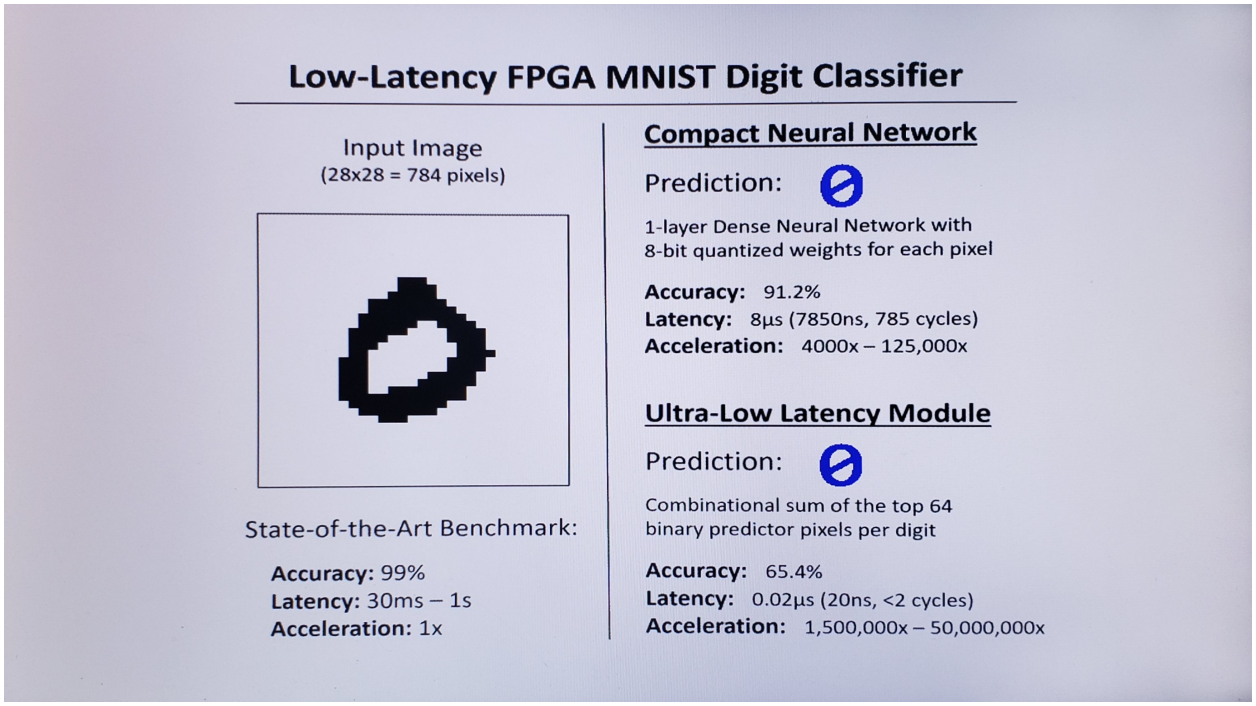


GUI: Draft 4 (final implementation)

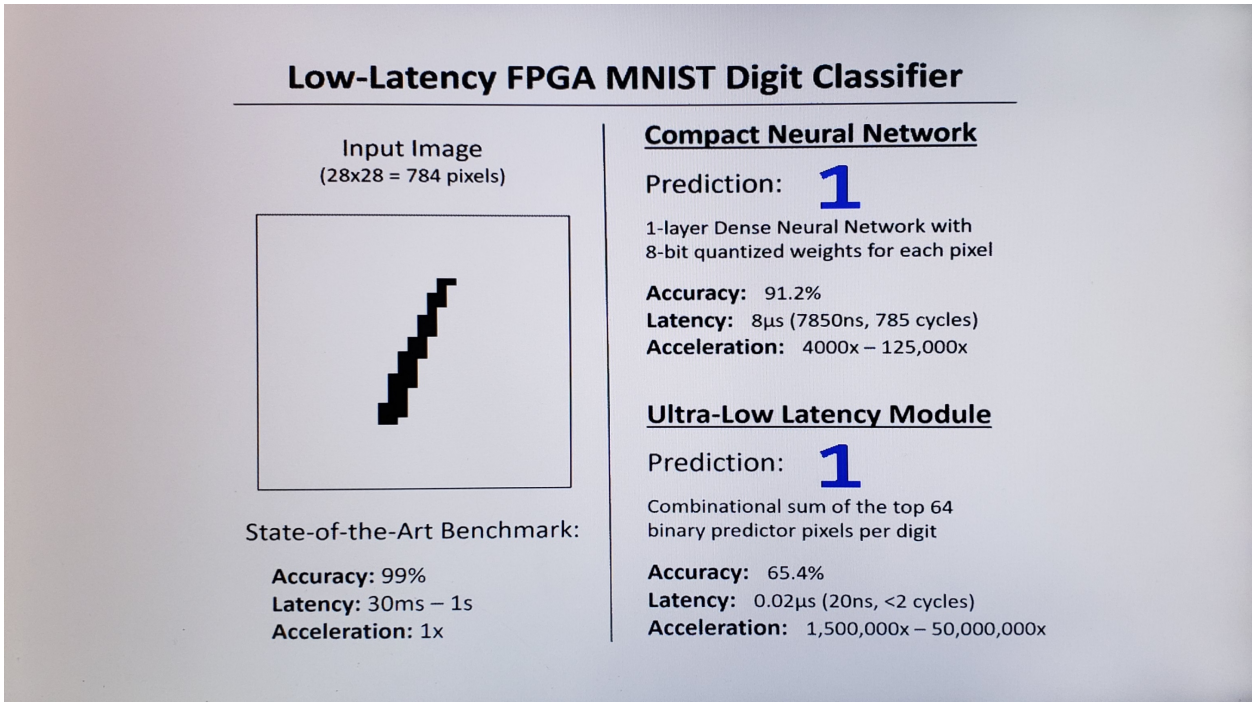
Once I had the GUI visible on the screen, the next task was to implement the dynamic elements: the input image, and the real-time predictions from the compact neural network module (arithmetic / DNN) and the ultra-low latency module (combinational). To show the predictions, I used the provided 48x48 number COEs and positioned them on the screen appropriately by measuring a pixel-accurate mockup in MS Paint. The normal COEs display a black background and a white number; I recolored the image with four steps. First, I added logic to the digit_blob module to output white if the output pixel was 0 (black), which created a white background. Next, I assigned the output values to the blue component of the {R,G,B} array output, which made the digit blue. Then, the anti-aliasing didn't match perfectly with the color map

COE I was using that was generated by the Python script for my GUI image, and there were little fragmented remnants in many digits. I went into the COE file for the numbers and adjusted those components manually. Finally, I noticed that the COE generation had created a marker value of '15' at the upper left (start) corner of each COE image, which resulted in one blue pixel at the top left of every digit. I edited that out, and that made everything look seamless.

Next, I needed to get the display of the actual input MNIST image. I added it to the end of the pipeline (before a little white blob performing patch-correction of some stray gray pixels) and created an mnist_blob module to display either black (1) or white (0) based on the pixels in the 784-bit-wide input test image. One main issue was that the input image is 28 x 28, and that would be way too small for this GUI, so I needed to scale the image up somehow. At first, I tried to figure out how exactly I could multiply the number of pixels by 10 and then do a modulo for the module to correctly increment from one row to another, but this would fall into the overly-complicated-implementation category of suboptimal approaches. I then realized that I could scale the image up by a factor of 8, simply by taking the input VGA pixel position and dividing it by 8 by right-shifting by 3 bits. This would cause the VGA module to stay at the same input digit pixel for 8 horizontal pixels, and 8 vertical pixels, resulting in the desired scale-up in proportion with the rest of the GUI. At first, this didn't display anything, but the issue turned out to be Vivado incorrectly parsing the parentheses in the bit-shift, and a wondrously complete GUI was feasible. I have included photographs of 10 test digits displayed over VGA on a monitor connected to the FPGA.

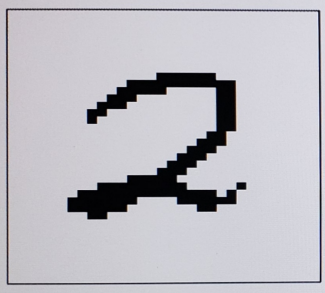


GUI: Final Implementation on VGA-connected monitor



Low-Latency FPGA MNIST Digit Classifier

Input Image
(28x28 = 784 pixels)



State-of-the-Art Benchmark:

Accuracy: 99%
Latency: 30ms – 1s
Acceleration: 1x

Compact Neural Network

Prediction: 2

1-layer Dense Neural Network with
8-bit quantized weights for each pixel

Accuracy: 91.2%
Latency: 8µs (7850ns, 785 cycles)
Acceleration: 4000x – 125,000x

Ultra-Low Latency Module

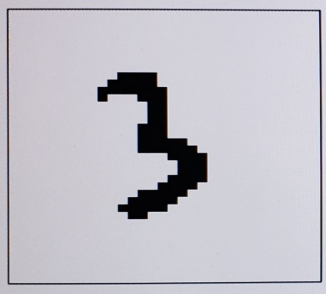
Prediction: 2

Combinational sum of the top 64
binary predictor pixels per digit

Accuracy: 65.4%
Latency: 0.02µs (20ns, <2 cycles)
Acceleration: 1,500,000x – 50,000,000x

Low-Latency FPGA MNIST Digit Classifier

Input Image
(28x28 = 784 pixels)



State-of-the-Art Benchmark:

Accuracy: 99%
Latency: 30ms – 1s
Acceleration: 1x

Compact Neural Network

Prediction: 3

1-layer Dense Neural Network with
8-bit quantized weights for each pixel

Accuracy: 91.2%
Latency: 8µs (7850ns, 785 cycles)
Acceleration: 4000x – 125,000x

Ultra-Low Latency Module

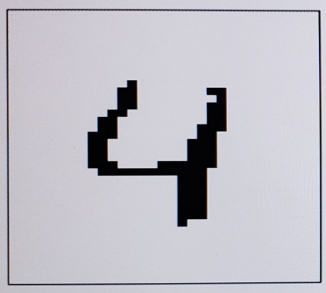
Prediction: 3

Combinational sum of the top 64
binary predictor pixels per digit

Accuracy: 65.4%
Latency: 0.02µs (20ns, <2 cycles)
Acceleration: 1,500,000x – 50,000,000x

Low-Latency FPGA MNIST Digit Classifier

Input Image
(28x28 = 784 pixels)



State-of-the-Art Benchmark:

Accuracy: 99%
Latency: 30ms – 1s
Acceleration: 1x

Compact Neural Network

Prediction: **4**

1-layer Dense Neural Network with
8-bit quantized weights for each pixel

Accuracy: 91.2%
Latency: 8µs (7850ns, 785 cycles)
Acceleration: 4000x – 125,000x

Ultra-Low Latency Module

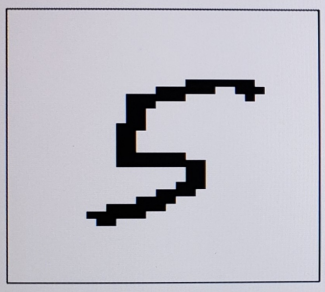
Prediction: **4**

Combinational sum of the top 64
binary predictor pixels per digit

Accuracy: 65.4%
Latency: 0.02µs (20ns, <2 cycles)
Acceleration: 1,500,000x – 50,000,000x

Low-Latency FPGA MNIST Digit Classifier

Input Image
(28x28 = 784 pixels)



State-of-the-Art Benchmark:

Accuracy: 99%
Latency: 30ms – 1s
Acceleration: 1x

Compact Neural Network

Prediction: **5**

1-layer Dense Neural Network with
8-bit quantized weights for each pixel

Accuracy: 91.2%
Latency: 8µs (7850ns, 785 cycles)
Acceleration: 4000x – 125,000x

Ultra-Low Latency Module

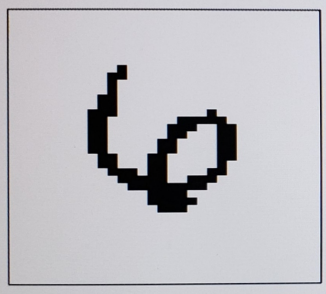
Prediction: **5**

Combinational sum of the top 64
binary predictor pixels per digit

Accuracy: 65.4%
Latency: 0.02µs (20ns, <2 cycles)
Acceleration: 1,500,000x – 50,000,000x

Low-Latency FPGA MNIST Digit Classifier

Input Image
(28x28 = 784 pixels)



State-of-the-Art Benchmark:

Accuracy: 99%
Latency: 30ms – 1s
Acceleration: 1x

Compact Neural Network

Prediction: **6**

1-layer Dense Neural Network with
8-bit quantized weights for each pixel

Accuracy: 91.2%
Latency: 8µs (7850ns, 785 cycles)
Acceleration: 4000x – 125,000x

Ultra-Low Latency Module

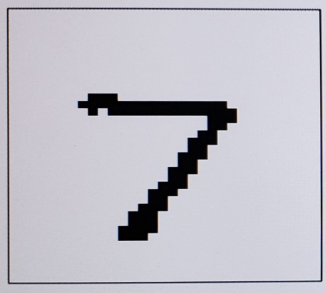
Prediction: **6**

Combinational sum of the top 64
binary predictor pixels per digit

Accuracy: 65.4%
Latency: 0.02µs (20ns, <2 cycles)
Acceleration: 1,500,000x – 50,000,000x

Low-Latency FPGA MNIST Digit Classifier

Input Image
(28x28 = 784 pixels)



State-of-the-Art Benchmark:

Accuracy: 99%
Latency: 30ms – 1s
Acceleration: 1x

Compact Neural Network

Prediction: **7**

1-layer Dense Neural Network with
8-bit quantized weights for each pixel

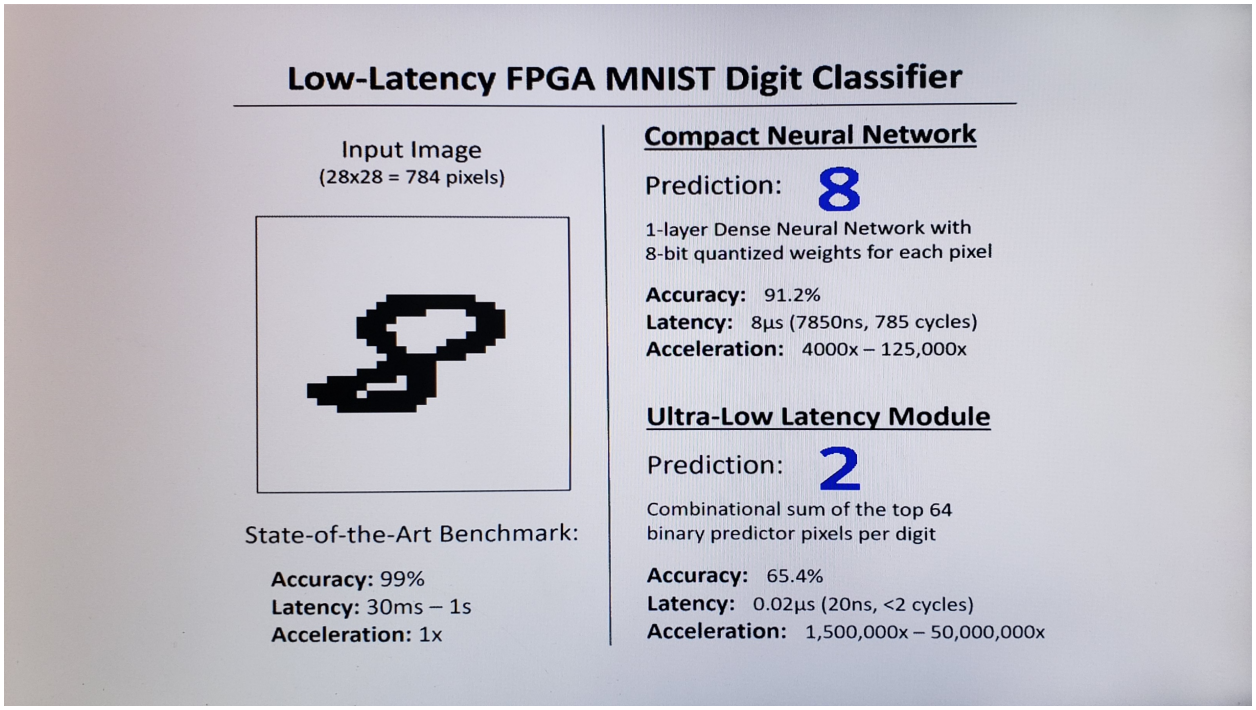
Accuracy: 91.2%
Latency: 8µs (7850ns, 785 cycles)
Acceleration: 4000x – 125,000x

Ultra-Low Latency Module

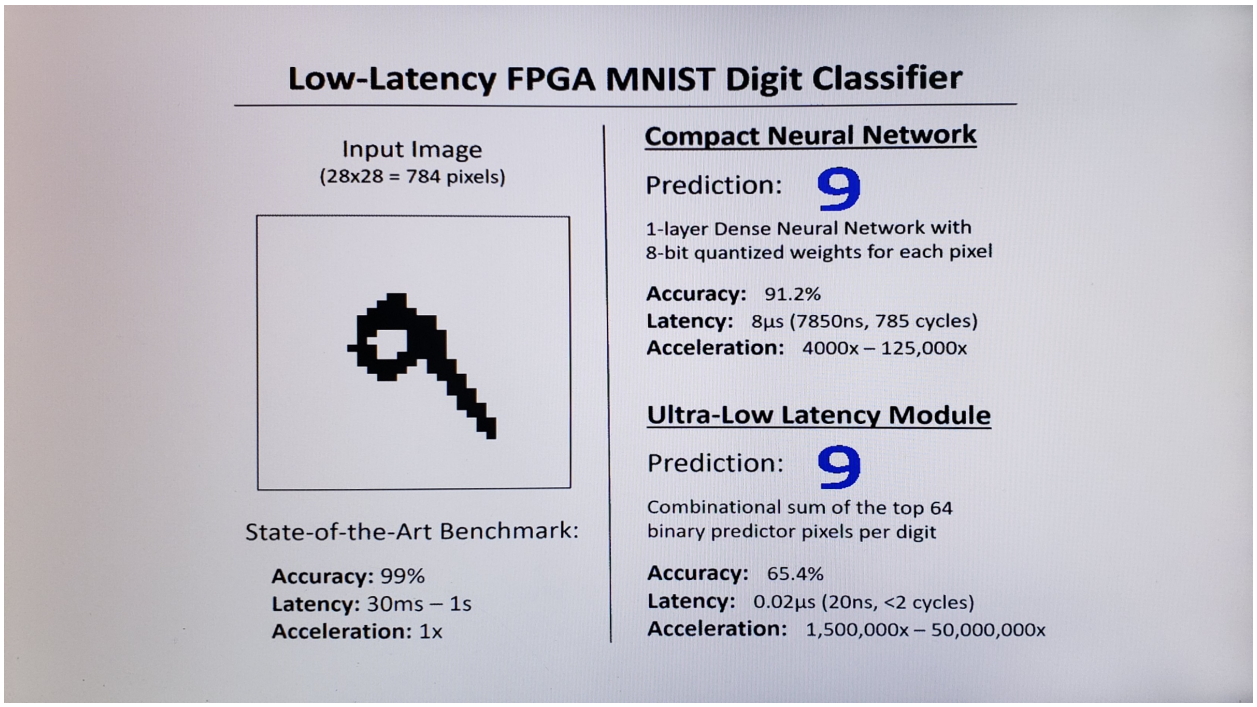
Prediction: **7**

Combinational sum of the top 64
binary predictor pixels per digit

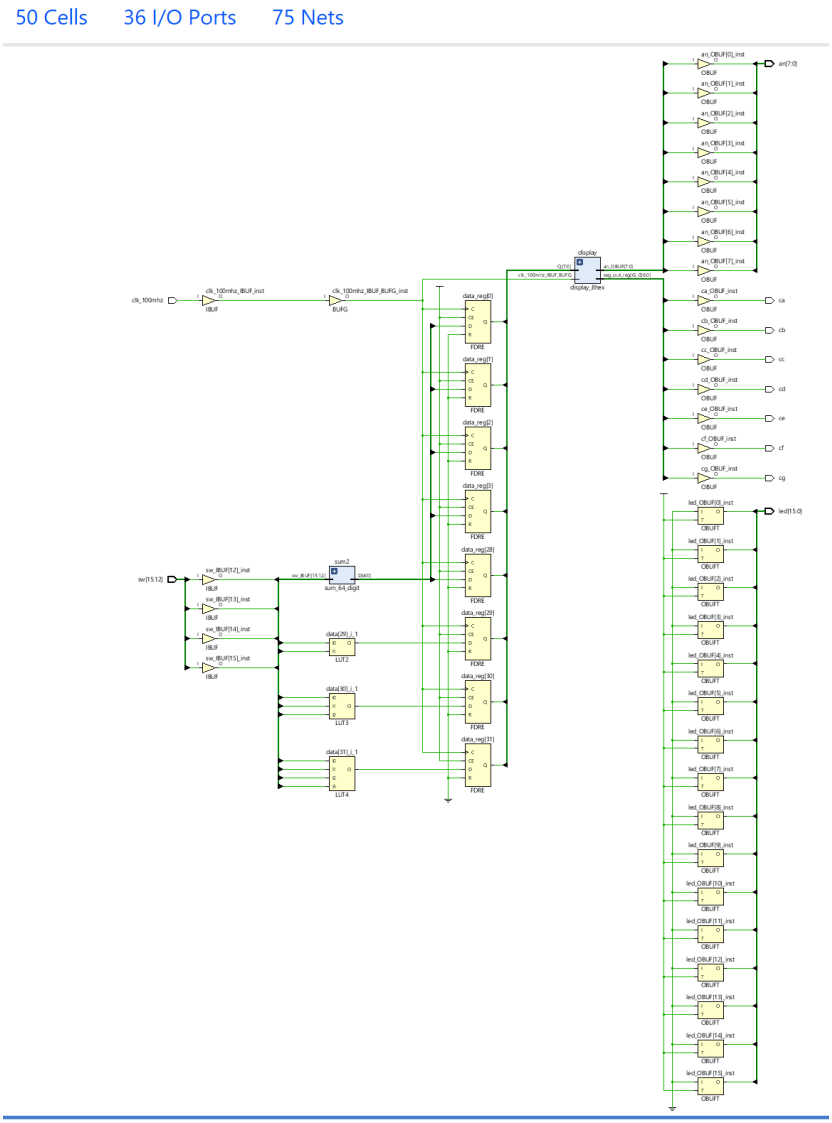
Accuracy: 65.4%
Latency: 0.02µs (20ns, <2 cycles)
Acceleration: 1,500,000x – 50,000,000x



This 8 is an example of a misclassification in the lower-accuracy combinational module. The side-slanted 8 more closely matches the pixel activations in the lower curve of a handwritten 2, so a 2 is picked over an 8.



7 Discussion: FPGA Implementation

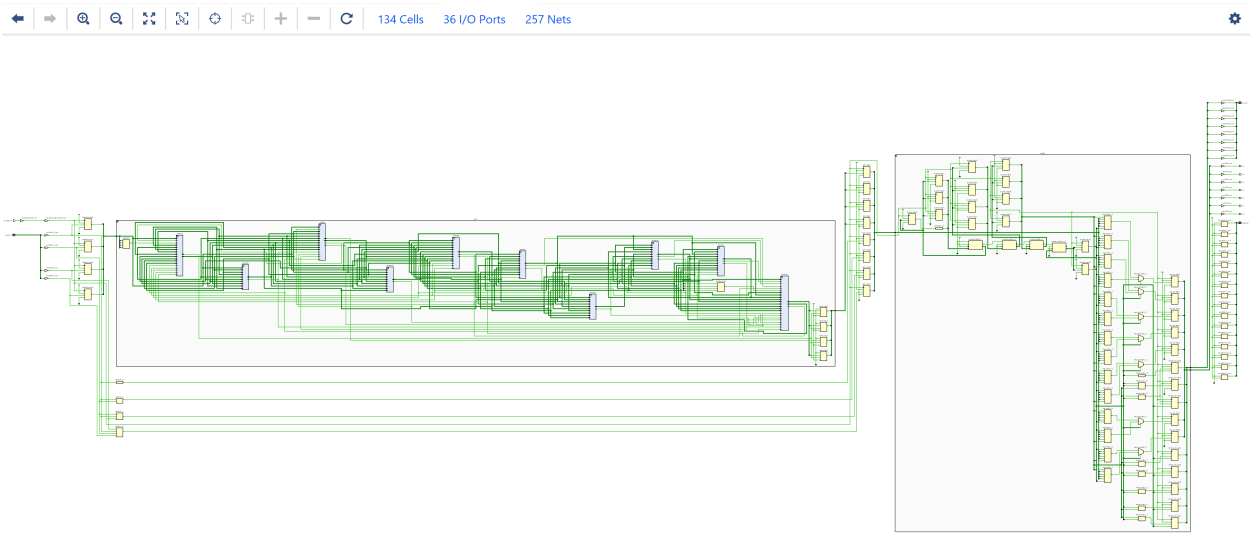


System Schematic: Condensed overview, combinational module + LED display

This diagram shows the condensed/collapsed overview of the combinational module. The input of 4 switches feeds into the module and controls which test digit is in use. The output feeds into the LED digit display for debugging.

The diagram below shows the same view, but expanded one level (sum module and digit module expanded). In the sum module, you can see the ten separate

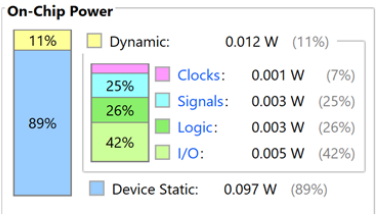
sub-modules that tally up the result for each digit, as well as the comparison signals that propagate through the series chain and result in a final arbitration to exit the module.



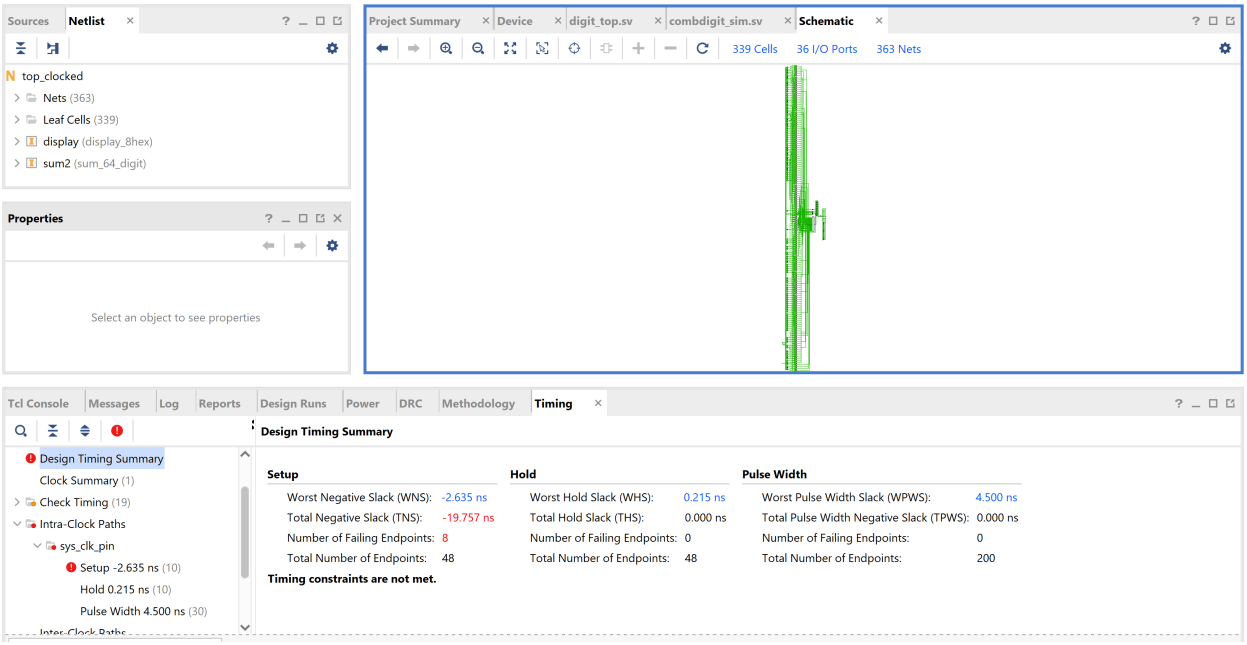
System Schematic: Expanded overview, combinational module + LED display

Because of its simplicity, this combinational module is also quite power-efficient, with a dynamic power consumption of only 0.012W.

WNS	TNS	WHS	THS	TPWS	Total Power	Failed Routes	LUT	FF	BRAM	URAM	DSP
							339	37	0.0	0	0
7.771	0.000	0.216	0.000	0.000	0.109	0	327	37	0.0	0	0

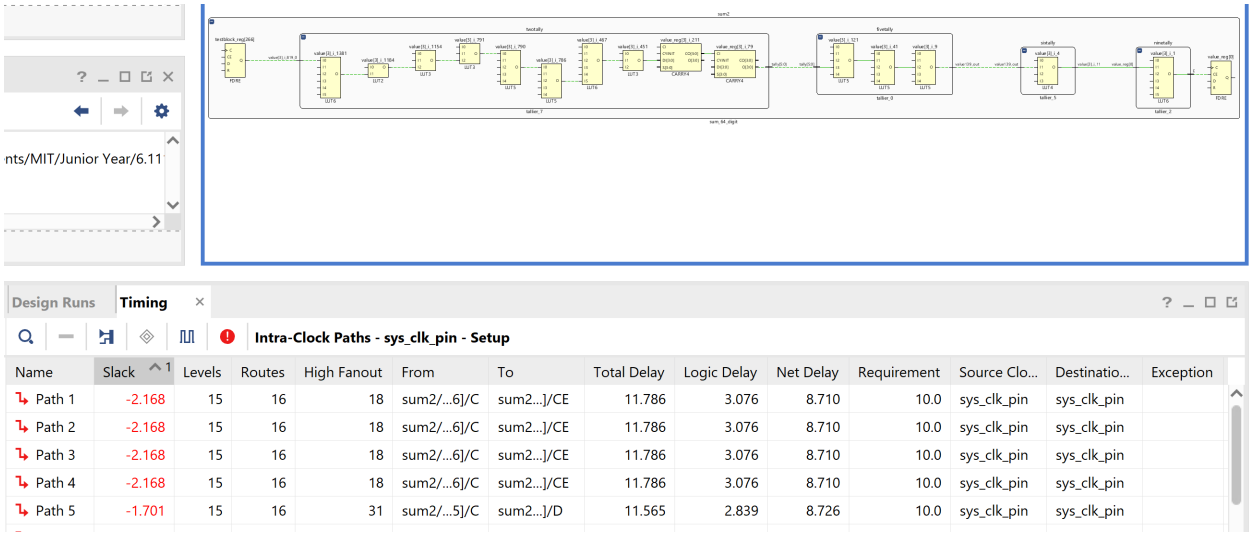


Resource Utilization: Combinational Module



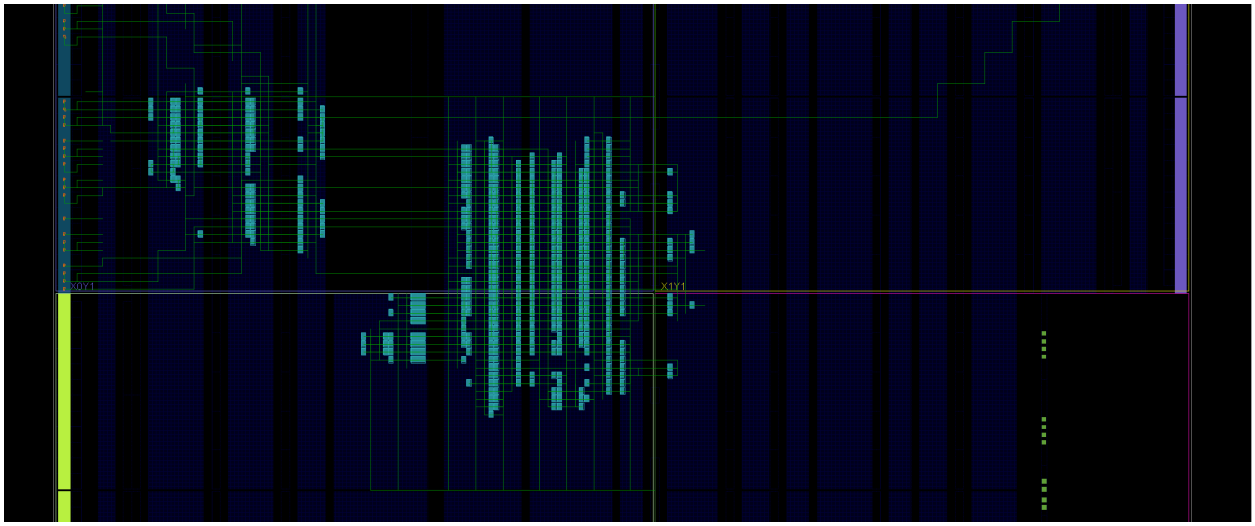
Signal Timing: Combinational Module

To estimate the signal propagation delay within the combinational module, I modified the top_level code so that the combinational module was clocked, with input changes and output queries happening within an always_ff loop. After Vivado optimized the design, the total negative slack within the module was measured to be ~20 nanoseconds. This was due to the long chain of digit-submodules that the signal needed to pass through, as shown in the figure below. The logic delays were small, but the total sum of all the circuit pathways necessary for these submodules to communicate resulted in a large net delay.



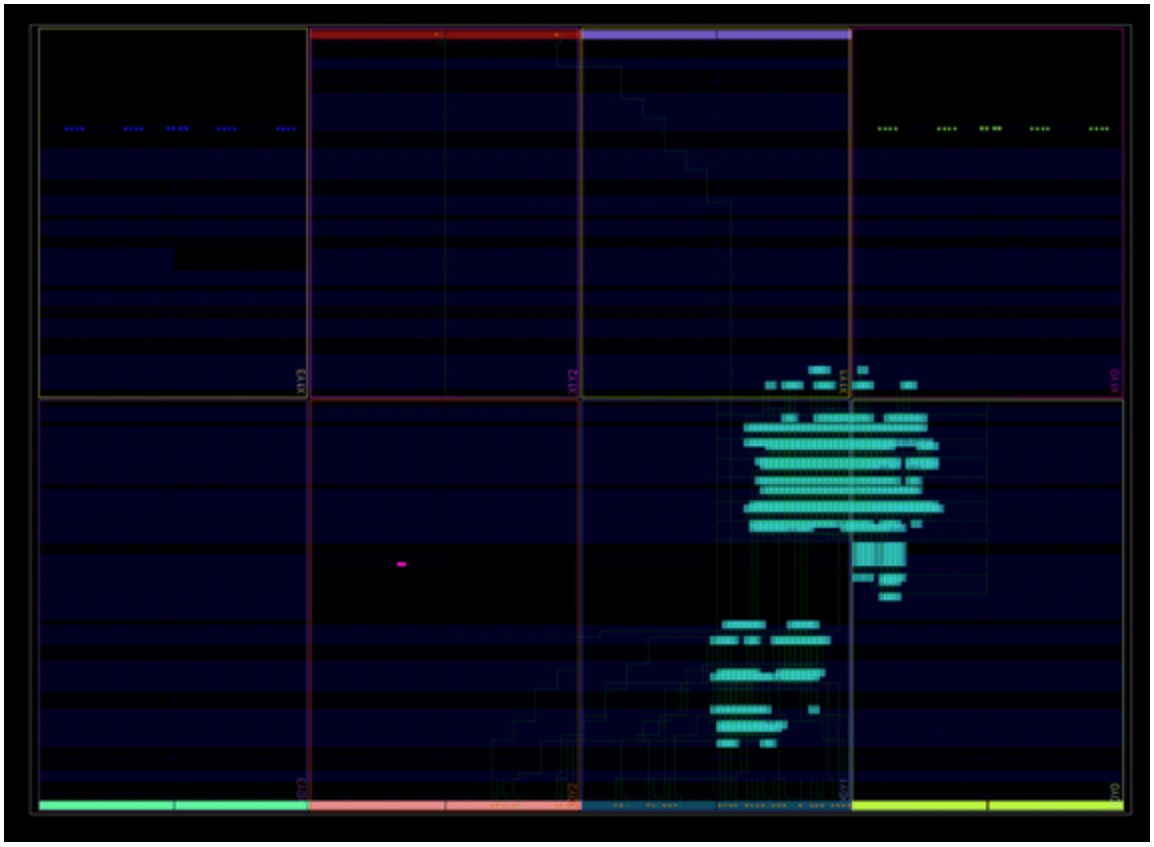
Name	Slack	Levels	Routes	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clo...	Destinatio...	Exception
Path 1	-2.168	15	16	18	sum2/...6]/C	sum2...]/CE	11.786	3.076	8.710	10.0	sys_clk_pin	sys_clk_pin	
Path 2	-2.168	15	16	18	sum2/...6]/C	sum2...]/CE	11.786	3.076	8.710	10.0	sys_clk_pin	sys_clk_pin	
Path 3	-2.168	15	16	18	sum2/...6]/C	sum2...]/CE	11.786	3.076	8.710	10.0	sys_clk_pin	sys_clk_pin	
Path 4	-2.168	15	16	18	sum2/...6]/C	sum2...]/CE	11.786	3.076	8.710	10.0	sys_clk_pin	sys_clk_pin	
Path 5	-1.701	15	16	31	sum2/...5]/C	sum2...]/D	11.565	2.839	8.726	10.0	sys_clk_pin	sys_clk_pin	

Signal Pathway: Combinational Module



Implementation Diagram: Combinational Module (left) and Arithmetic Module (right)

With both the combinational module and arithmetic module implemented, this is how Vivado optimized the on-chip design implementation. We can observe that the combinational module is smaller than the arithmetic-based model. This makes sense, because just storing the weights for the arithmetic-based model is going to take 8 times the amount of space (784 x 8-bit vs 784 x binary), and then larger register space is necessary to tally the sums.

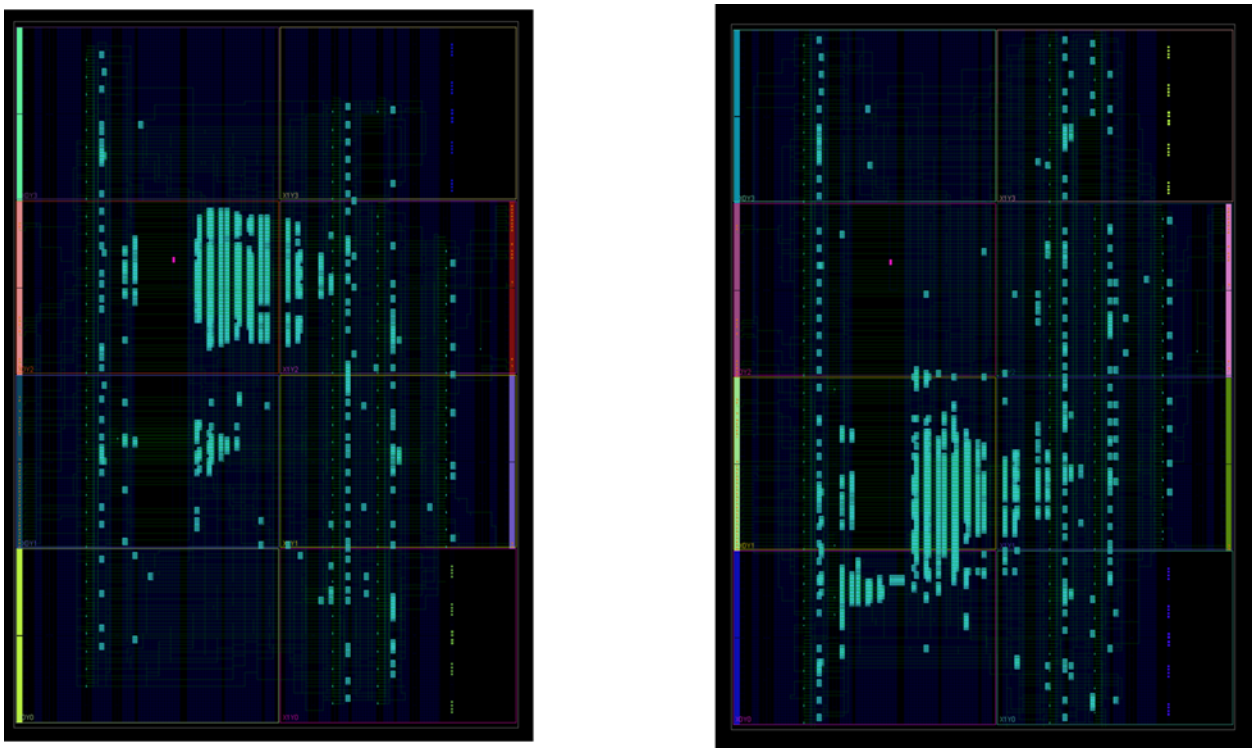


Implementation Diagram: Combinational Module (lower) and Arithmetic Module (upper)

We can quantify the difference with this resource utilization chart, which shows us that the arithmetic module uses 7.3x the Slice LUTs and 7.6x the Slices as the combinational module. In addition, the arithmetic module uses wide (F7/F8) muxes, while the combinational module does not, showing the relative simplicity of the comparison involved in the binarized network.

Name 1	Slice LUTs (63400)	Slice Registers (126800)	F7 Muxes (31700)	F8 Muxes (15850)	Slice (15850)	LUT as Logic (63400)	Bonded IOB (210)	BUFGCTRL (32)
N top_clocked	2674	475	362	120	808	2674	36	1
display (display_8hex)	28	29	0	0	14	28	0	0
neuron (densenet)	2267	288	362	120	700	2267	0	0
sum2 (sum_64_digit)	308	4	0	0	92	308	0	0

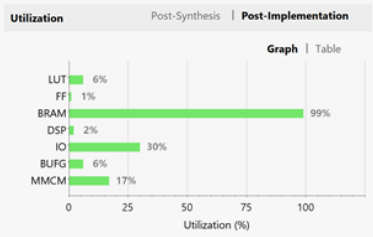
Resource Utilization: Combinational Module, Arithmetic Module, LED display

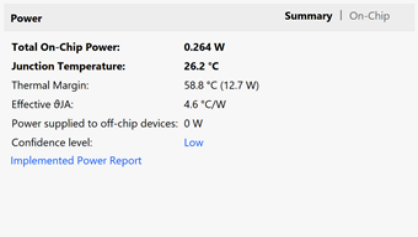


Implementation Diagrams: System without dynamic digit display (left); Full system with dynamic digit display (right)

As an interesting comparison, the addition of the dynamic digit displays (COE digit display of the predictions from the two modules, plus the MNIST digit display) not only increases BRAM usage scattered throughout the chip, but also changes the positioning of different components. As shown in the diagram below, the combination of various GUI components results in full utilization of the onboard BRAM, while my compact classifier modules use relatively light logic resources, as intended.

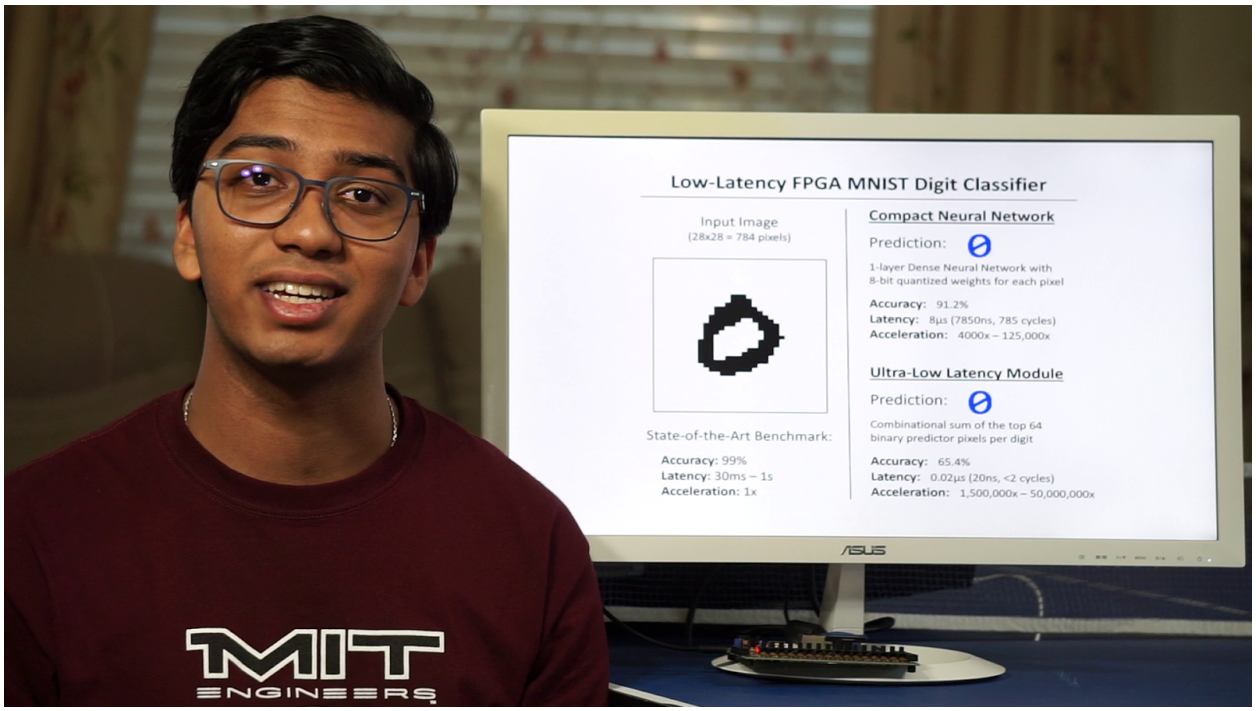
Resource	Utilization	Available	Utilization %
LUT	3724	63400	5.87
FF	590	126800	0.47
BRAM	133	135	98.52
DSP	4	240	1.67
IO	62	210	29.52
BUFG	2	32	6.25
MMCM	1	6	16.67





Resource Utilization: Final System (Combinational + Arithmetic + GUI)

8 Discussion: Results and Implications



Presentation Video: (link)

Reflecting on this project, I find it very interesting that this particular avenue of machine-learning optimization (marginal sacrifices to accuracy in favor of significant gains in speed, and forming these architectures in specialized hardware) has not been significantly studied beyond some industry-proprietary applications (Google's in-house TPU tensor-processing units) and generalized FPGA acceleration. Lots of research continues in machine learning optimization, as well as the use of FPGAs for low-latency tasks like high-frequency trading, but the intersection of the two has mostly been limited to acceleration rather than forming the entirety of the networks.

As far as challenges, lessons learned, and advice I would give to other students, I found it indispensable to create mockups of systems and algorithms both on paper (to work things out conceptually) and in other environments, like Python, before attempting to create them in Vivado for deployment on the FPGA. This allowed me to test and iterate designs in many cases before I discovered flaws in the FPGA approach,

and to improve flawed FPGA designs with more in-depth analysis based on large-scale simulations. As always, I also found it helpful to work in the Matlab-like environment of Spyder, where I could work with an exposed variable workspace and investigate tangible data structures, and effectively port data to include in the FPGA. If I were to to expand on this specific work, I would try to devise an intuitive way to represent convolutions in hardware - perhaps a register-focused variant enabled by shifting the registers to 'rotate' part of the input data, or having a chain of muxes form the convolution kernels - to create similarly high-performing variants of convolutional neural networks.

To recap on the results and some of the contents of the video summary: with the Arithmetic / Dense Neural Net module, it takes one clock cycle to sum each pixel, which outputs a result in under 8 microseconds. With just a 10% drop in accuracy, we can get a prediction system that is up to 4000x faster than conventional models. In addition, the network is substantially smaller than alternatives. A benchmark network may have around 303,000 weights, where each one is a 32-bit FLOP. Here, there are only 784 weights, each an 8-bit integer, which results in memory usage 1600x smaller than a comparable full network.

One interesting point to note is that my dense neural network implementation is especially compact because it loops through the different indices on different clock cycles, reducing the need for many concurrent lookups. Theoretically, all of the addition and comparison operations could be done in combinational logic, it would just involve a much wider space on-chip, and would likely not stabilize within one clock period. Crucially, however, that is not necessary in the case that digital data is being piped in to the FPGA (for example, from an external system) one bit at a time, as opposed to having the entirety of an image made present at once, as it is now. If that was the case, then the summations would happen in real-time as data was being passed in, and the result would be available instantaneously. This is an example of a case where the accuracy-latency tradeoff can be highly optimized, with highly accurate results available instantaneously after data transfer is completed.

My second module, the combinational ultra-low-latency module, involves no clock cycles. The direct signal propagation outputs a stable result in under 20 nanoseconds, as verified by Vivado signal timing analysis, which is a near-instantaneous result. With just a 30% drop in accuracy, we are able to achieve speeds 400x faster than the compact neural network and 1.5 million times faster than a competing, state-of-the-art system with 30ms latency. It also has a footprint only 1/10 of the combinational neural network, which results in multiple orders of magnitude space savings and resource conservation compared to conventional networks.

This combination of factors is remarkable because it showcases that we can push that performance envelope really effectively. With a minor tradeoff in accuracy, we achieve a system that is up to 6 orders of magnitude faster, and 5 orders of magnitude smaller, so it's more resource-efficient. This architecture and method of compressing networks can potentially be scaled and applied to a wider variety of progressively more complex tasks: the most common example is an autonomous vehicle, but even more complex cases are possible applications, such as safety systems in something like a chemical plant. Nothing happens instantaneously - even turning things off, it takes a moment to kick in. If you can detect that something is not going right - even with 70% accuracy, you can start to get those safety measures powered up within that time lag before you verify with 99% certainty.

Not only does this project have critical implications in a variety of use cases where low latency is crucial, it also demonstrates a significant advancement in terms of strategies for neural network construction and classification processing leveraging FPGA logic. We can see that competing MNIST classification systems take significantly longer: one set of privacy-preserving implementations take 2.2 seconds for a single prediction [Brutzkus, 2019] which was a significant improvement over previous privacy-preserving implementations, which took up to 205 seconds for a single prediction involving privacy-preserving operations [Dowlin, 2016]. While these are worst-case values arising from purposefully obfuscated algorithms, this does correspond to real-life observations. For example, a 2018 paper reported evaluation times between

7-12 seconds per image to evaluate various CNN models on MNIST [Palvanov and Cho, 2018].

In the high-performance regime, this system is still many orders of magnitude faster, achieving the low-latency objectives. An MIT-developed neural network displayed better performance than other privacy-preserving algorithms: 30ms latency [Juvekar, Vaikuntanathan, Chandrakasan; 2018]. Similar implementations are able to facilitate an online low-latency prediction-serving system with average latency of 47ms for MNIST digit recognition [Crankshaw, 2019]. However, even competing specialized FPGA systems do not perform significantly better than these systems. Specialized FPGA platforms emulating spiking neural networks achieve 20ms latencies for MNIST digits [Stromatias et al., 2015], so this system still achieves up to 6 orders of magnitude lower latency.

Notably, my system achieves significantly better overall results than the latest implementations of convolutional neural networks on similar FPGAs, in terms of accuracy, latency, power consumption, and resource utilization. Using the same FPGA as us, the Xilinx XC7A100T (Nexys4 DDR board), the paper "FPGA Implementation of Hand-written Number Recognition Based on CNN" [Giardino et al., 2019] achieved 90% accuracy with a 4-layer convolutional neural network that used a cumulative "29.69% of Slice LUTs, 4.42% of Slice Registers, and 52.5% of Block RAMs." By comparison, my Arithmetic / 1-Layer Dense Neural Network module was able to achieve 91.2% accuracy using only 3.58% of Slice LUTs, 0.23% of Slice Registers, and 0% of Block RAMs. This represents more than a 90% decrease in resource utilization, with no impact (in fact, a slight improvement) on classification performance. In addition, their implementation has a Vivado-estimated dynamic power consumption of 0.975W, while my entire system (combinational + arithmetic + GUI) has a dynamic power consumption of 0.160W (an 84% decrease) which further decreases to only 0.038W (a 96% decrease) if you remove the power required to re-clock for VGA output. Finally, their implementation requires 41 microseconds to classify an image, at a claimed clock frequency of 300MHz. This equates to a total of 12,300 clock cycles required to classify a single image. In the current implementation, my arithmetic/DNN module requires 785

clock cycles to classify a single image, which is a 94% reduction in latency (15x acceleration). This further supports the assessment of this approach as a novel contribution that can enable low-latency artificial intelligence algorithms while significantly improving hardware performance compared to state-of-the-art research.

9 References

Brelet, Jean-Louis et al. “Designing Flexible, Fast CAMs with Virtex Family FPGAs”. XILINX. September 23, 1999.

Brutzkus, A., Elisha, O., Gilad-Bachrach, R. “Low Latency Privacy Preserving Inference”. Proceedings of the 36th International Conference on Machine Learning. 2019.

Chen, F., Chen, N., Mao, H., Hu, H. "Assessing Four Neural Networks on Handwritten Digit Recognition Dataset (MNIST)". Chuangxinban Journal of Computing. June 2018.

Cherry, K. and Qian, L. "Scaling up molecular pattern recognition with DNA-based winner-take-all neural networks", Nature vol. 559, July 2018.

Dowlin, N. et al. "Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy." International Conference on Machine Learning, pp. 201–210, 2016.

Giardino, Daniele et al. "FPGA Implementation of Hand-written Number Recognition Based on CNN". International Journal on Advanced Science Engineering Information Technology, vol. 9, no. 1, pp. 167-171. 2019.

He, Kaiming et al. “Convolutional Neural Networks at Constrained Time Cost”. 2014.

Hubara et al, "Binarized Neural Networks", Conference on Neural Information Processing Systems (NIPS), 2016.

Juvekar, C., Vaikuntanathan, V., Chandrakasan, A. "GAZELLE: A Low Latency Framework for Secure Neural Network Inference". Proceedings of the 27th USENIX Conference on Security Symposium, pp. 1651-1668. August 2018.

Kim, M. and Smaragdis, P., "Bitwise Neural Networks", International Conference on Machine Learning, 2015.

Kouretas, I., Paliouras, V. "Hardware Implementation of a Softmax-Like Function for Deep Learning." Technologies, vol. 8, no. 46. MDPI, 2020.

Li, Hao et al. “Pruning Filters for Efficient ConvNets”. August, 2016.

Lin, X., Zhao, C., Pan, W., "Towards Accurate Binary Convolutional Neural Network", Conference on Neural Information Processing Systems, 2017.

Neelu et al, "Design and Implementation of Logic Gates and Adder Circuits on FPGA Using ANN", IJRASET vol. 4, May 2016.

Palvanov, A., Cho, Y.I. "Comparisons of Deep Learning Algorithms for MNIST in Real-Time Environment". International Journal of Fuzzy Logic and Intelligent Systems, vol. 18, no. 2, pp. 126-134. June 2018.

Qi, He et al. “An ultra-low-power FPGA for IoT applications”. 2017 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S), Burlingame, CA, 2017, pp. 1-3. IEEE, October 16-19, 2017.

Sander, Oliver et al. “Reducing latency times by accelerated routing mechanisms for an FPGA gateway in the automotive domain”. 2008 International Conference on Field-Programmable Technology, Taipei, 2008, pp. 97-104. IEEE, December 8-10, 2008.

Wielgosz, M. and Karwatowski, M. "Mapping Neural Networks to FPGA-based IoT Devices for Ultra-Low Latency Processing", Sensors vol. 19, 2019.

10 Appendix

Python Code:

MNISTconverter.py

```
# adapted from https://pjreddie.com/projects/mnist-in-csv/

def convert(imgf, labelf, outf, n):
    f = open(imgf, "rb")
    o = open(outf, "w")
    l = open(labelf, "rb")

    f.read(16)
    l.read(8)
    images = []

    for i in range(n):
        image = [ord(l.read(1))]
        for j in range(28*28):
            image.append(ord(f.read(1)))
        images.append(image)

    for image in images:
        o.write(",".join(str(pix) for pix in image)+"\n")
    f.close()
    o.close()
    l.close()

convert("train-images-idx3-ubyte", "train-labels-idx1-ubyte",
        "mnist_train.csv", 60000)
convert("t10k-images-idx3-ubyte", "t10k-labels-idx1-ubyte",
        "mnist_test.csv", 10000)
```

DigitHeatmaps.py

```
"""
Created on Sat Nov 21 16:16:16 2020
@author: Syamantak Payra

Overview:
    - parse through 1000 rows
    - put images into a dictionary, summing pixel values and count
    - parse through the dictionaries, creating weighted averages
    - process the weighted averages into images for visualization
"""


import csv
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import heapq

with open("data/mnist_train.csv") as csvfile:
    trainingdata = csv.reader(csvfile)
    count = 0
    limit = 49999

    digitdict = {0:{'count':0,'pixels':[]}}
    for digit in range(10):
        digitdict[digit] = {'count':0,'pixels':[0]*784}

    for row in trainingdata:
        if(count==limit):
            break
        else:
            count+=1
            # print(', '.join(row))
            # print([int(num) for num in list(row)])

            rowdata = [int(num) for num in list(row)]
            digit = rowdata[0]
```

```
            pixels = rowdata[1:]
            for index in range(len(pixels)):
                digitdict[digit]['pixels'][index]+=pixels[index]
            digitdict[digit]['count']+=1

fig, ax = plt.subplots()
sns.set_theme()
distributions = [{-1:1} for i in range(784)]
for digit in digitdict:

for simple heatmaps as export
    digitdict[digit]['array'] =
np.array(digitdict[digit]['pixels']).reshape((28,28))
    sns.heatmap(digitdict[digit]['array']) # ,cmap="Greys"
    plt.savefig(f"plots/{digit}.jpg")
    plt.clf()

# to create transparent square plots for overlays
    digitdict[digit]['scaled'] = [n//digitdict[digit]['count'] for n
in digitdict[digit]['pixels']]
    for index in range(len(digitdict[digit]['scaled'])):
        if digitdict[digit]['scaled'][index]<128:
digitdict[digit]['scaled'][index]=0
    digitdict[digit]['array'] =
np.array(digitdict[digit]['scaled']).reshape((28,28))

sns.heatmap(digitdict[digit]['array'],cmap="Greys",cbar=False,square=
True,xticklabels=False,yticklabels=False)
    plt.savefig(f"plots/{digit}.png",transparent=True)
    plt.clf()
    im = ax.imshow(digitdict[digit]['array'], color="Greys",
alpha=0.8)

# to create a weighted average
    for index in range(len(distributions)):
        distributions[index][digit]=digitdict[digit]['scaled'][index]

# to get the max 8 pixels for each digit
    digitdict[digit]['max8indices'] =
```

```
heapq.nlargest(64,range(784),digitdict[digit]['scaled'].__getitem__)
    digitdict[digit]['max8binary'] = [1 if index in
digitdict[digit]['max8indices'] else 0 for index in
range(len(digitdict[digit]['scaled']))]

    digitdict[digit]['array'] =
np.array(digitdict[digit]['max8binary']).reshape((28,28))
    sns.heatmap(digitdict[digit]['array']) # ,cmap="Greys"
    plt.savefig(f"plots/{digit}.jpg")
    plt.clf()


# to get the max digit at each pixel
distmaxes = [max(distdict,key=distdict.get) for distdict in
distributions]
# if you want to put a threshold on the maxes
for index in range(len(distmaxes)):
    if distributions[index][distmaxes[index]]<180:
distmaxes[index]=-1


# to create a plot of the different digits and their combinational
blobs

counts = [count(distmaxes,digit) for digit in digitdict]
masks = np.array([True if cell==-1 else False for cell in
distmaxes]).reshape((28,28))
sns.heatmap(np.array(distmaxes).reshape((28,28)),annot=True,mask=mask
s)
plt.savefig("plots/combinational.jpg",dpi=200)


# to output the binary references for the combinational blobs based
on above conditions
binaryfilters = dict()
for digit in digitdict:
    # digitdict[digit]['arbitrarymaxes'] = [1 if pixel==digit else 0
for pixel in distmaxes]
    binaryfilters[digit]="".join(["1" if pixel==digit else "0" for
```

```
pixel in distmaxes])

# to output the binary references for the 64 binary blobs
for digit in digitdict:
    binaryfilters[digit]="".join(["1" if pixel==1 else "0" for pixel
in digitdict[digit]['max8binary']])



def count(container, obj):
    count = 0
    for i in container:
        if i==obj:
            count+=1
    return(count)


def binary_tests(limit,threshold):
    with open("data/mnist_test.csv") as csvfile:
        testdata = csv.reader(csvfile)
        count = 0
        output = []

        for row in testdata:
            if(count==limit):
                break
            else:
                count+=1
                # print(', '.join(row))
                # print([int(num) for num in list(row)])

                binary_pixels = ["1" if int(num)>threshold else "0"
for num in list(row)[1:]]

output.append([int(list(row)[0]),"".join(binary_pixels)])
    return(output)

def combinationaltally():
```

```
    print("assign tally = val_in[0]"+"".join([f"+val_in[{index}]" for
index in range(1,784)]))


def test_binarymaxes(firstn,indices=[]):
    with open("data/mnist_test.csv") as csvfile:
        testdata = csv.reader(csvfile)
        count = 0
        testlist = []
        successes = 0

        for row in testdata:
            if(count==firstn):
                break
            else:
                count+=1
                # print(', '.join(row))
                # print([int(num) for num in list(row)])

                binary_pixels = ["1" if int(num)>100 else "0" for num
in list(row)[1:]]

testlist.append([int(list(row)[0]),"".join(binary_pixels)])

        for test in testlist:
            actual = test[0]
            tallies = []
            for digit in digitdict:
                #
tallies.append(sum([digitdict[digit]['arbitrarymaxes'][index]&int(tes
t[1][index]) for index in range(len(test[1]))]))
                #
tallies.append(sum([digitdict[digit]['max8binary'][index]&int(test[1]
[index]) for index in range(len(test[1]))]))

tallies.append(sum([indices[digit][index]&int(test[1][index]) for
index in range(len(test[1]))])) # for the indices[] list

            largest =
```

```
heapq.nlargest(1,range(len(tallies)),tallies.__getitem__)[0]
            # print(actual, largest)
            if largest==actual: successes+=1

    return(successes/count)

def plot_binary_accuracies():
    points = [1,2,4,8,16,32,64,128,192,256,320,384,512]
    stats = []
    for point in points:
        indices = []
        for digit in digitdict:
            maxes =
heapq.nlargest(point,range(784),digitdict[digit]['scaled'].__getitem_
_)
            indices.append([1 if index in maxes else 0 for index in
range(len(digitdict[digit]['scaled']))])

        stats.append(test_binarymaxes(4999,indices))

    plt.figure()
    plt.suptitle("Diminishing Returns from Binary Matchup")
    plt.xlabel("Pixels")
    plt.ylabel("Classification Accuracy")
    plt.plot(points,stats,marker='+',mec='r',linestyle='solid')
    plt.savefig("plots/binaryreturns.jpg",dpi=200)

def make_binary_datasets(threshold=128):

    with open("data/mnist_train.csv") as csvfile:
        trainingdata = csv.reader(csvfile)
        count = 0
        limit = 49999

        allrows = []
        for row in trainingdata:
            if(count==limit):
                break
            else:
```

```
                count+=1

                rowdata = [int(num) for num in list(row)]
                digit = [rowdata[0]]
                pixels = rowdata[1:]
                binaries = []
                for index in range(len(pixels)):
                    if pixels[index] > threshold:
                        binaries.append(1)
                    else:
                        binaries.append(0)
                allrows.append(digit+binaries)

    with open("data/mnist_binary_train.csv","w",newline='') as
csvfile:
        spamwriter = csv.writer(csvfile, delimiter=',')
        spamwriter.writerows(allrows)

    with open("data/mnist_test.csv") as csvfile:
        trainingdata = csv.reader(csvfile)
        count = 0
        limit = 9999

        allrows = []
        for row in trainingdata:
            if(count==limit):
                break
            else:
                count+=1

                rowdata = [int(num) for num in list(row)]
                # digit = [rowdata[0]]
                pixels = rowdata[1:]
                binaries = []
                for index in range(len(pixels)):
                    if pixels[index] > threshold:
                        binaries.append(1)
                    else:
                        binaries.append(0)
```

```
                allrows.append(binaries)

    with open("data/mnist_binary_test.csv","w",newline='') as
csvfile:
        spamwriter = csv.writer(csvfile, delimiter=',')
        spamwriter.writerows(allrows)

def make_binary_dataset_combo(threshold=128):

    with open("data/mnist_train.csv") as csvfile:
        trainingdata = csv.reader(csvfile)
        count = 0
        limit = 19999

        allrows = []
        for row in trainingdata:
            if(count==limit):
                break
            else:
                count+=1

                rowdata = [int(num) for num in list(row)]
                digit = [rowdata[0]]
                pixels = rowdata[1:]
                binaries = []
                for index in range(len(pixels)):
                    if pixels[index] > threshold:
                        binaries.append(1)
                    else:
                        binaries.append(0)
                allrows.append(digit+binaries)

    with open("data/mnist_test.csv") as csvfile:
        trainingdata = csv.reader(csvfile)
        count = 0
        limit = 0

        for row in trainingdata:
            if(count==limit):
```

```
                break
            else:
                count+=1

                rowdata = [int(num) for num in list(row)]
                digit = [rowdata[0]]
                pixels = rowdata[1:]
                binaries = []
                for index in range(len(pixels)):
                    if pixels[index] > threshold:
                        binaries.append(1)
                    else:
                        binaries.append(0)
                allrows.append(digit+binaries)

    with open("data/mnist_binary_combo2.csv","w",newline='') as
csvfile:
        spamwriter = csv.writer(csvfile, delimiter=',')
        header = ["label"]+[f"pixel{dec}" for dec in range(784)]
        spamwriter.writerow(header)
        spamwriter.writerows(allrows)
```

tinyCNN_v3.py

```
# Syamantak Payra
# December 2020

import os
os.environ['KERAS_BACKEND']='tensorflow'
import pandas as pd
import numpy as np
import tensorflow as tf
# sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))

from keras.layers import Convolution2D, Activation, MaxPooling2D,
Flatten, Dropout, Dense
```

```
from keras.utils import np_utils
from keras.models import Sequential
from keras.optimizers import SGD
import matplotlib.pyplot as plt
import seaborn as sns

def tf_tutorial_model(dropout=0.4, dense_size=0, lr=0.001):
    model = tf.keras.Sequential()
    # # first convolutional layer:
    # model.add(Convolution2D(32, (5,5), batch_input_shape=(None, 1,
28, 28),
    #     padding='same', data_format='channels_first'))
    # # model.add(Activation('relu'))
    # model.add(MaxPooling2D(pool_size=2, strides=2, padding='same',
data_format='channels_first'))
    # # second convolutional layer:
    # model.add(Convolution2D(64, (5,5), padding='same',
data_format='channels_first'))
    # model.add(Activation('relu'))
    # model.add(MaxPooling2D(pool_size=2, strides=2, padding='same',
data_format='channels_first'))
    # first dense layer:
    model.add(tf.keras.layers.Flatten())
    # if dense_size > 0:
    #     model.add(Dense(dense_size))
    #     model.add(Activation('relu'))
    #     if dropout > 0:
    #         model.add(Dropout(dropout))
    # final (dense) layer:
    model.add(tf.keras.layers.Dense(10))
    model.add(tf.keras.layers.Activation('softmax'))

    sgd = tf.keras.optimizers.SGD(lr=lr)
    model.compile(optimizer=sgd, loss='categorical_crossentropy',
metrics=['accuracy'])
    return model

# little plotter helper to see results. various dense layer data is
plotted simultaneously against the epochs.
```

```
def plotxy(hist, title="tf-mnist", ytitle="accuracy", xtitle="epoch",
legend=()):
    fig = plt.figure(figsize=(15, 10)) #change dpi to get bigger
resolution..
    for h in hist:
        plt.plot(h)
    plt.ylabel(ytitle)
    plt.xlabel(xtitle)
    plt.legend(legend, loc='lower right')
    plt.title(title)
    plt.show()
    plt.close(fig)

# this is just to pull data off the history object. Look at keras
documentation for history. It gives back data in columns, and their
labels.
def get_data_n_legend(hist, id):
    plot_data = []
    legend = []
    for d in sorted(hist.keys()):
        plot_data.append(hist[d].history[id])
        legend.append('d-' + str(d))  # legends e.g. d-128, d-256
denote the dense layer size.
    return((plot_data, legend))

print(os.listdir("data"))
train_data = pd.read_csv('data/mnist_binary_combo.csv')
x_train = train_data.drop('label',axis=1).values.reshape(-1,1,28,28)
y_train = np_utils.to_categorical(train_data['label'])

hist = {}
# for dense_units in (0): # (0,128,1024)
dense_units=0
lr = 0.00075   # lr=0.001 could be too high for dense_units = 0
modl = tf_tutorial_model(dense_size=dense_units, lr=lr)
print("Training for dense layer of size:", dense_units, "lr=", lr)
hist[dense_units] = modl.fit(x_train, y_train, epochs=48,
batch_size=32, validation_split=0.1) #0.5
```

```
id = 'val_accuracy'
(pl_data,legnd) = get_data_n_legend(hist, id)
plotxy(pl_data, title="tf-mnist & dense layer size", xtitle='epoch',
legend=legnd, ytitle=id)

fmt_str='{0:11} :   {1:6}'
print(fmt_str.format('Dense layer', id))
for d in sorted(hist.keys()):
    print(fmt_str.format('d-' + str(d),
round(hist[d].history[id][-1], 3)))


def dense_heatmap(flattenweights):
    fig, ax = plt.subplots()
    sns.set_theme()
    weights_transposed = np.transpose(flattenweights)
    heatmaps = {}
    for digit in range(10):
        heatmaps[digit] =
np.array(weights_transposed[digit]).reshape((28,28))
        sns.heatmap(heatmaps[digit], cmap='RdBu')
        plt.savefig(f"plots/{digit}.jpg", dpi=200)
        plt.clf()

def testOne():
    biases_transposed = np.transpose(flattenbiases)

    test1 =
np.array([0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

```
,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0])
    test2 =
np.array([0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1
,1,1,1,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,1,
1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,1,1,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,
1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,1,1,1,1,1,0,0,
0,0,0,0,0,1,1,1,1,0,0,0,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1
,0,0,0,0,1,1,1,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,1,1,0,0,0,1,1,1,0,
0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0])

    matches_1 = [sum(np.multiply(test1,weights_transposed[digit]))
for digit in range(10)]
```

```
    matches_2 = [sum(np.multiply(test2,weights_transposed[digit]))
for digit in range(10)]

    matches_1b = matches_1+biases_transposed
    matches_2b = matches_2+biases_transposed

def quantize8bit():
    converter = tf.lite.TFLiteConverter.from_saved_model("models/v4")
    converter.optimizations = [tf.lite.Optimize.DEFAULT]
    # def representative_dataset_gen():
    #   for _ in range(num_calibration_steps):
    #     # Get sample input data as a numpy array in a method of
your choosing.
    #     yield [input]
    converter.representative_dataset = test1
#representative_dataset_gen
    converter.target_spec.supported_ops =
[tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
    converter.inference_input_type = tf.int8  # or tf.uint8
    converter.inference_output_type = tf.int8  # or tf.uint8
    tflite_quant_model = converter.convert()
    with open("models/quantized.tflite","wb") as f:
        f.write(tflite_quant_model)

def interpretQuantized():
    '''
    Create interpreter, allocate tensors
    '''
    tflite_interpreter =
tf.lite.Interpreter(model_path='models/quantized.tflite')
    tflite_interpreter.allocate_tensors()

    '''
    Check input/output details
    '''
    input_details = tflite_interpreter.get_input_details()
    output_details = tflite_interpreter.get_output_details()

    print("== Input details ==")
```

```
    print("name:", input_details[0]['name'])
    print("shape:", input_details[0]['shape'])
    print("type:", input_details[0]['dtype'])
    print("\n== Output details ==")
    print("name:", output_details[0]['name'])
    print("shape:", output_details[0]['shape'])
    print("type:", output_details[0]['dtype'])

    '''
    Run prediction (optional), input_array has input's shape and
dtype
    '''
    tflite_interpreter.set_tensor(input_details[0]['index'],
input_array)
    tflite_interpreter.invoke()
    output_array =
tflite_interpreter.get_tensor(output_details[0]['index'])

    '''
    This gives a list of dictionaries.
    '''
    tensor_details = tflite_interpreter.get_tensor_details()

    for dict in tensor_details:
        i = dict['index']
        tensor_name = dict['name']
        scales = dict['quantization_parameters']['scales']
        zero_points = dict['quantization_parameters']['zero_points']
        tensor = tflite_interpreter.tensor(i)()

        print(i, type, name, scales.shape, zero_points.shape,
tensor.shape)

        '''
        See note below
        '''


def testQuantized():
```

```
    biases_transposed = np.transpose(flattenbiases)

    test1 =
np.array([0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0])
    test2 =
np.array([0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1
,1,1,1,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,1,
1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,0,0,0,0,1,1,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,
```

```
1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,1,1,1,1,0,0,0,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,1,1,1
,0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,1,0,0,0,0,0,0,0,0,1,1,0,0,0,1,1,1,0,
0,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0])

    q_matches_1 = [sum(np.multiply(test1,quantized_weights[digit]))
for digit in range(10)]
    q_matches_2 = [sum(np.multiply(test2,quantized_weights[digit]))
for digit in range(10)]

def weightImports():
    weights = [list(quantized_weights[i]) for i in range(10)]
    weights_reversed = [list(sublist) for sublist in weights]
    for sublist in weights_reversed:
        sublist.reverse()
```

FPGA Code:

digit_top.sv

```
//module comb_digit(
//    sw,
//    led
//    );

//    input logic[15:12] sw;
//    output logic[15:0] led;
```

```
//    logic [783:0] testblock;
//    logic [783:0] allzeroes =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000;

//    logic [783:0] testzero =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000011100000
0000000000000000001111000000000000000000000001111100000000000000
0000111111110000000000000000001111111100000000000000001111111111
1000000000000000011111001111000000000000001111100001111000000000
0000111100000111100000000000011100000001110000000000011000000000
0011110000000000011100000001111000000000001110000000111100000000
0000011100000111110000000000011100001111110000000000011001110000
1111100000000000001111111111110000000000001111111110000000000000
0000000011111111000000000000000000011100000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000;
//    logic [783:0] testone =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000001100
0000000000000000000001000000000000000000000001100000000000000000
0000000001100000000000000000000001000000000000000000000001100000
0000000000000000000001100000000000000000000001100000000000000000
0000000001100000000000000000000001100000000000000000000001110000
0000000000000000000001100000000000000000000001100000000000000000
0000000001110000000000000000000001110000000000000000000000110000
0000000000000000000001100000000000000000000001110000000000000000
0000000000111000000000000000000001100000000000000000000000000000
```

```
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
//    logic [783:0] testtwo =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000001111100000000000000000000111111100000
0000000000000011111111110000000000000000011100001110000000000000
0011000011100000000000000000000000111000000000000000000000111100
0000000000000000000011100000000000000000000011110000000000000000
0000001111000000000000000000000011100000000000000000000001111000
0000000000000000000011100000000000000000000011110000000000000000
0000011110000000000000000000000011100000000000000000000001110000
0000001111000000000011111111111111111100000000111111111111111100
0000000011111111111000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
//    logic [783:0] testthree =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000110000000000
0000000000001111110000000000000000001111111100000000000000000001
1111111100000000000000000111000011110000000000000011000001111000
0000000000000110000001110000000000000011000100011110000000000000
0010011111111100000000000000000011111111110000000000000000111111
1111100000000000000000000011111000000000000000000000001111000000
0000001100000000001110000000000011100000000011100000000000111000
0000001110000000000011110000111111000000000000011111111111110000
0000000000001111111110000000000000000001010000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
//    logic [783:0] testfour =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000100000000000000000000011000000110000000000000000
0000110000000100000000000000011000000110000000000000011000000000
1100000000000000110000000011000000000000011000000011100000000000
0001110000000011000000000000011000000011100000000000001100000000
1110000000000000110000000111000000000000111000011111000000000000
0000011111111111000000000000000000000011100000000000000000000000
0111000000000000000000000011100000000000000000000011100000000000
0000000000000001110000000000000000000011100000000000000000000000
```

```
0001000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000;
//    logic [783:0] testfive =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0100000000000000000001111111110000000000000001111111111100000000
0000001111111111110000000000100011111000000000000000011000000000
0000000000000001110000000000000000000011100000000000000000000000
0011000000000000000000000001110000000000000000000000011111000000
0000000000000001111110100000000000000000011111111111110000000000
0000011111111111110000000000000000011111111000000000000000001111
0111100000000000000000011111111000000000000000000011111110000000
0000000000001111100000000000000000000010000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000;
//    logic [783:0] testsix =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000001110000000000000000000000011110000000
0000000000000001110000000000000000000001110000000000000000000000
0001100000000000000000000000011000000000000000000000011100000000
0000000000000000110000011111110000000000000110000011111111000000
0001110001111100011000000000001110011111000011000000000001110011100
0000110000000000011101110000000110000000000011101110000000110000000
0000011100111000001100000000000111000111000111000000000000011100000
0011110000000000000111000111110000000000000000011111111100000000000
0000000001111110000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000;
//    logic [783:0] testseven =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0111000000000000000000000011111111111110000000000001011111111111
1110000000000000000000000001110000000000000000000000011000000000000
0000000000000110000000000000000000000111000000000000000000000000000
1100000000000000000000000011100000000000000000000000110000000000000
0000000000001110000000000000000000000110000000000000000000000001
1100000000000000000000000111000000000000000000000001100000000000000
0000000000011100000000000000000000000111000000000000000000000011
```

```
1000000000000000000000001111000000000000000000000001110000000000000
0000000000000000000000000000;
//    logic [783:0] testeight =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000001111111110000000000000
0000011111111111100000000000001111110001111000000000000111100000
0011100000000000011100000000011100000000000011110000001111000000000
0000011100000111110000000000000111000111110000000000000000011111
1110000000000000000011111100000000000000000001111111100000000000000
0000111111101110000000000001111110001110000000000000011111011111
1100000000000000011111111110000000000000000011111111000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000;
//    logic [783:0] testnine =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000110000000000000000000000
0000011100000000000000000000011111000000000000000000001111111100
0000000000000000011111111000000000000000000011000111100000000000000
0000010000111100000000000000001110001111000000000000000011001111
1000000000000000000111111011100000000000000000011110011100000000000
0000000001100001110000000000000000000000011000000000000000000000
0001110000000000000000000000000110000000000000000000000111000000
0000000000000000000110000000000000000000000011000000000000000000
0000000011000000000000000000000000010000000000000000000000000000
0000000000000000000000000000;

//    logic [783:0] zeroblock =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000011000000000000000000001111100000000000000
0000000100111110000000000000000000000011110000000000000000000000
0111000000000000011111000000111000000000000111000000000111000000000
0001110000000000111000000000011000000000001110000000000110000000000
0001110000000000111000000000011100000000001110000000000111100000000
0000111000000000111000000000011100000000001110000000000111000000000
0000110000000000011110000000110000000000000011111100100000000000
```

```
0000000011111100000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000;
//    logic [783:0] oneblock =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000011000000000000000000000011000000000000000000
0000000001100000000000000000000000011000000000000000000000000100
0000000000000000000001000000000000000000000011000000000000000000
0000000000110000000000000000000000010000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000;
//    logic [783:0] twoblock =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000011100000
0000000000000000000111000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000001000000000000000000000010100000000000000000000100000
0000000000000000000010000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000;
//    logic [783:0] threeblock =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000111000000000000000000011111110000000000000000000
0001110000000000000000000000000000000000000000000000000000000000
0000000000000000000010000000000000000000100100000000000000000000
0000000010010000000000000000000011000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000010000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000111000000000000
```

```
00000000000001111000000000000000000011000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
//    logic [783:0] fourblock =
784'b00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
//    logic [783:0] fiveblock =
784'b00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
//    logic [783:0] sixblock =
784'b00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
```

```
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000;
//    logic [783:0] sevenblock =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000011110011110000000000000000111111111111
1000000000000000100000001110000000000000000000000000111000000000000
0000000000000010000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000001000000000000000000000001000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000;
//    logic [783:0] eightblock =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000010000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000011010
0000000000000000000011000000000000000000000010000000000000000
0000000010000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000001111100000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000;
//    logic [783:0] nineblock =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000011000000000000000000000011000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000001100000000000000000000111000000000000000000000000001
1000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000001000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
```

```
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000;

//    always_comb begin
//        led[15:12] = sw[15:12];
//        case(sw)
//            default: begin
//                testblock = allzeroes;
////                led[9:0] = 10'b0000000000;
//            end
//            4'b0001: begin
//                testblock = testzero;
//            end
//            4'b0010: begin
//                testblock = testone;
//            end
//            4'b0011: begin
//                testblock = testtwo;
//            end
//            4'b0100: begin
//                testblock = testthree;
//            end
//            4'b0101: begin
//                testblock = testfour;
//            end
//            4'b0110: begin
//                testblock = testfive;
//            end
//            4'b0111: begin
//                testblock = testsix;
//            end
//            4'b1000: begin
//                testblock = testseven;
//            end
//            4'b1001: begin
//                testblock = testeight;
//            end
//            4'b1010: begin
```

```
//                testblock = testnine;
//            end
//        endcase

//        led[0] = (testblock&zeroblock)!=0;
//        led[1] = (testblock&oneblock)!=0;
//        led[2] = (testblock&twoblock)!=0;
//        led[3] = (testblock&threeblock)!=0;
//        led[4] = (testblock&fourblock)!=0;
//        led[5] = (testblock&fiveblock)!=0;
//        led[6] = (testblock&sixblock)!=0;
//        led[7] = (testblock&sevenblock)!=0;
//        led[8] = (testblock&eightblock)!=0;
//        led[9] = (testblock&nineblock)!=0;

//    end

//endmodule

module top_clocked(
    input clk_100mhz,
    input [15:0] sw,
    input btnc,
    output logic[3:0] vga_r,
    output logic[3:0] vga_b,
    output logic[3:0] vga_g,
    output logic vga_hs,
    output logic vga_vs,
    output logic ca,cb,cc,cd,ce,cf,cg,
    output logic [7:0] an,
    output logic [15:0] led
);

    // create 65mhz system clock, happens to match 1024 x 768 XVGA
timing
    clk_wiz_lab3 clkdivider(.clk_in1(clk_100mhz),
.clk_out1(clk_65mhz));

    logic [10:0] hcount;    // pixel on current line
```

```
    logic [9:0] vcount;     // line number
    logic hsync, vsync;
    logic [11:0] pixel;
    logic [11:0] rgb;

    xvga
xvga1(.vclock_in(clk_65mhz),.hcount_out(hcount),.vcount_out(vcount),
          .hsync_out(hsync),.vsync_out(vsync),.blank_out(blank));

    // the following lines are required for the Nexys4 VGA circuit -
do not change
    assign vga_r = ~b ? rgb[11:8]: 0;
    assign vga_g = ~b ? rgb[7:4] : 0;
    assign vga_b = ~b ? rgb[3:0] : 0;

    assign vga_hs = ~hs;
    assign vga_vs = ~vs;

    logic up,down;
    logic b,hs,vs;
    logic phsync,pvsync,pblank;
    pong_game pg(.vclock_in(clk_65mhz),.reset_in(reset),
                 .up_in(up),.down_in(down),.pspeed_in(sw[15:12]),

.valuecalc(valuecalc),.value64(value64),.testblock(testblock),
                 .hcount_in(hcount),.vcount_in(vcount),
                 .hsync_in(hsync),.vsync_in(vsync),.blank_in(blank),

.phsync_out(phsync),.pvsync_out(pvsync),.pblank_out(pblank),.pixel_ou
t(pixel));

    always_ff @(posedge clk_65mhz) begin
         // default: pong
         hs <= phsync;
         vs <= pvsync;
         b <= pblank;
         rgb <= pixel;
    end
```

```
    logic [3:0] actual, valuearb, value64, valuecalc;
//    sum_comb_digit sum1(.sw(sw), .value(valuearb));
    sum_64_digit sum2(.clk_in(clk_65mhz), .sw(sw), .value(value64));

    logic reset = 0; logic [3:0] prevsw = 4'b0000;
    densenet neuron(.clk_in(clk_65mhz), .rst_in(reset),
.val_in(testblock), .outputval(valuecalc));

//    comb_digit sum0(.sw(sw), .led(led));

    logic [31:0] data;      //  instantiate 7-segment display;
display (8) 4-bit hex
    logic [6:0] segments;
    logic [783:0] testblock;
    assign {cg, cf, ce, cd, cc, cb, ca} = segments[6:0];
    display_8hex display(.clk_in(clk_65mhz), .data_in(data),
.seg_out(segments), .strobe_out(an));

    assign actual = sw-1'b1;

    logic [783:0] testzero =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000011100000
0000000000000000001111000000000000000000001111000000000000000000000
0000111111110000000000000000001111111110000000000000000001111111111
1000000000000000011111001111000000000000001111100001111000000000000
0000111100000011110000000000001110000000111000000000000110000000000
0011110000000000011100000001111000000000001110000000111100000000000
0000011100000011111000000000001110000111111000000000000011100111000
1111100000000000001111111111110000000000000011111111110000000000000
0000000011111111000000000000000000111000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000;
    logic [783:0] testone =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000001100000
0000000000000000000000100000000000000000000000001100000000000000000
0000000001100000000000000000000000100000000000000000000000001100000
0000000000000000000000110000000000000000000000001100000000000000000
```

```
0000000001100000000000000000000000011000000000000000000000001110
0000000000000000000000011000000000000000000000001100000000000000
0000000001110000000000000000000000111000000000000000000000000110
0000000000000000000000011000000000000000000000001110000000000000
0000000000111000000000000000000000110000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
//    logic [783:0] testtwo =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000111111000000000000000000011111110000
0000000000000011111111110000000000000000011100011100000000000000
0011000011100000000000000000000000011100000000000000000000111100
0000000000000000000011100000000000000000000011110000000000000000
0000001111000000000000000000000011100000000000000000000001111000
0000000000000000000011100000000000000000000011110000000000000000
0000001111000000000011111111111111111100000000111111111111111100
0000000011111111110000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
    logic [783:0] testtwo =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000001111110
0000000000000000111111111110000000000000011111000011000000000000
0001110000000011000000000000011100000000110000000000001100000000
0110000000000001000000000000110000000000000000000000001100000000
0000000000000011000000000000000000000111000000000000000000000000
1110000000000000000000000011100000000000000000000001110000000000
0000000000011100000000000000000011111000000000000000001111111
1100000100000000001111111111111010000000001111111000111111000000
0000011111100000011000000000000011000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
//    logic [783:0] testthree =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000110000000000
0000000000001111110000000000000000001111111100000000000000000001
1111111100000000000000000111000011110000000000000011000000111100
0000000000000110000000111000000000000000011000100011110000000000
```

```
00100111111111000000000000000000011111111110000000000000000001111111
11111000000000000000000000000011111000000000000000000000000001111000000
00000001100000000001110000000000011100000000011100000000000111000
00000001110000000000011111000011111100000000000011111111111110000
00000000000011111111110000000000000000010100000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
    logic [783:0] testthree =
784'b00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000011110000000
00000000000000011111000000000000000000001111111000000000000000000
00100011100000000000000000000000011000000000000000000000000110000
00000000000000000000110000000000000000000000111000000000000000000
00000001110000000000000000000000011111000000000000000000000011111
10000000000000000000000011110000000000000000000000001110000000000
00000000000001110000000000000000000011110000000000000000000001
11100000000000000000001111100000000000000000000011111000000000000
00000000011111000000000000000000000110000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
    logic [783:0] testfour =
784'b00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000100000000000000000000011000000110000000000000
00001100000000100000000000000011000000110000000000000011000000000
11000000000000001100000000110000000000000011000000011100000000000
00011100000000110000000000000011000000011100000000000000110000000
11100000000000000110000000011100000000000001110000111111000000000
00000111111111110000000000000000000001110000000000000000000000000
01110000000000000000000000011100000000000000000000011100000000
00000000000000011100000000000000000001110000000000000000000000
00010000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
//    logic [783:0] testfive =
784'b00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
01000000000000000000011111111100000000000000011111111111000000000
00000011111111111100000000000100011111000000000000000011000000000
00000000000000011100000000000000000001110000000000000000000000000
```

```
00110000000000000000000000001110000000000000000000000001111000000000
00000000000000001111110100000000000000000000111111111111100000000000
00000111111111111100000000000000000011111111000000000000000000001111
01111000000000000000000011111111000000000000000000011111110000000000
00000000000011111000000000000000000000100000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
    logic [783:0] testfive =
784'b00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000001111111000000000000000001111110011100000000000
00000111111000000100000000000000111000000000000000000000011000000000
00000000000000000001100000000000000000000000111000000000000000000000
00000011000000000000000000000000011000000000000000000000001100000000
00000000000000000001111111000000000000000000001111111110000000000000
00000000000000110000000000000000000001100000000000000000000000000000
01110000000000000000000001111000000000000000000001111000000000000000
00000000111111000000000000000000111111000000000000000000000011000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
//    logic [783:0] testsix =
784'b00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000001110000000000000000000000001111000000000
00000000000000001110000000000000000000000001110000000000000000000000
00011000000000000000000000000011000000000000000000000001110000000000
00000000000000001100000111111100000000000001100001111111000000000000
00011100011111000110000000000011100111100000110000000000011100111000
00001100000000000011101110000000110000000000011101110000000110000000
00000111001110000011000000000000111000111000111000000000000011100000
00111100000000000001110001111100000000000000000011111111000000000000
00000000011111100000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
    logic [783:0] testsix =
784'b00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000001000000000000000000000000001100000000000
00000000000000010000000000000000000000000000010000000000000000000000
00110000000000000000000000000011000000000000000000000001110000000000
10000000000000001100000011111000000000000001100000011100110000000000
```

```
00011000000110001100000000000011000001100001100000000000001100000110
00011000000000000011000110000011000000000000011000110000010000000000
00000001110110000110000000000000001111100111000000000000000000001111
11110000000000000000000001111000000000000000000000001111100000000000
00000000000001110000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
    logic [783:0] testseven =
784'b00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
01110000000000000000000001111111111111110000000000000101111111111
11100000000000000000000001110000000000000000000000001100000000000
00000000000001100000000000000000000001110000000000000000000000000
11000000000000000000000001110000000000000000000000001100000000000
00000000000011100000000000000000000011000000000000000000000001
11000000000000000000000111000000000000000000000001100000000000
00000000000111000000000000000000000111000000000000000000000011
10000000000000000000001111000000000000000000000011100000000000
00000000000000000000000000;
    logic [783:0] testeight =
784'b00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000011111111100000000000000
00000111111111111000000000000001111110001111000000000000111100000000
00111000000000000111000000001110000000000001111000000111100000000000
00000011100000111110000000000000011100011111000000000000000011111
11100000000000000000111111000000000000000000001111111100000000000000
00001111111011100000000000001111110001110000000000000111110111111
11000000000000000111111111110000000000000000011111111000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
    logic [783:0] testnine =
784'b00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000011000000000000000000000
00000111000000000000000000000001111100000000000000000001111111100
00000000000000000111111110000000000000000000011000111100000000000000
```

```
000001000011110000000000000000011100011110000000000000000000011001111
100000000000000000001111110111000000000000000000011110011100000000000
000000000110000111000000000000000000000011000000000000000000000000000
000111000000000000000000000000011000000000000000000000000111000000000
000000000000000000011000000000000000000000011000000000000000000000000
000000011000000000000000000000000010000000000000000000000000000000000
000000000000000000000000000;


    logic [783:0] zeroblock =
784'b000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000111110000000000000
000000011111110000000000000000001111011110000000000000000011000000010
000000000000000000110000000010000000000000000010000000001000000000000
000001000000000010000000000001100000000001000000000000100000000000000
000000000000000001100000000000000000000011000000000000000000000000000
000001100000000000000000000011000000000000000000000011000000000000000
000100000000000000011000000110000000000000000011111111100000000000000
000000001111110000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000;
    logic [783:0] oneblock =
784'b000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000111000000000000000
000000011110000000000000000000011110000000000000000000011110000000000
000000000000000000001110000000000000000000011110000000000000000000000
000000001111000000000000000000001111000000000000000000000001110000000
000000000000000000001111000000000000000000001111000000000000000000000
000000000111100000000000000000000111000000000000000000000001111000000
000000000011110000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000;
    logic [783:0] twoblock =
784'b000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000
000000000000000001111110000000000000000000001111110000000000000000000
```

```
0000000001100000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000010000000000000000000000000011
1000000000000000000001111000000000000000000001111110000000000000
0000000111111100000000000000001111111100000000000000001111111
1100000000000000000111111100000000000000000001111100000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
    logic [783:0] threeblock =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000001110000000000000000000001111111000000000000000
0000001111100000000000000000000000001000000000000000000000000010
0000000000000000000000110000000000000000000011110000000000000000
0000000111111000000000000000001111111000000000000000000000011111
1000000000000000000000000000000000000000000000000000100000000000
0000000000000000000000000000000000001000000000000000000000000000
0001000000000000000000000011000000000000000000000011110000000000
0000000011111111000000000000000011111000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
    logic [783:0] fourblock =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000001
0000000000000000010000011100000000000000001100000011100000000000
0000111000001110000000000000001110000011100000000000001111100111
1110000000000000011111111111100000000000000011111111111000000000
0000000000011110000000000000000011100000000000000000000000000000
1000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
    logic [783:0] fiveblock =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000001111100000000000000
```

```
000000111111100000000000000000000111100000000000000000000011100000
000000000000000000111000000000000000000000000111000000000000000000
000001111100000000000000000000011111100000000000000000001111100
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000011000000000000000001111111100000000000
000000001111111100000000000000000001111000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
    logic [783:0] sixblock =
784'b000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000001000000
000000000000000000010000000000000000000000010000000000000000000
000000110000000000000000000000011000001000000000000000011100011
110000000000000000111001110000000000000000001110110000000000000000
000000111100001100000000000000011111000111000000000000000111111
111100000000000000000111111110000000000000000011111100000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
    logic [783:0] sevenblock =
784'b000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000011111111110000000000000011111111111
100000000000000001110001111000000000000000000000000111000000000000
000000000000111000000000000000000011100000000000000000000000000001
110000000000000000000000111100000000000000000000000111000000000000
000000000001111000000000000000000011100000000000000000000000000001
100000000000000000000001110000000000000000000000011000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
    logic [783:0] eightblock =
784'b000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000001111110000000000000
```

```
000000111111100000000000000000001110000000000000000000000010000000
000000000000000000000000000000000000000000000000000000110000000000
000000011001111000000000000000001111111000000000000000000000011111
000000000000000000001111111000000000000000000001111100000000000000
000000000111000000000000000000000110000000000000000000000010000000
000000000000000000000000000000000000000000000000000000000000000000
000000000011111100000000000000000001111000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000;
    logic [783:0] nineblock =
784'b000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000
000000111111000000000000000000001111111100000000000000000111000011
000000000000000000111000001100000000000000000011000000111000000000000
000001000000111000000000000000011000011110000000000000000011000111
110000000000000000011111111100000000000000000000000111000000000000
000000000000111000000000000000000011000000000000000000000000000000
100000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000;

    always_ff @(posedge clk_65mhz)begin
        data <= {actual, 4'b0000, 4'b0000, 4'b0000, valuecalc,
4'b0000, 4'b0000, value64};

        if(sw==prevsw)begin
            reset <= 1'b0;
        end else begin
            reset <= 1'b1;
            prevsw <= sw;
        end

        case(sw)
            default: begin
                testblock <= testzero;
            end
            4'b0001: begin
```

```
                testblock <= testzero;
            end
            4'b0010: begin
                testblock <= testone;
            end
            4'b0011: begin
                testblock <= testtwo;
            end
            4'b0100: begin
                testblock <= testthree;
            end
            4'b0101: begin
                testblock <= testfour;
            end
            4'b0110: begin
                testblock <= testfive;
            end
            4'b0111: begin
                testblock <= testsix;
            end
            4'b1000: begin
                testblock <= testseven;
            end
            4'b1001: begin
                testblock <= testeight;
            end
            4'b1010: begin
                testblock <= testnine;
            end
        endcase

    end

endmodule

module tallier(
    input[783:0] val_in,
    output[5:0] tally); // up to 64
```

```
    assign tally =
val_in[0]+val_in[1]+val_in[2]+val_in[3]+val_in[4]+val_in[5]+val_in[6]
+val_in[7]+val_in[8]+val_in[9]+val_in[10]+val_in[11]+val_in[12]+val_i
n[13]+val_in[14]+val_in[15]+val_in[16]+val_in[17]+val_in[18]+val_in[1
9]+val_in[20]+val_in[21]+val_in[22]+val_in[23]+val_in[24]+val_in[25]+
val_in[26]+val_in[27]+val_in[28]+val_in[29]+val_in[30]+val_in[31]+val
_in[32]+val_in[33]+val_in[34]+val_in[35]+val_in[36]+val_in[37]+val_in
[38]+val_in[39]+val_in[40]+val_in[41]+val_in[42]+val_in[43]+val_in[44
]+val_in[45]+val_in[46]+val_in[47]+val_in[48]+val_in[49]+val_in[50]+v
al_in[51]+val_in[52]+val_in[53]+val_in[54]+val_in[55]+val_in[56]+val_
in[57]+val_in[58]+val_in[59]+val_in[60]+val_in[61]+val_in[62]+val_in[
63]+val_in[64]+val_in[65]+val_in[66]+val_in[67]+val_in[68]+val_in[69]
+val_in[70]+val_in[71]+val_in[72]+val_in[73]+val_in[74]+val_in[75]+va
l_in[76]+val_in[77]+val_in[78]+val_in[79]+val_in[80]+val_in[81]+val_i
n[82]+val_in[83]+val_in[84]+val_in[85]+val_in[86]+val_in[87]+val_in[8
8]+val_in[89]+val_in[90]+val_in[91]+val_in[92]+val_in[93]+val_in[94]+
val_in[95]+val_in[96]+val_in[97]+val_in[98]+val_in[99]+val_in[100]+va
l_in[101]+val_in[102]+val_in[103]+val_in[104]+val_in[105]+val_in[106]
+val_in[107]+val_in[108]+val_in[109]+val_in[110]+val_in[111]+val_in[1
12]+val_in[113]+val_in[114]+val_in[115]+val_in[116]+val_in[117]+val_i
n[118]+val_in[119]+val_in[120]+val_in[121]+val_in[122]+val_in[123]+va
l_in[124]+val_in[125]+val_in[126]+val_in[127]+val_in[128]+val_in[129]
+val_in[130]+val_in[131]+val_in[132]+val_in[133]+val_in[134]+val_in[1
35]+val_in[136]+val_in[137]+val_in[138]+val_in[139]+val_in[140]+val_i
n[141]+val_in[142]+val_in[143]+val_in[144]+val_in[145]+val_in[146]+va
l_in[147]+val_in[148]+val_in[149]+val_in[150]+val_in[151]+val_in[152]
+val_in[153]+val_in[154]+val_in[155]+val_in[156]+val_in[157]+val_in[1
58]+val_in[159]+val_in[160]+val_in[161]+val_in[162]+val_in[163]+val_i
n[164]+val_in[165]+val_in[166]+val_in[167]+val_in[168]+val_in[169]+va
l_in[170]+val_in[171]+val_in[172]+val_in[173]+val_in[174]+val_in[175]
+val_in[176]+val_in[177]+val_in[178]+val_in[179]+val_in[180]+val_in[1
81]+val_in[182]+val_in[183]+val_in[184]+val_in[185]+val_in[186]+val_i
n[187]+val_in[188]+val_in[189]+val_in[190]+val_in[191]+val_in[192]+va
l_in[193]+val_in[194]+val_in[195]+val_in[196]+val_in[197]+val_in[198]
+val_in[199]+val_in[200]+val_in[201]+val_in[202]+val_in[203]+val_in[2
04]+val_in[205]+val_in[206]+val_in[207]+val_in[208]+val_in[209]+val_i
n[210]+val_in[211]+val_in[212]+val_in[213]+val_in[214]+val_in[215]+va
l_in[216]+val_in[217]+val_in[218]+val_in[219]+val_in[220]+val_in[221]
+val_in[222]+val_in[223]+val_in[224]+val_in[225]+val_in[226]+val_in[2
```

```
27]+val_in[228]+val_in[229]+val_in[230]+val_in[231]+val_in[232]+val_i
n[233]+val_in[234]+val_in[235]+val_in[236]+val_in[237]+val_in[238]+va
l_in[239]+val_in[240]+val_in[241]+val_in[242]+val_in[243]+val_in[244]
+val_in[245]+val_in[246]+val_in[247]+val_in[248]+val_in[249]+val_in[2
50]+val_in[251]+val_in[252]+val_in[253]+val_in[254]+val_in[255]+val_i
n[256]+val_in[257]+val_in[258]+val_in[259]+val_in[260]+val_in[261]+va
l_in[262]+val_in[263]+val_in[264]+val_in[265]+val_in[266]+val_in[267]
+val_in[268]+val_in[269]+val_in[270]+val_in[271]+val_in[272]+val_in[2
73]+val_in[274]+val_in[275]+val_in[276]+val_in[277]+val_in[278]+val_i
n[279]+val_in[280]+val_in[281]+val_in[282]+val_in[283]+val_in[284]+va
l_in[285]+val_in[286]+val_in[287]+val_in[288]+val_in[289]+val_in[290]
+val_in[291]+val_in[292]+val_in[293]+val_in[294]+val_in[295]+val_in[2
96]+val_in[297]+val_in[298]+val_in[299]+val_in[300]+val_in[301]+val_i
n[302]+val_in[303]+val_in[304]+val_in[305]+val_in[306]+val_in[307]+va
l_in[308]+val_in[309]+val_in[310]+val_in[311]+val_in[312]+val_in[313]
+val_in[314]+val_in[315]+val_in[316]+val_in[317]+val_in[318]+val_in[3
19]+val_in[320]+val_in[321]+val_in[322]+val_in[323]+val_in[324]+val_i
n[325]+val_in[326]+val_in[327]+val_in[328]+val_in[329]+val_in[330]+va
l_in[331]+val_in[332]+val_in[333]+val_in[334]+val_in[335]+val_in[336]
+val_in[337]+val_in[338]+val_in[339]+val_in[340]+val_in[341]+val_in[3
42]+val_in[343]+val_in[344]+val_in[345]+val_in[346]+val_in[347]+val_i
n[348]+val_in[349]+val_in[350]+val_in[351]+val_in[352]+val_in[353]+va
l_in[354]+val_in[355]+val_in[356]+val_in[357]+val_in[358]+val_in[359]
+val_in[360]+val_in[361]+val_in[362]+val_in[363]+val_in[364]+val_in[3
65]+val_in[366]+val_in[367]+val_in[368]+val_in[369]+val_in[370]+val_i
n[371]+val_in[372]+val_in[373]+val_in[374]+val_in[375]+val_in[376]+va
l_in[377]+val_in[378]+val_in[379]+val_in[380]+val_in[381]+val_in[382]
+val_in[383]+val_in[384]+val_in[385]+val_in[386]+val_in[387]+val_in[3
88]+val_in[389]+val_in[390]+val_in[391]+val_in[392]+val_in[393]+val_i
n[394]+val_in[395]+val_in[396]+val_in[397]+val_in[398]+val_in[399]+va
l_in[400]+val_in[401]+val_in[402]+val_in[403]+val_in[404]+val_in[405]
+val_in[406]+val_in[407]+val_in[408]+val_in[409]+val_in[410]+val_in[4
11]+val_in[412]+val_in[413]+val_in[414]+val_in[415]+val_in[416]+val_i
n[417]+val_in[418]+val_in[419]+val_in[420]+val_in[421]+val_in[422]+va
l_in[423]+val_in[424]+val_in[425]+val_in[426]+val_in[427]+val_in[428]
+val_in[429]+val_in[430]+val_in[431]+val_in[432]+val_in[433]+val_in[4
34]+val_in[435]+val_in[436]+val_in[437]+val_in[438]+val_in[439]+val_i
n[440]+val_in[441]+val_in[442]+val_in[443]+val_in[444]+val_in[445]+va
l_in[446]+val_in[447]+val_in[448]+val_in[449]+val_in[450]+val_in[451]
```

```
+val_in[452]+val_in[453]+val_in[454]+val_in[455]+val_in[456]+val_in[4
57]+val_in[458]+val_in[459]+val_in[460]+val_in[461]+val_in[462]+val_i
n[463]+val_in[464]+val_in[465]+val_in[466]+val_in[467]+val_in[468]+va
l_in[469]+val_in[470]+val_in[471]+val_in[472]+val_in[473]+val_in[474]
+val_in[475]+val_in[476]+val_in[477]+val_in[478]+val_in[479]+val_in[4
80]+val_in[481]+val_in[482]+val_in[483]+val_in[484]+val_in[485]+val_i
n[486]+val_in[487]+val_in[488]+val_in[489]+val_in[490]+val_in[491]+va
l_in[492]+val_in[493]+val_in[494]+val_in[495]+val_in[496]+val_in[497]
+val_in[498]+val_in[499]+val_in[500]+val_in[501]+val_in[502]+val_in[5
03]+val_in[504]+val_in[505]+val_in[506]+val_in[507]+val_in[508]+val_i
n[509]+val_in[510]+val_in[511]+val_in[512]+val_in[513]+val_in[514]+va
l_in[515]+val_in[516]+val_in[517]+val_in[518]+val_in[519]+val_in[520]
+val_in[521]+val_in[522]+val_in[523]+val_in[524]+val_in[525]+val_in[5
26]+val_in[527]+val_in[528]+val_in[529]+val_in[530]+val_in[531]+val_i
n[532]+val_in[533]+val_in[534]+val_in[535]+val_in[536]+val_in[537]+va
l_in[538]+val_in[539]+val_in[540]+val_in[541]+val_in[542]+val_in[543]
+val_in[544]+val_in[545]+val_in[546]+val_in[547]+val_in[548]+val_in[5
49]+val_in[550]+val_in[551]+val_in[552]+val_in[553]+val_in[554]+val_i
n[555]+val_in[556]+val_in[557]+val_in[558]+val_in[559]+val_in[560]+va
l_in[561]+val_in[562]+val_in[563]+val_in[564]+val_in[565]+val_in[566]
+val_in[567]+val_in[568]+val_in[569]+val_in[570]+val_in[571]+val_in[5
72]+val_in[573]+val_in[574]+val_in[575]+val_in[576]+val_in[577]+val_i
n[578]+val_in[579]+val_in[580]+val_in[581]+val_in[582]+val_in[583]+va
l_in[584]+val_in[585]+val_in[586]+val_in[587]+val_in[588]+val_in[589]
+val_in[590]+val_in[591]+val_in[592]+val_in[593]+val_in[594]+val_in[5
95]+val_in[596]+val_in[597]+val_in[598]+val_in[599]+val_in[600]+val_i
n[601]+val_in[602]+val_in[603]+val_in[604]+val_in[605]+val_in[606]+va
l_in[607]+val_in[608]+val_in[609]+val_in[610]+val_in[611]+val_in[612]
+val_in[613]+val_in[614]+val_in[615]+val_in[616]+val_in[617]+val_in[6
18]+val_in[619]+val_in[620]+val_in[621]+val_in[622]+val_in[623]+val_i
n[624]+val_in[625]+val_in[626]+val_in[627]+val_in[628]+val_in[629]+va
l_in[630]+val_in[631]+val_in[632]+val_in[633]+val_in[634]+val_in[635]
+val_in[636]+val_in[637]+val_in[638]+val_in[639]+val_in[640]+val_in[6
41]+val_in[642]+val_in[643]+val_in[644]+val_in[645]+val_in[646]+val_i
n[647]+val_in[648]+val_in[649]+val_in[650]+val_in[651]+val_in[652]+va
l_in[653]+val_in[654]+val_in[655]+val_in[656]+val_in[657]+val_in[658]
+val_in[659]+val_in[660]+val_in[661]+val_in[662]+val_in[663]+val_in[6
64]+val_in[665]+val_in[666]+val_in[667]+val_in[668]+val_in[669]+val_i
n[670]+val_in[671]+val_in[672]+val_in[673]+val_in[674]+val_in[675]+va
```

```
l_in[676]+val_in[677]+val_in[678]+val_in[679]+val_in[680]+val_in[681]
+val_in[682]+val_in[683]+val_in[684]+val_in[685]+val_in[686]+val_in[6
87]+val_in[688]+val_in[689]+val_in[690]+val_in[691]+val_in[692]+val_i
n[693]+val_in[694]+val_in[695]+val_in[696]+val_in[697]+val_in[698]+va
l_in[699]+val_in[700]+val_in[701]+val_in[702]+val_in[703]+val_in[704]
+val_in[705]+val_in[706]+val_in[707]+val_in[708]+val_in[709]+val_in[7
10]+val_in[711]+val_in[712]+val_in[713]+val_in[714]+val_in[715]+val_i
n[716]+val_in[717]+val_in[718]+val_in[719]+val_in[720]+val_in[721]+va
l_in[722]+val_in[723]+val_in[724]+val_in[725]+val_in[726]+val_in[727]
+val_in[728]+val_in[729]+val_in[730]+val_in[731]+val_in[732]+val_in[7
33]+val_in[734]+val_in[735]+val_in[736]+val_in[737]+val_in[738]+val_i
n[739]+val_in[740]+val_in[741]+val_in[742]+val_in[743]+val_in[744]+va
l_in[745]+val_in[746]+val_in[747]+val_in[748]+val_in[749]+val_in[750]
+val_in[751]+val_in[752]+val_in[753]+val_in[754]+val_in[755]+val_in[7
56]+val_in[757]+val_in[758]+val_in[759]+val_in[760]+val_in[761]+val_i
n[762]+val_in[763]+val_in[764]+val_in[765]+val_in[766]+val_in[767]+va
l_in[768]+val_in[769]+val_in[770]+val_in[771]+val_in[772]+val_in[773]
+val_in[774]+val_in[775]+val_in[776]+val_in[777]+val_in[778]+val_in[7
79]+val_in[780]+val_in[781]+val_in[782]+val_in[783];

endmodule

//module sum_comb_digit(
//    input [15:12] sw,
//    output logic [3:0] value
//);

//    logic [783:0] testblock;
//    logic [783:0] allzeroes =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
```

```
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;

//    logic [783:0] testzero =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000011100000
0000000000000000001111000000000000000000000001111000000000000000000
0000111111110000000000000000001111111100000000000000001111111111
1000000000000000011111001111000000000000001111100001111000000000
0000111100000011110000000000001110000000111000000000000110000000
0011110000000000111000000011110000000000001110000000111100000000
0000011100000011111000000000000111000011111100000000000011100111
1111100000000000001111111111110000000000000011111111100000000000
0000000011111111000000000000000000001110000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
//    logic [783:0] testone =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000011000
0000000000000000000000100000000000000000000001100000000000000000
0000000001100000000000000000000000010000000000000000000000011000
0000000000000000000000110000000000000000000001100000000000000000
0000000001100000000000000000000000011000000000000000000000011100
0000000000000000000001100000000000000000000001100000000000000000
0000000001110000000000000000000000011100000000000000000000001100
0000000000000000000001100000000000000000000001110000000000000000
0000000001110000000000000000000000011000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
//    logic [783:0] testtwo =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000001111110000000000000000001111111000000
0000000000000111111111100000000000000011100011100000000000000000
0011000011100000000000000000000000011100000000000000000000111100
0000000000000000000011100000000000000000000011110000000000000000
0000000111100000000000000000000001110000000000000000000011110000
0000000000000000000011100000000000000000000011110000000000000000
0000001111000000000000000000000011100000000000000000000011100000
0000001111000000000111111111111111110000000011111111111111100
```

```
00000000011111111110000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
//    logic [783:0] testthree =
784'b000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000110000000000
00000000000011111100000000000000000011111111000000000000000000000001
11111111100000000000000000111000011110000000000000000011000001111000
00000000000001100000011100000000000001100010001111000000000000000000
00100111111111000000000000000000011111111110000000000000000001111111
11111000000000000000000000001111100000000000000000000000001111000000
00000011000000000011100000000000011100000000001110000000000011100000
00000011100000000000111100001111110000000000000111111111111000000000
00000000000011111111110000000000000000000010100000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
//    logic [783:0] testfour =
784'b000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
00000000000000000100000000000000000000011000000110000000000000000000
00001100000000100000000000000001100000011000000000000001100000000000
11000000000000011000000011000000000000011000000011100000000000000000
00011100000000110000000000000011000000011100000000000000110000000000
11100000000000001100000001110000000000000111000011111000000000000000
00000111111111111000000000000000000000011100000000000000000000000000
01110000000000000000000000011100000000000000000000000111000000000000
00000000000000011100000000000000000000011100000000000000000000000000
00010000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
//    logic [783:0] testfive =
784'b000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000
01000000000000000000011111111100000000000000111111111111000000000000
00000011111111111110000000000010001111100000000000000000110000000000
00000000000000011100000000000000000000011100000000000000000000000000
00110000000000000000000000011100000000000000000000000111100000000000
00000000000000011111101000000000000000000111111111111000000000000000
00000111111111111100000000000000000011111111100000000000000001111
01111000000000000000000011111111000000000000000000000111111000000000
```

```
00000000000011111000000000000000000000001000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
//    logic [783:0] testsix =
784'b00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000011100000000000000000000000011110000000
00000000000000011100000000000000000000011100000000000000000000000
00011000000000000000000000000110000000000000000000000011100000000
00000000000000001100000111111100000000000011000001111111000000000
00011100011111000110000000000011100111100000110000000000011100111100
00001100000000000111011100000011000000000001110111000000011000000
00000111001110000011000000000000111000111000111000000000000011100000
00111100000000000001110001111100000000000000000011111111000000000
00000000011111100000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
//    logic [783:0] testseven =
784'b00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
01110000000000000000000000111111111111111000000000000010111111111
11100000000000000000000000111000000000000000000000011000000000000
00000000000001100000000000000000000001110000000000000000000000000
11000000000000000000000011100000000000000000000000011000000000000
00000000000011100000000000000000000011000000000000000000000000001
11000000000000000000000011100000000000000000000000011000000000000
00000000000111000000000000000000000011100000000000000000000000011
10000000000000000000000011110000000000000000000000011100000000000
00000000000000000000000000;
//    logic [783:0] testeight =
784'b00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000011111111000000000000
00000111111111111000000000000001111110001111000000000000111100000
00111000000000000111000000001110000000000011110000001111000000000
00000011100000111110000000000000011100011111000000000000000011111
11100000000000000000111111000000000000000000001111111000000000000
00001111111011100000000000001111110001110000000000001111011111
11000000000000001111111111000000000000000000011111111000000000000
```

```
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000;
//    logic [783:0] testnine =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000011000000000000000000000
0000011100000000000000000000000011111000000000000000000001111111100
0000000000000000011111111000000000000000000011000111100000000000000
0000100001111000000000000000000111000111100000000000000000011001111
1000000000000000000111111011100000000000000000111001110000000000000
0000000011000011100000000000000000000000001100000000000000000000000
0001110000000000000000000000001100000000000000000000000111000000000
0000000000000000000110000000000000000000000000011000000000000000000
0000000011000000000000000000000000000010000000000000000000000000000
0000000000000000000000000000;

//    logic [783:0] zeroblock =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000011000000000000000000000001111000000000000000
0000000100111110000000000000000000000011110000000000000000000000000
0111000000000000011111000001110000000000001110000000011100000000000
0001110000000001110000000000011000000000011100000000011000000000000
0001110000000001110000000000111000000000011100000000011110000000000
0000111000000000111000000000011100000000011100000000000111000000000
0000110000000000011110000000110000000000000111111001000000000000000
0000000011111100000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000;
//    logic [783:0] oneblock =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000011000000000000000000000001100000000000000000000
0000000001100000000000000000000000000110000000000000000000000000100
0000000000000000000000010000000000000000000000011000000000000000000
0000000000110000000000000000000000010000000000000000000000000000000
```

```
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000;
//    logic [783:0] twoblock =
784'b000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000011100000
000000000000000000011100000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000010000000000000000000000010100000000000000000000100000
000000000000000000001000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000;
//    logic [783:0] threeblock =
784'b000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000001110000000000000000001111111000000000000000000000
000111000000000000000000000000000000000000000000000000000000000000
000000000000000000000010000000000000000000010010000000000000000000
000000001001000000000000000000000110000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000001000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000111000000
000000000000111100000000000000000110000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000;
//    logic [783:0] fourblock =
784'b000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000111000000
010000000000000000111000111100000000000000000010001111000000000000
000000000000011000000000000000000000000000000000000000000000000000
```

```
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
//    logic [783:0] fiveblock =
784'b000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
//    logic [783:0] sixblock =
784'b000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
//    logic [783:0] sevenblock =
784'b000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
```

```
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000;
//    logic [783:0] eightblock =
784'b000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000001000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000011010
000000000000000000001100000000000000000000000010000000000000000000
000000001000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000001111110000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000;
//    logic [783:0] nineblock =
784'b000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000011000000000000000000000011000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000110000000000000000000001110000000000000000000000011
100000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000010000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000;

//    logic [5:0] zero, one, two, three, four, five, six, seven,
eight, nine;
//    tallier zerotally(.val_in((testblock&zeroblock)),
.tally(zero));
//    tallier onetally(.val_in((testblock&oneblock)), .tally(one));
//    tallier twotally(.val_in((testblock&twoblock)), .tally(two));
//    tallier threetally(.val_in((testblock&threeblock)),
.tally(three));
```

```
//    tallier fourtally(.val_in((testblock&fourblock)),
.tally(four));
//    tallier fivetally(.val_in((testblock&fiveblock)),
.tally(five));
//    tallier sixtally(.val_in((testblock&sixblock)), .tally(six));
//    tallier seventally(.val_in((testblock&sevenblock)),
.tally(seven));
//    tallier eighttally(.val_in((testblock&eightblock)),
.tally(eight));
//    tallier ninetally(.val_in((testblock&nineblock)),
.tally(nine));

//    always_comb begin
//        case(sw)
//            default: begin
//                testblock = allzeroes;
////                  led[9:0] = 10'b0000000000;
//            end
//            4'b0001: begin
//                testblock = testzero;
//            end
//            4'b0010: begin
//                testblock = testone;
//            end
//            4'b0011: begin
//                testblock = testtwo;
//            end
//            4'b0100: begin
//                testblock = testthree;
//            end
//            4'b0101: begin
//                testblock = testfour;
//            end
//            4'b0110: begin
//                testblock = testfive;
//            end
//            4'b0111: begin
//                testblock = testsix;
//            end
```

```
//            4'b1000: begin
//                testblock = testseven;
//            end
//            4'b1001: begin
//                testblock = testeight;
//            end
//            4'b1010: begin
//                testblock = testnine;
//            end
//        endcase

//        if(zero>one & zero>two & zero>three & zero>four & zero>five
& zero>six & zero>seven & zero>eight & zero>nine)begin //zero
//            value = 4'b0000;
//        end else if (one>zero & one>two & one>three & one>four &
one>five & one>six & one>seven & one>eight & one>nine)begin //one
//            value = 4'b0001;
//        end else if (two>zero & two>one & two>three & two>four &
two>five & two>six & two>seven & two>eight & two>nine)begin //two
//            value = 4'b0010;
//        end else if (three>zero & three>one & three>two &
three>four & three>five & three>six & three>seven & three>eight &
three>nine)begin //three
//            value = 4'b0011;
//        end else if (four>zero & four>one & four>two & four>three &
four>five & four>six & four>seven & four>eight & four>nine)begin
//four
//            value = 4'b0100;
//        end else if (five>zero & five>one & five>two & five>three &
five>four & five>six & five>seven & five>eight & five>nine)begin
//five
//            value = 4'b0101;
//        end else if (six>zero & six>one & six>two & six>three &
six>four & six>five & six>seven & six>eight & six>nine)begin //six
//            value = 4'b0110;
//        end else if (seven>zero & seven>one & seven>two &
seven>three & seven>four & seven>five & seven>six & seven>eight &
seven>nine)begin //seven
//            value = 4'b0111;
```

```
//        end else if (eight>zero & eight>one & eight>two &
eight>three & eight>four & eight>five & eight>six & eight>seven &
eight>nine)begin //eight
//            value = 4'b1000;
//        end else if (nine>zero & nine>one & nine>two & nine>three &
nine>four & nine>five & nine>six & nine>seven & nine>eight)begin
//nine
//            value = 4'b1001;
//        end

//    end

//endmodule

module sum_64_digit(
    input clk_in,
    input [3:0] sw,
    output logic [3:0] value
);
    logic [783:0] testblock;

    logic [783:0] allzeroes =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000;

    logic [783:0] testzero =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000011100000
0000000000000000000011110000000000000000000000001111000000000000000000
```

```
000011111111000000000000000000001111111100000000000000000001111111111
100000000000000000111111001111000000000000000011111100001111000000000
000011110000001111000000000000011100000001110000000000000110000000
001111000000000000111000000011110000000000001110000000111100000000
000001110000001111100000000000011100001111110000000000011100111
111100000000000000011111111111100000000000000011111111110000000000
000000001111111100000000000000000011100000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
    logic [783:0] testone =
784'b000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000001100
000000000000000000000010000000000000000000000011000000000000000000
000000000110000000000000000000000010000000000000000000000001100
000000000000000000000110000000000000000000000110000000000000000
000000000110000000000000000000000110000000000000000000001110
000000000000000000000110000000000000000000000110000000000000000
000000000111000000000000000000000111000000000000000000000110
000000000000000000000110000000000000000000000111000000000000000
000000000011100000000000000000000110000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
//    logic [783:0] testtwo =
784'b000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000001111100000000000000000000011111110000
000000000000001111111111000000000000000001110000111000000000000000
001100001110000000000000000000000011100000000000000000000111100
000000000000000000001110000000000000000000000111100000000000000000
000000111100000000000000000000000111000000000000000000000001111000
000000000000000000001110000000000000000000000111100000000000000000
000000111100000000000000000000000111000000000000000000000001110000
000000111100000000011111111111111111100000000011111111111111100
000000001111111111000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
    logic [783:0] testtwo =
784'b000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000001111110
000000000000000011111111110000000000000001111100001100000000000000
```

```
000111000000001100000000000001110000000011000000000000110000000000
011000000000000100000000000011000000000000000000000000110000000000
000000000000001100000000000000000000011100000000000000000000000000
111000000000000000000000011100000000000000000000011100000000000000
000000000001110000000000000000011111000000000000000001111111000000
110000100000000000111111111111010000000001111111000111111000000000
000011111100000011000000000000011000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000;
//    logic [783:0] testthree =
784'b000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000110000000000000
000000000000111111000000000000000000011111111000000000000000000001
111111111000000000000000001110001111000000000000000011000001111000
000000000000011000000011100000000000000011000100011110000000000000
001001111111110000000000000000111111111110000000000000000011111111
111100000000000000000000000011111000000000000000000000011110000000
000000110000000001110000000000011100000000011100000000000111000000
000000111000000000000111100001111110000000000000111111111111000000
000000000000111111111100000000000000000010100000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000;
    logic [783:0] testthree =
784'b000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000011110000000000
000000000000000111110000000000000000000011111110000000000000000000
001000111000000000000000000000001100000000000000000000000110000000
000000000000000000001100000000000000000000000011100000000000000000
000000011100000000000000000000011111000000000000000000011111111111
100000000000000000000001111000000000000000000000011100000000000000
000000000000111000000000000000000011110000000000000000000000000001
111000000000000000000011111100000000000000000000111110000000000000
000000000111110000000000000000011000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000;
    logic [783:0] testfour =
784'b000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000001000000000000000000000000011000000011000000000000
```

```
000011000000001000000000000000001100000001100000000000000001100000000
110000000000000001100000000011000000000000000110000000011100000000000
000111000000001100000000000000011000000001110000000000000001100000000
111000000000000000110000000011100000000000000011100001111110000000000
000000111111111110000000000000000000000000111000000000000000000000000
011100000000000000000000000001110000000000000000000000000111000000000
000000000000000011100000000000000000000000001110000000000000000000000
000100000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000;
//    logic [783:0] testfive =
784'b0000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000
010000000000000000000001111111110000000000000000111111111110000000000
000000011111111111100000000000010001111100000000000000000001100000000
000000000000000011100000000000000000000000001110000000000000000000000
001100000000000000000000000001110000000000000000000000001111000000000
000000000000000011111010000000000000000000001111111111111110000000000
000001111111111111000000000000000000011111111000000000000000000001111
011110000000000000000000111111110000000000000000000011111100000000000
000000000000011111000000000000000000000000010000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000;
    logic [783:0] testfive =
784'b0000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000111111000000000000000000111111001110000000000
000001111110000000100000000000000111000000000000000000000000001100000
000000000000000000001100000000000000000000000011100000000000000000000
000001100000000000000000000000000110000000000000000000000000011000000
000000000000000000000111111000000000000000000000111111111000000000000
000000000000000110000000000000000000000001100000000000000000000000000
011100000000000000000000000111100000000000000000000001111000000000000
000000000111111000000000000000000000111111000000000000000000000011000
000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000;
//    logic [783:0] testsix =
784'b0000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000011100000000000000000000000011110000000
000000000000000011100000000000000000000000001110000000000000000000000
```

```
0001100000000000000000000000001100000000000000000000000011100000000
0000000000000000110000011111110000000000001100000111111100000000
0001110001111100011000000000001110011111000011000000000001110011100
0000110000000000011101110000001100000000000111011100000011000000
0000011100111000011000000000001110001110001110000000000011100000
0011110000000000000111000111110000000000000000011111111000000000
0000000001111110000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000;
    logic [783:0] testsix =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000001000000000000000000000000110000000000
0000000000000001000000000000000000000001000000000000000000000000
0011000000000000000000000001100000000000000000000001110000000000
1000000000000001100000011111000000000000011000000111001100000000
0001100000110001100000000000011000011000011000000000001100000110
0001100000000000011000110000011000000000000110001100000100000000
0000001110110000110000000000000011111001110000000000000000001111
1111000000000000000000001111000000000000000000000001111100000000
0000000000001110000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000;
    logic [783:0] testseven =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0111000000000000000000000111111111111111000000000000101111111111
1110000000000000000000000011100000000000000000000001100000000000
0000000000001100000000000000000000011100000000000000000000000000
1100000000000000000000000111000000000000000000000011000000000000
0000000000011100000000000000000000011000000000000000000000000001
1100000000000000000000001110000000000000000000000110000000000000
0000000000111000000000000000000000111000000000000000000000001100
1000000000000000000000011110000000000000000000001110000000000000
0000000000000000000000000;
    logic [783:0] testeight =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000011111111100000000000
```

```
000001111111111110000000000000011111000011110000000000001111000000
001110000000000001110000000001110000000000001111000000111100000000
000001110000011111000000000000011100011111000000000000000011111
111000000000000000001111110000000000000000001111111100000000000000
000011111110111000000000000111111000011100000000000011111011111
110000000000000001111111111000000000000000001111111100000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000;
    logic [783:0] testnine =
784'b000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000011000000000000000000000
000001110000000000000000000001111100000000000000000001111111100
000000000000000001111111100000000000000000011000111100000000000000
000001000011110000000000000011100011110000000000000000011001111
100000000000000000111111011100000000000000000111100111000000000000
000000001100001110000000000000000000001100000000000000000000
000111000000000000000000000001100000000000000000000000111000000
000000000000000001100000000000000000000011000000000000000000
000000011000000000000000000000000010000000000000000000000000000
000000000000000000000000;


    logic [783:0] zeroblock =
784'b000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000001111100000000000000000
000000111111100000000000000000111101111000000000000000011000000
010000000000000001100000001000000000000001000000000100000000000
000010000000001000000000000110000000001000000000000100000000
000000000000000011000000000000000000001100000000000000000000
000001100000000000000000000011000000000000000000000011000000
000100000000000000011000000110000000000000001111111100000000000
000000001111110000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000;
    logic [783:0] oneblock =
784'b000000000000000000000000000000000000000000000000000000000000000000
```

```
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000011100000000000000
0000000011110000000000000000000001111000000000000000000000011110
0000000000000000000001110000000000000000000001111000000000000000
0000000011110000000000000000000011110000000000000000000001110
0000000000000000000011110000000000000000000011110000000000000000
0000000001111000000000000000000011100000000000000000000001111
0000000000000000000011110000000000000000000011110000000000000000
0000000000111100000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
    logic [783:0] twoblock =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000111111000000000000000000001111110000000000000000
0000000001100000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000010000000000000000000000011
1000000000000000000011110000000000000000001111110000000000000000
0000000111111100000000000000001111111100000000000000001111111
1100000000000000000111111000000000000000000011111000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
    logic [783:0] threeblock =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000011100000000000000000000011111110000000000000000
0000001111100000000000000000000010000000000000000000000010
0000000000000000000001100000000000000000000011110000000000000000
0000000111111000000000000000011111110000000000000000011111
1000000000000000000000000000000000000000000000000100000000000
0000000000000000000000000000000010000000000000000000000000
0001000000000000000000000011000000000000000000000011110000000000
0000000011111111000000000000000011111000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000;
    logic [783:0] fourblock =
784'b0000000000000000000000000000000000000000000000000000000000000000
```

```
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000001
0000000000000000010000011100000000000000011000000111000000000000
0000111000001110000000000000011100000111000000000000001111100111
1110000000000000011111111111000000000000000011111111110000000000
0000000000011110000000000000000000011100000000000000000000000000
1000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000;
    logic [783:0] fiveblock =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000001111100000000000000
0000011111111000000000000000000111100000000000000000000011100000
0000000000000000011100000000000000000000011100000000000000000000
0000011111000000000000000000011111100000000000000000001111100
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000011000000000000000011111111110000000000
0000000011111111000000000000000000011110000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000;
    logic [783:0] sixblock =
784'b0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000001000000
0000000000000000010000000000000000000000010000000000000000
0000011000000000000000000000001100000100000000000000011100011
1100000000000000001110011100000000000000000011101100000000000000
0000001111000001100000000000000111110001110000000000000111111
1111000000000000000011111111100000000000000000011111100000000000
0000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000;
    logic [783:0] sevenblock =
784'b0000000000000000000000000000000000000000000000000000000000000000
```

```
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000001111111111000000000000000011111111111
100000000000000001111000111100000000000000000000000001110000000000
000000000000111000000000000000000000111000000000000000000000001
110000000000000000000000111100000000000000000000000111000000000000
000000000001111000000000000000000001110000000000000000000000001
100000000000000000000000111000000000000000000000011000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000;
    logic [783:0] eightblock =
784'b000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000001111110000000000000000
000001111111000000000000000000111000000000000000000000010000000
000000000000000000000000000000000000000000000000000000110000000000000
000000110011110000000000000000011111110000000000000000011111
000000000000000000011111100000000000000000000011111000000000000000
000000001110000000000000000000011000000000000000000000010000
000000000000000000000000000000000000000000000000000000000000000000
000000000111111000000000000000000011110000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000;
    logic [783:0] nineblock =
784'b000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000
000000111111000000000000000000111111110000000000000000111000011
000000000000000001110000110000000000000000011000000111000000000000
000001000001110000000000000011000011110000000000000000011000111
110000000000000000111111111000000000000000000000000111000000000000
000000000000111000000000000000000011000000000000000000000000000000
100000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000;

    logic [5:0] zero, one, two, three, four, five, six, seven, eight,
```

```
nine;
    tallier zerotally(.val_in((testblock&zeroblock)), .tally(zero));
    tallier onetally(.val_in((testblock&oneblock)), .tally(one));
    tallier twotally(.val_in((testblock&twoblock)), .tally(two));
    tallier threetally(.val_in((testblock&threeblock)),
.tally(three));
    tallier fourtally(.val_in((testblock&fourblock)), .tally(four));
    tallier fivetally(.val_in((testblock&fiveblock)), .tally(five));
    tallier sixtally(.val_in((testblock&sixblock)), .tally(six));
    tallier seventally(.val_in((testblock&sevenblock)),
.tally(seven));
    tallier eighttally(.val_in((testblock&eightblock)),
.tally(eight));
    tallier ninetally(.val_in((testblock&nineblock)), .tally(nine));

    always_comb begin
        case(sw)
            default: begin
                testblock = testzero; // this way, it displays zero
//                led[9:0] = 10'b0000000000;
            end
            4'b0001: begin
                testblock = testzero;
            end
            4'b0010: begin
                testblock = testone;
            end
            4'b0011: begin
                testblock = testtwo;
            end
            4'b0100: begin
                testblock = testthree;
            end
            4'b0101: begin
                testblock = testfour;
            end
            4'b0110: begin
                testblock = testfive;
            end
```

```
            4'b0111: begin
                testblock = testsix;
            end
            4'b1000: begin
                testblock = testseven;
            end
            4'b1001: begin
                testblock = testeight;
            end
            4'b1010: begin
                testblock = testnine;
            end
        endcase
//    end

//    always_ff @(posedge clk_in)begin

//        case(sw)
//            default: begin
//                testblock <= allzeroes;
//            end
//            4'b0001: begin
//                testblock <= testzero;
//            end
//            4'b0010: begin
//                testblock <= testone;
//            end
//            4'b0011: begin
//                testblock <= testtwo;
//            end
//            4'b0100: begin
//                testblock <= testthree;
//            end
//            4'b0101: begin
//                testblock <= testfour;
//            end
//            4'b0110: begin
//                testblock <= testfive;
//            end
```

```
//            4'b0111: begin
//                testblock <= testsix;
//            end
//            4'b1000: begin
//                testblock <= testseven;
//            end
//            4'b1001: begin
//                testblock <= testeight;
//            end
//            4'b1010: begin
//                testblock <= testnine;
//            end
//        endcase

        if(zero>one & zero>two & zero>three & zero>four & zero>five &
zero>six & zero>seven & zero>eight & zero>nine)begin //zero
            value = 4'b0000;
        end else if (one>zero & one>two & one>three & one>four &
one>five & one>six & one>seven & one>eight & one>nine)begin //one
            value = 4'b0001;
        end else if (two>zero & two>one & two>three & two>four &
two>five & two>six & two>seven & two>eight & two>nine)begin //two
            value = 4'b0010;
        end else if (three>zero & three>one & three>two & three>four
& three>five & three>six & three>seven & three>eight &
three>nine)begin //three
            value = 4'b0011;
        end else if (four>zero & four>one & four>two & four>three &
four>five & four>six & four>seven & four>eight & four>nine)begin
//four
            value = 4'b0100;
        end else if (five>zero & five>one & five>two & five>three &
five>four & five>six & five>seven & five>eight & five>nine)begin
//five
            value = 4'b0101;
        end else if (six>zero & six>one & six>two & six>three &
six>four & six>five & six>seven & six>eight & six>nine)begin //six
            value = 4'b0110;
        end else if (seven>zero & seven>one & seven>two & seven>three
```

```
& seven>four & seven>five & seven>six & seven>eight &
seven>nine)begin //seven
            value = 4'b0111;
        end else if (eight>zero & eight>one & eight>two & eight>three
& eight>four & eight>five & eight>six & eight>seven &
eight>nine)begin //eight
            value = 4'b1000;
        end else if (nine>zero & nine>one & nine>two & nine>three &
nine>four & nine>five & nine>six & nine>seven & nine>eight)begin
//nine
            value = 4'b1001;
        end

    end

endmodule
```

densenet.sv

```
module densenet(
    input clk_in,
    input rst_in,
    input[783:0] val_in,
    output logic [3:0] outputval
    );

    logic signed [7:0] zeroweights[783:0] = {-20, -3, 15, -15, 4,
-10, -14, -18, -16, 13, 13, 4, 23, 6, 24, 0, 11, -7, -1, 12, -16, -7,
23, -9, 24, -17, 20, -19, 0, 13, 11, 21, -14, 24, -9, 5, -19, -10,
-24, -13, -16, 1, -16, -22, -4, -9, -19, 15, -19, -24, -23, -11, -18,
13, -6, -10, -17, 4, 10, 1, -10, 21, -9, -1, -24, 13, -9, 2, -9, 2,
-22, 5, 9, -8, -20, 5, -1, 19, -25, 15, -23, -11, -12, 4, -17, 3,
-24, 24, -15, -21, -4, -23, -17, 12, -14, -6, -6, -7, -5, -27, -15,
-7, -10, -20, 7, 4, -20, -22, -14, -18, -1, 2, -25, -8, -8, -22, 14,
13, -24, 19, -18, 15, -12, 3, -21, 20, -8, 27, 3, 5, 0, -1, -12, -1,
-19, 17, 1, -5, 13, 23, 4, 0, -14, 11, -1, -23, -10, 12, 0, 5, 5, -3,
```

```
-2, 22, 27, 19, 50, 17, 16, 25, -5, 9, -6, -31, -18, 19, 22, -16, -9,
-4, 14, 24, -21, -13, -20, -22, -1, -26, 7, -15, 8, 2, 24, 44, 58,
26, 35, 41, 3, 26, -17, 1, 7, -9, 7, -3, 6, -4, 13, -9, -22, -14, 8,
-15, 4, -10, -19, 16, 22, 21, 33, 34, 12, 40, 47, 12, 33, -6, 27,
-22, -4, 16, 13, -24, -7, -23, -12, -4, 7, 0, -5, -27, -7, -22, 7, 3,
-13, 11, 13, 22, 56, 50, 55, 17, 6, 37, 16, 7, 4, -26, 19, -12, -4,
14, -13, 6, 8, -8, 12, -8, 6, 3, 24, -6, 22, 40, 9, 27, -1, 36, 21,
38, 21, 13, 44, 9, -2, 6, -1, 22, -13, 17, 9, -6, -24, 0, -29, 13,
12, 18, 6, 27, 34, -9, 17, -13, -41, 12, 20, 43, 16, 42, 23, 33, 19,
16, 6, 17, 10, 1, 13, -12, 4, -3, 18, 1, -14, 9, -9, 22, -7, 9, -18,
-44, -61, -27, -6, -3, 39, 25, 60, 44, 17, 19, -22, -8, 24, -15, 23,
12, 16, 1, 25, -14, 6, -2, 43, 3, 10, -40, -62, -65, -100, -45, -31,
17, -2, 40, 38, 44, 13, 18, -16, 8, -17, 10, -16, 18, 9, -11, -2, 24,
6, 24, 29, 1, -18, -50, -78, -79, -97, -56, -21, 6, -5, 51, 45, 32,
4, 6, 20, 2, -17, 10, -6, 17, -26, 24, 42, 19, 20, 28, 18, 8, -35,
-79, -100, -94, -98, -68, -24, -9, 22, 23, 51, 21, 40, 21, 11, 0, 22,
-12, 7, 21, 20, 39, 11, 40, 27, 27, 53, -29, -47, -102, -90, -96,
-95, -36, -33, -1, 13, 35, 36, 56, 19, -15, 22, 3, -14, 11, 15, 4,
-22, 34, 34, 35, 53, 34, 35, -30, -46, -91, -74, -95, -46, -38, 16,
-3, 10, 1, 3, 38, 16, -10, 15, -1, 16, -7, 12, -24, -19, 7, 38, 46,
23, 59, 38, -4, -70, -63, -83, -69, -29, 14, -8, -1, 22, 32, 26, 11,
22, -2, 23, -8, -8, 24, -22, 5, 4, 16, 40, 59, 40, 56, 31, -7, -30,
-73, -31, -43, -21, -8, 0, 9, -13, 6, 11, 11, -3, 14, -9, 1, -3, -8,
-22, -24, -8, 12, 23, 19, 27, 32, 19, 7, 11, -41, -25, -7, 13, 21, 7,
-9, -12, -11, -18, -17, 4, 20, -1, 23, 3, -8, -25, 13, -11, 34, 10,
35, 42, 56, 47, 21, 33, -13, 2, -6, -22, 10, -16, 1, 17, -13, -21, 1,
-9, 22, -4, -5, 10, 1, 13, -12, -1, -15, 25, 8, 15, 33, 21, 37, 26,
-5, 13, 16, -12, -2, -19, -4, -15, -26, 17, 9, 19, -2, 15, -6, -3, 9,
5, 0, -18, 5, -2, 13, 34, 19, 31, 62, 41, 54, 41, 7, -5, -8, 9, -2,
2, -14, -8, 0, 19, -17, 18, -10, -15, 0, -22, 24, -10, 2, 7, 1, -4,
31, 41, 29, 25, 26, 7, 2, 14, -19, -25, -20, -34, -30, -22, -5, -4,
-5, -16, 14, -1, 22, -15, 2, -9, 13, -6, 1, -1, -17, -19, 3, -14, 21,
2, 0, -31, -1, -18, 0, -17, 11, -5, 18, -24, 4, -8, 4, 0, -17, -1,
-4, 16, -10, -13, 20, -7, 13, -15, -25, -28, -17, -14, -3, -37, -17,
5, -28, 12, -21, 7, -25, -10, 8, -2, -17, 15, -17, 1, 17, -11, -19,
-19, 20, -10, 6, -22, -27, -8, 6, 15, 6, 19, -24, -19, 4, 5, -24, 21,
24, -8, -10, 20, 5, 3, -2, -19, -16, -13, 3, -18, 4, 15, 1, 2, 16,
-4, -8, 3, -18, 3, -21, 23, -8, 11, -22, -18, 19, 10, -9, -18, -6};
    logic signed [7:0] oneweights[783:0] = {5, 16, -12, 3, 3, 3, 25,
```

5, -20, 11, 5, -1, 24, 5, 15, 16, 8, 21, -21, 25, 16, 25, 23, -15,
-2, -18, -10, 1, -23, 21, 24, -10, -2, -21, 5, 19, 13, -17, 16, 3, 4,
-1, 22, 10, 18, 19, 23, -23, 14, 5, -13, -21, 16, 21, 9, 0, 6, -12,
24, 15, 4, 11, -18, -2, -15, 3, -7, 1, 10, 3, -8, 21, 8, -19, -24,
-2, -2, 20, 10, 5, 5, -12, 25, 22, -5, 3, 5, -17, -10, -16, 15, -18,
-15, -25, 4, -18, -27, 9, -2, 8, -19, -4, -19, -4, -28, 13, 18, -10,
11, -12, 22, 17, -5, -24, 9, 22, 21, -4, -18, 0, 11, 5, -30, -4, 15,
-1, 21, 6, 18, 18, 0, -2, 43, 37, 14, 1, 17, 1, 1, -3, -15, 2, 6, 9,
-2, 10, -25, 10, -6, 5, -3, 7, 12, -3, 19, 8, 5, -28, 3, 14, 7, 12,
1, 28, -15, -16, 13, -24, -3, 14, -23, 22, -11, -17, 16, -17, -42,
-34, -18, -22, -18, 11, -5, -37, -23, -6, 19, 6, 35, 29, 4, -16, -13,
-11, -11, 9, -3, 5, -22, -9, 10, -10, -11, -13, -38, -54, -21, -57,
-25, -3, -19, -40, -18, -22, -16, -7, 11, -1, 13, -4, -24, -10, 4,
-5, 22, 7, 23, -23, 19, 16, 7, -5, -8, -27, -64, -53, -20, -20, 29,
20, 12, -11, -1, 14, 0, -8, -10, -9, 0, -14, -2, -4, 4, -12, -24,
-21, -12, -9, -8, -43, -6, -41, -47, -44, -34, -6, 24, 37, -2, -16,
-19, -42, -50, -30, -20, -10, -16, -10, -6, 23, 2, 16, 7, -12, 20,
-10, 2, 0, -3, -53, -34, -17, -14, 7, 94, 65, 50, 6, -27, -32, -56,
-50, -7, -4, -18, -3, 13, -12, 11, 6, 3, 3, -6, 0, -16, -24, -26,
-53, -31, -53, -27, 23, 127, 109, 41, -6, -14, -45, -25, -22, -35, 3,
-13, -5, -12, 13, -1, 6, 24, -1, -10, -3, 13, 9, -21, -31, -66, -60,
-31, 19, 116, 107, 22, -14, -23, -29, 5, -10, 0, -9, 2, 13, 17, 0, 6,
-4, -23, -10, 3, -5, -12, -15, -44, -14, -59, -68, -53, 78, 115, 80,
-1, -49, -56, -45, -4, -9, -31, 6, 11, -20, -17, -20, -2, 15, -23,
-21, 5, -23, -22, 6, -31, -52, -62, -79, -22, 75, 118, 56, -44, -75,
-61, -20, -28, -36, -21, 0, -9, -11, -25, 18, 10, -16, -7, 16, -3,
-21, -16, -17, -16, -40, -47, -40, -6, 85, 113, 43, -28, -40, -42,
-15, -41, 3, -19, -13, -4, 21, 19, 4, -25, 21, -8, -5, 7, -10, -18,
-26, -17, -57, -26, -40, 12, 86, 112, -8, -73, -54, -43, -21, -11,
-27, -2, -26, -11, -17, 18, 1, -8, 18, 4, 6, -14, -27, -26, -44, -18,
-47, -19, 9, 49, 72, 64, 5, -24, -46, -65, -15, -21, -25, -34, 8, -1,
20, -20, 7, 8, 11, -24, 20, -16, -1, -3, -18, -4, -26, -8, 38, 21,
56, 29, 15, -26, -59, -29, -43, -19, -19, -24, -21, 18, 19, -2, -8,
8, 5, -10, -14, -12, -39, -29, -7, -24, -16, -8, 9, 6, 35, 37, -9,
-19, -13, -19, -39, 0, 8, -24, -16, -17, -25, -18, 15, -19, -22, -20,
17, 19, 11, -36, -16, 1, -8, 24, 0, 7, 11, 11, 8, 8, -31, -28, -34,
-7, -12, 15, -25, -8, 9, -19, -4, 17, 13, 1, 24, -9, -1, -17, -8, 15,
0, 7, -17, -33, -17, -7, 9, -2, 13, -29, -11, -26, -13, -23, -15, 10,
9, 9, -20, 6, 23, -9, -8, 15, 13, -9, 2, 38, 16, 8, -36, -39, -15,

```
-8, 0, 33, 9, -3, -23, -26, 0, -3, 21, -6, -1, -14, -2, -2, 18, -18,
12, -15, -14, 5, 39, 1, -10, -3, -36, -16, -11, -17, 13, 14, -1, 7,
5, -20, 13, -15, -14, -4, -9, -17, -9, 16, -7, 4, 14, -16, 8, -11,
14, -1, -37, -19, -45, -37, -46, -25, -21, 1, -6, -11, 4, -4, 2, 15,
-12, 18, -23, 23, 8, -24, 22, 15, 7, -3, 14, 8, -10, 14, -8, -8, 2,
-11, -31, -10, -32, -4, 6, 13, 0, 22, 11, 21, -13, -19, -18, 21, -23,
23, 1, 0, 0, 10, -9, -6, 11, 0, 18, 18, -19, -25, 16, -9, -20, 20,
-16, -10, 17, 9, 21, -10, -21, 18, 15, -6, -20, -15, -19, 8, 10, 1,
-4, -11, 21, 7, -4, -6, -14, -9, 18, 23, -21, 10, 0, -10, 18, -6, -6,
-6, -13, 20, 7, 10, 16};
    logic signed [7:0] twoweights[783:0] = {0, -19, 11, -8, -11, -4,
1, -12, -20, 10, -15, -21, 20, -19, 7, 0, 17, -4, -7, -18, 10, -9,
-17, -15, 23, 10, 13, -14, 9, 21, 17, -5, 13, 17, -20, -2, -6, 21,
-19, -17, 6, 14, 17, 10, 24, 3, -13, 6, 16, 1, -7, -18, -4, 23, -24,
8, -5, 1, 3, -11, 21, -9, 23, -3, -3, -3, -21, 20, 2, 13, -3, -23,
-18, -26, -18, 17, 14, -13, 14, 3, -21, 13, 13, -4, 10, 23, -13, -11,
21, 13, 21, -6, -3, 13, -4, 35, 12, 30, 41, 9, -3, 8, 19, 7, 13, -16,
2, -20, 10, 11, -3, -8, -14, 23, -12, -18, 17, 19, 5, 14, 12, 36, 52,
43, 41, 45, 53, 37, 56, 10, 26, -7, -3, -28, -7, -11, -13, -8, -22,
-9, 18, -12, 20, -19, 12, -18, -8, 33, 34, 28, 62, 76, 47, 41, 74,
69, 44, 39, 40, -7, -17, 5, -17, -14, 1, 9, 1, -19, 25, -24, 14, 3,
-14, -16, 30, 35, 43, 52, 40, 35, 39, 38, 15, 39, 30, 18, 20, -5, 16,
-8, -6, -6, -31, -5, 17, -9, 1, -24, 6, -17, -4, -4, 38, 15, 27, 8,
25, 22, 12, 16, 3, -2, 21, 2, 6, -13, 11, -18, -5, -2, 6, -8, 12,
-12, -6, 23, -16, 25, 15, 21, 0, 13, 21, 20, 18, 17, 17, -6, 27, 13,
27, -6, -15, 4, -5, -12, -20, -5, -37, -15, -7, -10, -2, -23, -21,
-18, -7, 22, 7, -19, -13, -23, 7, -1, 0, -1, 11, 9, 2, 2, 25, -8, -6,
-9, -17, -2, -19, -12, -15, 18, 12, 10, -18, -15, 16, -7, -21, -14,
-47, -32, -32, -67, -48, -47, -29, -25, 18, 12, 17, 1, -12, -21, -6,
2, 14, 10, -16, -5, -15, 23, 9, 23, -9, 7, -28, -66, -71, -47, -75,
-67, -66, -66, -66, -10, -9, -8, -1, 2, 22, -2, -30, -20, -3, -24,
-8, 17, -2, -8, -3, -9, -4, -27, -48, -65, -60, -94, -66, -78, -93,
-69, -67, -43, -41, -36, -25, -5, 7, 2, -22, 1, 17, 1, 20, 6, 1, 22,
-23, -19, -26, -30, -41, -72, -51, -70, -81, -48, -54, -36, -43, -41,
-12, -17, -19, 5, -24, -5, -44, 4, -22, -3, -13, -1, 6, -16, 13, 23,
1, -27, -56, -32, -36, -30, -3, -24, 11, 12, 16, -38, -4, -5, -12, 5,
-26, -19, -34, -29, -15, 21, -11, 19, -6, 20, -10, -18, -8, 8, -34,
0, -12, 13, 14, 25, 15, 32, -2, 1, -22, 6, -33, -1, -30, 3, -10, 8,
-10, 27, 20, 19, -23, -6, 1, -20, 14, 12, 24, 20, 15, 16, 25, 48, 33,
```

```
38, 7, 26, 32, 21, 1, -2, -18, 3, -17, 7, 33, 22, 21, 12, 3, 9, 19,
-9, 24, 26, 31, 41, 21, 37, 15, 42, 54, 46, 47, 39, 16, 4, 1, 8, 20,
-7, 43, 38, 52, 46, -15, 21, -12, -23, -8, 8, 6, 29, 34, 36, 34, 39,
55, 41, 74, 49, 29, 6, 22, 31, 4, 25, 33, 16, 45, 46, 34, 11, -4, 7,
-19, -19, 7, 14, 34, 37, 27, 21, 53, 31, 46, 43, 30, 35, 15, 34, 24,
11, 1, 17, 29, 46, 48, 39, 33, 32, -11, -17, 18, 16, 4, -8, 35, 46,
35, 58, 36, 27, 40, 35, 18, -7, 17, -5, 17, 15, 31, 18, 21, 50, 60,
58, 39, -8, -4, 8, -21, -9, -17, -1, 4, 12, 27, 35, 40, 36, 12, 32,
-7, -2, 1, -28, -24, 24, 41, 28, 55, 50, 26, 40, 28, -8, -22, -6, -1,
-8, -1, -15, -2, -9, 19, 37, 6, 41, 11, 13, -18, -24, -27, -26, -1,
4, 38, 22, 9, 38, 21, 20, 7, -15, -1, 13, -16, 8, -5, -24, 1, 18, -9,
-31, -18, -25, -18, -11, -32, -22, -33, -51, -45, -22, -19, 6, -13,
7, 5, 23, 13, -14, 24, -8, -21, -24, 3, -25, -8, 14, 3, -18, -19,
-20, -41, -12, -44, -16, -22, -20, -12, -5, -12, 9, -28, 9, -20, 2,
-18, -9, -17, 17, 0, 19, 8, 7, -5, 17, 3, -17, -16, -16, 11, -9, 1,
-4, -8, -26, -13, 10, 3, -3, 18, 12, 18, 4, 5, -11, -20, -3, -1, -18,
-8, -9, 22, 11, -10, -4, -19, 19, 3, 18, -6, -23, -24, -1, -18, -17,
-8, -20, -14, 12, -4, 2, 16, -1, -18, 21, 12, 18, 7, -1, -12, 5, -9,
5, -10, 4, 19, -5, -6, -7, 23, -6, 7, -9, 0, -14, -15, -11, -5, 6,
-12, -8, -19, 23};
    logic signed [7:0] threeweights[783:0] = {-23, 22, 23, -20, 23,
-20, -22, 24, -4, 21, 5, -12, -4, 0, -7, -10, 15, 15, -16, -5, 22,
-9, 18, 3, 20, 18, -20, -2, 4, 2, 12, -6, -9, 11, -20, -16, 15, 5,
-11, -1, 1, 11, -11, 1, 21, -8, 17, 19, 8, 6, 0, 10, 4, 22, 20, -7,
-6, -4, 5, -20, -22, -3, -7, -14, -9, -17, -11, -14, -7, -22, 4, 8,
8, 1, -5, 24, 2, 19, 15, -19, 1, -24, -12, -7, 19, -9, -13, 19, -7,
-11, -22, -10, -12, 5, -2, 5, -6, 19, -4, -3, 17, 14, 8, 12, -11, 17,
-14, -22, 23, -8, 2, -22, -16, 9, -2, 18, 21, 10, 7, 23, 14, 22, 19,
29, 42, 17, 24, 17, 5, 12, -19, 1, 3, -22, -7, -25, 8, -11, -5, 4,
24, -11, 4, 21, 18, 20, 23, 9, 24, 57, 41, 41, 11, 12, 32, 14, 16, 6,
-23, -15, -31, -10, -34, -29, -18, 4, -1, -14, -18, -24, -1, -9, 32,
10, 38, 18, 40, 49, 29, 49, 35, 27, 20, 27, 39, 19, 28, -10, 0, -7,
-25, 4, -20, 7, 11, 4, 19, -13, -19, 13, -9, 30, 20, 17, 40, 23, 21,
8, 7, 12, 24, 40, 40, 6, 12, 26, 32, -11, 6, -11, -9, -3, -16, 22, 1,
1, 21, -16, 35, 29, 23, 26, -27, 7, -12, 7, -7, -4, 10, 43, 47, 17,
22, 18, 31, 27, -14, -25, -6, 12, 3, 5, 17, 18, 9, -8, 9, 7, 19, -5,
-13, -35, -42, -62, -37, -16, 8, 51, 20, 21, 40, 18, 15, 11, -22,
-30, -2, -1, 11, 5, 4, 4, -11, -1, 25, -2, 1, -32, -69, -72, -81,
-55, -61, 3, 27, 42, 60, 36, 24, 12, 22, 9, 7, -18, 16, -16, 13, 9,
```

```
-21, -21, -17, -16, 3, -8, -34, -61, -84, -56, -73, -51, -33, 32, 67,
45, 36, 25, 37, 32, -6, -9, -24, -31, 1, -1, 20, 9, -13, -13, -5, 12,
-22, -33, -44, -55, -34, -65, -35, 2, 7, 35, 64, 38, 31, 13, 30, 5,
-35, -8, -2, -31, 7, -6, 12, 17, 2, -24, 5, -25, -12, 12, -31, -56,
-57, -25, -40, 0, 21, 48, 35, 24, 26, 26, 6, -33, -10, -12, -32, -2,
-26, 15, -20, 2, 3, -6, 5, 19, -15, -32, -22, -36, -49, -23, -9, -5,
44, 19, 20, 2, -12, -5, 22, 17, 6, -6, -27, -25, -25, 11, -20, -17,
19, 14, -15, 23, 6, -26, 0, -17, -28, -53, -32, -5, -1, 5, -26, -2,
12, 4, 28, 18, 28, 3, 7, -12, 19, 21, 23, -15, 7, 4, -17, -18, -4, 6,
-35, -7, -52, -61, -34, -26, -18, -44, -35, -16, -6, 18, 8, 37, 14,
27, 28, -12, 1, -22, -20, 22, -17, 2, 16, 5, -1, -8, -23, -24, -47,
-42, -87, -85, -41, -72, -41, -25, 8, 22, 41, 29, 36, 41, -2, 26, 7,
13, -12, 19, 16, 11, 5, 13, 36, -2, 6, -6, -14, -55, -66, -83, -74,
-49, -68, -13, 18, 13, 23, 35, 12, 13, 35, -19, -13, 6, -4, -16, -16,
-2, -3, 36, 14, 49, 29, 2, -38, -18, -30, -38, -26, -47, -40, -4, 26,
34, 29, 46, 46, 35, 0, -4, -4, -25, 21, -21, -21, 1, 5, 31, 25, 23,
50, 18, 28, -18, -20, -31, -28, -34, 19, -5, 33, 39, 38, 25, 16, 28,
-10, -29, -4, -6, 19, 18, -5, 3, 23, -7, 27, 30, 48, 11, 25, 18, 15,
-25, -9, 13, -11, 21, 17, 32, 27, 31, 20, -19, 6, -2, 5, -3, 8, -4,
8, -15, 27, -10, 4, 28, 38, 51, 28, 24, 24, -2, -4, 24, 14, 2, -5,
16, 14, -13, 5, -11, -6, 15, -26, -22, -17, -2, -19, 18, -16, 16, 5,
12, 17, 32, 23, 36, 44, 5, 14, 18, -14, 24, -18, 18, 17, 6, 13, -19,
-22, 18, 20, -21, 3, -18, 22, -20, -19, 6, -14, 16, 37, 36, 28, 51,
27, 19, 30, 47, 24, 10, 5, -18, 14, -23, 15, 18, -20, 11, 7, 12, 0,
11, -17, -21, -13, 0, 1, -19, 22, 6, 23, 2, 21, 8, 16, 29, 6, -10,
11, -5, 0, -3, 17, 11, -11, 17, -7, -4, -7, 9, 3, -11, 24, 22, -2, 6,
-23, 3, -19, -14, 15, 10, -12, -4, -5, 3, 3, 12, -10, 20, 13, 0, 4,
-23, -8, -24, -20, 19, 17, 0, 13, -1, 18, 2, -21, 19, 16, 0, -11, -8,
-23, -8, 21, 14, -9, -12, -1, 13, 17, -10, 1, -5, -23, 10, 1, 19};
    logic signed [7:0] fourweights[783:0] = {3, 12, -9, 7, -19, 5,
22, 11, 10, 7, -18, 3, 21, -13, 12, -2, 0, 3, -14, -20, -24, 7, -17,
15, 14, -7, -2, 18, -14, 24, -24, -9, -13, 6, -21, 20, -1, 9, -11, 9,
-11, 4, 17, -17, 17, -7, 6, 2, -18, 20, -24, -23, 14, -13, 4, 3, -16,
15, -7, -6, 6, 16, -24, -5, 0, -11, 0, 10, 9, -23, -26, -31, 0, 15,
-1, 8, 11, -19, -18, 9, -18, -19, -4, -14, -2, -6, -22, 10, 17, -1,
-24, 21, -3, -10, -18, 8, -31, -11, -3, -19, -28, -30, -3, 4, -2, 6,
-15, -16, -4, -12, 17, 4, 2, 8, -16, 7, 7, 11, 5, 1, 4, -8, 0, -25,
-49, -10, -38, -33, -24, -25, 3, 2, -5, 6, 23, 13, 14, 3, -20, -4,
-19, -24, 13, -12, -17, 0, 25, 13, -24, -4, -1, -5, -10, -19, -35,
```

```
-26, -8, -2, 17, 5, 25, -8, 28, 11, -15, 16, 24, -20, -18, 9, 3, -23,
10, -19, -2, 18, -20, 13, -1, -37, -22, -48, -77, -46, -47, -37, -17,
10, 24, 3, 32, 37, 14, -5, -1, -21, 21, 5, -24, 9, -11, -5, 3, -7,
12, -7, -23, -54, -54, -75, -69, -79, -56, -51, -32, -12, -22, 34,
28, 4, -1, 8, -1, 14, -16, 2, -18, -14, 10, -8, 8, -19, -16, 0, -37,
-19, -29, -75, -103, -78, -64, -31, -15, -31, 12, -4, 33, 5, -11,
-22, 10, 5, -7, 15, 8, 19, -16, -17, -6, -8, -12, -26, 6, -22, -19,
-25, -70, -88, -16, 4, -17, 9, 9, -18, -4, 4, -5, -21, 19, -1, 13,
14, 11, -20, -14, -24, -5, 0, -17, 19, 25, 6, -3, -17, -84, -29, 3,
13, 14, 7, -18, -24, -8, -3, 7, -4, 3, 21, 6, -21, 20, 19, -22, -19,
-19, 15, -7, 17, 45, 24, 38, -41, -37, -19, 26, 52, 28, -2, -11, -25,
-26, -23, -2, 9, -6, 3, 8, -10, 6, -27, -8, -5, 21, 34, 47, 59, 70,
67, 35, -35, -60, 7, 53, 25, 22, -3, -21, 3, 8, -19, -8, 21, -3, -21,
-20, -17, -20, 17, 17, 29, 33, 28, 48, 65, 77, 66, 21, -4, -13, -8,
40, 76, 31, 3, 21, 4, 8, 8, -8, -12, -20, 6, -20, -25, -19, 8, -17,
2, 40, 70, 57, 51, 65, 63, 41, 12, 12, 23, 68, 77, 49, 48, 33, 9,
-12, -1, 1, 6, -17, 10, 17, -10, -8, -6, 18, 0, 44, 42, 84, 45, 61,
40, 47, 6, 13, 40, 76, 59, 54, 38, 16, 10, -9, 13, 0, -12, 19, 1,
-13, 7, 7, -25, 15, -2, 42, 54, 53, 51, 37, 20, 13, 20, 70, 83, 85,
76, 54, 15, 18, -19, -5, -18, -7, 6, 23, -14, 8, 16, -2, -21, 3, -20,
12, 34, 21, 21, 29, -15, 8, 45, 66, 71, 65, 40, -6, -20, 1, -7, -30,
-12, -24, -24, 10, -12, 9, 8, 0, 4, 6, -10, -14, -22, -6, -20, -25,
-31, 9, 27, 35, 36, 31, 4, -23, -14, -23, -33, -4, 0, -22, 10, 21, 4,
21, 18, -21, 20, -15, -10, -46, -50, -51, -54, -63, -59, -38, -31,
14, -2, -1, -4, -31, -29, -27, -22, -31, -8, -19, -20, 20, 18, -11,
24, 11, 18, -2, -23, -35, -50, -72, -40, -69, -38, -38, -43, -16, 10,
-19, -28, -21, -28, -29, -2, -7, -13, 17, 15, 10, -5, -1, 0, 23, 0,
4, -1, -30, -58, -36, -44, -60, -36, -26, -34, -39, -22, -19, 0, 18,
-13, 16, -22, 14, 18, 8, 6, 4, -5, 6, -8, 14, 22, 13, -16, -37, -40,
-6, -47, -55, -23, -14, -3, -38, -12, -18, 2, 24, 3, 24, 18, 13, 3,
-11, 24, -23, 7, 19, -9, -24, 0, 5, 13, 3, -18, -31, -24, -12, -19,
-5, -6, 0, -17, 0, 20, 1, 38, -6, -1, -13, -21, 24, 21, 19, 19, 22,
10, -5, 8, -16, 6, 5, -30, -15, -3, -13, -8, -23, -8, -31, -8, 4,
-13, -9, 4, 13, 16, 16, -20, -12, -4, 17, -20, -15, -7, -1, 7, 5,
-24, 22, 6, 0, -19, -1, -44, -40, -29, -4, -5, -21, -37, -35, -24,
-12, -11, -20, 13, 21, 18, -9, -9, 23, 24, 14, 3, -22, -8, 17, -13,
-28, 1, -2, 0, -31, -17, 3, -11, -26, -35, -22, 0, 9, 3, 6, 5, -19,
11, 23, 2, -17, 24, 21, -24, 15, 21, -3, 11, -15, 8, -22, 18, -18,
-9, 12, 20, -10, 13, -21, 17, 3, -3, -7, 17, 14, -11, 10, 14};
```

```
    logic signed [7:0] fiveweights[783:0] = {7, 24, 11, 25, 11, -19,
-12, 20, 18, -14, 22, -20, 3, -23, 25, -24, -21, -18, -6, 12, 21,
-20, -1, -25, 6, -3, 17, 1, 9, -20, -10, 2, 18, -22, 9, -21, -19,
-12, 4, -12, 19, -6, 21, -3, 11, -23, 13, 0, 25, -3, -21, -13, -17,
-4, -17, -11, -17, -14, -12, 12, 5, 2, 22, 4, -12, 1, -12, -2, -3, 7,
4, -16, -21, -18, -13, 8, -24, 1, -18, -7, -15, -8, 4, 15, 17, 7, 1,
9, 5, -16, -8, -25, -3, 12, 1, 15, 11, -23, 1, -11, 8, 10, 7, 21, -6,
-15, -16, -24, 21, 14, 23, -23, 20, 15, -16, -10, 11, -12, -10, -10,
-21, -15, -12, -33, -5, -32, -9, 5, 18, -15, 23, -19, 15, 20, -1, 33,
-17, -20, 17, -13, 16, 23, 21, 14, 15, -3, -5, -34, -32, -3, -1, 15,
-27, -16, -11, 12, 9, -14, 19, 27, 17, 35, 52, 32, 11, 23, -15, 21,
17, 19, 9, 21, 6, -20, -11, -3, 6, 12, 18, -3, -2, 4, -10, -5, 9, 7,
30, 27, 39, 22, 37, 66, 45, 15, 28, 2, 17, -11, 17, -8, 10, -31, -37,
-1, -4, 21, 40, 18, 24, -17, -3, -19, -12, 4, 2, 10, 19, 39, 74, 62,
67, 30, -4, 10, -20, -17, -21, 1, -11, -13, -4, -9, 16, -2, 27, 37,
26, 16, -17, -32, -11, -13, 5, 12, 27, 49, 35, 70, 74, 71, 30, -24,
-23, 6, -6, -2, 10, 0, 0, 11, 23, 42, 18, 45, 20, -10, 12, -22, -38,
-9, -34, -25, 16, 28, 29, 77, 54, 24, -8, -9, 11, 18, -4, -16, -28,
6, -21, 21, 37, 27, 53, 47, 53, 39, 13, -24, -22, -55, -51, -26, -43,
-10, 12, 15, 47, 37, 26, 22, 24, -13, -21, 20, -20, 3, 14, 34, 25,
32, 65, 43, 55, 69, 35, -13, -29, -63, -48, -79, -73, -67, -44, -40,
-21, 24, 18, -24, 4, -19, 11, -18, -5, 23, 1, 10, 22, 45, 64, 97, 92,
37, 9, -3, -48, -65, -47, -79, -63, -56, -57, -50, -11, 11, -11, -8,
6, -10, 6, 7, -5, -5, 12, 25, 31, 19, 58, 52, 58, 29, 8, -50, -42,
-31, -23, -8, -52, -22, -45, -20, -1, 10, 23, -14, 4, -15, 22, -21,
-10, 11, -21, -18, 17, 48, 56, 52, 52, 24, 0, -55, -41, -31, -22,
-23, 2, -23, -15, -2, 11, -21, 22, -25, 24, -15, -23, 21, 0, -22,
-26, -26, -23, -5, 29, 21, 7, -32, -51, -31, -37, -22, -27, 10, -21,
-3, -23, 0, -11, -9, 12, -4, 12, -2, 4, -9, 2, -24, -6, -18, -47,
-37, -22, -14, -21, -43, -30, -19, -34, -30, 4, -16, 5, -20, -18,
-25, -1, -20, -5, -5, -5, 5, 0, 18, -8, -17, -5, -15, -54, -63, -66,
-69, -66, -35, -26, -37, 6, -15, 23, 9, 0, -3, 2, 15, -18, 19, 16,
-12, 14, -21, 11, -22, 17, -2, 25, 13, -29, -25, -31, -58, -16, -25,
7, 0, 5, 27, 20, -2, 2, 3, -1, 11, -2, -15, 9, -3, 7, 16, -20, 1, 4,
28, 27, 42, 42, 1, -25, -1, 6, 1, -5, 1, 2, -13, 20, 16, 25, 20, 22,
6, -1, 7, 3, -12, 18, 21, -12, -19, -14, 20, 16, 40, 32, 48, 38, 28,
-11, 4, 6, -11, 22, 0, 4, 25, 13, 2, -4, -4, 22, 9, -20, -8, -15, 17,
2, 20, 2, -3, 5, 40, 37, 36, 28, 8, 15, 5, 18, 6, 14, 29, 9, 14, 20,
33, 34, 1, -1, -24, -8, 18, -22, 13, -15, 0, -4, 14, -24, 0, 22, 32,
```

```
26, 30, 42, 11, 10, 16, 5, 31, 7, 27, 28, 14, 22, -4, 25, 20, -24,
-19, 20, -1, -19, -7, 11, -10, -4, 19, 4, 31, 38, 39, 18, 32, 25, 32,
22, 21, -5, 19, -11, 22, -1, 25, 19, 2, -24, 11, 0, 6, 7, -20, -14,
-18, -2, -21, -3, 23, 6, 17, 39, 42, 28, -1, 5, 9, -17, -10, -7, 3,
2, 9, 6, 14, 2, -15, 20, 7, 12, 1, -4, 1, 10, -17, 11, -18, -1, 17,
23, 26, -13, -23, 12, -11, -19, 9, -19, -1, 2, 24, 20, -11, 19, 24,
10, -23, -11, 22, -9, -20, -21, -10, 21, -3, -21, -23, 17, -18, 21,
-14, -18, -16, -1, 10, -10, -14, 22, 15, -9, -11, -10, 19, 17, 10,
-24, 16, -17, -4, 3, -23, 7, 10, -24, -18, 22, -13, 13, 19, 23, 5, 7,
15, 3, 10, -6, -4, 15, 15, 3, -10};
    logic signed [7:0] sixweights[783:0] = {-3, -7, 7, -23, 1, 7, 24,
8, 8, -13, -8, -5, -9, 8, -8, 23, -10, -19, 20, 13, 2, -9, -18, 6,
-19, -4, 5, 17, -20, -16, -16, 22, -2, 23, -16, 20, 15, 11, -5, 8,
-8, -12, 25, -13, 1, 10, -15, -6, -19, 24, 0, -23, -22, -5, -4, -5,
-1, 21, 5, -13, -22, -7, 20, 11, -17, 6, 22, -8, 27, 40, 40, 2, 6, 8,
39, -1, 4, 4, -12, 11, -18, 3, 7, -20, -18, -13, 4, -22, 4, -21, 1,
4, 27, -16, 23, 32, 31, 21, 20, 26, 23, 69, 64, 42, 23, 35, -1, 24,
-20, -13, -11, -19, -1, 16, -8, 24, -18, 14, -4, -7, -2, 8, 2, -6, 2,
19, 29, 23, 43, 46, 34, 72, 19, 14, 31, 18, -1, 3, -17, 13, -18, -11,
10, 18, -10, 18, 1, -23, 10, -21, 27, -26, -14, -20, -23, -23, 3, 14,
40, 23, 30, 29, 12, 17, 18, -7, 2, 17, 7, -1, -19, 6, 23, -28, 13, 8,
7, -6, -16, -9, -34, -7, -27, -9, -3, -24, -15, -26, -12, -35, -2,
-34, -21, 1, 15, -8, 12, 0, -21, -17, -14, -5, -26, -12, -15, -21,
-14, -12, -38, -26, -33, -11, -44, -44, -42, -45, -42, -52, -15, -18,
3, -10, 10, -8, 6, 23, 15, -17, 15, 9, -10, -28, 3, 15, -31, -30,
-23, -20, -38, -47, -54, -70, -92, -63, -69, -44, -25, -23, -4, 18,
-1, -7, 12, -10, 16, 16, -11, -31, 11, -14, -21, -25, -15, 7, -17,
-12, 5, -20, -45, -72, -103, -87, -43, -62, -14, -32, -18, -5, 7,
-24, 19, 9, -15, -25, 6, -1, -4, -12, 13, 19, -10, -13, -8, 10, 1,
-68, -76, -62, -55, -47, -35, -44, -6, -28, -30, 0, 0, -17, 18, -22,
15, 0, -21, -9, 18, -9, -13, -1, -6, 7, 1, -4, -32, -70, -41, -51,
-40, -26, 2, -23, -11, -2, 17, -2, 0, 17, -9, -11, 24, -25, -18, -6,
24, 2, 35, 10, 18, 11, 50, 23, -12, -22, -22, -21, -9, -26, -22, 34,
38, 35, -1, 3, 9, 7, 14, 8, 1, -20, 20, 10, -9, 14, 13, 17, 17, 47,
13, -24, -23, -17, -13, 1, 8, -16, 37, 22, 44, 54, 14, -7, -19, 22,
22, -21, -8, -1, 5, 6, 30, -6, 32, 41, 39, 38, 8, 6, -10, 39, -2, 1,
19, -5, 24, 31, 27, 48, 27, -12, 14, 21, -17, -18, -8, 5, 7, 6, 15,
31, 26, 25, 59, 54, 8, -9, 14, 24, 26, -18, -24, 2, 9, 33, 20, 15,
35, 11, 2, -24, 3, -20, -14, -21, -21, -2, 0, 1, 28, 36, 73, 38, 15,
```

```
-3, 19, -11, -3, -6, -8, 34, 18, 13, 38, 24, -6, -22, -13, -17, 10,
24, 2, 10, -18, -6, -8, 32, 40, 43, 70, 78, 50, 3, 14, 0, -5, -16,
20, 24, 43, 17, -6, 11, 5, 10, 0, 21, 22, 1, -15, -17, 2, -32, -1,
-10, 24, 59, 74, 51, 56, 1, 22, -16, 21, 10, 27, 28, 20, 29, -13, 14,
-11, 12, -25, 18, 10, 6, 21, 8, -33, -39, -10, -34, 14, 30, 81, 83,
75, 39, 25, 26, 47, 57, 31, 12, 2, -12, 2, -18, -22, 1, -2, -10, -15,
19, 22, -10, -17, -25, -26, -37, 7, 30, 44, 83, 77, 57, 47, 63, 52,
52, 45, 3, 19, -7, 8, -13, 10, 4, -4, 9, -15, 24, -21, 1, 7, -27,
-27, -17, -13, -10, -3, 25, 52, 76, 63, 35, 42, 15, -7, 2, -30, -38,
11, -21, 20, 5, 18, -21, 11, -17, 10, -8, -12, -32, -19, -8, -23,
-44, -15, -9, -8, 8, 0, -12, -10, -36, -40, 2, 6, -11, -2, 16, 22,
-7, -20, -17, -18, -8, 24, 11, -16, -26, 13, -27, -43, -47, -52, -64,
-42, -65, -35, -62, -54, -35, -39, -2, -17, -19, -4, -19, 2, -9, -20,
-9, 12, 6, 3, 21, 17, 8, -26, -22, -7, -7, -37, -43, -15, -39, -34,
-9, -8, -37, 12, -21, -13, -7, 9, 11, 6, -10, -12, 1, -21, -10, 18,
-13, -12, 10, -21, 4, -14, 8, -17, -16, 0, -16, -4, -32, 0, -25, -5,
-13, -9, -18, 15, -7, 4, 12, -22, 10, -19, 25, -23, -19, -9, 6, -16,
20, 2, -8, -6, 3, -21, -19, 12, 7, 12, -10, -17, -16, -12, -1, 19,
-5, -3, -3, 2, -12, -17, 8, 23, -4, -3, 5, -19, 20, -7, -21, 0, -23,
-6, 0, 21, -15, 2, -5, -5, -6, -3, 23, -1, 9, -19, -8, -12, 2};
    logic signed [7:0] sevenweights[783:0] = {-13, -21, -4, 23, 22,
-1, 5, 12, 17, 18, 5, -19, 2, 5, -18, -15, 11, -3, -9, 25, 2, 20, 13,
-1, -8, -22, 23, 10, -11, 22, 24, -16, 10, 14, 0, 5, 6, 16, -7, 4,
14, -9, 14, -6, -17, -17, -6, 10, -5, -22, -10, 24, -5, -17, 22, -21,
2, -19, 18, -3, -20, -24, -20, -18, 16, 9, 22, -15, 12, -12, -6, 2,
-22, 13, -22, -17, 11, 3, 11, 18, -24, -2, -19, -20, 11, 22, 15, 13,
-14, 9, -18, 17, -10, -26, -6, -22, -7, 13, -6, -27, 13, -7, 6, 8,
18, -18, -23, 5, -1, -10, -23, 20, -4, -3, 21, 15, 14, -17, -23, -15,
-13, -13, 7, -19, -3, -37, -43, 0, -21, -2, -28, -2, -18, -28, 22,
-9, 8, -10, -22, -13, -3, -13, -5, 17, 12, -24, -18, 3, -25, -16,
-20, -21, -30, -25, -70, -26, -36, -50, -28, -41, -21, -34, -3, -25,
12, -4, 4, -13, 6, 5, -7, 18, 21, 1, 25, -20, 11, 23, 6, 7, -20, -42,
-49, -53, -40, -37, -19, -46, -22, -2, -14, -10, -17, -21, 6, -21, 8,
-18, 10, -1, 26, 11, 32, 19, 17, 19, 45, 25, 29, -2, 8, 18, 2, 18,
16, -7, 11, -7, -5, 7, -24, -2, -25, -5, 17, 1, 27, -3, 15, 39, 22,
15, 34, 38, 56, 47, 33, 33, 31, 54, 64, 65, 52, 13, 21, 2, -6, 13, 3,
-25, -12, -11, -22, -10, 0, -7, 16, 39, 33, 22, 51, 16, 43, 43, 37,
49, 52, 82, 64, 87, 70, 47, 11, 13, 8, -25, 3, -12, -24, -5, 0, 24,
10, 7, 43, 28, 39, 6, 13, 17, 19, 17, 37, 26, 34, 49, 75, 71, 48, 57,
```

```
39, 15, 2, 0, -18, 18, 7, -7, -6, 4, 9, 23, 22, 13, -7, 36, -1, 16,
27, 13, -20, 20, 22, 28, 71, 79, 51, 44, 17, -1, 7, -20, 8, -20, 10,
17, 6, -14, 4, 13, 2, 12, 22, 14, -4, -7, -34, -54, -68, -46, -41, 0,
32, 36, 33, 36, 15, 14, 16, 8, 0, 18, -15, -19, 1, -14, 26, 16, -9,
-10, 21, -16, -26, -41, -41, -74, -110, -83, -87, -48, 23, 27, 17,
11, 4, 3, -10, 9, -3, -16, -22, 10, -5, -14, 15, 23, 17, 0, -24, 4,
-47, -61, -93, -82, -91, -114, -61, -38, 9, 44, 44, 50, 39, 11, 24,
2, -21, -11, -24, 6, -12, -4, 19, -23, 19, -24, -30, -3, -30, -59,
-58, -72, -85, -77, -35, 16, 29, 43, 54, 49, 3, 27, 18, 13, -25, 13,
21, 20, -3, -21, 13, 23, 22, 13, -22, -31, -26, -28, -51, -38, -64,
-26, -1, 10, 22, 47, 7, 10, 23, 6, 7, 7, -27, -24, 4, -11, -8, -8,
21, -10, -1, -20, 0, -37, -54, -42, -63, -40, -26, -10, 16, 41, 24,
28, 15, -7, -15, -7, -15, -23, -15, -22, 3, -16, 19, -2, -16, -18,
-24, -8, -48, -40, -21, -31, -41, -43, -17, 24, 44, 31, 17, -20, -10,
-23, -46, -26, -23, 6, -28, -8, -7, 1, -20, 3, -23, -21, 2, -9, -19,
-12, -50, -47, -49, -11, 19, 15, 27, -4, -31, -38, -42, -39, -16,
-17, -38, -2, 14, -24, 10, -8, -13, -23, 24, 1, 13, -4, -34, -51,
-49, -53, -30, -31, 26, 18, 10, -18, 6, -37, -43, -18, -47, -17, -19,
-29, -22, -16, 6, 12, -21, 15, 14, -19, 4, -2, -15, -18, -15, -31,
-27, -24, -15, 3, -12, -15, -32, -24, -26, -27, -12, -27, -2, -24,
22, 16, -7, -1, 24, 15, 2, -4, -5, -29, -7, 5, -33, -11, 12, -28, -1,
-10, -11, -23, 7, -3, -25, -35, -36, -9, -19, 8, -13, 13, -19, 15,
-1, 4, -16, 13, 14, 9, -2, 10, 29, -8, 22, 6, -3, 14, 5, 13, -23,
-24, -8, -33, -19, -1, -3, 17, -25, -15, 14, -4, 12, -10, 3, 11, -6,
14, -11, 32, 42, 15, 14, 27, 26, 39, 34, 21, 17, 2, 6, -1, -8, -20,
-8, 2, 21, 2, -2, -2, -15, -7, -19, -2, 14, 4, 26, 47, 24, 32, 40,
58, 45, 52, 62, 49, 22, 14, 7, 5, -2, -7, -9, 17, 0, -6, -10, -12,
-21, -24, 20, 0, 1, -18, -12, 21, -4, 7, 5, 27, 22, 25, 34, 23, 31,
31, -13, 13, 18, 7, 25, -24, -15, 12, -16, -9, -10, -13, 22, -2, -7,
15, -20, 0, 12, 16, 16, 9, 26, -8, 7, -12, -19, 19, 1, -19, -3, 2,
-1, 5, -2, 14, 18, -7};
    logic signed [7:0] eightweights[783:0] = {-3, 6, 12, 20, 0, -21,
-24, 12, -8, -15, -22, 16, 14, 5, 11, -2, 4, 12, 1, 5, 20, 6, 5, 4,
-18, -8, 12, -13, -7, 18, -7, -9, 3, 0, -17, -10, 18, 21, -3, -10,
-21, -15, -20, 18, 11, 6, 19, -10, -19, -4, 22, 18, -14, -16, 15,
-19, -4, 1, -10, 9, -12, 17, 10, 7, 7, 20, 18, 1, 5, -6, 11, -23, 3,
2, 20, -22, -3, -9, -10, 12, -19, 2, 24, -14, -15, -14, -5, -20, -6,
21, 12, 7, -25, 8, -16, 3, -10, 6, 6, 5, -11, 7, -25, -2, 17, -7,
-23, 22, -21, 16, -15, -20, -18, -10, 1, 7, 5, 14, 21, -16, 4, -27,
```

2, 6, -32, 4, -1, 15, -13, -2, 8, -18, -5, -14, -4, -12, 14, 15, -2,
5, -24, 18, -4, 16, -24, -25, -6, 6, -25, -1, 8, -5, 18, 0, 24, 39,
31, 17, 13, -6, -19, 4, -16, 15, 19, 7, -4, 0, 21, 13, -24, -13, -7,
-19, 6, -19, -16, -15, 8, 14, 24, 18, 4, 53, 28, 37, 27, 1, 25, 7, 8,
5, -3, -5, 18, -7, -3, -9, 3, -13, -8, 3, -19, -17, -1, 26, -7, 10,
14, -15, -17, 17, -15, -28, 12, -6, 14, 4, -7, 14, 3, -20, -5, -1,
-13, -22, -10, 15, 17, -24, 7, 20, 3, 15, 3, 27, -8, 1, -31, -2, 8,
-16, -8, 16, 25, 14, 41, 12, 23, 7, -20, 4, -11, -11, -22, 18, 18,
12, 4, 8, 57, 30, 13, 34, 20, 8, -36, -54, -36, -17, 13, 13, 21, 35,
35, 41, 2, 17, 0, 6, 21, 1, 8, -12, -1, 20, 41, 38, 19, 43, 53, 35,
24, 2, -4, -50, -15, -35, -10, 3, 3, 44, 35, 6, -2, -18, -2, 0, -10,
9, 15, 3, 12, -7, 19, 41, 33, 45, 7, 9, 38, 15, 14, 7, -37, -7, 26,
15, 30, 43, 42, 37, 25, -12, 18, -4, 23, -23, 10, -14, 24, 24, -19,
-4, -4, 29, 1, 23, 55, 68, 38, 27, -3, 6, 13, 11, 38, 5, 37, 26, 6,
23, -18, -12, 18, -14, -23, 18, -24, -28, -5, 1, 4, -3, -4, 7, 61,
52, 17, 19, 10, -9, -10, 7, 1, -4, -14, -23, -12, -1, -15, 11, -2,
13, -20, -1, 5, 12, -37, -31, -46, -60, -37, 8, 61, 62, 37, 16, 2,
-16, -18, -23, -48, -38, -13, 1, 6, -10, 10, -20, 13, -24, 16, -10,
-18, -1, -10, -56, -49, -35, -1, 26, 60, 47, 29, 37, 3, -31, -48,
-52, -34, -30, -43, -9, -1, 19, -2, 22, -1, -1, -18, 12, -7, -17,
-17, -20, 1, 5, 20, 20, 54, 37, -2, -8, -21, -13, -48, -44, -19, -17,
-10, -10, -8, -20, 16, -1, -13, -2, -22, 20, -29, -4, -26, 12, 23,
47, 32, 28, 74, 12, 5, 8, 6, -17, -14, -8, -39, -24, -36, -3, -24,
11, 18, -3, -16, -2, -12, -20, 9, -10, -21, -11, 27, 34, 62, 64, 33,
14, -28, 5, -6, -8, -9, -21, -25, -12, 2, 3, -23, 21, 13, 2, -19, -7,
22, -28, -34, -23, -15, 4, 38, 13, 32, -14, 1, -22, -11, -38, -38,
-28, -16, 0, -13, -18, 2, 9, 4, -16, -6, -19, -21, -3, 21, -3, -4, 9,
0, 7, 3, 27, 4, 2, -20, -14, -49, -11, -2, -12, 5, 18, -9, -15, -3,
9, 18, 8, -6, -1, -16, -8, 8, -22, -31, -6, -6, 19, 14, 8, -1, -2,
-3, 19, 7, -26, -21, -15, -15, 19, 24, 14, 7, 3, -9, -1, -16, -23,
-22, -16, 10, -20, -11, -6, -11, 24, 2, -9, 8, 17, 36, 43, 15, 24,
23, 5, 39, 2, -4, 7, -7, -22, 5, -1, 18, -7, 1, 5, 6, 4, -16, -19,
-4, -33, -4, 26, 6, 32, 70, 87, 54, 39, 36, 16, 0, 24, -1, 22, 12, 4,
15, 23, -9, -12, -14, -23, 0, -2, -19, -13, 5, -11, -33, 0, -9, 23,
15, 24, 25, 33, 25, 33, 25, 31, -20, -18, -7, -9, 1, -14, -10, -19,
20, 15, -19, 22, -20, -10, -29, -30, 2, -15, 0, -20, -28, -7, 16, -6,
-15, -20, -5, -18, -9, -15, 7, 16, -4, -3, 24, 16, 10, -16, -23, -9,
17, 8, -23, 1, 14, -22, -15, -29, -17, -28, -13, 11, 2, 6, 4, -4, 18,
-17, 2, -17, -8, 16, -7, 17, 21, -21, -5, 23, -6, -18, 22, 21, 21,

```
11, -10, -14, -23, -10, 6, 10, 13, -24, 17, 3, 14, 18, 22, -17, 2,
20, -2, -10};
    logic signed [7:0] nineweights[783:0] = {4, 14, -8, -10, 0, -24,
0, -15, 18, -4, -9, -22, 2, -22, -4, -1, -4, -16, 3, 19, -22, -7, 24,
19, -16, 9, 16, -15, -13, -23, -13, -16, 1, -2, 0, 13, -2, 11, -2, 0,
18, -3, 14, -7, -23, -16, -1, -14, 2, 15, 12, 18, 24, -12, -10, -3,
-10, 23, -23, 24, -15, 19, 7, 24, 13, -15, 14, -9, 5, -16, -9, -19,
11, -12, 4, -8, -17, 22, -18, 24, 19, -24, 4, -6, -13, -11, 18, 14,
-20, -8, -6, -7, 6, -16, 13, -5, -11, 5, -24, -19, -6, 0, -14, 18,
19, -19, -15, -16, 24, 22, 20, -6, 9, -7, 14, -24, 14, -15, 14, -10,
-27, 11, 0, -5, -26, -32, -29, -44, -25, -38, -9, -13, -25, 4, -8,
15, -14, -6, 8, 6, -2, 11, -9, -18, -6, 19, 0, 15, -36, -32, -24,
-27, -42, -19, -49, -62, -63, -64, -44, -46, -50, -4, -14, -4, -17,
13, 19, -17, 20, 11, -5, 16, -16, 7, -11, -27, -6, -10, -38, -4, 14,
4, 30, 30, 25, -2, 11, -21, -44, -16, -41, -41, 5, -5, -12, 22, -4,
24, -21, -5, -16, -24, -39, -25, -38, -34, -5, -12, 44, 49, 101, 106,
100, 54, 23, -7, -19, -30, -19, -15, -13, -31, -26, 11, -4, -6, -6,
-24, -2, -3, -26, -30, -34, -11, -17, 21, 15, 48, 56, 54, 38, 30, 3,
-10, -25, -31, -53, -38, -19, 10, 7, -4, 5, 15, 17, 2, -30, -21, -8,
-33, 4, 3, -4, 25, 4, 17, 22, 22, 20, 0, 4, 8, 6, -8, -5, -41, -8, 4,
-24, -4, 15, -1, -7, -11, -24, 10, 14, 27, 35, 38, 52, 49, 10, -14,
-20, -31, -24, 2, 8, 4, 30, -10, -24, 12, -11, -17, -21, 1, -23, -11,
10, -16, 9, -2, 26, 41, 20, 38, 33, 3, 10, -35, -1, -4, 12, 55, 56,
64, 33, 21, 20, -11, -5, -14, 12, 15, -1, 20, -6, -1, 31, 21, 23, 39,
15, 43, 46, -6, -25, -14, 15, 30, 45, 63, 43, 70, 29, 41, -7, 17,
-27, 9, -3, 23, 8, -14, 5, 5, 26, 7, 44, 52, 46, 16, 0, -4, 7, -2,
17, 44, 50, 66, 65, 38, 20, 0, 17, 15, -16, -21, -23, 19, -13, -24,
2, 25, -13, 17, 35, 3, 13, 6, 6, 16, -14, 24, 15, 29, 54, 68, 52, 23,
-1, -14, -1, -32, -5, -13, -13, -23, 17, 9, 10, -18, -9, 27, -8, 28,
31, 12, 0, 13, 4, -1, -30, -9, 63, 34, 31, 9, -15, -10, -38, -39, 1,
-19, 13, 7, 1, -2, -5, -8, 6, -14, -19, 5, 11, 39, 14, 6, 1, -15,
-28, 23, 33, 26, -22, -16, -38, -26, 2, -36, 1, 3, -22, -16, -5, -6,
-20, -5, 15, -27, -6, 6, 11, -1, -1, 25, -20, -21, 3, 33, 31, -5, -3,
-15, -42, -32, -41, -31, -17, -8, 14, 15, 7, -10, -12, 6, 8, -30,
-28, -35, 1, -10, -24, -31, -15, -4, 5, 18, 21, 6, -27, -19, -34, 5,
-13, 5, -24, -4, -14, -19, -8, 15, 17, -16, 1, -20, -12, -21, -40,
-63, -27, -53, -52, -29, -4, -3, -24, -43, -29, 5, -6, -34, -23, -23,
20, -25, 13, -23, 0, -13, -15, 11, -27, -8, -26, -37, -38, -42, -76,
-41, -48, -38, -35, -20, -16, -6, -31, 3, -13, -1, -7, -24, -3, -11,
```

```
-19, -7, 12, -18, -5, 14, 4, 7, -6, -39, -37, -57, -65, -41, -54, -8,
-28, -23, -35, -35, -5, 4, -25, -21, -23, 5, -19, 10, 19, -21, -10,
13, 8, 9, 17, -27, -3, -25, -21, -15, -45, -14, -16, -21, -50, -58,
-10, -26, -15, -16, 18, -8, 3, -19, 16, 11, 11, -21, -1, -8, -20, 8,
21, -28, -30, -31, -36, -7, -2, -30, -31, -7, -17, -31, -38, -20, -3,
-8, 33, 25, 9, -11, 10, -17, -21, 8, 16, -18, -6, 23, 8, -19, 7, -1,
21, 3, 12, -15, 4, -8, -20, -18, 15, 19, 25, 18, 37, 28, 35, 11, -7,
18, -24, 18, -15, 25, 0, 12, 16, -4, -2, 1, 27, 16, 39, 43, 37, 40,
36, 33, 21, 56, 54, 33, 30, 22, 26, -4, 16, -8, -22, 11, -5, -23, 20,
-1, -15, -4, -20, 6, 18, -11, 18, -12, 1, 28, -5, 5, -6, -4, 27, 28,
-15, -5, 24, 8, 19, -23, -12, 17, 0, 24, 5, 16, -2, 4, -3, 16, -21,
24, -9, -17, -23, 11, -18, -23, 9, 9, -12, 11, 5, -9, -13, 16, -10,
-21, 9, 11};

//    logic signed [7:0] zeroweights[783:0] = {-6, -18, -9, 10, 19,
-18, -22, 11, -8, 23, -21, 3, -18, 3, -8, -4, 16, 2, 1, 15, 4, -18,
3, -13, -16, -19, -2, 3, 5, 20, -10, -8, 24, 21, -24, 5, 4, -19, -24,
19, 6, 15, 6, -8, -27, -22, 6, -10, 20, -19, -19, -11, 17, 1, -17,
15, -17, -2, 8, -10, -25, 7, -21, 12, -28, 5, -17, -37, -3, -14, -17,
-28, -25, -15, 13, -7, 20, -13, -10, 16, -4, -1, -17, 0, 4, -8, 4,
-24, 18, -5, 11, -17, 0, -18, -1, -31, 0, 2, 21, -14, 3, -19, -17,
-1, 1, -6, 13, -9, 2, -15, 22, -1, 14, -16, -5, -4, -5, -22, -30,
-34, -20, -25, -19, 14, 2, 7, 26, 25, 29, 41, 31, -4, 1, 7, 2, -10,
24, -22, 0, -15, -10, 18, -17, 19, 0, -8, -14, 2, -2, 9, -8, -5, 7,
41, 54, 41, 62, 31, 19, 34, 13, -2, 5, -18, 0, 5, 9, -3, -6, 15, -2,
19, 9, 17, -26, -15, -4, -19, -2, -12, 16, 13, -5, 26, 37, 21, 33,
15, 8, 25, -15, -1, -12, 13, 1, 10, -5, -4, 22, -9, 1, -21, -13, 17,
1, -16, 10, -22, -6, 2, -13, 33, 21, 47, 56, 42, 35, 10, 34, -11, 13,
-25, -8, 3, 23, -1, 20, 4, -17, -18, -11, -12, -9, 7, 21, 13, -7,
-25, -41, 11, 7, 19, 32, 27, 19, 23, 12, -8, -24, -22, -8, -3, 1, -9,
14, -3, 11, 11, 6, -13, 9, 0, -8, -21, -43, -31, -73, -30, -7, 31,
56, 40, 59, 40, 16, 4, 5, -22, 24, -8, -8, 23, -2, 22, 11, 26, 32,
22, -1, -8, 14, -29, -69, -83, -63, -70, -4, 38, 59, 23, 46, 38, 7,
-19, -24, 12, -7, 16, -1, 15, -10, 16, 38, 3, 1, 10, -3, 16, -38,
-46, -95, -74, -91, -46, -30, 35, 34, 53, 35, 34, 34, -22, 4, 15, 11,
-14, 3, 22, -15, 19, 56, 36, 35, 13, -1, -33, -36, -95, -96, -90,
-102, -47, -29, 53, 27, 27, 40, 11, 39, 20, 21, 7, -12, 22, 0, 11,
21, 40, 21, 51, 23, 22, -9, -24, -68, -98, -94, -100, -79, -35, 8,
18, 28, 20, 19, 42, 24, -26, 17, -6, 10, -17, 2, 20, 6, 4, 32, 45,
```

```
51, -5, 6, -21, -56, -97, -79, -78, -50, -18, 1, 29, 24, 6, 24, -2,
-11, 9, 18, -16, 10, -17, 8, -16, 18, 13, 44, 38, 40, -2, 17, -31,
-45, -100, -65, -62, -40, 10, 3, 43, -2, 6, -14, 25, 1, 16, 12, 23,
-15, 24, -8, -22, 19, 17, 44, 60, 25, 39, -3, -6, -27, -61, -44, -18,
9, -7, 22, -9, 9, -14, 1, 18, -3, 4, -12, 13, 1, 10, 17, 6, 16, 19,
33, 23, 42, 16, 43, 20, 12, -41, -13, 17, -9, 34, 27, 6, 18, 12, 13,
-29, 0, -24, -6, 9, 17, -13, 22, -1, 6, -2, 9, 44, 13, 21, 38, 21,
36, -1, 27, 9, 40, 22, -6, 24, 3, 6, -8, 12, -8, 8, 6, -13, 14, -4,
-12, 19, -26, 4, 7, 16, 37, 6, 17, 55, 50, 56, 22, 13, 11, -13, 3, 7,
-22, -7, -27, -5, 0, 7, -4, -12, -23, -7, -24, 13, 16, -4, -22, 27,
-6, 33, 12, 47, 40, 12, 34, 33, 21, 22, 16, -19, -10, 4, -15, 8, -14,
-22, -9, 13, -4, 6, -3, 7, -9, 7, 1, -17, 26, 3, 41, 35, 26, 58, 44,
24, 2, 8, -15, 7, -26, -1, -22, -20, -13, -21, 24, 14, -4, -9, -16,
22, 19, -18, -31, -6, 9, -5, 25, 16, 17, 50, 19, 27, 22, -2, -3, 5,
5, 0, 12, -10, -23, -1, 11, -14, 0, 4, 23, 13, -5, 1, 17, -19, -1,
-12, -1, 0, 5, 3, 27, -8, 20, -21, 3, -12, 15, -18, 19, -24, 13, 14,
-22, -8, -8, -25, 2, -1, -18, -14, -22, -20, 4, 7, -20, -10, -7, -15,
-27, -5, -7, -6, -6, -14, 12, -17, -23, -4, -21, -15, 24, -24, 3,
-17, 4, -12, -11, -23, 15, -25, 19, -1, 5, -20, -8, 9, 5, -22, 2, -9,
2, -9, 13, -24, -1, -9, 21, -10, 1, 10, 4, -17, -10, -6, 13, -18,
-11, -23, -24, -19, 15, -19, -9, -4, -22, -16, 1, -16, -13, -24, -10,
-19, 5, -9, 24, -14, 21, 11, 13, 0, -19, 20, -17, 24, -9, 23, -7,
-16, 12, -1, -7, 11, 0, 24, 6, 23, 4, 13, 13, -16, -18, -14, -10, 4,
-15, 15, -3, -20};
//    logic signed [7:0] oneweights[783:0] = {16, 10, 7, 20, -13, -6,
-6, -6, 18, -10, 0, 10, -21, 23, 18, -9, -14, -6, -4, 7, 21, -11, -4,
1, 10, 8, -19, -15, -20, -6, 15, 18, -21, -10, 21, 9, 17, -10, -16,
20, -20, -9, 16, -25, -19, 18, 18, 0, 11, -6, -9, 10, 0, 0, 1, 23,
-23, 21, -18, -19, -13, 21, 11, 22, 0, 13, 6, -4, -32, -10, -31, -11,
2, -8, -8, 14, -10, 8, 14, -3, 7, 15, 22, -24, 8, 23, -23, 18, -12,
15, 2, -4, 4, -11, -6, 1, -21, -25, -46, -37, -45, -19, -37, -1, 14,
-11, 8, -16, 14, 4, -7, 16, -9, -17, -9, -4, -14, -15, 13, -20, 5, 7,
-1, 14, 13, -17, -11, -16, -36, -3, -10, 1, 39, 5, -14, -15, 12, -18,
18, -2, -2, -14, -1, -6, 21, -3, 0, -26, -23, -3, 9, 33, 0, -8, -15,
-39, -36, 8, 16, 38, 2, -9, 13, 15, -8, -9, 23, 6, -20, 9, 9, 10,
-15, -23, -13, -26, -11, -29, 13, -2, 9, -7, -17, -33, -17, 7, 0, 15,
-8, -17, -1, -9, 24, 1, 13, 17, -4, -19, 9, -8, -25, 15, -12, -7,
-34, -28, -31, 8, 8, 11, 11, 7, 0, 24, -8, 1, -16, -36, 11, 19, 17,
-20, -22, -19, 15, -18, -25, -17, -16, -24, 8, 0, -39, -19, -13, -19,
```

```
-9, 37, 35, 6, 9, -8, -16, -24, -7, -29, -39, -12, -14, -10, 5, 8,
-8, -2, 19, 18, -21, -24, -19, -19, -43, -29, -59, -26, 15, 29, 56,
21, 38, -8, -26, -4, -18, -3, -1, -16, 20, -24, 11, 8, 7, -20, 20,
-1, 8, -34, -25, -21, -15, -65, -46, -24, 5, 64, 72, 49, 9, -19, -47,
-18, -44, -26, -27, -14, 6, 4, 18, -8, 1, 18, -17, -11, -26, -2, -27,
-11, -21, -43, -54, -73, -8, 112, 86, 12, -40, -26, -57, -17, -26,
-18, -10, 7, -5, -8, 21, -25, 4, 19, 21, -4, -13, -19, 3, -41, -15,
-42, -40, -28, 43, 113, 85, -6, -40, -47, -40, -16, -17, -16, -21,
-3, 16, -7, -16, 10, 18, -25, -11, -9, 0, -21, -36, -28, -20, -61,
-75, -44, 56, 118, 75, -22, -79, -62, -52, -31, 6, -22, -23, 5, -21,
-23, 15, -2, -20, -17, -20, 11, 6, -31, -9, -4, -45, -56, -49, -1,
80, 115, 78, -53, -68, -59, -14, -44, -15, -12, -5, 3, -10, -23, -4,
6, 0, 17, 13, 2, -9, 0, -10, 5, -29, -23, -14, 22, 107, 116, 19, -31,
-60, -66, -31, -21, 9, 13, -3, -10, -1, 24, 6, -1, 13, -12, -5, -13,
3, -35, -22, -25, -45, -14, -6, 41, 109, 127, 23, -27, -53, -31, -53,
-26, -24, -16, 0, -6, 3, 3, 6, 11, -12, 13, -3, -18, -4, -7, -50,
-56, -32, -27, 6, 50, 65, 94, 7, -14, -17, -34, -53, -3, 0, 2, -10,
20, -12, 7, 16, 2, 23, -6, -10, -16, -10, -20, -30, -50, -42, -19,
-16, -2, 37, 24, -6, -34, -44, -47, -41, -6, -43, -8, -9, -12, -21,
-24, -12, 4, -4, -2, -14, 0, -9, -10, -8, 0, 14, -1, -11, 12, 20, 29,
-20, -20, -53, -64, -27, -8, -5, 7, 16, 19, -23, 23, 7, 22, -5, 4,
-10, -24, -4, 13, -1, 11, -7, -16, -22, -18, -40, -19, -3, -25, -57,
-21, -54, -38, -13, -11, -10, 10, -9, -22, 5, -3, 9, -11, -11, -13,
-16, 4, 29, 35, 6, 19, -6, -23, -37, -5, 11, -18, -22, -18, -34, -42,
-17, 16, -17, -11, 22, -23, 14, -3, -24, 13, -16, -15, 28, 1, 12, 7,
14, 3, -28, 5, 8, 19, -3, 12, 7, -3, 5, -6, 10, -25, 10, -2, 9, 6, 2,
-15, -3, 1, 1, 17, 1, 14, 37, 43, -2, 0, 18, 18, 6, 21, -1, 15, -4,
-30, 5, 11, 0, -18, -4, 21, 22, 9, -24, -5, 17, 22, -12, 11, -10, 18,
13, -28, -4, -19, -4, -19, 8, -2, 9, -27, -18, 4, -25, -15, -18, 15,
-16, -10, -17, 5, 3, -5, 22, 25, -12, 5, 5, 10, 20, -2, -2, -24, -19,
8, 21, -8, 3, 10, 1, -7, 3, -15, -2, -18, 11, 4, 15, 24, -12, 6, 0,
9, 21, 16, -21, -13, 5, 14, -23, 23, 19, 18, 10, 22, -1, 4, 3, 16,
-17, 13, 19, 5, -21, -2, -10, 24, 21, -23, 1, -10, -18, -2, -15, 23,
25, 16, 25, -21, 21, 8, 16, 15, 5, 24, -1, 5, 11, -20, 5, 25, 3, 3,
3, -12, 16, 5};
//    logic signed [7:0] twoweights[783:0] = {23, -19, -8, -12, 6,
-5, -11, -15, -14, 0, -9, 7, -6, 23, -7, -6, -5, 19, 4, -10, 5, -9,
5, -12, -1, 7, 18, 12, 21, -18, -1, 16, 2, -4, 12, -14, -20, -8, -17,
-18, -1, -24, -23, -6, 18, 3, 19, -19, -4, -10, 11, 22, -9, -8, -18,
```

-1, -3, -20, -11, 5, 4, 18, 12, 18, -3, 3, 10, -13, -26, -8, -4, 1,
-9, 11, -16, -16, -17, 3, 17, -5, 7, 8, 19, 0, 17, -17, -9, -18, 2,
-20, 9, -28, 9, -12, -5, -12, -20, -22, -16, -44, -12, -41, -20, -19,
-18, 3, 14, -8, -25, 3, -24, -21, -8, 24, -14, 13, 23, 5, 7, -13, 6,
-19, -22, -45, -51, -33, -22, -32, -11, -18, -25, -18, -31, -9, 18,
1, -24, -5, 8, -16, 13, -1, -15, 7, 20, 21, 38, 9, 22, 38, 4, -1,
-26, -27, -24, -18, 13, 11, 41, 6, 37, 19, -9, -2, -15, -1, -8, -1,
-6, -22, -8, 28, 40, 26, 50, 55, 28, 41, 24, -24, -28, 1, -2, -7, 32,
12, 36, 40, 35, 27, 12, 4, -1, -17, -9, -21, 8, -4, -8, 39, 58, 60,
50, 21, 18, 31, 15, 17, -5, 17, -7, 18, 35, 40, 27, 36, 58, 35, 46,
35, -8, 4, 16, 18, -17, -11, 32, 33, 39, 48, 46, 29, 17, 1, 11, 24,
34, 15, 35, 30, 43, 46, 31, 53, 21, 27, 37, 34, 14, 7, -19, -19, 7,
-4, 11, 34, 46, 45, 16, 33, 25, 4, 31, 22, 6, 29, 49, 74, 41, 55, 39,
34, 36, 34, 29, 6, 8, -8, -23, -12, 21, -15, 46, 52, 38, 43, -7, 20,
8, 1, 4, 16, 39, 47, 46, 54, 42, 15, 37, 21, 41, 31, 26, 24, -9, 19,
9, 3, 12, 21, 22, 33, 7, -17, 3, -18, -2, 1, 21, 32, 26, 7, 38, 33,
48, 25, 16, 15, 20, 24, 12, 14, -20, 1, -6, -23, 19, 20, 27, -10, 8,
-10, 3, -30, -1, -33, 6, -22, 1, -2, 32, 15, 25, 14, 13, -12, 0, -34,
8, -8, -18, -10, 20, -6, 19, -11, 21, -15, -29, -34, -19, -26, 5,
-12, -5, -4, -38, 16, 12, 11, -24, -3, -30, -36, -32, -56, -27, 1,
23, 13, -16, 6, -1, -13, -3, -22, 4, -44, -5, -24, 5, -19, -17, -12,
-41, -43, -36, -54, -48, -81, -70, -51, -72, -41, -30, -26, -19, -23,
22, 1, 6, 20, 1, 17, 1, -22, 2, 7, -5, -25, -36, -41, -43, -67, -69,
-93, -78, -66, -94, -60, -65, -48, -27, -4, -9, -3, -8, -2, 17, -8,
-24, -3, -20, -30, -2, 22, 2, -1, -8, -9, -10, -66, -66, -66, -67,
-75, -47, -71, -66, -28, 7, -9, 23, 9, 23, -15, -5, -16, 10, 14, 2,
-6, -21, -12, 1, 17, 12, 18, -25, -29, -47, -48, -67, -32, -32, -47,
-14, -21, -7, 16, -15, -18, 10, 12, 18, -15, -12, -19, -2, -17, -9,
-6, -8, 25, 2, 2, 9, 11, -1, 0, -1, 7, -23, -13, -19, 7, 22, -7, -18,
-21, -23, -2, -10, -7, -15, -37, -5, -20, -12, -5, 4, -15, -6, 27,
13, 27, -6, 17, 17, 18, 20, 21, 13, 0, 21, 15, 25, -16, 23, -6, -12,
12, -8, 6, -2, -5, -18, 11, -13, 6, 2, 21, -2, 3, 16, 12, 22, 25, 8,
27, 15, 38, -4, -4, -17, 6, -24, 1, -9, 17, -5, -31, -6, -6, -8, 16,
-5, 20, 18, 30, 39, 15, 38, 39, 35, 40, 52, 43, 35, 30, -16, -14, 3,
14, -24, 25, -19, 1, 9, 1, -14, -17, 5, -17, -7, 40, 39, 44, 69, 74,
41, 47, 76, 62, 28, 34, 33, -8, -18, 12, -19, 20, -12, 18, -9, -22,
-8, -13, -11, -7, -28, -3, -7, 26, 10, 56, 37, 53, 45, 41, 43, 52,
36, 12, 14, 5, 19, 17, -18, -12, 23, -14, -8, -3, 11, 10, -20, 2,
-16, 13, 7, 19, 8, -3, 9, 41, 30, 12, 35, -4, 13, -3, -6, 21, 13, 21,

```
-11, -13, 23, 10, -4, 13, 13, -21, 3, 14, -13, 14, 17, -18, -26, -18,
-23, -3, 13, 2, 20, -21, -3, -3, -3, 23, -9, 21, -11, 3, 1, -5, 8,
-24, 23, -4, -18, -7, 1, 16, 6, -13, 3, 24, 10, 17, 14, 6, -17, -19,
21, -6, -2, -20, 17, 13, -5, 17, 21, 9, -14, 13, 10, 23, -15, -17,
-9, 10, -18, -7, -4, 17, 0, 7, -19, 20, -21, -15, 10, -20, -12, 1,
-4, -11, -8, 11, -19, 0};
//    logic signed [7:0] threeweights[783:0] = {19, 1, 10, -23, -5,
1, -10, 17, 13, -1, -12, -9, 14, 21, -8, -23, -8, -11, 0, 16, 19,
-21, 2, 18, -1, 13, 0, 17, 19, -20, -24, -8, -23, 4, 0, 13, 20, -10,
12, 3, 3, -5, -4, -12, 10, 15, -14, -19, 3, -23, 6, -2, 22, 24, -11,
3, 9, -7, -4, -7, 17, -11, 11, 17, -3, 0, -5, 11, -10, 6, 29, 16, 8,
21, 2, 23, 6, 22, -19, 1, 0, -13, -21, -17, 11, 0, 12, 7, 11, -20,
18, 15, -23, 14, -18, 5, 10, 24, 47, 30, 19, 27, 51, 28, 36, 37, 16,
-14, 6, -19, -20, 22, -18, 3, -21, 20, 18, -22, -19, 13, 6, 17, 18,
-18, 24, -14, 18, 14, 5, 44, 36, 23, 32, 17, 12, 5, 16, -16, 18, -19,
-2, -17, -22, -26, 15, -6, -11, 5, -13, 14, 16, -5, 2, 14, 24, -4,
-2, 24, 24, 28, 51, 38, 28, 4, -10, 27, -15, 8, -4, 8, -3, 5, -2, 6,
-19, 20, 31, 27, 32, 17, 21, -11, 13, -9, -25, 15, 18, 25, 11, 48,
30, 27, -7, 23, 3, -5, 18, 19, -6, -4, -29, -10, 28, 16, 25, 38, 39,
33, -5, 19, -34, -28, -31, -20, -18, 28, 18, 50, 23, 25, 31, 5, 1,
-21, -21, 21, -25, -4, -4, 0, 35, 46, 46, 29, 34, 26, -4, -40, -47,
-26, -38, -30, -18, -38, 2, 29, 49, 14, 36, -3, -2, -16, -16, -4, 6,
-13, -19, 35, 13, 12, 35, 23, 13, 18, -13, -68, -49, -74, -83, -66,
-55, -14, -6, 6, -2, 36, 13, 5, 11, 16, 19, -12, 13, 7, 26, -2, 41,
36, 29, 41, 22, 8, -25, -41, -72, -41, -85, -87, -42, -47, -24, -23,
-8, -1, 5, 16, 2, -17, 22, -20, -22, 1, -12, 28, 27, 14, 37, 8, 18,
-6, -16, -35, -44, -18, -26, -34, -61, -52, -7, -35, 6, -4, -18, -17,
4, 7, -15, 23, 21, 19, -12, 7, 3, 28, 18, 28, 4, 12, -2, -26, 5, -1,
-5, -32, -53, -28, -17, 0, -26, 6, 23, -15, 14, 19, -17, -20, 11,
-25, -25, -27, -6, 6, 17, 22, -5, -12, 2, 20, 19, 44, -5, -9, -23,
-49, -36, -22, -32, -15, 19, 5, -6, 3, 2, -20, 15, -26, -2, -32, -12,
-10, -33, 6, 26, 26, 24, 35, 48, 21, 0, -40, -25, -57, -56, -31, 12,
-12, -25, 5, -24, 2, 17, 12, -6, 7, -31, -2, -8, -35, 5, 30, 13, 31,
38, 64, 35, 7, 2, -35, -65, -34, -55, -44, -33, -22, 12, -5, -13,
-13, 9, 20, -1, 1, -31, -24, -9, -6, 32, 37, 25, 36, 45, 67, 32, -33,
-51, -73, -56, -84, -61, -34, -8, 3, -16, -17, -21, -21, 9, 13, -16,
16, -18, 7, 9, 22, 12, 24, 36, 60, 42, 27, 3, -61, -55, -81, -72,
-69, -32, 1, -2, 25, -1, -11, 4, 4, 5, 11, -1, -2, -30, -22, 11, 15,
18, 40, 21, 20, 51, 8, -16, -37, -62, -42, -35, -13, -5, 19, 7, 9,
```

```
-8, 9, 18, 17, 5, 3, 12, -6, -25, -14, 27, 31, 18, 22, 17, 47, 43,
10, -4, -7, 7, -12, 7, -27, 26, 23, 29, 35, -16, 21, 1, 1, 22, -16,
-3, -9, -11, 6, -11, 32, 26, 12, 6, 40, 40, 24, 12, 7, 8, 21, 23, 40,
17, 20, 30, -9, 13, -19, -13, 19, 4, 11, 7, -20, 4, -25, -7, 0, -10,
28, 19, 39, 27, 20, 27, 35, 49, 29, 49, 40, 18, 38, 10, 32, -9, -1,
-24, -18, -14, -1, 4, -18, -29, -34, -10, -31, -15, -23, 6, 16, 14,
32, 12, 11, 41, 41, 57, 24, 9, 23, 20, 18, 21, 4, -11, 24, 4, -5,
-11, 8, -25, -7, -22, 3, 1, -19, 12, 5, 17, 24, 17, 42, 29, 19, 22,
14, 23, 7, 10, 21, 18, -2, 9, -16, -22, 2, -8, 23, -22, -14, 17, -11,
12, 8, 14, 17, -3, -4, 19, -6, 5, -2, 5, -12, -10, -22, -11, -7, 19,
-13, -9, 19, -7, -12, -24, 1, -19, 15, 19, 2, 24, -5, 1, 8, 8, 4,
-22, -7, -14, -11, -17, -9, -14, -7, -3, -22, -20, 5, -4, -6, -7, 20,
22, 4, 10, 0, 6, 8, 19, 17, -8, 21, 1, -11, 11, 1, -1, -11, 5, 15,
-16, -20, 11, -9, -6, 12, 2, 4, -2, -20, 18, 20, 3, 18, -9, 22, -5,
-16, 15, 15, -10, -7, 0, -4, -12, 5, 21, -4, 24, -22, -20, 23, -20,
23, 22, -23};
//    logic signed [7:0] fourweights[783:0] = {14, 10, -11, 14, 17,
-7, -3, 3, 17, -21, 13, -10, 20, 12, -9, -18, 18, -22, 8, -15, 11,
-3, 21, 15, -24, 21, 24, -17, 2, 23, 11, -19, 5, 6, 3, 9, 0, -22,
-35, -26, -11, 3, -17, -31, 0, -2, 1, -28, -13, 17, -8, -22, 3, 14,
24, 23, -9, -9, 18, 21, 13, -20, -11, -12, -24, -35, -37, -21, -5,
-4, -29, -40, -44, -1, -19, 0, 6, 22, -24, 5, 7, -1, -7, -15, -20,
17, -4, -12, -20, 16, 16, 13, 4, -9, -13, 4, -8, -31, -8, -23, -8,
-13, -3, -15, -30, 5, 6, -16, 8, -5, 10, 22, 19, 19, 21, 24, -21,
-13, -1, -6, 38, 1, 20, 0, -17, 0, -6, -5, -19, -12, -24, -31, -18,
3, 13, 5, 0, -24, -9, 19, 7, -23, 24, -11, 3, 13, 18, 24, 3, 24, 2,
-18, -12, -38, -3, -14, -23, -55, -47, -6, -40, -37, -16, 13, 22, 14,
-8, 6, -5, 4, 6, 8, 18, 14, -22, 16, -13, 18, 0, -19, -22, -39, -34,
-26, -36, -60, -44, -36, -58, -30, -1, 4, 0, 23, 0, -1, -5, 10, 15,
17, -13, -7, -2, -29, -28, -21, -28, -19, 10, -16, -43, -38, -38,
-69, -40, -72, -50, -35, -23, -2, 18, 11, 24, -11, 18, 20, -20, -19,
-8, -31, -22, -27, -29, -31, -4, -1, -2, 14, -31, -38, -59, -63, -54,
-51, -50, -46, -10, -15, 20, -21, 18, 21, 4, 21, 10, -22, 0, -4, -33,
-23, -14, -23, 4, 31, 36, 35, 27, 9, -31, -25, -20, -6, -22, -14,
-10, 6, 4, 0, 8, 9, -12, 10, -24, -24, -12, -30, -7, 1, -20, -6, 40,
65, 71, 66, 45, 8, -15, 29, 21, 21, 34, 12, -20, 3, -21, -2, 16, 8,
-14, 23, 6, -7, -18, -5, -19, 18, 15, 54, 76, 85, 83, 70, 20, 13, 20,
37, 51, 53, 54, 42, -2, 15, -25, 7, 7, -13, 1, 19, -12, 0, 13, -9,
10, 16, 38, 54, 59, 76, 40, 13, 6, 47, 40, 61, 45, 84, 42, 44, 0, 18,
```

```
-6, -8, -10, 17, 10, -17, 6, 1, -1, -12, 9, 33, 48, 49, 77, 68, 23,
12, 12, 41, 63, 65, 51, 57, 70, 40, 2, -17, 8, -19, -25, -20, 6, -20,
-12, -8, 8, 8, 4, 21, 3, 31, 76, 40, -8, -13, -4, 21, 66, 77, 65, 48,
28, 33, 29, 17, 17, -20, -17, -20, -21, -3, 21, -8, -19, 8, 3, -21,
-3, 22, 25, 53, 7, -60, -35, 35, 67, 70, 59, 47, 34, 21, -5, -8, -27,
6, -10, 8, 3, -6, 9, -2, -23, -26, -25, -11, -2, 28, 52, 26, -19,
-37, -41, 38, 24, 45, 17, -7, 15, -19, -19, -22, 19, 20, -21, 6, 21,
3, -4, 7, -3, -8, -24, -18, 7, 14, 13, 3, -29, -84, -17, -3, 6, 25,
19, -17, 0, -5, -24, -14, -20, 11, 14, 13, -1, 19, -21, -5, 4, -4,
-18, 9, 9, -17, 4, -16, -88, -70, -25, -19, -22, 6, -26, -12, -8, -6,
-17, -16, 19, 8, 15, -7, 5, 10, -22, -11, 5, 33, -4, 12, -31, -15,
-31, -64, -78, -103, -75, -29, -19, -37, 0, -16, -19, 8, -8, 10, -14,
-18, 2, -16, 14, -1, 8, -1, 4, 28, 34, -22, -12, -32, -51, -56, -79,
-69, -75, -54, -54, -23, -7, 12, -7, 3, -5, -11, 9, -24, 5, 21, -21,
-1, -5, 14, 37, 32, 3, 24, 10, -17, -37, -47, -46, -77, -48, -22,
-37, -1, 13, -20, 18, -2, -19, 10, -23, 3, 9, -18, -20, 24, 16, -15,
11, 28, -8, 25, 5, 17, -2, -8, -26, -35, -19, -10, -5, -1, -4, -24,
13, 25, 0, -17, -12, 13, -24, -19, -4, -20, 3, 14, 13, 23, 6, -5, 2,
3, -25, -24, -33, -38, -10, -49, -25, 0, -8, 4, 1, 5, 11, 7, 7, -16,
8, 2, 4, 17, -12, -4, -16, -15, 6, -2, 4, -3, -30, -28, -19, -3, -11,
-31, 8, -18, -10, -3, 21, -24, -1, 17, 10, -22, -6, -2, -14, -4, -19,
-18, 9, -18, -19, 11, 8, -1, 15, 0, -31, -26, -23, 9, 10, 0, -11, 0,
-5, -24, 16, 6, -6, -7, 15, -16, 3, 4, -13, 14, -23, -24, 20, -18, 2,
6, -7, 17, -17, 17, 4, -11, 9, -11, 9, -1, 20, -21, 6, -13, -9, -24,
24, -14, 18, -2, -7, 14, 15, -17, 7, -24, -20, -14, 3, 0, -2, 12,
-13, 21, 3, -18, 7, 10, 11, 22, 5, -19, 7, -9, 12, 3};
//    logic signed [7:0] fiveweights[783:0] = {-10, 3, 15, 15, -4,
-6, 10, 3, 15, 7, 5, 23, 19, 13, -13, 22, -18, -24, 10, 7, -23, 3,
-4, -17, 16, -24, 10, 17, 19, -10, -11, -9, 15, 22, -14, -10, 10, -1,
-16, -18, -14, 21, -18, 17, -23, -21, -3, 21, -10, -21, -20, -9, 22,
-11, -23, 10, 24, 19, -11, 20, 24, 2, -1, -19, 9, -19, -11, 12, -23,
-13, 26, 23, 17, -1, -18, 11, -17, 10, 1, -4, 1, 12, 7, 20, -15, 2,
14, 6, 9, 2, 3, -7, -10, -17, 9, 5, -1, 28, 42, 39, 17, 6, 23, -3,
-21, -2, -18, -14, -20, 7, 6, 0, 11, -24, 2, 19, 25, -1, 22, -11, 19,
-5, 21, 22, 32, 25, 32, 18, 39, 38, 31, 4, 19, -4, -10, 11, -7, -19,
-1, 20, -19, -24, 20, 25, -4, 22, 14, 28, 27, 7, 31, 5, 16, 10, 11,
42, 30, 26, 32, 22, 0, -24, 14, -4, 0, -15, 13, -22, 18, -8, -24, -1,
1, 34, 33, 20, 14, 9, 29, 14, 6, 18, 5, 15, 8, 28, 36, 37, 40, 5, -3,
2, 20, 2, 17, -15, -8, -20, 9, 22, -4, -4, 2, 13, 25, 4, 0, 22, -11,
```

```
6, 4, -11, 28, 38, 48, 32, 40, 16, 20, -14, -19, -12, 21, 18, -12, 3,
7, -1, 6, 22, 20, 25, 16, 20, -13, 2, 1, -5, 1, 6, -1, -25, 1, 42,
42, 27, 28, 4, 1, -20, 16, 7, -3, 9, -15, -2, 11, -1, 3, 2, -2, 20,
27, 5, 0, 7, -25, -16, -58, -31, -25, -29, 13, 25, -2, 17, -22, 11,
-21, 14, -12, 16, 19, -18, 15, 2, -3, 0, 9, 23, -15, 6, -37, -26,
-35, -66, -69, -66, -63, -54, -15, -5, -17, -8, 18, 0, 5, -5, -5, -5,
-20, -1, -25, -18, -20, 5, -16, 4, -30, -34, -19, -30, -43, -21, -14,
-22, -37, -47, -18, -6, -24, 2, -9, 4, -2, 12, -4, 12, -9, -11, 0,
-23, -3, -21, 10, -27, -22, -37, -31, -51, -32, 7, 21, 29, -5, -23,
-26, -26, -22, 0, 21, -23, -15, 24, -25, 22, -21, 11, -2, -15, -23,
2, -23, -22, -31, -41, -55, 0, 24, 52, 52, 56, 48, 17, -18, -21, 11,
-10, -21, 22, -15, 4, -14, 23, 10, -1, -20, -45, -22, -52, -8, -23,
-31, -42, -50, 8, 29, 58, 52, 58, 19, 31, 25, 12, -5, -5, 7, 6, -10,
6, -8, -11, 11, -11, -50, -57, -56, -63, -79, -47, -65, -48, -3, 9,
37, 92, 97, 64, 45, 22, 10, 1, 23, -5, -18, 11, -19, 4, -24, 18, 24,
-21, -40, -44, -67, -73, -79, -48, -63, -29, -13, 35, 69, 55, 43, 65,
32, 25, 34, 14, 3, -20, 20, -21, -13, 24, 22, 26, 37, 47, 15, 12,
-10, -43, -26, -51, -55, -22, -24, 13, 39, 53, 47, 53, 27, 37, 21,
-21, 6, -28, -16, -4, 18, 11, -9, -8, 24, 54, 77, 29, 28, 16, -25,
-34, -9, -38, -22, 12, -10, 20, 45, 18, 42, 23, 11, 0, 0, 10, -2, -6,
6, -23, -24, 30, 71, 74, 70, 35, 49, 27, 12, 5, -13, -11, -32, -17,
16, 26, 37, 27, -2, 16, -9, -4, -13, -11, 1, -21, -17, -20, 10, -4,
30, 67, 62, 74, 39, 19, 10, 2, 4, -12, -19, -3, -17, 24, 18, 40, 21,
-4, -1, -37, -31, 10, -8, 17, -11, 17, 2, 28, 15, 45, 66, 37, 22, 39,
27, 30, 7, 9, -5, -10, 4, -2, -3, 18, 12, 6, -3, -11, -20, 6, 21, 9,
19, 17, 21, -15, 23, 11, 32, 52, 35, 17, 27, 19, -14, 9, 12, -11,
-16, -27, 15, -1, -3, -32, -34, -5, -3, 15, 14, 21, 23, 16, -13, 17,
-20, -17, 33, -1, 20, 15, -19, 23, -15, 18, 5, -9, -32, -5, -33, -12,
-15, -21, -10, -10, -12, 11, -10, -16, 15, 20, -23, 23, 14, 21, -24,
-16, -15, -6, 21, 7, 10, 8, -11, 1, -23, 11, 15, 1, 12, -3, -25, -8,
-16, 5, 9, 1, 7, 17, 15, 4, -8, -15, -7, -18, 1, -24, 8, -13, -18,
-21, -16, 4, 7, -3, -2, -12, 1, -12, 4, 22, 2, 5, 12, -12, -14, -17,
-11, -17, -4, -17, -13, -21, -3, 25, 0, 13, -23, 11, -3, 21, -6, 19,
-12, 4, -12, -19, -21, 9, -22, 18, 2, -10, -20, 9, 1, 17, -3, 6, -25,
-1, -20, 21, 12, -6, -18, -21, -24, 25, -23, 3, -20, 22, -14, 18, 20,
-12, -19, 11, 25, 11, 24, 7};
//    logic signed [7:0] sixweights[783:0] = {2, -12, -8, -19, 9, -1,
23, -3, -6, -5, -5, 2, -15, 21, 0, -6, -23, 0, -21, -7, 20, -19, 5,
-3, -4, 23, 8, -17, -12, 2, -3, -3, -5, 19, -1, -12, -16, -17, -10,
```

12, 7, 12, -19, -21, 3, -6, -8, 2, 20, -16, 6, -9, -19, -23, 25, -19, 10, -22, 12, 4, -7, 15, -18, -9, -13, -5, -25, 0, -32, -4, -16, 0, -16, -17, 8, -14, 4, -21, 10, -12, -13, 18, -10, -21, 1, -12, -10, 6, 11, 9, -7, -13, -21, 12, -37, -8, -9, -34, -39, -15, -43, -37, -7, -7, -22, -26, 8, 17, 21, 3, 6, 12, -9, -20, -9, 2, -19, -4, -19, -17, -2, -39, -35, -54, -62, -35, -65, -42, -64, -52, -47, -43, -27, 13, -26, -16, 11, 24, -8, -18, -17, -20, -7, 22, 16, -2, -11, 6, 2, -40, -36, -10, -12, 0, 8, -8, -9, -15, -44, -23, -8, -19, -32, -12, -8, 10, -17, 11, -21, 18, 5, 20, -21, 11, -38, -30, 2, -7, 15, 42, 35, 63, 76, 52, 25, -3, -10, -13, -17, -27, -27, 7, 1, -21, 24, -15, 9, -4, 4, 10, -13, 8, -7, 19, 3, 45, 52, 52, 63, 47, 57, 77, 83, 44, 30, 7, -37, -26, -25, -17, -10, 22, 19, -15, -10, -2, 1, -22, -18, 2, -12, 2, 12, 31, 57, 47, 26, 25, 39, 75, 83, 81, 30, 14, -34, -10, -39, -33, 8, 21, 6, 10, 18, -25, 12, -11, 14, -13, 29, 20, 28, 27, 10, 21, -16, 22, 1, 56, 51, 74, 59, 24, -10, -1, -32, 2, -17, -15, 1, 22, 21, 0, 10, 5, 11, -6, 17, 43, 24, 20, -16, -5, 0, 14, 3, 50, 78, 70, 43, 40, 32, -8, -6, -18, 10, 2, 24, 10, -17, -13, -22, -6, 24, 38, 13, 18, 34, -8, -6, -3, -11, 19, -3, 15, 38, 73, 36, 28, 1, 0, -2, -21, -21, -14, -20, 3, -24, 2, 11, 35, 15, 20, 33, 9, 2, -24, -18, 26, 24, 14, -9, 8, 54, 59, 25, 26, 31, 15, 6, 7, 5, -8, -18, -17, 21, 14, -12, 27, 48, 27, 31, 24, -5, 19, 1, -2, 39, -10, 6, 8, 38, 39, 41, 32, -6, 30, 6, 5, -1, -8, -21, 22, 22, -19, -7, 14, 54, 44, 22, 37, -16, 8, 1, -13, -17, -23, -24, 13, 47, 17, 17, 13, 14, -9, 10, 20, -20, 1, 8, 14, 7, 9, 3, -1, 35, 38, 34, -22, -26, -9, -21, -22, -22, -12, 23, 50, 11, 18, 10, 35, 2, 24, -6, -18, -25, 24, -11, -9, 17, 0, -2, 17, -2, -11, -23, 2, -26, -40, -51, -41, -70, -32, -4, 1, 7, -6, -1, -13, -9, 18, -9, -21, 0, 15, -22, 18, -17, 0, 0, -30, -28, -6, -44, -35, -47, -55, -62, -76, -68, 1, 10, -8, -13, -10, 19, 13, -12, -4, -1, 6, -25, -15, 9, 19, -24, 7, -5, -18, -32, -14, -62, -43, -87, -103, -72, -45, -20, 5, -12, -17, 7, -15, -25, -21, -14, 11, -31, -11, 16, 16, -10, 12, -7, -1, 18, -4, -23, -25, -44, -69, -63, -92, -70, -54, -47, -38, -20, -23, -30, -31, 15, 3, -28, -10, 9, 15, -17, 15, 23, 6, -8, 10, -10, 3, -18, -15, -52, -42, -45, -42, -44, -44, -11, -33, -26, -38, -12, -14, -21, -15, -12, -26, -5, -14, -17, -21, 0, 12, -8, 15, 1, -21, -34, -2, -35, -12, -26, -15, -24, -3, -9, -27, -7, -34, -9, -16, -6, 7, 8, 13, -28, 23, 6, -19, -1, 7, 17, 2, -7, 18, 17, 12, 29, 30, 23, 40, 14, 3, -23, -23, -20, -14, -26, 27, -21, 10, -23, 1, 18, -10, 18, 10, -11, -18, 13, -17, 3, -1, 18, 31, 14, 19, 72, 34, 46, 43, 23, 29, 19, 2, -6, 2, 8,

```
-2, -7, -4, 14, -18, 24, -8, 16, -1, -19, -11, -13, -20, 24, -1, 35,
23, 42, 64, 69, 23, 26, 20, 21, 31, 32, 23, -16, 27, 4, 1, -21, 4,
-22, 4, -13, -18, -20, 7, 3, -18, 11, -12, 4, 4, -1, 39, 8, 6, 2, 40,
40, 27, -8, 22, 6, -17, 11, 20, -7, -22, -13, 5, 21, -1, -5, -4, -5,
-22, -23, 0, 24, -19, -6, -15, 10, 1, -13, 25, -12, -8, 8, -5, 11,
15, 20, -16, 23, -2, 22, -16, -16, -20, 17, 5, -4, -19, 6, -18, -9,
2, 13, 20, -19, -10, 23, -8, 8, -9, -5, -8, -13, 8, 8, 24, 7, 1, -23,
7, -7, -3};
//    logic signed [7:0] sevenweights[783:0] = {-7, 18, 14, -2, 5,
-1, 2, -3, -19, 1, 19, -19, -12, 7, -8, 26, 9, 16, 16, 12, 0, -20,
15, -7, -2, 22, -13, -10, -9, -16, 12, -15, -24, 25, 7, 18, 13, -13,
31, 31, 23, 34, 25, 22, 27, 5, 7, -4, 21, -12, -18, 1, 0, 20, -24,
-21, -12, -10, -6, 0, 17, -9, -7, -2, 5, 7, 14, 22, 49, 62, 52, 45,
58, 40, 32, 24, 47, 26, 4, 14, -2, -19, -7, -15, -2, -2, 2, 21, 2,
-8, -20, -8, -1, 6, 2, 17, 21, 34, 39, 26, 27, 14, 15, 42, 32, -11,
14, -6, 11, 3, -10, 12, -4, 14, -15, -25, 17, -3, -1, -19, -33, -8,
-24, -23, 13, 5, 14, -3, 6, 22, -8, 29, 10, -2, 9, 14, 13, -16, 4,
-1, 15, -19, 13, -13, 8, -19, -9, -36, -35, -25, -3, 7, -23, -11,
-10, -1, -28, 12, -11, -33, 5, -7, -29, -5, -4, 2, 15, 24, -1, -7,
16, 22, -24, -2, -27, -12, -27, -26, -24, -32, -15, -12, 3, -15, -24,
-27, -31, -15, -18, -15, -2, 4, -19, 14, 15, -21, 12, 6, -16, -22,
-29, -19, -17, -47, -18, -43, -37, 6, -18, 10, 18, 26, -31, -30, -53,
-49, -51, -34, -4, 13, 1, 24, -23, -13, -8, 10, -24, 14, -2, -38,
-17, -16, -39, -42, -38, -31, -4, 27, 15, 19, -11, -49, -47, -50,
-12, -19, -9, 2, -21, -23, 3, -20, 1, -7, -8, -28, 6, -23, -26, -46,
-23, -10, -20, 17, 31, 44, 24, -17, -43, -41, -31, -21, -40, -48, -8,
-24, -18, -16, -2, 19, -16, 3, -22, -15, -23, -15, -7, -15, -7, 15,
28, 24, 41, 16, -10, -26, -40, -63, -42, -54, -37, 0, -20, -1, -10,
21, -8, -8, -11, 4, -24, -27, 7, 7, 6, 23, 10, 7, 47, 22, 10, -1,
-26, -64, -38, -51, -28, -26, -31, -22, 13, 22, 23, 13, -21, -3, 20,
21, 13, -25, 13, 18, 27, 3, 49, 54, 43, 29, 16, -35, -77, -85, -72,
-58, -59, -30, -3, -30, -24, 19, -23, 19, -4, -12, 6, -24, -11, -21,
2, 24, 11, 39, 50, 44, 44, 9, -38, -61, -114, -91, -82, -93, -61,
-47, 4, -24, 0, 17, 23, 15, -14, -5, 10, -22, -16, -3, 9, -10, 3, 4,
11, 17, 27, 23, -48, -87, -83, -110, -74, -41, -41, -26, -16, 21,
-10, -9, 16, 26, -14, 1, -19, -15, 18, 0, 8, 16, 14, 15, 36, 33, 36,
32, 0, -41, -46, -68, -54, -34, -7, -4, 14, 22, 12, 2, 13, 4, -14, 6,
17, 10, -20, 8, -20, 7, -1, 17, 44, 51, 79, 71, 28, 22, 20, -20, 13,
27, 16, -1, 36, -7, 13, 22, 23, 9, 4, -6, -7, 7, 18, -18, 0, 2, 15,
```

```
39, 57, 48, 71, 75, 49, 34, 26, 37, 17, 19, 17, 13, 6, 39, 28, 43, 7,
10, 24, 0, -5, -24, -12, 3, -25, 8, 13, 11, 47, 70, 87, 64, 82, 52,
49, 37, 43, 43, 16, 51, 22, 33, 39, 16, -7, 0, -10, -22, -11, -12,
-25, 3, 13, -6, 2, 21, 13, 52, 65, 64, 54, 31, 33, 33, 47, 56, 38,
34, 15, 22, 39, 15, -3, 27, 1, 17, -5, -25, -2, -24, 7, -5, -7, 11,
-7, 16, 18, 2, 18, 8, -2, 29, 25, 45, 19, 17, 19, 32, 11, 26, -1, 10,
-18, 8, -21, 6, -21, -17, -10, -14, -2, -22, -46, -19, -37, -40, -53,
-49, -42, -20, 7, 6, 23, 11, -20, 25, 1, 21, 18, -7, 5, 6, -13, 4,
-4, 12, -25, -3, -34, -21, -41, -28, -50, -36, -26, -70, -25, -30,
-21, -20, -16, -25, 3, -18, -24, 12, 17, -5, -13, -3, -13, -22, -10,
8, -9, 22, -28, -18, -2, -28, -2, -21, 0, -43, -37, -3, -19, 7, -13,
-13, -15, -23, -17, 14, 15, 21, -3, -4, 20, -23, -10, -1, 5, -23,
-18, 18, 8, 6, -7, 13, -27, -6, 13, -7, -22, -6, -26, -10, 17, -18,
9, -14, 13, 15, 22, 11, -20, -19, -2, -24, 18, 11, 3, 11, -17, -22,
13, -22, 2, -6, -12, 12, -15, 22, 9, 16, -18, -20, -24, -20, -3, 18,
-19, 2, -21, 22, -17, -5, 24, -10, -22, -5, 10, -6, -17, -17, -6, 14,
-9, 14, 4, -7, 16, 6, 5, 0, 14, 10, -16, 24, 22, -11, 10, 23, -22,
-8, -1, 13, 20, 2, 25, -9, -3, 11, -15, -18, 5, 2, -19, 5, 18, 17,
12, 5, -1, 22, 23, -4, -21, -13};
//    logic signed [7:0] eightweights[783:0] = {-10, -2, 20, 2, -17,
22, 18, 14, 3, 17, -24, 13, 10, 6, -10, -23, -14, -10, 11, 21, 21,
22, -18, -6, 23, -5, -21, 21, 17, -7, 16, -8, -17, 2, -17, 18, -4, 4,
6, 2, 11, -13, -28, -17, -29, -15, -22, 14, 1, -23, 8, 17, -9, -23,
-16, 10, 16, 24, -3, -4, 16, 7, -15, -9, -18, -5, -20, -15, -6, 16,
-7, -28, -20, 0, -15, 2, -30, -29, -10, -20, 22, -19, 15, 20, -19,
-10, -14, 1, -9, -7, -18, -20, 31, 25, 33, 25, 33, 25, 24, 15, 23,
-9, 0, -33, -11, 5, -13, -19, -2, 0, -23, -14, -12, -9, 23, 15, 4,
12, 22, -1, 24, 0, 16, 36, 39, 54, 87, 70, 32, 6, 26, -4, -33, -4,
-19, -16, 4, 6, 5, 1, -7, 18, -1, 5, -22, -7, 7, -4, 2, 39, 5, 23,
24, 15, 43, 36, 17, 8, -9, 2, 24, -11, -6, -11, -20, 10, -16, -22,
-23, -16, -1, -9, 3, 7, 14, 24, 19, -15, -15, -21, -26, 7, 19, -3,
-2, -1, 8, 14, 19, -6, -6, -31, -22, 8, -8, -16, -1, -6, 8, 18, 9,
-3, -15, -9, 18, 5, -12, -2, -11, -49, -14, -20, 2, 4, 27, 3, 7, 0,
9, -4, -3, 21, -3, -21, -19, -6, -16, 4, 9, 2, -18, -13, 0, -16, -28,
-38, -38, -11, -22, 1, -14, 32, 13, 38, 4, -15, -23, -34, -28, 22,
-7, -19, 2, 13, 21, -23, 3, 2, -12, -25, -21, -9, -8, -6, 5, -28, 14,
33, 64, 62, 34, 27, -11, -21, -10, 9, -20, -12, -2, -16, -3, 18, 11,
-24, -3, -36, -24, -39, -8, -14, -17, 6, 8, 5, 12, 74, 28, 32, 47,
23, 12, -26, -4, -29, 20, -22, -2, -13, -1, 16, -20, -8, -10, -10,
```

```
-17, -19, -44, -48, -13, -21, -8, -2, 37, 54, 20, 20, 5, 1, -20, -17,
-17, -7, 12, -18, -1, -1, 22, -2, 19, -1, -9, -43, -30, -34, -52,
-48, -31, 3, 37, 29, 47, 60, 26, -1, -35, -49, -56, -10, -1, -18,
-10, 16, -24, 13, -20, 10, -10, 6, 1, -13, -38, -48, -23, -18, -16,
2, 16, 37, 62, 61, 8, -37, -60, -46, -31, -37, 12, 5, -1, -20, 13,
-2, 11, -15, -1, -12, -23, -14, -4, 1, 7, -10, -9, 10, 19, 17, 52,
61, 7, -4, -3, 4, 1, -5, -28, -24, 18, -23, -14, 18, -12, -18, 23, 6,
26, 37, 5, 38, 11, 13, 6, -3, 27, 38, 68, 55, 23, 1, 29, -4, -4, -19,
24, 24, -14, 10, -23, 23, -4, 18, -12, 25, 37, 42, 43, 30, 15, 26,
-7, -37, 7, 14, 15, 38, 9, 7, 45, 33, 41, 19, -7, 12, 3, 15, 9, -10,
0, -2, -18, -2, 6, 35, 44, 3, 3, -10, -35, -15, -50, -4, 2, 24, 35,
53, 43, 19, 38, 41, 20, -1, -12, 8, 1, 21, 6, 0, 17, 2, 41, 35, 35,
21, 13, 13, -17, -36, -54, -36, 8, 20, 34, 13, 30, 57, 8, 4, 12, 18,
18, -22, -11, -11, 4, -20, 7, 23, 12, 41, 14, 25, 16, -8, -16, 8, -2,
-31, 1, -8, 27, 3, 15, 3, 20, 7, -24, 17, 15, -10, -22, -13, -1, -5,
-20, 3, 14, -7, 4, 14, -6, 12, -28, -15, 17, -17, -15, 14, 10, -7,
26, -1, -17, -19, 3, -8, -13, 3, -9, -3, -7, 18, -5, -3, 5, 8, 7, 25,
1, 27, 37, 28, 53, 4, 18, 24, 14, 8, -15, -16, -19, 6, -19, -7, -13,
-24, 13, 21, 0, -4, 7, 19, 15, -16, 4, -19, -6, 13, 17, 31, 39, 24,
0, 18, -5, 8, -1, -25, 6, -6, -25, -24, 16, -4, 18, -24, 5, -2, 15,
14, -12, -4, -14, -5, -18, 8, -2, -13, 15, -1, 4, -32, 6, 2, -27, 4,
-16, 21, 14, 5, 7, 1, -10, -18, -20, -15, 16, -21, 22, -23, -7, 17,
-2, -25, 7, -11, 5, 6, 6, -10, 3, -16, 8, -25, 7, 12, 21, -6, -20,
-5, -14, -15, -14, 24, 2, -19, 12, -10, -9, -3, -22, 20, 2, 3, -23,
11, -6, 5, 1, 18, 20, 7, 7, 10, 17, -12, 9, -10, 1, -4, -19, 15, -16,
-14, 18, 22, -4, -19, -10, 19, 6, 11, 18, -20, -15, -21, -10, -3, 21,
18, -10, -17, 0, 3, -9, -7, 18, -7, -13, 12, -8, -18, 4, 5, 6, 20, 5,
1, 12, 4, -2, 11, 5, 14, 16, -22, -15, -8, 12, -24, -21, 0, 20, 12,
6, -3};
//    logic signed [7:0] nineweights[783:0] = {11, 9, -21, -10, 16,
-13, -9, 5, 11, -12, 9, 9, -23, -18, 11, -23, -17, -9, 24, -21, 16,
-3, 4, -2, 16, 5, 24, 0, 17, -12, -23, 19, 8, 24, -5, -15, 28, 27,
-4, -6, 5, -5, 28, 1, -12, 18, -11, 18, 6, -20, -4, -15, -1, 20, -23,
-5, 11, -22, -8, 16, -4, 26, 22, 30, 33, 54, 56, 21, 33, 36, 40, 37,
43, 39, 16, 27, 1, -2, -4, 16, 12, 0, 25, -15, 18, -24, 18, -7, 11,
35, 28, 37, 18, 25, 19, 15, -18, -20, -8, 4, -15, 12, 3, 21, -1, 7,
-19, 8, 23, -6, -18, 16, 8, -21, -17, 10, -11, 9, 25, 33, -8, -3,
-20, -38, -31, -17, -7, -31, -30, -2, -7, -36, -31, -30, -28, 21, 8,
-20, -8, -1, -21, 11, 11, 16, -19, 3, -8, 18, -16, -15, -26, -10,
```

-58, -50, -21, -16, -14, -45, -15, -21, -25, -3, -27, 17, 9, 8, 13,
-10, -21, 19, 10, -19, 5, -23, -21, -25, 4, -5, -35, -35, -23, -28,
-8, -54, -41, -65, -57, -37, -39, -6, 7, 4, 14, -5, -18, 12, -7, -19,
-11, -3, -24, -7, -1, -13, 3, -31, -6, -16, -20, -35, -38, -48, -41,
-76, -42, -38, -37, -26, -8, -27, 11, -15, -13, 0, -23, 13, -25, 20,
-23, -23, -34, -6, 5, -29, -43, -24, -3, -4, -29, -52, -53, -27, -63,
-40, -21, -12, -20, 1, -16, 17, 15, -8, -19, -14, -4, -24, 5, -13, 5,
-34, -19, -27, 6, 21, 18, 5, -4, -15, -31, -24, -10, 1, -35, -28,
-30, 8, 6, -12, -10, 7, 15, 14, -8, -17, -31, -41, -32, -42, -15, -3,
-5, 31, 33, 3, -21, -20, 25, -1, -1, 11, 6, -6, -27, 15, -5, -20, -6,
-5, -16, -22, 3, 1, -36, 2, -26, -38, -16, -22, 26, 33, 23, -28, -15,
1, 6, 14, 39, 11, 5, -19, -14, 6, -8, -5, -2, 1, 7, 13, -19, 1, -39,
-38, -10, -15, 9, 31, 34, 63, -9, -30, -1, 4, 13, 0, 12, 31, 28, -8,
27, -9, -18, 10, 9, 17, -23, -13, -13, -5, -32, -1, -14, -1, 23, 52,
68, 54, 29, 15, 24, -14, 16, 6, 6, 13, 3, 35, 17, -13, 25, 2, -24,
-13, 19, -23, -21, -16, 15, 17, 0, 20, 38, 65, 66, 50, 44, 17, -2, 7,
-4, 0, 16, 46, 52, 44, 7, 26, 5, 5, -14, 8, 23, -3, 9, -27, 17, -7,
41, 29, 70, 43, 63, 45, 30, 15, -14, -25, -6, 46, 43, 15, 39, 23, 21,
31, -1, -6, 20, -1, 15, 12, -14, -5, -11, 20, 21, 33, 64, 56, 55, 12,
-4, -1, -35, 10, 3, 33, 38, 20, 41, 26, -2, 9, -16, 10, -11, -23, 1,
-21, -17, -11, 12, -24, -10, 30, 4, 8, 2, -24, -31, -20, -14, 10, 49,
52, 38, 35, 27, 14, 10, -24, -11, -7, -1, 15, -4, -24, 4, -8, -41,
-5, -8, 6, 8, 4, 0, 20, 22, 22, 17, 4, 25, -4, 3, 4, -33, -8, -21,
-30, 2, 17, 15, 5, -4, 7, 10, -19, -38, -53, -31, -25, -10, 3, 30,
38, 54, 56, 48, 15, 21, -17, -11, -34, -30, -26, -3, -2, -24, -6, -6,
-4, 11, -26, -31, -13, -15, -19, -30, -19, -7, 23, 54, 100, 106, 101,
49, 44, -12, -5, -34, -38, -25, -39, -24, -16, -5, -21, 24, -4, 22,
-12, -5, 5, -41, -41, -16, -44, -21, 11, -2, 25, 30, 30, 4, 14, -4,
-38, -10, -6, -27, -11, 7, -16, 16, -5, 11, 20, -17, 19, 13, -17, -4,
-14, -4, -50, -46, -44, -64, -63, -62, -49, -19, -42, -27, -24, -32,
-36, 15, 0, 19, -6, -18, -9, 11, -2, 6, 8, -6, -14, 15, -8, 4, -25,
-13, -9, -38, -25, -44, -29, -32, -26, -5, 0, 11, -27, -10, 14, -15,
14, -24, 14, -7, 9, -6, 20, 22, 24, -16, -15, -19, 19, 18, -14, 0,
-6, -19, -24, 5, -11, -5, 13, -16, 6, -7, -6, -8, -20, 14, 18, -11,
-13, -6, 4, -24, 19, 24, -18, 22, -17, -8, 4, -12, 11, -19, -9, -16,
5, -9, 14, -15, 13, 24, 7, 19, -15, 24, -23, 23, -10, -3, -10, -12,
24, 18, 12, 15, 2, -14, -1, -16, -23, -7, 14, -3, 18, 0, -2, 11, -2,
13, 0, -2, 1, -16, -13, -23, -13, -15, 16, 9, -16, 19, 24, -7, -22,
19, 3, -16, -4, -1, -4, -22, 2, -22, -9, -4, 18, -15, 0, -24, 0, -10,

```
-8, 14, 4};

    logic [3:0] value;
    logic [9:0] state = 0;

    logic signed [16:0] zero = 0;
    logic signed [16:0] one = 0;
    logic signed [16:0] two = 0;
    logic signed [16:0] three = 0;
    logic signed [16:0] four = 0;
    logic signed [16:0] five = 0;
    logic signed [16:0] six = 0;
    logic signed [16:0] seven = 0;
    logic signed [16:0] eight = 0;
    logic signed [16:0] nine = 0;

    always_ff @(posedge clk_in)begin
        if(rst_in)begin
            state <= 0;
        end else begin
            state <= state + 1'b1;

            if(state<10'b1100010000)begin // state<784
                if(val_in[state]==1'b1)begin
                    zero <= zero + zeroweights[state];
                    one <= one + oneweights[state];
                    two <= two + twoweights[state];
                    three <= three + threeweights[state];
                    four <= four + fourweights[state];
                    five <= five + fiveweights[state];
                    six <= six + sixweights[state];
                    seven <= seven + sevenweights[state];
                    eight <= eight + eightweights[state];
                    nine <= nine + nineweights[state];
                end
            end

            if (state>10'b1100100000) begin
                outputval <= value;
```

```
                state <= 0;
                zero <= 0;
                one <= 0;
                two <= 0;
                three <= 0;
                four <= 0;
                five <= 0;
                six <= 0;
                seven <= 0;
                eight <= 0;
                nine <= 0;
            end
        end
    end

    always_comb begin

        if(zero>one & zero>two & zero>three & zero>four & zero>five &
zero>six & zero>seven & zero>eight & zero>nine)begin //zero
            value = 4'b0000;
        end else if (one>zero & one>two & one>three & one>four &
one>five & one>six & one>seven & one>eight & one>nine)begin //one
            value = 4'b0001;
        end else if (two>zero & two>one & two>three & two>four &
two>five & two>six & two>seven & two>eight & two>nine)begin //two
            value = 4'b0010;
        end else if (three>zero & three>one & three>two & three>four
& three>five & three>six & three>seven & three>eight &
three>nine)begin //three
            value = 4'b0011;
        end else if (four>zero & four>one & four>two & four>three &
four>five & four>six & four>seven & four>eight & four>nine)begin
//four
            value = 4'b0100;
        end else if (five>zero & five>one & five>two & five>three &
five>four & five>six & five>seven & five>eight & five>nine)begin
//five
            value = 4'b0101;
        end else if (six>zero & six>one & six>two & six>three &
```

```
six>four & six>five & six>seven & six>eight & six>nine)begin //six
            value = 4'b0110;
        end else if (seven>zero & seven>one & seven>two & seven>three
& seven>four & seven>five & seven>six & seven>eight &
seven>nine)begin //seven
            value = 4'b0111;
        end else if (eight>zero & eight>one & eight>two & eight>three
& eight>four & eight>five & eight>six & eight>seven &
eight>nine)begin //eight
            value = 4'b1000;
        end else if (nine>zero & nine>one & nine>two & nine>three &
nine>four & nine>five & nine>six & nine>seven & nine>eight)begin
//nine
            value = 4'b1001;
        end

    end

endmodule
```

VGA.sv

```
//////////////////////////////////////////////////////////////////////
//////////
//
// pong_game: the GUI itself!
//
//////////////////////////////////////////////////////////////////////
//////////

module pong_game (
   input vclock_in,        // 65MHz clock
   input reset_in,         // 1 to initialize module
   input up_in,            // 1 when paddle should move up
```

```
   input down_in,          // 1 when paddle should move down
   input [3:0] valuecalc,
   input [3:0] value64,
   input [783:0] testblock,
   input [3:0] pspeed_in,  // puck speed in pixels/tick
   input [10:0] hcount_in, // horizontal index of current pixel
(0..1023)
   input [9:0]  vcount_in, // vertical index of current pixel
(0..767)
   input hsync_in,         // XVGA horizontal sync signal (active
low)
   input vsync_in,         // XVGA vertical sync signal (active low)
   input blank_in,         // XVGA blanking (1 means output black
pixel)

   output logic phsync_out,       // pong game's horizontal sync
   output logic pvsync_out,       // pong game's vertical sync
   output logic pblank_out,       // pong game's blanking
   output logic [11:0] pixel_out  // pong game's pixel  // r=11:8,
g=7:4, b=3:0
   );

    // these are the parameters for display, paddle, and death star
dimensions
    parameter DISPLAY_HEIGHT = 768;
    parameter DISPLAY_WIDTH = 1024;
    parameter PADDLE_HEIGHT = 120;
    parameter PADDLE_WIDTH = 16;
    parameter STAR_HEIGHT = 240;
    parameter STAR_WIDTH = 256;
    parameter OVERLAP_THRESHOLD = 10; // how far an overlap is
interpreted as a collision
//   logic [2:0] checkerboard;

//   // REPLACE ME! The code below just generates a color
checkerboard
//   // using 64 pixel by 64 pixel squares.

   assign phsync_out = hsync_in;
```

```
   assign pvsync_out = vsync_in;
   assign pblank_out = blank_in;
//   assign checkerboard = hcount_in[8:6] + vcount_in[8:6];

//   // here we use three bits from hcount and vcount to generate the
//   // checkerboard
//   assign pixel_out = {{4{checkerboard[2]}}, {4{checkerboard[1]}},
{4{checkerboard[0]}}} ;

    // instantiate the paddle as a white rectangular blob
    logic [9:0] paddle_y = 320; // start out at 320 (halfway down)
    logic [11:0] paddle_pixel;
//    blob #(.WIDTH(16),.HEIGHT(128),.COLOR(12'hFFF))   // white!
//
paddle1(.x_in(11'd0),.y_in(paddle_y),.hcount_in(hcount_in),.vcount_in
(vcount_in),.pixel_out(paddle_pixel));

    // instantiate the red block as a red square blob
    logic [11:0] redblock_pixel;
    blob #(.WIDTH(1024),.HEIGHT(768),.COLOR(12'hFFF))   // red hF00

redblock(.x_in(0),.y_in(0),.hcount_in(hcount_in),.vcount_in(vcount_in
),.pixel_in(12'b111111111111),.pixel_out(redblock_pixel));

    logic [11:0] whitepixel;
    blob #(.WIDTH(170),.HEIGHT(80),.COLOR(12'hFFF))   // red hF00

whiteblock(.x_in(0),.y_in(0),.hcount_in(hcount_in),.vcount_in(vcount_
in),.pixel_in(mnpixel),.pixel_out(whitepixel));

    // death star picture_blob
    logic [11:0] star_y = 63;
    logic [11:0] star_x = 131;
    logic [11:0] star_pixel;
    logic [1:0] orientation = 2'b00; // 00 = SE, 01 = SW, 10 = NW, 11
= NE
    picture_blob
deathstar_test(.pixel_clk_in(vclock_in),.x_in(star_x),.hcount_in(hcou
nt_in),.y_in(star_y),.vcount_in(vcount_in),
```

```
    .pixel_in(redblock_pixel),.pixel_out(star_pixel));

    // first digit blob
    logic [11:0] d1_y = 178; //156 //115
    logic [11:0] d1_x = 656; //555 //525
    logic [11:0] d1pixel;
    digit_blob
digitcalc(.pixel_clk_in(vclock_in),.adjust(valuecalc),.x_in(d1_x),.hc
ount_in(hcount_in),.y_in(d1_y),.vcount_in(vcount_in),
    .pixel_in(star_pixel),.pixel_out(d1pixel));

    // second digit blob
    logic [11:0] d2_y = 483; //455 // 420
    logic [11:0] d2_x = 656; //559 // 525
    logic [11:0] d2pixel;
    digit_blob
digit64(.pixel_clk_in(vclock_in),.adjust(value64),.x_in(d2_x),.hcount
_in(hcount_in),.y_in(d2_y),.vcount_in(vcount_in),
    .pixel_in(d1pixel),.pixel_out(d2pixel));


    // mnist digit blob
    logic [11:0] mn_y = 268; //210 //205
    logic [11:0] mn_x = 216; //55 //85
    logic [11:0] mnpixel;
    mnist_blob
handwritten(.pixel_clk_in(vclock_in),.testblock(testblock),.x_in(mn_x
),.hcount_in(hcount_in),.y_in(mn_y),.vcount_in(vcount_in),
    .pixel_in(d2pixel),.pixel_out(mnpixel));

    // making the "split" death star with a superimposed black sprite
    logic [11:0] split_star;
    sprite #(.WIDTH(300),.HEIGHT(20),.COLOR(12'h000))   // black for
break!

horizbreak(.x_in(PADDLE_WIDTH),.y_in(star_y+STAR_HEIGHT/2-10),.hcount
_in(hcount_in),.vcount_in(vcount_in),
        .pixel_in(star_pixel),.pixel_out(split_star));
```

```
    logic state = 0;

    always_ff @(posedge vsync_in) begin
        if(reset_in)begin // reset all the sprite positions when BTNC
is pressed
            state <= 0;
            paddle_y <= 320;
            star_x <= 500;
            star_y <= 320;
            orientation <= 2'b00;
        end else begin
            if(state==0)begin // if state==1, then game over and
everything is frozen

                // paddle up/down logic

if(down_in&&(paddle_y<(DISPLAY_HEIGHT-PADDLE_HEIGHT-4)))begin
                        paddle_y <= paddle_y + 4;
                end else if(up_in && (paddle_y>4))begin
                        paddle_y <= paddle_y - 4;
                end

                // death star left-right collision logic

if(star_x>(DISPLAY_WIDTH-STAR_WIDTH-OVERLAP_THRESHOLD)) begin
                    // if hit right edge, bounce left
                    orientation[1]<=1'b1;
                end else if(star_x<OVERLAP_THRESHOLD) begin

if((star_y>(paddle_y-PADDLE_HEIGHT))&&(star_y<(paddle_y)))begin
                        // if hit paddle, bounce right
                        orientation[1]<=1'b0;
                    end else begin
                        // if hit left wall, game over
                        state<=1;
                    end
                end

                // death star up/down collision logic
```

```
                if
(star_y>(DISPLAY_HEIGHT-STAR_HEIGHT-OVERLAP_THRESHOLD)) begin
                    orientation[0]<=1'b1;
                end else if (star_y<OVERLAP_THRESHOLD) begin
                    orientation[0]<=1'b0;
                end

                // death star motion based on the "orientation"
variable
                case(orientation)
                    default: begin //southeast = +x, +y
                        if(star_y<(DISPLAY_HEIGHT-STAR_HEIGHT))begin
star_y <= star_y + pspeed_in; end
                        if(star_x<(DISPLAY_WIDTH-STAR_WIDTH)) begin
star_x <= star_x + pspeed_in; end
                    end
                    2'b10: begin //southwest = -x, +y
                        if(star_y<(DISPLAY_HEIGHT-STAR_HEIGHT))begin
star_y <= star_y + pspeed_in; end
                        if(star_x>0)begin star_x <= star_x -
pspeed_in; end
                    end
                    2'b11: begin //northwest = -x, -y
                        if(star_y>0)begin star_y <= star_y -
pspeed_in; end
                        if(star_x>0)begin star_x <= star_x -
pspeed_in; end
                    end
                    2'b01: begin //northeast = +x, -y
                        if(star_y>0)begin star_y <= star_y -
pspeed_in; end
                        if(star_x<(DISPLAY_WIDTH-STAR_WIDTH)) begin
star_x <= star_x + pspeed_in; end
                    end
                endcase
            end
        end
    end
```

```
//    always_comb begin
//        logic [11:0] temppixel = 12'b0000_0000_0000;
//
if((vcount_in>paddle_y)&&(vcount_in<(paddle_y+PADDLE_HEIGHT)))begin
//            temppixel = temppixel | paddle_pixel;
//        end
//
if((vcount_in>star_y)&&(vcount_in<(star_y+STAR_HEIGHT))&&(hcount_in>s
tar_x)&&(hcount_in<(star_x+STAR_WIDTH)))begin
//            temppixel = temppixel | star_pixel;
//        end
//        pixel_out = temppixel;
//    end

    // this shows the split star bar blob when the game is over.
    always_comb begin
        if(state==0)
            pixel_out = paddle_pixel | whitepixel;
        else
            pixel_out = paddle_pixel | split_star;
    end
//    assign pixel_out = paddle_pixel | star_pixel;
endmodule

module synchronize #(parameter NSYNC = 3)  // number of sync flops.
must be >= 2
                  (input clk,in,
                   output logic out);

  logic [NSYNC-2:0] sync;

  always_ff @ (posedge clk) begin
    {out,sync} <= {sync[NSYNC-2:0],in};
  end
endmodule


//////////////////////////////////////////////////////////////////
/////////////
```

```
// Update: 8/8/2019 GH
// Create Date: 10/02/2015 02:05:19 AM
// Module Name: xvga
//
// xvga: Generate VGA display signals (1024 x 768 @ 60Hz)
//
//                              ---- HORIZONTAL -----
------VERTICAL -----
//                              Active                    Active
//                    Freq      Video   FP  Sync   BP      Video   FP
Sync  BP
//   640x480, 60Hz    25.175    640     16    96   48       480    11
2    31
//   800x600, 60Hz    40.000    800     40   128   88       600     1
4    23
//   1024x768, 60Hz   65.000    1024    24   136  160       768     3
6    29
//   1280x1024, 60Hz  108.00    1280    48   112  248       768     1
3    38
//   1280x720p 60Hz   75.25     1280    72    80  216       720     3
5    30
//   1920x1080 60Hz   148.5     1920    88    44  148      1080     4
5    36
//
// change the clock frequency, front porches, sync's, and back
porches to create
// other screen resolutions
////////////////////////////////////////////////////////////////////
//////////

module xvga(input vclock_in,
            output logic [10:0] hcount_out,    // pixel number on
current line
            output logic [9:0] vcount_out,     // line number
            output logic vsync_out, hsync_out,
            output logic blank_out);

   parameter DISPLAY_WIDTH  = 1024;      // display width
   parameter DISPLAY_HEIGHT = 768;       // number of lines
```

```
   parameter  H_FP = 24;                 // horizontal front porch
   parameter  H_SYNC_PULSE = 136;        // horizontal sync
   parameter  H_BP = 160;                // horizontal back porch

   parameter  V_FP = 3;                  // vertical front porch
   parameter  V_SYNC_PULSE = 6;          // vertical sync
   parameter  V_BP = 29;                 // vertical back porch

   // horizontal: 1344 pixels total
   // display 1024 pixels per line
   logic hblank,vblank;
   logic hsyncon,hsyncoff,hreset,hblankon;
   assign hblankon = (hcount_out == (DISPLAY_WIDTH -1));
   assign hsyncon = (hcount_out == (DISPLAY_WIDTH + H_FP - 1));
//1047
   assign hsyncoff = (hcount_out == (DISPLAY_WIDTH + H_FP +
H_SYNC_PULSE - 1));  // 1183
   assign hreset = (hcount_out == (DISPLAY_WIDTH + H_FP +
H_SYNC_PULSE + H_BP - 1));  //1343

   // vertical: 806 lines total
   // display 768 lines
   logic vsyncon,vsyncoff,vreset,vblankon;
   assign vblankon = hreset & (vcount_out == (DISPLAY_HEIGHT - 1));
// 767
   assign vsyncon = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP -
1));  // 771
   assign vsyncoff = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP +
V_SYNC_PULSE - 1));  // 777
   assign vreset = hreset & (vcount_out == (DISPLAY_HEIGHT + V_FP +
V_SYNC_PULSE + V_BP - 1)); // 805

   // sync and blanking
   logic next_hblank,next_vblank;
   assign next_hblank = hreset ? 0 : hblankon ? 1 : hblank;
   assign next_vblank = vreset ? 0 : vblankon ? 1 : vblank;
   always_ff @(posedge vclock_in) begin
      hcount_out <= hreset ? 0 : hcount_out + 1;
```

```
      hblank <= next_hblank;
      hsync_out <= hsyncon ? 0 : hsyncoff ? 1 : hsync_out;  // active
low

      vcount_out <= hreset ? (vreset ? 0 : vcount_out + 1) :
vcount_out;
      vblank <= next_vblank;
      vsync_out <= vsyncon ? 0 : vsyncoff ? 1 : vsync_out;  // active
low

      blank_out <= next_vblank | (next_hblank & ~hreset);
   end
endmodule

//////////////////////////////////////////////////////////////////////
//
// blob: generate rectangle on screen
//
//////////////////////////////////////////////////////////////////////
module blob
   #(parameter WIDTH = 64,            // default width: 64 pixels
               HEIGHT = 64,           // default height: 64 pixels
               COLOR = 12'hFFF)  // default color: white
   (input [10:0] x_in,hcount_in,
    input [9:0] y_in,vcount_in,
    input [11:0] pixel_in,
    output logic [11:0] pixel_out);

   always_comb begin
      if  ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
           (vcount_in >= y_in && vcount_in < (y_in+HEIGHT)))
        pixel_out = COLOR;
      else
        pixel_out = pixel_in;
   end
endmodule
```

```
module sprite // version with a pixel in
   #(parameter WIDTH = 64,            // default width: 64 pixels
               HEIGHT = 64,           // default height: 64 pixels
               COLOR = 12'hFFF)  // default color: white
   (input [10:0] x_in,hcount_in,
    input [9:0] y_in,vcount_in,
    input logic [11:0] pixel_in,
    output logic [11:0] pixel_out);

   always_comb begin
      if  ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
            (vcount_in >= y_in && vcount_in < (y_in+HEIGHT)))
        pixel_out = COLOR;
      else
        pixel_out = pixel_in;
   end
endmodule

//////////////////////////////////////////////////
//
// picture_blob: display a picture
//
//////////////////////////////////////////////////
module picture_blob
   #(parameter WIDTH = 761,     // default picture width
               HEIGHT = 641)    // default picture height
   (input pixel_clk_in,
    input [10:0] x_in,hcount_in,
    input [9:0] y_in,vcount_in,
    input [11:0] pixel_in, // pipeline the square sprite
    output logic [11:0] pixel_out);

   logic [19:0] image_addr;   // num of bits for 761*641 ROM
   logic [7:0] image_bits, red_mapped, green_mapped, blue_mapped;

   // calculate rom address and read the location
   assign image_addr = (hcount_in-x_in) + (vcount_in-y_in) * WIDTH;
   image_rom  rom1(.clka(pixel_clk_in), .addra(image_addr),
.douta(image_bits));
```

```
   // use color map to create 4 bits R, 4 bits G, 4 bits B
   // since the image is greyscale, just replicate the red pixels
   // and not bother with the other two color maps.
//   red_coe rcm (.clka(pixel_clk_in), .addra(image_bits),
.douta(red_mapped));
//   green_coe gcm (.clka(pixel_clk_in), .addra(image_bits),
.douta(green_mapped));
//   blue_coe bcm (.clka(pixel_clk_in), .addra(image_bits),
.douta(blue_mapped));

   grayscale_coe rcm (.clka(pixel_clk_in), .addra(image_bits),
.douta(red_mapped));


//   logic[3:0] red_val = red_mapped[7:4]>>1+redblock_pixel[11:8]>>1;
//   logic[3:0] green_val =
red_mapped[7:4]>>1+redblock_pixel[7:4]>>1;
//   logic[3:0] blue_val = red_mapped[7:4]>>1+redblock_pixel[3:0]>>1;
    logic[3:0] red_val;
    assign red_val = (red_mapped[7:4]*3>>2)+(pixel_in[11:8]>>2);
    logic[3:0] green_val;
    assign green_val = (red_mapped[7:4]*3>>2)+(pixel_in[7:4]>>2);
    logic[3:0] blue_val;
    assign blue_val = (red_mapped[7:4]*3>>2)+(pixel_in[3:0]>>2);

// set up an ila for debugging with Mark
//   ila_0
myila(.clk(pixel_clk_in),.probe0(pixel_out),.probe1(red_mapped),.prob
e2(pixel_in));

   // note the one clock cycle delay in pixel!
   always_ff @ (posedge pixel_clk_in) begin
//   pixel_out <= {red_val, green_val, blue_val}; // greyscale
     if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
          (vcount_in >= y_in && vcount_in < (y_in+HEIGHT)))
        // use MSB 4 bits
//        pixel_out <= {red_val, green_val, blue_val}; // greyscale
        pixel_out <= {red_mapped[7:4], red_mapped[7:4],
```

```
red_mapped[7:4]}; // greyscale
        //pixel_out <= {red_mapped[7:4], 8h'0}; // only red hues
        else pixel_out <= pixel_in;
   end


endmodule

//////////////////////////////////////////////////
//
// digit_blob: display a picture
//
//////////////////////////////////////////////////
module digit_blob
   #(parameter WIDTH = 48,     // default picture width
               HEIGHT = 48)    // default picture height
   (input pixel_clk_in,
    input [3:0] adjust,
    input [10:0] x_in,hcount_in,
    input [9:0] y_in,vcount_in,
    input [11:0] pixel_in, // pipeline the square sprite
    output logic [11:0] pixel_out);

   logic [19:0] image_addr;   // num of bits for 761*641 ROM
   logic [7:0] image_bits, red_mapped, green_mapped, blue_mapped;

   // calculate rom address and read the location
   assign image_addr = (hcount_in-x_in) + (vcount_in-y_in) * WIDTH +
2304*adjust;
   digit_rom  digit(.clka(pixel_clk_in), .addra(image_addr),
.douta(image_bits));

   // use color map to create 4 bits R, 4 bits G, 4 bits B
   // since the image is greyscale, just replicate the red pixels
   // and not bother with the other two color maps.
//   red_coe rcm (.clka(pixel_clk_in), .addra(image_bits),
.douta(red_mapped));
//   green_coe gcm (.clka(pixel_clk_in), .addra(image_bits),
.douta(green_mapped));
```

```
//   blue_coe bcm (.clka(pixel_clk_in), .addra(image_bits),
.douta(blue_mapped));

   grayscale_coe rcm (.clka(pixel_clk_in), .addra(image_bits),
.douta(red_mapped));

   // note the one clock cycle delay in pixel!
   always_ff @ (posedge pixel_clk_in) begin
//   pixel_out <= {red_val, green_val, blue_val}; // greyscale
     if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
          (vcount_in >= y_in && vcount_in < (y_in+HEIGHT))) begin
        // use MSB 4 bits
//        pixel_out <= {red_val, green_val, blue_val}; // greyscale
//        pixel_out <= {red_mapped[7:4], red_mapped[7:4],
red_mapped[7:4]}; // greyscale
//        pixel_out <= {red_mapped[7:4], 4'b0000,4'b0000}; // only
red hues

            if(red_mapped[7:4]>0)begin
                pixel_out <= {4'b0000,4'b0000, red_mapped[7:4]};
            end else begin
                pixel_out <= {4'b1111,4'b1111, 4'b1111};
            end

        end else pixel_out <= pixel_in;
   end

endmodule

//////////////////////////////////////////////////
//
// mnist_blob: display a picture
//
//////////////////////////////////////////////////
module mnist_blob
   #(parameter WIDTH = 224,     // default picture width
               HEIGHT = 224)    // default picture height
   (input pixel_clk_in,
    input [783:0] testblock,
```

```
    input [10:0] x_in,hcount_in,
    input [9:0] y_in,vcount_in,
    input [11:0] pixel_in, // pipeline the square sprite
    output logic [11:0] pixel_out);

   logic [19:0] image_addr;   // num of bits for 761*641 ROM
   logic [7:0] image_bits, red_mapped, green_mapped, blue_mapped;
   logic [9:0] index;
   logic testval;

   // calculate rom address and read the location
   assign image_addr = ((hcount_in-x_in)>>3) +
(((vcount_in-y_in)>>3)*28);
   assign index = 783 - image_addr;
   assign testval = testblock[index];

   // note the one clock cycle delay in pixel!
   always_ff @ (posedge pixel_clk_in) begin

       if(testval)begin
            red_mapped <= 8'b00000000;
        end else begin
            red_mapped <= 8'b11111111;
        end

//   pixel_out <= {red_val, green_val, blue_val}; // greyscale
     if ((hcount_in >= x_in && hcount_in < (x_in+WIDTH)) &&
          (vcount_in >= y_in && vcount_in < (y_in+HEIGHT)))
        // use MSB 4 bits
//        pixel_out <= {red_val, green_val, blue_val}; // greyscale
        pixel_out <= {red_mapped[7:4], red_mapped[7:4],
red_mapped[7:4]}; // greyscale
//        pixel_out <= {red_mapped[7:4], 4'b0000,4'b0000}; // only
red hues
        else pixel_out <= pixel_in;
   end

endmodule
```