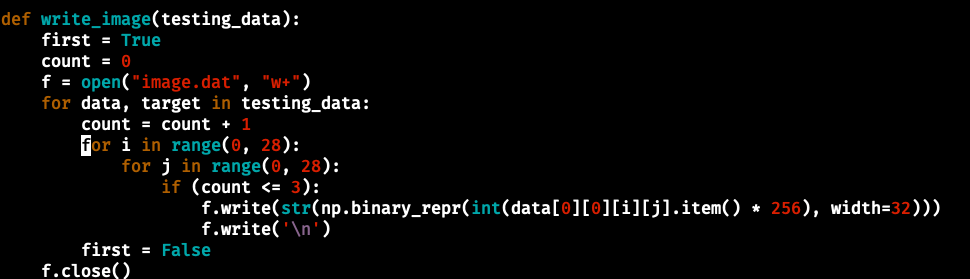
EECS4010 Final Project

Student ID: 106062113 Name: 鄭柏文

1. Design Concept

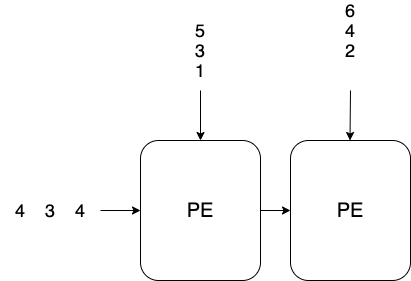
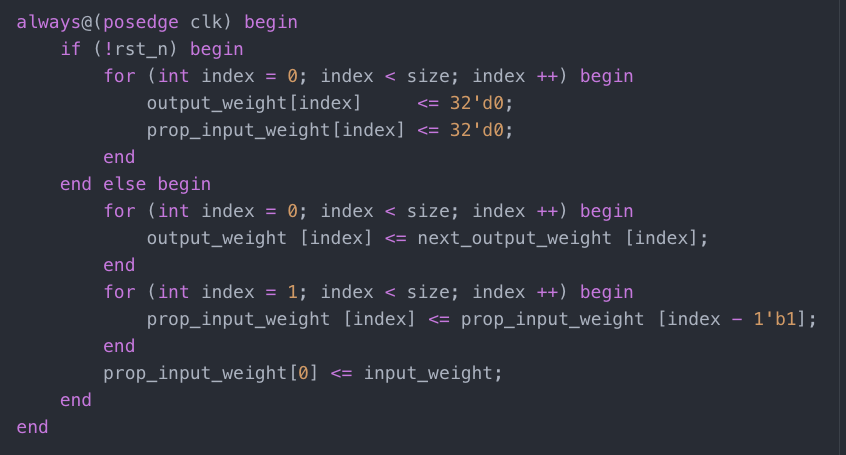
Before implementing the coprocessor, I trained the Mnist classifier model using python and the write the train weight into binary file called “weight.dat”. The model consists of three layer of fully connected neuron layer and the number of neuron is 784-100-10 Notice that to relieve us from handling floating point computation in verilog simulation, which can be really hard to implement, I multiply all weight by 256 and truncate the value decimal point after multiplication. Although truncating the floating point value will affect the classifier’s ability to classify input image, the loss in accuracy is 1 only one percent. The following python code shows the implementation of the process mentioned above.

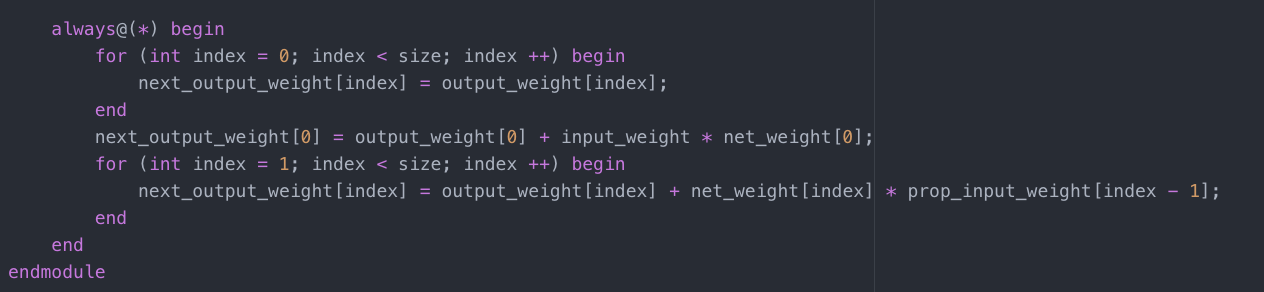


Next I need to write the input image from Mnist data set into a binary file in order to allow Verilog testbench to read it into the CPU’s memory. However, the data in the original is not in binary representation, instead, it’s compressed and contain some other meta data. Furthermore, the data is normalized and is in floating point. It took me some time to figure out how to extract the image data, but eventually I did work it out. With the help of data loader in pytorch, I was able to get the raw data of the image in tensor. After that, I multiply the normalized image data stored in tensor by 256 and truncate the value below the floating point. Finally, I write the integer image data into a binary file “image.dat”. The code below shows the process described above.

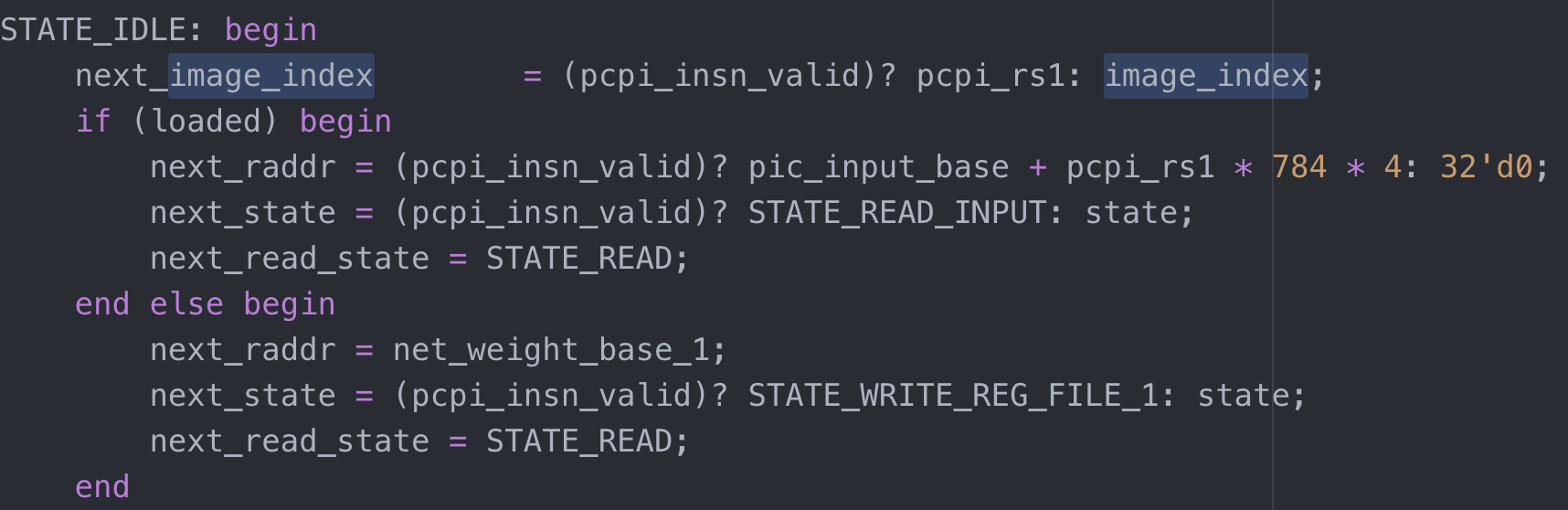
With weight and image taken care of, I can start to implement my hardware design. First we need a systolic array to compute the matrix multiplication in each layer quickly. The following diagram shows the structure of the systolic array. In the diagram, the systolic array will perform the following matrix computation:

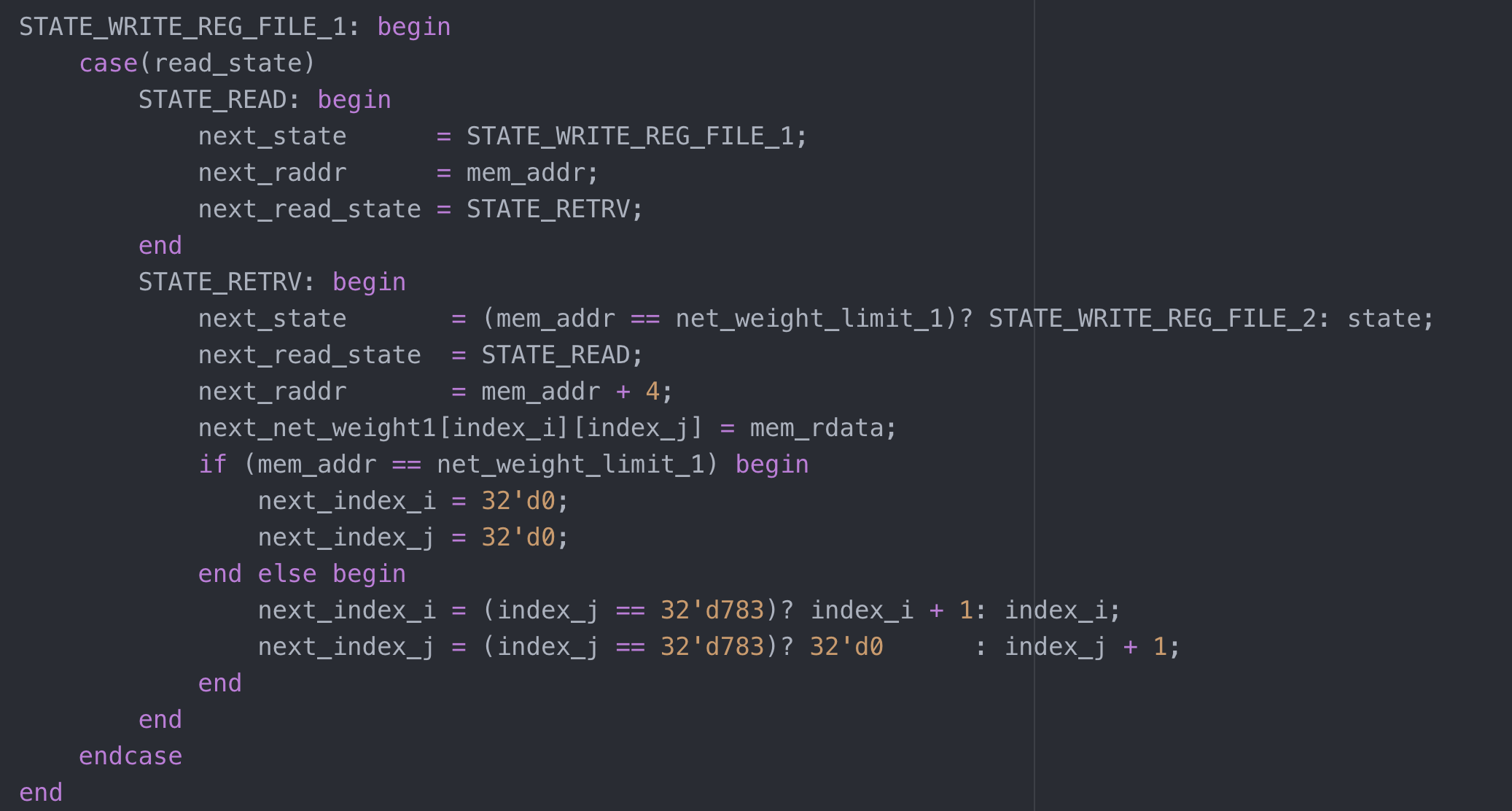
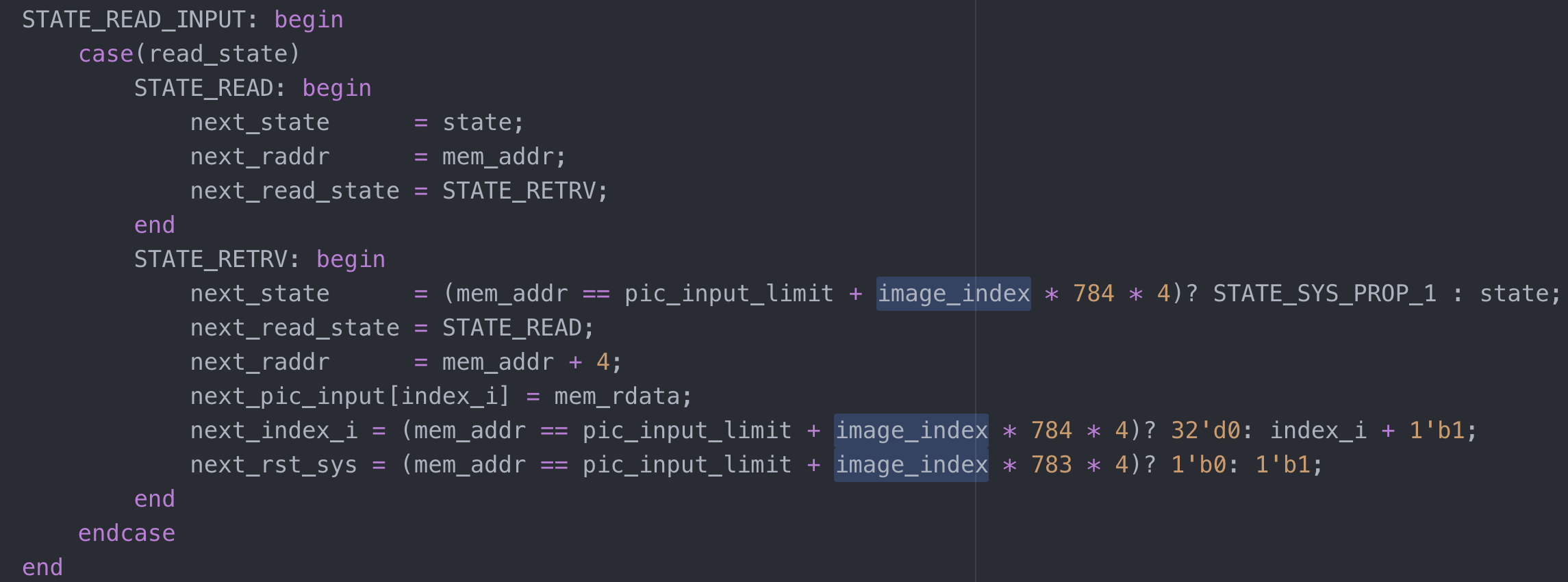
\*

To build systolic array in for our coprocessor, we need 100 PEs(since the first hidden layer have 100 neurons). Every cycle each processing element performs the multiply and accumulate (MAC) computation on the image input and the net weight and propagate the image input to the next processing element. Also, we added an reset signal rst\_n to allow the systolic array to be initialized. The sequential part of the design carry out the propagation of the image input and the reset. The combinational part take care of the MAC operation in the process elements. The output weight DFFs store the intermediate and eventually the final answer of the computation. The prop\_input\_weight DFFs is used to propagate the input image weight as its name indicated. The following code shows the implementation of the systolic array in Verilog (the for loop here is should be synthesizable.) The first image is the sequential part of the design and the second one is the combinational part of the design.

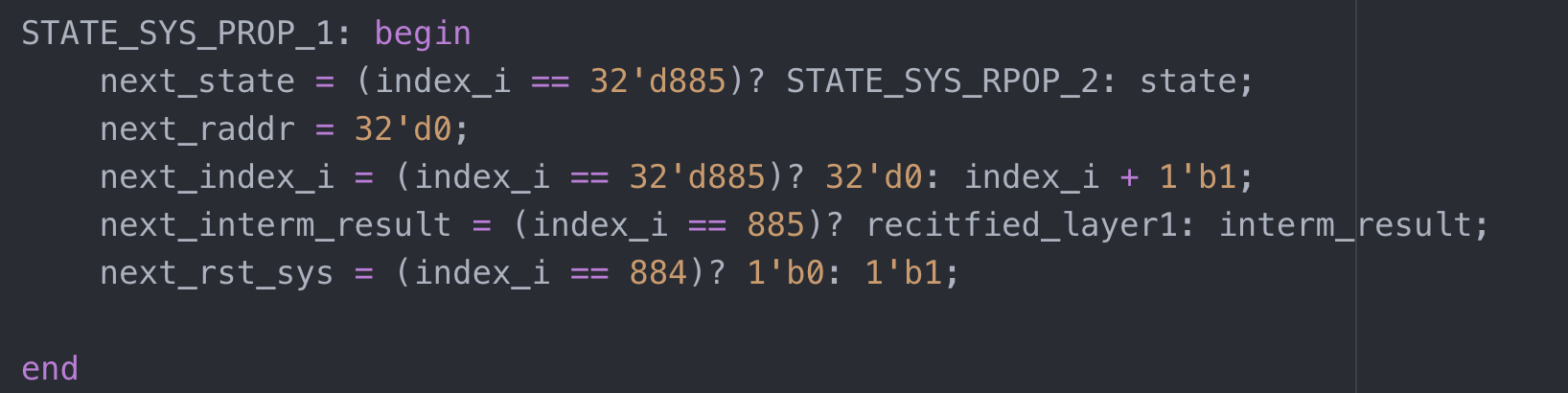
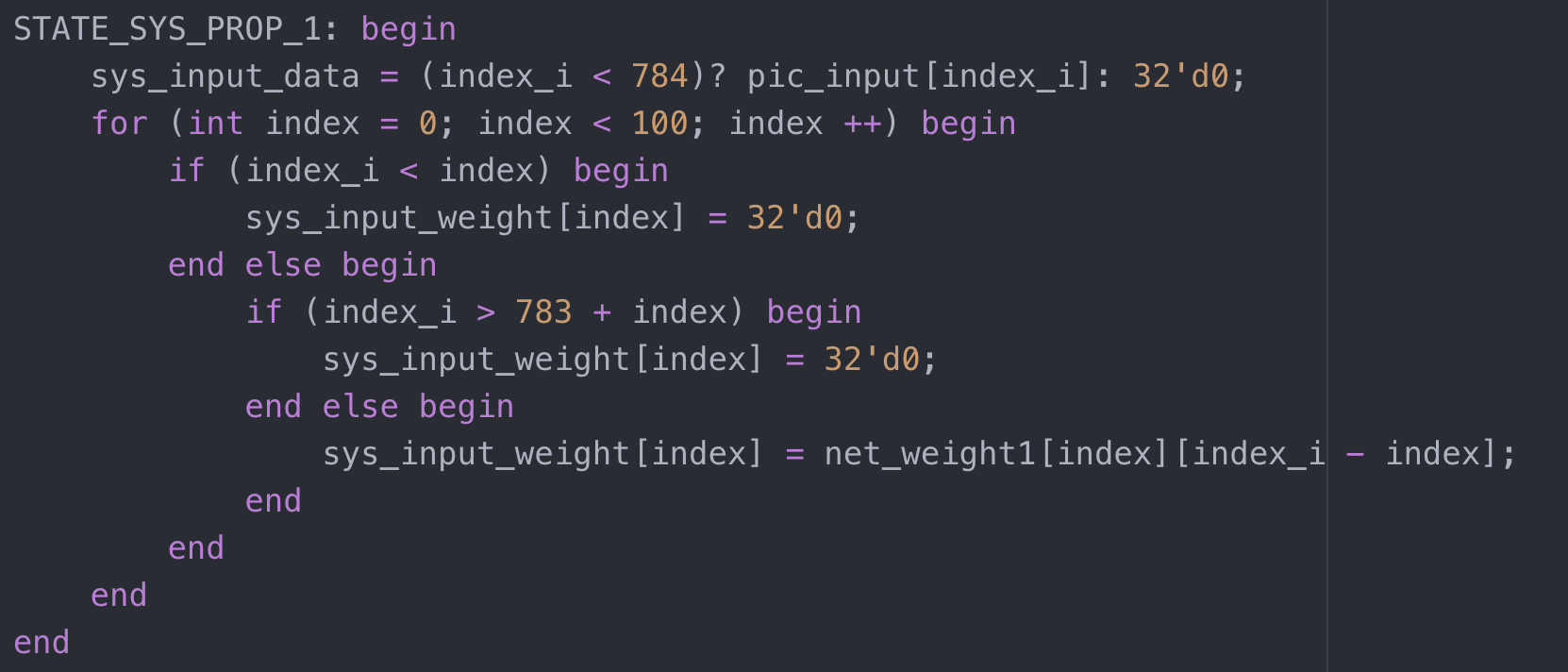
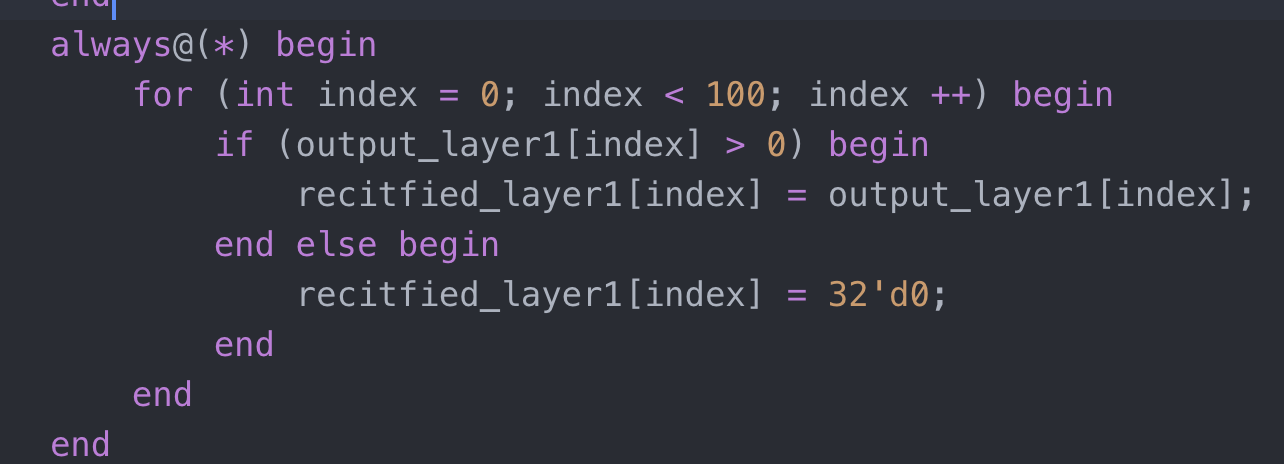
However, in order to use systolic array to compute the result of each layer, we need to feed multiple data every single cycle into the systolic array. At the mean time, the memory of the picorv CPU can only output one word of data every two cycle. This indicate that to use the systolic array to its max effect, we need to first read in the weight data and image data from the memory and cache them in large register files. This is a design tradeoff, to make the computation of the neural network faster, we sacrifices the area of the design to achieve this high performance.

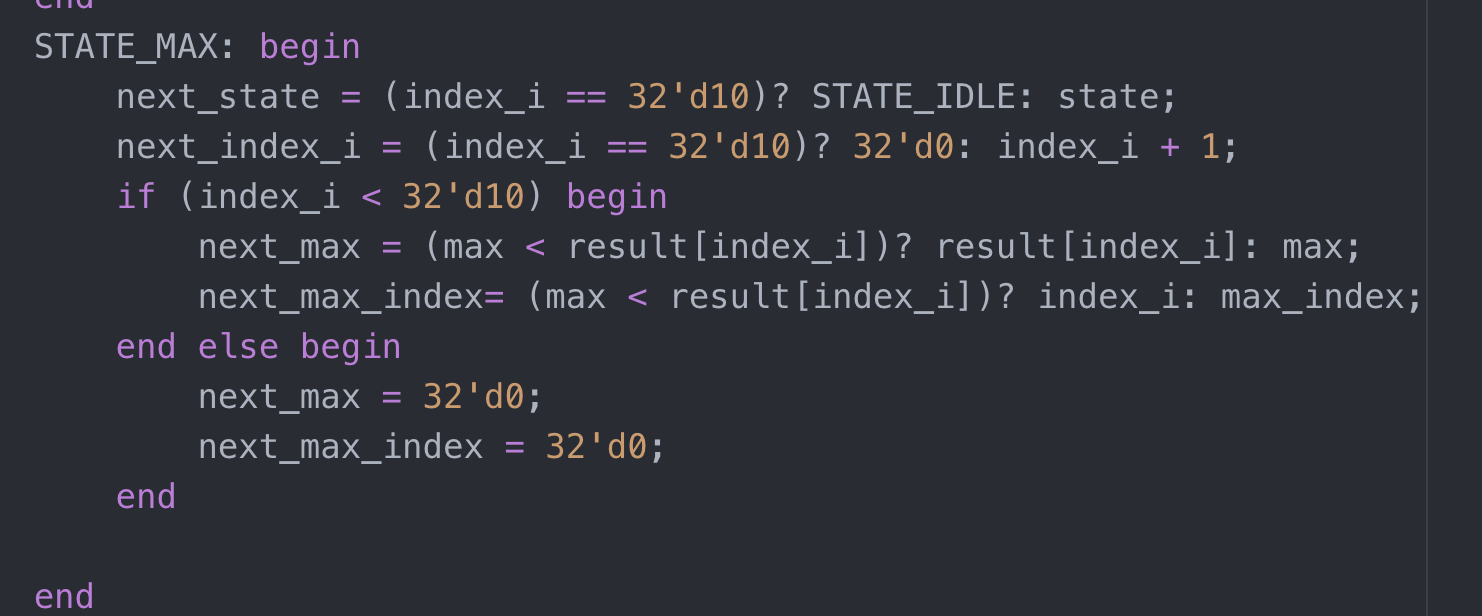
Next I am going to introduce the coprocessor. The coprocessor have seven states, I will explain them one by one in the following section. The first state is the STATE\_IDLE. In this state, the coprocessor sits idle and wait for the valid instruction, indicated by the pcpi\_insn\_valid signal, to come in and begin operation. Notice there’s an extra variable called loaded. If this is set, this means that this is probably the second or third time we are running the coprocessor and the weight are already inside the coprocessor, so we can proceed to read new image input without reading in weight again. When valid instruction comes in, it will set read the will first check whether the weight of the neural network has already been loaded in to the register files by checking the loaded signal. If so, We can set next\_state to STATE\_READ\_INPUT and set next\_raddr to the base address of the image we want to classify. If not, that means we need to first load all weights stored in the memory in to the register file. In this case, we set next\_state to STATE\_WRITE\_REG\_1 and next\_raddr to the base address of the first layer network weight to read in weights from the memory.

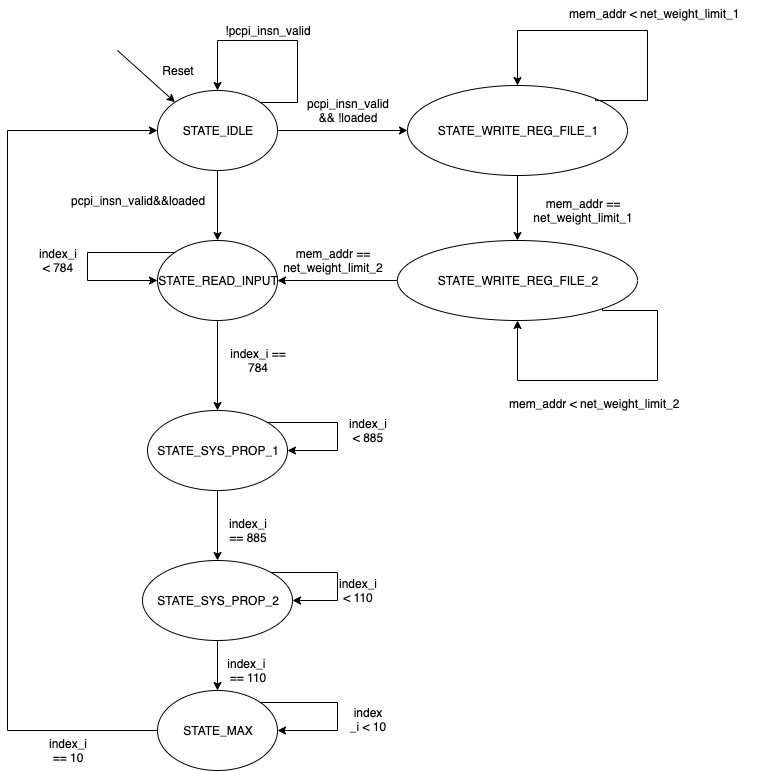
The second state and the third state are STATE\_WRITE\_REG\_FILE\_1 and STATE\_WRITE\_REG\_FILE\_2 respectively. In these two states, the coprocessor read in the weight of the neural network stored in memory. Controlling index\_i and index\_j to read the correct sequence of weight and store them in to the entry indicated by index\_i and index\_j in the 2D array. The register read\_state is present since reading memory takes 2 cycle to do so. In STATE\_READ, we issue the memory address to the memory, while in STATE\_RETRV we get the data from the memory. We read in data every two cycle by alternating between these two states and issue our read request and retrieve data from the memory. Notice that since the weight of the layer is flatten out and store in the memory sequentially, we can simply increment the mem\_addr by 4 (since we are reading in a word every time) and set read\_state to STATE\_READ to request the next data. Also, when we have read in all the weight in the first layer(check by comparing mem\_addr to the limit address of the first layer weight) the next\_state will be set to STATE\_WRITE\_REG\_FILE\_2. Similarly when we have read In all the weight of the second file, we can then proceed to read the input image. We do so by setting next\_state to STATE\_READ\_INPUT. Take notice that when we leave STATE\_WRITE\_REG\_FILE\_2, we have read in all the weights and these weight will stay in the coprocessor as long as the power supply to this module is not loss. Therefore, we set loaded to indicate that all the weights are loaded and there’s no need to read them in again if the CPU issue another instruction to the coprocessor in the future.

The Next state is STATE\_READ\_INPUT. In this state, we read in the binary value of the grayscale image stored in memory and store it in yet another register file. Notice that sys\_rst\_n is set to 0 to reset the systolic array during the second to last cycle of the state to initialize the systolic array. Similar to the previous states that read in the weight of the neural net. We use two read state to meet the timing requirement of the memory. Notice that since the 28 \* 28 image is flatten out in one big 784 vector, we only need to control the register index\_i in this state. When we read in the last word of the image (mem\_addr equals to the limit register of image data) we will set next state to STATE\_SYS\_PROP\_1 to starting propogating the first layer weight and the image into the systolic array to compute the value of the hidden layer.

The next two state STATE\_SYS\_PROP\_1 and STATE\_SYS\_PROP\_2. In these two state, we propagate our data into the systolic array and retrieve the store the result of the hidden layer and the output layer. Here I used unsynthesizable for loop to make my task easier. However, this can be turn into synthesizable code if wanted). I use index\_i as a counter to count what is the number of cycle we are currently in during the propagation process. With the help from the counter, the task becomes much easier. First we will discuss the image input part. The image input is pretty straight forward, each cycle we send in the next image data waiting at the entrance of the systolic array. However, if all data has been sent, we will propagate 0 into the systolic array(using the counter index\_i we can know how many data we have sent since it keeps record of the number of cycle we have been propagating and we send 1 data into the systolic array every cycle). Then there’s the weight coming from the top of the systolic array. If index\_i is less than the index of the processing element in the systolic array, that means that the image data have not yet arrive at that processing element just yet. Therefore, we should sent 0 into that processing element to keep the PE waiting. If value of index\_i is between the index of the PE and the size of the systolic array + its index, that means the valid image data is currently flowing through the PE so we should send our weight into the systolic array to perform the computation. Finally, if index\_i is greater than the size of the array + the index of the PE, that means the image data flowing through PE has ended there will be no computation performed anymore in that PE. In this case, we propagate 0 into the PE to preserve the value computed in the previous cycles. Also notice that after each propagation throught the systolic array, we pass the signal through a relu array to implement relu non-linearity in the model.

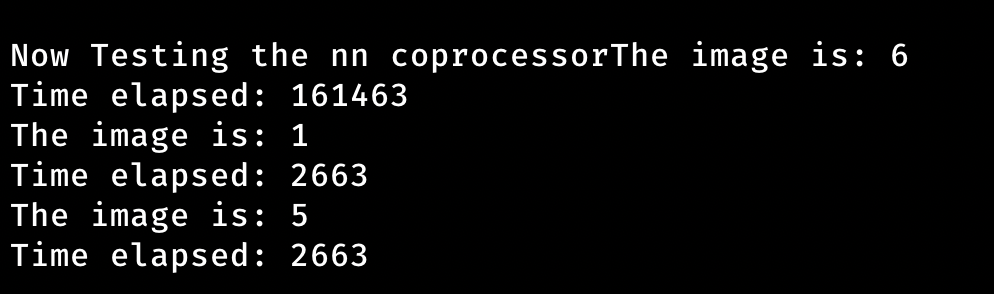
weight propagation & relu

Last, after obtaining the value of the output layer, we can simply find the max value index and and output that index, finishing our work and set coprocessor back to idle.

Here is the state transition diagram of my design.

1. Simulation and Discussion

Because the memory of the processor is limited, I only loaded three image into memory to test the correctness of the module using c program. We call the hardware nn instruction and give the function the image index we want to classify. (To know the image we are running, I provided an python program to plot the binary image of the grayscale image of the three image in the memory. After running the program, I did verify that the implementation of the module is correct (we send in image of digit 6, 1 and 5. And the output showed that it matches the image.

The image above shows the running result. As mentioned above, all three classification is correct. Also, we can see that the first run took much longer since it needs to read in all the weight of the neural net. After the first run the running time is well within 3000 tick. Just image how long this will take if we run the neural net with pure C software.

1. Summary

This project took me about 1 month to complete. During this one month, I encounter many problems. One of them is the trade off I mentioned in the previous section. To speed up the computation process, I needed to use huge register files. Also, since this is actually the first time I program pytorch, I spend quite some time to implement the neural network. However, these hard work did paid off at the end and I was able to finishi this project. Finally, I want to thank the professor and the TAs of this class. When I post a question on ILMS, my problem was always taken seriously and answered quickly. Also, the homework are challenging and I bet that you guys do spend a lot of time making those up. After all, I really enjoyed my time in this class! Thank you ☺.