CSEE 4840 Embedded System Design
Final Report

Project: CNN Accelerator

Haichun Zhao (hz2833)
Haomiao Li (hl3619)
Qixiao Zhang (qz2487)
Tianchen Yu (ty2485)
Yue Niu (yn2433)

Contents

1. Introduction

FPGAs have gained popularity as an accelerator for neural network inference due to their high-performance computing capabilities and flexibility. FPGA-based hardware accelerators can achieve high throughput and low latency, enabling real-time inference and training of large-scale models. They have shown promising results in energy efficiency and outperforming GPU-based solutions in terms of performance per watt. For our project, we plan to use the given SoC as a CNN (Convolution Neural Network) accelerator for a typical image recognition task.

CNN is a standard neural network structure that is particularly useful for image classification tasks. Its structure usually consists of a combination of convolutional layers, fully connected layers, pooling layers, flattening layers. The recognition results could be obtained by comparing the neuron values in the output layer. The core operation in CNN is Multiply and Accumulate (MAC), which is essentially doing multiplication and addition repetitively. Dedicated MAC module could complete this process efficiently. Our idea is to design a multi-MAC neural processing unit to realize parallelism during the inference process of CNN, hence the inference could be accelerated.

2. Milestones

1. CNN architecture prototype in MATLAB and implementation research

2. FPGA hardware upgrade to include memory blocks and optimized interface

3. Software controller development and overall system integration with benchmarking

3. Overall Block Diagram

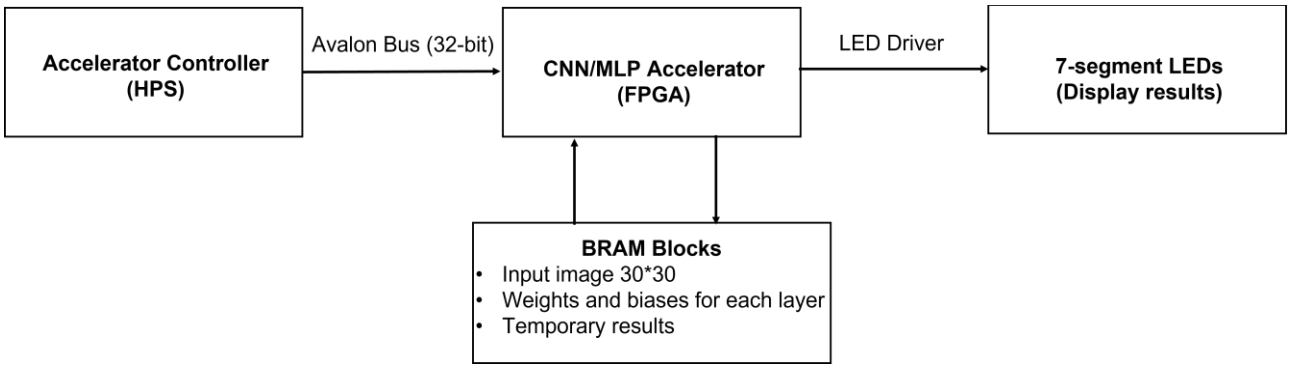

Figure 1. Overall block diagram of the system

The overall system block diagram of our project is shown in Figure 1. The input to the accelerator will be pre-stored image pixel value, network weight, network bias or the computation results from last layer. The hardware will take the control signals (instructions) from software (HPS) and fetch the input values from corresponding memory addresses until all the computation is completed. The output results of the neural network (results of an image recognition task) will be displayed directly by the hardware (VGA driver) on the VGA display in text format or simply represented by numbers and displayed on 7-segment LEDs.

In terms of memory, the main idea is to increase the IO bandwidth as much as possible. Usually the IO bandwidth is the ultimate limitation of inference speed, rather than the amount of hardware resource. The weights, biases, input image, temporary computation results to/from the hardware will to be stored to carefully allocated memory blocks so that the reading/writing speed will be as fast as computation. The detailed description of memory operation is included in **10 Memory Allocation and Operation.**

4. Neural Network Structure

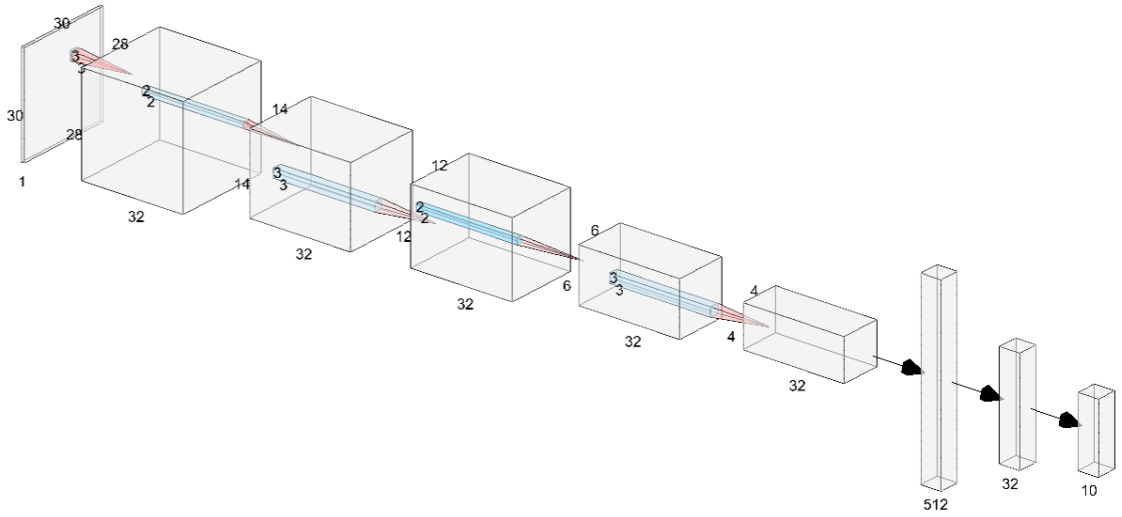

Figure 2. CNN architecture of our algorithm

```
model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(30, 30, 1)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(32, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(32, (3, 3), activation='relu'))
model.add(layers.Flatten())
model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dense(10))
```

Figure 3. Network specification in TensorFlow

A typical CNN architecture is chosen for our project as shown in Figure 2. Totally there are four possible operations in the algorithm, 2D convolution, maxpooling, image flatten, fully connected layer. All of them are standard CNN operations. Fashion MNIST dataset will be used to benchmark the performance of our system, thus the input matrix size is (30,30,1). The Fashion MNIST has 10 output possibilities thus the final output layer has 10 neurons. The network structure in the middle should be clear as shown in Figure 2 and 3. This architecture achieves ~95% of accuracy when tested in TensorFlow for the fashion MNIST dataset, which is high enough to be implemented as a course project.

(*Note: originally we were trying to accelerate an architecture called MobileNet, but we simplified our network structure to this typical CNN architecture for the feasibility concern of the project.)

5. Memory Estimation

The memory usage of our CNN architecture could be calculated as follows. This estimation is based on the design choice of quantizing all weights, biases, intermediate neuron values to 8-bit fixed number representation. All numbers are signed fixed point numbers in this project and the decimal point is always in the middle. For example for an 8-bit number, there is one sign bit, three integer bits and 4 fraction bits.

$$CNN\ Coefficients$$
$$= \big((3 \times 3 + 1) \times 32 + (3 \times 3 \times 32 + 1) \times 32 + (3 \times 3 \times 32 + 1) \times 32\big) \times 8\ bit$$
$$= 18816 \times 8bit = 18816\ B$$

Eqn. 1

$$FC\ Coefficients = (512 \times 32 + 32 + 32 \times 10 + 10) \times 8\ bit = 16746\ B \quad \text{Eqn. 2}$$

$$Max\ data\ memory\ of\ two\ adjacent\ layers = (12 \times 12 \times 32 + 4 \times 4 \times 32) \times 8\ bit = 5120\ B \quad \text{Eqn. 3}$$

$$Input\ picture = 30 \times 30 \times 8\ bit = 900\ B \quad \text{Eqn. 4}$$

Which means the memory requirement for our project is:

$$Total\ memory = 18816\ B + 16746\ B + 5120\ B + 900\ B = 41582\ B \approx 42\ kB \quad \text{Eqn. 5}$$

This rough estimation is much smaller than the total amount of block memory (BRAM) in the FPGA (500 kB), indicating that no external SDRAM is needed.

6. Software and Hardware Interface

The software and hardware interface is responsible for passing the data from HPS to FPGA hardware to process. In this project this interface is straightforward. All 32-bit Avalon bus is used to pass the image, weights and biases data before all the computation. These 32 bits are connected to two 32-bit registers in hardware, namely "data_reg" and "control_reg". Two functions called "set_data" and "set_control" are developed in software to pass data to "data_reg" and "control_reg".

The whole process will begin when "set_control" function is invoked. The current data on the Avalon bus will go to "control_reg" and configure the accelerator to prepare for the incoming data. From the next cycle on, the "set_data" function will be invoked so that the data on Avalon bus will go to "data_reg". The built-in FSM in the accelerator will automatically allocate the incoming data to an appropriate memory block with an appropriate address.

After all memory blocks are filled with correct data, the "set_control" will be invoked again to let the inference begin. Both "set_control" and "set_data" will be disabled afterwards. The recognition result will be fed back to software after inference. The recognition results will also be directly displayed through hardware, specifically displayed on 7-segment LEDs.

7. Detailed Block Diagram for Top-level Hardware

The detailed block diagram for top-level hardware is shown in **Figure 4** below. The Mem_Write module will be directly connected to the Avalon bus from HPS to receive all required data including images, weights, biases for performing inference. This module is also connected to 9 memory blocks, specifically 4 image rams (each contains one fourth information of one image), 4 dense rams (each contains one fourth of parameters for dense inference layer) and 1 convolution ram (contains all parameters for all convolutions). Mem_Write module will take the received data from HPS and allocate all image pixels and parameters to appropriate ram at appropriate addressed. A dedicated FSM will handle the storing sequence and provide controls signals to Mem_Write.

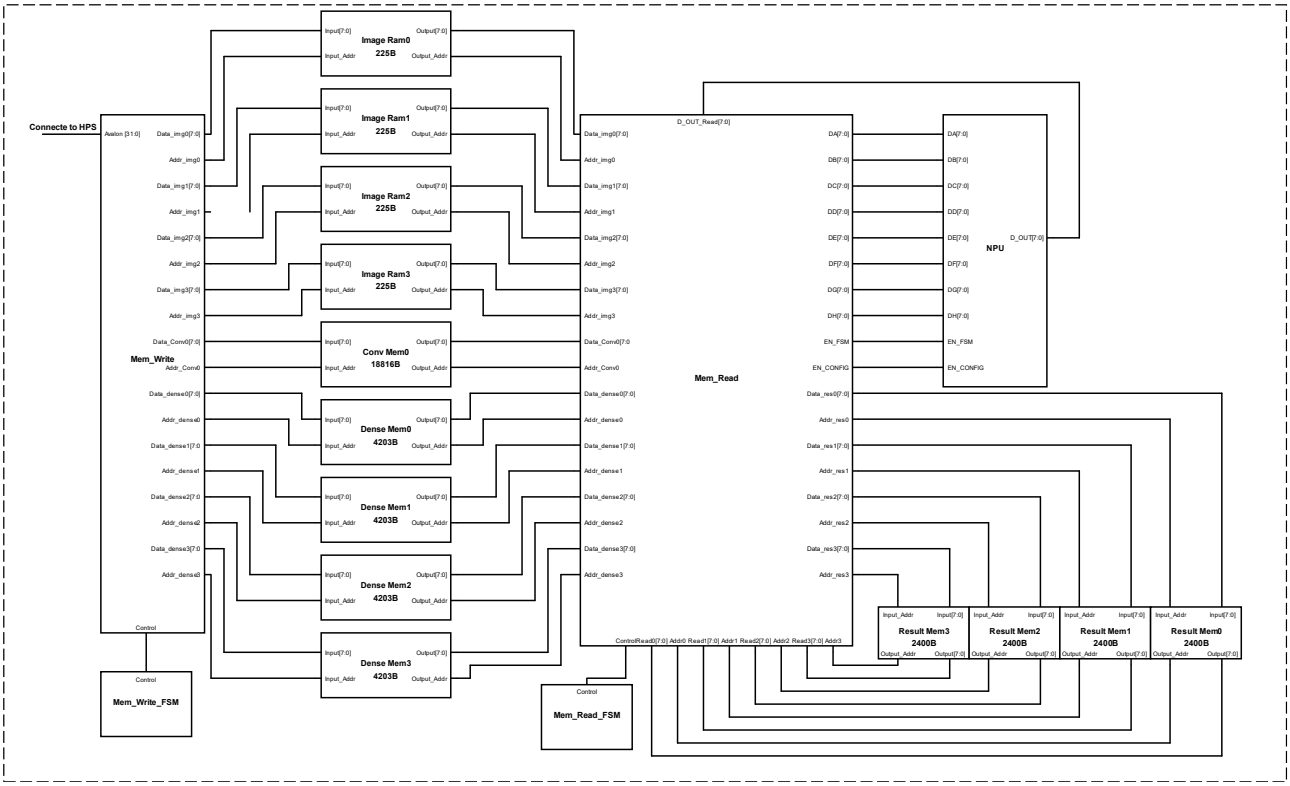

Figure 4. Top-level hardware block diagram for CNN accelerator

Once these data are properly stored in rams, the inference will begin and the Mem_Read module will begin to read data from those 9 memory blocks. Detailed writing and reading process are explained in **10 Memory Allocation and Operation** section. The data from 9 memory blocks will be sent to NPU for processing. There are four parallel MAC channels and their inputs are DA/DB, DC/DD, DE/DF, DG/DH. All outputs retrieved from NPU will be in 8-bit format and these outputs are feedbacked to Mem_Read module as intermediate results or final results. All intermediate results will be stored to 4 Result Mem blocks and these data are accessible by Mem_Read module to perform convolution/dense inference for next layer. Similar to Mem_Write, there is also a dedicated FSM to control all reading/writing sequence for Mem_Read.

8. Architecture Specification of NPU

The top-level architecture consists of three blocks, a computing core, an FSM and an SSFR as shown in **Figure 5**. The computing core is responsible for taking inputs from memory blocks and generating MAC results to the output. Relevant control signals for driving the computing core will be generated automatically from FSM. The SSFR stores some configuration options of the core and is specified in **Table 1**. The detailed implementation of computing core (NPU core) is shown in **Figure 6**. The architecture and state transition table of FSM are shown in **Figure 7**, **Table 2** and **Table 3**. The operation timing diagram is shown in **Figure 8**.

In brief, this computation core has four MAC channels that could operate independently. Each MAC module takes two 8-bit values as inputs and generates a 16-bit value as output. The output of MAC is connected to a ReLU module as hardware execute of the active function. The ReLU function could be expressed by the equation below. The user can choose to bypass the ReLU function or not for each channel by configuring SSFR.

$$ReLU\ Output = \begin{cases} \max(input, 0) & if\ ReLU\_Bypass = 0 \\ input, & if\ ReLU\_Bypass = 1 \end{cases} \quad \text{Eqn. 6}$$

There are multiple output options for the user to choose, depending on the requirement. As specified in **Figure 6**, there are 7 different output choices as the inputs to the output MUX. Option 0 is the

output from the FIFO, which will automatically record all outputs from PISO_OUT if enabled. Option 1 is the output from PISO_OUT, which will provide the original 16-bit output from all ReLU module 8-bit by 8-bit in sequence. Option 2 is the output from 16-bit to 8-bit converter, which will compress 16-bit number down to 8-bit number with automatic overflow, underflow, rounding operation. Option 3 is the index from comparator, representing the index of the largest number to the comparator so far (after last reset operation). Option 4/5 together represent the 16-bit largest number to the comparator based on the last 4 inputs. Option 6 is the 8-bit largest number to the comparator based on last 4 inputs, which is obtained using a built-in 16-bit to 8-bit converter in comparator.

The required operation timing and output timing for each option are covered in the timing diagram in **Figure 8**. The general idea is that the user should configure SSFR after each MAC operation to select the desired output format. EN_CONFIG pin should be pulled high when changing the values of SSFR, otherwise the content will remain unchanged.

The inference process mainly utilizes three options. It uses Option 6 to generate the result from one convolution + maxpooling operation for one neuron value. It uses Option 2 to generate the neuron values in dense layers as there is no maxpooling operation in dense layer. It uses Option 3 to generate the final recognition from the final output layer. No 16-bit result will be retrieved from computing core, indicating that the entire computation only takes 8-bit inputs and generate 8-bit outputs, but the computation is done in 16-bit precision.

The SSFR values for each layer are:

- conv2d1 + maxpooling 1: 1100_0001_0010_1000
- conv2d2 + maxpooling 2: 1100_0001_0010_1000
- conv2d3: 0100_0000_1011_0000
- dense1: 0100_0000_1011_0000
- dense2: 0111_1111_0010_1000

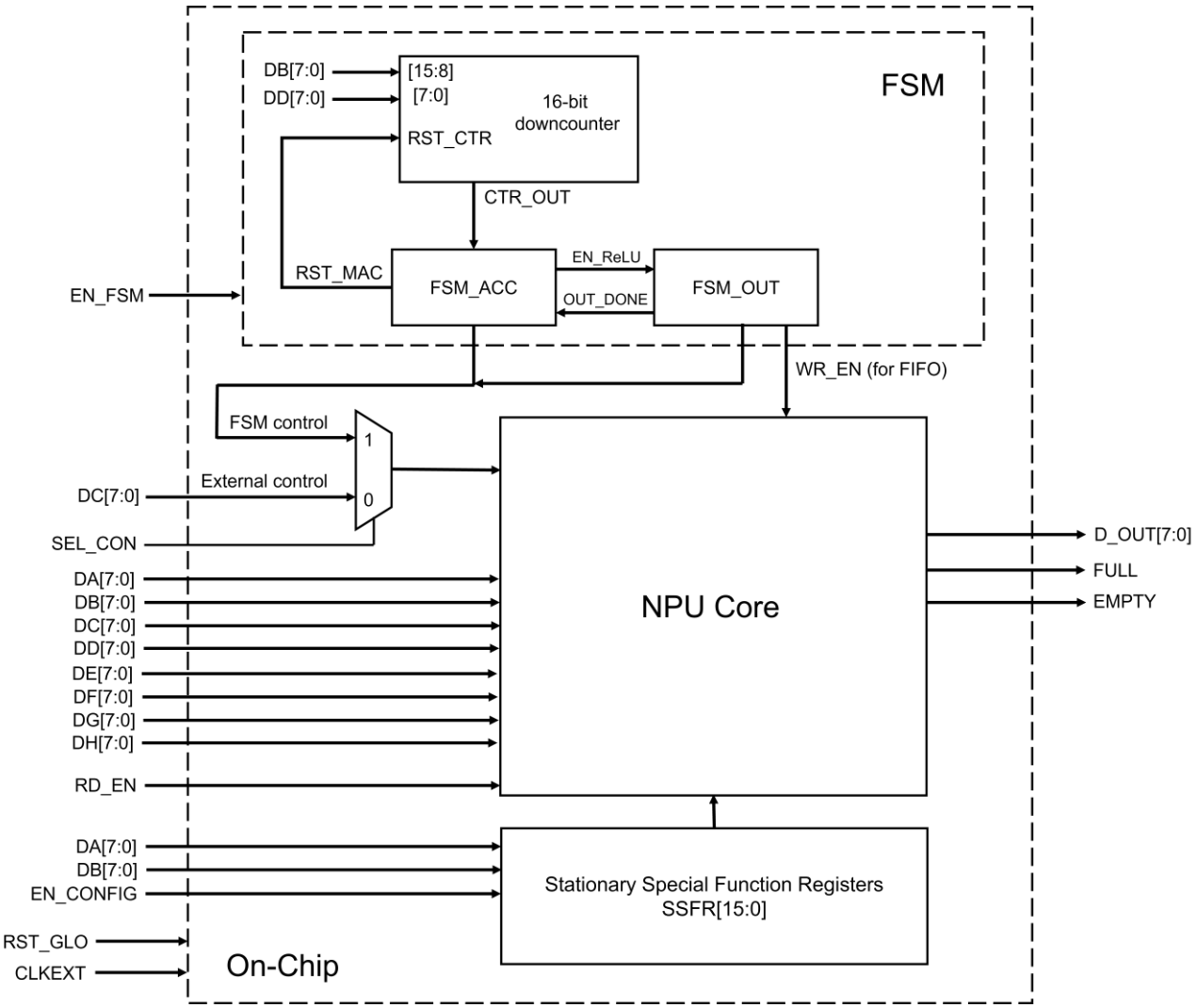

Figure 5. Top level architecture of accelerator

Table 1. Specification of each bit of SSFR

Stationary Special Function Registers (SSFR[15:0])							
SSFR[15]	SSFR[14]	SSFR[13]	SSFR[12]	SSFR[11]	SSFR[10]	SSFR[9]	SSFR[8]
SEL_OUT[2]	SEL_OUT[1]	SEL_OUT[0]	BYPASS_ReLU1	BYPASS_ReLU2	BYPASS_ReLU3	BYPASS_ReLU4	EN_COMP
SSFR[7]	SSFR[6]	SSFR[5]	SSFR[4]	SSFR[3]	SSFR[2]	SSFR[1]	SSFR[0]
RST_COMP	EN_FIFO	RST_FIFO	EN_CONV	RST_CONV	Unused		
Default Values (if reset)							
SSFR[15:8]	SSFR[7:0]						
00100000	10101000						

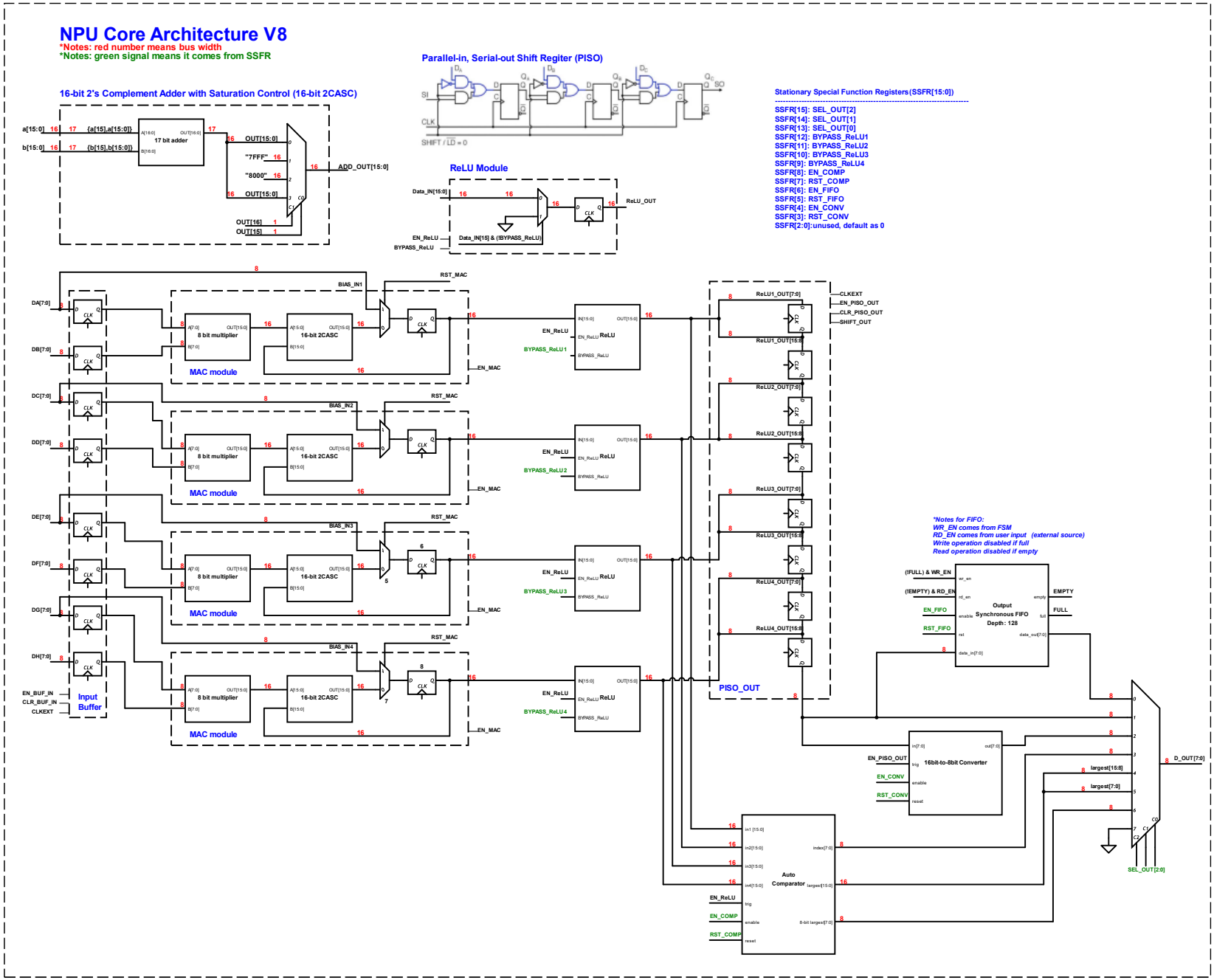

Figure 6. Detailed block diagram of the NPU core

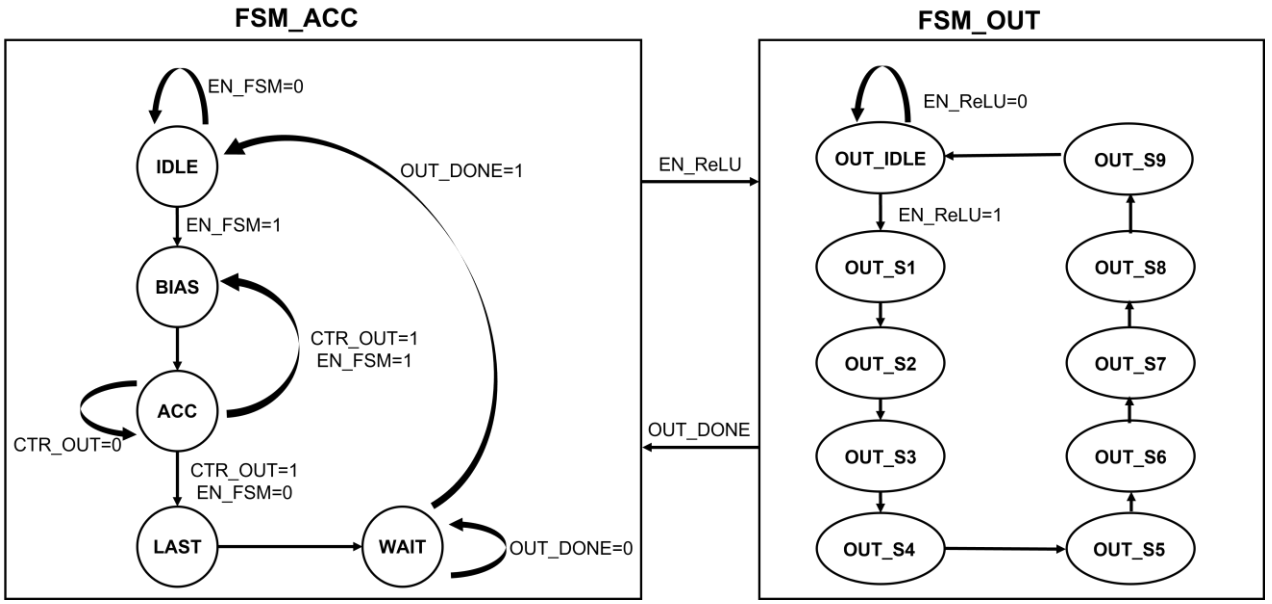

Figure 7. FSM architecture

Table 2. FSM_ACC Transition Table

FSM_ACC State Transition Table (Architecture V8)													
Current State	**Next State**				**Output**								
	EN_FSM=0 CTR_OUT=0	**EN_FSM=0 CTR_OUT=1**	**EN_FSM=1 CTR_OUT=0**	**EN_FSM=1 CTR_OUT=1**	**EN_BUF_IN**	**CLR_BUF_IN**	**EN_MAC**	**RST_MAC**	**CLR_PISO_OUT**	**EN_ReLU**		**ACC_FLAG**
										ACC_FLAG =0	**ACC_FLAG =1**	
IDLE	**IDLE**	**IDLE**	**BIAS**	**BIAS**	0	1	0	0	1	0	0	0
BIAS	**ACC**	**ACC**	**ACC**	**ACC**	0	1	1	1	0	0	1	No change
ACC	**ACC**	**LAST**	**ACC**	**BIAS**	1	0	1	0	0	0	0	1
LAST	**WAIT**	**WAIT**	**WAIT**	**WAIT**	0	0	1	0	0	1	1	No change
	OUT_DONE=0		**OUT_DONE=1**									
WAIT	**WAIT**		**IDLE**		0	0	0	0	0	0	0	No change

Table 3. FSM_OUT Transition Table

FSM_OUT State Transition Table (Architecture V8)						
Current State	**Next State**		**Output**			
	EN_ReLU=0	**EN_ReLU=1**	**Shift/~LD_OUT**	**EN_PISO_OUT**	**OUT_DONE**	**WR_EN**
OUT_IDLE	**OUT_IDLE**	**OUT_S1**	1	0	0	0
OUT_S1	**OUT_S2**		0	1	0	0
OUT_S2	**OUT_S3**		1	1	0	1
OUT_S3	**OUT_S4**		1	1	0	1
OUT_S4	**OUT_S5**		1	1	0	1
OUT_S5	**OUT_S6**		1	1	0	1
OUT_S6	**OUT_S7**		1	1	0	1
OUT_S7	**OUT_S8**		1	1	0	1
OUT_S8	**OUT_S9**		1	1	0	1
OUT_S9	**OUT_IDLE**		1	0	1	1

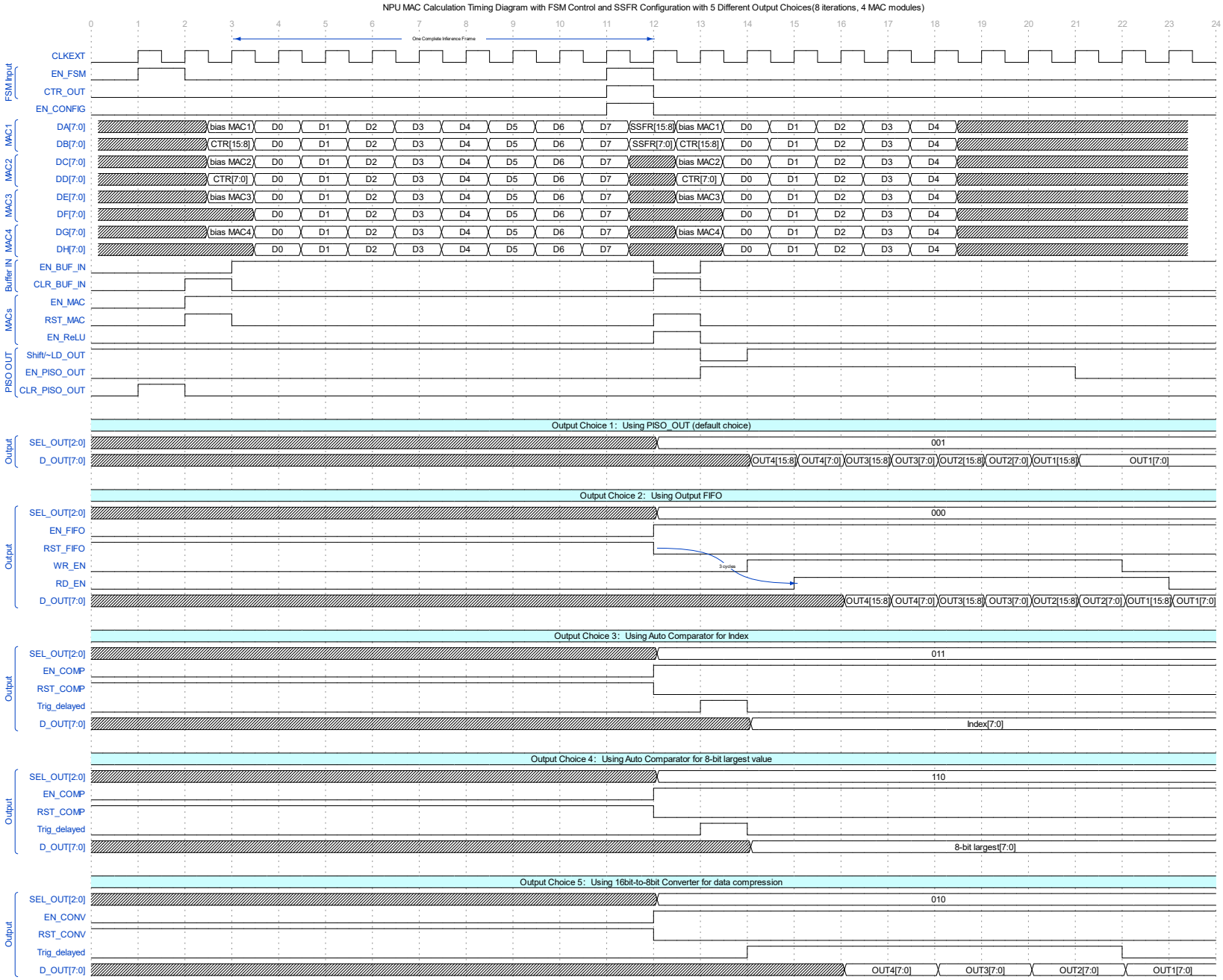

Figure 8. Computing core operation timing diagram

9. MATLAB Golden Model

A MATLAB golden model is constructed to accurately model the computation process of the accelerator. It could also verify how much loss will be brought by 8-bit quantization and whether the accelerator could still successfully classify the images, with respect to the performance of the model in TensorFlow, where all numbers are represented by 32-bit floating numbers. All image files, model weights and biases are obtained from TensorFlow and quantized in MATLAB to 8-bit. The quantized files are also exported for FPGA to use.

Four general-purpose functions called conv2d, maxpooling2by2, dense and flatten are developed to support the matrix multiplication in the inference process. MATLAB has built-in support for fixed-point number representation and operation precision, but no support for fixed point convolution, thus these customized layer operation functions are developed from scratch. With these functions, the inference process of the model could be represented by **Figure 9**.

```
%perform the convolution and max pooling process
conv2d1_result = conv2d(img_f8,  conv2d1_weight_2D_f8, conv2d1_bias_f8, F8, wordlength8, fractionlength8,  1);      %f
pooling1_result = maxpooling2by2(conv2d1_result, F8, wordlength8, fractionlength8);                              %fir

conv2d2_result = conv2d(pooling1_result, conv2d2_weight_2D_f8, conv2d2_bias_f8, F8, wordlength8, fractionlength8, 1);
pooling2_result = maxpooling2by2(conv2d2_result, F8, wordlength8, fractionlength8);                               %s

conv2d3_result = conv2d(pooling2_result, conv2d3_weight_2D_f8, conv2d3_bias_f8, F8, wordlength8, fractionlength8, 1);

flatten_result = flatten(conv2d3_result, F8, wordlength8, fractionlength8);

dense1_result = dense(flatten_result, dense1_weight_f8, dense1_bias_f8, F8, wordlength8, fractionlength8, 1);

dense2_result = dense(dense1_result, dense2_weight_f8, dense2_bias_f8, F8, wordlength8, fractionlength8, 0);
```

Figure 9. MATLAB representation of all layers in the model with custom functions

The intermediate results such as “conv2d1_result” and “pooling1_result” are used to provide golden reference to the accelerator operation. These results are also compared to the results from TensorFlow to verify how quantization is affecting each layer and how error is accumulating layer by layer. 8-bit quantization results of each layer are compared to 32-bit quantization results in **Figure 10**. The average RMS error here represents the error between a single neuron derived in MATLAB and the corresponding neuron in TensorFlow quantized using the same precision. It is clear that lower precision will cause error regarding ideal results, and the error will accumulate layer by layer. If the quantization goes to 32-bit, i.e., a very high precision, the error will vanish.

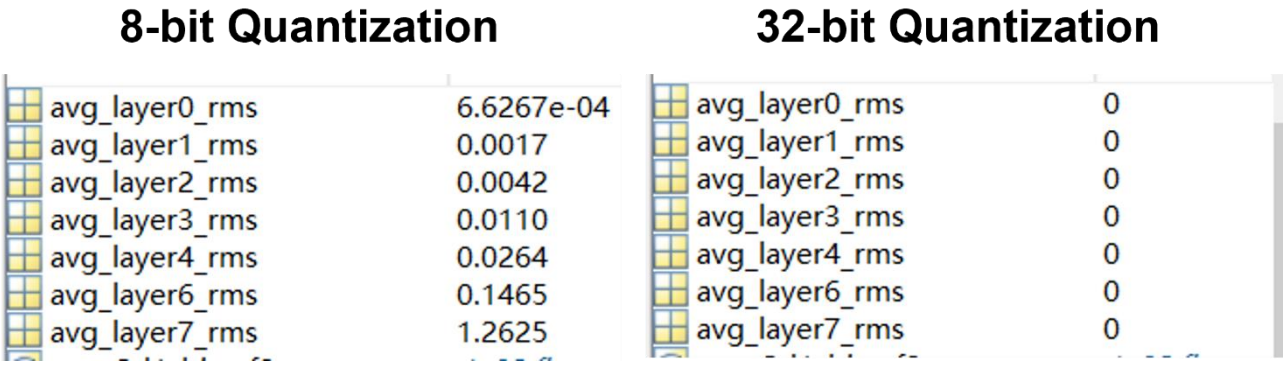

Figure 10. Quantization error comparison

Using 8-bit quantization for weights and biases, and 16-bit computation precision, the recognition results from the golden model could be represented by the confusion matrix below:

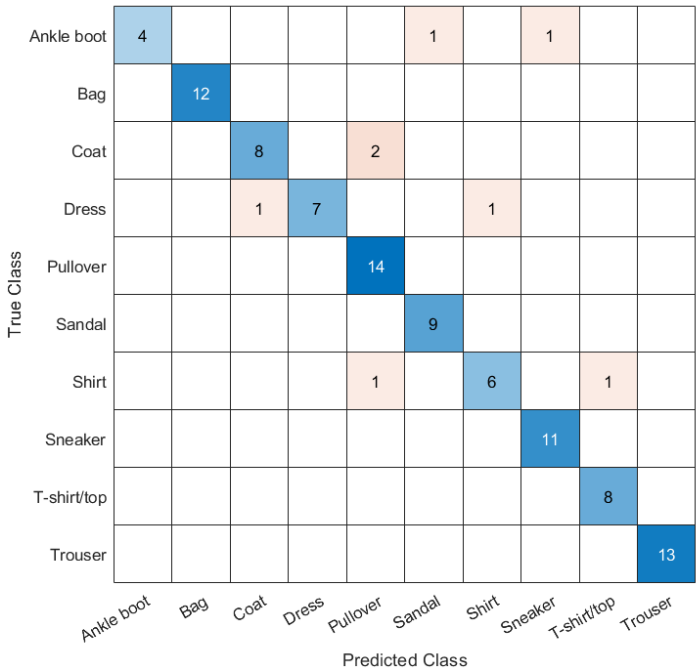

Figure 11. Confusion matrix of recognition results from MATLAB golden model (100 images)

According to the confusion matrix, 8 images are misclassified and the rest 92 images are correct, indicating that in this test the model accuracy is 92%. This result could prove that the quantization precision chosen for the accelerator does not degrade accuracy very much, as the original accuracy in TensorFlow is approximately 95%.

10. Memory Allocation and Operation

To realize four channel MAC parallelism, the memory allocation and reading operation are specially customized to reach high efficiency and satisfy the timing requirement of NPU. This section is divided into convolution layer/maxpooling layer, flattening layer and dense layer.

10.1. Convolution layer/Maxpooling layer

These two layers are grouped together because they are always adjacent layers in our CNN and the NPU is designed to complete two layers together in an efficient manner. Assuming the input to a convolution layer is a 2D image shown in **Figure 12**. The current convolution kernel is 3 by 3 and the maxpooling window is 2 by 2, indicating that the rectangle in each color in Figure 12 represents a 3 by 3 convolution region and the convolution results from all four of them will be the inputs to a 2 by 2 maxpooling region.

D11	D12	D13	D14	D15	D16	D17	D18	D19	D110
D21	D22	D23	D24	D25	D26	D27	D28	D29	D210
D31	D32	D33	D34	D35	D36	D37	D38	D39	D310
D41	D42	D43	D44	D45	D46	D47	D48	D49	D410
D51									
D61									
D71									
D81									
D91									
D101									

Figure 12. Example of a 2D image (10 by 10, single channel)

To complete this convolution and maxpooling operation together, data should be fed into NPU in the following sequence (4 data at a time for four channels): D11/D12/D21/D22, D12/D13/D22/D23, D13/D14/D23D24…..D33/D34/D43/D44. Totally 9 times. Every 4 by 4 region will follow this sequence. Since the image will be flattened when stored in ram and the address for each data will be very irregular based on this sequence, the following approach is adopted:

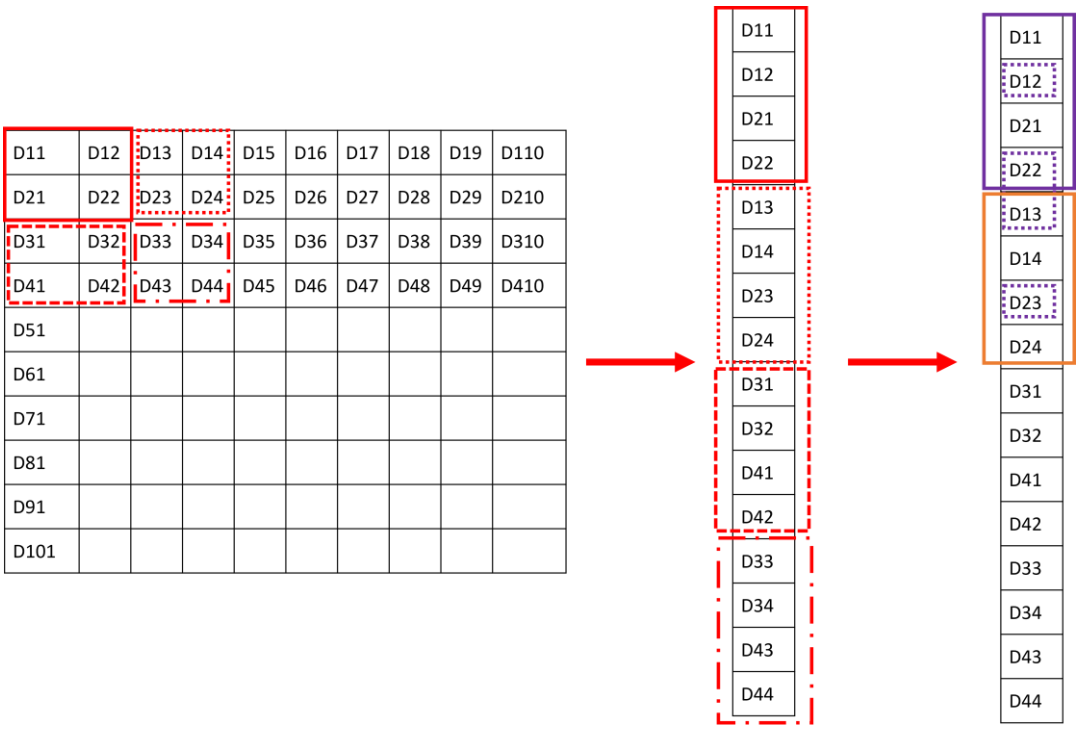

Figure 13. Temporary buffer approach for generated the required sequence

Each 4 by 4 region could be divided into 4 2 by 2 regions. The data at each corner of 2 by 2 region will be stored into one ram, thus a total of four rams (so we have 4 image rams). For example in this 4 by 4 region, D11/D13/D31/D33 will be stored into one ram in sequence, D12/D14/D32/D34 will be stored into another ram in sequence, and so on. When reading these four rams, four data could be obtained at one time and they are flattened and stored into one array, as shown in the middle array in **Figure 9**. When the original 4 by 4 region is transformed into this flatten array, the reading patten will be the same for all 4 by 4 regions. It will always be the purple rectangle, the purple dot rectangle, the orange rectangle and so on.

Each of these 4 by 4 regions will generate one data after convolution and maxpooling. The following figure could explain how four 4 by 4 regions are transformed into 4 results, and the relative positions of these four results in the new array.

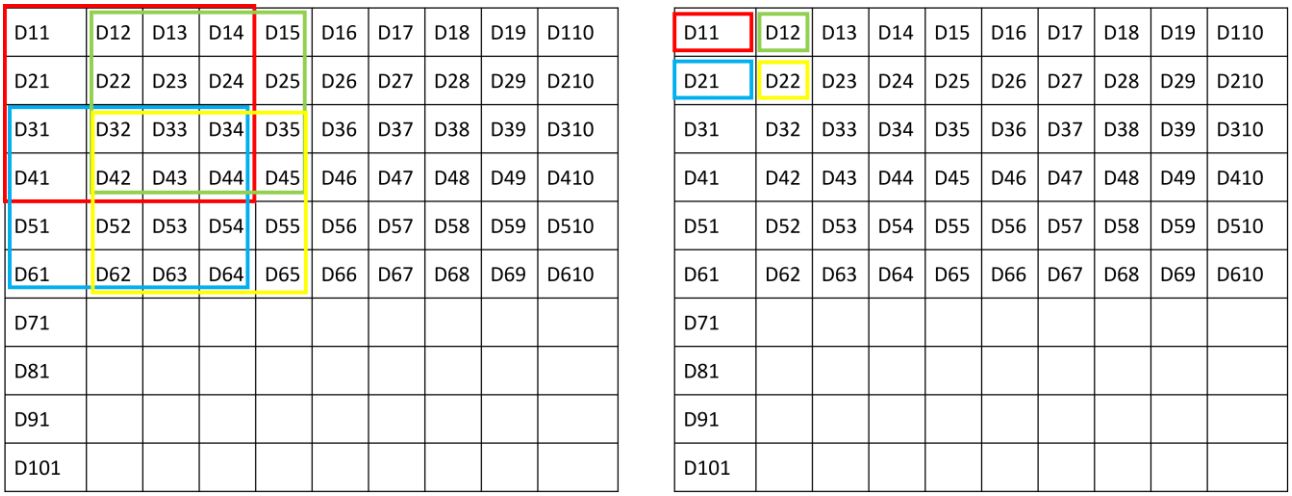

Figure 14. The transformation of four 4 by 4 regions after convolution and maxpooling

By using this methodology, all convolution and maxpooling layers could be finished and the timing could satisfy the requirement of NPU.

10.2. Flattening Layer and Dense Layer

The logic for flattening operation is straightforward. The result from the final convolution layer is a 4 by 4 by 32 matrix. As there is only 1D convolution in dense layer, it is intuitive to flatten the result regarding the sequence of width, height, depth. To still achieve highest parallelism in dense layer, the parameters (weights and biases) for each dense layer need to be stored in four different rams, so we have four dense rams as shown in **Figure 5** . Since four different sets of parameters are convolving with respect to the same result from final convolution layer (4 by 4 by 32, thus 512 data), only one Result_Mem is required to store all 512 data. 32 neuron values will be derived from the first dense layer. These 32 values, again, will be stored only in one of the Result_Mem for the derivation of final 10 output neurons. There is no flattening required for the second dense layer and the 1D convolution logic is identical to the first dense layer. It should be noted that the NPU computing core only supports 4-channel parallelism, but there are 10 output neurons in the final output layer. Two dummy output neurons are added so that in total there are 12 output neurons, thus can be calculated as 4+4+4. These dummy neurons are set to smallest value (10000000) hence will not affect the recognition result.

11. Accelerator Evaluation and Results

In the evaluation section, an image of “ankle boot” from Fashion MNIST dataset is used for illustration purposes. The image is shown in **Figure 15** and has the dimension of 30 by 30 pixels. Originally all pixels are in the range of 0 to 255. They are normalized to the range of 0 to 2 in **Figure 15** to avoid saturation issue during computation, since the 8-bit binary representation in accelerator can support -8 to 7.9375.

The representation of 10 classes follows the sequence of “'T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot'”, thus the final recognition result should be 10, i.e., the last class.

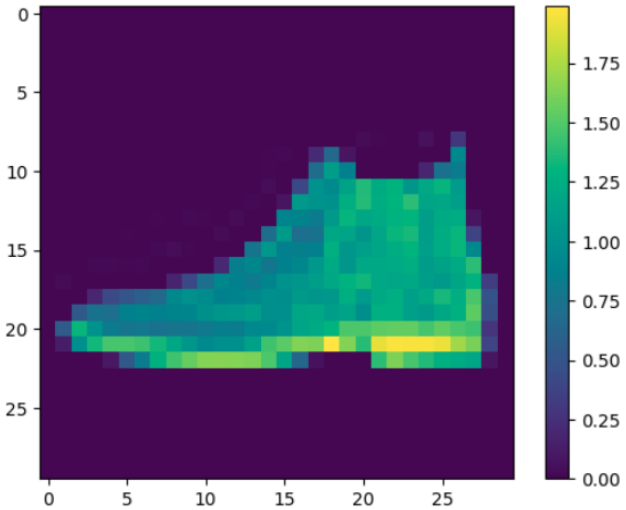

Figure 15. An example of ankle boot image for evaluation

11.1. Data transfer between software and hardware memory

The waveform of data transfer is shown in **Figure 16** and **Figure 17**. As mentioned in the memory design section, there are totally 4 image rams + 4 dense rams + 4 result rams + 1 conv ram. The **Figure 16** showed the process of transferring the image data. It could be seen that the current state is

"WRITE_FOUR", indicating the image data are being copied to image rams. The writing ports of image rams (data_image0/ data_image1/ data_image2/ data_image3) keep changing as expected. The **Figure 17** showed the process of transferring weights and biases to parameter rams (dense rams and conv ram). During the state of "WRITE_SEQ_CONV", the weights and biases of all convolution layers are being imported to the conv_ram. It could be seen in **Figure 17** that the "conv_ram_addr_a" and "data_conv" keep changing during this state. In the next state of "WRITE_FOUR_DENSE", the weights and biases of all dense layers are being inputted into the dense_rams. Similarly the "dense_ram_addr_a" and "data_dense0/1/2/3" keep changing during this state.

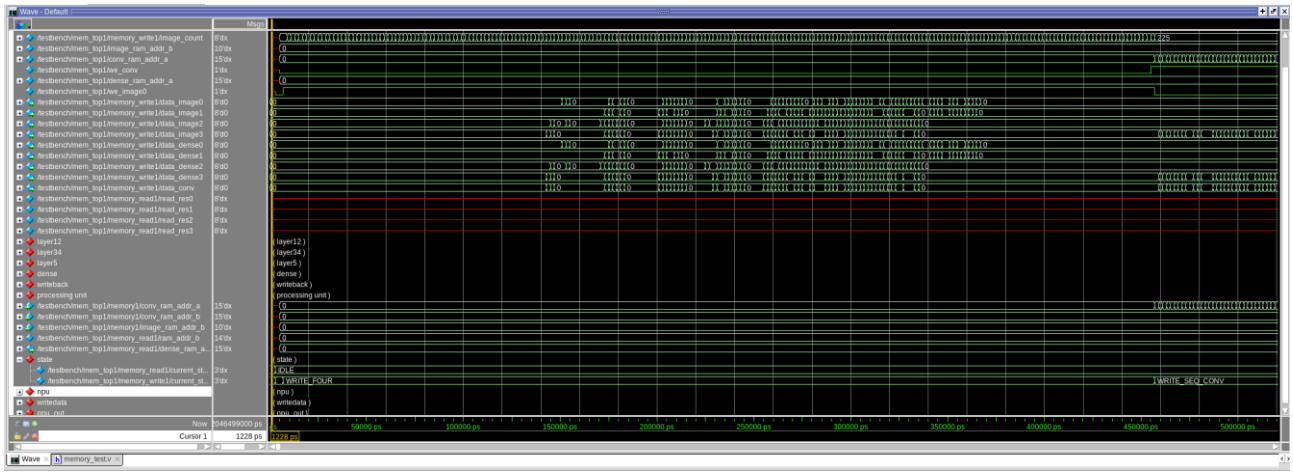

Figure 16. Data transfer of image data

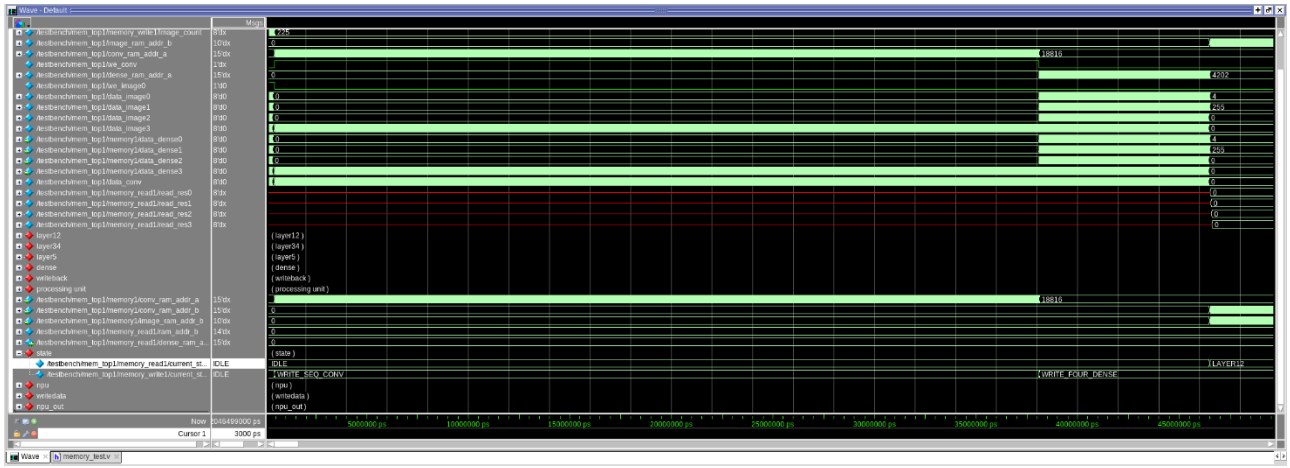

Figure 17. Data transfer of convolution/dense layer parameters

11.2. Results of conv2d1 and maxpooling1

The waveform of conv2d1 and maxpooling1 is shown in **Figure 18**. The data changing density is very high in the figure so not everything is visible. It could be seen from "channel32" that the accelerator is convolving from channel 1 to channel 32. The "current_state" is "LAYER12" for this section. It can also be observed that the data from conv_ram and image_rams are being fed into the NPU for computation and "D_OUT" port of NPU is generating outputs. These outputs will be stored to result_rams as temporary results.

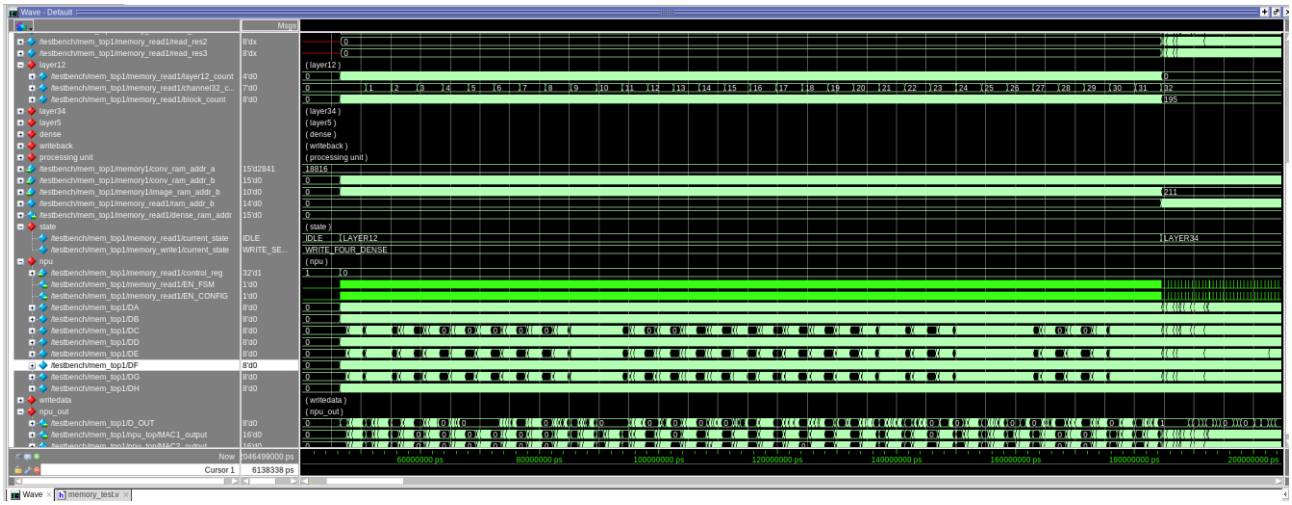

Figure 18. Results of first convolution and first maxpooling layer

11.3. Results of conv2d2 and maxpooling2

The waveform of conv2d2 and maxpooling2 is shown in **Figure 19**. This convolution process is very similar to the results in **Figure 18**. The convolution weights and biases, together with the results from previous layer, are fed to NPU for computation and the results from "D_OUT" will be stored to result_rams.

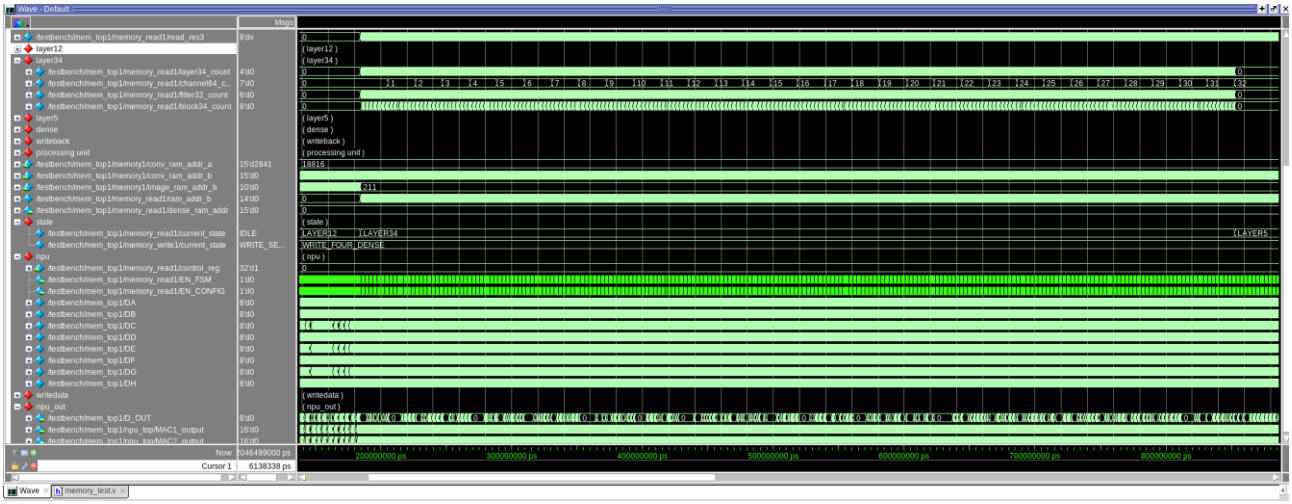

Figure 19. Results of second convolution and second maxpooling layer

11.4. Results of conv2d3

The waveform of conv2d3 is shown in **Figure 20**. There is no maxpooling layer after this convolution so the raw data after convolution are directly stored into result_rams through "16-bit to 8-bit converter" inside NPU. Similarly, there are still 32 channels (32 filters) in this layer so the channel index keeps increasing. The "current_state" is LAYER5 for this layer. Since the results of each layer will be generated sequentially from NPU, the outputs are already flattened and only need to be stored to appropriate rams with appropriate addresses.

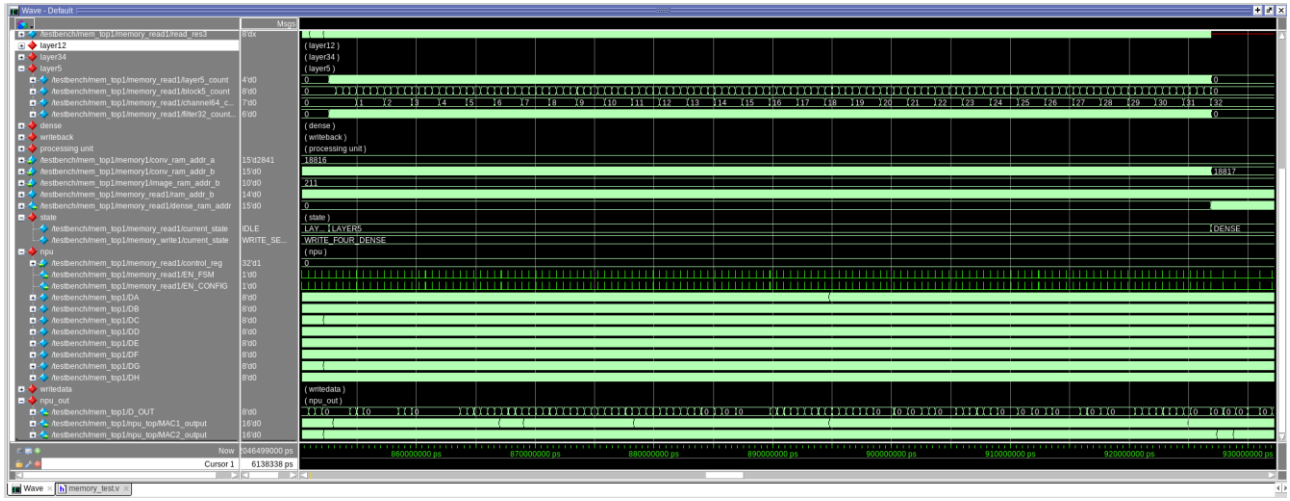

Figure 20. Results of third convolution layer

11.5. Results of dense1

The waveform of dense1 is shown in **Figure 21**. The inference process of each dense layer is just many 1D convolution, which is much more straightforward than convolution layers. The 1D convolution is performed by fetching data from result_rams (results from previous layer) and dense_rams (weights and biases). There are four MAC channels in NPU thus four neurons could be derived together. It could be seen from **Figure 21** that the "dense_bias_count" changes from 0, to 4, 8, ……until 32, indicating four biases for four channels are used in each inference cycle. The current state of this layer is called "DENSE". It could be seen that the data switching density in the screenshot region, such as the switching activity of "EN_FSM" is much lower now, compared to convolution layers. This is because there will only be 32 output neurons for this layer.

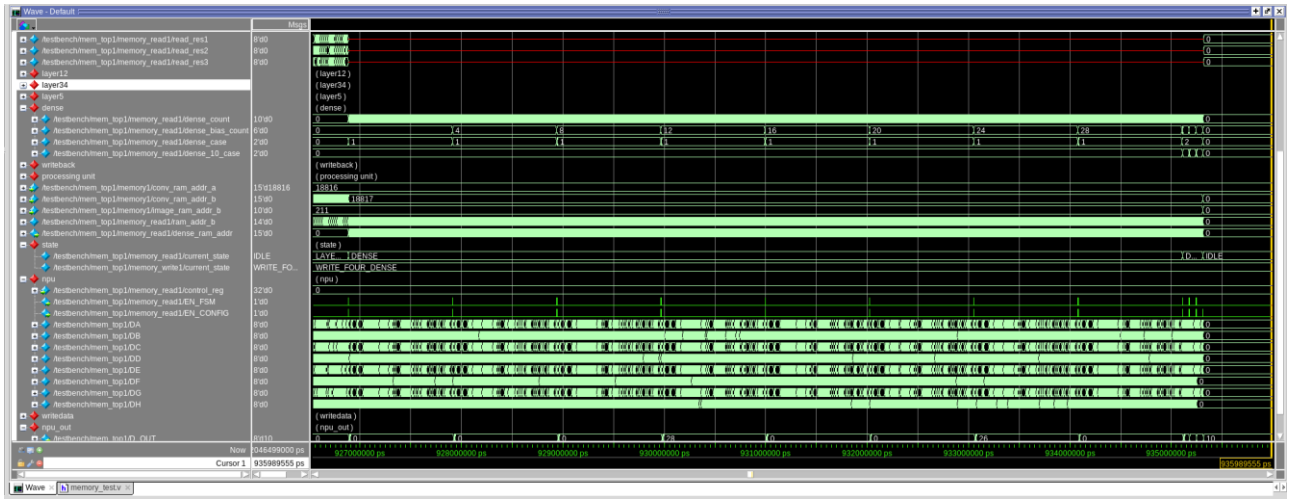

Figure 21. Results of first dense layer

11.6. Results of dense2 and final recognition

The waveform of dense2 and final recognition are shown in **Figure 22**. Now the waveform density is even lower and everything is visible. The 1D convolution of dense2 layer is same as dense1 layer, though the output option is different. The output of dense2 layer is obtained from the "index" port of automatic comparator in NPU, while the output of dense1 layer is obtained from "16-bit to 8-bit converter" in NPU. The automatic comparator can directly compare the values of 10 output neurons and choose the index of the largest neuron, i.e., the recognition result. As mentioned before, this

example is an Ankle boot and the recognition result should be 10. It could be seen from **Figure 22** that the final value of "D_OUT" is 10, which represents the correct index.

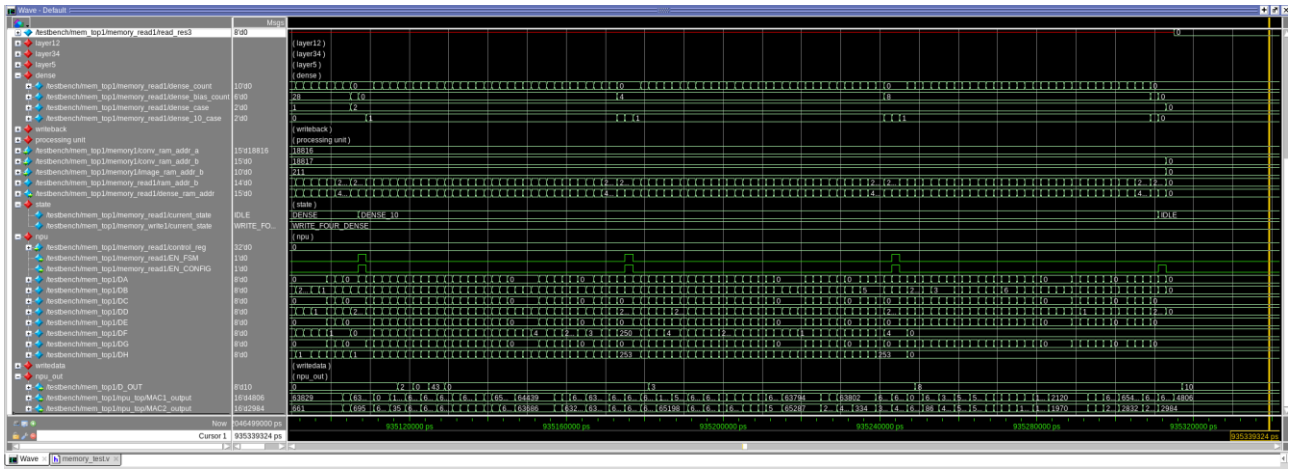

Figure 22. Results of the second dense layer

12. Member Contribution and Advice for Future Projects

12.1. Tianchen Yu

I am responsible for the RTL development of NPU (computing part) and construction of MATLAB golden model. I am also responsible for all documentation of NPU and writing the entire report. I used the MATLAB model to verify the performance of my NPU and accurately model every single output from NPU. The major difficulty for MATLAB model development is to fully understand the exported parameters format from TensorFlow. Because MATLAB does not have built-in support for fixed-point 2D convolution, every layer operation must be developed from scratch. Although MATLAB provides very good support for matrix operation, careful attention needs to be paid for any data manipulation, such as the multiplying priority for width, height, depth, filter index etc. The raw data from TensorFlow may follow a very efficient pattern, but at least it is not intuitive. A few attempts are needed to fully figure out the data format from TensorFlow. One suggestion is that the future group could use a very accurate quantization first (such as 32-bit) to figure out the data format based on an error estimation (such as the RMS method I used). If everything is properly aligned the error will be very close to zero, or even just zero. Then the chosen quantization level could be applied to obtain realistic performance.

The trickiest part of NPU design is the alignment of all timing and the collaboration of all modules. The nature of hardware accelerator requires a very efficient design of timing to ensure most cycles are dedicated for computing, rather than memory operation or even waiting. This means every module needs careful alignment so that the overall throughput can be maximized. Although the clock frequency is limited to 50 MHz in this platform, the entire NPU follows a pipelined design that could be potentially deployed at a much higher frequency for a higher throughput. Another challenging part of NPU design is the binary number operation, particularly during 8-bit to 16-bit transition or backward case, or binary number multiplication. In our design we replicate the addition, multiplication, underflow/overflow control, rounding operation logic from MATLAB (one of typical operation logics from MATLAB) so that everything could be precisely modelled. Not all of these operations are easily achievable in Verilog design. Operations such as multiplication and data compression require lots of tuning to get fully correct results.

12.2. Haichun Zhao

I was participating in the design and implementation of memory management. In this project, we had designed 13 memory modules with various sizes and functionality. It was crucial to have memory to

execute in our designed sequence in both storing: store image data, store bias + parameters and executing: read image data and corresponding parameters, write back, read other data and parameters, etc. We developed a unique reading/writing mechanism that fit our purpose and current architecture. I wrote the state machine for memory management logic to ensure this process is done properly without error in both stages. Both memory_write and memory_read files had undergone several revisions in order to achieve the correct results.

If we had more time, this accelerator could be more flexible, for example taking different sizes of pictures and permanently storing the parameters in the memories so we do not have to feed the parameters first for every picture. We could also reduce the memory size required since some memories kept slightly larger size for consistency and ease of development.

12.3. Qixiao Zhang

My job in this project is to design the dataflow of different memories and data feeding from block RAM to the NPU computation part. Because the NPU cannot store any information and it needs data consistency to run effectively, we must design a specific way to transfer data from HPS to the FPGA and then do the computation. Therefore, it is necessary to use the on-chip M10K block RAM and we need to code it by ourselves. We make data saved in sequence to make life easier so the data from Avalon Bus can be stored one by one. However, after doing this we also need to collect the output from each layer so we also designed an inner memory in Inference part to store the temporary data for next layer. These are very hard and took us a lot of time to finish them.

I also took care of the debugging and designing the hierarchy of these memories. This is almost the hardest part of this project in that we need to ensure that all the signals are in the right timeline.

The biggest limitation of this project is that this accelerator is way too specific and cannot take care of other CNN structures. I think if we have more time we can design a more general one and can make the most of the HW/SW interface.

12.4. Haomiao Li

I was participating in the implementation and testing of memory arrangement. The memory management sections connect the software interface and NPU hardware. In this project, we designed 13 memory modules with various sizes and functionalities. Therefore, it was important to arrange the memory execution following our designed sequence in both storing: storing image data, storing bias + parameters and executing: reading image data and corresponding parameters, write-back, reading other data and parameters, etc. We developed a unique reading/writing mechanism that fits our purpose and current architecture. I helped to debug and write the testbench for this structure.

12.5. Yue Niu

I am mainly responsible for the software part. I trained a simple convolution neural network on the dataset fashion-MNIST. It consists of three convolutional layers and two dense layers. I changed the network several times to compromise the memory size and difficulty of the hardware part. I also ran the demo in TensorFlow light and quantized it to 8-bit to ensure it still has acceptable accuracy. I also exported the model weights and test images and used the same interface in lab 3 to send these data to the registers for hardware to use. In future work, it’s important to find a way to store weights in hardware to save transmit time.

13. Complete Listing of Project Files

Table 4. Complete Listing of Files

Module Name	File Type	File Name
NPU	FSM Verilog	FSM.v
		FSM_ACC.v
		FSM_OUT.v
	Top Cell Verilog	npu_top.v
	Block Verilog	auto_comparator.v
		data_converter.v
		input_buffer.v
		MAC.v
		npu_v8.v
		piso_out.v
		ReLU.v
		syn_fifo.v
MATLAB Golden Block	Golden model	golden_model.m
	Custom layer function	conv2d.m
		dense.m
		flatten.m
		maxpooling2by2.m
TensorFlow model	Jupyter notebook	Fashion MNIST.ipynb
Top hardware module	System Verilog	vga_ball.sv
Memory	System Verilog	mem_top.sv
		memory.sv
		memory_read_sim.sv
		memory_write.sv
	SV Testbench	memory_test.sv
Software	C files	hello.c
		read_weights.c
		vga_ball.c
		read.c