TECHNISCHE UNIVERSITÄT DRESDEN

Department of Electrical Engineering and Information Technology

Chair for Chair for Highly-Parallel VLSI Systems and Neuromorphic Circuits

Student research thesis

with Topic

Implementation of a Convolutional Neural Network on an FPGA

Author	Binyi Wu
Course of Study	Informationstechnik, Jg. 2015
Matrikel-Nr.	4571474
Birthday	12.05.1991 in Guangdong China

Supervisor:	Dr.-Ing. Johannes Partzsch
	Dipl.-Ing. Florian Kelber
Professor:	Prof. Dr.-Ing. habil. Christian Mayr
Day Of Submission:	October 1, 2018

TECHNISCHE UNIVERSITÄT DRESDEN

Department of Electrical Engineering and Information Technology

Chair for Chair for Highly-Parallel VLSI Systems and Neuromorphic Circuits

Abstract

Convolutional neural networks currently have achieved near or even beyond-human performance in visual perception but at the cost of high computational complexity. This work designs a dedicated accelerator to speed up the inference phase of quantized models with minimum resource usage and maximum energy efficiency, making running neural networks in mobile devices possible. To validate the design, FPGA board and the open source library TensorFlow are employed.

Keywords: Convolutional neural networks, accelerator, inference, quantized models, energy efficiency, mobile devices, FPGA, TensorFlow

Tutor:	Dr.-Ing. Johannes Partzsch
	Dipl.-Ing. Florian Kelber
Supervisor:	Prof. Dr.-Ing. habil. Christian Mayr
Day of Submission:	October 1, 2018

STUDENT RESEARCH THESIS Author: Binyi Wu

Statement of Authorship

I hereby certify that I have authored this Student Research entitled *Implementation of a Convolutional Neural Network on an FPGA* independently and without undue assistance from third parties. No other than the resources and references indicated in this thesis have been used. I have marked both literal and accordingly adopted quotations as such. They were no additional persons involved in the spiritual preparation of the present thesis. I am aware that violations of this declaration may lead to subsequent withdrawal of the degree.

Dresden, October 1, 2018 .....

Binyi Wu

Acknowledgements

I thank my supervisors Dr.-Ing. Johannes Partzsch and Dipl.-Ing. Florian Kelber for their constant help and patient guidance. For the first time I entered the field, they gave me a lot of useful advice, led me to step by step familiar with the convolutional neural network, using TensorFlow training neural network, and then design your own convolutional neural network accelerator. In this project, I learned a lot of new knowledge and once again thanked my supervisors for encouraging and accepting me to use my ideas for the project. Finally, I would like to thank all the scholars who have made valuable research in related fields. Special thanks to the author of *DianNao*[1] and *Cnvlutin*[2], for giving me the idea of designing CNNA.

Contents

List of Figures

List of Tables

List of Abbreviations

AAA	Accumulate Adder Array (Unit)
CNNA	Convolutional Neural Network Accelerator
CNN	Convolutional Neural Network
CONV	convolutional layer
CPU	Central Processing Unit
CU	Computation Unit
DMA	Direct Memory Access
FC	fully-connected layer
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
LOSS	loss layer
MAC	Multiply-Accumulate/Multiplier–Accumulator
MAX	maximum
ML	machine learning
MUX	multiplexer
NA	neuron activation
POOL	pooling layer
RNN	Recurrent Neural Network
ReLU	Rectified Linear Units layer
SIMD	Single Instruction Multiple Data

SRAM	Static Random Access Memory
SVM	Support Vector Machine
TF	TensorFlow
UDP	User Datagram Protocol
VP	Validation Program

1 Introduction

Convolutional Neural Networks (CNNs) is a state-of-the-art classification, recognition machine learning (ML) algorithm, achieving near or even beyond-human performance in visual perception but at the cost of high computational complexity[3][4]. In spite of CNN's characteristics of sparse connectivity and weight sharing, CNN needs huge numbers of multiply–accumulate (MAC) operations in high-dimensional convolutions, which can be realize through highly-parallel compute paradigms, such as SIMD (Single Instruction Multiple Data). This is also what are done in graphics processing unit (GPU), which are widely used to accelerate ML algorithms. However, it involves a significant amount of data movement, requiring high energy consumption.

In this document, we present a novel CNN accelerator, which aims to be employed in mobile devices due to optimized energy consumption. In order to design a high energy efficiency CNN accelerator, two caches for filter weight and neuron activation, which can severely decrease memory fetch operations, are added into the accelerator. However, during CNN computation, the power consumption comes not only from memory operation but also from the MAC unit. As mentioned above, CNN computation requires lots of MAC operation. So, another effective method to reduce energy is through lower precision computation. In our design, 8-bit-integer multiplication, X-bit-integer addition are used instead of 32-bit-float-point multiplication and 32-bit-floating-point addition respectively.

In order to verify the design, a quantized CNN model trained through TensorFlow is employed. TensorFlow, an open-source software library for Machine Intelligence from Google, is the most widely used library for deep learning[5].

The document is organized as follows. Chapter 2 provides an overview on deep learning, a description of convolutional neural network and the opensource machine learning library TensorFlow. Chapter 3 fully describes the architecture and data flow of Convolutional Neural Network Accelerator (CNNA). Chapter 4 illustrates the validation and experimental methodology for CNNA, including how to map convolutional neural layer and fully-connected layer into CNNA. The last chapter summarizes this paper.

2 Deep Learning and Convolutional Neural Networks

Deep learning, also referred to as deep neural networks, are part of the broad field of machine learning (ML). Machine learning is the field of study that gives computers the ability to learn without being explicitly programmed. By the way, neural networks take their inspiration from the neuronscience that a neuron's computation involves a weighted sum of the input values[6][7][8]. The neural networks has two major forms: feed forward and recurrent. The convolutional neural networks that this paper focused on belongs to the forward network.

Here, we provide a description of the relevant elements of deep learning and convolutional network.

2.1 Deep Learning and Neural Networks

Under normal circumstances, deep learning refers to multi-layer neural network, generally not less than 3 layers. So, in the following, deep learning, deep learning neural network and deep neural network are referring to the same thing.

This paper mainly fouces on feedforward networks. As shown in Figure 2.1, there are one input layer, at least one hidden layer and one output layer in a feedforward deep learning neural network. Each layer can have different number of neurons. Usually, the deeper the neural network, the stronger its learning ability or generalization ability. However, this ability results in massive computation during training phase. This is the reason why neural networks were not popular years ago.

There are many variations of feedforward neural networks, including fully connected neural networks, convolutional neural networks (CNN) and etc. In this paper we mainly discuss and analyze convolutional neural networks. The accelerator, we designed, is also mainly used to speed up the computation of CNNs[9].

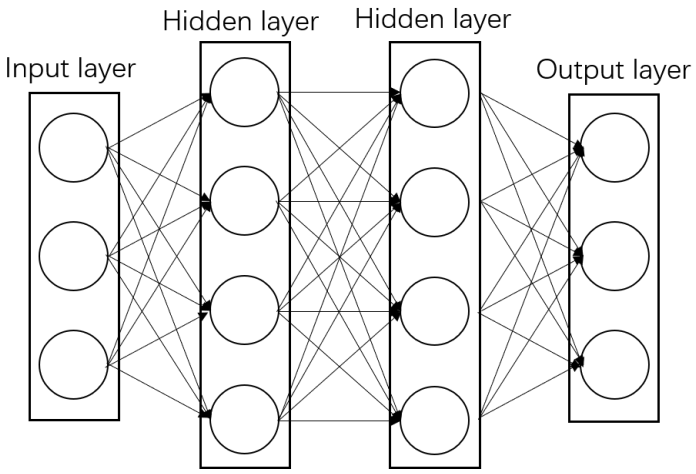

Figure 2.1: Example of a deep neural network

2.2 Training and Inference

There are two phases in designing deep neural networks, training phase and inference phase. Inference phase is after training phase, which uses the backpropagation algorithm[10].

Currently, we have only the inference part (forward-propagation) rather than training part (backward-propagation) implemented in the hardware accelerator (CNNA). The training involves floating-point arithmetic operation, which not only consumes much more energy comparing to integer arithmetic operation in circuit, but also takes more area. On the other hand, training phase is essential for developers and researchers rather than common customer, whose population is much larger than the former. Therefore, our accelerator only speed up the inference. Moreover, so far, most state-of-art neural network model can only be trained by large corporations, like google and etc. So, developing an inference CNNs accelerator for common customers are much more productive[9].

2.3 General Structure of Convolutional Neural Networks

Convolutional neural network are widely used in image and video recognition. A typical convolutional neural network (CNN) has multi-layer, like deep neural netowrk. The input layer is normally the raw pixel values of the image. The hidden layers contain convolutional layer (CONV), non-linearity layer

(NONL), pooling layer (POOL) and fully connected layer (FC). The output layer is always the loss layer (LOSS)[9].

CNN arranges its neurons in each layer in three dimensions (width, height, depth/channel). The dimension "depth" is also named "channel" in the following. Every layer of a CNN transforms the 3D input volume to a 3D output volume of neuron activations. Figure 2.2 is an example of CNN. The grey parts can be accelerated by CNNA.

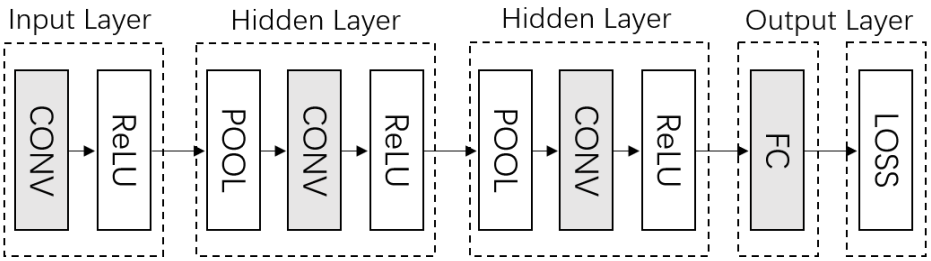

Figure 2.2: Example of convolutional neural network

2.3.1 Convolutional Layer

The convolutional layer is the core part of a CNN. It is the heaviest computation part of CNNs.

A convolutional layer has two multi-dimension operands (or inputs). One is called feature map, which can be an image (for input layer) or an output of the previous layer, and the other operand is called filter, which is learnable and is used to extract deeper and more abstract features from feature map. The feature map is also called neuron activation. The former mainly comes from the traditional image processing field, and the latter comes from the field of neocognition. In this article, both of them will be used and refer to same thing. The way to do the extraction is the convolution operation, which is a bit similar to the convolution operation in signal processing, but there are differences between them[9].

Both operands are 3-dimensions [*width,height,depth*] data. The *depth* of both operands must be the same. The other two dimensions of filter are always smaller than feature map.

The convolution involves the concept of stride, which represents the number of pixels filter moves. When stride is 1, then the filter is moved over the feature map one pixel at a time. When stride is 2, the filter is slide two pixels over the feature map at a time.

Based on the concept introduced above, the convolution computation can be decomposed into 2 steps:

- multiplier–accumulator (MAC) operation over filter and part of feature map captured by filter (also called receptive field);
- slide the filter according parameter stride.

Convolution repeats the above two steps until filter traverse the entire feature map. Let's set feature map's size as $[fm_{width}, fm_{height}, depth]$, filter's size as $[filter_{width}, filter_{height}, depth]$ and stride as $[s]$. The size of output of convolution is $[\frac{fm_{width}-filter_{width}}{s}+1, \frac{fm_{height}-filter_{height}}{s}+1, 1]$. Figure 2.3 shows a convolutional computation example.

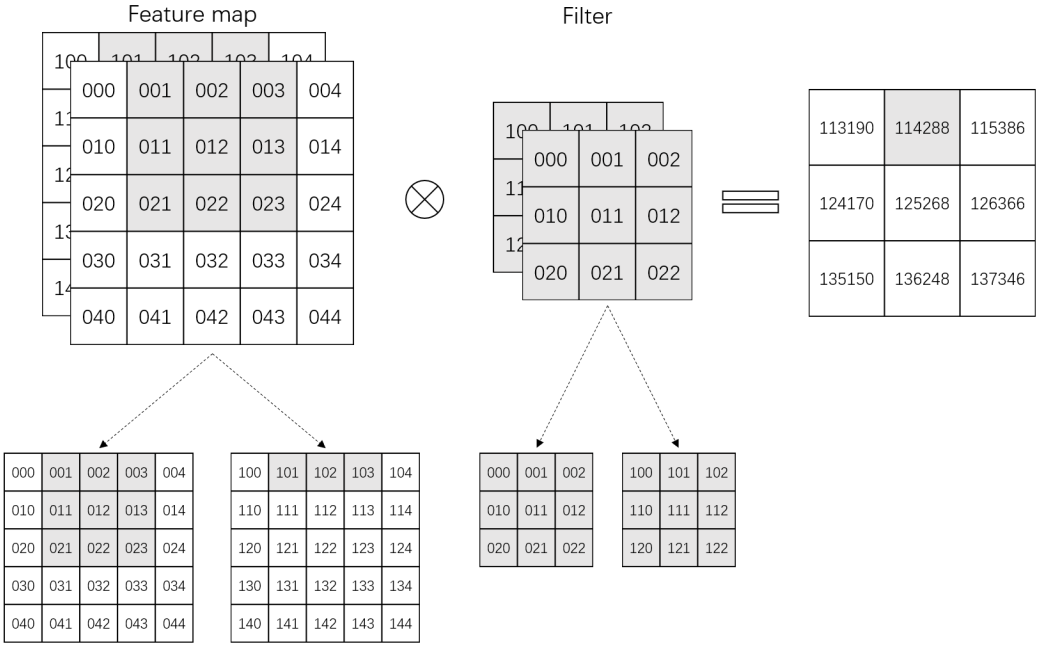

Figure 2.3: Example of convolutional computation

2.3.2 Zero-padding

From the example showed above, we can find that the size of feature map and output of convolution is not the same. The feature map become smaller after each layer, which restricts us to build deeper network and extract deeper feature. So, some time we need to preserve the spatial size. For this purpose

we need to introduce a new concept: zero-padding. That means padding the input feature map with zeros around the border. In addition to maintaining the size, zero-padding can retain the border information.

Apart from zero-padding, there are other padding method, but zero padding is the popular choice in practice. Zeros in a feature map represents nothing, so it does not affect the information carried by the input feature map[9].

Example in figure 2.3 with zero-padding is shown in figure 2.4.

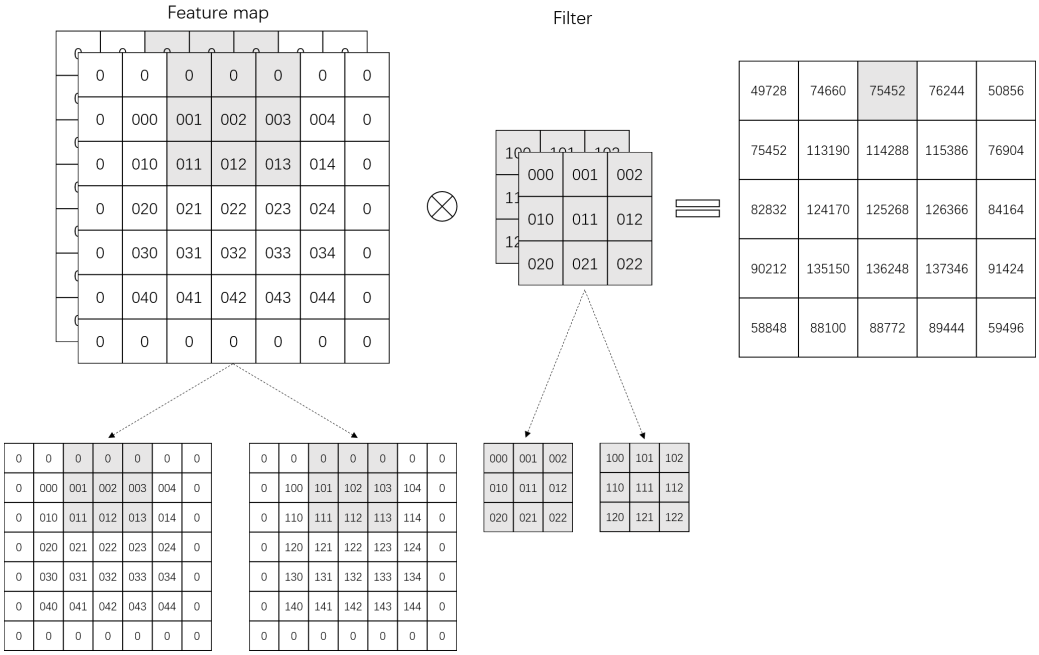

Figure 2.4: Example of convolutional computation with zero padding

2.3.3 Non-Linearity Layer

Non-linearity layer is also referred as activation function layer. The activation function always appears with every convolution layer and fully-connected layer (illustrated in section 2.3.5). There are a lot of activation functions and all of them are nonlinear, in order to increase the nonlinear properties of the overall neural network. As part of biologically inspired neural networks, activation function is usually an abstraction representing action potential firing rate in a neural cell[11].

The rectifier is most widely used activation function in deep convolution neural network. It has proven that it can improve the performance of neural

network model comparing to other activation function[12]. The definition of Rectified Linear Units (ReLU) is $f(x) = max(0,x)$ and shows in figure 2.5[9].

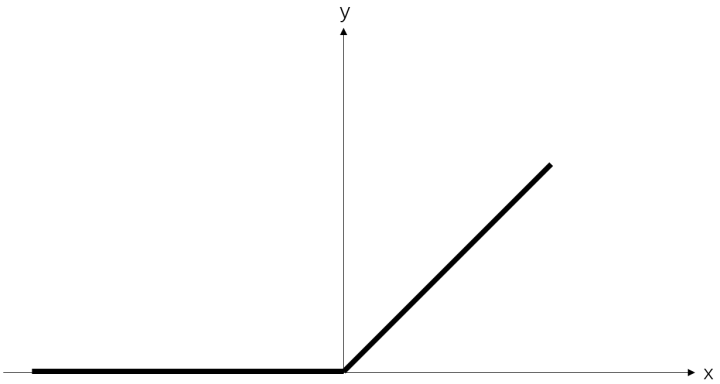

Figure 2.5: Activation function: ReLU

2.3.4 Pooling Layer

Pooling layer is common in convolutional neural networks, especially in large neural network models. Its aim is to progressively reduce the spatial size of the feature map, decreasing the amount of calculation in the next layer and the number of parameters needed to be trained over the whole neural network (reduce the model) and avoiding overfitting.

The input and output of the pooling layer are also called feature map. The pooling layer operates on every channel (depth) of the feature map independently and resize every channel. Although every channel is operating independently, the same operation must be applied over them, ensuring every channel has same size after resizing[9].

The pooling operation also has parameter stride and same computation steps as convolutiona operation, but using maximum (MAX) operation instead of MAC operation. Its computation process is shown below.

- MAX operation over part of feature map captured by filter;
- slide the filter according stride

For pooling layer, MAX operation is not the only option, there are still other operations, like average operation, but maximum is the most widely

used. Figure 2.6 is an example of MAX pooling with pooling size of [2,2] and $stride = 2$.

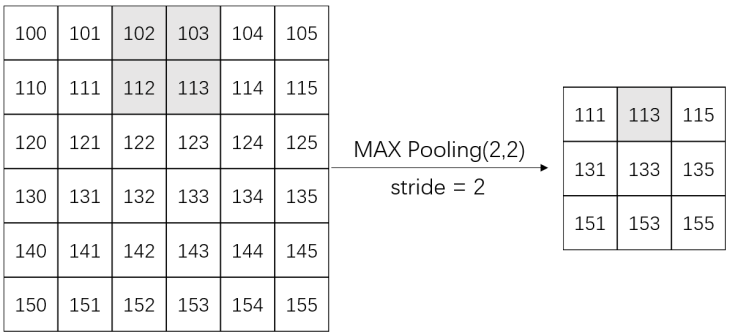

Figure 2.6: Example of MAX pooling

2.3.5 Fully-connected Layer

Fully-connected layer is different from convolution layer, which has only local connectivity, connecting neurons to part of the neurons in the previous layer, but has full connectivity. It connects each neuron to all of the neurons in the previous layer. Multilayer Perceptron is a typical example of a neural network based on fully-connected layers. Figure 2.1 an example of Multilayer Perceptron. As you can observe, fully-connected layer is more complex than convolution layer, has more parameter, and thus more difficult to train and overfitting[9].

In CNN model, the number of output in this layer equals to the number of classes in image classification.

The computation of fully-connected layers is a matrix multiplication. At first, the output of the previous layer is reshaped into vectors with dimension [1,X], then multiplied by a weight matrix with dimension[X,S] and finally summated with bias[1,S]. Figure 2.7 shows the calculation process.

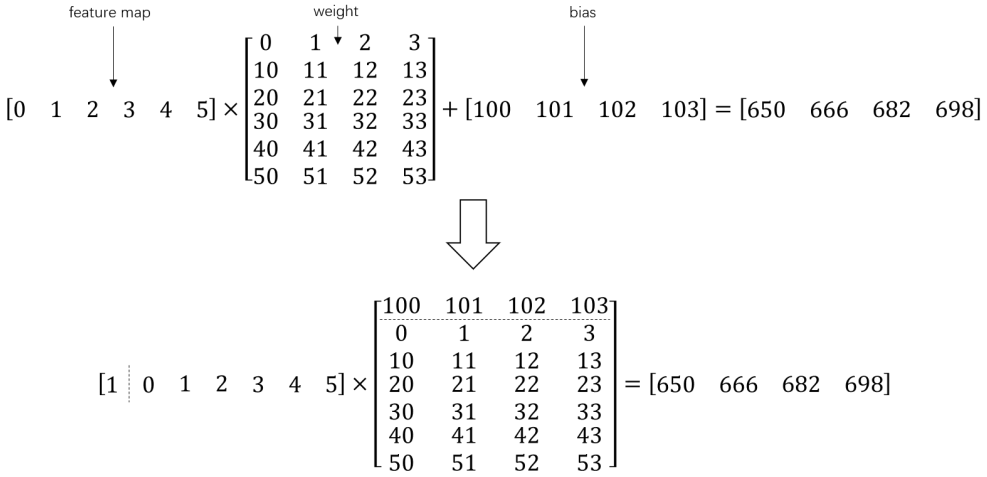

Figure 2.7: Example of fully-connected computation

2.4 Characteristics of CONV and FC

If you calculate the examples of CONV and FC in section 2.3.1 and 2.3.5 with a memoryless and non-programmable calculator, you can find some important features, which are the same as computation in computer.

- Lots of data reuse/retype
  - for CONV:
    1. Filters are reused for each sliding and for each new feature map. This is called filter-reused.
    2. Most of the data elements in a feature map are also reused. If there are more than one filter, feature maps are also reused by all filters. This is so-called feature-map-reused.
  - for FC:
    1. Weights are reused for different feature map. This is so-called weight-reused
    2. Feature maps are reused for different weights. This is so-called feature map-reused.
- memory access bottleneck

– Lots of data reused causes a large number of memory operations during computation lowering the calculation speed.

2.5 Quantization Workflow of TensorFlow

TensorFlow is an open source software library for numerical computation using data flow graphs. It is supported by Google and being the most widely used library for deep learning[5].

Because of training algorithm backpropagation[10] and challenge for modern neural network optimizing for high accuracy and high memory usage, using floating point arithmetic and powerful GPU is an optimal way to realize them. However, as deep learning model are deployed to mobile devices, inference speed and power consumption are the critical issue when mobile devices do not have powerful computing power and use battery as power source. This problem has become more and more serious because the state-of-art models have become larger and larger. It limits the application of deep learning.

As reasons mentioned above, models need to be quantized. Quantized models using unsigned 8-bit-integer calculation help models run faster and use less power. It makes running deep learning models in mobile devices possible, especially import for mobile devices that cannot run floating point code efficiently.

Please keep in mind that the following operations are based on the following environment.

-Operation system: Ubuntu 16.04 LTS

-Version of TensorFlow: TensorFlow R1.2

In order to design an hardware accelerator for TensorFlow quantization model, we must figure out how quantization works in TensorFlow. For how to generate a quantized model in Tensorflow, please see this document[13].

1. Quantization equation
   To map arithmetic on real values (32-bit float point) directly to arithmetic on 8-bit unsigned integer values, the real and quantized value muss keep the following relationship. The symbol S represents a scaling factor and B represents an offset.

$$real\_value = S * quantized\_value + B \tag{2.1}$$

Equivalently, the equation can be express as (D is also an offset.)

$$real\_value = S * (quantized\_value + D) \tag{2.2}$$

After quantization, the real value 0 should still be exactly representable with the quantized set of numbers.
In equation (2.1), setting $real\_value = 0$ and corresponding $quantized\_value = quantized\_zero$, we get

$$0 = S * quantized\_zero + B$$

Equivalently

$$quantized\_zero = \frac{-B}{S}$$

This equation guarantees that real value 0 can be exactly represented with one quantized number.
Now in equation (2.2), setting $real\_value = 0$ and $quantized\_value = quantized\_zero$, we get

$$0 = S * (quantized\_zero + D)$$

It means

$$quantized\_zero = -D$$

With this equivalent, rewriting equation (2.2) into the form (2.3)

$$real\_value = S * (quantized\_value - quantized\_zero) \tag{2.3}$$

Note that the final quantizaton equation (2.3) has two constants, S is an positive real number, another $quantized\_zero$ is an integral.
But how to get S and $quantized\_zero$? Actually, the quantization represents the overall minimum and maximum real values by the lowest (0) and highest (255) quantized value.The other real values are distributed linearly between 0 and 255. As an example, table 2.1 shows the quantization representation for real value minimum = -30.0 and maximum = 10.0. From this example, we get $S = 6.375$ and $quantized\_zero = 191$.

2. Quantizing a matrix multiplication
After getting the quantization equation, now we need to specify what

Quantized	Real
0	-30.0
191	0
255	10

Table 2.1: Example of quantization

the quantized matrix multiplication should do. Here we define $\mathbf{X}$ as matrix for neuron activation, $\mathbf{X[i][j]}$ as an element in $\mathbf{X}$, $\mathbf{W}$ as matrix for filter weight, $\mathbf{W[i][j]}$ as an element in $\mathbf{W}$. The computation of convolution as real value is expressed as equation (2.4).

$$\mathbf{Y} = \mathbf{X} \otimes \mathbf{W} \tag{2.4}$$

According equation (2.3) $\mathbf{X}$ and $\mathbf{W}$ can be expressed as following equations. Note that $\mathbf{X_{z_q}}$ is matrix shaped like $\mathbf{X}$, but filled with quantized zero value of $\mathbf{X}$. It is similar to $\mathbf{W_{z_q}}$.

$$\mathbf{X} = S_X * (\mathbf{X_q} - \mathbf{X_{z_q}}) \tag{2.5}$$

$$\mathbf{W} = S_W * (\mathbf{W_q} - \mathbf{W_{z_q}}) \tag{2.6}$$

With equation (2,5) and (2.6), rewrite equation (2.4) into form (2.7).

$$\mathbf{Y} = S_x S_w * ((\mathbf{X_q} - \mathbf{X_{z_q}}) \otimes (\mathbf{W_q} - \mathbf{W_{z_q}})) \tag{2.7}$$

Each entry of product $\mathbf{Y}$ is the sum (accumulation) of many products of individual matrix entries of $\mathbf{X}$ and $\mathbf{W}$, say

$$\mathbf{Y} = \mathbf{Y[i][j]} = S_X S_W * \sum_{i=1}^{m} \sum_{j=1}^{n} (((\mathbf{X_q[i][j]} - X_{z_q}) * (\mathbf{W_q[i][j]} - W_{z_q}))) \tag{2.8}$$

According to equation (2.3), we have

$$\mathbf{Y} = S_Y * (\mathbf{Y_q} - \mathbf{Y_{z_q}}) \tag{2.9}$$

Here we need to keep in mind that S_X, S_W and S_Y are positive real number, not integer in general. As CNNA only support integer computation part, only the accumulation loop, which is the bulk of the

computational cost, in equation (2.8) is excuted in CNNA.

$$computation\_in\_CNNA = \sum_{i=1}^{m}\sum_{j=1}^{n}(((\mathbf{X_q}[\mathbf{i}][\mathbf{j}] - X_{z_q}) * (\mathbf{W_q}[\mathbf{i}][\mathbf{j}] - W_{z_q}))) \tag{2.10}$$

But equation (2.10) is pretty difficult to compute, we need to disscuss how to avoid the overhead of having to substrate the quantized zero point in the loop. The idea is as follows.
Expanding equation(2.10) using distributivity of matrix multiplication over addition, then the product is equal to the following 4 terms.

1. term: $\sum_{i=1}^{m}\sum_{j=1}^{n}(\mathbf{X_q}[\mathbf{i}][\mathbf{j}] * \mathbf{W_q}[\mathbf{i}][\mathbf{j}])$
   This term is pure the matrix multiplication without offset value.
2. term: $W_{z_q}\sum_{i=1}^{m}\sum_{j=1}^{n}(\mathbf{X_q}[\mathbf{i}][\mathbf{j}])$
   We can compute this term by summing all element of matrix $\mathbf{X}$ together, then multiplying offset W_{z_q}.
3. term: $X_{z_q}\sum_{i=1}^{m}\sum_{j=1}^{n}(\mathbf{W_q}[\mathbf{i}][\mathbf{j}])$
   This term is similar to the 2nd term. We can compute this term by summing all element of matrix $\mathbf{W}$ together, then multiplying offset X_{z_q}.
4. term: $\sum_{i=1}^{m}\sum_{j=1}^{n}(X_{z_q}W_{z_q})$
   This term equals $m * n * X_{z_q}W_{z_q}$.

With these 4 terms, we can take the handling of offset (quantized zero point) out of the accumulation loop. This make the accelerator design easier. Moreover, as ReLU are the most widely used activation function and the range of it's output is [0,∞), therefore $X_{z_q} = 0$. You find that if $X_{z_q} = 0$, the results of the 3rd and 4th term becomes 0 and the computation of the 2nd term becomes easier. So, the computation of equation (2.10) needs only two terms as equation (2.11) shown.

$$computation\_in\_CNNA = \left[\sum_{i=1}^{m}\sum_{j=1}^{n}(\mathbf{X_q}[\mathbf{i}][\mathbf{j}] * \mathbf{W_q}[\mathbf{i}][\mathbf{j}])\right] + \left[W_{z_q}\sum_{i=1}^{m}\sum_{j=1}^{n}(\mathbf{X_q}[\mathbf{i}][\mathbf{j}])\right] \tag{2.11}$$

In chapter 3, you can see we will employ equation (2.11) to design the tensorflow quantization part. The knowledge mentioned above is already enough for accelerator design. But careful readers will find that the result of eqution (2.10) or (2.11) is not uint8 (8-bit unsigned integer). Actually, in TensorFlow, it is int32 (32-bit signed integer). It means, the result need to be requantized into uint8. Now we explain how to requantize into uint8. Firstly we write the matrix product result according eqaution (2.3). Note that $\mathbf{Y_z}$ in equation (2.12)is uint8.

$$\mathbf{Y} = S_Y * (\mathbf{Y_q} - \mathbf{Y_{z_q}}) \tag{2.12}$$

The equation above is equivalent to

$$\mathbf{Y_q} = \frac{\mathbf{Y}}{S_Y} + \mathbf{Y_{z_q}} \tag{2.13}$$

Matrix $\mathbf{Y}$ is already express in equation (2.8). Now we plug equation (2.8) into (2.13) and obtain:

$$\mathbf{Y_q} = \frac{\mathbf{S_x S_w}}{S_Y} * \sum_{i=1}^{m}\sum_{j=1}^{n}(((\mathbf{X_q}[\mathbf{i}][\mathbf{j}] - X_{z_q}) * (\mathbf{W_q}[\mathbf{i}][\mathbf{j}] - W_{z_q}))) + \mathbf{Y_{z_q}} \tag{2.14}$$

In equation (2.13), we already know the value of $\mathbf{S_X}$ and $\mathbf{S_W}$, the maximum and minimum element of matrix $\mathbf{Y_{int32}} = \mathbf{Y_{int32}}[\mathbf{i}][\mathbf{j}] = * \sum_{i=1}^{m}\sum_{j=1}^{n}(((\mathbf{X_q}[\mathbf{i}][\mathbf{j}] - X_{z_q}) * (\mathbf{W_q}[\mathbf{i}][\mathbf{j}] - W_{z_q})))$, and the maximum (255) and minimum (0) value of $\mathbf{Y_q}$. Let me make a example, if the maximum and minimum values of $\mathbf{Y_{int32}}$ are 40000 and -10000, respectively, we obtain equation set (2.15).

$$\begin{cases} 255 = S_{Y_{total}} * 40000 + Y_{z_q} \\ 0 = S_{Y_{total}} * (-10000) + Y_{z_q} \end{cases} \tag{2.15}$$

Solving this equation set get

$$\begin{cases} S_{Y_{total}} = 0.0051 \\ Y_{z_q} = 51 \end{cases} \tag{2.16}$$

Then we can requantize Y_{int32} from int32 into uint8 using $Y_q = 0.0051 * Y_{int32} + 51$ for the next layer computation.

3 Convolutional Neural Network Accelerator

In this chapter, I will describe the architecture and workflow of the Convolutional Neural Network Accelerator (CNNA). CNNA is a new architecture inspired by *Cnvlutin*[2] and improved on the architecture of *DianNao*[1].

As I mentioned before, the heaviest computational layers of Convolutional Neural Network (CNN) is CONV and FC. Among them, the convolution in CONV is the heaviest. So, in order to speed up the computation of CNN, we designed CNNA to accelerate the convolution operation. In addition to this, the design aims to apply in the mobile field, so the power consumption is the most serious problem. To solve these problems, we adopted the following three solutions:

- Convolution needs a lot of dynamic memory access, consuming a lot of power and causing memory bandwidth limitation. To overcome this, two banks of caches (static memory) for neuron activations, also named feature map, and filters are employed. These will heavily reduce memory operation.

- We realized a specially designed architecture for reuse of feature maps (neuron activations) and filters weights. It combines with the first solution to severely decrease the memory operation. About the reuse of feature maps/neuron activations and filters weights, please refer to section 2.4

- Using quantized models/low precision operations, which use 8-bit-integer instead of 32-bit-double-float. Arithmetic operation based on 32-bit-double-float takes much more chip area and consumes much more energy than that based on 8-bit-integer. But due to low precision arithmetic, this accelerator also loses its ability to train neural networks, since the gradients need higher precision for correct convergence.

On the other hand, in order to increase the utilization of the chip, CNNA also supports fully-connected layer as convolution based on matrix operation.

In this chapter, module names, register names, and signal names will be covered. To avoid confusion, the following writing rules are set for them.

- Module name: **Module Name**
- Register name: "register_name"
- Signal name: "signal_name"

In the block diagram, the following rules are used to clearly list the bit width and direction of each signal.

- signal_name(x): denotes bit width of the signal is x. if $x = 1$, it will be omitted.
- **"Direction"**: indicates which module the signal comes from or it goes to.

3.1 Limitation of CNNA

Before going deep into the CNNA architecture, I would like to briefly introduce the features and limitations of CNNA. After reading this chapter you can also understand why there are these restrictions.

According to section 2.3, there are mainly 5 components of CNNs. However, CNNA doesn't support all of them. Table 3.1 shows which are supported by CNNA.

CNNs Component/Feature	Support
Convolutional Layer	Yes
Non-Linearity Layer	No
Pooling Layer	No
Fully Connected Layer	Yes
Zero Padding	No
Loss Layer	No

Table 3.1: CNNA Supported Feature

Even CNNA supports convolutional layer (CONV) and fully-connected layer (FC), there are still some restrictions. For CONV, CNNA is only capable of $stride = 1$. Beside this, CNNA has also limitations on the shape of the feature maps/neuron activations and filters. As the state-of-art CNN models shows, very small convolution filters [3,3] significantly improvement

the performance[14] and larger filter are getting less and less common. Therefore, CNNA's filter size is limited to [6,6]. Other limitations are from a perspective of cost and application convenience. Our original intention is to properly reduce the size and computing power of a single chip and then the software can flexibly select multi-chips to match the size of the model to achieve the best energy efficiency ratio. Given an feature map tensor of shape $[NA\_batch, NA\_height, NA\_width, NA\_channels]$ and a filter tensor of shape $[filter\_height, filter\_width, filter\_in\_channels, filter\_out\_channels]$[15], their restrictions are shown in table 3.2.

Tensor Shape Parameter	Limitations
Feature map: NA_batch	1
Feature map: NA_height	[1: 32]
Feature map: NA_width	No limitation
Feature map: NA_channels	[1: 16]
Filter: filter_height	[1: 6]
Filter: filter_width	[1: 6]
Filter: filter_in_channels	[1: 16]
Filter: filter_out_channels	[1: 16]

Table 3.2: Tensor shape limitation for CONV in CNNA

As FC performs matrix multiplication operations without convolution[16], its shape limitation is different from CONV's. Given the first tensor of shape $[1, X]$ and the second tensor of shape $[X, C]$, X's maximum value is $36 * 16 = 576$ and C's maximum value is 16.

When one of the tensors shape is over the limitation, the computation need to be distributed on 2 or more CNNA chips or seperated into several step.

3.2 Placement of CNNA in computer system

The following brief block diagram figure 3.1 shows the placement of CNNA in computer system. CPU places corresponding data in memory through DMA (Direct Memory Access) controller, then sends notification to the CNNA (through signal "cpu_to_cnn"), waking up CNNA to begin computation. After waking up, CNNA fetches data from memory, executes convolution and stores results back to memory. Finally, CNNA informs CPU (through signal

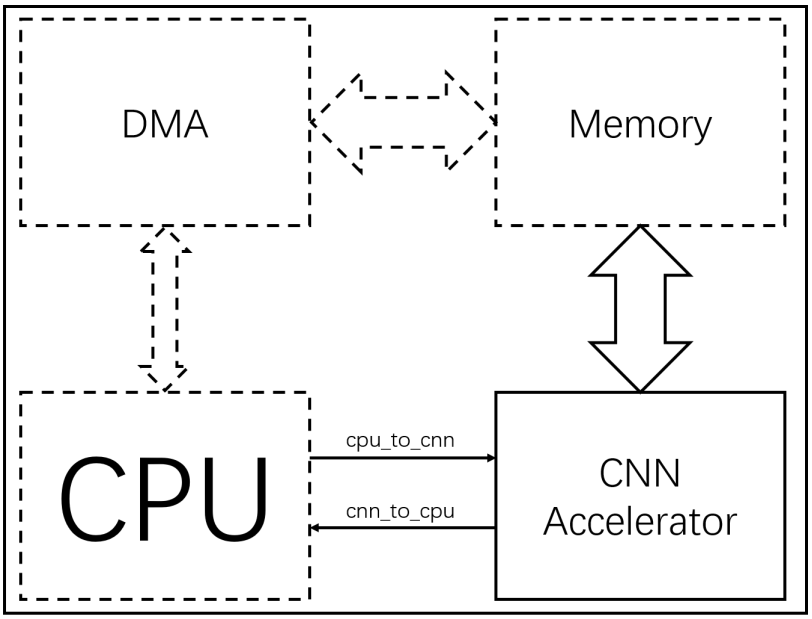

Figure 3.1: Placement of CNNA

"cnn_to_cpu") when it finished the work. The part drawn with solid line is my work, which includes CNNA, memory interface and CPU interface.

One of the goals of CNNA is to provide as co-processor without changing the existing computer system, and to help the CPU perform CNN accelerated tasks. This setup is easy for us to iteratively update the accelerator, minimizing changes to current computer systems. Like the GPU graphics card can be upgraded at any time without having to do the same for the CPU and the motherboard. The second goal is no instruction operation, which reduces the coupling to the system and facilitates modularization.

3.3 Architecture of CNNA

The architecture of CNNA is showed in figure 3.2, which is a brief block diagram. In the following step by step explanation, I will gradually elaborate on each submodule, letting the entire specific structure shown in front of you.

There are three blocks in CNNA: **CNNA Control Unit**, **Distributor** and **Quad Computation Unit Set**.

- **CNNA Control Unit** is responsible for controlling the CNNA.

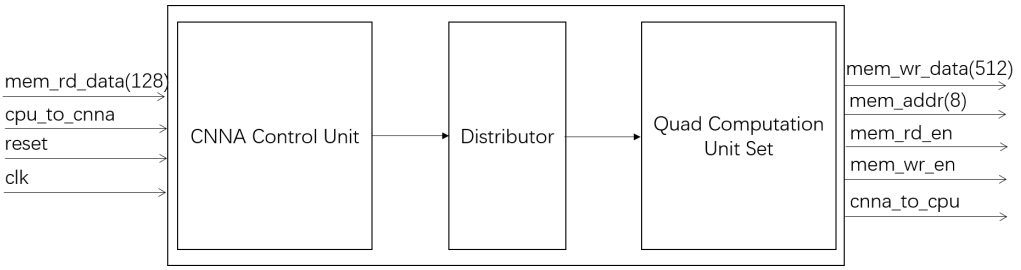

Figure 3.2: General architecture of CNNA

- **Quad Computation Unit Set** has 4 **Quad Computation Unit**s, each **Quad Computation Unit** has 4 **Computation Unit**s, which is used for convolution computing.
- **Distributor** is responsible for distributing data (from memory) and control signal (from **CNNA Control Unit**) to the corresponding **Computation Unit** respectively. As **Computation Unit** employs pipeline technique, **Distributor** pipelines some control signals.

The input and output signals are shown in table 3.3. As mentioned before, CNNA is based on 8-bit integer and can run $16 * 16 = 256$ convolution operations in parallel. As there are 16 **Computation Unit**s, the memory input data bit-width is $8 * 16 = 128$ bits. On the other hand, since the convolution operation involves the MAC (Multiply-Accumulate), the result of the operation will be reserved for 32 bits instead of 8 bits in order to maintain accuracy and facilitate subsequent operations (See section 2.5). So, the output data to memory is $32 * 16 = 512$ bits. As input and output bandwidth is not the same, two block memories outside CNNA are employed to do simulation and validation in chapter 4. Last, there are two types of reset signal in CNNA. The first one is the "reset" signal, which is external and only for **CNNA Control Unit**. Another one is an internal reset signal, which is for **Distributor** and **Quad Computation Unit Set**. The internal reset signal is active before each convolution layer computation, so, it named "layer_reset".

Signal	In/Out	Bit Width	Function
clk	in	1	Clock
reset	in	1	Reset signal
mem_rd_data	in	128	Data readed from 128-bits-memory
cpu_to_cnna	in	1	Signal to active CNNA from CPU
mem_wr_data	out	512	Data written into 512-bits-memory
mem_addr	out	8	Memory address for both memory
mem_rd_en	out	1	Reading enable signal for 128-bits-memory
mem_wr_en	out	1	Writing enable signal for 512-bits-memory
cnna_to_cpu	out	1	Signal to notify CPU when computation is finish

Table 3.3: Signals of CNNA

3.4 CNNA Control Unit

In this section we will discuss **CNNA Control Unit**, which controls the CNNA. Its location in CNNA is shown in figure 3.3. The block diagram

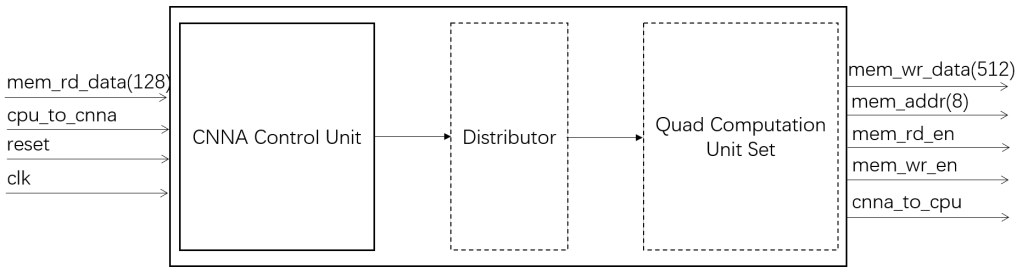

Figure 3.3: CNNA Control Unit in CNNA

of **CNNA Control Unit** is shown in figure 3.4. Note that NA is the abbreviation of Neuron Activation. Next, we briefly introduce the function of each block.

- **FSM Control Logic** is the core control module. **CNNA Control Unit** is under its control.
- **Configuration Register** Set is the module to store the configuration information. It will affect every part of **CNNA Control Unit**.
- **Filter Load Control** is the module, which controls how to load filter weight into filter cache.

- **NA Load Control** is similar to **Filter Load Control** except that it is responsible for neuron activation (or feature map).
- **NA Fetch Control** is responsible for fetching neuron activation from **Neuron Activation Cache** into MAC units of **Computation Unit**.
- **Address Generator** generates memory address signal for both reading and writing memory.

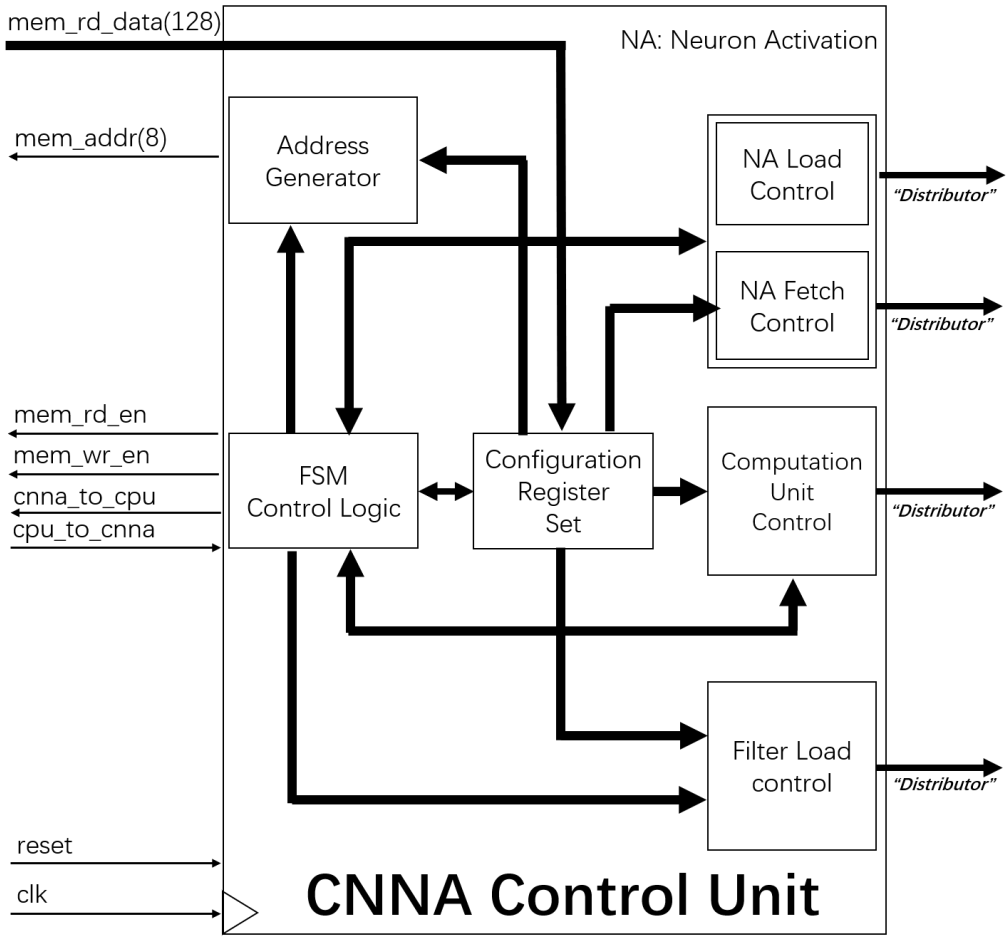

Figure 3.4: Block diagram of **CNNA Control Unit**

Finally, the "reset" signal is not for the whole **CNNA Control Unit** but only for **FSM Control Logic** and **Configuration Registers Set**. The

rest parts are reset by signal "layer_reset", which is generated by **FSM Control Logic**.

3.4.1 Configuration Register Set

Configuration Register Set is one of the two logic unit, which is reset by external signal "reset". **Configuration Register Set** stores all configuration information for CNNA. Before CNNA is excuting a convolution operation, it must read its configuration from memory. That means, every time, when signal "cpu_to_cnna" becomes active (high logic), CNNA will acess the external memory and read 256-bit configuration (two memory-read-operations), then configures and resets itself through "layer_reset", finally, it executes the convolution operation. In our design, the addresses, where CPU stores the configuration information for CNNA, is fixed – the two lowest addresses, 0x00 and 0x01. Note that all configuration information is stored in **Configuration Register Set**.

Table 3.4 is an introduction of all configuration registers in **Configuration Register Set**. The regsiter "platform_support" will generate two enable signal for **Computation Unit Control** and **NA Fetch Control**. They are respectively "tf_quantization_en" and "fully_connected_en". The former is for supporting TensorFlow CNN quantized model and the latter is for supporting fully connected network. The register bit width is a direct response to the CNNA limitation. Please refer to section 3.1 for their reason.

Table 3.5 shows how to calculate the value of each registers. Note that *filter_width, filter_height, NA_width, NA_height, filter_out_channels and filter_in_channels* are the hyperparameter of convolutional layer. In other words, they are the tensor shape of parameter *neuron activation* and *filter*.

Table 3.6 show you the data arrangement of them in memory.
The 128-bit data shown below is separate into 16 blocks (bytes). Each small block has a number and the number represents the corresponding register (please see table 3.5). As we can see from the previous table, every register's bit width is different. But every configuration data in memory still occupies multiples of bytes to ensure alignment. For example, number 8 represent register "neuron_fetch_time", which is 10-bit width and still occupies 2 bytes in memory but only the low 10 bits of 16 bits are valid. It means, CNNA will only store the low 10-bit into "neuron_fetch_time" register, ignoring the high 6-bit.

No.	Register	Bit width	Functionality
1	filter_width	3	width of filter
2	filter_size	6	size of filter
3	picture_width	5	width of feature map
4	picture_height	5	height of feature map
5	neuron_fetch_time	16	number of clocks to fetch NA from NA-Cache
6	num_of_filters	4	number of filters in each layer
7	num_of_channels	4	number of channels in each filter
8	filter_fetch_time	10	number of clocks to fetch filter weight from memory to filter cache
9	load_neuron_time	8	number of clocks to load NA from memory to NA-Cache
10	platform_support	8	0: normal CNN model 1: tensorflow quantizated CNN model 2: fully connected model Other: reserved
11	quan_weight_zero	8	zero quantified value for filter weight
12	filter_height	5	Height of filter (used only for fully connected model)
13	reading_address	8	the beginning address of memory block, where stores filter weight and NA
14	writing_address	8	the beginning address of memory block, where CNNA stores its computation results

Table 3.4: Registers in **Configuration Register Set**

No.	Register	Value
1	filter_width	$filter\_width - 1$
2	filter_size	$filter\_width \times filter\_height - 1$
3	picture_width	$NA\_width - 1$
4	picture_height	$NA\_height - 1$
5	neuron_fetch_time	$(NA\_width - filter\_width + 1)$ $\times (NA\_height - filter\_height + 1) - 3$
6	num_of_filters	$filter\_out\_channels - 1$
7	num_of_channels	$filter\_in\_channels - 1$
8	filter_fetch_time	$A \times filters\_out\_channels - 1$ $A = filter\_width \times filter\_height$
9	load_neuron_time	$\min[A, B]$ $A = NA\_width \times NA\_height - 1$ $B = 7 \times NA\_height - 1$
10	platform_support	$0\ or\ 1\ or\ 2$
11	quan_weight_zero	"from quantizated tensorflow model"
12	filter_height	$filter\_height - 1$ $(only\ for\ fully\ connected\ model)$
13	reading_address	"begining address of memory storing data"
14	writing_address	"begining address of free memory space"

Table 3.5: How to calculate the value of registers

Address	128-bit data														
0x00			12	11	10	9	8		7	6	5		4	3	2	1
0x01															14	13

Table 3.6: Memory arrangement of registers

3.4.2 FSM Control Logic

FSM Control Logic is one of the two logic unit, which is reset by external reset signal "reset". **FSM Control Logic** is the kernel of CNNA, it controls the running of CNNA. Finite State Machine (FSM) is a systematic content, which facilitates you to have a clear understanding of the running order of CNNA.

There are 15 states in **FSM Control Logic** as shown in table 3.7. As you can see from the table, **FSM Control Logic** needs three configuration information from **Configuration Register Set**. They are "filter_fetch_time", "load_neuron_time" and "neuron_fetch_time". In order to get these configuration information, **FSM Control Logic** needs to read them from memory at first in state *FETCH_CONFIG_REG* and *FETCH_CONFIG_REG_1*. Another thing, the "layer_reset" signal, which is mentioned several times before, is generated here in state *LAYER_RESET_EN*. Last, I describe each state in detail.

- After signal "cpu_to_cnna" from CPU becoming active (Logic HIGH), FSM turns from *IDLE* to *FETCH_CONFIG_REG*. **FSM Control Logic** generates read enable signal and controls **Address Generator** to generate address 0x00 to memory.
- After one clock, FSM turns into *FETCH_CONFIG_REG_1*: **FSM Control Logic** enable registers in **Configuration Register Set** to store the data.
- After one clock FSM turns into *FETCH_MEM_ADDR_REG*: **FSM Control Logic** generates read enable signal and controls **Address Generator** to generate address 0x01 to memory.
- After one clock, FSM turns into *FETCH_MEM_ADDR_REG_1*: **FSM Control Logic** enable registers in **Configuration Register Set** to store the reading address and writing address, which are used for **Address Generator** in the later memory operation.
- After one clock, CNNA has fetched all the configuration information, FSM turns into *LAYER_RESET_EN*: **FSM Control Logic** generates an internal reset signal "layer_reset", which resets the whole CNNA except **FSM Control Logic** and **Configuration Register Set**.
- After one clock, CNNA has been reset for computation, FSM turns into *FETCH_FILTER_WEIGHT* and *FETCH_FILTER_WEIGHT_1*:

The former state for reading memory, latter one state for storing the memory data into filter cache. These two states are looped until all filters are cached. This process takes "filter_fetch_time" clocks, which is stored in **Configuration Registers Set**.

- After "filter_fetch_time" clocks, FSM turns into *CACHE_LOADING* and *CACHE_LOADING_1*: the former state for reading memory, another state for storing the memory data into **Neuron Activation (NA) Cache**, which is located in **Quad Computation Unit Set**. These two states are looped until **NA Cache** is full or all neuron activations (feature map) is cached. Under normal circumstances, the size of neuron activations are larger than that of **NA Cache**, therefore, CNNA can only cache part of them. This process takes "load_neuron_time" clocks, which is stored in **Configuration Registers Set**.

- After "load_neuron_time" clocks, FSM turns into *NEURON_FETCH*: CNNA fetches data from **NA Cache** into MAC unit, which is the main calculation unit in **Quad Computation Unit Set**. Because of pipeline technique, this state needs 4 clocks.

- After 5 clocks, FSM turns into *NEURON_FETCH_AND_OPERAND_FETCH*: In this state, CNNA fetches filter weights into the MAC unit. Note that, the filter weights goes directly into the MAC unit (1 clock), but neuron activation needs 4 clocks (It is done in state *NEURON_FETCH*). During the fetching of operand into MAC unit, CNNA also caches the rest neuron activation from memory into **NA Cache**, updating **NA Cache**. In addition to this, CNNA stores its computation result into memory at the same time to fully utilize memory bandwidth. So, this state is the busiest state. This process takes "neuron_fetch_time" clocks, which is stored in **Configuration Registers Set**.

- After "neuron_fetch_time" clocks, FSM turns into *OPERAND_FETCH*: In this state, CNNA keeps fetching operands into MAC units and writing results to memory but never reading data from memory again.

- After 3 clocks, FSM turns into *WAITING*: Because of pipeline technique, CNNA still runs until data flow out of the pipeline. At the same time, CNNA still writes results to memory.

- After 8 clocks, FSM turns into *CNN_NOTI_CPU*: CNNA notifies CPU that the computation is finished through pulling up signal "cnna_to_cpu"

for one clock.

No.	State	functionality	Duration
0	IDLE	Hibernation	--
1	FETCH_CONFIG_REG	Fetch configurations into CNNA: fetch phase	1 clock
2	FETCH_CONFIG_REG_1	Fetch configurations into CNNA: configuration phase	1 clock
3	FETCH_MEM_ADDR_REG	Fetch memory address information into CNNA: fetch phase	1 clock
4	FETCH_MEM_ADDR_REG_1	Fetch memory address information into CNNA: configuration phase	1 clock
5	LAYER_RESET_EN	Reset CNNA except CNNA configuration and FSM	1 clock
6	FETCH_FILTER_WEIGHT	Fetch filter weight into filter cache: fetch phase	"filter_fetch_time" (configuration info)
7	FETCH_FILTER_WEIGHT_1	Fetch filter weight into filter cache: load phase	"filter_fetch_time" (configuration info)
8	CACHE_LOADING	Fetch neuron activation into neuron cache: fetch phase	"load_neuron_time" (configuration info)
9	CACHE_LOADING_1	Fetch neuron activation into neuron cache: load phase	"load_neuron_time" (configuration info)
10	NEURON_FETCH	Fetch neuron activation from neuron cache	5 clocks
11	NEURON_FETCH_AND_OPERAND_FETCH	Fetch neuron activation from neuron cache and fetch operands (neuron activation and filter weight) into MAC units	"neuron_fetch_time" (configuration info)
12	OPERAND_FETCH	fetch operands (neuron activation and filter weight) into MAC units	3 clocks
13	WAITING	Wait for the computation stream in the pipeline	8 clocks
14	CNN_NOTI_CPU	CNNA notifies CPU: finish signal	1 clock

Table 3.7: States in FSM

Figure 3.5 is the state transition diagram.

From what we discussed earlier, we know that **FSM Control Logic** is also a signal transfer and processing station, exchanging information frequently with other modules in **CNNA Control Unit** so that various modules work in harmony with each other, especially for memory operation. Figure 3.6 shows the signal connection between **FSM Control Logic** and other modules. **FSM Control Logic** generates an enable signal for each module. Beisde this, **FSM Control Logic** also has feedback signals from

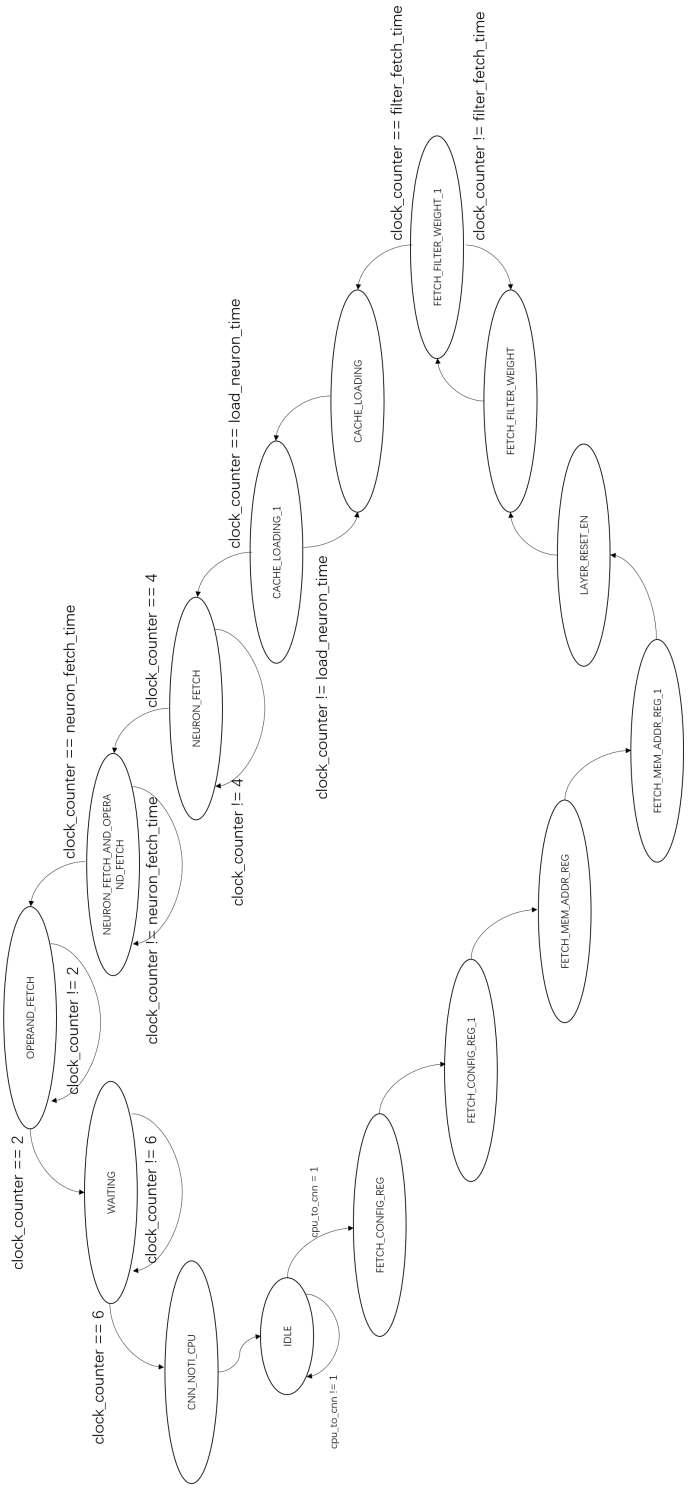

Figure 3.5: State transition

them. Signal "mem_read_active" from **NA Fetch Control** notifies **FSM Control Logic** to cache data from memory into **NA Cache**, which locates in **Quad Computation Unit Set** and is not shown below. **Computation Unit Control** informs **FSM Control Logic** through "mem_write_active" to generate memory control signal in time when computation results are going to be on the data bus.

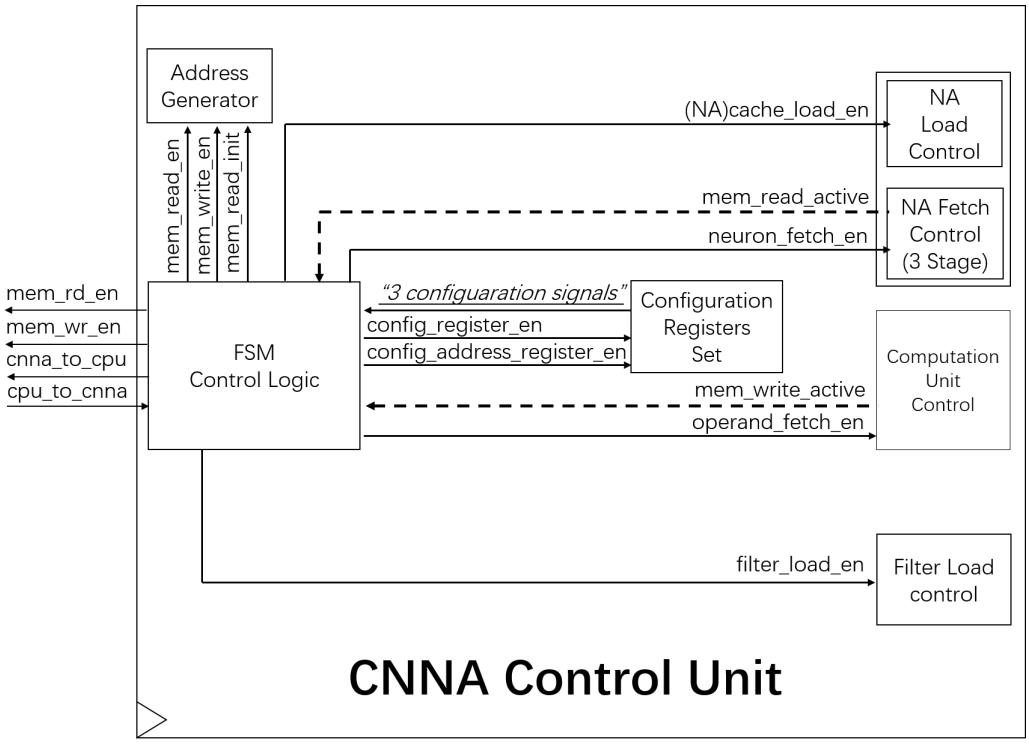

Figure 3.6: Signal connection from/to **FSM Control Logic**

3.4.3 Address Generator

From the above section, we discover that **Address Generator** generates 3 types of memory address signal:

- memory reading address signal during CNNA's initialization, which are fixed, 0x00 and 0x01.
- memory reading address signal during computation, which is depending on the register "reading_address" in **Configuration Register Set**.

- memory writing address signal during computation, which is depending on the register "writing_address" in **Configuration Register Set**.

Address Generator is reset by internal reset signal "layer_reset". Figure 3.7 is the block diagram of **Address Generator**.

generatorblockdiagrmam.png

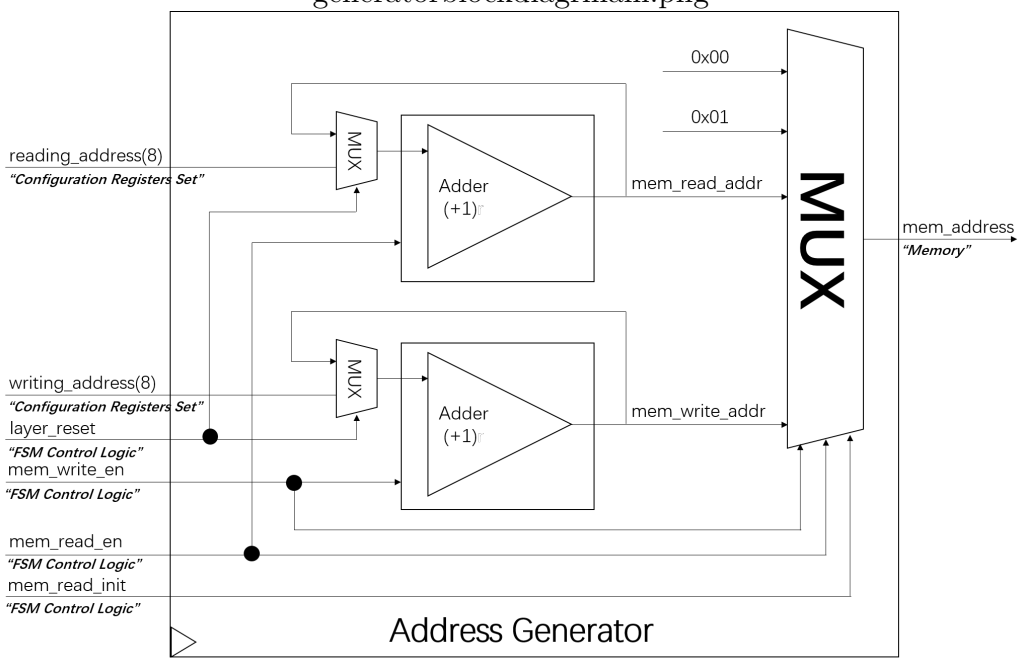

Figure 3.7: Block diagram of **Address Generator**

From the above discussion we can draw the external memory space distribution in figure 3.8. The memory space for reading must be continuous. The same as for writing space. Note that, the reading space is not muss be previous to the writing space.

3.4.4 Computation Unit Control

Computation Unit Control is partly reset by internal reset signal "layer_reset". Its block diagram is shown in figure 3.9. As we explained in section 3.3, there are 16 **Computation Unit**s in CNNA. **Computation Unit Control** is used to control the running of **Computation Unit**. As we can see from figure 3.4, **Computation Unit Control** needs configuration information from **Configuration Register Set**: "num_of_channels", "num_of_filters" and "quan_weight_zero". **Computation Unit Control** has following functions:

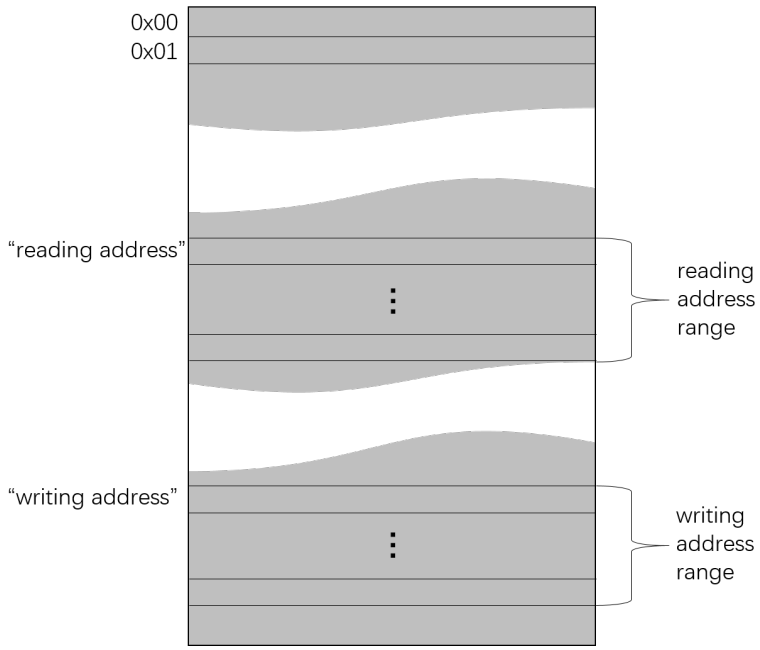

Figure 3.8: External memory arrangment for CNNA

- controls which **Computation Unit**s are active according configuration signal "num_of_channels", saving power consumption

- controls which MAC units in each employed **Computation Unit** are active according configuration signal "num_of_filters", saving power consumption.

- controls when to activate the **Accumulate Adder Array Unit** (AAA), which is located in **Quad Computation Unit Set**. There are two stage AAAs. They are placed after **Computation Unit**s, summing results of them together, converting $16 * 16$ results into 16 final results of CNNA. MAC units in **Computation Unit** work several clocks to accumulate the results. Only after it's work is finish, two stage AAAs will then work for only 1 clock. That means, AAAs are free for most of the time. In order to save power consumption, AAAs are precisely controlled when they are active.

- distributes TensorFlow quantization supporting signals to **TensorFlow Quantization Unit**s in 16 **Computation Unit**s.

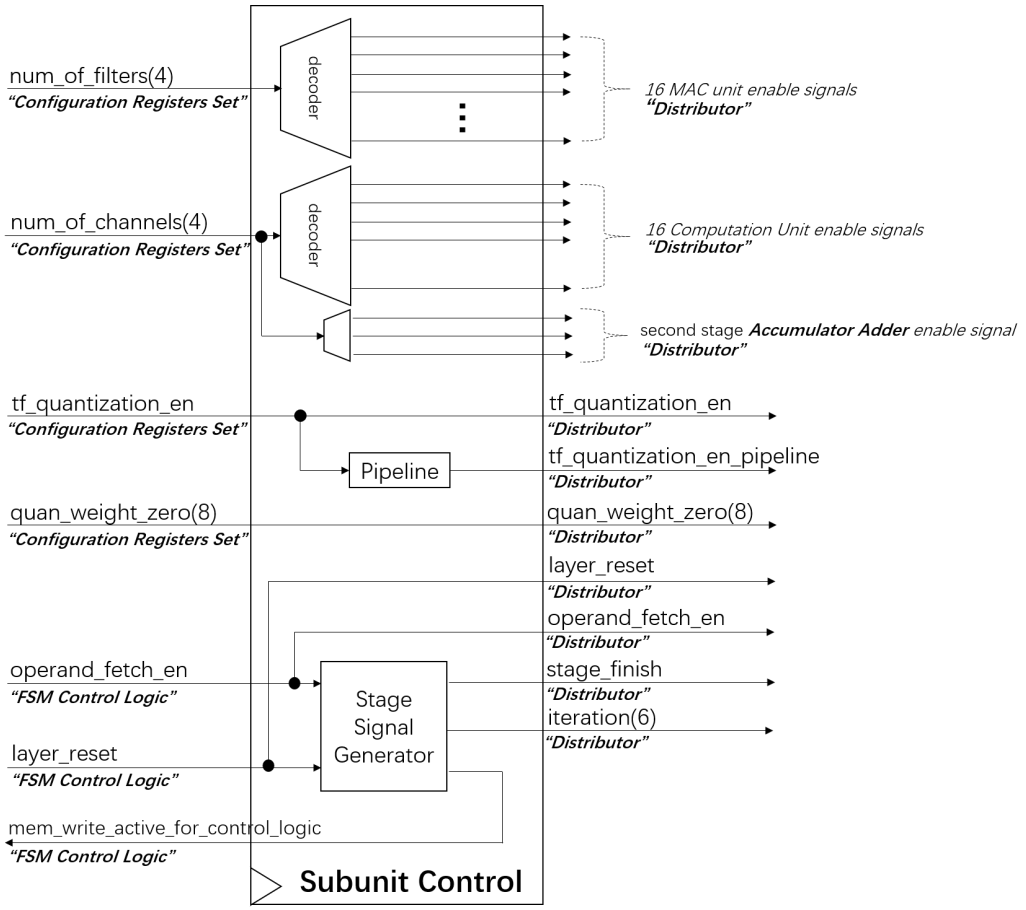

Figure 3.9: Block diagram of **Computation Unit Control**

- generates stage finish signal for one full filter-subfeature map MAC. It is pipelined to control **Operand Fetch Control**, **MAC Array Unit**, **TF Adder Array Unit** and **Accumulate Adder Array Unit**.
- feedbacks **FSM Control Logic** to inform that when is time to write computation results into memory.

3.4.5 Neuron Activation Control

Firstly, note that NA is an abbreviation for neuron activation. As shown in figure 3.4 and figure 3.10, **Neuron Activation Control** is separated into 2 parts. One is **NA Load Control**, which loads neuron activation from memory to **NA Cache**, another is **NA Fetch Control**, which fetches neuron activation from **NA Cache** (in **Computation Unit**) to MAC units (in **Computation Unit**). In this section, we will firstly talk about **NA Load Control**, then turns to **NA Fetch Control**.

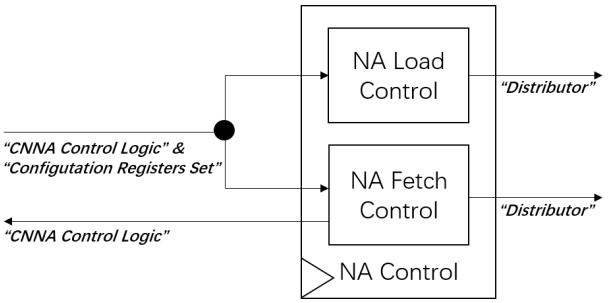

Figure 3.10: Block diagram of **NA Control**

(i) NA Load Control

NA Load Control's block diagramm is shown in figure 3.11. It is reset by internal signal "reset_layer". For NA loading operation, there are other modules, which cooperate with **NA Loading Control** to fulfill the loading operation. These modules are **FSM Control Logic**, illustrated in section 3.4.2 and **Address Generator**, illustrated in section 3.4.3. As we known already, **Address Generator** is responsible for memory address, generating memory address signal. **FSM Control Logic** generates memory write/read enable signal. **NA Loading Control** then stores the data from memory into the right place in **NA Cache**. **FSM Control Logic** will properly control the timing of all these signals. For more properly understanding

how **NA Load Control** works, **NA Cache** is important, please refer to section 3.6.1 for detailed information. Next, I explain the functionality of each

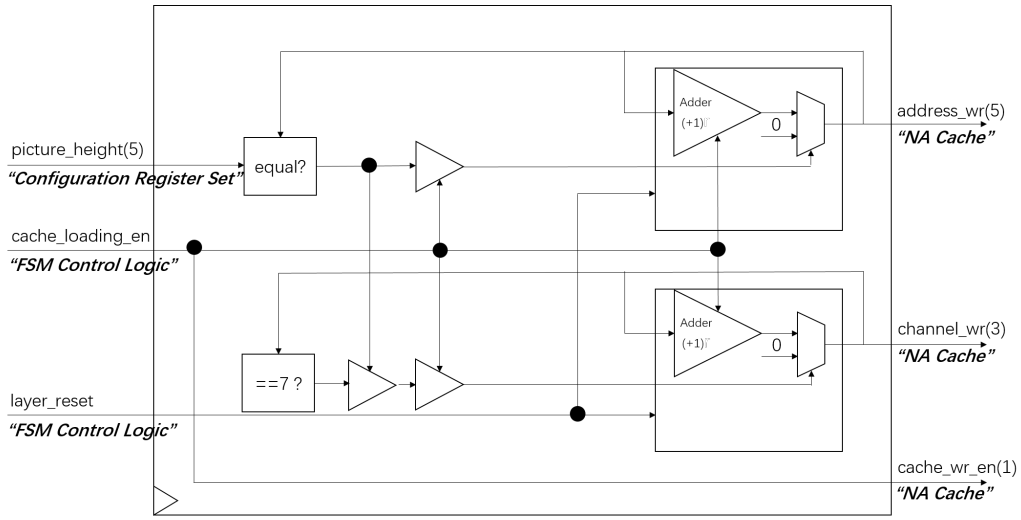

Figure 3.11: Block diagram of **NA Load Control**

signal. "cache_loading_en" is the enable signal for buffers and adders. NA Cache is a two-dimension cache, so, two addresses "address_wr" and "channel_wr" are needed. Only the value of "address_wr" equals "picture_height", then "address_wr" returns to 0. Only the value of "address_wr" equals "picture_height" and the value of "channel_wr" equals 7, then "channel_wr" returns to 0. Otherwise if "cache_loading_en" is active, "address_wr" and "channel_wr" will accumulate 1.

(ii) NA Fetch Control

For fetching operation, **NA Fetch Unit**, which locates in **Computation Unit**, is needed to corporate with **NA Fetch Control**. The details about **NA Fetch Unit** please refer to section 3.6.2. In addition to fetch NA, **NA Fetch Control** also sends feedback to **CNNA Control Unit** and informs **CNNA Control Unit** that when is time to load new data from memory into **NA Cache** and how many data should be loaded this time. The block diagram of **NA Fetch Control** is shown in figure 3.12. Each block will be detailed explained below.

(1)Pipeline Control Logic

The kernel of **NA Fetch Control** is **Pipeline Control Logic**. There is

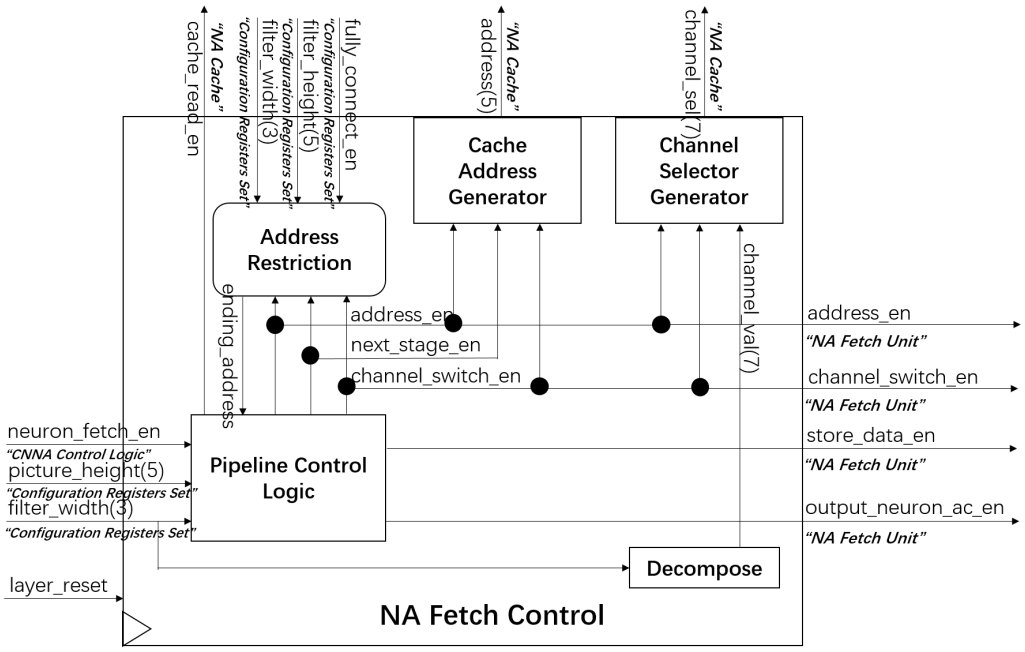

Figure 3.12: Block diagram of **NA Fetch Control**

dedicated FSM control logics inside it. Each of them is a thread generating controlling sequence. The controlling sequence is separated into 4 steps. The order is shown below.

Cache Reading (CR) -> Data Rearrangement (DR) -> Data Store/Address Update(DSAU) -> Neuron Activation Output (NAO)

Output signal "cache_read_en" is generated in step *Cache Reading (CR).*
Output signal "store_data_en" and "address_en" is generated in step *Data Store/Address Update(DSAU).*
Output signal "output_neuron_ac_en" is generated in step *Neuron Activation Output (NAO).*
There is no corresponding control signal for the 2nd step *Data Rearrangement (DR).* This step is reserved for data arrangement. Because data from NA Cache is not suitable for convolution operation, this step is used for **NA Fetch Unit** to rearrange the data to be fit for convolution.

Because there are 4 step, 4 threads are needed in order to fully utilize the hardware resource, keeping outputting neuron activation to MAC. The output of **Pipeline Control Logic** is an OR operation of the outputs of

these 4 threads. In addition to this, different "filter_width" causes different controlling sequence. Figure 3.13, 3.14 and 3.15 show how controlling sequence varies with different "filter_width".

filter_width = 1									
Clock	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Thread 0	CR	DR	DSAU	NAO	CR	DR	DSAU	NAO	CR
Thread 1	/	CR	DR	DSAU	NAO	CR	DR	DSAU	NAO
Thread 2	/	/	CR	DR	DSAU	NAO	CR	DR	DSAU
Thread 3	/	/	/	CR	DR	DSAU	NAO	CR	DR
OR Operation (Threads)	CR	DR CR	DSAU DR CR	NAO	NAO	NAO	NAO	NAO	NAO
				DSAU DR CR	DSAU DR CR	DSAU DR CR	DSAU DR CR	DSAU DR CR	DSAU DR CR

Figure 3.13: Pipeline process: filter_width=1

For "filter_wdith"=1, each thread's period is 4 clocks. *NAO* is valid for 1 clock during one period. 1 clock difference between adjacent threads. The output of neuron activation to MAC continues from the 4th clock cycle.

filter_width = 2																			
Clock	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15				
Thread 0	CR	DR	DSAU	NAO	NAO	/	/	/	CR	DR	DSAU	NAO	NAO	/	/	/	CR	DR	DSAU
Thread 1	/	/	CR	DR	DSAU	NAO	NAO	/	/	/	CR	DR	DSAU	NAO	NAO	/	/	/	CR
Thread 2	/	/	/	/	CR	DR	DSAU	NAO	NAO	/	/	/	CR	DR	DSAU	NAO	NAO	/	/
Thread 3	/	/	/	/	/	/	CR	DR	DSAU	NAO	NAO	/	/	/	CR	DR	DSAU	NAO	NAO
OR Operaion	CR	DR	DSAU CR	NAO	NAO	NAO	NAO	NAO	NAO	NAO	NAO	NAO	NAO	NAO	NAO	NAO	NAO	NAO	NAO
				DR	DSAU CR	DR	DSAU CR	DR	DSAU CR	DR	DSAU CR	DR	DSAU CR	DR	DSAU CR	DR	DSAU CR	DR	DSAU CR

Figure 3.14: Pipeline process: filter_width=2

For "filter_wdith"=2, each thread's period is 8 clocks but only 5 clocks are valid, another 3 clocks are empty. *NAO* is valid for 2 clocks during one period. 2 clocks difference between adjacent threads. The output of neuron activation to MAC continues from the 4th clock cycle.

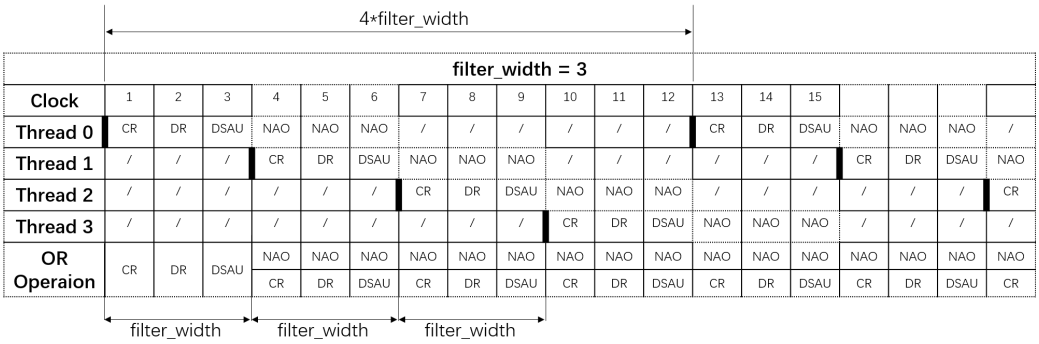

Figure 3.15: Pipeline process: filter_width=3

For "filter_wdith"=3, each thread's period is 12 clocks but only 6 clocks are valid. *NAO* is valid for 3 clocks during one period. 3 clocks difference between adjacent threads. The output of neuron activation to MAC continues from the 4th clock cycle.
The law can be obtained from the previous observation. The number of clocks in a period:

$$T_{clocks} = 4 * filter\_width$$

The number of valid clocks in a period:

$$T_{clocks\_valid} = 3 + filter\_width$$

The difference clocks between adjacent threads:

$$T_{difference} = filter\_width$$

(2)Address Restriction

Address Restriction predicts the stopping address in advance, then feedback to **Pipeline Control Logic** to generate an appropriate control signal, satisfying the NA cache operation for convolution. In convolutional acceleration, input signal "filter_height" and "fully_connetced_en" is not used, they are only active during accelerating fully connected layer. This is the only place, where CNN and fully connected network are different in hardware design. **Address Restriction** and **Pipeline Control Logic** restrict that the height and width of filter muss be equal in convolutional layer but not muss be equal in fully connected layer.

(3)Cache Address Generator and Channel Selector Generator

NA Cache is a two-dimension cache, reading it needs also two-dimension addresses.

(iii) Connection between NA Control, NA Cache and NA Fetch Unit

The block diagram is shown in figure 3.16. The solid line parts belong to **Neuron Activation Control**.**NA Cache** and **NA Fetch Unit** locate in **Computation Unit** and are dedicated for each **Computation Unit**. Therefore, there are 16 **NA Cache**s and **NA Fetch Unit**s in CNNA. All of them share one **NA Control**. For the detailed information about **NA Cache** and **NA Fetch Unit**, please refer to section 3.6.1 and section 3.6.2.

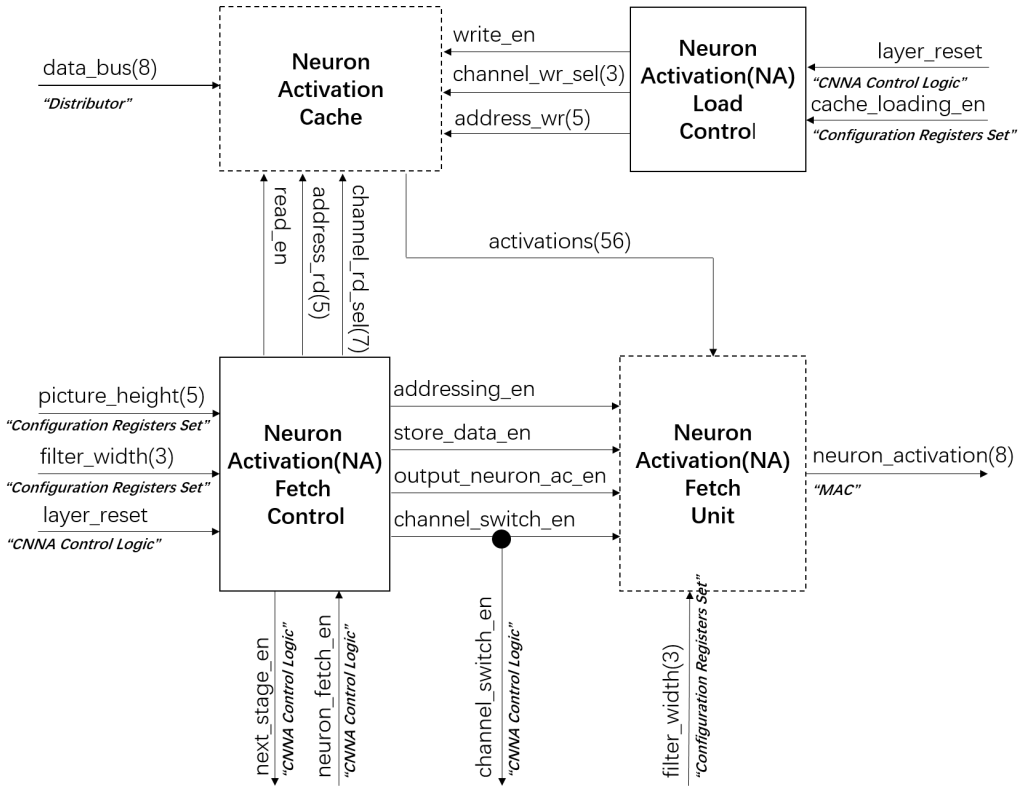

Figure 3.16: Connection between **NA Control**, **NA Cache** and **NA Fetch Unit**

3.4.6 Filter Load Control

Figure 3.17 shows the construction of **Filter Load Control**, which is reset be internal signal "layer_reset". **Filter Load Control** cooperates with **Address Generator** and **FSM Control Logic**, reading data from memory and storing them into **Filter Weight Cache**s. CNNA reads 128 bits data every time and cache them into 16 **Computation Unit**s. Every **Computation Unit** has a **Filter Weight Cache**, so, it obtains 8bits (1 Byte). All 16 **Computation Unit**s share one **Filter Load Control**. For the detailed design of **Filter Weight Cache** and how CNNA caches filter weight, please refer to section 3.6.3.

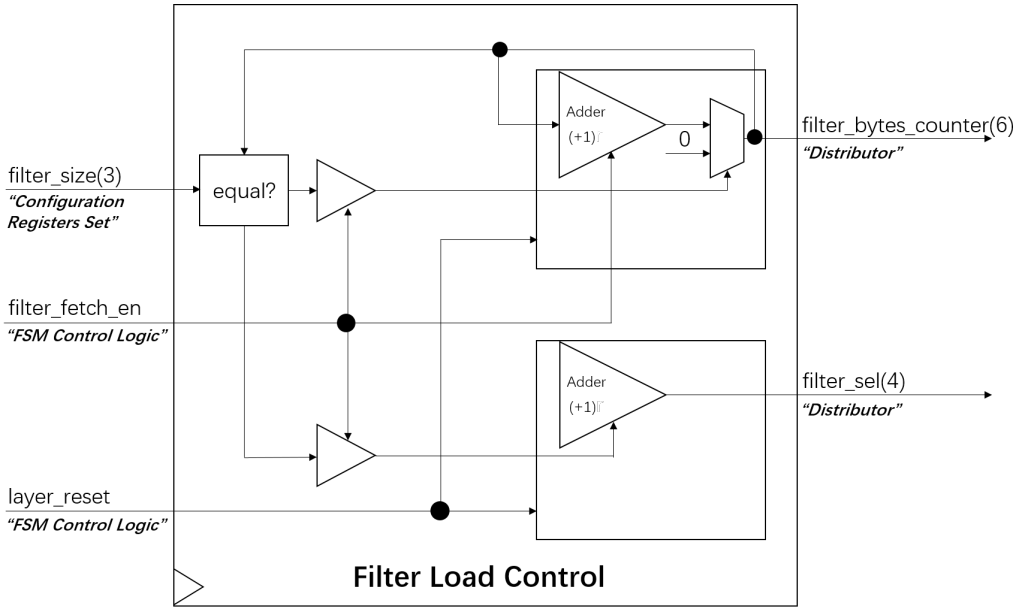

Figure 3.17: Block diagram of **Filter Fetch Control**

Connection between **Filter Load Control**, **Filter Weight Cache** and data bus from memory is shown in figure 3.18. The functionality of signals are shown in table 3.8.

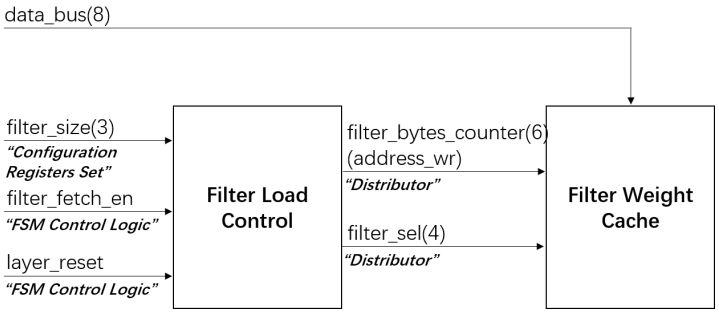

Figure 3.18: Connection between **Filter Fetch Control** and **Filter Weight Cache**

Signal	Bit Width	Function
layer_reset	1	Reset **Filter Load Control**
filter_size	3	Compare with signal "filter_counter_bytes", if equal, set "filter_counter_bytes" to be 0, let "filter_sel" add 1, switching to next filter.
filter_fetch_en	1	**Filter Load Control** enable signal
data_bus	8	data from memory
filter_counter_bytes	6	Address to index depth of **Filter Weight Cache**; After reset, the value is 0.
filter_sel	4	Address to index width of **Filter Weight Cache**; After reset, the value is 0.

Table 3.8: Signals of **Filter Fetch Control**

3.5 Distributor

Distributor, which locates in CNNA like figure 3.19 shows, is a bridge between **CNNA Control Unit** and **Quad Computation Unit Set**. Its block diagram is shown in figure 3.20. Mentioned earlier, **Distributor** distributes 128 bits data into 16 **Computation Unit**s. Each **Computation Unit**'s data bus bandwidth is 8-bit. Beside this, all the control signals are AND with computation unit active signal so that only active **Computation Unit**s will work to save power consumption, which is also called signal gating.

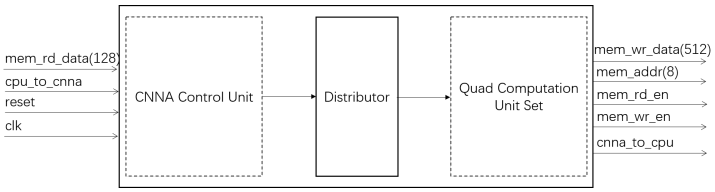

Figure 3.19: Distributor in CNNA

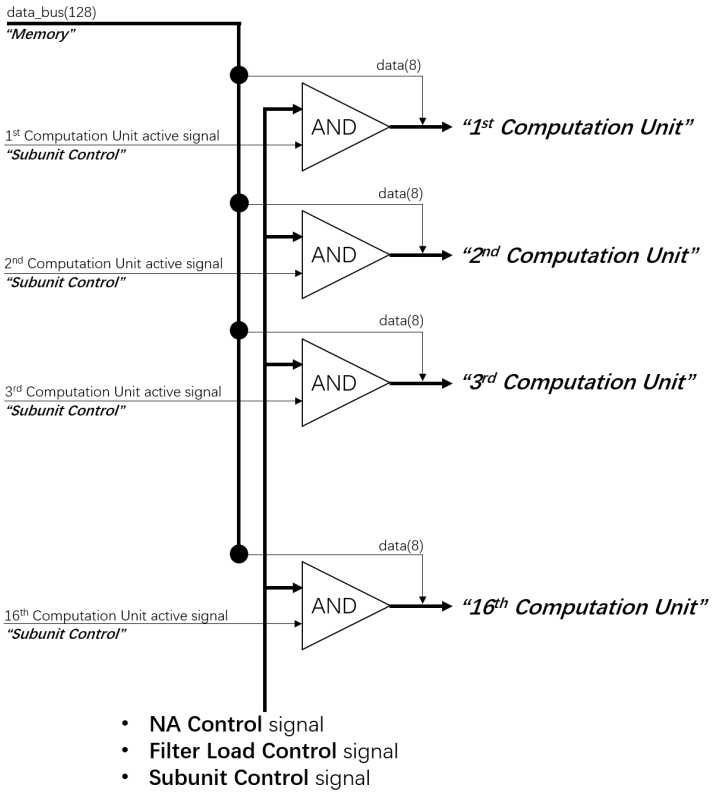

Figure 3.20: Block diagram of **Distributor**

3.6 Quad Computation Unit Set

Figure 3.21 shows the **Quad Computation Unit Set** in CNNA. It has 4

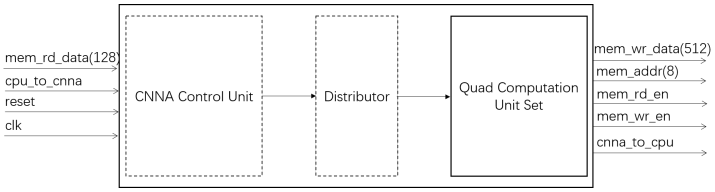

Figure 3.21: Quad Computation Unit Set in CNNA

Quad Computation Units. Each of them has 4 **Computation Unit**s. Therefore, CNNA has a total of 16 **Computation Unit**s, which are identical. The first limitation of CNNA for supporting CNN model is generated here: maximum number of channels of neuron activation/filter is 16.

Each **Computation Unit** (CU) has 16 outputs, the same as CNNA. So, there are **Accumulate Adder**s (AA) between CU's output and CNNA's output. In my design, there are two stages AA. The first stage locates in each **Quad Computation Unit** and the second stage is in **Quad Computation Unit Set**.

The block diagram of **Computation Unit** is shown in figure 3.22. Neuron activations (or feature maps) are stored in **NA Cache**. **NA Fetch Unit** is responsible for data rearrangement and fetches neuron activation into MAC units. Filter weights are stored in **Filter Weight Cache**. **Operand Fetch Control** will deactive **Filter Weight Cache**, **MAC Array Unit** and **TF Adder Array Unit** to save power. Beside this, **Operand Fetch Control** has a cache control logic inside, which is used to fetch filter weight from cache. **TensorFlow Quantization Unit** is employed to support quantized CNN model. **Computation Unit** employs pipeline design to speed up the calculation. The pipelined signal is "mac_en". The output signals of **Computation Unit** are going to block **First Stage Accumulate Adder**. All these blocks will be explained in details in the following subsection.

The block diagram of **Quad Computation Unit** is shown in figure 3.23. Each **Computation Unit** has 16 outputs and bit width of each output is 24 bits. Each **Quad Computation Unit** has also 16 outputs, but bit width is 28 bits. The first stage **Accumulate Adder** locates here to sum over the outputs of **Computation Unit**s. The first output of **Quad Computation Unit** is the sum of the first output of 4 **Computation Unit**s. The second

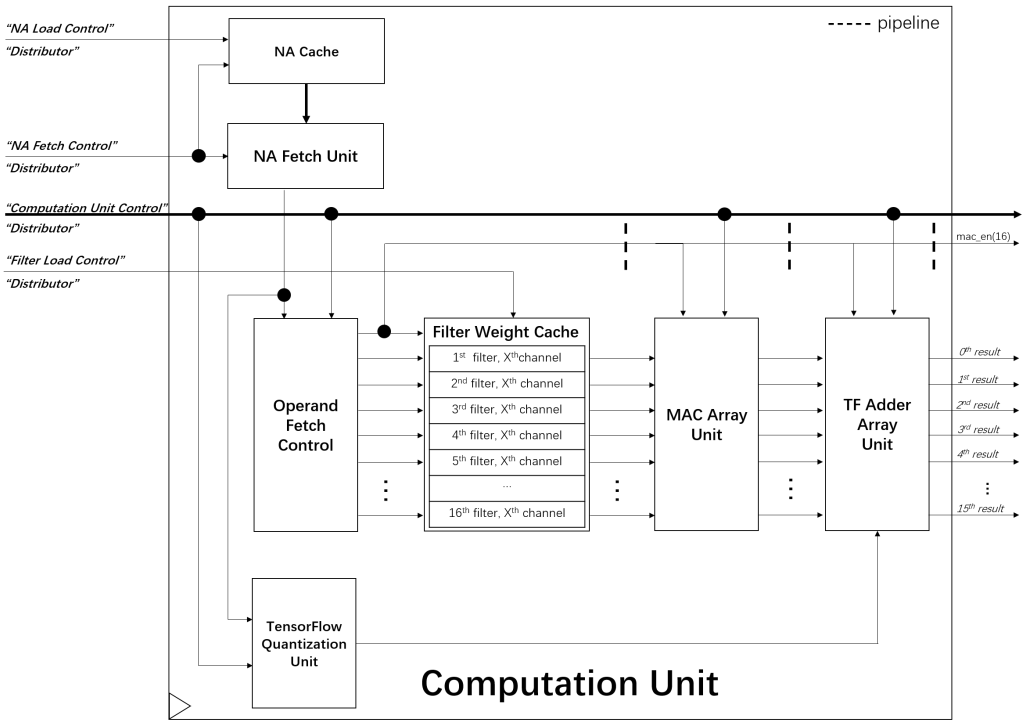

Figure 3.22: Block diagram of **Computation Unit**

output of **Quad Computation Unit** is the sum of the second output of 4 **Computation Unit**s, and so on.

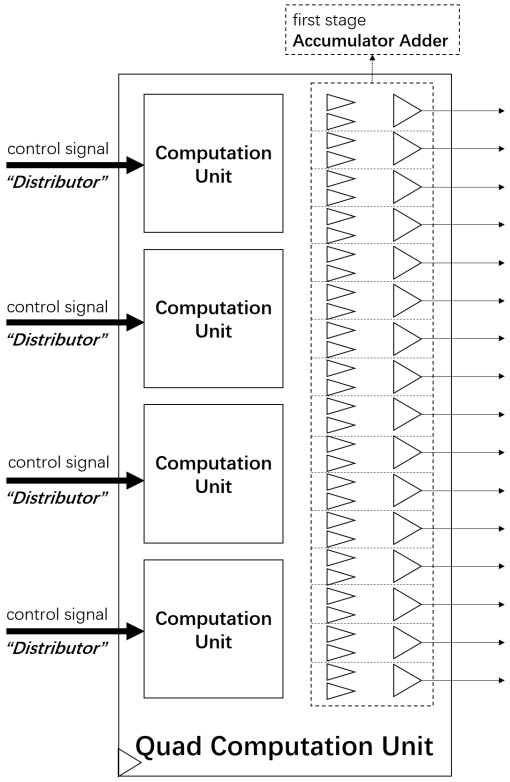

Figure 3.23: Block diagram of **Quad Computation Unit**

The block diagram of **Quad Computation Unit Set** is shown in figure 3.24. Each **Quad Computation Unit** has 16 outputs and bit width of each output is 28 bits. **Quad Computation Unit** Set has also 16 outputs, but bit width is 32 bits. The second stage **Accumulate Adder** sums over the outputs of **Quad Computation Unit**s. As before, the first output of **Quad Computation Unit Set** is the sum of the first output of 4 **Quad Computation Unit**s. The second output of **Quad Computation Unit Set** is the sum of the second output of 4 **Quad Computation Unit**. And so on. Note that the output of **Quad Computation Unit Set** is the output of CNNA and bit width is $16 * 32 = 512$ bits.

3.6.1 Neuron Activation (NA) Cache

The block diagram is shown in figure 3.25.

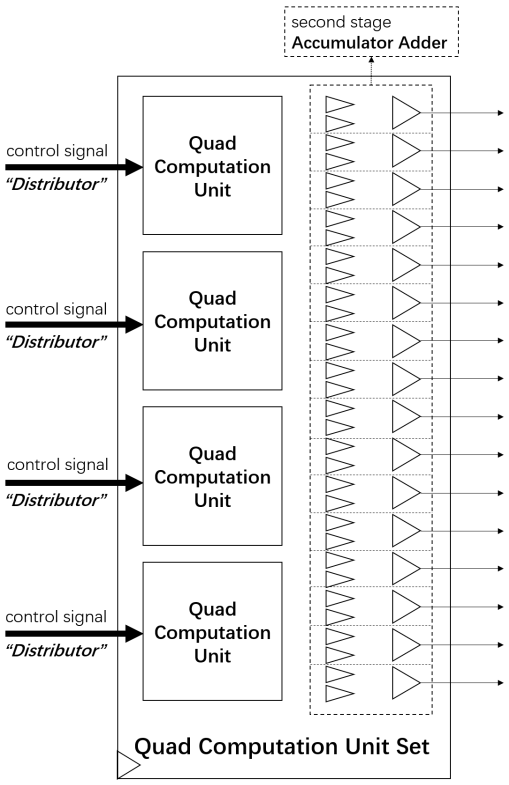

Figure 3.24: Block diagram of **Quad Computation Unit Set**

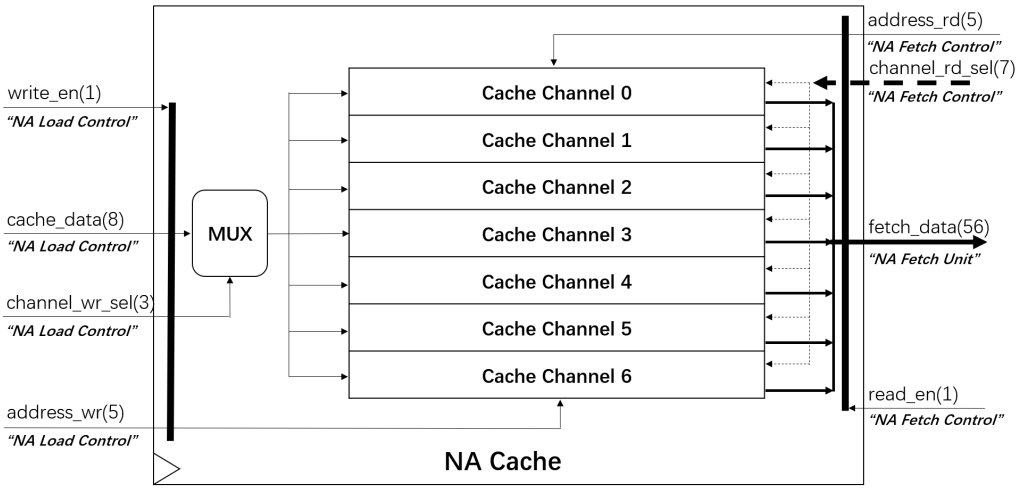

Figure 3.25: Block diagram of **NA Cache**

Each Computation Unit has a dedicated **NA Cache**, which is separated into 7 channels. Each channel's bit width is 8 bits and depth is 32. The second limitation of CNNA to support CNN model is generated here: maximum height of neuron activation is 32. A **NA Cache** occupies $7 * 32 = 224 Bytes$. In CNNA there are totally $224 * 16 = 3584 Bytes = 3.5 Kbytes$ for **NA Cache**.

For writing cache, each time only one byte can be written into one of the channels because the data bus to each **Computation Unit** is 8-bit. Signal "write_en" is the global writing enable signal. Only it is valid, data can be written into the cache. Signal "channel_wr_sel" selects which channel and "address_wr" indexes the write address.

For reading cache, signal "read_en" is the global reading enable signal. Only if it is valid, data can be read out. Unlike writing to the cache, reading from the cache needs another enable signal "channel_rd_sel", which is 7 bits wide and each bit connects to one channel to enable it. If the corresponding bit in "channel_rd_sel" is invalid, the output of the channel is 0x00. The data read out from **NA Cache** is always 56 bits. If we see each channel as row, the same address of the 7 channels are a column. Then a reading operation reads one column of **NA Cache**.

Next, I will explain how to update **NA Cache** during computation in figure 3.27. I assume filter size is [3,3] and size of neuron activation is [5,9]. The filter and NA are shown in figure 3.26. After each stage, **NA Cache** will be updated until the whole neuron activation has been cached into **NA Cache**. The dashed box is where to be update. For **NA Cache**, reading and writing operations are not performed at the same time avoiding conflict. **NA Fetch Control** fetch NA (read **NA Cache**) firstly, then **NA Load Control** updates **NA Cache** (write **NA Cache**).

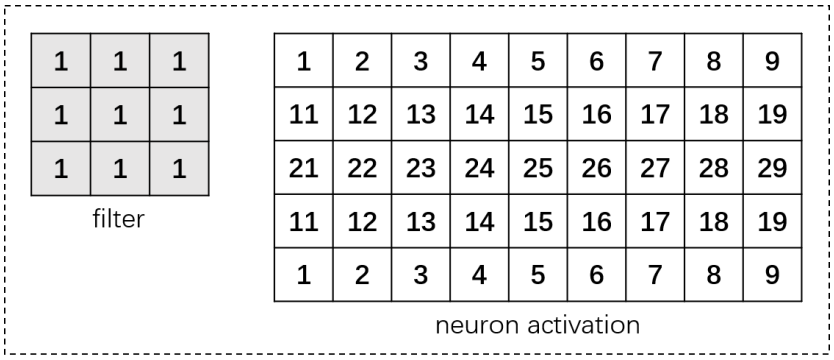

Figure 3.26: Filter and Neuron Activation

3.6.2 NA Fetch Unit

NA Fetch Unit, whose block diagram is shown in figure 3.28, accepts 7 bytes data from **NA Cache**, then rearranges these data (needs for convolution), stores 6 bytes of them (because maximum value of "filter_width" equals 6) and outputs only one byte each time for MAC operation. Although it accepts 7 bytes every time but only "filter_width" of bytes are useful. So, **NA Fetch Unit** needs 1 clock to fetch data from **NA Cache**,but needs "filter_width" times to output data.

3.6.3 Filter Weight Cache

Filter Weight Cache, whose block diagram and signals description are respectively shown in figure 3.29 and table 3.9, is separated into 16 channels. Each channel's depth is 36. The third limitation of CNNA to support CNN model is generated here: maximum size of filter is 36. With the limitation of **NA Fetch Control**, CNNA only support filter size of 1, 4, 9, 16, 25 and 36 for convolutional layer.

For writing **Filter Weight Cache** operation, only one byte (because of bandwidth of **Computation Unit**) can be written into **Filter Weight Cache**. **Filter Weight Cache** is written according the following order, firstly the 1st channel, then the 2nd channel, then the 3rd channel, and so on. In each channel, writing begins from address 0, then is incremented by 1 each time. Writing operation is controlled by **Filter Load Control** in **CNNA Control Unit**.

For reading **Filter Weight Cache** operation, signal "channel_en_rd" is used to enable corresponding channels of **Filter Weight Cache**. The disabled channel's output is 0x00. Reading begins from address 0, then is also incremented by 1 each time. Reading operation is controlled by **Operand Fetch Control**.

3.6.4 Operand Fetch Control

Operand Fetch Control, whose block diagram and signals description are respectively shown in figure 3.30 and table 3.10, has a neuron-activation-zero-judgement inside, which judges whether a neuron activation is zero. If the neuron activation is zero, **Operand Fetch Control** will not fetch filter weight and disable **MAC Array Unit** and **TF Adder Array** Unit to save power. This helps to decrease power consumption, because most of the neuron activations of previous layer are zero[2]. If a neuron activation is non-zero,

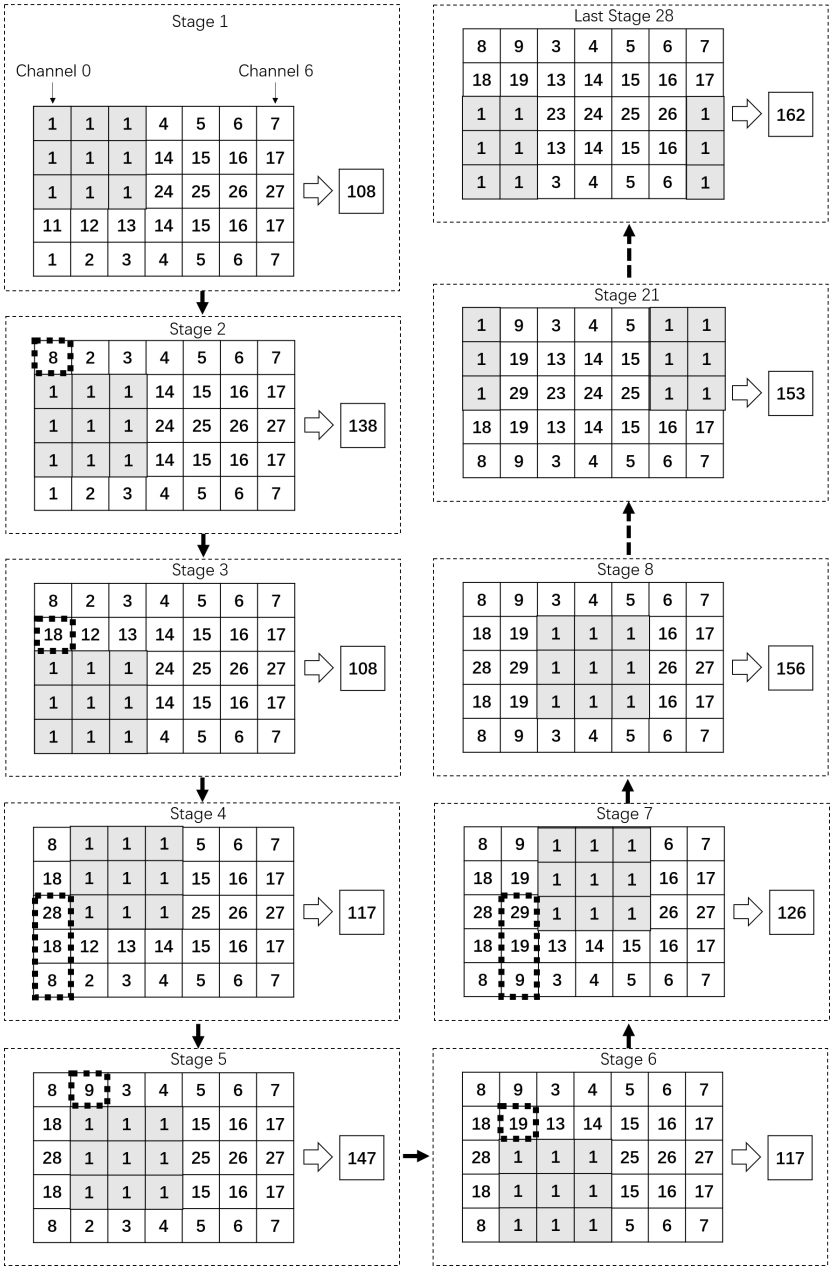

Figure 3.27: How to update **NA Cache**

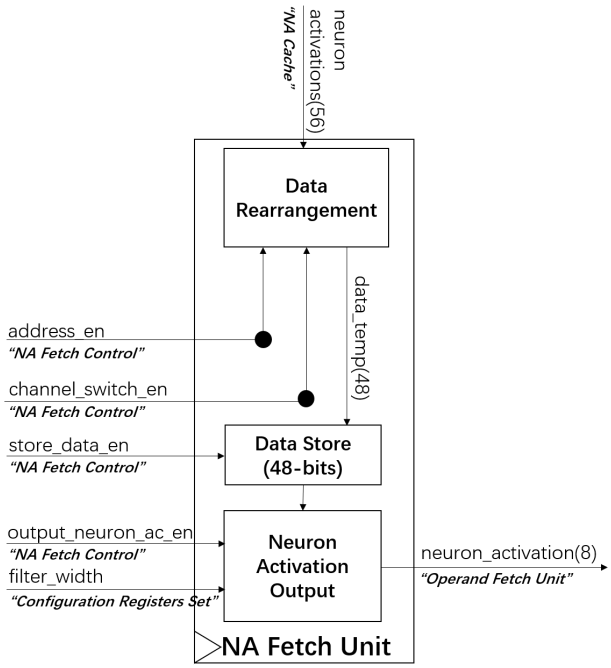

Figure 3.28: Block diagram of **NA Fetch Unit**

Signal	Bit Width	Function
filter_write_en	1	**Filter Weight Cache** write enable
channel_sel	4	Select one channel to write data
address_wr	6	Index the depth of channel
data	8	Data from memory are 128 bits, divided by **Distributor**
channel_en_rd	16	Read enable signals for channels; Indicate which channels are enabled, one bit for each channel
address_rd	6	Index the depth of channel
"signal to MAC Array Unit"	8	Output data from **Filter Weight Cache to MAC Array Unit**

Table 3.9: Signals description of **Filter Weight Cache**

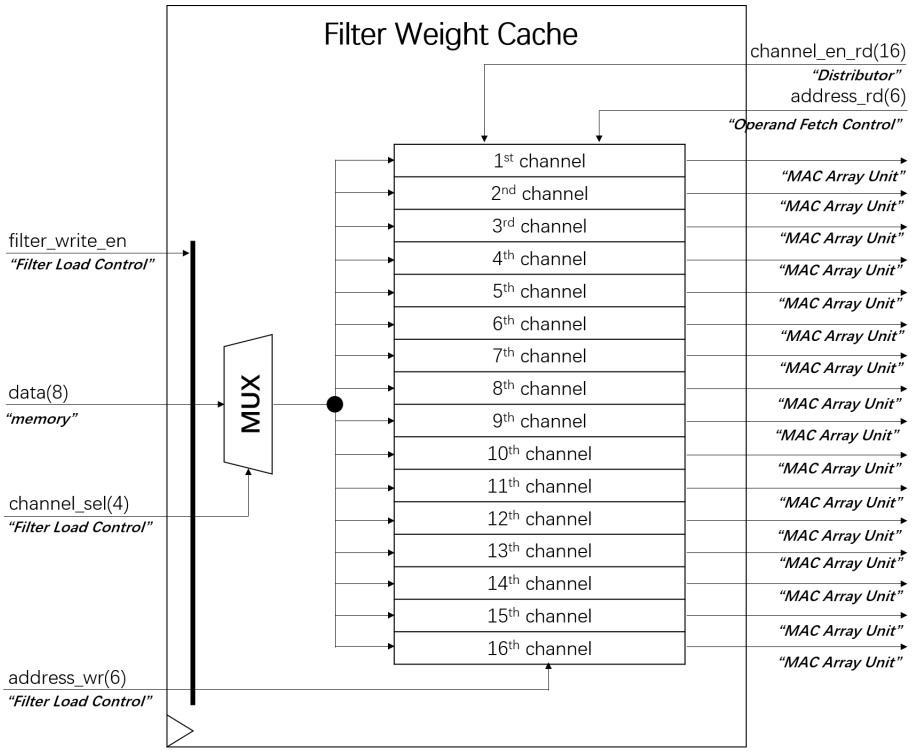

Figure 3.29: Block diagram of **Filter Weight Cache**

Operand Fetch Control will only enable part of **Filter Weight Cache** and **MAC Array Unit** to save power through control signal "mac_en". **Operand Fetch Control** directly controls **Filter Weight Cache** to read filter weights. Note that **Operand Fetch Control** is a pure combinational logic circuit.

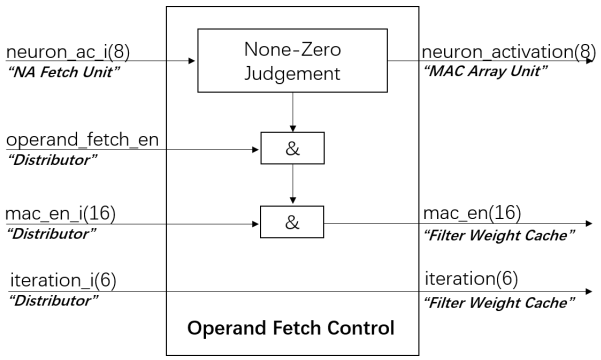

Figure 3.30: Block diagram of **Operand Fetch Control**

Signal	In/Out	Bit Width	Function
neuron_ac_i	In	8	One byte neuron activation from **NA Fetch Unit**
operand_fetch_en	In	1	Enable signal to fetch operands for MAC units
mac_en_i	In	16	16 MAC enable signal from **Distributor**
iteration_i	In	6	Counter during stage from **Distributor**
neuron_activation	Out	8	None-Zero Judgement also serves as buffer for signal "neuron_ac_i"
mac_en	Out	16	Enable signal for 16 MAC units. AND with other signal to save power consumption. Read enable signal for **Filter Weight Cache**.
iteration	Out	6	Address signal for **Filter Weight Cache** during reading cache operation

Table 3.10: Signals description of **Operand Fetch Control**

3.6.5 MAC Array Unit

The block diagrams of **MAC** and **MAC Array Unit** are showed in figure 3.31 and figure 3.32. The signale description is shown in table 3.11. **MAC Array Unit** is reset by internal signal "layer_reset". This part uses a pipeline design to increase speed. The forth limitation of CNNA to support CNN model is generated here: maximum number of filter is 16 as there are only 16 **MAC**s.

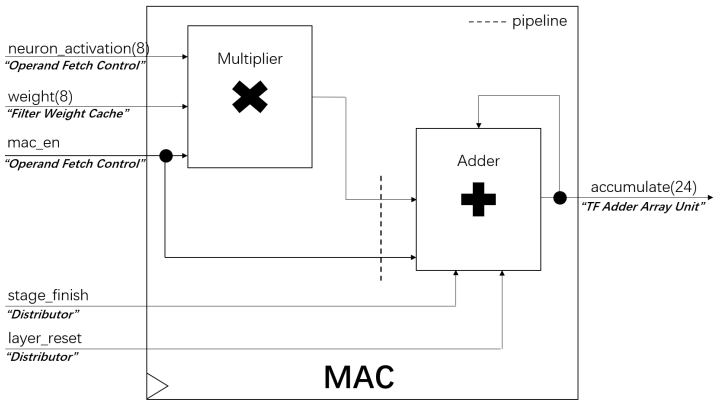

Figure 3.31: Block diagram of **MAC**

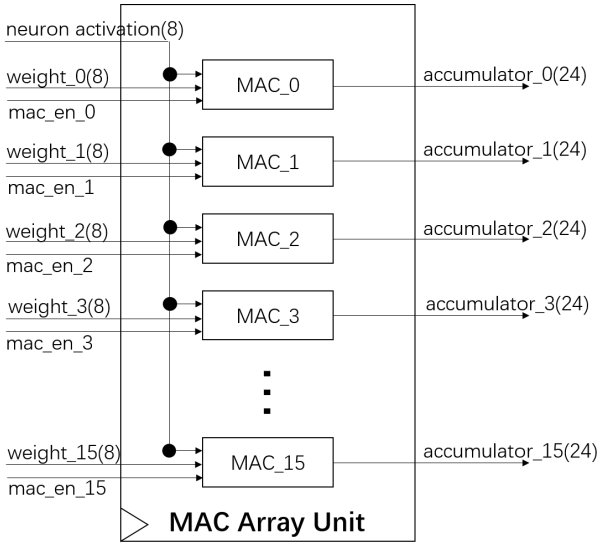

Figure 3.32: Block diagram of **MAC Array Unit**

Signal	In/Out	Bit width	Function
layer_reset	In	1	Reset **MAC Array Unit.**
stage_finish	In	1	Clear result of Adder, set signal "accumulate" to 0, preparing for next stage computation
mac_en	In	1	Enable signal (pipelined signal) for Multiplier and Adder from **Operand Fetch Control**
weight	In	8	One operand for Multiplier.
neuron_activation	In	8	Another operand for Multiplier.
accumulate	In	24	Result of Adder/MAC

Table 3.11: Signals description of **MAC Array Unit**

3.6.6 TensorFlow Quantization Unit

TensorFlow Quantization Unit is realized according equation (2.11) in section 2.5. The block diagram and the description of the signals are respectively shown in figure 3.33 and table 3.12.

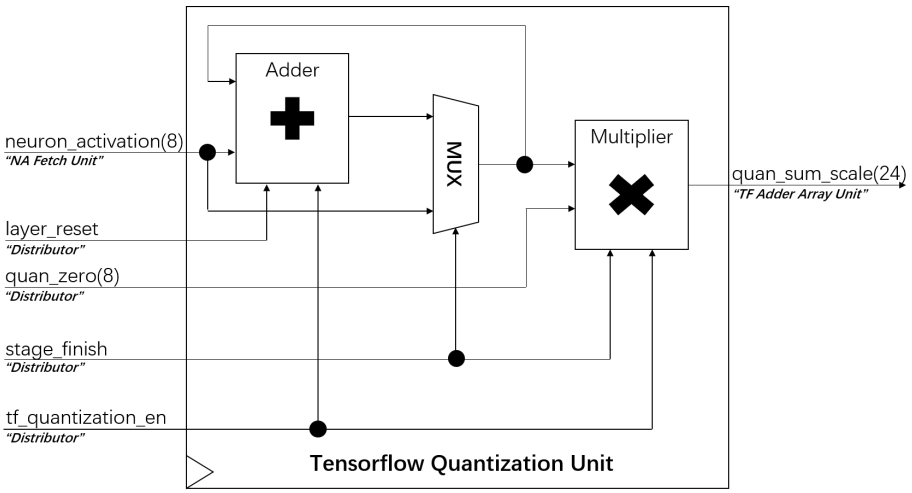

Figure 3.33: Block diagram of **TensorFlow Quantization Unit**

3.6.7 TF Adder Array Unit

TF Adder Array Unit is designed for emulating the TensorFlow quantized model. When signal "tf_quantization_en_pipeline" equals 0, output signal "accumulate_tf" equals input signal "accumulate". When signal "tf_quantization_en_pipeline" equals 1, output signal "accumulate_tf"

Signal	In/Out	Bit Width	Function
layer_reset	In	1	Reset signal for **TensorFlow Quantization Unit**
stage_finish	In	1	Select signal for MUX; Enable signal for Multiplier
tf_quantization_en	In	1	Enable signal for Adder and Multiplier
neuron_activation	In	8	Neuron activation from **NA Fetch Unit**
quan_zero	In	8	Zero quantization value
quan_sum_scale	Out	24	

Table 3.12: Signals description of **TensorFlow Quantization Unit**

equals to input signal "accumulate" minus input signal "quan_sum_scale". The block diagram of **TF Adder** and **TF Adder Array Unit** are shown in figure 3.34 and figure 3.35.

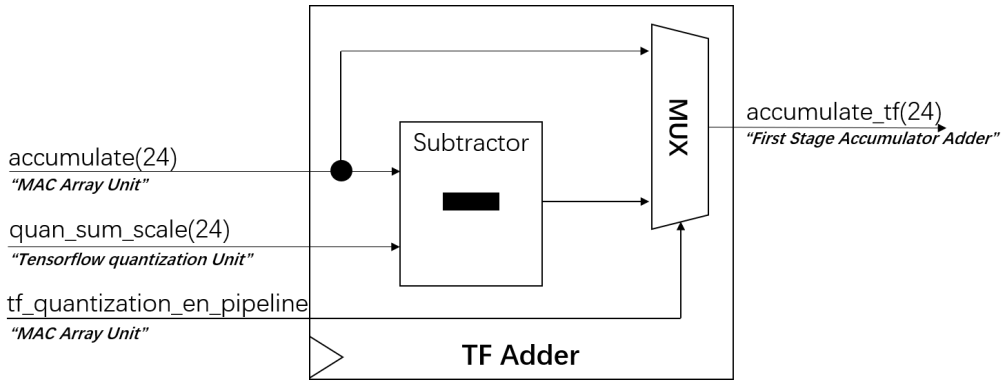

Figure 3.34: Block diagram of **TF Adder**

3.6.8 First Stage Accumulate Adder Unit

First Stage Accumulate Adder Unit locates in **Quad Computation Unit**. There are 16 **Accumulate Adder**s in **First Stage Accumulate Adder Unit**. The detail about this please see section 3.5. In this subsection, the active signals for 16 **Accumulate Adder**s are the core part. The initial idea of applying active signals was to use it to reduce power consumption when CNNA is not fully utilized. However, given that the neural network models are

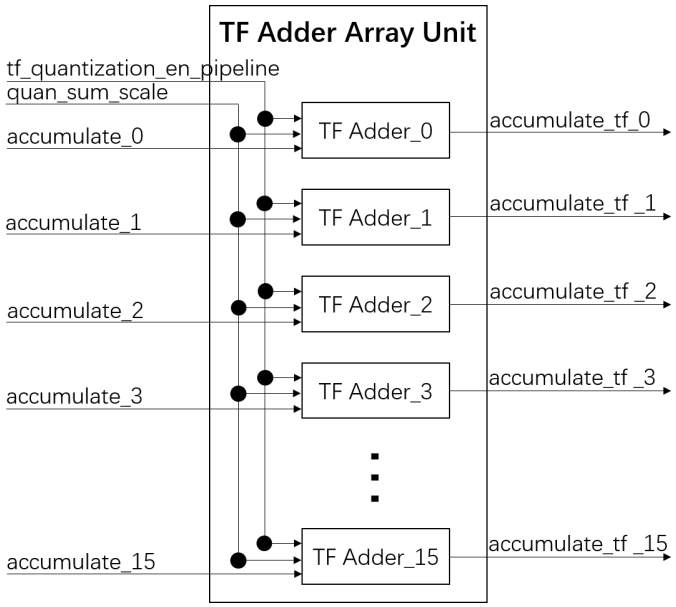

Figure 3.35: Block diagram of **TF Adder Array Unit**

relatively large, CNNA is usually under fully utilization. Therefore, the power consumption should not be noticeably reduced in pratical. As mentioned earlier, the i^{th} output of **Quad Computation Unit** is the sum of the i^{th} output of 4 **Computation Unit**s. Besides that, **Computation Unit**s are continuously activated. Using these two conditions, we can generate the active signals. But firstly I will explain what means continuously activated. It means there are only four kinds of situations for how **Computation Unit**s are activated. They are shown in figure 3.36. Note that only gray cell indicates the activated **Computation Unit**.

As you can see from figure 3.37, every **Accumulate Adder** has 3 adders inside. Only if all of **Computation Unit**s are active, then the 3 adders can be fully utilized. For other situation, we can only enable none or 1 or 2 of them to reduce power consumption. Specifically, for the first situation in figure 3.36, no adder is activated and the result of the first **Computation Unit** directly propagates to the output of **Quad Computation Unit**. There are respectively 1 and 2 activated adders for situation 2 and 3. Beside these active signal, 16 **Accumulate Adder**s are enabled by signal "mac_en", which can also save power.

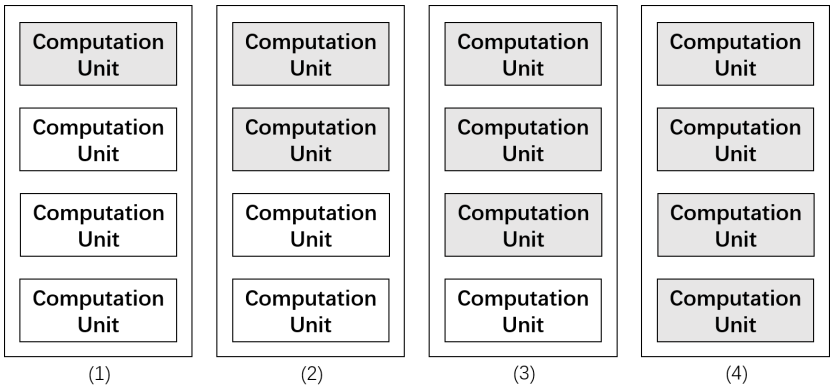

Figure 3.36: Situations of How **Computation Unit**s are activated

3.6.9 Second Stage Accumulator Adder Unit

Second Stage Accumulator Adder Unit locates in **Quad Computation Unit Set**. Similar to **First Stage Accumulator Adder Unit**,

- there are 16 **Accumulator Adder**s inside.
- the i^{th} output of **Quad Computation Unit Set** is the sum of the i^{th} output of 4 **Quad Computation Unit**s.

Like **First Stage Accumulator Adder Unit**, there are also four kinds of situations of how **Quad Computation Unit**s are activated. They are shown in figure 3.38. Note that gray cell indicates the activated **Quad Computation Unit**. As long as one of the 4 **Computation Unit**s inside a **Quad Computation Unit** is active, the **Quad Computation Unit** is seen as active.

Similar as **First Stage Accumulator Adder Unit**, each **Accumulate Adder** in second stage has also 3 adders inside. For each situation shown in previous figure, different adders are activated. There are respectively none, 1, 2 and 3 activated adders for situation 1, 2, 3 and 4.

The block diagram of **Second Stage Accumulate Adder** is shown figure 3.39.

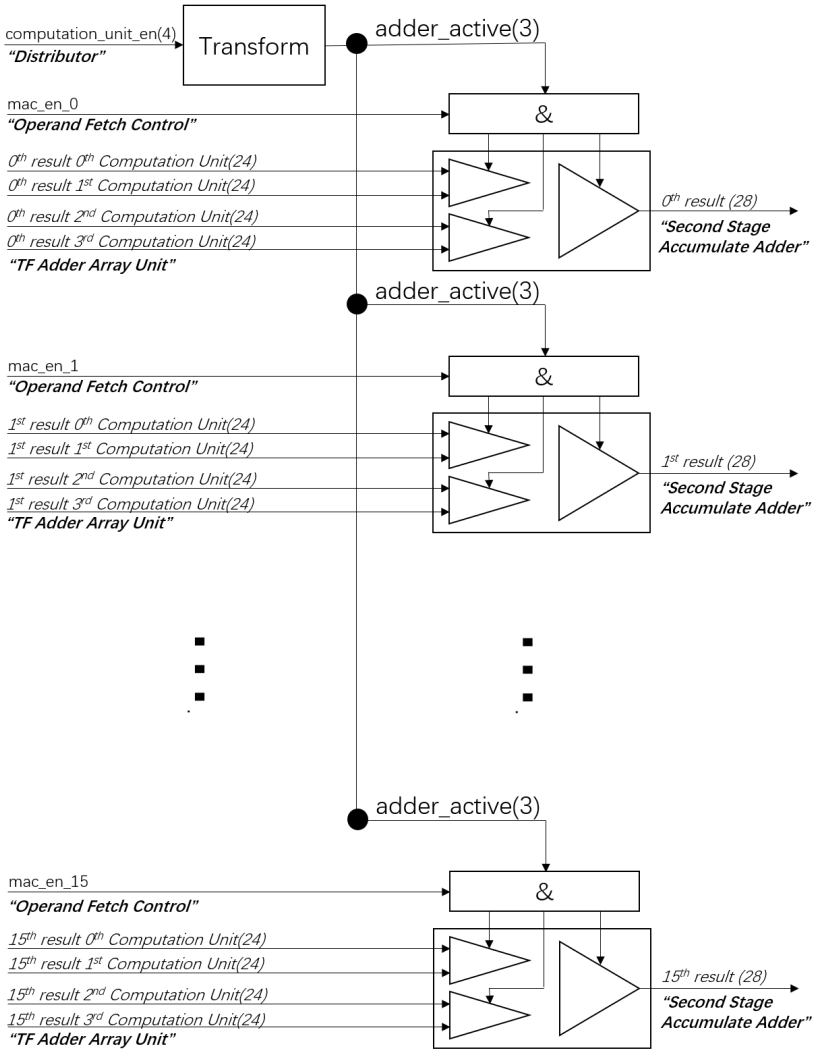

Figure 3.37: Block diagram of **First Stage Accumulate Adder**

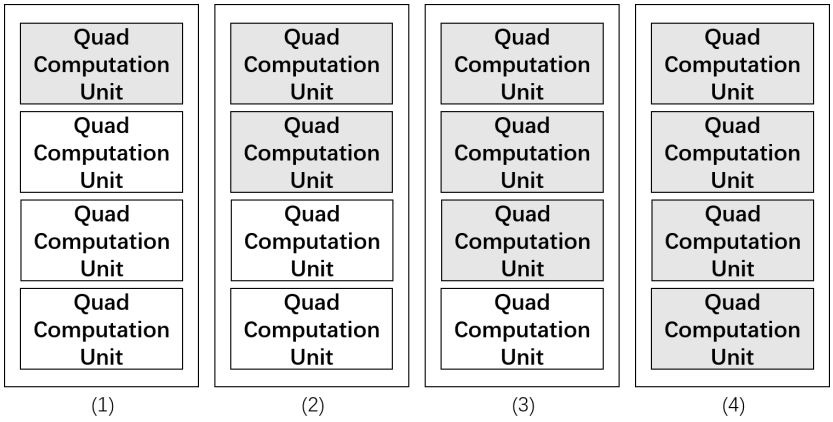

Figure 3.38: 4 activated **Quad Computation Unit**s situation

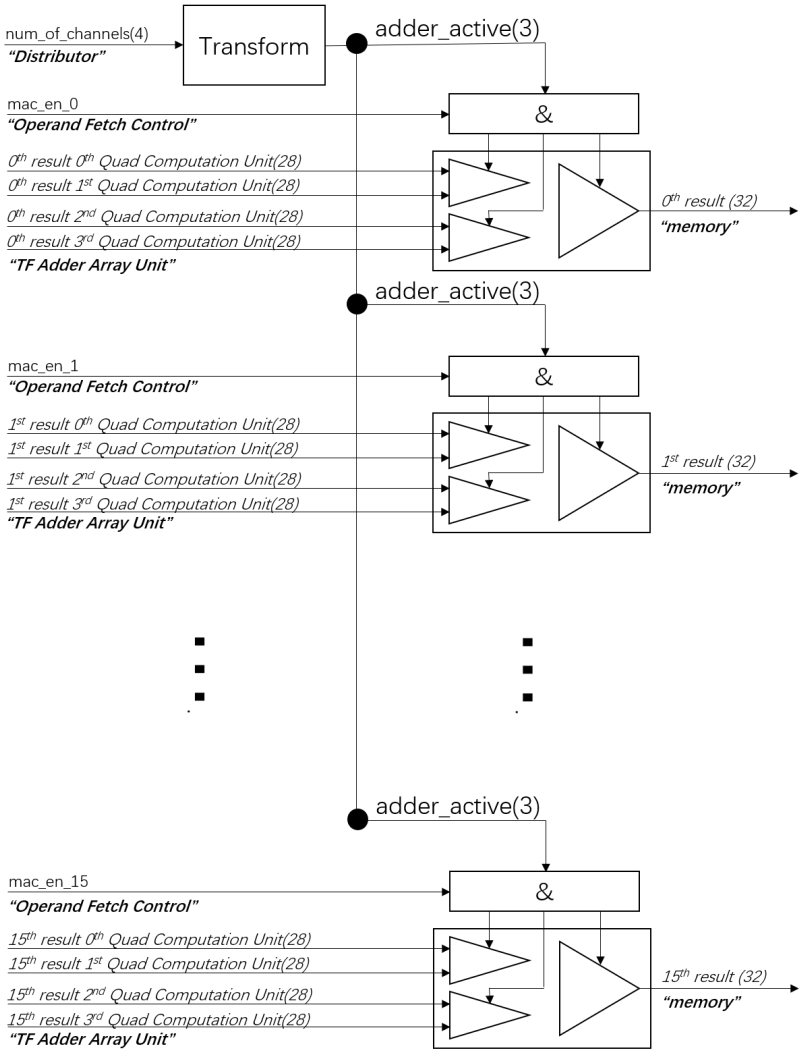

Figure 3.39: Block diagram of **Second Stage Accumulate Adder**

4 Validation and Experiment

In this chapter, we will illustrate the verification environment and how to map convolutional layer and fully-connected layer into CNNA in order to verify CNNA's functionality. Beside these, performance testing will also be carried out.

4.1 Validation Environment

The architecture, which is dedicated to verify CNNA, is shown in Figure 4.1. As CNNA's bandwidth of input (128-bit) and output (512 bit) are different, two SRAM, **SRAM-I** (128-bit bandwidth) and **SRAM-O** (512-bit bandwidth) are employed during validation. Signals "cnna_begin" and "cnna_end" are connected to CNNA's pins "cpu_to_cnna" and "cnna_to_cpu" respectively. **UDP Controller** performs 3 tasks according the command (special packet/magic packet) from Validation Program: storing data into **SRAM-I**, waking up CNNA and reading data from **SRAM-O**. When the calculation of CNNA is finished (signal "cnna_end" becomes active), **UDP Controller** will send a magic packet to inform PC automatically. The communication through FPGA and PC uses the UDP protocol.

Validation Platform Introduction:

- Operation System: Ubuntu 16.04
- PC Hardware Configuration: Intel® Core™ i7-4720HQ, 16GB RAM, NVIDIA GeForce GTX 860M (4GB RAM)
- Version of Tensorflow: R1.2 with GPU
- Version of Python: 3.6
- FPGA Platform: Xilinx Spantan-6

CNNA is synthesized with clock of 125 MHz. The summary power report of synthesis by Vivado is showed in table 4.1. The resource utilization report of synthesis by Vivado is in Appendix A. Note that the reports include

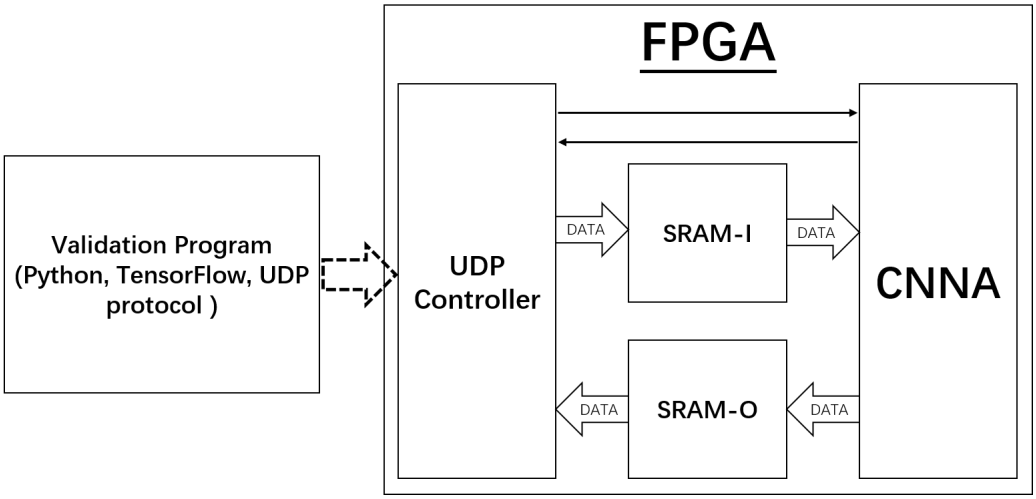

Figure 4.1: Architecture of Validation Enviourment

the modules **SRAM-I** (Depth: 256), **SRAM-O** (Depth: 256) and **UDP Controller**.

Item	Power(W)
Total On-Chip Power	0.614
Dynamic	0.521
Device Static	0.094

Table 4.1: Power Report

4.2 Convolutional Layer Supporting

For validating CNNA's convolutional layer functionality, method *tf.nn.conv2d*[15] from TensorFlow is employed. Note that the input channels of neuron activation and filter must be same. Beside this, *filter_wieght* and *filter_height* must also be same. Figure 4.2 is the mapping of CNNA into convolutional layer. Note that NA is an abbreviation for neuronal activation, AA is an abbreviation for **Accumulate Adder**, **TF Adder** is included inside of **MAC unit** for simplicity. Please refer to section 3.6 **Quad Computation Unit Set** for better understanding of the mapping.

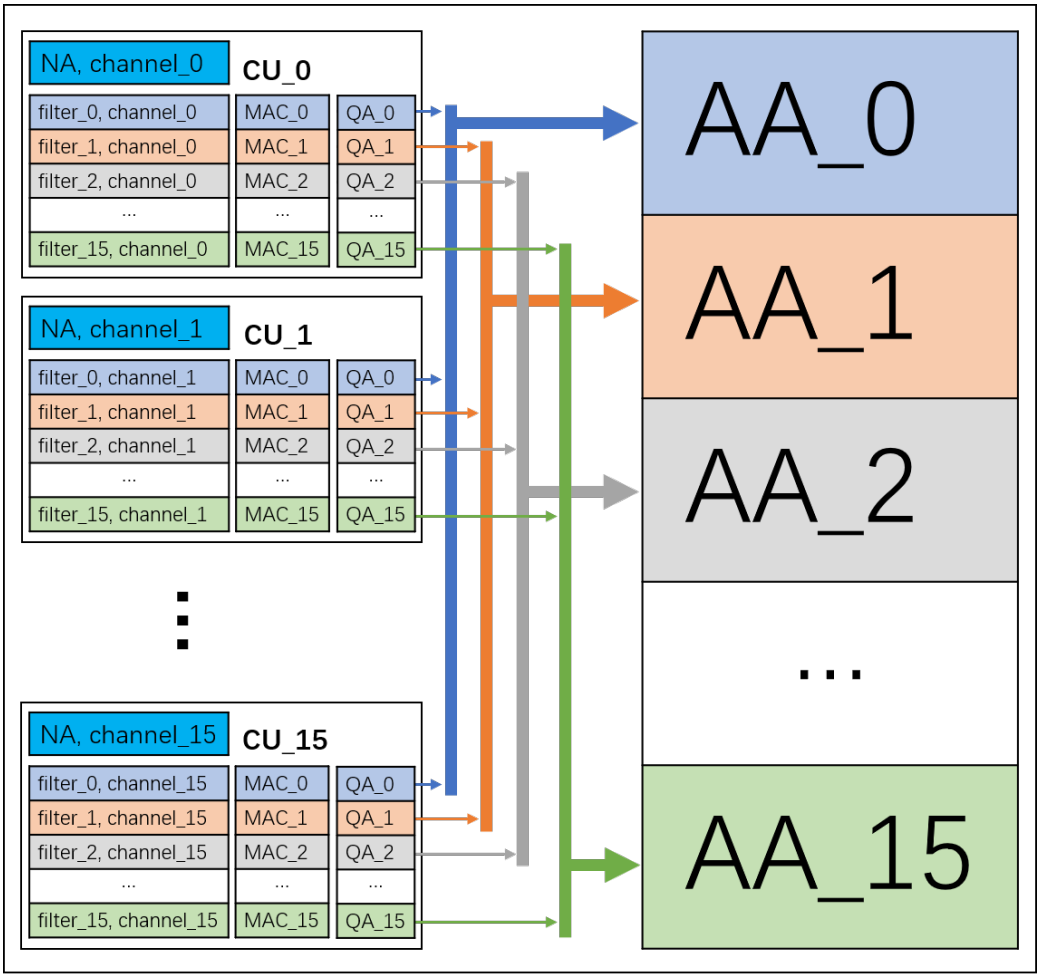

Figure 4.2: Mapping of CNNA into Convolutional Layer

4.2.1 Configuration Information Generation

From section 3.4.2 **FSM Control Logic**, we know that CNNA will read the configuration information from address 0x00 and 0x01 immediately after waking up. Therefore, we need to compute the configuration information firstly. Table 3.5 in section 3.4.1 **Configuration Register Set** already apply enough information for that. CNNA support both unquantizated and quantizated convolutional layer. The only different configurations between them are these two configuration information: "platform_support" and "quan_weight_zero".

4.2.2 Neuron Activation Mapping

After generating the configuration information, we need to consider the neuron activation. As the data in Tensorflow is multidimensional (4-dimension), but memory arrangment is one-dimensional, therefore data need to be transfered to be fit for memory arrangment. Given that NA's dimension is [*NA_batch, NA_height, NA_width, NA_channels*], according the way how CNNA fetching NA from **SRAM-I** thourgh **NA Cache** to **MAC Unit**, *Algorithm 1* is dedicate to rearrange NA data into one-dimention.

Algorithm 1 Transfer Algorithm for Neuron Activation

Input: neuron activation: NA[NA_batch, NA_height, NA_width, NA_channels]; memory: MEM[X];
Output: memory: MEM[X];

```
for i ∈ [1,NA_batch] do
    for j ∈ [1,NA_width] do
        for k ∈ [1,NA_height] do
            index_h = i * j * k * NA_channels
            index_l = index_h − NA_channels
            reverse_data = NA[i, k, j, :].reverse()  ▷ Method reverse() is used to solve endianness problem.
            MEM[index_l : index_h] = reverse_data;
        end for
    end for
end for
```

4.2.3 Filter Mapping

For filter weight, a similar transfer process is needed. Given that filter's dimension is [filter_height, filter_width, filter_in_channels, filter_out_channels], according the way how CNNA fetches filter thorugh **Filter Weight Cache** to **MAC Unit**, *Algorithm 2* is dedicate to rearrange filter data into one-dimention.

Algorithm 2 Transfer Algorithm for Filter Weight

Input: filter: FW[filter_height, filter_width, filter_in_channels, filter_out_channels]; memory: MEM[Y];
Output: memory: MEM[Y];

```
for i ∈ [1,filter_out_channels] do
    for j ∈ [1,filter_height] do
        for k ∈ [1,filter_width] do
            index_h = i * j * k * filter_in_channels
            index_l = index_h − input_channels
            reverse_data = FW[j, k, :, i].reverse() ▷ Method reverse() is used to solve endianness problem.
            MEM[index_l : index_h] = reverse_data;
        end for
    end for
end for
```

4.2.4 Verifidation Process

In order to verify CNNA, the Validation Program (**VP**) firstly runs an arbitrary CNN model in TensoFlow or restores a quantized model to generate a reference and get the computation node result, then gererate corresponding configuration information for CNNA according section 4.2.1. After converting the data from TensoFlow computation node into one dimensional data using *Algorithm 1* in section 4.2.2 and *Algorithm 2* in section 4.2.3, **VP** transmits the configuraion and converted data to **SRAM-I**. Next, CNNA is woken up by **UDP Controller** through a magic packet from **VP** to accelerate convolutional layer computation. After computation, CNNA will active the signal "cnna_end" and **UDP Controller** will send a magic packet to **UDP Controller** informing that CNNA has finished computation. Lastly, **VP** will read the data from **SRAM-O** through **UDP Controller** and convert

the data to compare with the result from TensoFlow. The whole validation process is shown in figure 4.3.

4.2.5 Performance Test

In this section, I will give the benchmark of CNNA versus different parameter, including width of filter, height of neuron activation, width of neuron activation, input channels and output channels. Note that the performance test result is the same for both quantized and unquantized model, as **TF Adder Array Unit** (section 3.6.7) will run no matter what kind of model. The performance test mainly measures the excution clocks and the correponding number of operations versus different parameter. The exuction time is seperate into 3 part: configuration, prefetch and computation. The configuration time is fixed, 6 clocks, which corresponds to the first 6 states in FSM. The prefetch time corresponds to 6th, 7th, 8th, 9th state in FSM. The computation time represent the rest of states. The operation includes multiplication and addition, whose number are almost the same.

(i) Width of Filter

As the limitation of CNNA, width of filter equals height of filter, only width of filter will be discussed in this section. Figure 4.4 is the benchmark of excution clocks and number of operations versus filter width. The parameters are defined as following for the performance test: [*NA_batch, NA_height, NA_width, NA_channels*]=[*1, 28, 28, 5*], [*filter_height, filter_width, filter_in_channels, filter_out_channels*]=[*X, X, 5, 10*] (X is a variable). From the figure, we can see, in this performance test, the clocks used in computation takes the most and the accelerator factor is around 100 times. Beside this, the number of operation increases in square rate, as $filter\_size = filter\_width * filter\_height = filter\_width^2$.

(ii) Height of Neuron Activation

Figure 4.5 is the benchmark of excution clocks and number of operations versus neuron actiavtion height. The parameters are defined as following for the performance test: [*NA_batch, NA_height, NA_width, NA_channels*]=[*1, X, 28, 5*] (X is a variable), [*filter_height, filter_width, filter_in_channels, filter_out_channels*]=[*3, 3, 5, 10*]. The accelerator factor is also around 100 times. The computation phase takes still the most of time.

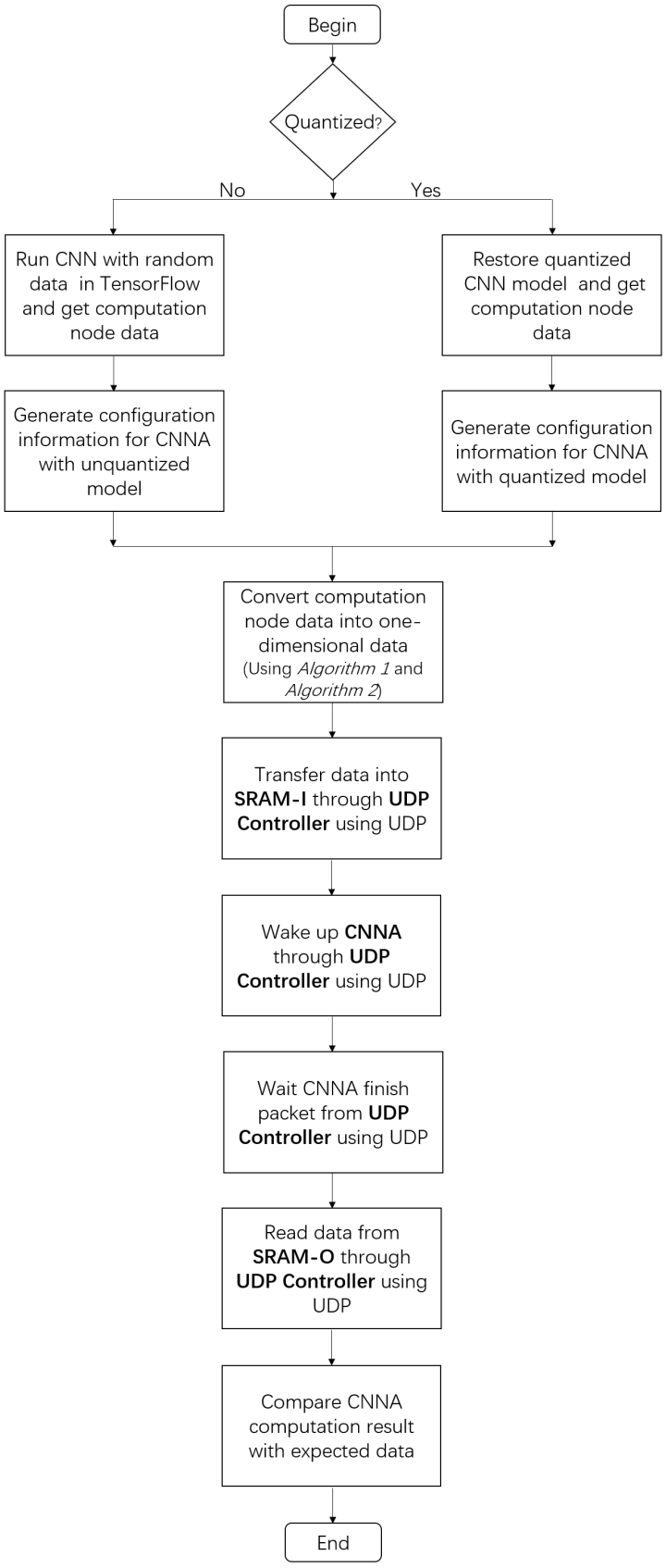

Figure 4.3: Verification Process

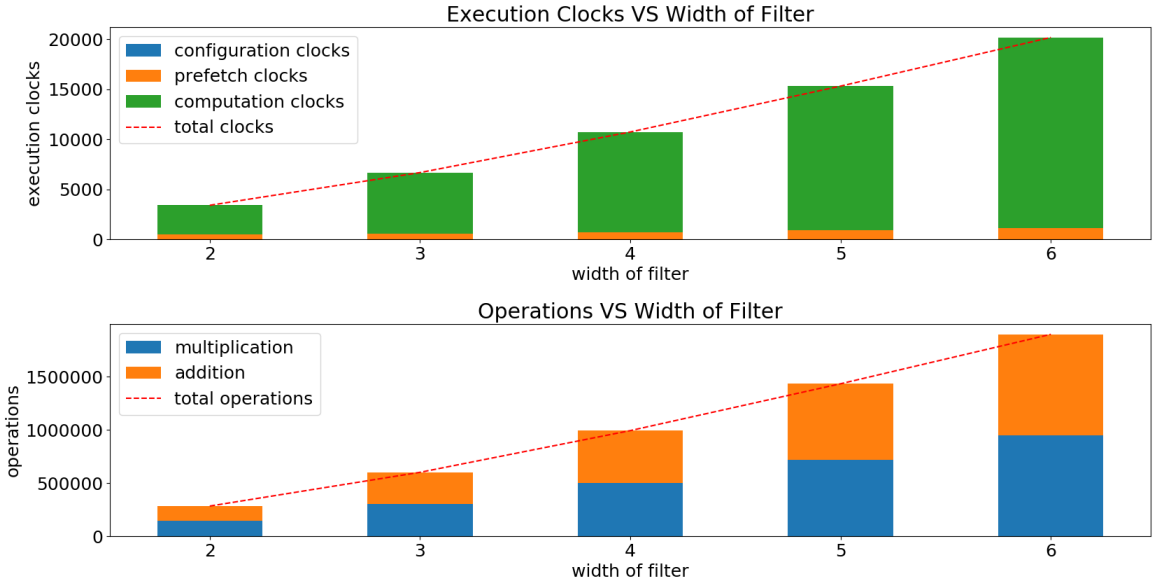

Figure 4.4: Benchmark versus Filter Width

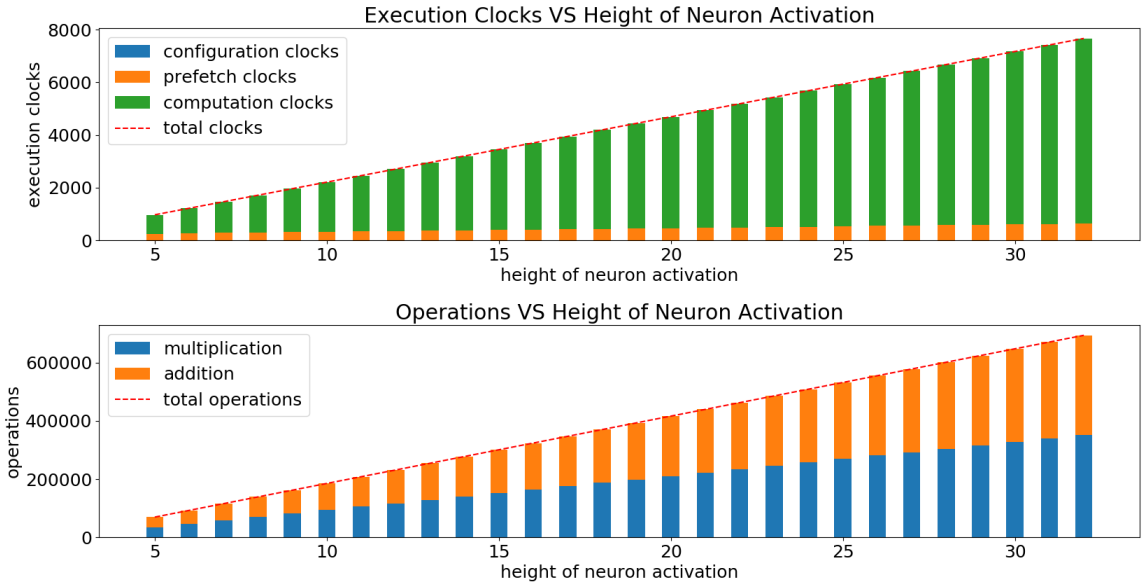

Figure 4.5: Benchmark versus Neuron Activation Height

(iii) Width of Neuron Activation

Figure 4.6 is the benchmark of excution clocks and number of operations versus neuron actiavtion width. The parameters are defined as following for the performance test: [*NA_batch, NA_height, NA_width, NA_channels*]=[*1, 28, X, 5*], [*filter_height, filter_width, filter_in_channels, filter_out_channels*]=[*3, 3, 5, 10*]. The accelerator factor is also around 100 times and the computation phase still takes the most of time.

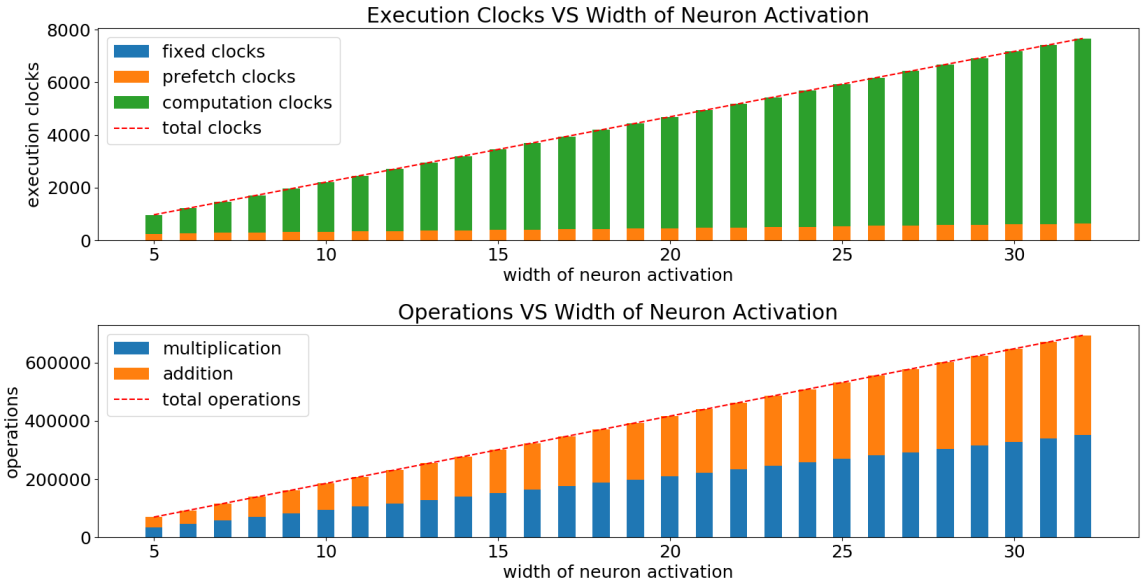

Figure 4.6: Benchmark versus Neuron Activation Width

(iv) Input Channels

Figure 4.7 is the benchmark of excution clocks and number of operations versus number of input channel. The parameters are defined as following for the performance test: [*NA_batch, NA_height, NA_width, NA_channels*]=[*1, 28, 28, X*], [*filter_height, filter_width, filter_in_channels, filter_out_channels*]=[*3, 3, X, 10*]. The accelerator factor is up to around 300 times and the computation phase still takes the most of time. Beside this, the excution time stays the same versus numbers of input channels, as input channels affect the memory bandwidth utilization but not the time. Therefore, it is encouraged to fully utilize the bandwidth/input channels in order to fully utilize the

computation resource (MAC units).

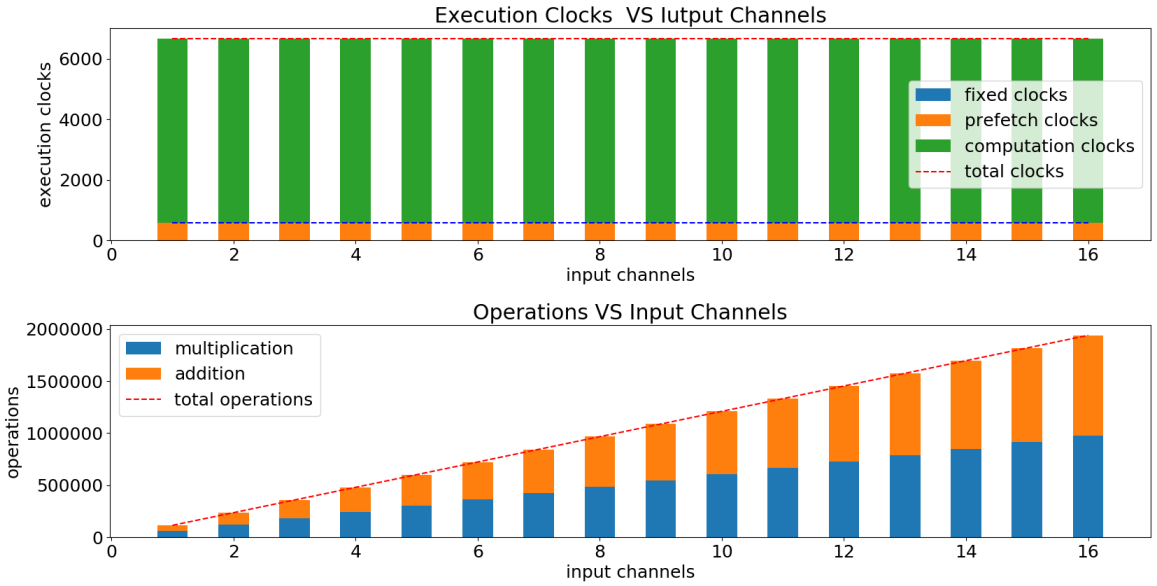

Figure 4.7: Benchmark versus Input Channel

(v) Output Channels

Figure 4.8 is the benchmark of excution clocks and number of operations versus number of output channel. The parameters are defined as following for the performance test: [*NA_batch, NA_height, NA_width, NA_channels*]=[*1, 28, 28, 5*], [*filter_height, filter_width, filter_in_channels, filter_out_channels*]=[*3, 3, 5, X*]. The accelerator factor is up to around 150 times and the computation phase still takes the most of time. Beside this, the computation time stays the same and the prefetch time increase slowly versus numbers of output channels. This is because different output channels affects the time of fetching filter weight into cache. The table 3.5 in section 3.4.1 gives the most intuitive reason for this.

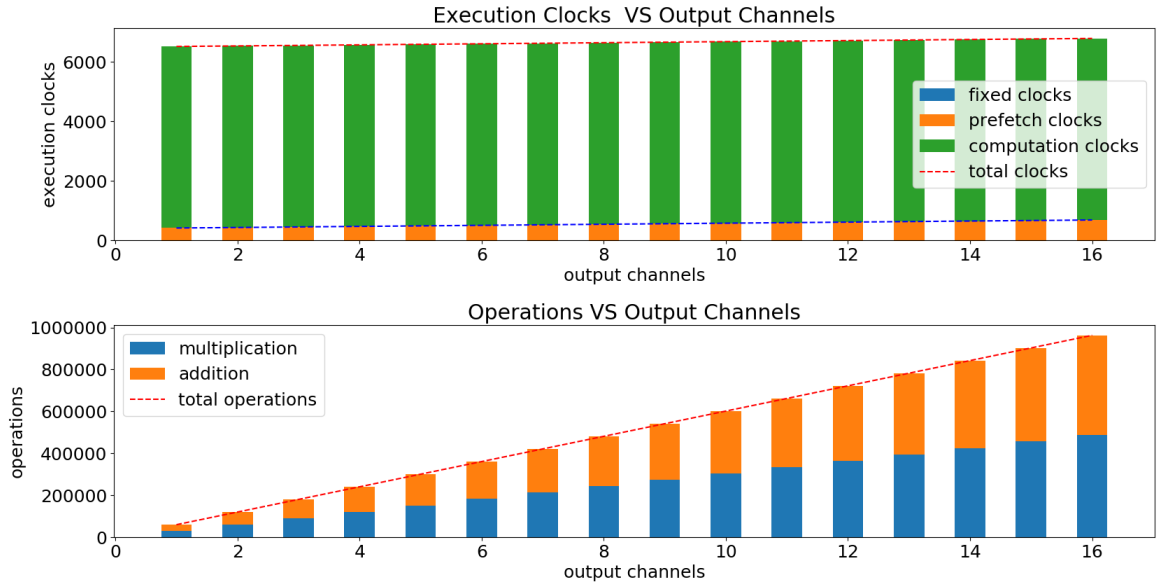

Figure 4.8: Benchmark versus Output Channel

4.3 Fully-Connected Layer Supporting

FC is a special case of convolutional layer. When the dimension of filter is the same as that of neuron activation, a convolutional layer becomes a FC. Method *tf.matmul*[16] from TensorFlow is employed to validate the FC functionality. As a distinction, name **a** and **b** are used instead of neuron activation and filter. As **a** and **b** in FC are vector and two-dimensional-matrix. To adapt CNNA for accelerating FC, parametric dimension mismatches need to be solved. Therefore, we divide **a** and each column of **b** into X parts with same size S. Moreover, each part is converted into submatrix with size of $[S\_r, S\_c]$. Lastly, we convert **a** and **b** into matrixs with wanted dimension. The parameter mapping relation of FC to CNNA is shown below.

$$\mathbf{a} = [1, X, S\_r, S\_c] = [NA\_batch, NA\_channels, NA\_height, NA\_width] \tag{4.1}$$

$$\begin{aligned}\mathbf{b} &= [R, X, S\_r, S\_c] \\ &= [filter\_out\_chanels, filter\_in\_chanels, filter\_height, filter\_width]\end{aligned} \tag{4.2}$$

With help of equation 4.1 and 4.2, the mapping of CNNA into FC for $X = 16$ shown in figure 4.9. For a better understanding, figure 4.10 shows how to map FC into CNNA.

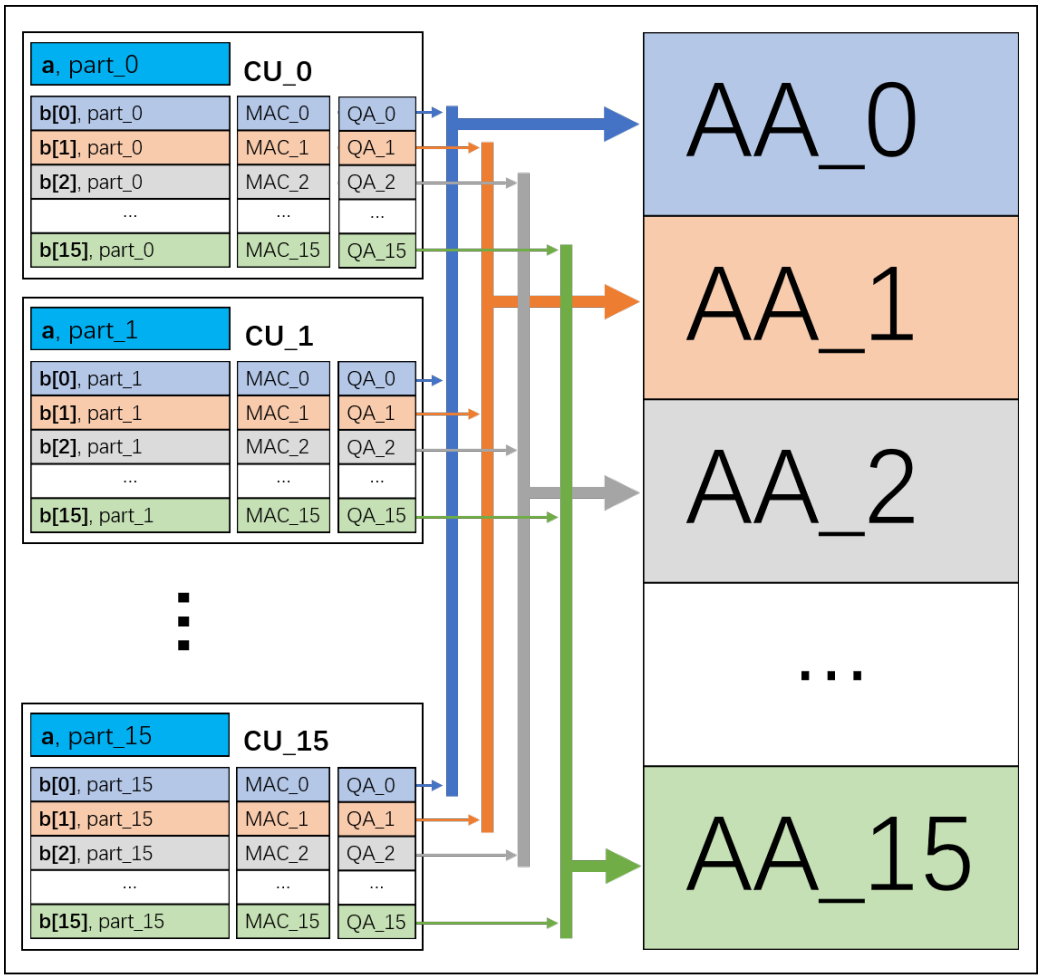

Figure 4.9: Mapping of CNNA into FC

After applying conversion, data mapping and validation process of FC are the same as those of a convolutional layer. There is only a little difference of configuration information generation between them, which is located in "platform_support" (see section 3.4.1).

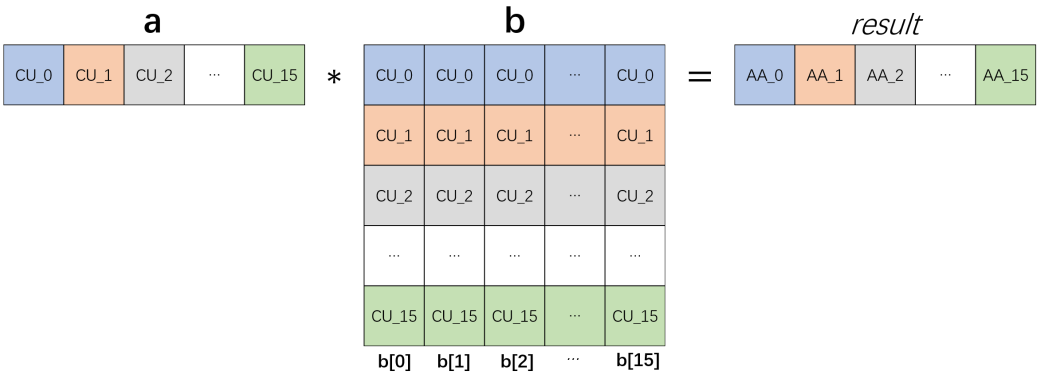

Figure 4.10: Mapping of FC into CNNA

4.3.1 Performance Test

In this section, we will give the benchmark of CNNA for FC versus different parameter, including parts (X), size of **a** ($16 * S\_r * S\_c$) and columns of **b** (R). The performance test mainly measures the excution clocks and the correponding number of operaions versus different parameter. The exuction time is seperate into 4 part: configuration, prefetch of **a**, prefetch of **b** and computation time. The configuration time is fixed, 6 clocks, which corresponds to the first 6 states in FSM. The prefetch time of **a** corresponds to 8th, 9th state in FSM. The prefetch time of **b** corresponds to 6th, 7th. The rest of states are the computation time. The operation includes multiplication and addition, whose number are almost the same.

(i) Size of a

In this performance test, the dimensions of **a** and **b** are defined as below.

$$\mathbf{a} = [1, X, S\_r, S\_c] = [1, 8, S\_r, S\_c]$$

$$\mathbf{b} = [R, X, S\_r, S\_c] = [8, 8, S\_r, S\_c]$$

The size of **a** equals $8 * S\_r * S\_c$. From the result in figure 4.11, it is easy to find that the prefetch time of **b** takes the most time and the computation time takes only a little part, which is severely different from the result for conlutional layer. The accelerate factor is around 7, much lower as for convolutional layer. The result indicates that memory bandwidth is much more important for FC than for convolutional layer.

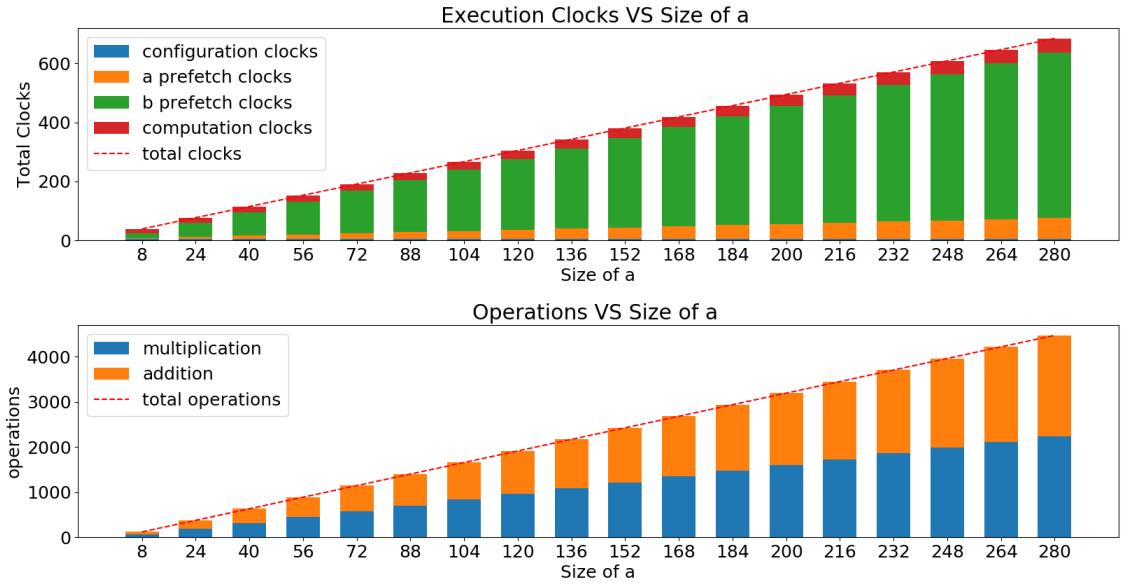

Figure 4.11: Benchmark versus Size of **a**

(ii) Columns of b

In this performance test, the dimensions of **a** and **b** are defined as below.

$$\mathbf{a} = [1, X, S\_r, S\_c] = [1, 8, 6, 6]$$

$$\mathbf{b} = [R, X, S\_r, S\_c] = [R, 8, 6, 6]$$

The prefetch time of **b** still takes the most time and the accelerate factor is till around 7. The benchmarch result is shown in figure 4.12

(iii) Parts

In this performance test, the dimensions of **a** and **b** are defined as below.

$$\mathbf{a} = [1, X, S\_r, S\_c] = [1, X, 6, 6]$$

$$\mathbf{b} = [R, X, S\_r, S\_c] = [8, X, 6, 6]$$

The benchmarch result is shown in figure 4.13. Same as before, the prefetch time of **b** still takes the most time. However, with the increase in X and the amount of calculation, the operation time has not increased. Finally, it reaches the peak performance of accelerate factor of around 13. The result

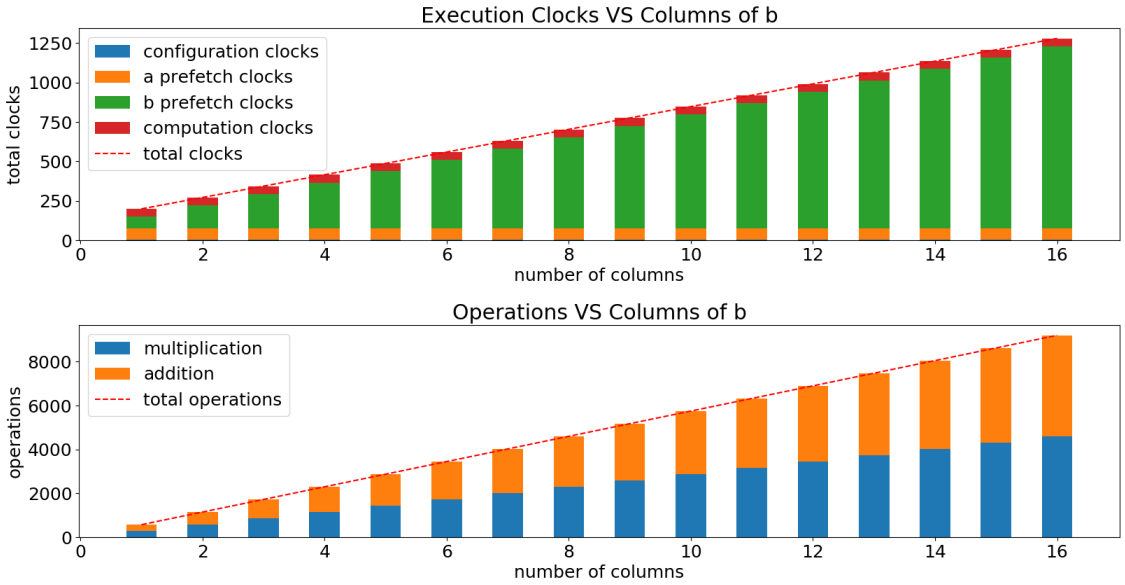

Figure 4.12: Benchmark versus Columns of the **b**

suggests that we should divide **a** and each column of **b** into multiple parts as much as possible, as it uses as many computation resources as possible.

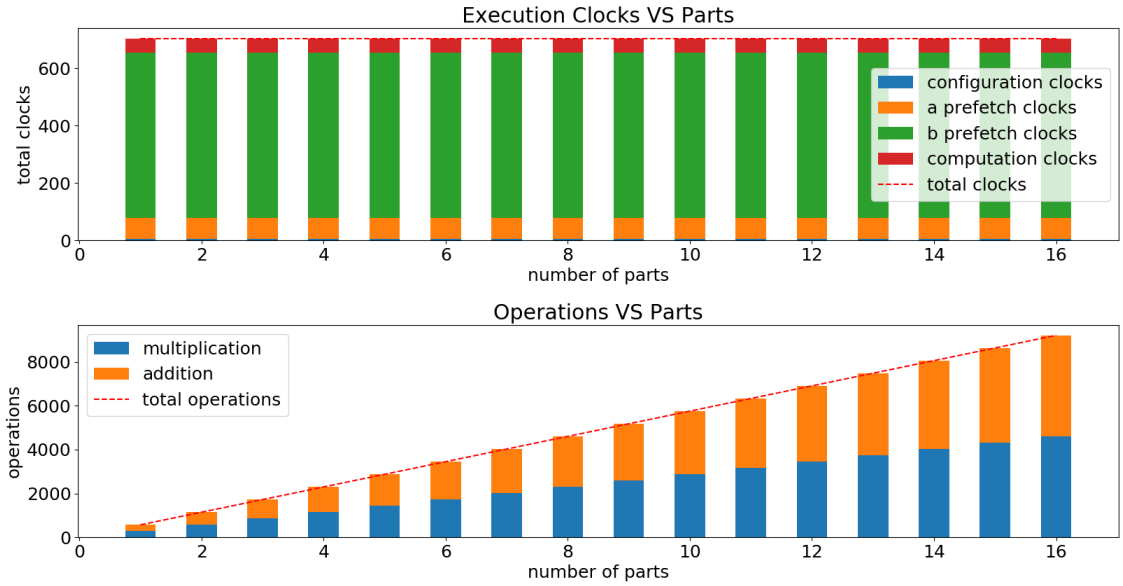

Figure 4.13: Benchmark versus Parts

4.4 Summary of Validation

As CNNA has $16*16 = 169$ parallel computing channels, theorical accelerated time is $169 * 2 = 338$ (Each MAC unit has one multiplier and one adder). However, due to the need of configuration and pre-caching some of the data, the actual acceleration is less than 338 times. For accelerating convolutional layer, we can see from the validation result in figure 4.7, the peak performance of CNNA is around 300 and can be close to 338 if NA is larger. But unfortunately, this goal cannot be achieved in a fully connected layer, as the pre-cache phase is the heaviest part and much heavier than the computation phase.

The performance gap between CNNA in FC and convolutional layer is caused by the following two reasons: On one hand, for the same amount of computation, it is true that full-connected layers require more data inputs than convolutional layers, as convolution has much more data reuse. On the other hand, as CNNA must prefetch **b** completely, the larger of **b** makes prefetch phase heavier, which is different from **a**, which is mostly cached during computation.

Finally, in order to fully/better utilize CNNA, it is recommended to build

a wider network (larger channels, both input channels and output channels), as wider network can make better use of memory bandwidth and computing resources.

5 Conclusion

5.1 Evaluation of CNNA

From validation/simulation in chapter 4, it is proven that the architecture of CNNA is feasible and very suitable for the characteristics of data reuse in convolutional layer. However, since the convolutional layer focuses on data reuse rather than memory bandwidth, which is on the contrary of fully-connected layer focusing on memory bandwidth and CNNA is mainly designed for convolutional layer, the performance of the CNNA for accelerating fully-connected layer is not as good as for convolutional layer.

5.2 Improvement of CNNA

During the verification and writing of the thesis, we have discovered some places, where CNNA can be improved.

- Firstly, CNNA pre-fetches as much data as possible before computation. But it is not necessary. CNNA can start to computate before the prefetch phase is finished. This allows to fully utilize the computation resource during pre-fetch phase. It is estimated that there is about 10% performance improvement.

- Secondly, CNNA does not support filter/NA reuse. After being woken up, it needs to re-prefetch data even though filter/NA is the same as last time. If supporting filter/NA reuse, the performance improvement for convolutional layer is about 10%, but for FC about 80%, as the prefetch phase takes more then 80% time in FC.

- Thirdly, the input and output bandwidths are inconsistent, which will cause great difficulties in actual use.

- Fourthly, the performance of CNNA single chips is limited and actual application requires multi-chip cascades. However, there are no relevant modules to support it.

- Fifthly, adding more supporting mode of CNNA though enabling/disabling different modules, such as **Accumulate Adder Unit**, to increases CNNA's functionality, especially for matrix operation, enhancing the application of CNNA. At present, CNNA can perform matrix operations, but needs to be split into multiplication of vectors and matrices. This is due to the always enabling of the Accumulate Adder Unit.
- Sixthly, discarding Adder enable function in **Accumulate Adder Unit** to save hardware resources, as in practical, its function to save power consumption is minimal.

Appendix

A Resource Utilization Report

```
Copyright 1986-2015 Xilinx, Inc. All Rights Reserved.
------------------------------------------------------------------------------------
| Tool Version : Vivado v.2015.4.1 (lin64) Build 1431336 Fri Dec 11 14:52:39 MST
2015
| Date         : Sun Feb 18 14:45:54 2018
| Host         : atacama.hpsn.et.tu-dresden.de running 64-bit Scientific Linux
release 6.9 (Carbon)
| Command      : report_utilization -file reports/mars_ax3-a100t-1/
post_synthesis_utilization.rpt
| Design       : cnna_udp_top
| Device       : 7a100tcsg324-1
| Design State : Synthesized
------------------------------------------------------------------------------------

Utilization Design Information

Table of Contents
-----------------
1. Slice Logic
1.1 Summary of Registers by Type
2. Memory
3. DSP
4. IO and GT Specific
5. Clocking
6. Specific Feature
7. Primitives
8. Black Boxes
9. Instantiated Netlists

1. Slice Logic
--------------

+----------------------------+-------+-------+-----------+-------+
```

Site Type	Used	Fixed	Available	Util%
Slice LUTs*	49386	0	63400	77.90
LUT as Logic	45264	0	63400	71.39
LUT as Memory	4122	0	19000	21.69
LUT as Distributed RAM	4104	0		
LUT as Shift Register	18	0		
Slice Registers	33312	0	126800	26.27
Register as Flip Flop	33312	0	126800	26.27
Register as Latch	0	0	126800	0.00
F7 Muxes	45	0	31700	0.14
F8 Muxes	2	0	15850	0.01

* Warning! The Final LUT count, after physical optimizations and full implementation, is typically lower. Run opt_design after synthesis, if not already completed, for a more realistic count.

1.1 Summary of Registers by Type

Total	Clock Enable	Synchronous	Asynchronous
0	_	-	-
0	_	-	Set
0	_	-	Reset
0	_	Set	-
0	_	Reset	-
0	Yes	-	-
34	Yes	-	Set
7790	Yes	-	Reset
116	Yes	Set	-
25372	Yes	Reset	-

2. Memory

Site Type	Used	Fixed	Available	Util%
Block RAM Tile	16.5	0	135	12.22
RAMB36/FIFO*	16	0	135	11.85
RAMB36E1 only	16			
RAMB18	1	0	270	0.37
RAMB18E1 only	1			

* Note: Each Block RAM Tile only has one FIFO logic available and therefore can accommodate only one FIFO36E1 or one FIFO18E1. However, if a FIFO18E1 occupies a Block RAM Tile, that tile can still accommodate a RAMB18E1

3. DSP

Site Type	Used	Fixed	Available	Util%
DSPs	16	0	240	6.67
DSP48E1 only	16			

4. IO and GT Specific

Site Type	Used	Fixed	Available	Util%
Bonded IOB	21	0	210	10.00
Bonded IPADs	0	0	2	0.00
PHY_CONTROL	0	0	6	0.00
PHASER_REF	0	0	6	0.00
OUT_FIFO	0	0	24	0.00
IN_FIFO	0	0	24	0.00
IDELAYCTRL	1	0	6	16.67

IBUFGDS	0	0	202	0.00
PHASER_OUT/PHASER_OUT_PHY	0	0	24	0.00
PHASER_IN/PHASER_IN_PHY	0	0	24	0.00
IDELAYE2/IDELAYE2_FINEDELAY	5	0	300	1.67
IDELAYE2 only	5	0		
IBUFDS_GTE2	0	0	4	0.00
ILOGIC	5	0	210	2.38
IDDR	5			
OLOGIC	6	0	210	2.86
ODDR	6			

5. Clocking

Site Type	Used	Fixed	Available	Util%
BUFGCTRL	2	0	32	6.25
BUFIO	1	0	24	4.17
BUFIO only	1	0		
MMCME2_ADV	1	0	6	16.67
PLLE2_ADV	0	0	6	0.00
BUFMRCE	0	0	12	0.00
BUFHCE	2	0	96	2.08
BUFR	1	0	24	4.17

6. Specific Feature

Site Type	Used	Fixed	Available	Util%
BSCANE2	0	0	4	0.00
CAPTUREE2	0	0	1	0.00
DNA_PORT	0	0	1	0.00

EFUSE_USR	0	0	1	0.00
FRAME_ECCE2	0	0	1	0.00
ICAPE2	0	0	2	0.00
PCIE_2_1	0	0	1	0.00
STARTUPE2	0	0	1	0.00
XADC	0	0	1	0.00

7. Primitives

Ref Name	Used	Functional Category
FDRE	25372	Flop & Latch
LUT6	16025	LUT
LUT3	13677	LUT
LUT2	9000	LUT
FDCE	7790	Flop & Latch
CARRY4	7524	CarryLogic
LUT4	6122	LUT
LUT5	5307	LUT
RAMD64E	3104	Distributed Memory
RAMD32	1380	Distributed Memory
LUT1	851	LUT
RAMS32	540	Distributed Memory
FDSE	116	Flop & Latch
MUXF7	45	MuxFx
FDPE	34	Flop & Latch
SRL16E	18	Distributed Memory
RAMB36E1	16	Block Memory
DSP48E1	16	Block Arithmetic
OBUF	12	IO
IBUF	9	IO
ODDR	6	IO
IDELAYE2	5	IO
IDDR	5	IO
MUXF8	2	MuxFx

BUFHCE	2	Clock
BUFGCTRL	2	Clock
RAMB18E1	1	Block Memory
OBUFT	1	IO
MMCME2_ADV	1	Clock
IDELAYCTRL	1	IO
BUFR	1	Clock
BUFIO	1	Clock

8. Black Boxes

Ref Name	Used

9. Instantiated Netlists

Ref Name	Used
udp_ethernet_if	1

References

[1] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. "DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '14. Salt Lake City, Utah, USA: ACM, 2014, pp. 269–284. ISBN: 978-1-4503-2305-5. DOI: 10.1145/2541940.2541967. URL: http://doi.acm.org/10.1145/2541940.2541967 (cit. on pp. vii, 17).

[2] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos. "Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing". In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 2016, pp. 1–13. DOI: 10.1109/ISCA.2016.11 (cit. on pp. vii, 17, 50).

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger. Curran Associates, Inc., 2012, pp. 1097–1105. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf (cit. on p. 1).

[4] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. "ImageNet Large Scale Visual Recognition Challenge". In: *Int. J. Comput. Vision* 115.3 (Dec. 2015), pp. 211–252. ISSN: 0920-5691. DOI: 10.1007/s11263-015-0816-y. URL: http://dx.doi.org/10.1007/s11263-015-0816-y (cit. on p. 1).

[5] Daniel Sonntag Jan Zacharias Michael Barz. *A Survey on Deep Learning Toolkits and Libraries for Intelligent User Interfaces*. 2018. URL: https://arxiv.org/abs/1803.04818 (cit. on pp. 1, 11).

[6] Frontiers Research. *Artificial Neural Networks as Models of Neural Information Processing*. 2018. URL: https://www.frontiersin.org/research-topics/4817/artificial-neural-networks-as-models-of-neural-information-processing#overview (cit. on p. 3).

[7] K. Biol. Cybernetics Fukushima. "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position". In: (1980). URL: https://doi.org/10.1007/BF00344251 (cit. on p. 3).

[8] K. Fukushima, S. Miyake, and T. Ito. "Neocognitron: A neural network model for a mechanism of visual pattern recognition". In: *IEEE Transactions on Systems, Man, and Cybernetics* SMC-13.5 (1983), pp. 826–834. ISSN: 0018-9472. DOI: 10.1109/TSMC.1983.6313076 (cit. on p. 3).

[9] V. Sze, Y. Chen, T. Yang, and J. S. Emer. "Efficient Processing of Deep Neural Networks: A Tutorial and Survey". In: *Proceedings of the IEEE* 105.12 (2017), pp. 2295–2329. ISSN: 0018-9219. DOI: 10.1109/JPROC.2017.2761740 (cit. on pp. 3–5, 7–9).

[10] Ronald J. Williams David E. Rumelhart Geoffrey E. Hinton. "Learning representations by back-propagating errors". In: *Nature* (1986). URL: http://dx.doi.org/10.1038/323533a0 (cit. on pp. 4, 11).

[11] *Activation function*. https://en.wikipedia.org/wiki/Activation_function. Accessed: 2018-02-30 (cit. on p. 7).

[12] A. Maas, A. Hannun, and A. Ng. "Rectifier Nonlinearities improve Neural Network Acoustic Models". In: *Proc. 30th Int. Conf. on Machine Learning (ICML)*. 2013 (cit. on p. 8).

[13] Pete Warden. *How to Quantize Neural Networks with TensorFlow*. https://petewarden.com/2016/05/03/how-to-quantize-neural-networks-with-tensorflow/. Accessed: 2018-02-30 (cit. on p. 11).

[14] S. Liu and W. Deng. "Very deep convolutional neural network based image classification using small training sample size". In: *2015 3rd IAPR Asian Conference on Pattern Recognition (ACPR)*. 2015, pp. 730–734. DOI: 10.1109/ACPR.2015.7486599 (cit. on p. 19).

[15] *tf.nn.conv2d in TensorFlow*. https://www.tensorflow.org/versions/r1.2/api_docs/python/tf/nn/conv2d. Accessed: 2018-02-30 (cit. on pp. 19, 64).

[16] *tf.matmul in TensorFlow*. https://www.tensorflow.org/versions/r1.2/api_docs/python/tf/matmul. Accessed: 2018-02-30 (cit. on pp. 19, 73).