TEAM NAME : COLORIZER

Team Member	Ferriady Setiawan	Fransiskus Yoga Esa	Naufal Ridho H.
Email	kim.setiawan @gmail.com	fransiskusyoga @gmail.com	naufal.ridho8 @gmail.com
Grade	8th semester	8th semester	8th semester
Size of T-Shirt	L	L	L

Bandung Institute of Technology
Phone Number : (+62 22) 2500935
Address : Labtek VIII Electrical Engineering School of Electrical and Engineering
Jalan Ganesha No. 10 Bandung

COLORIZER: IMAGE COLORIZATION ON FPGA USING ARTIFICIAL NEURAL NETWORK (ANN) BY APPLYING SLIDING WINDOW TECHNIQUE

Naufal Ridho H (13214008)
Fransiskus Yoga Esa (13214012)
Ferriady Setiawan (13214034)
Date: 09/02/2018
EL4138- VLSI System Design
SEEI – ITB

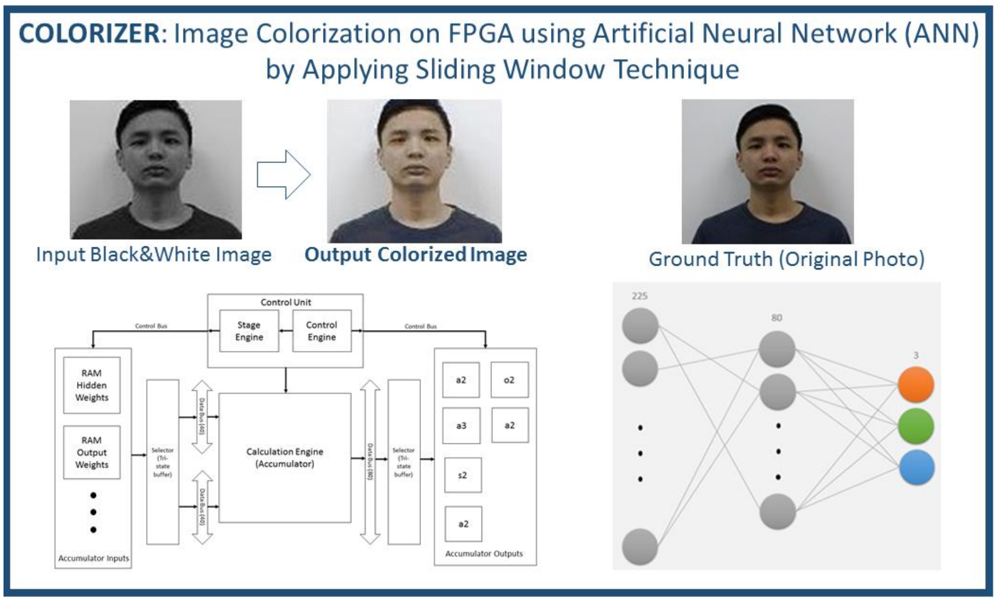

COLORIZER: Image Colorization on FPGA using Artificial Neural Network (ANN) by Applying Sliding Window Technique
Input Black&White Image
Output Colorized Image
Ground Truth (Original Photo)
Control Unit
Stage Engine
Control Engine
Control Bus
Control Bus
RAM Hidden Weights
RAM Output Weights
Accumulator Inputs
Selector (Tri-state buffer)
Data Bus
Calculation Engine (Accumulator)
Data Bus
Selector (Tri-state buffer)
a2
o2
a3
a2
s2
a2
Accumulator Outputs
225
80
3

Fig. 1.1 Overview of the COLORIZER System

ABSTRACT

COLORIZER is a proposed image colorization system that is specifically designed for FPGA. Image colorization itself is a process of adding color to a grayscale image. Using COLORIZER, the system is let to colorize the photo automatically by mapping a grayscale image (1-channel pixel) to a colorized output image (3-channel RGB pixel). The COLORIZER system uses deep learning approach and Artificial Neural Network (ANN) to recognize patterns in an image so image colorization can be done automatically. Training process is usually very slow using single-threaded CPU because of its very heavy computational load. The computation, which is involves bulk of simple multiplication and accumulation, actually can be done parralel. Thus, we need to 'parallelize' the computation and find a method to compensate its resource-consuming tasks. We implement the COLORIZER system using FPGA Altera DE2-115 and also applies Sliding Window Technique. Take a note that the neural networks is fully implemented within the FPGA, this means that the FPGA is not only used as a hardware accelerator. By applying Sliding Window Technique to the system we also only need 3 output perceptrons rather than the number of all image pixels (3 perceptrons to represent Red, Green Blue channels). Furthermore, with this technique we can create many training sets only from one image. Sliding Window Technique is actually a widely used technique in object detection. But, we saw opportunity that this technique can be used 'hand-in-hand' with ANN to implement image colorization on FPGA.

1. INTRODUCTION

Image colorization is a process of adding color to a grayscale image. Traditionally this was done by hand with human effort because it is such a difficult task. Recent deep learning algorithm can be used to recognize patterns within the photograph to color the image, much like human might approach the problem (recognizing blue skies, green leaves, black hairs, and so on). So, days of colorizing image manually by hand can be saved with this approach.

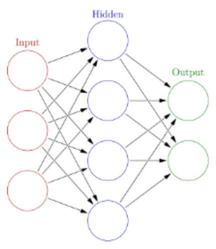


Fig. 1.2 Artificial Neural Network (ANN) model

Training process-transforming input patterns into processed outputs- generally involves very large neural networks application consisting a lot of perceptrons. Moreover, deep learning is commonly implemented on single-threaded CPU. This implementation is very slow that the process requires several days of training because of its very heavy computational load. The computation, which involves bulk of simple multiplications and accumulations, actually has a lot of computation that can be parallelized. Thus, we need to 'parallelize' the computation and find a method to compensate its resource-consuming system

COLORIZER is a proposed system of image colorization application that is specifically designed for FPGA. FPGA is needed to do parallel computation so the learning process is much faster than single-threaded computation using CPU. As described in the passage above, the COLORIZER system colorizes the photo automatically by mapping a grayscale image (1-channel pixel) to a colorized output image (3-channel RGB pixel). We also applies Sliding Window Technique to make implementation of fully-connected neural network on FPGA possible (thus the FPGA is not only used as a hardware accelerator). We exploit the FPGA constraints to present an application that is both unique and satisfying the specifications while still keeping to boost performance.

COLORIZER system can be used in the entertainment and educational areas. People can enhance their old black and white photos using COLORIZER to recall old memories in color photo. Old historical photos can also be enhanced with this system. Soon, old videos such as movies and historical documentaries-that consists of thousands of thousands of frame images-can also be colorized by using this image colorization system. Currently, most implementations of image colorization are only limited to software, which is running on CPU (slow). Our team presents a new concept of image colorization which is running on hardware-that is much faster. FPGA is the answer to do COLORIZER much faster than CPU and GPU.

2. Appeal Point and Originality

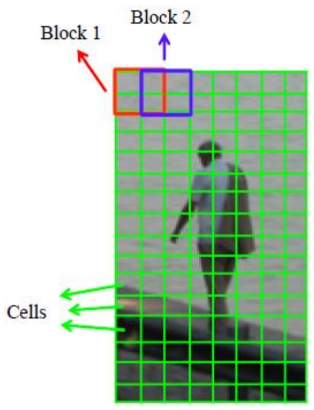


Fig. 2.1 Sliding Window Technique

Our COLORIZER system is an image colorization system that uses *Artificial Neural Networks* (ANN) as the classifier. The ANN classifier is fully-implemented using FPGA Altera DE2-115. In other words, the system is not only used as a hardware accelerator. But, the neural network itself is synthesized directly to the hardware--FPGA. So, the expected performance of the colorization process must be high.

Training process involves the use of a very large neural networks consisting a lot of perceptrons. By applying Sliding Window Technique to the system, we only need 3 output perceptrons rather than as many as the number of all image pixels. With this technique, we can compensate the resource requirements of the system. Sliding Window Technique is actually a widely used technique in object detection. But, we saw opportunity that this technique can be used 'hand-in-hand' with ANN to implement image colorization on FPGA that has limited logic elements.

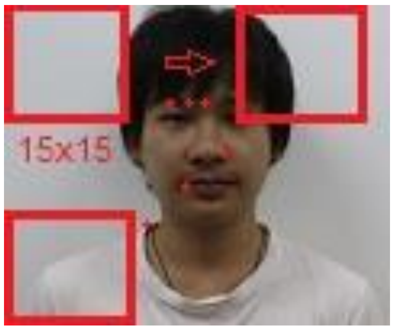


Fig. 2.2 Sliding Window Technique in the COLORIZER system

Image pixels will be scanned through this rectangular window, which has a fixed number of dimension (sub-image). The window will slide and take a small part of the picture as a sample. The sample then will become input for the system ANN classifier. In our design, the ANN classifier is a two layer perceptron neural network which is in accordance to the specified challenge (Level 2: For Experts). The rectangular window's size is 15x15 pixel. Because there have been nobody has ever used this Sliding Window Technique for coloring greyscale image, our design can be categorized as original.

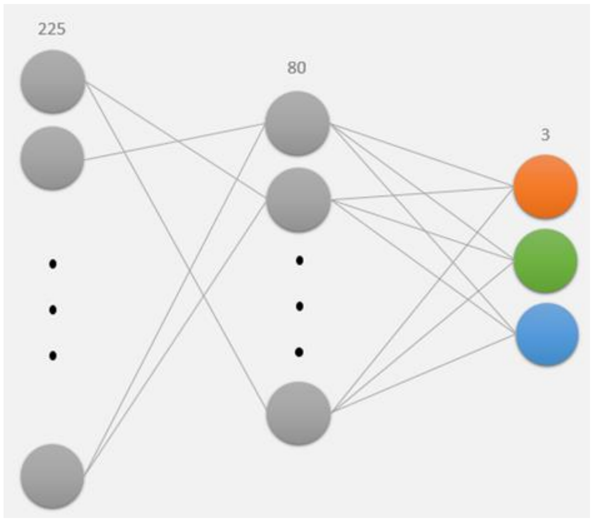


Fig. 2.3 Proposed Artificial Neural Network (ANN) Model

The proposed Artificial Neural Network (ANN) model has the following parameters:

- Input layer : 225 perceptrons
  The number of 225 perceptrons comes from chosen 15x15 sliding window dimension, which is 225.
- Hidden layer : 80 perceptrons
  Arbitrarily chosen 80 perceptrons. Proved to be suitable through simulation and testing.
- Output layer : 3 perceptrons
  Representing 3-channel RGB color values of only one pixel.
- All weights value are randomized with value [0 1]
- Bias is initialized with 0 and is not update-able. So, we are not use bias in our Colorizer system
- Learning rate of 0,1
- Activation function is an approximation of sigmoid function using partial linearization

2.1 GENERAL ADVANTAGE

Design features in COLORIZER system is described below,

1. Faster than CPU Implementation
   Deep learning is commonly implemented on single-threaded CPU which is very slow because of its very heavy computational load. Use of FPGA allows parallel computation that is much faster than single-threaded computation using CPU.

   To implement the system we need to do *scheduling*. With scheduling we decide when to calculate which perceptrons or when to store values to embedded memory. Operations such as multiplication and accumulation (addition) are scheduled to be performed in stages. It is controlled by blocks we called Stage Engine and Control Engine. Operations are performed by Perceptron Units within Calculation Engine.

   Scheduling scheme is explained more detail by Block Diagram and Table in the RTL Diagram and Scheduling section. The RTL diagrams also described there.

2. Uses Less Resources

   Training process generally involves the use of very large neural networks consisting a lot of perceptrons. By applying Sliding Window Technique to the system we only need 3 output perceptrons rather than as many as the number of all image pixels. With this technique we can compensate the resource requirements of the system.

   Sliding Window Technique is actually a widely used technique in object detection. But, we see opportunity that this technique can be used 'hand-in-hand' with ANN to implement image colorization on FPGA that has limited logic elements.

3. Creates Many Training Sets from one Color Image

With Sliding Window Technique, the system may create many training sets only from one image. The system extracts patterns samples from an image to be used as training sets.

$$\#samples = (h_{image} - h_{window} - 1) \times (w_{image} - w_{window} - 1)$$

$h_{image} = height\ of\ training\ image\ (in\ pixels)$
$w_{image} = width\ of\ training\ image\ (in\ pixels)$
$h_{window} = height\ of\ sliding\ window\ (in\ pixels)$
$w_{window} = width\ of\ sliding\ window\ (in\ pixels)$

Note: Numbers of sliding window height and width pixels must be odd

3. PARAMETER AND APPROXIMATION

There are several design parameter and work flow be determined before the designing the hardware. Those parameter and alternative design are observe and determined by MATLAB software.

3.1 PARAMETER AND GRADIENT DESCENT METHOD

Neural network is a classification method. Nowadays, there are many variations of neural network. In general, there isn't the best neural network method. Specific method that is used in this system will be explained in the passage below.

Implemented neural network design is determined by the result of MATLAB simulation. There are several parameters and alternative designs that are simulated in the software. Several neural network alternatives that are tested on this process are batch gradient descent, mini batch gradient descent, and stochastic gradient descent. Neural networks also has several parameters which need to be tuned first so we can get convergent output (The error is small). Those parameters are :

1. number of perceptrons in the hidden layer
2. initial weights and bias
3. learning rate
4. activation function type
5. number of iteration

Stochastic gradient descent is a variation of the gradient descent algorithm that calculates the error and updates the model for each example in the training dataset. This means that stochastic gradient descent is an online machine learning algorithm. There are several upsides of this method. The frequent updates immediately give an insight into the performance of the model and the rate of improvement. This variant may also be the simplest to understand and implement, especially for beginners. But the most important aspect of this variant is the noisy update process. The noisy update process can allow the model to avoid local minima or premature convergence.

Batch gradient descent is a variation of the gradient descent algorithm that calculates the error for each example in the training dataset, but only updates the model after all training examples have been evaluated. One cycle through the entire training dataset is called a training epoch. Therefore, it is often said that batch gradient descent performs model updates at the end of each training epoch.

Mini-batch gradient descent is a variation of the gradient descent algorithm that splits the training dataset into small batches that are used to calculate model error and update model coefficients. Implementations may choose to sum the gradient over the mini-batch or take the average of the gradient which further reduces the variance of the gradient. Mini-batch gradient descent seeks to find a balance between the robustness of stochastic gradient descent and the efficiency of batch gradient descent.

Among tested methods, stochastic gradient descent and mini batch give slight different result. Otherwise, the batch design tend to give blurry image. In batch gradient descend, neural network system approach overall data. This system is better on minimizing the error data in general but loses its ability to classifying the local region.

In hardware design, stochastic gradient descent has significant advantage. Batch and mini batch gradient descend need memory bank to save delta weight value. Weight and delta weight require a lot of space.

As shown above memory of weight and delta weight immensely bigger compare to the memory. Stochastic gradient descent method does not require this memory consuming variable. Implementing stochastic gradient descent will prevent memory explosion with fair output result.

3.2 SIGMOID AND DERIVATIVE OF SIGMOID APPROXIMATION

Sigmoid is a nonlinear function involving exponential and division. Without any approximation the system demand large space on FPGA. Due to limited resource of FPGA's LUTs, using LUT for sigmoid function approximation will limit the number of sigmoid function in the system and may not be feasible for several design.

3.2.1 Choosing activation function

High linearity function such as ReLU are easy to implement and has minimum area but ReLU are less effective for neural net with few layers. This activation prone to fail to classify simple model with shallow neural network. ReLU are more effective for excessive layer such as convolutional neural network (CNN).

Tanh and sigmoid have similar characteristic. The main difference between those function is their output range. Tanh output value is between -1 and 1, sigmoid output range is 0 to 1. Because of that characteristic, sigmoid function is more convenient for classification.

Equation for sigmoid function:

$$A = \frac{1}{1+e^{-x}}$$

3.2.2 Sigmoid function approach

Sigmoid function are estimated using partial linearization. One sigmoid function are devided into 5 zones as shown in figure below.

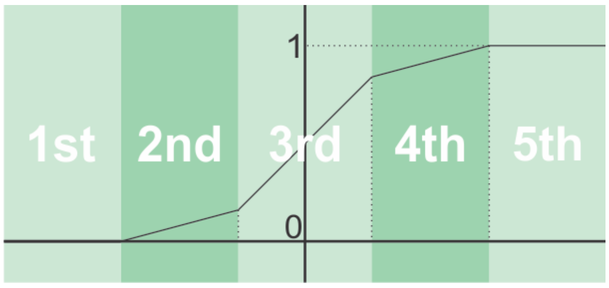


Fig. 3.2.2.1 Function partition zone

First and fifth zone has zero gradient with 0 and 1 constant value respectively. The gradient of second zone is the same with the gradient of fourth zone which are not specified yet. To minimize FPGA design complexity, value of 2nd, 3rd, and 4th zone are multiples of 2 or ½.

Approximation fitness are measured by total error area and maximum error. Two best approximation are partial linearization with 0.25 (2nd zone) – 0.125 (3rd and 4th zone) and 0.25 (2nd zone) – 0.0625 (3rd and 4th zone). The approximation graph and approximation function are presented in section below.

Sigmoid Function

In this task, the calculation method for our neural network system will be the same as we will implement using FPGA. The sigmoid function cannot be implemented in FPGA without simplification. The simplification will be used for sigmoid function and its derivative. Comparison between the real equation and the approximation can be seen below.

Approximation for sigmoid function :

$$A = \begin{cases} 0 & ;\ x < -3.026 \\ 0.2565 + m/8 & ;\ -3.026 \le x < -0.974 \\ 0.5 + m/4 & ;\ -0.974 \le x < 0.974 \\ 0.7435 + m/8 & ;\ 0.974 \le x < 3.026 \\ 1 & ;\ x \ge 3.026 \end{cases}$$

We simulate the real function and the approximation function using MATLAB. And picture above shows the comparison and the error value.

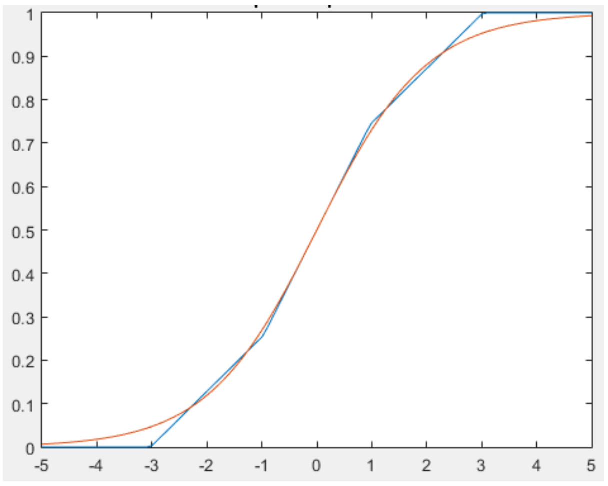


Fig. 3.2.2.2 Sigmoid approximation with gradient 1/4 and 1/8

The red line is the real sigmoid function and the blue line is the approximation function. The error can be calculated below :

$$Max\ error = 0.0461$$

$$Total\ error\ area = 0.0772$$

Derivative of Sigmoid Function

The derivative of sigmoid function can be seen at equation below :

$$y = 1./(1 + exp(-x))$$

The approximation of this function is a discrete function which is :

$$y = \begin{cases} 0 & ;\ x < -3.026 \\ 1/8 & ;\ -3.026 \le x < -0.974 \\ 1/4 & ;\ -0.974 \le x < 0.974 \\ 1/8 & ;\ 0.974 \le x < 3.026 \\ 0 & ;\ x \ge 3.026 \end{cases}$$

Because the function is a linear function, derivative is only a constant number. So, the implementation of derivative of sigmoid function is a purely combinational circuit (no register needed).

Function approximation B

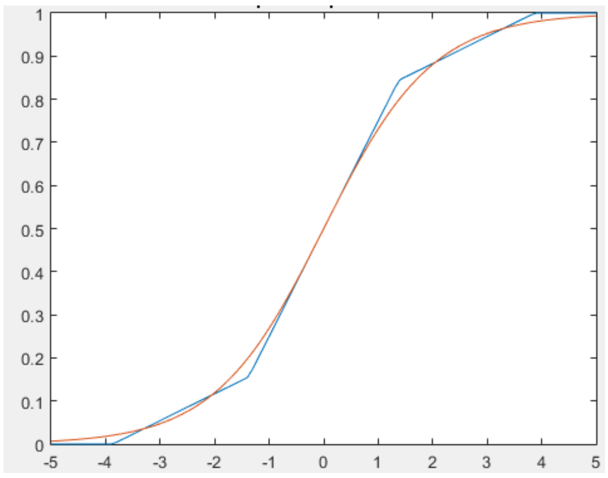
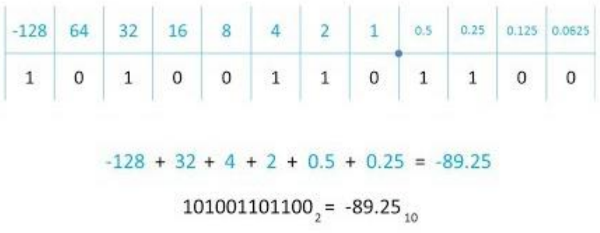


Fig. 3.2.2.3 Sigmoid approximation with gradient 1/4 and 1/16

$$Max.error = 0.0453$$

$$Total\ error\ area = 0.0644$$

3.3 FIXED POINT BIT WIDTH



-128	64	32	16	8	4	2	1	0.5	0.25	0.125	0.0625
1	0	1	0	0	1	1	0	1	1	0	0

-128 + 32 + 4 + 2 + 0.5 + 0.25 = -89.25

$101001101100_2 = -89.25_{10}$

Fig. 3.3.1 Fixed-point representation

There are two methods of representing numbers in a computer system, fixed- or floating-point number systems. Fixed-point representation maintains the decimal point within a fixed position, allowing for straightforward and simple arithmetic operations. The major drawback of the fixed-point system is that to represent larger numbers or to achieve a more accurate result with fractional numbers, we will need to use a larger number of bits. So, fixed point number representation has constant resolution and smaller range compared to floating point representation. Its limitation can be suppressed by increasing the word length. Contrast to that, larger bit word requires a lot of space. However, due to the complexity of floating-point numbers, we tend to use fixed-point representations within the design.

Because of its constant resolution, we had to choose the proper number of bits to make the system as efficient as possible. We tried to evaluate first the values that will be stored in the memory. The purpose of the evaluation is to choose the right fixed-point format *s:m:f* number of bits, with *s*, *m* and *f* respectively being the sign, integer, and fractional bits. Because fixed point numbers have constant resolution, we have to choose the appropriate number of integer and fractional bits beforehand.

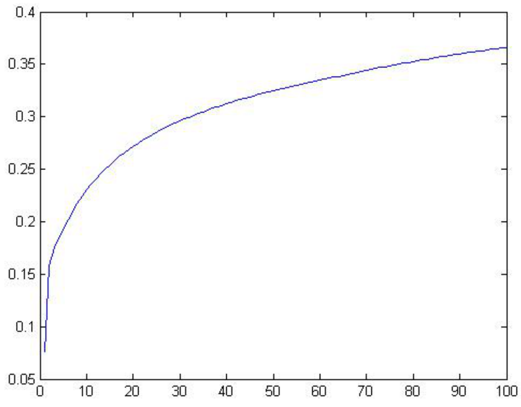


Fig. 3.3.2 Evaluation of W3 values average (x-axis represents number of iterations). Increasing number means higher required MSB bit

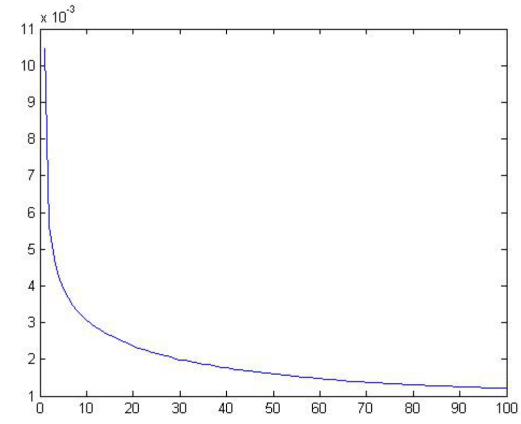
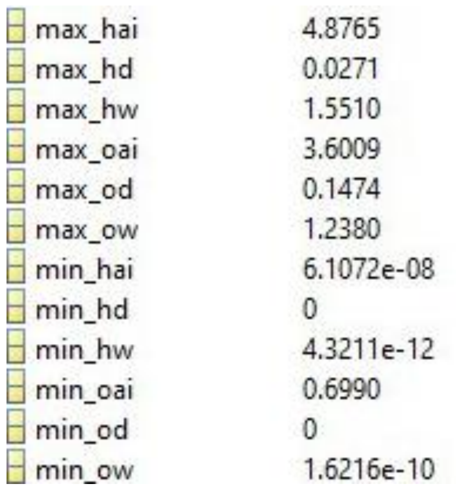


Fig. 3.3.3 Evaluation of ΔW3 values average (x-axis represents number of iterations). Increasing number means lower required LSB bit



max_hai	4.8765
max_hd	0.0271
max_hw	1.5510
max_oai	3.6009
max_od	0.1474
max_ow	1.2380
min_hai	6.1072e-08
min_hd	0
min_hw	4.3211e-12
min_oai	0.6990
min_od	0
min_ow	1.6216e-10

Fig. 3.3.4 Evaluation of values using MATLAB

We use two different formats in this system, 0:4:20 and 1:0:15, meaning that the former format is unsigned, has 4 integer bits, and has 20 fractional bits (word length is 24-bit). This format entire word length is two's complements number. The latter number format, 1:0:15, has 1 signed bit and 15 fractional bits.

As described above, there are 2 fixed point types which is used:

1. 0:4.20, 4 integer, 20 fractional bits, which later will be used for
   - W2, W3 (hidden and output) weight values
   - Z3, Z3 sum of weighted inputs
   - S2, S3 delta bias
   - O2, O3 derivate of weights
2. and 1:0:15, 1 signed, 15 fractional bits, which will be used for,
   - activation function output a2, a3 (min value =0, max value =1)
   - image input (8-bit for each pixel, using shifting combinational)
   - image output (8-bit for each color channel in a pixel, 24-bit true color)

   Actually, we only need 8 bits for activation function, image input, and image output values. So, we only use 8 fractional bits out of 15 bits in the second format (1:0:15).

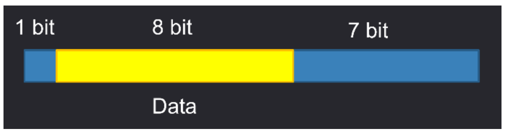


Fig. 3.3.5 Diagram of 1:0:15 number format. Yellow: used bits. 1-bit blue: sign bit. 7-bits blue: unused bits.

Later, we created 40 memory blocks with depth of 512 words. The width of the data every memory block is 24-bit. More detailed explanation of the memory addressing and scheduling scheme will be explained later on 4.1.3 and 4.2.2.

4. Block Diagram and Architecture Description

4.1 Overall Architecture

4.1.1 Hardware Overview

On the overall Colorizer System that will be developed can be seen at diagram below:

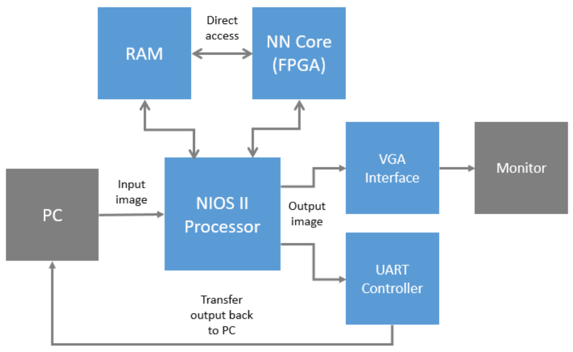


Fig 4.1.1.1 System Architecture of COLORIZER System (Look Appendix: Diagrams for more detailed diagram)

Blocks that are built in the COLORIZER system:

- NN_Core
  A block to handle the neural networks, either forward or backward propagation. Neural networks is fully implemented into the FPGA
- Embedded Memory (RAM)
  A block to store input image and update neural network weights. Used for compensation of FPGA limited logic elements (limited register)
- VGA Controller
  Used for displaying output image to VGA monitor
- UART Controller
  Used for transferring back output image to PC
- NIOS II Processor
  We also implement NIOS II soft processor. Soft processor is used to utilize peripherals, mainly UART Controller, which is used to transfer data back to PC. Soft processor integrates all the blocks as one system. Note that RAM is directly accessed by NN Core,

so calculation can be done much faster (with Scheduling scheme that will be discussed later).

4.1.2 SKEWED ACCUMULATIN AND SYSTOLIC CALCULATION

Our computaional core are able to perform either skewed accumulation and systolic calculation. The utilization of those basic calculation enable the system to gain high computaion utilization. This section will discuss the fundametal consideration of computation scheme that are implemented in FPGA.

Skewed Accumulator

Accumulator is a module that perform sequence of addition. Accumulator add current input data with current then store it in register. This module is represented as mathematical sigma operator that calculate queues of addition. Furthermore, accumulator support matrix multiplication which basically series of additioin.

In neural network, both forward propagation and back propagation are sum of weighted input which highly rely on series of addition. This operation are handled by clusters of accumulator. This cluster do summation and multiplication of respective input data.

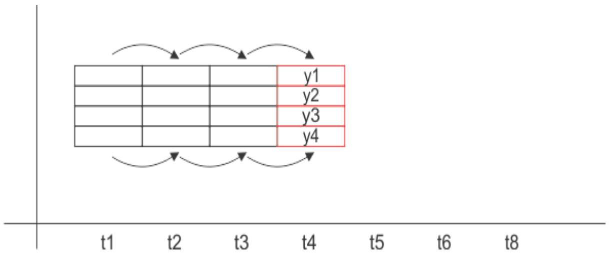


Fig 4.1.2.1 Illustration of simultaneous accumulator

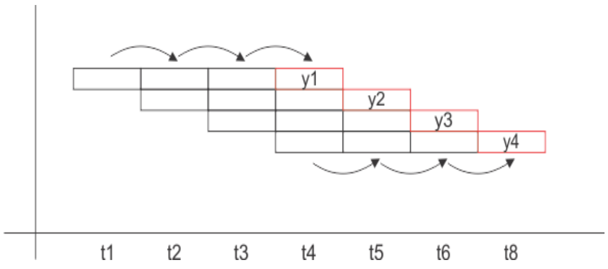


Fig 4.1.2.2 Illustration of skewed accumulator

Rather than simultaneous accumulator, the cluster do skewed calculation. Ilustration above show the difference of simultaneous and skewed calculation. Each row represents one acumulator module, each column represent time domain. Every accumulator will execute several additions from the first data until the last data. The desired ouput is indicated by red-bordered box. In the simultaneous calculation, the computation process of each accumulator begin and finish at the same time. Maintaning usability of simultaneous calculation require multiple input register which need numerous logic cells. Otherwise, skewed calculation outputs one data every single clock tick. This calculation method is able to utilize single input register such as onchip ram.

Systolic calculation

Accumulator module has great performance on mutiple output calculation but less aplicable on single ouput. In example, accumulator are more suitable to compute calcualtion a rather than calculation b.

Calculation a

Out1 = a11*b11 + a12*b12

Out2 = a21*b21 + a22*b22

Calcualtion b

Out = a11*b11 + a12*b12 + a21*b21 + a22*b22

One output value will be handled by one accumulator or processing unit. In the example above, by using 2 accumulators, calculation a will be done on 2 clock tick. At the first clock tick, processing units calculate out1_1 = a11*b11 + 0 and out2_1 = a21*b21 + 0 at the same time. At the next clock, it calculates out1 = a12*b12 + out1_1 and out2 = a22*b22 + out2_1 at the same time. Calculation process in calculation b is different. There is only 1 equation to be processed. This process can only be hanled by 1 accumulator. Although the system contains 2 accumulators, only 1 accumulator is active. In consequence, the troughput of system is 4 clock cycle, independent of accumulator number.

Concering the accumulator weakness, other calculation method should be applied in few-ouput calculation. If output is fewer than the accumulator module, then cluster of accumulators will give low hardware utilization. Since the outputs of third layer (a3) are 3, at best, only 3 accumulator can be used for a3 calculation. This condition will significatly decrease the system speed. Other calculation method should be implemented to over pass this limitation.

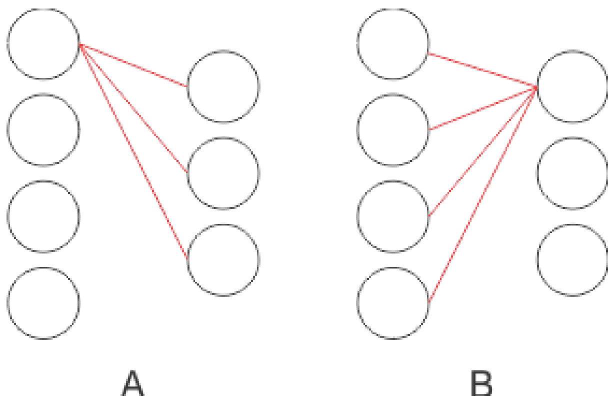


Fig 4.1.2.3 a. Divergent calculation (best on accumulator) b. convergent calcuation (best on systolic)

As the solution, our system enables systolic calculation alongside skewed accumulation. Contrast to accumulator, systolic calculation is able to produce one output value using many processing unit.

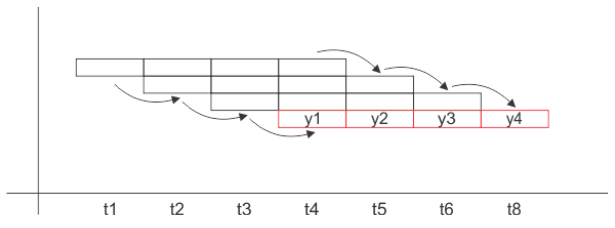


Fig 4.1.2.4 Illustration of systolic calculation

4.1.3 CALCULATION OVERVIEW AND SCHEDULING

This section will provide information about function being process in whole system cycle. This section is a fundamental concept of further discussion.

MATLAB function call

Calculation below is a fully functional MATLAB code for full forward-backward calculation that is implemented in hardware. The function below are separated into different blocks. Each block is a different stage system that will not processed at the same time. Combinational process such as sigmoid function, derivative of sigmoid function, and adder allows FPGA to hold respectively z and a value, err and s3 value, o and wn value at the same time. Those blocks can be processed in one stage of calculation.

Table 4.1.3.1 MATLAB calulation of forward and backward propagation

Block	Desc	Functions

0	forw.	`z2 = w2*in;`
		`a2 = Sigmoid(z2);`

1	forw.	`z3 = w3*a2;`
		`a3 = Sigmoid(z3);`

2	back.	`err = (out-a3);`
		`s3 = err.*dSigmoid(z3);`

3	back.	`s2 =`
		`(w3'*s3).*Sigmoid(z2);`

4	back.	`o3 = s3*a2';`
		`wn3 = w3 + o3;`

5	back.	`o2 = s2*in';`
		`wn2 = w2 + o2;`

Scheduling

Our design has 255 input, 80 hidden layer perceptrons, and 3 output layer perceptrons. Preventing the excessive amount of logic cell, our system only use 40 processsing units The size of preceptron out scale the number or processing unit. In order to maintain the usability, scheduling are applied to the system.

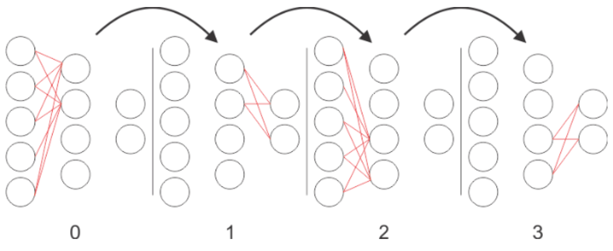


Fig 4.1.3.5 Calcualtion process of stage 0 till stage 3

The above figure illustrates the scheduling process of forward propagation. Every block in table 4.1.3.1 are divided into 2 pieces. This process requires 4 calculation stage, even though only 2 blocks are needed for forward propagation. Stage 0 and stage 2 are basically one calculation block. Since there are 80 perceptrons in the hidden layer and there are only 40 processing units, a2 calculation are divided into 2 separate stages. In stage 0, processing units compute the first 40 data. In stage 2, processing units compute the next 40 data. After stage 0 is done, 40 data of a2 will be used to process a3 value. The next 40 data will be processed right after stage 2 is done.

Above illustation only describes the scheduling of forward propagation. The full cycle of neural network process require forward and backward propagation. Full process of scheduling is described in the table 4.1.3.2

Table 4.1.3.2 Full scheduling (staging) calculation description

Stage	Out	Operation				
		a(24 bit)	*	b (16 bit)	+	x (24 bit)
0	z2[1st]	w2[1st]	*	in	-	-
	a2[1st]	Sigmoid(z2[1st]);				

1	zt3	w3[1st]	*	a2[1st]	+	0
2	z2[2nd]	w2[2nd]	*	in	-	-
	a2[2nd]	Sigmoid(z2[2nd]);				
3	z3	w3[2nd]	*	a2[2nd]	+	zt3
	a3	Sigmoid(z3)				
	Err	out – a3				
	s3	err.*dSigmoid(z3)				
4						
5	r2[1st]	w3[1st]]’	*	s3	-	-
	s2[1st]	r2[1st].*dSigmoid(z2)				
6	wn3[1st]	s3	*	a2[1st]’	+	w3[1st]
7	wn2[1st]	s2	*	in[2nd]'	+	w2[1st]
8	r2[2nd]	w3[2nd]’	*	s3	-	-
	s2[2nd]	r2[2nd].*dSigmoid(z2)				
9	wn3[2nd]	s3	*	a2[1st]’	+	w3[2nd]
10	wn2[2nd]	s2	*	i[2nd]'	+	w2[2nd]
11						

Legend

1st = 1:40

2nd = 41:80

| ▨ | = Iddle (No Process)

| 🟥 | = Weight update (mode_CE=1)

| 🟧 | = Acumulation calculation

| 🟩 | = Systolic array calculation

| 🟨 | = Combinational logic

To sychronize the data between forward and backward propagation, iddle stage inserted between those stage. This idle stage is stage 4 and stage 11.

Output Mode

This section will briefly explain the differences in the output values. Comprehensive description of this mode will be presented in 4.2, section calculation engine. Illustation below shows the calculation process and output data in time manner.

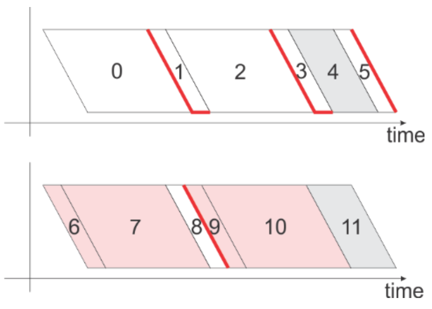


Fig 4.1.3.6 One whole epoch process

Legend

| ▨ | = Idle (No Process)

| 🟥 | = Weight update (mode_CE=1)

| | = Normal mode (mode_CE=0)

---- (red line) = output data

There are 2 modes of calculation, weight update mode and normal mode. Input and output of calculation engine are different on those two mode. While mode_CE=0 the output only discharged on the edge of calculation, either sloped output or staright horixontal ouput. This mode calculates mathematical expression that solved using either skewed calculation or stystolic calulation. Sloped output (red line) indicate that the system is processing skewed accumulation. Straight horizontal line indicates that the system is processing systolic calculation. Figure 4.1.2.2 and 4.1.2.4 may help you to understand the this output behavior. While mode_CE = 1, all data insite region output data that will be fetched by respective memory. In this region, processing units only need single clock operation to process the output.

4.2 Hardware Implementation of Neural Network Core

Here are the hardware architectures which are developed,

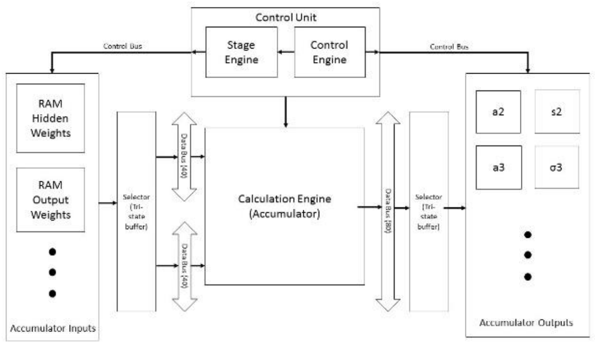


Fig. 4.2.1 COLORIZER FPGA (Neural Network) hardware architecture

The modules in the hardware architecture are,

- Calculation Engine : used to calculate input to become output using sigmoid function
- Stage Engine outputs control signals *epoch*, *stage*, and *step*.
  - Epoch : iteration of the neural network training process
  - Stage : related to mathematical operation

- Step : a counter of each operation (for multiplication operation etc).

- Control engine will control *read enable* and *write enable* signals.

In this image colorization implementation we need to transfer data from PC to FPGA. The image is stored in the internal memory of the FPGA which is called as Embedded Memory (later we would also call it as Block RAM). We used the Embedded Memory to compensate the number of logic elements limited by the FPGA specification. Other than the images, we also stored weight values within the memory.

Based on the neural networks parameters described in the section 2, we have to adjust several hardware parameters so it can be implemented using FPGA. Some constraints that has to be observed in the hardware implementation are:

- Number of registers which will be used
- Number of multiplexers which will be used
- Number of multipliers which will be used
- Number of clock cycles which will be needed

Hardware interconnection illustrated in the figure below.

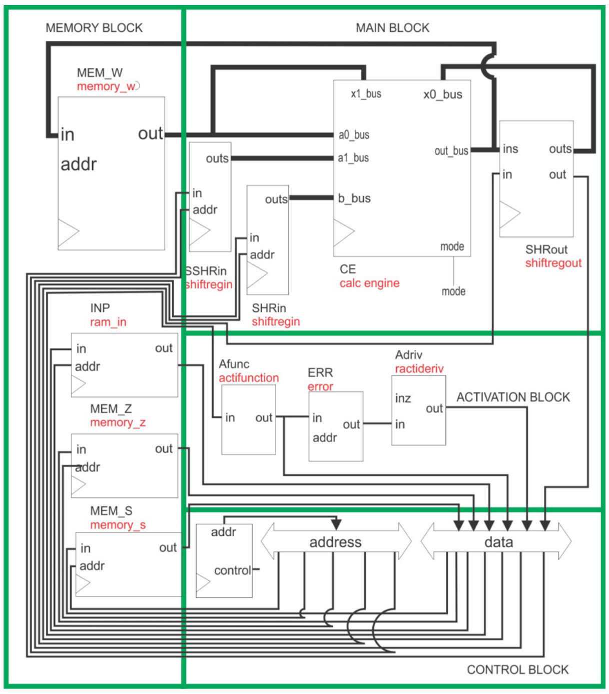


Fig. 4.2.2 More detailed hardware architecture

4.2.1 Main Calculation Core

Main calculation core consists of several modules to perform the main task which is accumulation, systolic calculation, and weight update calculation. The core contains calculation engine, weight memory, and shift registers. The modules are designed to accommodate those three types of calculation. Interconnections between those components are shown in the diagram below.

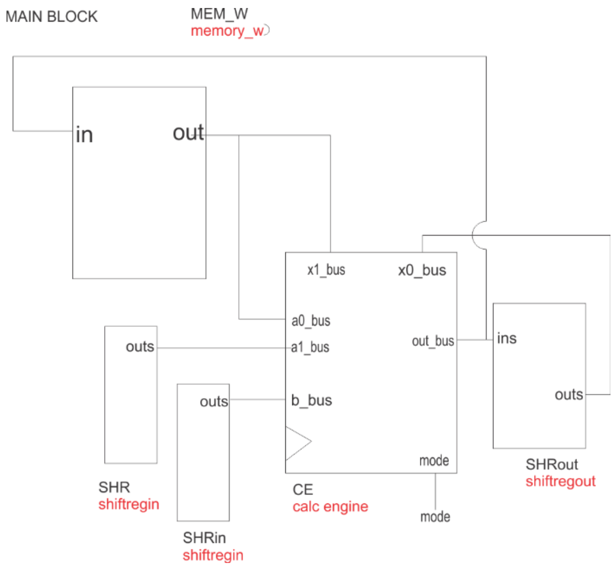


Fig. 4.2.1.1 RTL diagram of main calculation core

To meet the design requirements, every module shall have certain capabilities. List of the capabilities are shown below.

`calcengine`:

- ➢ This module is able to switch between weight update mode and normal mode.
- ➢ This module is able to do one addition and one multiplication in one clock cycle
- ➢ This module is able to match the behaviour of input and output register.

`shiftregin`:

- ➢ This module is able to do register shifting
- ➢ This module is able to hold the shifting mechanism.
- ➢ Data can be input either from the first register or any location within module.

`shiftregout`:

- ➢ This module is able to support systolic array calculation that require moving calculation result.

- This module is able to insert initial values of systolic calculation.
- This module is able to support skewed accumulation by providing feedback on current data on the register.

  This module have output controller to select 1 ouput from its concatenated output bus.

4.2.1.1 MODULE CALCENGINE

calcengine module is the main part of the calculation system. Data calculation that includes addition and multiplication are processed in this module. This module contains 2 parts which are clusters of perceptrons in the calculation engine and controller mode.

Perceptrons process digital signal in two mode. Each mode determines which x input (x0 or x1) and a input (a0 and a1) being processed. The calculation of perceptron is as follow

Mode 0 -> Out = x0 + a0*b

Mode 1 -> Out = x1 + a1*b

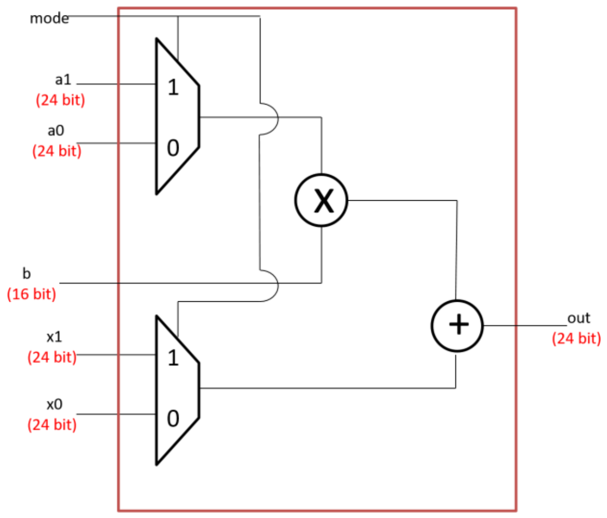


Fig. 4.2.1.1.1 RTL diagram of perceptron module

A perceptron have 5 data wires. Each perceptron data wire has different size. The a0 and a1 data have 16 bit length. The other data wires have 24 bit length. This different size length accommodate the data length requirement as explained in the section 3. In the calcengine module, all inputs and output of perceptrons, a0, a1, b, x0, x1, and out, are concatenated into one big bus named a0_bus, a1_bus, b_bus, x0_bus, x1_bus, and out_bus.

Since the calculation is skewed, both accumulation and systolic calculation, the calcengine have to do shifting. This is done by executing shifting mode, in the same way as before. A shift register is used to control the shifting mode. Full diagram of calcengine module is as follow.

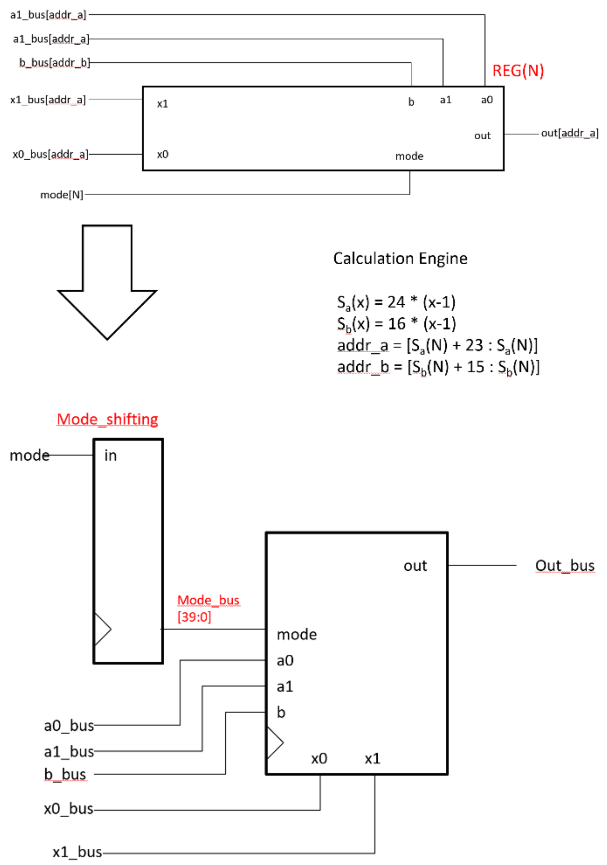


Fig. 4.2.1.1.2 RTL diagram of calcengine module

4.2.1.2 MODULE SHIFTREGIN

shiftregin module contains shift data register, shift control register, and address controller. Shift data register and shift control register functionality are executed by a set of modules called reguintin. And the address control is handled by 6-to-64 decoder.

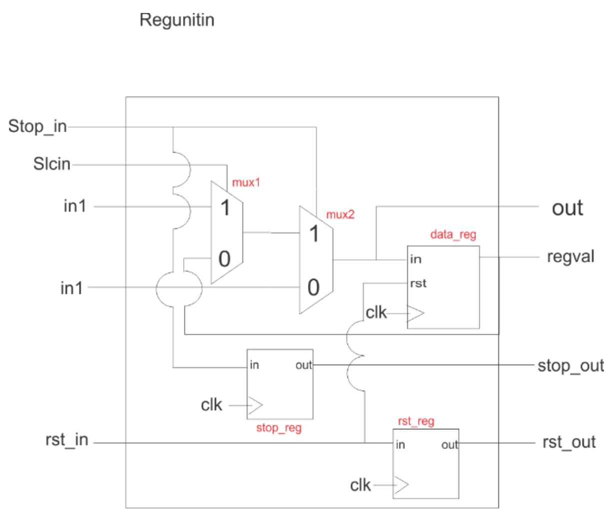
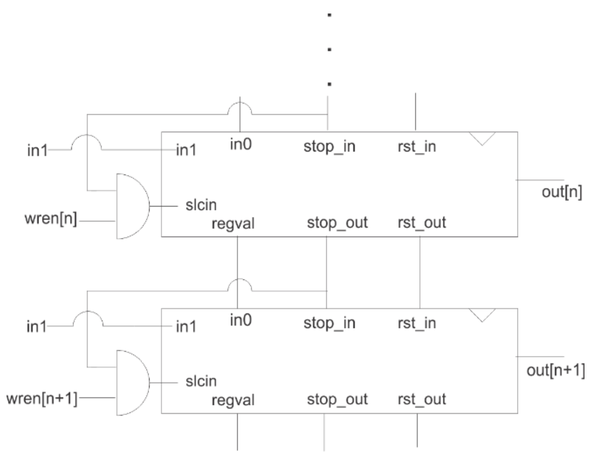


Fig. 4.2.1.2.1 RTL diagram of `regunitin` module

`regunitin` modules have 2 data input (in0 and in1), 2 output (out and regval), shift register for rst control, and several control wire. Control data, stop_out and rst_out, are stored and connected to the next control data register input (stop_in and rst_in). regval wires are connected to the next register. These configurations establish shift register mechanism.

There are two selectors inside the module stop wire and slcin wire. stop wire controls the data shifting mechanism. If the stop wire is high then the shifting procedure will be held. Otherwise, the data register will receive in0 value which has previous module register value. slcin wire controls the input data. If slcin wire is high then the register will be updated by a new input data from wire in1. Otherwise, current register value is fed back to the register so register value remain unchanged. For consideration, in1 will only stored in the register if both stop and slcin wire are high.



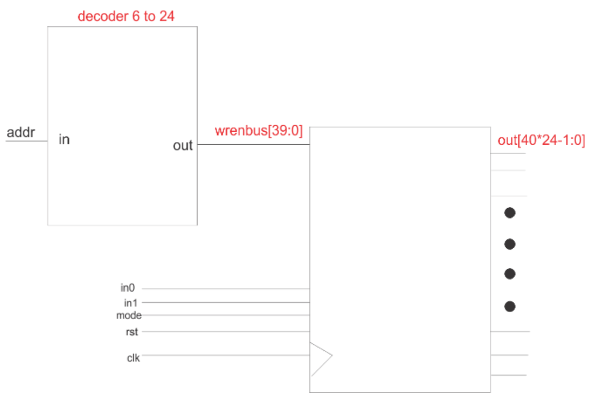


Fig. 4.2.1.2.2 RTL diagram of `shiftregin` module

4.2.1.3 Module `SHIFTREGOUT`

`regunitout` is main component of `shiftregout` module. This module is a simple module that propagate data and reset data from previous module to the next module.

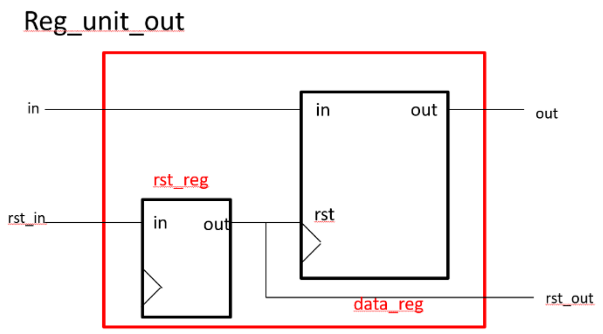


Fig. 4.2.1.3.1 RTL diagram of regunitout module

Similar to `shiftregin` module, `shiftregout` module has 2 mode of operation. Mode = 1 represent systolic calculation, mode = 0 represent skewed calculation. On systolic mode, the output bus are in cross configuration. On skewed accumulation mode, output data are in straight configuration.

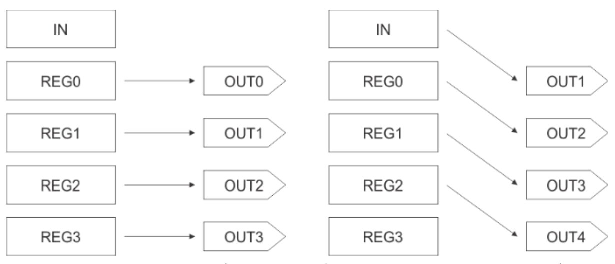
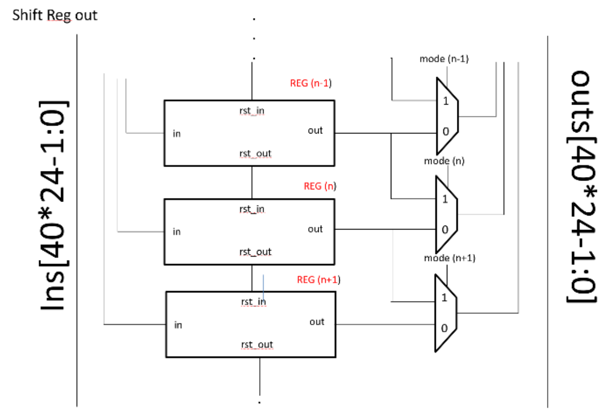


Fig. 4.2.1.3.2 RTL diagram of `shiftregout` module

This module has N bit input as the initial value of systolic calculation, 40*N bit data that are store to the respective register on every clock cycle, N bit output data with addressing control to fetch the current output value, and 40*N bit output for `calcengine` input.



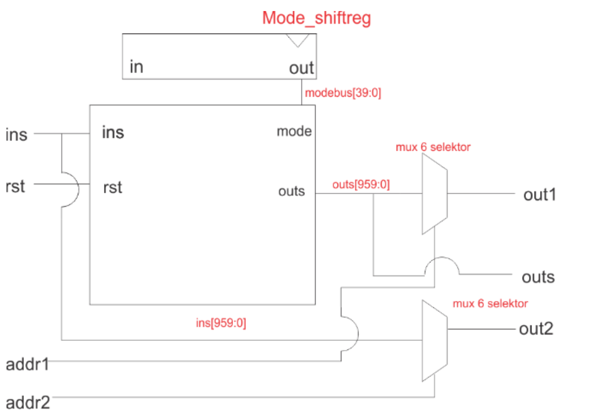


Fig. 4.2.1.3.3 RTL diagram of `shiftregout` module

4.2.2 MEMORY AND ADDRESSING

Weight Memory

Because we did full implementation of neural networks within the FPGA, we should have quite a lot of registers to store only the weight values (225 perceptrons, 80 perceptrons, and 3 perceptrons in the input layer, the hidden layer, and the output layer respectively). It is resource consuming component in FPGA which of has a limited number of logic elements. So, we use RAM to keep the weight values. Block RAM can be implemented using IP (Intellectual Property) which is provided by the software. IP that is used for access the Block RAM is called Altsyncram.

We created blocks of memory with specifications below:

- 40 memory blocks
- depth of 512 words
- 24-bit word length (data width)

However, there are disadvantages in using RAM in FPGA which is the number of port that are limited. Furthermore, the transfer data can only be done in one clock cycle. In order to cope with this problem, we use scheduling so the transfer data and number of clock cycle can be balanced. We needed to synchronize the memory with the scheduling that had been described on 4.1.3 to achieve the most effective hardware implementation. So, we divided each memory block to two partitions. Within each partition there are 256 data (which comes from depth, 512, divided by two). We then use the partition to store one value of weight value, output value, etc. We acually need 225 only to store values. Therefore, after the 225th data, we assign the values to zeroes. Then, we start storing values again after the 255th data, which is the start of second partition. Block RAM implementation is described by diagram below,

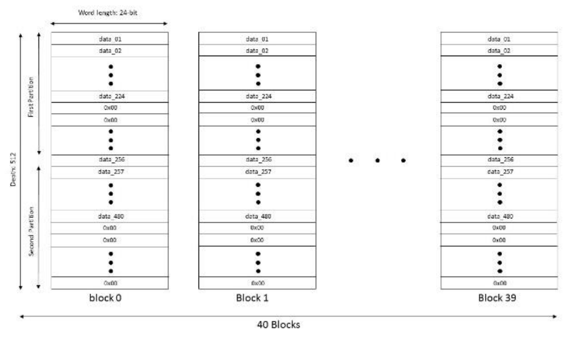


Fig. 4.2.2.1 Diagram of RAM implementation (Look Appendix: Diagrams for more detailed diagram)

S and Z Memory

For the memory mapping, we also do partitioning to achieve the minimum logic elements requirements in the synthesis stage. We divided the address (with the last address being 82) to 3 partitions. The addressing is controlled by a multiplexer and an adder. The multiplexer has a 2-bit selector to select the value to 0, 40, or 80. The value selected then added with read/write address which is 0..39. The memory mapping explanation and the RTL is described by figure below,

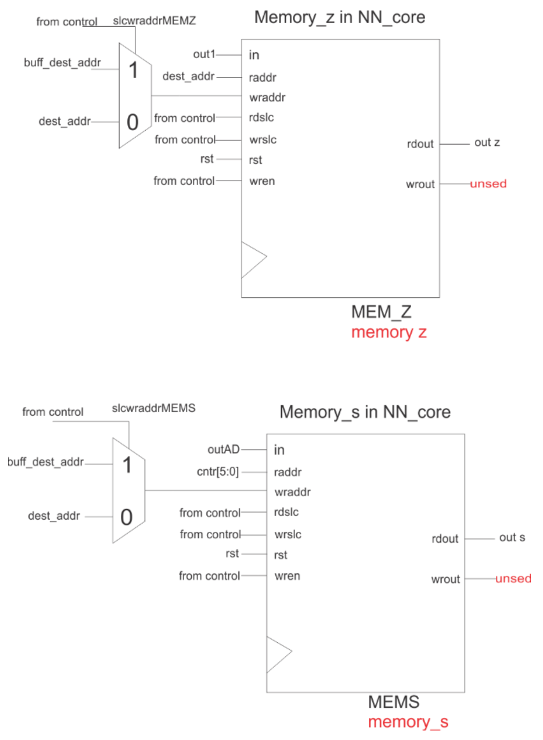


Fig. 4.2.2.2 Input data and output data connection of memory_z and memory_s in top level design

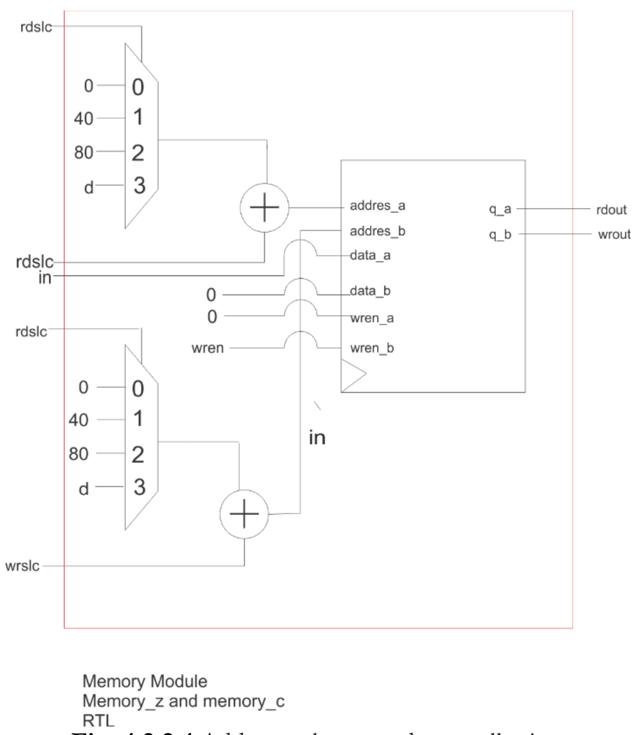


Fig. 4.2.2.4 Address selector and controller in memory_z and memory_s

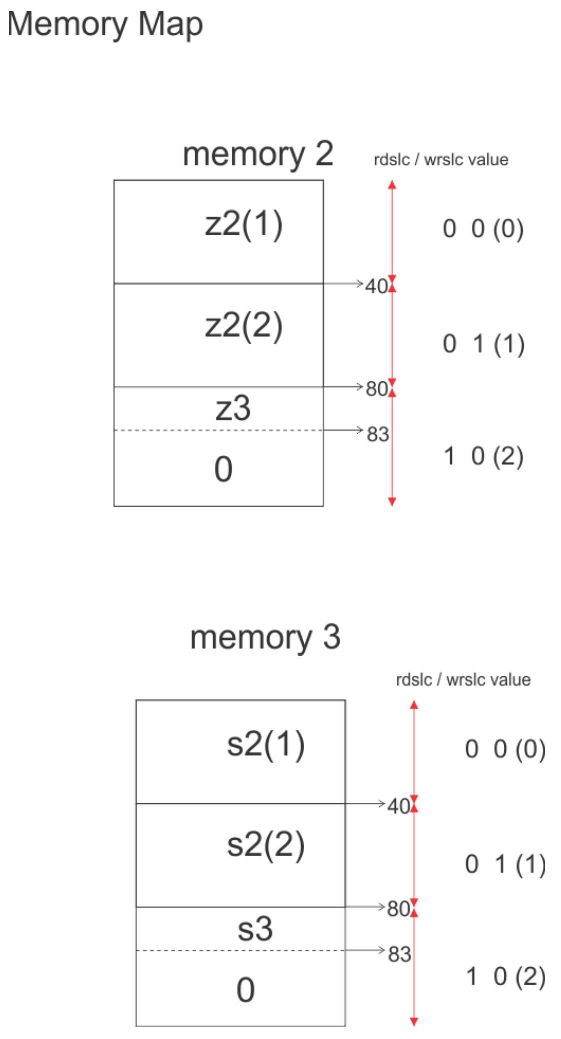


Fig. 4.2.2.3 Address map of memory_z and memory_s

4.2.3 Activation and Error

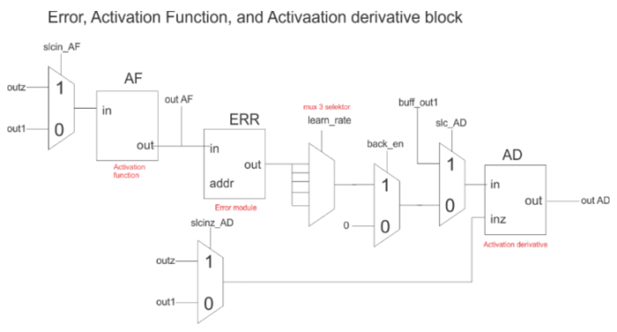


Fig. 4.2.3.1 RTL diagram of atication function, derivative of activation function, and error module.

This module consist of 3 main module which is error activation function module, error function module and derifative of activation function module. This block of function is main calcultion engine to computer the sigmoid approximation.

Error value is adjustable by learning rate selecter. This selector determine the multiplier on a3 to supervisor value. Available learning constant are 2, 1, 1/2, 1/4, 1/8, 1/16, 1/32,and 1/64. Futermore, the error value can be nullified in control of back_en wire. if back_en wire is low then the errror value goes to 0 and wight are remain the same.

4.2.3.1 MODULE ACTIFUNCTION

Writer only describe the implementation of approximation A (gradient 1/4 and 1/8). The implementation of approximation B (gradient 1/4 and 1/16) is quite similar to approximation A.

Activation function module is combinational module with a 32 bit input and a 32 bit output. The data path are shown below.

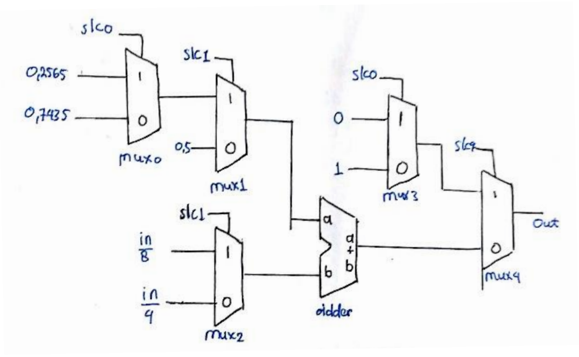


Fig. 1.2.3.1.1 Datapath activation function

This module uses an adder and 5 multiplexers with 3 distinct selector slc0, slc1, and slc4. Value of the selectors are illustrated in the picture below.

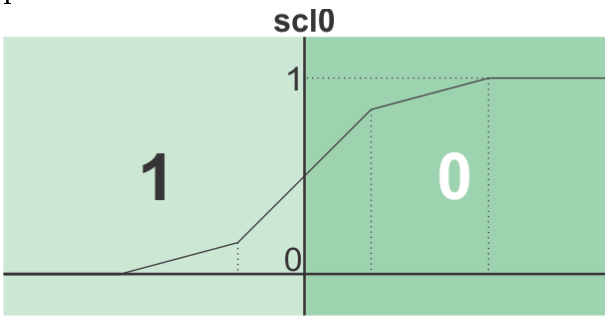


Fig. 2.2.3.1.2 Selector slc0 value

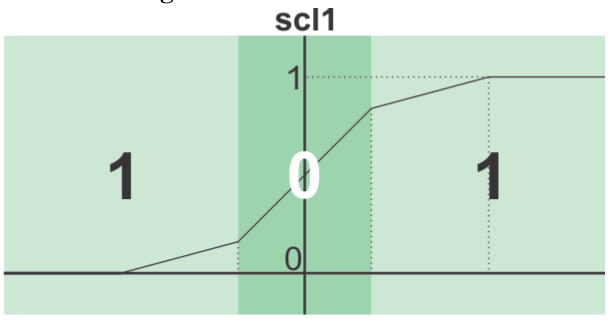


Fig. 3.2.3.1.3 Selector slc1 value

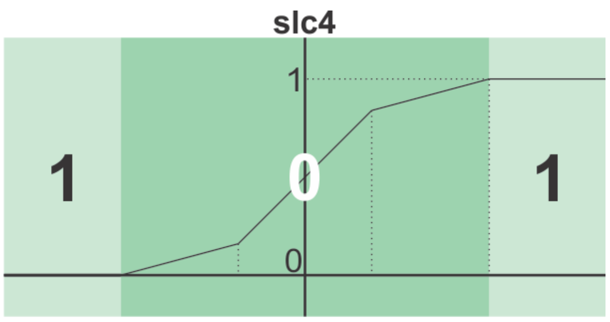


Fig. 4.2.3.1.4 Selector slc4 value

Function of each multiplexer:

- mux0 & mux1 : select appropriate coefficient (c) between 0.2565, 0.5, or 0.7435.
- mux2 : select apropiriate gradient*input (mx) value
- mux3 & mux 4 : detect and select saturation (0 or 1) value.

Output of mux1 is function coefficient either 0.2565, 0.5,or 0.7435. This value is added by value of mux2 output which correspond to mx, gradient times input, value. The output of adder is not the final output value. This value must be evaluated whether the output value is staturated or linear. If the input is in 1st zone or in 5th zone, the output will be 0 or 1 respectively. Otherwise, the output value is equal to output of adder module. Source code is available on appendix.

Module are tested to perform calculation on every zone. The value then cross validated by MATLAB output. The module outputs have exactly same result with MATLAB outputs.

4.2.3.2 MODULE ACTIDERIV

The approximation of the derivative is a discrete function which is :

$$y = \begin{cases} 0 & : x < -3.026 \\ 1/8 & : -3.026 \le x < -0.974 \\ 1/4 & : -0.974 \le x < 0.974 \\ 1/8 & : 0.974 \le x < 3.026 \\ 0 & : x \ge 3.026 \end{cases}$$

The function is a linear function, derivative is only a constant number. Therefore, the implementation of the derivative is a purely combinational circuit (no register needed).

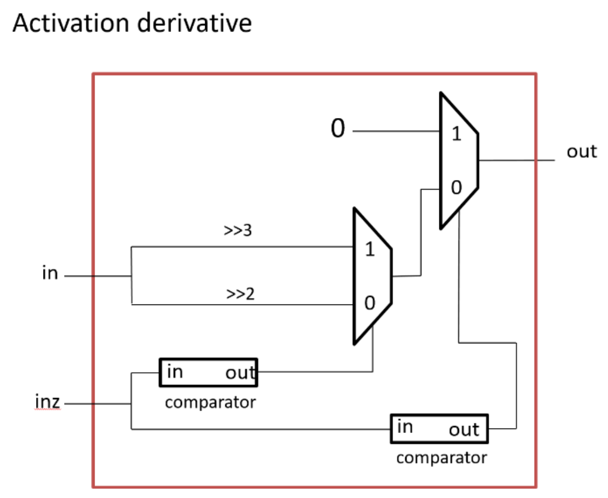


Fig. 4.2.3.2.1 RTL diagram of derivative of activation function

Compared to activation function, this module generate same comparator signal slc1 and slc4, but only perform contact selection either 0, ¼ and 1/8.

4.2.4 Control Engine

All module have control input to manage the activity of certain functionality. Control engine is the main source that produce signal control guiding modules operation. Control engine also works as counter and address generator.

All the signal wires depend on stage position. There are 2 stage position. The first one is starting process position. The other one is a same signal that are delayed by 40 clock cycle, indicating the edge of calculation process.

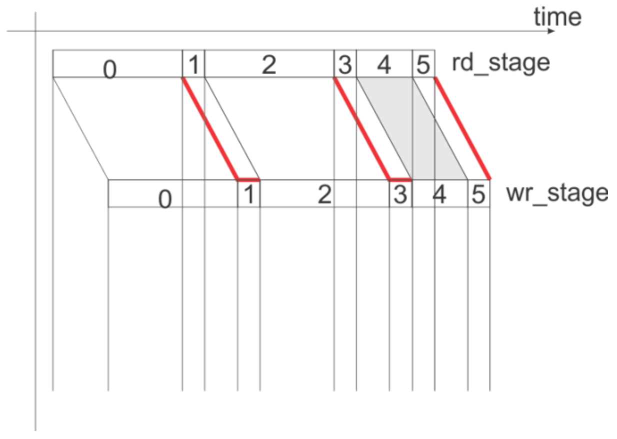


Fig. 4.2.4.1 Example of wr_stage and rd_stage output for first 6 step

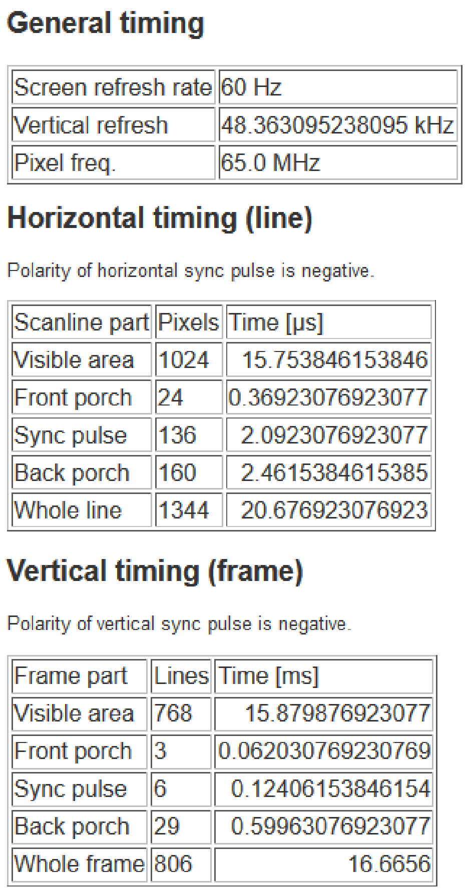
The control signal are generated combinational circuit using the current value of wr_stage dan rd_stage. Full signals control value are attached in appendix.

4.3 Hardware Implementation of Peripheral

4.3.1 VGA Controller

To display the output result to the monitor we built a VGA module in the FPGA. Altera FPGA has a Video ADC ADV7123 that can be used to output display to VGA monitor. The VGA module consists of child modules that are described below,

- Phase-locked loop (PLL) clock
  To generate clock that matches with standard pixel clock that is being used. We use XGA standard (1024x768 @60Hz, with pixel freq. 65.0 MHz) to interface with the VGA monitor.



General timing

Screen refresh rate	60 Hz
Vertical refresh	48.363095238095 kHz
Pixel freq.	65.0 MHz

Horizontal timing (line)

Polarity of horizontal sync pulse is negative.

Scanline part	Pixels	Time [µs]
Visible area	1024	15.753846153846
Front porch	24	0.36923076923077
Sync pulse	136	2.0923076923077
Back porch	160	2.4615384615385
Whole line	1344	20.676923076923

Vertical timing (frame)

Polarity of vertical sync pulse is negative.

Frame part	Lines	Time [ms]
Visible area	768	15.879876923077
Front porch	3	0.062030769230769
Sync pulse	6	0.12406153846154
Back porch	29	0.59963076923077
Whole frame	806	16.6656

Fig 4.3.1.1 VGA Controller RTL Diagram

- VGA Generator
  The module generates VGA control signals for example H_SYNC, Y_SYNC, etc.
- Image Generator
  To specifiy pixel RGB colors at a specific time.

Figure below is the RTL Diagram of the VGA Controller,

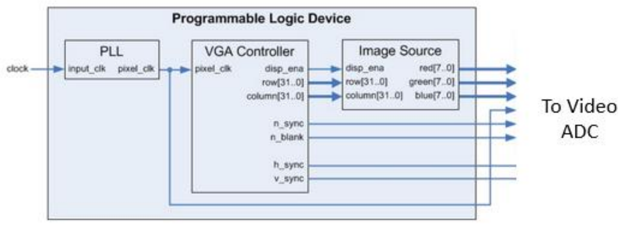
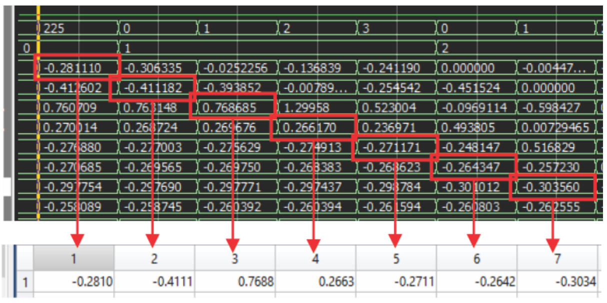


Fig 4.3.1.2 VGA Controller RTL Diagram

5. Result and Analysis

5.1 Neural Network Core Module

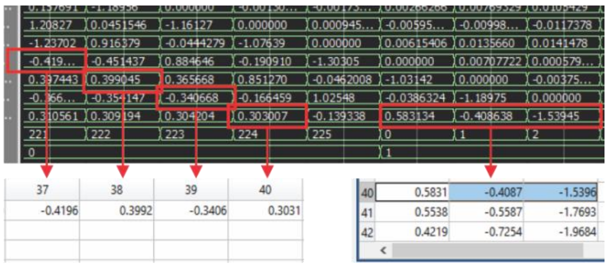
Hardware calculation result are compared to software result from MATLAB. Stages with similar matemahical expresion such as stage 1 and stage 2 are preented once.



```
225        0          1           2          3          0           1
0          1                                            2
-0.281110  -0.306335  -0.0252256  -0.136839  -0.241190  0.000000    -0.00447...
-0.412602  -0.411182  -0.393852   -0.00789...-0.254542  -0.451524   0.000000
0.760709   0.763148   0.768685    1.29958    0.523004   -0.0969114  -0.598427
0.270014   0.268724   0.269676    0.266170   0.236971   0.493805    0.00729465
-0.276880  -0.277003  -0.275629   -0.274913  -0.271171  -0.248147   0.516829
-0.270685  -0.269565  -0.269750   -0.268383  -0.268623  -0.264347   -0.257230
-0.297754  -0.297690  -0.297771   -0.297437  -0.298784  -0.301012   -0.303560
-0.258089  -0.258745  -0.260392   -0.260394  -0.261594  -0.260803   -0.262555
```

	1	2	3	4	5	6	7
1	-0.2810	-0.4111	0.7688	0.2663	-0.2711	-0.2642	-0.3034

Fig. 5.1.1 Result of first 7th data result of 0th stage (calculation engine mode =1, skewed accumulation)



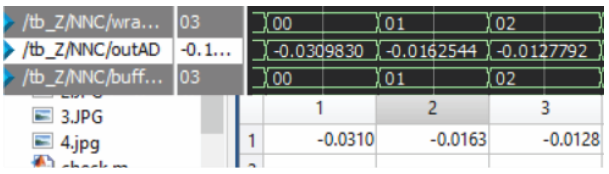
```
1.20827    0.0451546  -1.16127    0.000000   0.000945...  -0.00595...  -0.00998...  -0.0117378
-1.23702   0.916379   -0.0444279  -1.07639   0.000000     0.00615406   0.0135660    0.0141478
-0.419...  -0.451437  0.884646    -0.190910  -1.30305     0.000000     0.00707722   0.000579...
0.397443   0.399045   0.365668    0.851270   -0.0462008   -1.03142     0.000000     -0.00375...
-0.366...  -0.354147  -0.340668   -0.166459  1.02548      -0.0386324   -1.18975     0.000000
0.310561   0.309194   0.304204    0.303007   -0.139338    0.583134     -0.408638    -1.53945
221        222        223         224        225          0            1            2
0                                                         1
```

37	38	39	40
-0.4196	0.3992	-0.3406	0.3031

40	0.5831	-0.4087	-1.5396
41	0.5538	-0.5587	-1.7693
42	0.4219	-0.7254	-1.9684

Fig. 5.1.2 Result of first 37th – 40th data result of stage 0 (calculation engine mode =1, skewed accumulation) and result of stage 1

There are one clock gap between result of stage 0 and stage 1. The control signal shall accommodate this behaviour and prevent that data on disturbing the memory z data.

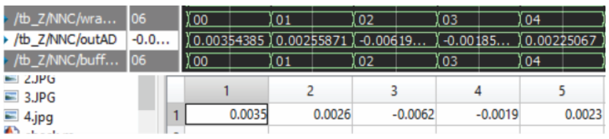


```
/tb_Z/NNC/wra...   03     00          01          02
/tb_Z/NNC/outAD    -0.1...  -0.0309830  -0.0162544  -0.0127792
/tb_Z/NNC/buff...  03     00          01          02
3.JPG
4.jpg
```

	1	2	3
1	-0.0310	-0.0163	-0.0128

Fig. 5.1.3 S3 data result is output of Derivative module in the end of stage 3

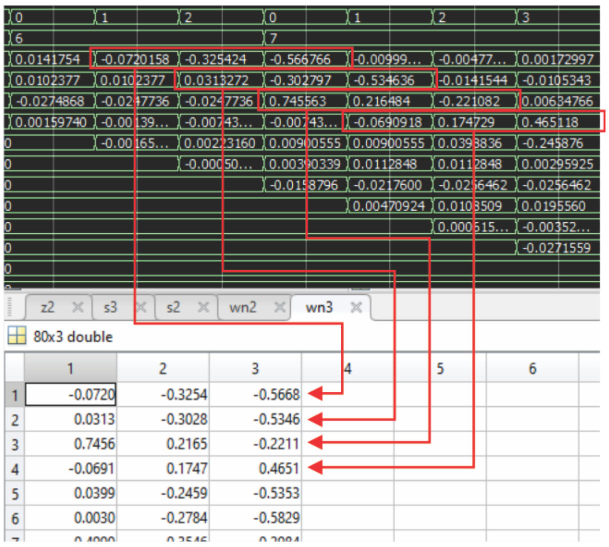
S (s2 and s3) output differ from other module in term of signal location. S signal is located on the output port of module activation derivative.



```
/tb_Z/NNC/wra...   06      00          01          02         03         04
/tb_Z/NNC/outAD    -0.0...  0.00354385  0.00255871  -0.00619...  -0.00185...  0.00225067
/tb_Z/NNC/buff...  06      00          01          02         03         04
3.JPG
4.jpg
```

	1	2	3	4	5
1	0.0035	0.0026	-0.0062	-0.0019	0.0023

Fig. 5.1.4 S2 data result is output of derivative module along sloped line in the stage 5 and stage 8.



```
0           1            2           0            1            2           3
6                                    7
0.0141754   -0.0720158   -0.325424   -0.566766    -0.00999...  -0.00477...  0.00172997
0.0102377   0.0102377    0.0313272   -0.302797    -0.534636    -0.0141544   -0.0105343
-0.0274868  -0.0247736   -0.0247736  0.745563     0.216484     -0.221082    0.00634766
0.00159740  -0.00139...  -0.00743... -0.00743...  -0.0690918   0.174729     0.465118
0           -0.00165...  0.00223160  0.00900555   0.00900555   0.0398836    -0.245876
0                        -0.00050... 0.00390339   0.0112848    0.0112848    0.00295925
0                                    -0.0158796   -0.0217600   -0.0256462   -0.0256462
0                                                 0.00470924   0.0103509    0.0195560
0                                                              0.000615...  -0.00352...
0                                                                           -0.0271559
0
```

z2 | s3 | s2 | wn2 | wn3

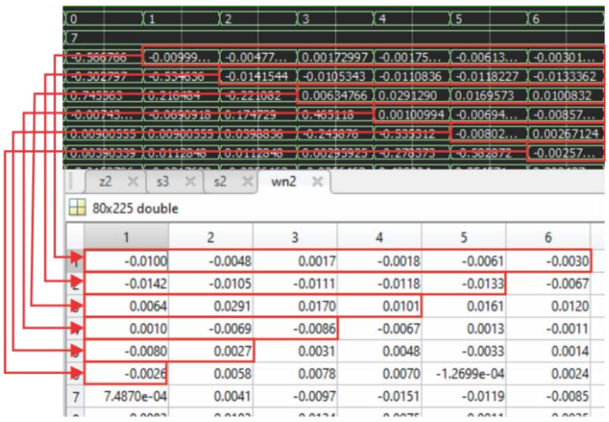
80x3 double

	1	2	3	4	5	6
1	-0.0720	-0.3254	-0.5668			
2	0.0313	-0.3028	-0.5346			
3	0.7456	0.2165	-0.2211			
4	-0.0691	0.1747	0.4651			
5	0.0399	-0.2459	-0.5353			
6	0.0030	-0.2784	-0.5829			

Fig. 5.1.5 New w3 data in stage 6 and 9 (calculation engine mode = 1)

In this image section we are able to see the 0 output regien below the higlited region. That region is iddle phase on stage 4.

While calculation mode is high. Every data within stage region is output data that are directly store on representative memory module.



```
0           1            2            3            4            5            6
7
-0.566766   -0.00999...  -0.00477...  0.00172997   -0.00175...  -0.00613...  -0.00301...
-0.302797   -0.534636    -0.0141544   -0.0105343   -0.0110836   -0.0118227   -0.0133362
0.745563    0.216484     -0.221082    0.00634766   0.0291290    0.0169573    0.0100832
-0.00743... -0.0690918   0.174729     0.465118     0.00100994   -0.00694...  -0.00857...
0.00900555  0.00900555   0.0398836    -0.245876    -0.535312    -0.00802...  0.00267124
0.00390339  0.0112848    0.0112848    0.00295925   -0.278373    -0.582872    -0.00257...
```

z2 | s3 | s2 | wn2

80x225 double

	1	2	3	4	5	6
1	-0.0100	-0.0048	0.0017	-0.0018	-0.0061	-0.0030
2	-0.0142	-0.0105	-0.0111	-0.0118	-0.0133	-0.0067
3	0.0064	0.0291	0.0170	0.0101	0.0161	0.0120
4	0.0010	-0.0069	-0.0086	-0.0067	0.0013	-0.0011
5	-0.0080	0.0027	0.0031	0.0048	-0.0033	0.0014
6	-0.0026	0.0058	0.0078	0.0070	-1.2699e-04	0.0024
7	7.4870e-04	0.0041	-0.0097	-0.0151	-0.0119	-0.0085

Fig. 5.1.5 New w2 data in stage 7 and 10 (calculation engine mode = 1

5.2 Matlab Simulation

Here is the cost function plot of our COLORIZER application using MATLAB,

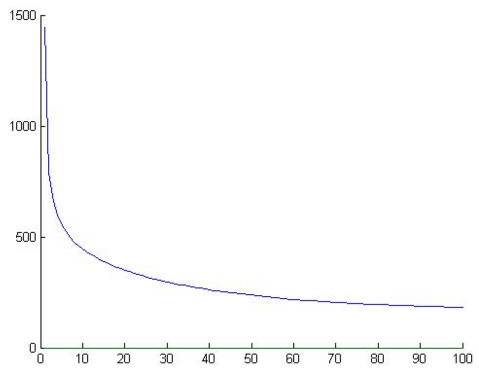


Fig. 5.2.1 Cost Function plot in MATLAB

From the above graph, we can see that the cost function plot has been convergent.

We test the MATLAB simulation of our application using data sets from class participants. The algorithm can be explained by several points below,

1. Initialization of required values.
2. Read input picture file
3. Make training set data of neural network from the data set.
4. Training
5. Doing forward propagation testing.

5.3 Forward Propagation Simulation

From the result above, we can see that the forward propagation has been successfully implemented into hardware and the output is the same with the MATLAB simulation provided in 5.1.

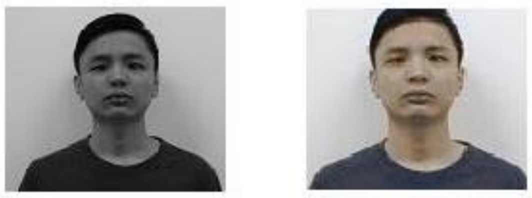


Fig. 5.2.1 Process result with input A

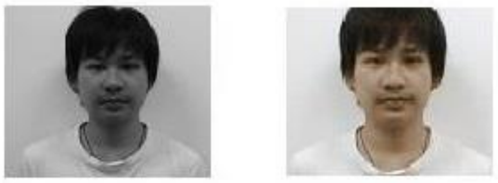


Fig. 5.2.2 Process result with input B

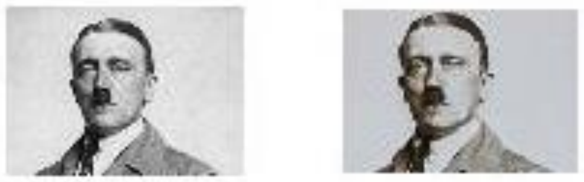


Fig. 5.2.2 Process result with input C

The images above show the result of the overall implementation of our system. Three images are used. Qualitatively, the output result is good. We can see that the grayscale pictures can become RGB picture which is similar to the real picture.

The output result depends on the training images used. If we want the system to colorize an image of a scene for example-as a test image, then one of the images used in the training process must contain pattern of a scene. So, the system can classify the color for each pattern in an image.

Interestingly, the colorization we had done above only used one image as a training set. Image A is used as the training set which will derive many datasets from only one image. After the training process by using that single image, we tested the system with other different images (Image B). The images were colorized. We also tested the system with an originally black and white image (Image C). The image is somewhat colorized, providing that we use only one image as the training set.

With the technique we used, the system can create many training sets only from one image. The system extracts patterns samples from an image to be used as training sets.

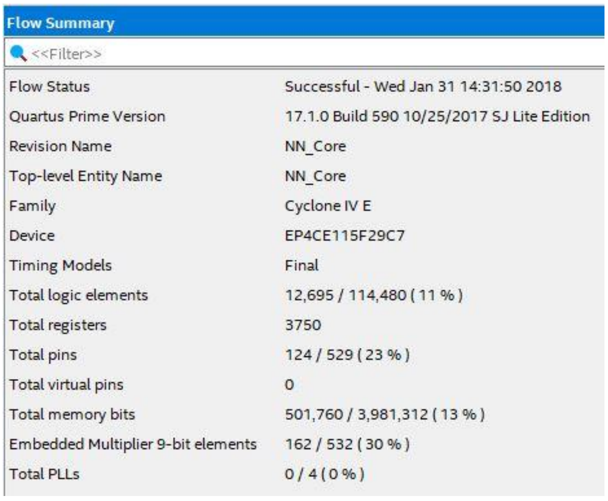
5.4 Hardware Compilation and Performance

The summary of the NN_core hardware implementation is shown in the table below.

Hardware Utilization

Table 5.4.1 Hardware utilization

	Used	Available	%
Number of Logic Elements	12,695	114,480	11
Number of Registers	3750	28,800	13%
Number of fully used LUT-FF pairs	0	23,109	0
Number of Memory bits	501,760	3,900,000	13%
Embeder Multiplier 9 bit	162	532	30
PLLs	0	4	0


Flow Summary

<<Filter>>

Flow Status	Successful - Wed Jan 31 14:31:50 2018
Quartus Prime Version	17.1.0 Build 590 10/25/2017 SJ Lite Edition
Revision Name	NN_Core
Top-level Entity Name	NN_Core
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	12,695 / 114,480 (11 %)
Total registers	3750
Total pins	124 / 529 (23 %)
Total virtual pins	0
Total memory bits	501,760 / 3,981,312 (13 %)
Embedded Multiplier 9-bit elements	162 / 532 (30 %)
Total PLLs	0 / 4 (0 %)

Fig. 5.4.1 Hardware compilation

Utilization of ram memory reduce the register and total logic element in significantly. As explained before, this setting only compatible on skewed accumulation and systolic array calculation.

Worst case delay

Execution setting on worst case delay simulation is as follow

Type : Base
Period : 1ns
Frequencty: 1000.0Mhz
Rise= 0.00
Fall= 0.500

Result of simulation:
Worst-case setup slack = =11.912 ns
Worst-case hold slack =0.128 ns
In this setting total worst-case setup delay = 12.912

And total, worst-case hold delay = 0.872.

Throughput

Clock cycle needed for every stage are shown in the table below.

Table 5.4.2 Scate clock cycle

Stage	Total Operation	Clock	desc
0	225x40	226	Reset
1	40x3	41	Reset
2	225x40	226	Reset
3	40x3	41	Reset
4	0	40	-
5	3x40	4	Reset
6	40x3	3	No Rst
7	225x40	225	No Rst
8	3x40	4	Reset
9	40x3	3	No Rst
10	225x40	225	No Rst
11	0	40	-
Tot.		1078	

To process one cycle of forward and backward propagation the system need 1078 clock cycle.

6. Conclusion

- Our COLORIZER System is able to colorize grayscale image with only 1 training set. The system can colorize the grayscale image to RGB image
- The COLORIZER system can be implemented using ALTERA DE2-115. Simulation result of the ANN module has given the same result between hardware calculation and MATLAB calculation.
- Our COLORIZER system has worst-case hold delay 0,872 s with one cycle of forward and backward propagation need 1078 clock cycle.

References

[1] http://www.lsi-contest.com/shiyou_1e.html, accessed on 27 December 2017

[2] Adiono, Trio. *Perancangan Sistem VLSI* (VLSI Systems Design), Pusat Mikroelektronika ITB, Bandung, 2012.

[3] Ciletti, M.D. *Advanced Digital Design With the Verilog HDL*, Publishing House of Electonics Industry, Beijing, 2010.

[4] http://tinyvga.com/vga-timing, accessed in 21 January 2018

[5] Altera Internal Memory (RAM and ROM), User Guide

Appendix: Diagrams

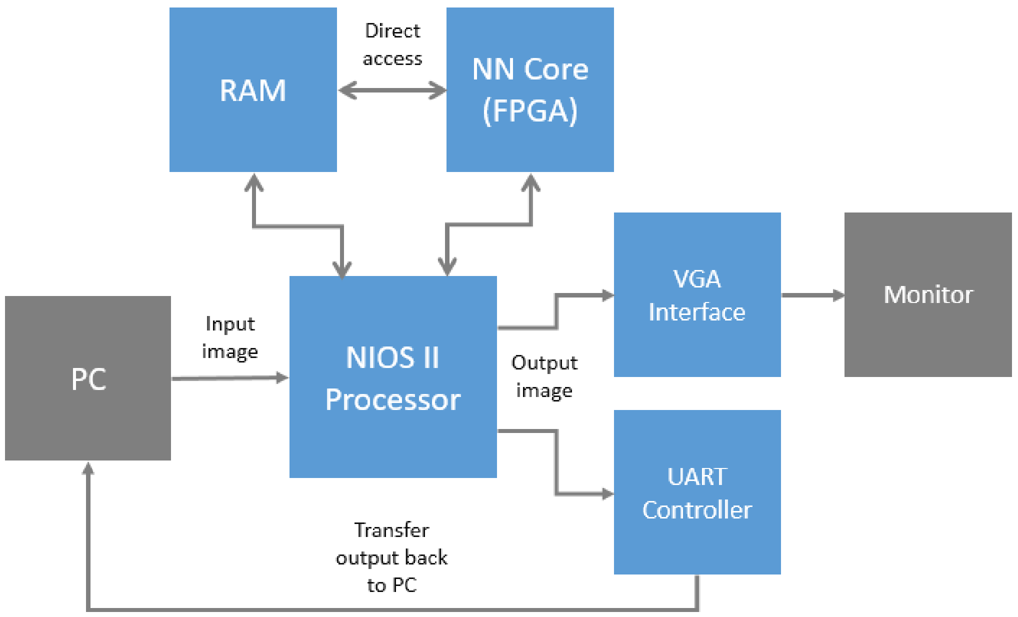


Fig 4.1.1.1 System Architecture of COLORIZER System

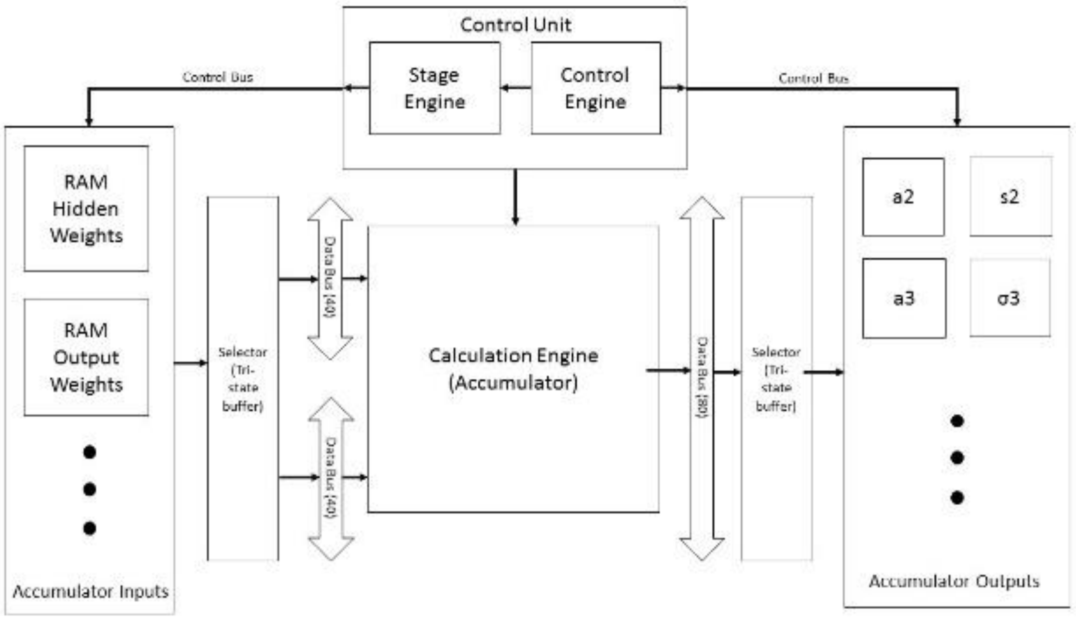


Fig. 4.2.1 COLORIZER FPGA (Neural Network) hardware architecture

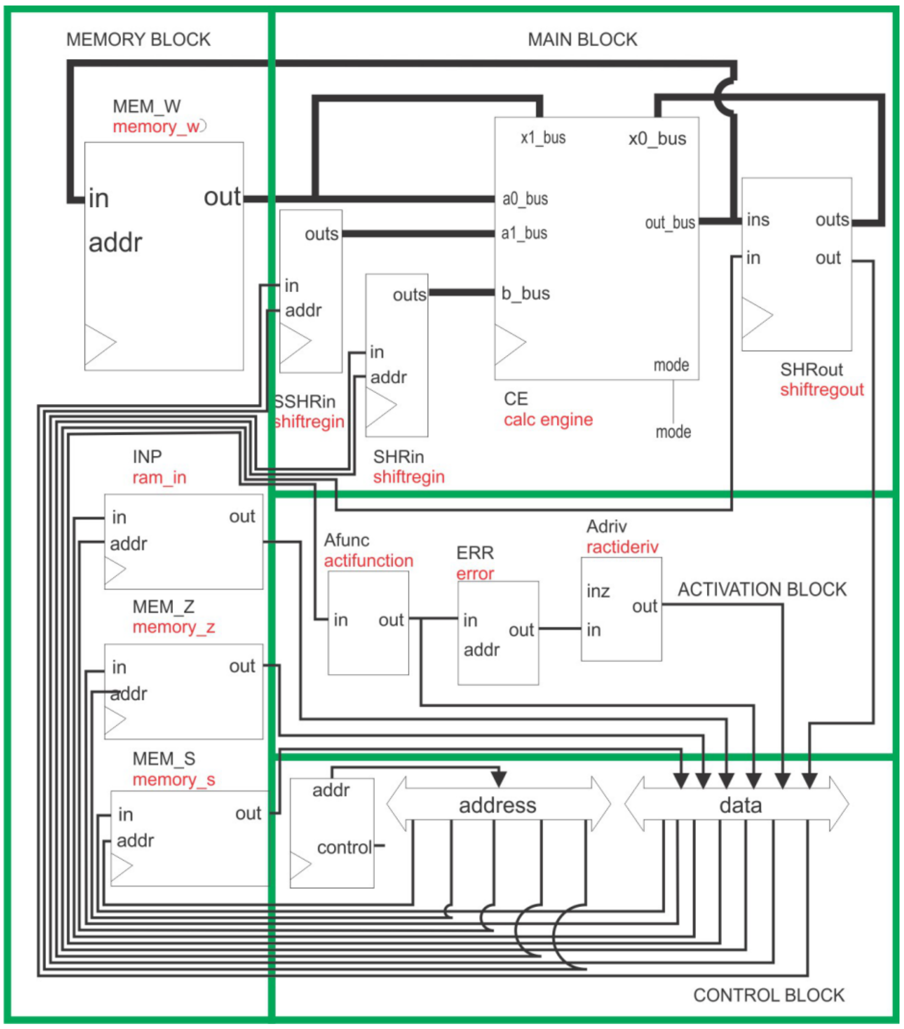


Fig. 4.2.2 More detailed hardware architecture

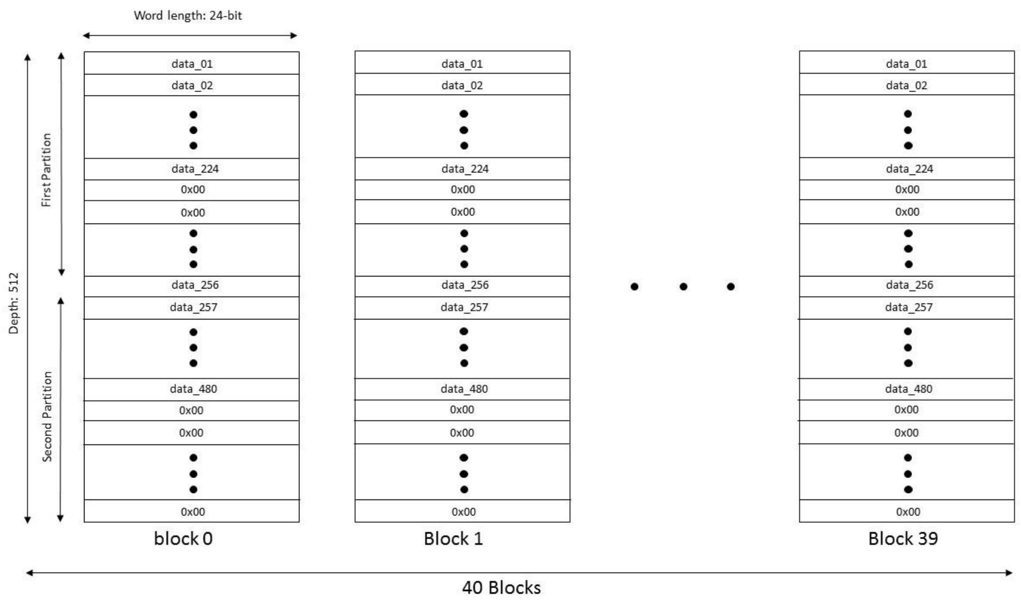


Fig. 4.2.2.1 Diagram of RAM implementation

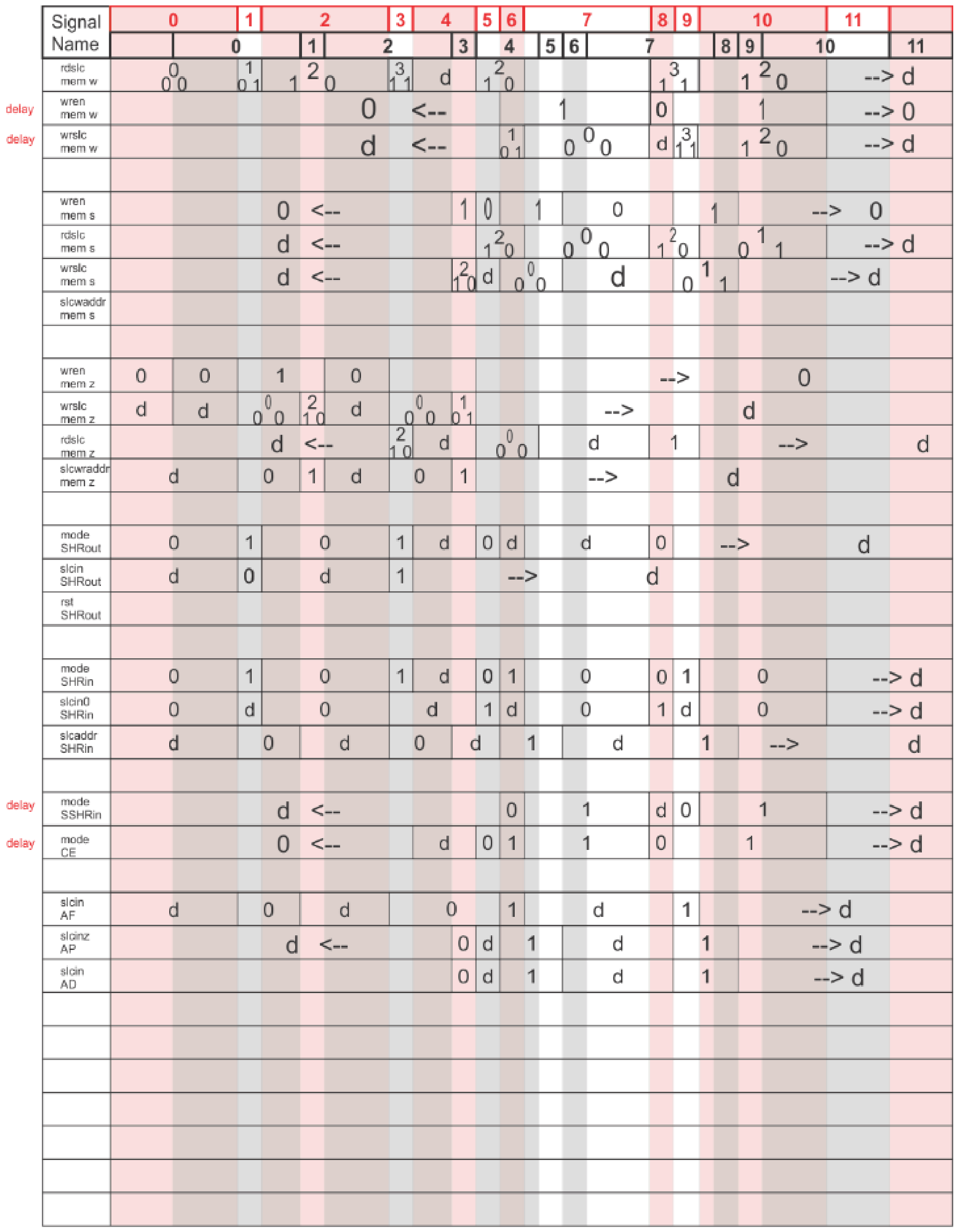


Figure Control signals used in the NN Core

Appendix: MATLAB Code

MATLAB simulation code of COLORIZER,

main.m

```
%MAIN FUNCTION - Run fungsi utama main() ini untuk menjalankan program
%
% Inputs:
%    input training set - sub-image sliding window 15x15, sampel 9116 buah
%    input test set - gambar grayscale dengan ukuran bebas
%
% Target:
%    Target training set - gambar berwarna dengan ukuran 120x100
%
% m-files dibutuhkan: main.m, train.m, logisticSigmoid.m, dLogisticSigmoid.m
% MAT-files tersedia: nilai-nilai pre-trained hidden weights dan output
weights
%
% Nama anggota: 13214008 Naufal Ridho H, 13214012 Fransiskus Yoga Esa,
13214034 Ferriady Setiawan
% EL4138 Perancangan Sistem VLSI
% Institut Teknologi Bandung, Teknik Elektro
% Tanggal: 8-December-2017

%------------- CODE --------------

clear all; close all; clc
%%%% Paramet nerual network  dan data%%%%%%%%%s
% Menentukan jumlah hidden units
numberOfHiddenUnits = 80;
% Menentukan learning rate
learningRate = 0.1;
% Menentukan fungsi activation function yang digunakan
activationFunction = @logisticSigmoid    %(x) 1./(1+exp(-x));
dActivationFunction = @dLogisticSigmoid  %(x) 1./(2+exp(x)+exp(-x));
% Menentukan batch size dan epoch yang digunakan
batchSize = 9116;
epochs = 100;
%%menentukan besar kernel input
h_in = 15;
w_in = 15;

fprintf('%d hidden units.\n', numberOfHiddenUnits);
fprintf('Learning rate: %d.\n', learningRate);

%%%% Meng-load Dataset Wajah %%%%%%%%%%%
% Membuat data training %%
% Membaca gambar
color_image = imread('1.jpg');
gray_image = rgb2gray(color_image);
[h_img w_img]= size(gray_image);
% Menghitung jumlah pixelyang valid
valid_x = w_img-w_in+1;
valid_y = h_img-h_in+1;
%inisialisasi data input dan supervisor
train_in = [];
train_out = zeros(3,valid_x*valid_y);
for i=1:valid_x
    for j=1:valid_y
        temp = gray_image([j:j+h_in-1],[i:i+w_in-1]);
        train_idx = (i-1)*valid_y+j;
        train_in = [train_in;temp(:)'];
        pos_x = uint16(i+(w_in+1)/2);
        pos_y = uint16(j+(h_in+1)/2);
        train_out(1,train_idx) = color_image(pos_y,pos_x,1);
        train_out(2,train_idx) = color_image(pos_y,pos_x,2);
        train_out(3,train_idx) = color_image(pos_y,pos_x,3);
    end
end
train_in = im2double(train_in');
train_out = double(train_out);
train_out = train_out/255;

%%% Prosees training %%%%%%%%%%%%%%%%%%
[hiddenWeights, outputWeights] = train(activationFunction,
dActivationFunction, ...
      numberOfHiddenUnits, train_in, train_out, epochs, batchSize,
learningRate);

%%% Menyimpan workspace MAT %%%
% Untuk menyimpan hidden weights dan output weights %
% sehingga tidak perlu melakukan training ulang nantinya %
filename = 'data_sigmoidpartial.mat';
save(filename);
% % load hidden weights dan output weights yang telah ditrain
% close all; clear all; load('data_sigmoidpartial.mat');

%%% Proses Testing %%%%%%%%%%%%%%%%%%%%
% Membuat data testing %%
% Membaca gambar
test_image = imread('4.jpg');
test_gray = rgb2gray(test_image);
test_gray = double(test_gray)/255;
[h_imgtest w_imgtest]= size(test_gray);
% Menghitung jumlah pixel yang valid
valid_x_test = w_imgtest-w_in+1;
valid_y_test = h_imgtest-h_in+1;
%inisiasi data input dan supervisor
test_out = ones(valid_y_test,valid_x_test,3);
for i=1:valid_x_test
    for j=1:valid_y_test
        temp = test_gray([j:j+h_in-1],[i:i+w_in-1]);
        temp = double(temp(:));
        %forward
        hiddenActualInput = hiddenWeights*temp;
        hiddenOutputVector = activationFunction(hiddenActualInput);
        outputActualInput = outputWeights*hiddenOutputVector;
        outputVector = activationFunction(outputActualInput);
        test_out(j,i,:) = outputVector;
    end
end

% test_out_hsv = rgb2hsv(test_out);
%test_out_hsv(:,:,2) = 1.5*test_out_hsv(:,:,2)
% test_out_rgb = hsv2rgb(test_out_hsv);
%test_out_rgb(:,:,3) = 0.5*test_out_rgb(:,:,3);
image8Bit = uint8(255 * test_out);
figure;
subplot(1,4,1), imshow(test_image);
subplot(1,4,2), imshow(test_gray);
subplot(1,4,3), imshow(test_out);
subplot(1,4,4), imshow(image8Bit);
```

train.m

```
function [hiddenWeights, outputWeights] = train(activationFunction,
dActivationFunction, numberOfHiddenUnits, inputValues, targetValues, epochs,
batchSize, learningRate)

    % Ukuran matriks trainingSet, input, dan output
    trainingSetSize = size(inputValues, 2);
    inputDimensions = size(inputValues, 1);
    outputDimensions = size(targetValues, 1);

    % Inisialisasi hidden weights dan output weights
    % dengan nilai random yang ternormalisasi antara 0..1
    hiddenWeights = rand(numberOfHiddenUnits, inputDimensions);
    outputWeights = rand(outputDimensions, numberOfHiddenUnits);

    hiddenWeights = hiddenWeights./size(hiddenWeights, 2);
    outputWeights = outputWeights./size(outputWeights, 2);

    figure; hold on;
    costgraph = zeros(epochs);
    for t = 1: epochs
        temp = 0;
        for n = 1: batchSize
            % Forward propagate
            inputVector = inputValues(:, n);
            hiddenActualInput = hiddenWeights*inputVector;
            hiddenOutputVector = activationFunction(hiddenActualInput);
            outputActualInput = outputWeights*hiddenOutputVector;
            outputVector = activationFunction(outputActualInput);

            targetVector = targetValues(:, n);

            % Backpropagation
            outputDelta =
dActivationFunction(outputActualInput).*(outputVector - targetVector);
            hiddenDelta =
dActivationFunction(hiddenActualInput).*(outputWeights'*outputDelta);

            outputWeights = outputWeights -
learningRate.*outputDelta*hiddenOutputVector';
            hiddenWeights = hiddenWeights -
learningRate.*hiddenDelta*inputVector';
            temp = temp + sum(abs(outputVector - targetVector));
        end;
        fprintf('Epoch: ');
        disp(t);
        costgraph(t)=temp;
    end;
    plot(costgraph);
end
```

logisticSigmoid.m

```
function out = logisticSigmoid(in)
% logisticSigmoid(). Logistic sigmoid activation function
% dengan pendekatan partial linearization
%
% INPUT:
% x      : Input vector.
%
% OUTPUT:
% y      : Output vector where the logistic sigmoid was applied element by
% element.
%
  x1 = 0.974;
  y1 = 0.1218;
  x2 = 3.0260;

  %% Output
  out = in/4 - (in/8 - y1).*(in>x1) - (in/8 - 0.5 + y1).*(in>x2) ...
      - (in/8 + y1).*(in<-x1) - (in/8 + 0.5 - y1).*(in<-x2);
  out = out +0.5;
end
```

dLogisticSigmoid.m

```
function out = dLogisticSigmoid(in)
% dLogisticSigmoid(). Derivatif dari logistic sigmoid.
%
% INPUT:
% x      : Input vector.
%
% OUTPUT:
% y      : Output vector
%
  x1 = 0.974;
  y1 = 0.1218;
  x2 = 3.0260;
  out = 0.125*((in<x1)&(in>-x1)) + 0.125*((in<x2)&(in>-x2));
end
```

Appendix: Verilog Code

RTL code of `NN_Core` module, which is the top-level of the NN Core block (as illustrated in the figure 4.1.1.1 ,

```
////////////////////////////////////////////////////////////////////////////////
/
 //
 // Module: NN_Core.v
 // Project: COLORIZER
 // Description: Top-level of the NN Core block
 //
 // Last updated: 2//2018
 //

////////////////////////////////////////////////////////////////////////////////
/
module NN_Core(
    input [15:0] data,      //15x15 pixel input data
    input [7:0] wraddress, //address of input data
    input wren,            //enable data input
    input [15:0] sprdata,  //3 supervisor value
    input [1:0] spraddress,//address of supervisor value
    input spren,           //enable dupervisor data input
    input [2:0]learn_rate, //learning rate = 2^(2-learn_rate)
    input back_en,         //1=forward and backward, 0 >forward only
    output status,         //0=forward prop is being processed, 1=back prop
    output request_in,     //inputing datas are only allowed if request_in = 1
    output [15:0] q0,      //1'st output data
    output [15:0] q1,      //2'nd output data
    output [15:0] q2,      //3'rd output data
    output [23:0] cost,    //output cost value
    input rst,             //reset signal
    input clk);            //clock signal


  /**********LOCAL PARAMETER**********/
  localparam N= 40;
  localparam Ba= 24;
  localparam Bb= 16;
  localparam Bm=6;
  localparam Bcntr=8;

  /**********DATA WIRES**********/
  //Input data
  wire [Bb-1:0] in;
  //Main data bus
  wire [N*Ba-1:0] a_bus,x_bus,out_bus;
  wire [N*Bb-1:0] b_bus;
  wire [N*Ba-1:0] s_bus;
  //Read and wire address signa;
  wire [Bm-1:0] source_addr,dest_addr;
  wire [Bcntr-1:0] cntr;
  //SHRin wire
  wire [Bb-1:0] in0_SHRin;
  wire [Bm-1:0] addr_SHRin;
  //SHRout wire
  wire [Ba-1:0] out1;
  wire [Ba-1:0] in_SHRout;
  //Memoru Z and S output wire
  wire [Ba-1:0] outZ,outS;
  wire [Bm-1:0] wraddr_MEMZ,wraddr_MEMS;
  //Activation function wire
  wire [Ba-1:0] inAF,inzAD;
  wire [Bb-1:0] outAF;
  //Activation derivation and Error wire
  wire [Bb-1:0] out_ERR;
  wire [Ba-1:0] learn_ERR,final_ERR;
  wire [Ba-1:0] inAD,outAD;
  //Buffered data
  wire [Bm-1:0] buff_dest_addr;
  wire [Ba-1:0] buff_outAD, buff_out1;
  wire [Bcntr-1:0] buff_cntr;



  /**********CONTROL SIGNAL WIRES**********/
  wire [1:0] rdslc_MEMW,wrslc_MEMW;  //MEMW
  wire wren_MEMW;
  wire wren_MEMZ,slcwraddr_MEMZ;     //MEMZ
  wire [1:0] wrslc_MEMZ, rdslc_MEMZ;
  wire wren_MEMS, slcwraddr_MEMS;    //MEMS
  wire [1:0] wrslc_MEMS, rdslc_MEMS;
  wire mode_SHRin,slcin0_SHRin;      //SHRin
  wire slcaddr_SHRin;
  wire rst_SHRout,mode_SHRout;       //SHRout
  wire slcin_SHRout;
  wire mode_SSHRin;                  //SSHRin
  wire mode_CE;                      //CE Calculation engine
  wire slcin_AF;                     //AF Activation function
  wire slcin_AD,slcinz_AD;           //AD Derivatife function
  wire clear_ERR,stop_ERR;            //ERR (Error function)
  //edit for further devp
  wire wren_OU;

  /**********MAIN PROCESSING**********/
  //Input shift register
  shiftregin #(Bb) SHRin(
    .in0(in0_SHRin),.in1(outAF),.out(b_bus),.addr(addr_SHRin),     //edit for
further devp (addr)
    .mode(mode_SHRin),.rst(rst),.clk(clk));
  assign in0_SHRin = slcin0_SHRin ? outS[20:5] : in;
  assign addr_SHRin = slcaddr_SHRin ? buff_dest_addr : dest_addr;
  //output shift register
  shiftregout #(Ba) SHRout(
    .ins(out_bus),.in(in_SHRout),
    .outs(x_bus),.out1(out1),.out2(),
    .addr1(source_addr),.addr2({Bm{1'b0}}),
    .mode(mode_SHRout),.rst(rst_SHRout),.clk(clk));
  assign in_SHRout = slcin_SHRout ? outZ : {Ba{1'b0}};

  //shift register for memory s data
  shiftregin #(Ba) SSHRin( //MEMS Shift reg in
    .in0(outS),.in1(outS),.out(s_bus),.addr(buff_cntr[5:0]),   //edit for
further devp (addr)
    .mode(mode_SSHRin),.rst(rst),.clk(clk));

  //Main DSP cluster
  calcengine CE(
    .a0_bus(a_bus),.a1_bus(s_bus),.b_bus(b_bus),
    .x0_bus(x_bus),.x1_bus(a_bus),.out_bus(out_bus),
    .mode(mode_CE),.rst(rst),.clk(clk));

  /**********ACTIVATION FUNCTION**********/
  //forward function
  actifunction AFunc(
    .in(inAF),.out(outAF));
  assign inAF = slcin_AF ? outZ : out1;
  //edit for further devp
  //derivation function
  actideriv ADeriv(
    .in(inAD),.z(inzAD),.out(outAD));
  assign inAD =  slcin_AD ? buff_out1 : final_ERR;
  assign inzAD = slcinz_AD ? outZ : out1;

  /********** BUFFERED DATAS***********/
  buffer #(Ba) BUFF_AD(
    .in(outAD),.out(buff_outAD),.rst(rst),.clk(clk));
  buffer #(Ba) BUFF_out1(
    .in(out1),.out(buff_out1),.rst(rst),.clk(clk));
  buffer #(Bm) BUFF_dest(
    .in(dest_addr),.out(buff_dest_addr),.rst(rst),.clk(clk));
  buffer #(Bcntr) BUFF_cntr(
    .in(cntr),.out(buff_cntr),.rst(rst),.clk(clk));

  /**********LEARNING RATE AND ERROR**********/
  //Error to supervisor value
  error ERR(
    .in(outAF),.rdaddr(buff_dest_addr[1:0]),.out(out_ERR),
    .data(sprdata), .wraddr(spraddress),.wren(spren),
    .cost(cost),.clear(clear_ERR),.stop(stop_ERR),
    .rst(rst),.clk(clk));
  //Learning rate choose
  mux8in #(Ba) MUXLearn(
    .in0({{1{out_ERR[15]}} , out_ERR , {7{1'b0}}}),
    .in1({{2{out_ERR[15]}} , out_ERR , {6{1'b0}}}),
    .in2({{3{out_ERR[15]}} , out_ERR , {5{1'b0}}}),
    .in3({{4{out_ERR[15]}} , out_ERR , {4{1'b0}}}),
    .in4({{5{out_ERR[15]}} , out_ERR , {3{1'b0}}}),
    .in5({{6{out_ERR[15]}} , out_ERR , {2{1'b0}}}),
    .in6({{7{out_ERR[15]}} , out_ERR , {1{1'b0}}}),
    .in7({{8{out_ERR[15]}} , out_ERR}),
    .out(learn_ERR),.slc(learn_rate));
  //If backward is not active than error = 0
  assign final_ERR = back_en ? learn_ERR : {Ba{1'b0}};

  /**********ALL MEMORY COMPONENT**********/
  //Weight memory bank
  memory_w MEMW(
    .in(out_bus),.out(a_bus),.rdaddr(cntr),
    .wraddr(buff_cntr),.rdslc(rdslc_MEMW),.wrslc(wrslc_MEMW),
    .rst(rst),.wren(wren_MEMW),.clk(clk));
  //Z memory
  memory_z MEMZ(
    .in(out1),.rdout(outZ),.wrout(),
    .rdaddr(dest_addr),.rdslc(rdslc_MEMZ),
    .wraddr(wraddr_MEMZ),.wrslc(wrslc_MEMZ),.wren(wren_MEMZ),
    .clk(clk));
  assign wraddr_MEMZ = slcwraddr_MEMZ ? buff_dest_addr : dest_addr;
  //S memory
  memory_s MEMS(
    .in(outAD),.rdout(outS),.wrout(),
    .rdaddr(cntr[5:0]),.rdslc(rdslc_MEMS),
    .wraddr(wraddr_MEMS),.wrslc(wrslc_MEMS),.wren(wren_MEMS),
    .clk(clk));
  assign wraddr_MEMS = slcwraddr_MEMS ? buff_dest_addr : dest_addr;
  //Input memory
  ram_in INP(
    .clock(clk),.data(data),.rdaddress(cntr),
    .wraddress(wraddress),.wren(wren),.q(in));
  /**********PHERIPERAL CONTROLLER**********/
  controlengine CTRLEng(
    .rd_step(cntr),.wr_step(),
    .source_addr(source_addr),.dest_addr(dest_addr),
    .wren_MEMZ(wren_MEMZ),.slcwraddr_MEMZ(slcwraddr_MEMZ),
//MEMZ control signal
    .wrslc_MEMZ(wrslc_MEMZ),.rdslc_MEMZ(rdslc_MEMZ),
    .wren_MEMS(wren_MEMS),.slcwraddr_MEMS(slcwraddr_MEMS),
//MEMS control signal
    .wrslc_MEMS(wrslc_MEMS),.rdslc_MEMS(rdslc_MEMS),
    .rdslc_MEMW(rdslc_MEMW),.wrslc_MEMW(wrslc_MEMW),
//MEMW control signal
    .wren_MEMW(wren_MEMW),
    .mode_SHRin(mode_SHRin),.slcin0_SHRin(slcin0_SHRin),
//SHRin control signal
    .slcaddr_SHRin(slcaddr_SHRin),
    .rst_SHRout(rst_SHRout),.mode_SHRout(mode_SHRout),
//SHRout control signal
    .slcin_SHRout(slcin_SHRout),
    .mode_SSHRin(mode_SSHRin),
//SSHRin control signal
    .mode_CE(mode_CE),
//CE control signal
    .slcin_AF(slcin_AF),
//AF control signal
    .slcin_AD(slcin_AD),.slcinz_AD(slcinz_AD),
//AD control signal
    .clear_ERR(clear_ERR),.stop_ERR(stop_ERR),
    .status(status),.request_in(request_in),
    .wren_OU(wren_OU),
    .rst(rst),.clk(clk));

  /**********OUTPUT MODULE**********/
  output_module OU(
    .in(outAF),.addr(buff_dest_addr[1:0]),
    .out0(q0), .out1(q1), .out2(q2),
    .wren(wren_OU), .rst(rst), .clk(clk));
endmodule
```