Fall 1-31-1998

Design and implementation of generalized adaptive neural filters

Chun Yip Tam
New Jersey Institute of Technology

ABSTRACT

DESIGN AND IMPLEMENTATION OF GENERALIZED ADAPTIVE NEURAL FILTERS

by
Chun Yip Tam

Generalized Adaptive Neural Filters (GANF) are a class of adaptive non-linear filters. This thesis presents a hardware implementation of GANF. Two designs are considered: the single neuron implementation and the multi-neuron implementation. The GANF design includes the window generator, threshold decomposer, training and filtering unit. The designs are verified through a logic design/simulation tool, LogicWorks.

DESIGN AND IMPLEMENTATION OF GENERALIZED ADAPTIVE NEURAL FILTERS

by
Chun Yip Tam

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Engineering

Department of Electrical and Computer Engineering

January 1998

APPROVAL PAGE

DESIGN AND IMPLEMENTATION OF GENERALIZED ADAPTIVE NEURAL FILTERS

Chun Yip Tam

Dr. Edwin Hou, Thesis Advisor — Date
Associate Professor of Electrical and Computer Engineering, NJIT

[illegible]
Dr. Nirwan Ansari, Committee Member — Date
Professor of Electrical and Computer Engineering, NJIT

[illegible]
Dr. Mengchu Zhou, Committee Member — Date
Associate Professor of Electrical and Computer Engineering, NJIT

BIOGRAPHICAL SKETCH

Author:	Chun Yip Tam
Degree:	Master of Science
Date:	January 1998

Undergraduate and Graduate Education:

- Master of Science in Computer Engineering
  New Jersey Institute of Technology, Newark, NJ, 1998
- Bachelor of Science in Computer Engineering
  New Jersey Institute of Technology, Newark, NJ, 1994

Major: Computer Engineering

To my beloved family

ACKNOWLEDGMENT

I would like to express my deepest appreciation to Dr. Edwin Hou, who not only served as my thesis advisor, providing valuable and countless resources, insight, and intuition, but also constantly gave me support, encouragement, and reassurance. Special thanks are given to Dr. Nirwan Ansari and Dr. Mengchu Zhou for participating in my committee.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Signals encountered in real life can appear in many shapes and forms. For electrical signals, which can be analog (that is continuous in time and amplitude) or digital (that is discrete in amplitude), they can represent physical quantities, such as pressure, image brightness, etc. or they can convey timing or control information.

To convert signals into voltages or currents, a sensor or transducer is needed. However, sensors are not perfect and can introduce noise. Furthermore signals may be corrupted when being transmitted. The performance of a system against noise can be measured by the signal-to-noise ratio (SNR).

1.1 Signal Filtering

In order to improve the SNR, the noisy signal can be processed by filtering. In many cases, a linear filter, whose output is a linear function of the input can be used. The mathematical analyses of linear filters are quite straightforward. However, there are situations where linear filtering does not adequately accomplish the objective.

Linear filters are most useful for additive Gaussian noise and tend to mask out high frequency components in the signal [1]. When applied to images, a linear filter will blur the edges and other high contrast areas which are needed for image clarity. Also, in many cases the noise encountered may be non-Gaussian, non-additive and may also be related to the desired signal. As a result, non-linear filters must be used to achieve satisfactory results. However, choosing a non-linear function requires complicated mathematical analysis which does not work well in practice, and non-linear filters may be difficult to implement.

1.2 Introduction to Stack Filter

Recently, there is need for easy to use non-linear filters which means that they must be easy to design (deciding on which non-linear function to use) and easy to implement (building the hardware). While all digital filters can be implemented in software, it is desirable to build fast, dedicated hardware to do the filtering. One non-linear filtering structure that is easy to design and implement are "stack filters". It enabled a large group of non-linear filters to be easily implemented with VLSI (Very Large Scale Integration) technology and it allows the construction of many non-linear filters in a compact and modular form. The dedicated hardware will also permit much faster filtering operations as opposed to an algorithm implemented with a DSP (Digital Signal Processor) chip [2]. Stack filters will be described in detail in the next section.

The problem of configuring an optimal stack filter has resulted in the development of a new class of nonlinear adaptive filters: *Generalized Adaptive Neural Filters (GANF).* The theoretical implications of GANFs are derived from the theories of stack filters and neural networks. GANFs encompass a large class of nonlinear filters which includes stack filters. It has been shown that the optimal GANF performs better under the mean absolute error (MAE) criterion than do stack filters, and that a tight upper-bound of its MAE exists. Though the neural filters reported in [1] shared some similarities with the proposed GANFs in [2], the analyses and simulations in [2] have been restricted to a single neuron structure consisting of a linear discriminant function followed by a hard or soft limiter, referred to as the hard or soft neural filter, respectively. In contrast, most of the analyses and properties derived for GANFs do not assume any specific neural architecture, and thus GANFs are more general. The objective of this thesis is to develop a hardware implementation of GANF.

1.3 Stack Filter

Figure 1 shows an overview of the stack filtering process [2]. The input to the filter is a sequence of integers belonging to the set {0, 1, 2, ... M-1}. The filter would examine a portion of the input, called a window, and produces an output which is also an integer belonging to the set {0, 1, 2, ... M-1}.

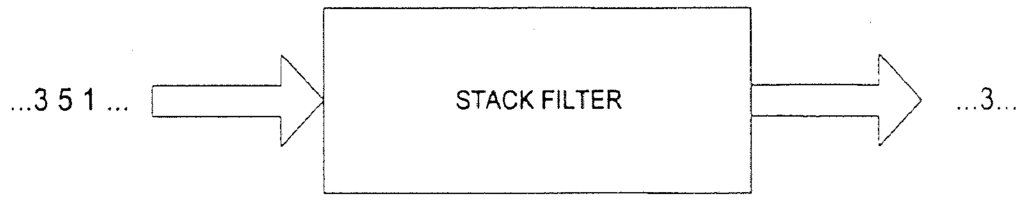

Figure 1: The general function of a stack filter

Figure 2 illustrates a stack filter with M=8, window size, B=3 and shows the low level operation of the input vector $r_B(n)$, which is composed of elements in the set {0, 1, 2, ..., M-1}. The operation can be written as follows:

$$r_B(n) = [r(n-\frac{B-1}{2}) \ \dots \ r(n) \ \dots \ r(n+\frac{B-1}{2})]^T \qquad (1)$$

where B is the window size. This vector can be uniquely decomposed into (M-1) binary vectors of length B by a threshold decomposition operation defined as follows:

$$X_B^i(n) = T^i[r_B(n)] \tag{2}$$

where

$$T^i[r_B(n)] = [T^i[r(n-\frac{B-1}{2})]\ldots T[r(n)]\ldots T^i[r(n+\frac{B-1}{2})] \tag{3}$$

and

$$T^i[x] = \begin{cases} 1, & if\ x \geq i \\ 0, & otherwise \end{cases}$$

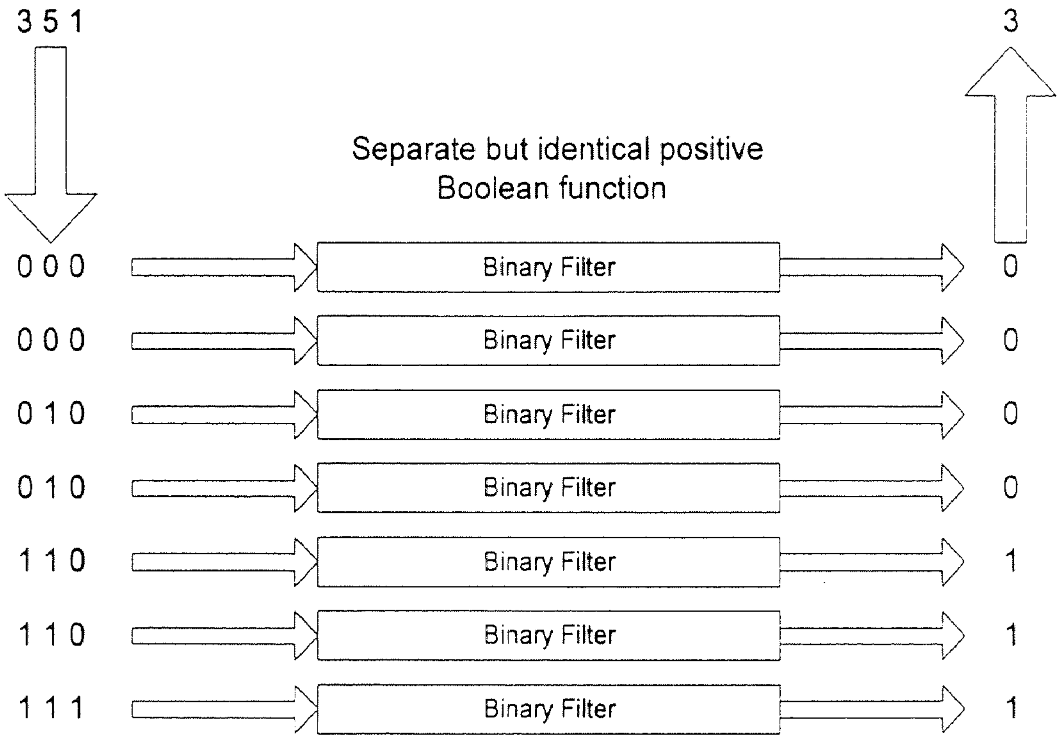

Figure 2: Stack filter with window width B=3 and M=8

At this point, we have (M-1) binary vectors of length B. Each vector is then used as an input to a separate Boolean function on each level. We have a total of (M-1) such Boolean functions, operating on B binary inputs and producing (M-1) binary outputs. For a stack filter, all of the (M-1) Boolean functions are identical. However, since the inputs (the vectors $X_B^i(n)$) are not all the same, the Boolean function outputs may be different. In addition, the Boolean function is required to be a positive Boolean function.

After obtaining the (M-1) binary outputs from the Boolean functions on each level, the integer output y(n), is computed by summing the outputs of the Boolean functions. Note that the inputs to the Boolean functions, and the Boolean functions are positive. Let $X_B^i(n)$ and $X_B^j(n)$ be the binary input vectors of two separate but identical Boolean functions. The outputs of these two functions are $y^i(n)$ and $y^j(n)$ respectively. Then, if $i<j$ the outputs must satisfy $y^i(n)$ less than or equal to $y^j(n)$. It can be seen that all level outputs can "stack". This means that there will always be a column of 0's above a level output of 0 and a column of 1's below level output of 1. There is only one point where a transition between 0 and 1 can occur and the output y(n) will be equal to the largest level number which has an output of 1. As a result, the filter output can be obtain through a binary search of the Boolean function outputs to determine where the 1 to 0 transition occurs. This enables a savings in VLSI chip area.

The median and other rank-order operators possess two important properties: the threshold decomposition property and the stacking property. The first is a limited superposition property which leads to a new architecture for filters. The second is an ordering property which allows efficient VLSI implementation of filters. Any filter which possesses both the threshold decomposition property and the stacking property is known as a stack filter. Thus, they are constructed as a "stack" of positive Boolean functions based on the threshold decomposition property and the stacking property.

The Boolean function used on each level defines the operation of the stack filter. By selecting an appropriate Boolean function, many types of nonlinear filters can be implemented, such as rank order filters and all morphological filters. For example, a median filter for window size 3 can be achieved using a Boolean function described by

$$y = x_1x_2 + x_2x_3 + x_1x_3 \tag{4}$$

As pointed out previously, there are a large number of positive Boolean functions, and therefore a large number of stack filters. The next question is how to pick a positive Boolean function suitable for a given filtering problem. There are many methods which can be used to accomplished this, and this thesis will implement the Ansari-Lin method [2].

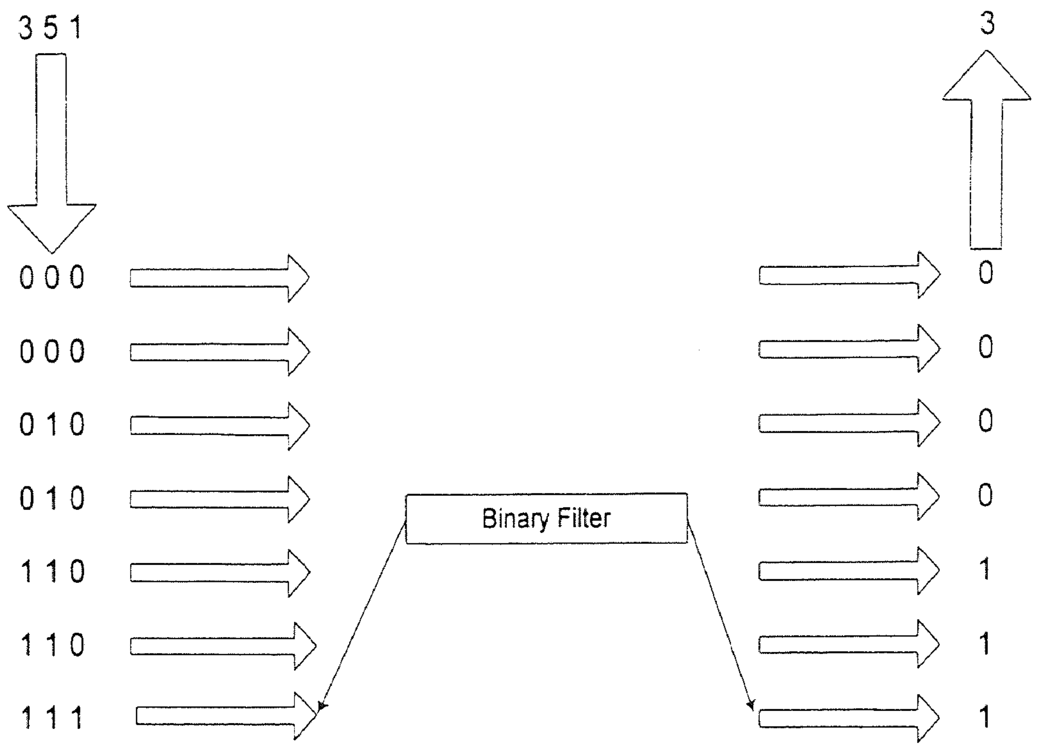

Figure 3: Single neuron for adaptive stack filtering

The Ansari-Lin method involves using a single neuron to implement the positive Boolean functions required in the stack filter. The basic structure of the filter is shown in Figure 3. Its operation is similar to the stack filter with the exception of the positive Boolean function. Each separate positive Boolean function is replaced by a single neuron which rides up the level to provide separate level outputs. In other words, the single neuron looks at the binary vector at a certain level and produces a binary output. Then it moves up a level and does the same thing. This is done for all (M-1) levels. The stack filter outputs is taken to be the level at which the outputs changes from 1 to 0.

The neuron consists of a summation node with (B+1) weighted inputs and a non-linear threshold function. One of the inputs is permanently assigned the value of one; the others come from the binary input vector on a certain level. Because of the threshold function, the output is binary.

By adjusting the weights, different classifications of the binary input vectors can be achieved. The weights can be adjusted in many ways. Ansari *et al* [2] used perception learning and have demonstrated good results. The equation for updating the weight is given :

$$\omega(n+1) = \omega(n) + \mu * x(n)\varepsilon \tag{5}$$

where ε is LMS error and μ is the factor decide whether to update the weight.

This thesis focuses on the hardware implementation of the GANF. It includes schematic design, selection of hardware, and the communication method for inter-module communication. Information is provided to construct a complete GANF which include three major operations.

There are three major operations involved in the GANF. They are the window generation, the training operation and the filtering operation. The window generation is implemented by the Window Generator which is responsible for reading and reordering the data and forward them to the training and filtering unit. The latter two operations are performed by the training/filtering unit. It provides the overall control of the computation process, collects results from the neurons to form the final output, and is also used for computation in the training phase.

For normal filtering operations, the weights would first be broadcast by the training/production unit to the neuron. The input signals (provided by the Window Generator) are then threshold decomposed and then passed to the corresponding neurons.

After B (window size) cycles of computations by the neuron, the results are passed back to the training/production unit from all the neurons and assembled to form the final output.

During the learning phase, the operation is similar to the normal filtering operation except at the last cycle of computation, the desired output is also threshold decomposed with the binary values sent to the corresponding neuron and compared with the final output. A weight update signal is then generated according to the result of the comparison and the weights will be updated according to the learning rule.

The rest of this thesis is organized into four chapters.

Chapter 2 describes the Window Generation operation and its hardware implementation. Chapter 3 discuses the filtering operation and its implementation. Chapter 4 discusses the training process and its hardware implementation and Chapter 5 presents the conclusion and future research.

CHAPTER 2

WINDOW GENERATION

Due to the size of image and the limited number of neurons that are available in the GANF, the image is typically processed in small pieces, called windows. In most instances, a square window is used; for example 3 x 3 window with 9 pixels is shown in Figure 4 and Figure 5. At each iteration, the window is allowed to slide across the image thus generating new data for the filtering operation.

2.1 Overview

Figure 4 illustrates a 3 x 3 windows with pixels, B,C,D,L,M,N,T,U,V. The new window is generated by sliding towards the right one pixel. As a result, the new window shown in Figure 5 contains the new data C, D, E, M, N, O, U, V, W.

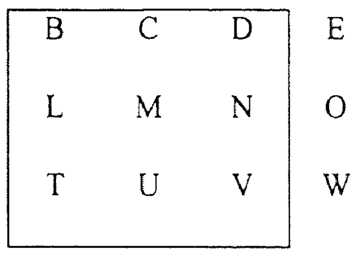

Figure 4: The current window's position in a picture

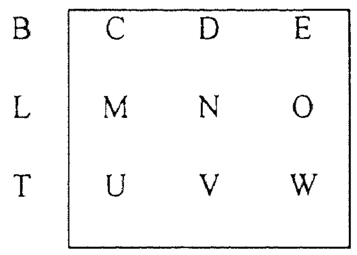

Figure 5: The next window's position in a picture

2.2 Window Generator

As illustrated in Figure 4 and 5, there are common data in the old window and the new window. In order to reduce the I/O complexity of re-reading identical pixels as the window move across the image, the window generator should be designed so that useful pixels are retained.

2.3 Total Number of Buffer Unit Required

The first step is to determine how many and which pixels are needed to store. For example, given a (W x H) 10 X 10 image and a (w x h) 3 X 3 window as shown in Figure 6, we will determine the number of buffers needed. In Figure 6, the top figure is a image, the middle is a window, the bottom is the image overlapped with the window. The scanning starts at the upper left corner and it scans all the way to the right. Then it goes back to the left size, and moves down one line of pixel. This process repeats itself until the entire image has been scanned.

If we visualize this scanning process, we will discover that the first line of the image is used only once (all successive scan will not use this line of pixel). It is because the Window will shift down one line when it starts itself from the left end, and only the second and the third line of the current window are useful (for window size = 3). This means that the number of pixels need to be stored is (h-1) * W.

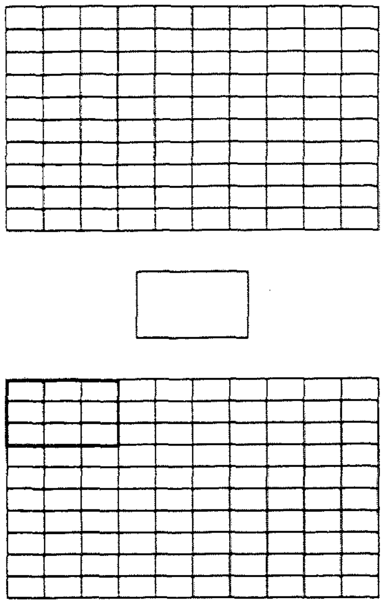

Figure 6: The relationship between window and image

As the window moves right, it reads a new value that is usable for the next two passes. Since we do not want to re-read any of these elements, they must be stored temporary for two passes and then overwrote by other elements as the scan progresses. As a result, in addition to the above number of elements required to be stored, we also need to store another w element, which is 3 in this case.

The total number of buffer unit required to store all the necessary elements are

$$bufferrequired = (h-1)*W + \omega \tag{6}$$

For example, as shown in Figure 7, a 4 x 3 window in a 10 x 5 image needs buffer space to store (2*10)+4=24 pixels.

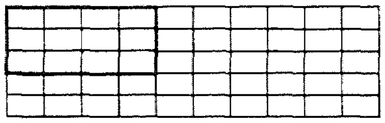

Figure 7: An example of 4 x 3 window in a 10 x 5 image

2.4 The Scanning Logic

At the start of the window generation process, all buffers must be initialized with pixels from the image. To fill the buffers of the window generator, the pixels will be read from left to right and then top to bottom until all buffer space is filled. For example, a buffer of 23 pixels will be initialized with first two lines and first 3 pixels from the third line of a 10 by X image with 3 by 3 window (X is used to denote the height of the picture because it is not a factor in determining buffer size).

First of all, windows are generated by reading appropriate data from the buffer. When the window moves to a new position, the buffer must be updated with new values. The buffers are updated by a shifting mechanism.

1. First round (starts at first line of the picture) : Top left of the window generator is placed on the first pixel of the picture. Inside this window, there are no new pixels because all of them have already been stored into the buffer during the initialization phase. The first set of data (a window) is generated by reading the buffer. The window then move left one pixel. Top left hand corner of window generator is now located on the second pixel of the picture. Inside this window, there is a new pixels in the rightmost, bottom-most corner. The buffer will be updated with this new value. Then a window generates. This pattern will continue until the window can not be moved to left anymore because the rightmost of the window is already at the far right side of the picture. This denotes the end of the First round.
2. Second round (starts at second line of picture) : Top left corner of window generator is placed on the first pixel of the picture's second line. The bottom row of the window

contains all new pixels, so the buffer is updated with these new values. A window is generated. Window moves one pixel towards the right, and the lower right hand covers a new value. Buffer is updated and a window is generated. The pattern continues until it reaches the rightmost side of the picture and this denotes the end of the second round.

3. Third and subsequent round : They all follows the same pattern as in the second round until (the rightmost, bottom-most) reaches the lower right hand corner which is the last pixel of the picture. This denotes the end of the last round and the window generation process.

Figure 8 shows an example of the window generation process. The top part displays the content of the buffer chain after each round of processing. The bottom displays the picture that is being scanned. It shows the content of the buffers as the window move across the image. The first line shows the content of the buffer after initialization. The bolded data will be read to generate a 3 by 3 window. After the first window is generated by reading the nine bolded data in the first line, all data in the buffers will be shifted towards left one pixel to prepare for the second window. After the shift, the content of buffers would look like line 2. This process continues until all pixels in the original picture are visited by the window generator.

For reference purpose, two programs written in C++ to simulate the process of window generation are included in Appendix B and C. Program picgen (Appendix B) creates an image of any specified size and program wingen (Appendix C) will generate all the windows.

buf01	buf02	buf03	buf04	buf05	buf06	buf07	buf08	buf09	buf10	buf11	buf12	buf13
1	2	3	4	5	6	7	8	9	10	11	12	13
2	3	4	5	6	7	8	9	10	11	12	13	14
3	4	5	6	7	8	9	10	11	12	13	14	15
4	5	6	7	8	9	10	11	12	13	14	15	16
5	6	7	8	9	10	11	12	13	14	15	16	17
6	7	8	9	10	11	12	13	14	15	16	17	18

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17	18	19	20
21	22	23	24	25

Figure 8: Content of buffer chain and the image

2.5 Hardware Design and Implementation

In this section, we will examine the design and hardware implementation of the window generator buffer in detail. Throughout this section, a 3 by 4 window on a 50 by 50 image (each pixel is 8 bit) will be used as an example. An equation is also provided to determine the corresponding variable for other situations and requirements, such as different size of window or image.

The **number of buffers** defined above is implemented by using shift registers. In the above example, it requires 153 shift registers (each of them is 8 bit wide).

There is a certain number of **reading pointer** that are set up to indicate where to read along the shift registers. Each of them is anchored to a fixed position along the shift registers. Their function is to tell the window generator where to start reading data to assemble a window. The example requires 4 pointers. The equation defining the reading pointer is :

The number of reading pointer = Height of window (7)

At each reading pointer, a value called **BufRead** controls how many buffer must be read. The above example has BufRead equals to 3. It means at each reading pointer, 3 buffers will be read. The general equation is :

BufRead = Width of window (8)

From the above calculation, 4 reading pointers combines with BufRead=3 producing 12 values which is exactly one window.

To determine the placement of the reading pointers, the rule of thumb is putting the 1st anchor at the head of the shift registers, and each additional reading pointer is to be placed every (Width of picture - Width of Window BufRead (B)) units.

The generator starts from the left and moves right one pixel after each round of scanning. When it reaches the right side of the picture, it goes down one row and starts from the leftmost again. When it arrives at the lower right hand corner of the picture, the process is finished.

The number of **scanning required to finish a horizontal row** is simply

Width of picture - Width of window + 1 (9)

For the above example, there will be 48 scanning in each row (each row contains 50 pixels).

The number of **horizontal scans required to finish an entire picture** is simply

Height of picture - Height of window + 1 (10)

For the above example, it takes 47 horizontal scans.

With all necessary parameters defined, we can examine the entire window generation step by step.

1. At the start of the window generation process, the following happens:
    a) All buffers will be initialized by reading in data from the image. This is done by reading the 1st pixel and writing it to the end of the register chain.
    b) Shift left the registers.
    c) Repeat step (a) and (b) until all shift registers are occupied.
2. Generate windows for the picture in the following steps:
    a) Read the 1st line of the window at the 1st reading pointer on the shift register chain. BufRead (representing the wide of the window) determines the number of buffers to be read (including the buffer pointed by the reading pointer).
    b) Read the 2nd line of the window at the 2nd reading pointer on the shift register chain. Again BufRead determines the number of buffers to be read.
    c) Repeat the above step by continue reading subsequent lines of the window started at the corresponding reading pointer until an entire window is generated.
    d) Shift left the register chain by 8 bits (depend on the number of bits required to represent a pixel).
    e) Feed the next pixel (never read before) from the picture to the right end of shift registers.

f) Generate another window by repeating step (a) through (e) until the last window of the current row is generated, i.e. Window reaches the rightmost pixel of current row of the image.

g) At this point, the current row is finished and the window will start scanning the next row. Left shift the register chain by the number of buffers equal to BufRead. Each time a buffer is left shifted, feed in a new pixel from the image.

h) Repeat from the first step until no new pixel is available from the image, i.e. The entire image is scanned.

2.6 Threshold Decomposition

This unit is responsible for threshold decomposition which occurs between window generation and filtering process. It decomposes a X-bit pixel to (2^x - 1) levels. The value of 1 in level y implies the value of the pixel must be bigger than or equal to y. If the pixel has a value 7, all 7 lower levels would become 1 after threshold decomposition. If the pixel is 0, all levels would become 0. At this point, the thresholded data will be processed by a neuron. Table 1 shows an example of for a 3-bit pixel. A LogicWorks simulation of a 3-bit pixel threshold decomposer is shown in the next section.

The number of threshold decomposer needed in a system depends on the size of the Window Generator. Since the neuron requires all pixel threshold decomposed, window size of B would then require B threshold decomposers to process all pixels. The output of each pixel unit in the Window Generator is connected to the input of a threshold decomposer. At the output of threshold decomposer, data is then setup for the neuron. A threshold decomposer contains only combination circuit, so there is no clock or any control signal associated with it. When input data arrived, it would be processed immediately and result would be available after hardware propagation delay.

Table 1 - Example of 3-bit decomposer: each column represents a pixel value and each row represents the value of the a level, 1 or 0(empty)

	0	**1**	**2**	**5**	**6**	**7**
VII						1
VI					1	1
V				1	1	1
IV				1	1	1
III				1	1	1
II			1	1	1	1
I		1	1	1	1	1

2.7 LogicWorks Example

An interactive circuit design software, LogicWorks, is used as a design and simulation tool. The software provides a schematic drawing feature that has full digital simulation capability. Circuit output may be displayed in the form of timing diagrams or on simulated output devices.

Figure 10 provides a sample circuit implementation of a window generator. The circuit implements a 3X3 Window Generator circuit and each pixel is four bits wide.

The lower right hand corner show the value of each pixel enclosed by the window at a particular instance. W1 denotes the first pixel in the window and W1-0 denotes the least significant bit of this pixel.

The Hex keyboard is used as an input device for entering pixel values. In an actual implementation, a sensor or digital imaging devices would replace it and provide the pixel value.

All CLR (clear, active low) signals are connected together and are controlled by a binary switch. This switch is only used in the beginning to reset the system. CLK (clock) signals are also connected together, every time the CLK signal is asserted (transition from low to high level), the input will be stored in the flip-flop and its value is reflected at the output.

Extending this circuit from 4-bits to 8-bits pixel is straight forward. This can be done by duplicating the 4-bit circuit and combining them together. The original circuit will handle the least significant 4-bits and the duplicate handles the most significant 4-bit.

To change the window size from 3 by 3 windows to a X by X window would require more modification than just changing the bit size. The first step is to find out how many buffer units are required using Equation (6), the number of reading pointer by Equation (7) and the number of BufRead by Equation (8). With all the data, we can then expand to any size window generator needed.

Figure 9 provides a LogicWorks simulation of a 3-bit pixel threshold decomposer. L7 through L1 denotes the output of the data in each level, i.e. L7 means level 7. S0 through S2 are inputs from the actual value of a pixel. Here, three bits input positions are used to read a 3-bit pixel. Its function is to threshold decompose pixel, which will then be processed by a neuron. The two components involved are AND gates and Binary To Decimal Converter(Decoder). If the pixel was 5, all outputs below level 6 (not including level 6) became 1. If the pixel was 0, then all levels output 0.

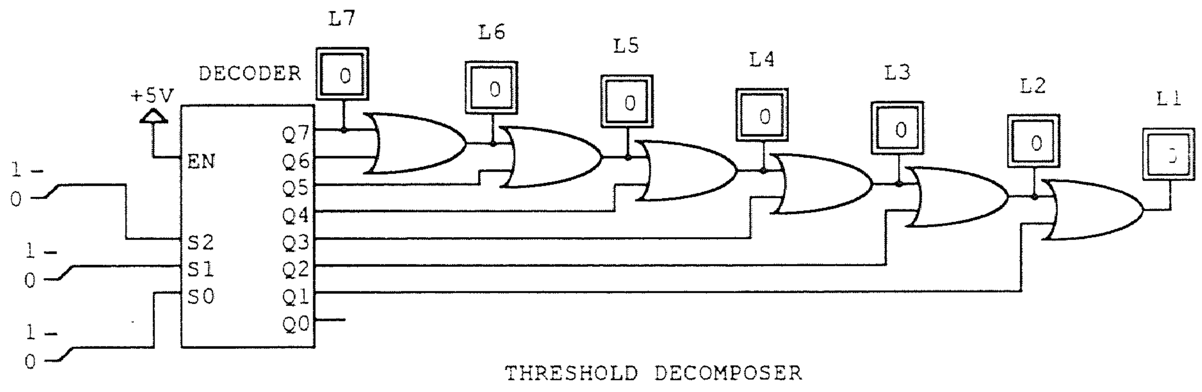

Figure 9: Circuit of the 3-Bit Threshold Decomposer

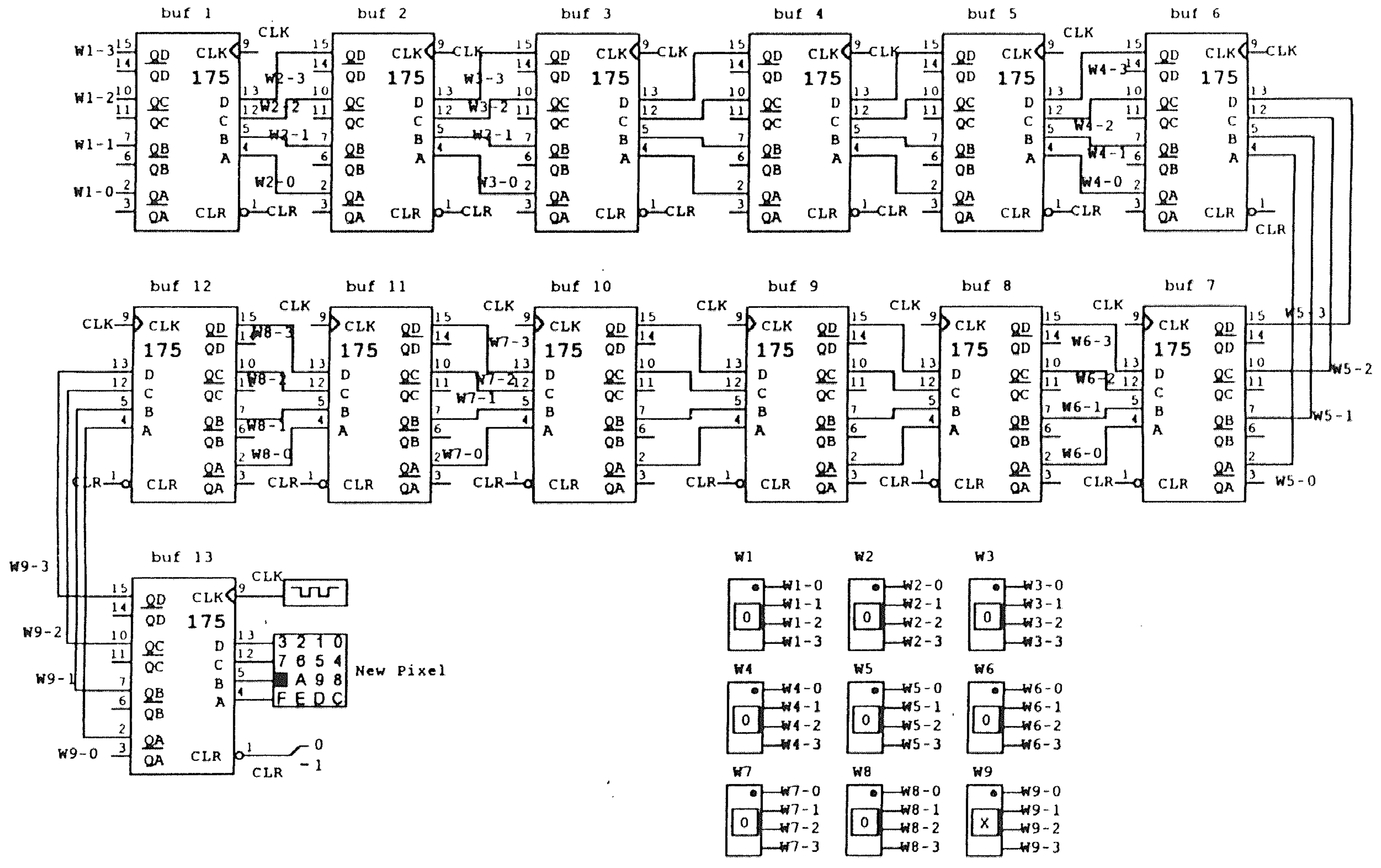

Figure 10: Circuit of the Window Generator

CHAPTER 3

FILTERING OPERATION

The operation of the GANF can be separated into 2 operations: training and filtering. In the training operation, a set of training data and the desire output is fed into the GANF so that the weights in Equation (5) will be adjusted appropriately.

3.1 Overview

The hardware requirements for the training and filtering operation are quite similar and can be combined together. To reduce the hardware complexity, thus simplifying the construction of the neuron, the computation of Equation (5) is performed in a number of steps. The weighted sum is first computed, and the thresholded value will determine the output. Figure 11 shows a block diagram of the hardware involved in this process.

3.2 Initial State

The initialization process includes resetting the register that stores the output of adder, loading the weights into the register file, resetting the counter and the window generator.

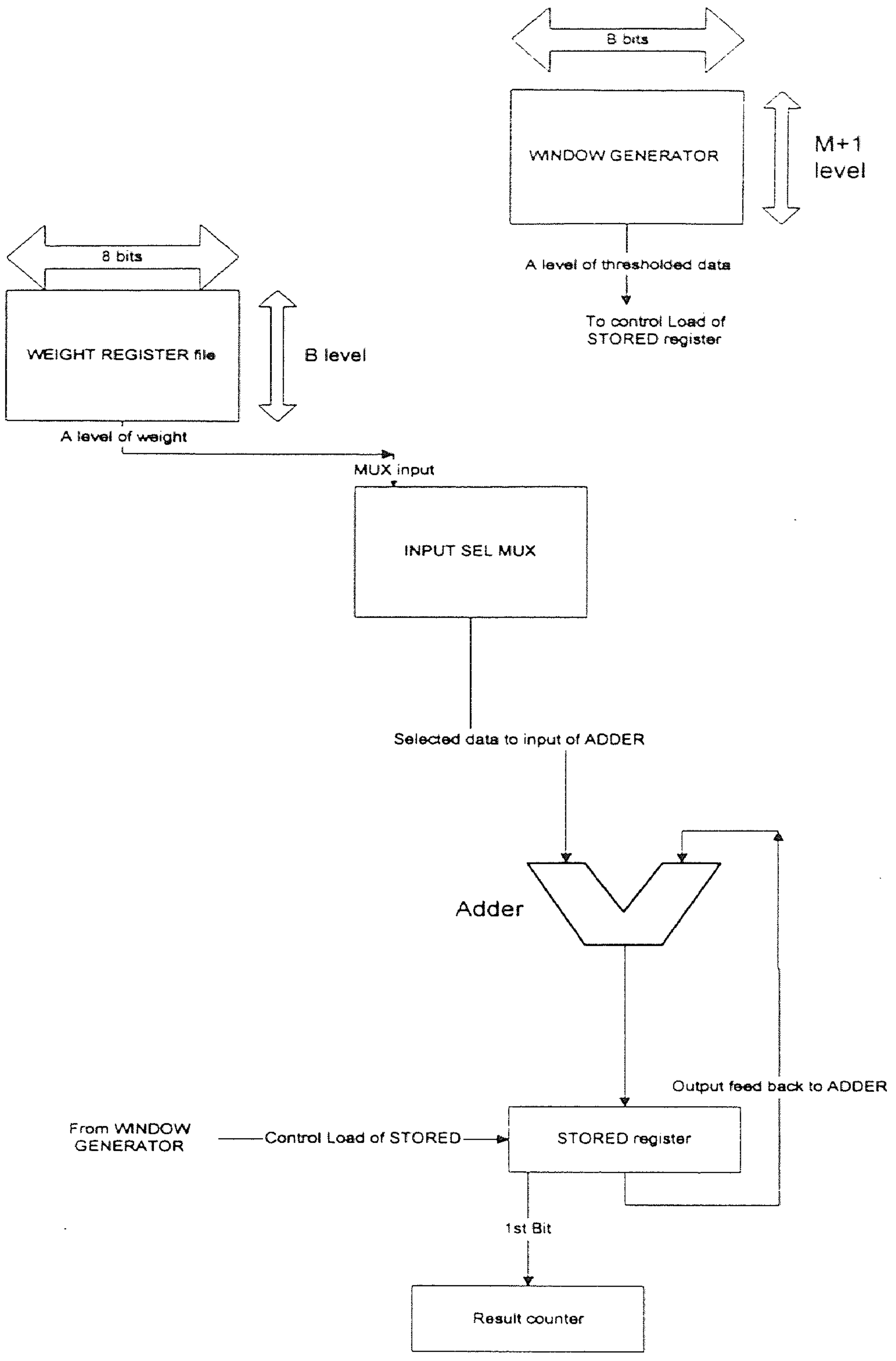

Figure 11: Hardware block diagram of filtering process

3.3 The Control Logic

The procedure is basically controlled by the CLOCK and RESTART signal of the system. There are two loops involved and one is nested inside the other one.

The inner loop determines a partial value (it is partial because the final result would be the combination of all level of the thresholded data) of the result. The inner loop calculates a thresholded portion of the result. That portion is formulated by adding up appropriate weights in the Weight Register File. A set of thresholded data provided by the Window generator is used to decide which weights to add up. By examining one level of thresholded data from the window generator, this loop will add up the appropriate weights in the weight register file. The sum is the partial value of the clean pixel. The number of times it loops is defined by the Window size. A 9-pixel window means 9 bits of data in one level of the thresholded value, which also implies there are 9 weights in the weight register file.

The outer loop performs initialization and calculates the final output. After inner loop completes, the most significant bit (MSB) of the STORE register is recorded in each outer loop. Gathering all recorded value is the final result. The number of times it loops is determined by the number of level a pixel is thresholded to. If a pixel was thresholded to 10 levels, the outer loops would loop 10 times to generate 10 MSB value which would be the final output.

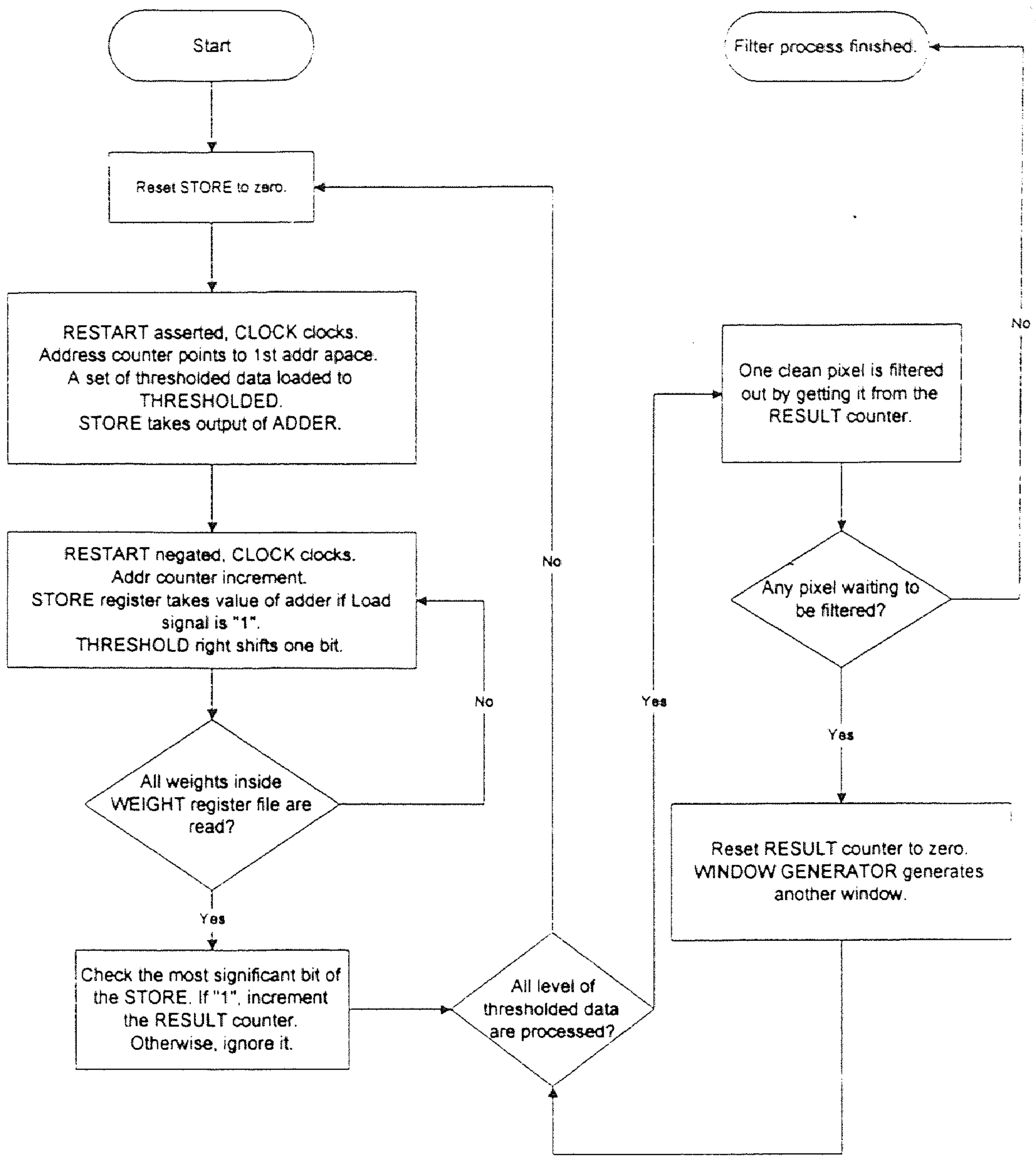
Start
Reset STORE to zero.
RESTART asserted, CLOCK clocks.
Address counter points to 1st addr apace.
A set of thresholded data loaded to THRESHOLDED.
STORE takes output of ADDER.
RESTART negated, CLOCK clocks.
Addr counter increment.
STORE register takes value of adder if Load signal is "1".
THRESHOLD right shifts one bit.
All weights inside WEIGHT register file are read?
No
Yes
Check the most significant bit of the STORE. If "1", increment the RESULT counter. Otherwise, ignore it.
All level of thresholded data are processed?
No
Yes
One clean pixel is filtered out by getting it from the RESULT counter.
Any pixel waiting to be filtered?
Yes
No
Reset RESULT counter to zero. WINDOW GENERATOR generates another window.
Filter process finished.

Figure 12: The filtering process flowchart

3.4 Implementation

The following section provides a brief description of the hardware components involved in the implementation. Figure 11 provides a logical organization of all the components.

- An 8-bit ADDER with two operands as input. One from the Weight register file, W7-W0. The other one is from an 8-bit register called STORE, Y7-Y0. The sum goes into the input of STORE register, S7-S0.
- The STORE register gets S7-S0 from the ADDER and output Y7-Y0. The clock signal connects to CLOCK of the system. The clear signal connects to RESTART of the system. The load signal connects to the least significant bit of a right-shift register called THRESHOLDED, T0.
- THRESHOLDED is a right-shift register. The number of bits required is governed by the Window size. Shift signal links to CLOCK of the system. Load signal links to the RESTART of the system. All bits will be set with data from a level of the thresholded value after the Window generation process when RESTART is clocked.
- Weight Register File - A memory chip is used to store all the weights generated during the training process. The number of address space is determined by the Window size + 1 (for the always added value). The size of each address space is the number of bits required to represent a pixel. There is a counter pointing to the address input of this memory module to provide location of the weight wanted. The clock of this counter is connected to the CLOCK of the system. The counter will be reset to zero when RESTART is clocked.
- A result counter, RESULT, is used to remember the thresholded final value before it is assembled back to a pixel. Its clock signal connects to the RESTART signal of the system. The increment signal connects to the most significant bit of the Store register. If this significant bit is 1, counter will increment one after RESTART clocked.

All of the above components are also used for the Training Operation since the filtering operation is a subset of the training operation. The detail actions taken in each clock during the filtering operation is as follows.

It starts with both Store and Thresholded register reset to zero.

1. RESTART is asserted and CLOCK clocks. Load the w_0 from the first address space of Weight Register File to Store register by performing steps below:
   a) Send weight from the Weight Register File which connects to one operand input of Adder by having the counter of Weight Register File pointing to the first address space.
   b) Assert the Load signal of Store register to load the output of Adder which is the sum of 0 and the "always added weight" after CLOCK clocks.
   c) At this same clock tick, Thresholded register loads all data bits with the bottom set of thresholded value which is just generated by Window Generator.
2. The CLOCK ticks again. The counter of Weight Register File increments one unit and points to the next address space. The Store register will be clocked by the same CLOCK. Depending the current value of the Load signal, it determines whether to load the sum from the Adder. If current signal is 1, then the sum will be loaded, otherwise it will be ignored. In the same clock tick, Thresholded register is right shifted one bit by clocking the right-shift signal. Repeat this step until the last Weight has been read and processed.
3. The RESTART is asserted and CLOCK clocks the system. Result counter performs increment depends on the most significant bit of Store (increment if the bit is 1,

otherwise do nothing). The address counter of Weight Register File is reset to zero and thus points to the first address space of the file. The Thresholded register loads the next set of thresholded data from Window Generator. Store register loads the sum of Adder which is the always added Weight. Repeat the previous step. Recursively repeat this step until an entire set of window is processed.

4. An output is generated and the result is stored inside the Result counter. For example, if the counter is 6, then the pixel value is 6.
5. Window Generator performs another sampling and threshold operation.
6. Go back to step 1 to begin filter another element.

The above steps will be repeated until there are no more windows available from the Window Generator.

3.5 Multiple Neuron Implementation

The filtering operation unit discussed previously was a single neuron unit. A single neuron is utilized to service all levels of threshold decomposed data by sliding back and forth between levels. An alternative is to provide every level a neuron. This is the multiple neuron implementation. The advantage of using a single neuron is the filtering operation unit becomes more compact on size and less expensive because of hardware reduction by having all level of data sharing a neuron, but trade off is slower throughput.

In the multiple neuron implementation, data from all level of threshold decomposition is output to its own level's neuron. All data will then be processed simultaneously and the algorithm remains the same. The hardware connection diagram in fact is simpler in term of design because the elimination of the logic to slide a neuron and feed data from an appropriate level. Size and cost of the unit is increased due to the duplicated hardware components in building neurons. The gain in here is higher throughput. With 15 levels of data ready to be processed, throughput measurement in multiple neuron unit can perform 15 times better then a single neuron unit. Figure 13 shows a block diagram of the implementation with multiple neurons. In the figure, an instant of the filtering process is captured. The top block labeled "One instant of the window" represents the content of a 3 by 3 window (which contains 9 pixels). The block below represents a Threshold Decomposer for each pixel. After a pixel is decomposed, its decomposed data will be distributed among the groups of Thresholded Register. Each decomposed data bit goes to the corresponding Register which belongs to its level. The next block is the neuron which responsible for the particular level.

3.6 Implementation of System Control Unit

So far we have described the control signals needed for the filtering operation such as the CLOCK, Reset of THRESHOLD and Load of THRESHOLD, etc. Control signal contains binary information which tells a device how to carry out a particular action. These signals control every single operation took place throughout the process and must coordinate with other modules such as the Window Generator to obtain faultless result.

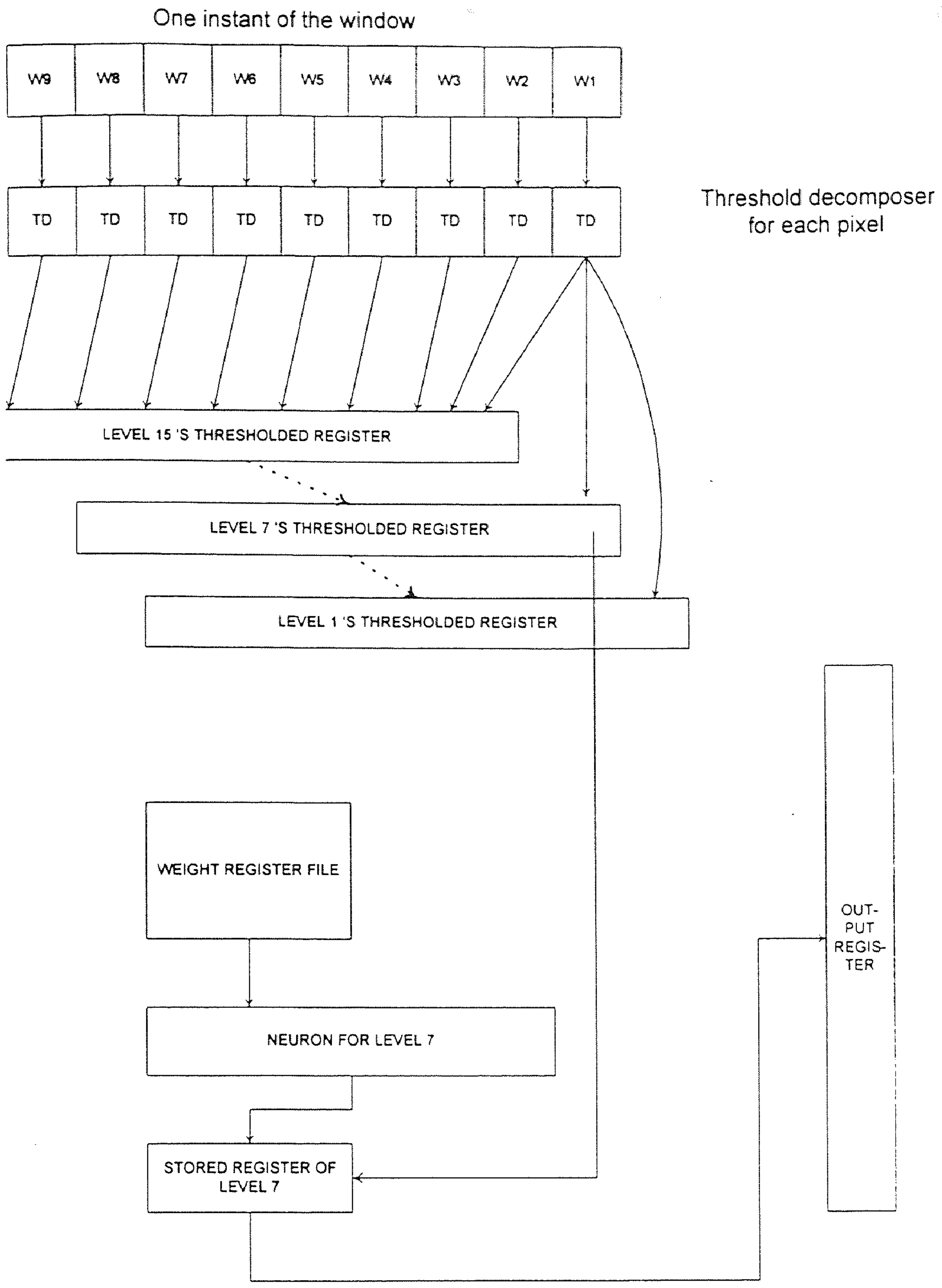
One instant of the window
W9
W8
W7
W6
W5
W4
W3
W2
W1
TD
Threshold decomposer for each pixel
LEVEL 15 'S THRESHOLDED REGISTER
LEVEL 7 'S THRESHOLDED REGISTER
LEVEL 1 'S THRESHOLDED REGISTER
WEIGHT REGISTER FILE
NEURON FOR LEVEL 7
STORED REGISTER OF LEVEL 7
OUT-PUT REGIS-TER

Figure 13: The block diagram of the multiple neuron implementation

The two common methods used to build control units are sequential circuit and microprogramming. Sequential circuit are state machines where the operation of the system is described by state. The advantage of this method is that the state of the system is easy to keep track and if an error did happen, the system will not go to the next state and wait for user intervention until the current state exit successfully. The disadvantage is the complexity involved in designing and building the combination circuit which would drive the cost up.

The second method of implementing a control unit is microprogramming. The control signals are embedded into read only memory (ROM) where each step of the operation is stored [8].

Table 2 shows the content of the microprogramming code to control the Window Generator. Data input and output signals are all listed. The value will be represented by different variable which represents any valid data.

Table 2 - The microcode content of Window Generator

CLR	4 - bit new pixel D C B A	W 9	W 8	W 7	W 6	W 5	W 4	W 3	W 2	W 1	CLK
0											0
0											1
1	X										0
1	X	X									1
1	Y	X									0
1	Y	Y	X								1
1	Z	Y	X								0
1	Z	Z	Y	X							1
1	A	Z	Y	X							0
1	A	A	Z	Y	X						1
1	B	A	Z	Y	X						0
1	B	B	A	Z	Y	X					1
1	C	B	A	Z	Y	X					0
1	C	C	B	A	Z	Y	X				1
1	D	C	B	A	Z	Y	X				0
1	D	D	C	B	A	Z	Y	X			1
1	E	D	C	B	A	Z	Y	X			0
1	E	E	D	C	B	A	Z	Y	X		1
1	F	E	D	C	B	A	Z	Y	X		0
1	F	F	E	D	C	B	A	Z	Y	X	1
Completed initialize and 1st set of window is ready for threshold decompose.											
1	G	F	E	D	C	B	A	Z	Y	X	0
1	G	G	F	E	D	C	B	A	Z	Y	1
Another set of window is ready for threshold decompose.											

The next table shows the content of the microprogramming code used to control the filtering process within the neuron. It also control the shifting of the Thresholded register, and incrementing the Address counter of the Weight Register File so that the proper weight is fetched. The Load signal of Store register gets input from the least significant bit of the Thresholded register. The variable A is the input. At the last stage which adds the always added weight, the Load signal is always asserted.

Table 3 - The microcode content of filtering process

Thresholded Register (right shift)			Weight	Store Register		
CLK	LOAD	SHIFT	Addr Ctr	CLK	CLR	LOAD
0	1	0	0001	0	1	A
1	0	0	0001	1	0	A
0	0	0	0010	0	0	A
1	0	1	0010	1	0	A
0	0	0	0011	0	0	A
1	0	1	0011	1	0	A
0	0	0	0100	0	0	A
1	0	1	0100	1	0	A
0	0	0	0101	0	0	A
1	0	1	0101	1	0	A
0	0	0	0110	0	0	A
1	0	1	0110	1	0	A
0	0	0	0111	0	0	A
1	0	1	0111	1	0	A
0	0	0	1000	0	0	A
1	0	1	1000	1	0	A
0	0	0	1001	0	0	A
1	0	1	1001	1	0	A
0	0	0	0000	0	0	1
1	0	1	0000	1	0	1

CHAPTER 4

TRAINING PROCESS

In the filtering operation, we have seen how unwanted signal is filtered out by the system. During this phase, original data is altered by adding appropriate weights from the WEIGHT-REGISTER file. These weights have to be determined beforehand. The value of these weight plays a critical role on how accurate the result will be.

To customize the filter to a particular type of noise, the filter has to be trained. In other word, it needs to know the characteristic of the noise so the filter can distinguish between noise and data. By learning the noise characteristics, the filter became equipped and ready to filter the noisy data signaled. In order for it to learn the noise characteristics, the filter has to go through a training process.

4.1 Overview

The training process will use samples that include both the original data and noisy data. The filter will compare both set of data and then derive an updated weight and store it to the WEIGHT-REGISTER file. If the number of samples are sufficient, the updated weight will be closer to the ideal value (the value which will produce an output exactly the same to the original). Getting the ideal value requires huge amount of data samples and time. To justify between the cost and result, a certain amount of data samples will be provide to obtain a result that meets minimum requirement and expectation.

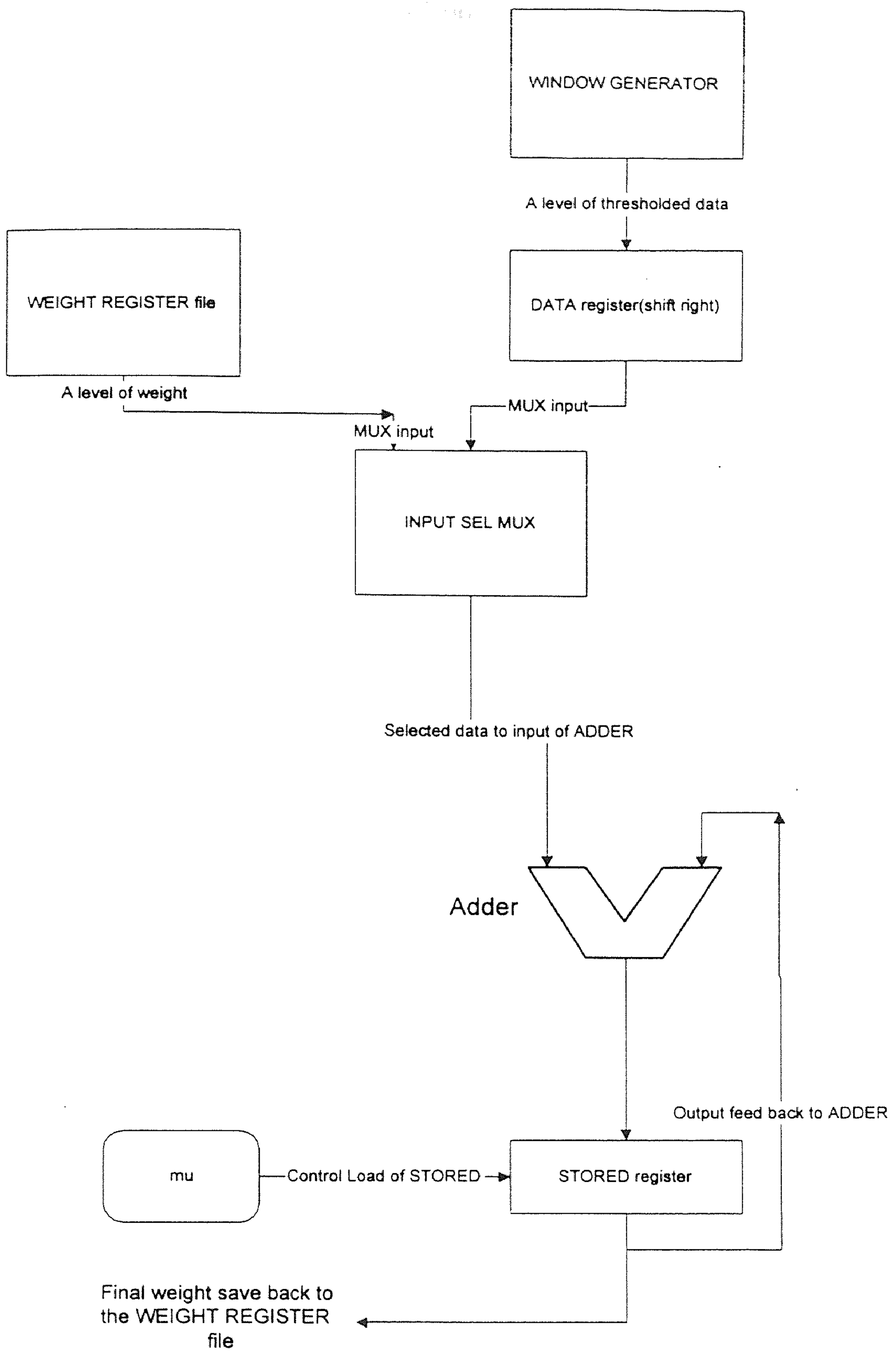
WINDOW GENERATOR
A level of thresholded data
WEIGHT REGISTER file
DATA register(shift right)
A level of weight
MUX input
MUX input
INPUT SEL MUX
Selected data to input of ADDER
Adder
Output feed back to ADDER
mu
Control Load of STORED
STORED register
Final weight save back to the WEIGHT REGISTER file

Figure 14: Hardware block diagram of training unit

4.2 Initial State

Training process is the first step when a GANF is assigned with any task. Two sets of data are needed, a clean image and the same image corrupted by noise. This noisy image is altered by feeding the clean image through the same channel or medium where it will be transmitted.

At this point, the default value is loaded into the WEIGHT-REGISTER file so it has data to start with. All registers and counters will be reset before anything starts. Some special registers (as we will describe later) will be loaded with predefined constant. When all hardware is ready, the process will start feeding in sample data and calculate updated weight.

4.3 The Training Logic

The weights can be adjusted in many ways. The method used is Ansari *et al.* [2] which uses both LMS and perception learning. Basically, a signal for which the desired response is known is processed by the filter. For each sample processed, an error is generated which is used to update the weights. For LMS, the error is analog and adaptation attempts to minimize this error for future samples. The perception learning scheme is similar, but uses a discrete error. For both methods, however, there is no guarantee that the neuron will implement a positive Boolean function. In order to achieve this, negative weights can be set to zero. Also, it should be pointed out that a single neuron may not be able to implement all possible positive Boolean functions. This means that the Ansari-Lin method may not find the optimal stack filter among all positive Boolean functions. It will, however, find the optimal stack filter among all threshold functions.

Based on [2], the learning equation is

$$\omega(n+1) = \omega(n) + \mu x(n)\varepsilon \tag{12}$$

In this equation, μ is preset with a constant is always less than one. In this thesis, it is assumed to be 4 bits wide after the decimal point. The derivation of μ will not be discussed. ω(n+1) is the updated weight while ω(n) is the current value of the weight. x(n) is the corresponding level of threshold decomposed value being process. ε is a bit resulted by comparing the desired value with the output. If they matched with each other, ε would be 0 which means the weight will be unchanged. Otherwise ε would be 1 which means the weight in current level (from 1 to B) requires an update. The equation shown above is in scalar form. The vector form of the equation will make it more meaningful and understandable later in the logic of implementation. The vector is written as

$$\begin{bmatrix} \omega_1(n+1) \\ \omega_2(n+1) \\ \vdots \\ \omega_B(n+1) \end{bmatrix} = \begin{bmatrix} \omega_1(n) \\ \omega_2(n) \\ \vdots \\ \omega_B(n) \end{bmatrix} + \mu \begin{bmatrix} x_1(n) \\ x_2(n) \\ \vdots \\ x_B(n) \end{bmatrix} \varepsilon \tag{13}$$

This equation dictates the updated value of the weight and performing the calculation makes up the major part of the training hardware and process.

When the system is ready to start learning, both the noisy and original data will be threshold decomposed continuously until the threshold decompose device is saturated and decomposed data is waiting to be processed by next stage. Noisy data will be fed into the

Window Generator. Original data will be decomposed by another Window Generator and its output represents the desired output.

When the threshold data is ready, the bottom level, $x_B(n)$ will be processed as in the filtering operation for one cycle (goes through all the weights and adds up appropriate one). After this cycle is finished, instead of storing the most significant bit (MSB), we compare this bit to the MSB of the desire output, the original data of $x_B(n)$. If they are the same, ε would be equal to 0, otherwise 1. According to Equation (12), if ε equals to 0, $x_B(n)$ is good and no update is required. If ε equals to 1, it needs to calculate the updated weight. After finding out the updated weight, it would be written back to the location of $x_B(n)$.

At this point, the bottom level of the threshold value can be discarded and then process the next level of threshold data, x_{B-1} using the same rule that determines $w_{B-1}(n+1)$. The above steps are repeated until all samples have been consumed. The logic of the training process is described in the flowchart shown in Figure 15.

A multiplication is needed in Equation (12). This complicates the process because multiplication required iterations of addition and is time consuming. The multiplication is implemented as follows.

After the system finds out ε equals to 1, the process flow will go to a calculation routine that will find the updated weight using the equation $\omega(n+1) = \omega(n) + \mu * x(n) * \varepsilon$. First, μ is multiplied with the current level of the threshold data, $x(n)$. Then the product is added to the current weight of same level.

```
   AAAAAABB
 *     CDEF
 -----------
 AAAAAABB000
  AAAAAABB00
   AAAAAABB0
 -----------
????????XXXX
```

Figure 16: Binary multiplication performed by hand

Multiplication is done by iteration of the addition function. Figure 16 shows the required multiplication. This figure calculates AAAAAA.BB * 0.CDEF.

1. If bit C is 1, AAAAAABB000 will be stored in STORED. Otherwise do nothing.
2. If bit D is 1, AAAAAABB00 will be added to value in STORED.
3. If bit E is 1, AAAAAABB0 will be added to value in STORED.
4. If bit F is 1, AAAAAABB will be added to value in STORED. The product is now in STORED and its value has 6 bits after decimal point.
5. Shift right STORED 4 positions to get rid of the least 4 bits after decimal point because the final value should be 6 integer bits and 2 decimal bits as set by the entire system.

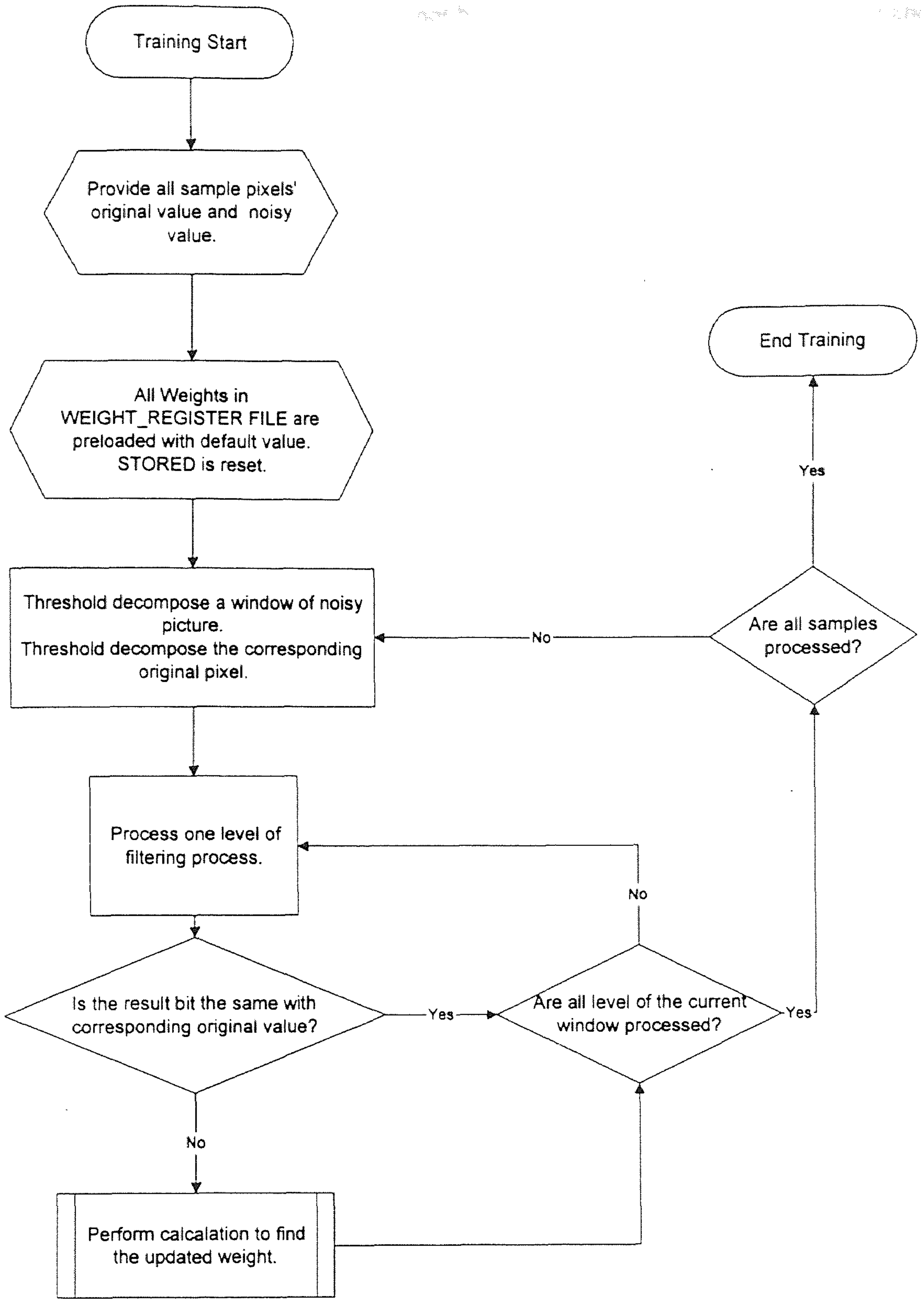
Training Start
Provide all sample pixels' original value and noisy value.
All Weights in WEIGHT_REGISTER FILE are preloaded with default value. STORED is reset.
Threshold decompose a window of noisy picture. Threshold decompose the corresponding original pixel.
Process one level of filtering process.
Is the result bit the same with corresponding original value?
No
Perform calcalation to find the updated weight.
Yes
Are all level of the current window processed?
No
Yes
Are all samples processed?
No
Yes
End Training

Figure 15: Flow chart of complete training process

In our implementation, we will not build a logic circuit exactly like the one above but with a little modification. In the above procedure, the system would require 11-bit register (AAAAAABB000 has 11 bits) and adder to do addition in order to take care of all bits. The reason for the last three bits is to make room for the biggest possible value after multiply by a 4-bit binary integer. If the multiplier has 5 bits, then the system would put in 4 bits into the register and of course, using a bigger register to handle the larger data. Since other system registers are 8 bits wide and 6 bits are allocated as integer bits and the other 2 bits are decimal bits, we will restrict the multiplication implementation to conform to the system standard. For the 1st bit of μ, add AAAAA.AB if condition to add is valid. In the 2nd bit of μ, add AAAA.AA if condition to add is valid. In the 3rd bit of μ, add AAA.AA if condition to add is valid. For the last bit, add AA.AA if condition to add is valid. At the end, we do not need to do alignment to the final result because its decimal point is already aligned. The only drawback of this modification is everything after the second bit of the decimal part will not be considered.

We have the μ * x(n). Now the last step is to add $\omega(n)$ to it. After adding $\omega(n)$, STORED contains the updated weight and be written back to the corresponding level (1,2, ... , B). The training process will proceed to the next level or pixel sample. See Figure 17 for complete logic of update weight calculation.

4.4 Implementation

The hardware required in this section is a superset of the hardware used in the filtering operation. The training phase will reuse the hardware component setup for filtering, and adds more complexity in terms of logic and components.

The following is an implementation of μ * X(n) where X(n) is noisy data.

1. INPUT-SEL-MUX must be set to select DATA as the input of the ADDER.
2. Reset STORE register to zero and makes μMUX to select the most significant bit of μ as the input to Load signal of STORED. By asserting Clear of STORED, select appropriate input of μMUX and clocks the system. Load the current level (threshold decomposed) of noisy data into DATA right-shift register. Then shift right one bit.
3. Start multiplication of the 1st (most significant bit of μ) bit.
4. Check the 1st bit of μ, if 1, perform the addition by clocking the system.
5. Select 2nd bit of μ, shift right DATA one bit. By asserting the Shift of DATA and clocking the system.
6. Start multiplication of the 2nd bit.
7. Check the 2nd bit of μ, if 1, perform the addition by clocking the system.
8. Select 3rd bit of μ, shift right DATA one bit. By asserting the Shift of DATA and clocking the system.
9. Start multiplication of the 3rd bit.
10. Check the 3rd bit of μ, if 1, perform the addition by clocking the system.
11. Select last bit of μ, shift right DATA one bit. By asserting the Shift of DATA and clocking the system.

12. Start multiplication of the 4th (last) bit.
13. Check the 4th bit of μ, if 1, perform the addition by clocking the system.
14. INPUT-SEL-MUX is set to select WEIGHT-REGISTER file as the input of the ADDER.
15. Add the current weight with value in STORED (the adjustment to the weight) by asserting Load of STORED and clocking the system.
16. Save the updated weight back to the WEIGHT-REGISTER file from STORED.

At this point, one level of threshold decomposed noisy data has been learned by the system. The above step will be repeated for the remaining threshold decomposed level and then for the remaining training data.

4.5 LogicWorks Sample

Figure 18 illustrates the complete system which can handle 8-bit pixel.

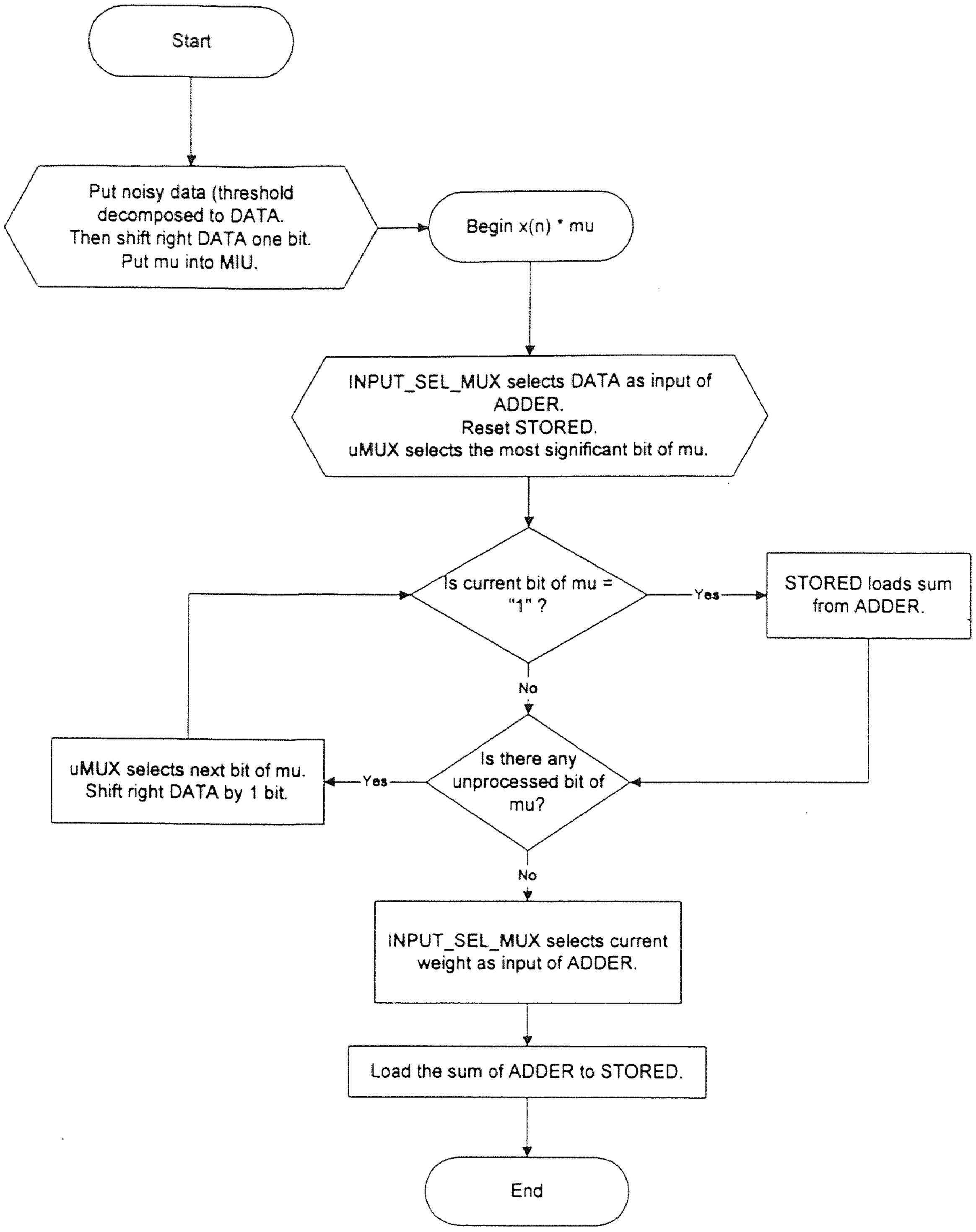
Start
Put noisy data (threshold decomposed to DATA. Then shift right DATA one bit. Put mu into MIU.
Begin x(n) * mu
INPUT_SEL_MUX selects DATA as input of ADDER. Reset STORED. uMUX selects the most significant bit of mu.
Is current bit of mu = "1" ?
Yes
STORED loads sum from ADDER.
No
Is there any unprocessed bit of mu?
Yes
uMUX selects next bit of mu. Shift right DATA by 1 bit.
No
INPUT_SEL_MUX selects current weight as input of ADDER.
Load the sum of ADDER to STORED.
End

Figure 17: Implementation logic of update weight

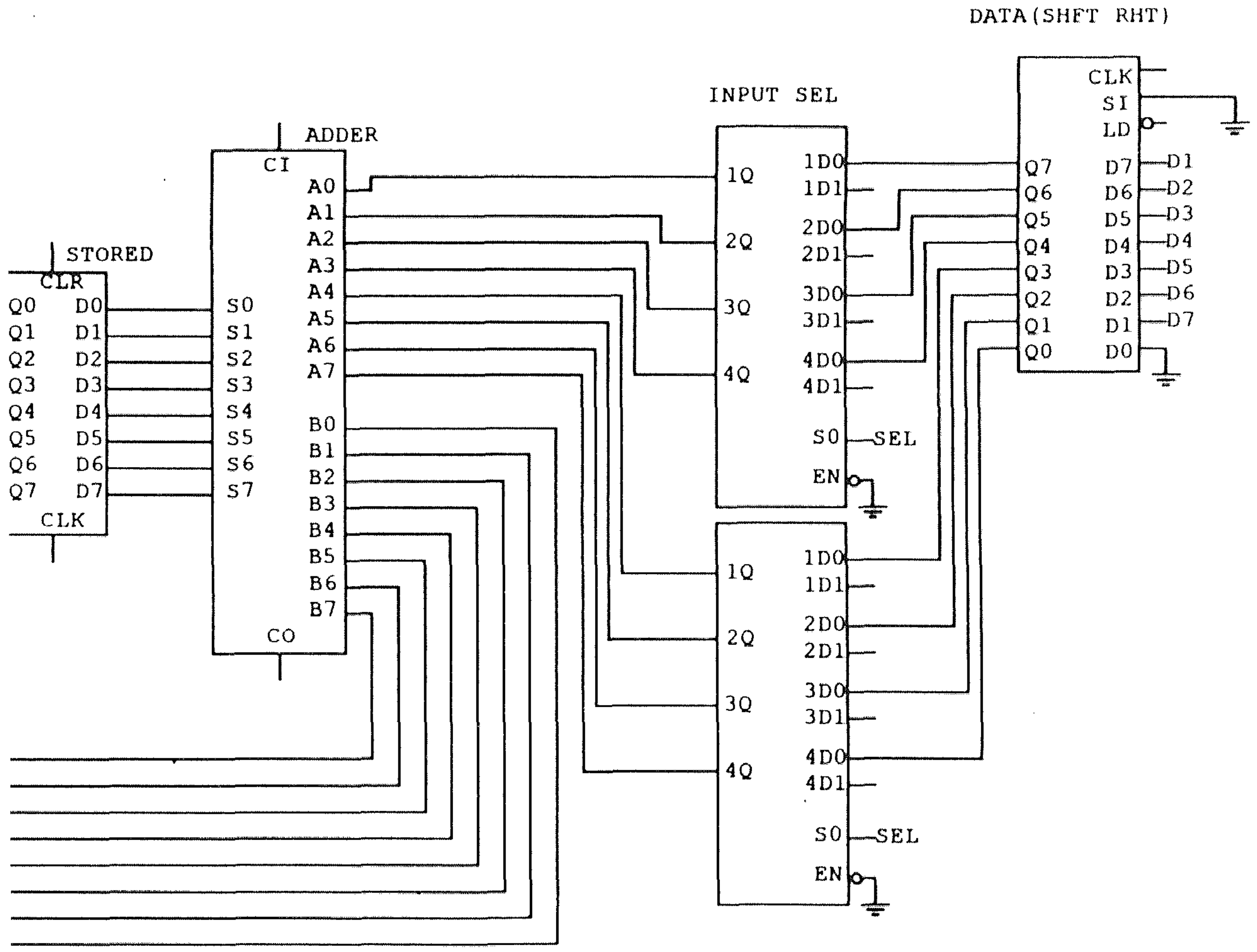

Figure 18: Circuit of Training Unit

CHAPTER 5

CONCLUSIONS & FUTURE RESEARCH

5.1 Conclusions

Two designs, one for single neuron implementation and another for multiple neuron implementation of GANF are presented in this thesis. The architectural differences between the single neuron implementation and the multiple neuron implementation is in the duplicated neurons. Using the current VLSI technology, putting in additional neuron would not result in any significant overhead. But the gain in performance is huge. Implementing a 7-level GANF in multiple neuron would speed up as much as 7 times than implementing it with a single neuron.

The flexibility in expanding a GANF is also an advantage. Chapter two gave an example where the window generator can accommodate different situations and requirements. The other components can also be expanded similar to the window generator.

5.2 Future Research

To use the GANF in a practical situation, a good compromise would be the use of a quadric neuron. Smaller window sizes would be desirable at the start of training to achieve proper generalization. Then, the window can be expanded to achieve increased performance.

A complete simulation of the multiple neuron design and a VLSI implementation should be considered.

APPENDIX A

HARDWARE BLOCK DIAGRAM

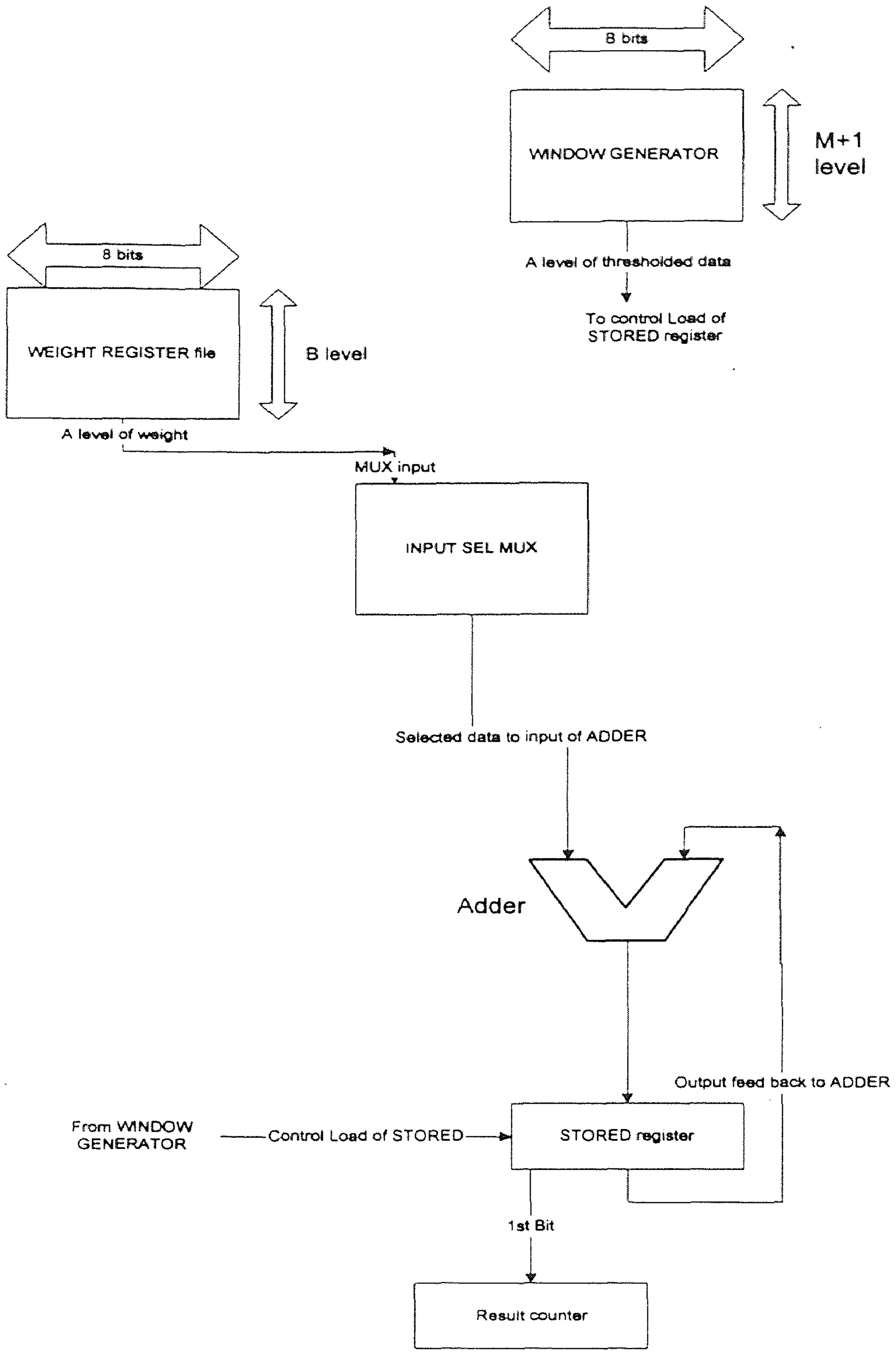

Figure A1: Hardware block diagram of filtering process

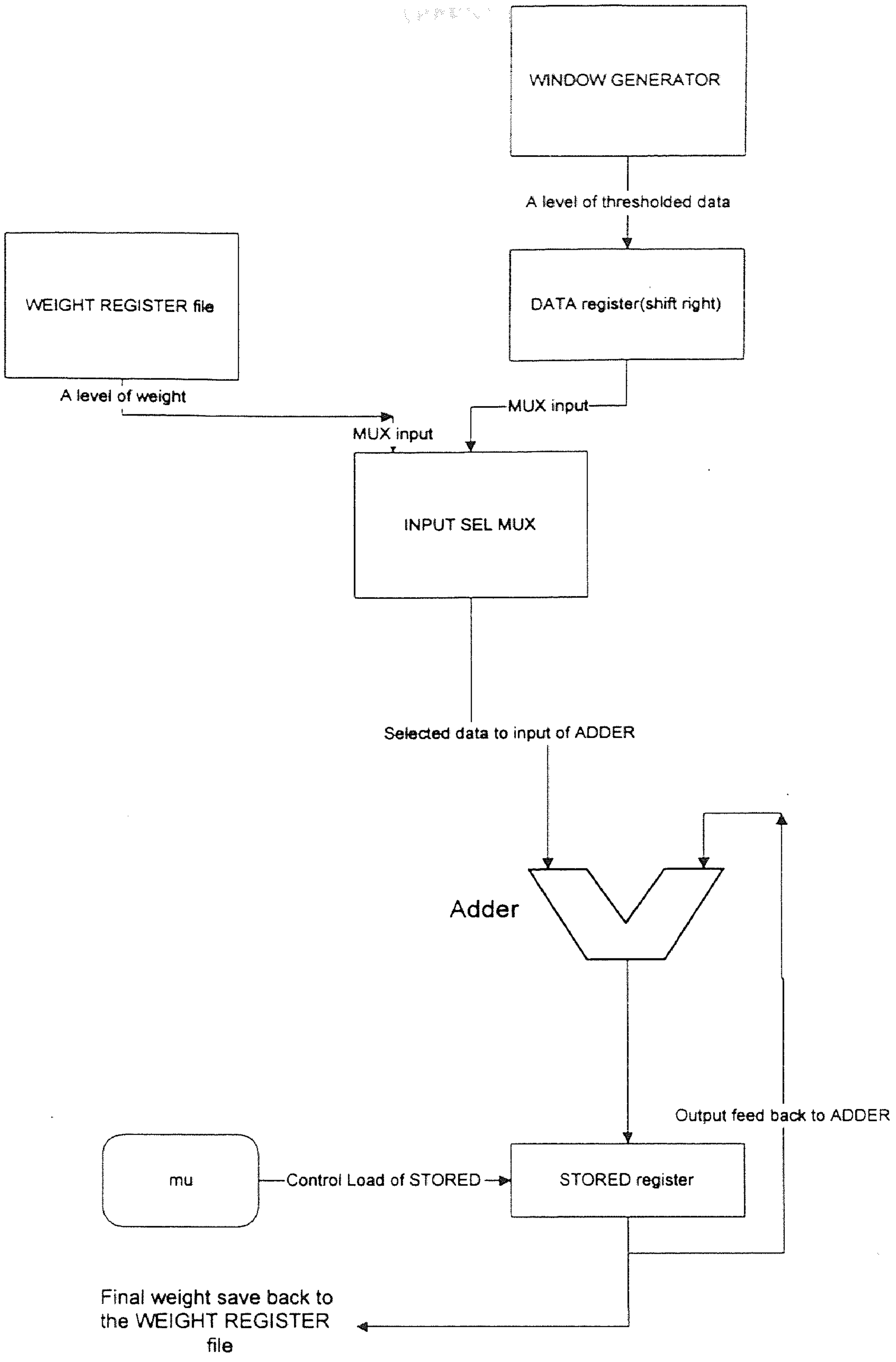
WINDOW GENERATOR
A level of thresholded data
WEIGHT REGISTER file
DATA register(shift right)
A level of weight
MUX input
MUX input
INPUT SEL MUX
Selected data to input of ADDER
Adder
Output feed back to ADDER
mu
Control Load of STORED
STORED register
Final weight save back to the WEIGHT REGISTER file

Figure A2: Hardware block diagram of training unit

APPENDIX B

THE PROGRAM THAT GENERATE A PICTURE

```
/*****************************************************
 * Author Name : Chunyip Tam
 * Purpose : This program is to generate a picture which has
 *  height equal to picture_height and width equal to picture_width.
 *  The value of each pixel represent the color or grayscale of
 *  that pixel.  For easy recognition, the value is a combination
 *  of three integers, X, 0 and Y.  They combine as X0Y to form the
 *  value.  X represents the number of row while zero is the top.
 *  Y represents the number of column from left to right.  0 is inserted
 *  between X and Y.
 *
 * To use : edit the value of picture width and height with the desired value
 *        then compile the program as "c++ picgen.c"
 *        then type "a.out"
 *        then result is stored in the file "picture"
 *****************************************************/


#include <iostream.h>
#include <fstream.h>

const int picture_width=12;
const int picture_height=13;

main ()
{
        ofstream out("picture", ios::out);

        if ( !out ) {
                cerr << "cannot open picture " << endl;
                exit (-1);
                }

        for (int x=0; x<picture_height; x++ )  {
          for (int y=0; y<picture_width; y++ )
            out << x << "0" << y << " ";
          out << endl;  }

        out.close();
}
```

THE SIMULATION PROGRAM OF WINDOW GENERATOR

```
/* This Program is to simulate the window generation     */
/* process for GANF                    */
/* To run this program, compile with C++ compiler first.  */
/* example: g++ wingen.c                    */
/* After compilation, an executable named a.out is created.  */

/* Then the next step is to run the window generation    */
/* program which will generate windows according to input */
/* provided.  Do the following to run the program:     */
/* At the prompt, type a.out input <enter>          */
/* where "input" is the input file name.          */

/* The format of the input file has to be in ASCII.     */
/* Use the output generated by PICGEN.C      */
/* Each line represents one horizontal line of pixels    */
/* on the picture.                    */

#include <iostream.h>
#include <fstream.h>

// Modify the following line with proper value

const int WindowHeight = 3;
const int WindowWidth  = 3;
const int PictureHeight = 8;
const int PictureWidth  = 8;

// Do not modify anything from this point

int NumOfReg;         //Total number of Shift Registers required
int SkipBeforeRead;   //Number of registers skip before next line of window
int TotalReadPerRow;  //Number of windows generated in each horizontal scan
int TotalRow;         //Total number of horizontal scan

void take(const int []);
void shift(int [], int, ifstream&);

// Calculate value from the parameter given
void calculate ()
{
        NumOfReg = PictureWidth * (WindowHeight - 1) + WindowWidth;
        SkipBeforeRead = PictureWidth - WindowWidth;
        TotalReadPerRow = PictureWidth - WindowWidth + 1;
        TotalRow = PictureHeight - WindowHeight + 1;
}


// Subroutine to shift the entire register set to left
void shift(int Array[], int Frequency, ifstream& fin)
{
        int x,y;

        for (x=1 ; x<=Frequency ; x++)
        {
                for (y=0 ; y<NumOfReg-1 ; y++)
                        Array[y] = Array[y+1];
                fin>>Array[y];
        }
}

// Subroutine to read one set of window
void take(const int Array[])
{
        int ReadingPointer = 0;

        for (int x=1 ; x<=WindowHeight ; x++ )
        {
                for (int y=1 ; y<=WindowWidth ; y++)
                {
                        cout<<Array[ReadingPointer]<<"  ";
                        ReadingPointer++;
                }
                ReadingPointer = ReadingPointer + SkipBeforeRead;
                cout<<endl;
        }
}

main (int argc, char **argv)
{

        //Program section to declare reading pixel from a file
        ifstream f_in(argv[1]);
        if (!f_in) { cerr<<"error opening file\n"; exit(1); }

        calculate(); //calculate all parameter for the process
        // Declaration of shift registers required for the process
        int Register[NumOfReg];

        // Program section to initialize the register set
        for (int x=0 ; x<NumOfReg ; x++)
                f_in>>Register[x];

        // Program section to generate window
        for (x=1 ; x<=TotalRow ; x++)
        {
                for (int y=1 ; y<=TotalReadPerRow ; y++)
                {
                        take(Register);
                        shift(Register, 1, f_in); //Shift once before next window on same
horizontal scan
                        cout<<endl;  //Separate the next set of window
                }
                shift(Register, WindowWidth-1, f_in); //Before new horizontal scan, shift
out all unused register

                //The parameter above passed to shift is WindowWidth-1
                //rather than WindowWidth because one has been shift by
                //the regular shift after each reading of a window set.
        }
}
```

REFERENCES

1. Henry Steven Hanek, "Simplification Of The Generalized Adaptive Neural Filter and Comparative Studies With Other Nonlinear Filters," *Master Thesis, Department of Electrical and Computer Engineering, New Jersey Institute of Technology, October 1993.*

2. Z. Zhang and N. Ansari, "Structure and Properties of Generalized Adaptive Neural Filters for Signal Enhancement," *IEEE Trans. on Neural Networks,* vol. 7, no. 4, pp. 857-868, July 1996.

3. H. Hanek and N. Ansari, "Speeding Up the Generalized Adaptive Neural Filters," *IEEE Trans. on Image Processing,* vol. 5, no. 5, pp. 705-712, May 1996.

4. L. Yin, J. Astola, and Y. Neuvo, "A new class of nonlinear filter-neural filters," *IEEE Trans. signal Processing*, vol. 41, no. 5, pp.1201-1222, Mar. 1993.

5. V. N. Vapnik and A. Ya. Chervonenkis, "On the Uniform Convergence of Relative Frequencies of Events to Their Probabilities," *Theory of Probability and its Applications*, vol. 16, no. 2, 1971.

6. J. H. Lin, Y. T. Kim and G. Soemarwoto, "Nonlinear Filtering Techniques Based on a Threshold Decomposition Architecture," *Proc. of the 26th Annual Conference of Information, Science and System,* Princeton, NJ, Mar. 18-20, 1993.

7. J. H. Lin and Y. T. Kim, "Fast Algorithms for Training Stack Filters," *IEEE Trans. on Signal Processing*, vol. 42, no. 4, pp. 772-781, April 1994.

8. M. Morris Mano, *Computer Engineering: Hardware Design.* Englewood Cliffs, NJ: Prentice-Hall, 1988.