ECE 590
Digital Systems Design using Hardware Description Language

Final Project Report

General associative memory based on incremental neural network

Project Group

Vishwas Reddy Pulugu
Bharath Reddy Godi
Surendra Maddula
Vasu Gankidi
Harsh Sharma
Nikhil Marda

Contents

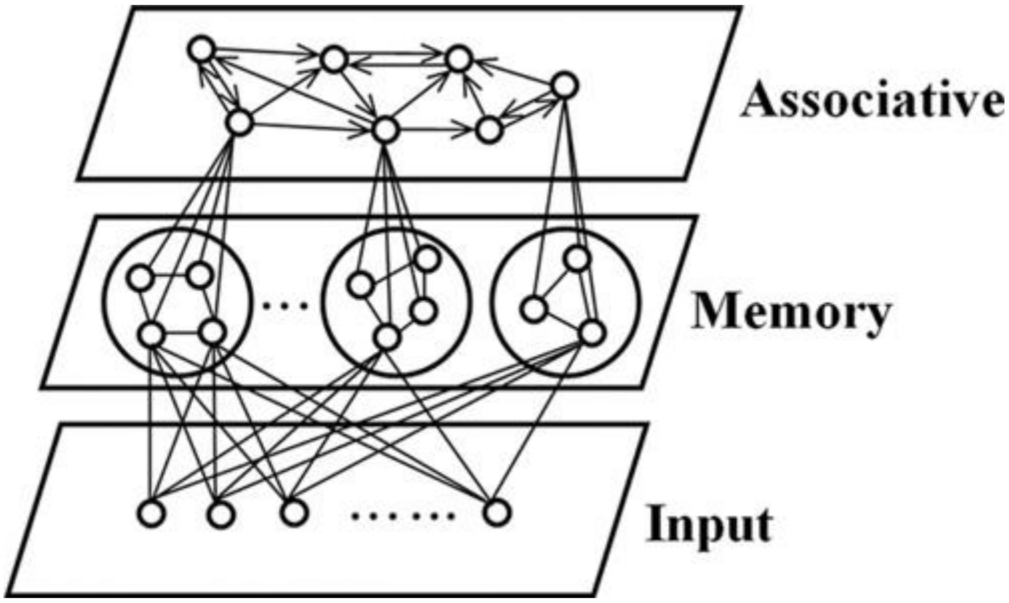
Chapter 1

1. INTRODUCTION

1.1 GENERAL ASSOCIATIVE MEMORY (GAM)

General Associative memory (GAM) system stores data in distributed fashion, which is addressed through contents. GAM can recall information from incomplete or garbled inputs. The GAM is a network consisting of three layers: an input layer, a memory layer and associative layer.

1.2 GAM Structure



1.2.1 Input Layer:

This layer accepts key vectors, response vectors and associative relationships between these vectors.

Key vector and response vectors are the vector inputs to the system which can be vectors of images, phrases etc.

1.2.2 Memory Layer:

Memory Layer stores the input vectors into subnetworks called classes. A subnetwork is a group of nodes and relations between the nodes. Each node represents an image vector. A Weight vector is associated with each node and this vector is updated in learning phase.

Contents of a node vector in memory layer:

Notation	Meaning
c_i	Class name of node i
$\mathbf{W}_i$	Weight vector of node i
Th_i	Similarity threshold of node i
M_i	The number of patterns represented by node i
E_i	Set of nodes connected with node i
I_i	Index of node i in sub-network c_i

1.2.3 Associative Layer:

Every class in memory layer is represented by a node in associative layer. It also stores the association between the key vector and response vectors belonging to key and response class respectively. Nodes are connected with arrows where node at beginning of arrow indicates key class and end of arrow indicates response class.

Contents of a node in associative layer:

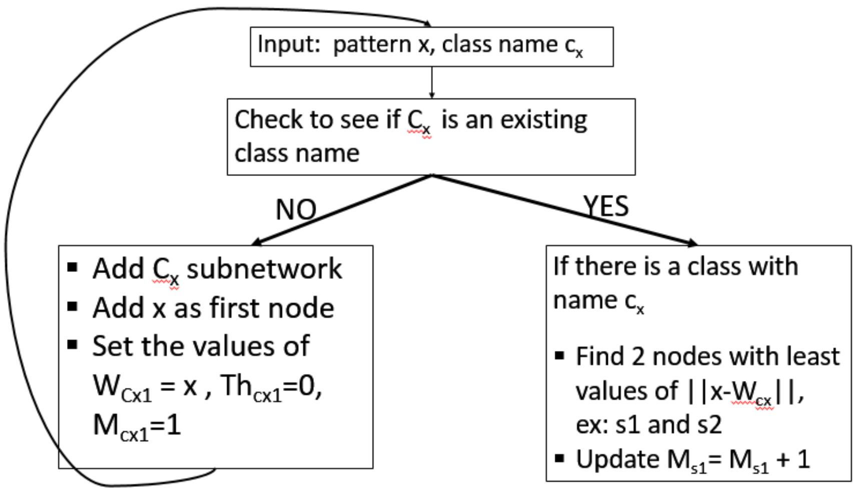
Notation	Meaning
c_i	Class name of node i
m_i	Associative index of node i
$\mathbf{W}_i$	Weight of node i
RC_i	Response classes of node i
T_i	Time order of current node i in a temporal sequence
TL_i	Time order of the latter item of node i in a temporal sequence
TF_i	Time order of the former item of node i in a temporal sequence
B	Node set of the associative layer
D	Connection (arrow edge) set of the associative layer
(i,j)	Connection (arrow edge) from node i to node j
$W_{(i,j)}$	Weight of connection (i,j)

2. GAM Learning Algorithm

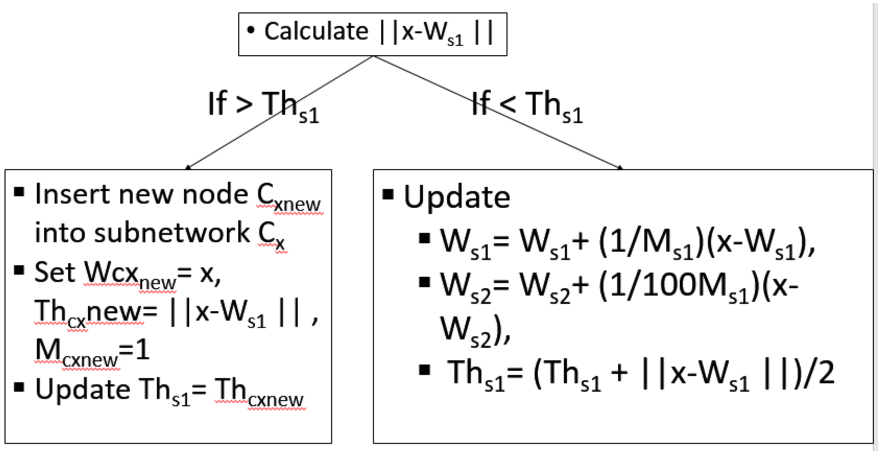
2.1 Algorithm 1 - Memory Layer Learning:

- Consider, X is an input image vector, and it belongs to class Cx.

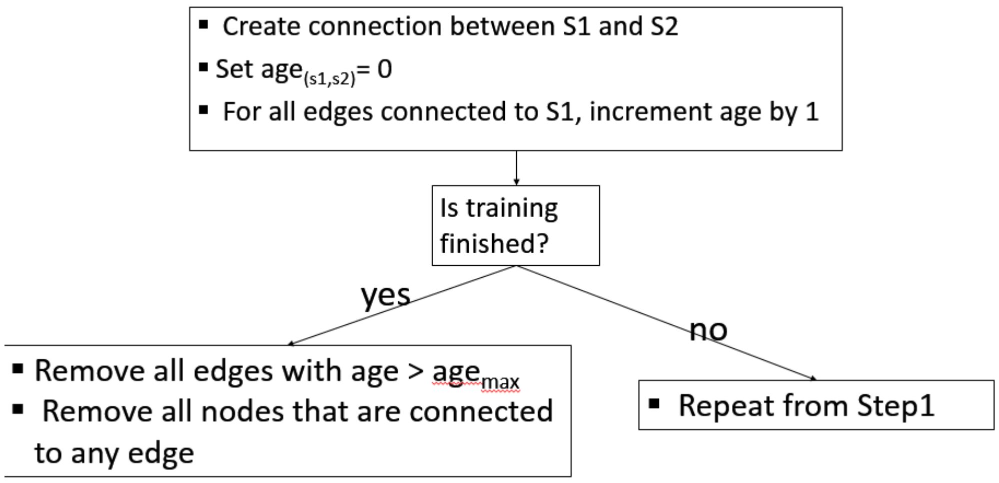
- For every new input we check the memory layer to see if the class already exists in the memory.
- If the class is not in the memory, then we insert a class with name Cx and add X as its first node. Set the values of $W_{Cx1} = x$, $Th_{cx1}=0$, $M_{cx1}=1$.
- If the class exists in memory find 2 closest nodes for the input vector X.


Input: pattern x, class name c_x
Check to see if C_x is an existing class name
NO
YES
Add C_x subnetwork
Add x as first node
Set the values of $W_{Cx1} = x$, $Th_{cx1}=0$, $M_{cx1}=1$
If there is a class with name c_x
Find 2 nodes with least values of $||x-W_{cx}||$, ex: s1 and s2
Update $M_{s1}= M_{s1} + 1$

- Compare the Euclidean Distance of first minimum with input vector to Th of the first minimum.
- If it is $>Th_{s1}$, then input vector should be added as new node to the class. And update the values: $Wcx_{new}= x$, $Th_{cx}new= ||x-W_{s1}||$, $M_{cxnew}=1$, $Th_{s1}= Th_{cxnew}$.
- If it is <Ths1 , then input vector will not be inserted into the class. Update the values: $W_{s1}= W_{s1}+ (1/M_{s1})(x-W_{s1})$, $W_{s2}= W_{s2}+ (1/100M_{s1})(x-W_{s2})$, $Th_{s1}= (Th_{s1} + ||x-W_{s1}||)/2$


Calculate $||x-W_{s1}||$
If $> Th_{s1}$
If $< Th_{s1}$
Insert new node C_{xnew} into subnetwork C_x
Set $Wcx_{new}= x$, $Th_{cx}new= ||x-W_{s1}||$, $M_{cxnew}=1$
Update $Th_{s1}= Th_{cxnew}$
Update
$W_{s1}= W_{s1}+ (1/M_{s1})(x-W_{s1})$,
$W_{s2}= W_{s2}+ (1/100M_{s1})(x-W_{s2})$,
$Th_{s1}= (Th_{s1} + ||x-W_{s1}||)/2$

- Create a connection between the first and second minimum nodes, set the ag of the connection to zero. Increment age of all edges connected to first minimum by 1.
- If training is finished, remove all the edges with age> age_{max} and remove all the isolated nodes from all classes.


▪ Create connection between S1 and S2
▪ Set age(s1,s2)= 0
▪ For all edges connected to S1, increment age by 1
Is training finished?
yes
no
▪ Remove all edges with age > agemax
▪ Remove all nodes that are connected to any edge
▪ Repeat from Step1

3. Design

3.1 **Algorithm 1**: Memory Layer Learning: Controller and Data-path design

The design for algorithm is a FSMD.

3.1.1 Memory

The memory holds the contents of memory layer including the classes, subnetworks of nodes, node vectors, weight vectors of the nodes, Threshold.

3.1.2 Registers

There are 11 registers.

Reg_x-: input image vector

Reg_cx: class name

Reg_node_min1, Reg_node_min2: nodes of first and second minimums.

Reg_ED_min1, Reg_ED_min2: Euclidean distance of input vector to with first and second minimums

Reg_Ws1, Reg_Ws2: Weight vectors of first and second minimums.

Reg_Ths1: Threshold of first minimum

Reg_Ms1: M value of first minimum

Reg_node_max: stores the fixed number of nodes per class

3.1.3 comparator

This compares two input vectors and gives the results of greater than, less than and equal. The output of this comparator is of 2 bits.

3.1.4 Up-Counter

Counts from zero when enable is asserted. Resets to zero when load is asserted.

3.1.5 Update Ws1_Ws2_Ths1

The inputs of this block will be from Reg_Ws1, Reg_Ws2, Reg_Ths1, Reg_Ms1. This calculates the values of Ws1, Ws2, Ths1 according to the equations: $W_{s1}= W_{s1}+ (1/M_{s1})(x-W_{s1})$, $W_{s2}= W_{s2}+ (1/100M_{s1})(x-W_{s2})$, $Th_{s1}= (Th_{s1} + ||x-W_{s1}||)/2$

3.1.6 Node Counter

Keeps track of the number of nodes inserted into each class

3.1.7 ED Calculator

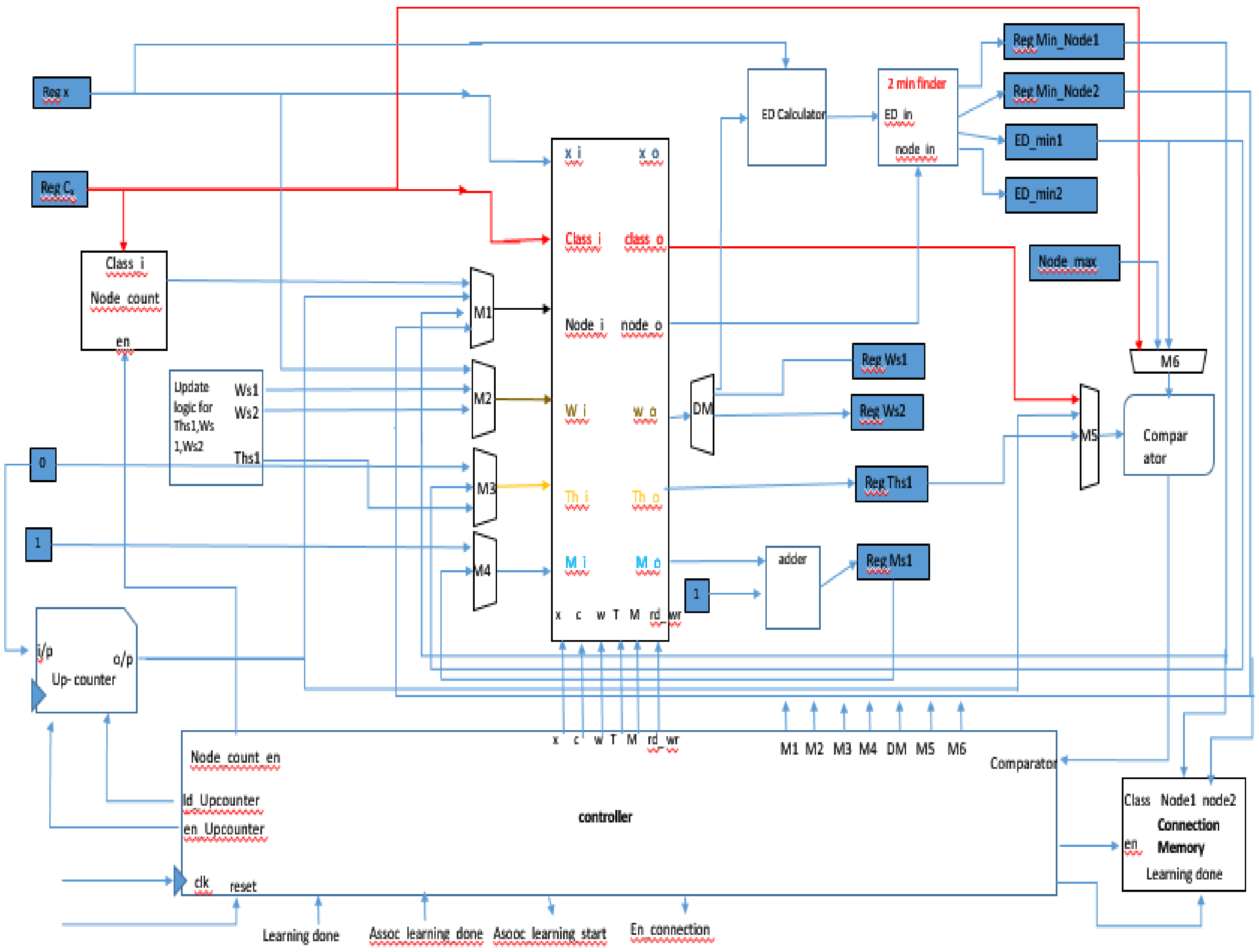
This block calculates the Euclidean distance between input vector and weight vectors of other nodes in the input class.

3.1.8 2 Min Finder

This blocks finds two minimum values among the input vectors. One input is given every cycle. It compares the input value with previous values and decides the 2 minimums. It outputs two min nodes and their Euclidean distances.

3.1.9 Connection Memory

Stores the connection information between all nodes in all the classes of the memory layer. Increments and updates the age values accordingly.

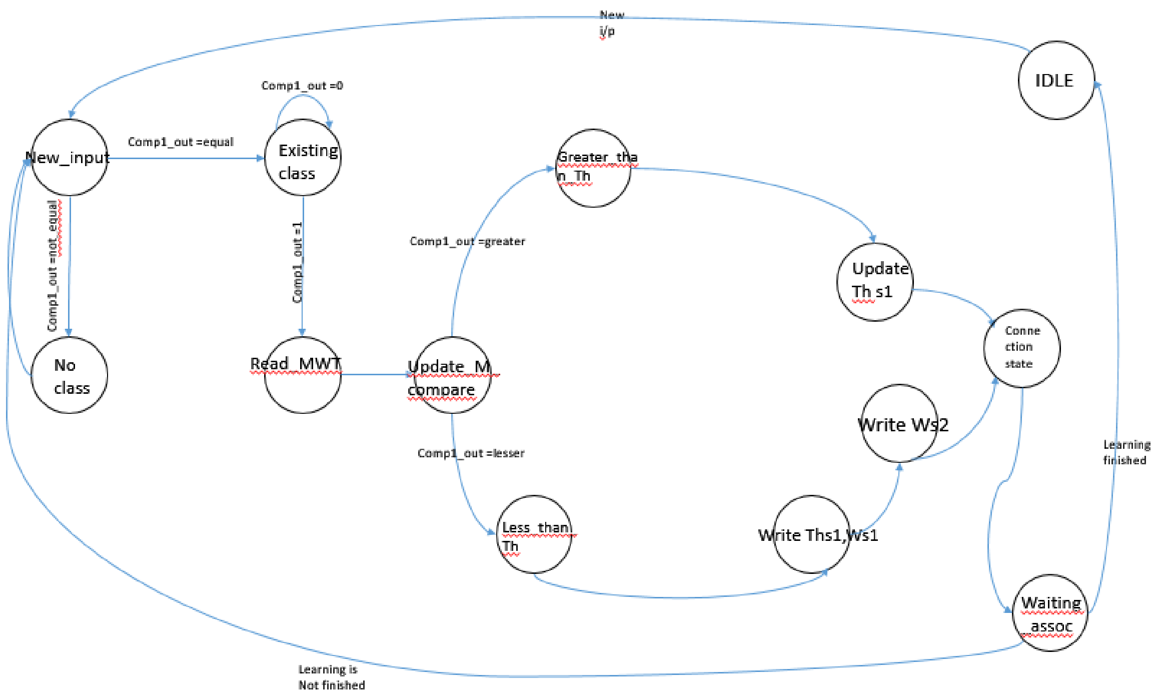


3.1.10: **Controller**

Controller is a finite state machine with 13 states: idle, waiting_assoc, new_input, no_class, existing_class, read_MWT, update_M_compare_Th_ED, greater_than_Th, less_than_Th, update_Ths1, write_Ws1_Ths1, write_Ws2, Connections.

The state transition is as shown in the figure.

,



State Transition Table:

Present State	Learning_finished	Assoc learning done	Comparator	Next state
Idle	1	x	X	Idle
Idle	0	x	x	New_input
Waiting_assoc	X	1	x	Idle
Waiting_assoc	x	0	x	Waiting_assoc
New_input	x	X	00	Existing_class
New_input	x	X	11 or 10 or 01	No_class
No_class	x	x	x	New_input
Existing_class	x	x	00	Read_MWT
Existing_class	X	X	11 or 10 or 01	Existing_class
Read_MWT	x	x	X	Update_M_compare_Th_ED
Update_M_compare_Th_ED	x	x	10	Greater_than_Th
Update_M_compare_Th_ED	x	x	00 or 01 or 11	Less_than_Th
Greater_than_Th	x	x	x	Update_Ths1
Less_than_Th	x	x	x	Write_Ws1_Ths1
Update_Ths1	x	x	x	connections
Write_Ws1_Ths1	x	x	x	Waitin_assoc
Write_Ws2	x	x	x	connections
connections	x	x	x	Waiting_assoc

State Outputs Table:

State	x	c	w	T	M	Rd_wr	M1	M2	M3	M4	M5	M6	DM	Ld_upcounter	En_upcounter	En_node_counter	Assoc_learning_start	En_connections
Idle	0	0	0	0	0	0	11	11	11	11	11	11	11	1	0	0	0	0
Waiting_assoc	0	0	0	0	0	0	11	11	11	11	11	11	11	1	0	0	1	0
New_input	0	1	0	0	0	0	11	11	11	11	00	00	11	1	0	0	0	0
No_class	1	1	1	1	1	1	00	00	00	00	11	11	11	1	0	1	0	0
Existing_class	0	0	1	0	0	0	01	11	11	11	01	01	00	0	1	0	0	0
Read_MWT	0	0	1	1	1	0	10	11	11	11	11	11	01	1	0	0	0	0
Update_M_compare_Th_ED	0	0	0	0	1	1	01	11	11	01	10	10	11	1	0	0	0	0
Greater_than_Th	1	1	1	1	1	1	00	00	01	00	11	11	11	1	0	1	0	0
Less_than_Th	0	0	1	0	0	0	11	11	11	11	11	11	10	1	0	0	0	0
Update_Ths1	0	0	0	1	0	0	10	11	01	11	11	11	11	1	0	0	0	0
Write_Ws1_Ths1	0	0	1	0	0	0	10	01	10	11	11	11	11	1	0	0	0	0
Write_Ws2	0	0	1	0	0	0	11	10	11	11	11	11	11	1	0	0	0	0
connections	0	0	0	0	0	0	11	11	11	11	11	11	11	1	0	0	0	1

4 VHDL Code

4.1 Memory Layer Structure:

4.1.1 **Memory Structure for Memory Layer**

- ➔ We chose record type for the memory because it is easy to access all the values of a node and class if node and class address are given.
- ➔ We chose separate memory structure to represent connections between the nodes and age of the connections.

- Structure of single node: each node structure stores the values of node vector,class name, weight (W), Threshold (Th), patterns represented (M).

```
type node_T is record      --single node structure
x:std_logic_vector (image_vector_len-1 downto 0);
C:std_logic_vector (7 downto 0);
w: std_logic_vector (7 downto 0);
Th: std_logic_vector (7 downto 0);
M:std_logic_vector (7 downto 0);
E:std_logic_vector (7 downto 0);
```

```
l:std_logic_vector (7 downto 0);
end record node_T;
```

- Group of nodes for a class and structure of a class

```
type nodes_T is array (node_count downto 1) of node_T;   --array of nodes

type class_T is record          --single class structure
class_name: std_logic_vector(7 downto 0);
node: nodes_T;
end record class_T;
```

- Structure of entire memory layer

```
type memory_T is array (class_count downto 1) of class_T;    --mem is array of classes
```

- Structure of Connections memory

```
      -------------------connection mem-------------------

 type connection_T is record
connection_presence: std_logic;
age:integer;
end record connection_T;

 type connections_for_node_T is array (node_count downto 1) of connection_T;

type connections_for_single_node_T is record
connected_node:connections_for_node_T;
end record;

type connection_set_for_class_T is array (node_count downto 1) of
connections_for_single_node_T ;

type connection_set_for_single_class_T is record
node: connection_set_for_class_T;
end record;

type connection_mem_T is array (class_count downto 1) of
connection_set_for_single_class_T;

        -------------------connection mem-------------------
```

4.1.2 Algorithm 1- Modules

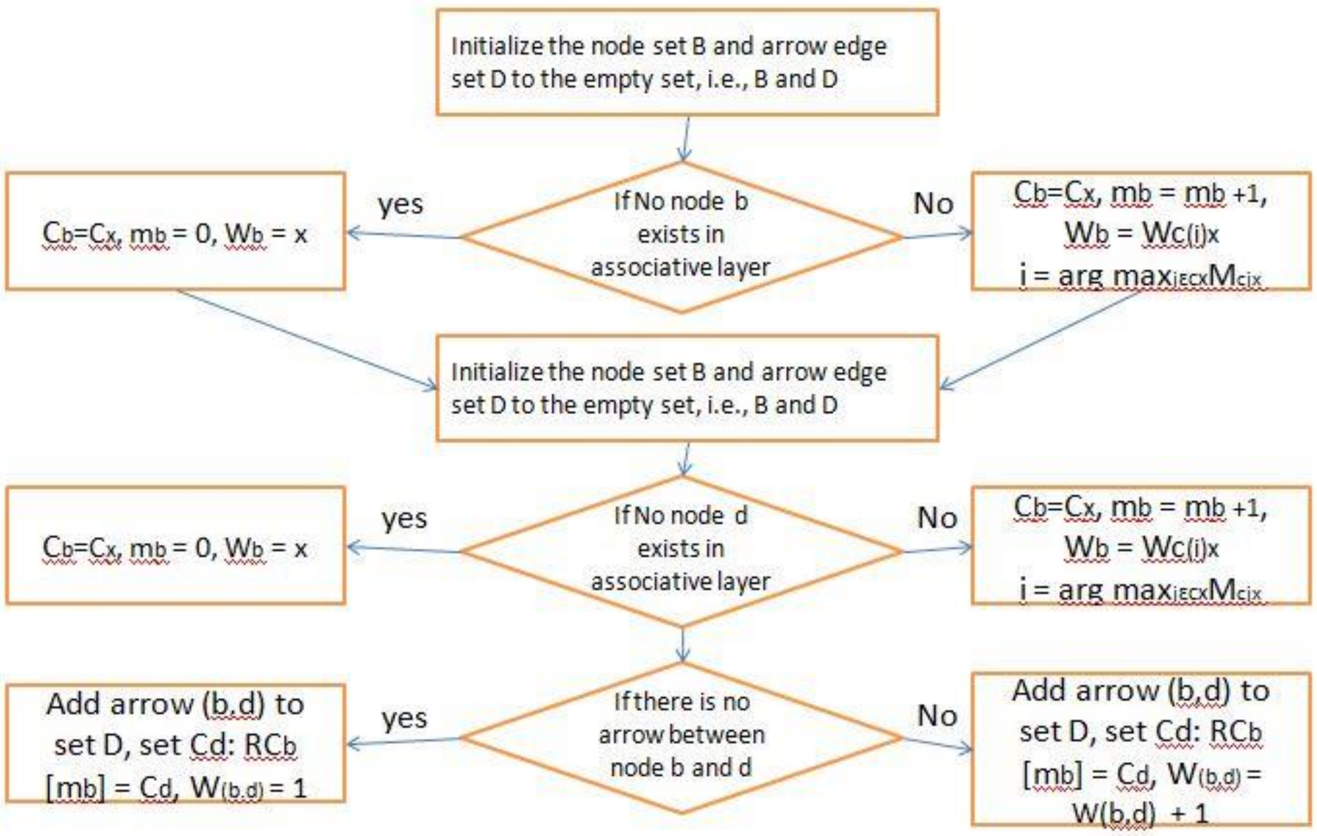
All the modules are attached in the Zip folder

Chapter 2

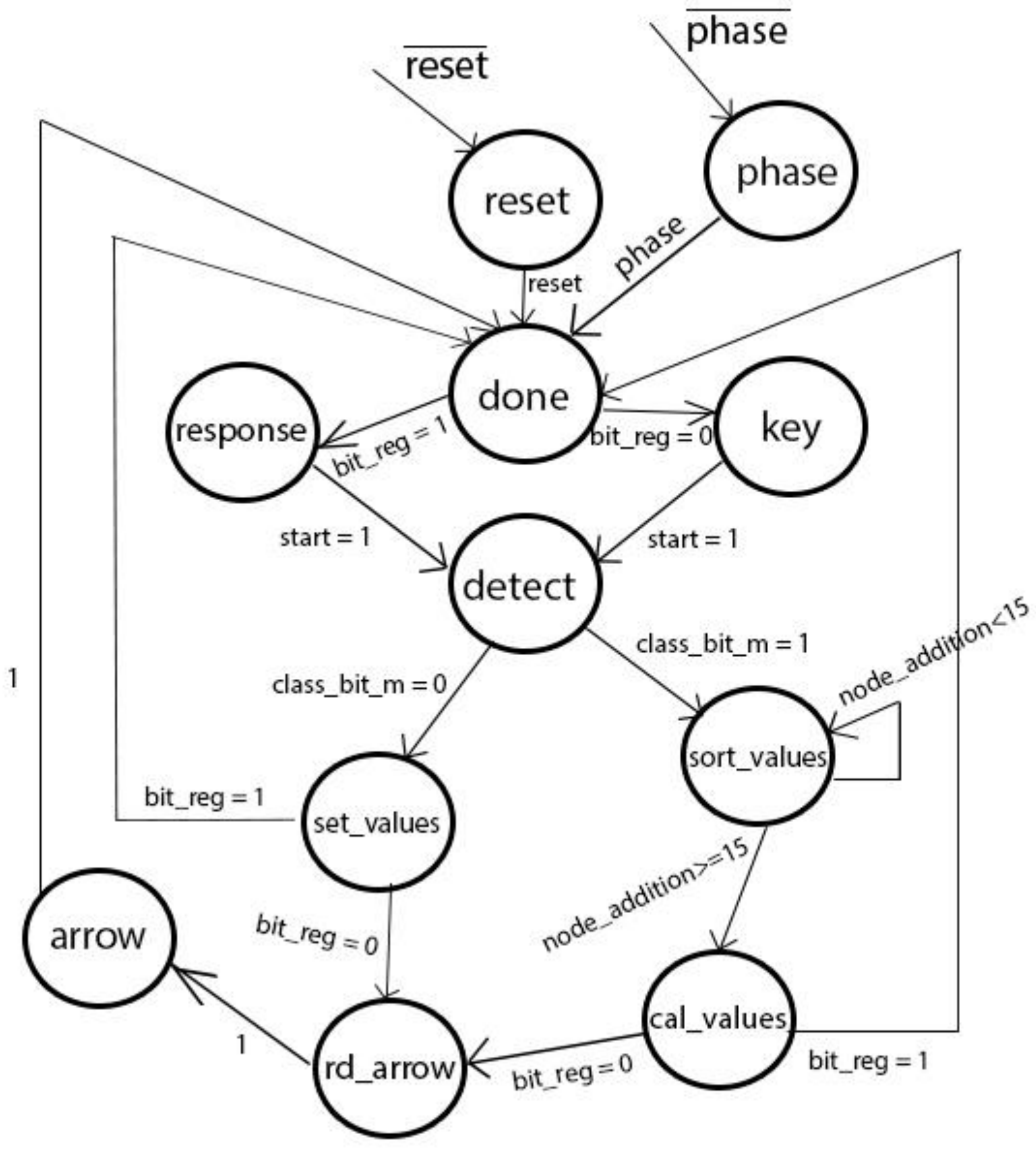
2.1 Algorithm – 2: Introduction

The associative layer builds associations between key and response vectors. Key vectors belong to a key class and response vectors belong to a response class. The nodes are connected with arrow edges. Each node represents one class—the beginning of the arrow indicates the key class and the end of the arrow indicates the corresponding response class. During training of the associative layer, we use association pair data –the key vector and response vector–as the training data. Such data input incrementally into the system. First, Algorithm 1 is used to memorize information of both the key and the response vectors. If the key class (or response class) already exists in the memory layer, the memory layer will learn the information of the key vector (or the response vector) by adding new nodes or tuning weights of nodes in the corresponding sub network. If the key or response class does not exist in the memory layer, it builds a new sub network to memorize the new key or response class with the key or response vector as the first node of the new sub network. The class name of the new class is sent to the associative layer. In the associative layer, if nodes that represent the key and response class already exist, we connect their nodes with an arrow edge. The beginning of the arrow corresponds to the key class node and the end corresponds to the response class node. This creates an associative relationship between the key class and the response class. If no node represents the key (or response) class within the associative layer, we add a node to the associative layer and use that node to express the new class. Then, we build an arrow edge between the key class and response class. Algorithm 2 gives the details for training the associative layer with the key and response vectors as the input data. In Table 3, we list the contents of the node in the associative layer and some notations used in the following algorithms.

2.2 Flow chart for an Algorithm - 2:


Initialize the node set B and arrow edge set D to the empty set, i.e., B and D
If No node b exists in associative layer
yes
C_b=C_x, m_b = 0, W_b = x
No
C_b=C_x, m_b = m_b +1, W_b = $W_{c(i)x}$ i = arg $\max_{j\in c_x} M_{cjx}$
Initialize the node set B and arrow edge set D to the empty set, i.e., B and D
If No node d exists in associative layer
yes
C_b=C_x, m_b = 0, W_b = x
No
C_b=C_x, m_b = m_b +1, W_b = $W_{c(i)x}$ i = arg $\max_{j\in c_x} M_{cjx}$
If there is no arrow between node b and d
yes
Add arrow (b,d) to set D, set C_d: RC_b [m_b] = C_d, $W_{(b,d)}$ = 1
No
Add arrow (b,d) to set D, set C_d: RC_b [m_b] = C_d, $W_{(b,d)}$ = W(b,d) + 1

2.3 State Machine diagram:



There are total 11 states in the design of an associative memory controller. The states are reset, phase, response, done, key, detect, set_values, sort_values, cal_values, rd_arrow and arrow states. Below is the detail explanation of each state,

reset state: The controller enters into the **reset phase** when an active low signal is received. During this phase all internal registers are cleared and goes to done state immediately after the reset has been removed.

phase state: Associative memory controller sits in **phase state**, whenever the system is in out of learning state. When phase is asserted, it is called learning state of associate memory. When the phase is de-asserted, the system is said to be in recall state of the machine and it comes out of phase state, whenever phase is ‘1’ and enters into the done state.

done state: In done state, start output signal is asserted which triggers the memory controller to start the process.

response state: Similar to the associative layer controller, the memory controller will also assumed to have a done state, whose signal becomes a handshake signal to the other module. It moves from response state to **detect state**.

key_state: When the response signal is detected, the system processes and waits in the response state. Untill and unless the **start** signal is asserted high, the machine transfers from **key_state** to **detect state**.

detect state: This state checks the associative memory and tells whether the class given is already been accessed and created a corresponding associative node. When the **class_bit_m = '1'**, it enters into the **sort_values** else if **class_bit_m = '0'**, it enters into the **set_values**.

sort_values state: Controller enters into this state whenever the node of a corresponding class is present in an associative memory. During this state, it finds out the node in a class with the highest Mp index rank.

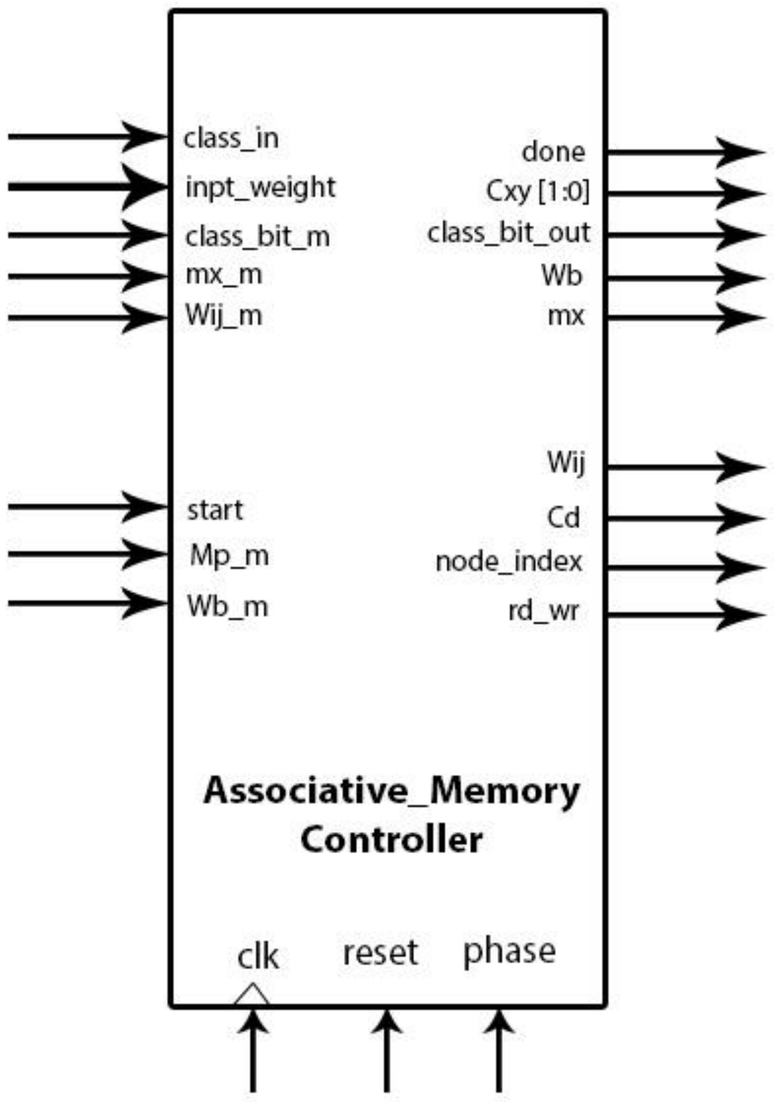
cal_values: Controller enters into this state once after it comes out of sort_values state inorder to set the associative index, class name and input weights. Class weights are adjusted tuned to the one which have the node rank Mp highest in **sort_values state**.

set_values state: Controller enters into this state when the class input is doesn't have a corresponding node in associative memory. During this state it sets the associative index, class name, input weights are set.

rd_arrow state: Controller enters into this state when it receives both the key and response inputs. This state is to check whether there is any relation built already between the given key and response inputs.

arrow state: This state occurs after the **rd_arrow** state finishes. The arrow weight will be added by one and the relation between key and response is loaded with respect to the associative index.

<u>2.4 Block Diagram of the Associative Memory controller chip</u>:



As shown in above figure,

INPUTS:

i. class_in = label of a class.
ii. input_weight = The input vector weight.
iii. Class_bit_m = One bit indicator tells the presence of a class in associative memory
iv. mx_m = The highest associative index value
v. Wij_m = The weight of an arrow relation between i(key) and j(response) inputs
vi. start = One bit input, which starts the process of storing the data in associative memory
vii. Mp_m = No.of patterns represented by a node

viii. Wb_m = Input weight vector of that node
ix. reset = One bit input which reset the controller at active low signal
x. phase = One bit input which set’s the phase of an input. (‘1’ = Learning, ‘0’ = recall)

OUTPUTS:

i. done = It is an one bit output which indicates the completion of the parameters record into an associative memory
ii. Cxy = The class label output which works as index to associative memory
iii. Class_bit_out = It is one bit value which sets to indicate the presence of the class in associative memory as a node.
iv. Wb = Input weight vector that writes into an associative memory
v. mx = The associative index that writes into an associative memory
vi. Wij = The arrow weight between i and j node that writes into associative memory
vii. Cd = The reponse class writes into the associative memory.
viii. node_index = The value to index the node set in a class
ix. rd_wr = One bit output which sets ‘0’ for read and ‘1’ for write.

2.5 Control & Data Path:

Following is the code for the control unit,

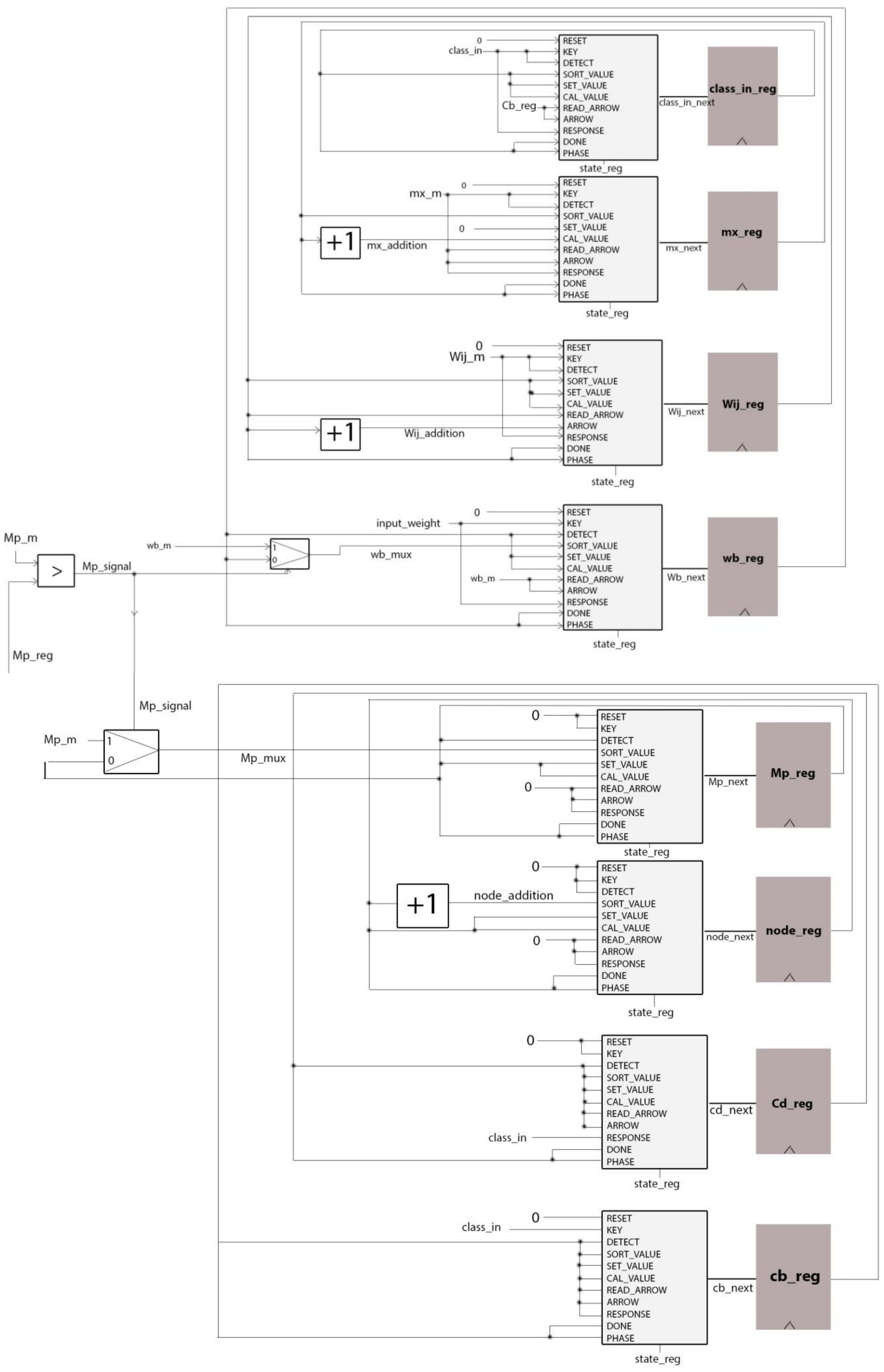
```
        -- control path: state register
        process(clk, reset, phase)
        begin
                if (phase = '0')then
                        state_reg <= phase_state;
                elsif (reset = '0') then
                        state_reg <= done_state;
                elsif (clk'event and clk = '1') then
                        state_reg <= state_next;
                end if;
        end process;


        -- control path: next-state/output logic
        process(state_reg, start, class_bit_m, node_addition, bit_reg)          --Sensitive
list with parameters that needs to go in
        begin
                case state_reg is

                        when phase_state =>

                                state_next <= done_state;

                        when reset_state =>

                                state_next <= done_state;
```

```
when idle =>

        if(start = '1')then
                state_next <= detect;
        else
                state_next <= idle;
        end if;

when detect =>

        if(class_bit_m = '1')then
                state_next <= sort_values;
        else
                state_next <= set_values;
        end if;

when sort_values =>

        if(node_addition < FIFTEEN)then
                state_next <= sort_values;
        else
                state_next <= cal_values;
        end if;

when set_values =>

        if(bit_next = '1')then
                state_next <= done_state;
        else
                state_next <= read_arrow;
        end if;

when cal_values =>

        if(bit_next = '1')then
                state_next <= done_state;
        else
                state_next <= read_arrow;
        end if;

when done_state =>

        if(bit_next = '1')then
                state_next <= response;
        else
                state_next <= idle;
        end if;

when response =>
```

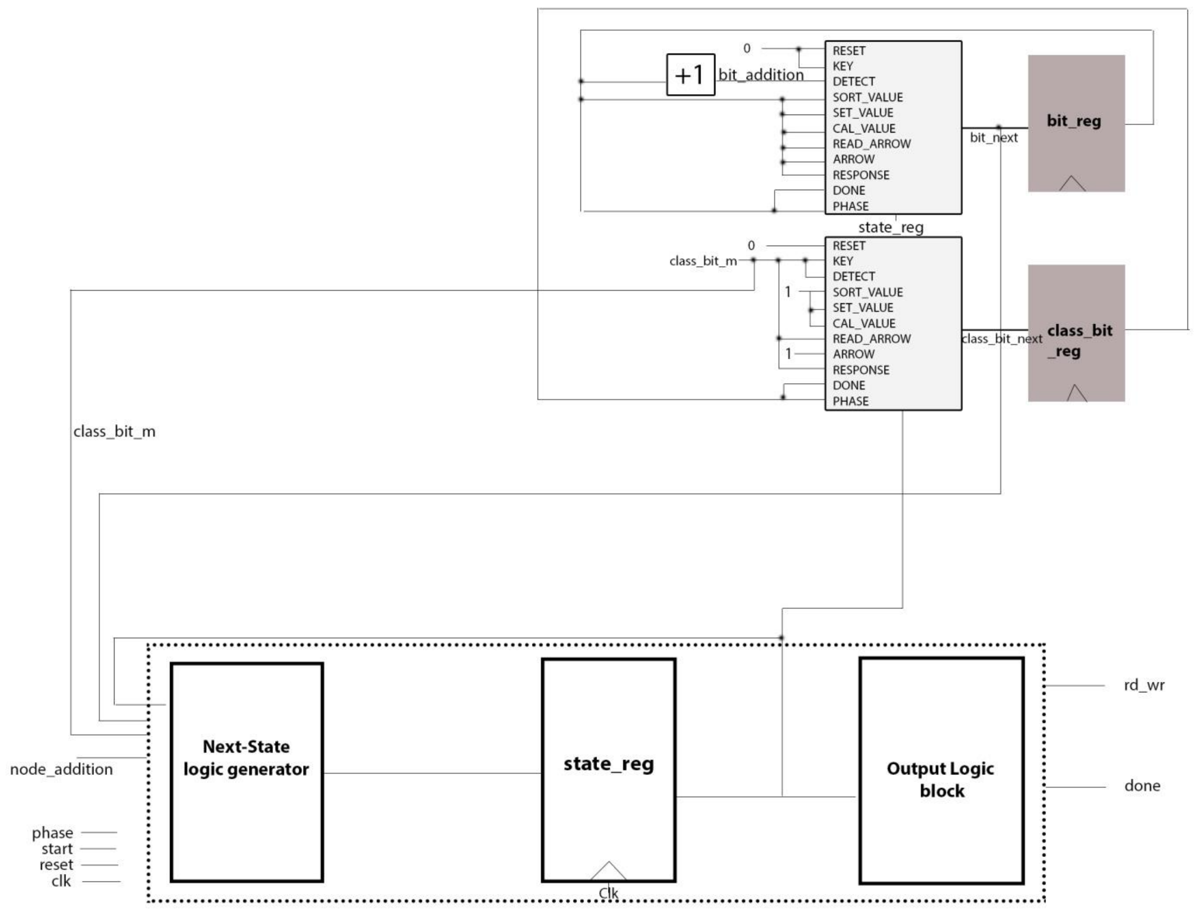
```
                    if(start = '1')then
                              state_next <= detect;
                    else
                              state_next <= response;
                    end if;

          when read_arrow =>

                    state_next <= arrow;

          when arrow =>

                    state_next <= done_state;
end case;
```

end process;

```
-- control path: output logic

rd_wr <= '1' when (state_reg = set_values) or (state_reg = cal_values) or (state_reg =
arrow) else '0'; -- 1 is write and 0 is read
done <= '1' when (state_reg = done_state) else '0';
```



0
class_in
Cb_reg
RESET
KEY
DETECT
SORT_VALUE
SET_VALUE
CAL_VALUE
READ_ARROW
ARROW
RESPONSE
DONE
PHASE
state_reg
class_in_next
class_in_reg
mx_m
+1
mx_addition
mx_next
mx_reg
Wij_m
Wij_addition
Wij_next
Wij_reg
input_weight
wb_m
wb_mux
Wb_next
wb_reg
Mp_m
Mp_signal
Mp_reg
Mp_mux
Mp_next
Mp_reg
node_addition
node_next
node_reg
cd_next
Cd_reg
cb_next
cb_reg



A generic adder component for adding

```
 mx_adder: entity work.generic_adder
   generic map (
               bits => NODES
   )
   port map (
     A  => mx_reg,
     B  => ONE,
     CI => ZERO,
     O  => mx_addition,
     CO => CO
   );
```

This component will just add the value to mx_reg by one, which we receive from the associative memory from the previous state. The addition is done whenever we access the same class again to represent the associative index. It gives the output **mx_addition.**

```
 wij_adder: entity work.generic_adder
   generic map (
            bits => NODES
   )
   port map (
     A  => wij_m,
     B  => ONE,
     CI => ZERO,
     O  => wij_addition,
     CO => CO
   );
```

This component will just increment the weight of an arrow, whenever the same relation is built again and again. It gives the output **Wij_addition**.

```
 node_adder: entity work.generic_adder
   generic map (
            bits => NODES
   )
   port map (
     A  => node_reg,
     B  => ONE,
     CI => ZERO,
     O  => node_addition,
     CO => CO
   );
```

This component will be incremented by one at every clock cycle. The **node_addition** is sent to the associative memory as an index to read values of each node.

```
 bit_adder: entity work.BIT_ADDER
      port map( a => bit_reg,
                b => '1',
                cin => '0',
                sum => bit_addition,
                cout => CO);
```

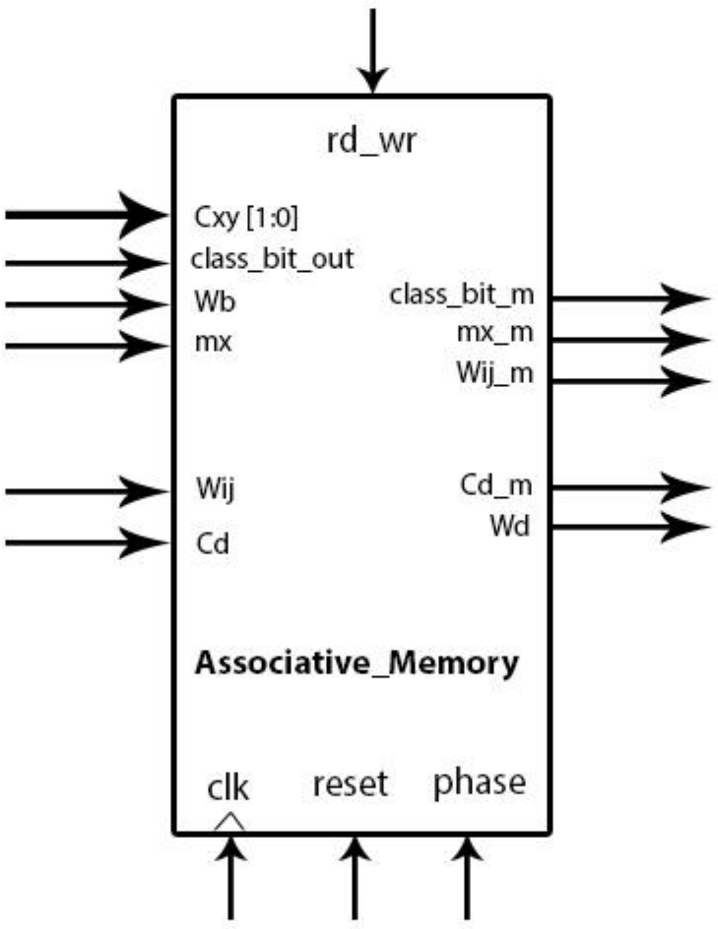
bit_adder component is a one bit adder component. It adds by one on every detect state. This bit is used to distinguish the between method of key and response process.

```
Mp_signal <= '1' when Mp_m > Mp_reg else '0';
```

This signal is used to find out the highest rank associative pattern holding node, which is used to tune the' Wb' value of an associative node in an associative memory. Below are the components used to realize the above statement.

```
 mux_mp:entity work.mux_2to1_Mp
   port map( SEL => Mp_signal,
          A   => Mp_m,
          B   => Mp_reg,
          X   => Mp_mux);

 mux_wb:entity work.mux_2to1_Wb
   port map( SEL => Mp_signal,
          A   => wb_m,
          B   => wb_reg,
          X   => wb_mux );
```

2.6 Associative Memory:



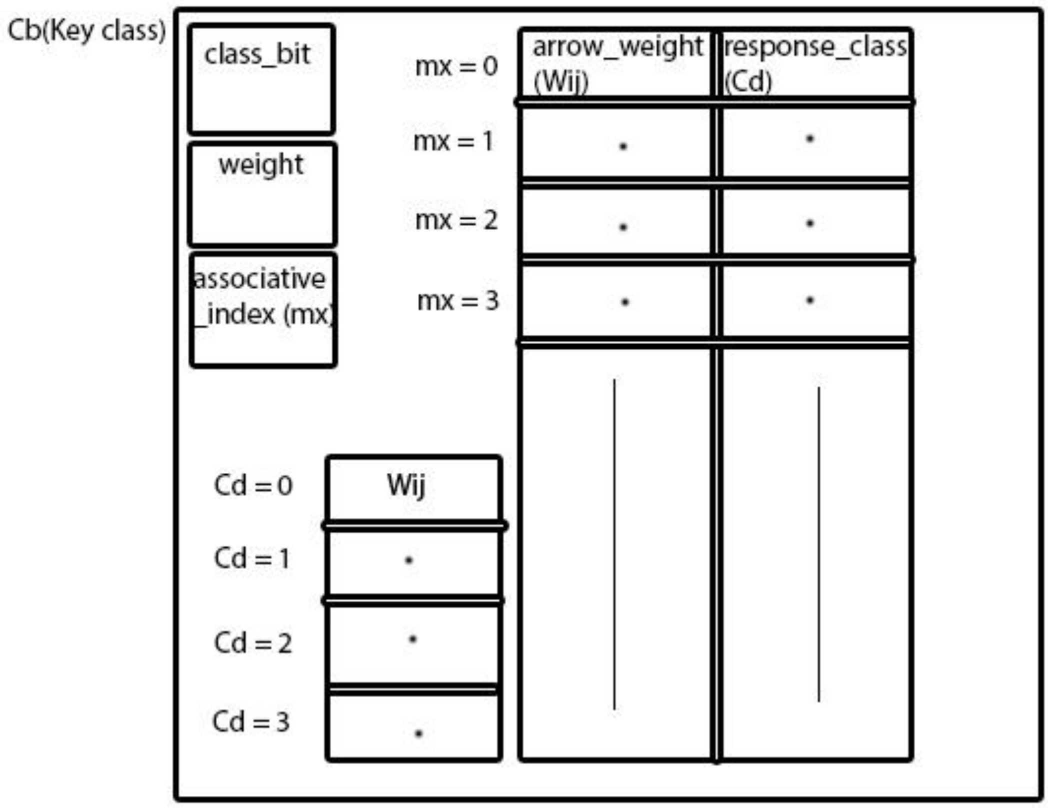
INPUTS:

i. Cxy: The class label output which works as index to associative memory
ii. Class_bit_out: It is one bit value which sets to indicate the presence of the class in associative memory as a node.
iii. Wb: Input weight vector that writes into an associative memory
iv. mx: The associative index that writes into an associative memory
v. Wij: The arrow weight between i and j node that writes into associative memory
vi. Cd: The reponse class writes into the associative memory.
vii. reset: Active low reset signal will clear the data of associative memory
viii. phase: When phase is ‘1’, associative memory allows the users to write data, but when phase is ‘0’ it doesn’t allow user to write the data.
ix. rd_wr = One bit output which sets ‘0’ for read and ‘1’ for write.

OUTPUTS:

i. class_bit_m: One bit indicator tells the presence of a class in associative memory
ii. mx_m: The highest associative index value
iii. Wij_m: The weight of an arrow relation between i(key) and j(response) inputs
iv. Cd_m: The response class that was mapped to the key class
v. Wd: The input vector weight.

The memory structure of associative memory:



The above shown diagram is a part of single class in an array of classes. It consists of one bit 'class_bit', vector weight storage location called 'weight', no.of associative index count 'mx' and two arrays of same size with index as 'mx'. Each associative index will have an arrow weight and an response class stored according ly.

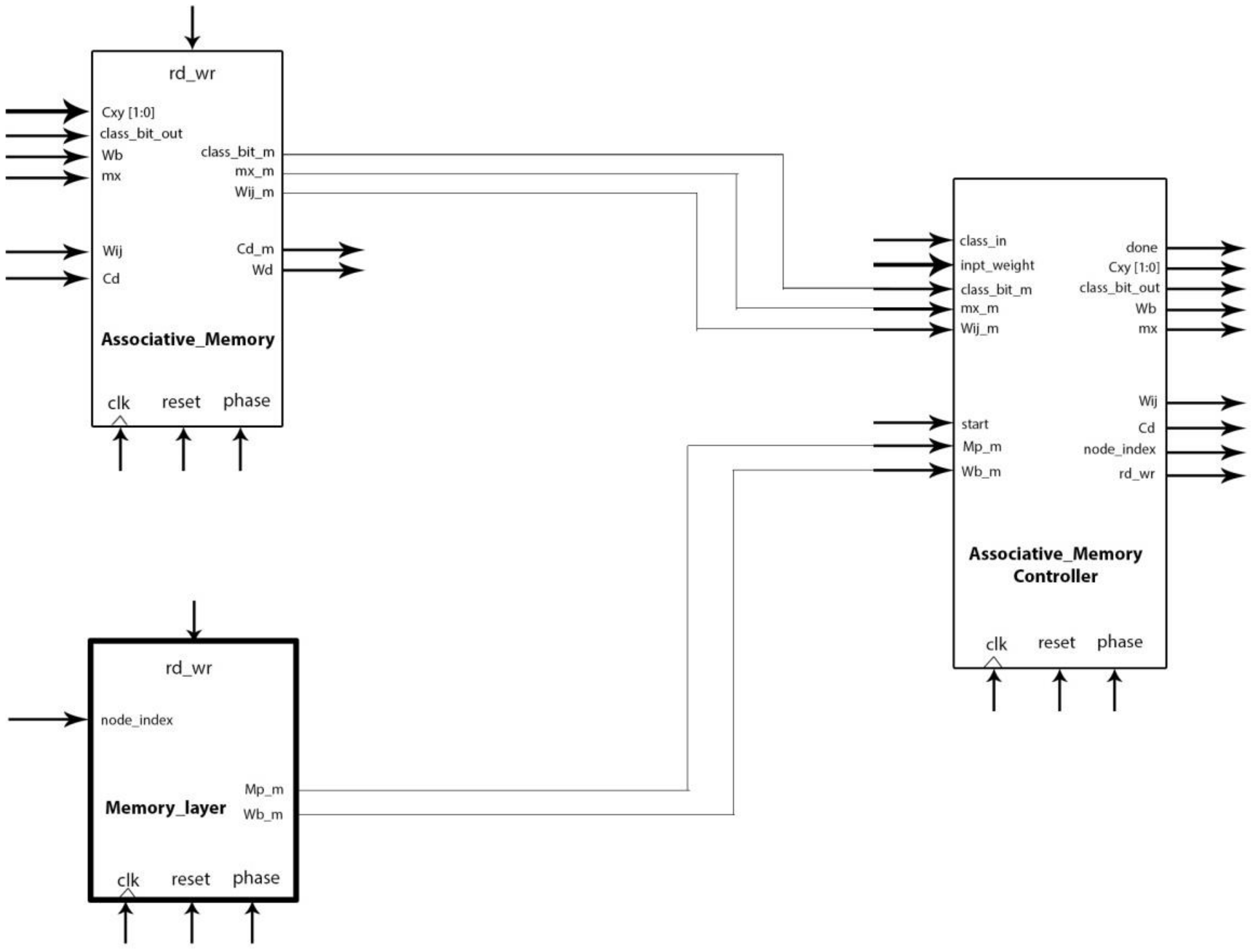
There is another table to the left bottom corner which is used to make a track of highest arrow weight for relation between i and j node arrow weights. They are only allowed to write and read during learning phase.

When Cxy, 'mx', wij, Cd comes as input from associative memory controller into associative memory, it overwrites the associative_index(mx), then it writes the Wij value into the bottom left corner table by making Cd as the index to it. At the same time Wij and Cd are written into the array on right side by making 'mx' as index to the table.

➔ associative_memory.vhd module is in Appendix

2.7 How to Test:

Inorder to test the system we need to build the environment as shown below.

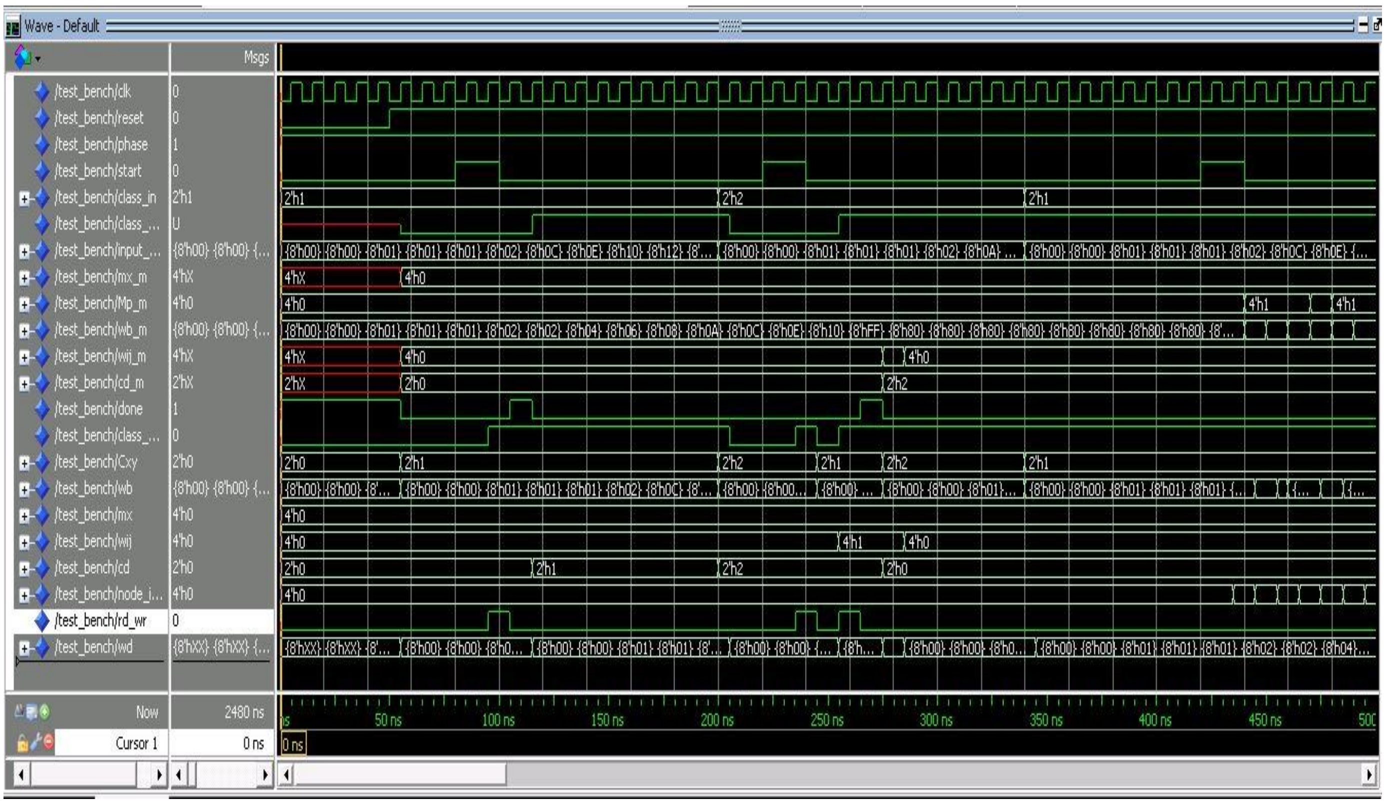


The modules that are present to include them in environment are Associate memory and Associate memory controller. But the memory layer which needs to be builded, which will just mimics as the actual memory layer in order to test the Associative memory controller and Associative memory. Below is the VHDL implementation of Memory_layer stub. The memory layer stub consist of database of 64 input weight vectors each of 0 to 255 resolution and an array of Mp values i.e., for all nodes. Each input weight vectors are of 5 Euclidian distance between two consecutive weight vectors. Below is the code for memory_layer stub and following is the test bench code.

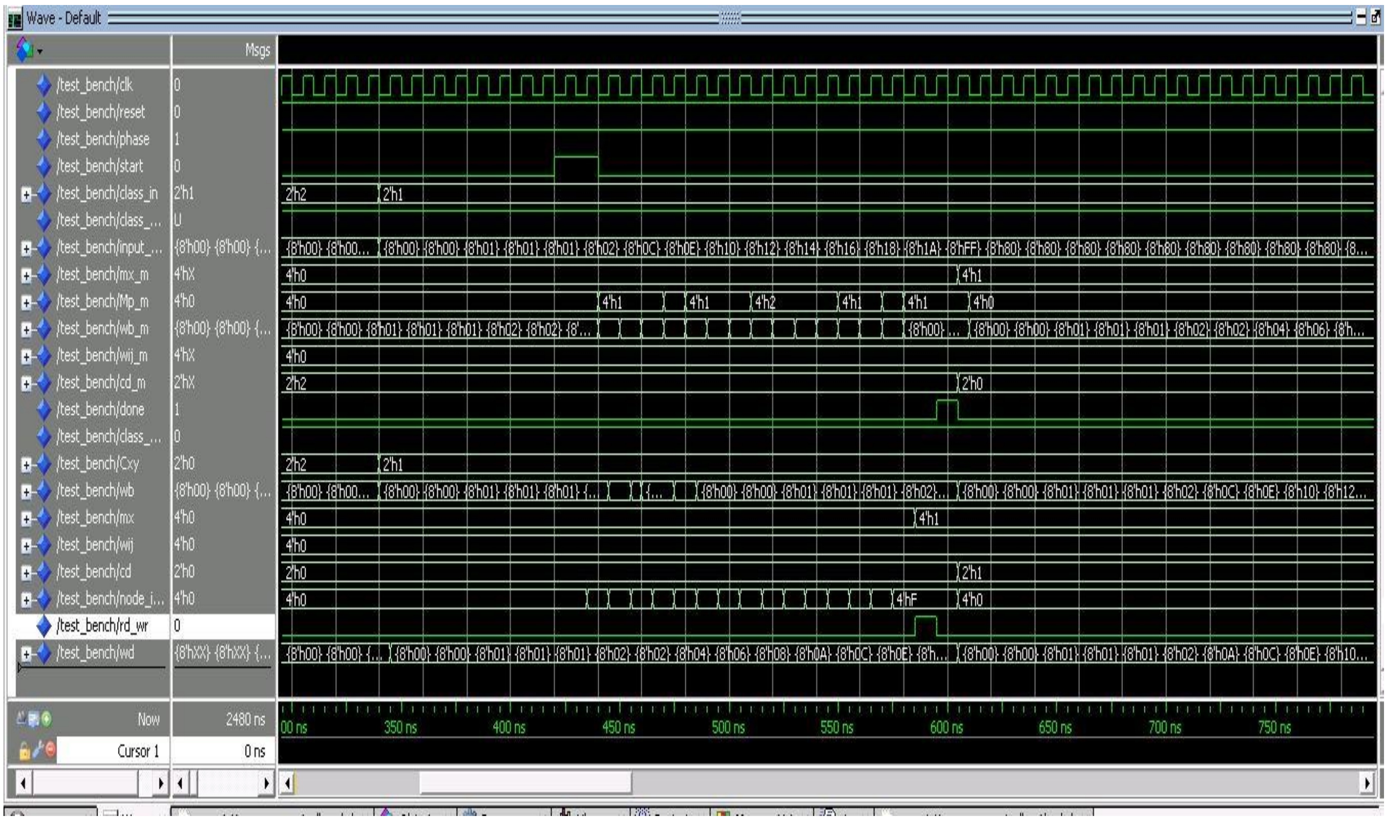
➔ Test bench code is in Appendix

2.8 Simulation Results:

In test Bench where both the inputs for class 1 and 2 are given (key and response) when no node in associative memory for a class has been initialised.

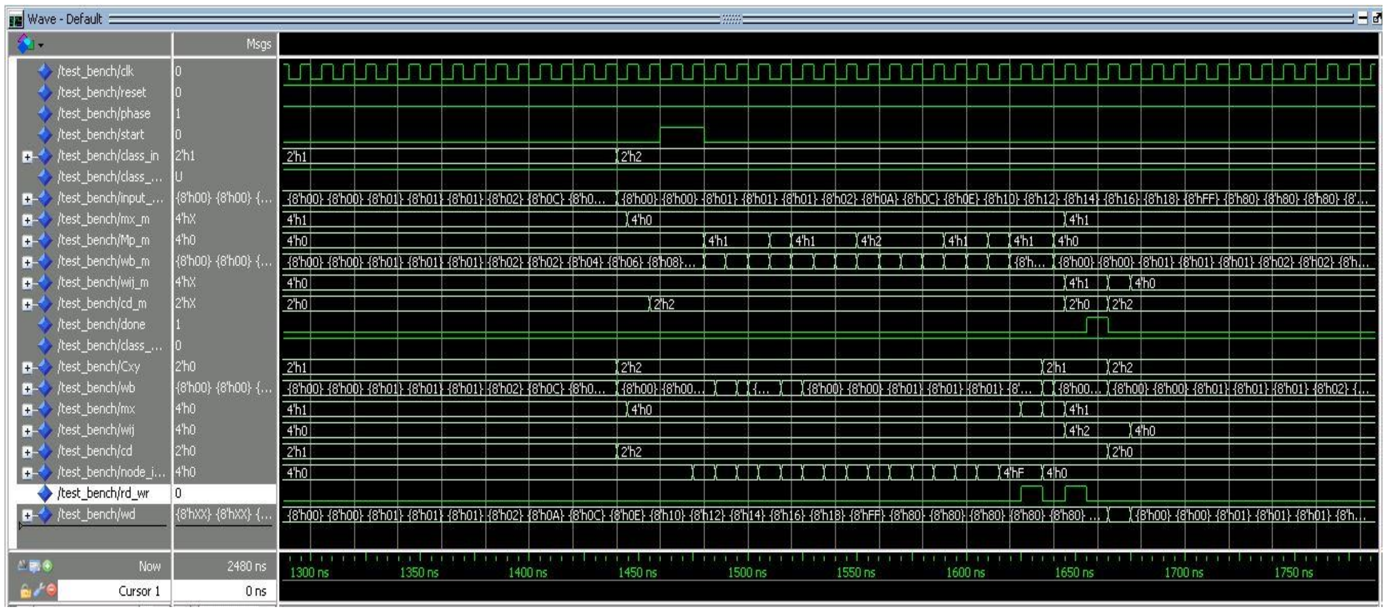


When class 1 is sent again after it was initialised in previous stimulus as key



```
Wave - Default
                         Msgs
/test_bench/clk          0
/test_bench/reset        0
/test_bench/phase        1
/test_bench/start        0
/test_bench/class_in     2'h1
/test_bench/class_...    U
/test_bench/input_...    {8'h00} {8'h00} {...
/test_bench/mx_m         4'hX
/test_bench/Mp_m         4'h0
/test_bench/wb_m         {8'h00} {8'h00} {...
/test_bench/wij_m        4'hX
/test_bench/cd_m         2'hX
/test_bench/done         1
/test_bench/class_...    0
/test_bench/Cxy          2'h0
/test_bench/wb           {8'h00} {8'h00} {...
/test_bench/mx           4'h0
/test_bench/wij          4'h0
/test_bench/cd           2'h0
/test_bench/node_i...    4'h0
/test_bench/rd_wr        0
/test_bench/wd           {8'hXX} {8'hXX} {...

Now       2480 ns
Cursor 1  0 ns
```

When class 2 is sent again after it was initialised in previous stimulus as response



```
Wave - Default
                         Msgs
/test_bench/clk          0
/test_bench/reset        0
/test_bench/phase        1
/test_bench/start        0
/test_bench/class_in     2'h1
/test_bench/class_...    U
/test_bench/input_...    {8'h00} {8'h00} {...
/test_bench/mx_m         4'hX
/test_bench/Mp_m         4'h0
/test_bench/wb_m         {8'h00} {8'h00} {...
/test_bench/wij_m        4'hX
/test_bench/cd_m         2'hX
/test_bench/done         1
/test_bench/class_...    0
/test_bench/Cxy          2'h0
/test_bench/wb           {8'h00} {8'h00} {...
/test_bench/mx           4'h0
/test_bench/wij          4'h0
/test_bench/cd           2'h0
/test_bench/node_i...    4'h0
/test_bench/rd_wr        0
/test_bench/wd           {8'hXX} {8'hXX} {...

Now       2480 ns
Cursor 1  0 ns
```

Chapter 3

Algorithm 4: Recall and associate

3.1 Introduction:

The paper General associative memory based on self-organizing incremental neural network, is a network consisting of three layers: an input layer, a memory layer, and an associative layer. The input layer accepts key vectors, response vectors, and the associative relationships between these vectors. The memory layer stores the input vectors incrementally to corresponding classes. The associative layer builds associative relationships between classes. The GAM can store and recall binary or non-binary information, learn key vectors and response vectors incrementally, realize many-to-many associations with no predefined conditions, store and recall both static and temporal sequence information, and recall information from incomplete or noise- polluted inputs.

The Algorithms 1, 2 discuss about the GAM learning algorithms (learning of memory layer and learning of associative layer). **Algorithm 4 explains the recall algorithm for auto-associative tasks**, and subsequently discuss the associating algorithm of hetero- associative tasks; the hetero-associative tasks include one-to-one, one-to-many, many-to-one, and many-to-many associations. We also describe the recall algorithm of the temporal sequence.

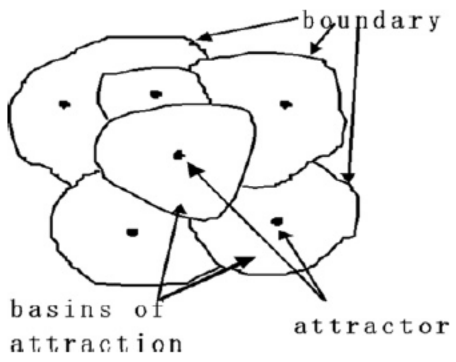


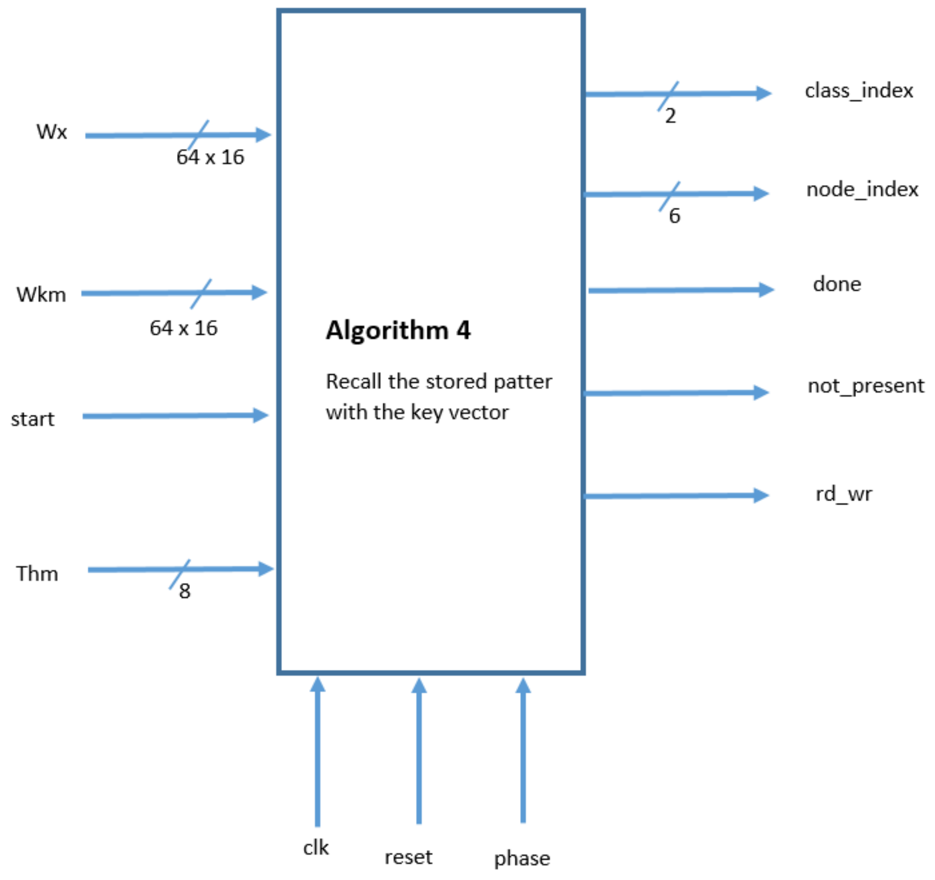
Fig. 5. Every attractor has an attraction basin. If the input pattern (key vector) is located in an attraction basin, the corresponding attractor is the associated result. If the input key vector lies outside all attraction basins, the key vector fails to recall the memorized pattern.

3.2 Steps for Algorithm 4:

1. Assume there are n nodes in the memory layer, input a key vector x.
2. **for i = 1, 2, 3…..n nodes, do**
   Calculate the Euclidean distance between the key vector x and all the nodes.
3. **end for**

4. Compare the Euclidian distance calculated in the above step with the similarity Threshold Tk(which is the centre of the radius of the region.)
5. For each class Calculate the maximum number of nodes, for which the Euclidian distance is less than the Tk.
6. **If** there is no nodes whose Euclidian distance is less then Tk **then** output message: the input vector failed to recall the memorized pattern.
7. **else**
8. Output the Class of the node k as the class of x.
9. **end if**

3.2 Block Diagram of Algorithm 4:



Explanation:

From the figure,

Inputs:

1. clk: The clock for the system.
2. reset: One bit input which reset the controller at active low signal.
3. phase: One bit input which set's the phase of an input ('1' = learning, '0' = recall).
4. start: one bit input, which starts the process of calculating of Euclidian distance and stuff.
5. Wx = Input key vector.
6. Wkm = Node present in memory layer.
7. Thm = Similarity Threshold Tk.

Outputs:

1. Class_index: class index to which the input key vector belong.
2. node_index: node_index acts like a counter to select a node from class.
3. done: one bit output flag, represents the recalling pattern has been found.
4. not_present : one bit output flag represents that the recalling pattern not found in any of the classes.
5. rd_wr: one bit output which sets '0' for read and '1' for write.

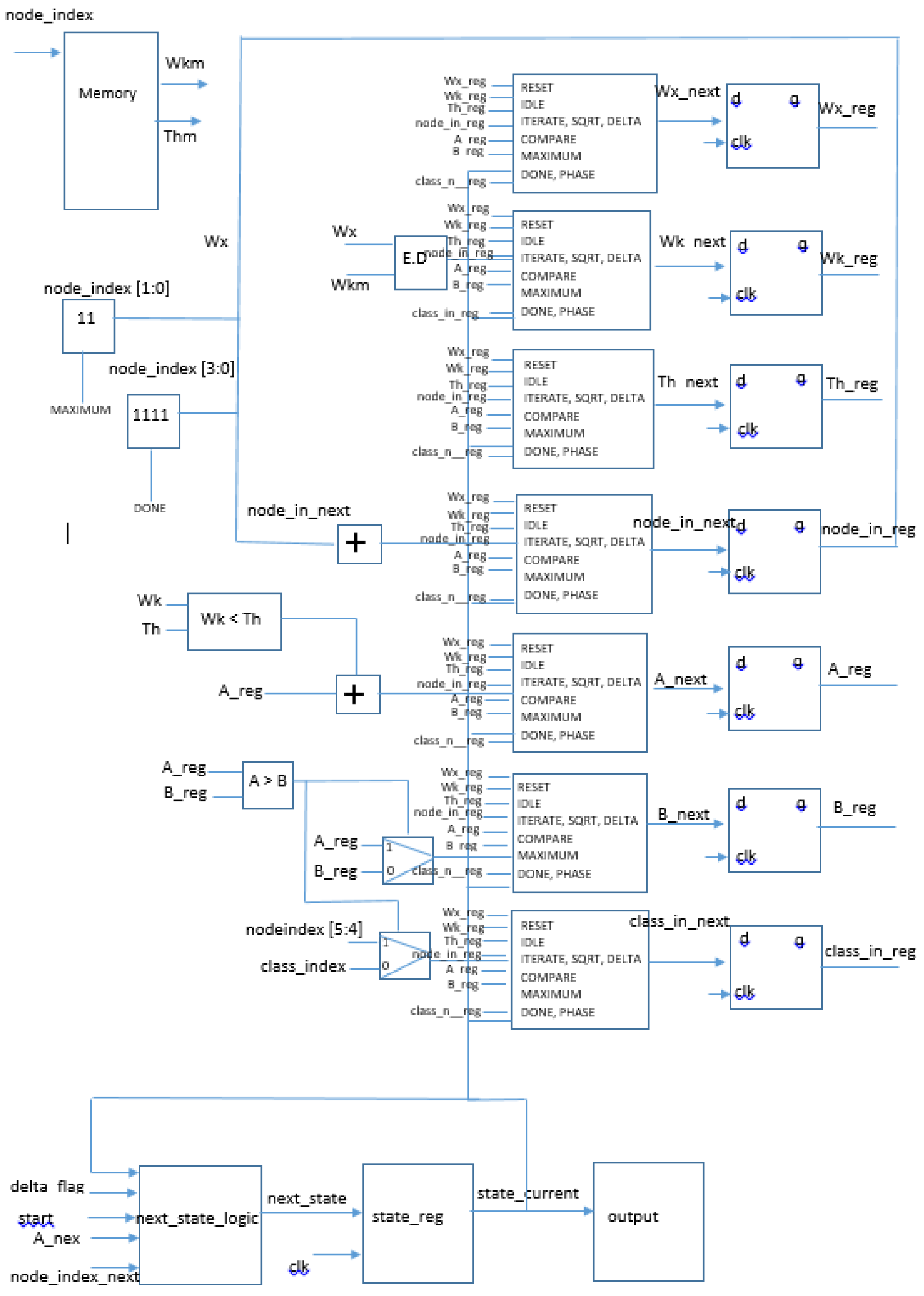
<u>Control & Data Path:</u>

Following is the code for the control unit,

```
            -- control path: state register

        process(clk, reset, phase)
        begin
         if(phase ='1') then
                state_current <= PHASE_STATE;
        elsif (reset = '0') then
                state_current <= RST_STATE;
        elsif (clk'event and clk = '1') then
                state_current <= state_next;
        end if;
        end process;

        -- control path: next-state/output logic

process(state_current, start, delta_flag, node_index_next,A_next)
        variable  count : integer := 0;
        begin
                case state_current is
                        when RST_STATE =>
                                state_next <= DONE_ST;

                        when PHASE_STATE =>
                                state_next <= DONE_ST;
                        when IDLE =>
                                if(start = '1')then
                                        state_next <= ITERATE_FOR_SUM;
                                else
```

```
                                        state_next <= IDLE;
                                end if;

                        when ITERATE_FOR_SUM =>
                        -- calculate Euclidean distance and move to compare state.
                                        state_next <= DELTA;

                        when DELTA => --newly added state
                                if(delta_flag = '1')then
                                        state_next <= SQUARE_ROOT;
                                elsif(delta_flag = '0')then
                                        state_next <= DELTA;
                                end if;


                        when SQUARE_ROOT=>
                        state_next <= COMPARE;

                        when COMPARE =>
                        -- Compare the euclidean distance with the Threshold.
                        -- If It is less than Threshold Increment A_next. and move to maximum state.

                        state_next <= MAXIMUM;

                        when MAXIMUM =>
                        -- Compare A and B which ever is greater, that one goes to B_next.
                        --Move to Iterate state next.
                                if(node_index_next < full)then
                                                state_next <= ITERATE_FOR_SUM;
                                else
                                        if(A_next = a_const) then
                                                state_next <= ERROR_ST;
                                        else
                                                state_next <= DONE_ST;
                                        end if;
                                end if;

                        when DONE_ST =>
                        -- Assert Done flag and move to IDLE state.
                                state_next <= IDLE;

                        when ERROR_ST =>
                                state_next <= IDLE;

                end case;
end process;

-- control path: output logic
done<= '1' when (state_current = DONE_ST) else '0';
not_present <= '1' when (state_current = ERROR_ST) else '0';
```

3.3 Complete FSMD Diagram:



3.4 Data path:

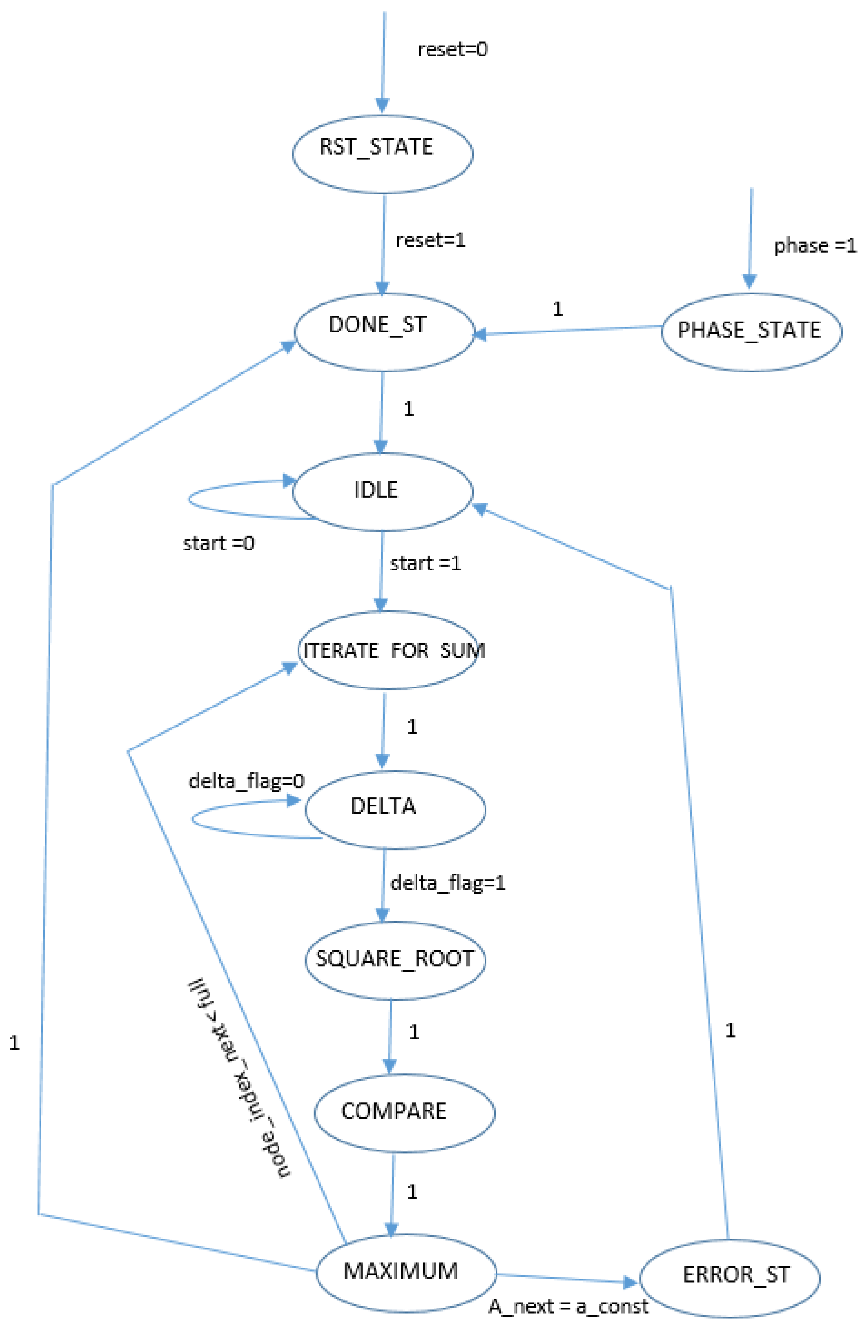
The data path consists of 7 registers, 2 multiplexers, 2 comparators, 2 adders, Euclidean distance block and memory block. As data enters the data path, we try to calculate the Euclidean distance between the input key vector and the node present in the memory. This is done by the E.D block present in the above diagram. E.D is nothing but Euclidean distance calculation block.

3.5 Control path:

The control logic does nine things:

- RST_STATE: The machine will be entering this state by default, unless it is set to 1.
- DONE_STATE: This state is to assert the done signal whenever there is a key vector belongs to a particular class.
- IDLE: In this state it will wait for the start signal to be asserted high. It keep on waiting in this state.
- ITERATE_FOR_SUM: In this state it calculates the Euclidean distance.
- DELTA: In this state, we calculate the square root of Euclidean distance.
- SQUARE_ROOT: We assign the square root value to reg in this state for the delay.
- COMPARE: This state compares, the E.D and increments A_reg and B_reg.
- MAXIMUM: Purpose of this state is to, move to ITERATE_FOR_SUM if there still nodes present in the memory to make calculations. Move to error state if there is no match found.
- ERROR_ST: The purpose of this state is to, indicate there is no matching for our key input. And asserts not_present signal.

3.6 Finite State Machine Diagram for Algorithm 4



3.7 Sub-Circuits: We have used below components in the main FSMD.

Euclidean distance Calculation Process: In the process of calculating the Euclidean distance. We do subtraction of weight of key vector and weight of the node in the memory. And then do the square subtracted value, we do this for all the weights associated with that node. After that we do the square of the subtracted values and add them together and perform a square root.

generic_adder: This sub component is a generic adder which is used for doing the addition for incrementing the A_reg, and then it is also used to incrementing the temporary register delta while doing the square root calculation.

Subtraction component: This component is used in the process of calculating the Euclidean distance. We do subtraction of weight of key vector and weight of the node in the memory.

Multiplication Component: This component is used to do square of the subtracted weights, that we get after the subtraction component operation.

For doing the square root of the sum of square of subtracted values of the input key vector and weight of the nodes. We use below algorithm.

```
unsigned long sqrt(unsigned long a){
  unsigned long square = 1;
  unsigned long delta = 3;
  while(square <= a){
     square = square + delta;
     delta = delta + 2;
  }
     return (delta/2 - 1);
}
```

Below table shows the illustration of the above square root algorithm.

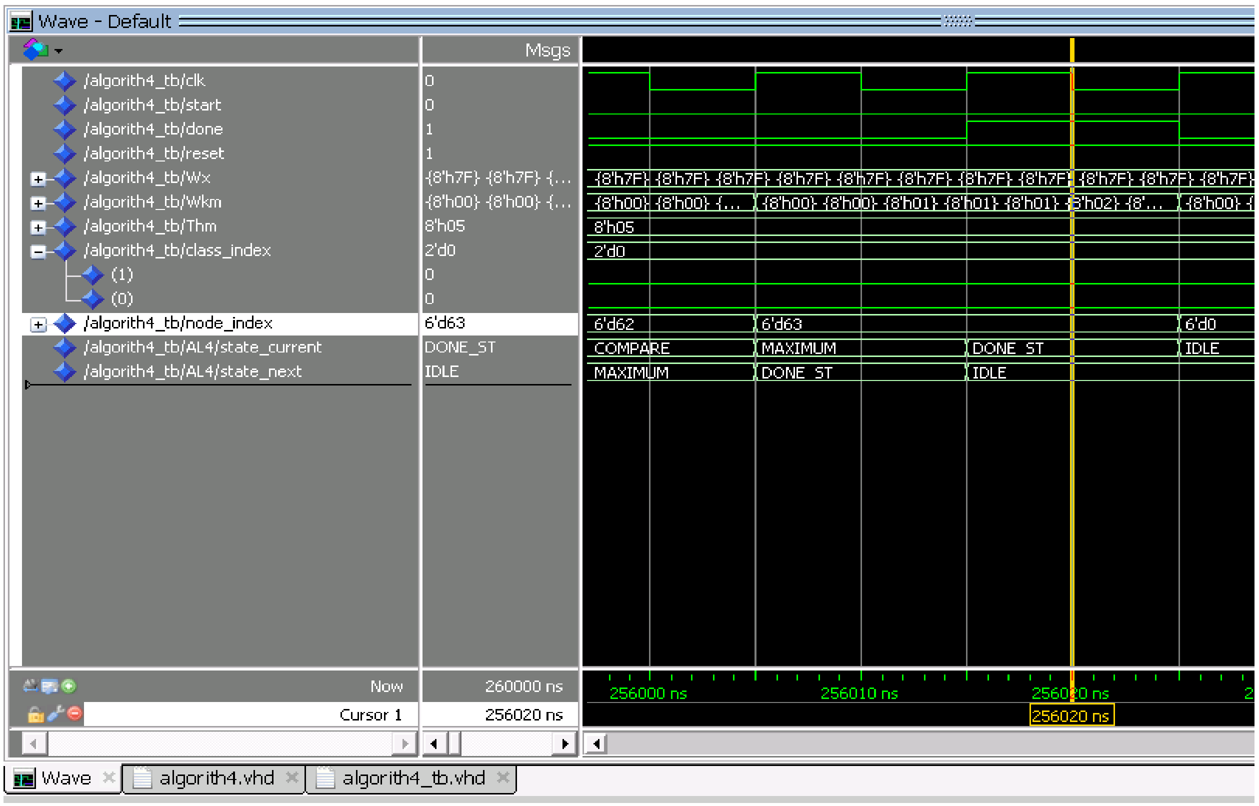
Illustrating the algorithm in Fig. 2.8

n	square = n^2	delta	delta/2-1
0	0		
1	1	3	
2	4	5	1
3	9	7	2
4	16	9	3
5	25	11	4
6	36	13	5
7	49	15	6
8	64	17	7
9	81	19	8
10	100	21	9
11	121	23	10
12	144	25	11
13	169	27	12
14	196	29	13
15	225	31	14
16	256	33	15
17	289		

➔ VHDL Code for Algorithm 4 and subcomponents is in Chapter- 3 of Appendix

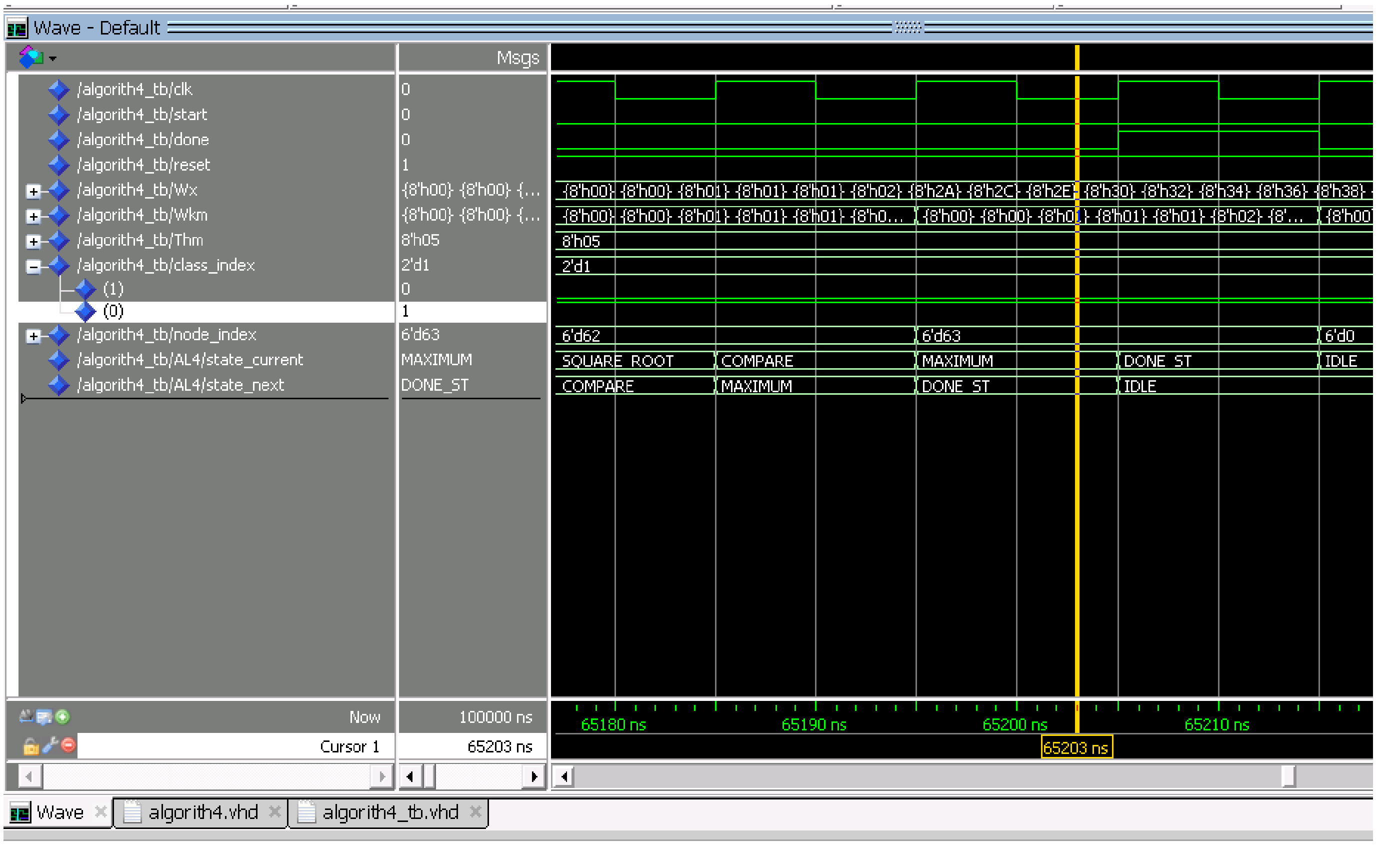
3.8 Simulation Results:

1. Input vector which belong to class 0 is given as input.



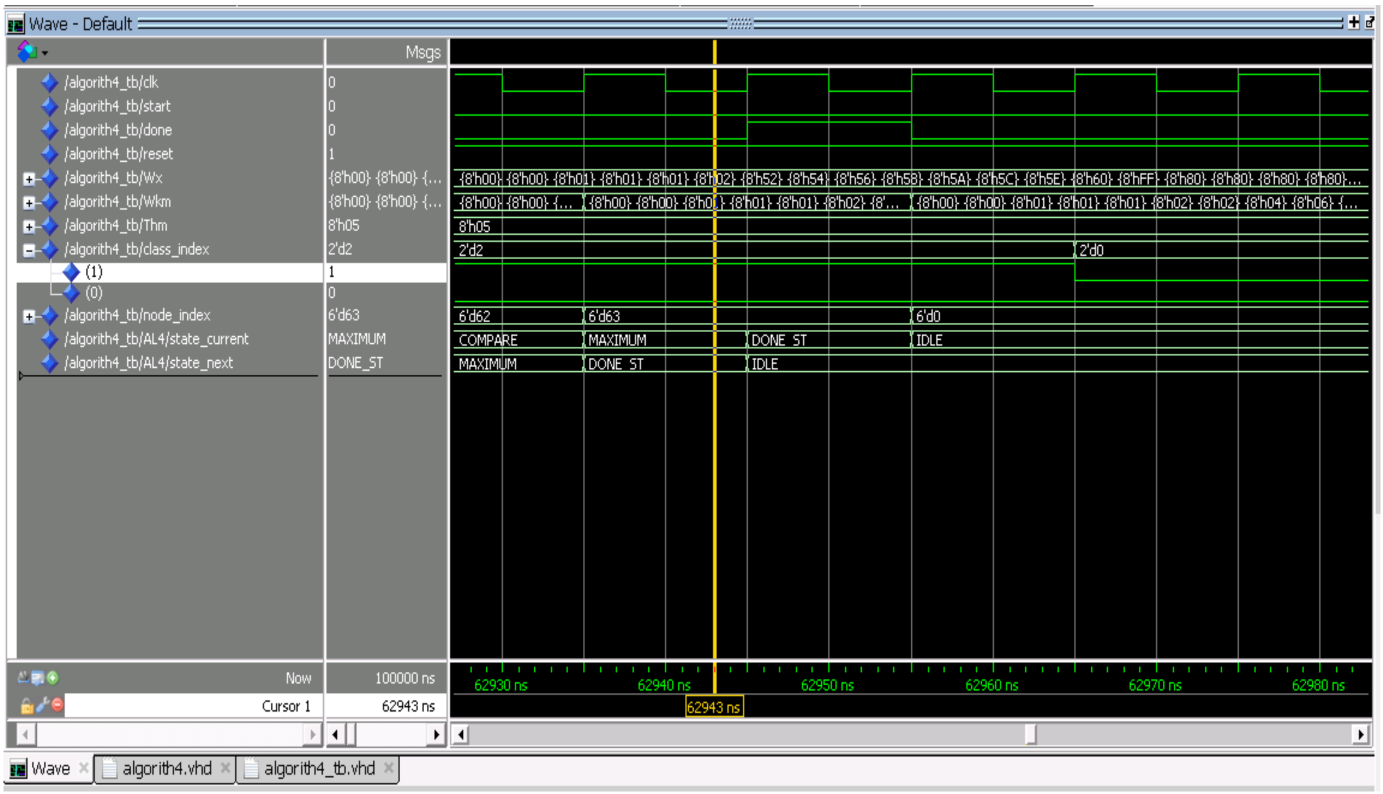
```
Wave - Default
                                    Msgs
/algorith4_tb/clk                   0
/algorith4_tb/start                 0
/algorith4_tb/done                  1
/algorith4_tb/reset                 1
/algorith4_tb/Wx                    {8'h7F} {8'h7F} {...
/algorith4_tb/Wkm                   {8'h00} {8'h00} {...
/algorith4_tb/Thm                   8'h05
/algorith4_tb/class_index           2'd0
    (1)                             0
    (0)                             0
/algorith4_tb/node_index            6'd63
/algorith4_tb/AL4/state_current     DONE_ST
/algorith4_tb/AL4/state_next        IDLE

Now        260000 ns
Cursor 1   256020 ns

Wx:          {8'h7F} {8'h7F} {8'h7F} {8'h7F} {8'h7F} {8'h7F} {8'h7F} {8'h7F} {8'h7F} {8'h7F} {8'h7F}
Wkm:         {8'h00} {8'h00} {... {8'h00} {8'h00} {8'h01} {8'h01} {8'h01} {8'h02} {8'... {8'h00} {
Thm:         8'h05
class_index: 2'd0
node_index:  6'd62 | 6'd63 | 6'd0
state_current: COMPARE | MAXIMUM | DONE_ST | IDLE
state_next:    MAXIMUM | DONE_ST | IDLE

256000 ns    256010 ns    256020 ns
256020 ns

Wave    algorith4.vhd    algorith4_tb.vhd
```

2. Input vector which belong to class 1 is given as input.



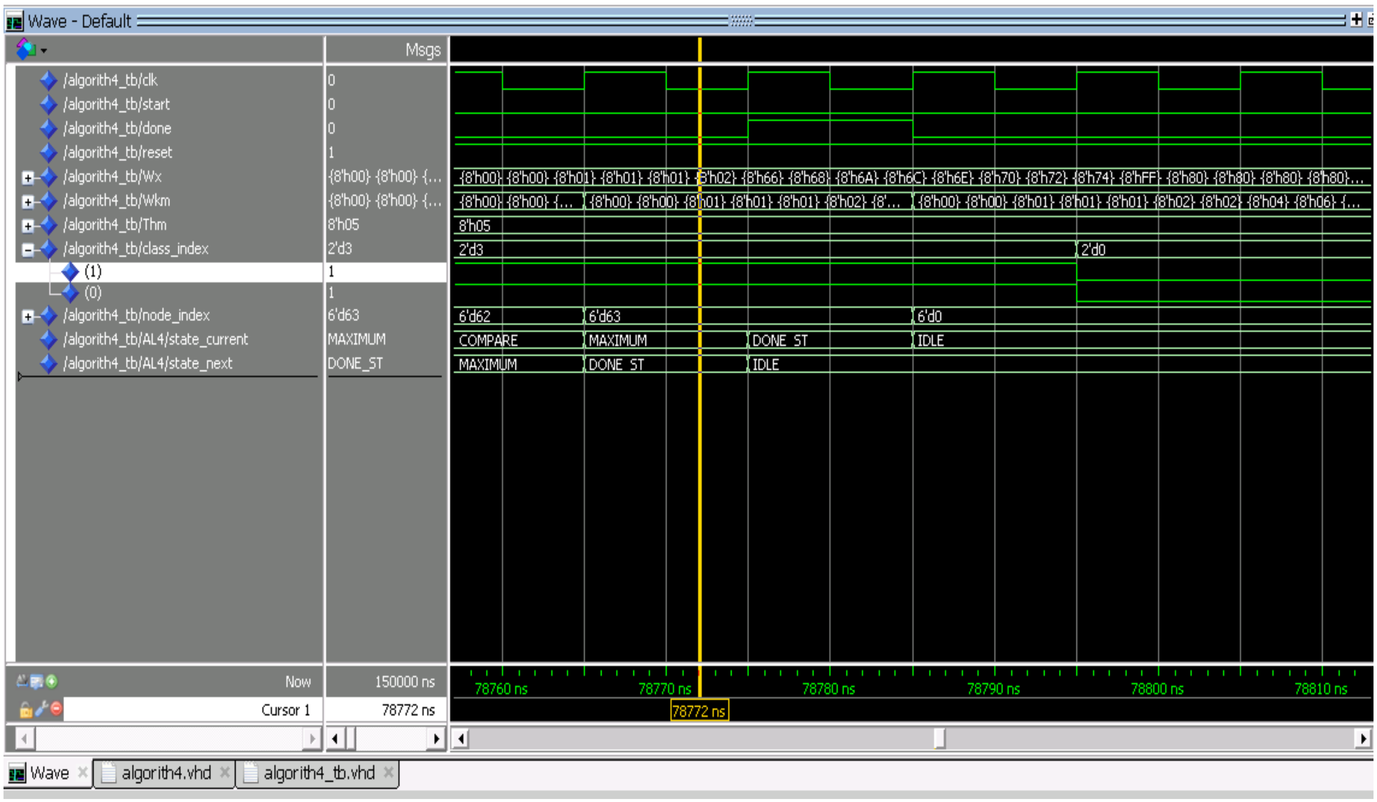
```
Wave - Default
                                    Msgs
/algorith4_tb/clk                   0
/algorith4_tb/start                 0
/algorith4_tb/done                  0
/algorith4_tb/reset                 1
/algorith4_tb/Wx                    {8'h00} {8'h00} {...
/algorith4_tb/Wkm                   {8'h00} {8'h00} {...
/algorith4_tb/Thm                   8'h05
/algorith4_tb/class_index           2'd1
    (1)                             0
    (0)                             1
/algorith4_tb/node_index            6'd63
/algorith4_tb/AL4/state_current     MAXIMUM
/algorith4_tb/AL4/state_next        DONE_ST

Now        100000 ns
Cursor 1   65203 ns

Wx:          {8'h00} {8'h00} {8'h01} {8'h01} {8'h01} {8'h02} {8'h2A} {8'h2C} {8'h2E} {8'h30} {8'h32} {8'h34} {8'h36} {8'h38}
Wkm:         {8'h00} {8'h00} {8'h01} {8'h01} {8'h01} {8'h0... {8'h00} {8'h00} {8'h0... {8'h01} {8'h01} {8'h02} {8'... {8'h00
Thm:         8'h05
class_index: 2'd1
node_index:  6'd62 | 6'd63 | 6'd0
state_current: SQUARE_ROOT | COMPARE | MAXIMUM | DONE_ST | IDLE
state_next:    COMPARE | MAXIMUM | DONE_ST | IDLE

65180 ns    65190 ns    65200 ns    65210 ns
65203 ns

Wave    algorith4.vhd    algorith4_tb.vhd
```

3. Input vector which belong to class 2 is given as input.



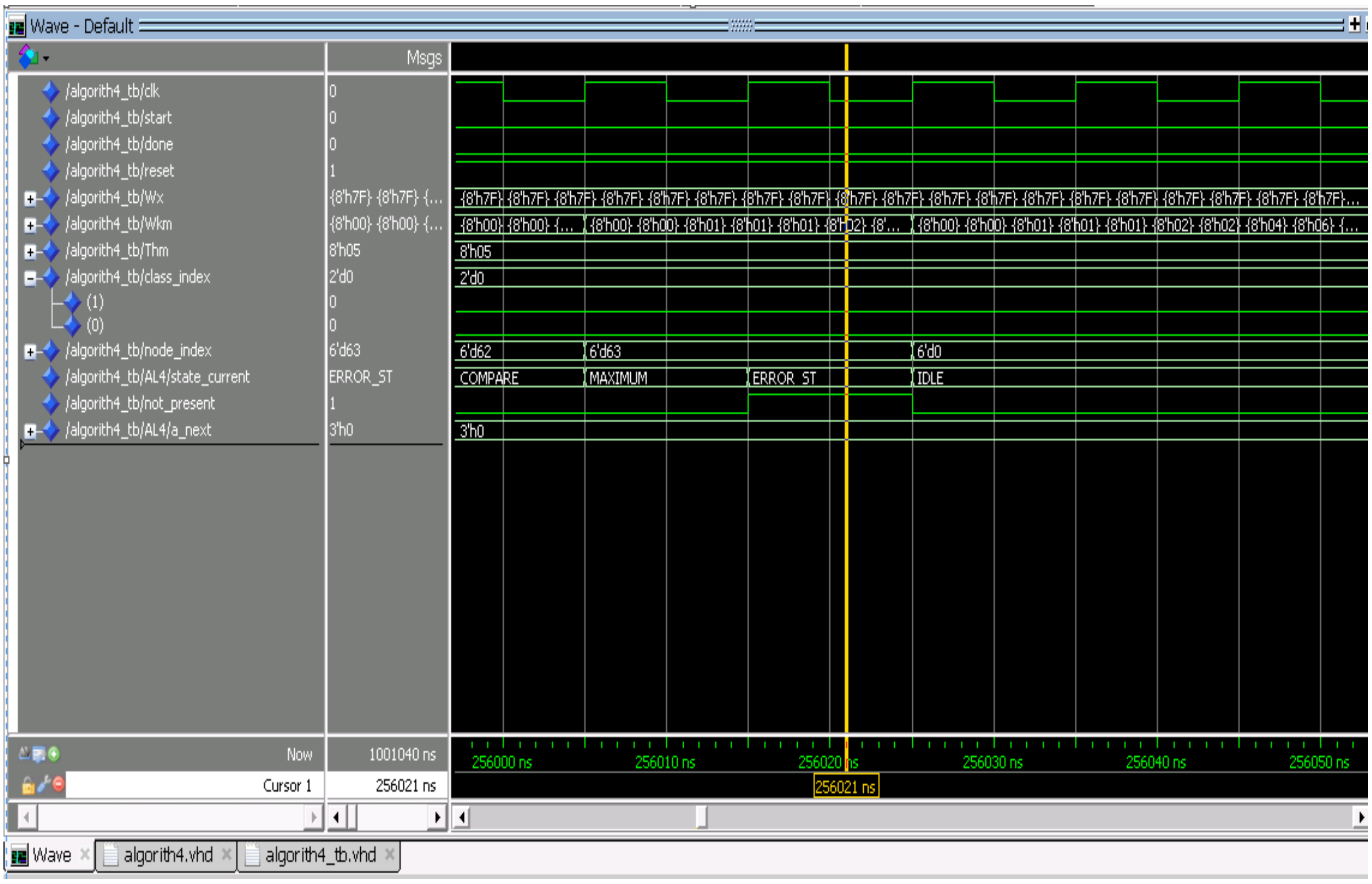
```
Wave - Default
                                 Msgs
/algorith4_tb/clk                0
/algorith4_tb/start              0
/algorith4_tb/done               0
/algorith4_tb/reset              1
/algorith4_tb/Wx                 {8'h00} {8'h00} {...
/algorith4_tb/Wkm                {8'h00} {8'h00} {...
/algorith4_tb/Thm                8'h05
/algorith4_tb/class_index        2'd2
    (1)                          1
    (0)                          0
/algorith4_tb/node_index         6'd63
/algorith4_tb/AL4/state_current  MAXIMUM
/algorith4_tb/AL4/state_next     DONE_ST

Wx:          {8'h00} {8'h00} {8'h01} {8'h01} {8'h01} {8'h02} {8'h52} {8'h54} {8'h56} {8'h58} {8'h5A} {8'h5C} {8'h5E} {8'h60} {8'hFF} {8'h80} {8'h80} {8'h80} {8'h80}...
Wkm:         {8'h00} {8'h00} {... {8'h00} {8'h00} {8'h01} {8'h01} {8'h01} {8'h02} {8'... {8'h00} {8'h00} {8'h01} {8'h01} {8'h01} {8'h02} {8'h02} {8'h04} {8'h06} {...
Thm:         8'h05
class_index: 2'd2 | 2'd0
node_index:  6'd62 | 6'd63 | 6'd0
state_current: COMPARE | MAXIMUM | DONE_ST | IDLE
state_next:    MAXIMUM | DONE_ST | IDLE

Now 100000 ns          62930 ns  62940 ns  62950 ns  62960 ns  62970 ns  62980 ns
Cursor 1 62943 ns      62943 ns

Wave | algorith4.vhd | algorith4_tb.vhd
```

4. Input vector which belong to class 3 is given as input.



```
Wave - Default
                                 Msgs
/algorith4_tb/clk                0
/algorith4_tb/start              0
/algorith4_tb/done               0
/algorith4_tb/reset              1
/algorith4_tb/Wx                 {8'h00} {8'h00} {...
/algorith4_tb/Wkm                {8'h00} {8'h00} {...
/algorith4_tb/Thm                8'h05
/algorith4_tb/class_index        2'd3
    (1)                          1
    (0)                          1
/algorith4_tb/node_index         6'd63
/algorith4_tb/AL4/state_current  MAXIMUM
/algorith4_tb/AL4/state_next     DONE_ST

Wx:          {8'h00} {8'h00} {8'h01} {8'h01} {8'h01} {8'h02} {8'h66} {8'h68} {8'h6A} {8'h6C} {8'h6E} {8'h70} {8'h72} {8'h74} {8'hFF} {8'h80} {8'h80} {8'h80} {8'h80}...
Wkm:         {8'h00} {8'h00} {... {8'h00} {8'h00} {8'h01} {8'h01} {8'h01} {8'h02} {8'... {8'h00} {8'h00} {8'h01} {8'h01} {8'h01} {8'h02} {8'h02} {8'h04} {8'h06} {...
Thm:         8'h05
class_index: 2'd3 | 2'd0
node_index:  6'd62 | 6'd63 | 6'd0
state_current: COMPARE | MAXIMUM | DONE_ST | IDLE
state_next:    MAXIMUM | DONE_ST | IDLE

Now 150000 ns          78760 ns  78770 ns  78780 ns  78790 ns  78800 ns  78810 ns
Cursor 1 78772 ns      78772 ns

Wave | algorith4.vhd | algorith4_tb.vhd
```

5. Input vector which doesn't belong to any class is given as input.



```
/algorith4_tb/clk               0
/algorith4_tb/start             0
/algorith4_tb/done              0
/algorith4_tb/reset             1
/algorith4_tb/Wx                {8'h7F} {8'h7F} {...
/algorith4_tb/Wkm               {8'h00} {8'h00} {...
/algorith4_tb/Thm               8'h05
/algorith4_tb/class_index       2'd0
    (1)                         0
    (0)                         0
/algorith4_tb/node_index        6'd63
/algorith4_tb/AL4/state_current ERROR_ST
/algorith4_tb/not_present       1
/algorith4_tb/AL4/a_next        3'h0

Now        1001040 ns
Cursor 1   256021 ns
```

```
Wx:          {8'h7F} {8'h7F} {8'h7F} {8'h7F} {8'h7F} {8'h7F} {8'h7F} {8'h7F} {8'h7F} {8'h7F} {8'h7F} {8'h7F} {8'h7F} {8'h7F} {8'h7F} {8'h7F} {8'h7F} {8'h7F} {8'h7F}...
Wkm:         {8'h00} {8'h00} {...  {8'h00} {8'h00} {8'h01} {8'h01} {8'h02} {8'...  {8'h00} {8'h00} {8'h01} {8'h01} {8'h02} {8'h02} {8'h04} {8'h06} {...
Thm:         8'h05
class_index: 2'd0
node_index:  6'd62   6'd63   6'd0
state:       COMPARE   MAXIMUM   ERROR_ST   IDLE
a_next:      3'h0
256000 ns   256010 ns   256020 ns   256030 ns   256040 ns   256050 ns
256021 ns
```

Chapter 4

4.1 <u>Introduction: Algorithm 5:</u>

Association in hetero association mode, during the learning phase the key vector and response vector are input to the memory, and the Learning algorithms make sure that both the pair keyvector→Responsevector pair is remembered through the connections between the key node and response node in the associative layer (with the help of values of Mbx on the nodes and the Wij the connection weight between the two nodes). During the recall phase (testing) the stored pattern (response vector: i.e. the output of this algorithm) is recalled for the class name Cx (for an input key vector) which is obtained from the algorithm 4.

So the algorithm 4 gives input as class Cx (corresponds to the input or key vector) to this module, here we find the associated node with class in the associative layer and then find the corresponding response classes in the memory layer based on the value of Mbx for the key node in the associative layer. After finding the response classes we sort those classes (the sorting is done in order to make sure the correct sequences are outputted for the input, if the input is expecting several outputs in order). The weight vectors in the associative layer which corresponds to the response class in memory layer are output from the memory as the output of the modules.

The notations used in this algorithm are already described in the algorithms 1,2 and 4.

This algorithm is implemented in traditional FSMD model, The Controller and Data path are explained below:

In the data path design all the components are connected to memory to a specific port there are no control signals sent to memory to read a specific value, i.e. for every memory read operation all outputs are outputted at the memory output.

It is assumed that there are 4 classes in total in memory layer.

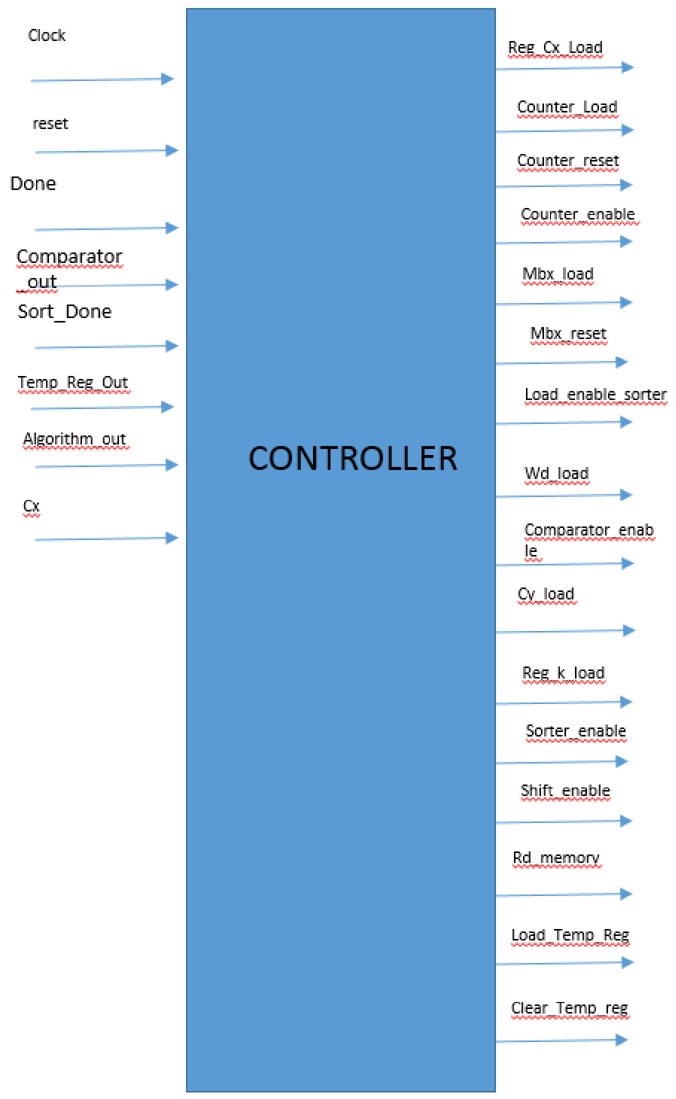
So the registers storing the class names are assumed to be 2 bits.

The weight vector in the associative layer is assumed to be 8 bits.

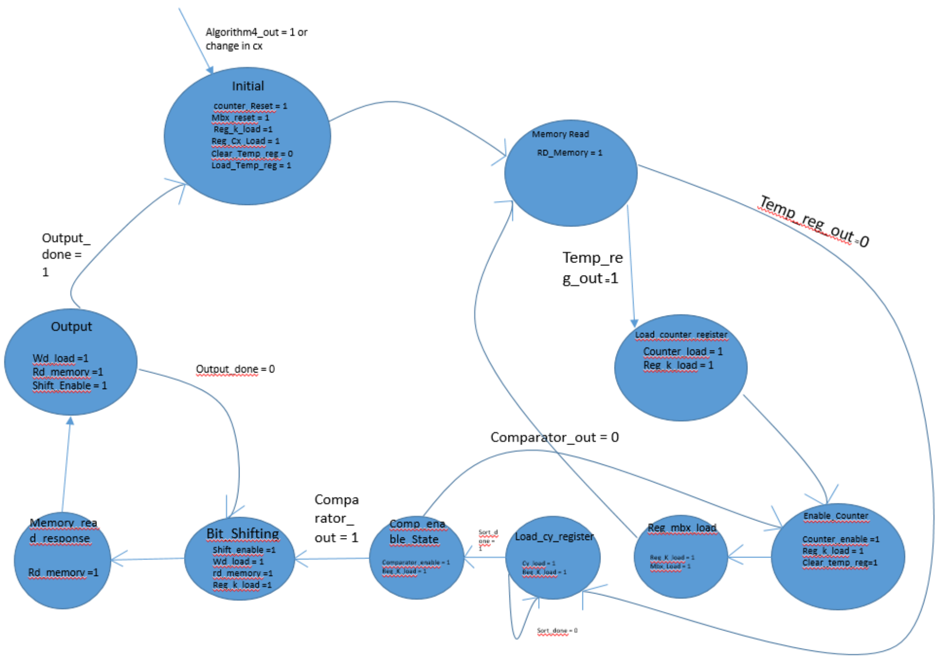
The connection weight between the two nodes in the associative layer is assumed to be 6 bits.

And the size of the sorter can be dynamically sorted I.e. it can sort n inputs.

4.2 CONTROLLER FOR THIS ALGORITHM :



<u>The finite state machine for this algorithm is:</u>

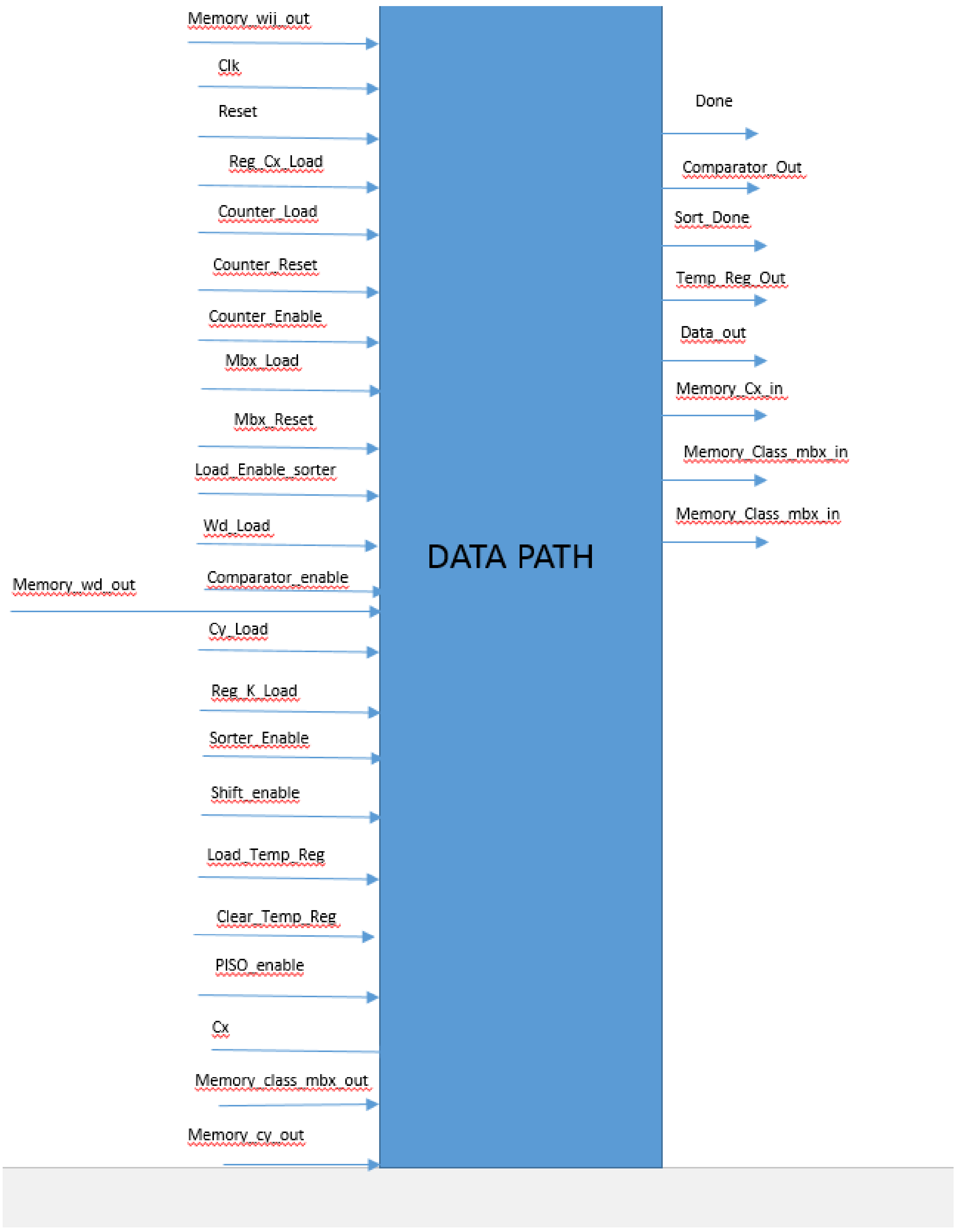


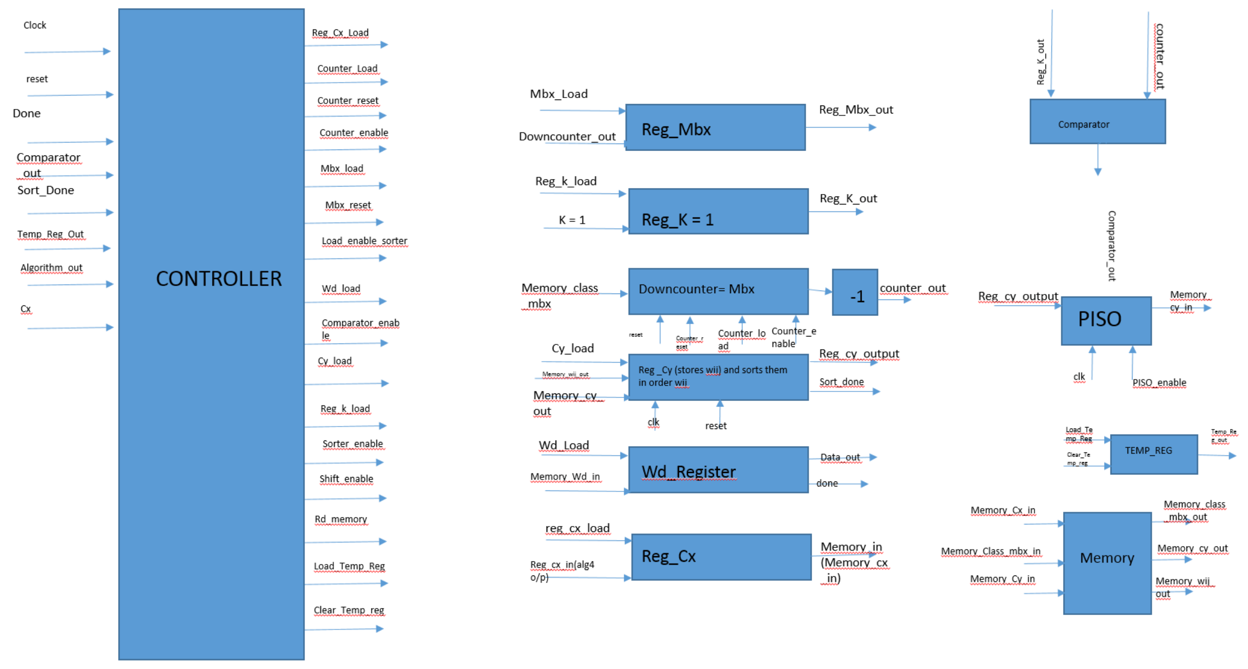
The ouput signals that are high or active in the corresponding states are represented in states and assigned logic 1 and those are low and not active are not mentioned in the state diagram.

4.3 STATE TRANSITION TABLE:

PRESENT_STATE	INPUTS	NEXT_STATE
INITIAL		MEMORY_READ
MEMORY_READ	TEMP_REG_OUT =1	LOAD_COUNTER_REGISTER
MEMORY_READ	TEMP_REG_OUT =0	LOAD_CY _REGISTER
LOAD_COUNTER_REGISTERS		ENABLE_COUNTER
ENABLE_COUNTER		REG_Mbx_LOAD
REG_MBX_LOAD		MEMORY_READ
LOAD_CY_REGISTER	SORT_DONE = 1	COMPARATOR_ENABLE_STATE
LOAD_CY_REGISTER	SORT_DONE = 0	LOAD_CY_REGISTER
COMPARATOR_ENABLE_STATE	COMPARATOR_OUT = 0	ENABLE_COUNTER
COMPARATOR_ENABLE_STATE	COMPARATOR_OUT = 1	BIT_SHIFTING
BIT_SHIFTING		MEMORY_READ_RESPONSE
MEMORY_READ_RESPONSE		OUTPUT
OUTPUT	DONE = 1	INITIAL
OUTPUT	DONE = 0	BIT_SHIFTING

4.4 DATA PATH FOR THIS ALGORITHM:





Rd_memory

4.5 DESCRIPTON OF CONTROLLER:

States in the Controller

- INITIAL:

The outputs of this state are:

Counter_Reset <= '1';
Mbx_Reset <= '1';
Counter_Enable <= '0';
Reg_K_Load <= '1';
Mbx_Load <= '0';
Clear_Temp_Reg <= '0';
Load_Temp_Reg <= '1';
Reg_Cx_Load <= '1';
-- added now
Counter_Load <= '0';
Load_Enable_Sorter <= '0';
Wd_Load <= '0';
Comparator_Enable <= '0';
Cy_Load <= '0';
Sorter_Enable <= '0';
Shift_Enable <= '0';
Rd_Memory <= '0';

This state is triggered if there is a change in the input Cx (i.e. a Algorithm4_out signal from Algorithm 4) , or algorithm 5 has finished its operation or if the reset is triggered.

If the reset is high then the initial state is triggered or the normal process is continued.

- MEMORY_READ:

```
                    Rd_memory <= '1';  -- read mbx from memory
                                        -- added now
                    Counter_Load <= '0';
                    Load_Enable_Sorter <= '0';
                    Wd_Load <= '0';
                    Comparator_Enable <= '0';
                    Cy_Load <= '0';
                    Sorter_Enable <= '0';
                    Shift_Enable <= '0';
                    Counter_Reset <= '0';
                    Mbx_Reset <= '0';
                    Counter_Enable <='0';
                    Reg_K_Load <= '0';
                    Mbx_Load <= '0';
                    Reg_Cx_Load <= '0';
```

This state is triggered after two states they are:

Initial state

And load_cy_register state.

Only one signal is high during this state (i.e. Rd_Memory). And the temp_reg_out input signal to the controller is used to detect what is the next state after MEMORY_READ (i.e. LOAD_COUNTER_REGIDTER or LOAD_CY_REGISTER).

- LOAD_COUNTER_REGISTERS:

```
                    Rd_memory <= '0';  -- read mbx from memory
                                -- added now
                    Counter_Load <= '1';
                    Load_Enable_Sorter <= '0';
                    Wd_Load <= '0';
                    Comparator_Enable <= '0';
                    Cy_Load <= '0';
                    Sorter_Enable <= '0';
                    Shift_Enable <= '0';
                    Counter_Reset <= '0';
                    Mbx_Reset <= '0';
                    Counter_Enable <= '0';
                    Reg_K_Load <= '1';
                    Mbx_Load <= '0';
                    Reg_Cx_Load <= '0';
```

In this state the counter registers are loaded and the counter is not enabled.

- ENABLE_COUNTER:

```
            Rd_memory <= '0';  -- read mbx from memory
                        -- added now
            Counter_Load <= '0';
            Load_Enable_Sorter <= '0';
```

```
Wd_Load <= '0';
Comparator_Enable <= '0';
Cy_Load <= '0';
Sorter_Enable <= '0';
Shift_Enable <= '0';
Counter_Reset <= '0';
Mbx_Reset <= '0';
Counter_Enable <= '1';
Reg_K_Load <= '1';
Mbx_Load <= '0';
Clear_Temp_Reg <= '1';
Load_Temp_Reg <= '0';
Reg_Cx_Load <= '0';
```

In this state the counter is enabled and load counter is disabled.

- REG_MBX_LOAD:

```
Rd_memory <= '0';  -- read mbx from memory
                -- added now
                        Counter_Load <= '0';
                        Load_Enable_Sorter <= '0';
                        Wd_Load <= '0';
                        Comparator_Enable <= '0';
                        Cy_Load <= '0';
                        Sorter_Enable <= '0';
                        Shift_Enable <= '0';
                        Counter_Reset <= '0';
                        Mbx_Reset <= '0';
                        Counter_Enable <= '0';
                        Reg_K_Load <= '1';
                        Mbx_Load <= '1';
                        Reg_Cx_Load <= '0';
```

In this state the Register Mbx is loaded with the output of the counter.

- LOAD_CY_REGISTER:

```
Rd_memory <= '0';  -- read mbx from memory
                -- added now
                        Counter_Load <= '0';
                        Wd_Load <= '0';
                        Comparator_Enable <= '0';
                        Cy_Load <= '1';
                        Shift_Enable <= '0';
                        Counter_Reset <= '0';
                        Mbx_Reset <= '0';
                        Counter_Enable <= '0';
                        Reg_K_Load <= '1';
                        Mbx_Load <= '0';
                        Reg_Cx_Load <= '0';
```

In this state both the CY_register and the sorter are enabled.

- COMPARATOR_ENABLE_STATE:

```
Rd_memory <='0' ;  -- read mbx from memory
                    -- added now
                            Counter_Load <='0' ;
                            Wd_Load <= '0';
                            Comparator_Enable <='1' ;
                            Cy_Load <= '0';
                            Shift_Enable <= '0';
                            Counter_Reset <= '0';
                            Mbx_Reset <= '0';
                            Counter_Enable <='0';
                            Reg_K_Load <= '1';
                            Mbx_Load <= '0';
                            Reg_Cx_Load <= '0';
```

In this state the comparator is enabled. The state following this state can be either

BIT_SHIFITNG (if comparator_out is 1)

Enable_counter (if comparator_out is 0)

- BIT_SHIFTING:

```
Rd_memory <= '1';  -- read mbx from memory
                    -- added now
                            Counter_Load <= '0';
                            Load_Enable_Sorter <= '0';
                            Wd_Load <= '1';
                            Comparator_Enable <= '0';
                            Cy_Load <= '0';
                            Sorter_Enable <= '0';
                            Shift_Enable <= '1';
                            Counter_Reset <= '0';
                            Mbx_Reset <= '0';
                            Counter_Enable <= '0';
                            Reg_K_Load <= '1';
                            Mbx_Load <= '0';
                            Reg_Cx_Load <= '0';
```

In this state the Bit_shifter, rd_memory (to read data from memory) and wd_load (to ouput the data) are enabled, and all work in parallel.

- MEMORY_READ_RESPONSE:

```
Rd_memory <= '1'; -- read Mbx from memory
                    -- added now
                            Counter_Load <= '0';
```

```
--          Load_Enable_Sorter <= '0';
            Wd_Load <= '0';
            Comparator_Enable <= '0';
            Cy_Load <= '0';
            Sorter_Enable <= '0';
            Shift_Enable <= '0';
            Load_Temp_Reg <= '0';
            Clear_Temp_Reg <= '0';

            Counter_Reset <= '0';
            Mbx_Reset <= '0';
            Counter_Enable <='0';
            Reg_K_Load <= '0';
            Mbx_Load <='0' ;
            Clear_Temp_Reg <= '0';
            Load_Temp_Reg <= '0';
            Reg_Cx_Load <= '0';
```

This is a second Memory read state (Memory_read_response different from Memory_read but same functionality) used to output the data to the wd_register. This state is used to reduce the extra hardware and control signals from the memory.

- OUTPUT:

```
Rd_memory <= '1';    -- read mbx from memory
                     -- added now
                           Counter_Load <= '0';
                           Load_Enable_Sorter <= '0';
                           Wd_Load <='1';
                           Comparator_Enable <= '0';
                           Cy_Load <= '0';
                           Sorter_Enable <= '0';
                           Shift_Enable <='1';
                           Load_Temp_Reg <= '0';
                           Clear_Temp_Reg <= '0';

                           Counter_Reset <= '0';
                           Mbx_Reset <= '0';
                           Counter_Enable <='0';
                           Reg_K_Load <= '0';
                           Mbx_Load <= '0';
                           Clear_Temp_Reg <= '0';
                           Load_Temp_Reg <= '0';
                           Reg_Cx_Load <= '0';
```

This is the state the output is obtained, once all the outputs (i.e. the values of the Mbx values are obtained) are outputted, the done signal is asserted from the Wd_register, which triggers the initial state where all the components are rested with the initial values.

If the done signal is not high then the bit shifted values are sequentially shifted to the output till all the output values are finished.

4.6 DESCRIPTION OF DATA PATH:

Components in the data path are:

Reg_cx –

The register is 2 bits, this register stores 2 bits which are class names. The inputs to the register are **reg_cx_load** and **reg_cx_in** and ouput of register is **Memory_in** (input to the memory).

Enabled in the Initial state.

Reg_Mbx –

Inputs: Mbx_load and downcounter_out.
Output: reg_mbx_out

Enabled in the Reg_mbx_load

Reg_k –
Input: Reg_k_load and k =1
Output: Reg_K_out

Enabled in all the states.

Counter –
Input: Memory_class_mbx
Output: counter_out

Used in several states:
1. Load_counter_register and
2. Enable counter states.

Reg_Cy and sorter (Cy_register) –
Input to this register is the memory (i.e. the class Cy and Wij is the input) and the output of this register is the sorted classes (Cy).
The classes are sorted based on the value of the Wij.

This register is 8 bits. The first 6 bits is used to store the value of the "Wij" whereas the last 2 bits are used to store class name.

Wij	Wij	Wij	Wij	Wij	Wij	Cy	Cy

This format is given as input to the sorter, which sorts the classes (Cy) in the order of value of Wij.

Input:
Cy_load

Memory_wij_out
Memory_Cy_out

<u>Output:</u>
Reg_Cy_output
Sort_done

Comparator –
Comparator is used to compare the value of the K and Mbx. This is the equality checking comparator. Once the value of the comparator is one, it is known that the all response classes for different values of Mbx are obtained and are sorted in order. Then the output of the Reg_Cy and sorter are send to the input of the PISO.
<u>Input:</u>
Reg_K_out
Counter_out

<u>Output:</u>
Comparator_out

PISO –
The input from the comparator are send to the memory each clock cycle and are outputted to the Wd_register from the memory.

<u>Input:</u>
Reg_cy_output
Clk
PISO_enable

<u>Output:</u>
Memory_Cy_in

Temp_Reg –
This register is used to know whether the Memory read is a "Mbx" or "Cy and wij" read operation (used in the state machine to determine whether the next state is "Load_Cy_registers" or "Load_counter_registers") after Memory read operation.
<u>Input:</u>
Load_temp_Reg
Clear_temp_reg

<u>Output:</u>
Temp_reg_out

Memory –
<u>Input:</u>
Memory_Cx_in
Memory_Class_mbx_in
Memory_Cy_in

Output:
Memory_class_mbx_out
Memory_cy_out
Memory_wij_out

Wd_Register –
This register is used to output the final output of this implementation.
Input:
Wd_load
Memory_wd_in

Output:
Data_out
Done

4.7 DESIGN HIERARCHY:

The Design is implemented in the following order as shown:
Testbench

- Controller_datapath

Datapath

- Reg_k
- Reg_cx
- Counter
- Reg_mbx
- Temp_reg
- Compartor
- Cy_reigster
- PISO
- Wd_register

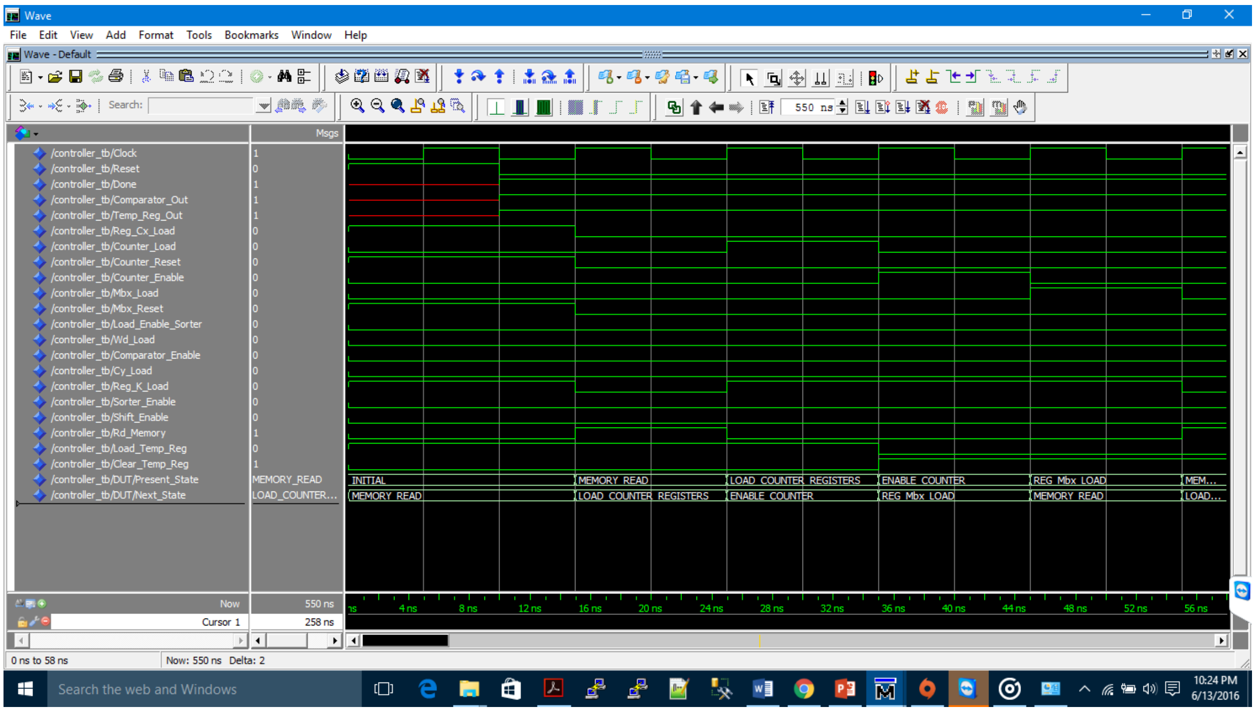
Controller

All the names in the above mentioned hierarchy are names of the entity's in the design.

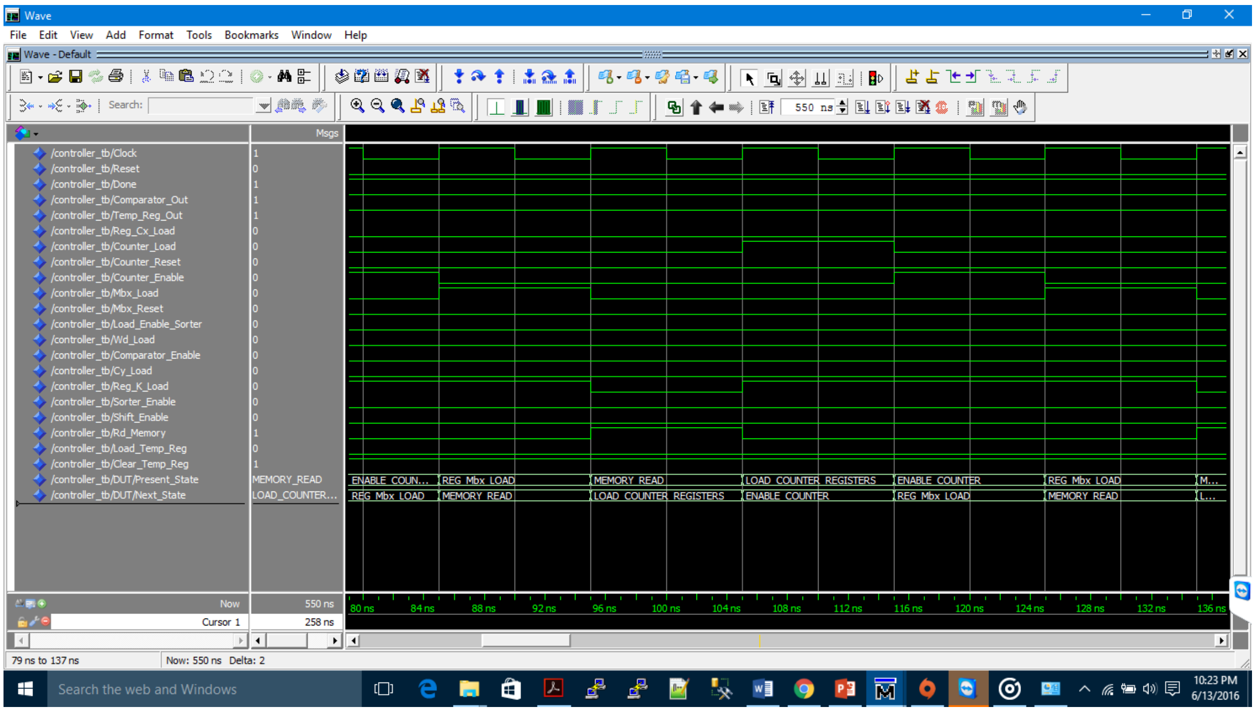
4.8 SIMULATION RESULTS:

For the Controller:

The Test Bench used for the simulation of the controller is “Controller_tb” name of the entity.



```
/controller_tb/Clock                1
/controller_tb/Reset                0
/controller_tb/Done                 1
/controller_tb/Comparator_Out       1
/controller_tb/Temp_Reg_Out         1
/controller_tb/Reg_Cx_Load          0
/controller_tb/Counter_Load         0
/controller_tb/Counter_Reset        0
/controller_tb/Counter_Enable       0
/controller_tb/Mbx_Load             0
/controller_tb/Mbx_Reset            0
/controller_tb/Load_Enable_Sorter   0
/controller_tb/Wd_Load              0
/controller_tb/Comparator_Enable    0
/controller_tb/Cy_Load              0
/controller_tb/Reg_K_Load           0
/controller_tb/Sorter_Enable        0
/controller_tb/Shift_Enable         0
/controller_tb/Rd_Memory            1
/controller_tb/Load_Temp_Reg        0
/controller_tb/Clear_Temp_Reg       1
/controller_tb/DUT/Present_State    MEMORY_READ
/controller_tb/DUT/Next_State       LOAD_COUNTER...

Present_State: INITIAL | MEMORY_READ | LOAD_COUNTER_REGISTERS | ENABLE_COUNTER | REG_Mbx_LOAD | MEM...
Next_State:    MEMORY_READ | LOAD_COUNTER_REGISTERS | ENABLE_COUNTER | REG_Mbx_LOAD | MEMORY_READ | LOAD...

Now 550 ns   Cursor 1 258 ns
0 ns to 58 ns   Now: 550 ns  Delta: 2
```



```
/controller_tb/Clock                1
/controller_tb/Reset                0
/controller_tb/Done                 1
/controller_tb/Comparator_Out       1
/controller_tb/Temp_Reg_Out         1
/controller_tb/Reg_Cx_Load          0
/controller_tb/Counter_Load         0
/controller_tb/Counter_Reset        0
/controller_tb/Counter_Enable       0
/controller_tb/Mbx_Load             0
/controller_tb/Mbx_Reset            0
/controller_tb/Load_Enable_Sorter   0
/controller_tb/Wd_Load              0
/controller_tb/Comparator_Enable    0
/controller_tb/Cy_Load              0
/controller_tb/Reg_K_Load           0
/controller_tb/Sorter_Enable        0
/controller_tb/Shift_Enable         0
/controller_tb/Rd_Memory            1
/controller_tb/Load_Temp_Reg        0
/controller_tb/Clear_Temp_Reg       1
/controller_tb/DUT/Present_State    MEMORY_READ
/controller_tb/DUT/Next_State       LOAD_COUNTER...

Present_State: ENABLE_COUN... | REG_Mbx_LOAD | MEMORY_READ | LOAD_COUNTER_REGISTERS | ENABLE_COUNTER | REG_Mbx_LOAD | M...
Next_State:    REG_Mbx_LOAD | MEMORY_READ | LOAD_COUNTER_REGISTERS | ENABLE_COUNTER | REG_Mbx_LOAD | MEMORY_READ | L...

Now 550 ns   Cursor 1 258 ns
79 ns to 137 ns   Now: 550 ns  Delta: 2
```

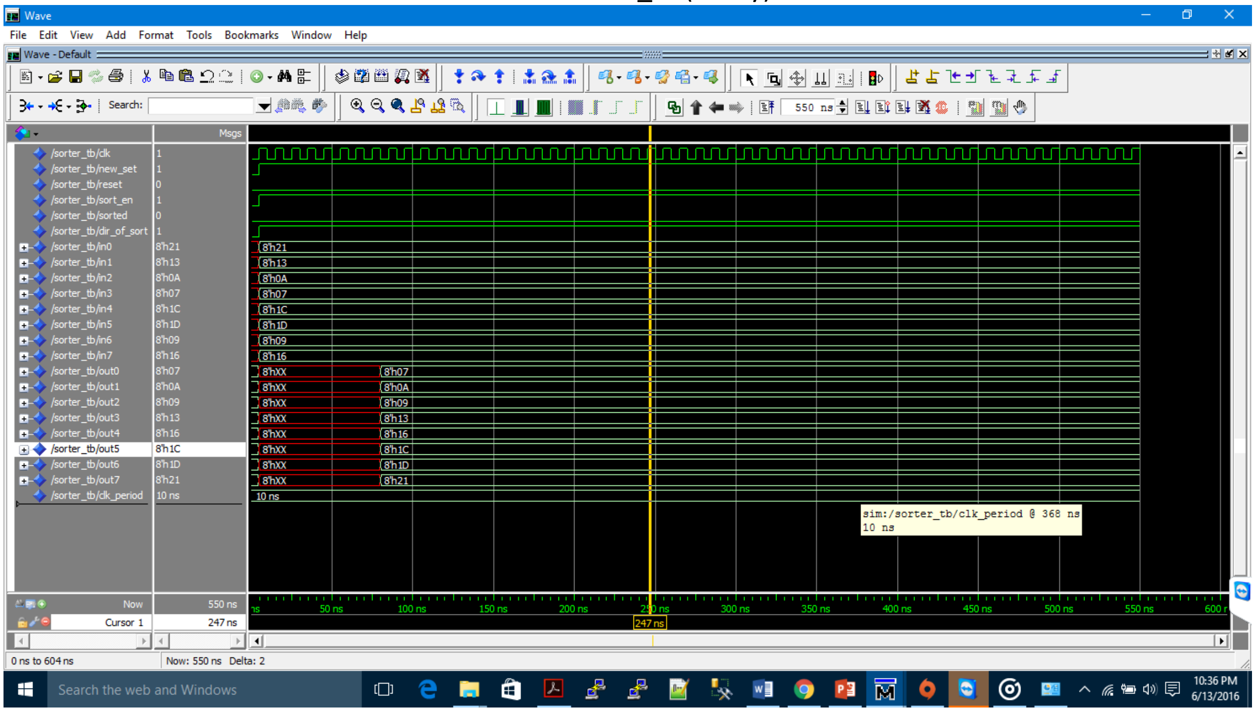
These are simulation results of the controller for these values of the inputs:

```
Reset <= '0';
temp_reg_out <= '1';
Comparator_Out <= '1';
Done <= '1';

wait for temp_period;

temp_reg_out <= '0';
reset <= '0';
Comparator_Out <= '1';
Done <= '1';
```

This Temp_period is given as 450 ns whereas the clock_period is 10 ns. The value of the temp_period is given in order to check whether the controller is functioning properly while fetching the response classes from the memory for all the range of mbx values.

For Sorter:

The test bench used for simulation of sorter is: sorter_td (entity)

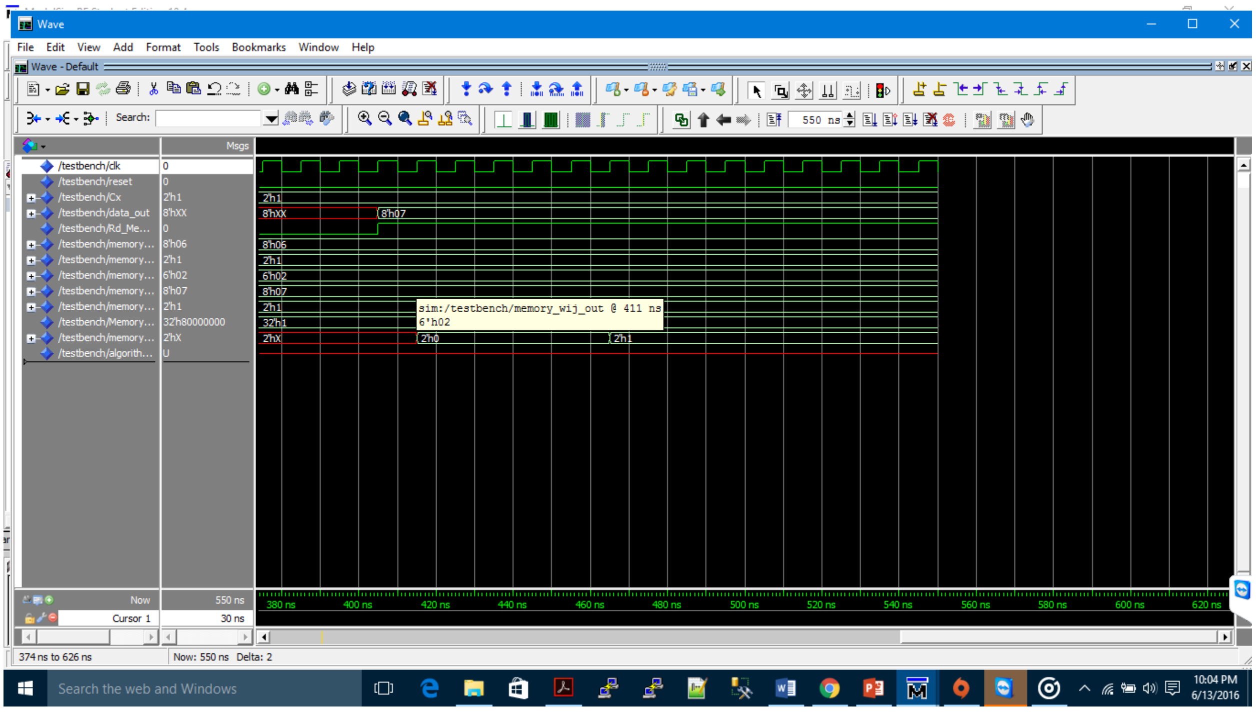


```
/sorter_tb/clk          1
/sorter_tb/new_set      1
/sorter_tb/reset        0
/sorter_tb/sort_en      1
/sorter_tb/sorted       0
/sorter_tb/dir_of_sort  1
/sorter_tb/in0          8'h21
/sorter_tb/in1          8'h13
/sorter_tb/in2          8'h0A
/sorter_tb/in3          8'h07
/sorter_tb/in4          8'h1C
/sorter_tb/in5          8'h1D
/sorter_tb/in6          8'h09
/sorter_tb/in7          8'h16
/sorter_tb/out0         8'h07
/sorter_tb/out1         8'h0A
/sorter_tb/out2         8'h09
/sorter_tb/out3         8'h13
/sorter_tb/out4         8'h16
/sorter_tb/out5         8'h1C
/sorter_tb/out6         8'h1D
/sorter_tb/out7         8'h21
/sorter_tb/clk_period   10 ns

Now       550 ns
Cursor 1  247 ns

sim:/sorter_tb/clk_period @ 368 ns
10 ns

0 ns to 604 ns    Now: 550 ns  Delta: 2
```

The output values of the sorter are out0, out1,..., out7. The inputs of the sorter are in0, in1,..., in7

The outputs (lower 2 bits (response class Cy)) are sorted based on the first 6 bits of inputs to the sorter.

FOR Algorithm 5 final output:



```
/testbench/clk            0
/testbench/reset          0
/testbench/Cx             2'h1
/testbench/data_out       8'hXX
/testbench/Rd_Me...       0
/testbench/memory...      8'h06
/testbench/memory...      2'h1
/testbench/memory...      6'h02
/testbench/memory...      8'h07
/testbench/memory...      2'h1
/testbench/Memory...      32'h80000000
/testbench/memory...      2'hX
/testbench/algorith...    U

sim:/testbench/memory_wij_out @ 411 ns
6'h02

Now       550 ns
Cursor 1   30 ns
374 ns to 626 ns      Now: 550 ns  Delta: 2
```

The inputs given to the design are from the test bench and the memory inputs to the design are also given from the test bench.

The Cx value given is “01”, which is the input to the whole module and the reset value is 0 and the clk_period is 10 ns.

And the values from the memory for the values of Mbx, cy, wij and wd_out are:
memory_class_mbx_out <= "00000110";
memory_cy_out <= "01";
memory_wij_out <= "000010";
memory_wd_out <= "00000111";

The Memory_wd_out which is the response vector of the response class is reflected at the data_out port of the design.

Chapter 5

5.1 Contributions

5.1.1 **Vishwas Reddy Pulugu**: Algorithm 1

- Design of data-path and controller for Algorithm 1

- VHDL code
  mem_structure.vhd
  memory_layer.vhd
  connection_mem.vhd
  controller_mem_layer.vhd
  calculate_ws1_ws2_ths1.vhd
  memory.vhd
  comparator.vhd
  min2.vhd
  node_counter
  mux4X1.vhd
  demux4X1.vhd

5.1.2 **Bharath Reddy Godi** : Algorithm 2

- Design of data-path and controller for Algorithm 1
- VHDL code
  pacakage.vhd
  memory_layer.vhd
  2to1_mux.vhd
  generic_adder.vhd
  associative_mem_controller.vhd

5.1.3 **Surendra Maddula** : Algorithm 4

- Design of data-path and controller for Algorithm 1
- VHDL code

  package.vhd
  algorith4.vhd
  algorith4_tb.vhd
  sub.vhd
  mul.vhd
  generic_adder.vhd

5.1.4 **Vasu Gankidi**: Algorithm 5

- Design of datapath and controller for Algorithm 5
- VHDL code for algorithm 5
  Controller_datapath.vhd
  Datapath.vhd
  Controller.vhd
  Testbench.vhd
  Register.vhd (sorter)

5.1.5 Harsh Sharma :

- Documentation
- VHDL code for modules in Algorithm 1

Adder_ED_calc.vhd
Euclidean_distance.vhd
Squarerootfunction.vhd

5.1.6 **Nikhil Marda :** Algorithm 2

- VHDL code for modules in Algorithm 2
  associative_memory.vhd
  associative_memory_controller_tb.vhd

5.2 Conclusion:

The GAM system implemented here utilizes three layers: Input, Memory, Associative layers. The inputs are memorized in the memory layer and the associative relationships are built in the associative layer. The associative layer accommodates the one to one and one to many and many to many associations.

References:

- Furao Shen, Quibao Ouyang, Wataru Kasai, Osamu Hasegawa. A general associative memory based on self- organizing incremental neural network.
- Kamela Choudary Rahman, Memristive stateful imply logic based reconfigurable architecture.
- http://web.cecs.pdx.edu/~mperkows/CLASS_VHDL/index.html
- www.Stackoverflow.com
- http://web.cecs.pdx.edu/~mperkows/CLASS_VHDL/VHDL/Yvonne_sorter/sorter.htm l