Design Specifications

Andrew Gilbert, Calli Clark, Thomas Becnel[1]

[1]Department of Electrical and Computer Engineering, University of Utah, Salt Lake City, UT

December 17, 2015

Contents

1 Project Block Diagram **3**

2 Floorplan **3**

3 External Interface Diagram **5**

4 Pinout **5**

5 Component Block Diagrams and Layouts **8**
5.1 Neuron Block Diagram 8
5.2 Arbiter Block Diagram 8
5.3 Neuron Layout 8
5.4 Arbiter Layout 8

6 Verilog **12**
6.1 Neural Network (Top Module) 12
6.2 Arbiter 15
6.3 Neuron 19
6.4 Neural Network Testbench 25

7 Test Results **28**
7.1 Verilog Testbench 28
7.2 Python Module 31

8 Project Directories **35**

List of Figures

1 Overall Project Block Diagram 3
2 Neural Network Floorplan 4
3 External Interface 5
4 Neuron Block Diagram 8
5 Arbiter Block Diagram 9
6 Neuron Block Diagram 10
7 Arbiter Block Diagram 11
8 Testing Waveform 1: Request lines 28
9 Testing Waveform 2: Receiving Neuron Cycle 29
10 Testing Waveform 3: Neuron operation 30
11 Testing Waveform 4: Setting thresholds 30
12 Testing Waveform 5: Setting leaks 31

List of Tables

1 Pinout specifications for the final chip design 7

1 Project Block Diagram

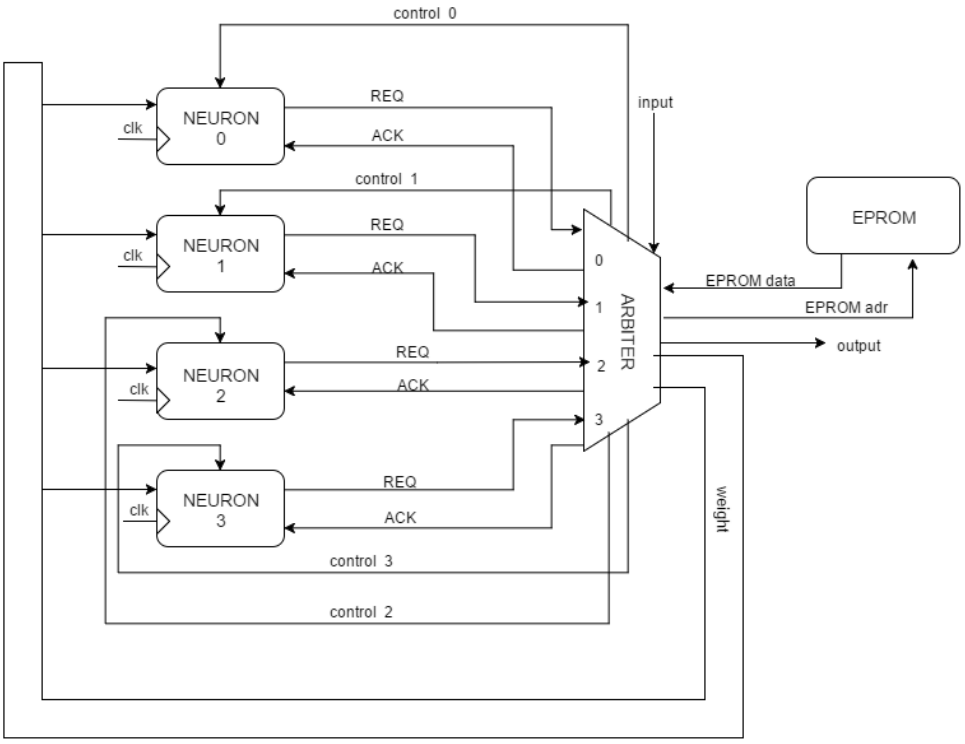

Figure 1: Overall Project Block Diagram

Fig. 1 shows the overall block diagram of our project. There are two main models, the neuron and the arbiter. There are 16 neurons connected to the arbiter although the diagram only shows 4 for simplicity. Each neuron has a request line to the arbiter which controls a acknowledge line and a set of enable signals to each neuron. There is a main bus which allows the arbiter to sequentially pass data to the neurons.

2 Floorplan

Fig. 2 shows the final layout floorplan of our project. We made the decision to implement our project as one large layout to ensure that the maximum possible number of neurons could be fit on the chip. We did do a layout and test of both the neuron (layout: Fig. 6) and arbiter (layout: Fig. 7) to ensure functionality, but the final project contained only the top module synthesized.

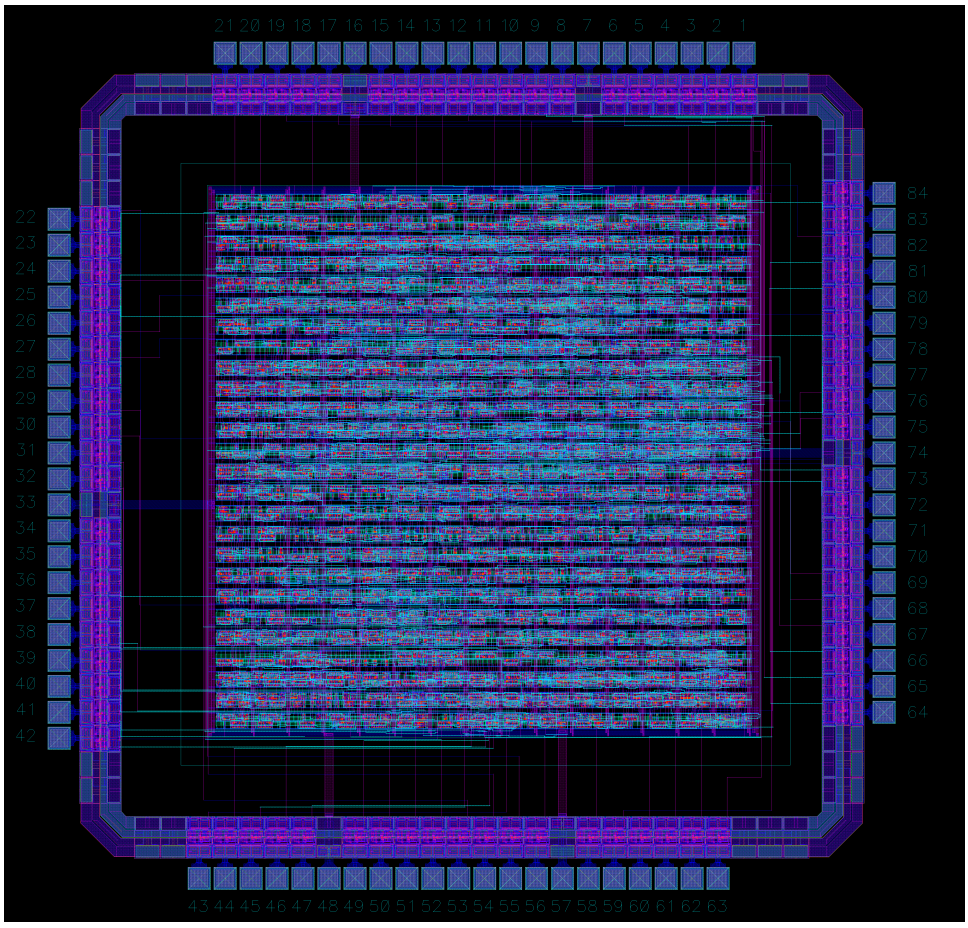

Figure 2: Neural Network Floorplan

3 External Interface Diagram

Fig. 3 shows the external interface design. The input to the network (in the form of a weight to be applied to the neurons) may either come from any 8 bit bus, or from an external EPROM. The output is set up in the same manner. The addresses for the input EPROM and output RAM are assigned to be the same to save on pins and so that the input output sequence should be recorded in a sequential manner if storage is desired such that an input signal can be matched to the corresponding output signal.

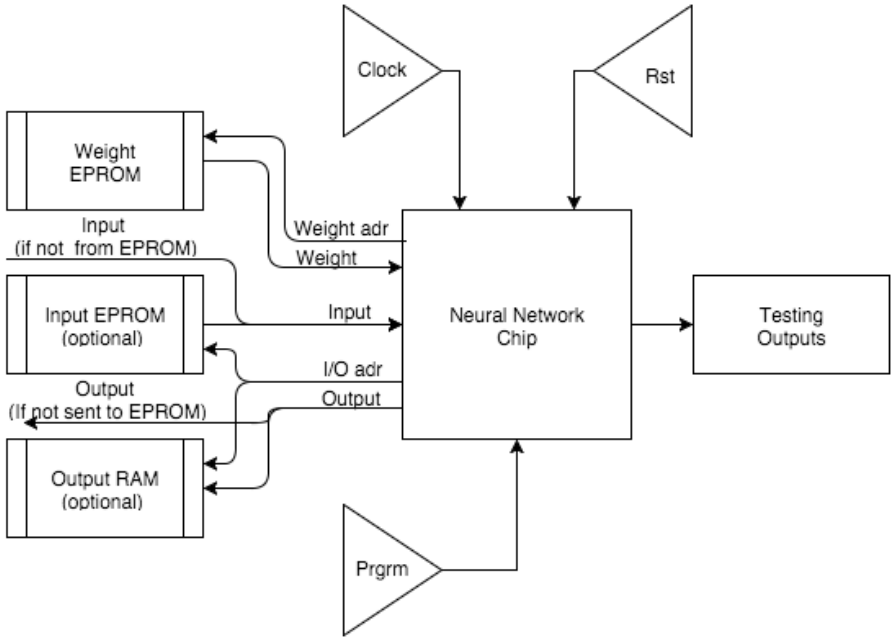

Figure 3: External Interface

4 Pinout

Number	Name	Index	Dir	Description
1	Clk	-	I	Clock
2	Rst	-	I	Global reset
3	InWght	7	I	Incoming weights between neurons
4	InWght	6	I	Incoming weights between neurons
5	InWght	5	I	Incoming weights between neurons
6	InWght	4	I	Incoming weights between neurons
7	Gnd	-	I/O	Global ground
8	InWght	3	I	Incoming weights between neurons
9	InWght	2	I	Incoming weights between neurons
10	InWght	1	I	Incoming weights between neurons
11	InWght	0	I	Incoming weights between neurons
12	InPr	1	I	Programming input to set leak and threshold
13	InPr	0	I	Programming input to set leak and threshold
14	EPROM_Data	7	I	Input signal to neurons
15	EPROM_Data	6	I	Input signal to neurons
16	Vdd	-	I/O	Global power
17	EPROM_Data	5	I	Input signal to neurons
18	EPROM_Data	4	I	Input signal to neurons
19	EPROM_Data	3	I	Input signal to neurons

20	EPROM_Data	2	I	Input signal to neurons
21	EPROM_Data	1	I	Input signal to neurons
22	EPROM_Data	0	I	Input signal to neurons
23	EPROM_adr	9	O	Address for weight EPROM
24	EPROM_adr	8	O	Address for weight EPROM
25	EPROM_adr	7	O	Address for weight EPROM
26	EPROM_adr	6	O	Address for weight EPROM
27	EPROM_adr	5	O	Address for weight EPROM
28	EPROM_adr	4	O	Address for weight EPROM
29	EPROM_adr	3	O	Address for weight EPROM
30	EPROM_adr	2	O	Address for weight EPROM
31	EPROM_adr	1	O	Address for weight EPROM
32	EPROM_adr	0	O	Address for weight EPROM
33	Vdd	-	I/O	Global power
34	In_adr	14	O	Address for Input/Output EPROM
35	In_adr	13	O	Address for Input/Output EPROM
36	In_adr	12	O	Address for Input/Output EPROM
37	In_adr	11	O	Address for Input/Output EPROM
38	In_adr	10	O	Address for Input/Output EPROM
39	In_adr	9	O	Address for Input/Output EPROM
40	In_adr	8	O	Address for Input/Output EPROM
41	In_adr	7	O	Address for Input/Output EPROM
42	In_adr	6	O	Address for Input/Output EPROM
43	In_adr	5	O	Address for Input/Output EPROM
44	In_adr	4	O	Address for Input/Output EPROM
45	In_adr	3	O	Address for Input/Output EPROM
46	In_adr	2	O	Address for Input/Output EPROM
47	In_adr	1	O	Address for Input/Output EPROM
48	Gnd	-	I/O	Global ground
49	In_adr	0	O	Address for Input/Output EPROM
50	LIFNO	7	O	DEBUG: Neuron 0 current membrane potential
51	LIFNO	6	O	DEBUG: Neuron 0 current membrane potential
52	LIFNO	5	O	DEBUG: Neuron 0 current membrane potential
53	LIFNO	4	O	DEBUG: Neuron 0 current membrane potential
54	LIFNO	3	O	DEBUG: Neuron 0 current membrane potential
55	LIFNO	2	O	DEBUG: Neuron 0 current membrane potential
56	LIFNO	1	O	DEBUG: Neuron 0 current membrane potential
57	Vdd	-	I/O	Global power
58	LIFNO	0	O	DEBUG: Neuron 0 current membrane potential
59	Output	7	O	System output: contains current spikes
60	Output	6	O	System output: contains current spikes
61	Output	5	O	System output: contains current spikes
62	Output	4	O	System output: contains current spikes
63	Output	3	O	System output: contains current spikes
64	Output	2	O	System output: contains current spikes
65	Output	1	O	System output: contains current spikes

66	Output	0	O	System output: contains current spikes
67	REQ	15	O	DEBUG: Neuron request lines
68	REQ	14	O	DEBUG: Neuron request lines
69	REQ	13	O	DEBUG: Neuron request lines
70	REQ	12	O	DEBUG: Neuron request lines
71	REQ	11	O	DEBUG: Neuron request lines
72	REQ	10	O	DEBUG: Neuron request lines
73	REQ	9	O	DEBUG: Neuron request lines
74	Gnd	-	I/O	Global ground
75	REQ	8	O	DEBUG: Neuron request lines
76	REQ	7	O	DEBUG: Neuron request lines
77	REQ	6	O	DEBUG: Neuron request lines
78	REQ	5	O	DEBUG: Neuron request lines
79	REQ	4	O	DEBUG: Neuron request lines
80	REQ	3	O	DEBUG: Neuron request lines
81	REQ	2	O	DEBUG: Neuron request lines
82	REQ	1	O	DEBUG: Neuron request lines
83	REQ	0	O	DEBUG: Neuron request lines
84	-	-	-	-

Table 1: Pinout specifications for the final chip design

Table 1 shows the pinout for the fabricated chip. Note that not all address lines are output by the chip. This is because only a 256 Byte sized space is needed for the weight array with an additional bit reserved so that the leak and threshold values can be stored in this EPROM. So the remaining address bits, as well as the output enable lines, are tied to ground. This gives more space for debugging signals.

5 Component Block Diagrams and Layouts

5.1 Neuron Block Diagram

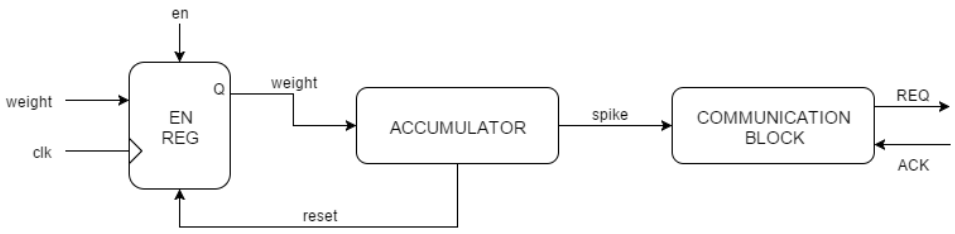

Figure 4: Neuron Block Diagram

The patented Calli, Andy, & Tom (CAT) Leaky Integrate and Fire (LIF) neuron model design can be seen in Fig. 4.

5.2 Arbiter Block Diagram

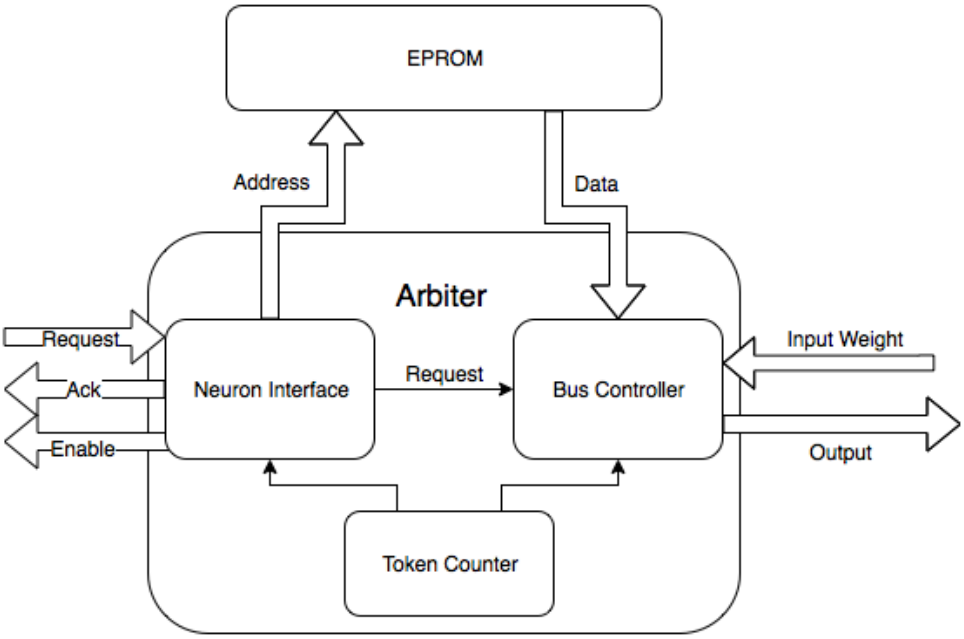

Figure 5: Arbiter Block Diagram

The patented CAT token based bus arbitration system design can be seen in Fig. 5.

5.3 Neuron Layout

The patented CAT neuron layout can be seen in Fig. 6. Note that this design was not implemented in the final chip, but rather just synthesized inside the top module.

5.4 Arbiter Layout

The patented CAT arbiter layout can be seen in Fig. 6. Note that this design was not implemented in the final chip, but rather just synthesized inside the top module.

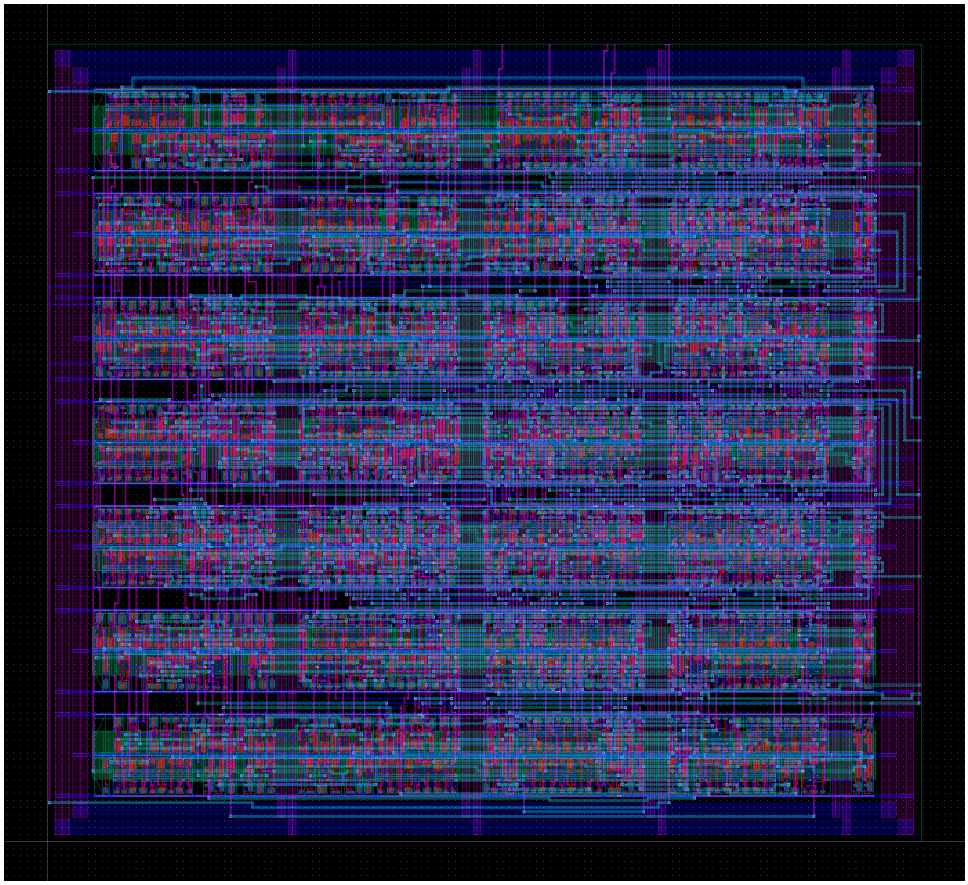

Figure 6: Neuron Block Diagram

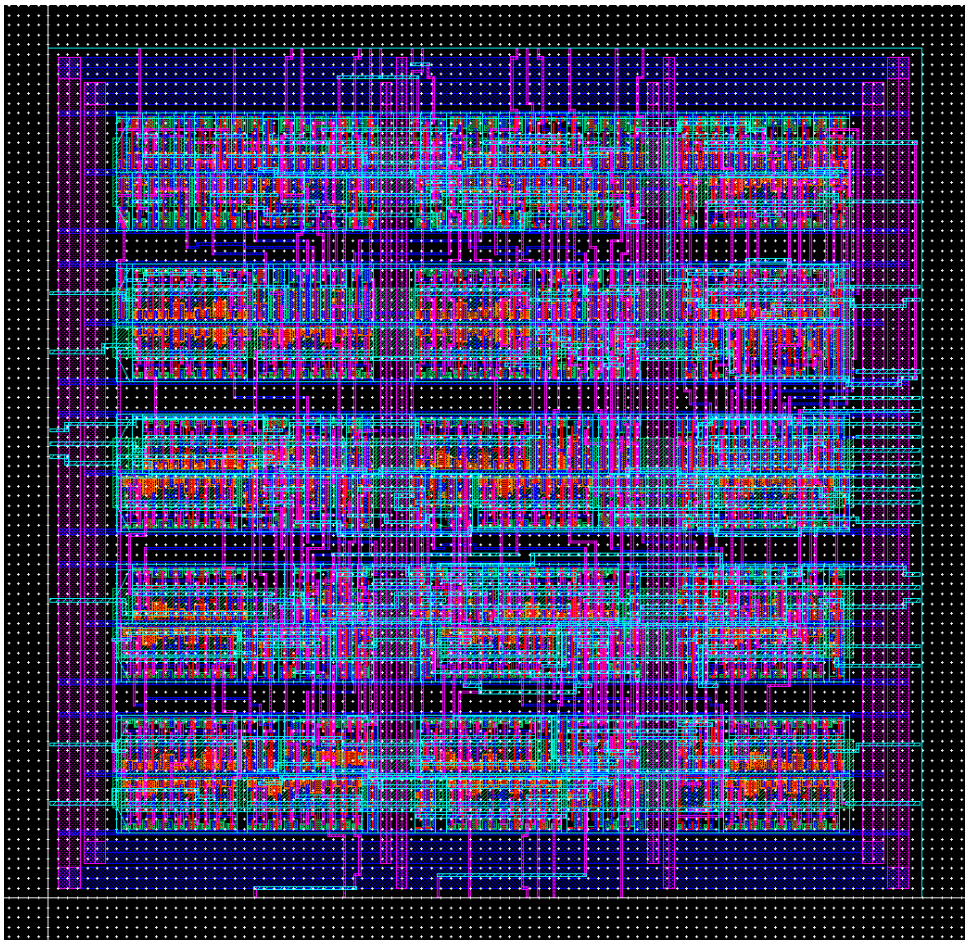

Figure 7: Arbiter Block Diagram

6 Verilog

6.1 Neural Network (Top Module)

```
`timescale 1ns / 1ps
//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
// Create Date:  15:32:48 12/01/2015
// Design Name:
// Module Name:  top_module
// Project Name:
// Target Devices:
// Tool versions:
// Description:
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
//////////////////////////////////////////////////////////////////////////////////
module neural_network #(parameter WEIGHT_SZ = 8,   // size of weight bus
                  parameter NUM_NEURONS = 16,  // number of neurons
                  parameter NRN_WIDTH = $clog2(NUM_NEURONS), // Number
                      of bits necessary for token
                  parameter STARTING_ADR = 0) // starting memory address
                      (unused currently)
                  (clk, rst, input_weight, EPROM_data, input_pr,
                      output_stub, EPROM_adr,
                  input_address, OE, request, token_counter,
                      LIF_counter0);
   input clk, rst; // done
   input signed [WEIGHT_SZ-1:0] input_weight; // done
   input signed [1:0] input_pr; // done
   input [7:0] EPROM_data; // done
   output [WEIGHT_SZ-1:0] output_stub; // done
   output [14:0] EPROM_adr; // done
   output reg [14:0] input_address; // done
   output OE; // done
   output [NUM_NEURONS-1:0] request; // done
   output [2:0] token_counter; // done
   output [WEIGHT_SZ-1:0] LIF_counter0;

//arbiter neuron interface
wire [NUM_NEURONS-1:0] acknowledge, nrn_en; // control signals between
    arbiter and neurons
```

```
wire [WEIGHT_SZ-1:0] weight;

// EEPROM interface
//wire [7:0] EPROM_data;
//wire [14:0] EPROM_adr;
//wire OE; // memory enable, active low
assign OE = 0;
assign output_stub = weight - input_weight;

always @(posedge clk, posedge rst) begin
   if (rst) begin
      input_address <= 15'b0;
   end
   else begin
      input_address <= input_address + 1;
   end
end


////////////////////////////////////////////////////////////////////////////////
// ARBITER
// clk, RST: global
// input [7:0] EEPROM_data;        // data from EEPROM regarding neuron
    weight
// input [WEIGHT-1:0] input_weight;   // input from external source
// input [NUM_NEURONS-1:0] request;   // input spike from neurons
// output [NUM_NEURONS-1:0] acknowledge;  // acknowledge
//  output [NUM_NEURONS-1:0] en;    // enable register of neurons for
    input
// output [14:0] EEPROM_adr;          // means we are looking at input
    data
// output output_stub;             // no clue what output is yet
////////////////////////////////////////////////////////////////////////////////

arbiter #(WEIGHT_SZ, NUM_NEURONS, NRN_WIDTH, STARTING_ADR) // defines
    EEPROM starting address
   arb (clk,rst,EPROM_data,input_weight,request,input_pr,
       acknowledge,EPROM_adr, weight, nrn_en, token_counter);

////////////////////////////////////////////////////////////////////////////////
// NEURONS
//  input clk, en, A,reset;
//  input signed [WEIGHT_SZ-1:0] weight;
//  output        R;
////////////////////////////////////////////////////////////////////////////////
neuron_out #(WEIGHT_SZ)
   nrn0 (clk, nrn_en[0], acknowledge[0], rst, input_pr, weight,
       request[0], LIF_counter0);

neuron #(WEIGHT_SZ)
```

```
   nrn1 (clk, nrn_en[1], acknowledge[1], rst, input_pr, weight,
       request[1]);

neuron #(WEIGHT_SZ)
   nrn2 (clk, nrn_en[2], acknowledge[2], rst, input_pr, weight,
       request[2]);

neuron #(WEIGHT_SZ)
   nrn3 (clk, nrn_en[3], acknowledge[3], rst, input_pr, weight,
       request[3]);

neuron #(WEIGHT_SZ)
   nrn4 (clk, nrn_en[4], acknowledge[4], rst, input_pr, weight,
       request[4]);

neuron #(WEIGHT_SZ)
   nrn5 (clk, nrn_en[5], acknowledge[5], rst, input_pr, weight,
       request[5]);

neuron #(WEIGHT_SZ)
   nrn6 (clk, nrn_en[6], acknowledge[6], rst, input_pr, weight,
       request[6]);

neuron #(WEIGHT_SZ)
   nrn7 (clk, nrn_en[7], acknowledge[7], rst, input_pr, weight,
       request[7]);

neuron #(WEIGHT_SZ)
   nrn8 (clk, nrn_en[8], acknowledge[8], rst, input_pr, weight,
       request[8]);

neuron #(WEIGHT_SZ)
   nrn9 (clk, nrn_en[9], acknowledge[9], rst, input_pr, weight,
       request[9]);

neuron #(WEIGHT_SZ)
   nrn10 (clk, nrn_en[10], acknowledge[10], rst, input_pr, weight,
       request[10]);

neuron #(WEIGHT_SZ)
   nrn11 (clk, nrn_en[11], acknowledge[11], rst, input_pr, weight,
       request[11]);

neuron #(WEIGHT_SZ)
   nrn12 (clk, nrn_en[12], acknowledge[12], rst, input_pr, weight,
       request[12]);

neuron #(WEIGHT_SZ)
   nrn13 (clk, nrn_en[13], acknowledge[13], rst, input_pr, weight,
       request[13]);
```

```
neuron #(WEIGHT_SZ)
   nrn14 (clk, nrn_en[14], acknowledge[14], rst, input_pr, weight,
       request[14]);

neuron #(WEIGHT_SZ)
   nrn15 (clk, nrn_en[15], acknowledge[15], rst, input_pr, weight,
       request[15]);



endmodule
```

6.2 Arbiter

```
//////////////////////////////////////////////////////////////////////
//
// Arbiter
//
// Written by: Calli Clark, Andy Gilbert, Tom Becnel
//
// Inputs:
//    clk
//    rst
//    data_in (input data to network)
//    request (request lines from neurons)
//    ack     (acknowledge lines from neurons)
// Outputs:
//
//////////////////////////////////////////////////////////////////////

module arbiter #(
   parameter WEIGHT     = 8,  // bit width of weight bus
   parameter NUM_NEURONS = 3, // 1 hot encoding bus for all neurons
   parameter NRN_WIDTH  = $clog2(NUM_NEURONS), // bit width for counting
       through neurons
   parameter STARTING_ADR = 0) // defines EEPROM starting address
   (clk,RST,EEPROM_data,input_weight,request,input_pr,acknowledge,EEPROM_adr,output_stub,
       nrn_en, token_counter);

   input clk, RST;
   input [7:0] EEPROM_data;           // data from EEPROM regarding
       neuron weight
   input [WEIGHT-1:0] input_weight;   // input from external source
   input [NUM_NEURONS-1:0] request;   // input spike from neurons
   input [1:0] input_pr;
```

```
    output reg [NUM_NEURONS-1:0] acknowledge;  // acknowledge
    output [14:0] EEPROM_adr;          // means we are looking at input
        data
    output [WEIGHT-1:0] output_stub;              // no clue what output
        is yet
    output [NUM_NEURONS-1:0] nrn_en;
    output [2:0] token_counter;

    reg [NRN_WIDTH-1:0] token; // token counter
    reg [NRN_WIDTH-1:0] nrn_receive;
    reg nrn_pulse; // this clock pulses (1 clk cycle pulse) every n
        cycles where n is the number of neurons

    //wire [14:0] EEPROM_adr_wire;
    //wire [WEIGHT-1:0] weight_wire;

    //assign output_stub = weight_wire; //just a little stubbly stub
    //assign EEPROM_adr = EEPROM_adr_wire;
    assign token_counter = token[2:0];

    read #(NRN_WIDTH, STARTING_ADR)
        r(clk, RST, request[token], token, nrn_receive, input_pr,
            EEPROM_adr);

    controller #(NUM_NEURONS, WEIGHT, NRN_WIDTH)
         c(clk, EEPROM_data, input_weight, nrn_receive, input_pr,
             output_stub, nrn_en);


// 'nrn-receive' counter for neuron to output weight to from input neuron
    always @ (posedge clk, posedge RST) begin

       if (RST) nrn_receive <= 0;
       else
          if (nrn_receive >= NUM_NEURONS-1) nrn_receive <= 0;
          else nrn_receive <= nrn_receive + 1;
    end

// pulse wave after all neurons have been cycled through
    always @ (posedge clk) begin

       if (RST) nrn_pulse <= 0;
       else if (nrn_receive == NUM_NEURONS-1) nrn_pulse <= 1'b1;
       else nrn_pulse <= 0;

    end

// 'token' counter loops through neurons for receive/ack
//      updates
    always @ (posedge nrn_pulse, posedge RST) begin
```

```
      if (RST) token <= 0;
      else
         if (token == NUM_NEURONS-1) token <= 0;
         else token <= token + 1;

   end

// set 'acknowledge' output at every positive edge of this pulse
//      if request was received
//
   always @ (posedge clk, posedge RST) begin

      if (RST) acknowledge <= 0;
      else if (input_pr) begin
         acknowledge <= 1;

      end
      else begin
         acknowledge <= 0;
         if (request[token] == 1)
            if (nrn_receive == NUM_NEURONS-1)
               acknowledge[token] <= 1;
         else if (acknowledge == 1)
               acknowledge <= 0;
      end
   end


endmodule


// Reads the current neuron and sends address to EEPROM
module read #(
   parameter NRN_WIDTH  = 0,
   parameter STARTING_ADR = 0)
   (clk, RST, request, nrn_send, nrn_receive, input_pr, adr);

   input clk, RST, request;  // request signals from neuron
   //input [NUM_NEURONS-1:0] acknowledge; // acknowledge signals from
       neuron. --> set in 'arbiter' module now
   input [NRN_WIDTH-1:0] nrn_send; // address of sending neuron.
       corresponds to token in outer arbiter module
   input [NRN_WIDTH-1:0] nrn_receive; // address of receiving neuron
   input  [1:0] input_pr;
   output reg [14:0] adr;

   parameter [13-(2*NRN_WIDTH):0] filler = 0; // to be concatonated. 13
       minus b/c 1 added to nrn_send + nrn_receive
```

```
   // this module reads the token neuron on every positive edge of the
       clk.
   // if request is high then it sends a request to the EEPROM for the
       weight.


   // Assign address - 0 address means no spike
   always @ (posedge clk) begin

      if (RST) adr <= STARTING_ADR;
      else
         case (input_pr)
            2'b01: adr <= {14'b0, 1'b1}; // getting threshold value
            2'b10: adr <= {13'b0, 2'b10}; // getting leak value
            default: begin
               adr <= 15'b0;
               if (request == 1'b1) begin // spike occured at the
                   specified neuron
                  adr <= {filler, 1'b1, nrn_send,nrn_receive};
               end
            end
      endcase

   end

endmodule


// adds weight from input and ROM table for neurons
module controller #(
   parameter NUM_NEURONS = 0,
   parameter WEIGHT  = 0,
   parameter NRN_WIDTH = 0)
   (clk, EEPROM_data, input_weight, receiving_nrn, input_pr, weight,
       nrn_en);

   input clk;
   input [7:0] EEPROM_data;
   input [WEIGHT-1:0] input_weight; // System input
   input [NRN_WIDTH-1:0] receiving_nrn;
   input [1:0] input_pr;
   output reg [WEIGHT-1:0] weight; // Weight to put on bus
   output reg [NUM_NEURONS-1:0] nrn_en; // which neuron to enable

   // Update weight
   always @ (posedge clk) begin

      case(input_pr)
         0: begin // normal operation
```

```
            weight <= EEPROM_data + input_weight; // ??? overflow
                detection ???
            nrn_en <= 'b0 | (1 << receiving_nrn);
          end
          default: begin // we are setting a leak or threshold
            weight <= EEPROM_data;
            nrn_en <= 2**NUM_NEURONS-1; // all enable signals go high
          end

      endcase


    end

endmodule
```

6.3 Neuron

There are two neuron modules below. They are identical, except one outputs the value of its counter. This was implemented as a debugging output at the top level to make sure it was changing properly.

```
`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////
//
// Top Module: neuron
// 11/24/2015
//
// Written by: Calli Clark, Andy Gilbert, Tom Becnel
//
// Inputs: clk, en, ack, weight, bus_program
// Output: req
//
//////////////////////////////////////////////////////////////////////////////////


module neuron #(parameter WEIGHT_SZ = 8)
              (clk, en, ack, reset, program_bus, weight, req);
  input clk, en, ack ,reset;
  input [1:0] program_bus;
  input signed [WEIGHT_SZ-1:0] weight;
  output        req;

//connect en_reg, accumulator, and comm_block modules
   wire            reg_reset;
   wire signed [WEIGHT_SZ-1:0] Q;
```

```
   wire            spike;

   en_reg #(WEIGHT_SZ)
   en_reg0(clk, en, reg_reset, weight, Q);

   accumulator #(WEIGHT_SZ)
   acc0(clk, reset, program_bus, Q, reg_reset, spike);

   comm_block comm0(clk, reset, spike, ack, req);

endmodule // neuron

//////////////////////////////////////////////////////
//
// module: en_reg
//
// inputs: clk, en, reset, weight
// output: Q
//
// Enabled register that passes the input weight as the
// output if the enabled. Reset is active low.
//
//////////////////////////////////////////////////////
module en_reg #(WEIGHT_SZ = 8)
              (clk, en, reset, weight, Q);

   input clk, en, reset;
   input signed [WEIGHT_SZ-1:0] weight;
   output reg signed [WEIGHT_SZ-1:0] Q;

   always@(posedge clk, negedge reset)begin
     if(reset == 0)
       Q <= 0;
     else if(en)
       Q <= weight;
   end
endmodule // en_reg

//////////////////////////////////////////////////////
//
// module: accumulator
//
// inputs: clk, reset, program_bus, weight
// outputs: reg_reset, spike
// this module contains the accumulator which has the current neuron
    voltage level stored
// if a weight appears on the enabled register it adds that weight,
    otherwise it subtracts the leak
// if the counter goes over a set threshold value then a spike is fired
    and the accumulator is reset to 0
```

```
//
//////////////////////////////////////////////////////
module accumulator #(parameter WEIGHT_SZ = 8)
                    (clk, reset, program_bus, weight, reg_reset, spike);
   input clk, reset;
   input [1:0] program_bus;
   input signed [WEIGHT_SZ-1:0] weight;
   output reg    reg_reset, spike;


   reg signed [WEIGHT_SZ-1:0] counter;

   //counter reg
   //parameter COUNTER_SZ = 8;

   reg [WEIGHT_SZ-1:0] FIRE_POTEN, LEAK;

//   //fire potential parameter
//   parameter FIRE_POTENT = 63;
//   parameter LEAK = 2;
// // eventually this should be an input!!!!!!!!


   always@(posedge clk) begin
    reg_reset <= 1;
    if(reset) begin
      counter <= 0;
      reg_reset <= 0;
    end
    else
       case(program_bus)
       2'b00:
         if ($signed(counter) >= $signed(FIRE_POTEN)) begin
            counter <= 0;
         end
         else if(weight != 0) begin
            counter <= counter + weight;
            reg_reset <= 0; //reset reg
         end
         else if($signed(counter) > 0) begin
            counter <= counter - LEAK;
         end // else: !if(weight != 0)
         else if ($signed(counter) < 0) begin
            counter <= counter + LEAK;
         end

       2'b01: //bus value sets fire potential
       begin
         counter <= 0;
         FIRE_POTEN <= weight;
```

```
      end

      2'b10: //bus value sets leak
      begin
        counter <= 0;
        LEAK <= weight;
      end

      default:
      begin
        counter <= 0;
        FIRE_POTEN <= 60;
        LEAK <= 2;
      end
      endcase
   end // always@ (posedge clk)

   always @(counter, reset) begin
      if (reset) begin
        spike = 0;
      end
      else if ($signed(counter) >= $signed(FIRE_POTEN)) begin
        spike = 1;
      end
      else  begin
        spike = 0;
      end
   end

endmodule // accumulator

//////////////////////////////////////////////////////
//
// module: comm_block
//
// inputs: clk, RST, spike, ack
// output: req
//
// when this module records that a spike occurred in the neuron
// the comm_block sets req high and waits for an acknowledge to set req
    low
//
//////////////////////////////////////////////////////
module comm_block (clk, RST, spike, ack, req);
   input clk, RST, spike, ack;
   output req;

   wire reset_R;

   assign reset_R = ~(ack | RST);
```

```
   en_reg #(1) //use an enabled reg for the output block which is
       enabled when spike comes in and reset to 0 when ack occurs
   en_reg0(clk, spike, reset_R, spike, req);

endmodule // comm_block


`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////
//
// Top Module: neuron
// 11/24/2015
//
// Written by: Calli Clark, Andy Gilbert, Tom Becnel
//
// Inputs: clk, en, ack, weight, bus_program
// Output: req
//
//////////////////////////////////////////////////////////////////////////////////


module neuron_out #(parameter WEIGHT_SZ = 8)
             (clk, en, ack, reset, program_bus, weight, req,
                 LIF_counter);
  input clk, en, ack ,reset;
  input [1:0] program_bus;
  input signed [WEIGHT_SZ-1:0] weight;
  output        req;
  output [WEIGHT_SZ-1: 0] LIF_counter;

//connect en_reg, accumulator, and comm_block modules
   wire            reg_reset;
   wire signed [WEIGHT_SZ-1:0] Q;
   wire            spike;

   en_reg #(WEIGHT_SZ)
   en_reg0(clk, en, reg_reset, weight, Q);

   accumulator_out #(WEIGHT_SZ)
   acc0(clk, reset, program_bus, Q, reg_reset, spike, LIF_counter);

   comm_block comm0(clk, reset, spike, ack, req);

endmodule // neuron


//////////////////////////////////////////////////////
//
```

```
// module: accumulator
//
// inputs: clk, reset, program_bus, weight
// outputs: reg_reset, spike
// this module contains the accumulator which has the current neuron
    voltage level stored
// if a weight appears on the enabled register it adds that weight,
    otherwise it subtracts the leak
// if the counter goes over a set threshold value then a spike is fired
    and the accumulator is reset to 0
//
////////////////////////////////////////////////////////
module accumulator_out #(parameter WEIGHT_SZ = 8)
                  (clk, reset, program_bus, weight, reg_reset, spike,
                      counter);
   input clk, reset;
   input [1:0] program_bus;
   input signed [WEIGHT_SZ-1:0] weight;
   output reg    reg_reset, spike;
   output reg signed [WEIGHT_SZ-1:0] counter;

   //counter reg
   //parameter COUNTER_SZ = 8;

   reg [WEIGHT_SZ-1:0] FIRE_POTEN, LEAK;

//   //fire potential parameter
//   parameter FIRE_POTENT = 63;
//   parameter LEAK = 2;
// // eventually this should be an input!!!!!!!!


   always@(posedge clk) begin
    reg_reset <= 1;
    if(reset) begin
      counter <= 0;
      reg_reset <= 0;
    end
    else
       case(program_bus)
       2'b00:
         if ($signed(counter) >= $signed(FIRE_POTEN)) begin
            counter <= 0;
         end
         else if(weight != 0) begin
            counter <= counter + weight;
            reg_reset <= 0; //reset reg
         end
         else if($signed(counter) > 0) begin
            counter <= counter - LEAK;
```

```
        end // else: !if(weight != 0)
        else if ($signed(counter) < 0) begin
           counter <= counter + LEAK;
        end

      2'b01: //bus value sets fire potential
      begin
        counter <= 0;
        FIRE_POTEN <= weight;
      end

      2'b10: //bus value sets leak
      begin
        counter <= 0;
        LEAK <= weight;
      end

      default:
      begin
        counter <= 0;
        FIRE_POTEN <= 60;
        LEAK <= 2;
      end
      endcase
   end // always@ (posedge clk)

   always @(counter, reset) begin
      if (reset) begin
        spike = 0;
      end
      else if ($signed(counter) >= $signed(FIRE_POTEN)) begin
        spike = 1;
      end
      else  begin
        spike = 0;
      end
   end

endmodule // accumulator
```

6.4 Neural Network Testbench

```
`timescale 1ns / 1ps

//////////////////////////////////////////////////////////////////////////////////
// Company:
// Engineer:
//
```

```
// Create Date: 17:38:06 12/03/2015
// Design Name: top_module
// Module Name:
    Z:/Users/Andy/Documents/School/2015F/VLSI/Project/Verilog//top_module_tb.v
// Project Name: NeuralNet1
// Target Device:
// Tool versions:
// Description:
//
// Verilog Test Fixture created by ISE for module: top_module
//
// Dependencies:
//
// Revision:
// Revision 0.01 - File Created
// Additional Comments:
//
////////////////////////////////////////////////////////////////////////////////

module top_module_tb;

   // Inputs
   reg clk;
   reg rst;
   reg [7:0] input_weight;
   reg [1:0] input_pr;
   reg [7:0] EPROM_data;

   // Outputs
   wire [7:0] output_stub;
   wire [14:0] EPROM_adr;
   wire OE;

   // Instantiate the Unit Under Test (UUT)
   neural_network uut (
      .clk(clk),
      .rst(rst),
      .input_weight(input_weight),
      .input_pr(input_pr),
      .EPROM_data(EPROM_data),
      .output_stub(output_stub),
      .EPROM_adr(EPROM_adr),
      .OE(OE)
   );

   integer i;
   reg [7:0] input_levels [32767:0];
   reg [7:0] EPROM [32767:0];

   initial $readmemh("memory.dat", input_levels);
```

```
  initial $readmemh("memory.dat", EPROM);
  initial begin
    // File
    //Out = $fopen("neural_outpur_vlsi.txt", "w");
    // Initialize Inputs
    clk = 0;
    rst = 0;
    input_weight = 0;


    // Wait 100 ns for global reset to finish
    #100;
    rst = 1;
    # 50
    rst = 0;
    #20
    input_pr = 2'b01;
    input_weight = 8'b01111010;
    #50
    input_pr = 2'b10;
    input_weight = 8'b00000010;
    #50
    input_pr = 2'b00;
    #50
    rst = 1;
    #100
    rst = 0;
    #50
    // Add stimulus here

    for(i=0; i < 32000; i=i+1) begin
      input_weight = input_levels[i];
      #50;

    end


  end

  always #5 clk = ~clk;

  // EPROM
  always @(negedge clk)
    if (~OE)   EPROM_data <= EPROM[EPROM_adr];

endmodule
```

7 Test Results

7.1 Verilog Testbench

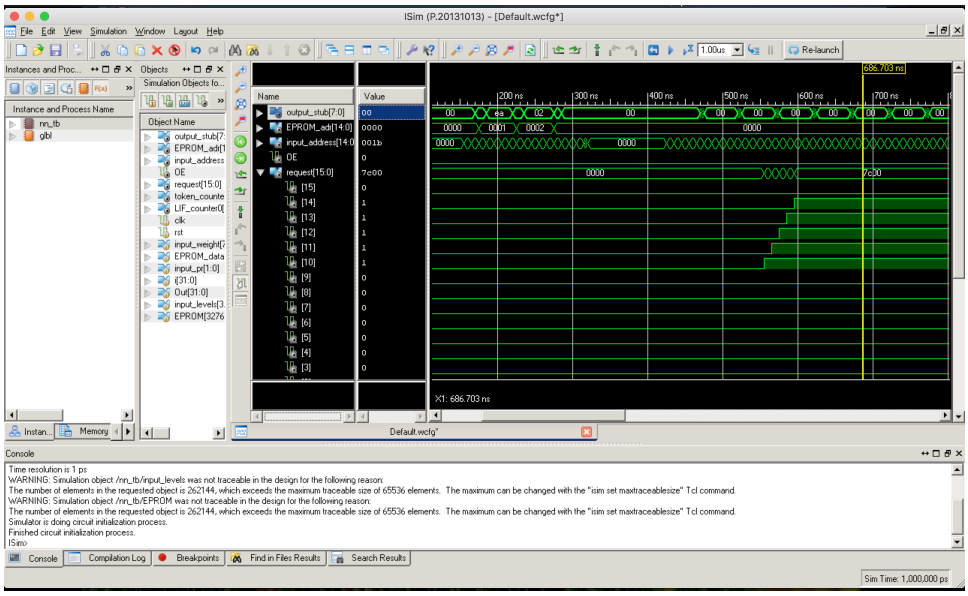

Figure 8: Testing Waveform 1: Request lines

Here is a series of waveforms which documents the working functionality of our chip. Fig. 8 shows a series of neurons spiking. This can be seen in the request lines which sequentially go high. Note that this sequential behavior is not caused by the neurons spiking, but rather by a single input affecting each neuron. Since the time is only a few clock cycles off, biological systems would perceive all of these as happening a the same time, and theoretically our hardware neurons should perceive it the same way.

Fig. 9 shows the proces of sending data out to the neurons. The token variable indicates which neuron is currently being checked for a spike (i.e. sending neuron). If there is no spike, as can be seen when token is at 9, then the arbiter still cycles through the receiving neurons (the current receiving neuron is contained in the nrn_receive variable) but in this case the data on the bus comes exclusively from the input line. This can be seen the EPROM address, which tells the memory where to fetch the next weight from, sits at 0 (which will always have a 0 stored inside of it). As soon as the token has finished processing neuron 9 it moves to neuron 10, which did have its request line high. Here it is immediately clear that the arbiter begins sending out the combined input and spike. This can be seen in the EPROM address as well as the output. Note that the change in the output in the latter half of clock cycles is just a product of the way the testbench EPROM was set up and should not affect actual operation.

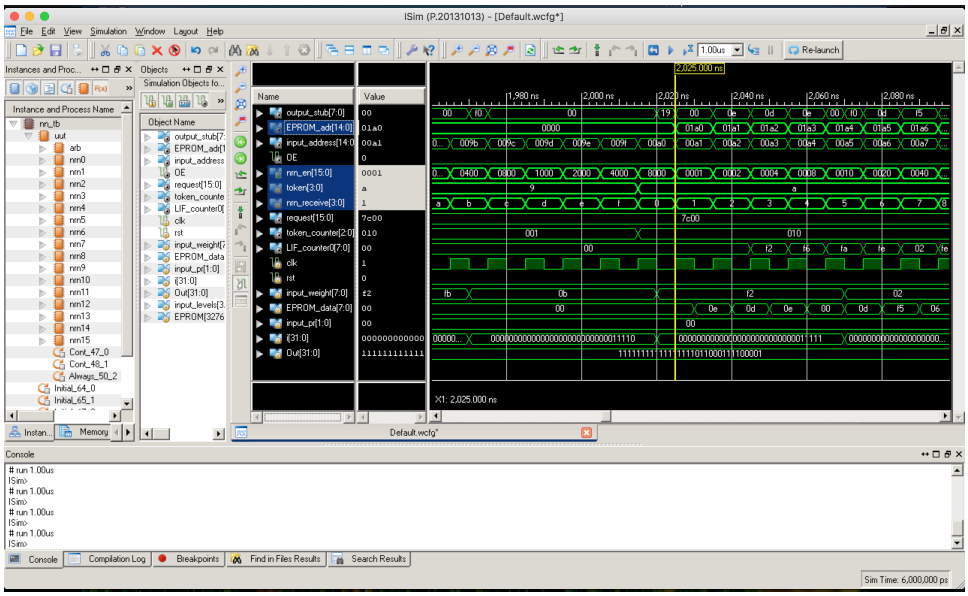

Figure 9: Testing Waveform 2: Receiving Neuron Cycle

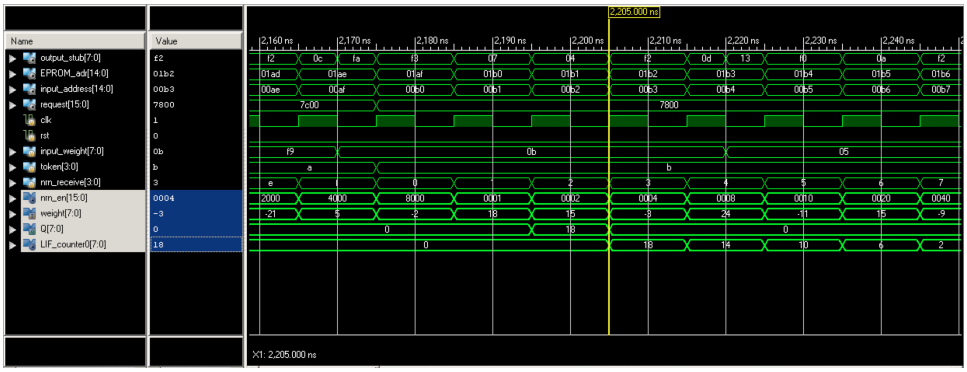

Figure 10: Testing Waveform 3: Neuron operation

Fig. 10 shows neuron function. This image is targeting the operation of neuron 0. In this case the network is attempting to send a weight of 13 (0xd) to the neuron. It takes several clock cycles for this signal to propagate through the neurons accumulator but it does get there. The following is the operations performed in each clock cycle:

1. In the first clock cycle the nrn_receive is set to 0.
2. nrn_en is one clock cycle behind the nrn_receive counter so nrn_en[0] is set high and neuron 0 the correct weight is put on the bus

3. At the next posedge Neuron 0 sees the bus and pulls the output into its enabled register.

4. The accumulator reads the enabled register, determines that it is non-zero, and adds it to the current potential. It sends a reset signal to the enabled register. The weight is added on to the current value in the accumulator

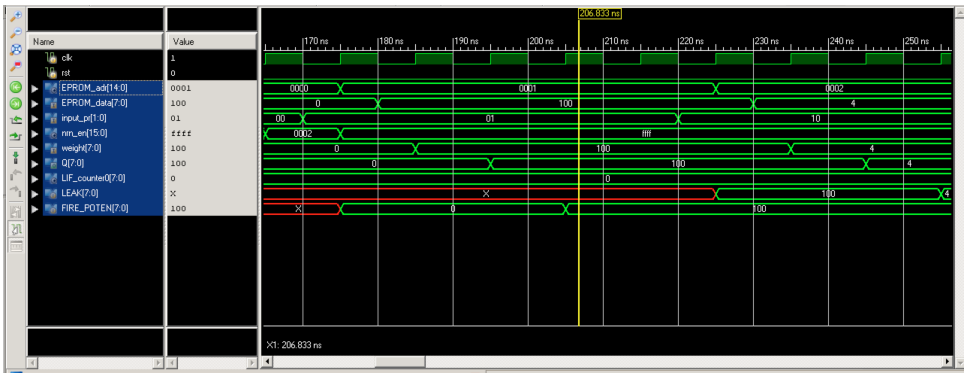

Figure 11: Testing Waveform 4: Setting thresholds

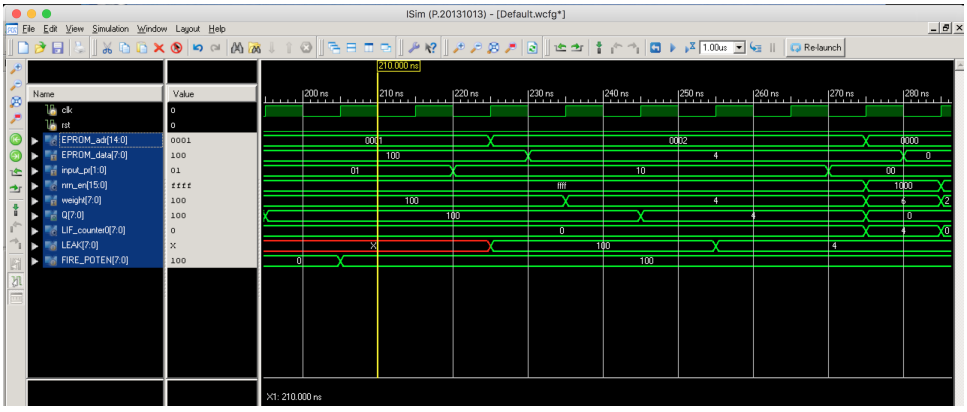

Figure 12: Testing Waveform 5: Setting leaks

Fig. 11 and Fig. 12 show the process of setting a leak and threshold value for the neurons. In Fig. 11 the threshold is set. The threshold is initially not set (although it can be set to a default value by simply setting the input_pr to 2'b11). Then the input_pr is set to 2'b01 and the EPROM address is set to 'b1 (where the threshold must always be stored). The value is loaded into the EPROM and placed on the bus (when the input_pr is not 2'b00 the input is ignored). All of the neuron enable signals are set high so each neuron loads the

value in and updates its threshold register. A similar process is performed with the leak in Fig. 12 although in this case input_pr is 2'b10.

7.2 Python Module

An idealized version of the code was also implemented in a Python module which is shown below. This code was implemented using Python 3.4. The code requires Numpy to run. The code (along with the verilog module) also require several text files which serve as the EPROM for the weights and the input. Address 0 of the EPROM should always be a 0. Address 1 should be the programmed threshold value and Address 2 should be the programmed leak value. The actual weights run from 0x100 to 0x1FF. The input reads sequentially from address 0 until it overflows and starts at 0 again. For the initial comparison between the Python module and Verilog modules random values were used for the input and the weights to minimize any user error. Therefore the same file was loaded in the verilog testbench for both. The same data was used for the Python module but the files had to be stored in different formats because the weights were input into a matrix so separate files were used.

The code was first used for validation that our Verilog correctly. We validated that the verilog was indeed correct by comparing the outputs of the two designs. The Verilog code successfully replicated the neural behavior (spikes produced in the same neurons at the same time). This code will also be used to correctly set the weights of the chip as it is much faster to try a different weights in software rather than downloading them to an EPROM each time. Currently this setting process works through a genetic algorithm. The algorithm has a variable number of iterations and test tries but essentially goes through and adjusts each non-zero weight by a random amount in each try (non-zero so that the user can set up layers of neurons and these layers will not be modified by the algorithm). The algorithm takes the winner of each iteration by which result was closes to the desired output and uses this winner as the base for the next iteration of tries.

The python module is shown below (with just the validation code and not the genetic algorithm iterative module).

```
__author__ = 'Andy'
import numpy as np


class LIF_match(object):
    """
    Leaky integrate and fire neuron
    ID is neuron id
    potential, weights and spikes are pointers to common numpy arrays
        for all neurons
    """

    def __init__(self, ID, reset, threshold, leak, potential, weights,
```

```
        spikes):
        self.ID = ID
        self.potential = reset
        self.potential = reset
        self.reset = reset
        self.threshold = threshold
        self.leak = leak
        self.time = 1
        self.weights = weights
        self.spikes = spikes
        self.request = 0

    def update_potential(self, input, t):
        if self.potential >= self.threshold:
            self.potential = self.reset
            self.request = 1
            print('===============SPIKE SET nrn: {0}, time:
                {1}'.format(self.ID, t))
            return 1
        elif input != 0:
            self.potential = self.potential + input
            return 0
        elif self.potential < 0:
            self.potential += self.leak # leak is reversible for
                inhibitory synapse
            return 0
        elif self.potential > 0:
            self.potential -= self.leak
            return 0
        else:
            return 0

    def acknowledge(self):
        self.request = 0
    def get_request(self):
        return self.request
    def get_potential(self):
        return self.potential

    def time_step(self):
        self.time += 1

    def reset_time(self):
        self.time = 0




# Set up parameters of function run
```

```
num_neurons = 16
t_end = 1000

input_data = np.loadtxt('./input.txt')
weights = np.loadtxt('./weights.txt')
def input_address(receiving_nrn, sending_nrn, input_data):
    sn = hex(sending_nrn).split('x')[1]
    rn = hex(receiving_nrn).split('x')[1]
    adr = int(str(1) + str(sn) + str(rn), 16)
    return input_data[adr]



# Neuron params
reset = 0
threshold = 100
leak = 4
# refractory = 2
# if middle_neurons == 0:
#     t_add = 1
# else:
# t_add = 2


# Set up script variables
#=========================
spikes = np.zeros([t_end, num_neurons])
potential = np.zeros([t_end, num_neurons+1])

neurons = []
for i in range(num_neurons):
    neurons.append(LIF_match(ID=i, reset=reset, threshold=threshold,
        leak=leak, potential=potential, weights=weights, spikes=spikes))

t = 0
token = 0 # starts by adding one so start at negative 1
output = list()
io = list()
tc = list()
rc = list()
nc0 = list()
receiving_neuron = 0
input_t = 0
spike = 0
while (t < t_end):
    input = np.zeros(num_neurons)
    if spike: # if current neuron spiked
        # print('spike', token, t)
        input[receiving_neuron] += input_address(receiving_neuron,
            token, input_data)
```

```
        output.append(weights[token, receiving_neuron])
    else:
        output.append(0)
    input[receiving_neuron] += input_data[input_t]
    for nrn_num in range(num_neurons):
        neurons[nrn_num].update_potential(input[nrn_num],t)
    # increment counter 1
    receiving_neuron += 1
    if receiving_neuron == num_neurons:
        if spike:
            neurons[token].acknowledge()
        token += 1 # increment sender counter
        if token == num_neurons:
            token = 0
        if neurons[token].get_request():
            spike = 1
            print('processed {0} high -------------\nt=
                {1}'.format(token, t))
        else:
            spike = 0
        receiving_neuron = 0
        # print('MOVING TO NEURON: ', token)
    io.append(input_data[input_t])
    tc.append(token)
    rc.append(receiving_neuron)
    nc0.append(neurons[0].get_potential())
    t += 1
    if divmod(t, 5)[1] == 0 and t >= 10:
        input_t += 1

np.savetxt('./weights_out.txt', weights, fmt='%d', delimiter='\t')
output = np.array(output)
io = np.array(io)
tc = np.array(tc)
rc = np.array(rc)
ot = np.zeros([t_end, 6])
nc0 = np.array(nc0)
ot[:, 0] = np.arange(t_end)
ot[:, 1] = output
ot[:, 2] = io
ot[:, 3] = tc
ot[:, 4] = rc
ot[:, 5] = nc0
np.savetxt('./output.txt', ot, fmt='%d')
```

8 Project Directories

Lib6710_01: /home/becnel/VLSI/cadence_f15/Lib6710_01

Neural Network project: /home/becnel/VLSI/cadence_f15/Neural_Network

Project verilog: /home/becnel/VLSI/cadence_f15/neural_network_project/verilog

Project synthesis results: /home/becnel/VLSI/cadence_f15/neural_network_project/synth_results

Project EDI results: /home/becnel/IC_CAD/edi/neural_network