Spiking Neural Network Accelerator

Contents

1.Introduction

Learning from biological systems, Spiking Neural Networks have great power optimization in computing devices.However, with the large amount of input data, how

to optimize the running time of the system has become an urgent problem to be solved[1].

This design is to build an asynchronous SNN accelerator using an array of processing elements described using SystemVerilogCSP. The project will thus include aspects of architecture, micro-architecture, circuit design, modeling, and verification.

In our project, we adopt 2D-Mesh topology to build the NOC structure with 9 routers in total using the XY routing algorithm. Inside the NOC, we take 3 decentralized PEs and 3 partial sum adders to process this calculation.

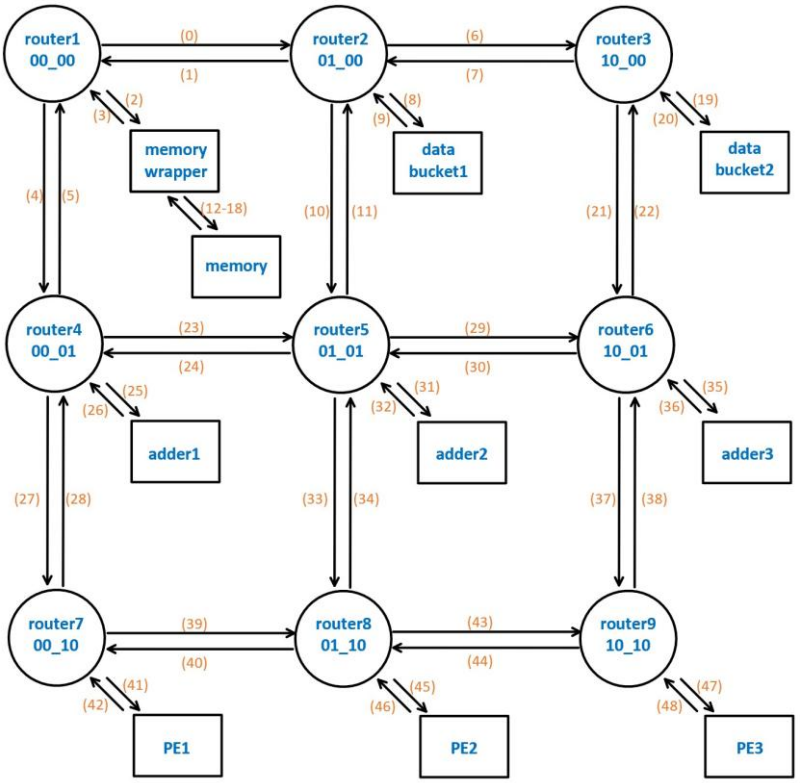


Figure 1. 2D Mesh Topology Structure

2.File and Version Management

In the previous project or collaborative work, sharing files between group members is less efficient. Regarding this project, we use Github as our online repository to store, share, and trace our files.

In the main branch, we upload all the project system verilog files. When someone wants to get a clone of current project files, it will be very easy for them to download directly from Github instead of using email or other software to share files. Apart from the main branch, some other sub-branches are also used. For example, we create a test branch to store some test files.

Another significant advantage is that Github provides us with a function that we can clearly see which lines of this upload have been modified compared to the previous one. It is not uncommon that we frequently change some lines of the file to achieve a function. Sometimes the new lines do not work properly. This is the time we want to trace back and reuse some lines from the old version.

3.Dataflow

The data flow of the entire project is described as follows. In the whole system, we have 3 PEs(PE1,PE2,PE3) and 3 Partial Sum Adders(Adder1, Adder2, Adder3). For the 1st input map, memory each time sends one row of input map (or one row of kernel) to PE.

After receiving all the values, PE can start calculating. For the first round, all the PEs send their results to Adder1. For the second round, all the PEs send their results to Adder2. For the third round all the PEs send their results to Adder3. After calculating, Adder will send out the membrane potential to memory.

And after simple comparison in Adder, only when output spike is 1, Adder will send a packet (contains the row address and column address of the output spike) to memory. After Adder 3 sends out results, Adder 3 will also send a done signal(the row address and column address are both 11, which is impossible to show up in the 3x3 output matrix) to memory. The done signal tells memory that Adder1, Adder2, Adder3 all have finished calculating and sending results and it is time for memory to send the next row of input map.

In order to minimize the time consumption and power consumption (switching activity), after Adder3 receives the next row of input map from memory, Adder3 will begin local sharing. First, Adder3 will send its old input map row to Adder2. Second, after Adder receives an old input map row from Adder3, Adder2 will send its old input map row to Adder1. After local sharing, Adders will do the same things as stated before.

For the second input map and beyond, not only will the Adder receive the result from PEs, but also the memory will send old membrane potential (calculated in the last input map) to Adder. Then Adder will add these four values together to generate results[2].

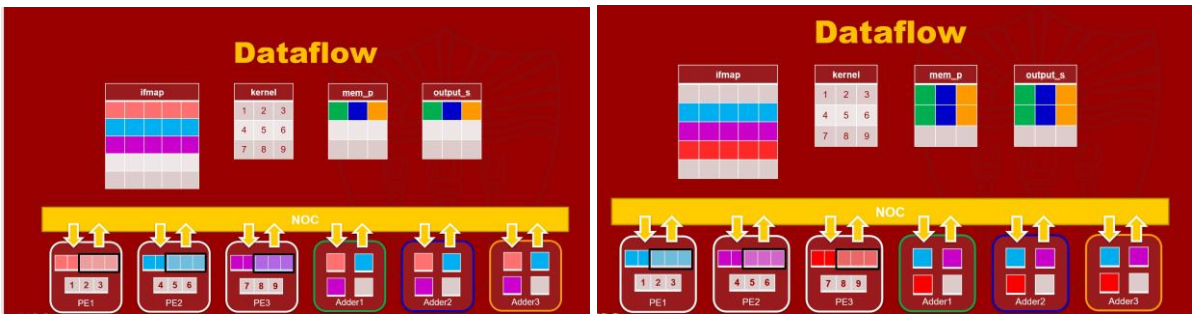


Figure 3. Input Map Calculation

4.Mesh topology NOC structure

4.1Top-Level view

In our design, we use the 2D- Mesh topology with 9 routers. There are 3 PEs, 3 adders, 2 data buckets and a memory wrapper. Two data buckets here are to fill the blank router and to check if the packets passing through them are correct or not[3].

4.2 XY Routing Algorithm

We use the XY routing algorithm to describe the address of each router. We use a 4-bit address to represent the XY coordinates. The first 2 bits represent the X coordinate and the last 2 bits represent the Y coordinate. For example, for the router connected to PE1, X coordinate is 00 and Y coordinate is 10. In XY routing, the comparison of the X coordinates values of the current router is done with the destination X coordinate. Based on the comparison the packet is transferred in the eastward or westward direction ports. Once the X coordinates of the current router and destination address are the same, then the comparison base on the Y coordinates of the router is done. This comparison will route the packet to the north, south, or the local port of the router.

XY algorithm could reduce average latency per packet and increase average throughput. Furthermore, XY routing defines that all transfers in the mesh structure should go in a fixed order. The algorithm first directs the packet in the X direction, then in Y direction so that it could avoid deadlock in transferring packets.

4.3 Router

Each router in the mesh structure has 5 ports which contain 10 channels in total. Four ports are connected to other routers and one port is a local port that is connected to the processing element. In this way, the next state of the system can be any one of the five ports east, west, north, and south, or the local port.

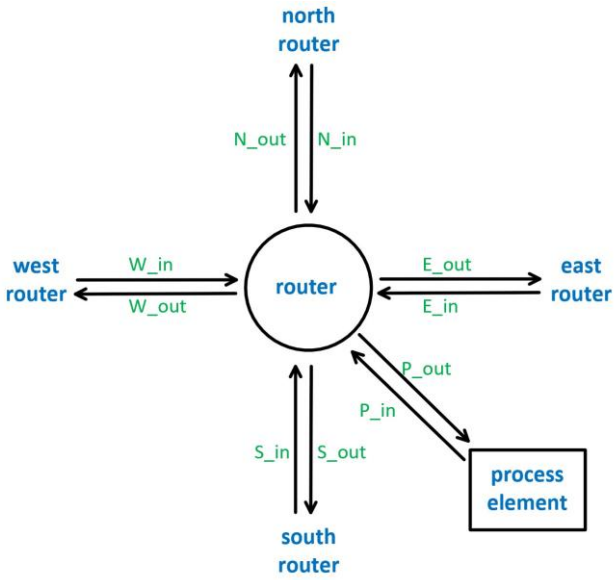


Figure 4. Router Design

4.4 Arbitered Merge

For each router, considering it connects to 5 ports, we should guarantee all input channels have an equal opportunity to get to the router. There is the possible situation that two or more packets come from different directions simultaneously and they would like to go out in the same direction. This conflict will cause deadlock and blocking inside the router. So we use an arbiter to prevent this circumstance.

Because the router has 5 directions, we should at least build a 4 input arbiter. In this way, we adopt a pipeline tree arbiter design. We decompose this arbiter into two blocks: an arbitered merge and packets transferring block. The mission of the arbitered merge is to make the decision of which packet could pass first. The mission of the packets transferring block is to pass these concurrently coming packets in the sequence decided by the arbitered merge.

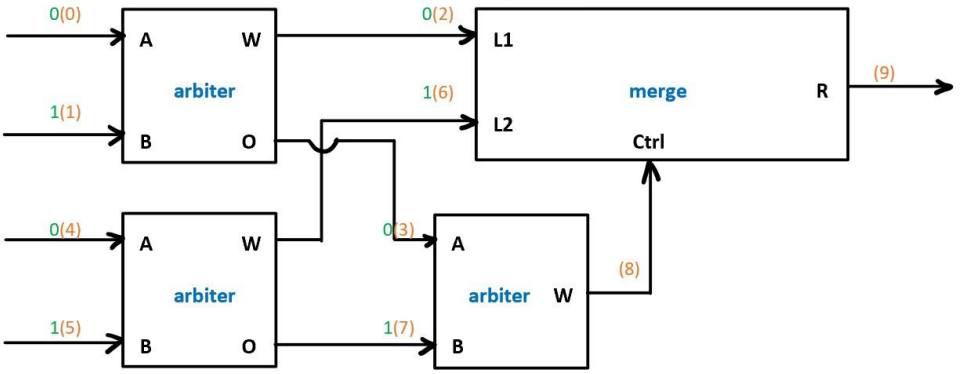


Figure 5. Arbitered Merge

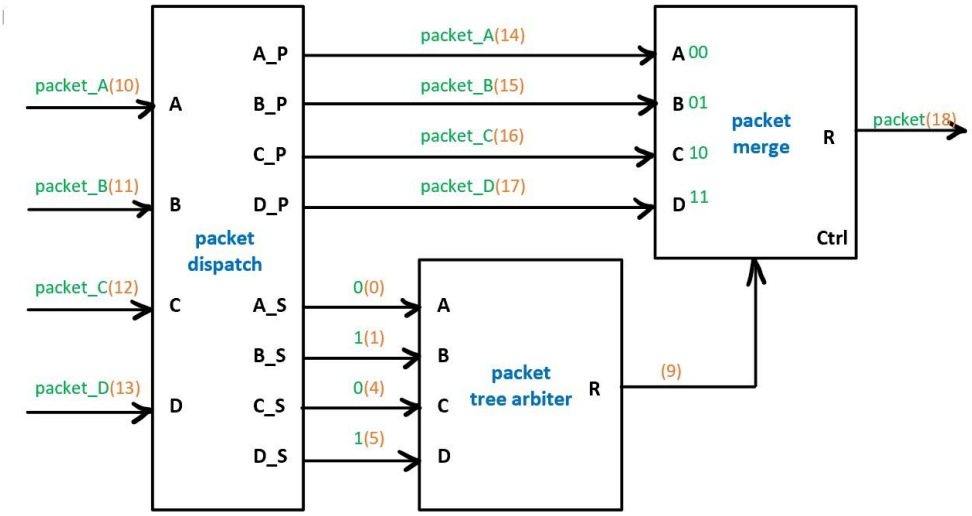


Figure 6. Packets Transferring Block

Additionally, to send the concurrently coming packets into the corresponding arbiter, we also adopt a packet analyzer to read the destination directly and 5 splits to send these packets into the corresponding arbiter. (The orange labels are the channels connected in between.)

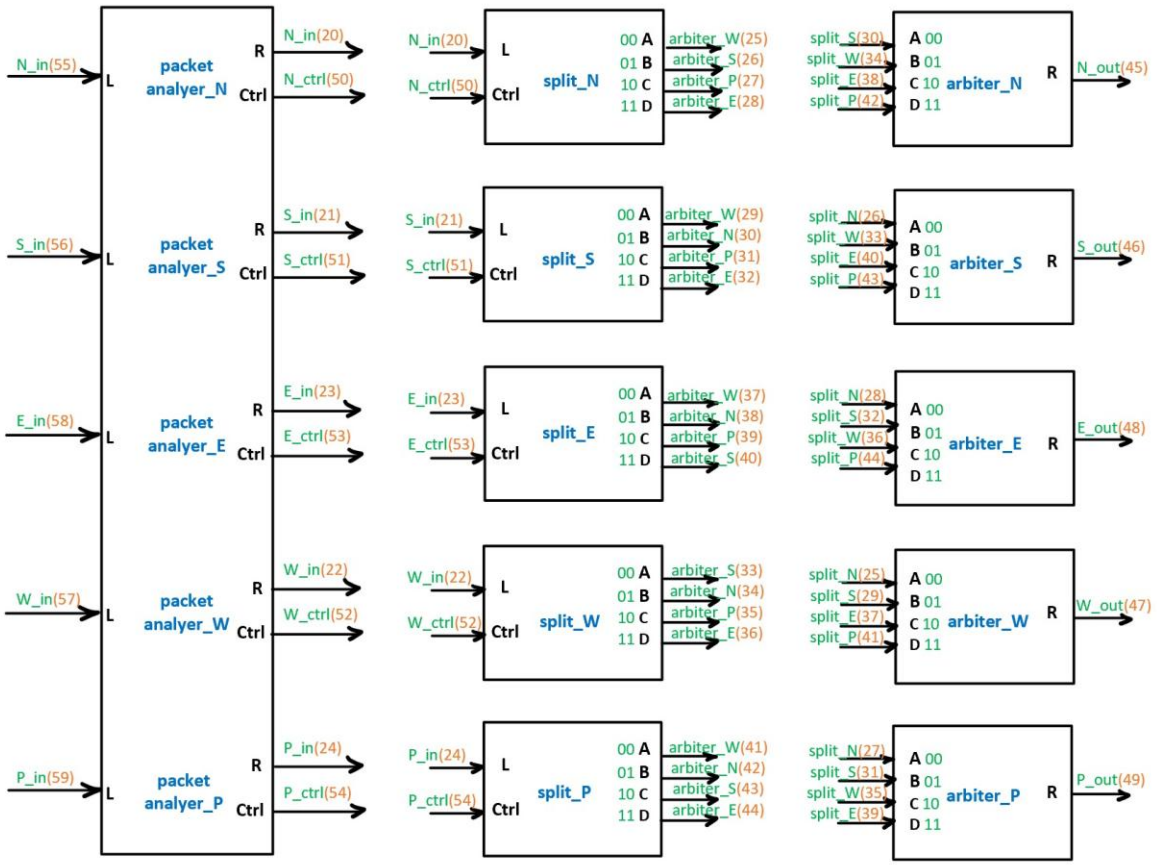


Figure 7. Whole Design of Arbiter

5.Packet Format

We use only **one** packet format to match different interfaces:

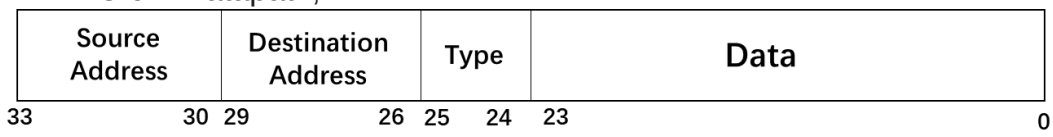
Unified format: 34bits in total:

33-30: Source address;

29-26: Destination address;

25-24: Type of the datapath (From 00 to 11);

23-0: Datapath;



Bit 33-26 indicates the address of the router. In our design, we employ 9 routers in total and put them into a 3×3 framework. So the number of both columns and rows is 3. The X and Y coordinate should be represented by 2 bit at least. In this way, the address of each router is 4 bits.

Bit 25-24 indicates the type of the data we are transferring. There are 6 differential data types during this calculation process. They are: a. the input sipke value transferred from the memory wrapper to the PEs; b. the kernel value transferred from the memory

wrapper to the PEs; c. the result calculated by PEs that should be transferred from PEs to adders; d. the pervious membrane potential value stored in the memory should be transferred from the memory wrapper to the adders; e. the membrane potential value calculated by adders should be transferred from adders to memory wrapper; f. the output sipke value calculated by adders transferred from adders to memory wrapper. We could simplify these 6 types into 4 by combining the same length of the datapath. So, the type index could be counted from 00 to 11.

Bit 24-0 indicates the datapath. This length is mainly determined by the kernel value. For shortening latency, we would like to transfer the one row of datas in the kernel map at one time, which has 3 values and each data occupies 8 bits.

Besides, the packet of the output sipke is a little bit different from the format. For the last 24 bits datapath position, we write the address of the output sipke instead of the value. As we mentioned above, after we calculated the output sipke, we found that most values inside this table are ZERO. For reducing latency and improving the performance, we only send the output sipke with value equal to ONE. In this way, we should tell where these ONEs are located in the matrix. We use 4 bits to tell the address of the output sipke metrix. The first two bits indicate the row of the matrix and the last to bit indicates the column of the matrix.

a. Input Spike from Wrapper to PEs:

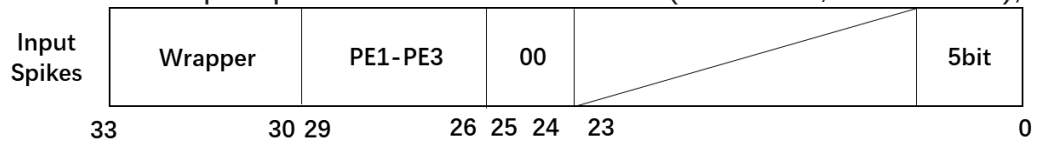
33-30: Wrapper address (00_00);

29-26: PE1-PE3 address (00_10, 01_10, 10_10);

25-24: 00 (indicate input spike);

23-5: all zeros;

4-0: 5 input spike value of the whole row (1-bit each, 5-bit in total);



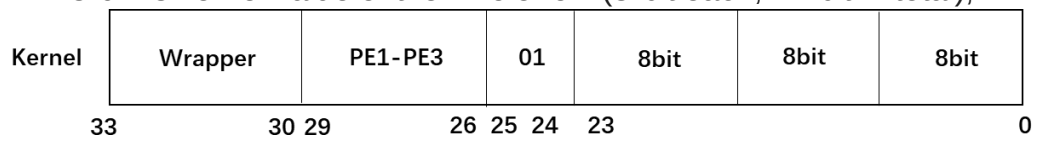
b. Kernel from Wrapper to PEs:

33-30: Wrapper address (00_00);

29-26: PE1-PE3 address (00_10, 01_10, 10_10);

25-24: 01 (indicate kernel);

23-0: 3 kernel value of the whole row (8-bit each, 24-bit in total);



c. PE Calculated Result from PEs to Adders:

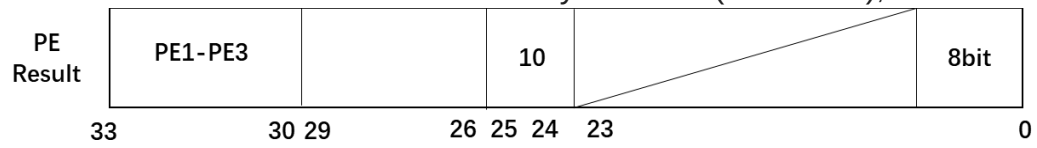
33-30: PE1-PE3 address (00_10, 01_10, 10_10);

29-26: Adder1-Adder3 address (00_01, 01_01, 10_01);

25-24: 10 (indicate membrane potential);

23-8: all zeros;

7-0: each value after calculated by PE once (8-bit each);



d. Previous Membrane Potential from Wrapper to Adders:

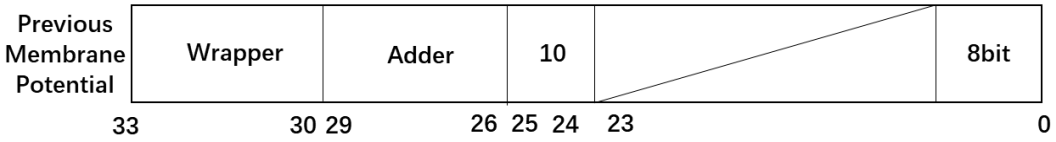
33-30: Wrapper address (00_00);

29-26: Adder1-Adder3 address (00_01, 01_01, 10_01);

25-24: 10 (indicate membrane potential);

23-8: all zeros;

7-0: the membrane potential value from the previous time step (8-bit each);



e. Membrane Potential from Adders to Wrapper:

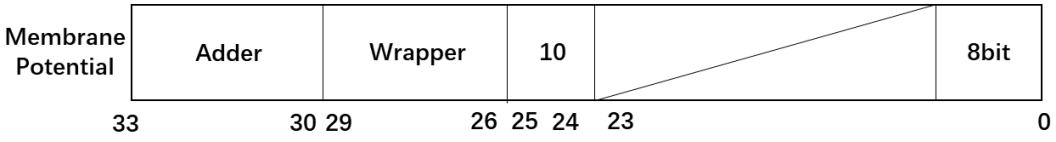
33-30: Adder1-Adder3 address (00_01, 01_01, 10_01);

29-26: Wrapper address (00_00);

25-24: 10 (indicate membrane potential);

23-8: all zeros;

7-0: each membrane potential value calculated by adders (8-bit each);



f. Output Spike from Adders to Wrapper :

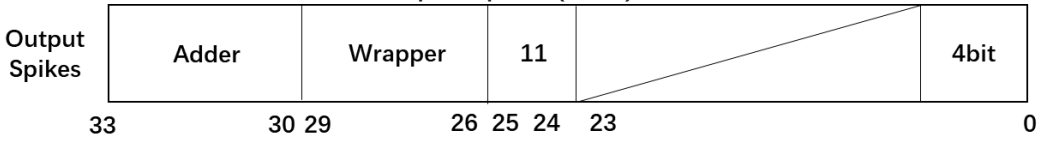
33-30: Adder1-Adder3 address (00_01, 01_01, 10_01);

29-26: Wrapper address (00_00);

25-24: 11 (indicate output spike);

23-10: all zeros;

3-0: address of one output spike (4 bit);



6.Memory and Memory Wrapper Module:

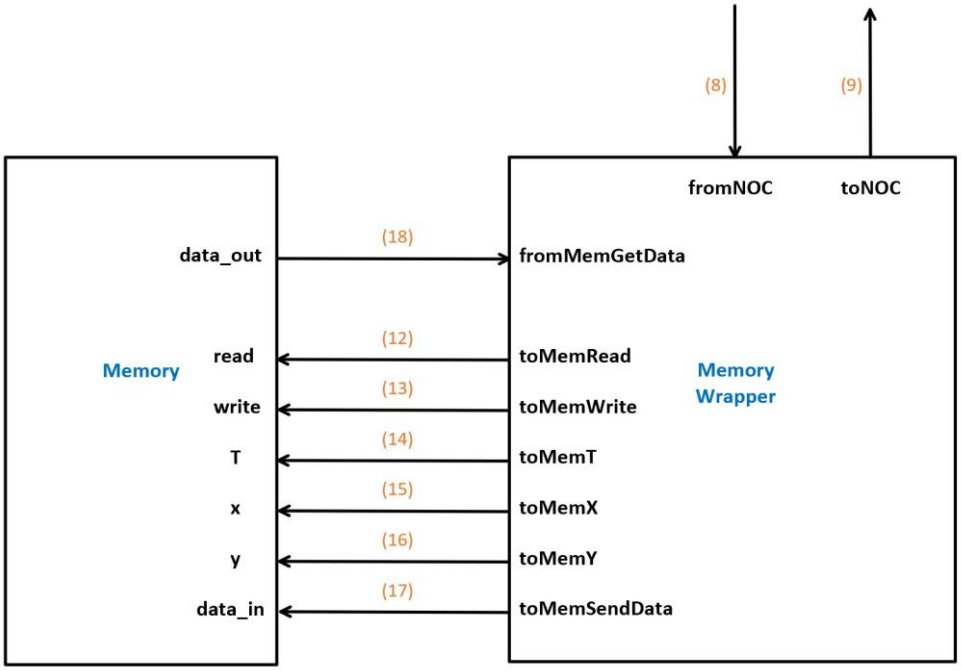


Figure 8. Memory and Memory Wrapper

For our memory and memory wrapper modules, we make lots of modifications on them. Memory wrapper modules can send proper old membrane potential to adders and send proper input spike value to PE3 at exact time. Moreover, memory can only receive “1” for output spikes.

For receiving membrane potential and output spike value, we set to judge whether the value from the router is DONE signal (4’b1111) with input spike type. If it is not, then it will send membrane potential or output spike value to memory through different data types. memory will let the relevant position to be 1, because adder only sends the address of output spike for “1” to wrapper and wrapper sends it to memory. In that way, we need to initialize all output spike values at the beginning of operation.

When the wrapper receives the DONE signal, it sends one row of the last two row input spike values to PE3 to let PE to be local with each other. Moreover, it will also send one row of old membrane potential to each adder. For the first row of old membrane potential, the wrapper will send it at the beginning of the loop. All old membrane potential will be sent since the second timestep.

To better control sending old membrane potential and input spike and receiving value, we add a variable “flag” to mark each time the wrapper receives a DONE signal. We need to send two rows of old membrane potential and two rows of input spike values

after receiving the first two DONE signals. Moreover, we let the “DONE” signal and output spike value not take over the times of loop. We also don’t want the last times of receiving membrane potential to take over the time of loop to receive the last output spike and the final DONE signal.

7.Processing Element (PE) Module

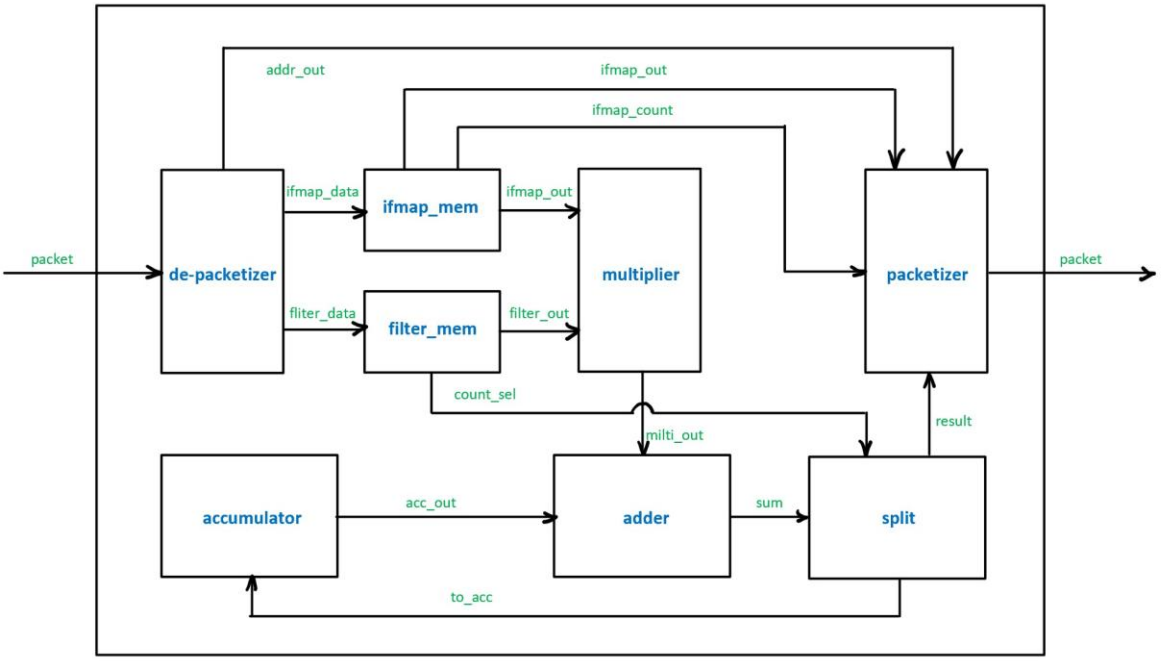


Figure 9. Processing Element

The PE module in our design is to compute a dot product, using one row of “ifmap_mem”(input spike) and one row of “filter_mem”(kernel). It will produce a partial sum and send this sum to the adder module to generate the output feature map. Our PE module can provide local storage, continuing to send filter value, and transfer data to other PEs. Moreover, our PE uses decentralized design.

The PE uses a depacketizer to analyze incoming packets with data type and send proper data to two memories. The length of each data is a function of the size of the SNN layer. For filter memory, it will receive one row of kernel data, so it is 24 bits. For ifmap memory, it will receive one row of input spike data. Due to the different speeds of calculation for different PEs, we decide to choose 3 adders to control each PE calculation, and each adder can generate one result for one membrane potential or output spike result of one row. Therefore, each PE should send one of three results to each adder. Due to there being only one time for sending data to filter memory of each PE, it needs to make the filter continue to send data to the multiplier. Thus, I utilize an infinite while loop to control this process. The multiplier will wait until the ifmap value arrives.

We find that some rows of the input feature map will be used two or more times, but if we regularly transfer the next row to ifmap after the one-row calculation has been

finished, which will be a minor inefficiency. Thus, we design that PEs can transfer their ifmap data to each other after the calculation has been finished. To let packetizer know when it should send proper ifmap value to the other PE, we add a variable “flag” in ifmap memory to mark how many times the new ifmap value arrives in ifmap memory. When it is bigger than 1, it shows that ifmap memory has received the next ifmap value and it will send the old ifmap value to the packetizer. Moreover, ifmap memory will restore the old ifmap value after sending all ifmap values to the multiplier. Through address and value transferred from depacketizer and ifmap, packetizer can send a proper packet to the correct PE.

We eliminate the control block in this PE to reduce the delays and wires necessary for communication within the PE block. For the original control block, we decentralized it and each block is responsible for slightly more. For the split block, it should know when to send the sum to the packetizer or send the sum to the accumulator, so we let filter memory send the times of the outside loop to tell it when the adder is calculating the last member. We also make ifmap memory send the times of the outside loop to let the packetizer know which adder it should send data to. We believe this design can help to facilitate higher throughput while reducing the complexity of the PE.

8.Partial Sum Adder

8.1 Partial Sum Adder

The partial sum adder mainly uses an input channel to receive value, for example PE output value or membrane potential. After receiving all the required values, the partial sum adder adds up the four values(PE1,PE2,PE3,Membrane_potential) to output a new Membrane_potential and an output_spike. For the 1st input map, the Partial Sum Adder will only add up three values and output results[4].

Because of the memory space limit(no more than 8% of Memory) in partial sum adder, we decide to use each 8bit for PE1,PE2,PE3,Membrane_potential in adder(Total 32bits). In each PE, we still give them one row of input spike and one row of kernel value. In this case, after calculating once(123 of the input spike) and sending the partial sum to the adder, each PE will wait for the signal from the adder to tell them to calculate the next value(234 of the input spike). Also because the Membrane_potential only uses 8bit wide memory in adder; except for the 1st input map, everytime when we want to add up, memory will send each old membrane potential out to each adder.

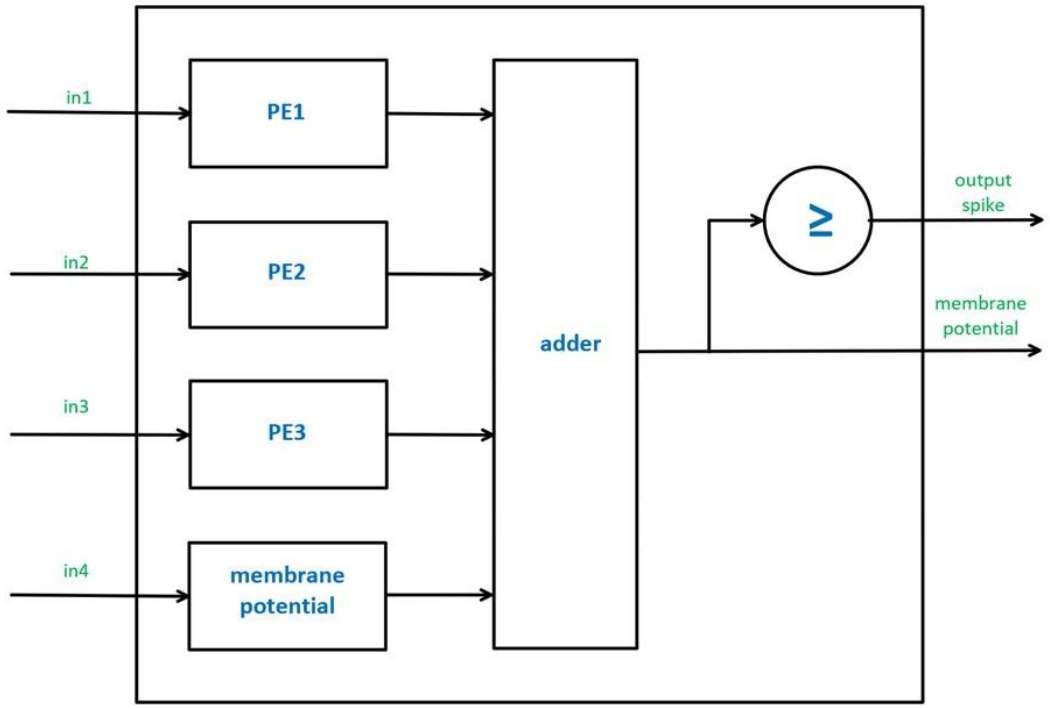


Figure 10. Partial Sum Adder

Also, the comparator block is integrated into the adder. After calculating results, the comparator will compare the membrane potential with the threshold, only when the output spike is one, the adder will send output spike packet.

Regarding the code, the receiving part is simply accomplished by four “in.Receive(value)” sentences. Four if sentences have been used to determine where the result is from, and put them into the right parietal area(partial_PE1, partial_PE2, partial_PE3, membrane_potential). After receiving all the results from PE, Adder can start calculating. Also in order to have different actions in the 1st input map and following input map, a flag named “count” is used to count the running time of each time. Starting at the 1st input map, the count flag is zero. After each calculation, the count flag will increment by one. Because for each input map, each Adder will work for three times, so when the count flag turns into three, it will make the variable first_time equals one(representing right now it is calculating in the 1st input map). Then Adder will receive four values and add up four values.

OUTPUT DUMP FILE:

SNN_Accelerator123.adder3 first receive----
packet:00001001100000000000000000000111111 #
SNN_Accelerator123.adder3 second receive----
packet:00101001100000000000000000000010011

SNN_Accelerator123.adder3 third receive----
packet:01101001100000000000000000000000111

SNN_Accelerator123.adder3 forth receive----
packet:10101001100000000000000000000001010

add_num:SNN_Accelerator123.adder3---Membrane_P after
compare10010000100000000000000000000100011

add_num:SNN_Accelerator123.adder3---output_spike

no_zero_send:10010000110000000000000000000000010,row:00---col:10 #

add_num:SNN_Accelerator123.adder3---Done signal Sent!!!

8.2 Gate Level Partial Sum Adder

Description:
As shown in Figure 11, a gate-level design of parital_sum_adder is provided. This adder is using a 4-phase bundled data micropipeline design and implemented a click control block and a delay unit(in order to make sure when b.Req is about to send, the data out is ready). The delay unit uses “#delay out=in;” statements in SystemVerilog to delay for a period of time. The click control block processes the handshake protocol and generates a clock signal to four D flip-flops.

These four D flip flops receive data respectively from PE1, PE2, PE3, and Membrane potential(from memory), and driven by the clock send the data out to the sum functional block. The sum functional block receives the values then adds them together and sends the result out. This gate-level design also has a reset input, this reset signal connects to both the control block and DFFs.

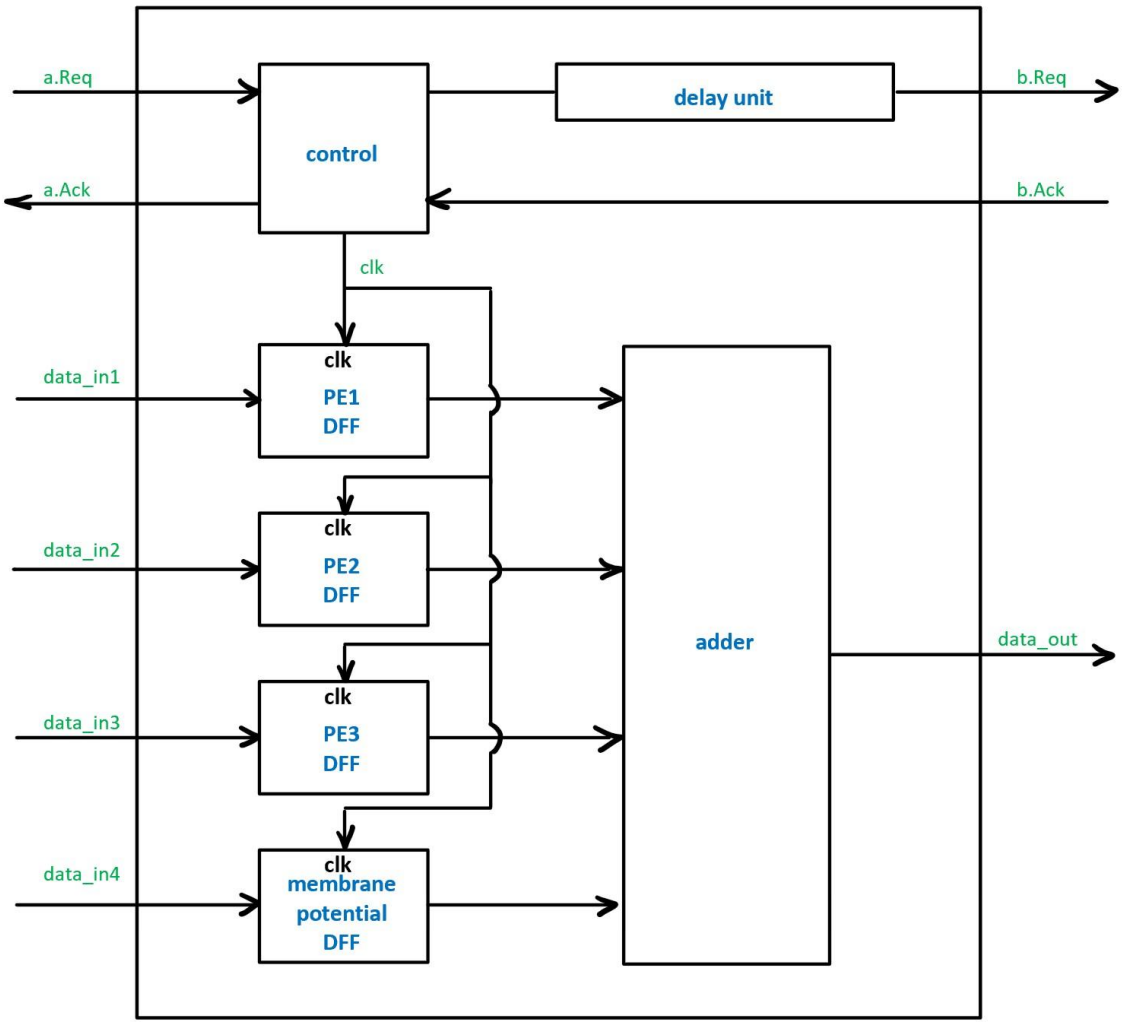


Figure 12. Gate Level Adder

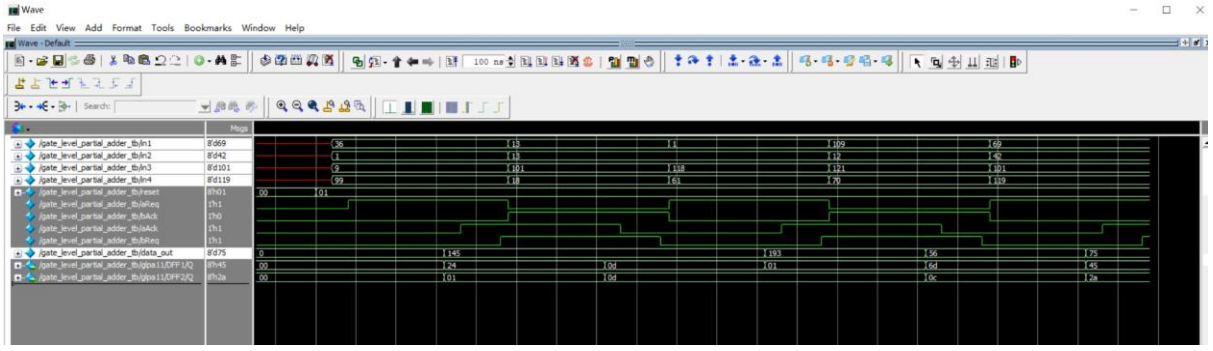


Figure 13. Waveform of Gate Level Adder

9.Analysis

9.1 Performance and Bottleneck

We run our code in regular version and sparse version. Here is the cycle time for each timestep:

Timestep	Regular Cycle Time	Sparse Cycle Time
1	910ns	910ns
2	897ns	876ns
3	883ns	876ns
4	890ns	883ns
5	890ns	876ns
6	897ns	883ns
7	890ns	876ns
8	883ns	876ns
9	890ns	883ns
10	890ns	876ns
Avg	892ns	881.5ns

Table 1. Cycle time of all timesteps

From the table, we can see that the average cycle time for sparse versions is less than regular versions, which means that our send only “1” strategy works. However, whatever version of cycle time, the whole operating time is slightly larger than other similar structure spike neural network accelerators. After analyzing the transcript, we found that PEs will wait for the wrapper to send new input spike value to PE3, then they begin to locally share ifmap value to each other PEs after completing the calculation. Therefore, this waiting time will become one of bottlenecks, which is 153ns.

Additionally, The router design in this project also limits the cycle time in large measure. Adding arbitrary functions in each router to avoid conflict will increase the latency inside the routers. Under this circumstance, once the packet goes through these routers, this latency would sum up and undermine the overall performance, especially for longer paths. The values calculated from PEs and adders may need to wait inside the routers and then cause congestion in the system.

9.2 Enhancement

Compared with the basic tree or ring NOC structure Spiking Neural Network Accelerator, mesh topology can manage a high level of traffic. It can handle a tremendous volume of data since it is possible to connect multiple devices at a time and transmit data simultaneously. It is exceptional when it comes to being resistant to problems. The structure provides its users with a sufficient level of redundancy so that they can keep it running even if some malfunctions occur. If one node goes down, the network has the strength to use the other ones and complete the mesh. Besides, adding mesh topology is easy and goes without any problem. One needs to connect the nodes to the gateways so that the messages can pass through to the remaining network for it to work. It allows the technology to become self-optimized.

Deployed decentralized PE to achieve higher throughput as well as having lower PE area and lower power consumption due to less interaction activities. Regarding dispatch the data in the input map, local sharing tremendously lowers the interaction activities and also speeds up the calculation.

Three Partial Sum Adders have been used, in this way calculation of one row of membrane potential is speeded up. What's more, as for the output spike, adders only send a packet when output spike is one.

Communication (Tree)	# of Routers through	# of packets sent	Total
Mem→PE1	1	4	4
Mem→PE2	3	4	12
Mem→PE3	3	4	12
PE1→Adder	3	9	27
PE2→Adder	3	9	27
PE3→Adder	3	9	27
Adder→Mem	3	18	54
			Total: 163

Table 2. Tree Topology Transmission

Communication (Mesh-Regular)	# of Routers through (avg)	# of packets sent (avg)	Total
Mem→PE1	3	2	4
Mem→PE2	4	2	12
Mem→PE3	5	4	12
PE1→Adder	3	9	27
PE2→Adder	2.67	9	27
PE3→Adder	3	9	27
Adder→Mem	3	17.2	51.6
			Total: 160.6

Table 3. Mesh Topology Transmission with Regular Input

Communication (Mesh-Sparse)	# of Routers through (avg)	# of packets sent (avg)	Total
Mem→PE1	3	2	4
Mem→PE2	4	2	12
Mem→PE3	5	4	12
PE1→Adder	3	9	27
PE2→Adder	2.67	9	27
PE3→Adder	3	9	27
Adder→Mem	3	12.6	37.8

	Total: 146.8

Table 4. Mesh Topology Transmission with Sparse Input

After carefully calculating the total number of transfers, our mesh-sparse design has an overall 10% performance improvement (146.8-163/163)[5].

Another major advantage is the scalability of our design. In a 3x3 output spike matrix, sending one row of output spike maxtrix needs three packets, which may not make a huge difference. If we scale up to a 7x7 or 10x10 output spike matrix, the only send 1's will have more improvement when the input is sparse.

9.3 Future Work

The XY routing algorithm has some limitations. Therefore we can improve routing algorithms by using different algorithms like Level based routing using Dynamic programming.

For waiting for a new ifmap value in PE, which causes a bottleneck in our project, we can seek another advanced way to send the ifmap value to PE to save more time.

Current 2D-mesh NOC structure still has significant delay and redundant connections, regarding this the 3D-mesh structure may bring an obvious improvement[6]. Not only does it have less travel distance between blocks, but also the latency and data calculation is tremendously decreased.

10. Testbench

```
`timescale 1ns/1ps


module tb_snn_network;


    // Parameters
    parameter NUM_INPUT_NEURONS = 2;
    parameter NUM_OUTPUT_NEURONS = 2;
    parameter TIME_STEP = 10;
    parameter SIM_TIME = 500;
```

```
// Testbench signals
reg clk;
reg reset;
reg [NUM_INPUT_NEURONS-1:0] input_spike;
wire [NUM_OUTPUT_NEURONS-1:0] output_spike;

integer t;

// Clock generation
initial begin
    clk = 0;
    forever #(TIME_STEP/2) clk = ~clk;
end

// Reset and input spike generation
initial begin
    reset = 1;
    input_spike = 0;
    #20 reset = 0;

    for (t = 0; t < SIM_TIME; t = t + TIME_STEP) begin
        @(posedge clk);
        if (t % 40 == 0) begin
            input_spike = 2'b01;
        end else if (t % 60 == 0) begin
            input_spike = 2'b10;
        end else begin
            input_spike = 2'b00;
        end
```

```
        end
        $finish;
    end

    // Instantiate SNN
    snn_layer uut (
        .clk(clk),
        .reset(reset),
        .input_spike(input_spike),
        .output_spike(output_spike)
    );

endmodule
module snn_layer (
    input clk,
    input reset,
    input [1:0] input_spike,
    output reg [1:0] output_spike
);

    // Parameters
    real membrane_potential [1:0];
    real threshold = 1.0;
    real decay = 0.9;
    real weight_matrix [1:0][1:0]; // [output][input]

    integer i, j;

    // Initialization
    initial begin
```

```
    weight_matrix[0][0] = 0.6;
    weight_matrix[0][1] = 0.3;
    weight_matrix[1][0] = 0.4;
    weight_matrix[1][1] = 0.7;
    for (i = 0; i < 2; i = i + 1) begin
        membrane_potential[i] = 0.0;
        output_spike[i] = 0;
    end
end

// Processing logic
always @(posedge clk) begin
    if (reset) begin
        for (i = 0; i < 2; i = i + 1) begin
            membrane_potential[i] <= 0.0;
            output_spike[i] <= 0;
        end
    end else begin
        // Decay previous potentials
        for (i = 0; i < 2; i = i + 1) begin
            membrane_potential[i] <= membrane_potential[i] * decay;
        end

        // Input integration
        for (i = 0; i < 2; i = i + 1) begin
            for (j = 0; j < 2; j = j + 1) begin
                if (input_spike[j])
                    membrane_potential[i] <= membrane_potential[i] + weight_matrix[i][j];
            end
        end
```

```
        // Firing condition
        for (i = 0; i < 2; i = i + 1) begin
            if (membrane_potential[i] >= threshold) begin
                output_spike[i] <= 1;
                membrane_potential[i] <= 0.0;

                // STDP mock update
                for (j = 0; j < 2; j = j + 1) begin
                    if (input_spike[j])
                        weight_matrix[i][j] <= weight_matrix[i][j] + 0.01;
                end
            end else begin
                output_spike[i] <= 0;

    end

endmodule
```

11. Reference

1.IBM TrueNorth Neuromorphic Chip

2.Paul A. Merolla et al.,"A million spiking-neuron integrated circuit with a scalable communication network and interface",

3.Intel Loihi Neuromorphic Architectur

4.Mike Davies et al.,"Loihi: A Neuromorphic Manycore Processor with On-Chip Learning",W. Maass,

5."Networks of spiking neurons: the third generation of neural network

6."Going Deeper in Spiking Neural Networks: VGG and Residual