Network-On-Chip Spiking Neural Network Accelerator

1. Project Overview

In an asynchronous environment, the primary objective is to implement the calculation of the Spiking Neural Network(SNN) on Network on Chip(NoC) architecture from scratch using SystemVerilog and Verilog.

2. NoC Architecture

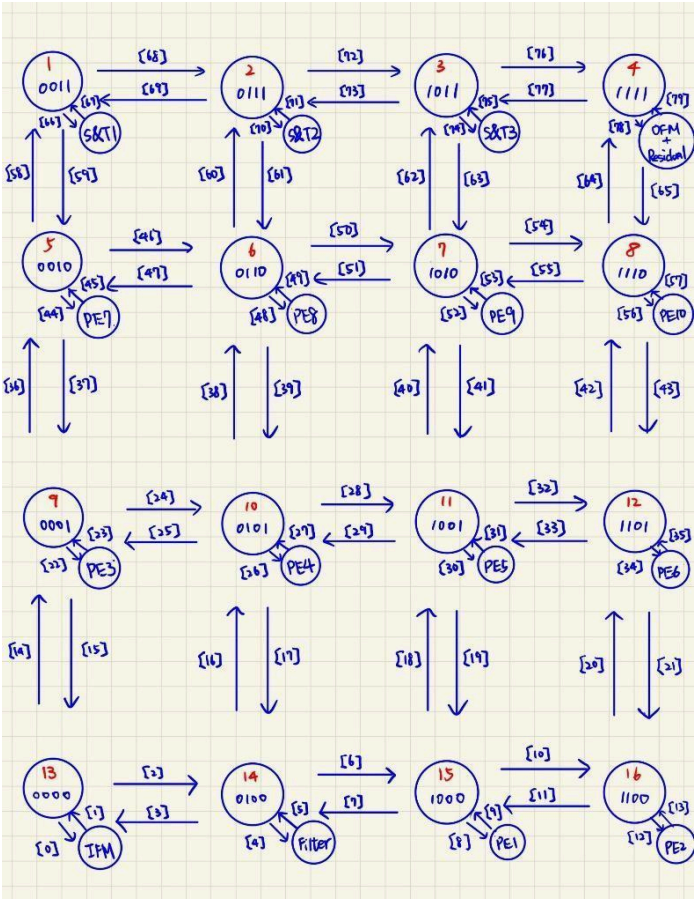

Fig. 2D 4X4 Mesh Structure

This picture above shows the 4X4 mesh coordinate numbers and Channels. For the coordinate numbers, I design 4 bits to represent because based on the X-Y algorithm I use, the former 2 bits means the X-direction positions and the later 2 bits means Y-direction positions. Hence, the X address would be from 0 to 3, and the Y address from 0 to 3 in decimal.The coordinate (0,0) is set at the left bottom corner. (3,3) is set at the right top corner.

The circles are routers in the intersections. Each router has five directions: North, South, East, West, and the processing elements or other functional elements. The numbers with brackets are the channels.

3. Router

3.1 Router Architecture

The most important part of the router is the routing algorithm. It can be implemented in many ways. Here is my explanation and the structure.

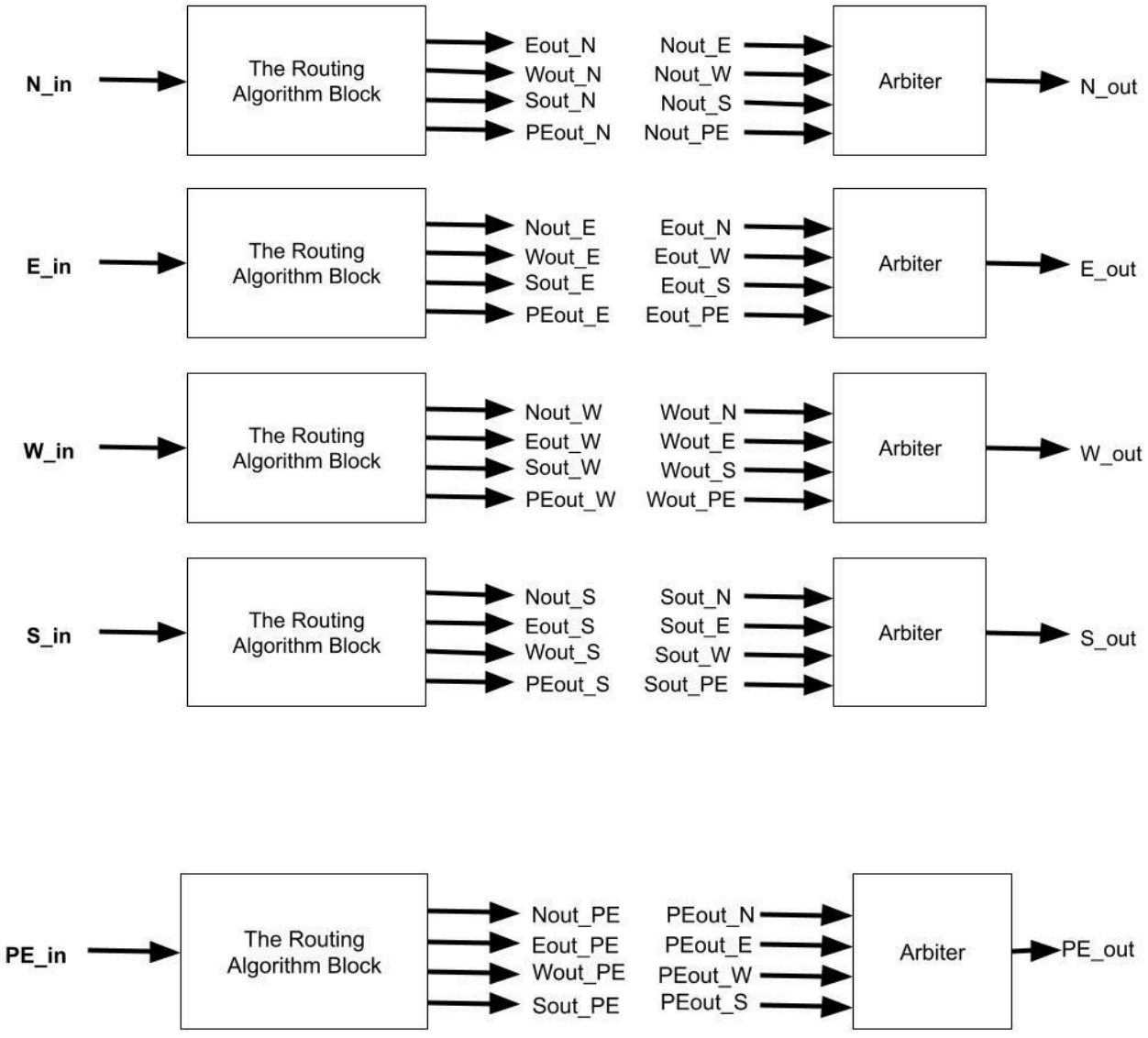

Fig. Router Architecture

For a router, there are five directions to consider: North, South, East, West, and Processing Element. Each direction has both input and output, so I defined 10 interfaces to manage bidirectional transmissions for each direction.

To simplify the design, I separated each direction into its own module. Each module is responsible for packets coming from the North, East, West, South, and PE directions, respectively. Additionally, I design 5 4-input arbiters to determine which packet gets priority to be sent to the output port.

Regarding the naming of internal channels, let’s take the North submodule as an example. If a packet is received from the North interface and is destined for PE, I name this channel "PEout_N," and similarly for other cases.

Now, let me take the module handling the packet coming from North as an example and explain how each module functions.

3.2 Router Algorithm

```
`timescale 1ns/1fs

import SystemVerilogCSP::*;

module router_dir_N(interface in, interface E, interface W,interface S, interface PE);

parameter FL=2;
parameter BL=1;
parameter WIDTH=32;

logic [WIDTH-1:0]packet_in;
logic [WIDTH-1:0]packet_out;

logic [1:0]src_x;
logic [1:0]new_src_x;
logic [1:0]src_y;
logic [1:0]new_src_y;
logic [1:0]dst_x;
logic [1:0]dst_y;
logic [1:0]sel;

// Routing from North
always begin
    $display("always block begins");
    in.Receive(packet_in);
    #FL;
    $display("I got the packet");
    packet_out=packet_in;
    fork
    src_x=packet_in[WIDTH-1:WIDTH-2];
    src_y=packet_in[WIDTH-3:WIDTH-4];
    dst_x=packet_in[WIDTH-5:WIDTH-6];
    dst_y=packet_in[WIDTH-7:WIDTH-8];
    join
    $display("I separate the address");
    //go to east
    if(dst_x>src_x)begin
        new_src_x=src_x+2'b01;
        packet_out[WIDTH-1:WIDTH-2]=new_src_x;
        E.Send(packet_out);
        $display("East send");
        end
    //go to west
    else if(dst_x<src_x)begin
```

Fig. North Code 1

```
    join
    $display("I separate the address");
    //go to east
    if(dst_x>src_x)begin
        new_src_x=src_x+2'b01;
        packet_out[WIDTH-1:WIDTH-2]=new_src_x;
        E.Send(packet_out);
        $display("East send");
        end
    //go to west
    else if(dst_x<src_x)begin
        new_src_x=src_x-2'b01;
        packet_out[WIDTH-1:WIDTH-2]=new_src_x;
        W.Send(packet_out);
        $display("West send");
    end
    //if source X=Destination X, we compared Y direction
    // go to south
    else if(dst_x==src_x&dst_y<src_y)begin
        packet_out[WIDTH-3:WIDTH-4]=new_src_y;
        S.Send(packet_out);
        $display("South send");
    end
    // Source X=Destination X and Source Y= Destination Y, got to PE
    else if(dst_x==src_x&dst_y==src_y)begin
        PE.Send(packet_out);
        $display("PE send");
    end
    #BL;
    $display("Finish always");
end
endmodule
```

Fig. North Code 2

In this algorithm, I first disassemble the packet to extract the source and destination addresses. Once I have the addresses, I compare the source and destination positions. I start by comparing the X direction (West and East). If the destination X address is greater or smaller than the source X address, indicating that the destination is to the right or left of the source, the packet is directed to either the East or West, respectively.

If the source and destination X addresses are the same, I then compare the Y addresses. If the destination Y address is smaller than the source Y address, the packet is directed to the South. If the source and destination Y addresses are identical, the packet is sent to function blocks, such as PE or the functional block.

The routing algorithm block does not include a North direction because it would be redundant. Sending a packet back to the Northern router from which it came would result in no forward movement. Therefore, one routing algorithm block only has four output directions.

After determining the appropriate direction for the packet, I update the source address. If the packet is directed East, I increment the source X address by one. If it's directed West, I decrement the source X address by one. If the packet is directed South, I increment the source Y address by one. If the packet is directed to PE, the source address remains unchanged. Before the packet leaves the PE block, I update the destination address within the PE.

3.3 Arbiter

For one arbiter, it will receive 4 packets from different router output ports. Hence, I need a 4-input arbiter. I build arbiters into submodules, too. Here is 2-input arbiter coming first.

3.3.1 Two-Input Arbiter

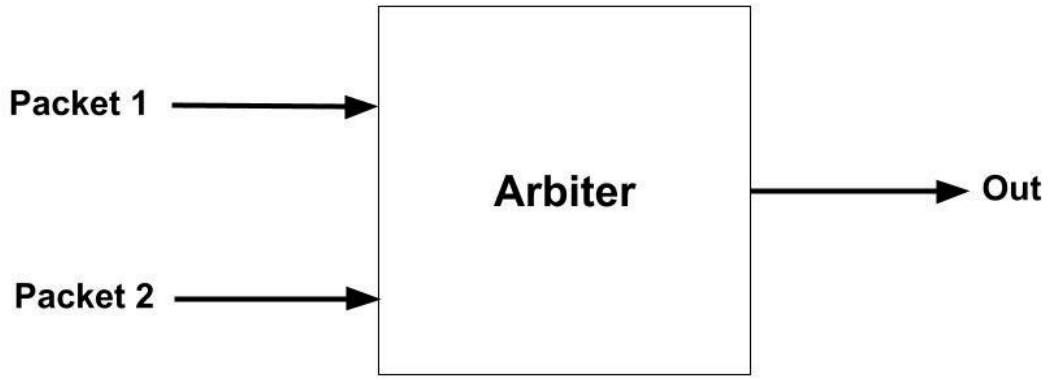

Fig. 2-input arbiter

```
`timescale 1ns/1fs

import SystemVerilogCSP::*;

module arbiter_two(interface in0, interface in1, interface out);

parameter FL=2;
parameter BL=1;
parameter WIDTH=35;
logic winner=0;
logic [WIDTH-1:0]data0, data1;

always begin
    wait(in0.req==1||in1.req==1);
    if(in0.req==1&&in1.req==1)begin
        winner=($random%2==0)?1:0;
    end
    else if(in0.req==1)begin
        winner=1'b0;
    end
    else if(in1.req==1)begin
        winner=1'b1;
    end

    if(winner==1'b0)begin
        in0.Receive(data0);
        #FL;
        out.Send(data0);
        //$display("Send data0 in router arbiter %m");
        #BL;
    end
    else if(winner==1'b1)begin
        in1.Receive(data1);
        #FL;
        out.Send(data1);
        //$display("Send data1 in router arbiter %m");
        #BL;
    end
end
endmodule
```

Fig. 2-Input Arbiter Code

First, the function waits for one of the requests to arrive. If in0.req=1, it indicates that packet 1 has arrived at the arbiter first, and "winner" is set to 0. If in1.req=1, it means packet 2 has

arrived first, and "winner" is set to 1. If in0.req=1 and in1.req=1 simultaneously, indicating that both packets arrive at the same time, the "winner" is chosen randomly as either 0 or 1.

If the "winner" is 0, the arbiter selects packet 1 and directs it to the output port first. If the "winner" is 1, the arbiter selects packet 2 and sends it to the output port first.

3.3.2 Four-Input Arbiter

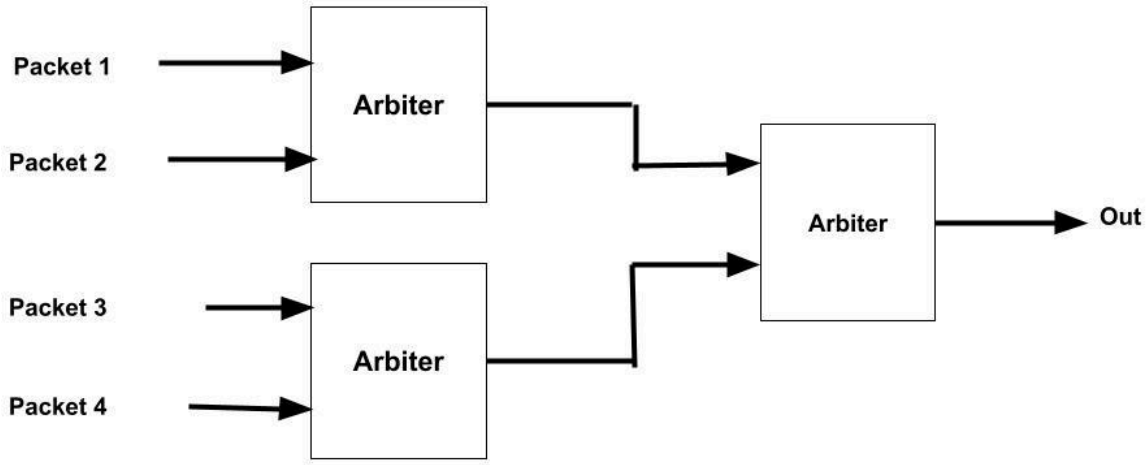

Fig. Four-Input Arbiter

```
`timescale 1ns/1fs

import SystemVerilogCSP::*;

module arbiter_four(interface in0, interface in1, interface in2, interface in3, interface out);

Channel #(.WIDTH(35),.hsProtocol(P4PhaseBD))  out1(),out2();

arbiter_two ar0(in0, in1, out1);
arbiter_two ar1(in2,in3,out2);
arbiter_two ar3(out1, out2,out);


endmodule
```

Fig. 4-input Arbiter Code

3.4 Router Top Module

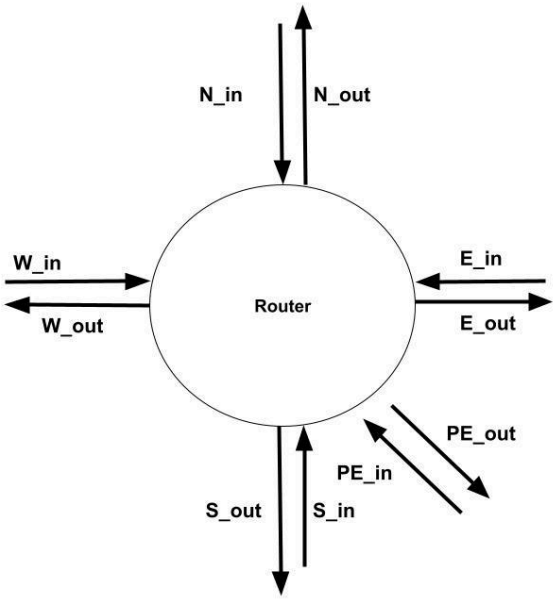

Fig. Router

```
`timescale 1ns/1fs

import SystemVerilogCSP::*;

module router(interface N_in, interface N_out, interface E_in, interface E_out, interface W_in, interface W_out, interface S_in, interface S_out, interface PE_in


// Internal out from North
Channel #(.WIDTH(35),.hsProtocol(P4PhaseBD)) Eout_N(),  Wout_N(),Sout_N(),PEout_N();
// Internal out from East
Channel #(.WIDTH(35),.hsProtocol(P4PhaseBD)) Nout_E(),  Wout_E(),Sout_E(),PEout_E();
// Internal out from West
Channel #(.WIDTH(35),.hsProtocol(P4PhaseBD)) Nout_W(), Eout_W(),Sout_W(),PEout_W();
// Internal out from South
Channel #(.WIDTH(35),.hsProtocol(P4PhaseBD)) Nout_S(), Eout_S(),  Wout_S(),PEout_S();
// Internal out from PE
Channel #(.WIDTH(35),.hsProtocol(P4PhaseBD)) Nout_PE(), Eout_PE(),  Wout_PE(),Sout_PE();

parameter FL=2;
parameter BL=1;
parameter WIDTH=35;

//Router
router_dir_N routt1(N_in,Eout_N, Wout_N,Sout_N, PEout_N);
router_dir_E routt2(E_in,Nout_E,Wout_E,Sout_E,PEout_E);
router_dir_W routt3(W_in,Nout_W, Eout_W, Sout_W,PEout_W);
router_dir_S routt4(S_in,Nout_S, Eout_S,Wout_S,PEout_S);
router_dir_PE routt5(PE_in,Nout_PE, Eout_PE, Wout_PE, Sout_PE);

//  Output Arbiter with buffers
//NORTH
arbiter_four arb_N(Nout_E,Nout_W,Nout_S,Nout_PE,N_out);
//East
arbiter_four arb_E(Eout_N,Eout_W,Eout_S,Eout_PE, E_out);
//West
arbiter_four arb_W(Wout_N,Wout_E, Wout_S,Wout_PE,W_out);
//South
arbiter_four arb_S(Sout_N,Sout_E,Sout_W,Sout_PE,S_out);
//PE
arbiter_four arb_PE(PEout_N,PEout_E,PEout_W,PEout_S,PE_out);

endmodule
```

Fig. Router Top Module

3.5 Simulation

Here, I take the packets sent to PE output ports as an example. In this case, I make all packets go to the PE output from different input ports.

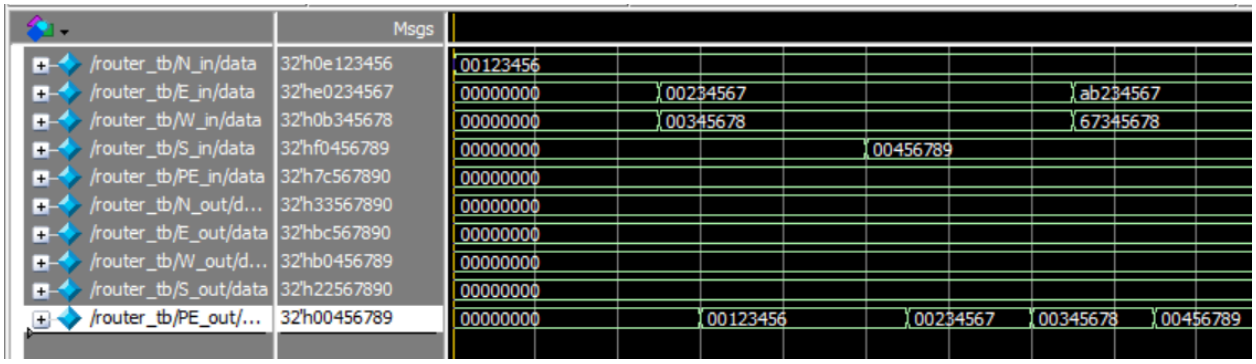

Fig. PE out Simulation

Let's examine the packet that starts with 00. We can observe that the packets from the North, East, West, and South directions are successfully routed to PE_out. Additionally, even when two packets arrive simultaneously, PE_out remains correct.

4. Data Flow

First, I send data into the filter and input feature map, dividing them into rows of 3 and 5 data points respectively. Both are packetized and then sent to the Processing Elements (PEs). In each PE, the packet undergoes depacketization and performs multiplication between the filter data and the input feature map data. The PE generates three results from this operation, referred to as partial sums A, B, and C. Each of these partial sums is packetized and sent to the corresponding sum & threshold blocks 1, 2, and 3.

As noted, there are 10 PEs handling the entire 5x5 filter map simultaneously. Each PE sends 3 packets to the 3 sum & threshold blocks independently. For example, if sum & threshold block 1 receives all 10 packets from the PEs for partial sum A, it will sum them up and compare the total with a threshold value, which is 64 in this case.

If the sum is less than the threshold, the output spike value is 0. The sum & threshold block sends a single packet containing both the sum value (which is less than the threshold) and the output spike value (0) to the combined output feature map and residue memory.

If the sum exceeds the threshold, the output spike value is 1. The sum & threshold block sends a packet with the difference between the sum and the threshold value, along with the output spike value (1), to the combined output feature map and residue memory.

The combined output feature map and residue memory then store this information. Afterward, residue memory sends back 3 packets to the 3 sum & threshold blocks, which are used to update the sum values with new data from the next timestep of the input feature map, such as timestep 2 from the testbench.

5. Packet Type

In different modules I have different packet types to suit the specific need. The examples are listed below:

(a)Filter

Source addr	Destination addr	Filter type	3 data
34-31(4 bits)	30-27(4 bits)	26-24(3 bits)	23-0(24bits)

(b)Input feature map

Source addr	Destination addr	IFM type	Zeros	5 data
34-31(4 bits)	30-27(4 bits)	26-24(3 bits)	23-5(19 bits)	4-0(5 bits)

(c)Processing element

Source addr	Destination addr	PE type (source addr)	Zeros	Partial sum data
34-31(4 bits)	30-27(4 bits)	26-23(4 bits)	22-8(15 bits)	7-0(8 bits)

(d)Sum & threshold Block

Source addr	Destination addr	S & T type (source addr)	Output spike	Zeros	Residue data
34-31(4 bits)	30-27(4 bits)	26-23(4 bits)	22(1 bit)	21-8(14 bits)	7-0(8 bits)

(e)Residue memory

Source addr	Destination addr	Residue type (source addr)	Zeros	Residue data
34-31(4 bits)	30-27(4 bits)	26-23(4 bits)	22-8(15 bits)	7-0(8 bits)

6. Filter

I designed the 5x5 filter so that odd-numbered PEs handle 3 values, while even-numbered PEs handle 3 values with the last value set to 0. This configuration aligns with the 5 input feature map values in a row, allowing us to perform one convolution per cycle. The filter then sends a packet containing 3 filter data points to each PE.

```
always begin
    load_start.Receive(flag1);
    if(flag1==1)begin
        $display("start loading filter value!");
    end
    #FL;
    while(flag1)begin
        fork
        filter_addr.Receive(addr);
        filter_data.Receive(data);
        join
        #FL;
        row=addr/5;
        col=addr%5;
        filter_mem[row][col]=data;
        counter=counter+1;
        if(counter==25)begin
            $display("finish loading data");
            flag1=0;//RReset the flag1
            counter=0;// Reset counter to 0
        end
    end
    #BL;
    $display("Start sending data to PEs");
        for(row_cnt=0;row_cnt<5;row_cnt=row_cnt+1)begin
            if(row_cnt==0) begin
                packet_out1={filter_addrr,PE1_addr,3'b000,filter_mem[row_cnt][0],filter_mem[row_cnt][1],filter_mem[row_cnt][2]};
                out.Send(packet_out1);
                $display("send packet with 3 data to PE1");
                #BL;
                packet_out2={filter_addrr,PE2_addr,3'b000,filter_mem[row_cnt][3],filter_mem[row_cnt][4],8'h00};
                out.Send(packet_out2);
                $display("send packet with 3 data to PE2");
                #BL;
            end
            else if(row_cnt==1) begin
```

Fig. Filter Code 1

```
            $display("send packet with 3 data to PE5");
            #BL;
            packet_out2={filter_addrr,PE6_addr,3'b000,filter_mem[row_cnt][3],filter_mem[row_cnt][4],8'h00};
            out.Send(packet_out2);
            $display("send packet with 3 data to PE6");
            #BL;
        end
        else if(row_cnt==3) begin
            packet_out1={filter_addrr,PE7_addr,3'b000,filter_mem[row_cnt][0],filter_mem[row_cnt][1],filter_mem[row_cnt][2]};
            out.Send(packet_out1);
            $display("send packet with 3 data to PE7");
            #BL;
            packet_out2={filter_addrr,PE8_addr,3'b000,filter_mem[row_cnt][3],filter_mem[row_cnt][4],8'h00};
            out.Send(packet_out2);
            $display("send packet with 3 data to PE8");
            #BL;
        end
        else if(row_cnt==4) begin
            packet_out1={filter_addrr,PE9_addr,3'b000,filter_mem[row_cnt][0],filter_mem[row_cnt][1],filter_mem[row_cnt][2]};
            out.Send(packet_out1);
            $display("send packet with 3 data to PE9");
            #BL;
            packet_out2={filter_addrr,PE10_addr,3'b000,filter_mem[row_cnt][3],filter_mem[row_cnt][4],8'h00};
            out.Send(packet_out2);
            $display("send packet with 3 data to PE10");
            #BL;
        end

    $display("Send all packets to PEs from %m");
    end
end
endmodule
```

Fig. Filter Code 2

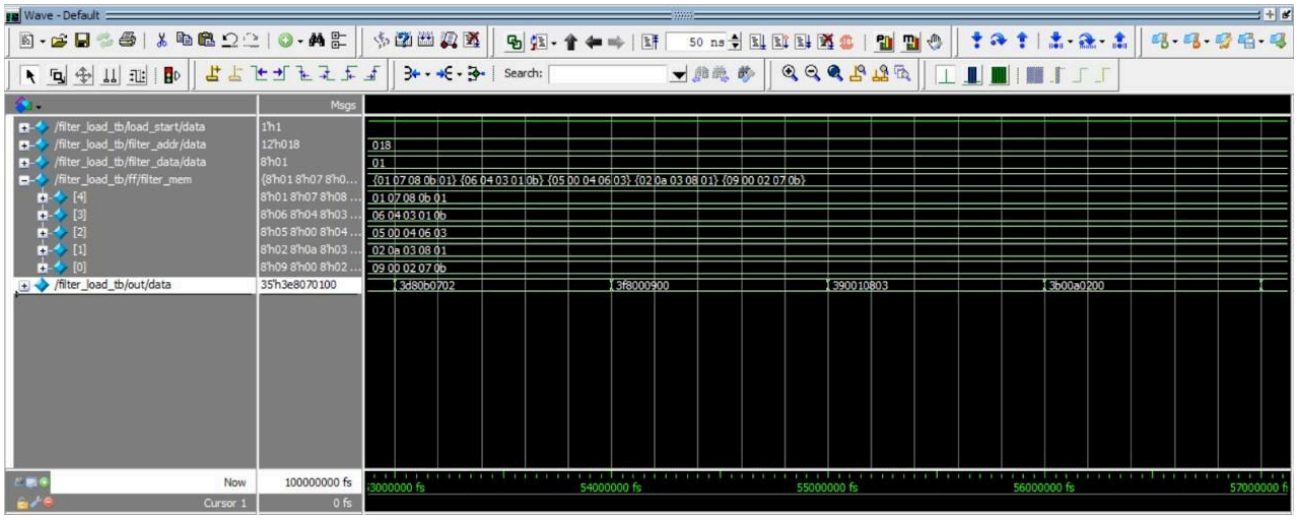

Fig. Filter Simulation

Read the data in filter successfully and divided it into the form I want.

7. Input Feature Map(IFM)

I designed a 32x32 input feature map. To create one of the five rows in the convolution with a 5X5 filter,I overlap two of the five data points between two sets of input feature maps when multiplying with two filter data points. For example, the filter data and input feature map values for PE1 and PE2 combined can generate one row of the filter map. With the help of PE3 through PE10, I can construct the entire convolution for each 5X5 data in IFM.

Additionally, I apply an offset to right-shift the input feature map by 3 for the next set of 5 values. The input feature map then sends each value to the 10 PEs. It's important to note that the input feature map waits for a "done" signal from the output feature map, indicating that the entire convolution for partial sums A, B, and C is complete. Once this signal is received, the input feature map sends a new set of values to each PE to ensure the correct data is calculated.

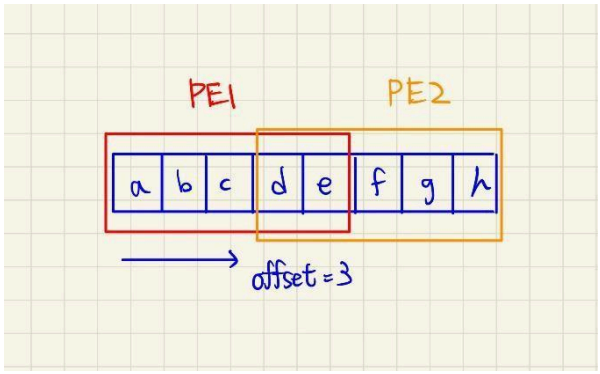

Fig. IFM Diagram

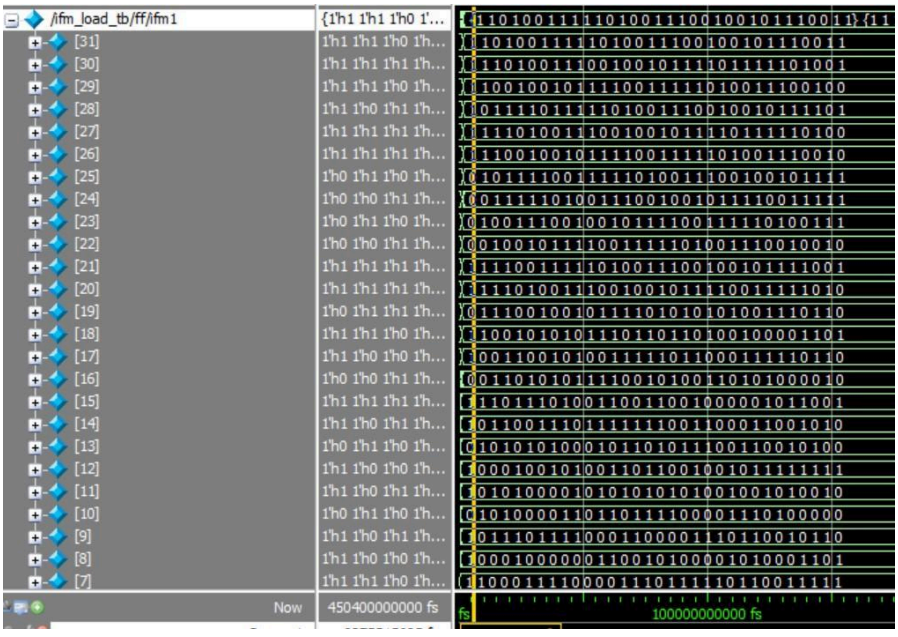

Fig. IFM Simulation

8. Processing Element (PE)

I designed our processing element (PE) to receive packets from both the input feature map and the filter. After receiving these packets, the PE calculates partial sums A, B, and C. The interesting aspect here is that I use conditional statements instead of traditional multiplication. This approach reduces delay and increases efficiency. The implementation is shown in the code below.

I also addressed the border issue. Since our output feature map is 28x28, I can create 3 partial sums at once. However, when 28%3 equals 1, the output feature map only needs to store one value on the 10th iteration. To handle this, I set a counter, and when it equals 10, the PE sends only one packet—specifically partial sum A—instead of three.

For the variable “counter_receive” means that after PE receives packets from filter and IFM, it would be 1 so that PE can continue to calculate the multiplication and partial sum.

```
always begin
        $display("Depacket");
        in.Receive(packet_in);
        packet_type=packet_in[WIDTH-9:WIDTH-11];
        if(packet_type==3'b000)begin
        $display("this packet is from filter");
            filter_value3=packet_in[23:16];
            filter_value2=packet_in[15:8];
            filter_value1=packet_in[7:0];
            #FL;
        end
        if(packet_type==3'b001)begin
        $display("this packet is from Map");
            ifmap_mem5=packet_in[4];
            ifmap_mem4=packet_in[3];
            ifmap_mem3=packet_in[2];
            ifmap_mem2=packet_in[1];
            ifmap_mem1=packet_in[0];
            #FL;
            counter_receive=counter_receive+1;
            counter=counter+1;
        end
        #delay;
        if(counter_receive==1)begin
            counter_receive=0;
            if(counter<10)begin
            $display("Finish receiving 2 packets");
            $display("Calculate Partial Sum");
                psumA=(ifmap_mem5?filter_value3:0)+(ifmap_mem4?filter_value2:0)+(ifmap_mem3?filter_value1:0);
                packet1={PE_addr,sum_thr1,PE_addr,15'b000000000000000, psumA};
                out.Send(packet1);
                $display("send Packet to S&T1");
                //#delay;
                psumB=(ifmap_mem4?filter_value3:0)+(ifmap_mem3?filter_value2:0)+(ifmap_mem2?filter_value1:0);
                packet2={PE_addr,sum_thr2,PE_addr,15'b000000000000000, psumB};
                out.Send(packet2);
                $display("send Packet to S&T2");
                //#delay;
                psumC=(ifmap_mem3?filter_value3:0)+(ifmap_mem2?filter_value2:0)+(ifmap_mem1?filter_value1:0);
                packet3={PE_addr,sum_thr3,PE_addr,15'b000000000000000, psumC};
                out.Send(packet3);
                $display("send Packet to S&T3");
                #delay;
            end
```

Fig. PE Code 1

```
            if(counter==10)begin
                psumA=(ifmap_mem5?filter_value3:0)+(ifmap_mem4?filter_value2:0)+(ifmap_mem3?filter_value1:0);
                packet1={PE_addr,sum_thr1,PE_addr,15'b000000000000000, psumA};
                out.Send(packet1);
                $display("send Packet to S&T1 only one value");
                #delay;
                counter=0;
            end
        end
end
endmodule
```

Fig. PE Code 2

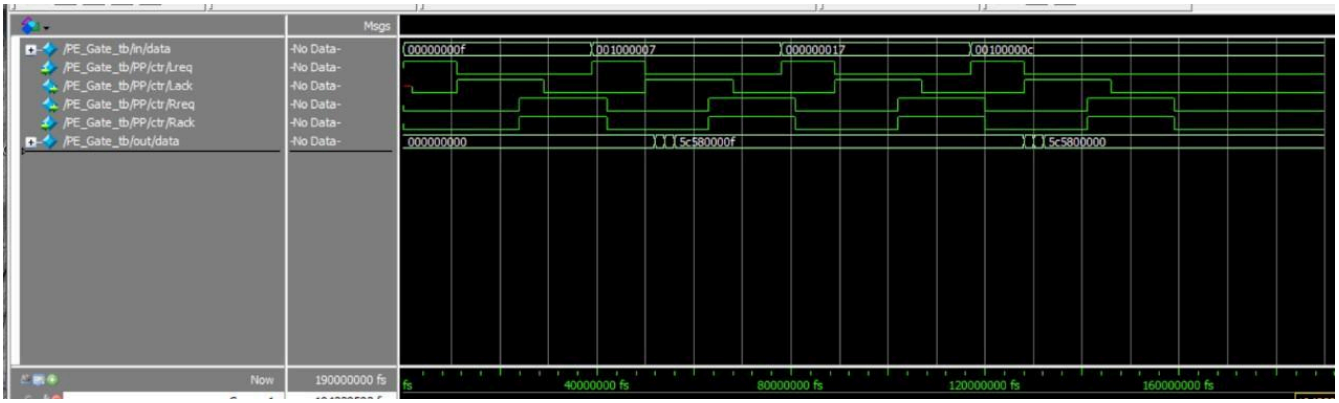

Fig. PE Simulation

9. Sum and Threshold Block

For the sum and threshold block, I divided the process into two steps. First, in the first time step for the first IFM, I sum all 10 values from PEs and then compare the result with the threshold value. If the sum is greater than the threshold, the output spike is set to 1, and the residue value is calculated as the sum minus the threshold. This information is then sent in a single packet to the block of output feature map and residue memory. If the sum is less than the threshold, the output spike value is 0, and the residue value is equal to the sum.

In the second time step for the second IFM, I sum all 10 values from PEs along with the residue value already calculated in the first IFM and the residue values return to PEs from the residue memory.

An important point to note is that I split the sum and threshold block into two separate files: one for sum & threshold 1 and another for sum & threshold 2_3. The sum & threshold 1 block handles the normal case, receiving all partial sum values from the PEs. In contrast, the sum & threshold 2_3 blocks have a different design, as they do not receive packets when a border issue occurs. This requires adjusting the counter by subtracting 28, resulting in a counter value of 252. This means that sum & threshold 2_3 receives packets 252 times instead of 280, as it

misses 28 packets due to the border issue compared to sum & threshold 1. The simulation graph below demonstrates that the process runs perfectly, highlighting the transition from timestep 1 to timestep 2, where the residue value plays a crucial role.

Without accounting for the border issue in sum & threshold 2_3, a deadlock would occur, causing the system to fail in completing the calculation of the Spiking Neural Network (SNN) on the Network on Chip (NoC).

```
        if(counter_receive==10)begin
            $display("%m receive all data from PE1 to PE10 in %m");
            Add_value = pe1_value + pe2_value + pe3_value + pe4_value + pe5_value + pe6_value + pe7_value + pe8_value + pe9_value + pe10_value;
            $display("Add value in map 1:%b in %m", Add_value);
        //Threshold
            if (Add_value>=threshold) begin
                output_spike = 1;
                new_value = Add_value - threshold;
            end
            else if(Add_value<threshold)begin
                output_spike = 0;
                new_value = Add_value;
            end
            packet_out = {sum_addr, res_addr,sum_addr,output_spike, res_zeros, new_value};//4+4+4+1+14+8=35 bits
            $display("Send packet to OFM from %m");
            out.Send(packet_out);
            counter=counter+1'b1;
            $display("counter number %d is in %m",counter);
            counter_receive=0;

        #BL;
        end
        if(counter==280)begin//trst
        counter=0;
        tstep2=1;
        counter_receive=0;
        //end
        end
    end
```

Fig. Sum & Threshold 1

```
        if(counter_receive==10)begin
            $display("%m receive all data from PE1 to PE10 in %m");
            Add_value = pe1_value + pe2_value + pe3_value + pe4_value + pe5_value + pe6_value + pe7_value + pe8_value + pe9_value + pe10_value;
            $display("Add value in map 1:%b in %m", Add_value);
        //Threshold
            if (Add_value>=threshold) begin
                output_spike = 1;
                new_value = Add_value - threshold;
            end
            else if(Add_value<threshold)begin
                output_spike = 0;
                new_value = Add_value;
            end
            packet_out = {sum_addr, res_addr,sum_addr,output_spike, res_zeros, new_value};//4+4+4+1+14+8=35 bits
            $display("Send packet to OFM form %m");
            #6;
            out.Send(packet_out);
            counter=counter+1'b1;
            counter_receive=0;
            $display("counter number %d is in %m",counter);

        #BL;
        end
        if(counter==252)begin//trst
        counter=0;
        tstep2=1;
        end
    end
```

Fig. Sum & Threshold 2_3

We can see the difference in counter values between different blocks.

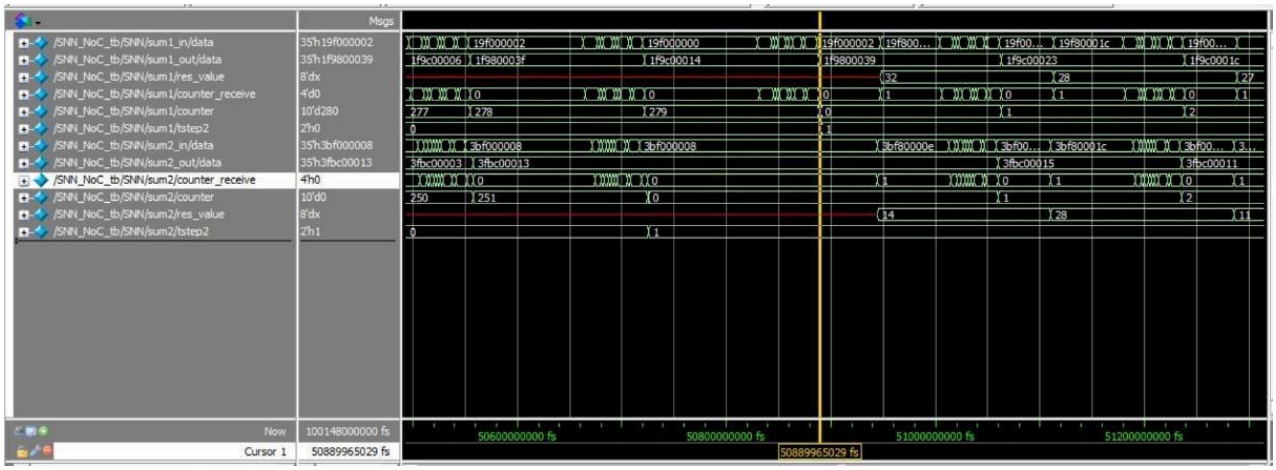

Fig. Simulation

10. Output Feature Map (OFM) + Residue Memory

I designed the output feature map (OFM) and residue memory to be combined into a single block, optimized for the 4x4 2D mesh. This integration increases throughput since packets don't need to be sent separately to different blocks. In our design, when a packet is received, it is simultaneously stored in both the output feature map and residue memory. Once the OFM and residue memory have received three packets from the sum & threshold blocks 1, 2, and 3, they send a "done" packet back to the input feature map, requesting a new set of input feature map values. Additionally, I store the values in one dimension array instead of two, which simplifies debugging.

I still need to manage the border issue. When the counter reaches 28, the OFM and residue memory receive only one value from the sum & threshold 1 to prevent a deadlock. When the counter reaches 784 (28x28), the calculation for time step 1 is complete, and the first three residue memory values are sent back to the three sum & threshold blocks to initiate the timestep 2 calculations, avoiding deadlock. For the timestep 2 map calculation, we add the residue value to the sum value to ensure accurate results, following the same process as time step 1. Finally, the OFM and residue memory compare their outputs with the golden results.

```
if(timestep==1)begin
in.Receive(packet_in);
    if(counter_28!=28)begin
        if (sum1_addr==packet_in[WIDTH-9:WIDTH-12]) begin
            sum1_value =packet_in[7:0];
            spike1=packet_in[WIDTH-13];
            #FL;
            counter_receive=counter_receive+1;
            counter_28=counter_28+1;

        end

        if (sum2_addr==packet_in[WIDTH-9:WIDTH-12]) begin
            sum2_value =packet_in[7:0];
            spike2=packet_in[WIDTH-13];
            #FL;
            counter_receive=counter_receive+1;
            counter_28=counter_28+1;

        end

        if (sum3_addr==packet_in[WIDTH-9:WIDTH-12]) begin
            sum3_value =packet_in[7:0];
            spike3=packet_in[WIDTH-13];
            #FL;
            counter_receive=counter_receive+1;
            counter_28=counter_28+1;


        end
        if(counter_receive==3)begin
            res_mem1 [addr1+2]=sum3_value;
            ofm_mem1[addr1+2] = spike3;
            res_mem1 [addr1+1]=sum2_value;
            ofm_mem1[addr1+1] = spike2;
            res_mem1 [addr1]=sum1_value;
```

Fig. OFM+ Residue Mem Code 1

```
    end
    if(counter_28==28)begin
        if (sum1_addr==packet_in[WIDTH-9:WIDTH-12]) begin
                res_mem1 [addr1]=sum1_value;
                ofm_mem1[addr1] = spike1;
                #FL;
                counter_receive=0;
                addr1=addr1+1;
                counter_28=0;
            packet_done={ofm_addr,ifmap_addr,27'b000000000000000000000000001};
            out.Send(packet_done);
            #2;
        end
    end
    if(addr1==784)begin
                $display("FInal done packet in ts=1");
                $display("################################ Finish Loading Timestep=1 in %m");
                packet_done={ofm_addr,ifmap_addr,27'b000000000000000000000000001};
                out.Send(packet_done);
                //Send first 3 residual values in residual memory of timestep1 to sum1, sum2,sum3
                res1[7:0]=res_mem1[0];
                res2[7:0]=res_mem1[1];
                res3[7:0]=res_mem1[2];
                packet_send1={ofm_addr,sum1_addr,ofm_addr,15'b000000000000000,res1};//4+4+4+15+8=35
                out.Send(packet_send1);
                #BL;
                packet_send2={ofm_addr,sum2_addr,ofm_addr,15'b000000000000000,res2};
                out.Send(packet_send2);
                #BL;
                packet_send3={ofm_addr,sum3_addr,ofm_addr,15'b000000000000000,res3};
                out.Send(packet_send3);
                #BL;
                counter_28=0;
                counter_receive=0;
                timestep=2;
    end
```

Fig. OFM+ Residue Mem Code 2

```
if(timestep==2)begin
    in.Receive(packet_in);
    if(counter_28!=28)begin
        if (sum1_addr==packet_in[WIDTH-9:WIDTH-12]) begin
            sum1_value =packet_in[7:0];
            spike1=packet_in[WIDTH-13];
            #FL;

            counter_receive=counter_receive+1;
            counter_28=counter_28+1;

        end

        if (sum2_addr==packet_in[WIDTH-9:WIDTH-12]) begin
            sum2_value =packet_in[7:0];
            spike2=packet_in[WIDTH-13];
            #FL;
            counter_receive=counter_receive+1;
            counter_28=counter_28+1;

        end

        if (sum3_addr==packet_in[WIDTH-9:WIDTH-12]) begin
            sum3_value =packet_in[7:0];
            spike3=packet_in[WIDTH-13];
            counter_receive=counter_receive+1;
            counter_28=counter_28+1;
        end
        if(counter_receive==3)begin
                res_mem2 [addr2+2]=sum3_value;
                ofm_mem2[addr2+2] = spike3;
                res_mem2 [addr2+1]=sum2_value;
                ofm_mem2[addr2+1] = spike2;
                res_mem2 [addr2]=sum1_value;
                ofm_mem2[addr2] = spike1;
                addr2=addr2+3;
            if(counter_28!=27)begin
                res1=res_mem1[addr2];
                packet_send1={ofm_addr,sum1_addr,ofm_addr,15'b000000000000000,res1};//4+4+4+15+8=35
                out.Send(packet_send1);
                #BL;
                res2[7:0]=res_mem1[addr2+1];
                packet_send2={ofm_addr,sum2_addr,ofm_addr,15'b000000000000000,res2};
                out.Send(packet_send2);
                #BL;
```

Fig. OFM+ Residue Mem Code 3

```
                packet_send2={ofm_addr,sum2_addr,ofm_addr,15'b000000000000000,res2};
                out.Send(packet_send2);
                #BL;
                res3[7:0]=res_mem1[addr2+2];
                packet_send3={ofm_addr,sum3_addr,ofm_addr,15'b000000000000000,res3};
                out.Send(packet_send3);
                counter_receive=0;
                packet_done={ofm_addr,ifmap_addr,27'b000000000000000000000000001};
                out.Send(packet_done);
            end
            if(counter_28==27)begin
                res1=res_mem1[addr2];
                packet_send1={ofm_addr,sum1_addr,ofm_addr,15'b000000000000000,res1};//4+4+4+15+8=35
                out.Send(packet_send1);
                #BL;
                counter_receive=0;
                packet_done={ofm_addr,ifmap_addr,27'b000000000000000000000000001};
                out.Send(packet_done);
            end
        end
            #FL;
    end
    if(counter_28==28)begin
        if (sum1_addr==packet_in[WIDTH-9:WIDTH-12]) begin
                res_mem2 [addr2]=sum1_value;
                ofm_mem2[addr2] = spike1;
                #FL;
                addr2=addr2+1;
                res1[7:0]=res_mem1[addr2];
                packet_send1={ofm_addr,sum1_addr,ofm_addr,15'b000000000000000,res1};//4+4+4+15+8=35
                out.Send(packet_send1);
                #BL;
                res2[7:0]=res_mem1[addr2+1];
                packet_send2={ofm_addr,sum2_addr,ofm_addr,15'b000000000000000,res2};
                out.Send(packet_send2);
                #BL;
                res3[7:0]=res_mem1[addr2+2];
                packet_send3={ofm_addr,sum3_addr,ofm_addr,15'b000000000000000,res3};
                out.Send(packet_send3);

                counter_28=0;
                counter_receive=0;
                packet_done={ofm_addr,ifmap_addr,27'b000000000000000000000000001};
                out.Send(packet_done);
```

Fig. OFM+ Residue Mem Code 4

```
                end
                if(counter_28==27)begin
                    res1=res_mem1[addr2];
                    packet_send1={ofm_addr,sum1_addr,ofm_addr,15'b000000000000000,res1};//4+4+4+15+8=35
                    out.Send(packet_send1);
                    #BL;
                    counter_receive=0;
                    packet_done={ofm_addr,ifmap_addr,27'b000000000000000000000000001};
                    out.Send(packet_done);
                end
            end
                #FL;
        end
        if(counter_28==28)begin
            if (sum1_addr==packet_in[WIDTH-9:WIDTH-12]) begin
                    res_mem2 [addr2]=sum1_value;
                    ofm_mem2[addr2] = spike1;
                    #FL;
                    addr2=addr2+1;
                    res1[7:0]=res_mem1[addr2];
                    packet_send1={ofm_addr,sum1_addr,ofm_addr,15'b000000000000000,res1};//4+4+4+15+8=35
                    out.Send(packet_send1);
                    #BL;
                    res2[7:0]=res_mem1[addr2+1];
                    packet_send2={ofm_addr,sum2_addr,ofm_addr,15'b000000000000000,res2};
                    out.Send(packet_send2);
                    #BL;
                    res3[7:0]=res_mem1[addr2+2];
                    packet_send3={ofm_addr,sum3_addr,ofm_addr,15'b000000000000000,res3};
                    out.Send(packet_send3);

                    counter_28=0;
                    counter_receive=0;
                    packet_done={ofm_addr,ifmap_addr,27'b000000000000000000000000001};
                    out.Send(packet_done);
            end
        end
            if(addr2==784)begin
                timestep=3;
                ts=1;
                break;
            end
    end
end
```

Fig. OFM+ Residue Mem Code 5

Figure 23: OFM simulation (value)

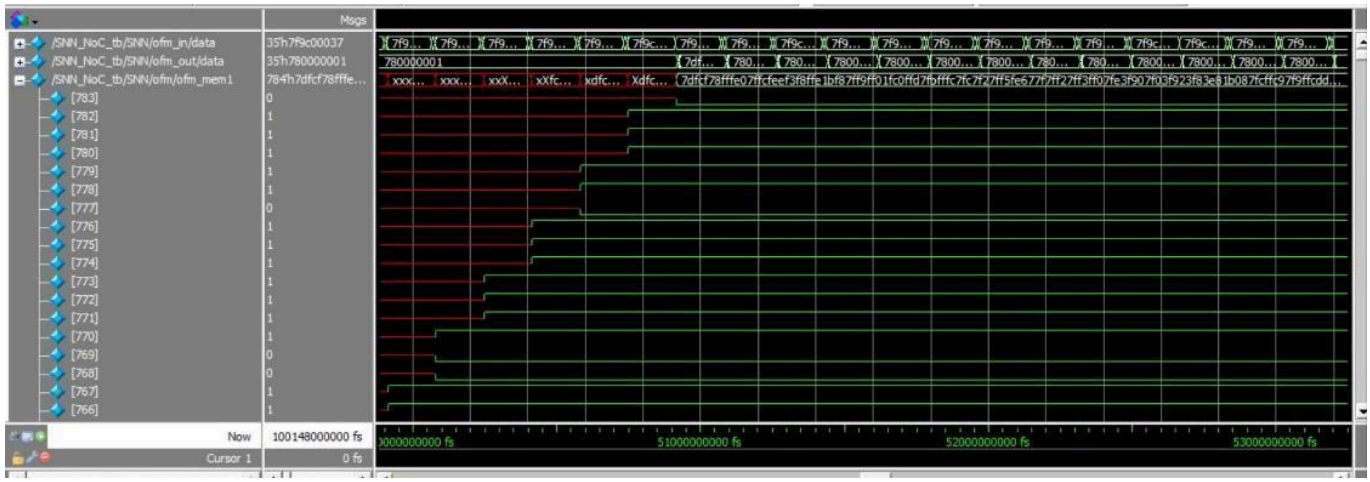

Figure 24: OFM simulation (timestep1 → timestep2)

Figure 25: Residue memory simulation

11. Top Module

```
`timescale 1ns/1fs

import SystemVerilogCSP::*;

module SNN_NoC(interface load_start, ifmap_data, ifmap_addr, timestep, filter_data, filter_addr, load_done, start_r, ts_r, layer_r, out_spike_a

parameter ifm_addrr=4'b0000;
parameter filter_addrr=4'b0100;
parameter PE1_addr=4'b1000;
parameter PE2_addr=4'b1100;
parameter PE3_addr=4'b0001;
parameter PE4_addr=4'b0101;
parameter PE5_addr=4'b1001;
parameter PE6_addr=4'b1101;
parameter PE7_addr=4'b0010;
parameter PE8_addr=4'b0110;
parameter PE9_addr=4'b1010;
parameter PE10_addr=4'b1110;
parameter sum_thr1=4'b0011;
parameter sum_thr2=4'b0111;
parameter sum_thr3=4'b1011;
parameter ofm_addr = 4'b1111;

Channel #(.WIDTH(35),.hsProtocol(P4PhaseBD)) filter_in(), filter_out(), ifmap_in(),ifmap_out(),PE1_in(), PE1_out(),PE2_in(), PE2_out(),PE3_in()

// 4X4 Mesh Routers
two_d_mesh mesh(filter_in, filter_out, ifmap_in,ifmap_out,PE1_in, PE1_out,PE2_in, PE2_out,PE3_in, PE3_out,PE4_in, PE4_out,PE5_in, PE5_out,PE6_i

//Filter
filter_load filter(load_start,filter_data,filter_addr, filter_in,filter_out);

//Input Feature Map
ifp_load ifm(timestep,ifmap_data,ifmap_addr, load_done, ifmap_in, ifmap_out);

//Processing Elements
PE #(.PE_addr(PE1_addr))  PE1(PE1_in, PE1_out);
PE #(.PE_addr(PE2_addr))  PE2(PE2_in, PE2_out);
```

Fig. Top Module Code 1

```
//Processing Elements
PE #(.PE_addr(PE1_addr))  PE1(PE1_in, PE1_out);
PE #(.PE_addr(PE2_addr))  PE2(PE2_in, PE2_out);
PE #(.PE_addr(PE3_addr))  PE3(PE3_in, PE3_out);
PE #(.PE_addr(PE4_addr))  PE4(PE4_in, PE4_out);
PE #(.PE_addr(PE5_addr))  PE5(PE5_in, PE5_out);
PE #(.PE_addr(PE6_addr))  PE6(PE6_in, PE6_out);
PE #(.PE_addr(PE7_addr))  PE7(PE7_in, PE7_out);
PE #(.PE_addr(PE8_addr))  PE8(PE8_in, PE8_out);
PE #(.PE_addr(PE9_addr))  PE9(PE9_in, PE9_out);
PE #(.PE_addr(PE10_addr)) PE10(PE10_in, PE10_out);

// Sum & Threshold Blocks
Sum_Threshold1   #(.sum_addr(sum_thr1)) sum1(sum1_in, sum1_out);
Sum_Threshold2_3 #(.sum_addr(sum_thr2)) sum2(sum2_in, sum2_out);
Sum_Threshold2_3 #(.sum_addr(sum_thr3)) sum3(sum3_in, sum3_out);

//Output Feature Map + Residual Memory
 OF_Residual_mem ofm(ofm_in,ofm_out, start_r, ts_r, layer_r, out_spike_addr, out_spike_data, done_r);


 endmodule
```

Fig. Top Module Code 2

```
`timescale 1ns/1fs

import SystemVerilogCSP::*;

module SNN_NoC_tb;


Channel #(.WIDTH(2),.hsProtocol(P4PhaseBD)) ts_r(),timestep(),layer_r();
Channel #(.WIDTH(12),.hsProtocol(P4PhaseBD)) out_spike_addr(),ifmap_addr(),filter_addr();
Channel #(.WIDTH(13),.hsProtocol(P4PhaseBD)) out_spike_data(),filter_data();
Channel #(.WIDTH(1),.hsProtocol(P4PhaseBD)) ifmap_data(),done_r(),load_start(),start_r(),load_done();

noc_snn_tb test(load_start, ifmap_data, ifmap_addr, timestep, filter_data, filter_addr, load_done, start_r, ts_r, layer_r, out_spike_addr, out_sp
SNN_NoC SNN(load_start, ifmap_data, ifmap_addr, timestep, filter_data, filter_addr, load_done, start_r, ts_r, layer_r, out_spike_addr, out_spike_


initial begin
#1000;
end

endmodule
```

Fig. Top Module Testbench

12. Result and Analysis

As you can see, the result is perfect with zero errors, and I managed to lower the execution time to approximately 100 microseconds, with 50 microseconds for each timestep. In the entire system, the most time-consuming part is loading the filter values and input feature map values from the testbench. To minimize the overall delay, I implemented several enhancements, including sending three filter data in one packet, using conditional statements in PEs instead of multiplication, employing three sum and threshold blocks for partial sums A, B, and C to create parallelism, combining output spike information with the residue value in one packet to increase throughput, and sending back residue values to the three sum & threshold blocks only when needed to avoid unnecessary packets. Overall, I achieved a relatively short execution time and am pleased with the results.

For future work, I plan to explore more enhancements, not only in increasing throughput but also in areas such as error detection and alternative routing methods. Professor Beerel mentioned that adaptive routing can be extremely challenging with asynchronous design, and I am interested in exploring whether there is a method that could be more effective than XY routing. Ultimately, I've learned a lot from this project; it was a time-consuming but rewarding process.

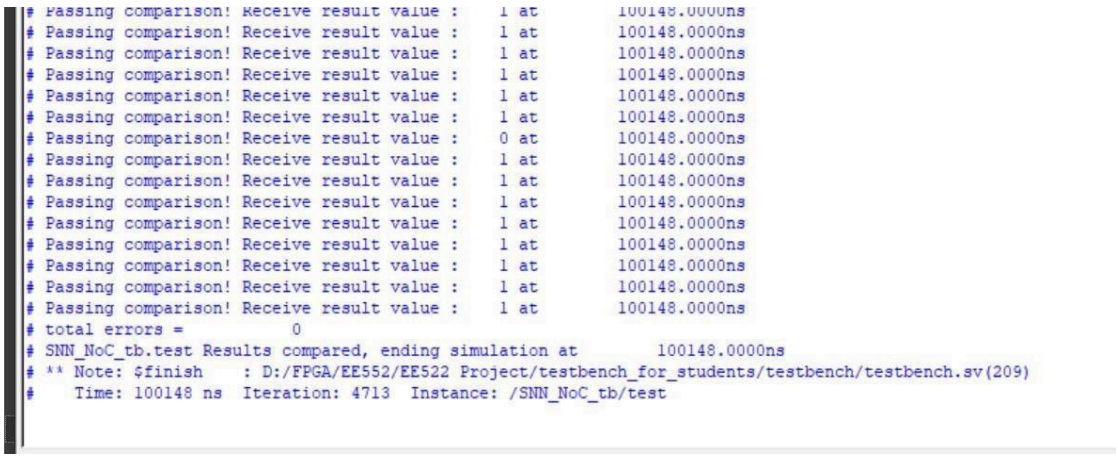

Fig. The result

13. Reference

(1)Review of XY Routing Algorithm for 2D Torus Topology of NoC Architecture
https://research.ijcaonline.org/retret/number1/retret1310.pdf

(2)The Turn Model for Adaptive Routing
https://ieeexplore.ieee.org/document/753324

(3) An Asynchronous Reconfigurable SNN Accelerator With Event-Driven Time Step Update

https://ieeexplore.ieee.org/document/9056903

(4) Deadlock Free Routing in Mesh Networks on Chip with Regions
https://www.diva-portal.org/smash/get/diva2:233717/FULLTEXT01.pdf

(5) SE_20 Network-on-chip (NOC) Topologies slides