Asynchronous Spiking Neural Network

on a Network-on-Chip

Stage 2 Progress Report

Yu-Ting Chiu 2717896258 ychiu078@usc.edu

Beila Zhao 9821347549 beilazha@usc.edu

Chenjie Weng 4599107598 cweng701@usc.edu

Jiahui Wang 2436803430 jwang246@usc.edu

Group GitHub Link: https://github.com/Jefferyzzo/EE552_Final_Project

1. Introduction

In the baseline design we intend to realize PE that calculates one convolutional output for 4X4 ifmap and 3X3 filter map with a 6-node Tree NoC. Our advanced design aims at making both ifmap and filter map size configurable while PE is still able to calculate one convolutional output. To further improve throughput of the NoC, we applied a 4X4 mesh to support more complicated data flow between PE, memories and control units.

Currently we have finished configurable-sized mesh and baseline PE. Tree NoC is still under test. The following parts elaborates our finished design and future plan.

1.1 Block diagram

For tree topology, node number is log2 the number of (PE + Input/output Memory Unit), as shown in the figure below.

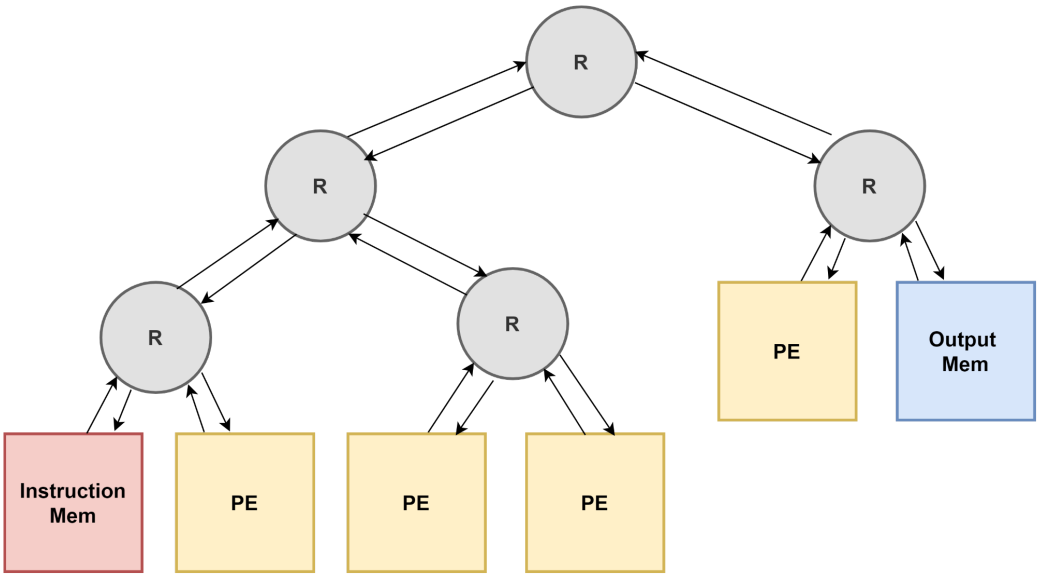

Figure 1. Tree topology

For mesh topology, node number is the concatenation of its (x,y) coordinate in binary form, as shown in the below figure.

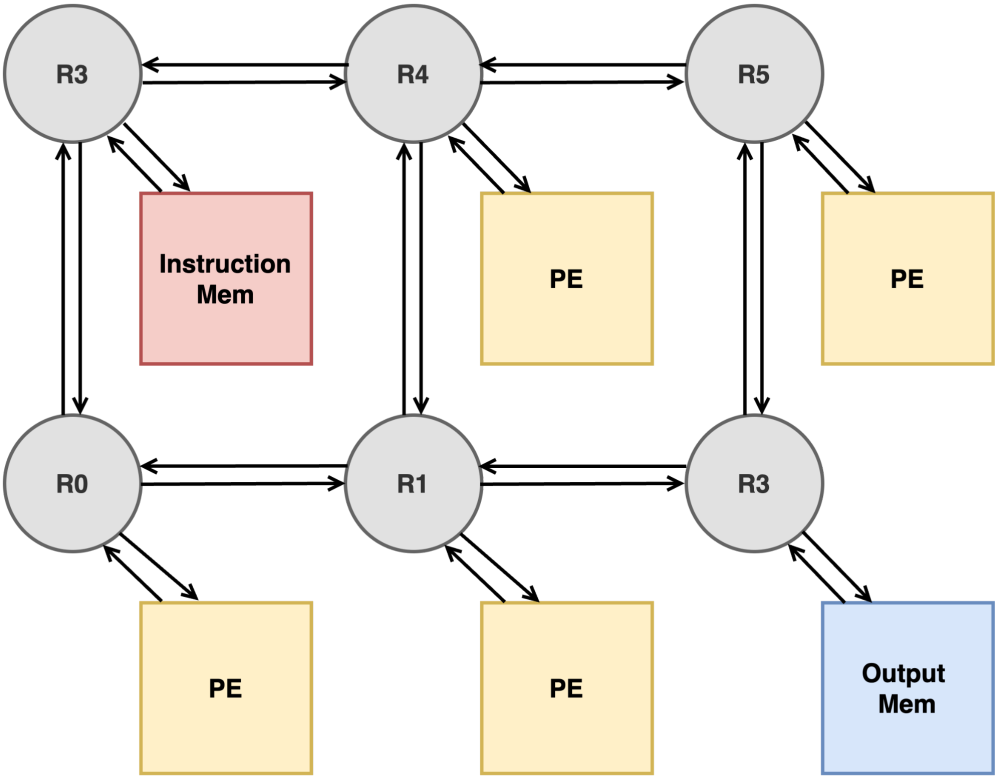

Figure 2. Mesh topology

1.2 Packet Format

1.2.1 Packet Header Format for Tree

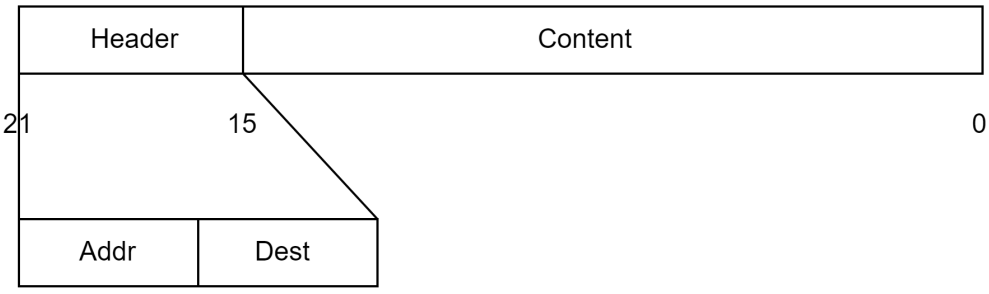

Figure 3. Packet organization before entering to PE for tree topology

- Packet[21:16]: the header for packet transmission among 2x3 mesh routers
    - Packet[18:16] : These 3 bits represent the destination address of the packet to be sent. They are also used to generate the routing mask, which helps the router determine the correct direction to forward the packet.
    - Packet[21:19] : These three bits represent the source address of the packet and are also used during packet transmission for logical determination of the sending direction.

1.2.2 Packet Header Format for Mesh

packet orgnization									
Address	[8+3*filter_width:9+3*filter_width-output_width	[8+3*filter_width-output_width:12]	[11:10]	[9]	[8:6]	[5]	[4]	[3:2]	[1:0]
	residue	0s	PE node	outspike	3'b000	timestep	y_hop	x_hop	direction

Figure 4. Packet organization before entering to PE for mesh topology

- Packet[4:0]: the header for packet transmission among 2x3 mesh routers
    - Packet[1:0] : direction that packet travels. For direction[0], 0 means West and 1 means East while for direction[1], 0 means North and 1 means South.
    - Packet[4:2] : Unary-encoded hop value for x and y direction. For example, if there are 2 x hops, Packet[3:2]=11, when a x hop is taken, this value shifts right a bit. This design simplifies checking logic which only needs to check the last bit of x hop and y hop field. Also, shifters for changing hop values are simpler than implementing subtractors.

1.2.3 Packet Content Format

When the packet enters PE, header bits are aborted and only contents will be transferred to the PE, whose format is shown below.

********** content orgnization **********				
Address	[3*filter_width+3:4]	[3:2]	[1]	[0]
	3*fliter_width	filter_row	ifmapb(0)_filter(1)	timestep
**********				

Figure 5. Content organization after entering to PE

- Content[3*filter_width+3:0]: the content for PE computation
    - Content[3*filter_width+3:4]: one filter row weights or the whole ifmap bits depending on the content[1].
    - Content[3:2]: which row of the 3×3 filter this data is for(0, 1, or 2).
    - Content[1]: indicates whether the data is an ifmap or a filter (1 for filter, 0 for ifmap).
    - Content [0]: which timestep the data is used for.

2. PE

2.1 PE Block diagram

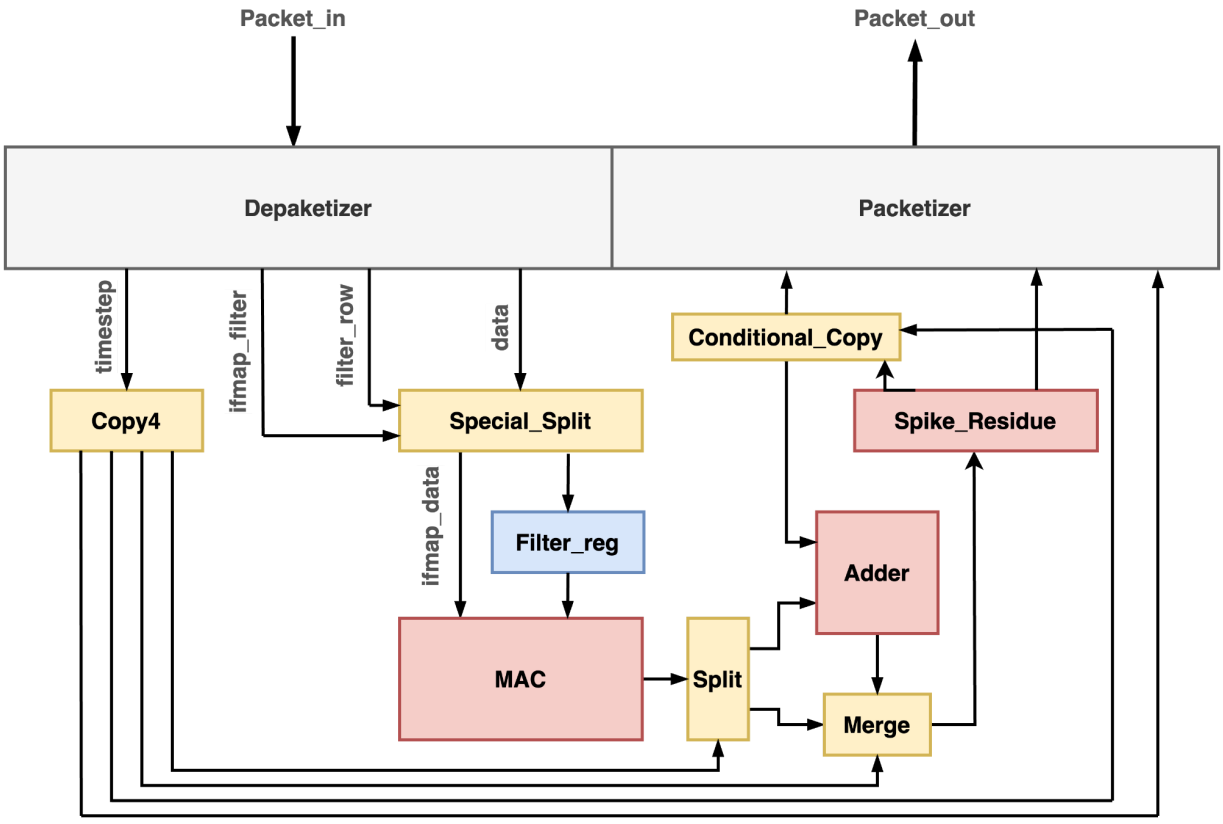

Figure 6. PE block diagram

2.2 PE Design

The **PE (Processing Element)** consists of three main stages: depacketization, calculation, and packetization. When a packet arrives at the PE, it first undergoes depacketization. In this stage, the packet is unpacked into several components: the timestep, ifmapb_filter, filter_row, and a block of data containing three rows of filter weights and the ifmap data. Based on the values of ifmapb_filter and filter_row, the unpacked data is routed to the appropriate internal channels. These channels feed

into the calculation logic, where Multiply-Accumulate (MAC) operations are performed on the corresponding filter rows and ifmap bits.

Depending on the value of the timestep, the result of the MAC operation is either passed directly to the spike residue stage, or combined with a previously stored residue through an adder. If the accumulated result exceeds a predefined threshold, a spike is generated; otherwise, the result is stored as residue to be used in subsequent computations.

After spike detection, the output data—consisting of the spike and updated residue—is reassembled into a new packet. This packet includes not only the computation results but also data such as the timestep, direction, and PE coordinates. The new packet is then sent to the next stage of processing or routing.

2.3 PE Verification

The verification of the PE design is structured to ensure modularity and thorough testing of each subcomponent within a well-organized directory system. The directory is divided into two main parts:

*Our PE design file organization can be found in the path ./Baseline/PE/PE_tree.txt

(1) `/src` — Design Files

This folder contains all the RTL source files written in SystemVerilog. Each sub-module used in the PE architecture has its own dedicated `.sv` file. For example:

- `mac.sv`, `merge.sv`, `split.sv`, etc., represent core functional blocks.
- `PE.sv` is the top-level Processing Element module that integrates all submodules.

(2) `/sim` — Simulation Files and Testbenaches

This directory includes all the testbenches and test scripts for verifying individual submodules as well as the integrated PE unit.

Each subdirectory under `sim/` corresponds to a module from `src/`. For example:

- `sim/mac/mac_tb.sv` tests `src/mac.sv`
- `sim/merge/merge_tb.sv` tests `src/merge.sv`

Each of these contains:

- A `.f` file listing all the source files needed for compilation.
- A `*_tb.sv` testbench file used to simulate the module.
- For some components, text files are included to feed input data or capture outputs (e.g., `recv_data.txt`, `send_values.txt`).

(3) Build and Run

A `Makefile` at the root of the `sim/` directory is used for the build and simulation process for different modules, streamlining the testing workflow. Simulations can be run using the following command:

```
$ make run DIR=<test_module_name>
```

This command compiles the necessary design and testbench files listed in the corresponding `.f` file and launches the simulation using **Synopsys VCS**.

Additionally, the `Makefile` provides a make clean target to remove redundant and generated simulation log files. This is particularly useful for cleaning up the workspace between test runs.

```
$ make clean
```

(4) PE Top Module Verification

Within `sim/PE/`, the integrated functionality of the PE is tested:

- `PE_tb.sv` serves as the main testbench for the top-level `PE.sv`.
- A Python script `test_scnn.py` is used to generate input files and golden results to the `scnn_script/` folder.

The verification of the PE module was carried out using Synopsys VCS. All simulation inputs were read from the `scnn_script/` directory located in `sim/PE/`, which contains test vectors such as

- `ifmap_t1.txt`, `ifmap_t2.txt` for input feature maps
- `L1_filter.txt` for convolutional filter values
- `L1_residue_t1.txt`, `L1_residue_t2.txt` for expected residue outputs
- `L1_out_spike_t1.txt`, `L1_out_spike_t2.txt` for expected output spikes

These input files were used by the `PE_tb.sv` testbench to inject data into the PE design during simulation. Throughout the test, the design-generated outputs were captured and compared directly against the golden reference files from the same `scnn_script/` folder.

To improve verification clarity, custom mismatch messages were added to the testbench. We tested their functionality by deliberately causing test failures, confirming that the messages are correctly triggered and clearly reported when outputs differ from golden values.

```
/usr/local/synopsys//VCS_2016/L-2016.06-1/linux64/lib/libzerosoft_rt_stubs.so /usr/local/synopsys//VCS_2016/L-2016.06-1/linux64/lib/libvirsim.so /
usr/local/synopsys//VCS_2016/L-2016.06-1/linux64/lib/liberrorinf.so /usr/local/synopsys//VCS_2016/L-2016.06-1/linux64/lib/libsnpsmalloc.so    /usr
/local/synopsys//VCS_2016/L-2016.06-1/linux64/lib/libvcsnew.so /usr/local/synopsys//VCS_2016/L-2016.06-1/linux64/lib/libsimprofile.so /usr/local/s
ynopsys//VCS_2016/L-2016.06-1/linux64/lib/libuclinative.so   -Wl,-whole-archive /usr/local/synopsys//VCS_2016/L-2016.06-1/linux64/lib/libvcsucli.s
o -Wl,-no-whole-archive         /usr/local/synopsys//VCS_2016/L-2016.06-1/linux64/lib/vcs_save_restore_new.o -ldl  -lc -lm -lpthread -ldl
../simv up to date
make[1]: Leaving directory `/home/viterbi/06/ychiu078/552/EE552_Final_Project/Baseline/PE/sim/PE/csrc'
Chronologic VCS simulator copyright 1991-2016
Contains Synopsys proprietary information.
Compiler version L-2016.06-1_Full64; Runtime version L-2016.06-1_Full64;  Apr 14 22:37 2025
Start simulation!!!
DG PE_tb.dg_content sends filter row 1 data = 0505056 @                  0
DG PE_tb.dg_content sends filter row 2 data = 030405a @             3000000
DG PE_tb.dg_content sends filter row 3 data = 050200e @             6000000
DG PE_tb.dg_content sends ifmap data = 00000000000000001110001110000 @              9000000
DG PE_tb.dg_content sends ifmap data = 00000000000000000001110010001 @             12000000
DG PE_tb.dg_content sends ifmap data = 00000000000000001110001110000 @             15000000

========================= Congratulations =========================
Golden value for L1_residue_t1 = 000000010110 and L1_out_spike_t1 = 0
DB PE_tb.db_content received L1_residue_t1 = 000000010110 and L1_out_spike_t1 = 0 @             27000000
========================= Congratulations =========================

DG PE_tb.dg_content sends ifmap data = 00000000000000000001110010001 @             30000000

========================= Congratulations =========================
Golden value for L1_residue_t1 = 000000010110 and L1_out_spike_t1 = 0
DB PE_tb.db_content received L1_residue_t2 = 000000100111 and L1_out_spike_t2 = 0 @             35000000
========================= Congratulations =========================

DG PE_tb.dg_content sends ifmap data = 00000000000000001110001110000 @             38000000
$finish called from file "./PE_tb.sv", line 258.
$finish at simulation time             40000000
          V C S   S i m u l a t i o n   R e p o r t
Time: 40000000 fs
CPU Time:      0.290 seconds;       Data structure size:   0.1Mb
Mon Apr 14 22:37:35 2025
CPU time: .821 seconds to compile + .405 seconds to elab + .376 seconds to link + .528 seconds in simulation
[ychiu078@viterbi-scf2 sim]$
```

Figure 7. Correct PE test result

```
and may be used and disclosed only as authorized in a license agreement
controlling such use and disclosure.

The design hasn't changed and need not be recompiled.
If you really want to, delete file simv.daidir/.vcs.timestamp and
run VCS again.

Chronologic VCS simulator copyright 1991-2016
Contains Synopsys proprietary information.
Compiler version L-2016.06-1_Full64; Runtime version L-2016.06-1_Full64;  Apr 14 23:26 2025
Start simulation!!!
DG PE_tb.dg_content sends filter row 1 data = 0505056 @                  0
DG PE_tb.dg_content sends filter row 2 data = 030405a @             3000000
DG PE_tb.dg_content sends filter row 3 data = 050200e @             6000000
DG PE_tb.dg_content sends ifmap data = 00000000000000001110001110000 @              9000000
DG PE_tb.dg_content sends ifmap data = 00000000000000000001110010001 @             12000000
DG PE_tb.dg_content sends ifmap data = 00000000000000001110001110000 @             15000000

xxxxxxxxxxxxxxxxxxxxx Comparison Fail xxxxxxxxxxxxxxxxxxxxx
Golden value for L1_residue_t1 = 000000010110 and L1_out_spike_t1 = 1
DB PE_tb.db_content received L1_residue_t1 = 000000010110 and L1_out_spike_t1 = 0 @             27000000
xxxxxxxxxxxxxxxxxxxxx Comparison Fail xxxxxxxxxxxxxxxxxxxxx

DG PE_tb.dg_content sends ifmap data = 00000000000000000001110010001 @             30000000

xxxxxxxxxxxxxxxxxxxxx Comparison Fail xxxxxxxxxxxxxxxxxxxxx
Golden value for L1_residue_t1 = 000000010110 and L1_out_spike_t1 = 1
DB PE_tb.db_content received L1_residue_t2 = 000000100111 and L1_out_spike_t2 = 0 @             35000000
xxxxxxxxxxxxxxxxxxxxx Comparison Fail xxxxxxxxxxxxxxxxxxxxx

DG PE_tb.dg_content sends ifmap data = 00000000000000001110001110000 @             38000000
$finish called from file "./PE_tb.sv", line 258.
$finish at simulation time             40000000
          V C S   S i m u l a t i o n   R e p o r t
Time: 40000000 fs
CPU Time:      0.280 seconds;       Data structure size:   0.1Mb
Mon Apr 14 23:26:41 2025
CPU time: .610 seconds in simulation
[ychiu078@viterbi-scf2 sim]$
```

Figure 8. Deliberate PE test failure with mismatch message

3. NOC

3.1 NOC Block diagram and Design

3.1.1 Tree Topology

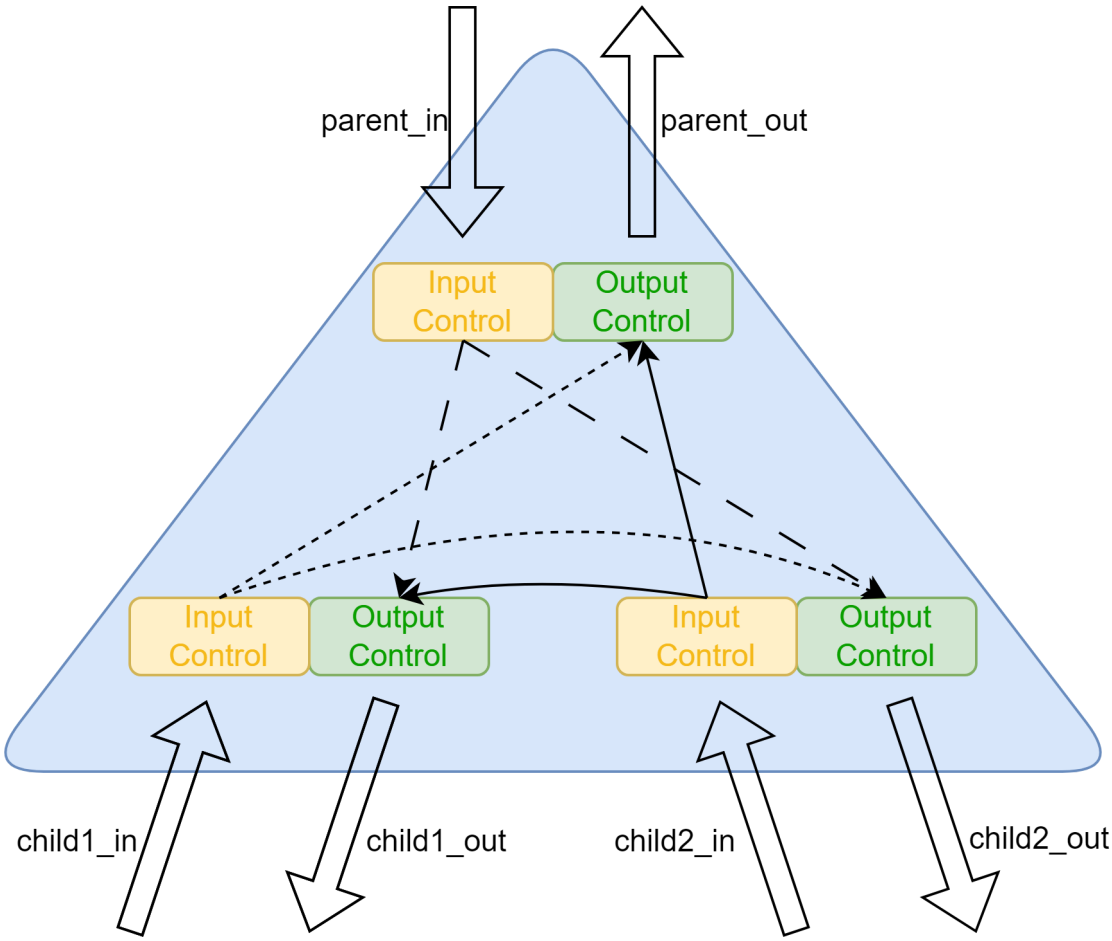

Figure 9. Router for Tree

The tree module instantiates seven router modules, each responsible for directing packets between parent and child nodes based on the destination address embedded in the packet.

The input_ctrl module is responsible for determining the appropriate output direction for an incoming packet based on its destination and source addresses. It supports routing logic for both parent and child nodes in a hierarchical tree-based NoC. After receiving a packet from the input channel, the module extracts the source address (addr) and destination address (dest) fields from the packet. The routing behavior then differs depending on the current node's role and level in the hierarchy: At the Root Level (LEVEL == 0): The packet is directly forwarded to the right child (out2). For Parent Nodes (IS_PARENT == 1): The module examines the destination bit at position WIDTH_dest - 1 - LEVEL, which corresponds to the current routing level. This bit determines the direction. For Child Nodes: A bitmask is generated based on the current level to isolate the most significant bits of the destination address. The masked_dest_bit is then compared with a specific bit of the source address (addr_bit) to decide the direction.

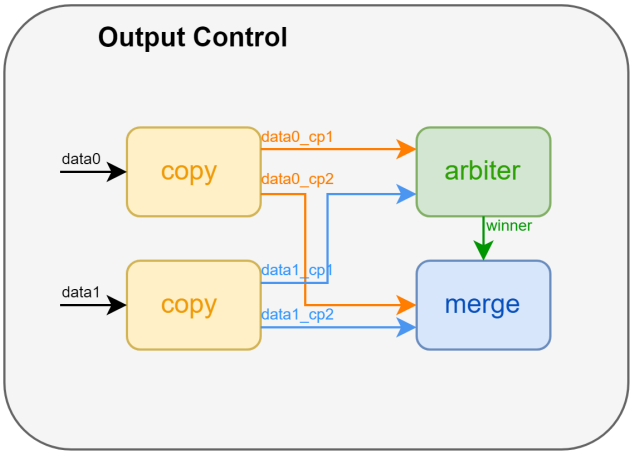

Figure 10. Output Control for Tree

Output Control is an arbiter that arbitrates among the inputs it receives as shown in figure 10.

3.1.2 Mesh Topology

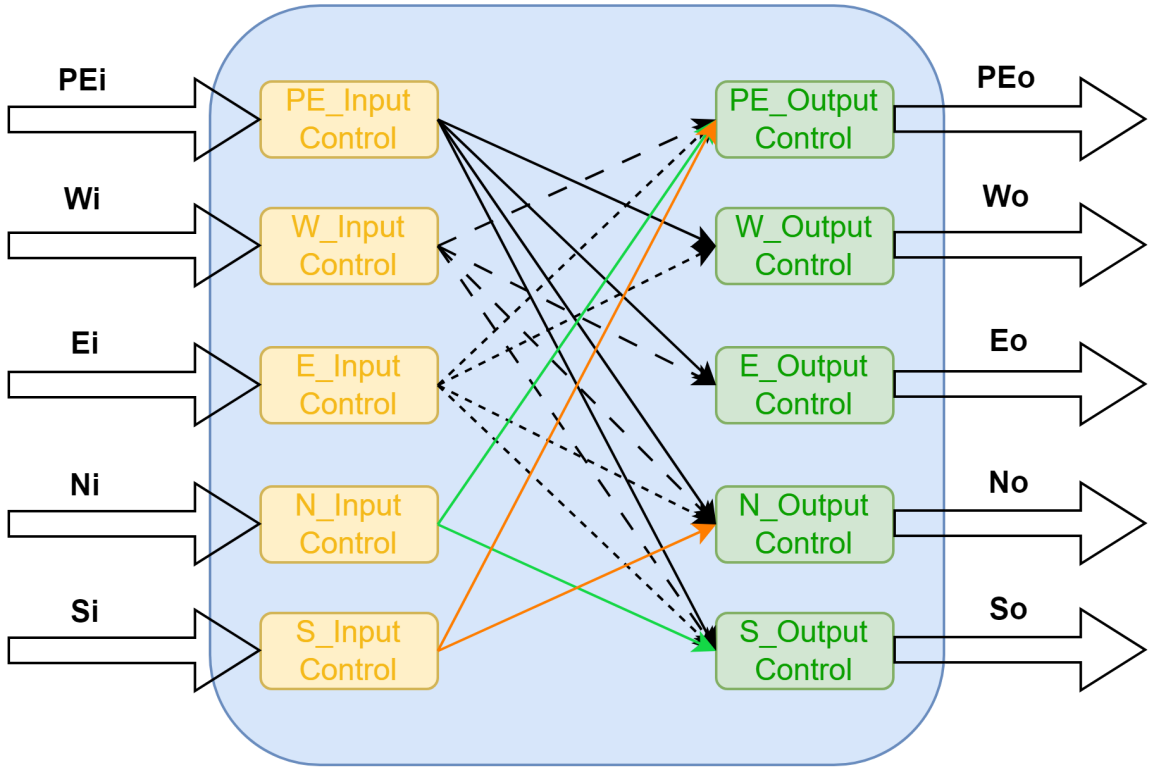

Figure 11. Router for Mesh

Input Control are buffers that receive packets from the input channel and decide which Output Control to send based on header information.

Our mesh uses XY routing algorithm. When PE input sends a packet into the mesh, it will first check x hop. The packet won’ t start travelling in y direction until it finishes x hops. This means that the router will not check x hop when a packet inputs from the North or South channel.

Output Control is an arbiter_merge that arbitrates among the two other input control’ s outputs it receives. As shown in the router diagram, West and East

Output Control use a 2-input arbiter that is the same as the tree router design. PE, North and South Output Control each receives packets from 4 Input Controls, so we use 4-input arbiters that are assembled by 3 2-input arbiters as our HW2 implementation.

3.2 Router Verification

3.2.1 Router Verification for Tree topology

Top module for the router is ./Baseline/router/EE552_NoC_tree/router_module/router.sv. The testbench file for this module is ./Baseline/router/EE552_NoC_tree/router_module/router_tb.sv. You can run the testbench by executing the following command in **ModelSim** under ./Baseline/router/router_mesh/:

```
$ do router.do
```

The testbench is designed to validate the functionality of a single router module by sending fixed packets to each of its input control modules. A total of six test cases are defined(parent to child1, parent to child2, child1 to parent, child1 to child2, child2 to parent, child2 to child1), with each test case representing a distinct routing scenario where a packet traverses through the internal logic of the router and reaches one of the output control modules .

Each output control module is connected to an external data bucket, which captures the transmitted packets. During simulation, both waveform signals and display messages are observed to track the flow of each packet. The correctness of the router is verified by checking whether the expected packet is successfully

received by the corresponding data bucket in each case. This ensures that all routing paths within the router are exercised and function as intended.

3.2.2 Router Verification for Mesh topology

Top module for the router is ./Baseline/router/router_mesh/design/router.sv. The testbench file for this module is ./Baseline/router/router_mesh/tb/tb_router.sv. You can run the testbench by executing the following command in **ModelSim** under ./Baseline/router/router_mesh/:

```
$ do router_mesh.do
```

Result file router_result.txt is located in ./Baseline/router/router_mesh/result/. Each input channel sends 5 packets. Total packet width is 15 bits and the most significant 5 bits are header as explained in section 1.1. Direction and hop count change in these packets to guarantee packet sending to all directions. Packet[5:7] is for marking the input directions, in which 000-100 means East, West, North, South inputs respectively. Packet[8:14] shows the sequence of packets sent from one input. In the PE output channel we aborted the header so only 10-bit content is sent to PE.

```
21  tb_router.dg_PEi starts sending 111011000000011 at   10.00 ns
22  tb_router.dg_Ei starts sending 001000000000100 at   12.00 ns
23  tb_router.dg_Wi starts sending 101000010000100 at   12.00 ns
24  tb_router.db_Wo receives packet 010010000000001 at time   12.00 ns
25  tb_router.db_Eo receives packet 110010010000001 at time   12.00 ns
26  tb_router.dg_PEi starts sending 000011000000100 at   14.00 ns
27  tb_router.db_PEo receives packet 0000000000 at time   14.00 ns
28  tb_router.db_No receives packet 000000010000000 at time   14.00 ns
29  tb_router.dg_Si starts sending 001000110000100 at   16.00 ns
30  tb_router.dg_Ni starts sending 011000100000100 at   16.00 ns
31  tb_router.db_Wo receives packet 010101000000001 at time   16.00 ns
32  tb_router.db_Eo receives packet 100001000000010 at time   18.00 ns
33  tb_router.db_PEo receives packet 0000000000 at time   18.00 ns
34  tb_router.db_No receives packet 100000000000000 at time   18.00 ns
```

Figure 12. Result File for Router Testbench

By examining the result file we can see that input packets are sent to correct outputs according to the XY routing algorithm. So the router passes the testbench.

3.3 Mesh Verification

Top module for the mesh is ./Baseline/router/router_mesh/design/mesh.sv. The testbench file for this module is ./Baseline/router/router_mesh/tb/tb_mesh.sv. You can run the testbench by executing the following command in **ModelSim** under ./Baseline/router/router_mesh/:

```
$ do tb_mesh.do
```

The mesh is written in a configurable way so that by assigning parameters ROW, COL, X_HOP_LOC, Y_HOP_LOC, it can generate a mesh of any size. X_HOP_LOC and Y_HOP_LOC are the last bit of x hop and y hop field.

This testbench implements a gather testbench, which means every node receives packets from all other nodes. We set the packet width as 20, in which the first 2 bits are direction bits, the Packet[2:4] and Packet[5:7] are for x hop and y hop respectively. Packet[8:11] and Packet[12:15] record source and destination node number. The last 4 bits show the packet sequence number of one source destination pair.

By changing ROW and COL of the top module in tb_mesh.sv, we tested both 2x3 mesh and 4x4 mesh, and result files are in ./Baseline/router/router_mesh/gather_result_2x3 and ./Baseline/router/router_mesh/gather_result_4x4 folders. Each txt file keeps the packets one node received in the format of Fig.10. There are three hex numbers in Packet value, they represent source node, destination node, and packet sequence

as previously explained. We can see that all nodes are able to receive 3 packets from each of all other nodes. This verifies the functionality of the mesh.

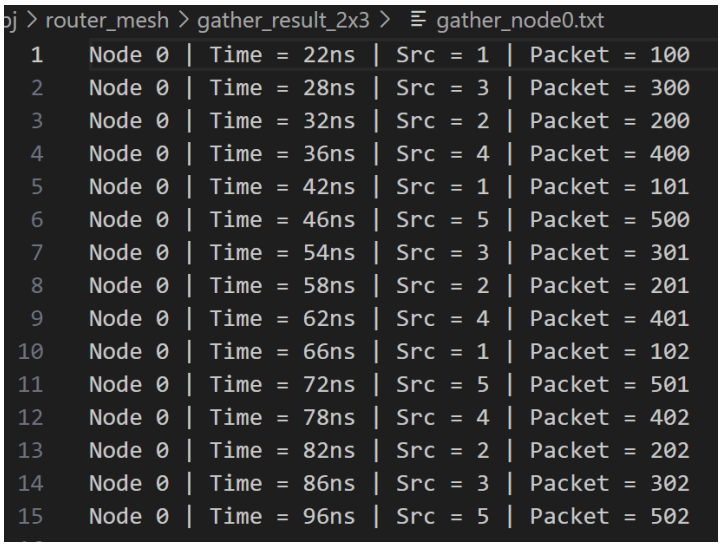

Figure 13. Result File for Mesh Testbench

4. Works to be done

As we move forward, there are several enhancements and pending tasks that we aim to address to complete and further optimize our PE and NoC design:

4.1 Gate-Level Design

We have not yet implemented any gate-level modules in our current setup. One of our next goals is to **optimize parts of the MAC (Multiply-Accumulate) unit at the gate level** to improve performance and gain a better understanding of its hardware characteristics.

4.2 Tree Topology for NoC

While we currently use a mesh topology for the NoC, we plan to design and implement a Tree-Based Topology to compare performance and scalability. This tree topology is still under development and has **not yet passed the testbench**.

4.3 Integration of PE and NoC

Currently, the PE module and the NoC are verified independently. We need to **integrate the PE with the NoC** to validate end-to-end data movement, timing, and functional correctness across the full processing pipeline.

4.4 Configurable NoC Design

Our overarching goal is to develop a **configurable NoC**. While the baseline version is functional, we aim to enhance configurability by:

- Supporting **configurable filter sizes** to accommodate different SNN layer needs.
- Supporting **configurable input feature map (ifmap) sizes** for more adaptable and reusable hardware blocks.