Asynchronous Spiking Neural Network on a Network-on-Chip

Final 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

Contents

1. Introduction

In the baseline design, our goal is to build a PE capable of producing one convolutional output based on a 4×4 ifmap and a 3×3 filter map, connected through a 6-node Tree NoC. In the advanced design, we introduce configurability for both the ifmap and filter map sizes, while maintaining the PE's ability to generate a single convolutional output. To enhance NoC throughput, we adopt a 4×4 mesh structure to better handle complex data flow among PEs, memory units, and control logic.

2. Baseline Design

2.1 Top Level Structure

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

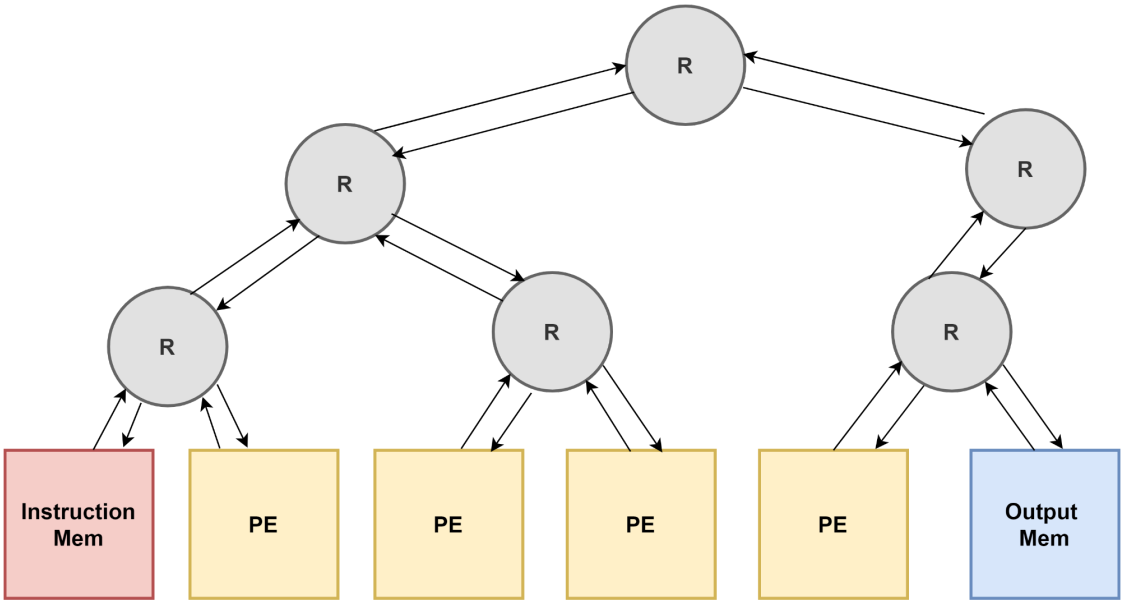

Figure 1. Tree topology

2.2 Packet Format

2.2.1 Packet Header Format

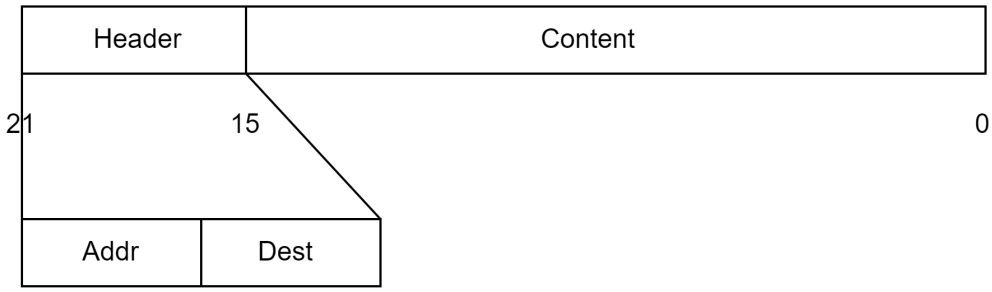

Figure 2. Packet organization before entering the PE for the tree topology

- Packet[21:16]: the header for packet transmission among 2x3 mesh routers
    - Packet[18:16]: These 3 bits represent the packet's destination address 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.

2.2.2 Packet Content Format

When the packet enters PE, header bits are aborted, and only the 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 3. Content organization after entering 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.

2.3 Tree Topology

2.3.1 Router Design

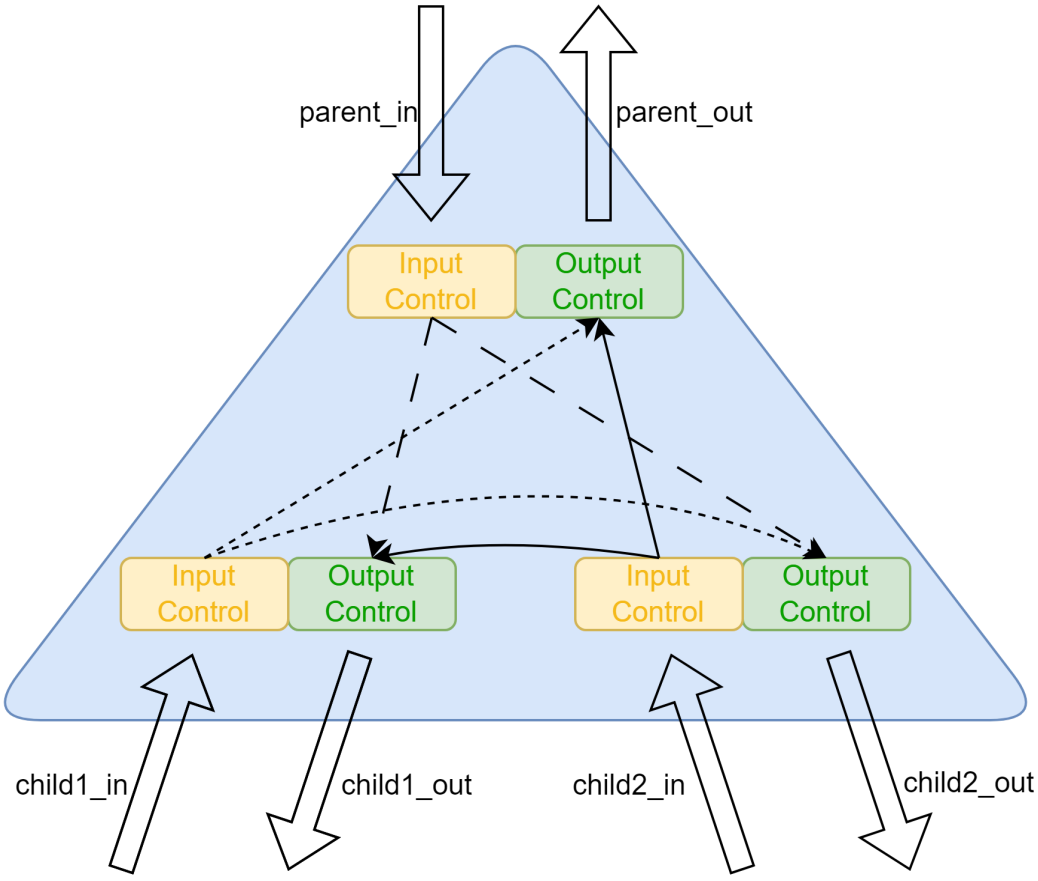

Figure 4. Router for Tree

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.

In the **input_ctrl** module, the packet routing logic is determined based on the current tree LEVEL and whether the node is a parent or child. When **LEVEL is 0**, all incoming packets are directly forwarded to **out2** without any further checks. When **LEVEL is not 0**, the behavior splits into two cases. If the node is a **parent (IS_PARENT == 1)**, it extracts the destination bit corresponding to the current LEVEL from the packet's destination address. If this bit is 0, the packet is routed to **out1** (left child); if the bit is 1, the packet is routed to **out2** (right child). On the other hand, if the node is a **child (IS_PARENT == 0)**, it computes a mask based on LEVEL to extract the higher-order bits of the destination and compares this masked destination with its own masked address. If the masked destination does not match the masked address, the packet is routed to **out1** (parent). If the masked destination matches the masked address, the packet is routed to **out2** (sibling). Additionally, if the destination exactly equals the node's address (dest == addr), the packet is ignored, which typically indicates that the packet has reached its final destination. This unified decision-making logic ensures that packets are properly forwarded through the tree topology based on hierarchical addressing and routing rules.

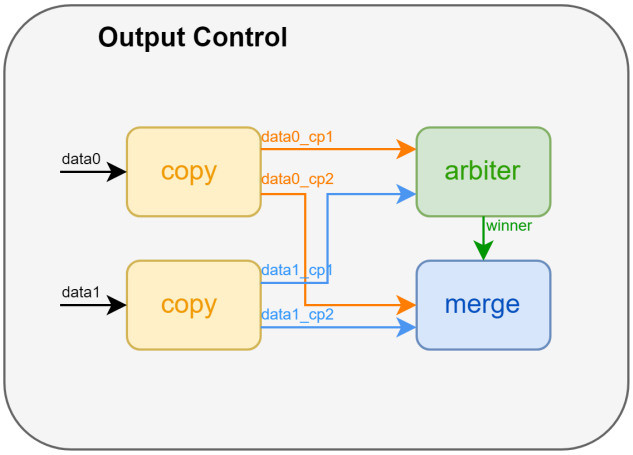

Figure 5. Output Control for Tree

Output Control is an arbiter that arbitrates among the inputs it receives, as shown in Figure 5.

2.3.2 Gate-Level Implementation

We chose to adopt a gate-level implementation for the input controller. The gated input_ctrl module (input_ctrl_gate) is designed to control packet forwarding in a tree-based NoC while introducing handshaking and gating mechanisms for improved power efficiency and robust data flow control.

Upon receiving a packet through the `click_controller`, which handles 4-phase handshake signaling with the upstream router（shown in Figure 6）, the module extracts the destination and current node addresses. The output direction decision logic, identical to the behavioral `input_ctrl`, determines whether the packet should be routed to out1 or out2.

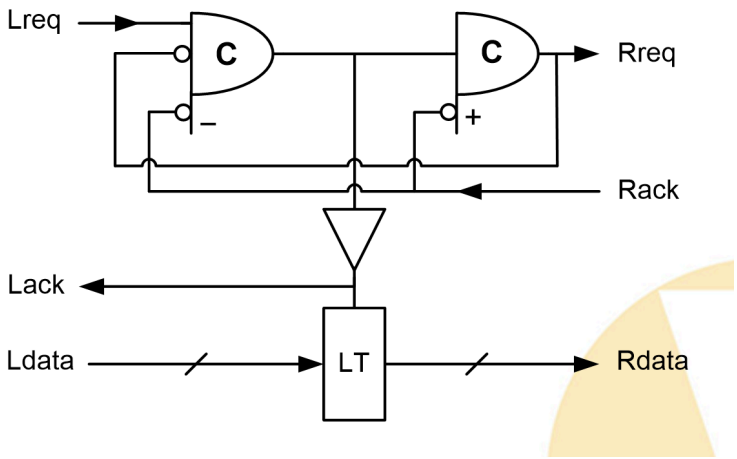

Figure 6. 4 phase controller[1]

Instead of directly transmitting packets, this gated version generates a selection signal (`sel_rreq`) that activates either out1_req or out2_req based on the routing decision(shown in Figure 6). These gated requests are connected to dedicated buffer_channel modules for each output port. Each `buffer_channel` handles local buffering, manages handshake acknowledgment (`bAck`), and issues the final Send command to the downstream channels once the handshake is complete.

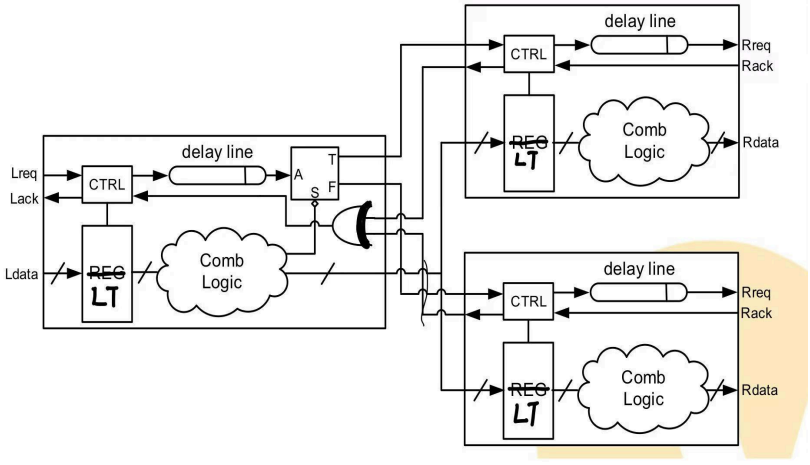

Figure 7. split-like gate-level input_ctrl implementation[2]

In addition, an internal acknowledgment signal (`rack`) monitors the status of both output channels and communicates with the upstream `click_controller` to complete the data transfer cycle.

2.3.3 Tree Verification

Since we directly integrated the tree into the top-level implementation, I have removed the standalone tree testbench and replaced it with a top-level testbench. The verification of the tree can now be demonstrated through the verification of the top module.

2.4 Processing Element(PE)

2.4.1 PE Block Diagram

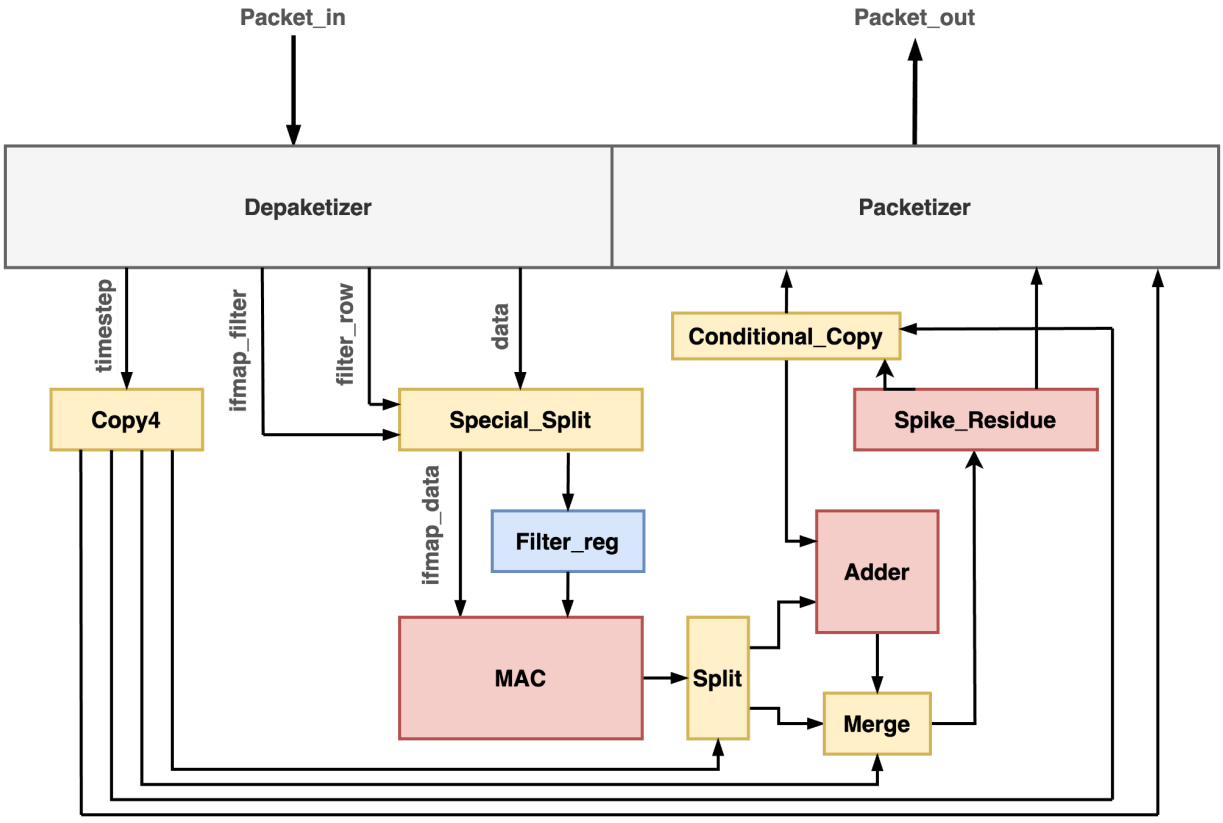

Figure 8. PE block diagram

2.4.2 PE Design Description

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.

2.4.3 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:

*The file organization of our PE design can be found in the path: /baseline_design/mesh_noc/PE_mesh/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 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 Testbenches

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 is 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 in 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 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 9. 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 10. Deliberate PE test failure with mismatch mess

2.5 Top Level Verification

In the Top Module Verification process, we use a Python script (`./Tree_noc/top_tree/test_scnn.py`) to generate the golden model results, which are stored in the `./Tree_noc/top_tree/scnn_script/` directory. The randomly generated input data is saved to `./Tree_noc/top_tree/scnn_script/send_values_bin.txt`. The top module testbench fetches these input values from the `input_buffer`, transmits them through the NoC, and performs the computations within the PE nodes. The computed results are then passed to the `output_buffer` and written to `./Tree_noc/top_tree/scnn_script/received_values.txt`.

For verification, the top 12 bits (residue) of the received results are compared against the golden model outputs stored in `./Tree_noc/top_tree/scnn_script/L1_residue_t1.txt` and `./Tree_noc/top_tree/scnn_script/L1_residue_t2.txt`.

As shown in the figure below, the residue values perfectly match the golden model results, demonstrating the correctness of the implementation.

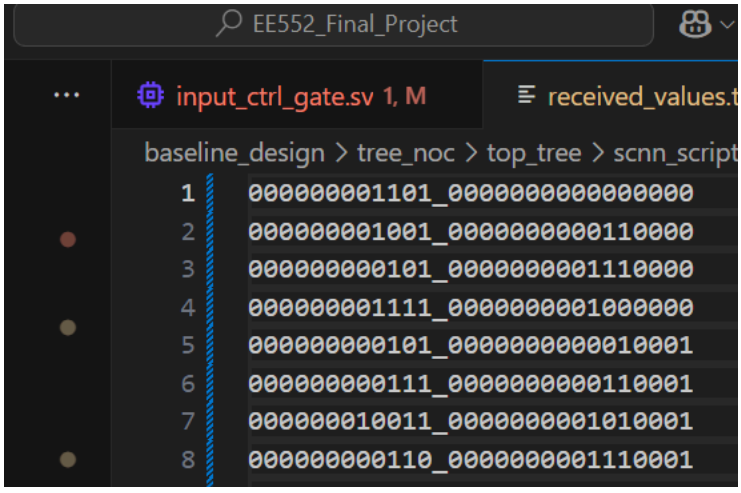

Figure 11. Result File for Tree Top module Testbench

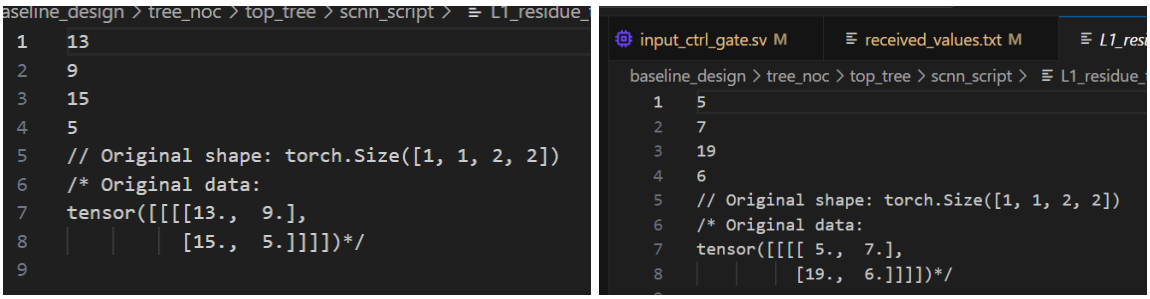

Figure 12. Golden Model results

As shown in the output(Figure 11), the results — 13, 9, 5, 15, 5, 7, 19, and 6 — perfectly match the golden model results displayed in the golden model results(Figure 12).

You can run the testbench by executing the following command in **ModelSim** under `./Tree_noc/top_tree/`:

```
$ do top.do
```

3. Advanced Design

3.1 Top Level Structure

We found that the fixed filter and Ifmap size limit the functionality of our baseline implementation, so we moved forward to support a flexible filter and ifmap size. Our design allows filter size from 2x2-5x5, and ifmap size up to 11x11.

Unlike the baseline design, a PE may be assigned multiple data sets for convolutional outputs. Since PE is only able to process one set of data at a time, we added an acknowledgement mechanism so that a new set of data won't be sent until PE finishes calculation and sends an ack packet back to the input control.

We noticed that each combination of filter and ifmap size requires a different instruction set to control the data flow among processors. Even more, when hardware implementation changes, instructions must be changed again. This adds great pressure to the programmer. Therefore, we developed a Control Unit (CU) that automatically generates instruction sets for each PE so that the programmer only needs to input size and data without knowing the actual hardware implementation.

To further improve the throughput of the NoC, we applied a 4X4 mesh to support more complicated data flow between the PE, memories, and CU, as shown in the following diagram. Input instructions enter the system through router R12, then CU generates instructions and sends them to PEs through the mesh network. When a PE calculation finishes, it returns an ack packet to the control unit and outputs the result to the output memory at R3.

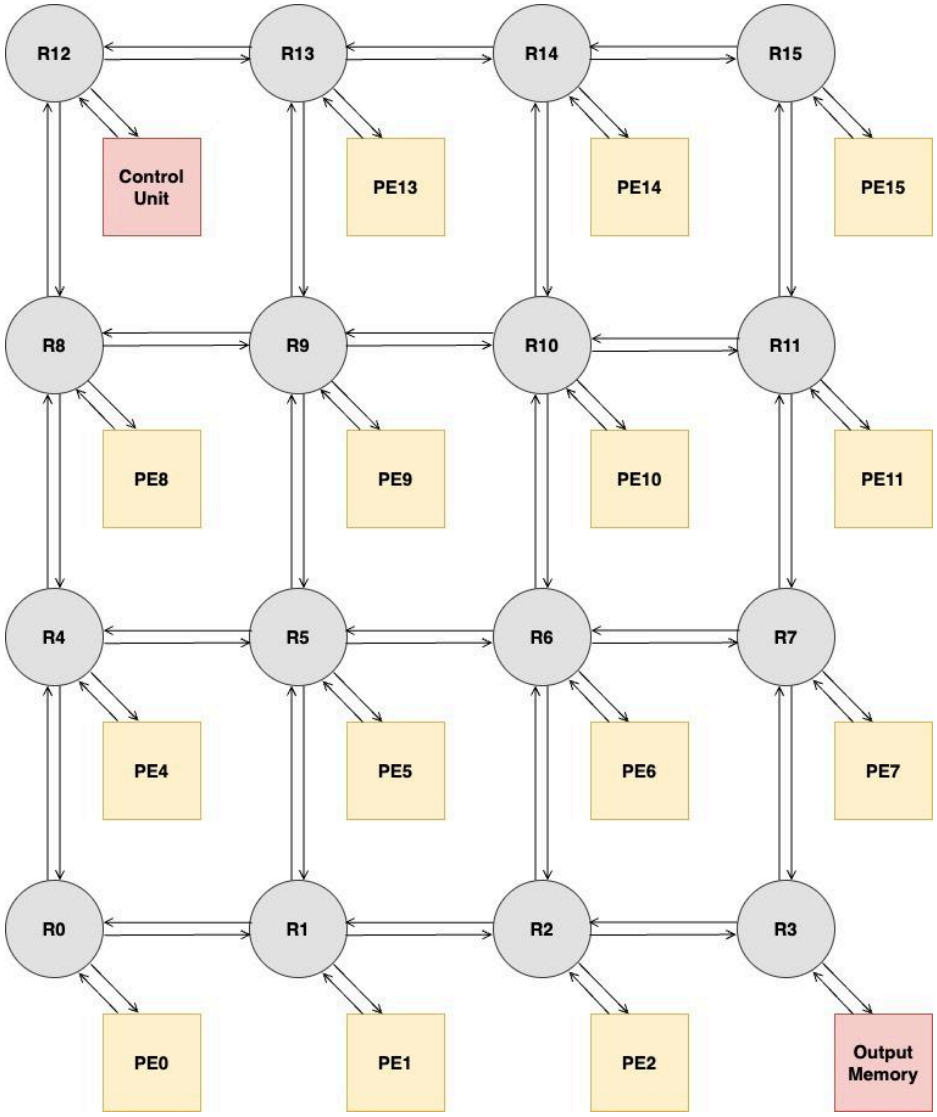

Figure 13. 4x4 Mesh SCNN

3.2 Packet Format

3.2.1 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 14. Packet organization before entering to PE for the 2x3 mesh topology

Packet[4:0]: the header for packet transmission among 2x3 mesh routers

- Packet[1:0]: direction that the 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 the x-hop and y-hop fields. Also, shifters for changing hop values are simpler than implementing subtractors.

3.2.2 Packet Format for PE

```
// ****************************** Packet format for PE************************************
//  filter packet format:
//  Address    [5*FILTER_WIDTH+4:5]         [4:2]       [1]                [0]
//             data                         filter_row  ifmap(0)_filter(1)  timestep
//  data format for ifmap data:
//  Address    [5*FILTER_WIDTH+4:5*FILTER_WIDTH-20]    [5*FILTER_WIDTH-21:7]    [6:5]
//             ifmap data                               conv_loc                 size
// ************************************************************************************
```

Figure 15. Packet organization after entering to PE

When a packet enters PE, the header field is dispensed, and the organization for the rest of the content is shown above. In our design FILTER_WIDTH= 8.

- Packet[0] is the timestep for the input ifmap data.
- Packet[1] marks whether this packet is sending ifmap or filter information.
- Packet[4:2] tells which row of filter data this packet contains.
- Packet[44:5] is the field for filter data. When the packet is an ifmap packet(packet[1] = 0), this field has a different organization as described in the following.

- Packet[6:5] represents filter size. 2’ b00-2’ b11 stands for 2x2-5x5, respectively.
- Packet[19:7] marks which location of the convolutional output this packet is sending the ifmap data for. Packet[18:13] and Packet[12:7] resemble y and x coordinates.
- Packet[44:20] stores ifmap data.

3.2.3 Packet Format for CU

```
// ********************************************** Input Filter Packet Format*****************************
//  Address    [5*FILTER_WIDTH+4:5]     [4:3]       [2]         [1]                [0]
//             filter_data              filter_size timestep    ifmap(0)_filter(1)  ack(0)_input(1)
// ********************************************************************************************************


// ********************************************** Input Ifmap Packet Format*****************************
//  Address    [5*FILTER_WIDTH+4:9]     [8:3]       [2]         [1]                [0]
//             ifmap_data               ifmap_size  timestep    ifmap(0)_filter(1)  ack(0)_input(1)
// ********************************************************************************************************


// ********************************************** Input Ack Packet Format*******************************
//
//  Address    [5*FILTER_WIDTH+4:5]     [4:1]       [0]
//             0s                       PE_node     ack(0)_input(1)
// ********************************************************************************************************
```

Figure 16. Packet format for CU

CU receives external input and ack from PE and automatically generates instructions for each PE according to the filter and ifmap size.

- Packet[0] = 1 marks the packet from external input, while Packet[0] = 0 marks the ack packet from PE after they finish calculating a convolutional output.
- For external input, Packet[1] = 1 means this is a filter packet, and Packet[4:3] tells the filter size as mentioned in the previous section. Packet[44:5] is for 1 row of filter data. Filter data must be sent in the sequence of row1, row2, etc.

- For external input, Packet[1] = 0 means this is an ifmap packet. Packet[2] and Packet[8:3] inform the timestep of the ifmap data and the ifmap size, respectively. Packet[44:9] contains 36-bit ifmap data. If the field is not enough, the remaining data is filled in another packet. The Ifmap packet must be sent in sequence, also.
- Ack packet contains PE node in Packet[4:1].

3.3 Mesh Topology

3.3.1 Router Design

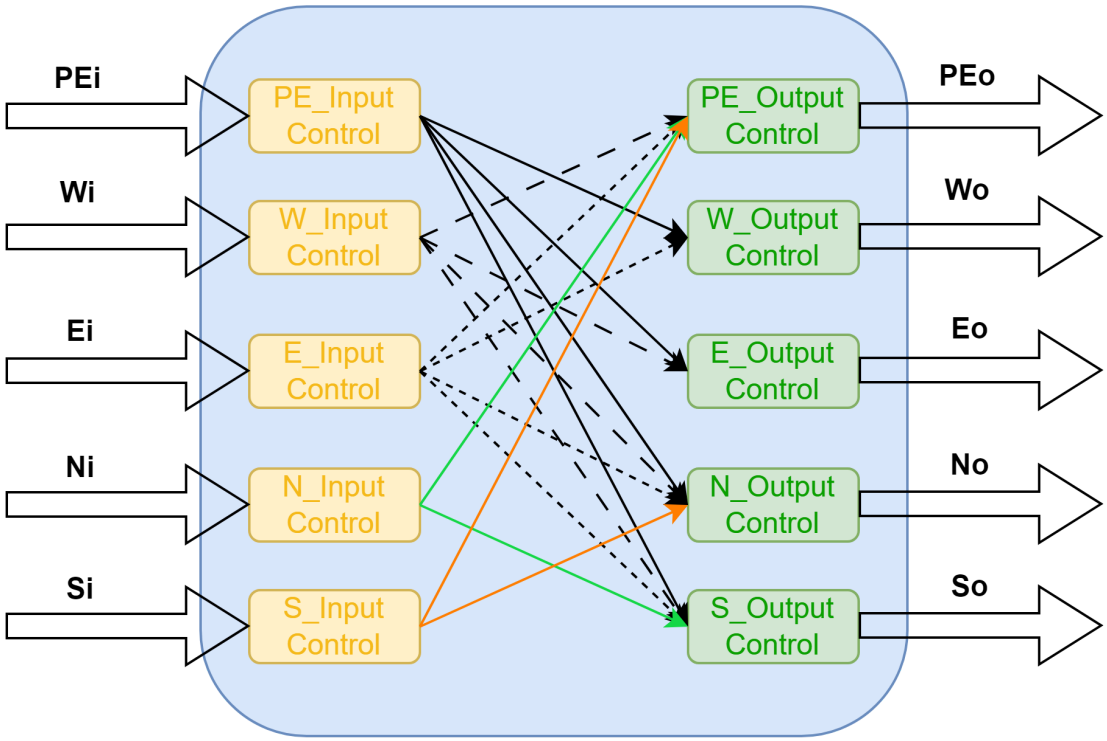

Figure 17. Router for Mesh

This is the diagram for each router in the mesh. Input Controls are buffers that receive packets from the input channel and decide which Output Control to send based on the header information.

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

An Output Control is an arbiter_merge that arbitrates among the two other input controls' outputs that 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 receive packets from 4 Input Controls, so we use 4-input arbiters that are assembled by 3 2-input arbiters as our HW2 implementation.

3.3.2 Router Verification

Root path for router and mesh verification is ./baseline_design/mesh_noc/ router_mesh. All design files are included in the design/ folder, and the top module for the router is router.sv. The testbench file for this module is ./baseline_design/mesh_noc/ router_mesh/tb/tb_router.sv. You can run the testbench by executing the following command in **ModelSim** under ./baseline_design/mesh_noc/ router_mesh:

```
$ do router_mesh.do
```

Result file router_result.txt is located in ./baseline_design/mesh_noc/router_mesh/result/. Each input channel sends 5 packets. Total packet width is 15 bits, and the most significant 5 bits are the header as explained in section 3.2.1. 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 that 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 18. Result File for Router Testbench

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

3.3.3 Mesh Verification

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

```
$ do tb_mesh.do
```

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

This testbench implements a gather testbench, in which every node receives 3 packets from each of the other nodes. We set the packet width as 20, in which the first 2 bits are direction bits, the Packet[2:7] are for x-hop and y-hop. Packet[8:11] and Packet[12:15] record the source and destination node numbers. The last 4 bits show the packet sequence number of one source destination pair.

ROW	2
COL	3
X_HOP_LOC	3
Y_HOP_LOC	4

Table 1. Parameter settings for 2x3 Mesh gather testbench

ROW	4
COL	4
X_HOP_LOC	4
Y_HOP_LOC	7

Table 2. Parameter settings for 4x4 Mesh gather testbench

By changing parameters of the top module in tb_mesh.sv as shown in Table 1 and Table 2, we tested both 2x3 mesh and 4x4 mesh, and the result files are stored in

./Baseline/router/router_mesh/gather_result_2x3 and ./Baseline/router/router_mesh/gather_result_4x4 folders. Each text file records the packets one node receives. There are three hex numbers in the Packet value, representing the source node, the destination node, and the packet sequence as previously explained. We can see that all nodes can receive the assigned packet. This verifies the functionality of the mesh.

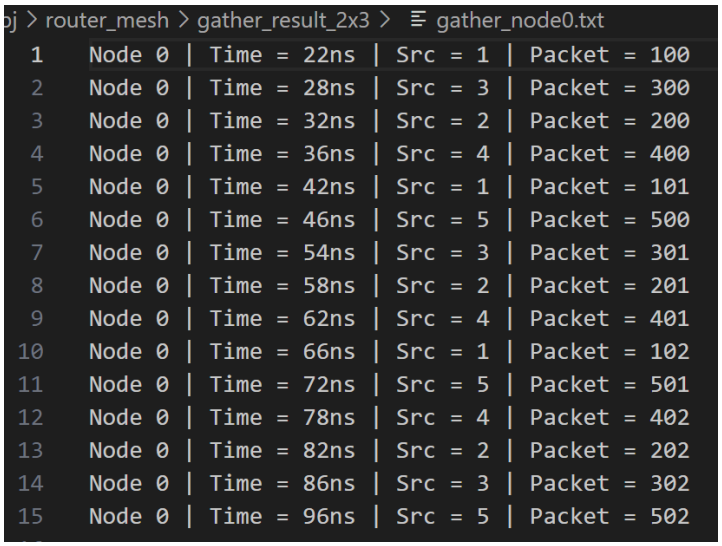

Figure 19. Result File for Mesh Testbench

3.4 Control Unit

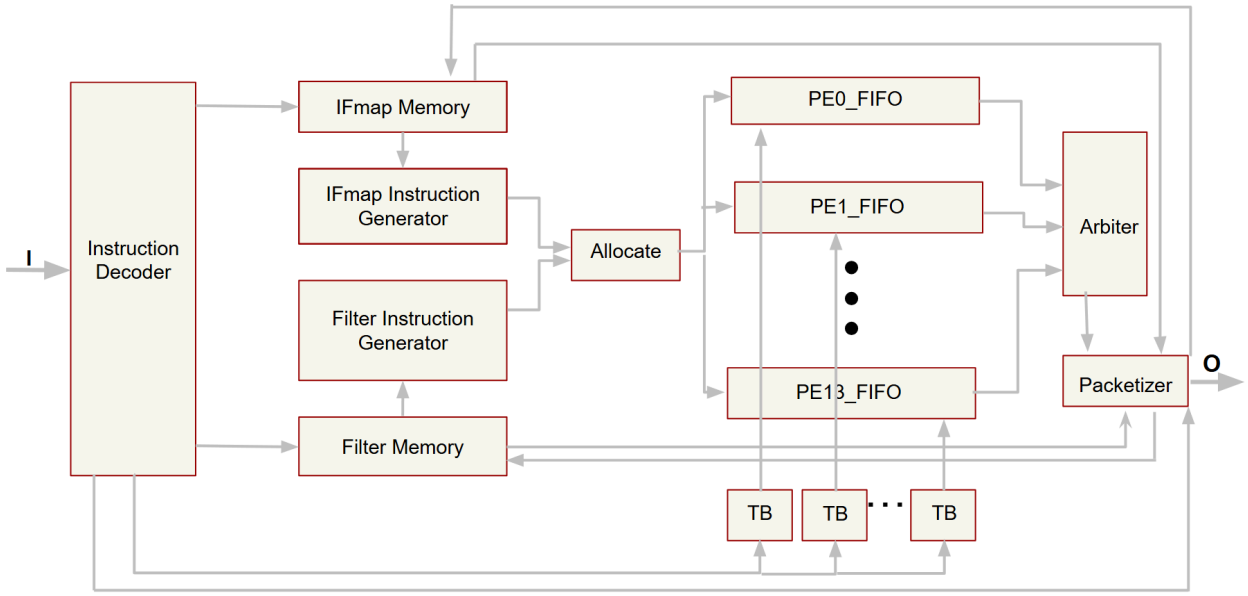

Figure 20. Control Unit Structure

In the CU diagram, the ports I and O are connected to the node 12 PEi and PEo ports of the mesh. There are 14 FIFOs, PE0_FIFO-PE13_FIFO, storing instructions for 14 PEs respectively. Programmers need to pay attention that filter data is sent first, and ifmap second. This is because we reuse filter data, so it only needs to be sent once, and for the rest of the time, we are sending ifmap data for different locations and timesteps. So the Allocate component is designed to pass ifmap instructions only after all filter instructions are sent to the FIFOs. To avoid carrying data all the way around, data is fetched only in the packetizer, and the data address is carried around instead. This helps reduce channel width and hardware cost in instruction generators, allocate, FIFOs, and the arbiter. Detailed data flow of CU is illustrated in the following diagrams.

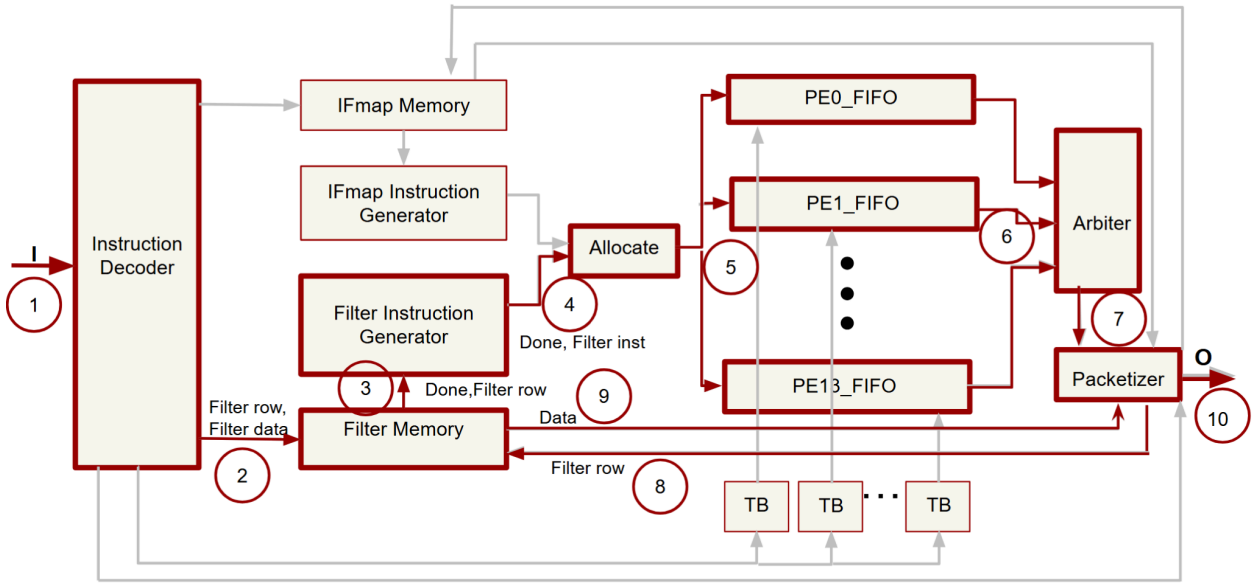

Figure 21. Data flow for filter instruction generation

When the filter instruction comes in, Instruction Decoder identifies the packet type and stores the row of data in Filter Memory. Each filter row data storage triggers the Filter Instruction Generator to generate a new instruction. Allocate will send this instruction to all FIFOs because all PEs need the same filter data. When all filter instructions are sent, the Done signal will inform Allocate to switch to the ifmap channel. For each FIFO, whenever there is a valid instruction, it will send the instruction at the read pointer along with the PE node number to the arbiter. Then the arbiter would choose one of the instructions and send it to Packetizer. Finally, the Packetizer requests the data from Filter Memory and packetizer all necessary information and sends the packet to the Mesh.

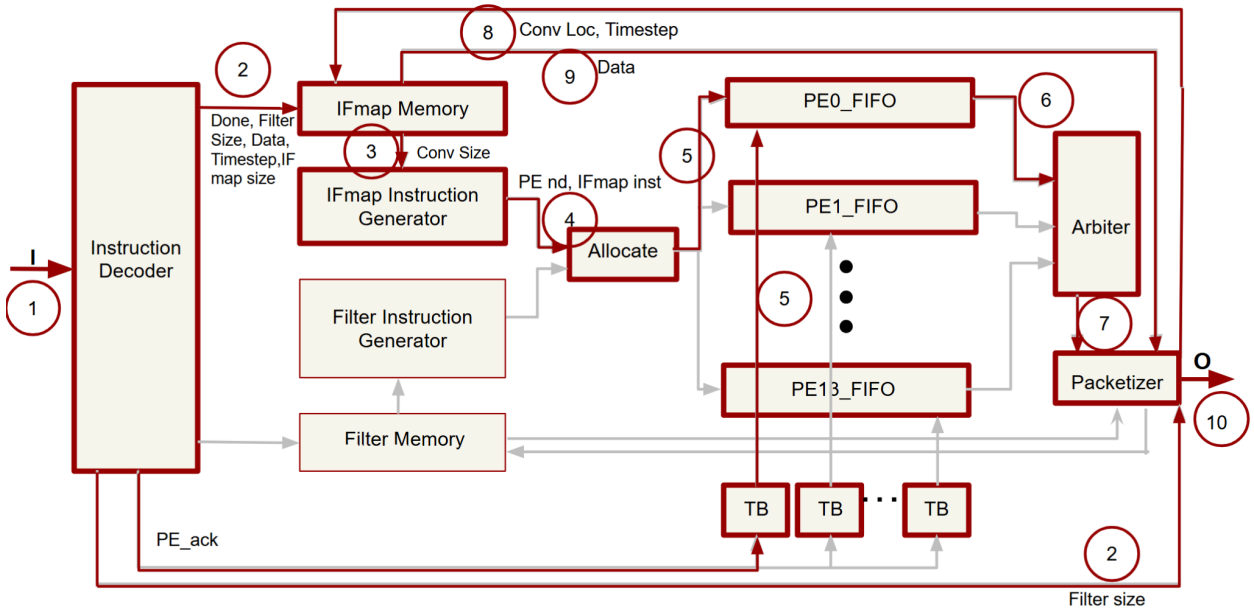

Figure 22. Data flow for filter instruction generation

The Ifmap instruction generation process is similar to filter's. What is different is that the instruction generator waits until all ifmap data is stored in memory to start the generation process according to the convolution output size. Each time, it sends a set of 2-timestep instructions of a PE to its FIFO through the Allocate component. In the FIFO, each set waits until an ack packet permits its release to the arbiter. For the initial set, there is a Token Buffer (TB) to initiate the sending process.

3.5 Top Level Verification

*The file organization of our advanced design can be found in the path: /advanced_design/top/tb/tb_tree.txt

(1) `test_scnn.py`

The Python script `test_scnn.py` at the root of the `tb/` folder does the following:

- **Generate Golden Reference Files (`/advanced_design/top/tb/scnn_script`)**
  It creates random convolution filters and input feature maps, performs spike-based convolution for two timesteps, and saves the resulting spike outputs and residues as golden reference files in the `scnn_script/` directory. These files represent the correct output against which the hardware design will be verified.
- **Generate Instruction Packets for the Control Unit (`/advanced_design/top/tb/scnn_script`)**
  Based on the generated input and filter data, the script formats binary packets `send_values_bin.txt` that encode instruction fields such as filter size, ifmap size, and data. These packets are intended to be read by the control unit in the SystemVerilog testbench using `$readmemb`, enabling accurate stimulus for hardware verification.
- **Support for Any Ifmap and Filter Size (2×2 to 5×5)**
  A key feature of this script is its flexibility—it dynamically adapts to different ifmap and filter sizes. When you run the script, it will prompt you with two input messages regarding the filter and ifmap size. This ensures that the

generated packets and reference outputs are always consistent with the specified configurations, making it a robust tool for verifying designs under varying input conditions.

(2) `Makefile`

This Makefile automates the simulation and verification workflow for the SCNN hardware design. It first compiles and runs the SystemVerilog testbench using Synopsys VCS, capturing the entire simulation process in `sim.log`. After simulation, it compares the design's output files (e.g., spikes and residues at two timesteps) stored in `sim_results/` against the golden reference files generated by the Python script in `scnn_script/`. Comment lines are stripped before comparison to ensure clean data matching. The Makefile reports whether each output matches or mismatches, helping quickly identify correctness. Additionally, a clean target is provided to remove generated files and reset the working directory for a fresh run.

(3) `/sim_results` — Simulation Results Files

This folder contains the simulation output results of our design.

- `sim.log`: logs the simulation process and runtime messages
- `timestep1_outspike.txt`, `timestep2_outspike.txt`: output spikes from the design at timestep 1 and 2
- `timestep1_residue.txt`, and `timestep2_residue.txt`: residue values from the design at timestep 1 and 2

These files capture the output spike and residue values generated by our design at each timestep and are used to compare against the golden reference results during functional verification.

(4) Verification Flow

To verify the SCNN hardware design, begin by specifying the desired input feature map (ifmap) and filter sizes in the `test_scnn.py` script. This script will generate the corresponding golden reference files and instruction packets tailored to the specified configuration. Once the data is generated, simply run `make` in the terminal. This command triggers the Makefile to compile and simulate the SystemVerilog testbench using **VCS**, and then automatically compares the simulation outputs against the Python-generated golden files. The flow provides immediate feedback on whether the design behaves correctly under the given input conditions.

```
$ cd /EE552_Final_Project/advanced_design/top/tb
$ python3 test_scnn.py
$ make
```

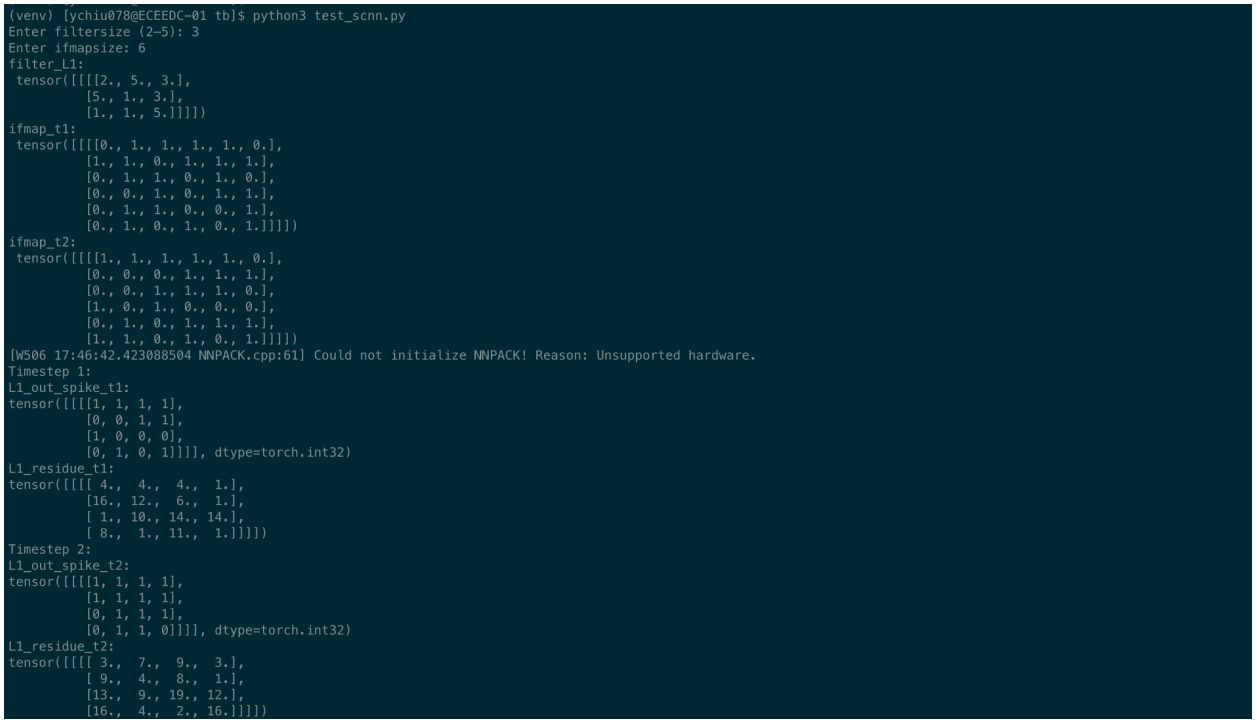

Figure 23. Python script result

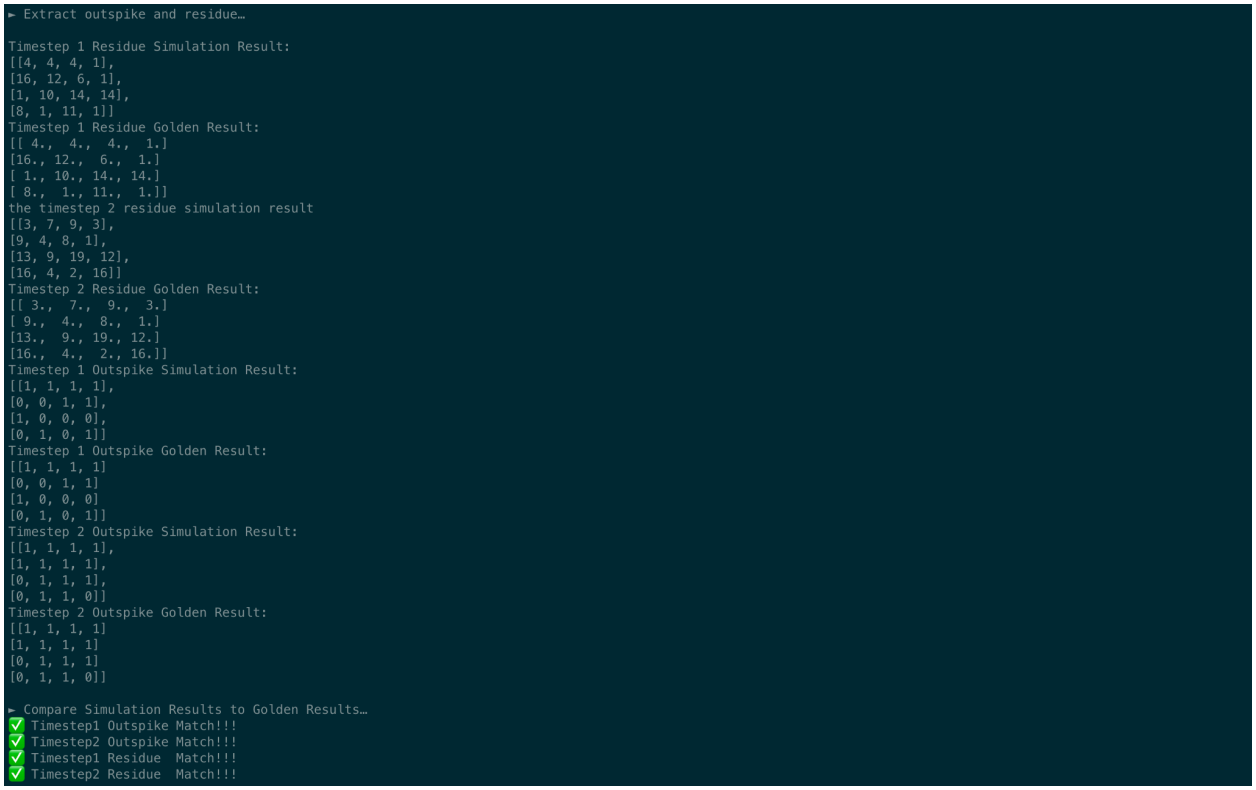

Figure 24. Simulation result

4. Design Analysis

Our modules generally applied 2ns forward latency and 2ns backward latency. However, this does not match real hardware performance because their combinational logic delays are different. Here we can only give a comparison of the performance.

For the baseline design, we implemented Tree and 2x3 Mesh as NoC. Mesh does not help improve performance because baseline SCNN is a pipelined dataflow that travels from input memory to PE, then from PE to output. Since the data travels in south or east directions only in a 2x3 Mesh, the north and west channel resources are wasted.

In the advanced design, we improved the functionality of the SCNN system by realizing configurable ifmap and filter size, and achieved higher throughput by using a 4x4 Mesh to handle complex dataflow. Also, the north and west channels of 4x4 Mesh are utilized for ack packages from PE to CU. The bottleneck lies in the packetizer of CU because it can only send 1 instruction even though more are ready in the instruction FIFOs, and it needs to request data from the Ifmap Memory or Filter Memory before sending the packet out. There is a trade-off between carrying data all along in the FIFO and fetching data only in the packetizer. The former needs much more hardware for storing data in every entry of the 14 FIFOs. This is not scalable when we want to have more PEs for calculation. The latter adds latency to the bottleneck process, so it slows down the whole system. A solution to this problem is to add one more packetizer to do data fetching and packetizing in parallel. This reduces the latency at the packetizing process and improves the throughput of the whole system. The cost is one more packetizer, an arbiter to select between 2 packetizer outputs, and dual read ports for Ifmap and Filter Memories.

Another issue slowing down the whole system is the ack mechanism, where a new set of ifmap data cannot be sent to PE before it finishes calculating the previous set. This mechanism was added out of the concern that PE can only process one set of data at a time, so the new set of data has to wait in the NoC and may cause a deadlock. However, the ack mechanism also adds a lot of waiting time in the CU. A solution is to reduce the number of waiting times. For example, we can expand PE's Ifmap data memory to hold 2 sets of data so that CU's instruction FIFO only needs to wait for an ack for every 2 sets of Ifmap instructions. The cost is relatively low since

ifmap data are 1-bit numbers and require 50 bits of extra storage, 2 timesteps of 5x5 filter size, for each PE.

Finally, we can still optimize the CU process. In the current design, filter data is sent to all 14 PEs even when the filter size is 2x2 or 3x3 filter size, so there are redundant filter instructions. Removing these instructions can reduce the latency of the SCNN process.

5. Citations

[1] EE 552 Lecture Slides, *Chapter 9 – Micropipelines and Bundled Data*, version 1.7, p. 18.

[2] EE 552 Lecture Slides, *Chapter 9 – Micropipelines and Bundled Data*, version 1.7, p. 16.