Network on Chip Spiking Neural Network
Architecture and microarchitecture Specification

Peiliang Du and Yifu Zhou

Professor Peter Beerel

EE552 - Asynchronous VLSI Design

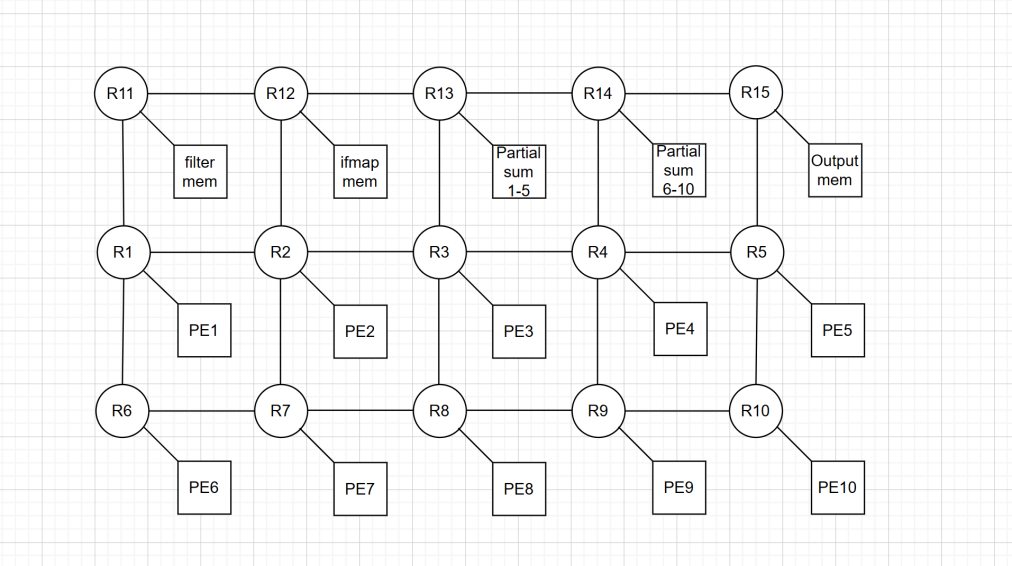
April 4, 2023

Viterbi School of Engineering

University of Southern California

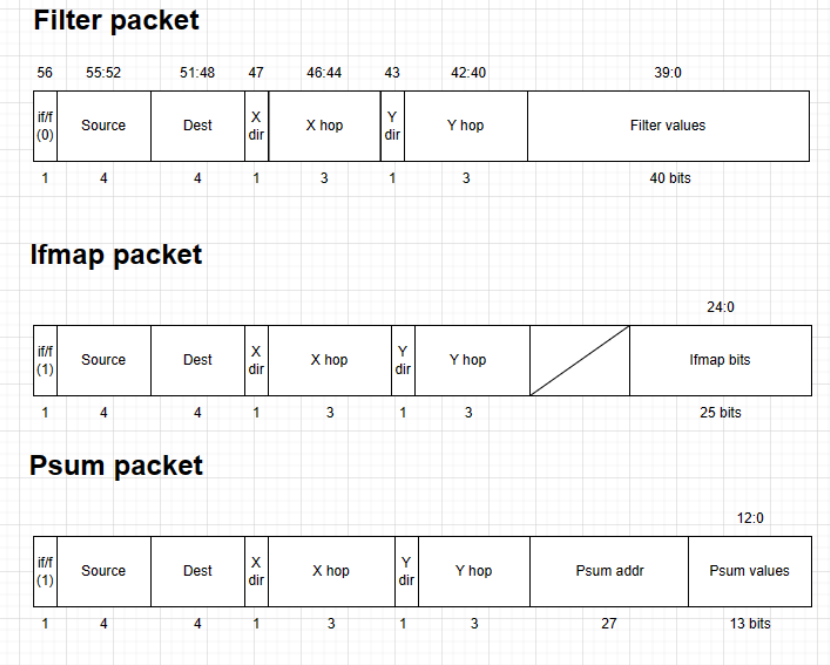
Introduction

NOC Mesh Layout



Our mesh spiking neural network consists of 10 convolution PEs, 2 partial sum PEs, an input feature map memory, a filter memory, and an output memory. In our design, PE 1-5 are responsible for calculating partial sums for timestep 1, while PE 6-10 are for timestep 2. In such an implementation, our design can generate partial sums for two timestep concurrently, and output them to corresponding partial sum modules, adding residue and subtracting threshold, and output to the memory eventually. In more detail, parietal sum 1 sends every entry of residue from timestep 1 to partial sum 2 which calculates its output spike. Memory is a likely bottleneck of the system. To ameliorate the cost of memory operation, the payload is set to 40 bits, allowing 5 input features or filters per packet. Furthermore, we forward filters from PE1-5 to PE6-10 and forward input filters between each timestep PE group. For example, PE 1 and PE 6 are using the same row of the filter data, we can forward the filter data to PE 6 right after PE 1 receives the filter from filter memory. PE 2 can forward the 2nd row of ifmap data to the PE1 after producing the partial sum output, since the ifmap data can be reused. In such a design, the ifmap memory only needs to forward the data once in every time step to the PEs, and the filter memory only needs to send the filter to PE 1 to 5 once, which tremendously improves the performance.We use X-Y direction bits and X-hop Y-hop fields to navigate the packets through mesh.

Data packet structure



NOC has 3 types of packets (57 bits each): filter, ifmap, and psum packets. A filter packet can hold an entire row of the filter data that's 8 bits each, and the ifmap packet also holds the entire row of the ifmap bits.

Format

- [56] if/f type:
  defines a packet whether it's ifmap or filter. The receiving PE will forward data based on the iff_type. Since the partial sum module only receives psum packets, the module doesn't care what the iff_type is.
- [55:52] 4-bit source:
  indicates where the packet is sent, whether it is from the memory or PE.
- [51:48] 4-bit dest:
  indicates which node the packet is heading toward. For receiving PEs, this 4-bit data is also used as PE id in the state machine, so that each PE operates accordingly.
- [47] X-direction [46:44] X-Hop value:
  indicates the X-direction, 1 for +X (East), and 0 for -X (West). 3-bit indicates the hop value in X-direction, and it will be manipulated by the router. An "000" means that the packet reaches its destination column.
- [43] Y-direction [42:40] Y-Hop value:
  indicates the Y-direction, 1 for +Y (North), and 0 for -Y (South). 3-bit indicates the hop value in Y-direction, and it will be manipulated by the router. An "000" means that the packet reaches its destination row.
- [39:0] 40-bit data:
  For the filter packet, the data field holds 5 filter values with 8 bits each. For the ifmap packet, the data field holds 25 input spike bits. In the psum packet, the 27-bit (don't necessarily use all 27 for address) psum address shows the memory location the psum value needed to be stored, and the 13-bit psum field stores the psum output.

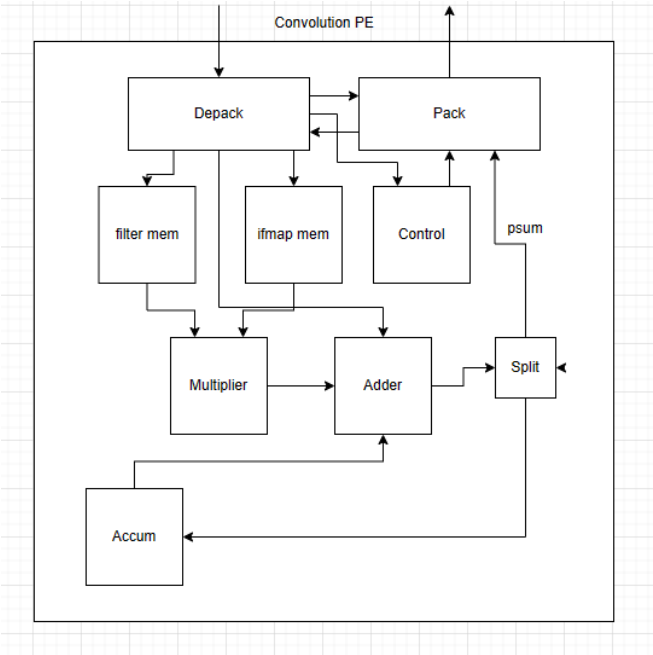
Convolution PE

Based on the convolution PE in the HW, we added a state machine, a depacketizer, and a packetizer to the convolution PE. The NOC mesh includes various processing elements, each processing element needs to know its location, and where it should forward the packet, so we added a state machine in the PE such that each PE can operate accordingly based on its PE id. When the PE receives the first packet, the destination address will be its PE id. Since each PE has a distinct distance to the destination, I create a table to record the forwarding process for each PE, and build a state machine based on it. For example, PE 2 needs to forward the filter data to PE 7 when it first receives the filter packet, then it starts convolution calculation and sends the psum value to the partial sum module at node 13. Finally, it forwards the ifmap data to the PE 1. Each PE will inject the X-direction, Y-direction, X hop, and Y hop values, and forward the packet.

PE forwarding table

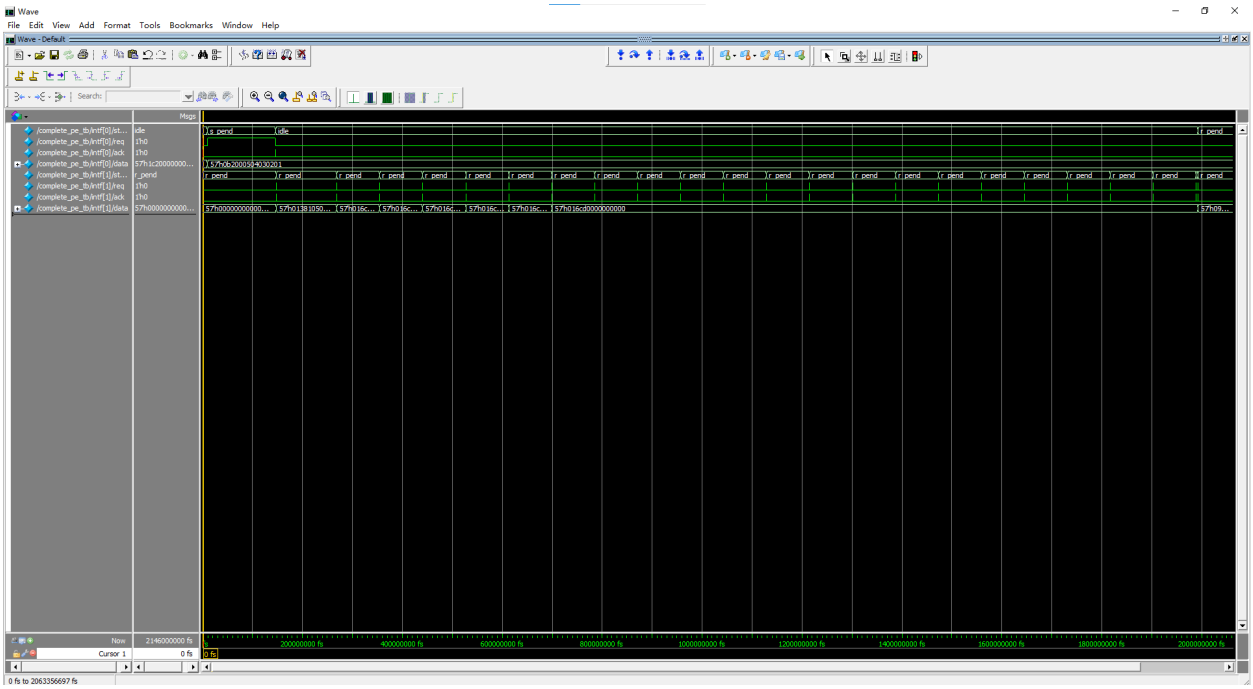
PE	R/S	dest addr	data type	x dir	y dir	x hop	y hop	filter forward	ifmap forward
1	R	1							
	S	13	psum	1	1	010	01		
	S	6	filter	0	0	000	01	1	0
2	R	2							
	S	13	psum	1	1	001	01		
	S	7	filter	0	0	000	01	1	1
3	R	3							
	S	13	psum	0	1	000	01		
	S	8	filter	0	0	000	01	1	1
4	R	4							
	S	13	psum	0	1	001	01		
	S	9	filter	0	0	000	01	1	1
5	R	5							
	S	13	psum	0	1	010	01		
	S	10	filter	0	0	000	01	1	1
6	R	6							
	S	14	psum	1	1	011	10		
								0	0
7	R	7							
	S	14	psum	1	1	010	10		
								0	1
8	R	8							
	S	14	psum	1	1	001	10		
								0	1
9	R	9							
	S	14	psum	1	1	000	10		
								0	1
10	R	10							
	S	14	psum	0	1	001	10		
								0	1

Convolution PE Micro-architecture



When the convolution PE receives a packet, the depacketizer needs to decode the iff_type, dest address, and the data. Let's use PE 2 as an example. The dest address will be treated as PE id in the depacketizer, if the iff_type received is 0 (filter), then the depacketizer sends a f_fordward signal to the packetizer to let it start receiving source, dest address, X-direction, Y-direction, X-hop, Y-hop values. In this case, PE 2 is forwarding filter to PE 7, and PE 7 is in the -Y direction of PE 2; source, dest address, X-direction, Y-direction, X-hop, Y-hop values will be concatenated with the filter value as following: source = 010 (2), dest = 0111 (7), x-dir = 0, y-dir = 0, x-dir = 0, x-hop = 000, y-hop = 001. After receiving both ifmap and filter, the PE will start convolution calculation and output 21 psums with psum addresses, and send the packet to the partial sum module located in node 13. When the psum packet is sent, the packetizer will send a signal to the depacketizer to request forwarding ifmap data, pack the new destination data, and forward to PE1.

Testbench waveform



Sample outputs:

```
//PE2
intf[0].Send('h1C200000000001F);
#10;
intf[0].Send('h0B2000504030201);
#2000;
```

Figure 1.

```
# Sending forwarding data
# iff type = 0
# data = 00000101000001000000001100000001000000001
# source =  2
# dest =  7
# x_dir = 0
# y_dir = 0
# x_hop = 0
# y_hop = 1
#
# packet = 00001001110000001000001010000010000000011000000100000001
```

Figure 2.

```
# Sending psum
# iff type = 0
# psum_out =   15
# psum_addr =         0
# source =  2
# dest = 13
# x_dir = 1
# y_dir = 1
# x_hop = 1
# y_hop = 1
#
# packet = 00001011011001101000000000000000000000000000000000000001111
```

Figure 3.

```
# Sending forwarding data
# iff type = 1
# data = 00000000000000000000000000000011111
# source =  2
# dest =  1
# x_dir = 0
# y_dir = 0
# x_hop = 1
# y_hop = 0
#
# packet = 01001000010001000000000000000000000000000000000000011111
```

Figure 4.

To test the functionality of the convolution PE module, I send several sample packets using hex to the interface 0. For example, in the first screenshot, ifmap data is 11111 followed by 20 0's, and the filter data is "1 2 3 4 5." PE 2 first receives an ifmap packet, then it receives a filter packet. So it will forward the filter data to PE 7 as shown in Figure 2. Then PE 2 starts convolution calculation, and sends 21 psum outputs to the partial sum module in node 13 as shown in Figure 3. Lastly, it forwards the ifmap data to PE 1 as shown in Figure 4. The output signal in the waveform (Figure 1) interface 1 follows the transcript as discussed above.

PE microarchitecture

```
module pe(interface packet_in, packet_out);
    parameter WIDTH = 8;
    parameter DEPTH_I = 25;
    parameter ADDR_I = 5;
    parameter DEPTH_F = 5;
    parameter ADDR_F = 3;

    parameter FL = 4;
    parameter BL = 2;

    parameter packet_width = 57;
    parameter addr_width = 4;
    parameter data_width = 40;
    parameter hop_width = 2;
```

```
module pe_depacketizer(interface packet_in, ifmap_in, ifmap_addr, filter_in, filter_addr, psum_in, start,
    iff_typeIntf, sourceIntf, destIntf, x_dirIntf, y_dirIntf, x_hopIntf, y_hopIntf, if_forwardIntf, f_forwardIntf, data_forwardIntf);
    parameter WIDTH = 8;
    parameter DEPTH_I = 25;
    parameter ADDR_I = 5;
    parameter DEPTH_F = 5;
    parameter ADDR_F = 3;

    parameter packet_width = 57;
    parameter addr_width = 4;
    parameter data_width = 40;
    parameter x_hop_width = 3;
    parameter y_hop_width = 2;


    parameter FL = 1;
    parameter BL = 1;

    logic [packet_width-1:0] packet_data;
    logic [data_width-1:0] if_data, f_data, data;
    logic [data_width-1:0] forward_data = 0;
    logic [x_hop_width-1:0] x_hop;
    logic [y_hop_width-1:0] y_hop;

    logic x_dir, y_dir;
    logic [addr_width-1:0] source, dest, f_source, f_dest, if_source, if_dest;
    logic iff_type;

    logic [WIDTH-1:0] filter_data, ifmap_data;

    logic [ADDR_F-1:0] addr_filter = 0;
    logic [ADDR_I-1:0] addr_ifmap = 0;

    logic filter_ready = 0;
    logic ifmap_ready = 0;
    logic f_forward = 0;
    logic if_forward = 0;
```

```
module pe_packetizer(interface f_forwardIntf, if_forwardIntf, data_forwardIntf, done, psum_out, iff_typeIntf, sourceIntf, destIntf, x_dirIntf, y_dirIntf, x_hopIntf, y_hopIntf, packet_out);
    parameter WIDTH = 8;
    parameter DEPTH_I = 25;
    parameter ADDR_I = 5;
    parameter DEPTH_F = 5;
    parameter ADDR_F = 3;

    parameter FL = 1;
    parameter BL = 1;

    parameter packet_width = 57;
    parameter addr_width = 4;
    parameter data_width = 40;
    parameter psum_width = 13;
    parameter x_hop_width = 3;
    parameter y_hop_width = 2;
    parameter psum_addr_width = 27;

    logic [packet_width-1:0] packet_data = 0;
    logic [data_width-1:0] data = 0;
    logic [psum_width-1:0] psum = 0;
    logic [psum_addr_width-1:0] psum_addr = 0;

    logic [x_hop_width-1:0] x_hop;
    logic [y_hop_width-1:0] y_hop;

    logic x_dir, y_dir;
    logic [addr_width-1:0] source, dest;
    logic iff_type = 0;

    logic f_forward;
    logic if_forward;
    logic don_e;

    int no_iterations = DEPTH_I - DEPTH_F + 1;
```

Depacketizer:

Packet_in: input packet.

ifmap_in: provide ifmap data to ifmap mem.

Ifmap_addr: provide ifmap address to ifrmap mem.

Filter_in: provide filter data to filter mem.

Filter_addr: provide filter address to filter mem.

Start: request the control to start.

Channels to forward headers and data: iff_typeIntf, sourceIntf, destIntf, x_dirIntf, y_dirIntf, x_hopIntf, y_hopIntf, if_forwardIntf, f_forwardIntf, data_forwardIntf

Packetizer: f_forwardIntf, if_forwardIntf, data_forwardIntf,

Done: received from control, shows convolution calculation is done.

Psum_out: convolution calculation output

Channels to received headers from depacketizer: iff_typeIntf, sourceIntf, destIntf, x_dirIntf, y_dirIntf, x_hopIntf, y_hopIntf

Packet_out: send the packet out.

Works to be done:

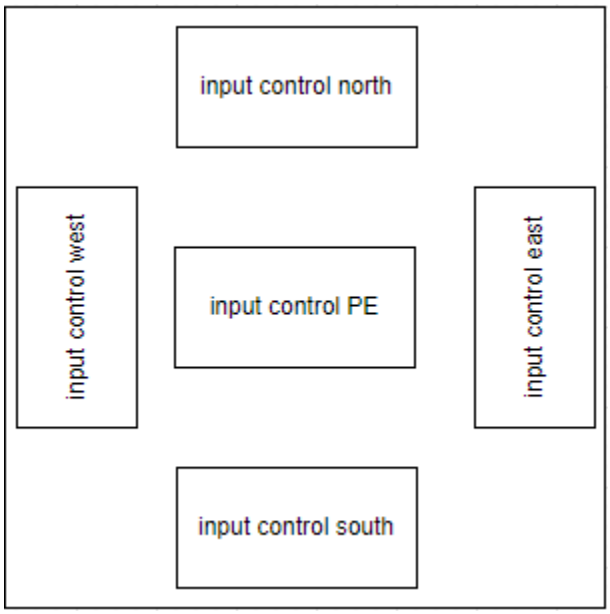
1. Maybe PE can forward ifmap data as soon as it receives it with some kind of guarding mechanism without overwriting the ifmap mem of another PE. Need to experiment with the testbench.

2. Write ifmap mem, filter mem, partial sum module, output mem, connect the NOC, and perform testbench.

Mesh

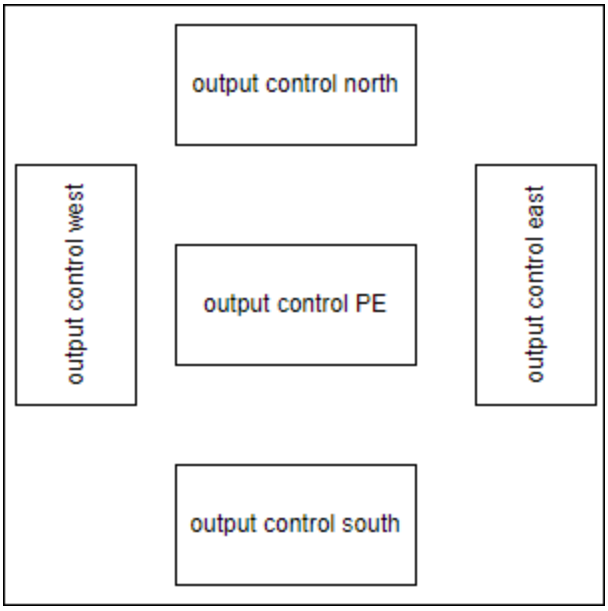
The 3 by 5 mesh consists of nine routers. Each router has five input control modules for north, south, west, east and PE to receive data, and five output control modules to arbit requests from four input control modules.

Input control receives data from its input channel depending on the x y direction and x y hop value. It issues requests to the appropriate output port. In the case of x hop equals 0, the packet has the correct x coordinate, and now it moves its way north or south. Same condition applies to Y hop. If both x hop and y hop equal 0, the packet has arrived at its destination and is sent to PE. Otherwise it proceeds traveling in the current direction which is specified by x dir and y dir.. X dir 0 is west, 1 east. Y dir 0 is south, 1 north. Routing scheme is X-Y, so input control always checks x hop value before y hop. When a packet is sent to output control unit, its hop value (either x or y) is decremented by 1 to indicate a successful hop.

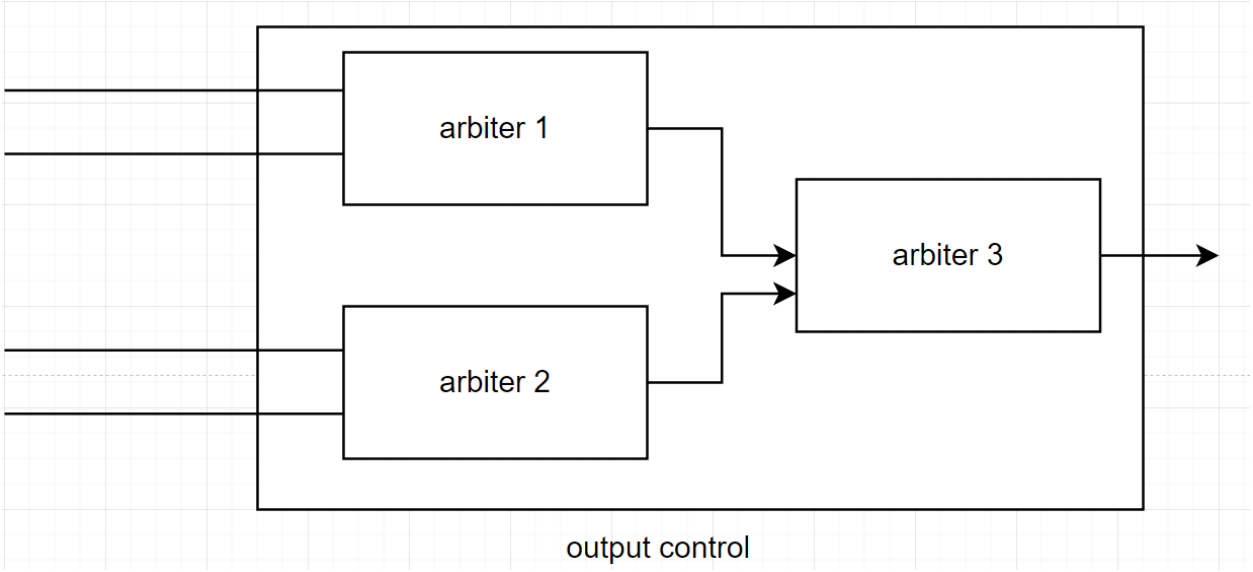


Router Showing Only Input Control

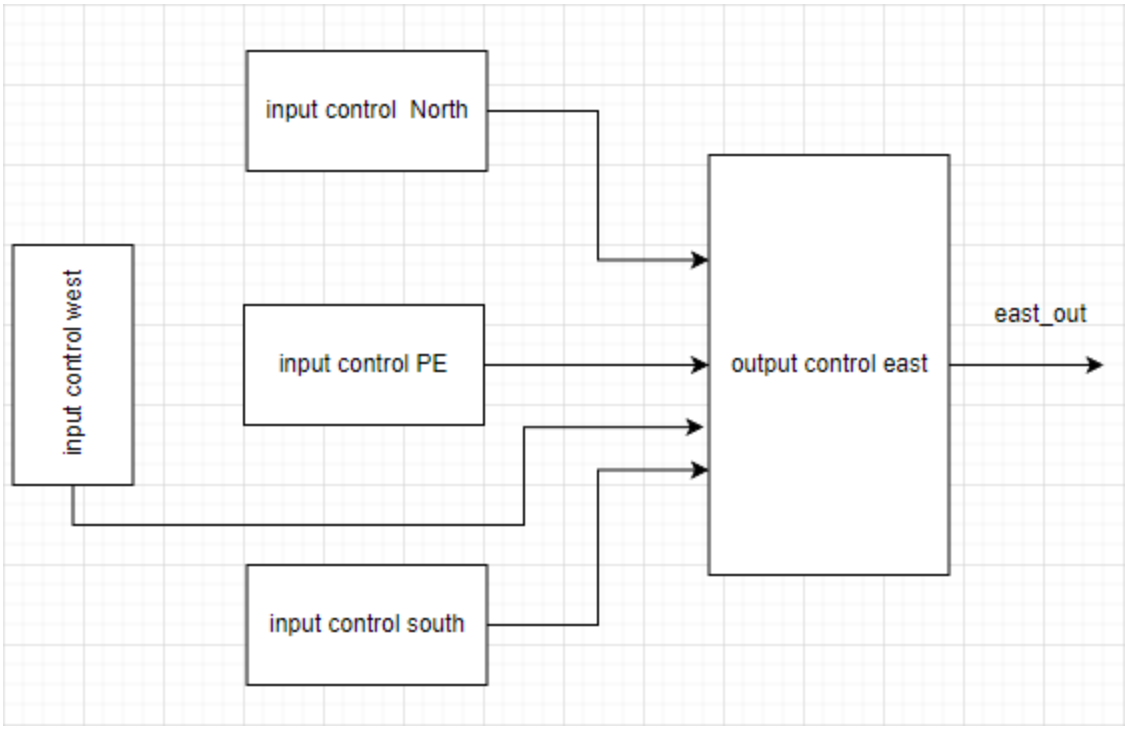
Output control consists of three 2 to 1 arbiters. It follows the first come first serve principle when only one input control module out of total four has requested its resource. In the case of contention, round robin is implemented to prevent deadlock. Details of the arbiter are included in the microarchitecture description.



Router Showing Only Output Control



Output Control Consists of Three Arbiters



Example: Four Input Control Connected to East Out Port

Duplicate this connection for all five output control modules and that’s the router.

Mesh Microarchitecture

```
module mesh (
    interface node1_PE_in, node1_PE_out,
    node2_PE_in, node2_PE_out,
    node3_PE_in, node3_PE_out,
    node4_PE_in, node4_PE_out,
    node5_PE_in, node5_PE_out,
    node6_PE_in, node6_PE_out,
    node7_PE_in, node7_PE_out,
    node8_PE_in, node8_PE_out,
    node9_PE_in, node9_PE_out,
    node10_PE_in, node10_PE_out,
    node11_PE_in, node11_PE_out,
    node12_PE_in, node12_PE_out,
    node13_PE_in, node13_PE_out,
    node14_PE_in, node14_PE_out,
    node15_PE_in, node15_PE_out
);
```

PE_in: channel for data writing into PE.

PE_out: channel for data injected into the NOC by PE.

```
module router(
    interface north_in, north_out,
    south_in, south_out,
    east_in, east_out,
    west_in, west_out,
    PE_in, PE_out
    );
```

xxxx_in: channel for packet arriving at the input control unit and needs to be sent to output control.

xxxx_out: the channel sending the eventual packet which either wins the arbitration or arrives first.

```
module input_ctrl (
    interface in, north_out, south_out, east_out, west_out, PE_out
);
```

in: one of five xxxx_ in channel of the router.

out: sends packet to one of four output control module depending on x y hop value and direction.

```
module output_ctrl (
    interface in1, in2, in3, in4, out
);
```

in1-4: input channels from four possible source contenders.

out: the eventual winner of arbitration.

```
parameter WIDTH_packet = 57;

module arbiter_2to1 (
    input logic in1_req,
    output logic  in1_ack,
    input logic [WIDTH_packet-1:0] in1_data,
    input logic in2_req,
    output logic in2_ack,
    input logic [WIDTH_packet-1:0] in2_data,
    interface out
);
```

Different from the project proposal, arbiter becomes our choice of gate level implementation due to its nature to peek req signals before performing the actual arbitration.

Works to be done:

All mesh related modules are implemented. Need to verify each module and create a wrapper module that connects mesh, input filter memory node, input feature map node and output memory node to the top level noc_snn specified by TA.

Github:

https://github.com/Chris-Du/EE552_NOC