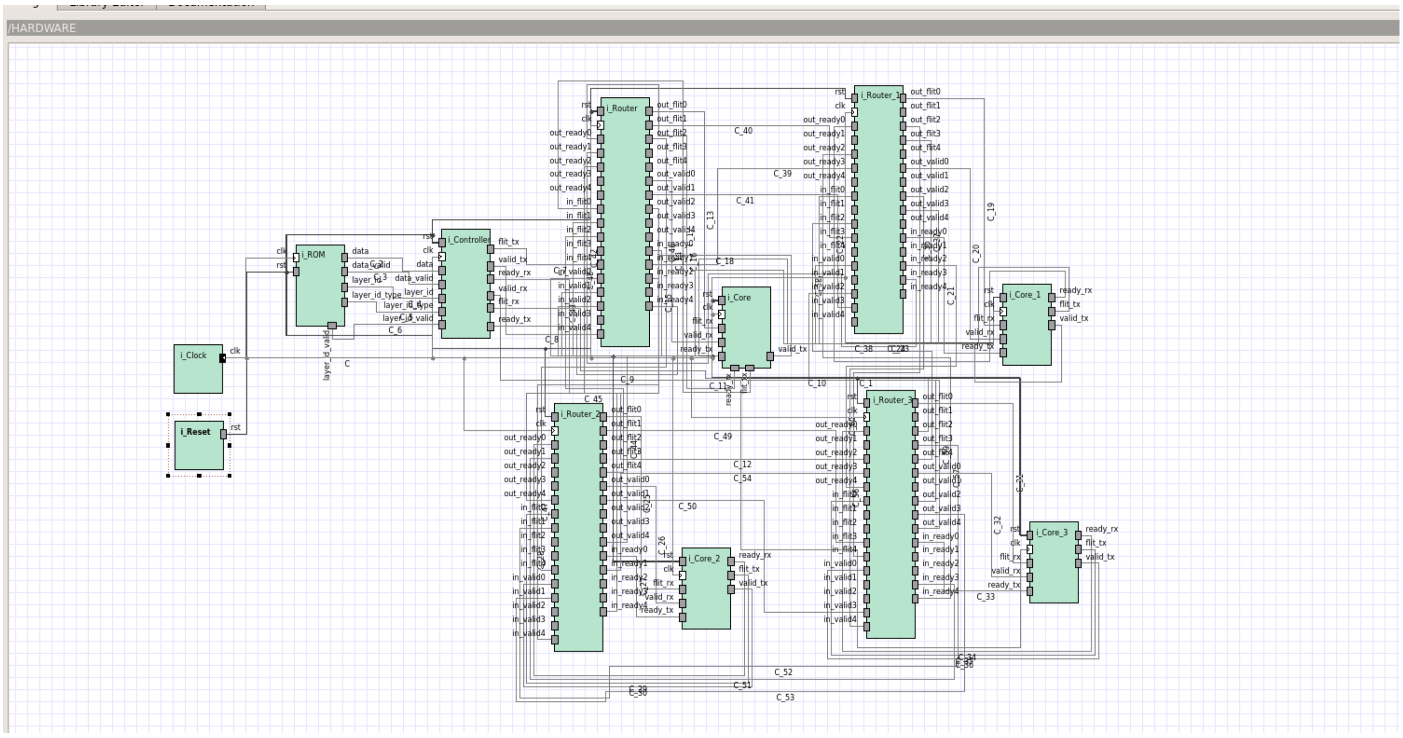
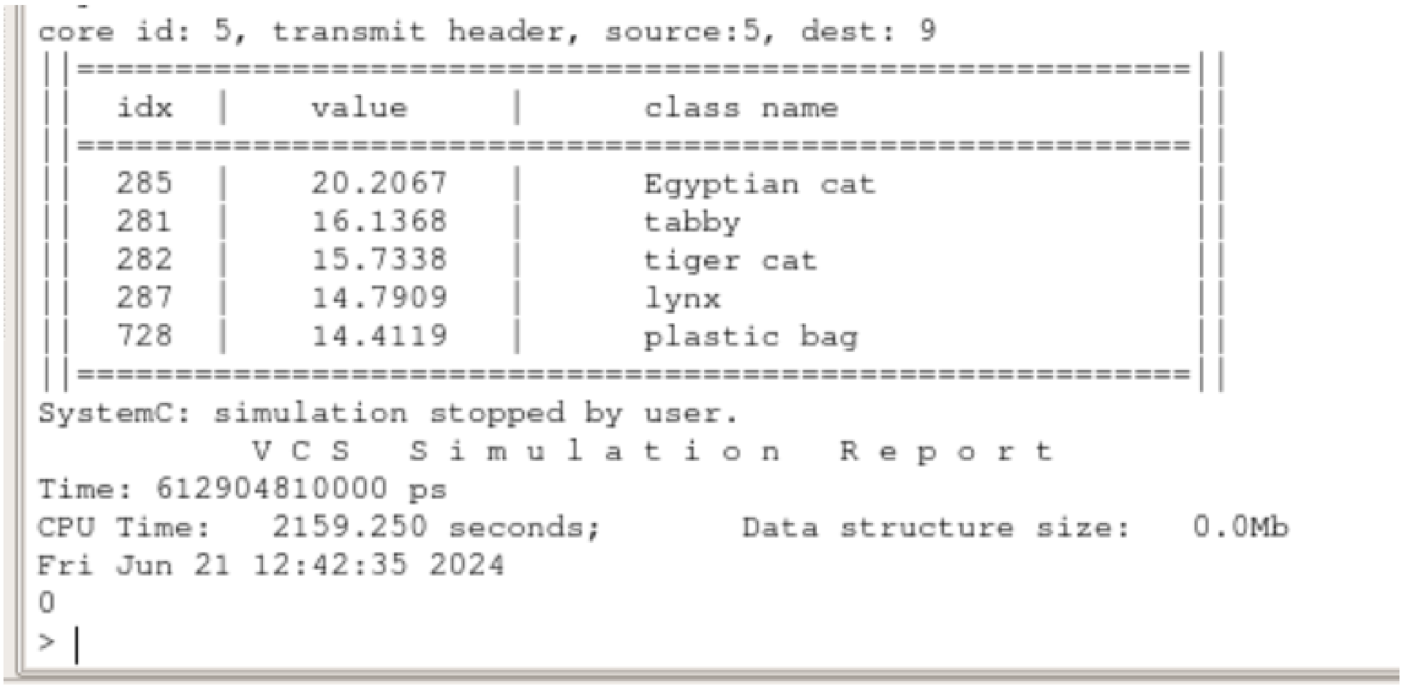
312510190 張家瑋

Final Project Report

Screenshots of the block diagram and simulation result of PA:

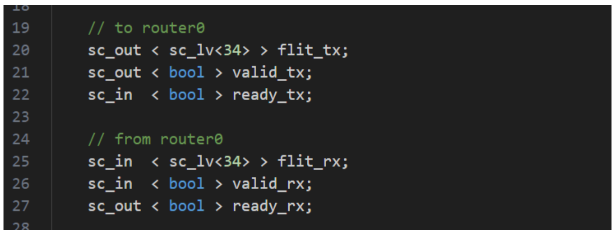




```
core id: 5, transmit header, source:5, dest: 9
||==============================================================||
||   idx   |     value      |       class name                  ||
||==============================================================||
||   285   |     20.2067    |       Egyptian cat                ||
||   281   |     16.1368    |       tabby                       ||
||   282   |     15.7338    |       tiger cat                   ||
||   287   |     14.7909    |       lynx                        ||
||   728   |     14.4119    |       plastic bag                 ||
||==============================================================||
SystemC: simulation stopped by user.
           V C S   S i m u l a t i o n   R e p o r t
Time: 612904810000 ps
CPU Time:   2159.250 seconds;       Data structure size:   0.0Mb
Fri Jun 21 12:42:35 2024
0
> |
```

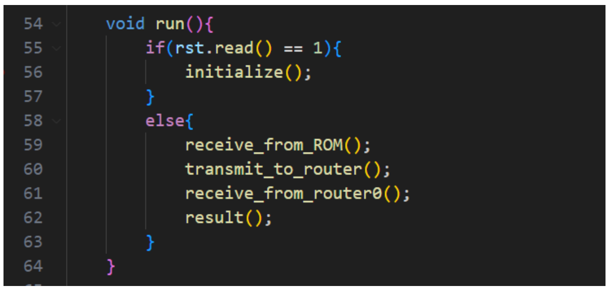
How do you control the NoC? What is each module responsible for? How do you design the router and NI? What routing algorithm do you use? What is the depth of the buffer? Do you use virtual channels?

I use the same communication protocol in both router and network interface, which is data_valid, data_ready signal in AXI-4 protocol.



```
    // to router0
    sc_out < sc_lv<34> > flit_tx;
    sc_out < bool > valid_tx;
    sc_in  < bool > ready_tx;

    // from router0
    sc_in  < sc_lv<34> > flit_rx;
    sc_in  < bool > valid_rx;
    sc_out < bool > ready_rx;
```

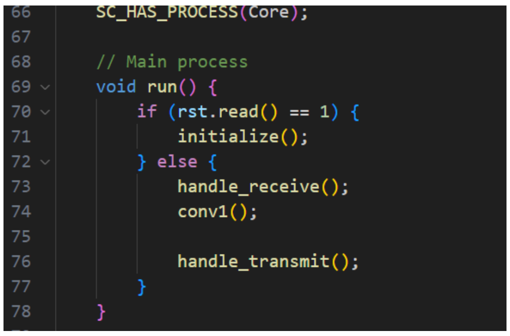
At the controller, all flits needed is received from ROM and stored in input buffer. The purpose of controller is to assign packets to router, decide the packet destination, and receive the output result from NOC design. I made one transmit packet included data of one output-channel of alexnet structure.



```
    void run(){
        if(rst.read() == 1){
            initialize();
        }
        else{
            receive_from_ROM();
            transmit_to_router();
            receive_from_router0();
            result();
        }
    }
```

At router, I use XY-routing algorithm in this project, each router has a input buffer size of 8, and no virtual channel used in this case. The purpose of router module is to schedule the path from source to destination, avoiding deadlock, and check whether the input buffer is empty or full status.

At core, 4 cores are used to build a NOC structure, one computes the five convolution layers of alexnet, while the other three cores responsible for three fully-connected layer, respectively. The purpose of core module is receive packets from router, divide alexnet neurons into computation operation efficiently, and send the correct data to NOC.



```
    SC_HAS_PROCESS(Core);

    // Main process
    void run() {
        if (rst.read() == 1) {
            initialize();
        } else {
            handle_receive();
            conv1();

            handle_transmit();
        }
    }
```

implementation approach:

At controller:

A. **Receive Data from ROM**:

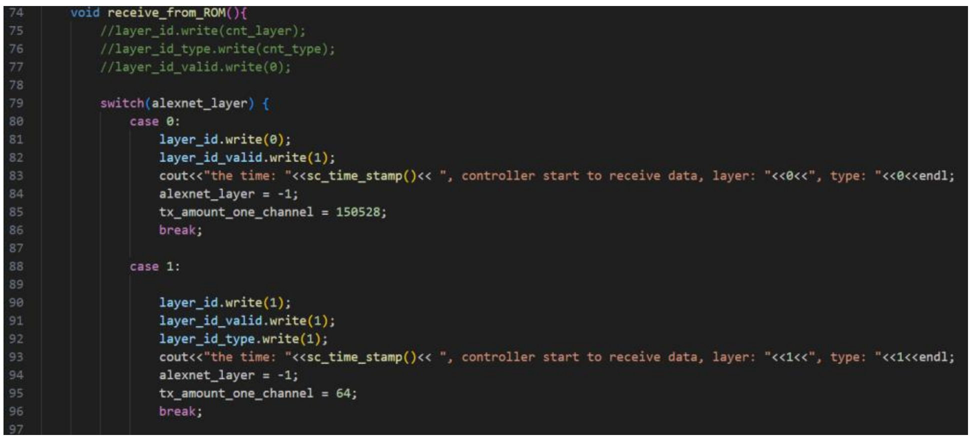
1. **Initialization**:

- The method uses a switch-case structure to handle different layers of the AlexNet architecture.

- For each layer, it sets layer_id, layer_id_type, and layer_id_valid signals, which are then used to fetch data from the ROM.

2.**Layer and Type Handling**:

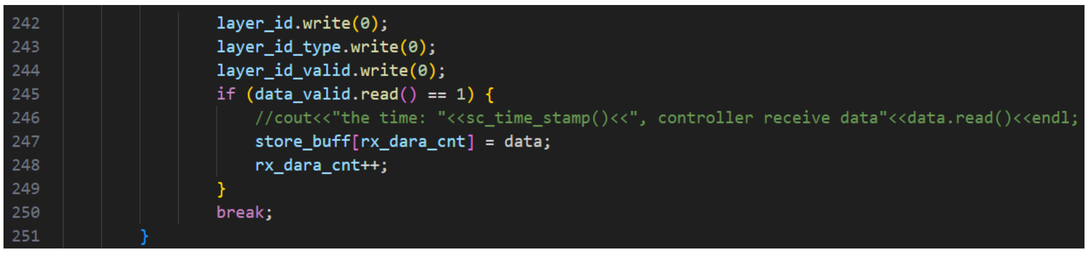
- Each case sets the appropriate layer_id, layer_id_valid, and sometimes layer_id_type signals based on the current alexnet_layer.
- After setting the necessary signals, it sets tx_total_amount to specify how much data to expect for transmission.
- input_size is set for certain layers, indicating how much data corresponds to the input size.



```cpp
void receive_from_ROM(){
    //layer_id.write(cnt_layer);
    //layer_id_type.write(cnt_type);
    //layer_id_valid.write(0);

    switch(alexnet_layer) {
        case 0:
            layer_id.write(0);
            layer_id_valid.write(1);
            cout<<"the time: "<<sc_time_stamp()<< ", controller start to receive data, layer: "<<0<<", type: "<<0<<endl;
            alexnet_layer = -1;
            tx_amount_one_channel = 150528;
            break;

        case 1:

            layer_id.write(1);
            layer_id_valid.write(1);
            layer_id_type.write(1);
            cout<<"the time: "<<sc_time_stamp()<< ", controller start to receive data, layer: "<<1<<", type: "<<1<<endl;
            alexnet_layer = -1;
            tx_amount_one_channel = 64;
            break;

```

3. **Data Reception**:

- In the default case, it reads the data_valid signal. If data is valid, it reads the data signal and stores it in the store_buff array.
- It increments the rx_dara_cnt counter each time data is read successfully.

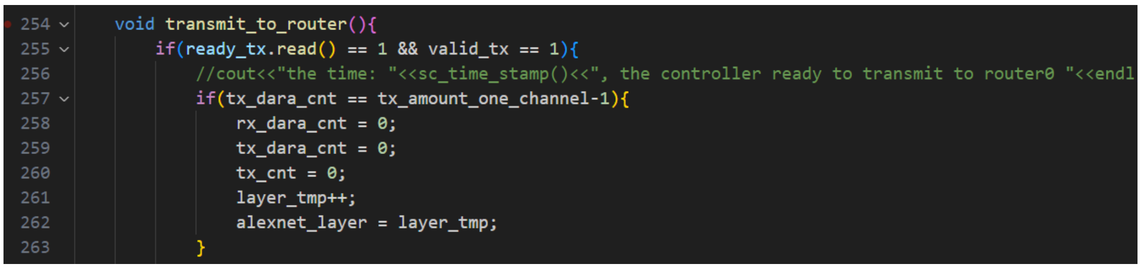


```cpp
                layer_id.write(0);
                layer_id_type.write(0);
                layer_id_valid.write(0);
                if (data_valid.read() == 1) {
                    //cout<<"the time: "<<sc_time_stamp()<<", controller receive data"<<data.read()<<endl;
                    store_buff[rx_dara_cnt] = data;
                    rx_dara_cnt++;
                }
                break;
        }
```

B. **Transmit Data to Router**:

1. **Transmission Control**:

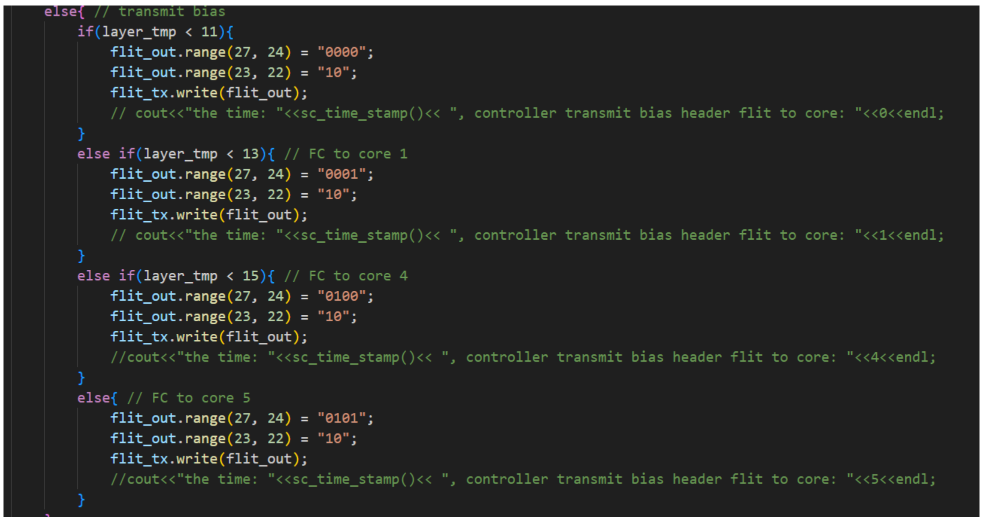
- If the ready_tx signal is high and valid_tx is true, it prepares to transmit data. It checks if tx_dara_cnt has reached tx_total_amount - 1, which indicates that all data has been transmitted. If so, it resets counters and updates the layer.
- If tx_cnt equals input_size for even layers (indicating weights), it resets tx_cnt and increments tx_dara_cnt. Otherwise, it increments tx_dara_cnt and tx_cnt appropriately.



```
void transmit_to_router(){
    if(ready_tx.read() == 1 && valid_tx == 1){
        //cout<<"the time: "<<sc_time_stamp()<<", the controller ready to transmit to router0 "<<endl
        if(tx_dara_cnt == tx_amount_one_channel-1){
            rx_dara_cnt = 0;
            tx_dara_cnt = 0;
            tx_cnt = 0;
            layer_tmp++;
            alexnet_layer = layer_tmp;
        }
```

2. **Flit Construction**:

- If rx_dara_cnt is not zero (indicating there is data to send), it constructs a flit (flow control digit) for transmission. The first flit is a header flit, constructed based on whether the data is an image, weight, or bias, and the target core.
- Subsequent flits are body flits containing actual data, converted to sc_lv<32> format using the convert_to_sc_lv method. If it's the end of a packet, it sets the flit type to "01" and prepares the next flit.

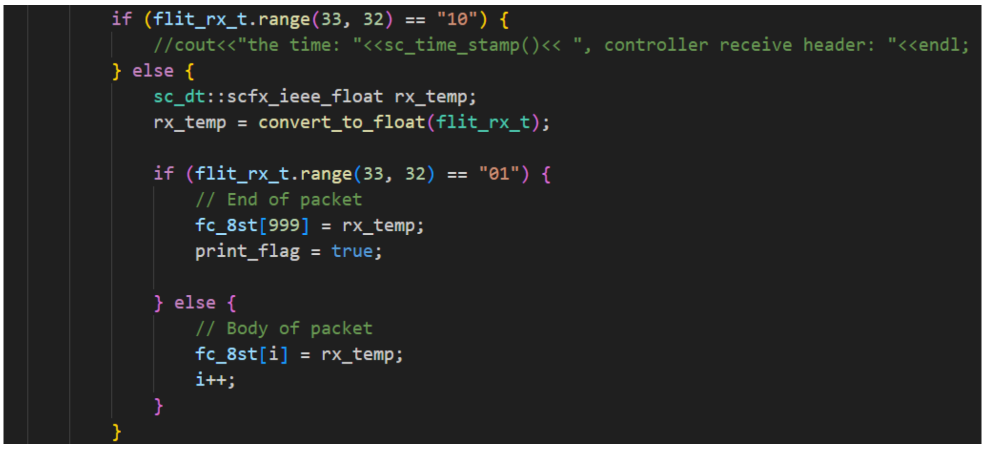


```
else{ // transmit bias
    if(layer_tmp < 11){
        flit_out.range(27, 24) = "0000";
        flit_out.range(23, 22) = "10";
        flit_tx.write(flit_out);
        // cout<<"the time: "<<sc_time_stamp()<< ", controller transmit bias header flit to core: "<<0<<endl;
    }
    else if(layer_tmp < 13){ // FC to core 1
        flit_out.range(27, 24) = "0001";
        flit_out.range(23, 22) = "10";
        flit_tx.write(flit_out);
        // cout<<"the time: "<<sc_time_stamp()<< ", controller transmit bias header flit to core: "<<1<<endl;
    }
    else if(layer_tmp < 15){ // FC to core 4
        flit_out.range(27, 24) = "0100";
        flit_out.range(23, 22) = "10";
        flit_tx.write(flit_out);
        //cout<<"the time: "<<sc_time_stamp()<< ", controller transmit bias header flit to core: "<<4<<endl;
    }
    else{ // FC to core 5
        flit_out.range(27, 24) = "0101";
        flit_out.range(23, 22) = "10";
        flit_tx.write(flit_out);
        //cout<<"the time: "<<sc_time_stamp()<< ", controller transmit bias header flit to core: "<<5<<endl;
    }
```

C. **Receive Data from Router**:

1.**Data Reception**:

- If valid_rx is true, it reads the incoming flit (flit_rx). It checks if the flit is a header flit or a body flit by examining flit_rx.range(33, 32).
- For body flits, it converts the flit to a float using the convert_to_float method and stores it in the fc_8st array. If it's the end of a packet (flit type "01"), it sets the print_flag to true to indicate that the data reception is complete.

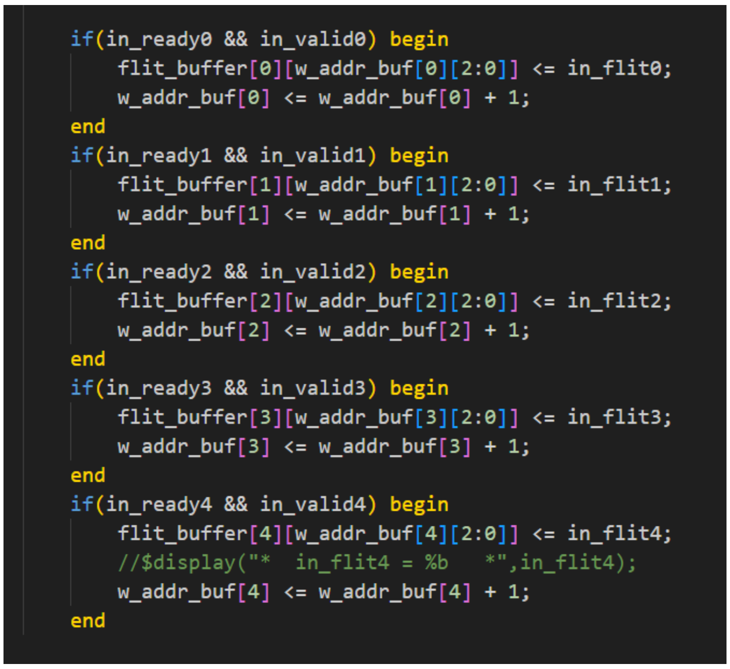


```
if (flit_rx_t.range(33, 32) == "10") {
    //cout<<"the time: "<<sc_time_stamp()<< ", controller receive header: "<<endl;
} else {
    sc_dt::scfx_ieee_float rx_temp;
    rx_temp = convert_to_float(flit_rx_t);

    if (flit_rx_t.range(33, 32) == "01") {
        // End of packet
        fc_8st[999] = rx_temp;
        print_flag = true;

    } else {
        // Body of packet
        fc_8st[i] = rx_temp;
        i++;
    }
}
```

At router:

A. handle_rx():

- Handles receiving flits from input ports (in_valid[i]) and writes them into the appropriate input buffer (flit_buffer[i]).
- Check if Input channel is ready, ensures that the input channel (in_ready[i]) is ready to accept the incoming flit.
- Write address calculation, determines the write address (addr_rx) within the input buffer (flit_buffer[i][]) for port i.
- Store incoming flit, reads the incoming flit from port i (in_flit[i].read()) and stores it into the buffer at the calculated address (flit_buffer[i][addr_rx]).

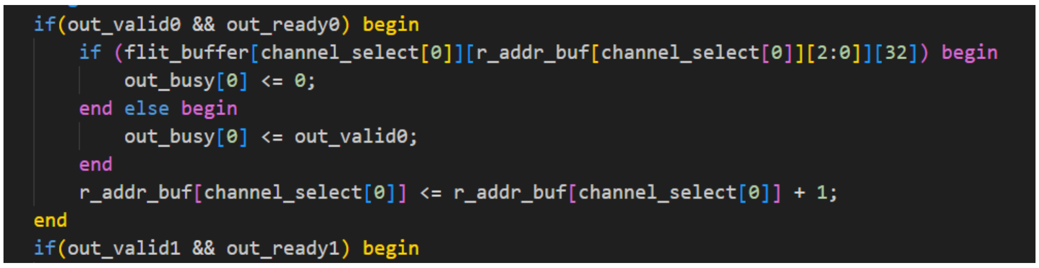


```
if(in_ready0 && in_valid0) begin
    flit_buffer[0][w_addr_buf[0][2:0]] <= in_flit0;
    w_addr_buf[0] <= w_addr_buf[0] + 1;
end
if(in_ready1 && in_valid1) begin
    flit_buffer[1][w_addr_buf[1][2:0]] <= in_flit1;
    w_addr_buf[1] <= w_addr_buf[1] + 1;
end
if(in_ready2 && in_valid2) begin
    flit_buffer[2][w_addr_buf[2][2:0]] <= in_flit2;
    w_addr_buf[2] <= w_addr_buf[2] + 1;
end
if(in_ready3 && in_valid3) begin
    flit_buffer[3][w_addr_buf[3][2:0]] <= in_flit3;
    w_addr_buf[3] <= w_addr_buf[3] + 1;
end
if(in_ready4 && in_valid4) begin
    flit_buffer[4][w_addr_buf[4][2:0]] <= in_flit4;
    //$display("*  in_flit4 = %b   *",in_flit4);
    w_addr_buf[4] <= w_addr_buf[4] + 1;
end
```

B. handle_tx():

- Determine output port, retrieves the selected output port (port) based on the routing decision stored in channel_select[i].
- Check if last flit , determines if the flit being transmitted is the last flit of the packet (flit_buffer[port][addr_tx][32] == 1). Updates out_busy[i] accordingly.

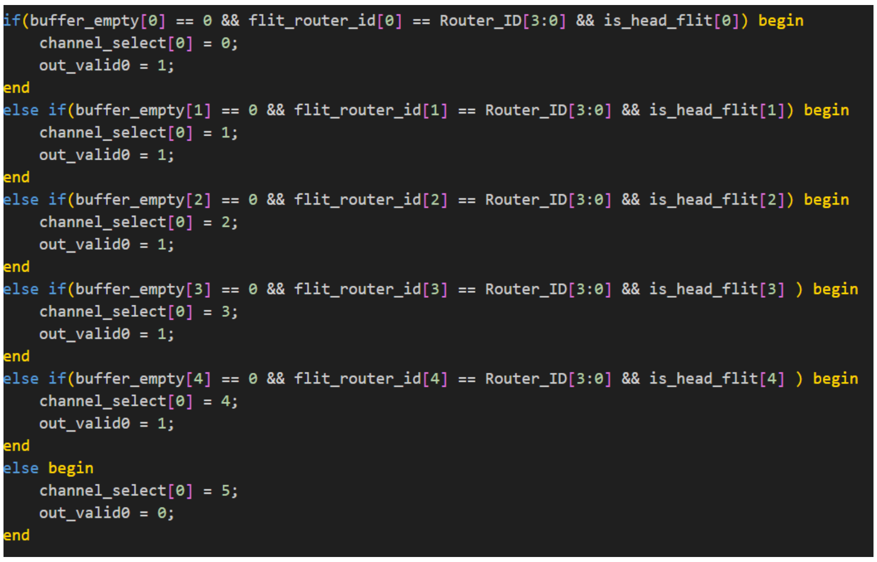
- Transmit flit, writes the flit from the buffer (flit_buffer[port][addr_tx]) to the output channel.
- Update read address, increments the read address (r_addr_buf[channel_select[i]]) for the selected output channel, preparing it for the next flit transmission.



```
if(out_valid0 && out_ready0) begin
    if (flit_buffer[channel_select[0]][r_addr_buf[channel_select[0]][2:0]][32]) begin
        out_busy[0] <= 0;
    end else begin
        out_busy[0] <= out_valid0;
    end
    r_addr_buf[channel_select[0]] <= r_addr_buf[channel_select[0]] + 1;
end
if(out_valid1 && out_ready1) begin
```

C. xy_routing():

- Local Output (0): Directs flits locally if the head flit matches the current router ID.
- East (1): Routes flits to the East if the destination router ID modulo 4 is greater than the current router ID modulo 4.
- South (2): Routes flits to the South if the destination router ID divided by 4 is greater than the current router ID divided by 4, and ensures that East and West channels are not chosen.
- West (3): Routes flits to the West if the destination router ID modulo 4 is less than the current router ID modulo 4, and ensures that East and West channels are not chosen.
- North (4): Routes flits to the North if the destination router ID divided by 4 is less than the current router ID divided by 4, and ensures that East and West channels are not chosen.

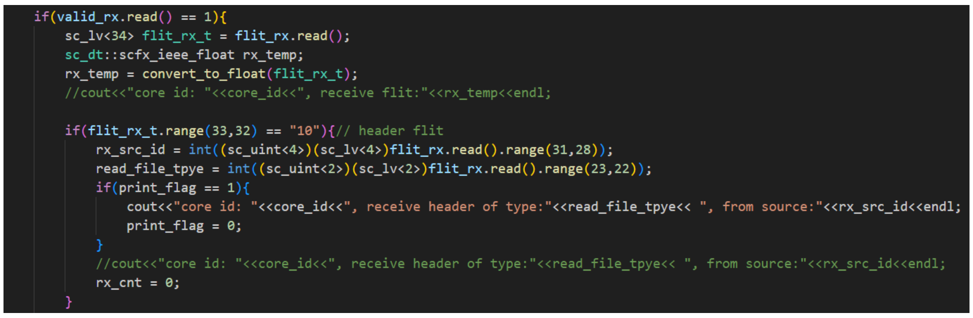


```
if(buffer_empty[0] == 0 && flit_router_id[0] == Router_ID[3:0] && is_head_flit[0]) begin
    channel_select[0] = 0;
    out_valid0 = 1;
end
else if(buffer_empty[1] == 0 && flit_router_id[1] == Router_ID[3:0] && is_head_flit[1]) begin
    channel_select[0] = 1;
    out_valid0 = 1;
end
else if(buffer_empty[2] == 0 && flit_router_id[2] == Router_ID[3:0] && is_head_flit[2]) begin
    channel_select[0] = 2;
    out_valid0 = 1;
end
else if(buffer_empty[3] == 0 && flit_router_id[3] == Router_ID[3:0] && is_head_flit[3] ) begin
    channel_select[0] = 3;
    out_valid0 = 1;
end
else if(buffer_empty[4] == 0 && flit_router_id[4] == Router_ID[3:0] && is_head_flit[4] ) begin
    channel_select[0] = 4;
    out_valid0 = 1;
end
else begin
    channel_select[0] = 5;
    out_valid0 = 0;
end
```

At core:

A. **Receive Data (handle_receive())**:

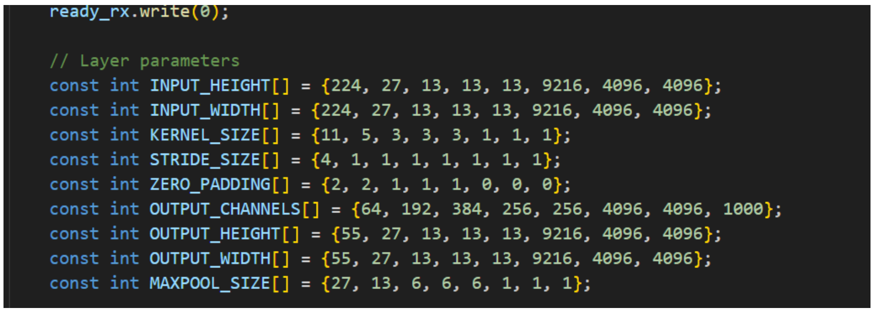
- Manages the reception of data (flit_rx) and its interpretation based on the flit format. It determines the type of data (image, weight, bias) based on the received header (flit with specific header bits).
- Stores received data into appropriate buffers (alexnet_buff1, alexnet_buff2, fc_7st). and also sets flags (read_image_finish, read_weight_finish, read_bias_finish) when data reception completes.



```
if(valid_rx.read() == 1){
    sc_lv<34> flit_rx_t = flit_rx.read();
    sc_dt::scfx_ieee_float rx_temp;
    rx_temp = convert_to_float(flit_rx_t);
    //cout<<"core id: "<<core_id<<", receive flit:"<<rx_temp<<endl;

    if(flit_rx_t.range(33,32) == "10"){// header flit
        rx_src_id = int((sc_uint<4>)(sc_lv<4>)flit_rx.read().range(31,28));
        read_file_tpye = int((sc_uint<2>)(sc_lv<2>)flit_rx.read().range(23,22));
        if(print_flag == 1){
            cout<<"core id: "<<core_id<<", receive header of type:"<<read_file_tpye<< ", from source:"<<rx_src_id<<endl;
            print_flag = 0;
        }
        //cout<<"core id: "<<core_id<<", receive header of type:"<<read_file_tpye<< ", from source:"<<rx_src_id<<endl;
        rx_cnt = 0;
    }
```

B. **Compute (conv1())**:

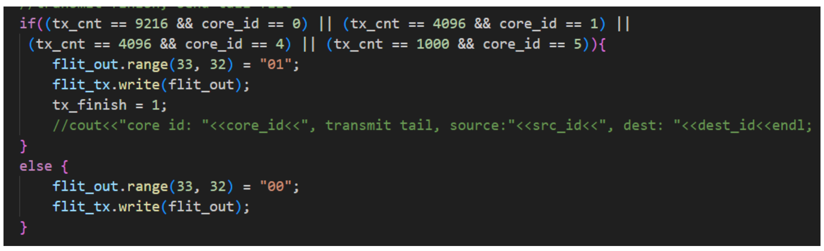
- Implements convolution and pooling operations for each layer of AlexNet.
- Uses pre-defined parameters for each layer (INPUT_HEIGHT, INPUT_WIDTH, etc.) to calculate output feature maps.
- Iterates through channels, rows, and columns to perform convolution operations (alexnet_layer specific). Once convolution and pooling are completed for a layer and output channels, sets flags (read_bias_finish, compute_finish).



```
ready_rx.write(0);

// Layer parameters
const int INPUT_HEIGHT[] = {224, 27, 13, 13, 13, 9216, 4096, 4096};
const int INPUT_WIDTH[] = {224, 27, 13, 13, 13, 9216, 4096, 4096};
const int KERNEL_SIZE[] = {11, 5, 3, 3, 3, 1, 1, 1};
const int STRIDE_SIZE[] = {4, 1, 1, 1, 1, 1, 1, 1};
const int ZERO_PADDING[] = {2, 2, 1, 1, 1, 0, 0, 0};
const int OUTPUT_CHANNELS[] = {64, 192, 384, 256, 256, 4096, 4096, 1000};
const int OUTPUT_HEIGHT[] = {55, 27, 13, 13, 13, 9216, 4096, 4096};
const int OUTPUT_WIDTH[] = {55, 27, 13, 13, 13, 9216, 4096, 4096};
const int MAXPOOL_SIZE[] = {27, 13, 6, 6, 6, 1, 1, 1};
```

C. **Transmit Data (handle_transmit())**:

- Prepares data (flit_tx) to transmit to the next core/module or output interface.
- Controls transmission based on readiness (ready_tx), valid data (valid_tx), and acknowledgment (ready_tx).



```
if((tx_cnt == 9216 && core_id == 0) || (tx_cnt == 4096 && core_id == 1) ||
 (tx_cnt == 4096 && core_id == 4) || (tx_cnt == 1000 && core_id == 5)){
    flit_out.range(33, 32) = "01";
    flit_tx.write(flit_out);
    tx_finish = 1;
    //cout<<"core id: "<<core_id<<", transmit tail, source:"<<src_id<<", dest: "<<dest_id<<endl;
}
else {
    flit_out.range(33, 32) = "00";
    flit_tx.write(flit_out);
}
```

challenges faced:

Implementing a router in Verilog can present several challenges, especially when dealing with complex logic and timing constraints typical in networking hardware. Here are some challenges I faced in the implement process:

1.Timing and Clock Domain Management

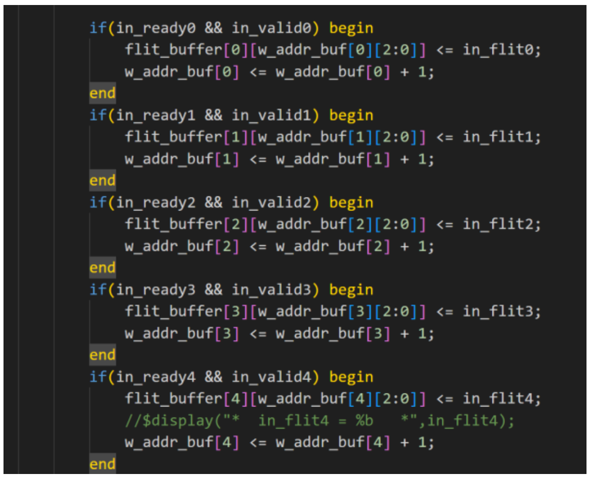
- **Clock Synchronization:** Ensuring all operations are synchronized to the same clock edge is crucial. I found that violating timing constraints can lead to unpredictable behavior or timing errors. Make sure the router algorithm works in Verilog as same as SystemC HW is a great challenge in this project.

2. State Management

- **Reset Synchronization:** Properly initializing and resetting internal states (rst signal handling) is critical for correct operation during power-up or system resets.
- **State Machine Complexity:** Implementing complex state machines to handle various routing decisions and buffer management efficiently can be challenging.

3. Buffer Management

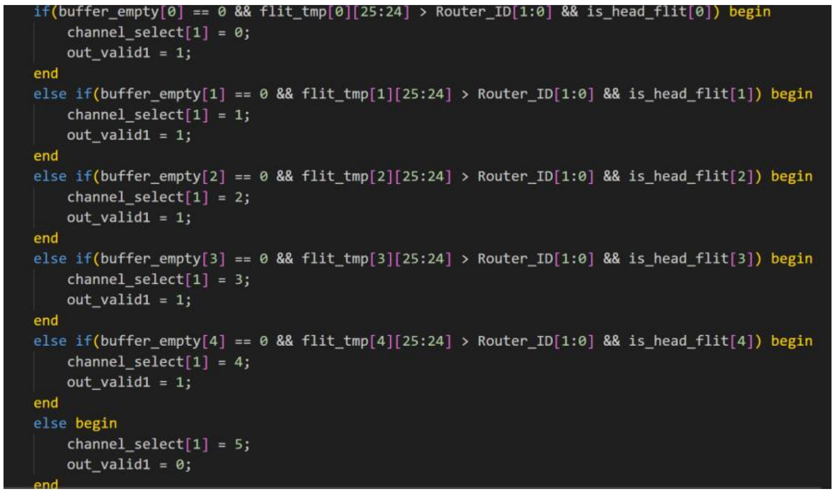
- **Buffer Read/Write Control:** Ensuring that reads (r_addr_buf) and writes (w_addr_buf) to the buffer arrays are properly synchronized and that data integrity is maintained under all operating conditions.
- **Buffer Overflow/Underflow:** Preventing buffer overflow (when the buffer is full) and underflow (when trying to read from an empty buffer) requires careful state management and control logic.



```
if(in_ready0 && in_valid0) begin
    flit_buffer[0][w_addr_buf[0][2:0]] <= in_flit0;
    w_addr_buf[0] <= w_addr_buf[0] + 1;
end
if(in_ready1 && in_valid1) begin
    flit_buffer[1][w_addr_buf[1][2:0]] <= in_flit1;
    w_addr_buf[1] <= w_addr_buf[1] + 1;
end
if(in_ready2 && in_valid2) begin
    flit_buffer[2][w_addr_buf[2][2:0]] <= in_flit2;
    w_addr_buf[2] <= w_addr_buf[2] + 1;
end
if(in_ready3 && in_valid3) begin
    flit_buffer[3][w_addr_buf[3][2:0]] <= in_flit3;
    w_addr_buf[3] <= w_addr_buf[3] + 1;
end
if(in_ready4 && in_valid4) begin
    flit_buffer[4][w_addr_buf[4][2:0]] <= in_flit4;
    //$display("*  in_flit4 = %b   *",in_flit4);
    w_addr_buf[4] <= w_addr_buf[4] + 1;
end
```

4.Routing Logic and Arbitration

- **Channel Arbitration:** Implementing arbitration logic to select among multiple input/output channels based on various criteria (e.g., destination address, priority, availability) can be complex and critical to ensuring efficient data flow through the router.
- **Deadlock Prevention:** Designing the router to prevent deadlocks (e.g., circular dependencies in channel allocation) by carefully managing resource allocation and arbitration.



```
if(buffer_empty[0] == 0 && flit_tmp[0][25:24] > Router_ID[1:0] && is_head_flit[0]) begin
    channel_select[1] = 0;
    out_valid1 = 1;
end
else if(buffer_empty[1] == 0 && flit_tmp[1][25:24] > Router_ID[1:0] && is_head_flit[1]) begin
    channel_select[1] = 1;
    out_valid1 = 1;
end
else if(buffer_empty[2] == 0 && flit_tmp[2][25:24] > Router_ID[1:0] && is_head_flit[2]) begin
    channel_select[1] = 2;
    out_valid1 = 1;
end
else if(buffer_empty[3] == 0 && flit_tmp[3][25:24] > Router_ID[1:0] && is_head_flit[3]) begin
    channel_select[1] = 3;
    out_valid1 = 1;
end
else if(buffer_empty[4] == 0 && flit_tmp[4][25:24] > Router_ID[1:0] && is_head_flit[4]) begin
    channel_select[1] = 4;
    out_valid1 = 1;
end
else begin
    channel_select[1] = 5;
    out_valid1 = 0;
end
```

5. Verification and Testing

- **Functional Verification:** Thoroughly verifying the router design under various traffic patterns, including edge cases and stress conditions, to ensure correct operation and adherence to protocol specifications.

other observations or insights gained:

Routers determine the path or route that packets take from a source to a destination. This involves making decisions based on the packet's destination address and network topology. Once the route is determined, routers switch packets between different channels or links within the network, ensuring they reach their intended destination efficiently.

Routers manage the flow of packets to prevent congestion and ensure that data is transmitted at a rate that the receiving components can handle. This is typically achieved through mechanisms like credit-based flow control or buffer-based flow control. Routers monitor network traffic and use various algorithms to manage congestion, such as prioritization of packets or dynamic routing adjustments to avoid overloaded channels. So decide routing algorithm and way of flow control in implementing router HDL is a great work need to bbe considered.