Application Acceleration with High-Level Synthesis

Final Project Report

Group03 鍾宇騫 許鏡瑋

1. Project Statements

Since our research is related to multi-DNN accelerators, which include an interconnection network on chip (NoC) for the data transmission between each neural network processor, we want to evaluate Vitis and Xilinx data center acceleration card is suitable for these kinds of design or not.

For the software part, we think Vitis provides a complete software stack for hardware prototyping, host-card communication, and result evaluation, which is the main reason that we want to try this flow.

For the essential hardware part, Xilinx provides High-Level Synthesis (HLS), a fast architecture-exploring method to design hardware. Due to its features, we want to explore whether Xilinx HLS is suitable for structural design or not.

2. System Overall Architecture

To achieve our goal, we plan to design a two-by-two 2D mesh interconnection network (figure 1), including four 5-port routers. A router has three directions (scalable to the five-directions router if needed), two are for other routers' connections, and one is for the network interface (NI), where NI is an interface for processors to send and receive packets from the router. The **processor** in figure 1 connected to NI can be any design that fits the NI specification, e.g., RISC-V processor, domain-specific accelerator, and I/O devices. The HBM in figure 1 is High-Bandwidth Memory, which plays the main memory role in this system, used for data and instruction storage. But in this project, we only implement the core part, which contains four routers and NIs.

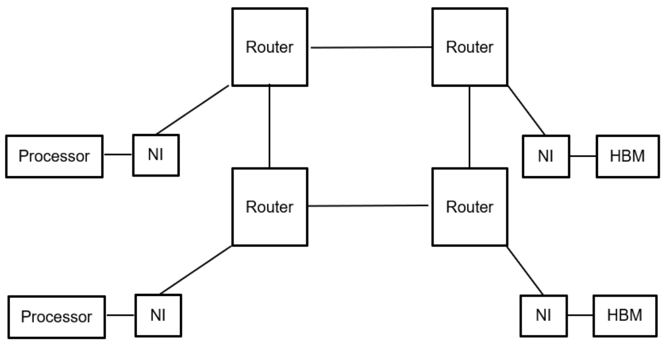


Figure 1: 2D Mesh NoC Block Diagram

3. Project Scope

Firstly, we introduce the data structure of the message in figure 2; for every testcase, we generate thousands of 520-bit packets, each including sequence number (Seq#), destination router ID (Dest), and Payload. In the real system, a 520-bit wide data port is not practical; thus, we split a packet into five flits, as shown in the lower part of figure 2; each flit takes 104 bits and is marked into three types, Head, Body, and Tail.

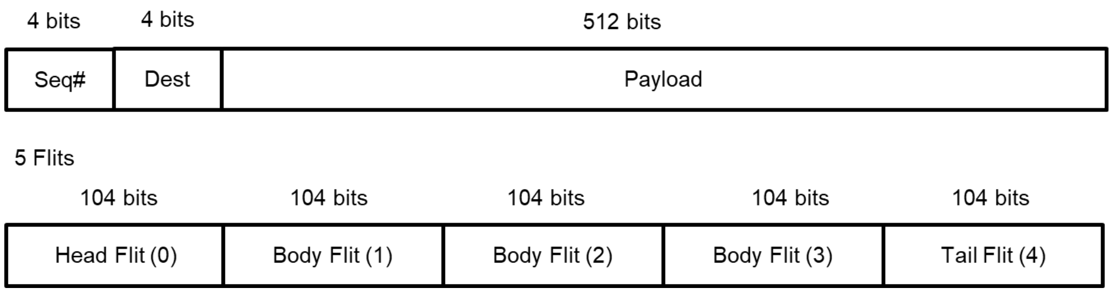


Figure 2: Packet and Flit structure

For the scalability of this system, all routers have the same architecture, as shown in figure 2. In the design ***router_top.h***, we specify each router contains five input ports, with three data I/O ports:

- ➢ Input Flit: data port of flit
- ➢ Input Valid: whether the data of the data port is valid
- ➢ VC Full: Indicate input unit is full or not

Every input port is connected to an input unit, which forwards input flit to the target buffer to wait for the scheduling of Arbiter to an output unit.

Arbiter considers the current state of input units and output units to schedule the Switch to transmit data from input units to corresponding output units.

The output unit is a buffer that takes the flit from the input unit and sends them out in FIFO order. Also, it contains three output ports:

- Output Flit: data port of flit.
- Output Valid: whether the data of the data port is valid.
- VC Full: Indicate input unit of the downstream router is full or not.

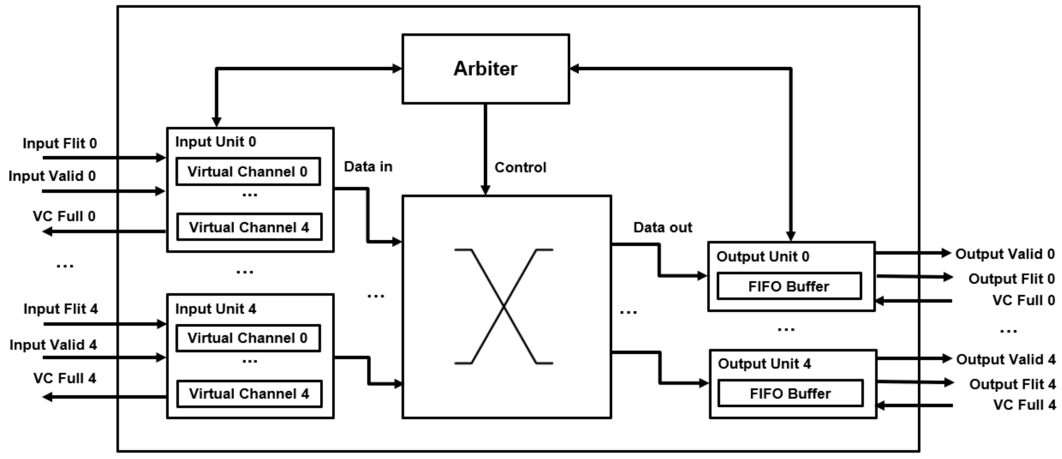


Figure 3: Router architecture

Secondly, we introduce the details of all components.

(a) FIFO Buffer

Since the behavior of hls::stream provided by Xilinx Vitis HLS can not satisfy our requirements (Need one cycle delay to get front data), we decided to implement our one. As figure 4 shows, We use 64 104-bit registers to build the memory part and maintain the read, write, and prefetch mechanism in design ***flit_buffer.h***.

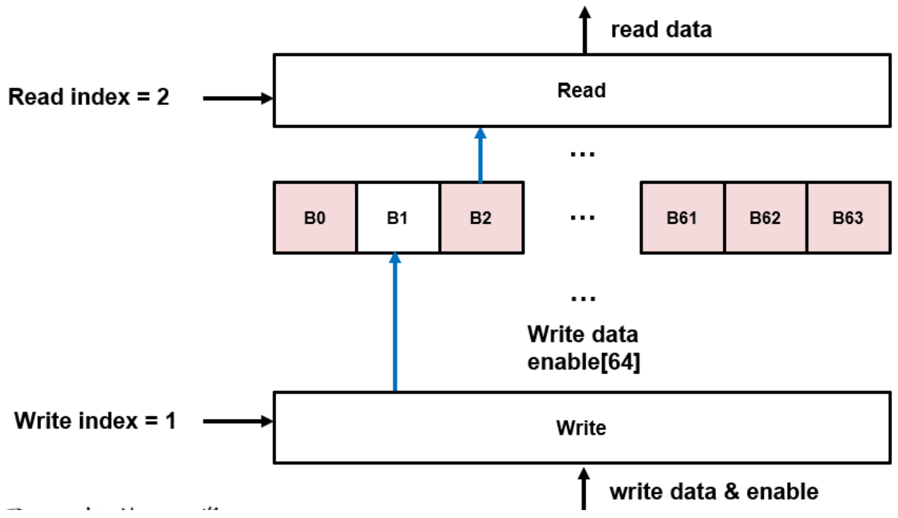


Figure 4: Input FIFO buffer

(b) Input Unit

An input unit contains three main components, first is the Routing Unit, which forward flits to the target virtual channel based on a built-in routing algorithm. The routing algorithm first route flits to the same Yaxis, then X-axis, which is the simplest way of routing, not considering any congestion. Virtual Channels are FIFO Buffer we mentioned before, which hold the flit data and send them via Switch to the corresponding output unit. For the Arbiter to allocate which Virtual Channel can send data, the Input Unit output a 5-bit request and receive 5-bit grant signals from the Arbiter.

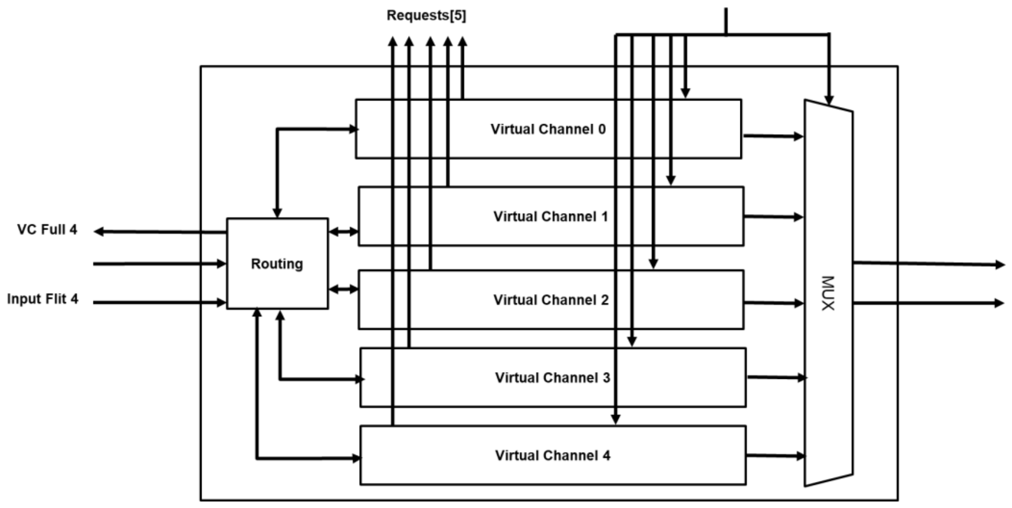


Figure 5: Input Unit Architecture

(c) Output Unit

This part is more straightforward than the Input Unit. It just stores the flit from Switch based on Arbiter and sends the flit in FIFO order if the downstream buffer is not full.

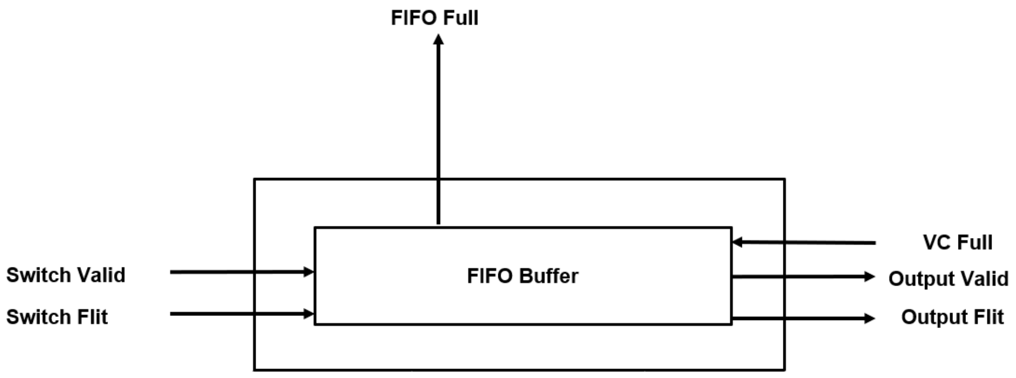


Figure 6: Output Unit Architecture

(d) Arbiter

Arbiter will arbitrate the priority for each Input Unit in Round-Robin (RR) and allocate the Switch connection based on the current request and states of Output Units. And for the arbitration, follows the rule:

- If send 0~4 flits and the request is valid, **keep** the priority

- If send 5 flits, **pass** the priority
- If send 0 flits and no request, **pass** the priority
- If send 1~4 flits but no request, **keep** the priority

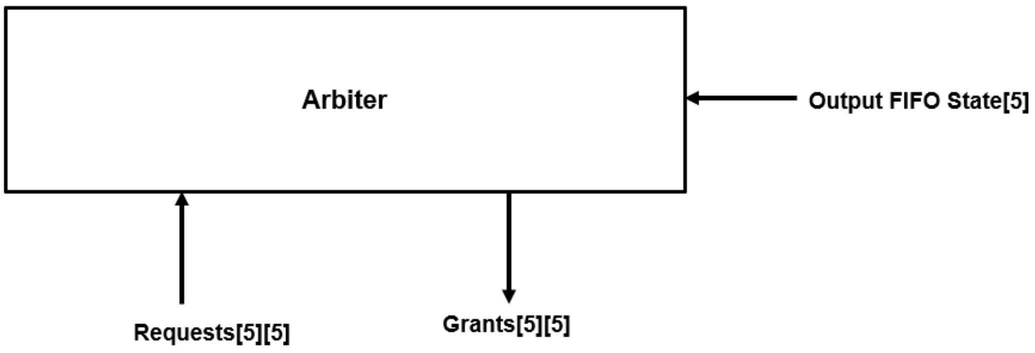


Figure 7: Arbiter Architecture

(e) Switch

Figure 8 shows a mux-based switch, which selects the input based on grants[5][5] from Arbiter.

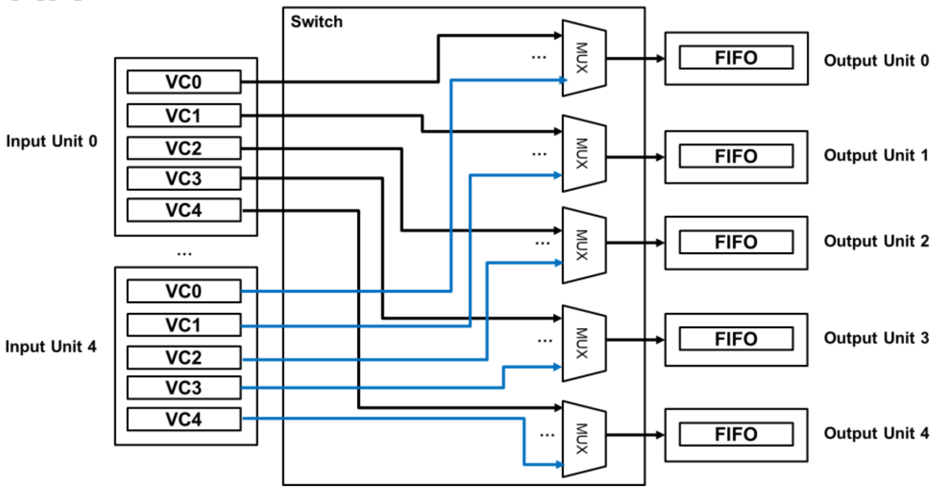


Figure 8: Mux-based Switch

4. Structural Design Technique

For the high-level synthesis, we must want to describe some hardware details; thus, we use the structural design method to implement our project for this kind of design. Firstly we recall the lecture in class. Teacher Lai provides two types of techniques. One is for stage design that all components can be ordered in a direct acyclic graph (DAG), shown in Lab E RISC-V HLS. The other one is for the design that has a cyclic relationship, as shown in figure 9.

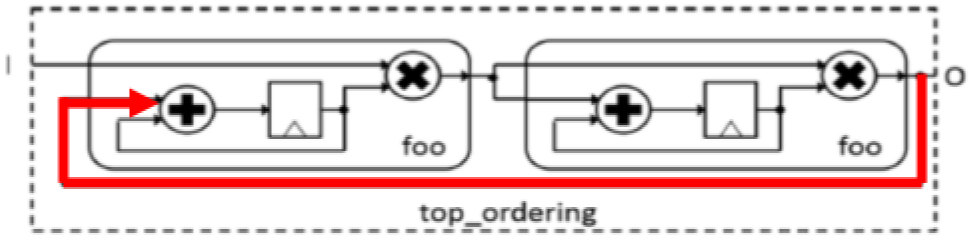
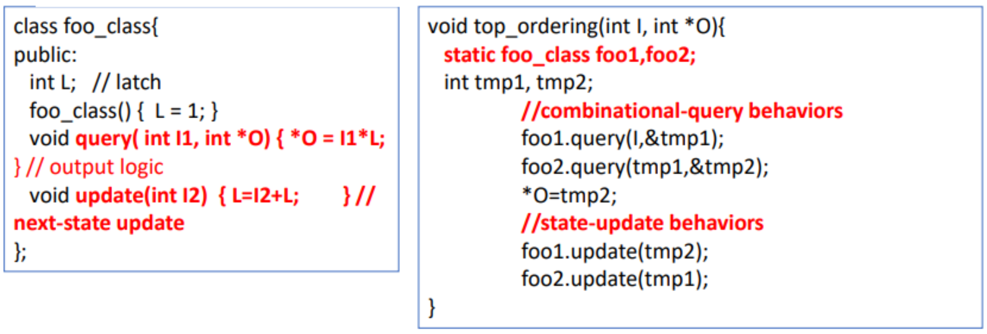


Figure 9: Cyclic relationship structure

If the design has a cyclic relationship, we can not describe them in the order from back to front. We should split the part into combinational and sequential parts like Register-Transfer Level (RTL) in Verilog or VHDL like figure 10.



```
class foo_class{
public:
  int L;   // latch
  foo_class() {  L = 1; }
  void query( int I1, int *O) { *O = I1*L;
} // output logic
  void update(int I2)  { L=I2+L;      } //
next-state update
};
```

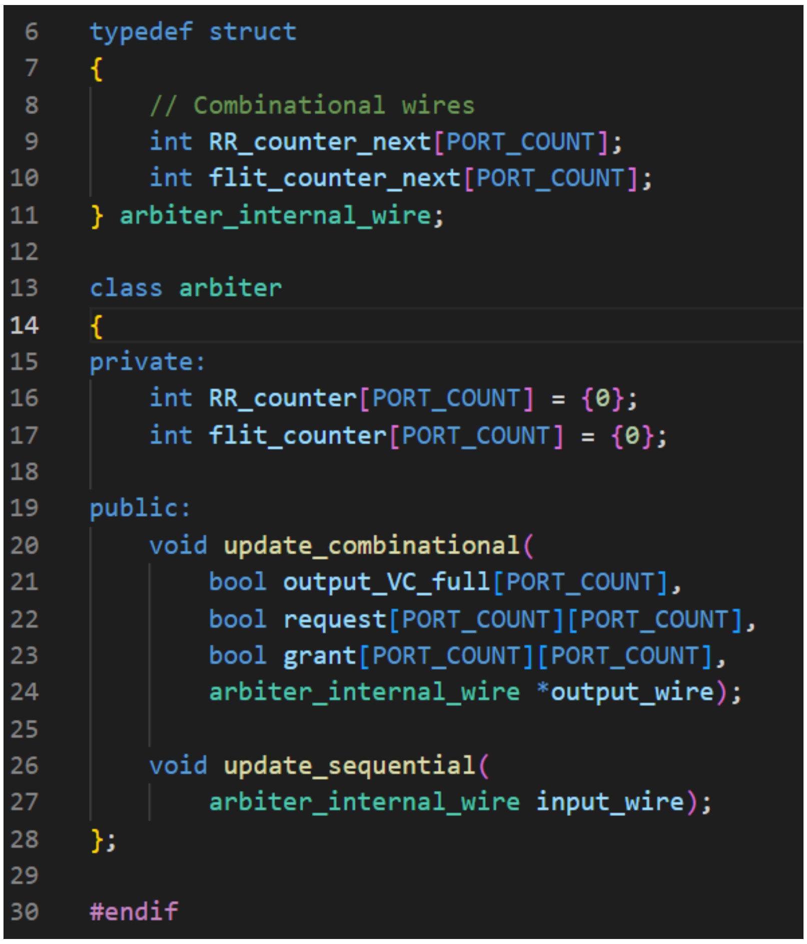
```
void top_ordering(int I, int *O){
  static foo_class foo1,foo2;
  int tmp1, tmp2;
          //combinational-query behaviors
          foo1.query(I,&tmp1);
          foo2.query(tmp1,&tmp2);
          *O=tmp2;
          //state-update behaviors
          foo1.update(tmp2);
          foo2.update(tmp1);
}
```

Figure 10: RTL-like design style

We implement our project in this kind of RTL-like design style, where we split the design into two parts, combinational and sequential. As figure 11 shown, we declare registers **RR_counter** and **flit_counter** in the class's private variable, and two functions combinational and sequential in the public function, which can be called by the top module's combinational and sequential function, respectively.

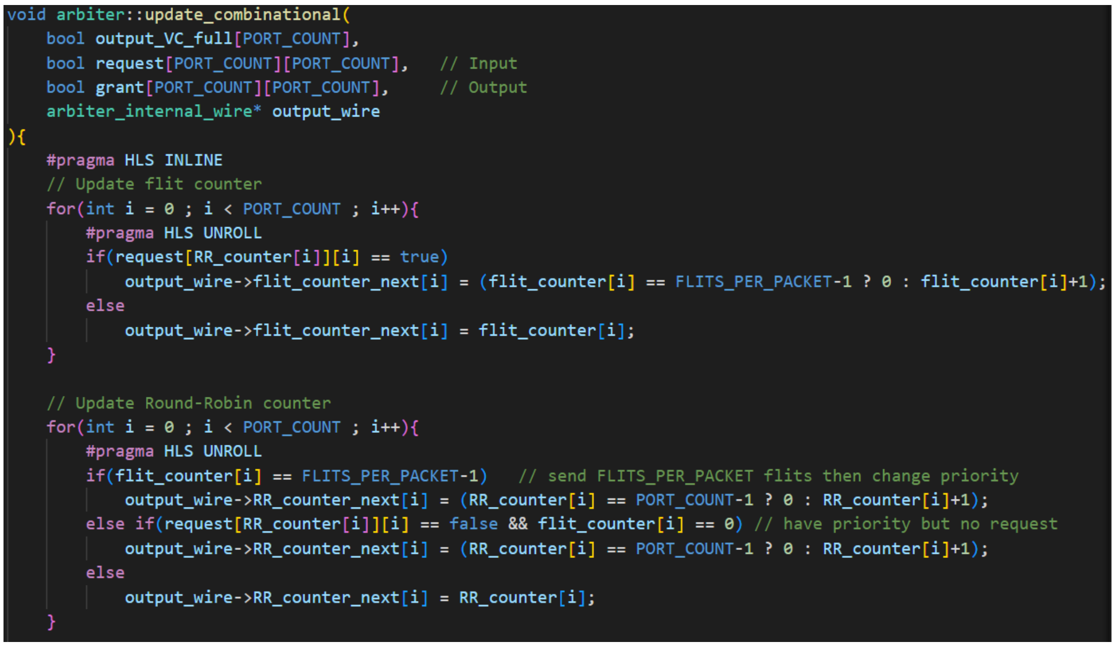
Since the combinational part will generate wires that contain the value that needs to be stored in registers or memory, we define a structure that includes all temporal wire signals. E.g., **RR_counter_next** and **flit_counter_next**, which will be assigned to RR_counter and flit_counter at the clock edge, respectively.

Figure 12 and 13 show the details of the combinational and sequential function, the first one generates the next value based on register and inputs, and the other one assigns the next value to registers, which is very similar to **always block** in RTL coding.



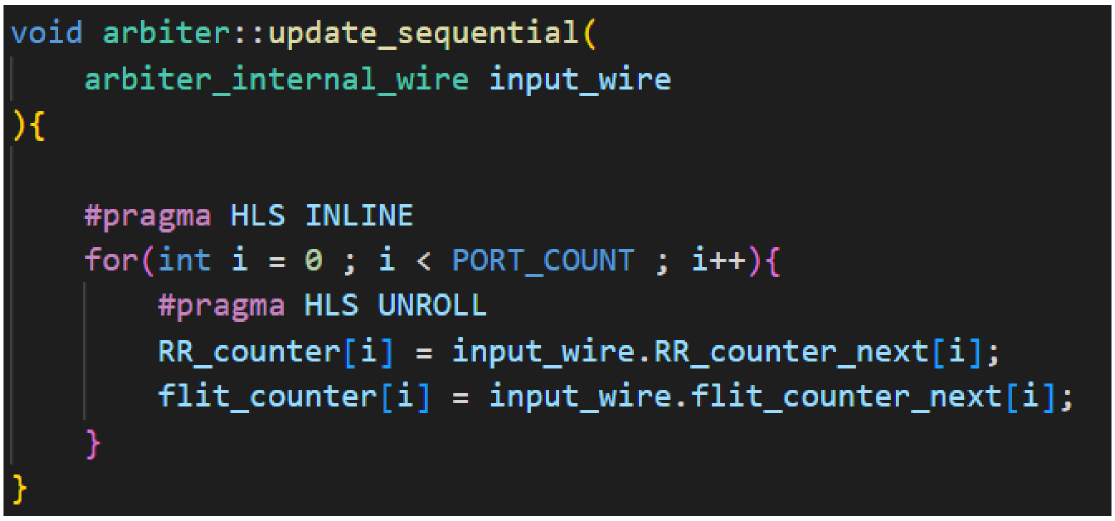
```cpp
typedef struct
{
    // Combinational wires
    int RR_counter_next[PORT_COUNT];
    int flit_counter_next[PORT_COUNT];
} arbiter_internal_wire;

class arbiter
{
private:
    int RR_counter[PORT_COUNT] = {0};
    int flit_counter[PORT_COUNT] = {0};

public:
    void update_combinational(
        bool output_VC_full[PORT_COUNT],
        bool request[PORT_COUNT][PORT_COUNT],
        bool grant[PORT_COUNT][PORT_COUNT],
        arbiter_internal_wire *output_wire);

    void update_sequential(
        arbiter_internal_wire input_wire);
};

#endif
```

Figure 11: Prototype of Arbiter in RTL-like design style



```cpp
void arbiter::update_combinational(
    bool output_VC_full[PORT_COUNT],
    bool request[PORT_COUNT][PORT_COUNT],   // Input
    bool grant[PORT_COUNT][PORT_COUNT],     // Output
    arbiter_internal_wire* output_wire
){
    #pragma HLS INLINE
    // Update flit counter
    for(int i = 0 ; i < PORT_COUNT ; i++){
        #pragma HLS UNROLL
        if(request[RR_counter[i]][i] == true)
            output_wire->flit_counter_next[i] = (flit_counter[i] == FLITS_PER_PACKET-1 ? 0 : flit_counter[i]+1);
        else
            output_wire->flit_counter_next[i] = flit_counter[i];
    }

    // Update Round-Robin counter
    for(int i = 0 ; i < PORT_COUNT ; i++){
        #pragma HLS UNROLL
        if(flit_counter[i] == FLITS_PER_PACKET-1)   // send FLITS_PER_PACKET flits then change priority
            output_wire->RR_counter_next[i] = (RR_counter[i] == PORT_COUNT-1 ? 0 : RR_counter[i]+1);
        else if(request[RR_counter[i]][i] == false && flit_counter[i] == 0) // have priority but no request
            output_wire->RR_counter_next[i] = (RR_counter[i] == PORT_COUNT-1 ? 0 : RR_counter[i]+1);
        else
            output_wire->RR_counter_next[i] = RR_counter[i];
    }
```

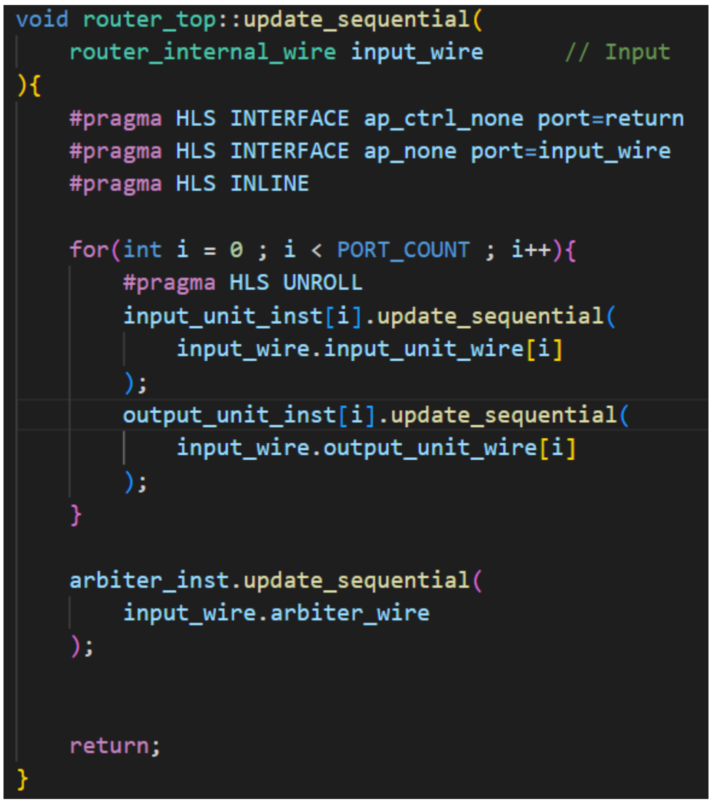
Figure 12: Part of combinational circuit description



```
void arbiter::update_sequential(
    arbiter_internal_wire input_wire
){

    #pragma HLS INLINE
    for(int i = 0 ; i < PORT_COUNT ; i++){
        #pragma HLS UNROLL
        RR_counter[i] = input_wire.RR_counter_next[i];
        flit_counter[i] = input_wire.flit_counter_next[i];
    }
}
```

Figure 13: Sequential circuit description

We keep this update mechanism in the design hierarchy that the upper-level combinational function will include the lower-level combinational function, and so do the sequential one. Figure 14 shows the sequential part of top module, which update all lower-level modules' sequential part.



```
void router_top::update_sequential(
    router_internal_wire input_wire      // Input
){
    #pragma HLS INTERFACE ap_ctrl_none port=return
    #pragma HLS INTERFACE ap_none port=input_wire
    #pragma HLS INLINE

    for(int i = 0 ; i < PORT_COUNT ; i++){
        #pragma HLS UNROLL
        input_unit_inst[i].update_sequential(
            input_wire.input_unit_wire[i]
        );
        output_unit_inst[i].update_sequential(
            input_wire.output_unit_wire[i]
        );
    }

    arbiter_inst.update_sequential(
        input_wire.arbiter_wire
    );


    return;
}
```

Figure 14: Top module's sequential function

5. Performance and Results

We finished Vitis HLS flow and Vitis acceleration card prototyping with the following result in table 1. Since we use flip-flops (FF) to build our FIFO buffer, it consumes lots of FF, and the critical part is LUT usage. I have experience in RTL design on FPGA, and I don't think this kind of design without complicated long critical paths needs to consume 10% LUT resource; it may be the HLS tool pitfall that it doesn't suitable for structural design.

For the single router throughput part, we suppose the downstream router is always non-blocking. That is, VC_full from the downstream router is always full. Each router has $104\ bits \times 5\ ports \times 100MHZ = \mathbf{6.5GBps}$ maximum input rate, so the ideal throughput will be $6.5GBps$, which happened at the condition that there is no competition between input units. The worst case will occur when all input flits are forwarded to the same output ports, which have only $104\ bits \times 1\ port \times 100MHZ = \mathbf{1.3\ GBps}$. In our experiment, We create random 10,000 flits to 5 input ports, and each port will be fed 2,000 flits. The total latency is 2188 cycles, so the throughput is $2{,}000\ flits \times 5\ ports * 104bits \div (2188\ cycles * 10ns) = \mathbf{5.94\ GBps}$

Then we consider a 4-routers system, which has 4 NI output as shown in figure 1. The ideal throughput should be $104\ bits \times 1\ port \times 100MHZ \times 4\ NIs = \mathbf{5.2\ GBps}$. The worst case throughput should be **1.3** GBps if all flits are going to the same NI. In our random testcase, it takes 3,475 cycles to transmit 2,500 flits each NI, a total of 10,000 flits. The actual throughput is $2{,}500\ flits \times 4\ NIs * 104bits \div (3{,}475\ cycles * 10ns) = \mathbf{3.74\ GBps}$

Table 1: Vitis HLS result

Item	Result	Utilization
Board	AlveoU280	
Clock rate	100 MHZ	
FF	49987	1%
DSP	0	0%
LUT	131283	10%
Actual throughput (single router)	5.98 GBps	
Ideal throughput (single router)	6.50 GBps	92%
Worst throughput (single router)	1.30 GBps	
Actual throughput (system)	3.74 GBps	
Ideal throughput (system)	5.20 GBps	72%
Worst throughput (system)	1.30 GBps	

6. Conclusion & Contribution

Xilinx Vitis is a powerful tool for application acceleration in HLS, but it is not developed for structural design like SystemC or Chisel. Its purpose is for fast architecture exploration of high-level applications, for example, RestNet50 and signal processing acceleration. In our experimental result, Xilinx HLS supports the structural design method but does not perform well. Also, the lack of structural design documentation and related discussion increased the workload in this project. In the project, we explore the probability of structural design and show that it can work but not behave as we expect (Synthesized result and resource utilization). In the end, we decide to use Verilog to describe our multi-core system in the future by Xilinx Vivado as IP, and integrate it with Vitis, whose software stack is still useful.

7. Reference

[1] Using Vivado-HLS for Structural Design: a NoC Case Study

[2] principle and practice of interconnection network

[3] HLS text book from Bridge of Life Education

[4] Parallel Programming for FPGA