
DaFPGA Switch

PROJECT REPORT

Teng Jiang	tj2488
Ilgar Mammadov	im2703
Irfan Tamim	it2304
Lauren Chin	lmc2265
Fathima Hakeem	fh2486

Contents

1	Overview	4
2	Block Diagram	5
3	Packet Processing and Metadata Handling	6
4	Control Memory Management Unit (CMU)	9
5	Egress Buffer	11
6	Virtual Output Queues Management Unit: VMU	12
7	Networking Fabric: Crossbar and buffers	13
8	Scheduling Algorithm	14
8.1	Doubly Round Robin	14
8.2	Priority-Based Scheduling	14
9	Simple Switch: Hardware	15
10	Software	16
10.1	Kernel Space Device Driver	16
10.2	User Space Program	17
11	Miscellaneous	19
11.1	Time line	19
11.2	Task Distribution	19
11.3	Takeaways	19
12	Appendix: Source Code	20
12.1	Simple Switch: Hardware	20
12.1.1	Crossbar	20
12.1.2	Egress	21
12.1.3	Ingress	22
12.1.4	Virtual Output Queues	23
12.1.5	Scheduler	25
12.1.6	Simple Dual Port	28
12.1.7	Simple Interface	29
12.1.8	Simple Switch	32
12.1.9	Verilator Makefile	37
12.1.10	Virtual Memory Unit	39
12.2	Simple Switch: Software	41
12.2.1	Even Load	41
12.2.2	Even Load (w/ Priorities)	45
12.2.3	Simple Switch	49
12.2.4	Skewed Load	55
12.2.5	Skewed Load (w/ Priorities)	59
12.3	Simple Switch: Verilator	63
12.3.1	Pick VOQ	63
12.3.2	VMU	65

12.3.3	Scheduler	66
12.3.4	Simple Switch	68
12.4	daFPGASwitch: Hardware	71
12.4.1	CMU	72
12.4.2	Crossbar	77
12.4.3	daFPGASwitch	78
12.4.4	Egress (Packet Validation)	80
12.4.5	Hardware-Software Interface	85
12.4.6	Ingress	88
12.4.7	Scheduler	95
12.4.8	Packet Generation	98
12.4.9	True Dual Port Memory	102
12.4.10	VMU	103

1 Overview

The primary objective of this project is to implement a hardware-based network switch using an FPGA (Field-Programmable Gate Array).

A network switch is a dedicated hardware component tasked with connecting multiple computers and networking devices within a computer network. The primary function of a network switch is to route incoming data packets to the correct destination ports, guided by the destination addresses encoded within each packet.

This process facilitates the transfer of data across a Local Area Network (LAN) or Wide Area Network (WAN). For our project, we aim to replicate the functionality of a network switch using an FPGA, operating with a simplified, yet streamlined packet structure for simulation purposes.

For this project, we simulate a 4-input, 4-output switch, and conduct performance evaluations.

For more, please directly refer to our GitHub repo: <https://github.com/daFPGASwitch/daFPGASwitch/>

2 Block Diagram

Conceptually, our system runs in the following fashion: The software generates the packet metadata and sends them to the ingress modules. The ingress modules generate packets and decide which packet gets sent onto the crossbar (switch fabric). The egress modules receive data from the crossbar and send it back to the software.

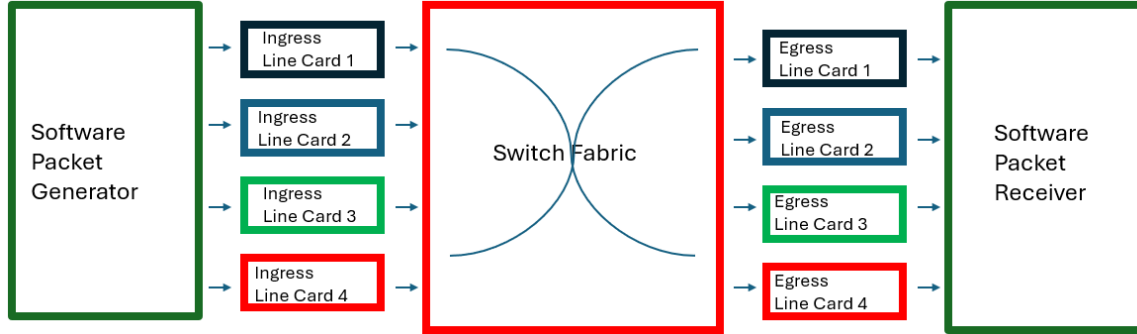


Figure 1: Block Diagram of the whole system, with 2 ingresses and 2 egresses as an example.

As we proposed in the proposal, we're going to maintain virtual output queues per egress port, as illustrated below, inside the ingress port module. So most of the functionalities are inside each ingress port.

For simplicity of explanation, we have illustrated two ingress and two egress ports in the following block diagram and individual stages contained in the ingress. Once the metadata is received through the ingress, converted to packets, processed in the memories, it is passed to the crossbar to be sent to the correct output port. In order to generate a packet from the metadata, we have created a state machine. The metadata format and packet format have been explained in the next section.

Input state machine:

- 1) Length is described as blocks in the metadata, and it is multiplied by 32 to be stored as the number of bytes in the real packet. 16 bits of the Destination MAC is stored in the remaining part of the packet.
- 2) The remaining 32-bits of the Destination MAC is stored.
- 3) Current time is added as a "Start Time".
- 4) End Time is set to all zeros in the ingress state machine, which will be modified in the egress.
- 5) 32-bits of Source MAC is stored.
- 6) The remaining 16 bits of Source MAC is stored and 16 bits are padded with zeros.
- 7) Data Payload is generated which is never checked in the system.
- 8) Data Payload is continued to be generated according to the specified packet length.

Output state machine:

- 1) Scheduler Enable signal is received with a decision.
- 2) The selected VOQ dequeues the address and sends the address to CMU.
- 3) The data stored in the received address is sent to the crossbar and CMU returns the next block address.

4) As the data is stored as 32-bit chunks in the data memory, an offset starts to be counted and at the end of the 8th cycle, if the next block address received by CMU is not 0, it means the packet is continuing and we move to the third state. If the packet is finished, the state machine can move to the idle state again.

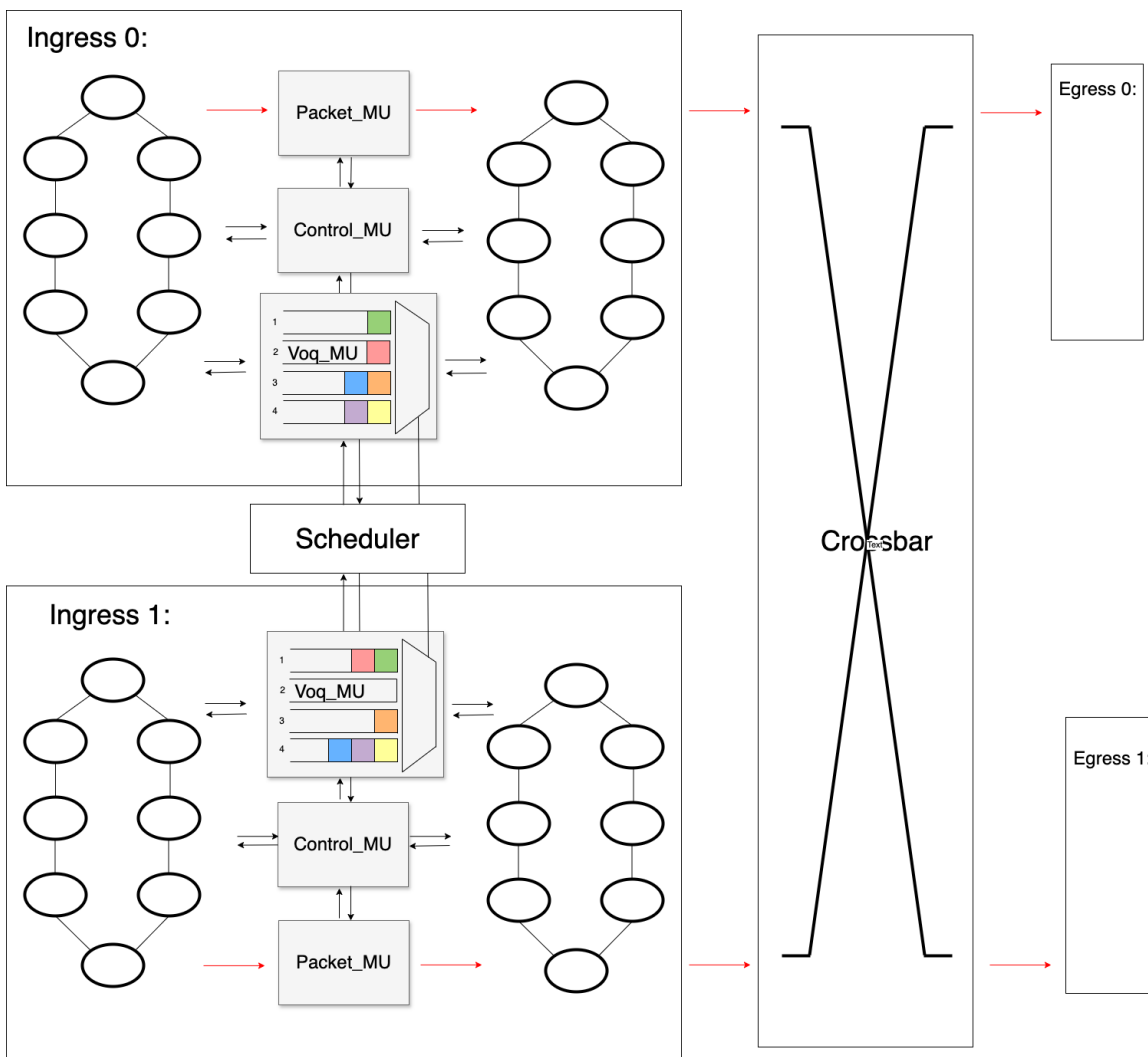


Figure 2: Block Diagram of the whole system.

3 Packet Processing and Metadata Handling

Each packet contains a 32-byte header, composed of five encoded segments used for packet handling and processing:

1. length, 2. destination MAC address, 3. start and end timestamps, 4. source MAC address, and 5. a payload.

Length (2 bytes)	Dest MAC (6 bytes)	Start Time (4 bytes)	End Time (4 bytes)	Src MAC (6 + 2 empty bytes)	Data Payload Variable length
---------------------	-----------------------	-------------------------	-----------------------	--------------------------------	---------------------------------

Figure 3: Packet Format

As the maximum length can be 1500 bytes in the Ethernet data frame, the "Length" segment should be 2 bytes to accommodate "1500" as a length. The Source MAC address will be 48 bits long and does not

have any role in the functionality of the switch, but we keep it for realistic implementation. 32-bits are used to record the time stamp of when the packet transmission starts. This is when the packet first arrives at the ingress port. Another 32 bits are used to record the time stamp of when the entire packet arrives at the egress, to mark when the packet transmission is complete. The Destination MAC address will be 48 bits long and it will be used for MAC-to-Port translation. Lastly, the data part will be variable length.

We need to know how many memory chunks we have to assign to the packet, so the "Length" part will be first in the data frame.

The software sends metadata instead of a real packet and the packet is generated in the hardware as described in the input state machine. The metadata format has been described below. The destination and source ports are 2 bits each, as we have 4 ports in each side. As length is stored as the number of blocks, 6 bits are enough to store the maximum length $n=in$ blocks.

Dest Port (2 bits)	Src Port (2 bits)	Length (6 bits)	Start Time (11 bits)	End Time (11 bits)
-----------------------	----------------------	--------------------	-------------------------	-----------------------

Figure 4: Metadata Format

The following figure shows the timing diagram of generating new packets based on the software-defined metadata. The metadata is read as an input. Metadata arrives once every cycle and represents information necessary to create an entire packet. The module then creates the packet based on the defined packet format. A new packet is sent out of the output port 4 bytes every cycle, as well as an enable signal for the ingress module to start receiving.

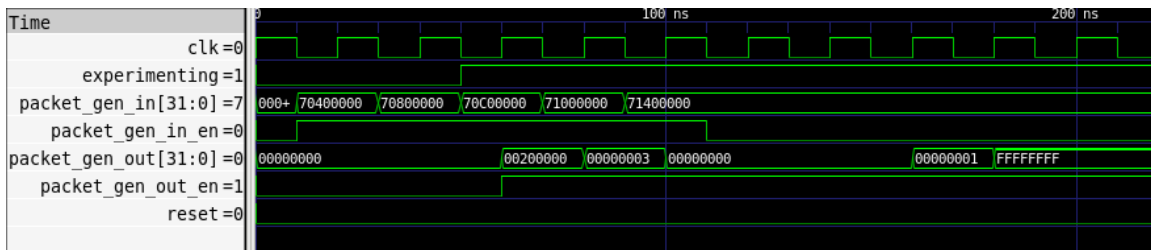


Figure 5: Hardware generating packets from software instructions

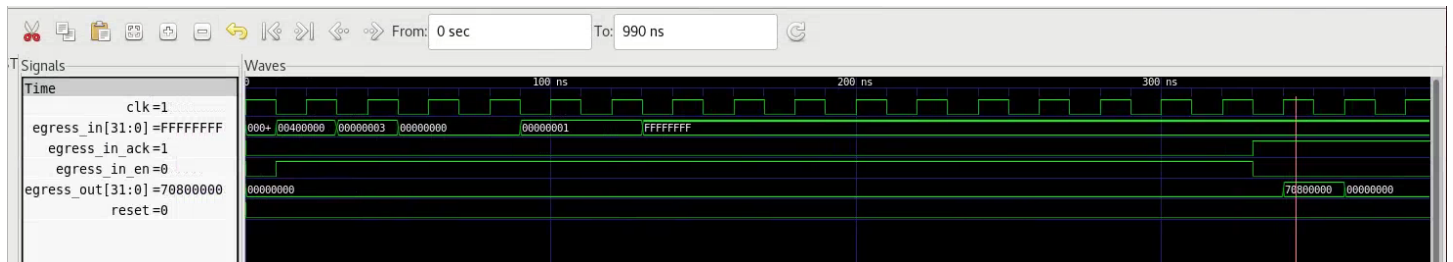


Figure 6: Packet to Meta conversion

In Figure 5, the verilog output of the egress state machine has been described which is about converting the whole packet to metadata to be sent to the software for evaluation. It can be seen that a packet of length 64 bytes (2 blocks) is received by the egress: In the first cycle (2 bytes length + 2 bytes DMAC) is received, in the second cycle the rest of the destination MAC is received, which is followed by start time,

end time, source MAC and data payload. Note that the payload is not important for us for analysis, so we pad it with ones. The main functionality of the state machine is to receive all the 32-bit chunks of the packet and combine them to one 32-bit metadata, and assert "packet out". It can be seen that when the "packet out" is asserted, the correctly combined metadata is sent to output, which is:

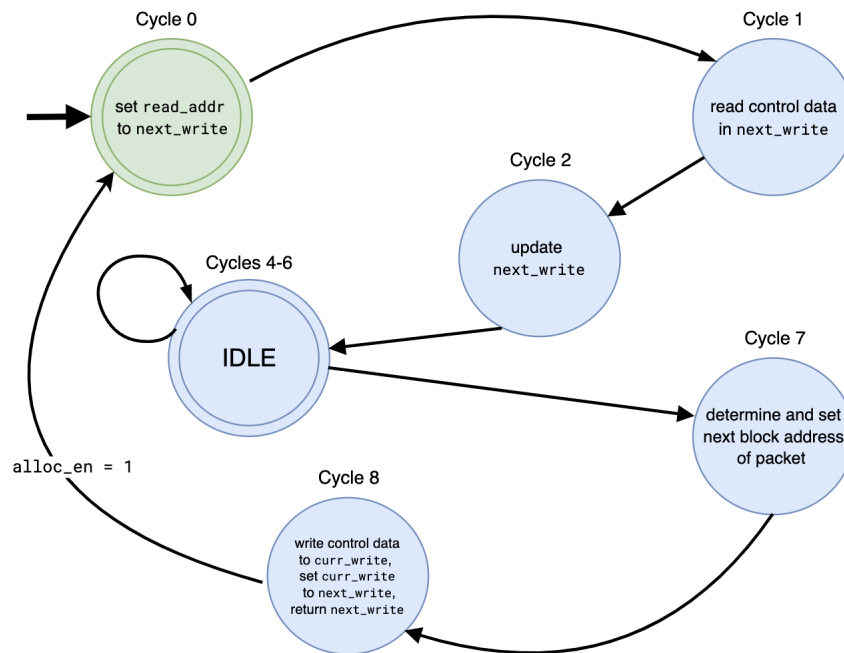
70800000 → (01 11 000010 000000000000000000000000)

It can be seen that the source and destination ports have been assigned correctly as from 1 to 3, the length is 2 blocks. As we did not assign any time stamp in the testbench, the time stamp is padded with zeros.

4 Control Memory Management Unit (CMU)

There are four ingress ports. Each ingress has a Control Memory (CMEM), which is solely managed by the Control Memory Management Unit (CMU). CMEM is modeled as a true dual-port memory. In other words, both ports of the CMEM are able to handle read and write requests. Due to the specifications of the M10K RAM blocks of the Altera FPGA, the data width of the CMEM is maintained at 16 bits.

Each control data written to CMEM has 11 bits. The most significant bit (MSB) asserts if the associated address is already allocated. The remaining 10 bits point to the next available memory address, essentially creating a linked list, or chain, of allocated control blocks. This keeps track of where the next segments of the packet are. These 10 bits can be 10'b0 to indicate the end of the chain, implying that the current address represents the last packet segment.

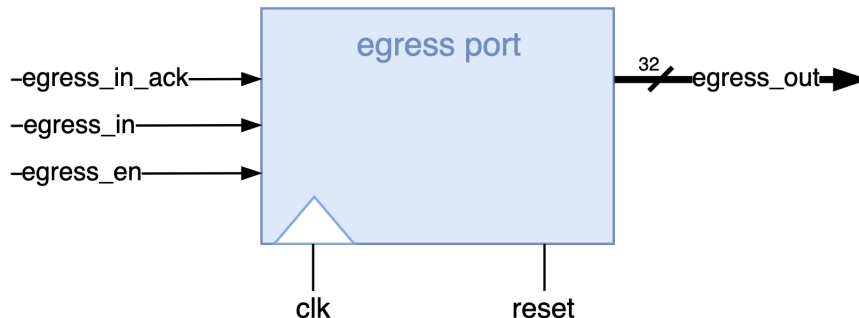


CMU is in charge of determining and allocating an address to which the new packet segment should be written to. CMU also manages read and de-allocate requests. The CMU contains two machines: input state machine and output state machine. The input state machine, shown in the figure above, is triggered by an enable signal from the ingress module. The CMU algorithm successfully assigns the next available memory address on the current 8-cycle heartbeat, so delays can be minimized when new packet segments arrive on the next heartbeat.

The input state machine starts by reading the control of the next available address, called `next_write`. The least significant 10 bits of this control can either be 10'b0 or indicate a different address. If it is 10'b0 then `next_write` is incremented by 1. Otherwise, `next_write` adopts the value from the control. On the last cycle of the heartbeat, an updated control is formatted to write to the address where the current packet is being written. The formatted control contains 1 as the MSB and either `next_write` or 10'b0 based on the necessity of more space to store the entire packet. Finally, the address where the next packet will be written to adopts `next_write`.

5 Egress Buffer

There are four egress ports and each of them will have an associated ring buffer and a state machine. Like typical ring buffers, the memory will be connected end to end. Two pointers, one head and one tail, will chase each other to read and write at the correct addresses. The buffers will read out data in a FIFO manner. The egress buffer will inform the software when new packets arrive. The software will request the egress buffers to read out the 32-bit metadata at a time.



The state machine: The state machine in each egress receives the packet in 32 bits at a time and converts it to metadata using the reverse process described in the input state machine.

Additional details regarding the egress buffer are as follows:

- The parameters 'PACKET_CNT', 'BLOCK_SIZE', and 'META_WIDTH' define the scale and data structure of the module.
- Inputs include 'clk' (the clock signal), 'reset' (to reinitialize the module), 'egress_in' (data incoming from the crossbar module), 'egress_in_en' (a signal indicating whether the incoming data is valid), and 'egress_in_ack' (signal acknowledging receipt or processing of previously sent data).
- A single output 'egress_out' sends processed data to another part of the system.
- Two indices, 'start_idx' and 'end_idx', manage where data is written to and read from within a dual-port memory. These indices ensure data integrity by implementing a circular buffer mechanism, where 'end_idx' is incremented with each new data entry and 'start_idx' is incremented following an acknowledgment signal.
- A 'simple_dual_port_mem' memory instantiation is parameterized with the size equal to 'PACKET_CNT' and a data width of 32. This memory allows simultaneous read and write operations. The memory's read and write addresses are controlled by 'start_idx' and 'end_idx', respectively.

6 Virtual Output Queues Management Unit: VMU

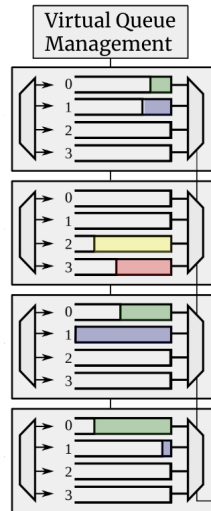


Figure 8: Virtual Output Queues of four input modules. Each of the queues inside one module corresponds to one output module.

The Virtual Output Queue Management Unit (VMU) contains the logic and a memory block (herein referred to as VOQs). It is used to store the first address of each packet. When a new packet comes, the ingress stores the first address of the packet in the VMU. When the scheduler decision is made, the corresponding first address of the packet is dequeued and returned. The first address of the packet is then sent to the CMU to be freed, and retrieve the next block of the packet from the data memory.

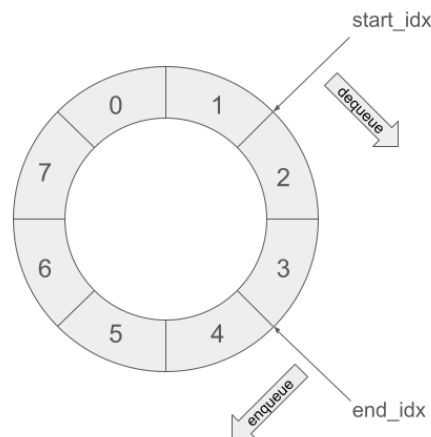


Figure 9: Illustration of the Ring buffer.

VMU is implemented as a ring buffer: The ingress dequeues from `start_idx` and enqueued from `end_idx`.

VMU also provides information about how full each ingress is and whether the ingress is busy with a packet, for the scheduling to make up its scheduling decision.

Each ingress port has four VOQs embedded within each ingress buffer, each corresponding to one egress port; each VOQ has a dedicated block of memory within its corresponding ingress port's memory block.

When we dequeue a packet from the VMU, we will take the stored value and send it to the CMU, to get the next address of the packet. If we hit a zero, then it means that we hit the end of the packet.

The CMU and MAC-to-port translation table facilitate the arrival of an incoming packet to its corresponding VOQ.

7 Networking Fabric: Crossbar and buffers

A crossbar, also known as a switching fabric, is a network topology that consists of a grid of intersecting buses, enabling direct and exclusive connections such that at any given time, each output port is connected to only one input port. Crossbar switches are a critical component of any network switch, as it improves throughput by allowing multiple, non-interfering data connections to be in use at any given time. Since crossbars provide a single unique path from an input to an output, only one packet must be chosen from the buffers at the input.

8 Scheduling Algorithm

8.1 Doubly Round Robin

We chose to mainly use a scheduling algorithm which we call "doubly round robin".

This algorithm targets fairness, a common practice in the real world. On average, each ingress gets an equal chance to be picked first; each virtual queue of ingress gets an equal chance to be picked first. The scheduling decision takes 5 cycles which is a reasonable time for as we are using an 8-cycle heartbeat.

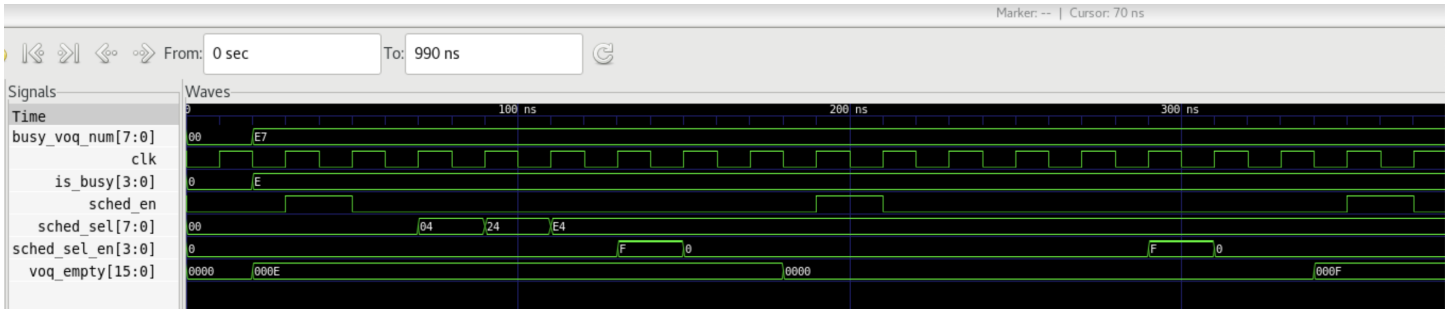


Figure 10: Verilator testbench for Scheduler

8.2 Priority-Based Scheduling

Another scheduling algorithm that we tried was priority-based scheduling, and we wanted to bias towards a specific egress in terms of VOQ selection.

We implement the priority by passing in a list that sets the priority of each egress. When each ingress considers its scheduling decision, it will prioritize sending the packets with higher priorities.

9 Simple Switch: Hardware

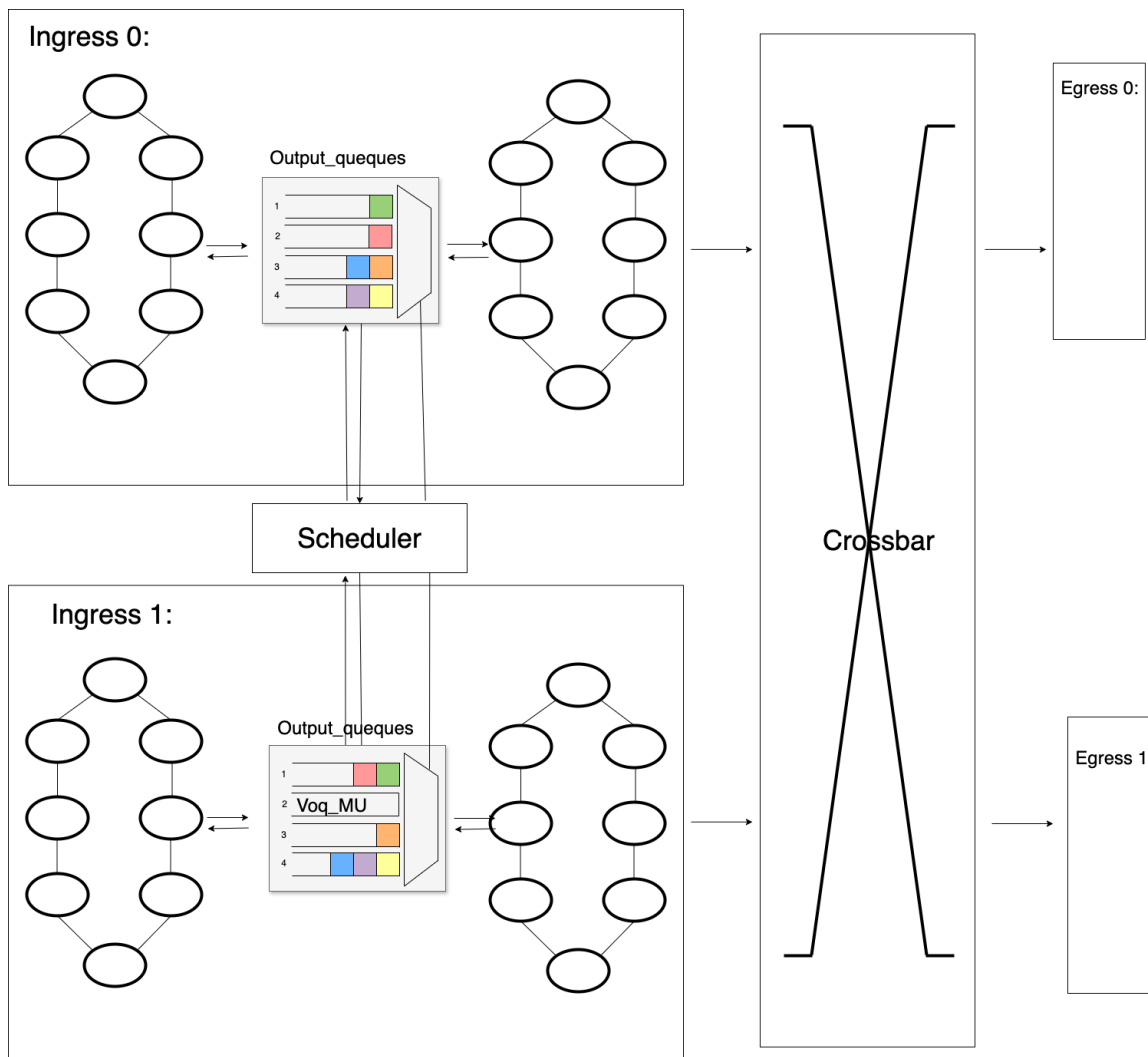


Figure 11: Block diagram of Simple Switch system

Everything we've talked about are implemented in SystemVerilog and tested successfully with the testbench tool Verilator. But it's intrinsically hard to integrate all parts together and we had to simplify some parts to be able to embed the project to the FPGA. Instead of having a memory controller to manage the data written to the memory and having virtual queues, we ended up using a real output queue. The data coming from software is being stored in the respective output queues and once the scheduler decision is made, the data is dequeued and send to the corresponding egress port through the crossbar. Once the data is received in the egress, the software can actively pull it.

10 Software

10.1 Kernel Space Device Driver

This is the layout of our registers lies between the software and the hardware:

We have one 32-bit register for ctrl data, usage marked in the figure; and four 32-bit registers for the 32-bit packet metadata, one for each port.

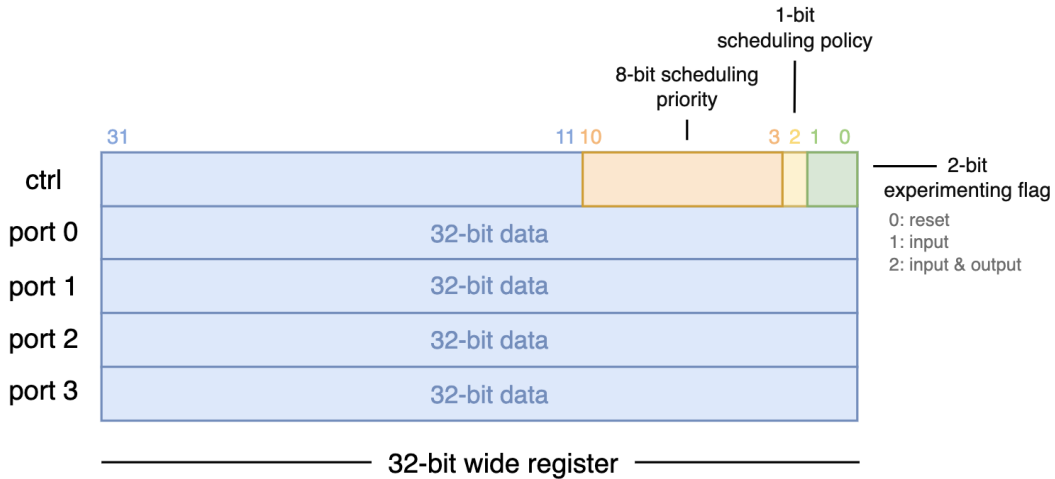


Figure 12: We use 5 32-bit registers for SW-HW communication. For the ctrl bit, the lowest two digits are for controlling the state of our experiment: 0 means we're resetting everything, 1 means that our switch only accepts input but does not output (so that we can create congestion), and the rest of the bits are for the scheduler.

Our peripheral is implemented as an Avalon Slave as shown below.

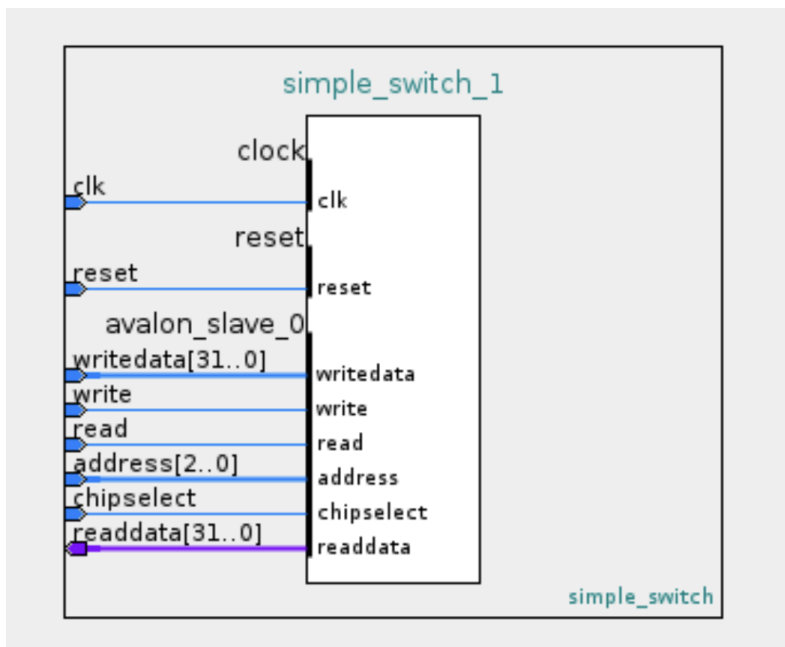


Figure 13: The simple switch Avalon bus slave.

Software "polls" with `ioread32`, which generates a high "read" signal for the Avalon slave. For HW, the read signal is like an ack signal ("read" means that sw has already consumed the packet segment).

Software "interrupts" with `iowrite32`, which generates a high "write" signal for the Avalon slave. For HW, the write signal is like an enable signal ("write" means that sw has already put the `packet_data` on the `writedata` wire.)

10.2 User Space Program

We read from and write to hardware with `ioctl()`:

```
1 // Write to HW
2 if (ioctl(simple_switch_fd, SIMPLE_WRITE_PACKET, pkt_meta) < 0) {
3     perror("Failed to send packet");
4     return;
5 }
6 // Read from HW
7 if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_0, &rcvd_pkt_meta) < 0) {
8     perror("ioctl() read packet failed");
9     close(simple_switch_fd);
10    return -1;
11 }
```

Test Output from Userspace Program

Starting userspace program...

```
Sent: metadata 0x0001      0 0 0 0
Sent: metadata 0x1040      0 1 1 0
Sent: metadata 0x4040      1 0 1 0
Sent: metadata 0xb040      2 3 1 0
Sent: metadata 0x9040      2 1 1 0
Sent: metadata 0xc040      3 0 1 0
Sent: metadata 0xe1c0      3 2 7 0
Sent: metadata 0x8040      2 0 1 0
```

...

/* omitted for brevity */

...

Requested received packets from ports 0-3

Requested 54 packets

```
Port 1: metadata 0x9040    2 1 1 0 1
Port 0: metadata 0x8040    2 0 1 0
Port 3: metadata 0xb040    2 3 1 0
Port 0: metadata 0xc040    3 0 1 0
Port 1: metadata 0x9040    2 1 1 0
Port 2: metadata 0xe1c0    3 2 7 0
```

...

Got 54 packets

Total latency: 222

Userspace program terminating

We generate 2 different kinds of patterns to test out the validity of our scheduling algorithm.

11 Miscellaneous

11.1 Time line

We divided the switch implementation into the following steps:

- Architecture and interface definition: Considering the involvement of several stages in the system, we had to first draw the block diagram and determine the protocol between individual stages.
- Simulation: We simulated the whole system successfully starting from packet generation to packet validation. This step helped us a lot to generate and understand the necessary algorithms.
- Hardware implementation: Once we defined the algorithms in the simulation, we implemented the same system in the hardware.
- Software-hardware interaction: After testing the modules with verilator, we configured the device driver and created the communication between software and hardware.

11.2 Task Distribution

As our switch had several individual modules, we could successfully divide work while everyone also had an input in all parts.

- Teng: Scheduler simulation, Top-level software simulation, scheduler hardware implementation, all software-hardware interface implementation, device driver implementation, all voq implementation. Helped with packet generation and validation.
- Irfan: Control Memory and Data Memory simulation and hardware implementation. Helped with packet generation. Worked on debugging individual modules for verilator simulation
- Ilgar: CMU simulation and input and egress state machine software and hardware implementation.
- Fathima: Scheduler simulation and egress hardware implementation.
- Lauren: Scheduler simulation, output and egress state machine hardware implementation.

11.3 Takeaways

- Teng: Get an MVP and load it onto FPGA as soon as possible. Also, the verilator helped us a lot because we're super hardware-heavy.
- Irfan: It is a lot harder to make the entire system work together with all the modules, than to have individual modules working by themselves.
- Ilgar: Combinational logic can be dangerous for compiling and we should prevent latches in state machines.
- Fathima: Simulating the design in software is crucial for pinning down how different algorithms operate and for ensuring that all the interfaces integrate properly.
- Lauren: Have trust for your teammates.

12 Appendix: Source Code

12.1 Simple Switch: Hardware

12.1.1 Crossbar

```
1
2 module crossbar #(
3     parameter DATA_WIDTH = 32, // For testing
4     parameter EGRESS_CNT = 4
5 ) (
6     input clk,
7     input logic [ $clog2 ( EGRESS_CNT ) * EGRESS_CNT - 1 : 0 ] sched_sel,
8     input logic [ EGRESS_CNT - 1 : 0 ] crossbar_in_en,
9     input logic [ DATA_WIDTH * EGRESS_CNT - 1 : 0 ] crossbar_in,
10
11     output logic [ EGRESS_CNT - 1 : 0 ] crossbar_out_en,
12     output logic [ DATA_WIDTH * EGRESS_CNT - 1 : 0 ] crossbar_out
13 );
14
15 always_ff @(posedge clk) begin
16     crossbar_out_en [ sched_sel [ 0 * $clog2 ( EGRESS_CNT ) + :
17         $clog2 ( EGRESS_CNT ) ] ] <= crossbar_in_en [ 0 ];
18     crossbar_out [ ( sched_sel [ 0 * $clog2 ( EGRESS_CNT ) + :
19         $clog2 ( EGRESS_CNT ) ] ) * DATA_WIDTH + : DATA_WIDTH
20 ] <= crossbar_in [ 0 * DATA_WIDTH + : DATA_WIDTH ];
21     crossbar_out_en [ sched_sel [ 1 * $clog2 ( EGRESS_CNT ) + :
22         $clog2 ( EGRESS_CNT ) ] ] <= crossbar_in_en [ 1 ];
23     crossbar_out [ ( sched_sel [ 1 * $clog2 ( EGRESS_CNT ) + :
24         $clog2 ( EGRESS_CNT ) ] ) * DATA_WIDTH + : DATA_WIDTH
25 ] <= crossbar_in [ 1 * DATA_WIDTH + : DATA_WIDTH ];
26     crossbar_out_en [ sched_sel [ 2 * $clog2 ( EGRESS_CNT ) + :
27         $clog2 ( EGRESS_CNT ) ] ] <= crossbar_in_en [ 2 ];
28     crossbar_out [ ( sched_sel [ 2 * $clog2 ( EGRESS_CNT ) + :
29         $clog2 ( EGRESS_CNT ) ] ) * DATA_WIDTH + : DATA_WIDTH
30 ] <= crossbar_in [ 2 * DATA_WIDTH + : DATA_WIDTH ];
31     crossbar_out_en [ sched_sel [ 3 * $clog2 ( EGRESS_CNT ) + :
32         $clog2 ( EGRESS_CNT ) ] ] <= crossbar_in_en [ 3 ];
33     crossbar_out [ ( sched_sel [ 3 * $clog2 ( EGRESS_CNT ) + :
34         $clog2 ( EGRESS_CNT ) ] ) * DATA_WIDTH + : DATA_WIDTH
35 ] <= crossbar_in [ 3 * DATA_WIDTH + : DATA_WIDTH ];
36
37 end
38
39 endmodule
```

12.1.2 Egress

```
1 module egress #(
2     parameter PACKET_CNT = 1024,
3     BLOCK_SIZE = 32,
4     META_WIDTH = 32
5 ) (
6     input logic clk,
7     input logic reset,
8
9     // From crossbar
10    input logic [ META_WIDTH -1 : 0 ] egress_in,
11    input logic          egress_in_en,
12
13    // From interface
14    input logic egress_in_ack,
15
16    // To interface
17    output logic [ 31 : 0 ] egress_out
18 );
19
20 logic [ $clog2 ( PACKET_CNT ) -1 : 0 ] start_idx;
21 logic [ $clog2 ( PACKET_CNT ) -1 : 0 ] end_idx;
22
23 always @(posedge clk) begin
24     if (reset) begin
25         start_idx <= 0;
26         end_idx <= 0;
27     end
28     if (egress_in_en) begin
29         end_idx <= (end_idx != 1023) ? end_idx + 1 :0;
30     end
31
32     if (egress_in_ack) begin
33         start_idx <= (start_idx != 1023) ? start_idx + 1 :0;
34     end
35 end
36
37 simple_dual_port_mem #(
38     .MEM_SIZE (PACKET_CNT),
39     .DATA_WIDTH(32)
40 ) meta_mem (
41     .clk(clk),
42     .ra(start_idx),
43     .wa(end_idx),
44     .d(egress_in),
45     .q(egress_out),
46     .write(egress_in_en)
47 );
48
49 endmodule
```

12.1.3 Ingress

```
1
2
3 /* verilator lint_off UNUSED */
4 module ingress #(
5     parameter PACKET_CNT = 1024,
6     BLOCK_SIZE = 32,
7     META_WIDTH = 32
8 ) (
9     input logic          clk,
10    input logic          reset,
11
12    input logic  [ META_WIDTH -1 : 0 ] ingress_in,
13    input logic          ingress_in_en,
14    input logic          experimenting,
15    input logic          sched_en,
16    input logic  [ 1 : 0 ]          sched_sel,
17    input logic  [ 10 : 0 ]         time_stamp,
18
19    // To crossbar
20    output logic  [ 31 : 0 ]        ingress_out,
21
22    // To scheduler
23    output logic  [ 3 : 0 ]         is_empty
24 );
25 logic  [ 1 : 0 ] port_num;
26 logic  [ 3 : 0 ] is_full;
27 logic  [ 31 : 0 ] meta_out;
28 assign port_num = ingress_in [ 29 : 28 ];
29 assign ingress_out = {meta_out [ 31 : 11 ], time_stamp};
30
31    vmu #(.PACKET_CNT(1024), .EGRESS_CNT(4)) voq_mu
32    (
33        // Input
34        .clk(clk), .reset(reset),
35        .voq_enqueue_en(ingress_in_en && !is_full [ port_num
36            ]), .voq_enqueue_sel(port_num),
37        .voq_dequeue_en(sched_en && !is_empty [ sched_sel
38            ]), .voq_dequeue_sel(sched_sel),
39        .meta_in({ingress_in [ 31 : 22 ], time_stamp, ingress_in [ 10 : 0
40            ]}),
41        .time_stamp(time_stamp),
42
43        // Output
44        .meta_out(meta_out),
45        .is_empty(is_empty), .is_full(is_full)
46    );
47
48 endmodule
```

12.1.4 Virtual Output Queues

```
1
2 module pick_voq (
3   input logic [ 1 : 0
4     ] start_voq_num, // the idx'th voq has the highest prio to be selected.
5   input logic [ 3 : 0 ] voq_empty, // Which egress/voq is empty.
6   input logic [ 7 : 0 ] prio,
7   input logic policy,
8   input logic [ 3 : 0 ] voq_picked, // Which egress/voq is picked.
9   output logic no_available_voq, // either all empty, or non-empty egress is taken.
10  output logic [ 1 : 0 ] voq_to_pick
11 );
12 always_comb begin
13   if (voq_empty == 4'b1111) begin
14     voq_to_pick = start_voq_num;
15     no_available_voq = 1'b1;
16   end else if (policy == 0) begin
17     if (voq_empty [ start_voq_num ] == 1'b0 && voq_picked [ start_voq_num
18       ] == 1'b0) begin
19       voq_to_pick = start_voq_num; // Save the index of the first non-zero bit
20       no_available_voq = 1'b0;
21     end else if (voq_empty [ start_voq_num + 1 ] == 1'b0 && voq_picked [
22       start_voq_num + 1 ] == 1'b0) begin
23       voq_to_pick = start_voq_num + 1;
24       no_available_voq = 1'b0;
25     end else if (voq_empty [ start_voq_num + 2 ] == 1'b0 && voq_picked [
26       start_voq_num + 2 ] == 1'b0) begin
27       voq_to_pick = start_voq_num + 2;
28       no_available_voq = 1'b0;
29     end else if (voq_empty [ start_voq_num + 3 ] == 1'b0 && voq_picked [
30       start_voq_num + 3 ] == 1'b0) begin
31       voq_to_pick = start_voq_num + 3;
32       no_available_voq = 1'b0;
33     end else begin
34       voq_to_pick = start_voq_num;
35       no_available_voq = 1'b1;
36     end
37   end else begin
38     if (voq_empty [ prio [ 7 : 6 ] ] == 1'b0 && voq_picked [ prio [ 7 :
39       6 ] ] == 1'b0) begin
40       voq_to_pick = prio [ 7 : 6 ]; // Save the index of the first non-zero bit
41       no_available_voq = 1'b0;
42     end else if (voq_empty [ prio [ 5 : 4 ] ] == 1'b0 && voq_picked [ prio
43       [ 5 : 4 ] ] == 1'b0) begin
44       voq_to_pick = prio [ 5 : 4 ];
45       no_available_voq = 1'b0;
46     end else if (voq_empty [ prio [ 3 : 2 ] ] == 1'b0 && voq_picked [ prio
47       [ 3 : 2 ] ] == 1'b0) begin
48       voq_to_pick = prio [ 3 : 2 ];
49       no_available_voq = 1'b0;
50     end else if (voq_empty [ prio [ 1 : 0 ] ] == 1'b0 && voq_picked [ prio
51       [ 1 : 0 ] ] == 1'b0) begin
52       voq_to_pick = prio [ 1 : 0 ];
53       no_available_voq = 1'b0;
```

```
45     end else begin
46         voq_to_pick = start_voq_num;
47         no_available_voq = 1'b1;
48     end
49 end
50 end
51
52 endmodule
```


12.1.5 Scheduler

```
1 module sched (
2     input logic clk,
3     input logic sched_en,
4     input logic [ 3 : 0 ] is_busy,
5     input logic [ 7 : 0 ] busy_voq_num,
6     input logic [ 15 : 0 ] voq_empty,
7     input logic policy, // We have doubly RR, or priority based, can be controlled by
8                           software
9     input logic [ 7 : 0 ] prio,
10    output logic [ 3 : 0
11    ] sched_sel_en, // passed by to ingress, to know which ingress should dequeue
12    output logic [ 7 : 0
13    ] sched_sel // passed by to ingress, to know which voq to dequeue
14);
15
16 /*
17 Some design choices:
18 * Do we want the time to return a scheduling decision to be deterministic or random
19   (btw 1 and 4)
20 * Should each busy port to just start transmitting without waiting for the
21   scheduling decision
22 * Should the scheduler decision the scheduling decision for this cycle or next cycle
23 * RR on the ingress or egress side, or both
24 * Should we try to do all combinator?
25 * Do we use 1 voq_to_pick or 4 voq_to_pick? (1 since it's going to take 4 cycles
26   anyway)
27 */
28
29 /*
30 Some principles:
31 * We don't want egress 1 to always recv packet from ingress 0; we don't want
32   ingress 0 to always send to egress 2
33 * nested loop fully in combinator logic is too expensive; instead, we do the outer
34   for loop sequentially, and the inner 4 loop in comb logic
35 RR policy:
36 * ingress RR: Start_ingress_idx proceed in each cycle of ASSIGN_NEW (or one pass
37   who-ever gets to select first in this cycle)
38 * egress RR: Each start_voq_num is first_non_empty_num of this cycle + 1;
39 * fewer first: Prioritize queue with only one non-empty voq.
40 */
41
42 /*
43 Notice:
44 * Beware of the possiblity of input (voq_empty for example) change during the
45   process.
46   - Possibly, use some local variables to save the inputs.
47   - Or, let the ingress be in charge of only updating the signal when sched_en.
48 * Busy ports need to be handled first.
49 * All the data need to be prepared before sched_enable. So at T-1 prepare input,
50   and at T sched_enable.
51 */
```

```

41
42 // If the scheduler is in the process of assigning new packet
43 // 0: not assigning; 1~4: assigning. 5 enable 6 is /enable
44 logic [ 2 : 0 ] assigning_new;
45
46 logic [ 3 : 0 ] ingress_enable; // the enable signal ready to be passed to
    sched_sel_en when ingress_done = 4'b1111
47
48 // For RR
49 logic [ 1 : 0
    ] start_ingress_idx; // Which ingress has the highest priority in this cycle
50 logic [ 7 : 0 ] start_voq_num; // Which egress has the highest priority in
    this cycle, for each ingress.
51
52 logic [ 1 : 0 ] curr_ingress_idx; // Current ingress
53 logic [ 2 : 0 ] curr_in_2;
54 logic [ 3 : 0 ] curr_in_4;
55 logic [ 3 : 0 ] voq_picked; // is the voq/egress picked?
56 logic no_available_voq; // Is the voqs/egress of the current ingress all
    empty/occupied by other?
57 logic [ 1 : 0
    ] voq_to_pick; // What is the voq_to_pick for the current ingress
58 logic [ 3 : 0 ] busy_egress_mask;
59
60 logic [ 2 : 0 ] i;
61 logic [ 1 : 0 ] busy_port;
62
63 always_comb begin
64     busy_egress_mask = 0; // This is important: busy_egress_mask need a way to start
        with all unoccupied.
65     for (i = 0; i < 4; i = i + 1) begin
66         if (is_busy [ i [ 1 : 0 ] ] == 1'b1) begin
67             busy_egress_mask [ busy_voq_num [ ( i << 1) + : 2 ]
                ] = 1'b1;
68         end
69     end
70     curr_in_2 = {1'b0, curr_ingress_idx} << 1;
71     curr_in_4 = {2'b0, curr_ingress_idx} << 2; // * 4 is << 2
72 end
73
74 initial begin
75     start_ingress_idx = 0;
76     start_voq_num = 0;
77 end
78
79 always_ff @(posedge clk) begin
80
81     if (sched_en) begin
82         // If we begin to schedule
83         // reset sched_sel_en
84         sched_sel_en <= 0;
85         // all the busy ingress ports are automatically assigned.
86         ingress_enable <= 0;
87         // start to assign ports for non-empty

```

```

88     assigning_new <= 1;
89     voq_picked <= busy_egress_mask;
90     curr_ingress_idx <= start_ingress_idx; // Start with start_ingress_idx
91 end else if (assigning_new == 5) begin // alternatively, if we manage to go back
    to start_ingress_idx
92     // If all are assigned, we're going start enabling
93     sched_sel_en <= ingress_enable;
94     // Nex time it should start with another index.
95     start_ingress_idx <= (start_ingress_idx == 3) ? 0 :start_ingress_idx + 1;
96     assigning_new <= 6;
97 end else if (assigning_new == 6) begin
98     sched_sel_en <= 0;
99 end else if (assigning_new >= 1 && assigning_new <= 4) begin
100     curr_ingress_idx <= (curr_ingress_idx == 3) ? 0 :curr_ingress_idx + 1;
101     assigning_new <= assigning_new + 1;
102     if (!is_busy [ curr_ingress_idx ]) begin
103         if (!no_available_voq) begin
104             ingress_enable [ curr_ingress_idx ] <= 1'b1;
105             voq_picked [ voq_to_pick ] <= 1'b1;
106             sched_sel [ curr_in_2 + : 2 ] <= voq_to_pick;
107             start_voq_num [ curr_in_2 + : 2 ] <=
108                 (start_voq_num [ curr_in_2 + : 2 ] == 3) ? 0 :start_voq_num [
                    curr_in_2 + : 2 ] + 1; // Alternatively, we can choose not to
                    move forward when no_available_voq.
109         end
110     end else begin
111         busy_port <= busy_voq_num [ curr_in_2 + : 2 ];
112         ingress_enable [ curr_ingress_idx ] <= 1'b1;
113         sched_sel [ curr_in_2 + : 2 ] <= busy_port;
114         voq_picked [ busy_port ] <= 1'b1;
115     end
116 end
117 end
118
119 // pick_voq will pick return the current ingress's first non empty voq to dequeue
    from.
120 pick_voq pv (
121     .start_voq_num(start_voq_num [ curr_in_2 + : 2 ]),
122     .voq_empty(voq_empty [ curr_in_4 + : 4 ]),
123     .voq_picked(voq_picked),
124     .no_available_voq(no_available_voq),
125     .voq_to_pick(voq_to_pick),
126     .policy(policy),
127     .prio(prio)
128 );
129 endmodule

```

12.1.6 Simple Dual Port

```
1 module simple_dual_port_mem #(
2     parameter MEM_SIZE = 1024, /* How many addresses of memory in total, 1024 by
3         default */
4     parameter DATA_WIDTH = 32 /* How many bit of data per cycle, 32 by default */
5 ) (
6     input logic clk,
7     input logic [ $clog2 ( MEM_SIZE ) - 1 : 0 ] ra, wa, /* Address */
8     input logic [ DATA_WIDTH - 1 : 0 ] d, /* input data */
9     input logic write, /* Write enable */
10    output logic [ DATA_WIDTH - 1 : 0 ] q /* output data */
11 );
12
13 logic [ DATA_WIDTH - 1 : 0 ] mem [ MEM_SIZE - 1 : 0 ];
14
15 always_ff @(posedge clk) begin
16     if (write) begin
17         mem [ wa ] <= d;
18     end
19     q <= mem [ ra ];
20 end
21 endmodule
```

12.1.7 Simple Interface

```
1
2
3 /* verilator lint_off UNUSED */
4 module simple_interface(
5     input logic clk,
6
7     // From sw
8     input logic    reset,
9     input logic    [ 31 : 0 ] writedata,
10    input logic    write,
11    input logic    read,
12    input logic    [  2 : 0 ] address,
13    input logic    chipselect,
14
15    // From egress
16    input logic    [ 31 : 0 ] interface_in_0,
17    input logic    [ 31 : 0 ] interface_in_1,
18    input logic    [ 31 : 0 ] interface_in_2,
19    input logic    [ 31 : 0 ] interface_in_3,
20
21    // To sw
22    output logic   [ 31 : 0 ] readdata,
23
24    // To packet_val/ingress
25    output logic   [  3 : 0 ] interface_out_en,
26    output logic   [ 31 : 0 ] interface_out,
27
28    // Experimenting
29    output logic   experimenting,
30    output logic   simple_reset,
31    // output logic send_only,
32
33    // Special case: Because we're polling but not handling interrupt
34    // we need to acknowledge that this metadata is consumed by the software.
35    // This is the only ack in our program.
36    output logic   [  3 : 0 ] interface_out_ack,
37
38    output logic   sched_policy,
39    output logic   [  7 : 0 ] sched_prio
40 );
41
42 logic [ 31 : 0 ] ctrl;
43 logic prev_read;
44
45 always_comb begin
46     experimenting = (ctrl [ 1 : 0 ] == 2);
47     // send_only = (ctrl == 1);
48     simple_reset = (ctrl [ 1 : 0 ] == 0) || reset;
49     sched_policy = ctrl [ 2 ];
50     sched_prio = ctrl [ 10 : 3 ];
51 end
```

```

52
53 always_ff @(posedge clk) begin
54     prev_read <= read;
55     if (reset) begin
56         ctrl <= 32'h0;
57         readdata <= 32'h0;
58         interface_out_en <= 0;
59         interface_out_ack <= 0;
60         interface_out <= 0;
61     end else begin
62         if (chipselct && write) begin
63             case (address)
64                 3'h0: begin
65                     ctrl <= writedata;
66                 end
67                 3'h1: begin
68                     interface_out_en [ 0 ] <= 1;
69                     interface_out <= writedata;
70                 end
71                 3'h2: begin
72                     interface_out_en [ 1 ] <= 1;
73                     interface_out <= writedata;
74                 end
75                 3'h3: begin
76                     interface_out_en [ 2 ] <= 1;
77                     interface_out <= writedata;
78                 end
79                 3'h4: begin
80                     interface_out_en [ 3 ] <= 1;
81                     interface_out <= writedata;
82                 end
83                 default: begin
84                 end
85             endcase
86         end else begin
87             interface_out_en [ 0 ] <= 0;
88             interface_out_en [ 1 ] <= 0;
89             interface_out_en [ 2 ] <= 0;
90             interface_out_en [ 3 ] <= 0;
91         end
92         if (chipselct && read && prev_read == 0) begin // Need rising edge detection?
93             case (address)
94                 3'h1: begin
95                     readdata <= interface_in_0;
96                     interface_out_ack [ 0 ] <= 1;
97                 end
98                 3'h2: begin
99                     readdata <= interface_in_1;
100                    interface_out_ack [ 1 ] <= 1;
101                end
102                3'h3: begin
103                    readdata <= interface_in_2;
104                    interface_out_ack [ 2 ] <= 1;

```

```
105     end
106     3'h4: begin
107         readdata <= interface_in_3;
108         interface_out_ack [ 3 ] <= 1;
109     end
110     default: begin
111     end
112 endcase
113 end else begin
114     interface_out_ack [ 0 ] <= 0;
115     interface_out_ack [ 1 ] <= 0;
116     interface_out_ack [ 2 ] <= 0;
117     interface_out_ack [ 3 ] <= 0;
118 end
119 end
120 end
121
122 endmodule
```

12.1.8 Simple Switch

```
1
2 module simple_switch
3 (
4     input logic clk,
5     input logic reset,
6     input logic [ 31 : 0 ] writedata,
7     input logic     write,
8     input logic     read,
9     input logic [ 2 : 0 ] address,
10    input logic     chipselect,
11
12    output logic [ 31 : 0 ] readdata
13 );
14 /* verilator lint_off UNUSED */
15 logic [ 3 : 0 ] meta_in_en;
16 logic [ 3 : 0 ] packet_en;
17 logic [ 3 : 0 ] meta_out_ack;
18 logic [ 7 : 0 ] sched_sel;
19 logic [ 3 : 0 ] sched_sel_en;
20
21 logic [ 31 : 0 ] meta_in;
22 // logic [127:0] meta_out;
23 logic [ 31 : 0 ] meta_out [ 4 ];
24 logic [ 127 : 0 ] packet;
25 logic [ 127 : 0 ] packet_out;
26 logic [ 3 : 0 ] packet_out_en;
27 logic experimenting;
28 logic [ 15 : 0 ] empty;
29 logic [ 10 : 0 ] counter;
30 logic [ 3 : 0 ] cycle;
31 logic sched_en;
32 logic simple_reset;
33 logic global_reset;
34 logic sched_policy;
35 logic [ 7 : 0 ] sched_prio;
36
37 assign global_reset = simple_reset || reset;
38
39 always_ff @(posedge clk) begin
40     cycle <= (cycle == 15) ? 0 : cycle + 1;
41     if (experimenting) begin
42         if (cycle == 0) begin
43             counter <= counter + 1;
44             sched_en <= 1;
45         end else if (cycle == 1) begin
46             sched_en <= 0;
47         end
48     end else begin
49         counter <= 0;
50     end
51 end
```



```

52
53 ingress ingress_0 (
54     // Input
55     .clk(clk), .reset(global_reset),
56     .ingress_in_en(meta_in_en [ 0 ]), .experimenting(experimenting),
57     .ingress_in(meta_in),
58     .sched_en(sched_sel_en [ 0 ] && experimenting),
59     .sched_sel(sched_sel [ 1 : 0 ]),
60     .time_stamp(counter),
61
62     // Output
63     // .ingress_out_en(packet_en[0]),
64     .ingress_out(packet [ 31 : 0 ]),
65     .is_empty(empty [ 3 : 0 ])
66 );
67
68 ingress ingress_1 (
69     // Input
70     .clk(clk), .reset(global_reset),
71     .ingress_in_en(meta_in_en [ 1 ]), .experimenting(experimenting),
72     .ingress_in(meta_in),
73     .sched_en(sched_sel_en [ 1 ] && experimenting),
74     .sched_sel(sched_sel [ 3 : 2 ]),
75     .time_stamp(counter),
76
77     // Output
78     // .ingress_out_en(packet_en[1]),
79     .ingress_out(packet [ 63 : 32 ]),
80     .is_empty(empty [ 7 : 4 ])
81 );
82
83 ingress ingress_2 (
84     // Input
85     .clk(clk), .reset(global_reset),
86     .experimenting(experimenting),
87     .ingress_in_en(meta_in_en [ 2 ]),
88     .ingress_in(meta_in),
89     .sched_en(sched_sel_en [ 2 ] && experimenting),
90     .sched_sel(sched_sel [ 5 : 4 ]),
91     .time_stamp(counter),
92
93
94     // Output
95     // .ingress_out_en(packet_en[2]),
96     .ingress_out(packet [ 95 : 64 ]),
97     .is_empty(empty [ 11 : 8 ])
98 );
99
100 ingress ingress_3 (
101     // Input
102     .clk(clk), .reset(global_reset),
103     .ingress_in_en(meta_in_en [ 3 ]), .experimenting(experimenting),
104     .ingress_in(meta_in),

```

```

105     .sched_en(sched_sel_en [ 3 ] && experimenting),
106     .sched_sel(sched_sel [ 7 : 6 ]),
107     .time_stamp(counter),
108
109     // Output
110     // .ingress_out_en(packet_en[3]),
111     .ingress_out(packet [ 127 : 96 ]),
112     .is_empty(empty [ 15 : 12 ])
113 );
114
115 egress egress_0(
116     // Input
117     .clk(clk), .reset(global_reset),
118     .egress_in(packet_out [ 31 : 0 ]),
119     .egress_in_en(packet_out_en [ 0 ]), .egress_in_ack(meta_out_ack [ 0 ]),
120
121     // Output
122     .egress_out(meta_out [ 0 ])
123 );
124 egress egress_1(
125     // Input
126     .clk(clk), .reset(global_reset),
127     .egress_in(packet_out [ 63 : 32 ]),
128     .egress_in_en(packet_out_en [ 1 ]), .egress_in_ack(meta_out_ack [ 1 ]),
129
130     // Output
131     .egress_out(meta_out [ 1 ])
132 );
133 egress egress_2 (
134     // Input
135     .clk(clk), .reset(global_reset),
136     .egress_in(packet_out [ 95 : 64 ]),
137     .egress_in_en(packet_out_en [ 2 ]), .egress_in_ack(meta_out_ack [ 2 ]),
138
139     // Output
140     .egress_out(meta_out [ 2 ])
141 );
142 egress egress_3 (
143     // Input
144     .clk(clk), .reset(global_reset),
145     .egress_in(packet_out [ 127 : 96 ]),
146     .egress_in_en(packet_out_en [ 3 ]), .egress_in_ack(meta_out_ack [ 3 ]),
147
148     // Output
149     .egress_out(meta_out [ 3 ])
150 );
151
152 simple_interface simple_interface (
153     .clk(clk),
154
155     // Input: sw->interface
156     .reset(reset),
157     .writedata(writedata), // sw->hw

```

```

158     .write(write), // sw->hw
159     .read(read), // hw->sw
160     .address(address),
161     .chipselct(chipselct),
162
163     // Input: hw->interface
164     .interface_in_0(meta_out [ 0 ]),
165     .interface_in_1(meta_out [ 1 ]),
166     .interface_in_2(meta_out [ 2 ]),
167     .interface_in_3(meta_out [ 3 ]),
168
169     // Output: interface->sw
170     .readdata(readdata),
171
172     // Output: interface->egress (ack)
173     .interface_out_ack(meta_out_ack), // 4 bit
174
175     // Output: To ingress/packet_val
176     .interface_out_en(meta_in_en), // 4 bit
177     .interface_out(meta_in),
178
179     // Output: Ongoing experiment.
180     .experimenting(experimenting),
181     .simple_reset(simple_reset),
182
183     .sched_policy(sched_policy),
184     .sched_prio(sched_prio)
185
186 );
187
188 crossbar crossbar (
189     .clk(clk),
190     .sched_sel(sched_sel),
191     .crossbar_in_en(sched_sel_en),
192     .crossbar_in(packet),
193     .crossbar_out_en(packet_out_en),
194     .crossbar_out(packet_out)
195 );
196
197 sched scheduler (
198     .clk(clk),
199     .sched_en(sched_en),
200     .is_busy(0),
201     .busy_voq_num(0),
202     .voq_empty(empty),
203     .policy(sched_policy),
204     .prio(sched_prio),
205
206     .sched_sel_en(sched_sel_en), // passed by to ingress, to know which ingress
        should dequeue
207     .sched_sel(sched_sel) // passed by to ingress, to know which voq to dequeue
208 );
209

```


12.1.9 Verilator Makefile

```
1
2 .PHONY: lint
3
4 TOP_FILES = simple_switch.sv simple_interface.sv simple_dual_port_mem.sv crossbar.sv
5           sched.sv
6
7 SVFILES = sched.sv pick_voq.sv vmu.sv simple_dual_port_mem.sv crossbar.sv
8
9 SCHED_FILES = sched.sv pick_voq.sv
10
11 # Run Verilator simulations
12
13 default:
14     @echo "No target given. Try:"
15     @echo ""
16     @echo "make pick_voq"
17     @echo "make crossbar"
18     @echo "make sched.vcd"
19     @echo "make vmu.vcd"
20     @echo "make cmu.vcd"
21     @echo "make lint"
22     @echo "make simple_switch.vcd"
23
24 simple_switch.vcd :obj_dir/Vsimple_switch
25                   obj_dir/Vsimple_switch
26
27 pick_voq :obj_dir/Vpick_voq
28         (obj_dir/Vpick_voq && echo "SUCCESS") || echo "FAILED"
29
30 crossbar: obj_dir/Vcrossbar
31         (obj_dir/Vcrossbar && echo "SUCCESS") || echo "FAILED"
32
33 cmu.vcd :obj_dir/Vcmu
34         obj_dir/Vcmu
35
36 sched.vcd :obj_dir/Vsched
37         obj_dir/Vsched
38
39 vmu.vcd :obj_dir/Vvmu
40         obj_dir/Vvmu
41
42 obj_dir/Vsimple_switch :$(TOP_FILES) verilator/simple_switch.cpp
43                       verilator -trace -Wall -cc $(TOP_FILES) -exe verilator/simple_switch.cpp \
44                             -top-module simple_switch
45                       cd obj_dir && make -j -f Vsimple_switch.mk
46
47 obj_dir/Vcmu :$(CMU_FILES) verilator/cmu.cpp
48               verilator -trace -Wall -cc $(CMU_FILES) -exe verilator/cmu.cpp \
49                     -top-module cmu
50               cd obj_dir && make -j -f Vcmu.mk
```

```

51 obj_dir/Vpick_voq :pick_voq.sv verilator/pick_voq.cpp
52     verilator -Wall -cc pick_voq.sv -exe verilator/pick_voq.cpp \
53         -top-module pick_voq
54     cd obj_dir && make -j -f Vpick_voq.mk
55
56 obj_dir/Vcrossbar :crossbar.sv verilator/crossbar.cpp
57     verilator -Wall -cc crossbar.sv -exe verilator/crossbar.cpp \
58         -top-module crossbar
59     cd obj_dir && make -j -f Vcrossbar.mk
60
61 obj_dir/Vsched :$(SCHED_FILES) verilator/sched.cpp
62     verilator -trace -Wall -cc sched.sv pick_voq.sv -exe verilator/sched.cpp \
63         -top-module sched
64     cd obj_dir && make -j -f Vsched.mk
65
66 obj_dir/Vvmu :vmu.sv simple_dual_port_mem.sv verilator/vmu.cpp
67     verilator -trace -Wall -cc vmu.sv simple_dual_port_mem.sv -exe
68         verilator/vmu.cpp \
69         -top-module vmu
70     cd obj_dir && make -j -f Vvmu.mk
71
72 lint :
73     for file in $(SVFILES); do \
74         verilator --lint-only -Wall $$file; done
75
76 clean :
77     rm -rf obj_dir db incremental_db output_files \
78         lab1.qpf lab1.qsf lab1.sdc lab1.qws c5_pin_model_dump.txt

```

12.1.10 Virtual Memory Unit

```
1
2 /*
3 The virtual output queue management unit:
4 */
5
6 /*
7 voq memory is divided into 16-bit chunks
8 and each of them represents a whole packet
9
10 Inside each chunk:
11 * an 10-bit address, which is the ctrl address of the first segment of the packet
12
13 The same packet in cmu (that needs 2 block) would be:
14 Address 0: 10'b1
15 */
16
17 /*
18 The voq is implemented as a ring buffer. there are #egress voqs in each ingress.
19 Each voq can contain at most 1024 packets. (indexed by the lower 10 bits of the
20 memory)
21 The 2 higher bits represents which voq it is.
22
23     [11, 10]      [9,8,7,6,5,4,3,2,1,0]
24     ----voq_idx----addr of 1st seg of packet-----
25 */
26
27 /* verilator lint_off UNUSED */
28 module vmu #(
29     parameter PACKET_CNT = 1024, /* How many packets can there be in each VOQ, 1024 by
30         default */
31     parameter EGRESS_CNT = 4 /* How many egress there are; which is also how many voqs
32         there are. */
33 ) (
34     input logic clk,
35     input logic reset,
36     input logic voq_enqueue_en,
37     input logic [ $clog2 ( EGRESS_CNT ) -1 : 0 ] voq_enqueue_sel,
38     input logic voq_dequeue_en,
39     input logic [ $clog2 ( EGRESS_CNT ) -1 : 0 ] voq_dequeue_sel,
40     input logic [ 31 : 0
41         ] meta_in, // The address to find the first address of the packet
42     input logic [ 10 : 0 ] time_stamp,
43
44     /* TODO: How many bits for meta_out? */
45     output logic [ 31 : 0 ] meta_out, // The content (first addr of the packet)
46         saved for the dequeue packet
47     output logic [ EGRESS_CNT -1 : 0 ] is_empty, // For scheduler
48     output logic [ EGRESS_CNT -1 : 0
49         ] is_full // For potential packet drop. If is_full, then drop the current
50 );
```

```

46 logic [ $clog2 ( PACKET_CNT ) +1 : 0 ] start_idx [ 3 : 0
      ]; // first element
47 logic [ $clog2 ( PACKET_CNT ) +1 : 0 ] end_idx [ 3 : 0
      ]; // one pass the last element
48
49 logic [ $clog2 ( EGRESS_CNT ) : 0 ] i;
50 logic [ $clog2 ( EGRESS_CNT ) -1 : 0 ] _i;
51
52 always_comb begin
53     for (i = 0; i < EGRESS_CNT; i = i + 1) begin
54         _i = i [ $clog2 ( EGRESS_CNT ) -1 : 0 ];
55         is_empty [ _i ] = (start_idx [ _i ] == end_idx [ _i ]);
56         is_full [ _i ] = (start_idx [ _i ] == ((end_idx [ _i
              ] == PACKET_CNT - 1) ? 0 : end_idx [ _i ] + 1));
57     end
58 end
59
60
61 always @(posedge clk) begin
62     if (reset) begin
63         start_idx [ 0 ] <= 0;
64         start_idx [ 1 ] <= 0;
65         start_idx [ 0 ] <= 0;
66         start_idx [ 1 ] <= 0;
67         end_idx [ 0 ] <= 0;
68         end_idx [ 1 ] <= 0;
69         end_idx [ 2 ] <= 0;
70         end_idx [ 3 ] <= 0;
71     end else begin
72         if (voq_enqueue_en && !is_full [ voq_enqueue_sel ]) begin
73             end_idx [ voq_enqueue_sel ] <= (end_idx [ voq_enqueue_sel
                    ] != PACKET_CNT - 1) ? end_idx [ voq_enqueue_sel ] + 1 : 0;
74         end
75
76         if (voq_dequeue_en && !is_empty [ voq_dequeue_sel ]) begin
77             start_idx [ voq_dequeue_sel ] <= (start_idx [ voq_dequeue_sel
                    ] != PACKET_CNT - 1) ? start_idx [ voq_dequeue_sel ] + 1 : 0;
78         end
79     end
80 end
81
82 simple_dual_port_mem #(.MEM_SIZE(PACKET_CNT * EGRESS_CNT), .DATA_WIDTH(32))
      vmem
83 (
84     .clk(clk),
85     .ra(start_idx [ voq_dequeue_sel ]), .wa(end_idx [ voq_enqueue_sel
          ]),
86     .d({meta_in [ 31 : 22 ], time_stamp, meta_in [ 10 : 0
          ]}), .q(meta_out),
87     .write(voq_enqueue_en)
88 );
89
90 endmodule

```


12.2 Simple Switch: Software

12.2.1 Even Load

```
1  #include <stdio.h>
2  #include "simpleSwitch.h"
3  #include <sys/ioctl.h>
4  #include <sys/types.h>
5  #include <sys/stat.h>
6  #include <fcntl.h>
7  #include <string.h>
8  #include <unistd.h>
9
10 #define DEVICE_PATH "/dev/simple_driver"
11 int simple_switch_fd;
12 int num_sent = 0;
13 int num_get = 0;
14
15 void open_simple_device()
16 {
17     simple_switch_fd = open(DEVICE_PATH, O_RDWR);
18     if (simple_switch_fd < 0) {
19         perror("Failed to open simple device");
20         return;
21     }
22 }
23
24 void print_packet(void *packet_data)
25 {
26     unsigned int packet = *((unsigned int*) packet_data);
27
28     printf("\tmetadata: [%u | %u | %u | %u | %u ]\n",
29         (packet >> 30) & 0x3, // Extract bits 31:30
30         (packet >> 28) & 0x3, // Extract bits 29:28
31         (packet >> 22) & 0x3F, // Extract bits 27:22
32         (packet >> 11) & 0x7FF, // Extract bits 21:11
33         packet & 0x7FF // Extract bits 10:0
34     );
35 }
36
37 int extra_time_delta(unsigned int packet)
38 {
39     return (packet & 0x7FF) - ((packet >> 11) & 0x7FF);
40 }
41
42 int extra_dst_port(unsigned int packet)
43 {
44     return (packet >> 28) & 0x3;
45 }
46
47 void set_ctrl_register(const packet_ctrl_t *pkt_ctrl)
48 {
49     if (ioctl(simple_switch_fd, SIMPLE_WRITE_CTRL, pkt_ctrl) < 0) {
```

```

50     perror("ioctl(SIMPLE_WRITE_CTRL) set CTRL failed\n");
51     close(simple_switch_fd);
52     return;
53 }
54 }
55
56 void send_packet(const packet_meta_t *pkt_meta)
57 {
58     usleep(5000);
59     if (ioctl(simple_switch_fd, SIMPLE_WRITE_PACKET, pkt_meta) < 0) {
60         perror("Failed to send packet");
61         return;
62     }
63     num_sent++;
64 }
65
66
67 /**
68  * Sets the length and time_delta fields of the packet metadata.
69  * Assumes that @pkt_meta is cleared prior to calling this function.
70  * @param pkt_meta Pointer to the packet metadata.
71  * @param length Length of the packet, to be set in bits [27:22].
72  */
73 void set_packet_length(packet_meta_t *pkt_meta, unsigned int length)
74 {
75     if (length > 0x3F) {
76         perror("Failed to set length\n");
77         return;
78     }
79     // Mask and shift length @ bit pos [27:22]
80     *pkt_meta |= (length & 0x3F) << 22;
81 }
82
83
84 void set_all_packet_fields(packet_meta_t *pkt_meta, unsigned int dst,
85                             unsigned int src,
86                             unsigned int length)
87 {
88     packet_meta_t pkt_tmp;
89     *pkt_meta = 0;
90
91     set_packet_length(pkt_meta, length);
92     *pkt_meta = set_dst_port(*pkt_meta, dst);
93     *pkt_meta = set_src_port(*pkt_meta, src);
94 }
95
96 int main()
97 {
98     packet_meta_t pkt_meta, rcvd_pkt_meta;
99     packet_ctrl_t pkt_ctrl;
100     int total_latency = 0;
101

```

```

102 open_simple_device();
103
104     printf("Start: \n");
105 pkt_ctrl = 0;
106     set_ctrl_register(&pkt_ctrl);
107     print_packet(&pkt_ctrl);
108     usleep(1000);
109
110 // Set control register (CTRL=1)
111     printf("Start sending\n");
112 pkt_ctrl = 1;
113     set_ctrl_register(&pkt_ctrl);
114     print_packet(&pkt_ctrl);
115     usleep(1000);
116
117 for (int i = 0; i < 32; i++) {
118     set_all_packet_fields(&pkt_meta, (i+1)%4, i%4, 1);
119     send_packet(&pkt_meta);
120     print_packet(&pkt_meta);
121 }
122
123 usleep(10000);
124 printf("Start recving\n");
125     pkt_ctrl = 2;
126     set_ctrl_register(&pkt_ctrl);
127     print_packet(&pkt_ctrl);
128     usleep(10000);
129
130     printf("Requested %d packets\n", num_sent);
131     for (int i = 0; i < 200; i++) {
132         usleep(1000);
133         if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_0, &rcvd_pkt_meta) < 0) {
134             perror("ioctl read packet failed");
135             close(simple_switch_fd);
136             return -1;
137         }
138         if (rcvd_pkt_meta >> 22) {
139             printf("Port 0: \n");
140             print_packet(&rcvd_pkt_meta);
141             total_latency += extra_time_delta(rcvd_pkt_meta);
142             num_get++;
143         }
144         usleep(1000);
145         if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_1, &rcvd_pkt_meta) < 0) {
146             perror("ioctl read packet failed");
147             close(simple_switch_fd);
148             return -1;
149         }
150         if (rcvd_pkt_meta >> 22) {
151             printf("Port 1: \n");
152             print_packet(&rcvd_pkt_meta);
153             total_latency += extra_time_delta(rcvd_pkt_meta);
154             num_get++;

```

```

155     }
156     usleep(1000);
157     if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_2, &rcvd_pkt_meta) < 0) {
158         perror("ioctl read packet failed");
159         close(simple_switch_fd);
160         return -1;
161     }
162     if (rcvd_pkt_meta >> 22) {
163         printf("Port 2: \n");
164         print_packet(&rcvd_pkt_meta);
165         total_latency += extra_time_delta(rcvd_pkt_meta);
166         num_get++;
167     }
168     usleep(1000);
169     if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_3, &rcvd_pkt_meta) < 0) {
170         perror("ioctl read packet failed");
171         close(simple_switch_fd);
172         return -1;
173     }
174     if (rcvd_pkt_meta >> 22) {
175         printf("Port 3: \n");
176         print_packet(&rcvd_pkt_meta);
177         total_latency += extra_time_delta(rcvd_pkt_meta);
178         num_get++;
179     }
180 }
181 printf("Got %d packets\n", num_get);
182 printf("Total latency: %d\n", total_latency);
183
184 close(simple_switch_fd);
185 printf("Userspace program terminating\n");
186 return 0;
187 }

```

12.2.2 Even Load (w/ Priorities)

```
1  #include <stdio.h>
2  #include "simpleSwitch.h"
3  #include <sys/ioctl.h>
4  #include <sys/types.h>
5  #include <sys/stat.h>
6  #include <fcntl.h>
7  #include <string.h>
8  #include <unistd.h>
9
10 #define DEVICE_PATH "/dev/simple_driver"
11 int simple_switch_fd;
12
13 int num_sent = 0;
14 int num_get = 0;
15
16 void open_simple_device()
17 {
18     simple_switch_fd = open(DEVICE_PATH, O_RDWR);
19     if (simple_switch_fd < 0) {
20         perror("Failed to open simple device");
21         return;
22     }
23 }
24
25 void print_packet(void *packet_data)
26 {
27     unsigned int packet = *((unsigned int*) packet_data);
28
29     printf("\tmetadata: [%u | %u | %u | %u | %u ]\n",
30         (packet >> 30) & 0x3, // Extract bits 31:30
31         (packet >> 28) & 0x3, // Extract bits 29:28
32         (packet >> 22) & 0x3F, // Extract bits 27:22
33         (packet >> 11) & 0x7FF, // Extract bits 21:11
34         packet & 0x7FF // Extract bits 10:0
35     );
36 }
37
38 int extra_time_delta(unsigned int packet)
39 {
40     return (packet & 0x7FF) - ((packet >> 11) & 0x7FF);
41 }
42
43 int extra_dst_port(unsigned int packet)
44 {
45     return (packet >> 28) & 0x3;
46 }
47
48 void set_ctrl_register(const packet_ctrl_t *pkt_ctrl)
49 {
50     if (ioctl(simple_switch_fd, SIMPLE_WRITE_CTRL, pkt_ctrl) < 0) {
51         perror("ioctl(SIMPLE_WRITE_CTRL) set CTRL failed\n");
```

```

52     close(simple_switch_fd);
53     return;
54 }
55 }
56
57 void send_packet(const packet_meta_t *pkt_meta)
58 {
59     usleep(5000);
60     if (ioctl(simple_switch_fd, SIMPLE_WRITE_PACKET, pkt_meta) < 0) {
61         perror("Failed to send packet");
62         return;
63     }
64     num_sent++;
65 }
66
67 void set_packet_length(packet_meta_t *pkt_meta, unsigned int length)
68 {
69     if (length > 0x3F) {
70         perror("Failed to set length\n");
71         return;
72     }
73     // Mask and shift length @ bit pos [27:22]
74     *pkt_meta |= (length & 0x3F) << 22;
75 }
76
77
78 void set_all_packet_fields(packet_meta_t *pkt_meta, unsigned int dst,
79                             unsigned int src,
80                             unsigned int length)
81 {
82     packet_meta_t pkt_tmp;
83     *pkt_meta = 0;
84
85     set_packet_length(pkt_meta, length);
86     *pkt_meta = set_dst_port(*pkt_meta, dst);
87     *pkt_meta = set_src_port(*pkt_meta, src);
88 }
89
90 int main()
91 {
92     packet_meta_t pkt_meta, rcvd_pkt_meta;
93     packet_ctrl_t pkt_ctrl;
94     int total_latency = 0;
95
96     open_simple_device();
97
98     printf("Start: \n");
99     pkt_ctrl = 0;
100     set_ctrl_register(&pkt_ctrl);
101     print_packet(&pkt_ctrl);
102     usleep(1000);
103

```

```

104 // Set control register (CTRL=1)
105     printf("Start sending\n");
106     pkt_ctrl = 1;
107     set_ctrl_register(&pkt_ctrl);
108     print_packet(&pkt_ctrl);
109     usleep(1000);
110
111     for (int i = 0; i < 32; i++) {
112         set_all_packet_fields(&pkt_meta, (i+1)%4, i%4, 1);
113         send_packet(&pkt_meta);
114         print_packet(&pkt_meta);
115     }
116
117     usleep(10000);
118     printf("Start recving\n");
119     pkt_ctrl = 222;
120     set_ctrl_register(&pkt_ctrl);
121     print_packet(&pkt_ctrl);
122     usleep(10000);
123
124     printf("Requested %d packets\n", num_sent);
125     for (int i = 0; i < 200; i++) {
126         usleep(1000);
127         if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_0, &rcvd_pkt_meta) < 0) {
128             perror("ioctl read packet failed");
129             close(simple_switch_fd);
130             return -1;
131         }
132         if (rcvd_pkt_meta >> 22) {
133             printf("Port 0: \n");
134             print_packet(&rcvd_pkt_meta);
135             total_latency += extra_time_delta(rcvd_pkt_meta);
136             num_get++;
137         }
138         usleep(1000);
139         if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_1, &rcvd_pkt_meta) < 0) {
140             perror("ioctl read packet failed");
141             close(simple_switch_fd);
142             return -1;
143         }
144         if (rcvd_pkt_meta >> 22) {
145             printf("Port 1: \n");
146             print_packet(&rcvd_pkt_meta);
147             total_latency += extra_time_delta(rcvd_pkt_meta);
148             num_get++;
149         }
150         usleep(1000);
151         if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_2, &rcvd_pkt_meta) < 0) {
152             perror("ioctl read packet failed");
153             close(simple_switch_fd);
154             return -1;
155         }
156         if (rcvd_pkt_meta >> 22) {

```

```

157     printf("Port 2: \n");
158     print_packet(&rcvd_pkt_meta);
159     total_latency += extra_time_delta(rcvd_pkt_meta);
160     num_get++;
161 }
162 usleep(1000);
163 if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_3, &rcvd_pkt_meta) < 0) {
164     perror("ioctl read packet failed");
165     close(simple_switch_fd);
166     return -1;
167 }
168 if (rcvd_pkt_meta >> 22) {
169     printf("Port 3: \n");
170     print_packet(&rcvd_pkt_meta);
171     total_latency += extra_time_delta(rcvd_pkt_meta);
172     num_get++;
173 }
174 }
175 printf("Got %d packets\n", num_get);
176 printf("Total latency: %d\n", total_latency);
177
178 close(simple_switch_fd);
179 printf("Userspace program terminating\n");
180 return 0;
181 }

```


12.2.3 Simple Switch

```
1
2 /*
3  * TODO:
4  * [ ] write to CTRL register (CTRL=1)
5  *   inputting meta data
6  * [ ] start w/ single threaded test prog
7  * [ ] send 2 packets (call 2 ioctl w/ write + data flags & other combinations)
8  * [ ] try setting different parameters using ioctl (i.e., src, dest, len)
9  * [ ] len=1 or 2
10 * [ ] ioctl_write (set CTRL=2) for both read & write
11 * [ ] ioctl_read should return the 2 packets
12 *   (1) inf loop to read packets: can i write at the same time???
13 *   OR
14 *   (2) ioctl w/ different flags
15 *   (3)
16 *   [ ]
17 * [ ] later: create separate thread for polling
18 *
19 * ioctl_write w/ write flag (1) and data and actual packet metadata that we
20 *   assemble here
21 * all port 0 (src + dest)
22 */
23 #include <stdio.h>
24 #include "simpleSwitch.h"
25 #include <sys/ioctl.h>
26 #include <sys/types.h>
27 #include <sys/stat.h>
28 #include <fcntl.h>
29 #include <string.h>
30 #include <unistd.h>
31
32 #define MIN_PORT_NUM 0
33 #define MAX_PORT_NUM 3
34 #define DEVICE_PATH "/dev/simple_driver"
35
36
37 int simple_switch_fd;
38
39 // High [2 bit src] [2 bit dst] [6 bits for the # of 32 byte chunks] [22 bit time
40 // stamp] Low
41 // typedef unsigned int packet_meta_t;
42 // typedef unsigned int packet_ctrl_t;
43 // packet_info_t;
44
45 void open_simple_device()
46 {
47     simple_switch_fd = open(DEVICE_PATH, O_RDWR);
48     if (simple_switch_fd < 0) {
49         perror("Failed to open simple device");
50         return;
51     }
52 }
```

```

50     }
51 }
52
53 void print_packet_no_hw(void *packet_data)
54 {
55     unsigned int packet = *((unsigned int*) packet_data);
56
57     printf("\tmetadata (0x%X): [%u | %u | %u | %u]\n",
58           packet,
59           (packet >> 30) & 0x3, // Extract bits 31:30
60           (packet >> 28) & 0x3, // Extract bits 29:28
61           (packet >> 22) & 0x3F, // Extract bits 27:22
62           packet & 0x3FFFFFF // Extract bits 21:0
63     );
64 }
65
66 void print_packet(void *packet_data)
67 {
68     unsigned int packet = *((unsigned int*) packet_data);
69     if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_0, packet_data)) {
70         perror("ioctl(SIMPLE_READ_PACKET_0) failed");
71         return;
72     }
73
74     printf("\tmetadata: [%u | %u | %u | %u]\n",
75           (packet >> 30) & 0x3, // Extract bits 31:30
76           (packet >> 28) & 0x3, // Extract bits 29:28
77           (packet >> 22) & 0x3F, // Extract bits 27:22
78           packet & 0x3FFFFFF // Extract bits 21:0
79     );
80 }
81
82 void set_ctrl_register(const packet_ctrl_t *pkt_ctrl)
83 {
84     if (ioctl(simple_switch_fd, SIMPLE_WRITE_CTRL, pkt_ctrl) < 0) {
85         perror("ioctl(SIMPLE_WRITE_CTRL) set CTRL=1 failed\n");
86         close(simple_switch_fd);
87         return;
88     }
89 }
90
91 void send_packet(const packet_meta_t *pkt_meta)
92 {
93     sleep(0.5);
94     if (ioctl(simple_switch_fd, SIMPLE_WRITE_PACKET, pkt_meta) < 0) {
95         perror("Failed to send packet");
96         return;
97     }
98 }
99
100 // void receive_packet(int packet_id, packet_meta_t *pkt_meta)
101 // {
102 //     if (ioctl(simple_switch_fd, cmd, pkt_meta) < 0) {

```

```

103 //      perror("Failed to receive packet");
104 //      return;
105 //  }
106 // }
107
108 /**
109  * Sets the length and time_delta fields of the packet metadata.
110  * Assumes that @pkt_meta is cleared prior to calling this function.
111  * @param pkt_meta Pointer to the packet metadata.
112  * @param length Length of the packet, to be set in bits [27:22].
113  */
114 void set_packet_length(packet_meta_t *pkt_meta, unsigned int length)
115 {
116     if (length > 0x3F) {
117         perror("Failed to set length\n");
118         return;
119     }
120     // Mask and shift length @ bit pos [27:22]
121     *pkt_meta |= (length & 0x3F) << 22;
122 }
123
124
125 void set_all_packet_fields(packet_meta_t *pkt_meta, unsigned int dst,
126                           unsigned int src,
127                           unsigned int length)
128 {
129     packet_meta_t pkt_tmp;
130     *pkt_meta = 0;
131
132     set_packet_length(pkt_meta, length);
133     *pkt_meta = set_dst_port(*pkt_meta, dst);
134     *pkt_meta = set_src_port(*pkt_meta, src);
135 }
136
137 int main()
138 {
139     int write_num_packets = 2, num_sent = 0;
140     unsigned int dest = 0, src = 0, len = 1, t_delta = 10;
141     packet_meta_t pkt_meta, rcvd_pkt_meta;
142     packet_ctrl_t pkt_ctrl;
143
144     open_simple_device();
145
146     printf("Set CTRL register to 0\n");
147     pkt_ctrl = 0;
148     set_ctrl_register(&pkt_ctrl);
149     print_packet_no_hw(&pkt_ctrl);
150
151     // Set control register (CTRL=1)
152     printf("Set CTRL register to 1\n");
153     pkt_ctrl = 1;
154     set_ctrl_register(&pkt_ctrl);

```

```

155     print_packet_no_hw(&pkt_ctrl);
156
157     for (int i = 0; i < 10; i++) {
158         set_all_packet_fields(&pkt_meta, i%2, i%4, 4);
159         send_packet(&pkt_meta);
160         print_packet_no_hw(&pkt_meta);
161     }
162     num_sent += write_num_packets;
163
164     len = 2;
165
166     set_all_packet_fields(&pkt_meta, 0, 4, 4);
167     for (int i = 0; i < 10; i++)
168         set_all_packet_fields(&pkt_meta, i%4, (i+1)%4, 4);
169     send_packet(&pkt_meta);
170     print_packet_no_hw(&pkt_meta);
171     num_sent += write_num_packets;
172
173     printf("Set CTRL register to 2\n");
174     pkt_ctrl = 2;
175     set_ctrl_register(&pkt_ctrl);
176     print_packet_no_hw(&pkt_ctrl);
177
178     printf("Requested %d packets\n", num_sent);
179     for (int i = 0; i < 5; i++) {
180         if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_0, &rcvd_pkt_meta) < 0) {
181             perror("ioctl read packet failed");
182             close(simple_switch_fd);
183             return -1;
184         }
185         sleep(0.5);
186         printf("Port 0\n");
187         print_packet_no_hw(&rcvd_pkt_meta);
188         if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_1, &rcvd_pkt_meta) < 0) {
189             perror("ioctl read packet failed");
190             close(simple_switch_fd);
191             return -1;
192         }
193         sleep(0.5);
194         printf("Port 1\n");
195         print_packet_no_hw(&rcvd_pkt_meta);
196         if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_2, &rcvd_pkt_meta) < 0) {
197             perror("ioctl read packet failed");
198             close(simple_switch_fd);
199             return -1;
200         }
201         sleep(0.5);
202         printf("Port 2\n");
203         print_packet_no_hw(&rcvd_pkt_meta);
204         if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_3, &rcvd_pkt_meta) < 0) {
205             perror("ioctl read packet failed");
206             close(simple_switch_fd);
207             return -1;

```

```

208     }
209     sleep(0.5);
210     printf("Port 3\n");
211     print_packet_no_hw(&rcvd_pkt_meta);
212 }
213
214 close(simple_switch_fd);
215 printf("Userspace program terminating\n");
216 return 0;
217 }

```

```

1  #ifndef _simpleSwitch_H
2  #define _simpleSwitch_H
3  #include "driver/simple_driver.h"
4
5  packet_meta_t set_src_port(packet_meta_t meta, unsigned int port) {
6      if (port > 3) {
7          printf("Ports number (%u) should be btw 0 and 3. \n", port);
8          return meta;
9      }
10     meta &= ~(0x3 << 30);
11
12     switch(port) {
13         case 0:
14             break;
15         case 1:
16             meta |= (0x1 << 30);
17             break;
18         case 2:
19             meta |= (0x2 << 30);
20             break;
21         case 3:
22             meta |= (0x3 << 30);
23             break;
24         default:
25             break;
26     }
27     return meta;
28 }
29
30 packet_meta_t set_dst_port(packet_meta_t meta, unsigned int port) {
31     if (port > 3) {
32         printf("Ports number (%u) should be btw 0 and 3. \n", port);
33         return meta;
34     }
35     // Clear the bits 28 and 29
36     meta &= ~(0x3 << 28);
37
38     switch(port) {
39         case 0: // 00
40             break; // No action needed as bits 28 and 29 are already cleared
41         case 1: // 01

```

```
42     meta |= (0x1 << 28); // Set bit 28
43     break;
44     case 2: // 10
45         meta |= (0x2 << 28); // Set bit 29
46         break;
47     case 3: // 11
48         meta |= (0x3 << 28); // Set both bits 28 and 29
49         break;
50     default:
51         break;
52 }
53
54     return meta;
55 }
56
57 #endif
```

12.2.4 Skewed Load

```
1  #include <stdio.h>
2  #include "simpleSwitch.h"
3  #include <sys/ioctl.h>
4  #include <sys/types.h>
5  #include <sys/stat.h>
6  #include <fcntl.h>
7  #include <string.h>
8  #include <unistd.h>
9
10 #define DEVICE_PATH "/dev/simple_driver"
11 int simple_switch_fd;
12
13 int num_sent = 0;
14 int num_get = 0;
15
16 void open_simple_device()
17 {
18     simple_switch_fd = open(DEVICE_PATH, O_RDWR);
19     if (simple_switch_fd < 0) {
20         perror("Failed to open simple device");
21         return;
22     }
23 }
24
25 void print_packet(void *packet_data)
26 {
27     unsigned int packet = *((unsigned int*) packet_data);
28
29     printf("\tmetadata: [%u | %u | %u | %u | %u ]\n",
30         (packet >> 30) & 0x3, // Extract bits 31:30
31         (packet >> 28) & 0x3, // Extract bits 29:28
32         (packet >> 22) & 0x3F, // Extract bits 27:22
33         (packet >> 11) & 0x7FF, // Extract bits 21:11
34         packet & 0x7FF // Extract bits 10:0
35     );
36 }
37
38 int extra_time_delta(unsigned int packet)
39 {
40     return (packet & 0x7FF) - ((packet >> 11) & 0x7FF);
41 }
42
43 int extra_dst_port(unsigned int packet)
44 {
45     return (packet >> 28) & 0x3;
46 }
47
48 void set_ctrl_register(const packet_ctrl_t *pkt_ctrl)
49 {
50     if (ioctl(simple_switch_fd, SIMPLE_WRITE_CTRL, pkt_ctrl) < 0) {
51         perror("ioctl(SIMPLE_WRITE_CTRL) set CTRL failed\n");
```

```

52     close(simple_switch_fd);
53     return;
54 }
55 }
56
57 void send_packet(const packet_meta_t *pkt_meta)
58 {
59     usleep(5000);
60     if (ioctl(simple_switch_fd, SIMPLE_WRITE_PACKET, pkt_meta) < 0) {
61         perror("Failed to send packet");
62         return;
63     }
64     num_sent++;
65 }
66
67
68 /**
69  * Sets the length and time_delta fields of the packet metadata.
70  * Assumes that @pkt_meta is cleared prior to calling this function.
71  * @param pkt_meta Pointer to the packet metadata.
72  * @param length Length of the packet, to be set in bits [27:22].
73  */
74 void set_packet_length(packet_meta_t *pkt_meta, unsigned int length)
75 {
76     if (length > 0x3F) {
77         perror("Failed to set length\n");
78         return;
79     }
80     // Mask and shift length @ bit pos [27:22]
81     *pkt_meta |= (length & 0x3F) << 22;
82 }
83
84
85 void set_all_packet_fields(packet_meta_t *pkt_meta, unsigned int dst, unsigned int
86 src, unsigned int length)
87 {
88     packet_meta_t pkt_tmp;
89     *pkt_meta = 0;
90
91     set_packet_length(pkt_meta, length);
92     *pkt_meta = set_dst_port(*pkt_meta, dst);
93     *pkt_meta = set_src_port(*pkt_meta, src);
94 }
95
96 int main()
97 {
98     packet_meta_t pkt_meta, rcvd_pkt_meta;
99     packet_ctrl_t pkt_ctrl;
100     int total_latency = 0;
101
102     open_simple_device();
103

```



```

104     printf("Start: \n");
105     pkt_ctrl = 0;
106     set_ctrl_register(&pkt_ctrl);
107     print_packet(&pkt_ctrl);
108     usleep(1000);
109
110     // Set control register (CTRL=1)
111     printf("Start sending\n");
112     pkt_ctrl = 1;
113     set_ctrl_register(&pkt_ctrl);
114     print_packet(&pkt_ctrl);
115     usleep(1000);
116
117     for (int i = 0; i < 16; i++) {
118         set_all_packet_fields(&pkt_meta, (i+1)%2, i%4, 1);
119         send_packet(&pkt_meta);
120         print_packet(&pkt_meta);
121     }
122
123     for (int i = 0; i < 16; i++) {
124         set_all_packet_fields(&pkt_meta, (i+3)%4, (i+1)%4, 1);
125         send_packet(&pkt_meta);
126         print_packet(&pkt_meta);
127     }
128
129     usleep(10000);
130     printf("Start recving\n");
131     pkt_ctrl = 2;
132     set_ctrl_register(&pkt_ctrl);
133     print_packet(&pkt_ctrl);
134     usleep(10000);
135
136     printf("Requested %d packets\n", num_sent);
137     for (int i = 0; i < 1000; i++) {
138         usleep(1000);
139         if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_0, &rcvd_pkt_meta) < 0) {
140             perror("ioctl read packet failed");
141             close(simple_switch_fd);
142             return -1;
143         }
144         if (rcvd_pkt_meta >> 22) {
145             printf("Port 0: \n");
146             print_packet(&rcvd_pkt_meta);
147             total_latency += extra_time_delta(rcvd_pkt_meta);
148             num_get++;
149         }
150         usleep(1000);
151         if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_1, &rcvd_pkt_meta) < 0) {
152             perror("ioctl read packet failed");
153             close(simple_switch_fd);
154             return -1;
155         }
156         if (rcvd_pkt_meta >> 22) {

```

```

157     printf("Port 1: \n");
158     print_packet(&rcvd_pkt_meta);
159     total_latency += extra_time_delta(rcvd_pkt_meta);
160     num_get++;
161 }
162 usleep(1000);
163 if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_2, &rcvd_pkt_meta) < 0) {
164     perror("ioctl read packet failed");
165     close(simple_switch_fd);
166     return -1;
167 }
168 if (rcvd_pkt_meta >> 22) {
169     printf("Port 2: \n");
170     print_packet(&rcvd_pkt_meta);
171     total_latency += extra_time_delta(rcvd_pkt_meta);
172     num_get++;
173 }
174 usleep(1000);
175 if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_3, &rcvd_pkt_meta) < 0) {
176     perror("ioctl read packet failed");
177     close(simple_switch_fd);
178     return -1;
179 }
180 if (rcvd_pkt_meta >> 22) {
181     printf("Port 3: \n");
182     print_packet(&rcvd_pkt_meta);
183     total_latency += extra_time_delta(rcvd_pkt_meta);
184     num_get++;
185 }
186 }
187 printf("Got %d packets\n", num_get);
188 printf("Total latency: %d\n", total_latency);
189
190 close(simple_switch_fd);
191 printf("Userspace program terminating\n");
192 return 0;
193 }

```

12.2.5 Skewed Load (w/ Priorities)

```
1
2 #include <stdio.h>
3 #include "simpleSwitch.h"
4 #include <sys/ioctl.h>
5 #include <sys/types.h>
6 #include <sys/stat.h>
7 #include <fcntl.h>
8 #include <string.h>
9 #include <unistd.h>
10
11 #define DEVICE_PATH "/dev/simple_driver"
12 int simple_switch_fd;
13
14 int num_sent = 0;
15 int num_get = 0;
16
17 void open_simple_device()
18 {
19     simple_switch_fd = open(DEVICE_PATH, O_RDWR);
20     if (simple_switch_fd < 0) {
21         perror("Failed to open simple device");
22         return;
23     }
24 }
25
26 void print_packet(void *packet_data)
27 {
28     unsigned int packet = *((unsigned int*) packet_data);
29
30     printf("\tmetadata: [%u | %u | %u | %u | %u ]\n",
31         (packet >> 30) & 0x3, // Extract bits 31:30
32         (packet >> 28) & 0x3, // Extract bits 29:28
33         (packet >> 22) & 0x3F, // Extract bits 27:22
34         (packet >> 11) & 0x7FF, // Extract bits 21:11
35         packet & 0x7FF // Extract bits 10:0
36     );
37 }
38
39 int extra_time_delta(unsigned int packet)
40 {
41     return (packet & 0x7FF) - ((packet >> 11) & 0x7FF);
42 }
43
44 int extra_dst_port(unsigned int packet)
45 {
46     return (packet >> 28) & 0x3;
47 }
48
49 void set_ctrl_register(const packet_ctrl_t *pkt_ctrl)
50 {
51     if (ioctl(simple_switch_fd, SIMPLE_WRITE_CTRL, pkt_ctrl) < 0) {
```

```

52     perror("ioctl(SIMPLE_WRITE_CTRL) set CTRL failed\n");
53     close(simple_switch_fd);
54     return;
55 }
56 }
57
58 void send_packet(const packet_meta_t *pkt_meta)
59 {
60     usleep(5000);
61     if (ioctl(simple_switch_fd, SIMPLE_WRITE_PACKET, pkt_meta) < 0) {
62         perror("Failed to send packet");
63         return;
64     }
65     num_sent++;
66 }
67
68
69 void set_packet_length(packet_meta_t *pkt_meta, unsigned int length)
70 {
71     if (length > 0x3F) {
72         perror("Failed to set length\n");
73         return;
74     }
75     // Mask and shift length @ bit pos [27:22]
76     *pkt_meta |= (length & 0x3F) << 22;
77 }
78
79
80 void set_all_packet_fields(packet_meta_t *pkt_meta, unsigned int dst,
81                           unsigned int src,
82                           unsigned int length)
83 {
84     packet_meta_t pkt_tmp;
85     *pkt_meta = 0;
86
87     set_packet_length(pkt_meta, length);
88     *pkt_meta = set_dst_port(*pkt_meta, dst);
89     *pkt_meta = set_src_port(*pkt_meta, src);
90 }
91
92 int main()
93 {
94     packet_meta_t pkt_meta, rcvd_pkt_meta;
95     packet_ctrl_t pkt_ctrl;
96     int total_latency = 0;
97
98     open_simple_device();
99
100     printf("Start: \n");
101     pkt_ctrl = 0;
102     set_ctrl_register(&pkt_ctrl);
103     print_packet(&pkt_ctrl);

```

```

104     usleep(1000);
105
106     // Set control register (CTRL=1)
107     printf("Start sending\n");
108     pkt_ctrl = 1;
109     set_ctrl_register(&pkt_ctrl);
110     print_packet(&pkt_ctrl);
111     usleep(1000);
112
113     for (int i = 0; i < 16; i++) {
114         set_all_packet_fields(&pkt_meta, (i+1)%2, i%4, 1);
115         send_packet(&pkt_meta);
116         print_packet(&pkt_meta);
117     }
118
119     for (int i = 0; i < 16; i++) {
120         set_all_packet_fields(&pkt_meta, (i+3)%4, (i+1)%4, 1);
121         send_packet(&pkt_meta);
122         print_packet(&pkt_meta);
123     }
124
125     usleep(10000);
126     printf("Start recving\n");
127     pkt_ctrl = 222;
128     set_ctrl_register(&pkt_ctrl);
129     print_packet(&pkt_ctrl);
130     usleep(10000);
131
132     printf("Requested %d packets\n", num_sent);
133     for (int i = 0; i < 200; i++) {
134         usleep(1000);
135         if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_0, &rcvd_pkt_meta) < 0) {
136             perror("ioctl read packet failed");
137             close(simple_switch_fd);
138             return -1;
139         }
140         if (rcvd_pkt_meta >> 22) {
141             printf("Port 0: \n");
142             print_packet(&rcvd_pkt_meta);
143             total_latency += extra_time_delta(rcvd_pkt_meta);
144             num_get++;
145         }
146         usleep(1000);
147         if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_1, &rcvd_pkt_meta) < 0) {
148             perror("ioctl read packet failed");
149             close(simple_switch_fd);
150             return -1;
151         }
152         if (rcvd_pkt_meta >> 22) {
153             printf("Port 1: \n");
154             print_packet(&rcvd_pkt_meta);
155             total_latency += extra_time_delta(rcvd_pkt_meta);
156             num_get++;

```

```

157     }
158     usleep(1000);
159     if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_2, &rcvd_pkt_meta) < 0) {
160         perror("ioctl read packet failed");
161         close(simple_switch_fd);
162         return -1;
163     }
164     if (rcvd_pkt_meta >> 22) {
165         printf("Port 2: \n");
166         print_packet(&rcvd_pkt_meta);
167         total_latency += extra_time_delta(rcvd_pkt_meta);
168         num_get++;
169     }
170     usleep(1000);
171     if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_3, &rcvd_pkt_meta) < 0) {
172         perror("ioctl read packet failed");
173         close(simple_switch_fd);
174         return -1;
175     }
176     if (rcvd_pkt_meta >> 22) {
177         printf("Port 3: \n");
178         print_packet(&rcvd_pkt_meta);
179         total_latency += extra_time_delta(rcvd_pkt_meta);
180         num_get++;
181     }
182 }
183 printf("Got %d packets\n", num_get);
184 printf("Total latency: %d\n", total_latency);
185
186 close(simple_switch_fd);
187 printf("Userspace program terminating\n");
188 return 0;
189 }

```

12.3 Simple Switch: Verilator

12.3.1 Pick VOQ

```
1
2
3 #include <iostream>
4 #include <iomanip>
5 #include "Vpick_voq.h"
6 #include <verilated.h>
7 #include <bitset>
8
9 unsigned char input_voq_empty [ 8 ] = {0b1110, 0b0001, 0b0011, 0b1111, 0b0000,
10    0b1110, 0b0111, 0b0100, 0b0110, 0b0000};
11
12 unsigned char input_voq_picked [ 8 ] = {0b1110, 0b0001, 0b0011, 0b1111, 0b0000,
13    0b1110, 0b0111, 0b1000, 0b1001, 0b1111};
14
15 unsigned char input_start_voq_num [ 8 ] = {3, 3, 1, 2, 1, 1, 0, 2, 0, 0};
16
17 unsigned char output_no_available_voq [ 8 ] = {0, 0, 0, 1, 0, 0, 0, 0, 1, 1};
18
19 unsigned char output_voq_to_pick [ 8 ] = {0, 3, 2, 3, 1, 0, 3, 0, 0, 0};
20
21 int main(int argc, const char ** argv, const char ** env) {
22     int exitcode = 0;
23
24     Verilated::commandArgs(argc, argv);
25
26     Vpick_voq * dut = new Vpick_voq; // Instantiate the collatz module
27
28     for (int i = 0 ; i < 8; i++) {
29         dut->voq_empty = input_voq_empty [ i ];
30         dut->start_voq_num = input_start_voq_num [ i ];
31         dut->voq_picked = input_voq_picked [ i ];
32         dut->eval();
33         std::bitset<4> x(dut->voq_empty);
34         std::cout << "voq_empty: " << x << '\n';
35         std::bitset<4> y(dut->voq_picked);
36         std::cout << "voq_picked: " << y << '\n';
37
38         std::cout << "start_voq_num: " << (int) dut->start_voq_num << '\n';
39
40         if (dut->no_available_voq == output_no_available_voq [ i
41             ] && (dut->voq_to_pick == output_voq_to_pick [ i
42             ] || dut->no_available_voq == 1))
43             std::cout << " OK" << '\n';
44         else {
45             std::cout << " INCORRECT expected no_available_voq and voq_to_pick " <<
46                 std::endl;
47             exitcode = 1;
48         }
49     }
50     std::cout << "no_available_voq: " << (int) dut->no_available_voq << '\n';
```

```
46     std::cout << "voq_to_pick: " << (int) dut->voq_to_pick << '\n';
47     std::cout << std::endl;
48 }
49
50 dut->final(); // Stop the simulation
51 delete dut;
52
53 return exitcode;
54 }
```


12.3.2 VMU

```
1 #include <iostream>
2 #include "Vvmu.h"
3 #include <verilated.h>
4 #include <verilated_vcd_c.h>
5
6 using namespace std;
7
8 int main(int argc, const char ** argv, const char ** env) {
9     Verilated::commandArgs(argc, argv);
10    Vvmu * dut = new Vvmu;
11
12    Verilated::traceEverOn(true);
13    VerilatedVcdC * tfp = new VerilatedVcdC;
14    dut->trace(tfp, 99); // Verilator should trace signals up to 99 levels deep
15    tfp->open("vmu.vcd");
16
17
18    tfp->close(); // Stop dumping the VCD file
19    delete tfp;
20
21    dut->final(); // Stop the simulation
22    delete dut;
23    return 0;
24 }
```

12.3.3 Scheduler

```
1
2 #include <iostream>
3 #include "Vsched.h"
4 #include <verilated.h>
5 #include <verilated_vcd_c.h>
6
7 using namespace std;
8
9 unsigned short is_busy [ ] = {0b1110, 0b1110, 0b1110, 0b0000};
10
11 unsigned short busy_voq_num [ ] = {0b11100111, 0b11100111, 0b11100111, 0b00000000};
12
13 unsigned short voq_empty [ ] = {0b00000000000001110, 0b0000000000000000,
14     0b00000000000001111, 0b1011011111101101};
15
16 int main(int argc, const char ** argv, const char ** env) {
17     Verilated::commandArgs(argc, argv);
18
19     // Treat the argument on the command-line as the place to start
20     int n;
21     if (argc > 1 && argv [ 1 ] [ 0 ] != '+') n = atoi(argv [ 1 ]);
22     else n = 4; // Default
23
24     Vsched * dut = new Vsched; // Instantiate the sched module
25
26     // Enable dumping a VCD file
27
28     Verilated::traceEverOn(true);
29     VerilatedVcdC * tfp = new VerilatedVcdC;
30     dut->trace(tfp, 99); // Verilator should trace signals up to 99 levels deep
31     tfp->open("sched.vcd");
32
33     dut->sched_en = 0;
34     dut->is_busy = 0;
35     dut->busy_voq_num = 0;
36     dut->voq_empty = 0;
37
38     // std::cout << dut->n; // Print the starting value of the sequence
39
40     bool last_clk = true;
41     int time;
42     int iter = 0;
43     for (time = 0 ; time < 1000; time += 10) {
44         std::cout << "time: " << time << std::endl;
45         dut->clk = ((time % 20) >= 10) ? 1 : 0; // Simulate a 50 MHz clock
46         if ((time % 160) == 20) {
47             if (iter < n) {
48                 dut->sched_en = 0;
49                 dut->is_busy = is_busy [ iter ];
50                 dut->busy_voq_num = busy_voq_num [ iter ];
```

```

51     dut->voq_empty = voq_empty [ iter ];
52 }
53 } else if ((time % 160) == 30 || (time % 160) == 40) {
54     if (iter < n) {
55         dut->sched_en = 1;
56     }
57 } else {
58     dut->sched_en = 0;
59     if ((time % 160) == 50) {
60         iter += 1;
61         cout << iter << endl;
62     }
63 }
64
65 dut->eval(); // Run the simulation for a cycle
66 tfp->dump(time); // Write the VCD file for this cycle
67 }
68
69 std::cout << std::endl;
70
71 // Once "done" is received, run a few more clock cycles
72
73 // for (int k = 0 ; k < 4 ; k++, time += 10) {
74 //     dut->clk = ((time % 20) >= 10) ? 1 : 0;
75 //     dut->eval();
76 //     tfp->dump(time);
77 // }
78
79 tfp->close(); // Stop dumping the VCD file
80 delete tfp;
81
82 dut->final(); // Stop the simulation
83 delete dut;
84
85 return 0;
86 }

```

12.3.4 Simple Switch

```
1
2 #include <iostream>
3 #include "Vsimple_switch.h"
4 #include <verilated.h>
5 #include <verilated_vcd_c.h>
6
7 using namespace std;
8
9 int main(int argc, const char ** argv, const char ** env) {
10     Verilated::commandArgs(argc, argv);
11
12     // Treat the argument on the command-line as the place to start
13     int n;
14     if (argc > 1 && argv [ 1 ] [ 0 ] != '+') n = atoi(argv [ 1 ]);
15     else n = 4; // Default
16
17     Vsimple_switch * dut = new Vsimple_switch; // Instantiate the packet_gen module
18
19     // Enable dumping a VCD file
20
21     Verilated::traceEverOn(true);
22     VerilatedVcdC * tfp = new VerilatedVcdC;
23     dut->trace(tfp, 99); // Verilator should trace signals up to 99 levels deep
24     tfp->open("simple_switch.vcd");
25
26     // std::cout << dut->n; // Print the starting value of the sequence
27
28     bool last_clk = true;
29     int time;
30     int iter = 0;
31     dut->reset = 0;
32
33     for (time = 0 ; time < 10000; time += 10) {
34         std::cout << "time: " << time << std::endl;
35         dut->clk = ((time % 20) >= 10) ? 0 : 1; // Simulate a 50 MHz clock
36         if (time == 40) {
37             dut->chipselect = 1;
38             dut->address = 2;
39             dut -> write = 1;
40             dut -> read = 0;
41             dut->writedata = 0b01110000010000000000000000000000;
42         }
43         if (time == 60) {
44             dut -> chipselect = 0;
45         }
46
47         if (time == 1540) {
48             dut->chipselect = 1;
49             dut->address = 1;
50             dut -> write = 1;
51             dut -> read = 0;
```

```

52     dut->writedata = 0b00110000100000000000000000000000;
53 }
54 if (time == 1560) {
55     dut -> chipselect = 0;
56 }
57
58
59 if (time == 2440) {
60     dut->chipselect = 1;
61     dut->address = 3;
62     dut -> write = 1;
63     dut -> read = 0;
64     dut->writedata = 0b10110000100000000000000000000000;
65 }
66 if (time == 2460) {
67     dut -> chipselect = 0;
68     dut -> write = 0;
69 }
70
71 if (time == 3040) {
72     dut->chipselect = 1;
73     dut->address = 1;
74     dut -> write = 1;
75     dut -> read = 0;
76     dut->writedata = 0b00100000100000000000000000000000;
77 }
78
79 if (time == 3060) {
80     dut -> chipselect = 0;
81 }
82
83     if (time == 3040) {
84     dut->chipselect = 1;
85     dut->address = 1;
86     dut -> write = 1;
87     dut -> read = 0;
88     dut->writedata = 0b00100000100000000000000000000000;
89 }
90
91 if (time == 3560) {
92     dut -> chipselect = 0;
93 }
94     if (time == 3040) {
95     dut->chipselect = 1;
96     dut->address = 2;
97     dut -> write = 1;
98     dut -> read = 0;
99     dut->writedata = 0b01100000100000000000000000000000;
100 }
101 if (time == 3060) {
102     dut -> chipselect = 0;
103     dut -> write = 0;
104 }

```

```
105
106
107
108   if(time == 4000) {
109       dut -> write = 1;
110       dut -> chipselect = 1;
111       dut -> read = 0;
112       dut -> address = 0;
113       dut->writedata = 2;
114   }
115   if(time == 4020) {
116       dut -> chipselect = 0;
117   }
118
119   if(time == 5000) {
120       dut -> write = 0;
121       dut -> chipselect = 1;
122       dut -> read = 1;
123       dut -> address = 4;
124   }
125   if(time == 5020) {
126       dut -> chipselect = 0;
127   }
128
129   if(time == 7000) {
130       dut -> write = 0;
131       dut -> chipselect = 1;
132       dut -> read = 1;
133       dut -> address = 4;
134   }
135   if(time == 7020) {
136       dut -> chipselect = 0;
137   }
138
139   if(time == 8000) {
140       dut -> write = 0;
141       dut -> chipselect = 1;
142       dut -> read = 1;
143       dut -> address = 3;
144   }
145   if(time == 8020) {
146       dut -> chipselect = 0;
147   }
148   if(time == 9000) {
149       dut -> write = 0;
150       dut -> chipselect = 1;
151       dut -> read = 1;
152       dut -> address = 3;
153   }
154   if(time == 9020) {
155       dut -> chipselect = 0;
156       dut -> read = 0;
157   }
```

```
158
159     dut->eval(); // Run the simulation for a cycle
160     tfp->dump(time); // Write the VCD file for this cycle
161 }
162
163 std::cout << std::endl;
164
165 // Once "done" is received, run a few more clock cycles
166
167 // for (int k = 0 ; k < 4 ; k++, time += 10) {
168 //     dut->clk = ((time % 20) >= 10) ? 1 : 0;
169 //     dut->eval();
170 //     tfp->dump(time);
171 // }
172
173 tfp->close(); // Stop dumping the VCD file
174 delete tfp;
175
176 dut->final(); // Stop the simulation
177 delete dut;
178
179 return 0;
180 }
```

12.4 daFPGASwitch: Hardware

12.4.1 CMU

```
1  /*
2  The packet management unit:
3
4  */
5
6  /*
7  ctrl memory is divided into 16-bit chunks
8  and each of them represents a packet segment (32 bytes) inside the data memory
9
10 Inside each chunk:
11 * 1-bit of is_allocated
12 * 10-bit of next_addr (of ctrl memory, basically a linked list) / 10'b0 which is the
    end of packet chain.
13
14 A packet comes in that needs 2 block
15 Address 0: Null
16 Address 1: 1, 0000000002
17 Address 2: 1, 0000000000
18 */
19
20 define D_WIDTH 11
21 module cmu (
22     input logic clk,
23     input logic [ 5 : 0 ] remaining_packet_length, // in blocks
24     input logic alloc_en, free_en, reset,
25     input logic [ 9 : 0 ] free_addr,
26
27     output logic [ 9 : 0
28         ] alloc_addr = 10'b0, // the address in cmem for the first chunk of the packet
29     output logic [ 9 : 0 ] next_free_addr = 10'b0
30 );
31
32 // Registers
33 /* verilator lint_off UNOPTFLAT */
34 logic [ 9 : 0 ] curr_write;
35     logic [ 9 : 0 ] next_write;
36     logic [ 5 : 0 ] empty_blocks;
37 /* verilator lint_on UNOPTFLAT */
38 logic [ D_WIDTH -1 : 0 ] ctrl_in_a = D_WIDTH'b0;
39 logic [ D_WIDTH -1 : 0 ] ctrl_in_b = D_WIDTH'b0;
40 /* verilator lint_off UNUSED */
41 logic [ D_WIDTH -1 : 0 ] ctrl_out_a, ctrl_out_b;
42 /* verilator lint_on UNUSED */
43 logic ctrl_wen_a, ctrl_wen_b;
44 //logic[9:0] next_ctrl;
45 logic [ 9 : 0 ] addr_a, addr_b;
46
47 /* Create state control variables */
48 logic [ 2 : 0 ] input_state = 3'b0;
49 logic [ 2 : 0 ] next_input_state = 3'b0;
```



```

50 logic [ 1 : 0 ] output_state = 2'b0;
51 logic [ 1 : 0 ] next_output_state = 2'b0;
52
53 /* Temporary variables */
54 logic [ 5 : 0 ] temp_empty_blocks;
55 logic [ 9 : 0 ] temp_next_write;
56 logic [ 9 : 0 ] temp_curr_write;
57
58 always_comb begin
59     /* Create input state machine */
60     case (input_state)
61         /* Determine CMEM space availability
62          * Read the next control */
63         3'b000: begin
64             ctrl_wen_a = 1'b0;
65             if (alloc_en == 1'b1) begin
66                 if (remaining_packet_length < empty_blocks) begin
67                     temp_empty_blocks = empty_blocks - 1;
68                     addr_a = next_write;
69
70                     next_input_state = 3'b001;
71                 end else begin
72                     alloc_addr = 10'b0;
73                     next_input_state = 3'b000;
74                     temp_empty_blocks = empty_blocks;
75                 end
76             end
77         end else begin
78             next_input_state = 3'b000;
79             temp_empty_blocks = empty_blocks;
80         end
81     end
82
83     /* Determine the next write */
84     3'b001: begin
85         if (ctrl_out_a == 11'b0) begin
86             temp_next_write = next_write + 1;
87         end else begin
88             temp_next_write = ctrl_out_a [ 9 : 0 ];
89         end
90     end
91
92     next_input_state = 3'b010;
93 end
94
95 3'b010: begin
96
97     next_input_state = 3'b011;
98 end
99
100 3'b011: begin
101
102     next_input_state = 3'b100;

```

```

103         end
104
105     3'b100: begin
106
107         next_input_state = 3'b101;
108     end
109
110     3'b101: begin
111
112         next_input_state = 3'b110;
113     end
114
115     3'b110: begin
116
117         next_input_state = 3'b111;
118     end
119
120     /* Format the control and write to cmem
121     * Return the next free address */
122     3'b111: begin
123         ctrl_in_a [ 10 ] = 1'b1;
124
125         if (remaining_packet_length > 1) begin
126             ctrl_in_a [ 9 : 0 ] = next_write;
127         end else begin
128             ctrl_in_a [ 9 : 0 ] = 10'b0;
129         end
130
131         alloc_addr          = next_write;
132         temp_curr_write = next_write;
133
134         ctrl_wen_a          = 1'b1;
135         addr_a              = curr_write;
136
137         next_input_state = 3'b0;
138     end
139
140     default: begin
141     end
142
143 endcase
144
145
146 /* Create output state machine */
147 case (output_state)
148     /* Request to read control of the given address
149     * Format control to deallocate address*/
150     2'b00: begin
151         if (free_en == 1'b1) begin
152             ctrl_wen_b          = 1'b0;
153
154             addr_b              = free_addr;
155

```

```

156         ctrl_in_b = {1'b0, next_write};
157
158         next_output_state = 2'b01;
159     end
160 end
161
162     /* Read the control of the given address
163     * Write to deallocate control memory */
164 2'b01: begin
165     next_free_addr = ctrl_out_b [ 9 : 0 ];
166
167     ctrl_wen_b      = 1'b1;
168
169     temp_empty_blocks = empty_blocks + 1;
170     temp_next_write   = free_addr;
171     next_output_state = 2'b10;
172 end
173
174 2'b10: begin
175     ctrl_wen_b      = 1'b0;
176
177     next_output_state = 2'b00;
178 end
179
180 default: begin
181 end
182 endcase
183
184
185 end
186
187 always @(posedge clk) begin
188     if (reset == 1'b1) begin
189         empty_blocks <= 6'b111111;
190         next_write    <= 10'h1;
191         curr_write    <= 10'b1;
192
193         input_state <= 3'b0;
194         output_state <= 2'b0;
195
196     end else begin
197         input_state <= next_input_state;
198         output_state <= next_output_state;
199
200         empty_blocks <= temp_empty_blocks;
201         if (temp_next_write != 10'b0) begin
202             next_write    <= temp_next_write;
203         end
204
205         if (temp_curr_write != 10'b0) begin
206             curr_write <= temp_curr_write;
207         end
208

```

```
209         end
210
211
212     end
213
214     true_dual_port_mem #(.MEM_SIZE(1024), .DATA_WIDTH(D_WIDTH)) cmem
215     (
216         .clk(clk),
217         .aa(addr_a),          .ab(addr_b),
218         .da(ctrl_in_a),      .db(ctrl_in_b),
219         .wa(ctrl_wen_a),     .wb(ctrl_wen_b),
220         .qa(ctrl_out_a),     .qb(ctrl_out_b)
221     );
222 endmodule
```

12.4.2 Crossbar

```
1 module crossbar #(
2     parameter DATA_WIDTH = 2, // For testing
3     parameter EGRESS_CNT = 4
4 ) (
5     input logic [ $clog2 ( EGRESS_CNT ) * EGRESS_CNT -1 : 0 ] sched_sel,
6     input logic [ EGRESS_CNT -1 : 0 ] crossbar_in_en,
7     input logic [ DATA_WIDTH * EGRESS_CNT -1 : 0 ] crossbar_in,
8
9     output logic [ EGRESS_CNT -1 : 0 ] crossbar_out_en,
10    output logic [ DATA_WIDTH * EGRESS_CNT -1 : 0 ] crossbar_out
11 );
12
13 logic [ $clog2 ( EGRESS_CNT ) : 0 ] i;
14 logic [ $clog2 ( EGRESS_CNT ) -1 : 0 ] _i;
15
16 always_comb begin
17     for (i = 0; i < EGRESS_CNT; i = i + 1) begin
18         _i = i [ $clog2 ( EGRESS_CNT ) -1 : 0 ];
19         crossbar_out_en [ sched_sel [ i * $clog2 ( EGRESS_CNT ) + :
20             $clog2 ( EGRESS_CNT ) ] ] = crossbar_in_en [ _i ];
21         crossbar_out [ ( sched_sel [ i * $clog2 ( EGRESS_CNT ) + :
22             $clog2 ( EGRESS_CNT ) ] ) * DATA_WIDTH + :
23             DATA_WIDTH ] = crossbar_in [ _i * DATA_WIDTH + :
24             DATA_WIDTH ];
25     end
26 end
27 endmodule
```

12.4.3 daFPGASwitch

```
1 module da_fpga_switch
2 (
3     // From sw
4     input logic clk,
5     input logic reset,
6     input logic [ 31 : 0 ] writedata,
7     input logic write,
8     input logic read,
9     input logic [ 2 : 0 ] address,
10    input logic chipselect,
11
12    // To sw
13    output logic [ 31 : 0 ] readdata
14 );
15
16    logic meta_in_en, packet_en;
17    logic [ 31 : 0 ] meta_in, packet, meta_out;
18    logic experimenting;
19    // Change to 4 bit later.
20    logic meta_out_ack;
21
22    packet_gen ingress_0(
23        // Input
24        .clk(clk), .reset(reset),
25        .packet_gen_in_en(meta_in_en), .experimenting(experimenting),
26        .packet_gen_in(meta_in),
27
28        // Output
29        .packet_gen_out_en(packet_en), .packet_gen_out(packet));
30
31    packet_val egress_0 (
32        // Input
33        .clk(clk), .reset(reset),
34        .egress_in(packet),
35        .egress_in_en(packet_en), .egress_in_ack(meta_out_ack),
36
37        // Output
38        .egress_out(meta_out));
39
40    hw_sw_interface hw_sw_interface (
41        .clk(clk),
42
43        // Input: sw->interface
44        .reset(reset),
45        .writedata(writedata), // sw->hw
46        .write(write), // sw->hw
47        .read(read), // hw->sw
48        .address(address),
49        .chipselect(chipselect),
50
51        // Input: hw->interface
```

```
52     .interface_in(meta_out),
53
54     // Output: interface->sw
55     .readdata(readdata),
56
57     // Output: interface->egress (ack)
58     .interface_out_ack(meta_out_ack),
59
60     // Output: To ingress/packet_val
61     .interface_out_en(meta_in_en),
62     .interface_out(meta_in),
63
64     // Output: Ongoing experiment.
65     .experimenting(experimenting)
66
67 );
68
69 endmodule
```

12.4.4 Egress (Packet Validation)

```
1  /*
2
3  Packet metadata definition:
4  * src port: 2 bits
5  * dest port: 2 bits
6  * length: 12 bit (Packet size is 32 Bytes * length, min is 1 block = 32 Bytes, max is
   64 * 32 Bytes)
7
8
9  Part of packet (by definition)
10 * Length: 2 Bytes (00000xxxxx)
11 * Dest MAC: 6 Bytes
12 * Src MAC: 6 Bytes
13 * Start time
14 * End time
15
16 Part of packet (by block)
17 * Length: 2 Bytes + Dest MAC: 6 Bytes
18 * Time stamp: 8 bytes
19 * Src MAC: 2 garbage byte + 6 Bytes
20 * Data payload all 1's for now for the rest of the bits (at least 8 bytes)
21
22 */
23
24 define IDLE 4'b0000
25 define LENGTH_DMAC_FST 4'b0001
26 define LENGTH_DMAC_SND 4'b0010
27 define TIME_FST 4'b0011
28 define TIME_SND 4'b0100
29 define SMAC_FST 4'b0101
30 define SMAC_SND 4'b0110
31 define PAYLOAD 4'b0111
32
33
34
35 module packet_val #(
36     parameter PACKET_CNT = 1023,
37     BLOCK_SIZE = 32,
38     META_WIDTH = 32
39 ) (
40     input logic clk,
41     input logic reset,
42
43     // From crossbar
44     input logic [ META_WIDTH -1 : 0 ] egress_in,
45     input logic egress_in_en,
46
47     // From interface
48     input logic egress_in_ack,
49
50     // To interface
```



```

51     output logic [ 31 : 0 ] egress_out
52 );
53 /* verilator lint_off UNUSED */
54 logic [ META_WIDTH -1 : 0 ] next_meta;
55 logic [ META_WIDTH -1 : 0 ] meta;
56
57 logic [ $clog2 ( PACKET_CNT ) -1 : 0 ] start_idx; // The first element
58 logic [ $clog2 ( PACKET_CNT ) -1 : 0
59     ] end_idx; // One pass the last element
60 logic meta_ready;
61 logic [ 15 : 0 ] remaining_length;
62 logic [ 15 : 0 ] next_remaining_length;
63 logic [ 47 : 0 ] next_DMAC, DMAC;
64 logic [ 5 : 0 ] length_in_blocks;
65 logic [ 1 : 0 ] port_number;
66 logic [ 31 : 0 ] start_time, next_start_time;
67 logic [ 31 : 0 ] delta;
68
69 logic [ 3 : 0 ] state, next_state;
70 // logic [15:0] temp_length;
71 // assign temp_length = (egress_in[31:16] >> 5);
72 // assign length_in_blocks = temp_length[5:0];
73
74 // SMAC_FST, SMAC_SND Are not used
75
76 always_comb begin
77     length_in_blocks = egress_in [ 26 : 21 ];
78     delta = (egress_in - start_time);
79     case (state)
80         IDLE: begin
81             next_state          = LENGTH_DMAC_FST;
82             next_meta [ 31 : 28 ] = 0;
83             next_meta [ 21 : 0 ]  = 0;
84             meta_ready          = 0;
85             next_meta [ 27 : 22 ] = length_in_blocks;
86             next_start_time      = 0;
87
88             next_remaining_length = egress_in [ 31 : 16 ] - 4;
89             next_DMAC [ 47 : 32 ]      = egress_in [ 15 : 0 ];
90             next_DMAC [ 31 : 0 ] = DMAC [ 31 : 0 ];
91         end //START
92         LENGTH_DMAC_FST: begin
93             next_state = LENGTH_DMAC_SND;
94             meta_ready = 0;
95             next_DMAC [ 31 : 0 ] = egress_in;
96             next_DMAC [ 47 : 32 ] = DMAC [ 47 : 32 ];
97             next_meta = meta;
98             next_remaining_length = remaining_length - 4;
99             next_start_time = start_time;
100         end // LENGTH_DMAC_FST
101         LENGTH_DMAC_SND: begin
102             //next_remaining_length = egress_out[27:22];
103             next_remaining_length = remaining_length - 4;

```

```

103     next_state = TIME_FST;
104     next_meta [ 31 : 30 ] = meta [ 31 : 30 ];
105     next_meta [ 29 : 28 ] = port_number;
106     next_meta [ 27 : 0 ] = meta [ 27 : 0 ];
107     next_start_time = egress_in;
108     next_DMAC = DMAC;
109     meta_ready = 0;
110 end // LENGTH_DMAC_SND
111 TIME_FST: begin
112     next_state = TIME_SND;
113     next_meta [ 21 : 0 ] = delta [ 21 : 0 ];
114     next_meta [ 31 : 22 ] = meta [ 31 : 22 ];
115     meta_ready = 0;
116     next_remaining_length = remaining_length - 4;
117     next_start_time = start_time;
118     next_DMAC = DMAC;
119 end // TIME_FST
120 TIME_SND: begin
121     next_state = SMAC_FST;
122     next_remaining_length = remaining_length - 4;
123     next_meta = meta;
124     meta_ready = 0;
125     next_start_time = start_time;
126     next_DMAC = DMAC;
127 end // TIME_SND
128 SMAC_FST: begin
129     next_state = SMAC_SND;
130     next_meta [ 31 : 30 ] = egress_in [ 1 : 0 ];
131     next_meta [ 29 : 0 ] = meta [ 29 : 0 ];
132     meta_ready = 0;
133     next_remaining_length = remaining_length - 4;
134     next_start_time = start_time;
135     next_DMAC = DMAC;
136 end //SMAC_FST
137 SMAC_SND: begin
138     next_state = PAYLOAD;
139     next_remaining_length = remaining_length - 4;
140     next_meta = meta;
141     meta_ready = 0;
142     next_start_time = start_time;
143     next_DMAC = DMAC;
144 end //SMAC_SND
145 PAYLOAD: begin
146     if (remaining_length > 4) begin
147         next_remaining_length = remaining_length - 4;
148         next_state = PAYLOAD;
149         meta_ready = 0;
150         next_meta = meta;
151         next_start_time = start_time;
152         next_DMAC = DMAC;
153     end // remaining_length > 0
154     else begin
155         next_remaining_length = remaining_length;

```

```

156     next_state = IDLE; //UNDER QUESTION (GREATER THAN 0 OR GREATER THAN 1)
157     meta_ready = 1;
158     next_meta = meta;
159     next_start_time = start_time;
160     next_DMACE    = DMACE;
161     end // else remaining_length < 0
162 end // SRC_PAYLOAD
163 default: begin
164     meta_ready = 0;
165 end
166
167 endcase //end case
168
169 end // end always_comb
170
171
172 always_ff @(posedge clk) begin
173     if (reset) begin
174         start_idx <= 0;
175         end_idx <= 0;
176         state <= IDLE;
177     end //if reset
178     else begin
179
180         /* packet->meta, the input*/
181         if (egress_in_en) begin
182             if (next_state == IDLE) begin
183                 end_idx <= (end_idx != PACKET_CNT - 1) ? end_idx + 1 :0;
184             end
185             state <= next_state;
186             remaining_length <= next_remaining_length;
187             start_time <= next_start_time;
188             meta <= next_meta;
189             DMACE <= next_DMACE;
190         end //meta_en
191
192         /* Output */
193         if (egress_in_ack && (start_idx != end_idx)) begin
194
195             start_idx <= (start_idx != PACKET_CNT - 1) ? start_idx + 1 :0;
196
197         end // egress_in_ack
198     end // not reset
199
200 end //always_ff
201
202
203 simple_dual_port_mem #(
204     .MEM_SIZE (PACKET_CNT),
205     .DATA_WIDTH(32)
206 ) meta_mem (
207     .clk(clk),
208     .ra(start_idx),

```

```
209     .wa(end_idx),
210     .d(meta),
211     .q(egress_out),
212     .write(meta_ready)
213 );
214
215 mac_to_port mac_to_port_0 (
216     .MAC(DMAC),
217     .port_number(port_number)
218 );
219
220 endmodule
```

12.4.5 Hardware-Software Interface

```
1 module hw_sw_interface(  
2     input logic clk,  
3  
4     // From sw  
5     input logic    reset,  
6     input logic    [ 31 : 0 ] writedata,  
7     input logic    write,  
8     input logic    read,  
9     input logic    [  2 : 0 ] address,  
10    input logic    chipselect,  
11  
12    // From egress  
13    input logic    [ 31 : 0 ] interface_in,  
14  
15    // To sw  
16    output logic   [ 31 : 0 ] readdata,  
17  
18    // To packet_val/ingress  
19    // Change to 4 bits later  
20    output logic   interface_out_en,  
21    output logic   [ 31 : 0 ] interface_out,  
22  
23    // Experimenting  
24    output logic   experimenting,  
25  
26    // Special case: Because we're polling but not handling interrupt  
27    // we need to acknowledge that this metadata is consumed by the software.  
28    // This is the only ack in our program.  
29    // Change to 4 bits later  
30    output logic   interface_out_ack  
31 );  
32  
33 logic    [ 31 : 0 ] ctrl;  
34  
35 always_comb begin  
36     experimenting = (ctrl == 2);  
37 end  
38  
39 always_ff @(posedge clk) begin  
40     if (reset) begin  
41         ctrl <= 32'h0;  
42         readdata <= 32'h0;  
43         interface_out_en <= 0;  
44         interface_out_ack <= 0;  
45         interface_out <= 0;  
46     end else begin  
47         if (chipselect && write) begin  
48             case (address)  
49                 3'h0: begin  
50                     ctrl <= writedata;  
51                     end
```

```

52     3'h1: begin
53         interface_out_en <= 1;
54         // interface_out_en[0] <= 1;
55         interface_out <= writedata;
56     end
57     3'h2: begin
58         // interface_out_en[1] <= 1;
59         interface_out <= writedata;
60     end
61     3'h3: begin
62         // interface_out_en[2] <= 1;
63         interface_out <= writedata;
64     end
65     3'h4: begin
66         // interface_out_en[3] <= 1;
67         interface_out <= writedata;
68     end
69     default: begin
70     end
71 endcase
72 end else begin
73     // interface_out_en[0] <= 0;
74     // interface_out_en[1] <= 0;
75     // interface_out_en[2] <= 0;
76     // interface_out_en[3] <= 0;
77     interface_out_en <= 0;
78 end
79 if (chipselct && read) begin
80     readdata <= interface_in;
81     case (address)
82     3'h1: begin
83         readdata <= interface_in;
84         // interface_out_ack[0] <= 1;
85         interface_out_ack <= 1;
86     end
87     3'h2: begin
88         readdata <= interface_in;
89         // interface_out_ack[1] <= 1;
90     end
91     3'h3: begin
92         readdata <= interface_in;
93         // interface_out_ack[2] <= 1;
94     end
95     3'h4: begin
96         readdata <= interface_in;
97         // interface_out_ack[3] <= 1;
98     end
99     default: begin
100    end
101 endcase
102 end else begin
103     interface_out_ack <= 0;
104     // interface_out_ack[0] <= 0;

```

```
105     // interface_out_ack[1] <= 0;
106     // interface_out_ack[2] <= 0;
107     // interface_out_ack[3] <= 0;
108     end
109     end
110 end
111
112 endmodule
```

12.4.6 Ingress

```
1
2 define IN_IDLE 4'b0000
3 define IN_0 4'b0001
4 define IN_1 4'b0010
5 define IN_2 4'b0011
6 define IN_3 4'b0100
7 define IN_4 4'b0101
8 define IN_5 4'b0110
9 define IN_6 4'b0111
10 define IN_7 4'b1000
11
12
13 define IDLE 2'b00
14 define SEND_META_2_CMU 2'b01
15 define GET_NEXT_ADDR 2'b10
16 define SEND_PACKET 2'b11
17
18 module ingress (
19     input logic          clk,
20     input logic          reset,
21     input logic          [ 31 : 0 ] packet_in, // From packet gen, getting packet
22     input logic          packet_en, // From packet gen, meaning that we should
        start recvng a packet segment
23     //input logic          new_packet_en, // From packet gen, meaning that we should
        start recvng a new packet
24
25     input logic          [ 1 : 0
        ] sched_sel, // From scheduler, the voq to dequeue the packet
26     input logic          sched_done, // (actually SCHED_ENABLE) From
        scheduler, meaning that we should start sending a packet segment
27
28     // output logic          sched_en, // To sch
29     output logic          [ 31 : 0 ] packet_out, // To crossbar
30     output logic          packet_out_en // To crossbar
31 );
32
33     //logic [2:0] input_counter;
34     logic          [ 2 : 0 ] send_cycle_counter, next_send_cycle_counter;
35
36     logic          [ 5 : 0 ] next_remaining_packet_length, remaining_packet_length;
37     logic          [ 31 : 0 ] temp_packet_in;
38     /* verilator lint_off UNOPTFLAT */
39     logic          alloc_en;
40     logic          [ 47 : 0 ] d_mac, next_d_mac;
41     logic          [ 31 : 0 ] packet_start_time_logic;
42     logic          [ 3 : 0 ] in_state, next_in_state;
43     /* verilator lint_on UNOPTFLAT */
44
45     logic          [ 12 : 0 ] curr_d_write, curr_d_read;
46     logic          voq_enqueue_en;
47     logic          [ 1 : 0 ] voq_enqueue_sel, port_number;
```



```

48     logic         first_packet;
49
50
51
52     logic [ 9 : 0 ]   alloc_addr;
53     logic [ 9 : 0 ]   meta_in, meta_out;
54     assign meta_in = (first_packet == 1) ? 10'b1 : alloc_addr;
55     /* States */
56
57     logic [ 1 : 0 ]   out_state, next_out_state;
58
59     /* time */
60     logic [ 31 : 0 ]  curr_time;
61
62     logic [ 12 : 0 ]  start_addr;
63     /* verilator lint_off UNOPTFLAT */
64     logic [ 12 : 0 ]  start_addr_reg;
65     /* verilator lint_on UNOPTFLAT */
66
67     /* verilator lint_off UNUSED */
68     assign start_addr = {3'b0, alloc_addr} << 2;
69     logic [ 31 : 0 ]  offset_addr;
70     assign offset_addr = ((curr_time - packet_start_time_logic) % 8);
71
72
73     logic free_en;
74     logic [ 9 : 0 ]  free_addr, next_free_addr, voq_meta_out,
75         voq_meta_out_reg, free_addr_reg, next_free_addr_reg;
76     logic [ 3 : 0 ]  is_empty, is_full;
77     logic          voq_dequeue_en;
78     //logic [1:0] voq_dequeue_sel;
79
80     logic [ 12 : 0 ]  data_read_addr;
81     assign data_read_addr = {3'b000, voq_meta_out} << 2;
82
83     always_comb begin
84         case (in_state)
85             IN_IDLE: begin
86                 next_remaining_packet_length = 0;
87                 next_d_mac [ 47 : 32 ]      = 0;
88                 next_d_mac [ 31 : 0 ]      = 0;
89                 temp_packet_in              = 0;
90                 alloc_en                    = 0;
91                 packet_start_time_logic    = curr_time;
92                 next_in_state               = IN_0;
93                 curr_d_write                = 0;
94                 voq_enqueue_en             = 0;
95                 voq_enqueue_sel             = 0;
96             end
97             IN_0: begin
98                 next_remaining_packet_length = packet_in [ 26 : 21 ];

```

```

99         next_d_mac [ 47 : 32 ] = packet_in [
          15 : 0 ];
100        next_d_mac [ 31 : 0 ] = 0;
101        temp_packet_in = packet_in;
102        alloc_en = 1;
103        packet_start_time_logic = curr_time;
104        next_in_state = IN_1;
105        curr_d_write = (first_packet == 1) ?
          13'b1 : start_addr;
106        voq_enqueue_en = 0;
107        voq_enqueue_sel = port_number;
108        end
109        IN_1: begin
110            next_remaining_packet_length = remaining_packet_length;
111            next_d_mac [ 47 : 32 ] = d_mac [ 47
          : 32 ];
112            next_d_mac [ 31 : 0 ] = packet_in;
113            temp_packet_in = packet_in;
114            alloc_en = 0;
115            next_in_state = IN_2;
116            curr_d_write = start_addr+1;
117            start_addr_reg = start_addr;
118            voq_enqueue_en = 0;
119            voq_enqueue_sel = port_number;
120            end
121            IN_2: begin
122                next_remaining_packet_length = remaining_packet_length;
123                next_d_mac = d_mac;
124                temp_packet_in = curr_time;
125                alloc_en = 0;
126                next_in_state = IN_3;
127                curr_d_write = start_addr+2;
128                voq_enqueue_en = 1;
129                voq_enqueue_sel = port_number;
130            end
131            IN_3: begin
132                next_remaining_packet_length = remaining_packet_length;
133                next_d_mac = d_mac;
134                temp_packet_in = packet_in;
135                alloc_en = 0;
136                next_in_state = IN_4;
137                curr_d_write = start_addr+3;
138                voq_enqueue_en = 0;
139                voq_enqueue_sel = port_number;
140            end
141            IN_4: begin
142                next_remaining_packet_length = remaining_packet_length;
143                next_d_mac = d_mac;
144                temp_packet_in = packet_in;
145                alloc_en = 0;
146                next_in_state = IN_5;
147                curr_d_write = start_addr+4;
148            end

```

```

149         voq_enqueue_en           = 0;
150         voq_enqueue_sel         = port_number;
151     end
152     IN_5: begin
153         next_remaining_packet_length = remaining_packet_length;
154         next_d_mac                  = d_mac;
155         temp_packet_in              = packet_in;
156         alloc_en                    = 0;
157         next_in_state               = IN_6;
158         curr_d_write                = start_addr+5;
159         voq_enqueue_en             = 0;
160         voq_enqueue_sel            = port_number;
161     end
162     IN_6: begin
163         next_remaining_packet_length = remaining_packet_length;
164         next_d_mac                  = d_mac;
165         temp_packet_in              = packet_in;
166         alloc_en                    = 0;
167         next_in_state               = IN_7;
168         curr_d_write                = start_addr+6;
169         voq_enqueue_en             = 0;
170         voq_enqueue_sel            = port_number;
171     end
172     IN_7: begin
173         voq_enqueue_en             = 0;
174         voq_enqueue_sel            = port_number;
175         if(remaining_packet_length > 1) begin
176             next_in_state          = IN_7;
177         end else begin
178             next_in_state          = IN_0;
179         end
180
181         next_remaining_packet_length = remaining_packet_length - 1;
182         next_d_mac                  = d_mac;
183         temp_packet_in              = packet_in;
184
185         if((curr_time - packet_start_time_logic) % 8 == 0) begin
186             alloc_en                = 1;
187             curr_d_write            = start_addr;
188         end else if ((curr_time - packet_start_time_logic) % 8 ==
189             7) begin
190             alloc_en = 0;
191             curr_d_write                = start_addr_reg
192                 + 7;
193             //start_addr_reg          = start_addr;
194         end else begin
195             alloc_en = 0;
196             curr_d_write                = start_addr +
197                 offset_addr [ 12 : 0

```

```

198         end
199         default: begin end
200     endcase
201
202     case (out_state)
203         IDLE: begin
204             free_en = 1'b0;
205             free_addr = voq_meta_out;
206             voq_meta_out_reg = meta_out;
207             next_send_cycle_counter = 0;
208             next_out_state = (sched_done) ? SEND_META_2_CMU : IDLE; //
                no packet or no decision
209             packet_out_en = 0;
210         end
211         SEND_META_2_CMU: begin
212             free_en = 1'b1;
213             free_addr = voq_meta_out;
214             curr_d_read = data_read_addr;
215
216             next_send_cycle_counter = 1;
217             next_out_state = GET_NEXT_ADDR;
218
219             packet_out_en = 0;
220         end
221         GET_NEXT_ADDR: begin
222             free_en = 1'b0;
223             free_addr = voq_meta_out;
224             free_addr_reg = next_free_addr;
225             curr_d_read = data_read_addr + {9'b0, send_cycle_counter};
226             next_send_cycle_counter = 2;
227             next_out_state = SEND_PACKET;
228
229             packet_out_en = 1;
230         end
231         SEND_PACKET: begin
232             packet_out_en = 1;
233
234             free_en = 1'b0;
235             free_addr = voq_meta_out;
236             curr_d_read = data_read_addr + {9'b0, send_cycle_counter};
237             next_send_cycle_counter = (send_cycle_counter == 7) ? 0 :
                (send_cycle_counter + 1);
238             if (send_cycle_counter == 7 && next_free_addr_reg !=
                10'b0) begin
239                 next_out_state = SEND_META_2_CMU;
240                 voq_meta_out_reg = next_free_addr_reg;
241             end else if (send_cycle_counter == 7 && next_free_addr_reg
                == 10'b0) begin
242                 next_out_state = IDLE;
243             end else begin
244                 next_out_state = SEND_PACKET;
245             end
246         end

```

```

247         default: begin
248             end
249         endcase
250     end
251
252     assign next_free_addr_reg = free_addr_reg;
253
254
255     always_ff @(posedge clk) begin
256         if (reset) begin
257             in_state <= IN_IDLE;
258             out_state <= IDLE;
259             curr_time <= 0;
260             remaining_packet_length <= 0;
261             first_packet <= 0;
262             send_cycle_counter <= 0;
263         end // if reset
264         else begin
265             /* Time driver*/
266             curr_time <= curr_time + 1;
267
268             /* Input */
269             if (packet_en) begin
270                 if (in_state == IN_7) begin
271                     first_packet <= 0;
272                 end
273                 in_state <= next_in_state;
274                 d_mac <= next_d_mac;
275                 remaining_packet_length <= next_remaining_packet_length;
276             end
277
278             out_state <= next_out_state;
279             send_cycle_counter <= next_send_cycle_counter;
280
281             voq_meta_out <= voq_meta_out_reg;
282
283
284         end
285     end
286
287     vmu #(.PACKET_CNT(1024), .EGRESS_CNT(4)) voq_mu(
288         // Input
289         .clk(clk),
290         .voq_enqueue_en(voq_enqueue_en), .voq_enqueue_sel(voq_enqueue_sel),
291         .voq_dequeue_en(sched_done), .voq_dequeue_sel(sched_sel),
292         .meta_in(meta_in),
293         // Output
294         .meta_out(meta_out),
295         .is_empty(is_empty), .is_full(is_full)
296     );
297
298     cmu ctrl_mu(
299         // Input when inputting

```

```

300     .clk(clk),
301 .reset(reset),
302     // From input
303     .remaining_packet_length(remaining_packet_length), // in blocks
304     .alloc_en(alloc_en), // writing data in
305     // From output
306     .free_addr(free_addr),
307     .free_en(free_en),
308 // To input
309     .alloc_addr(alloc_addr),
310     // To output
311     .next_free_addr(next_free_addr) // retrieve the "next" of the free_addr
312 );
313
314 simple_dual_port_mem #(.MEM_SIZE(1024*8), .DATA_WIDTH(32)) dmem(
315     .clk(clk),
316     .ra(curr_d_read), .wa(curr_d_write),
317     .d(temp_packet_in), .q(packet_out),
318     .write(packet_en)
319 );
320
321 mac_to_port mac_to_port_2(.MAC(d_mac), .port_number(port_number));
322
323
324 endmodule

```

12.4.7 Scheduler

```
1 module sched (
2     input logic clk,
3     input logic sched_en,
4     input logic [ 3 : 0 ] is_busy,
5     input logic [ 7 : 0 ] busy_voq_num,
6     input logic [ 15 : 0 ] voq_empty,
7     input logic policy, // We have doubly RR, or priority based, can be controlled by
8                           software
9     input logic [ 7 : 0 ] prio,
10    output logic [ 3 : 0
11    ] sched_sel_en, // passed by to ingress, to know which ingress should dequeue
12    output logic [ 7 : 0
13    ] sched_sel // passed by to ingress, to know which voq to dequeue
14 );
15
16 /*
17 Some design choices:
18 * Do we want the time to return a scheduling decision to be deterministic or random
19   (btw 1 and 4)
20 * Should each busy port to just start transmitting without waiting for the
21   scheduling decision
22 * Should the scheduler decision the scheduling decision for this cycle or next cycle
23 * RR on the ingress or egress side, or both
24 * Should we try to do all combinator?
25 * Do we use 1 voq_to_pick or 4 voq_to_pick? (1 since it's going to take 4 cycles
26   anyway)
27 */
28
29 /*
30 Some principles:
31 * We don't want egress 1 to always recv packet from ingress 0; we don't want
32   ingress 0 to always send to egress 2
33 * nested loop fully in combinator logic is too expensive; instead, we do the outer
34   for loop sequentially, and the inner 4 loop in comb logic
35 RR policy:
36 * ingress RR: Start_ingress_idx proceed in each cycle of ASSIGN_NEW (or one pass
37   who-ever gets to select first in this cycle)
38 * egress RR: Each start_voq_num is first_non_empty_num of this cycle + 1;
39 * fewer first: Prioritize queue with only one non-empty voq.
40 */
41
42 /*
43 Notice:
44 * Beware of the possiblity of input (voq_empty for example) change during the
45   process.
46   - Possibly, use some local variables to save the inputs.
47   - Or, let the ingress be in charge of only updating the signal when sched_en.
48 * Busy ports need to be handled first.
49 * All the data need to be prepared before sched_enable. So at T-1 prepare input,
50   and at T sched_enable.
51 */
```

```

41
42 // If the scheduler is in the process of assigning new packet
43 // 0: not assigning; 1~4: assigning. 5 enable 6 is /enable
44 logic [ 2 : 0 ] assigning_new;
45
46 logic [ 3 : 0 ] ingress_enable; // the enable signal ready to be passed to
    sched_sel_en when ingress_done = 4'b1111
47
48 // For RR
49 logic [ 1 : 0
    ] start_ingress_idx; // Which ingress has the highest priority in this cycle
50 logic [ 7 : 0 ] start_voq_num; // Which egress has the highest priority in
    this cycle, for each ingress.
51
52 logic [ 1 : 0 ] curr_ingress_idx; // Current ingress
53 logic [ 2 : 0 ] curr_in_2;
54 logic [ 3 : 0 ] curr_in_4;
55 logic [ 3 : 0 ] voq_picked; // is the voq/egress picked?
56 logic no_available_voq; // Is the voqs/egress of the current ingress all
    empty/occupied by other?
57 logic [ 1 : 0
    ] voq_to_pick; // What is the voq_to_pick for the current ingress
58 logic [ 3 : 0 ] busy_egress_mask;
59
60 logic [ 2 : 0 ] i;
61 logic [ 1 : 0 ] busy_port;
62
63 always_comb begin
64     busy_egress_mask = 0; // This is important: busy_egress_mask need a way to start
        with all unoccupied.
65     for (i = 0; i < 4; i = i + 1) begin
66         if (is_busy [ i [ 1 : 0 ] ] == 1'b1) begin
67             busy_egress_mask [ busy_voq_num [ ( i << 1) + : 2 ]
                ] = 1'b1;
68         end
69     end
70     curr_in_2 = {1'b0, curr_ingress_idx} << 1;
71     curr_in_4 = {2'b0, curr_ingress_idx} << 2; // * 4 is << 2
72 end
73
74 initial begin
75     start_ingress_idx = 0;
76     start_voq_num = 0;
77 end
78
79 always_ff @(posedge clk) begin
80
81     if (sched_en) begin
82         // If we begin to schedule
83         // reset sched_sel_en
84         sched_sel_en <= 0;
85         // all the busy ingress ports are automatically assigned.
86         ingress_enable <= 0;
87         // start to assign ports for non-empty

```



```

88     assigning_new <= 1;
89     voq_picked <= busy_egress_mask;
90     curr_ingress_idx <= start_ingress_idx; // Start with start_ingress_idx
91 end else if (assigning_new == 5) begin // alternatively, if we manage to go back
    to start_ingress_idx
92     // If all are assigned, we're going start enabling
93     sched_sel_en <= ingress_enable;
94     // Nex time it should start with another index.
95     start_ingress_idx <= (start_ingress_idx == 3) ? 0 :start_ingress_idx + 1;
96     assigning_new <= 6;
97 end else if (assigning_new == 6) begin
98     sched_sel_en <= 0;
99 end else if (assigning_new >= 1 && assigning_new <= 4) begin
100     curr_ingress_idx <= (curr_ingress_idx == 3) ? 0 :curr_ingress_idx + 1;
101     assigning_new <= assigning_new + 1;
102     if (!is_busy [ curr_ingress_idx ]) begin
103         if (!no_available_voq) begin
104             ingress_enable [ curr_ingress_idx ] <= 1'b1;
105             voq_picked [ voq_to_pick ] <= 1'b1;
106             sched_sel [ curr_in_2 + : 2 ] <= voq_to_pick;
107             start_voq_num [ curr_in_2 + : 2 ] <=
108                 (start_voq_num [ curr_in_2 + : 2 ] == 3) ? 0 :start_voq_num [
                    curr_in_2 + : 2 ] + 1; // Alternatively, we can choose not to
                    move forward when no_available_voq.
109         end
110     end else begin
111         busy_port <= busy_voq_num [ curr_in_2 + : 2 ];
112         ingress_enable [ curr_ingress_idx ] <= 1'b1;
113         sched_sel [ curr_in_2 + : 2 ] <= busy_port;
114         voq_picked [ busy_port ] <= 1'b1;
115     end
116 end
117 end
118
119 // pick_voq will pick return the current ingress's first non empty voq to dequeue
    from.
120 pick_voq pv (
121     .start_voq_num(start_voq_num [ curr_in_2 + : 2 ]),
122     .voq_empty(voq_empty [ curr_in_4 + : 4 ]),
123     .voq_picked(voq_picked),
124     .no_available_voq(no_available_voq),
125     .voq_to_pick(voq_to_pick),
126     .policy(policy),
127     .prio(prio)
128 );
129 endmodule

```

12.4.8 Packet Generation

```
1
2
3 /*
4
5 Packet metadata definition:
6 * src port: 2 bits
7 * dest port: 2 bits
8 * length: 12 bit (Packet size is 32 Bytes * length, min is 1 block = 32 Bytes, max is
9     64 * 32 Bytes)
10
11 Part of packet (by definition)
12 * Length: 2 Bytes (0000xxxxx)
13 * Dest MAC: 6 Bytes
14 * Src MAC: 6 Bytes
15 * Start time
16 * End time
17
18 Part of packet (by block)
19 * Length: 2 Bytes + Dest MAC: 6 Bytes
20 * Time stamp: 8 bytes
21 * Src MAC: 2 garbage byte + 6 Bytes
22 * Data payload all 1's for now for the rest of the bits (at least 8 bytes)
23
24 */
25
26 define IDLE 4'b0000
27 define LENGTH_DMAC_FST 4'b0001
28 define LENGTH_DMAC_SND 4'b0010
29 define TIME_FST 4'b0011
30 define TIME_SND 4'b0100
31 define SMAC_FST 4'b0101
32 define SMAC_SND 4'b0110
33 define PAYLOAD 4'b0111
34
35
36
37 module packet_gen #(
38     parameter PACKET_CNT = 1024,
39     BLOCK_SIZE = 32,
40     META_WIDTH = 32
41 ) (
42     input logic          clk,
43     input logic          reset,
44
45     input logic [ META_WIDTH -1 : 0 ] packet_gen_in,
46     input logic          packet_gen_in_en,
47     input logic          experimenting,
48
49     // To Ingress
50     output logic         packet_gen_out_en,
```

```

51     output logic [ 31 : 0 ] packet_gen_out
52 );
53 logic [ META_WIDTH -1 : 0 ] meta_out;
54
55 logic [ $clog2 ( PACKET_CNT ) -1 : 0 ] start_idx; // The first element
56 logic [ $clog2 ( PACKET_CNT ) -1 : 0
57     ] end_idx; // One pass the last element
58
59 logic [ 15 : 0 ] remaining_length;
60 logic [ 15 : 0 ] next_remaining_length;
61 logic [ 47 : 0 ] DMAC, SMAC; // A lot of registers, be really careful!
62 logic [ 15 : 0 ] length_in_bytes;
63 logic [ 3 : 0 ] state, next_state;
64 assign length_in_bytes = meta_out [ 27 : 22 ] * 32;
65 // SMAC_FST, SMAC_SND Are not used
66
67 always_comb begin
68     case (state)
69         IDLE: begin
70             next_state = LENGTH_DMAC_FST;
71             packet_gen_out = 0;
72             packet_gen_out_en = 0;
73             next_remaining_length = remaining_length;
74         end //START
75         LENGTH_DMAC_FST: begin
76             next_state = LENGTH_DMAC_SND;
77             packet_gen_out = {length_in_bytes, DMAC [ 47 : 32 ]};
78             packet_gen_out_en = 1;
79             next_remaining_length = length_in_bytes - 4;
80         end // LENGTH_DMAC_FST
81         LENGTH_DMAC_SND: begin
82             next_state = TIME_FST;
83             packet_gen_out = DMAC [ 31 : 0 ];
84             packet_gen_out_en = 1;
85             next_remaining_length = remaining_length - 4;
86         end // LENGTH_DMAC_SND
87         TIME_FST: begin
88             next_state = TIME_SND;
89             packet_gen_out = {10'b0, meta_out [ 21 : 0 ]};
90             packet_gen_out_en = 1;
91             next_remaining_length = remaining_length - 4;
92         end // TIME_FST
93         TIME_SND: begin
94             next_state = SMAC_FST;
95             packet_gen_out = 32'b0;
96             packet_gen_out_en = 1;
97             next_remaining_length = remaining_length - 4;
98         end // TIME_SND
99         SMAC_FST: begin
100             next_state = SMAC_SND;
101             packet_gen_out = {16'b0, SMAC [ 47 : 32 ]};
102             packet_gen_out_en = 1;

```

```

103     next_remaining_length = remaining_length - 4;
104 end //SMAC_FST
105 SMAC_SND: begin
106     next_state = PAYLOAD;
107     packet_gen_out    = SMAC [ 31 : 0 ];
108     packet_gen_out_en = 1;
109     next_remaining_length = remaining_length - 4;
110 end //SMAC_SND
111 PAYLOAD: begin
112     if (remaining_length > 4) begin
113         next_state = PAYLOAD;
114     end else begin
115         next_state = LENGTH_DMAC_FST;
116     end
117
118     next_remaining_length = remaining_length - 4;
119     packet_gen_out = ~32'b0;
120     packet_gen_out_en = 1;
121
122 end
123 default: begin
124     packet_gen_out_en = 0;
125     packet_gen_out = 0;
126     next_remaining_length = remaining_length;
127     next_state = state;
128
129 end
130 endcase //end case
131
132 end // end always_comb
133
134
135 always_ff @(posedge clk) begin
136     if (reset) begin
137         start_idx <= 0;
138         end_idx <= 0;
139         state <= IDLE;
140         //packet_gen_out_en = 0;
141     end //if reset
142     else begin
143         if (packet_gen_in_en) begin
144             end_idx <= (end_idx == 1023) ? 0 : end_idx + 1;
145
146         end //packet_gen_in_en
147         if (experimenting) begin
148             if (next_state == IDLE) begin
149                 start_idx <= (start_idx != 1023) ? start_idx + 1 : 0;
150             end
151             state <= next_state;
152             remaining_length <= next_remaining_length;
153         end // experimenting
154     end // not reset
155

```

```
156 end //always_ff
157
158
159 simple_dual_port_mem #(
160     .MEM_SIZE (PACKET_CNT),
161     .DATA_WIDTH(32)
162 ) vmem (
163     .clk(clk),
164     .ra(start_idx),
165     .wa(end_idx),
166     .d(packet_gen_in),
167     .q(meta_out),
168     .write(packet_gen_in_en)
169 );
170
171 port_to_mac port_to_mac_0 (
172     .port_number(meta_out [ 29 : 28 ]),
173     .MAC(DMAC)
174 );
175 port_to_mac port_to_mac_1 (
176     .port_number(meta_out [ 31 : 30 ]),
177     .MAC(SMAC)
178 );
179
180 endmodule
```

12.4.9 True Dual Port Memory

```
1
2 module true_dual_port_mem #(
3     parameter MEM_SIZE = 1024, /* How many bits of memory in total, 1024 by default */
4     parameter DATA_WIDTH = 16 /* How many bit of data per cycle, 16 by default */
5 ) (
6     input logic clk,
7     input logic [ $clog2 ( MEM_SIZE ) - 1 : 0 ] aa, ab, /* Address */
8     input logic [ DATA_WIDTH - 1 : 0 ] da, db, /* input data */
9     input logic wa, wb, /* Write enable */
10    output logic [ DATA_WIDTH - 1 : 0 ] qa, qb /* output data */
11 );
12 logic [ DATA_WIDTH - 1 : 0 ] mem [ MEM_SIZE - 1 : 0 ];
13
14 // First port
15 always_ff @(posedge clk) begin
16     if (wa) begin
17         mem [ aa ] <= da;
18         qa <= da;
19     end else qa <= mem [ aa ];
20 end
21
22 // Second port
23 always_ff @(posedge clk) begin
24     if (wb) begin
25         mem [ ab ] <= db;
26         qb <= db;
27     end else qb <= mem [ ab ];
28 end
29 endmodule
```

12.4.10 VMU

```
1  /*
2  The virtual output queue management unit:
3  */
4
5  /*
6  voq memory is divided into 16-bit chunks
7  and each of them represents a whole packet
8
9  Inside each chunk:
10 * an 10-bit address, which is the ctrl address of the first segment of the packet
11
12 The same packet in cmu (that needs 2 block) would be:
13 Address 0: 10'b1
14 */
15
16 /*
17 The voq is implemented as a ring buffer. there are #egress voqs in each ingress.
18 Each voq can contain at most 1024 packets. (indexed by the lower 10 bits of the
19   memory)
20 The 2 higher bits represents which voq it is.
21
22   [11, 10]      [9,8,7,6,5,4,3,2,1,0]
23   ----voq_idx----addr of 1st seg of packet-----
24 */
25
26
27 module vmu #(
28     parameter PACKET_CNT = 1024, /* How many packets can there be in each VOQ, 1024 by
29       default */
30     parameter EGRESS_CNT = 4 /* How many egress there are; which is also how many voqs
31       there are. */
32 ) (
33     input logic clk,
34     input logic voq_enqueue_en,
35     input logic [ $clog2 ( EGRESS_CNT ) - 1 : 0 ] voq_enqueue_sel,
36     input logic voq_dequeue_en,
37     input logic [ $clog2 ( EGRESS_CNT ) - 1 : 0 ] voq_dequeue_sel,
38     input logic [ $clog2 ( PACKET_CNT ) - 1 : 0
39       ] meta_in, // The address to find the first address of the packet
40
41     /* TODO: How many bits for meta_out? */
42     output logic [ $clog2 ( PACKET_CNT ) - 1 : 0 ] meta_out, // The content
43       (first addr of the packet) saved for the dequeue packet
44     output logic [ EGRESS_CNT - 1 : 0 ] is_empty, // For scheduler
45     output logic [ EGRESS_CNT - 1 : 0
46       ] is_full // For potential packet drop. If is_full, then drop the current
47 );
48
49     logic [ $clog2 ( PACKET_CNT ) + 1 : 0 ] start_idx [ 3 : 0
50       ]; // first element
51     logic [ $clog2 ( PACKET_CNT ) + 1 : 0 ] end_idx [ 3 : 0
52       ]; // one pass the last element
```

```

45 logic [ $clog2 ( EGRESS_CNT ) : 0 ] i;
46 logic [ $clog2 ( EGRESS_CNT ) -1 : 0 ] _i;
47
48
49 always_comb begin
50     for (i = 0; i < EGRESS_CNT; i = i + 1) begin
51         _i = i [ $clog2 ( EGRESS_CNT ) -1 : 0 ];
52         is_empty [ _i ] = (start_idx [ _i ] == end_idx [ _i ]);
53         is_full [ _i ] = (start_idx [ _i ] == ((end_idx [ _i
54             ] == PACKET_CNT - 1) ? 0 :end_idx [ _i ] + 1));
55     end
56 end
57
58 always @(posedge clk) begin
59     if (voq_enqueue_en && !is_full [ voq_enqueue_sel ]) begin
60         end_idx [ voq_enqueue_sel ] <= (end_idx [ voq_enqueue_sel
61             ] != PACKET_CNT - 1) ? end_idx [ voq_enqueue_sel ] + 1 :0;
62     end
63
64     if (voq_dequeue_en && !is_empty [ voq_dequeue_sel ]) begin
65         start_idx [ voq_dequeue_sel ] <= (start_idx [ voq_dequeue_sel
66             ] != 0) ? start_idx [ voq_dequeue_sel ] + 1 :PACKET_CNT - 1;
67     end
68
69     simple_dual_port_mem #(.MEM_SIZE(PACKET_CNT * EGRESS_CNT),
70         .DATA_WIDTH($clog2(PACKET_CNT))) vmem
71     (
72         .clk(clk),
73         .ra(start_idx [ voq_dequeue_sel ]), .wa(end_idx [ voq_enqueue_sel
74             ]),
75         .d(meta_in), .q(meta_out),
76         .write(voq_enqueue_en)
77     );
78 endmodule

```