DaFPGA Switch

Project Report

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

Contents

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.

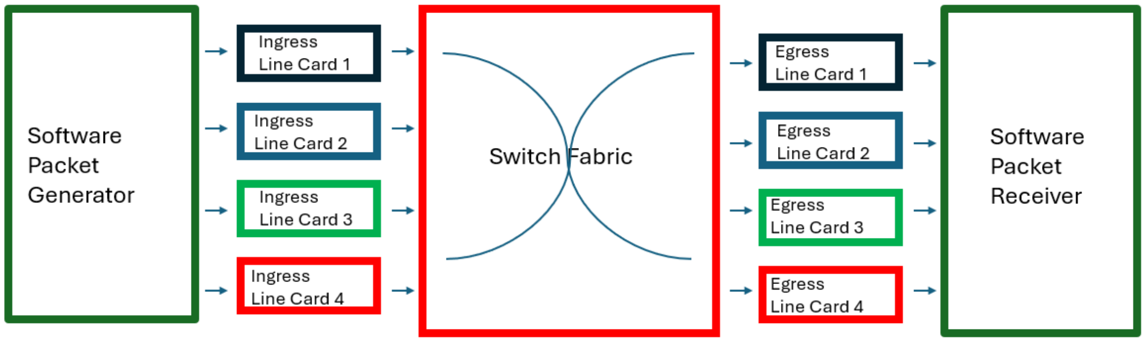


Figure 1: Block Diagram of the whole system, with 2 ingresses and 2 egresses as an example.

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 dequques 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.

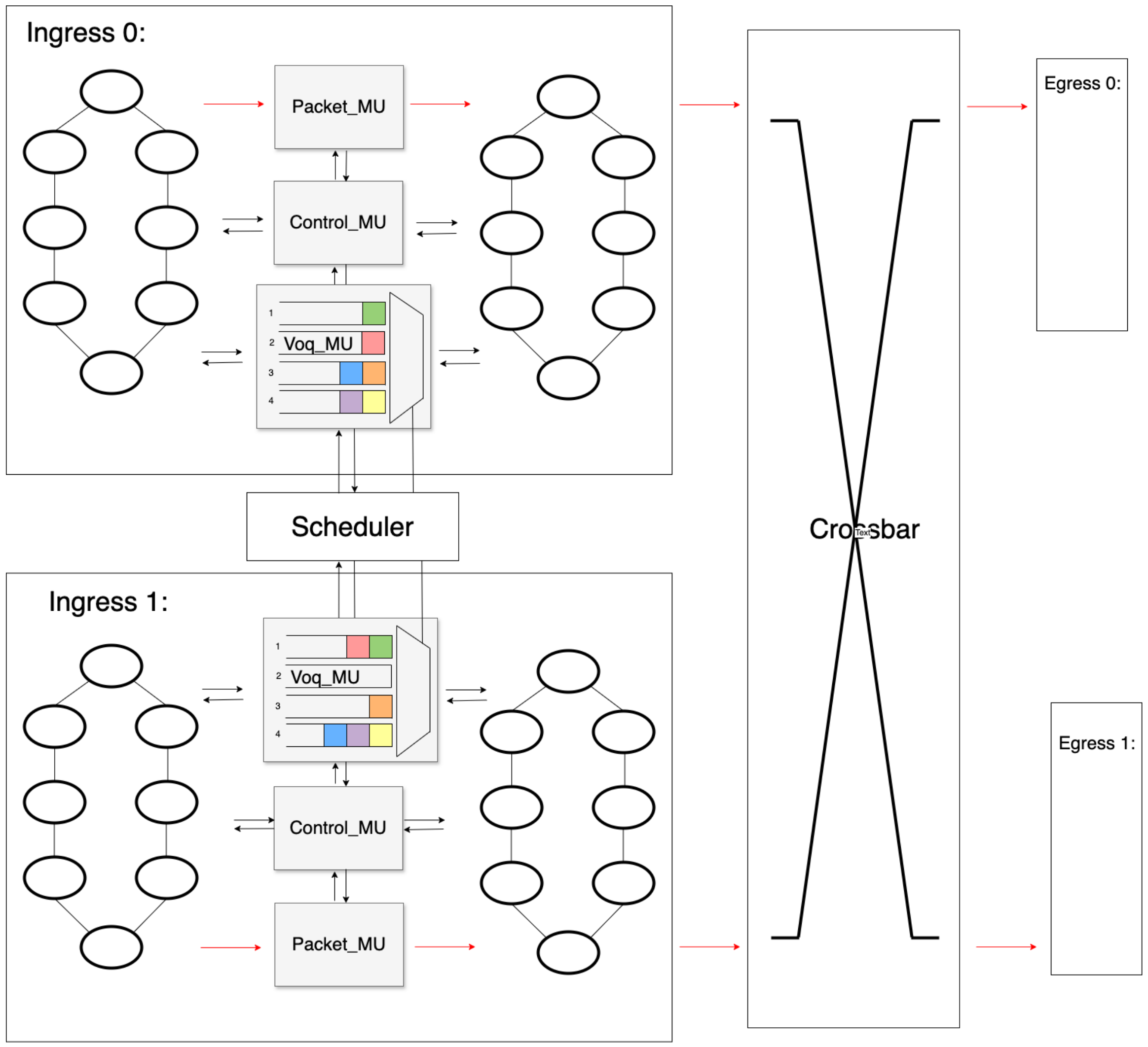


Figure 2: Block Diagram of the whole system.

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.

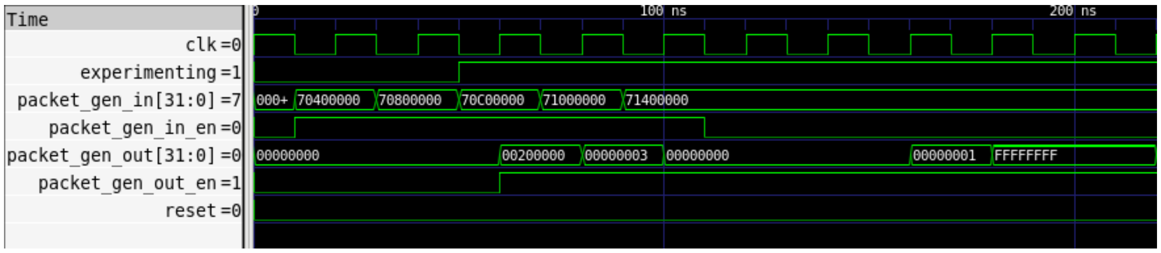


Figure 5: Hardware generating packets from software instructions

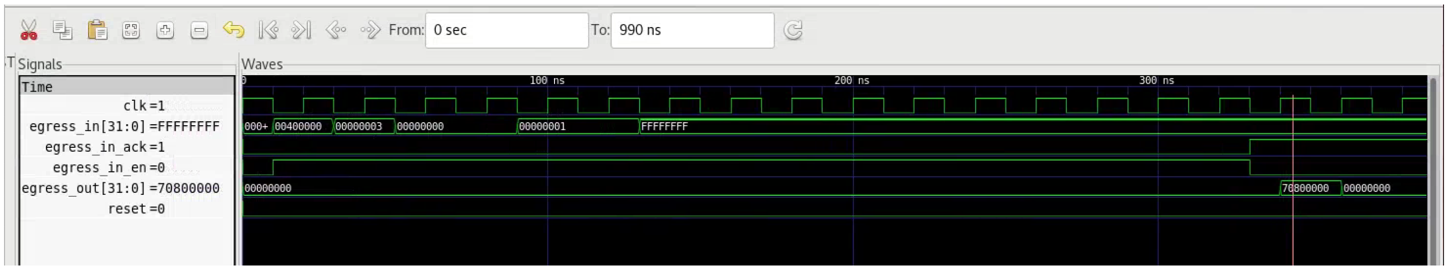


Figure 6: Packet to Meta conversion

In Figure 5, the verilator 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 is 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:

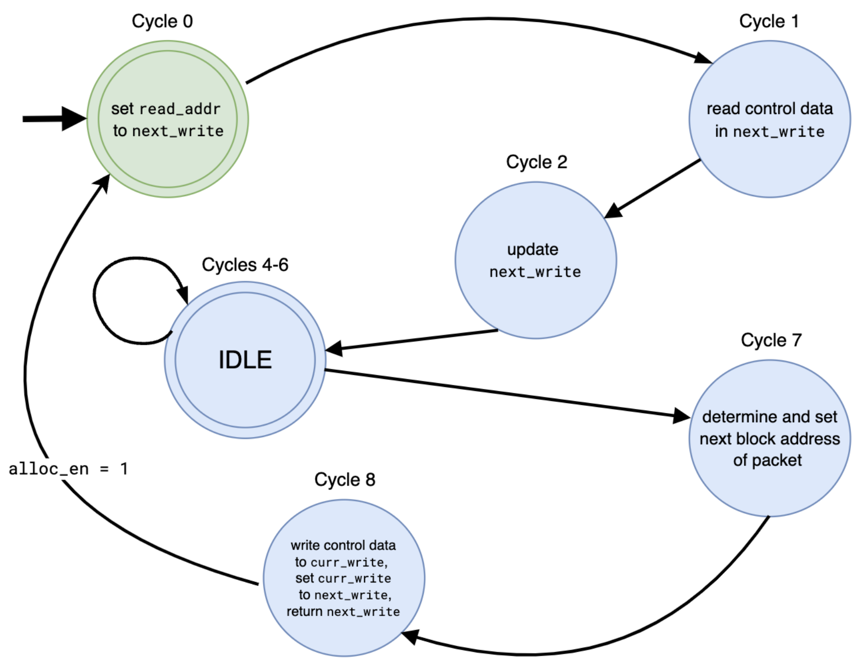
```
70800000 → (01 11 000010 0000000000000000000000)
```

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)

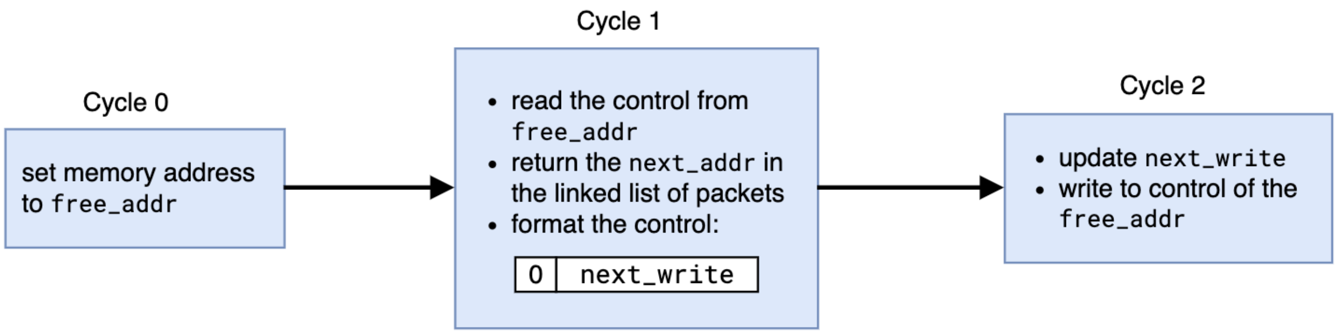
There are four ingress ports. Each ingress has a Control Memory (CMEM), which is solely managed by the Control Memory Management Unit (CMU). CMEM is modeled as a true dual-port memory. In other words, both ports of the CMEM are able to handle read and write requests. Due to the specifications of the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


Cycle 0
set read_addr to next_write
Cycle 1
read control data in next_write
Cycle 2
update next_write
Cycles 4-6
IDLE
Cycle 7
determine and set next block address of packet
Cycle 8
write control data to curr_write, set curr_write to next_write, return next_write
alloc_en = 1

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.


Cycle 0
set memory address to free_addr
Cycle 1
read the control from free_addr
return the next_addr in the linked list of packets
format the control:
0
next_write
Cycle 2
update next_write
write to control of the free_addr

The output state machine is triggered by a `free_en` signal from the ingress and it takes a `free\_addr` as an input. The control of the `free_addr` is read to determine if this address starts a chain. If so, the next address in the chain will be returned to ingress to continue with the next packet segment in the following heartbeat. `next_write` adopts the value of the `free_addr`.

Finally, we have CMU's test bench for your reference. You can see that we successfully get a free block/free an occupied block and get the next linked packet.

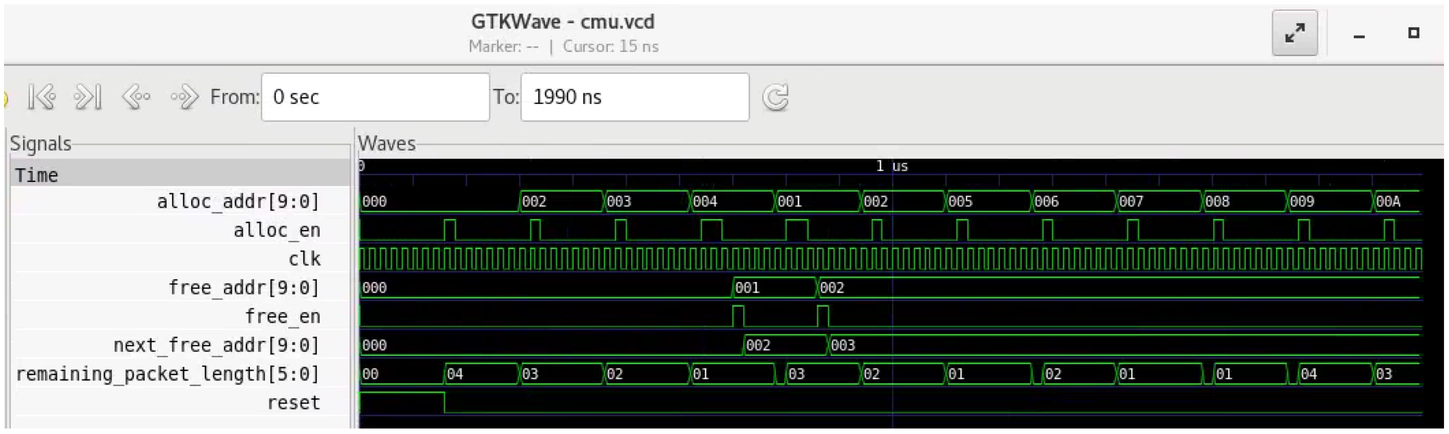
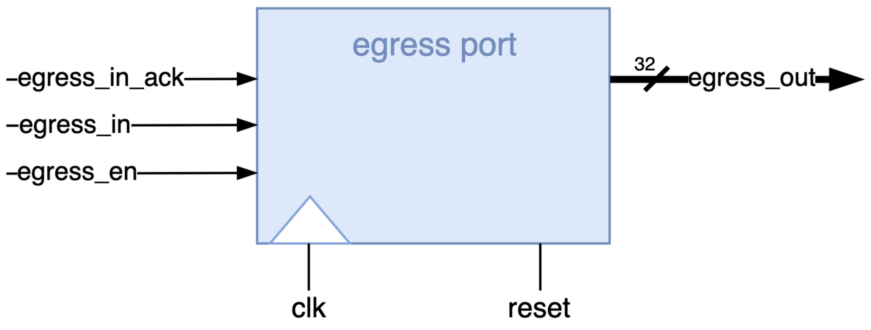


Figure 7: CMU's testbench.

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

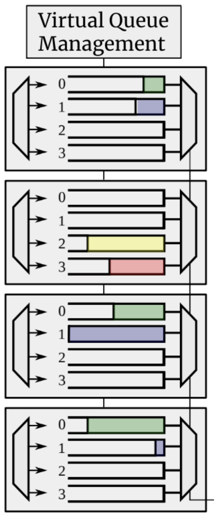


Figure 8: Virtual Output Queues of four input modules. Each of the queues inside one module corresponds to one output module.

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.

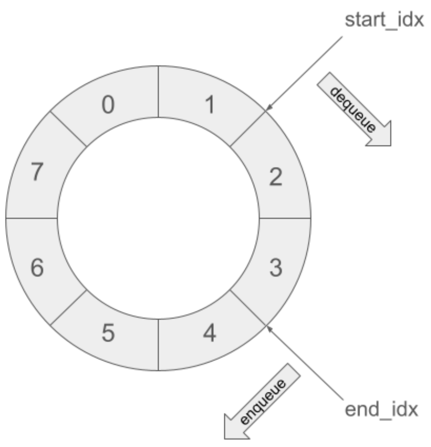


Figure 9: Illustration of the Ring buffer.

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-cyle heartbeat.

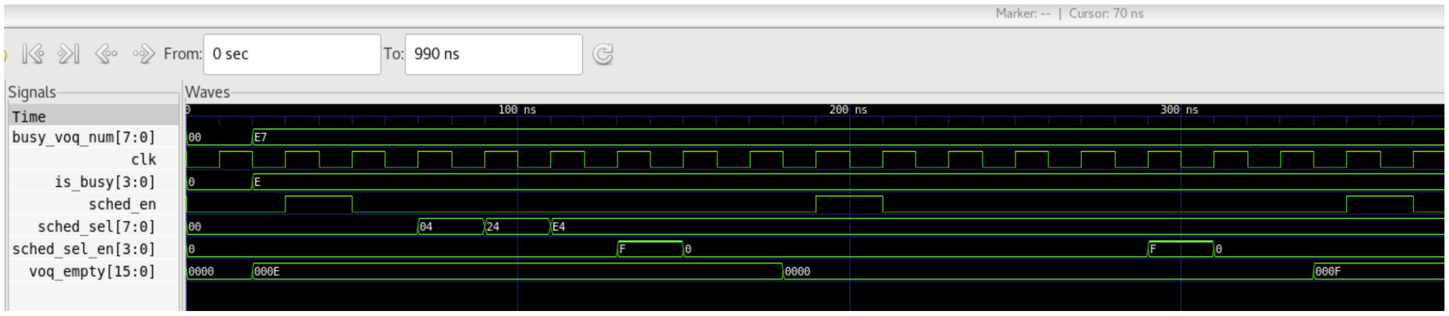


Figure 10: Verilator testbench for Scheduler

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

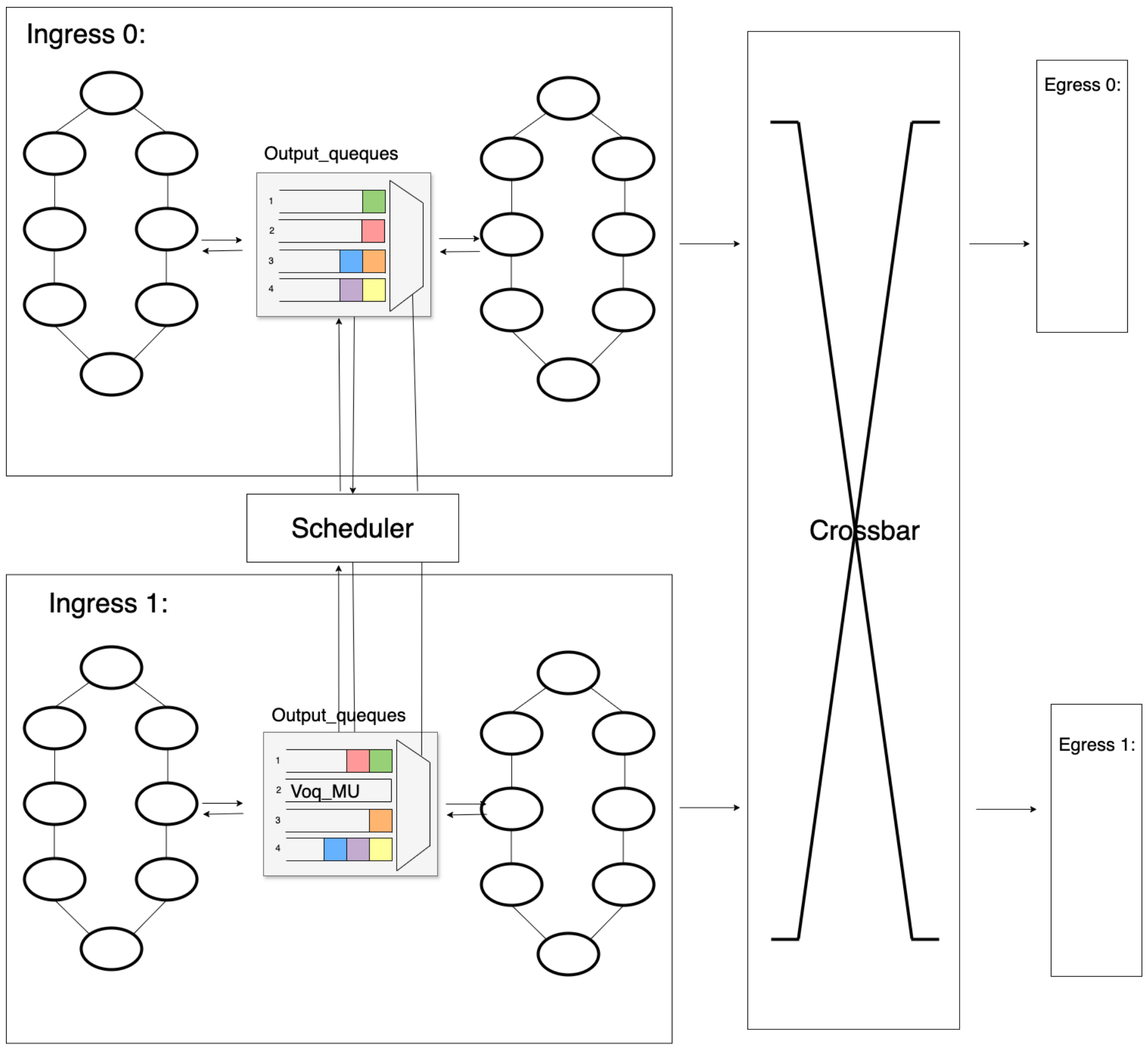


Figure 11: Block diagram of Simple Switch system

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.

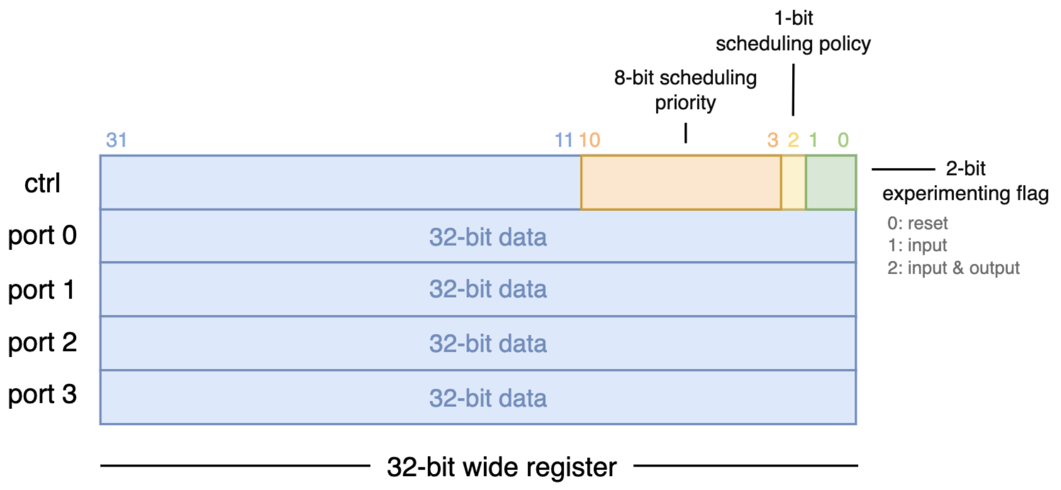


Figure 12: We use 5 32-bit registers for SW-HW communication. For the ctrl bit, the lowest two digits are for controlling the state of our experiment: 0 means we're resetting everything, 1 means that our switch only accepts input but does not output (so that we can create congestion), and the rest of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

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

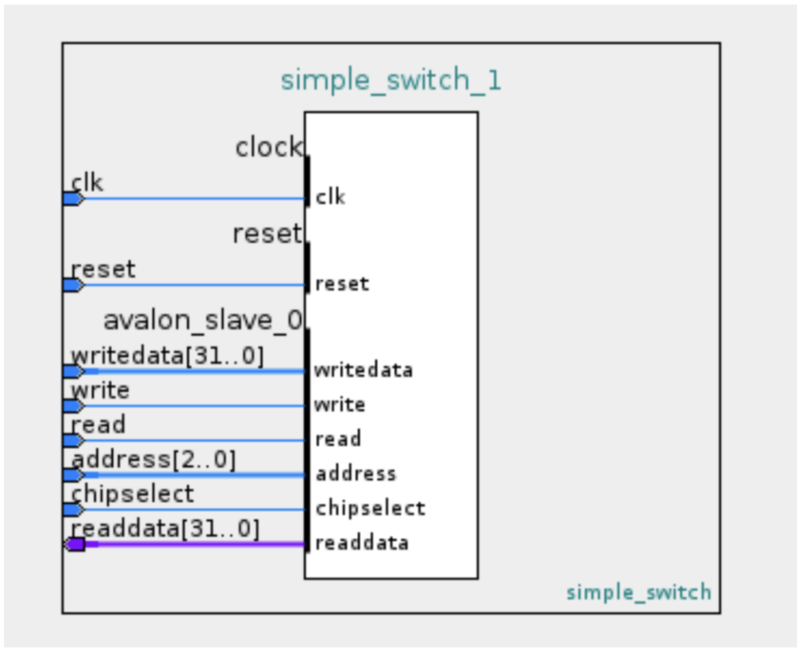


Figure 13: The simple switch Avalon bus slave.

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()`:

```c
// Write to HW
if (ioctl(simple_switch_fd, SIMPLE_WRITE_PACKET, pkt_meta) < 0) {
    perror("Failed to send packet");
    return;
}
// Read from HW
if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_0, &rcvd_pkt_meta) < 0) {
    perror("ioctl() read packet failed");
    close(simple_switch_fd);
    return -1;
}
```

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 generates 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 will 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

```

module crossbar #(
    parameter DATA_WIDTH = 32, // For testing
    parameter EGRESS_CNT = 4
) (
    input clk,
    input logic  [ $clog2 ( EGRESS_CNT ) * EGRESS_CNT -1 : 0 ] sched_sel,
    input logic  [ EGRESS_CNT -1 : 0 ] crossbar_in_en,
    input logic  [ DATA_WIDTH * EGRESS_CNT -1 : 0 ] crossbar_in,

    output logic  [ EGRESS_CNT -1 : 0 ] crossbar_out_en,
    output logic  [ DATA_WIDTH * EGRESS_CNT -1 : 0 ] crossbar_out
);

  always_ff @(posedge clk) begin
    crossbar_out_en [ sched_sel [ 0 * $clog2 ( EGRESS_CNT ) + :
        $clog2 ( EGRESS_CNT ) ] ] <= crossbar_in_en [ 0 ];
    crossbar_out [ ( sched_sel [ 0 * $clog2 ( EGRESS_CNT ) + :
        $clog2 ( EGRESS_CNT ) ] ) * DATA_WIDTH + : DATA_WIDTH
        ] <= crossbar_in [ 0 * DATA_WIDTH + : DATA_WIDTH ];
    crossbar_out_en [ sched_sel [ 1 * $clog2 ( EGRESS_CNT ) + :
        $clog2 ( EGRESS_CNT ) ] ] <= crossbar_in_en [ 1 ];
    crossbar_out [ ( sched_sel [ 1 * $clog2 ( EGRESS_CNT ) + :
        $clog2 ( EGRESS_CNT ) ] ) * DATA_WIDTH + : DATA_WIDTH
        ] <= crossbar_in [ 1 * DATA_WIDTH + : DATA_WIDTH ];
    crossbar_out_en [ sched_sel [ 2 * $clog2 ( EGRESS_CNT ) + :
        $clog2 ( EGRESS_CNT ) ] ] <= crossbar_in_en [ 2 ];
    crossbar_out [ ( sched_sel [ 2 * $clog2 ( EGRESS_CNT ) + :
        $clog2 ( EGRESS_CNT ) ] ) * DATA_WIDTH + : DATA_WIDTH
        ] <= crossbar_in [ 2 * DATA_WIDTH + : DATA_WIDTH ];
    crossbar_out_en [ sched_sel [ 3 * $clog2 ( EGRESS_CNT ) + :
        $clog2 ( EGRESS_CNT ) ] ] <= crossbar_in_en [ 3 ];
    crossbar_out [ ( sched_sel [ 3 * $clog2 ( EGRESS_CNT ) + :
        $clog2 ( EGRESS_CNT ) ] ) * DATA_WIDTH + : DATA_WIDTH
        ] <= crossbar_in [ 3 * DATA_WIDTH + : DATA_WIDTH ];

end

endmodule
```

12.1.2 Egress

```
module egress #(
    parameter PACKET_CNT = 1024,
    BLOCK_SIZE = 32,
    META_WIDTH = 32
) (
    input logic clk,
    input logic reset,

    // From crossbar
    input logic   [ META_WIDTH  -1  :  0  ] egress_in,
    input logic                 egress_in_en,

    // From interface
    input logic egress_in_ack,

    // To interface
    output logic   [  31   :   0   ] egress_out
);

  logic   [  $clog2  (  PACKET_CNT  )  -1  :  0  ] start_idx;
  logic   [  $clog2  (  PACKET_CNT  )  -1  :  0  ] end_idx;

    always @(posedge clk) begin
        if (reset) begin
            start_idx <= 0;
            end_idx <= 0;
        end
        if (egress_in_en) begin
            end_idx <= (end_idx != 1023) ? end_idx + 1 :0;
        end

        if (egress_in_ack) begin
            start_idx <= (start_idx != 1023) ? start_idx + 1 :0;
        end
    end

  simple_dual_port_mem #(
      .MEM_SIZE (PACKET_CNT),
      .DATA_WIDTH(32)
  ) meta_mem (
      .clk(clk),
      .ra(start_idx),
      .wa(end_idx),
      .d(egress_in),
      .q(egress_out),
      .write(egress_in_en)
  );

endmodule
```

12.1.3 Ingress

```


/* verilator lint_off UNUSED */
module ingress #(
    parameter PACKET_CNT = 1024,
    BLOCK_SIZE = 32,
    META_WIDTH = 32
) (
    input logic                 clk,
    input logic                 reset,

    input logic   [ META_WIDTH  -1  :  0  ] ingress_in,
    input logic                 ingress_in_en,
    input logic                 experimenting,
    input logic                 sched_en,
    input logic   [ 1  :  0  ]         sched_sel,
    input logic   [ 10  :  0  ]        time_stamp,

        // To crossbar
    output logic   [ 31  :  0  ]       ingress_out,

    // To scheduler
    output logic   [ 3  :  0  ]        is_empty
);
  logic   [ 1  :  0  ] port_num;
  logic   [ 3  :  0  ] is_full;
  logic   [ 31  :  0  ] meta_out;
  assign port_num = ingress_in  [ 29  :  28  ];
  assign ingress_out = {meta_out  [ 31  :  11  ], time_stamp};

        vmu #(.PACKET_CNT(1024), .EGRESS_CNT(4)) voq_mu
        (
                // Input
                .clk(clk), .reset(reset),
                .voq_enqueue_en(ingress_in_en && !is_full  [  port_num
                    ]), .voq_enqueue_sel(port_num),
                .voq_dequeue_en(sched_en && !is_empty  [  sched_sel
                    ]), .voq_dequeue_sel(sched_sel),
                .meta_in({ingress_in  [  31  :  22  ],time_stamp,ingress_in  [  10  :  0
                    ]}),
    .time_stamp(time_stamp),

                // Output
                .meta_out(meta_out),
                .is_empty(is_empty), .is_full(is_full)
        );

endmodule
```

12.1.4 Virtual Output Queues

```

module pick_voq (
  input logic   [ 1 : 0 ] start_voq_num, // the idx'th voq has the highest prio to be selected.
  input logic   [ 3 : 0 ] voq_empty, // Which egress/voq is empty.
  input logic   [ 7 : 0 ] prio,
  input logic policy,
  input logic   [ 3 : 0 ] voq_picked, // Which egress/voq is picked.
  output logic no_available_voq, // either all empty, or non-empty egress is taken.
  output logic   [ 1 : 0 ] voq_to_pick
);
  always_comb begin
    if (voq_empty == 4'b1111) begin
      voq_to_pick = start_voq_num;
      no_available_voq = 1'b1;
    end else if (policy == 0) begin
      if (voq_empty [ start_voq_num ] == 1'b0 && voq_picked [ start_voq_num ] == 1'b0) begin
        voq_to_pick = start_voq_num; // Save the index of the first non-zero bit
        no_available_voq = 1'b0;
      end else if (voq_empty [ start_voq_num + 1 ] == 1'b0 && voq_picked [ start_voq_num + 1 ] == 1'b0) begin
        voq_to_pick = start_voq_num + 1;
        no_available_voq = 1'b0;
      end else if (voq_empty [ start_voq_num + 2 ] == 1'b0 && voq_picked [ start_voq_num + 2 ] == 1'b0) begin
        voq_to_pick = start_voq_num + 2;
        no_available_voq = 1'b0;
      end else if (voq_empty [ start_voq_num + 3 ] == 1'b0 && voq_picked [ start_voq_num + 3 ] == 1'b0) begin
        voq_to_pick = start_voq_num + 3;
        no_available_voq = 1'b0;
      end else begin
        voq_to_pick = start_voq_num;
        no_available_voq = 1'b1;
      end
    end else begin
      if (voq_empty [ prio [ 7 : 6 ] ] == 1'b0 && voq_picked [ prio [ 7 : 6 ] ] == 1'b0) begin
        voq_to_pick = prio [ 7 : 6 ]; // Save the index of the first non-zero bit
        no_available_voq = 1'b0;
      end else if (voq_empty [ prio [ 5 : 4 ] ] == 1'b0 && voq_picked [ prio [ 5 : 4 ] ] == 1'b0) begin
        voq_to_pick = prio [ 5 : 4 ];
        no_available_voq = 1'b0;
      end else if (voq_empty [ prio [ 3 : 2 ] ] == 1'b0 && voq_picked [ prio [ 3 : 2 ] ] == 1'b0) begin
        voq_to_pick = prio [ 3 : 2 ];
        no_available_voq = 1'b0;
      end else if (voq_empty [ prio [ 1 : 0 ] ] == 1'b0 && voq_picked [ prio [ 1 : 0 ] ] == 1'b0) begin
        voq_to_pick = prio [ 1 : 0 ];
        no_available_voq = 1'b0;
      end else begin
        voq_to_pick = start_voq_num;
        no_available_voq = 1'b1;
      end
    end
  end

endmodule
```

12.1.5 Scheduler

```
module sched (
    input logic clk,
    input logic sched_en,
    input logic  [ 3 : 0 ] is_busy,
    input logic  [ 7 : 0 ] busy_voq_num,
    input logic  [ 15 : 0 ] voq_empty,
    input logic policy, // We have doubly RR, or priority based, can be controlled by software
    input logic  [ 7 : 0 ] prio,
    output logic  [ 3 : 0 ] sched_sel_en, // passed by to ingress, to know which ingress should dequeue
    output logic  [ 7 : 0 ] sched_sel // passed by to ingress, to know which voq to dequeue
);

  /*
  Some design choices:
  * Do we want the time to return a scheduling decision to be deterministic or random (btw 1 and 4)
  * Should each busy port to just start transmitting without waiting for the scheduling decision
  * Should the schduler decision the scheduling decision for this cycle or next cycle
  * RR on the ingress or egress side, or both
  * Should we try to do all combinator?
  * Do we use 1 voq_to_pick or 4 voq_to_pick? (1 since it's going to take 4 cycles anyway)
  */

  /*
  Some principles:
  * We don't want egress 1 to always recv packet from ingress 0; we don't want ingress 0 to always send to egress 2
  * nested loop fully in combinator logic is too expensive; instead, we do the outer for loop sequentially, and the inner 4 loop in comb logic
  RR policy:
    * ingress RR: Start_ingress_idx proceed in each cycle of ASSIGN_NEW (or one pass who-ever gets to select first in this cycle)
    * egress RR: Each start_voq_num is first_non_empty_num of this cycle + 1;
    * fewer first: Prioritze queue with only one non-empty voq.
  */

  /*
  Notice:
  * Beware of the possiblity of input (voq_empty for example) change during the process.
    - Possibly, use some local variables to save the inputs.
    - Or, let the ingress be in charge of only updating the signal when sched_en.
  * Busy ports need to be handled first.
  * All the data need to be prepared before sched_enable. So at T-1 prepare input, and at T sched_enable.
  */
```

```

// If the scheduler is in the process of assigning new packet
// 0: not assigning; 1~4: assigning. 5 enable 6 is /enable
logic   [  2  :  0  ] assigning_new;

logic   [  3  :  0  ] ingress_enable; // the enable signal ready to be passed to
    sched_sel_en when ingress_done = 4'b1111

// For RR
logic   [  1  :  0
    ] start_ingress_idx; // Which ingress has the highest priority in this cycle
logic   [  7  :  0  ] start_voq_num; // Which egress has the highest priority in
    this cycle, for each ingress.

logic   [  1  :  0  ] curr_ingress_idx; // Current ingress
logic   [  2  :  0  ] curr_in_2;
logic   [  3  :  0  ] curr_in_4;
logic   [  3  :  0  ] voq_picked; // is the voq/egress picked?
logic no_available_voq; // Is the voqs/egress of the current ingress all
    empty/occupied by other?
logic   [  1  :  0
    ] voq_to_pick; // What is the voq_to_pick for the current ingress
logic   [  3  :  0  ] busy_egress_mask;

logic   [  2  :  0  ] i;
logic   [  1  :  0  ] busy_port;

always_comb begin
  busy_egress_mask = 0; // This is important: busy_egress_mask need a way to start
      with all unoccupied.
  for (i = 0; i < 4; i = i + 1) begin
    if (is_busy  [  i  [  1  :  0  ]  ] == 1'b1) begin
      busy_egress_mask  [  busy_voq_num  [  (  i    <<    1)    +  :    2  ]
          ] = 1'b1;
    end
  end
  curr_in_2 = {1'b0, curr_ingress_idx} << 1;
  curr_in_4 = {2'b0, curr_ingress_idx} << 2; // * 4 is << 2
end

initial begin
  start_ingress_idx = 0;
  start_voq_num = 0;
end

always_ff @(posedge clk) begin

  if (sched_en) begin
    // If we begin to schedule
    // reset sched_sel_en
    sched_sel_en <= 0;
    // all the busy ingress ports are automatically assigned.
    ingress_enable <= 0;
    // start to assign ports for non-empty
```

```verilog
      assigning_new <= 1;
      voq_picked <= busy_egress_mask;
      curr_ingress_idx <= start_ingress_idx; // Start with start_ingress_idx
    end else if (assigning_new == 5) begin // alternatively, if we manage to go back to start_ingress_idx
      // If all are assigned, we're going start enabling
      sched_sel_en <= ingress_enable;
      // Nex time it should start with another index.
      start_ingress_idx <= (start_ingress_idx == 3) ? 0 :start_ingress_idx + 1;
      assigning_new <= 6;
    end else if (assigning_new == 6) begin
      sched_sel_en <= 0;
    end else if (assigning_new >= 1 && assigning_new <= 4) begin
      curr_ingress_idx <= (curr_ingress_idx == 3) ? 0 :curr_ingress_idx + 1;
      assigning_new <= assigning_new + 1;
      if (!is_busy  [  curr_ingress_idx  ]) begin
        if (!no_available_voq) begin
          ingress_enable  [  curr_ingress_idx  ] <= 1'b1;
          voq_picked  [  voq_to_pick  ] <= 1'b1;
          sched_sel  [  curr_in_2     +  :     2  ] <= voq_to_pick;
          start_voq_num  [  curr_in_2     +  :     2  ] <=
            (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.
        end
      end else begin
        busy_port <= busy_voq_num  [  curr_in_2     +  :     2  ];
        ingress_enable  [  curr_ingress_idx  ] <= 1'b1;
        sched_sel  [  curr_in_2     +  :     2  ] <= busy_port;
        voq_picked  [  busy_port  ] <= 1'b1;
      end
    end
  end

  // pick_voq will pick return the current ingress's first non empty voq to dequeue from.
  pick_voq pv (
    .start_voq_num(start_voq_num  [  curr_in_2     +  :     2  ]),
    .voq_empty(voq_empty  [  curr_in_4     +  :     4  ]),
    .voq_picked(voq_picked),
    .no_available_voq(no_available_voq),
    .voq_to_pick(voq_to_pick),
    .policy(policy),
    .prio(prio)
  );
endmodule
```

12.1.6 Simple Dual Port

```
module simple_dual_port_mem #(
   parameter MEM_SIZE = 1024, /* How many addresses of memory in total, 1024 by
      default */
   parameter DATA_WIDTH = 32 /* How many bit of data per cycle, 32 by default */
) (
   input logic clk,
   input logic   [  $clog2  (  MEM_SIZE  )      -      1   :   0  ] ra, wa, /* Address */
   input logic   [  DATA_WIDTH      -      1   :   0  ] d,             /* input data */
   input logic write,                                           /* Write enable */
   output logic   [  DATA_WIDTH      -      1   :   0  ] q             /* output data */
);
  logic   [  DATA_WIDTH  -1   :   0  ] mem   [  MEM_SIZE      -      1   :   0   ];

  always_ff @(posedge clk) begin
    if (write) begin
      mem   [  wa   ] <= d;
         end
    q <= mem   [  ra   ];
  end

endmodule
```

12.1.7 Simple Interface

```


/* verilator lint_off UNUSED */
module simple_interface(
   input logic clk,

   // From sw
   input logic     reset,
   input logic   [  31   :  0   ] writedata,
   input logic     write,
   input logic     read,
   input logic   [     2   :  0   ] address,
   input logic     chipselect,

   // From egress
   input logic   [  31   :  0   ] interface_in_0,
   input logic   [  31   :  0   ] interface_in_1,
   input logic   [  31   :  0   ] interface_in_2,
   input logic   [  31   :  0   ] interface_in_3,

   // To sw
   output logic   [  31   :  0   ] readdata,

   // To packet_val/ingress
   output logic   [  3   :  0   ] interface_out_en,
   output logic   [  31   :  0   ] interface_out,

   // Experimenting
   output logic experimenting,
   output logic simple_reset,
   // output logic send_only,

   // Special case: Because we're polling but not handling interrupt
   // we need to acknowledge that this metadata is consumed by the software.
   // This is the only ack in our program.
   output logic   [  3   :  0   ] interface_out_ack,

   output logic sched_policy,
   output logic   [  7   :  0   ] sched_prio
);

  logic   [  31   :  0   ] ctrl;
  logic prev_read;

  always_comb begin
    experimenting = (ctrl  [  1   :  0   ] == 2);
    // send_only = (ctrl == 1);
    simple_reset = (ctrl  [  1   :  0   ] == 0) || reset;
    sched_policy = ctrl  [  2   ];
    sched_prio = ctrl  [  10   :  3   ];
  end
```

```

always_ff @(posedge clk) begin
  prev_read <= read;
  if (reset) begin
    ctrl <= 32'h0;
    readdata <= 32'h0;
    interface_out_en <= 0;
    interface_out_ack <= 0;
    interface_out <= 0;
  end else begin
    if (chipselect && write) begin
      case (address)
        3'h0: begin
          ctrl <= writedata;
        end
        3'h1: begin
          interface_out_en  [ 0 ] <= 1;
          interface_out <= writedata;
        end
        3'h2: begin
          interface_out_en  [ 1 ] <= 1;
          interface_out <= writedata;
        end
        3'h3: begin
          interface_out_en  [ 2 ] <= 1;
          interface_out <= writedata;
        end
        3'h4: begin
          interface_out_en  [ 3 ] <= 1;
          interface_out <= writedata;
        end
        default: begin
        end
      endcase
    end else begin
      interface_out_en  [ 0 ] <= 0;
      interface_out_en  [ 1 ] <= 0;
      interface_out_en  [ 2 ] <= 0;
      interface_out_en  [ 3 ] <= 0;
    end
    if (chipselect && read && prev_read == 0) begin // Need rising edge detection?
      case (address)
        3'h1: begin
          readdata <= interface_in_0;
          interface_out_ack  [ 0 ] <= 1;
        end
        3'h2: begin
          readdata <= interface_in_1;
          interface_out_ack  [ 1 ] <= 1;
        end
        3'h3: begin
          readdata <= interface_in_2;
          interface_out_ack  [ 2 ] <= 1;
          end
          3'h4: begin
            readdata <= interface_in_3;
            interface_out_ack  [  3  ] <= 1;
          end
          default: begin
          end
        endcase
      end else begin
        interface_out_ack  [  0  ] <= 0;
        interface_out_ack  [  1  ] <= 0;
        interface_out_ack  [  2  ] <= 0;
        interface_out_ack  [  3  ] <= 0;
      end
    end
  end

endmodule
```

12.1.8 Simple Switch

```

module simple_switch
(
    input logic clk,
    input logic reset,
    input logic   [  31  :  0  ] writedata,
    input logic      write,
    input logic      read,
    input logic   [  2  :  0  ] address,
    input logic      chipselect,

    output logic   [  31  :  0  ] readdata
);
    /* verilator lint_off UNUSED */
    logic   [  3  :  0  ] meta_in_en;
    logic   [  3  :  0  ] packet_en;
    logic   [  3  :  0  ] meta_out_ack;
    logic   [  7  :  0  ] sched_sel;
    logic   [  3  :  0  ] sched_sel_en;

    logic   [  31  :  0  ] meta_in;
    // logic [127:0] meta_out;
    logic   [  31  :  0  ] meta_out   [  4  ];
    logic   [  127  :  0  ] packet;
    logic   [  127  :  0  ] packet_out;
    logic   [  3  :  0  ] packet_out_en;
    logic experimenting;
    logic   [  15  :  0  ] empty;
    logic   [  10  :  0  ] counter;
    logic   [  3  :  0  ] cycle;
    logic sched_en;
    logic simple_reset;
    logic global_reset;
    logic sched_policy;
    logic   [  7  :  0  ] sched_prio;

    assign global_reset = simple_reset || reset;

    always_ff @(posedge clk) begin
        cycle <= (cycle == 15) ? 0 :cycle + 1;
        if (experimenting) begin
            if (cycle == 0) begin
                counter <= counter + 1;
                sched_en <= 1;
            end else if (cycle == 1) begin
                sched_en <= 0;
            end
        end else begin
           counter <= 0;
        end
    end

    ingress ingress_0 (
        // Input
        .clk(clk), .reset(global_reset),
        .ingress_in_en(meta_in_en  [  0  ]), .experimenting(experimenting),
        .ingress_in(meta_in),
        .sched_en(sched_sel_en  [  0  ] && experimenting),
        .sched_sel(sched_sel  [  1  :  0  ]),
        .time_stamp(counter),

        // Output
        // .ingress_out_en(packet_en[0]),
        .ingress_out(packet  [  31  :  0  ]),
        .is_empty(empty  [  3  :  0  ])
    );

    ingress ingress_1 (
        // Input
        .clk(clk), .reset(global_reset),
        .ingress_in_en(meta_in_en  [  1  ]), .experimenting(experimenting),
        .ingress_in(meta_in),
        .sched_en(sched_sel_en  [  1  ] && experimenting),
        .sched_sel(sched_sel  [  3  :  2  ]),
        .time_stamp(counter),

        // Output
        // .ingress_out_en(packet_en[1]),
        .ingress_out(packet  [  63  :  32  ]),
        .is_empty(empty  [  7  :  4  ])
    );

    ingress ingress_2 (
        // Input
        .clk(clk), .reset(global_reset),
        .experimenting(experimenting),
        .ingress_in_en(meta_in_en  [  2  ]),
        .ingress_in(meta_in),
        .sched_en(sched_sel_en  [  2  ] && experimenting),
        .sched_sel(sched_sel  [  5  :  4  ]),
        .time_stamp(counter),


        // Output
        // .ingress_out_en(packet_en[2]),
        .ingress_out(packet  [  95  :  64  ]),
        .is_empty(empty  [  11  :  8  ])
    );

    ingress ingress_3 (
        // Input
        .clk(clk), .reset(global_reset),
        .ingress_in_en(meta_in_en  [  3  ]), .experimenting(experimenting),
        .ingress_in(meta_in),
        .sched_en(sched_sel_en  [ 3 ] && experimenting),
        .sched_sel(sched_sel  [ 7 : 6 ]),
        .time_stamp(counter),

        // Output
        // .ingress_out_en(packet_en[3]),
        .ingress_out(packet  [ 127 : 96 ]),
        .is_empty(empty  [ 15 : 12 ])
    );

    egress egress_0(
        // Input
        .clk(clk), .reset(global_reset),
        .egress_in(packet_out  [ 31 : 0 ]),
        .egress_in_en(packet_out_en  [ 0 ]), .egress_in_ack(meta_out_ack  [ 0 ]),

        // Output
        .egress_out(meta_out  [ 0 ])
    );
    egress egress_1(
        // Input
        .clk(clk), .reset(global_reset),
        .egress_in(packet_out  [ 63 : 32 ]),
        .egress_in_en(packet_out_en  [ 1 ]), .egress_in_ack(meta_out_ack  [ 1 ]),

        // Output
        .egress_out(meta_out  [ 1 ])
    );
    egress egress_2 (
        // Input
        .clk(clk), .reset(global_reset),
        .egress_in(packet_out  [ 95 : 64 ]),
        .egress_in_en(packet_out_en  [ 2 ]), .egress_in_ack(meta_out_ack  [ 2 ]),

        // Output
        .egress_out(meta_out  [ 2 ])
    );
    egress egress_3 (
        // Input
        .clk(clk), .reset(global_reset),
        .egress_in(packet_out  [ 127 : 96 ]),
        .egress_in_en(packet_out_en  [ 3 ]), .egress_in_ack(meta_out_ack  [ 3 ]),

        // Output
        .egress_out(meta_out  [ 3 ])
    );

    simple_interface simple_interface (
        .clk(clk),

        // Input: sw->interface
        .reset(reset),
        .writedata(writedata),// sw->hw
        .write(write), // sw->hw
        .read(read), // hw->sw
        .address(address),
        .chipselect(chipselect),

        // Input: hw->interface
        .interface_in_0(meta_out  [  0  ]),
        .interface_in_1(meta_out  [  1  ]),
        .interface_in_2(meta_out  [  2  ]),
        .interface_in_3(meta_out  [  3  ]),

        // Output: interface->sw
        .readdata(readdata),

        // Output: interface->egress (ack)
        .interface_out_ack(meta_out_ack), // 4 bit

        // Output: To ingress/packet_val
        .interface_out_en(meta_in_en), // 4 bit
        .interface_out(meta_in),

        // Output: Ongoing experiment.
        .experimenting(experimenting),
        .simple_reset(simple_reset),

        .sched_policy(sched_policy),
        .sched_prio(sched_prio)

    );

    crossbar crossbar (
        .clk(clk),
        .sched_sel(sched_sel),
        .crossbar_in_en(sched_sel_en),
        .crossbar_in(packet),
        .crossbar_out_en(packet_out_en),
        .crossbar_out(packet_out)
    );

    sched scheduler (
        .clk(clk),
        .sched_en(sched_en),
        .is_busy(0),
        .busy_voq_num(0),
        .voq_empty(empty),
        .policy(sched_policy),
        .prio(sched_prio),

        .sched_sel_en(sched_sel_en), // passed by to ingress, to know which ingress
           should dequeue
        .sched_sel(sched_sel) // passed by to ingress, to know which voq to dequeue
    );

```

```
endmodule
```

12.1.9 Verilator Makefile

```makefile

.PHONY: lint

TOP_FILES = simple_switch.sv simple_interface.sv simple_dual_port_mem.sv crossbar.sv
    sched.sv

SVFILES = sched.sv pick_voq.sv vmu.sv simple_dual_port_mem.sv crossbar.sv
SCHED_FILES = sched.sv pick_voq.sv

# Run Verilator simulations

default:
	@echo "No target given. Try:"
	@echo ""
	@echo "make pick_voq"
	@echo "make crossbar"
	@echo "make sched.vcd"
	@echo "make vmu.vcd"
	@echo "make cmu.vcd"
	@echo "make lint"
	@echo "make simple_switch.vcd"


simple_switch.vcd :obj_dir/Vsimple_switch
		obj_dir/Vsimple_switch

pick_voq :obj_dir/Vpick_voq
	(obj_dir/Vpick_voq && echo "SUCCESS") || echo "FAILED"

crossbar: obj_dir/Vcrossbar
	(obj_dir/Vcrossbar && echo "SUCCESS") || echo "FAILED"

cmu.vcd :obj_dir/Vcmu
	obj_dir/Vcmu

sched.vcd :obj_dir/Vsched
	obj_dir/Vsched

vmu.vcd :obj_dir/Vvmu
	obj_dir/Vvmu

obj_dir/Vsimple_switch :$(TOP_FILES) verilator/simple_switch.cpp
	verilator -trace -Wall -cc $(TOP_FILES) -exe verilator/simple_switch.cpp \
		-top-module simple_switch
	cd obj_dir && make -j -f Vsimple_switch.mk

obj_dir/Vcmu :$(CMU_FILES) verilator/cmu.cpp
	verilator -trace -Wall -cc $(CMU_FILES) -exe verilator/cmu.cpp \
		-top-module cmu
	cd obj_dir && make -j -f Vcmu.mk

```

```
obj_dir/Vpick_voq :pick_voq.sv verilator/pick_voq.cpp
	verilator -Wall -cc pick_voq.sv -exe verilator/pick_voq.cpp \
		-top-module pick_voq
	cd obj_dir && make -j -f Vpick_voq.mk

obj_dir/Vcrossbar :crossbar.sv verilator/crossbar.cpp
	verilator -Wall -cc crossbar.sv -exe verilator/crossbar.cpp \
		-top-module crossbar
	cd obj_dir && make -j -f Vcrossbar.mk

obj_dir/Vsched :$(SCHED_FILES) verilator/sched.cpp
	verilator -trace -Wall -cc sched.sv pick_voq.sv -exe verilator/sched.cpp \
		-top-module sched
	cd obj_dir && make -j -f Vsched.mk

obj_dir/Vvmu :vmu.sv simple_dual_port_mem.sv verilator/vmu.cpp
	verilator -trace -Wall -cc vmu.sv simple_dual_port_mem.sv -exe
	    verilator/vmu.cpp \
		-top-module vmu
	cd obj_dir && make -j -f Vvmu.mk

lint :
	for file in $(SVFILES); do \
	verilator --lint-only -Wall $$file; done

clean :
	rm -rf obj_dir db incremental_db output_files \
	lab1.qpf lab1.qsf lab1.sdc lab1.qws c5_pin_model_dump.txt
```

12.1.10 Virtual Memory Unit

```

/*
The virtual output queue management unit:
*/

/*
voq memory is divided into 16-bit chunks
and each of them represents a whole packet

Inside each chunk:
* an 10-bit address, which is the ctrl address of the first segment of the packet

The same packet in cmu (that needs 2 block) would be:
Address 0: 10'b1
*/

/*
The voq is implemented as a ring buffer. there are #egress voqs in each ingress.
Each voq can contain at most 1024 packets. (indexexd by the lower 10 bits of the memory)
The 2 higher bits represents which voq it is.

    [11, 10]     [9,8,7,6,5,4,3,2,1,0]
 ----voq_idx-----addr of 1st seg of packet------

*/

/* verilator lint_off UNUSED */
module vmu #(
    parameter PACKET_CNT = 1024, /* How many packets can there be in each VOQ, 1024 by default */
    parameter EGRESS_CNT = 4 /* How many egress there are; which is also how many voqs there are. */
) (
    input logic clk,
    input logic reset,
    input logic voq_enqueue_en,
    input logic   [  $clog2  (  EGRESS_CNT  )  -1  :  0  ] voq_enqueue_sel,
    input logic voq_dequeue_en,
    input logic   [  $clog2  (  EGRESS_CNT  )  -1  :  0  ] voq_dequeue_sel,
    input logic   [  31  :  0  ] meta_in, // The address to find the first address of the packet
    input logic   [  10  :  0  ] time_stamp,

    /* TODO: How many bits for meta_out? */
    output logic   [  31  :  0  ] meta_out, // The content (first addr of the packet) saved for the dequeue packet
    output logic   [  EGRESS_CNT  -1  :  0  ] is_empty, // For scheduler
    output logic   [  EGRESS_CNT  -1  :  0  ] is_full // For potential packet drop. If is_full, then drop the current
);
```

```
    logic   [  $clog2  (  PACKET_CNT  )  +1  :  0  ] start_idx  [  3  :  0
        ]; // first element
    logic   [  $clog2  (  PACKET_CNT  )  +1  :  0  ] end_idx  [  3  :  0
        ]; // one pass the last element

    logic   [  $clog2  (  EGRESS_CNT  )  :  0  ] i;
    logic   [  $clog2  (  EGRESS_CNT  )  -1  :  0  ] _i;

    always_comb begin
        for (i = 0; i < EGRESS_CNT; i = i + 1) begin
            _i = i  [  $clog2  (  EGRESS_CNT  )  -1  :  0  ];
            is_empty  [  _i  ] = (start_idx  [  _i  ] == end_idx  [  _i  ]);
            is_full  [  _i  ] = (start_idx  [  _i  ] == ((end_idx  [  _i
                ] == PACKET_CNT - 1) ? 0 :end_idx  [  _i  ] + 1));
        end
    end


    always @(posedge clk) begin
        if (reset) begin
            start_idx  [  0  ] <= 0;
            start_idx  [  1  ] <= 0;
            start_idx  [  0  ] <= 0;
            start_idx  [  1  ] <= 0;
            end_idx  [  0  ] <= 0;
            end_idx  [  1  ] <= 0;
            end_idx  [  2  ] <= 0;
            end_idx  [  3  ] <= 0;
        end else begin
            if (voq_enqueue_en && !is_full  [  voq_enqueue_sel  ]) begin
                end_idx  [  voq_enqueue_sel  ] <= (end_idx  [  voq_enqueue_sel
                    ] != PACKET_CNT - 1) ? end_idx  [  voq_enqueue_sel  ] + 1 :0;
            end

            if (voq_dequeue_en && !is_empty  [  voq_dequeue_sel  ]) begin
                start_idx  [  voq_dequeue_sel  ] <= (start_idx  [  voq_dequeue_sel
                    ] != PACKET_CNT - 1) ? start_idx  [  voq_dequeue_sel  ] + 1 :0;
            end
        end
    end

        simple_dual_port_mem #(.MEM_SIZE(PACKET_CNT * EGRESS_CNT), .DATA_WIDTH(32))
            vmem
        (
                .clk(clk),
                .ra(start_idx  [  voq_dequeue_sel  ]), .wa(end_idx  [  voq_enqueue_sel
                    ]),
                .d({meta_in  [  31  :  22  ], time_stamp, meta_in  [  10  :  0
                    ]}), .q(meta_out),
                .write(voq_enqueue_en)
        );

endmodule
```

12.2 Simple Switch: Software

12.2.1 Even Load

```c
#include <stdio.h>
#include "simpleSwitch.h"
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>

#define DEVICE_PATH "/dev/simple_driver"
int simple_switch_fd;
int num_sent = 0;
int num_get = 0;

void open_simple_device()
{
   simple_switch_fd = open(DEVICE_PATH, O_RDWR);
   if (simple_switch_fd < 0) {
      perror("Failed to open simple device");
      return;
   }
}

void print_packet(void *packet_data)
{
      unsigned int packet = *((unsigned int*) packet_data);

   printf("\tmetadata: [%u | %u | %u | %u | %u ]\n",
      (packet >> 30) & 0x3, // Extract bits 31:30
      (packet >> 28) & 0x3, // Extract bits 29:28
      (packet >> 22) & 0x3F, // Extract bits 27:22
      (packet >> 11) & 0x7FF, // Extract bits 21:11
       packet & 0x7FF   // Extract bits 10:0
   );
}

int extra_time_delta(unsigned int packet)
{
   return (packet & 0x7FF) - ((packet >> 11) & 0x7FF);
}

int extra_dst_port(unsigned int packet)
{
   return (packet >> 28) & 0x3;
}

void set_ctrl_register(const packet_ctrl_t *pkt_ctrl)
{
      if (ioctl(simple_switch_fd, SIMPLE_WRITE_CTRL, pkt_ctrl) < 0) {
        perror("ioctl(SIMPLE_WRITE_CTRL) set CTRL failed\n");
        close(simple_switch_fd);
        return;
    }
}

void send_packet(const packet_meta_t *pkt_meta)
{
    usleep(5000);
    if (ioctl(simple_switch_fd, SIMPLE_WRITE_PACKET, pkt_meta) < 0) {
        perror("Failed to send packet");
        return;
    }
    num_sent++;
}


/**
 * Sets the length and time_delta fields of the packet metadata.
 * Assumes that @pkt_meta is cleared prior to calling this function.
 * @param pkt_meta Pointer to the packet metadata.
 * @param length Length of the packet, to be set in bits [27:22].
 */
void set_packet_length(packet_meta_t *pkt_meta, unsigned int length)
{
        if (length > 0x3F) {
                perror("Failed to set length\n");
                return;
        }
    // Mask and shift length @ bit pos [27:22]
    *pkt_meta |= (length & 0x3F) << 22;
}


void set_all_packet_fields(packet_meta_t *pkt_meta, unsigned int dst,
                                                              unsigned int src,
                                                                 unsigned int length)
{
        packet_meta_t pkt_tmp;
        *pkt_meta = 0;

        set_packet_length(pkt_meta, length);
        *pkt_meta = set_dst_port(*pkt_meta, dst);
        *pkt_meta = set_src_port(*pkt_meta, src);
}


int main()
{
    packet_meta_t pkt_meta, rcvd_pkt_meta;
    packet_ctrl_t pkt_ctrl;
    int total_latency = 0;

    open_simple_device();

        printf("Start: \n");
    pkt_ctrl = 0;
        set_ctrl_register(&pkt_ctrl);
        print_packet(&pkt_ctrl);
    usleep(1000);

    // Set control register (CTRL=1)
        printf("Start sending\n");
    pkt_ctrl = 1;
        set_ctrl_register(&pkt_ctrl);
        print_packet(&pkt_ctrl);
    usleep(1000);

    for (int i = 0; i < 32; i++) {
        set_all_packet_fields(&pkt_meta, (i+1)%4, i%4, 1);
                send_packet(&pkt_meta);
                print_packet(&pkt_meta);
        }

    usleep(10000);
    printf("Start recving\n");
        pkt_ctrl = 2;
        set_ctrl_register(&pkt_ctrl);
        print_packet(&pkt_ctrl);
    usleep(10000);

        printf("Requested %d packets\n", num_sent);
        for (int i = 0; i < 200; i++) {
        usleep(1000);
        if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_0, &rcvd_pkt_meta) < 0) {
                        perror("ioctl read packet failed");
                        close(simple_switch_fd);
                        return -1;
                }
        if (rcvd_pkt_meta >> 22) {
            printf("Port 0: \n");
            print_packet(&rcvd_pkt_meta);
            total_latency += extra_time_delta(rcvd_pkt_meta);
            num_get++;
        }
        usleep(1000);
        if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_1, &rcvd_pkt_meta) < 0) {
                        perror("ioctl read packet failed");
                        close(simple_switch_fd);
                        return -1;
                }
        if (rcvd_pkt_meta >> 22) {
            printf("Port 1: \n");
            print_packet(&rcvd_pkt_meta);
            total_latency += extra_time_delta(rcvd_pkt_meta);
            num_get++;
        }
        usleep(1000);
        if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_2, &rcvd_pkt_meta) < 0) {
                        perror("ioctl read packet failed");
                        close(simple_switch_fd);
                        return -1;
                }
        if (rcvd_pkt_meta >> 22) {
            printf("Port 2: \n");
            print_packet(&rcvd_pkt_meta);
            total_latency += extra_time_delta(rcvd_pkt_meta);
            num_get++;
        }
        usleep(1000);
        if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_3, &rcvd_pkt_meta) < 0) {
                        perror("ioctl read packet failed");
                        close(simple_switch_fd);
                        return -1;
                }
        if (rcvd_pkt_meta >> 22) {
            printf("Port 3: \n");
            print_packet(&rcvd_pkt_meta);
            total_latency += extra_time_delta(rcvd_pkt_meta);
            num_get++;
        }
        }
    printf("Got %d packets\n", num_get);
    printf("Total latency: %d\n", total_latency);

    close(simple_switch_fd);
        printf("Userspace program terminating\n");
        return 0;
}
```

12.2.2 Even Load (w/ Priorities)

```c
#include <stdio.h>
#include "simpleSwitch.h"
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>

#define DEVICE_PATH "/dev/simple_driver"
int simple_switch_fd;

int num_sent = 0;
int num_get = 0;

void open_simple_device()
{
   simple_switch_fd = open(DEVICE_PATH, O_RDWR);
   if (simple_switch_fd < 0) {
      perror("Failed to open simple device");
      return;
   }
}

void print_packet(void *packet_data)
{
      unsigned int packet = *((unsigned int*) packet_data);

   printf("\tmetadata: [%u | %u | %u | %u | %u ]\n",
      (packet >> 30) & 0x3, // Extract bits 31:30
      (packet >> 28) & 0x3, // Extract bits 29:28
      (packet >> 22) & 0x3F, // Extract bits 27:22
      (packet >> 11) & 0x7FF, // Extract bits 21:11
       packet & 0x7FF   // Extract bits 10:0
   );
}

int extra_time_delta(unsigned int packet)
{
   return (packet & 0x7FF) - ((packet >> 11) & 0x7FF);
}

int extra_dst_port(unsigned int packet)
{
   return (packet >> 28) & 0x3;
}

void set_ctrl_register(const packet_ctrl_t *pkt_ctrl)
{
      if (ioctl(simple_switch_fd, SIMPLE_WRITE_CTRL, pkt_ctrl) < 0) {
      perror("ioctl(SIMPLE_WRITE_CTRL) set CTRL failed\n");
```

```
        close(simple_switch_fd);
        return;
    }
}

void send_packet(const packet_meta_t *pkt_meta)
{
    usleep(5000);
    if (ioctl(simple_switch_fd, SIMPLE_WRITE_PACKET, pkt_meta) < 0) {
        perror("Failed to send packet");
        return;
    }
    num_sent++;
}

void set_packet_length(packet_meta_t *pkt_meta, unsigned int length)
{
        if (length > 0x3F) {
                perror("Failed to set length\n");
                return;
        }
    // Mask and shift length @ bit pos [27:22]
    *pkt_meta |= (length & 0x3F) << 22;
}


void set_all_packet_fields(packet_meta_t *pkt_meta, unsigned int dst,
                                                            unsigned int src,
                                                                unsigned int length)
{
        packet_meta_t pkt_tmp;
        *pkt_meta = 0;

        set_packet_length(pkt_meta, length);
        *pkt_meta = set_dst_port(*pkt_meta, dst);
        *pkt_meta = set_src_port(*pkt_meta, src);
}


int main()
{
    packet_meta_t pkt_meta, rcvd_pkt_meta;
    packet_ctrl_t pkt_ctrl;
    int total_latency = 0;

    open_simple_device();

        printf("Start: \n");
    pkt_ctrl = 0;
        set_ctrl_register(&pkt_ctrl);
        print_packet(&pkt_ctrl);
    usleep(1000);

    // Set control register (CTRL=1)
        printf("Start sending\n");
    pkt_ctrl = 1;
        set_ctrl_register(&pkt_ctrl);
        print_packet(&pkt_ctrl);
    usleep(1000);

    for (int i = 0; i < 32; i++) {
        set_all_packet_fields(&pkt_meta, (i+1)%4, i%4, 1);
                send_packet(&pkt_meta);
                print_packet(&pkt_meta);
        }

    usleep(10000);
    printf("Start recving\n");
        pkt_ctrl = 222;
        set_ctrl_register(&pkt_ctrl);
        print_packet(&pkt_ctrl);
    usleep(10000);

        printf("Requested %d packets\n", num_sent);
        for (int i = 0; i < 200; i++) {
        usleep(1000);
        if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_0, &rcvd_pkt_meta) < 0) {
                        perror("ioctl read packet failed");
                        close(simple_switch_fd);
                        return -1;
                }
        if (rcvd_pkt_meta >> 22) {
            printf("Port 0: \n");
            print_packet(&rcvd_pkt_meta);
            total_latency += extra_time_delta(rcvd_pkt_meta);
            num_get++;
        }
        usleep(1000);
        if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_1, &rcvd_pkt_meta) < 0) {
                        perror("ioctl read packet failed");
                        close(simple_switch_fd);
                        return -1;
                }
        if (rcvd_pkt_meta >> 22) {
            printf("Port 1: \n");
            print_packet(&rcvd_pkt_meta);
            total_latency += extra_time_delta(rcvd_pkt_meta);
            num_get++;
        }
        usleep(1000);
        if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_2, &rcvd_pkt_meta) < 0) {
                        perror("ioctl read packet failed");
                        close(simple_switch_fd);
                        return -1;
                }
        if (rcvd_pkt_meta >> 22) {
            printf("Port 2: \n");
            print_packet(&rcvd_pkt_meta);
            total_latency += extra_time_delta(rcvd_pkt_meta);
            num_get++;
        }
        usleep(1000);
        if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_3, &rcvd_pkt_meta) < 0) {
                        perror("ioctl read packet failed");
                        close(simple_switch_fd);
                        return -1;
                }
        if (rcvd_pkt_meta >> 22) {
            printf("Port 3: \n");
            print_packet(&rcvd_pkt_meta);
            total_latency += extra_time_delta(rcvd_pkt_meta);
            num_get++;
        }
        }
    printf("Got %d packets\n", num_get);
    printf("Total latency: %d\n", total_latency);

    close(simple_switch_fd);
        printf("Userspace program terminating\n");
        return 0;
}
```

12.2.3 Simple Switch

```

/*
 * TODO:
 * [ ] write to CTRL register (CTRL=1)
 *     inputting meta data
 * [ ] start w/ single threaded test prog
 * [ ] send 2 packets (call 2 ioctl w/ write + data flags & other combinations)
 * [ ] try setting different parameters using ioctl (i.e., src, dest, len)
 * [ ] len=1 or 2
 * [ ] ioctl_write (set CTRL=2) for both read & write
 * [ ] ioctl_read should return the 2 packets
 *    (1) inf loop to read packets: can i write at the same time???
 *     OR
 *    (2) ioctl w/ different flags
 *    (3)
 *     [ ]
 * [ ] later: create separate thread for polling
 *
 * ioctrl_write w/ write flag (1) and data and actual packet metadata that we
    assemble here
 * all port 0 (src + dest)
 */

#include <stdio.h>
#include "simpleSwitch.h"
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>

#define MIN_PORT_NUM 0
#define MAX_PORT_NUM 3
#define DEVICE_PATH "/dev/simple_driver"


int simple_switch_fd;

// High [2 bit src] [2 bit dst] [6 bits for the # of 32 byte chunks] [22 bit time
   stamp] Low
// typedef unsigned int packet_meta_t;
// typedef unsigned int packet_ctrl_t;
// packet_info_t;

void open_simple_device()
{
   simple_switch_fd = open(DEVICE_PATH, O_RDWR);
   if (simple_switch_fd < 0) {
      perror("Failed to open simple device");
      return;
    }
}

void print_packet_no_hw(void *packet_data)
{
        unsigned int packet = *((unsigned int*) packet_data);

    printf("\tmetadata (0x%X): [%u | %u | %u | %u]\n",
            packet,
        (packet >> 30) & 0x3, // Extract bits 31:30
        (packet >> 28) & 0x3, // Extract bits 29:28
        (packet >> 22) & 0x3F, // Extract bits 27:22
         packet & 0x3FFFFF  // Extract bits 21:0
    );
}

void print_packet(void *packet_data)
{
        unsigned int packet = *((unsigned int*) packet_data);
    if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_0, packet_data)) {
        perror("ioctl(SIMPLE_READ_PACKET_0) failed");
        return;
    }

    printf("\tmetadata: [%u | %u | %u | %u]\n",
        (packet >> 30) & 0x3, // Extract bits 31:30
        (packet >> 28) & 0x3, // Extract bits 29:28
        (packet >> 22) & 0x3F, // Extract bits 27:22
         packet & 0x3FFFFF  // Extract bits 21:0
    );
}

void set_ctrl_register(const packet_ctrl_t *pkt_ctrl)
{
        if (ioctl(simple_switch_fd, SIMPLE_WRITE_CTRL, pkt_ctrl) < 0) {
        perror("ioctl(SIMPLE_WRITE_CTRL) set CTRL=1 failed\n");
        close(simple_switch_fd);
        return;
    }
}

void send_packet(const packet_meta_t *pkt_meta)
{
    sleep(0.5);
    if (ioctl(simple_switch_fd, SIMPLE_WRITE_PACKET, pkt_meta) < 0) {
        perror("Failed to send packet");
        return;
    }
}

// void receive_packet(int packet_id, packet_meta_t *pkt_meta)
// {
//    if (ioctl(simple_switch_fd, cmd, pkt_meta) < 0) {
```

```
//       perror("Failed to receive packet");
//       return;
//    }
// }

/**
 * Sets the length and time_delta fields of the packet metadata.
 * Assumes that @pkt_meta is cleared prior to calling this function.
 * @param pkt_meta Pointer to the packet metadata.
 * @param length Length of the packet, to be set in bits [27:22].
 */
void set_packet_length(packet_meta_t *pkt_meta, unsigned int length)
{
       if (length > 0x3F) {
              perror("Failed to set length\n");
              return;
       }
    // Mask and shift length @ bit pos [27:22]
    *pkt_meta |= (length & 0x3F) << 22;
}


void set_all_packet_fields(packet_meta_t *pkt_meta, unsigned int dst,
                                                                unsigned int src,
                                                                   unsigned int length)
{
       packet_meta_t pkt_tmp;
       *pkt_meta = 0;

       set_packet_length(pkt_meta, length);
       *pkt_meta = set_dst_port(*pkt_meta, dst);
       *pkt_meta = set_src_port(*pkt_meta, src);
}


int main()
{
    int write_num_packets = 2, num_sent = 0;
       unsigned int dest = 0, src = 0, len = 1, t_delta = 10;
    packet_meta_t pkt_meta, rcvd_pkt_meta;
    packet_ctrl_t pkt_ctrl;

    open_simple_device();

       printf("Set CTRL register to 0\n");
    pkt_ctrl = 0;
       set_ctrl_register(&pkt_ctrl);
       print_packet_no_hw(&pkt_ctrl);

    // Set control register (CTRL=1)
       printf("Set CTRL register to 1\n");
    pkt_ctrl = 1;
       set_ctrl_register(&pkt_ctrl);
```

```
        print_packet_no_hw(&pkt_ctrl);

    for (int i = 0; i < 10; i++) {
        set_all_packet_fields(&pkt_meta, i%2, i%4, 4);
               send_packet(&pkt_meta);
               print_packet_no_hw(&pkt_meta);
        }
        num_sent += write_num_packets;

        len = 2;

        set_all_packet_fields(&pkt_meta, 0, 4, 4);
    for (int i = 0; i < 10; i++)
        set_all_packet_fields(&pkt_meta, i%4, (i+1)%4, 4);
        send_packet(&pkt_meta);
               print_packet_no_hw(&pkt_meta);
        num_sent += write_num_packets;

        printf("Set CTRL register to 2\n");
        pkt_ctrl = 2;
        set_ctrl_register(&pkt_ctrl);
        print_packet_no_hw(&pkt_ctrl);

        printf("Requested %d packets\n", num_sent);
        for (int i = 0; i < 5; i++) {
        if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_0, &rcvd_pkt_meta) < 0) {
                      perror("ioctl read packet failed");
                      close(simple_switch_fd);
                      return -1;
               }
        sleep(0.5);
        printf("Port 0\n");
               print_packet_no_hw(&rcvd_pkt_meta);
               if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_1, &rcvd_pkt_meta) < 0) {
                      perror("ioctl read packet failed");
                      close(simple_switch_fd);
                      return -1;
               }
        sleep(0.5);
        printf("Port 1\n");
               print_packet_no_hw(&rcvd_pkt_meta);
               if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_2, &rcvd_pkt_meta) < 0) {
                      perror("ioctl read packet failed");
                      close(simple_switch_fd);
                      return -1;
               }
        sleep(0.5);
        printf("Port 2\n");
               print_packet_no_hw(&rcvd_pkt_meta);
               if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_3, &rcvd_pkt_meta) < 0) {
                      perror("ioctl read packet failed");
                      close(simple_switch_fd);
                      return -1;
```

```c
                }
                sleep(0.5);
        printf("Port 3\n");
                print_packet_no_hw(&rcvd_pkt_meta);
        }

    close(simple_switch_fd);
        printf("Userspace program terminating\n");
        return 0;
}
```

```c
#ifndef _simpleSwitch_H
#define _simpleSwitch_H
#include "driver/simple_driver.h"

packet_meta_t set_src_port(packet_meta_t meta, unsigned int port) {
    if (port > 3) {
        printf("Ports number (%u) should be btw 0 and 3. \n", port);
        return meta;
    }
    meta &= ~(0x3 << 30);

    switch(port) {
        case 0:
            break;
        case 1:
            meta |= (0x1 << 30);
            break;
        case 2:
            meta |= (0x2 << 30);
            break;
        case 3:
            meta |= (0x3 << 30);
            break;
        default:
            break;
    }
    return meta;
}

packet_meta_t set_dst_port(packet_meta_t meta, unsigned int port) {
    if (port > 3) {
        printf("Ports number (%u) should be btw 0 and 3. \n", port);
        return meta;
    }
    // Clear the bits 28 and 29
    meta &= ~(0x3 << 28);

    switch(port) {
        case 0: // 00
            break; // No action needed as bits 28 and 29 are already cleared
        case 1: // 01
```

```c
            meta |= (0x1 << 28); // Set bit 28
            break;
        case 2: // 10
            meta |= (0x2 << 28); // Set bit 29
            break;
        case 3: // 11
            meta |= (0x3 << 28); // Set both bits 28 and 29
            break;
        default:
            break;
    }

    return meta;
}

#endif
```

12.2.4 Skewed Load

```c
#include <stdio.h>
#include "simpleSwitch.h"
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>

#define DEVICE_PATH "/dev/simple_driver"
int simple_switch_fd;

int num_sent = 0;
int num_get = 0;

void open_simple_device()
{
   simple_switch_fd = open(DEVICE_PATH, O_RDWR);
   if (simple_switch_fd < 0) {
      perror("Failed to open simple device");
      return;
   }
}

void print_packet(void *packet_data)
{
      unsigned int packet = *((unsigned int*) packet_data);

   printf("\tmetadata: [%u | %u | %u | %u | %u ]\n",
      (packet >> 30) & 0x3, // Extract bits 31:30
      (packet >> 28) & 0x3, // Extract bits 29:28
      (packet >> 22) & 0x3F, // Extract bits 27:22
      (packet >> 11) & 0x7FF, // Extract bits 21:11
       packet & 0x7FF   // Extract bits 10:0
   );
}

int extra_time_delta(unsigned int packet)
{
   return (packet & 0x7FF) - ((packet >> 11) & 0x7FF);
}

int extra_dst_port(unsigned int packet)
{
   return (packet >> 28) & 0x3;
}

void set_ctrl_register(const packet_ctrl_t *pkt_ctrl)
{
      if (ioctl(simple_switch_fd, SIMPLE_WRITE_CTRL, pkt_ctrl) < 0) {
      perror("ioctl(SIMPLE_WRITE_CTRL) set CTRL failed\n");
```

```c
        close(simple_switch_fd);
        return;
    }
}

void send_packet(const packet_meta_t *pkt_meta)
{
    usleep(5000);
    if (ioctl(simple_switch_fd, SIMPLE_WRITE_PACKET, pkt_meta) < 0) {
        perror("Failed to send packet");
        return;
    }
    num_sent++;
}


/**
 * Sets the length and time_delta fields of the packet metadata.
 * Assumes that @pkt_meta is cleared prior to calling this function.
 * @param pkt_meta Pointer to the packet metadata.
 * @param length Length of the packet, to be set in bits [27:22].
 */
void set_packet_length(packet_meta_t *pkt_meta, unsigned int length)
{
        if (length > 0x3F) {
                perror("Failed to set length\n");
                return;
        }
    // Mask and shift length @ bit pos [27:22]
    *pkt_meta |= (length & 0x3F) << 22;
}


void set_all_packet_fields(packet_meta_t *pkt_meta, unsigned int dst, unsigned int
    src, unsigned int length)
{
        packet_meta_t pkt_tmp;
        *pkt_meta = 0;

        set_packet_length(pkt_meta, length);
        *pkt_meta = set_dst_port(*pkt_meta, dst);
        *pkt_meta = set_src_port(*pkt_meta, src);
}


int main()
{
    packet_meta_t pkt_meta, rcvd_pkt_meta;
    packet_ctrl_t pkt_ctrl;
    int total_latency = 0;

    open_simple_device();

```

```
        printf("Start: \n");
    pkt_ctrl = 0;
        set_ctrl_register(&pkt_ctrl);
        print_packet(&pkt_ctrl);
    usleep(1000);

    // Set control register (CTRL=1)
        printf("Start sending\n");
    pkt_ctrl = 1;
        set_ctrl_register(&pkt_ctrl);
        print_packet(&pkt_ctrl);
    usleep(1000);

    for (int i = 0; i < 16; i++) {
        set_all_packet_fields(&pkt_meta, (i+1)%2, i%4, 1);
                send_packet(&pkt_meta);
                print_packet(&pkt_meta);
        }

    for (int i = 0; i < 16; i++) {
        set_all_packet_fields(&pkt_meta, (i+3)%4, (i+1)%4, 1);
        send_packet(&pkt_meta);
                print_packet(&pkt_meta);
    }

    usleep(10000);
        printf("Start recving\n");
        pkt_ctrl = 2;
        set_ctrl_register(&pkt_ctrl);
        print_packet(&pkt_ctrl);
    usleep(10000);

        printf("Requested %d packets\n", num_sent);
        for (int i = 0; i < 1000; i++) {
        usleep(1000);
        if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_0, &rcvd_pkt_meta) < 0) {
                        perror("ioctl read packet failed");
                        close(simple_switch_fd);
                        return -1;
                }
        if (rcvd_pkt_meta >> 22) {
            printf("Port 0: \n");
            print_packet(&rcvd_pkt_meta);
            total_latency += extra_time_delta(rcvd_pkt_meta);
            num_get++;
        }
        usleep(1000);
        if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_1, &rcvd_pkt_meta) < 0) {
                        perror("ioctl read packet failed");
                        close(simple_switch_fd);
                        return -1;
                }
        if (rcvd_pkt_meta >> 22) {
            printf("Port 1: \n");
            print_packet(&rcvd_pkt_meta);
            total_latency += extra_time_delta(rcvd_pkt_meta);
            num_get++;
        }
        usleep(1000);
        if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_2, &rcvd_pkt_meta) < 0) {
                        perror("ioctl read packet failed");
                        close(simple_switch_fd);
                        return -1;
                }
        if (rcvd_pkt_meta >> 22) {
            printf("Port 2: \n");
            print_packet(&rcvd_pkt_meta);
            total_latency += extra_time_delta(rcvd_pkt_meta);
            num_get++;
        }
        usleep(1000);
        if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_3, &rcvd_pkt_meta) < 0) {
                        perror("ioctl read packet failed");
                        close(simple_switch_fd);
                        return -1;
                }
        if (rcvd_pkt_meta >> 22) {
            printf("Port 3: \n");
            print_packet(&rcvd_pkt_meta);
            total_latency += extra_time_delta(rcvd_pkt_meta);
            num_get++;
        }
        }
    printf("Got %d packets\n", num_get);
    printf("Total latency: %d\n", total_latency);

    close(simple_switch_fd);
        printf("Userspace program terminating\n");
        return 0;
}
```

12.2.5 Skewed Load (w/ Priorities)

```c

#include <stdio.h>
#include "simpleSwitch.h"
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>

#define DEVICE_PATH "/dev/simple_driver"
int simple_switch_fd;

int num_sent = 0;
int num_get = 0;

void open_simple_device()
{
    simple_switch_fd = open(DEVICE_PATH, O_RDWR);
    if (simple_switch_fd < 0) {
        perror("Failed to open simple device");
        return;
    }
}

void print_packet(void *packet_data)
{
        unsigned int packet = *((unsigned int*) packet_data);

    printf("\tmetadata: [%u | %u | %u | %u | %u ]\n",
        (packet >> 30) & 0x3, // Extract bits 31:30
        (packet >> 28) & 0x3, // Extract bits 29:28
        (packet >> 22) & 0x3F, // Extract bits 27:22
        (packet >> 11) & 0x7FF, // Extract bits 21:11
         packet & 0x7FF   // Extract bits 10:0
    );
}

int extra_time_delta(unsigned int packet)
{
    return (packet & 0x7FF) - ((packet >> 11) & 0x7FF);
}

int extra_dst_port(unsigned int packet)
{
    return (packet >> 28) & 0x3;
}

void set_ctrl_register(const packet_ctrl_t *pkt_ctrl)
{
        if (ioctl(simple_switch_fd, SIMPLE_WRITE_CTRL, pkt_ctrl) < 0) {
```

```
        perror("ioctl(SIMPLE_WRITE_CTRL) set CTRL failed\n");
        close(simple_switch_fd);
        return;
    }
}

void send_packet(const packet_meta_t *pkt_meta)
{
    usleep(5000);
    if (ioctl(simple_switch_fd, SIMPLE_WRITE_PACKET, pkt_meta) < 0) {
        perror("Failed to send packet");
        return;
    }
    num_sent++;
}


void set_packet_length(packet_meta_t *pkt_meta, unsigned int length)
{
        if (length > 0x3F) {
                perror("Failed to set length\n");
                return;
        }
    // Mask and shift length @ bit pos [27:22]
    *pkt_meta |= (length & 0x3F) << 22;
}


void set_all_packet_fields(packet_meta_t *pkt_meta, unsigned int dst,
                                                                          unsigned int src,
                                                                              unsigned int length)
{
        packet_meta_t pkt_tmp;
        *pkt_meta = 0;

        set_packet_length(pkt_meta, length);
        *pkt_meta = set_dst_port(*pkt_meta, dst);
        *pkt_meta = set_src_port(*pkt_meta, src);
}


int main()
{
    packet_meta_t pkt_meta, rcvd_pkt_meta;
    packet_ctrl_t pkt_ctrl;
    int total_latency = 0;

    open_simple_device();

        printf("Start: \n");
    pkt_ctrl = 0;
        set_ctrl_register(&pkt_ctrl);
        print_packet(&pkt_ctrl);
    usleep(1000);

    // Set control register (CTRL=1)
        printf("Start sending\n");
    pkt_ctrl = 1;
        set_ctrl_register(&pkt_ctrl);
        print_packet(&pkt_ctrl);
    usleep(1000);

    for (int i = 0; i < 16; i++) {
        set_all_packet_fields(&pkt_meta, (i+1)%2, i%4, 1);
                send_packet(&pkt_meta);
                print_packet(&pkt_meta);
        }

    for (int i = 0; i < 16; i++) {
        set_all_packet_fields(&pkt_meta, (i+3)%4, (i+1)%4, 1);
        send_packet(&pkt_meta);
                print_packet(&pkt_meta);
    }

    usleep(10000);
        printf("Start recving\n");
        pkt_ctrl = 222;
        set_ctrl_register(&pkt_ctrl);
        print_packet(&pkt_ctrl);
    usleep(10000);

        printf("Requested %d packets\n", num_sent);
        for (int i = 0; i < 200; i++) {
        usleep(1000);
        if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_0, &rcvd_pkt_meta) < 0) {
                    perror("ioctl read packet failed");
                    close(simple_switch_fd);
                    return -1;
                }
        if (rcvd_pkt_meta >> 22) {
            printf("Port 0: \n");
            print_packet(&rcvd_pkt_meta);
            total_latency += extra_time_delta(rcvd_pkt_meta);
            num_get++;
        }
        usleep(1000);
        if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_1, &rcvd_pkt_meta) < 0) {
                    perror("ioctl read packet failed");
                    close(simple_switch_fd);
                    return -1;
                }
        if (rcvd_pkt_meta >> 22) {
            printf("Port 1: \n");
            print_packet(&rcvd_pkt_meta);
            total_latency += extra_time_delta(rcvd_pkt_meta);
            num_get++;
        }
        usleep(1000);
        if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_2, &rcvd_pkt_meta) < 0) {
                        perror("ioctl read packet failed");
                        close(simple_switch_fd);
                        return -1;
                }
        if (rcvd_pkt_meta >> 22) {
            printf("Port 2: \n");
            print_packet(&rcvd_pkt_meta);
            total_latency += extra_time_delta(rcvd_pkt_meta);
            num_get++;
        }
        usleep(1000);
        if (ioctl(simple_switch_fd, SIMPLE_READ_PACKET_3, &rcvd_pkt_meta) < 0) {
                        perror("ioctl read packet failed");
                        close(simple_switch_fd);
                        return -1;
                }
        if (rcvd_pkt_meta >> 22) {
            printf("Port 3: \n");
            print_packet(&rcvd_pkt_meta);
            total_latency += extra_time_delta(rcvd_pkt_meta);
            num_get++;
        }
        }
    printf("Got %d packets\n", num_get);
    printf("Total latency: %d\n", total_latency);

    close(simple_switch_fd);
        printf("Userspace program terminating\n");
        return 0;
}
```

12.3 Simple Switch: Verilator

12.3.1 Pick VOQ

```cpp


#include <iostream>
#include <iomanip>
#include "Vpick_voq.h"
#include <verilated.h>
#include <bitset>

unsigned char input_voq_empty  [  ] = {0b1110, 0b0001, 0b0011, 0b1111, 0b0000,
    0b1110, 0b0111, 0b0100, 0b0110, 0b0000};

unsigned char input_voq_picked  [  ] = {0b1110, 0b0001, 0b0011, 0b1111, 0b0000,
    0b1110, 0b0111, 0b1000, 0b1001, 0b1111};

unsigned char input_start_voq_num  [  ] = {3, 3, 1, 2, 1, 1, 0, 2, 0, 0};

unsigned char output_no_available_voq  [  ] = {0, 0, 0, 1, 0, 0, 0, 0, 1, 1};

unsigned char output_voq_to_pick  [  ] = {0, 3, 2, 3, 1, 0, 3, 0, 0, 0};


int main(int argc, const char ** argv, const char ** env) {
  int exitcode = 0;

  Verilated::commandArgs(argc, argv);

  Vpick_voq * dut = new Vpick_voq; // Instantiate the collatz module

  for (int i = 0 ; i < 8; i++) {
    dut->voq_empty = input_voq_empty  [  i  ];
    dut->start_voq_num = input_start_voq_num  [  i  ];
    dut->voq_picked = input_voq_picked  [  i  ];
    dut->eval();
    std::bitset<4> x(dut->voq_empty);
    std::cout << "voq_empty: " << x << '\n';
    std::bitset<4> y(dut->voq_picked);
    std::cout << "voq_picked: " << y << '\n';

    std::cout << "start_voq_num: " << (int) dut->start_voq_num << '\n';

    if (dut->no_available_voq == output_no_available_voq  [  i
        ] && (dut->voq_to_pick == output_voq_to_pick  [  i
        ] || dut->no_available_voq == 1))
        std::cout << " OK" << '\n';
    else {
        std::cout << " INCORRECT expected no_available_voq and voq_to_pick " <<
            std::endl;
        exitcode = 1;
    }
    std::cout << "no_available_voq: " << (int) dut->no_available_voq << '\n';
    std::cout << "voq_to_pick: " << (int) dut->voq_to_pick << '\n';
    std::cout << std::endl;
  }

  dut->final(); // Stop the simulation
  delete dut;

  return exitcode;
}
```

12.3.2 VMU

```cpp
#include <iostream>
#include "Vvmu.h"
#include <verilated.h>
#include <verilated_vcd_c.h>

using namespace std;

int main(int argc, const char ** argv, const char ** env) {
    Verilated::commandArgs(argc, argv);
    Vvmu * dut = new Vvmu;

    Verilated::traceEverOn(true);
    VerilatedVcdC * tfp = new VerilatedVcdC;
    dut->trace(tfp, 99); // Verilator should trace signals up to 99 levels deep
    tfp->open("vmu.vcd");


    tfp->close(); // Stop dumping the VCD file
    delete tfp;

    dut->final(); // Stop the simulation
    delete dut;
    return 0;
}
```

12.3.3 Scheduler

```

#include <iostream>
#include "Vsched.h"
#include <verilated.h>
#include <verilated_vcd_c.h>

using namespace std;

unsigned short is_busy  [  ] = {0b1110, 0b1110, 0b1110, 0b0000};

unsigned short busy_voq_num  [  ] = {0b11100111, 0b11100111, 0b11100111, 0b00000000};

unsigned short voq_empty  [  ] = {0b0000000000001110, 0b0000000000000000,
    0b0000000000001111, 0b1011011111101101};


int main(int argc, const char ** argv, const char ** env) {
  Verilated::commandArgs(argc, argv);

  // Treat the argument on the command-line as the place to start
  int n;
  if (argc > 1 && argv  [  1  ]  [  0  ] != '+') n = atoi(argv  [  1  ]);
  else n = 4; // Default

  Vsched * dut = new Vsched; // Instantiate the sched module

  // Enable dumping a VCD file

  Verilated::traceEverOn(true);
  VerilatedVcdC * tfp = new VerilatedVcdC;
  dut->trace(tfp, 99); // Verilator should trace signals up to 99 levels deep
  tfp->open("sched.vcd");

  dut->sched_en = 0;
  dut->is_busy = 0;
  dut->busy_voq_num = 0;
  dut->voq_empty = 0;

  // std::cout << dut->n; // Print the starting value of the sequence

  bool last_clk = true;
  int time;
  int iter = 0;
  for (time = 0 ; time < 1000; time += 10) {
    std::cout << "time: " << time << std::endl;
    dut->clk = ((time % 20) >= 10) ? 1 :0; // Simulate a 50 MHz clock
    if ((time % 160) == 20) {
      if (iter < n) {
        dut->sched_en = 0;
        dut->is_busy = is_busy  [  iter  ];
        dut->busy_voq_num = busy_voq_num  [  iter  ];
        dut->voq_empty = voq_empty  [  iter  ];
      }
    } else if ((time % 160) == 30 || (time % 160) == 40) {
      if (iter < n) {
        dut->sched_en = 1;
      }
    } else {
      dut->sched_en = 0;
      if ((time % 160) == 50) {
          iter += 1;
          cout << iter << endl;
      }
    }

    dut->eval();   // Run the simulation for a cycle
    tfp->dump(time); // Write the VCD file for this cycle
  }

  std::cout << std::endl;

  // Once "done" is received, run a few more clock cycles

  // for (int k = 0 ; k < 4 ; k++, time += 10) {
  //   dut->clk = ((time % 20) >= 10) ? 1 : 0;
  //     dut->eval();
  //     tfp->dump(time);
  // }

  tfp->close(); // Stop dumping the VCD file
  delete tfp;

  dut->final(); // Stop the simulation
  delete dut;

  return 0;
}
```

12.3.4 Simple Switch

```

#include <iostream>
#include "Vsimple_switch.h"
#include <verilated.h>
#include <verilated_vcd_c.h>

using namespace std;

int main(int argc, const char ** argv, const char ** env) {
  Verilated::commandArgs(argc, argv);

  // Treat the argument on the command-line as the place to start
  int n;
  if (argc > 1 && argv  [  1  ]  [  0  ] != '+') n = atoi(argv  [  1  ]);
  else n = 4; // Default

  Vsimple_switch * dut = new Vsimple_switch; // Instantiate the packet_gen module

  // Enable dumping a VCD file

  Verilated::traceEverOn(true);
  VerilatedVcdC * tfp = new VerilatedVcdC;
  dut->trace(tfp, 99); // Verilator should trace signals up to 99 levels deep
  tfp->open("simple_switch.vcd");

  // std::cout << dut->n; // Print the starting value of the sequence

  bool last_clk = true;
  int time;
  int iter = 0;
  dut->reset = 0;

  for (time = 0 ; time < 10000; time += 10) {
    std::cout << "time: " << time << std::endl;
    dut->clk = ((time % 20) >= 10) ? 0 :1; // Simulate a 50 MHz clock
    if (time == 40) {
      dut->chipselect = 1;
      dut->address = 2;
      dut -> write = 1;
      dut -> read = 0;
      dut->writedata = 0b0111000001000000000000000000000;
    }
    if (time == 60) {
      dut -> chipselect = 0;
    }

  if (time == 1540) {
      dut->chipselect = 1;
      dut->address = 1;
      dut -> write = 1;
      dut -> read = 0;
      dut->writedata = 0b00110000100000000000000000000000;
    }
    if (time == 1560) {
      dut -> chipselect = 0;
    }


    if (time == 2440) {
      dut->chipselect = 1;
      dut->address = 3;
      dut -> write = 1;
      dut -> read = 0;
      dut->writedata = 0b10110000100000000000000000000000;
    }
    if (time == 2460) {
      dut -> chipselect = 0;
      dut -> write = 0;
    }

    if (time == 3040) {
      dut->chipselect = 1;
      dut->address = 1;
      dut -> write = 1;
      dut -> read = 0;
      dut->writedata = 0b00100000100000000000000000000000;
    }

    if (time == 3060) {
      dut -> chipselect = 0;
    }

        if (time == 3040) {
      dut->chipselect = 1;
      dut->address = 1;
      dut -> write = 1;
      dut -> read = 0;
      dut->writedata = 0b00100000100000000000000000000000;
    }

    if (time == 3560) {
      dut -> chipselect = 0;
    }
            if (time == 3040) {
      dut->chipselect = 1;
      dut->address = 2;
      dut -> write = 1;
      dut -> read = 0;
      dut->writedata = 0b01100000100000000000000000000000;
    }
    if (time == 3060) {
      dut -> chipselect = 0;
      dut -> write = 0;
    }



    if(time == 4000) {
      dut -> write = 1;
      dut -> chipselect = 1;
      dut -> read = 0;
      dut -> address = 0;
      dut->writedata = 2;
    }
    if(time == 4020) {
      dut -> chipselect = 0;
    }

    if(time == 5000) {
      dut -> write = 0;
      dut -> chipselect = 1;
      dut -> read = 1;
      dut -> address = 4;
    }
    if(time == 5020) {
      dut -> chipselect = 0;
    }

    if(time == 7000) {
      dut -> write = 0;
      dut -> chipselect = 1;
      dut -> read = 1;
      dut -> address = 4;
    }
    if(time == 7020) {
      dut -> chipselect = 0;
    }

    if(time == 8000) {
      dut -> write = 0;
      dut -> chipselect = 1;
      dut -> read = 1;
      dut -> address = 3;
    }
    if(time == 8020) {
      dut -> chipselect = 0;
    }
    if(time == 9000) {
      dut -> write = 0;
      dut -> chipselect = 1;
      dut -> read = 1;
      dut -> address = 3;
    }
    if(time == 9020) {
      dut -> chipselect = 0;
      dut -> read = 0;
    }

    dut->eval();   // Run the simulation for a cycle
    tfp->dump(time); // Write the VCD file for this cycle
  }

  std::cout << std::endl;

  // Once "done" is received, run a few more clock cycles

  // for (int k = 0 ; k < 4 ; k++, time += 10) {
  //   dut->clk = ((time % 20) >= 10) ? 1 : 0;
  //     dut->eval();
  //     tfp->dump(time);
  // }

  tfp->close(); // Stop dumping the VCD file
  delete tfp;

  dut->final(); // Stop the simulation
  delete dut;

  return 0;
}
```

12.4 daFPGASwitch: Hardware

12.4.1 CMU

```
/*
The packet management unit:

*/

/*
ctrl memory is divided into 16-bit chunks
and each of them represents a packet segment (32 bytes) inside the data memory

Inside each chunk:
* 1-bit of is_allocated
* 10-bit of next_addr (of ctrl memory, basically a linked list) / 10'b0 which is the
    end of packet chain.

A packet comes in that needs 2 block
Address 0: Null
Address 1: 1, 0000000002
Address 2: 1, 0000000000
*/

define D_WIDTH 11
module cmu (
    input logic clk,
    input logic   [  5   :   0   ] remaining_packet_length, // in blocks
    input logic alloc_en, free_en, reset,
    input logic   [  9   :   0   ] free_addr,

    output logic   [  9   :  0
       ] alloc_addr = 10'b0, // the address in cmem for the first chunk of the packet
        output logic   [  9   :  0   ] next_free_addr = 10'b0
);


    // Registers
    /* verilator lint_off UNOPTFLAT */
    logic  [  9   :  0   ] curr_write;
        logic  [  9   :  0  ]    next_write;
        logic  [  5   :  0  ]    empty_blocks;
        /* verilator lint_on UNOPTFLAT */
        logic  [  D_WIDTH  -1   :  0  ]   ctrl_in_a = D_WIDTH'b0;
        logic  [  D_WIDTH  -1   :  0  ]   ctrl_in_b = D_WIDTH'b0;
        /* verilator lint_off UNUSED */
        logic  [  D_WIDTH  -1   :  0  ]   ctrl_out_a, ctrl_out_b;
        /* verilator lint_on UNUSED */
        logic          ctrl_wen_a, ctrl_wen_b;
        //logic[9:0] next_ctrl;
        logic  [  9   :  0  ]    addr_a, addr_b;

        /* Create state control variables */
        logic  [  2   :  0  ]    input_state                  = 3'b0;
        logic  [  2   :  0  ]    next_input_state     = 3'b0;
```

```
	logic  [ 1  :  0  ]    output_state         = 2'b0;
	logic  [ 1  :  0  ]    next_output_state    = 2'b0;

	/* Temporary variables */
	logic  [ 5  :  0  ]    temp_empty_blocks;
	logic  [ 9  :  0  ]    temp_next_write;
	logic  [ 9  :  0  ]    temp_curr_write;

	always_comb begin
		/* Create input state machine */
		case (input_state)
			/* Determine CMEM space availability
			 * Read the next control */
			3'b000: begin
				ctrl_wen_a              = 1'b0;
				if (alloc_en == 1'b1) begin
					if (remaining_packet_length < empty_blocks) begin
						temp_empty_blocks = empty_blocks - 1;
						addr_a =      next_write;

						next_input_state      = 3'b001;
					end else begin
						alloc_addr                 = 10'b0;
						next_input_state     = 3'b000;
						temp_empty_blocks    = empty_blocks;
					end

				end else begin
					next_input_state     = 3'b000;
					temp_empty_blocks    = empty_blocks;
				end

			end

			/* Determine the next write */
			3'b001: begin
				if (ctrl_out_a == 11'b0) begin
					temp_next_write = next_write + 1;
				end else begin
					temp_next_write = ctrl_out_a  [  9  :  0  ];
				end

				next_input_state =   3'b010;
			end

			3'b010: begin

				next_input_state     =      3'b011;
			end

			3'b011: begin

				next_input_state = 3'b100;
```

```
                        end

                        3'b100: begin

                                next_input_state = 3'b101;
                        end

                        3'b101: begin

                                next_input_state = 3'b110;
                        end

                        3'b110: begin

                                next_input_state = 3'b111;
                        end

                        /* Format the control and write to cmem
                         * Return the next free address */
                        3'b111: begin
                                ctrl_in_a  [  10  ] = 1'b1;

                                if (remaining_packet_length > 1) begin
                                        ctrl_in_a  [  9  :  0  ] = next_write;
                                end else begin
                                        ctrl_in_a  [  9  :  0  ] = 10'b0;
                                end

                                alloc_addr           = next_write;
                                temp_curr_write = next_write;

                                ctrl_wen_a           = 1'b1;
                                addr_a               = curr_write;

                                next_input_state = 3'b0;
                        end

                        default: begin
                        end

                endcase


                /* Create output state machine */
                case (output_state)
                        /* Request to read control of the given address
                         * Format control to deallocate address*/
                        2'b00: begin
                                if (free_en == 1'b1) begin
                                        ctrl_wen_b     = 1'b0;

                                        addr_b         = free_addr;

                                        ctrl_in_b  = {1'b0, next_write};

                                        next_output_state = 2'b01;
                                end
                        end

                        /* Read the control of the given address
                         * Write to deallocate control memory */
                        2'b01: begin
                                next_free_addr = ctrl_out_b  [ 9  :  0  ];

                                ctrl_wen_b           =  1'b1;

                                temp_empty_blocks    =  empty_blocks + 1;
                                temp_next_write      =  free_addr;
                                next_output_state    =  2'b10;
                        end

                        2'b10: begin
                                ctrl_wen_b    = 1'b0;

                                next_output_state = 2'b00;
                        end

                        default: begin
                        end
                endcase


        end

        always @(posedge clk) begin
                if (reset == 1'b1) begin
                        empty_blocks <= 6'b111111;
                        next_write     <= 10'h1;
                        curr_write     <= 10'b1;

                        input_state <= 3'b0;
                        output_state <= 2'b0;

                end else begin
                        input_state <= next_input_state;
                        output_state <= next_output_state;

                        empty_blocks <= temp_empty_blocks;
                        if (temp_next_write != 10'b0) begin
                                next_write     <= temp_next_write;
                        end

                        if (temp_curr_write != 10'b0) begin
                                curr_write <= temp_curr_write;
                        end

```

```
              end


   end

   true_dual_port_mem #(.MEM_SIZE(1024), .DATA_WIDTH(D_WIDTH)) cmem
       (
              .clk(clk),
              .aa(addr_a),         .ab(addr_b),
              .da(ctrl_in_a),      .db(ctrl_in_b),
              .wa(ctrl_wen_a),     .wb(ctrl_wen_b),
              .qa(ctrl_out_a),     .qb(ctrl_out_b)
       );
endmodule
```

12.4.2 Crossbar

```
module crossbar #(
   parameter DATA_WIDTH = 2, // For testing
   parameter EGRESS_CNT = 4
) (
   input logic   [ $clog2 ( EGRESS_CNT ) * EGRESS_CNT -1 : 0 ] sched_sel,
   input logic   [ EGRESS_CNT -1 : 0 ] crossbar_in_en,
   input logic   [ DATA_WIDTH * EGRESS_CNT -1 : 0 ] crossbar_in,

   output logic   [ EGRESS_CNT -1 : 0 ] crossbar_out_en,
   output logic   [ DATA_WIDTH * EGRESS_CNT -1 : 0 ] crossbar_out
);

logic   [ $clog2 ( EGRESS_CNT ) : 0 ] i;
logic   [ $clog2 ( EGRESS_CNT ) -1 : 0 ] _i;

always_comb begin
   for (i = 0; i < EGRESS_CNT; i = i + 1) begin
      _i = i  [ $clog2 ( EGRESS_CNT ) -1 : 0 ];
      crossbar_out_en  [ sched_sel  [ i   *   $clog2 ( EGRESS_CNT )   + :
            $clog2 ( EGRESS_CNT ) ] ] = crossbar_in_en  [ _i ];
      crossbar_out  [ ( sched_sel  [ i   *   $clog2 ( EGRESS_CNT )   + :
            $clog2 ( EGRESS_CNT ) ]  )   *   DATA_WIDTH   + :
         DATA_WIDTH  ] = crossbar_in  [ _i   *   DATA_WIDTH   + :
         DATA_WIDTH  ];
   end
end

endmodule
```

12.4.3 daFPGASwitch

```
module da_fpga_switch
(
    // From sw
    input logic clk,
    input logic reset,
    input logic   [ 31  :  0  ] writedata,
    input logic      write,
    input logic      read,
    input logic   [ 2  :  0  ] address,
    input logic      chipselect,

    // To sw
    output logic   [  31  :  0  ] readdata
);

    logic meta_in_en, packet_en;
    logic   [  31  :  0  ] meta_in, packet, meta_out;
    logic experimenting;
    // Change to 4 bit later.
    logic meta_out_ack;

    packet_gen ingress_0(
        // Input
        .clk(clk), .reset(reset),
        .packet_gen_in_en(meta_in_en), .experimenting(experimenting),
        .packet_gen_in(meta_in),

        // Output
        .packet_gen_out_en(packet_en), .packet_gen_out(packet));

    packet_val egress_0 (
        // Input
        .clk(clk), .reset(reset),
        .egress_in(packet),
        .egress_in_en(packet_en), .egress_in_ack(meta_out_ack),

        // Output
        .egress_out(meta_out));

    hw_sw_interface hw_sw_interface (
        .clk(clk),

        // Input: sw->interface
        .reset(reset),
        .writedata(writedata),// sw->hw
        .write(write), // sw->hw
        .read(read), // hw->sw
        .address(address),
        .chipselect(chipselect),

        // Input: hw->interface
        .interface_in(meta_out),

        // Output: interface->sw
        .readdata(readdata),

        // Output: interface->egress (ack)
        .interface_out_ack(meta_out_ack),

        // Output: To ingress/packet_val
        .interface_out_en(meta_in_en),
        .interface_out(meta_in),

        // Output: Ongoing experiment.
        .experimenting(experimenting)

    );

endmodule
```

12.4.4 Egress (Packet Validation)

```
/*

Packet metadata definition:
* src port: 2 bits
* dest port: 2 bits
* length: 12 bit (Packet size is 32 Bytes * length, min is 1 block = 32 Bytes, max is
    64 * 32 Bytes)


Part of packet (by definition)
* Length: 2 Bytes (00000xxxxx)
* Dest MAC: 6 Bytes
* Src MAC: 6 Bytes
* Start time
* End time

Part of packet (by block)
* Length: 2 Bytes + Dest MAC: 6 Bytes
* Time stamp: 8 bytes
* Src MAC: 2 garbage byte + 6 Bytes
* Data payload all 1's for now for the rest of the bits (at least 8 bytes)

*/

define IDLE 4'b0000
define LENGTH_DMAC_FST 4'b0001
define LENGTH_DMAC_SND 4'b0010
define TIME_FST 4'b0011
define TIME_SND 4'b0100
define SMAC_FST 4'b0101
define SMAC_SND 4'b0110
define PAYLOAD 4'b0111



module packet_val #(
    parameter PACKET_CNT = 1023,
    BLOCK_SIZE = 32,
    META_WIDTH = 32
) (
    input logic clk,
    input logic reset,

    // From crossbar
    input logic   [ META_WIDTH  -1  :  0  ] egress_in,
    input logic                egress_in_en,

    // From interface
    input logic egress_in_ack,

    // To interface
```

```
  output logic   [ 31  :  0  ] egress_out
);
 /* verilator lint_off UNUSED */
 logic   [ META_WIDTH  -1  :  0  ] next_meta;
 logic   [ META_WIDTH  -1  :  0  ] meta;

 logic   [ $clog2  (  PACKET_CNT  )  -1  :  0  ] start_idx; // The first element
 logic   [ $clog2  (  PACKET_CNT  )  -1  :  0
     ] end_idx; // One pass the last element
 logic meta_ready;
 logic   [ 15  :  0  ] remaining_length;
 logic   [ 15  :  0  ] next_remaining_length;
 logic   [ 47  :  0  ] next_DMAC, DMAC;
 logic   [ 5  :  0  ] length_in_blocks;
 logic   [ 1  :  0  ] port_number;
 logic   [ 31  :  0  ] start_time, next_start_time;
 logic   [ 31  :  0  ] delta;

 logic   [ 3  :  0  ] state, next_state;
 // logic [15:0] temp_length;
 // assign temp_length = (egress_in[31:16] >> 5);
 // assign length_in_blocks = temp_length[5:0];

 // SMAC_FST, SMAC_SND Are not used

 always_comb begin
   length_in_blocks = egress_in  [ 26  :  21  ];
   delta = (egress_in - start_time);
   case (state)
     IDLE: begin
       next_state          = LENGTH_DMAC_FST;
       next_meta  [ 31  :  28  ]     = 0;
       next_meta  [ 21  :  0  ]      = 0;
       meta_ready          = 0;
       next_meta  [ 27  :  22  ] = length_in_blocks;
       next_start_time     = 0;

       next_remaining_length = egress_in  [  31  :  16  ] - 4;
       next_DMAC  [ 47  :  32  ]        = egress_in  [  15  :  0  ];
       next_DMAC  [ 31  :  0  ] = DMAC  [  31  :  0  ];
     end //START
     LENGTH_DMAC_FST: begin
       next_state = LENGTH_DMAC_SND;
       meta_ready = 0;
       next_DMAC  [ 31  :  0  ] = egress_in;
       next_DMAC  [ 47  :  32  ] = DMAC  [  47  :  32  ];
       next_meta = meta;
       next_remaining_length = remaining_length - 4;
       next_start_time = start_time;
     end // LENGTH_DMAC_FST
     LENGTH_DMAC_SND: begin
       //next_remaining_length = egress_out[27:22];
       next_remaining_length = remaining_length - 4;
```

```
      next_state = TIME_FST;
      next_meta  [ 31  :  30  ] = meta  [  31  :  30  ];
      next_meta  [ 29  :  28  ] = port_number;
      next_meta  [ 27  :  0  ] = meta  [  27  :  0  ];
      next_start_time = egress_in;
      next_DMAC = DMAC;
      meta_ready = 0;
    end // LENGTH_DMAC_SND
    TIME_FST: begin
      next_state = TIME_SND;
      next_meta  [ 21  :  0  ] = delta  [  21  :  0  ];
      next_meta  [ 31  :  22  ] = meta  [  31  :  22  ];
      meta_ready = 0;
      next_remaining_length = remaining_length - 4;
      next_start_time = start_time;
      next_DMAC = DMAC;
    end // TIME_FST
    TIME_SND: begin
      next_state = SMAC_FST;
      next_remaining_length = remaining_length - 4;
      next_meta = meta;
      meta_ready = 0;
      next_start_time = start_time;
      next_DMAC = DMAC;
    end // TIME_SND
    SMAC_FST: begin
      next_state = SMAC_SND;
      next_meta  [ 31  :  30  ] = egress_in  [  1  :  0  ];
      next_meta  [ 29  :  0  ] = meta  [  29  :  0  ];
      meta_ready = 0;
      next_remaining_length = remaining_length - 4;
      next_start_time = start_time;
      next_DMAC = DMAC;
    end //SMAC_FST
    SMAC_SND: begin
      next_state = PAYLOAD;
      next_remaining_length = remaining_length - 4;
      next_meta = meta;
      meta_ready = 0;
      next_start_time = start_time;
      next_DMAC = DMAC;
    end //SMAC_SND
    PAYLOAD: begin
      if (remaining_length > 4) begin
        next_remaining_length = remaining_length - 4;
        next_state          = PAYLOAD;
        meta_ready          = 0;
        next_meta           = meta;
        next_start_time = start_time;
        next_DMAC           = DMAC;
      end // remaining_length > 0
              else begin
        next_remaining_length = remaining_length;
        next_state = IDLE; //UNDER QUESTION (GREATER THAN 0 OR GREATER THAN 1)
        meta_ready = 1;
        next_meta = meta;
        next_start_time = start_time;
        next_DMAC     = DMAC;
      end // else remaining_length < 0
    end // SRC_PAYLOAD
    default: begin
      meta_ready = 0;
    end

  endcase //end case

end // end always_comb


always_ff @(posedge clk) begin
  if (reset) begin
    start_idx <= 0;
    end_idx  <= 0;
    state    <= IDLE;
  end //if reset
      else begin

    /* packet->meta, the input*/
    if (egress_in_en) begin
      if (next_state == IDLE) begin
        end_idx <= (end_idx != PACKET_CNT - 1) ? end_idx + 1 :0;
      end
      state <= next_state;
      remaining_length <= next_remaining_length;
      start_time <= next_start_time;
      meta <= next_meta;
      DMAC <= next_DMAC;
    end //meta_en

    /* Output */
    if (egress_in_ack && (start_idx != end_idx)) begin

      start_idx <= (start_idx != PACKET_CNT - 1) ? start_idx + 1 :0;

    end // egress_in_ack
  end // not reset

end //always_ff


simple_dual_port_mem #(
    .MEM_SIZE (PACKET_CNT),
    .DATA_WIDTH(32)
) meta_mem (
    .clk(clk),
    .ra(start_idx),
      .wa(end_idx),
      .d(meta),
      .q(egress_out),
      .write(meta_ready)
  );

  mac_to_port mac_to_port_0 (
      .MAC(DMAC),
      .port_number(port_number)
  );

endmodule
```

12.4.5 Hardware-Software Interface

```
module hw_sw_interface(
    input logic clk,

    // From sw
    input logic     reset,
    input logic   [  31   :  0   ] writedata,
    input logic     write,
    input logic     read,
    input logic   [     2   :  0   ] address,
    input logic     chipselect,

    // From egress
    input logic   [  31   :  0   ] interface_in,

    // To sw
    output logic   [  31   :  0   ] readdata,

    // To packet_val/ingress
    // Change to 4 bits later
    output logic interface_out_en,
    output logic   [  31   :  0   ] interface_out,

    // Experimenting
    output logic experimenting,

    // Special case: Because we're polling but not handling interrupt
    // we need to acknowledge that this metadata is consumed by the software.
    // This is the only ack in our program.
    // Change to 4 bits later
    output logic interface_out_ack
);

  logic   [  31   :  0   ] ctrl;

  always_comb begin
    experimenting = (ctrl == 2);
  end

  always_ff @(posedge clk) begin
    if (reset) begin
      ctrl <= 32'h0;
      readdata <= 32'h0;
      interface_out_en <= 0;
      interface_out_ack <= 0;
      interface_out <= 0;
    end else begin
      if (chipselect && write) begin
        case (address)
          3'h0: begin
            ctrl <= writedata;
          end
          3'h1: begin
            interface_out_en <= 1;
            // interface_out_en[0] <= 1;
            interface_out <= writedata;
          end
          3'h2: begin
            // interface_out_en[1] <= 1;
            interface_out <= writedata;
          end
          3'h3: begin
            // interface_out_en[2] <= 1;
            interface_out <= writedata;
          end
          3'h4: begin
            // interface_out_en[3] <= 1;
            interface_out <= writedata;
          end
          default: begin
          end
        endcase
      end else begin
        // interface_out_en[0] <= 0;
        // interface_out_en[1] <= 0;
        // interface_out_en[2] <= 0;
        // interface_out_en[3] <= 0;
        interface_out_en <= 0;
      end
      if (chipselect && read) begin
        readdata <= interface_in;
        case (address)
          3'h1: begin
            readdata <= interface_in;
            // interface_out_ack[0] <= 1;
            interface_out_ack <= 1;
          end
          3'h2: begin
            readdata <= interface_in;
            // interface_out_ack[1] <= 1;
          end
          3'h3: begin
            readdata <= interface_in;
            // interface_out_ack[2] <= 1;
          end
          3'h4: begin
            readdata <= interface_in;
            // interface_out_ack[3] <= 1;
          end
          default: begin
          end
        endcase
      end else begin
        interface_out_ack <= 0;
        // interface_out_ack[0] <= 0;
        // interface_out_ack[1] <= 0;
        // interface_out_ack[2] <= 0;
        // interface_out_ack[3] <= 0;
      end
    end
  end

endmodule
```

12.4.6 Ingress

```

define IN_IDLE 4'b0000
define IN_0 4'b0001
define IN_1 4'b0010
define IN_2 4'b0011
define IN_3 4'b0100
define IN_4 4'b0101
define IN_5 4'b0110
define IN_6 4'b0111
define IN_7 4'b1000


define IDLE 2'b00
define SEND_META_2_CMU 2'b01
define GET_NEXT_ADDR 2'b10
define SEND_PACKET 2'b11

module ingress (
       input logic         clk,
   input logic      reset,
       input logic     [  31  :  0  ] packet_in, // From packet gen, getting packet
       input logic         packet_en, // From packet gen, meaning that we should start recving a packet segment
       //input logic       new_packet_en, // From packet gen, meaning that we should start recving a new packet

       input logic   [  1  :  0  ]   sched_sel, // From scheduler, the voq to dequeue the packet
       input logic                sched_done, // (actually SCHED_ENABLE) From scheduler, meaning that we should start sending a packet segment

       // output logic     sched_en, // To sch
       output logic  [  31  :  0  ]   packet_out, // To crossbar
       output logic        packet_out_en // To crossbar
);

       //logic [2:0] input_counter;
       logic   [  2  :  0  ] send_cycle_counter, next_send_cycle_counter;

       logic   [  5  :  0  ] next_remaining_packet_length, remaining_packet_length;
       logic   [  31  :  0  ] temp_packet_in;
       /* verilator lint_off UNOPTFLAT */
       logic          alloc_en;
       logic   [  47  :  0  ] d_mac, next_d_mac;
       logic   [  31  :  0  ] packet_start_time_logic;
       logic   [  3  :  0  ] in_state, next_in_state;
       /* verilator lint_on UNOPTFLAT */

       logic   [  12  :  0  ] curr_d_write, curr_d_read;
       logic      voq_enqueue_en;
       logic   [  1  :  0  ] voq_enqueue_sel, port_number;
```

```
    logic       first_packet;



    logic  [ 9  :  0  ]    alloc_addr;
    logic  [ 9  :  0  ]    meta_in, meta_out;
    assign meta_in = (first_packet == 1) ? 10'b1 :alloc_addr;
    /* States */

    logic   [ 1  :  0  ] out_state, next_out_state;

    /* time */
    logic   [  31  :  0  ] curr_time;

    logic   [  12  :  0  ] start_addr;
    /* verilator lint_off UNOPTFLAT */
    logic   [  12  :  0  ] start_addr_reg;
    /* verilator lint_on UNOPTFLAT */

    /* verilator lint_off UNUSED */
    assign start_addr = {3'b0, alloc_addr} << 2;
    logic   [  31  :  0  ] offset_addr;
    assign offset_addr = ((curr_time - packet_start_time_logic) % 8);


    logic free_en;
    logic   [ 9  :  0  ] free_addr, next_free_addr, voq_meta_out,
       voq_meta_out_reg, free_addr_reg, next_free_addr_reg;
    logic   [ 3  :  0  ] is_empty, is_full;
    logic       voq_dequeue_en;
    //logic [1:0] voq_dequeue_sel;

    logic   [  12  :  0  ] data_read_addr;
    assign data_read_addr = {3'b000, voq_meta_out} << 2;

    always_comb begin
           case (in_state)
                  IN_IDLE: begin
                         next_remaining_packet_length = 0;
                         next_d_mac   [  47  :  32  ]                  = 0;
                         next_d_mac   [  31  :  0  ]                   = 0;
                         temp_packet_in                  = 0;
                         alloc_en                                  = 0;
                         packet_start_time_logic  = curr_time;
                         next_in_state                       = IN_0;
                         curr_d_write                        = 0;
                         voq_enqueue_en           = 0;
                         voq_enqueue_sel          = 0;
                   end

                  IN_0: begin
                         next_remaining_packet_length = packet_in  [  26  :  21  ];
```

```
                    next_d_mac  [  47  :  32  ]                 = packet_in  [
                       15  :  0  ];
                    next_d_mac  [  31  :  0  ]                  = 0;
                    temp_packet_in                  = packet_in;
                    alloc_en                                = 1;
                    packet_start_time_logic  = curr_time;
                    next_in_state                       = IN_1;
                    curr_d_write                        = (first_packet == 1) ?
                       13'b1 :start_addr;
                    voq_enqueue_en            = 0;
                    voq_enqueue_sel           = port_number;
                end
                IN_1: begin
                    next_remaining_packet_length = remaining_packet_length;
                    next_d_mac  [  47  :  32  ]                 = d_mac  [  47
                       :  32  ];
                    next_d_mac  [  31  :  0  ]                  = packet_in;
                    temp_packet_in                      = packet_in;
                    alloc_en                                = 0;
                    next_in_state                       = IN_2;
                    curr_d_write                        = start_addr+1;
                    start_addr_reg            = start_addr;
                    voq_enqueue_en            = 0;
                    voq_enqueue_sel           = port_number;
                end
                IN_2: begin
                    next_remaining_packet_length = remaining_packet_length;
                    next_d_mac                = d_mac;
                    temp_packet_in                      = curr_time;
                    alloc_en                                = 0;
                    next_in_state                       = IN_3;
                    curr_d_write                        = start_addr+2;
                    voq_enqueue_en            = 1;
                    voq_enqueue_sel           = port_number;

                end
                IN_3: begin
                    next_remaining_packet_length = remaining_packet_length;
                    next_d_mac                = d_mac;
                    temp_packet_in                      = packet_in;
                    alloc_en                                = 0;
                    next_in_state                       = IN_4;
                    curr_d_write                        = start_addr+3;
                    voq_enqueue_en            = 0;
                    voq_enqueue_sel           = port_number;
                end
                IN_4: begin
                    next_remaining_packet_length = remaining_packet_length;
                    next_d_mac                              = d_mac;
                    temp_packet_in                      = packet_in;
                    alloc_en                                = 0;
                    next_in_state                       = IN_5;
                    curr_d_write                        = start_addr+4;
```

```
                        voq_enqueue_en          = 0;
                        voq_enqueue_sel         = port_number;
                end
                IN_5: begin
                        next_remaining_packet_length = remaining_packet_length;
                        next_d_mac                                  = d_mac;
                        temp_packet_in                       = packet_in;
                        alloc_en                                    = 0;
                        next_in_state                        = IN_6;
                        curr_d_write                         = start_addr+5;
                        voq_enqueue_en          = 0;
                        voq_enqueue_sel         = port_number;
                end
                IN_6: begin
                        next_remaining_packet_length = remaining_packet_length;
                        next_d_mac                                  = d_mac;
                        temp_packet_in                       = packet_in;
                        alloc_en                                    = 0;
                        next_in_state                        = IN_7;
                        curr_d_write                         = start_addr+6;
                        voq_enqueue_en          = 0;
                        voq_enqueue_sel         = port_number;
                end
                IN_7: begin
                        voq_enqueue_en          = 0;
                        voq_enqueue_sel         = port_number;
                        if(remaining_packet_length > 1) begin
                                next_in_state               = IN_7;
                        end else begin
                                next_in_state                             = IN_0;
                        end

                        next_remaining_packet_length = remaining_packet_length - 1;
                        next_d_mac                                  = d_mac;
                        temp_packet_in                       = packet_in;

                        if((curr_time - packet_start_time_logic) % 8 == 0) begin
                                alloc_en                                    = 1;
                                curr_d_write                         = start_addr;
                        end else if ((curr_time - packet_start_time_logic) % 8 ==
                           7) begin
                                alloc_en = 0;
                                curr_d_write                         = start_addr_reg
                                   + 7;
                                //start_addr_reg          = start_addr;
                        end else begin
                                alloc_en = 0;
                                curr_d_write                         = start_addr +
                                   offset_addr  [  12  :  0
                                   ];
                        end


                end
                default: begin end
            endcase

            case (out_state)
                IDLE: begin
                    free_en = 1'b0;
                    free_addr = voq_meta_out;
                    voq_meta_out_reg = meta_out;
                    next_send_cycle_counter = 0;
                    next_out_state = (sched_done) ? SEND_META_2_CMU :IDLE; //
                        no packet or no decision
                    packet_out_en = 0;
                end
                SEND_META_2_CMU: begin
                    free_en = 1'b1;
                    free_addr = voq_meta_out;
                    curr_d_read = data_read_addr;

                    next_send_cycle_counter = 1;
                    next_out_state = GET_NEXT_ADDR;

                    packet_out_en = 0;
                end
                GET_NEXT_ADDR: begin
                    free_en = 1'b0;
                    free_addr = voq_meta_out;
                    free_addr_reg = next_free_addr;
                    curr_d_read = data_read_addr + {9'b0, send_cycle_counter};
                    next_send_cycle_counter = 2;
                    next_out_state = SEND_PACKET;

                    packet_out_en = 1;
                end
                SEND_PACKET: begin
                    packet_out_en = 1;

                    free_en = 1'b0;
                    free_addr = voq_meta_out;
                    curr_d_read = data_read_addr + {9'b0, send_cycle_counter};
                    next_send_cycle_counter = (send_cycle_counter == 7) ? 0 :
                        (send_cycle_counter + 1);
                    if (send_cycle_counter == 7 && next_free_addr_reg !=
                        10'b0) begin
                        next_out_state = SEND_META_2_CMU;
                        voq_meta_out_reg = next_free_addr_reg;
                    end else if (send_cycle_counter == 7 && next_free_addr_reg
                        == 10'b0) begin
                        next_out_state = IDLE;
                    end else begin
                        next_out_state = SEND_PACKET;
                    end
                end
                default: begin
                end
            endcase
    end

    assign next_free_addr_reg = free_addr_reg;


    always_ff @(posedge clk) begin
            if (reset) begin
                in_state <= IN_IDLE;
                out_state <= IDLE;
                curr_time <= 0;
                remaining_packet_length <= 0;
                first_packet         <= 0;
                send_cycle_counter <= 0;
            end // if reset
            else begin
                /* Time driver*/
                curr_time <= curr_time + 1;

                /* Input */
                if (packet_en) begin
                    if (in_state == IN_7) begin
                        first_packet <= 0;
                    end
                    in_state <= next_in_state;
                    d_mac <= next_d_mac;
                    remaining_packet_length <= next_remaining_packet_length;
                end

                out_state <= next_out_state;
                send_cycle_counter <= next_send_cycle_counter;

                voq_meta_out <= voq_meta_out_reg;


            end
    end

    vmu #(.PACKET_CNT(1024), .EGRESS_CNT(4)) voq_mu(
            // Input
            .clk(clk),
            .voq_enqueue_en(voq_enqueue_en), .voq_enqueue_sel(voq_enqueue_sel),
            .voq_dequeue_en(sched_done), .voq_dequeue_sel(sched_sel),
            .meta_in(meta_in),
            // Output
            .meta_out(meta_out),
            .is_empty(is_empty), .is_full(is_full)
    );

    cmu ctrl_mu(
            // Input when inputting
```

```
                .clk(clk),
        .reset(reset),
                // From input
                .remaining_packet_length(remaining_packet_length), // in blocks
                .alloc_en(alloc_en), // writing data in
                // From output
                .free_addr(free_addr),
                .free_en(free_en),
        // To input
                .alloc_addr(alloc_addr),
                // To output
                .next_free_addr(next_free_addr) // retrieve the "next" of the free_addr
        );

        simple_dual_port_mem #(.MEM_SIZE(1024*8), .DATA_WIDTH(32)) dmem(
                .clk(clk),
                .ra(curr_d_read), .wa(curr_d_write),
                .d(temp_packet_in),  .q(packet_out),
                .write(packet_en)
        );

        mac_to_port mac_to_port_2(.MAC(d_mac), .port_number(port_number));


endmodule
```

12.4.7 Scheduler

```
module sched (
    input logic clk,
    input logic sched_en,
    input logic   [  3  :  0  ] is_busy,
    input logic   [  7  :  0  ] busy_voq_num,
    input logic   [  15  :  0  ] voq_empty,
    input logic policy, // We have doubly RR, or priority based, can be controlled by
        software
    input logic   [  7  :  0  ] prio,
    output logic   [  3  :  0
        ] sched_sel_en, // passed by to ingress, to know which ingress should dequeue
    output logic   [  7  :  0
        ] sched_sel // passed by to ingress, to know which voq to dequeue
);

  /*
  Some design choices:
  * Do we want the time to return a scheduling decision to be deterministic or random
      (btw 1 and 4)
  * Should each busy port to just start transmitting without waiting for the
      scheduling decision
  * Should the schduler decision the scheduling decision for this cycle or next cycle
  * RR on the ingress or egress side, or both
  * Should we try to do all combinator?
  * Do we use 1 voq_to_pick or 4 voq_to_pick? (1 since it's going to take 4 cycles
      anyway)
  */

  /*
  Some principles:
  * We don't want egress 1 to always recv packet from ingress 0; we don't want
      ingress 0 to always send to egress 2
  * nested loop fully in combinator logic is too expensive; instead, we do the outer
      for loop sequentially, and the inner 4 loop in comb logic
  RR policy:
    * ingress RR: Start_ingress_idx proceed in each cycle of ASSIGN_NEW (or one pass
        who-ever gets to select first in this cycle)
    * egress RR: Each start_voq_num is first_non_empty_num of this cycle + 1;
    * fewer first: Prioritze queue with only one non-empty voq.
  */

  /*
  Notice:
  * Beware of the possiblity of input (voq_empty for example) change during the
      process.
    - Possibly, use some local variables to save the inputs.
    - Or, let the ingress be in charge of only updating the signal when sched_en.
  * Busy ports need to be handled first.
  * All the data need to be prepared before sched_enable. So at T-1 prepare input,
      and at T sched_enable.
  */
```

```

// If the scheduler is in the process of assigning new packet
// 0: not assigning; 1~4: assigning. 5 enable 6 is /enable
logic   [  2  :  0  ] assigning_new;

logic   [  3  :  0  ] ingress_enable; // the enable signal ready to be passed to sched_sel_en when ingress_done = 4'b1111

// For RR
logic   [  1  :  0  ] start_ingress_idx; // Which ingress has the highest priority in this cycle
logic   [  7  :  0  ] start_voq_num; // Which egress has the highest priority in this cycle, for each ingress.

logic   [  1  :  0  ] curr_ingress_idx; // Current ingress
logic   [  2  :  0  ] curr_in_2;
logic   [  3  :  0  ] curr_in_4;
logic   [  3  :  0  ] voq_picked; // is the voq/egress picked?
logic no_available_voq; // Is the voqs/egress of the current ingress all empty/occupied by other?
logic   [  1  :  0  ] voq_to_pick; // What is the voq_to_pick for the current ingress
logic   [  3  :  0  ] busy_egress_mask;

logic   [  2  :  0  ] i;
logic   [  1  :  0  ] busy_port;

always_comb begin
  busy_egress_mask = 0; // This is important: busy_egress_mask need a way to start with all unoccupied.
  for (i = 0; i < 4; i = i + 1) begin
    if (is_busy  [  i  [  1  :  0  ]  ] == 1'b1) begin
      busy_egress_mask  [  busy_voq_num  [  (  i     <<     1)     +  :     2  ]  ] = 1'b1;
    end
  end
  curr_in_2 = {1'b0, curr_ingress_idx} << 1;
  curr_in_4 = {2'b0, curr_ingress_idx} << 2; // * 4 is << 2
end

initial begin
  start_ingress_idx = 0;
  start_voq_num = 0;
end

always_ff @(posedge clk) begin

  if (sched_en) begin
    // If we begin to schedule
    // reset sched_sel_en
    sched_sel_en <= 0;
    // all the busy ingress ports are automatically assigned.
    ingress_enable <= 0;
    // start to assign ports for non-empty
```

```
   assigning_new <= 1;
   voq_picked <= busy_egress_mask;
   curr_ingress_idx <= start_ingress_idx; // Start with start_ingress_idx
  end else if (assigning_new == 5) begin // alternatively, if we manage to go back to start_ingress_idx
   // If all are assigned, we're going start enabling
   sched_sel_en <= ingress_enable;
   // Nex time it should start with another index.
   start_ingress_idx <= (start_ingress_idx == 3) ? 0 :start_ingress_idx + 1;
   assigning_new <= 6;
  end else if (assigning_new == 6) begin
   sched_sel_en <= 0;
  end else if (assigning_new >= 1 && assigning_new <= 4) begin
   curr_ingress_idx <= (curr_ingress_idx == 3) ? 0 :curr_ingress_idx + 1;
   assigning_new <= assigning_new + 1;
   if (!is_busy [ curr_ingress_idx ]) begin
    if (!no_available_voq) begin
     ingress_enable [ curr_ingress_idx ] <= 1'b1;
     voq_picked [ voq_to_pick ] <= 1'b1;
     sched_sel [ curr_in_2  + :  2 ] <= voq_to_pick;
     start_voq_num [ curr_in_2  + :  2 ] <=
      (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.
    end
   end else begin
    busy_port <= busy_voq_num [ curr_in_2  + :  2 ];
    ingress_enable [ curr_ingress_idx ] <= 1'b1;
    sched_sel [ curr_in_2  + :  2 ] <= busy_port;
    voq_picked [ busy_port ] <= 1'b1;
   end
  end
 end

 // pick_voq will pick return the current ingress's first non empty voq to dequeue from.
 pick_voq pv (
  .start_voq_num(start_voq_num [ curr_in_2  + :  2 ]),
  .voq_empty(voq_empty [ curr_in_4  + :  4 ]),
  .voq_picked(voq_picked),
  .no_available_voq(no_available_voq),
  .voq_to_pick(voq_to_pick),
  .policy(policy),
  .prio(prio)
 );
endmodule
```

12.4.8 Packet Generation

```


/*

Packet metadata definition:
* src port: 2 bits
* dest port: 2 bits
* length: 12 bit (Packet size is 32 Bytes * length, min is 1 block = 32 Bytes, max is
    64 * 32 Bytes)


Part of packet (by definition)
* Length: 2 Bytes (00000xxxxx)
* Dest MAC: 6 Bytes
* Src MAC: 6 Bytes
* Start time
* End time

Part of packet (by block)
* Length: 2 Bytes + Dest MAC: 6 Bytes
* Time stamp: 8 bytes
* Src MAC: 2 garbage byte + 6 Bytes
* Data payload all 1's for now for the rest of the bits (at least 8 bytes)

*/

define IDLE 4'b0000
define LENGTH_DMAC_FST 4'b0001
define LENGTH_DMAC_SND 4'b0010
define TIME_FST 4'b0011
define TIME_SND 4'b0100
define SMAC_FST 4'b0101
define SMAC_SND 4'b0110
define PAYLOAD 4'b0111



module packet_gen #(
    parameter PACKET_CNT = 1024,
    BLOCK_SIZE = 32,
    META_WIDTH = 32
) (
    input logic              clk,
    input logic              reset,

    input logic   [ META_WIDTH -1  :  0  ] packet_gen_in,
    input logic              packet_gen_in_en,
    input logic              experimenting,

        // To Ingress
    output logic             packet_gen_out_en,
  output logic   [ 31  :  0  ]       packet_gen_out
);
 logic   [  META_WIDTH  -1   :   0  ] meta_out;

 logic   [  $clog2  (  PACKET_CNT  )  -1   :   0  ] start_idx; // The first element
 logic   [  $clog2  (  PACKET_CNT  )  -1   :   0
    ] end_idx; // One pass the last element

 logic   [  15   :   0  ] remaining_length;
 logic   [  15   :   0  ] next_remaining_length;
 logic   [  47   :   0  ] DMAC, SMAC; // A lot of registers, be really careful!
 logic   [  15   :   0  ] length_in_bytes;
 logic   [  3   :   0  ] state, next_state;
 assign length_in_bytes = meta_out  [  27   :   22  ] * 32;
 // SMAC_FST, SMAC_SND Are not used


 always_comb begin
   case (state)
     IDLE: begin
       next_state = LENGTH_DMAC_FST;
       packet_gen_out = 0;
       packet_gen_out_en = 0;
       next_remaining_length = remaining_length;
     end //START
     LENGTH_DMAC_FST: begin
       next_state = LENGTH_DMAC_SND;
       packet_gen_out    = {length_in_bytes, DMAC  [  47   :   32  ]};
       packet_gen_out_en    = 1;
               next_remaining_length = length_in_bytes - 4;
     end // LENGTH_DMAC_FST
     LENGTH_DMAC_SND: begin
       next_state          = TIME_FST;
       packet_gen_out              = DMAC  [  31   :   0  ];
       packet_gen_out_en       = 1;
       next_remaining_length = remaining_length - 4;
     end // LENGTH_DMAC_SND
     TIME_FST: begin
       next_state = TIME_SND;
       packet_gen_out    = {10'b0, meta_out  [  21   :   0  ]};
       packet_gen_out_en = 1;
       next_remaining_length = remaining_length - 4;
     end // TIME_FST
     TIME_SND: begin
       next_state = SMAC_FST;
       packet_gen_out    = 32'b0;
       packet_gen_out_en = 1;
       next_remaining_length = remaining_length - 4;
     end // TIME_SND
     SMAC_FST: begin
       next_state = SMAC_SND;
       packet_gen_out    = {16'b0, SMAC  [  47   :   32  ]};
       packet_gen_out_en = 1;
```

```
      next_remaining_length = remaining_length - 4;
    end //SMAC_FST
    SMAC_SND: begin
      next_state = PAYLOAD;
      packet_gen_out    = SMAC  [  31  :  0  ];
      packet_gen_out_en = 1;
      next_remaining_length = remaining_length - 4;
    end //SMAC_SND
    PAYLOAD: begin
        if (remaining_length > 4) begin
                      next_state = PAYLOAD;
        end else begin
            next_state = LENGTH_DMAC_FST;
        end

                next_remaining_length = remaining_length - 4;
        packet_gen_out = ~32'b0;
        packet_gen_out_en = 1;

    end
    default: begin
      packet_gen_out_en = 0;
      packet_gen_out = 0;
      next_remaining_length = remaining_length;
      next_state = state;

    end
  endcase //end case

end // end always_comb


always_ff @(posedge clk) begin
  if (reset) begin
    start_idx <= 0;
    end_idx  <= 0;
    state    <= IDLE;
    //packet_gen_out_en = 0;
  end //if reset
      else begin
    if (packet_gen_in_en) begin
      end_idx <= (end_idx == 1023) ? 0 :end_idx + 1;

    end //packet_gen_in_en
    if (experimenting) begin
      if (next_state == IDLE) begin
        start_idx <= (start_idx != 1023) ? start_idx + 1 :0;
      end
      state <= next_state;
      remaining_length <= next_remaining_length;
    end // experimenting
  end // not reset

  end //always_ff


  simple_dual_port_mem #(
      .MEM_SIZE (PACKET_CNT),
      .DATA_WIDTH(32)
  ) vmem (
      .clk(clk),
      .ra(start_idx),
      .wa(end_idx),
      .d(packet_gen_in),
      .q(meta_out),
      .write(packet_gen_in_en)
  );

  port_to_mac port_to_mac_0 (
      .port_number(meta_out  [ 29  :  28  ]),
      .MAC(DMAC)
  );
  port_to_mac port_to_mac_1 (
      .port_number(meta_out  [ 31  :  30  ]),
      .MAC(SMAC)
  );

endmodule
```

12.4.9 True Dual Port Memory

```

module true_dual_port_mem #(
   parameter MEM_SIZE = 1024, /* How many bits of memory in total, 1024 by default */
   parameter DATA_WIDTH = 16 /* How many bit of data per cycle, 16 by default */
) (
   input logic clk,
   input logic   [  $clog2  (  MEM_SIZE  )     -     1  :  0  ] aa, ab, /* Address */
   input logic   [  DATA_WIDTH     -     1  :  0  ] da, db,        /* input data */
   input logic wa, wb,                                            /* Write enable */
   output logic   [  DATA_WIDTH     -     1  :  0  ] qa, qb        /* output data */
);
  logic   [  DATA_WIDTH  -1  :  0  ] mem  [  MEM_SIZE     -     1  :  0  ];

  // First port
  always_ff @(posedge clk) begin
    if (wa) begin
      mem  [  aa  ] <= da;
      qa <= da;
    end else qa <= mem  [  aa  ];
  end

  // Second port
  always_ff @(posedge clk) begin
    if (wb) begin
      mem  [  ab  ] <= db;
      qb <= db;
    end else qb <= mem  [  ab  ];
  end
endmodule
```

12.4.10 VMU

```
/*
The virtual output queue management unit:
*/

/*
voq memory is divided into 16-bit chunks
and each of them represents a whole packet

Inside each chunk:
* an 10-bit address, which is the ctrl address of the first segment of the packet

The same packet in cmu (that needs 2 block) would be:
Address 0: 10'b1
*/

/*
The voq is implemented as a ring buffer. there are #egress voqs in each ingress.
Each voq can contain at most 1024 packets. (indexexd by the lower 10 bits of the
    memory)
The 2 higher bits represents which voq it is.

    [11, 10]     [9,8,7,6,5,4,3,2,1,0]
 ----voq_idx-----addr of 1st seg of packet------

*/


module vmu #(
    parameter PACKET_CNT = 1024, /* How many packets can there be in each VOQ, 1024 by
        default */
    parameter EGRESS_CNT = 4 /* How many egress there are; which is also how many voqs
        there are. */
) (
    input logic clk,
    input logic voq_enqueue_en,
    input logic   [  $clog2  (  EGRESS_CNT  )  -1  :  0  ] voq_enqueue_sel,
    input logic voq_dequeue_en,
    input logic   [  $clog2  (  EGRESS_CNT  )  -1  :  0  ] voq_dequeue_sel,
    input logic   [  $clog2  (  PACKET_CNT  )  -1  :  0
        ] meta_in, // The address to find the first address of the packet

    /* TODO: How many bits for meta_out? */
    output logic   [  $clog2  (  PACKET_CNT  )  -1  :  0  ] meta_out, // The content
        (first addr of the packet) saved for the dequeue packet
    output logic   [  EGRESS_CNT  -1  :  0  ] is_empty, // For scheduler
    output logic   [  EGRESS_CNT  -1  :  0
        ] is_full // For potential packet drop. If is_full, then drop the current
);
    logic   [  $clog2  (  PACKET_CNT  )  +1  :  0  ] start_idx  [  3  :  0
        ]; // first element
    logic   [  $clog2  (  PACKET_CNT  )  +1  :  0  ] end_idx  [  3  :  0
        ]; // one pass the last element

    logic   [  $clog2  (  EGRESS_CNT  )  :  0  ] i;
    logic   [  $clog2  (  EGRESS_CNT  )  -1  :  0  ] _i;

    always_comb begin
       for (i = 0; i < EGRESS_CNT; i = i + 1) begin
          _i = i  [  $clog2  (  EGRESS_CNT  )  -1  :  0  ];
          is_empty  [  _i  ] = (start_idx  [  _i  ] == end_idx  [  _i  ]);
          is_full  [  _i  ] = (start_idx  [  _i  ] == ((end_idx  [  _i
             ] == PACKET_CNT - 1) ? 0 :end_idx  [  _i  ] + 1));
       end
    end


    always @(posedge clk) begin
       if (voq_enqueue_en && !is_full  [  voq_enqueue_sel  ]) begin
          end_idx  [  voq_enqueue_sel  ] <= (end_idx  [  voq_enqueue_sel
             ] != PACKET_CNT - 1) ? end_idx  [  voq_enqueue_sel  ] + 1 :0;
       end

       if (voq_dequeue_en && !is_empty  [  voq_dequeue_sel  ]) begin
          start_idx  [  voq_dequeue_sel  ] <= (start_idx  [  voq_dequeue_sel
             ] != 0) ? start_idx  [  voq_dequeue_sel  ] + 1 :PACKET_CNT - 1;
       end
    end

       simple_dual_port_mem #(.MEM_SIZE(PACKET_CNT * EGRESS_CNT),
          .DATA_WIDTH($clog2(PACKET_CNT))) vmem
       (
             .clk(clk),
             .ra(start_idx  [  voq_dequeue_sel  ]), .wa(end_idx  [  voq_enqueue_sel
                ]),
             .d(meta_in),  .q(meta_out),
             .write(voq_enqueue_en)
       );

endmodule
```