PacketFilter: Hardware-Accelerated Ethernet Frame Filtering and Switch

CSEE W4840 - Embedded Systems - Spring 2025

Michael Grieco
Adwyck Gupta
Harry Zhang
{mag2346,ag5016,hz3000}@columbia.edu

1 INTRODUCTION

As modern computation requires increasingly higher speed networks to handle ever growing volumes of data, efficient use becomes critical. General-purpose software network switch are often ill-suited for the real-time filtering and routing of network packets due to their sequential nature and multitasking responsibilities. Hardware-accelerated network interface controllers, such as SmartNICs, aim to address these bottlenecks by offloading packet processing tasks to dedicated logic near the data source.

In this project, we aim to implement a high speed Ethernet switch as an FPGA-based SoC implemented with an Altera Cyclone V chip on the DE1-SoC board. Our primary design routes frames from four ingress ports to four egress ports by calculating the destination port using the Ethernet frame headers. The switch integrates with drop logic that allows the system to drop frames that have been waiting for too long or those it deems invalid with basic processing. To complement that, we implement back-pressure through the switch with the AXI-Stream protocol such that stalls ripple throughout the system and queues never overflow.

We modularize the design to improve the reusability and reconfigurability at different granularities. To verify the correctness and measure the performance, we wrap a hardware-based testbench around it that generates Ethernet frames at a configurable rate. In the end, while we were unable to deploy our system on an FPGA, we verified high performance with reliable statistics and concluded that our design would be able to efficiently reduce the volume of data sent to downstream processing elements.

2 SYSTEM ORGANIZATION

2.1 Protocols

The main functional design has four ingress ports that function as an AXI-Stream data sink and four egress ports that function as an AXI-Stream data source. All ports use the data (16-bit), valid, and last signals from the source and the ready signal from the sink. The system can handle delays between frames and as such, will exert back-pressure by de-asserting its ready signals.

The switch's main function is to validate the Ethernet frames and route the ingress frames to the appropriate egress ports based on their header values. It is able to validate and route streams of data that follow the Ethernet 2.0 standard, as Fig. 1 shows.

One assumption is that all packets have valid checksums. We will not increment an internal checksum to validate the frame check sequence (FCS) at the end of the frame. However, if this were a requirement, the system could easily calculate a checksum and validate it at the end of the frame. This integrates nicely with the existing drop logic as this new calculation unit could drive a drop signal which would trigger the same logic as with the timeout.

2.2 System distribution

Fig. 2 shows a top-level view of the system, including interfaces between hardware and software along with the various hardware blocks we will implement. The primary functional part is the filter around each ingress port and the switch fabric to route frames. The generator and receptors are part of the hardware-based test harness which can generate frames much faster than via software.

To modularize the design and improve the effectiveness of verification, we break down the system (excluding the test harness) into the input filter and the switch. The input filter takes in a raw stream of data and computes sideband information so that the input switch can route it to the appropriate egress port. This way, the switch is only responsible for scheduling frames with a pre-determined destination and is not responsible for validating packets. The streams use the AXI-Stream (AXIS) protocol with the data, valid, last, and ready signals. The sideband information is implemented on a 2-bit destination signal, which is an optional extension of the AXIS protocol. This sideband information is only present in the interface between the filter and the switch. As a result, the main design implements an interface whose signals are

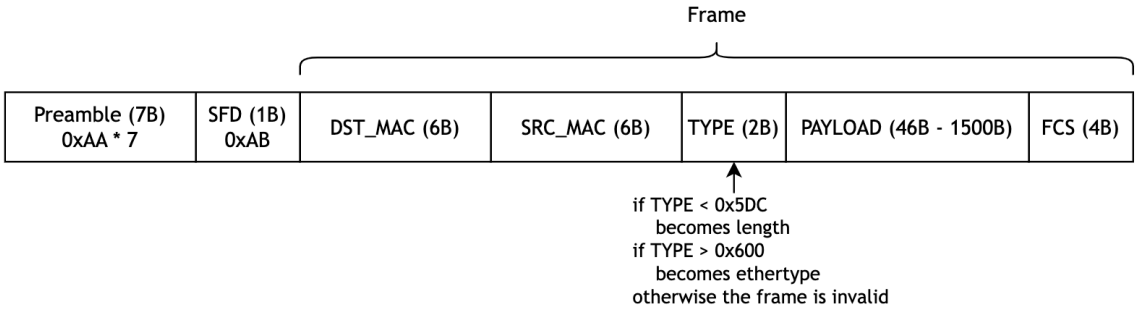

Figure 1: The applicable format of Ethernet frames our system can process.
[1]

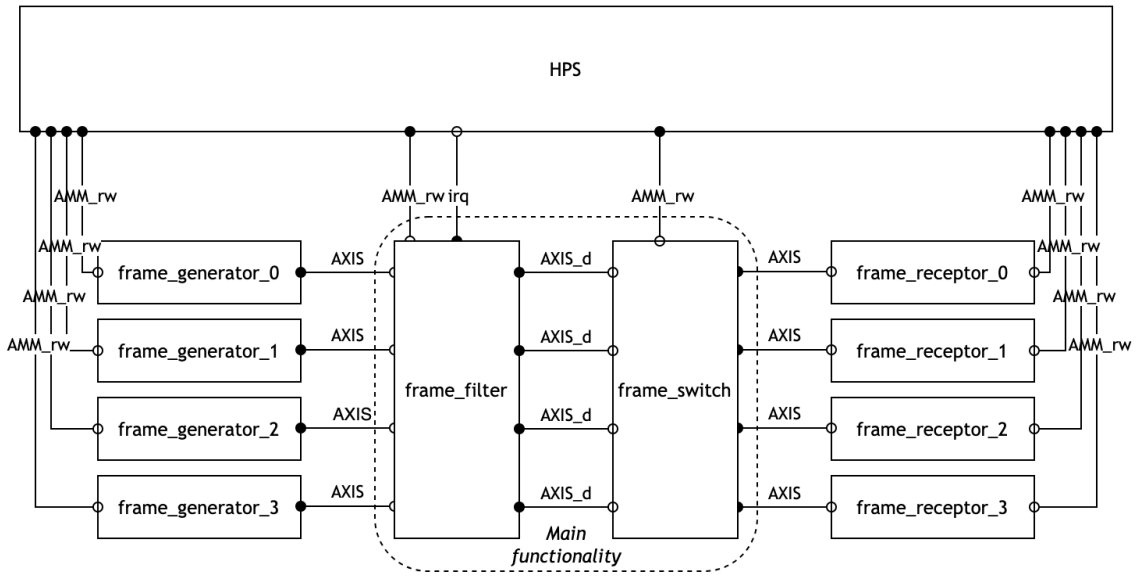

Figure 2: The breakdown of our system into hardware components that receive commands from software via the Avalon memory-mapped register interface. All hardware modules talk with the AXI-Stream protocol. Legend: filled dot is the producer; hollow dot is the consumer; AMM_rw is a read- and write-able Avalon register interface; AXIS is the AXI-Stream protocol with valid, 16-bit data, last, and ready signals; AXIS_d is the same as AXIS but with a two-bit dest signal.

entirely related to the stream structure, not the contents of the stream.

3 SYSTEM BREAKDOWN

This section describes the various blocks in our system.

3.1 Input filter

The function of the input filter is to extract sideband information for each Ethernet frame whilst also dropping frames which meet specific conditions. The input filter has four identical units, one for each ingress port. We elected to aggregate these units into one system-level submodule to simplify the register interface. Fig. 6 shows the filter structure for a single ingress port; the following description is with respect to a single one as well. We broke down the filtering into various state machines who coordinate to activate other units and control writing to and reading from input queues. This way, each piece has a more focused computation, and they can interact via signal passing instead of aggregating states into one large state machine.

Each of these blocks communicate with some variation of the AXI-Stream protocol. The data field represents frame and sideband data that is written to buffers. The valid and ready bits perform handshaking between submodules; though we

omit the ready bit from the consumer state machines because they have more deterministic behavior throughout a frame. All drop statuses are passed as a bit in the sideband signal, tuser.

Each ingress port has a corresponding enable bit in the software-writeable register, ingress_port_mask. If the software ever writes a '0' to the ingress port's enable bit, the input scanning units mask all activity on the input and the buffers drop a frame if it is currently ingressing. However, the frames that have already been stored in the queues are allowed to proceed through the system.

3.1.1 Input scanning. Each filter has an input state machine (input_fsm) which tracks progress through the stream relative to the Ethernet frame structure. It activates the computation units when their target field is active. Fig. 4 shows the state diagram. There are two recovery states, `FLUSH_FRAME` and `WAIT_DROPPED`, which allow the input FSM to return to idle when a frame ends prematurely (i.e., the last flag is asserted before getting through Ethernet headers) or when the processing units detect an invalid field. In the latter, the input FSM is also responsible for masking the frame (by masking the valid signal) that is still arriving even though it has been deemed invalid.

Additionally, the input state machine uses an enable signal from the filter's register file. If software writes a '0' to the ingress port's bit, the state machine masks all inputs so the ingress port is inactive.

The destination calculator (dest_calc) maps the destination MAC field received in the frame to a two-bit sideband field to pass alongside the stream to the switch. For simplicity, we implement this by taking the two least significant bits of the MAC address. However, in the future, this could be expanded to support more meaningful computations. Similarly, the type field validation (type_field_checker) simply verifies that the two bytes in the ethertype field are not in the invalid range (i.e., between `0x5DC` and `0x600`). It can extend to perform more thorough checking or activate a specific processing mechanism if it detects a recognized Ethertype. Both those units output an invalid signal which indicates to the input state machine to mask the current stream.

3.1.2 Input buffering. There are two queues that store data for the ingress frames. The frame buffer buffers the packets received from the ingress stream while the sideband FIFO stores metadata about each stream. Section 6 details the required number of memory blocks to implement these FIFO queues. This section details the organization of the allocated five blocks of memory. The frame FIFO maps one ingress AXIS packet to one entry in the FIFO. The memory blocks on the DE1-SoC FPGA chip store 20-bit addressable words. Since each ingress packet is 16 bits, we elected to pad it with four '0' bits instead of splitting packets across multiple addresses. In the future, we could implement parity bits for integrity checking; this is a crucial function for network switches.

The frame buffer stores many 16-bit packets for each frame in a FIFO that spans four 512x20 blocks of memory; it can store a total of 2048 twenty-bit words (thus the addresses are eleven bits wide, which extend to twelve to allow for simultaneous reading and writing). We chose four blocks to be able to store more than two full frames while keeping the number as a power of 2 for address simplicity. In storing more than two full frames, we can accommodate any back-pressure that the egress puts on the filter, even if it is within a single frame transmission. To avoid having to put intra-frame back-pressure on the input filter's ingress port, we use an almost full signal on the frame FIFO that it asserts when it cannot store a full frame (i.e., there are less than 759 open slots in the queue). Therefore, when the FIFO broadcasts not almost-full, we can be certain that it can buffer the worst-case size frame.

The frame FIFO also supports dropping frames via controllable cursors. When a new frame starts, the frame buffer latches the current write address. If it sees that the global drop signal is asserted (from the scanning section), it resets the write pointer to the saved address, which is at the tail of the previous frame. One assumption is that a frame will not be dropped if the filter is already scanning the payload. This is reasonable because all drop logic computes over the header fields. Hence, so long as frames do not issue to the switch until the input state machine asserts the payload scan signal (scan_payload), a frame will never be dropped during its payload. With this assumption, we do not have to worry about the case where resetting the write pointer to the head of a frame already being read would push the write pointer beyond the read pointer (thus making the FIFO appear almost full even though it has no valid data).

The sideband buffer stores several 20-bit queue entries for each frame in a single 512x20 block of memory. The first entry stores the computed destination (two bits) along with the address of the head of the frame (twelve bits). To be able to reproduce the AXIS last signal when issuing to the switch, the requestor needs to know how many 16-bit packets are in the frame. The maximum number is 759 as mentioned in Section 2.1, so this counter must be 10 bits wide. The buffer internally increments this count based on status signals from the input state machine. As with the frame buffer, there must be drop logic to recover when the frame is determined to be invalid.

3.1.3 Switch requests. As the two FIFO queues are separate, they need synchronization to ensure that the next frame's sideband entries are read only once the current frame has cleared the input filter. The switch requestor (switch_requestor) takes care of this by matching sideband information to the

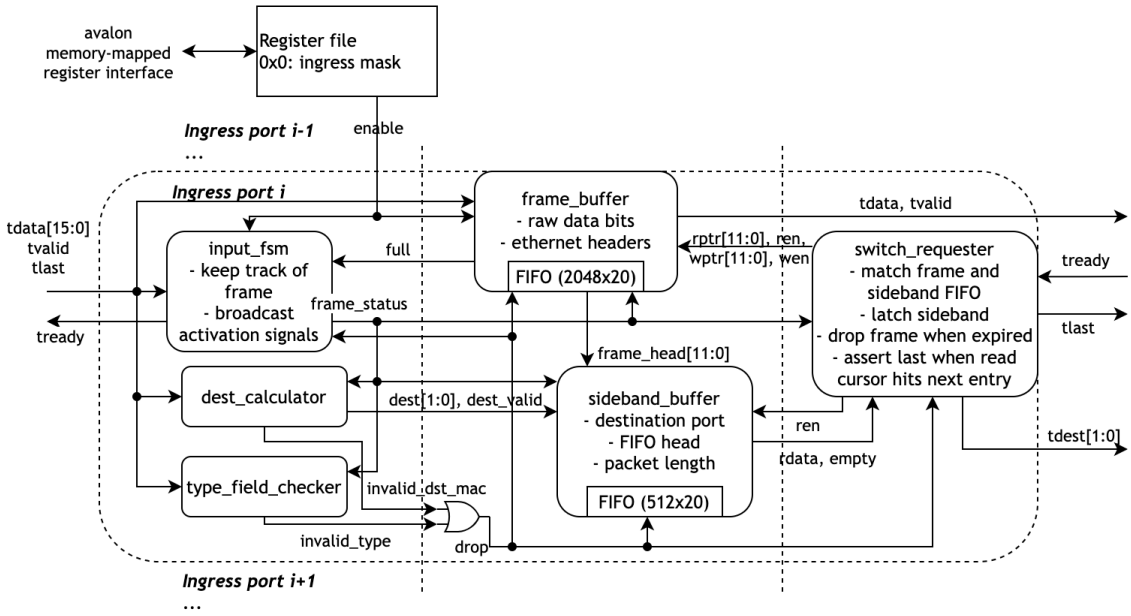

Figure 3: The connections within the input filter.

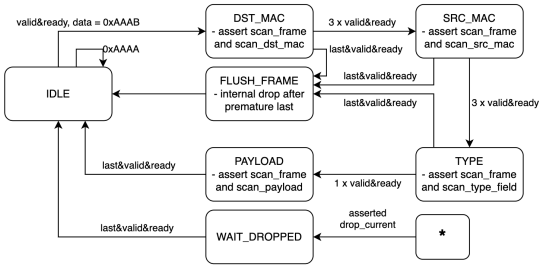

Figure 4: The input state machine diagram.

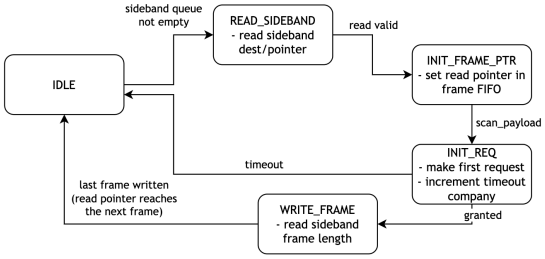

Figure 5: The request state machine diagram.

stream structure. Fig. 5 shows its state machine. It first waits for the sideband buffer to have an entry and reads the first one (which contains the destination port and queue head). With the queue head, it will set the read pointer in the frame FIFO to point to the head of the corresponding frame. It can then make a request to the switch with a corresponding destination; thus allowing the switch to match the stream request to an egress port.

After making the request and receiving a grant, the requestor must find the length to be able to accurately recreate the AXIS last signal. Whether or not it has this value, it counts the number of packets it has issued so that when the length is available, it can compare the issued count. After latching the length, it continues to send packets. Once the issued count is one less than the frame length, it asserts the last signal with the next packet.

One feature we will integrate to improve global throughput and prevent starvation is a frame request timeout. In the initial request state, the requestor increments a count while waiting. If it reaches a threshold, it drops the frame request returns to the request state to start a request with the next frame. This timeout mechanism is specifically helpful to avoid deadlock. For example, if all ingress ports make a request to an egress port exerting back-pressure, they will all stall with that frame in the head of the queue (thus stalling subsequent frames). One solution for this would be to implement separate queues for each egress port. However, this has several disadvantages. The separate queues would double the memory footprint of the filter from 4 blocks per ingress to 8 to be able to store a full-sized Ethernet frame for each egress port (2 blocks can store between 1 and 2 frames). Additionally, this would enforce that we buffer enough packets before the egress port is actually calculated. After calculation, the

centralized buffer would have to flush to the selected egress port. This would create the same issue of having a bottleneck on an ingress port that is susceptible to back-pressure. As a result, the input filter drops frames that have been at the head of the queue for a number of cycles equivalent to issuing a maximum length ethernet frame.

3.2 Frame switch

The Frame Switch is the nexus that joins the four filtered ingress streams to the four egress ports. As the destination, last and valid signals are side banded, that makes it easier for the switching fabric to handle crossing logic. It has the following organization:

3.2.1 Per-egress Round-Robin Scheduler and Destination select. a finite-state machine that stores the pointer next_rr. While in SEND it keeps the pointer frozen so the entire frame is drained from a single ingress port; when it observes tlast = 1 it cyclically increments the pointer to the next ingress port and returns to IDLE. This implements fair, frame-granular arbitration with no head-of-line blocking. This FSM is also responsible for matching the Ingress destination to the Egress Index. It matches the Destination of each ingress first, then sorts the Valid ingresses only to perform arbitration afterwards. The output from these Schedulers (per egress) is sent to the frame receptor in form of egress valid and egress last, and one output is sent to the per-egress data MUX as Select line which contains the Ingress id to route the correct ingress data to egress port for which scheduler is assigned. The Egress Port id is fed as input through a register to the egresses Schedulers in the switch fabric wrapper.

3.2.2 Cross-bar Data Path. four independent 4×16-bit multiplexers controlled by the four scheduler outputs (select[1:0]). Because all ports share the same word width, the cross-bar is purely combinational and adds only one 4-LUT level of latency per data bit. The data is routed from each Ingress data incoming from the filter to the egress and selected based on the select line from the scheduler.

3.2.3 Back-pressure Propagation. each egress port's tready is fanned backwards through its scheduler to the selected ingress port's tready after getting AND with the grant for that port if enabled. Thus, incase the Ingress has grant but egress exerts backpressure, the ingress is asserted with low Ingress tready. Consequently, when a receiver stalls in the middle of a frame, only the corresponding ingress port is paused, preventing FIFO overflow elsewhere in the switch.

3.3 Testbench

The testbench is implemented in software running on the Hard Processor System (HPS) and is responsible for verifying the functionality and performance of the Ethernet frame filtering switch. It communicates with the hardware system using Avalon memory-mapped (AMM) interfaces to configure modules, generate test cases, and validate output data.

The testbench software performs several key roles. It first configures the hardware-based frame generators by writing to their control registers through the AMM interface, setting parameters such as payload size, destination MAC address, and EtherType. Once the configuration is complete, the software initiates frame transmission by signaling the frame generators. As frames propagate through the system, the testbench monitors the output by reading from the frame receptors.

To validate correctness, the software compares the expected and actual outputs. This includes checking that the payload checksums computed by the frame receptors match those generated by the frame generators. It also verifies that each frame is routed to the correct destination port based on its MAC address, and confirms proper drop behavior for frames with invalid Ethertype values or those that exceed the configured timeout threshold. This software-driven testbench enables flexible and comprehensive testing of both the functional behavior and performance characteristics of the system. Because the frame generation is done in hardware, we can eventually evaluate the speed of our modules without having to interact with DDR4 or SDRAM blocks which would add latency and complexity to our testing.

4 ALGORITHM

Our packet switch implements a dedicated round-robin scheduler for each egress port implemented in the pointer and index based manner inside a simple finite state machine (FSM) with two states: IDLE and SEND. The purpose of this scheduler is to ensure fairness and bounded latency by allowing each ingress port to transmit data to a given egress port in turn. The FSM is also responsible for performing the Ingress_destination check to make sure that arbitration is happening amongst Ingress Ports that have valid destination id for the Egress FSM

4.0.1 FSM Implementation. Each egress-side FSM maintains a 2-bit pointer, next_rr, which tracks the starting ingress port for arbitration. This pointer rotates cyclically through all the ingress ports. Arbitration and selection are based on a combination of Index based round robin and ingress port metadata, such as tvalid, tdest, and tlast.

Should more differentiation be required in the future the pointer update can be swapped for a priority or weighted-RR table without touching the data path.

4.0.2 IDLE State. In the IDLE state, the FSM performs round-robin arbitration. It iterates from the current next_rr position and scans each ingress port (using an offset index =

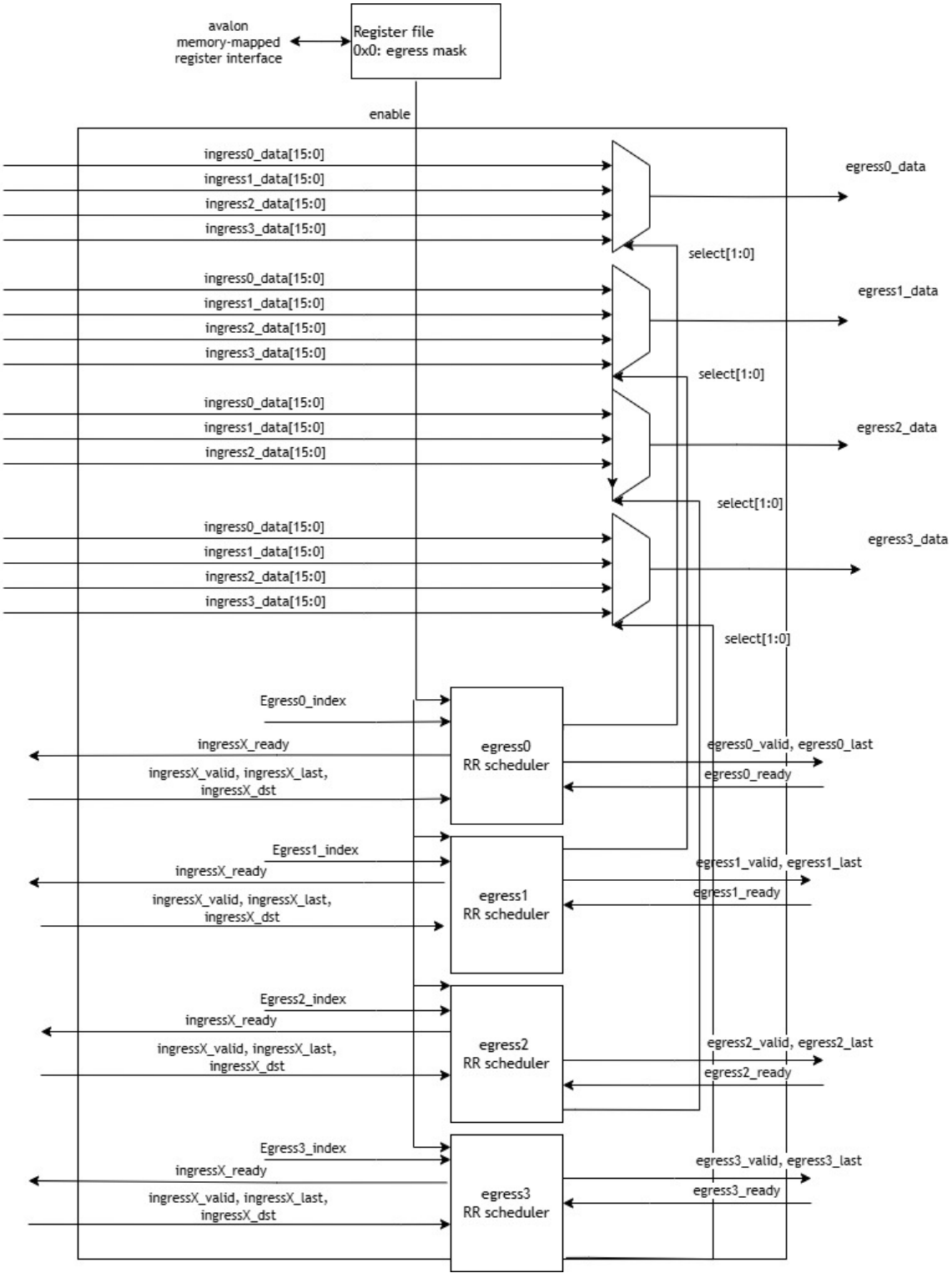

Figure 6: The Switch Fabric.

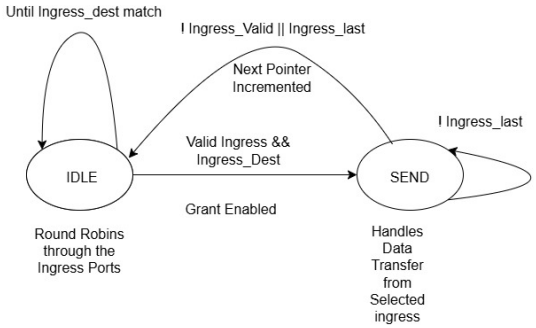

Figure 7: The Select FSM with Arbitration

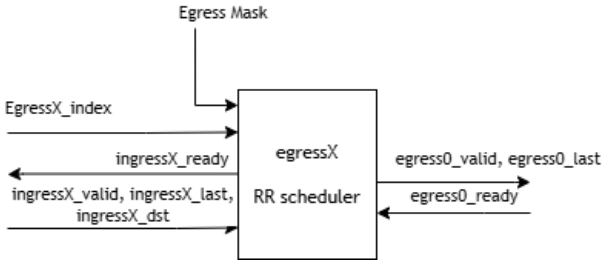

Figure 8: The Scheduler Interface

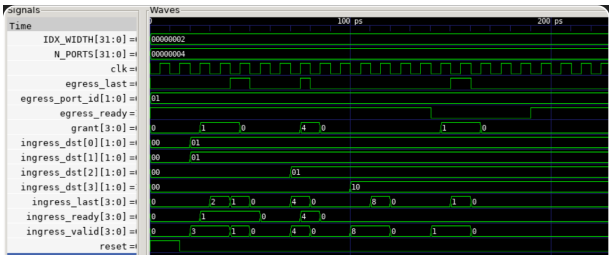

Figure 9: Grant Arbitration in Single Egress Scheduler

next_rr + i) to find: A valid ingress port (ingress_valid[index] == 1) and A matching destination (ingress_dst[index] == egress_port_id). Upon finding a match: The FSM selects that ingress index (select_next = index). It transitions to the SEND state. The grant signal is asserted for that ingress index.

4.0.3 SEND State. In the SEND state, the FSM manages data transfer from the selected ingress port to the corresponding egress. egress_valid and egress_last are directly driven by the selected ingress port. Backpressure is handled using egress_ready, and the ingress_ready signal is derived as the bitwise AND of grant and egress_ready.

The FSM remains in the SEND state as long as: The ingress stream is still valid or The frame has not ended

4.0.4 Back to IDLE. The FSM returns to IDLE in two cases 1)When the current word is the last in the frame and is successfully transferred. In this case, next_rr is incremented, rotating the pointer to the next ingress for future arbitration. 2) If the ingress source becomes invalid, indicating that the frame has ended unexpectedly or the sender has paused.

This logic ensures that no single ingress can monopolize the scheduler, and all ingress ports get a fair chance to transmit in round-robin order. Additionally, the grant signal acts as an internal token that is cleared automatically after a complete frame is transferred, or if the ingress stalls.

4.1 Arbitration Simulation

To validate our Round-Robin scheduling logic in isolation, we implemented a software simulation that mirrors the arbitration behavior of our hardware switch. This model acts as a proof-of-concept to ensure that fairness, frame-level

granularity, and ingress contention are handled as expected before RTL implementation.

We wanted to :

1) Emulate the behavior of each egress FSM.

2) Verify that each egress port grants access to at most one ingress at a time.

3) Confirm that arbitration strictly respects frame boundaries (tlast) before switching ingress sources.

We defined following C structs to model the ingress state and arbitration output

```
typedef struct {
  char valid;   // Whether ingress has a valid
    word
  char last;    // Whether this word is the end of
     the frame
  char dest;    // Which egress port the word is
    targeting
} axis_input;

typedef struct {
  axis_input ingress[4];
} switch_input;

typedef struct {
  char grant[4]; // Which ingress was granted
  char select;   // Which ingress was connected to
     the egress port
} switch_output;
```

Each egress port keeps track of:

1) A round-robin pointer (next_rr) indicating the next ingress to try.

2) A grant lock (cur_grant) which holds an ingress index until tlast is observed.

```
function arbitrate(input, egress_port):
    if cur_grant[egress] is locked:
        if ingress[cur_grant].valid and .dest ==
    egress:
            grant it
            if .last == 1: release lock, advance
    RR pointer
    else:
        for i in RR order:
            if ingress[i].valid and .dest ==
    egress:
                grant i, lock it
                if .last == 1: unlock immediately
```

Above is the Psuedo-Code for the Arbitration logic implementedfor simulation.

We simulated 5 clock cycles with the following test scenario:

Ingress 0 and 1 both send frames to Egress 0 (creating contention).

Ingress 3 sends a short 1-word frame to Egress 2.

Ingress 2 sends a multi-word frame to Egress 1 starting in Cycle 1.

Cycle	Ing0(v,l,d)	Ing1(v,l,d)	Ing2(v,l,d)	Ing3(v,l,d)
0	100	100	000	112
1	110	110	101	000
2	000	000	101	000
3	000	000	101	000
4	000	000	111	000

Table 1: Test_vector Table

After running the Simulation

The results show:

Ingress 0 was selected first for Egress 0 due to pointer ordering.

Ingress 1 waited 2 cycles before being able to transmit (shown in waiting stats).

Egress 1 FSM locks onto Ingress 2 and completes its 4-word frame across multiple cycles, as in Figure 4.

5 HARDWARE-SOFTWARE INTERFACES

Each of the modules host the agent side of the Avalon memory-mapped interface. They each have read-write registers which are addressable by an eight-bit address. Table 2 shows the base addresses for each of the modules. Otherwise, all communication logic is directly via AXIS interfaces between the modules as Section 3 discusses.

5.0.1 Frame generator. The Frame Generator module is responsible for constructing Ethernet frames based on control data received from the software testbench via the Avalon memory-mapped interface. This control data includes parameters such as payload size, destination Mac address, and packet type. Once the configuration is received, the module assembles an Ethernet frame following the format outlined in Figure 1.

Each frame includes standard Ethernet fields such as the preamble, destination and source Mac addresses, EtherType, payload, and frame check sequence. The source MAC address is selected dynamically based on which of the four Frame Generators instances is transmitting the packet, allowing for the simulation of traffic from multiple sources. After construction, the frame is transmitted to Frame filter module through the AXI-Stream interface.

In addition to frame generation, the module also performs a basic checksum calculation over the payload. This checksum is stored in a readable register, which can later be accessed and compared by software to verify data integrity. This helps validate correct transmission and detect potential errors during testing.

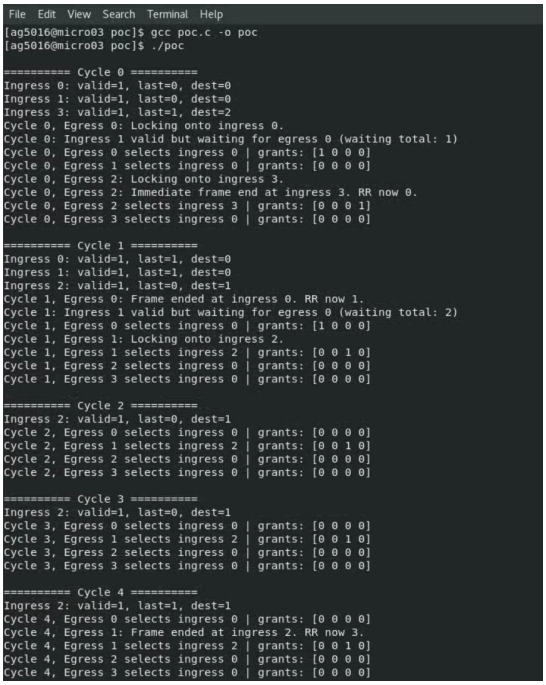

Figure 10: Arbitration Simulation Result

Address offset	Module type	Instance name
0x0000	frame_filter	Ethernet frame filter
0x1000	frame_switch	Ethernet frame switch
0x2000	frame_generator	Ethernet frame generator 0
0x2400	frame_generator	Ethernet frame generator 1
0x2800	frame_generator	Ethernet frame generator 2
0x2c00	frame_generator	Ethernet frame generator 3
0x3000	frame_receptor	Ethernet frame receptor 0
0x3400	frame_receptor	Ethernet frame receptor 1
0x3800	frame_receptor	Ethernet frame receptor 2
0x3c00	frame_receptor	Ethernet frame receptor 3

Table 2: Base addresses in the system.

5.0.2 Frame filter. The frame filter has one control register and many readable counters as indicated in Table 4.

5.0.3 Frame switch. The Frame Switch exposes just one read-/write register: egress_mask as per Table 6.

5.0.4 Frame receptor. For the Frame receptor module, as we described in the Frame Generator module that it will perform a basic checksum of the payload after the Ethernet frame passed from the network switch module, it will also store the information in a software readable register that we will be using during testbench to validate the data integrity and accuracy.

Furthermore, the frame receptor is the final step before checking the data integrity (via the checksum), it also verifies the correctness of the routing by checking the destination mac address matches a pattern that it stores in a software-writable register.

6 EVALUATION

6.1 Digital verification

We followed a bottom-up integration approach for verifying our functionality. This section details how we achieved verifying the packet_filter module. Each of the units have assertions that enforce interface constraints. For example, the input FSM and request buffers have assertions to check that ready grants are not provided in the same cycle, so our system complies with a valid-before-ready AXI-Stream style. Another unit on which we performed extensive verification

Address offset	Read/ Write	Name	Description
0x00	RW	dst_0	Dest Address Byte 0
0x01	RW	dst_1	Dest Address Byte 1
0x02	RW	dst_2	Dest Address Byte 2
0x03	RW	dst_3	Dest Address Byte 3
0x04	RW	dst_4	Dest Address Byte 4
0x05	RW	dst_5	Dest Address Byte 5
0x06	RW	src_0	Source Address Byte 0
0x07	RW	src_1	Source Address Byte 1
0x08	RW	src_2	Source Address Byte 2
0x09	RW	src_3	Source Address Byte 3
0x0A	RW	src_4	Source Address Byte 4
0x0B	RW	src_5	Source Address Byte 5
0x0C	RW	length_0	Payload length Byte 0
0x0D	RW	length_1	Payload length Byte 1
0x0E	RW	type_0	Frame Type Byte 0
0x0F	RW	type_1	Frame Type Byte 1
0x10	RW	inter_ frame_ wait	Cycles to wait between frames
0x11	R	checksum _0	Payload checksum byte 0
0x12	R	checksum _1	Payload checksum byte 1
0x13	R	checksum _2	Payload checksum byte 2
0x14	R	checksum _3	Payload checksum byte 3

Table 3: Register map for frame generator

Address offset	Read/ Write	Name	Description
0x00	RW	ingress_ port_ mask	Enable signal for ingress ports (active-high)
0x04-0x07	R	in_pkts_<i>	Number of accepted 16-bit ingress packets on ingress port i
0x08-0x0B	R	transf_pkts_<i>	Number of 16-bit packets transferred on egress port i
0x0C-0x0F	R	in_frames_<i>	Number of accepted ethernet frames on ingress port i
0x10-0x13	R	transf_frames_<i>	Number of ethernet frames transferred on egress port i
0x14-0x17	R	inv_frames_<i>	Number of frames dropped as invalid from ingress port i
0x18-0x1B	R	drop_frames_<i>	Number of frames dropped as timed out from egress port i

Table 4: Registers for the frame filter module.

Address offset	Read/ Write	Name	Description
0x00	RW	egress_ port_ mask	Enable signal for egress ports (active-high)

Table 5: Registers for the frame switch module.

was the FIFO with controllable cursors. This has many assertions to verify behavior integrity; one example is that the reader/writer interface does not allow that interface to enqueue or dequeue while it is resetting its cursor.

6.2 Resource consumption

The Cyclone 5CSEMA5 has 397 ten-Kb blocks of BRAM. Each block has 512 twenty-bit words. As mentioned in Section 3.1.2, we extend our data words to twenty bits to fit nicely into memory. Each frame has a maximum size of 759 16-bit words. We then decided to use four blocks in each ingress port which allows the system to buffer at least two full-sized frames. Specifically, the four blocks can store 2.7 full-sized frames, or many smaller frames. The four blocks create an addressable space of 2048 words, which requires 11 bits of address. For the sideband FIFO, we use a single BRAM module.

The chip has a total of 397 blocks. Using five total blocks for each of the four ingress ports results in a total usage of twenty blocks. This is just over five percent usage for BRAM. We will be able to present more specific statistics for register and DSP usage when we implement the design.

After synthesizing our design in Quartus, we found that our design used the exact number of blocks as we expected. Table 7 lists the key statistics.

6.3 Performance

We also obtained timing results from Quartus. The 50MHz clock signal can go to a maximum of 76.62MHz. However, our design blew past that mark and was able to synthesize at a restricted maximum frequency of 717.36MHz. These successes are in large part due to the pipelining of the filter's internal microarchitecture and the switch's low combinational complexity. Specifically, because we enforce the

Address offset	Read/ Write	Name	Description
0x00	RW	dst_0	Dest Address Byte 0
0x01	RW	dst_1	Dest Address Byte 1
0x02	RW	dst_2	Dest Address Byte 2
0x03	RW	dst_3	Dest Address Byte 3
0x04	RW	dst_4	Dest Address Byte 4
0x05	RW	dst_5	Dest Address Byte 5
0x06	RW	inter_ frame_ wait	Cycles to wait between frames
0x07	R	dstCheck	Destination MAC checking
0x08	R	checksum _0	Payload checksum byte 0
0x09	R	checksum _1	Payload checksum byte 1
0x0A	R	checksum _2	Payload checksum byte 2
0x0B	R	checksum _3	Payload checksum byte 3

Table 6: Register map for frame receptors

Metric	Value	Utilization
BRAM Bits	159744	4%
Pins	362	79%
ALMs	4118	13%
Registers	5556	
RAM Blocks	20	5%

Table 7: Quartus resource usage.

valid-before-ready interface, the switch cannot accept a new grant in the same cycle as a request; that would create a long combinational path between the switch and filter. Ideally, with no back-pressure, our switch can output 16 bits per cycle, allowing for a maximum bandwidth of 11.5Gbps or 1.43GBps.

However, we were unable to verify this with performance counters in deployment. The succeeding section discusses these shortcomings. We did verify the per-cycle throughput in simulation by reading the performance counters available in the packet filter. Table 8 shows the performance counters after issuing 100 frames with no back-pressure and one frame with back-pressure which times out.

We also measured the response time of the filter to be 17 cycles. Specifically, this means that without back-pressure, the input filter will make a request 17 cycles after receiving the start frame delimiter (SFD) of a valid frame. These 17 cycles allow the filter to process the eight packets for the ethernet headers (including the SFD) in its pipeline, write to the frame buffer, and then determine whether it can request to the egress.

6.4 FPGA deployment

Our entire system did not successfully deploy. Even though we had a bitstream that properly synthesized, we ran into issues with implementing the device drivers in Linux.

6.4.1 Frame Generator. Frame generator was successfully tested in hardware and can be seen in the below figure 8. While deployed onto the fpga, we are running into problems with multiple same drivers for frame generators as we couldn't create the all four frame generator module for software to run.

6.4.2 Packet filter. We implemented a simple device driver for the packet filter that is able to access registers. Since the generator and receptor did not have proper setups, we were unable to see the performance counters of the system changing as frames flowed.

6.4.3 Frame Receptor. The frame receptor is also tested in hardware and the behavior can be seen in the the figure 9. While deployed onto the fpga, we are running into a different problem as it was causing the software to stale during opening module drivers.

7 CONCLUSION

This project obviously had shortcomings that prevented us from demonstrating our system's potential efficiency in a deployed system. However, we had successes in the design of the filter portion. The robust verification we performed demonstrated that with extra time fighting through device drivers and corner cases in the switch fabric, we would be able to deploy our system.

REFERENCES

[1] Wikipedia contributors. Ethernet frame — wikipedia, the free encyclopedia, 2024. Accessed: 2025-04-18.

Ingress port	Ingress packets	Transferred packets	Ingress frames	Transferred frames	Dropped frames
0	12450	11550	101	100	1
0	9050	7350	100	100	0
0	17150	13550	100	100	0
0	14050	12750	100	100	0

Table 8: Performance counters as read in simulation. Note that the packets transferred from the filter is less than the ingress packets because the filter strips the preamble bytes.

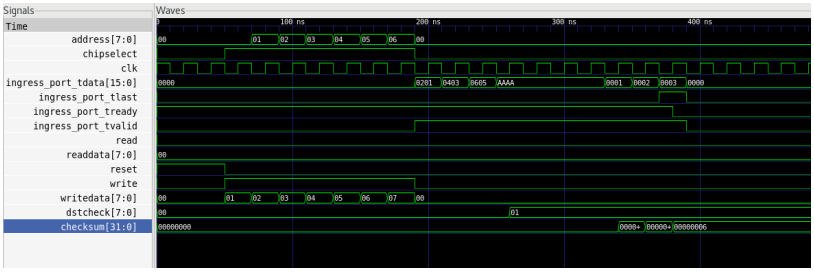

Figure 11: Frame Generator Simulation Result

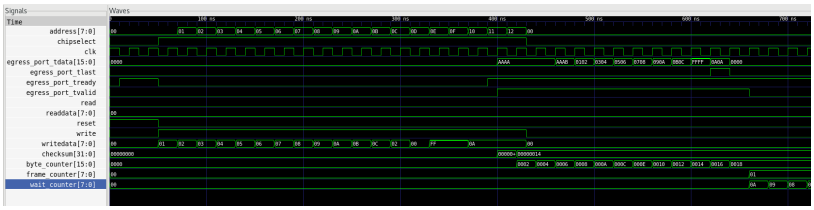

Figure 12: Frame Receptor Simulation Result

8 HARDWARE RTL

8.1 Common files

Listing 1: include/fifo_sync.sv

```
/**
 * Synchronous FIFO buffer.
 *
 * Main parameters:
 *   - ADDR_WIDTH: controls depth of the buffer
 *   - W_EL: data width
 *   - CAN_RESET_POINTERS: whether external control can reset the read and write pointers
 *
 * Configurable parameters for the Cyclone 5CSEMA5:
 *   - ADDR_WIDTH: address width = clog2(NUM_CYCLONE_5CSEMA5_BLOCKS) + 9
 *   - W_EL: data width (1 - 20)
 *   - NUM_CYCLONE_5CSEMA5_BLOCKS: number of 1280-Byte-blocks (512x20-word-blocks)
 */
`timescale 1 ps / 1 ps
```

```
module fifo_sync #(
    parameter ADDR_WIDTH = 11,
    parameter W_EL = 20,
    parameter NUM_CYCLONE_5CSEMA5_BLOCKS = 4,
    parameter CAN_RESET_POINTERS = 0,
    parameter RDATA_PIPELINE = 1
) (
    // clock and reset
    input  logic clk,
    input  logic reset,

    // read interface
    input  logic             ren,
    output logic [W_EL-1:0] rdata,
    output logic             empty,

    // write interface
    input  logic [W_EL-1:0] wdata,
    input  logic             wen,
    output logic             full,

    // cursor control (don't care if CAN_RESET_POINTERS == 0)
    input  logic                  rrst,
    input  logic                  wrst,
    input  logic [ADDR_WIDTH:0] rst_rptr,
    input  logic [ADDR_WIDTH:0] rst_wptr,
    output logic [ADDR_WIDTH:0] rptr,
    output logic [ADDR_WIDTH:0] wptr
);

    /* Assertions related to parameters. */
    initial begin
        if (
            NUM_CYCLONE_5CSEMA5_BLOCKS != 0 && (
                (ADDR_WIDTH != $clog2(NUM_CYCLONE_5CSEMA5_BLOCKS) + 9) ||
                (W_EL > 20)
            )
        )begin
            $error("Invalid generics in fifo_sync.");
            $finish;
        end
    end

    // memory valid signals
    logic mem_wvalid, mem_rvalid;

    // cursors
    logic [ADDR_WIDTH:0] next_rptr;
```

```
    logic [ADDR_WIDTH:0] rptr_access;
    logic [ADDR_WIDTH:0] next_wptr;
    logic ptr_overlap;
    logic next_full, next_empty;

    /* Write logic. */
    assign mem_wvalid = wen && !full;
    generate
        if (CAN_RESET_POINTERS == 1) begin: g_resettable_wptr
            always @(posedge clk) begin
                if (reset) begin
                    wptr <= '0;
                    full <= 1'b0;
                end else if (wrst) begin
                    wptr <= rst_wptr;
                    full <= 1'b0;
                end else begin
                    wptr <= next_wptr;
                    full <= next_full;
                end
            end
        end else begin: g_wptr
            always @(posedge clk) begin
                if (reset) begin
                    wptr <= '0;
                    full <= 1'b0;
                end else begin
                    wptr <= next_wptr;
                    full <= next_full;
                end
            end
        end
    endgenerate
    always @(*) begin
        if (mem_wvalid) begin
            next_wptr = wptr + 1;
        end else begin
            next_wptr = wptr;
        end
    end

    /* Read logic. */
    assign mem_rvalid = ren && !empty;
    generate
        if (CAN_RESET_POINTERS == 1) begin: g_resettable_rptr
            always @(posedge clk) begin
                if (reset) begin
                    rptr <= '0;
```

```
                empty <= 1'b0;
            end else if (rrst) begin
                rptr <= rst_rptr;
                empty <= 1'b0;
            end else begin
                rptr <= next_rptr;
                empty <= next_empty;
            end
        end
    end else begin: g_rptr
        always @(posedge clk) begin
            if (reset) begin
                rptr <= '0;
                empty <= 1'b0;
            end else begin
                rptr <= next_rptr;
                empty <= next_empty;
            end
        end
    end
endgenerate
always @(*) begin
    if (mem_rvalid) begin
        next_rptr = rptr + 1;
    end else begin
        next_rptr = rptr;
    end
end

generate
    if (RDATA_PIPELINE) begin: g_rptr_pipeline
        assign rptr_access = rptr;
    end else begin: g_rptr_no_pipeline
        assign rptr_access = next_rptr;
    end
endgenerate

// generate each block of memory
genvar mem_block_i;
generate
    if (NUM_CYCLONE_5CSEMA5_BLOCKS == 0) begin: g_mem
        logic [W_EL-1:0] mem [2**ADDR_WIDTH-1:0];

        // memory logic
        always_ff @(posedge clk) begin
            if (mem_wvalid) begin
                mem[wptr[ADDR_WIDTH-1:0]] <= wdata;
            end
```

```
                rdata <= mem[rptr_access[ADDR_WIDTH-1:0]];
            end
        end else if (NUM_CYCLONE_5CSEMA5_BLOCKS == 1) begin: g_single_block
            logic [W_EL-1:0] mem [511:0];

            // memory logic
            always_ff @(posedge clk) begin
                if (mem_wvalid) begin
                    mem[wptr[8:0]] <= wdata;
                end
                rdata <= mem[rptr_access[8:0]];
            end
        end else begin: g_mult_blocks
            logic [W_EL-1:0] mem_rdata [NUM_CYCLONE_5CSEMA5_BLOCKS-1:0];
            for (mem_block_i = 0; mem_block_i < NUM_CYCLONE_5CSEMA5_BLOCKS; ++mem_block_i) beg
                // current memory block
                logic [W_EL-1:0] mem [511:0];

                // masked enable signal
                logic mem_block_wvalid;
                assign mem_block_wvalid = (wptr[ADDR_WIDTH-1:9] == mem_block_i) ? mem_wvalid

                // memory logic
                always_ff @(posedge clk) begin
                    if (mem_block_wvalid) begin
                        mem[wptr[8:0]] <= wdata;
                    end
                    mem_rdata[mem_block_i] <= mem[rptr_access[8:0]];
                end
            end

            // select read output
            assign rdata = mem_rdata[rptr_access[ADDR_WIDTH-1:9]];
        end
    endgenerate

    // write intermediate signals
    assign ptr_overlap = (next_rptr[ADDR_WIDTH-1:0] === next_wptr[ADDR_WIDTH-1:0]) ? 1'b1 : 1
    assign next_full = (ptr_overlap && next_rptr[ADDR_WIDTH] !== next_wptr[ADDR_WIDTH]) ? 1'b
    assign next_empty =  (ptr_overlap && next_rptr[ADDR_WIDTH] === next_wptr[ADDR_WIDTH]) ? 1

`ifdef ASSERT
    // assert FIFO displays full until a read completes
    assertion_fifo_sync_full_until_read : assert property(
        @(posedge clk) disable iff (rrst || wrst)
        $rose(full) |=> full || $past(ren, 1)
    ) else $error($sformatf("assertion_fifo_sync_full_until_read failed at %0t", $realtime));
```

```
    // assert FIFO displays empty until a write completes
    assertion_fifo_sync_empty_until_write : assert property(
        @(posedge clk) disable iff (rrst || wrst)
        $rose(empty) |=> empty || $past(wen, 1)
    ) else $error($sformatf("assertion_fifo_sync_empty_until_write failed at %0t", $realtime))

    // assert read reset and read enable not asserted at the same time
    assertion_fifo_sync_read_reset_enable : assert property(
        @(posedge clk) disable iff (reset)
        ~(rrst & ren)
    ) else $error("Read enable and read reset asserted in the same cycle");

    // assert write reset and write enable not asserted at the same time
    assertion_fifo_sync_write_reset_enable : assert property(
        @(posedge clk) disable iff (reset)
        ~(wrst & wen)
    ) else $error("Write enable and write reset asserted in the same cycle");
`endif

endmodule
```

Listing 2: include/packet_filter.svh

```
`ifndef _PACKET_FILTER_SVH_
`define _PACKET_FILTER_SVH_

/**
 * Stubbing
 */

`define STUBBING_PASSTHROUGH 0 // only register interface
`define STUBBING_FUNCTIONAL  1 // full functionality

/**
 * AXIS interfaces
 */

`define AXIS_DWIDTH 16
`define AXIS_DEST_WIDTH 2

`define NUM_INGRESS_PORTS 4
`define NUM_EGRESS_PORTS 4

`define ETH_SFD 16'hAAAB

/* verilator lint_off UNPACKED */

// AXIS data source
```

```
typedef struct {
    logic [`AXIS_DWIDTH-1:0] tdata;
    logic                    tvalid;
    logic                    tlast;
} axis_source_t;

// AXIS data source with destination field
typedef struct {
    logic [`AXIS_DWIDTH-1:0]      tdata;
    logic                         tvalid;
    logic                         tlast;
    logic [`AXIS_DEST_WIDTH-1:0] tdest;
} axis_d_source_t;

// AXIS data sink
typedef struct {
    logic tready;
} axis_sink_t;
typedef axis_sink_t axis_d_sink_t;

/* verilator lint_on UNPACKED */

`endif // _PACKET_FILTER_SVH_
```

Listing 3: include/reg_set_clear.sv

```
module reg_set_clear (
    input  logic clk,
    input  logic reset,
    input  logic set,
    input  logic clear,
    output logic data
);

    always_ff @(posedge clk) begin
        if (reset) begin
            data <= 1'b0;
        end else begin
            if (set) begin
                data <= 1'b1;
            end else if (clear) begin
                data <= 1'b0;
            end else begin
                data <= data;
            end
        end
    end
```

```
endmodule
```

Listing 4: include/synth_defs.svh

```
`ifndef _SYNTH_DEFS_SVH_
`define _SYNTH_DEFS_SVH_

`define TOP_TESTING
`define INTG_TESTING_1

`endif // _SYNTH_DEFS_SVH_
```

8.2 Frame Generator

Listing 5: Frame Generator

```
// frame_generator.sv

/**
 * Register mapping
 *
 * Byte / mode | Name              | Meaning
 *         0W  |  Destination MAC  |  Destination MAC byte 0.
 *         1W  |  Destination MAC  |  Destination MAC byte 1.
 *         2W  |  Destination MAC  |  Destination MAC byte 2.
 *         3W  |  Destination MAC  |  Destination MAC byte 3.
 *         4W  |  Destination MAC  |  Destination MAC byte 4.
 *         5W  |  Destination MAC  |  Destination MAC byte 5.
 *         6W  |       Source MAC  |       Source MAC byte 0.
 *         7W  |       Source MAC  |       Source MAC byte 1.
 *         8W  |       Source MAC  |       Source MAC byte 2.
 *         9W  |       Source MAC  |       Source MAC byte 3.
 *        10W  |       Source MAC  |       Source MAC byte 4.
 *        11W  |       Source MAC  |       Source MAC byte 5.
 *        12W  |   Payload length  |   Payload length byte 0.
 *        13W  |   Payload length  |   Payload length byte 1.
 *        14W  |       Type field  |       Type field byte 0.
 *        15W  |       Type field  |       Type field byte 1.
 *        16W  | Inter-frame wait  |  Cycles to wait between frames.
 *        17R  |         Checksum  | Payload checksum byte 0.
 *        18R  |         Checksum  | Payload checksum byte 1.
 *        19R  |         Checksum  | Payload checksum byte 2.
 *        20R  |         Checksum  | Payload checksum byte 3.
 */

`ifdef VERILATOR
`include "packet_filter.svh"
`else
`include "../include/packet_filter.svh"
```

```
`endif

`timescale 1 ps / 1 ps
module frame_generator #(
        parameter STUBBING = `STUBBING_PASSTHROUGH,
        parameter CAN_RESET_POINTERS = 0
    ) (
                input  wire          clk,                //           clock.clk
                input  wire          reset,              //           reset.reset
                input  wire  [31:0]  writedata,          // avalon_slave_0.writedata
                input  wire          write,              //                .write
                input  wire          chipselect,         //                .chipselect
                input  wire  [7:0]   address,            //                .address
                input  wire          read,               //                .read
                output logic [31:0]  readdata,           //                .readdata
                output logic [15:0]  egress_port_tdata,  //    egress_port.tdata
                output logic         egress_port_tlast,  //                .tlast
                input  wire          egress_port_tready, //                .tready
                output logic         egress_port_tvalid  //                .tvalid
        );

        /* Register file. */
    logic [7:0]  reg_file [0:16];
    logic [31:0] checksum;
    logic [15:0] payload_byte;
    logic [15:0]  byte_counter;
    logic        sending;
    logic [7:0]  wait_counter;
    //logic [7:0] input_counter;
    logic [7:0] frame_counter;
    //logic [15:0] type_temp;
    // register write interface
    always_ff @(posedge clk) begin
        if (reset) begin
            for (int i = 0; i <= 16; i++)
                reg_file[i] <= 8'h00;
            checksum <= 0;
            payload_byte <= 0;
                //input_counter <= 0;
        end else if (chipselect && write) begin
            if(address <= 16) begin
                reg_file[address[4:0]] <= writedata[7:0];
            end else if(address > 16) begin
                        if({reg_file[13], reg_file[12]} != 0) begin
                            if({8'h00,address} <= (17 + {reg_file[13], reg_file[12]})) begin
                                if(!address[0])
                            payload_byte[7:0] <= writedata[7:0];
                        else
```

```
                    payload_byte[15:8] <= writedata[7:0];
                checksum <= checksum + {24'b0, writedata[7:0]};
                    end
            end
        end
            end
end


// register read interface
always_ff @(posedge clk) begin
    if (reset) begin
        readdata <= 32'h0;
    end else if (chipselect && read) begin
        if(address <= 16)
            readdata[7:0] <= reg_file[address[4:0]];
        else if (address >= 17 && address <= 20)
            case (address)
                17 : readdata[7:0] <= checksum[7:0];
                18 : readdata[7:0] <= checksum[15:8];
                19 : readdata[7:0] <= checksum[23:16];
                20 : readdata[7:0] <= checksum[31:24];
            endcase
    end
    else begin
        readdata <= 32'h00;
    end
end
//Frame State Machine
always_ff @(posedge clk) begin
    if(reset) begin
        sending <= 0;
        byte_counter <= 0;
        wait_counter <= 0;
        frame_counter <= 0;
    end
    else begin
        if(!sending && wait_counter == 0 && egress_port_tready) begin
            sending <= 1;
            byte_counter <= 0;
        end
        else if (sending && egress_port_tready) begin
            if(byte_counter >= (24 + {reg_file[13], reg_file[12]})) begin
                frame_counter <= frame_counter + 8'h01;
                sending <= 0;
                wait_counter <= reg_file[16];
            end else
                byte_counter <= byte_counter + 16'h02;
```

```
            end
            else if(!sending && wait_counter > 0) begin
                wait_counter <= wait_counter - 8'h01;
            end
        end
    end

    //Frame data
    always_comb begin
        egress_port_tvalid = sending;
        egress_port_tlast = (byte_counter == (24 + {reg_file[13], reg_file[12]} -2));
        egress_port_tdata = 16'h0000;
        if(sending) begin
            unique case (byte_counter)
                        //preamble
                        0  : egress_port_tdata = 16'hAAAA;
                        2  : egress_port_tdata = 16'hAAAA;
                        4  : egress_port_tdata = 16'hAAAA;
                        //preamble & sfd
                        6  : egress_port_tdata = 16'hAAAB;
                //dst
                8  : egress_port_tdata = {reg_file[0], reg_file[1]};
                10  : egress_port_tdata = {reg_file[2], reg_file[3]};
                12  : egress_port_tdata = {reg_file[4], reg_file[5]};
                //source
                14  : egress_port_tdata = {reg_file[6], reg_file[7]};
                16  : egress_port_tdata = {reg_file[8], reg_file[9]};
                18 : egress_port_tdata = {reg_file[10], reg_file[11]};
                //length
                20 : egress_port_tdata = {reg_file[12], reg_file[13]};
                //type
                22 : egress_port_tdata = {reg_file[14], reg_file[15]};
                default: begin
                    if(byte_counter >= 24) begin
                                if({reg_file[13], reg_file[12]} != 0) begin
                                        if(byte_counter < (24 + {reg_file[13], reg_file[12]}))
                                egress_port_tdata = payload_byte;
                                    end
                            end
                end
            endcase
            end
    end
endmodule
```

8.3 Frame Receptor

Listing 6: Frame Receptor

```
// frame_receptor.sv


/**
 * Register mapping
 *
 * Byte / mode | Name              | Meaning
 *         0W  |  Destination MAC  |  Destination MAC byte 0.
 *         1W  |  Destination MAC  |  Destination MAC byte 1.
 *         2W  |  Destination MAC  |  Destination MAC byte 2.
 *         3W  |  Destination MAC  |  Destination MAC byte 3.
 *         4W  |  Destination MAC  |  Destination MAC byte 4.
 *         5W  |  Destination MAC  |  Destination MAC byte 5.
 *         6W  | Inter-frame wait  |  Cycles to wait between frames.
  *        7R  |          dstcheck |  Destination check.
 *         8R  |          Checksum |  Payload checksum byte 0.
 *         9R  |          Checksum |  Payload checksum byte 1.
 *         10R |          Checksum |  Payload checksum byte 2.
 *         11R |          Checksum |  Payload checksum byte 3.
 */

`ifdef VERILATOR
`include "packet_filter.svh"
`else
`include "../include/packet_filter.svh"
`endif

`timescale 1 ps / 1 ps
module frame_receptor #(
        parameter STUBBING = `STUBBING_PASSTHROUGH,
        parameter CAN_RESET_POINTERS = 0
    ) (
                input  wire        clk,                 //          clock.clk
                input  wire        reset,               //          reset.reset
                input  wire [31:0] writedata,           // avalon_slave_0.writedata
                input  wire        write,               //                .write
                input  wire        chipselect,          //                .chipselect
                input  wire [7:0]  address,             //                .address
                input  wire        read,                //                .read
                output logic [31:0] readdata,           //                .readdata
                input  wire [15:0] ingress_port_tdata,  //   ingress_port.tdata
                input  wire        ingress_port_tvalid, //                .tvalid
                output logic        ingress_port_tready, //                .tready
                input  wire        ingress_port_tlast   //                .tlast
        );

        /* Register file. */
    logic [7:0] inter_frame_wait;
```

```
    logic [7:0]  reg_file [0:6];
    logic [7:0] dstcheck;
    logic [31:0] checksum;
    logic [7:0] input_counter;
    logic [7:0] frame_counter;
// generate
// if (STUBBING == `STUBBING_PASSTHROUGH) begin: g_passthrough

//      assign ingress_port_tready = 1'b0;

// end else begin: g_functional

// end
// endgenerate
    assign ingress_port_tready = (inter_frame_wait == 0) ? 1'b1 : 1'b0;
    // register write interface
    always_ff @(posedge clk) begin
        if(reset) begin
            for (int i = 0; i <= 6; i++)
                reg_file[i] <= 8'h00;
        end
        else if (chipselect && write)  begin
            if(address <= 6)
                reg_file[address[2:0]] <= writedata[7:0];
        end
    end
    //inter_frame_wait signal
    always_ff @(posedge clk) begin
        if(reset) begin
            inter_frame_wait <= 0;
        end
        if (ingress_port_tlast) begin
            inter_frame_wait <= reg_file[6];
        end else if (inter_frame_wait > 0 && (input_counter == 0)) begin
            inter_frame_wait <= inter_frame_wait - 8'h01;
        end
    end
    // register read interface
    always_ff @(posedge clk) begin
        if (reset) begin
            readdata <= 32'h0;
        end else if (chipselect && read) begin
            if(address <= 6)
                readdata <= reg_file[address[2:0]];
            else if (address >= 7 && address <= 11)
                case (address)
                    7  : readdata[7:0] <= dstcheck;
                    8  : readdata[7:0] <= checksum[7:0];
```

```
                9  : readdata[7:0] <= checksum[15:8];
                10 : readdata[7:0] <= checksum[23:16];
                11 : readdata[7:0] <= checksum[31:24];
            endcase
        end else begin
            readdata <= 32'h00;
        end
    end

    always_ff @(posedge clk) begin
        if (reset) begin
            checksum <= 0;
            input_counter <= 0;
            frame_counter <= 0;
            dstcheck <= 0;
        end else if (ingress_port_tvalid && (inter_frame_wait == 0)) begin
            if(input_counter == 0) begin
                if({reg_file[1], reg_file[0]} == ingress_port_tdata) begin
                    dstcheck <= dstcheck + 8'h01;
                end
                checksum <= 0;
                dstcheck <= 0;
                input_counter <= input_counter + 8'h01;
            end else if(input_counter == 1) begin
                if({reg_file[3], reg_file[2]} == ingress_port_tdata) begin
                    dstcheck <= dstcheck + 8'h01;
                end
                input_counter <= input_counter + 8'h01;
            end else if(input_counter == 2) begin
                if({reg_file[5], reg_file[4]} == ingress_port_tdata) begin
                    dstcheck <= dstcheck + 8'h01;
                end
                input_counter <= input_counter + 8'h01;
            end else if(input_counter >= 3 && input_counter <= 6) begin
                input_counter <= input_counter + 8'h01;
            end else if(input_counter >= 7 && !ingress_port_tlast) begin
                checksum <= checksum + {16'h00, ingress_port_tdata};
                input_counter <= input_counter + 8'h01;
            end else if(input_counter >= 7 && ingress_port_tlast) begin
                checksum <= checksum + {16'h00, ingress_port_tdata};
                frame_counter <= frame_counter + 8'h01;
                input_counter <= 0;
            end
        end
    end
endmodule
```

8.4 Packet Filter

Listing 7: packet_filter/dest_calculator.sv

```
`ifdef VERILATOR
`include "packet_filter.svh"
`else
`include "../include/packet_filter.svh"
`include "../include/synth_defs.svh"
`endif
`include "filter_defs.svh"

`timescale 1 ps / 1 ps
module dest_calculator #(
    parameter STUBBING = `STUBBING_PASSTHROUGH
) (
    input  logic clk,
    input  logic reset,

    input  packet_source_t dst_mac_pkt,

    output dest_source_t dest
);

    logic [15:0] dst_mac [2:0];
    logic [15:0] next_dst_mac [2:0];
    logic [ 1:0] cnt, next_cnt;
    logic        valid_output, invalid_dst_mac;

    // latch new state
    always_ff @(posedge clk) begin
        if (reset) begin
            dst_mac[0] <= 16'h0;
            dst_mac[1] <= 16'h0;
            dst_mac[2] <= 16'h0;
            cnt <= 2'b0;
        end else begin
            dst_mac <= next_dst_mac;
            cnt <= next_cnt;
        end
    end

    // shift in new bytes when scanning the destination MAC field
    always_comb begin: g_next_dst
        if (dst_mac_pkt.tvalid) begin
            next_dst_mac[2] = dst_mac[1];
            next_dst_mac[1] = dst_mac[0];
            next_dst_mac[0] = dst_mac_pkt.tdata;
        end else begin
```

```
            next_dst_mac = dst_mac;
        end
    end

    // increment counter
    always_comb begin: g_next_cnt
        if (cnt === 2'b11) begin
            next_cnt = dst_mac_pkt.tvalid ? 2'b01 : 2'b00;
        end else if (dst_mac_pkt.tvalid) begin
            next_cnt = cnt + 1;
        end else begin
            next_cnt = cnt;
        end
    end

    // set invalid bit when tdest_valid is high
    always_comb begin: g_invalid_detection
        if (cnt === 2'b11) begin
            valid_output = 1'b1;
            if (dst_mac[2][15:14] === 2'b11) begin
                invalid_dst_mac = 1'b1;
            end else begin
                invalid_dst_mac = 1'b0;
            end
        end else begin
            valid_output = 1'b0;
            invalid_dst_mac = 1'b0;
        end
    end

    assign dest.tdata  = dst_mac[0][1:0];
    assign dest.tvalid = valid_output;
    assign dest.tuser  = invalid_dst_mac;

endmodule
```

Listing 8: packet_filter/filter_defs.svh

```
`ifndef _FILTER_DEFS_SVH
`define _FILTER_DEFS_SVH

/* Integration testing setup. */

`ifndef INTG_TESTING_1
    `ifdef TOP_TESTING
        `define INTG_TESTING_1
    `endif
`endif
```

```
/* verilator lint_off UNPACKED */

// Frame status broadcast in input filter
typedef struct {
    logic scan_frame;
    logic scan_dst_mac;
    logic scan_src_mac;
    logic scan_type;
    logic scan_payload;
} frame_status;

// Downstream packet interface (always ready)
typedef struct {
    logic [15:0] tdata;
    logic        tvalid;
} packet_source_t;

// Dest data interface
typedef struct {
    logic [1:0] tdata;
    logic       tvalid;
    logic       tuser;
} dest_source_t;

// Drop indication interface
typedef struct {
    logic       tvalid;
    logic       tuser;
} drop_source_t;

/* verilator lint_on UNPACKED */

`endif // _FILTER_DEFS_SVH
```

Listing 9: packet_filter/frame_buffer.sv

```
`ifdef VERILATOR
`include "packet_filter.svh"
`else
`include "../include/packet_filter.svh"
`include "../include/synth_defs.svh"
`endif
`include "filter_defs.svh"

`timescale 1 ps / 1 ps
module frame_buffer #(
    parameter STUBBING = `STUBBING_PASSTHROUGH,
```

```
    parameter ALMOST_FULL_THRESHOLD = 10,
    parameter ADDR_WIDTH = 11,
    parameter NUM_CYCLONE_5CSEMA5_BLOCKS = 4
) (
    // clock and reset
    input  logic clk,
    input  logic reset,

    // ingress frame control
    input  axis_source_t ingress_pkt,
    input  logic         scan_frame,
    input  logic         drop_write,
    output logic         almost_full,
    output logic [ADDR_WIDTH:0] frame_wptr,

    // read frame control
    input  logic         frame_ren,
    input  logic         frame_rrst,
    input  logic [ADDR_WIDTH:0] frame_rst_rptr,
    output logic [ADDR_WIDTH:0] frame_rptr,
    output logic [19:0]  frame_rdata,
    output logic         last_entry
);

    // ingress frame control
    logic prev_scan_frame;
    logic frame_wen;
    logic frame_wrst;
    logic [ADDR_WIDTH:0] frame_rst_wptr;
    logic [ADDR_WIDTH:0] next_frame_rptr;
    logic [19:0] frame_wdata;
    logic frame_full;
    logic next_almost_full;
    logic [ADDR_WIDTH:0] frame_ptr_diff;

    // egress frame control
    logic next_last_entry;

    // Register logic
    always_ff @(posedge clk) begin
        frame_wdata[19:16] <= 4'b0;
        if (reset) begin
            prev_scan_frame   <= 1'b0;
            frame_rst_wptr    <= '0;
            almost_full       <= 1'b0;
            frame_wdata[15:0] <= '0;
            frame_wen         <= 1'b0;
            frame_wrst        <= 1'b0;
```

```
        end else begin
            // Frame start logic
            prev_scan_frame <= scan_frame;
            if (~prev_scan_frame & scan_frame) begin
                frame_rst_wptr <= frame_wptr;
            end else begin
                frame_rst_wptr <= frame_rst_wptr;
            end

            // Capacity logic
            almost_full <= next_almost_full;
            //last_entry  <= next_last_entry;

            // write logic
            frame_wdata[15:0] <= ingress_pkt.tdata;
            frame_wen <= ingress_pkt.tvalid & scan_frame & ~drop_write;
            frame_wrst <= drop_write;
        end
    end

    // Capacity logic
    assign next_frame_rptr = frame_rptr + 1;
    //assign next_last_entry = (next_frame_rptr === frame_wptr) ? 1'b1 : 1'b0;
    assign last_entry = (next_frame_rptr === frame_wptr) ? 1'b1 : 1'b0;
    assign frame_ptr_diff = frame_wptr - frame_rptr;
    always_comb begin
`ifdef VERILATOR
        if ({{(32-ADDR_WIDTH){1'b0}}, frame_ptr_diff[ADDR_WIDTH-1:0]} >= ALMOST_FULL_THRESHOLI
`else
        if (frame_ptr_diff[ADDR_WIDTH-1:0] >= ALMOST_FULL_THRESHOLD) begin
`endif
            next_almost_full = 1'b1;
        end else begin
            next_almost_full = frame_full;
        end
    end

    /*
     * Frame FIFO.
     *
     * Using 4 blocks = 512*4 = 2048x20b words
     * Must address 2048 words => 11-bit address (12-bit cursor)
     * Can store 2.69829 full-sized frames (1518 Bytes = 759 queue words each)
     * Can store 64 min-sized frames (64 Bytes = 32 queue words each)
     *
     * TODO: add parity bits to pad 16-bit streaming words to 20-bit memory words
     */
    fifo_sync #(
```

```
        .ADDR_WIDTH(ADDR_WIDTH),
        .W_EL(20),
        .NUM_CYCLONE_5CSEMA5_BLOCKS(NUM_CYCLONE_5CSEMA5_BLOCKS),
        .CAN_RESET_POINTERS(1),
        .RDATA_PIPELINE(0)
    ) u_frame_fifo (
        .clk      (clk),
        .reset    (reset),
        .ren      (frame_ren),      // from switch FSM
        .rdata    (frame_rdata),    // to egress
/* verilator lint_off PINCONNECTEMPTY */
        .empty    (),               // open
/* verilator lint_on PINCONNECTEMPTY */
        .wdata    (frame_wdata),    // from ingress
        .wen      (frame_wen),      // from switch FSM
        .full     (frame_full),
        .rrst     (frame_rrst),     //
        .wrst     (frame_wrst),     //
        .rst_rptr (frame_rst_rptr), // from switch FSM
        .rst_wptr (frame_rst_wptr), // from switch FSM
        .rptr     (frame_rptr),     // to switch FSM
        .wptr     (frame_wptr)      // to switch FSM
    );

`ifdef VERILATOR
    integer fifo_rst_rptr, fifo_rst_wptr, fifo_rptr, fifo_wptr;
    integer wcnt, rcnt, size;

    assign fifo_rst_rptr = {{(32-ADDR_WIDTH-1){1'b0}}, u_frame_fifo.rst_rptr};
    assign fifo_rst_wptr = {{(32-ADDR_WIDTH-1){1'b0}}, u_frame_fifo.rst_wptr};
    assign fifo_rptr = {{(32-ADDR_WIDTH-1){1'b0}}, u_frame_fifo.rptr};
    assign fifo_wptr = {{(32-ADDR_WIDTH-1){1'b0}}, u_frame_fifo.wptr};
    assign size = wcnt - rcnt;
    always_ff @(posedge clk) begin
        if (reset) begin
            rcnt <= 0;
            wcnt <= 0;
        end else begin
            if (u_frame_fifo.rrst && u_frame_fifo.wrst) begin
                rcnt <= 0;
                wcnt <= fifo_rst_wptr - fifo_rst_rptr;
            end else if (u_frame_fifo.rrst && ~u_frame_fifo.wrst) begin
                rcnt <= 0;
                if (u_frame_fifo.wen && ~u_frame_fifo.full) begin
                    wcnt <= fifo_wptr - fifo_rst_rptr + 1;
                end else begin
                    wcnt <= fifo_wptr - fifo_rst_rptr;
                end
```

```
            end else if (~u_frame_fifo.rrst && u_frame_fifo.wrst) begin
                if (u_frame_fifo.ren && ~u_frame_fifo.empty) begin
                    rcnt <= 1;
                end else begin
                    rcnt <= 0;
                end
                wcnt <= fifo_rst_wptr - fifo_rptr;
            end else begin
                if (u_frame_fifo.wen && ~u_frame_fifo.full) begin
                    wcnt <= wcnt + 1;
                end
                if (u_frame_fifo.ren && ~u_frame_fifo.empty) begin
                    rcnt <= rcnt + 1;
                end
            end
        end
    end

`ifdef ASSERT
    assertion_frame_buffer_almost_full : assert property(
        @(posedge clk) disable iff (reset)
        (size >= ALMOST_FULL_THRESHOLD) |=> almost_full
    ) else $error("Failed assertion");

    assertion_frame_buffer_last_entry : assert property(
        @(posedge clk) disable iff (reset)
        (size === 1) |-> last_entry
    ) else $error("Failed assertion");
`endif
`endif

endmodule
```

Listing 10: packet_filter/ingress_filter.sv

```
`ifdef VERILATOR
`include "packet_filter.svh"
`else
`include "../include/packet_filter.svh"
`include "../include/synth_defs.svh"
`endif
`include "filter_defs.svh"

`ifdef TOP_TESTING

// Integrate preliminary_processor and request_buffer
`timescale 1 ps / 1 ps
module ingress_filter #(
```

```
        parameter STUBBING = `STUBBING_PASSTHROUGH,
        parameter ALMOST_FULL_THRESHOLD = 10,
        parameter ADDR_WIDTH = 11,
        parameter NUM_CYCLONE_5CSEMA5_BLOCKS = 4,
        parameter TIMEOUT_CTR_WIDTH = 3
    ) (
                input  logic        clk,
                input  logic        reset,

                input  logic        en,

                input  axis_source_t ingress_source,
                output axis_sink_t   ingress_sink,

                output axis_d_source_t egress_source,
                input  axis_d_sink_t   egress_sink,

                // status signals
                output logic drop_write,
                output logic timeout
        );

    /*
     * Wires
     */

    axis_source_t ingress_source_q;

    /*
     * Register logic
     */
generate
if (STUBBING == `STUBBING_PASSTHROUGH) begin: g_passthrough
    always_ff @(posedge clk) begin
        if (reset) begin
            egress_source.tvalid <= 1'b0;
            egress_source.tdata <= 16'b0;
            egress_source.tdest <= 2'b0;
            egress_source.tlast <= 1'b0;
            ingress_sink.tready <= 1'b0;
        end else begin
            egress_source.tvalid <= ingress_source.tvalid;
            egress_source.tdata <= ingress_source.tdata;
            egress_source.tdest <= 2'b0;
            egress_source.tlast <= ingress_source.tlast;
            ingress_sink <= egress_sink;
        end
    end
```

```
end else begin: g_functional

    axis_source_t ingress_pkt;
    logic almost_full;
    frame_status status;
    dest_source_t frame_dest;
    drop_source_t frame_type;

    preliminary_processor #(
        .STUBBING(STUBBING)
    ) u_processor (
        .clk(clk),
        .reset(reset),
        .ingress_source(ingress_source),
        .ingress_sink(ingress_sink),
        .ingress_pkt(ingress_pkt),
        .drop_write(drop_write),
        .almost_full(almost_full),
        .status(status),
        .frame_dest(frame_dest),
        .frame_type(frame_type)
    );

    request_buffer #(
        .STUBBING(STUBBING),
        .ALMOST_FULL_THRESHOLD(ALMOST_FULL_THRESHOLD),
        .ADDR_WIDTH(ADDR_WIDTH),
        .NUM_CYCLONE_5CSEMA5_BLOCKS(NUM_CYCLONE_5CSEMA5_BLOCKS),
        .TIMEOUT_CTR_WIDTH(TIMEOUT_CTR_WIDTH)
    ) u_requester (
        .clk(clk),
        .reset(reset),
        .status(status),
        .frame_type(frame_type),
        .frame_dest(frame_dest),
        .ingress_pkt(ingress_pkt),
        .drop_write(drop_write),
        .almost_full(almost_full),
        .timeout(timeout),
        .egress_source(egress_source),
        .egress_sink(egress_sink)
    );

end
endgenerate

endmodule
```

```
`endif
```

Listing 11: packet_filter/input_fsm.sv

```
`ifdef VERILATOR
`include "packet_filter.svh"
`else
`include "../include/packet_filter.svh"
`include "../include/synth_defs.svh"
`endif
`include "filter_defs.svh"

`timescale 1 ps / 1 ps
module input_fsm #(
    parameter STUBBING = `STUBBING_PASSTHROUGH
) (
    // Clock and reset
    input  logic clk,
    input  logic reset,

    // Ingress AXIS interface
        input  axis_source_t ingress_source,
        output axis_sink_t   ingress_sink,

    // Input filter status
    input  logic drop_current,
    input  logic almost_full,

    // Output status
    output axis_source_t ingress_pkt,
    output logic         incomplete_frame,
    output frame_status  status
);

    // State definitions.
    localparam IDLE    = 3'b000; // waiting for start frame delimeter (SFD)
    localparam DST_MAC = 3'b010; // scanning destination MAC
    localparam SRC_MAC = 3'b011; // scanning source MAC
    localparam TYPE    = 3'b111; // scanning type field
    localparam FLUSH   = 3'b110; // flush after premature drop
    localparam PAYLOAD = 3'b101; // scanning payload
    localparam MASK    = 3'b100; // mask a dropped packet

    // State signals
    logic [2:0] state, state_d, next_state;
    logic handshake_complete, state_transition_en, next_state_transition_en;
    logic [1:0] cycle_counter;
```

```
    // Status definitions.
    localparam FRAME_STATUS_IDLE    = 5'b00000;
    localparam FRAME_STATUS_DST_MAC = 5'b00011;
    localparam FRAME_STATUS_SRC_MAC = 5'b00101;
    localparam FRAME_STATUS_TYPE    = 5'b01001;
    localparam FRAME_STATUS_PAYLOAD = 5'b10001;

    // Status signals
    logic end_of_frame;
    logic sfd_received;
    logic next_tready;
    logic [4:0] frame_status_str;

    // Ingress interface
    assign handshake_complete = next_tready & ingress_source.tvalid;
    always_comb begin
        // assert backpressure when not enough space for a full frame
        //   and not done with the current frame if a frame is ingressing
        if (almost_full) begin
            // hold ready until completed transaction
            next_tready = ingress_sink.tready & ~end_of_frame;
        end else begin
            // turn on with new request or while waiting for request to terminate
            next_tready = ingress_source.tvalid & ~(ingress_sink.tready & ingress_pkt.tlast);
        end
    end
    always_ff @(posedge clk) begin: p_ingress
        if (reset) begin
            ingress_pkt.tvalid <= 1'b0;
            ingress_pkt.tdata <= 16'b0;
            ingress_pkt.tlast <= 1'b0;
            ingress_sink.tready <= 1'b0;
        end else begin
            // save packet
            ingress_pkt.tvalid <= handshake_complete;
            ingress_pkt.tdata  <= ingress_source.tdata;
            ingress_pkt.tlast  <= ingress_source.tlast;
            ingress_sink.tready <= next_tready;
        end
    end

    // Compute duration in current state (only relevant for headers)
    always_ff @(posedge clk) begin: p_cnt
        if (reset) begin
            cycle_counter <= 2'b00;
        end else begin
            if (next_state !== state) begin
```

```
            // reset when transition to next field
            cycle_counter <= 2'b00;
        end else if (next_state_transition_en) begin
            // increment
            cycle_counter <= cycle_counter + 1;
        end else begin
            cycle_counter <= cycle_counter;
        end
    end
end

// Propagate next state to state
assign next_state_transition_en =
    next_tready || ingress_sink.tready || (next_state === FLUSH);
always_ff @(posedge clk) begin: p_propagate_next_state
    if (reset) begin
        state <= IDLE;
        state_transition_en <= 1'b0;
    end else begin
        state <= state_d;
        state_transition_en <= next_state_transition_en;
    end
end

// Calculate next state
assign state_d =
drop_current // all states transition to mask an invalid frame
    // do not wait for a second last signal from a frame that is already complete
    ? (ingress_pkt.tlast || state === IDLE
        ? IDLE : MASK)
    // otherwise, transition to next state
    : (state_transition_en
        ? next_state : state);
assign sfd_received = ingress_pkt.tdata === `ETH_SFD;
assign end_of_frame = ~ingress_pkt.tvalid | ingress_pkt.tlast;
always_comb begin: p_next_state
    next_state = state;
    case (state)
    IDLE: begin
        if (sfd_received) begin
            next_state = DST_MAC;
        end
    end
    DST_MAC: begin
        if (end_of_frame) begin
            next_state = FLUSH;
        end else if (cycle_counter === 2'b10) begin
            next_state = SRC_MAC;
```

```
            end
        end
        SRC_MAC: begin
            if (end_of_frame) begin
                next_state = FLUSH;
            end else if (cycle_counter === 2'b10) begin
                next_state = TYPE;
            end
        end
        TYPE: begin
            if (end_of_frame) begin
                next_state = FLUSH;
            end else begin
                next_state = PAYLOAD;
            end
        end
        FLUSH: begin
            next_state = IDLE;
        end
        PAYLOAD: begin
            if (end_of_frame) begin
                next_state = IDLE;
            end
        end
        MASK: begin
            if (end_of_frame) begin
                next_state = IDLE;
            end
        end
        default: begin
            next_state = IDLE;
        end
        endcase
    end

    // Generate output
    assign incomplete_frame = state === FLUSH ? 1'b1 : 1'b0;
    assign status.scan_frame   = frame_status_str[0] | (sfd_received & ingress_pkt.tvalid);
    //assign status.scan_frame   = frame_status_str[0];
    assign status.scan_dst_mac = frame_status_str[1];
    assign status.scan_src_mac = frame_status_str[2];
    assign status.scan_type    = frame_status_str[3];
    assign status.scan_payload = frame_status_str[4];
    always_comb begin: p_output
        case (state)
        IDLE: begin
            frame_status_str = FRAME_STATUS_IDLE;
        end
```

```
        DST_MAC: begin
            frame_status_str = FRAME_STATUS_DST_MAC;
        end
        SRC_MAC: begin
            frame_status_str = FRAME_STATUS_SRC_MAC;
        end
        TYPE: begin
            frame_status_str = FRAME_STATUS_TYPE;
        end
        PAYLOAD: begin
            frame_status_str = FRAME_STATUS_PAYLOAD;
        end
        FLUSH: begin
            frame_status_str = FRAME_STATUS_IDLE;
        end
        MASK: begin
            frame_status_str = FRAME_STATUS_IDLE;
        end
        default: begin
            frame_status_str = FRAME_STATUS_IDLE;
        end
        endcase
    end

`ifdef VERILATOR
    function string conv_state_to_str( logic [2:0] arg_state );
        case (arg_state)
        IDLE: begin
            conv_state_to_str = "IDLE";
        end
        DST_MAC: begin
            conv_state_to_str = "DST_MAC";
        end
        SRC_MAC: begin
            conv_state_to_str = "SRC_MAC";
        end
        TYPE: begin
            conv_state_to_str = "TYPE";
        end
        PAYLOAD: begin
            conv_state_to_str = "PAYLOAD";
        end
        FLUSH: begin
            conv_state_to_str = "FLUSH";
        end
        MASK: begin
            conv_state_to_str = "MASK";
        end
```

```
        default: begin
            conv_state_to_str = "dne";
        end
        endcase
    endfunction

    string state_str;
    assign state_str = conv_state_to_str(state);

    always @(posedge clk) begin
        $display("state is %s\n", state_str);
    end
`endif

`ifdef ASSERT
    /* Assertions */

    // assert tready provided one cycle after first tvalid
    assertion_input_fsm_tready_delay : assert property(
        @(posedge clk) disable iff (reset)
        ~ingress_source.tvalid |=> ~ingress_sink.tready
    ) else $error("Assertion failed");

    // assert grant is held for an entire frame
    assertion_input_fsm_frame_grant : assert property(
        @(posedge clk) disable iff (reset)
        ingress_source.tvalid & ingress_sink.tready & ~ingress_source.tlast
            |=> ingress_sink.tready || ingress_source.tlast
    ) else $error($sformatf("assertion_input_fsm_frame_grant failed at %0t", $realtime));

    // do not provide grants if almost full
    assertion_input_fsm_almost_full : assert property(
        @(posedge clk) disable iff (reset)
        almost_full && (~ingress_source.tvalid || ingress_pkt.tlast)
            |=> ~ingress_sink.tready
    ) else $error($sformatf("assertion_input_fsm_almost_full failed at %0t", $realtime));

    // assert do not accept packets when not ready
    assertion_input_fsm_tready : assert property(
        @(posedge clk) disable iff (reset)
        ~ingress_sink.tready |-> ~ingress_pkt.tvalid
    ) else $error("Failed assertion");

    // assert input has a gap between frames
    assertion_input_fsm_input_frame_gap : assert property(
        @(posedge clk) disable iff (reset)
        ingress_source.tlast & ingress_source.tvalid & ingress_sink.tready |=> ~ingress_source
    ) else $error("Failed assertion");
```

```
    // assert SFD is present
    assertion_input_fsm_sfd : assert property(
        @(posedge clk) disable iff (reset)
        $rose(status.scan_frame) & ingress_pkt.tvalid |-> ingress_pkt.tdata === `ETH_SFD
    ) else $error("Failed assertion");


`ifdef VERILATOR
    logic current_frame_dropped;
    reg_set_clear r_current_frame_dropped
        (clk, reset, drop_current, ingress_source.tlast, current_frame_dropped);

    // assert mask a dropped frame (do not broadcast frame status until tlast asserted)
    assertion_input_fsm_mask_dropped : assert property(
        @(posedge clk) disable iff (reset)
        current_frame_dropped |-> frame_status_str === FRAME_STATUS_IDLE || ingress_source.tla
    ) else $error("Failed assertion");

    // assert output has valid frames
    assertion_input_fsm_output_pkt : assert property(
        @(posedge clk) disable iff (reset)
        ~current_frame_dropped & $past(ingress_source.tvalid, 1) & ingress_sink.tready |-> ing
    ) else $error("Failed assertion");
`endif
`endif

endmodule
```

Listing 12: packet_filter/packet_filter.sv

```
// packet_filter.sv

/**
 * Register mapping
 *
 * Addr / mode | Name                | Meaning
 *         0RW |  ingress_port_mask  |  Enable signal for input ports (active-high).
 *         4R  |         in_pkts_in0 |  Number of 16-byte packets received by ingress port 0
 *         5R  |         in_pkts_in1 |  Number of 16-byte packets received by ingress port 1
 *         6R  |         in_pkts_in2 |  Number of 16-byte packets received by ingress port 2
 *         7R  |         in_pkts_in3 |  Number of 16-byte packets received by ingress port 3
 *         8R  |     transf_pkts_in0 |  Number of 16-byte packets transferred from ingress port
 *         9R  |     transf_pkts_in1 |  Number of 16-byte packets transferred from ingress port
 *        10R  |     transf_pkts_in2 |  Number of 16-byte packets transferred from ingress port
 *        11R  |     transf_pkts_in3 |  Number of 16-byte packets transferred from ingress port
 *        12R  |       in_frames_in0 |  Number of complete frames received by ingress port 0
 *        13R  |       in_frames_in1 |  Number of complete frames received by ingress port 1
 *        14R  |       in_frames_in2 |  Number of complete frames received by ingress port 2
```

```
 *       15R   |       in_frames_in3 |  Number of complete frames received by ingress port 3
 *       16R   |  transf_frames_in0  |  Number of frames transferred from ingress port 0
 *       17R   |  transf_frames_in1  |  Number of frames transferred from ingress port 1
 *       18R   |  transf_frames_in2  |  Number of frames transferred from ingress port 2
 *       19R   |  transf_frames_in3  |  Number of frames transferred from ingress port 3
 *       20R   |     inv_frames_in0  |  Number of frames detected as invalid from ingress port
 *       21R   |     inv_frames_in1  |  Number of frames detected as invalid from ingress port
 *       22R   |     inv_frames_in2  |  Number of frames detected as invalid from ingress port
 *       23R   |     inv_frames_in3  |  Number of frames detected as invalid from ingress port
 *       24R   |    drop_frames_in0  |  Number of dropped frames from ingress port 0
 *       25R   |    drop_frames_in1  |  Number of dropped frames from ingress port 1
 *       26R   |    drop_frames_in2  |  Number of dropped frames from ingress port 2
 *       27R   |    drop_frames_in3  |  Number of dropped frames from ingress port 3
 */

`ifdef VERILATOR
`include "packet_filter.svh"
`else
`include "../include/packet_filter.svh"
`include "../include/synth_defs.svh"
`endif
`include "filter_defs.svh"

`ifdef TOP_TESTING

`timescale 1 ps / 1 ps
module packet_filter #(
        parameter STUBBING = `STUBBING_FUNCTIONAL,
        // almost full when cannot store another full-sized frame
        parameter ALMOST_FULL_THRESHOLD = 760,
        // address 2048 words
        parameter ADDR_WIDTH = 11,
        parameter NUM_CYCLONE_5CSEMA5_BLOCKS = 4,
        // timeout after 512 waiting cycles
        parameter TIMEOUT_CTR_WIDTH = 9
    ) (
                input  wire        clk,                    //                    clock.clk
                input  wire        reset,                  //                    reset.reset
                input  wire [31:0] writedata,              //          avalon_slave_0.writedata
                input  wire        write,                  //                        .write
                input  wire        chipselect,             //                        .chipselect
                input  wire [7:0]  address,                //                        .address
                input  wire        read,                   //                        .read
                output wire [31:0] readdata,               //                        .readdata
                input  wire [15:0] ingress_port_0_tdata,   //          ingress_port_0.tdata
                input  wire        ingress_port_0_tvalid,  //                        .tvalid
                output wire        ingress_port_0_tready,  //                        .tready
                input  wire        ingress_port_0_tlast,   //                        .tlast
```

```
            input  wire [15:0] ingress_port_1_tdata,  //          ingress_port_1.tdata
            input  wire        ingress_port_1_tlast,  //                        .tlast
            input  wire        ingress_port_1_tvalid, //                        .tvalid
            output wire        ingress_port_1_tready, //                        .tready
            input  wire [15:0] ingress_port_2_tdata,  //          ingress_port_2.tdata
            input  wire        ingress_port_2_tlast,  //                        .tlast
            input  wire        ingress_port_2_tvalid, //                        .tvalid
            output wire        ingress_port_2_tready, //                        .tready
            input  wire [15:0] ingress_port_3_tdata,  //          ingress_port_3.tdata
            output wire        ingress_port_3_tready, //                        .tready
            input  wire        ingress_port_3_tlast,  //                        .tlast
            input  wire        ingress_port_3_tvalid, //                        .tvalid
            output wire [15:0] egress_port_0_tdata,   //           egress_port_0.tdata
            output wire        egress_port_0_tlast,   //                        .tlast
            input  wire        egress_port_0_tready,  //                        .tready
            output wire        egress_port_0_tvalid,  //                        .tvalid
            output wire [1:0]  egress_port_0_tdest,   //                        .tdest
            output wire [15:0] egress_port_1_tdata,   //           egress_port_1.tdata
            output wire        egress_port_1_tlast,   //                        .tlast
            input  wire        egress_port_1_tready,  //                        .tready
            output wire        egress_port_1_tvalid,  //                        .tvalid
            output wire [1:0]  egress_port_1_tdest,   //                        .tdest
            output wire [15:0] egress_port_2_tdata,   //           egress_port_2.tdata
            output wire        egress_port_2_tlast,   //                        .tlast
            input  wire        egress_port_2_tready,  //                        .tready
            output wire        egress_port_2_tvalid,  //                        .tvalid
            output wire [1:0]  egress_port_2_tdest,   //                        .tdest
            output wire [15:0] egress_port_3_tdata,   //           egress_port_3.tdata
            output wire        egress_port_3_tlast,   //                        .tlast
            input  wire        egress_port_3_tready,  //                        .tready
            output wire        egress_port_3_tvalid,  //                        .tvalid
            output wire [1:0]  egress_port_3_tdest,   //                        .tdest
            output wire        irq                    // packet_filter_interrupt.irq
    );

// registers
logic [31:0] readdata_int;
logic [`NUM_INGRESS_PORTS-1:0] ingress_port_mask;

// counter increment signals
logic [`NUM_INGRESS_PORTS-1:0] invalid_frame;
logic [`NUM_INGRESS_PORTS-1:0] timeout_req;

// counters
logic [1:0] counter_address;
logic [31:0]       in_pkts [`NUM_INGRESS_PORTS-1:0];
logic [31:0]   transf_pkts [`NUM_INGRESS_PORTS-1:0];
logic [31:0]     in_frames [`NUM_INGRESS_PORTS-1:0];
```

```
logic [31:0] transf_frames [`NUM_INGRESS_PORTS-1:0];
logic [31:0]    inv_frames [`NUM_INGRESS_PORTS-1:0];
logic [31:0]   drop_frames [`NUM_INGRESS_PORTS-1:0];

/* Convert between Avalon wires and internal structures. */

    // internal AXIS wires
    axis_source_t   ingress_port_source [`NUM_INGRESS_PORTS-1:0];
    axis_sink_t     ingress_port_sink   [`NUM_INGRESS_PORTS-1:0];
    axis_d_source_t egress_port_source  [`NUM_INGRESS_PORTS-1:0];
    axis_d_sink_t   egress_port_sink    [`NUM_INGRESS_PORTS-1:0];

    // intermediate assignments
assign ingress_port_source[0].tvalid = ingress_port_0_tvalid;
assign ingress_port_source[0].tdata  = ingress_port_0_tdata;
assign ingress_port_source[0].tlast  = ingress_port_0_tlast;
assign ingress_port_source[1].tvalid = ingress_port_1_tvalid;
assign ingress_port_source[1].tdata  = ingress_port_1_tdata;
assign ingress_port_source[1].tlast  = ingress_port_1_tlast;
assign ingress_port_source[2].tvalid = ingress_port_2_tvalid;
assign ingress_port_source[2].tdata  = ingress_port_2_tdata;
assign ingress_port_source[2].tlast  = ingress_port_2_tlast;
assign ingress_port_source[3].tvalid = ingress_port_3_tvalid;
assign ingress_port_source[3].tdata  = ingress_port_3_tdata;
assign ingress_port_source[3].tlast  = ingress_port_3_tlast;
    assign egress_port_sink[0].tready = egress_port_0_tready;
    assign egress_port_sink[1].tready = egress_port_1_tready;
    assign egress_port_sink[2].tready = egress_port_2_tready;
    assign egress_port_sink[3].tready = egress_port_3_tready;

// output AXIS assignments
    assign ingress_port_0_tready = ingress_port_sink[0].tready;
    assign ingress_port_1_tready = ingress_port_sink[1].tready;
    assign ingress_port_2_tready = ingress_port_sink[2].tready;
    assign ingress_port_3_tready = ingress_port_sink[3].tready;
    assign egress_port_0_tvalid  = egress_port_source[0].tvalid;
    assign egress_port_0_tdest   = egress_port_source[0].tdest;
    assign egress_port_0_tdata   = egress_port_source[0].tdata;
    assign egress_port_0_tlast   = egress_port_source[0].tlast;
    assign egress_port_1_tvalid  = egress_port_source[1].tvalid;
    assign egress_port_1_tdest   = egress_port_source[1].tdest;
    assign egress_port_1_tdata   = egress_port_source[1].tdata;
    assign egress_port_1_tlast   = egress_port_source[1].tlast;
    assign egress_port_2_tvalid  = egress_port_source[2].tvalid;
    assign egress_port_2_tdest   = egress_port_source[2].tdest;
    assign egress_port_2_tdata   = egress_port_source[2].tdata;
    assign egress_port_2_tlast   = egress_port_source[2].tlast;
    assign egress_port_3_tvalid  = egress_port_source[3].tvalid;
```

```
        assign egress_port_3_tdest   = egress_port_source[3].tdest;
        assign egress_port_3_tdata   = egress_port_source[3].tdata;
        assign egress_port_3_tlast   = egress_port_source[3].tlast;

    /* Functionality. */

generate
if (STUBBING == `STUBBING_PASSTHROUGH) begin: g_passthrough

    // output AXIS assignments
        assign ingress_port_sink[0].tready  = egress_port_sink[0].tready;
        assign ingress_port_sink[1].tready  = egress_port_sink[1].tready;
        assign ingress_port_sink[2].tready  = egress_port_sink[2].tready;
        assign ingress_port_sink[3].tready  = egress_port_sink[3].tready;
        assign egress_port_source[0].tvalid = ingress_port_source[0].tvalid;
        assign egress_port_source[0].tdest  = '0;
        assign egress_port_source[0].tdata  = ingress_port_source[0].tdata;
        assign egress_port_source[0].tlast  = ingress_port_source[0].tlast;
        assign egress_port_source[1].tvalid = ingress_port_source[1].tvalid;
        assign egress_port_source[1].tdest  = '0;
        assign egress_port_source[1].tdata  = ingress_port_source[1].tdata;
        assign egress_port_source[1].tlast  = ingress_port_source[1].tlast;
        assign egress_port_source[2].tvalid = ingress_port_source[2].tvalid;
        assign egress_port_source[2].tdest  = '0;
        assign egress_port_source[2].tdata  = ingress_port_source[2].tdata;
        assign egress_port_source[2].tlast  = ingress_port_source[2].tlast;
        assign egress_port_source[3].tvalid = ingress_port_source[3].tvalid;
        assign egress_port_source[3].tdest  = '0;
        assign egress_port_source[3].tdata  = ingress_port_source[3].tdata;
        assign egress_port_source[3].tlast  = ingress_port_source[3].tlast;

        assign invalid_frame = '0;
        assign timeout_req = '0;

end else begin: g_functional

        ingress_filter #(
            .STUBBING(STUBBING),
            .ALMOST_FULL_THRESHOLD(ALMOST_FULL_THRESHOLD),
            .ADDR_WIDTH(ADDR_WIDTH),
            .NUM_CYCLONE_5CSEMA5_BLOCKS(NUM_CYCLONE_5CSEMA5_BLOCKS),
            .TIMEOUT_CTR_WIDTH(TIMEOUT_CTR_WIDTH)
        ) u_filter[`NUM_INGRESS_PORTS-1:0] (
            .clk   (clk),
            .reset(reset),

            .en(ingress_port_mask),
```

```
            .ingress_source(ingress_port_source),
            .ingress_sink  (ingress_port_sink),

            .egress_source(egress_port_source),
            .egress_sink  (egress_port_sink),

            .drop_write(invalid_frame),
            .timeout(timeout_req)
        );
end
endgenerate

    /* Register interface. */

    // register write interface
    always_ff @(posedge clk) begin
        if (reset) begin
            ingress_port_mask <= 4'b0;
        end else if (chipselect && write) begin
            case (address)
                8'h0 : ingress_port_mask <= writedata[3:0];
                default : ingress_port_mask <= ingress_port_mask;
            endcase
        end
    end

    // register read interface
    assign readdata = readdata_int;
    assign counter_address = address[1:0];
    always_ff @(posedge clk) begin
        if (reset) begin
            readdata_int <= 32'b0;
        end else if (chipselect && read) begin
            case (address[7:2])
                6'h0 : readdata_int <= {28'b0, ingress_port_mask};
                6'h1 : readdata_int <= in_pkts[counter_address];
                6'h2 : readdata_int <= transf_pkts[counter_address];
                6'h3 : readdata_int <= in_frames[counter_address];
                6'h4 : readdata_int <= transf_frames[counter_address];
                6'h5 : readdata_int <= inv_frames[counter_address];
                6'h6 : readdata_int <= drop_frames[counter_address];
                default : readdata_int <= '0;
            endcase
        end
    end

    // counters
    genvar i;
```

```
generate
for (i = 0; i < 4; ++i) begin: g_ingress_count
    always_ff @(posedge clk) begin
        if (reset) begin
            in_pkts[i] <= 32'b0;
            transf_pkts[i] <= 32'b0;
            in_frames[i] <= 32'b0;
            transf_frames[i] <= 32'b0;
            inv_frames[i] <= 32'b0;
            drop_frames[i] <= 32'b0;
        end else begin
            // accepted ingress packets
            if (ingress_port_source[i].tvalid & ingress_port_sink[i].tready) begin
                in_pkts[i] <= in_pkts[i] + 1;
            end else begin
                in_pkts[i] <= in_pkts[i];
            end

            // transferred ingress packets
            if (egress_port_source[i].tvalid & egress_port_sink[i].tready) begin
                transf_pkts[i] <= transf_pkts[i] + 1;
            end else begin
                transf_pkts[i] <= transf_pkts[i];
            end

            // accepted ingress frames
            if (ingress_port_source[i].tvalid & ingress_port_source[i].tlast & ingress_po
                in_frames[i] <= in_frames[i] + 1;
            end else begin
                in_frames[i] <= in_frames[i];
            end

            // transferred ingress frames
            if (egress_port_source[i].tvalid & egress_port_source[i].tlast & egress_port_s
                transf_frames[i] <= transf_frames[i] + 1;
            end else begin
                transf_frames[i] <= transf_frames[i];
            end

            // invalid frames
            if (invalid_frame[i]) begin
                inv_frames[i] <= inv_frames[i] + 1;
            end else begin
                inv_frames[i] <= inv_frames[i];
            end

            // dropped requests
            if (timeout_req[i]) begin
```

```
                    drop_frames[i] <= drop_frames[i] + 1;
                end else begin
                    drop_frames[i] <= drop_frames[i];
                end
            end
        end
    end
    endgenerate

        // interrupt interface
        assign irq = 1'b0;

endmodule

`endif
```

Listing 13: packet_filter/preliminary_processor.sv

```
`ifdef VERILATOR
`include "packet_filter.svh"
`else
`include "../include/packet_filter.svh"
`include "../include/synth_defs.svh"
`endif
`include "filter_defs.svh"

`ifdef INTG_TESTING_1

// Integrate input_fsm, type_field_checker, and dest_calculator
`timescale 1 ps / 1 ps
module preliminary_processor #(
    parameter STUBBING = `STUBBING_PASSTHROUGH
) (
    input  logic clk,
    input  logic reset,

    input  axis_source_t ingress_source,
    output axis_sink_t   ingress_sink,
    output axis_source_t ingress_pkt,

    input  logic drop_write,
    input  logic almost_full,
    output frame_status status,
    output dest_source_t frame_dest,
    output drop_source_t frame_type

);
```

```
    frame_status f_status;
    packet_source_t dst_mac_source;
    packet_source_t type_source;

    assign status = f_status;
    assign dst_mac_source.tvalid = ingress_pkt.tvalid & f_status.scan_dst_mac;
    assign dst_mac_source.tdata = ingress_pkt.tdata;
    assign type_source.tvalid = ingress_pkt.tvalid & f_status.scan_type;
    assign type_source.tdata = ingress_pkt.tdata;

    // input FSM
    input_fsm #(
        .STUBBING(STUBBING)
    ) u_input_fsm (
        .clk(clk),
        .reset(reset),
        .ingress_source(ingress_source),
        .ingress_sink(ingress_sink),
        .drop_current(drop_write),
        .almost_full(almost_full),
        .ingress_pkt(ingress_pkt),
/* verilator lint_off PINCONNECTEMPTY */
        .incomplete_frame(), // TODO evaluate if need this
/* verilator lint_on PINCONNECTEMPTY */
        .status(f_status)
    );

    // destination calculator
    dest_calculator #(
        .STUBBING(STUBBING)
    ) u_dest_calculator (
        .clk(clk),
        .reset(reset),
        .dst_mac_pkt(dst_mac_source),
        .dest(frame_dest)
    );

    // type checker
    type_field_checker #(
        .STUBBING(STUBBING)
    ) u_type_checker (
        .clk(clk),
        .reset(reset),
        .type_pkt(type_source),
        .drop(frame_type)
    );

`ifdef ASSERT
```

```
    // assert grant is held for an entire frame
    assertion_preliminary_processor_frame_grant : assert property(
        @(posedge clk) disable iff (reset)
        ingress_source.tvalid & ingress_sink.tready & ~ingress_pkt.tlast
            |=> ingress_sink.tready || ingress_source.tlast
    ) else $error($sformatf("assertion_preliminary_processor_frame_grant failed at %0t", $real

    // do not provide grants if almost full
    assertion_preliminary_processor_almost_full : assert property(
        @(posedge clk) disable iff (reset)
        almost_full && (~$past(ingress_source.tvalid, 1) || $past(ingress_pkt.tlast, 1))
            |=> ~ingress_sink.tready
    ) else $error($sformatf("assertion_preliminary_processor_almost_full failed at %0t", $real


`endif

endmodule

`endif
```

Listing 14: packet_filter/request_buffer.sv

```
`ifdef VERILATOR
`include "packet_filter.svh"
`else
`include "../include/packet_filter.svh"
`include "../include/synth_defs.svh"
`endif
`include "filter_defs.svh"

`ifdef INTG_TESTING_1

// Integrate sideband_buffer, frame_buffer, and switch_requester
`timescale 1 ps / 1 ps
module request_buffer #(
    parameter STUBBING = `STUBBING_PASSTHROUGH,
    parameter ALMOST_FULL_THRESHOLD = 10,
    parameter ADDR_WIDTH = 11,
    parameter NUM_CYCLONE_5CSEMA5_BLOCKS = 4,
    parameter TIMEOUT_CTR_WIDTH = 3
) (
    input  logic clk,
    input  logic reset,

    input  frame_status status,
    input  drop_source_t frame_type,
```

```
    input  dest_source_t frame_dest,
    input  axis_source_t ingress_pkt,

    output logic drop_write,
    output logic almost_full,
    output logic timeout,

    output axis_d_source_t egress_source,
        input  axis_d_sink_t   egress_sink

);

    logic [ADDR_WIDTH:0] frame_wptr;

    logic sideband_ren;
    logic sideband_empty;
    logic [19:0] sideband_rdata;
    logic sideband_full;

    logic                frame_ren;
    logic                frame_rrst;
    logic [ADDR_WIDTH:0] frame_rptr;
    logic [ADDR_WIDTH:0] frame_rst_rptr;
    logic         [19:0] frame_rdata;
    logic                frame_last_entry;
    logic                frame_almost_full;

    // output assignments
    assign almost_full = sideband_full | frame_almost_full;

    // sideband buffer
    sideband_buffer #(
        .STUBBING(STUBBING),
        .ADDR_WIDTH(ADDR_WIDTH)
    ) u_sideband (
        .clk(clk),
        .reset(reset),
        .scan_frame(status.scan_frame),
        .scan_payload(status.scan_payload),
        .frame_type(frame_type),
        .frame_drop(drop_write),
        .frame_dest(frame_dest),
        .frame_wptr(frame_wptr),
        .ren(sideband_ren),
        .empty(sideband_empty),
        .rdata(sideband_rdata),
        .full(sideband_full)
    );
```

```
    // frame buffer
    frame_buffer #(
        .STUBBING(STUBBING),
        .ALMOST_FULL_THRESHOLD(ALMOST_FULL_THRESHOLD),
        .ADDR_WIDTH(ADDR_WIDTH),
        .NUM_CYCLONE_5CSEMA5_BLOCKS(NUM_CYCLONE_5CSEMA5_BLOCKS)
    ) u_frame (
        .clk(clk),
        .reset(reset),
        .ingress_pkt(ingress_pkt),
        .scan_frame(status.scan_frame),
        .drop_write(drop_write),
        .almost_full(frame_almost_full),
        .frame_wptr(frame_wptr),
        .frame_ren(frame_ren),
        .frame_rrst(frame_rrst),
        .frame_rst_rptr(frame_rst_rptr),
        .frame_rptr(frame_rptr),
        .frame_rdata(frame_rdata),
        .last_entry(frame_last_entry)
    );

    // switch requester
    switch_requester #(
        .STUBBING(STUBBING),
        .ADDR_WIDTH(ADDR_WIDTH),
        .TIMEOUT_CTR_WIDTH(TIMEOUT_CTR_WIDTH)
    ) u_requester (
        .clk(clk),
        .reset(reset),
        .scan_payload(status.scan_payload),
        .timeout(timeout),
        .sideband_rdata(sideband_rdata),
        .sideband_empty(sideband_empty),
        .sideband_ren(sideband_ren),
        .frame_rdata(frame_rdata),
        .frame_last_entry(frame_last_entry),
        .frame_rptr(frame_rptr),
        .frame_ren(frame_ren),
        .frame_rrst(frame_rrst),
        .frame_rst_rptr(frame_rst_rptr),
        .egress_sink(egress_sink),
        .egress_source(egress_source)
    );

endmodule
```

```
`endif
```

Listing 15: packet_filter/sideband_buffer.sv

```
`ifdef VERILATOR
`include "packet_filter.svh"
`else
`include "../include/packet_filter.svh"
`include "../include/synth_defs.svh"
`endif
`include "filter_defs.svh"

`timescale 1 ps / 1 ps
module sideband_buffer #(
    parameter STUBBING = `STUBBING_PASSTHROUGH,
    parameter ADDR_WIDTH = 11
) (
    input  logic clk,
    input  logic reset,

    // frame status
    input  logic scan_frame,
    input  logic scan_payload,
    input  drop_source_t frame_type,
    output logic frame_drop,

    // destination

    input  dest_source_t frame_dest,

    // frame buffer
    input  logic [ADDR_WIDTH:0] frame_wptr,

    // read interface
    input  logic        ren,
    output logic        empty,
    output logic [19:0] rdata,

    // write interface
    output logic        full

);

    localparam NUM_FIELDS = 3;
    localparam FIELD_IDX_DEST = 0;
    localparam FIELD_IDX_WPTR = 1;
    localparam FIELD_IDX_TYPE = 2;
```

```
    // write control
    logic [NUM_FIELDS-1:0] valid_fields;
    logic        wen;
    logic        written;
    logic [19:0] wdata;

    // registers to delay empty signaling
    logic prev_reset;
    logic fifo_empty;

    always_ff @(posedge clk) begin
        wdata[19:ADDR_WIDTH+`AXIS_DEST_WIDTH+1] <= '0;
        prev_reset <= reset;
        if (reset) begin
            wen <= 1'b0;
            written <= 1'b0;
            wdata <= '0;
            valid_fields <= '0;
            frame_drop <= 1'b0;
            empty <= 1'b1;
        end else begin
            empty <= fifo_empty | prev_reset;
            if (scan_frame) begin
                // save destination
                if (frame_dest.tvalid) begin
                    wdata[`AXIS_DEST_WIDTH-1:0] <= frame_dest.tdata;
                    valid_fields[FIELD_IDX_DEST] <= ~frame_dest.tuser;
                end

                // save type
                if (frame_type.tvalid) begin
                    valid_fields[FIELD_IDX_TYPE] <= ~frame_type.tuser;
                end

                // save frame pointer
                wdata[ADDR_WIDTH+`AXIS_DEST_WIDTH:`AXIS_DEST_WIDTH] <= wdata[ADDR_WIDTH+`AXIS_
                valid_fields[FIELD_IDX_WPTR] <= 1'b1;

                // write enable
                wen <= &valid_fields & ~written & ~wen;
                written <= wen | written;

                // drop detection
                frame_drop <= frame_drop
                    | (frame_dest.tvalid & frame_dest.tuser)
                    | (frame_type.tvalid & frame_type.tuser);
            end else begin
                wen <= 1'b0;
```

```
                written <= 1'b0;
                wdata[ADDR_WIDTH+`AXIS_DEST_WIDTH:`AXIS_DEST_WIDTH] <= frame_wptr;
                wdata[`AXIS_DEST_WIDTH-1:0] <= '0;
                valid_fields <= '0;
                frame_drop <= 1'b0;
            end
        end
    end

    /*
     * Sideband FIFO.
     *
     * Using 1 block
     * Can store 512 words (more than the maximum number of frames in the frame buffer)
     */
    /* verilator lint_off PINCONNECTEMPTY */
    fifo_sync #(
        .ADDR_WIDTH(9),
        .W_EL(20),
        .NUM_CYCLONE_5CSEMA5_BLOCKS(1),
        .CAN_RESET_POINTERS(0)
    ) u_sideband_fifo (
        .clk       (clk),
        .reset     (reset),
        .ren       (ren),
        .rdata     (rdata),
        .empty     (fifo_empty),
        .wdata     (wdata),
        .wen       (wen),
        .full      (full),
        .rrst      (),
        .wrst      (),
        .rst_rptr  (),
        .rst_wptr  (),
        .rptr      (),
        .wptr      ()
    );
    /* verilator lint_on PINCONNECTEMPTY */

`ifdef ASSERT
    /* Assertions. */

    // only write valid frames
    assertion_sideband_buffer_valid_frame_wen : assert property(
        @(posedge clk) disable iff (reset)
        wen |-> scan_frame & &valid_fields & ~frame_drop
    ) else $error("Failed assertion");
```

```
    // latch error status
    assertion_sideband_buffer_erroneous_dest : assert property(
        @(posedge clk) disable iff (reset)
        frame_dest.tvalid & frame_dest.tuser |=> (~valid_fields[FIELD_IDX_DEST] & frame_drop)
    ) else $error("Failed assertion");
    assertion_sideband_buffer_erroneous_type : assert property(
        @(posedge clk) disable iff (reset)
        frame_type.tvalid & frame_type.tuser |=> (~valid_fields[FIELD_IDX_TYPE] & frame_drop)
    ) else $error("Failed assertion");

    // only indicate drop when scanning a frame
    assertion_sideband_buffer_drop_scanned_frame : assert property(
        @(posedge clk) disable iff (reset)
        frame_drop |-> scan_frame || $past(scan_frame, 1)
    ) else $error("Failed assertion");
`endif

endmodule
```

Listing 16: packet_filter/switch_requester.sv

```
`ifdef VERILATOR
`include "packet_filter.svh"
`else
`include "../include/packet_filter.svh"
`include "../include/synth_defs.svh"
`endif
`include "filter_defs.svh"

`timescale 1 ps / 1 ps
module switch_requester #(
    parameter STUBBING = `STUBBING_PASSTHROUGH,
    parameter ADDR_WIDTH = 11,
    parameter TIMEOUT_CTR_WIDTH = 3
) (
    // clock and reset
    input  logic clk,
    input  logic reset,

    // frame status
    input  logic scan_payload,
    output logic timeout,

    // sideband buffer
    input  logic [19:0] sideband_rdata,
    input  logic sideband_empty,
    output logic sideband_ren,
```

```
    // frame buffer
    input  logic [19:0]         frame_rdata,
    input  logic                frame_last_entry,
    input  logic [ADDR_WIDTH:0] frame_rptr,
    output logic                frame_ren,
    output logic                frame_rrst,
    output logic [ADDR_WIDTH:0] frame_rst_rptr,

    // egress interface
    input  axis_d_sink_t   egress_sink,
    output axis_d_source_t egress_source

);

    // State definitions.
    localparam IDLE           = 3'b000; // waiting for sideband buffer to show an entry
    localparam READ_SIDEBAND  = 3'b010; // read sideband information
    localparam INIT_FRAME_PTR = 3'b011; // set pointers in frame buffer
    localparam INIT_REQ       = 3'b001; // make request to switch
    localparam WRITE_FRAME    = 3'b101; // writing frame

    // State signals
    logic [2:0]                 state, next_state;
    logic [TIMEOUT_CTR_WIDTH:0] timeout_ctr;
    logic                       next_rptr_is_last;
    logic                       first_req_next;
    logic                       first_req;

    // frame buffer control
    logic [ADDR_WIDTH:0] next_frame_rptr;
    logic                first_frame_rrst;
    logic                prev_frame_rrst;

    // sideband buffer control
    logic first_sideband_ren;

    // egress control
    logic first_last;

    // egress interface
    logic tvalid;
    logic tlast;
    logic [1:0] tdest;
    logic [15:0] tdata;
    logic tready;

    /* Save input data. */
```

```
    // latch sideband data
    always_ff @(posedge clk) begin
        if (reset) begin
            frame_rst_rptr <= '0;
            tdest <= '0;
        end else begin
            if (first_sideband_ren) begin
                frame_rst_rptr <= next_frame_rptr;
                tdest <= sideband_rdata[`AXIS_DEST_WIDTH-1:0];
            end else begin
                frame_rst_rptr <= frame_rst_rptr;
                tdest <= egress_source.tdest;
            end
        end
    end

    /* Propagate next state. */
    always_ff @(posedge clk) begin
        if (reset) begin
            state <= IDLE;
            timeout_ctr <= '0;
            first_req <= 1'b0;
        end else begin
            state <= next_state;
            if (state === IDLE || egress_sink.tready) begin
                // reset counter with no request or granted request
                timeout_ctr <= '0;
            end else if (egress_source.tvalid & ~egress_sink.tready) begin
                // increment counter when making a request that is not granted
                timeout_ctr <= timeout_ctr + 1;
            end else begin
                // persist count if pausing a request
                timeout_ctr <= timeout_ctr;
            end
            first_req <= first_req_next;
        end
    end

    /* Generate next state. */

    // find the boundary for the next frame
    assign next_frame_rptr = sideband_rdata[ADDR_WIDTH+`AXIS_DEST_WIDTH:`AXIS_DEST_WIDTH];
    assign next_rptr_is_last = ((frame_rptr + 1) === next_frame_rptr) ? 1'b1 : 1'b0;

    // next state (and state transition indicator) logic
    always_comb begin
        next_state = state;
        first_sideband_ren = 1'b0;
```

```
first_frame_rrst = 1'b0;
first_req_next = 1'b0;
first_last = 1'b0;
case (state)
IDLE: begin
    // start making requests when frames exist in the sideband
    if (~sideband_empty) begin
        next_state = READ_SIDEBAND;
        first_sideband_ren = 1'b1;
    end
end
READ_SIDEBAND: begin
    // allow cycle delay to read sideband buffer
    next_state = INIT_FRAME_PTR;
    first_frame_rrst = 1'b1;
end
INIT_FRAME_PTR: begin
    // set pointers in frame FIFO
    // can start request when have begun receiving payload or there are more frames ir
    if (~frame_rrst & (scan_payload | ~sideband_empty)) begin
        next_state = INIT_REQ;
        first_req_next = 1'b1;
    end
end
INIT_REQ: begin
    if (timeout_ctr[TIMEOUT_CTR_WIDTH]) begin
        // timeout request when counter overflows
        next_state = IDLE;
    end else if (egress_sink.tready) begin
        // start with the granted request
        next_state = WRITE_FRAME;
    end
end
WRITE_FRAME: begin
    // assert last when one more entry in the current frame
    if ((sideband_empty & frame_last_entry) | next_rptr_is_last) begin
        //next_state = LAST_PACKET;
        next_state = IDLE;
        first_last = 1'b1;
    end else if (timeout_ctr[TIMEOUT_CTR_WIDTH]) begin
        // timeout request when counter overflows
        next_state = IDLE;
    end
end
default: begin
    next_state = IDLE;
end
endcase
```

```
    end

    /* Write output data. */

    // control frame read enable
    always_comb begin
        frame_ren = ~reset & (tvalid & tready & ~first_last);
    end

    // control frame cursor reset
    always_ff @(posedge clk) begin
        if (reset) begin
            frame_rrst <= 1'b0;
            prev_frame_rrst <= 1'b0;
        end else begin
            prev_frame_rrst <= frame_rrst;

            // pulse frame read reset when enter the frame pointer state
            if (frame_rrst === 1'b1) begin
                frame_rrst <= 1'b0;
            end else if (first_frame_rrst) begin
                frame_rrst <= 1'b1;
            end else begin
                frame_rrst <= 1'b0;
            end
        end
    end

    // output generation
    assign egress_source.tvalid = (state === INIT_REQ
            || state == WRITE_FRAME
        ) ? 1'b1 : 1'b0;
    assign egress_source.tdata = frame_rdata[15:0];
    assign egress_source.tdest = tdest;
    assign egress_source.tlast = first_last | (tlast & ~tready);
    assign timeout = timeout_ctr[TIMEOUT_CTR_WIDTH];
    always_ff @(posedge clk) begin
        if (reset) begin
            sideband_ren <= 1'b0;
            tdata <= '0;
            tlast <= 1'b0;
            tready <= 1'b0;
        end else begin
            sideband_ren <= first_sideband_ren;

            tlast <= first_last;

            // valid when read from FIFO or previous request not granted
```

```
            tvalid <= (first_req || frame_ren || (~tready & egress_source.tvalid)) && ~timeou

            tready <= egress_sink.tready;
        end
    end

`ifdef ASSERT
    /* Assertions. */

    // assert data does not get lost (tdata does not change while tvalid is high if tready wa
    assertion_switch_requester_stable_tdata : assert property(
        @(posedge clk) disable iff (reset)
        egress_source.tvalid & ~egress_sink.tready |=> $stable(egress_source.tdata)
    ) else $error("Failed assertion");

    // assert tdest is stable while tvalid is high
    assertion_switch_requester_stable_tdest : assert property(
        @(posedge clk) disable iff (reset)
        egress_source.tvalid & egress_sink.tready & ~egress_source.tlast |=> $stable(egress_so
    ) else $error("Failed assertion");

    // assert tvalid stays high until timeout or last
    assertion_switch_requester_stable_tvalid : assert property(
        @(posedge clk) disable iff (reset)
        egress_source.tvalid & ~egress_source.tlast |=> egress_source.tvalid || timeout_ctr[TI
    ) else $error("Failed assertion");

    // assert do not read another frame when the last packet is being read
    assertion_switch_requester_last_packet_ren : assert property(
        @(posedge clk) disable iff (reset)
        egress_source.tlast |=> ~frame_ren
    ) else $error("Failed assertion");

    // assert do not read from FIFO until receive payload of the first frame
    assertion_switch_requester_premature_read : assert property(
        @(posedge clk) disable iff (reset)
        ~scan_payload && sideband_empty && ~$past(frame_ren, 1) |-> ~frame_ren
    ) else $error("Failed assertion");

    // assert do not read from an empty FIFO (will fail if empty transitions high in middle o
    //assertion_switch_requester_empty_read : assert property(
    //    @(posedge clk) disable iff (reset)
    //    sideband_empty |-> ~sideband_ren
    //) else $error("Failed assertion");

    // assert state transition signals are pulsed
    assertion_switch_requester_pulse_first_sideband_ren : assert property(
        @(posedge clk) disable iff (reset)
```

```
        first_sideband_ren |=> ~first_sideband_ren
    ) else $error("Failed assertion");
    assertion_switch_requester_pulse_first_frame_rrst : assert property(
        @(posedge clk) disable iff (reset)
        first_frame_rrst |=> ~first_frame_rrst
    ) else $error("Failed assertion");
    assertion_switch_requester_pulse_first_req : assert property(
        @(posedge clk) disable iff (reset)
        first_req |=> ~first_req
    ) else $error("Failed assertion");
    assertion_switch_requester_pulse_first_last : assert property(
        @(posedge clk) disable iff (reset)
        first_last |=> ~first_last
    ) else $error("Failed assertion");
`endif

endmodule
```

Listing 17: packet_filter/type_field_checker.sv

```
`ifdef VERILATOR
`include "packet_filter.svh"
`else
`include "../include/packet_filter.svh"
`include "../include/synth_defs.svh"
`endif
`include "filter_defs.svh"

`timescale 1 ps / 1 ps
module type_field_checker #(
    parameter STUBBING = `STUBBING_PASSTHROUGH
) (
    input  logic clk,
    input  logic reset,

    input  packet_source_t type_pkt,

    output drop_source_t drop
);

    packet_source_t type_pkt_q;
    logic valid_output, invalid_type;

    // validate type field
    always_ff @(posedge clk) begin
        if (reset) begin
            valid_output <= 1'b0;
            invalid_type <= 1'b0;
```

```
        end else begin
            valid_output <= type_pkt.tvalid;
            if (!(type_pkt.tdata < 16'h05DC || type_pkt.tdata > 16'h0600)) begin
                invalid_type <= 1'b1;
            end else begin
                invalid_type <= 1'b0;
            end
        end
    end

    // assign output
    assign drop.tvalid = valid_output;
    assign drop.tuser  = invalid_type;

endmodule
```

8.5 Switch Fabric

Listing 18: packet-switch/mux4to1.sv

```
module mux4to1 #(
    parameter N_PORTS = 4,
    parameter DATA_WIDTH = 16,
    parameter IDX_WIDTH = $clog2(N_PORTS)
)(
    input  logic [N_PORTS-1:0][DATA_WIDTH-1:0] ingress_data,
    input  logic [IDX_WIDTH-1:0]               selected_ingress,
    output logic [DATA_WIDTH-1:0]              egress_data
);

    always_comb begin
        egress_data = ingress_data[selected_ingress];
    end

endmodule
```

Listing 19: packet-switch/rr_scheduler.sv

```
`timescale 1ns/1ps


module rr_scheduler #(
    parameter N_PORTS = 4,
    parameter IDX_WIDTH = 2
)(
    input logic                  clk,
    input logic                  reset,
    input logic [N_PORTS-1:0]    ingress_valid,
```

```
    input logic [N_PORTS-1:0]     ingress_last,
    input logic [IDX_WIDTH-1:0]   ingress_dst [N_PORTS-1:0],
    input logic [IDX_WIDTH-1:0]   egress_port_id,
    input logic                   egress_ready,
    output logic [IDX_WIDTH-1:0] selected_ingress,
    output logic                  egress_valid,
    output logic                  egress_last,
    output logic [N_PORTS-1:0]    ingress_ready
);

    typedef enum logic [0:0] { IDLE, SEND } state_t;
    state_t state, next_state;

    logic [IDX_WIDTH-1:0] next_rr, next_rr_next;
    logic [IDX_WIDTH-1:0] select, select_next;
    logic [N_PORTS-1:0]   grant;

    logic found;

    always_ff @(posedge clk or posedge reset) begin
        if (reset) begin
            state    <= IDLE;
            next_rr  <= '0;
            select   <= '0;
            grant    <= '0;
        end else begin
            state    <= next_state;
            next_rr  <= next_rr_next;
            select   <= select_next;

        if (state == IDLE && next_state == SEND) begin
            grant <= '0;
            grant[select_next] <= 1'b1;
        end
        else if (state == SEND) begin
            if (egress_valid && egress_last && egress_ready) begin
                grant <= '0;
            end
            else if (!ingress_valid[select]) begin
                grant <= '0;
            end
        end
        else begin
            grant <= grant;
        end

        end
    end
```

```
logic [IDX_WIDTH-1:0] idx [N_PORTS-1:0];
always_comb begin
    for (int i = 0; i < N_PORTS; i++) begin
        idx[i] = next_rr + i[IDX_WIDTH-1:0];
    end
end

always_comb begin
    next_state    = state;
    next_rr_next  = next_rr;
    select_next   = select;
    egress_valid  = 1'b0;
    egress_last   = 1'b0;
    ingress_ready = '0;
    found = 1'b0;

    case (state)
        IDLE: begin
            for (int i = 0; i < N_PORTS; i++) begin
                if (!found && ingress_valid[idx[i]] && ingress_dst[idx[i]] == egress_port_
                    found = 1'b1;
                    select_next = idx[i];
                    next_state  = SEND;
                    egress_valid= 1'b1;
                end
            end
        end

        SEND: begin
            egress_valid = ingress_valid[select];
            egress_last  = ingress_last[select];
           ingress_ready = grant & {N_PORTS{egress_ready}};

            if (ingress_valid[select] && ingress_last[select]) begin
                next_rr_next = select + 1'b1;
                next_state   = IDLE;
            end
             else if (!ingress_valid[select]) begin
                next_state = IDLE;
            end
        end

        default: next_state = IDLE;
    endcase
    selected_ingress = select;
end
```

```
endmodule
```

Listing 20: packet-switch/packet_switch.sv

```
// packet_switch.sv
`include "../include/packet_filter.svh"
//`include "rr_scheduler.sv"
//`include "mux4to1.sv"
`timescale 1 ps / 1 ps

module packet_switch #(
    parameter N_PORTS    = 4,
    parameter DATA_WIDTH = 16,
    parameter IDX_WIDTH  = 2,
    parameter STUBBING   = `STUBBING_PASSTHROUGH
)(
    input  wire                  clk,
    input  wire                  reset,

    // Avalon-MM register interface
    input  wire [31:0]           writedata,
    input  wire                  write,
    input  wire                  chipselect,
    input  wire [7:0]            address,
    input  wire                  read,
    output reg  [31:0]           readdata,

    // Four AXIS ingress ports
    input  wire [DATA_WIDTH-1:0] ingress_port_0_tdata,
    input  wire                  ingress_port_0_tvalid,
    input  wire                  ingress_port_0_tlast,
    input  wire [IDX_WIDTH-1:0]  ingress_port_0_tdest,
    output wire                  ingress_port_0_tready,

    input  wire [DATA_WIDTH-1:0] ingress_port_1_tdata,
    input  wire                  ingress_port_1_tvalid,
    input  wire                  ingress_port_1_tlast,
    input  wire [IDX_WIDTH-1:0]  ingress_port_1_tdest,
    output wire                  ingress_port_1_tready,

    input  wire [DATA_WIDTH-1:0] ingress_port_2_tdata,
    input  wire                  ingress_port_2_tvalid,
    input  wire                  ingress_port_2_tlast,
    input  wire [IDX_WIDTH-1:0]  ingress_port_2_tdest,
    output wire                  ingress_port_2_tready,

    input  wire [DATA_WIDTH-1:0] ingress_port_3_tdata,
```

```
    input  wire                   ingress_port_3_tvalid,
    input  wire                   ingress_port_3_tlast,
    input  wire [IDX_WIDTH-1:0]   ingress_port_3_tdest,
    output wire                   ingress_port_3_tready,

    output wire [DATA_WIDTH-1:0] egress_port_0_tdata,
    output wire                   egress_port_0_tvalid,
    output wire                   egress_port_0_tlast,
    input  wire                   egress_port_0_tready,

    output wire [DATA_WIDTH-1:0] egress_port_1_tdata,
    output wire                   egress_port_1_tvalid,
    output wire                   egress_port_1_tlast,
    input  wire                   egress_port_1_tready,

    output wire [DATA_WIDTH-1:0] egress_port_2_tdata,
    output wire                   egress_port_2_tvalid,
    output wire                   egress_port_2_tlast,
    input  wire                   egress_port_2_tready,

    output wire [DATA_WIDTH-1:0] egress_port_3_tdata,
    output wire                   egress_port_3_tvalid,
    output wire                   egress_port_3_tlast,
    input  wire                   egress_port_3_tready,

    output wire                   irq
);

    logic [N_PORTS-1:0] egress_mask;
    always_ff @(posedge clk or posedge reset) begin
        if (reset)                          egress_mask <= 4'b1111;
        else if (chipselect && write && address == 8'h0)
                                            egress_mask <= writedata[N_PORTS-1:0];
    end
    always_ff @(posedge clk or posedge reset) begin
        if (reset)                          readdata <= 32'h00;
        else if (chipselect && read && address == 8'h0)
                                            readdata <= {{32-N_PORTS{1'b0}}, egress_mask};
    end
    assign irq = 1'b0;

generate
if (STUBBING == `STUBBING_PASSTHROUGH) begin: g_passthrough

    assign ingress_port_0_tready = egress_port_0_tready;
    assign ingress_port_1_tready = egress_port_1_tready;
    assign ingress_port_2_tready = egress_port_2_tready;
    assign ingress_port_3_tready = egress_port_3_tready;
```

```
    assign egress_port_0_tvalid = ingress_port_0_tvalid;
    assign egress_port_0_tdata  = ingress_port_0_tdata;
    assign egress_port_0_tlast  = ingress_port_0_tlast;
    assign egress_port_1_tvalid = ingress_port_1_tvalid;
    assign egress_port_1_tdata  = ingress_port_1_tdata;
    assign egress_port_1_tlast  = ingress_port_1_tlast;
    assign egress_port_2_tvalid = ingress_port_2_tvalid;
    assign egress_port_2_tdata  = ingress_port_2_tdata;
    assign egress_port_2_tlast  = ingress_port_2_tlast;
    assign egress_port_3_tvalid = ingress_port_3_tvalid;
    assign egress_port_3_tdata  = ingress_port_3_tdata;
    assign egress_port_3_tlast  = ingress_port_3_tlast;

end else begin: g_functional

    axis_d_source_t ingress_src   [N_PORTS-1:0];
    axis_d_sink_t   ingress_sink  [N_PORTS-1:0];
    axis_source_t   egress_src    [N_PORTS-1:0];
    axis_sink_t     egress_sink   [N_PORTS-1:0];

    logic [N_PORTS-1:0]           bus_ing_valid, bus_ing_last;
    logic [IDX_WIDTH-1:0]         bus_ing_dest  [N_PORTS-1:0];
    logic [IDX_WIDTH-1:0]         sched_sel     [N_PORTS-1:0];
    logic                         sched_val     [N_PORTS-1:0];
    logic                         sched_last    [N_PORTS-1:0];
    logic [N_PORTS-1:0]           sched_ing_rdy [N_PORTS-1:0];

    assign ingress_src[0].tdata  = ingress_port_0_tdata;
    assign ingress_src[0].tvalid = ingress_port_0_tvalid;
    assign ingress_src[0].tdest  = ingress_port_0_tdest;
    assign ingress_src[0].tlast  = ingress_port_0_tlast;

    assign ingress_src[1].tdata  = ingress_port_1_tdata;
    assign ingress_src[1].tvalid = ingress_port_1_tvalid;
    assign ingress_src[1].tdest  = ingress_port_1_tdest;
    assign ingress_src[1].tlast  = ingress_port_1_tlast;

    assign ingress_src[2].tdata  = ingress_port_2_tdata;
    assign ingress_src[2].tvalid = ingress_port_2_tvalid;
    assign ingress_src[2].tdest  = ingress_port_2_tdest;
    assign ingress_src[2].tlast  = ingress_port_2_tlast;

    assign ingress_src[3].tdata  = ingress_port_3_tdata;
    assign ingress_src[3].tvalid = ingress_port_3_tvalid;
    assign ingress_src[3].tdest  = ingress_port_3_tdest;
    assign ingress_src[3].tlast  = ingress_port_3_tlast;
```

```
genvar p;
for (p = 0; p < N_PORTS; p++) begin : R_IN_SINK
    assign ingress_sink[p].tready =
           |{ sched_ing_rdy[0][p],
              sched_ing_rdy[1][p],
              sched_ing_rdy[2][p],
              sched_ing_rdy[3][p] };
end

assign ingress_port_0_tready = ingress_sink[0].tready;
assign ingress_port_1_tready = ingress_sink[1].tready;
assign ingress_port_2_tready = ingress_sink[2].tready;
assign ingress_port_3_tready = ingress_sink[3].tready;

always_comb begin
    for (integer j = 0; j < N_PORTS; j++) begin
        bus_ing_valid[j] = ingress_src[j].tvalid;
        bus_ing_last [j] = ingress_src[j].tlast;
        bus_ing_dest [j] = ingress_src[j].tdest;
    end
end

assign egress_sink[0].tready = egress_port_0_tready;
assign egress_sink[1].tready = egress_port_1_tready;
assign egress_sink[2].tready = egress_port_2_tready;
assign egress_sink[3].tready = egress_port_3_tready;

for (p = 0; p < N_PORTS; p++) begin : SCHED_AND_MUX
    rr_scheduler #(
        .N_PORTS   (N_PORTS),
        .IDX_WIDTH (IDX_WIDTH)
    ) rr_sch (
        .clk             (clk),
        .reset           (reset),
        .ingress_valid   (bus_ing_valid),
        .ingress_last    (bus_ing_last),
        .ingress_dst     (bus_ing_dest),
        .egress_port_id  (p[IDX_WIDTH-1:0]),
        .egress_ready    (egress_sink[p].tready),
        .selected_ingress(sched_sel[p]),
        .egress_valid    (sched_val[p]),
        .egress_last     (sched_last[p]),
        .ingress_ready   (sched_ing_rdy[p])
    );
    mux4to1 #(
        .N_PORTS     (N_PORTS),
        .DATA_WIDTH (DATA_WIDTH),
        .IDX_WIDTH   (IDX_WIDTH)
```

```
        ) mr (
            .ingress_data     ({
                ingress_src[3].tdata,
                ingress_src[2].tdata,
                ingress_src[1].tdata,
                ingress_src[0].tdata
            }),
            .selected_ingress (sched_sel[p]),
            .egress_data      (egress_src[p].tdata)
        );
        assign egress_src[p].tvalid = sched_val[p] & egress_mask[p];
        assign egress_src[p].tlast  = sched_last[p];
    end
    assign egress_port_0_tvalid = egress_src[0].tvalid;
    assign egress_port_0_tdata  = egress_src[0].tdata;
    assign egress_port_0_tlast  = egress_src[0].tlast;

    assign egress_port_1_tvalid = egress_src[1].tvalid;
    assign egress_port_1_tdata  = egress_src[1].tdata;
    assign egress_port_1_tlast  = egress_src[1].tlast;

    assign egress_port_2_tvalid = egress_src[2].tvalid;
    assign egress_port_2_tdata  = egress_src[2].tdata;
    assign egress_port_2_tlast  = egress_src[2].tlast;

    assign egress_port_3_tvalid = egress_src[3].tvalid;
    assign egress_port_3_tdata  = egress_src[3].tdata;
    assign egress_port_3_tlast  = egress_src[3].tlast;

  end
 endgenerate

endmodule
```

9 SOFTWARE CODE

Listing 21: Frame Generator c

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
```

```
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "frame_generator_0.h"

#define DRIVER_NAME "frame_generator_0"
#define MAX_PAYLOAD_SIZE 100
//#define NUM_GENERATORS 4
/* Device registers */
#define dst_0(x) ((x)+0)
#define dst_1(x) ((x)+1)
#define dst_2(x) ((x)+2)
#define dst_3(x) ((x)+3)
#define dst_4(x) ((x)+4)
#define dst_5(x) ((x)+5)
#define src_0(x) ((x)+6)
#define src_1(x) ((x)+7)
#define src_2(x) ((x)+8)
#define src_3(x) ((x)+9)
#define src_4(x) ((x)+10)
#define src_5(x) ((x)+11)
#define length_0(x) ((x)+12)
#define length_1(x) ((x)+13)
#define type_0(x) ((x)+14)
#define type_1(x) ((x)+15)
#define inter_frame_wait(x) ((x)+16)
#define payload(x) ((x)+17)
#define checksum_0(x) ((x)+17)
#define checksum_1(x) ((x)+18)
#define checksum_2(x) ((x)+19)
#define checksum_3(x) ((x)+20)

struct frame_generator_dev {
    struct resource res;
    void __iomem *virtbase;
    packet_data_info_t data;
    packet_payload_t payload;
    frame_generator_read_t readdata;
} dev;

//static frame_generator_dev devs[NUM_GENERATORS];

static void writePacketInfo(packet_data_info_t *writedata) {
    iowrite8(writedata->dst_0, dst_0(dev.virtbase));
    iowrite8(writedata->dst_1, dst_1(dev.virtbase));
    iowrite8(writedata->dst_2, dst_2(dev.virtbase));
    iowrite8(writedata->dst_3, dst_3(dev.virtbase));
    iowrite8(writedata->dst_4, dst_4(dev.virtbase));
```

```
    iowrite8(writedata->dst_5, dst_5(dev.virtbase));

    iowrite8(writedata->src_0, src_0(dev.virtbase));
    iowrite8(writedata->src_1, src_1(dev.virtbase));
    iowrite8(writedata->src_2, src_2(dev.virtbase));
    iowrite8(writedata->src_3, src_3(dev.virtbase));
    iowrite8(writedata->src_4, src_4(dev.virtbase));
    iowrite8(writedata->src_5, src_5(dev.virtbase));

    iowrite8(writedata->length_0, length_0(dev.virtbase));
    iowrite8(writedata->length_1, length_1(dev.virtbase));

    iowrite8(writedata->type_0, type_0(dev.virtbase));
    iowrite8(writedata->type_1, type_1(dev.virtbase));

    iowrite8(writedata->frame_wait, inter_frame_wait(dev.virtbase));

    dev.data = *writedata;
}

static void writePayload(packet_payload_t *payload) {
    void __iomem *addr;
    uint16_t length = ((uint16_t)dev.data.length_1 << 8) | dev.data.length_0;
    int i = 0;
    for(; i < length; i++) {
        addr = payload(dev.virtbase) + i;
        iowrite8(payload->data[i], addr);
    }
    dev.payload = *payload;
}

static long frame_generator_ioctl(struct file *f, unsigned int cmd, unsigned long arg) {
    frame_generator_arg_t vla;
    pr_info("ioctl %d\n", cmd);

    switch(cmd){
        case FRAME_WRITE_PACKET_0:
            if(copy_from_user(&vla, (frame_generator_arg_t *)arg, sizeof(frame_generator_arg_t
                return -EACCES;
            writePacketInfo(&vla.writedata);
            writePayload(&vla.payload);
            break;
        case FRAME_READ_CHECKSUM_0:
            vla.readdata.checksum_0 = ioread8(checksum_0(dev.virtbase));
            vla.readdata.checksum_1 = ioread8(checksum_1(dev.virtbase));
            vla.readdata.checksum_2 = ioread8(checksum_2(dev.virtbase));
            vla.readdata.checksum_3 = ioread8(checksum_3(dev.virtbase));
            if(copy_to_user((frame_generator_arg_t *)arg, &vla, sizeof(frame_generator_arg_t))
```

```
                return -EACCES;
            break;
        default:
            return -EINVAL;
    }
    return 0;
}

static const struct file_operations frame_generator_fops = {
        .owner          = THIS_MODULE,
        .unlocked_ioctl = frame_generator_ioctl,
};

static struct miscdevice frame_generator_misc_device = {
        .minor          = MISC_DYNAMIC_MINOR,
        .name           = DRIVER_NAME,
        .fops           = &frame_generator_fops,
};

static int __init frame_generator_probe(struct platform_device *pdev)
{
    packet_data_info_t writedata = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
    packet_payload_t payload = {0};
        int ret;

        /* Register ourselves as a misc device: creates /dev/frame_generator_0 */
        ret = misc_register(&frame_generator_misc_device);

        /* Get the address of our registers from the device tree */
        ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
        if (ret) {
                ret = -ENOENT;
                goto out_deregister;
        }

        /* Make sure we can use these registers */
        if (request_mem_region(dev.res.start, resource_size(&dev.res),
                               DRIVER_NAME) == NULL) {
                ret = -EBUSY;
                goto out_deregister;
        }

        /* Arrange access to our registers */
        dev.virtbase = of_iomap(pdev->dev.of_node, 0);
        if (dev.virtbase == NULL) {
                ret = -ENOMEM;
                goto out_release_mem_region;
        }
```

```
        /* Set initial values */
        writePacketInfo(&writedata);
    writePayload(&payload);

        return 0;

out_release_mem_region:
        release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
        misc_deregister(&frame_generator_misc_device);
        return ret;
}

/* Clean-up code: release resources */
static int frame_generator_remove(struct platform_device *pdev)
{
        iounmap(dev.virtbase);
        release_mem_region(dev.res.start, resource_size(&dev.res));
        misc_deregister(&frame_generator_misc_device);
        return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
static const struct of_device_id frame_generator_of_match[] = {
        { .compatible = "csee4840,frame_generator-1.0" },
        {},
};
MODULE_DEVICE_TABLE(of, frame_generator_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver frame_generator_driver = {
        .driver = {
                .name   = DRIVER_NAME,
                .owner  = THIS_MODULE,
                .of_match_table = of_match_ptr(frame_generator_of_match),
        },
        .remove = __exit_p(frame_generator_remove),
};

/* Called when the module is loaded: set things up */
static int __init frame_generator_init(void)
{
        pr_info(DRIVER_NAME ": init\n");
        return platform_driver_probe(&frame_generator_driver, frame_generator_probe);
}
```

```
/* Calball when the module is unloaded: release resources */
static void __exit frame_generator_exit(void)
{
        platform_driver_unregister(&frame_generator_driver);
        pr_info(DRIVER_NAME ": exit\n");
}

module_init(frame_generator_init);
module_exit(frame_generator_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("frameGenerator group");
MODULE_DESCRIPTION("frame generator 0 driver");
```

Listing 22: Frame Generator h

```
#ifndef _FRAME_GENERATOR_0_H
#define _FRAME_GENERATOR_0_H

#include <linux/ioctl.h>
//#include <stdint.h>
typedef unsigned char uint8_t;
typedef unsigned short uint16_t;
typedef unsigned int uint32_t;

typedef struct {
    uint8_t dst_0;
    uint8_t dst_1;
    uint8_t dst_2;
    uint8_t dst_3;
    uint8_t dst_4;
    uint8_t dst_5;
    uint8_t src_0;
    uint8_t src_1;
    uint8_t src_2;
    uint8_t src_3;
    uint8_t src_4;
    uint8_t src_5;
    uint8_t length_0;
    uint8_t length_1;
    uint8_t type_0;
    uint8_t type_1;
    uint8_t frame_wait;
} packet_data_info_t;

typedef struct {
    uint8_t data[100];
} packet_payload_t;
```

```
typedef struct {
    uint8_t checksum_0;
    uint8_t checksum_1;
    uint8_t checksum_2;
    uint8_t checksum_3;
} frame_generator_read_t;

typedef union {
    packet_data_info_t writedata;
    packet_payload_t payload;
    frame_generator_read_t readdata;
} frame_generator_arg_t;

#define FRAME_GENERATOR_MAGIC 'f'

#define FRAME_WRITE_PACKET_0  _IOW(FRAME_GENERATOR_MAGIC, 1, frame_generator_arg_t)
#define FRAME_READ_CHECKSUM_0 _IOR(FRAME_GENERATOR_MAGIC, 2, frame_generator_arg_t)

#endif
```

Listing 23: hello c

```
/*
 * Userspace program that communicates with the various device drivers
 * through ioctls
 *
 * Stephen A. Edwards
 * Columbia University
 */

#include <stdio.h>
#include "packet_filter.h"
//#include "packet_switch.h"
#include "frame_generator_0.h"
#include "frame_receptor_0.h"
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>
#include <unistd.h>

int packet_filter_fd;
int frame_generator_0_fd;
int frame_receptor_0_fd;
/* Read and print the mask */
void print_ingress_mask() {
  packet_filter_arg_t vla;
```

```
  if (ioctl(packet_filter_fd, PACKET_FILTER_READ_INGRESS_PORT_MASK, &vla)) {
      perror("ioctl(PACKET_FILTER_READ_INGRESS_PORT_MASK) failed");
      return;
  }
  printf("%0x\n",
         (unsigned int)vla.ingress_port_mask.mask);
}

/* Set the mask */
void set_ingress_mask(int mask)
{
  packet_filter_arg_t vla;
  vla.ingress_port_mask.mask = mask;
  if (ioctl(packet_filter_fd, PACKET_FILTER_WRITE_INGRESS_PORT_MASK, &vla)) {
      perror("ioctl(PACKET_FILTER_WRITE_INGRESS_PORT_MASK) failed");
      return;
  }
}

void write_packet_0(frame_generator_arg_t input){
  frame_generator_arg_t vla = input;
  if(ioctl(frame_generator_0_fd, FRAME_WRITE_PACKET_0, &vla)) {
    perror("ioctl(FRAME_WRITE_PACKET_0) failed");
    return;
  }
}

void read_checksum_0(frame_generator_arg_t input) {
  frame_generator_arg_t vla = input;
  if(ioctl(frame_generator_0_fd, FRAME_READ_CHECKSUM_0, &vla)) {
    perror("ioctl(FRAME_READ_CHECKSUM_0) failed");
    return;
  }
  uint32_t checksum = (vla.readdata.checksum_3 << 24) |
                      (vla.readdata.checksum_2 << 16) |
                      (vla.readdata.checksum_1 << 8) |
                      (vla.readdata.checksum_0);
  printf("Checksum: 0x%08x\n", checksum);
}

void write_receptor_data_0(frame_receptor_arg_t receptorDST){
  frame_receptor_arg_t vla = receptorDST;
  if(ioctl(frame_receptor_0_fd, RECEPTOR_WRITE_0, &vla)) {
    perror("ioctl(RECEPTOR_WRITE_0) failed");
    return;
  }
}
```

```
void read_receptorChecksum_0(frame_receptor_arg_t input) {
  frame_receptor_arg_t vla = input;
  if(ioctl(frame_receptor_0_fd, RECEPTOR_READ_0, &vla)) {
    perror("ioctl(RECEPTOR_READ_0) failed");
    return;
  }
  uint8_t dstCheck = (vla.readdata.dstCheck);
  printf("dstCheck: 0x%08x\n", dstCheck);
  uint32_t checksum = (vla.readdata.checksum_3 << 24) |
                      (vla.readdata.checksum_2 << 16) |
                      (vla.readdata.checksum_1 << 8) |
                      (vla.readdata.checksum_0);
  printf("Checksum: 0x%08x\n", checksum);
}

int main()
{
  packet_filter_arg_t packet_filter_vla;
  frame_generator_arg_t input;
  input.writedata.dst_0 = 0;
  input.writedata.dst_1 = 0;
  input.writedata.dst_2 = 0;
  input.writedata.dst_3 = 0;
  input.writedata.dst_4 = 0;
  input.writedata.dst_5 = 0;

  input.writedata.src_0 = 0;
  input.writedata.src_1 = 0;
  input.writedata.src_2 = 0;
  input.writedata.src_3 = 0;
  input.writedata.src_4 = 0;
  input.writedata.src_5 = 0;

  input.writedata.length_0 = 2;
  input.writedata.length_1 = 0;
  input.writedata.type_0 = 0;
  input.writedata.type_1 = 0;

  input.writedata.frame_wait = 10;

  input.payload.data[0] = 10;
  input.payload.data[1] = 10;

  frame_receptor_arg_t receptorDST;
  receptorDST.writedata.dst_0 = 0;
  receptorDST.writedata.dst_1 = 0;
  receptorDST.writedata.dst_2 = 0;
```

```
  receptorDST.writedata.dst_3 = 0;
  receptorDST.writedata.dst_4 = 0;
  receptorDST.writedata.dst_5 = 0;
  receptorDST.writedata.frame_wait = 10;

  printf("Userspace program started\n");

  if ( (packet_filter_fd = open("/dev/packet_filter", O_RDWR)) == -1) {
    fprintf(stderr, "could not open /dev/packet_filter\n");
    return -1;
  }
  printf("1\n");
  if ( (frame_generator_0_fd = open("/dev/frame_generator_0", O_RDWR)) == -1) {
    fprintf(stderr, "could not open /dev/frame_generator_0\n");
    return -1;
  }
  printf("2\n");
  if ( (frame_receptor_0_fd = open("/dev/frame_receptor_0", O_RDWR)) == -1) {
    fprintf(stderr, "could not open /dev/frame_receptor_0\n");
    return -1;
  }

  printf("initial state: ");

  print_ingress_mask();
  set_ingress_mask(0xf);
  print_ingress_mask();
  write_receptor_data_0(receptorDST);
  write_packet_0(input);
  read_checksum_0(input);
  read_receptorChecksum_0(receptorDST);
  printf("Userspace program terminating\n");
  return 0;
}
```

Listing 24: packet filter

```
/* * Device driver for the ingress packet filter
 *
 * A Platform device implemented using the misc subsystem
 *
 * Stephen A. Edwards
 * Columbia University
 *
 * References:
 * Linux source: Documentation/driver-model/platform.txt
 *               drivers/misc/arm-charlcd.c
 * http://www.linuxforu.com/tag/linux-device-drivers/
 * http://free-electrons.com/docs/
```

```
 *
 * "make" to build
 * insmod packet_filter.ko
 *
 * Check code style with
 * checkpatch.pl --file --no-tree packet_filter.c
 */

#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/slab.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "packet_filter.h"

#define DRIVER_NAME "packet_filter"

/* Device registers */
#define INGRESS_PORT_FILTER(x) ((x)+0)
#define CTR_IN_PKTS(x,i) ((x)+4+(i))
#define CTR_TRANSF_PKTS(x,i) ((x)+8+(i))
#define CTR_IN_FRAMES(x,i) ((x)+12+(i))
#define CTR_TRANSF_FRAMES(x,i) ((x)+16+(i))
#define CTR_INV_FRAMES(x,i) ((x)+20+(i))
#define CTR_DROP_FRAMES(x,i) ((x)+24+(i))

/*
 * Information about our device
 */
struct packet_filter_dev {
        struct resource res; /* Resource: our registers */
        void __iomem *virtbase; /* Where registers can be accessed in memory */
  packet_filter_ingress_port_mask_t ingress_port_mask;
} dev;

static void write_ingress_port_mask(packet_filter_ingress_port_mask_t *mask) {
  iowrite8(mask->mask, INGRESS_PORT_FILTER(dev.virtbase));
  dev.ingress_port_mask = *mask;
}
```

```
static uint32_t read_packet_filter_in_pkts(int ingress) {
  return ioread32(CTR_IN_PKTS(dev.virtbase, ingress));
}

static uint32_t read_packet_filter_transf_pkts(int ingress) {
  return ioread32(CTR_TRANSF_PKTS(dev.virtbase, ingress));
}

static uint32_t read_packet_filter_in_frames(int ingress) {
  return ioread32(CTR_IN_FRAMES(dev.virtbase, ingress));
}

static uint32_t read_packet_filter_transf_frames(int ingress) {
  return ioread32(CTR_TRANSF_FRAMES(dev.virtbase, ingress));
}

static uint32_t read_packet_filter_inv_frames(int ingress) {
  return ioread32(CTR_INV_FRAMES(dev.virtbase, ingress));
}

static uint32_t read_packet_filter_drop_frames(int ingress) {
  return ioread32(CTR_DROP_FRAMES(dev.virtbase, ingress));
}

/*
 * Handle ioctl() calls from userspace:
 * Read or write the segments on single digits.
 * Note extensive error checking of arguments
 */
static long packet_filter_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
{
        packet_filter_arg_t vla;

  pr_info("ioctl %d\n", cmd);

        switch (cmd) {
        case PACKET_FILTER_WRITE_INGRESS_PORT_MASK:
                if (copy_from_user(&vla, (packet_filter_arg_t *) arg,
                                   sizeof(packet_filter_arg_t)))
                        return -EACCES;
                write_ingress_port_mask(&vla.ingress_port_mask);
                break;

        case PACKET_FILTER_READ_INGRESS_PORT_MASK:
          vla.ingress_port_mask = dev.ingress_port_mask;
                if (copy_to_user((packet_filter_arg_t *) arg, &vla,
                                 sizeof(packet_filter_arg_t)))
                        return -EACCES;
```

```
        break;

    case PACKET_FILTER_READ_CTR_IN_PKTS:
        if (copy_from_user(&vla, (packet_filter_arg_t *) arg,
                           sizeof(packet_filter_arg_t)))
            return -EACCES;
      vla.ingress_counter_target =
        read_packet_filter_in_pkts(vla.ingress_counter_target);
        if (copy_to_user((packet_filter_arg_t *) arg, &vla,
                         sizeof(packet_filter_arg_t)))
            return -EACCES;
      break;

    case PACKET_FILTER_READ_CTR_TRANSF_PKTS:
        if (copy_from_user(&vla, (packet_filter_arg_t *) arg,
                           sizeof(packet_filter_arg_t)))
            return -EACCES;
      vla.ingress_counter_target =
        read_packet_filter_transf_pkts(vla.ingress_counter_target);
        if (copy_to_user((packet_filter_arg_t *) arg, &vla,
                         sizeof(packet_filter_arg_t)))
            return -EACCES;
      break;

    case PACKET_FILTER_READ_CTR_IN_FRAMES:
        if (copy_from_user(&vla, (packet_filter_arg_t *) arg,
                           sizeof(packet_filter_arg_t)))
            return -EACCES;
      vla.ingress_counter_target =
        read_packet_filter_in_frames(vla.ingress_counter_target);
        if (copy_to_user((packet_filter_arg_t *) arg, &vla,
                         sizeof(packet_filter_arg_t)))
            return -EACCES;
      break;

    case PACKET_FILTER_READ_CTR_TRANSF_FRAMES:
        if (copy_from_user(&vla, (packet_filter_arg_t *) arg,
                           sizeof(packet_filter_arg_t)))
            return -EACCES;
      vla.ingress_counter_target =
        read_packet_filter_transf_frames(vla.ingress_counter_target);
        if (copy_to_user((packet_filter_arg_t *) arg, &vla,
                         sizeof(packet_filter_arg_t)))
            return -EACCES;
      break;

    case PACKET_FILTER_READ_CTR_INV_FRAMES:
        if (copy_from_user(&vla, (packet_filter_arg_t *) arg,
```

```
                                sizeof(packet_filter_arg_t)))
                        return -EACCES;
          vla.ingress_counter_target =
            read_packet_filter_inv_frames(vla.ingress_counter_target);
                if (copy_to_user((packet_filter_arg_t *) arg, &vla,
                                sizeof(packet_filter_arg_t)))
                        return -EACCES;
          break;

        case PACKET_FILTER_READ_CTR_DROP_FRAMES:
                if (copy_from_user(&vla, (packet_filter_arg_t *) arg,
                                sizeof(packet_filter_arg_t)))
                        return -EACCES;
          vla.ingress_counter_target =
            read_packet_filter_drop_frames(vla.ingress_counter_target);
                if (copy_to_user((packet_filter_arg_t *) arg, &vla,
                                sizeof(packet_filter_arg_t)))
                        return -EACCES;
          break;

        default:
                return -EINVAL;
        }

        return 0;
}

/* The operations our device knows how to do */
static const struct file_operations packet_filter_fops = {
        .owner          = THIS_MODULE,
        .unlocked_ioctl = packet_filter_ioctl,
};

/* Information about our device for the "misc" framework -- like a char dev */
static struct miscdevice packet_filter_misc_device = {
        .minor          = MISC_DYNAMIC_MINOR,
        .name           = DRIVER_NAME,
        .fops           = &packet_filter_fops,
};

/*
 * Initialization code: get resources (registers) and display
 * a welcome message
 */
static int __init packet_filter_probe(struct platform_device *pdev)
{
  packet_filter_ingress_port_mask_t ingress_mask = { 0b0000 };
        int ret;
```

```
        /* Register ourselves as a misc device: creates /dev/packet_filter */
        ret = misc_register(&packet_filter_misc_device);

        /* Get the address of our registers from the device tree */
        ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
        if (ret) {
                ret = -ENOENT;
                goto out_deregister;
        }

        /* Make sure we can use these registers */
        if (request_mem_region(dev.res.start, resource_size(&dev.res),
                               DRIVER_NAME) == NULL) {
                ret = -EBUSY;
                goto out_deregister;
        }

        /* Arrange access to our registers */
        dev.virtbase = of_iomap(pdev->dev.of_node, 0);
        if (dev.virtbase == NULL) {
                ret = -ENOMEM;
                goto out_release_mem_region;
        }

        /* Set initial values */
        write_ingress_port_mask(&ingress_mask);

        return 0;

out_release_mem_region:
        release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
        misc_deregister(&packet_filter_misc_device);
        return ret;
}

/* Clean-up code: release resources */
static int packet_filter_remove(struct platform_device *pdev)
{
        iounmap(dev.virtbase);
        release_mem_region(dev.res.start, resource_size(&dev.res));
        misc_deregister(&packet_filter_misc_device);
        return 0;
}

/* Which "compatible" string(s) to search for in the Device Tree */
#ifdef CONFIG_OF
```

```
static const struct of_device_id packet_filter_of_match[] = {
        { .compatible = "csee4840,packet_filter-1.0" },
        {},
};
MODULE_DEVICE_TABLE(of, packet_filter_of_match);
#endif

/* Information for registering ourselves as a "platform" driver */
static struct platform_driver packet_filter_driver = {
        .driver = {
                .name   = DRIVER_NAME,
                .owner  = THIS_MODULE,
                .of_match_table = of_match_ptr(packet_filter_of_match),
        },
        .remove = __exit_p(packet_filter_remove),
};

/* Called when the module is loaded: set things up */
static int __init packet_filter_init(void)
{
        pr_info(DRIVER_NAME ": init\n");
        return platform_driver_probe(&packet_filter_driver, packet_filter_probe);
}

/* Calball when the module is unloaded: release resources */
static void __exit packet_filter_exit(void)
{
        platform_driver_unregister(&packet_filter_driver);
        pr_info(DRIVER_NAME ": exit\n");
}

module_init(packet_filter_init);
module_exit(packet_filter_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("PacketFilter group");
MODULE_DESCRIPTION("Packet filter driver");
```

Listing 25: packet filter

```
#ifndef _PACKET_FILTER_H
#define _PACKET_FILTER_H

#include <linux/ioctl.h>

typedef struct {
  int mask;
} packet_filter_ingress_port_mask_t;
```

```
typedef union {
  packet_filter_ingress_port_mask_t ingress_port_mask;
  int ingress_counter_target;
} packet_filter_arg_t;

#define PACKET_FILTER_MAGIC 'q'

/* ioctls and their arguments */
#define PACKET_FILTER_WRITE_INGRESS_PORT_MASK _IOW(PACKET_FILTER_MAGIC, 1, packet_filter_arg_t
#define PACKET_FILTER_READ_INGRESS_PORT_MASK _IOR(PACKET_FILTER_MAGIC, 1, packet_filter_arg_t)
#define PACKET_FILTER_READ_CTR_IN_PKTS _IOR(PACKET_FILTER_MAGIC, 1, packet_filter_arg_t)
#define PACKET_FILTER_READ_CTR_TRANSF_PKTS _IOR(PACKET_FILTER_MAGIC, 1, packet_filter_arg_t)
#define PACKET_FILTER_READ_CTR_IN_FRAMES _IOR(PACKET_FILTER_MAGIC, 1, packet_filter_arg_t)
#define PACKET_FILTER_READ_CTR_TRANSF_FRAMES _IOR(PACKET_FILTER_MAGIC, 1, packet_filter_arg_t)
#define PACKET_FILTER_READ_CTR_INV_FRAMES _IOR(PACKET_FILTER_MAGIC, 1, packet_filter_arg_t)
#define PACKET_FILTER_READ_CTR_DROP_FRAMES _IOR(PACKET_FILTER_MAGIC, 1, packet_filter_arg_t)

#endif // _PACKET_FILTER_H
```