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 backpressure 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.

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

3 SYSTEM BREAKDOWN

This section describes the various blocks in our system.

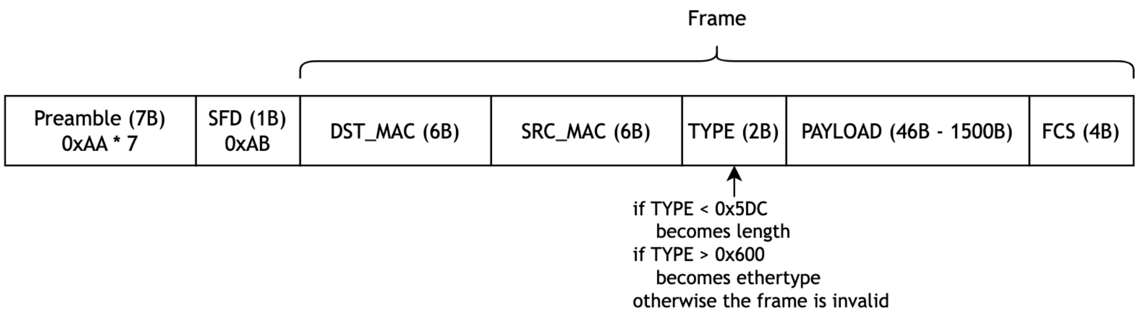


Figure 1: The applicable format of Ethernet frames our system can process.
[1]

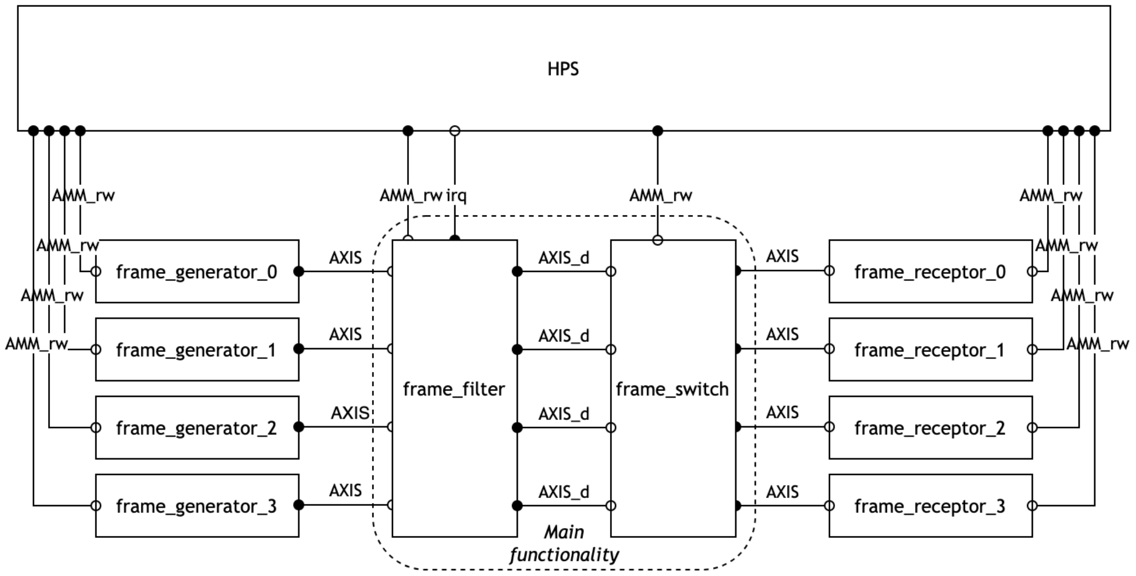


Figure 2: The breakdown of our system into hardware components that receive commands from software via the Avalon memory-mapped register interface. All hardware modules talk with the AXI-Stream protocol. Legend: filled dot is the producer; hollow dot is the consumer; AMM_rw is a read- and write-able 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.

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. 3 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 5 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 backpressure that the egress puts on the filter, even if it is within a single frame transmission. To avoid having to put intra-frame backpressure 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. 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 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

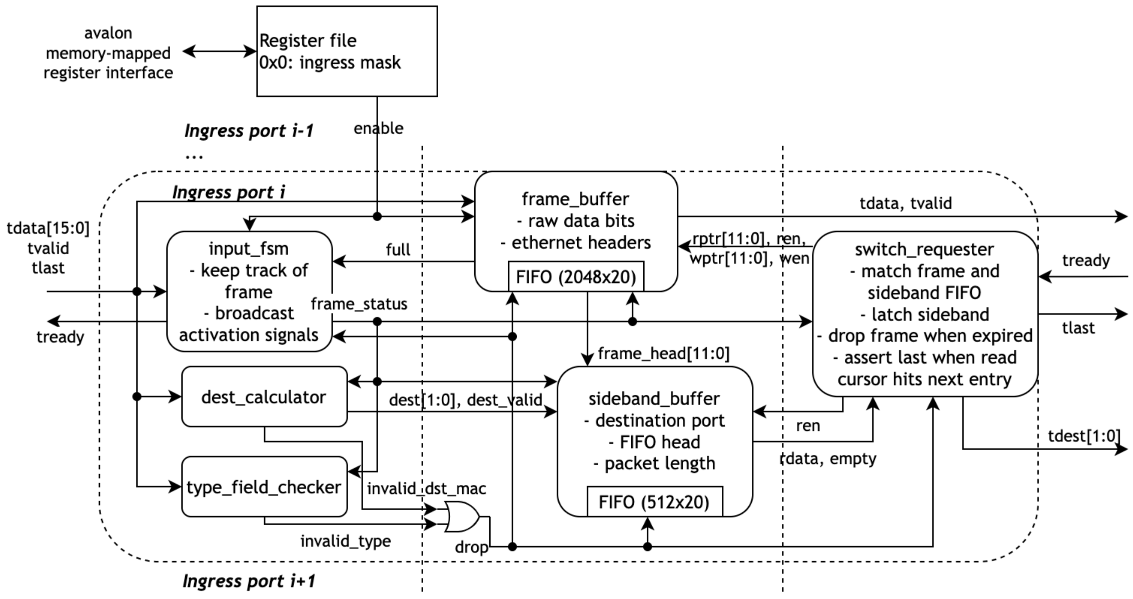


Figure 3: The connections within the input filter.

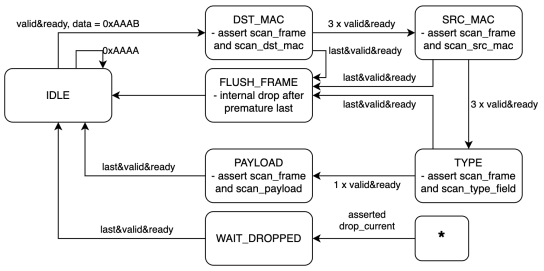

valid&ready, data = 0xAAAB
DST_MAC
- assert scan_frame and scan_dst_mac
3 x valid&ready
SRC_MAC
- assert scan_frame and scan_src_mac
0xAAAA
last&valid&ready
IDLE
FLUSH_FRAME
- internal drop after premature last
last&valid&ready
last&valid&ready
3 x valid&ready
last&valid&ready
PAYLOAD
- assert scan_frame and scan_payload
1 x valid&ready
TYPE
- assert scan_frame and scan_type_field
last&valid&ready
WAIT_DROPPED
asserted drop_current
*

Figure 4: The input state machine diagram.

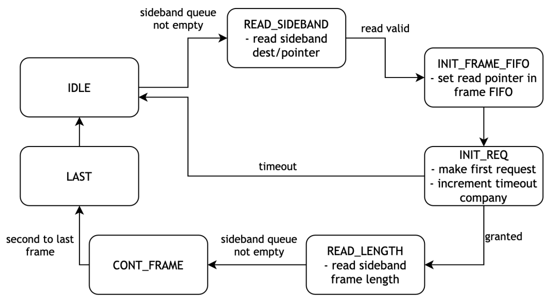


Figure 5: The request state machine diagram.

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 backpressure, 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 backpressure. 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 Egress Selection Network. a 4-to-1 multiplexer plus a match unit that, every cycle, looks only at the ingress port indicated by a two-bit request pointer (next_rr) coming from that egress-port's scheduler. If that ingress word is marked valid and its tdest equals the current egress index, the word is accepted and forwarded; otherwise the selector immediately reports miss back to the scheduler.

3.2.2 Per-egress Round-Robin Scheduler. 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.

3.2.3 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.

3.2.4 Back-pressure Propagation. each egress port's tready is fanned backwards through its scheduler to the selected ingress port's 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. Depending on experimental results, we may also have to implement FIFO queues before each egress port. The purpose would be to alleviate other egress ports if only one is stalling as discussed with backpressure in the input filter.

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 switch Round-Robin Scheduler uses for each egress port. Round robin satisfies two important constraints in network switching. The first is fairness. The scheduler serves any ingress in turn, and each ingress port relinquishes the grant following one frame transmission (or the input filter drops it after reaching the maximum frame length). Hence no single source can indefinitely starve other ingress ports from the same ingress port. Additionally, this deterministic maximum length provides a deterministic latency to each port. In the worst case, an ingress port must wait for three frames (with four ports) before the pointer wraps around. This assumes no backpressure in the egress ports. However, the request expiration functionality achieved with the input filter supports the switch's determinism once more.

4.0.1 Implementation. Each egress FSM keeps a 2-bit pointer next_rr that identifies "which ingress to try first". If the designated ingress has a valid word and its tdest matches the current egress port, the whole frame is accepted. When the tlast word passes, the FSM increments next_rr, and the process repeats as in Figure 3.

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.

4.1 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.

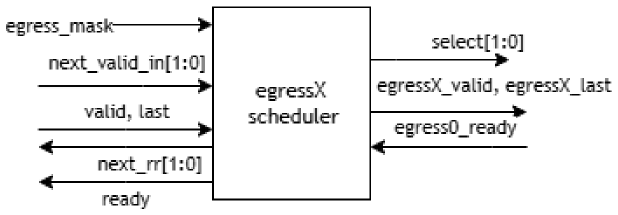


Figure 6: The Scheduler Interface

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.

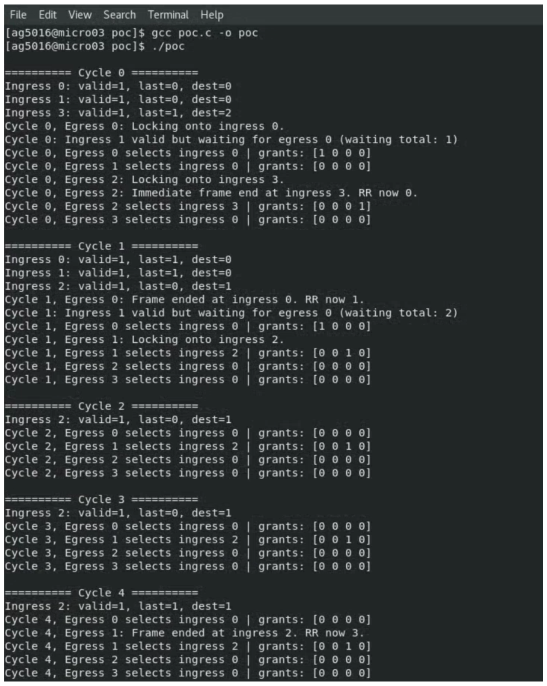
Ingress 1 waited 2 cycles before being able to transmit (shown in waiting stats).

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

5 RESOURCES

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



```
[ag5016@micro03 poc]$ gcc poc.c -o poc
[ag5016@micro03 poc]$ ./poc

========== Cycle 0 ==========
Ingress 0: valid=1, last=0, dest=0
Ingress 1: valid=1, last=0, dest=0
Ingress 3: valid=1, last=1, dest=2
Cycle 0, Egress 0: Locking onto ingress 0.
Cycle 0: Ingress 1 valid but waiting for egress 0 (waiting total: 1)
Cycle 0, Egress 0 selects ingress 0 | grants: [1 0 0 0]
Cycle 0, Egress 1 selects ingress 0 | grants: [0 0 0 0]
Cycle 0, Egress 2: Locking onto ingress 3.
Cycle 0, Egress 2: Immediate frame end at ingress 3. RR now 0.
Cycle 0, Egress 2 selects ingress 3 | grants: [0 0 0 1]
Cycle 0, Egress 3 selects ingress 0 | grants: [0 0 0 0]

========== Cycle 1 ==========
Ingress 0: valid=1, last=1, dest=0
Ingress 1: valid=1, last=1, dest=0
Ingress 2: valid=1, last=0, dest=1
Cycle 1, Egress 0: Frame ended at ingress 0. RR now 1.
Cycle 1: Ingress 1 valid but waiting for egress 0 (waiting total: 2)
Cycle 1, Egress 0 selects ingress 0 | grants: [1 0 0 0]
Cycle 1, Egress 1: Locking onto ingress 2.
Cycle 1, Egress 1 selects ingress 2 | grants: [0 0 1 0]
Cycle 1, Egress 2 selects ingress 0 | grants: [0 0 0 0]
Cycle 1, Egress 3 selects ingress 0 | grants: [0 0 0 0]

========== Cycle 2 ==========
Ingress 2: valid=1, last=0, dest=1
Cycle 2, Egress 0 selects ingress 0 | grants: [0 0 0 0]
Cycle 2, Egress 1 selects ingress 2 | grants: [0 0 1 0]
Cycle 2, Egress 2 selects ingress 0 | grants: [0 0 0 0]
Cycle 2, Egress 3 selects ingress 0 | grants: [0 0 0 0]

========== Cycle 3 ==========
Ingress 2: valid=1, last=0, dest=1
Cycle 3, Egress 0 selects ingress 0 | grants: [0 0 0 0]
Cycle 3, Egress 1 selects ingress 2 | grants: [0 0 1 0]
Cycle 3, Egress 2 selects ingress 0 | grants: [0 0 0 0]
Cycle 3, Egress 3 selects ingress 0 | grants: [0 0 0 0]

========== Cycle 4 ==========
Ingress 2: valid=1, last=1, dest=1
Cycle 4, Egress 0 selects ingress 0 | grants: [0 0 0 0]
Cycle 4, Egress 1: Frame ended at ingress 2. RR now 3.
Cycle 4, Egress 1 selects ingress 2 | grants: [0 0 1 0]
Cycle 4, Egress 2 selects ingress 0 | grants: [0 0 0 0]
Cycle 4, Egress 3 selects ingress 0 | grants: [0 0 0 0]
```

Figure 7: Arbitration Simulation Result

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.

6 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.

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
0x2100	frame_generator	Ethernet frame generator 1
0x2200	frame_generator	Ethernet frame generator 2
0x2300	frame_generator	Ethernet frame generator 3
0x2400	frame_receptor	Ethernet frame receptor 0
0x2500	frame_receptor	Ethernet frame receptor 1
0x2600	frame_receptor	Ethernet frame receptor 2
0x2700	frame_receptor	Ethernet frame receptor 3

Table 2: Base addresses in the system.

6.0.1 Frame generator. For 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.

Address offset	Read/ Write	Name	Description
0x00	RW	Check_ Sum_ Register	Readable register for testbenching purposes

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)

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.

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.

6.0.2 Frame filter. The frame filter has one register as per Table 4.

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

6.0.4 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.

Address offset	Read/ Write	Name	Description
0x00	RW	Check_ Sum_ Register	Readable register that stores the checksum result for testbenching purposes
0x04	RW	DST_ Check_ Register	Readable/Writable register that stores the result of destination comparison for testbenching purposes

Table 6: Register map for frame receptors

REFERENCES

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