VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

LOGIC DESIGN PROJECT

NetFPGA-SUME

High Level Synthesis

Instructors: Dr. Tran Ngoc Thinh
Mr. Le Tan Long
Mr. Tran Hoang Quoc Bao
Members: Nguyen Ly Dang Khoa - 1952785
Nguyen Dinh Nam - 1952858

HO CHI MINH City, December 2021

FOREWORD

"Chúng em xin cảm ơn thầy Trần Ngọc Thịnh cùng các anh trợ giảng của phòng thí nghiệm Kỹ thuật Máy Tính trường đại học Bách Khoa TpHCM đã dành thời gian, tạo mọi điều kiện, hướng dẫn chi tiết trong quá trình học tập môn đồ án thiết kế luận lý này. Dù tình hình dịch bệnh khiến chúng em không được phép có mặt tại trường nhưng thầy và các anh luôn hỗ trợ đầy đủ cho chúng em khi thực hiện đề tài này.

Chúng em xin cảm ơn anh Trần Hoàng Quốc Bảo, anh Lê Tấn Long và anh Huỳnh Phúc Nghị đã luôn hỗ trợ, giải đáp các thắc mắc, câu hỏi của chúng em trong quá trình tìm hiểu, setup công cụ hỗ trợ cho việc thực hiện đồ án của chúng em."

HO CHI MINH City, December 2021

Members

Nguyen Ly Dang Khoa, Nguyen Dinh Nam

INTRODUCTION

With this project, our group's job is to implement a partial module of the Learning switch using High Level Synthesis method on NetFPGA-SUME board.

In modern society like nowadays, more and more people are capable of learning how to program with some high-level programming languages such as C, C++, Python..., those programming languages have characteristics that are close to verbal language of human and can be read, spread out and understood easily. However, there are some other programming languages that are relating to hardware (hardware description languages or HDL) are still not so popular comparing to high-level programming languages, leading to some significant difficulties when doing hardware coding projects.

With this topic of High Level Synthesis method, we have a chance to learn and work with Vivado HLS tool of Xilinx, a tool that provides a really good environment for co-development between firmware and software. It facilitates interfacing with high-level programming languages like C, C++ or Python. After several weeks of studying, we observed that this method is significantly helpful with doing hardware projects using just high-level programming languages.

Contents

1 ABSTRACT

1.1 Project Overview

Work with Xilinx Vivado HLS tool and NetFPGA-SUME board to implement output port lookup module of the Learning Switch using High Level Synthesis method.

1.2 Objectives and Limitation

- Objectives:

  - Study the principle and operation of the switch and other partial modules, especially output port lookup as well as other knowledge corresponding to packets, computer network, High-Level Synthesis...

  - Practice using Xilinx Vivado and Vivado HLS tool.

  - Implement output port lookup module using C++ language, synthesis and simulate with Vivado HLS tool.

  - Export RTL IP and import to the Switch Vivado project.

- Project limitation:

  Project is hold on semester 211 and the specific module of the Switch that is implemented is the output port lookup.

1.3 Practical Significance

With this topic of High Level Synthesis method, we can learn and work with Vivado HLS tool of Xilinx, a tool that provides a really good environment for co-development between firmware and software. It facilitates interfacing with high-level programming languages like C, C++ or Python. In sum, we observed that this method is

significant in doing hardware projects using just high-level programming languages.

Another practical significance is that we can study and observe the principle and operation of the Learning Switch as well as other knowledge corresponding to the computer network such as network packets, AXI-Stream... and other hardware coding experience.

1.4 Work of members

Member	Work
Nguyen Ly Dang Khoa	MAC lookup table, Fifo and finishing implementing project.
Nguyen Dinh Nam	Ethernet parser and writing report.

2 BASE KNOWLEDGE

2.1 Basics

- **Programming language:** Verilog and C++

  Read and understand reference projects programmed with Verilog HDL, after that implement those modules using C++ programming language.

- **Environment:** Ubuntu/Linux

  Work with Xilinx Vivado HLS tool on Ubuntu or Linux operating system.
  Check **Software Requirement** section in the Appendix for installation.

- **Relevant basic knowledge:** FPGA, specifically with NetFPGA-SUME board.

2.2 Technical Knowledge

2.2.1 AXI-Stream

The AXI-Stream protocol is used as a standard interface to exchange data between connected components. AXI Stream is a point-to-point protocol, it can be used to connect a single master, that transmit data, and a single slave, that receive data. The stream protocol defines the association between Transfers and Packets.

Some frequently used signals in AXI-Stream:

Signal	Description
TVALID	Required, source from Transmitter, indicates the Transmitter is driving a valid transfer.
TREADY	Optional but highly recommended, source from Receiver to accept transfer.
TDATA	Provide data that is passing across the interface, source from Transmitter.
TUSER	Any sideband user-defined data along with data stream.
TLAST	Boundary of a packet, source from Transmitter
TID	Stream identifier, source from Transmitter
ACLK	Global clock signal, active HIGH
ARESETn	Global reset signal, during reset, TVALID must be LOW

For a transfer to occur, both TVALID and TREADY must be asserted. Either TVALID or TREADY can be asserted first, or both can be asserted in the same ACLK cycle.
Handshake in AXI-Stream:

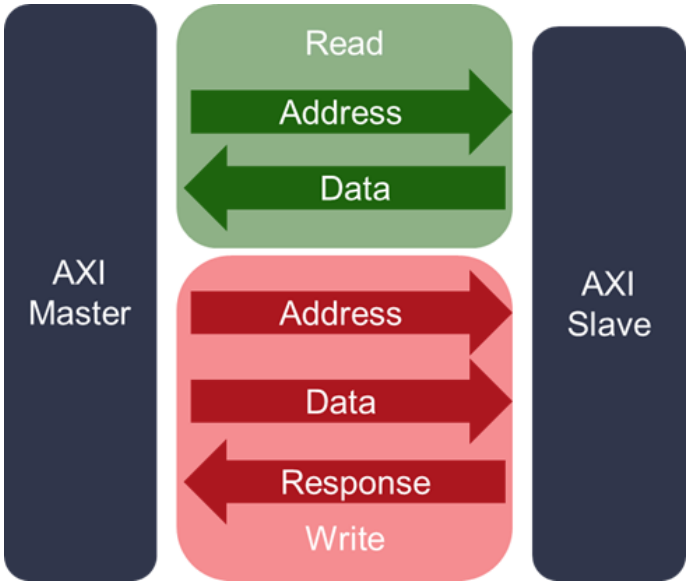

Figure 1: Five channels in an AXI-Stream

All 5 transaction channels (read and write address; read and write data and write response) use the same **TVALID/TREADY** handshake process to transfer address, data, and control information.
Handshake occurs as described in the steps below (in any channel):

- The source generates the **TVALID** signal to indicate when the address, data or control information is available.
- The destination generates the **TREADY** signal to indicate that it can accept the information.
- Transfer occurs only when both the **TVALID** and **TREADY** signals are HIGH.

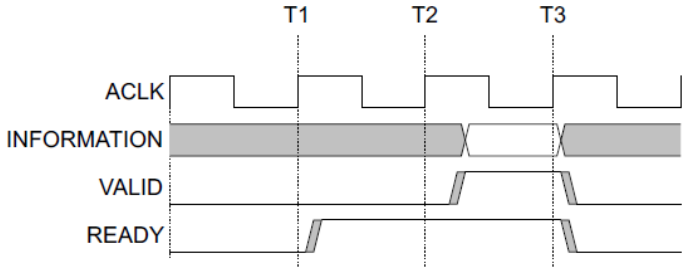

Figure 2: READY before VALID

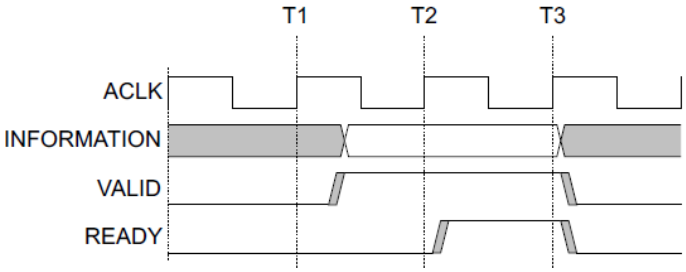

Figure 3: VALID before READY

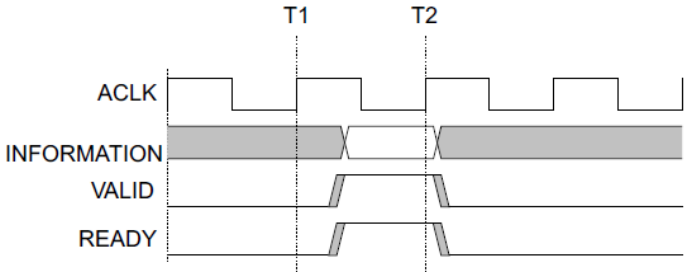

Figure 4: READY with VALID

2.2.2 Packets

Packets are a group of bytes that are transported together across an AXI-Stream interface. A network packet is a small amount of data sent over Transmission Control Protocol/Internet Protocol (TCP/IP) networks. The packet size is around 1.5 kilobytes for Ethernet and 64 KB for IP payloads. Interconnected components can use packets to deal more efficiently with a stream in packet-sized groups.

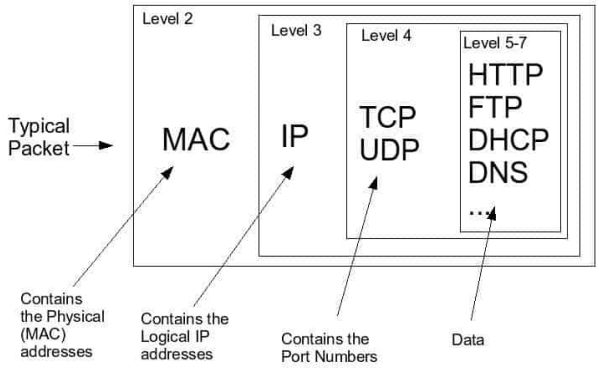

Figure 5: Network packets in OSI Model

The routing of network packets requires two network addresses, the source address of the sending host, and the destination address of the receiving host.

Similar to a real-life package, each packet includes a source and destination as well as the content (or data) being transferred. When the packets reach their destination, they are reassembled into a single file or other contiguous block of data.

Structure of a packet:

- **The header** contains instructions related to the data in the packet. These instructions can include: IP, Source and Destination IP address...
- **The payload** is the data within the packet. This is the basic information that the packet delivers to the destination. The payload is often padded with blank information to accommodate a fixed-length packet.
- **Trailers** are footer bits that signify the end of a packet. These

bits inform the receiving device that it has reached the end of the packet and may also include a type of error checking protocol.

2.2.3 Switch

A switch is a device that manage physical networks, as well as software-based virtual devices. A network switch connects devices (such as computers, printers, wireless access points) in a network to each other, and allows them to interact by exchanging data packets.

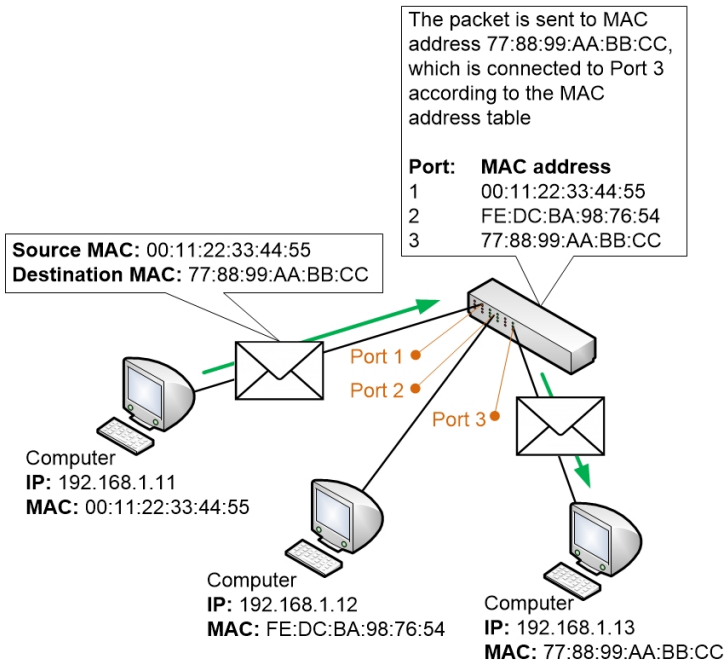

Figure 6: How the switch works

When a device is connected to a switch, it notes the device's MAC address, which identifies that physical device. With a list of port numbers and MAC addresses, it forms a lookup table used for searching and broadcasting packets.

The switch takes in packets being sent by transmitting devices that are connected to its physical ports and sends them out again, but only through the destination ports that lead to the receiving devices the packets are planned to reach. Detailed operations of selecting destination port is presented in the Output port lookup section.

2.2.4 Output port lookup

The function of this block is to set the destination port meta data field for all packets.

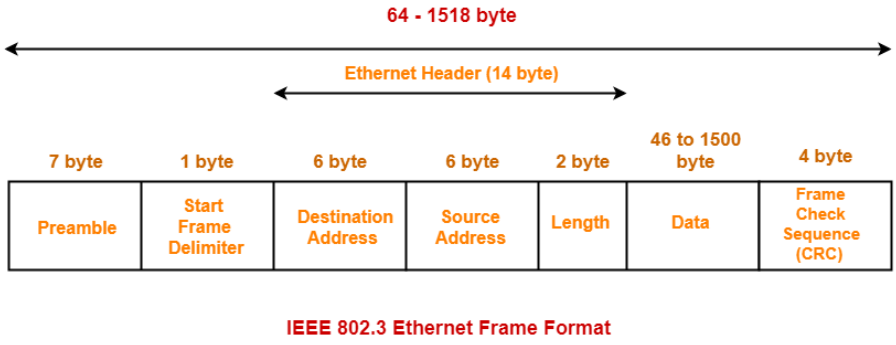

Figure 7: IEEE 802.3 Ethernet frame format

In the Learning switch each packet that arrives successfully (TLAST is High along with TVALID), it first goes through an Ethernet parser. The parser extracts the source and destination MAC address from TDATA field and the source port from TUSER field of the stream.

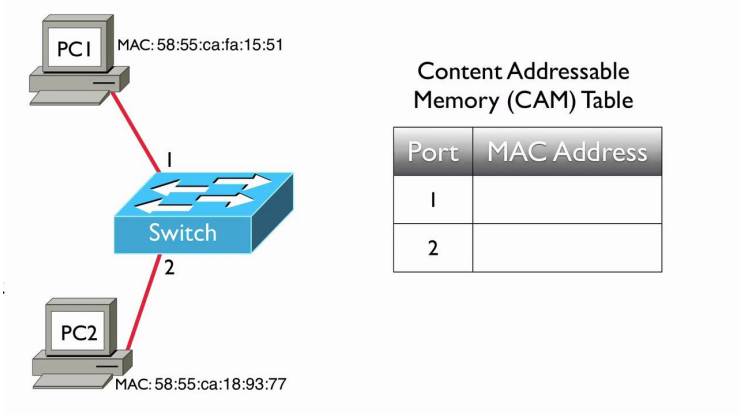

Figure 8: Lookup table

The source and destination MAC address are next looked up in the CAM, implementing a lookup table (LUT) of port numbers and MAC addresses. If the results is a hit (2 MAC addresses are found in LUT), the packet is sent to the destination ports indicated by the lookup (except for the source port), if the result is a miss, the destination ports are set to broadcast to all output ports (except for the source port). If the source MAC is not in the LUT, it is learned for future lookups.

2.2.5 Pragma HLS library

Many HLS tools include pragmas to gain optimization of area and performance. Efficient use of pragmas, together with code reorganizations to make pragmas effective is the key design and optimization technique for HLS. Unrolling, pipelining, memory partitioning... are some pragma supplementaries.

Classes provided by pragma:

Class	Operation
Interface	Define function interface
Loop optimization	Loop Unrolling Loop Pipelining Loop Merge operations
Memory controll	Array partition, etc.
Function call	Flow optimization

Some pragma syntaxes:

- **Array partition:**

```
#pragma HLS array_partition variable=<name> \
<type>  factor=<int>  dim=<int>
```

  where:

  * variable=<name>: A required argument that specifies the array variable to be partitioned.
  * <type> and <factor>: Factor is an integer n; Type can have the value of cyclic (n cyclic arrays), block (n equal blocks) or complete (default type, decompose the array into n individual elements).

- **Interface:**

```
#pragma HLS interface <mode> port=<name> \
bundle=<string> register register_mode=<mode> \
depth=<int> offset=<string> clock=<string> \
name=<string> num_read_outstanding=<int> \
```

```
num_write_outstanding=<int> \
max_read_burst_length=<int> \
max_write_burst_length=<int>
```

where:

* <mode>: Can have values of ap_none, ap_stable (No protocol. The interface is a data port, ap_none is default for HLS input), besides, there are some other values of ap_vld, ap_fifo, bram... corresponding with each specific interface.
* <bundle>: Groups function arguments into AXI interface ports, can have values of s_axilite, m_axi...

– **Unroll:**

```
#pragma HLS unroll factor=<N> region \
skip_exit_check
```

where:

* factor=<N>: Specifies a non-zero integer indicating that partial unrolling is requested.
* region: An optional keyword that unrolls all loops within the body (region) of the specified loop, without unrolling the enclosing loop itself.

2.3 Related Project

Related work that we choose for reference is the **reference_switch** project dedicated to NetFPGA that implements the whole network

switch. This project can be found at projects folder of **NetFPGA-SUME-live** private GitHub repository that can be accessed after registering NetFPGA Beta program.
IP Cores of the project:

- nf_10g_interface
- DMA
- input_arbiter
- switch_output_port_lookup
- output_queue

Below are some descriptions about the principle and operation of this reference switch.
Packets first enter the device through the **nf_10g_interface** module, which is an IP that combines Xilinx AXI 10G Ethernet subsystem, in addition to an AXI-Stream adapter. Every incoming packet is annotated with metadata and is finally transformed into 256-bit AXI-Stream.
The **nf_10g_interface** modules receiving side connect next to the **input_arbiter** module. The input arbiter has four input interfaces from the **nf_10g_interface** modules and one from a **DMA** module. Each input to the arbiter connects to an input queue, which is in fact a small fall-through FIFO. The simple arbiter rotates between all the input queues in a round robin manner, each time selecting a non-empty queue and writing one full packet from it to the next stage in the data-path.
The **output port lookup** module is responsible for deciding which

port a packet goes out of. After that decision is made, the packet is then handed to the **output queues** module. The lookup module implements a simple Ethernet Parser and a Learning MAC CAM lookup table (LUT). Packets with unknown destination MAC address are broadcasted.

Once a packet arrives to the **output queues** module, it already has a marked destination (provided on a side channel - The TUSER field). According to the destination it is entered to a dedicated output queue. There are five such output queues: one per each 10G port and one to the **DMA** block. When a packet reaches the head of its **output queue**, it is sent to the corresponding output port, being either an **nf_10g_interface** module or the **DMA** module.

In the Reference Switch design the DMA module is used only for register access. To the other NetFPGA modules it exposes AXIS (master+slave) interfaces for sending/receiving packets, as well as a AXI4-LITE master interface through which all AXI registers can be accessed from the host.

3 ARCHITECTURE ANALYSIS AND IMPLEMENTATION

3.1 Output port lookup architecture

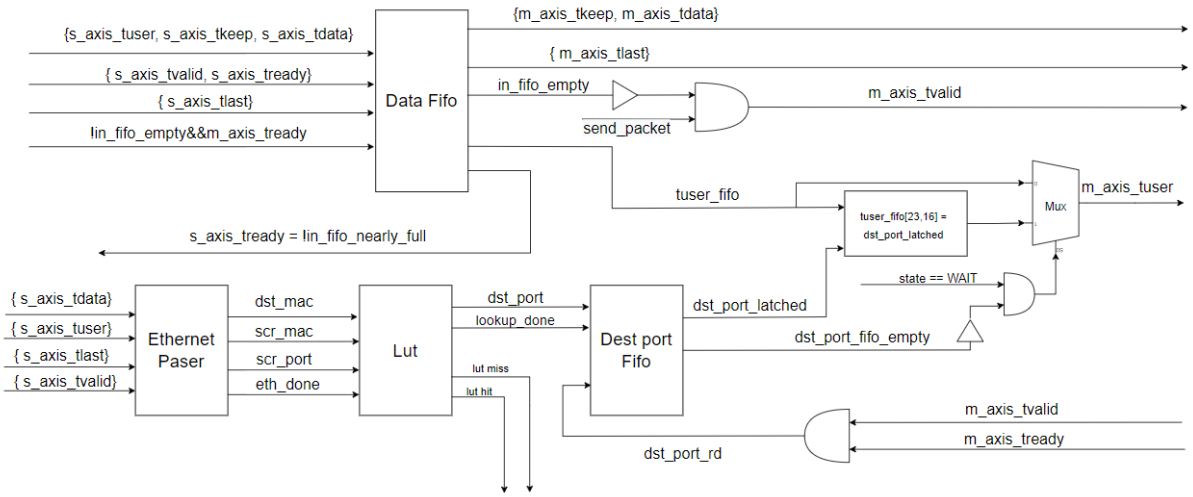

Figure 9: Output port lookup structure

The picture above shows the general structure of the output port lookup module which consists of 4 main components and a mechanism for transferring data from the module:

- 2 Fifos (Data Fifo and Dest port Fifo).
- An Ethernet parser module.
- A Lookup table module.

3.1.1 Data Fifo

Data fifo acts as a buffer that temporarily contains packets loaded in from a master module and transfers that data to a slave module, From master port interface, when both s_axis_tready and s_axis_tvalid are high, Data fifo is enabled to load in data, s_axis_tvalid

is an input signal from the master module indicating the data is valid when going high and the s_axis_tready signal is the invert of in_fifo_nearly_full signal indicating there is still available space in the fifo to load in data when going high. From the slave port interface, when in_fifo_empty is low and m_axis_tready is high, this indicates Data fifo has data to transfer and the slave module is ready to receive data, thus, Data fifo transfers out the oldest piece of data according to the FiFo data management policy, data stored in the fifo consists of 4 fields:

- **TDATA**: contains the actual content of the packet (256 bit width).

- **TKEEP**: distinguishes between the valid bytes and the invalid bytes in TDATA field (32 bit width).

- **TUSER**: contains information about the source port of the packet and the destination port that the packet should go to (128 bit width).

- **TLAST**: indicates the end of the packet when going high, this is used along with TVALID siganl to mark the end of one packet and the beginning the the next packet (1 bit width).

3.1.2 Ethernet parser

When a new packet arrives (signaled by TLAST going high in the previous clock cycle and TVALID being high in the current clock cycle) , its data fields are split to enter both Data Fifo and Ethernet parser in parallel. The Ethernet parser is responsible for parsing out the needed information in fields TDATA and TUSER.

DEST_MAC and **SRCS_MAC** representing the **destination mac address** and the **source mac address** respectively. At the arival time of the packet signaled by TLAST turning high at the previous clock, they are taken out from TDATA. The bits ranging

from 0 to 47 are the value of **DEST_MAC** and bits ranging from 48 to 95 are the value of **SRCS_MAC**.

SOURCE_PORT is the one hot encoded address of the source port, it's value ranges from bit 16 to bit 24 in field TUSER and is taken out when the package arrives. Just right after processing the first 128 bits of the arrived packet, the module halts, when the LAST signal goes high indicating the end of the packet, the module is re enabled to function.

After obtaining the required information from the packet, these in formation (**DEST_MAC**, **SRCS_MAC**, **SOURCE_PORT**) will then be passed to the next module LUT in the structure for further processing, by setting **eth_done** signal to high indicating the values passed to LUT module are valid.

3.1.3 LUT

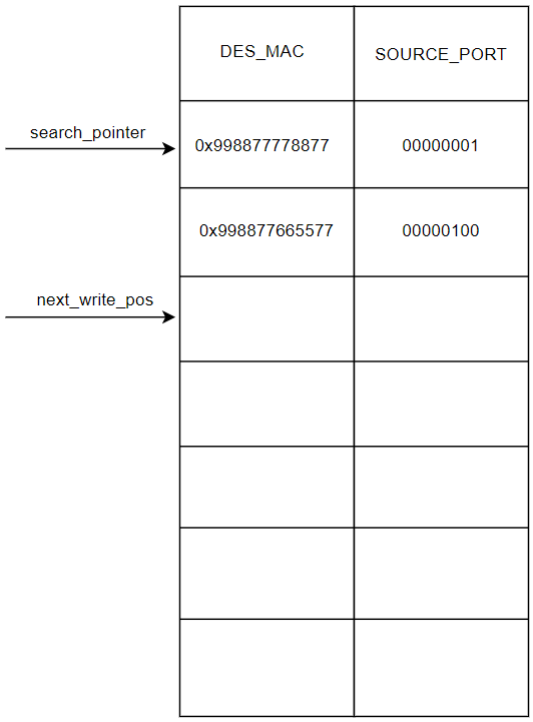

Figure 10: lookup table in LUT module

Lut module's main operation is as follow:

When the **eth_done** signal from Ethernet parser module goes high, Lut module will search in its lookup table for any matching in terms of value between the data arrived at input **dest_mac** port and the field **DES_MAC** in each entry of the table. If a match occurs the corresponding **SOURCE_PORT** field in the entry will be assigned to the **dest_port** output (8 bit width), otherwise a default port missing value (0x55) will be assigned instead.

At the same time when performing dest_mac matching, the LUT module also so does searching for the value arrived at input **src_mac** port in the lookup table, if no matching occurs at last, the entry whose position is the value of **next_write_pos** will be assigned with new value where its **DEST_MAC = src_mac** and **SOURCE_PORT = src_port** (8 bit width), after that, **next_write_pos** will increase by 1 or return to 0 if it is at the last entry.

After processing searching and learning **lut_hit** and **lut_miss** are asigned to 1 or 0 according to whether the table contains the current dest_mac value or not. **lookup_done** indicates the process is complete and is set at the end and reset at the next clock signal.

3.1.4 Dest port Fifo

Dest port Fifo module contains the address of the destination port of each packet in the Data Fifo. It's inputs are the **dest_port** bus

(input port of addresses to be buffered), **lookup_done** signal (representing **Write** signal) from Lut module and also **dst_prort_rd** signal (representing **Read** signal) which is a combinational signal between **m_axis_tvalid** and **m_axis_tready**. The module loads in data when **lookup_done** goes high, and transfers data when **dst_prort_rd** goes high, in the situation when both **Write** signal and **Read** signal goes high (lookup_done = dst_prort_rd = 1) Dest port Fifo works in pipeline manner, meaning that a new address from **dest_port** value can be loaded in at the same time the oldest dest_port address removed from Dest port Fifo. The outputs of Dest port Fifo are the **dst_port_latched**(8 bit width) bus carrying the one hot encoded address of the output queue and the **dst_port_fifo_empty** (1 bit width) signal indicating whether the fifo is empty or not.

3.1.5 How data get transferred from Output port lookup module:

A mechanism is used to control the **send_packet** signal, which partially affects the **m_axis_tvalid** signal, the **m_axis_tvalid** signal combines with the **m_axis_tready** signal through an AND gate to produce the **dst_port_rd** signal which is the Read enable signal of the Dest port Fifo. Hence when **send_packet** signal goes low the value at **dst_port_latch** port is held but not removed from the Dest port Fifo. By implementing a finite states machine to control the **send_packet** signal value the Output port lookup module ensures to function according to the master AXI stream transmision protocol.

The finite state machine is presented below.

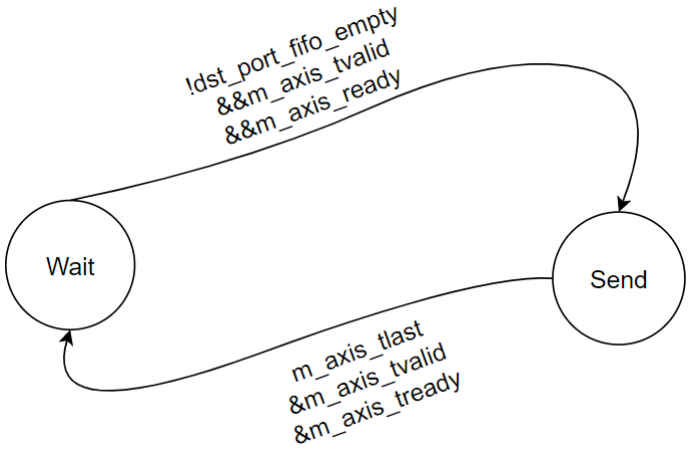

Figure 11: Finite state for sending mechanism

At each state the module performs certain specific tasks:

Wait: At wait state **send_packet** signal is assigned to 0 so the **m_axis_tvalid** signal is 0 to indicate the data is invalid. If the **dst_port_fifo_empty** signal is low this means data is available hence **send_packet** signal turns 1 at this time, also the value of TUSER field from bit 16 to 23 is replaced by the value at the **dst_port_latched**, if **in_fifo_empty** signal at Data Fifo is 0 then **m_axis_tvalid** signal goes high indicating the data is valid to send. To move from **WAIT** state to **SEND** state, **m_axis_tready** signal is required to go high, otherwise both Data Fifo and Dest port Fifo preserves their out put values until **m_axis_tready** signal goes high.

SEND: The operation in **SEND** state is a lot simpler. Basically it keeps sending the remaining part of the packet according to the behaviour of **m_axis_tready** signal, **dst_port_rd** turns low so Dest_port_Fifo preserves the value at its **dst_port_latched** port, this makes sense because the current value at **dst_port_latched** port is the destination port address of the next packet. The state

switches back to **WAIT** when detecting a high signal at **m_axis_t-last** port.

3.2 HLS implementation

The Output_Port_Lookup module has two main operations: Read and Write. These operations are implemented as two modules **Read** and **Write** functioning in parallel as well as sharing two common resources. The below figure shows the structure of Output_Port_Lookup module in HLS:

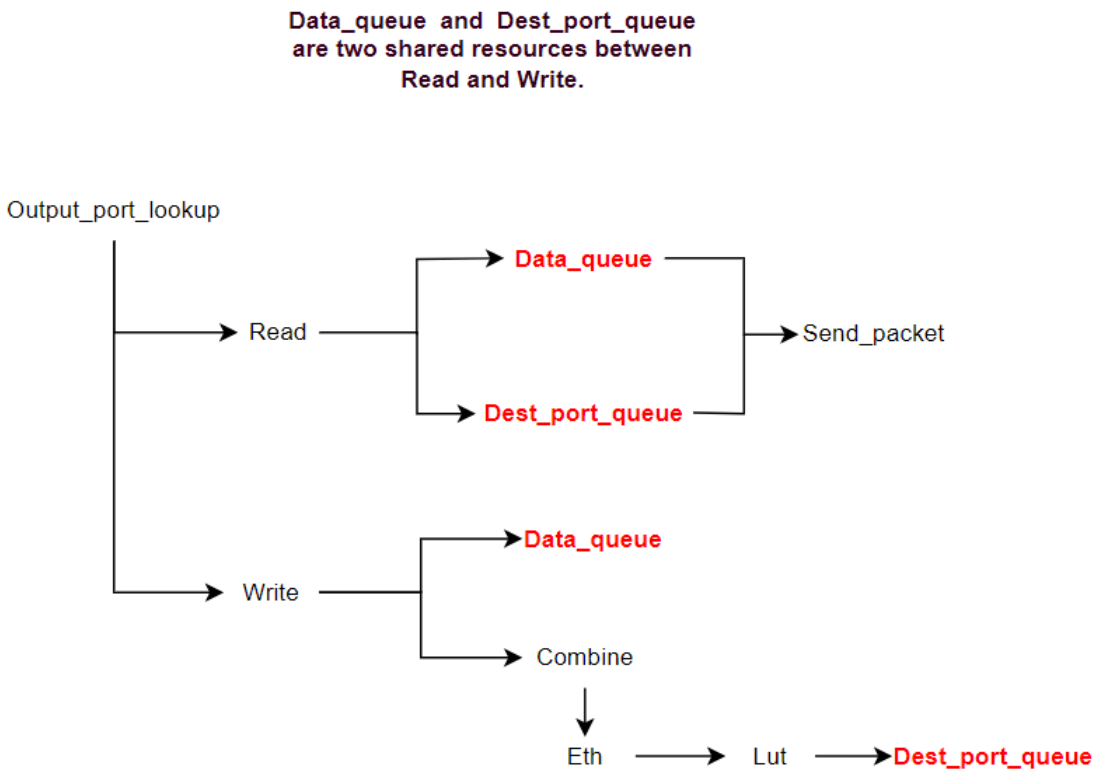

Figure 12: Structure of Output_Port_Lookup in HLS

Apart from the common resources (**Data_queue**, **Dest_port-**

_queue), the two modules have their own separate sub modules to assist their opposite functionalities.

3.2.1 Write module

It contains a sub module called **Combine** which is the combination of **Eth** and **Lut**, the work of **Combine** is first to extract **SOURCE_MAC**, **DEST_MAC** and **Src_PORT** from arrival packet via **Eth**, these information will then be passed to **Lut** to obtain either an address of **Dest_PORT** (if it was recorded) or a default missing value 0x55, at the same time performs learning the **Src_PORT** address of the **SOURCE_MAC** in **Lut**. The **Combine** sub module's output is the **DEST_PORT** address or the default missing value, either way, this value will be loaded to **Dest_port_queue**, the packet is also loaded to **Data_queue** at the same time.

Optimization: In order to optimize the searching and learning process, several HLS pragmas are added to the implementation of LUT as shown in the code below:

To give a little of context, the structure of lookup table is defined as:

```
1 #define LUT_BIT_LENGTH 4
2 #define NUM_OUTPUT_QUEUES 8
3 #define LUT_LENGTH 1<<4
4 #define DATA_WIDTH 48
5 #define DEFAULT_MISS_OUTPUT_PORTS  0x55
6
7 struct Lut{
8
```

```
9       struct{
10              ap_uint<NUM_OUTPUT_QUEUES> dst_port; //dest
    port address
11              ap_uint<DATA_WIDTH> dest_mac; //dest mac
    address
12        }table[LUT_LENGTH]; //LUT_LEHGTH is 16
13
14              ap_uint<LUT_BIT_LENGTH> replace_pos;
15              //replacing position starts at 0
16
17 };
```

Array partition is added right after declaring the lookup table object **lut**:

```
1  void Lut(*/...*/){
2
3  */...*/
4
5  static struct Lut lut; //All variables are initialized
       to 0 when declaring static
6
7  #pragma HLS ARRAY_PARTITION variable = lut.table dim =1
       complete
8
9  */...*/
10
11 }
```

Array in HLS are synthesized to RAM , each RAM is at most dual ported hence there will be delay when accessing multiple entries in RAM at the same time, pragma **ARRAY_PARTITION** is used to split the array's elements into separate units (in this case because **complete** mode is specified, each element is a unit) so multiple units can be accessed in parallel.The following piece of code will show when

this such a feature of partition is utilized.

```
for(int i=0; i<16; i++){

    #pragma HLS UNROLL    //This pragma is used so 16 entries will be processed in parallel

    // Search for dst mac

    if(lut.table[i].dest_mac==dst_mac)
    {
        dest_port = lut.table[i].dst_port;

        lookup_hit =1;
    }


    //Search for source mac

    if(lut.table[i].dest_mac==src_mac)
    {

        lut.table[i].dst_port = src_port;

        lut_learn_hit =1;

    }

}
```

The pragma **HLS UNROLL** makes the searching and matching process occurs in parallel over 16 entries, this means that 16 entries in the array will be accessed at the same time, without pragma **ARRAY_PARTITION** this will cost a tremendous amount of time

for accessing, thanks to the **ARRAY_PARTITION** pragma, each entry is now has it's own I/O ports so they can be accessed independently.

3.2.2 Read module

For clearer explanation, the interface of **Read** module in C++ is presented below:

```
//MASTER AXI Stream Data Width
#define  C_M_AXIS_DATA_WIDTH   256
#define  C_M_AXIS_TUSER_WIDTH  128


void Read(//Input
     ap_uint<1>                       m_axis_tready,
     ap_uint<1>                               reset,

     //Output
     ap_uint<C_M_AXIS_DATA_WIDTH>   &m_axis_tdata,
     ap_uint<C_M_AXIS_DATA_WIDTH/8> &m_axis_tkeep,
     ap_uint<C_M_AXIS_TUSER_WIDTH>  &m_axis_tuser,
     ap_uint<1>                     &m_axis_tvalid,
     ap_uint<1>                      &m_axis_tlast);
```

The **Read** module's main functionality is to transfer data whenever the following condition is satisfied: both **m_axis_tready** signal and **m_axis_tvalid** are high (AXI hand shake). For **m_axis_tready** signal, it is an external signal from a slave module used to tell whether the slave module is ready to receive or not. For **m_axis_tvalid** signal, it tells whether **Data_queue** and **Dest_port_queue** are empty, if it is low, this means both queues are empty and vice versa. When the slave module is ready to receive new packet and

the queues are not empty, the **Send_packet** module will assign the **Dest_port** address of the current packet to it's appropriate position in TUSER field in the first 128 bits and then wait until the last signal goes high to be enabled again.

3.2.3 Dest_port_queue and Data_queue

The two queues are declared as struct type named queue with template specified as below:

```
1
2  #ifndef _QUEUE_H
3  #define _QUEUE_H
4
5  #include "ap_int.h"
6
7  #define MAX_BIT_DEPTH 4
8  #define MAX_DEPTH 1<<4
9  #define NUM_OUTPUT_QUEUES 8
10
11 template<typename T>
12 struct Queue{
13
14     static T buffer[MAX_DEPTH];
15
16     static ap_uint<MAX_BIT_DEPTH> wr_pos;
17
18     static ap_uint<MAX_BIT_DEPTH> rd_pos;
19
20     static int size;
21
22     bool isNearlyFull();
23
24     bool isEmpty();
25
```

```
26      void Enqueue(T data);
27
28      void Dequeue();
29
30
31      T Front();
32
33
34      void Reset();
35
36  };
37
38  #endif
```

The static components in the queue ensures that for each data type replaced for template, all it instances in the program share a common set of static components, so that makes the declaration of **Data_queue** in **Write** module and that in **Read** module linked together on the same resource, one modification of each will affect the other, the same thing happends for **Dest_port_queue** in both **Read** and **Write** modules.

3.3 Output_port_lookup summary

```
1  void switch_output_port_lookup(
2      //SLAVE Stream Ports
3      ap_uint<C_S_AXIS_DATA_WIDTH>     s_axis_tdata,
4      ap_uint<C_S_AXIS_DATA_WIDTH/8>   s_axis_tkeep,
5      ap_uint<C_S_AXIS_TUSER_WIDTH>    s_axis_tuser,
6      ap_uint<1>                      s_axis_tvalid,
7      ap_uint<1>                     &s_axis_tready,
8      ap_uint<1>                      s_axis_tlast,
9
10     //Master Stream Ports
```

```
    ap_uint<C_M_AXIS_DATA_WIDTH>     &m_axis_tdata,
    ap_uint<C_M_AXIS_DATA_WIDTH/8>   &m_axis_tkeep,
    ap_uint<C_M_AXIS_TUSER_WIDTH>    &m_axis_tuser,
    ap_uint<1>                      &m_axis_tvalid,
    ap_uint<1>                       m_axis_tready,
    ap_uint<1>                       &m_axis_tlast,

    ap_uint<1>                         reset
);
```

Packets going in through SLAVE stream ports of the module with respect to the AXI stream protocol (**s_axis_tvalid** and **s_axis_tready** are high). It's content are stored within **Data_queue**, at the same time based on the information about **SOURCE_MAC**, **DEST_MAC** and **SOURCE_PORT** the module performs searching for destination output queue as well as learning that the packet with **SOURCE_MAC** comes from **SOURCE_PORT**, after that when a handshake occurs at MASTER interface (**m_axis_tvalid** and **m_axis_tready** are high) the packet will be transferred through MASTER stream ports, with its destination output queue number assigned the appropriate position in field TUSER.

For detail of the implementation refer to this link: HLS implementaion

4 RESULT

4.1 Packet preparation

In HLS test bench is written in C++, packets are represented as 2D arrays where each field has different data size, for instance as shown below is the 64 bit unsigned integer representation of Data field if hexadecimal format:

```
89
90    uint64_t datain[16][4]={
91            //                                                |   SOURCEMAC ||  DESTMAC |
92            {0x9999999999999990,0x1777777777703040,0x0012345099987654,0x3210304010203040},//Packet 1
93            {0x0000300010003040,0x1020304010203040,0x1020304010203040,0x1020304010203040},
94            {0x1900304010203040,0x1020304010203040,0x1020304010203040,0x1020304010203040},
95            {0x0000300010003040,0x1020304010203040,0x1020304010203040,0x1020304010203040},
96            {0x0000300010003940,0x1000304010203040,0x1020304010203040,0x1020304010203040},
97            {0x0000000000000040,0x1020304010203040,0x1020304010203040,0x1020304010203040},//End
98
99            //                                                |   SOURCEMAC ||  DESTMAC |
100           {0x3040777030409999,0x3333344444444420,0x1020304077841053,0x9090999876543210},//Packet 2
101           {0x0000300010003040,0x1020304010203040,0x1020304010203040,0x1020304010203040},
102           {0x1900304010203040,0x1020304010203040,0x1020304010203040,0x1020304010203040},
103           {0x8888888888888888,0x1020304010203040,0x1020304010203040,0x1020304010203040},//End
104
105           //                                                |   SOURCEMAC ||  DESTMAC |
106           {0x3040777030409999,0x3333344444444420,0x1020304010012345,0x6789778410539090},//Packet 3
107           {0x0000300010003040,0x1020304010203040,0x1020304010203040,0x1020304010203040},
108           {0x1900304010203040,0x1020304010203040,0x1020304010203040,0x1020304010203040},
109           {0x8888888888888888,0x1020304010203040,0x1020304010203040,0x1020304010203040},
110           {0x0000300010003940,0x1000304010203040,0x1020304010203040,0x1020304010203040},
111           {0x0000000000000040,0x1020304010203040,0x1020304010203040,0x1020304010203040} //End
112   };
```

Data field's format of packet in C++

In this case three packets are prepared to be loaded into Output port lookup module with their **DEST_MAC** and **SOURCE_MAC** occupying the first 48 bits and the second 48 bit respectively of the first 256 bits of data field of each packet. To show how the module can direct packet to correct destination port, **SOURCE_MAC** field of the first packet will match with **DEST_MAC** field of the second packet and **SOURCE_MAC** field of the second packet will

match with **DEST_MAC** field of the third packet as highlighted in the figure.

Information about Source port queue or Destination queue can be obtained in the TUSER field in the positions highlighted in the following figure:

```
115    uint64_t tuserin[16][2]={
116            //                                  D||S
117             {0x1000044010053040,0x1020304010013040},
118             {0x0000000010000000,0x1020304010203040},
119             {0x0000004010023040,0x1020304010203040},
120             {0x1000044010023040,0x1020304010203040},
121             {0x1000044010023040,0x1020304010203040},
122             {0x9000044010023040,0x1020304010203040},
123
124             //                                  D||S
125             {0x1000044010033040,0x1020304010043040},
126             {0x0000000010000000,0x1020304010203040},
127             {0x0000004010023040,0x1020304010203040},
128             {0x1000044010023040,0x1020304010203040},
129
130             //                                  D||S
131             {0x1000044010033040,0x1020304010203040},
132             {0x0000000010000000,0x1020304010203040},
133             {0x0000004010023040,0x1020304010203040},
134             {0x1000044010023040,0x1020304010203040},
135             {0x0000004010023040,0x1020304010203040},
136             {0x1000044010023040,0x1020304010203040}};
```

TUSER field's format of packet in C++

As shown in the figure, TUSER field is also represented as a 2D array of 64 bit unsigned integer data type, the red highlighted region is the range in which one hot encoded address of source port is stored, this is where the lookup table will refer to during learning process. The blue highlighted region is where the destination port address will be assigned.

4.2 Wave form

The testing strategy: 3 Packets are loaded into the module with the bandwith of 417 bits per 10 nano seconds without transferring packets to a slave module until it's full, then enable the module to release old packets and while loading in new packets at the same time.

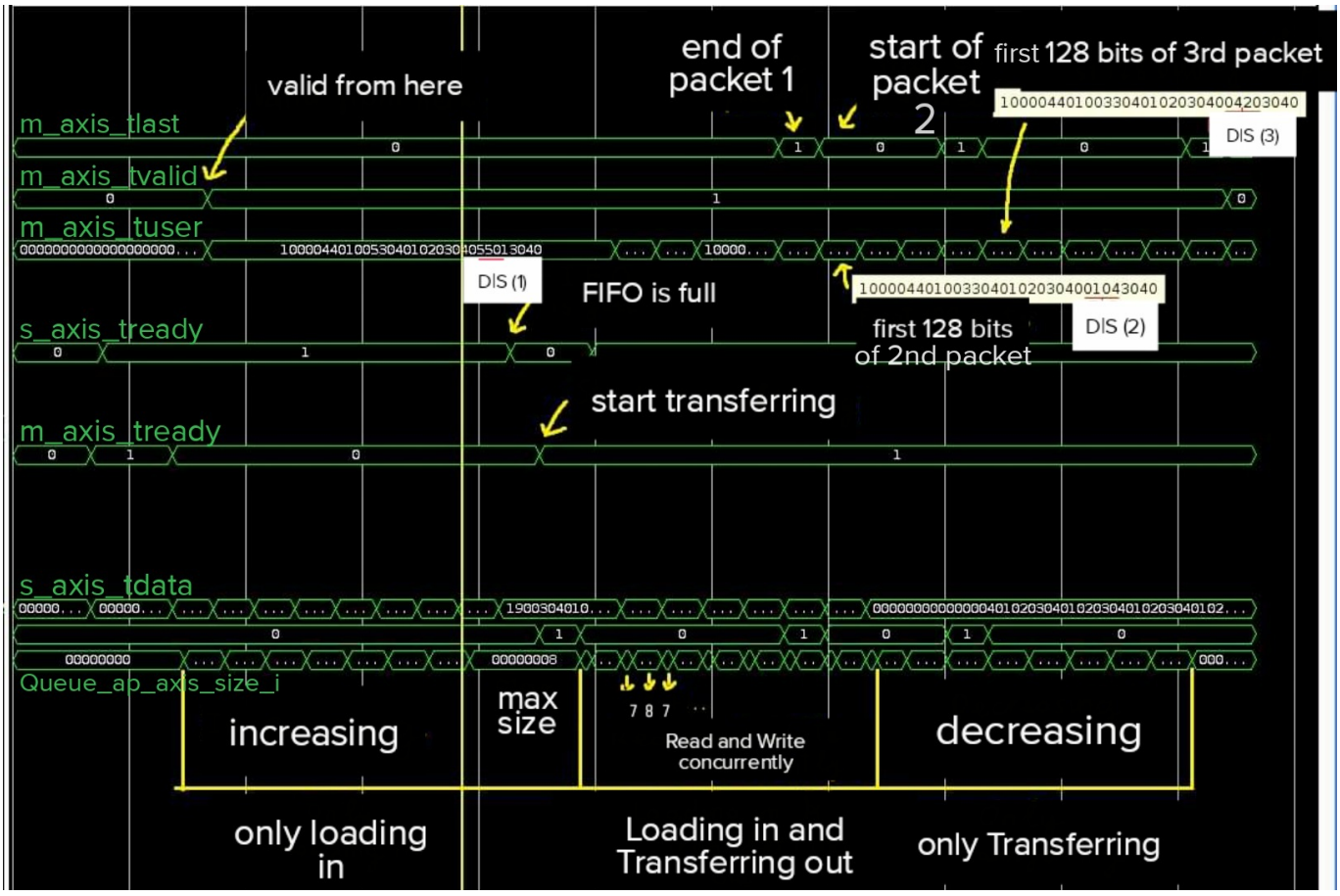

Figure 13: Waveform of processing 3 packets

Wave form analysis:

When the first packet leaves the module, its **Dest port** address is

unknown, hence the default missing value of 0x55 is assigned to **Dest port** position in TUSER field of packet 1 (**D** part of 1), however, the **Source Mac** and **Source port** of packet 1 has been learned by the module. The second packet has its **Dest Mac** is the **Dest Mac** of the first packet, this is why the **Dest port** address is the **Source port** address of the first packet, 0x01 to be specific (shown in **D** part of 2). The **Dest Mac** of packet 3 is the **Source Mac** of packet 2 hence **Dest port** address is 0x04, **Source port** of packet 2. The **m_axis_tvalid** signal turns low after transferring all 3 packets.

Technical parameters

- Clock cycle: 10 ns.
- Bandwidth: $4.17 * 10^{11}$ bits per second.
- Latency: 6 clock cycles both min and max.

5 CONCLUSION AND FUTURE PLAN

5.1 Finished Part

Finish implementing output port lookup module using C++ on Vivado HLS. Observe and analyze output signals, waveform, export IP.

Objectives covered: Learn principles of the switch, practice working with Xilinx Vivado HLS tool and implement output port lookup module.

5.2 Unfinished Part/Future Plan

Integrate IP into the reference_switch project on Vivado and finish the rest of the project. Can possibly implement other blocks of the switch such as input arbiter or output queue.

6 APPENDIX

6.1 Weekly Progress

Week	Progress
1	Install and practice working with Xilinx Vivado HLS.
2	Study AXI Stream, packets and principles of the Switch.
3	Study NetFPGA-SUME Reference Learning Switch on NetFPGA-SUME-public GitHub repository's wiki.
4	Study and practice using HLS Pragmas Library.
5	Study standard Ethernet frame format.
6	Study principles of pipeline, unroll, dependency and data flow.
7	Read reference project and illustrate Output port lookup module's structure.
8	Implement LUT block.
9	Implement Ethernet parser and Fifo block.
10, 11, 12	Complete implementation, export IP and synthesize with Switch project.

6.2 Software Requirement

- Download the Installer of Vivado HLx edition at Xilinx Downloads site.
- Register NetFPGA-SUME Beta Program at NetFPGA site.
- Open Download folder in Terminal on Linux and run these 2 commands:

```
chmod +x \
Xilinx_Vivado_SDK_Web_2018.2_0614_1954_Lin64.bin

sudo \
./Xilinx_Vivado_SDK_Web_2018.2_0614_1954_Lin64.bin
```

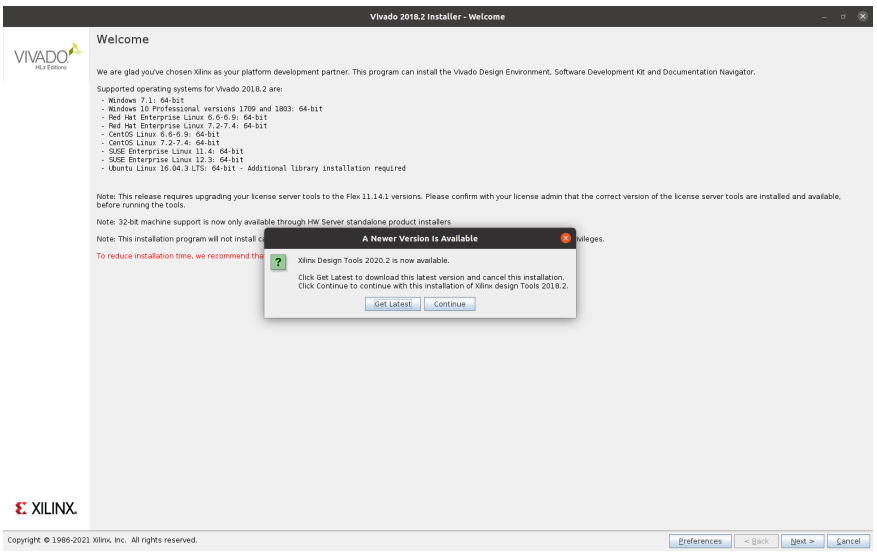

Figure 14

- Login with Xilinx account, accept License Agreement and continue installing.

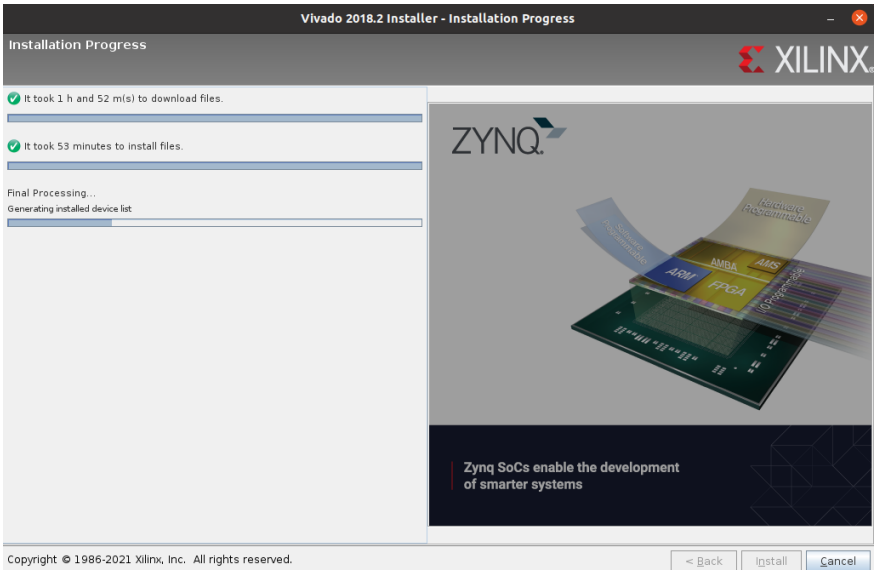

Figure 15

- At Final Processing step, it will be freezed intentionally. Click on Cancel but select the option that keeps all downloaded files, so that we will resume the rest of the installation later. In Terminal, run these commands and continue installing:

```
sudo apt-get install libncurses5
```

```
sudo \
./Xilinx_Vivado_SDK_Web_2018.2_0614_1954_Lin64.bin
```

- After the download completes, save the Certificate-Based License (.html file), open and install the License. Go back to the Vivado License Manager, click on Load License and select the file just downloaded.
- Open Terminal at path Xilinx/Vivado/2018.2 and run command to finish installation:

```
source /opt/Xilinx/Vivado/2018.2/settings64.sh
```

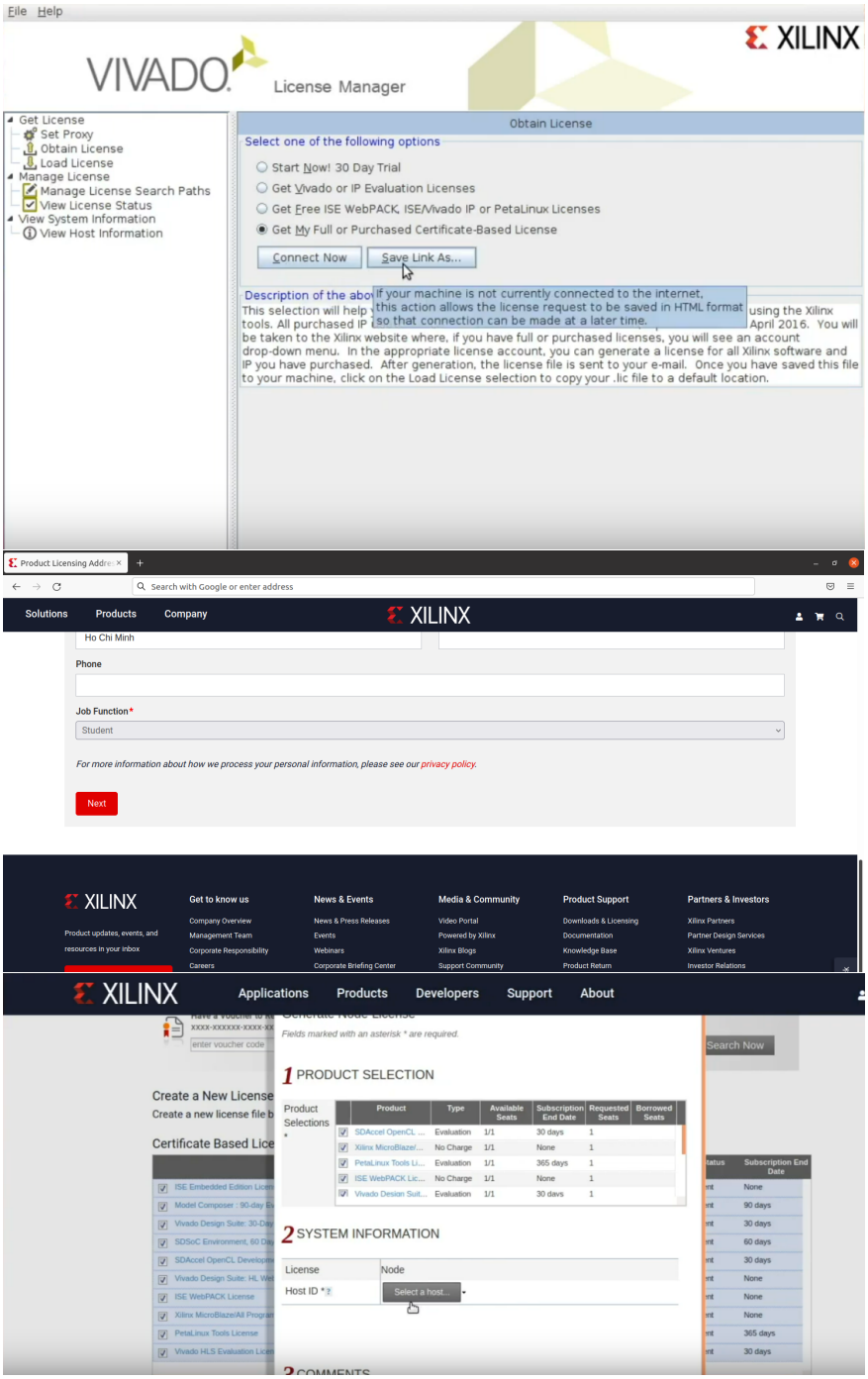

Figure 16, 17, 18

- To open Vivado HLS, run this command:

```
vivado_hls
```

References

[1] Xilinx. "**UG902 Vivado Design Suite User Guide: High-Level Synthesis**". July 25, 2012.

[2] Xilinx. "**UG871 Vivado Design Suite Tutorial: High-Level Synthesis**". May 6, 2014.

[3] Xilinx. "**PG157 10 Gigabit Ethernet Subsystem v3.1: Product Guide**". February 4, 2021.

[4] Arm Limited or its affiliates. "**IHI0051B AMBA AXI-Stream protocol specification**". 2010.

[5] NetFPGA. "**NetFPGA-SUME public GitHub repository-'s wiki**". 2015. [Online] Available: https://github.com/NetFPGA/NetFPGA-SUME-public/wiki

[6] NetFPGA. "**NetFPGA-SUME live GitHub repository**". 2015. [Online] Available: https://github.com/NetFPGA/NetFPGA-SUME-live

[7] Xilinx. "**HLS Pragmas**". June 19, 2019. [Online] Available: https://www.xilinx.com/html_docs/xilinx2019_1/sdaccel_doc/hls-pragmas-okr1504034364623.html

[8] Homenet Howto. "**Switches**". [Online] Available: https://www.homenethowto.com/switching/switches/

[9] Network Encyclopedia. "**Network Packet**". [Online] Available: https://networkencyclopedia.com/network-packet/