Ethernet Packet Detection

Hardware Implementation and Verification

Ahmed Mustafa.

ahmedmustafa0100@gmail.com

***Abstract*:** All network devices must parse packet headers to decide how packets should be processed. An Ethernet switch must parse several billions packets per second to extract fields used in forwarding decisions.

Introduction :

Despite their variety, every network device examines fields in the packet (see Figure 0) headers to decide what to do with each packet. For example, a router examines the IP destination address to decide where to send the packet next, and a firewall compares several fields against an access-control list to decide whether to drop a packet.

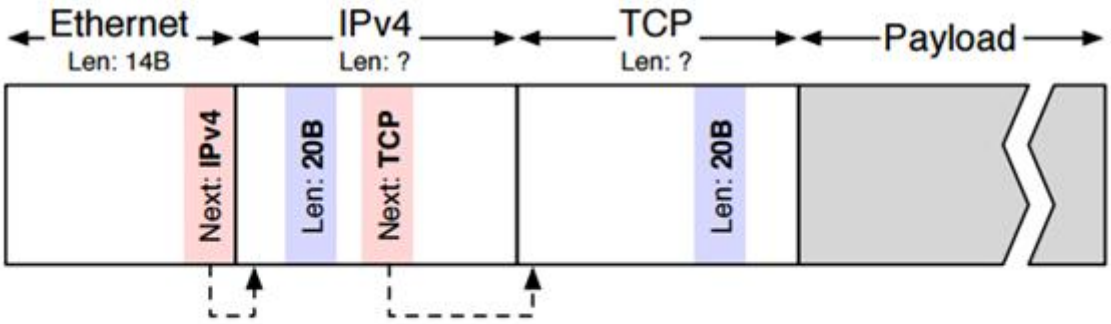

Figure 0: A TCP packet.

In practice, packets often contain many more headers. These extra headers carry information about higher level protocols (e.g., HTTP headers). It is common for a packet to have eight or more different packet headers during its lifetime.

The process of identifying and extracting the appropriate fields in a packet header is called parsing and is the subject of this paper.

To parse a packet, a network device has to identify the headers in sequence before extracting and processing specific fields. A packet parser knows a priori which header types to expect.

A packet detector (also known as a network analyzer or an Ethernet sniffer) is a piece of hardware that can intercept the traffic that passes over a digital network , decodes the packet's raw data, showing the values of various fields in the packet, and analyzes its content .

The Aim of this paper is to document the implementation of an Ethernet Packet Detector that parses the incoming stream in search for an Ethernet packet having specific fields values (see Figure 1).

Preamble (8 bytes)	Dst Address (6 bytes)	Src Address (6 bytes)	Type/Length (2 bytes)	Data (64 – 1500 bytes)	CRC (4 bytes)

Figure 1: Ethernet Packet format

Designed Block Description:

The following pin-out diagram (Figure 2) shows the different input/output of the Ethernet Packet Detector. Subsequent tables describe those I/Os

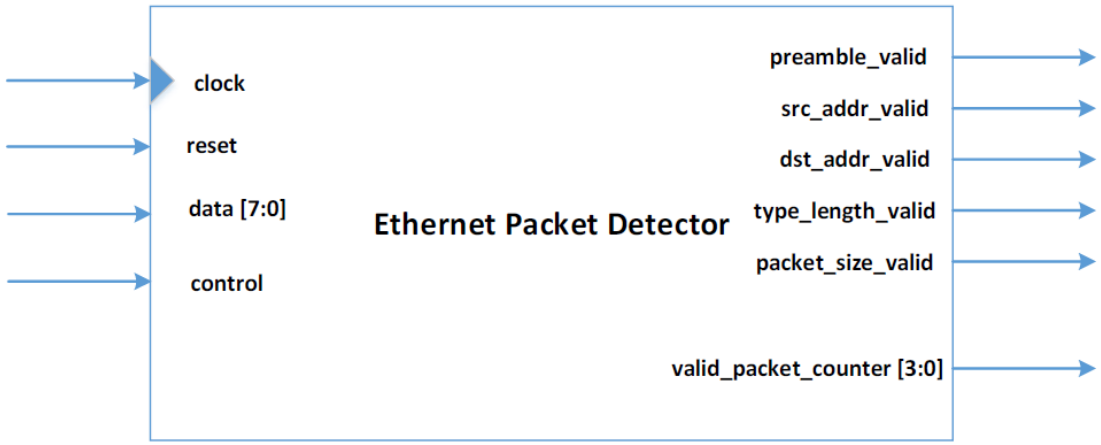

Figure 2: Pin-out Diagram

a. Inputs Description

Table 1: Input Pins

Input Name	Description
clock	Reference clock
reset	Synchronous reset signal (active low or high as you prefer)
data[7:0]	Incoming stream (1 byte per clock cycle). It can contain packet data or IFG. • Ethernet Packet data: **data[7:0]** = any value & **control** = 1 • IFG: **data[7:0]** = 0x00 & **control** = 0
control	A bit that indicates whether data[7:0] represents IFGs or Packet data • Ethernet Packet data: **control** = 1 • IFG: **control** = 0

b. Outputs Description

Table 2: Output Pins

Output Name	Description
preamble_valid	Asserted to 1 after receiving the correct preamble value (Please refer to subsection c below). It should stay asserted for one clock cycle only.
src_addr_valid	Asserted to 1 after receiving the correct source address value (Please refer to subsection c below). It should stay asserted for one clock cycle only.
dst_addr_valid	Asserted to 1 after receiving the correct destination address value (Please refer to subsection c below). It should stay asserted for one clock cycle only.
type_length_valid	Asserted to 1 after receiving the correct type/length value (Please refer to subsection c below). It should stay asserted for one clock cycle only.
packet_size_valid	Asserted to 1 after receiving a packet with size = 72 bytes (Please refer to subsection c below). It should stay asserted for one clock cycle only.
valid_packet_counter [3:0]	Hold the count of valid packets (Please refer to subsection c below for the definition of valid packets)

Validation Criteria:

This section describes the values of the fields of the specific Ethernet Packet that, when detected, the valid_packet_counter must be incremented. Also several output valid signals, as was shown in Table 2 are supposed to get asserted as the parsing process is ongoing.

1- Valid Preamble: A valid preamble is 8 bytes with the values and order of transmission/reception shown in in Figure 3. After a valid preamble is detected, preamble_valid output must be asserted for the duration of one clock cycle.

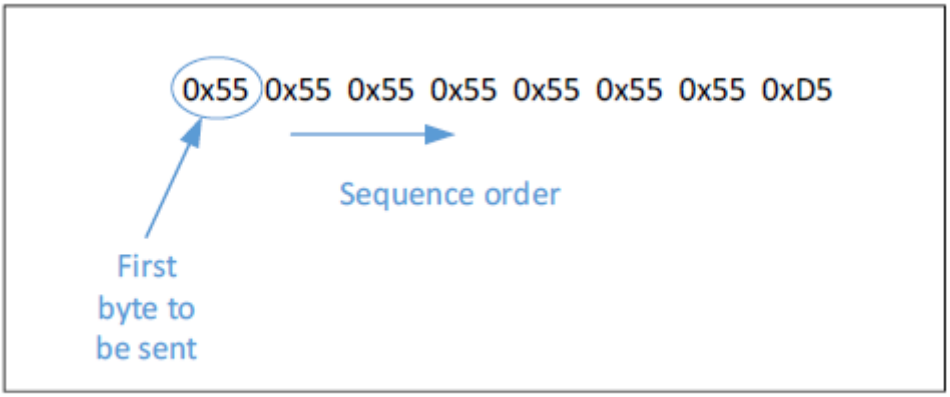

Figure 3: Valid Preamble

2- Destination Address: A valid destination address is 6 bytes with the values and order of transmission/reception as shown in Figure 4. After a valid preamble is detected ,dst_addr_valid output must be asserted for the duration of one clock cycle

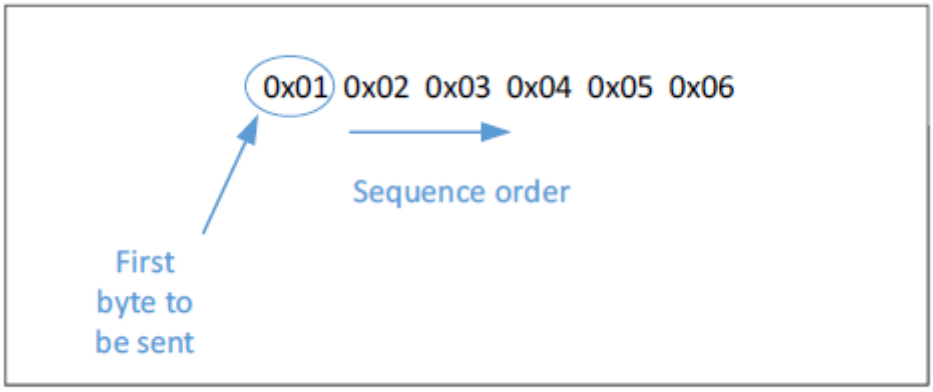

Figure 4: Valid Destination Address

3- Source Address: A valid destination address is 6 bytes with the values and order of transmission/reception shown in Figure 5. After a valid preamble is detected, src_addr_valid output must be asserted for the duration of one clock cycle.

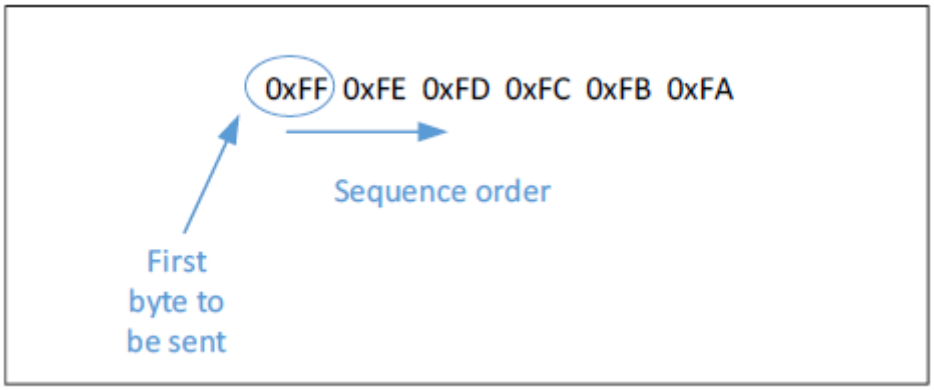

Figure 5: Valid Source Address

4- Type/Length: A valid type/length is 2 bytes with the values and order of transmission/reception shown in Figure 6. After a valid type/length is detected, type_length_valid output must be asserted for the duration of one clock cycle.

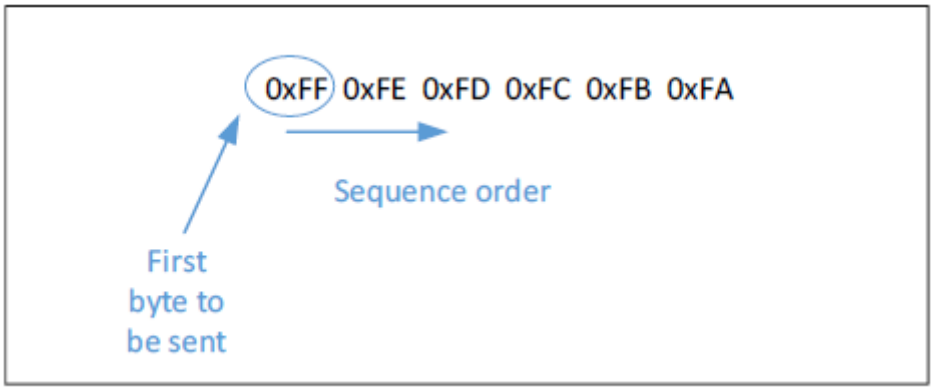

Figure 5: Valid Source Address

5- Packet Size: After the Type/Length field is received, it is expected to receive 50 bytes of data of any values (no validation is required for payload and/or CRC) then receive one or more IFGs (where data [7:0] == 0x00 & control = 0).
In total, for this assignment, the received packet size is expected to be 72 bytes (preamble + destination address + source address + type/length field + payload + CRC).packet_size_valid will be asserted after detecting an Ethernet packet with valid 72 bytes size. This signal must be asserted for the duration of one clock cycle.

The valid_packet_counter will be incremented by one if the received packet is valid according to the five criteria listed above.

The Detector's Hardware Implementation:

-The detector was implemented on a Field Programmable Gate Array (Xilinx: Spartan6) using Verilog hardware description language.

-The main detection mechanism depends upon a Controller which is a Moore's Finite State Machine and a Datapath which consists of a multi-purpose counter and simple data comparators.

-The Detector's Block Diagram is shown in Figure 6:

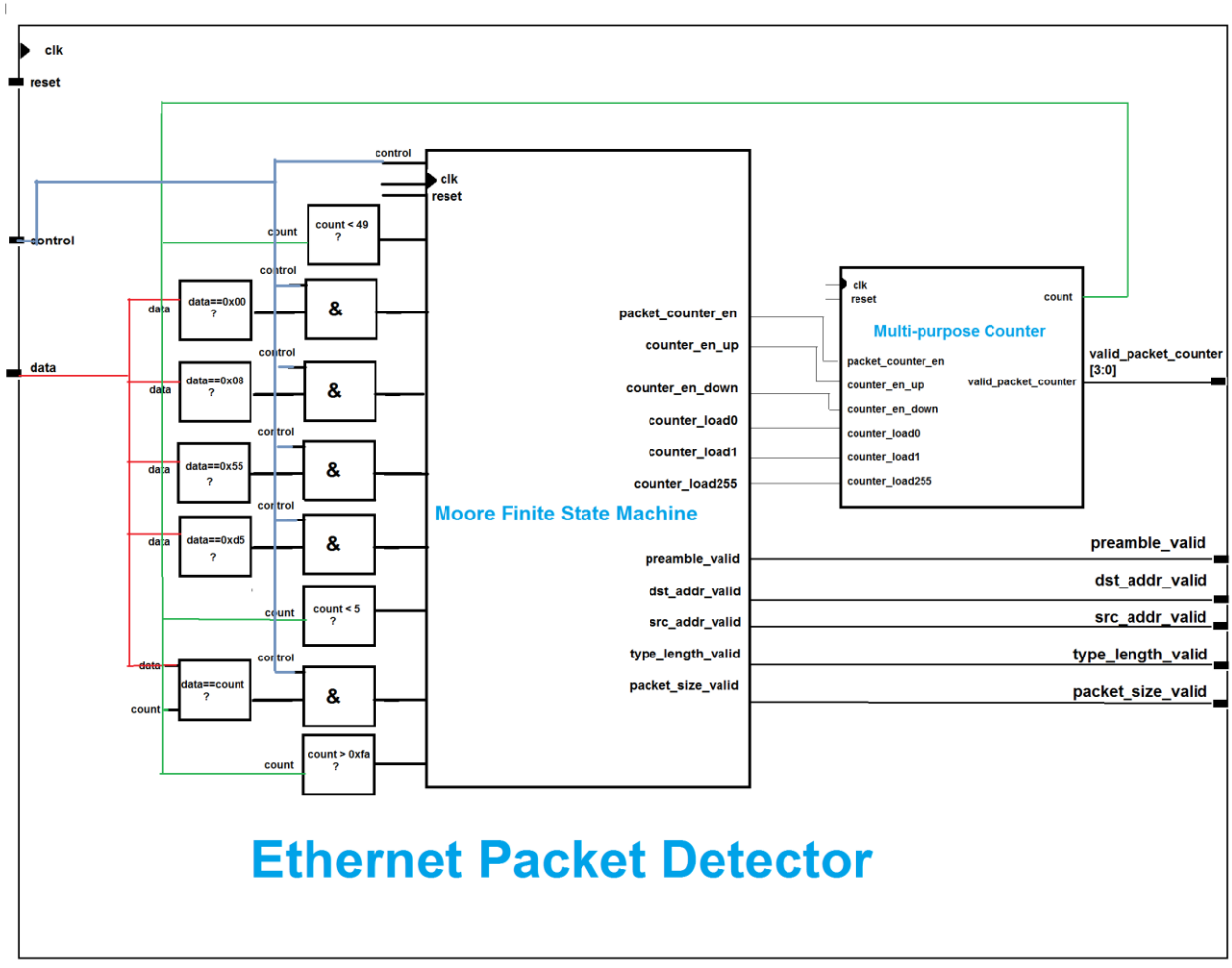

Figure 6: Detector's Hardware Implementation

-The Detector's Controller (Moore Finite State Machine) is shown in Figure 7:

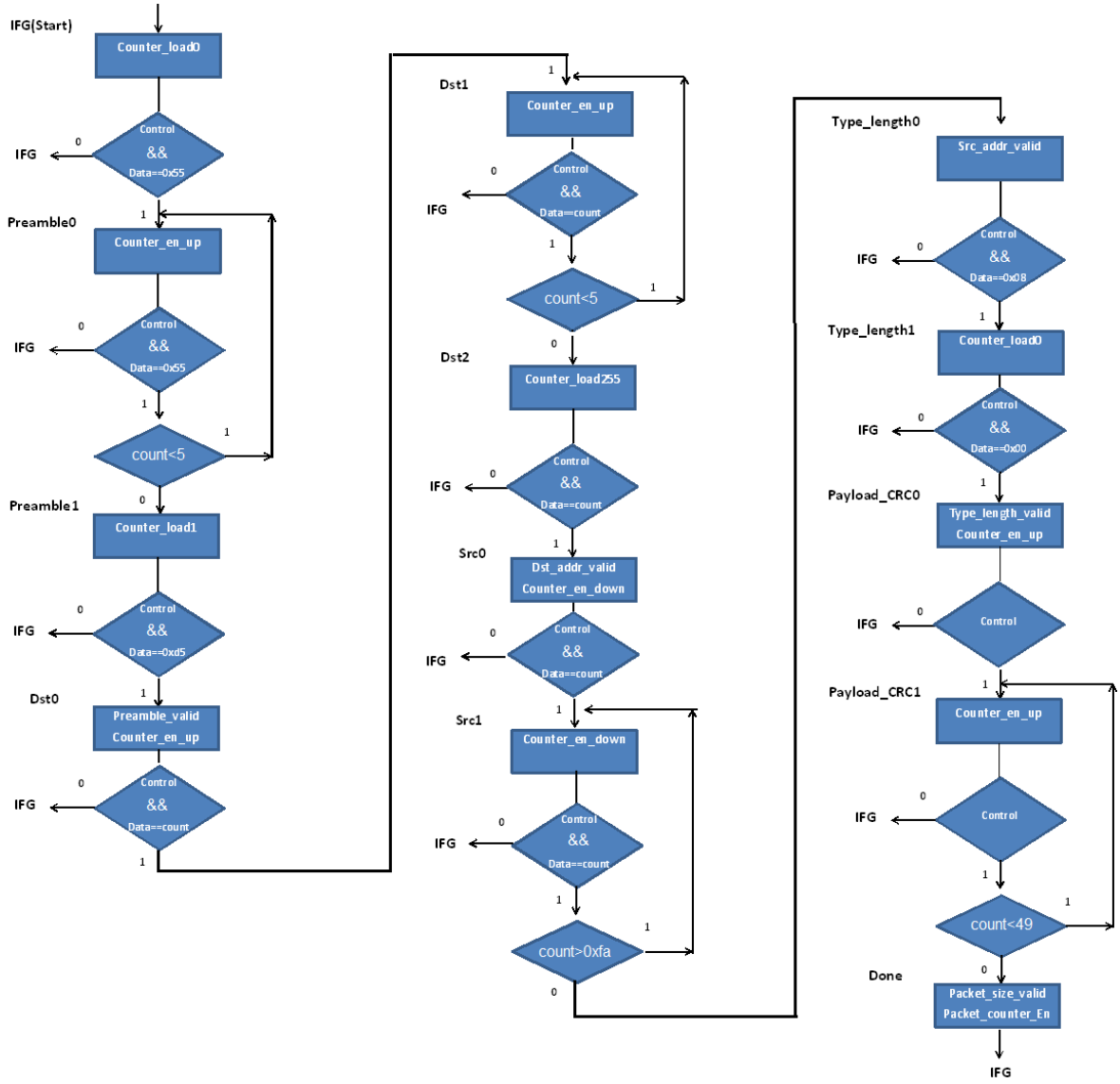

Figure 7: Detector's FSM

-The Datapath's Detailed Working:

The datapath mainly consists of a multi-purpose counter which is used in the different detection stages of the Ethernet packet.

The counter inputs and outputs description:

clk: Circuit's Clock.

reset: Circuit global reset.

valid_packet_counter[3:0]: Output register which holds the number of valid packets received.

count [7:0]: Output register which will be used mainly in the different stages of the detection process .

packet_counter_en: Increments the register that holds the number of the valid packets received (valid_packets_counter).

counter_en_up: Increments the (count) register.

counter_en_down: Decrements the (count) register.

counter_load0: Loads the (count) register with (0) for design purposes.

counter_load1: Loads the (count) register with (1) for design purposes.

counter_load255: Loads the (count) register with (0xff) for design purposes.

At the beginning of the packet detection while sniffing for the starting byte from the PREAMBLE field, the (count) register is loaded with (0) (using signal counter_load0) and when the detection process starts the (count) value is incremented (using signal counter_en_up) to count the predefined number of the PREAMBLE field bytes (which is for this design is 7 bytes 0x55 and 1 byte 0xd5) and then the (count) register is loaded with (1) (using signal counter_load1) to be compared with the first byte received from the DST ADDR field (which is here 0x01) and then the (count) value is incremented (using signal counter_en_up) to be compared with the remaining received bytes from the DST ADDR field (0x02 to 0x06) and then the (count) register is loaded with (0xff) (using signal counter_load255) to be compared with the first byte received from the SRC ADDR field (which is 0xff) and then the (count)value is decremented (using signal counter_en_down) to be compared with the remaining received bytes from the SRC ADDR field (0xfe to 0xfa) and then the (count) register is loaded with (0) (using signal counter_load0)to be used in counting the received PAYLOAD/CRC field (which is here 50 bytes) and finally it increments (using signal packet_counter_en) the (valid_packet_counter) after receiving the whole packet.

The Controller's (FSM) Detailed Working:

The design is implemented with a Moore's Algorithm State Machine that has 13 states and the working of each state will be explained individually:

1- IFG (Inter Frame Gap) state: It's the initial state and in that state the detector sniffs for a valid starting byte (0x55) from the PREAMBLE header and also if any unexpected byte was received the state machine returns to that state.

(In that state we load (0) in the (count) register using signal counter_load0 to be used later in the detection process).

2- Preamble0 state: In that state we already have received the first byte (0x55) from the PREAMBLE field and there remain (7) bytes (6 bytes 0x55 and 1 byte 0xd5) to be received. In that state we detect the remaining 6 bytes (0x55) using the (count) value as a loop termination variable when it reaches 5.

(In that state the detector increments the (count) register using the signal counter_en_up).

3- Preamble1 state: In that state the detector sniffs for the last byte from the PREAMBLE field (0xd5).

(In that state the detector loads the (count) register with (1) to be used in the next detection stage).

4- Dst0 state: In that state the detector sniffs for the starting byte from the DST ADDR field (0x01) using the value in the (count) register to be compared with the received byte .

(In that state the preamble_valid flag is asserted to indicate the successful receipt of the PREAMBLE field and also the value of the (count) register is incremented using the signal counter_en_up).

5- Dst1 state: In that state the detector has already received the first byte from the DST ADDR field and there remain (5) bytes to be received (0x02 to 0x06). In that state the detector receives only (4) bytes (0x02 to 0x05) using the (count) value to be compared with the received bytes. The last byte is detected in a different state for the reason that with the last byte the detector loads the (count) register with (0xff) to be used in the next detection stage.

(The reason why the first byte from the DST ADDR field was received in a different state is that in that state we will assert the Preamble_valid flag and it must be asserted for only one clock cycle).

(In that state the (count) value is incremented using the signal counter_en_up).

6- Dst2 state: In that state the detector sniffs for the last byte from the DST ADDR field (0x06).

(In that state the detector loads the (count) register with 0xff using the signal counter_load255 to be used in the next detection stage).

7- Src0 state: In that state the detector sniffs for the starting byte from the SRC ADDR field (0xff) using the (count) value to be compared with received byte.

(In that state the detectors asserts the dst_addr_valid flag to indicate the successful receipt of the DST ADDR field and also it decrements the (count) value using the signal counter_en_down).

8- Src1 state: In that state the detector has already received the first byte (0xff) from the SRC ADDR field and there remain (5) bytes (0xfe to 0xfa) to be received using the (count) value to be compared with the received bytes.

(In that state the detector decrements the (count) register using counter_en_down).

9- Type_length0 state: In that state the detector sniffs for the starting byte from the TYPE/LENGTH field (0x08).

(In that state the detector asserts the src_addr_valid flag to indicate the successful receipt of the SRC ADDR field).

10- Type_length1: In that state the detector sniffs for the last byte from the TYPE/LENGTH field (0x00).

(In that state the detector loads the (count) register with (0) using signal counter_load0 to be used in the detection of the next field).

11- Payload_CRC0 state: In that state the detector receives the starting byte from the PAYLOAD field.

(In that state the detector asserts the type_length_valid flag to indicate the successful receipt of the TYPE/LENGTH field and also it increments the (count) value using signal counter_en_up).

12- Payload_CRC1: In that state the detector receives the last 49 bytes from the PAYLOAD/CRC field.

(In that state the detector increments the (count) value using signal counter_en_up).

13- Done state: In that state the detector has finished receiving the Ethernet Packet and expects one or more IFG packets.

(In that state the detector asserts the packet_size_valid flag and also increments the (valid_packet_counter) using signal packet_counter_en).

Simulation Results:

-Valid packet (Figure 8-1) & (Figure 8-2):

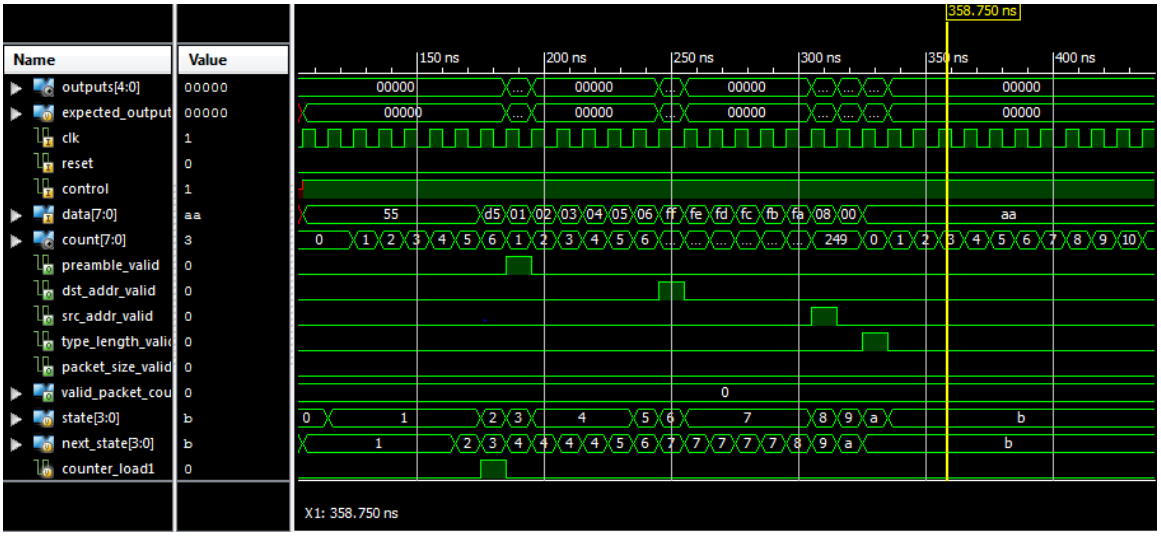

Figure 8-1: Valid packet

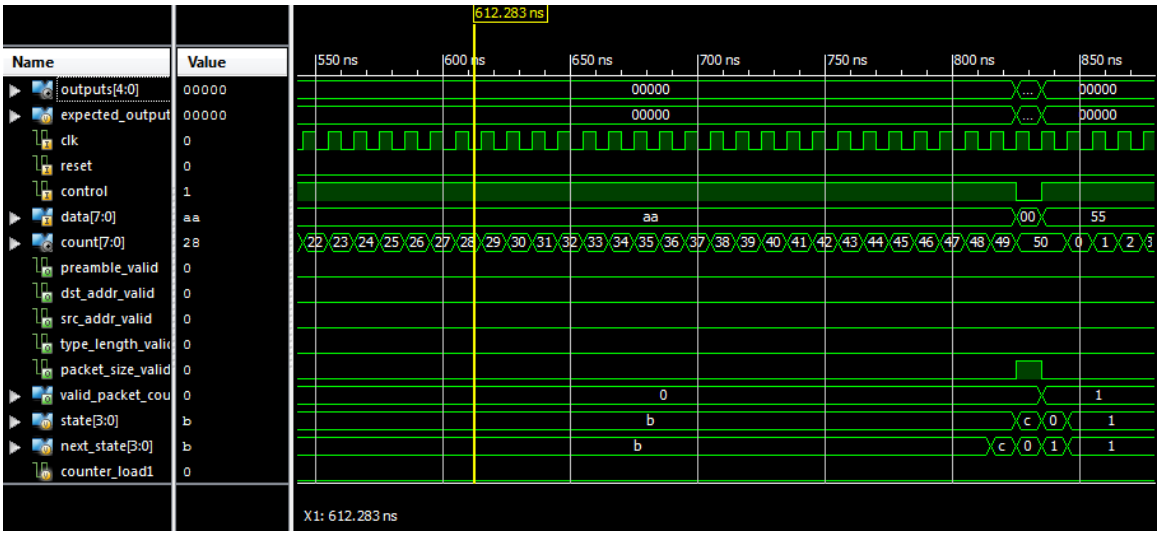

Figure 8-2: Valid packet

-Invalid packet (Figure 9-1) & (Figure 9-2):

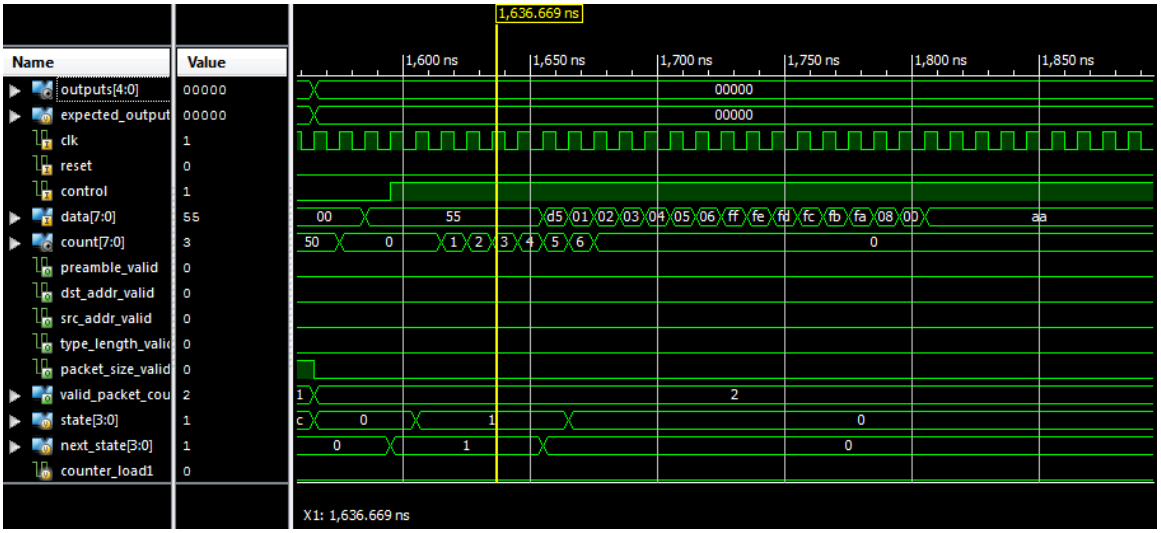

Figure 9-1: Invalid packet

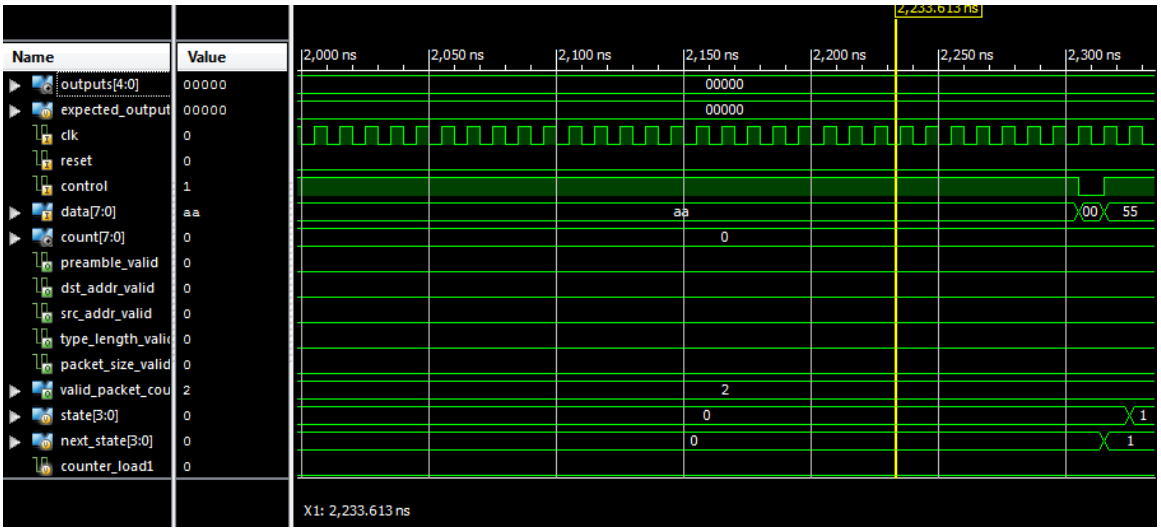

Figure 9-2: Invalid packet

Implementation Results:

-RTL schematic (Figure 10):

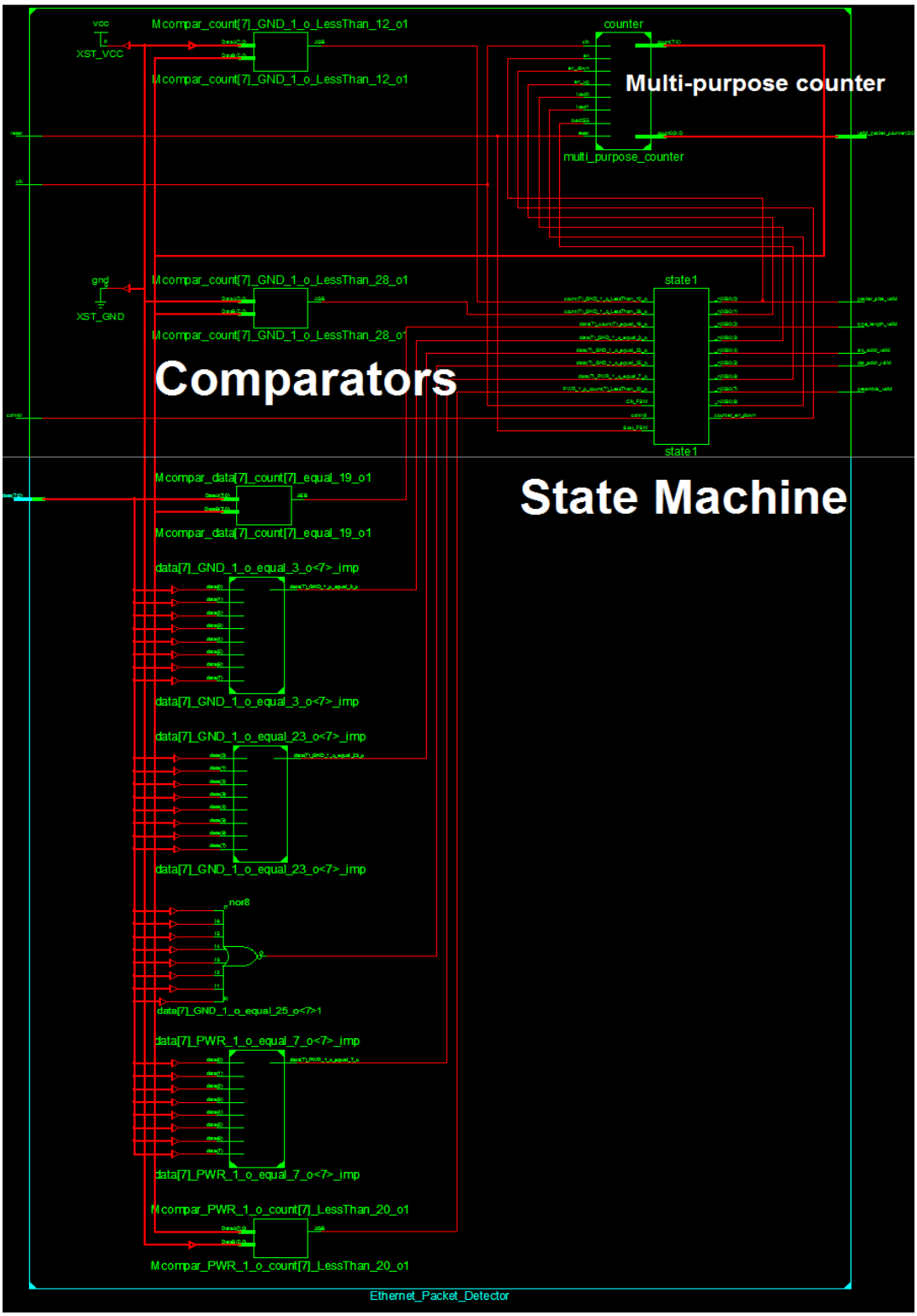

Figure 10: RTL schematic

-Synthesis Results (Figure 11-1) & (Figure 11-2) & (Figure 11-3):

```
Synthesizing Unit <Ethernet_Packet_Detector>.
    Related source file is "D:\EthernetPacketDetector\Ethernet_Packet_Detector.v".
        IFG = 0
        Preamble0 = 1
        Preamble1 = 2
        Dst0 = 3
        Dst1 = 4
        Dst2 = 5
        Src0 = 6
        Src1 = 7
        Type_length0 = 8
        Type_length1 = 9
        Payload_CRC0 = 10
        Payload_CRC1 = 11
        Done = 12
    Found 4-bit register for signal <state>.
    Found finite state machine <FSM_0> for signal <state>.
    -----------------------------------------------------------------------
    | States             | 13                                             |
    | Transitions        | 39                                             |
    | Inputs             | 9                                              |
    | Outputs            | 10                                             |
    | Clock              | clk (rising_edge)                              |
    | Reset              | reset (positive)                               |
    | Reset type         | synchronous                                    |
    | Reset State        | 0000                                           |
    | Encoding           | auto                                           |
    | Implementation     | LUT                                            |
```

Figure 11-1: State machine

```
=========================================================================
Advanced HDL Synthesis Report

Macro Statistics
# Adders/Subtractors                                   : 1
 8-bit addsub                                         : 1
# Counters                                             : 1
 4-bit up counter                                     : 1
# Registers                                            : 8
 Flip-Flops                                           : 8
# Comparators                                          : 4
 8-bit comparator equal                               : 1
 8-bit comparator greater                             : 3
# Multiplexers                                         : 2
 8-bit 2-to-1 multiplexer                             : 2
# FSMs                                                 : 1

=========================================================================
```

Figure 11-2: HDL synthesis report

```
Device utilization summary:
---------------------------

Selected Device : 6slx9tqg144-2


Slice Logic Utilization:
 Number of Slice Registers:              26  out of  11440     0%
 Number of Slice LUTs:                   78  out of   5720     1%
    Number used as Logic:                78  out of   5720     1%

Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:     85
   Number with an unused Flip Flop:      59  out of     85    69%
   Number with an unused LUT:             7  out of     85     8%
   Number of fully used LUT-FF pairs:    19  out of     85    22%
   Number of unique control sets:         3

IO Utilization:
 Number of IOs:                          20
 Number of bonded IOBs:                  20  out of    102    19%

Specific Feature Utilization:
 Number of BUFG/BUFGCTRLs:               1  out of     16     6%
```

Figure 11-3 Device utilization summary

-Timing Results (Figure 12):

```
Timing summary:
---------------

Timing errors: 0  Score: 0  (Setup/Max: 0, Hold: 0)

Constraints cover 822 paths, 0 nets, and 334 connections

Design statistics:
   Minimum period:   3.968ns{1}   (Maximum frequency: 252.016MHz)

================================================================================
Timing constraint: TS_clk = PERIOD TIMEGRP "clk_group" 230 MHz HIGH 50%;
For more information, see Period Analysis in the Timing Closure User Guide (UG612).
 822 paths analyzed, 84 endpoints analyzed, 0 failing endpoints
 0 timing errors detected. (0 setup errors, 0 hold errors, 0 component switching limit errors)
 Minimum period is   3.968ns.
--------------------------------------------------------------------------------
```

Figure 12: Timing Results

Ethernet Packet Detector

Verification Strategy:

The Detector's verification process depends upon a Verilog testbench which uses stimulus Ethernet packets both valid and invalid through loading them from an external text file (vectors.tv) to a local register file (vectors) and then it loops over that register file on every positive edge of the global clock and through an index (vector_num) it stimulates the UUT (Unit Under Test) with those test vectors with every clock.

The test vector format is: [Stimulus data + Expected outputs]

Stimulus data format:
Control bit _ Packet data [7:4](4bits) _ Packet data [3:0] (4 bit).
Expected outputs format:
[preamble_valid,dst_addr_valid,src_addr_valid,type_length_valid,packet_size_valid].

Verification process:
On every positive edge from the clock the test vector stimulus is loaded in the inputs of the Unit Under Test through the code line:
[control,data,expected_outputs}=vectors[vector_num];

And then the test bench waits 1ns for the outputs to be generated and then it displays the test vector's contents and both generated outputs and expected outputs to the terminal screen and also it writes them to an external file (outputs.txt), then it compares the generated outputs with the expected outputs and if there was a mismatch it displays a message on the terminal and also writes the message to the external file (outputs.txt).

There is a termination condition for the test bench which is loading an expected outputs value of 5'b11111 to the test environment.
The stimulus text file is shown in Figure 12:

```
//Test Vector format (Stimulus-Expected outputs)
//Stimulus format
//Control[0]_Data[7:4]_Data[3:0]

//Expected Outputs format
//[preamble_valid,dst_addr_valid,src_addr_valid,type_length_valid,packet_size_valid]

//(12 Ethernet packets -- 9 valid -- 3 invalid)

//First packet (valid)
//PREAMBLE field
1_0101_0101_00000
1_0101_0101_00000
1_0101_0101_00000
1_0101_0101_00000
1_0101_0101_00000
1_0101_0101_00000
1_0101_0101_00000
1_1101_0101_00000
//DST ADDR field
1_0000_0001_10000
1_0000_0010_00000
1_0000_0011_00000
1_0000_0100_00000
1_0000_0101_00000
1_0000_0110_00000
//SRC ADDR field
1_1111_1111_01000
1_1111_1110_00000
1_1111_1101_00000
1_1111_1100_00000
1_1111_1011_00000
1_1111_1010_00000
//TYPE/LENGTH field
1_0000_1000_00100
1_0000_0000_00000
//Payload field
1_1010_1010_00010
1_1010_1010_00000
1_1010_1010_00000
1_1010_1010_00000
```

Figure 12: Stimulus text file

-Verification Results:

The Unit Under Test was stimulated with 12 Ethernet packets (9 valid – 3 invalid) (Figure 13):

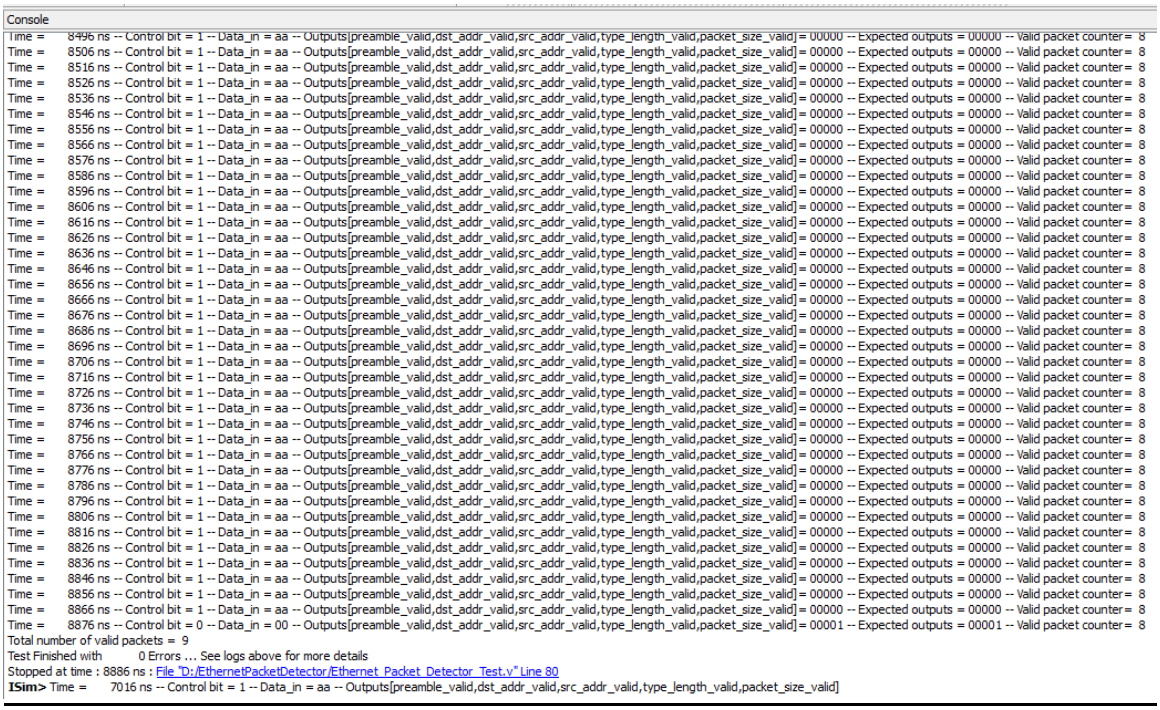

Figure 13: Terminal messages
