Ethernet Receiver Specification: Assignment 1 Report

Patrick Creighton - 70067053

ELEC 402 2023W1

Contents

1 Overview

This IP receives ethernet packets, outputting the source MAC address and payload of the ones that have a (1) destination MAC address matching its own and (2) correct frame check sequence.

An IEEE 802.3 ethernet packet has the following sections (adapted from Ethernet frame - Wikipedia):

Preamble	Start Frame Delimiter	MAC Destination	MAC Source	802.1Q Tag (Optional)	Length	Payload	Frame Check Sequence
7 bytes	1 byte	6 bytes	6 bytes	4 bytes	2 bytes	46-1500 bytes	4 bytes

This IP differs from the 802.3 specification in the following ways to simplify the design:

- **802.1Q tag:** this optional section is not supported.
- **Payload:** the lower and upper bounds are not enforced.
- **Frame check sequence:** use longitudinal redundancy check 4 times instead of 32-bit CRC.

1.1 Acronyms

Acronym	Meaning
FCS	Frame Check Sequence section of the ethernet packet
LRC	Longitudinal Redundancy Check
PL	Payload section of the ethernet packet
SFD	Start Frame Delimiter section of the ethernet packet

2 Components

2.1 Recv_top

This is the only module in the IP.

2.1.1 I/O

Name	I/O	Width	Description
clk	I	1	Clock signal
rst	I	1	Synchronous, active high reset signal
data	I	8	Byte of an ethernet packet
start	I	1	High when data is the first byte of an ethernet packet, else low
out	O	8	Byte of the output
vld	O	1	High while output is valid and meant to be read, else low
rdy	O	1	High when ready to receive a new ethernet packet, else low

An ethernet packet is streamed in one byte at a time. As per the ethernet specification, the most significant byte within each section comes first, and within each byte the least-significant bit comes first. The start of an ethernet packet is signified by the start signal and should only occur if the rdy signal is low. This causes the rdy signal to go low in the next cycle as the module processes the packet.

The output is streamed out one byte at a time, during which the vld signal is high. The output length varies based on the type of error (if any) and payload length. The last byte of the output will always be the return value: 0 if there was no error else {4'hF, last_state} if there was an error in state. If there was no error, the first 6 bytes will be the MAC address of the sender and the remaining will be the payload. After the output ends, the vld signal goes low and the rdy signal goes high.

Unexpected data will result in different behavior. If a byte of Preamble or SFD differs from the specification (8'b1010_1010 and 8'b1010_1011, respectively), an error will be returned. If the destination MAC address does not match the corresponding module parameter, packet processing stops and no output is returned. If a byte of FCS differs from the LRC, an error will be returned. Otherwise, 0 will be returned.

2.1.2 States

There is a state for each section of the ethernet packet, plus idle, success, and error states. The sections and their lengths can be found in Overview. A counter tracks the number of cycles the state has not changed to go to the next state when it is greater than or equal to the current section length.

State	Function
IDLE	Initial state (after reset)
PREAMBLE	No function in this receiver, but typically used for clock synchronization
SFD	Signal start of the actual ethernet frame
MACDST	Check that the receiver is the intended recipient
MACSRC	Output the transceiver's MAC address
PLLEN	Store the payload length in payload_length
PL	Output the payload
FCS	Detect corrupted data
SUCCESS	Indicate transaction succeeded
ERROR	Indicate transaction failed

2.1.3 Always Blocks

The logic of recv_top is organized into the following always blocks (sequential always blocks are indicated with an asterisk*):

- **stateCounter*:** computes state_counter, the number of cycles the state hasn't changed.
- **delayLogic*:** registers data and delays it by 6 cycles.
- **nextStateLogic:** determines next_state.
- **stateRegister*:** sets state to next_state and last_state to state.
- **payloadLengthLogic*:** payload_length is set during PLLEN: the upper byte in the first cycle and lower in the second. It is held during PL to determine when to go to FCS, then reset to 0.
- **lrcLogic*:** lrc is computed on the bytes in MACDST, MACSRC, PLLEN, and PL. Specifically, these bytes are summed together in this byte-wide signal during these states, then XORed with 8'hFF before being incremented to get the final LRC which is checked in FCS. Otherwise, it is 0.
- **outputLogic:** out is delayed 4 or 6 cycles from the registered input. This is so that the return value occurs after FCS. Specifically, the latter sections of the ethernet packet are MACSRC, PLLEN, PL, and FCS, but the output is only MACSRC and PL. Thus, the output is initially delayed

by the combined size of PLLEN and FCS, 6. After MACSRC is output, it is switched to a delay of 4 to skip over PLLEN. During this time, and for the return value in SUCCESS or ERROR, vld is high. rdy is only high during IDLE.

2.2 Testbench

The testbench was written in Python using the cocotb library. It uses ModelSim to simulate the design. Cocotb was chosen over the conventional SystemVerilog testbench because Python is fast to write, allowing for more comprehensive testing that is easier to understand.

2.2.1 Tests

There is one test for each of the return cases:

- **Test_recv_pass:** When there is no error, the correct source MAC and payload are returned.
- **Test_recv_wrong_preamble:** When a Preamble byte is wrong, return error in state 1.
- **Test_recv_wrong_sfd:** When a SFD byte is wrong, return error in state 2.
- **Test_recv_wrong_fcs:** When an FCS byte is wrong, return error in state 7.
- **Test_recv_wrong_macdst:** When the destination MAC address does not match the receiver, the packet should be ignored.

2.2.2 Testbench Architecture

Each test calls 3 functions: init, driver, and monitor:

- **Init:** starts the clock, toggles the rst signal, and gives the inputs initial values.
- **Driver:** drive the DUT inputs. But before doing so, it checks that the rdy signal is low. For each section, it parses the human-readable input value into bytes that are fed into the DUT one at a time. It also accepts flags as arguments to support feeding in the wrong value to a section.
- **Monitor:** waits for the vld signal to go high then reads the out signal until vld goes back low. It splits out into its components (source MAC address, payload, return value) and parses them into human-readable formats. If an error is returned, it logs the state that it occurs. If there is no error, it checks that the source MAC address and payload are correct.

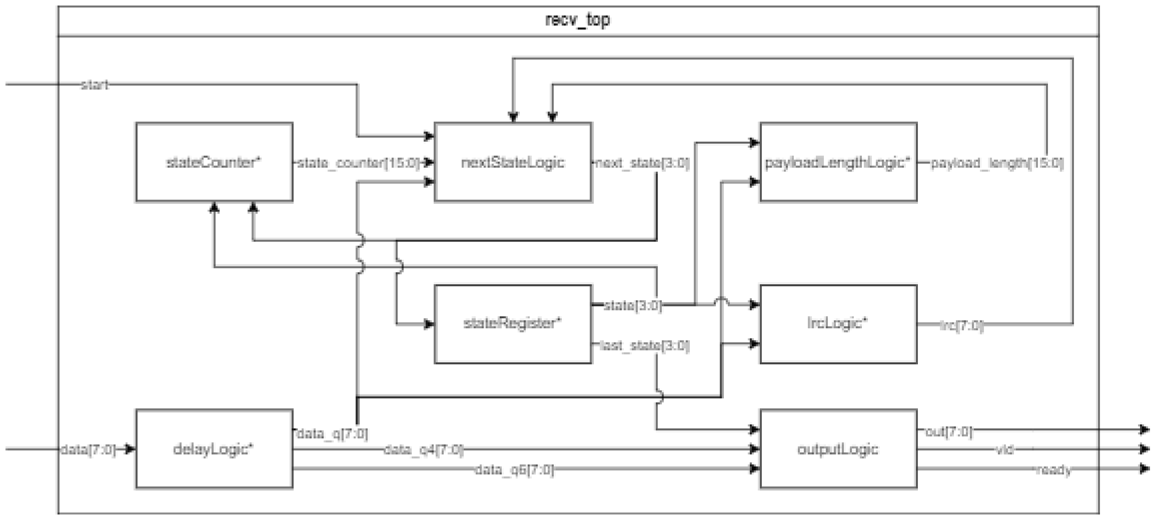
Init is run first, then driver and monitor are run concurrently. Tests are run for a set number of cycles that are enough for them to complete successfully so that they finish even if driver and monitor don't return. Then the result is asserted.

2.2.3 Helper Functions

Driver makes use of the helper functions to parse human-readable input values into lists of bytes that are streamed into the DUT one byte at a time:

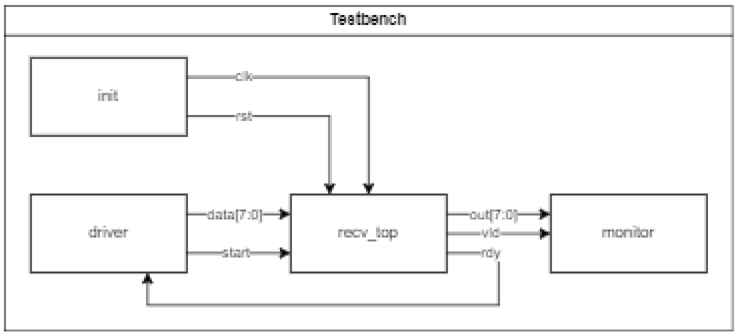
- **Parse_mac_address:** parses a MAC address string into a list of its bytes.
- **Get_pllen_bytes:** gets the length of an ASCII payload string and returns it as a list of its bytes.
- **Parse_payload:** parses an ASCII payload string and returns it as a list of its bytes.
- **Compute_fcs_bytes:** computes the FRC given the lists of bytes of the relevant sections.
- **Compute_lrc:** computes the LRC of the given list of bytes.

3 Top Level Block Diagram



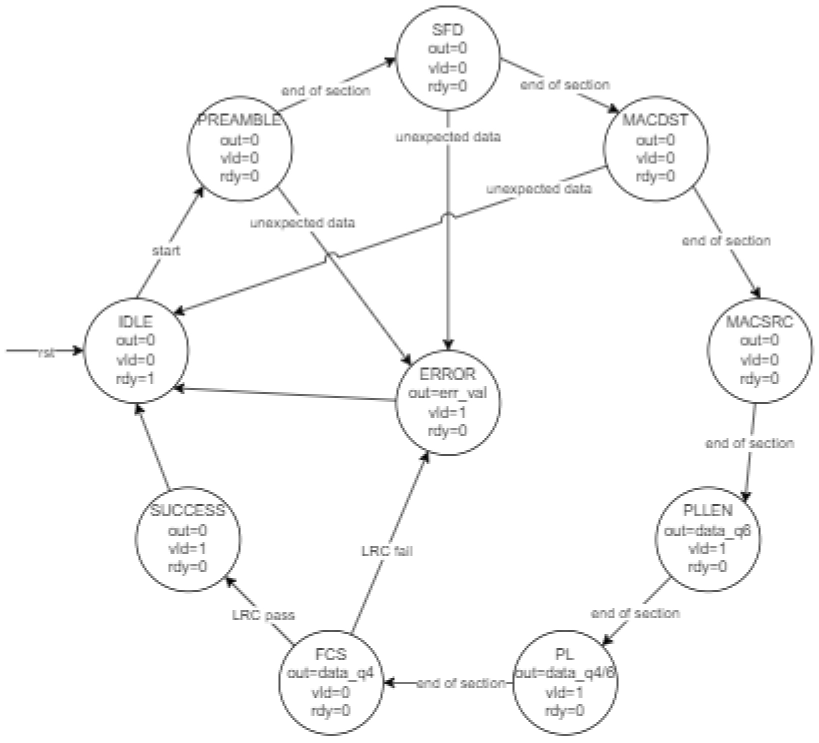
In the above diagram, the blocks are the always blocks in recv_top. Inputs are on the left, top, or bottom, while outputs are on the right. The inputs and outputs of recv_top come out of the block. The clk and rst signals were omitted to reduce complexity but are used in the sequential always blocks (indicated with an asterisk *). The functionality of each always block is described in Always Blocks.

4 System Level Block Diagram



In the above diagram, init, driver, and monitor are the main functions of the cocotb testbench, and recv_top is the DUT. Inputs are on the left, top, or bottom, while outputs are on the right. The functionality of and data flow between each testbench main function is described in Testbench Architecture.

5 State Diagram

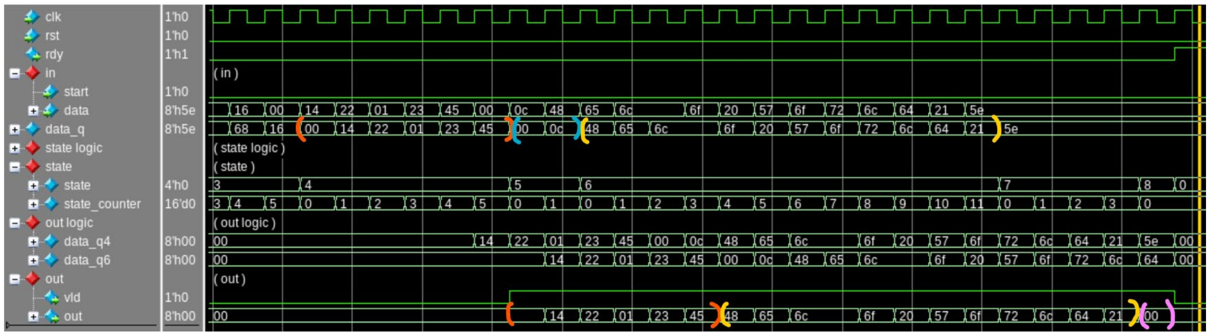


In the above diagram, each state is represented by a circle. Its outputs are inside the circle and conditions for transitioning to another state on its outbound arrows. The following simplifications were made to reduce complexity:

- Outputs:
  - Err_val: described in the second paragraph of I/O.
  - Data_q4/6: described in outputLogic in Always Blocks.
- Transitions (in order of precedence):
  - Unexpected data: input data is not correct for this section.
    - PREAMBLE: 8'b1010_1010 expected.
    - SFD: 8'b1010_1011 expected.
    - MACDST: byte of receiver's MAC address, most significant first. The receiver's MAC address is a module parameter, DEST_MAC_ADDR.
  - End of section: ethernet packet section lengths can be found in Overview.
  - LRC fail/pass: compared to the output of lrcLogic, documented in Always Blocks.
  - Every state has a self-loop, except for those without transition conditions, but not depicted to reduce complexity.

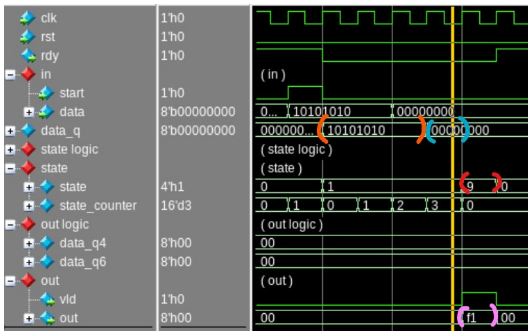
6 Simulation Waveforms

From test_recv_pass:



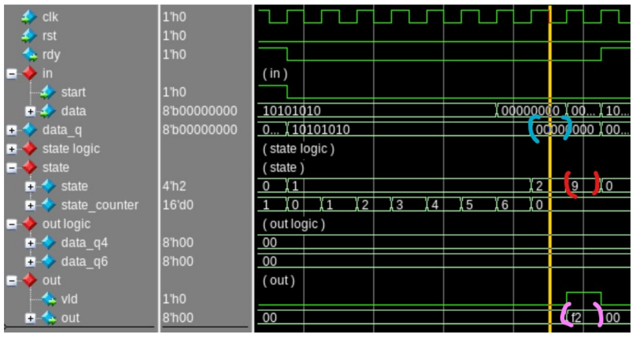
- Orange: source MAC address in MACSRC (state 4) is output 6 cycles later.
- Blue: payload length in PLLEN (state 5) is 12, which matches the length of the yellow section.
- Yellow: payload in PL (state 6) is output 4 cycles later.
- Pink: return 0 in SUCCESS (state 8) at end of output.

From test_recv_wrong_preamble:



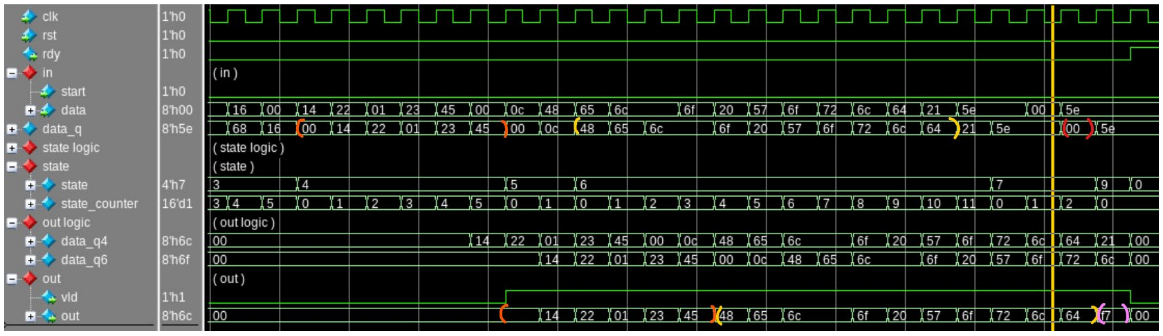
- Orange: expected value in PREAMBLE (state 1).
- Blue: unexpected value in PREAMBLE (state 1)
- Red: next state is ERROR (state 9).
- Pink: return error value: {4’hF, last_state}, where last_state is PREAMBLE (state 1).

From test_recv_wrong_sfd:



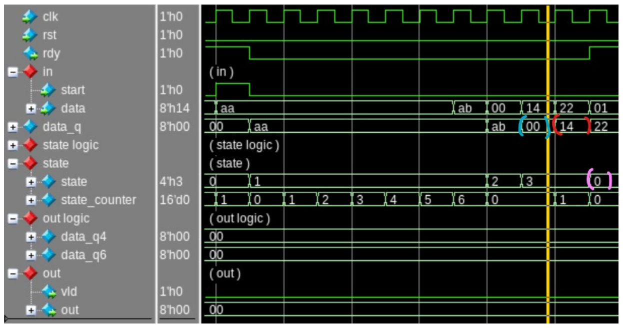
- Blue: unexpected value in SFD (state 2)
- Red: next state is ERROR (state 9).
- Pink: return error value: {4’hF, last_state}, where last_state is SFD (state 2).

From test_recv_wrong_fcs:



- Orange: source MAC address in MACSRC (state 4) is output 6 cycles later.
- Yellow: payload in PL (state 6) is output 4 cycles later until error.
- Blue: expected value in FCS (state 7).
- Red: unexpected value in FCS (state 7).
- Pink: return error value: {4'hF, last_state}, where last_state is FCS (state 7).

From test_recv_wrong_macdst:



- Blue: expected value in MACDST (state 3).
- Red: unexpected value in MACDST (state 4).
- Pink: next state is IDLE (state 0)

7 Appendix

7.1 Recv_top Source Code

```
module recv_top #(
  parameter logic [47:0] DEST_MAC_ADDR = 48'h00_0a_95_9d_68_16
) (
  output logic [7:0] out,
  output logic vld,
  output logic rdy,

  input logic [7:0] data,
  input logic start,

  input logic clk,
  input logic rst
);

 // state lengths
 localparam logic [15:0] PreambleLength = 16'h7;
 localparam logic [15:0] SFDLength = 16'h1;
 localparam logic [15:0] MACLength = 16'h6;
 localparam logic [15:0] PLLenLength = 16'h2;
 localparam logic [15:0] FCSLength = 16'h4;

 // data expected values
 localparam logic [7:0] PreambleOctet = 8'b1010_1010;
 localparam logic [7:0] SFDOctet = 8'b1010_1011;

 typedef enum logic [3:0] {
  IDLE,
  PREAMBLE,
  SFD,
  MACDST,
  MACSRC,
  PLLEN,
  PL,
  FCS,
  SUCCESS,
  ERROR
 } state_t;

 logic [15:0] payload_length, state_counter;
 logic [7:0] data_q, data_q1, data_q2, data_q3, data_q4, data_q5, data_q6;
 logic [7:0] lrc;
 logic [3:0] state, next_state, last_state;

 always_ff @(posedge clk) begin : stateCounter
  if (rst || state != next_state) begin
   state_counter <= 16'h0;
  end else begin
   state_counter <= state_counter + 16'h1;
  end
 end

 always_ff @(posedge clk) begin : stateRegister
  if (rst) begin
   last_state <= IDLE;
   state <= IDLE;
  end else begin
   last_state <= state;
   state <= next_state;
  end
 end

 always_comb begin : nextStateLogic
```

```
  case (state)
    IDLE: begin
      if (start) begin
        next_state = PREAMBLE;
      end
    end
    PREAMBLE: begin
      if (data_q != PreambleOctet) begin
        next_state = ERROR;
      end else if (state_counter >= PreambleLength - 16'h1) begin
        next_state = SFD;
      end
    end
    SFD: begin
      if (data_q != SFDOctet) begin
        next_state = ERROR;
      end else if (state_counter >= SFDLength - 16'h1) begin
        next_state = MACDST;
      end
    end
    MACDST: begin
      if (data_q != DEST_MAC_ADDR[(5-state_counter)*8+:8]) begin
        next_state = IDLE;
      end else if (state_counter >= MACLength - 16'h1) begin
        next_state = MACSRC;
      end
    end
    MACSRC: begin
      if (state_counter >= MACLength - 16'h1) begin
        next_state = PLLEN;
      end
    end
    PLLEN: begin
      if (state_counter >= PLLenLength - 16'h1) begin
        next_state = PL;
      end
    end
    PL: begin
      if (state_counter >= payload_length - 16'h1) begin
        next_state = FCS;
      end
    end
    FCS: begin
      if (data_q != (lrc ^ 8'hFF) + 1) begin
        next_state = ERROR;
      end else if (state_counter >= FCSLength - 16'h1) begin
        next_state = SUCCESS;
      end
    end
    SUCCESS: next_state = IDLE;
    ERROR:   next_state = IDLE;
    default: next_state = IDLE;
  endcase
end

always_comb begin : outputLogic
  case (state)
    IDLE: begin
      out = 8'h0;
      vld = 1'b0;
      rdy = 1'b1;
    end
    PREAMBLE, SFD, MACDST, MACSRC: begin
      out = 8'h0;
      vld = 1'b0;
      rdy = 1'b0;
      end
      PLLEN: begin
        out = data_q6;
        vld = 1'b1;
        rdy = 1'b0;
      end
      PL: begin
        out = (state_counter >= 4) ? data_q4 : data_q6;
        vld = 1'b1;
        rdy = 1'b0;
      end
      FCS: begin
        out = data_q4;
        vld = 1'b1;
        rdy = 1'b0;
      end
      SUCCESS: begin
        out = 8'h0;
        vld = 1'b1;
        rdy = 1'b0;
      end
      ERROR: begin
        out = {4'hF, last_state};
        vld = 1'b1;
        rdy = 1'b0;
      end
      default: begin
        out = 8'h0;
        vld = 1'b0;
        rdy = 1'b0;
      end
    endcase
  end

  always_ff @(posedge clk) begin : delayLogic
    if (!rst && (state == MACSRC || state == PLLEN || state == PL || state == FCS)) begin
      data_q  <= data;
      data_q1 <= data_q;
      data_q2 <= data_q1;
      data_q3 <= data_q2;
      data_q4 <= data_q3;
      data_q5 <= data_q4;
      data_q6 <= data_q5;
    end else if (!rst) begin
      data_q  <= data;
      data_q1 <= 8'h0;
      data_q2 <= 8'h0;
      data_q3 <= 8'h0;
      data_q4 <= 8'h0;
      data_q5 <= 8'h0;
      data_q6 <= 8'h0;
    end else begin
      data_q  <= 8'h0;
      data_q1 <= 8'h0;
      data_q2 <= 8'h0;
      data_q3 <= 8'h0;
      data_q4 <= 8'h0;
      data_q5 <= 8'h0;
      data_q6 <= 8'h0;
    end
  end

  always_ff @(posedge clk) begin : payloadLengthLogic
    if (!rst && (state == PLLEN)) begin
      payload_length[(1-state_counter)*8+:8] <= data_q;
    end else if (!rst && (state == PL)) begin
```

```
      payload_length <= payload_length;
    end else begin
      payload_length <= 16'h0;
    end
  end

  always_ff @(posedge clk) begin : lrcLogic
    if (!rst && (state == MACDST || state == MACSRC || state == PLLEN || state == PL)) begin
      lrc <= lrc + data_q;
    end else if (!rst && state == FCS) begin
      lrc <= lrc;
    end else begin
      lrc <= 8'h0;
    end
  end

endmodule
```

7.2 Testbench Source Code

```
import cocotb
from cocotb.clock import Clock
from cocotb.triggers import ClockCycles, RisingEdge

MACDST = "00:0a:95:9d:68:16"
MACSRC = "00:14:22:01:23:45"
PAYLOAD = "Hello World!"

@cocotb.test()
async def test_recv_pass(dut):
    """Test that receiver can successfully receive good data."""
    await init(dut)
    cocotb.start_soon(driver(dut, MACDST, PAYLOAD))
    m = cocotb.start_soon(monitor(dut))
    await ClockCycles(dut.clk, 45)
    assert m.result() == 0, "there was an error"

@cocotb.test()
async def test_recv_wrong_preamble(dut):
    """Test that receiver can detect that the Preamble section is wrong."""
    await init(dut)
    cocotb.start_soon(driver(dut, MACDST, PAYLOAD, wrong_preamble=True))
    m = cocotb.start_soon(monitor(dut))
    await ClockCycles(dut.clk, 15)
    assert m.result() == 1, "did not error out in state 1"

@cocotb.test()
async def test_recv_wrong_sfd(dut):
    """Test that receiver can detect that the SFD section is wrong."""
    await init(dut)
    cocotb.start_soon(driver(dut, MACDST, PAYLOAD, wrong_sfd=True))
    m = cocotb.start_soon(monitor(dut))
    await ClockCycles(dut.clk, 15)
    assert m.result() == 2, "did not error out in state 2"

@cocotb.test()
async def test_recv_wrong_fcs(dut):
    """Test that receiver can detect that the FCS section is wrong."""
    await init(dut)
    cocotb.start_soon(driver(dut, MACDST, PAYLOAD, wrong_fcs=True))
    m = cocotb.start_soon(monitor(dut))
    await ClockCycles(dut.clk, 45)
    assert m.result() == 7, "did not error out in state 7"

@cocotb.test()
async def test_recv_wrong_macdst(dut):
    """Test that receiver can ignore data that is not intended for it."""
    await init(dut)
    cocotb.start_soon(driver(dut, MACSRC, PAYLOAD))
    m = cocotb.start_soon(monitor(dut))
    await ClockCycles(dut.clk, 15)
    assert not m.done(), "monitor completed"

async def init(dut):
    clock = Clock(dut.clk, 1, units="us")  # Create a 1us period clock on port clk
    # Start the clock. Start it low to avoid issues on the first RisingEdge
    cocotb.start_soon(clock.start(start_high=False))

    dut._log.info("toggling rst and initializing inputs")
    dut.rst.value = 1
    dut.data.value = 0
    dut.start.value = 0
    await RisingEdge(dut.clk)
    dut.rst.value = 0
    await RisingEdge(dut.clk)

async def driver(
    dut, mac_dest, payload, wrong_preamble=False, wrong_sfd=False, wrong_fcs=False
):
    assert dut.rdy.value == 1, "rdy should be 1 after reset"

    dut._log.info("Driver: preamble start")
    preamble_byte = "1010_1010"
    dut.data.value = int(preamble_byte, 2)
    dut.start.value = 1
    await RisingEdge(dut.clk)
    dut.start.value = 0
    for i in range(6):
        if i == 2 and wrong_preamble:
            dut.data.value = 0
        await RisingEdge(dut.clk)

    dut._log.info("Driver: sfd start")
    sfd_byte = "1010_1011"
    if wrong_sfd:
        dut.data.value = 0
    else:
        dut.data.value = int(sfd_byte, 2)
    await RisingEdge(dut.clk)

    dut._log.info("Driver: macdst start")
    macdst_bytes = parse_mac_address(mac_dest)
    for b in macdst_bytes:
        dut.data.value = b
        await RisingEdge(dut.clk)

    dut._log.info("Driver: macsrc start")
    macsrc_bytes = parse_mac_address(MACSRC)
    for b in macsrc_bytes:
        dut.data.value = b
        await RisingEdge(dut.clk)

    dut._log.info("Driver: pllen start")
    pllen_bytes = get_pllen_bytes(payload)
    for b in pllen_bytes:
        dut.data.value = b
        await RisingEdge(dut.clk)

    dut._log.info("Driver: pl start")
    payload_bytes = parse_payload(payload)
    # dut._log.info([hex(b)[2:] for b in payload_bytes])
    for b in payload_bytes:
        dut.data.value = b
        await RisingEdge(dut.clk)

    dut._log.info("Driver: fcs start")
    fcs_bytes = compute_fcs_bytes(
        macdst_bytes, macsrc_bytes, pllen_bytes, payload_bytes
    )
    for i, b in enumerate(fcs_bytes):
        if i == 2 and wrong_fcs:
            dut.data.value = 0
        else:
            dut.data.value = b
        await RisingEdge(dut.clk)

async def monitor(dut):
    assert dut.out.value == 0, "out should be 0 after reset"
    assert dut.vld.value == 0, "vld should be 0 after reset"
    while dut.vld.value == 0:
        await RisingEdge(dut.clk)

    dut._log.info("Monitor: output start")
    out_int = []
    while dut.vld.value == 1:
        # dut._log.info(f"{hex(int(str(dut.out.value), 2))[2:]}")
        out_int.append(int(dut.out.value))
        await RisingEdge(dut.clk)

    mac_src, payload, ret = out_int[:6], out_int[6:-1], out_int[-1]
    mac_src = ":".join([hex(b)[2:].zfill(2) for b in mac_src])
    payload = "".join([chr(b) for b in payload])
    if ret != 0:
        err_state = ret & 0xF
        dut._log.error(f"Monitor: there was an error in state {err_state}")
    else:
        dut._log.info(f"Monitor: payload received from {mac_src}: {payload}")
        assert mac_src == MACSRC, "source MAC address incorrect"
        assert payload == PAYLOAD, "payload incorrect"
    dut._log.info("Monitor: output end")
    return err_state if ret != 0 else 0

def parse_mac_address(mac_str):
    """
    Example
        Input: "00:0a:95:9d:68:16"
        Output: [0, 10, 149, 157, 104, 22]
    """
    return [int(b, 16) for b in mac_str.split(":")]

def get_pllen_bytes(payload_str):
    payload_len = len(payload_str)
    payload_bytes = [
        payload_len >> 8,
        payload_len & 0xFF,
    ]
    return payload_bytes

def parse_payload(payload_str):
    """
    Example
        Input: "Hello World!"
        Output: [72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100, 33]
    """
    return [ord(c) for c in payload_str]

def compute_fcs_bytes(macdst_bytes, macsrc_bytes, pllen_bytes, payload_bytes):
    buffer = macdst_bytes + macsrc_bytes + pllen_bytes + payload_bytes
    lrc = compute_lrc(buffer)
    return [lrc] * 4

def compute_lrc(buffer):
    lrc = 0
    for b in buffer:
        lrc = (lrc + b) & 0xFF
    lrc = ((lrc ^ 0xFF) + 1) & 0xFF
    return lrc
```