Frame Aligner Verification Project

Chip Design & Verification Course

Presented by Vova Kostetsky

Project Deliverables

Specification, Assumptions, and Limitations:
Document the behavior of the frame aligner block, including assumptions and limitations.

Functional Verification Plan:
Outline the approach to test all functional aspects, including the detection of aligned and misaligned frames.

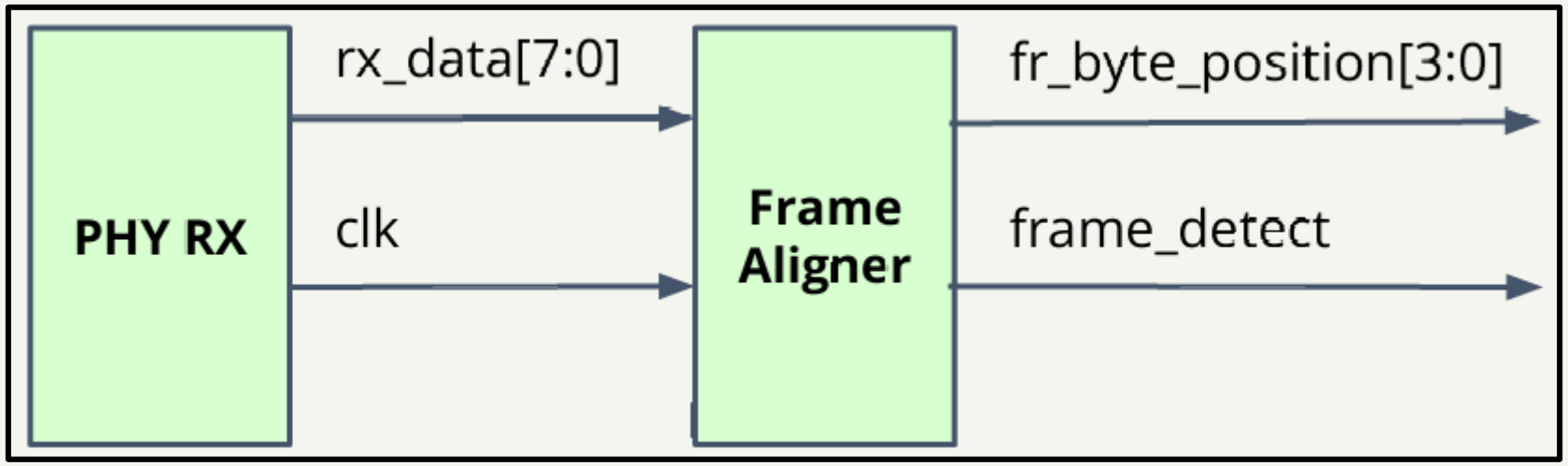
Coverage Plan:
Define the strategy to achieve coverage for all possible scenarios, such as correct/incorrect headers and payload lengths.

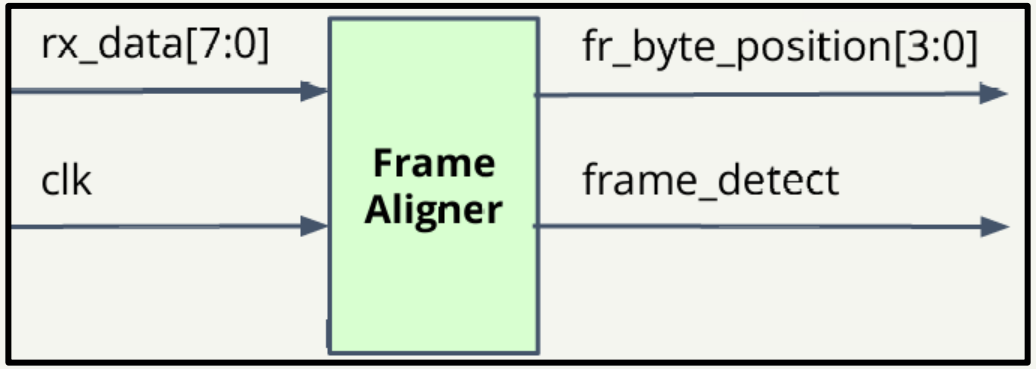
Test Plan:
Specify the test cases to be executed to verify the frame aligner's functionality.

Design Introduction

Brief overview of the design: A frame aligner is a component in communication systems, often used in data transmission protocols, that synchronizes incoming data by detecting patterns marking the start of each data frame. In this project, each frame has a header indicating its start and a payload containing the transmitted data.

Block Diagram





Input Interface:
The design has two inputs: an 8-bit data stream (rx_data[7:0]) from the PHY (Physical Layer) and a clock signal (clk).
The frame aligner uses these inputs to process the incoming data stream.

Frame Aligner Unit:
The frame aligner unit organizes incoming data into frames, each consisting of a 16-bit header and an 80-bit payload. The header contains a fixed pattern, either 0xAFAA or 0xBA55, marking the start of the frame, while the payload is random data. Frames are transmitted continuously without gaps.

Output Interface:
When the frame aligner detects a header, it outputs the 16-bit header followed by the 80-bit payload, ready for further processing.

Frame Aligner Functionality

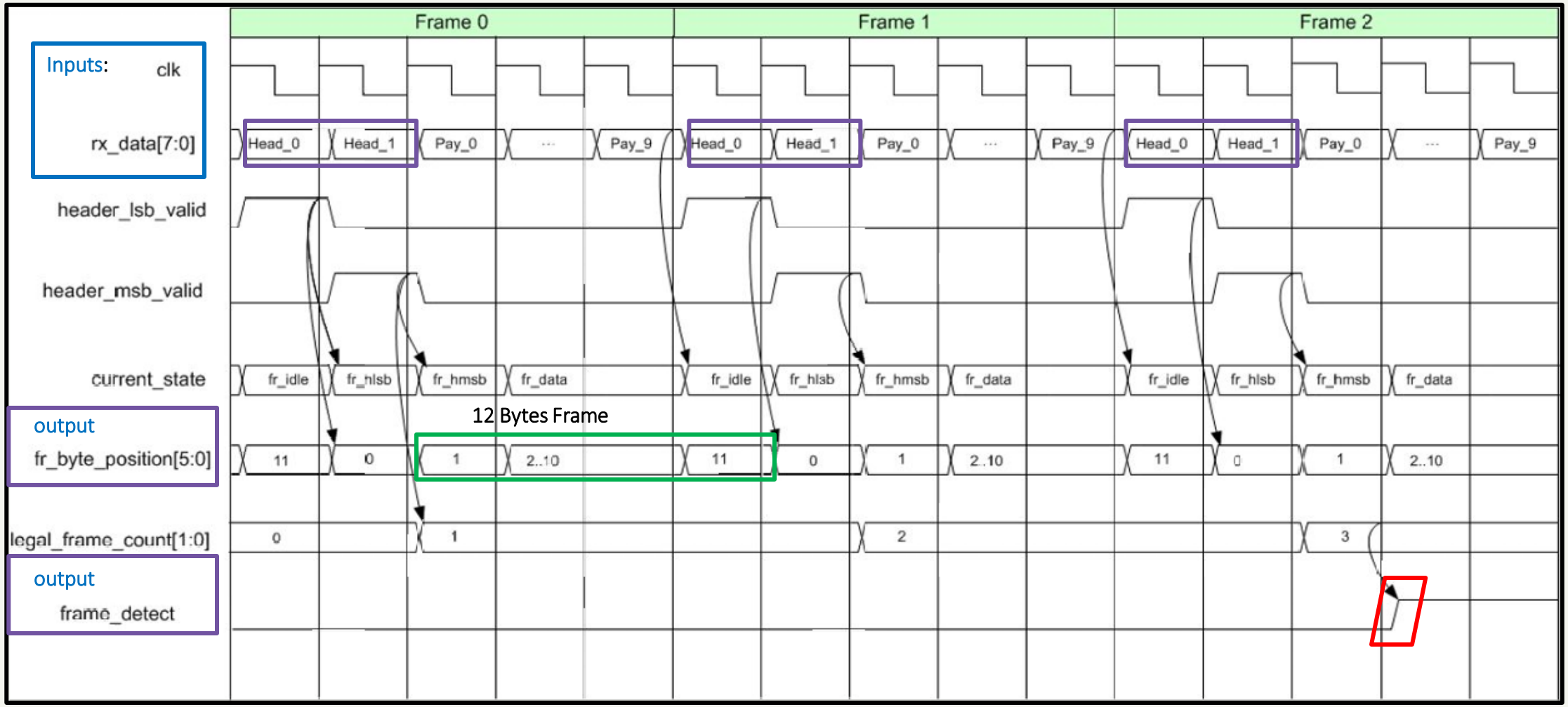
Frame Detection: The aligner monitors frame alignment status, signaling successful alignment with frame_detect.

Byte Position Tracking: Tracks the current byte position within each frame (header and payload) using fr_byte_position[3:0].

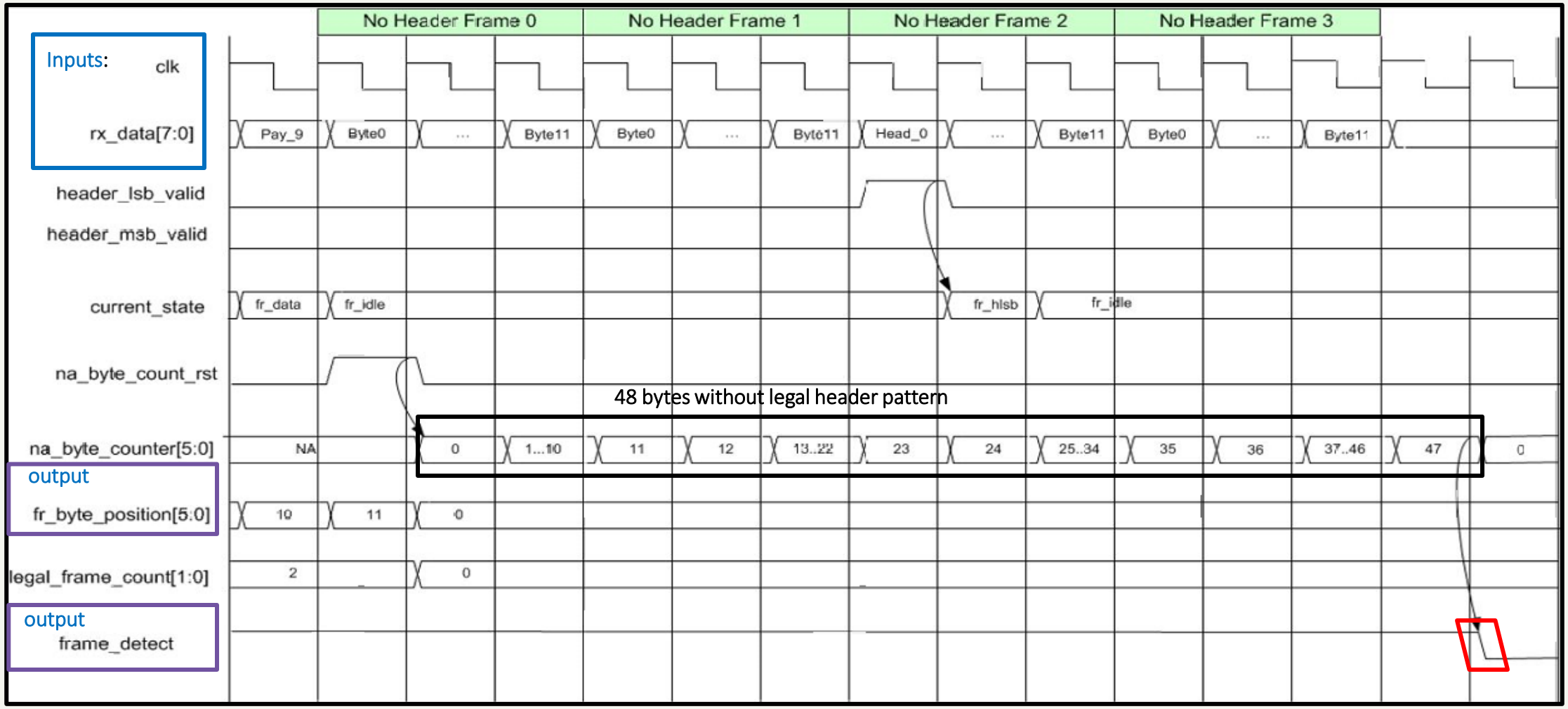
Alignment Algorithm: In-Frame Alignment: Achieved when three consecutive frames contain the correct header pattern.

Out-of-Frame Alignment: Declared when four consecutive frames have incorrect headers. This algorithm ensures reliable synchronization with incoming data frames.

An example of a waveform showing three consecutive frames containing the correct header pattern. The Frame_detect signal rises to 1. HEAD_1 and HEAD_2 are defined as the bytes of a legal header.



An example of a waveform with 48 bytes that do not contain a legal header pattern. The frame_detect signal becomes 0.



Verification Plan

In this section, we outline the plan to test the design early in the project, enabling prompt detection and correction of errors.

This also confirms the clarity and precision of the initial requirements.

The effectiveness of the testbench architecture is directly influenced by the quality of verification plan.

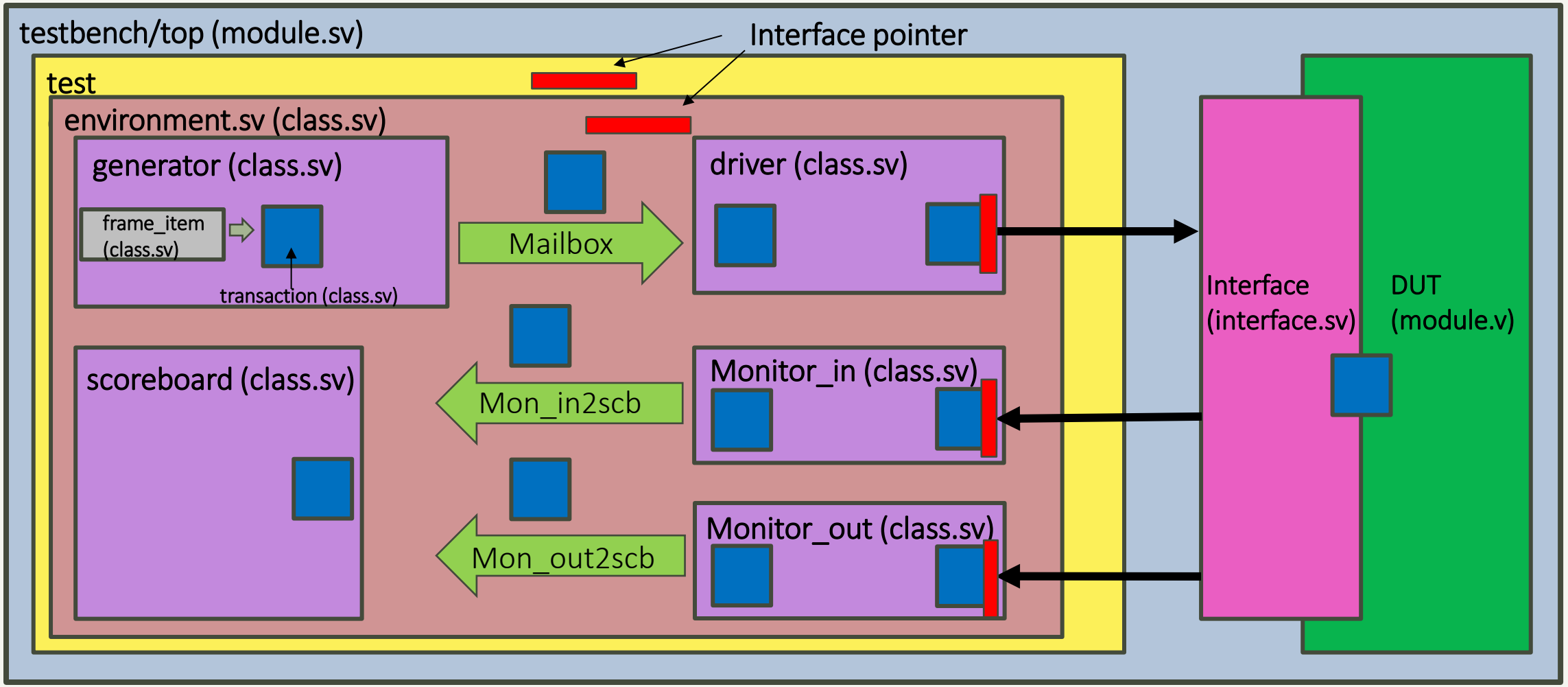
Data Item

To ensure correct frame structure, a frame_item class is defined to hold the frame content and the necessary constraints for control.

```
class frame_item;
    // Definitions
    typedef enum bit [1:0] {
        HEAD_1 = 2'b00,
        HEAD_2 = 2'b01,
        ILLEGAL = 2'b10
    } header_type_t;

    // Class members
    rand header_type_t header_type; // Header type (HEAD_1, HEAD_2, or ILLEGAL)
    logic [15:0]header;          // First two bytes as header
    rand byte payload[];            //Dynamic array Payload


  function new();
    this.payload=new[0]; //Initialize dynamic array for the payload
  endfunction


    // Post-randomization function to set header values based on header type
    function void post_randomize();
        // Set header values based on header type
        case(header_type)
           HEAD_1:begin
                header = 16'hAFAA;
            end
            HEAD_2: begin
                header = 16'hBA55;
            end
          ILLEGAL: begin
                header = $urandom_range(16'h00, 16'hFF); // Generate a random byte for illegal header
        end
        endcase
    endfunction
```

```
        // Constraints
    constraint header_constraint {
      // Randomly assign one of the header types with distribution
      header_type dist {HEAD_1 := 40 , HEAD_2 := 40, ILLEGAL := 20};
    }

    constraint payload_constraint {
      if (header_type == HEAD_1)
        {payload.size == 10;}

      else if (header_type == HEAD_2)
      {payload.size == 10;}

      else {
        payload.size inside {[0:46]}; //In a Case of a illegal header get a random playload
      }
        foreach (payload[i]) payload[i] inside {[8'h00:8'hFF]};
    }


endclass
```

Verification Environment



Components Description

DUT: Design of the frame aligner (frame_aligner.v)

Interface: Verifies the presence of proper communication between systems. Defines the connectivity direction for the input signals (clk, rx_data) and the output signals (frame_detect, fr_byte_position).

Transaction: represents a complete unit of communication between system components, including data and control signals. In our case, the signals are rx_data, fr_byte_position, and frame_detect.

frame_item: is a class that holds the frame content and the constraints needed to control it. It creates a random legal header (HEAD_1 and HEAD_2), which define the alignment of the frame, as well as a random illegal header. The class also generates a random size and bytes for the payload.

Generator: responsible for generating a random transaction, in this case, a random frame from the frame_item class, and then transmitting it to the transaction. In our case, the frame is divided into bytes. The generator does not contain a clock, but each time we insert the transaction and place it in the mailbox, the driver can retrieve it only when it is received from the mailbox on the negative edge of the clock (negedge clk).

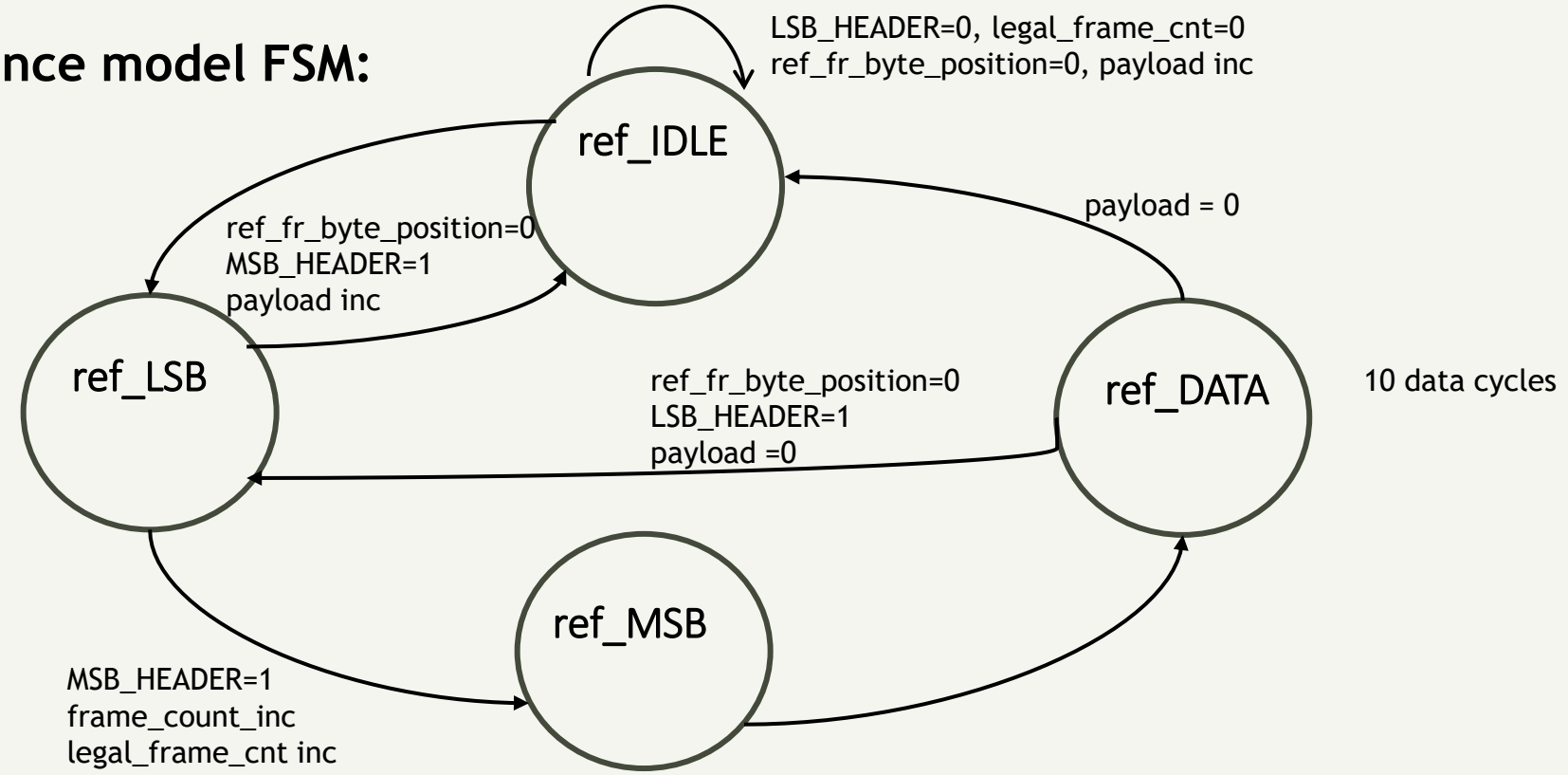
Driver: sends the generated rx_data transaction to the interface connected to the DUT. (negedge clk)

monitor_in: sits on the interface and listens. As soon as it detects an incoming signal, it assembles it and places it in the mailbox. In our design, monitor_in samples rx_data on the negative edge of the clock (negedge clk).

monitor_out: samples the output signals from the DUT and places them in the mailbox. In our design, the outputs are fr_byte_position and frame_detect.

Scoreboard: receives rx_data from monitor_in, passes it to the reference model, and compares the signals from the reference model to those from monitor_out. The reference model is built based on the model's description. We treat the DUT as a black box, and, according to the description, we build the reference model.

Reference model FSM:



Environment: includes all the classes in the correct order. It defines the mailbox and passes the pointer to both the generator and the driver. (The driver receives the pointer to the interface from the top/testbench.)

Test: creates a customized environment for specific test cases. In our case, we define the repeat counts for the generated transactions

testbench/top: contains the entire verification environment, passes the interface pointer to the driver, and connects the DUT, interface, and test.

Features to be Verified

1. **Header Detection:** Ensure the frame aligner correctly detects both legal headers (0xAFAA and 0xBA55) to initiate frame alignment. Verify the behavior when non-legal headers appear and check that they are correctly ignored.
2. **Output Data Integrity**: Verify that once alignment is detected, the frame aligner outputs the exact 16-bit header and the subsequent 80-bit payload without any modifications.\
3. **Alignment State Tracking**: Check that the frame_detect signal correctly indicates when in-frame alignment has been achieved or lost (based on three consecutive correct headers for alignment and four consecutive incorrect headers for misalignment). Ensure that the frame aligner transitions in and out of alignment accurately based on these rules.
4. **Byte Position Tracking**: Verify the accuracy of [3:0]fr_byte_position, ensuring it correctly tracks the byte position within both the header and payload across the frame.

Features to be Verified

5. **Back-to-Back Frame Processing:** Confirm that the aligner can handle back-to-back frames with no gaps in between, correctly transitioning from one frame to the next.
6. **Clock Synchronization:** Since the input data stream is clocked, verify that the frame aligner's operations are correctly synchronized to the clock signal.

Functional Coverage

1. **Header Detection Coverage**: Cover legal headers: Ensure both legal headers (0xAFAA and 0xBA55) are detected at least once. Cover illegal headers: Track a variety of illegal headers to ensure the aligner properly ignores them.
2. **Alignment and Misalignment Coverage**: In-frame alignment: Cover scenarios where three consecutive frames contain legal headers, confirming in-frame alignment is declared. Out-of-frame alignment: Cover scenarios where four consecutive frames have illegal headers, ensuring misalignment is declared.
   Transition cases: Cover transitions from in-frame to out-of-frame alignment and vice versa to ensure robustness at state boundaries.
3. **Frame Position and Byte Position Tracking**: Frame position tracking: Cover the entire frame sequence (header and payload), verifying that [3:0] fr_byte_positioncorrectly tracks each byte position within the frame (from 0 to 11 for each frame, assuming 2 header bytes + 10 payload bytes).Misaligned frame tracking: Verify fr_byte_position behavior in scenarios with consecutive incorrect headers.

Functional Coverage

4. **Output Interface Coverage**: Output data accuracy: Cover all values of the 16-bit header and 80-bit payload and ensure they are accurately output once detected.

   Back-to-back frame transitions: Cover scenarios where frames are transmitted back-to-back without gaps, ensuring that the output properly transitions from one frame to the next without errors.
5. **Payload Coverage**: Random payload data: Ensure a broad range of payload values is generated and checked to confirm the aligner can handle varied data content.

   Payload consistency: Cover payload consistency by ensuring that payload data from the input matches exactly with what is output once a frame is detected.
6. **Edge and Corner Cases**: Boundary transitions: Cover cases where headers change from legal to illegal, or vice versa, at frame boundaries to verify proper alignment or misalignment.

   Minimum and maximum payload values: Ensure the aligner correctly handles both minimum and maximum payload (size and values).

List of Tests

Test Number	Test Name	Objective	Description
1	Basic Header Detection Test	Verify that the aligner correctly identifies both legal headers (0xAFAA and 0xBA55).	Send frames with legal headers and check if the frame aligner detects each frame correctly.
2	Basic Misalignment Test	Verify misalignment behavior when receiving consecutive incorrect headers.	Send four frames with illegal headers and ensure the aligner declares misalignment.
3	Alignment Establishment Test	Confirm that alignment is achieved after three consecutive valid frames.	Send three consecutive frames with legal headers and verify the aligner declares alignment after the third frame.
4	Misalignment Recovery Test	Ensure misalignment is declared after four incorrect headers, even after achieving alignment.	First achieve alignment, then send four consecutive frames with incorrect headers and check if misalignment is declared.

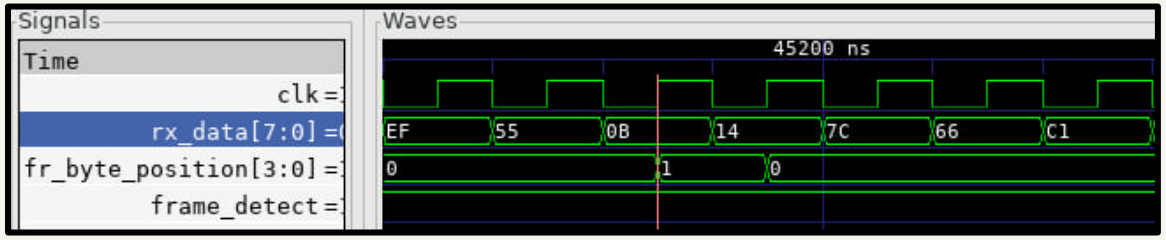
Test Number	Test Name	Objective	Description
5	Back-to-Back Frames Test	Verify handling of back-to-back frames without gaps.	Send a series of consecutive frames with legal headers, without gaps, and confirm correct transitions.
6	Frame Position Tracking Test	Verify that [3:0]fr_byte_positionaccurately tracks the byte position within each frame.	Send frames and monitor [3:0]fr_byte_position to ensure it updates correctly through header and payload bytes (0 to 11).
7	Payload Integrity Test	Confirm that the 80-bit payload is accurately output after frame detection.	Send frames with known payload data and verify the detected frames output the exact payload received.
8	Random Payload Test	Test robustness with random payload data.	Generate frames with random payloads and check that each frame is detected correctly without altering payload integrity.
9	Boundary Condition Test	Test behavior at boundary between valid and invalid frames.	Send frames that alternate between valid and invalid headers, checking the transition between aligned and misaligned states. After 46 bytes of payload send a legal header.

Test Number	Test Name	Objective	Description
10	Max/Min Payload Test	Verify aligner's handling of all-zeroes and all-ones payload data.	Send frames with all-zeroes and all-ones payloads, confirming that extreme values are handled correctly.
11	Edge Case Header Test	Verify response to an LSB of a header at the end of a 47-byte-long payload.	While frame_detect = 1, send 47 bytes of payload, then send 1 byte of the LSB from a legal header.
12	All header combination	Verify 2^3 legal headers combination.	While frame_detect=0 send all the possible combination of HEAD_1 and HEAD_2 and 3 times to verify frame_detect rising.
13	Corner Case frame_detect	Verify the if the frame_detect don't change after 46 payload bytes and 1 legal header.	While frame_detect=1 and 46 bytes of payload send legal header to check if frame_detect doesn't fall.
14	Corner Case fr_byte_position	Verify for a 3rd LSB byte the fr_byte_position don't become 0.	Send a legal header a right after an a legal LSB.
15	Payload with LSB	Verify a scenario when a last byte of payload I LSB and then legal header received	Send LSB LSB MSB and verify if na_byte_position increasing.

Bug Report

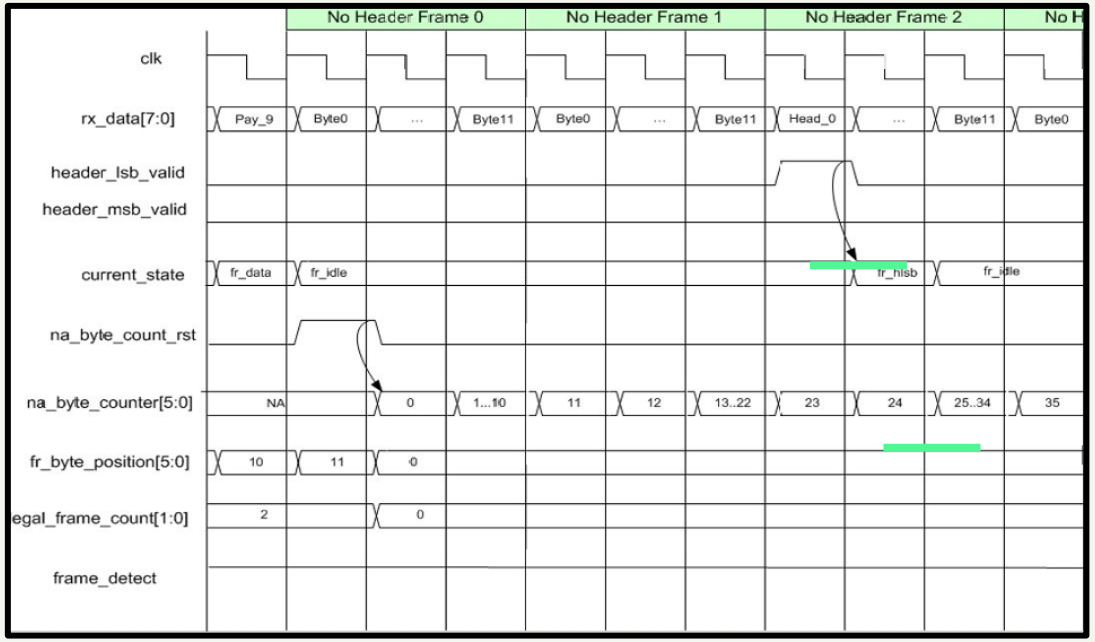
During rigorous testing, three unique bugs were detected that directly impact frame alignment functionality:

1. After receiving the first byte of the header, known as the LSB, fr_byte_position is set to 1, in contrast to what was shown in the design waveforms.



Here we can see the byte 0x55, which is the LSB of a header, rising, and fr_byte_position increases even though there is no MSB following it.

```
Reference: 0 Actual: 1 Reference_FD: 1 Actual_FD: 1
time is:                45190
```



2. The second bug occurs when frame_detect is 1, and we have 46 bytes of payload followed by a legal header. According to the specifications, this should reset the illegal bytes counter and keep frame_detect at 1. However, as shown here, the opposite happens, and frame_detect decreases to 0.


```
~Wrong Result.
Reference: 2 Actual: 2 Reference_FD: 1 Actual_FD: 0
```

3. When we have a sequence of LSB, LSB, MSB arriving in this order, the design fails to recognize the legal header. An additional transition from LSB to LSB may need to be added in the FSM.