Frame Aligner Verification Project

Yoav Dror

November 2024

Verification Plan

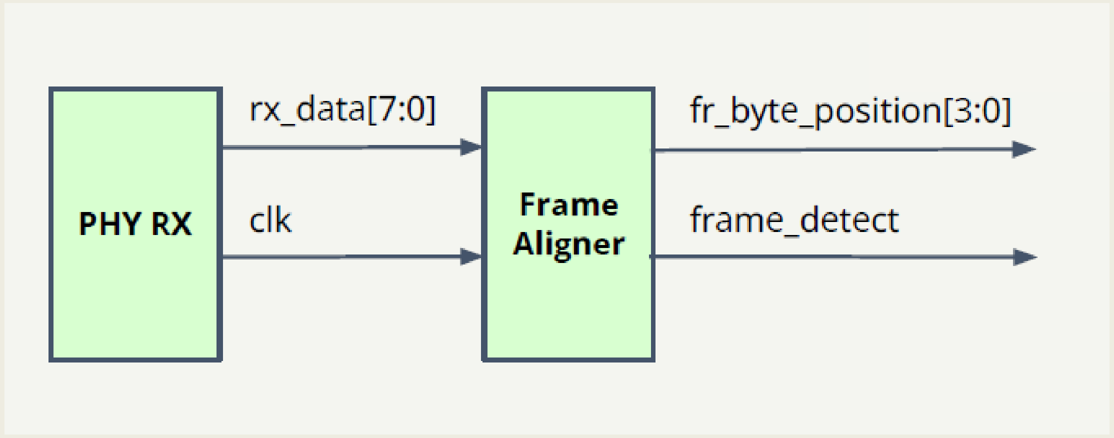
Introduction

Verification Environment

Features to be Verified

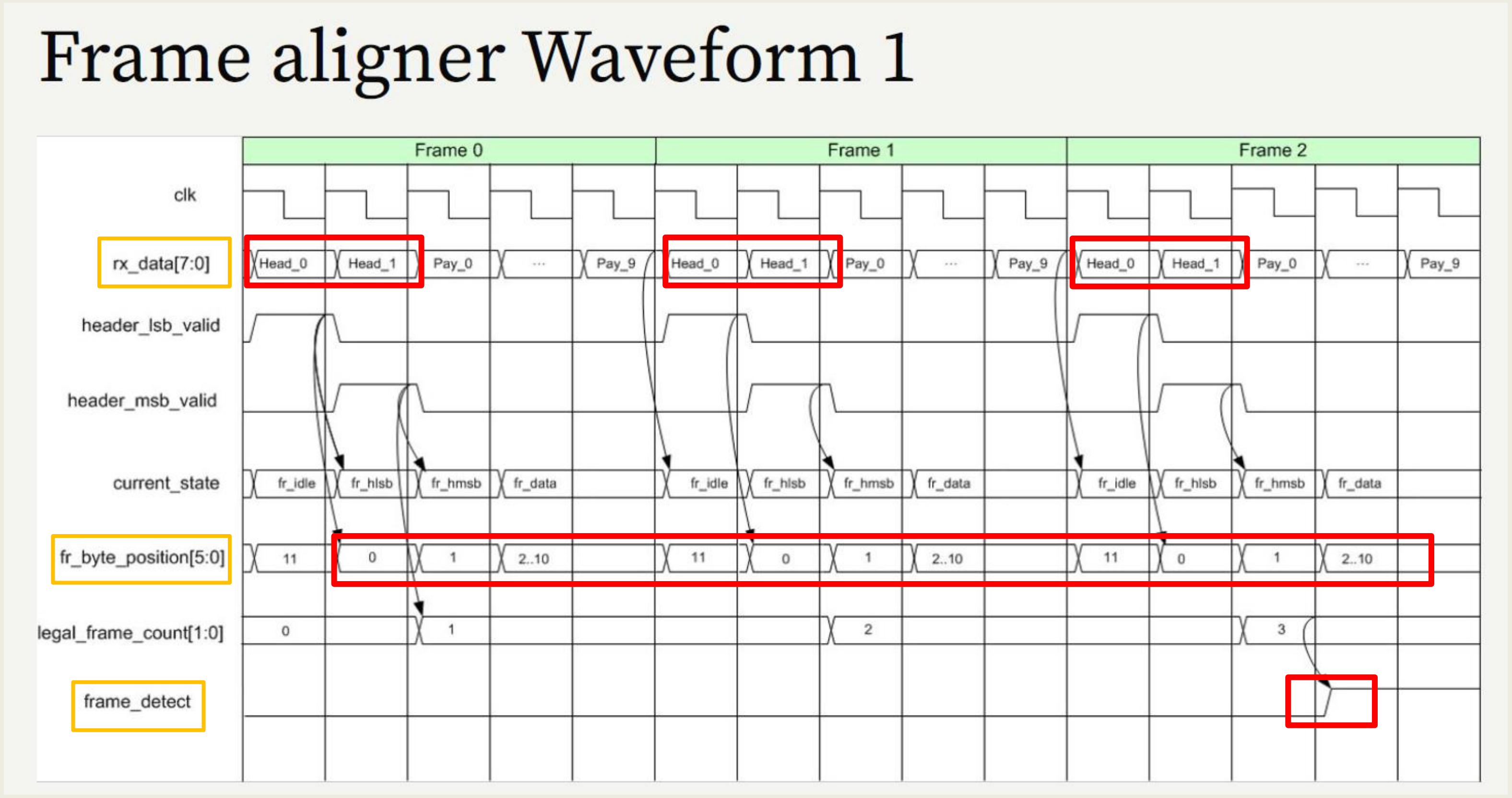
Introduction

- **Design Overview**
  - A frame aligner is a component in communication systems used in data transmission protocols. Its main role is to synchronize incoming data streams by detecting specific patterns within data frames.
  - The frame aligner synchronizes the incoming data stream by detecting specific 2-byte header patterns (0xAFAA or 0xBA55) within 12-byte frames, each consisting of a 2-byte header and an 8-byte payload.



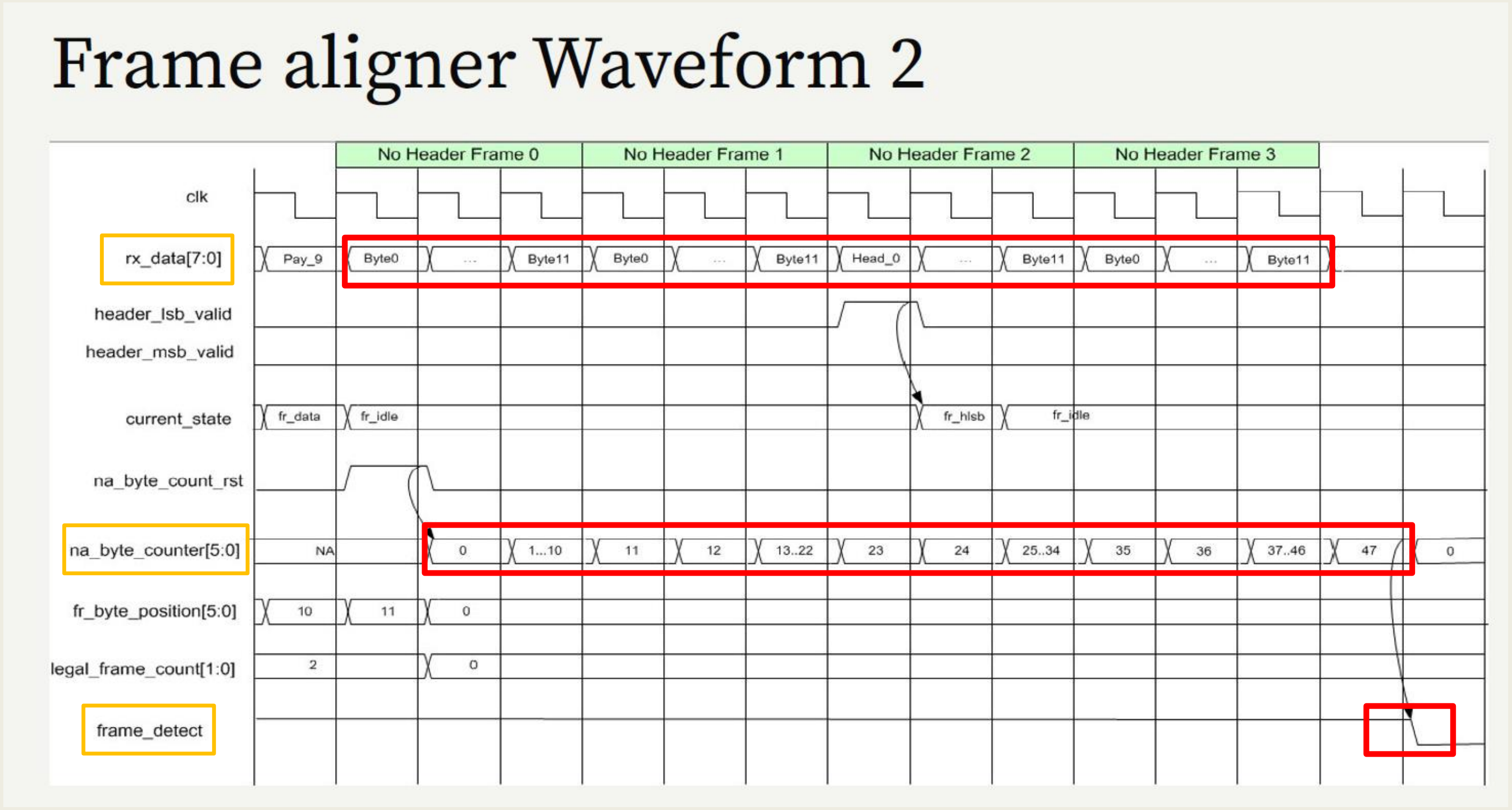
Introduction

- **Waveform - Aligment**



Introduction

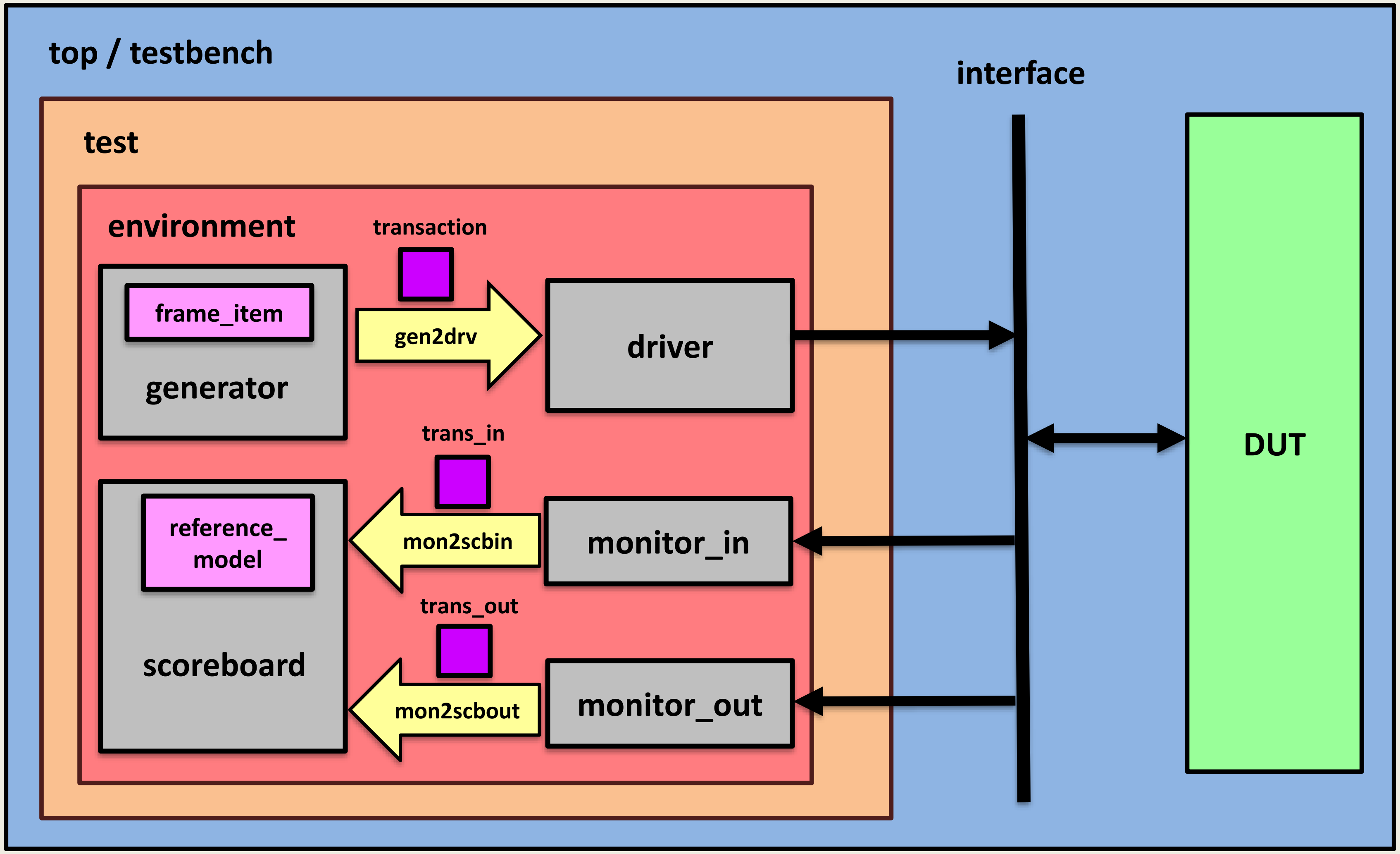
- **Waveform - Misligment**



Introduction

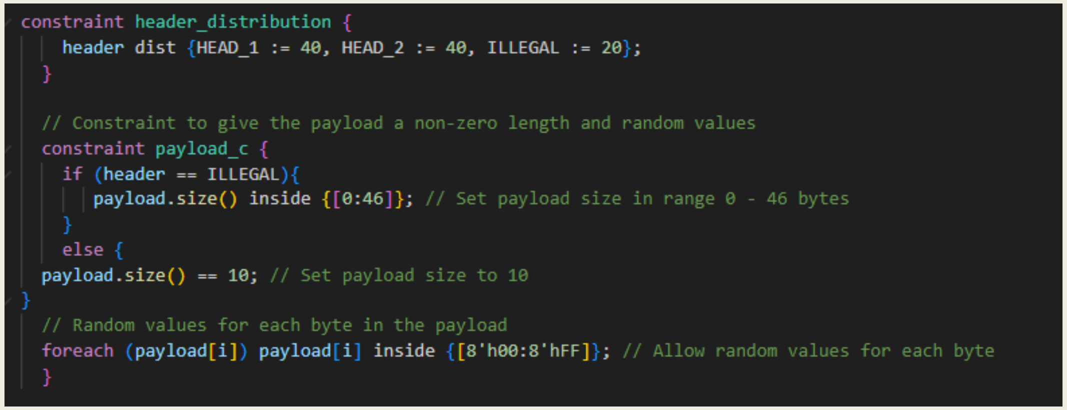
- **Goals of Verification Process**
  - **Achieve 100% Functional Coverage:** Test all critical aspects of the frame aligner's operation to confirm comprehensive functionality.
  - **Identify all Bugs:** Detect issues in frame detection, such as misalignment errors or incorrect byte position tracking, to prevent data synchronization failures.
  - **Validate Design functionality under Edge Conditions:** Test the frame aligner's reliability and accuracy with edge cases, such as the smallest and largest possible frame sizes or unusual header sequences, to confirm robust operation.

Verification Environment



Components

- **DUT:** frame_aligner.sv.
- **Interface:** centralizes DUT-testbench communication, managing signals like clk, reset, rx_data, fr_byte_position, and frame_detect.
- **Transaction**: contains the signals - Input [7:0] rx_data Output [3:0] fr_byte_position Output frame_detect.
- **Generator:** Generates random frame using frame_item class and transmit them to the driver byte after byte.
- **frame_item:**
  - Assigns header values using header_type_t enum for HEAD_1 (0xAFAA), HEAD_2 (0xBA55), or ILLEGAL. header distribution: 40% **each** for HEAD_1 and HEAD_2, and 20% for ILLEGAL.
  - Defines a frame transaction structure with a 16-bit header. with randomized payload.

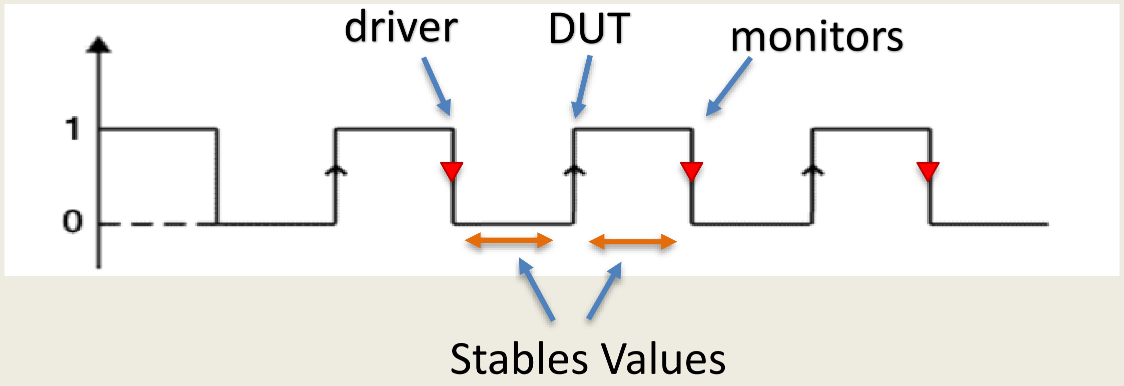


```
constraint header_distribution {
    header dist {HEAD_1 := 40, HEAD_2 := 40, ILLEGAL := 20};
  }

  // Constraint to give the payload a non-zero length and random values
  constraint payload_c {
    if (header == ILLEGAL){
       payload.size() inside {[0:46]}; // Set payload size in range 0 - 46 bytes
    }
    else {
  payload.size() == 10; // Set payload size to 10
}
  // Random values for each byte in the payload
  foreach (payload[i]) payload[i] inside {[8'h00:8'hFF]}; // Allow random values for each byte
  }
```

Components

- **Driver:** sends generated transactions to the DUT via rx_data in the interface.
- **Monitor_in:** Samples rx_data from DUT inputs via the interface, sending data to the scoreboard for expected result calculation.
- **Monitor_out:** Samples DUT outputs (frame_detect and fr_byte_position), forwarding data to the scoreboard for comparison with expected values.
- **Driver and Monitors Sampling on Falling Edge:** Sampling on the **falling edge clock** captures stable values set by the DUT on the rising edge, **preventing timing overlaps and race conditions for accurate synchronization**.
- **Scoreboard:** Receives rx_data from monitor_in, calculates expected frame_detect and fr_byte_position values via a reference module, and compares them with DUT outputs to report pass/fail for each test.



Components

- **Reference Model:** Contains state machine that processes rx_data to detect valid frames, with four states:
  - **IDLE:** Monitors for header start values (0xAA or 0x55), increments na_byte_counter for non-aligned bytes, and resets at a threshold of 47, declaring misalignment (frame_detect = 0). Transitions to H_LSB if a header start byte is detected.
  - **H_LSB:** Validates the LSB of the header, setting fr_byte_position to 0. Transitions to H_MSB if the byte forms a valid header pair with previous_byte, otherwise, returns to IDLE.
  - **H_MSB:** Confirms the header's MSB and increments frame_counter to track consecutive valid headers.
  - **PAYLOAD:** Processes each byte within the frame, incrementing fr_byte_position. Declares alignment (frame_detect = 1) if three consecutive headers are detected. After the payload completes, the FSM transitions to either IDLE or H_LSB based on whether the next byte matches a header start value.
- **Purpose and Flags:** The FSM uses update_fr_byte_position_clock_after and update_frame_detect_clock_after to control clock-sensitive updates for fr_byte_position and frame_detect, ensuring changes occur precisely one clock cycle later

Components

- **Environment:** Contains all the components, including the generator, driver, monitors, scoreboard, and transactions, ensuring a complete verification setup.
- **Test:** Creates a tailored environment for specific test cases.
- **Top:** The top-level module that connects the DUT, test, and interface

Features to Be Verified

- **Accurate Frame Detection:** Ensure the frame aligner declares alignment when three consecutive frames have correct headers and declares misalignment when four consecutive frames lack valid headers.
- **Byte Position Tracking:** Verify precise tracking of the byte position within each frame, distinguishing between header and payload sections in all scenarios.
- **Reliability in Edge Cases:** Evaluate the performance of the system under challenging conditions, such as:
  - <u>Alignment and Misalignment Resilience</u>: Assess the system's ability to correctly align and detect frames even when facing potential misalignment issues.
  - <u>Handling Unusual Header Patterns</u>: Ensure proper handling of irregular header patterns, including mis-ordered headers, to avoid incorrect detections or failures.

Coverage Plan

Code Coverage

Functional Converge

Code Coverage

- **Statement Coverage:** Ensures every line of code is executed with a **target of 100%.**
- **Branch Coverage:** Verifies all possible true/false branches are tested with a **target of 100%.**
- **Toggle Coverage**: Ensures all Boolean sub-expressions are tested with a **target of 100%.**
- **FSM Coverage:** Covers all states and transitions in the frame aligner FSM with a **target of 100%** of states and transitions.

Functional Coverage

Coverage	**valid_frame_inst**
Location	**interface**
target	**Alignment Detection:** Identifies all 8 valid patterns
Expected Result	**Frame_detect = 1** until 4 consecutive frames with missing headers
Initial condition	**Frame_detect = 0**
Initial input	rx_data inside 0x00 – 0xFF for 48 time with no valid header (0xAFAA, 0XBA55)
Input: Pattern_1	0xAFAA, 0xAFAA, 0xAFAA + 10-byte payload for each header
Input: Pattern_2	0xBA55, 0xBA55, 0xBA55 + 10-byte payload for each header
Input: Pattern_3	0xAFAA, 0XBA55, 0xAFAA + 10-byte payload for each header
Input: Pattern_4	0xBA55, 0xAFAA, 0XBA55 + 10-byte payload for each header
Input: Pattern_5	0xAFAA, 0xBA55, 0xBA55 + 10-byte payload for each header
Input: Pattern_6	0xBA55, 0xAFAA, 0xAFAA + 10-byte payload for each header
Input: Pattern_7	0xAFAA, 0xAFAA, 0xBA55 + 10-byte payload for each header
Input: Pattern_8	0xBA55, 0xBA55, 0xAFAA + 10-byte payload for each header

Functional Coverage

Assertion	**assert_alignment_resilience**
Location	**interface**
target	**Frame detect stays 1 if there are 4 not consecutive non valid frames**
Expected Result	**Frame_detect = 1**
Initial conditon	**Frame_detect = 1**
Input: Frame_Pattern_1	ILLEGAL, LEGAL, ILLEGAL, ILLEGAL, ILLEGAL
Input: Frame_Pattern_2	ILLEGAL, ILLEGAL, LEGAL, ILLEGAL, ILLEGAL
Input: Frame_Pattern_3	ILLEGAL, ILLEGAL, ILLEGAL, LEGAL, ILLEGAL

Functional Coverage

Coverage	**byte_position_tracking_cg**
Location	**interface**
target	**Byte Position Tracking:** Tracks position from 0 to 11 in each valid frame
Expected Result	**Frame_byte_position = 0:11 when frame is valid**
Initial condition	**Correct header**
Initial input	0xAFAA or 0XBA55 (starts count from LSB)

Functional Coverage

Coverage	**cov_misalignment**
Location	**interface**
target	**Misalignment Detection**
Expected Result	**Frame_detect = 0**
Initial conditon	**Frame_detect = 1**
Input	rx_data inside {8'h00:8'hFF} for 48 time with no legal headers

Functional Coverage

Assertion	**assert_misaligment_resilience**
Location	**interface**
target	**Frame detect stays 0 if there are 3 not consecutive valid frames**
Expected Result	**Frame_detect = 0**
Initial condition	**Frame_detect = 0**
Input: Frame_Pattern_1	LEGAL, LEGAL, LSB, LEGAL
Input: Frame_Pattern_2	LEGAL, LSB, LEGAL, LEGAL

Functional Coverage

Coverage and Assertion	**mixed_header_patterns**
Location	**interface**
target	**Doesn’t detect wrong header as a correct one and vice versa (Doesn’t miss valid headers when there is a stream of valid LSB and MSB)**
Expected Result	**header_msb_valid == 0 or 1 according to the pattern and if its 1, fr_byte_position counts accurately**
Initial condition	**Frame_detect = 0**
Input	Non valid headers made from : 0xAF, 0xAA, 0xBA, 0x55

Functional Coverage

Coverage and Assertion	**correct_lsb_last_na_byte_counter**
Location	**interface**
target	**Valid LSB at position 47 doesn’t restart illegal count**
Expected Result	**Frame_detect = 0**
Initial condition	**Frame_detect = 1**
Input	Non valid header (2-bytes) flowed by 45 bytes != (0xAA, 0x55) and than 0xAA or 0x55 (1-byte)

Functional Coverage

Coverage and Assertion	**correct_header_last_na_byte_counter**
Location	**interface**
target	**Valid header restarts illegal byte count**
Expected Result	**Frame_detect stays 1**
Initial condition	**Frame_detect = 1**
Input	Non valid header(2-bytes) flowed by 44 bytes != (0xAA, 0x55) and than 0xAFAA or 0xBA55 (2-bytes)

Bugs

Detection and Analysis

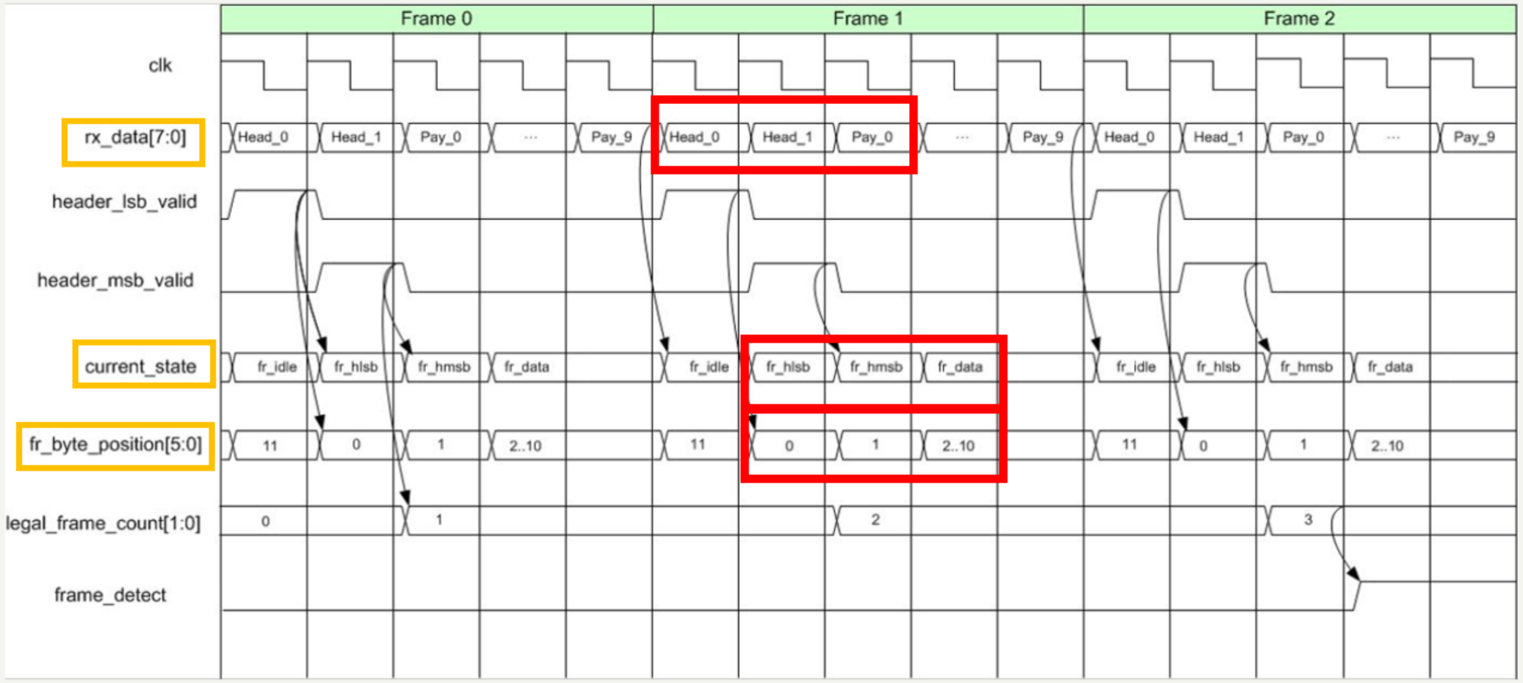
Bug A: fr_byte_position Sets to 1 After LSB Detection, Before MSB Verification

Detection and Analysis

Bug Detection and Analysis

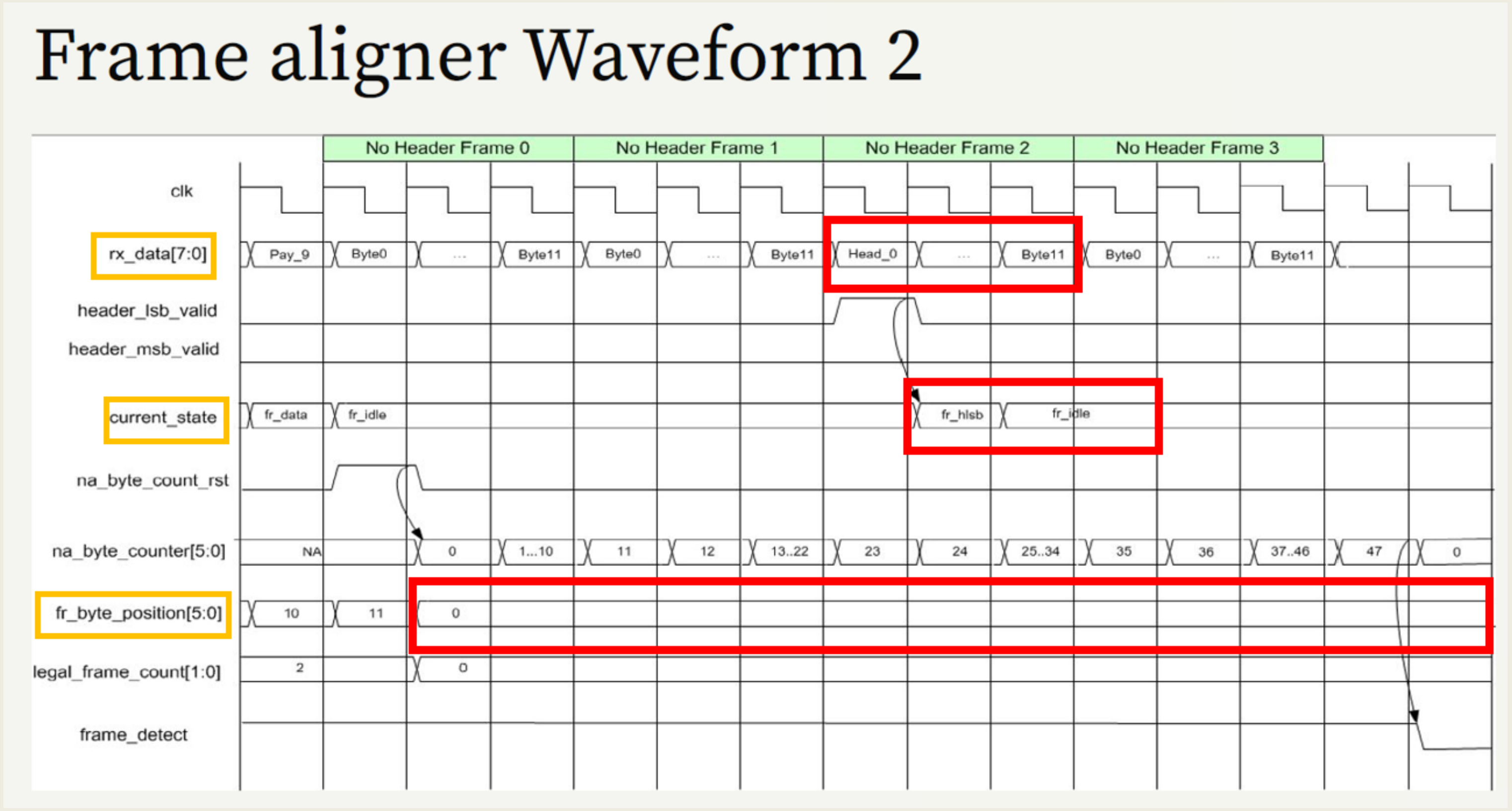
- **Expected Results:**
  - **Correct Header Detected**: When the system detects a correct header, the fr_byte_position should accurately indicate the position within the frame.

Frame aligner Waveform 1



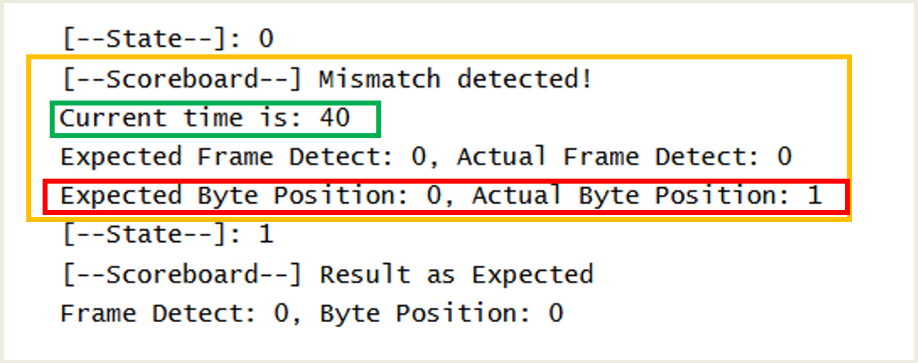
Bug Detection and Analysis

- **Expected Results:**
  - **Incorrect Header Detection**: For invalid headers (e.g., 0xAAAA, 0xAAAF, 0x55BA), fr_byte_position remains at 0, indicating no valid header was recognized.



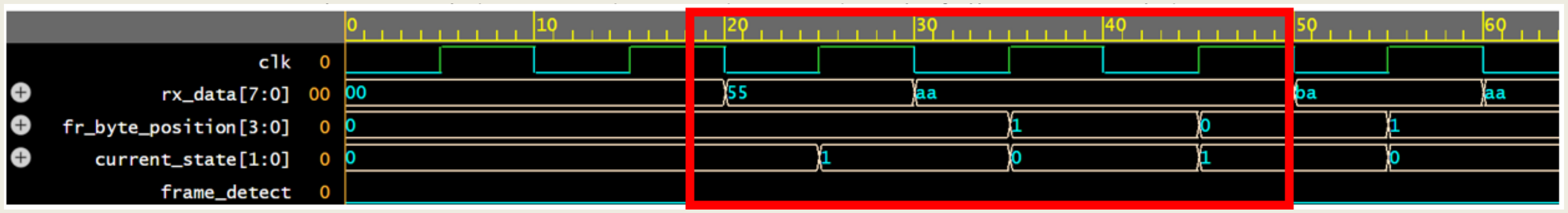
Bug Detection and Analysis

- **Scoreboard:**



```
[--State--]: 0
[--Scoreboard--] Mismatch detected!
Current time is: 40
Expected Frame Detect: 0, Actual Frame Detect: 0
Expected Byte Position: 0, Actual Byte Position: 1
[--State--]: 1
[--Scoreboard--] Result as Expected
Frame Detect: 0, Byte Position: 0
```

- **Waveform:** According to the waveform, our input consists of a series of 0x55 byte followed by two 0xAA bytes. Per the specifications, when a new frame begins, fr_byte_position should reset to 0. It should only

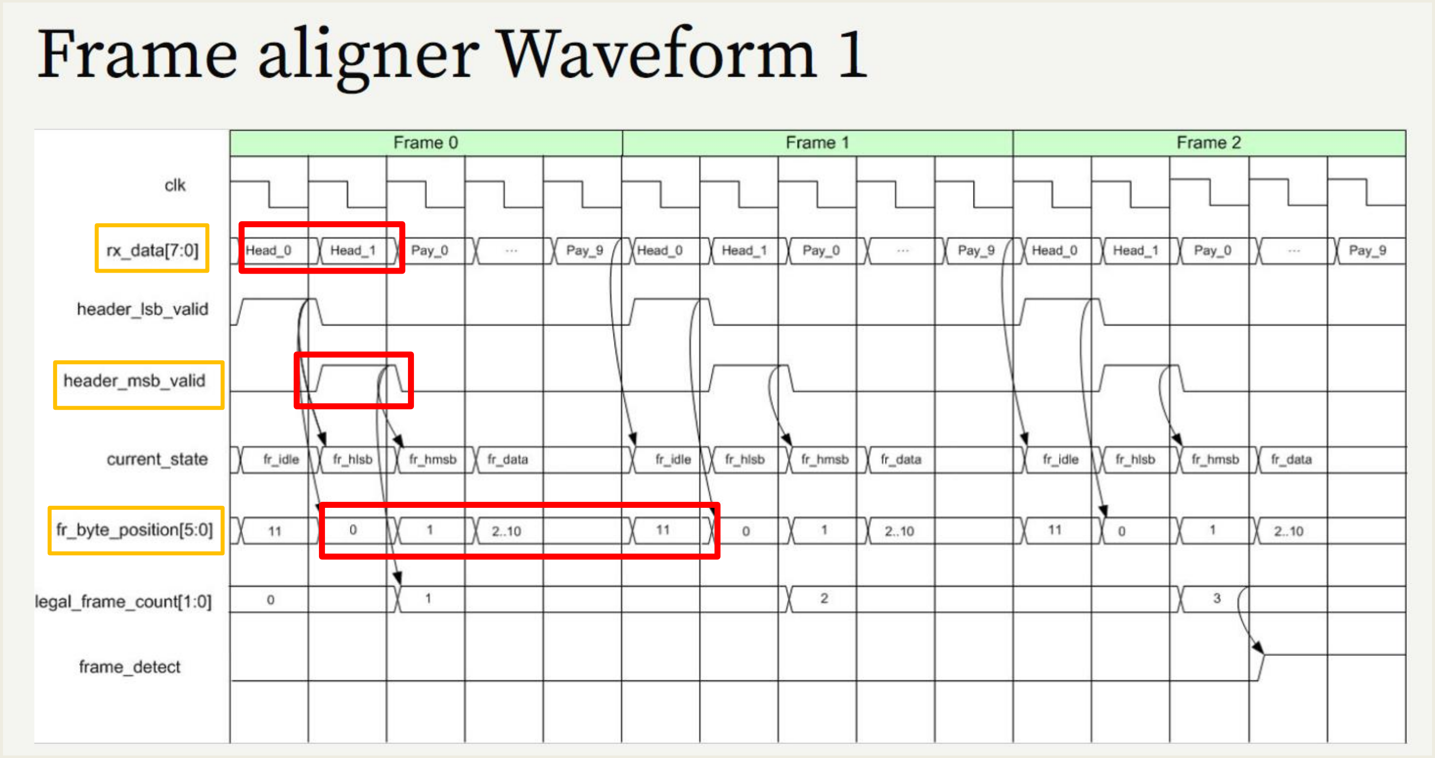


Bug B: Fails to increment fr_byte_position when the sequence is LSB-LSB-MSB

Detection and Analysis

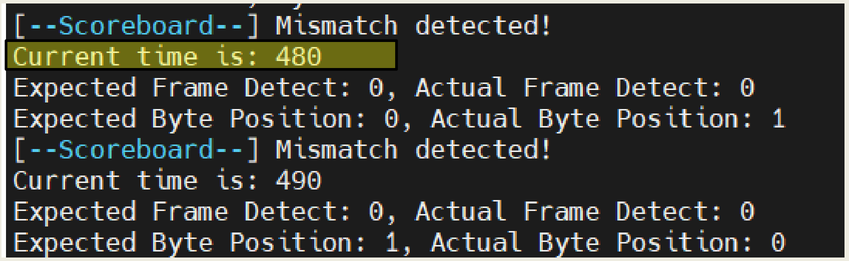
Bug Detection and Analysis

- **Expected Result:** When a pattern of LSB followed by MSB, forming a valid header, is detected during the counting of the 48 "illegal bytes", the design should recognize it as the start of a valid header and subsequently track each byte's position within the frame accurately.



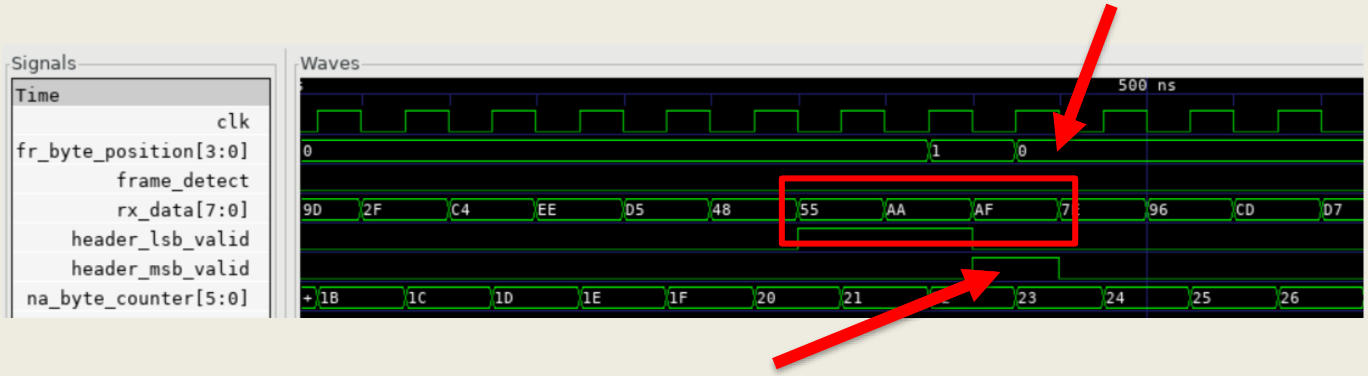
Bug Detection and Analysis

- **Scoreboard:**



```
[--Scoreboard--] Mismatch detected!
Current time is: 480
Expected Frame Detect: 0, Actual Frame Detect: 0
Expected Byte Position: 0, Actual Byte Position: 1
[--Scoreboard--] Mismatch detected!
Current time is: 490
Expected Frame Detect: 0, Actual Frame Detect: 0
Expected Byte Position: 1, Actual Byte Position: 0
```

- **Waveform:** It can be observed that while we are counting "illegal" byte (na_byte_counter) a valid header is detected (header_msb_valid = 1), but the byte position within the valid frame is not incremented afterward, leaving fr_byte_position = 0.

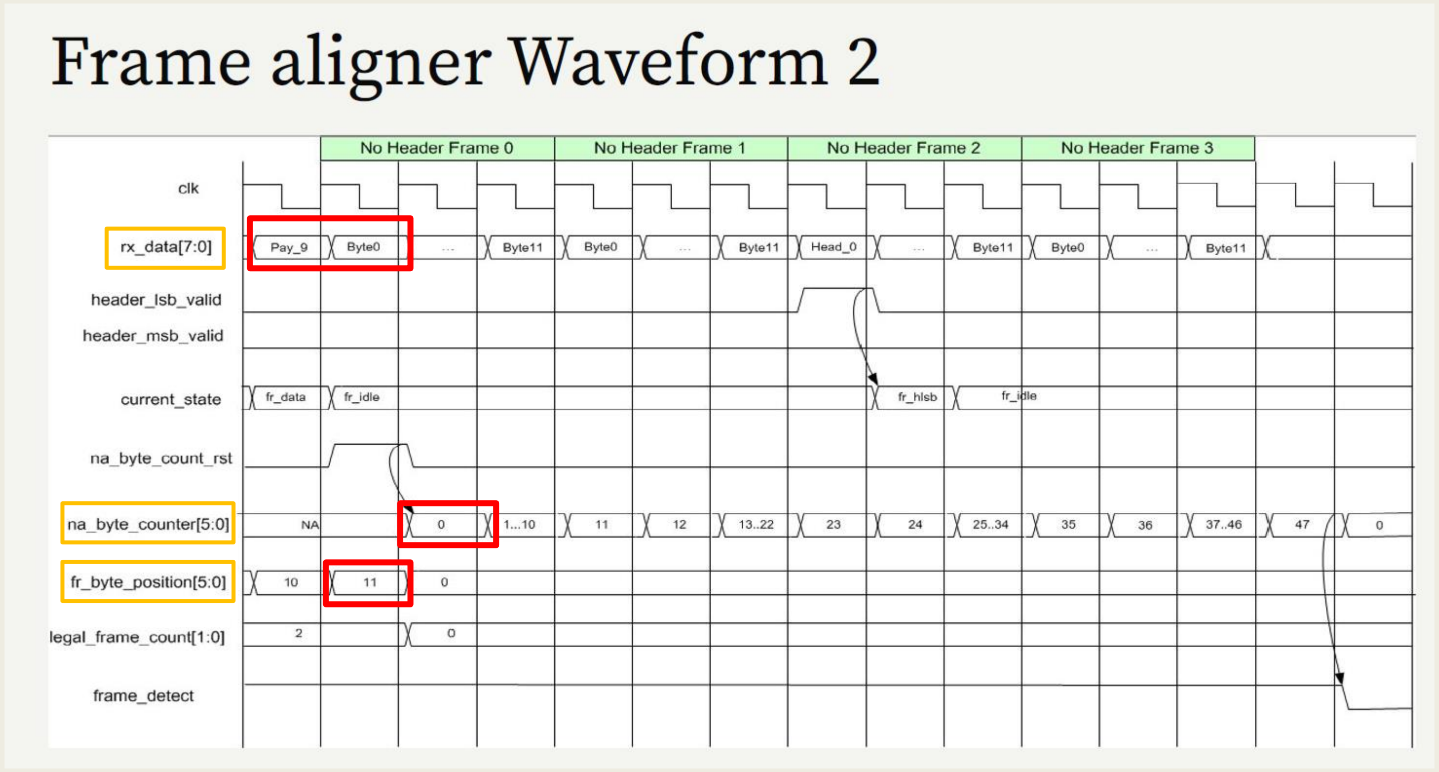


Bug C: na_byte_counter begins counting with the last payload byte, causing frame_detect to be set to 1 prematurely

Detection and Analysis

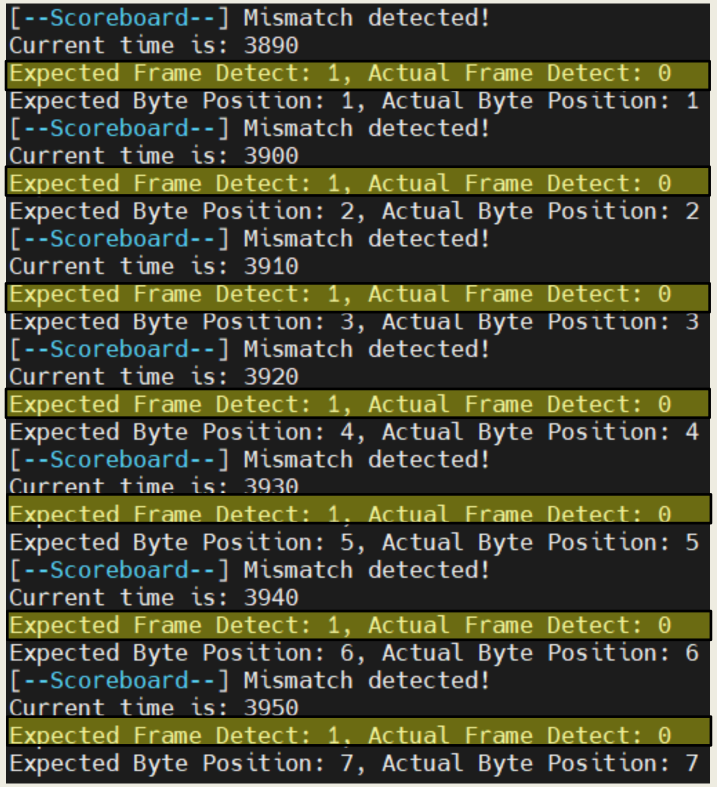
Bug Detection and Analysis

- **Expected Result:** Immediately after the payload of a valid frame ends, if no valid header is detected, we begin counting illegal bytes. Once this count completes, with na_byte_counter reaching 47, frame_detect will reset.



Bug Detection and Analysis

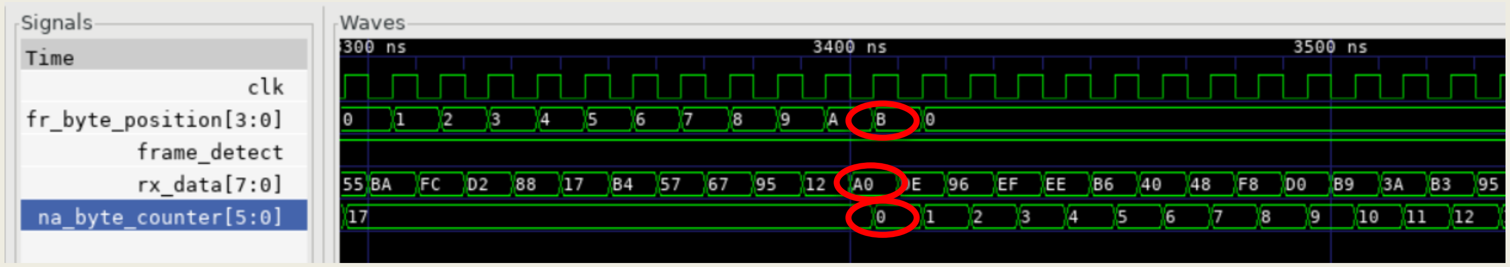
- **Scoreboard:**

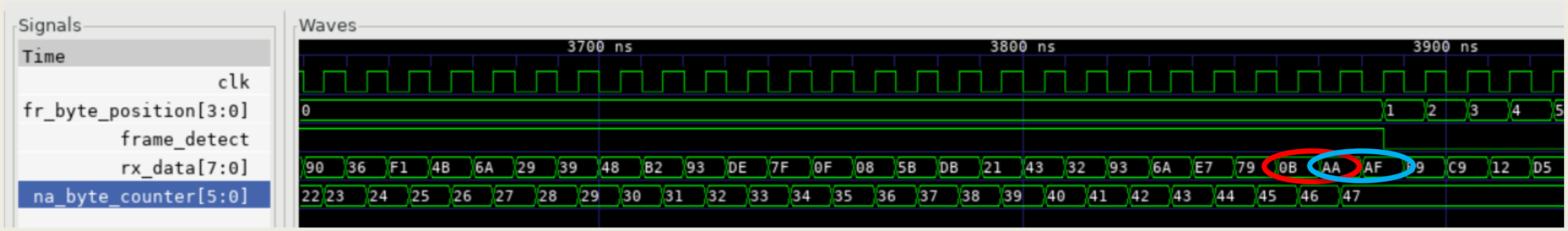


```
[--Scoreboard--] Mismatch detected!
Current time is: 3890
Expected Frame Detect: 1, Actual Frame Detect: 0
Expected Byte Position: 1, Actual Byte Position: 1
[--Scoreboard--] Mismatch detected!
Current time is: 3900
Expected Frame Detect: 1, Actual Frame Detect: 0
Expected Byte Position: 2, Actual Byte Position: 2
[--Scoreboard--] Mismatch detected!
Current time is: 3910
Expected Frame Detect: 1, Actual Frame Detect: 0
Expected Byte Position: 3, Actual Byte Position: 3
[--Scoreboard--] Mismatch detected!
Current time is: 3920
Expected Frame Detect: 1, Actual Frame Detect: 0
Expected Byte Position: 4, Actual Byte Position: 4
[--Scoreboard--] Mismatch detected!
Current time is: 3930
Expected Frame Detect: 1, Actual Frame Detect: 0
Expected Byte Position: 5, Actual Byte Position: 5
[--Scoreboard--] Mismatch detected!
Current time is: 3940
Expected Frame Detect: 1, Actual Frame Detect: 0
Expected Byte Position: 6, Actual Byte Position: 6
[--Scoreboard--] Mismatch detected!
Current time is: 3950
Expected Frame Detect: 1, Actual Frame Detect: 0
Expected Byte Position: 7, Actual Byte Position: 7
```

Bug Detection and Analysis

- **Waveform:**
  - The diagram shows that as the fr_byte_position is at 11 (0xB) is counted, the counting of the first byte in the illegal sequence (0x0) begins simultaneously.
  - We also observe that in bytes 46 and 47, the design identifies 0x0B and 0xAA, whereas the reference model identifies them as bytes 45 and 46, with byte 47 as 0xAF. Therefore, the bytes in positions 46 and 47 form a valid header (0xAFAA), which should reset the illegal byte counter and keep frame_detect set to 1.


Signals
Waves
Time
clk
fr_byte_position[3:0]
frame_detect
rx_data[7:0]
na_byte_counter[5:0]
3300 ns
3400 ns
3500 ns


Signals
Waves
Time
clk
fr_byte_position[3:0]
frame_detect
rx_data[7:0]
na_byte_counter[5:0]
3700 ns
3800 ns
3900 ns