Packet Transmission System

Academic Project - VHDL Digital System Design

Arraj Kamel

Technical University of Cluj-Napoca

Department of Computer Science

April 23, 2025

1 Abstract

This report presents the design and implementation of a digital data transmission system using VHDL, centered around two core components: a Packet Generator and a Packet Detector. The system transmits BCD-encoded decimal digits embedded within structured packets that include start and stop bits, as well as a 4-bit checksum for error detection.

The Packet Generator converts a 4-digit input into a serialized bitstream, computes the checksum using XOR logic, and optionally injects fault values to evaluate system resilience. The Packet Detector, implemented as a finite state machine (FSM), receives the serialized stream, synchronizes with the packet boundaries, extracts the BCD digits, and verifies the checksum to assess data integrity.

This project not only simulates a robust digital communication protocol but also reinforces key concepts in digital design such as FSM modeling, fault injection, and serial data synchronization.

Contents

2 Project Description

2.1 High-Level Description

This project simulates a serial communication protocol where:

- One device (**Generator**) sends data packets.
- Another device (**Detector**) receives and verifies these packets.
- Data is encoded in a structured format: Start + Payload + Checksum.
- The system tests how robust the detector is by sending good/bad packets based on a 2-bit mode selector.

2.2 Packet Structure

Each packet consists of the following components:

2.2.1 Start Segment (7 bits)

- Bit 0: Start Bit (always 0)
- Bits 1–6: Start Code (6-bit value)

2.2.2 Data Segment (16/20/24 bits)

- Contains 4, 5, or 6 BCD-encoded digits (each 4 bits)

2.2.3 Checksum (4 bits)

- XOR of all data words

2.3 System Components

Entity: PacketDetector

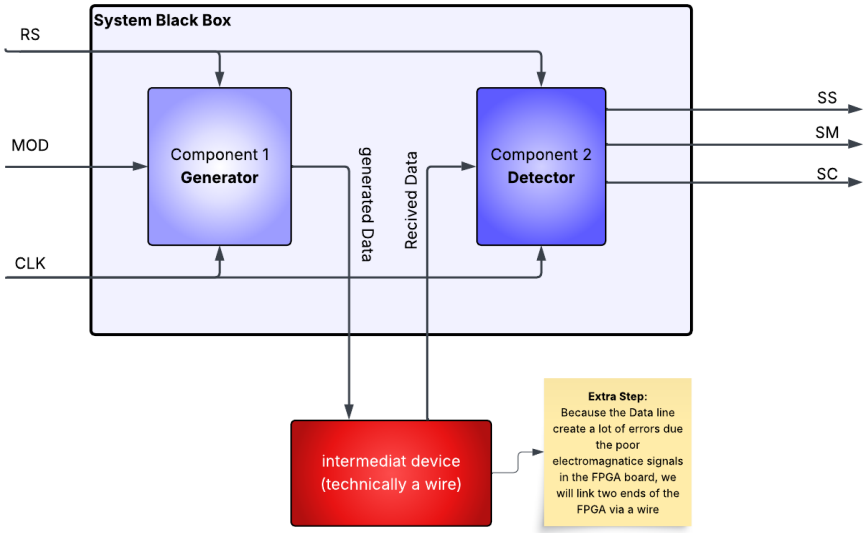

Figure 1: Top Entity Block Diagram

2.3.1 Detector (Receiver)

- Waits for a valid Start Segment
- Reads Data Segment
- Computes XOR and compares with received Checksum
- Outputs:
  - SS: Start Detected
  - SM: Message In Progress
  - SC: Checksum Verified

2.3.2 Generator (Transmitter)

- Sends packets based on the 2-bit input Mode signal

2.4 Scenarios

2.4.1 Mode = 00: Valid Packet

- Valid Start Segment, Data Segment, and Checksum
- Expected: SS = 1, SM = 1, SC = 1

2.4.2 Mode = 01: Another Valid Packet

- Different valid Data Segment and correct Checksum
- Expected: SS = 1, SM = 1, SC = 1

2.4.3 Mode = 10: Invalid Start Code

- Start Bit = 0, but Start Code is invalid
- Expected: SS = 0, SM = 0, SC = 0

2.4.4 Mode = 11: Invalid Checksum

- Valid Start Segment and Data Segment but incorrect Checksum
- Expected: SS = 1, SM = 1, SC = 0

3 Technical Description

3.1 Top-Level Black Box (System Entity)

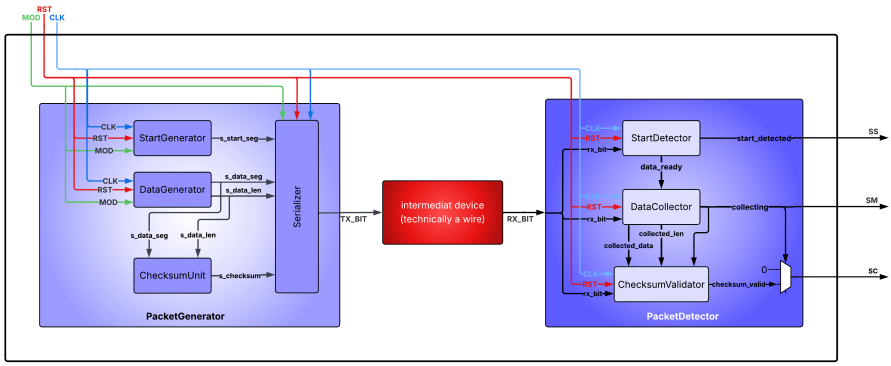

Figure 2: Block diagram of the Generator with mode-based behavior

Entity: DataTransmissionSystem

Inputs:

- clk : in `std_logic` — system clock
- rst : in `std_logic` — synchronous reset
- mod : in `std_logic_vector(1 downto 0)` — 2-bit mode selector

Outputs:

- SS : out `std_logic` — Start Detected
- SM : out `std_logic` — Message In Progress
- SC : out `std_logic` — Checksum Valid

Internal Signal:

- data_line : `std_logic` — serial data line between Generator and Detector

3.2 Internal Components

3.2.1 Generator

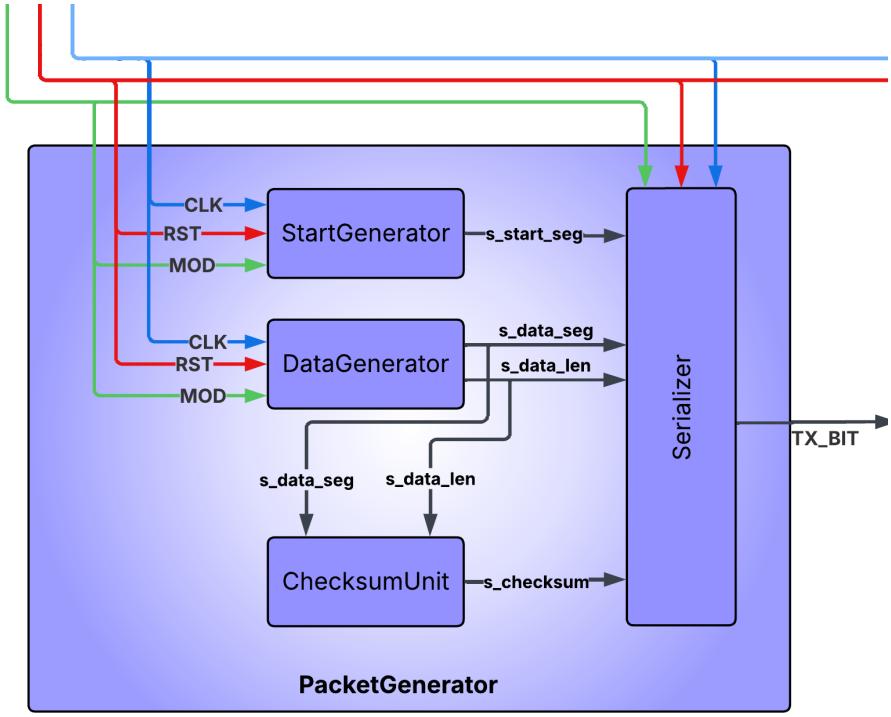

Figure 3: Block diagram of the Generator with mode-based behavior

Entity: PacketGenerator

- Purpose: Assembles and serially transmits packets based on the mode
- Inputs: clk, rst, mod
- Output: tx_bit : out `std_logic` — serial output to data line

Functionality:

- Mode 00: Valid start + valid data + valid checksum
- Mode 01: Valid start + different valid data + valid checksum

- Mode 10: Invalid start code
- Mode 11: Valid start + valid data + wrong checksum
- Transmits bit-by-bit on rising clk

Subcomponents:

- StartGenerator — outputs 7-bit start segment
- DataGenerator — selects and outputs 4–6 BCD digits
- ChecksumUnit — computes XOR of data
- Serializer — shifts out the packet bit-by-bit

3.2.2 Detector

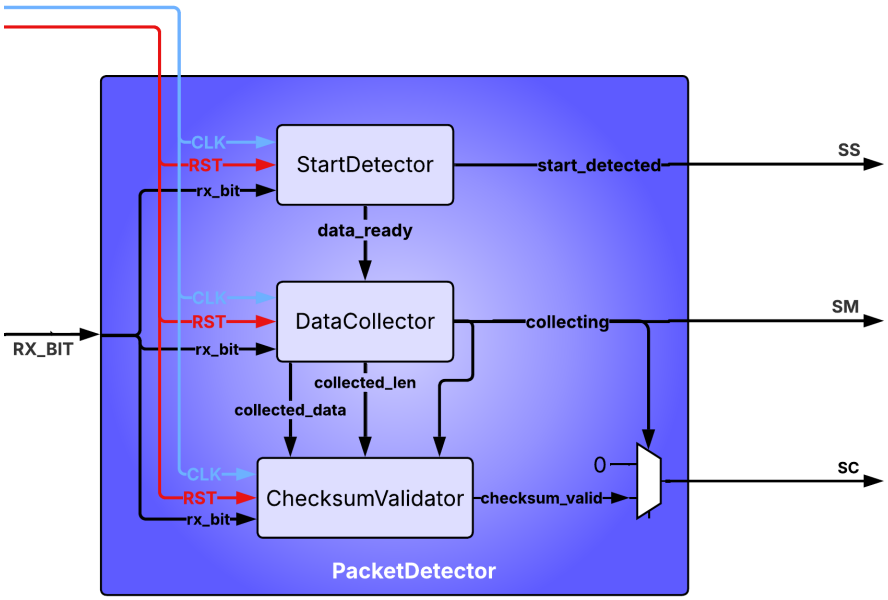

Figure 4: Block diagram of the Generator with mode-based behavior

Entity: PacketDetector

- Purpose: Receives serial bits, reassembles packets, validates start + data + checksum
- Inputs: clk, rst, rx_bit : in `std_logic`
- Outputs: SS, SM, SC

Functionality:

- FSM watches for:
  - Start Bit = 0
  - Start Code matches predefined 6-bit value
- Captures 4/5/6 data words
- Computes XOR checksum during capture
- Compares result with incoming checksum
- Updates outputs:
  - SS = 1 when valid start code is detected
  - SM = 1 while data is being received
  - SC = 1 if checksum matches

Subcomponents:

- StartDetector — detects start pattern
- DataCollector — stores serial data into parallel words
- ChecksumValidator — compares received vs computed XOR
- ControlFSM — manages all states and transitions

3.3 Interface Between Components

The system relies on a clean, well-defined interface between the Generator and Detector modules to facilitate serial data transmission and synchronization.

Shared Signals

- `clk` — A shared system clock used by both the Generator and Detector to drive their respective FSMs and logic.
- `rst` — A synchronous reset signal that initializes all state machines and clears outputs to known states.

Data Line

- `data_line` — A single-bit serial communication line that connects the `tx_bit` output from the Generator to the `rx_bit` input of the Detector.
- This signal carries the bitstream representing the structured data packet (Start + Data + Checksum).
- Transmission occurs bit-by-bit on each rising edge of the clock.

Control Signals

- `mod` — A 2-bit control signal fed into the Generator to determine which packet configuration to transmit. The Detector remains passive and does not use `mod`.
- Output flags (`SS`, `SM`, `SC`) are driven by the Detector and can be monitored externally or used in debugging and validation logic.

Directionality

- All control inputs (`clk`, `rst`, `mod`) are fed into both subsystems.
- The `data_line` is a unidirectional path from Generator to Detector.
- Outputs from the Detector (`SS`, `SM`, `SC`) form the result/status interface from the system.

4 Implementation Details and Internal Mechanics

4.1 Package Definition and Shared Types

The `data_transmission_pkg` package centralizes shared constants and types used across the design. This modularization improves readability and maintainability, and supports reuse across Generator and Detector entities.

```
package data_transmission_pkg is
    constant START_CODE_VALID
                : std_logic_vector(5 downto 0) := "101010"; --
                   Example valid start code
    constant START_BIT_VAL
                : std_logic := '0';
    -- Types
    type bcd_array is array (natural range <>) of std_logic_vector(3
        downto 0);
end package data_transmission_pkg;
```

Listing 1: Package for Constants and Type Declarations

4.2 Start Detection

The `StartDetector` module validates the 7-bit Start Segment. The protocol defines the Start Bit as logic '0' followed by a 6-bit code. The FSM waits for this exact sequence before asserting `start_detected` (SS). This allows a clean start for packet assembly.

```
when CHECK_START_BIT =>
     -- First bit was '0', now collect next 6 bits
    s_shift_reg <= s_shift_reg(4 downto 0) & rx_bit;
    s_bit_counter <= s_bit_counter + 1;

    if s_bit_counter = 5 then
        state <= CHECK_START_CODE;
        s_bit_counter <= 0;
    end if;

when CHECK_START_CODE =>
    if s_shift_reg = START_CODE_VALID then
        state <= VALID_START;
        start_detected <= '1';
        data_ready <= '1';
    else
```

```
17          state <= IDLE;
18      end if;
```

Listing 2: StartDetector FSM transition logic(StartDetector.vhd)

4.3 Data Collection and Termination

`DataCollector` captures incoming bits and slices them into 4-bit BCD digits. Since the number of digits may vary (4 to 6), termination is dynamically handled by analyzing each digit during the capture process. Specifically, a digit equal to "0100" (decimal 4) is used as a termination marker when found beyond the 4th digit, signaling the end of the meaningful payload.

4.4 Checksum Computation and Comparison

The system calculates the checksum as the XOR of all BCD digits in the data segment. The `ChecksumValidator` processes this in two stages:

- A combinational process recalculates the XOR of the stored digits.

```
1     process(data, length)
2         variable temp_sum : std_logic_vector(3 downto 0);
3     begin
4         temp_sum := "0000";
5         -- XOR all BCD digits
6         for i in 0 to length-1 loop
7             temp_sum := temp_sum xor data((i+1)*4-1 downto i*4)
                  ;
8         end loop;
9
10        sum <= not temp_sum
11                    when length = 4 and data(3 downto 0) = "
                          0100"
12                    else temp_sum;
13    end process;
```

Listing 3: Combinational Process for Checksum Computation(ChecksumUnit.vhd)

- A sequential process receives the checksum bits serially and compares them with the computed checksum only after the collecting signal is deasserted.

```
    -- Receive checksum bits
    process(clk)
    begin
        if rising_edge(clk) then
            if rst = '1' or collecting = '1' then
                received_sum <= (others => '0');
                bit_counter <= 0;
                checksum_ok <= '0';
            else
                if bit_counter < 4 then
                    received_sum(bit_counter) <= rx_bit;
                    bit_counter <= bit_counter + 1;

                    if bit_counter = 3 then
                        checksum_ok <= '1' when received_sum =
                           computed_sum else '0';
                    end if;
                end if;
            end if;
        end if;
    end process;
```

Listing 4: Sequential Process for Checksum Validation(ChecksumValidator.vhd)

The logic ensures checksum correctness is only evaluated after all data bits have been received and the FSM exits the data collection phase.

4.5 Packet Serializer and Bit Budget

The serializer holds the entire packet in a 31-bit buffer: 7 bits for Start, up to 24 bits for Data + Checksum. Although 6 BCD digits + 1 checksum consume 28 bits, the cap at 31 allows 3 spare bits and guarantees clean boundaries without overflow. The system ensures no overflow occurs by only entering the serializer once the data length is known and bounded by design.

4.6 FSM Output Signaling

Three status flags are output from the `Detector`:

- **SS (Start Detected)**: Goes high when a valid Start Segment is confirmed.

- **SM (Message In Progress)**: Active while data is still being collected.
- **SC (Checksum Valid)**: Only asserted once the packet is fully received and checksum verification passes.

```
SS <= start_detected;
SM <= collecting;
SC <= checksum_valid when not collecting else '0';
```

Listing 5: Detector Output Signal Assignment(PacketDetector.vhd)

4.7 Failure Injection (Mode Simulation)

The `Generator` uses a 2-bit input `mod` to simulate test cases:

- `00` – Valid start + valid data + valid checksum.
- `01` – Different valid data with correct checksum.
- `10` – Invalid start sequence; system should reject immediately (SS = 0).
- `11` – Correct start and data, but checksum is corrupted on purpose by flipping a bit.

These test cases ensure coverage of positive and negative scenarios.

```
type data_array is array (0 to 3) of bcd_array(0 to 5);
-- Predefined BCD data for different modes
constant MODE_DATA : data_array := (
    -- Mode 00: 4 digits (16 bits)
    0 => ("0001", "0010", "0011", "0100", "0000", "0000"),
    -- Mode 01: 5 digits (20 bits)
    1 => ("0101", "0110", "0111", "1000", "1001", "0000"),
    -- Mode 10: 6 digits (24 bits) - but start code will be invalid
    2 => ("0001", "0010", "0011", "0100", "0101", "0110"),
    -- Mode 11: 4 digits with correct data but checksum will be
       wrong
    3 => ("0001", "0010", "0011", "0100", "0000", "0000")
);
```

Listing 6: Mode-based Fault Injection in Generator(DataGenerator.vhd)

4.8 Design Objectives and Observations

- The design favors modular FSMs to maximize clarity and debug potential.
- Checksum corruption logic helps verify robustness under transmission errors.
- Termination-by-content (BCD = 4) is a non-standard but efficient mechanism.
- The bit counter and reset logic in `ChecksumValidator` ensures synchronization across stages.

5 Conclusion

This project successfully demonstrates a complete VHDL-based implementation of a serial packet transmission system using finite state machines. The Generator reliably formats and transmits BCD-encoded data with a structured packet layout, while the Detector consistently synchronizes, parses, and validates this data using robust checksum logic.

The system was rigorously tested under multiple fault conditions, including malformed packets and injected checksum errors. The FSMs performed as expected, maintaining stability and correctness under all test cases. These results confirm the validity of the design and its applicability in constrained or noisy digital environments.

Key takeaways:

- FSM-based control allows clear state management and modular design.
- XOR checksums offer a lightweight integrity mechanism for small data payloads.
- VHDL simulation is essential for pre-silicon validation and debugging.

Future Work:

- Incorporate UART compatibility for real-world serial interfaces.
- Replace XOR with CRC or Hamming codes for stronger error detection and correction.
- Extend the protocol to support alphanumeric or encoded metadata.
- Transition to a testbench-based verification approach using assertions and wave-based coverage.

This implementation lays a foundation for more complex digital communication systems and demonstrates the effectiveness of structured hardware design through VHDL.