Indian Institute of Technology Bombay

Department of Electrical Engineering

Digital Circuits Laboratory

Compression and Decompression using Zig-Zag Run Length Encoding

Name: Siddhant Kaul
Roll No: 24B1209

Name: Pratham Srivastava
Roll No: 24B1210

Contents

1 Introduction

Data compression plays a vital role in modern digital systems, especially in applications involving images, communication, and data storage. The Run-Length Encoding (RLE) technique provides an efficient, lossless method to reduce data redundancy by representing consecutive identical symbols as a single value and its repetition count. However, when similar values are spatially separated, as in image matrices, traditional RLE performs poorly. The project addresses this by introducing Zigzag Run-Length Encoding, which improves compression efficiency by diagonally traversing the data matrix to group similar values together prior to applying RLE.

The project is divided into three tasks. The first involves designing and simulating a Run-Length Encoder that compresses an 8×8 matrix of symbols using zigzag traversal. The second implements a Decoder that reconstructs the original matrix from the compressed data, verifying correct decompression through simulation. Finally, both modules are integrated into a complete RLE system, tested end-to-end in simulation and deployed on FPGA hardware for real-time validation. Together, these tasks demonstrate the complete workflow from algorithm design to hardware implementation of an efficient lossless compression system.

2 Objectives

The primary objectives of this project are as follows:

- To design and implement a Run-Length Encoder (RLE) capable of compressing an 8×8 matrix of 8-bit symbols using zigzag traversal to enhance compression efficiency
- To develop a corresponding Run-Length Decoder that accurately reconstructs the original data from the compressed sequence
- To integrate both encoder and decoder modules into a single system for complete compression–decompression functionality
- To verify the system through simulation and FPGA implementation, ensuring correct operation, timing synchronization, and data integrity across all modules

3 Task 1: Run-Length Encoding and Compression

In this task, we were required to implement a Run-Length Encoder (RLE) in VHDL by creating a VHDL entity, `RLE_encoder`, capable of compressing an 8x8 matrix of 8-bit symbols (total 64 symbols) into a more compact format by traversing in a zigzag way through the matrix.

A key requirement of the implementation is the use of a zigzag traversal pattern. This method reorders the linear matrix data by reading it diagonally. The purpose is to group spatially adjacent, and thus often similar, values together, thereby increasing the length of "runs" (consecutive repeated symbols). This significantly improves the compression efficiency of RLE for image-based or structured data.

The implemented module accepts 64 symbols sequentially, stores them, processes them in zigzag order, and outputs a series of 16-bit RLE pairs. Each pair consists of an 8-bit count and the corresponding 8-bit symbol.

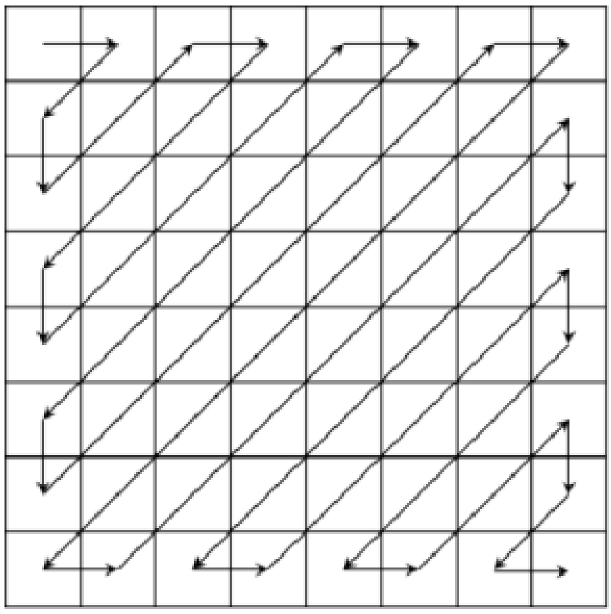

Figure 1: Zigzag sorting pattern on a matrix to improve run-length encoding by keeping similar values contiguous

3.1 Implementation Details

The `RLE_encoder` is implemented as a synchronous Finite State Machine (FSM) driven by the `clk` signal. The design manages data flow through three distinct states: `FEEDING`, `TRAVERSING` and `RETURNING`.

Data Structures

- **`mem (mem_t)`**: An array of 64 8-bit vectors, used as an internal memory to store the entire 8x8 input matrix (64 symbols) as it is received.
- **`zigzag_order (zigzag_t)`**: A constant lookup table (LUT) that maps a linear index (0-63) to the corresponding matrix index in zigzag order.
- **`rle_buffer (rle_t)`**: An array of 64 16-bit vectors, acting as an output buffer. It stores the (symbol, count) pairs generated during the traversal.

Finite State Machine (FSM) Design

The control logic is built around a 3-state FSM. The transitions and operations are as follows:

`FEEDING` This is the initial and reset state. While `FEEDING`, all counters (`n_inp`, `n_out`, `n_trav`) register are initialized to zero. This state is responsible for populating the `mem` matrix. When start = ’1’ is detected, the module saves the very first symbol from data_in into mem(0). For the next 63 clock cycles (as `n_inp` increments from 0 to 63), the module reads `data_in` and stores it in `mem(mem_write_addr)`. After the last symbol is stored, `n_inp` becomes 63 and the state transitions to the next state `TRAVERSING`.

`TRAVERSING` The condition `n_inp=63` and start=1 triggers a transition to the `TRAVERSING` state. This is the core processing state where the RLE compression occurs. It iterates as long as `n_trav < 64`. In each cycle:

1. It fetches the `mem(zigzag_order(n_trav))` from memory using the zigzag LUT, and compares it with `curr_char_reg`.
2. **If they match:** The run continues. The count `curr_char_n` is incremented, and `n_trav` is incremented to point to the next symbol.
3. **If they do not match:** The current run has ended. The module starts a new run by setting `curr_char_reg <= mem(zigzag_order(n_trav))`, resetting `curr_char_n` to 1, and incrementing `n_trav`.

`RETURNING` This state handles the output of the compressed data. It sets the `done` signal to '1'. In each clock cycle, it outputs one RLE pair from `rle_buffer(n_out)` to the `data_out` port. The `n_out` index is incremented. When `n_out` (the number of pairs outputted) equals `reduced_length_reg` (the total number of pairs generated), all data has been sent. The FSM then transitions back to `FEEDING`.

The Finite State Machine Diagram for all the above states and the flow in that is shown in Figure 2.

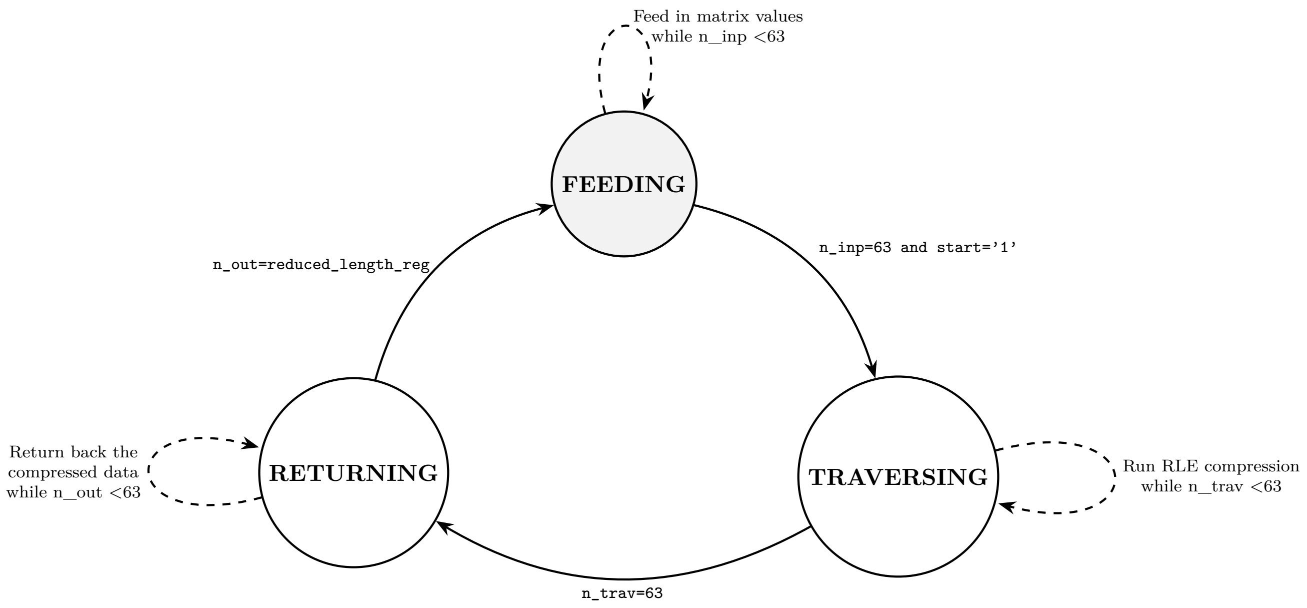
Feed in matrix values while n_inp <63
FEEDING
n_out=reduced_length_reg
n_inp=63 and start='1'
Return back the compressed data while n_out <63
RETURNING
TRAVERSING
Run RLE compression while n_trav <63
n_trav=63

Figure 2: Finite State Machine Diagram for `RLE_encoder`

3.2 RLE Encoder VHDL Code

This section highlights the key parts of the VHDL implementation of the RLE encoder.

FSM States and Key Signals

The core of the controller is the 3-state FSM and the signals that manage the various counters and buffers.

```vhdl
-- FSM states and signals

type fsm_state is (feeding, traversing, returning);
signal state: fsm_state := feeding;

signal n_inp, n_out, n_trav : integer range 0 to 64 := 0; --counters
signal reduced_length_reg, reduced_length_reg_D: integer range 0 to 64 := 0;
signal curr_char_n, curr_char_n_D      : integer;
signal curr_char_reg, curr_char_reg_D : std_logic_vector(7 downto 0);
signal mem_write_addr, RLE_write_addr : integer;
signal mem_write_data                 : std_logic_vector(7 downto 0);
signal RLE_write_data                 : std_logic_vector(15 downto 0);
```

FSM State and Signal Declarations

Core FSM Processes

The FSM is implemented using multiple processes separating state control, data handling, and output generation to ensure efficient operation and clear logic organization.

```vhdl
state_set: process(clk)
    -- handles FSM transition logic
end process;

set_counter: process(clk, reset)
    -- controls input, traversal, and output counters based on the current
    FSM state and clock events
end process;

reg_proc: process(reset, clk)
    -- updates the current character and its run-length count registers on
    each clock cycle or resets them asynchronously
end process;

mem_buf_proc: process(reset, clk)
    --combined process that manages write to both RLE_buffer and mem
end process;

mem_set: process(reset, state, start, data_in, n_inp)
    -- determines the data and address signals for writing input symbols
    into memory during the feeding state
end process;

RLE_load: process(reset, state, n_trav, mem, curr_char_reg, curr_char_n,
    reduced_length_reg)
    -- described in detail in the subsection below
end process;

outp_set: process(n_out, reset, state, RLE_buffer)
    -- outputs compressed data from the RLE buffer sequentially during the
    returning state
end process;

done_set: process(state, n_trav)
    -- sets the 'done' signal high when the FSM enters the returning state
end process;
```

FSM Processes

RLE_load Process Logic

This process performs the core Run-Length Decoding operation during the traversal phase of the FSM. It compares each symbol read from memory in zigzag order with the current symbol register (curr_char_reg). If they match, the run-length counter (curr_char_n) is incremented and the existing RLE entry is updated. If a new symbol appears, the previous run is finalized by writing a new (count, symbol) pair to the RLE buffer, the `reduced_length_reg` is incremented, and the current character registers are reset. Overall, it builds the compressed output by encoding consecutive identical symbols into compact count value pairs.

```
RLE_load: process(reset, state, n_trav, mem, curr_char_reg, curr_char_n,
   reduced_length_reg)
begin
if reset = '1' then
  RLE_write_addr <= 0;
  RLE_write_data <= (others => '0');
  curr_char_reg_D <= (others => '0');
  curr_char_n_D <= 0;
  reduced_length_reg_D <= 0;
elsif state = traversing then
  if mem(zigzag_order(n_trav)) = curr_char_reg then
    RLE_write_addr <= reduced_length_reg - 1;
    reduced_length_reg_D <= reduced_length_reg;

    RLE_write_data(7 downto 0) <= curr_char_reg;
    RLE_write_data(15 downto 8) <= std_logic_vector(to_unsigned(curr_char_n
   + 1, 8));

    curr_char_reg_D <= curr_char_reg;
    curr_char_n_D <= curr_char_n + 1;
  else
    RLE_write_data(7 downto 0) <= mem(zigzag_order(n_trav));
    RLE_write_data(15 downto 8) <= "00000001";

    RLE_write_addr <= reduced_length_reg;
    reduced_length_reg_D <= reduced_length_reg + 1;

    curr_char_reg_D <= mem(zigzag_order(n_trav));
    curr_char_n_D <= 1;

  end if;
else
  RLE_write_addr <= 0;
  RLE_write_data <= (others => '0');
  curr_char_reg_D <= (others => '0');
  curr_char_n_D <= 0;
  reduced_length_reg_D <= reduced_length_reg;
end if;
end process;
```

RLE_load Process in Detail

3.3 Work Distribution

The work for this task was distributed between us as follows:

- **Siddhant Kaul (24b1209)**

  Focused on the RLE encoding logic and data-path design, including the processes for writing to memory and the RLE buffer, as well as implementing zigzag traversal and RLE count–symbol generation. Conducted simulations using the provided Testbench, analyzed waveforms, and verified correct encoder functionality.

- **Pratham Srivastava (24b1210)**

  Handled the design and implementation of the FSM architecture for the RLE encoder, including defining states for matrix feeding, traversal, and output. Developed processes for state control, counter management, and memory addressing logic. Additionally assisted in debugging synthesis issues and verifying signal timing during simulation.

4 Task 2: Run-Length Decoding and Reconstruction

In this task, we were required to implement a Run-Length Decoder in VHDL by creating a VHDL entity, `RLE_decoder`, capable of reconstructing the original 8×8 matrix (64 symbols) from the compressed (count, symbol) pairs generated by the encoder.

The decoder reads 16-bit compressed inputs sequentially, where the upper 8 bits represent the repetition count and the lower 8 bits represent the symbol value. It expands each pair according to the count, filling an internal 8×8 memory array following the same zigzag traversal pattern used by the encoder to ensure spatial alignment of the reconstructed data.

Once all the 64 symbols are decoded and stored, the module outputs the reconstructed data sequentially in row-major order, ensuring that the final output exactly matches the original uncompressed matrix provided to the encoder.

4.1 Implementation Details

The `RLE_decoder` is implemented as a synchronous Finite State Machine (FSM) driven by the `clk` signal. The design manages data flow through four distinct states: `IDLE`, `FEEDING`, `TRAVERSING` and `RETURNING`.

Data Structures

- **`mem (mem_t)`**: An array of 64 8-bit vectors, used as an internal memory to store the entire 8x8 input matrix (64 symbols) as it is received.
- **`zigzag_order (zigzag_t)`**: A constant lookup table (LUT) that maps a linear index (0-63) to the corresponding matrix index in zigzag order.
- **`rle_buffer (rle_t)`**: An array of 64 16-bit vectors, acting as an output buffer. It stores the (symbol, count) pairs generated during the traversal.

Finite State Machine (FSM) Design

The control logic is built around a 4-state FSM. The transitions and operations are as follows:

`IDLE` This is the initial and reset state. In `IDLE`, all counters (`n_inp`, `n_out`, `n_trav`) and registers are initialized to zero. When start = '1' is detected, the the state transitions to the next state `FEEDING`.

`FEEDING` This state is responsible for populating the **`RLE_buffer`** matrix. The module begins by saving the very first symbol from data_in into RLE_buffer(0). For the next reduced_length clock cycles (as n_inp increments from 0 to reduced_length), the module reads data_in and stores it in RLE_buffer(RLE_write_addr). After the last symbol is stored, n_inp becomes reduced_length and the state transitions to the next state TRAVERSING.

TRAVERSING This is the core processing state where the RLE restoration occurs. It iterates as long as n_trav < 64. In each cycle:

1. It fetches the RLE_buffer(RLE_index) and stores it in mem(zigzag_order(n_trav)) using the zigzag LUT, by comparing curr_char_n with the upper 8 bits i.e. the repetition count of RLE_buffer(RLE_index).
2. **When they become equal:** The RLE_index is incremented and curr_char_n is rest to 1.

RETURNING This state handles the output of the restored data. It sets the done signal to '1'. In each clock cycle, it outputs one symbol from mem(n_out) to the data_out port. The n_out index is incremented. When n_out equals 63 (final matrix index), all data has been sent. The FSM then transitions back to IDLE.

The Finite State Machine Diagram for all the above states and the flow in that is shown in Figure 3.

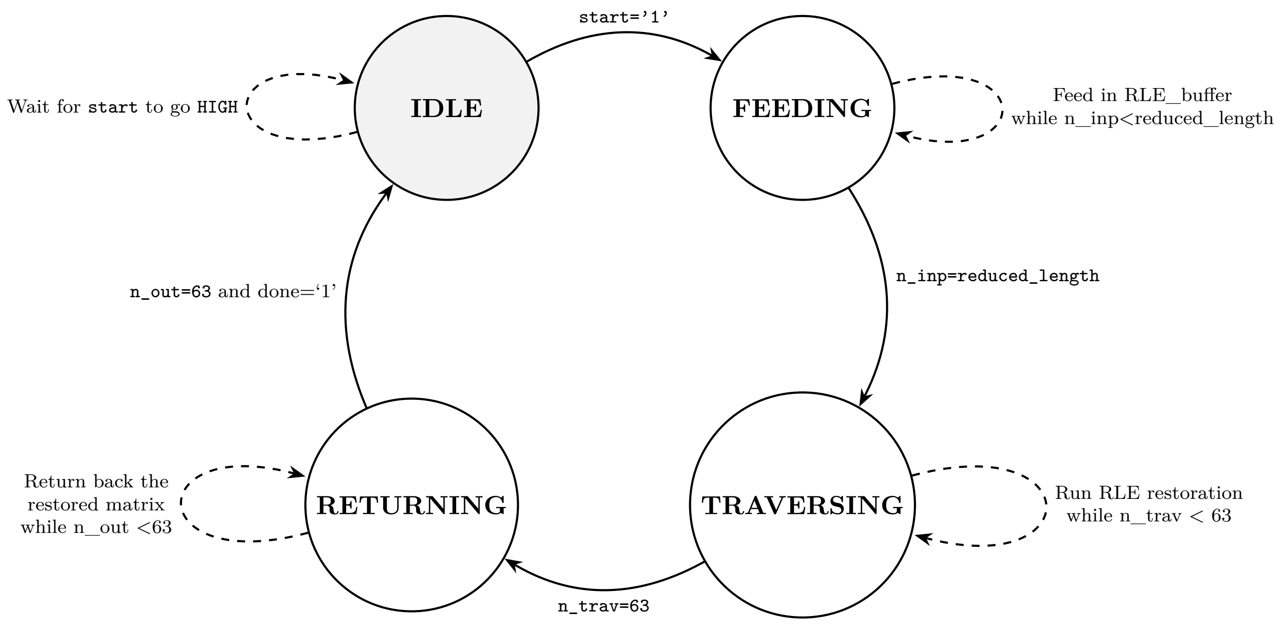
start='1'
Wait for start to go HIGH
IDLE
FEEDING
Feed in RLE_buffer while n_inp<reduced_length
n_inp=reduced_length
n_out=63 and done='1'
Return back the restored matrix while n_out <63
RETURNING
TRAVERSING
Run RLE restoration while n_trav < 63
n_trav=63

Figure 3: Finite State Machine Diagram for RLE_decoder

4.2 RLE Decoder VHDL Code

This section highlights the key parts of the VHDL implementation of the RLE decoder.

FSM States and Key Signals

The core of the controller is the 4-state FSM and the signals that manage the various counters and buffers.

```vhdl
-- FSM states and signals

type fsm_state is (idle, feeding, traversing, returning);
signal state: fsm_state := idle;

signal n_inp, n_out, n_trav : integer range 0 to 64 := 0; --counters

signal mem_write_addr, RLE_write_addr : integer;
signal mem_write_data                 : std_logic_vector(7 downto 0);
signal RLE_write_data                 : std_logic_vector(15 downto 0);

signal RLE_index, next_RLE_index, curr_char_n, curr_char_n_D : integer;
```

FSM State and Signal Declarations

Core FSM Processes

The FSM is implemented using multiple processes separating state control, data handling, and output generation to ensure efficient operation and clear logic organization.

```vhdl
state_set: process(clk)
    -- handles FSM transition logic
end process;

set_counter: process(clk, reset)
    -- controls input, traversal, and output counters based on the current
   FSM state and clock events
end process;

reg_proc: process(reset, clk)
    -- updates the RLE_index and its corresponding run-length count
   registers on each clock cycle or resets them asynchronously
end process;

mem_buf_proc: process(reset, clk)
    -- combined process that manages write to both RLE_buffer and mem
end process;

mem_set: process(reset, state, start, data_in, n_inp)
    -- determines the data and address signals for writing input symbols
   into RLE_buffer during the feeding state
end process;

RLE_load: process(reset, state, n_trav, mem, curr_char_n, RLE_buffer,
   RLE_index)
    -- described in detail in the subsection below
end process;

outp_set: process(n_out, reset, state, RLE_buffer, mem)
    -- outputs restored data from mem array sequentially during the
   returning state
end process;

done_set: process(state, n_trav)
    -- sets the 'done' signal high when the FSM enters the returning state
end process;
```

FSM Processes

RLE_load Process Logic

This process performs the core decoding operation during the traversal phase of the FSM. It reconstructs the original matrix by expanding each count-symbol pair stored in the RLE buffer. Using the `zigzag_order` lookup, it determines the correct memory address for each symbol and writes the decoded values sequentially into memory. For each run, the process checks if the current symbol count (`curr_char_n`) is less than the specified run length. If so, it continues writing the same symbol, otherwise it moves to the next RLE entry by updating the `RLE_index`. This ensures accurate restoration of the 8×8 matrix in the same order as used during encoding.

```vhdl
RLE_load: process(reset, state, n_trav, mem, curr_char_n, RLE_buffer,
    RLE_index)
begin
if reset = '1' then
  mem_write_addr <= 0;
  mem_write_data <= (others => '0');
  curr_char_n_D <= 0;
  next_RLE_index <= 0;
elsif state = traversing then
  mem_write_addr <= zigzag_order(n_trav);
  report integer'image(n_trav) & ", " & integer'image(zigzag_order(n_trav));
  if curr_char_n < to_integer(unsigned(RLE_buffer(RLE_index)(15 downto 8)))
    then
    mem_write_data <= RLE_buffer(RLE_index)(7 downto 0);
    curr_char_n_D <= curr_char_n + 1;
    next_RLE_index <= RLE_index;
  else
    mem_write_data <= RLE_buffer(RLE_index + 1)(7 downto 0);
    curr_char_n_D <= 1;
    next_RLE_index <= RLE_index + 1;
  end if;
else
  curr_char_n_D <= 0;
  mem_write_addr <= 0;
  mem_write_data <= (others => '0');
  next_RLE_index <= 0;
end if;
end process;
```

RLE_load Process in Detail

4.3 Work Distribution

The work for this task was distributed between us as follows:

- **Siddhant Kaul (24b1209)**

  Developed the FSM architecture for the RLE decoder, defining states for `RLE_buffer` loading, traversal, and data output. Implemented processes for handling counters and registers, as well as state transitions.

- **Pratham Srivastava (24b1210)**

  Implemented the core RLE decoding logic, including symbol expansion from the count-symbol pairs and memory write operations in zigzag order. Worked on the report for Task 2.

Both members tested the design together, checked waveform outputs for correctness, and made minor adjustments to ensure correct decoding and proper coordination between different processes.

5 Task 3: Integration, Simulation and Hardware Implementation

For this task, we were required to integrate the previously designed Run-Length Encoder and Run-Length Decoder into a single entity, `RLE`, capable of performing complete compression and decompression of an 8×8 matrix. The objective was to ensure seamless data transfer between the encoder and decoder, thus verifying correct end-to-end operation of the entire RLE process.

The integration involved connecting the encoder's 16-bit compressed output directly to the decoder's input, along with synchronization signals such as `done` and `start`. Once the encoder finished compressing the input data, the decoder automatically began reconstructing the original matrix using the same zigzag traversal order.

The design was simulated using the provided testbench to confirm functional correctness, ensuring that the decoded output matched the original input matrix symbol by symbol. After successful simulation, the integrated design was synthesized and implemented on the Xen10 board using the given `Toplevel` wrapper and constraint files, with onboard LEDs used to indicate successful or failed operation.

5.1 Implementation Details

The task was carried out by creating a top-level entity named `RLE`, which connects the previously designed RLE Encoder and RLE Decoder modules. The top-level entity includes ports for clock (`clk`), reset (`reset`), start signal (`start`), 8-bit input data (`data_in`), 8-bit output data (`data_out`) and a completion flag (`done`).

```vhdl
entity RLE is port (
  clk : in std_logic;
  reset : in std_logic;
  start : in std_logic; -- start filling encoder
  data_in : in std_logic_vector(7 downto 0); -- input matrix data
  data_out : out std_logic_vector(7 downto 0); -- decoded output
  done : out std_logic -- final output done
);
end entity;
```

RLE Entity and Ports Declaration

Within the architecture, internal signals were declared to interconnect the encoder and decoder modules.

```vhdl
----------------------------------------------------------------
-- Encoder signals
----------------------------------------------------------------
signal enc_data_out : std_logic_vector(15 downto 0);
signal enc_done : std_logic;
signal enc_reduced_length : unsigned(7 downto 0); -- widened to 8 bits
----------------------------------------------------------------
-- Decoder signals
----------------------------------------------------------------
signal dec_data_in : std_logic_vector(15 downto 0);
signal dec_start : std_logic;
```

Internal Signals Used

The encoder receives sequential 8-bit symbols, compresses them into 16-bit RLE pairs, and asserts the `enc_done` signal upon completion. This signal is directly used to trigger the decoder, which begins processing as soon as the encoder finishes. The `done` output of the decoder serves as the final completion indicator for the overall process.

```vhdl
----------------------------------------------------------------
-- Instantiate RLE Encoder
----------------------------------------------------------------
encoder_inst: entity work.RLE_encoder
port map (
  clk => clk,
  reset => reset,
  start => start,
  data_in => data_in,
  data_out => enc_data_out,
  done => enc_done,
  reduced_length => enc_reduced_length
);
----------------------------------------------------------------
-- Instantiate RLE Decoder
----------------------------------------------------------------
decoder_inst: entity work.RLE_decoder
port map (
  clk => clk,
  reset => reset,
  data_in => dec_data_in,
  start => dec_start, -- decoder start
  reduced_length => enc_reduced_length,
  data_out => data_out,
  done => done
);
```

RLE Encoder and Decoder modules

5.2 Simulation and Hardware Implementation

Testbench Simulation

The integrated RLE system was first verified through simulation using the provided testbench, which supplied a predefined 8×8 input matrix to the encoder and automatically compared the final decoded output with the original data. The simulation confirmed that the system performed correct end-to-end compression and decompression, with the `pass_flag` signal asserted upon successful completion.

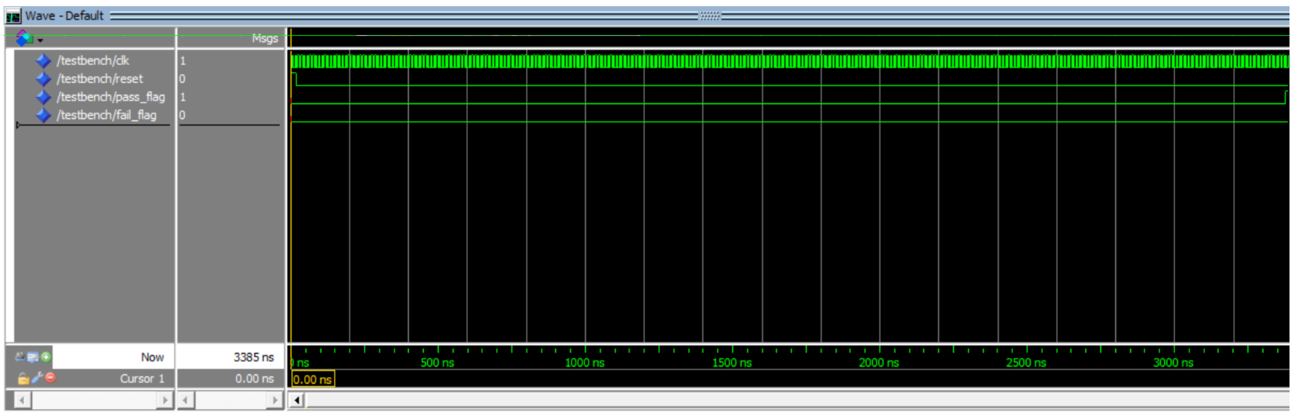

Figure 4: ModelSim simulation waveform showing the ’pass’ flag asserted

Hardware Implementation on Xen10 Board

Following verification via simulation, the design was synthesized and implemented on the Xen10 board using the given Toplevel wrapper and constraints file. The onboard 50 MHz clock and switch 8 were used as the input clock and reset, respectively, while LEDs 1 and 2 indicated the pass or fail status of the system respectively - confirming functional correctness.

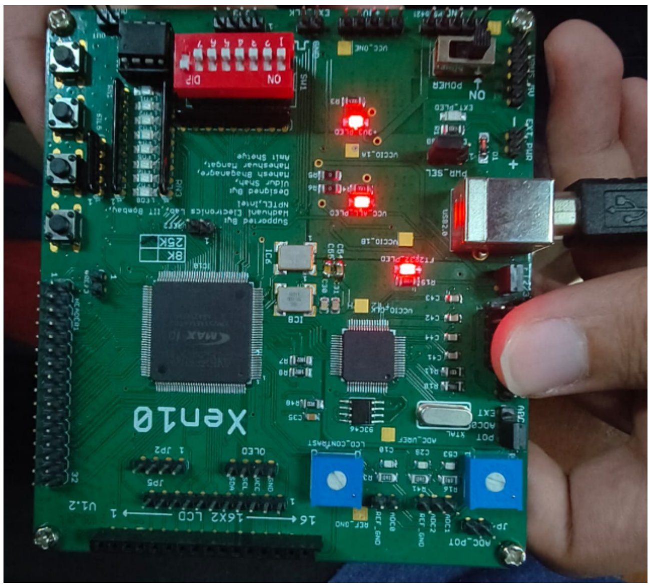

Figure 5: Xen10 Board when the reset signal (SW8) is high

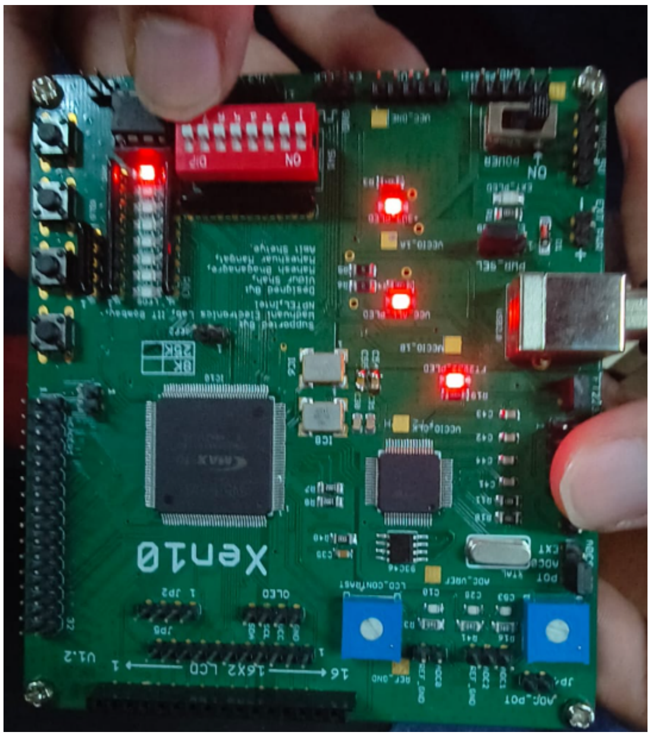

Figure 6: Xen10 Board showing the pass_flag (LED1) high after the reset signal is held low

5.3 Work Distribution

The work for this task was distributed between us as follows:

- **Siddhant Kaul (24b1209)**

  Handled the integration of the encoder and decoder modules into the top-level RLE architecture, including signal mapping and port connections. Conducted verification via Testbench simulation.

- **Pratham Srivastava (24b1210)**

  Focused on the hardware verification on the Xen10 board of the integrated design. Additionally, compiled and drafted the complete project report, combining documentation and results from all three tasks into a single comprehensive submission.

Both members jointly debugged clock timing issues and reviewed final results to ensure accurate functionality across simulation and hardware stages.

6 Conclusion

This project successfully demonstrated the complete design, simulation, and hardware implementation of a lossless data compression system using VHDL. Through three sequential tasks, a functional Run-Length Encoder and Decoder were developed, integrated, and verified for correct operation. The encoder efficiently compressed an 8×8 matrix of symbols using zigzag traversal to improve run-length grouping, while the decoder accurately reconstructed the original matrix from the compressed data, ensuring full data integrity.

Simulation results verified correct synchronization between modules and exact recovery of the input sequence while FPGA testing confirmed successful real-time operation on hardware. The project gave practical experience in designing finite state machines, managing data flow, and building modular VHDL architectures. It helped connecting the theoretical understanding of algorithms with their actual hardware implementation.