EE 214 Digital Circuits Laboratory
Wadhwani Electronics Laboratory
Electrical Engineering IIT Bombay

Project Date: October 30, 2025

Compression and Decompression using Zig Zag Run length Encoding

Instructions

1. **Team Composition:** Each group should consist of two members for the project.
2. **Project Structure:** The project is divided into three main tasks. **Create three separate projects for each task.**
3. **Submission Timeline:** The submission of tasks will be in parts, and for the final submission, you will need to combine all task reports into a single comprehensive report.
4. **Task Reports:** For each individual task, you are required to submit a report detailing the work completed and the distribution of tasks between group members.
5. **Code Restrictions:** You are allowed to make changes in designated VHDL code areas. You may also modify the architecture of the given files (Testbench, or Toplevel). However, do not change any port names in the entity declarations. This is to ensure cohesive functionality during final integration.

Problem Description

Run-Length Encoding is a simple way to compress data by replacing consecutive repeated symbols with a single *value* and a *count*.
Example:

$$\text{AAAAABBCCCC} \quad\longrightarrow\quad (5, A)(2, B)(4, C)$$

Instead of storing every symbol, we just store the symbol once with how many times it repeats.
Original message size: Suppose, for a message of 64 symbols, where each symbol is 8 bits and counter for RLE is also 8 bit.

$$64 \times 8 = 512 \text{ bits}$$

Encoding size: Each run needs:

$$8 \text{ bits (value)} + 8 \text{ bits (count)} = 16 \text{ bits}$$

Best case (all symbols same):

$$\underbrace{AAAAAAAA \cdots A}_{\text{64 times}} \quad\longrightarrow\quad (64, A)$$

Storage needed:

$$8 + 8 = 16 \text{ bits (much smaller than 512)}$$

Worst case (no repetition):

$$ABCDABCD\cdots$$

Each symbol is its own run, so:

$$64 \times 16 = 1024 \text{ bits (larger than 512)}$$

Summary:

Best case: 16 bits Original: 512 bits Worst case: 1024 bits

RLE works well when data has long runs of repetition (images with large flat areas, spaces in text), which makes it extremely suitable for compression on such data. But it can also expand data rapidly when there is little repetition.

Zigzag Sort to Improve Compression

Sometimes in images/data matrices, repeated values are arranged in spatial blocks. Direct RLE (row-by-row) might get broken by row boundaries, leading to shorter runs. Zigzag Sort is a reordering method: you traverse the data matrix in a zigzag pattern (diagonally) so that repeated or similar values stay adjacent in the traversal. This increases the chance of longer runs, hence better compression with RLE. Example: For the given matrix the order of index to do RLE will look as follows - 0, 1, 8 (second row), 16 (third row), 9 (second row), 2 (first row) and so on and so forth.

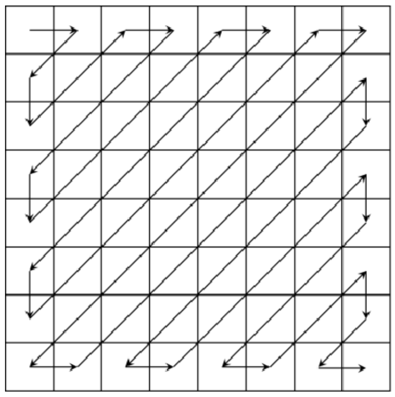

Figure: Zigzag sorting pattern on a matrix to improve run-length encoding by keeping similar values contiguous.

Effect on our 64-symbol example:

- Without zigzag (just linear or row-wise), repeated symbols split when moving to next row → more runs → closer to worst case.
- With zigzag, runs "flow" more smoothly across rows → potentially fewer runs → compression moves closer to best case.
- In ideal matrix where large blocks of same symbol exist, zigzag might reduce run count significantly.

Using zigzag reordering + RLE can significantly reduce the total number of runs in structured data (like images), making compression much more efficient than naive RLE in many practical cases.

Task 1: Run-Length Encoding and Compression (Simulation)

In this task, you are required to implement a Run-Length Encoder (RLE) in VHDL. The top-level entity is already defined for you as given in attached vhdl file:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity RLE_encoder is
    port (
        clk            : in  std_logic;
        reset          : in  std_logic;
        start          : in  std_logic;
        data_in        : in  std_logic_vector(7 downto 0);
        data_out       : out std_logic_vector(15 downto 0);
        done           : out std_logic;
        reduced_length : out unsigned(7 downto 0)
    );
end entity;

architecture arch of RLE_encoder is
```

```
    ----------------------------------------------------------------
    -- Matrix storage
    ----------------------------------------------------------------
    type mem_t is array (0 to 63) of std_logic_vector(7 downto 0);
    signal mem : mem_t;

    ----------------------------------------------------------------
    -- Zigzag order for 8 x 8  matrix
    ----------------------------------------------------------------
    type zigzag_t is array (0 to 63) of integer range 0 to 63;
    constant zigzag_order : zigzag_t := (
        0, 1, 8,
        16, 9, 2,
        3, 10, 17, 24,
        32, 25, 18, 11, 4,
        5, 12, 19, 26, 33, 40,
        48, 41, 34, 27, 20, 13, 6,
        7, 14, 21, 28, 35, 42, 49, 56,
        57, 50, 43, 36, 29, 22, 15,
        23, 30, 37, 44, 51, 58,
        59, 52, 45, 38, 31,
        39, 46, 53, 60,
        61, 54, 47,
        55, 62,
        63
    );

    ----------------------------------------------------------------
    -- RLE Buffer
    ----------------------------------------------------------------
    type rle_t is array (0 to 63) of std_logic_vector(15 downto 0);
    signal rle_buffer : rle_t;
    ----------------------------------------------------------------
    insert all your signal declarations and fsm states here
    ----------------------------------------------------------------


begin
    ----------------------------------------------------------------
    insert your architecture design here
    ----------------------------------------------------------------
end architecture;
```

Description:

- Step 1: Input data is fed symbol by symbol on `data_in`. When `start=1` is observed, the encoder should begin filling an internal 8×8 matrix (64 entries) each with 8 bits of coming data on every clock cycle in given declared matrix as 512 bit memory.
- Step 2: Once the matrix is populated, your fsm should traverse it in a *zigzag order*, as described above. During traversal, repeated symbols must be grouped into 16 bits. Each run is stored as a pair (count Upper 8 bits, symbol Lower 8 bit) and is stored in declared `rle_buffer` of capacity of 1024 bits. If the message fits in less memory then rest of the memory is left uninitialised.
- Step 3: The signal `done = 1` should indicate when `rle_buffer` is ready to output compressed message . At every clock cycle elements are put on `data_out` from `rle_buffer` on `data_out`. Additionally the signal `reduced_length` should output the total number of valid entries in `rle_buffer` generated from the 64 input symbols. One pair (count 8 bit, symbol 8 bit) counts as one valid entry.

Instructions:

- This design does **not** require DUT or Tracefile. Your design will be your top level module, and your Testbench will be used as usual.
- You must design the architecture to manage matrix filling, zigzag traversal, and RLE compression.
- The provided entity declaration, ports, and memory declaration are fixed. Do not modify the interface.
- You are free to choose your own internal design style (state machines, counters, etc.) to achieve correct functionality.
- Use the Testbench attached here.

Task 2: Run-Length Decoding and Reconstruction (Simulation)

In this task, you are required to implement a Run-Length Decoder (RLE) in VHDL. The top-level entity is already defined for you as given in attached vhdl file:

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity RLE_decoder is
    port (
        clk            : in  std_logic;
        reset          : in  std_logic;
        data_in        : in  std_logic_vector(15 downto 0);
        start          : in  std_logic;
        reduced_length : in  unsigned(7 downto 0);
        data_out       : out std_logic_vector(7 downto 0);
        done           : out std_logic
    );
end entity;

architecture rtl of RLE_decoder is

    ----------------------------------------------------------------
    -- Matrix storage
    ----------------------------------------------------------------
    type mem_t is array (0 to 63) of std_logic_vector(7 downto 0);
    signal mem : mem_t;

    ----------------------------------------------------------------
    -- RLE buffer
    ----------------------------------------------------------------
    type rle_t is array (0 to 255) of std_logic_vector(15 downto 0);
    signal rle_buffer : rle_t;

    ----------------------------------------------------------------
    -- Zigzag LUT (same as encoder)
    ----------------------------------------------------------------
    type zigzag_t is array (0 to 63) of integer range 0 to 63;
    constant zigzag_order : zigzag_t := (
        0, 1, 8,
        16, 9, 2,
        3, 10, 17, 24,
        32, 25, 18, 11, 4,
        5, 12, 19, 26, 33, 40,
        48, 41, 34, 27, 20, 13, 6,
        7, 14, 21, 28, 35, 42, 49, 56,
        57, 50, 43, 36, 29, 22, 15,
        23, 30, 37, 44, 51, 58,
        59, 52, 45, 38, 31,
        39, 46, 53, 60,
        61, 54, 47,
        55, 62,
        63
    );

    ----------------------------------------------------------------
    insert all your signal declarations and fsm states here
    ----------------------------------------------------------------

begin

    ----------------------------------------------------------------
    insert your architecture design here
    ----------------------------------------------------------------

end architecture;
```

Description:

- Step 1: The decoder receives the compressed sequence on `data_in`. Each element consists of a pair (count, symbol), where the upper 8 bits represent the repetition count and the lower 8 bits represent the symbol.
- Step 2: When `start=1` is observed, the decoder reads from data_in. Read exactly `reduced_length` number of pairs into its internal `rle_buffer`.
- Step 3: For each entry, the decoder expands the symbol `count` times and places the resulting sequence into the 8×8 matrix memory `mem`. The order of filling follows the *zigzag order* defined in the constant table, identical to the encoder.
- Step 4: Once all 64 entries of the matrix are reconstructed, the signal `done=1` should indicate that the decoded image block is ready. Symbols are output sequentially on `data_out` in normal matrix order (row-major), one symbol per clock cycle.

Instructions:

- This design does **not** require DUT or Tracefile. Your design will be your top level module, and your Testbench will be used as usual.
- You must design the architecture to manage RLE expansion, zigzag addressing, and matrix reconstruction.
- The provided entity declaration, ports, and memory declaration are fixed. Do not modify the interface.
- You are free to choose your own internal design style (state machines, counters, etc.) to achieve correct functionality.
- Use the Testbench attached here.

Task 3: Integration, Simulation, and Hardware Implementation

In this task, you are required to integrate the RLE encoder and decoder into a single top-level design. The provided entity declaration is fixed; **DO NOT** change the names of ports or signals.

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

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;

architecture rtl of RLE is

    ----------------------------------------------------------------
    -- 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;

begin

    ----------------------------------------------------------------
    -- 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
        );

    ----------------------------------------------------------------
    -- Connection logic
    -- Feed encoder output to decoder
    ----------------------------------------------------------------
    dec_data_in <= enc_data_out;
    dec_start   <= enc_done;  -- decoder starts when encoder asserts done

end architecture;
```

Description:

(a) **Simulation Testbench:** Use above code along with provided Testbench and your RLE_encoder and

RLE_decoder codes. The Testbench will send a test matrix and check if the output matrix matches exactly. A pass flag is raised on success else a fail flag is raised.
Note: The entity RLE will be your top level entity for simulation purposes.

(b) **Hardware Implementation Toplevel:** For hardware deployment, include your completed `RLE` entity and Toplevel wrapper given below. Also include the given Constraints file (SDC file) in the project, it is required for correct synthesis and implementation.
Using pinplanning, connect `clk` to onboard 50MHz clock,`reset` to switch 8, and pass and fail flags to LEDs. Compile and test as usual.
Note: The entity Toplevel will be your top level entity for hardware implementation.

Instructions:

- Attach your previously completed encoder and decoder designs without modifying their entity names or port lists.
- Use the following links for testing and implementation:
  - Testbench
  - Toplevel
  - Constraints file
- Ensure that your simulation shows correct compression and decompression for sample input data.
- After successful simulation, proceed to synthesize the design for hardware implementation.