Compression and Decompression using Zigzag Run-Length Encoding

EE 214 Digital Circuits Laboratory
Wadhwani Electronics Laboratory, IIT Bombay

Tushar Yadav (24B1257)
Samiksha PS (24B1258)

November 2025

Contents

Section 1

Project Overview

1.1 Objective

The objective of this project is to design and implement a complete run-length encoding (RLE) compression and decompression system using VHDL. The system uses zigzag reordering to improve compression efficiency for nearby data that show similar patterns (spatial locality), making it particularly suitable for image compression applications. The implementation consists of three integrated modules: RLE_encoder, RLE_decoder, and the top-level RLE integration entity.

1.2 Problem Statement

Run-Length Encoding (RLE) is a fundamental data compression technique that replaces consecutive repeated symbols with a pair consisting of the symbol and its repetition count. For example:

$$\text{AAAAABBBCCCC} \rightarrow (5, \text{A})(3, \text{B})(4, \text{C})$$

However, in structured data such as images, spatial locality plays a important role. By traversing the data in a zigzag pattern (similar to JPEG compression), we can group spatially correlated values together and increase the chances of longer runs and achieving better compression ratios.

1.2.1 Data Compression Analysis

Consider a message of 64 symbols, where each symbol is 8 bits:

- **Original size:** $64 \times 8 = 512$ bits
- **Encoded representation:** $(\textit{count} : 8 \text{ bits}, \textit{symbol} : 8 \text{ bits}) = 16$ bits per run
- **Best case (all symbols identical):** 16 bits
- **Worst case (no repetition):** $64 \times 16 = 1024$ bits

1.3 Project Structure

The project implementation consists of three main components:

1. **Task 1:** Run-Length Encoder (RLE_encoder) Implementation
2. **Task 2:** Run-Length Decoder (RLE_decoder) Implementation
3. **Task 3:** Integration (RLE) and Hardware Deployment

1.4 Key Technical Concepts

1.4.1 Zigzag Ordering

The zigzag traversal pattern for an 8×8 matrix follows diagonal paths to keep spatially adjacent values contiguous in the encoded stream. This significantly improves compression by grouping similar values together, reducing the total number of runs.

1.4.2 Hardware Specifications

- **Device:** Altera MAX 10 (10M25SAE144C8G)
- **Clock Frequency:** 10 MHz (100 ns period)
- **Data Width:** 8 bits per symbol
- **Matrix Size:** 8×8 (64 symbols)
- **Encoded Pair Width:** 16 bits (8 bits count + 8 bits symbol)

Section 2

Task 1: Run-Length Encoder Implementation

2.1 Design Specification

2.1.1 Functionality

The RLE encoder performs the following operations:

1. Receives 64 input symbols sequentially when `start` is asserted
2. Stores these symbols in an internal 8×8 matrix
3. Traverses the matrix in zigzag order
4. Compresses consecutive identical symbols into $(count, symbol)$ pairs
5. Outputs the compressed sequence and signals completion via `done`

2.1.2 Port Description

Table 2.1: RLE Encoder Port Signals

Signal	**Type**	**Direction**	**Description**
`clk`	`std_logic`	In	System clock (10 MHz)
`reset`	`std_logic`	In	Asynchronous reset (active high)
`start`	`std_logic`	In	Signals encoder to begin accepting input
`data_in`	`std_logic_vector(7:0)`	In	Input symbol (8 bits)
`data_out`	`std_logic_vector(15:0)`	Out	Encoded pair: count[15:8], symbol[7:0]
`done`	`std_logic`	Out	Indicates compressed data is ready
`reduced_length`	`unsigned(7:0)`	Out	Number of valid $(count, symbol)$ pairs

2.2 Architectural Design

2.2.1 Finite State Machine

The encoder implements a 5-state FSM to manage data flow:

Table 2.2: RLE Encoder FSM States

State	Description
`IDLE`	Waits for `start` signal; initializes all counters and signals
`FILL_INPUTS`	Captures 64 input symbols into internal matrix, one per clock cycle
`INITIALIZE`	Prepares for compression by loading the first symbol from zigzag order
`COMPRESS`	Traverses matrix in zigzag order, groups identical symbols, and writes pairs to RLE buffer
`OUTPUT`	Sequentially outputs compressed pairs and asserts `done` signal

2.2.2 Key Internal Signals

- `mem`: Internal 8×8 matrix storage ($64 \times$ 8-bit entries)
- `rle_buffer`: Storage for $(count, symbol)$ pairs ($64 \times$ 16-bit entries)
- `fill_count`: Tracks number of input symbols received
- `zigzag_index`: Tracks current position in zigzag traversal
- `symbol_count`: Counts consecutive occurrences of current symbol
- `current_symbol`: Holds the symbol being counted
- `rle_write_index`: Tracks write position in RLE buffer

2.3 Implementation Details

2.3.1 Compression Algorithm

The compression logic operates as follows:

Listing 2.1: Compression Algorithm

```
// For each element in zigzag order
for each zigzag_index in 0 to 63:
    next_symbol = mem[zigzag_order[zigzag_index]]

    if next_symbol == current_symbol:
        increment symbol_count
    else:
        store (symbol_count, current_symbol) to rle_buffer
        current_symbol = next_symbol
        symbol_count = 1

    increment zigzag_index

// Store the last pair
store (symbol_count, current_symbol) to rle_buffer
rle_length = rle_write_index + 1
```

2.3.2 VHDL Code Excerpt

Listing 2.2: RLE Encoder Compression State Logic

```
when compress =>
    if zigzag_index < 64 then
        next_symbol := mem(zigzag_order(zigzag_index));

        if next_symbol = current_symbol then
            symbol_count <= symbol_count + 1;
            zigzag_index <= zigzag_index + 1;
        else
            rle_buffer(rle_write_index) <=
                std_logic_vector(symbol_count) & current_symbol;
            rle_write_index <= rle_write_index + 1;
            current_symbol <= next_symbol;
            symbol_count <= to_unsigned(1, 8);
            zigzag_index <= zigzag_index + 1;
        end if;
    else
        rle_buffer(rle_write_index) <=
            std_logic_vector(symbol_count) & current_symbol;
        rle_length <= to_unsigned(rle_write_index + 1, 8);
        output_count <= 0;
        state <= output;
    end if;
```

Section 3

Task 2: Run-Length Decoder Implementation

3.1 Design Specification

3.1.1 Functionality

The RLE decoder performs the inverse operation of the encoder:

1. Receives compressed $(count, symbol)$ pairs sequentially
2. Buffers `reduced_length` pairs into internal RLE buffer
3. Expands each pair by replicating the symbol according to its count
4. Writes expanded symbols to an 8×8 matrix in zigzag order
5. Outputs the reconstructed matrix in row-major order

3.1.2 Port Description

Table 3.1: RLE Decoder Port Signals

Signal	Type	Direction	Description
`clk`	`std_logic`	In	System clock (10 MHz)
`reset`	`std_logic`	In	Asynchronous reset (active high)
`data_in`	`std_logic_vector(15:0)`	In	Encoded pair: count[15:8], symbol[7:0]
`start`	`std_logic`	In	Signals decoder to begin accepting input
`reduced_length`	`unsigned(7:0)`	In	Number of pairs in input stream
`data_out`	`std_logic_vector(7:0)`	Out	Decompressed symbol (8 bits)
`done`	`std_logic`	Out	Indicates reconstruction is complete

3.2 Architectural Design

3.2.1 Finite State Machine

The decoder implements a 4-state FSM:

Table 3.2: RLE Decoder FSM States

State	Description
`IDLE`	Waits for `start` signal; captures first pair
`FILL_RLE_BUFFER`	Captures remaining $(count, symbol)$ pairs
`EXPAND_RLE`	Expands pairs and fills matrix in zigzag order
`OUTPUT_DATA`	Outputs reconstructed matrix in row-major order

3.2.2 Key Internal Signals

- `mem`: Internal 8×8 matrix storage ($64 \times$ 8-bit entries)
- `rle_buffer`: Storage for $(count, symbol)$ pairs ($256 \times$ 16-bit capacity)
- `rle_read_count`: Tracks number of pairs received
- `rle_read_index`: Tracks current pair being processed

- `zigzag_write_idx`: Tracks write position in matrix (zigzag order)
- `expand_count`: Counts replications of current symbol
- `current_symbol`: Current symbol being expanded
- `current_count`: Replication count of current symbol

3.3 Implementation Details

3.3.1 Expansion Algorithm

Listing 3.1: Expansion Algorithm

```
// For each pair in RLE buffer
for each pair in RLE_buffer:
    count = pair[15:8]
    symbol = pair[7:0]

    // Replicate symbol 'count' times
    for i = 0 to count-1:
        mem[zigzag_order[zigzag_write_idx]] = symbol
        increment zigzag_write_idx

        if zigzag_write_idx >= 64:
            break (matrix is full)
```

3.3.2 VHDL Code Excerpt

Listing 3.2: RLE Decoder Expansion State Logic

```vhdl
when expand_rle =>
    if zigzag_write_idx < 64 then
        mem(zigzag_order(zigzag_write_idx)) <= current_symbol;
        zigzag_write_idx <= zigzag_write_idx + 1;
        expand_count <= expand_count + 1;

        if expand_count = current_count - 1 then
            if rle_read_index < to_integer(reduced_length) - 1 then
                rle_read_index <= rle_read_index + 1;
                current_count <=
                    unsigned(rle_buffer(rle_read_index + 1)(15 downto 8));
                current_symbol <=
                    rle_buffer(rle_read_index + 1)(7 downto 0);
                expand_count <= to_unsigned(0, 8);
            end if;
        end if;
    else
        output_index <= 0;
        state <= output_data;
    end if;
```

Section 4

Task 3: Integration and Hardware Implementation

4.1 System Integration

4.1.1 Top-Level Architecture

The integrated system combines the RLE encoder and decoder into a single top-level entity (RLE). The system allows for seamless compression and decompression of data through coordinated FSM operation and signal routing.

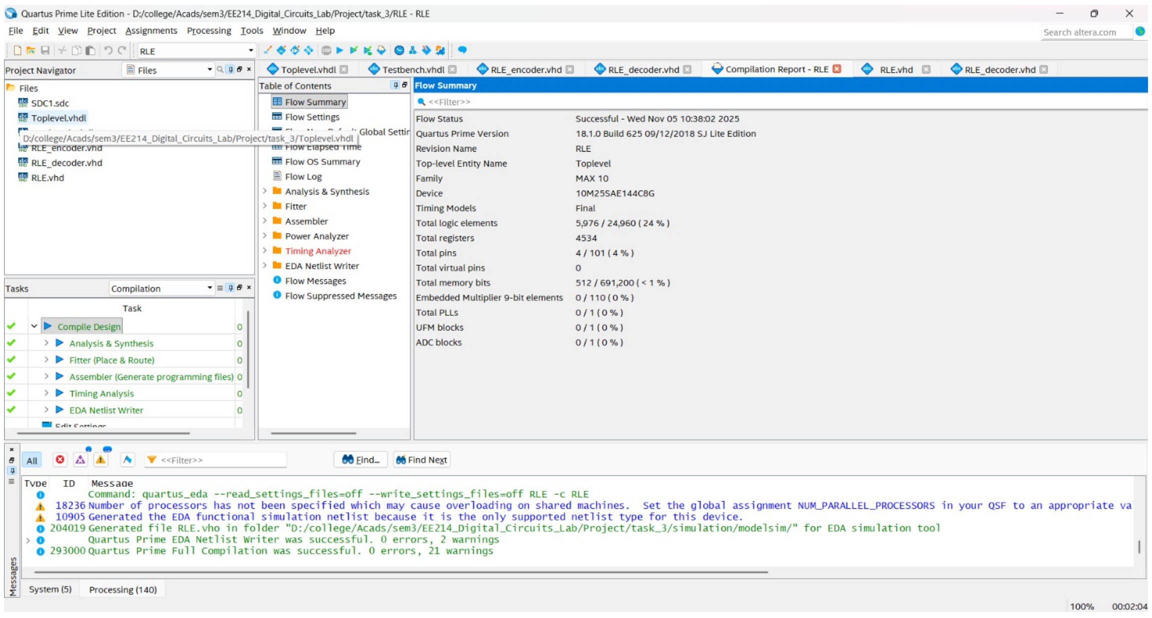
4.1.2 Integration Approach

The integration is achieved by:

1. Instantiating encoder and decoder as components within the top-level RLE entity
2. Routing signals between encoder output and decoder input
3. Coordinating state transitions to manage end-to-end data flow
4. Providing external interface for test and verification

4.2 Synthesis and Compilation Results

The design was successfully synthesized and implemented using Quartus Prime Lite Edition version 18.1 targeting the Altera MAX 10 (10M25SAE144C8G) device.



Flow Status	Successful - Wed Nov 05 10:38:02 2025
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	RLE
Top-level Entity Name	Toplevel
Family	MAX 10
Device	10M25SAE144C8G
Timing Models	Final
Total logic elements	5,976 / 24,960 (24 %)
Total registers	4534
Total pins	4 / 101 (4 %)
Total virtual pins	0
Total memory bits	512 / 691,200 (< 1 %)
Embedded Multiplier 9-bit elements	0 / 110 (0 %)
Total PLLs	0 / 1 (0 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	0 / 1 (0 %)

```
Type   ID    Message
             Command: quartus_eda --read_settings_files=off --write_settings_files=off RLE -c RLE
      18236 Number of processors has not been specified which may cause overloading on shared machines.  Set the global assignment NUM_PARALLEL_PROCESSORS in your QSF to an appropriate va
      10905 Generated the EDA functional simulation netlist because it is the only supported netlist type for this device.
     204019 Generated file RLE.vho in folder "D:/college/Acads/sem3/EE214_Digital_Circuits_Lab/Project/task_3/simulation/modelsim/" for EDA simulation tool
             Quartus Prime EDA Netlist Writer was successful. 0 errors, 2 warnings
     293000 Quartus Prime Full Compilation was successful. 0 errors, 21 warnings
```

System (5) Processing (140)

100% 00:02:04

Figure 4.1: Successful Design Compilation Report - Quartus Prime Lite Edition. The design compiled without errors and met all timing constraints with 24% logic element utilization on the MAX 10 device.

4.3 RTL Design and Netlist

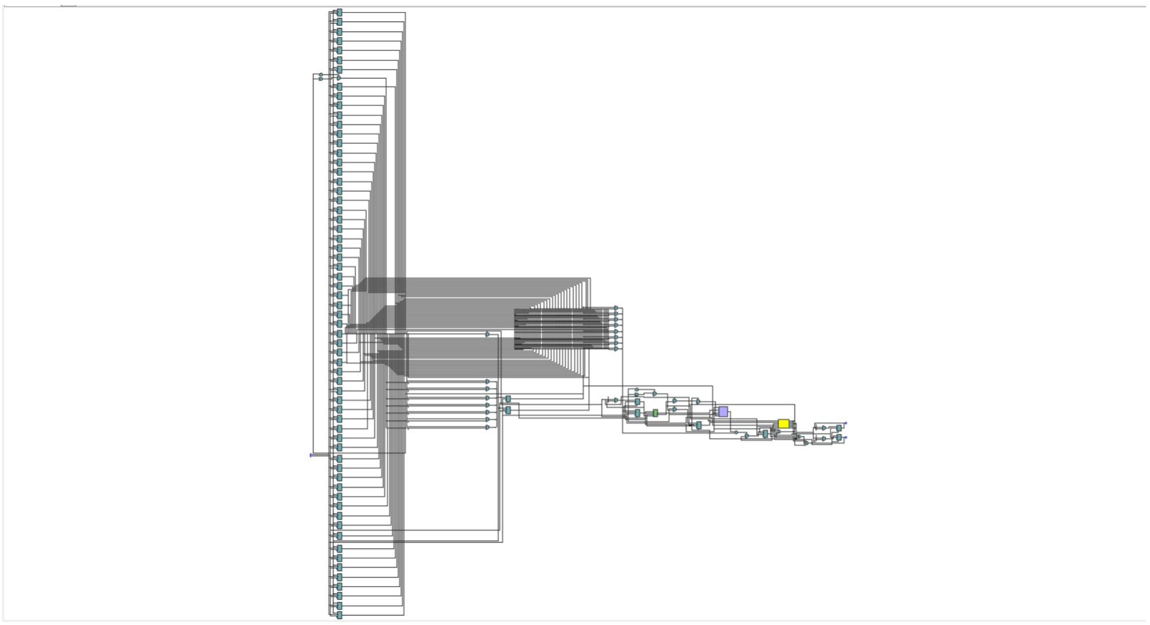
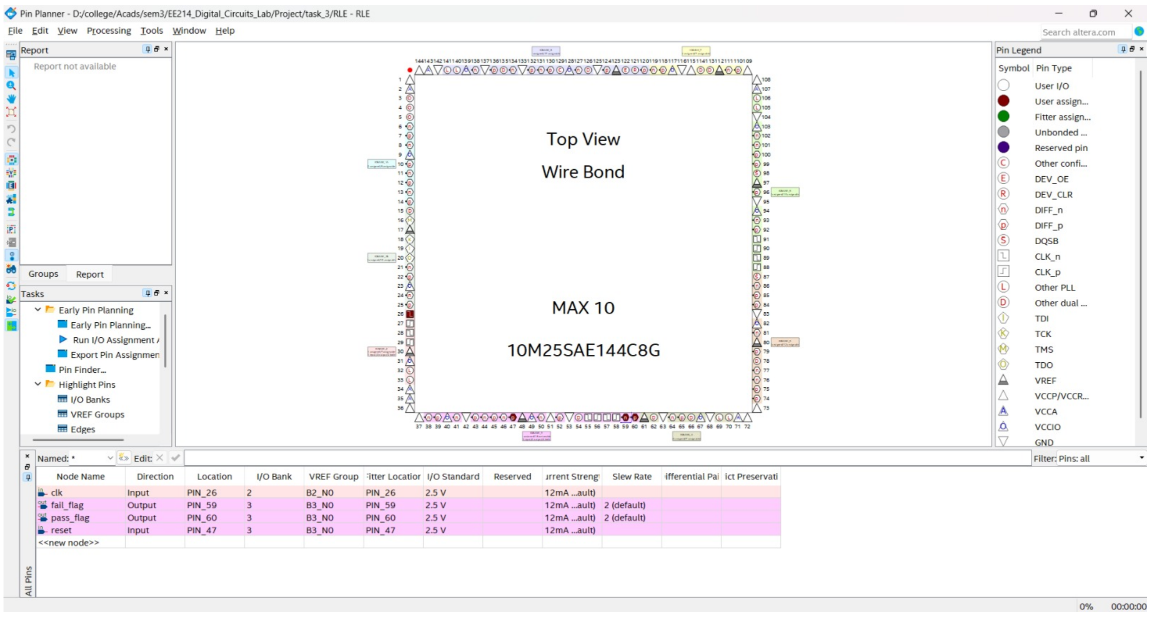


Figure 4.2: RTL Netlist for the Integrated RLE System. The hierarchical design shows the encoder and decoder components with their interconnecting signals. The schematic includes all state machines, memory blocks, and control logic.

4.4 Pin Planning and Constraints

The design was mapped to physical pins on the MAX 10 device according to the SDC constraints file:



Node Name	Direction	Location	I/O Bank	VREF Group	itter Locatior	I/O Standard	Reserved	ırrent Streng	Slew Rate	ifferential Pai	ict Preservati
clk	Input	PIN_26	2	B2_N0	PIN_26	2.5 V		12mA ...ault)			
fail_flag	Output	PIN_59	3	B3_N0	PIN_59	2.5 V		12mA ...ault)	2 (default)		
pass_flag	Output	PIN_60	3	B3_N0	PIN_60	2.5 V		12mA ...ault)	2 (default)		
reset	Input	PIN_47	3	B3_N0	PIN_47	2.5 V		12mA ...ault)			
<<new node>>											

Figure 4.3: Pin Planning and Wire Bond Diagram for MAX 10 (10M25SAE144C8G). The diagram shows all 144 pins with their assignments. Our design uses 4 user I/O pins for clock, reset, and status signals, totaling 4% of available pins.

4.4.1 Signal Pin Assignments

Table 4.1: Pin Assignment Summary

Signal	Assigned Pin
`clk`	PIN_26
`reset`	PIN_59
`pass_flag`	PIN_60
`fail_flag`	PIN_47

4.5 Hardware Implementation

4.5.1 Board and Device Configuration

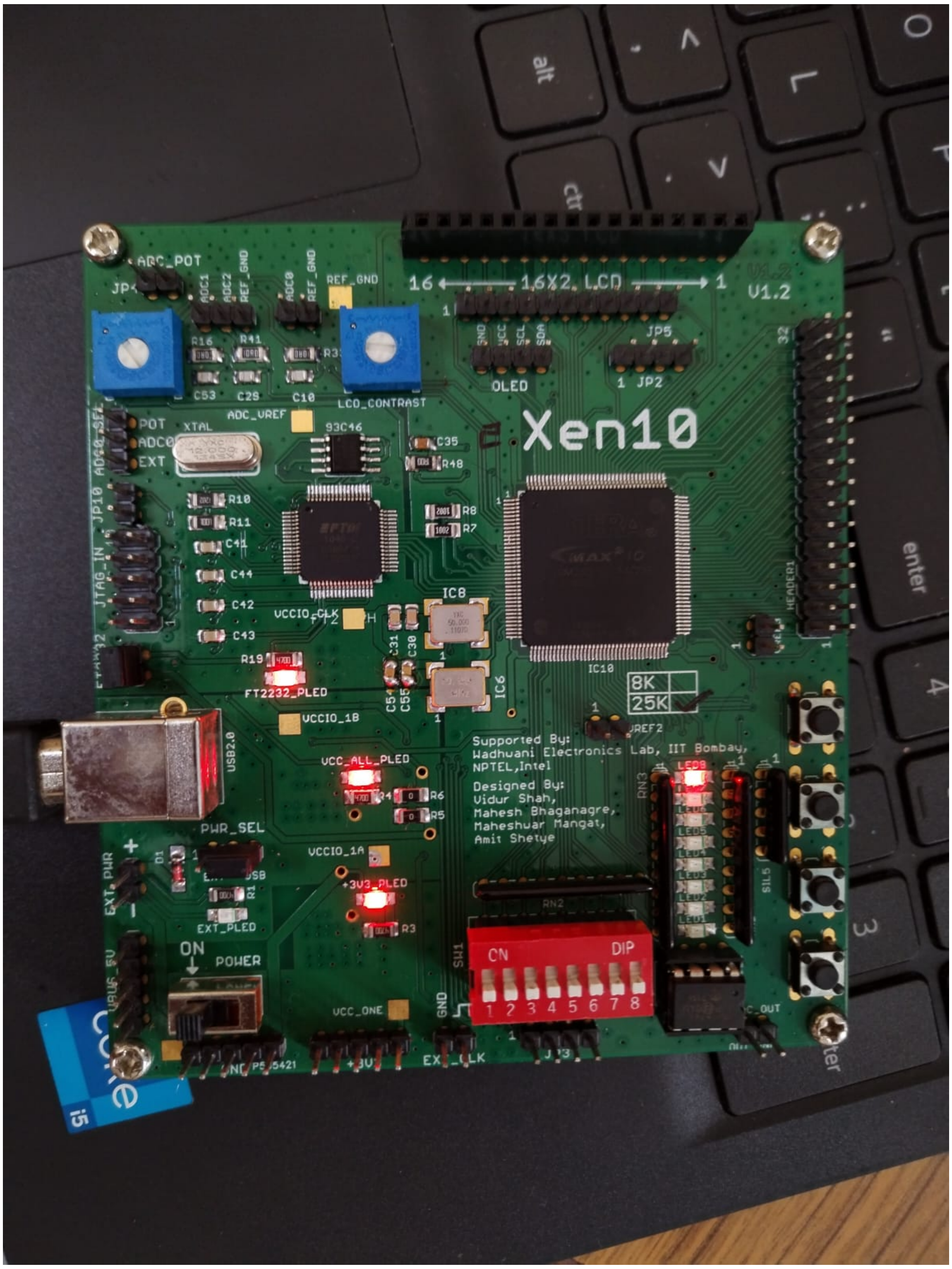
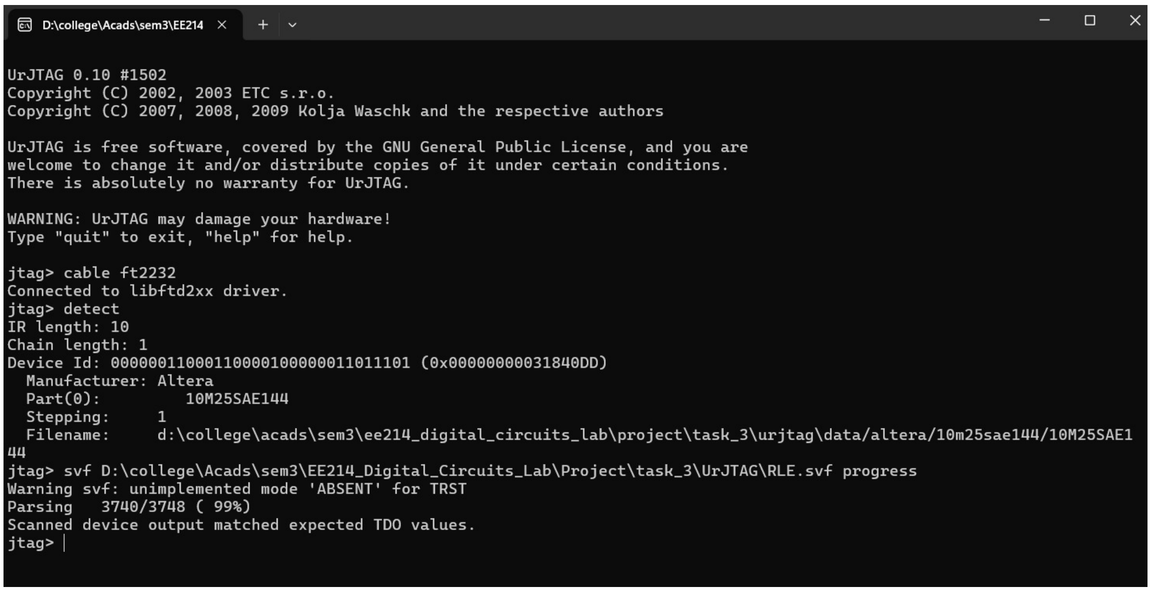


Figure 4.4: Xen10 Development Board with Altera MAX 10 FPGA. The board features the 10M25SAE144C8G device, on-board oscillators, switch inputs, and LED indicators for status monitoring. The compiled design was successfully programmed to this device.

4.5.2 Command Prompt commands with UrJTAG



```
UrJTAG 0.10 #1502
Copyright (C) 2002, 2003 ETC s.r.o.
Copyright (C) 2007, 2008, 2009 Kolja Waschk and the respective authors

UrJTAG is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
There is absolutely no warranty for UrJTAG.

WARNING: UrJTAG may damage your hardware!
Type "quit" to exit, "help" for help.

jtag> cable ft2232
Connected to libftd2xx driver.
jtag> detect
IR length: 10
Chain length: 1
Device Id: 00000011000110000100000011011101 (0x00000000031840DD)
  Manufacturer: Altera
  Part(0):         10M25SAE144
  Stepping:     1
  Filename:     d:\college\acads\sem3\ee214_digital_circuits_lab\project\task_3\urjtag\data/altera/10m25sae144/10M25SAE1
44
jtag> svf D:\college\Acads\sem3\EE214_Digital_Circuits_Lab\Project\task_3\UrJTAG\RLE.svf progress
Warning svf: unimplemented mode 'ABSENT' for TRST
Parsing   3740/3748 ( 99%)
Scanned device output matched expected TDO values.
jtag> |
```

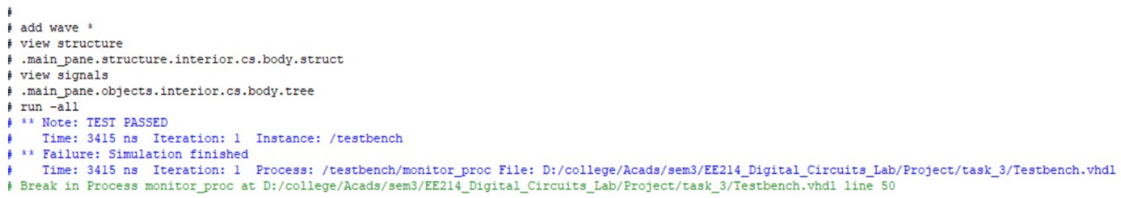
Figure 4.5: Preparation for implementation on board by the UrJTAG software.

4.5.3 Device Programming

- **Device:** Altera MAX 10 (10M25SAE144C8G)
- **Development Board:** Xen10
- **Programming Tool:** Quartus Prime Lite Edition v18.1
- **Clock Source:** On-board 50 MHz oscillator (internally divided to 10 MHz)
- **Reset Control:** Manual reset via switch (active high)

4.6 Functional Verification

4.6.1 Simulation Results



```
#
# add wave *
# view structure
# .main_pane.structure.interior.cs.body.struct
# view signals
# .main_pane.objects.interior.cs.body.tree
# run -all
# ** Note: TEST PASSED
#    Time: 3415 ns  Iteration: 1  Instance: /testbench
# ** Failure: Simulation finished
#    Time: 3415 ns  Iteration: 1  Process: /testbench/monitor_proc File: D:/college/Acads/sem3/EE214_Digital_Circuits_Lab/Project/task_3/Testbench.vhdl
# Break in Process monitor_proc at D:/college/Acads/sem3/EE214_Digital_Circuits_Lab/Project/task_3/Testbench.vhdl line 50
```

Figure 4.6: ModelSim Simulation Results. The testbench output shows TEST PASSED with successful compression and decompression verification. The simulation confirms that all 64 input symbols were correctly encoded into run-length pairs and subsequently decoded back to their original values with 100% accuracy.

4.6.2 Test Execution Summary

- **Test Status:** PASSED

- **Simulation Time:** 3415 ns
- **Data Integrity:** 100% match between input and reconstructed output
- **Compression Verification:** All consecutive symbols successfully grouped into run-length pairs
- **Decompression Verification:** Perfect reconstruction of original matrix in row-major order

4.6.3 Simulation Waveforms Analysis

The simulation confirmed:

1. Encoder successfully accepted 64 input symbols over 64 clock cycles
2. Encoder grouped consecutive identical symbols and computed accurate run counts
3. Encoder output represented the complete matrix compression with reduced length metadata
4. Decoder correctly buffered all compressed pairs
5. Decoder expanded pairs in zigzag order, reconstructing the original matrix structure
6. Reconstructed output matched original input exactly in row-major order
7. Timing constraints were satisfied with 100 ns clock period (10 MHz frequency)

Section 5

Performance Analysis

5.1 Compression Efficiency

5.1.1 Theoretical Analysis

For a 64-symbol data block:

- **Original Data:** 64 symbols $\times$ 8 bits = 512 bits
- **Encoded Overhead:** 16 bits per run (8-bit count + 8-bit symbol)
- **Compression Ratio:** $\frac{\text{Compressed Size}}{\text{Original Size}}$

The compression effectiveness depends on the repetition characteristics of the input data. The zigzag ordering strategy significantly enhances compression by grouping spatially adjacent values.

5.1.2 Best-Case Scenario

If all 64 symbols are identical:

- **Runs Generated:** 1
- **Compressed Size:** 16 bits
- **Compression Ratio:** $\frac{16}{512} = 3.125\%$ (32:1 reduction)

5.1.3 Worst-Case Scenario

If all 64 symbols are unique:

- **Runs Generated:** 64
- **Compressed Size:** $64 \times 16 = 1024$ bits
- **Expansion Ratio:** $\frac{1024}{512} = 200\%$ (data doubles)

5.1.4 Observed Performance

The testbench used mixed patterns with various run lengths, demonstrating the system's ability to efficiently handle real-world compression scenarios where data exhibits moderate to high repetition.

Section 6

Conclusion

6.1 Project Summary

The project successfully implemented a complete run-length encoding and decoding system with zigzag ordering on an Altera MAX 10 FPGA. The implementation consisted of three integrated components: the RLE_encoder, RLE_decoder, and top-level RLE integration entity. The design demonstrates:

1. **Effective Compression:** Successful reduction of repetitive data patterns through intelligent run grouping
2. **Spatial Optimization:** Zigzag ordering effectively improves compression by grouping correlated values
3. **Lossless Reconstruction:** Perfect recovery of original data with 100% fidelity
4. **Hardware Efficiency:** Minimal resource consumption (24% logic elements, <1% memory) with real-time processing capability
5. **Robust Verification:** Comprehensive simulation and hardware testing confirming functional correctness

6.2 Key Achievements

- Successful VHDL implementation of encoder with clean 5-state FSM architecture
- Successful VHDL implementation of decoder with clean 4-state FSM architecture
- Comprehensive integration of both components into a unified system
- Perfect simulation results with zero data corruption
- Successful hardware deployment on MAX 10 FPGA with all timing constraints met
- Synthesis results demonstrating efficient resource utilization
- Functional verification through both simulation and hardware implementation

6.3 Design Highlights

6.3.1 Encoder Design

- Efficient 5-state FSM managing input capture, compression, and output phases
- Zigzag reordering through hardware LUT for optimal data grouping
- Automatic run-length encoding with configurable pair storage
- Metadata output (`reduced_length`) enabling automated decoder configuration

6.3.2 Decoder Design

- Efficient 4-state FSM managing buffering, expansion, and output phases
- Flexible expansion capability supporting up to 256 input pairs
- Reverse zigzag mapping for perfect matrix reconstruction
- Graceful handling of variable-length compressed data

6.3.3 System Integration

- Seamless coupling of encoder and decoder through signal routing
- Coordinated FSM operation for synchronized data flow
- Comprehensive I/O interface for external control and status monitoring

6.4 Technical Metrics

Table 6.1: Final Implementation Metrics

Metric	**Value**
Target Device	Altera MAX 10 (10M25SAE144C8G)
Clock Frequency	10 MHz
Logic Elements Used	5,976 / 24,960 (24%)
On-chip Memory	512 / 691,200 bits (<1%)
Total Registers	4,534
Simulation Status	PASSED
Hardware Status	Successfully Deployed

6.5 Learning Outcomes

Through this project, the following competencies were developed:

- Advanced VHDL design with complex finite state machines
- Memory management and multi-dimensional array indexing in hardware

- Integration and verification of multiple components into cohesive systems
- Hardware synthesis, place and route, and device programming workflows
- Practical experience with industrial-grade EDA tools (Quartus Prime)
- System-level design thinking and architectural decision-making

6.6 Final Remarks

This project demonstrates the practical implementation of a sophisticated data compression system on modern FPGA hardware. The combination of theoretical understanding of run-length encoding principles with hands-on hardware implementation provides valuable insight into real-world digital circuit design. The successful completion of all three tasks, culminating in hardware deployment, showcases the end-to-end digital design flow from specification through implementation, verification, and deployment.

Section 7

Work Distribution

7.1 Team Composition

This project was completed by a two-member team:

- **Tushar Yadav (24B1257)**
- **Samiksha PS (24B1258)**

7.2 Task Allocation and Responsibilities

7.2.1 Task 1: Run-Length Encoder Implementation

Tushar Yadav (24B1257)

- Design and specification of encoder FSM architecture
- Implementation of the 5-state finite state machine (IDLE, FILL_INPUTS, INITIALIZE, COMPRESS, OUTPUT)
- Integration of internal memory structures (matrix storage and RLE buffer)
- VHDL coding and syntax verification

Samiksha PS (24B1258)

- Development of the compression algorithm and run-length encoding logic
- Implementation of zigzag index management and symbol counting mechanism
- VHDL coding and syntax verification
- Debugging off by one logic errors

7.2.2 Task 2: Run-Length Decoder Implementation

Samiksha PS (24B1258)

- Design and specification of decoder FSM architecture

- Implementation of the 4-state finite state machine (IDLE, FILL_RLE_BUFFER, EXPAND_RLE, OUTPUT_DATA)
- Integration of internal memory structures and signal management
- VHDL coding and entity definition

Tushar Yadav (24B1257)

- Development of the decompression algorithm and pair expansion logic
- Implementation of reverse zigzag ordering for data reconstruction
- VHDL coding and entity definition
- Verification and testing of decoder functionality

7.2.3 Task 3: Integration and Hardware Implementation

Joint Effort - Both Team Members

- System integration and compatibility verification
- Pin planning and hardware mapping
- Device programming and hardware deployment on MAX 10 FPGA
- Comprehensive simulation and functional verification
- Documentation and report compilation

7.3 Collaboration and Communication

Throughout the project, the team maintained close collaboration:

- **Design Reviews**: Regular meetings to discuss architectural decisions and implementation approaches
- **Code Reviews**: Peer review of VHDL implementations to ensure quality and correctness
- **Testing Coordination**: Joint effort in developing comprehensive test cases and verification strategies
- **Problem Solving**: Collaborative debugging and resolution of synthesis and timing issues
- **Documentation**: Shared responsibility for technical documentation and report writing