

FPGA-Based LZW Image Compression Using Vitis HLS

Anshu Biswas –IMT2021534

Rohan Rajesh –IMT2022575

Margasahayam Venkatesh Chirag -IMT2022583



Introduction

An input image undergoes LZW (Lempel–Ziv–Welch) compression to reduce storage size for efficient transmission. Following compression, the image is decompressed, and the output image is displayed.



Implementation

1.LZW Compression code for HLS

2.RTL/Verilog code

3.Image Decompression code

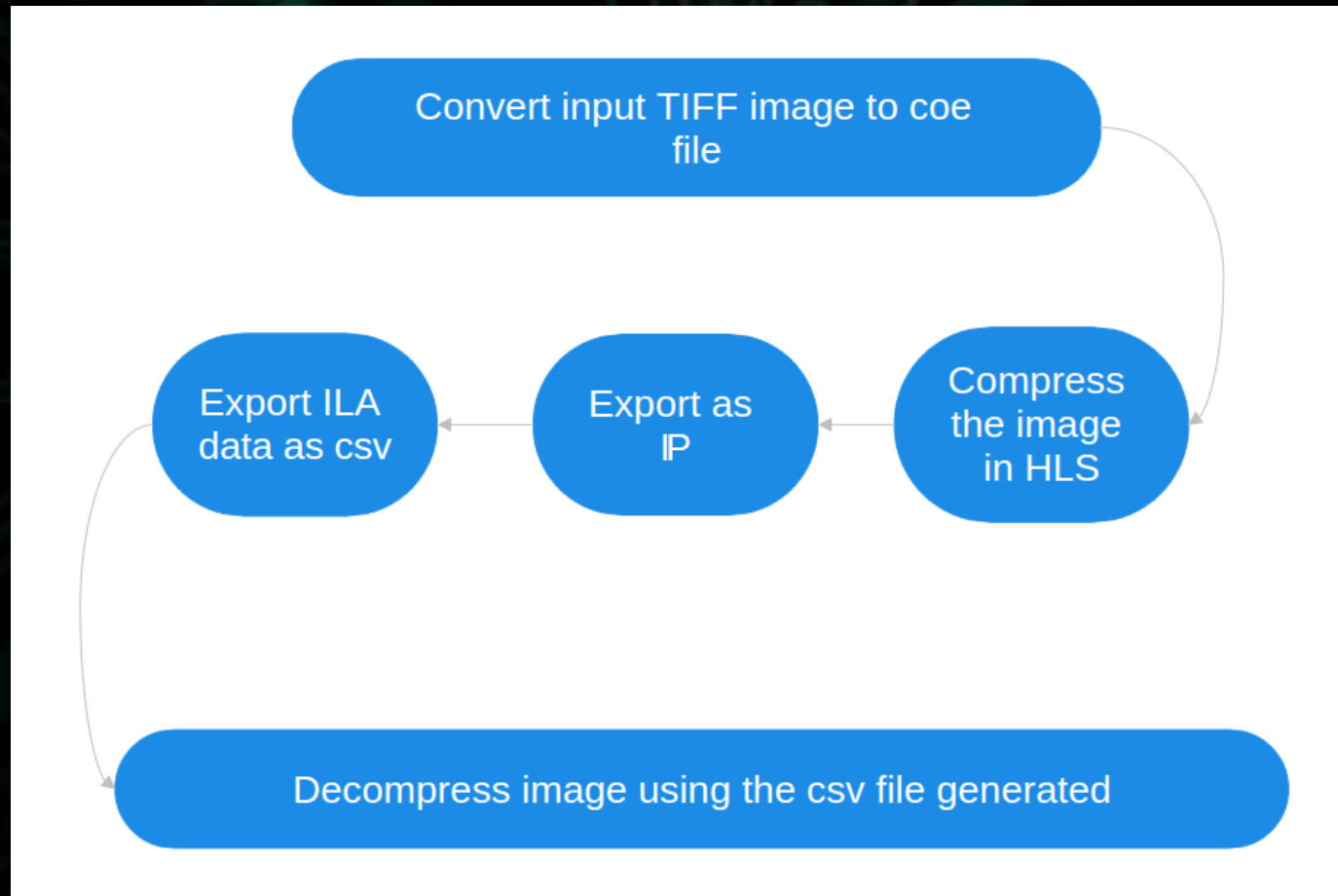
Github Link:- <https://github.com/MChiragV/FPGA-Course-Project>



How to Run Files from the GitHub Link

- First, run the `rgb_tiff_to_coe.c` file, which converts the TIFF image to coe file.
- Open Vitis HLS, create a new project and upload the `lzw_compress.cpp` code in the sources and `lzw_compress_tb.cpp` in the testbench. Then, for board, use the zedboard.
- Run the simulation, synthesis, and export as IP.
- Then, open Vivado, create a new project(using the same board used in Vitis HLS), and upload the `code.v` as source, `constraints.xdc` as constraints file.
- Run the design till bitstream and generate the csv file from the ILA.
- Then, run the `decompress_and_display.py` file and the output TIFF image is generated.

Block diagram





LZW Compression

- LZW compression is a lossless algorithm i.e., there is no data loss while compressing. It compresses data by replacing repeated sequences with shorter codes.
- LZW compression is efficient for files with repeating patterns, like text and images. This method is also preferred due to its simplicity, thus allowing fast execution.
- Initially, a dictionary stores all unique characters.
- As the algorithm processes the input, it builds longer sequences in the dictionary. When a new sequence isn't in the dictionary, the algorithm outputs the code for the longest match found. Then, it adds the new sequence to the dictionary.



LZW Compression C++ Code

- lzw_compress.cpp:
 - Init_dictionary function initializes the dictionary with 256 unique entries(since ASCII characters range from 0 to 255).
 - find_in_dictionary searches for the sequence in the dictionary and returns true if the sequence is found, else return false.
 - The top function lzw_compress takes in an array as input(along with its size) and output array(along with its size), uses the above 2 functions for compressing the input, and modifies the output array.
 - Note that the code is slowed down by the find_in_dictionary function since it has to traverse the dictionary in order to search the sequence. So, loop unrolling was applied to this function in order to make the process faster.



Verilog Code

The code performs the following steps to compress the given image.

- Load the COE file into a BRAM.
- Split the elements of the BRAM in three arrays, one for each of R,G,B. (8 bits wide each)
- Pass these arrays through 3 instances of the compression IP in parallel.
- Probe the compressed output elements and output size in the ILA.
- Export the ILA contents to CSV for further processing.



Decompression Code

- This code reads compressed RGB data from a CSV file(that was exported using the ILA outputs), decompresses it with LZW, and reconstructs an RGB image.
- **LZW Decompression:** The lzw_decompress function decompresses LZW-encoded integer codes, rebuilding the original data using a dynamic dictionary.
- **Reading and Extracting Active Segments:** The read_and_decompress function reads RGB columns from the CSV and identifies segments of compressed data for each color, marked by zeroes.
- **Image Reconstruction:** The decompressed data is converted to pixel values, arranged into an RGB array, and saved as a TIFF image .

This process turns compressed CSV RGB data into a visible image file.

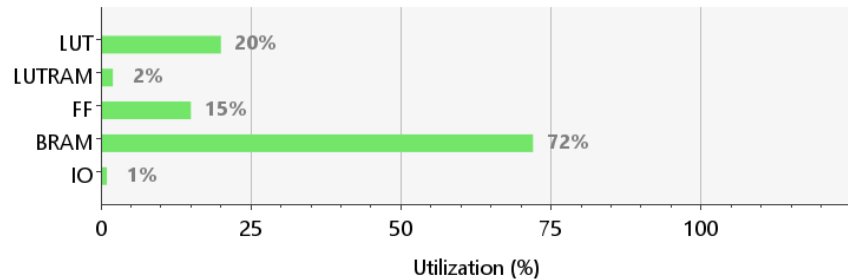


Results

Resource Utilization

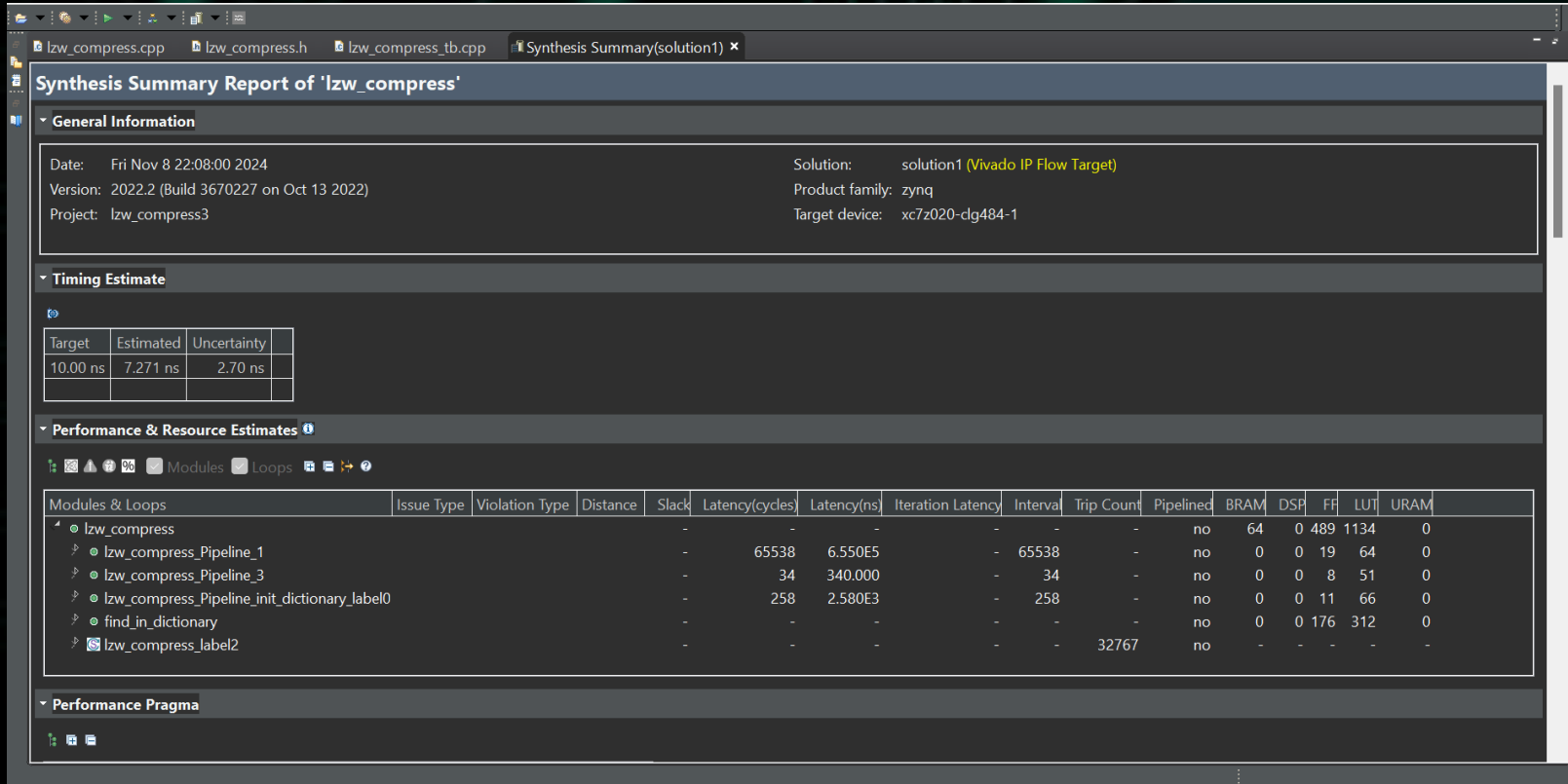
Summary

Resource	Utilization	Available	Utilization %
LUT	10516	53200	19.77
LUTRAM	275	17400	1.58
FF	15942	106400	14.98
BRAM	101	140	72.14
IO	1	200	0.50



Name	Slice LUTs (53200)	Slice Registers (106400)	F7 Muxes (26600)	F8 Muxes (13300)	Block RAM Tile (140)	Bonded IOB (200)	BUFGCTRL (32)
image_compress	10516	15942	772	384	101	1	1
BRAM (blk_mem_gen_0)	0	0	0	0	0.5	0	0
dbg_hub (dbg_hub_CV)	0	0	0	0	0	0	0
ILA (ila_0)	1184	1843	4	0	3	0	0
lzw_inst_b (lzw_compress_0)	497	479	0	0	32	0	0
lzw_inst_g (lzw_compress_0)	497	479	0	0	32	0	0
lzw_inst_r (lzw_compress_0)	497	479	0	0	32	0	0

Latency



The screenshot displays the 'Synthesis Summary Report of 'lzw_compress'' in a Vivado IDE window. The report is organized into several sections: General Information, Timing Estimate, and Performance & Resource Estimates.

General Information

Date:	Fri Nov 8 22:08:00 2024	Solution:	solution1 (Vivado IP Flow Target)
Version:	2022.2 (Build 3670227 on Oct 13 2022)	Product family:	zynq
Project:	lzw_compress3	Target device:	xc7z020-clg484-1

Timing Estimate

Target	Estimated	Uncertainty
10.00 ns	7.271 ns	2.70 ns

Performance & Resource Estimates

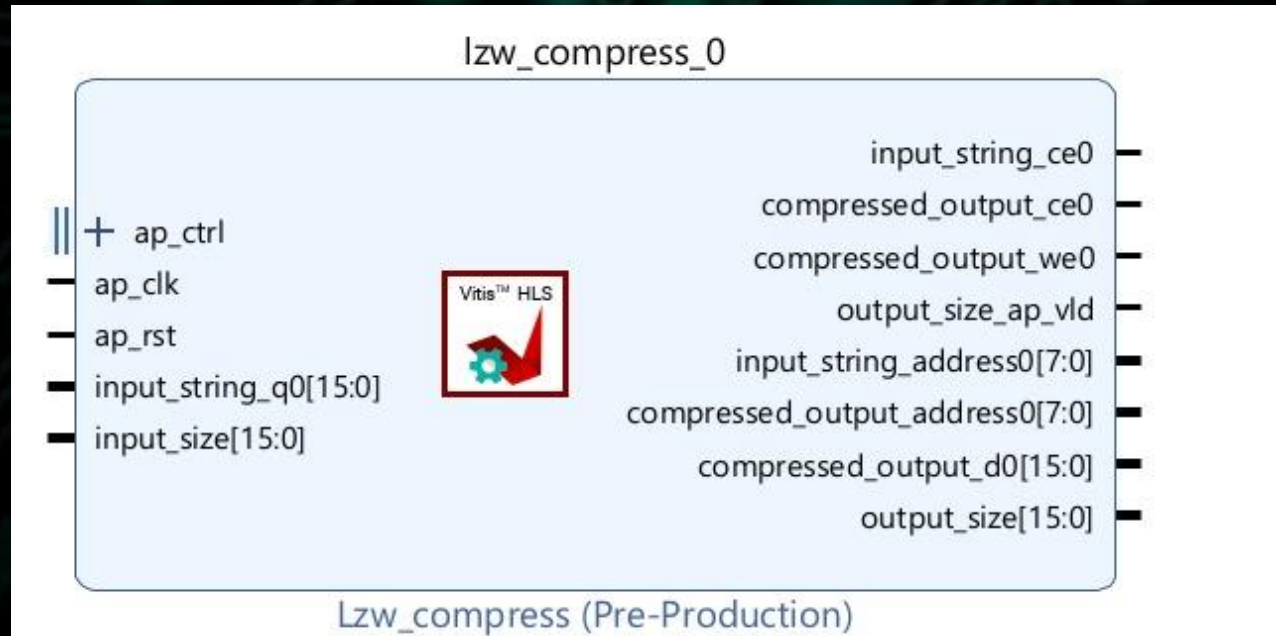
Modules & Loops

Module/Loop	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
lzw_compress	-	-	-	-	-	-	-	-	-	no	64	0	489	1134	0
lzw_compress_Pipeline_1	-	-	-	-	65538	6.550E5	-	65538	-	no	0	0	19	64	0
lzw_compress_Pipeline_3	-	-	-	-	34	340.000	-	34	-	no	0	0	8	51	0
lzw_compress_Pipeline_init_dictionary_label0	-	-	-	-	258	2.580E3	-	258	-	no	0	0	11	66	0
find_in_dictionary	-	-	-	-	-	-	-	-	-	no	0	0	176	312	0
lzw_compress_label2	-	-	-	-	-	-	-	-	32767	no	-	-	-	-	-

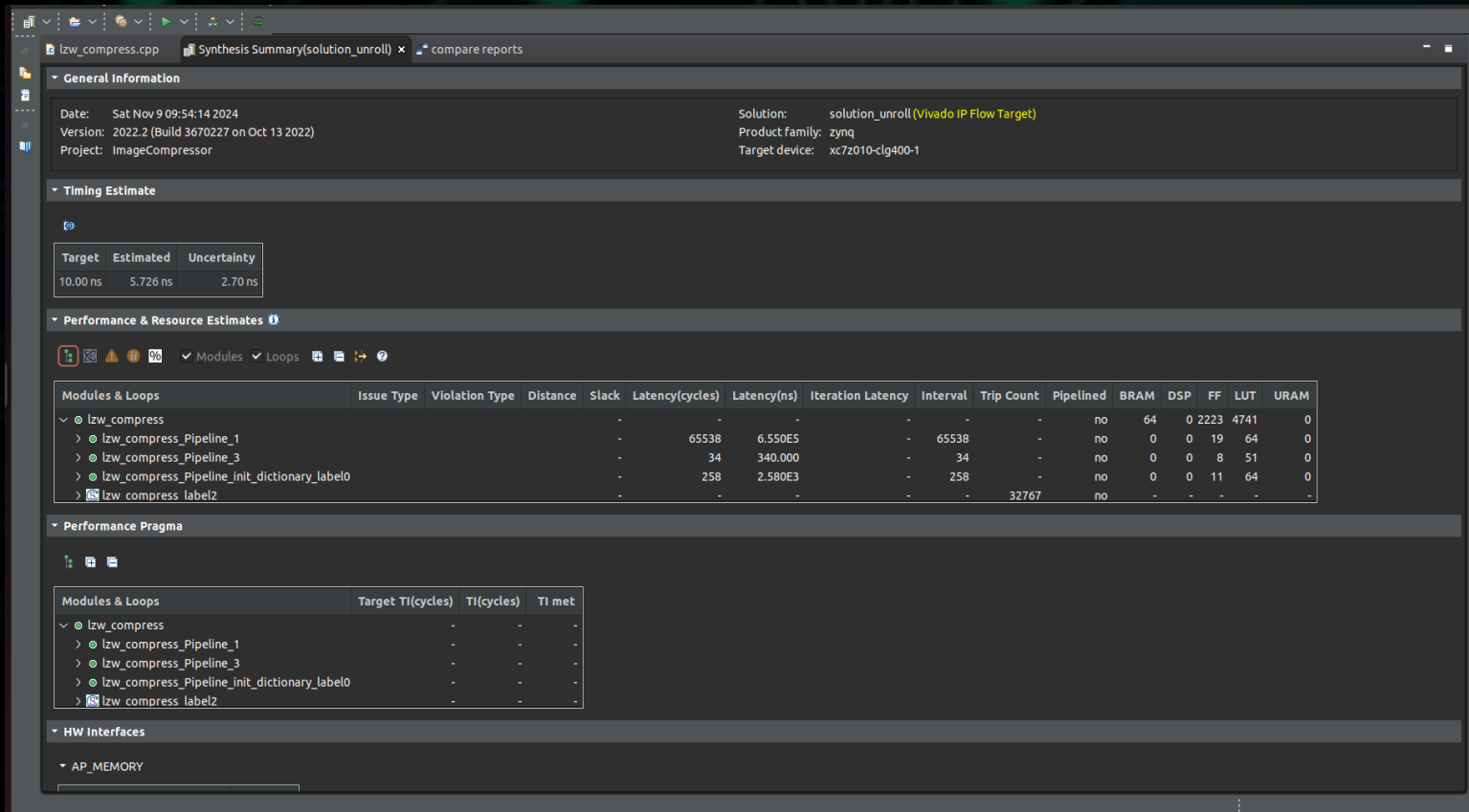
Performance Pragma

Synthesis summary report obtained in Vitis HLS

LZW IP



Latency for unrolling(factor=20) and pipelining



The screenshot shows the Vivado Synthesis Summary report for a solution named 'solution_unroll'. The report is divided into several sections: General Information, Timing Estimate, Performance & Resource Estimates, Performance Pragma, and HW Interfaces.

General Information

Field	Value
Date	Sat Nov 9 09:54:14 2024
Version	2022.2 (Build 3670227 on Oct 13 2022)
Project	ImageCompressor
Solution	solution_unroll (Vivado IP Flow Target)
Product Family	zynq
Target device	xc7z010-clg400-1

Timing Estimate

Target	Estimated	Uncertainty
10.00 ns	5.726 ns	2.70 ns

Performance & Resource Estimates

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
lzw_compress				-	-	-	-	-	-	no	64	0	2223	4741	0
lzw_compress_Pipeline_1				-	65538	6.550E5	-	65538	-	no	0	0	19	64	0
lzw_compress_Pipeline_3				-	34	340.000	-	34	-	no	0	0	8	51	0
lzw_compress_Pipeline_init_dictionary_label0				-	258	2.580E3	-	258	-	no	0	0	11	64	0
lzw_compress_label2				-	-	-	-	-	32767	no	-	-	-	-	-

Performance Pragma

Modules & Loops	Target TI(cycles)	TI(cycles)	TI met
lzw_compress	-	-	-
lzw_compress_Pipeline_1	-	-	-
lzw_compress_Pipeline_3	-	-	-
lzw_compress_Pipeline_init_dictionary_label0	-	-	-
lzw_compress_label2	-	-	-

HW Interfaces

AP_MEMORY

Synthesis summary report obtained in Vitis HLS



Tradeoff between resource utilization and kernel size

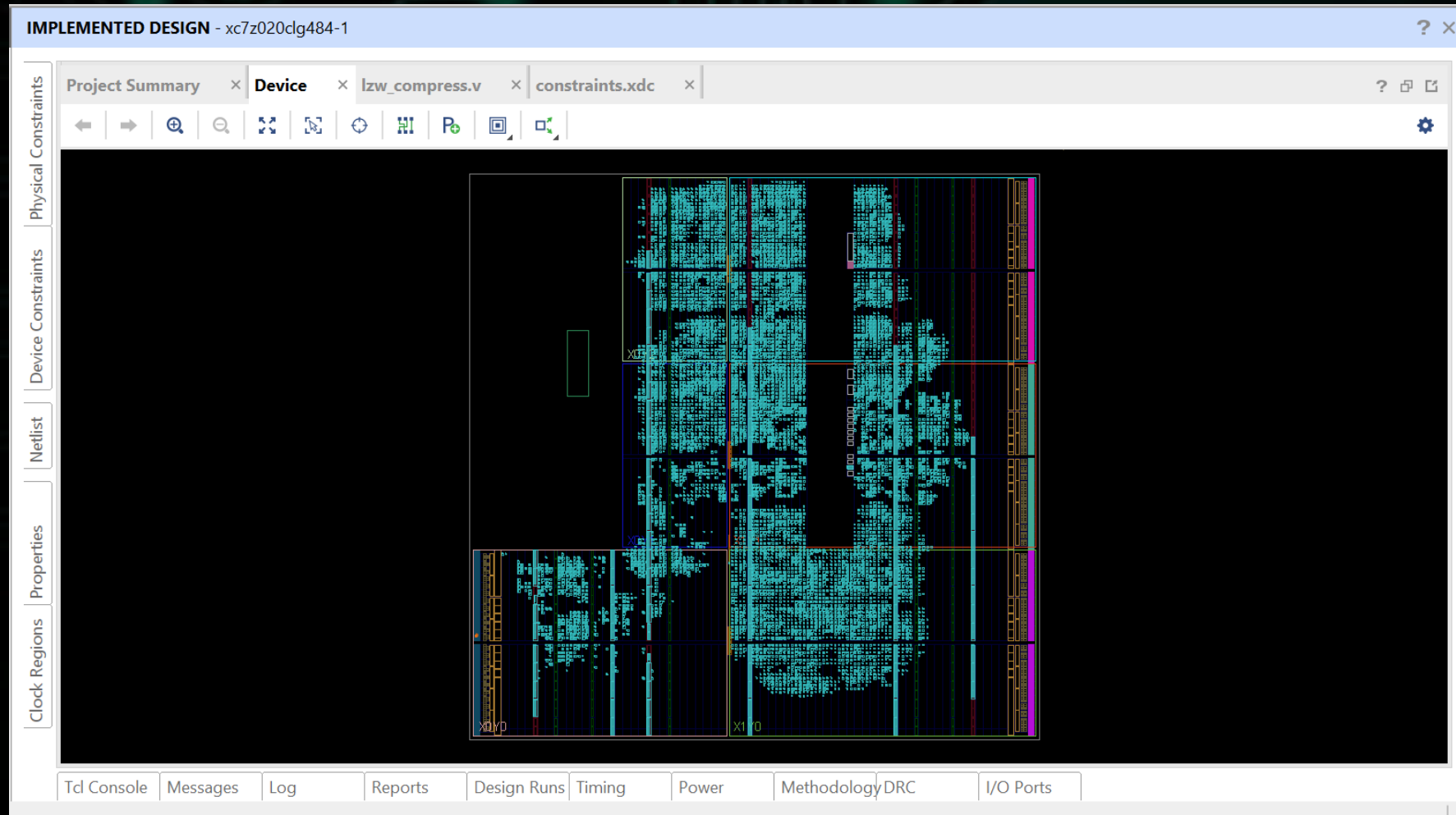
On applying loop unrolling and pipelining, the resources utilized increased drastically, with LUTs increasing from 1134 to 4741 and FFs increasing from 489 to 2223. This would prevent user from increasing the kernel size from 16×16 to higher sizes due to high resource utilization. But small kernel size would slower the traversal time. So, in order to maintain the tradeoff between resource utilized and kernel size, we chose the one with lesser resource utilization.



Max Clock Frequency

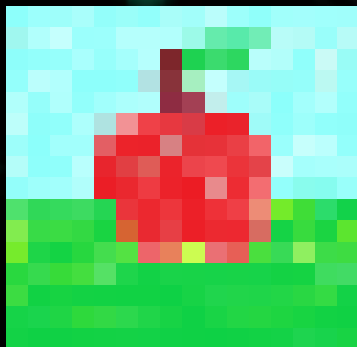
- Max clock frequency achieved :90.9 MHz

Implemented Layout

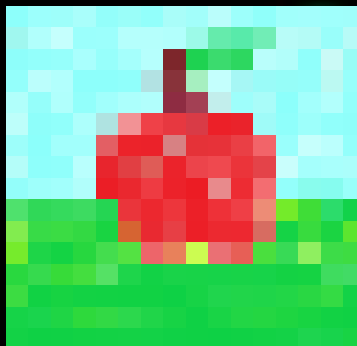


Sample Input and Output

Input



Output



Original size: 768 bytes (0.75 KB)

Compressing data...

Compressed size: 652 bytes (0.64 KB)

Compression ratio: 1.18:1

Space saved: 15.10%

Image saved as 'output_image_new.tiff'



Problems Faced

- **1. Probing array elements in ILA**
- The ILA buffer captures the array in a circular fashion. Say the window size is 4096 and array size is 1024, then we got the array to repeat 4 times, but the start index at the first window sample was random every time we ran the ILA. To solve this issue, we used a `capture_active` signal as a trigger. What this signal does is that it is high when the array elements are valid, and low when they're not.
- Github link for the same: <https://github.com/MChiragV/FPGA-Course-Project/tree/main/Errors>



Advantages of LZW technique

- LZW technique can compress the input array by traversing it only once.
- It is faster than other compression algorithms due to its simplicity.
- It performs better for images with repetitive patterns.



References

- <https://gitlab.com/libtiff/libtiff/-/blob/master/tools/tiffcp.c>



THANK
YOU