UCSD CSE 237C
Final Project
Error Correcting Codes - SECDED

Naman Sehgal
nsehgal@ucsd.edu
A59005471

Apurva Salvi
asalvi@ucsd.edu
A59005541

Objective:

- To understand and create a functionally working ECC encoder/decoder capable of correcting single bit flips and detecting double bit flips.
- Analyze the performance, resource and accuracy metrics with different optimisations
- Since the operation is primarily a gate level operation (bunch of AND and XOR gates), reach a throughput of 1 operation per cycle
- Integrate the ECC (SECDED) IP core onto the programmable logic using PYNQ and implement streaming

Overview:

We have implemented a functional encoder/decoder for ECC logic. The data bits flow through prefixed patterns and get bit-wised ANDed and XORed to create the ECC bits. The prefixed patterns are programmed such that the codeword created by concatenating data bits and ECC bits will have a minimum hamming distance of 5. Hence, SECDED will be implementable through them. More information on the theory of the prefixed patterns is provided in this paper.

Testing:
For testing, we hook the complementary module into the testbench. For example, if we are synthesizing and studying the Encoder, then the Decoder will be the part of the testbench, and vice versa.
Before passing the data to the Decoder, we introduced single and double bit errors within the testbench at all possible bit-locations to verify the functionality.

Introduction:

The motivation for this project stemmed from the desire to implement an algorithm that is implemented heavily in industry but only briefly touched upon in our general coursework. We wanted to implement this algorithm on the Pynq board, and along the way, experiment with the various optimization techniques taught in CSE 237C.

Error Correction Codes work on the basis of Hamming distance. Data bits are padded with code bits to increase the Hamming distance between valid codewords, thereby making errors in the transmission of message detectable and correctable.

The first step in the process was to generate the prefixed patterns which are used to generate the code bits from the data bits. For Single Error Correction Double Error Detection (SECDED), these patterns had to follow certain rules which are described in detail in the reference paper. We generated these patterns by writing a Python program.

Five such 8-bit patterns were used to create five check-bits (code bits). This was done by a bitwise AND operation of one 8-bit pattern with the 8-bit data, followed by a bitwise XOR to obtain one check-bit. This process was repeated five times with each of the patterns to generate the 5-bit code. The encoder sends the data appended with the code to form a 13-bit codeword.

At the decoder end, the message is received and received code-bits are saved. The decoder performs the operations of the encoder to generate its own code-bits to compare with the received code-bits. They are compared with bitwise-XOR, and the result of the bitwise-XOR operation is called the Syndrome. If the Syndrome is zero, it indicates that the message received is correct. If there is a single error in the message, the Syndrome will match with the codename corresponding to the erroneous bit, and an XOR operation with the data bits will correct the error. If there are two errors in the message, the errors will not be corrected, however, the decoder will be able to indicate there are at least two errors based on the Syndrome.

1. Design Space Exploration:

1.1 Encoder Implementation:

Design	Throughput (MHz)	Timing (ns)	Latency	FF	LUT
ecc_encoder_baseline	2.61	5.634	68	65	159
ecc_encoder_optimized1	4.95	7.212	28	62	142
ecc_encoder_optimized2	11.83	5.634	15	39	184
ecc_encoder_optimized3	11.83	5.634	15	39	184
ecc_encoder_optimized4	511.25	1.956	1	0	46

- **ecc_encoder_baseline:** functionally verified implementation with no optimizations
- **ecc_encoder_optimized1:** inner loop was unrolled
- **ecc_encoder_optimized2:** both inner and outer loop were unrolled
- **ecc_encoder_optimized3:** loops were interchanged
- **ecc_encoder_optimized4:** pipelining enabled in addition to existing optimizations

We manually ensured that Pipelining is off till a certain point in our optimization process, to better attribute changes in results to changes that we made in our code.

We observed that unrolling inner and outer loops led to reduced latency when done together, compared to unrolling just the inner loop.

Interchanging loops on top of unrolling did not have any effect on both latency as well as resource usage. This is to be expected: as both loops are completely unrolled, any recurrence dependence has already been removed. When pipelining was enabled, we were able to achieve latency of 1 cycle which is what we were striving for.

1.2 Decoder Implementation:

Design	Throughput (MHz)	Timing (ns)	Latency	FF	LUT
ecc_decoder_baseline	1.02	5.634	174	112	373
ecc_decoder_optimized1	1.05	6.846	139	116	362
ecc_decoder_optimized2	1.47	5.634	121	110	398
ecc_decoder_optimized3	72.91	6.858	2	7	150
ecc_decoder_optimized4	155.26	6.441	1	15	151

- **ecc_decoder_baseline:** functionally verified implementation with no optimizations
- **ecc_decoder_optimized1:** inner loop was unrolled
- **ecc_decoder_optimized2:** both inner and outer loop were unrolled
- **ecc_decoder_optimized3:** auxiliary for-loops were also unrolled in addition
- **ecc_decoder_optimized4:** pipelining enabled in addition to existing optimizations

We observed that iteratively unrolling the main for-loops in the code led to improved latency as well as reduced resource usage. This consequently led to an increase in performance (throughput) as exhibited in the change in throughput from 1.02 MHz to 72.91 MHz. When pipelining was enabled, we were able to achieve latency of 1 cycle which is what we were striving for.

2. Streaming Architecture:

We implemented streaming using an axis interface as depicted in the image below. We were able to stream an array of 256 inputs into the encoder and streamed the corresponding ECC_bits out also as an array. We verified the results with the precomputed expected values and they matched.

```
void ecc_encoder(
                hls::stream<axis_data_8> &data1,        //in
                hls::stream<axis_data_5> &output        //out
                )
{
#pragma HLS INTERFACE axis port=data1
#pragma HLS INTERFACE axis port=output
#pragma HLS INTERFACE s_axilite port=return
```

We observed that adding streaming to our architecture led to significant degradation of latency as well as increased resource utilization. An interesting observation is that BRAM usage was introduced for the first time in the streaming architecture.

Design	Throughput (MHz)	Timing (ns)	Latency	FF	LUT	BRAM_18K
Streaming	0.25	5.21	783	108	432	2

3. Codebase:

Our various Encoder/Decoder designs, testbench and results are documented in the below github repository.

https://github.com/asalvi98/SECDED