XTEA Coursework (WSC354)

F130812

1. Introduction

Implementing cryptographic algorithms to hardware offers significant advantages in terms of performance, power efficiency, and security, especially in systems with constrained resources. This report presents the development, simulation, and deployment to a partially working implementation of the eXtended Tiny Encryption Algorithm (XTEA) on an Altera DE1-SoC FPGA board. XTEA is a lightweight, symmetric block cipher widely known for its simplicity and suitability for resource-constrained environments.

2. Design and Implementation

This project implements a duplex approach to the XTEA algorithm in VHDL and is designed to manage 128-bit plaintext and key inputs, split across four 32-bit words. The design is comprised of subcomponents that are managed by Finite State Machines to stream data through encryption and decryption pipelines.

2.1 Main Architecture

The top level of this system is the xtea_top_duplex module, which coordinates the behaviour of both the encryption (xtea_enc) and decryption (xtea_dec) cores. Supporting this are FSMs to manage the input sequencing, block assembly and output streaming. These include:

- `key_state`: captures 4x 32-bit words to assemble a 128-bit encryption key.
- `data_state`: buffers incoming plaintext into a 128-bit block.
- `cipher_state`: reconstructs a 128-bit ciphertext from four 32-bit input words.
- `enc_stream_state`: feeds plaintext words to the encryption core sequentially.
- `dec_index`: streams decrypted output words from the decryption core.

2.2 Data and Key Handling

The system first waits for `key_valid` to go high. Four consecutive 32-bit key words are then loaded via the key_state FSM, proceeding to transition through `KEY_IDLE → KEY_0 → KEY_1 → KEY_2 → KEY_3`, then returning to `KEY_IDLE`. Once the key is fully assembled, it becomes available to both cipher cores via the shared `full_key` signal.

Once the key is ready, the system waits until `data_valid` is high and begins buffering the four 32-bit words to the `plaintext_buf` via the `data_state` FSM. When this is accomplished, the `plaintext_valid` flag is raised, upon which the encryption stream FSM begins feeding the input the cipher.

When the ciphertext is received, the `cipher_state` FSM captures it across four clock cycles and captures it as `ciphertext_buf`. Following this, `ciphertext_valid_internal` is triggered which initiates the decryption.

2.3 Encryption and Decryption Cores

Each of the encryption and decryption cores process the 128-bit data in a 32-round XTEA operation, converting four 32-bit input words into four 32-bit output words.

For encryption, `xtea_enc` begins once `plaintext_valid` is asserted, and then the `enc_stream_state` FSM sends each 32-bit word to the core, one at a time.

For decryption, xtea_dec is initiated via `ciphertext_valid_internal`, after all four 32-bit words are buffered in.

Each time either of the encryption or decryption cores output a 32-bit word, it raises a `*_valid` signal to show that the output is ready. The top-level controller uses this to store the word. For instance, the `dec_index` FSM tracks and stores four 32-bit decrypted outputs from `xtea_dec`.

This behaviour is shown in Figure X, showing the progress through the loading and advancement of the four 32-bit words to progress through the process.

2.4 XTEA Core Internals and Round Logic

The `xtea_enc` and `xtea_dec` modules implement the functionality of the XTEA algorithm, operating on 128-bit blocks and perform 32 rounds of Feistel transformations.

Each 128-bit input block is split into four 32-bit words named `v0, v1, v2,` and `v3`, which are loaded into internal registers at the start LOAD state. These values are then iteratively modified within the EXEC state where each cycle performs one round of the XTEA algorithm.

The 128-bit key is divided into four subkeys labelled as k[0] to k[3] and each round uses different subkeys selected by using parts of the evolving `sum` value to decide which of the four keys to use. This creates variation between the rounds and increases the cipher's complexity. The

The constant known as `delta` = 0x9E3779B9 is added or subtracted from `sum` each round to add variation to the algorithm and increase diffusion. This constant is selected to ensure that small changes in input produce large changes to the output. By varying `sum` each round, non-linearity is introduced to the algorithm.

The main difference between the encryption and decryption is the direction of the round progression:

- Encryption core `sum` value starts at 0 and is incremented by delta each iteration
- Decryption core's `sum` value starts at `delta × 32` and is decremented each iteration

Upon completion of all 32 rounds, the `v0-v3` values are the encrypted or decrypted 128-bit results when concatenated. Though to output these results, they are then streamed back out the `OUT0` to `OUT3` with each word having a corresponding `*_valid` signal so that the top controller knows that the output is ready to be latched.

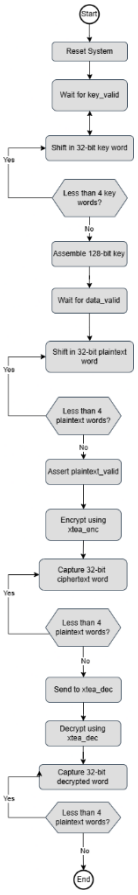

Figure 1 - Key and Text Loading Flow for XTEA

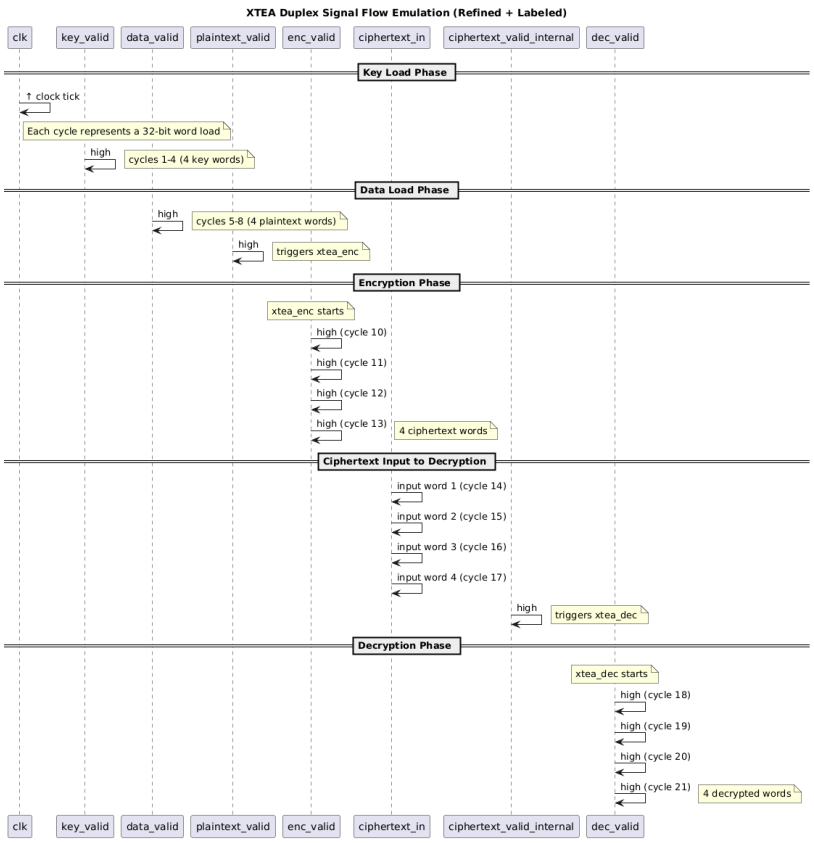

Figure 2 - Signal Flow for XTEA Duplex

3. Simulation and Validation

For verification of the design of the XTEA duplex system, simulation was performed first using TerosHDL and then Questasim 10.1d for waveform viewing. The provided VHDL testbench, provided by the module leader, was used without modification. The top level `xtea_top_duplex` was structured to interface with the testbench which applies known key and plaintext vectors, captures the resulting ciphertext which is then re-fed into the decryption path to validate the full encryption and decryption cycle.

The DUT was designed to follow this structure of the testbench exactly, with the FSM timing and handshaking signals verified to align with the testbench's expectation. The design correctly raised the expected valid and ready signals during key and data input, encryption, ciphertext output and decryption.

Encryption processes and behaviours were observed to be correct. The waveforms in the simulation show that the plaintext is correctly processed into ciphertext through the expected timing window. The four ciphertext words appear in correct sequence consistently.

Decryption on the other hand was not deemed successful. Although the ciphertext outputted by the encryption core was fed back to the decryption core, and `xtea_dec` produced an output by iterating the ciphertext, `data_word_out` did not match the original inputted plaintext. This was

seen for all results tested and suggests a logic-level issue within the decryption round execution. The FSM signalling and synchronisation seemed correct, suggesting that with more work the isolated bug in `xtea_dec` could be resolved with fixes to the internal logic.

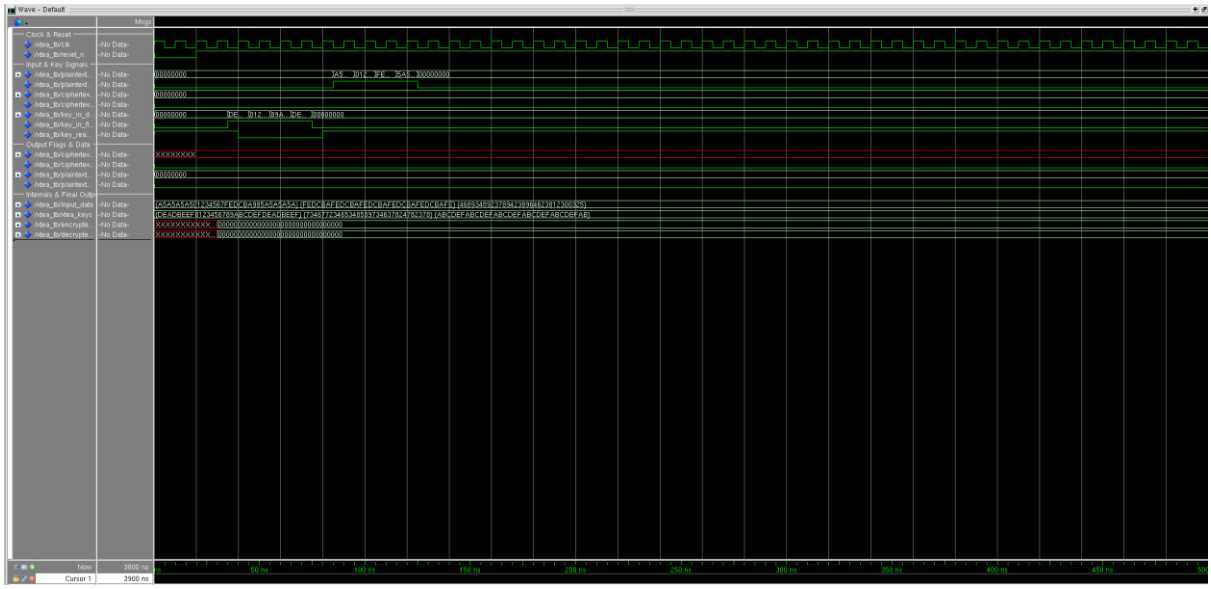

Figure 3 - Partial Waveform Showing Key Loading, Plaintext and Ciphertext

This waveform segment shows the initial startup of the testbench where the correct loading of the 128-bit key and plaintext input can be seen across four cycles.

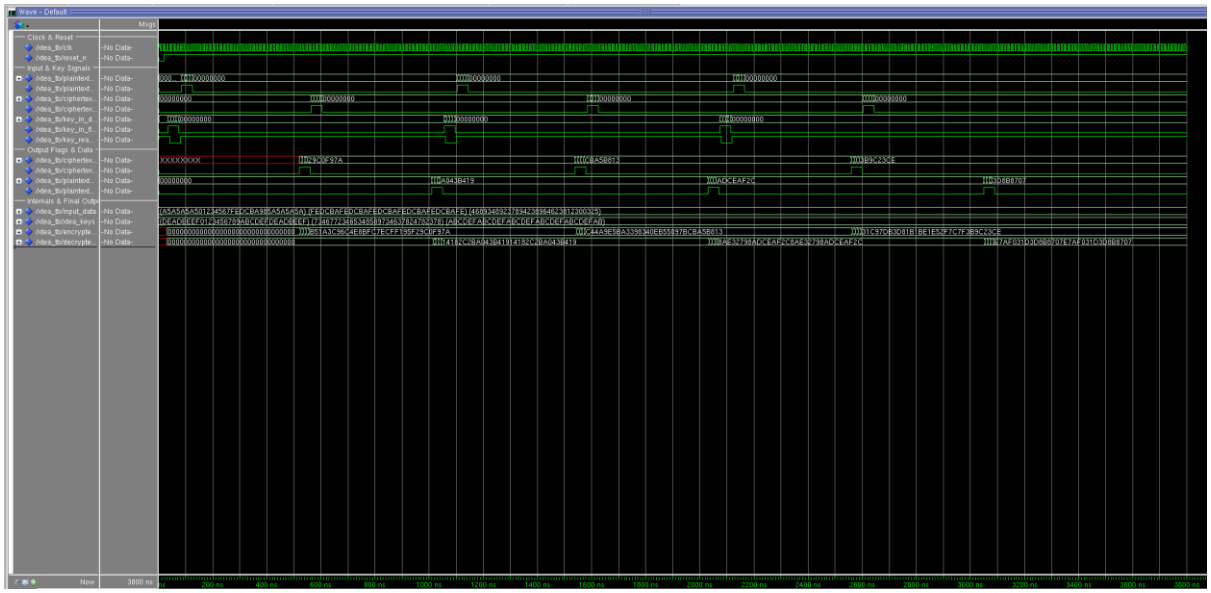

Figure 4 - Full Waveform Showing Entire Duplex Path

Following encryption, the resulting ciphertext is then fed into the decryption path. The `ciphertext_valid_internal` signal is raised to indicate that decryption is to begin, and the decryption core outputs words. However, these values do not match the loaded plaintext, confirming the functional failure suspected in the decryption logic.

4. Deployment and Evaluation

This section describes the physical deployment of the `xtea_top_duplex` design to the Altera DE1-SoC development board, as well as an evaluation of its performance in area, power and timing.

4.1 Deployment Environment and Toolchain

The FPGA development was completed in a virtual Linux environment using Ubuntu installed inside VirtualBox on a Windows host. This setup allowed for compatibility with the Intel Quartus Prime Lite 24.1 software, used for synthesis, place and route, timing and power analysis.

Due to the limitations of USB pass-through in virtual environments, significant setup and lots of time was required to route the JTAG interface to the virtual system. After these issues were resolved, Tcl commands which were then made into scripts were written and used to start project creation, file inclusion and device configuration.

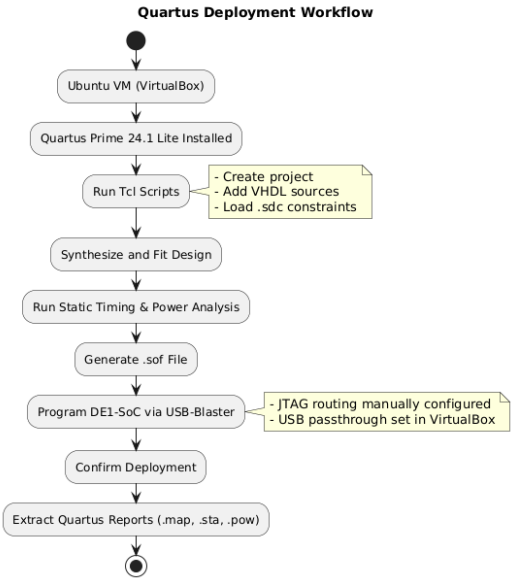

Figure 5 - Visual summary of the complete Quartus deployment workflow, from environment setup to the programming of the DE1-SoC board.

4.2 Results

The `xtea_top_duplex` design was compiled and fitted for the Cyclone V 5CSEMAF316 device. Upon compilation, Quartus automatically generated reports and summaries which are detailed below:

4.2.1 Timing

Metric	Value
Worst-case Slack	3.247ns
Fmax (clk)	70.37 MHz
Required Clock Period	20.000ns
Achieved Clock Period	16.753ns

Table 1 - Timing Metrics for Part 1 Deployment

The design comfortably meets the 50 MHz timing requirement with comfortable margin and in theory shows that it could operate up to ~70MHz based on the generated synthesis results. Such low slack suggests that there is space for future pipeline extensions or more complex logic or cryptography algorithms.

4.2.2 Power

Power results were gathered post-fitting using a cusom Quartus TCL script, which ran the `quartus_power` command and left the setting and outputs to be handled programmatically.

```
exec quartus_pow $designName --read_settings_files=on --write_settings_files=off
```

Component	Power (mW)
Total Thermal Power	451.33
Core Static Power	411.41
Core Dynamic Power	9.44
I/O Power	30.48
Toggle Rate (Average)	5.45 Mtrans/s

Table 2 - Power Metrics for Part 1 Deployment

Static power is observed to take up the majority of the power consumption at 91.2%, which is consistent with a low-activity functional design that doesn't use aggressive switching.

These results support that the design is highly energy-efficient and has the potential for low-power integration into security pipelines when not heavily switching.

Although not implemented here, UART transmission to log output registers would hope to be added with further development. This would more accurately represent the activity levels of the deployed design.

4.2.3 Resource Utilisation (Area)

Resource Type	Used	Available	Ulitisation
Logic ALMs	920	32,070	3.0%
Total Registers	1,622	32,070	5.0%
RAM Blocks	0	397	0%
Block Memory Bits	0	4,065,280	0%
DSP Blocks	0	87	0%
Total LABs Used	128	3,207	~4.0%

Table 3 - Resouce Utilisation for Part 1 Deployment

The core used 920 ALMs and no RAM or DSPs, the supports the lightweight profile and shows suitability for small-footprint security applications.

4.3 Deployment and Board Testing

The .sof file generated following compilation was successfully flashed to the DE1-SoC board using Quartus Programmer tools. However, due to the few output mappings there was no external validation of the functional behaviour. However, deployment still confirmed:

- End-to-end Quartus toolchain compatibility
- Working JTAG programming despite complications with VirtualBox passthrough
- Clean timing and routing closure

- Minimal resource and power usage measured

4.4 Future Considerations

Despite not developing an implementation that exercises the full I/O capabilities of the board, the results confirm the expectation that the XTEA duplex core is highly efficient in terms of area and energy. The high slack margin and number of unused memory blocks indicate that there is room for scalability in future.

Opportunities for future work could explore integration of UART to allow for inspection of encryption and decryption outputs. Another addition could be a more realistic SystemVerilog input generator, for instance more varied and realistic data could be streamed using a range of random keys to give a better idea of how the system might perform under real-world conditions.

5. System Integration with Traffic and Routing

5.1 Overview and objectives

To develop the standalone XTEA core developed in Part 1 of the coursework, this next section integrates a router system that streams data to emulate a secure communication between multiple sources.

Two input generators (`ip_enc_gen` and `ip_dec_gen`) simulate data sources each emitting 8 data bits and 2 priority bits. These are routed by a the `mini_router` module, which selects one input per cycle based on priority and round-robin logic. The selected data is buffered and processed by the `cs_top` level controller which handles the triggering of encryption and decryption whilst keeping modules in sync.

5.2 Input Generators

Two SystemVerilog modules, `ip_enc_gen` and `ip_dec_gen,` were implemented for a simulation of data feeding into the system. Each sends a 10-bit packet of 8 bits of data and 2 bits of priority.

The `ip_enc_gen` outputs a fixed 128-bit plaintext which gets split into 16 bytes sent over 16 cycles. Each 8-bit data byte is combined with a 2-bit priority tag to from a 10-bit packet.

The `ip_dec_gen` follows the same pattern however it transmits encrypted traffic and is used to test the mini routers arbitration logic.

These modules test the routers' ability to arbitrate the competing data streams, thus reflecting real-world traffic conditions.

5.3 Router Logic

A custom VHDL router (`mini_router`) was developed to manage the concurrent inputs generated from `ip_enc_gen` and `ip_dec_gen.` Each generator sends 10-bit outputs, and the router selects one per clock cycle and forwards the 8-bit payload to the next stage.

Precedence is given to the received data with the higher input priority value, if both signals are active and have equal priority, a round-robin method is used to alternate between the `ip_enc_gen` and `ip_dec_gen` outputs, preventing one of them from being avoided completely. Only one source is granted access per cycle, and this is shown by corresponding `grant1` and `grant2` to acknowledge which signal was chosen.

5.4 Top Level Control

The `cs_top` module oversees, organises and synchronises the entire pipeline. It receives 8-bit packets from the router and concatenates them into a 128-bit register over 16 cycles. Once full, `cs_top` asserts a valid signal to initiate the encryption of this message using `xtea_core`. Four 32-bit ciphertext words are then transmitted back over successive cycles.

The ciphertext words are then again collected into 128-bits and then passed to the `xtea_dec` for decryption. The results of which are then again assembled using four 32-bit words and stored as the final decrypted result. During this whole process, internal FSMs manage the handshaking sequencing of each section to ensure clean continuity through the stages.

5.5 Simulation and Testing

Simulation was conducted very similarly to Part 1 whereby QuestaSim was used for its waveform tools to analyse behaviour of the input stream into the encryption and then looped back to decryption.

The ip_enc_gen was observed to successfully transmit the full 128-bit plaintext block which was routed, buffered and then passed to the encryption core. The `xtea_enc` returned the ciphertext words, affirming encryption was working. These are then forwarded to the decryption core.

Sadly, the `xtea_dec` module did not return the original plaintext. Although through inspection of the FSM transitions and handshaking signals, and a present ciphertext, the output did not output a valid result. More development would need to be done to the `cs_top` and `xtea_dec` files to resolve the issue.

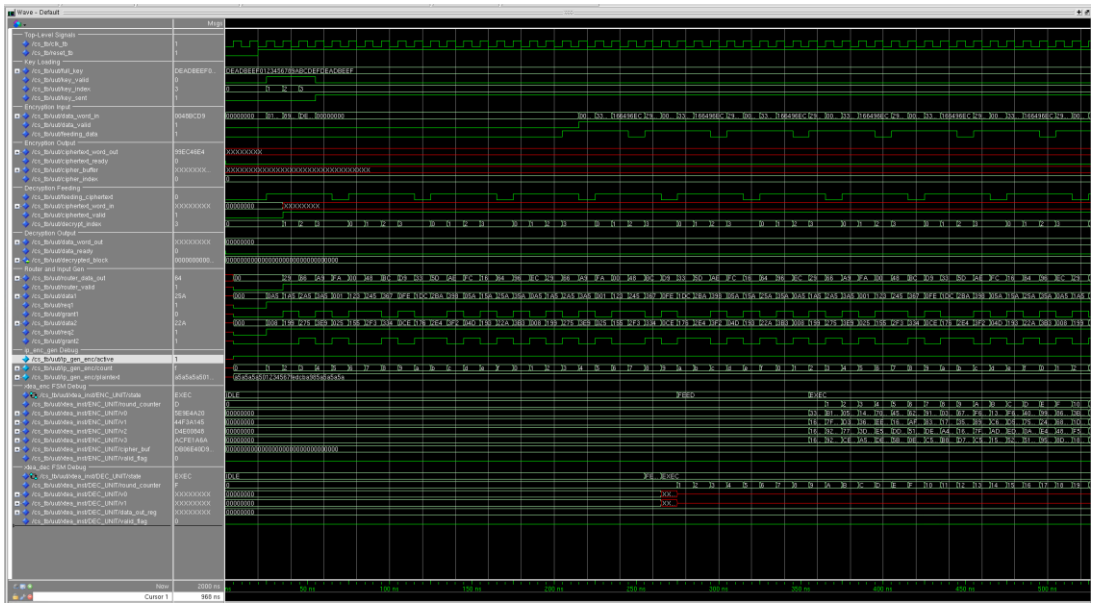

Figure 6 - Zoomed waveform of startup and encryption cycle

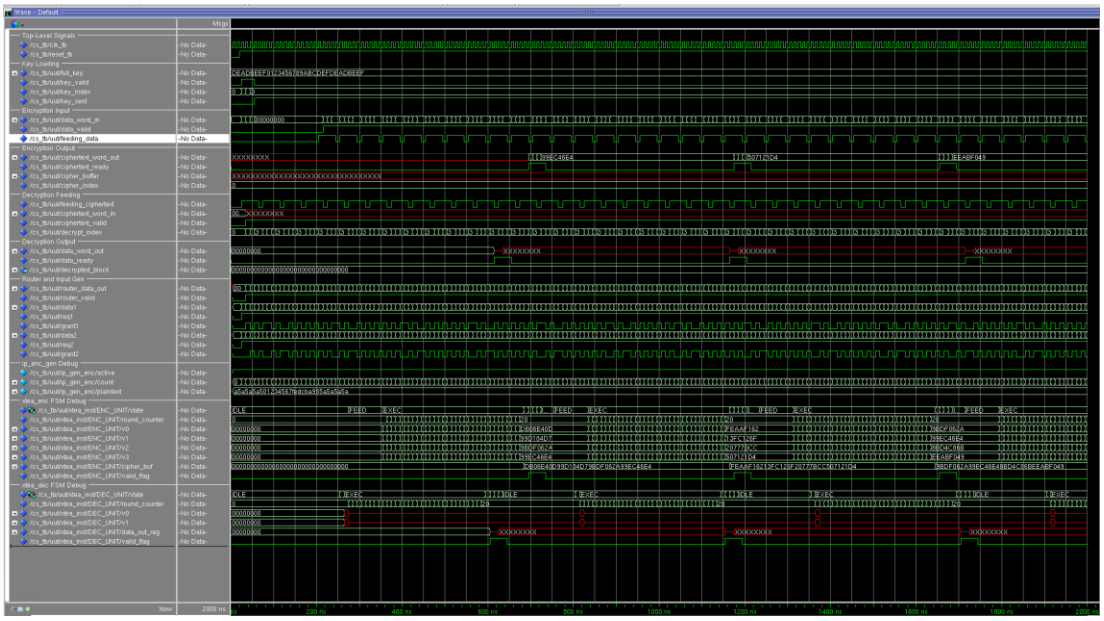

Figure 7 - Full simulation showing encryption and decryption sequence

5.6 Synthesis and Deployment

Synthesis and deployment were integrated very similarly to Part 1, showcased in Section 4.1. Differences were seen regarding the additional SystemVerilog files added to the project and a slightly modified version of the Tcl script. Synthesis was completed but I am sceptical of the results considering how few ALMs were used and how quickly, compilation was completed. More work needs to be done to move from simulation to synthesis of this design. Regardless, what results were obtained as presented in the following sections.

5.6.1 Synthesis Results

The synthesis was completed without errors shown by cs_top.fit and cs_top.map reports. The overall usage was minimal

Resource Type	**Used**	**Available**	**Ulitisation**
Logic ALMs	13	32,070	<1%
Total Registers	24	32,070	<1%
RAM Blocks	0	397	0%
Block Memory Bits	0	4,065,280	0%
DSP Blocks	0	87	0%
Total LABs Used	140	3,207	31%

Table 4 - Memory Utilisation for Part 2 Deployment

No memory or DSP blocks were, used as expected given the design's lightweight design and minimal processing.

5.6.2 Timing Analysis

Timing closure was comfortably achieved with no timing errors shown and good slack values.

Model	Worst-case Slack	Fmax (clk)
Slow 1100mV 85°C	17.557 ns	409.33 MHz
Fast 1100 0°C	18.668 ns	389.26 MHz

Table 5 - Timing for Part 2 Deployment

Given the 50 MHz clock, system exceeded the performance requirements.

5.6.3 Deployment and Limitations

Although synthesis resulted in a generated .sof file, hardware deployment failed due to reported JTAG errors. Although it is reported that JTAG errors caused failure, it is hard to believe this since Part 1 deployment worked without issue. One potential cause may have been the addition of SystemVerilog files which caused mixed language errors within Quartus compilation leading to poor deployment. Synthesis logs also report missing or removed registers suggesting the design was insufficient and lacked realistic toggling. This also explain that power consumption could not be measured due to a failed programming sequence.

6. Conclusion

This project showed the implementation and simulation of XTEA encryption and decryption on an FPGA, first introduced as an isolated system on FPGA, then combined with a larger refeeding pipeline. Correct encryption was demonstrated but decryption failed due to suspected errors in logic. Despite this successful deployment was done, though the system lacked real data transmission so could not be evaluated fully.

In Part 2, the system was further expanded with traffic generators and a priority and round-robin based router. Simulation again showed correct data for routing and encryption, but decryption failed due to incorrect top-level design and decryption logic. Synthesis did complete but results were not convincing and likely prevented the successful deployment to hardware.

Despite these limitations through development, the system architecture proved that it could be functionally sound with more time. Future work should aim to resolve decryption errors, improve signal visibility and ensure better deployment through better I/O mapping and validation.