Implementation of the XTEA Algorithm Utilising a DE1-SoC FPGA

Chris Holland

***Abstract*— This report covers how the XTEA cryptography algorithm can be implemented utilising VHDL, comparing its VHDL implementation to its C based counterpart. This report shall be conducting all FPGA work on the DE1-SoC board provided, monitoring each step of the design process, explaining why decisions were made and what impact they had on the final solution. The end goal is to get an FPGA implementation working in unison with the C based implementation, encoding in software and decoding in hardware.**

***Index Terms*— XTEA, FPGA, Cortex A9, VHDL, Quartus Prime, Encryption, Decryption, C, Cryptography, Performance, Area & Power, Instantiation, HPS**

Table of Contents

I. INTRODUCTION

The task provided to our cohort was titled 'Embedded Cryptography'. This can be broken down into the process of taking a set of characters and a key, using bitwise operations on that character and key, and receiving a seemingly random stream of characters that no human could understand known as cyphertext. Only when this process is performed in reverse, reconstructing the message can the user understand what was originally sent.

The task was further described as an FPGA implementation of the XTEA (eXtended TEA) algorithm [1] utilising a DE1-SoC by Altera [2]. The end goal is to run a C application on the HPS (High-Performance Substrate) that retrieves a message from the user, then utilising the XTEA algorithm creates cyphertext using a provided key. This cyphertext is then passed to the FPGA internally, where it is then decoded in VHDL (VHSIC (Very High-Speed Integrated Circuits) Hardware Description Language) utilising the same key, then outputted back to the HPS.

In the current hardware case, the DE1-SoC's HPS is a cortex A9, the two methods of running applications on the A9 is either bare metal or via an operating system. For this scenario, it was recommended by our lecturer to use an operating system, specifically Linux.

Breaking down the task into sections, it can be split into:

- Research of the XTEA algorithm & provided hardware.
- The algorithm's C implementation.
- The algorithm's FPGA implementation.
- How the FPGA implementation performs.
- Communication between the FPGA and the HPS.

These sections will be the structure for this report, sharing the discoveries made along the way, explaining how those discoveries shaped the eventual solution.

Given that an Altera board has been provided, it has been recommended that Quartus Prime be used for VHDL implementation, whilst utilising ModelSim to collect and check the resulting waveform produced by the FPGA [3][4]. For the C implementation, VSCode (Visual Studio Code) will be used [5].

A. *Motivation – Commercial Relevance*

FPGAs are an emerging technology in industry, as computational tasks get more and more complex it becomes valuable to companies to have access to hardware that is capable of computing tasks as fast as possible. Conventional CPU & GPU architecture is good at generic tasks, being able to perform a never-ending wide range of functions, whilst for tasks such as encryption and decryption other computational architectures and styles are available.

One such architecture is ASICs (Application Specific Integrated Circuits). These ASICs are designed purely to perform a single task, such as encryption/decryption. There is a wide range of tasks where ASICs are better suited when compared to standardised CPU & GPU based computing, some examples of these are computing cryptographic hash functions such as SHA256, which is the algorithm used to mine bitcoin. As mentioned in [6] "The hash rate of most GPU units is below 1GH/s, and as of 2014, some single ASIC units are able to reach speeds of over 1,000GH/s while consuming far less power than used by a GPU". These application-specific chips are non-reprogrammable instances of an FPGA, giving FPGAs all the bonuses of an ASIC, with added field programmability, allowing the user to redesign their FPGA, reducing costs of developing a circuit relative to an ASIC.

Cryptography is well known to be one of the fastest-growing industries in modern electronics, in a recent survey, monitoring the likelihood of a business adopting emerging technology, cybersecurity and encryption was ranked at 83%, meaning 83% of companies surveyed planned to adopt encryption and cybersecurity by the year 2025, beating Artificial intelligence and automation within industry [7].

B. *Goal of Report*

The goal of this report is to share the process of designing and implementing the XTEA algorithm on both traditional CPU architecture and FPGA architecture, finding points in the algorithm that can be simplified or made more efficient using FPGA based design.

II. DISCOVERY

Before any design could be created, it was imperative to perform some research on the algorithm and the hardware provided. This would allow a design to be created considering all steps of the process, allowing simple transitions between different phases of the project.

A. *XTEA Algorithm*

Initially, the algorithm was studied to understand how the algorithm broke down text and what components of the inputs provided were used to generate the output provided. The full 64-bit implementation of the XTEA algorithm in C can be seen below:

```c
#include <stdint.h>

/* take 64 bits of data in v[0] and v[1] and 128 bits of key[0] - key[3] */

void encipher(unsigned int num_rounds, uint32_t v[2], uint32_t const key[4]) {
    unsigned int i;
    uint32_t v0=v[0], v1=v[1], sum=0, delta=0x9E3779B9;
    for (i=0; i < num_rounds; i++) {
        v0 += (((v1 << 4) ^ (v1 >> 5)) + v1) ^ (sum + key[sum & 3]);
        sum += delta;
        v1 += (((v0 << 4) ^ (v0 >> 5)) + v0) ^ (sum + key[(sum>>11) & 3]);
    }
    v[0]=v0; v[1]=v1;
}

void decipher(unsigned int num_rounds, uint32_t v[2], uint32_t const key[4]) {
    unsigned int i;
    uint32_t v0=v[0], v1=v[1], delta=0x9E3779B9, sum=delta*num_rounds;
    for (i=0; i < num_rounds; i++) {
        v1 -= (((v0 << 4) ^ (v0 >> 5)) + v0) ^ (sum + key[(sum>>11) & 3]);
        sum -= delta;
        v0 -= (((v1 << 4) ^ (v1 >> 5)) + v1) ^ (sum + key[sum & 3]);
    }
    v[0]=v0; v[1]=v1;
}
```

Figure 1 - 64 Bit C XTEA Algorithm implementation [8]

This implementation of the XTEA algorithm takes 64 bits of data and after an unspecified number of iterations returns 64 bits of output cyphertext. For example, Unicode uses

between 8 and 32 bits per character. This is so that Unicode can represent characters from languages all over the world. If encoding and decoding Unicode characters, there is some relevance in utilising a larger implementation of the XTEA algorithm. For example, a 128-bit algorithm was provided. This in theory would double the speed of the algorithm, considering it is working on double the number of bits at any one time. Here is the C implementation of the 128-bit XTEA algorithm:

```
// XTEA: 128-bits
void xtea_enc(uint32_t **dest, const uint32_t **v, const uint32_t **k) {
  uint8_t i;
  uint32_t y0 = *v[0], z0 = *v[1], y1 = *v[2], z1 = *v[3]
  uint32_t sum = 0, delta = 0x9E3779B9;
  for(i = 0; i < 32; i++) {
    y0  += ((z0 << 4 ^ z0 >> 5) + z0) ^ (sum + *k[sum & 3]);
    y1  += ((z1 << 4 ^ z1 >> 5) + z1) ^ (sum + *k[sum & 3]);

    sum += delta;
    z0  += ((y0 << 4 ^ y0 >> 5) + y0) ^ (sum + *k[sum>>11 & 3]);
    z1  += ((y1 << 4 ^ y1 >> 5) + y1) ^ (sum + *k[sum>>11 & 3]);

  }
  *dest[0]=y0; *dest[1]=z0; *dest[2]=y1; *dest[3]=z1;
}

void xtea_dec(uint32_t **dest, const uint32_t **v, const uint32_t **k) {
  uint8_t i;
  uint32_t y0 = *v[0], z0 = *v[1], y1 = *v[2], z1 = *v[3];
  uint32_t sum = 0xC6EF3720, delta = 0x9E3779B9;
  for(i = 0; i < 32; i++) {
    z1  -= ((y1 << 4 ^ y1 >> 5) + y1) ^ (sum + *k[sum>>11 & 3]);
    z0  -= ((y0 << 4 ^ y0 >> 5) + y0) ^ (sum + *k[sum>>11 & 3]);
    sum -= delta;
    y1  -= ((z1 << 4 ^ z1 >> 5) + z1) ^ (sum + *k[sum & 3]);
    y0  -= ((z0 << 4 ^ z0 >> 5) + z0) ^ (sum + *k[sum & 3]);
  }
  *dest[0]=y0; *dest[1]=z0; *dest[2]=y1; *dest[3]=z1;
}
```

Figure 2 - 128-Bit C XTEA Algorithm Implementation

There are plenty of similarities between the 64-bit and 128-bit implementations, with the only difference being that instead of having two 32-bit values incremented over the loop, v0 and v1, there are four 32-bit values incremented over the loop. Those values being y0, y1, z0, and z1.

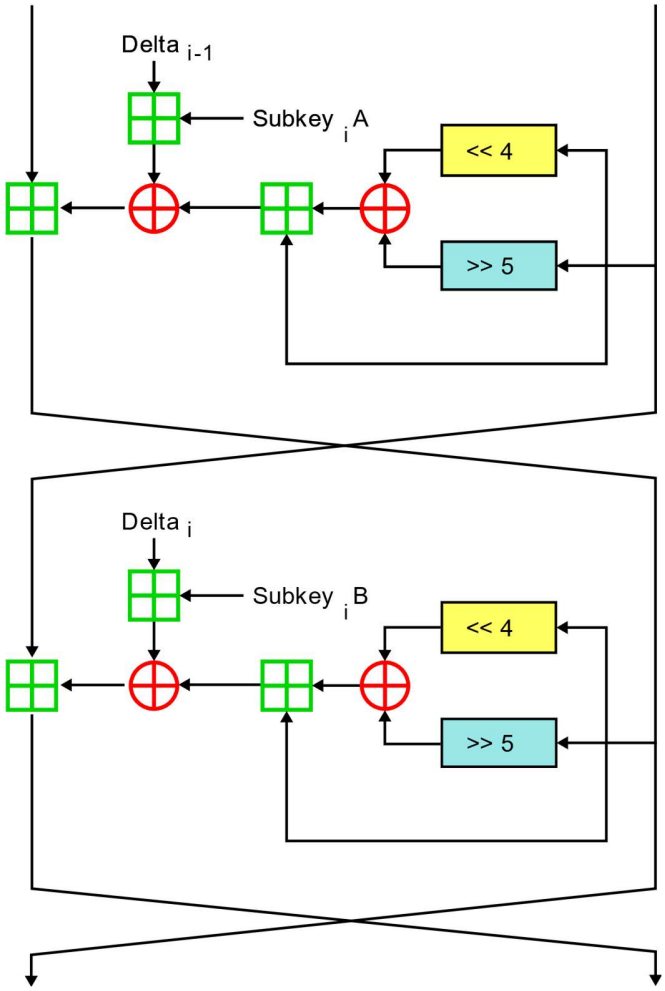

Figure 3- Two Feistel rounds (Once Cycle) of XTEA

The algorithm can be described as taking 32 bits in, performing some bitwise operations such as XOR (^) and bitwise shifts (<< / >>) alongside the addition of a constantly updating sum value which is incremented or decremented each cycle based on decryption or encryption being performed. The value of this calculation is then added to the current value of the 32-bit input value. This process is then repeated for another feistel round, with the use of a different subkey. A subkey can be described as a 32-bit part of a 128-bit key. The subkey used in feistel round 1 for encryption, subkey $_i$A can be seen as the 2 least significant bits of the sum variable, whereas subkey $_i$B can be seen as the 11^{th} and 12^{th} bits of the sum variable. These two bits are used to select which 32-bit word of the key shall be selected, with “00” selecting the first word in the key, and “11” selecting the last. Once this process has been repeated for two feistel rounds, the process is repeated an arbitrary number of times, for the sake of this implementation, 32 iterations were utilised.

Whilst understanding the algorithm, it was noticed that the encoding and decoding modules shared computational work, with the first feistel round of encoding being identical to the second feistel round of decoding, and vice-versa. This realisation shaped the design of the FPGA, allowing the same instantiated component to run as either decoding or encoding module, depending on the order that the blocks are run in. This

kind of optimization allows for both power and area savings, instead of instantiating two components, one decoder and one encoder. It is instead possible to utilise the same component to perform both encryption and decryption. Finding more use out of components already on a design helps to increase the overall efficiency of the design.

B. HPS – FPGA Bridge

As a part of the initial project brief was to include communication between the HPS and the FPGA, research was conducted into how this would be made possible.

After consulting the DE1-SoC Computer System with ARM* Cortex* A9 manual, section 2.5 – FPGA Bridges [9] and the SoC-FPGA Design Guide, section 7.3 – HPS-FPGA Interfaces [10], it was understood that the DE1-SoC board has an inbuilt set of bridges that connect the HPS and the FPGA. These bridges are enabled and disabled using the bridge reset register, with a given address. Three bridges exist, those being the HPS-to-FPGA bridge, a high-performance bus with a configurable data width of 32, 64 or 128 bits of data. A lightweight HPS-to-FPGA bridge, which is a 32-bit fixed data width bus, designed for lightweight data transfer. And the FPGA-to-HPS bridge, another 32-bit to 128-bit bus configurable to the desired width, facilitating data transfer from the FPGA back to the HPS. These bridges can be referenced with their base addresses, which can be found in section 2.6.2 of the DE1-SoC_Embedded_Linux_Systems_1.9 Document [11].

Slave Identifier	Slave Title	Base Address	Size
STM	STM	0xFC000000	48 MB
DAP	DAP	0xFF000000	2 MB
LWFPGASLAVES	FPGA slaves accessed with lightweight HPS-to-FPGA bridge	0xFF200000	2 MB
LWHPS2FPGAREGS	Lightweight HPS-to-FPGA bridge GPV	0xFF400000	1 MB
HPS2FPGAREGS	HPS-to-FPGA bridge GPV	0xFF500000	1 MB
FPGA2HPSREGS	FPGA-to-HPS bridge GPV	0xFF600000	1 MB

Figure 4 - HPS Peripheral Region Address Map

With both the information about the algorithm and the bridges understood, it was possible to move onto the various implementations of the XTEA algorithm.

III. C IMPLEMENTATION

Implementing the XTEA algorithm in C has its benefits, for one, console printing allows the algorithm to be stepped through, and each step of the algorithm can be displayed for comparison when creating the FPGA implantation. C gives a lot more flexibility with outputting and is generally much easier than implementing in FPGA.

In addition to the step-by-step breakdown, as a part of the project brief, it is imperative to encode in software. This means the encoding step must be done in C.

These two reasons to create a C implementation can be split into two separate C files. The C XTEA implementation itself, with prints at each step, and another C file that requests an input, splits this input up into 128-bit chunks and applies the algorithm. Taking the result and passing it through the HPS-FPGA bridge and retrieving the result for printing.

A. Step by Step Breakdown of Algorithm

This first C code as previously mentioned can be used to test the FPGA implementation, to ensure that the output and internal signals at each iteration of the algorithm match as they should.

This Involved taking the supplied 128-bit XTEA implementation and adding in additional prints to output the internal variables of the algorithm to a console. This involved splitting the code into an increment variable, what the expected y0, y1, z0, and z1 values should be for a given iteration, alongside outputting the result of the algorithm for comparison.

```
// XTEA: 128-bits
void xtea_enc(uint32_t **dest, const uint32_t **v, const uint32_t **k) {
  uint8_t i;
  uint32_t y0 = *v[0], z0 = *v[1], y1 = *v[2], z1 = *v[3], y0increment = 0, y1increment = 0, z0increment = 0, z1increment = 0;
  uint32_t sum = 0, delta = 0x9E3779B9;
  printf("Encoding...\n");
  printf("Initial Values y0 : %x, y1 : %x, z0 : %x, z1 : %x, sum: %x\n", y0, y1, z0, z1, sum);
  for(i = 0; i < 32; i++) {
    printf("-----------Iteration : %d ----------------\n", i);
    y0increment  = ((z0 << 4 ^ z0 >> 5) + z0) ^ (sum + *k[sum & 3]);
    y1increment  = ((z1 << 4 ^ z1 >> 5) + z1) ^ (sum + *k[sum & 3]);
    y0 = y0 + y0increment;
    y1 = y1 + y1increment;
    printf("y0inc : %x, y0 : %x, y1inc : %x, y1 : %x\n", y0increment, y0, y1increment, y1);
    sum += delta;
    z0increment  = ((y0 << 4 ^ y0 >> 5) + y0) ^ (sum + *k[sum>>11 & 3]);
    z1increment  = ((y1 << 4 ^ y1 >> 5) + y1) ^ (sum + *k[sum>>11 & 3]);
    z0 = z0 + z0increment;
    z1 = z1 + z1increment;
    printf("sum: %x, z0inc : %x, z0 : %x, z1inc : %x, z1 : %x\n", sum, z0increment, z0, z1increment, z1);
  }
  *dest[0]=y0; *dest[1]=z0; *dest[2]=y1; *dest[3]=z1;
}
```

Figure 5 – 128-Bit XTEA Algorithm with Internal Variables Outputted

Comparing this to figure 2, the base 128-bit XTEA C implementation, multiple prints have been added in addition to splitting the increment step into two stages so that this increment can be outputted. Compiling this C file alongside a file that calls the function with parameters equal to the provided VHDL testbench provides a console output with the internal variables for each step of the algorithm.

```
Input Data  : 0xa5a5a5a51234567fedcba985a5a5a5a
Input Key   : 0xdeadbeef123456789abcdefdeadbeef
Encoding...
Initial Values y0 : 5a5a5a5a, y1 : 1234567, z0 : fedcba98, z1 : a5a5a5a5, sum: 0
-----------Iteration : 0 ----------------
y0inc : 37b7b803, y0 : 9212125d, y1inc : dbb0a3cd, y1 : dcd3e934
sum: 9e3779b9, z0inc : cb26ff37, z0 : ca03b9cf, z1inc : d44ecd95, z1 : 79f4733a
-----------Iteration : 1 ----------------
y0inc : 578c7ca5, y0 : e99e8f02, y1inc : 319e44db, y1 : e722e0f
sum: 3c6ef372, z0inc : b5d0ab83, z0 : 7fd46552, z1inc : c851a756, z1 : 42461a90
-----------Iteration : 2 ----------------
y0inc : 431f6385, y0 : 2cbdf287, y1inc : 552b8cbd, y1 : 639dbacc
sum: daa66d2b, z0inc : 932a8171, z0 : 12fee6c3, z1inc : fa36c0f8, z1 : 3c7cdb88
-----------Iteration : 3 ----------------
y0inc : fb2cadd3, y0 : 27eaa05a, y1inc : bbff15fe, y1 : 1f9cd0ca
sum: 78dde6e4, z0inc : f0f4552f, z0 : 3f33bf2, z1inc : 4f451f23, z1 : 8bc1faab
-----------Iteration : 30 ----------------
y0inc : 964fbfcb, y0 : 81121888, y1inc : 26d61c99, y1 : 5dec63e5
sum: 28b7bd67, z0inc : 915e4c9a, z0 : ed93c353, z1inc : 3d70bd62, z1 : 56bf895f
-----------Iteration : 31 ----------------
y0inc : cc81122b, y0 : 4d932ab3, y1inc : c7698f4f, y1 : 2555f334
sum: c6ef3720, z0inc : e0e3219f, z0 : ce76e4f2, z1inc : b1d9ec8a, z1 : 89975e9
Data : 0x89975e92555f334ce76e4f24d932ab3
```

Figure 6 - Console Output of XTEA Algorithm with Internal Variables Outputted steps 4-29 skipped.

This console output also shows decoding, however, it is identical, for this report, only console output for encoding will be displayed. This was validated by checking the output relative to an identical set of data and keys attempted in [1].

B. Main C Implementation

The purpose of the next C file is to utilise the algorithm previously implemented and tested and create a script that will handle inputted data into 128-bit chunks, alongside communication with the FPGA in both directions. It is simple enough to include the previous script and call the encode function, passing the correct parameters. Due to time constraints, this file was not finished, however, the current working file can be found as part of the files submitted alongside this report.

IV. FPGA Implementation

A. Testbench

As part of the project brief, we were provided with a testbench that our FPGA solution must interface with. Because of this limitation provided, it is imperative that the top-level solution interfaces with this testbench, to retrieve data and send data meeting the standard set by the testbench.

Starting with the data incoming portion of the testbench:

```
-- Set mode to encryption
encryption_flag  <= '1';
WAIT FOR clk_period;
-- Write in key and data, updating data on falling edge
WAIT UNTIL FALLING_EDGE(clk);
key_data_in_flag <= '1';
input_key        <= xtea_keys(i)(127 DOWNTO 96);
input_data       <= input_data_array(i)(127 DOWNTO 96);
WAIT FOR clk_period;
input_key        <= xtea_keys(i)(95 DOWNTO 64);
input_data       <= input_data_array(i)(95 DOWNTO 64);
WAIT FOR clk_period;
input_key        <= xtea_keys(i)(63 DOWNTO 32);
input_data       <= input_data_array(i)(63 DOWNTO 32);
WAIT FOR clk_period;
input_key        <= xtea_keys(i)(31 DOWNTO 0);
input_data       <= input_data_array(i)(31 DOWNTO 0);
WAIT FOR clk_period;
-- Stop key/data input
key_data_in_flag <= '0';
input_key        <= (OTHERS => '0');
input_data       <= (OTHERS => '0');
-- Wait until encryption complete
WAIT UNTIL output_data_flag = '1';
```

Figure 7 - XTEA Testbench Data Input

A single STD_LOGIC flag called encryption_flag is used to set the proposed solution into either encryption mode or decryption mode, seen here is an encryption data input cycle, meaning a '1' in the encryption_flag. This should set the FPGA solution to encoding mode. In addition to this encryption_flag, there is also a key_data_in_flag. This is set high when data is inputted and then set low when the data has been sent to the solution.

Whilst the key_data_in_flag is set high; data is sent through two 32-bit logic vectors. Those being the input_key and the input_data. The initial key and data are stored in two 128-bit logic vectors set as xtea_keys and input_data_array. These are postfixed with an (i) which allows the testbench to iterate over this process multiple times, for now, this can be ignored as we are only interested in the interface, as this will remain the same over iterations. For each clock cycle that the key_data_in_flag is set high, the input_data and input_key 32-bit logic vectors are updated with a 32-bit partition of the 128-bit data and key. This means the top level of our solution must take in these 4 32-bit words and concatenate them into a 128-bit data and key pair for the 128-bit XTEA algorithm to use. Another thing to note in this image is the final line, the testbench will wait until it receives an input called output_data_flag, meaning this logic must be set high once the algorithm is complete. Looking into the next step of the testbench gives the data retrieval:

```
-- Wait until encryption complete
WAIT UNTIL output_data_flag = '1';
-- Read ciphertext output on falling edge
WAIT UNTIL FALLING_EDGE(clk);
encrypted_data(127 DOWNTO 96) <= output_data;
WAIT FOR clk_period;
encrypted_data(95 DOWNTO 64)  <= output_data;
WAIT FOR clk_period;
encrypted_data(63 DOWNTO 32)  <= output_data;
WAIT FOR clk_period;
encrypted_data(31 DOWNTO 0)   <= output_data;
WAIT FOR clk_period;
-- Set mode to decryption
encryption_flag  <= '0';
```

Figure 8 - XTEA Testbench Output Retrieval

Leading on from figure 7, figure 8 shows how the testbench retrieves data from a potential solution. After retrieving the output_data_flag input from the top level of the solution, it reads one word of data per clock cycle from the output_data logic vector, this means that the 128-bit output must be split back into 4 32-bit words whilst outputting. Following on from this the process is repeated with the encryption_flag set low. The encryption process uses identical signals as the decryption phase, with identical clock period breaks between assignments.

B. Structure

After consulting the testbench and picking apart the XTEA algorithm, it is possible to construct a proposed structure for the entire FPGA implementation. The proposed structure is as follows:

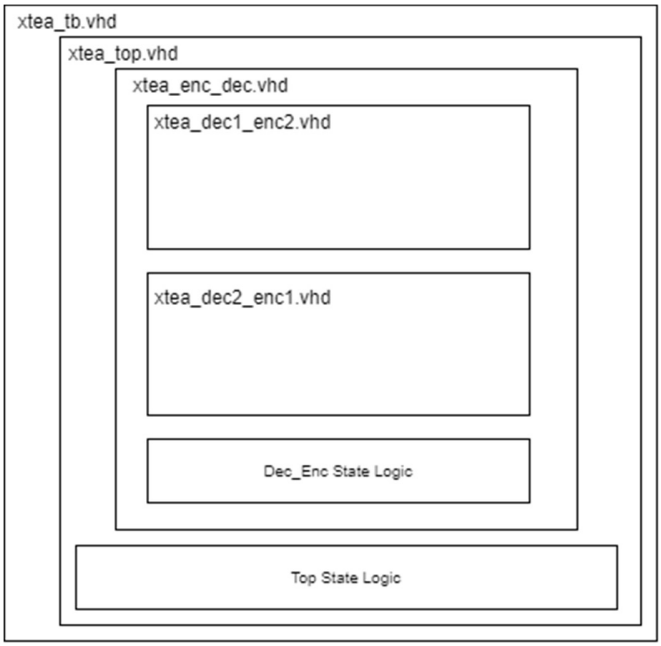

Figure 9 – XTEA FPGA Instantiation Proposed Structure Simplified

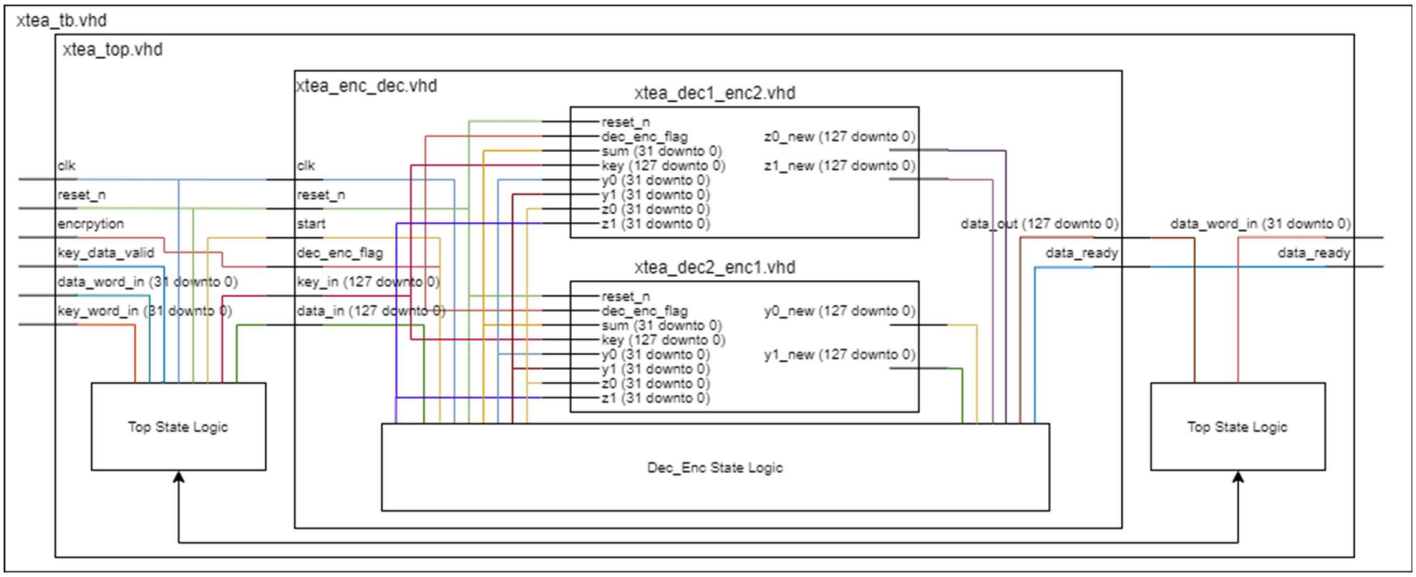

Figure 10 - XTEA FPGA Implementation Block Diagram

Described in figures 9 and 10 is the structure for the proposed XTEA FPGA implementation, it shows a top module with identical ports to the testbench provided, this is vital so that the testbench can instantiate the top level and in turn, the top can instantiate the rest of the components within.

This top-level module is comprised of a component called xtea_enc_dec. This component in turn instantiates two smaller components, which handle each feistel step of the XTEA algorithm. Combining these components with state-based logic, allows the top level to accept data from the testbench, trigger the xtea_enc_dec component to perform the XTEA algorithm, then retrieve its 128-bit result and break it back down into 32-bit words to output back to the testbench.

This structure allows for a logically compact encoder and decoder, meaning that instead of instantiating 4 feistel step modules, 2 for encoding and 2 for decoding, the solution can re-use the same feistel step modules for both encoding and decoding. Of course, if duplex (encoding and decoding concurrently) was being used an additional xtea_enc_dec would need to be instantiated. For this project brief, however, only one operation needs to be performed at any one time.

C. *Encoding & Decoding Feistel Step Modules*

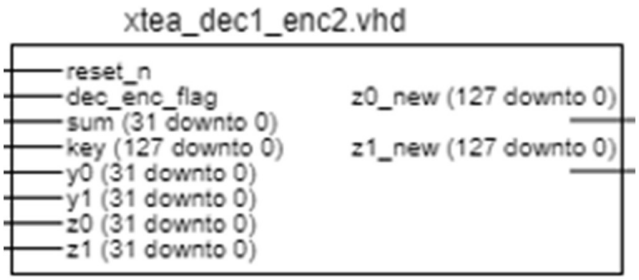

Figure 11- XTEA decoding/encoding feistel step Block Diagram

Starting with the deepest level of instantiation, the feistel step encoding and decoding modules (xtea_dec1_enc2.vhd, xtea_dec2_enc1.vhd) take in the current values of the internal 32-bit signals, these being y0, y1, z0, and z1. In addition to these signals, it also requires the 128-bit key, the current value of sum (32-bit) and due to the nature of these components being able to handle both encoding and decoding the components require a flag to set the mode of the feistel step to encoding or decoding. These modules only output a new value for y0, y1, z0, and z1 with the first feistel step for encoding outputting new y0 and y1 values, and the second encoding step outputting new z0 and z1 values. When looking at the input and output ports of each this is the only difference between the two, however internally each feistel step uses different variables to compute the output, with the first encoding step using the z values to compute the increment and adding this increment to the current y values. This is reversed for the second feistel step.

Each Module can be broken down into two processes, the output_decode process and the key_decode process. As noted in the C implementation section, figures 5 and 2 show the algorithm incrementing/decrementing a variable. For this component, the increment step has been split into two. One process works out the increment, and the other process increments the input value by the computed increment value. The key_decode process handles the computation of the increment value, the first step of this is selecting which subkey to use for the calculation. The FPGA implementation of this is seen below:

```
-- As per the XTEA algorithm, selecting the 1st and 2
-- word of the key to use for this iteration of the a
case (sum(1 DOWNTO 0)) is
   when "00" =>
      -- Assign the addition of sum and the selected
      -- keycalc = sum + *k[sum & 3];
      key_calc := sum + unsigned(key(127 DOWNTO 96));

   when "01" =>
      key_calc := sum + unsigned(key(95 DOWNTO 64));

   when "10" =>
      key_calc := sum + unsigned(key(63 DOWNTO 32));

   when "11" =>
      key_calc := sum + unsigned(key(31 DOWNTO 0));
   when others =>
end case;
```

Figure 12- XTEA Feistel round Subkey Calculation

Performing a case statement on the 2 least significant bits and using this to select which subkey to add to sum matches the algorithm described in figures 2 & 5.

This subkey calculation is then used in the following line to compute the increment:

```
-- Perform the XTEA algorithm to work out the amount to icrement z1 and z0 by ba
-- y1  := ((z1 << 4 ^ z1 >> 5) + z1) ^ keycalc;
y1_increment <= ((shift_left(z1u,4) XOR shift_right(z1u,5)) + z1u) XOR key_calc;
y0_increment <= ((shift_left(z0u,4) XOR shift_right(z0u,5)) + z0u) XOR key_calc;
```

Figure 13 - XTEA Feistel Step Increment Logic

Comparing this line to figures 2 & 5 the logic implemented is identical.

Once this increment is computed, the increment now must be used to increment or decrement the current input values provided to the feistel step decoder/encoder. This is performed in the output_decode process:

```
-- If reset is triggered, reset the outputs to 0.
if reset_n = '0' then
    y0_new <= (others => '0');
    y1_new <= (others => '0');
-- If reset is not triggered, then check the mode (
-- If in decoding, then decrement the z0 and z1 va
elsif dec_enc_flag = '1' then
    y1_new <= STD_LOGIC_VECTOR(y1u + y1_increment);
    y0_new <= STD_LOGIC_VECTOR(y0u + y0_increment);
else
    y1_new <= STD_LOGIC_VECTOR(y1u - y1_increment);
    y0_new <= STD_LOGIC_VECTOR(y0u - y0_increment);
end if;
```

Figure 14 - XTEA Feistel Step Output Decode

This process is simple, taking the input of reset and the decoding/encoding flag, and based on the combination of those inputs either resets the y0 values to 0 for a reset or in the case of no reset it applies an increment or decrement to the values based on the mode selected.

It is also worth noting that no clock is used for these components, allowing them to function between clock cycles, always outputting the correct value based on the inputs provided. The clock synchronization is provided by the higher-level components.

D. *Encoding & Decoding Module*

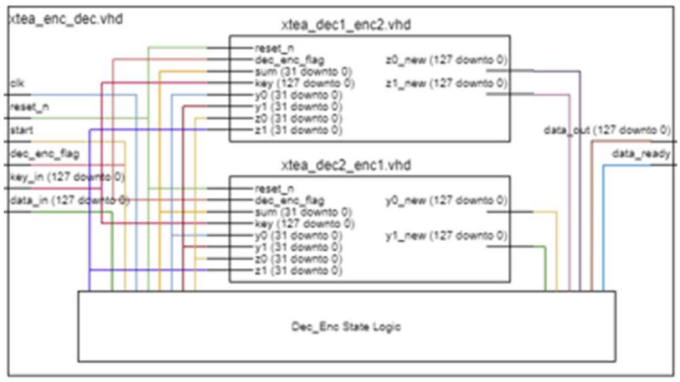

Figure 16 - XTEA Encoding and Decoding Module Block Diagram

One level higher than the feistel steps is the component that handles the XTEA algorithm, as shown in the image it instantiates one of each of the feistel steps in the XTEA algorithm. In addition to this, it also implements some state-based logic. The states required of the system are an idle state (StartState), a state where the module is waiting for input from components higher up. This state also serves as a reset state, setting all outputs and internal signals to 0 when in this state. Following on from this state there are two states for each step of the algorithm, these states are known as ProcessStateD1E2 (decoding step 1, encoding step 2) and ProcessStateD2E1 (decoding step 2, encoding step 1). These states simply increment, or decrement sum based on the mode of the system, in addition to applying the new values for y0, y1, z0, and z1 based on which state the system is currently in.

This module can be split into 3 processes:

A next state decode; where the next state of the system is computed, controlling the flow of the module. This process works by taking a case statement based on the current state of the system, and with a process variable ns_iterator alongside the current decryption/encryption mode allows the system to move between states.

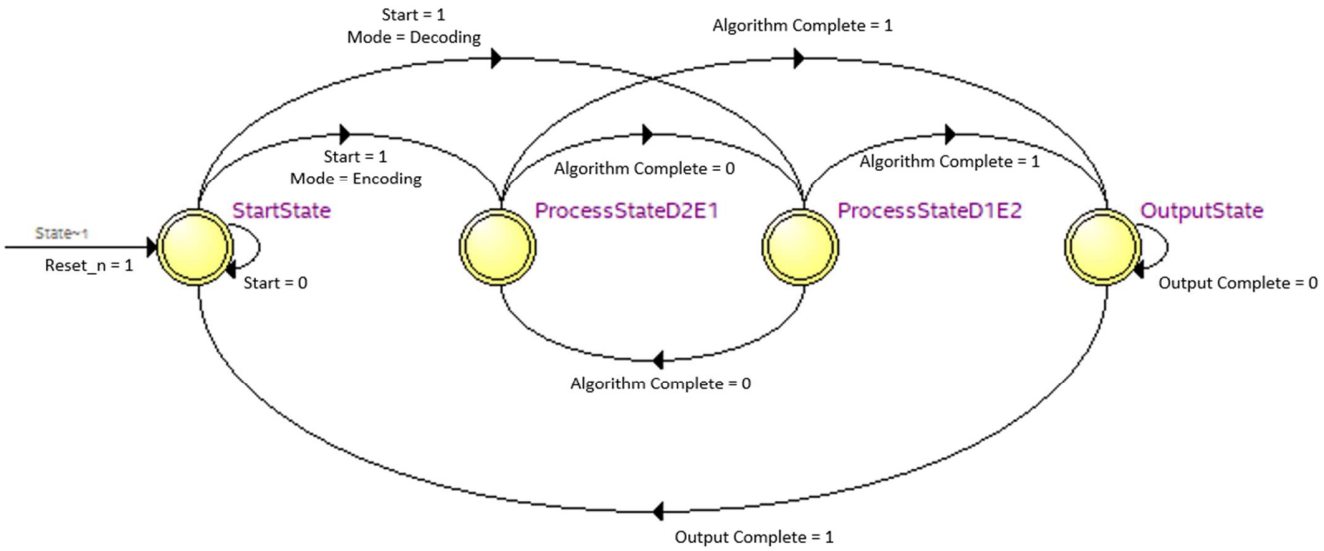

Figure 15 - XTEA Decoding / Encoding Module State Machine Logic

When in the starting state if an internal signal from a higher-level component is triggered, the system will move states into either decoding step1, or encoding step1 depending on the mode of the system. This state also resets the iterator to 0.

```
case (State) is
   when StartState =>
      -- When in the Reset/Idle state, reset the
      ns_iterator := to_unsigned(0, 8);
      -- If the internal start signal from the t
      if start = '1' then
         -- If in decoding mode then set the next
         if dec_enc_flag = '0' then
            NextState <= ProcessStateD1E2;
         -- If not in decoding mode (encoding mo
         else NextState <= ProcessStateD2E1;
         end if;
      -- If the start signal is not high, then r
      else Nextstate <= StartState;
      end if;
```

Figure 17 - XTEA Encoding / Decoding Module Start State Decoding Logic

With some logic based on the iterator, the code can compute if the algorithm has done its total amount of cycles and if so, the state machine can move onto the output state. If this is not the case, however, the state machine will flip between these two states until this logic returns a high value.

```
   when ProcessStateD1E2 =>
      data_ready <= '0';
      data_out <= (others => '0');
      -- When in the Decode 1st Step, Enc
      s_z0 <= s_z0_new;
      s_z1 <= s_z1_new;
   when ProcessStateD2E1 =>
      data_ready <= '0';
      data_out <= (others => '0');
      -- When in the Decode 2nd Step, Enc
      s_y1 <= s_y1_new;
      s_y0 <= s_y0_new;
```

Figure 18 - XTEA Encoding / Decoding Module Process State Outputs

Even in the output state, the iterator is used to ensure that enough time is spent in this state for the higher-level components to pick 4 32-bit words out of the 128-bit output provided. This then resets back to the start state, waiting for the next algorithm to be performed.

```
   when OutputState =>
   -- When in the output state reamin incrementi
      ns_iterator := ns_iterator + 1;
      -- Hold the state in output state for long
      -- 8 Clock cylces (9 Greater than the prev
      -- Making this number exactly correct has
      -- would make any differnence to the outpu
      if ns_iterator >= (total_cycles*4)+8 then
          NextState <= StartState;
      else NextState <= OutputState;
      end if;
```

Figure 19 - XTEA Encoding / Decoding Module Next State Decode Process Output State Logic

The second process is the sum block process, this handles the increment and decrementing of the sum signal by the constant delta as well as initializing values. This summing process only happens on the rising edge of the clock, meaning that the logic happening on the falling edge of the clock can always have an updated sum value.

```
case (State) is
   when StartState =>
      -- If the system is in enco
      if dec_enc_flag = '1' then
         s_sum <= sum_enc_init;
      else s_sum <= sum_dec_init;
      end if;

   when ProcessStateD1E2 =>
      -- If the system is in deco
      if dec_enc_flag = '0' then
         s_sum <= s_sum - delta;
      end if;

   when ProcessStateD2E1 =>
      -- If the system is in enco
      if dec_enc_flag = '1' then
         s_sum <= s_sum + delta;
      end if;
```

Figure 20 - XTEA Encoding / Decoding Module Sum Process

And finally, the output decode process handles the 32-bit words updating to their new values, in addition to setting them to the input provided at the start of the algorithm, and their output at the end. This process only happens on the falling edge of the clock, allowing the sum process to happen between output decodes. This is because the sum must happen between the two states, not concurrently.

```
case (State) is
   when StartState =>
      -- When in idle, reset the data rea
      data_ready <= '0';
      data_out <= (others => '0');
      -- When in idle, set the processing
      s_y0 <= data_in(127 DOWNTO 96);
      s_z0 <= data_in(95 DOWNTO 64);
      s_y1 <= data_in(63 DOWNTO 32);
      s_z1 <= data_in(31 DOWNTO 0);
   when ProcessStateD1E2 =>
      data_ready <= '0';
      data_out <= (others => '0');
      -- When in the Decode 1st Step, Enc
      s_z0 <= s_z0_new;
      s_z1 <= s_z1_new;
   when ProcessStateD2E1 =>
      data_ready <= '0';
      data_out <= (others => '0');
      -- When in the Decode 2nd Step, Enc
      s_y1 <= s_y1_new;
      s_y0 <= s_y0_new;
   when OutputState =>
      -- Set the internal data ready signa
      -- state machine on the higher leve
      data_ready <= '1';

      -- When in the output step, set the
      data_out <= STD_LOGIC_VECTOR(s_y0)
               & STD_LOGIC_VECTOR(s_z0)
               & STD_LOGIC_VECTOR(s_y1)
               & STD_LOGIC_VECTOR(s_z1);
```

Figure 21 - XTEA Encoding / Decoding Module Output Decode Process State Logic

Instantiating the feistel steps helps to keep this component simple, hiding the complex algorithm behind a component allows for a single assignment rather than a large string of logic within the higher-level components. These 32-bit values are then concatenated back to the 128-bit data_out signal which is in turn read by the top-level component.

E. Top-Level Module

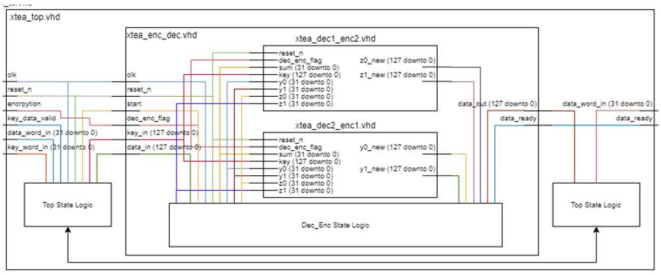

Figure 22 - XTEA Top Block Diagram

Now that the lower-level instantiated components have been described, it is possible to understand the top-level component. The purpose of the top-level component is to interface between the 128-bit XTEA algorithm implementation in xtea_enc_dec and the 32-bit inputs provided in the testbench xtea_tb.
Due to the wide array of tasks that the top-level must compute there is a requirement for state-based logic, akin to the state system utilised in the enc_dec instantiated component. The states required are:

An idle/reset state (idle), this state is responsible for resetting the outputs of the device to 0, whilst monitoring the inputs from the testbench. As previously mentioned in the testbench section, in figure 7 the testbench provides a signal of logic that will turn high when data is being inputted. This signal is what starts the device.

```
-- Idle / Reset State
when idle =>
   if key_data_valid = '1' then -- If data ir
      -- Apply the first 32 bits in the reset
      s_key_in(31 DOWNTO 0) <= key_word_in;
      s_data_in(31 DOWNTO 0) <= data_word_in;
   end if;

when keydatain1 =>
   -- Insert the next word into the 2nd 32 bi
   s_key_in(63 DOWNTO 32) <= key_word_in;
   s_data_in(63 DOWNTO 32) <= data_word_in;

when keydatain2 =>
   -- Insert the next word into the 3rd 32 bi
   s_key_in(95 DOWNTO 64) <= key_word_in;
   s_data_in(95 DOWNTO 64) <= data_word_in;

when keydatain3 =>
   -- Insert the next word into the 4th 32 bi
   s_key_in(127 DOWNTO 96) <= key_word_in;
   s_data_in(127 DOWNTO 96) <= data_word_in;

when encdec =>

   s_start <= '1'; -- Trigger the encoding /
```

Figure 24 - Top Level State Machine Idle, KeyDataIn & EncDec States

Once the key_data_valid signal has been received the system moves to the key data in states

```
when idle =>
   if key_data_valid = '1' then
      NextState <= keydatain1;
   else NextState <= idle;
   end if;
```

Figure 25 - XTEA Top Level Idle State Logic

These are sequential states with no logic controlling them, each state takes in the current input in key_word_in and data_word_in and applies it to the matching 32-bit part of the 128-bit input signal. The first assignment is performed in the idle state, this is due to the test benches data input routine, as seen in figure 7, the key_data_in_flag and first 32 bits of data and key are set at the same time, leaving no time to switch between states. Once these sequential states have been executed, the state machine moves to the encoding state, which uses the internal start signal to set the enc_dec module to start encoding or decrypting depending on the selected mode. The state machine sits within this state until another internal signal from the instantiated enc_dec module is returned. This signal is called s_data_ready.

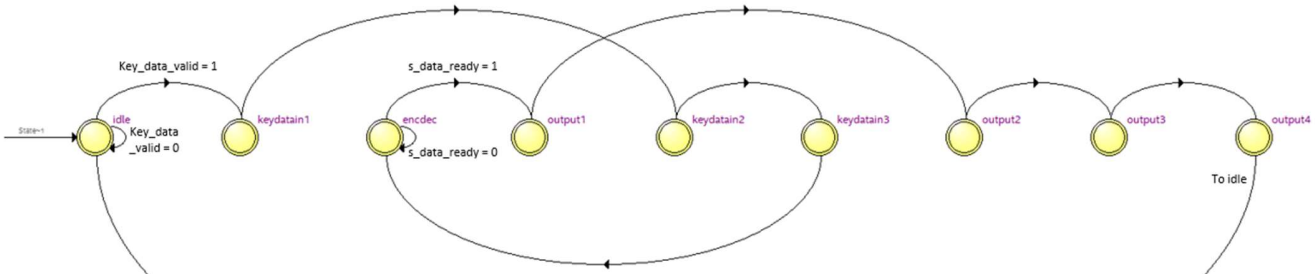

Figure 23 - Top Level XTEA Interface State Machine Logic

```
when encdec =>

    if s_data_ready = '1' then
        NextState <= output1;
    else NextState <= encdec;
    end if;
```

Figure 26 - XTEA Top Level EncDec Next State Logic

When this signal goes high the next state is set to the output states, which similarly to the input states have no controlling logic. And sequentially move through the states, outputting a 32-bit part of the 128-bit output from the internal enc_dec module.

```
when output1 =>
    --Set the first word out as the first 32 bi
    data_word_out <= s_data_out(31 DOWNTO 0);

when output2 =>
    --Set the second word out as the second 32 b
    data_word_out <= s_data_out(63 DOWNTO 32);

when output3 =>
    --Set the third word out as the third 32 bi
    data_word_out <= s_data_out(95 DOWNTO 64);

when output4 =>
    --Set the fourth word out as the last 32 bi
    data_word_out <= s_data_out(127 DOWNTO 96);
```

Figure 27 - XTEA Top output states Logic

Once the data has been outputted, the state reverts to the idle state, resetting the system and waiting on the next set of data entry.

F. Validation

To validate the FPGA implementation, it is possible to simulate the xtea_top file with the provided testbench. Compiling these files within ModelSim provides a waveform, showing the external and internal signals of the solution at all points of the program. In addition to this form of validation, also available to compare to is the step-by-step implementation in C, which as shown in figure 6 outputs the expected internal variables for each step of the algorithm. Also available within the testbench is a set of messages, this will compare the expected result in the testbench with the actual result and print a message to the console (transcript) that reads key/data pair passed/failed.

```
    -- Compare decrypted data with original plaintext
    IF decrypted_data = input_data_array(i) THEN
        REPORT "NOTE: Key/data pair " & INTEGER'IMAGE(i+1) & " passed" SEVERITY NOTE;
    ELSE
        REPORT "ERROR: Key/data pair " & INTEGER'IMAGE(i+1) & " failed" SEVERITY ERROR;
        fail_flag    := '1';
        fail_counter := fail_counter + 1;
    END IF;
END LOOP;
-- Print final results
IF fail_flag = '0' THEN
    REPORT "NOTE: All tests passed" SEVERITY NOTE;
ELSE
    REPORT "ERROR: " & INTEGER'IMAGE(fail_counter) & " tests failed" SEVERITY ERROR;
END IF;
```

Figure 28 – XTEA Testbench Notes & Error Messages

Simulating within ModelSim provides the following console output:

```
VSIM 2> run
# ** Note: NOTE: Key/data pair 1 passed
#    Time: 1535 ns  Iteration: 0  Instance: /xtea_tb
# ** Note: NOTE: Key/data pair 2 passed
#    Time: 3035 ns  Iteration: 0  Instance: /xtea_tb
# ** Note: NOTE: Key/data pair 3 passed
#    Time: 4535 ns  Iteration: 0  Instance: /xtea_tb
# ** Note: NOTE: All tests passed
#    Time: 4535 ns  Iteration: 0  Instance: /xtea_tb
```

Figure 29 - Console Response after running provided XTEA Testbench.

These messages confirm that the algorithm is functioning as expected, and the data input and output stages are functioning correctly also. Also shown to validate the solution is the waveform provided by simulating ModelSim. Only a select number of internal signals will be shown for simplicities sake.

Figures 28 to 31 show the waveform and a breakdown of each of the sections of the waveform. Figure 28 shows a full run-through of 3 iterations of decoding and encoding, whilst it is hard to see details of this image, it gives a reference for the other figures.

Figure 29 shows a single encoding sequence. This is further broken down into 2 smaller graphs, that highlight the data in and the data out sequences, these being figures 30 and 31, respectively.

Figure 30 shows the system navigating through states at the correct interval, slowly building up the data in and key in 128-bit signals to the point they stay at one the program has reached keydatain3. This is followed by the s_start signal from the encdec state which triggers the internal state machine in enc_dec.vhd to move to ProcessStateD2E1, considering this is an encoding waveform it is working as intended.

Figure 31 then shows the tail end of the algorithm, after iterating the next_state_iterator (bottom signal) to 128 the system sets the data ready flag to high and exits out of the process states, hanging in the output state while the top-level outputs the 4 32-bit words from the 128-bit data out signal. These four waveforms in addition to the console output retrieved in figure 27 are enough to validate the proposed solution.

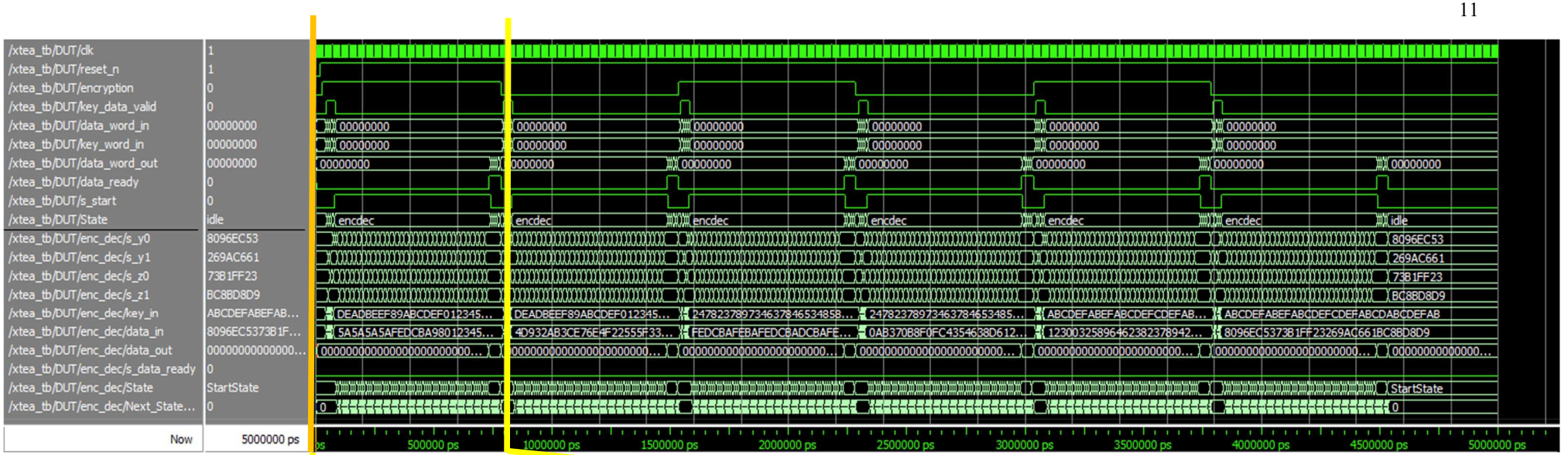

Figure 30 - Full 3 Iterations of XTEA Algorithm Simulation

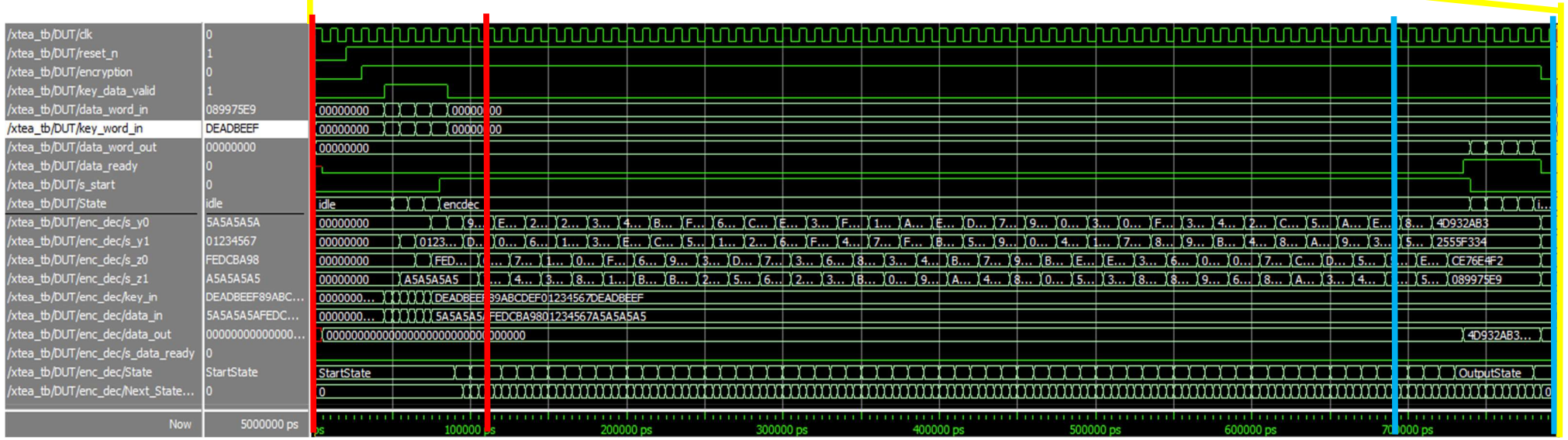

Figure 31 - Encoding of One Iteration of the XTEA Algorithm Simulation

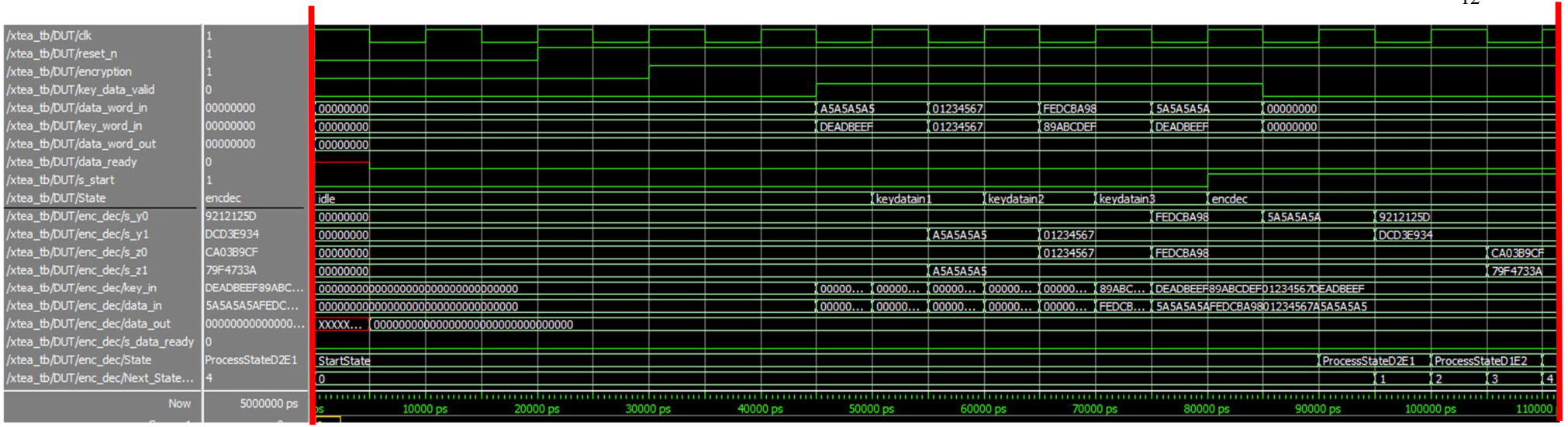

Figure 32 - Data Input Sequence for XTEA Algorithm Simulation

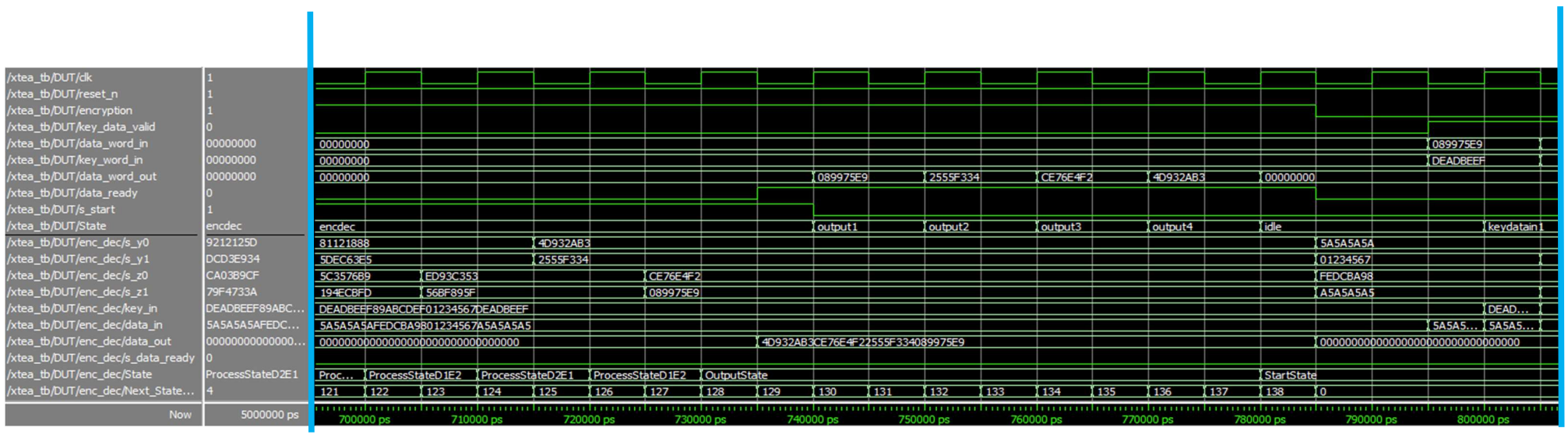

Figure 33 - Data Output Sequence for XTEA Algorithm Simulation

V. Performance

Given the style of design chosen and the efficiency of reusing each feistel step of the algorithm, I would expect the power and area used to be low relative to a design that does not implement these design choices. This can be monitored by running power and area tests within Quartus.

A. Power

After running the power analyser tool, the output received from testing the design is as follows:

Total Thermal Power Dissipation	424.44 mW
Core Dynamic Thermal Power Dissipation	0.00 mW
Core Static Thermal Power Dissipation	411.25 mW
I/O Thermal Power Dissipation	13.18 mW

Figure 34 - Power Analyser Tool Results

These results align with what was expected, low power usage. If more time were available, it would be good to implement a solution without the reuse of the feistel steps, to compare this result against it however because of a lack of time this is not possible within this report.

B. Area

When compiling a solution in Quartus after providing the target board, Quartus provides details on the number of registers and pins alongside the total amount of ALMs (adaptive logic modules) The report on the design for the area and logic utilization is as follows:

Logic utilization (in ALMs)	477 / 32,070 (1 %)
Total registers	304
Total pins	101 / 457 (22 %)
Total virtual pins	0
Total block memory bits	0 / 4,065,280 (0 %)
Total RAM Blocks	0 / 397 (0 %)
Total DSP Blocks	0 / 87 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 6 (0 %)
Total DLLs	0 / 4 (0 %)

Figure 35 - Logic Utilisation Report

As seen from the report, the design uses a minuscule amount of the logic blocks available on the board, only 447 out of a possible 32,070 making up only 1% of the potential ALMs. 304 Registers were also used, in addition to 101 pins. This means there is potential to instantiate multiple decoders and encoders onto the board at any one time, performing multiple instances of the XTEA algorithm at once.

VI. Conclusion

Whilst not all initial objectives have been met, most of the objectives have been met. An FPGA implementation of the XTEA algorithm has been created, validated, and tested for power and area. Alongside this, a C implementation has also been created with a script to break down the algorithm into its respective parts, alongside the work in progress of a script that takes in user input and breaks it down into 128-bit chunks for the algorithm to encrypt.

Unfortunately, due to time constraints, it has not been possible to implement the FPGA solution onto the FPGA, and whilst Linux has been put on the A9 running alongside the FPGA, no communication between the two was ever possible, hence the full design has not been completed.

A. Limitations & Future Improvements

If I were to continue working on this project, I would focus on completing the work on the HPS-FPGA bridge using the platform designer in Quartus (formerly known as Qsys) to get the working solution implemented on the FPGA rather than in simulation.

Following on from this some other improvements that would help further the solution and my understanding would be:

Pipelining – Reading in the inputs before the system is ready for them, allowing the algorithm to continually run without breaks waiting for data input and output.

Duplex – Running both encryption and decryption simultaneously on separate instantiations of the enc_dec module. This would allow the system to run twice as fast in the theoretical situation where the hardware needs to both encrypt and decrypt at the same time.

Multiple Instantiations of the enc_dec module for faster decryption of a larger data set, this would allow multiple 128-bit chunks to be processed at any one time, given the area analysis the potential to instantiate a large number of encoding and decoding modules is there.

VII. REFERENCES

[1]"XTEA (eXtended TEA)", *Asecuritysite.com*, 2021. [Online]. Available: https://asecuritysite.com/encryption/xtea. [Accessed: 23- May- 2021].

[2]"Altera Cyclone® FPGAs", *Intel.com*, 2021. [Online]. Available: https://www.intel.com/content/www/us/en/programmable/b/cyclone-v.html. [Accessed: 23- May- 2021].

[3]"FPGA Design Software - Intel® Quartus® Prime", *Intel*, 2021. [Online]. Available: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html. [Accessed: 23- May- 2021].

[4]"Intel® FPGA Simulation - ModelSim*-Intel® FPGA", *Intel*, 2021. [Online]. Available: https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/model-sim.html. [Accessed: 23- May- 2021].

[5]V. Code, "Visual Studio Code - Code Editing. Redefined", *Code.visualstudio.com*, 2021. [Online]. Available: https://code.visualstudio.com/. [Accessed: 23- May- 2021].

[6]"Non-specialized hardware comparison - Bitcoin Wiki", *En.bitcoin.it*, 2021. [Online]. Available: https://en.bitcoin.it/wiki/Non-specialized_hardware_comparison. [Accessed: 23- May- 2021].

[7]"The Future of Jobs Report 2020", *World Economic Forum*, 2021. [Online]. Available: https://www.weforum.org/reports/the-future-of-jobs-report-2020. [Accessed: 23- May- 2021].

[8]"XTEA - Wikipedia", *En.wikipedia.org*, 2021. [Online]. Available: https://en.wikipedia.org/wiki/XTEA. [Accessed: 23- May- 2021].

[9]*Ftp.intel.com*, 2021. [Online]. Available: https://ftp.intel.com/Public/Pub/fpgaup/pub/Intel_Material/18.1/Computer_Systems/DE1-SoC/DE1-SoC_Computer_ARM.pdf. [Accessed: 23- May- 2021].

[10]*People.ece.cornell.edu*, 2021. [Online]. Available: https://people.ece.cornell.edu/land/courses/ece5760/DE1_SOC/SoC-FPGA%20Design%20Guide_EPFL.pdf. [Accessed: 23- May- 2021].

[11]*Oa.upm.es*, 2021. [Online]. Available: http://oa.upm.es/45352/1/DE1-SoC_Embedded_Linux_Systems_1_9.pdf. [Accessed: 23- May- 2021].