Tea Algorithm for FPGA Hardware Acceleration and Reconfigurable Encryption

Hmayak Apetyan
hapetyan@cpp.edu

Jacob London
jelondon@cpp.edu

Ziyan Lin
ziyanlin@cpp.edu

Hyung Jin Kim
hyungjinkim@cpp.edu

Yongyuan Zhang
yongyuanz@cpp.edu

Mohamed El-Hadedy Aly
mealy@cpp.edu

Electrical and Computer Engineering (ECE) Department
Cal Poly Pomona
Pomona, CA

***Abstract*—The Tea algorithm is a decent example of a simple algorithm which can be implemented on hardware for reconfigurability. By designing other hardware interfaces, it is possible to easily see the data which is inside the FPGA with designs that interface with UART, universal asynchronous receiver-transmitter, and VGA, video graphics array, IO, input-output. By assembling this design, it becomes easier to visualize and interact with the encryption algorithm, and the simple nature of the design allows for much more complex algorithms to be easily implemented while not needing to change the design of the entire hardware system.**

***Keywords*—block cipher; encryption; IO; LWC; RECO-ENC; tea; UART; VGA; VHDL;**

I. INTRODUCTION

Implementing the Tea, tiny encryption algorithm, on an FPGA, field-programmable gate array, allows for a simple reconfigurable encryption system implemented into a hardware design. This algorithm is simple enough to create a base design which can be easily modified to allow more complex designs to be implemented. Through the use of hardware design in VHDL, a wide variety of FPGAs can be targeted and designed for, while utilizing the simple base design implemented by this system.

II. DEVELOPMENT

The development of this project was broken into different parts and distributed to the work group so that each section could be tested and work by itself in a closed system. After each section was designed, implemented, and tested, a higher level module was designed to control each section together and interface with the FPGA IO.

III. ARCHITECTURE

The Tea was implemented through a series of states, where each state performs a binary operation on some data. These binary operations were derived from the public domain implementation of Tea, broken down into its steps, as seen in Algorithm 1 and Algorithm 2.

Algorithm 1: Pseudocode to encipher

```
algorithm tea-encipher is
  input: Number of rounds N,
         A 64-bit data V,
         A 128-bit key K
  output: A 64-bit enciphered data R
  let D be the key schedule constant
  D ← 2^32/Φ = 0x9E3779B9
  let S be the current schedule
  S ← 0
  let K_f be the cached key
  K_f(0) ← bit range 0 to 31 in K
```

```
K_f(1) ← bit range 32 to 63 in K
K_f(2) ← bit range 64 to 95 in K
K_f(3) ← bit range 96 to 127 in K
for each round (R) in N do
        S ← S + D
        V(0) ← V(0) + (sll{V(1), 4} + K_f(0)) xor
(V(1) + S) xor ((srl{V(1), 5}) + K_f(1))
        V(1) ← V(1) + (sll{V(0), 4} + K_f(2)) xor
(V(0) + S) xor ((srl{V(0), 5}) + K_f(3))
R(0) ← V(0),   R(1) ← V(1)
return R
```

Algorithm 2: Pseudocode to decipher

```
algorithm tea-decipher is
input: Number of rounds N,
        A 64-bit data V,
        A 128-bit key K
output: A 64-bit deciphered data R
let D be the key schedule constant
D ← 2^32/Φ = 0x9E3779B9
let S be the current schedule
S ← 0
let K_f be the cached key
K_f(0) ← bit range 0 to 31 in K
K_f(1) ← bit range 32 to 63 in K
K_f(2) ← bit range 64 to 95 in K
K_f(3) ← bit range 96 to 127 in K
for each round (R) in N do
        V(1) ← V(1) - (sll{V(0), 4} + K_f(2)) xor
(V(0) + S) xor ((srl{V(0), 5}) + K_f(3))
        V(0) ← V(0) - (sll{V(1), 4} + K_f(0)) xor
(V(1) + S) xor ((srl{V(1), 5}) + K_f(1))
        S ← S - D
R(0) ← V(0),  R(1) ← V(1)
return R
```

These algorithms were broken down where each state in the state machine did one of the binary operations in each of the for loops. This method successfully produced the algorithm on hardware.

A. Encipher

The encipher operation performs many binary operations on data in order to produce an enciphered result. These operations take up twenty-five states, and greatly simplify the steps required to produce Algorithm 1. State zero resets all registers to their default value, then the machine proceeds through states one through twenty-four. At state twenty-four, it is checked whether the correct number of rounds has been reached. If it has not been, go back to state one and repeat the process, otherwise the system goes into the idle state, or state twenty-five where it waits for a reset signal before beginning again.

B. Decipher

The decipher module of the VHDL code is very similar to the encipher module. This is due to decipher basically being the backward operation of the encipher. There just like the encipher the decipher also uses a finite state machine to do its operation. It has twenty five different states to do the operation. While the encipher operation adds every round decipher does the opposite and subtract every round. The encipher starts with the sum, then V(0), and finally ends with V(1). While the decipher starts with the V(1), then V(0), and finally ends with the sum to undo the encipher.

C. VGA

The display of VGA is similar to the display of different numbers on 7-segment display, which means that it needs a clock or a counter to assign values to display for each line of numbers (in this case, the numbers are enciphered code, key and deciphered code). So we will need at least 2 bits counter to display the lines on the monitor.

In order to display different values instead of a fixed value from the memory, we need a look-up table to search for each character with the input and output the selected area of the RAM which includes VGA code for all hex numbers with size 7 bits (columns) * 5 bits (rows) for each number. And the total size of the RAM will be 112 (columns) * 5 (rows). We use the input as the index to look for specific part of the RAM, and combine them together as a complete output for VGA. Since the output lines have different sizes (64 bits for the codes and 128 bits for the key), we made the VGA outputs a generic module which is able to deal with all sizes of inputs.

D. UART

The UART communication circuitry utilizes a keyboard as a method of producing inputs (data and key) for the encryption algorithm; a Mealy finite state machine (FSM) controls the individual UART receiver, which was designed and implemented using VHDL within the Vivado IDE. The Mealy FSM is required for the FPGA to be capable of receiving multiple bytes, one at a time (since UART is a form of serial communication); this is accomplished through the use of the UART receiver's done flag, which is raised/set when the receiver has successfully received a byte of data (triggered by the stop bit in UART circuitry/communication). Once the receiver raises its done flag, the FSM takes and stores the data from the receiver, before it begins to receive another byte. It is important to note that the stand-alone circuitry of the UART receiver is only capable of receiving/storing a single byte. Therefore, the FSMs are used to control and coordinate the FPGA in a manner that not only allows it to receive multiple bytes from a computer's serial port, but also process the data it receives.

The FSM also provides the FPGA with the ability to pick and choose between the data it receives, so that only hex characters (0 through 9, "A"/"a" through "F"/"f") are passed on as inputs to the algorithm. A separate module, which is required in between receiving the data and passing it to the algorithm, converts the ASCII character (8 bit hex), received from the keyboard, to its appropriate 4 bit hex representation (0 through F, where F = 15); in the case that an ASCII character is not represented in 4 bit hex, the FPGA simply ignores this byte of data and waits for the next one. The conversion from 8 bit ASCII to 4 bit hex occurs through a series of if/else-if statements, where each one accounts for one of the 16 possible hex characters.

Once the FPGA has collected the appropriate amount of bytes for the data input (based on a generic value initialized in the VHDL code), it moves into the next state and begins collecting the inputs for the key. The inputs for the key are acquired in the same manner as previously described. Finally, once the appropriate amount of bytes for the key has been received, the FPGA returns to its initial state in which it is waiting to receive ASCII characters from the keyboard and determine if they are valid for the data input to the algorithm (as previously described).

IV. EVALUATION

The resulting hardware implementation of Tea utilized a state machine which processed different parts of the calculation of each round in order. Because of this state machine, the process for performing rounds of encryption or decryption became simpler than it would have otherwise been in a combination design for example. However, due to the sequential nature of the design, the implementation is not as fast as it could be.

The design was built for the Nexys 4 FPGA, with a clock speed of 100 MHz, and utilized a state machine with twenty-five states total, with twenty-three of those states dedicated to performing each round out of sixty-four rounds. In addition to this, along with the nature of state machines, the algorithm spends two clock cycles in each state. With this information, the execution time of an encryption or decryption operation can be calculated.

$$(1)\quad (100MHz)^{-1} * 2\ \frac{cycles}{state} * 25\ states * 64\ rounds = 32\ \frac{s}{operation}$$

With a combinational design, the speed of an encrypt or decrypt operation would be relative to the propagation delay of the circuit rather than the clock speed of the FPGA, however this design would need to be much more complex than the sequential design due to optimization issues which VHDL compilers perform such as K-map optimization. This kind of optimization is quite useful for designs, especially for FPGAs where area usage is important, however complex mathematical formulas become difficult to implement this way as they are often optimized away.

V. CONCLUSION

For the encipher and decipher modules, it was difficult to implement the tea algorithm at the start. The initial design was based on the assumption that VHDL can do the complicated math directly like

the algorithm definition shows, similar to the C code designed for the system. However, after running testbenches it was found that this was not the case and after many different implementations, rechecking the code, and rerunning testbenches, the finite state machine design was chosen to perform the math operation in the algorithm. With this design, the VHDL system was able to recreate the results that the C code was generating.

For the VGA module, the most difficult part is that we need to make the output a generic. Since the lines that we need to display on the monitor is not fixed, we need to use loop to generate individual RAM for each display. The logic of the loop is complex since the display of the RAM is in reverse order, so the design needs to reverse the order of the RAM character for each row, as well as the order for each character.

This project demonstrates the design for an encryption and decryption system with visual which can show the user the process of the chosen algorithm. These algorithms can be easily chosen and implemented into the current system simply by changing the finite state machine to perform the binary operations for each step in the selected algorithm. This design is successful in showing the use of encryption with a keyboard and VGA interface and the code shows that the system can be easily modified to work on a variety of algorithms.