ECE 341 Digital Systems Design Serpent Project Final Report

Group 5: Megan Witherow and Josh Collins

Introduction

Serpent is block cipher supporting a 128-bit block size and up to 256-bit key that was the second ranked finalist in the Advanced Encryption Standard contest in the late nineties. This ECE 341 final project seeks to implement the Serpent cipher in VHDL as behavioral, dataflow, and structural models. The requirements for the project specify that all models support both encryption and decryption, implement 128-bit keys, use the same entity, and use signals of std_logic/std_logic_vector or related types. The dataflow and structural models are required to use a minimum of one clock cycle per encryption phase, or 34 clock cycles. The system should be able to handle as many encryption schemes as possible. Two options are given for implementing the structural model: configurable logic blocks or extension of the gate library. Technology specifications include maximum of 64 memory units of 16,384 bits, acceptable clock rates of 25 MHz to 350 MHz, maximum of 80 input/output pins, and maximum number of logic gates/ flip flops of 600,000.

The remainder of the report is organized as follows: description of the gold model, description of the data flow model, description of the partial structural model, concurrent verification and comparison of all three models to the gold model, performance analysis, and summary.

Gold (Behavioral) Model

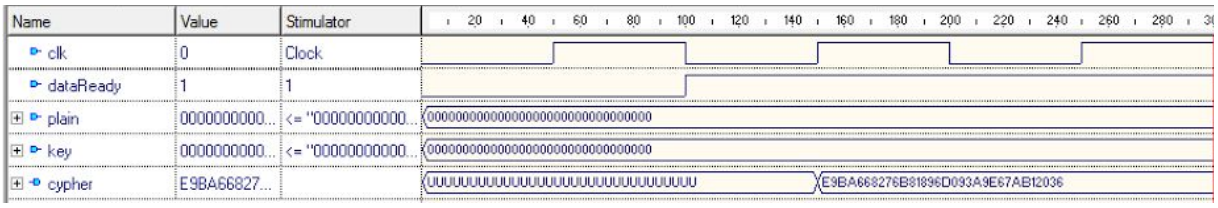
The Gold Model is designed to model the behavior of the Serpent encryption system while meeting all project requirements and constraints. The entity developed for use with the gold model is also used for the dataflow and partial structural implementations. This entity takes in portions of the 128-bit data (either plaintext or ciphertext) and 128-bit key sixteen bits at a time. Taking in the data and key sixteen bits at a time ensures that the entity is well below the 80 pin limitation, and allows for the 128-bit internal signals used to hold the full plaintext/ciphertext and key to be filled with information in the same number of cycles. The entity has three other std_logic inputs: *CLK*, *start*, and *encrypt*. *CLK* is the clock input. The *start* signal is used to indicate when the system should begin processing and should be given once both the data and key are ready. The system expects that the sixteen bit pieces of the data and key be given in consecutive clock cycles beginning with the cycle after *start* goes from '0' to '1'. The *encrypt* signal is used to specify whether the system should perform encryption (*encrypt = '1'*) or decryption (*encrypt = '0'*). The system outputs either the ciphertext (*encrypt = '1'*) or plaintext (*encrypt = '0'*) sixteen bits at a time from the *encrypted* output of the entity in eight consecutive clock cycles. A *done* output signal is activated once the encryption or decryption process is complete. The entity uses a total of 52 pins.

The behavioral model is implemented in bitslice mode. Bitslice mode is the more efficient implementation of Serpent designed to execute the cipher with 32-fold parallelism. The initial and final permutations used in the standard mode are not used in the bitslice implementation as these serve the purpose of converting to and from bitslice representation.

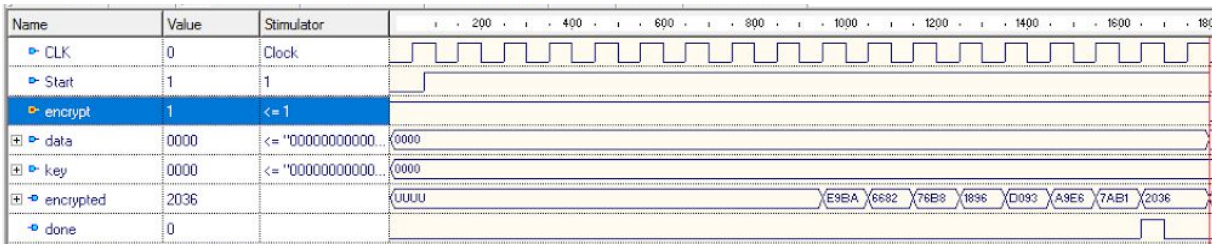
The behavioral model defines VHDL types for the subkeys, round keys, prekeys, and Sboxes. These are appropriate length arrays of unsigned vectors in the case of the subkeys, round keys, and prekeys, and in the case of the Sbox, of a defined Sbox_type that is an array itself. Variables are used

throughout the model and signals are kept to the minimum inputs and outputs. A case-when structure is implemented for reading in the data and key 16 bits at a time. The prekey generation has been modified to incorporate Dr. Belfore's more straightforward method of negative indexing. The subkeys are generated at the beginning of the model prior to testing on the *encrypt* input. The encryption is implemented per the paper's instructions on bitslice mode. Following the completion of the last cycle, the encrypted result is output 16 bits at a time on consecutive clock cycles. The decryption is implemented similarly, using the inverse of the Sboxes in the reverse order, inverse linear transformation, and reverse order of subkeys.

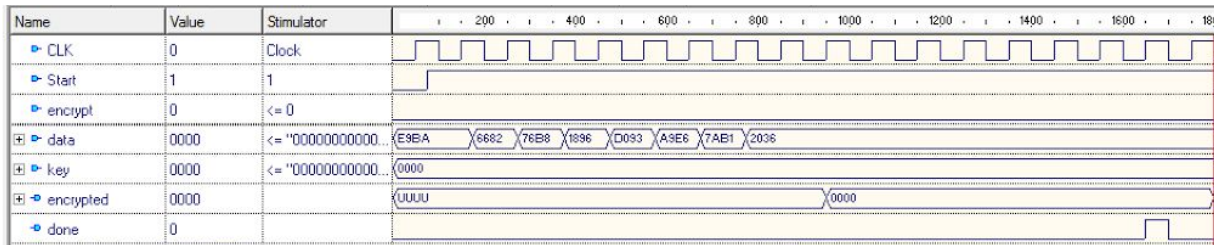
The gold model is verified individually for both encryption and decryption. Temporary signals are declared and used to output the contents of variables, such as the prekeys (*w*), round keys (*ik*), subkeys (*K*), and intermediate data (*Bi*), in order to verify intermediate steps. The results of the encryption are verified by comparison to Dr. Belfore's behavioral model (encrypt_r2.vhd), and the outputs are confirmed to match. The individual verification of the proper function of the behavioral model is shown in Fig. 1.



(a) Encryption using Dr. Belfore's model



(b) Encryption Waveform



(c) Decryption Waveform

Fig. 1. Verification of proper encryption and decryption; (a) shows the output of Dr. Belfore's behavioral model for reference; 128-bit plaintext and 128-bit key are given as all 0's, producing the ciphertext x"E9BA669276B81896D093A9E67AB12036" in (b) matching (a), and is shown to be reversible in (c)

Data-Flow

The data flow model was made by separating each task performed by behavioral model, designing a component to do the task, and linking them together. These tasks are: reading in the data and key, creation of the subkeys, linear transformation, s-box operation, and sending out the data. Each component (with the exception of the s-box) operates as a state machine, though the cycle/decycle have an asynchronous state change. The components begin their state machine operation when they receive a start signal and send out a done signal when finished. The data_in component begins first, reading the serpent data and key inputs and outputting the full key and data. The key is then sent to the subkey component which breaks the key down into the 33 subkeys. The data is sent to the first of the cycles or decycles, depending on whether the *encrypt* signal indicates encrypting or decrypting. Each of the cycles/decycles reads in its respective subkey, outputting to its respective s-box and reading the s-box output when needed. The 32 cycles feed into each other until the last one sends its data into the data_out component which reads the data out one piece at a time. Additionally the final cycles *done* output feeds into the serpent's *done* output. This allows the timing to be correct if one serpent implementation's input is connected to another's output.

To test the data flow model, a test bench is used that connects an encrypter to a decrypter and then gives both the proper inputs, waiting to start the decrypter until the encrypter was finished. The data flow model is also compared directly to the gold model as shown in the Verification section.

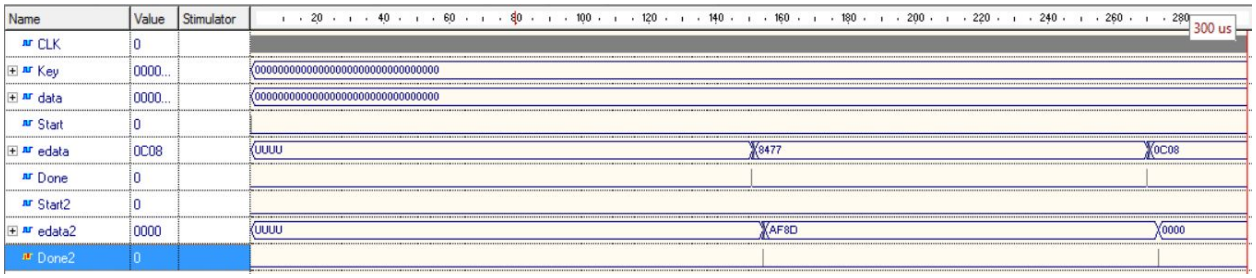


Fig. 2. Encryption and Decryption Data_Flow waveform

Structural

For the partial implementation of the structural model, we choose to extend the gate_lib library. Due to time constraints, we select one module, the cycle, to implement at the gate level. Four D flip-flops are used to store the states of the cycle's state machine and look-up tables are used for the next states. Four 32-bit registers are used to store X0, X1, X2, and X3. Components for 32-bit xor, 128-bit xor, 32-bit SLL, and 32-bit ROL are declared and defined in a separate library, renamed extended_gatelib. To make decisions based on the current state, 16 to 1 muxes are used.

The structural implementation of the cycle is swapped in for the cycle in a copy of the dataflow model to test its functionality in the overall system. This hybrid structural/dataflow model is compared directly to the gold model in a testbench. It is also shown that the function of the partial structural model matches that of the dataflow model, meaning that the cycle is working the same way in both implementations. This is shown in Fig. 3.

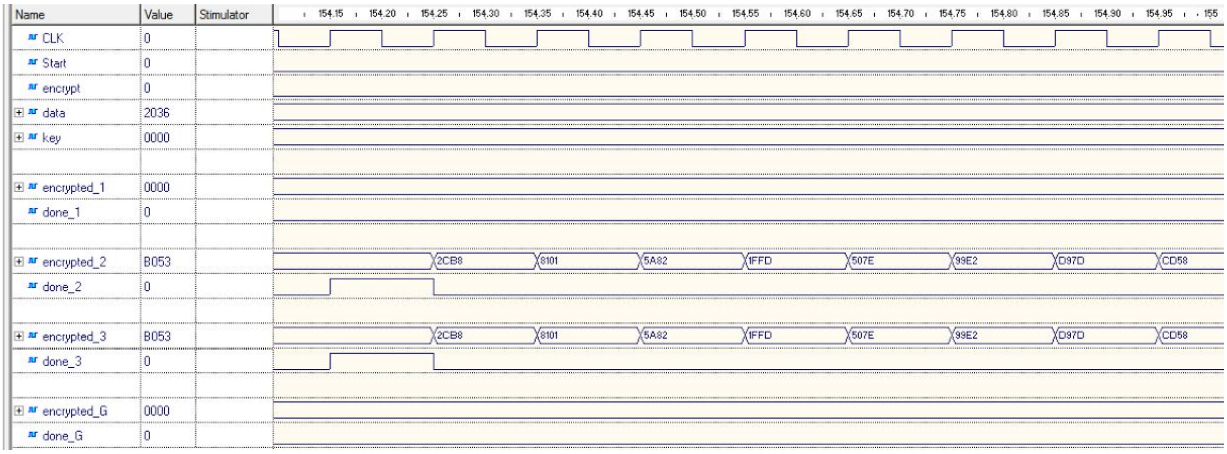


Fig 3. Zoomed-in view showing that the data flow and structural models agree

Verification

A testbench is designed to test all three models concurrently for comparison with the gold (behavioral) model as shown in Fig. 4. The dataflow and structural models are shown to match in functionality; however, although both of the models can reproduce the plaintext after running an encryption cycle then decrypting, the ciphertext in between does not match that of the gold model.

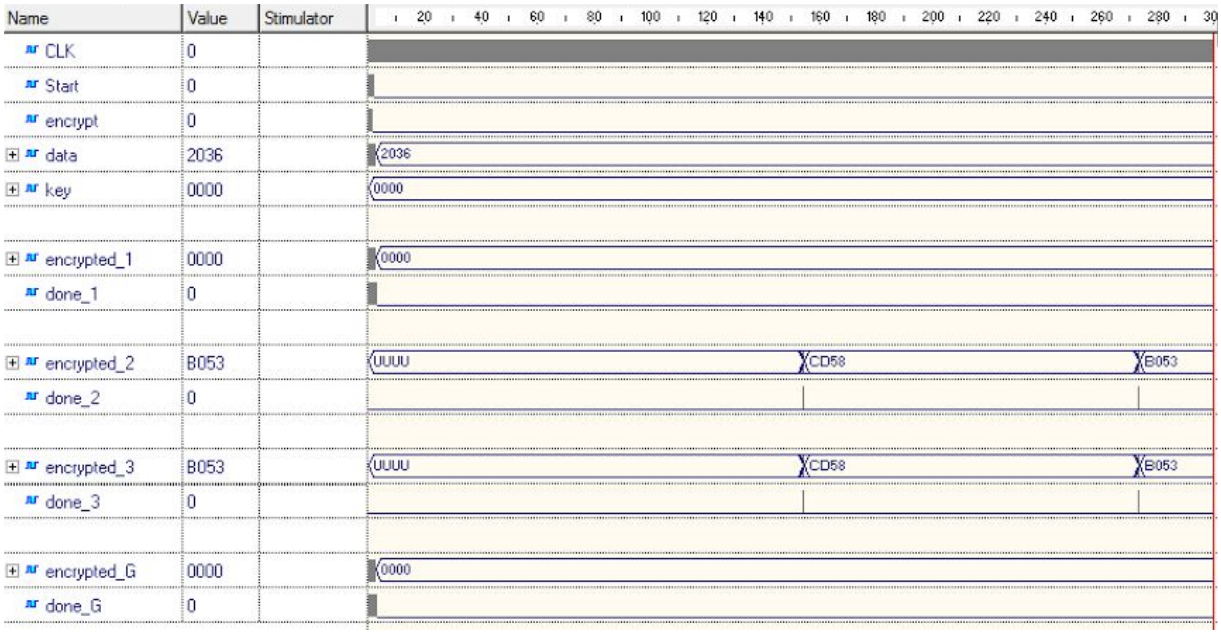


Fig. 4. Testbench waveform showing concurrent testing of all models

Performance Analysis

The data flow and structural models take 1508 clock cycles to encrypt or decrypt data from start signal to done signal. This does not include the 8 additional clock cycle to read out the encrypted data. Each cycle also contains parallelism during the linear transformation, allowing multiple steps to be performed at a time. This reduces the cycle/decycle cycle count by about 30%.

Summary

Given the time constraints and challenges posed by initially misinterpreting the paper, the project was not completed as it was initially assigned. A complete behavioral model was developed as a gold model and a full working data flow model was designed. One major component of the structural model, the cycle, was completed. Although the data flow model, and by extension the partial structural model, did not produce a matching cipher to the gold model, this is likely the result of initially misinterpreting the endianness of the system (vectors of bits are N downto 0 but the words are 0 to N) and could be corrected given more time. It should be noted however, that while the cypher was different between the Gold and Data_flow/structural, the model still decrypts it accurately. This is likely due to a consistent variation in the subkey generation between the gold and data_flow/structural models.

Had we had more time to work on the project, the process for moving forward to finish the structural model would have been to implement a structural architecture for each of the serpent sub-components used in the data flow model and replace them one at a time for testing. The full structural implementation would be done using existing and additional components in the extended_gate_lib.