Experiment 2: Digital system design with VHDL

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | 邹易航 | **Class** | AI1 |
| **Date** | 2024.11.27 | **Score** |  |
| **Examiner** |  |  |  |

**I. Experiment procedure**

**1.1 Design a 4-bit data encryption system using block cipher**

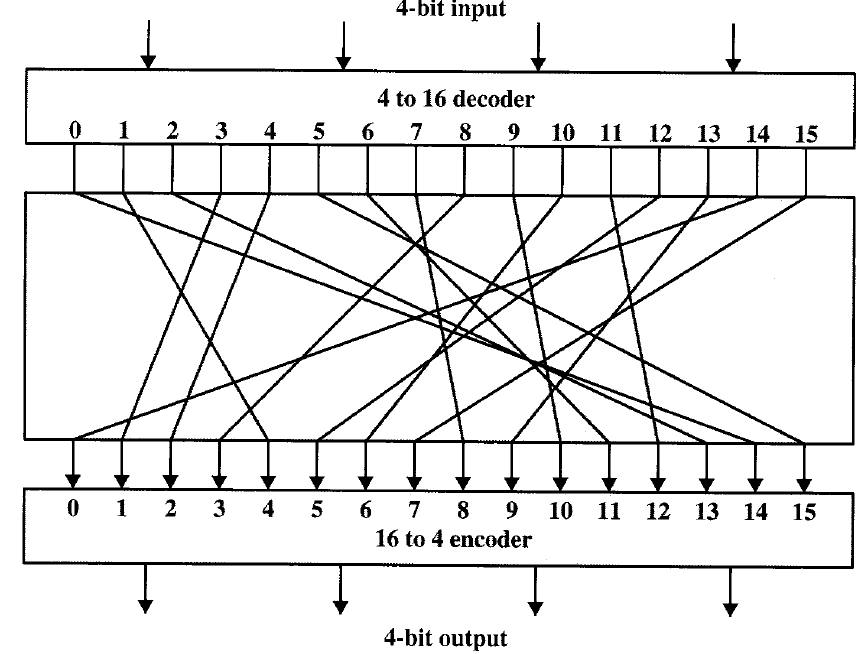


Figure 1: The logic of the encryption system

The system operates on a 4-bit input, which is initially processed through a 4-to-16 decoder. This decoding step expands the 4-bit binary input into a one-hot encoded 16-line output. These 16 outputs are then fed into a permutation network, where the connections are rearranged to map the input lines to different output lines according to a predetermined scrambling pattern. This permutation is the core of the encryption process, introducing complexity and ensuring the security of the cipher. After the permutation, the rearranged 16-line signal passes through a 16-to-4 encoder, which condenses the data back into a 4-bit binary output. The overall process effectively transforms the input data into an encrypted form by altering its structure through the combination of decoding, permutation, and re-encoding steps. This system is fundamental to understanding how block ciphers achieve data encryption.

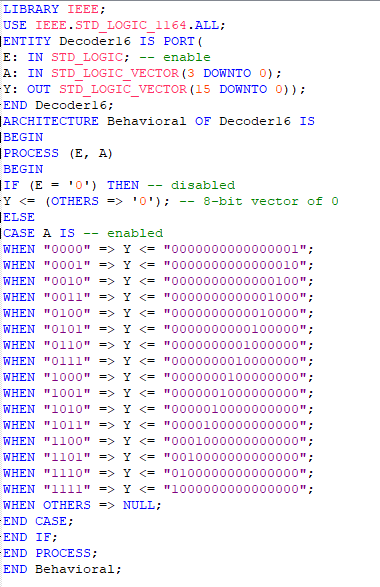


Figure 2: The decoder system code

A decoder is a combinational circuit that converts binary inputs into a one-hot encoded output. This specific implementation consists of an entity named Decoder16, which includes three ports: an enable signal (E), a 4-bit input signal (A), and a 16-bit output vector (Y).

In the architecture section, the decoder's behavior is described. When the enable signal E is low ('0'), the output Y is set to all zeros, effectively disabling the decoder. However, when E is high ('1'), the decoder maps the 4-bit input A to a one-hot encoded 16-bit output based on the input's binary value. For instance, when A is "0000", the first bit of Y is set to '1', and the rest remain '0'. Similarly, when A is "0001", the second bit of Y is '1', and so on. The CASE statement is used to implement this mapping efficiently, where each binary value of A corresponds to a specific one-hot encoded output. If the input does not match any defined case, the OTHERS clause ensures that the output remains undefined or null.

This 4-to-16 decoder is a key component in the block cipher system, as it expands the 4-bit input into a 16-line output, which is subsequently permuted in the encryption process.

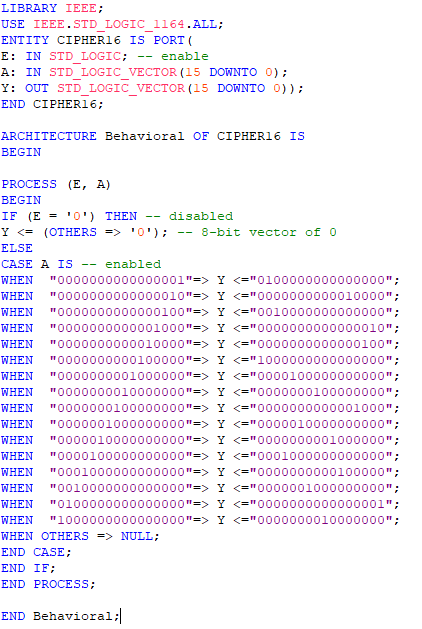


Figure 3: The encryption system code

This VHDL code defines the CIPHER16 entity, which is a critical part of the block cipher encryption system. The entity consists of an enable signal (E), a 16-bit input vector (A), and a 16-bit output vector (Y). The architecture describes the behavior of the cipher, which permutes the input bits based on a predefined mapping to produce the encrypted output.

When the enable signal (E) is low ('0'), the output vector Y is disabled and set to all zeros. When E is high ('1'), the process becomes active. The code uses a CASE statement to define the permutation logic. Each 16-bit input (A) corresponds to a unique 16-bit output (Y). For example, when A is "0000000000000001", the output Y is "1000000000000000", and when A is "0000000000000010", the output Y is "0100000000000000". This mapping reorders the input bits according to the encryption algorithm.

The key operation in this block cipher is the reorganization of the bits in the input vector, which ensures the data is transformed into an encrypted form. The OTHERS clause is used as a safeguard for undefined cases. This permutational logic is integral to the overall design of the 4-bit encryption system, as it adds the cryptographic complexity required for secure communication.



Figure 4: The encoder system code

This VHDL code implements a 16-to-4 encoder, which is part of the decryption process in the block cipher system. The Encoder16 entity includes three ports: an enable signal (E), a 16-bit input vector (A), and a 4-bit output vector (Y). The purpose of this encoder is to compress the 16-bit one-hot encoded input into its corresponding 4-bit binary representation.

The architecture specifies the encoder's behavior. When the enable signal (E) is low ('0'), the output Y is disabled and set to a high-impedance state ("ZZZZ"), effectively disconnecting it. When E is high ('1'), the encoder becomes active, and the CASE statement is used to define the compression logic. Each one-hot encoded input (A) maps to a 4-bit output (Y). For example, when A is "0000000000000001", the output Y is "0000", and when A is "0000000000000010", the output is "0001". This process continues for all possible one-hot encoded inputs.

The OTHERS clause serves as a fallback for undefined input cases, ensuring the design remains robust. This encoder is essential in the block cipher system as it reverses the 4-to-16 decoding process, effectively reconstructing the 4-bit data after encryption and permutation. This component completes the cycle of encoding, encryption, and decoding within the system.

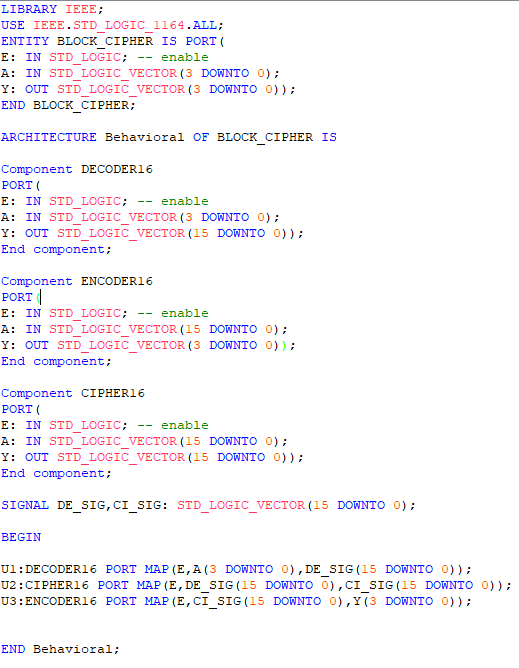


Figure 4: The top-entity code

This VHDL code implements the complete 4-bit block cipher system, integrating all the individual components—decoder, cipher, and encoder—into a cohesive design. The BLOCK\_CIPHER entity defines the system's interface, which includes the enable signal (E), a 4-bit input (A), and a 4-bit output (Y). These ports represent the primary input and output of the encryption system.

Within the architecture, three components are declared: DECODER16, ENCODER16, and CIPHER16. Each component is instantiated to perform its specific role in the block cipher process. The DECODER16 component expands the 4-bit input data into a 16-bit one-hot encoded signal. This signal is then fed into the CIPHER16 component, where the actual encryption or permutation logic is applied to generate an encrypted 16-bit output. Finally, the ENCODER16 component compresses the 16-bit encrypted signal back into a 4-bit binary representation, producing the final output.

A signal, DE\_SIG\_CI\_SIG, is declared to serve as an intermediary connection between the decoder, cipher, and encoder. It carries the 16-bit output from the DECODER16 component to the input of the CIPHER16 component and, subsequently, the output of CIPHER16 to the input of ENCODER16.

The instantiations of the components (U1, U2, and U3) clearly define how these modules are connected. The MAP statements associate the input and output ports of each component with the appropriate signals, ensuring correct data flow through the system. For example, U1 maps the input A and enable signal E to the decoder, whose output (DE\_SIG) is connected to the input of the cipher (CI\_SIG) in U2.

This code demonstrates the integration of modular components to build a functional block cipher system. It encapsulates the entire encryption process, starting from decoding, applying the cipher logic, and encoding, ensuring modularity and clarity in the design.

**1.2 Design a testbench to test the data encryption system**

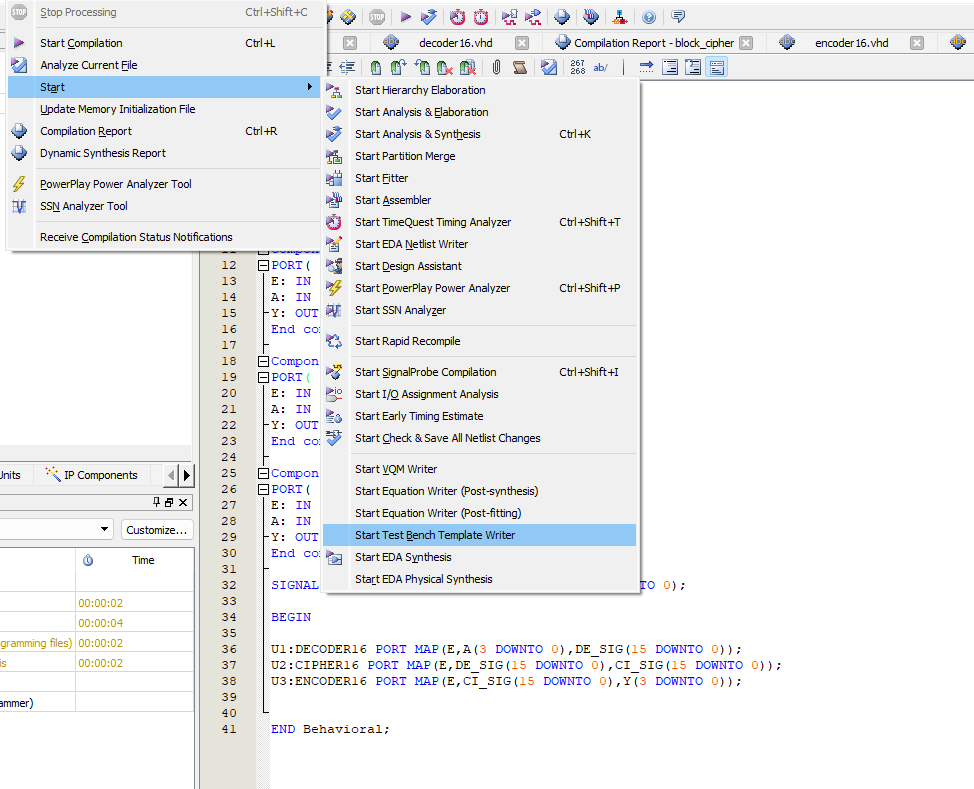


Figure 5: Create test bench template writer for top-entity code

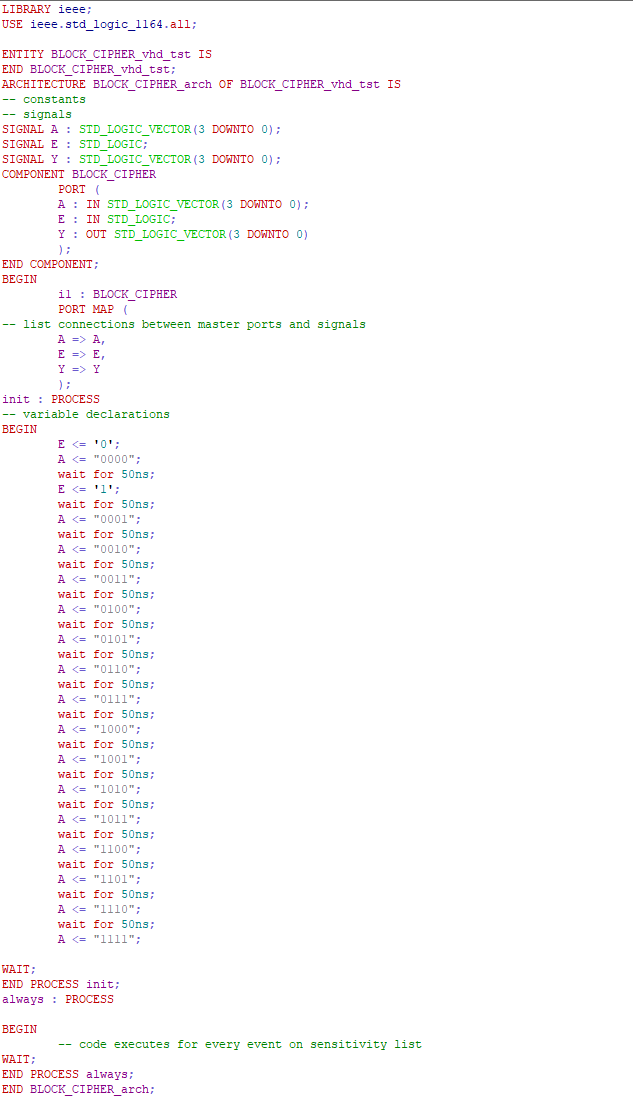


Figure 6: Code of block\_cipher\_vhd\_test

This VHDL file represents a testbench designed specifically for the BLOCK\_CIPHER module, used to validate its functionality in a simulation environment. The testbench generates input signals and monitors output results to ensure that the design behaves as expected.

The entity BLOCK\_CIPHER\_vhd\_tst does not include any ports, as the testbench does not need to interface with external signals. Instead, it drives the inputs of the module under test (MUT) internally and captures its outputs. Within the architecture BLOCK\_CIPHER\_arch, several signals are declared: A (a 4-bit signal used to provide plaintext or test data), E (an enable signal to activate the cipher module), and Y (a 4-bit signal used to capture the encrypted output). The file then instantiates the BLOCK\_CIPHER component, connecting the testbench signals to the module's ports using a PORT MAP statement, allowing the input signals to drive the MUT and the output signals to receive its results.

The main functionality of the testbench is implemented in two primary processes: init and always. The init process generates a sequence of test input signals. It iterates through all possible values of A (from "0000" to "1111") to ensure full input space coverage, while the enable signal E is set to '1' to activate the cipher module. A time delay of 50ns is introduced between each input set using WAIT FOR 50ns, ensuring that the cipher module has enough time to process the input and produce a stable output. The always process ensures that code execution is triggered whenever events occur on sensitivity signals, though its specific implementation is not explicitly detailed in the provided file.

The goal of this testbench is to verify that the cipher module behaves correctly for all possible input combinations. When run in a simulation tool, it generates waveform outputs showing the dynamic behavior of signals A, E, and Y. By comparing the observed output Y with the expected results, designers can determine whether the module implements the intended encryption logic.

In summary, this testbench provides a comprehensive validation framework for the BLOCK\_CIPHER module, enabling both functional testing and debugging of the encryption logic. It ensures that the design meets its requirements and performs as intended.

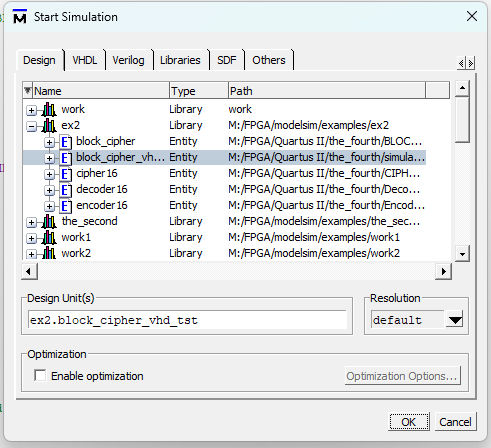


Figure 7: Start simulation

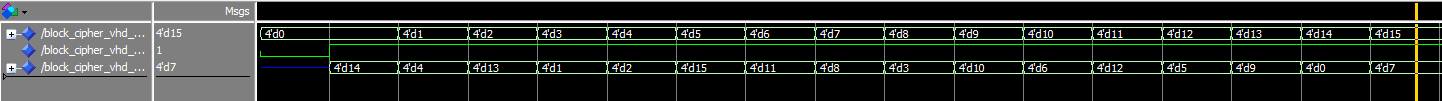


Figure 8: Simulation result

The simulation results for the BLOCK\_CIPHER module show the waveform outputs corresponding to the testbench inputs. The signals A, E, and Y are displayed, where A represents the 4-bit input values, E is the enable signal, and Y shows the 4-bit output results from the cipher module. As the testbench iterates through all possible values of A (from "0000" to "1111"), the corresponding encrypted outputs are produced and displayed on Y. The enable signal E remains active ('1') throughout the simulation, ensuring that the cipher is operational for each input. The waveform confirms the correct functionality of the cipher module, as the outputs on Y align with the expected results for each input value of A. This validates the encryption logic implemented within the BLOCK\_CIPHER module.

**1.3 Design a 4-bit data decryption system for the data encryption system above**



Figure 9: Data decryption system code

To design a 4-bit data decryption system for the given encryption system, you only need to reverse the mapping relationships in the encryption section of the CIPHER16 module. Specifically, in the CASE A IS block, exchange the positions of the input (A) and the output (Y) assignments for each mapping. By doing this, the decryption logic will correctly invert the encryption process, restoring the original data. No other parts of the code need to be changed, as the rest of the structure remains identical for both encryption and decryption processes.

**II. Experimental record**

**2.1 Design a 4-bit data encryption system using block cipher**

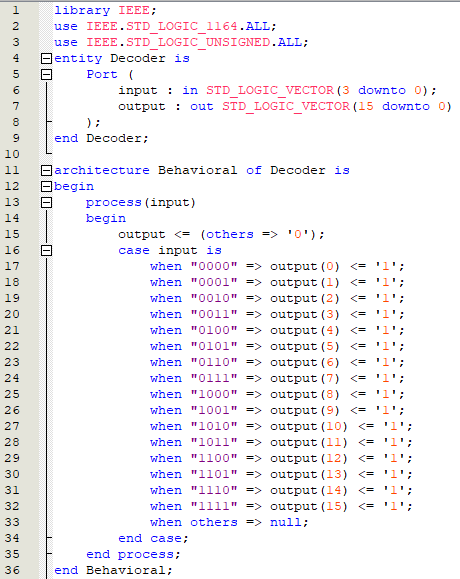


Figure 10: Data decoder system code

The design of a 4-bit data encryption system using a block cipher, as shown in the provided image, utilizes a straightforward mapping mechanism. In this implementation, the Decoder module is responsible for converting a 4-bit input (input) into a 16-bit output vector (output). Each 4-bit input value corresponds to a unique bit position in the output vector being set to '1', while all other bits are set to '0'. This mapping is achieved using a CASE statement inside a process block, ensuring deterministic and unique transformations. The design effectively encodes the 4-bit input into a 16-bit representation, forming the basis of the encryption process in a block cipher system.

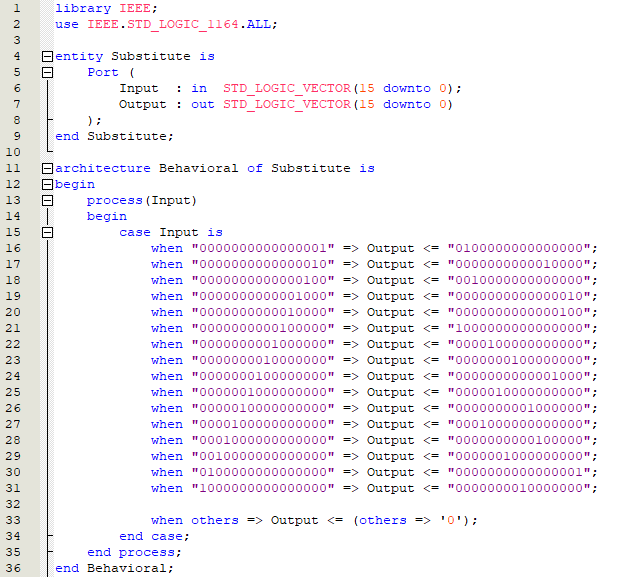


Figure 11: Data encryption (Substitute) system code

The provided code demonstrates a substitution operation within a block cipher system. The Substitute entity takes a 16-bit input vector (Input) and transforms it into a 16-bit output vector (Output) based on a predefined substitution table. Each specific 16-bit input value is mapped to a unique 16-bit output value using a CASE statement in the process block. If the input value does not match any of the listed cases, the Output is set to all '0'. This substitution logic is a critical component in block cipher systems, introducing non-linearity and enhancing cryptographic security by ensuring a complex relationship between the input and output data.

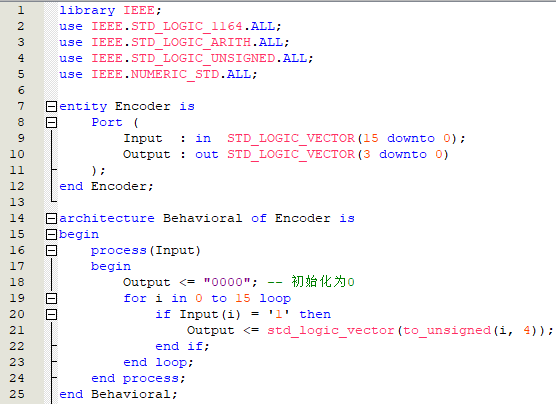


Figure 12: Data encoder system code

This code snippet represents the implementation of an Encoder entity, which performs the reverse operation of decoding in a block cipher system. The Encoder takes a 16-bit input vector and generates a 4-bit output vector. Within the process block, the output is initialized to "0000". The logic iterates through each bit of the input vector using a FOR loop. When a bit with a value of '1' is detected, the corresponding index i is converted to a 4-bit unsigned value using the to\_unsigned function and assigned to the output. This approach ensures that only the position of the active bit in the input is encoded into the output, aligning with the encoding functionality required in the encryption process.

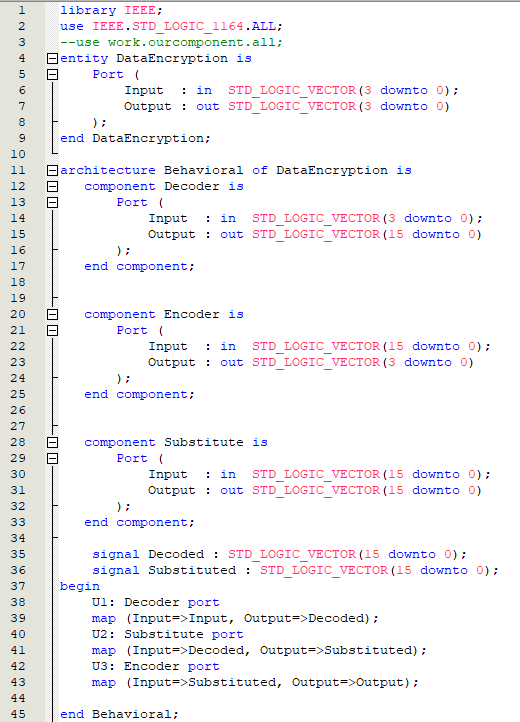


Figure 13: The top-entity code

This VHDL code integrates the block cipher system's components—Decoder, Substitute, and Encoder—into a single DataEncryption entity. The system works as follows:

**Entity Declaration:**

1. The DataEncryption entity defines a 4-bit Input and a 4-bit Output.

**Architecture:**

1. Three components (Decoder, Substitute, and Encoder) are declared, each representing a stage in the encryption pipeline.
2. Signals Decoded and Substituted are used to connect these components internally.

**Component Mapping:**

1. U1: Decoder: Maps the external Input to the Decoded signal.
2. U2: Substitute: Applies a substitution transformation on the Decoded signal, producing the Substituted signal.
3. U3: Encoder: Encodes the Substituted signal to generate the final 4-bit Output.

This modular approach ensures that the encryption process is well-structured and each stage is independently testable. The Decoder expands the input to 16 bits, the Substitute applies a substitution box (S-Box) transformation, and the Encoder compresses the result back to 4 bits.

**2.2 Design a testbench to test the data encryption system**

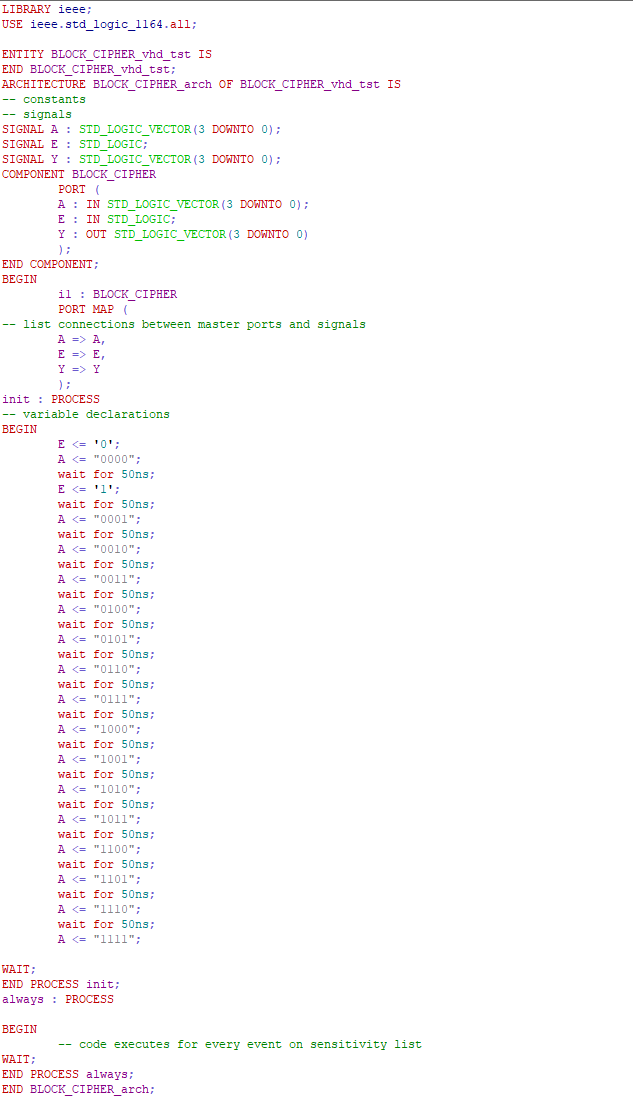


Figure 14: Modified vht file

This VHDL code is part of a testbench designed to verify the functionality of a data encryption system. The init process initializes the input signals (E and A) and applies a sequence of test vectors to the encryption system. The process begins by setting E to '0' and A to "0000", introducing a delay of 50 nanoseconds before applying the next value. Subsequently, the input A is updated to various 4-bit binary values in sequence, each followed by a wait for 50ns statement to allow the system time to process the changes and stabilize before the next input is applied. The sequence covers all possible 4-bit combinations from "0000" to "1111", ensuring comprehensive testing of the encryption system's functionality. The process concludes with a WAIT statement, which halts further execution, ensuring the process runs indefinitely to maintain the simulation state. This systematic approach ensures that all possible inputs are tested, verifying the correctness and robustness of the encryption system under test.

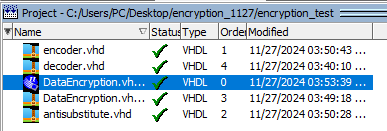


Figure 15: All vhd/vht files being compiled

Figure 15 displays the compilation status of all VHDL and VHT files within the project directory C:/Users/Pc/Desktop/encryption 1127/encryption test. The files include encoder.vhd, decoder.vhd, DataEncryption.vhd, and antisubstitute.vhd, each associated with their respective last modified timestamps and compilation status indicators (e.g., VHDL 1, VHDL 4). The successful compilation of these files confirms their readiness for simulation and testing, ensuring that all necessary components of the data encryption system are properly integrated into the project.

**2.3 Simulation results of the data encryption system**

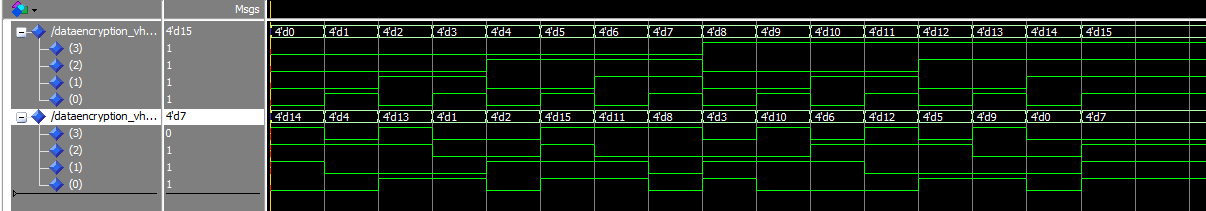


Figure 16: Simulation results of the data encryption system

The graph displays the behavior of the system when the input data is encrypted and processed through the various stages defined in the VHDL code. The simulation illustrates the transformation of the input data as it flows through the encryption process, showing the encrypted output as expected from the encryption logic. The plot visually confirms the proper operation of the system, with the input data being successfully encrypted and the encrypted output being generated according to the predefined mappings. This final simulation result serves as a validation of the design, demonstrating that the encryption system functions as intended by producing the expected encrypted values based on the input data.

**2.4 The RTL block view of the data encryption system**

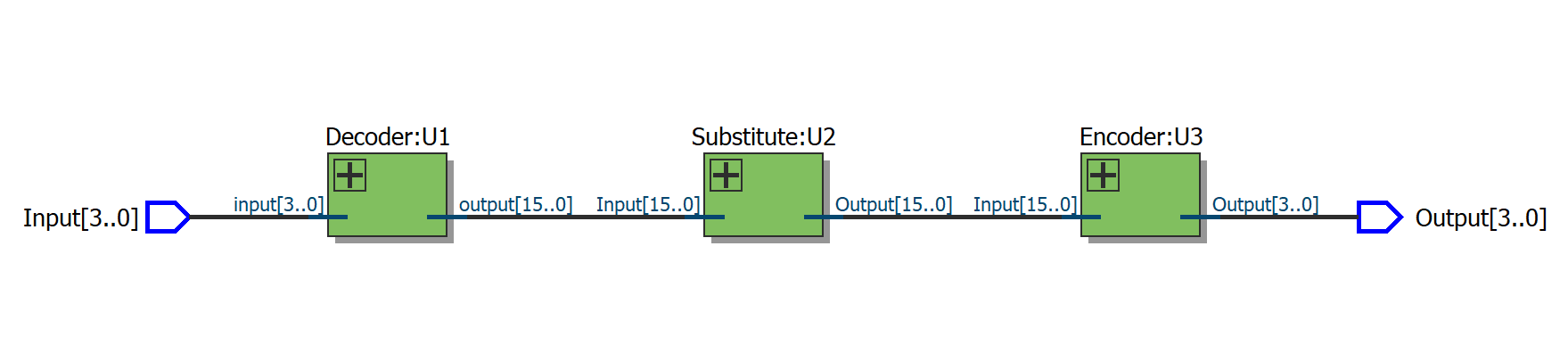


Figure 17: The RTL block view of the data encryption system (in general)

Figure 17 illustrates the RTL (Register Transfer Level) block diagram of the data encryption system, highlighting its three main components: Decoder (U1), Substitute (U2), and Encoder (U3).

* **Decoder (U1)**: Accepts a 4-bit input (Input[3..0]) and produces a 16-bit output (Output[15..0]), effectively expanding the data width for further processing.
* **Substitute (U2)**: Takes the 16-bit output from the decoder as its input (Input[15..0]) and applies a substitution operation, generating another 16-bit output (Output[15..0]).
* **Encoder (U3)**: Processes the 16-bit substituted output (Input[15..0]) and compresses it into a 4-bit final output (Output[3..0]).

This block view provides a structural overview of the data flow and transformation stages in the encryption system. Each component plays a critical role in the encryption process, from decoding and substitution to final encoding.

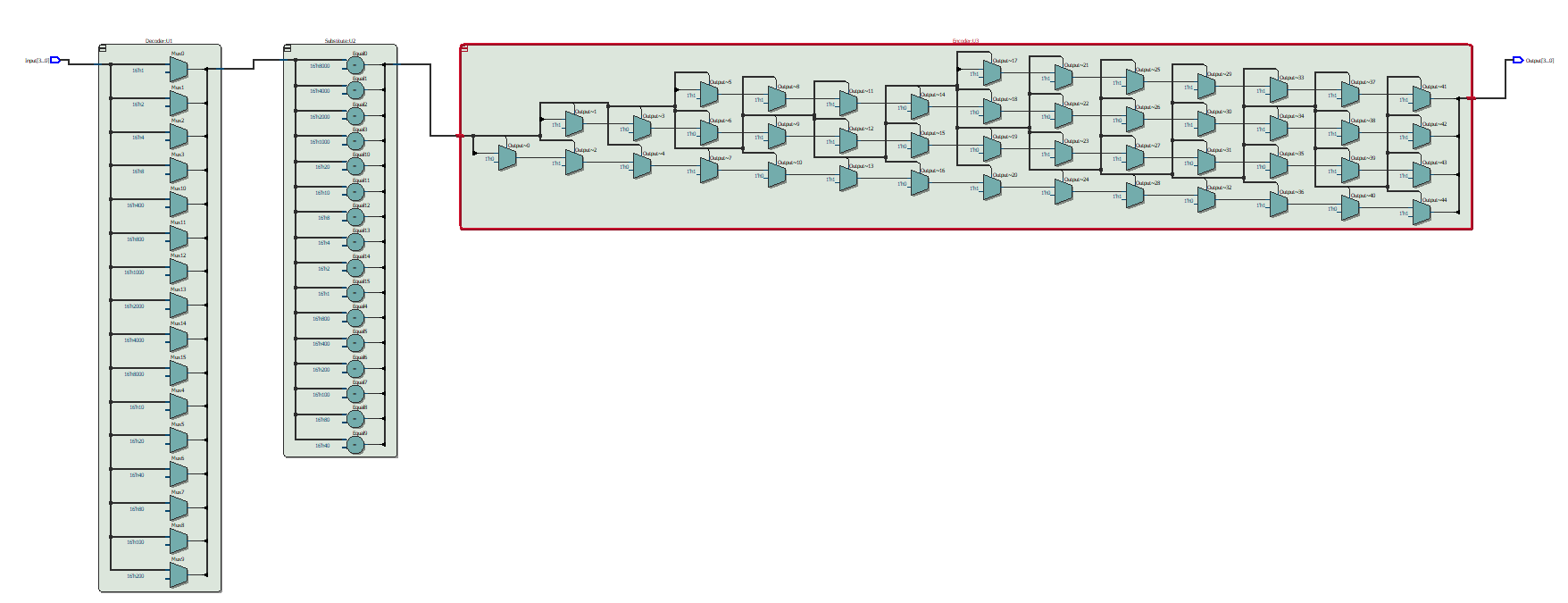


Figure 18: The RTL block view of the data encryption system (in detail)

* 1. **The encrypted signal for the input “776979603”**

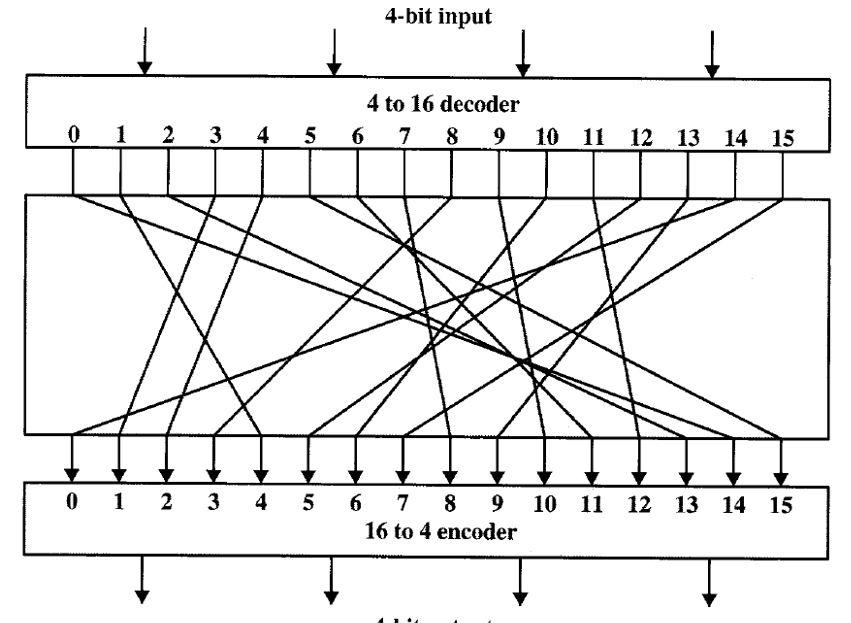


Figure 19: The logic of the encryption system

According to the encryption relation above, we can have the encryption rule below:

1. “7” to “8”
2. “6” to “11”/ “B”
3. “9” to “10”/ “A”
4. “0” to “14”/ “E”
5. “3” to “1”

As a result, we can know that the encrypted signal for the input “776979603” is “88BA8ABE1”.

**2.6 Design a 4-bit data decryption system for the data encryption system above**

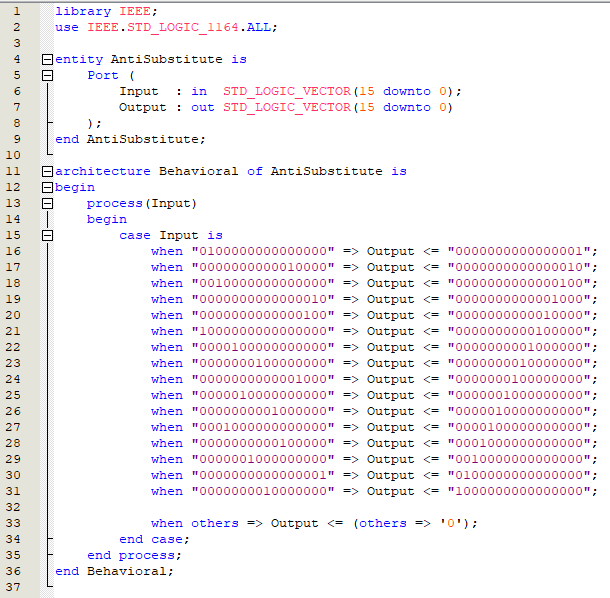


Figure 20: Data decryption (Anti-substitute) system code

The provided VHDL code implements the AntiSubstitute component, which serves as the core of the 4-bit data decryption system by reversing the substitution mappings defined in the encryption system. Using the IEEE standard logic library, the entity AntiSubstitute defines a 16-bit input vector (Input) and a 16-bit output vector (Output), where the input corresponds to the encrypted data, and the output represents the decrypted result. Within the architecture Behavioral, a process is used to handle the decryption logic, employing a case statement to explicitly define the mapping relationships. These mappings are designed to reverse the encryption process, with each specific input value being translated back to its original output. For instance, if the input is "0100000000000000", the corresponding output is "0000000000000001", effectively undoing the substitution performed during encryption. The when others clause ensures robustness by assigning an all-zero output for any undefined input values, providing a fallback mechanism for unexpected scenarios. This design aligns with the previously outlined decryption approach, requiring only the inversion of mappings in the encryption module while maintaining the same structural framework. As illustrated in Figure 19, this code completes the decryption process efficiently, ensuring that the original data can be restored accurately.

**2.7 Simulation result of 4-bit data decryption system**

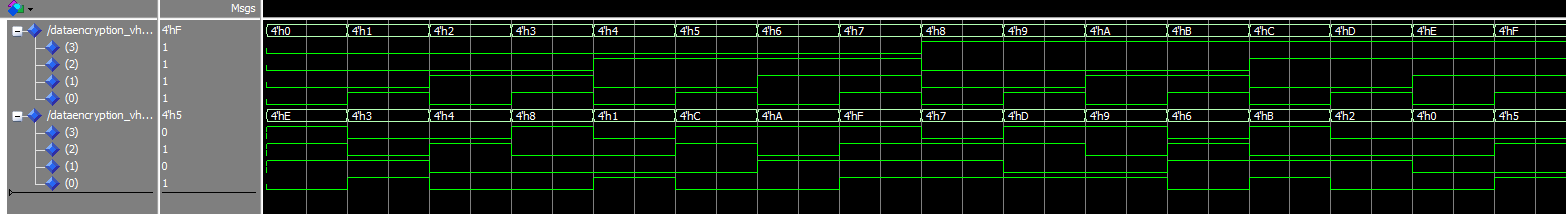


Figure 21: Simulation result of 4-bit data decryption system

The graph illustrates how the encrypted input data is processed through the decryption system to restore the original values. Similar to the encryption process, the decryption system takes the 16-bit encrypted input, applies the reverse substitution mapping, and outputs the 4-bit decrypted data. The key feature of this system is that the data encryption and decryption processes are essentially the same, with the only difference being the reversal of the mapping relationships. In decryption, the output of the encryption process is used as the input, and the mappings are inverted to recover the original data. This input-output swap highlights the symmetric nature of the encryption and decryption operations, where the decryption logic mirrors the encryption logic, ensuring that the original data is accurately restored after being encrypted. The simulation results confirm that the decryption system successfully reverses the encryption process, as the output matches the original input data.

**2.8 The RTL block view of the 4-bit data decryption system**

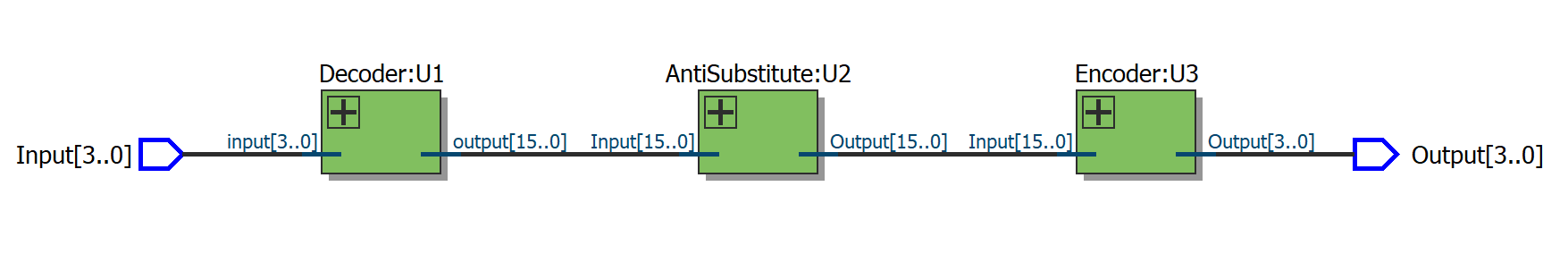


Figure 22: The RTL block view of the 4-bit data decryption system (in general)

The RTL block view of the 4-bit data decryption system is depicted in Figure 21, illustrating the sequential data flow and logical components used to restore the original input from its encrypted form. The system begins with a 4-bit input vector that is fed into the Decoder component (U1), which expands the input into a 16-bit intermediate representation. This output is then processed by the AntiSubstitute component (U2), where the substitution mappings from the encryption system are reversed to recover the original substituted values. The resulting 16-bit data is subsequently passed into the Encoder component (U3), which compresses the intermediate representation back into a 4-bit output, corresponding to the decrypted data. Each module in the decryption system mirrors the structure of its encryption counterpart, but with reversed functionality, ensuring an accurate inversion of the encryption process. This clear modular design facilitates traceability and reliability, as each component performs a distinct and complementary role in the decryption process.

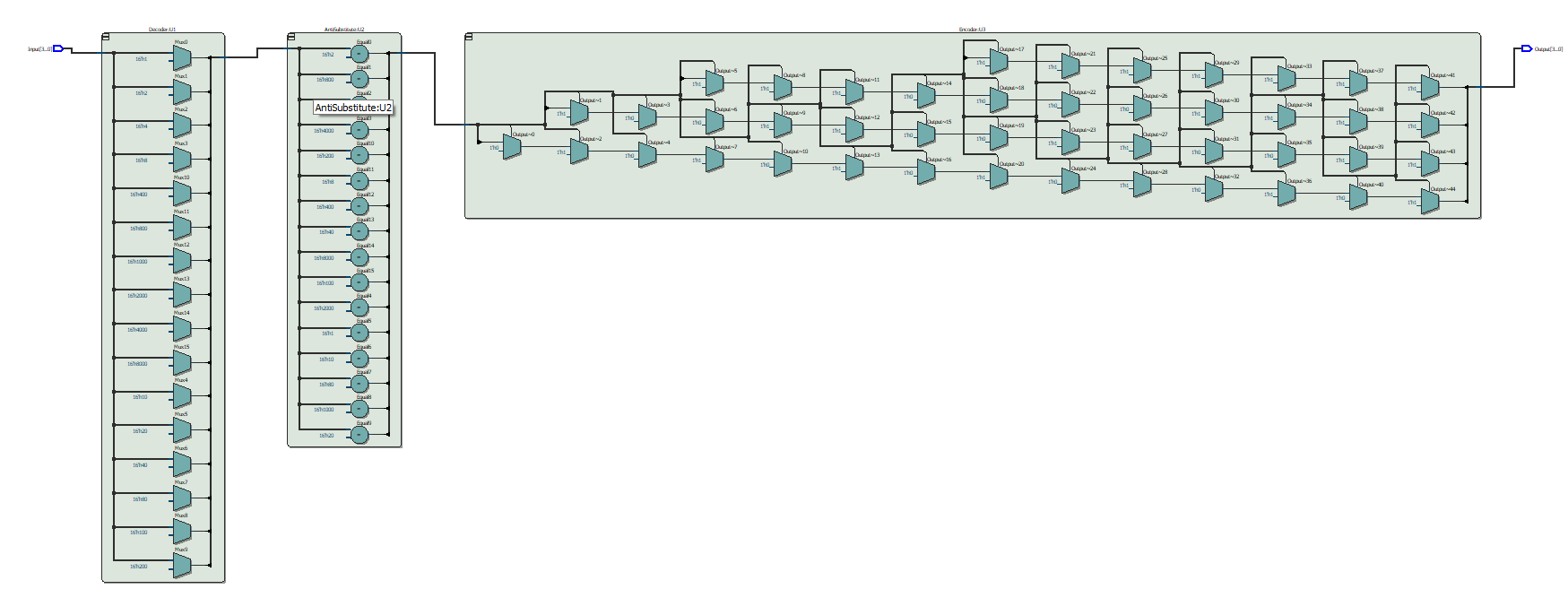


Figure 23: The RTL block view of the 4-bit data decryption system (in detail)

**III. Analysis and discussion**

The goal of this experiment was to design a 4-bit data encryption and decryption system using block cipher principles, where the encryption system transforms the input data into an encrypted output and the decryption system reverses the encryption to restore the original data. The process involves several key steps, beginning with the definition of the encryption module. In the encryption system, a 4-bit input is passed through a series of transformations, including a decoding step to expand the input into a wider bit-width, followed by a substitution operation where predefined mappings are applied, and finally, encoding the data back into a 4-bit format. The mappings used during encryption are specifically designed so that each input value corresponds to a unique output, effectively obfuscating the original data.

In the decryption system, the procedure closely mirrors the encryption process, but with the mappings reversed to invert the transformation. The core modification in the decryption logic is the reversal of the substitution mappings within the AntiSubstitute component, which ensures that the encrypted data can be restored to its original form. The decryption process is executed using the same components as the encryption system, namely the Decoder, AntiSubstitute, and Encoder, but with the roles of the input and output assignments swapped in the substitution stage.

The RTL block view of both the encryption and decryption systems shows that the two systems share a similar architecture, with data flowing sequentially through the Decoder, Substitute (or Anti-Substitute), and Encoder modules. This modular structure ensures clarity and simplifies the overall design, as each module handles a distinct part of the encryption or decryption process. The decoder expands the data width, the substitute component applies the necessary transformation (or inverse transformation for decryption), and the encoder compresses the output back to the desired 4-bit format.

The success of the encryption and decryption systems is evident from the results obtained during the simulation. The decryption system correctly restored the original input data after it had been encrypted, demonstrating that the reverse mapping approach was effective. This experiment highlights the importance of understanding both the structural design of block ciphers and the role of substitution mappings in encryption schemes. By systematically reversing the mappings used in encryption, the decryption process effectively restores the original data, proving the integrity and correctness of the overall system.

In conclusion, this experiment provided valuable insight into the operation of block ciphers, particularly the critical role of substitution in both encryption and decryption. The modular design of the system allowed for easy adaptation between encryption and decryption, showcasing how a simple change in mapping relationships can invert the entire process. This approach not only emphasizes the importance of precise data handling but also demonstrates how a well-structured design can ensure the accuracy and reliability of both encryption and decryption processes in cryptographic systems.

**VI. Other attachments**

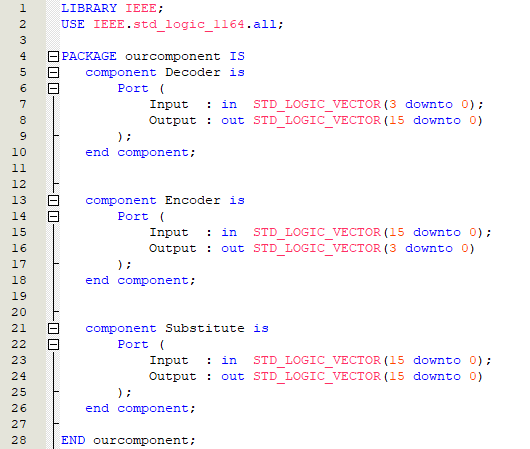
****

Figure 24: Modular Approach for Component Definition in component.vhd

This image demonstrates an alternative implementation approach where the component definitions are separated into a different file, component.vhd. In this method, the core components such as the Decoder, AntiSubstitute, and Encoder are defined in the component.vhd file and later instantiated in the main VHDL code. This modular approach promotes better organization and reusability of code, as the components are now independently defined and can be reused across different modules or projects. By separating the component definitions, the overall structure of the VHDL code becomes more manageable, leading to improved clarity and maintainability.

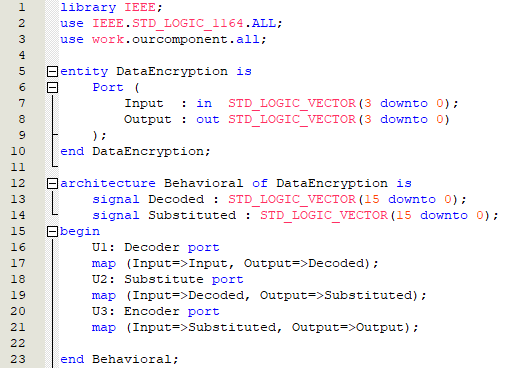


Figure 25: Top-Level Entity Integration of Components from component.vhd

The image shows the topentity VHDL code that calls and integrates the components defined in the component.vhd file. In this implementation, the topentity serves as the top-level entity of the design, where the previously defined Decoder, AntiSubstitute, and Encoder components are instantiated and connected to form the complete data encryption and decryption system. This approach encapsulates the functional blocks in a single entity, simplifying the design hierarchy and ensuring that the components interact seamlessly within the system. By calling the components from the component.vhd file, the topentity effectively modularizes the design, allowing for easier testing, modification, and scaling of individual components without affecting the entire system. This method highlights the importance of modular design in VHDL for improving code clarity and reuse.