



UNIVERSITY OF PISA
MASTER'S DEGREE IN CYBERSECURITY

SIMPLE ASYMMETRIC ENCRYPTION
ALGORITHM

COURSE HARDWARE AND EMBEDDED SECURITY

Author(s)
Conti Simone
Galli Gabriele

Academic year 2023/2024

Contents

1	Project Specifications	1
1.1	Asymmetric Encryption	1
1.2	Module specifications	1
1.3	Protocol scheme	2
1.4	Additional specifications	3
2	High-level Model	4
2.1	Key generation, encryption and decryption functions	4
2.2	reset_files auxiliary function	5
2.3	main function	6
3	RTL Design	8
3.1	Block diagrams	8
3.1.1	Key generation module	9
3.1.2	Encryption submodule	10
3.1.3	Decryption submodule	11
3.2	Design choices	11
4	Interface Specifications and Expected Behavior	14
4.1	Signal specification	14
4.1.1	Overview	14
4.2	Inputs and Outputs	14
4.3	Timing and Usage	15
4.4	Mode-Specific Operations	15
4.5	Waveforms	16
4.5.1	Key generation waveform	16
4.5.2	Encryption waveform	17
4.5.3	Decryption waveform	18
4.6	Summary	18
5	Functional Verification	19

5.1	Testbench architecture	20
5.2	Test vectors file	21
6	FPGA Implementation Results	22
6.1	Virtual pins assignment	22
6.2	Synthesized netlist	23
6.3	Static timing analysis	25

CHAPTER 1

Project Specifications

1.1 Asymmetric Encryption

Asymmetric encryption, also known as public key cryptography, uses a pair of keys: a public key, which can be freely distributed, and a private key, which must remain secret. A message encrypted with the public key can only be decrypted with the corresponding private key, and vice versa. This mechanism is the foundation of many current security infrastructures, such as digital signatures and secure key exchange.

One of the reasons asymmetric encryption is considered secure is the computational difficulty associated with the mathematical problems it is based on. These problems are easy to perform in one direction but extremely complex to solve in the reverse direction. This asymmetry makes public key cryptography particularly resistant to brute force attacks, ensuring a high level of security even against adversaries with significant computational resources.

1.2 Module specifications

The project specifications are the following:

- $C[i]$ is the 8-bit ASCII code of the i^{th} character of ciphertext.
- $P[i]$ is the 8-bit ASCII code of the i^{th} character of plaintext.
- Pk is the public key.
- Sk is the (secret) private key.
- p is the modulo equal to 223.

The asymmetric encryption system has three functions:

- **Key-pair generation** which given the (secret) private key Sk (random number) in the range 1 to $p - 1$, it generates the corresponding public key:

$$Pk = p - Sk$$

- **Encryption** which given a plaintext byte $P[i]$, it encrypts it (i.e., generate the corresponding ciphertext byte $C[i]$) according to the following encryption law:

$$C[i] = (P[i] + Pk) \bmod p$$

- **Decryption** which given a ciphertext byte $C[i]$, it decrypts it (i.e., generate the corresponding plaintext byte $P[i]$) according to the following encryption law:

$$P[i] = (C[i] + Sk) \bmod p$$

1.3 Protocol scheme

Assuming we have two identities, *Walter* and *Jesse*, the protocol will follow this pattern:

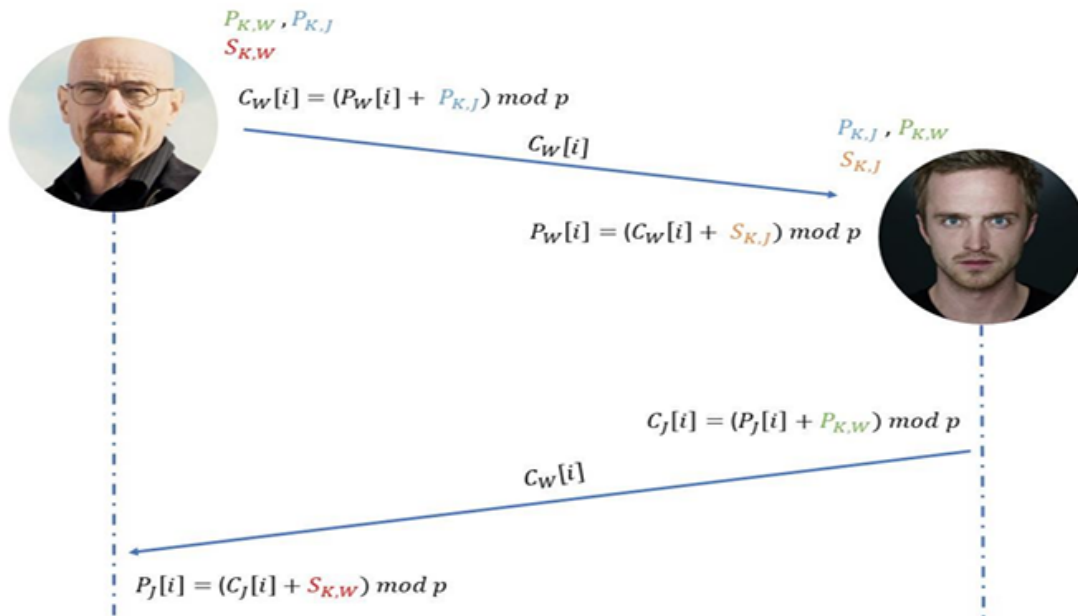


Figure 1: Simple asymmetric encryption scheme

Figure 1.1: Symple Asymmetric encryption scheme

The figure 1.1 shows how the encryption scheme is supposed to work. Initially, public keys are created for both Walter and Jesse using the key generation algorithm by passing the private keys held by the two entities. Subsequently, Walter encrypts his plaintext using the encryption algorithm with Jesse public key and sends it to Jesse. Jesse then decrypts the received message using its own private key. Afterward, he will send his own encrypted plaintext to Walter using Walter's public key, who will also decrypt it using its own private key.

1.4 Additional specifications

There are also additional design specifications:

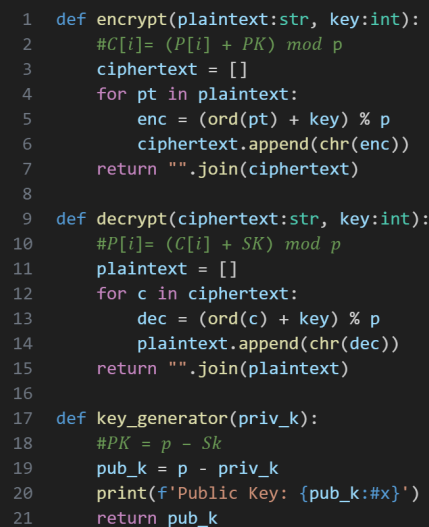
- The module shall support all the 3 functions, but it must execute one function at time.
- The module shall have an asynchronous active-low reset port.
- The module interface shall include input flags to be driven as it follows: 1'b1, when the corresponding input data on the corresponding input port is valid and stable (i.e. it can be used by the internal module logic), 1'b0, otherwise.
- The module interface shall include an output flag to be driven as it follows: 1'b1, when the corresponding output data on the corresponding output port is ready and stable (i.e., external modules can read and use it), 1'b0, otherwise.

CHAPTER 2

High-level Model

Before proceeding with the RTL simulation, a high-level model written in Python was created to test and debug the SAE module, which was implemented in hardware using SystemVerilog.

2.1 Key generation, encryption and decryption functions



```
1 def encrypt(plaintext:str, key:int):
2     #C[i]= (P[i] + PK) mod p
3     ciphertext = []
4     for pt in plaintext:
5         enc = (ord(pt) + key) % p
6         ciphertext.append(chr(enc))
7     return "".join(ciphertext)
8
9 def decrypt(ciphertext:str, key:int):
10    #P[i]= (C[i] + SK) mod p
11    plaintext = []
12    for c in ciphertext:
13        dec = (ord(c) + key) % p
14        plaintext.append(chr(dec))
15    return "".join(plaintext)
16
17 def key_generator(priv_k):
18    #PK = p - Sk
19    pub_k = p - priv_k
20    print(f'Public Key: {pub_k:#x}')
21    return pub_k
```

Figure 2.1: Encryption, decryption and key generation functions

The first functions written are for key generation (*key_generator(priv_k)*), encryption (*encrypt(plaintext:str, key:int)*), and decryption (*decrypt(ciphertext:str, key:int)*), according to the rules described in the section 1.2 of chapter 1.

The encrypt function takes a plaintext and a key as inputs. For each character in the plaintext, the function calculates the encrypted value using the formula

$$(ord(pt) + key) \% p$$

where *ord(pt)* converts the character into its Unicode integer value, *key* is the encryption key, and *p* is the modulus value. The result is then converted back into a character using *chr(enc)* and added to the ciphertext list. Finally, the list is joined into a string and returned as the ciphertext.

The decrypt function operates similarly but takes a ciphertext and a key as inputs. For each character in the ciphertext, the function calculates the decrypted value using the formula

$$(ord(c) + key) \% p$$

where *ord(c)* converts the encrypted character into its Unicode integer value. The result is then converted back into a character using *chr(dec)* and added to the plaintext list. Finally, the list is joined into a string and returned as the plaintext.

The *key_generator* function takes a private key (*priv_k*) as input and calculates the public key using the formula

$$p - priv_k$$

where *p* is the modulus value. The public key is then printed in hexadecimal format and returned.

2.2 reset_files auxiliary function

Another useful function is *reset_files()*, which clears the contents of the files where messages will be written. This is helpful for starting with a clean state when generating the content of the files that will be used during the protocol.

```
1 def reset_files():
2     current_folder = os.getcwd()
3     path_to_tv = current_folder + '/../modelsim/tv'
4     for filename in os.listdir(path_to_tv):
5         if filename != 'dictionary.txt':
6             with open('../modelsim/tv/' + filename, 'w') as file:
7                 file.write('')
```

Figure 2.2: Function *reset_files()*

2.3 main function

```
1 def main():
2     reset_files()
3
4     secretsGenerator = secrets.SystemRandom()
5
6     W_sK = bytes(f'{secretsGenerator.randrange(1, 222):08b}', 'ascii')
7     with open('../modelsim/tv/Walter_SK.txt', 'wb') as file:
8         file.write(W_sK)
9
10    J_sK = bytes(f'{secretsGenerator.randrange(1, 222):08b}', 'ascii')
11    with open('../modelsim/tv/Jesse_SK.txt', 'wb') as file:
12        file.write(J_sK)
13
14    with open('../modelsim/tv/dictionary.txt', 'r') as file:
15        all_the_lines = file.readlines()
16        line_to_read = secretsGenerator.randrange(1, len(all_the_lines))
17        plaintext_w = all_the_lines[line_to_read - 1]
18        line_to_read2 = secretsGenerator.randrange(1, len(all_the_lines))
19        plaintext_j = all_the_lines[line_to_read2 - 1]
20
21    with open('../modelsim/tv/Walter_PT.txt', 'w') as file:
22        file.write(plaintext_w)
23
24    with open('../modelsim/tv/Jesse_PT.txt', 'w') as file:
25        file.write(plaintext_j)
```

Figure 2.3: *main function*

At the beginning, an instance of `secrets.SystemRandom()` is created, which is a cryptographically secure random number generator.

Next, two secret keys, *W_sK* and *J_sK*, are generated using the *randrange* method to obtain a random number between 1 and 221. These numbers are then converted into 8-bit binary strings and subsequently into ASCII-encoded bytes. The generated keys are written to the files *Walter_SK.txt* and *Jesse_SK.txt* in binary mode.

The function continues by opening the file *dictionary.txt* in read mode and reading all the lines into a list named *all_the_lines*. Two random numbers are generated to select two different lines from the dictionary, which are then assigned to the variables *plaintext_w* and *plaintext_j*.

Finally, the selected plaintexts are written to the files *Walter_PT.txt* and *Jesse_PT.txt* in write mode. This process ensures that every time the function is executed, new secret keys and random plaintexts are generated and saved in their respective files.

```

1
2 W_sK = int(W_sK, 2)
3 print(f'Walter Key: {int(str(W_sK).encode("utf-8")).hex(), 16):#x}')
4 J_sK = int(J_sK, 2)
5 print(f'Jesse Key: {int(str(J_sK).encode("utf-8")).hex(), 16):#x}')
6
7 W_pK = key_generator(W_sK)
8 J_pK = key_generator(J_sK)
9
10 print(f'Walter PT: {int(plaintext_w.encode("utf-8")).hex(), 16):#x}')
11 ct_w = encrypt(plaintext_w, J_pK)
12 print(f'Walter CT: {int(ct_w.encode("utf-8")).hex(), 16):#x}')
13 decipher_text_w = decrypt(ct_w, J_sK)
14 print(f'Jesse should decipher text: {int(decipher_text_w.encode("utf-8")).hex(), 16):#x}')
15
16 print(f'Jesse PT: {int(plaintext_j.encode("utf-8")).hex(), 16):#x}')
17 ct_j = encrypt(plaintext_j, W_pK)
18 print(f'Jesse CT: {int(ct_j.encode("utf-8")).hex(), 16):#x}')
19 decipher_text_j = decrypt(ct_j, W_sK)
20 print(f'Walter should decipher text: {int(decipher_text_j.encode("utf-8")).hex(), 16):#x}')
21

```

Figure 2.4: *main*

Initially, Walter and Jesse's secret keys, W_sK and J_sK , are converted from binary strings to integers using the *int* function with base 2. The keys are then printed in hexadecimal format for an easier readability.

Next, Walter and Jesse's public keys are generated using the *key_generator* function. This function calculates the public key as the difference between a modulus value p and the secret key.

Walter's plaintext is converted to hexadecimal format and printed. It is then encrypted using Jesse's public key through the *encrypt* function. The resulting ciphertext is converted to hexadecimal and printed. The ciphertext is then decrypted using Jesse's secret key via the *decrypt* function, and the result is printed in hexadecimal.

The process is repeated for Jesse: his plaintext is converted to hexadecimal and printed, encrypted with Walter's public key, and the resulting ciphertext is printed. Finally, Jesse's ciphertext is decrypted with Walter's secret key, and the result is printed in hexadecimal.

CHAPTER 3

RTL Design

3.1 Block diagrams

We decided to implement the three functions (encryption, decryption and key-pair generation) as 3 different submodules. The instances of the 3 submodules are used by the main SAE module.

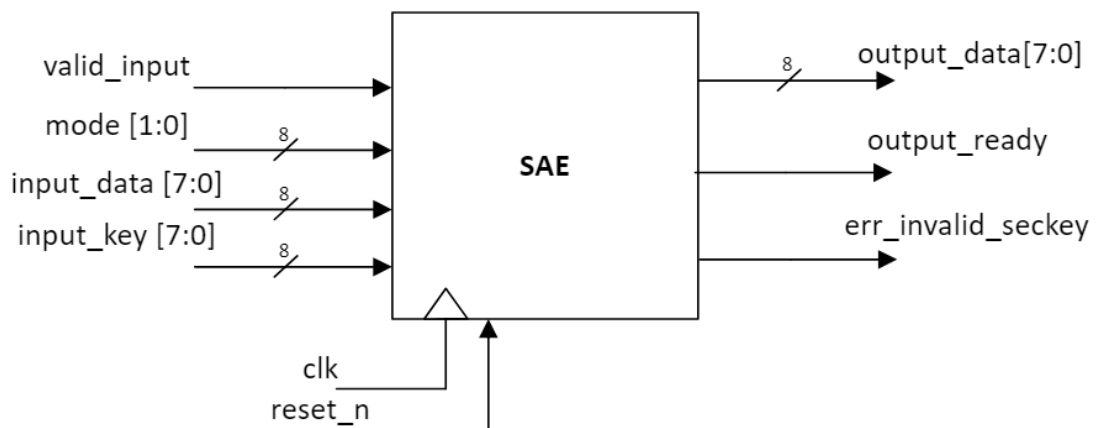


Figure 3.1: *SAE top level module*

Figure 3.1 illustrates the block diagram of the SAE module. Below there is the description of the ports used in the module:

- Input ***valid_input*** which indicates if the inputs are valid and can be sampled by the module.

- Input 2-bit **mode**, which allows the user to select which operations will be done.
- Input 8-bit **input_data** which represent the input data given by the user, which can either be the plaintext or ciphertext, depending on the mode.
- Input 8-bit **input_key** which represent the input key given by the user. It can be either the private or public key depending on the mode selected.
- Input **clk** which refers to the clock signal, all inputs are sampled in the rising edge of the clock.
- Input **reset_n** which refers to the active low asynchronous reset signal.
- Output 8-bit **output_data** which refers to the output of the module. It can be the public key, the ciphertext or plaintext depending on the mode.
- Output **output_ready** which indicates if the output is ready.
- Output **err_invalid_seckey** which indicates if the secret key is not a valid secret key.

Inside the SAE module we have 3 different submodules which implements the 3 different operating mode of the system.

3.1.1 Key generation module

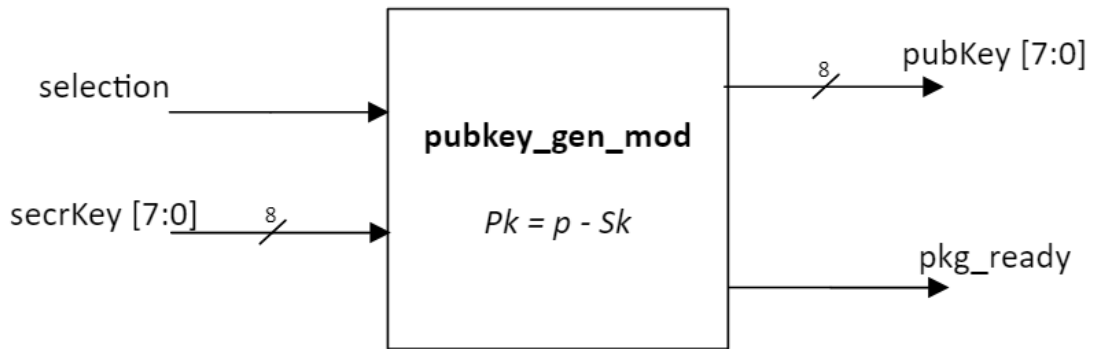


Figure 3.2: Public key generation submodule

Figure 3.2 shows the block diagram for the key generation submodule. It has the following ports:

- Input **selection**, when it is asserted to 1 the circuit is activated and the submodule calculates the public key with the formula

$$p - Sk$$

- Input 8-bit **secrKey**. This parameter must be between 1 and 223 (p value). It represents the private key of the user. If it is not between this range the operation of the public key generation will not be performed.

- Output 8-bit **pubKey** this is the output result of the calculation of the public key.
- Output **pkg_ready** which is the output bit that indicates when the output (public key) is ready.

3.1.2 Encryption submodule

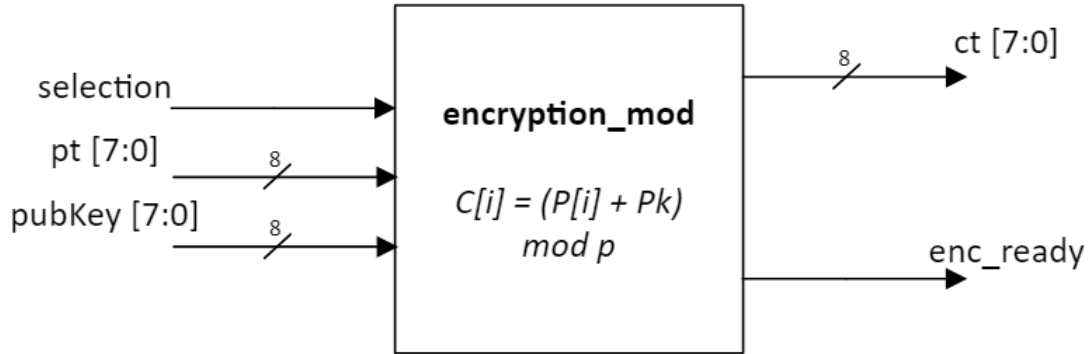


Figure 3.3: Encryption module

Figure 3.3 shows the block diagram for the encryption submodule. It has the following ports:

- Input **selection**, when it is asserted to 1 the circuit is activated and the submodule calculates the ciphertext from the plaintext and public key provided as input using the formula:

$$ct[i] = (pt[i] + Pk) \bmod p$$

where $ct[i]$ represents the bit of ciphertext, $pt[i]$ represents the bit of plaintext, Pk represents the public key and p is the modulo.

- Input 8-bit **pubKey** represents the public key given in input.
- Input 8-bit **pt** which represent the plaintext to be encrypted.
- Output 8-bit **ct** which represent the result from the encryption of the plaintext.
- Output **enc_ready** which is the output bit indicating when the output (the ct) is ready.

3.1.3 Decryption submodule

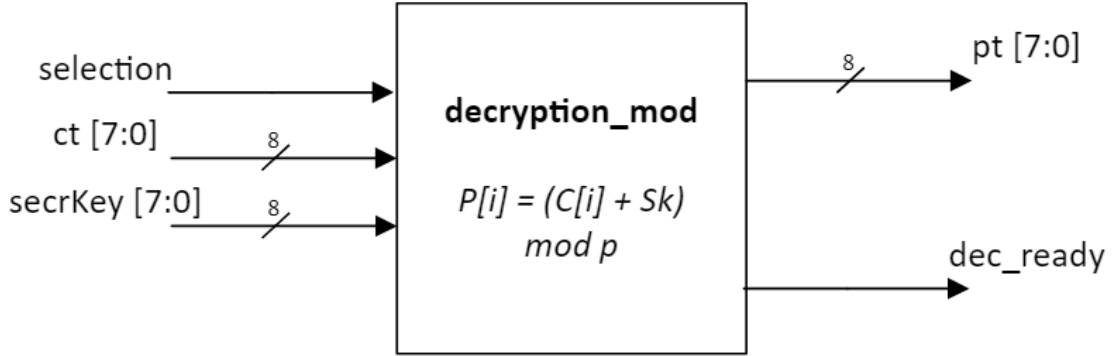


Figure 3.4: Decryption module

Figure 3.4 shows the block diagram for the decryption submodule. It has the following ports:

- Input **selection**, when it is asserted to 1, the circuit is activated and the submodule calculates the plaintext from the ciphertext and the secret key provided as inputs using the formula

$$pt[i] = (ct[i] + Sk) \bmod p$$

- Input 8-bit **secrKey** represents the secret key given in input.
- Input 8-bit **ct** which represent the ciphertext to be decrypted.
- Output 8-bit **pt** which represent the result from the decryption of the ciphertext.
- Output **dec_ready** which is the output bit indicating when the output (the pt) is ready.

3.2 Design choices

Figure 3.5 shows the circuitry to route the input to the submodules.

The 2-bit encoding of the input mode is used to select the corresponding submodule via a demultiplexer, which activates only one of its three output signals. Specifically:

- 01 sets the **pkg_sel** signal to 1 and activates the submodule for key-pair generation.
- 10 sets the **enc_sel** signal to 1 and activates the submodule for encryption.
- 11 sets the **dec_sel** signal to 1 and activates the submodule for decryption.
- 00 sets all three selection signals to 0 and does not activate any submodule.

Additionally, the **mode** signal is used to select which output to activate in two demultiplexers. The first allows handling **input_data**, which can be either plaintext or ciphertext depending on the selected mode. The second manages **input_key**, which can be a private key or a public key. Finally, the **mode** signal is also used to verify whether

the secret key or plaintext violates the requirement, specifically a secret key not between 1 and 222.

Regarding outputs, there are two multiplexers, also controlled by *mode*, which select the *output_ready* signal from one of the three operation completion signals (*pkg_ready*, *enc_ready*, *dec_ready*) and the *output_data* from the three submodule outputs (*pkg_output*, *enc_output*, and *dec_output*). The selected values are sampled by two registers and then output to the circuit.

To use the module, the user must set the *mode* input with the encoding of the desired operation, enter the *input_key* value (public or private depending on the operation), enter the *input_data* value (ciphertext or plaintext depending on the operation), and set the *valid_input* signal to 1. After one clock cycle, the output of the operation will be provided. The key-pair generation operation requires only the secret key as input; in that case, the user can enter any value in the *input_data* since it will be irrelevant.

The modulo operation, required by all three functions, was implemented through a series of if statements and subtractions. This method allowed the implementation of the operation without using dividers, which would have introduced longer delays.

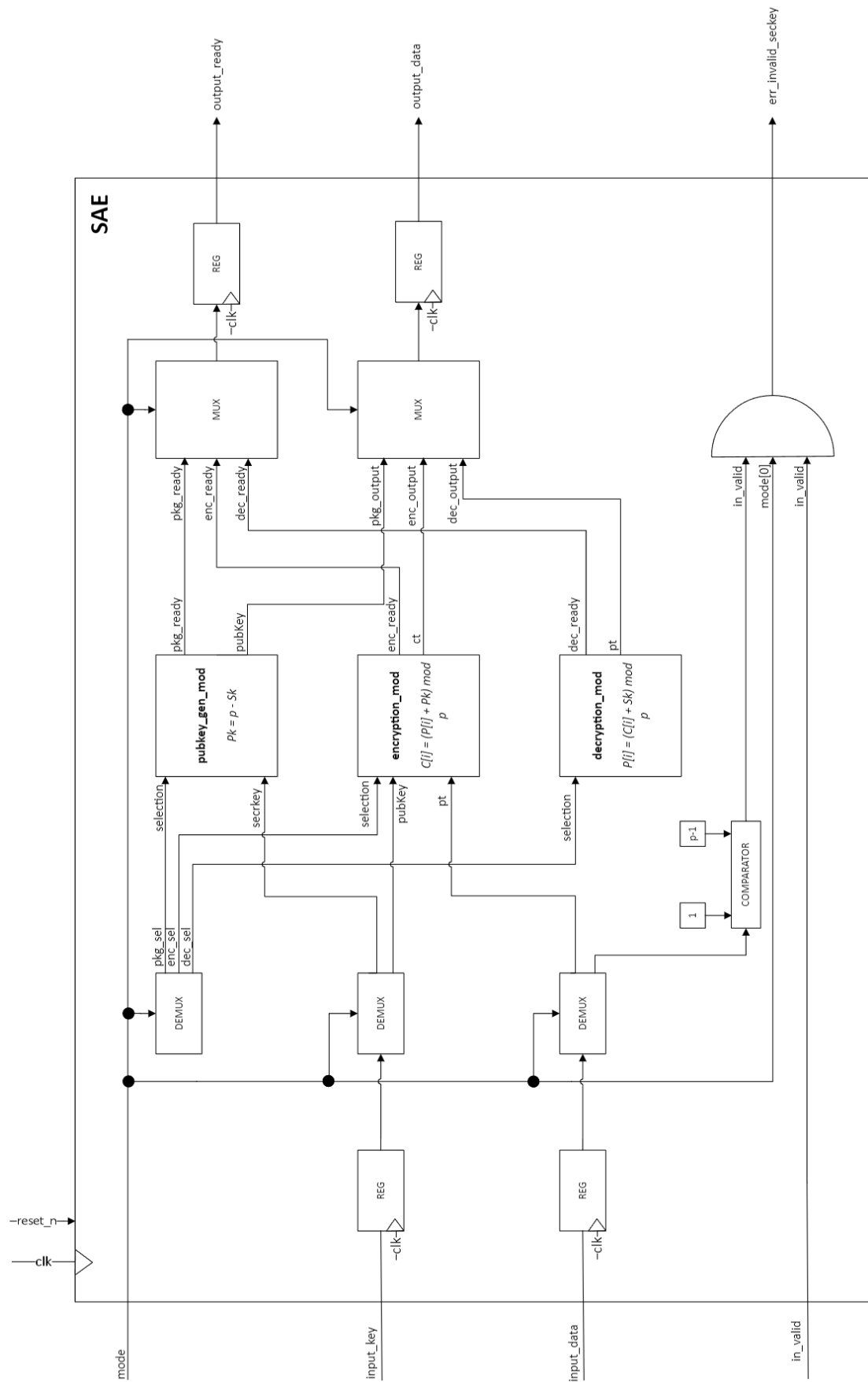


Figure 3.5: Circuit scheme

CHAPTER 4

Interface Specifications and Expected Behavior

In this chapter we discuss how the SAE module should be used in terms of how inputs should be provided, the timings and the corresponding outputs.

4.1 Signal specification

4.1.1 Overview

The SAE (Simple Asymmetric Encryption) module provides three primary cryptographic operations:

- Key-Pair Generation ($mode = 2'b01$).
- Encryption ($mode = 2'b10$).
- Decryption ($mode = 2'b11$).

These operations are driven by the control signal ($mode$), with input data and keys externally provided. The module processes the inputs according to the selected operation and outputs the corresponding result.

4.2 Inputs and Outputs

- ***clk***: Clock signal. All operations are synchronized with this clock.
- ***reset_n***: Active-low reset. Resets the internal state of the module.
- ***mode***: A 2-bit signal that selects the operation mode.
 - $2'b00$: inactive

- 2'b01: Key-pair generation.
- 2'b10: Encryption.
- 2'b11: Decryption.
- **input_data**: 8-bit input data. Depending on the mode, this can be the plaintext (pt) for encryption, the ciphertext (ct) for decryption, or unused in key-pair generation.
- **input_key**: 8-bit input key. Depending on the mode, this can be the secret key (Sk) or the public key (Pk).
- **valid_input**: A control signal that indicates when *input_data* and *input_key* are valid.
- **output_data**: 8-bit output data. Depending on the mode, This can be the public key (Pk), ciphertext (ct), or plaintext (pt).
- **output_ready**: Indicates when the *output_data* is ready.
- **err_invalid_seckey**: Error signal that indicates when an invalid secret key is provided.

4.3 Timing and Usage

1. Reset the module
 - Apply a low pulse to *reset_n* to initialize the module.
2. Provide Inputs
 - Set the *mode*, *input_data*, *input_key*, and *valid_input*.
 - Ensure *valid_input* is high for at least one clock cycle when providing new inputs.
3. Wait for Output
 - Monitor *output_ready*. When it goes high, the *output_data* is valid.
4. Error Handling
 - Check *err_invalid_seckey* for modes 2'b01 (key-pair generation) and 2'b11 (decryption). If high, the provided secret key is invalid.

4.4 Mode-Specific Operations

1. Key-Pair Generation (mode = 2'b01)
 - Inputs:
 - **input_key**: Secret key (Sk) (must be between 1 and 222).
 - **valid_input**: Should be high to indicate that *input_key* is valid.
 - Outputs:
 - **output_data**: Public key (Pk) calculated as $Pk = p - Sk$.

- **output_ready**: Goes high when *output_data* is valid.
- Timing:
 - After providing the valid input, wait for **output_ready** to go high to obtain the public key.
- Error:
 - **err_invalid_seckey** goes high if *input_key* is 0 or greater than 222.

2. Encryption (mode = 2'b10)

- Inputs:
 - **input_data**: Plaintext (pt).
 - **input_key**: Public key (Pk).
 - **valid_input**: Should be high to indicate that *input_data* and *input_key* are valid.
- Outputs:
 - **output_data**: Ciphertext (ct) calculated as $C[i] = (P[i] + Pk) \bmod p$.
 - **output_ready**: Goes high when 'output_data' is valid.
- Timing:
 - After providing the valid input, wait for **output_ready** to go high to obtain the ciphertext.

3. Decryption (mode = 2'b11)

- Inputs:
 - **input_data**: Ciphertext (ct).
 - **input_key**: Secret key (Sk) (must be between 1 and 222).
 - **valid_input**: Should be high to indicate that *input_data* and *input_key* are valid.
- Outputs:
 - **output_data**: Plaintext (pt) calculated as $P[i] = (C[i] + Sk) \bmod p$.
 - **output_ready**: Goes high when *output_data* is valid.
- Timing:
 - After providing the valid input, wait for **output_ready** to go high to obtain the plaintext.
- Error:
 - **err_invalid_seckey** goes high if *input_key* is 0 or greater than 222.

4.5 Waveforms

4.5.1 Key generation waveform

Figure 4.1 shows the waveform during the public key generation.

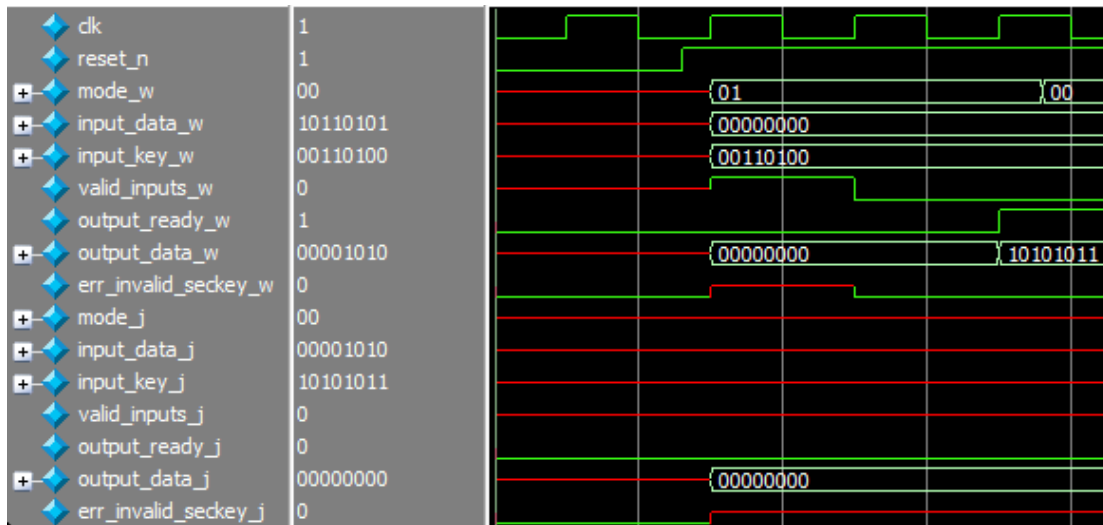


Figure 4.1: Key generation wave example

In the example the secret key has value 00110100, data has 00000000 (because it is not used in this mode), mode is set to 01, and at the end of the operation of key generation we have in the output the value 10101011 which is the corresponding public key.

The signal *output_ready_j* goes to 1 when the output is ready to be used for the next operations.

4.5.2 Encryption waveform

Figure 4.2 shows the waveform during the encryption.

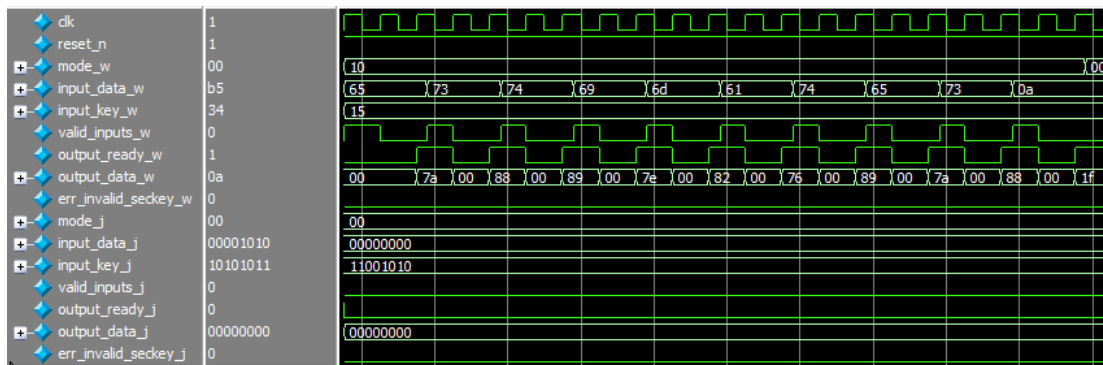


Figure 4.2: Encryption wave example

In the example the value are shown as hexadecimal to improve the readability. This is a waveform taken from one of the step of the testbench, as we can see we have in input the public key of Jesse placed into *input_key_w* and we have the pt of Walter inside *input_data_w*. In the output we can see all the resulting encryption of the bytes of plaintext which are sent to Jesse for the decryption phase. All of these operations are done when the mode is set to 10 (in binary form)

4.5.3 Decryption waveform

Figure 4.3 shows the waveform during the decryption phase

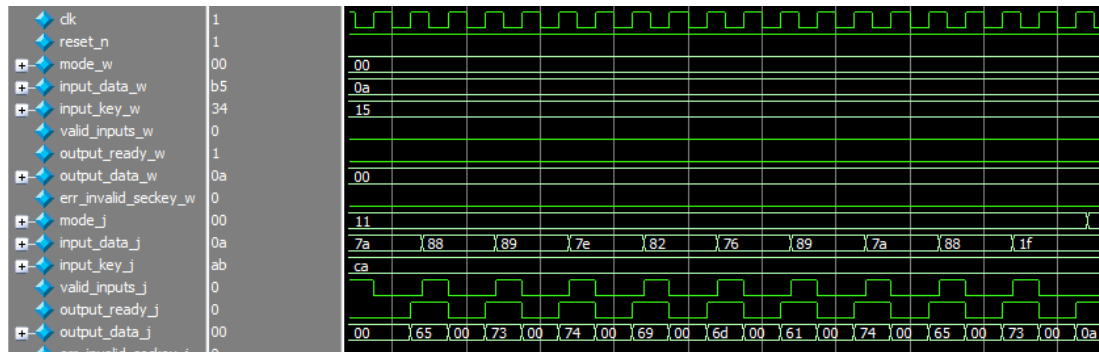


Figure 4.3: Encryption wave example

This is a waveform taken from one of the step of the testbench, right after the encryption shown in Figure 4.2. This time we are at Jesse side, the receiver of the encrypted msg. In input we have the private key of Jesse which will be used to decrypt Walter's encrypted message. The resulting plaintext can be seen in the variable `output_data_j`. All of these operations are done when the mode is set to 11 (in binary form).

4.6 Summary

- Ensure proper synchronization with the clock.
- Always check `output_ready` before reading `output_data`.
- Handle errors using `err_invalid_seckey` for key-pair generation and decryption modes.
- For any operation, ensure `valid_input` is asserted correctly and the mode is stable before sampling the output.

CHAPTER 5

Functional Verification

The high level module written in python has two roles:

- setting all the files used by the testbench,
- output the expected outputs for encryption and key generation.

By comparing the value obtained by the wave from the testbench with the output of the python script we can debug the correct functionality of the modules.

Because it might be better to have a testbench that checks the correct decryption and encryption by itself we decided to implement a check between the decrypted value obtained and the original content of the message sent by the user inside our testbench. If the two messages are equal, the testbench prints *"Plaintexts MATCH!"* as shown in figure 5.1, otherwise it prints *"Plaintexts NOT match"*.

```
# File Walter_SK was opened successfully : -2147483645
# Walt's private key was successfully loaded
# Walt's public key was generated
# File Walter_PK was opened successfully : -2147483645
# File Jesse_SK was opened successfully : -2147483645
# Jesse's private key was successfully loaded
# Jesse's public key was generated
# File Jesse_PK was opened successfully : -2147483645
# File Jesse_PK was opened successfully : -2147483645
# Jesse's public key was successfully loaded
# File Walter_PT was opened successfully : -2147483645
# File Jesse_CT was opened successfully : -2147483645
# File Jesse_SK was opened successfully : -2147483645
# Jesse's private key was successfully loaded
# File Jesse_CT was opened successfully : -2147483645
# File Walter_PT was opened successfully : -2147483645
# Plaintexts MATCH!
```

Figure 5.1: Testbench output example

We also print some debug information during the testbench in order to easily understand in which step the process may have failed. For example in the figure 5.1 we can also see some outputs such as *"Walt's private key was successfully loaded"*. If the key was not loaded correctly, we would have a different message here.

5.1 Testbench architecture

Our testbench architecture follows the protocol described in the chapter 1 Figure 1.1 as shown in Figure 5.2. We implemented the generation of public keys, encryption and decryption of a message both for Walter and Jesse. The part of generation of the public key is not shown in the figure 5.2 because we wrote the keys on the file so that we simulate that the keys were sent to the other party. The scheme shows after the key exchange between the two party what would happen for the exchange of the encrypted message.

The only main difference between the testbench of Figure 5.2 and the protocol described in the Figure 1.1 is the fact that, in the testbench, we check also if the obtained pt and the original pt are the same, as shown by the code of Figure 5.3.

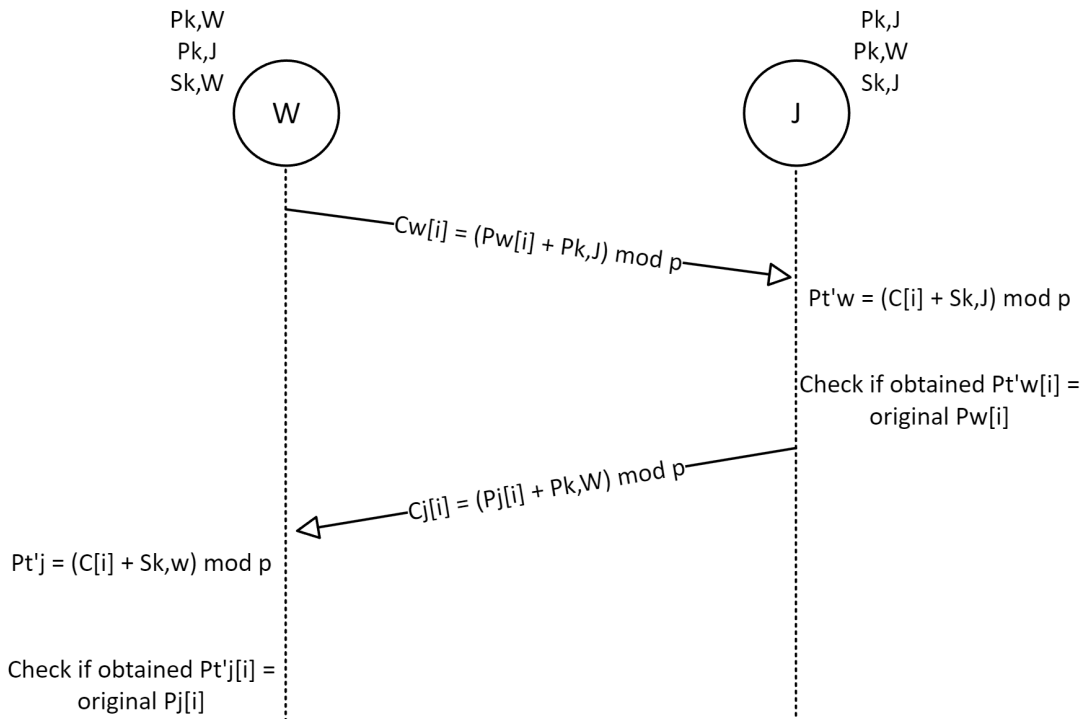


Figure 5.2: Testbench architecture

```

1  /* *
2      * CHECK THAT ORIGINAL PT = DECRYPTED PT
3      * */
4
5      // File containing the initial pt is opened, if the opening fails the execution is aborted.
6      FILE = $fopen("tv/Jesse_PT.txt", "r");
7      if (FILE) begin
8          $display("File Jesse_PT was opened successfully : %0d", FILE);
9      end
10     else begin
11         $display("File Jesse_PT was NOT opened successfully : %0d", FILE);
12         $finish;
13     end
14
15     // Initial pt is saved within the PT_J queue.
16     while($fscanf(FILE, "%c", char_w2) == 1) begin
17         PT_J.push_back(int'(char_w2));
18     end
19     $fclose(FILE);
20
21     // The two plaintexts are compared to verify that they are the same.
22     if(PT_J == PT_W) begin
23         $display("Plaintexts MATCH!");
24     end
25     else begin
26         $display("Plaintexts NOT match");
27     end

```

Figure 5.3: Matching comparison of decrypted ct and original pt

5.2 Test vectors file

The python scripts uses the dictionary.txt file to select two pt that will be used. It also generate the private keys for Walter and Jesse respectively. In the testbench we use the following files:

- Jesse_PK.txt: containing the public key of Jesse.
- Jesse_CT.txt: containing the ct sent to Jesse.
- Jesse_PT.txt: containing the pt of Jesse.
- Jesse_SK.txt: containing the private key of Jesse.
- Walter_PK.txt: containing the public key of Walter.
- Walter_CT.txt: containing the ct sent to Walter.
- Walter_PT.txt: containing the pt of Walter.
- Walter_SK.txt: containing the private key of Walter.

CHAPTER 6

FPGA Implementation Results

Quartus Prime tool has been used for the Logic Synthesis phase, Physical Design and Static Timing Analysis. For the synthesis of the module, the selected device was 5CGXFC9D6F27C7 FPGA, which is produced by INTEL and belongs to Cyclone V family.

6.1 Virtual pins assignment









	Status	From	To	Assignment Name	Value	Enabled	Entity
1	✓ Ok		 output_ready	Virtual Pin	On	Yes	SAE
2	✓ Ok		 reset_n	Virtual Pin	On	Yes	SAE
3	✓ Ok		 valid_input	Virtual Pin	On	Yes	SAE
4	✓ Ok		 input_data	Virtual Pin	On	Yes	SAE
5	✓ Ok		 input_key	Virtual Pin	On	Yes	SAE
6	✓ Ok		 mode	Virtual Pin	On	Yes	SAE
7	✓ Ok		 output_data	Virtual Pin	On	Yes	SAE
8	✓ Ok		 err_i...eckey	Virtual Pin	On	Yes	SAE
9		<<new>>	<<new>>	<<new>>			

Figure 6.1: *Virtual pins assignment on Quartus*

Before the compilation we assigned the virtual pins as shown in Figure 6.1. The table below shows the amount of virtual pins assigned. We have 8 pins for the 8-bit input/output ports, 1 pin for the 1 bit input/output ports and 2 pins for the 2-bit ports

port	virtual pin
valid_input	1
reset_n	1
output_ready	1
output_data	8
input_data	8
mode	2
input_key	8
err_invalid_seckey	1

After the virtual pin assignment, we did the compilation. Figure 6.2 shows the compilation report. Cyclone V offers a logic utilization (Adaptive Logic Modules) of 113560 but we only used 80 of them, which is less than 1%. This means that we are wasting a lot of resources only to implement this SAE module. We had a total of 30 registers and we used a total of 1 pin out of 378.

Flow Summary	
<<Filter>>	
Flow Status	Successful - Sun Sep 1 16:25:29 2024
Quartus Prime Version	21.1.0 Build 842 10/21/2021 SJ Lite Edition
Revision Name	SAE
Top-level Entity Name	SAE
Family	Cyclone V
Device	5CGXFC9D6F27C7
Timing Models	Final
Logic utilization (in ALMs)	80 / 113,560 (< 1 %)
Total registers	30
Total pins	1 / 378 (< 1 %)
Total virtual pins	30
Total block memory bits	0 / 12,492,800 (0 %)
Total DSP Blocks	0 / 342 (0 %)
Total HSSI RX PCSs	0 / 9 (0 %)
Total HSSI PMA RX Deserializers	0 / 9 (0 %)
Total HSSI TX PCSs	0 / 9 (0 %)
Total HSSI PMA TX Serializers	0 / 9 (0 %)
Total PLLs	0 / 17 (0 %)
Total DLLs	0 / 4 (0 %)

Figure 6.2: *Quartus Prime report generated after compilation*

6.2 Synthetized netlist

In this section we show the Physical Design as generated by Quartus Prime.

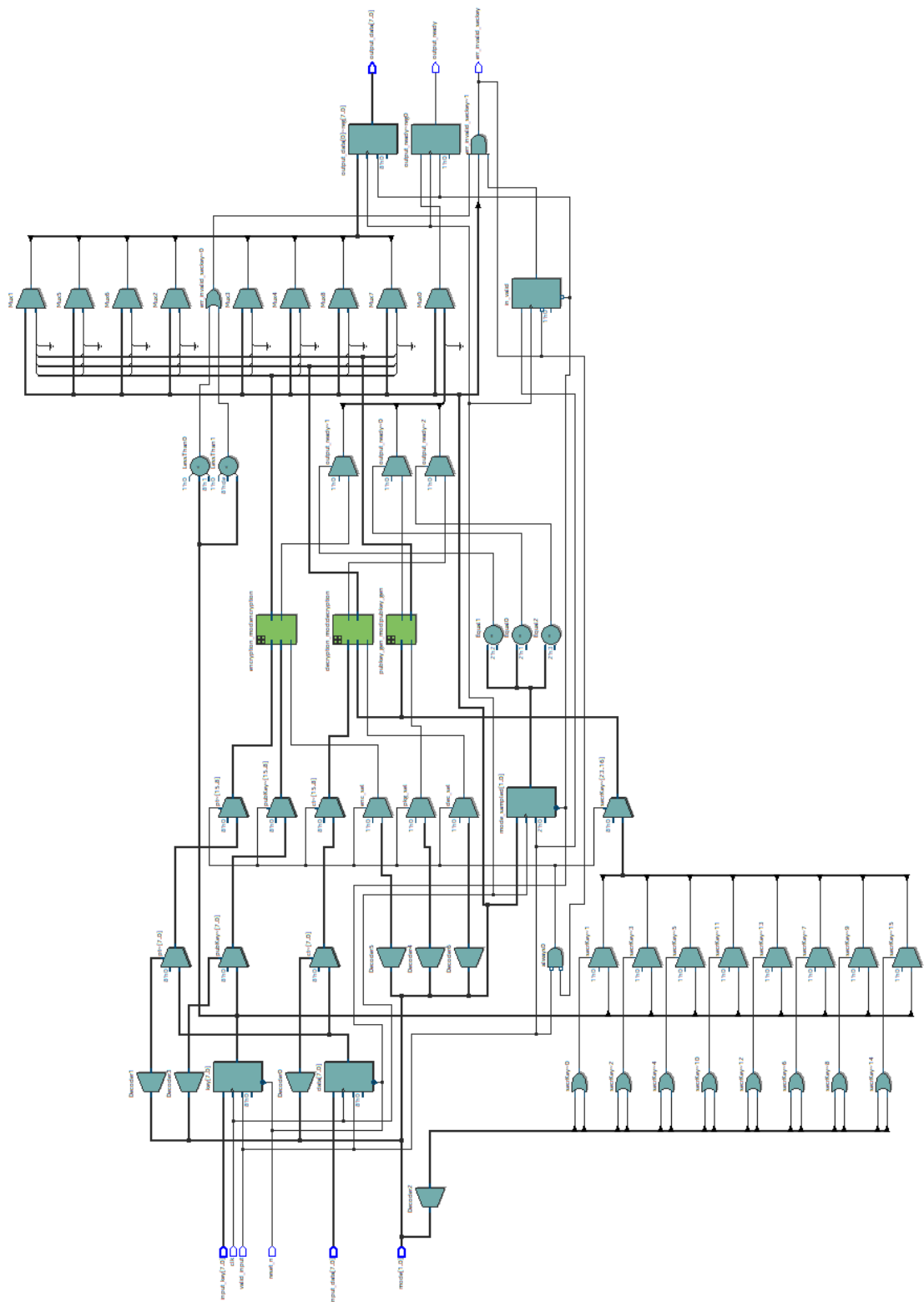


Figure 6.3: Quartus Prime report generated after compilation

Figure 6.3 shows the RTL design of the SAE.

6.3 Static timing analysis

To be able to make the Static Timing Analysis and optimize the frequencies, we set these constraints values in the *time_constr_template.sdc* file:

```
1 create_clock -name clk -period 15 [get_ports clk]
2
3 set_false_path -from [get_ports reset_n] -to [get_clocks clk]
4
5 set_input_delay -min 1.5 -clock [get_clocks clk] [all_inputs ]
6 set_input_delay -max 3 -clock [get_clocks clk] [all_inputs ]
7 set_output_delay -min 1.5 -clock [get_clocks clk] [all_outputs]
8 set_output_delay -max 3 -clock [get_clocks clk] [all_outputs]
```

Figure 6.4: Constraints in *time_constr_template.sdc* file

- The clock signal *clk* has been set to a period of 18 time units.
- Minimum and maximum input delays for all input ports relative to the *clk* clock are set to 1.8 and 3.6 time units (which are 10% and 20% of the clock period), respectively.
- Minimum and maximum output delays for all input ports relative to the *clk* clock are set to 1.8 and 3.6 time units (which are 10% and 20% of the clock period), respectively.

The command *set_false_path*, identifies paths in a design that are to be marked as false, so that they are not considered during timing analysis. In this case, all the path from the port *reset* to the clock are ignored.

The module was tested at a voltage of 1,100 mV in two temperatures: 0 and 85 Celsius degrees, that is the temperature corner values of Commercial Grade chips. The maximum frequency obtained at 0°C, as show in Figure 6.5 is 87.2 MHZ.


Slow 1100mV 0C Model Fmax Summary				
 <<Filter>>				
	Fmax	Restricted Fmax	Clock Name	Note
1	87.2 MHz	87.2 MHz	clk	

Figure 6.5: 0°C frequencies

The maximum frequency obtained at 85°C, as show in Figure 6.6 is 87.8 MHZ.


Slow 1100mV 85C Model Fmax Summary				
 <<Filter>>				
	Fmax	Restricted Fmax	Clock Name	Note
1	87.8 MHz	87.8 MHz	clk	

Figure 6.6: 85°C frequencies

The variation of the maximum frequency does not change a lot depending on environmental conditions.

The maximum frequency between the two is the one shown in Figure 6.5 which is the lowest one between the two, because if we can assure that our module works in the worst case it works for all the others as well.