UNIVERSITÀ DI PISA

Hardware and Embedded Security

Simple Asymmetric Encryption Algorithm v2

Sofia Bonapace
Roberto Paone
Giuseppe Pericone

Academic Year 2021/2022

Chapter 1

Specification Analysis

1.1 Introduction

The goal of the project is to design a simple asymmetric encryption system. Asymmetric encryption is also called public key encryption, but it actually relies on a key pair: two mathematically related keys, one called the public key (P_k) and another called the private key or secret key (S_k), are generated to be used together. The private key is never shared, it is kept secret and is used only by its owner. The public key is made available to anyone who wants it. Because of the time and amount of computer processing power required, it is considered "mathematically unfeasible" for anyone to be able to use the public key to re-create the private key, so this form of encryption is considered very secure.

C[i] is 8-bit ASCII code for the i^{th} character of ciphertext;

P[i] is 8-bit ASCII code for the i^{th} character of plaintext;

$\mathbf{P}_k$ is the Public key;

$\mathbf{S}_k$ is the Secret key;

p is the modulo equal to 227;

q is a parameter equal to 225.

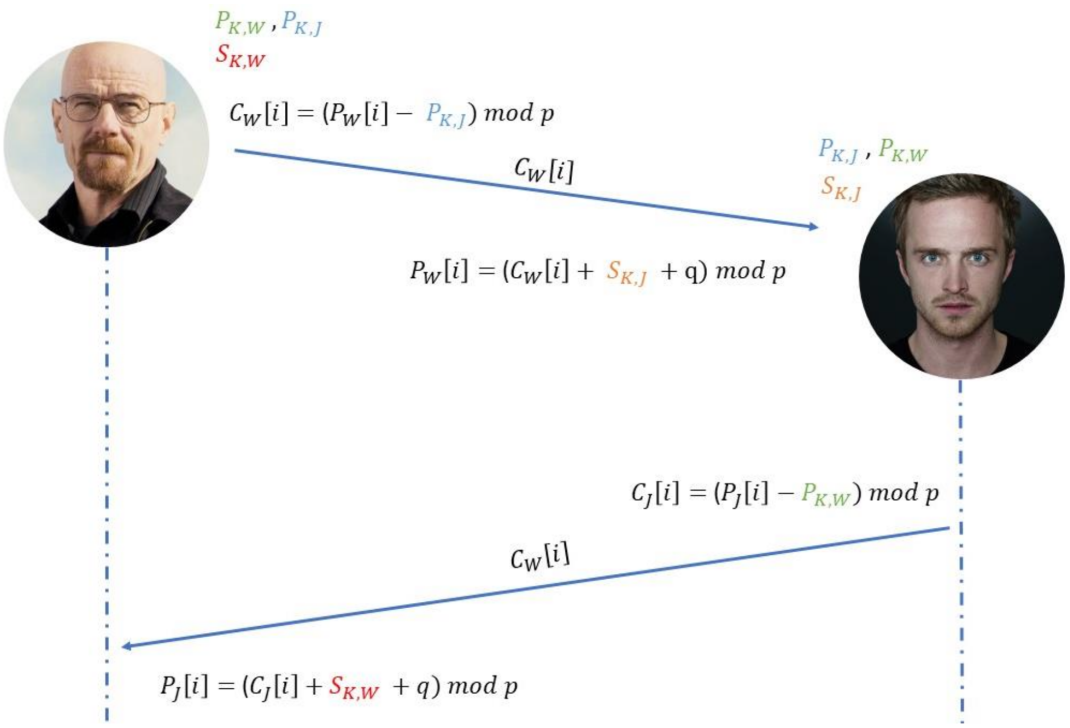

Figure 1.1: Communication scheme for the SAE described within the specifications.

The asymmetric encryption system has three functions:

- Generation of the key-pair, the key-pair generation function is described by the formula $P_k = (S_k + q)modp$. Given a secret key (S_k), generated in a random way and having a value between 1 and 226, it generates the corresponding public key (P_k).
- Encryption, the encryption function is described by the formula $C[i] = (P[i] - P_k)modp$. Given each character of the plaintext, the formula above is used to encrypt it.
- Decryption, the decryption function is described by the formula $P[i] = (C[i] + S_k + q)modp$. Given each character of the ciphertext, the formula above is used to decrypt it.

In Figure 1.1 it is possible to observe how the algorithm works. We have two entities, each with a key-pair (S_k and P_k). To encrypt the message, an entity uses the public key of the other entity, whereas for the decryption the entity uses its own secret key. In our case, we have two entities, Walt and Jesse, that want to communicate. The first step is the key-pair generation, made by each of the entities. Once they obtained the pair, they have to exchange the public

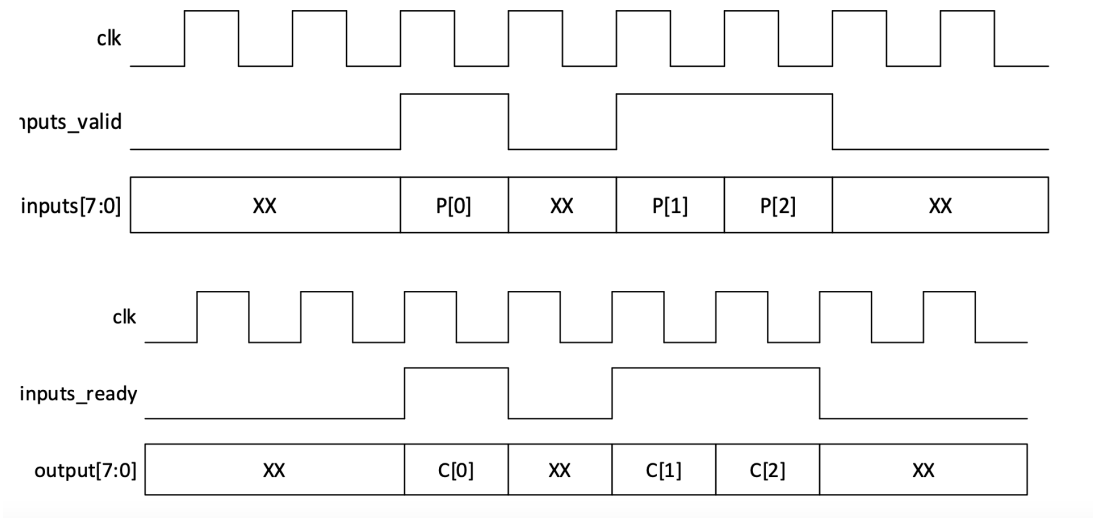

Figure 1.2: Expected waveforms for input and output

keys. At this point Walt would like to send a message to Jesse. Walt encrypts each character of the plaintext, to obtain the ciphertext, using the public key of Jesse and then sends it in encrypted mode. Jesse receives Walt's message and decrypts it using his secret key. In the case he wants to reply to the message, so Jesse does the same thing as Walt, but he uses the public key of Walt, and then sends the message. When Walt receives the message, he decrypts it using his secret key.

Figure 1.2 shows the waveforms required by the project guidelines. The inputs_valid signal will indicate that the circuit inputs are valid and that you can begin operations of one of the three functions. When the operation is completed, the outputs_ready signal will be activated and the result will be given as the output. One of the requirements of the design specification is that each of the three operations must be performed in one clock cycle.

Chapter 2

Block diagram and design choices

2.1 Block diagram

For the implementation of the project, we implemented the three functions (key-pair generation, encryption and decryption) as three different submodules. Instances of the three submodules are placed within the main sae module.

The Figure 2.1 shows the inputs and outputs of the main sae module. Specifically:

- input *clk*, clock signal, all the input ports are sampled on the rising edge of the clock;
- input *rst_n*, low active asynchronous reset signal;
- input 2-bit *mode*, allows the user to select which of the three operations to perform;
- input 8-bit *data_input*, represents the data input provided by the user, that can be either the plaintext or the ciphertext, depending on the mode selected;
- input 8-bit *key_input*, represents the key input provided by the user, that can be either a secret key or a public key, depending on the mode selected;
- input *inputs_valid*, indicates when the inputs are valid and can be sampled by the module;
- output 8-bit *data_output*, represents the output of the module, that can either be the plaintext, the ciphertext or the public key depending on the selected mode;

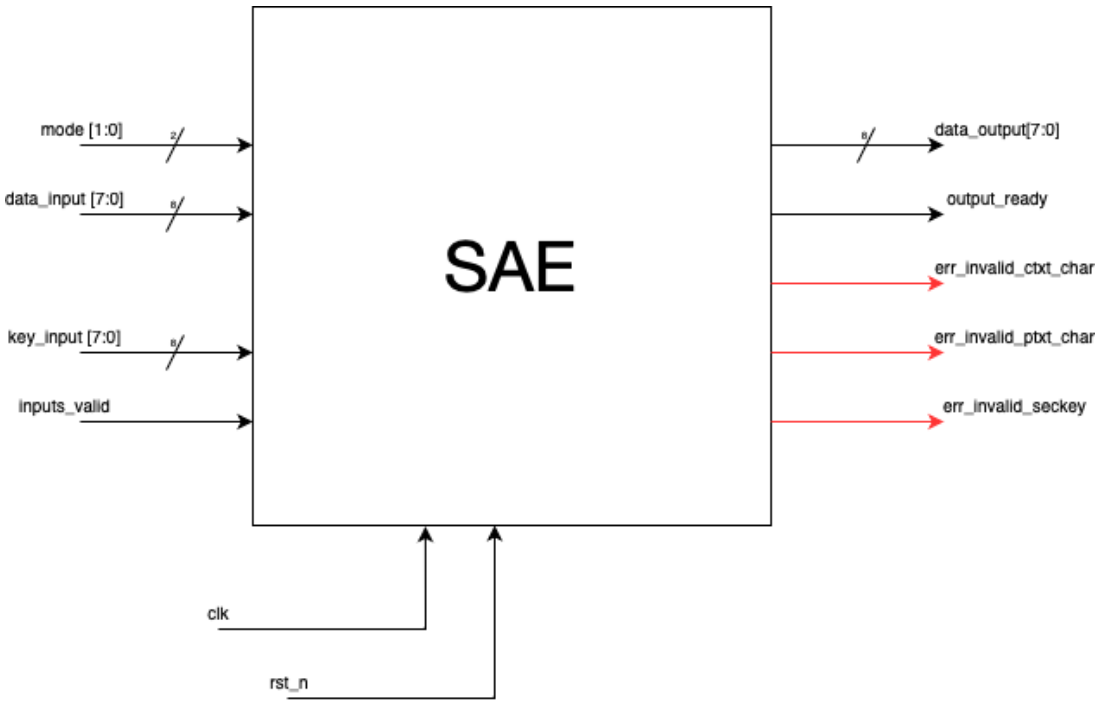

Figure 2.1: Block diagram of SAE

- output *output_ready*, indicates when the output is ready;
- output *err_invalid_ctxt_char*, indicates that the ciphertext is not valid;
- output *err_invalid_ptxt_char*, indicates that the plaintext is not valid;
- output *err_invalid_seckey*, indicates that the secret key is not valid.

Within the SAE module we find the three submodules that each implement one of the operations. Let us examine them in detail.

2.1.1 Key-pair generation submodule

Figure 2.2 shows the block diagram for the key-pair generation submodule. When the input *selection* is worth 1, the circuit is activated and calculates the public key, from the *secret_key* provided as input, through the formula $public\_key = (secret\_key + q) mod p$. The secret key must take a value between 1 and 226 (p-1), otherwise the public key generation operation will not be performed. The result of the operation will be output through *public_key*.

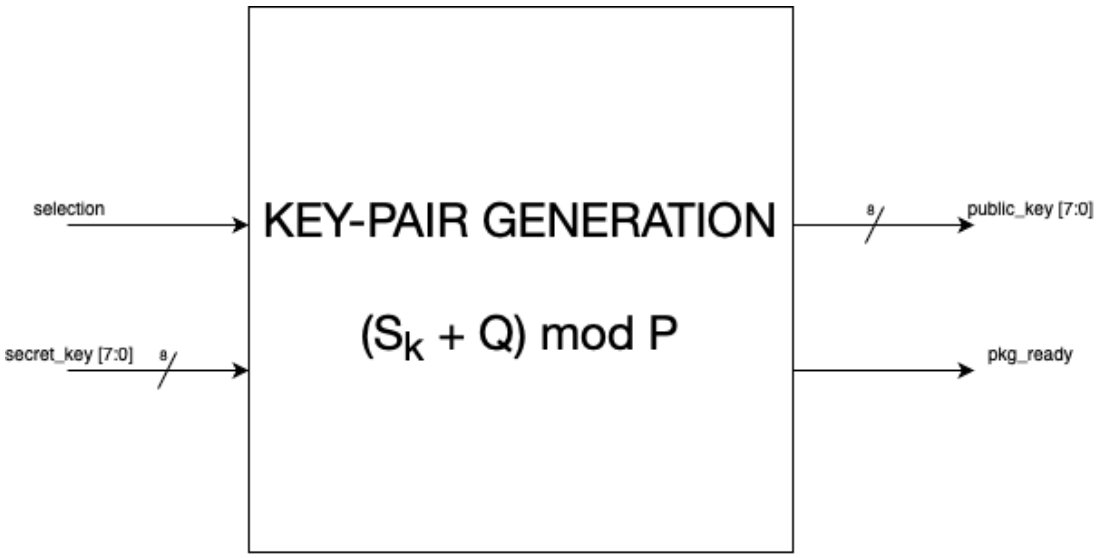

Figure 2.2: Block diagram of the key-pair generation submodule

2.1.2 Encryption submodule

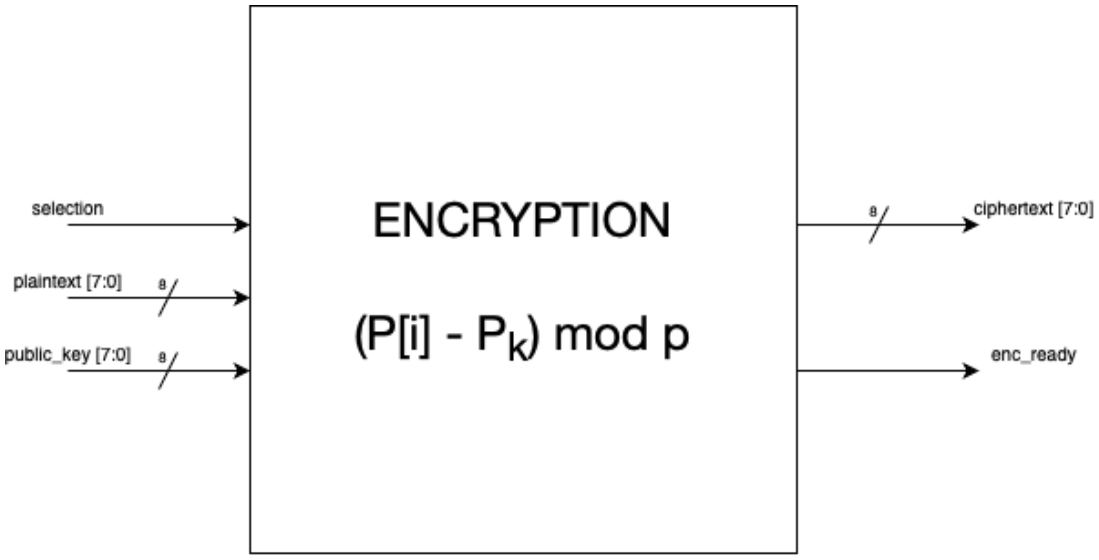

Figure 2.3: Block diagram of the encryption submodule

Figure 2.3 shows the block diagram for the encryption submodule. When the input *selection* is worth 1, the circuit is activated and calculates the ciphertext, from the *plaintext* and the *public_key* provided as inputs, through the formula $ciphertext = (plaintext - public\_key) mod p$. The plaintext must take a decimal value between 97 (the decimal value of the lowercase letter ”a” in ASCII encoding) and 122 (the value of the lowercase letter ”z”), otherwise the encryption

operation will not be performed. The result of the operation will be output through *ciphertext*.

2.1.3 Decryption submodule

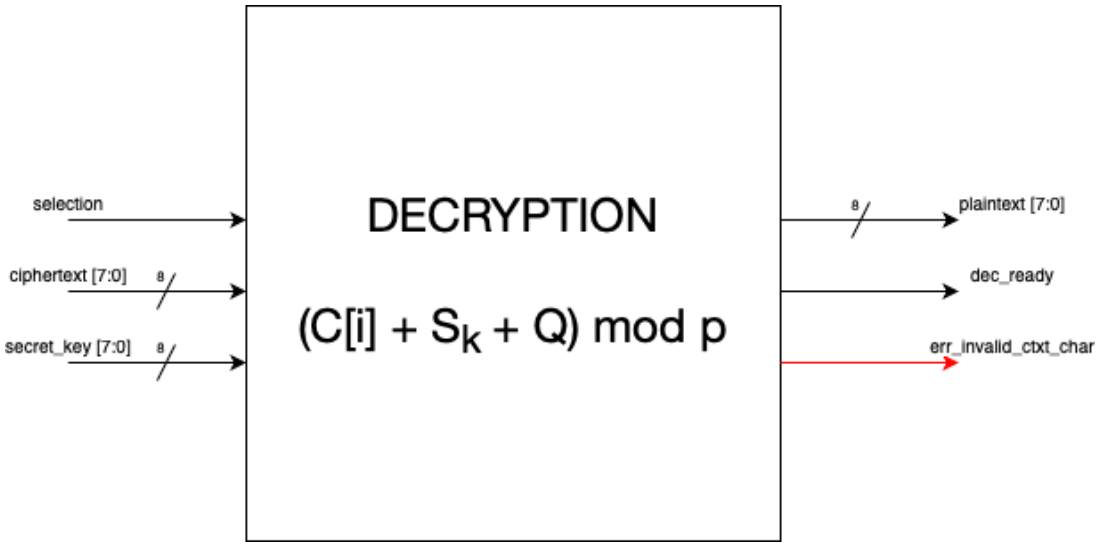

Figure 2.4: Block diagram of the decryption submodule

Figure 2.4 shows the block diagram for the encryption submodule. When the input *selection* is worth 1, the circuit is activated and calculates the plaintext, from the *ciphertext* and the *secret_key* provided as inputs, through the formula $plaintext = (ciphertext + secret\_key + q)modp$. The ciphertext can take any value between 0 and 255, but it is essential that the plaintext calculated by the form has a value between 97 and 122 (the set of lowercase letters), otherwise the error *err_invalid_ctxt_char* will be reported. The result of the operation will be output through *plaintext*.

2.2 Design choices

Figure 2.5 shows the block diagram of the entire SAE module, focusing on the circuitry required to properly route input values to the submodules and to handle the outputs in a manner consistent with the design requirements. The 2-bit encoding of the input *mode* is used to select the corresponding submodule, through a demultiplexer that activates only one of its three output signals. More 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.

In addition, the *mode* signal will be used to select which output to activate in two demultiplexers. The first allows you to handle *data_input*, which can be either a plaintext or a ciphertext depending on the mode indicated. The second one allows to manage *key_input*, which can be a private key or a public key. Finally, the mode signal is also used to check whether the secret key or plaintext violates the requirements, which we recall are secret key not between 1 and 226 and plaintext character that is not a lowercase letter.

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

Because of the requirement to provide the output one clock after entering the input values, our implementation choice was to insert registers only to sample the input values and to sample the values to be provided on the output. By acting in this way, we were able to implement a module that provided the outputs one clock after receiving the inputs.

To use the module, the user will have to set the mode input with the encoding of the desired operation, enter in *key_input* the value of the key (public or private depending on the operation), enter the *data_input* the value of the input data (ciphertext or plaintext depending on the operation), and set the value of *inputs_valid* to 1. After a clock it will receive the output of the operation. The key-pair generation operation requires only the input secret key; in that case the user can enter any value in *data_input* since it will be irrelevant.

The modulo operation, required by all three functions, was implemented through a series of ifs and subtractions. Acting in this way made it possible to implement the operation without using dividers, which would have introduced longer delays.

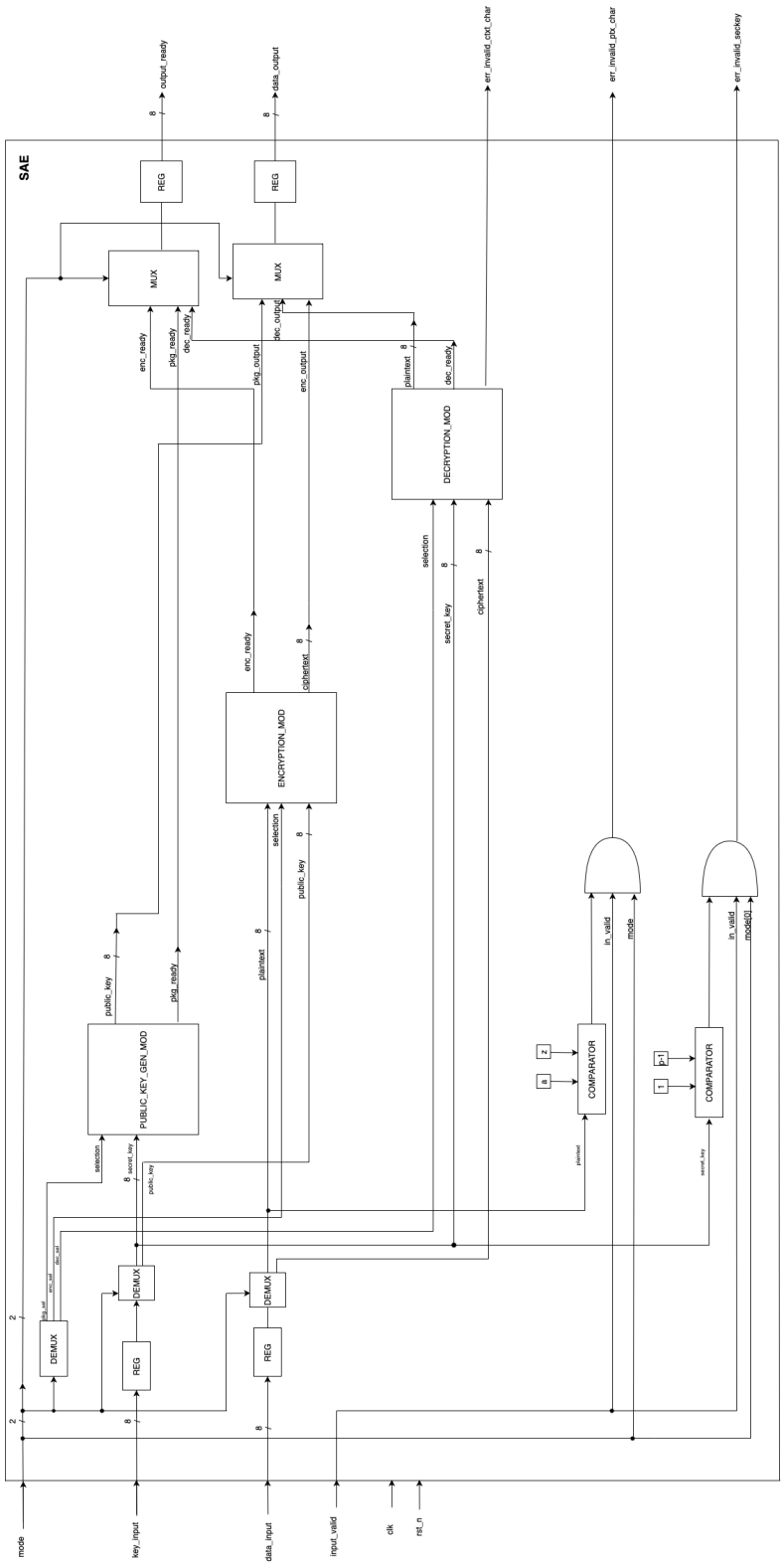

Figure 2.5: Block diagram of the design

Chapter 3

Expected waveforms

Figure 3.1 shows the waveforms during the key-pair generation operation. As can be seen, the circuit receives the following inputs:

- mode = 01
- data_input = 00000000 (could take any value, because it was irrelevant in this operation)
- key_input = 1 (in binary encoding, in the image it is given in decimal value to improve readability)
- inputs_valid = 1

After one clock cycle, *output_ready* goes to 1 and the public key value of 226 is provided as output.

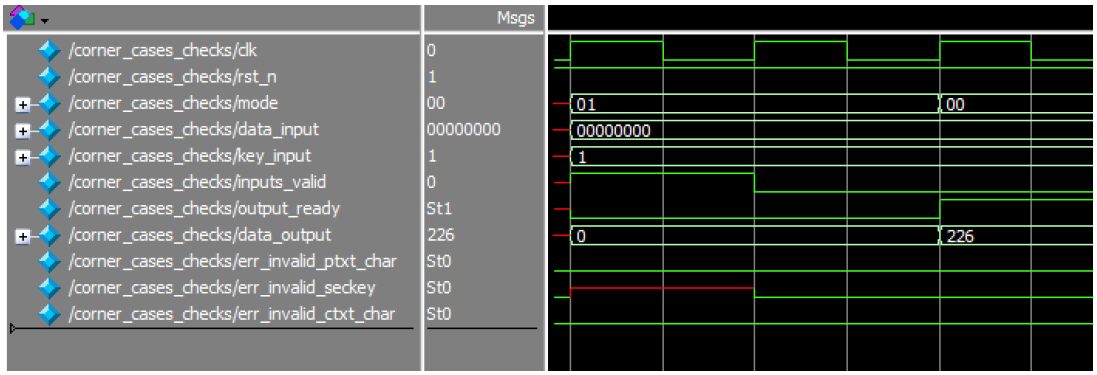

Figure 3.1: Snapshot of the waveform of the key-pair generation

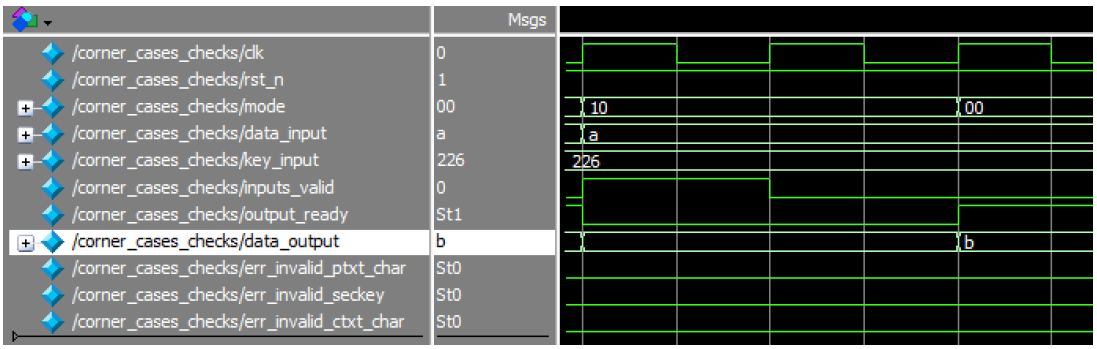

Figure 3.2: Snapshot of the waveform of the encryption

Figure 3.2 shows the waveforms during the encryption operation. As can be seen, the circuit receives the following inputs:

- mode = 10
- data_input = a (in binary encoding)
- key_input = 226
- inputs_valid = 1

After one clock cycle, *output_ready* goes to 1 and the ciphertext value of ”b” is provided as output.

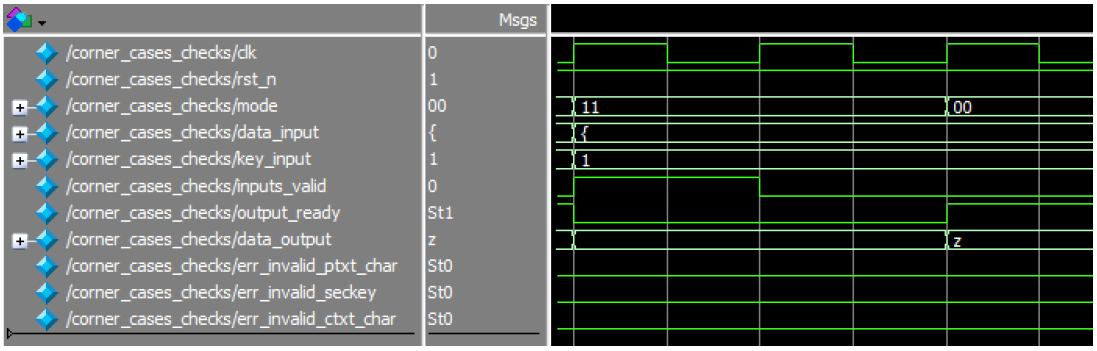

Figure 3.3: Snapshot of the waveform of the decryption

Figure 3.3 shows the waveforms during the decryption operation. As can be seen, the circuit receives the following inputs:

- mode = 11

- data_input = { (in binary encoding)
- key_input = 1
- inputs_valid = 1

After one clock cycle, *output_ready* goes to 1 and the plaintext value of ”z” is provided as output.

Finally, in the figure 3.4 we can appreciate the waveforms for the whole system. The result was obtained by simulating one of the two testbenches developed to test the module. The error reporting operation can also be seen, in case of incorrect secret keys or characters.

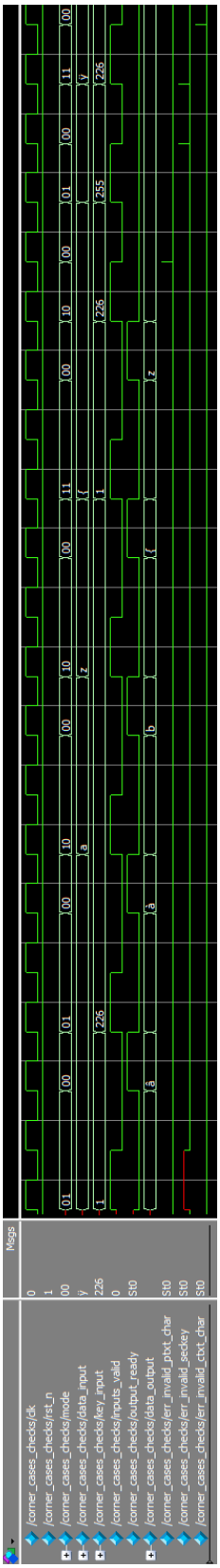

Figure 3.4: Snapshot of the waveform for all the operations

Chapter 4

Testbench

We made two testbenches to test the operation of the circuit. The first one (sae_tb.sv) simulates the communication shown in Figure 1.1 . The second (corner_cases.sv) analyzes all the corner cases and verifies the correct operation and error handling. Let us analyze them in detail.

4.1 sae_tb.sv

The testbench simulates communication between Walt and Jesse, as shown in Figure 1.1. To simulate communication between the two parties, two instances of the SAE module are defined in the testbench. In addition, the mechanism of sending the outputs to the other party is simulated by writing the results inside some text files. Specifically:

- Walt's public key will be saved in the *publickey_j.txt* file, to simulate sending the key to Jesse.
- Jesse's public key will be saved in the *publickey_w.txt* file, to simulate sending the key to Walt.
- The ciphertext produced by Walt by encrypting his own plaintext will be saved in the *ciphertext_j.txt* file, to simulate sending the ciphertext to Jesse.

All text files storing inputs and results are contained within the "tv" folder.

The test initially involves generating the public keys of both parties through the key-pair generation operation as shown in Figure 4.1. To perform the operation, the file containing

```
//-----------------------------------------------------------------
// WALT GENERATES THE PUBLIC KEY
//-----------------------------------------------------------------

    @(posedge clk);
    FILE = $fopen("tv/privatekey_w.txt", "rb");
    if (FILE) begin
       $display("File privatekey_w was opened successfully : %0d", FILE);
    end
    else begin
       $display("File privatekey_w was NOT opened successfully : %0d", FILE);
       $finish;
    end
    if($fscanf(FILE, "%b", key_input_w) == 1)begin
       $display("Walt's private key was successfully loaded");
    end
    else begin
       $display("Walt's private key was NOT loaded correctly");
       $finish;
    end
    $fclose(FILE);
    mode_w = 2'b01;
    data_input_w = 8'd0;
    inputs_valid_w = 1'b1;
    @(posedge clk);
    inputs_valid_w = 1'b0;
    @(posedge clk);
    #3 while (output_ready_w != 1'b1) begin
       if(err_invalid_seckey_w == 1'b1) begin
          $display("Walt's secret key has an invalid value");
          $finish;
       end
       $display("Walt's public key is NOT yet ready");
    end
    $display("Walt's public key was generated");
    FILE = $fopen("tv/publickey_j.txt", "wb");
    if (FILE) begin
       $display("File publickey_j was opened successfully : %0d", FILE);
    end
    else begin
       $display("File publickey_j was NOT opened successfully : %0d", FILE);
       $finish;
    end

    $fdisplay(FILE, "%b", data_output_w);
    $fclose(FILE);
    mode_w = 2'b00;
    @(posedge clk);
```

```
//-----------------------------------------------------------------
// JESSE GENERATES THE PUBLIC KEY
//-----------------------------------------------------------------

    FILE = $fopen("tv/privatekey_j.txt", "rb");
    if (FILE) begin
       $display("File privatekey_j was opened successfully : %0d", FILE);
    end
    else begin
       $display("File privatekey_j was NOT opened successfully : %0d", FILE);
       $finish;
    end
    if($fscanf(FILE, "%b", key_input_j) == 1) begin
       $display("Jesse's private key was successfully loaded");
    end
    else begin
       $display("Jesse's private key was NOT loaded correctly");
       $finish;
    end
    $fclose(FILE);
    mode_j = 2'b01;
    data_input_j = 8'd0;
    inputs_valid_j = 1'b1;
    @(posedge clk);
    inputs_valid_j = 1'b0;
    @(posedge clk);
    #3 while (output_ready_j != 1'b1) begin
       if(err_invalid_seckey_j == 1'b1) begin
          $display("Jesse's secret key has an invalid value");
          $finish;
       end
       $display("Jesse's public key is NOT yet ready");
    end
    $display("Jesse's public key was generated");
    FILE = $fopen("tv/publickey_w.txt", "wb");
    if (FILE) begin
       $display("File publickey_w was opened successfully : %0d", FILE);
    end
    else begin
       $display("File publickey_w was NOT opened successfully : %0d", FILE);
       $finish;
    end
    $fdisplay(FILE, "%b", data_output_j);
    $fclose(FILE);
    mode_j = 2'b00;
    @(posedge clk);
```

Figure 4.1: Key-pair generation in testbench sae_tb.sv

the secret key is opened, the key value is extracted and given as input to the SAE module, key-pair generation mode is selected, and execution is started. Then the calculation is waited for completion, in the meantime it is verified that the secret key does not generate an error due to its value. Once the calculation is completed, the value of the public key is saved in the corresponding text file. The operation is performed by both instances of the SAE module.

In Figure 4.2 we can see the next two phases of the test. The first involves Walt encrypting his plaintext and sending it to Jesse. This is done by opening the file containing the plaintext, extracting one character at a time and giving it as input to the SAE module. When the encryption operation is complete, the result is placed in a queue. When there are no more characters to encrypt, the values within the queue will be written to the file that will contain the ciphertext. The decryption operation performed by Jesse involves the same actions, changing the

```
//----------------------------------------------------------------------
// WALT ENCRYPTS THE PLAINTEXT
//----------------------------------------------------------------------
FILE = $fopen("tv/publickey_w.txt", "rb");
if (FILE)  begin
   $display("File publickey_w was opened successfully : %0d", FILE);
end
else begin
   $display("File publickey_w was NOT opened successfully : %0d", FILE);
   $finish;
end
if($fscanf(FILE, "%b", key_input_w) == 1) begin
   $display("Jesse's public key was successfully loaded");
end
else begin
   $display("Jesse's public key was NOT loaded correctly");
   $finish;
end
$fclose(FILE);
FILE = $fopen("tv/plaintext_w.txt", "r");
if (FILE)  begin
   $display("File plaintext_w was opened successfully : %0d", FILE);
end
else begin
   $display("File plaintext_w was NOT opened successfully : %0d", FILE);
   $finish;
end
while($fscanf(FILE, "%c", char_w) == 1) begin
   data_input_w = int'(char_w);
   mode_w = 2'b10;
   inputs_valid_w = 1'b1;
   @(posedge clk);
   inputs_valid_w = 1'b0;
   @(posedge clk);
   #3 while(output_ready_w != 1'b1) begin
      if(err_invalid_ptxt_char_w == 1'b1) begin
         $display("An invalid character was inserted in the plaintext");
         $finish;
      end
   end
   CTXT_W.push_back(data_output_w);
end
$fclose(FILE);
FILE = $fopen("tv/ciphertext_j.txt", "w");
if (FILE) begin
   $display("File ciphertext_j was opened successfully : %0d", FILE);
end
else begin
   $display("File ciphertext_j was NOT opened successfully : %0d", FILE);
   $finish;
end
foreach(CTXT_W[i]) begin
   $fwrite(FILE, "%c", CTXT_W[i]);
end
$fclose(FILE);
mode_w = 2'b00;
@(posedge clk);
```

```
//----------------------------------------------------------------------
// JESSE DECRYPTS THE CIPHERTEXT
//----------------------------------------------------------------------
FILE = $fopen("tv/privatekey_j.txt", "rb");
if (FILE) begin
   $display("File privatekey_j was opened successfully : %0d", FILE);
end
else begin
   $display("File privatekey_j was NOT opened successfully : %0d", FILE);
   $finish;
end
if($fscanf(FILE, "%b", key_input_j) == 1) begin
   $display("Jesse's private key was successfully loaded");
end
else begin
   $display("Jesse's private key was NOT loaded correctly");
   $finish;
end
$fclose(FILE);
FILE = $fopen("tv/ciphertext_j.txt", "r");
if (FILE) begin
   $display("File ciphertext_j was opened successfully : %0d", FILE);
end
else begin
   $display("File ciphertext_j was NOT opened successfully : %0d", FILE);
   $finish;
end
while($fscanf(FILE, "%c", char_j) == 1) begin
   data_input_j = int'(char_j);
   mode_j = 2'b11;
   inputs_valid_j = 1'b1;
   @(posedge clk);
   inputs_valid_j = 1'b0;
   @(posedge clk);
   #3 while(output_ready_j != 1'b1) begin
      if(err_invalid_ctxt_char_j == 1'b1) begin
         $display("An invalid character was inserted in the ciphertext");
         $finish;
      end
   end
   PTXT_J.push_back(data_output_j);
end
$fclose(FILE);
FILE = $fopen("tv/plaintext_j.txt", "w");
if (FILE) begin
   $display("File plaintext_j was opened successfully : %0d", FILE);
end
else begin
   $display("File plaintext_j was NOT opened successfully : %0d", FILE);
   $finish;
end
foreach(PTXT_J[i]) begin
   $fwrite(FILE, "%c", PTXT_J[i]);
end
$fclose(FILE);
PTXT_J.delete();
mode_j = 2'b00;
@(posedge clk);
```

Figure 4.2: Encryption and decryption in testbench sae_tb.sv

```
//------------------------------------------------------------------
// I CHECK THAT THE ORIGINAL PLAINTEXT AND THE PLAINTEXT OBTAINED BY DECRYPTION MATCH
//------------------------------------------------------------------

FILE = $fopen("tv/plaintext_w.txt", "r");
if (FILE) begin
   $display("File plaintext_w was opened successfully : %0d", FILE);
end
else begin
   $display("File plaintext_w was NOT opened successfully : %0d", FILE);
   $finish;
end
while($fscanf(FILE, "%c", char_w2) == 1) begin
   PTXT_W.push_back(int'(char_w2));
end
$fclose(FILE);
FILE = $fopen("tv/plaintext_j.txt", "r");
if (FILE) begin
   $display("File plaintext_j was opened successfully : %0d", FILE);
end
else begin
   $display("File plaintext_j was NOT opened successfully : %0d", FILE);
   $finish;
end
while($fscanf(FILE, "%c", char_j2) == 1) begin
   PTXT_J.push_back(int'(char_j2));
end
$fclose(FILE);
if(PTXT_J == PTXT_W) begin
   $display("Plaintexts MATCH!");
end
else begin
   $display("Plaintexts NOT match");
end
```

Figure 4.3: Final check in testbench sae_tb.sv

mode and the file from which to take the ciphertext to be decrypted. Once the plaintext has been computed and saved within the corresponding file, the queue containing it is cleaned to avoid leaving the information accessible.

Finally, Figure 4.3 shows the check made between the original plaintext and the calculated plaintext, so as to verify that they match.

For the generation of the testing vectors, a Python script was made that randomly generates secret keys for Walt and Jesse and extracts a line from a dictionary as a plaintext. The dictionary includes the 10000 long words (9+ characters) most searched on Google. The script

code is shown in Figure 4.4.

```
import os
import secrets

def reset_files():
    current_folder = os.getcwd()
    path_to_tv = current_folder + '/modelsim/tv'
    for filename in os.listdir(path_to_tv):
        if filename != 'dictionary.txt':
            with open('./modelsim/tv/' + filename, 'w') as file:
                file.write('')

def main():

    reset_files()

    secretsGenerator = secrets.SystemRandom()

    waltKey = bytes(f'{secretsGenerator.randrange(1, 226):08b}', 'ascii')
    with open('./modelsim/tv/privatekey_w.txt', 'wb') as file:
        file.write(waltKey)


    jesseKey = bytes(f'{secretsGenerator.randrange(1, 226):08b}', 'ascii')
    with open('./modelsim/tv/privatekey_j.txt', 'wb') as file:
        file.write(jesseKey)

    with open('./modelsim/tv/dictionary.txt', 'r') as file:
        all_the_lines = file.readlines()
        line_to_read = secretsGenerator.randrange(1, len(all_the_lines))
        plaintext = all_the_lines[line_to_read - 1]

    with open('./modelsim/tv/plaintext_w.txt', 'w') as file:
        file.write(plaintext)

if __name__ == '__main__':
    main()
```

Figure 4.4: generate_tv.py

4.2 corner_cases.sv

The testbench analyzes the limit cases of the module, namely the letters ”a” and ”z” as plaintext characters and the values 1 and 226 as secret key. Tests are also performed to evaluate the operation of decryption and error handling. A total of 8 test cases are performed, and the waveform is as shown in Figure 3.4.

Chapter 5

Implementation of RTL design on FPGA and results

Analysis, synthesis and fitting was performed using Quartus 21.1, using FPGA 5CGXFC9D6F27C7 of the Cyclone V family by Intel. Figure 5.1 represent the RTL (Register Transfer Level) design of the SAE.

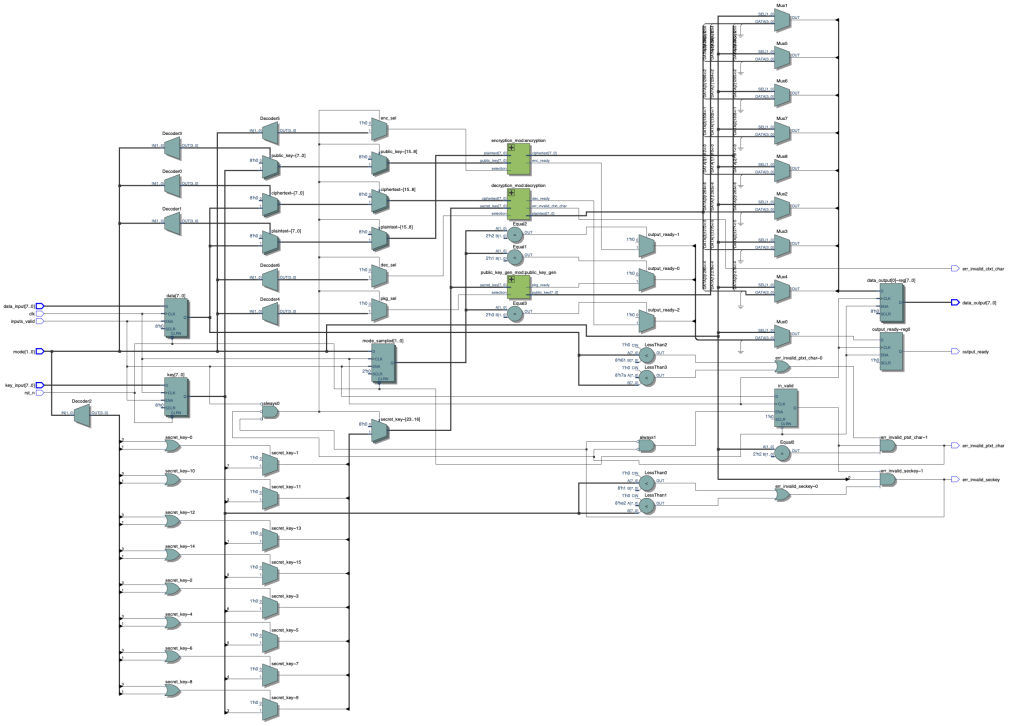

Figure 5.1: RTL view

Fitter Summary

<<Filter>>

Fitter Status	Successful - Thu Jan 5 19:04:12 2023
Quartus Prime Version	21.1.1 Build 850 06/23/2022 SJ Lite Edition
Revision Name	sae
Top-level Entity Name	sae
Family	Cyclone V
Device	5CGXFC9D6F27C7
Timing Models	Final
Logic utilization (in ALMs)	103 / 113,560 (< 1 %)
Total registers	33
Total pins	1 / 378 (< 1 %)
Total virtual pins	32
Total block memory bits	0 / 12,492,800 (0 %)
Total RAM Blocks	0 / 1,220 (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 5.2: Fitter results

Figure 5.2 shows the results of the analysis, synthesis and fitting operations performed by Quartus. We can see that the circuit occupies a very small portion of the FPGA (<1%). This result makes us realize that this FPGA is not suitable for accommodating this single circuit, as we would be wasting a large amount of resources. In that case only one physical pin, the clock pin, would be used, while all other input and output pins would be virtual. Assigning virtual pins would also improve the frequency of the circuit (more details in the next chapter).

Chapter 6

Static Timing Analysis

The first step to perform the Static Time Analysis was to configure the constr.sdc file as in Figure 6.1.

The first line of the code creates a clock object with a period of 10ns, and links it with the input signal clk of the SAE module. The rst_n signal is decoupled from the clock because rst_n has to be asynchronous from the clock signal. Commands set_input_delay and set_output_delay define port delay for inputs and outputs. For both input and output ports there is a minimum and maximum delay. The rule of thumb is that the minimum delay is 10% of the clock period and the maximum delay is 20%.

Setting a clock period of 10ns consequently means that the frequency to be achieved is 100 MHz. Timing Analysis shows that the maximum frequency is 54.76 MHz for the Slow Test at 85° as shown in Figure 6.2. The Slow Test at 0° reaches a frequency of 54.51 MHz.

The main reason why the target frequency cannot be achieved is related to the constraint

```
create_clock -name clk -period 10 [get_ports clk]

set_false_path -from [get_ports rst_n] -to [get_clocks clk]

set_input_delay  -min 1 -clock [get_clocks clk] [all_inputs ]
set_input_delay  -max 2 -clock [get_clocks clk] [all_inputs ]
set_output_delay -min 1 -clock [get_clocks clk] [all_outputs]
set_output_delay -max 2 -clock [get_clocks clk] [all_outputs]
```

Figure 6.1: constr.sdc

Slow 1100mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	54.76 MHz	54.76 MHz	clk	

Figure 6.2: Non-optimized Max Frequency

	Status	From	To	Assignment Name	Value	Enabled	Entity
1	Ok		err_in...t_char	Virtual Pin	On	Yes	sae
2	Ok		err_in...seckey	Virtual Pin	On	Yes	sae
3	Ok		inputs_valid	Virtual Pin	On	Yes	sae
4	Ok		output_ready	Virtual Pin	On	Yes	sae
5	Ok		rst_n	Virtual Pin	On	Yes	sae
6	Ok		data_input	Virtual Pin	On	Yes	sae
7	Ok		data_output	Virtual Pin	On	Yes	sae
8	Ok		key_input	Virtual Pin	On	Yes	sae
9	Ok		mode	Virtual Pin	On	Yes	sae
10	Ok		err_in...t_char	Virtual Pin	On	Yes	sae
11		...	<<new>>	<<new>>			

Figure 6.3: Virtual pins setup

of computing the output in a single clock cycle. Because of this constraint, it is not possible to insert intermediate registers between those used to sample the inputs and those used to provide the outputs, otherwise an extra clock would be needed to obtain the results. The longest combinational path is from the date register, where the data_input value is sampled, to the err_invalid_ctxt_char output.

The only way to increase the frequency is through the use of virtual pins, as described in the previous chapter. In Figure 6.3 we can see the setup of the virtual pins.

Through this modification we achieve slight improvements in frequency in the Slow Test, both at 0° and 85°. As we can see from Figure 6.4 we get a frequency of 60.93 MHz at 85° and 60.09 MHz at 0°, still not enough to meet the timing requirements. The worst case is at 0°, with the frequency equal to 60.09 MHz, and this is the one that should be taken as a reference to be sure that the module works under all circumstances.

Slow 1100mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	60.93 MHz	60.93 MHz	clk	

Slow 1100mV 0C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	60.09 MHz	60.09 MHz	clk	

Figure 6.4: Optimized Frequencies