UNIVERSITÀ DI PISA

DEPARTMENT OF COMPUTER ENGINEERING

MASTER DEGREE COURSE IN CYBERSECURITY

PROJECT REPORT

SUBSTITUTION_TABLE_DEC MODULE

STUDENT:

Alessio Breglia

Professor

Sergio Saponara

Referent

Luca Crocetti

Academic Year 2022/2023

SPECIFICATION ANALYSIS

In this chapter I will briefly introduce the overall system specifications.

The *substitution_table_dec* module represents an hardware implementation of a substitution table that, given the ciphertext of 16 bit, will return the respective plaintext of 8 bit. The table is initialized by a key of 12 characters, represeting the possible characters of the ciphertext. Ciphertext characters are replaced with the corresponding character in the table in row-column order.

The table is structured in this way:

////////////	K[10]	K[8]	K[1]	K[2]	K[4]	K[7]
K[0]	a/A	b/B	c/C	d/D	e/E	f/F
K[11]	g/G	h/H	i/I	j/J	k/K	l/L
K[9]	m/M	n/N	o/O	p/P	q/Q	r/R
K[3]	s/S	t/t	u/U	v/V	w/W	x/X
K[5]	y/Y	z/Z	0	1	2	3
K[6]	4	5	6	7	8	9

The module decrypt one ciphertext symbol per clock cycle (a ciphertext symbol is represented by a pair of characters), and generate one plaintext character per clock cycle. Each character of ciphertext has to be any 8-bit ASCII character of alphabet or digits. The key has to contain only 8-bit ASCII characters of alphabet or digits with no repetitions. The module has 3 fundamental flag, implemented with 1-bit signal:

- **Ctxt_valid**: 1-bit signal that assert if the characters of the ciphertext are valid and stable: 1 if they are, 0 otherwise.
- **Upper_lower**: 1-bit signal that indicates if the corresponding char in the plaintext is uppercase or lowercase: 1 for uppercase, 0 for lowercase.
- **Ptxt_valid**: 1-bit signal that assert if the character of plaintext is valid and stable: 1 if it is, 0 otherwise.

MODEL

In this chapter I am going to describe the model through C code. To demonstrate that the C function correctly describes the model, I also implemented a small testbench to call the function via DPI.

The C function that perform the substitution is called **substitution** and it has the following parameters:

- const uint8_t **key**: the key used for substitution.
- uint8_t **plaintext**: the resulting plaintext.
- const uint8_t **ciphertext1**: the first character of ciphertext.
- const uint8_t **ciphertext2**: the second character of ciphertext.
- const bool **upper_lower**: store the upper_lower flag.
- uint8_t **error_key**: the flag that assert if key is correct.
- uint8_t **error_ciphertext**: the flag that assert if ciphertext is correct.

The table is implemented with two 2d arrays: one for uppercase and one for lowercase (below is reported only lowercase array). Based on the *upper_lower* value, it can be chosen one or another to perform the decryption. These array are called *combinations*:

```
{0, 10, 'a'}, {0, 8, 'b'}, {0, 1, 'c'}, {0, 2, 'd'}, {0, 4, 'e'}, {0, 7, 'f'},
{11, 10, 'g'}, {11, 8, 'h'}, {11, 1, 'i'}, {11, 2, 'j'}, {11, 4, 'k'}, {11, 7, 'l'},
{9, 10, 'm'}, {9, 8, 'n'}, {9, 1, 'o'}, {9, 2, 'p'}, {9, 4, 'q'}, {9, 7, 'r'},
{3, 10, 's'}, {3, 8, 't'}, {3, 1, 'u'}, {3, 2, 'v'}, {3, 4, 'w'}, {3, 7, 'x'},
{5, 10, 'y'}, {5, 8, 'z'}, {5, 1, '0'}, {5, 2, '1'}, {5, 4, '2'}, {5, 7, '3'},
{6, 10, '4'}, {6, 8, '5'}, {6, 1, '6'}, {6, 2, '7'}, {6, 4, '8'}, {6, 7, '9'}
```

The algorithm that perform the substitution is implemented by the following code:

```
for (i = 0; i < 36; i++) {
if (key[combinations_lower[i][0]] == ciphertext1 && key[combinations_lower[i][1]] == ciphertext2) {
plaintext[0] = combinations_lower[i][2];  // Assign the corresponding plaintext value
break;  // Exit the loop once a match is found
}
}
```

The code iterate through the 2d array and assign the corresponding plaintext when it find the right couple of ciphertext character.

Finally, I implemented the functions that verify if ciphertext and key are correct. Below it is reported the function that verify if the key is correct and compliant with the specifications:

```
bool found = false;
uint8_t i = 0x00;
uint8_t j = 0x00;
// loop to check if key is valid
for (i = 0; i < 12; i++) {
if((!(key[i] >= 'a' && key[i] <= 'z') ||
(key[i] >= 'A' && key[i] <= 'Z') ||
(key[i] >= '0' && key[i] <= '9'))){
found = true; // not admitted character found
break;
}
```

```
for (j = i + 1; j < 12; j++) {
if (key[i] == key[j]) {
found = true; // repeated character found
break;
}
}
}
```

The function that verify if ciphertext is correct and respect the specifications is the following:

```
if((!(ciphertext1 >= 'a' && ciphertext1 <= 'z') ||
(ciphertext1 >= 'A' && ciphertext1 <= 'Z') ||
(ciphertext1 >= '0' && ciphertext1 <= '9')) &&
!((ciphertext2 >= 'a' && ciphertext2 <= 'z') ||
(ciphertext2 >= 'A' && ciphertext2 <= 'Z') ||
(ciphertext2 >= '0' && ciphertext2 <= '9'))){
found = 1;
}
if(found) error_ciphertext = 1; // invalid ciphertext
```

For what concern the testbench, i called the function *substitution* via DPI:

```
import "DPI-C" function void substitution(byte key[12], byte plaintext[1], byte ciphertext1, byte ciphertext2, bit upper_lower, byte error_key[1], byte error_ciphertext[1]);
```

After that, I defined some testbench to see if the module and the model return the same output given the same input vectors. The results are described in the testbench section.

DESIGN CHOICES AND BLOCK DIAGRAMS

In this chapter, I am going to discuss the design choices applied. Then, I will show as the code has been turned into hardware by Quartus, briefly showing the most important RTL diagrams and the match between them and some snippet of code.

DESIGN CHOICES

In my implementation, there are present 4 main modules:

1. **InputLoading**: this module loads the ciphertext character pair and the related signals, ctxt_valid and upper_lower, into a set of input registers. It have 5 inputs and 3 outputs:
   I. Input:
      - clk: the clock.
      - rst_n: reset signal.
      - ciphertext: the pair of the ciphertext characters (8 + 8 bits).
      - ctxt_valid: if characters of the ciphertext are valid.
      - upper_lower: specify uppercase or lowercase.
   II. Output:
      - ciphertext_reg: registered version of ciphertext.
      - ctxt_valid_reg: registered version of ctxt_valid signal.
      - upper_lower reg: registered version of upper_lower flag.
2. **KeyLoading**: this module loads the characters of the key one byte at a time; 12 cycles are required to load the complete key. For every loaded byte, a validation signal (key_byte_val) must be asserted and the position of the byte in the key must be specified (byte_pos). It has 5 input and 2 output:
   I. Input:
      - clk: the clock.
      - rst_n: reset signal.
      - key_byte: one of the key bytes.
      - byte_pos: position of the byte in the key.
      - key_byte_val: 1-bit validation signal that assert if I can start loading the key byte (if it's value is 1) or not (if it's value is 0).
   II. Output:
      - key: the 12 characters of the key.
      - key_valid: 1-bit flag that asserts if the key is well-formed and does not contain repetitions: values 1 if the key is correct, 0 otherwise.
3. **SubstitutionTableDecryption**: this module implements the substitution table. The substitution function, given the two characters of the ciphertext, returns the decripted character of the plaintext. This module has 3 inputs and one output:
   I. Input:
      - key: the key used to initialized the table.
      - ciphertext: the characters of the ciphertext.
      - upper_lower: specify uppercase or lowercase.
   II. Output:
      - plaintext: the character of the plaintext.
4. **CiphertextDecryptor**: this module implement the decryption algorithm. In this module we're loading the previous modules. It have 8 input and 4 output:
   I. Input:
      - clk: the clock of the module.

- rst_n: the reset signal.
- key_byte: one of the key bytes.
- byte_pos: position of the key byte.
- key_byte_val: the 1-bit validation signal that assert if I can start loading the key.
- ciphertext: the characters of the ciphertext.
- upper_lower: specify uppercase or lowercase.
- ctxt_valid: if characters of the ciphertext are valid and stable.

II. Output:

- ptxt_ready: if plaintext character is valid and stable.
- plaintext: the character of the plaintext.
- error_flag_key: 1-bit signal that assert if the key is not correct (are set to 1 if the key has repetitions or not valid character, 0 otherwise).
- error_flag_ciphertext: 1-bit signal that assert if the ciphertext is not correct (are set to 1 if the ciphertext has not valid character, 0 otherwise).

I also defined two useful function:

1. **isValidCharacter**: this is the function that takes the pair of ciphertext characters and verifies if they are valid. It will return 1 if the character is valid, 0 otherwise.
2. **isKeyValid**: this is the function that verify if the key has only admitted characters and no repetitions. It will respond 1 if the characters are valid, 0 otherwise.

The pair of ciphertext characters, the *ctxt_signal* and the *upper_lower* flag are all stored in dedicated registers to avoid long combinational paths, that can increase the delay of the circuit. Also, the registers serve to avoid concatenation of internal delays in the FPGA. This is the job of the *InputLoading* module.

The key is loaded character by character to limit the number of required I/O pins. The key will be loaded in 12 clock with a validation signal *key_byte_val* that indicates when a character is available to be loaded. The byte_pos signal works as an index, which indicates the position in the key where the received byte must be loaded. Thus, for example, with *key_byte_val* = 1, key_byte = ”a” and byte_pos = 5, the ASCII code of “a” will be loaded as the sixth byte of the key. This is the job of the *KeyLoading* module.

The output plaintext is not sent directly to the pins but is stored in the register *decrypted* to avoid, also in this case, long combinational delay paths.

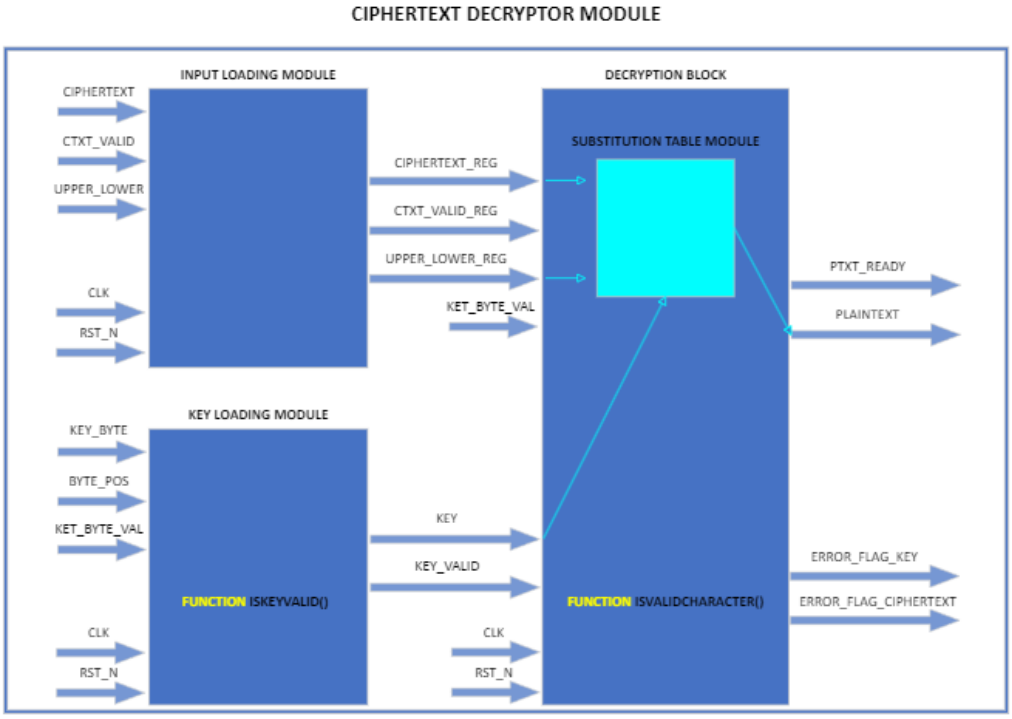
For the key there are have two kind of signal: the *key_valid* is an internal signal that indicates if the key is valid and compliant with the requirements; the *error_flag_key* is an output signal that indicates if the key is not valid. I could do it all with a single signal, but for readability of waveforms and greater understanding, I modeled this behavior with two separate signals.

For the *IsKeyValid* function, I make it purely combinatorial. It verifies in a completely parallel way if the characters in the key are correct and if there are no replicas. It is possible to perform these functions sequentially, for example, one verification per clock cycle, but I opted to make it combinatorial because I wanted low latency: as soon as the key is loaded, in one cycle you know whether it is correct or not.

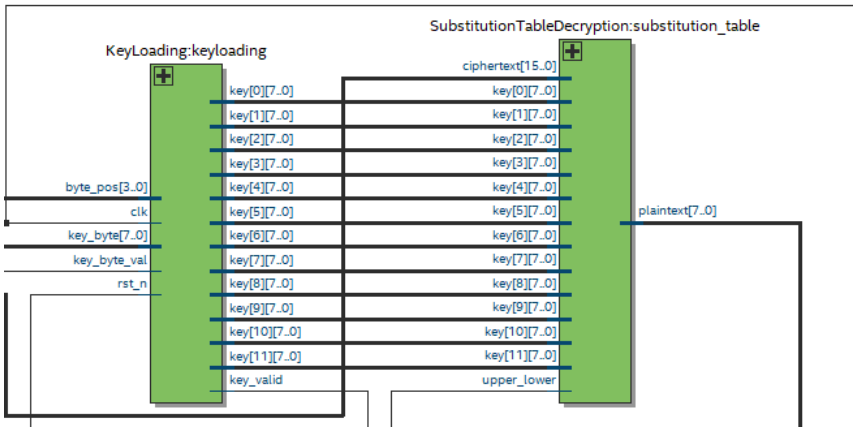
Also the *isValidCharacter* function verify if the pair of ciphertext characters is compliant with the requirements in a parallel way, as it is possible to notice in the block diagrams section.

BLOCK DIAGRAMS

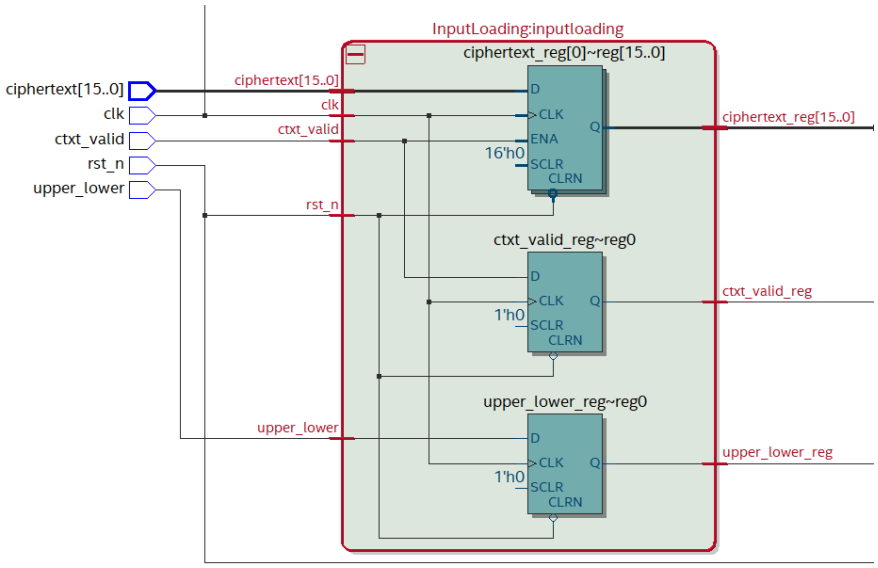
First of all, an high-level diagram of how the various modules interact with each other is specified below.



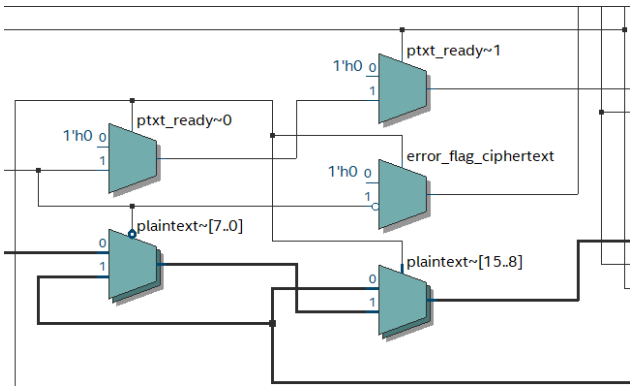
Now I analyze it in a more detailed way using Quartus. I report only the most significant parts of the circuit.



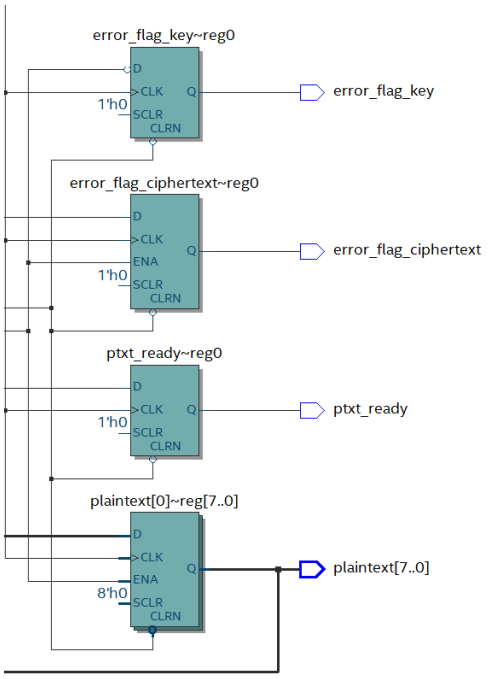
This part shows how the bytes of the key and the pair of ciphertext characters are passed to the *substitution_table* module. Notice that the substitution_table module received the key byte to byte. It respect the high-level schema defined before and the design choices explained in the previous section.



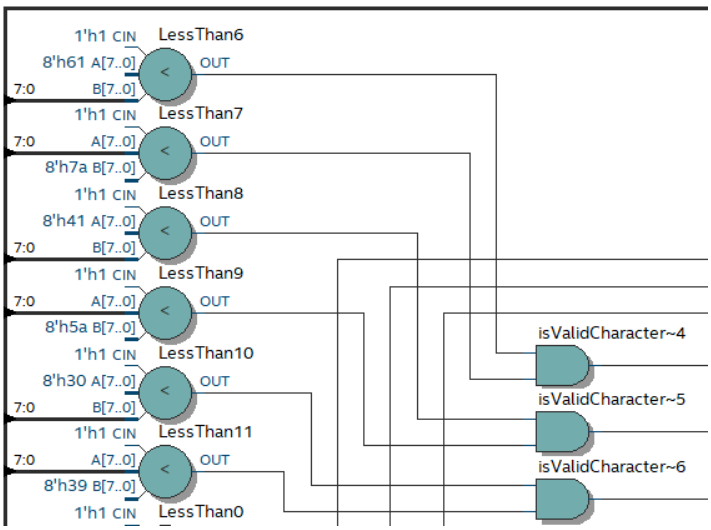
This part show how the *InputLoading* save *ciphertext*, *ctxt_valid* and *upper_lower* in different registers. The structure perfectly match with the behaviour specified before: it is possible to notice that the module has 16 register to save the 16 bit of the ciphertext pair, and 1 register for each of the 2 output signals (*ctxt_valid* and *upper_lower*).



This part show how are prepared the *ptxt_ready*, the *error_flag* relative to ciphertext and the plaintext.



This part show how the circuit save the plaintext, the error flags and the ptxt_ready signal. For the plaintext the module has 8 registers to save the 8 bit of the character. For the others, 1 register is enough. This part too is coherent with the high-level schema explained before.



This part shows how the *isValidCharacter* function was implemented at the hardware level. It is possible to notice, in the left part, how the body of the function is implemented, while in the right part it is possible to observe how the if check was implemented (it is an AND since both characters of the ciphertext must be valid). This is coherent with the design choices discussed in the previous section: verification occurs in a parallel manner. For completeness, below I put the snippets of the associated codes.

begin

isValidCharacter = ((input_char >= "a" && input_char <= "z") ||

(input_char >= "A" && input_char <= "Z") ||

(input_char >= "0" && input_char <= "9"));

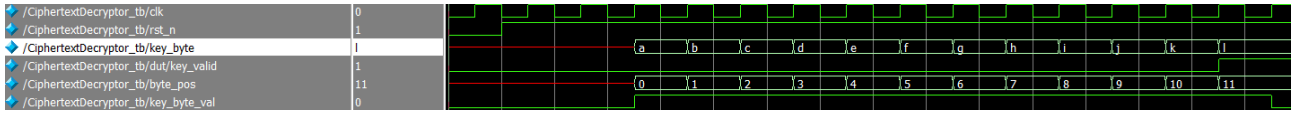
end

if (!(isValidCharacter(ciphertext_reg[15:8]) && isValidCharacter(ciphertext_reg[7:0])))

WAVEFORM DISCUSSION

In this chapter I will briefly discuss the most relevant pieces of waveforms made in Modelsim.

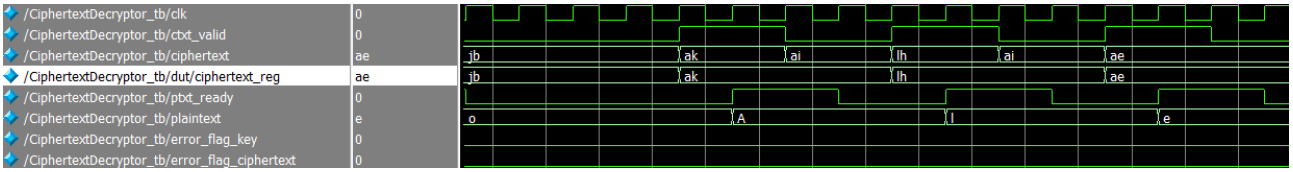
- **Loading valid key**: the waveform below show the correct evolution of the loading of a valid key. It is possible to notice that the *key_valid* flag goes up when the last character of the key is loaded (to indicate that the key is effectively valid and respect the requirements), and the *key_byte_val* that remains up for all the loading process. It's loaded a char every clock cycle. In this case, the key is "abcdefghijkl".



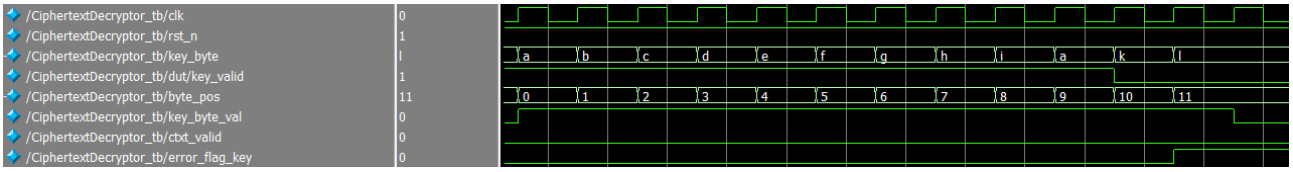
- **"Hello" word processing**: the waveform below explain the evolution of the decryption of the ciphertext that correspond to the word "Hello", with the key loaded in the previous point. It is

possible to notice that the *ctxt_valid* signal respect the requirements: when goes high, the ciphertext are correctly loaded. The plaintext is delivered one clock cycle later than when the corresponding ciphertext is loaded. Also the *ptxt_ready* respect the requirements: when goes high, the plaintext is shown. The *upper_lower* flag goes high to the first word, that is correctly capitalized ("H").

- **"Ale" word processing**: the waveform below explain the evolution of the decryption of the ciphertext that correspond to the word "Ale", with the key loaded in the first point. The complete ciphertext is "ak-ai-lh-ai-ae", but the ciphertext in 2nd and 4th positions are not loaded, because the *ctxt_valid* are set to 0. It is possible to notice that the *ctxt_valid* signal respect the requirements: when goes high, the ciphertext are correctly loaded into the registers; when it goes low, the ciphertext are not loaded. For *ptxt_valid*, it is possible to observe that it follows the behavior of ctxt_valid with a delay of one clock cycle, and it is compliant with the requirements: when goes high, the plaintext is shown. For the *upper_lower* flag, the considerations are the same as in the previous point.

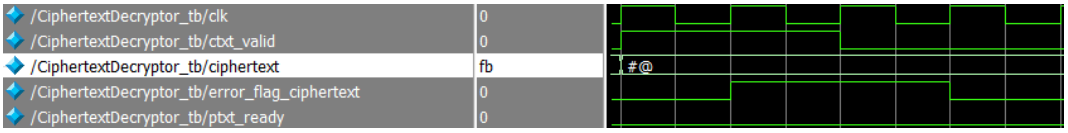


- **Loading invalid key:** the waveform below show the evolution of a loading of an incorrect key: it is incorrect because it contains a repeated character ("a").



The *error_flag_key* flag goes high one clock cycle away to notify that the key is incorrect.

- **Loading invalid ciphertext**: the following waveform shows the evolution of the loading of an incorrect ciphertext: in this case, the not admitted characters ("#@") are loaded. Notice that, in the same test, a valid key is used. From the waveforms, it can be seen that the error_flag_ciphertext signal goes to 1 one clock cycle after receiving the wrong pair of input characters. The one cycle delay is simply due to the registers allocated at the inputs.



From the waveforms, it can also be noticed that the ptxt_ready signal is not raised, which means that the received ciphertext is not decripted and no plaintext is returned in this case.

TESTBENCH

In this chapter, I will explain how the test benches are designed and what they do. To test also the model described in C code, I passed the same input vectors to the function invoked by DPI, and then I check if the results calculated from the module and from the C model are the same. An example of a function calling is the following:

substitution(key_c, plaintext_c, ciphertext[15:8], ciphertext[7:0], upper_lower, error_flag_key_c, error_flag_ciphertext_c)

The test I made are the following:

- **TEST 1**: in this test I provide the correct ciphertext “li-ae-lh-lh-jb”, that correspond to the word “Hello” with a correct key = “abcdefghijkl”. To check also the correct behaviour of the *upper_lower* signal, I set it to 1 when I provide to the circuit the characters “li”. So, the expected output word is “Hello”. The waveform result is presented in the previous section at the “Hello word processing” point. The C model return the correct word “Hello”, so the test is passed.
- **TEST 2**: in this test I provide the correct ciphertext “ak-ai-lh-ai-ae” with a correct key = “abcdefghijkl”. To check also the correct behaviour of the *upper_lower* signal, I set it to 1 when I provide to the circuit the characters “ak”. After every ciphertext pair, I set ctxt_valid to 0 to check whether the behavior is consistent with that described in the specification. The waveform result is presented in the previous section at the “Ale word processing” point. The C model return the correct word “Ale”, so the test is passed.
- **TEST 3**: in this test I provide a correct ciphertext “fb” but with an incorrect key “abcdefghiakl” The key is incorrect because contains the character “a” that is repeated twice. I see that the circuit set correctly the *error_flag_key* signal. The waveform result is presented in the previous section at the “Loading invalid key” point. The C model set correctly to 1 the *error_flag_key_c*, so the test is passed.
- **TEST 4**: in this test I provide a correct ciphertext “fb” but with an incorrect key “abc?efghijkl”. In this case, the key is not valid because contains a not admitted character “?”. Also in this case, the circuit set correctly the *error_flag_key* signal. The C model set correctly to 1 the *error_flag_key_c*, so the test is passed.
- **TEST 5**: in this test I provide an incorrect ciphertext “#@” with a correct key “abcdefghijkl”. In this case, the ciphertext contains not allowed characters: it is possible to notice that the circuit correctly set the *error_flag_ciphertext* signal. The waveform result is presented in the previous section at the “Loading invalid ciphertext” point. The C model set correctly to 1 the *error_flag_ciphertext_c*, so the test is passed.
- **TEST 6**: in this test I provide a correct ciphertext “ai-fk-ae” with a correct key = “abcdefghijkl”, that correspond to the word “Bye” (I set to 1 the upper_lower flag for the first character). I did this test to verify that the circuit, after returning error flags correctly (and thus after passing tests with errors) resumed normal operations. The C model return the correct word “Bye”, so the test is passed.

If a user want to use the module, it has to set the first byte of the key *key_byte*, the position of the byte *byte_pos* and the *key_byte_val* flag (to assert that the byte is stable). Then, it has to set the second byte of the key, the position and so on. It must repeat this process until the 12 bytes of the key are completely loaded. Then, it has to set the pair of ciphertext character *ciphertext*, the *ctxt_valid* flag (to assert that the ciphertext is stable) and the *upper_lower* flag to specify if it wants the corresponding plaintext in upper or lower case. The module signals when the process is completed by setting the *ptxt_ready* flag, then the user can read the plaintext at the *plaintext* port. It has to repeat this process until the ciphertext is completely deciphered.

SYNTHESIS RESULT AND STATIC TIMING ANALYSIS

I discuss the synthesis result in terms of resources employed over the target device and the summary of the timing analysis, also making a comparison between the version without virtual pins and the one with.

Starting from analyzing the summary of the *Analysis and Synthesis* section, it is possibile to notice that the number of total registers is 125:

Analysis & Synthesis Summary

<<Filter>>

Analysis & Synthesis Status	Successful - Tue Jul 11 12:18:55 2023
Quartus Prime Version	22.1std.1 Build 917 02/14/2023 SC Lite Edition
Revision Name	substitution_table_dec
Top-level Entity Name	CiphertextDecryptor
Family	Cyclone V
Logic utilization (in ALMs)	N/A
Total registers	125
Total pins	44
Total virtual pins	0
Total block memory bits	0
Total DSP Blocks	0
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0
Total DLLs	0

The number of used registers I estimated from the code is 134, 9 more registers than Quartus counted. If the registers are counted by the code, the result is:

- 18 register for the InputLoading module.
- 97 register for the KeyLoading module.
- 8 register for the SubstitutionTable module.
- 11 register for CiphertextDecryptor module.

The number **125** can be obtained analyzing the *Resource Utilization by Entity* report:

Analysis & Synthesis Resource Utilization by Entity

<<Filter>>

	Compilation Hierarchy Node	Combinational ALUTs	Dedicated Logic Registers
1	|CiphertextDecryptor	452 (10)	125 (10)
1	|InputLoading:inputloading|	0 (0)	18 (18)
2	|KeyLoading:keyloading|	309 (309)	97 (97)
3	|SubstitutionTableDecryption:substitution_table|	133 (133)	0 (0)

What it's noticeable is that Quartus perform an optimization not assigning registers to the *substitution_table* module. The remaining numbers are the same as I calculated earlier by looking at the code, except for the *CiphertextDecryptor* module, which has one less register.

So, if from the 134 registers are subtracted the 8 of the *substitution_table* module, the result is 126, one register more than those calculated by Quartus. But, going to analyze the optimization aspects in more detail, particularly in the *Registers Statistics*, it is possibile to see that Quartus removed one register during the synthesis:

Registers Removed During Synthesis

<<Filter>>

	Register name	Reason for Removal
1	plaintext[7]~reg0	Stuck at GND due to stuck port data_in
2	Total Number of Removed Registers = 1	

Quartus remove a register in the *CiphertextDecryptor* module, as has been noted previously: this is coherent with the design, because from the code I counted 11 registers for this module. Notice that, at the *Resource Utilization* report, Quartus used 10 registers (11 minus the one Quartus removed). So, the number of registers I calculated from code is coherent with the design optimized by Quartus.

Now I analyze the fitting results.

Flow Summary

<<Filter>>

Flow Status	Successful - Tue Jul 11 12:22:55 2023
Quartus Prime Version	22.1std.1 Build 917 02/14/2023 SC Lite Edition
Revision Name	substitution_table_dec
Top-level Entity Name	CiphertextDecryptor
Family	Cyclone V
Device	5CEBA2F17C7
Timing Models	Final
Logic utilization (in ALMs)	300 / 9,430 (3 %)
Total registers	132
Total pins	44 / 128 (34 %)
Total virtual pins	0
Total block memory bits	0 / 1,802,240 (0 %)
Total DSP Blocks	0 / 25 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 4 (0 %)
Total DLLs	0 / 4 (0 %)

This image shows the summary of the resources used by the module after the fitting, without using the virtual pins. It is possibile to notice the low usage of resources, which is 3%, in relation to those available in the target. Also, pins used are the 34%.

If the virtual pins are used, the summary is the following:

Flow Summary

<<Filter>>

Flow Status	Successful - Thu Jul 13 16:40:34 2023
Quartus Prime Version	22.1std.1 Build 917 02/14/2023 SC Lite Edition
Revision Name	substitution_table_dec
Top-level Entity Name	CiphertextDecryptor
Family	Cyclone V
Device	5CEBA2F17C7
Timing Models	Final
Logic utilization (in ALMs)	324 / 9,430 (3 %)
Total registers	133
Total pins	1 / 128 (< 1 %)
Total virtual pins	43
Total block memory bits	0 / 1,802,240 (0 %)
Total DSP Blocks	0 / 25 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 4 (0 %)
Total DLLs	0 / 4 (0 %)

It is possible to observe two main differences between the two reports:

- The resources usage, when virtual pins are used, is slightly higher.
- The number of register used is 133, one more register than the version without virtual pins.

The number of used pins, in both versions, is coherent with our expectations:

- Rst_n: 1 bit.
- Key_byte: 8 bit.
- Byte_pos: 4 bit.
- Key_byte_val: 1 bit.
- Ciphertext: 16 bit.
- upper_lower: 1 bit.
- ctxt_valid: 1 bit.
- ptxt_ready: 1 bit.
- Plaintext: 8 bit.
- error_flag_key: 1 bit.
- error_flag_ciphertext: 1 bit.
  43 total pins. If I also add the clock, it becomes 44 pins.

The observations on the registers are the same as for the model without virtual pins, except for the register removed from *CiphertextDecryptor* module.

For the Static Timing Analysis I have defined a timing constraints file (SDC.sdc) specifying a target frequency of 100MHz and defining a 10 nanoseconds period clock.

	Clock Name	Type	Period	Frequency	Rise	Fall
1	clk	Base	10.000	100.0 MHz	0.000	5.000

I'm going to analyze the results for both versions to see if there are differences between them.

I defined minimum and maximum input delay for the listed input ports (10% and 20% respectively) and minimum and maximum output delay for the listed output ports (10% and 20% respectively). The frequencies obtained during the slow tests (85C and 0C), for the version without virtual pins, are reported below:

Slow 1100mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	105.09 MHz	105.09 MHz	clk	

Slow 1100mV 0C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	105.15 MHz	105.15 MHz	clk	

It is observable that the obtained frequencies meet the requirements even in the worst case (on the left), with a frequency equal to 105.09 MHz, and up to 105.15 MHz with a much lower temperature (on the right). For the version with virtual pins, the frequencies obtained are:

Slow 1100mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	117.29 MHz	117.29 MHz	clk	

Slow 1100mV 0C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	117.63 MHz	117.63 MHz	clk	

Notice that the Fmax increase in both case: in the model with higher temperature, it goes from 105.09 to 117.29; in the model with lower temperature, it goes from 105.15 to 117.63. So, the virtual pins better suits with the design as they increase the maximum frequency decreasing paths delay.