



UNIVERSITY OF PISA  
MASTER'S DEGREE IN CYBERSECURITY

FEEDBACK CIPHER  
COURSE HARDWARE AND EMBEDDED SECURITY

Author(s)  
**Lombardi Giacomo**  
**Zarulli Nicolò**

Academic year 2023/2024

---

# Contents

---

<b>1</b>	<b>Project Specifications</b>	<b>1</b>
<b>2</b>	<b>High-level Model</b>	<b>4</b>
2.1	utils.py . . . . .	5
2.1.1	SBox . . . . .	5
2.1.2	Feedback Cipher . . . . .	6
2.2	main.py . . . . .	6
<b>3</b>	<b>RTL Design</b>	<b>8</b>
3.1	Feedback Cipher Architecture . . . . .	8
3.2	Feedback Cipher FSM . . . . .	10
3.3	Feedback Cipher System Verilog Implementation . . . . .	10
3.3.1	aes_sbox.sv . . . . .	10
3.3.2	aes_feedback_cipher.sv . . . . .	11
<b>4</b>	<b>Interface Specifications and Expected Behavior</b>	<b>14</b>
4.1	Correct Input Sequence . . . . .	15
4.2	Wrong use case (encryption) . . . . .	15
<b>5</b>	<b>Functional Verification</b>	<b>16</b>
5.1	Testbench for Correct Usage (aes_feedback_cipher_tb.sv) . . . . .	16
5.2	Testbench for Incorrect Usage (wrong_use_tb.sv) . . . . .	19
<b>6</b>	<b>FPGA Implementation Results</b>	<b>22</b>
6.1	Quartus steps . . . . .	22
6.2	Synthesis and Fitting . . . . .	22
6.3	Static Timing Analysis results . . . . .	25

---

# CHAPTER 1

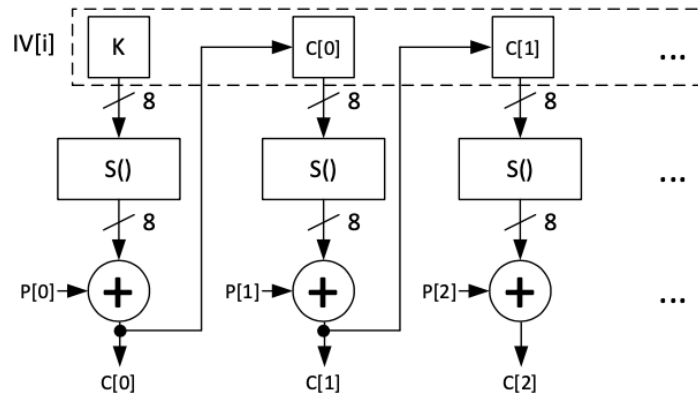
---

## Project Specifications

---

This project is the result of the course *Hardware and Embedded Security* for the *Cybersecurity* Master Course at *University of Pisa*, academic year 2023-2024.

The goal is to design, implement and test the **Feedback Cipher (AES-Sbox based)** schema. The project follows a FPGA design workflow, which sees the implementation of both the encryption and decryption functionalities of the cipher through System Verilog design files. These are then processed through the *Modelsim* software to observe the cipher's behavior on a zero-delay simulation. As a last step, the design is loaded onto *Quartus Prime* to carry out the Logic Synthesis, Fitting and Static Timing Analysis phases. The project also outlines an high-level representation of the logic's of the cipher, which is presented in Python.



**Figure 1.1:** Feedback Cipher AES SBox Schema

The Feedback Cipher design uses a simplified version of the *AES-SBox* and is able to support both encryption and decryption functionalities. The encryption algorithm of the cipher is performed by XORing each plaintext  $P[i]$  byte with an 8-bit value obtained by substituting an 8-bit Initialization Vector (*IV*) with the *S-box transformation* of the AES algorithm. The 8-bit *IV* has to be initialized with the value of an 8-bit symmetric key,  $K$ , then it must update with the value of the previous ciphertext byte. The encryption law of the feedback cipher can be expressed as it follows:

$$C[i] = P[i] \oplus S(IV[i])$$

Where:

- $C[i]$  is the  $i$ -th byte of the ciphertext.
- $P[i]$  is the  $i$ -th byte of the plaintext.
- $IV[i]$  is the  $i$ -th 8-bit value of the Initialization Vector (*IV*), for  $i = 0, 1, 2, \dots$ .  $IV[0] = K$ , being  $K$  the 8-bit symmetric key, whereas  $IV[i] = C[i - 1]$  for  $i = 1, 2, 3, \dots$  i.e. the previous ciphertext byte.
- $S()$  is the S-box transformation of AES algorithm, that works over a byte.
- $\oplus$  is the XOR operator.

The design also implements:

- An asynchronous *active-low reset* port.
- An *input flag*, which is driven as it follows: *value 1* when input data byte on the corresponding input port is valid and stable (it can be used by the internal logic), *value 0* otherwise.
- An *output flag*, which is driven as it follows: *value 1* when output data byte on the corresponding output port is ready and stable (external modules can read and use it), *value 0* otherwise.
- A *new\_message* flag: for each arbitrary length plaintext/ciphertext message, the counter block shall start from the initialization value  $K$  (the 8-bit key). Therefore, the module interface should also include an input flag to signal when a new message begins.
- A *functionality flag*: to signal if the input data must be encrypted (i.e. the input data corresponds to a plaintext byte) or decrypted (i.e. the input data corresponds to a ciphertext byte), since the usage of the input data byte (and eventually of the previous input data byte) change according to the process to be performed (encryption or decryption).

The SBox model adopted is a simplified version of the Substitution Box of the AES algorithm. For faster developing, we used the LUT version. This SBox is a table of values which is accessed based on the *LSB* and *MSB* of the input byte. An example:

assuming to apply the S-box transformation to the byte (hex)  $8'hD3$ , the result (hex) is  $8'h66$ , i.e. the cross between row  $D0$  and column  $03$ :  $S(8'hD3) = 8'h66$ . See Figure 1.2

	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

**Figure 1.2:** AES SBox LUT

For the decryption function, we follow the rule that the AES SBox is always accessed in "*encryption mode*", and thus we are able to exploit the properties of the XOR  $\oplus$  operator to be able to reuse the same design, but with a modified round function to extract the plaintext block from a given ciphertext block, as it follows:

$$P[i] = C[i] \oplus S(IV[i])$$

where  $IV[]$  has the same characteristics of the encryption mode. The XOR  $\oplus$  property used:

$$S \oplus S = 0$$

thus, if  $C = P \oplus S$ , then  $C \oplus S = (P \oplus S) \oplus S = P \oplus S \oplus S = P \oplus (S \oplus S) = P$

---

## CHAPTER 2

---

### High-level Model

---

The high-level model is presented to show a simple and direct implementation of the *Feedback Cipher* schema. The language of choice is **Python** (version 3.10 or higher).

---

#### Feedback Cipher High-Level Model Sub-Tree

```
project
├── model/
│   ├── main.py
│   └── utils.py
```

---

The model consists of two python scripts:

- **utils.py**: a module which incapsulates the classes **SBox** and **FeedbackCipher** used to define the logic of the cipher.
- **main.py**: a script which contains examples of the cipher's possible executions and tests.

## 2.1 utils.py

This is a python module script which is used to implement the backbone of the cipher. The main.py script will implement this module's classes to show the functionalities of the cipher. The choice of using python classes is justified to impose an higher level of control and security over the cipher characteristics and internal parameters.

The file consists of two classes:

- **class *SBox***: used to implement the SBox functionality required for the Feedback Cipher.
- **class *FeedbackCipher***: the actual cipher's logic.

### 2.1.1 SBox

The class SBox implements an internal field which is the required AES SBox needed for the Feedback Cipher. The AES SBox values are accessed through a *getter* function, Figure [2.1].

```
1 class SBox:
2     def __init__(self):
3         self._aes_matrix = [
4             [0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01, 0x67, 0x2B, 0xFE, 0xD7, 0xAB, 0x76],
5             [0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4, 0xA2, 0xAF, 0x9C, 0xA4, 0x72, 0xC0],
6             [0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC, 0x34, 0xA5, 0xE5, 0xF1, 0x71, 0xD8, 0x31, 0x15],
7             [0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80, 0xE2, 0xEB, 0x27, 0xB2, 0x75],
8             [0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52, 0x3B, 0xD6, 0xB3, 0x29, 0xE3, 0x2F, 0x84],
9             [0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE, 0x39, 0x4A, 0x4C, 0x58, 0xCF],
10            [0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02, 0x7F, 0x50, 0x3C, 0x9F, 0xA8],
11            [0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5, 0xBC, 0xB6, 0xDA, 0x21, 0x10, 0xFF, 0xF3, 0xD2],
12            [0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E, 0x3D, 0x64, 0x5D, 0x19, 0x73],
13            [0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46, 0xEE, 0xB8, 0x14, 0xDE, 0x5E, 0x0B, 0xDB],
14            [0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2, 0xD3, 0xAC, 0x62, 0x91, 0x95, 0xE4, 0x79],
15            [0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0xEA, 0x65, 0x7A, 0xAE, 0x08],
16            [0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74, 0x1F, 0x4B, 0xBD, 0x8B, 0x8A],
17            [0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35, 0x57, 0xB9, 0x86, 0xC1, 0x1D, 0x9E],
18            [0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E, 0x87, 0xE9, 0xCE, 0x55, 0x28, 0xDF],
19            [0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99, 0x2D, 0x0F, 0xB0, 0x54, 0xBB, 0x16]
20        ]
21    def access_aes(self, x, y):
22        return self._aes_matrix[x][y]
23
```

Figure 2.1: *utils.py*, class *SBox*

### 2.1.2 Feedback Cipher

The class FeedbackCipher has two private fields, Figure [2.2]:

- **key**: the 8-bit hexadecimal symmetric key used for the encryption and decryption functions.
- **sbox**: the chosen SBox for the cipher logic. This field is publicly set and is saved into the internal field and to ensure security when the setter method is called, it goes through a type checking routine.

```
24 class FeedbackCipher:
25     def __init__(self, sbox):
26         self._key = 0xA5
27         self._sbox = None
28         self.sbox = sbox
29
30     # SBox TypeChecking
31     @property
32     def sbox(self):
33         return self._sbox
34
35     @sbox.setter
36     def sbox(self, value):
37         if not isinstance(value, SBox):
38             raise TypeError(f"Expected value of type SBox, got {type(value)} instead.")
39         self._sbox = value
40
```

Figure 2.2: *utils.py*, FeedbackCipher fields

The class has two main methods:

- **encrypt (self, p, mode, test)** : takes a plaintext *p* to encrypt it with the chosen *mode* ("1" for a string, "2" for an array with hex values). The test parameter is used to enforce a test routine in the *main.py* script.
- **decrypt (self, c, mode)** : takes a ciphertext *c* and decrypts it with the chosen *mode* ("1" for a string, "2" for an array with hex values).

The class also presents the functions **encrypt\_string()**, **decrypt\_string()**, **encrypt\_array()** and **decrypt\_array()**, which are the independent implementations of, respectively, mode 1 and 2 of the **encrypt()** and **decrypt()** functions. Even though the result of their executions does not change the final results, it is recommended to use the latter ones mentioned, Figure [2.3].

## 2.2 main.py

This python script is used to load the *utils.py* module and show the functionalities of the cipher. In this file there can be found three simple case studies:

1. **String**: a string is given as a plaintext to the cipher. The produced ciphertext is then decrypted to show the validity of the process.
2. **Array of Hex Values**: an array of hex values is given as a plaintext to the cipher. The produced ciphertext is then decrypted to show the validity of the process.



```

24 class FeedbackCipher:
25     def __init__(self, sbbox):
26         self._key = 0xA5
27         self._sbbox = None
28         self.sbbox = sbbox
29
30     # SBbox TypeChecking
31     @property
32 > def sbbox(self):--
33
34
35     @sbbox.setter
36 > def sbbox(self, value):--
37
38
39
40
41     # FeedBack Cipher Implementations
42
43     # ----- Array CBC -----
44 > def encrypt_array(self, a, test):--
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71 > def decrypt_array(self, a):--
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96 # -----
97
98 # --- CBC String Enc / Dec ---
99 > def encrypt_string(self, p):--
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116 > def decrypt_string(self, c):--
117
118
119
120
121
122
123
124
125 # -----
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162 # --- Multiple Ciphers -----
163 > def encrypt(self, p, mode, test):--
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216 > def decrypt(self, c, mode):--
217
218
219
220
221
222
223
224 # -----

```

**Figure 2.3:** *utils.py*, *FeedbackCipher* methods

3. **Propagation Delay Test:** In the encryption process the first round is delayed to the second stage. This test is done to mimic and compare the results with the same test being carried on later on in the report with a testbench over the system verilog implementation of the feedback cipher.

```

● nicolo_zarulli@192 model % python3 main.py
-- (1) TEST STRING --
SETUP
-> Plaintext: LanaDelRey
AES SBbox ENCRYPTION
-> Ciphertext: 4ab7c7a718c8840db24e
AES SBbox DECRYPTION
-> Original Plaintext: LanaDelRey

-- (2) TEST HEX ARRAY --
SETUP
-> Plaintext: ['0x3c', '0x7f', '0x9b', '0x5e', '0x1a', '0x9c', '0x2a', '0x12', '0x39', '0xb5']
AES SBbox ENCRYPTION
-> Ciphertext ['0x3a', '0xff', '0x8d', '0x3', '0x61', '0x73', '0xa5', '0x14', '0xc3', '0x9b']
AES SBbox DECRYPTION
-> Original Plaintext: ['0x3c', '0x7f', '0x9b', '0x5e', '0x1a', '0x9c', '0x2a', '0x12', '0x39', '0xb5']

-- (3) Propagation Delay --
SETUP
-> Plaintext: ['0x3c', '0x7f', '0x9b', '0x5e', '0x1a', '0x9c', '0x2a', '0x12', '0x39', '0xb5']
AES SBbox ENCRYPTION
-> Ciphertext ['0x0', '0x79', '0x2d', '0x86', '0x5e', '0xc4', '0x36', '0x17', '0xc9', '0x68']
AES SBbox DECRYPTION
-> Original Plaintext: ['0x6', '0x1a', '0x9b', '0x5e', '0x1a', '0x9c', '0x2a', '0x12', '0x39', '0xb5']

```

**Figure 2.4:** *main.py* execution

---

# CHAPTER 3

---

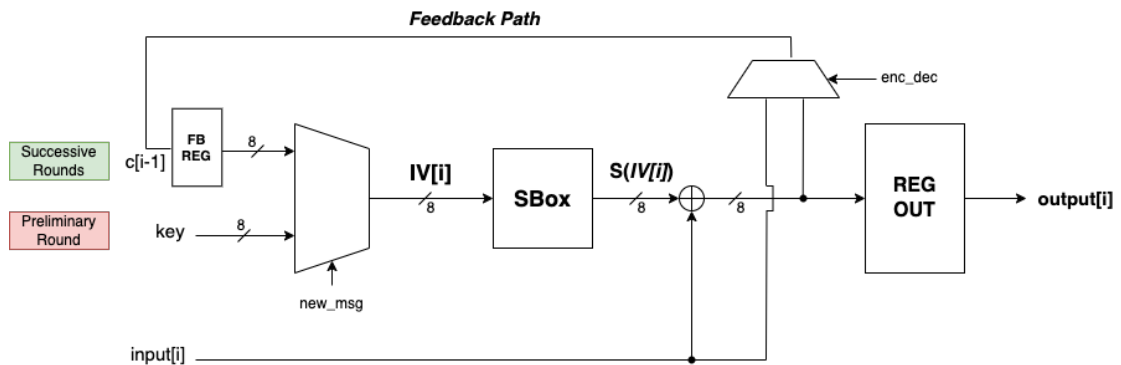
## RTL Design

---

Based on the analysis of the specifications of the *Feedback Cipher* schema, this chapter illustrates the architectural and design choices made for the System Verilog implementation of the cipher.

### 3.1 Feedback Cipher Architecture

The *Feedback Cipher* requires the implementation of a feedback path to execute the  $i = 1, 2, 3, \dots, n - 1$  rounds (with  $n$  being the number of bytes of the input), so we opted to implement a revised and adapted version of the *Single-inter-round-pipelined architecture*, as illustrated in Figure 3.1.



**Figure 3.1:** *Feedback Cipher RTL Design*

The schema highlights the division of the flow of the cipher, w.r.t. the feedback path that feeds the SBox combinational circuit, into two stages that are managed from the

*new\_msg* flag. We have:

- **Preliminary Round** ( $i = 0$ ): The IV loads the symmetric key and uses it to first access the SBox.
- **Successive Rounds** ( $i = 1, 2, \dots, n - 1$ ): The IV loads the feedback data received from the previous round.

Another characteristic of this architecture is what kind of data needs to be sent into the feedback path. This data does not come from the same path, and that is due to the fact that this cipher handles both encryption and decryption onto the same schema. By taking a look at the rules highlighted in the specifications chapter, we can see what type of data is loaded to the feedback register:

- **Encryption:**  $C[i] = P[i] \oplus S(IV[i])$
- **Decryption:**  $P[i] = C[i] \oplus S(IV[i])$

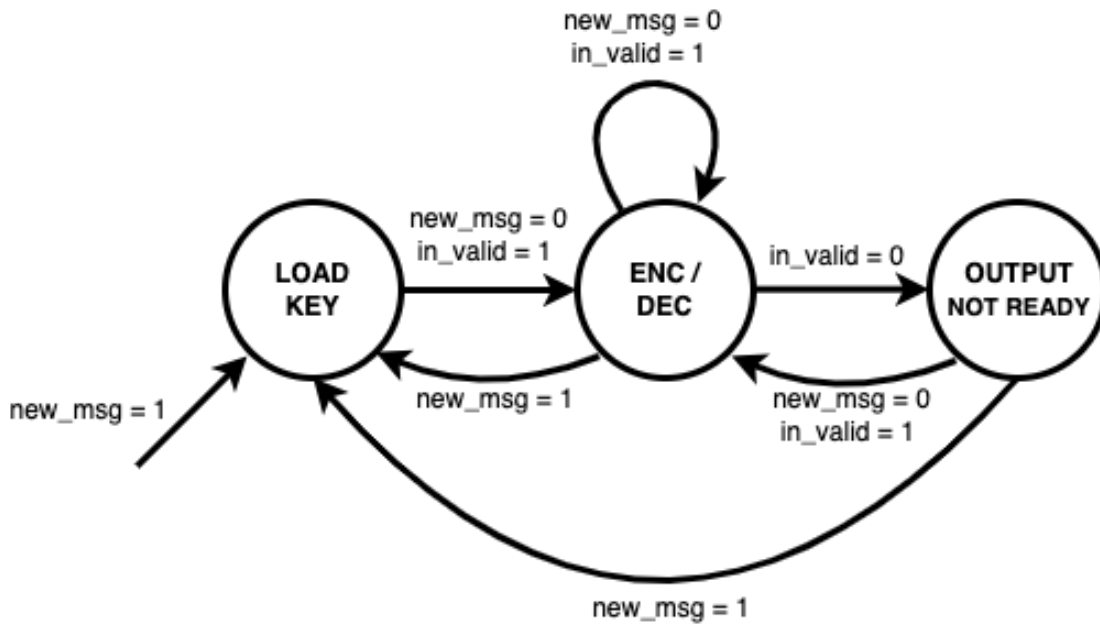
Where  $IV[i] = C[i - 1]$  for  $i = 1, 2, \dots, n - 1$  (i.e. the previous ciphertext byte). But that ciphertext byte comes from different paths based on whether the cipher is handling encryption or decryption. Therefore, we added a multiplexer that manages this difference and loads to the feedback register the right data. The Feedback MUX is controlled from the *enc\_dec* flag:

- **Encryption (*enc\_dec* = 1):** The input to the SBox of the next round is obtained from the output of the XOR operation.
- **Decryption (*enc\_dec* = 0):** The input to the SBox step of the next round is obtained directly from the input path of the round.

## 3.2 Feedback Cipher FSM

---

The next step made was to define a Finite State Machine which would handle the flow of this architecture schema. Figure 3.2 outlines the FSM we designed and that will be implemented with the System Verilog cipher files.



**Figure 3.2:** *Feedback Cipher FSM*

## 3.3 Feedback Cipher System Verilog Implementation

---

The implementation of the Feedback Cipher schema can be found in the *db* folder of the project and is organized in two distinct files: *aes\_feedback\_cipher.sv* and *aes\_sbox.sv*.

---

### Feedback Cipher System Verilog

```
project
├── db/
│   ├── aes_feedback_cipher.sv
│   └── aes_sbox.sv
```

---

#### 3.3.1 aes\_sbox.sv

This file provides the actual S-box transformation logic, referencing a pre-defined LUT to map each 8-bit input to its corresponding 8-bit output in the AES substitution process, Figure 3.3. This module is instantiated and used in the *aes\_feedback\_cipher.sv* file.

```

db >  aes_sbox.sv
1  module aes_sbox (
2      input  [7:0] in
3      ,output [7:0] out
4  );
5
6      wire [0:255][7:0] aes_sbox_lut = {
7          8'h63, 8'h7c, 8'h77, 8'h7b, 8'hf2, 8'h6b, 8'h6f, 8'hc5, 8'h30, 8'h01, 8'h67, 8'h2b, 8'hfe, 8'hd7, 8'hab, 8'h76,
8          8'hca, 8'h82, 8'hc9, 8'h7d, 8'hfa, 8'h59, 8'h47, 8'hf0, 8'had, 8'hd4, 8'ha2, 8'haf, 8'h9c, 8'ha4, 8'h72, 8'hc0,
9          8'hb7, 8'hfd, 8'h93, 8'h26, 8'h36, 8'h3f, 8'hf7, 8'hcc, 8'h34, 8'ha5, 8'he5, 8'hf1, 8'h71, 8'hd8, 8'h31, 8'h15,
10         8'h04, 8'hc7, 8'h23, 8'hc3, 8'h18, 8'h96, 8'h05, 8'h9a, 8'h07, 8'h12, 8'h80, 8'he2, 8'heb, 8'h27, 8'hb2, 8'h75,
11         8'h09, 8'h83, 8'h2c, 8'h1a, 8'h1b, 8'h6e, 8'h5a, 8'ha0, 8'h52, 8'h3b, 8'hd6, 8'hb3, 8'h29, 8'he3, 8'h2f, 8'h84,
12         8'h53, 8'hd1, 8'h00, 8'hed, 8'h20, 8'hfc, 8'hb1, 8'h5b, 8'h6a, 8'hcb, 8'hbe, 8'h39, 8'h4a, 8'h4c, 8'h58, 8'hcf,
13         8'hd0, 8'hcf, 8'haa, 8'hfb, 8'h43, 8'h4d, 8'h33, 8'h85, 8'h45, 8'hf9, 8'h02, 8'h7f, 8'h50, 8'h3c, 8'h9f, 8'ha8,
14         8'h51, 8'ha3, 8'h40, 8'h8f, 8'h92, 8'h9d, 8'h38, 8'hf5, 8'hbc, 8'hb6, 8'hda, 8'h21, 8'h10, 8'hff, 8'hf3, 8'hd2,
15         8'hcd, 8'h0c, 8'h13, 8'hec, 8'h5f, 8'h97, 8'h44, 8'h17, 8'hc4, 8'ha7, 8'h7e, 8'h3d, 8'h64, 8'h5d, 8'h19, 8'h73,
16         8'h60, 8'h81, 8'h4f, 8'hdc, 8'h22, 8'h2a, 8'h90, 8'h88, 8'h46, 8'hee, 8'hb8, 8'h14, 8'hde, 8'h5e, 8'h0b, 8'hdb,
17         8'he0, 8'h32, 8'h3a, 8'h0a, 8'h49, 8'h06, 8'h24, 8'h5c, 8'hc2, 8'hd3, 8'hac, 8'h62, 8'h91, 8'h95, 8'he4, 8'h79,
18         8'he7, 8'hc8, 8'h37, 8'h6d, 8'h8d, 8'hd5, 8'h4e, 8'ha9, 8'h6c, 8'h56, 8'hf4, 8'hea, 8'h65, 8'h7a, 8'hae, 8'h08,
19         8'hba, 8'h78, 8'h25, 8'h2e, 8'h1c, 8'ha6, 8'hb4, 8'hc6, 8'he8, 8'hdd, 8'h74, 8'h1f, 8'h4b, 8'hbd, 8'h8b, 8'h8a,
20         8'h70, 8'h3e, 8'hb5, 8'h66, 8'h48, 8'h03, 8'hf6, 8'h0e, 8'h61, 8'h35, 8'h57, 8'hb9, 8'h86, 8'hcl, 8'h1d, 8'h9e,
21         8'he1, 8'hf8, 8'h98, 8'h11, 8'h69, 8'hd9, 8'h8e, 8'h94, 8'h9b, 8'h1e, 8'h87, 8'he9, 8'hce, 8'h55, 8'h28, 8'hdf,
22         8'h8c, 8'ha1, 8'h89, 8'h0d, 8'hbf, 8'he6, 8'h42, 8'h68, 8'h41, 8'h99, 8'h2d, 8'h0f, 8'hb0, 8'h54, 8'hbb, 8'h16
23     };
24
25     assign out = aes_sbox_lut[in];
26
27 endmodule

```

**Figure 3.3:** AES SBox System Verilog

### 3.3.2 aes\_feedback\_cipher.sv

This file implements the feedback cipher schema as a System Verilog module handling both encryption and decryption, utilizing an S-box transformation for security. The logic and the code of the module is analyzed and discussed in the following points:

#### 1. Inputs and Outputs, Figure 3.4:

- `clk, rst`: Clock and reset signals.
- `new_msg`: Signals the start of a new encryption/decryption process.
- `enc_dec`: Indicates whether the process is encryption (1) or decryption (0).
- `in_valid`: Marks the input as valid, allowing the processing to begin.
- `key`: Used as the initialization vector (IV) for the first round.
- `in_msg`: The input message to be encrypted (plaintext) or decrypted (ciphertext).
- `out_msg`: The resulting output message (ciphertext for encryption or plaintext for decryption).
- `out_ready`: Indicates that the output is ready.

#### 2. Internal Registers, Figure 3.4:

- `iv`: Holds the initialization vector (IV). For round 0, it is set to the key; for subsequent rounds, it is updated depending on the type of operation.
- `sbox_out`: Stores the result of applying an S-box transformation, provided by the `aes_sbox` module.
- `round_out`: The output of the XOR operation that produces either the ciphertext or plaintext.


- **done:** Signals the successful completion of a round.

### 3. Functionality, Figure 3.5:

- **Reset:** On reset (!rst), all internal registers are cleared.
- **New Message:** When new\_msg is asserted, the iv register is initialized with the provided key.
- **Encryption/Decryption Logic:** When in\_valid is asserted:
  - **Encryption** (enc\_dec = 1): The module computes  $C[i] = P[i] \oplus S(IV)$ , where  $S(IV)$  is the S-box transformation of iv. The feedback updates iv with the value of  $C[i]$  just computed.
  - **Decryption** (enc\_dec = 0): It computes  $P[i] = C[i] \oplus S(IV)$ , and feedback sets iv to  $C[i]$ .

The output of each round (round\_out) is assigned to out\_msg, and out\_ready is set when done is high.

To end this section, Figure 3.6 highlights the actual diagram of the Feedback Cipher RTL module, which was obtained from the Quartus Prime software.

```
db >  aes_feedback_cipher.vh
1  module aes_feedback_cipher (
2      input      clk,          // S: Clock
3      input      rst,          // S: Reset
4      input      new_msg,      // S: New message, start the process
5      input      enc_dec,      // S: Encryption or Decryption
6      input      in_valid,     // S: Input valid
7      output     out_ready,    // S: Output valid signal
8      input  [7:0] key,        // W: key for IV Round Zero
9      input  [7:0] in_msg,     // W: Enc: P[i] - Dec: C[i]
10     output [7:0] out_msg     // W: Enc: C[i] - Dec: P[i]
11 );
12 // Internal Circuit Registers
13 logic [7:0] iv;              // Initialization Vector (Round 0: Key, Round i: C[i-1])
14 logic [7:0] sbox_out;        // Output of SBox Combinational Circuit
15 logic [7:0] round_out;       // Output of the XOR operation
16 logic done;                  // Successful Round Completion Flag
17
18 // AES SBox Instance
19 aes_sbox feedback_cipher_sbox(
20     .in(iv),
21     .out(sbox_out)
22 );
```

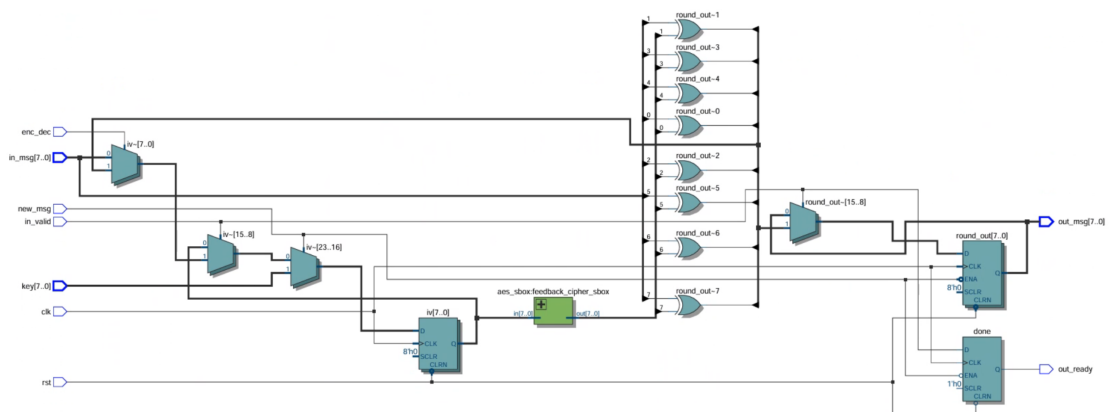
**Figure 3.4:** Feedback Cipher Inputs/Outputs and Internal Registers

```

24     always_ff @(posedge clk or negedge rst) begin
25         if (!rst) begin
26             // Reset all internal registers
27             iv <= 8'b0;
28             round_out <= 8'b0;
29             done <= 1'b0;
30         end
31         else if (new_msg) begin
32             // A new message begins, load the key for Round 0
33             iv <= key;
34         end
35         else if (in_valid) begin
36             // The input is declared valid, proceed with cipher logic
37             if (enc_dec) begin
38                 // Encryption: C[i] = P[i] ^ S(IV)
39                 round_out <= in_msg ^ sbox_out;
40                 // Feedback: IV[i+1] = C[i]
41                 iv <= in_msg ^ sbox_out;
42             end
43             else begin
44                 // Decryption: P[i] = C[i] ^ S(IV)
45                 round_out <= in_msg ^ sbox_out;
46                 // Feedback: IV[i+1] = C[i]
47                 // The input message is the actual ciphertext, which we also need for the
48                 iv <= in_msg;
49             end
50             done <= 1'b1;
51         end
52         else begin
53             done <= 1'b0;
54         end
55     end
56
57     assign out_msg = round_out;
58     assign out_ready = done;
59 endmodule

```

**Figure 3.5:** Feedback Cipher Functionalities



**Figure 3.6:** Feedback Cipher RTL Diagram, from Quartus

---

## CHAPTER 4

---

### Interface Specifications and Expected Behavior

---

To ensure proper usage of the module for encryption or decryption:

- **Reset Phase:**

- Initially, assert the reset signal ( $rst=0$ ) to initialize the internal state of the module. This will clear the internal registers ( $iv$ ,  $round\_out$ , etc.).
- Deassert the reset ( $rst=1$ ) to begin normal operation.

- **Starting a New Message:**

- To start processing a new message, provide the key and set  $new\_msg=1$  to load the key into the initialization vector ( $iv$ ) register.
- Ensure that  $new\_msg$  is set back to 0 before providing the first byte of the message. The first byte should only be provided when  $new\_msg=0$ , along with  $in\_valid=1$ . Alternatively, if you want to provide the first byte of the message at the same time as the key i.e. when  $new\_msg=1$ , you must hold the first byte stable until the positive edge of the clock in which  $new\_msg=0$ . After this, you can continue providing a new byte of the message at each clock cycle with  $in\_valid=1$ .
- If the first byte is handled correctly, it is then possible to provide the message one byte per clock cycle.

- **Providing the first byte:**

- Once  $new\_msg=0$ , you can start providing the message bytes one at a time.
- Set  $in\_valid=1$  and provide the first byte of plaintext (or ciphertext) on  $in\_msg$ . The corresponding output byte will be available in the next clock cycle. It can be read when  $out\_ready=1$ .

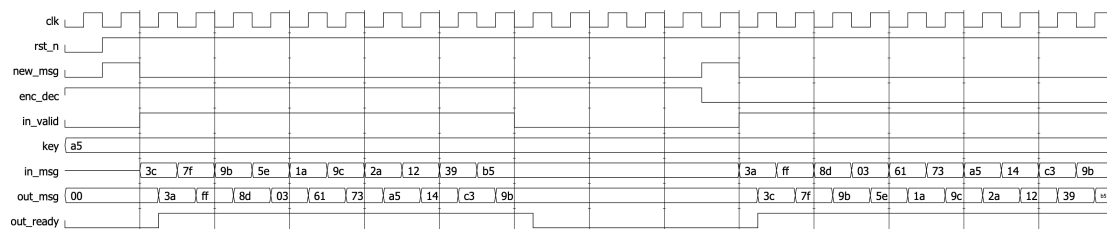


- **Subsequent bytes:**

- For subsequent bytes, you can provide one byte at each clock cycle with  $in\_valid=1$ .
- The output for each byte will appear on the next clock cycle, and  $out\_ready=1$  will indicate that it can be read.

## 4.1 Correct Input Sequence

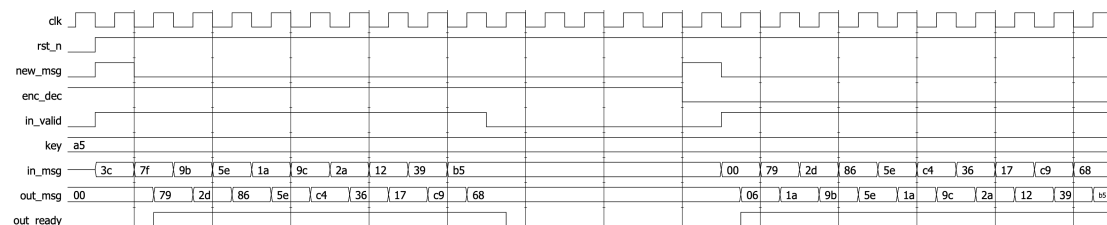
To ensure proper encryption or decryption one possibility is to load the key into  $iv$  before the first message byte is provided. This means that  $new\_msg$  is set to 0 before the first byte of the message is provided. This guarantees that the first byte is processed correctly, and encryption or decryption starts as expected. Figure 4.1 illustrates the waveform of a correct usage of the cipher in both encryption and decryption.



**Figure 4.1:** *Correct waveform*

## 4.2 Wrong use case (encryption)

If  $new\_msg=1$  and  $in\_valid=1$  are asserted at the same time (i.e. the key and first message byte are provided in the same clock cycle),  $new\_msg$  is kept high for one clock cycle and a new message block is provided at each clock cycle, the first byte won't be encrypted. Encryption will start from the second byte, computing  $P[1] \oplus S(K)$ . Figure 4.2 illustrates the waveform of a wrong usage in encryption together with a correct usage in decryption.



**Figure 4.2:** *Wrong waveform*

---

# CHAPTER 5

---

## Functional Verification

---

To ensure that the `aes_feedback_cipher` module functions as intended, two testbenches were created: one demonstrating the correct usage of the module and the other showcasing improper usage.

### 5.1 Testbench for Correct Usage (`aes_feedback_cipher_tb.sv`)

---

The testbench architecture includes the following elements:

- **Inputs and Outputs:**

- Inputs: `clk`, `rst_n`, `new_msg`, `enc_dec`, `in_valid`, `key`, `in_msg`.
- Outputs: `out_msg`, `out_ready`.

- **Internal Variables:**

- Arrays to hold the plaintext input bytes (`plaintext`) and the expected ciphertext output (`expected_ciphertext`).
- Arrays to store the results of the encryption (`encrypted_output`) and decryption (`decrypted_output`).

- **Testing Procedure:**

1. **Initialization:** The reset signal is asserted, and the clock is configured to toggle every 5 time units (clock period is 10 time units).
2. **Encryption Test:** The test reads input values from a file into the `plaintext` array and the expected ciphertext into the `expected_ciphertext` array. The key is provided in the corresponding port. The `new_msg` signal is set high

to indicate the start of a new message, kept high for 10 time units, and then set to 0 to ensure that it is low when the first byte is provided. Each byte of plaintext is provided to the module one at a time, every 10 time units, with in\_valid=1. Each output byte is checked against the expected ciphertext byte, and the results are displayed.

```
# Starting encryption test
# Encryption 1 correct:
#     Input byte: 3c
#     Output byte: 3a
#     Expected output byte: 3a
# Encryption 2 correct:
#     Input byte: 7f
#     Output byte: ff
#     Expected output byte: ff
# Encryption 3 correct:
#     Input byte: 9b
#     Output byte: 8d
#     Expected output byte: 8d
# Encryption 4 correct:
#     Input byte: 5e
#     Output byte: 03
#     Expected output byte: 03
# Encryption 5 correct:
#     Input byte: 1a
#     Output byte: 61
#     Expected output byte: 61
# Encryption 6 correct:
#     Input byte: 9c
#     Output byte: 73
#     Expected output byte: 73
# Encryption 7 correct:
#     Input byte: 2a
#     Output byte: a5
#     Expected output byte: a5
# Encryption 8 correct:
#     Input byte: 12
#     Output byte: 14
#     Expected output byte: 14
# Encryption 9 correct:
#     Input byte: 39
#     Output byte: c3
#     Expected output byte: c3
# Encryption 10 correct:
#     Input byte: b5
#     Output byte: 9b
#     Expected output byte: 9b
```

3. **Decryption Test:** The module is switched to decryption mode (enc\_dec set to 0). The previously encrypted output bytes are fed into the module as input, using the same approach used in the encryption phase. Each output byte is verified against the original plaintext bytes.

```
# Starting decryption test
#   (decryption of the bytes resulting
#   from the previous encryption)
# Decryption 1 correct:
#   Input byte: 3a
#   Output byte: 3c
#   Expected output byte: 3c
# Decryption 2 correct:
#   Input byte: ff
#   Output byte: 7f
#   Expected output byte: 7f
# Decryption 3 correct:
#   Input byte: 8d
#   Output byte: 9b
#   Expected output byte: 9b
# Decryption 4 correct:
#   Input byte: 03
#   Output byte: 5e
#   Expected output byte: 5e
# Decryption 5 correct:
#   Input byte: 61
#   Output byte: 1a
#   Expected output byte: 1a
# Decryption 6 correct:
#   Input byte: 73
#   Output byte: 9c
#   Expected output byte: 9c
# Decryption 7 correct:
#   Input byte: a5
#   Output byte: 2a
#   Expected output byte: 2a
# Decryption 8 correct:
#   Input byte: 14
#   Output byte: 12
#   Expected output byte: 12
# Decryption 9 correct:
#   Input byte: c3
#   Output byte: 39
#   Expected output byte: 39
# Decryption 10 correct:
#   Input byte: 9b
```

```
#           Output byte: b5
#           Expected output byte: b5
```

## 5.2 Testbench for Incorrect Usage (wrong\_use\_tb.sv)

---

This illustrates the misuse of the module where the `new_msg` and `in_valid` signals are asserted simultaneously. This testbench follows a similar architecture as the first one but highlights the potential pitfalls:

- **Testing Procedure:**

1. **Initialization:** Same as the first testbench, with clock and reset signals configured.
2. **Incorrect Encryption Test:** When `new_msg` is asserted high, the first message block is provided with `in_valid=1` and kept for 10 time units (one clock period), violating the proper timing protocol. As a result, the first byte will not be encrypted, and the actual encryption will start from the second byte, leading to incorrect ciphertext output. The output is checked against the expected values, demonstrating how the encryption fails when the module is misused.

```
# Starting encryption test
# Encryption 1 failed:
#     Input byte: 3c
#     Output byte: 00
#     Expected output byte: 3a
# Encryption 2 failed:
#     Input byte: 7f
#     Output byte: 79
#     Expected output byte: ff
# Encryption 3 failed:
#     Input byte: 9b
#     Output byte: 2d
#     Expected output byte: 8d
# Encryption 4 failed:
#     Input byte: 5e
#     Output byte: 86
#     Expected output byte: 03
# Encryption 5 failed:
#     Input byte: 1a
#     Output byte: 5e
#     Expected output byte: 61
# Encryption 6 failed:
#     Input byte: 9c
#     Output byte: c4
#     Expected output byte: 73
# Encryption 7 failed:
```

```

#       Input byte: 2a
#       Output byte: 36
#       Expected output byte: a5
# Encryption 8 failed:
#       Input byte: 12
#       Output byte: 17
#       Expected output byte: 14
# Encryption 9 failed:
#       Input byte: 39
#       Output byte: c9
#       Expected output byte: c3
# Encryption 10 failed:
#       Input byte: b5
#       Output byte: 68
#       Expected output byte: 9b

```

3. **Decryption Test:** Similar to the encryption test, but it checks whether decrypting the incorrectly generated ciphertext produces the expected original plaintext. It is expected that the output will differ from the original plaintext in the first two bytes.

```

# Starting decryption test
#   (decryption of the bytes resulting
#     from the previous encryption)
# Decryption 1 failed:
#       Input byte: 00
#       Output byte: 06
#       Expected output byte: 3c
# Decryption 2 failed:
#       Input byte: 79
#       Output byte: 1a
#       Expected output byte: 7f
# Decryption 3 correct:
#       Input byte: 2d
#       Output byte: 9b
#       Expected output byte: 9b
# Decryption 4 correct:
#       Input byte: 86
#       Output byte: 5e
#       Expected output byte: 5e
# Decryption 5 correct:
#       Input byte: 5e
#       Output byte: 1a
#       Expected output byte: 1a
# Decryption 6 correct:
#       Input byte: c4
#       Output byte: 9c

```

```
#           Expected output byte: 9c
# Decryption 7 correct:
#           Input byte: 36
#           Output byte: 2a
#           Expected output byte: 2a
# Decryption 8 correct:
#           Input byte: 17
#           Output byte: 12
#           Expected output byte: 12
# Decryption 9 correct:
#           Input byte: c9
#           Output byte: 39
#           Expected output byte: 39
# Decryption 10 correct:
#           Input byte: 68
#           Output byte: b5
#           Expected output byte: b5
```

---

# CHAPTER 6

---

## FPGA Implementation Results

---

### 6.1 Quartus steps

---

The first step was to run the analysis and synthesis of the module, followed by its fitting on the 5CGXFC9D6F27C7 FPGA device. Once the fitting process was complete, the Static Timing Analysis (STA) was performed to determine the maximum clock frequency supported by the module. The STA was conducted in two scenarios:

- **AES\_feedback\_cipher without virtual pins:** this is the scenario in which the module's input/output ports are connected directly to the input/output pins of the FPGA.
- **AES\_feedback\_cipher with virtual pins:** this configuration used virtual pins to emulate the scenario where the module is integrated into a more complex system. In such a system, the input/output ports of the module are connected to other internal logic blocks rather than to the FPGA's pins.

Since the input/output delays are shorter when using virtual pins, this configuration results in a higher maximum clock frequency. In both cases, the followed approach involved starting with a high clock frequency and gradually reducing it until the timing analysis passed without violations.

### 6.2 Synthesis and Fitting

---

After running the Analysis and Synthesis and Fitter (Place and Route) processes, the reports shown in Figure 6.1 and 6.2 were generated, displaying details such as the logic utilization (in ALMs) and the total number of registers.



Table of Contents	Flow Summary																																						
<ul style="list-style-type: none"> <li>Flow Summary</li> <li>Flow Settings</li> <li>Flow Non-Default Global Settings</li> <li>Flow Elapsed Time</li> <li>Flow OS Summary</li> <li>Flow Log</li> <li>Analysis &amp; Synthesis</li> <li>Fitter <ul style="list-style-type: none"> <li>Flow Messages</li> <li>Flow Suppressed Messages</li> </ul> </li> </ul>	<div>&lt;&lt;Filter&gt;&gt;</div> <table> <tr><td>Flow Status</td><td>Successful - Sun Oct 20 16:18:08 2024</td></tr> <tr><td>Quartus Prime Version</td><td>21.1.1 Build 850 06/23/2022 SJ Lite Edition</td></tr> <tr><td>Revision Name</td><td>aes_feedback_cipher</td></tr> <tr><td>Top-level Entity Name</td><td>aes_feedback_cipher</td></tr> <tr><td>Family</td><td>Cyclone V</td></tr> <tr><td>Device</td><td>5CGXFC9D6F27C7</td></tr> <tr><td>Timing Models</td><td>Final</td></tr> <tr><td>Logic utilization (in ALMs)</td><td>50 / 113,560 (&lt; 1 %)</td></tr> <tr><td>Total registers</td><td>18</td></tr> <tr><td>Total pins</td><td>30 / 378 (8 %)</td></tr> <tr><td>Total virtual pins</td><td>0</td></tr> <tr><td>Total block memory bits</td><td>0 / 12,492,800 (0 %)</td></tr> <tr><td>Total DSP Blocks</td><td>0 / 342 (0 %)</td></tr> <tr><td>Total HSSI RX PCSs</td><td>0 / 9 (0 %)</td></tr> <tr><td>Total HSSI PMA RX Deserializers</td><td>0 / 9 (0 %)</td></tr> <tr><td>Total HSSI TX PCSs</td><td>0 / 9 (0 %)</td></tr> <tr><td>Total HSSI PMA TX Serializers</td><td>0 / 9 (0 %)</td></tr> <tr><td>Total PLLs</td><td>0 / 17 (0 %)</td></tr> <tr><td>Total DLLs</td><td>0 / 4 (0 %)</td></tr> </table>	Flow Status	Successful - Sun Oct 20 16:18:08 2024	Quartus Prime Version	21.1.1 Build 850 06/23/2022 SJ Lite Edition	Revision Name	aes_feedback_cipher	Top-level Entity Name	aes_feedback_cipher	Family	Cyclone V	Device	5CGXFC9D6F27C7	Timing Models	Final	Logic utilization (in ALMs)	50 / 113,560 (< 1 %)	Total registers	18	Total pins	30 / 378 (8 %)	Total virtual pins	0	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 %)
Flow Status	Successful - Sun Oct 20 16:18:08 2024																																						
Quartus Prime Version	21.1.1 Build 850 06/23/2022 SJ Lite Edition																																						
Revision Name	aes_feedback_cipher																																						
Top-level Entity Name	aes_feedback_cipher																																						
Family	Cyclone V																																						
Device	5CGXFC9D6F27C7																																						
Timing Models	Final																																						
Logic utilization (in ALMs)	50 / 113,560 (< 1 %)																																						
Total registers	18																																						
Total pins	30 / 378 (8 %)																																						
Total virtual pins	0																																						
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 6.1:** Flow Summary

The Flow Summary section provides a comprehensive overview of the resource utilization within the design. This includes the total number of registers, logic elements, and pin usage.

Table of Contents

Flow Log

Analysis & Synthesis

Fitter

Summary

Settings

Parallel Compilation

Netlist Optimizations

Incremental Compilation Section

Pin-Out File

Resource Section

Resource Usage Summary

Partition Statistics

Input Pins

Output Pins

I/O Bank Usage

All Package Pins

I/O Standards Section

Resource Utilization by Entity

Fitter Resource Usage Summary

<<Filter>>

	Resource	Usage	%
1	Logic utilization (ALMs needed / total ALMs on device)	50 / 113,560	< 1 %
2	ALMs needed [=A-B+C]	50	
1	[A] ALMs used in final placement [=a+b+c+d]	51 / 113,560	< 1 %
1	[a] ALMs used for LUT logic and registers	5	
2	[b] ALMs used for LUT logic	46	
3	[c] ALMs used for registers	0	
4	[d] ALMs used for memory (up to half of total ALMs)	0	
2	[B] Estimate of ALMs recoverable by dense packing	1 / 113,560	< 1 %
3	[C] Estimate of ALMs unavailable [=a+b+c+d]	0 / 113,560	0 %
1	[a] Due to location constrained logic	0	
2	[b] Due to LAB-wide signal conflicts	0	
3	[c] Due to LAB input limits	0	
4	[d] Due to virtual I/Os	0	
3			
4	Difficulty packing design	Low	
5			
6	Total LABs: partially or completely used	6 / 11,356	< 1 %
1	-- Logic LABs	6	

**Figure 6.2:** Fitter Resource Usage Summary

In addition to the Flow Summary, the Fitter Resource Usage Summary report provides a more detailed break down of logic utilization. This includes specific information on how many ALMs (Adaptive Logic Modules) are used for different purposes: the ALMs utilized for both LUTs and registers, the ALMS dedicated solely to LUT logic, and those for registers only.

These results are important to better outline the benefits and drawbacks of the presented Feedback Cipher RTL Design. From these data, it can be said that:

- **Logic Utilization:** The cipher uses less than 1% of all the ALMs (50 out of

113560) available, which indicates a lightweight design that can be easily scaled up or can see the addition of additional encryption layers with the possibility of using the same FPGA Device.

- **Registers:** The cipher uses 18 registers, which again suggests a lightweight design with the usage of a minimal fraction of the total resources available.
- **Pins:** A drawback can be observed with reference to the pin usage. It can be seen that the cipher uses 30 out of the 378 available pins, corresponding approximately to the 8% of the total availability. This suggests an higher I/O communication activity employed by the design compared to the logic utilization results.

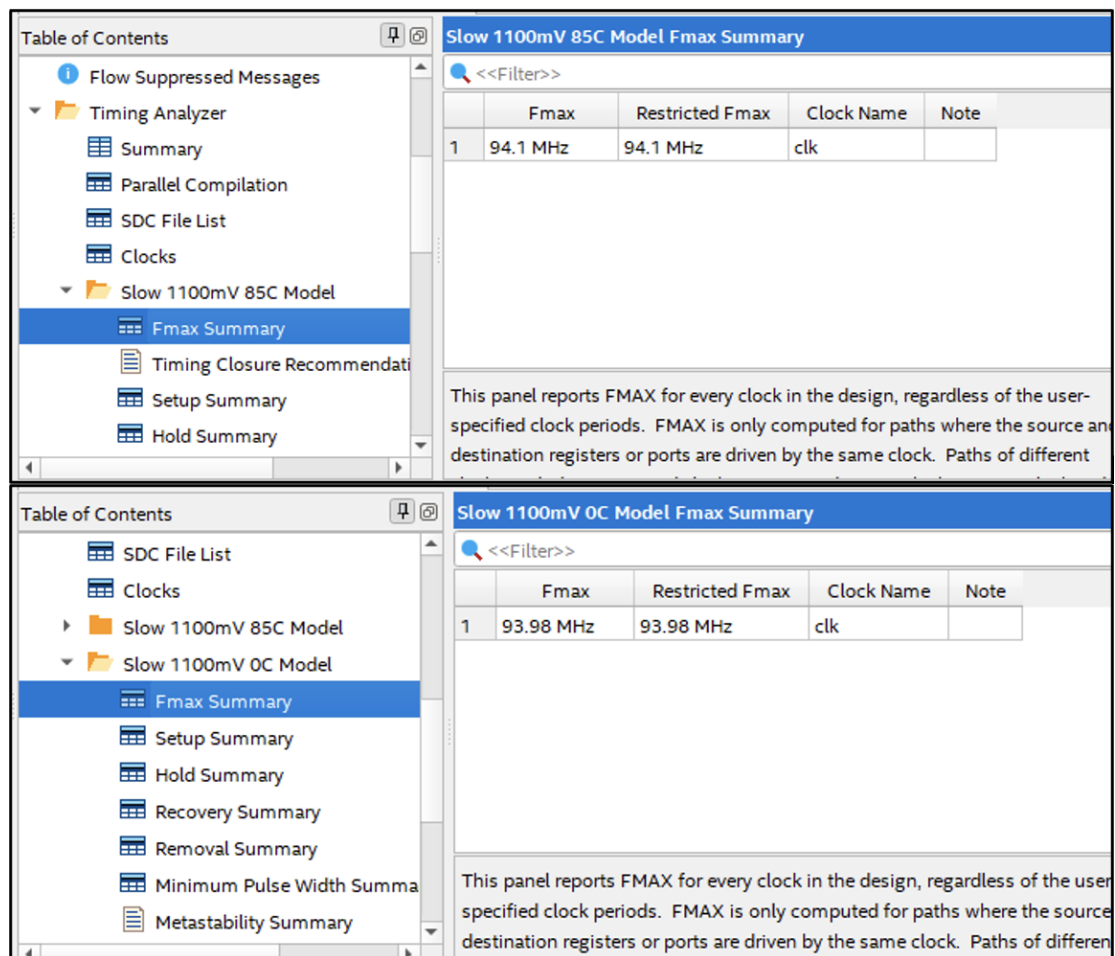
This Feedback Cipher design can be seen as a resource-efficient cipher, which has a very low logic usage percentage of the FPGA device used, describing it as a flexible and scalable design for future iterations. An eye for improvements can be applied to the I/O logic, in case the design would need the inclusion of other modules, notably in the case of restricted I/O resources availability.

### 6.3 Static Timing Analysis results

The Static Timing Analysis (STA) revealed a maximum clock frequency of 93.98 MHz (corresponding to a period of 10.75 nanoseconds) for the configuration of the cipher without virtual pins. In contrast, the maximum frequency increased to 150.69 MHz (period of 6.65 nanoseconds) when virtual pins were used. For the configuration without virtual pins, the constraints provided to Quartus through SDC files to achieve the reported result are the following:

```
create_clock -name clk -period 10.75 [get_ports clk]
set_false_path -from [get_ports rst_n] -to [get_clocks clk]
set_input_delay -min 1.075 [all_inputs] -clock [get_clocks clk]
set_input_delay -max 2.15 [all_inputs] -clock [get_clocks clk]
set_output_delay -min 1.075 [all_outputs] -clock [get_clocks clk]
set_output_delay -max 2.15 [all_outputs] -clock [get_clocks clk]
```

The STA results obtained from Quartus for this configuration are presented in Figure 6.3. These results indicate the maximum frequency for two corner cases. The maximum clock frequency for this configuration of the module is determined to be 93.98 MHz, corresponding to the lowest one (worst-case condition).

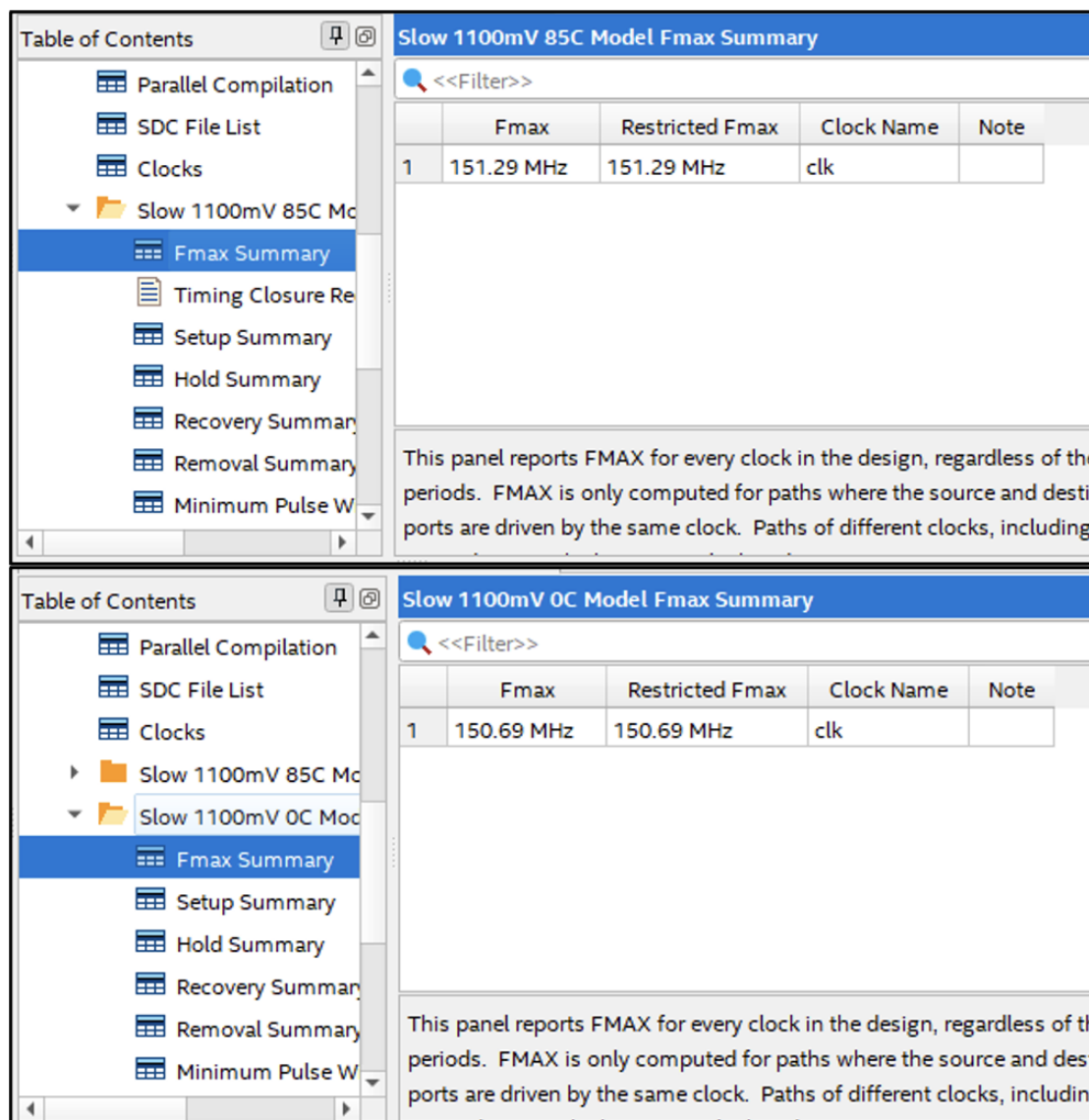


**Figure 6.3:** STA results without virtual pins

For the cipher configured with virtual pins, the constraints provided to Quartus were updated as follows:

```
create_clock -name clk -period 6.65 [get_ports clk]
set_false_path -from [get_ports rst_n] -to [get_clocks clk]
set_input_delay -min 0.665 [all_inputs] -clock [get_clocks clk]
set_input_delay -max 1.33 [all_inputs] -clock [get_clocks clk]
set_output_delay -min 0.665 [all_outputs] -clock [get_clocks clk]
set_output_delay -max 1.33 [all_outputs] -clock [get_clocks clk]
```

The STA results for this configuration are shown in Figure 6.4. As you can see, the analysis highlights two corner cases. As done in the previous case, the lower frequency of 150.69 MHz is considered.



**Figure 6.4:** STA results with virtual pins