AES Encryption in FPGA hardware

Patrik Dahlström
Electronic Design

Daniel Josefsson
Electronic Design

Staffan Sjöqvist
Electronic Design

January 19, 2012

Abstract

The steady incline of valuable and vulnerable data, transferred between an increasing amount of nodes, requires a swift, secure and cost effectively encryption solution to keep the information safe. A dedicated FPGA solution fills these requirements and also; your more valuable components can concentrate on their special abilities.

This project report describes how we implemented AES encryption and decryption routines on a FPGA, written in VHDL. We also present how we used test benches for testing component and subcomponent, to verify all parts of the implementation. This report focuses on the encryption routine, but the full source code implements both encryption and decryption. The source code is available at `https://github.com/Risca/Fierce-Gravel`.

The result of the project is two fully functional algorithms without any delays other than gate delays. The implementations are easy to expand so they can be implemented in a transmission circuit, or improved by adding, for example, pipelining.

Chapter 1

Introduction

AES encryption is a widely recognized standard encryption that is well used in many modern applications of today. It can be found not only in modern wireless networks, but also in encryption devices designed for secure storage as well as in secure wired networks.

The project group's intention is to learn how this encryption works and how to implement it in FPGA hardware.

Chapter 2

Definitions

Before describing the AES encryption, some key definitions need to be mentioned.

2.1 Addition

In the AES standard, addition is performed bitwise with the XOR operator. No regard is taken to carry bits. Consequently, subtraction is identical to addition.

2.2 Multiplication

In the AES standard, multiplication is performed by using the multiplication of polynomial modulo an irreducible polynomial of degree 8, $\mathrm{GF}\left(2^8\right)$ where GF is short for Rijndael's finite field or Galois field.[**?**][**?**] This means that multiplication is performed by using addition defined in **??** between the resulting terms and then dividing the result with a given polynomial.

The polynomia used in the AES standard is

$$x^8 + x^4 + x^3 + x + 1 = 11B_{16} = 283_{10}. \tag{2.1}$$

2.3 Cipher key

A cipher key is a secret "passphrase" shared between the encryption and decryption side. It is the single most important part of a crypto; once the cipher key is known, all data encrypted with that key can be decrypted.

2.4 Key length

The key length determines how many bits that are used to encrypt input data. The AES standard defines 3 key lengths: AES-128, AES-192 and AES-256. The

trailing number defines the key length in bits.

The key length is commonly used to denote the strength of the AES encryption.

2.5 Block

Data and keys are organized in 4xN matrices called blocks (Figure **??**). Each element defines 1 byte and each column is 1 word (32 bits). A block is read one column at a time and each column from the top down.

W_0	W_1	$\cdots$	W_n
B_0	B_0	$\cdots$	B_0
B_1	B_1	$\cdots$	B_1
B_2	B_2	$\cdots$	B_2
B_3	B_3	$\cdots$	B_3

Figure 2.1: Block definition

Input, state and output data is always in 4x4 blocks (128 bits) while cipher keys can have varying lengths (see section **??**).

2.6 Round

Encryption of data works by manipulating the input data in an iterative process where each iteration is called a round. The number of rounds needed is determined by the cipher key length.

To calculate the number of rounds needed for a specific key length, (**??**) can be used

$$N_r = 6 + \frac{A_l}{W_l} \tag{2.2}$$

where A_l and W_l denotes key length (128, 192 or 256 bit) and word length (32 bit) respectively.

2.7 Round key

For each round in the encryption process a different round key is used. In total there are $N_r + 1$ round keys. Section **??** contain more information about round keys and how they are derived.

2.8 State block or state array

Data to be encrypted (input data) is copied to a state block. During intermediate encryption rounds, data is manipulated on the state block. After the final round the state block is copied to the output block.

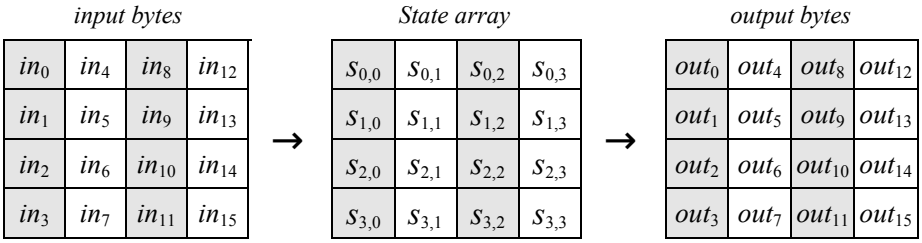

Figure 2.2: State array input and output.[**?**]

Chapter 3

Theory

The AES encryption process can be divided in two different sections:

- Encryption process
- Key schedule

In encryption process, an unencrypted data packet is encrypted using a cipher key and in key schedule the different round keys needed for the encryption process are generated.

3.1 Encryption process

Encrypting data with the AES algorithm begins by copying the input data to a state block and then adding the first round key. The state block is then transformed in $N_r - 1$ rounds, where each round consists of four transformations:

1. Substitution using a substitution table
2. Row shifting
3. Column mixing
4. Round key addition

In the final round, no column mixing is performed since it does not add to the security of the encryption.

As the last step, the state block is copied to the output block.

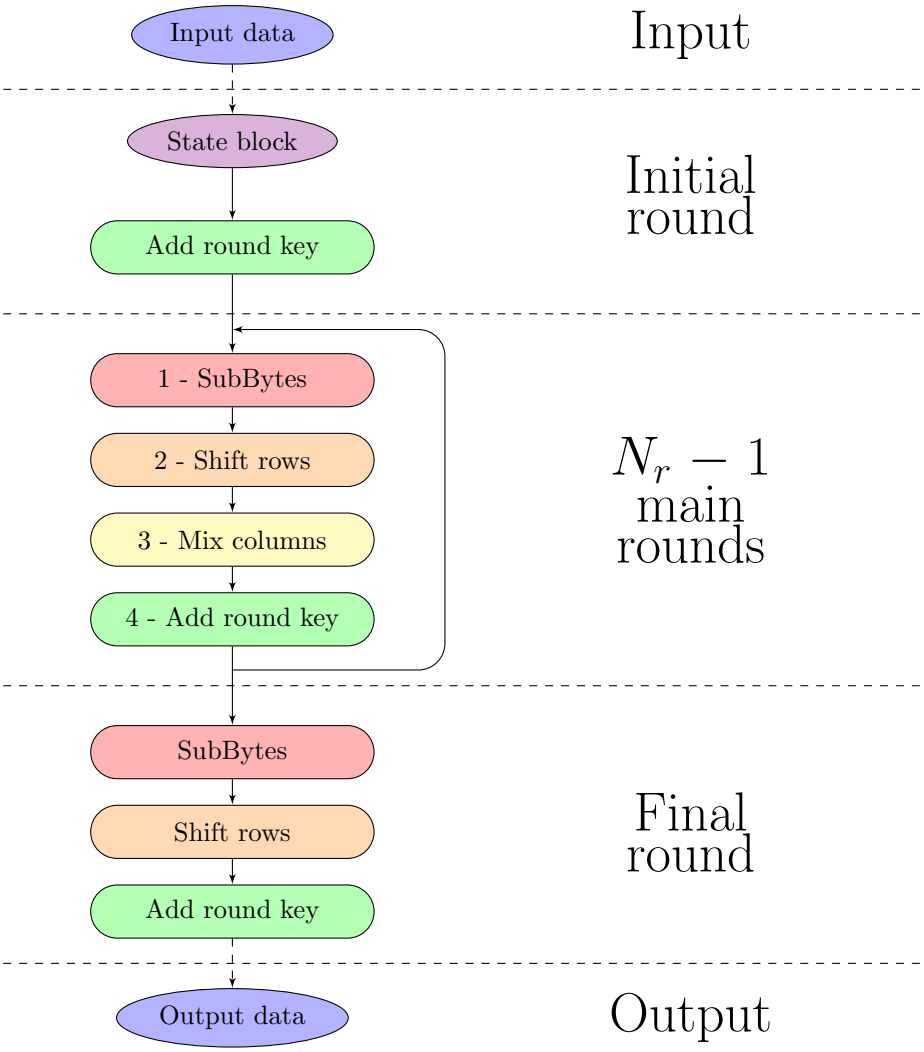

Figure 3.1: Encryption process overview

3.1.1 Substitution using a substitution table

Each byte in a state block is represented by two hexadecimal digits. These two digits are then used as row and column index in the substitution table in figure **??**.

For example, the value 42_{10} is $2A_{16}$ in hexadecimal form and would be transformed to $E5_{16}$.

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

Figure 3.2: Substitution table.[**?**]

This transformation is performed on every byte in a state block.

3.1.2 Row shifting

Row shifting means that each row in a state block is cyclically shifted left N times, where N is the row number - 1, i.e. the first row is shifted 0 times, the second row 1 time, etc.

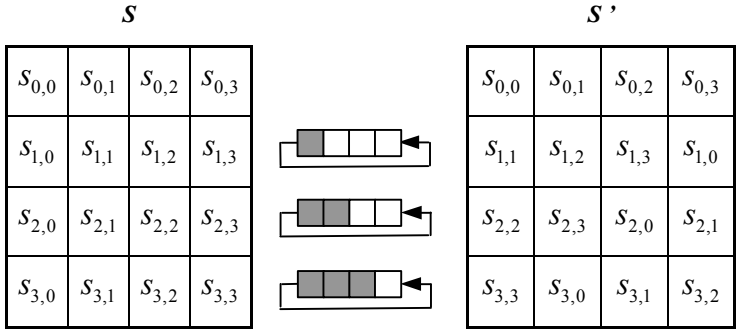

Figure 3.3: Illustration of row shifting.[**?**]

3.1.3 Column mixing

The mix column transformation is done by multiplying, using the Galois Field multiplication described in section **??**, each column by a given matrix as shown in equation **??** below.

$$\begin{bmatrix} S'_{0,c} \\ S'_{1,c} \\ S'_{2,c} \\ S'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \bullet \begin{bmatrix} S_{0,c} \\ S_{1,c} \\ S_{2,c} \\ S_{3,c} \end{bmatrix} \text{ for } 0 \leq c \leq 3 \tag{3.1}$$

3.1.4 Round key addition

The last step of every round is to add the round key to the state block.

$$\mathbf{S}' = \overbrace{\begin{bmatrix} S_{0,0} & S_{0,1} & S_{0,2} & S_{0,3} \\ S_{1,0} & S_{1,1} & S_{1,2} & S_{1,3} \\ S_{2,0} & S_{2,1} & S_{2,2} & S_{2,3} \\ S_{3,0} & S_{3,1} & S_{3,2} & S_{3,3} \end{bmatrix}}^{\text{State block}} \bigoplus \overbrace{\begin{bmatrix} W_{0,0} & W_{0,1} & W_{0,2} & W_{0,3} \\ W_{1,0} & W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,0} & W_{2,1} & W_{2,2} & W_{2,3} \\ W_{3,0} & W_{3,1} & W_{3,2} & W_{3,3} \end{bmatrix}}^{\text{Round key}} \tag{3.2}$$

3.2 Key Schedule Expansion

The AES standard uses a cipher key which is determined by the implementers or users of the standard and a key schedule to compute the rest of the keys used in the encryption process. The predetermined cipher key is used in the first iteration and the round keys are used from the second iteration to the final iteration.

The round keys have the same length as the original cipher and are all generated by information on the former round key. The key expansion process is divided into four different parts: word rotation, word substitution, round constant addition and the procedure of generation.

3.2.1 Word Rotation

The Word Rotation function is a cyclic single right byte shift where the least significant byte is shifted to the place of the most significant byte (see fig. **??**).

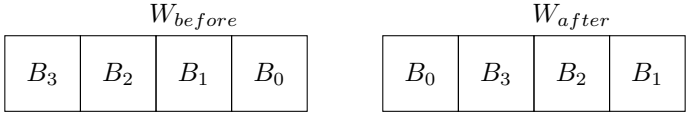

Figure 3.4: Graphical interpretation of the word rotation function.

3.2.2 Round Constant Array

The round constant array is defined as a word array where the least significant byte is defined as

$$\text{rcon}\,(i) = x^{i-1} \tag{3.3}$$

and the other bytes is filled with zeros and where x^{i-1} is calculated using the multiplication defined in **??** and i denotes the round. A figure showing the interesting bytes of this array is given in Figure **??**.

$W_{1,0}$	$W_{2,0}$	$W_{3,0}$	$W_{4,0}$	$W_{5,0}$	$W_{6,0}$	$W_{7,0}$	$W_{8,0}$	$W_{9,0}$	$W_{10,0}$	$W_{11,0}$	$W_{12,0}$	$W_{13,0}$	$W_{14,0}$
01	02	04	08	10	20	40	80	$1B$	36	$6C$	$D8$	AB	$4D$

Figure 3.5: Least significant byte row of the round constant array (128 bit).

Note that the length of the array is determined of the strength of the encryption.

3.2.3 Word Substitution

The word substitution process follows the same pattern as in **??** and every byte in the word is substituted according to the look-up table given in **??**.

3.2.4 Round Key Generation

The round keys are generated by performing the above mentioned steps in a special sequence:

The first column in the first round key is generated by performing word rotation on the last column in the cipher key followed by word substitution and addition of the first round constant.

The next column in the round key is generated by adding the second column in the cipher key to the previously generated round key column, and so on for the rest of the columns in the round key.

The first column in the rest of the round keys is generated using the last column in the previous round key.

Chapter 4

Implementation Details

Approximately each section in chapter **??** is translated into a separate VHDL component. Some components are in themselves also divided in subcomponents to make the code more manageable. The complete and most up-to-date code is found on `https://github.com/Risca/Fierce-Gravel`.

Since almost each operation in chapter **??** is a separate component and the fact that the source code available on-line is well commented, this section will only highlight the topics of the VHDL implementation that require further explanation. These topics include:

- The *Resources* package
- The *Variables* Package
- Test Benches

4.1 The *resources* and *variables* packages

The VHDL standard defines an elegant way to summarize all different types, subtypes, functions, components, etc. in what is called a *package*. This project have two packages for all its parts — The *resources* and the *variables* package.

4.1.1 *Resources* package

This package contains type and component declarations, and a few functions.

The type definitions were introduced to make it clear what signals the input and outputs of each component are supposed to be fed. Listing **??** show the six different types used throughout the project.

The component declarations are not very informative to list here and the reader is therefore referred to the on-line source.

To be able to use the type definitions above, the XOR operator function had to be overloaded.

Listing 1: FG_package.vhd, type definitions

```
package resources is

  -- TYPE DEFINITIONS
  type byte is array (7 downto 0) of std_logic;
  -- To index column Foo: Foo(byte)(bit);
  type column is array (0 to 3) of byte;
  -- To index state_array Foo: Foo(column)(byte)(bit);
  type state_array is array (0 to 3) of column;

  subtype round_key is state_array;
  -- Nr+1 round keys
  type round_keys_t is array (0 to Nr) of round_key;
  type cipher_key is array (0 to 4*Nk-1) of column;
```

The round constant is also defined as a function, mainly because of simplicity.

Listing **??** and listing **??** show the function declarations and definitions respectively.

Listing 2: FG_package.vhd, function declarations

```
  -- OVERLOADED OPERATORS
  function "xor" (L,R : byte) return byte;
  function "xor" (L,R : column) return column;
  function "xor" (L,R : state_array) return state_array;
  -- function "xor" (L : state_array; R : round_key) return
      state_array;
  function rcon(INPUT : integer range 0 to 15) return column;
```

Listing 3: FG_package.vhd, function definitions

```
package body resources is
  function "xor" (L,R : byte) return byte is
    variable result : byte;
  begin
    for n in 7 downto 0 loop
      result(n) := L(n) xor R(n);
    end loop;
    return result;
  end function;

  function "xor" (L,R : column) return column is
    variable result : column;
  begin
    result(3) := L(3) xor R(3);
    result(2) := L(2) xor R(2);
    result(1) := L(1) xor R(1);
    result(0) := L(0) xor R(0);
    return result;
  end function;

  function "xor" (L,R : state_array) return state_array is
    variable result : state_array;
```

```
184   begin
        result(3) := L(3) xor R(3);
186     result(2) := L(2) xor R(2);
        result(1) := L(1) xor R(1);
188     result(0) := L(0) xor R(0);
        return result;
190   end function;

192 --  function "xor" (L : state_array; R : round_key) return
      state_array is
--      variable result : state_array;
194 --  begin
--      result(3) := L(3) xor R(3);
196 --      result(2) := L(2) xor R(2);
--      result(1) := L(1) xor R(1);
198 --      result(0) := L(0) xor R(0);
--      return result;
200 --  end function;

202   function rcon(INPUT : integer range 0 to 15) return column is
      type rconROM_type is array (0 to 15) of byte;
204   constant rconROMTable : rconROM_type :=(
      --    0       1       2       3       4       5       6       7
206     x"00", x"01", x"02", x"04", x"08", x"10", x"20", x"40",
      --    8       9       a       b       c       d       e       f
208     x"80", x"1B", x"36", x"6C", x"D8", x"AB", x"4D", x"9A"
      );
210   begin
        return (rconROMTable(INPUT),x"00",x"00",x"00");
212   end function;
    end package body resources;
```

4.1.2 *Variables* package

This package is not part of *resources* mostly for convenience. I gathers all the constants and variables needed throughout the project, and can be seen in listing **??**

Listing 4: FG_variables.vhd, Constants and variables used

```
 6  package variables is
      -- This package contains variables (constants) that is needed to
 8    -- change the function of the algorithm.

10    -- CONSTANTS
      -- Set the key length
12    -- 128, 192 or 256
      constant key_length : integer := 128;
14      -- Block length
      constant Nb : integer := 4;
16      -- Number of columns defined by key
      constant Nk : integer := key_length/32;
18      -- Number of rounds
      constant Nr : integer := 6+Nk;
20
    end package variables;
```

4.2 Test benches

To facilitate testing of each component during this project, most components take at least a state_array as an input and a state_array as an output. Some components also accept a round_key. This generalized input and output make it very easy to test each component, or even operator, individually.

Three test benches are pre-made in the online sources

- AES_enc.vhd – For testing the encryption and it's components, subcomponents and operators
- AES_dec.vhd – For testing the decryption and it's components, subcomponents and operators
- KeySchExp.vhd – For testing the key expansion

The test benches have an input for selecting a few pre-programmed sets of input data. The data can be changed at compile-time.

All test benches also have outputs for viewing the output data on the 7-segment displays on the Altera DE2 development board. The 7-segment displays can display one column (four bytes) at a time. Two switches are used to select which column of the output that should be displayed.

See figure **??** for a schematic overview on how the test benches work.

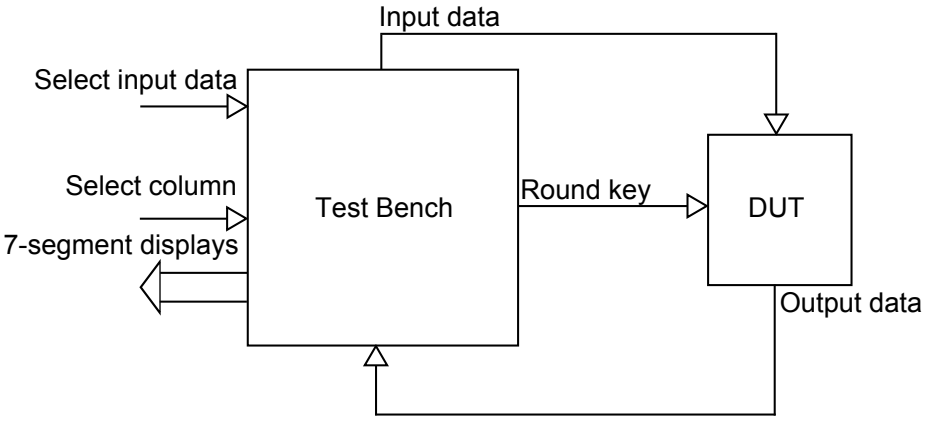

Figure 4.1: Test bench overview

Chapter 5

Future improvements

This implementation of the AES encryption is not without flaws. There are some parts that can be improved by further work. These parts include:

- The ability to send data, and then receive the encrypted/decrypted data, through the serial port.[1]
- A more pipelined algorithm to increase throughput.
- A more controlled encryption/decryption process. Signals should be available to indicate when the key expansion and the encryption/decryption process are complete.
- An even more secure pipelined algorithm by implementing e.g. cipher-block chaining.
- The current implementation can be further optimized for size and speed.

[1] It should be fairly easy to implement serial communication. Components for sending and receiving data is available, and tested, in the online sources. However, they are not used for the encryption/decryption processes at the time of writing.

Chapter 6

Conclusion

The Advanced Encryption Standard is a widely used and trusted encryption standard with no known major weaknesses.[**?**] It is fast, simple and very secure. It is used, among others, by the U.S. government to secure TOP SECRET information and can be considered the most widespread encryption standard used today.

AES describes an iterating encryption/decryption process that is very simple and at the same time very fast. This makes it easy to implement and optimize the AES.

The implementation described within this document and online (available at `https://github.com/Risca/Fierce-Gravel/`) show how a possible implementation using FPGA hardware can be achieved.

Bibliography

[1] *FIPS PUB 197: the official AES standard*, 2001. Available from: `http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf` [updated January 7, 2012]

[2] Wikimedia Foundation Inc. Encyclopedia on-line, *Finite field arithmetic: Rijndael's finite field*. Available from: `http://en.wikipedia.org/wiki/Finite_field_arithmetic#Rijndael.27s_finite_field` [updated January 8, 2012]

[3] Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger, *Biclique Cryptanalysis of the Full AES*, 2011. Available from: `http://research.microsoft.com/en-us/projects/cryptanalysis/aesbc.pdf` [updated January 7, 2012]