Final 83-004 Verilog Project

Dear Students,

During the last semester you have gained understanding and some (small) practical in-Lab experience with *Designing-With-Verilog HDL*. The home-works and tasks you got so far were quite small and tailored to gain understanding in the basic syntax, operators, procedural, behavioral and structural constructs of Verilog. In this project you will design a real system.

You will design the system in the Labs (as scheduled) for ~4 sessions whereas the remaining work will be by your own. The submission deadline is until 8.7.16 (~3 weeks after the semester ends). The project along with functionally verified (test-benches) and wave-forms will be submitted through the Moodle site.

Through this final project (which will reflect 50% of your grade) we have 3 goals:

1. Giving you the first Engineer experience in which you receive a spec (specification) of a system you do not know or have prior knowledge of, and getting you to understand the system constraints, structure and operation on your own!! Though this part seems like it has no close contact with Verilog, the contrary is correct, this is exactly what is expected from a designer (an engineer) to do. To thoroughly read and understand the system and the structure, to analyze the critical design points, to design architecture and to think how best will the system be coded in Verilog.
2. Getting you to define system ports and communication busses and interface, define architecture and state-machine if necessary.
3. Having you build the systems top level and individual blocks in Verilog. And verify the system through a test-bench.

This year's project was selected to design quite a simple system which is the commonly used algorithm for encryption called the AES. That is, the system receives a word (called plaintext) which is typically 128-bit wide and produces a 128-bit encrypted word (called ciphertext).

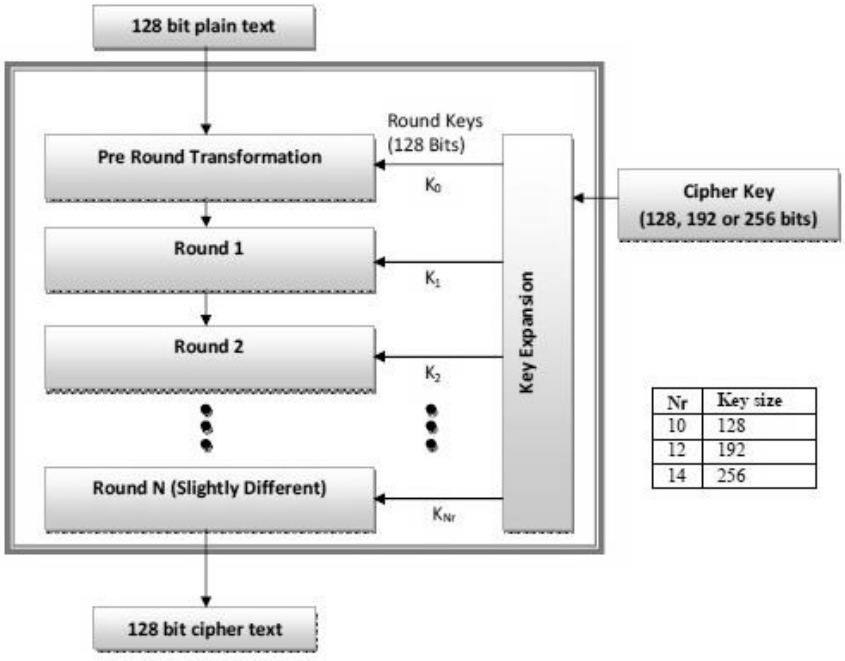
We will give you a general description of the system here to start with.

The Advanced Encryption Standard (AES) in general, architecture:

In general, our goal is that you will read the AES published spec which was uploaded for you and given here and try to understand on your own the system structure you need to build: http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf.

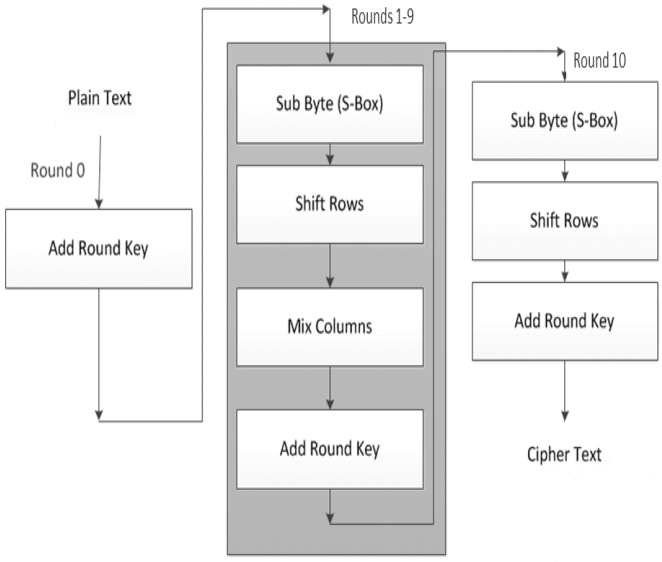
However, to make it a little bit user friendly we give you some explanation on the structure of the system. The AES is everywhere, in your cell phone, computer, implemented in software when you connect to the internet for secured sites, connect to VPN and in hardware, in your credit card etc.

The general structure of the protocol is as follows. The algorithm works in **rounds** we start with 128 bit ***plaintext*** (input data). To encrypt we also have an input secret key used for encryption (which the no. of bits can vary and it impacts the no. of rounds as will be explained). The first and last rounds are a bit different than the other **rounds**. In this scheme we have N rounds therefore round 1 and N are different. The key which is an input to each round is some transformation on the previous round key where the block which produces it entitled ***Key Expansion***, therefore in each round we have a different key (denoted K_0, K_1,…). If the no. of bits in a key is 128 we need 10 rounds, that is N=10, if 256 bits key N=14 as detailed in the table. In this project you will work with 128 bit key and input and with 10 rounds.

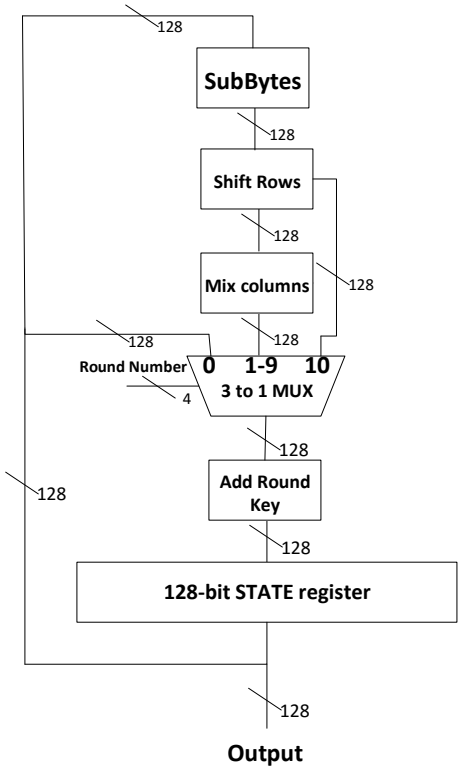


Nr	Key size
10	128
12	192
14	256

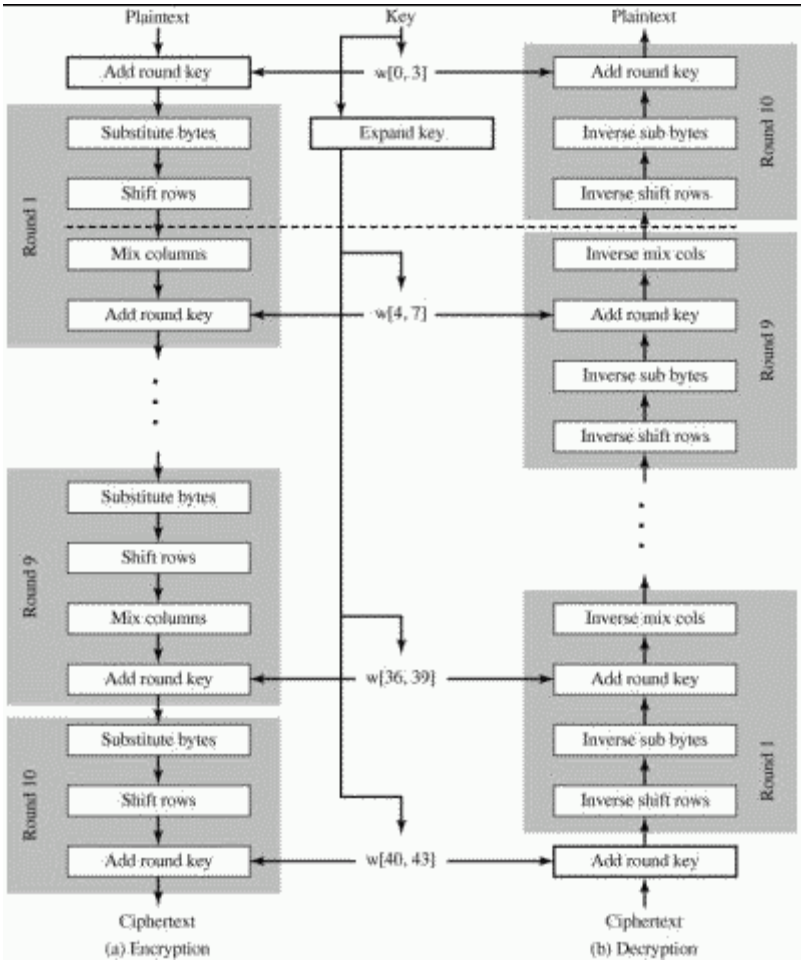
In the next figure we show the same scheme but more detailed. Where for the case of 128-bit key we have 10 round (starting from 0 in the scheme). The first and last round are separated from rounds 1-9. In this scheme we do not show the Key Expansion module. We see that the system is basically composed of 4 main components: ***AddRoundKey***, ***SubByte(S-Box)***, ***ShiftRows*** and ***MixColumns*** to finally produce the ***ciphertext***.



It is important to note that the system can be built in a pipeline structure of 2 forms, ***folded*** and ***unfolded.*** For example, the last scheme can be realized in a block level architecture like the one follows. Where, there exist one state register of 128 bits and according to the mux control signal which provides with the round no. we will perform only the ***AddRoundKey*** (for 0)**,** the whole four modules serially (for 1-9) and only the sub-set of three modules (for round 10). The advantages of this scheme are clearly minimum area and consequently power however what is the ***Throughput*** and ***Latency***???? (answer this question, recall in the course *Logic-Design*). This design is denoted by ***folded*** design. Some of you will implement this design.



The next scheme (the left part is encryption) illustrates an ***unfolded*** architecture where we are not reusing the same blocks for whole of the rounds of the encryption. In this architecture we are concatenating the different rounds hardware to provide with 10 serial rounds. That is, we pay by ~X10 in hardware, area and power. Now, this architecture can be pipelined for each stage of the computation to have a register at the output or we can only have registers at the inputs and outputs of the whole system. Answer the next question: what will be the latency and throughput of a: (a) completely ***pipelined unfolded*** design and of (b) ***unfolded not pipelined*** design (assume registers only at the beginning and end)???



The individual modules to design:

We will now detail on individual units that are needed to construct the AES.

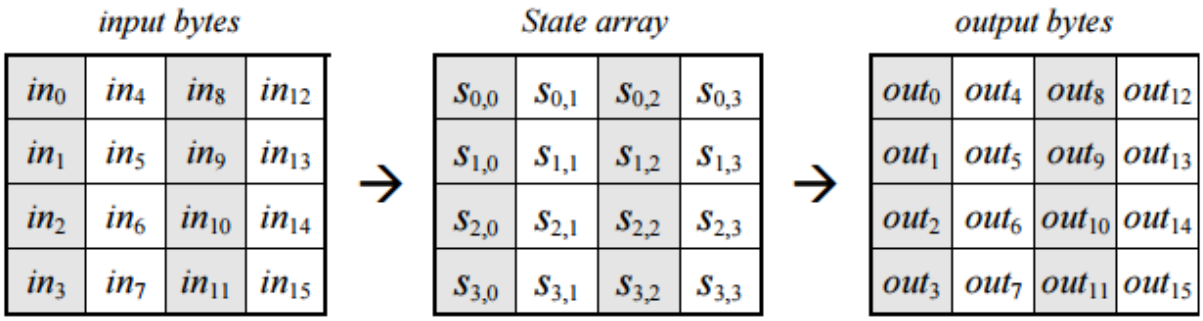
1. ***AddRoundKey***
2. ***SubByte(S-Box)***
3. ***ShiftRows***
4. ***MixColumns***
5. ***KeyExpansion***

Note that some of the modules are not "flat" and have hierarchy.

1. ***AddRoundKey*** *module* in details:

Each word of the plaintext and roundkey which are (for all our projects) 128-bit (with 10 rounds) wide are conceptually divided into segments of bytes (8bits) that is we have 16 such bytes in a full-word. For illustration proposes we place these 16 bytes in a 4X4 matric.

Internally, the AES algorithm's operations are performed on a two-dimensional array of bytes called the State. The State consists of four rows of bytes, each containing Nb=4 bytes, where Nb is the block length divided by 32. In the State array denoted by the symbol s, each individual byte has two indices, with its row number r in the range 0 to 4 and its column number c in the range 0 to Nb=4. This allows an individual byte of the State to be referred to as either $s_{r,c}$ or s[r,c]. For this standard, Nb=4. At the start of the Cipher, the input – the array of bytes in0, in1, … in15 (16 bytes)– is copied into the State array as illustrated below. The Cipher operations is then conducted on this State array, after which its final value is copied to the output – the array of bytes out0, out1, … out15. Therefore at the beginning of the cipher we have a mapping of s[r, c] = in[r + 4c]. and at the end of the cipher we have a mapping of out[r + 4c] = s[r, c]. **Therefore, our design parameters are no. of columns Nb=4, no. of rows Nk=4 and no. of rounds Nr=10.**

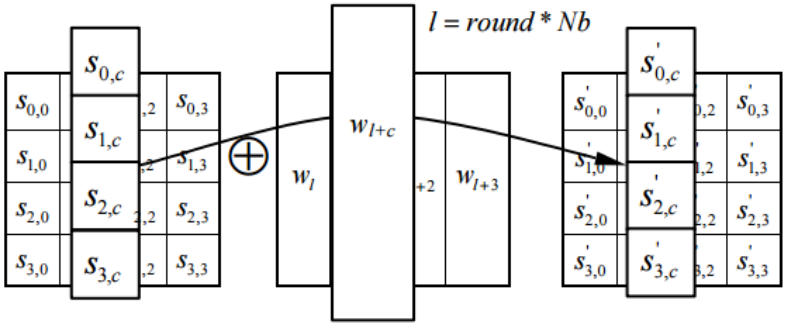


In the ***AddRoundKey*** transformation, a RoundKey is added to the State by a simple bitwise XOR operation.

The four bytes in each column of the State array form 32-bit words, where the row number r provides an index for the four bytes within each word. The state can hence be interpreted as a one-dimensional array of 32 bit words (columns), w0...w3, where the column number c provides an index into this array. Hence, for the example in the above figure, the State can be considered as an array of four words, as follows:

w0 = s0,0 s1,0 s2,0 s3,0 w2 = s0,2 s1,2 s2,2 s3,2 w1 = s0,1 s1,1 s2,1 s3,1 w3 = s0,3 s1,3 s2,3 s3,3.

The ***AddroundKey*** in this case works according to:



`AddRoundKey()` XORs each column of the State with a word from the key schedule.

Where in each ***KeyExpansion*** round we get a 128-bit new key which is divided into 4 words 32-bit each. For the first round (round 0) l=0 so we have 4 words W_0, W_1, W_2, W_3 , c is in the range 0:3.

$$[s'_{0,c}, s'_{1,c}, s'_{2,c}, s'_{3,c}] = [s_{0,c}, s_{1,c}, s_{2,c}, s_{3,c}] \oplus [w_{round*Nb+c}] \quad \text{for } 0 \le c < \mathbf{Nb},$$

And the XOR is performed column wise to produce the output.

Provide with a combinational module for the ***AddRoundKey***.

2. ***SubBytes(S-Box)*** *module* in details:

This module has unique cryptographic features which are not detailed here (you are encourage to take Dr. Hazai course introduction to Cryptography for the purpose). We detail here the operation of the module through a lookup table.

The S-box used in the ***SubBytes*** transformation is presented in hexadecimal form in the following figure. For example, if $s_{1,1}$ ={53}, then the substitution value would be determined by the intersection of the row with index '5' and the column with index '3'. This would result in $s'_{1,1}$ having a value of {ed}:

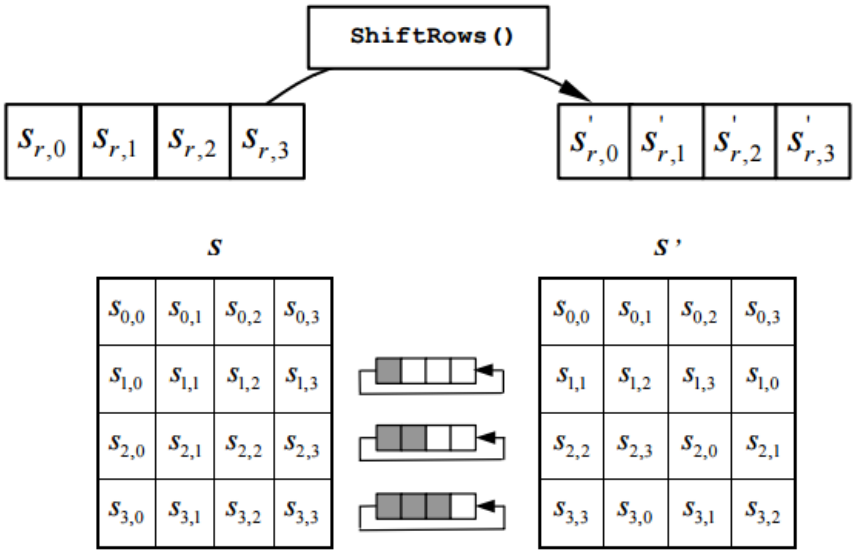
		y															
		0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
x	0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
	1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
	2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
	3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
	4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	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

S-box: substitution values for the byte `xy` (in hexadecimal format).

The implementation of this logic Verilog wise should be purely combinatorial. That is, you can implement this in any desired combinational form for example in a switch-***case*** based *module*.

3. ***ShiftRows*** *module* in details:

In the ***ShiftRows*** transformation, the bytes in the last three rows of the State are cyclically shifted over different numbers of bytes (offsets). The first row, r = 0, is not shifted. That is best visually described by:

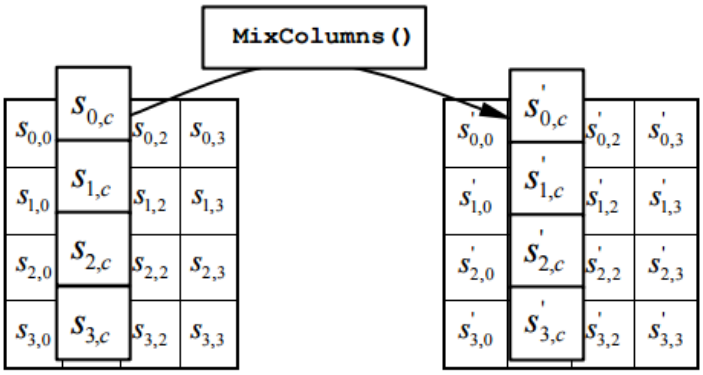


4. ***MixColumns*** *module* in details:

This step replaces each byte of a column by a function of all the bytes in the same column. More precisely, each byte in a column is replaced by two times that byte, plus three times the next byte, plus the byte that comes next, plus the byte that follows. During this operation, each column is transformed using a fixed matrix (matrix multiplied by column gives new value of column in the state):

$$\begin{bmatrix} s'_{0,c} \\ s'_{1,c} \\ s'_{2,c} \\ s'_{3,c} \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} s_{0,c} \\ s_{1,c} \\ s_{2,c} \\ s_{3,c} \end{bmatrix} \quad \text{for } 0 \le c < \boldsymbol{Nb}.$$

Where the additions are bit wise XORS (explained below) and the multiplications are performed over a *finite field algebra (below).* This operation produces the new four bytes in a column:



MixColumns() operates on the State column-by-column.

$$s'_{0,c} = (\{02\} \bullet s_{0,c}) \oplus (\{03\} \bullet s_{1,c}) \oplus s_{2,c} \oplus s_{3,c}$$

$$s'_{1,c} = s_{0,c} \oplus (\{02\} \bullet s_{1,c}) \oplus (\{03\} \bullet s_{2,c}) \oplus s_{3,c}$$

$$s'_{2,c} = s_{0,c} \oplus s_{1,c} \oplus (\{02\} \bullet s_{2,c}) \oplus (\{03\} \bullet s_{3,c})$$

$$s'_{3,c} = (\{03\} \bullet s_{0,c}) \oplus s_{1,c} \oplus s_{2,c} \oplus (\{02\} \bullet s_{3,c}).$$

The algebra you need to build this logic is simple… You don't really need to learn finite fields to accomplish this task:

Elements (or bytes) from above are treated as polynomials therefore $\{12\}_H=\{0001\_0010\}_B$ is a polynomial where its coefficients are the bits: $0*X^7+0*X^6+0*X^5+1*X^4+0*X^3+0*X^2+1*X^1+0*X^0=X^4+ X^1$.

Addition in a finite field is easy it just means to XOR polynomials coefficients one by one (^ in verilog) that is: {32}+{DA}={0011_0010}+{1101_1010} is equivalent to XORing the elements. In this notation + is treated as addition modulo 2 which is a XOR, $\oplus$. That is, it equals: {1110_1000}= $X^7+ X^6+X^5+ X^3$.

Multiplication however is more complex. Polynomials (bytes) multiplication over finite fields requires in general to perform long multiplication (כפל ארוך) and to perform division modulo the field irreducible polynomial. However, let us give you an easy way to do this for your special case:

In our case (see above) we only need to multiple with the polynomials {02} or {03}. Multiplying by $\{02\}=1*X^1=X$ is easy because it means to **shift left** the operand (we know how to do this in Verilog right ☺) because for example: $\{00000010\}_B*\{00000101\}_B=\{02\}_H*\{05\}_H=X*(X^2+1)=X^3+X=\{0A\}_H=\{00001010\}_B$. And multiplying by {03} is once to do a shift left and following to add (XOR) the same polynomial to the result… 1+X … So it is a shift left and following it XORing with the original.

However, there is a special case: what if our operand that we try to multiple by X has 1 in the MSB, (that is, bit[7]) then the results is needed to by divided modulo the irreducible polynomial. So we give you the solution to this problem. If bit-7 != 1 just shift left. If bit-7 = 1 XOR the shift left result with $\{1B\}=1+X+X^3+X^4$

.

To further make this module easy on you guys we give you the code for doing a multiplication by X of the input byte to produce the correct output byte.

```
module xmult (Input_Byte, Output_Byte);
```

```
    input [7:0] Input_Byte;
    output wire [7:0] Output_Byte;
    // checks the MSB of the Input_Byte: if it is 0 → shift left
    // if it is a 1 → shift to the left and XOR with {1B}.
    assign Output_Byte = (!Input_Byte[7]) ? {Input_Byte[6:0],1'b0} : {Input_Byte[6:0],1'b0} ^ 8'h1B;
endmodule
```

So now you know all that is needed to construct this module !

5. ***KeyExpansion*** module in details:

The key expansion block diagram is provided as follows.

SubWord is a function (or module) that takes a four-byte input word and applies the S-box to each of the four bytes to produce an output word (you need to instantiate 4 ***SubBytes***). The function ***RotWord*** (or module) takes a word [a0,a1,a2,a3] as input, performs a cyclic permutation, and returns the word [a1,a2,a3,a0].That is the MS byte shifts the LS byte. The round constant word array, ***Rcon***[i], contains the values given by $[x^{i-1},\{00\}_H,\{00\}_H,\{00\}_H]$, with x^{i-1} being powers of x (x is denoted as $\{02\}_H$ as discussed above), note that i starts at 1, not 0. So to make this clear some examples for ***Rcon***:

Rcon[1]→ $X^{l-1}=x^0=1=\{01\}_H$. Therefore, the result is: $\{01\}_H,\{00\}_H,\{00\}_H,\{00\}_H$ = {8'h01,24'h0}.

Rcon[2]→ $X^{l-1}=x^1=x=\{02\}_H$. Therefore, the result is: $\{02\}_H,\{00\}_H,\{00\}_H,\{00\}_H$ = {8'h02,24'h0}.

Rcon[3]→ $X^{l-1}=x^2=x=\{04\}_H$. Therefore, the result is: $\{04\}_H,\{00\}_H,\{00\}_H,\{00\}_H$ = {8'h04,24'h0}.

Rcon[4]→ $X^{l-1}=x^3=x=\{08\}_H$. Therefore, the result is: $\{08\}_H,\{00\}_H,\{00\}_H,\{00\}_H$ = {8'h08,24'h0}.

Rcon[5]→ $X^{l-1}=x^4=x=\{10\}_H$. Therefore, the result is: $\{10\}_H,\{00\}_H,\{00\}_H,\{00\}_H$ = {8'h10,24'h0}.

…

However, i=9 we are overflowed and we are then moving to $\{1B\}_H$ (as discussed before):

Rcon[9]→ $X^{l-1}=x^8=x=\{1B\}_H$. Therefore, the result is: $\{1B\}_H,\{00\}_H,\{00\}_H,\{00\}_H$ = {8'h1B,24'h0}.

The following x shifts (increasing the power) are shifts of 1B.. as follows:

Rcon[10=A_H]→ $X^{l-1}=x^9=x=\{36\}_H$. Therefore, the result is: $\{36\}_H,\{00\}_H,\{00\}_H,\{00\}_H$ = {8'h36,24'h0}.

Rcon[11=B_H]→ $X^{l-1}=x^{10}=x=\{6C\}_H$. Therefore, the result is: $\{6C\}_H,\{00\}_H,\{00\}_H,\{00\}_H$ = {8'h6C,24'h0}.

Rcon[12=C_H]→ $X^{l-1}=x^{11}=x=\{D8\}_H$. Therefore, the result is: $\{D8\}_H,\{00\}_H,\{00\}_H,\{00\}_H$ = {8'hD8,24'h0}.

Another overflow… Shift and add $\{1B\}_H$…

Rcon[13=D_H]→ $X^{l-1}=x^{12}=x=\{AB\}_H$. Therefore, the result is: $\{AB\}_H,\{00\}_H,\{00\}_H,\{00\}_H$ = {8'hAB,24'h0}.

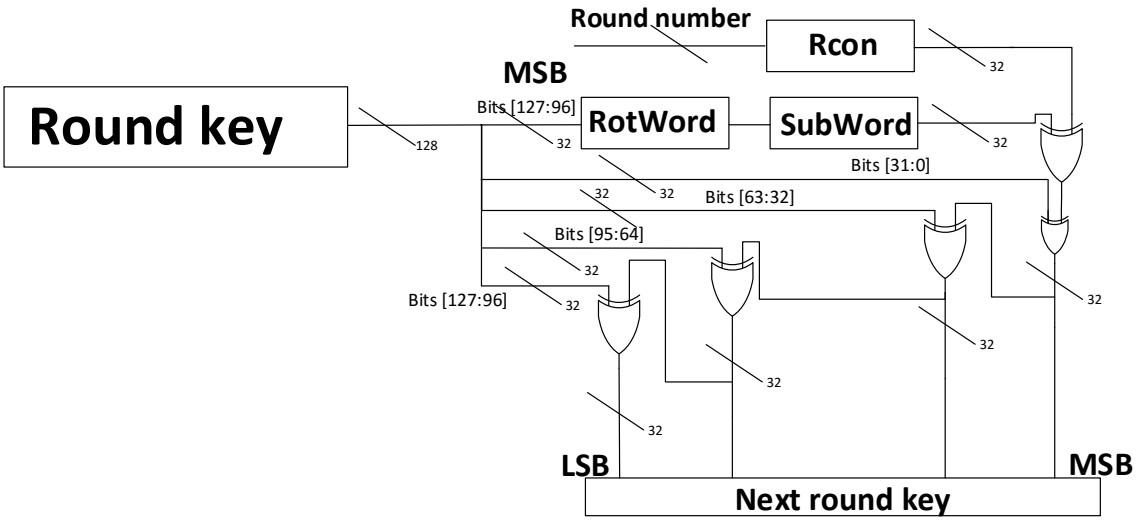
Another overflow… Shift and add $\{1B\}_H$…

Rcon[14=E_H]→ $X^{l-1}=x^{13}=x=\{4D\}_H$. Therefore, the result is: $\{4D\}_H,\{00\}_H,\{00\}_H,\{00\}_H$ = {8'h4D,24'h0}.

Rcon[15=F_H]→ $X^{l-1}=x^{14}=x=\{9A\}_H$. Therefore, the result is: $\{9A\}_H$,$\{00\}_H$,$\{00\}_H$,$\{00\}_H$ = {8'h9A,24'h0}.

So you have all that is needed to construct the ***Rcon*** module. i that enters the *Rcon* is the round no. that you will count **(so in your case it will not pass 10…).**

If you lack understanding in any of the above detailed you are welcome to go through the explanation of the key expansion algorithm in the AES standard link above or in any of the millions of places on the net it is published ☺.



Guidelines and instructions:

Assignment per group and instructions:

Divide to groups of 2 (no more than 2 under any circumstances, 3 is not allowed even if there is an odd no. of students in the course).

Take the LSB digit of the ID no. (the control digit) of both of the students and add them (if it is a group of 1 than only his LSB). If the result LSB digit is:

- 0, 1, 5, 7- Assignment 3
- 2, 4, 8 - Assignment 2
- 3, 6, 9 - Assignment 1

Assignment 1:

You should design the ***unfolded not-pipelined*** design (purely combinational) with registers at the beginning and at the end of the design. Have a ***Reset*** strategy. Your header should look something like this:

module AES_cipher (CLK, Reset, Plain_Text, Key, Cipher_Text);

Results: waveform with each clock producing new cipher.

Assignment 2:

You should design the ***unfolded pipelined*** design (each stage is a pipe and each ***KeyExpansion*** iteration is a stage). Have a ***Reset*** strategy. Your header should look something like this:

module AES_cipher (CLK, Reset, Plain_Text, Key, Cipher_Text);

Results: waveform with each clock producing new cipher.

Assignment 3:

You should design the ***folded pipelined*** design (the ***KeyExpansion*** is also piped). Have a ***Reset*** strategy. Your header should look something like this:

module AES_cipher (CLK, Start, Plain_Text, Key, Cipher_Text, Done);

Results: waveform with producing new cipher once every 10 cycles (10 clocks). In this design you will need to design a very simple state machine for the main loop and depends on how you design for the key expansion as well (however, if you know and understand what you are doing you can just go ahead and code it straight forward and it is quite simple – very small amount of code lines…).

Recall that state machine is needed everywhere you need to "remember" something… –In our case, only once every 10 cycles you will insert new plaintext and produce results.

For readability: aid the 2 control signals ***Start*** (input) and ***Done*** (output) to initiate encryption and start the round counting and ***Done*** to acknowledge the computation has ended and a valid cypher is @ the outputs. This will help the wave forms to become clearer.

General guidelines:

- All should be parametric:

no. of bits for key and word Nb=128

no. of columns Nb=4

no. of rows Nk=4

no. of rounds Nr=10.

parameter BYTE=8;

parameter WORD=32;

Do not hard code ranges of vectors!! [7:1]→[BYTE-1:0]

- Always have **default** states for **case** statements and a complete if else for all states!! →do not infer unwanted latches...
- **USE GENERATE !!!!! → instead of writing very long codes!!! (especially for instantiating!)**

Verification:
The top level test-bench must have the following two tests:

1.
Plain_Text = 128'h3243f6a8885a308d313198a2e0370734;
Key = 128'h2b7e151628aed2a6abf7158809cf4f3c; //example from appendix B in the AES standard
//expected output is: 3925841D02DC09FBDC118597196A0B32

2.
Plain_Text = 128'h00112233445566778899aabbccddeeff;
Key=128'h000102030405060708090a0b0c0d0e0f;//example from appendix C in the AES standard
//expected output is: 69c4e0d86a7b0430d8cdb78070b4c55a.