Project Conception of Numeric Systems

AES Decoder

ISMIN 2A

FENG Yuan

December 2018

1. **Introduction**

AES is an algorithmic data encryption standard that uses symmetric keys to encode and decode data.

The process of encoding and decoding is composed of 11 rounds.

The original data is processed by passing AddRoundKey in the first round, following by nine times in a set of four functions: SubBytes, ShiftRows, MixColumns, AddRoundKey, then passed into the last round with only SubBytes, ShiftRows and AddRoundKey.

For decoding, The encrypted data is processed by passing AddRoundKey in the first round, following by nine times in four steps: InvSubbytes, InvShiftRows, InvMixColumns, AddRoundKey, then passed into the last round with only InvSubBytes, InvShiftRows and ADDRoundKey. Figure 1.1 shows the pseudo code for AES decryption algorithm.

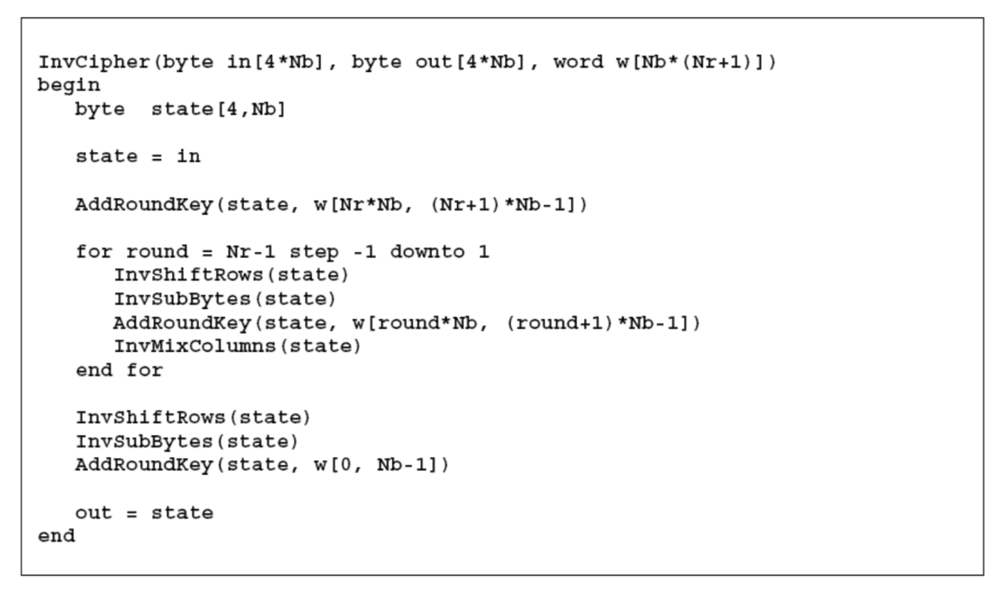


Figure 1.1 pseudo code for AES decryption algorithm

A 16-byte data block (128 bit i.e.) forms the message to be inputted into the decoder. A secret key used in the initial round and then derived in the next rounds is a parameter of the AES algorithm. An additional function is required for the generation of the round keys. This function is called Key Expansion.

In this project, VHDL is used to develop a AES decoder. VHDL is a hardware description language used in electronic design to describe digital systems. It has higher efficiency compared to high-level programming languages like C. It is also a good chance to apply our knowledge and comprehension of VHDL and numeric systems.

1. **General Design**

Figure 2.1 shows the inputs and outputs of InvAES entity. A 128 bits ciphered text *Data\_i* is sent into the decoder. *Clock\_i* permits all the components in InvAES work simultaneously. *Start\_i* boots the process of decryption. *Reset\_i* allows to reset the decoder. *Aes\_on\_o=1* when a whole decryption cycle finishes. *Data\_o* outputs a 128 bits deciphered text.

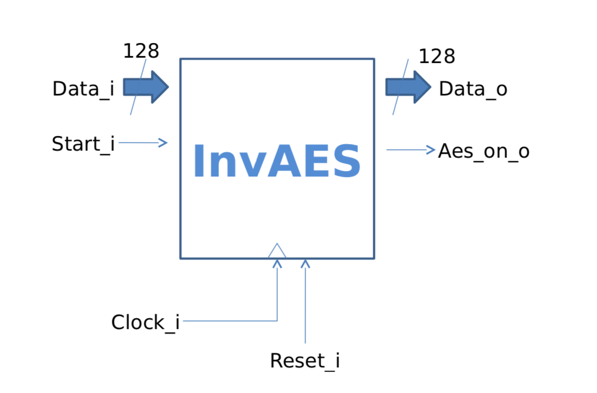


Figure 2.1 InvAES entity

1. **Component Description**
   1. S\_Box.vhd

SBox is a table for substitution. Index X comes from the high 4 bits (7 to 4) of the input while index Y comes from the low 4 bits (3 to 0) of the input. For encoding and decoding, we use different substitution table. Figure 3.1.1 shows the substitution table for decoding.

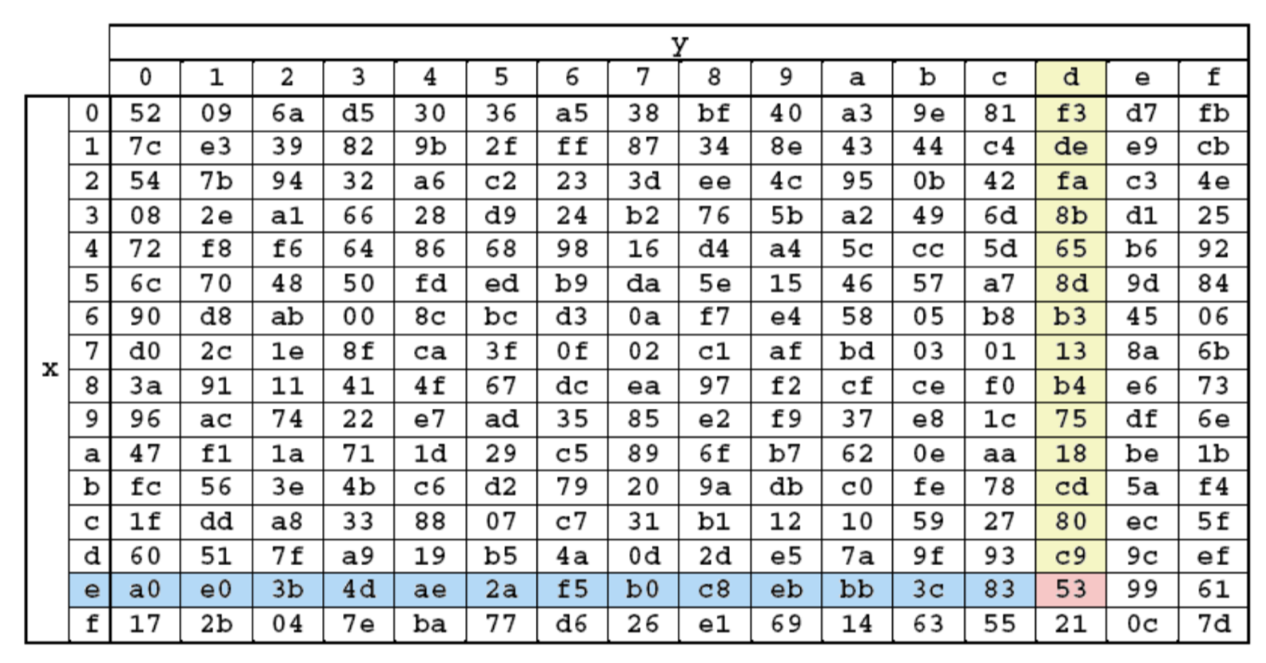


Figure 3.1.1 substitution table for decoding.

In this figure, for example, the input of S\_Box is “11101101” (“ed” in hexadecimal). So index X is “1110” (“e” in hexadecimal), index Y is “1101” (“d” in hexadecimal). Therefore we can locate the substitution “53” in hexadecimal and translate it into binary “01010011” for output.

Figure 3.1.2 shows the entity S\_Box.

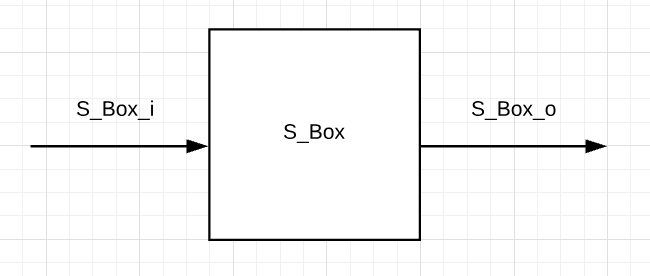


Figure 3.1.2 entity S\_Box

S\_Box\_i is a 8 bits input.

S\_Box\_o is a 8 bits output.

In the test bench of S\_Box.vhd, “ed”, “ee”, “ef” are sent to S\_Box\_i as input, then S\_Box\_o outputs the right substitution.



Figure 3.1.3 waves of S\_Box\_tb.vhd

3.3 InvSubBytes.vhd

InvSubBytes is the first function for round 0 to round 9.

This function replaces each element of the input state by using the substitution table of SBox. Figure 3.3.1 shows its principle.

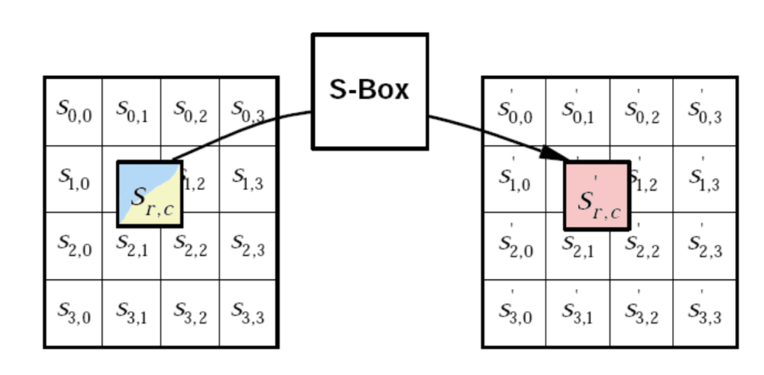


Figure 3.3.1 principle of InvSubBytes

Figure 3.3.2 shows the entity InvSubBytes.

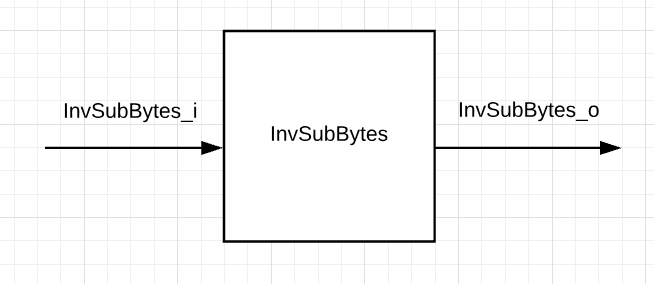


Figure 3.3.2 entity InvSubBytes.

The input and output is a 4\*4 state block (*type\_state*), each element of the block contains 2 hexadecimal digits.

In order to substitute 16 elements, the entity of InvSubBytes uses S\_Box as a component, and calls it 16 times for substitution. Figure 3.3.3 shows internal structure of InvSubBytes.

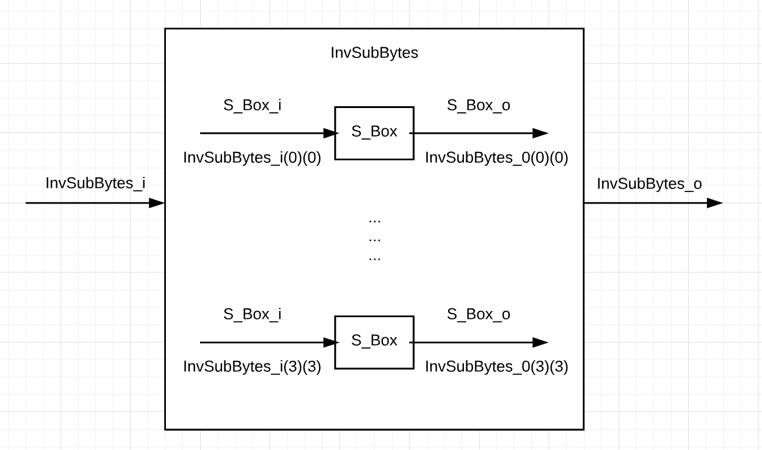


Figure 3.3.3 internal structure of InvSubBytes

The VHDL code of port map *S\_Box\_i=>InvSubBytes\_i(0)(0), S\_Box\_o=>InvSubBytes\_o(0)(0)* indicates the substitution of one element by using S\_Box, idem for other elements.

Figure 3.3.4 shows the wave of testbench simulation. Compared with substitution table for decoding, the output wave gives right result.

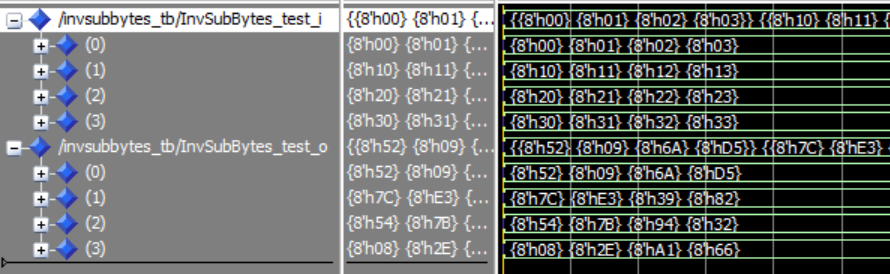


Figure 3.3.4 waves of InvSubBytes\_tb.vhd

3.4 InvShiftRows.vhd

InvShiftRows is the second function of round 9 to round 0.

This function performs a cyclic permutation of the bytes of the state lines. The offset of the bytes corresponds to the index of the considered line (0 ≤ r <4).

For example: in line 3, x = {0xaa, 0xbb, 0xcc, 0xdd}, the application of the function will right shift it 3 times and get x’ = {0xbb, 0xcc, 0xdd, 0xaa}. In other words, S3,0S3,1S3,2S3,3 is replaced by S3,1S3,2S3,3S3,0.

An illustration of the functionality is detailed in Figure 3.4.1.

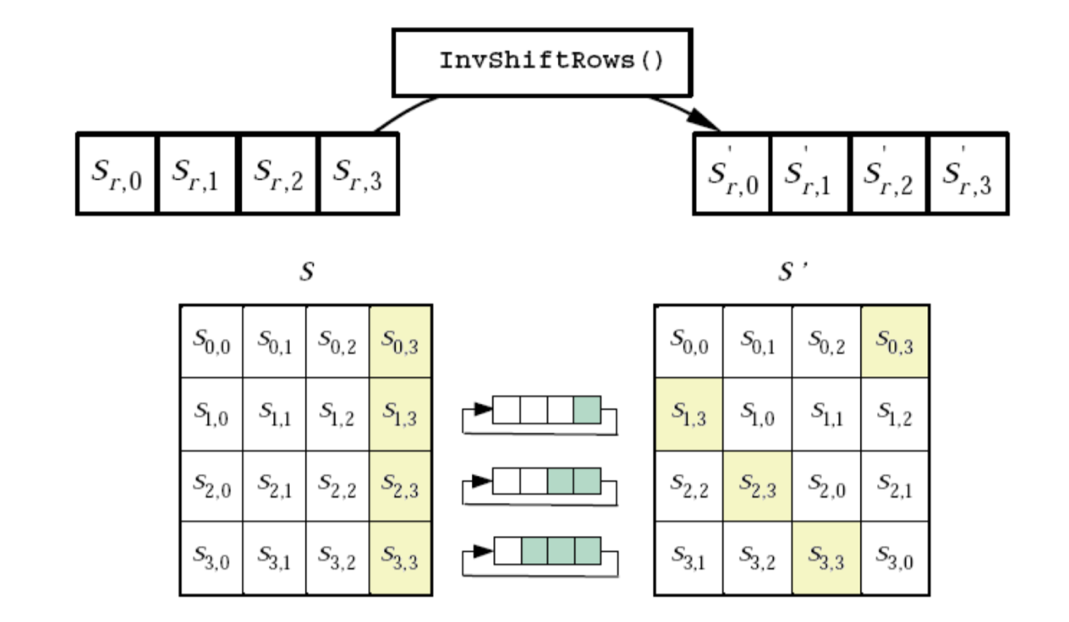


Figure 3.4.1 functionality of InvShiftRows

Figure 3.4.2 shows the entity InvShiftRows.

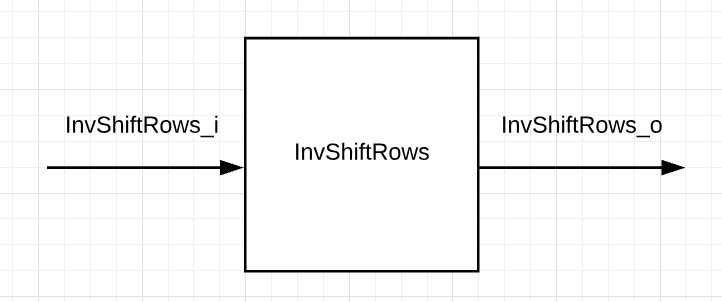


Figure 3.4.2 entity InvShiftRows

The input and output is a 4\*4 state block (*type\_state*), each element of the block contains 2 hexadecimal digits.

In VHDL code, *InvShiftRows\_o(3)(3)<=InvShiftRows\_i(3)(0)* indicates that S3,3 is replaced by S3,0 , idem for other elements in the state.

Figure 3.4.3 shows the wave of testbench simulation and gives the correct result.

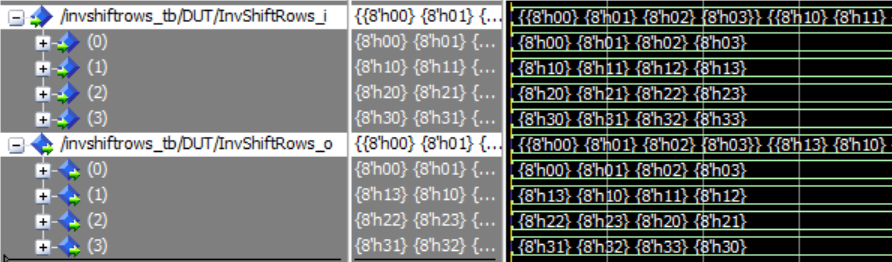


Figure 3.4.3 waves of InvShiftRows\_tb.vhd

3.5 KeyExpansion\_table.vhd

This vhd file is provided. It’s functionality is to supply key for InvAddRound in each round.

Figure 3.5.1 shows the entity KeyExpansion\_table.

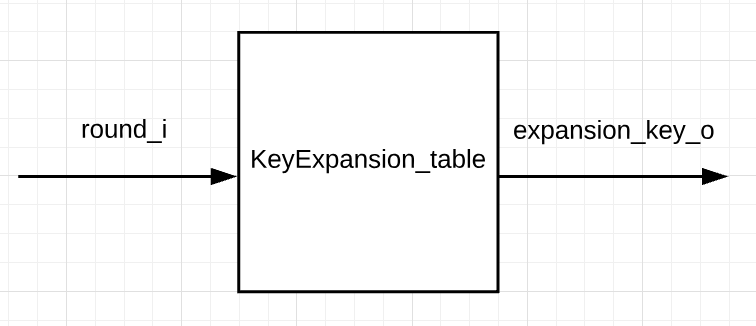


Figure 3.5.1 entity KeyExpansion\_table

The input round\_i is bit4, which indicates the round.

The output gives a key of 128 bits according to the round.

3.6 InvAddRoundKey.vhd

InvAddRoundKey is the third function for round 0 to round 9.

This function adds the current round key (round 10 to round 0) to the state. It is equivalent to the AddRoundKey function in encoding process. Therefore, the Boolean XOR function is applied bit-by-bit between the bytes of the state and the bytes of the round key.

Figure 3.6.1 shows the calculation of InvAddRoundKey function.

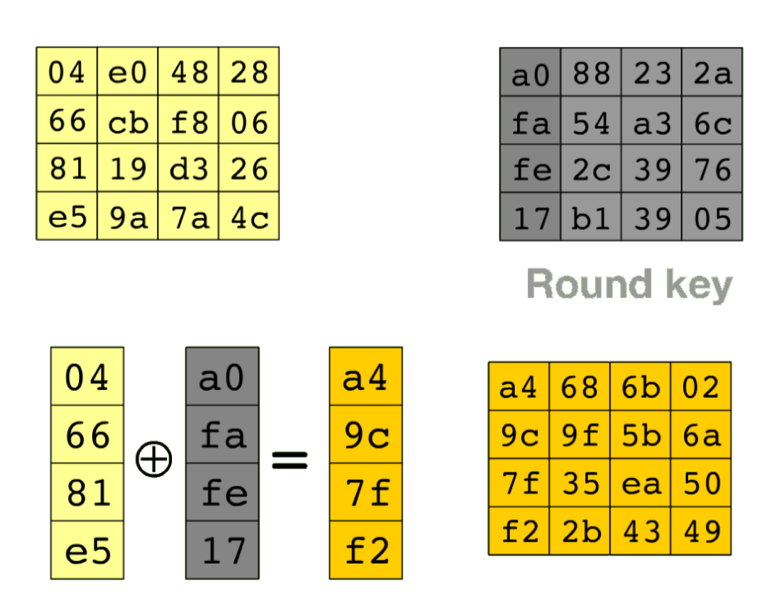


Figure 3.6.1 calculation of InvAddRoundKey

Figure 3.6.2 shows the entity InvAddRoundKey.

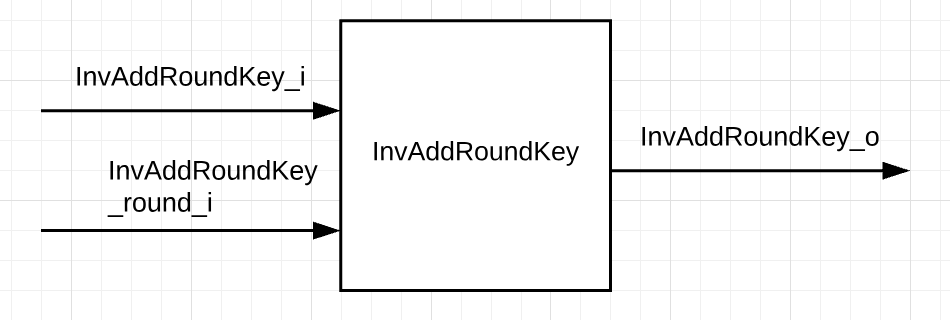


Figure 3.6.2 entity InvAddRoundKey

The input InvAddRoundKey\_i and output InvAddRoundKey\_o is *type\_state*.

The input InvAddRoundKey\_round\_i is 4 bits which indicates the round.

KeyExpansion\_table is integrated as component in the InvAddRoundKey entity. Figure 3.6.3 shows internal structure of InvAddRoundKey.

Signal *ek\_s* temporally saves the key of current round, it is divided into continuous bit8 as *ek0\_s ... ek15\_s*, then XOR with each element of input state.

VHDL code *InvAddRoundKey\_o(0)(0)<= ek0\_s xor InvAddRoundKey\_i(0)(0)* represents one of the XOR operation, idem for other 15 elements.

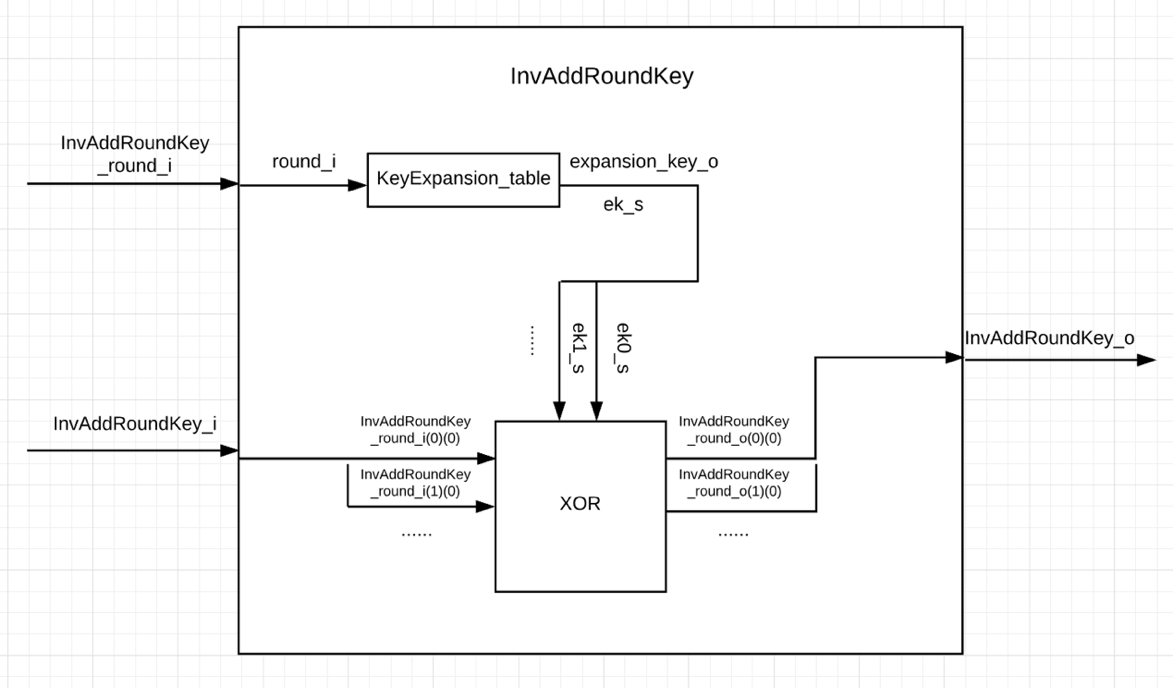


Figure 3.6.3 internal structure of InvAddRoundKey

Figure 3.6.4 shows the waves of testbench simulation and gives the correct results of round 10 and round 7.

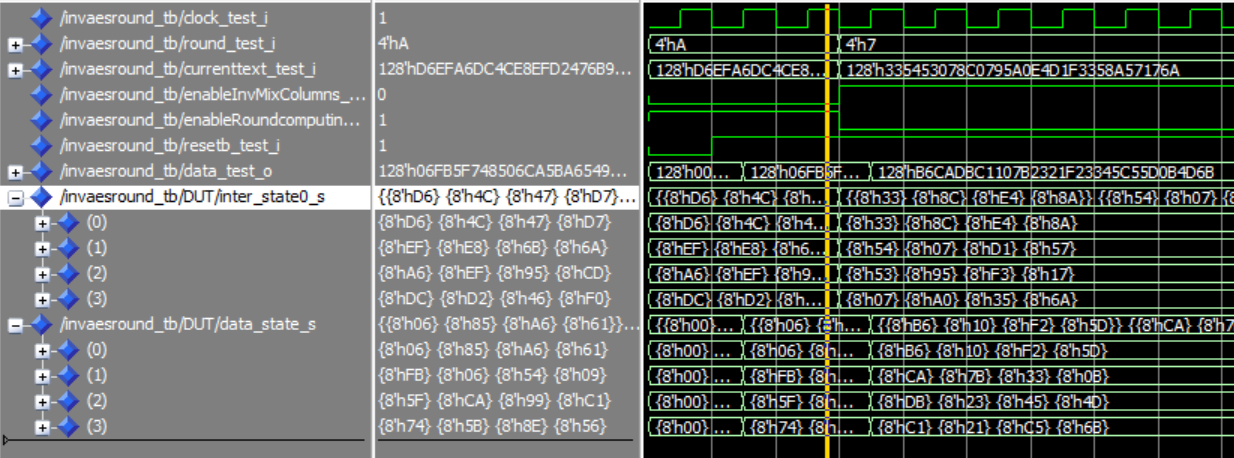


Figure 3.6.4 waves of InvAddRoundKey\_tb.vhd

3.7 InvMixColumns.vhd

InvAddRoundKey is the fourth function for round 1 to round 9.

It applies a column-by-column transformation to a state. This linear transformation of a matrix product using the 4 bytes of a column. The columns are treated as polynomials in GF (28)2 and multiplied modulo x4 + 1 (noted ⊗) with the fixed polynomials in Figure 3.7.1. These operations performed in the Welsh field are calculated for the additions using an XOR function.

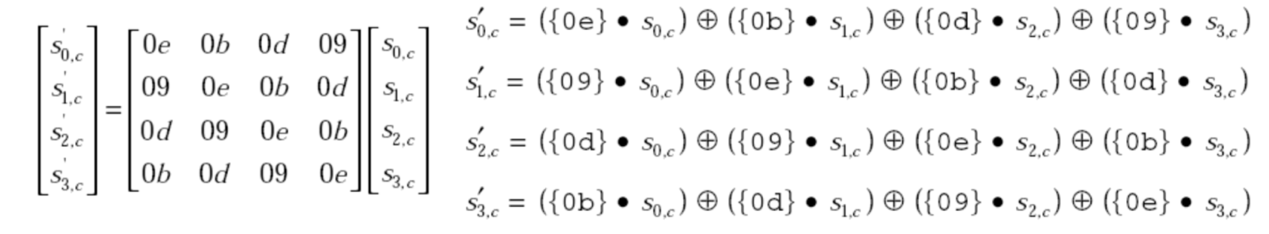


Figure 3.7.1 polynomials for InvMixColumns

Figure 3.7.2 shows the entity InvMixColumns.

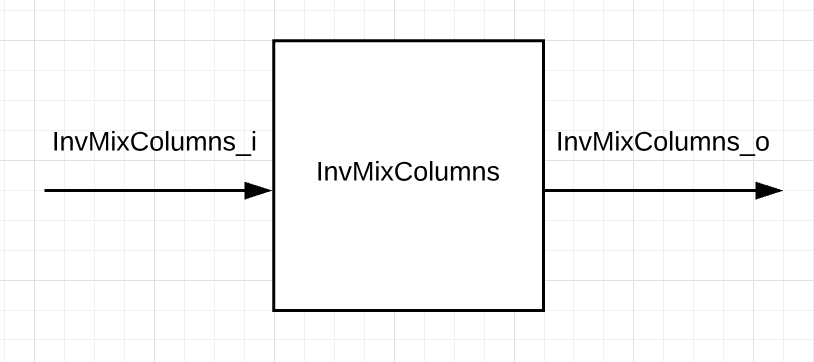


Figure 3.7.2 entity InvMixColumns

The input InvMixColumns \_i and output InvMixColumns \_o is *type\_state*.

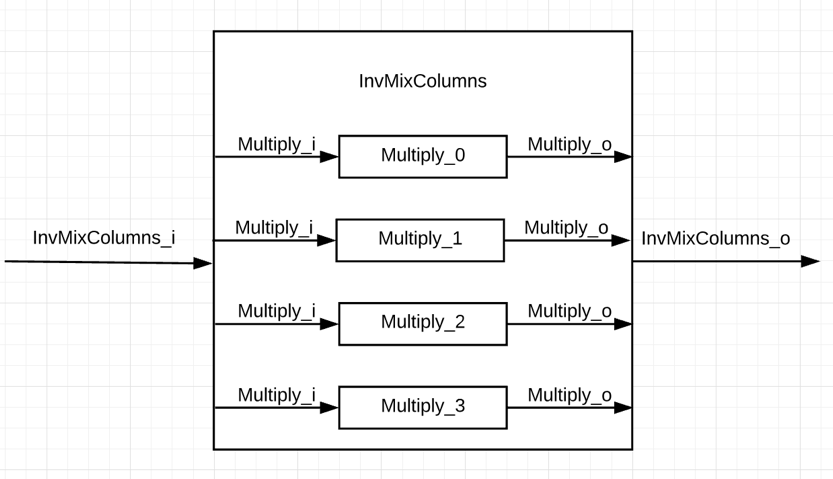


Figure 3.7.3 internal structure of InvMixColumns

All the calculation of InvMixColumns is written in *3.8 Multiply.vhd*. In order to calculate 4 columns, the entity of InvMixColumns uses Multiply as a component, and calls it 4 times for calculation of each column. Figure 3.7.3 shows internal structure of InvMixColumns.

Figure 3.7.4 shows the wave of testbench simulation and gives the correct result of round 7.

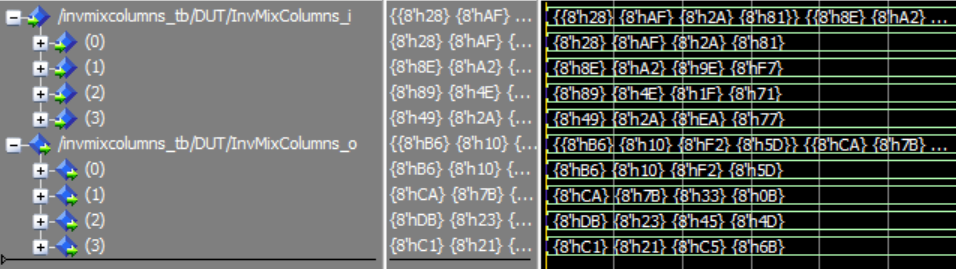


Figure 3.7.4 waves of InvMixColumns\_tb.vhd

3.8 Multiply.vhd

Multiply is a component of InvMixColumns and it calculates one column of the input state by using invMixColumn algorithm. Figure 3.8.1 shows the entity Multiply.

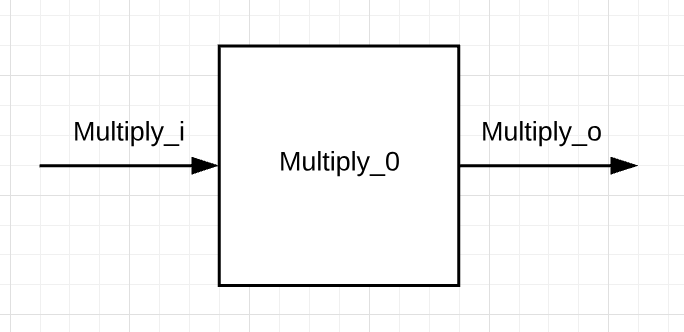


Figure 3.8.1 entity Multiply

From Figure 3.7.1, we know that each element must have to be multiplied (modulo x4 + 1) by {0E},{0B},{0D} and{09}.

So take S0,0 as example:

{0E}S0,0 = (23+22+21) S0,0 = 8S0,0⊕4S0,0⊕2S0,0

{0B}S0,0 = (23+20) S0,0 = 8S0,0⊕S0,0

{0D}S0,0 = (23+22+20) S0,0 = 8S0,0⊕4S0,0⊕S0,0

{09}S0,0 = (23+21+20) S0,0 = 8S0,0⊕2S0,0⊕S0,0

That also means each element must have to be multiplied by 8, 4 and 2.

Suppose S0,0 = a7 a6 a5 a4 a3 a2 a1 a0 , so

{02}S0,0 =a6 a5 a4 a3 a2 a1 a0 0 when a7 = 0

{02}S0,0 =a6 a5 a4 a3 a2 a1 a0 0 ⊕ 00011011 when a7 = 1

In VHDL code, the calculation of {02}S0,0 is implemented as:

*ER0\_Mul2\_s <= ( ElementRow0\_s(6 downto 0) & '0' ) xor ( "000" & ElementRow0\_s(7) & ElementRow0\_s(7) & '0' & ElementRow0\_s(7) & ElementRow0\_s(7) );*

which evades the judgement of a7.

Then it is easy to calculate {04}S0,0 , {08}S0,0 , {0E}S0,0 , {0B}S0,0 , {0D}S0,0 , {09}S0,0 .

Finally, use the formula in Figure 3.7.1 to calculate final result and send it to output.

3.9 InvAESRound.vhd

InvAESRound integrates four functions: InvSubbytes, InvShiftRows, InvMixColumns, InvAddRoundKey, which compose a whole round.

Figure 3.9.1 shows the entity InvAESRound. Calculation will be triggered by up edge of *clock\_i* when *resetb\_i*=1.

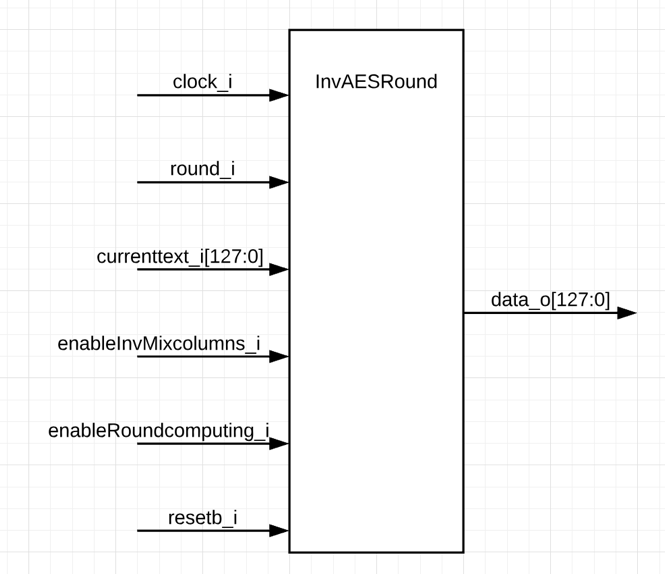


Figure 3.9.1 entity InvAESRound

Figure 3.9.2 shows internal process of InvAESRound. The internal functions (InvSubbytes, InvShiftRows, InvMixColumns, InvAddRoundKey) are used according to the values of *enableInvMixcolumns\_i* and *enableRoundcomputing\_i*.

There are internal signals to save current state or transform text between bit128 and *type\_state*.

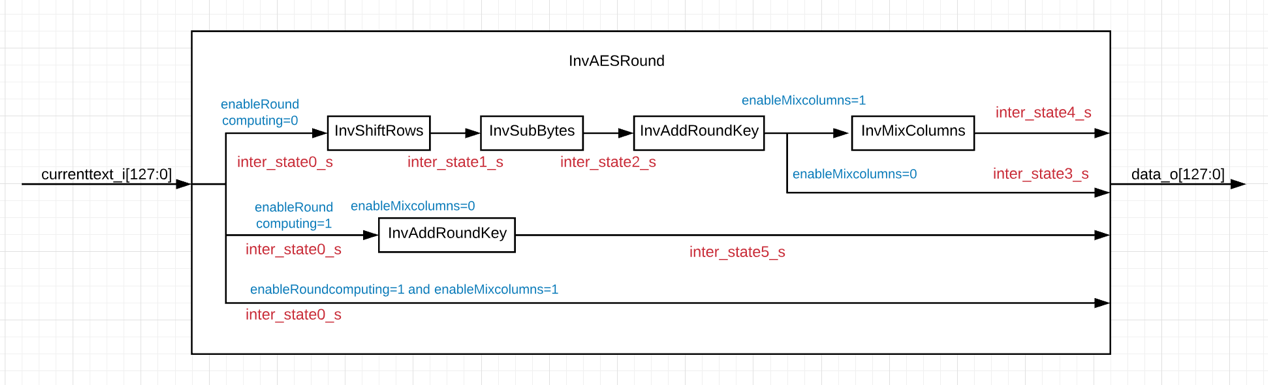


Figure 3.9.2 internal process of InvAESRound

Figure 3.9.3 shows the wave of testbench simulation and gives the correct result of round 10 and round 7.

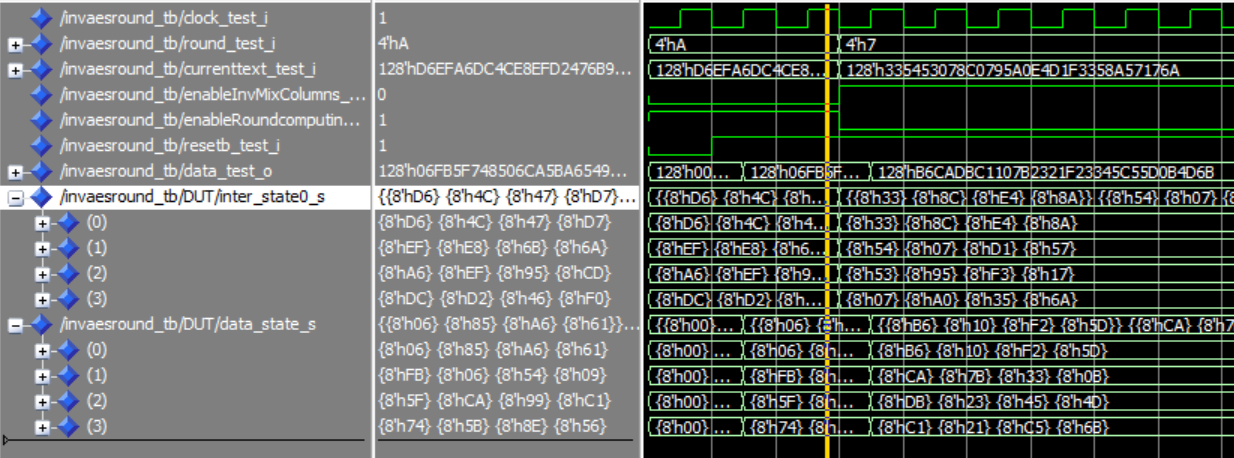


Figure 3.9.3 waves of InvAESRound\_tb.vhd

3.10 FSM\_InvAES.vhd

FSM\_InvAES is a finite-state machine (type Moore) , whose functionality is to generate the signals that controlling the entity InvAES.

Figure 3.10.1 shows the entity FSM\_InvAES.

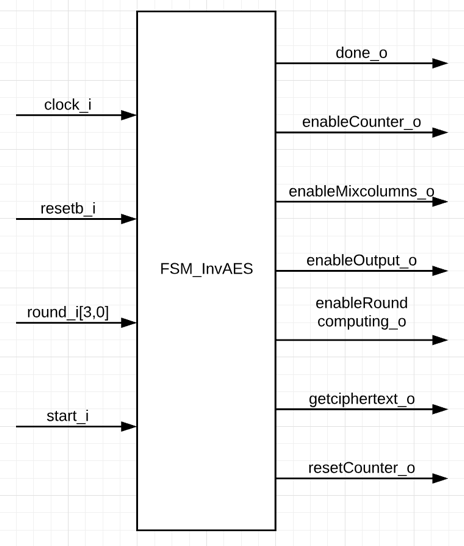


Figure 3.10.1 entity FSM\_InvAES

Figure 3.10.2 shows the state transition diagram. There are totally 5 states (*init, waiting, round10, round9to1, round0*). Besides that, *state\_present* saves present state while *state\_future* saves future state.

The FSM boots when *resetb\_i*=0. Present state enters future state when *resetb\_i*=1 and is triggered by an up edge of *clock\_i*.

When *round\_i=A*, the state should enter state *round9to1* in advance because *state\_present* has one clock period lag than *state\_future*, idem when *round\_i=1*, *round9to1* should enter *round0*.

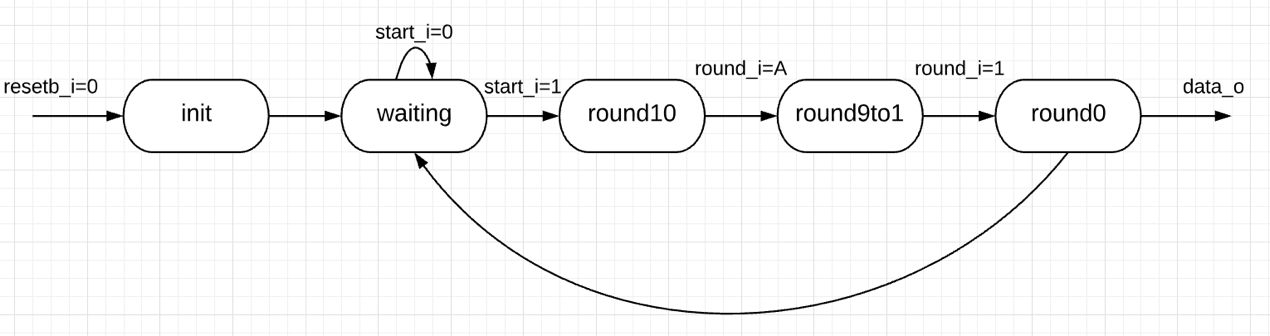


Figure 3.10.2 state transition diagram

Figure 3.10.3 shows the truth table of FSM.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | init | waiting | round10 | round9to1 | round0 |
| done\_o | 0 | 0 | 0 | 0 | 1 |
| enableCounter\_o | 0 | 0 | 1 | 1 | 1 |
| enableMixcolumns\_o | 1 | 1 | 0 | 1 | 0 |
| enableOutput\_o | 0 | 1 | 1 | 1 | 1 |
| enableRoundcomputing\_o | 1 | 1 | 1 | 1 | 0 |
| getciphertext\_o | 0 | 1 | 1 | 0 | 0 |
| resetCounter\_o | 0 | 0 | 1 | 1 | 1 |

Figure 3.10.3 truth table of FSM

Figure 3.9.3 shows the wave of testbench simulation. *State\_present* enters next state correctly. The FSM works according to state transition diagram and truth table.

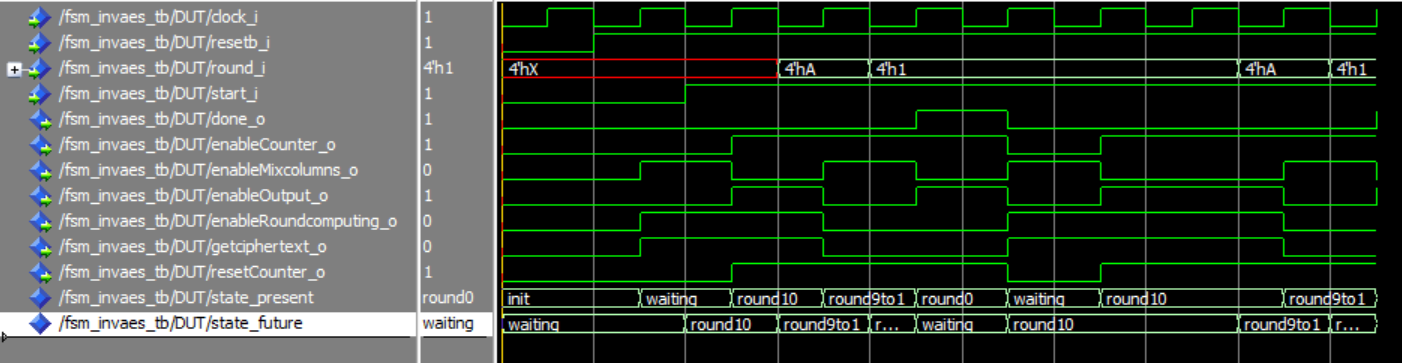


Figure 3.9.3 waves of FSM\_InvAES\_tb.vhd

3.11 Counter.vhd

The counter is controlled by FSM to fix the address of current round key.

Figure 3.11.1 shows the entity Counter.

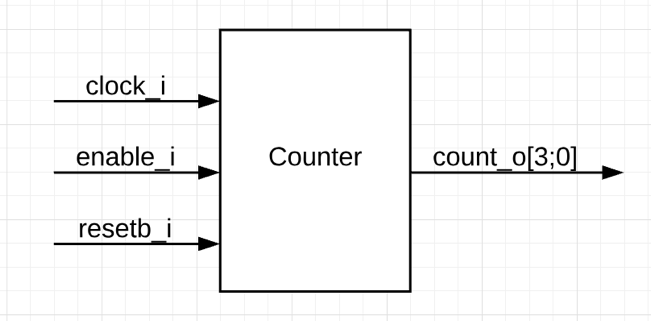


Figure 3.11.1 entity Counter

Figure 3.9.3 shows the wave of testbench simulation, when *enable\_i*=1 and *resetb\_i*=1, the counter outputs correct signal for counting rounds.

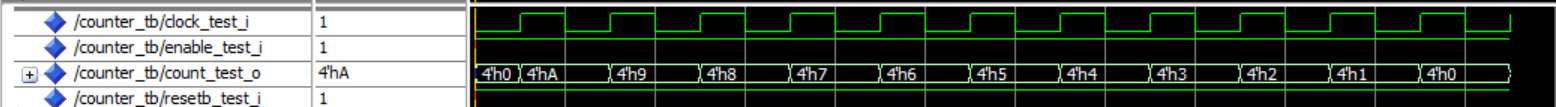


Figure 3.9.3 waves of Counter\_tb.vhd

3.12 RTL\_MUX.vhd

RTL\_MUX is a multiplexer to control the text which is sent to InvAESRound for decoding.

Figure 3.12.1 shows the entity RTL\_LUX.

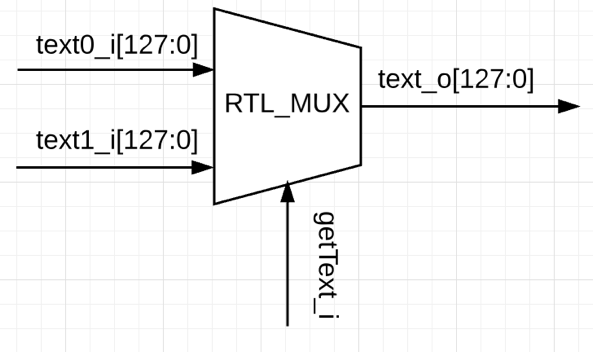


Figure 3.12.1 entity RTL\_LUX

Input signal *text0\_i* gets ciphered 128 bits original text.

Input signal *text1\_i* gets deciphered 128 bits text from InvAESRound.

Input signal *getText\_i* comes from *getciphertext\_o* of FSM\_InvAES.

If *getText\_i=*1, multiplexer outputs *text0\_i.*

If *getText\_i=*0, multiplexer outputs *text1\_i.*

Figure 3.12.2 shows the waves of RTL\_MUX\_tb.vhd.

*getText\_i=*1, multiplexer outputs *text0\_i.*

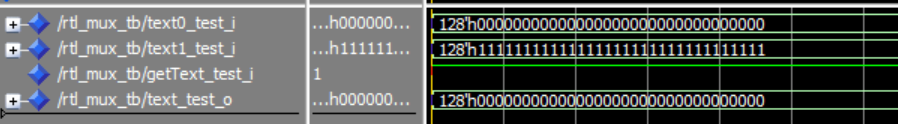


Figure 3.12.2 waves of RTL\_MUX\_tb.vhd.

3.13 InvAES.vhd

InvAES integrates all previous components (Counter, FSM\_InvAES, InvAESRound, RTL\_MUX). Figure 3.13.1 shows the global architecture of InvAES.

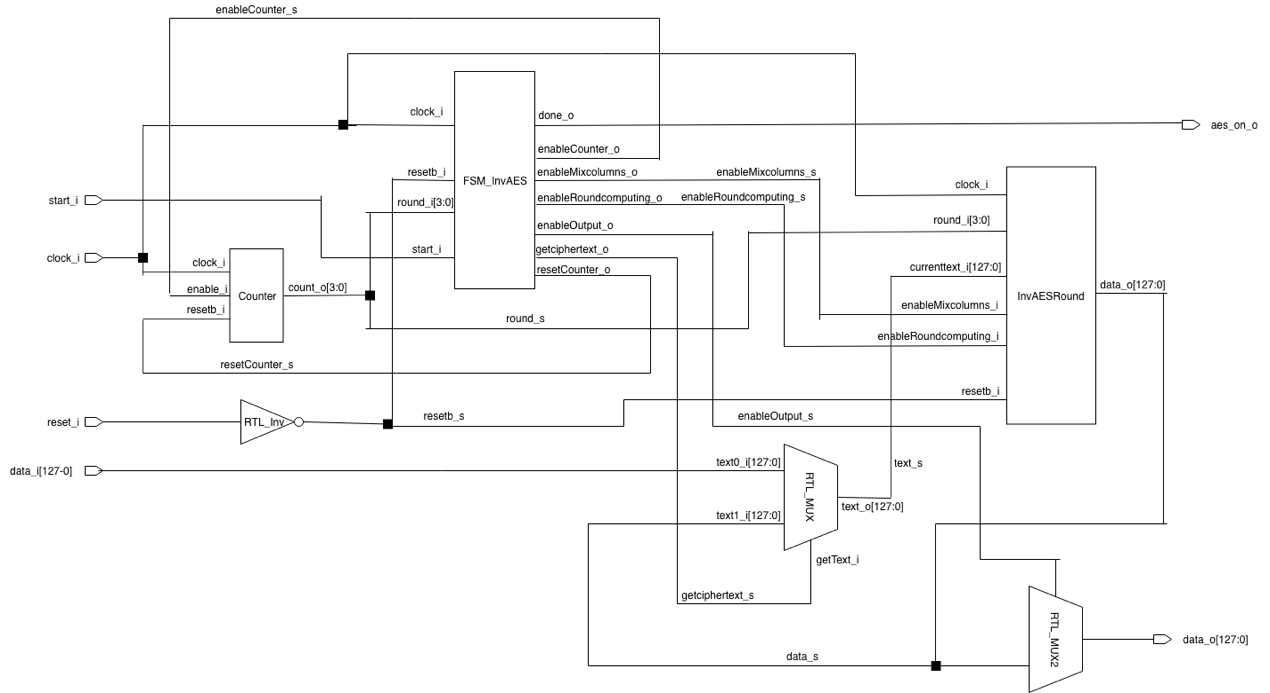


Figure 3.13.1 global architecture of InvAES

RTL\_MUX2 is another multiplexer. If it receives *enableOutput\_o=*1 from FSM, the multiplexer outputs decipher result from InvAESRound*,* else the multiplexer outputs 128 bits of zero*.*

RTL\_Inv receives *reset\_i* then outputs its inverse signal *reset\_s.*

Figure 3.13.2 and Figure 3.13.3 show the waves of InvAES\_tb.vhd. The AES Decoder receives ciphered text *X“d6efa6dc4ce8efd2476b9546d76acdf0”* then outputs the right deciphered result *X“526573746f20656e2076696c6c65203f”* after state *round0.*

Decode result of each round is correct.

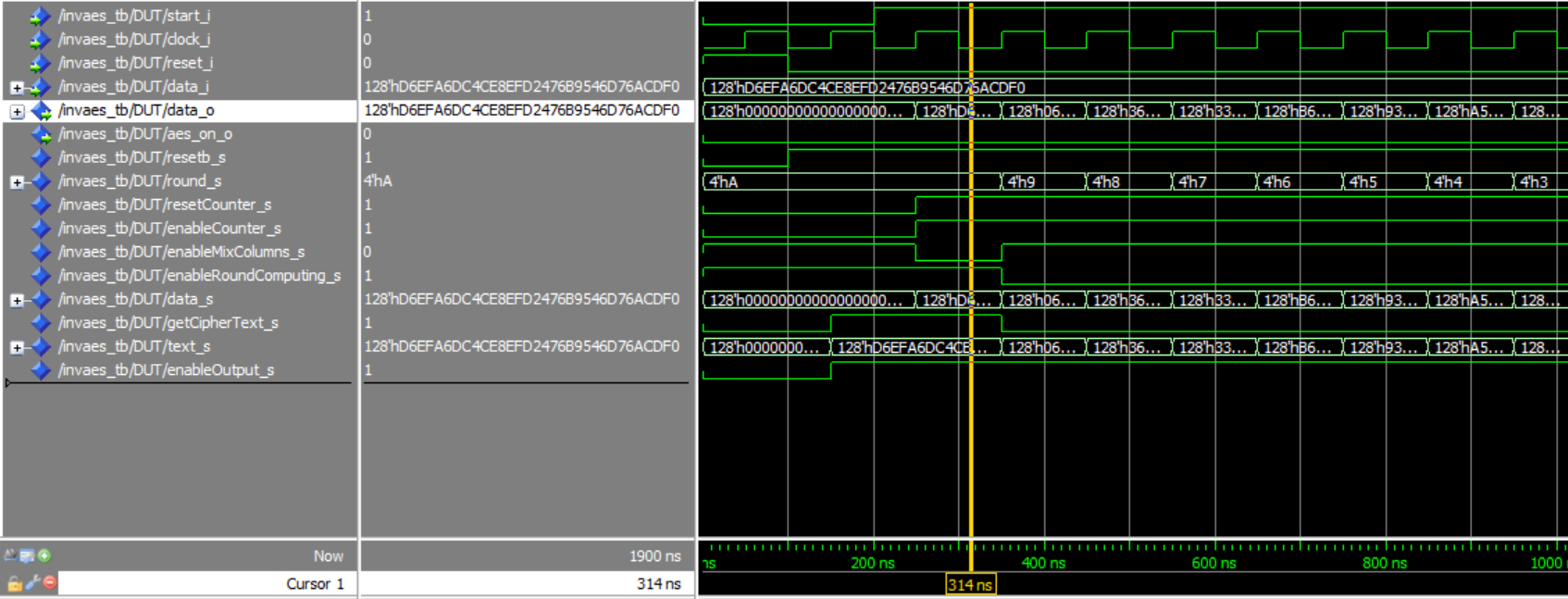


Figure 3.13.2 waves of InvAES\_tb.vhd

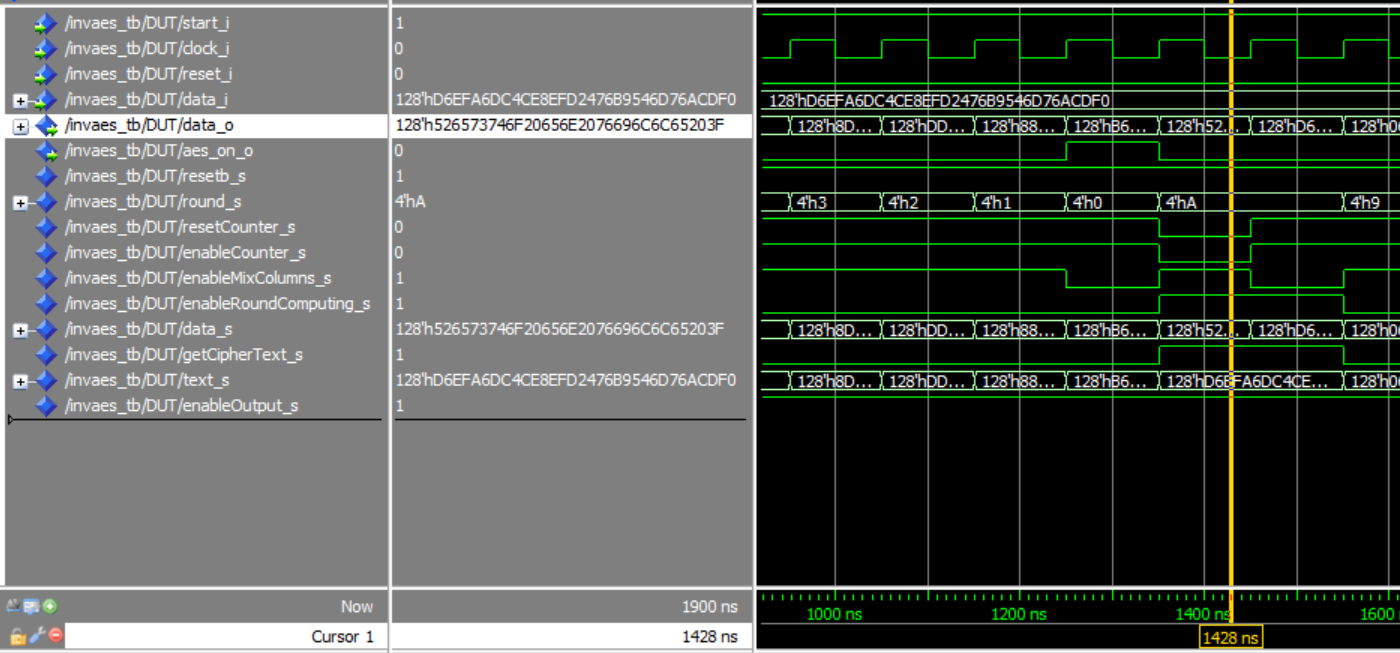


Figure 3.13.3 waves of InvAES\_tb.vhd

1. **Conclusions**

An in-depth comprehension of the AES decryption numeric system and a reasonable and effective application of VHDL is the basis for completing this project.

In the process of system designing, coding and debugging, I found and solved many problems and difficulties. Here lists some typical examples of them:

1. *If* or *when..else* ?

In RTL\_MUX (multiplexer), I used 2 *if* conditions in a process to control the output (text\_o*[127:0]*):

*if getText\_i='1' then*

*text\_s<=text0\_i*

*end if;*

*if getText\_i='0' then*

*text\_s<=text1\_i*

*end if;*

When I did the simulation of InvAES\_tb.vhd, I found *currenttext\_i* do not update in the following rounds. So I deleted the process and used *when..else* to replace 2 *if* conditions:

*text\_s<=text0\_i when getText\_i='1' else*

*text1\_i when getText\_i='0';*

then the problem solved.

The reason is that 2 *if* conditions are executed sequentially, in simulation the code was only executed once, that made *currenttext\_i* do not update. But *when..else* is executed parallelly.

1. Lag between state\_present and state\_future

There is a lag of one clock period between state\_present and state\_future, which will cause false calculation in simulation. The solution is let the state enter next state in advance of one round:

(In FSM\_InvAES.vhd)

*when round10=>*

*if round\_i=X"a" then*

*state\_future<=round9to1;*

*end if;*

*when round9to1=>*

*if round\_i=X"1" then*

*state\_future<=round0;*

*end if;*

1. How to evade the judgement of a7?

In Multiply.vhd, each element must have to be multiplied (modulo x4 + 1) by 2.

Suppose S0,0 = a7 a6 a5 a4 a3 a2 a1 a0 , so

{02}S0,0 =a6 a5 a4 a3 a2 a1 a0 0 when a7 = 0

{02}S0,0 =a6 a5 a4 a3 a2 a1 a0 0 ⊕ 00011011 when a7 = 1

The judgement of a7 will create many complicated condition sentences. In order to evade them, the calculation of {02}S0,0 can be implemented as:

*ER0\_Mul2\_s <= ( ElementRow0\_s(6 downto 0) & '0' ) xor ( "000" & ElementRow0\_s(7) & ElementRow0\_s(7) & '0' & ElementRow0\_s(7) & ElementRow0\_s(7) );*

*ElementRow0\_s(6 downto 0) & '0'* left shifts S0,0.

When a7=0, left shifted S0,0 XOR with “00000000”, the result is S0,0 itself.

When a7=0, left shifted S0,0 XOR with “00011011”.