
AES implementation on embedded system

ENGG*4560

Name: Yingcai Dong

ID: 0888325



Introduction

The Advanced Encryption Standard(AES) is also known as Rijndael. It is been established by the U.S. National Institute of Standards and Technology for doing encryption on electronic data. And it is been wildly used for decade. So the implementation of AES on hardware is meaningful, since using hardware to achieve plain text encryption(especially using co-processor) can make encryption more efficiency and less energy consuming.

In this report, I am going to implement the AES algorithm on an FPGA which is based on ARM Cortex - M1 Embedded Processor. And after the implementation, the hardware should be able to achieve plain text encryption and cipher text decipher.

The implementation can be break down in to several steps, they are:

- Program the AES algorithm using BSV
- Remove hard coded input and add test bench
- Convert BSV code into Verilog
- Use Modulesim to check if the output is correct
- Write the wrapper of the AES module
- Use Libero IDE to implement hardware development

Background

Introduction of AES algorithm

Encryption part

AES algorithm mainly include 4 parts:

1. SubBytes
2. ShiftRows
3. MixColumns
4. AddRoundKey

Initialization

Initially, the plain text will first be put into a matrix called “State” which is a 4 by 4 matrix. The way of plain text be organized is shown below:

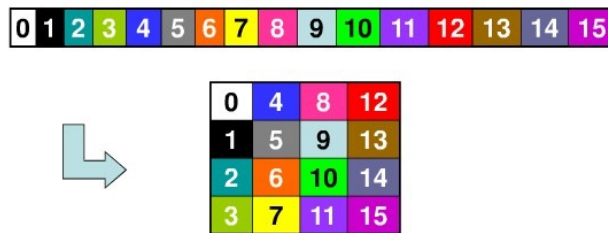


Figure1-1: Way of organize plain text

Then do XOR with the add round key. Because it is the initialization of encryption, so the round key is actually the cipher key. And the cipher key is also organized by the same way shown in Figure1-1.

ByteSubstitution

To do the ByteSubstitution, we need the accompany of the sBox. The sBox is a 16 by 16 matrix. Shown below:

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xa	xb	xc	xd	xe	xf
0x	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1x	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2x	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3x	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4x	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5x	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6x	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7x	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8x	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9x	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
ax	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
bx	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
cx	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
dx	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
ex	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
fx	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure1-2: sBox

So, we simply pick the element in the State for example: 0xf8. Then, we divide this element into two parts:

1. Row number in sBox -> f
2. Column number in sBox -> 8

Then we will be able to find an element in the sBox to replace the element in the State. And what we need to do is just follow the same step to replace every element in the State using the sBox.

ShiftRows

Shift rows is relatively easy, just using the result of ByteSubstitution:

1. First row don't shift
2. Second row shift left 1 byte
3. Third row shift left 2 bytes
4. Fourth row shift left 3 bytes

Which is shown in the figure below,

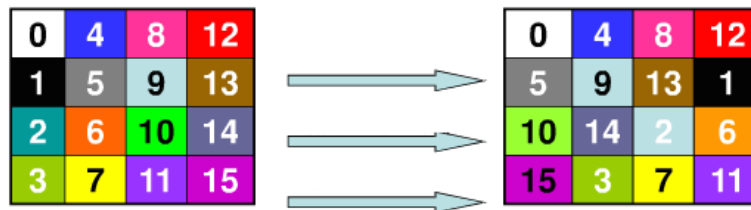


Figure1-3: ShiftRows

MixColumn

Doing MixColumn, we use the intermediate result from the ShiftRows to multiply with a 4 by 4 matrix. The matrix is shown below:

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}$$

Figure1-4: the matrix c(x) used for MixColumn

After doing the matrix multiply. And then, when doing the actual calculation, we use Finite Field Arithmetic.

For example: we pick a column from the intermediate result(which can be tread as a 4 by 1 matrix) and we name it "a₄₁", do the matrix multiply with the matrix shown in Figure1-4 and then we get a new 4 by 1 matrix "b₄₁" which can replace the original intermediate "a₄₁".

$$\begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix}$$

Figure1-5: Example of matrix multiply

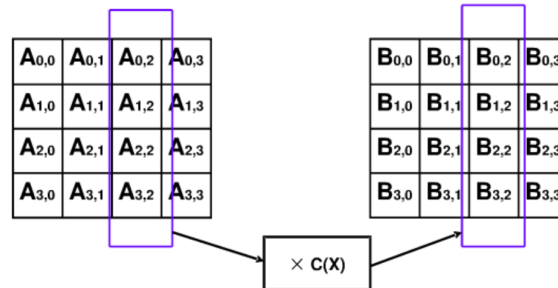


Figure1-6: Replace intermediate result

Here I'll introduce the way to do the Finite Field Arithmetic. In the Finite Field Arithmetic (also known as GF2⁸):

1. $a * 0x01 = a$
2. $a * 0x02$
 - if $a < 0x80$ begin
 - $a = a \ll 1\text{bit}$
 - end
 - else begin
 - $a = a \ll 1\text{bit}$
 - $a = a \text{ XOR } 0x1b$
 - end
3. $a * 0x0d = (a * 0x08) + a$

Add RoundKey

Use the round key to do XOR with the intermediate result from the MixColumn. Be careful, when using round key every iteration in encryption, must follow the order of group 2 to group 11.

Iteration

Do 9 iterations of ByteSubstitution, ShiftRows, MixColumn, AddRoundKey. And then do 1 iteration of ByteSubstitution, ShiftRows and AddRoundKey

KeyExpansion

KeyExpansion is used to generate enough cipher key(also known as round key) to let AddRoundKey section have different cipher key to XOR with the intermediate result. So the round key should be a 4 by 44 matrix. Which can be divide into 11 groups. And each group is a 4 by 4 matrix. As I introduced in the previous section, the first group of the round is initialized by the cipher key. And the rest of the round key is generate using particular method, which I'll introduce in the next paragraph.

The second group of the round key can be generate by following the steps shown in the figure down below.

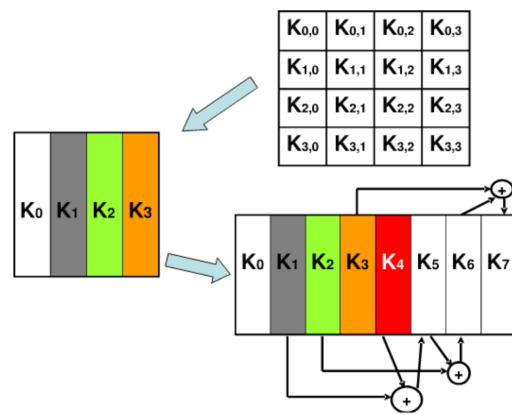


Figure1-7: KeyExpansion

And for the first column in every group, for example K₄ in Figure1-7, is using a different way to achieve:

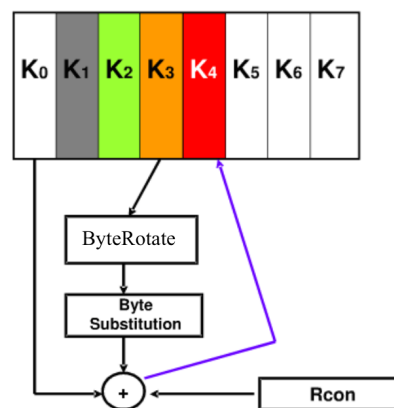


Figure1-8: Way of generate first column in every group

Where ByteRotate means shift every column's first element to the end. It's similar to ShiftRows but instead shifting different number of elements for each row, ByteRotate only shift up one byte per column; ByteSubstitution is using the same way as I showed in ByteSubstitution Section; Rcon is a 4 by 10 matrix, shown in Figure1-9.

```

{8'h01, 8'h02, 8'h04, 8'h08, 8'h10, 8'h20, 8'h40, 8'h80, 8'h1b, 8'h36}
{8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00}
{8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00}
{8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00}

```

Figure1-9: Rcon

Decipher Part

Initialization

Use round key to do XOR with the cipher text, but in deciphering, we use the last group(also known as group 11) of the round key to do the initialization.

Inverse ShiftRows

When doing the inverse shift rows:

1. First row don't shift.
2. Second row shift right 1 bytes.
3. Third row shift right 2 bytes.
4. Fourth row shift right 3 bytes.

Inverse ByteSubstitution

Doing the inverse byte substitution follows the same steps as shown in ByteSubstitution section. But instead of using sBox, we use inverse-sBox.

	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xa	xb	xc	xd	xe	xf
0x	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
1x	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
2x	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
3x	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
4x	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
5x	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
6x	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
7x	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
8x	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
9x	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
ax	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
bx	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
cx	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
dx	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
ex	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
fx	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Figure1-10: Inverse-sBox

Add RoundKey

Follows the same way shown in the Encryption Add RoundKey section. But the different is in every iteration, we use the round key in the order of from group 11 to group 2.

Inverse MixColumn

Doing Inverse MixColumn, we use the intermediate result from the Add RoundKey to multiply with a 4 by 4 matrix. The matrix is shown below:

$$\begin{bmatrix} 0e & 0b & 0d & 09 \\ 09 & 0e & 0b & 0d \\ 0d & 09 & 0e & 0b \\ 0b & 0d & 09 & 0e \end{bmatrix}$$

Figure1-11: the matrix used for inverse-mixcolumn

And the Finite Field Arithmetic for this matrix:

1. $a * 0x0d = (a * 0x02 * 0x02 * 0x02) + (a * 0x02 * 0x02) + (a * 0x01)$
2. $a * 0x09 = (a * 0x02 * 0x02 * 0x02) + (a * 0x01)$
3. $a * 0x0b = (a * 0x02 * 0x02 * 0x02) + (a * 0x02) + (a * 0x01)$
4. $a * 0x0e = (a * 0x02 * 0x02 * 0x02) + (a * 0x02 * 0x02) + (a * 0x02)$

Iteration

Do 9 iterations of Inverse ShiftRows, Inverse ByteSubstitution, Add RoundKey and Inverse MixColumn. And then do 1 iteration of ShiftRows, Inverse ByteSubstitution and Add RoundKey.

Hardware/Software information

Software

Programming language: BSV, Verilog

IDE: Bluespec

Hardware

Emulation tools: ModuleSim, Libero

Core: ARM® CortexTM-M1 Embedded Processor

Hardware: Fusion Mixed Signal FPGA

Design Methodology

Software design

Module design

When I build my program, first thing came into my mind is to hard code some essential data. For example when I do the encryption, I have to hard code the input plain text, cipher key, sBox, Rcon and mix column matrix. This may make my program more easy to test when building the module.

And after finish the module part, then I remove the input plain text and cipher key, that's because the module need to provide some interface to fetch input data such as the text and cipher key, since we don't want this module can only encryption the build-in plain text, this may cause the module useless.

After providing the interface from the module, I made a test bench to provide the input data. And I also considered that since the module can do either encryption and decipher, so the test bench are supposed to have the ability to control the module to do encryption or decipher.

```
//----- Encryption part -----//
Reg#(Bit#(8)) state[4][4];          //Used to store encryption intermediate value
Reg#(Bit#(8)) keym[4][4];           //Used to store the value of the cipher key

Reg#(Bit#(8)) sbox[16][16];
Bit#(8) s_box[16][16]={
    // 0   1   2   3   4   5   6   7   8   9   a   b   c   d   e   f */
    {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}, // 0
    {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}, // 1
    {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}, // 2
    {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}, // 3
    {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}, // 4
    {8'h53,8'hdl,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}, // 5
    {8'hd0,8'hef,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}, // 6
    {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}, // 7
    {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}, // 8
    {8'h06,8'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}, // 9
    {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'hea,8'h79}, // a
    {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}, // b
    {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}, // c
    {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}, // d
    {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}, // e
    {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} // f
};
```

Figure2-1: Declare variables and hard code essential data

As you can see in Figure2-1, when I declare the variables, I use:

```
Reg#(Bit#(8)) invsbox[16][16];
```

That's because in Bluespec, you can declare the variables without "Reg#" and run the code without any error. But the thing is we have to converter the BSV into Verilog. And if there is no "Reg#" before the variable, then the data store in it will vanish. The "Reg#" means register, it maps to the hardware register where all the data are supposed to stored in it.

It's important to initialize no matter variables or matrixes. Because if you don't initialize it, you can't do further process with that variable or matrix, and you will always get "0xaa" which is the default value of the variable or matrix elements.

In the appendix, you can see from I first generate the round key(a 4 by 44 matrix) from line 327 to line 338, is to use cipher key to initialize the round key first group. In my code, I generate the first column in each group(line 342 - line 410), and then generate the rest three column in that group(line 415 - line 425). From line 427 - line 450 is the for loop: k_rk and p_rk are the counter, the p_rk is the inner loop and the k_rk is the outer loop. The p_rk add 1 every loop and then set the state to 11 where is the loop beginning. And when p_rk exceed 3, it will empty itself(line 433) and then jump out of the inner loop and go to the outer loop, which k_rk will add 1 and jump back to the loop beginning and then go into inner loop again.

The reason why I don't use the normal "for loop" is because, in BSV, every rule represent a clock cycle, within the same clock cycle, the value of the variable couldn't change, it have to wait until next clock cycle to update the new value. But if I code with "for loop", I have to include a lot of variable and they have to change their value instantly, if not, it will cause the wrong result. And that is way it's not possible to code the whole for loop in a same clock cycle(same rule). And the you can't use the same "for loop" in different rules because the counter can't control other rules.

From line 455 - line 466, is where the test bench can send encryption/decipher signal to control the module to do encryption/decipher. The principle is simple, because I use rule to build up my code, so I simply used a "if-else" function to determine whether jump to the rule that can start doing encryption or to the rule that can do the decipher.

And in both line 714 and line 1104, where are the end of encryption function and decipher function respectively. I use a integer variable "cp" which it's initial value is 0, but when it change to 1, it will out put to test bench through the interface, and let the test bench start to fetch the final result from the module, more details about the test bench, I will explain later.

From line 1115 to line 1227 is where I define the method for the interface. For method: loadInput, importKey and loadDestate can carry 32 bits per clock cycle. And they are for import plain text, cipher key and cipher text. We can choose to load plain text or cipher text depends on what we want to do. Either plain text or cipher text can have not input and then the matrix will initialize to aa, and won't cause any error. For example, if we want to do the encryption, then we only need to load the plain text and the cipher key, then the module will only run the encryption part, and only get the cipher text as an output.

Method: check_signal is the method that can get the control signal from the text bench and control the module to do encryption/decipher. Method: checkpoint can return a integer to the text

bench and let the test bench to continue process the next code. Method: output_state and output_result are two method that can return the final result by 8 bit per clock cycle to the test bench.

test bench design

In the test bench, I simply give the module input by every clock cycle as you can see from line 17 to line 68. And in each rule, I call the method in module and give the method two parameters, the first one is a 32 bits input data, and the second one is the address that tells the method which row and column should the data been stored. From line 71 to line 76 is where we can control the module to do encryption or decipher by call check_signal method and give it a parameter either 1 or 0 indicate encryption or decipher. And this is the essential data that we need to transfer into the module.

In the test bench, when the code has been execute to line 76, the module will start to generate the result, so normally it will take some time, and we don't know when it will finish, so the check point signal becomes very import, as you can see from line 79, the rule: getoutput will first check the method: checkpoint return value, if it's still 0(it's initial value), it will keep the state(named as flag) at 400, which is same as a while loop, and wait the module to finish the processing. When the module finished, it will also turn the variable: cp into 1, which when check the method: checkpoint return value, will let the rule: getoutput start to execute line 83.

And from line 93 to line 129 is just let the result output get ready for display. And from line 132 to the end, is just display the final result.

Hardware design

Building SOC

I followed the ARM Cortex-M1 embedded processor tutorial to build up the SOC. And at the first step I try to set up the USB-tu-UART driver, and it doesn't work, then I try to reboot the computer and found out it still doesn't work. So I login to another desktop and found out that computer has already install the drive then I contented to do the next step.

Building up the SOC is just follow the instructions in the tutorial, but the tutorial is not the actual SOC that was used in the lab(I decided to build the lab use SOC and then do some changes on it to make it suitable for this project) so I switch to use the lab manual start from page 49 when I finished Step 2- create a SmartDesign component with Libero SOC.

The next issue I run into is when I try to add comment on the file: M1AFS_EMB_KIT.PDC. At first, I thought I should not only comment out the dataIn[0] and dataIn[1] but also add some code shown in below:

```
388 #Replace the above SW2 and SW3 pin assignments with the following lines
389 set_io dataIn -pinname P18 -fixed yes -DIRECTION Input
390 set_io dataIn_0 -pinname R18 -fixed yes -DIRECTION Input
391
392 #The dataOut[3:0] should have the same assignments for the LEDs
393 set_io {dataOut[0]} -pinname D8 -fixed yes -DIRECTION Output
394 set_io {dataOut[1]} -pinname D9 -fixed yes -DIRECTION Output
395 set_io {dataOut[2]} -pinname D10 -fixed yes -DIRECTION Output
396 set_io {dataOut[3]} -pinname D6 -fixed yes -DIRECTION Output
397
398 #Add the following assignments for the TSP port
399 set_io {TPS[4]} -pinname T19 -fixed yes -DIRECTION Output
400 set_io {TPS[3]} -pinname L1 -fixed yes -DIRECTION Output
401 set_io {TPS[2]} -pinname T3 -fixed yes -DIRECTION Output
402 set_io {TPS[1]} -pinname P3 -fixed yes -DIRECTION Output
403 set_io {TPS[0]} -pinname P2 -fixed yes -DIRECTION Output
```

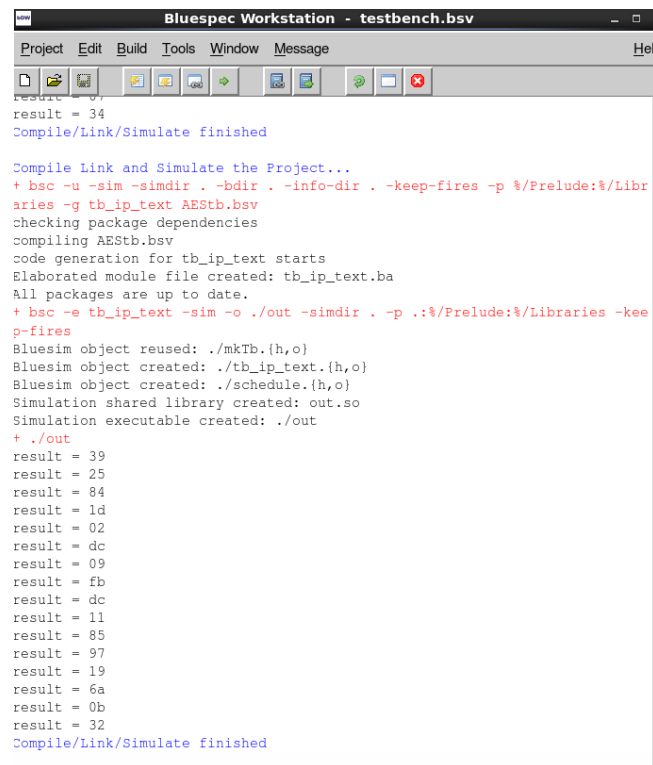
Figure2-2: Few changes for M1AFS_EMB_KIT.PDC

But when I add all those code, it complains error, which says those constraints are not exist on the the hardware. After trying for a few times, I found out that when I comment out the dataIn[0] and dataIn[1], I should only add line 389 and line 390 as you can see from Figure2-2.

For the component error: Error: PDC01: port name doesn't exist in the netlist or is not connected to an IoCell macro at PDC Line : set_io RX -pinname Y1 -fixed yes -DIRECTION Input. I tried both add a INBUF and use a new version of UARTapb to replace the old one. And found out the first one can't fix my problem, but the second one works perfect for me.

And then I continue to follow the steps in the ARM Cortex-M1 Embedded processor tutorial and when I try to implement Step 10 - Debugging the Project and run into a break point that will always terminal the program. And I stuck at this step and don't know how to continue.

Results

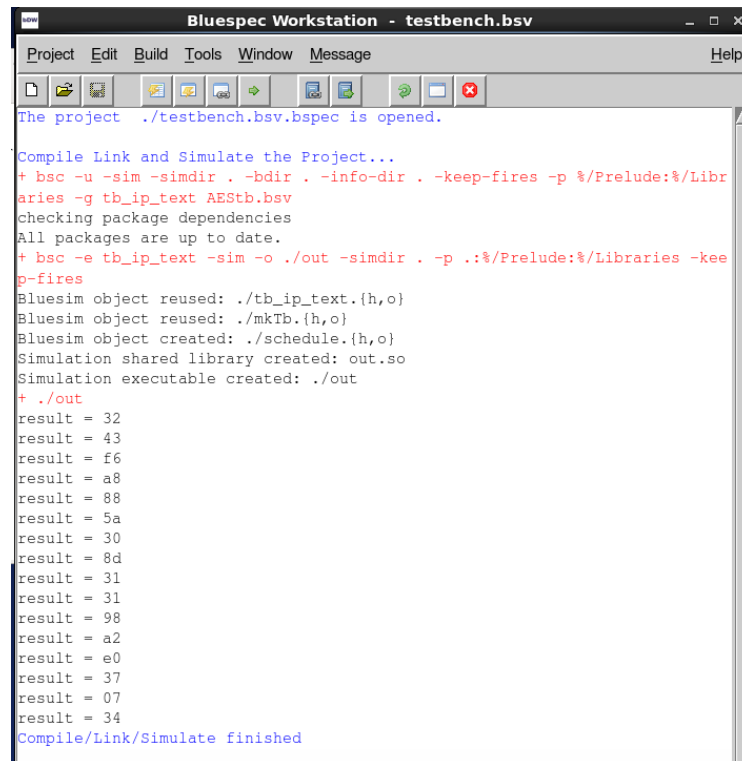


```
Bluespec Workstation - testbench.bsv
Project Edit Build Tools Window Message Help

result = 07
result = 34
Compile/Link/Simulate finished

Compile Link and Simulate the Project...
+ bsc -u -sim -simdir . -bdir . -info-dir . -keep-fires -p %/Prelude:/Libraries -g tb_ip_text AESTb.bsv
checking package dependencies
compiling AESTb.bsv
code generation for tb_ip_text starts
Elaborated module file created: tb_ip_text.ba
All packages are up to date.
+ bsc -e tb_ip_text -sim -o ./out -simdir . -p .:/Prelude:/Libraries -keep-fires
Bluesim object reused: ./mkTb.{h,o}
Bluesim object created: ./tb_ip_text.{h,o}
Bluesim object created: ./schedule.{h,o}
Simulation shared library created: out.so
Simulation executable created: ./out
+ ./out
result = 39
result = 25
result = 84
result = 1d
result = 02
result = dc
result = 09
result = fb
result = dc
result = 11
result = 85
result = 97
result = 19
result = 6a
result = 0b
result = 32
Compile/Link/Simulate finished
```

Figure3-1: Encryption



```
Bluespec Workstation - testbench.bsv
Project Edit Build Tools Window Message Help

The project ./testbench.bsv.bspect is opened.

Compile Link and Simulate the Project...
+ bsc -u -sim -simdir . -bdir . -info-dir . -keep-fires -p %/Prelude:/Libraries -g tb_ip_text AESTb.bsv
checking package dependencies
All packages are up to date.
+ bsc -e tb_ip_text -sim -o ./out -simdir . -p .:/Prelude:/Libraries -keep-fires
Bluesim object reused: ./tb_ip_text.{h,o}
Bluesim object reused: ./mkTb.{h,o}
Bluesim object created: ./schedule.{h,o}
Simulation shared library created: out.so
Simulation executable created: ./out
+ ./out
result = 32
result = 43
result = f6
result = a8
result = 88
result = 5a
result = 30
result = 8d
result = 31
result = 31
result = 98
result = a2
result = e0
result = 37
result = 07
result = 34
Compile/Link/Simulate finished
```

Figure3-2: Decipher

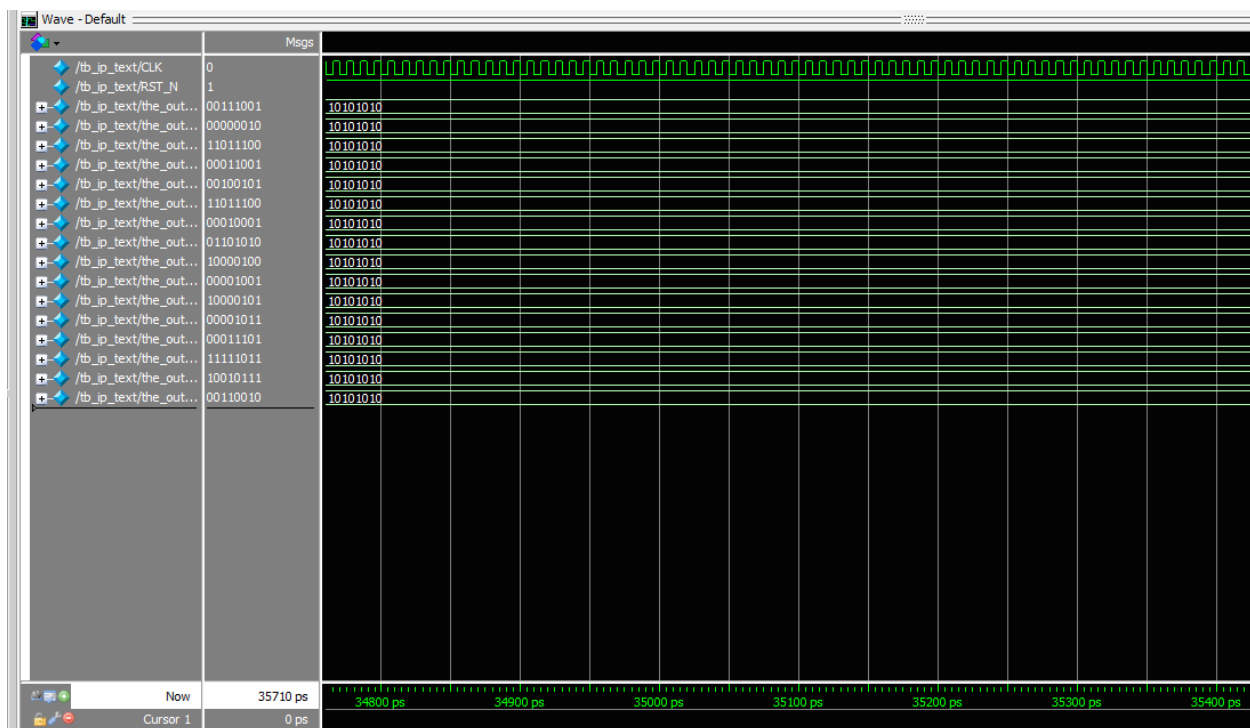


Figure3-3: ModuleSim encryption start

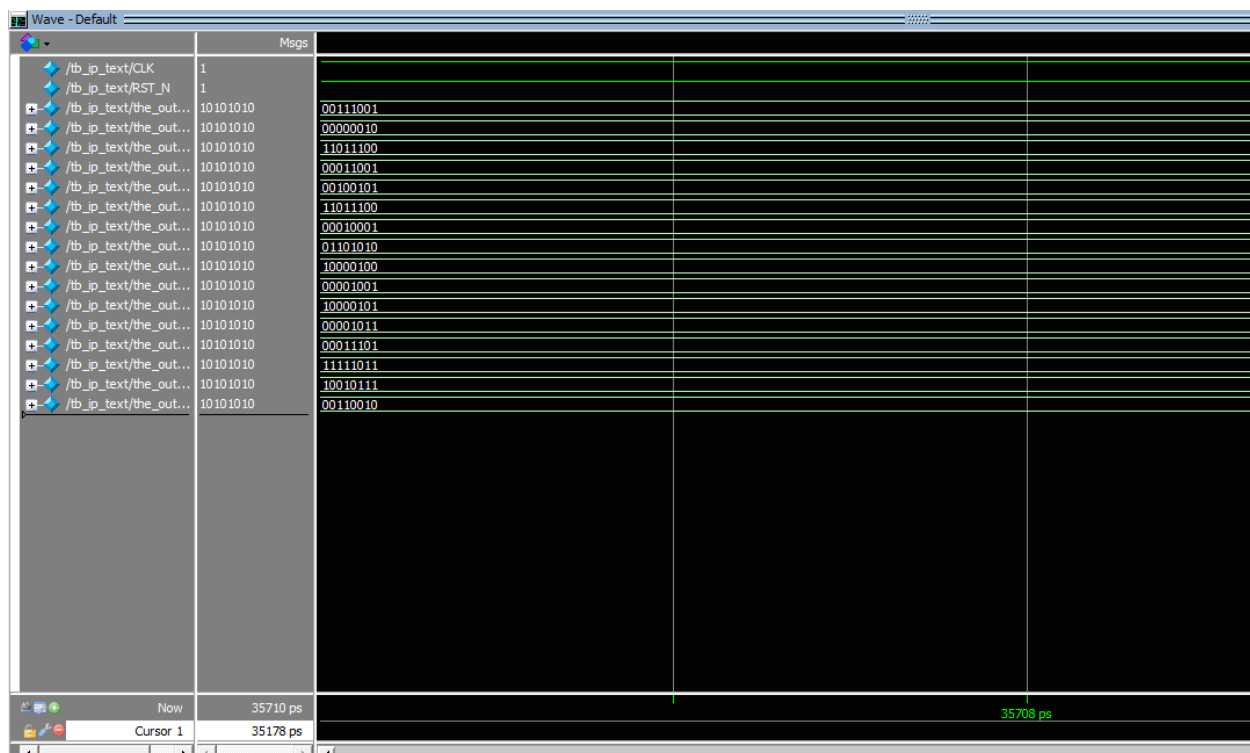


Figure3-4: ModuleSim encryption finish

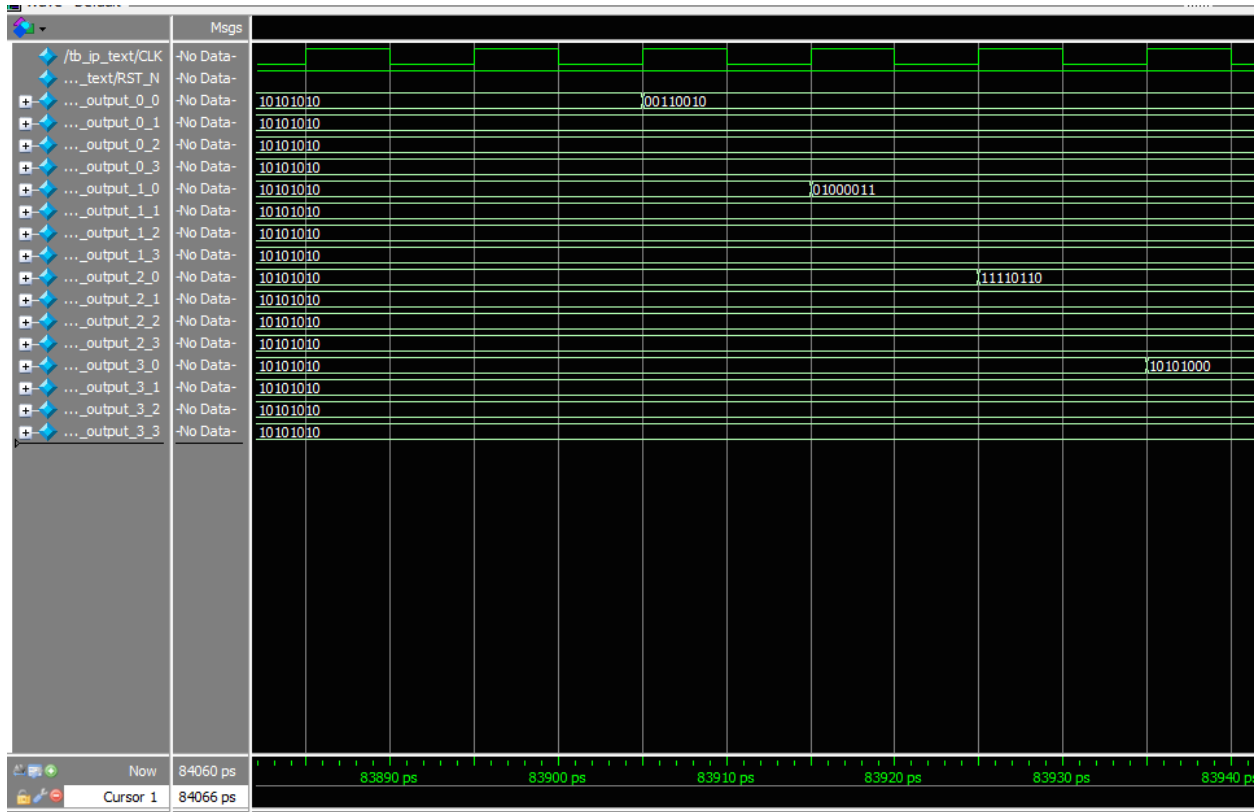


Figure3-5: ModuleSim decipher processing

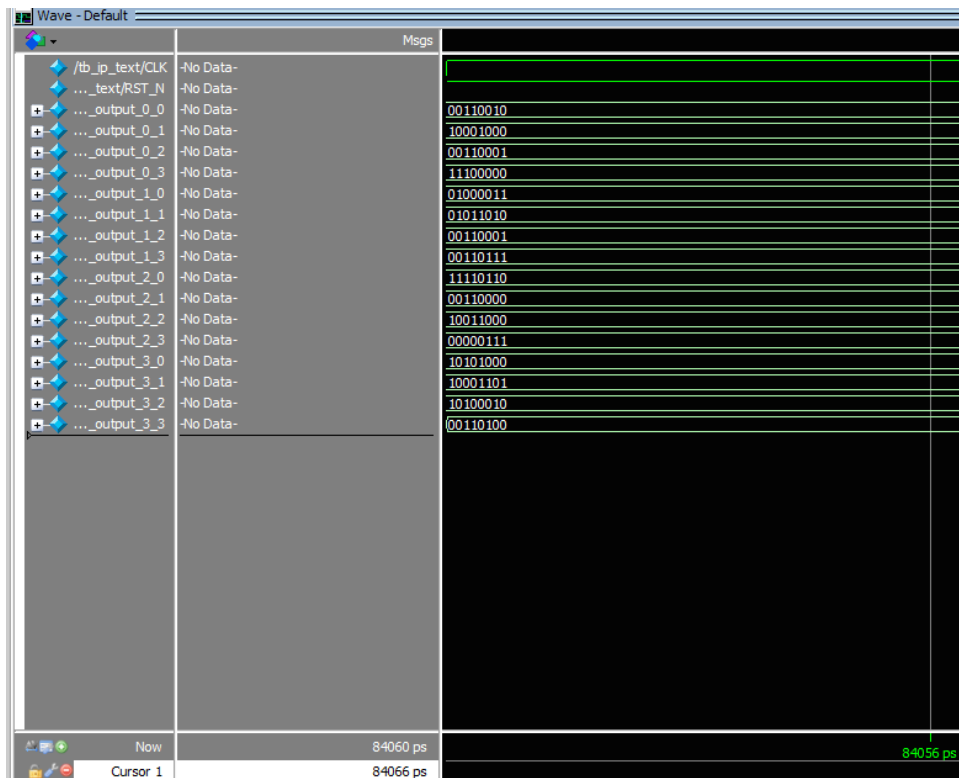


Figure3-6: ModuleSim decipher finish

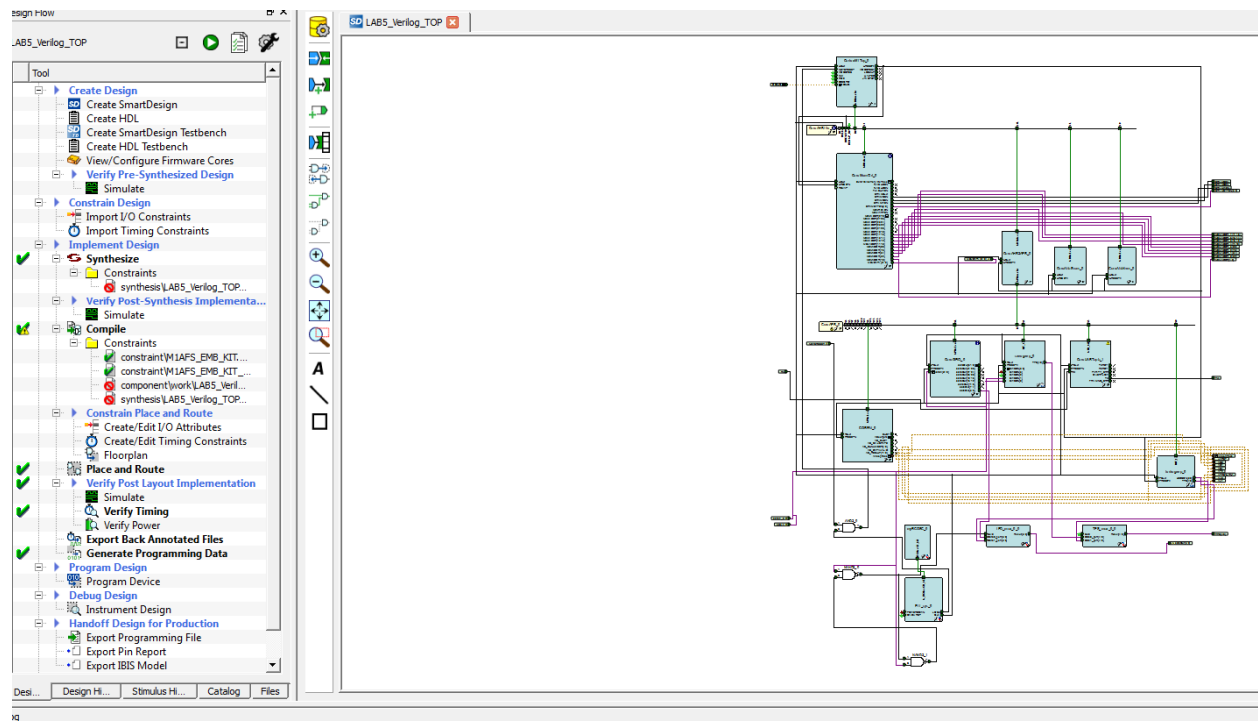


Figure3-7: Libero simulate succeed

Conclusions

In this project, I finished the software part which contains test bench and module encryption and decipher. I try to to make some change on lab 5's wrapper to make a wrapper for the module, but failed. That wrapper can input 32 bit data per clock cycle into the module but can't get the correct result. Maybe is because the output part interface have some error. Or maybe the input data interface have some error that the wrong data has been transferred in to the module. I try to build up the SOC and it did work fine but I run in to some issue when I try to do the soft control using the tutorial as a reference.

Reference

1. ARM Cortex-M1 Embedded Processor Tutorial
2. Lab Manual ENGG4560 W15 Edition 1
3. BSV_by_example
4. Federal Information Processing Standards Publication 197

Appendix - AES module

```
1 package AESchange;
2
3 interface Ifc_main_type;
4     method Action loadInput(Bit#(32) dIn, int addr);
5     method Action importKey(Bit#(32) imKey, int addr);
6     method Action loadDestate(Bit#(32) des, int addr);
7     method Action check_signal(int signal);
8     method int checkpoint ();
9     method int output_signal();
10    method Bit#(8) output_state(int i, int j);
11    method Bit#(8) output_result(int i, int j);
12    method Bit#(32) checkTemp();
13 endinterface: Ifc_main_type
14
15 (*synthesize*)
16 module mkTb (Ifc_main_type);
17
18
19 ///////////////////////////////////////////////////
20 //                      Essential data
21 ///////////////////////////////////////////////////
22
23 //----- Encryption part -----//
24
25 Reg#(Bit#(8)) state[4][4];           //Used to store encryption intermediate value
26
27 Reg#(Bit#(8)) keym[4][4];           //Used to store the value of the cipher key
28
29 Reg#(Bit#(8)) sbox[16][16];
30 Bit#(8) s_box[16][16]={
31     // 0      1      2      3      4      5      6      7      8      9      a      b      c      d      e      f */
32     {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}, // 0
33     {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}, // 1
34     {8'hb7,8'hfd,8'h93,8'h26,8'h36,8'h3f,8'hf7,8'hcc,8'h34,8'ha5,8'he5,8'hfl,8'h71,8'hd8,8'h31,8'h15}, // 2
35     {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}, // 3
36     {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}, // 4
37     {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}, // 5
38     {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}, // 6
39     {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}, // 7
40     {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}, // 8
41     {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}, // 9
42     {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}, // a
43     {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}, // b
44     {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}, // c
45     {8'h70,8'h3e,8'hb5,8'h66,8'h46,8'h03,8'hf6,8'h0e,8'h61,8'h35,8'h57,8'hb9,8'h86,8'hcl,8'h1d,8'h9e}, // d
46     {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}, // e
47     {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}, // f
48     };
49
50 Reg#(Bit#(8)) rcon[4][10];
51 Bit#(8) r_con[4][10]={
52     // 0      1      2      3
53     {8'h01, 8'h02, 8'h04, 8'h08, 8'h10, 8'h20, 8'h40, 8'h80, 8'h1b, 8'h36}, // 0
54     {8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00}, // 1
55     {8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00}, // 2
56     {8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00, 8'h00}, // 3
57     };
58
59 Reg#(Bit#(8)) mix[4][4];
60 Bit#(8) m_ix[4][4] = {
61     // 0      1      2      3
62     {8'h02, 8'h03, 8'h01, 8'h01}, // 0
63     {8'h01, 8'h02, 8'h03, 8'h01}, // 1
64     {8'h01, 8'h01, 8'h02, 8'h03}, // 2
65     {8'h03, 8'h01, 8'h01, 8'h02}, // 3
66     };
67
68 //----- Decipher part -----//
69
70 Reg#(Bit#(8)) resultt[4][4];           //Used to store decipher intermediate data
71
72 Reg#(Bit#(8)) invsbox[16][16];
73 Bit#(8) inv_sbox[16][16]={
74     {8'h52,8'h09,8'h6a,8'hd5,8'h30,8'h36,8'ha5,8'h38,8'hbf,8'h40,8'ha3,8'h9e,8'h81,8'hf3,8'hd7,8'hfb}, /*0*/
75     {8'h7c,8'he3,8'h39,8'h82,8'h9b,8'h2f,8'hff,8'h87,8'h34,8'h8e,8'h43,8'h44,8'hc4,8'hde,8'he9,8'hcb}, /*1*/
76     {8'h54,8'h7b,8'h94,8'h32,8'ha6,8'hc2,8'h23,8'h3d,8'hee,8'h4c,8'h95,8'h0b,8'h42,8'hfa,8'hc3,8'h4e}, /*2*/
77     {8'h08,8'h2e,8'ha1,8'h66,8'h28,8'hd9,8'h24,8'hb2,8'h76,8'h5b,8'ha2,8'h49,8'h6d,8'h8b,8'hd1,8'h25}, /*3*/
78     {8'h72,8'hf8,8'hf6,8'h64,8'h68,8'h98,8'h16,8'hd4,8'ha4,8'h5c,8'hcc,8'h5d,8'h65,8'hb6,8'h92}, /*4*/
79     {8'h6c,8'h70,8'h48,8'h50,8'hfd,8'hed,8'hb9,8'hda,8'h5e,8'h15,8'h46,8'h57,8'ha7,8'h8d,8'h9d,8'h84}, /*5*/
80     {8'h90,8'hd8,8'hab,8'h00,8'h8c,8'hbc,8'hd3,8'h0a,8'hf7,8'he4,8'h58,8'h05,8'hb8,8'hb3,8'h45,8'h06}, /*6*/
81     {8'hd0,8'h2c,8'h1e,8'h8f,8'hca,8'h3f,8'h0f,8'h02,8'hcl,8'haf,8'hbd,8'h03,8'h01,8'h13,8'h8a,8'h6b}, /*7*/
82     {8'h3a,8'h91,8'h11,8'h41,8'h4f,8'h67,8'hdc,8'hea,8'h97,8'hf2,8'hcf,8'hce,8'hf0,8'hb4,8'he6,8'h73}, /*8*/
83     {8'h96,8'hac,8'h74,8'h22,8'he7,8'had,8'h35,8'h85,8'he2,8'hf9,8'h37,8'he8,8'h1c,8'h75,8'hdf,8'h6e}, /*9*/
84     {8'h47,8'hf1,8'h1a,8'h71,8'h1d,8'h29,8'hc5,8'h89,8'h6f,8'hb7,8'h62,8'h0e,8'haa,8'h18,8'hbe,8'h1b}, /*a*/
85     {8'hfc,8'h56,8'h3e,8'h4b,8'hc6,8'hd2,8'h79,8'h20,8'h9a,8'hdb,8'hc0,8'hfe,8'h78,8'hcd,8'h5a,8'hf4}, /*b*/
86     {8'h1f,8'hdd,8'ha8,8'h33,8'h88,8'h07,8'hc7,8'h31,8'hb1,8'h12,8'h10,8'h59,8'h27,8'h80,8'hec,8'h5f}, /*c*/
87     {8'h60,8'h51,8'h7f,8'ha9,8'h19,8'hb5,8'h4a,8'h0d,8'h2d,8'he5,8'h7a,8'h9f,8'h93,8'hc9,8'h9c,8'hcf}, /*d*/
88     {8'ha0,8'he0,8'h3b,8'h4d,8'hae,8'h2a,8'hf5,8'hb0,8'hc8,8'heb,8'hbb,8'h3c,8'h83,8'h53,8'h99,8'h61}, /*e*/
89     {8'h17,8'h2b,8'h04,8'h7e,8'hba,8'h77,8'hd6,8'h26,8'he1,8'h69,8'h14,8'h63,8'h55,8'h21,8'h0c,8'h7d}, /*f*/
90     };
91
92 Reg#(Bit#(8)) invmix[4][4];
93 Bit#(8) inv_mix[4][4] = {
94     {8'h0e, 8'h0b, 8'h0d, 8'h09},
95     {8'h09, 8'h0e, 8'h0b, 8'h0d},
96     {8'h0d, 8'h09, 8'h0e, 8'h0b},
97     {8'h0e, 8'h09, 8'h0e, 8'h0b},
98     };
99 }
```

```

97         {8'h0b, 8'h0d, 8'h09, 8'h0e}
98     };
99
100
101 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
102 //                               Declare variables
103 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
104     Reg#(Bit#(8)) keySched[4][44];                                // Used to store the round key
105                                                                    // the round key can be divide into 11 groups
106                                                                    // each group contains 4 by 4 elements
107
108     Reg#(Bit#(8)) mReword[4][10];                                // Used to store every first column of the each group
109
110     Reg#(int) step <- mkReg(700);                                // The state of the rules
111
112 //----- Generate round key -----//
113     Reg#(int) i_rk <- mkReg(0);
114     Reg#(Bit#(8)) tempFirstBit <- mkRegU;
115     Reg#(Bit#(8)) tempShiftBits[4][1];
116     Reg#(int) z_rk <- mkReg(0);
117     Reg#(Bit#(8)) tempSub <- mkRegU;
118     Reg#(Bit#(4)) tempSub1 <- mkRegU;
119     Reg#(Bit#(4)) tempSub2 <- mkRegU;
120     Reg#(Bit#(8)) tempRcon <- mkRegU;
121     Reg#(Bit#(8)) tempSubOne[4][1];                                // Fetch the previous row
122     Reg#(Bit#(8)) tempSubFour[4][1];                                // Fetch the row which row number is 4 smaller than the current o
123     Reg#(int) k_rk <- mkReg(1);
124     Reg#(int) p_rk <- mkReg(0);
125
126 //----- AES encryption -----//
127     Reg#(int) a <- mkReg(0);
128
129 //----- AES encrypton: Byte substitution -----//
130     Reg#(int) b <- mkReg(0);
131     Reg#(int) c <- mkReg(0);
132     Reg#(Bit#(8)) tSub <- mkRegU;
133     Reg#(Bit#(4)) tSub1 <- mkRegU;
134     Reg#(Bit#(4)) tSub2 <- mkRegU;
135
136 //----- AES encrypton: Shift rows -----//
137     Reg#(Bit#(8)) tempshift1 <- mkRegU;
138     Reg#(Bit#(8)) tempshift2 <- mkRegU;
139     Reg#(Bit#(8)) tempshift3 <- mkRegU;
140     Reg#(Bit#(8)) tempshift4 <- mkRegU;
141     Reg#(Bit#(8)) tempshift5 <- mkRegU;
142     Reg#(Bit#(8)) tempshift6 <- mkRegU;
143     Reg#(Bit#(8)) tempshift7 <- mkRegU;
144
145 //----- AES encrypton: Mixcolumn -----//
146     Reg#(int) d <- mkReg(0);
147     Reg#(int) e <- mkReg(0);
148     Reg#(int) f <- mkReg(0);
149     Reg#(Bit#(8)) tempstate[4][4];
150     Reg#(Bit#(8)) tempmix <- mkRegU;
151     Reg#(Bit#(8)) tempfile <- mkRegU;
152     Reg#(Bit#(8)) tempfile1 <- mkRegU;
153
154 //----- AES encrypton: Add round key -----//
155     Reg#(int) h <- mkReg(0);
156     Reg#(int) g <- mkReg(0);
157
158 //----- AES decipher -----//
159     Reg#(int) k <- mkReg(0);
160     Reg#(Bit#(8)) tshift1 <- mkRegU;
161     Reg#(Bit#(8)) tshift2 <- mkRegU;
162     Reg#(Bit#(8)) tshift3 <- mkRegU;
163     Reg#(Bit#(8)) tshift4 <- mkRegU;
164     Reg#(Bit#(8)) tshift5 <- mkRegU;
165     Reg#(Bit#(8)) tshift6 <- mkRegU;
166     Reg#(Bit#(8)) tshift7 <- mkRegU;
167     Reg#(Bit#(8)) tshift8 <- mkRegU;
168     Reg#(Bit#(8)) tshift11 <- mkRegU;
169     Reg#(Bit#(8)) tshift22 <- mkRegU;
170     Reg#(Bit#(8)) tshift33 <- mkRegU;
171     Reg#(Bit#(8)) tshift44 <- mkRegU;
172
173 //----- AES decipher: Inverse byte substitution -----//
174     Reg#(int) bz <- mkReg(0);
175     Reg#(int) cz <- mkReg(0);
176     Reg#(Bit#(8)) tSb <- mkRegU;
177     Reg#(Bit#(4)) tSb1 <- mkRegU;
178     Reg#(Bit#(4)) tSb2 <- mkRegU;
179
180 //----- AES decipher: Inverse add round key -----//
181     Reg#(int) gz <- mkReg(0);
182     Reg#(int) hz <- mkReg(0);
183
184 //----- AES decipher: Inverse mixcolumn -----//
185     Reg#(Bit#(8)) invtempstate[4][4];
186     Reg#(Bit#(8)) invtempmix <- mkRegU;
187     Reg#(Bit#(8)) invtempfile <- mkRegU;
188     Reg#(Bit#(8)) invtempfileOri <- mkRegU;
189     Reg#(Bit#(8)) invtempfileSec <- mkRegU;
190     Reg#(Bit#(8)) invtempfileFir <- mkRegU;
191     Reg#(Bit#(8)) invtempfileThr <- mkRegU;
192     Reg#(int) id <- mkReg(0);
193     Reg#(int) ie <- mkReg(0);

```

```

194 Reg#(int) ff <- mkReg(0);
195
196 //----- The interface -----//
197 Reg#(Bit#(8)) statetemp[4][4];
198 Reg#(Bit#(8)) keymtemp[4][4];
199 Reg#(Bit#(8)) resultttemp[4][4];
200 Reg#(int) sigtemp <- mkReg(0);
201
202 Reg#(int) i_int <- mkReg(0);
203 Reg#(int) j_int <- mkReg(0);
204 Reg#(int) sig <- mkReg(0);
205 Reg#(int) cp <- mkReg(0);
206
207
208 //////////////////////////////////////
209 //                               Initialize matrix
210 //////////////////////////////////////
211
212 //----- The interface -----//
213 for(Integer i = 0; i<4; i = i+1)
214     for(Integer j = 0; j<4; j = j+1)
215         statetemp[j][i] <- mkReg(0);
216
217 for(Integer i = 0; i<4; i = i+1)
218     for(Integer j = 0; j<4; j = j+1)
219         keymtemp[j][i] <- mkReg(0);
220
221 for(Integer i = 0; i<4; i = i+1)
222     for(Integer j = 0; j<4; j = j+1)
223         resultttemp[j][i] <- mkReg(0);
224
225 //----- Some essential matrix -----//
226 for(Integer i = 0; i<4; i = i+1)
227     for(Integer j = 0; j<4; j = j+1)
228         state[j][i] <- mkRegU;
229
230 for(Integer i = 0; i<4; i = i+1)
231     for(Integer j = 0; j<4; j = j+1)
232         keym[j][i] <- mkRegU;
233
234 for(Integer i = 0; i<4; i = i+1)
235     for(Integer j = 0; j<4; j = j+1)
236         resultt[j][i] <- mkRegU;
237
238 for(Integer i = 0; i<16; i = i+1)
239     for(Integer j = 0; j<16; j = j+1)
240         sbox[j][i] <- mkRegU;
241
242 for(Integer i = 0; i<44; i = i+1)
243     for(Integer j = 0; j<4; j = j+1)
244         keySched[j][i] <- mkRegU;
245
246 for(Integer i = 0; i<10; i = i+1)
247     for(Integer j = 0; j<4; j = j+1)
248         mReword[j][i] <- mkRegU;
249
250 //----- Round key -----//
251 for(Integer i = 0; i<4; i = i+1)
252     tempShiftBits[i][0] <- mkRegU;
253
254 for(Integer i = 0; i<10; i = i+1)
255     for(Integer j = 0; j<4; j = j+1)
256         rcon[j][i] <- mkRegU;
257
258 for(Integer i = 0; i<4; i = i+1)
259     tempSubOne[i][0] <- mkRegU;
260
261 for(Integer i = 0; i<4; i = i+1)
262     tempSubFour[i][0] <- mkRegU;
263
264 for(Integer i = 0; i<4; i = i+1)
265     test[i] <- mkReg(33);
266
267 //----- AES encryption mixcolumn -----//
268 for(Integer i = 0; i<4; i = i+1)
269     for(Integer j = 0; j<4; j = j+1)
270         tempstate[j][i] <- mkRegU;
271
272 for(Integer i = 0; i<4; i = i+1)
273     for(Integer j = 0; j<4; j = j+1)
274         mix[j][i] <- mkRegU;
275
276 //----- AES decipher -----//
277 for(Integer i = 0; i<16; i = i+1)
278     for(Integer j = 0; j<16; j = j+1)
279         invsbox[j][i] <- mkRegU;
280
281 for(Integer i = 0; i<4; i = i+1)
282     for(Integer j = 0; j<4; j = j+1)
283         invmix[j][i] <- mkRegU;
284
285 for(Integer i = 0; i<4; i = i+1)
286     for(Integer j = 0; j<4; j = j+1)
287         invtempstate[j][i] <- mkRegU;
288
289
290 //////////////////////////////////////

```

```

291 //          Prepare for AES
292 //////////////////////////////////////////////////
293
294 //----- Put data into register -----//
295 rule init_statel_1 (step == 0);
296     for(Integer i = 0; i<4; i = i+1)
297         for(Integer j = 0; j<4; j = j+1)
298             state[j][i] <= statetemp[j][i];
299
300     for(Integer i = 0; i<4; i = i+1)
301         for(Integer j = 0; j<4; j = j+1)
302             keym[j][i] <= keymtemp[j][i];
303
304     for(Integer i = 0; i<4; i = i+1)
305         for(Integer j = 0; j<4; j = j+1)
306             resultt[j][i] <= resultttemp[j][i];
307
308     for(Integer i = 0; i<16; i = i+1)
309         for(Integer j = 0; j<16; j = j+1)
310             sbox[j][i] <= s_box[j][i];
311
312     for(Integer i = 0; i<10; i = i+1)
313         for(Integer j = 0; j<4; j = j+1)
314             rcon[j][i] <= r_con[j][i];
315
316     for(Integer i = 0; i<4; i = i+1)
317         for(Integer j = 0; j<4; j = j+1)
318             mix[j][i] <= m_ix[j][i];
319     step <= 1;
320 endrule
321
322 //////////////////////////////////////////////////
323 //          Generate round key
324 //////////////////////////////////////////////////
325
326 //----- Initialize the first 4*4 round key using key -----//
327 rule init_statel (step == 1);
328     for (Integer ia = 0; ia < 44; ia = ia+1)
329         for (Integer ja = 0; ja < 4; ja = ja+1)begin
330             if (ia < 4)begin
331                 keySched[ja][ia] <= keym[ja][ia];
332             end
333             else begin
334                 keySched[ja][ia] <= 8'h00;
335             end
336         end
337     step <= 2;
338 endrule
339
340 //----- Generate every first column in each group in rest of the round key -----//
341 // i_rk change from 0 - 9
342 rule rest_sched (step == 2);
343     for (Integer j = 0; j < 4; j = j+1)
344         mReword[j][i_rk] <= keySched[j][(i_rk+1)*4-1];
345     step <= 3;
346 endrule
347
348 rule rest_sched1 (step == 3);
349     tempFirstBit <= keySched[0][(i_rk+1)*4-1];
350     step <= 4;
351 endrule
352
353 // 1.0. Move each row's mReword parameters 1 bit left
354 rule rest_sched2 (step == 4);
355     for (Integer p = 1; p < 4; p = p+1)begin
356         tempShiftBits[p-1][0] <= mReword[p][i_rk];
357     end
358     step <= 5;
359 endrule
360
361 // 1.2. Put the original first parameter of mReword at the end
362 //this step can achieve each row shift left one bit
363 rule rest_sched3 (step == 5);
364     tempShiftBits[3][0] <= tempFirstBit;
365     step <= 6;
366 endrule
367
368 // 2.0. Doing Subword
369 // z_rk change from 0 - 3
370 rule rest_sched4 (step == 6);
371     mReword[z_rk][i_rk] <= tempShiftBits[z_rk][0];
372     step <= 200;
373 endrule
374
375 rule rest_sched4_0 (step == 200);
376     tempSub <= mReword[z_rk][i_rk];
377     step <= 7;
378 endrule
379
380 // 2.1. Fetch X & Y
381 rule rest_sched4_1 (step == 7);
382     tempSub1 <= tempSub[7:4];
383     tempSub2 <= tempSub[3:0];
384     step <= 8;
385 endrule
386
387

```

```

388 // 2.1. Doing Subword, check the value using X and Y
389 rule rest_sched4_2 (step == 8);
390     tempRcon <= sbox[tempSub1][tempSub2];
391     step <= 9;
392 endrule
393
394 // 2.2. Do XOR with Rcon
395 // 2.2. Do XOR with the same row but previous group
396 rule rest_sched4_3 (step == 9);
397     keySched[z_rk][(i_rk+1)*4] <= tempRcon ^ rcon[z_rk][i_rk] ^ keySched[z_rk][i_rk*4];
398     step <= 10;
399 endrule
400
401 rule rest_sched5 (step == 10);
402     if (z_rk < 3)begin
403         z_rk <= z_rk+1;
404         step <= 6;
405     end
406     else begin
407         z_rk <= 0;
408         step <= 11;
409     end
410 endrule
411
412 //----- Generate the rest 3 column in each group -----//
413 // k_rk change from 1 - 3
414 // p_rk change from 0 - 3
415 rule rest_sched6 (step == 11);
416     //doing Subword, fetch X and Y
417     tempSubOne[p_rk][0] <= keySched[p_rk][(4*i_rk)+k_rk+3];
418     tempSubFour[p_rk][0] <= keySched[p_rk][(4*i_rk)+k_rk];
419     step <= 12;
420 endrule
421
422 rule rest_sched7 (step == 12);
423     keySched[p_rk][(4*i_rk)+4+k_rk] <= tempSubOne[p_rk][0] ^ tempSubFour[p_rk][0];
424     step <= 13;
425 endrule
426
427 rule rest_sched8 (step == 13);
428     if (p_rk < 3)begin
429         p_rk <= p_rk + 1;
430         step <= 11;
431     end
432     else begin
433         p_rk <= 0;
434         if (k_rk < 3)begin
435             k_rk <= k_rk + 1;
436             step <= 11;
437         end
438         else begin
439             k_rk <= 1;
440             if (i_rk < 9)begin
441                 i_rk <= i_rk + 1;
442                 step <= 2;
443             end
444             else begin
445                 step <= 14; //test 14
446             end
447         end
448     end
449 endrule
450
451
452 //----- Decide to do Encryption/Decipher -----//
453 //
454 //-----
455 rule fetch_signal (step == 14);
456     sig <= sigtemp;
457     step <= 400;
458 endrule
459 rule fetch_signal1 (step == 400);
460     if (sig == 1)begin
461         step <= 401;
462     end
463     else begin
464         step <= 39;
465     end
466 endrule
467
468 //----- Start Encryption -----//
469 //
470 //-----
471
472 //----- Initialize -> Add round key -----//
473 rule add_round_key (step == 401);
474     for (Integer im = 0; im < 4; im = im+1)
475         for (Integer jm = 0; jm < 4; jm = jm+1)
476             state[jm][im] <= keySched[jm][im] ^ state[jm][im];
477     step <= 15;
478 endrule
479
480 //----- Start Round Encryption -----//
481 //
482 //-----
483
484 //----- Step 1 -> ByteSubstitution -----//

```

```

485 // a change from 0 - 9
486 // 1. Fetch each element
487 rule aes_byte_sub (step == 15);
488     tSub <= state[c][b];
489     step <= 16;
490 endrule
491
492 // 2. Fetch X & Y
493 rule aes_byte_sub2 (step == 16);
494     tSub1 <= tSub[7:4];
495     tSub2 <= tSub[3:0];
496     step <= 17;
497 endrule
498
499 // 3. Do substitution
500 rule aes_byte_sub3 (step == 17);
501     state[c][b] <= sbox[tSub1][tSub2];
502     if (c < 3)begin
503         c <= c+1;
504         step <= 15;
505     end
506     else begin
507         c <= 0;
508         if (b < 3)begin
509             b <= b+1;
510             step <= 15;
511         end
512         else begin
513             b <= 0;
514             step <= 18;
515         end
516     end
517 endrule
518
519 //----- Step 2 -> ShiftRows -----//
520 // 1. Second row shift
521 rule aes_sr (step == 18);
522     tempshift1 <= state[1][0];
523     step <= 19;
524 endrule
525
526 rule aes_sr1 (step == 19);
527     for (Integer i = 1; i < 4; i = i+1)
528         state[1][i-1] <= state[1][i];
529     step <= 20;
530 endrule
531
532 rule aes_sr2 (step == 20);
533     state[1][3] <= tempshift1;
534     step <= 21;
535 endrule
536
537 // 2. Third row shift
538 rule aes_sr3 (step == 21);
539     tempshift2 <= state[2][0];
540     tempshift3 <= state[2][1];
541     step <= 22;
542 endrule
543
544 rule aes_sr4 (step == 22);
545     for (Integer i = 2; i < 4; i = i+1)
546         state[2][i-2] <= state[2][i];
547     step <= 23;
548 endrule
549
550 rule aes_sr5 (step == 23);
551     state[2][2] <= tempshift2;
552     state[2][3] <= tempshift3;
553     step <= 24;
554 endrule
555
556 // 3. Fourth row shift
557 rule aes_sr6 (step == 24);
558     tempshift4 <= state[3][3];
559     tempshift5 <= state[3][0];
560     tempshift6 <= state[3][1];
561     tempshift7 <= state[3][2];
562     step <= 25;
563 endrule
564
565 rule aes_sr7 (step == 25);
566     state[3][0] <= tempshift4;
567     state[3][1] <= tempshift5;
568     state[3][2] <= tempshift6;
569     state[3][3] <= tempshift7;
570     step <= 26;
571 endrule
572
573 //----- Step 3 -> Mixcolumn -----//
574 // 1. Judge the iteration time, if equal 9(10 time iteration) then jump this step
575 rule mixcolumn (step == 26);
576     if (a < 9)begin
577         step <= 27;
578     end
579     else begin
580         step <= 36;
581     end

```



```

582     endrule
583
584     // 2. Fetch element from the intermediate matrix
585     // id change from 0 - 3
586     // ie change from 0 - 3
587     // ff change from 0 - 3
588     rule mixcolumn1 (step == 27);
589         tempfile <= state[e][d];
590         tempmix <= mix[f][e];
591         step <= 28; //check
592     endrule
593
594     // 3. Do the Finite field arithmetic(also known as GF2^8)
595     rule mixcolumn2 (step == 28);
596         if (tempfile < 8'h80)begin //p1
597             if (tempmix == 8'h01)begin //p2
598                 step <= 34;
599             end
600             else begin //p3
601                 if (tempmix == 8'h02)begin //p4
602                     tempfile <= tempfile << 1;
603                     step <= 34;
604                 end
605                 else begin //p5
606                     tempfile1 <= tempfile << 1;
607                     step <= 29;
608                 end
609             end
610         end
611         else begin //p6
612             if (tempmix == 8'h01)begin //p7
613                 step <= 34;
614             end
615             else begin //p8
616                 if (tempmix == 8'h02)begin //p9
617                     tempfile <= tempfile << 1;
618                     step <= 30;
619                 end
620                 else begin //p10
621                     tempfile1 <= tempfile;
622                     step <= 31;
623                 end
624             end
625         end
626     endrule
627
628     rule mixcolumn_p5 (step == 29);
629         tempfile <= tempfile ^ tempfile1;
630         step <= 34;
631     endrule
632
633     rule mixcolumn_p9 (step == 30);
634         tempfile <= tempfile ^ 8'h1b;
635         step <= 34;
636     endrule
637
638     rule mixcolumn_p10 (step == 31);
639         tempfile <= tempfile << 1;
640         step <= 32;
641     endrule
642
643     rule mixcolumn_p10_1 (step == 32);
644         tempfile <= tempfile ^ 8'h1b;
645         step <= 33;
646     endrule
647
648     rule mixcolumn_p10_2 (step == 33);
649         tempfile <= tempfile ^ tempfile1;
650         step <= 34;
651     endrule
652
653     rule mixcolumn3 (step == 34);
654         tempstate[f][e] <= tempfile;
655         if (f < 3)begin
656             f <= f+1;
657             step <= 27; //27 check
658         end
659         else begin
660             f <= 0;
661             if (e < 3)begin
662                 e <= e+1;
663                 step <= 27; //27 above
664             end
665             else begin
666                 e <= 0;
667                 step <= 35; //35 above
668             end
669         end
670     endrule
671
672     // 4. Do the matrix multiplication
673     rule mixcolumn4 (step == 35);
674         for (Integer i = 0; i<4; i = i+1)
675             state[i][d] <= tempstate[i][0] ^ tempstate[i][1] ^ tempstate[i][2] ^ tempstate[i][3];
676         if (d < 3)begin
677             d <= d+1;
678             step <= 27;

```

```

679         end
680     else begin
681         d <= 0;
682         step <= 36;
683     end
684
685 endrule
686
687 //----- Step 4 -> Add round key -----//
688 rule last_step (step == 36);
689     state[h][g] <= keySched[h][4*a+4+g] ^ state[h][g];
690     if (h < 3)begin
691         h <= h+1;
692         step <= 36;
693     end
694     else begin
695         h <= 0;
696         if (g < 3)begin
697             g <= g+1;
698             step <= 36;
699         end
700         else begin
701             g <= 0;
702             step <= 37;
703         end
704     end
705 endrule
706
707 rule last_step2 (step == 37);
708     if (a < 9)begin
709         a <= a+1;
710         step <= 15;
711     end
712     else begin
713         //===== !Check point: wait for interface output! =====//
714         cp <= 1;
715     end
716 endrule
717
718 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
719 //                                Start Decipher
720 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
721
722 //----- Put data into register -----//
723 rule init_decipher (step == 39);
724     for(Integer i = 0; i<16; i = i+1)
725         for(Integer j = 0; j<16; j = j+1)
726             invsbox[j][i] <= inv_sbox[j][i];
727
728         for(Integer i = 0; i<4; i = i+1)
729             for(Integer j = 0; j<4; j = j+1)
730                 invmix[j][i] <= inv_mix[j][i];
731         step <= 40;
732 endrule
733
734 //----- Initialize -> Add round key -----//
735 rule init_add_round_key (step == 40);
736     for (Integer im = 0; im < 4; im = im+1)
737         for (Integer jm = 0; jm < 4; jm = jm+1)
738             resultt[jm][im] <= keySched[jm][im+40] ^ resultt[jm][im];
739     step <= 41;
740 endrule
741
742 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
743 //                                Start Round Decipher
744 ///////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
745
746 //----- Step 1 -> Inverse shift rows -----//
747 // k change from 0 - 9
748
749 // 1. Second row shift
750 rule inv_sr (step == 41);
751     tshift1 <= resultt[1][3];
752     tshift2 <= resultt[1][0];
753     tshift3 <= resultt[1][1];
754     tshift4 <= resultt[1][2];
755     step <= 42;
756 endrule
757
758 rule inv_sr1 (step == 42);
759     resultt[1][0] <= tshift1;
760     resultt[1][1] <= tshift2;
761     resultt[1][2] <= tshift3;
762     resultt[1][3] <= tshift4;
763     step <= 43;
764 endrule
765
766 // 2. Third row shift
767 rule inv_sr2 (step == 43);
768     tshift11 <= resultt[2][2];
769     tshift22 <= resultt[2][3];
770     tshift33 <= resultt[2][0];
771     tshift44 <= resultt[2][1];
772     step <= 44;
773 endrule
774
775 rule inv_sr3 (step == 44);

```

```

776         resultt[2][0] <= tshift11;
777         resultt[2][1] <= tshift22;
778         resultt[2][2] <= tshift33;
779         resultt[2][3] <= tshift44;
780         step <= 45;
781     endrule
782
783     // 3. Fourth row shift
784     rule inv_sr4 (step == 45);
785         tshift5 <= resultt[3][1];
786         tshift6 <= resultt[3][2];
787         tshift7 <= resultt[3][3];
788         tshift8 <= resultt[3][0];
789         step <= 46;
790     endrule
791
792     rule inv_sr5 (step == 46);
793         resultt[3][0] <= tshift5;
794         resultt[3][1] <= tshift6;
795         resultt[3][2] <= tshift7;
796         resultt[3][3] <= tshift8;
797         step <= 47;
798     endrule
799
800     //----- Step 2-> Inverse byte substitution -----//
801     // bz change from 0-3
802     // cz change from 0-3
803     rule inv_sb (step == 47);
804         tSb <= resultt[cz][bz];
805         step <= 48;
806     endrule
807
808     rule inv_sb1 (step == 48);
809         tSb1 <= tSb[7:4];
810         tSb2 <= tSb[3:0];
811         step <= 49;
812     endrule
813
814     rule inv_sb2 (step == 49);
815         resultt[cz][bz] <= invsbox[tSb1][tSb2];
816         if (cz < 3)begin
817             cz <= cz+1;
818             step <= 47;
819         end
820         else begin
821             cz <= 0;
822             if (bz < 3)begin
823                 bz <= bz+1;
824                 step <= 47;
825             end
826             else begin
827                 bz <= 0;
828                 step <= 50;
829             end
830         end
831     endrule
832
833     //----- Step 3 -> Inverse add round key -----//
834     // hz change from 0 - 3
835     // gz change from 0 - 3
836     rule inv_ark (step == 50);
837         resultt[hz][gz] <= keySched[hz][gz+36-4*k] ^ resultt[hz][gz];
838         if (hz < 3)begin
839             hz <= hz+1;
840             step <= 50;
841         end
842         else begin
843             hz <= 0;
844             if (gz < 3)begin
845                 gz <= gz+1;
846                 step <= 50;
847             end
848             else begin
849                 gz <= 0;
850                 step <= 51;
851             end
852         end
853     endrule
854
855     //----- Step 4 -> Inverse mixcolumn -----//
856     rule inv_mixcolumn (step == 51);
857         invtempmix <= 8'h00;
858         invtempfile <= 8'h00;
859         invtempfileOri <= 8'h00; //Store Original Element
860         invtempfileFir <= 8'h00; //Store First iteration of Finite Field Arithmetic Result
861         invtempfileSec <= 8'h00; //Store Second Iteration Of Finite Field Arithmetic Result
862         invtempfileThr <= 8'h00; //Store Third Iteration Of Finite Field Arithmetic Result
863
864         // Judge the Decipher iteration time, if equal to 9 then jump this step
865         if (k < 9)begin
866             step <= 52;
867         end
868         else begin
869             step <= 81;
870         end
871     endrule
872

```

```

873 // 1. Do the Finite Field Arithmetic
874 // id change from 0 - 3
875 // ie change from 0 - 3
876 // ff change from 0 - 3
877 rule inv_mixcolumn1 (step == 52);
878     test[ie] <= id;
879     invtempfile <= resultt[ie][id];
880     invtempmix <= invmix[ff][ie];
881     step <= 53;
882 endrule
883
884 rule inv_mixc (step == 53);
885     invtempfileOri <= invtempfile;
886     if (invtempfile < 8'h80)begin //p1
887         invtempfile <= invtempfile << 1;
888         step <= 54;
889     end
890     else begin //p8
891         invtempfile <= invtempfile << 1;
892         step <= 55;
893     end
894 endrule
895
896 rule inv_p1 (step == 54); // from p1
897     invtempfileFir <= invtempfile;
898     if (invtempfile < 8'h80)begin //p2
899         invtempfile <= invtempfile << 1;
900         step <= 56;
901     end
902     else begin //p5
903         invtempfile <= invtempfile << 1;
904         step <= 57;
905     end
906 endrule
907
908 rule inv_p2 (step == 56); // from p2
909     invtempfileSec <= invtempfile;
910     if (invtempfile < 8'h80)begin //p3
911         invtempfile <= invtempfile << 1;
912         step <= 58;
913     end
914     else begin //p4
915         invtempfile <= invtempfile << 1;
916         step <= 59;
917     end
918 endrule
919
920 rule inv_p3 (step == 58); // jump from p3
921     invtempfileThr <= invtempfile;
922     step <= 77; // jump to judge
923 endrule
924
925 rule inv_p4 (step == 59); // jump from p4
926     invtempfile <= invtempfile ^ 8'h1b;
927     step <= 60;
928 endrule
929 rule inv_p4_1 (step == 60);
930     invtempfileThr <= invtempfile;
931     step <= 77; // jump to judge
932 endrule
933
934 rule inv_p5 (step == 57); // jump from p5
935     invtempfile <= invtempfile ^ 8'h1b;
936     step <= 61;
937 endrule
938 rule inv_p5_1 (step == 61);
939     invtempfileSec <= invtempfile;
940     if (invtempfile < 8'h80)begin //p6
941         invtempfile <= invtempfile << 1;
942         step <= 62;
943     end
944     else begin //p7
945         invtempfile <= invtempfile << 1;
946         step <= 63;
947     end
948 endrule
949
950 rule inv_p6 (step == 62); // jump from p6
951     invtempfileThr <= invtempfile;
952     step <= 77; // jump to judge
953 endrule
954
955 rule inv_p7 (step == 63); // jump from p7
956     invtempfile <= invtempfile ^ 8'h1b;
957     step <= 64;
958 endrule
959 rule inv_p7_1 (step == 64);
960     invtempfileThr <= invtempfile;
961     step <= 77; // jump to judge
962 endrule
963
964 rule inv_p8 (step == 55); // jump from p8
965     invtempfile <= invtempfile ^ 8'h1b;
966     step <= 65;
967 endrule
968 rule inv_p8_1 (step == 65);
969     invtempfileFir <= invtempfile;

```

```

970         if (invtempfile < 8'h80)begin                                     // p9
971             invtempfile <= invtempfile << 1;
972             step <= 66;
973         end
974         else begin                                                         // p12
975             invtempfile <= invtempfile << 1;
976             step <= 67;
977         end
978     endrule
979
980     rule inv_p9 (step == 66);                                              // jump from p9
981         invtempfileSec <= invtempfile;
982         if (invtempfile < 8'h80)begin                                     // p10
983             invtempfile <= invtempfile << 1;
984             step <= 68;
985         end
986         else begin                                                         // p11
987             invtempfile <= invtempfile << 1;
988             step <= 69;
989         end
990     endrule
991
992     rule inv_10 (step == 68);                                              // jump from p10
993         step <= 70;
994     endrule
995     rule inv_10_1 (step == 70);
996         invtempfileThr <= invtempfile;
997         step <= 77;                                                         // jump to judge
998     endrule
999
1000    rule inv_11 (step == 69);                                              // jump from p11
1001        step <= 71;
1002    endrule
1003    rule inv_11_1 (step == 71);
1004        invtempfile <= invtempfile ^ 8'h1b;
1005        step <= 72;
1006    endrule
1007    rule inv_11_2 (step == 72);
1008        invtempfileThr <= invtempfile;
1009        step <= 77;                                                         // jump to judge
1010    endrule
1011
1012    rule inv_12 (step == 67);                                              // jump from p12
1013        invtempfile <= invtempfile ^ 8'h1b;
1014        step <= 73;
1015    endrule
1016    rule inv_12_1 (step == 73);
1017        invtempfileSec <= invtempfile;
1018        if (invtempfile < 8'h80)begin                                     //p13
1019            invtempfile <= invtempfile << 1;
1020            step <= 74;
1021        end
1022        else begin                                                         //p14
1023            invtempfile <= invtempfile << 1;
1024            step <= 75;
1025        end
1026    endrule
1027
1028    rule inv_p13 (step == 74);                                              // jump from p13
1029        invtempfileThr <= invtempfile;
1030        step <= 77;                                                         // jump to judge
1031    endrule
1032
1033    rule inv_p14 (step == 75);                                              // jump from p14
1034        invtempfile <= invtempfile ^ 8'h1b;
1035        step <= 76;
1036    endrule
1037    rule inv_p14_1 (step == 76);
1038        invtempfileThr <= invtempfile;
1039        step <= 77;                                                         // jump to judge
1040    endrule
1041
1042    // 1.1 Complete the Finite Field Arithmetic
1043    rule judge (step == 77);
1044        if (invtempmix == 8'h09)begin //Judge 0x09
1045            invtempfile <= invtempfileThr ^ invtempfileOri;
1046        end
1047        if (invtempmix == 8'h0b)begin
1048            invtempfile <= invtempfileThr ^ invtempfileFir ^ invtempfileOri;
1049        end
1050        if (invtempmix == 8'h0d)begin
1051            invtempfile <= invtempfileThr ^ invtempfileSec ^ invtempfileOri;
1052        end
1053        if (invtempmix == 8'h0e)begin
1054            invtempfile <= invtempfileThr ^ invtempfileSec ^ invtempfileFir;
1055        end
1056        step <= 78;
1057    endrule
1058
1059    rule judge1 (step == 78);
1060        invtempstate[ff][ie] <= invtempfile;
1061        step <= 79;
1062    endrule
1063
1064    rule judge2 (step == 79);
1065        if (ff < 3)begin
1066            ff <= ff+1;

```

```

1067         step <= 52;
1068     end
1069     else begin
1070         ff <= 0;
1071         if (ie < 3)begin
1072             ie <= ie+1;
1073             step <= 52;
1074         end
1075         else begin
1076             ie <= 0;
1077             if (id < 4)begin
1078                 // id <= id+1;
1079                 step <= 80;
1080             end
1081             else begin
1082                 id <= 0;
1083                 step <= 81;
1084             end
1085         end
1086     end
1087 endrule
1088
1089 // 2. Do the matrix multiplication
1090 rule judge3 (step == 80);
1091     for (Integer i4 = 0; i4 < 4; i4 = i4+1)begin
1092         resultt[i4][id] <= invtempstate[i4][0] ^ invtempstate[i4][1] ^ invtempstate[i4][2] ^ invtempstate[i4][3];
1093     end
1094     id <= id+1;
1095     step <= 52;
1096 endrule
1097
1098 rule loop_end (step == 81);
1099     if (k < 9)begin
1100         k <= k+1;
1101         step <= 41;
1102     end
1103     else begin
1104         //===== !Check point: wait for interface output! =====//
1105         cp <= 1;
1106     end
1107 endrule
1108
1109 ////////////////////////////////////////
1110 // Interface Method
1111 ////////////////////////////////////////
1112
1113 //----- Method for loading input -----//
1114 // Transfer 32bit per clock cycle
1115 method Action loadInput(Bit#(32) dIn, int addr);
1116     //statetemp[0][0] <= dIn;
1117     temp <= dIn;
1118
1119     if(addr == 0)begin
1120         statetemp[0][0] <= dIn[31:24];
1121         statetemp[1][0] <= dIn[23:16];
1122         statetemp[2][0] <= dIn[15:8];
1123         statetemp[3][0] <= dIn[7:0];
1124     end
1125
1126     if(addr == 1)begin
1127         statetemp[0][1] <= dIn[31:24];
1128         statetemp[1][1] <= dIn[23:16];
1129         statetemp[2][1] <= dIn[15:8];
1130         statetemp[3][1] <= dIn[7:0];
1131     end
1132
1133     if(addr == 2)begin
1134         statetemp[0][2] <= dIn[31:24];
1135         statetemp[1][2] <= dIn[23:16];
1136         statetemp[2][2] <= dIn[15:8];
1137         statetemp[3][2] <= dIn[7:0];
1138     end
1139
1140     if(addr == 3)begin
1141         statetemp[0][3] <= dIn[31:24];
1142         statetemp[1][3] <= dIn[23:16];
1143         statetemp[2][3] <= dIn[15:8];
1144         statetemp[3][3] <= dIn[7:0];
1145     end
1146 endmethod
1147
1148 //----- Method for import cipher key -----//
1149 method Action importKey(Bit#(32) imKey, int addr);
1150     if(addr == 0)begin
1151         keymtemp[0][0] <= imKey[31:24];
1152         keymtemp[1][0] <= imKey[23:16];
1153         keymtemp[2][0] <= imKey[15:8];
1154         keymtemp[3][0] <= imKey[7:0];
1155     end
1156
1157     if(addr == 1)begin
1158         keymtemp[0][1] <= imKey[31:24];
1159         keymtemp[1][1] <= imKey[23:16];
1160         keymtemp[2][1] <= imKey[15:8];
1161         keymtemp[3][1] <= imKey[7:0];
1162     end
1163
1164     if(addr == 2)begin
1165         keymtemp[0][2] <= imKey[31:24];
1166         keymtemp[1][2] <= imKey[23:16];
1167         keymtemp[2][2] <= imKey[15:8];

```

```

1164         keymtemp[3][2] <= imKey[7:0];
1165     end
1166     if(addr == 3)begin
1167         keymtemp[0][3] <= imKey[31:24];
1168         keymtemp[1][3] <= imKey[23:16];
1169         keymtemp[2][3] <= imKey[15:8];
1170         keymtemp[3][3] <= imKey[7:0];
1171     end
1172 endmethod
1173
1174 //----- Method for load Encrypted text -----//
1175 method Action loadDestate(Bit#(32) des, int addr);
1176     if(addr == 0)begin
1177         resultttemp[0][0] <= des[31:24];
1178         resultttemp[1][0] <= des[23:16];
1179         resultttemp[2][0] <= des[15:8];
1180         resultttemp[3][0] <= des[7:0];
1181     end
1182     if(addr == 1)begin
1183         resultttemp[0][1] <= des[31:24];
1184         resultttemp[1][1] <= des[23:16];
1185         resultttemp[2][1] <= des[15:8];
1186         resultttemp[3][1] <= des[7:0];
1187     end
1188     if(addr == 2)begin
1189         resultttemp[0][2] <= des[31:24];
1190         resultttemp[1][2] <= des[23:16];
1191         resultttemp[2][2] <= des[15:8];
1192         resultttemp[3][2] <= des[7:0];
1193     end
1194     if(addr == 3)begin
1195         resultttemp[0][3] <= des[31:24];
1196         resultttemp[1][3] <= des[23:16];
1197         resultttemp[2][3] <= des[15:8];
1198         resultttemp[3][3] <= des[7:0];
1199     end
1200 endmethod
1201
1202 //----- Import the Encryption/Decipher signal -----//
1203 method Action check_signal(int signal) if (step == 700);
1204     sigtemp <= signal;
1205     step <= 0; //debug change 0 ori
1206 endmethod
1207
1208 //----- Output Encryption/Decipher DONE signal -----//
1209 method int checkpoint ();
1210     return cp;
1211 endmethod
1212 method int output_signal();
1213     return sig;
1214 endmethod
1215
1216 //----- Output the Encryption result -----//
1217 method Bit#(8) output_state (int i, int j);
1218     Bit#(8) out_k;
1219     out_k = state[i][j];
1220     return out_k;
1221 endmethod
1222
1223 //----- Output the Decipher result -----//
1224 method Bit#(8) output_result (int i, int j);
1225     Bit#(8) out_r;
1226     out_r = resultt[i][j];
1227     return out_r;
1228 endmethod
1229
1230
1231 endmodule: mkTb
1232
1233 endpackage: AESchange

```

Appendix - AES test bench

```
1 package AESTb;
2   import AESchange::*;
3   (*synthesize*)
4   module tb_ip_text (Empty);
5       Ifc_main_type dataInput <- mkTb;
6
7       Reg#(int) flag <- mkReg(0);
8       Reg#(Bit#(8)) the_output[4][4];
9       Reg#(int) ii <- mkReg(0);
10      Reg#(int) jj <- mkReg(0);
11
12      for (Integer i=0;i<4;i=i+1)
13          for(Integer j=0;j<4;j=j+1)
14              the_output[j][i] <- mkRegU;
15
16      //----- Input plain text -----//
17      rule input_data (flag == 0);
18          dataInput.loadInput('h3243f6a8, 0);
19          flag <= 1;
20      endrule
21      rule input_data1 (flag == 1);
22          dataInput.loadInput('h885a308d, 1);
23          flag <= 2;
24      endrule
25      rule input_data2 (flag == 2);
26          dataInput.loadInput('h313198a2, 2);
27          flag <= 3;
28      endrule
29      rule input_data3 (flag == 3);
30          dataInput.loadInput('he0370734, 3);
31          flag <= 4;
32      endrule
33
34      //----- Input cipher key -----//
35      rule input_key (flag == 4);
36          dataInput.importKey(32'h2b7e1516, 0);
37          flag <= 5;
38      endrule
39      rule input_key1 (flag == 5);
40          dataInput.importKey(32'h28aed2a6, 1);
41          flag <= 6;
42      endrule
43      rule input_key2 (flag == 6);
44          dataInput.importKey(32'habf71588, 2);
45          flag <= 7;
46      endrule
47      rule input_key3 (flag == 7);
48          dataInput.importKey(32'h09cf4f3c, 3);
49          flag <= 8;
50      endrule
51
52      //----- Input encrypted text -----//
53      rule input_destate (flag == 8);
54          dataInput.loadDestate(32'h3925841d, 0);
55          flag <= 9;
56      endrule
57      rule input_destate1 (flag == 9);
58          dataInput.loadDestate(32'h02dc09fb, 1);
59          flag <= 10;
60      endrule
61      rule input_destate2 (flag == 10);
62          dataInput.loadDestate(32'hdc118597, 2);
```



```

63         flag <= 11;
64     endrule
65     rule input_destate3 (flag == 11);
66         dataInput.loadDestate(32'h196a0b32, 3);
67         flag <= 12;
68     endrule
69
70     //----- Set the Encryption/Decipher signal -----//
71     rule sendsignal (flag == 12);
72         //1 means enc
73         //0 means deci
74         dataInput.check_signal(0);
75         flag <= 400;
76     endrule
77
78     //----- Get the result from the Module -----//
79     rule getoutput (flag == 400);
80         if (dataInput.checkpoint == 0)begin
81             flag <= 400;
82         end
83         else begin
84             if (dataInput.output_signal == 1)begin
85                 flag <= 13;
86             end
87             else begin
88                 flag <= 14;
89             end
90         end
91     endrule
92
93     rule getoutput1 (flag == 13);
94         the_output[jj][ii] <= dataInput.output_state(jj, ii);
95         if (jj < 3)begin
96             jj <= jj+1;
97             flag <= 13;
98         end
99         else begin
100             jj <= 0;
101             if (ii < 3)begin
102                 ii <= ii+1;
103                 flag <= 13;
104             end
105             else begin
106                 ii <= 0;
107                 flag <= 15;
108             end
109         end
110     endrule
111
112     rule getoutput2 (flag == 14);
113         the_output[jj][ii] <= dataInput.output_result(jj, ii);
114         if (jj < 3)begin
115             jj <= jj+1;
116             flag <= 14;
117         end
118         else begin
119             jj <= 0;
120             if (ii < 3)begin
121                 ii <= ii+1;
122                 flag <= 14;
123             end
124             else begin
125                 ii <= 0;

```

```

126             flag <= 15;
127         end
128     end
129 endrule
130
131 //----- Display the result -----//
132 rule displayresult (flag == 15);
133     for(Integer i = 0; i<4; i = i+1)
134         for(Integer j = 0; j<4; j = j+1)
135             $display ("result = %h", the_output[j][i]);
136             $finish (0);
137         $finish (0);
138     endrule
139
140 endmodule: tb_ip_text
141
142 endpackage: AEstb

```