ECE-GY 6463 Advanced Computer Hardware Design
Final Project Report
NYU Processor Design - Group A
Team - 12

Virinchi Roy Surabhi (vs1930)
Akhil Wadhwa(aw3509)
Siddharth Saptharishi Murali (ssm706)
Sravan Reddy Chintareddy (src572)

Video Link : https://youtu.be/TCuIPBlUtqk

Rubric - Final Report (Due 5:00 PM on Dec 16) 11 points

a. design block diagram: (0.5)

a1	Draw a block diagram of **your** processor (this should be at the level corresponding to your structural implementation of the processor, i.e., at port map component instantiation level).	0.25	
a2	Annotate the block diagram of your processor, specifying important information (e.g., bus widths, control signals…)	0.25	

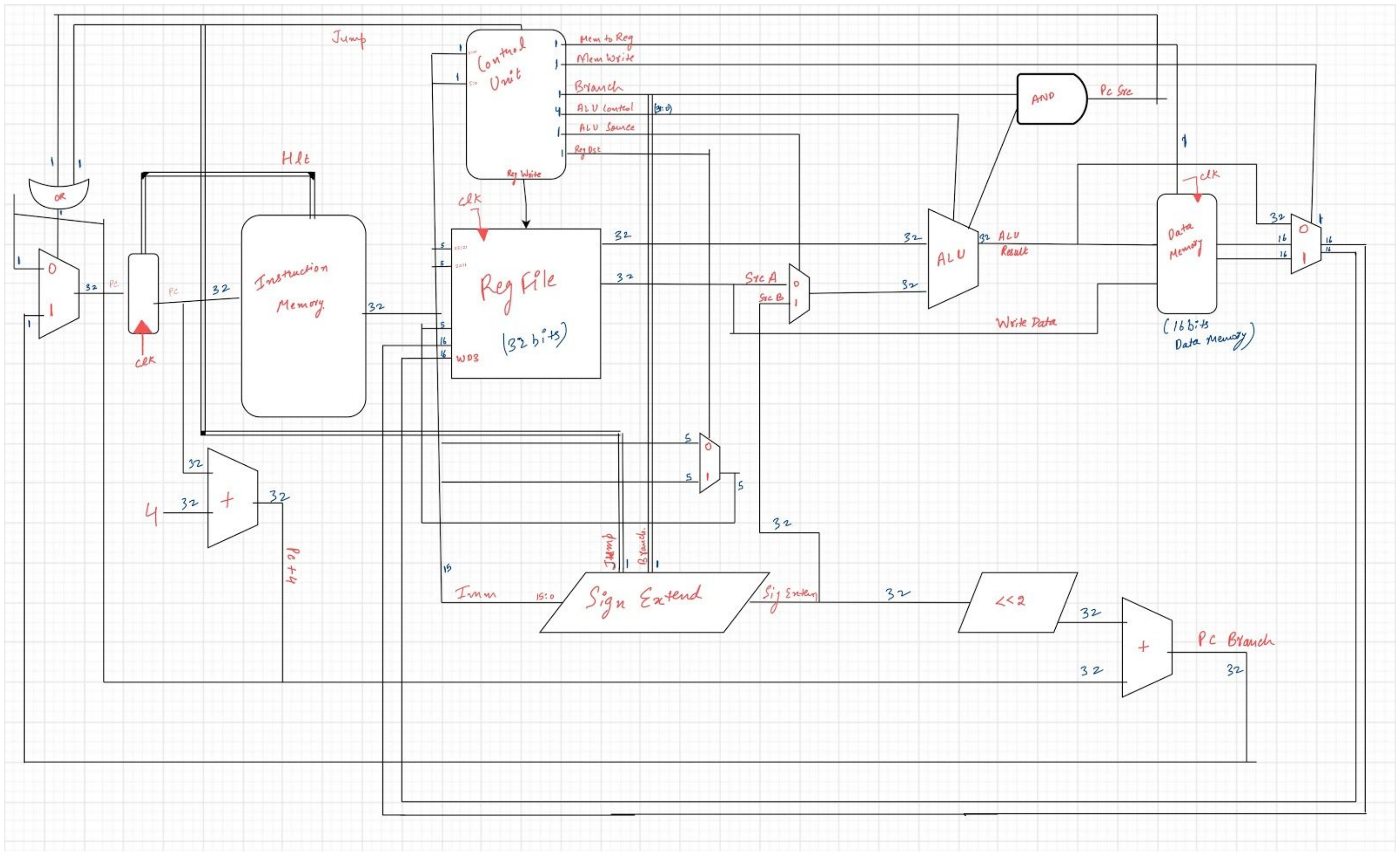


Figure: NYU Processor-6463

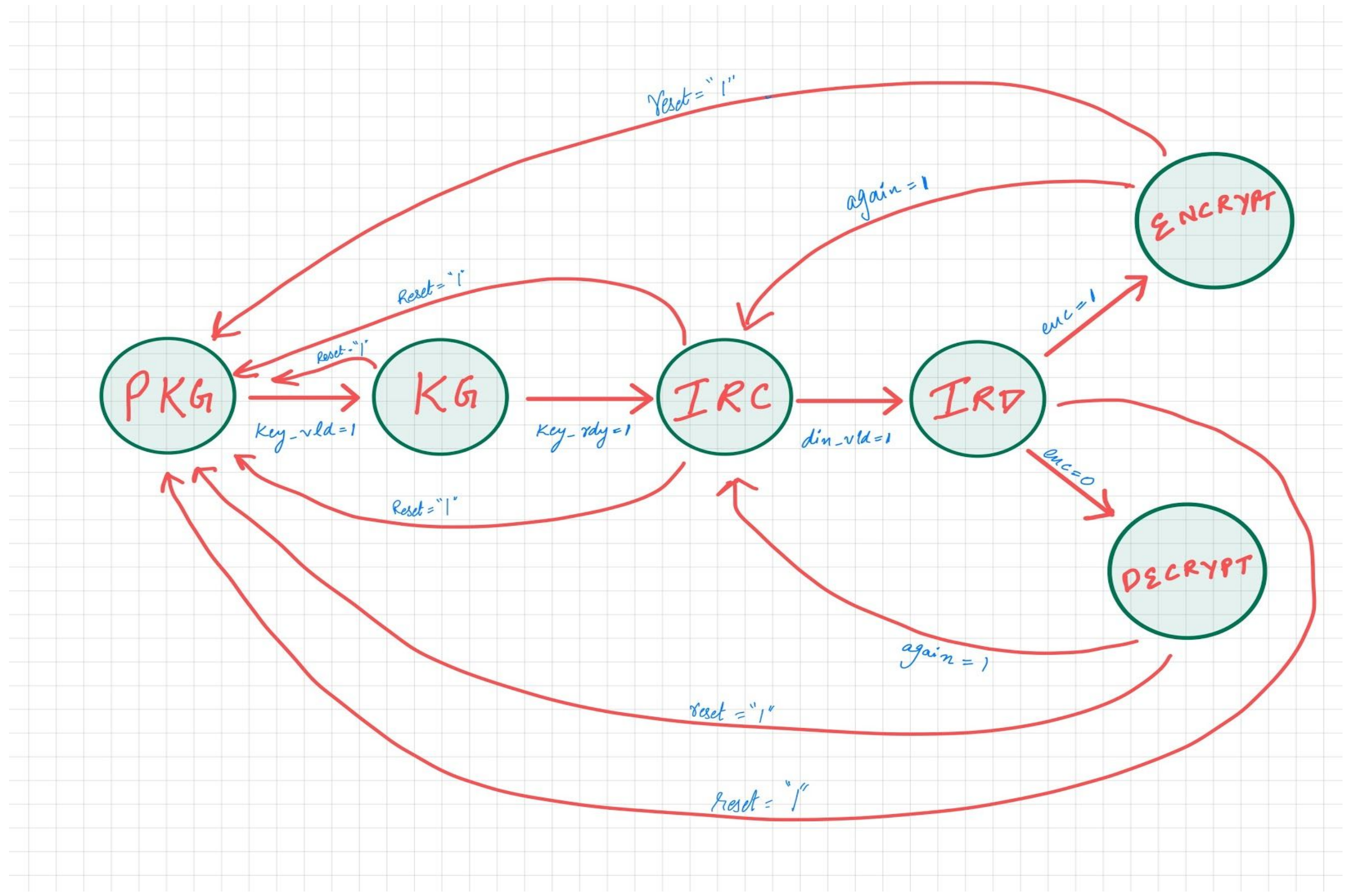


Figure : Finite State Machine

b. simulation screenshots:

b1	Functional simulation using testbench on 10 key/plaintext combinations for encryption.	0.05 x10	
	Annotations to show the input/outputs of interest	0.05 x10	

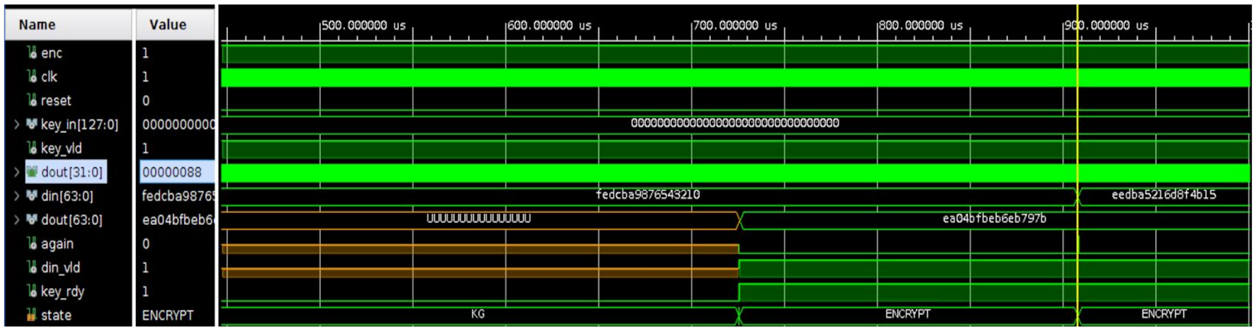
b2	Functional simulation using testbench on 10 key/ciphertext combinations for decryption	0.05 x10	
	Annotations to show the input/outputs of interest	0.05 x10	

b) Functional Simulation Using Testbench (Encryption and Decryption):

din => plaintext input,
clk => clock input,
reset => reset button for the process,
key_in => user input key of 128 bits,
key_vld => signal for key input is valid,
dout => Encryption/Decryption output,
again => signal for giving a new input without a new key
din_vld => signal for plaintext is valid,
key_rdy => signal for key generation is done,
state => shows the present state of the state machine.

b1) Encryption:

Test Case 1:
key_in = x“00000000000000000000000000000000”
din[63:0] = x“FEDCBA9876543210”
dout[63:0] = x“ea04bfbeb6eb797b”

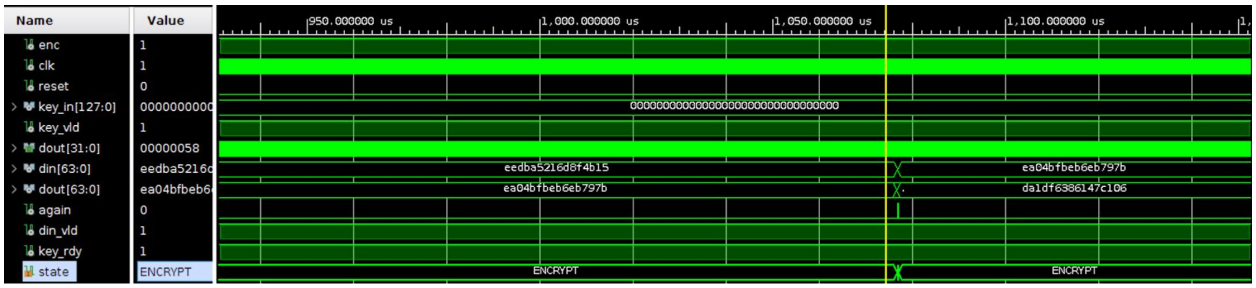


Test Case 2:

key_in = x“00000000000000000000000000000000”

din[63:0] = x“eedba5216d8f4b15”

dout[63:0] = x“da1df6386147c106”

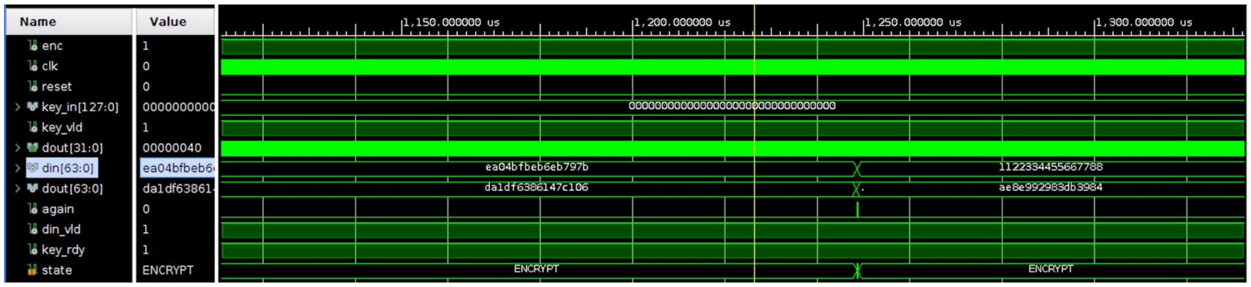


Test Case 3:

key_in = x“00000000000000000000000000000000”

din[63:0] = x“EA04BFBEB6EB797B”

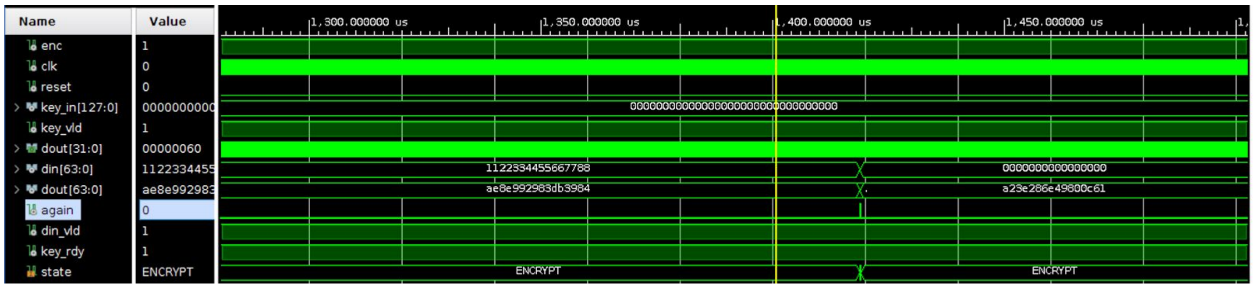
dout[63:0] = x“ae8e992983db3984”



Test Case 4:

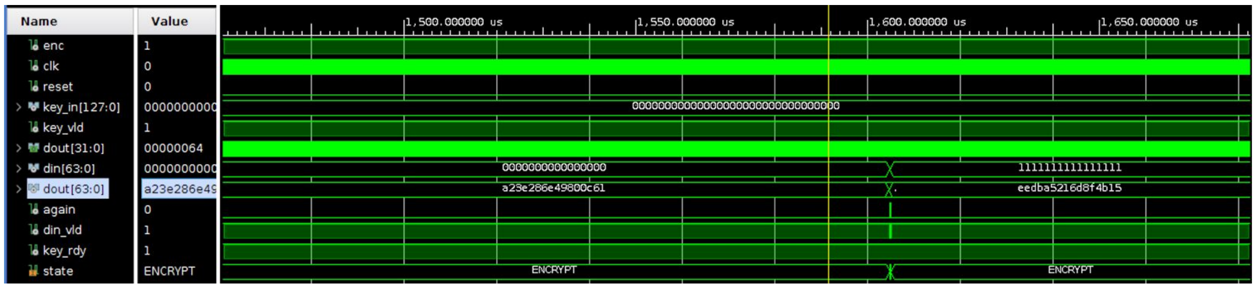
key_in = x“00000000000000000000000000000000”

din[63:0] = x“1122334455667788”
dout[63:0] = x“a23e286e49800c61”



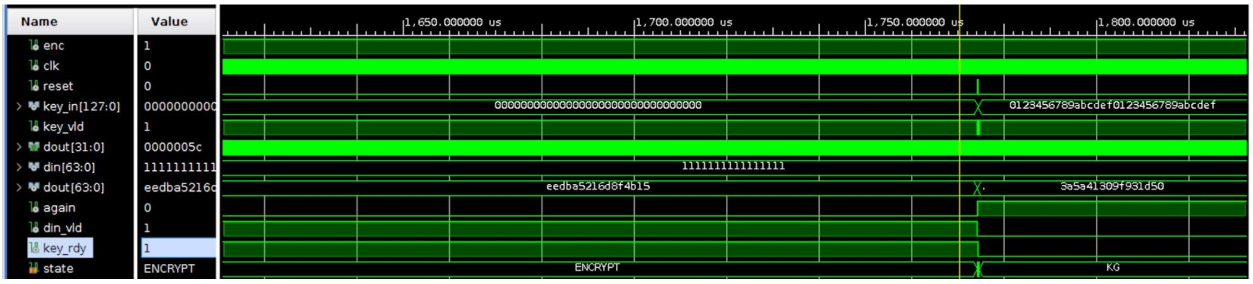
Test Case 5:

key_in = x“00000000000000000000000000000000”
din[63:0] = x“0000000000000000”
dout[63:0] = x“eedba5216d8f4b15”



Test Case 6:

key_in = x“00000000000000000000000000000000”
din[63:0] = x“1111111111111111”
dout[63:0] = x“a2b909f04ab63b0d”

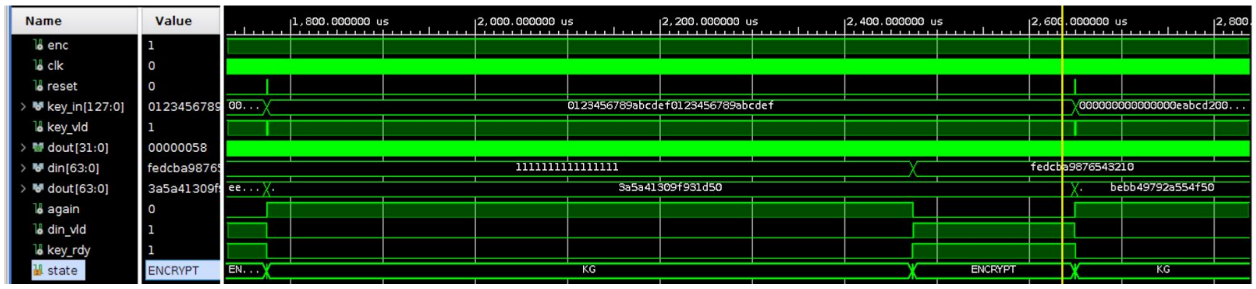


Test Case 7:

key_in = x“0123456789abcdef0123456789abcdef”

din[63:0] = x“FEDCBA9876543210”

dout[63:0] = x“bebb49792a554f50”

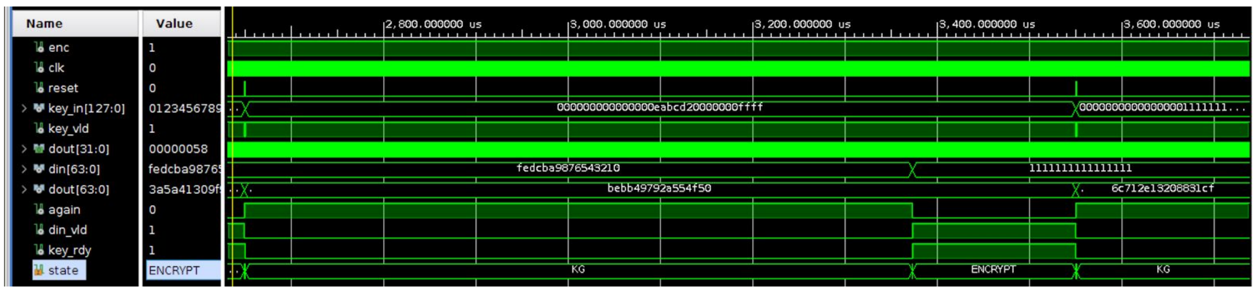


Test Case 8:

key_in = x“000000000000000eabcd20000000ffff”

din[63:0] = x“1111111111111111”

dout[63:0] = x“b5c62f43b5897c8e”

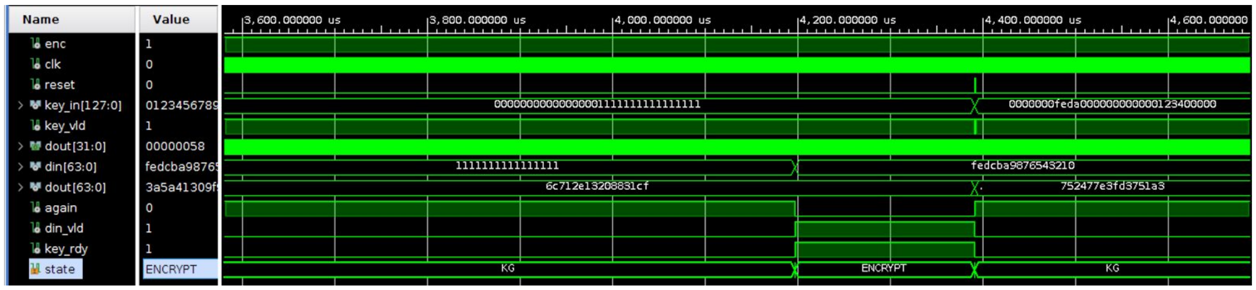


Test Case 9:

key_in = x“00000000000000001111111111111111”

din[63:0] = x“FEDCBA9876543210”

dout[63:0] = x“752477e3fd3751a3”

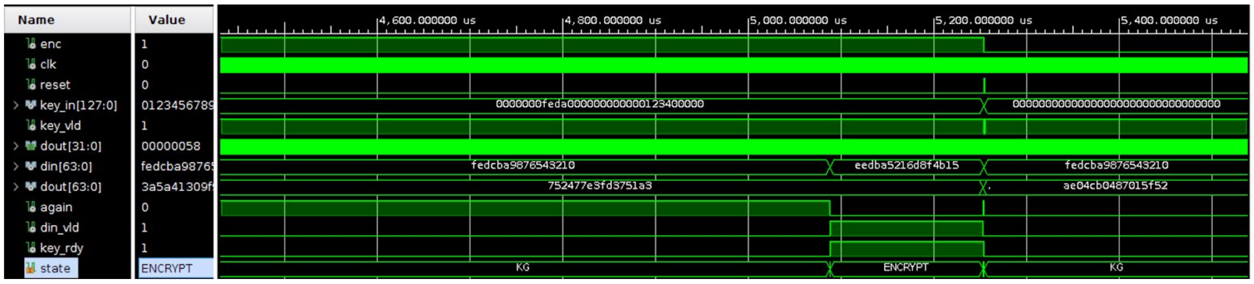


Test Case 10:

key_in = x“0000000feda000000000000123400000”

din[63:0] = x“eedba5216d8f4b15”

dout[63:0] = x“ae04cb0487015f52”



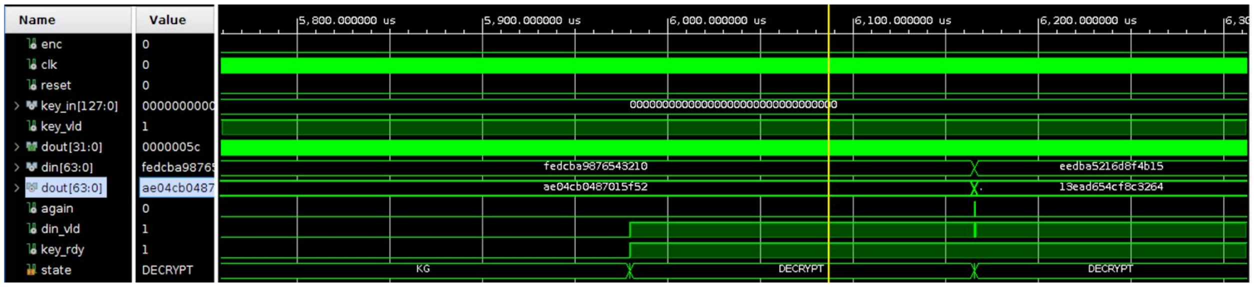
b2) Decryption:

Test Case 1:

key_in = x“00000000000000000000000000000000”

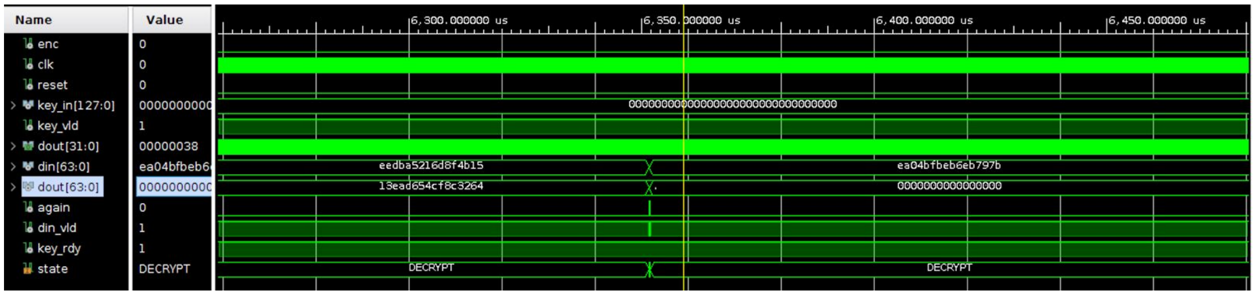
din[63:0] = x“FEDCBA9876543210”

dout[63:0] = x“ae04cb0487015f52”

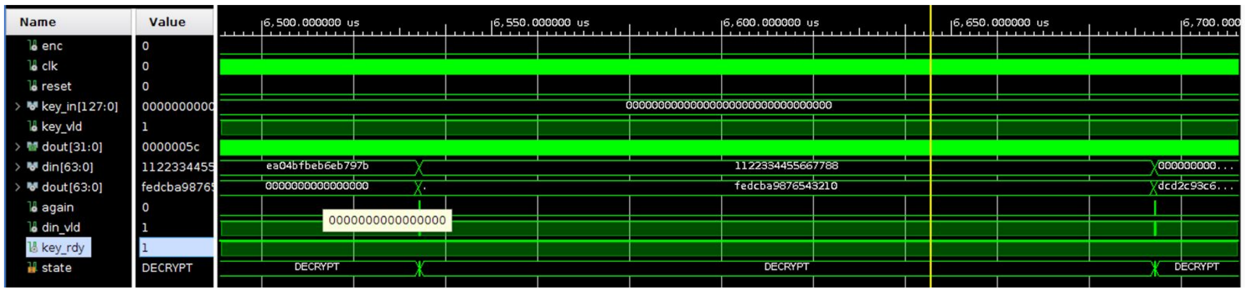


Test Case 2:

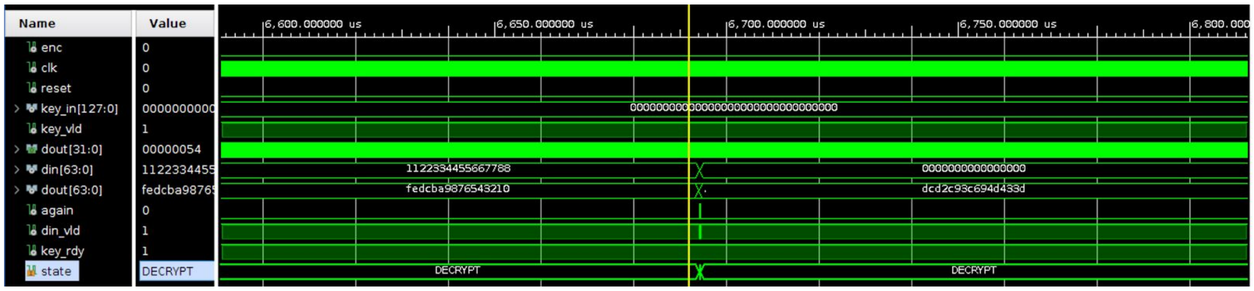
key_in = x“00000000000000000000000000000000”
din[63:0] = x“eedba5216d8f4b15”
dout[63:0] = x“0000000000000000”



Test Case 3:
key_in = x“00000000000000000000000000000000”
din[63:0] = x“EA04BFBEB6EB797B”
dout[63:0] = x“fedcba9876543210”



Test Case 4:
key_in = x“00000000000000000000000000000000”
din[63:0] = x“1122334455667788”
dout[63:0] = x“dcd2c93c694d433d”

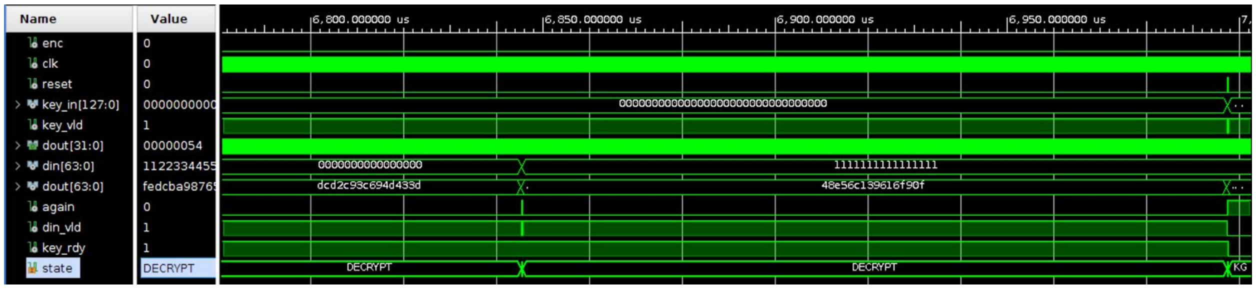


Test Case 5:

key_in = x“00000000000000000000000000000000”

din[63:0] = x“0000000000000000”

dout[63:0] = x“48e56c139616f90f”

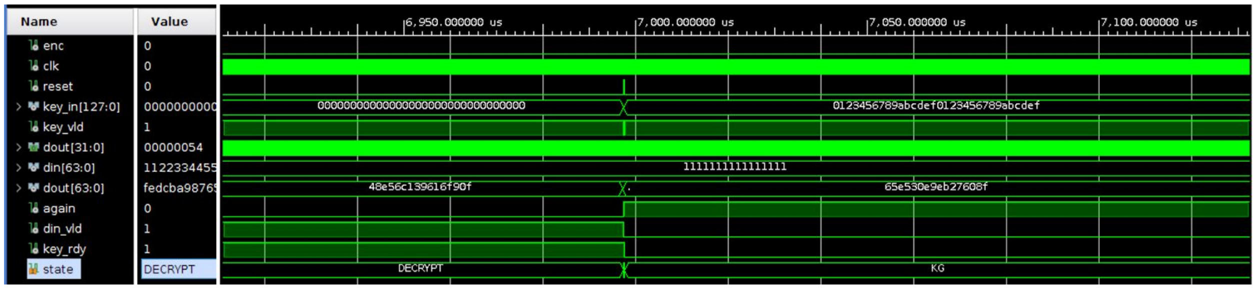


Test Case 6:

key_in = x“00000000000000000000000000000000”

din[63:0] = x“1111111111111111”

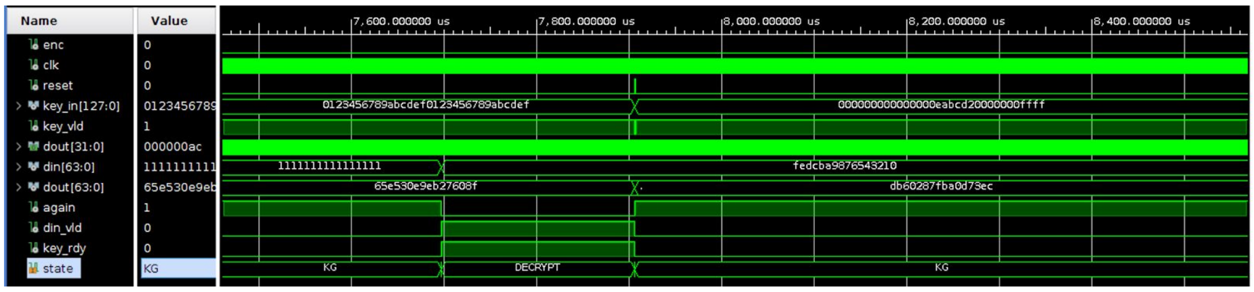
dout[63:0] = x“21b4bebfec6edda7”



Test Case 7:

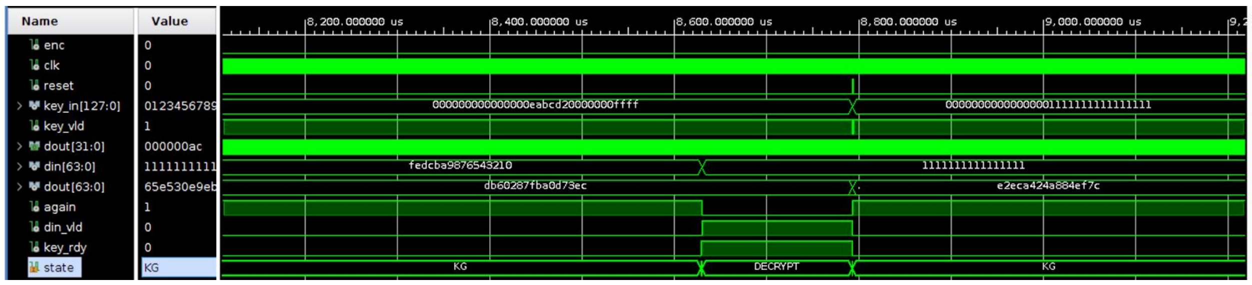
key_in = x“0123456789abcdef0123456789abcdef”

din[63:0] = x“FEDCBA9876543210”
dout[63:0] = x“db60287fba0d73ec”



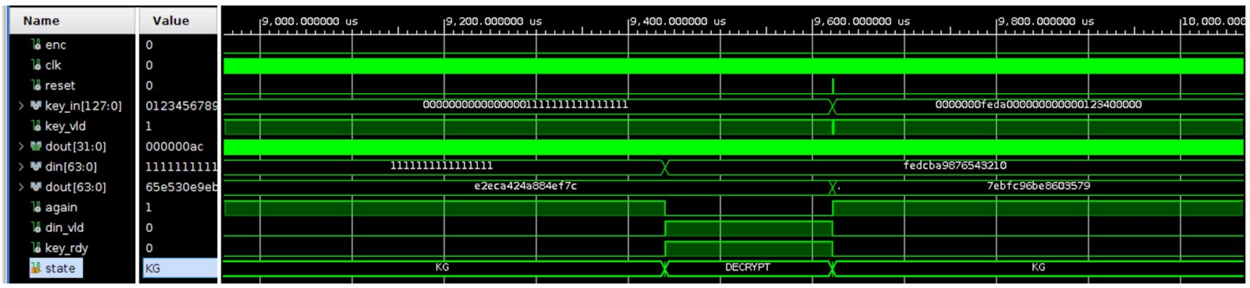
Test Case 8:

key_in = x“000000000000000eabcd20000000ffff”
din[63:0] = x“1111111111111111”
dout[63:0] = x“0e32239d147cb521”



Test Case 9:

key_in = x“00000000000000001111111111111111”
din[63:0] = x“FEDCBA9876543210”
dout[63:0] = x“7ebfc96be8603579”

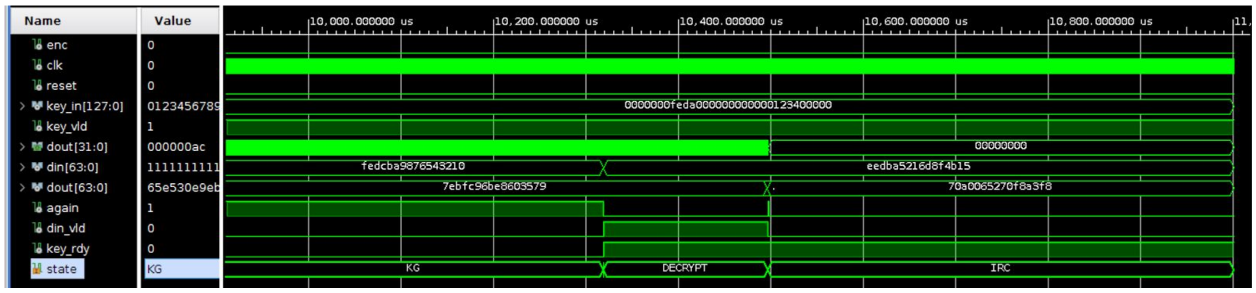


<u>Test Case 10:</u>

key_in = x“0000000feda00000000000001234000000”

din[63:0] = x“eedba5216d8f4b15”

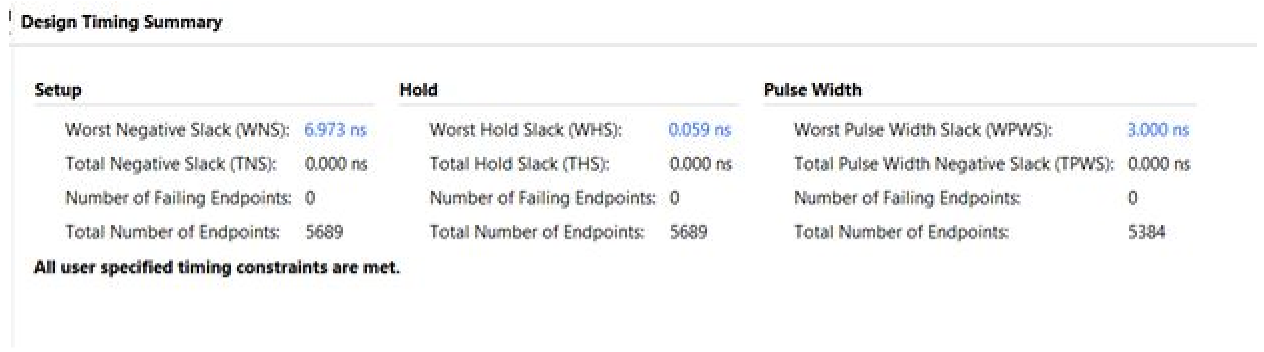
dout[63:0] = x“70a0065270f8a3f8”



c. Performance and area analysis of processor design

c1	Identify and explain the timing critical path in your processor design		
c2	Explain which instruction(s) use the critical path		
c3	Analyze and explain the area (i.e., LUTs, block RAM, FFs, etc.) for each sub-module in your processor, i.e., corresponding to the blocks in the block design		
c4	Explain how one could optimize and improve your design in the future		

C1 : The clock cycle time of the processor must accommodate the time taken for the longest critical path of any instruction.

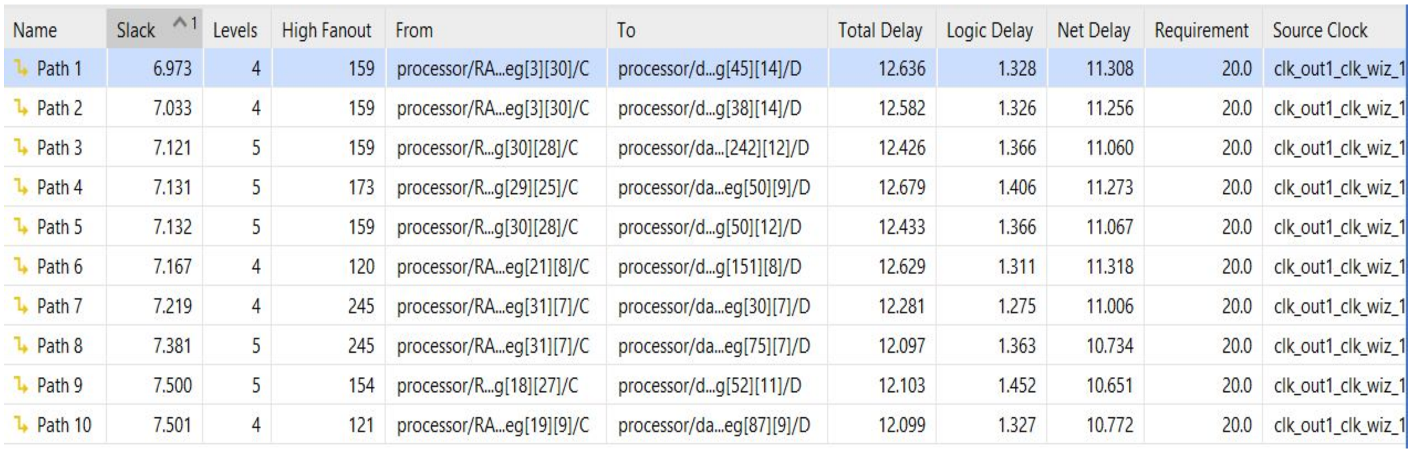


Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	6.973 ns	Worst Hold Slack (WHS):	0.059 ns	Worst Pulse Width Slack (WPWS):	3.000 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	5689	Total Number of Endpoints:	5689	Total Number of Endpoints:	5384

All user specified timing constraints are met.

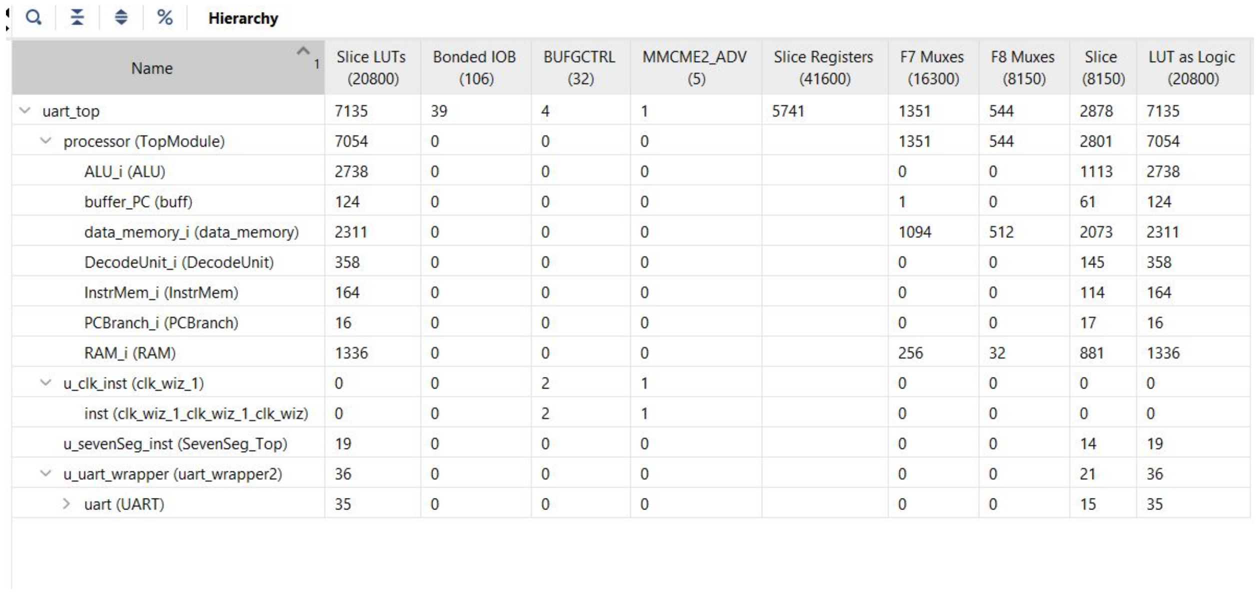
C2 : The load word (lw) and store word (sw) instructions use the critical path as can be seen from the figure below. Critical path is between Data Memory and Registers.



Name	Slack ^1	Levels	High Fanout	From	To	Total Delay	Logic Delay	Net Delay	Requirement	Source Clock
Path 1	6.973	4	159	processor/RA...eg[3][30]/C	processor/d...g[45][14]/D	12.636	1.328	11.308	20.0	clk_out1_clk_wiz_1
Path 2	7.033	4	159	processor/RA...eg[3][30]/C	processor/d...g[38][14]/D	12.582	1.326	11.256	20.0	clk_out1_clk_wiz_1
Path 3	7.121	5	159	processor/R...g[30][28]/C	processor/da...[242][12]/D	12.426	1.366	11.060	20.0	clk_out1_clk_wiz_1
Path 4	7.131	5	173	processor/R...g[29][25]/C	processor/da...eg[50][9]/D	12.679	1.406	11.273	20.0	clk_out1_clk_wiz_1
Path 5	7.132	5	159	processor/R...g[30][28]/C	processor/d...g[50][12]/D	12.433	1.366	11.067	20.0	clk_out1_clk_wiz_1
Path 6	7.167	4	120	processor/RA...eg[21][8]/C	processor/d...g[151][8]/D	12.629	1.311	11.318	20.0	clk_out1_clk_wiz_1
Path 7	7.219	4	245	processor/RA...eg[31][7]/C	processor/da...eg[30][7]/D	12.281	1.275	11.006	20.0	clk_out1_clk_wiz_1
Path 8	7.381	5	245	processor/RA...eg[31][7]/C	processor/da...eg[75][7]/D	12.097	1.363	10.734	20.0	clk_out1_clk_wiz_1
Path 9	7.500	5	154	processor/R...g[18][27]/C	processor/d...g[52][11]/D	12.103	1.452	10.651	20.0	clk_out1_clk_wiz_1
Path 10	7.501	4	121	processor/RA...eg[19][9]/C	processor/da...eg[87][9]/D	12.099	1.327	10.772	20.0	clk_out1_clk_wiz_1

C3:

The following table shows the area usage by all the submodules and the whole processor.



Hierarchy

Name	Slice LUTs (20800)	Bonded IOB (106)	BUFGCTRL (32)	MMCME2_ADV (5)	Slice Registers (41600)	F7 Muxes (16300)	F8 Muxes (8150)	Slice (8150)	LUT as Logic (20800)
uart_top	7135	39	4	1	5741	1351	544	2878	7135
processor (TopModule)	7054	0	0	0		1351	544	2801	7054
ALU_i (ALU)	2738	0	0	0		0	0	1113	2738
buffer_PC (buff)	124	0	0	0		1	0	61	124
data_memory_i (data_memory)	2311	0	0	0		1094	512	2073	2311
DecodeUnit_i (DecodeUnit)	358	0	0	0		0	0	145	358
InstrMem_i (InstrMem)	164	0	0	0		0	0	114	164
PCBranch_i (PCBranch)	16	0	0	0		0	0	17	16
RAM_i (RAM)	1336	0	0	0		256	32	881	1336
u_clk_inst (clk_wiz_1)	0	0	2	1		0	0	0	0
inst (clk_wiz_1_clk_wiz_1_clk_wiz)	0	0	2	1		0	0	0	0
u_sevenSeg_inst (SevenSeg_Top)	19	0	0	0		0	0	14	19
u_uart_wrapper (uart_wrapper2)	36	0	0	0		0	0	21	36
uart (UART)	35	0	0	0		0	0	15	35

C4:

To optimize the processor, we can implement pipelining or multi threading, or can increase the bus width of input.

d. Description of RC5 implementation in assembly

d1	Commented assembly-level code for RC5 encryption		
d2	Commented assembly-level code for RC5 decryption		
d3	Commented assembly-level code for RC5 round-key generation		
d4	Analyze the performance of RC5 Encryption using *your* processor		
	Explain the number of clock cycles required to perform the data-dependent rotation operation		
	Explain the number of clock cycles required to perform the other operations		
	Explain the total number of clock cycles required overall		
d5	Analyze the performance of RC5 Decryption using *your* processor		
	Explain the number of clock cycles required to perform the data-dependent rotation operation		
	Explain the number of clock cycles required to perform the other operations		
	Explain the total number of clock cycles required overall		
d6	Analyze the performance of RC5 Round Key Generation using *your* processor		
	Explain the number of clock cycles required to perform the data-dependent rotation operation		
	Explain the number of clock cycles required to perform the other operations		

	Explain the total number of clock cycles required overall		

d1) Key Expansion Assembly

```
000000 00000 00000 00000 0000 0 010010    //AND R0, R0, R0
-- Load User Key(128 bits) from the data memory
000111 00000 10101 0000000000111100       // LW R0, R21,#60
000111 00000 10100 0000000000111110       // LW R0, R20,#62
000111 00000 10011 0000000001000000       // LW R0, R19,#64
000111 00000 10010 0000000001000010       // LW R0, R18,#66
-- Store User Key(128 bits) to a location in the data memory
001000 00000 10010 0000000010011000       //SW R0, R18, #152
001000 00000 10011 0000000010011010       //SW R0, R18, #154
001000 00000 10100 0000000010011100       //SW R0, R18, #156
001000 00000 10101 0000000010011110       //SW R0, R18, #158
-- R10=2
000100 00000 01010 0000000000000010       //ORI R0, R10, #2
--R19=4
000100 00000 10011 0000000000000100       //ORI R0, R19, #4
--R20=100
000100 00000 10100 0000000001100100       //ORI R0, R20, #100
--R20=56
000100 00000 11010 0000000000111000       //ORI R0, R26, #56
-- Branch if R19=R26
001010 10011 11010 0000000000000101       //BEQ R19, R26, #5
--Load from location mentioned in R19 to R21
000111 10011 10101 0000000000000000       //LW R19, R21, #0
-Store the contents of R21to location mentioned in R20
001000 10100 10101 0000000000000000       //SW R20, R21,0
 --counter increment for L
000000 10011 01010 10011 0000 0 010101    //LRAD R19,R10,R19
--counter increment for S
000000 10100 01010 10100 0000 0 010101    //LRAD R20, R10, R20
```

```
--Jump to a location
001100 11111111111111111111111010                // JMP #-3
000011 00000 11000 0000000000000000       //ANDI R0,R24,#0
000011 00000 11001 0000000000000000       //ANDI R0,R25,#0

000011 00000 11010 0000000000000000       //ANDI R0,R26,#0
000011 00000 11011 0000000000000000       //ANDI R0,R27,#0
000011 00000 11100 0000000000000000       //ANDI R0,R28,#0
000100 00000 11101 0000000000011010       //ORI R0,R29,#26
000100 00000 11110 0000000000000100       //ORI R0,R30,#4
000100 00000 10101 0000000001001110       //ORI R0,R21,#78
 -- counter for S and L
000100 00000 00111 0000000000000001       //ORI R0, R7,#1
000100 00000 10011 0000000010011000       //ORI R0, R19,#152
000100 00000 10100 0000000001100100       //ORI R0, R20,#100
000000 11000 11001 11000 0000 0 010101    //LRAD R24,R25,R24
000111 10100 10110 0000000000000000       //LW R20, R22, #0
-- S+A+B
000000 11000 10110 10110 0000 0 010101    //LRAD R24, R22, R22
--(S+A+B)<<<3=>A
000000 10110 00000 10110 0011 0 010000    //XRLR R22, R0, R22
000000 10110 00000 11000 0000 0 010011    //OR R22, R0, R24
--updating S[i] in DM
001000 10100 10110 0000000000000000       //SW R20,R22,#0
000000 11000 11001 11001 0000 0 010101    //LRAD R24, R25, R25
000111 10011 10010 0000000000000000       //LW R19 , R18, #0
-- L+A+B
000000 11001 10010 10010 0000 0 010101    //LRAD R25, R18, R18, #0
 -- Rotate Amount
000011 11001 11001 0000000000011111       //ANDI R25, R25, # 31
--branch if R0= R25
001010 00000 11001 0000000000000011       //BEQ R0,R25, #3
--Left Rotate by 1
000000 10010 00000 10010 0001 0 010101    //LRAD R18, R0, R18
-- Subtract the counter by 1 (Inner Loop1)
000000 11001 00111 11001 0000 0 010110    //SBRR R25, R7, R25
-- Jump to A location everytime this is encountered
001100 11111111111111111111111100                //JMP #-4
000000 10010 00000 11001 0000 0 010011    //OR R18,R0,R25
```

```
--updating L[j] in DM
001000 10011 10010 0000000000000000       //SW R19, R18, #0
--Counter increment for L
000000 10011 01010 10011 0000 0 010101    //LRAD R19, R10, R19
--Counter increment for S
000000 10100 01010 10100 0000 0 010101    //LRAD R20, R10, R20
--i=i+1
000000 11010 00111 11010 0000 0 010101    //LRAD R26, R7, R26
--j=j+1
000000 11011 00111 11011 0000 0 010101    //LRAD R27,R7, R27
--k=k+1
000000 11100 00111 11100 0000 0 010101    //LRAD R28, R7, R28
 --Branch if R29/=R26
001011 11101 11010 0000000000000010       //BNE R29, R26, #2
000011 00000 11010 0000000000000000       //ANDI R0, R26, #0
000100 00000 10100 0000000001100100       //ORI R0, R20, #100
--Branch if R30/=R27
001011 11110 11011 0000000000000010       //BNE R30,R27, #2
-- j=0
000011 00000 11011 0000000000000000       //ANDI R0, R27, #0
-- offset for L
000100 00000 10011 0000000010011000       //ORI R0,R19, #152
Branch if R21= R28
001010 10101 11100 0000000000000001       //BEQ R21,R28, #1
-- Jump to A location everytime this is encountered
001100 11111111111111111111100011                 //JMP #29
 111111 00000000000000000000000000        //HLT
```

d2) <u>ENCRYPTION ASSEMBLY</u>

```
000000 00000 00000 00000 0000 0 010010    //AND R0, R0,R0
-- Load A and B from the data memory
000111 00000 00001 0000000000000000       //LW R0, R1,#0
000111 00000 00010 0000000000000010       //LW R0, R2,#2
-- Load skey(0) and add to A => R3
000111 00000 11111 0000000001100100       //LW R0, R31, #100
000000 00001 11111 00011 0000 0 010101    //LRAD R1, R31, R3
-- Load skey(1) and add to B => R4
000111 00000 11111 0000000001100110       //LW R0, R31, #102
```

```
000000 00010 11111 00100 0000 0 010101    //LRAD R2, R31,R4
--Load R8 with 12 for the main loop and R7 as 1 for decrementing the
counters
000100 00000 00100 0000000000001100       //ORI R0, R8, #12
000100 00000 00111 0000000000000001       //ORI R0, R7,#1
-- Load R9 and R10 with initial DM location for Skey and the counter
000100 00000 01001 0000000001101000       //ORI R0, R9,#104
000100 00000 01010 0000000000000010       //ORI R0, R10,#2
--Make a copy of A to implement the counters for left-rotates.  We use only
the LSB 5 bits for rotation
--       ********************    Main loop start  ********************
000011 00100 00101 0000000000011111       //ANDI R4, R5, #31
-- Xor A and B
000000 00011 00100 00011 0000 0 010000    //XRLR R3, R4, R3, #0
-- Branch to location 44 when the counter is zero

--       ************************  Label for start of Inner loop 1  *************
 001010 00000 00101 0000000000000011      //BEQ R0, R5, #3
--Left Rotate by 1
00000 00011 00000 00011 0 010101          //LRAD R3, R0, R4, #1
-- Subtract the counter by 1 (Inner Loop1)
00000 00101 00111 00101 0000 0 010110     //SBRR R5, R7, R5, #0
-- Jump to location 34 everytime this is encountered
001100 11111111111111111111111100                 //JMP #-4

--       ************************  End of Inner Loop 1  *****************
-- Load the new Skey from the DM
000111 01001 11111 0000000000000000       //LW R9, R31,#0
-- Increase the Skey address counter  by 2
000000 01001 01010 010101 0000 0 010101 //LRAD R9, R10,R9
-- Post round addition of the Skey to Reg3 => (Updated A)
000000 00011 11111 00011 0000 0 010101    //LRAD R3, R31, R3
--Make a copy of B to implement the counters for left-rotates.  We use only
the LSB 5 bits for rotation
000011 00011 00110 0000000000011111       //ANDI R3, R6, #31
-- Xor the updated value of A and the previous value of B
000000 00011 00100 00100 0000 0 010000    //XRLR R3, R4, R4
--       ****************   Label for start of Inner loop 2   ***********************
-- Branch to location 68 when the counter is zero
```

```
001010 00000 00110 0000000000000011       //BEQ R0, R6,#3
--Left Rotate by 1
000000 00100 00000 00100 0001 0 010000    //LRAD R4, R0, R4
-- Subtract the counter by 1 (Inner Loop2)
000000 00110 00111 00110 0000 0 010110     //SBRR R6, R7, R6
-- Jump to location 58 everytime this is encountered
001100 11111111111111111111111100                //JMP, #-4
--******************  End of Inner Loop 2  ****************************
-- Update the value of the Skey into register R31
000111 01001 11111 0000000000000000       //LW R9, R31, #0
-- Increase the Skey address counter  by 2
000000 01001 01010 01001 0000 0 010000    //LRAD R9, R10, R9
-- Post round addition of the Skey to Reg3 => (Updated B)
000000 00100 11111 00100 0000 0 010000    //LRAD R4, R31, R4

-- Decrement the Main loop counter by 1
000000 01000 00111 01000 0000 0 010110    //SBRR R8, R7, R8
-- If the counter is zero, Branch to location 80 which is the Halt instruction
001010 00000 01000 0000000000000001       //BEQ R0,R8,#1
--Jump to Location 2c everytime this is encountered
001100 11111111111111111111101011                //JMP,# 21
--****************  End of Main Loop   ************************
001000 00000 00011 0000000000111000       //SW R0, R3, #56
001000 00000 00100 0000000000111010       //SW R0, R4, #58
-- Halt the processor
111111 00000000000000000000000000                //HLT
```

d3) **Decryption Assembly Code**

```
000000 00000 00000 00000 0000 0 010010    //AND R0, R0, R0
-- Load X and Y from the data memory
000111 00000 01101 0000000000000000       // LW R0, R31, #0
000111 00000 01110 0000000000000010       //LW R0,R14, #2
--Load R17 with 54 for the main loop
-- 000100 00000 10001 0000000010010110    //ORI R0, R17, #150
-- LOad R10 as 2 for decrementing the skey counter
```

```
000100 00000 01010 0000000000000010       //ORI R0, R10, #2
-- LOad R7 as 1 for decrementing the Main Loop
000100 00000 00111 0000000000000001       //ORI R0, R7,#1
--Load R8 with 12 for the main loop
000100 00000 01000 0000000000001100       //ORI R0, R8, #12
--Make a copy of X to implement the counters for left-rotates.  We use only
the LSB 5 bits for rotation
 --     ***********************  Start of Main Loop   ***********************
000011 01101 10000 0000000000011111       //ANDI R13, R16, #31
 -- Load skey(n) and subtract R17 by 2
 000111 10001 11111 0000000000000000      //LW R17, R31, #0
000000 10001 01010 10001 0000 0 010110    //SBRR R17,R10,R17
000000 01110 11111 01110 0000 0 010110    //SBRR R14, R31, R14
--***********************  Label for start of Inner loop 1  ****************
001010 00000 10000 0000000000000011       //BEQ R0, R16, #3
-- Right Rotate R14(Ys) by 1
000000 01110 00000 01110 0001 0 010110    //SBRR R14, R0, R14
-- Subtract the counter by 1 (Inner Loop1)
 000000 10000 00111 10000 0000 0 010110   //SBRR R16, R7, R16
-- Jump to location 34 everytime this is encountered
001100 11111111111111111111111100                 // JMP #-4
-- Xor the updated value of A and the previous value of B
000000 01110 01101 01110 0000 0 010000    //XRLR R14, R13, R14
 -- Load skey(n-1) and subtract R17 by 2
000111 10001 11111 0000000000000000       //LW R17, R31, #0
000000 10001 01010 10001 0000 0 010110    //SBRR R17, R10, R17
000000 01101 11111 01101 0000 0 010110    //SBRR R13, R31, R13
000011 01110 01111 0000000000011111       //ANDI R14, R15, #31
 001010 00000 01111 0000000000000011      //BEQ R0, R15, #3
-- Right Rotate R14(Ys) by 1
000000 01101 00000 01101 0001 0 010110    //SBRR R13, R0,R13
-- Subtract the counter by 1 (Inner Loop2)
000000 01111 00111 01111 0000 0 010110    //SBRR R15, R7, R15
-- Jump to location 34 everytime this is encountered
 001100 11111111111111111111111100                //JMP #-4
-- Xor the updated value of A and the previous value of B
000000 01110 01101 01101 0000 0 010000    //XRLR R14, R13, R13
   -- Decrement the Main loop counter by 1
000000 01000 00111 01000 0000 0 010110    //SBRR R8, R7, R8
```

```
-- If the counter is zero, Branch to location 80 which is the Halt instruction
001010 00000 01000 0000000000000001       //BEQ R0,R8, #1
--Jump to Location 2c everytime this is encountered
001100 11111111111111111111101011               //JMP ,#21
 --      *************  End of Main Loop   *********************
000111 10001 11111 0000000000000000       //LW R17,R31,#0
000000 01110 11111 01100 0000 0 010110    //SBRR R14, R31, R12
000000 10001 01010 10001 0000 0 010110    //SBRR R17, R10, R17
000111 10001 11111 0000000000000000       //LW R17, R31,#0
000000 01101 11111 01011 0000 0 010110    //SBRR R13, R31, R11    when
001000 00000 01011 0000000000111000       //SW R0, R11, #56
000 01100 0000000000111010                //SW R0, R12,#58
111111 00000000000000000000000000         //HLT
```

<u>d4-6) Performance of the Processor:</u>

<u>RC5 Encryption:</u>

There are two data dependent rotation operations in each loop in RC5 Encryption, One is dependent on “A” and the other one is dependent on “B”.

Therefore, the total number of clock cycles required to perform data dependent rotation operation is (4 * rot_amt(A) + 4 * rot_amt(B)) * 13. Note that the rotate amounts are calculated and updated in every iteration of the loop.

The number of clock cycles required to perform other operations is 11 + (2 + 5 + 6) * 13 + 3.

The main loop runs for 13 iterations. Hence the total number of clock cycles required to perform RC5 Encryption is 11 + (2 + 5 + 6 + 4 * rot_amt(A) + 4 * rot_amt(B)) * 13 + 3.

RC5 Decryption:

There are two data dependent rotation operations in each loop in RC5 Decryption, One is dependent on “A” and the other one is dependent on “B”.

Therefore, the total number of clock cycles required to perform data dependent rotation operation is (4 * rot_amt(A) + 4 * rot_amt(B)) * 13. Note that the rotate amounts are calculated and updated in every iteration of the loop.

The number of clock cycles required to perform other operations is 11 + (2 + 5 + 6) * 13 + 3.

The main loop runs for 13 iterations. Hence the total number of clock cycles required to perform RC5 Decryption is 11 + (2 + 5 + 6 + 4 * rot_amt(A) + 4 * rot_amt(B)) * 13 + 3.

RC5 Round Key Generation:

There are two data dependent rotation operations in each loop in Round Key Generation, One is a constant value (3) and the other one is dependent on the last five bits of “A+B”.

Therefore, the total number of clock cycles required to perform data dependent rotation is (1 + 4 * rot_amt(A+B)) * 78.

The number of clock cycles required to perform other operations is 12 + 6*26 + 11 + (10 + 7 + 3) * 78 + 6 + 40 + 1.

The total number of clock cycles required to perform RC5 Round Key Generation is 12 + 6*26 + 11 + (10 + 4 * rot_amt(A+B) 7 + 3 + 1) * 78 + 6 + 40 + 1.

e. Description of processor interfaces (how do you provide inputs and display results)

e1	Describe the user input/output interfaces for interacting with the processor		
	Interface for single-stepping		
	Interface for loading new program instructions		
	Interface for loading key/ciphertext/plaintext		
	Other FPGA-board interfaces (e.g., how the 7-segs, LEDs, SWs are used)		

<u>e1) Interfacing and Port Mapping:</u>

For single stepping, we are using BTNL button to control the processor.

For loading new program instructions, we are using UART to give the instructions.

UART is used to input “user key” and “plaintext” and output encryption/decryption output. Using a combination of switches, Seven segment display is used to display the contents of registers, user key, last 16 bits of encryption/decryption output, states, last 4 HEX digits of skey array, program counter value.

Switches are used to input some control signals such as clkselect (to select slower clock or faster clock), again (to change plaintext without changing user key), enc (whether to perform encryption or decryption).

A combination of BTND and SW(6) is used to display regfile, BTNL and SW(5) to control slow clock.

Contents of Data Memory are as follows:

Data Memory Locations	Contents
x“0000”-x“0011”	Input for Encryption/Decryption
x“0038”-x“003B”	Output for Encryption/Decryption
x“003C”-x“0043”	User Key(128 bit)
x“0064”-x“0097”	Skey(0)- Skey(25)
x“0098”-x“009F”	L_array(0)- L_array(7)

Switches/Buttons	Function
BTNC	Reset
BTNL	Single Stepping Clock
SW(8:7) = “11”	Single Stepping Clock Select
SW(8:7) = others	50 MHZ Clock Select

f. Details about how you verified your overall design

f1	For the simulation screenshots in b., explain how you designed the test cases and how you worked out what the correct output should be.		

f) For the simulation screenshots in b, we used different combinations of symmetric and unsymmetric plaintexts and keys so as to run our processor in all the critical cases. We cross-checked the outputs with our known-good working behavioural simulation of a model from Lab-6B.