EL6463 Project Report

32-bit Processor Design

Group No.08

Junchen Su	js9133
Liu Yang	ly1032
Zhiyu Wen	zw1305
Yuchen Zhang	yz3814

1. Introduction

In this project a 32-bit processor called NYU-6463 Processor was designed and implemented on the Nexys 4 Artix-7. The processor has three instruction types: (a) R-Type for arithmetic instructions, (b) I-Type for immediate value operations, load and store instructions, and (c) J-Type for jump instructions. To test the performance of the processor, we wrote the assembly program for RC5 algorithm, including key expansion, encryption, and decryption.

GitHub Repo

https://github.com/ConnorWen9398/AHD-Project.git

Performance Summary

Encryption Cycles for Assigned Test Vector	1030
Decryption Cycles for Assigned Test Vector	1032
Round Key Generation Cycles for Assigned Test Vector	4663
Maximum Frequency	137.8 MHz
LUT Utilization	4.5%
Number of Test Vectors for Functional Simulation	1001
Number of Test Vectors for Timing Simulation	5

2. Processor Structure

Key components includes ALU, Instruction Fetch, Control Unit, Data Memory, Register Memory, and etc. As shown in Figure 2.1, our design is based on a hierarchy of three levels, and the block diagram can be found in Appendix I.

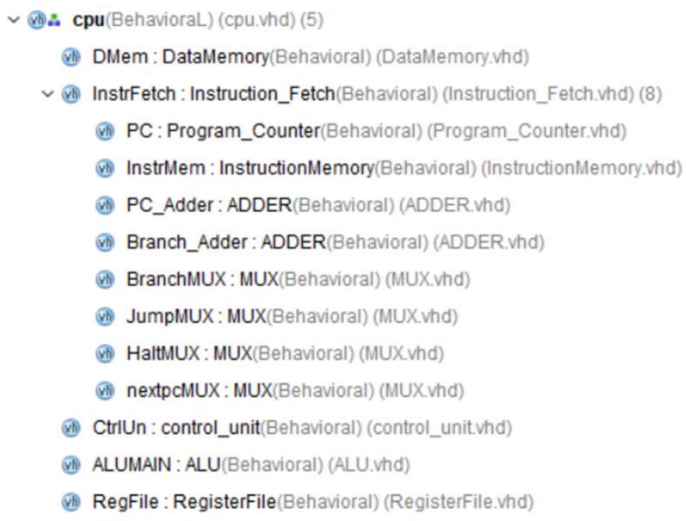


Figure 2.1. Architecture of the Processor.

3. Functional Simulation

The key expansion part starts start from 1 to 246 in instruction address, encryption from 247 to 658, and decryption from 247 to 260 & 659 to 1056.

For this testing case,

ukey:=x”a1b21111a1b22222a1b28888abcd1111” for key expansion

din=”dd0000bb00991111” for encryption

din=”6854b2a051e26180” for decryption

To verify the results in a more general range, 1000 cases are tested in Part 6 using TextIO

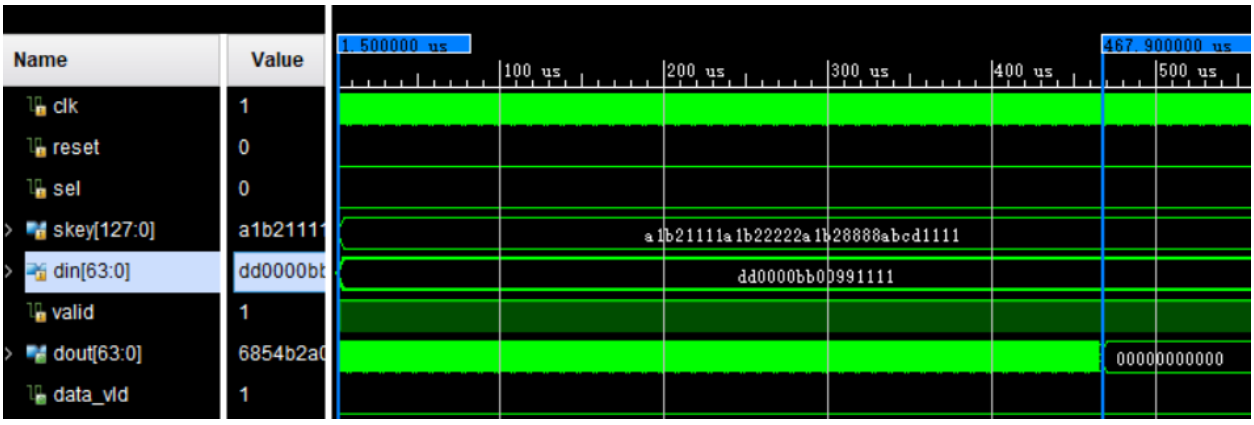


Figure 3.1. Key Expansion.

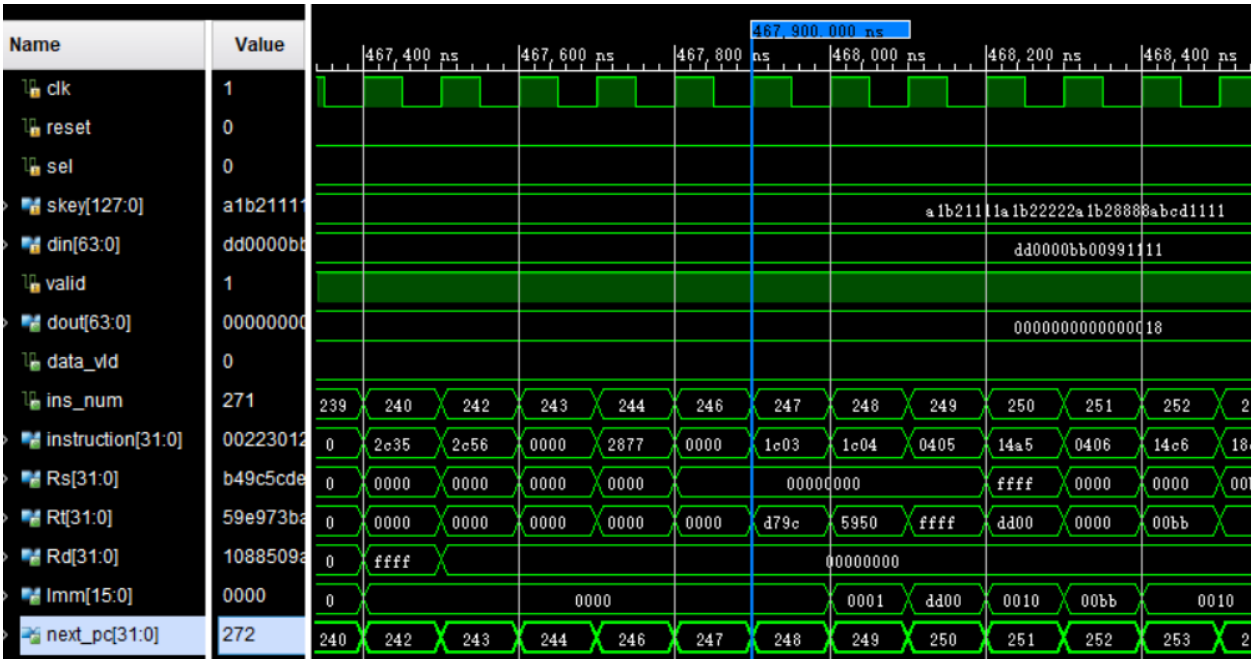


Figure 3.2. Key Expansion Detailed Waveform.

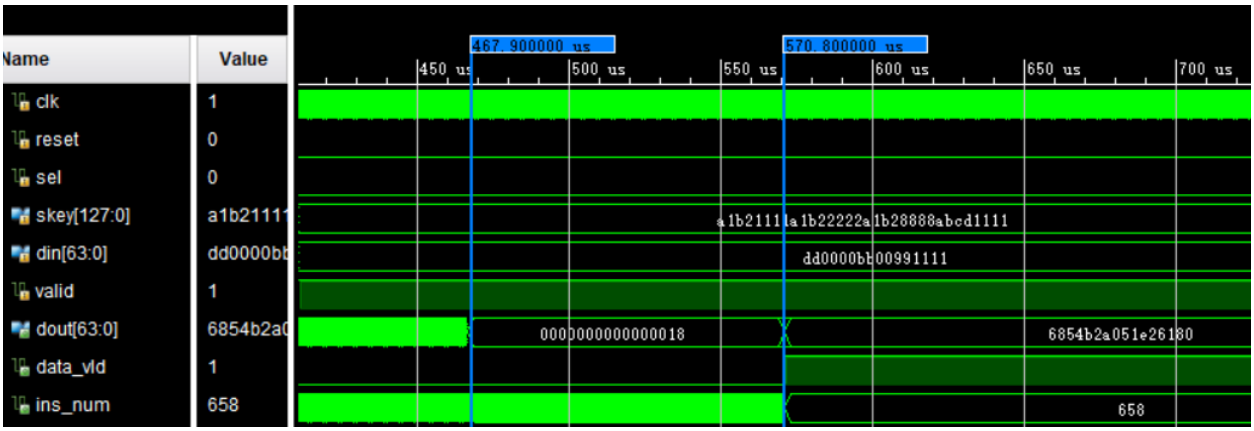


Figure 3.3. Encryption, with result 6854b2a051e26180.

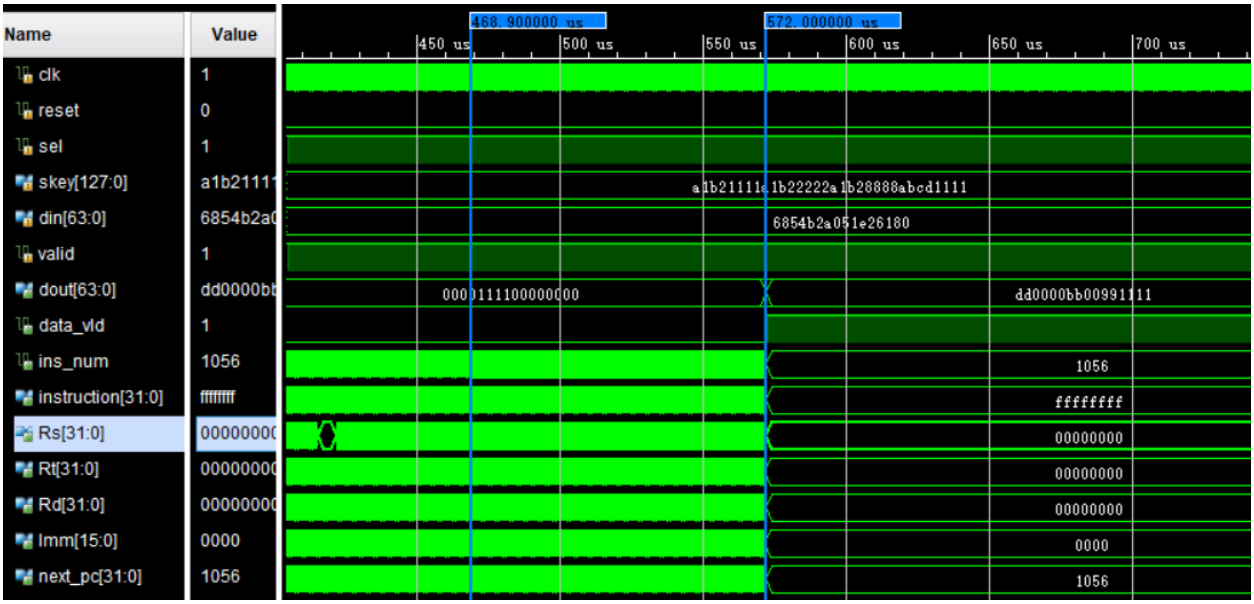


Figure 3.4. Decryption, with result dd0000bb00991111 .

4. Timing Simulation

4.1 Screenshots of Waveform

The skey will be stored in data memory [25 downto 0], we used five test cases for both encode and decode, and before them get executed, they key expansion will be run too. So 10 test cases for key expansion. All of our test case starts from 291ns, which is the time that the instruction begin to be executed, so we can calculate the total cycles of each function, and finally we will calculate the average latency of these functions.

Figure 4.1.0

Encoding Test case1:

ukey:=x”215447ea3c30c306e990034a87d6664e” din=”26f249bec27b6a35”

Results generated by C code:

skey: 9bbbd8c8 1a37f7fb 46f8e8c5 460c6085 70f83b8a 284b8303 513e1454 f621ed22 3125065d 11a83a5d d427686b 713ad82d 4b792f99 2799a4dd a7901c49 dede871a 36c03196 a7efc249 61a78bb8 3b0a1d2b 4dbfca76 ae162167 30d76b0a 43192304 f6cc1431 65046380

dout: 6c2ccfc34482ae08

Key Expansion:

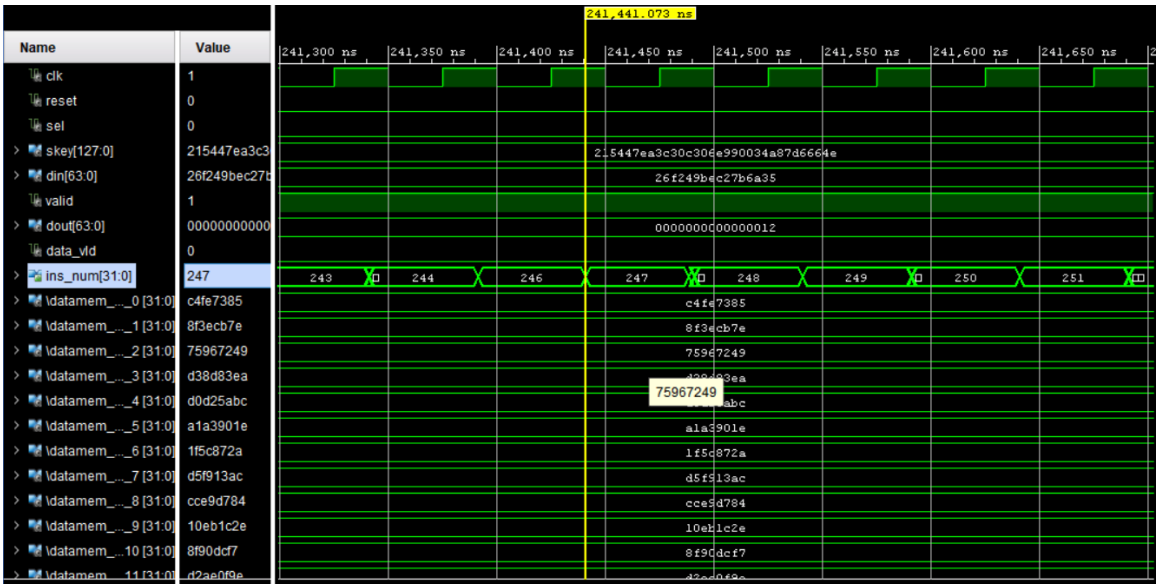


Figure 4.1.1

Encoding:

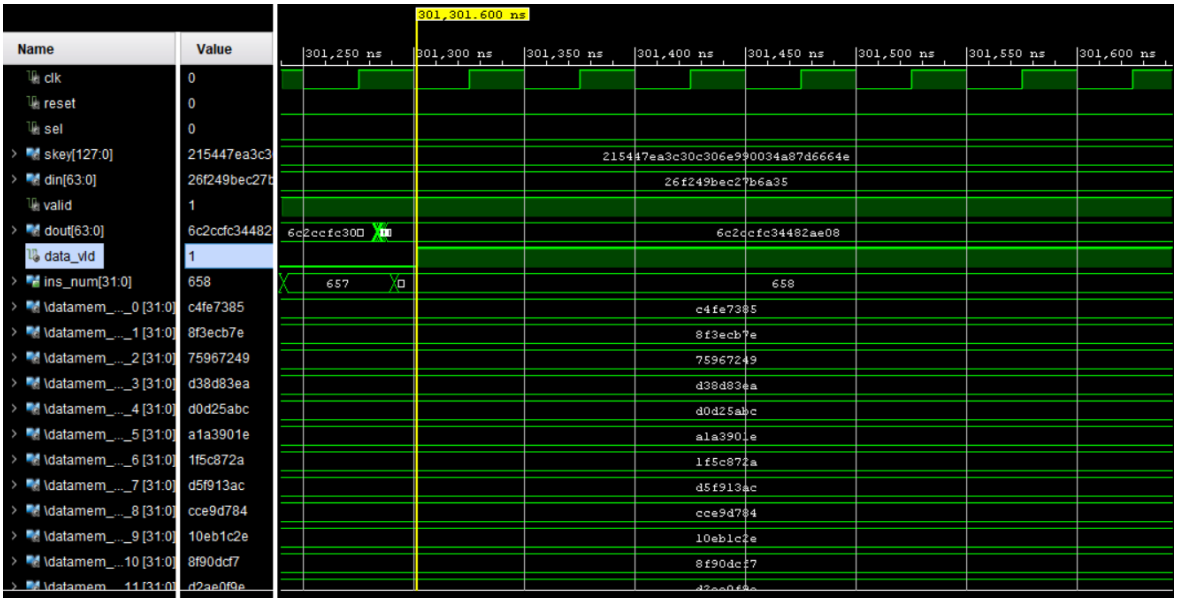


Figure 4.1.2

Encoding Test case2:

ukey:=x”915f4619be41b2516355a50110a9ce91” din=”eedba5216d8f4b15”

Results generated by C code:

skey: da22168a d00cc77e c36b960f ddcdd485 b7afcd42 2a013c76 b49bfa76 d55d677e 6ca89043 2d1f2e62 f2a3096c 5eb4f29c 7a1ff0b3 999f4e5d 22443887 9d6e90af cb96af38 97bcb916 f195c8c5 5feb96df fcac3a65 21e070f7 764c1b47 65fa14e2 13f9851f 31e62f57

dout: 2ab8bc63a296b969

Key Expansion:

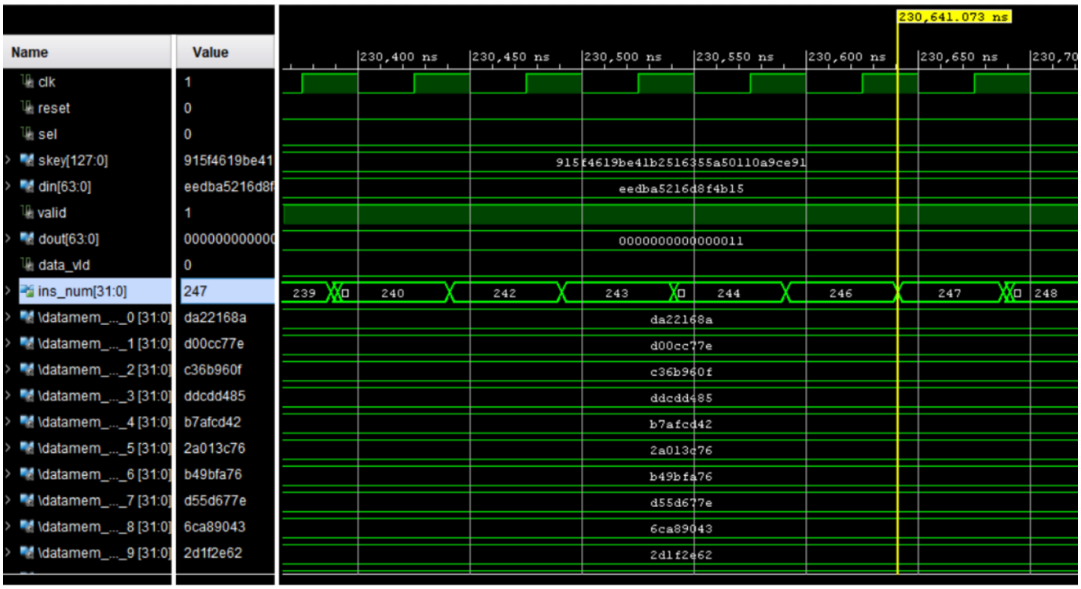


Figure 4.1.3

Encode:

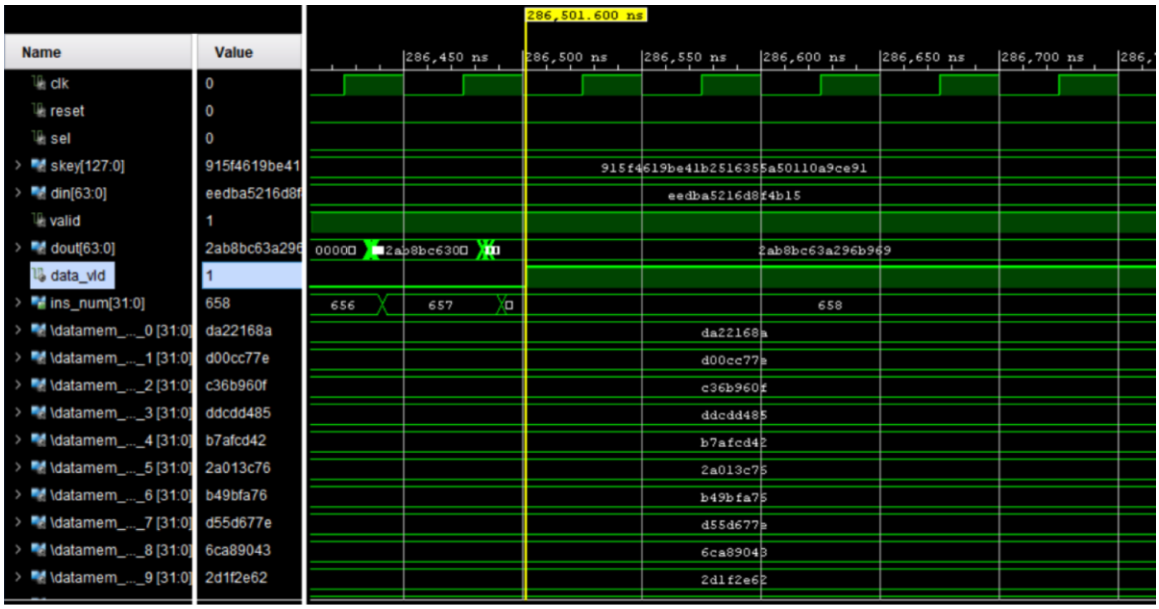


Figure 4.1.4

Encoding Test case3:

ukey:=x”0317a4b760adbaa3888ddaa727831103” din=”2ab8bc63a296b969”

Results generated by C code:

skey: 76145621 200edca6 f5689e9f e30540e4 8961fee2 326f7fdf 8b539794 6691637c aa8de1c8 8d952b58 0717ae17 5605d2cb 4c7d9520 69eebf50 ee79a5dd 9b13a3a5 ab9444f1 91bd3630 7d61b3a2 bdb1d70a 746f682f bf047814 605b9a27 702eccef a028fe3d 26bbab42

dout: ca890f566e6d7511

Key Expansion:

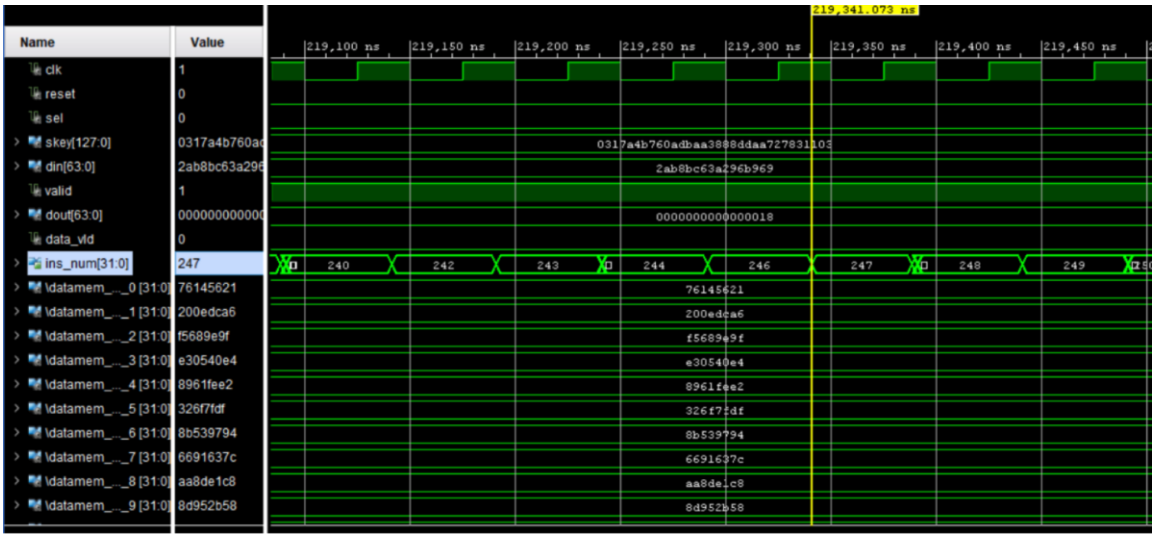


Figure 4.1.5

Encode:

Figure 4.1.6

Encoding Test case4:

ukey:=x”b9f8f092def0f53602c2147ab69808e6” din=”ca890f566e6d7511”

Results generated by C code:

skey: 62aff981 7293a930 dbaac5f3 dc6e0b4a f0f29881 773ec5db 8f25847b f4119559 ab2628a9 39323f32 303c58a6 7b9a708a 60e54974 7a107051 38aaacc4 9ff1ade1 dec16101 03690bda 8c970453 6557a021 78673a75 a8959705 9b3bfcbc 0e1f57e0 e5e755c2 e86c3228

dout: b2945e5eb833de33

Key Expansion:

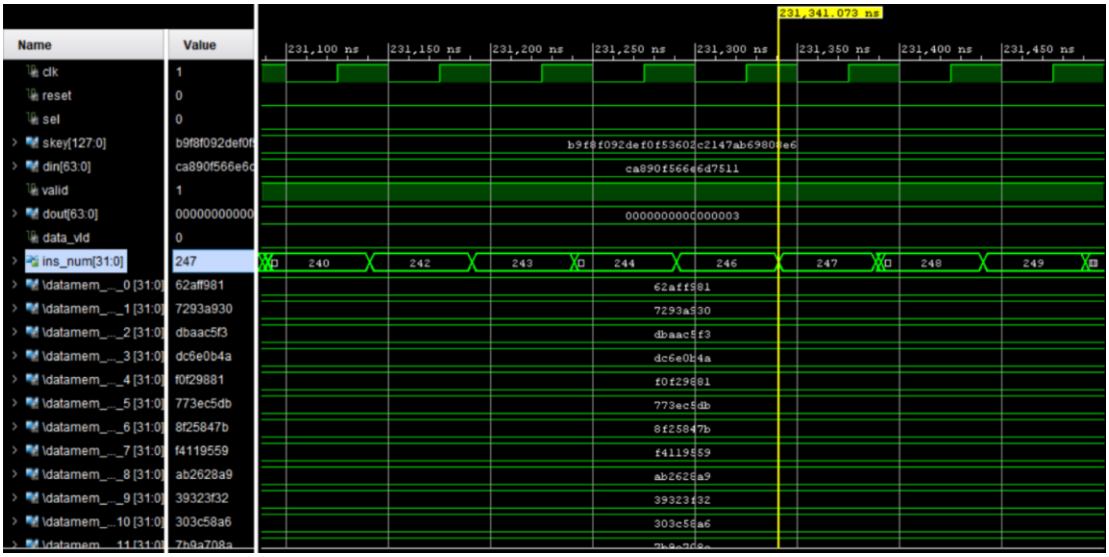


Figure 4.1.7

Encode:

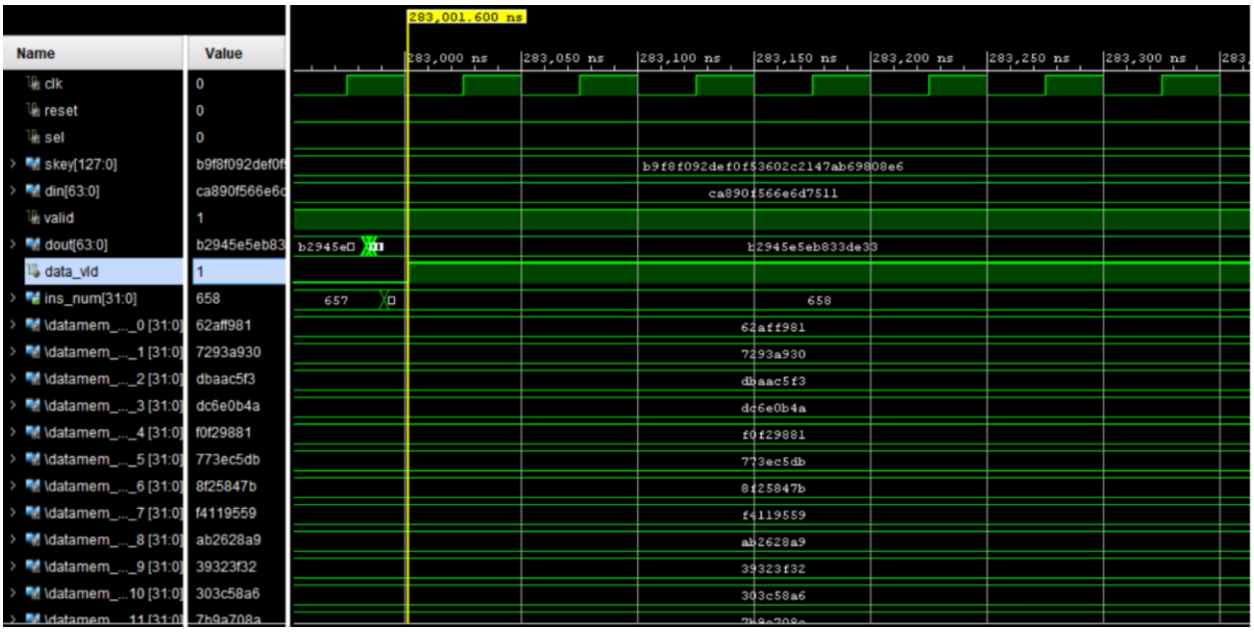


Figure 4.1.8

Encoding Test case5:

ukey:=x”ue040cbd769fb85516abe8aee6a302255e” din=”b2945e5eb833de33”

Results generated by C code:

skey: 55886c33 b7253089 385a037a cf39203d c0462d1e bf22be0e b3a26f0e b02087de 220b6129 a5d00e1c 42653bc3 17ea11f3 220c14ca 0e348b9d a140cff0 3b5a5c00 72549888 fd0a5fb5 fa2eafc1 1139b8a4 926a425a 06900e4a 753935a2 aa94c7c7 4f6fb6c5 ee22c8f0

dout: 26f249bec27b6a35

Key Expansion:

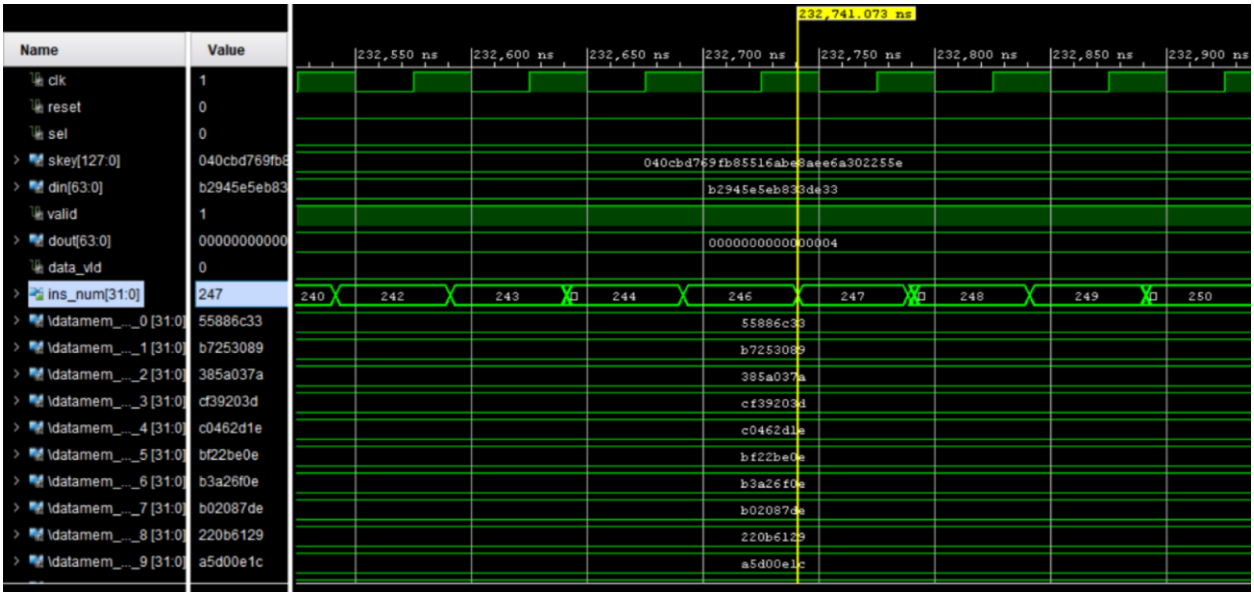


Figure 4.1.9

Encode:

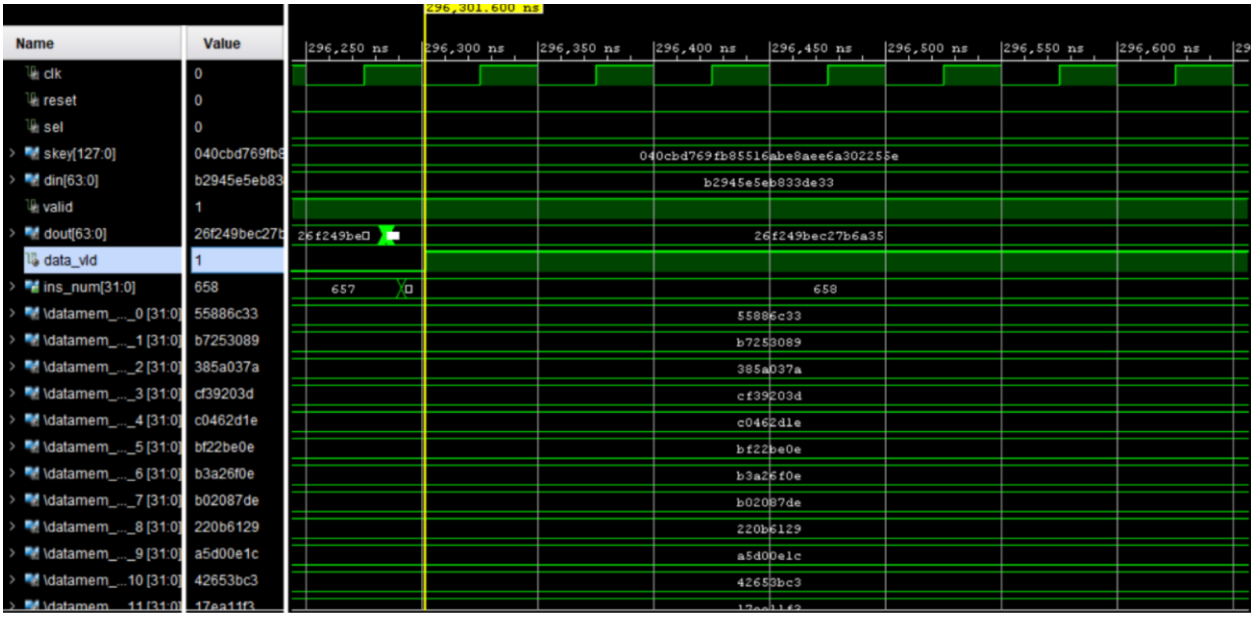


Figure 4.1.10

Decode Test case1:

ukey:=x”915f4619be41b2516355a50110a9ce91” din=”2ab8bc63a296b969”

Results generated by C code:

skey: da22168a d00cc77e c36b960f ddcdd485 b7afcd42 2a013c76 b49bfa76 d55d677e 6ca89043 2d1f2e62 f2a3096c 5eb4f29c 7a1ff0b3 999f4e5d 22443887 9d6e90af cb96af38 97bcb916 f195c8c5 5feb96df fcac3a65 21e070f7 764c1b47 65fa14e2 13f9851f 31e62f57

dout: eedba5216d8f4b15

Key Expansion

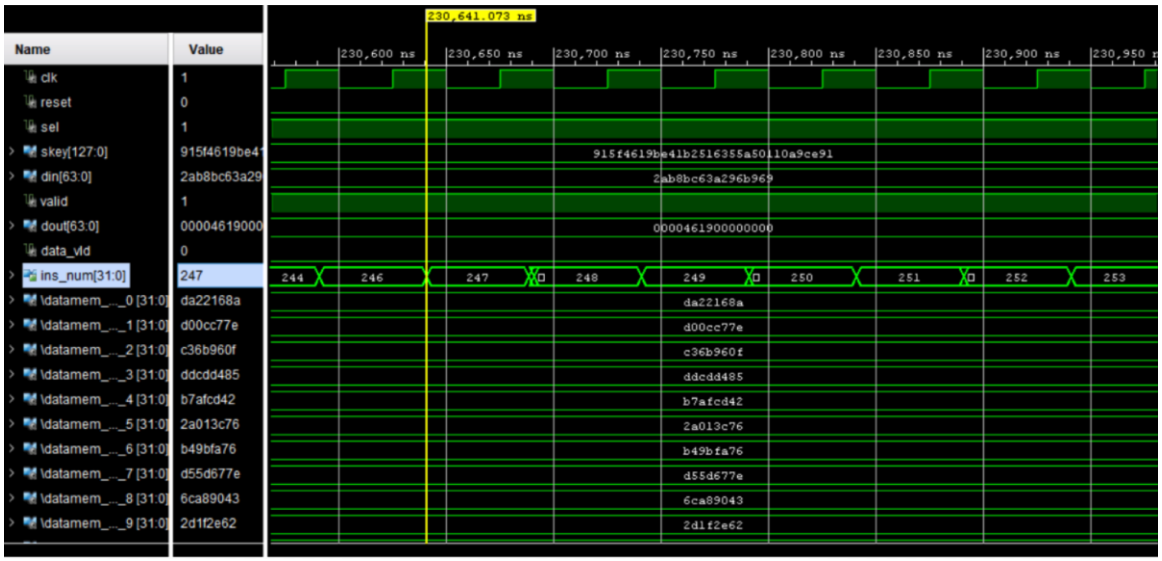


Figure 4.1.11

Decode:

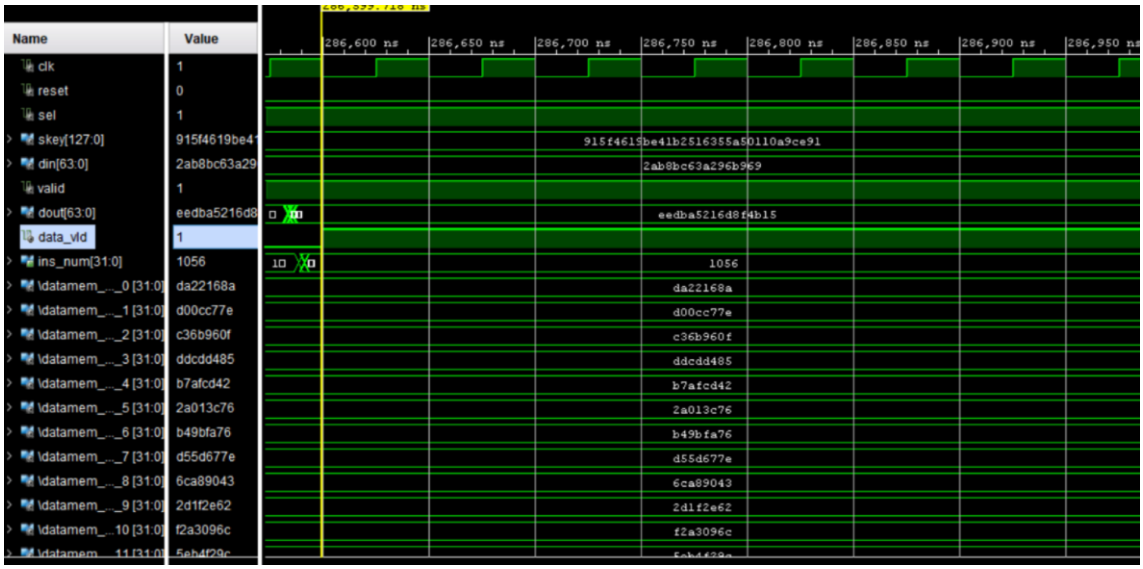


Figure 4.1.12

Decode Test case2:

ukey:=x”0317a4b760adbaa3888ddaa727831103” din=”ca890f566e6d7511”

Results generated by C code:

skey: 76145621 200edca6 f5689e9f e30540e4 8961fee2 326f7fdf 8b539794 6691637c aa8de1c8 8d952b58 0717ae17 5605d2cb 4c7d9520 69eebf50 ee79a5dd 9b13a3a5 ab9444f1 91bd3630 7d61b3a2 bdb1d70a 746f682f bf047814 605b9a27 702eccef a028fe3d 26bbab42

dout: 2ab8bc63a296b969

Key Expansion

Figure 4.1.13

Decode:

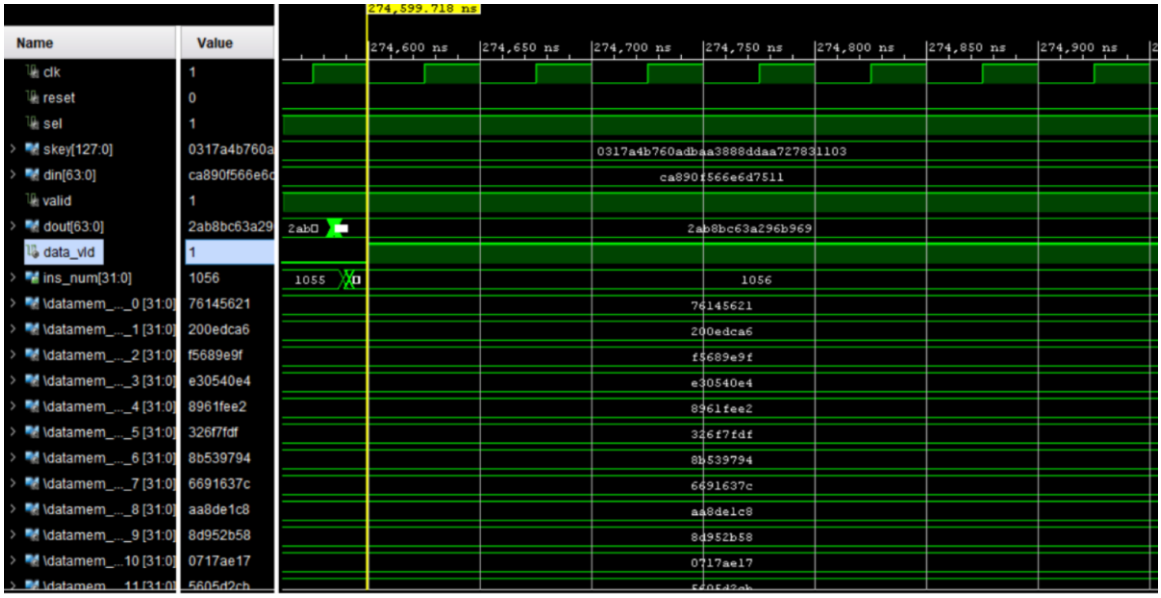


Figure 4.1.14

Decode Test case3:

ukey:=x”b9f8f092def0f53602c2147ab69808e6” din=”b2945e5eb833de33”

Results generated by C code:

skey: 62aff981 7293a930 dbaac5f3 dc6e0b4a f0f29881 773ec5db 8f25847b f4119559 ab2628a9 39323f32 303c58a6 7b9a708a 60e54974 7a107051 38aaacc4 9ff1ade1 dec16101 03690bda 8c970453 6557a021 78673a75 a8959705 9b3bfcbc 0e1f57e0 e5e755c2 e86c3228

dout: ca890f566e6d7511

Key:

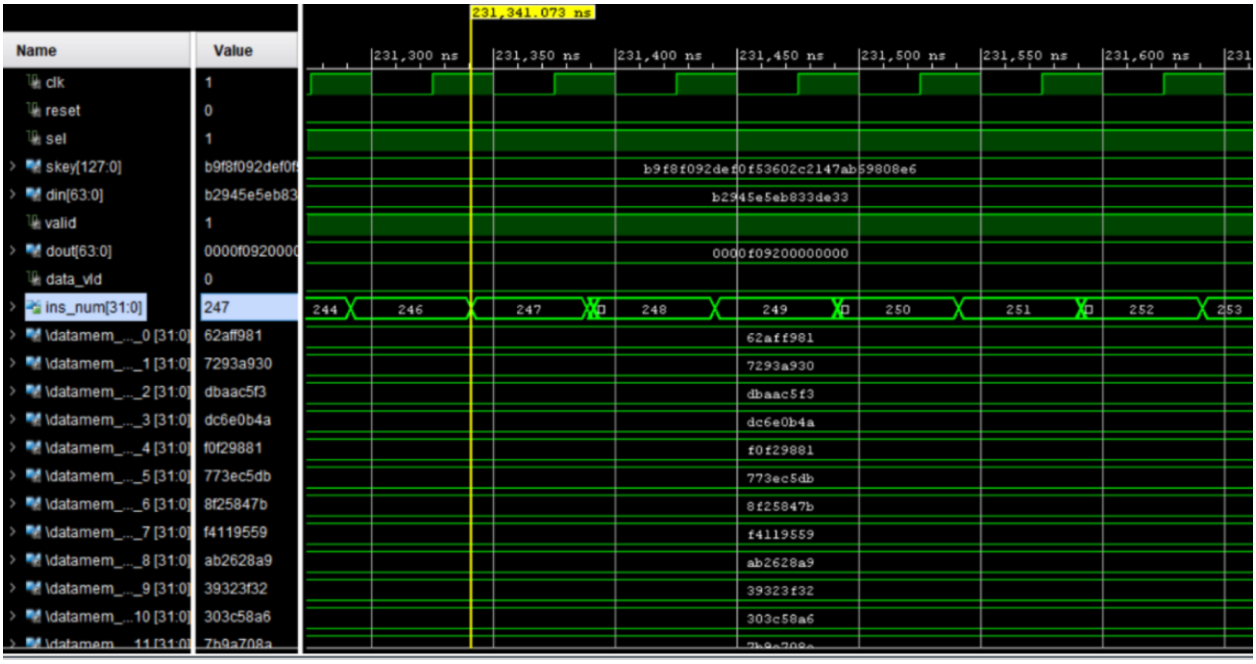


Figure 4.1.15

Decode:

Figure 4.1.16

Decode Test case4:

ukey:=x”040cbd769fb85516abe8aee6a302255e” din=”26f249bec27b6a35”

Results generated by C code:

skey: 55886c33 b7253089 385a037a cf39203d c0462d1e bf22be0e b3a26f0e b02087de 220b6129 a5d00e1c 42653bc3 17ea11f3 220c14ca 0e348b9d a140cff0 3b5a5c00 72549888 fd0a5fb5 fa2eafc1 1139b8a4 926a425a 06900e4a 753935a2 aa94c7c7 4f6fb6c5 ee22c8f0

dout: b2945e5eb833de33

Key Expansion:

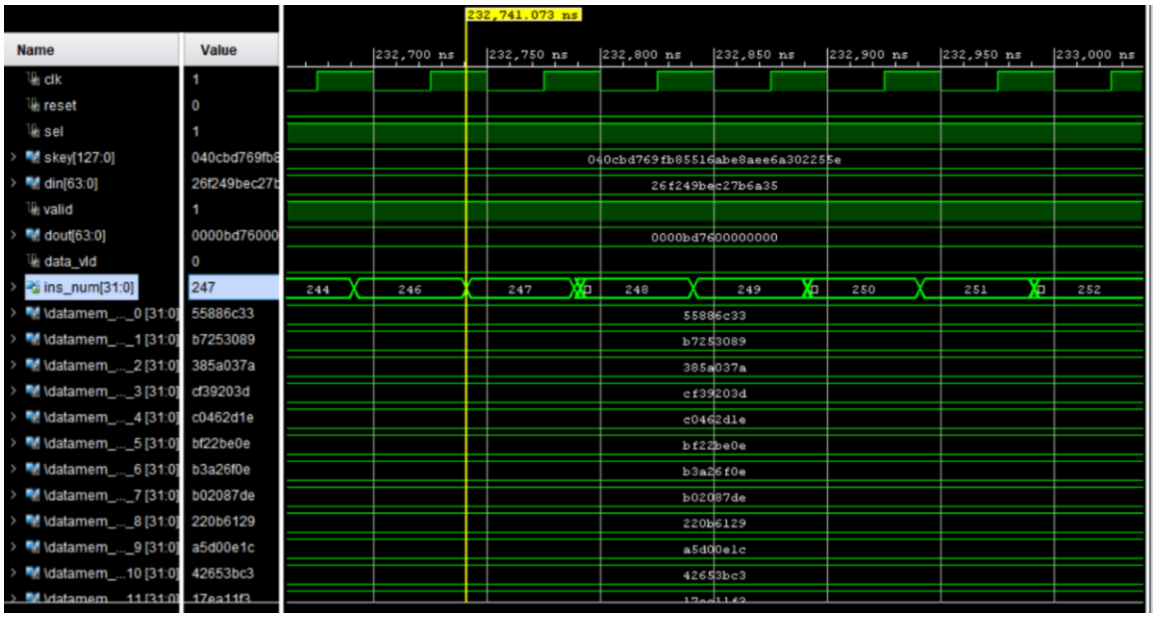


Figure 4.1.17

Decode:

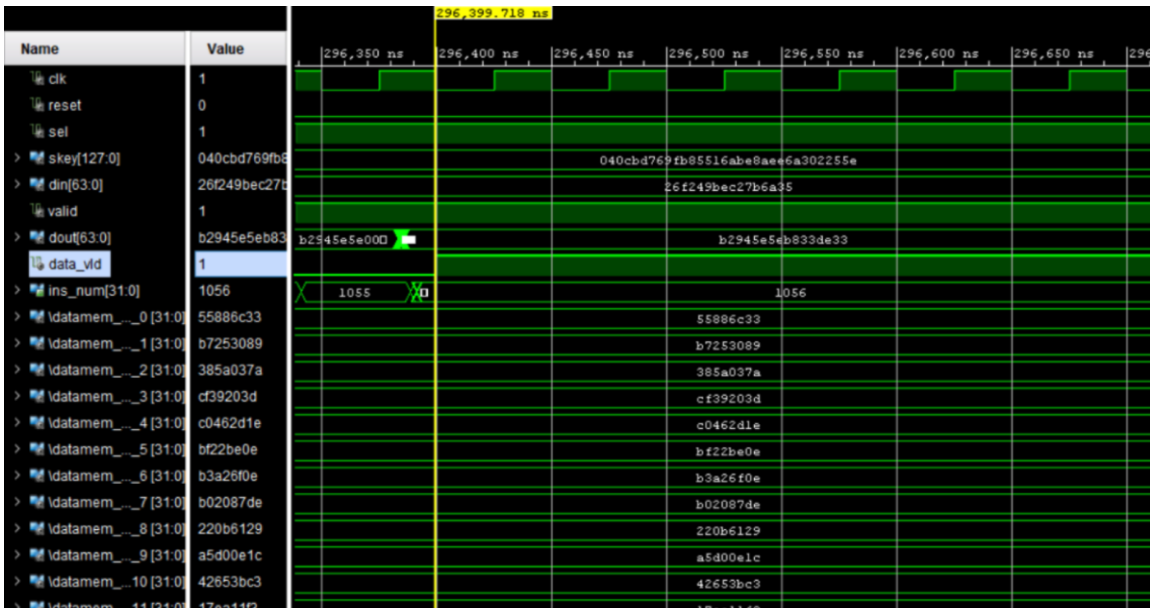


Figure 4.1.18

Decode Test case5:

ukey:=x”215447ea3c30c306e990034a87d6664e” din=”6c2ccfc34482ae08”

Results generated by C code:

skey: c4fe7385 8f3ecb7e 75967249 d38d83ea d0d25abc a1a3901e 1f5c872a d5f913ac cce9d784 10eb1c2e 8f90dcf7 d2ae0f9e 156df151 43ab34a6 c3e66456 31b6ea41 e7e88ad1 22129b1f 4d051e51 6d6791b5 7a0f5281 7190ade8 d37ad057 f683b767 3a94ddfa e3015728

dout: 26f249bec27b6a35

Key Expansion:

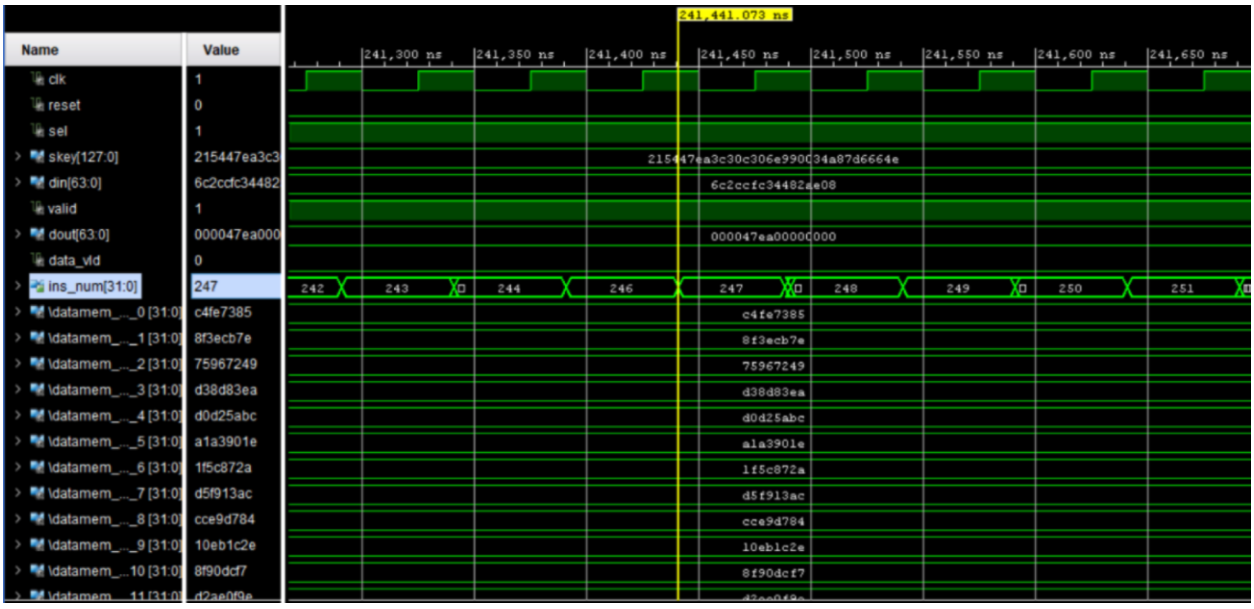


Figure 4.1.19

Decode:

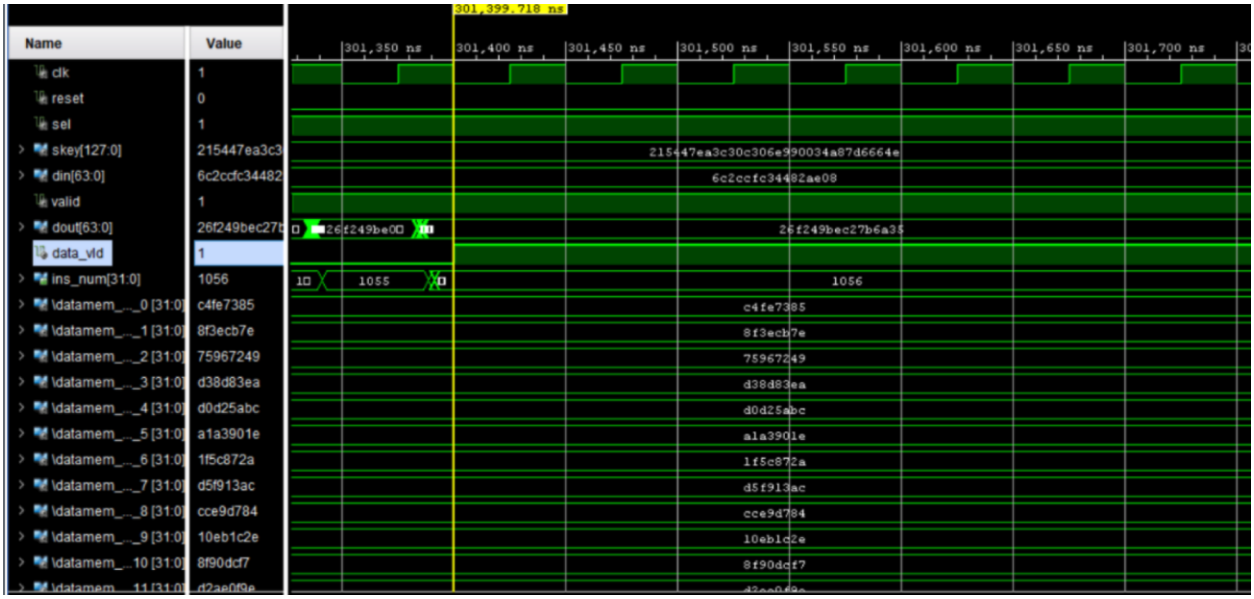


Figure 4.1.20

4.2 Resources Utilization

The resource utility of the synthesis is illustrated as below：

Resource	Utilization	Available	Utilization %
LUT	2876	63400	4.54
FF	2320	126800	1.83
IO	49	210	23.33

Figure 4.2.1

The time summary of the synthesis is illustrated as below:

```
Timing Summary:
---------------
Speed Grade: -3

   Minimum period: 11.928ns (Maximum Frequency: 83.837MHz)
   Minimum input arrival time before clock: 11.168ns
   Maximum output required time after clock: 15.961ns
   Maximum combinational path delay: 15.201ns

Timing Details:
---------------
All values displayed in nanoseconds (ns)

=========================================================================
Timing constraint: Default period analysis for Clock 'CLK'
  Clock period: 2.655ns (frequency: 376.634MHz)
  Total number of paths / destination ports: 1262 / 84
-------------------------------------------------------------------------
Delay:               2.655ns (Levels of Logic = 2)
  Source:            i_cnt2_13 (FF)
  Destination:       i_cnt2_0 (FF)
  Source Clock:      CLK rising
  Destination Clock: CLK rising

  Data Path: i_cnt2_13 to i_cnt2_0
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     FDR:C->Q              2   0.361   0.697  i_cnt2_13 (i_cnt2_13)
     LUT6:I0->O            1   0.097   0.683  i_cnt2[19]_GND_8_o_equal_32_o<19>3 (i_cnt2[19]_GND_8_o_equal_32_o<
     LUT5:I0->O           21   0.097   0.370  i_cnt2[19]_GND_8_o_equal_32_o<19>4 (i_cnt2[19]_GND_8_o_equal_32_o)
     FDR:R                     0.349          i_cnt2_0
    ----------------------------------------
```

Figure 4.2.2

The resource utility of the place and route is illustrated as below:

Resource	Utilization	Available	Utilization %
LUT	2874	63400	4.53
FF	2320	126800	1.83
IO	49	210	23.33

Figure 4.2.3

The time summary of the place and route is illustrated as below:

```
Timing Score: 0 (Setup: 0, Hold: 0, Component Switching Limit: 0)

Asterisk (*) preceding a constraint indicates it was not met.
   This may be due to a setup or hold violation.

--------------------------------------------------------------------------------------------------------
  Constraint                                |    Check    | Worst Case |  Best Case | Timing |   Timing
                                            |             |    Slack   | Achievable | Errors |    Score
--------------------------------------------------------------------------------------------------------
  TS_sys_clk_pin = PERIOD TIMEGRP "sys_clk_ | SETUP       |     7.258ns|     2.742ns|       0|           0
  pin" 100 MHz HIGH 50%                     | HOLD        |     0.264ns|            |       0|           0
--------------------------------------------------------------------------------------------------------
```

Figure 4.2.4

4.3 Timing Performance

Table 1. Performance Summary.

Critical path delay	7.258 ns
Maximum clock frequency	137.779 MHz
Latency	4616 clock cycles for key expansion 1136 clock cycles for enc 1146 clock cycles for dec
Propagation delay	33.50 us for key expansion 8.25 us for enc 8.32 us for dec

5. RC5 in Assembly Code and Machine Code

5.1 Key Expansion

```
"00000000000000000000010000010000",--0 ADD RO,RO, R1 R1=0
"00000000000000000000100000010000",--1 ADD R0,RO, R2 R2=0
"00000000000000000000110000010000",--2 ADD RO,RO, R3 R3=0
"00000100000111100000000000000000",--3 ****ADDI,RO,R30,skey(31 downto 16) R30=XXX..skey(31 downto 16)
"00010111110111100000000000010000",--4 SHL R30 R30 16 R30=skey(31 downto 16)...000
"00000100000111010000000000000000",--5 ****ADDI,RO,R29,SKEY(15 downto 0) R29=XXXi..skey(15 downto 0)
"00010111101111010000000000010000",--6 SHL R29 R29 16 R29=skey(15 downto 0)i000     remove sign extension
"00011101101111010000000000010000",--7 SHR R29 R29 16 R29=000i..skey(15 downto 0)     remove sign extension
"00000011110111011111000000010000",--8 ADD,R30 R29,R30 R30=skey(31 downto 0)
"00100000000111100000000000011010",--9 SW RO, R30 offset M[0+offset1]<-R30
"00000100000111100000000000000000",--10 ******ADDI,RO,R30,skey(63 downto 48) R30=XXX..skey(63 downto 48)
"00010111110111100000000000010000",--11 SHL R30 R30 16 R30=skey(63 downto 48)...000
"00000100000111010000000000000000",--12 **** ADDI,RO,R29,SKEY(47 downto 32) R29=XXXi..skey(47 downto 32)
"00010111101111010000000000010000",--13 SHL R29 R29 16 R29=skey(47 downto 32)i000     remove sign extension
"00011101101111010000000000010000",--14 SHR R29 R29 16 R29=000i..skey(47 downto 32)     remove sign extension
"00000011110111011111000000010000",--15 ADD,R30 R29,R30 R30=skey(63 downto 32)
"00100000000111100000000000011011",--16 SW RO, R30 offset M[0+offset2]<-R30
"00000100000111100000000000000000",--17 ****ADDI,RO,R30,skey(95 downto 80) R30=XXX..skey(95 downto 80)
"00010111110111100000000000010000",--18 SHL R30 R30 16 R30=skey(95 downto 80)...000
"00000100000111010000000000000000",--19 **** ADDI,RO,R29,SKEY(79 downto 64) R29=XXXi..skey(79 downto 64)
"00010111101111010000000000010000",--20 SHL R29 R29 16 R29=skey(79 downto 64)i000     remove sign extension
"00011101101111010000000000010000",--21 SHR R29 R29 16 R29=000i..skey(79 downto 64)     remove sign extension
"00000011110111011111000000010000",--22 ADD,R30 R29,R30 R30=skey(95 downto 64)
"00100000000111100000000000011100",--23 SW RO, R30 offset M[0+offset3]<-R30
"00000100000111100000000000000000",--24 ******ADDI,RO,R30,skey(127 downto 112) R30=XXX..skey(127 downto 112)
"00010111110111100000000000010000",--25 SHL R30 R30 16 R30=skey(127 downto 112)...000
"00000100000111010000000000000000",--26******* ADDI,RO,R29,SKEY(111 downto 96) R29=XXXi..skey(111 downto 96)
"00010111101111010000000000010000",--27 SHL R29 R29 16 R29=skey(111 downto 96)i000     remove sign extension
"00011101101111010000000000010000",--28 SHR R29 R29 16 R29=000i..skey(111 downto 96)     remove sign extension
"00000011110111011111000000010000",--29 ADD,R30 R29,R30 R30=skey(127 downto 96)
"00100000000111100000000000011101",--30 SW RO, R30 offset M[0+offset4]<-R30
```

```
"00000100000101010000000000000100",--31 ADDI,RO,R21,4 R21=4
"00000100000101100000000000011010",--32 ADDI RO R22,26 R22=26
"00000100000101110000000001001110",--33 ADDI R0 R23,78 R23=78
"00000000000000000001000000010000",--34 ADD RO RO R4 R4=0
"00000000000000000101000000010000",--35 ADD RO RO R5 R5=0
"00011100010010000000000000000000",--36 R8<-MEM[R2+IMM_S) S[i]
"00011100001010010000000000011010",--37 R9<-MEM(R1+IMM_l) L[j]
"00000000100001010011000000010000",--38 ADD R4 R5,R6  A+B
"00000001000001100101000000010000",--39 ADD R8 R6 R10 A+B+S[i]

"00000000000000000110000000010000",--40 ADD RO RO R12

"00010101010011010000000000000011",--41 SHL R10 R13 3
"00011001010011100000000000011101",--42 SHR R10 R14 29
"00000001101011100101000000010000",--43 ADD R13 R14 R10  rotate 1 bit of R10, do it 3 times


"00100000010010100000000000000000",--44 SW R2 R10 IMM_S branch here update S[i]
"00000000000010100010000000010000",--45 add RO R10 R4 put value of A into R4
"00000000100001010011000000010000",--46 add R4 R5 R6 A + B
"00000001001001100101000000010000",--47 add R9 R6 R10 R10 = L[i] + A + B
"00001100110010110000000000011111",--48 ANDI R6 R11 00....11111

-- 1
"00000100000111100000000000000001",     --49    Addi    0(R00)  counter(R30)           1
"00101101011111100000000000000100",     --50    Bne     Rotator(R11)    counter(R30)           5
"00010101010011010000000000000001",--51 SHL R10 R13 1
"00011001010011100000000000011111",--52 SHR R10 R14 31
"00000001101011100101000000010000",--53 ADD R13 R14 R10
"00110000000000000000000011101011",     --54    Jmp     ½×
```

-
- *This part is used to compare the value of A+B with 1 to 31*
-

```
--31
"00000100000111100000000000011111",     --229   Addi    0(R00)  counter(R30)           1
"00101101011111100000000000000100",     --230   Bne     Rotator(R11)    counter(R30)           5
"00010101010011010000000000011111",--231 SHL R10 R13 31
"00011001010011100000000000000001",--232 SHR R10 R14 1
"00000001101011100101000000010000",--233 ADD R13 R14 R10
"00110000000000000000000011101011",     --234   Jmp     ½×



"00100000001010100000000000011010",--235 SW R1 R10 IMML branch here  update L[i]
"00000000000010100010100000010000",--236 ADD RO R10 R5
"00000100001000010000000000000001",--237 ADDI R1 R1 1
"00000100010000100000000000000001",--238 ADDI R2 R2 1
"00000100011000110000000000000001",--239 ADDI R3 R3 1
"00101100001101010000000000000001",--240 BNE R1 R21 IMM
"00000000000000000000100000010000",--241 ADD RO RO R1
"00101100010101100000000000000001",--242 BNE R2 R22 IMM
"00000000000000000001000000010000",--243 ADD RO RO R2
"00101000011101110000000000000001",--244 BEQ R3 R22 IMM
"00110000000000000000000000100100",--245 JMP RETURN
"00000000000000000011000000010000",--246 ADD RO RO R3
```

```
"00011100000000110000000000000000",     --247   Lw      0(R00)  S[0](R03)              0(Imm£© Load s[0] to R3
"00011100000001000000000000000001",     --248   Lw      0(R00)  S[1](R04)              1(Imm)  Load s[1] to R4
"00000100000001010000000000000000",     --249   Addi    0(R00)  Temp1(R05)             Din(63 downto 48)
"00010100101001010000000000010000",     --250   Shl     Temp1(R05)      Temp1(R05)             16      Load Din(63 downto 48)
"00000100000001100000000000000000",     --251   Addi    0(R00)  Temp2(R06)             Din(47 downto 32)
"00010100110001100000000000010000",     --252   Shl     Temp2(R06)      Temp2(R06)             16
"00011000110001100000000000010000",     --253   Shr     Temp2(R06)      Temp2(R06)             16      Load Din(47 downto 32)
"00000000101001100000100000010000",     --254   Add     Temp1(R05)      Temp2(R06)     A(R01)          Load A
"00000100000001010000000000000000",     --255   Addi    0(R00)  Temp1(R05)             Din(31 downto 16)
"00010100101001010000000000010000",     --256   Shl     Temp1(R05)      Temp1(R05)             16      Load Din(31 downto 16)
"00000100000001100000000000000000",     --257   Addi    0(R00)  Temp2(R06)             Din(15 downto 0)
"00010100110001100000000000010000",     --258   Shl     Temp2(R06)      Temp2(R06)             16
"00011000110001100000000000010000",     --259   Shr     Temp2(R06)      Temp2(R06)             16      Load Din(15 downto 0)
"00000000101001100001000000010000",     --260   Add     Temp1(R05)      Temp2(R06)     B(R02)          Load A

----------------SELECTION-------------


"00000100000010110000000000000000",     --261   Addi    0(R00)  Mode(R11)                      0--encode  1-- decode
"00100100000010110000000110001100",     --262   Blt     0(R00)  Mode(R11)
"00000000001000110000100000010000",     --263   Add     A(R01)  S[0](R03)       A(R01)         A+s[0]
"00000000010001000001000000010000",     --264   Add     B(R02)  S[1](R04)       B(R02)         B+s[1]
"00000000000000000010100000010000",     --265   Add     0(R00)  0(R00)  I(R05)          Initialize I
"00000100000010100000000000011000",     --266   Addi    0(R00)  24(R10)         24
"00101000101010100000000110000100",     --267   Beq     I(R05)  24(R10)         388     ±û

 "00000100101001010000000000000010",     --268   Addi    I(R05)  I(R05)          2       I=I+2
 "00011100101000110000000000000000",     --269   Lw      I(R05)  S[2*i](R03)             0       Load s[2*i]
 "00011100101001000000000000000001",     --270   Lw      I(R05)  S[2*i+1](R04)           1       Load s[2*i+1]
 "00000000001000100011000000010010",     --271   And     A(R01)  B(R02)  C(R06)
 "00000000110000000011000000010100",     --272   Nor     C(R06)  0(R00)  C(R06)
 "00000000001000100011100000010011",     --273   Or      A(R01)  B(R02)  D(R07)
 "00000000110001110000100000010010",     --274   And     C(R06)  D(R07)  A(R01)          A Xor B
 "00001100010001110000000000011111",     --275   Andi    B(R02)  Rotator(R07)            0000i11111      B(4 downto0)
 "00000100000010110000000000000001",     --276   Addi    0(R00)  counter(R11)            1
 "00101100111010110000000000000100",     --277   Bne     Rotator(R07)    counter(R11)           5
 "00010100001010000000000000000001",     --278   Shl     A(R01)  Temp_left(R08)          1
 "00011000001010010000000000011111",     --279   Shr     A(R01)  Temp_right(R09)         31
 "00000001000010010000100000010000",     --280   Add     Temp_left(R08)  Temp_right(R09) A(R01)          Round rotate A
 "00110000000000000000000111001110",     --281   Jmp                                     ¼×
 "00000101011010110000000000000001",     --282   Addi    counter(R11)    counter(R11)           1
 "00101100111010110000000000000100",     --283   Bne     Rotator(R07)    counter(R11)           4
 "00010100001010000000000000000010",     --284   Shl     A(R01)  Temp_left(R08)          2
 "00011000001010010000000000011110",     --285   Shr     A(R01)  Temp_right(R09)         30
 "00000001000010010000100000010000",     --286   Add     Temp_left(R08)  Temp_right(R09) A(R01)          Round rotate A
 "00110000000000000000000111001110",     --287   Jmp                                     ¼×
 "00000101011010110000000000000001",     --288   Addi    counter(R11)    counter(R11)           1
 "00101100111010110000000000000100",     --289   Bne     Rotator(R07)    counter(R11)           4
 "00010100001010000000000000000011",     --290   Shl     A(R01)  Temp_left(R08)          3
 "00011000001010010000000000011101",     --291   Shr     A(R01)  Temp_right(R09)         29
 "00000001000010010000100000010000",     --292   Add     Temp_left(R08)  Temp_right(R09) A(R01)          Round rotate A
 "00110000000000000000000111001110",     --293   Jmp                                     ¼×
 "00000101011010110000000000000001",     --294   Addi    counter(R11)    counter(R11)           1
 "00101100111010110000000000000100",     --295   Bne     Rotator(R07)    counter(R11)           4
 "00010100001010000000000000000100",     --296   Shl     A(R01)  Temp_left(R08)          4
 "00011000001010010000000000011100",     --297   Shr     A(R01)  Temp_right(R09)         28
 "00000001000010010000100000010000",     --298   Add     Temp_left(R08)  Temp_right(R09) A(R01)          Round rotate A
 "00110000000000000000000111001110",     --299   Jmp                                     ¼×
 "00000101011010110000000000000001",     --300   Addi    counter(R11)    counter(R11)           1
 "00101100111010110000000000000100",     --301   Bne     Rotator(R07)    counter(R11)           4
 "00010100001010000000000000000101",     --302   Shl     A(R01)  Temp_left(R08)          5
 "00011000001010010000000000011011",     --303   Shr     A(R01)  Temp_right(R09)         27
 "00000001000010010000100000010000",     --304   Add     Temp_left(R08)  Temp_right(R09) A(R01)          Round rotate A
 "00110000000000000000000111001110",     --305   Jmp                                     ¼×
 "00000101011010110000000000000001",     --306   Addi    counter(R11)    counter(R11)           1
 "00101100111010110000000000000100",     --307   Bne     Rotator(R07)    counter(R11)           4
```

.

.

```
"00101100111010110000000000000100",     --649   Bne     Rotator(R07)    counter(R11)           4
"00010100010010000000000000011111",     --650   Shl     B(R02)  Temp_left(R08)          31
"00011000010010010000000000000001",     --651   Shr     B(R02)  Temp_right(R09)         1
"00000001000010010001000000010000",     --652   Add     Temp_left(R08)  Temp_right(R09) B(R02)          Round rotate A
"00110000000000000000001010001110",     --653   Jmp                                     ÒÒ
"00000000010001000001000000010000",     --654   Add     B(R02)  S[2*i+1](R04)   B(R02)  ÒÒ      B=xxx+s[2*i+1]
"00110000000000000000000100001011",     --655   Jmp                             ±û      Loop back
"00000000001000001111100000010000",     --656   Add     A(R01)  0(R00)  Dec_out(R31)
"00000000010000001111100000010000",     --657   Add     B(R02)  0(R00)  Dec_out(R30)
"11111111111111111111111111111111",     --658   Halt

----------DECODE--------------------
```

```
"00000100000010100000000000011000",     --659   Addi    0(R00)  24(R10)         24      Start of DEC
"00000000000010100010100000010000",     --660   Add     0(R00)  24(R10) I(R05)          Initialize I
"00101000101000000000000110000100",     --661   Beq     I(R05)  0(R00)          388     ±û
"00011100101000110000000000000000",     --662   Lw      I(R05)  S[2*i](R03)             0       Load s[2*i]
"00011100101001000000000000000001",     --663   Lw      I(R05)  S[2*i+1](R04)           1       Load s[2*i+1]
"00001000101001010000000000000010",     --664   Subi    I(R05)  I(R05)          2       I=I-2
"00000000010001000001000000010001",     --665   Sub     B(R02)  S[2*i+1](R04)   B(R02)           B=xxx-s[2*i+1]
"00001100001001110000000000011111",     --666   Andi    A(R01)  Rotator(R07)            0000i11111      A(4 downto0)
"00000100000010110000000000000001",     --667   Addi    0(R00)  counter(R11)            1
"00101100111010110000000000000100",     --668   Bne     Rotator(R07)    counter(R11)            4
"00011000010010000000000000000001",     --669   Shr     B(R02)  Temp_left(R08)          1
"00010100010010010000000000011111",     --670   Shl     B(R02)  Temp_right(R09)         31
"00000001000010010001000000010000",     --671   Add     Temp_left(R08)  Temp_right(R09) B(R02)          Round rotate B
"00110000000000000000001101010101",     --672   Jmp                                     ¶i
"00000101011010110000000000000001",     --673   Addi    counter(R11)    counter(R11)            1
"00101100111010110000000000000100",     --674   Bne     Rotator(R07)    counter(R11)            4
"00011000010010000000000000000010",     --675   Shr     B(R02)  Temp_left(R08)          2
"00010100010010010000000000011110",     --676   Shl     B(R02)  Temp_right(R09)         30
"00000001000010010001000000010000",     --677   Add     Temp_left(R08)  Temp_right(R09) B(R02)          Round rotate B
"00110000000000000000001101010101",     --678   Jmp                                     ¶i
"00000101011010110000000000000001",     --679   Addi    counter(R11)    counter(R11)            1
"00101100111010110000000000000100",     --680   Bne     Rotator(R07)    counter(R11)            4
"00011000010010000000000000000011",     --681   Shr     B(R02)  Temp_left(R08)          3
"00010100010010010000000000011101",     --682   Shl     B(R02)  Temp_right(R09)         29
"00000001000010010001000000010000",     --683   Add     Temp_left(R08)  Temp_right(R09) B(R02)          Round rotate B
"00110000000000000000001101010101",     --684   Jmp                                     ¶i
"00000101011010110000000000000001",     --685   Addi    counter(R11)    counter(R11)            1
"00101100111010110000000000000100",     --686   Bne     Rotator(R07)    counter(R11)            4
"00011000010010000000000000000100",     --687   Shr     B(R02)  Temp_left(R08)          4
"00010100010010010000000000011100",     --688   Shl     B(R02)  Temp_right(R09)         28
"00000001000010010001000000010000",     --689   Add     Temp_left(R08)  Temp_right(R09) B(R02)          Round rotate B
"00110000000000000000001101010101",     --690   Jmp                                     ¶i
"00000101011010110000000000000001",     --691   Addi    counter(R11)    counter(R11)            1
"00101100111010110000000000000100",     --692   Bne     Rotator(R07)    counter(R11)            4
"00011000010010000000000000000101",     --693   Shr     B(R02)  Temp_left(R08)          5
"00010100010010010000000000011011",     --694   Shl     B(R02)  Temp_right(R09)         27
```

.

```
"00000101011010110000000000000001",     --1039  Addi    counter(R11)    counter(R11)            1
"00101100111010110000000000000100",     --1040  Bne     Rotator(R07)    counter(R11)            4
"00011000001010000000000000011111",     --1041  Shr     A(R01)  Temp_left(R08)          31
"00010100001010010000000000000001",     --1042  Shl     A(R01)  Temp_right(R09)         1
"00000001000010010000100000010000",     --1043  Add     Temp_left(R08)  Temp_right(R09) A(R01)          Round rotate A
"00110000000000000000010000010101",     --1044  Jmp                                     Îì
"00000000001000100011000000010010",     --1045  And     A(R01)  B(R02)  C(R06)  Îì
"00000000110000000011000000010100",     --1046  Nor     C(R06)  0(R00)  C(R06)
"00000000001000100011100000010011",     --1047  Or      A(R01)  B(R02)  D(R07)
"00000000110001110000100000010010",     --1048  And     C(R06)  D(R07)  A(R01)          A = A Xor B
"00110000000000000000010010010101",     --1049  Jmp                             ±û      Outer Loop back
"00011100101000110000000000000000",     --1050  Lw      I(R05)  S[0](R03)               0       Load s[0]
"00011100101001000000000000000001",     --1051  Lw      I(R05)  S[1](R04)               1       Load s[1]
"00000000001000110000100000010001",     --1052  Sub     A(R01)  S[0](R03)       A(R01)          A=A-s[0]
"00000000010001000001000000010001",     --1053  Sub     B(R02)  S[1](R04)       B(R02)           B=B-s[1]
"00000000001000001110100000010000",     --1054  Add     A(R01)  0(R00)  Dec_out(R29)
"00000000010000001110000000010000",     --1055  Add     B(R02)  0(R00)  Dec_out(R28)
"11111111111111111111111111111111"      --1056  Halt
```

We firstly import the ukey into the data memory by the addi operation, and store it in data memory 26 to 29 as L array. Since the addi operation can only get 16 bits of immediate number, so in order to import 128 bits to 4 32-bits slot, we need to do two addi operations for each slot.

For example, in order to import the ukey[31 downto 0], we first addi ukey[31 downto 16], and store it into a register, then import ukey[15 downto 0], Finally, we shift the ukey[31 downto 16] to left by 16 bits, and then add it to ukey[15 downto 0]. But there is a trick here: when we use addi operation, we actually get the sign extension of the immediate number, so if the ukey[15 downto 0] has MSB '1', then the final addition result will be wrong. The following key expansion operations are regular loops that keep updating S array.

5.2 Enc&Dec:

A select mode is used to determine running encryption or decryption, from line 247 to 262 in instruction. Din is loaded by 16 bit per time before the actual instruction for selection is being executed, since the Initialization for encryption and decryption is same. The output of encryption is storage in R30&R31, and the output of decryption is storage in R28&R29.

5.3 Optimization:

In our previous design, when we want to do

$$B = L[j] = (L[j] + A + B) <<< (A + B);$$

we set the A+B value into a register, say R11, and we set an index register, start in loop with value 0, say R12. Then we use BNE to to compare R11 and R12, if R11 not equal to R12, we shift (L[j] + A + B) to left one bit, then finally jump back to the BNE instruction. But we found that this method could be really time consuming. It cost us 9000 cycles to finish the key expansion.

Then we optimized our algorithm, when we get the A+B value, we compared it with 1 to 31. For example, we put immediate number 1 into R11, and we compared it with A+B with BNE, if equal, shift 1 bit left, if not, jump to the next step BNE and compare it with 2 and so on so forth. Although it takes more instruction memory, it does save us a lot of time, we can finish the key expansion about 4800 cycles.

6. Verification of the Results

6.1 TextIO Verification

To verify the results in a more general range, 1001 cases generated by C code are tested using TextIO. Results are shown in the lower windows as Round Key Success, Encryption Success, and Decryption Success.

Encryption:

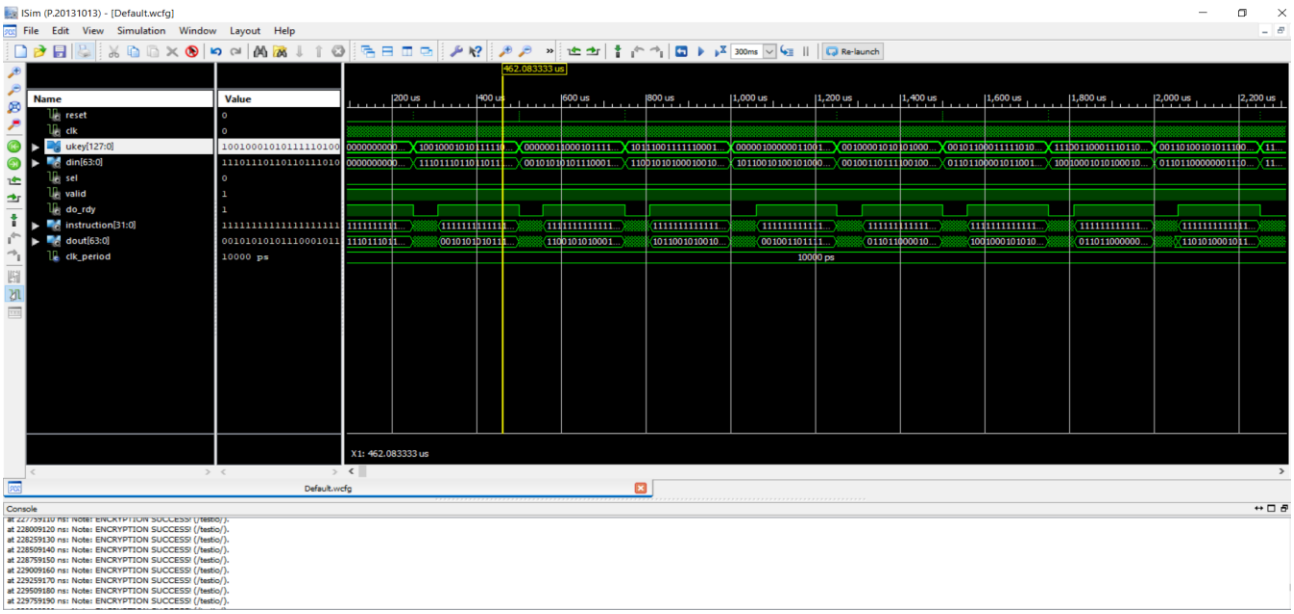


Figure 6.1

Decryption:

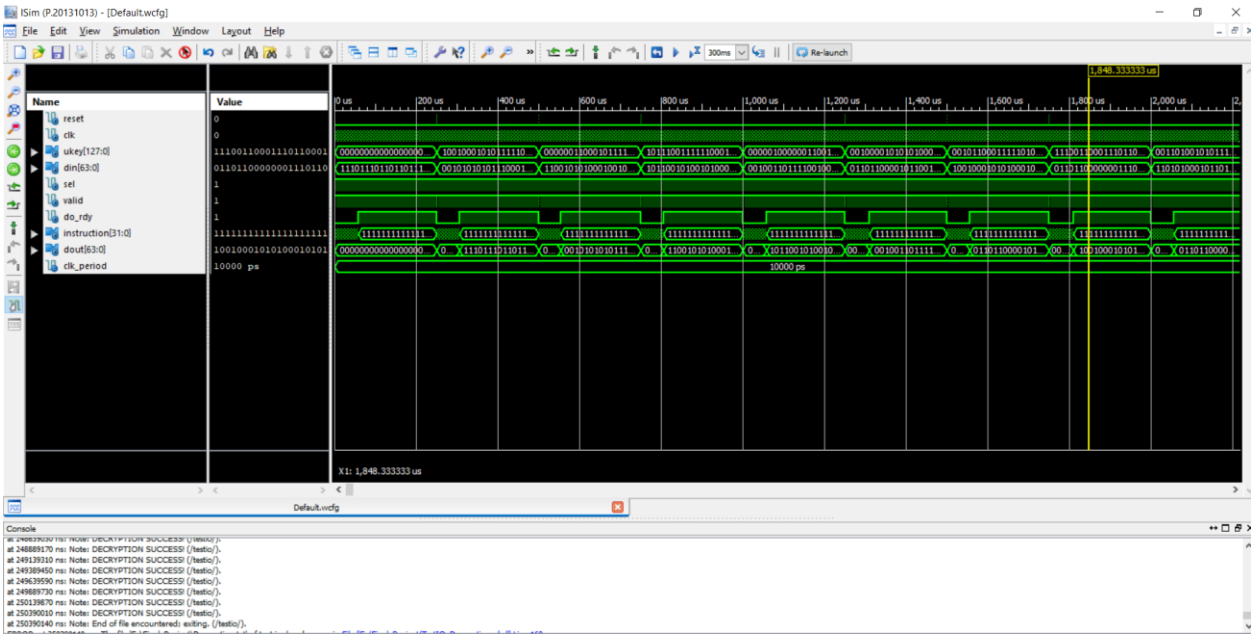


Figure 6.2

Round Key:

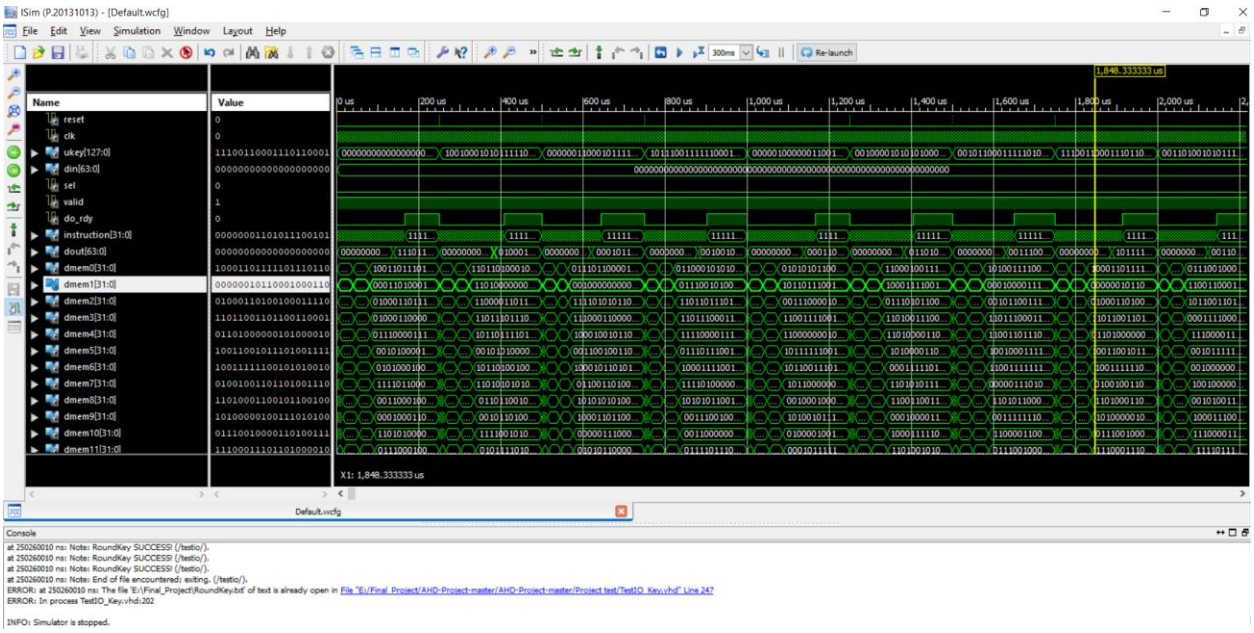


Figure 6.3

6.2 Test Vector for Final Analysis:

Encode Test:

ukey: x”a1b21111 a1b22222 a1b28888 abcd11161”, din: x”dd0000bb 00991111”

skey: d79c5c23 595062a9 243d544f f8403345 442273c1 ce49b445 48f9b1bd e2962dc9 ae01748c 8b021996 2a9a3f93 5a159f09 5b33435f 14c0deae b2f89959 e6ce3921 d2b1be3b 6ceac0c7 87c826ec 70cd4587 edca72f2 eb1327f4 ae7326d0 7f02eb80 4196cb32 7c0738a3

dout: x”6854b2a0 51e26180“

Key Expansion:

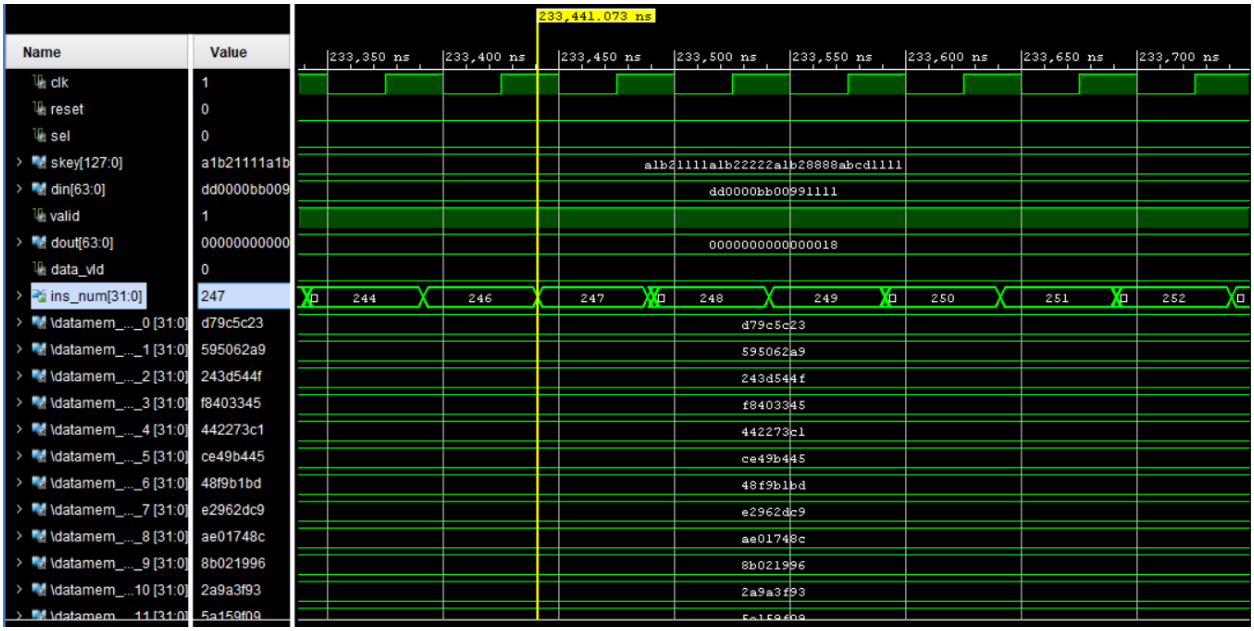


Figure 6.4

Encode:

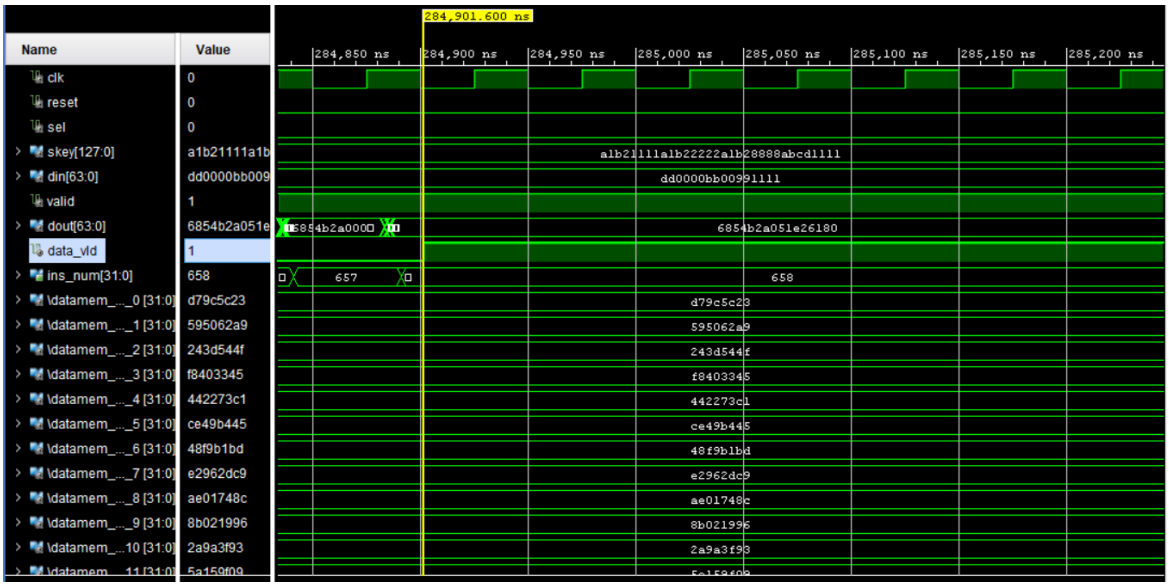


Figure 6.5

Decode Test:

ukey: x”a1b21111 a1b22222 a1b28888 abcd1111”, din: x”6854b2a0 51e26180“

skey: d79c5c23 595062a9 243d544f f8403345 442273c1 ce49b445 48f9b1bd e2962dc9 ae01748c 8b021996 2a9a3f93 5a159f09 5b33435f 14c0deae b2f89959 e6ce3921 d2b1be3b 6ceac0c7 87c826ec 70cd4587 edca72f2 eb1327f4 ae7326d0 7f02eb80 4196cb32 7c0738a3

dout: x”dd0000bb 00991111”

Key Expansion:

Figure 6.6

Decode:

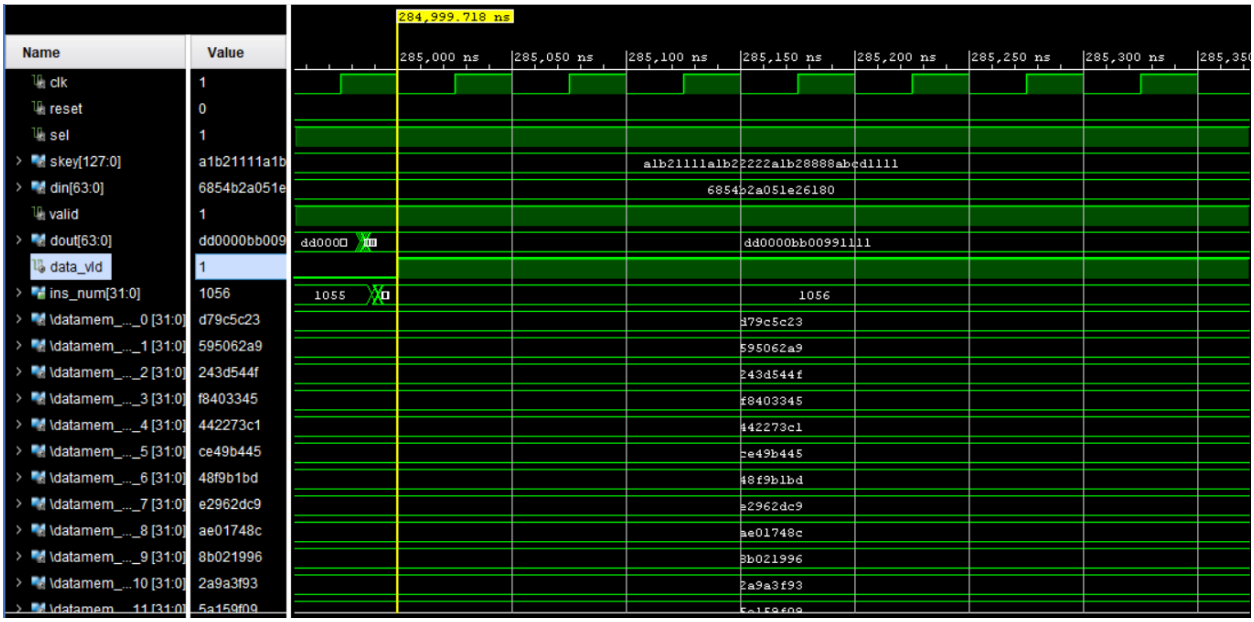


Figure 6.7

The number of cycles for round key generation: 4663 cycles

The number of cycles for encryption: 1030 cycles

The number of cycles for decryption: 1032 cycles

7. Implementation on FPGA

The input of the FPGA board uses 3 buttons, 13 switches.

Buttons:

There are 3 buttons used for input: Button center, Button down and Button right.

Button right: press to +1 to the temporary bit for input user key and Din. The temporary bit will cycle from 0 to F and back to 0. The temporary bit can be select to replace any hex bit in the Input.

Button center: press to reset the whole FPGA.

Button down: in single instruction mode, press to execute the next instruction

Switches:

Switch<0>: used to select the between single instruction mode or full execution mode. sw<0>=0 means full execution mode. sw<0>=1 means single instruction mode.

Switch<1>: used to start the encode or decode process. sw<1>=1 means start encode/decode. sw<1>=0 is when user will input Din and user key.

Switch<2>: used to select to input for Din or Ukey. sw<2>=1 means input for Ukey. sw<2>=0 means input for Din.

Switch<3>: used to select encode or decode operation. sw<3>=1 means decode. sw<3>=0 means encode.

Switch<8 downto 4> : used to select which hex bit of Din or Ukey does user want to input. “00000” means input for the first hex bit. “00001” means input for the second hex bit.

Notice: the Din only has 16 hex bits. So any combination larger than”01111” will not change any bit on Din.

Switch<10>: used to push the temporary bit into the selected hex bit. Push the switch up once to push the temporary hex bit into the selected hex bit.

Switch <15 downto 13>: used to select which 8 hex bit to display on the 7 segment Digital LED display. Combining with the mode selector sw<0>. It can display several value. the combinations are: sw<15 downto 13> & sw<0>:

Full execution mode:

“0000” Din(31 downto 0)

“1000” Din(63 downto 32)

“0100” Dout(31 downto 0)

“1100” Dout(63 downto 0)

“0010” ukey(31 downto 0)

“0110” ukey(63 downto 32)

“1010” ukey(95 downto 64)

“1110” ukey(127 downto 96)

Single instruction mode:

"0001" Current Instruction (32 bit)

"0011" Relevant Rs value (32 bit)

"0101" Relevant Rt value (32 bit)

"0111" Relevant Rd value(32 bit)

"1001" Relevant Immediate number value (32 bit)

"1011" Current instruction number

"1101" Next instruction number

Output:

The 7 segment digital LED display shows hex value of input and output.

LED lights:

led<15>: when led<15> is on, it indicates the Dout is ready.

In single instruction mode, led<6 downto 0> are useful.

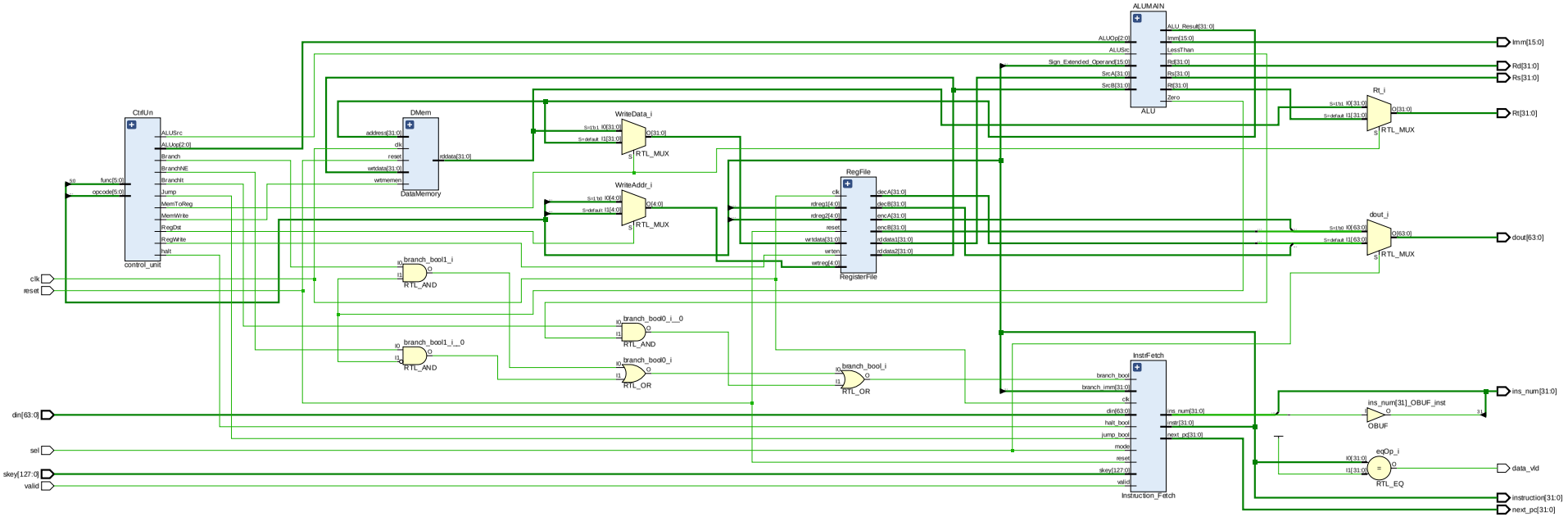
led<3 downto 0>: hard-wired to opcode<3 downto 0>. So that it can distinguish R type instruction and addi, subi, jump, halt etc.

led<6 downto 4>: hard_wired to Function code<2 downto 0>. So when led<3 downto 0> are all off (all 0s). It can distinguish different R type instruction, add, and, or, etc.

Demo Video Address

https://youtu.be/JrCvFFhndmw

Appendix I. Block Diagram (In Next Page)


ALUMAIN
ALU_Result[31:0]
ALUOp[2:0]
imm[15:0]
ALUSrc
LessThan
Sign_Extended_Operand[15:0]
Rd[31:0]
SrcA[31:0]
Rs[31:0]
SrcB[31:0]
Rt[31:0]
Zero
ALU
imm[15:0]
Rd[31:0]
Rs[31:0]
Rt_i
S=1'b1 I0[31:0]
S=default I1[31:0]
O[31:0]
S
RTL_MUX
Rt[31:0]
CtrlUn
ALUSrc
ALUop[2:0]
Branch
BranchNE
BranchHit
Jump
MemToReg
MemWrite
RegDst
RegWrite
halt
func[5:0]
opcode[5:0]
control_unit
DMem
address[31:0]
clk
reset
wrtdata[31:0]
wrtmemen
rddata[31:0]
DataMemory
WriteData_i
S=1'b1 I0[31:0]
S=default I1[31:0]
O[31:0]
S
RTL_MUX
WriteAddr_i
S=1'b1 I0[4:0]
S=default I1[4:0]
O[4:0]
S
RTL_MUX
RegFile
clk
rdreq1[4:0]
rdreq2[4:0]
reset
wrtdata[31:0]
wrten
wrtreq[4:0]
decA[31:0]
decB[31:0]
encA[31:0]
encB[31:0]
rd1data1[31:0]
rd1data2[31:0]
RegisterFile
dout_i
S=1'b1 I0[63:0]
S=default I1[63:0]
O[63:0]
S
RTL_MUX
dout[63:0]
clk
reset
branch_bool1_i
I0
I1
O
RTL_AND
branch_bool0_i__0
I0
I1
O
RTL_AND
branch_bool1_i__0
I0
I1
O
RTL_AND
branch_bool0_i
I0
I1
O
RTL_OR
branch_bool_i
I0
I1
O
RTL_OR
InstrFetch
branch_bool
branch_imm[31:0]
clk
din[63:0]
halt_bool
jump_bool
mode
reset
skey[127:0]
valid
ins_num[31:0]
instr[31:0]
next_pc[31:0]
Instruction_Fetch
ins_num[31]_OBUF_inst
O
OBUF
ins_num[31:0]
eqOp_i
I0[31:0]
I1[31:0]
O
=
RTL_EQ
data_vld
din[63:0]
sel
skey[127:0]
valid
instruction[31:0]
next_pc[31:0]