# AHD Project

# Report


## Team 4

Krutik Shah (kcs431)

Shefali Patel (sdp412)

Monica Ovalekar (mso269)

Akhil Gudge Subramanya (ags586)

- **RC5 Key Expansion and Decryption Implementation in NYU-6463 MIPS Processor on FPGA:**

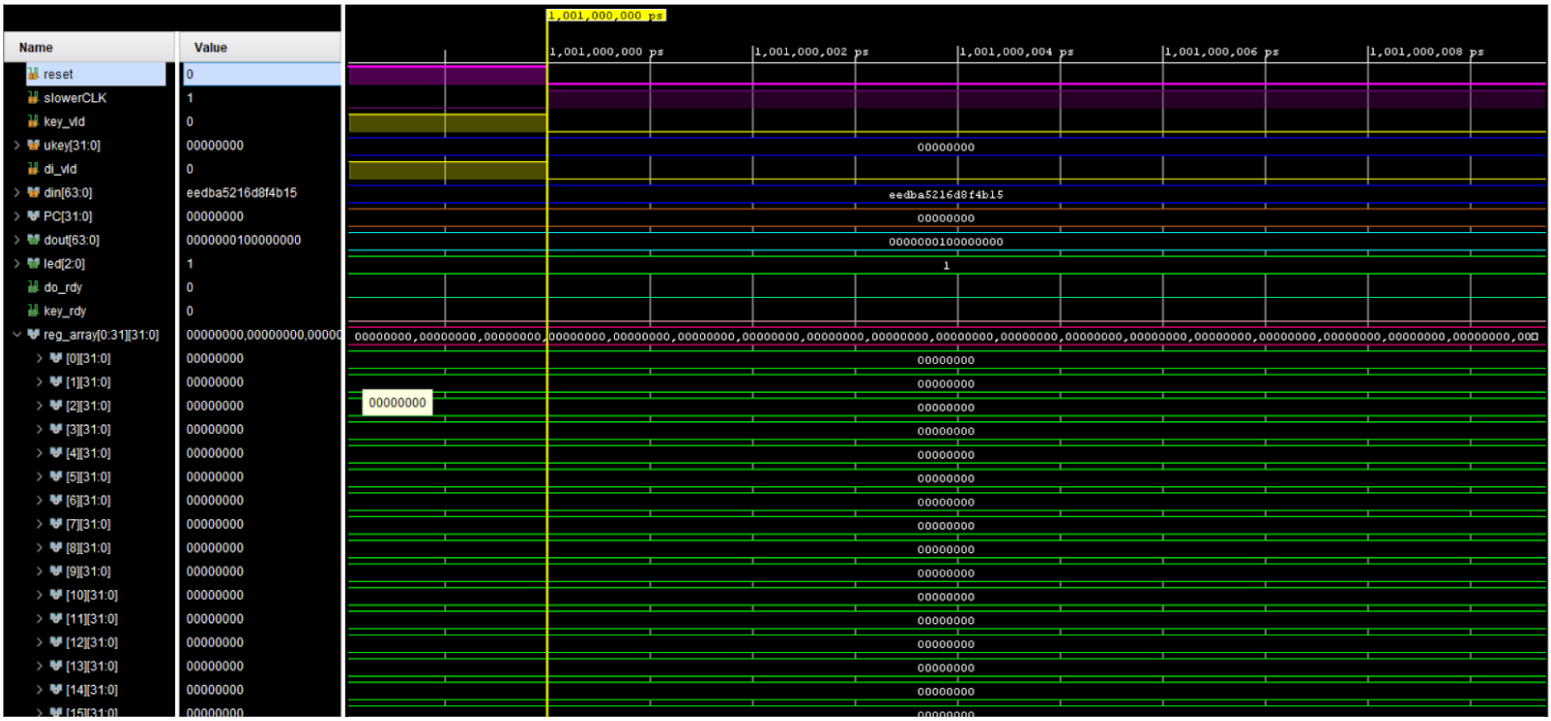
  → YouTube link:
  https://youtu.be/1Y8er05iDG0

Reset is SW[15].
We take 8-bit input at a time, so for 64 bits we input 8-bit data eight times, starting from LSB.
BTNL selects where does the 8-bit input go in 64-bit Din.(It uses a counter)
U16, U17 and V17 LEDs shows the value of counter used to load 64-bit Din.
di_vld is SW[13].
BTND shows the upper 32 bits of dout i.e. A when it is pressed and displays lower 32 bits of dout i.e. B when not pressed.
When SW[14] is '1', it shows dout in every instruction from in step mode. When it is '0', it executes normally.
In the step mode,
When RF write data is the output, J13 LED is on
When Mem write data is the output, N14 LED is on
When Branch address is the output, R18 LED is on

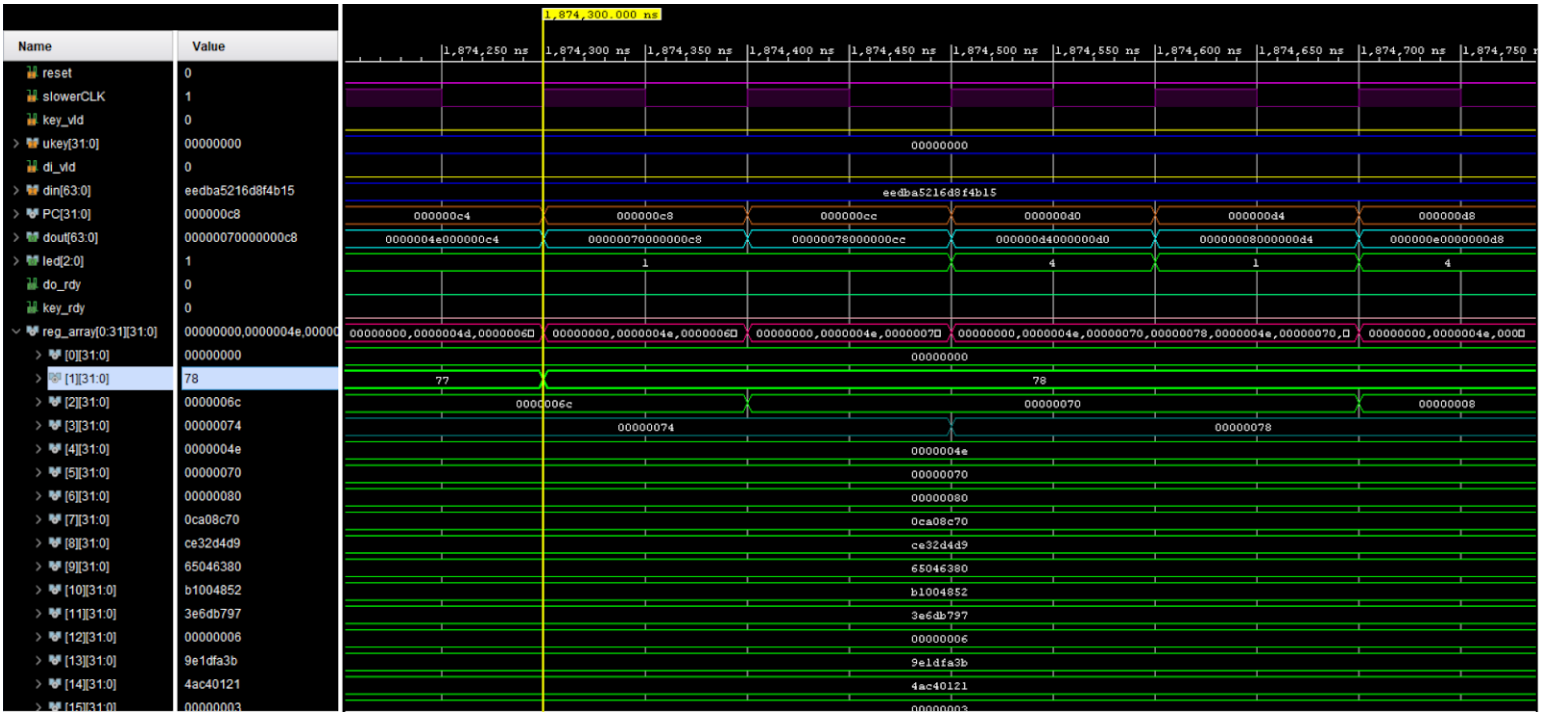
BTNR selects where does the 8-bit input of user key go in 32-bit ukey. (It uses a counter)
H17 LED is on when k_rdy is 1. When BTNU is pressed, k_vld is 1.
K15 LED is on when do_rdy is 1.

## ➢ Functional Diagram without test bench:

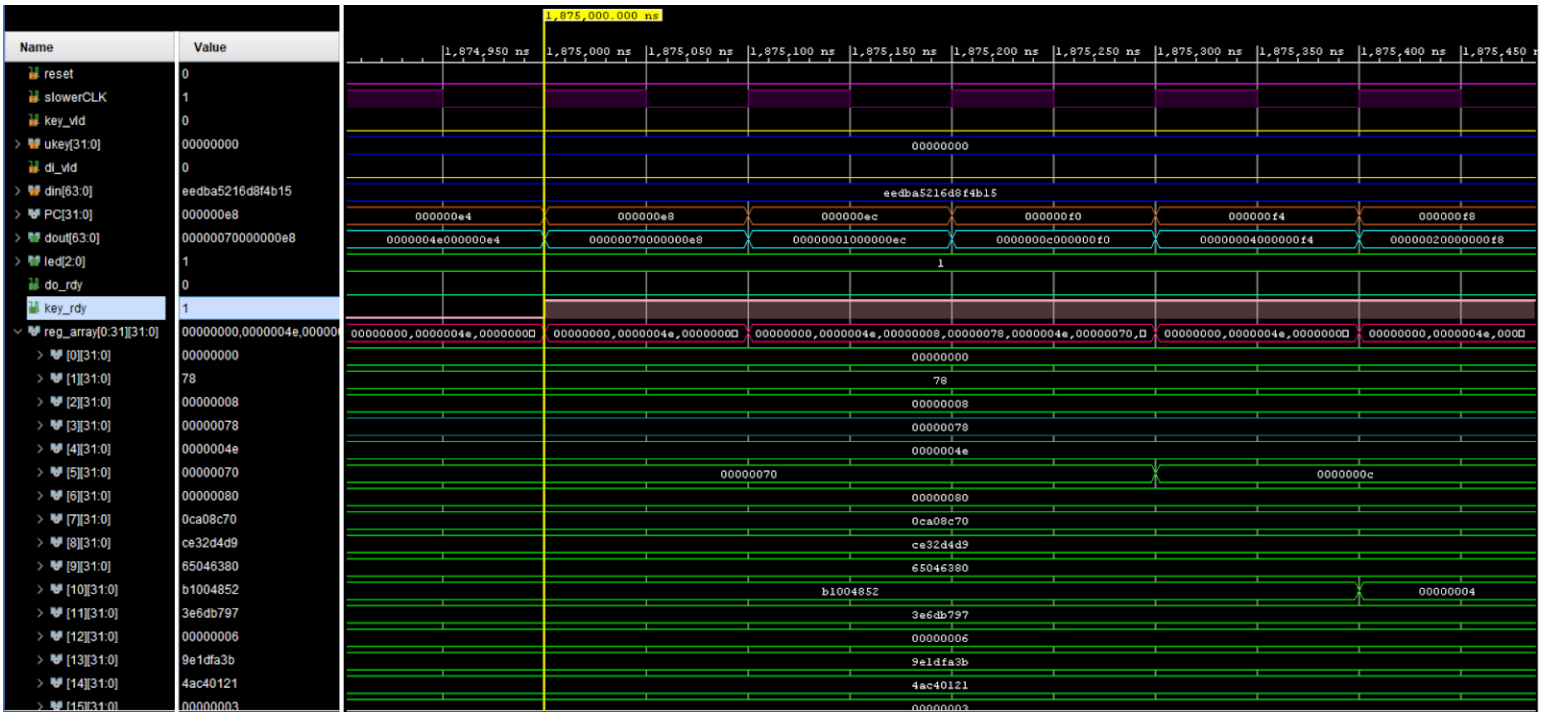
In the below functional simulation results we have done decryption of data = x" **eedba5216d8f4b15**". When the reset is '1', PC is 0 and all registers are "00000000".
Here, PC is the value of current PC. Here, di_vld and key_vld is '1' and we are able to take inputs ukey and din. When the reset is '0', key expansion starts. Dout's lower 32 bits shows the value of PC and upper bits shows ALU result or Branch address or data to be written in the memory according to the type of instructions.



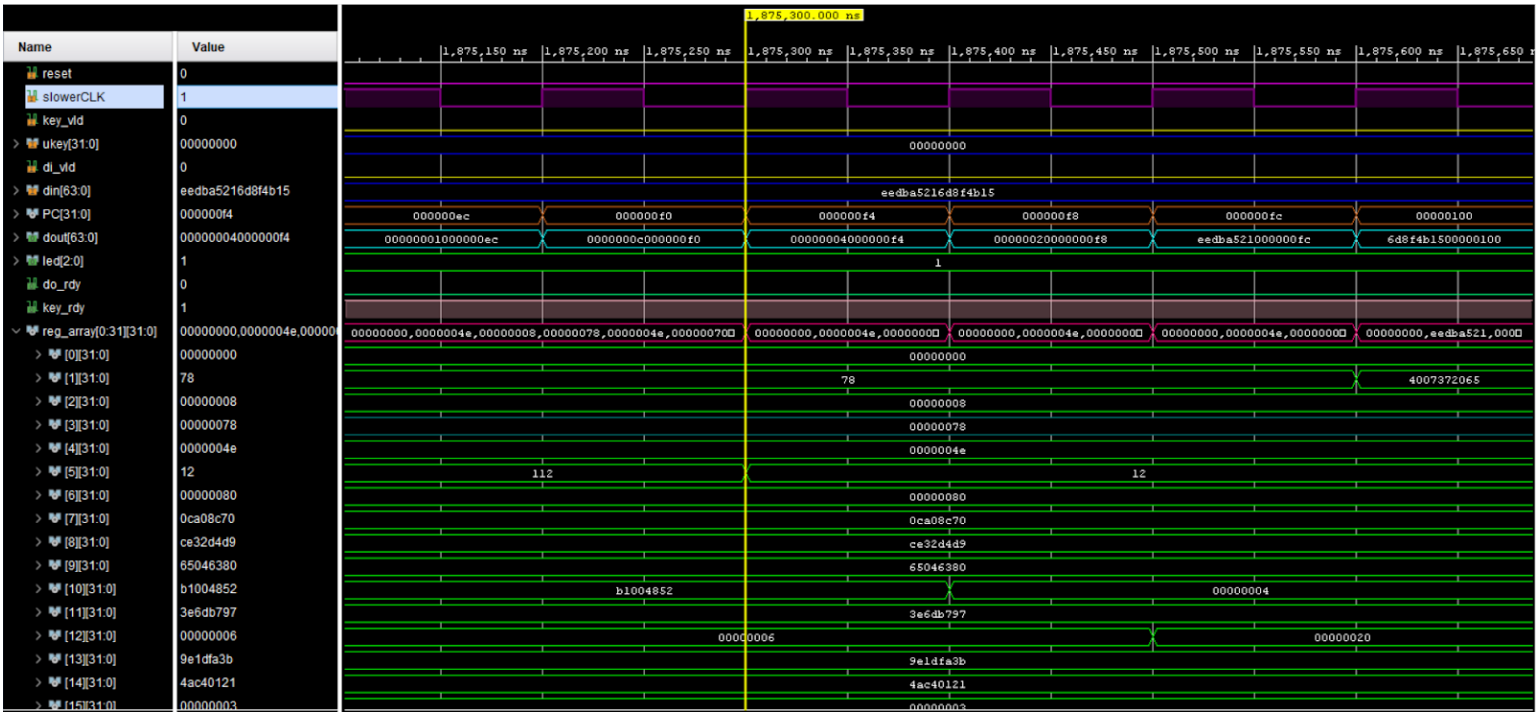
Here, we can see that when the k_cnt which is in R[1] reaches to 78, we get the value of S[25] after all the counts of k in the register R[9]. Which shows that all the values of skey after expansion are generated.



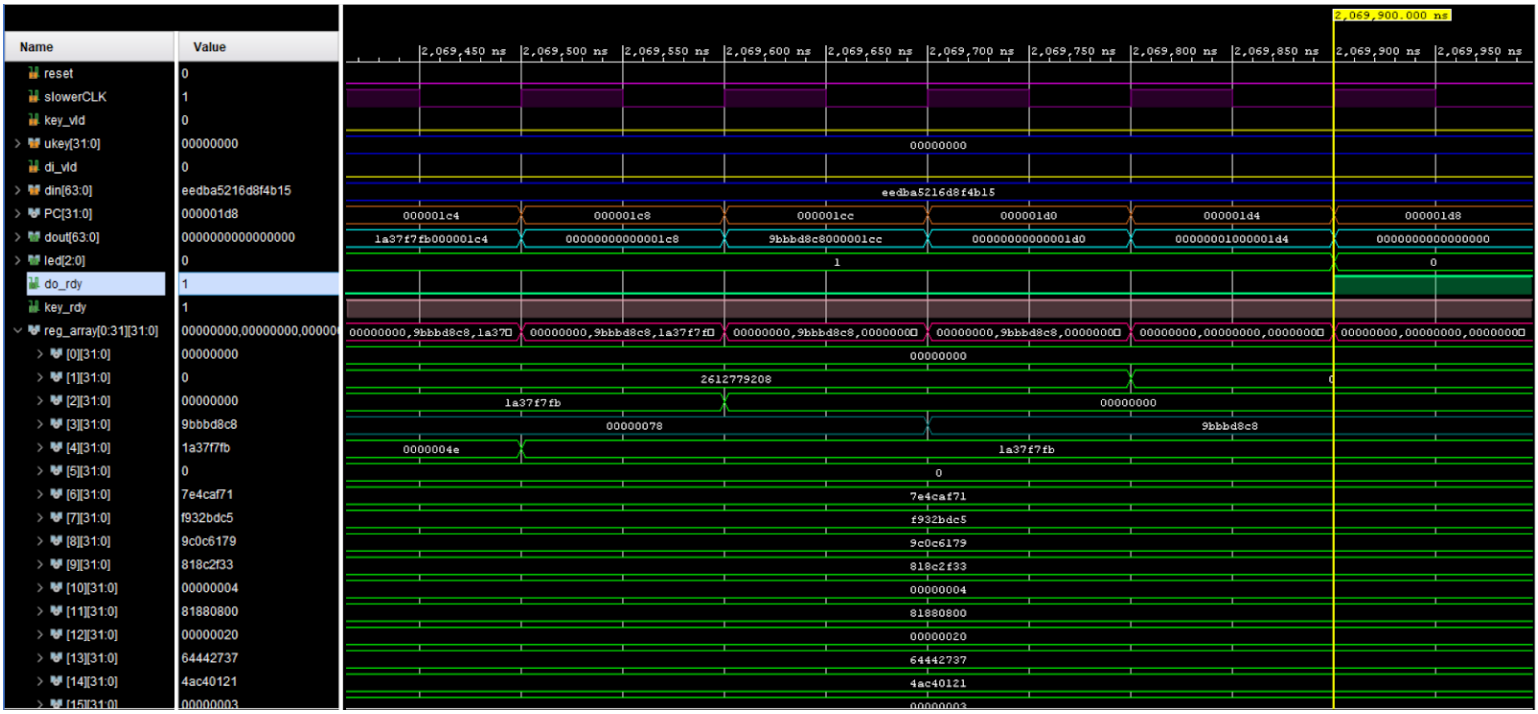

Here, we can see that when the key is ready, key_rdy signal goes high and indicates that key expansion is done.



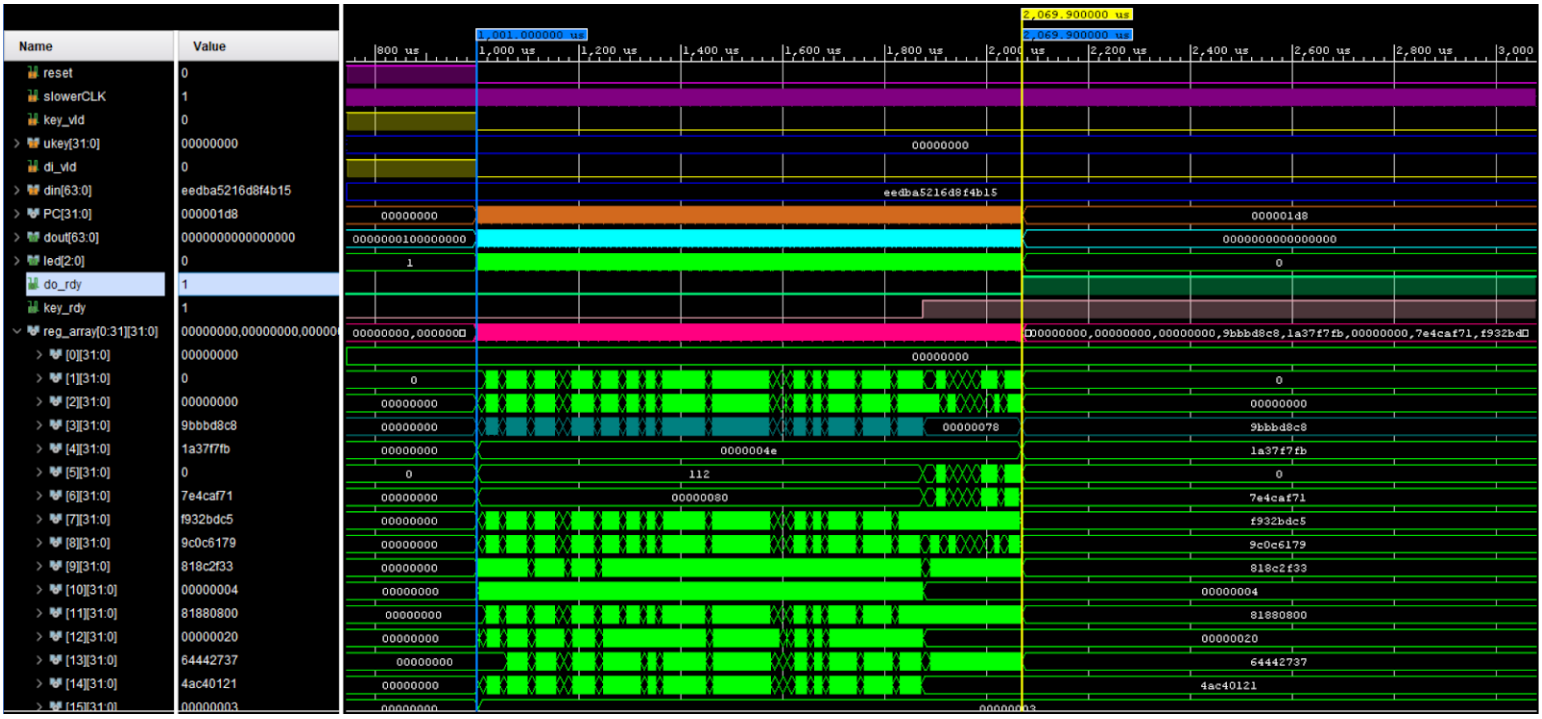
In the below simulation we can see that the din is available which is the encrypted value of x“0000000000000000”.




In the below simulation, we can see that do_rdy is high, that means the decryption is done and decrypted value is available which is x”0000000000000000”.



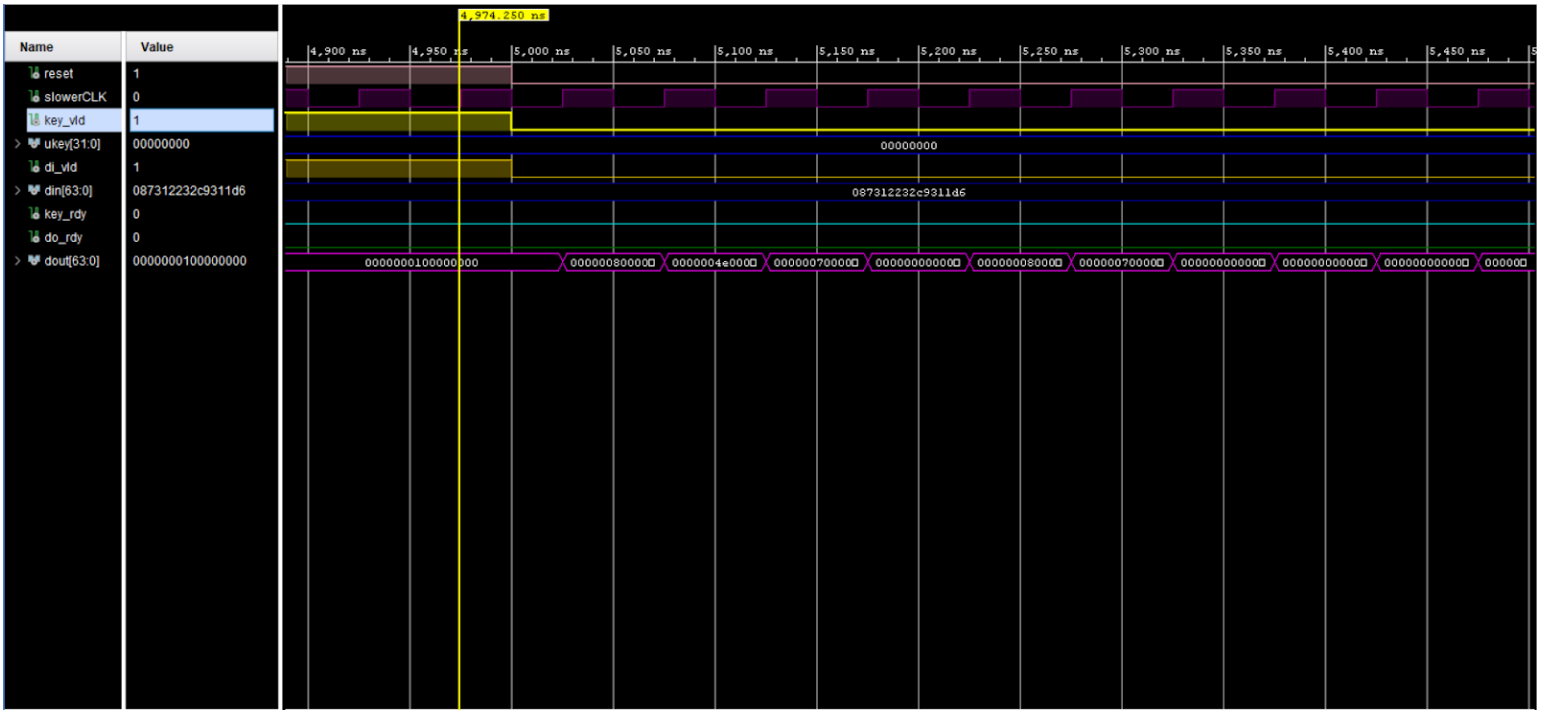
In this simulation result, we can see in how many clock cycles, the whole process was done. Here, we can see that it is done in 10,689 clock cycles.



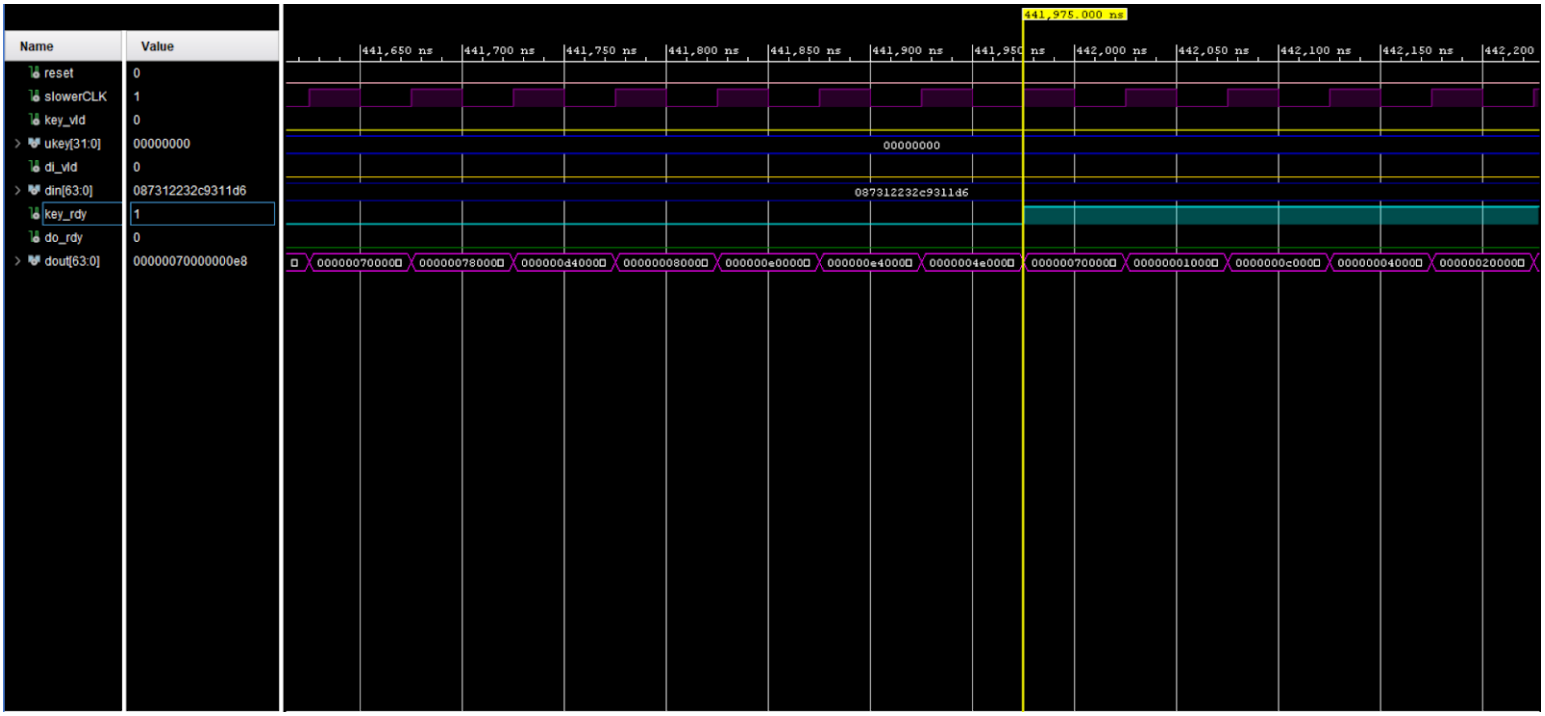
## ➢ Functional Diagram with test bench:

In the below functional simulation results we have done decryption of data = x" **087312232c9311d6**". When the reset is '1', PC is 0 and all registers are "00000000".

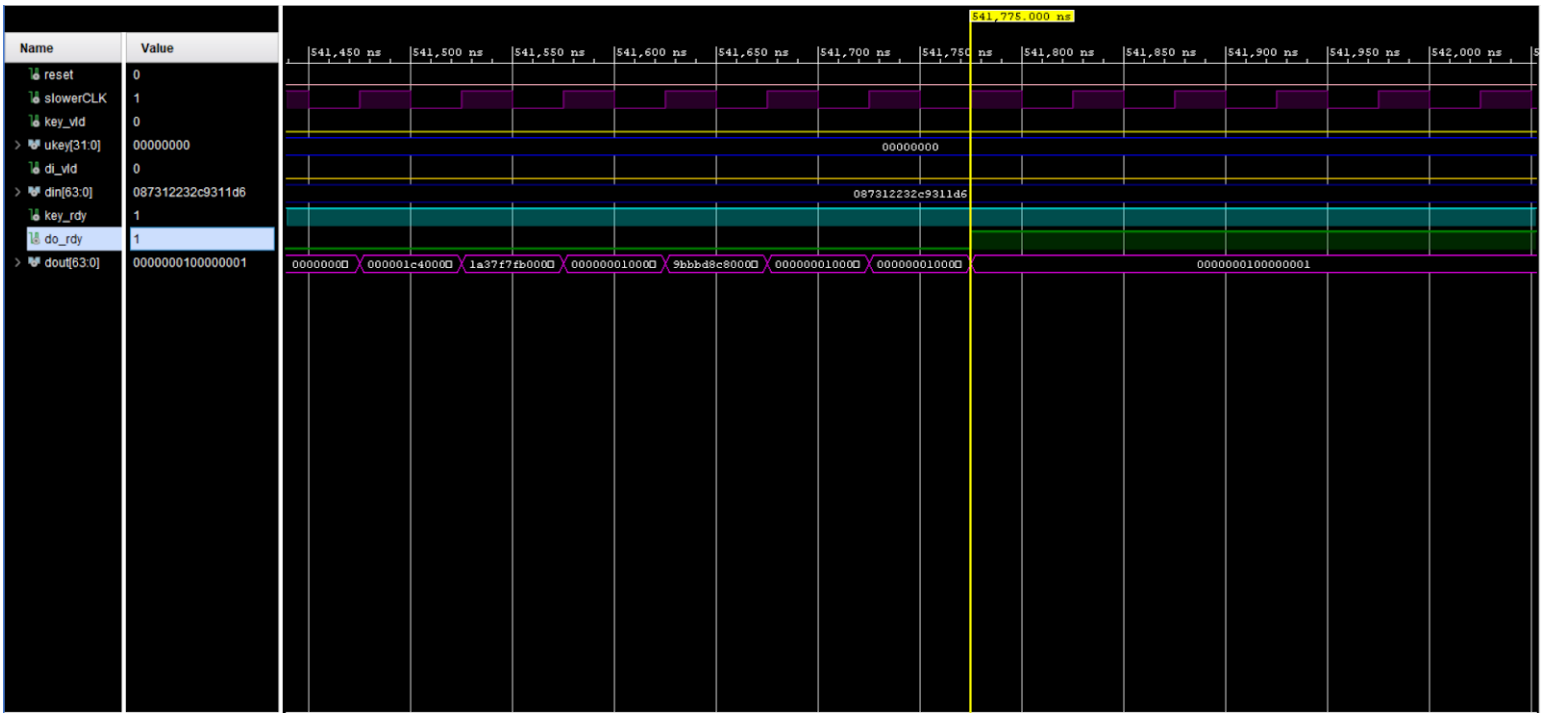
Here, PC is the value of current PC. Here, di_vld and key_vld is '1' and we are able to take inputs ukey and din. When the reset is '0', key expansion starts. Dout's lower 32 bits shows the value of PC and upper bits shows ALU result or Branch address or data to be written in the memory according to the type of instructions.



Here, we can see that when the key is ready, key_rdy signal goes high and indicates that key expansion is done.

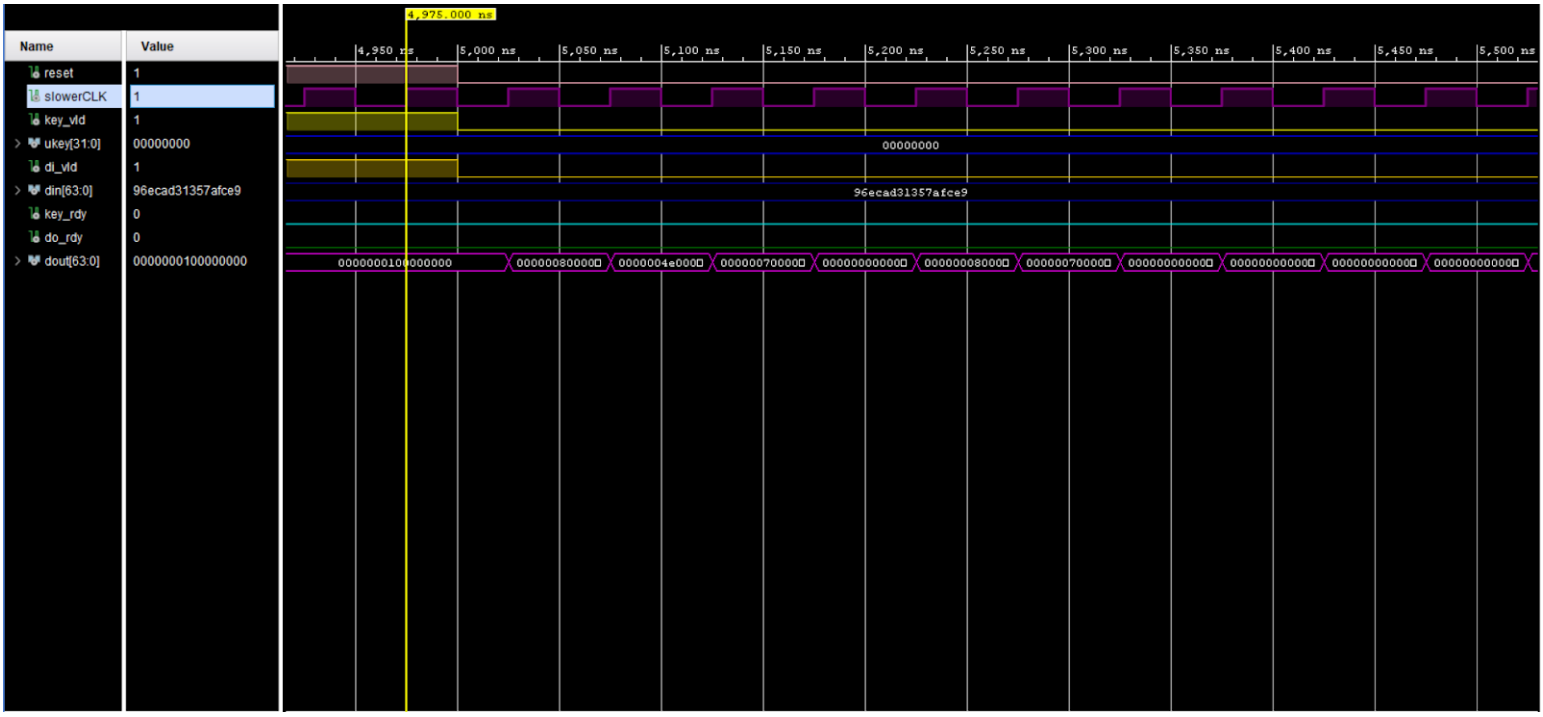



In the below simulation, we can see that do_rdy is high, that means the decryption is done and decrypted value is available which is x"0000000100000001"

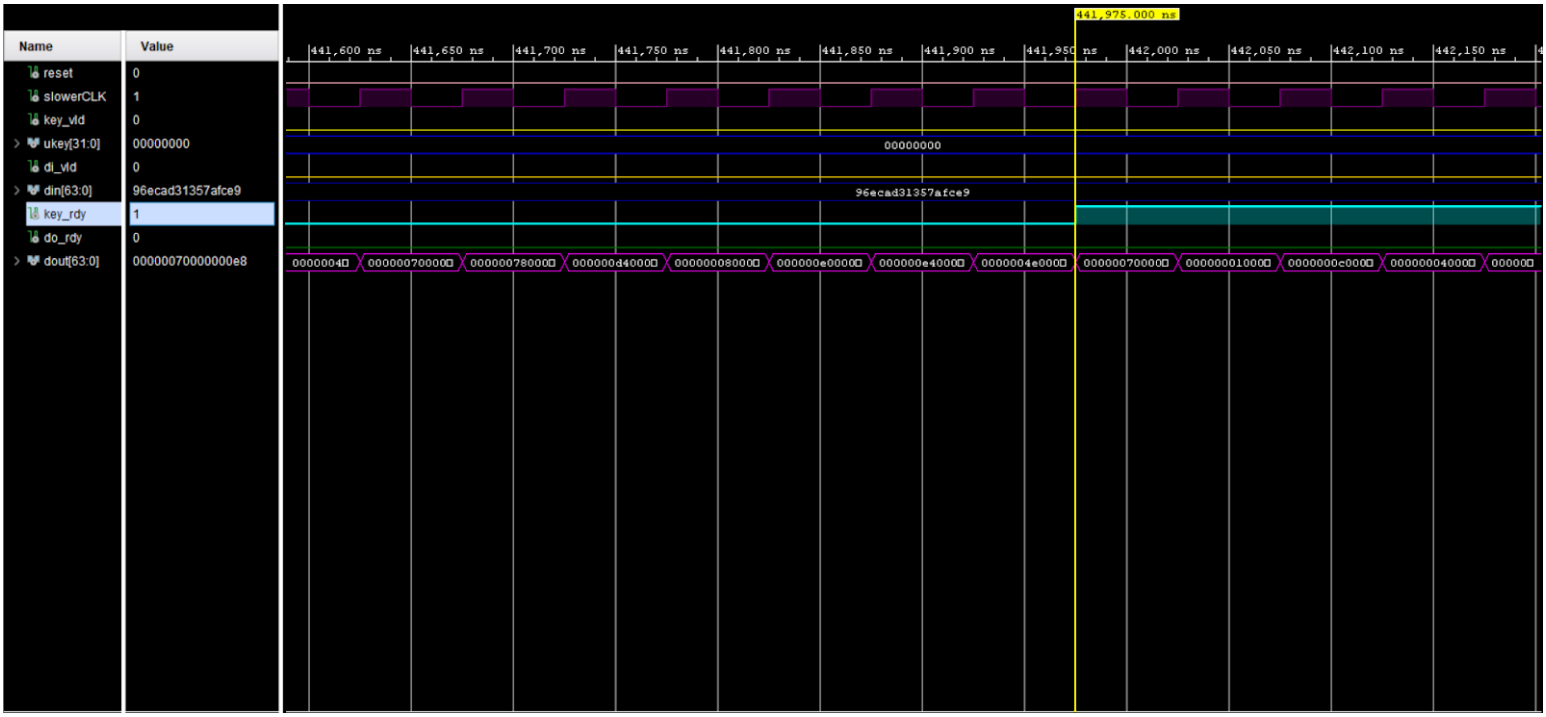


In the below functional simulation results we have done decryption of data = x" **96ecad31357afce9**". When the reset is '1', PC is 0 and all registers are "00000000".

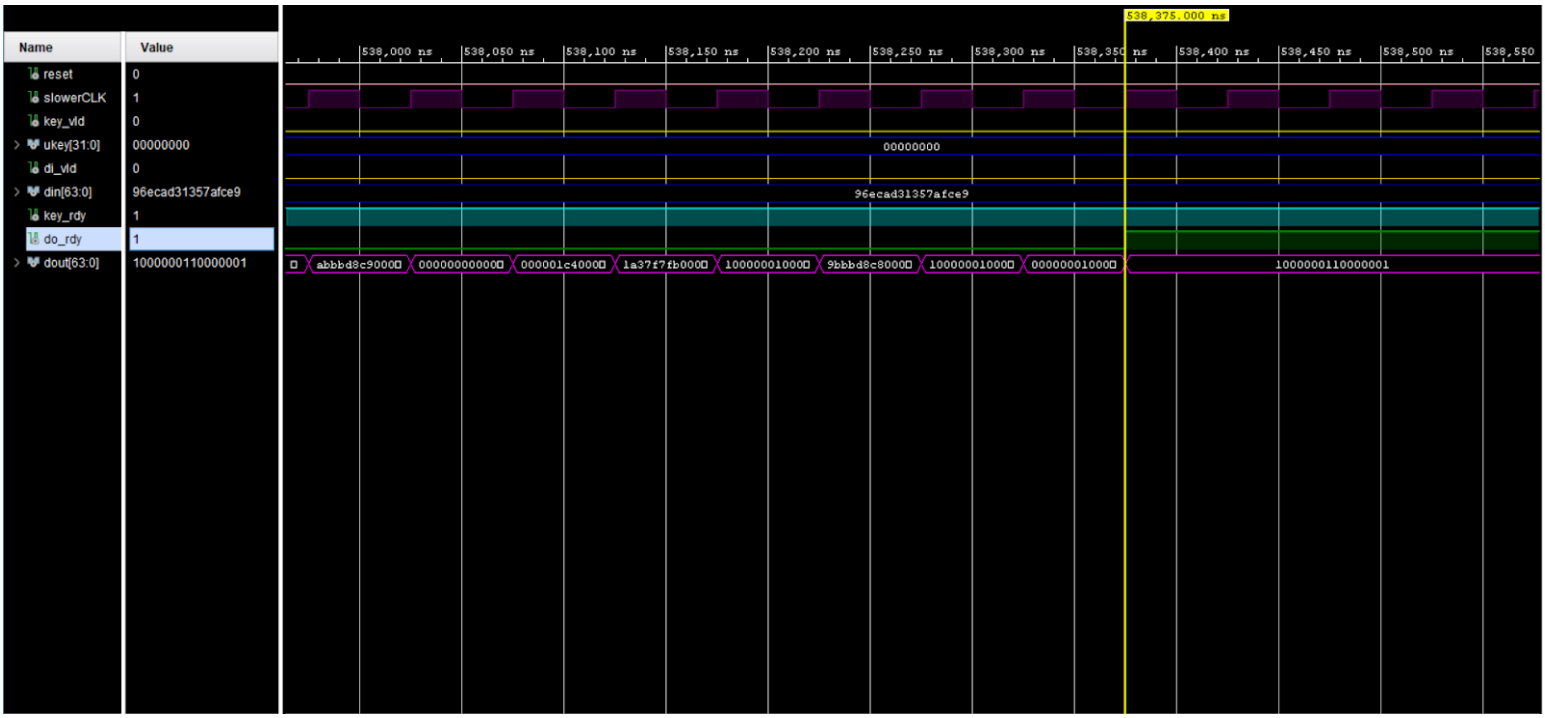
Here, PC is the value of current PC. Here, di_vld and key_vld is '1' and we are able to take inputs ukey and din. When the reset is '0', key expansion starts. Dout's lower 32 bits shows the value of PC and upper bits shows ALU result or Branch address or data to be written in the memory according to the type of instructions.



Here, we can see that when the key is ready, key_rdy signal goes high and indicates that key expansion is done.

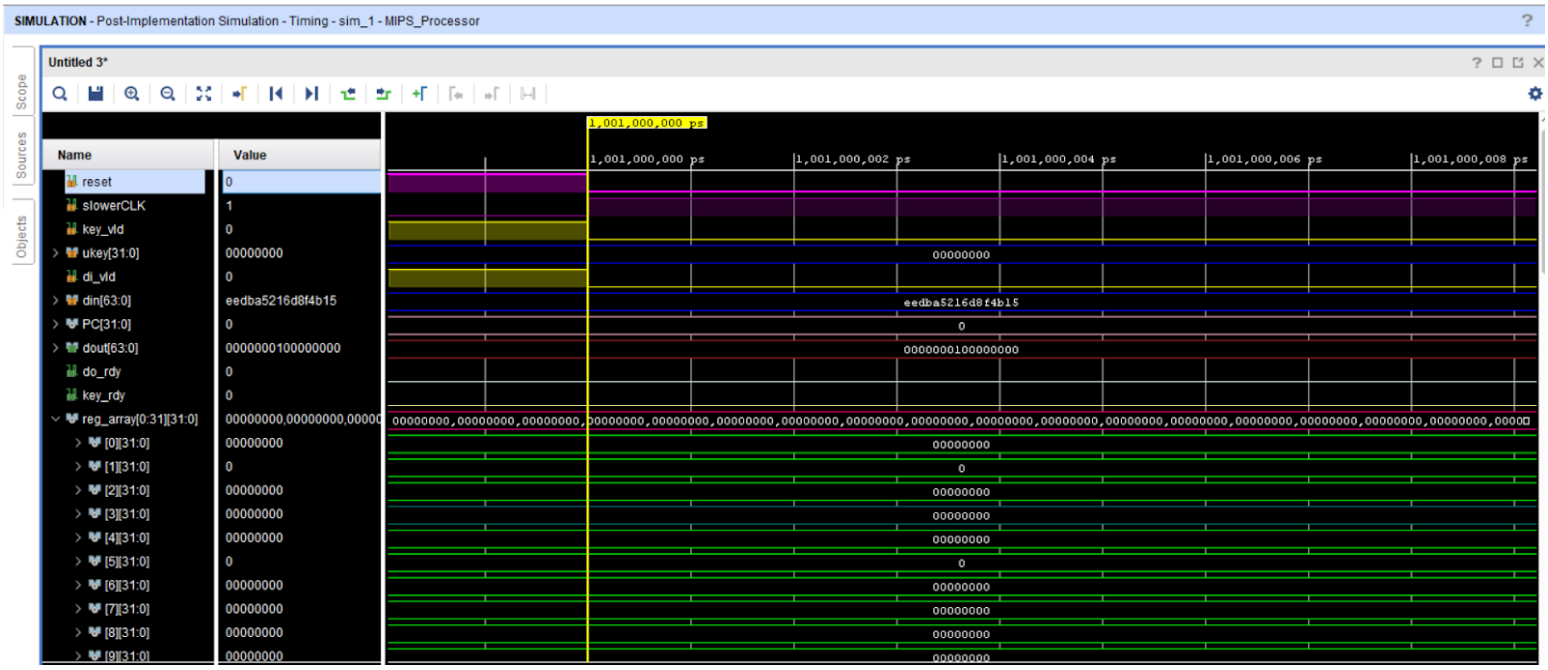



In the below simulation, we can see that do_rdy is high, that means the decryption is done and decrypted value is available which is x"1000000110000001"

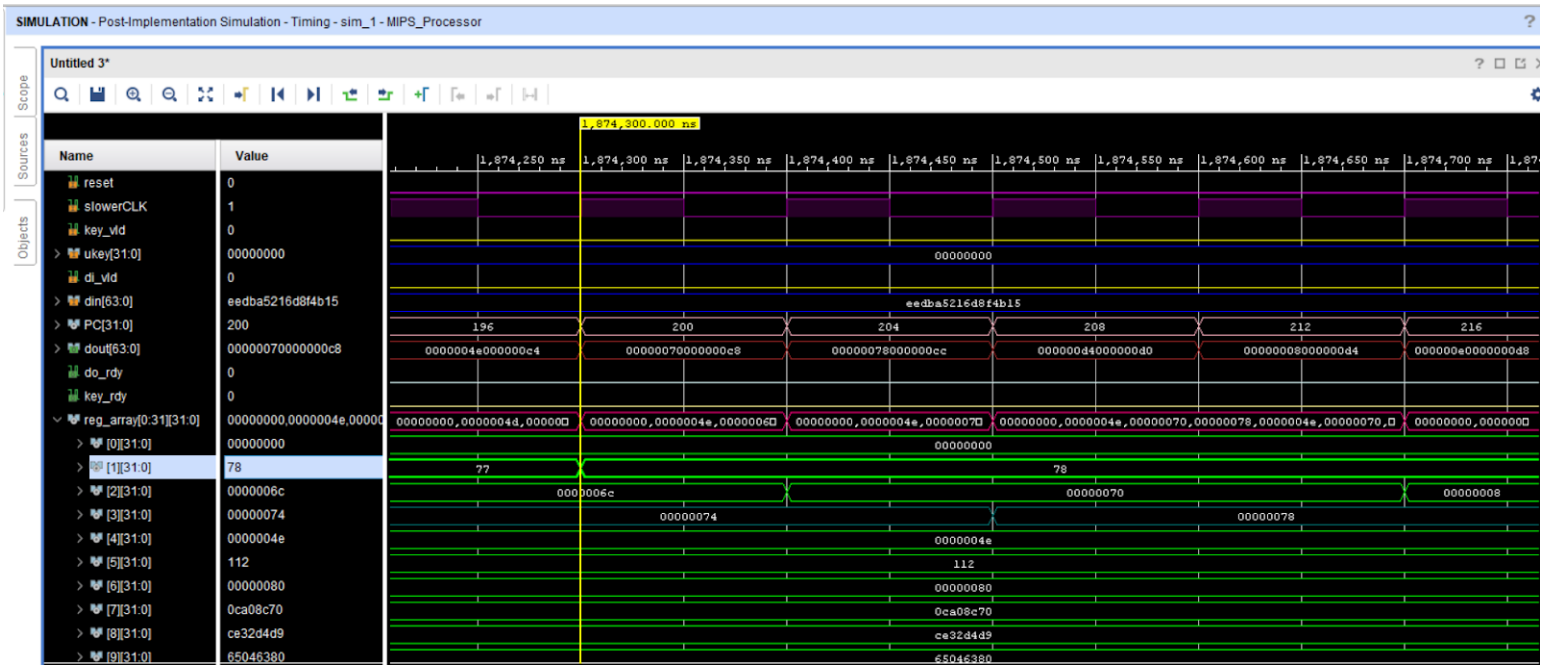


## ➢ Timing Diagram without test bench:

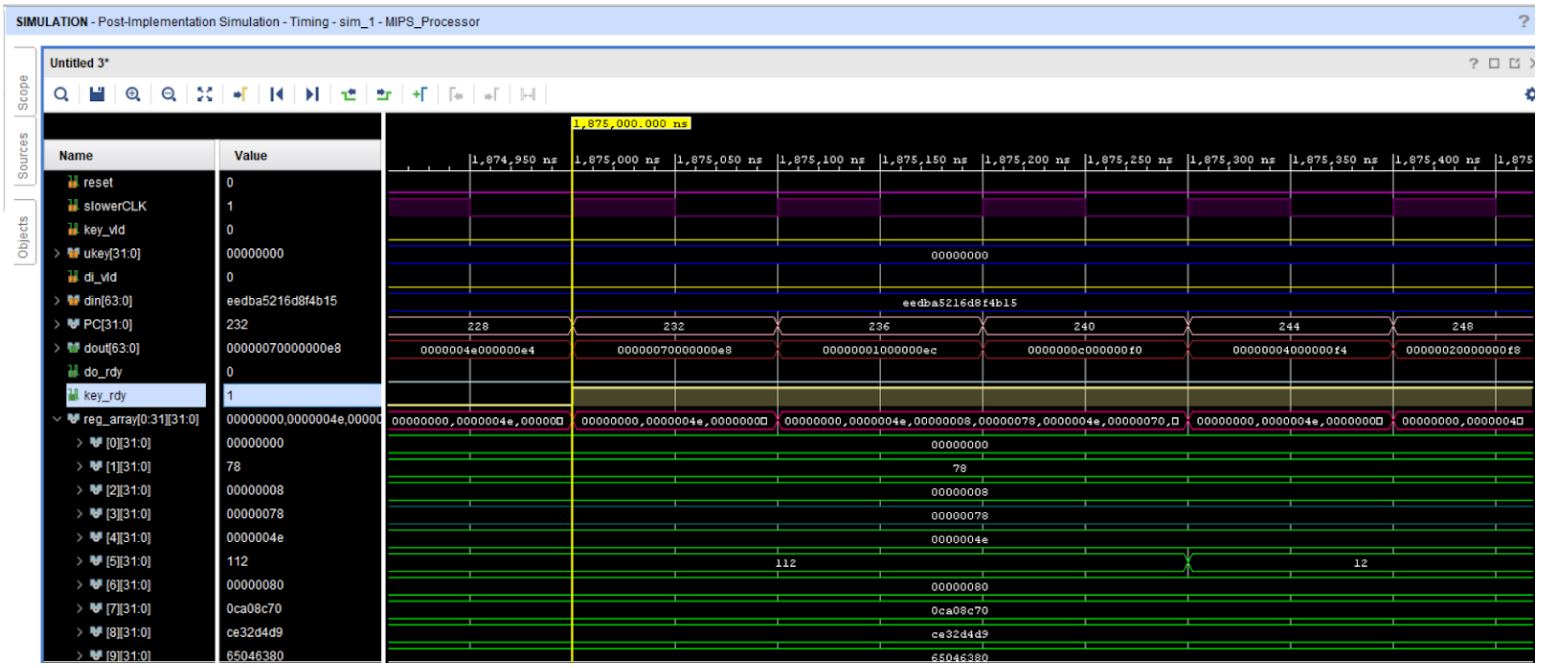
In the below timing simulation results we have done decryption of data = x" **eedba5216d8f4b15**". When the reset is '1', PC is 0 and all registers are "00000000".
Here, PC is the value of current PC. Here, di_vld and key_vld is '1' and we are able to take inputs ukey and din. When the reset is '0', key expansion starts. Dout's lower 32 bits shows the value of PC and upper bits shows ALU result or Branch address or data to be written in the memory according to the type of instructions.



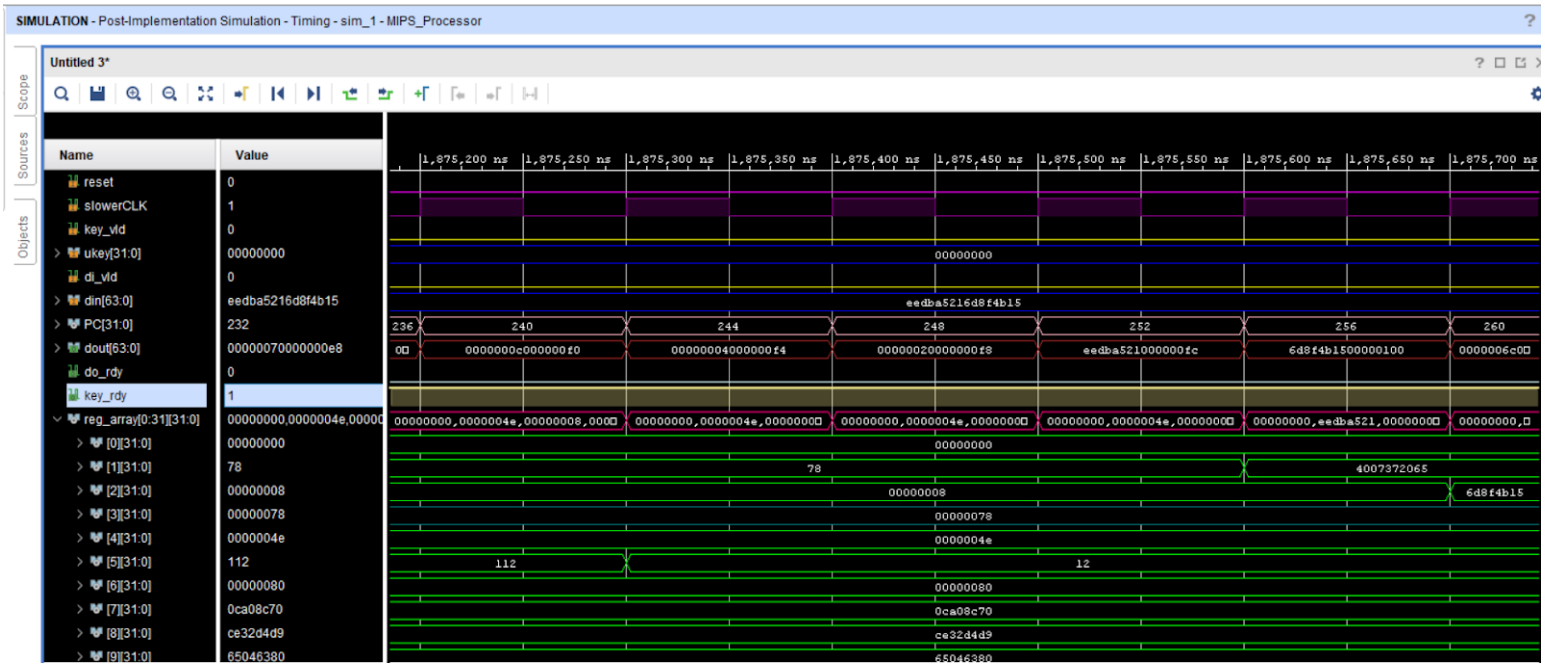
Here, we can see that when the k_cnt which is in R[1] reaches to 78, we get the value of S[25] after all the counts of k in the register R[9]. Which shows that all the values of skey after expansion are generated.



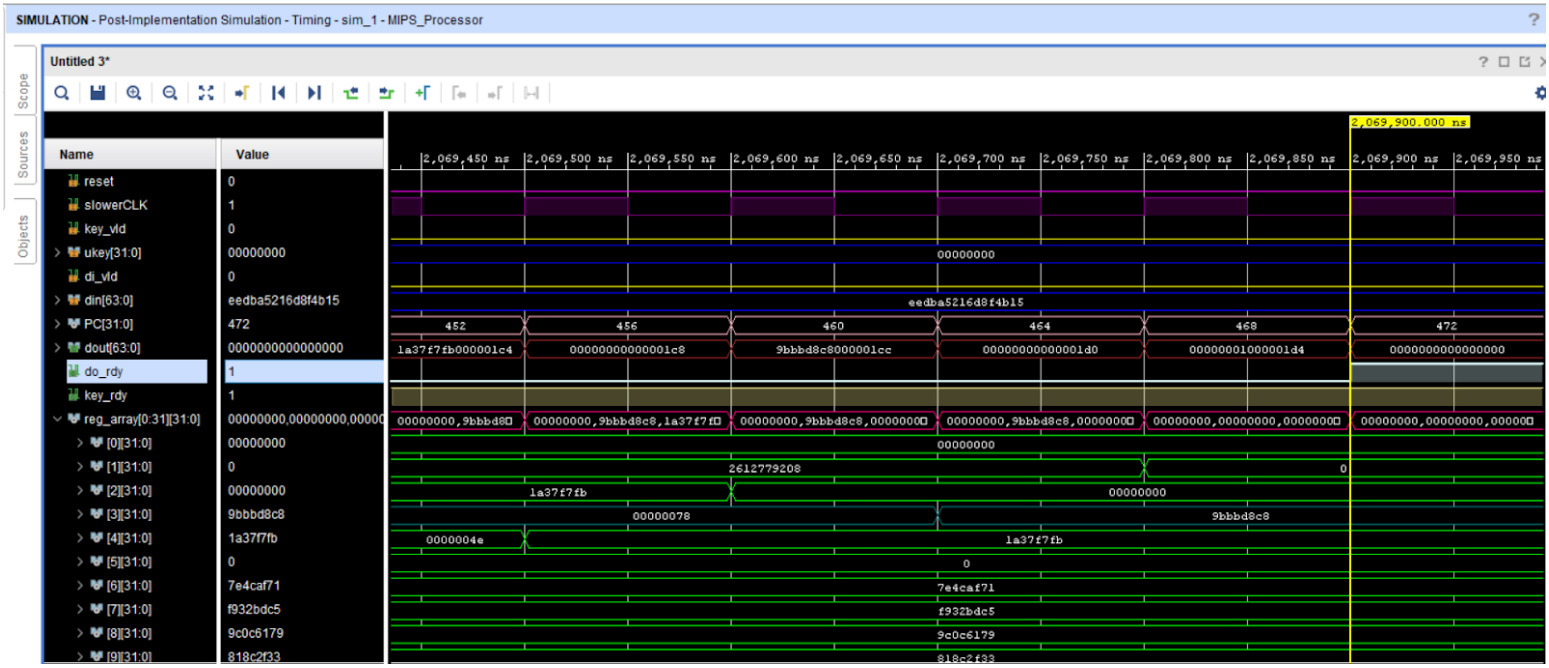

Here, we can see that when the key is ready, key_rdy signal goes high and indicates that key expansion is done.



In the below simulation we can see that the din is available which is the encrypted value of x“0000000000000000”.




In the below simulation, we can see that do_rdy is high, that means the decryption is done and decrypted value is available which is x”0000000000000000”.

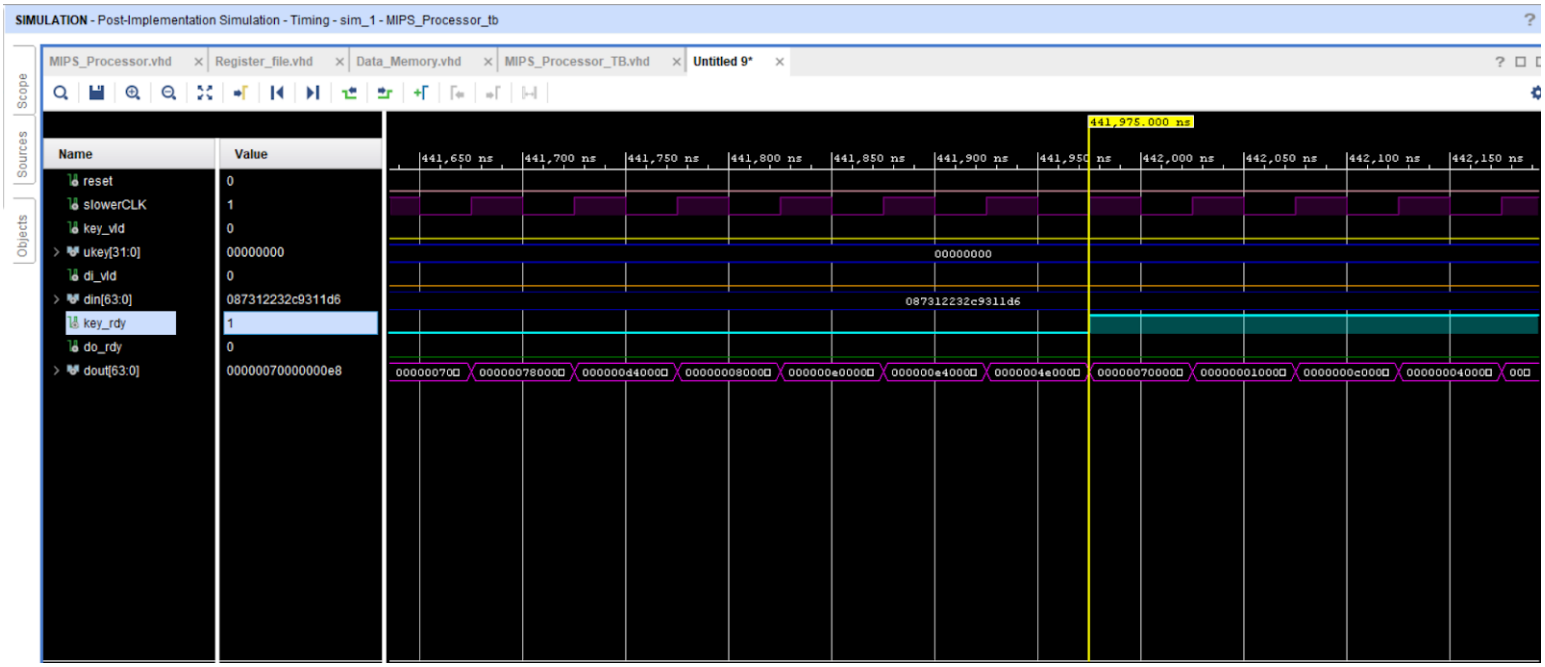


## ➢ Timing Diagram with test bench:

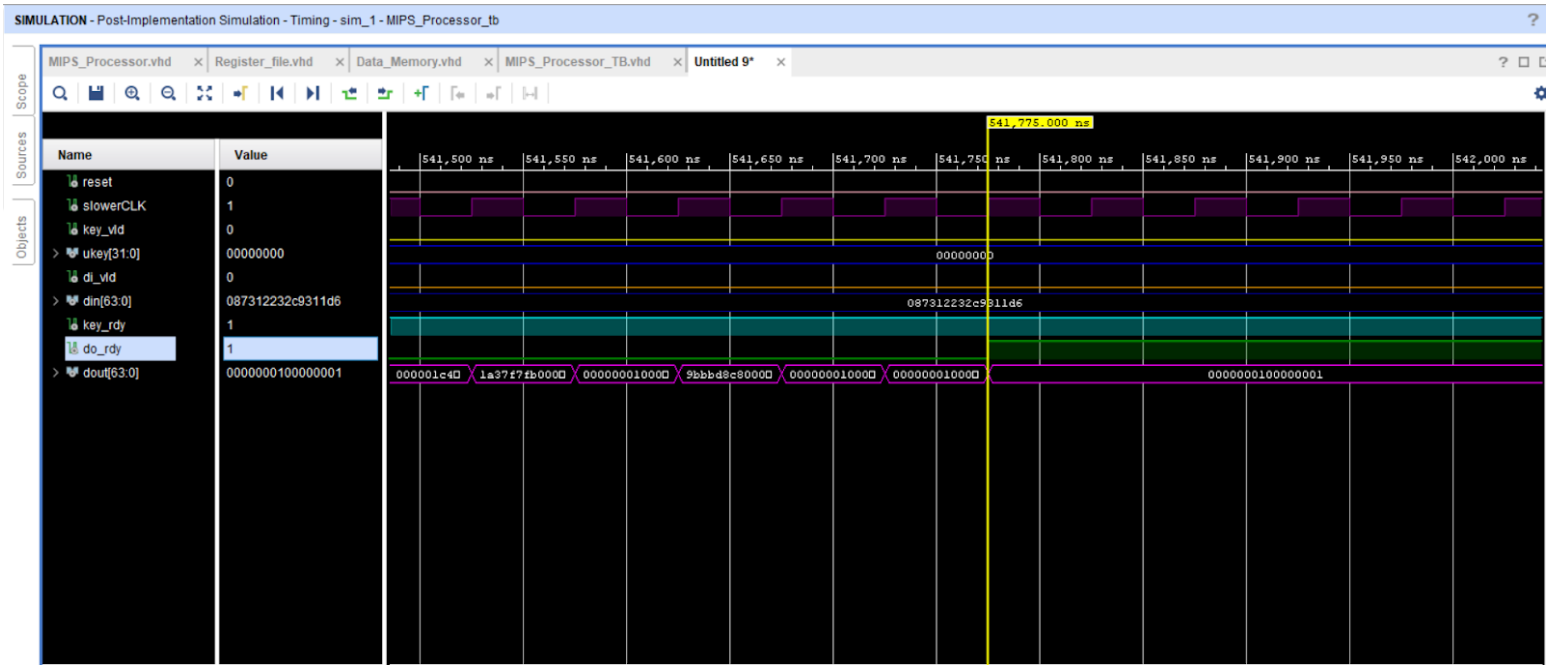
In the below timing simulation results we have done decryption of data = x" **087312232c9311d6**". When the reset is '1', PC is 0 and all registers are "00000000".

Here, PC is the value of current PC. Here, di_vld and key_vld is '1' and we are able to take inputs ukey and din. When the reset is '0', key expansion starts. Dout's lower 32 bits shows the value of PC and upper bits shows ALU result or Branch address or data to be written in the memory according to the type of instructions.

Here, we can see that when the key is ready, key_rdy signal goes high and indicates that key expansion is done.

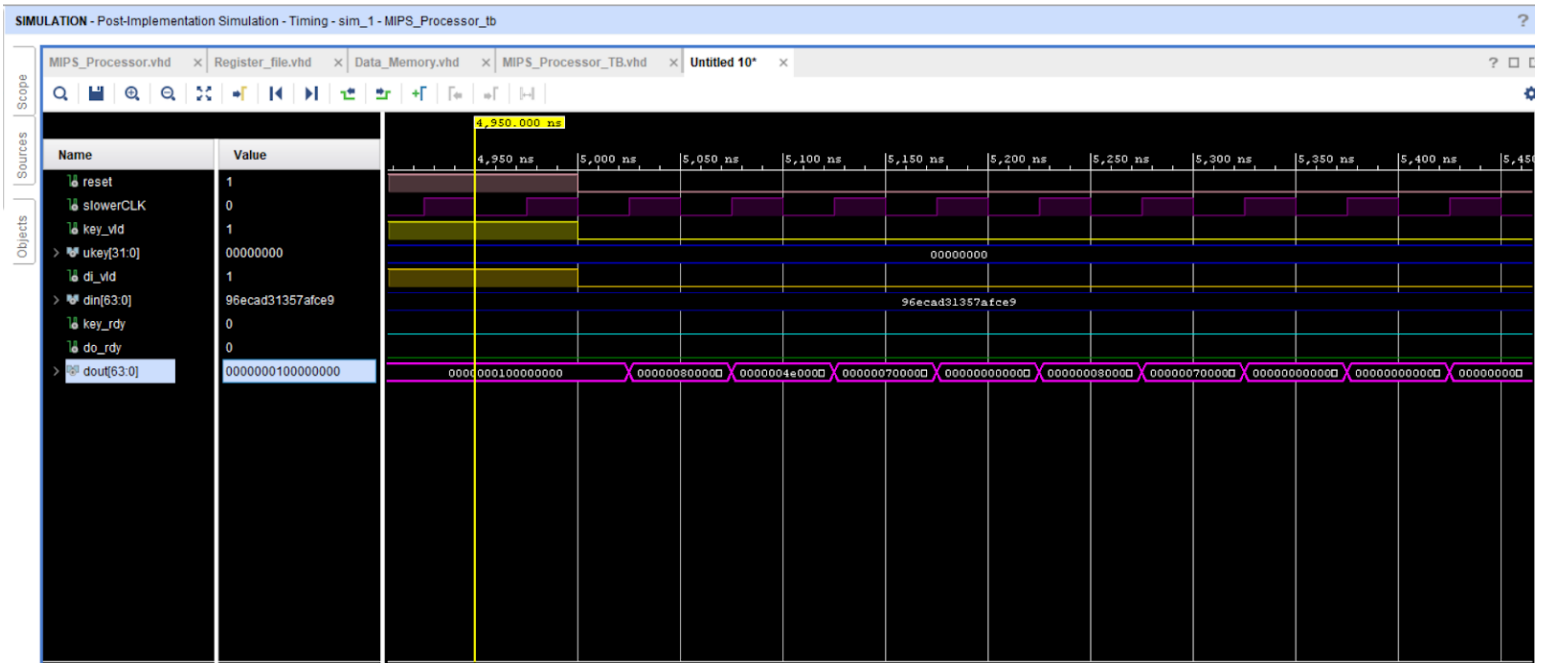


In the below simulation, we can see that do_rdy is high, that means the decryption is done and decrypted value is available which is x”0000000100000001”

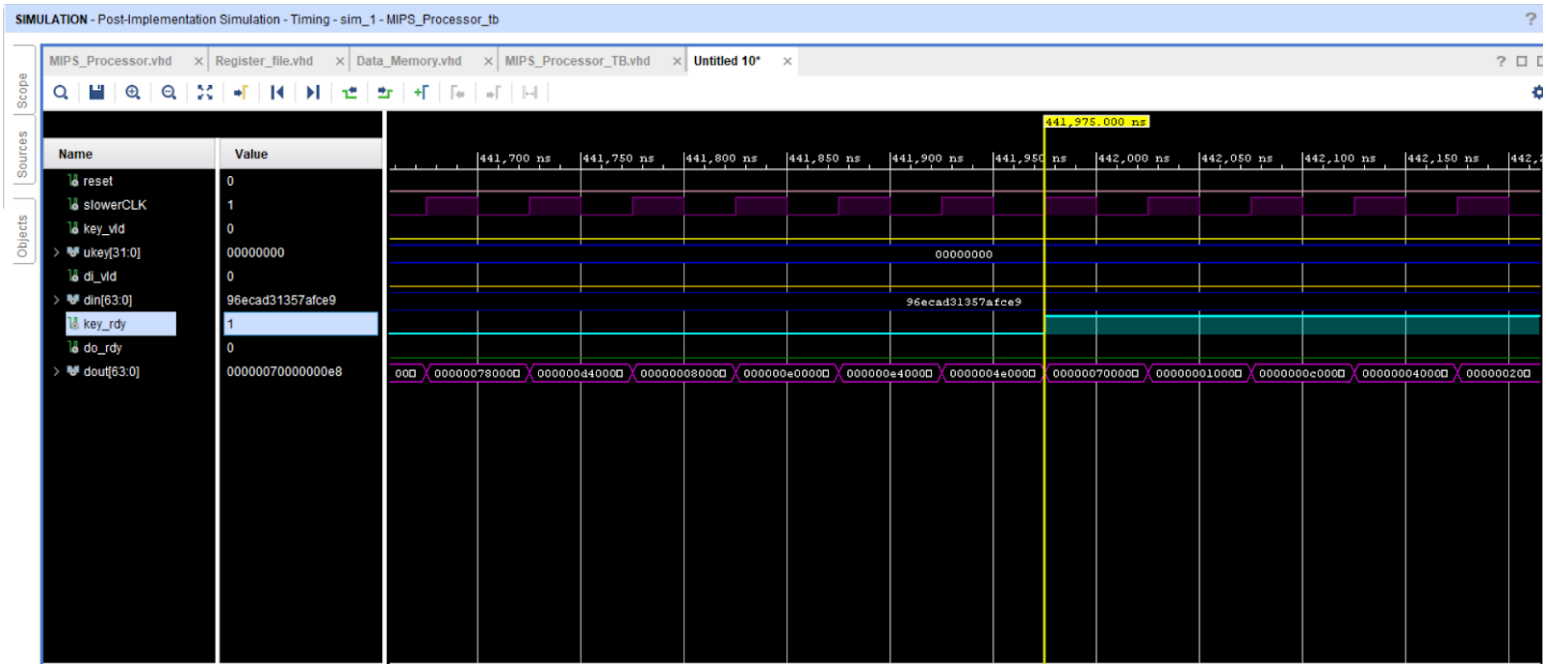


In the below timing simulation results we have done decryption of data = x" **96ecad31357afce9**". When the reset is '1', PC is 0 and all registers are "00000000".

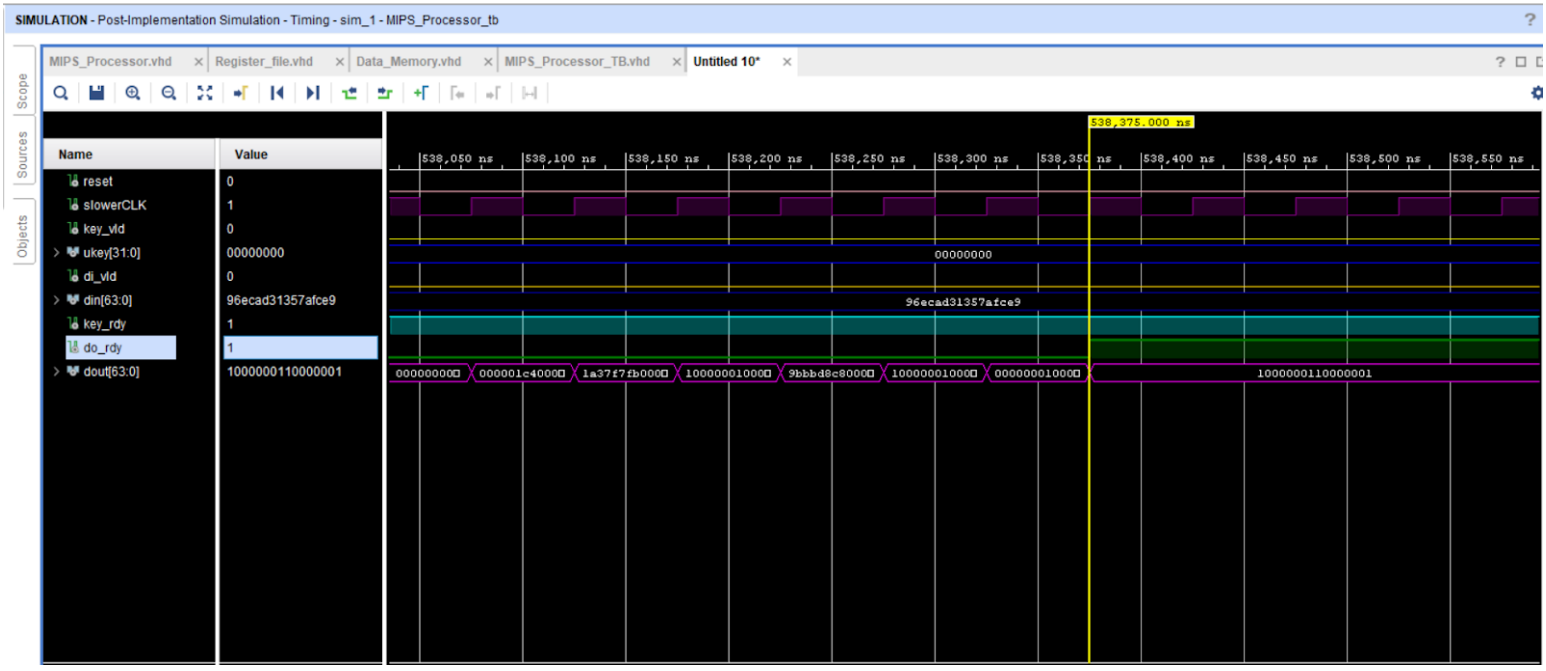
Here, PC is the value of current PC. Here, di_vld and key_vld is '1' and we are able to take inputs ukey and din. When the reset is '0', key expansion starts. Dout's lower 32 bits shows the value of PC and upper bits shows ALU result or Branch address or data to be written in the memory according to the type of instructions.



Here, we can see that when the key is ready, key_rdy signal goes high and indicates that key expansion is done.




In the below simulation, we can see that do_rdy is high, that means the decryption is done and decrypted value is available which is x"1000000110000001"



## ➢ Resource Utilization:

→ Post Synthesis Resource Utilization:

| Name | Slice LUTs (63400) | Slice Registers (126800) | F7 Muxes (31700) | F8 Muxes (15850) | Bonded IOB (210) | BUFGCTRL (32) |
|---|---|---|---|---|---|---|
| MIPS_Processor | 5341 | 3399 | 587 | 108 | 169 | 1 |
| DM (Data_Memory) | 2843 | 2240 | 235 | 36 | 0 | 0 |
| DU (Decode_Unit) | 68 | 63 | 0 | 0 | 0 | 0 |
| Mux1 (Mux2_1_32bit) | 16 | 0 | 0 | 0 | 0 | 0 |
| Mux2 (Mux2_1_32bit_0) | 32 | 0 | 0 | 0 | 0 | 0 |
| Mux3_1 (Mux3_1_32bit) | 0 | 31 | 0 | 0 | 0 | 0 |
| PC (Program_Counter) | 1323 | 41 | 96 | 8 | 0 | 0 |
| PCB (PCBranch) | 30 | 0 | 0 | 0 | 0 | 0 |
| RF (Register_file) | 1029 | 1024 | 256 | 64 | 0 | 0 |

→ Post Implementation Resource Utilization:

| Name | Slice LUTs (63400) | Slice Registers (126800) | F7 Muxes (31700) | F8 Muxes (15850) | Slice (15850) | LUT as Logic (63400) | LUT Flip Flop Pairs (63400) | Bonded IOB (210) | BUFGCTRL (32) |
|---|---|---|---|---|---|---|---|---|---|
| MIPS_Processor | 5341 | 3399 | 587 | 108 | 2093 | 5341 | 472 | 169 | 2 |
| DM (Data_Memory) | 2843 | 2240 | 235 | 36 | 1347 | 2843 | 2 | 0 | 0 |
| DU (Decode_Unit) | 68 | 63 | 0 | 0 | 65 | 68 | 0 | 0 | 0 |
| Mux1 (Mux2_1_32bit) | 16 | 0 | 0 | 0 | 13 | 16 | 0 | 0 | 0 |
| Mux2 (Mux2_1_32bit_0) | 32 | 0 | 0 | 0 | 21 | 32 | 0 | 0 | 0 |
| Mux3_1 (Mux3_1_32bit) | 0 | 31 | 0 | 0 | 9 | 0 | 0 | 0 | 0 |
| PC (Program_Counter) | 1323 | 41 | 96 | 8 | 593 | 1323 | 0 | 0 | 1 |
| PCB (PCBranch) | 30 | 0 | 0 | 0 | 8 | 30 | 0 | 0 | 0 |
| RF (Register_file) | 1029 | 1024 | 256 | 64 | 778 | 1029 | 0 | 0 | 0 |

## ➢ Critical Path Delay and Maximum Clock Frequency:

**Design Timing Summary**

| Setup | | Hold | | Pulse Width | |
|---|---|---|---|---|---|
| Worst Negative Slack (WNS): | 0.899 ns | Worst Hold Slack (WHS): | 0.218 ns | Worst Pulse Width Slack (WPWS): | 19.500 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: | 6240 | Total Number of Endpoints: | 6240 | Total Number of Endpoints: | 3210 |

**All user specified timing constraints are met.**

Here, we can see that the Worst Negative Slack is 0.899 ns and the clock constraint is 40 ns. Therefore, the **critical path delay is 39.101 ns.**

Hence, the **maximum clock frequency is 25.57MHz.**

## ➢ Assembly Code for Key Expansion and Decryption:

➔ Key Expansion:

```
//R19 =1, R4 = 78, R5 = 112, R6 = 128

//Data memory is byte addressable

//S[0] starts from mem[8] to mem[11],....,S[25] starts from mem[108] to mem[111]

//L[3] starts from mem[112] to mem[115],....,L[0] is from mem[124] to mem[127]


// Initialization of counter

(0)  ADDI R19, R0, 1
(1)  ADDI R6, R0, 128
(2)  ADDI R4, R0, 78
(3)  ADDI R5, R0, 112
(4)  ADDI R1,R0,0  //R1 = K_CNT = 0
(5)  ADDI R2,R0,8  //R2 = I_CNT = 8
(6)  ADDI R3,R0,112  //R3 = J_CNT = 112
(7)  ADD R7,R0,R0   //R7 = a_tmp1
(8)  ADD R8,R0,R0   //R8 = S[i]
(9)  ADD R9,R0,R0   //R9 = a_reg
(10)        ADD R10,R0,R0   //R10 = b_reg
(11)        ADD R11,R0,R0   //R11 = a_reg + b_reg
(12)        ADD R12,R0,R0   //R12 = ab_tmp
(13)        ADD R13,R0,R0   //R13 = L[j]
(14)        ADD R14,R0,R0   //R14 = b_temp1
(15)        ADDI R15, R0, 3 //R15 = 3, DATA INDEPENDENT LEFT ROTATE

//A = S[i] = ROTL(S[i] + (A+B), 3)

//R11 = R9 + R10 = a_reg + b_reg

(16)        78_loops: ADD R11, R9, R10

//a_tmp1 = R7 = R8 + R11 = S[i] + a_reg + b_reg

(17)        LB R8, R2, 0    //R8 = S[i]
(18)        ADD R7, R8, R11

//a_reg = R9 = R7 << R15 = a_tmp1 << 3

(19)        ADD R9, R7, R0          //R9 INITAILIZED TO R7
(20)        ADD R16, R15, R0          // R16 = R15  // COUNTER
```

```
(21)        START:BLT R9, R0, <Branch address:MSB1> // JUMP TO MSB1 WHEN R9 < 0
(22)        SHL R9, R9, 1                  // SHIFT R9 BY 1
(23)        SUB R16, R16, R19                                    // DECREMENT COUNTER
(24)        BNE R16, R0, <Branch address: START>
(25)        BEQ R16, R0, <Branch address: AFTER_ROTATE>
(26)        MSB1: SHL R9, R9, 1                                                  // SHIFT R9 BY 1
(27)        ADD R9, R9, R19
(28)        SUB R16, R16, R19                                                    // DECREMENT COUNTER
(29)        BNE R16, R0, <Branch address: START>

//S[i]= a_reg = R9

(30)        AFTER_ROTATE: SB R9,R2,0            //Mem[R2 + 0] = R9

//B = L[j] = ROTL(L[j] + (A+B), (A+B))

//ab_tmp = R12 = R9 + R10 = a_reg + b_reg

(31)        ADD R12, R9, R10

//b_tmp1 = R14 = R13 + R12 = L[j] + ab_tmp

(32)        LB R13, R3, 0                  // R13 = L[j]
(33)        ADD R14, R13, R12
(34)        ANDI R12, R12, 1F

//b_reg = R10 = R14 << R12 = b_tmp1 << ab_tmp

(35)        ADD R10, R14, R0          //R10 INITAILIZED TO R14
(36)        ADD R16, R12, R0          // R16 = R12  // COUNTER
(37)        BEQ R16, R0, <Branch address: AFTER_ROTATE >
(38)        START:BLT R10, R0, <Branch address:MSB1>          // JUMP TO MSB1 WHEN R10 < 0
(39)        SHL R10, R10, 1                  // SHIFT R10 BY 1
(40)        SUB R16, R16, R19                                    // DECREMENT COUNTER
(41)        BNE R16, R0, <Branch address: START>
(42)        BEQ R16, R0, <Branch address: AFTER_ROTATE>
(43)        MSB1: SHL R10, R10, 1                                                  // SHIFT R10 BY 1
(44)        ADD R10, R10, R19
(45)        SUB R16, R16, R19                                                    // DECREMENT COUNTER
(46)        BNE R16, R0, <Branch address: START>

//L[j]= b_reg = R10

(47)        AFTER_ROTATE: ADD R10, R10, R0
(48)        SB R10,R3,0            //Mem[R3 + 0] = R10
```

```
//Incrementing counter

(49)        ADDI R1, R1, 1                  // R1 = k, k = k+1
(50)        ADDI R2, R2, 4                  // R2 = i, i = i+4
(51)        ADDI R3, R3, 4                  // R3 = j, j = j+4
(52)        BNE R2, R5, <Branch address: j_cnt>

//When i_cnt reaches max value, it resets to 8

(53)        ADDI R2, R0, 8
(54)        j_cnt: BNE R3, R6, <Branch Address:k_cnt>

//When j_cnt reaches max value, it resets to 112

(55)        ADDI R3, R0, 112
(56)        k_cnt: BNE R1, R4, <Branch Address: 78_loops>
(57)        ADD R31, R1, R0
```

➔ Decryption:

```
//R1 = A, R2 = B, R19 =1, R0 = 0, R16 = 112, R5 = 12, R10 =4, R12 = 32

(58) ADDI R16, Ox70, R0          //R16 = 112, MEM COUNTER
(59) ADDI R19, 0x01, R0          //R19 = 1
(60) ADDI R5, 0X0C, R0           //R5 = 12, RC5 DECRYPTION COUNTER FROM 12 TO 1
(61) ADDI R10, 0x04, R0          //R10 = 4
(62) ADDI R12, 0x20, R0          //R12 = 32

(63) LB R1,0(R0)                 //R1 = A, FROM MEM[0]
(64) LB R2,4(R0)                 //R2 = B, FROM MEM[4]

//START OF 12 ITERATIONS

//B = (B - S[2*i + 1]) >> A) XOR A

//R6 = B - S[2*i + 1]
(65) Dec_Loop: SUB R16, R16, R10          //R16 = R16 - 4 = 108, 100,...
(66) LB R17, 0(R16)              //R17 = MEM[R16] = S[25], S[23], S[21],....S[3]
(67) SUB R6, R2, R17             //R6 = B - S[2*i + 1]

//R7 = (B - S[2*i + 1]) >> A) = R6 >> A = R6 >> R1

(68) ANDI R11, 0x1F, R1          //R11 = R1(4 DOWNTO 0)
```

```
//RIGHT ROTATE BY X = LEFT ROTATE BY (32 - X)
(69) SUB R11, R12, R11              //R11 = 32 - R11

//R7 = R6 << R11, LEFT ROTATE BY R11, RIGHT ROTATED BY R1

(70) ADD R7, R6, R0           //R7 INITAILIZED TO R6
(71) ADD R20, R11, R0          // R20 = R11  // I COUNTER = R11

(72) BEQ R20, R0, <Branch address: shift_zero> // JUMP TO shift_zero when true //BRANCH +9

(73) START:BLT R7, R0, <Branch address:MSB1> // JUMP TO MSB1 WHEN R7 < 0 //BRANCH +4
(74) SHL R7, R7, 1                                                  // SHIFT R7 BY 1
(75) SUB R20, R20, R19                                              // DECREMENT COUNTER
(76) BNE R20, R0, <Branch address: START>  // JUMP TO START WHEN R20 IS NOT R0 //
(77) BEQ R20, R0, <Branch address: shift_zero>   // JUMP TO HALT WHEN R20 IF 0 // BRANCH +4

(78) MSB1: SHL R7, R7, 1                                            // SHIFT R7 BY 1
(79) ADD R7, R7, R19
(80) SUB R20, R20, R19                                              // DECREMENT COUNTER
(81) BNE R20, R0, <Branch address: START> //JUMP TO START WHEN R20 IS NOT 0 // BRANCH -9
(82) shift_zero: ADD R7, R7, R0      // COPY R7 IN R7

//B = (B - S[2*i + 1]) >> A) XOR A
//R2 = R7 XOR R1
(83) NOR R13,R1,R7
(84) NOR R11,R13,R1
(85) NOR R18,R13,R7
(86) NOR R13,R11,R18
(87) NOR R2,R13,R13       //R2 <- R7 XOR R1


//A = (A - S[2*i]) >> B) XOR B
//R8 = A - S[2*i]
(88) SUB R16, R16, R10              //R16 = R16 - 4 = 104, 96,...
(89) LB R17, 0(R16)                 //R17 = MEM[R16] = S[24], S[22], S[20],....S[2]
(90) SUB R8, R1, R17                //R8 = A - S[2*i]

//R9 = (A - S[2*i]) >> B) = R8 >> B = R8 >> R2

(91) ANDI R11, 0x1F, R2             //R11 = R2(4 DOWNTO 0)
```

```
//RIGHT ROTATE BY X = LEFT ROTATE BY (32 - X)
(92) SUB R11, R12, R11              //R11 = 32 - R11

//R9 = R8 << R11, LEFT ROTATE BY R11, RIGHT ROTATED BY R2

(93) ADD R9, R8, R0           //R9 INITAILIZED TO R8
(94) ADD R20, R11, R0          // R20 = R11  // I COUNTER = R11

(95) BEQ R20, R0, <Branch address: shift_zero> // JUMP TO shift_zero when true //BRANCH +9

(96) START:BLT R9, R0, <Branch address:MSB1> // JUMP TO MSB1 WHEN R9 < 0 //BRANCH +4
(97) SHL R9, R9, 1                                                  // SHIFT R9 BY 1
(98) SUB R20, R20, R19                                                  // DECREMENT COUNTER
(99) BNE R20, R0, <Branch address: START>  // JUMP TO START WHEN R20 IS NOT R0 //
(100) BEQ R20, R0, <Branch address: HALT>   // JUMP TO HALT WHEN R20 IF 0 // BRANCH +4

(101) MSB1: SHL R9, R9, 1                                               // SHIFT R9 BY 1
(102) ADD R9, R9, R19
(103) SUB R20, R20, R19                                                 // DECREMENT COUNTER
(104) BNE R20, R0, <Branch address: START> //JUMP TO START WHEN R20 IS NOT 0 // BRANCH -9
(105) shift_zero: ADD R9, R9, R0      // COPY R9 IN R9

//A = (A - S[2*i]) >> B) XOR B
//R1 = R9 XOR R2
(106) NOR R13,R2,R9
(107) NOR R11,R13,R2
(108) NOR R12,R13,R9
(109) NOR R13,R11,R12
(110) NOR R1,R13,R13      //R1 <- R9 XOR R2


//Decrementing Decryption Counter
(111) SUB R5,R5,R19               //R5 = 11 TO 0
(112) BNE R5,R0,<<BRANCH ADDR = Dec_Loop: > // IF BRANCH IS NOT EQUAL THEN GO TO Dec_Loop:


//B = B - S[1]
(113) LB R4,12(R0)                //R4 <- S[1]
(114) SUB R2,R2,R4                //R6 <- B - S[1]
```

```
//A = A - S[0]
(115) LB R3,8(R0)          //R3 <- S[0]
(116) SUB R1,R1,R3         //R5 = A - S[0]
(117) ADDI R30, R0, 1
(118) HALT
```