EL 6463 Advanced Hardware Design Term Project Report NYU Processor Design and RC5 Encryption Algorithm Implementation

Name: Tuna Bicim
NYU ID: 13940901
Net ID: tb2358

25.10.2018

Table of Contents

1. Introduction

In this report details of design and implementation of the EL6463 Term Project NYU Processor Design will be explained in a detailed manner. The design is implemented on a Digilent Nexys 4 DDR FPGA Board using VHDL language. After covering the design steps, the performance, timing analysis, functional analysis and area analysis of the implementation will be eloborated.

After covering the details of the microprocessor explaination of RC5 algorithm, assembly codes and the interface to supply the inputs and observe the outputs will be made. A video demonstration on the implementation of the algorithm on Nexys 4 DDR FPGA can also be seen on: https://youtu.be/XiiIkfdefbc

2. Block Diagram

The NYU Processor has a very similar architecture with the MIPS Single Cycle Processor. It is capable of performing a reduced instruction set of MIPS Processor. These instructions cover most of the R type instructions (Add,Sub,And,Or,Nor), some of the I type instructions (Immidiate versions of R type instructions, LSL,STR,LD,BLT,BEQ,BNE) and J type (Jump) instructions. The processor uses word (32 bits) addressable memory (512kB each) for storing Data and Instructions and has 32 registers on the register file. The branches are taken with respect to PC+1 value with an offset.

At each clock cycle the instruction pc points to an address on the instruction memory and this instruction is read. After tht in the same cycle, the instruction is decoded by the controller and depending on the type different number of registers are read, some operations are done on ALU and memory can be read or not. After finishing the calculations for the operation at hand, the values are stored in a register, data memory or pc register. The block diagram of the system can be seen in Figure 1. Block Diagram of the NYU Processor. There are couple of changes that were made to the block diagram provided with the Term Project report, these are:

- Addition of another multiplexer for Jump instruction
- Change of Branch signal into PC Source and moving the branch decision inside the controller
- Removal of the shifter before PC Branch adder because the usage of word addressable memory

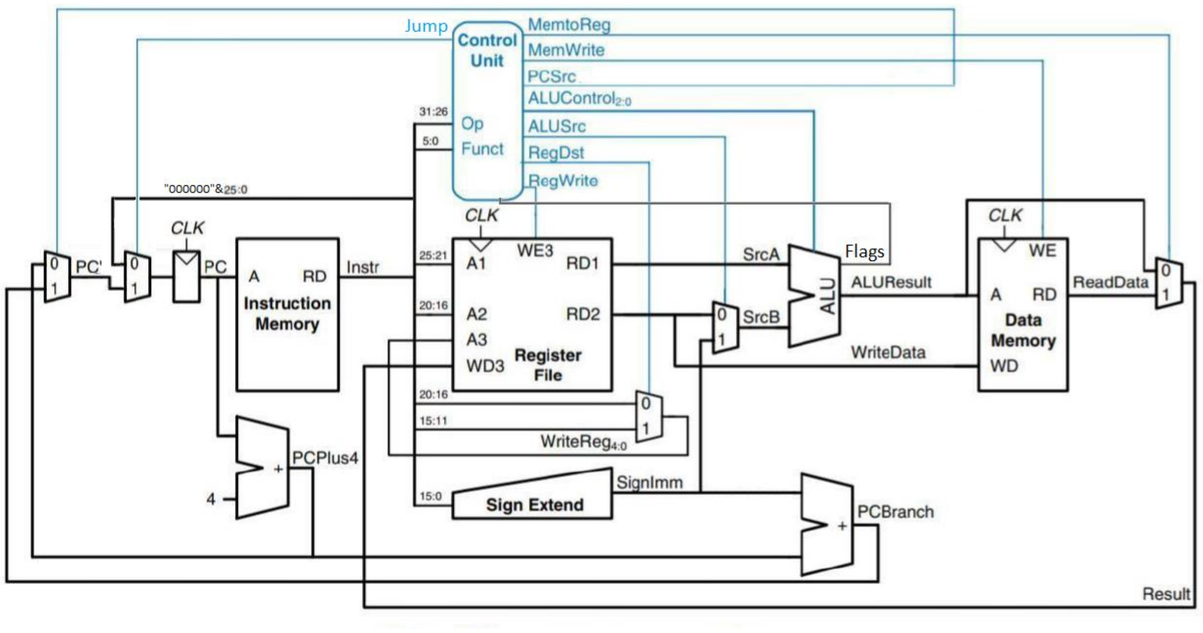


Figure 1. Block Diagram of the NYU Processor

3. RC5 Assembly Code

Before going into simulations of the microprocessor explanation of the RC5 algorithm and implementation on the assembly will be covered so that the operations of the simulation are clear. This part will have 3 subsections each covering different parts of the RC5 algorithm. We will start with Key Generation, after that we will move on to Encrpytion algorithm and finally we will finish with Decryption algorithm.

Key Generation

Key generation algorithm normally has 3 subparts however, the first part is copying the bytes of the key into words which is different for our case because our keys will be loaded as halfwords instead of bytes and this will be done by the interface, not the processor. This leaves us with initializing the S table and mixing in the secret key. The user supplies a 128 bit key and we have 2 constants P_ω and Q_ω which are fixed numbers stored in the data memory.

Initialization of the S table

The flowchart for this step can be seen in Figure 2.Flowchart of the S array initialization , the formula which is from The RC5 Encryption Algorithm by Ronald R. Rivest[1] is in Figure 3. Initialization Algorithm and the assembly code is in Figure 4. Assembly Code

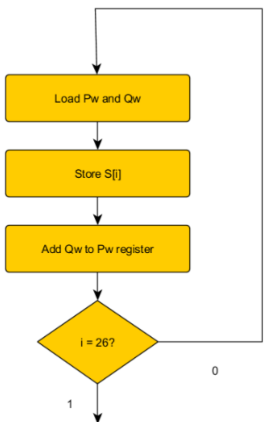
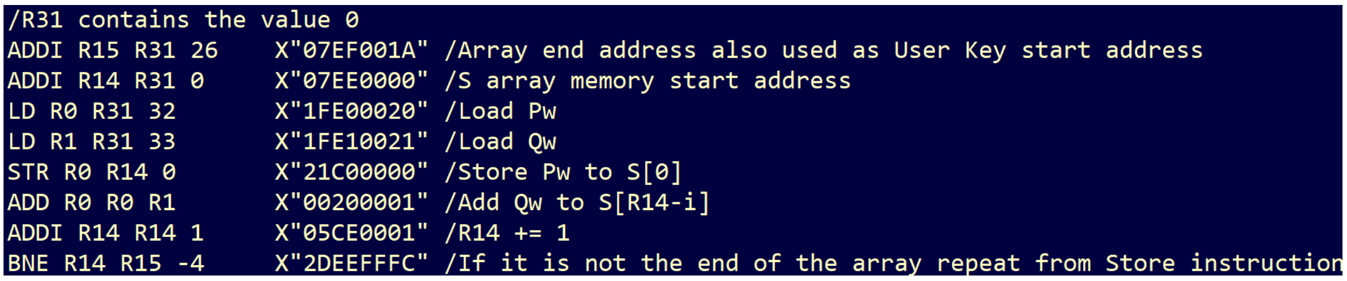


Figure 2.Flowchart of the S array initialization

```
$S[0] = P_w$ ;
for $i = 1$ to $t - 1$ do
        $S[i] = S[i-1] + Q_w$ ;
```

Figure 3. Initialization Algorithm



```
/R31 contains the value 0
ADDI R15 R31 26    X"07EF001A" /Array end address also used as User Key start address
ADDI R14 R31 0     X"07EE0000" /S array memory start address
LD R0 R31 32       X"1FE00020" /Load Pw
LD R1 R31 33       X"1FE10021" /Load Qw
STR R0 R14 0       X"21C00000" /Store Pw to S[0]
ADD R0 R0 R1       X"00200001" /Add Qw to S[R14-i]
ADDI R14 R14 1     X"05CE0001" /R14 += 1
BNE R14 R15 -4     X"2DEEFFFC" /If it is not the end of the array repeat from Store instruction
```

Figure 4. Assembly Code

In the code we have R15 as end of i and R14 as i. The P and Q values are loaded from the memory and S[0] is populated with P. After this, each memory location has the previous location's value plus Q so we create a loop with the Branch not Equal instruction which makes us repeat the procedure like a for loop. After storing values i is incremented to point to the next memory location.

Mixing in the User Key

The flowchart for this step can be seen in Figure 5. Flowchart of , formula which is from The RC5 Encryption Algorithm by Ronald R. Rivest[1] can be seen in Figure 6. Mixing in the User Key of the algorithm and the assembly code is in Figure 7. Implementation of the algorithm.

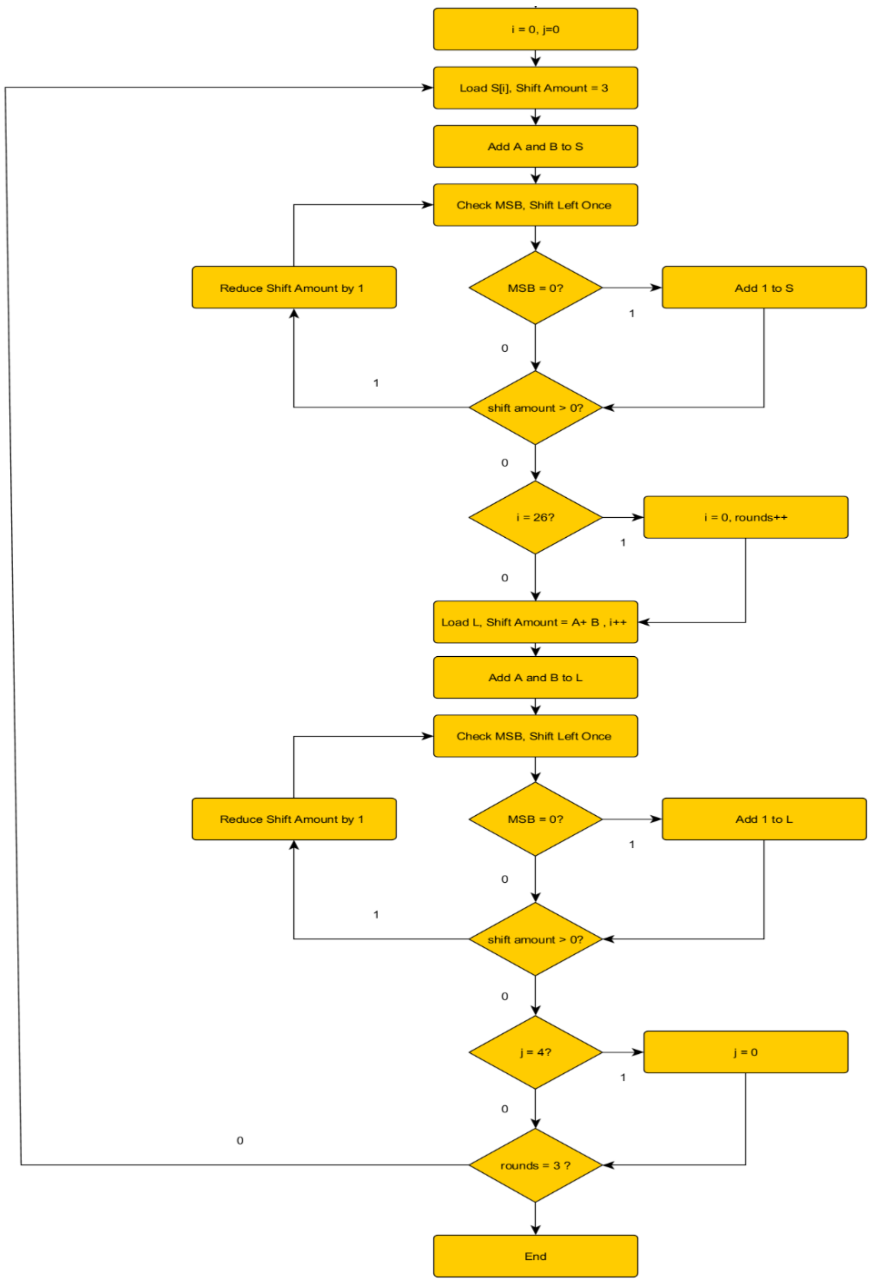
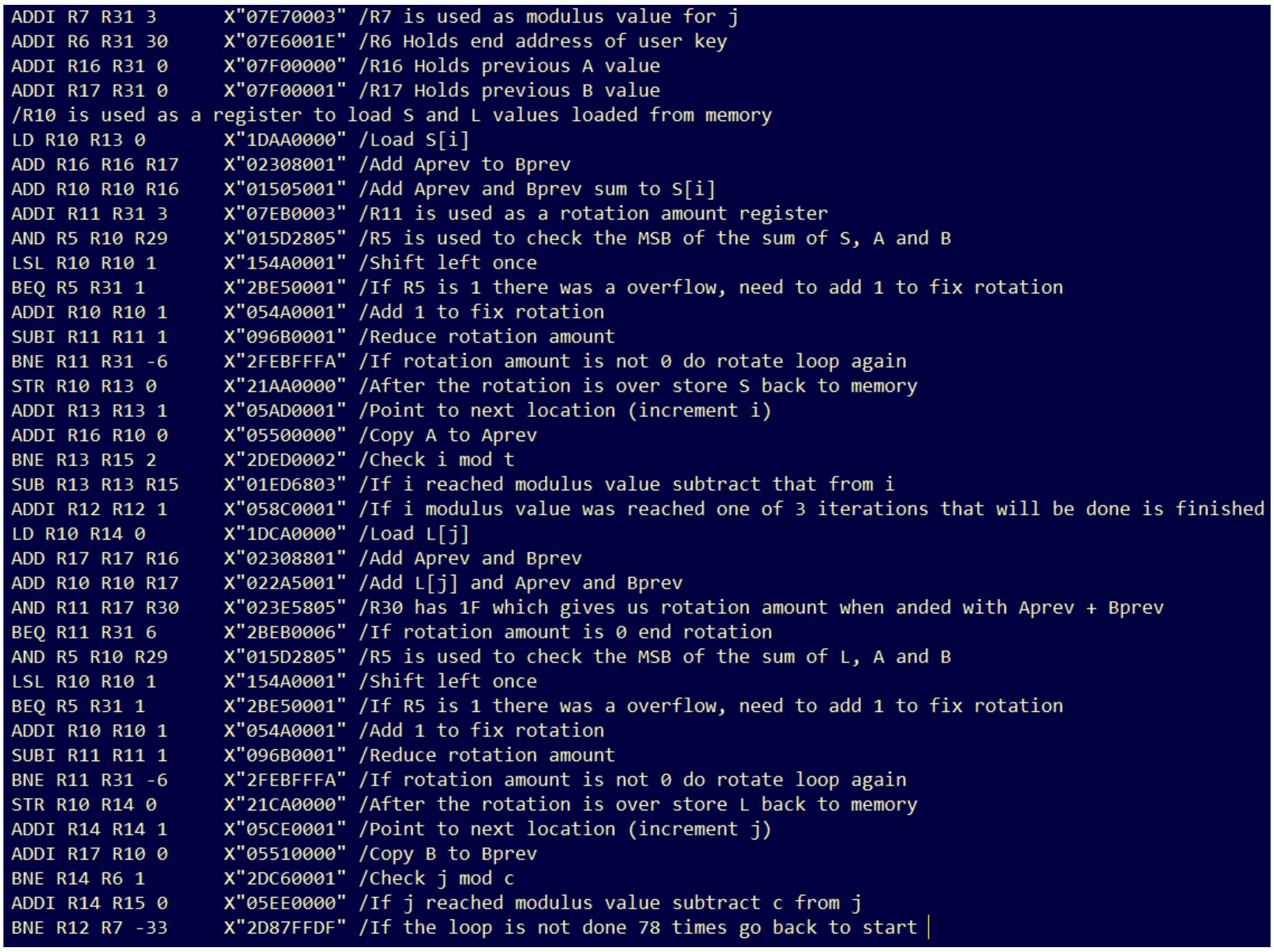


Figure 5. Flowchart of Mixing in the user key

```
$i = j = 0;$
$A = B = 0;$
do $3 * \max(t, c)$ times:
    $A = S[i] = (S[i] + A + B) \lll 3;$
    $B = L[j] = (L[j] + A + B) \lll (A + B);$
    $i = (i + 1) \bmod (t);$
    $j = (j + 1) \bmod (c);$
```

Figure 6. Mixing in the User Key of the algorithm



```
ADDI R7 R31 3       X"07E70003" /R7 is used as modulus value for j
ADDI R6 R31 30      X"07E6001E" /R6 Holds end address of user key
ADDI R16 R31 0      X"07F00000" /R16 Holds previous A value
ADDI R17 R31 0      X"07F00001" /R17 Holds previous B value
/R10 is used as a register to load S and L values loaded from memory
LD R10 R13 0        X"1DAA0000" /Load S[i]
ADD R16 R16 R17     X"02308001" /Add Aprev to Bprev
ADD R10 R10 R16     X"01505001" /Add Aprev and Bprev sum to S[i]
ADDI R11 R31 3      X"07EB0003" /R11 is used as a rotation amount register
AND R5 R10 R29      X"015D2805" /R5 is used to check the MSB of the sum of S, A and B
LSL R10 R10 1       X"154A0001" /Shift left once
BEQ R5 R31 1        X"2BE50001" /If R5 is 1 there was a overflow, need to add 1 to fix rotation
ADDI R10 R10 1      X"054A0001" /Add 1 to fix rotation
SUBI R11 R11 1      X"096B0001" /Reduce rotation amount
BNE R11 R31 -6      X"2FEBFFFA" /If rotation amount is not 0 do rotate loop again
STR R10 R13 0       X"21AA0000" /After the rotation is over store S back to memory
ADDI R13 R13 1      X"05AD0001" /Point to next location (increment i)
ADDI R16 R10 0      X"05500000" /Copy A to Aprev
BNE R13 R15 2       X"2DED0002" /Check i mod t
SUB R13 R13 R15     X"01ED6803" /If i reached modulus value subtract that from i
ADDI R12 R12 1      X"058C0001" /If i modulus value was reached one of 3 iterations that will be done is finished
LD R10 R14 0        X"1DCA0000" /Load L[j]
ADD R17 R17 R16     X"02308801" /Add Aprev and Bprev
ADD R10 R10 R17     X"022A5001" /Add L[j] and Aprev and Bprev
AND R11 R17 R30     X"023E5805" /R30 has 1F which gives us rotation amount when anded with Aprev + Bprev
BEQ R11 R31 6       X"2BEB0006" /If rotation amount is 0 end rotation
AND R5 R10 R29      X"015D2805" /R5 is used to check the MSB of the sum of L, A and B
LSL R10 R10 1       X"154A0001" /Shift left once
BEQ R5 R31 1        X"2BE50001" /If R5 is 1 there was a overflow, need to add 1 to fix rotation
ADDI R10 R10 1      X"054A0001" /Add 1 to fix rotation
SUBI R11 R11 1      X"096B0001" /Reduce rotation amount
BNE R11 R31 -6      X"2FEBFFFA" /If rotation amount is not 0 do rotate loop again
STR R10 R14 0       X"21CA0000" /After the rotation is over store L back to memory
ADDI R14 R14 1      X"05CE0001" /Point to next location (increment j)
ADDI R17 R10 0      X"05510000" /Copy B to Bprev
BNE R14 R6 1        X"2DC60001" /Check j mod c
ADDI R14 R15 0      X"05EE0000" /If j reached modulus value subtract c from j
BNE R12 R7 -33      X"2D87FFDF" /If the loop is not done 78 times go back to start
```

Figure 7. Implementation of the algorithm

In the formula and the code t is related to number of rounds and is 26 times where c is number of words in a key which is 4 for our case. For this part comments are self explanatory. We are doing the loop 3*t times.

Encryption

The flowchart for this step can be seen in Figure 8. Flowchart of the encryption , formula which is from The RC5 Encryption Algorithm by Ronald R. Rivest[1] in Figure 9. Encryption algorithmand assembly code in Figure 10. Implementation of the algorithm.

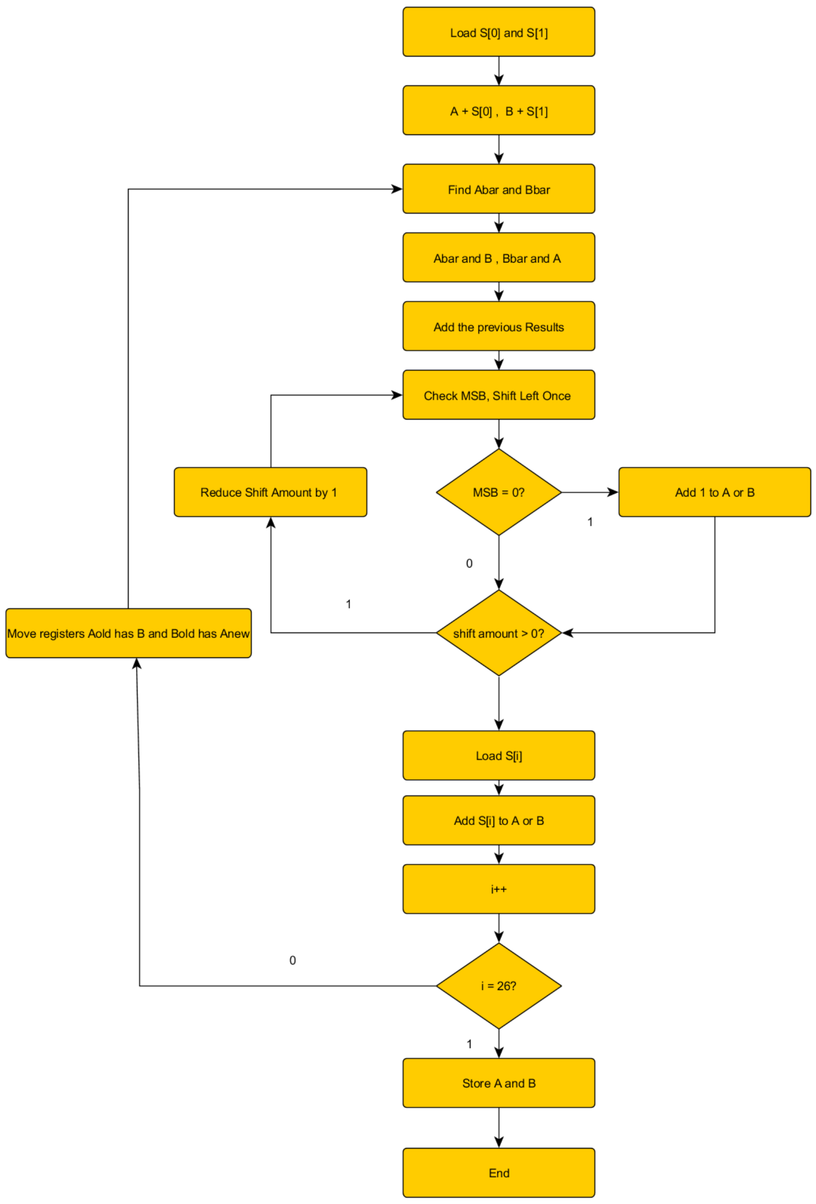
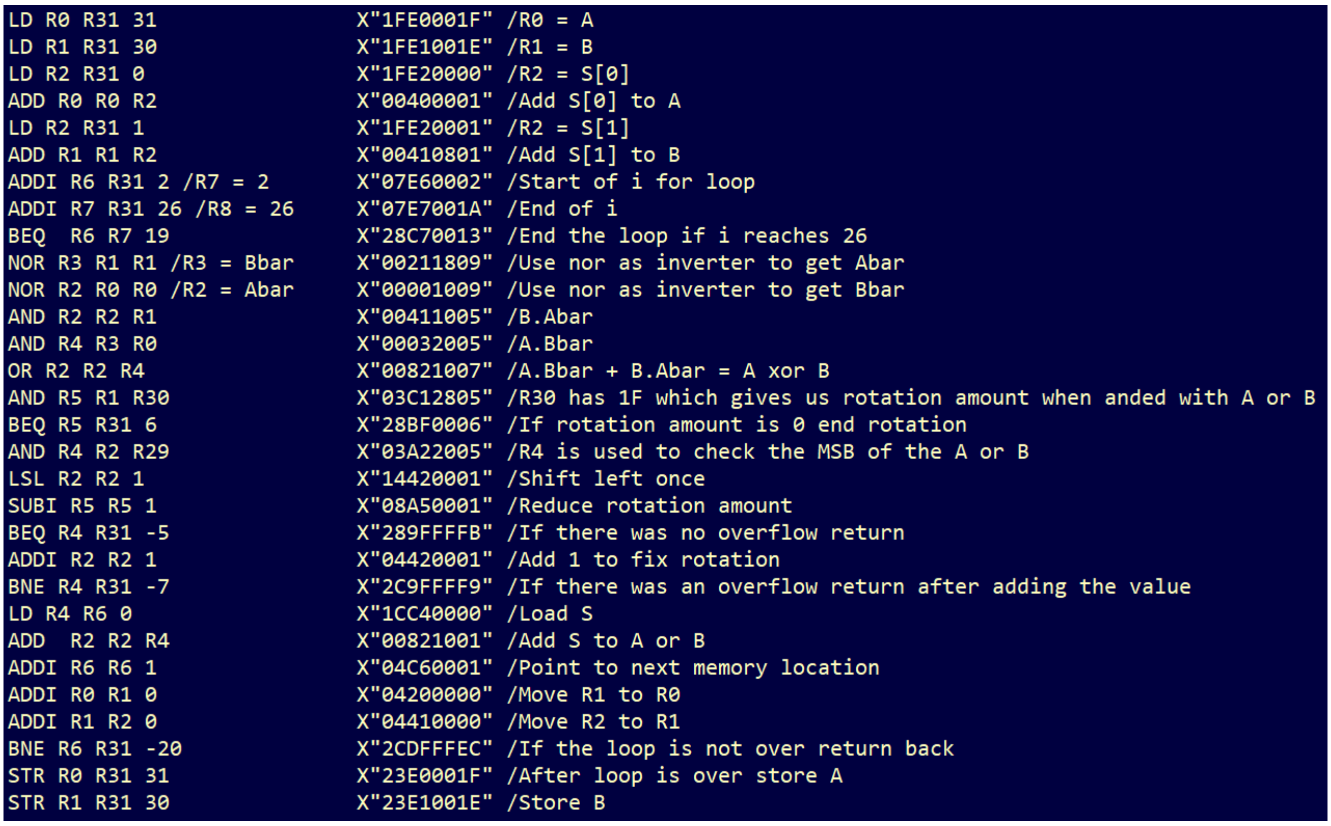


Figure 8. Flowchart of the encryption

```
$A = A + S[0];$
$B = B + S[1];$
for $i = 1$ to $r$ do
        $A = ((A \oplus B) \lll B) + S[2 * i];$
        $B = ((B \oplus A) \lll A) + S[2 * i + 1];$
```

Figure 9. Encryption algorithm



```
LD R0 R31 31                X"1FE0001F" /R0 = A
LD R1 R31 30                X"1FE1001E" /R1 = B
LD R2 R31 0                 X"1FE20000" /R2 = S[0]
ADD R0 R0 R2                X"00400001" /Add S[0] to A
LD R2 R31 1                 X"1FE20001" /R2 = S[1]
ADD R1 R1 R2                X"00410801" /Add S[1] to B
ADDI R6 R31 2 /R7 = 2       X"07E60002" /Start of i for loop
ADDI R7 R31 26 /R8 = 26     X"07E7001A" /End of i
BEQ  R6 R7 19               X"28C70013" /End the loop if i reaches 26
NOR R3 R1 R1 /R3 = Bbar     X"00211809" /Use nor as inverter to get Abar
NOR R2 R0 R0 /R2 = Abar     X"00001009" /Use nor as inverter to get Bbar
AND R2 R2 R1                X"00411005" /B.Abar
AND R4 R3 R0                X"00032005" /A.Bbar
OR R2 R2 R4                 X"00821007" /A.Bbar + B.Abar = A xor B
AND R5 R1 R30               X"03C12805" /R30 has 1F which gives us rotation amount when anded with A or B
BEQ R5 R31 6                X"28BF0006" /If rotation amount is 0 end rotation
AND R4 R2 R29               X"03A22005" /R4 is used to check the MSB of the A or B
LSL R2 R2 1                 X"14420001" /Shift left once
SUBI R5 R5 1                X"08A50001" /Reduce rotation amount
BEQ R4 R31 -5               X"289FFFFB" /If there was no overflow return
ADDI R2 R2 1                X"04420001" /Add 1 to fix rotation
BNE R4 R31 -7               X"2C9FFFF9" /If there was an overflow return after adding the value
LD R4 R6 0                  X"1CC40000" /Load S
ADD  R2 R2 R4               X"00821001" /Add S to A or B
ADDI R6 R6 1                X"04C60001" /Point to next memory location
ADDI R0 R1 0                X"04200000" /Move R1 to R0
ADDI R1 R2 0                X"04410000" /Move R2 to R1
BNE R6 R31 -20              X"2CDFFFEC" /If the loop is not over return back
STR R0 R31 31               X"23E0001F" /After loop is over store A
STR R1 R31 30               X"23E1001E" /Store B
```

Figure 10. Implementation of the algorithm

This is a optimized assembly code that uses the same steps for A and B. The operations are done for the value in R0 register and R1 holds the other part of the plaintext. By this way instead of having the same lines for A and B we have a much shorter code.

Decryption

The formula for this step which is from The RC5 Encryption Algorithm by Ronald R. Rivest[1] can be seen in Figure 11. Decryption algorithm, flowchart in Figure 12. Flowchart of Decryption and assembly code in Figure 13. Implementation of the algorithm.

```
for $i = r$ downto 1 do
        $B = ((B - S[2 * i + 1]) \ggg A) \oplus A;$
        $A = ((A - S[2 * i]) \ggg B) \oplus B;$
$B = B - S[1];$
$A = A - S[0];$
```

Figure 11. Decryption algorithm

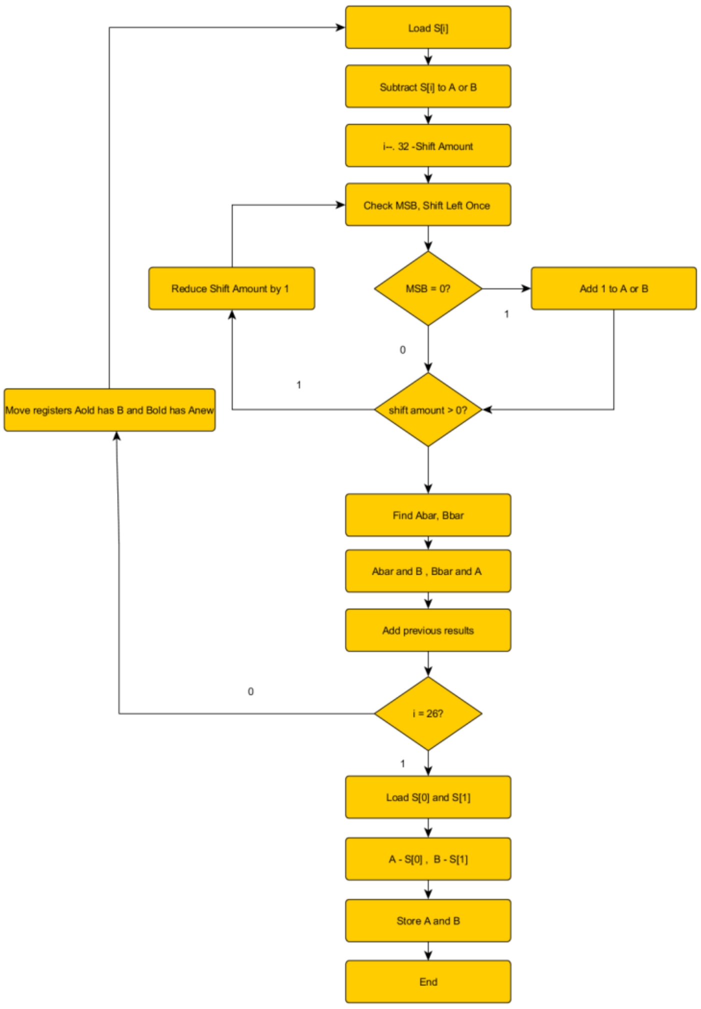
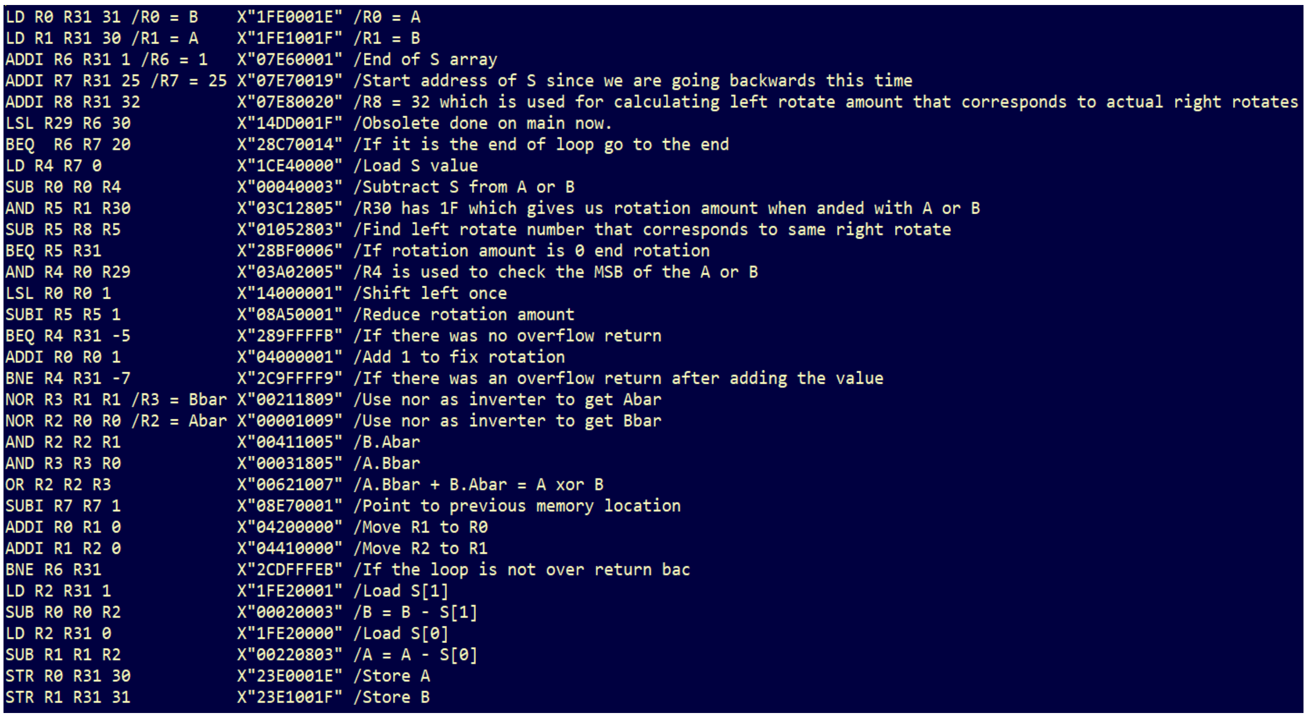


Figure 12. Flowchart of Decryption



```
LD R0 R31 31 /R0 = B    X"1FE0001E" /R0 = A
LD R1 R31 30 /R1 = A    X"1FE1001F" /R1 = B
ADDI R6 R31 1 /R6 = 1   X"07E60001" /End of S array
ADDI R7 R31 25 /R7 = 25 X"07E70019" /Start address of S since we are going backwards this time
ADDI R8 R31 32          X"07E80020" /R8 = 32 which is used for calculating left rotate amount that corresponds to actual right rotates
LSL R29 R6 30           X"14DD001F" /Obsolete done on main now.
BEQ  R6 R7 20           X"28C70014" /If it is the end of loop go to the end
LD R4 R7 0              X"1CE40000" /Load S value
SUB R0 R0 R4            X"00040003" /Subtract S from A or B
AND R5 R1 R30           X"03C12805" /R30 has 1F which gives us rotation amount when anded with A or B
SUB R5 R8 R5            X"01052803" /Find left rotate number that corresponds to same right rotate
BEQ R5 R31              X"28BF0006" /If rotation amount is 0 end rotation
AND R4 R0 R29           X"03A02005" /R4 is used to check the MSB of the A or B
LSL R0 R0 1             X"14000001" /Shift left once
SUBI R5 R5 1            X"08A50001" /Reduce rotation amount
BEQ R4 R31 -5           X"289FFFFB" /If there was no overflow return
ADDI R0 R0 1            X"04000001" /Add 1 to fix rotation
BNE R4 R31 -7           X"2C9FFFF9" /If there was an overflow return after adding the value
NOR R3 R1 R1 /R3 = Bbar X"00211809" /Use nor as inverter to get Abar
NOR R2 R0 R0 /R2 = Abar X"00001009" /Use nor as inverter to get Bbar
AND R2 R2 R1            X"00411005" /B.Abar
AND R3 R3 R0            X"00031805" /A.Bbar
OR R2 R2 R3             X"00621007" /A.Bbar + B.Abar = A xor B
SUBI R7 R7 1            X"08E70001" /Point to previous memory location
ADDI R0 R1 0            X"04200000" /Move R1 to R0
ADDI R1 R2 0            X"04410000" /Move R2 to R1
BNE R6 R31              X"2CDFFFEB" /If the loop is not over return bac
LD R2 R31 1             X"1FE20001" /Load S[1]
SUB R0 R0 R2            X"00020003" /B = B - S[1]
LD R2 R31 0             X"1FE20000" /Load S[0]
SUB R1 R1 R2            X"00220803" /A = A - S[0]
STR R0 R31 30           X"23E0001E" /Store A
STR R1 R31 31           X"23E1001F" /Store B
```

Figure 13. Implementation of the algorithm

This is a optimized assembly code that uses the same steps for A and B. The operations are done for the value in R0 register and R1 holds the other part of the plaintext. By this way instead of having the same lines for A and B we have a much shorter code.

4. Functional Simulation

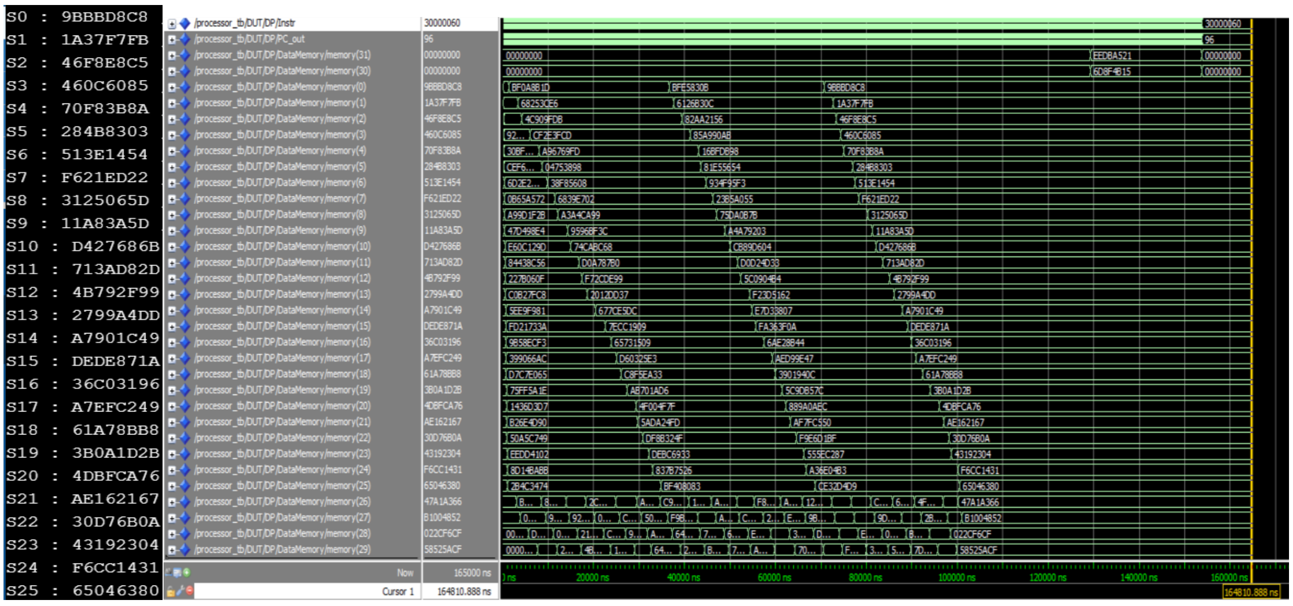
Firstly some test with known inputs and outputs were done and with the given inputs, outputs are observed in order to verify they match the expected values. Two cases on the The RC5 Encryption Algorithm by Ronald R. Rivest[1] are used to test the inputs and outputs. Also, during the laboratory assignments the values for the S array was provided for the key with all zeroes which was used to check the values of the key generation algorithm. Also, if encryption or decryption does not work we wouldn't be able to get the inputs we provided to encryption after decryption. Also a c algorithm was used to verify the values of the key generation values.

First Case:

User Key "0000 0000 0000 0000 0000 0000 0000 0000", Plain Text: "0000 0000 0000 0000"

Expected Output For Cipher Text: "EEDBA521 6D8F4B15"

Please note that on the paper bits are given in big endian configuration and our processor is using little endian. Expected values of the S array as outputed by the c algorithm are given in Figure 14. The expected values of S array and Testbench Results along with the outputs of our microprocessor. Memory locations 0 to 25 correspond to the S array, 26 to 29 to the L array and 30 and 31 correspond to B and A respectively. We can see that all the S values match, other than that our microproccesor outputs the correct values after encryption and reverts back to the plaintext after the decypher algorithm.



```
S0  : 9BBBD8C8
S1  : 1A37F7FB
S2  : 46F8E8C5
S3  : 460C6085
S4  : 70F83B8A
S5  : 284B8303
S6  : 513E1454
S7  : F621ED22
S8  : 3125065D
S9  : 11A83A5D
S10 : D427686B
S11 : 713AD82D
S12 : 4B792F99
S13 : 2799A4DD
S14 : A7901C49
S15 : DEDE871A
S16 : 36C03196
S17 : A7EFC249
S18 : 61A78BB8
S19 : 3B0A1D2B
S20 : 4DBFCA76
S21 : AE162167
S22 : 30D76B0A
S23 : 43192304
S24 : F6CC1431
S25 : 65046380
```

```
/processor_tb/DUT/DP/Instr                       30000060
/processor_tb/DUT/DP/PC_out                      96
/processor_tb/DUT/DP/DataMemory/memory(31)       00000000
/processor_tb/DUT/DP/DataMemory/memory(30)       00000000
/processor_tb/DUT/DP/DataMemory/memory(0)        9BBBD8C8
/processor_tb/DUT/DP/DataMemory/memory(1)        1A37F7FB
/processor_tb/DUT/DP/DataMemory/memory(2)        46F8E8C5
/processor_tb/DUT/DP/DataMemory/memory(3)        460C6085
/processor_tb/DUT/DP/DataMemory/memory(4)        70F83B8A
/processor_tb/DUT/DP/DataMemory/memory(5)        284B8303
/processor_tb/DUT/DP/DataMemory/memory(6)        513E1454
/processor_tb/DUT/DP/DataMemory/memory(7)        F621ED22
/processor_tb/DUT/DP/DataMemory/memory(8)        3125065D
/processor_tb/DUT/DP/DataMemory/memory(9)        11A83A5D
/processor_tb/DUT/DP/DataMemory/memory(10)       D427686B
/processor_tb/DUT/DP/DataMemory/memory(11)       713AD82D
/processor_tb/DUT/DP/DataMemory/memory(12)       4B792F99
/processor_tb/DUT/DP/DataMemory/memory(13)       2799A4DD
/processor_tb/DUT/DP/DataMemory/memory(14)       A7901C49
/processor_tb/DUT/DP/DataMemory/memory(15)       DEDE871A
/processor_tb/DUT/DP/DataMemory/memory(16)       36C03196
/processor_tb/DUT/DP/DataMemory/memory(17)       A7EFC249
/processor_tb/DUT/DP/DataMemory/memory(18)       61A78BB8
/processor_tb/DUT/DP/DataMemory/memory(19)       3B0A1D2B
/processor_tb/DUT/DP/DataMemory/memory(20)       4DBFCA76
/processor_tb/DUT/DP/DataMemory/memory(21)       AE162167
/processor_tb/DUT/DP/DataMemory/memory(22)       30D76B0A
/processor_tb/DUT/DP/DataMemory/memory(23)       43192304
/processor_tb/DUT/DP/DataMemory/memory(24)       F6CC1431
/processor_tb/DUT/DP/DataMemory/memory(25)       65046380
/processor_tb/DUT/DP/DataMemory/memory(26)       47A1A366
/processor_tb/DUT/DP/DataMemory/memory(27)       B1004852
/processor_tb/DUT/DP/DataMemory/memory(28)       022CF6CF
/processor_tb/DUT/DP/DataMemory/memory(29)       58525ACF
                                          Now    165000 ns
                                     Cursor 1    164810.888 ns
```

Figure 14. The expected values of S array and Testbench Results

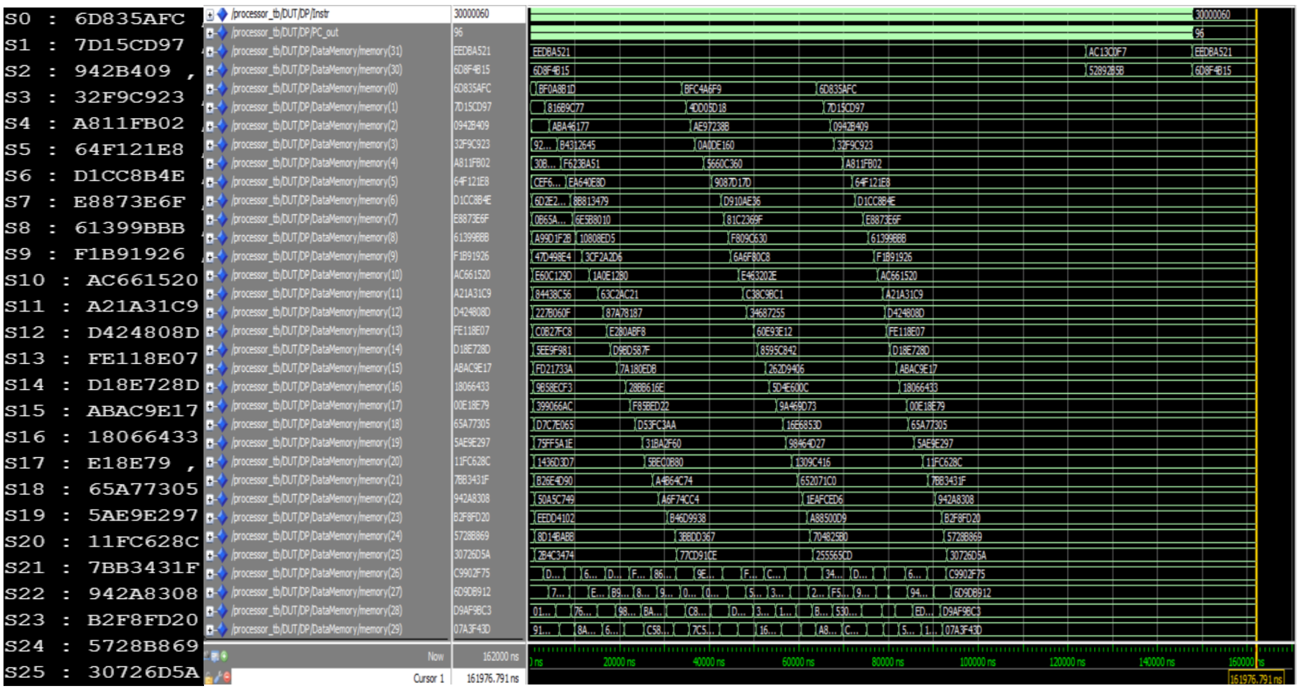
Second Case:

Values shown below are again in little endian.

User Key "91CE A910 01A5 5563 51B2 41BE 1946 5F91", Plain Text: "EEDB A521 6D8F 4B15"

Expected Output For Cipher Text: "AC13 C0F7 5289 2B5B"

Expected values of the S array as outputed by the c algorithm are given in Figure 15. The expected values of S array and Testbench Results along with the outputs of our microprocessor. Memory locations 0 to 25 correspond to the S array, 26 to 29 to the L array and 30 and 31 correspond to B and A respectively. We can see that all the S values match, other than that our microproccesor outputs the correct values after encryption and reverts back to the plaintext after the decypher algorithm.

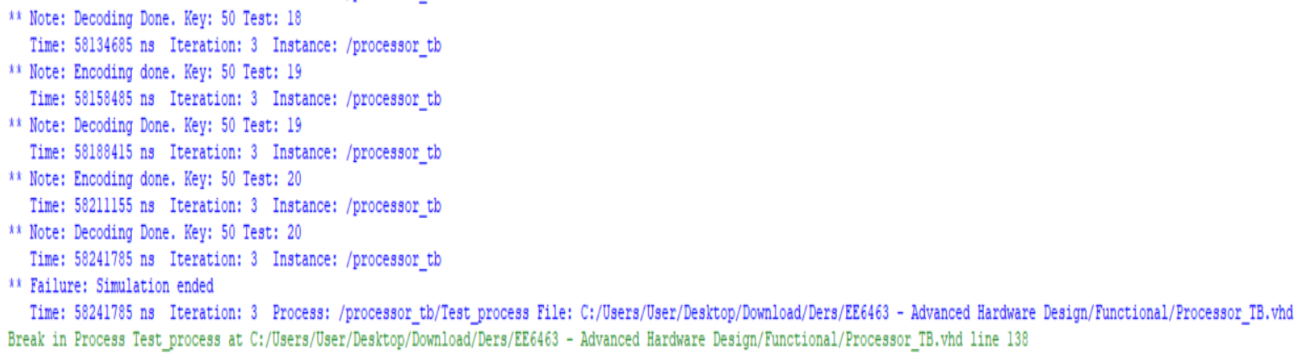


```
S0 : 6D835AFC
S1 : 7D15CD97
S2 : 942B409 ,
S3 : 32F9C923
S4 : A811FB02
S5 : 64F121E8
S6 : D1CC8B4E
S7 : E8873E6F
S8 : 61399BBB
S9 : F1B91926
S10 : AC661520
S11 : A21A31C9
S12 : D424808D
S13 : FE118E07
S14 : D18E728D
S15 : ABAC9E17
S16 : 18066433
S17 : E18E79 ,
S18 : 65A77305
S19 : 5AE9E297
S20 : 11FC628C
S21 : 7BB3431F
S22 : 942A8308
S23 : B2F8FD20
S24 : 5728B869
S25 : 30726D5A
```

```
/processor_tb/DUT/DP/Instr                        30000060
/processor_tb/DUT/DP/PC_out                       96
/processor_tb/DUT/DP/DataMemory/memory(31)        EEDBA521
/processor_tb/DUT/DP/DataMemory/memory(30)        6D8F4B15
/processor_tb/DUT/DP/DataMemory/memory(0)         6D835AFC
/processor_tb/DUT/DP/DataMemory/memory(1)         7D15CD97
/processor_tb/DUT/DP/DataMemory/memory(2)         0942B409
/processor_tb/DUT/DP/DataMemory/memory(3)         32F9C923
/processor_tb/DUT/DP/DataMemory/memory(4)         A811FB02
/processor_tb/DUT/DP/DataMemory/memory(5)         64F121E8
/processor_tb/DUT/DP/DataMemory/memory(6)         D1CC8B4E
/processor_tb/DUT/DP/DataMemory/memory(7)         E8873E6F
/processor_tb/DUT/DP/DataMemory/memory(8)         61399BBB
/processor_tb/DUT/DP/DataMemory/memory(9)         F1B91926
/processor_tb/DUT/DP/DataMemory/memory(10)        AC661520
/processor_tb/DUT/DP/DataMemory/memory(11)        A21A31C9
/processor_tb/DUT/DP/DataMemory/memory(12)        D424808D
/processor_tb/DUT/DP/DataMemory/memory(13)        FE118E07
/processor_tb/DUT/DP/DataMemory/memory(14)        D18E728D
/processor_tb/DUT/DP/DataMemory/memory(15)        ABAC9E17
/processor_tb/DUT/DP/DataMemory/memory(16)        18066433
/processor_tb/DUT/DP/DataMemory/memory(17)        00E18E79
/processor_tb/DUT/DP/DataMemory/memory(18)        65A77305
/processor_tb/DUT/DP/DataMemory/memory(19)        5AE9E297
/processor_tb/DUT/DP/DataMemory/memory(20)        11FC628C
/processor_tb/DUT/DP/DataMemory/memory(21)        7BB3431F
/processor_tb/DUT/DP/DataMemory/memory(22)        942A8308
/processor_tb/DUT/DP/DataMemory/memory(23)        B2F8FD20
/processor_tb/DUT/DP/DataMemory/memory(24)        5728B869
/processor_tb/DUT/DP/DataMemory/memory(25)        30726D5A
/processor_tb/DUT/DP/DataMemory/memory(26)        C9902F75
/processor_tb/DUT/DP/DataMemory/memory(27)        6D9DB912
/processor_tb/DUT/DP/DataMemory/memory(28)        D9AF9BC3
/processor_tb/DUT/DP/DataMemory/memory(29)        07A3F43D
                                          Now     162000 ns
                                     Cursor 1     161976.791 ns
```

Figure 15. The expected values of S array and Testbench Results

After these tests an automated testbench was used to test the design. For these tests 50 different keys were tested with 20 encode operations and 20 decode operations. Assertions with fatal conditions were used to test thus, if there is an error the execution of the testbench would be cut short. A screenshot that the test finished without errors can be seen in Figure 16.



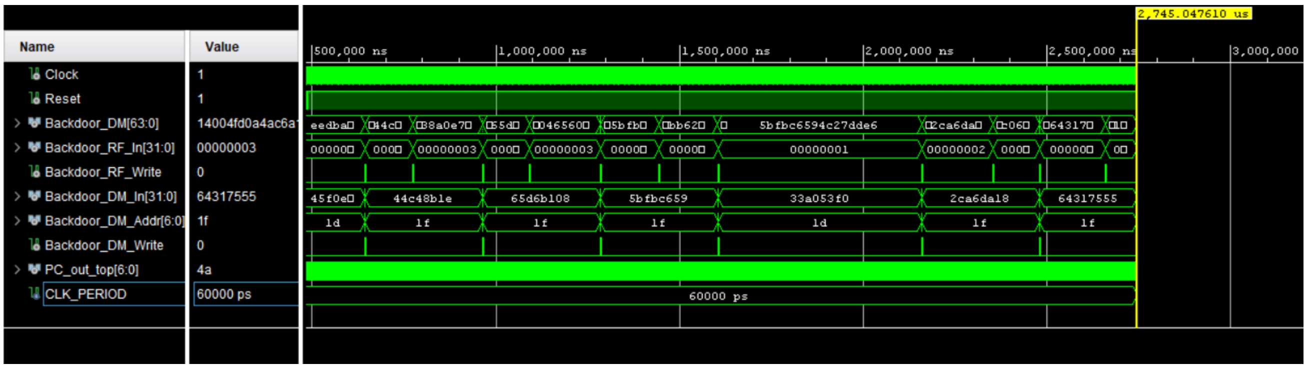
```
** Note: Decoding Done. Key: 50 Test: 18
   Time: 58134685 ns  Iteration: 3  Instance: /processor_tb
** Note: Encoding done. Key: 50 Test: 19
   Time: 58158485 ns  Iteration: 3  Instance: /processor_tb
** Note: Decoding Done. Key: 50 Test: 19
   Time: 58188415 ns  Iteration: 3  Instance: /processor_tb
** Note: Encoding done. Key: 50 Test: 20
   Time: 58211155 ns  Iteration: 3  Instance: /processor_tb
** Note: Decoding Done. Key: 50 Test: 20
   Time: 58241785 ns  Iteration: 3  Instance: /processor_tb
** Failure: Simulation ended
   Time: 58241785 ns  Iteration: 3  Process: /processor_tb/Test_process File: C:/Users/User/Desktop/Download/Ders/EE6463 - Advanced Hardware Design/Functional/Processor_TB.vhd
Break in Process Test_process at C:/Users/User/Desktop/Download/Ders/EE6463 - Advanced Hardware Design/Functional/Processor_TB.vhd line 138
```

Figure 16. Automated Testbench Results

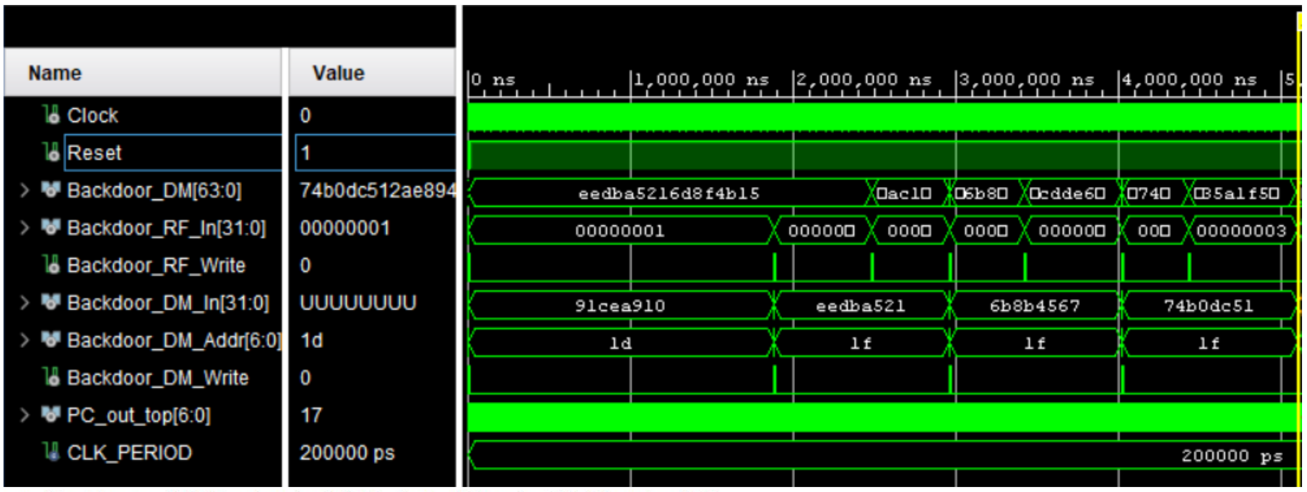
5. Performance Analysis

In this part we will examine the minimum clock period and maximum frequency of the design. In addition to that theorytical and experimental latency calculations will also be done.

Timing simulation results and timing analysis results can be seen in Figure 17. Timing Simulation and Figure 18. Timing Analysis respectively. For the timing simulation a clock with 100 ns period was used. 2 different keys with 3 and 4 inputs was tested.

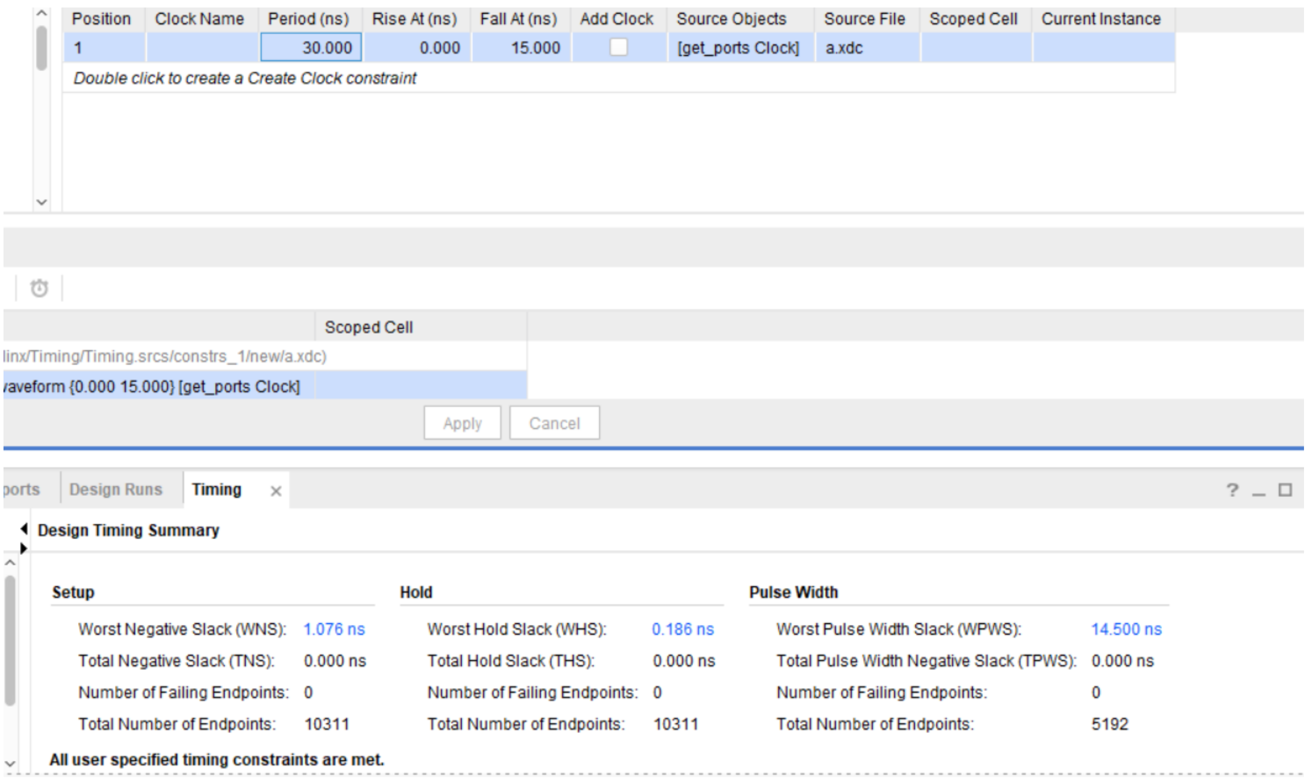


```
run all
Note: Encoding done. Key: 1 Test: 1
Time: 773799320 ps  Iteration: 1  Process: /Processor_TB/Test_process  File: C:/Users/User/Desktop/Download/Ders/EE6463 - Advanced Hardware Design/Functional/Processor_TB.vhd
Note: Decoding Done. Key: 1 Test: 1
Time: 963039874 ps  Iteration: 1  Process: /Processor_TB/Test_process  File: C:/Users/User/Desktop/Download/Ders/EE6463 - Advanced Hardware Design/Functional/Processor_TB.vhd
Note: Encoding done. Key: 1 Test: 2
Time: 1091379320 ps  Iteration: 1  Process: /Processor_TB/Test_process  File: C:/Users/User/Desktop/Download/Ders/EE6463 - Advanced Hardware Design/Functional/Processor_TB.vhd
Note: Decoding Done. Key: 1 Test: 2
Time: 1284939874 ps  Iteration: 1  Process: /Processor_TB/Test_process  File: C:/Users/User/Desktop/Download/Ders/EE6463 - Advanced Hardware Design/Functional/Processor_TB.vhd
Note: Encoding done. Key: 1 Test: 3
Time: 1442259320 ps  Iteration: 1  Process: /Processor_TB/Test_process  File: C:/Users/User/Desktop/Download/Ders/EE6463 - Advanced Hardware Design/Functional/Processor_TB.vhd
Note: Decoding Done. Key: 1 Test: 3
Time: 1602879874 ps  Iteration: 1  Process: /Processor_TB/Test_process  File: C:/Users/User/Desktop/Download/Ders/EE6463 - Advanced Hardware Design/Functional/Processor_TB.vhd
Note: Encoding done. Key: 2 Test: 1
Time: 2353479320 ps  Iteration: 1  Process: /Processor_TB/Test_process  File: C:/Users/User/Desktop/Download/Ders/EE6463 - Advanced Hardware Design/Functional/Processor_TB.vhd
Note: Decoding Done. Key: 2 Test: 1
Time: 2479119874 ps  Iteration: 1  Process: /Processor_TB/Test_process  File: C:/Users/User/Desktop/Download/Ders/EE6463 - Advanced Hardware Design/Functional/Processor_TB.vhd
Note: Encoding done. Key: 2 Test: 2
Time: 2658339320 ps  Iteration: 1  Process: /Processor_TB/Test_process  File: C:/Users/User/Desktop/Download/Ders/EE6463 - Advanced Hardware Design/Functional/Processor_TB.vhd
```



```
run: Time (s): cpu = 00:02:59 ; elapsed = 00:08:47 . Memory (MB): peak = 2495.387 ; gain = 0.000
run all
Note: Encoding done. Key: 1 Test: 1
Time: 2485509320 ps  Iteration: 1  Process: /Processor_TB/Test_process  File: C:/Users/User/Desktop/Download/Ders/EE6463 - Advanced Hardware Design/Functional/Processor_TB.vhd
Note: Decoding Done. Key: 1 Test: 1
Time: 2961309874 ps  Iteration: 1  Process: /Processor_TB/Test_process  File: C:/Users/User/Desktop/Download/Ders/EE6463 - Advanced Hardware Design/Functional/Processor_TB.vhd
Note: Encoding done. Key: 1 Test: 2
Time: 3429109320 ps  Iteration: 1  Process: /Processor_TB/Test_process  File: C:/Users/User/Desktop/Download/Ders/EE6463 - Advanced Hardware Design/Functional/Processor_TB.vhd
Note: Decoding Done. Key: 1 Test: 2
Time: 4027509874 ps  Iteration: 1  Process: /Processor_TB/Test_process  File: C:/Users/User/Desktop/Download/Ders/EE6463 - Advanced Hardware Design/Functional/Processor_TB.vhd
Note: Encoding done. Key: 1 Test: 3
Time: 4435709320 ps  Iteration: 1  Process: /Processor_TB/Test_process  File: C:/Users/User/Desktop/Download/Ders/EE6463 - Advanced Hardware Design/Functional/Processor_TB.vhd
Note: Decoding Done. Key: 1 Test: 3
Time: 5098909874 ps  Iteration: 1  Process: /Processor_TB/Test_process  File: C:/Users/User/Desktop/Download/Ders/EE6463 - Advanced Hardware Design/Functional/Processor_TB.vhd
Error: STD_LOGIC_1164.HREAD End of string encountered
Time: 5100909874 ps  Iteration: 0  Process: /Processor_TB/Test_process  File: C:/Users/User/Desktop/Download/Ders/EE6463 - Advanced Hardware Design/Functional/Processor_TB.vhd
```

Figure 17. Timing Simulation



Position	Clock Name	Period (ns)	Rise At (ns)	Fall At (ns)	Add Clock	Source Objects	Source File	Scoped Cell	Current Instance
1		30.000	0.000	15.000	☐	[get_ports Clock]	a.xdc		
Double click to create a Create Clock constraint									

	Scoped Cell
linx/Timing/Timing.srcs/constrs_1/new/a.xdc)	
/aveform {0.000 15.000} [get_ports Clock]	

Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	1.076 ns	Worst Hold Slack (WHS):	0.186 ns	Worst Pulse Width Slack (WPWS):	14.500 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:	10311	Total Number of Endpoints:	10311	Total Number of Endpoints:	5192

All user specified timing constraints are met.

Figure 18. Timing Analysis

Ledout signal corresponds to the lower 16 bits of Register 30 on the register file and it is transferred to outside at each rising edge. This register was chosen to be demonstrated because It is the first register that is loaded with a value on the first instruction. For the timing simulation after reset at the second clock, value we provide to the register is observed. The 1 clock delay is because vivado buffered the clock signal. It would take too long of a time to actually see the value of A or B because the microprocessor takes more than 10000 cycles to process through the whole encryption algorithm. Our timing analysis also shows that at 14ns period we meet our timing requirements.

Minimum Clock Period: 30 ns

Maximum Frequency: 33.3 MHz

Latency:

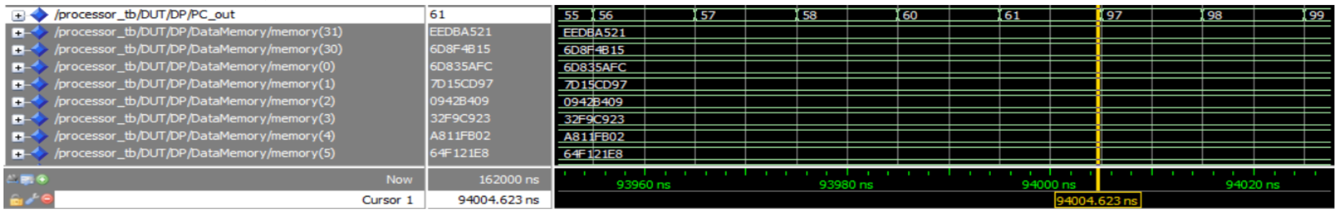
Finding out the exact latency is impossible with this arrangement because It is rotate dependent. Since we do not have rotate operations on our processor, each rotate consists of shifts and corrections on the number if necessary. This means for each rotate we have 0 to 31 loops depending on the rotate amount. During the simulation a clock with a 10ns period was used.

Theoritical Case Key Generation:

Operation	Clock Cycles	Repetitions	Total Clocks
Initialization of Key Generation	10	1	10
Populating S with initial values	4	26	104
Initialization for Mixing the key	4	1	4
The mixing in the key loop without the shifts	19	78	1482
Modulus overhead i	2	3	6
Modulus overhead of j	1	19	19
Shift S	5	234	1170
Shift S correction overhead	1	0 to 78 times 3	0-234
Shift L	5	0 to 78 times 31	0-12090
Shift L correction overhead	1	0 to 78 times 31	0-2418
Total Ranges			2795-17537
Average Case			10166

As we see from our calculation our key generation algorithm will take 2795 to 17537 times depending on the amount of rotations dependent on the inputs. It is nearly impossible to predict how a input pattern will behave without running it.

Experimental Case Key Generation:



```
/processor_tb/DUT/DP/PC_out                     61
/processor_tb/DUT/DP/DataMemory/memory(31)      EEDBA521
/processor_tb/DUT/DP/DataMemory/memory(30)      6D8F4B15
/processor_tb/DUT/DP/DataMemory/memory(0)       6D835AFC
/processor_tb/DUT/DP/DataMemory/memory(1)       7D15CD97
/processor_tb/DUT/DP/DataMemory/memory(2)       0942B409
/processor_tb/DUT/DP/DataMemory/memory(3)       32F9C923
/processor_tb/DUT/DP/DataMemory/memory(4)       A811FB02
/processor_tb/DUT/DP/DataMemory/memory(5)       64F121E8
Now                                             162000 ns
Cursor 1                                        94004.623 ns
```

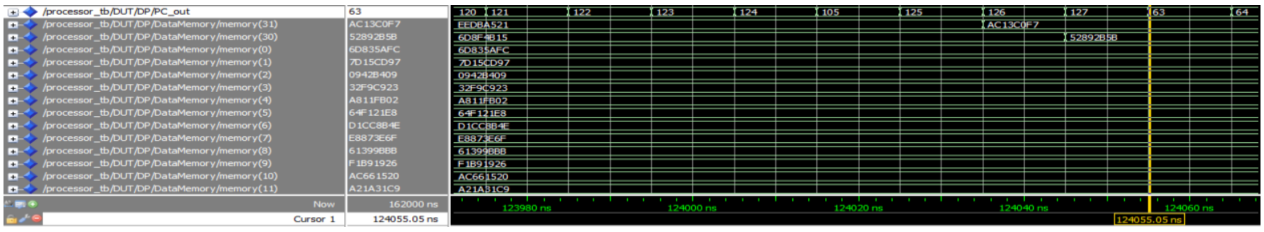
When the timing value for key generation was observed for test case 2 it was seen that it took our proccessor 9400 clock cycles to generate the S array. This value is close to average and is inside our limits which we expected.

Theoritical Case Encoder:

Operation	Clock Cycles	Repetitions	Total Clocks
Initialization of Encoder	9	1	9
Encoder without the shifts	12	24	288
Encoder shifts	5	0 to 24 times 31	0-3720
Shift Correction	2	0 to 24 times 31	0-1488
Total Ranges			297-5505
Average Case			2901

As we see from our calculation our encoder algorithm will take 297 to 5505 times depending on the amount of rotations dependent on the inputs. It is nearly impossible to predict how a input pattern will behave without running it.

Experimental Case Encoder:



```
/processor_tb/DUT/DP/PC_out                      63         120 121 122 123 124 105 125 126 127 63 64
/processor_tb/DUT/DP/DataMemory/memory(31)       AC13C0F7   EEDBA521 AC13C0F7
/processor_tb/DUT/DP/DataMemory/memory(30)       52892B5B   6D8F4B15 52892B5B
/processor_tb/DUT/DP/DataMemory/memory(0)        6D835AFC   6D835AFC
/processor_tb/DUT/DP/DataMemory/memory(1)        7D15CD97   7D15CD97
/processor_tb/DUT/DP/DataMemory/memory(2)        0942B409   0942B409
/processor_tb/DUT/DP/DataMemory/memory(3)        32F9C923   32F9C923
/processor_tb/DUT/DP/DataMemory/memory(4)        A811FB02   A811FB02
/processor_tb/DUT/DP/DataMemory/memory(5)        64F121E8   64F121E8
/processor_tb/DUT/DP/DataMemory/memory(6)        D1CC8B4E   D1CC8B4E
/processor_tb/DUT/DP/DataMemory/memory(7)        E8873E6F   E8873E6F
/processor_tb/DUT/DP/DataMemory/memory(8)        61399BBB   61399BBB
/processor_tb/DUT/DP/DataMemory/memory(9)        F1B91926   F1B91926
/processor_tb/DUT/DP/DataMemory/memory(10)       AC661520   AC661520
/processor_tb/DUT/DP/DataMemory/memory(11)       A21A31C9   A21A31C9
                                         Now     162000 ns  123980 ns  124000 ns  124020 ns  124040 ns  124060 ns
                                    Cursor 1     124055.05 ns                                 124055.05 ns
```

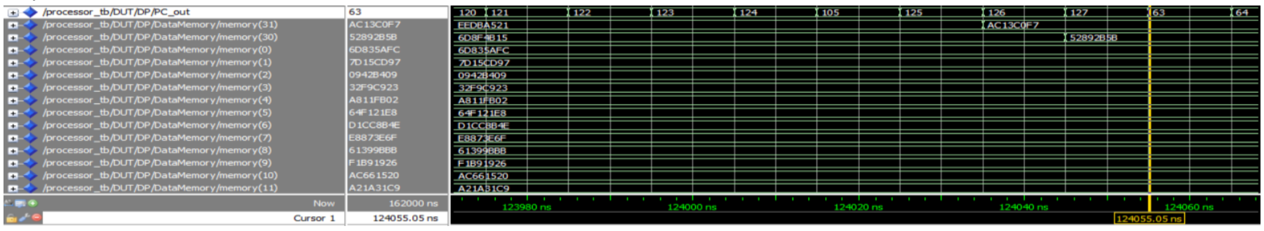
When the timing value for key generation was observed for test case 2 it was seen that it took our proccessor 12405-9400=3005 clock cycles to generate ciphertext. This value is less than average and is inside our limits which we expected. Since this is less than average it means for most of the operation there was left shifts less than the average case.

Theoritical Case Decoder:

Operation	Clock Cycles	Repetitions	Total Clocks
Initialization of Decoder	5	1	5
Decoder without the shifts	14	24	336
Encoder shifts	5	0 to 24 times 31	0-3720
Shift Correction	2	0 to 24 times 31	0-1488
Total Ranges			341-5549
Average Case			2945

As we see from our calculation our encoder algorithm will take 297 to 5549 times depending on the amount of rotations dependent on the inputs. It is nearly impossible to predict how a input pattern will behave without running it.

Experimental Case Decoder:

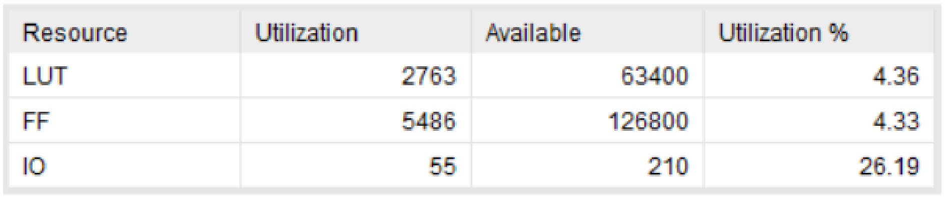


```
/processor_tb/DUT/DP/PC_out                      63         120 121 122 123 124 105 125 126 127 63 64
/processor_tb/DUT/DP/DataMemory/memory(31)       AC13C0F7   EEDBA521 AC13C0F7
/processor_tb/DUT/DP/DataMemory/memory(30)       52892B5B   6D8F4B15 52892B5B
/processor_tb/DUT/DP/DataMemory/memory(0)        6D835AFC   6D835AFC
/processor_tb/DUT/DP/DataMemory/memory(1)        7D15CD97   7D15CD97
/processor_tb/DUT/DP/DataMemory/memory(2)        0942B409   0942B409
/processor_tb/DUT/DP/DataMemory/memory(3)        32F9C923   32F9C923
/processor_tb/DUT/DP/DataMemory/memory(4)        A811FB02   A811FB02
/processor_tb/DUT/DP/DataMemory/memory(5)        64F121E8   64F121E8
/processor_tb/DUT/DP/DataMemory/memory(6)        D1CC8B4E   D1CC8B4E
/processor_tb/DUT/DP/DataMemory/memory(7)        E8873E6F   E8873E6F
/processor_tb/DUT/DP/DataMemory/memory(8)        61399BBB   61399BBB
/processor_tb/DUT/DP/DataMemory/memory(9)        F1B91926   F1B91926
/processor_tb/DUT/DP/DataMemory/memory(10)       AC661520   AC661520
/processor_tb/DUT/DP/DataMemory/memory(11)       A21A31C9   A21A31C9
                                         Now     162000 ns  123980 ns  124000 ns  124020 ns  124040 ns  124060 ns
                                    Cursor 1     124055.05 ns                                 124055.05 ns
```

When the timing value for key generation was observed for test case 2 it was seen that it took our proccessor 14777-12405=2372 clock cycles to generate ciphertext. This value is less than average and is inside our limits which we expected. Since this is less than average it means for most of the operation there was left shifts less than the average case.

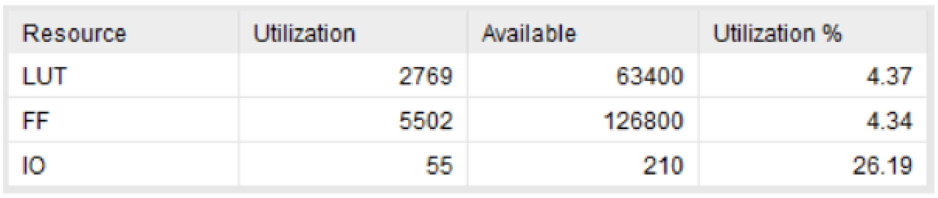
6. Area Analysis

The resource utilization after synthesis and implementation are given in Figure 19. Resource Utilization After Synthesis and Figure 20. Resource Utilization After Implementation respectively.



Resource	Utilization	Available	Utilization %
LUT	2763	63400	4.36
FF	5486	126800	4.33
IO	55	210	26.19

Figure 19. Resource Utilization After Synthesis



Resource	Utilization	Available	Utilization %
LUT	2769	63400	4.37
FF	5502	126800	4.34
IO	55	210	26.19

Figure 20. Resource Utilization After Implementation

We can see that since we have 2 different memories and a register file that is constructed of lots of registers, our Flip Flop usage is higher than our Look up table usage. We are not using many resources and our chip could fit nearly 19 of similar processors inside with the space that is left.

7. Verification Details

For verification of the microprocessor, first a set of instructions were supplied that covered every operation and the register values were checked to verify each and every operation worked fine.

After this encryption instructions were implemented and tested with test case 1 first. At this stage first fixes were made to first iteration of the loop. I found some mistakes that made it so that with zero shift amount the shift would produce wrong results which was fixed. After fixing the first iteration all the values in the middle was as they should be also checked by the output of the c code. However, the last stage would give wrong results or sometimes woud go infinite. This was related to the checking of end condition which was also corrected after detailed observations.

Following the encryption, decryption was implemented. This cycle also had nearly the similar verification cycle because it is the reciprocal of the encryption.

Key generation was the hardest function to debug because it has many loops that are inside eachother and has fixed and data dependent rotations. However, by also following through similar procedures I was able to debug and verify the system with outputs of c code and observation.

Finally when a case that was different than all zeros were used, I would get wrong results. After reading through the RC5 Paper many times I realized the plaintext, ciphertext and user key were provided in Big Endian structure and my processor used little endian on it. After fixing this problem the overall system was working well. However, after running the same function more than once It would provide wrong values. This was because some of the registers had different values than their reset values and they were not being loaded with reset values after their use. This was also an easy fix by adding necessary initialization instructions.

In addition to all these an automated testbench was ran for 50 different keys each with 20 encode and decode operations. The results of this was also explained on the functional simulation.

For timing simulation automated testbenches were run and 8 encode and 7 decode simulations with 2 keys were completed and all of them passed the test.

8. Interface Explaination

To control the processor and provide the inputs we are using the 16 switches and 6 push buttons present on the Nexys 4 device. If there will be a backdoor write to one of the registers or the memory, first the value is loaded with the help of 2 push buttons. One of the buttons load the value on the switch to lower 16 bits and the other one loads it to higher 16 bits. By this way 32 bit data is constructed (To verify also shown on the 7 segment display) and then loaded into the device. When storing the value the switches provide the address to which the store will be done and since we want to access 6 places of Data memory and 8 registers other switches are used as function bits that determine if the value will be stored on a register or memory. Other than that with the help of the function bits instead of storing we can load a register file or memory value to show the contents on the 7 segment display. Finally to determine if the one instruction mode or free runnning mode is used another push button is used to switch between. These different run mode information is also be displayed on leds.

9. Future Optimizations

There are couple of things that can be improved on the design. First, the assembly code can be more efficent even though it is a clever implementation. This is especially true for the initialization of some numbers for the loops. The ones that are the same on different functions could be set on startup and left like that. Second, the interface could probably implemented in a more efficent way. This was the solution I found since I had limited time. I would also add some important logical functions to the ALU like Xor and some shift and rotate types of instructions.

10. Conclusion

In this report the details of the implementation of NYU processor were elaborated. Overall, the project was a fun challenge for me and made me think in more clever ways in assembly language on top of the hardships of implementing a sequential design in VHDL. In addition to that since the outputs of RC5 algorithm is quite random to human perception learning how to test the algorithm also taugth me a lot. All in all, this project taught me how to implement complex algorithms in a very limited processor and finding tricks to increase the efficiency of an algorithm.

11. References

- Rivest R.L. (1994). The RC5 Encryption Algorithm. FSE.