ADVANCED HARDWARE DESIGN
PROCESSOR 51
FINAL PROJECT REPORT

Chaitanya R. Shah-crs562

Chaitanya M. Pattewar- cmp694

Esha Sarkar- es4211

Namrata Vichare-nav272

Contents

1. Block Diagram of Processor 51

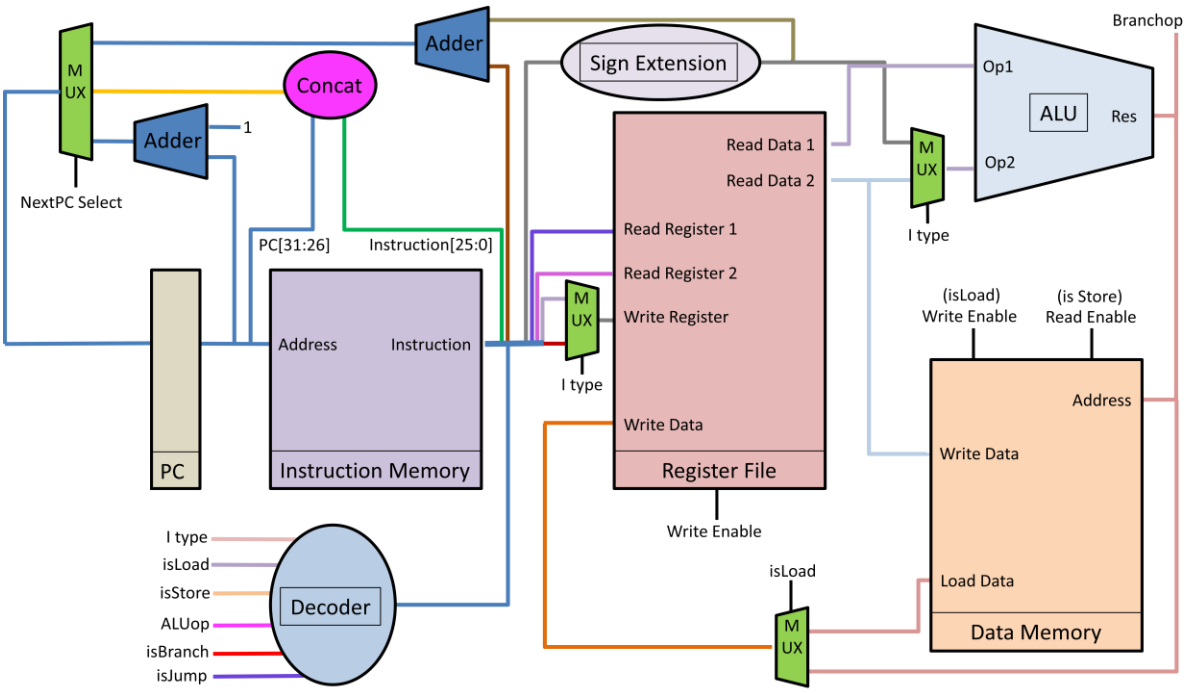


Fig: Block diagram of Processor

2. Processor Components

2.1 Program Counter

- It is a 32- bit register, incrementing its value every clock cycle.
- It points to the address of the next instruction to be executed.
- As soon as it executes the halt instruction, it stops incrementing (the system freezes).
- Implementation on Processor in hardware- Like a counter with conditions of jump and branch instructions

2.2 Instruction Memory

- It stores instructions (in machine codes) to be executed.
- It obtains the address as an input from the Program Counter and gives the instruction as the output written at the specified address.
- The size of the instruction memory is 2KB
- Implementation-Like a ROM which has hardcoded instructions

2.3 Arithmetic Logic Unit (ALU)

- It is responsible for all computations in the processor. It performs logical, arithmetic, shift and branch operations.
- Logical instructions such as and, or, etc., arithmetic instructions such as add, subtract, etc., shift left and right and branch instructions as branch if equal, if not equal, etc. are implemented by the ALU.
- The inputs to ALU depend on the type of instruction. For R-type and branch, they are register files, for I-type, it is one register file, for jump it is only the address
- The output of the ALU is a 32-bit result which is either an operation result or an address calculation. For branch instructions, it acts as a comparator.

- Implementation: It is a component which has all the operations with output selected on the basis of instruction.

2.4 Decoder

- The input to the decoder unit is a 32-bit instruction from the instruction memory.
- With the first 6 bits (Opcode) and the last 6 bits (Function code) of the instruction, it determines the type of instruction and the flags are generated.
- Implementation- Decoder

2.5 Register File

- Input Ports- 2 input Read Register ports to read the 5 bit address, 1 Write register address port and 1 input port for the 32 bit data to be written in the register file
- Output Ports- 2 output ports which give the 32 bit values of the read registers
- Implementation- RAM:Array of 32 registers of size 32 bits.

3. Component block diagrams and simulations

3.1 ALU Block Diagram

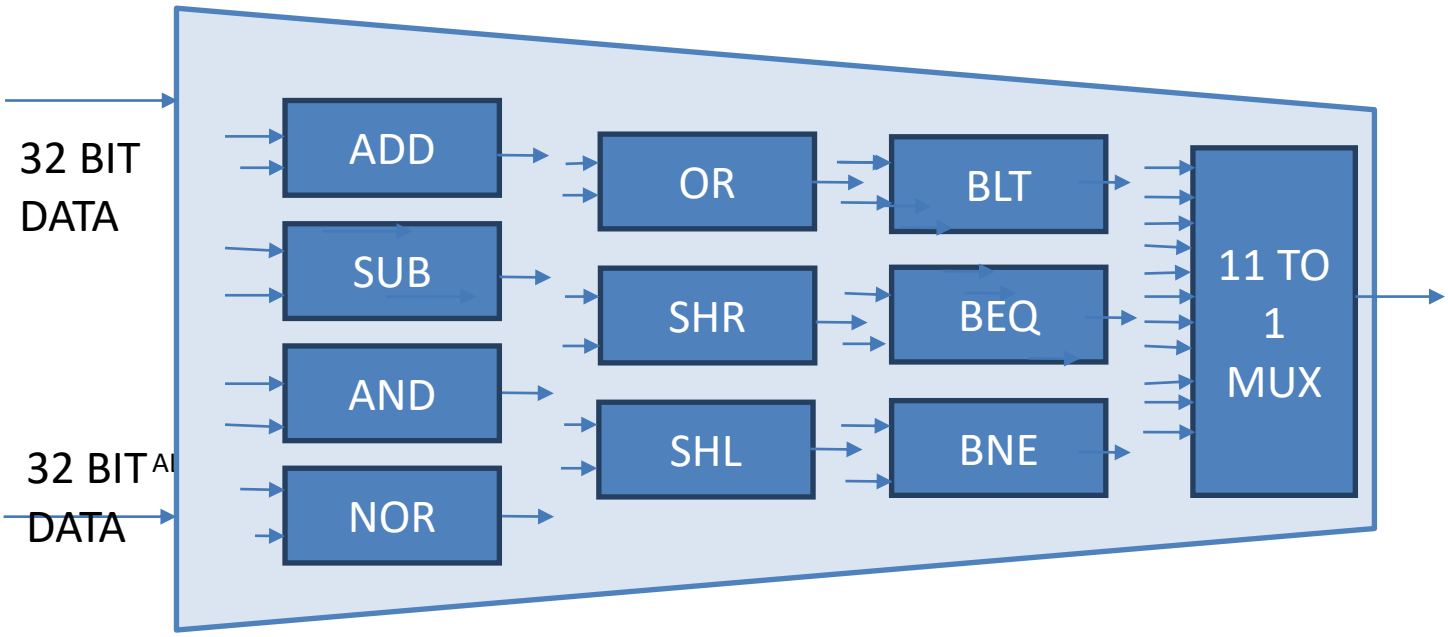


Fig 1:ALU Block Diagram

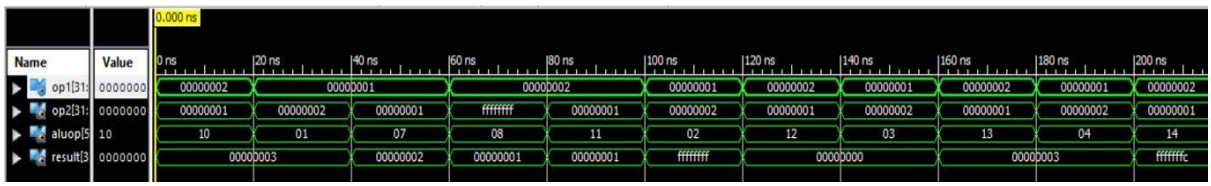


Fig 2: ALU Simulation

Instruction	Operand 1	Operand 2	Result
Add	00000002	00000001	00000003
Addi	00000001	00000002	00000003
Lw	00000001	00000001	00000002
Sw	00000002	Ffffffff	00000001
Sub	00000002	00000001	00000001
Subi	00000001	00000002	ffffffff
And	00000002	00000001	00000000
Andi	00000001	00000002	00000000
Or	00000002	00000001	00000003
Ori	00000001	00000002	00000003
Nor	00000002	00000001	fffffffc

Table 1: Instructions used for ALU simulation

3.2Decoder Unit

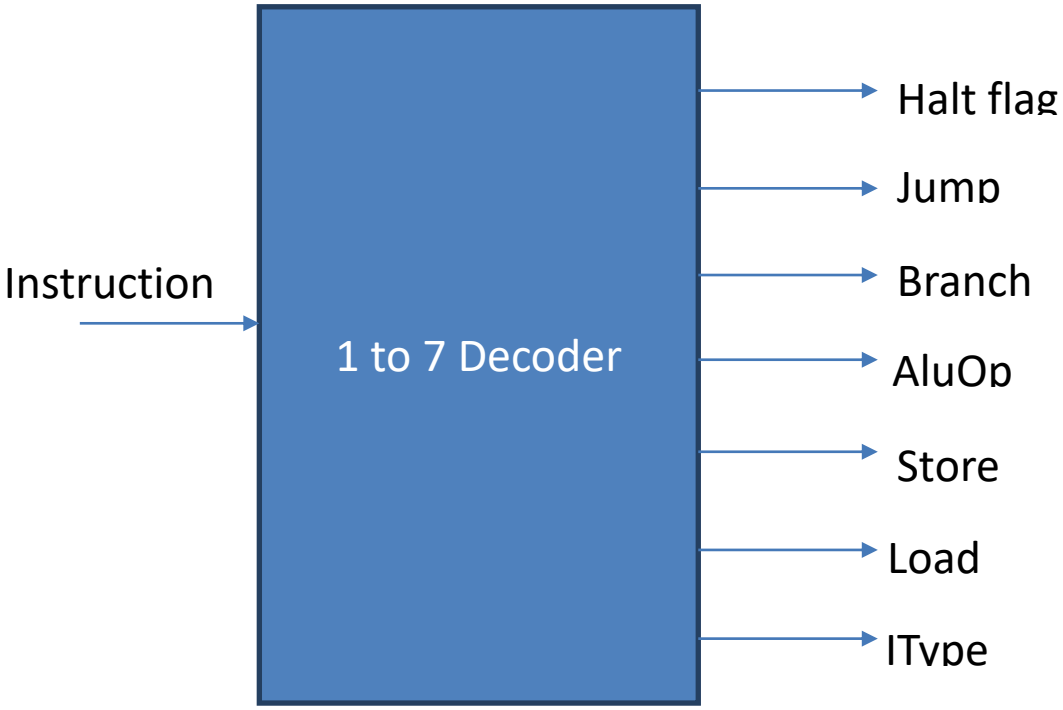


Fig 3: Decoder Block Diagram

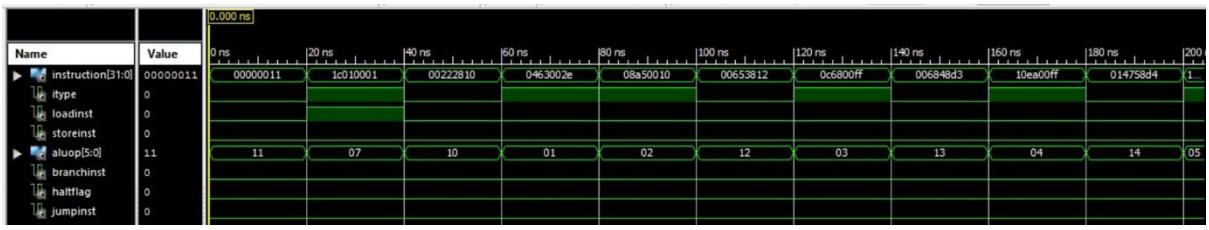


Fig 4: Decoder Simulation

4. Verification of Processor design

4.1 Assembly Code of Test Program:

```
sub, 0,0,0
lw, 0,1,1
lw, 1,2,1
lw, 0,3,3
lw, 0,4,4
add, 1,2,5
sub, 3,4,6
addi, 3,3,2E
subi, 5,5,10
and, 3,5,7,
andi, 3,8,FF
or, 3,8,9
ori, 7,10,FF
nor, 10,7,11
shl, 11,12,1
shr, 12,13,3
blt, 0,1,1,
lw, 0,31,1,
bne, 0,2,1,
lw, 0,30,2,
beq, 0,0,1
lw, 0,29,3,
sw, 0,5,3,
jmp, 1
```

4.2 Machine code of Test Program

```
X"00000011", X"1C010001", X"1C220001", X"1C030003", X"1C040004",
X"00222810", X"00643011", X"0463002E", X"08A50010", X"00653812",
X"0C6800FF", X"006848D3”, X"10EA00FF", X"014758D4", X"156C0001",
X"198D0003", X"24010001", X"1C1F0001", X"2C020001", X"1C1E0002",
X"28000001", X"1C1D0003", X"20050003", X"30000001"
```

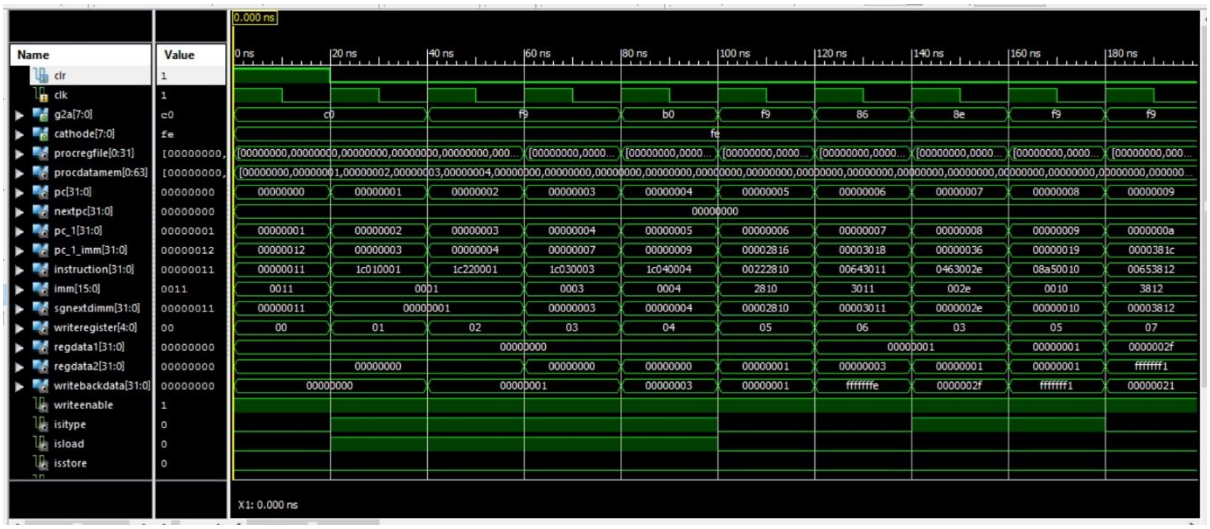


Fig 5: Simulation of the test program

5. Sample programs

5.1 Sample Program 1

```
ADDI R1, R0, 7 // R1 = 7
ADDI R2, R0, 8 // R2 = 8
ADD R3, R1, R2 // R3 = R1 + R2 =15
HAL // HALT
```

Fig 6: Simulation of Sample Program 1

5.2 Sample Program 2

```
ADDI R1, R0, 2 // R1=2
ADDI R3, R0, 10 // R3=10D (D=Decimal)
ADDI R4, R0, 14 // R4=14D
ADDI R5, R0, 2 // R5=2
SW R4, 2(R3) // 14D is stored in Data memory location 12D
SW R3, 1(R3) // 10D is stored in Data memory location 11D
SUB R4, R4, R3
SUBI R4, R0, 1
AND R4, R2, R3
```

```
ANDI R4, R2, 10
OR R4, R2, R3
LW R2, 1(R3) // R2=10D (Loaded back from memory)
ORI R4, R2, 10
NOR R4, R2, R3
SHL R4, R2, 10
SHR R4, R2, 10
BEQ R5, R0, -2
BLT R5, R4, 0
BNE R5, R4, 0 JMP 20
HAL
```

Fig 7: Simulation of Sample Program 2

6. Instruction Set Architecture

The instruction set architecture used in the processor and the assembler is a 3-address type and it follows the following syntax and convention in writing assembly code:

R-type instruction: Operation, Operand 1 (Rs) , Operand 2 (Rt), Operand 3 (Rd)
For example: add r1,r2,r3 means the addition of r1 and r2 is stored in r3

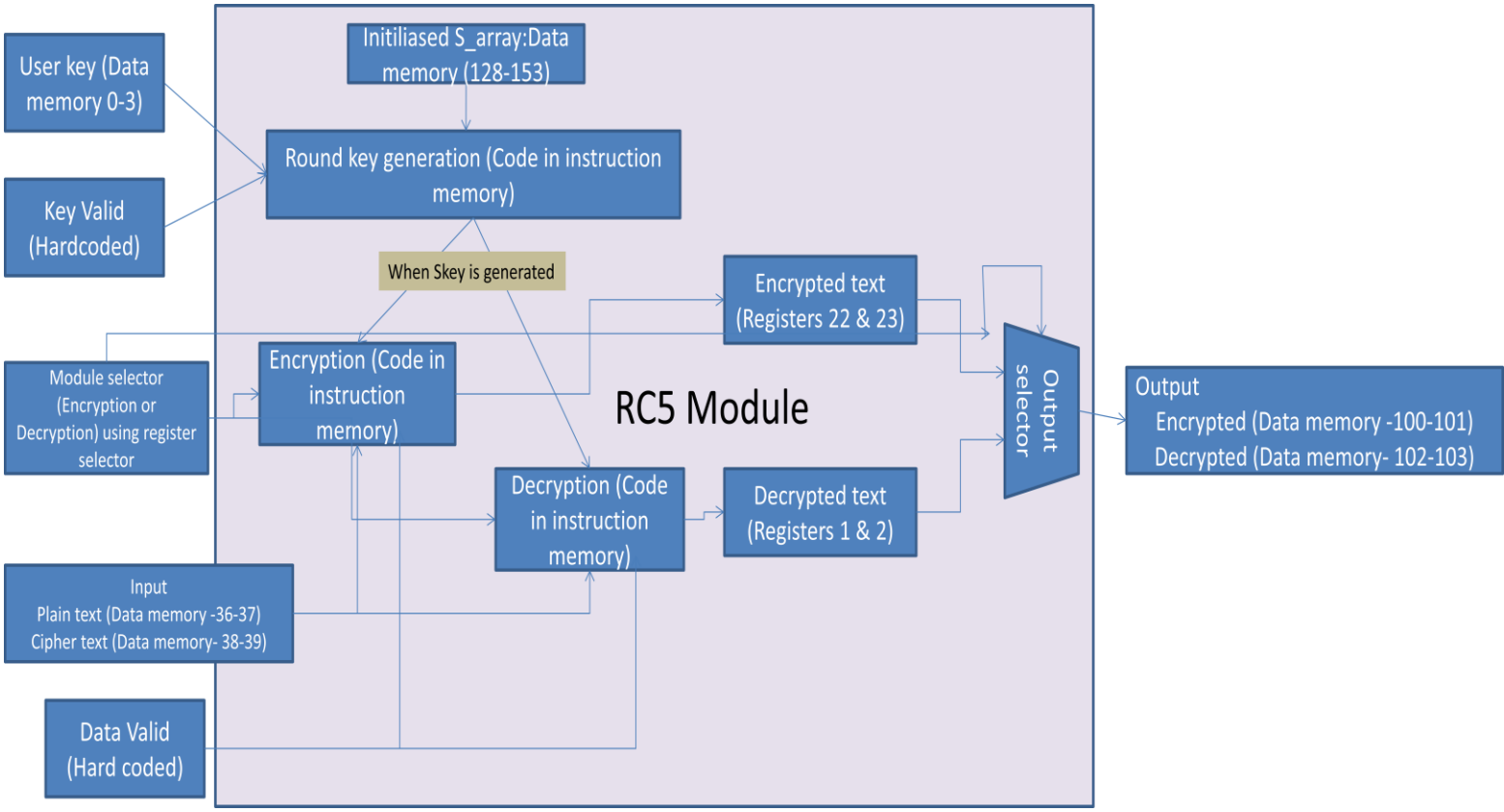
I-type instruction: Operation, Operand 1(Rs), Operand 2(Rd), Immediate
For example: addi r1, r2, 2 means that 2 is added with value of r1 and stored in r2

J-type instruction: Operation, 26 bit address
For example: jmp 4

7. High level description of RC5 implementation

- Initial state-
  - User_key: stored in locations 0-3 of data memory
  - L-array: stored in locations 4-7 of data memory
  - Skey(generated): stored in location 128-153 of data memory
  - Initialised S_array: stored in locations 10-35 of data memory
  - DIN for encryption is stored in 36-37 of data memory
  - DIN for decryption is stored in 38-39 of data memory

- The initial values of S array and L array are copied to the working memory locations which are 128-153 for skey and 4-7 for L-array, using load and store instructions so that the initial values are not messed.
- The arithmetic instructions are done using R-type instructions
- Rotation is done using shifting the required register in the desired direction by rotate amount and shifting it in the opposite direction for (32-rotate amount) and ORing these two values
- Since we do not have enough registers, so as soon as one L-array value and one S-array value is generated, it is immediately stored in the data memory.
- The loops are implemented using branch instructions.
- After k (loop counter for round) loops for 78, encryption begins.
- For Encryption or Decryption the skey values are loaded from memory locations 128-153 which are actually the generated values from round key generation
- Since XOR instruction is not present in the list of instructions, these operations of Encryption and decryption are performed using the basic definition of XOR which is
  A XOR B= (A AND (B NOR B)) OR ((A NOR A) AND B)
- The other operations are similar to those used in Round key generation and follow he same principle.
- Finally, when Encryption and Decryption are performed, the final results are displayed in register file (on FPGA) and also loaded back to Data memory to keep a record



The following is the block diagram for RC5 running on the processor

8. RC 5

This section is arranged in such a way that if one needs to perform only one of the three modules, individual codes are given and finally the consolidated code of entire RC5 is mentioned.

8.1 Encryption

8.1.1 Assembly code

```
encrypt_start:sub r0, r0, r0 //flush register 0
lw r0, r1, 24 //Din value is stored in 36 = A
lw r0, r2, 80 //first value of skey, S[0] is stored in r2
add r1, r2, r1 //Now r1 has value of A_pre=A+ S[0]
lw r0, r2, 25 //Din value is stored in 37 = B
lw r0, r3, 81 //second value of S[1]
add r2, r3, r2 // Now r2 has the value of B_pre=B+S[1]
sub r3, r3, r3 //flush register 3, to be used as a max value of i
addi r0, r3, D // Load counter register r3 with 13 as a blt counter
will be run
sub r31, r31, r31
addi r31, r31, 1 //initialise i with 1
sub r30, r30, r30 //flush r30
addi r30, r30, 81 // initialise corresoding to the memory position of
s[1]
encryption:nor r1, r1, r4 //A'
nor r2, r2, r5 //B'
and r1, r5, r6 //AB'
and r2, r4, r7 //A'B
or r6, r7, r4 // A xor B in r4
shl r2, r5, 1B // shift left 'B' by 27 bits
shr r5, r5, 1B // shift right 'B' by 27 bits. Now, r5 has the rotate
amount
sub r7, r7, r7 //rotation aide counter
beq r5, r7, don't rotate for A
sub r6, r6, r6// assign the left shift register as r6
sub r9, r9, r9// assign the right shift register as r9
add r0, r4, r6
add r0, r4, r9
shift_1:shl r6, r6, 1 //shift A XOR B by 1 bit
addi r7, r7, 1
blt r7, r5, shift_1 //keep on shifting left untill it is equal to shift
amount
sub r8, r8, r8 //rotation aide counter
sub r10, r10, r10
addi r10, r10, 20
sub r10, r5, r11 //store 32-rotate amount in r11
shift_2:shr r9, r9, 1 //shift A XOR B by 1 bit
addi r8, r8, 1
blt r8, r11, shift_2 //keep on shifting right untill it is equal to 32-
shift amount
or r6, r9, r4 //A XOR B <<<< B
don't rotate for A: addi r30, r30, 1
lw r30, r20, 0 // load Skey values for A
add r4, r20, r1 //Final value of A
nor r1, r1, r4 //A'
nor r2, r2, r5 //B'
and r1, r5, r6 //AB'
and r2, r4, r7 //A'B
```

```
or r6, r7, r4 // B xor A in r4
shl r1, r5, 1B // shift left 'A' by 27 bits
shr r5, r5, 1B // shift right 'A' by 27 bits. Now, r5 has the rotate
amount
sub r7, r7, r7 //rotation aide counter
beq r5, r7, don't rotate for B
sub r6, r6, r6// assign the left shift register as r6
sub r9, r9, r9// assign the right shift register as r9
add r0, r4, r6
add r0, r4, r9
shift_3:shl r6, r6, 1 //shift B XOR A by 1 bit
addi r7, r7, 1
blt r7, r5, shift_3 //keep on shifting left untill it is equal to shift
amount
sub r8, r8, r8 //rotation aide counter
sub r10, r10, r10
addi r10, r10, 20
sub r10, r5, r11 //store 32-rotate amount in r11
shift_4:shr r9, r9, 1 //shift A XOR B by 1 bit
addi r8, r8, 1
blt r8, r11, shift_4 //keep on shifting right untill it is equal to 32-
shift amount
or r6, r11, r4 //B XOR A <<<< A
don't rotate for B:addi r30, r30, 1
lw r30, r21, 0 // load Skey values for A
add r4, r21, r2 //Final value of B
addi r31, r31, 1
blt r31, r3, encryption
sw r0, r1, 64 //store the upper half of encrypted output in 100
sw r0, r2, 65 //store the upper half of encrypted output in 101
hal //Encryption over for only encryption
```

7.1.2 Machine Code

```
X"00000011",X"1C010024",X"1C020080",X"1C030025",X"1C040081",X"00220890"
,X"00641090",X"00631891",X"0403000D",X"03FFF811",X"07FF0001",X"03DEF011
",X"07DE0081",X"00212094",X"00422894",X"00253092",X"00443892",X"00C7209
3",X"1445001B",X"18A5001B",X"00E73811",X"28A7000F",X"00C63011",X"012948
11",X"00043010",X"00044810",X"14C60001",X"04E70001",X"24E5FFFD",X"01084
7D1",X"014A57D1",X"054A0020",X"01455811",X"19290001",X"05080001",X"250B
FFFD",X"00C927D3",X"07DE0001",X"1FD40000",X"00940810",X"00212014",X"004
22814",X"00253012",X"00443812",X"00C72013",X"1425001B",X"18A5001B",X"00
E73811",X"28A7000F",X"00C63011",X"01294811",X"00043010",X"00044810",X"C
60001",X"04E70001",X"24E5FFFD",X"010847D1",X"014A57D1",X"054A0020",X"01
455811",X"19290001",X"05080001",X"250BFFFD",X"00C927D3",X"07DE0001",X"1
FD50000",X"00951010",X"07FF0001",X"27E3FFC8",X"20010064",X"20020065",X"
FC000000",others=>(others=>'0')
```

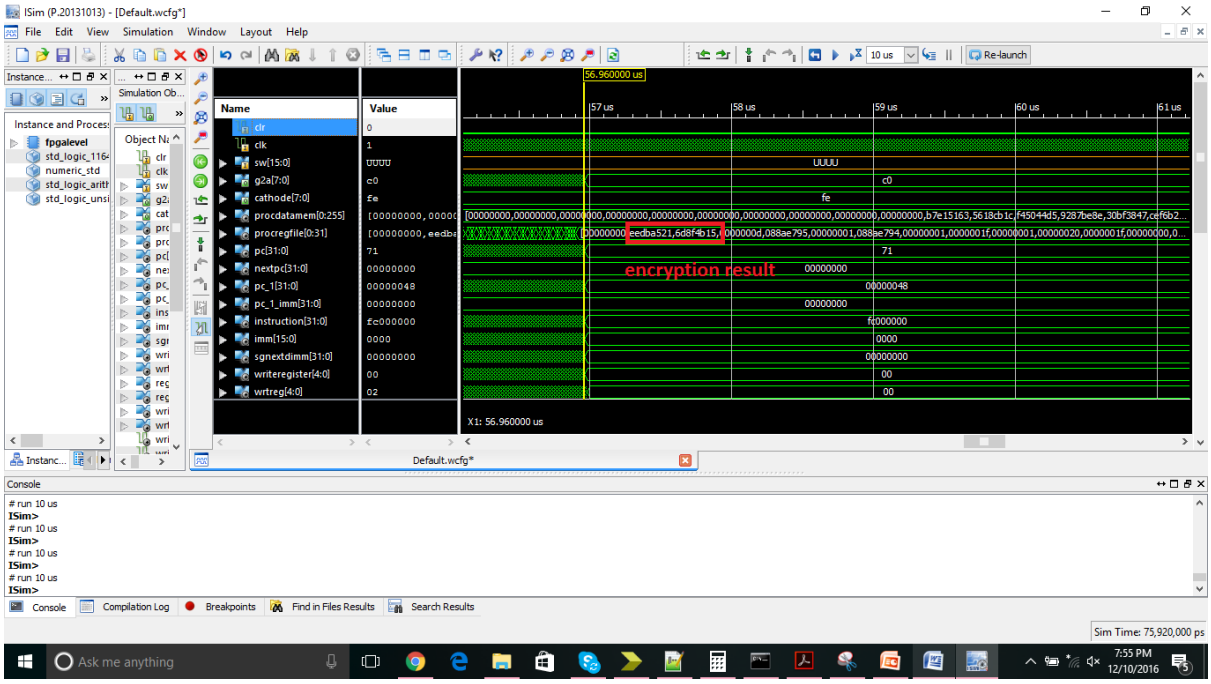


Fig 8: Simulation of RC5 Encryption

8.2 Decryption

8.2.1 Assembly code

```
decrypt_start: sub r0, r0, r0 //flush register 0
lw r0, r1, 26 //Din value is stored in 38 = A
lw r0, r2, 27 //Din value is stored in 39 = B
sub r29, r29, r29
addi r29, r29, 1
sub r3, r3, r3 //flush register 3, to be used as a counter
addi r0, r3, C // Load counter register r3 with 12 as a blt counter
will be run
sub r31, r31, r31
addi r31, r31, 1 //initialise i with 1
sub r30, r30, r30 //flush r30
addi r30, r30, 99 // initialise corresoding to the memory position of
S[25]
decryption: lw r30, r21, 0 // load Skey values for B
sub r2, r21, r17 //B-S[2i+1]
subi r30, r30, 1 //2i
shl r1, r5, 1B // shift left 'A' by 27 bits
shr r5, r5, 1B // shift right 'A' by 27 bits. Now, r5 has the rotate
amount
sub r7, r7, r7 //rotation aide counter
beq r5, r7, don't rotate for B
sub r6, r6, r6// assign the left shift register as r6
sub r9, r9, r9// assign the right shift register as r9
add r0, r17, r6
add r0, r17, r9
shift_5:shr r6, r6, 1 //shift B-S[2i+1] by 1 bit
addi r7, r7, 1
blt r7, r5, shift_5 //keep on shifting right untill it is equal to
shift amount
sub r8, r8, r8 //rotation aide counter
```

```
sub r10, r10, 10
addi r10, r10, 20
sub r10, r5, r11 //store 32-rotate amount in r11
shift_6:shl r9, r9, 1 //shift B-S[2i+1] by 1 bit
addi r8, r8, 1
blt r8, r11, shift_6 //keep on shifting right untill it is equal to 32-
shift amount
or r6, r9, r17 //B-S[2i+1] <<<< A
don't rotate for B:nor r1, r1, r4 //A'
nor r17, r17, r5 //(B-S[2i+1])'
and r1, r5, r6 //A(B-S[2i+1])'
and r17, r4, r7 //A'(B-S[2i+1])
or r6, r7, r2 // B-S[2i+1] <<<< A XOR A (Final value of B)
lw r30, r20, 0 // load Skey values for A
sub r1, r20, r18 //A-S[2i]
subi r30, r30, 1 //2i+1 for next cycle
shl r2, r5, 1B // shift left 'B' by 27 bits
shr r5, r5, 1B // shift right 'B' by 27 bits. Now, r5 has the rotate
amount
sub r7, r7, r7 //rotation aide counter
beq r5, r7, don't rotate for A
sub r6, r6, r6// assign the left shift register as r6
sub r9, r9, r9// assign the right shift register as r9
add r0, r18, r6
add r0, r18, r9
shift_7:shr r6, r6, 1 //shift A-S[2i] by 1 bit
addi r7, r7, 1
blt r7, r5, shift_7 //keep on shifting right untill it is equal to
shift amount
sub r8, r8, r8 //rotation aide counter
sub r10, r10, 10
addi r10, r10, 20
sub r10, r5, r11 //store 32-rotate amount in r11
shift_8:shl r9, r9, 1 //shift A-S[2i] by 1 bit
addi r8, r8, 1
blt r8, r11, shift_6 //keep on shifting right untill it is equal to 32-
shift amount
or r6, r9, r18 //A-S[2i] <<<< B
don't rotate for A: nor r2, r2, r4 //B'
nor r18, r18, r5 //(A-S[2i])'
and r2, r5, r6 //(A-S[2i])'B
and r18, r4, r7 //A-S[2i])B'
or r6, r7, r1 // A-S[2i]) <<<< B XOR B (Final value of A)
subi r3, r3, 1
blt r0, r3, decryption
lw r0, r26, 80//r26=S[0]
lw r0, r28, 81//r28=S[1]
sub r1, r26, r1//r1=A-S[0]
sub r2, r28, r2//r1=B-S[1]
sw r0, r1, 66 //store the upper half of encrypted output in 102
sw r0, r2, 67 //store the upper half of encrypted output in 103
hal//Decryption over for only decryption
```

8.2.2 Machine Code

```
X"00000011",X"1C010026",X"1C020027",X"03BDE811",X"07BD0001",X"00631811”
,X"0403000C",X"03FFF811",X"07FF0001",X"03DEF011",X"07DE0099",X"00000091
",X"1FD50000",X"00558811",X"0BDE0001",X"1425001B",X"18A5001B",X"00E7381
```

```
1",X"28A7000F",X"00C63011",X"01294811",X"00113010",X"00114810",X"18C600
01",X"04E70001",X"24E5FFFD",X"010847D1",X"014A57D1",X"054A0020",X"01455
811",X"15290001",X"05080001",X"250BFFFD",X"00C98FD3",X"002127D4",X"0231
2FD4",X"002537D2",X"02243FD2",X"00C717D3",X"1FD40000",X"00349011",X"0BD
E0001",X"1445001B",X"18A5001B",X"00E73811",X"28A7000F",X"00C63011",X"01
294811",X"00123010",X"00124810",X"18C60001",X"04E70001",X"24E5FFFD",X"1
0847D1",X"014A57D1",X"054A0020",X"01455811",X"15290001",X"05080001",X"2
50BFFFD",X"00C997D3",X"004227D4",X"02522FD4",X"004537D2",X"02443FD2",X"
00C70FD3",X"08630001",X"2403FFC8",X"1C1A0080",X"1C1C0081",X"003A0891",X
"005C1091",X"20010066",X"20020067",X"FC000000", others=>(others=>'0')
```

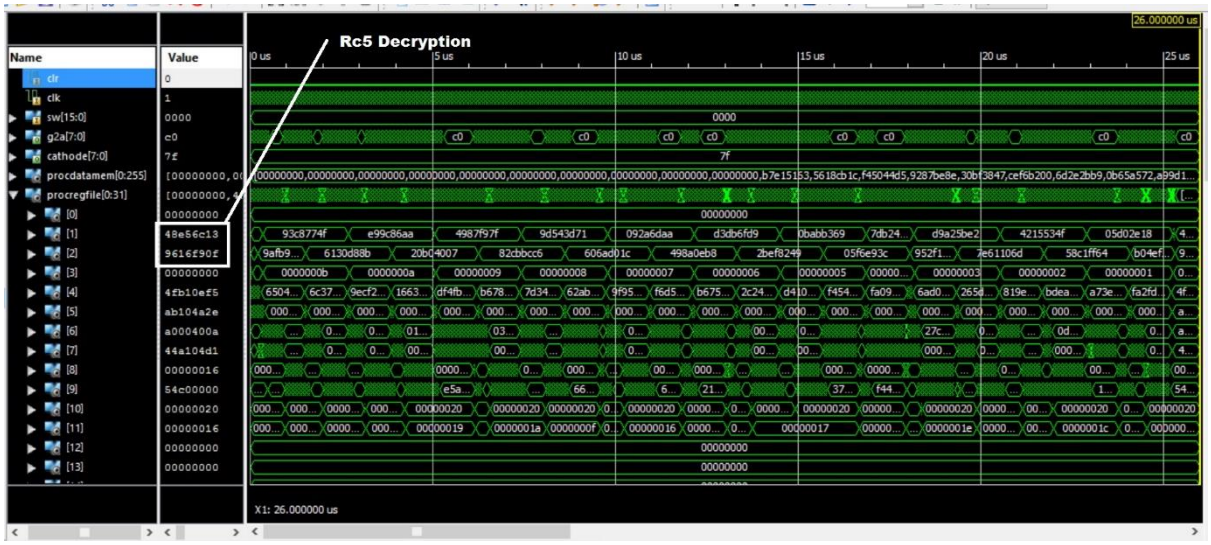


Fig 9: Decryption Simulation

8.3 Round Key Generation

8.3.1 Assembly Code

```
sub r27, r27, r27 //r27 = CLEAR
sub r0, r0, r0 //r0 = use r0 as always a zero register
start_round:bne r27, r0, start_key //at starting initialize S arrays;
when clear is zero, then initialise system or else go to encryption or
decryption
lw r0, r20, 0//User key[0]=r20; Don't want to mess with original plain
text that is why taking from 0-3 and storing to 4-7
lw r0, r21, 1//User key[1]=r21
lw r0, r22, 2//User key[2]=r22
lw r0, r23, 3//User key[3]=r23
sw r0, r20, 4//L_array[0]=r20
sw r0, r21, 5//L_array[0]=r21
sw r0, r22, 6//L_array[0]=r22
sw r0, r23, 7//L_array[0]=r23 //L_array initialisation is complete
lw r0, r1, A //S_array_initialised --> r1
lw r0, r2, B //S_array_initialised --> r2
lw r0, r3, C //S_array_initialised --> r3
lw r0, r4, D //S_array_initialised --> r4
lw r0, r5, E //S_array_initialised --> r5
lw r0, r6, F //S_array_initialised --> r6
lw r0, r7, 10 //S_array_initialised --> r7
lw r0, r8, 11 //S_array_initialised --> r8
lw r0, r9, 12 //S_array_initialised --> r9
lw r0, r10, 13 //S_array_initialised --> r10
lw r0, r11, 14 //S_array_initialised --> r11
```

```
lw r0, r12, 15 //S_array_initialised --> r12
lw r0, r13, 16 //S_array_initialised --> r13
lw r0, r14, 17 //S_array_initialised --> r14
lw r0, r15, 18 //S_array_initialised --> r15
lw r0, r16, 19 //S_array_initialised --> r16
lw r0, r17, 1A //S_array_initialised --> r17
lw r0, r18, 1B //S_array_initialised --> r18
lw r0, r19, 1C //S_array_initialised --> r19
lw r0, r20, 1D //S_array_initialised --> r20
lw r0, r21, 1E //S_array_initialised --> r21
lw r0, r22, 1F //S_array_initialised --> r22
lw r0, r23, 20 //S_array_initialised --> r23
lw r0, r24, 21 //S_array_initialised --> r24
lw r0, r25, 22 //S_array_initialised --> r25
lw r0, r26, 23 //S_array_initialised --> r26
sw r0, r1, 80 // r1-->S_key_array
sw r0, r2, 81 // r2-->S_key_array
sw r0, r3, 82 // r3-->S_key_array
sw r0, r4, 83 // r4-->S_key_array
sw r0, r5, 84 // r5-->S_key_array
sw r0, r6, 85 // r6-->S_key_array
sw r0, r7, 86 // r7-->S_key_array
sw r0, r8, 87 // r8-->S_key_array
sw r0, r9, 88 // r9-->S_key_array
sw r0, r10, 89 // r10-->S_key_array
sw r0, r11, 8A // r11-->S_key_array
sw r0, r12, 8B // r12-->S_key_array
sw r0, r13, 8C // r13-->S_key_array
sw r0, r14, 8D // r14-->S_key_array
sw r0, r15, 8E // r15-->S_key_array
sw r0, r16, 8F // r16-->S_key_array
sw r0, r17, 90 // r17-->S_key_array
sw r0, r18, 91 // r18-->S_key_array
sw r0, r19, 92 // r19-->S_key_array
sw r0, r20, 93 // r20-->S_key_array
sw r0, r21, 94 // r21-->S_key_array
sw r0, r22, 95 // r22-->S_key_array
sw r0, r23, 96 // r23-->S_key_array
sw r0, r24, 97 // r24-->S_key_array
sw r0, r25, 98 // r25-->S_key_array
sw r0, r26, 99 // r26-->S_key_array, S_array initialisation complete
sub r10, r10, r10 //A_pre=x"00000000"
sub r11, r11, r11 //B_pre=x"00000000"
sub r1, r1, r1 //flush register 1, counter i
sub r2, r2, r2 //flush register 2, counter j
sub r3, r3, r3 //flush register 3, counter k
sub r4, r4, r4 //flush register 4, for max of counter i
sub r5, r5, r5 //flush register 5, for max of counter j
sub r6, r6, r6 //flush register 6, for max of counter k
addi r4, r4, 1A //store max value of i
addi r5, r5, 4 //store max value of j
addi r6, r6, 4E //store max value of k
addi r27, r27, 1 //Change clear to 1 to denote that initialisation is
complete
start_key:add r10, r11, r12//r12= A+B
sub r7, r7, r7
addi r1, r7, 80 //counter to get S values from memory [128-153]
lw r7, r20, 0 //r20 has the S values
```

```
add r12, r20, r21//r21=S[i]+A+B
shl r21, r22, 3//r22=shifted S[i]+A+B to left
shr r21, r23, 1D//r22=shifted S[i]+A+B to right
or r23, r22, r10//r10=S[i]+A+B <<< 3
sw r7, r10, 0 //store S[i] in skey array in Dmem (locations: 128-153)
add r10, r11, r12//r12_new= A+B with new value of A
shl r12, r13, 1B
shr r13, r13, 1B//r13=data dependent rotate amount
sub r14, r14, r14
addi r14, r14, 20
sub r14, r13, r14 //store 32-rotate amount in r14
addi r2, r8, 4//r8=address of L_array
lw r8, r22, 0 //r22=L[j]
add r22, r12, r11 //r11=L[j]+A+B
sub r15, r15, r15 //rotation aide counter
beq r13, r15, don't rotate
sub r24, r24, r24//r24 being assigned for left shift
sub r25, r25, r25//r25 is being assigned for right shift
add r11, r0, r24//r24=r23
add r11, r0, r25//r25=r23
shift_left:shl r24, r24, 1 //shift L[j]+A+B by 1 bit
addi r15, r15, 1
blt r15, r13, shift_left
sub r16, r16, r16 //rotation aide counter
shift_right:shr r25, r25, 1 //shift L[j]+A+B by 1 bit to the right
addi r16, r16, 1
blt r16, r14, shift_right
or r24, r25, r11// B= L[j] = r11
don't_rotate:sw r8, r11, 0 //store L[j] in 4-7
addi r3, r3, 1//increment k
addi r2, r2, 1//increment j
addi r1, r1, 1//increment 1
blt r2, r5, don't initiliase j
sub r2, r2, r2 //initialise j
don't initialise j: blt r1, r4, don't initialise i
sub r1, r1, r1//initialise i
don't initialise i: beq r6, r3, go to hal
blt r3, r6, start_round
hal
```

8.3.2 Machine code

```
X"001BDED1",X"00000211",X"2F600049",X"1C140000",X"1C150001",X"1C160002"
,X"1C170003",X"20140004",X"20150005",X"20160006",X"20170007",X"1C01000A
",X"1C02000B",X"1C03000C",X"1C04000D",X"1C05000E",X"1C06000F",X"1C07001
0",X"1C080011",X"1C090012",X"1C0A0013",X"1C0B0014",X"1C0C0015",X"1C0D00
16",X"1C0E0017",X"1C0F0018",X"1C100019",X"1C11001A",X"1C12001B",X"1C130
01C",X"1C14001D",X"1C15001E",X"1C16001F",X"1C170020",X"1C180021",X"1C19
0022",X"1C1A0023",X"20010080",X"20020081",X"20030082",X"20040083",X"200
50084",X"20060085",X"20070086",X"20080087",X"20090088",X"200A0089",X"20
0B008A",X"200C008B",X"200D008C",X"200E008D",X"200F008E",X"2010008F",X"2
0110090",X"20120091",X"20130092",X"20140093",X"20150094",X"20160095",X"
20170096",X"20180097",X"20190098",X"201A0099",X"014A5091",
X"016B5891",X"00210891",X"00421091",X"00631891",X"00842091",X"00A52891"
,X"00C63091",X"0484001A",X"04A50004",X"04C6004E",X"077B0001",X"00E73811
",X"014B6010",X"04270080",X"00000091",X"1CF40000",X"00000011",X"0194A81
0",X"00000011",X"16B60003",X"1AB7001D",X"00000011",X"02F65013",X"000000
11",X"20EA0000",X"014B6010",X"00000011",X"158D001B",X"19AD001B",X"01CE7
011",X"05CE0020",X"01CD7011",X"04480004",X"00000011",X"1D160000",X"0000
```

```
0011",X"02CC5810",X"01EF7811",X"29AF000C",X"0318C011",X"0339C811",X"016
0C010",X"0160C810",X"17180001",X"05EF0001",X"25EDFFFD",X"021087D1",X"1B
390001",X"06100001",X"260EFFFD",X"03195FD3",X"210B0000",X"04630001",X"0
4420001",X"04210001",X"24450001",X"00421011",X"24240001",X"00210811",X"
28C30001",X"2466FF84", others=>(others=>'0')
```

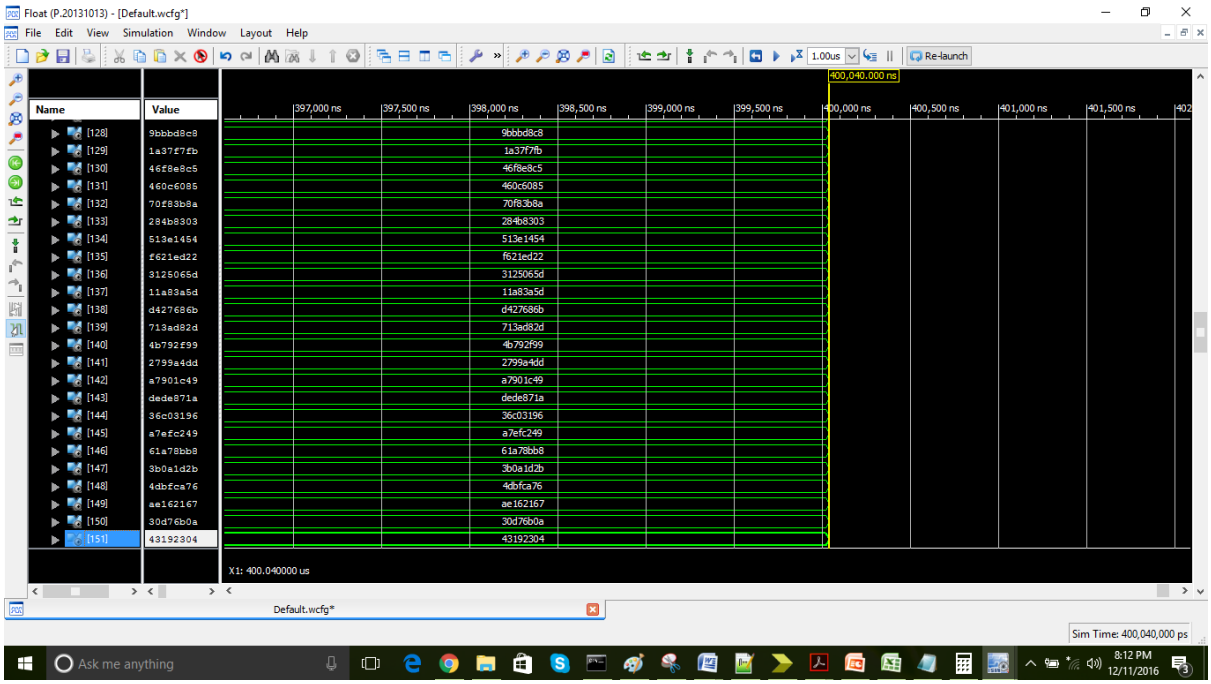


Fig 10: Round key simulation part (1)

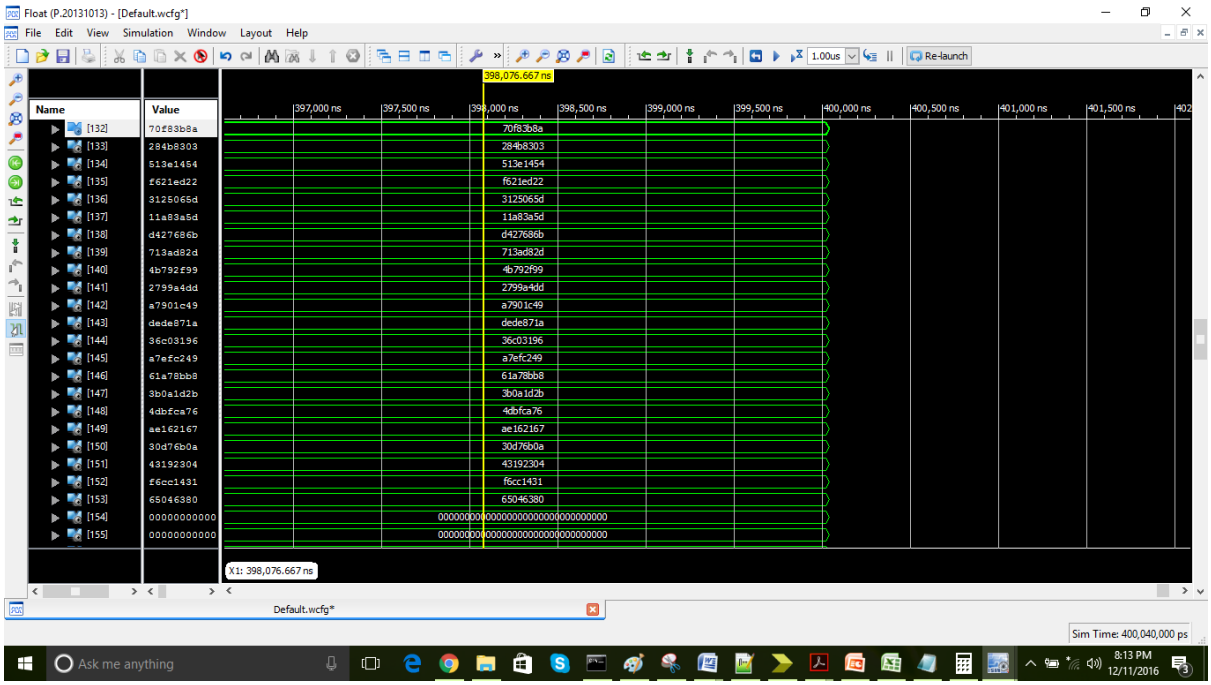


Fig 11: Round key simulation part (2)

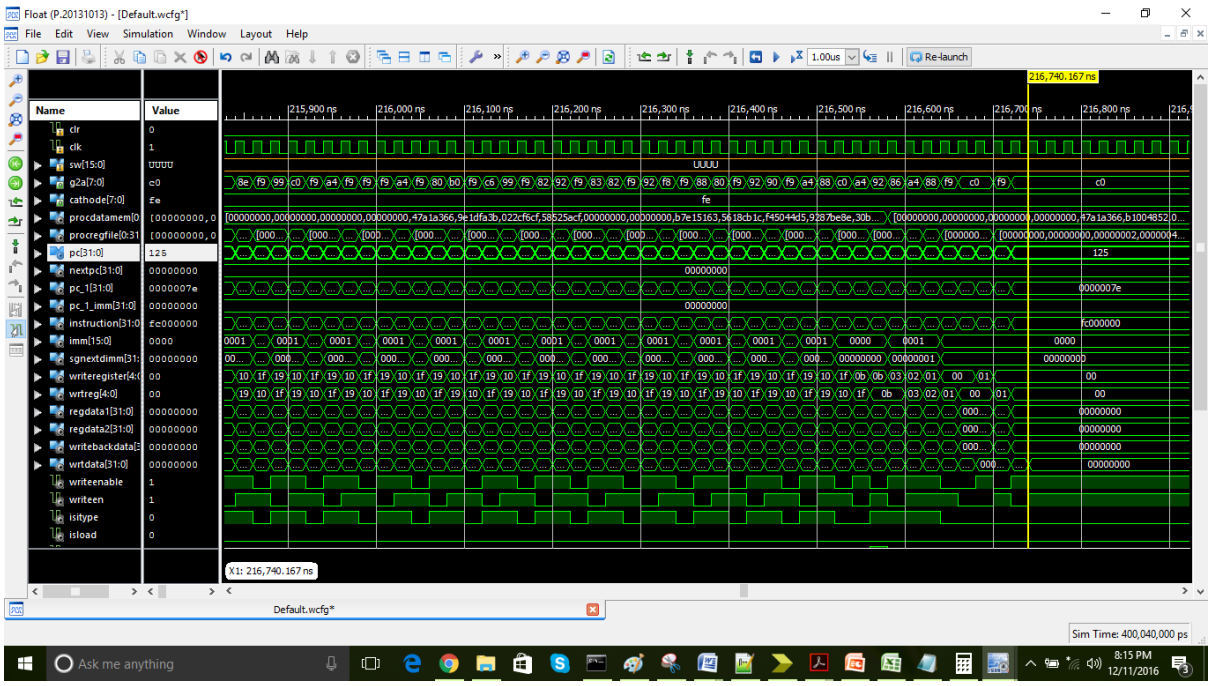


Fig 12: Round key simulation part (3)

8.4 Consolidated RC5

8.4.1 Assembly Code

```
sub r27, r27, r27 //r27 = CLEAR
sub r0, r0, r0 //r0 = use r0 as always a zero register
start_round:bne r27, r0, start_key //at starting initialize S arrays;
when clear is zero, then initialise system or else go to encryption or
decryption
lw r0, r20, 0//User key[0]=r20; Don't want to mess with original plain
text that is why taking from 0-3 and storing to 4-7
lw r0, r21, 1//User key[1]=r21
lw r0, r22, 2//User key[2]=r22
lw r0, r23, 3//User key[3]=r23
sw r0, r20, 4//L_array[0]=r20
sw r0, r21, 5//L_array[0]=r21
sw r0, r22, 6//L_array[0]=r22
sw r0, r23, 7//L_array[0]=r23 //L_array initialisation is complete
lw r0, r1, A //S_array_initialised --> r1
lw r0, r2, B //S_array_initialised --> r2
lw r0, r3, C //S_array_initialised --> r3
lw r0, r4, D //S_array_initialised --> r4
lw r0, r5, E //S_array_initialised --> r5
lw r0, r6, F //S_array_initialised --> r6
lw r0, r7, 10 //S_array_initialised --> r7
lw r0, r8, 11 //S_array_initialised --> r8
lw r0, r9, 12 //S_array_initialised --> r9
lw r0, r10, 13 //S_array_initialised --> r10
lw r0, r11, 14 //S_array_initialised --> r11
lw r0, r12, 15 //S_array_initialised --> r12
lw r0, r13, 16 //S_array_initialised --> r13
lw r0, r14, 17 //S_array_initialised --> r14
lw r0, r15, 18 //S_array_initialised --> r15
lw r0, r16, 19 //S_array_initialised --> r16
lw r0, r17, 1A //S_array_initialised --> r17
lw r0, r18, 1B //S_array_initialised --> r18
```

```
lw r0, r19, 1C //S_array_initialised --> r19
lw r0, r20, 1D //S_array_initialised --> r20
lw r0, r21, 1E //S_array_initialised --> r21
lw r0, r22, 1F //S_array_initialised --> r22
lw r0, r23, 20 //S_array_initialised --> r23
lw r0, r24, 21 //S_array_initialised --> r24
lw r0, r25, 22 //S_array_initialised --> r25
lw r0, r26, 23 //S_array_initialised --> r26
sw r0, r1, 80 // r1--> S_key_array
sw r0, r2, 81 // r2--> S_key_array
sw r0, r3, 82 // r3--> S_key_array
sw r0, r4, 83 // r4--> S_key_array
sw r0, r5, 84 // r5--> S_key_array
sw r0, r6, 85 // r6--> S_key_array
sw r0, r7, 86 // r7--> S_key_array
sw r0, r8, 87 // r8--> S_key_array
sw r0, r9, 88 // r9--> S_key_array
sw r0, r10, 89 // r10--> S_key_array
sw r0, r11, 8A // r11--> S_key_array
sw r0, r12, 8B // r12--> S_key_array
sw r0, r13, 8C // r13--> S_key_array
sw r0, r14, 8D // r14--> S_key_array
sw r0, r15, 8E // r15--> S_key_array
sw r0, r16, 8F // r16--> S_key_array
sw r0, r17, 90 // r17--> S_key_array
sw r0, r18, 91 // r18--> S_key_array
sw r0, r19, 92 // r19--> S_key_array
sw r0, r20, 93 // r20--> S_key_array
sw r0, r21, 94 // r21--> S_key_array
sw r0, r22, 95 // r22--> S_key_array
sw r0, r23, 96 // r23--> S_key_array
sw r0, r24, 97 // r24--> S_key_array
sw r0, r25, 98 // r25--> S_key_array
sw r0, r26, 99 // r26--> S_key_array, S_array initialisation complete
sub r10, r10, r10 //A_pre=x"00000000"
sub r11, r11, r11 //B_pre=x"00000000"
sub r1, r1, r1 //flush register 1, counter i
sub r2, r2, r2 //flush register 2, counter j
sub r3, r3, r3 //flush register 3, counter k
sub r4, r4, r4 //flush register 4, for max of counter i
sub r5, r5, r5 //flush register 5, for max of counter j
sub r6, r6, r6 //flush register 6, for max of counter k
addi r4, r4, 1A //store max value of i
addi r5, r5, 4 //store max value of j
addi r6, r6, 4E //store max value of k
addi r27, r27, 1 //Change clear to 1 to denote that initialisation is
complete
start_key:add r10, r11, r12//r12= A+B
sub r7, r7, r7
addi r1, r7, 80 //counter to get S values from memory [128-153]
lw r7, r20, 0 //r20 has the S values
add r12, r20, r21//r21=S[i]+A+B
shl r21, r22, 3//r22=shifted S[i]+A+B to left
shr r21, r23, 1D//r22=shifted S[i]+A+B to right
or r23, r22, r10//r10=S[i]+A+B <<< 3
sw r7, r10, 0 //store S[i] in skey array in Dmem (locations: 128-153)
add r10, r11, r12//r12_new= A+B with new value of A
shl r12, r13, 1B
```

```
shr r13, r13, 1B//r13=data dependent rotate amount
sub r14, r14, r14
addi r14, r14, 20
sub r14, r13, r14 //store 32-rotate amount in r14
addi r2, r8, 4//r8=address of L_array
lw r8, r22, 0 //r22=L[j]
add r22, r12, r11 //r11=L[j]+A+B
sub r15, r15, r15 //rotation aide counter
beq r13, r15, don't rotate
sub r24, r24, r24//r24 being assigned for left shift
sub r25, r25, r25//r25 is being assigned for right shift
add r11, r0, r24//r24=r23
add r11, r0, r25//r25=r23
shift_left:shl r24, r24, 1 //shift L[j]+A+B by 1 bit
addi r15, r15, 1
blt r15, r13, shift_left
sub r16, r16, r16 //rotation aide counter
shift_right:shr r25, r25, 1 //shift L[j]+A+B by 1 bit to the right
addi r16, r16, 1
blt r16, r14, shift_right
or r24, r25, r11// B= L[j] = r11
don't_rotate:sw r8, r11, 0 //store L[j] in 4-7
addi r3, r3, 1//increment k
addi r2, r2, 1//increment j
addi r1, r1, 1//increment 1
blt r2, r5, don't initiliase j
sub r2, r2, r2 //initialise j
don't initialise j: blt r1, r4, don't initialise i
sub r1, r1, r1//initialise i
don't initialise i: beq r6, r3, encrypt_start
blt r3, r6, start_round

encrypt_start:sub r0, r0, r0 //flush register 0
lw r0, r1, 24 //Din value is stored in 36 = A
lw r0, r2, 80 //first value of skey, S[0] is stored in r2
add r1, r2, r1 //Now r1 has value of A_pre=A+ S[0]
lw r0, r2, 25 //Din value is stored in 37 = B
lw r0, r3, 81 //second value of S[1]
add r2, r3, r2 // Now r2 has the value of B_pre=B+S[1]
sub r3, r3, r3 //flush register 3, to be used as a max value of i
addi r0, r3, D // Load counter register r3 with 13 as a blt counter
will be run
sub r31, r31, r31
addi r31, r31, 1 //initialise i with 1
sub r30, r30, r30 //flush r30
addi r30, r30, 81 // initialise corresoding to the memory position of
s[1]
encryption:nor r1, r1, r4 //A'
nor r2, r2, r5 //B'
and r1, r5, r6 //AB'
and r2, r4, r7 //A'B
or r6, r7, r4 // A xor B in r4
shl r2, r5, 1B // shift left 'B' by 27 bits
shr r5, r5, 1B // shift right 'B' by 27 bits. Now, r5 has the rotate
amount
sub r7, r7, r7 //rotation aide counter
beq r5, r7, don't rotate for A
sub r6, r6, r6// assign the left shift register as r6
```

```
sub r9, r9, r9// assign the right shift register as r9
add r0, r4, r6
add r0, r4, r9
shift_1:shl r6, r6, 1 //shift A XOR B by 1 bit
addi r7, r7, 1
blt r7, r5, shift_1 //keep on shifting left untill it is equal to shift
amount
sub r8, r8, r8 //rotation aide counter
sub r10, r10, r10
addi r10, r10, 20
sub r10, r5, r11 //store 32-rotate amount in r11
shift_2:shr r9, r9, 1 //shift A XOR B by 1 bit
addi r8, r8, 1
blt r8, r11, shift_2 //keep on shifting right untill it is equal to 32-
shift amount
or r6, r9, r4 //A XOR B <<<< B
don't rotate for A: addi r30, r30, 1
lw r30, r20, 0 // load Skey values for A
add r4, r20, r1 //Final value of A
nor r1, r1, r4 //A'
nor r2, r2, r5 //B'
and r1, r5, r6 //AB'
and r2, r4, r7 //A'B
or r6, r7, r4 // B xor A in r4
shl r1, r5, 1B // shift left 'A' by 27 bits
shr r5, r5, 1B // shift right 'A' by 27 bits. Now, r5 has the rotate
amount
sub r7, r7, r7 //rotation aide counter
beq r5, r7, don't rotate for B
sub r6, r6, r6// assign the left shift register as r6
sub r9, r9, r9// assign the right shift register as r9
add r0, r4, r6
add r0, r4, r9
shift_3:shl r6, r6, 1 //shift B XOR A by 1 bit
addi r7, r7, 1
blt r7, r5, shift_3 //keep on shifting left untill it is equal to shift
amount
sub r8, r8, r8 //rotation aide counter
sub r10, r10, r10
addi r10, r10, 20
sub r10, r5, r11 //store 32-rotate amount in r11
shift_4:shr r9, r9, 1 //shift A XOR B by 1 bit
addi r8, r8, 1
blt r8, r11, shift_4 //keep on shifting right untill it is equal to 32-
shift amount
or r6, r11, r4 //B XOR A <<<< A
don't rotate for B:addi r30, r30, 1
lw r30, r21, 0 // load Skey values for A
add r4, r21, r2 //Final value of B
addi r31, r31, 1
blt r31, r3, encryption
sw r0, r1, 64 //store the upper half of encrypted output in 100
sw r0, r2, 65 //store the upper half of encrypted output in 101
sub r22, r22, r22
sub r23, r23, r23
add r1, r0, r22
add r2, r0, r23
hal //Encryption over for only encryption
```

```
decrypt_start: sub r0, r0, r0 //flush register 0
lw r0, r1, 26 //Din value is stored in 38 = A
lw r0, r2, 27 //Din value is stored in 39 = B
sub r29, r29, r29
addi r29, r29, 1
sub r3, r3, r3 //flush register 3, to be used as a counter
addi r0, r3, C // Load counter register r3 with 12 as a blt counter
will be run
sub r31, r31, r31
addi r31, r31, 1 //initialise i with 1
sub r30, r30, r30 //flush r30
addi r30, r30, 99 // initialise corresoding to the memory position of
S[25]
decryption: lw r30, r21, 0 // load Skey values for B
sub r2, r21, r17 //B-S[2i+1]
subi r30, r30, 1 //2i
shl r1, r5, 1B // shift left 'A' by 27 bits
shr r5, r5, 1B // shift right 'A' by 27 bits. Now, r5 has the rotate
amount
sub r7, r7, r7 //rotation aide counter
beq r5, r7, don't rotate for B
sub r6, r6, r6// assign the left shift register as r6
sub r9, r9, r9// assign the right shift register as r9
add r0, r17, r6
add r0, r17, r9
shift_5:shr r6, r6, 1 //shift B-S[2i+1] by 1 bit
addi r7, r7, 1
blt r7, r5, shift_5 //keep on shifting right untill it is equal to
shift amount
sub r8, r8, r8 //rotation aide counter
sub r10, r10, 10
addi r10, r10, 20
sub r10, r5, r11 //store 32-rotate amount in r11
shift_6:shl r9, r9, 1 //shift B-S[2i+1] by 1 bit
addi r8, r8, 1
blt r8, r11, shift_6 //keep on shifting right untill it is equal to 32-
shift amount
or r6, r9, r17 //B-S[2i+1] <<<< A
don't rotate for B:nor r1, r1, r4 //A'
nor r17, r17, r5 //(B-S[2i+1])'
and r1, r5, r6 //A(B-S[2i+1])'
and r17, r4, r7 //A'(B-S[2i+1])
or r6, r7, r2 // B-S[2i+1] <<<< A XOR A (Final value of B)
lw r30, r20, 0 // load Skey values for A
sub r1, r20, r18 //A-S[2i]
subi r30, r30, 1 //2i+1 for next cycle
shl r2, r5, 1B // shift left 'B' by 27 bits
shr r5, r5, 1B // shift right 'B' by 27 bits. Now, r5 has the rotate
amount
sub r7, r7, r7 //rotation aide counter
beq r5, r7, don't rotate for A
sub r6, r6, r6// assign the left shift register as r6
sub r9, r9, r9// assign the right shift register as r9
add r0, r18, r6
add r0, r18, r9
shift_7:shr r6, r6, 1 //shift A-S[2i] by 1 bit
addi r7, r7, 1
```

```
blt r7, r5, shift_7 //keep on shifting right untill it is equal to
shift amount
sub r8, r8, r8 //rotation aide counter
sub r10, r10, 10
addi r10, r10, 20
sub r10, r5, r11 //store 32-rotate amount in r11
shift_8:shl r9, r9, 1 //shift A-S[2i] by 1 bit
addi r8, r8, 1
blt r8, r11, shift_6 //keep on shifting right untill it is equal to 32-
shift amount
or r6, r9, r18 //A-S[2i] <<<< B
don't rotate for A: nor r2, r2, r4 //B'
nor r18, r18, r5 //(A-S[2i])'
and r2, r5, r6 //(A-S[2i])'B
and r18, r4, r7 //A-S[2i])B'
or r6, r7, r1 // A-S[2i]) <<<< B XOR B (Final value of A)
subi r3, r3, 1
blt r0, r3, decryption
lw r0, r26, 80//r26=S[0]
lw r0, r28, 81//r28=S[1]
sub r1, r26, r1//r1=A-S[0]
sub r2, r28, r2//r1=B-S[1]
sw r0, r1, 66 //store the upper half of encrypted output in 102
sw r0, r2, 67 //store the upper half of encrypted output in 103
sub,0,0,0//in the subsequent instructions load the skey values on
register file for viewing
lw,0,3,80
lw,0,4,81
lw,0,5,82
lw,0,6,83
lw,0,7,84
lw,0,8,85
lw,0,9,86
lw,0,10,87
lw,0,11,88
lw,0,12,89
lw,0,13,8A
lw,0,14,8B
lw,0,15,8C
lw,0,16,8D
lw,0,17,8E
lw,0,18,8F
lw,0,19,90
lw,0,20,91
lw,0,21,92
lw,0,24,93
lw,0,25,94
lw,0,26,95
lw,0,27,96
lw,0,28,97
lw,0,29,98
lw,0,30,99
hal//Decryption over for only decryption
```

8.4.2 Machine code

```
(X"001BDED1",X"00000211",X"2F600049",X"1C140000",X"1C150001",
X"1C160002",X"1C170003",X"20140004",X"20150005",X"20160006",X"20170007",X"1C01000A",X"1
C02000B",X"1C03000C",X"1C04000D",X"1C05000E",X"1C06000F",X"1C070010",X"1C080011",X"1C0
```

90012",X"1C0A0013",X"1C0B0014",X"1C0C0015",X"1C0D0016",X"1C0E0017",X"1C0F0018",X"1C100019",X"1C11001A",X"1C12001B",X"1C13001C",X"1C14001D",X"1C15001E",X"1C16001F",X"1C170020",X"1C180021",X"1C190022",X"1C1A0023",X"20010080",X"20020081",X"20030082",X"20040083",X"20050084",X"20060085",X"20070086",X"20080087",X"20090088",X"200A0089",X"200B008A",X"200C008B",X"200D008C",X"200E008D",X"200F008E",X"2010008F",X"20110090",X"20120091",X"20130092",X"20140093",X"20150094",X"20160095",X"20170096",X"20180097",X"20190098",X"201A0099",X"014A5091",X"016B5891",X"00210891",X"00421091",X"00631891",X"00842091",X"00A52891",X"00C63091",X"0484001A",X"04A50004",X"04C6004E",X"077B0001",X"00E73811",X"014B6010",X"04270080",X"00000091",X"1CF40000",X"00000011",X"0194A810",X"00000011",X"16B60003",X"1AB7001D",X"00000011",X"02F65013",X"00000011",X"20EA0000",X"014B6010",X"00000011",X"158D001B",X"19AD001B",X"01CE7011",X"05CE0020",X"01CD7011",X"04480004",X"00000011",X"1D160000",X"00000011",X"02CC5810",X"01EF7811",X"29AF000C",X"0318C011",X"0339C811",X"0160C010",X"0160C810",X"17180001",X"05EF0001",X"25EDFFFD",X"021087D1",X"1B390001",X"06100001",X"260EFFFD",X"03195FD3",X"210B0000",X"04630001",X"04420001",X"04210001",X"24450001",X"00421011",X"24240001",X"00210811",X"28C30001",X"2466FF84",X"00000011",X"1C010024",X"1C020080",X"1C030025",X"1C040081",X"00220890",X"00641090",X"00631891",X"0403000D",X"03FFF811",X"07FF0001",X"03DEF011",X"07DE0081",X"00212094",X"00422894",X"00253092",X"00443892",X"00C72093",X"1445001B",X"18A5001B",X"00E73811",X"28A7000F",X"00C63011",X"01294811",X"00043010",X"00044810",X"14C60001",X"04E70001",X"24E5FFFD",X"010847D1",X"014A57D1",X"054A0020",X"01455811",X"19290001",X"05080001",X"250BFFFD",X"00C927D3",X"07DE0001",X"1FD40000",X"00940810",X"00212014",X"00422814",X"00253012",X"00443812",X"00C72013",X"1425001B",X"18A5001B",X"00E73811",X"28A7000F",X"00C63011",X"01294811",X"00043010",X"00044810",X"14C60001",X"04E70001",X"24E5FFFD",X"010847D1",X"014A57D1",X"054A0020",X"01455811",X"19290001",X"05080001",X"250BFFFD",X"00C927D3",X"07DE0001",X"1FD50000",X"00951010",X"07FF0001",X"27E3FFC8",X"20010064",X"20020065",X"0016B591",X"02F7BA11",X"0020B210",X"0040BA10",X"00000011",X"1C010026",X"1C020027",X"03BDE811",X"07BD0001",X"00631811",X"0403000C",X"03FFF811",X"07FF0001",X"03DEF011",X"07DE0099",X"00000091",X"1FD50000",X"00558811",X"0BDE0001",X"1425001B",X"18A5001B",X"00E73811",X"28A7000F",X"00C63011",X"01294811",X"00113010",X"00114810",X"18C60001",X"04E70001",X"24E5FFFD",X"010847D1",X"014A57D1",X"054A0020",X"01455811",X"15290001",X"05080001",X"250BFFFD",X"00C98FD3",X"002127D4",X"02312FD4",X"002537D2",X"02243FD2",X"00C717D3",X"1FD40000",X"00349011",X"0BDE0001",X"1445001B",X"18A5001B",X"00E73811",X"28A7000F",X"00C63011",X"01294811",X"00123010",X"00124810",X"18C60001",X"04E70001",X"24E5FFFD",X"010847D1",X"014A57D1",X"054A0020",X"01455811",X"15290001",X"05080001",X"250BFFFD",X"00C997D3",X"004227D4",X"02522FD4",X"004537D2",X"02443FD2",X"00C70FD3",X"08630001",X"2403FFC8",X"1C1A0080",X"1C1C0081",X"003A0891",X"005C1091",X"20010066",X"20020067",X"00000011",X"1C030080",X"1C040081",X"1C050082",X"1C060083",X"1C070084",X"1C080085",X"1C090086",X"1C0A0087",X"1C0B0088",X"1C0C0089",X"1C0D008A",X"1C0E008B",X"1C0F008C",X"1C10008D",X"1C11008E",X"1C12008F",X"1C130090",X"1C140091",X"1C150092",X"1C180093",X"1C190094",X"1C1A0095",X"1C1B0096",X"1C1C0097",X"1C1D0098",X"1C1E0099",X"FC000000",others=>(others =>'0'));
Screenshot is attached in the next section with Latency calculation

9. Latency Calculation for RC5

9.1 For Round key generation

Process	Sub-process	Cycles needed	Requirement
Round key generation	Initial loading of initialized values and copying them to	76	60 are optional because we can use the initialized memory

	working registers and memory location and rest of the initialisation		locations to perform operations. This program keeps the initialized values undisturbed 16 are mandatory initialization procedure
	Other operations till rotation	19	Required
	Rotation	12*n	Required: where n is the rotate amount
	Preparation for next cycle	10	Required

Absolute minimum number of cycle for only Round key generation with code with assembly code reduction: 16+(19+0+10)*78=2294

Minimum number of clock cycles for Round key generation with no code reduction (i.e. without messing with initialized values)=76+(19+0+10)=2338

9.2 For Encryption

Process	Sub-process	Cycles needed	Requirement
Encryption	Entire Encryption apart from rotation	45	If we decide to not store the final results in data memory then 2 instructions are not mandatory
	Rotation for A and rotation for B	(15+15)*n	Required where n is the rotation amount

Absolute minimum number of cycle for only Round key generation with code with assembly code reduction: 13+32*12= 397

Minimum number of clock cycles for Round key generation with no code reduction (i.e. without messing with initialized values)=13+32*12+2=399

9.3 For Decryption

Process	Sub-process	Cycles needed	Requirement
Decryption	Entire Decryption apart from rotation	43	If we decide to not store the final results in data memory then 2 instructions are not mandatory
	Rotation for A and rotation for B	(15+15)*n	Required where n is the rotation amount

Absolute minimum number of cycle for only Round key generation with code with assembly code reduction: 13+30*12= 373

Minimum number of clock cycles for Round key generation with no code reduction (i.e. without messing with initialized values)=13+32*12+2=375

9.4 Actual Scenario

RC5 was run with User key=(00000000000000000000000000000000) Hex and plain text for encryption as (0000000000000000) Hex and cipher text for decryption as (0000000000000000) Hex. The first one was verified from Ron Rivest's original paper and decryption was verified from golden model. And we got the final (verified output) for entire encryption, decryption and round key generation after 16220 cycles (Please see screenshot attached).

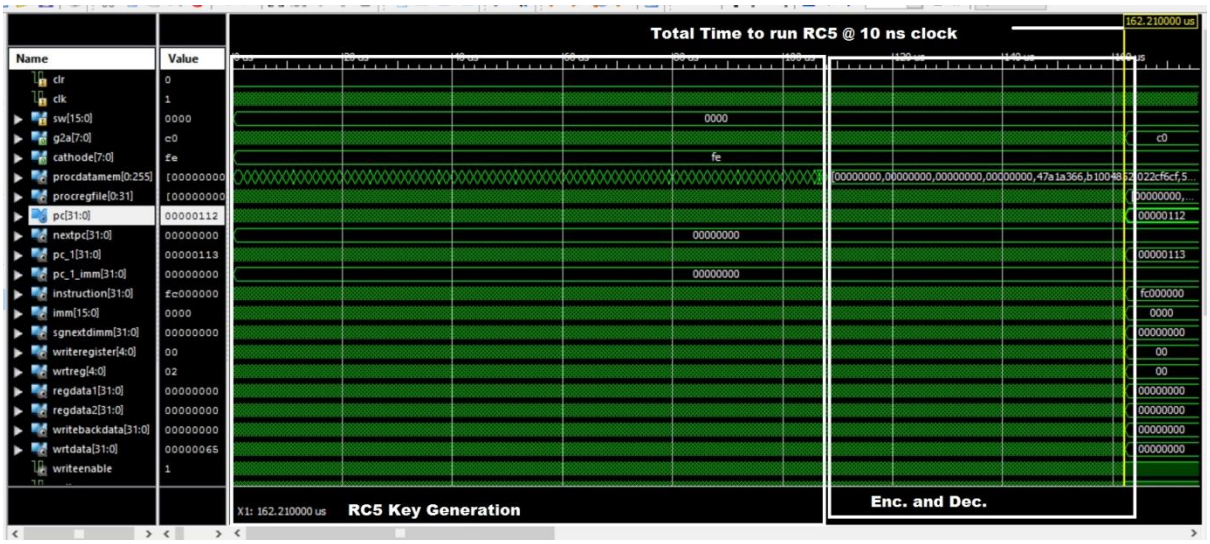


Fig 11: Entire RC5 simulation showing the total clock cycles

10. Timing Simulation

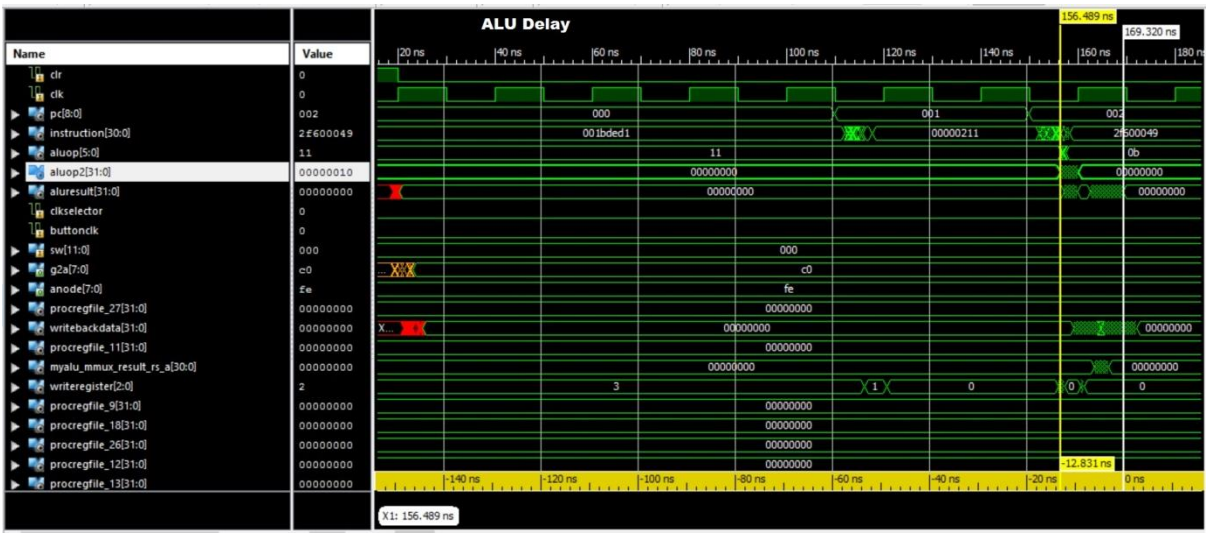


Fig 12: timing simulation of ALU running RC5

Fig 13: Timing simulation of Decoder running RC5

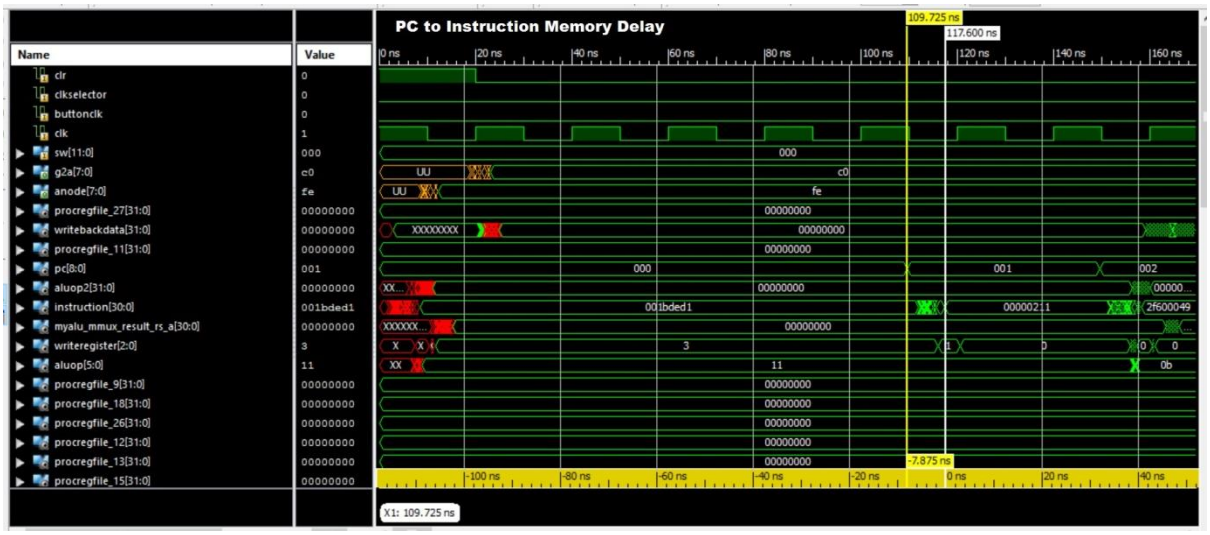


Fig 14: Timing Simulation of Instruction Memory running RC5

Fig 15: Timing Simulation of Write Back Data in RC5

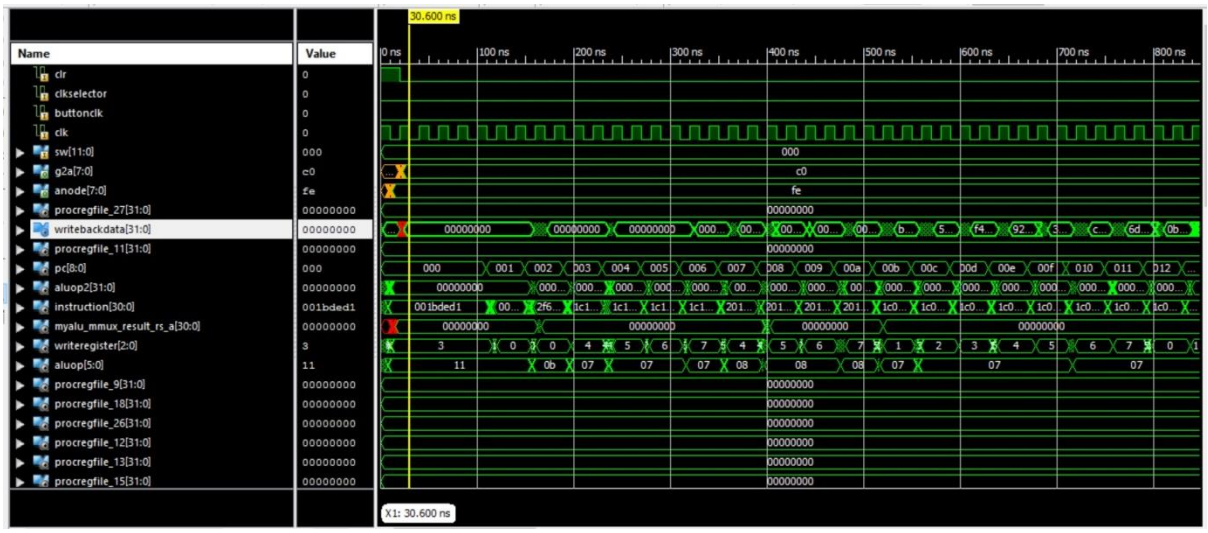


Fig 16: Timing Simulation of RC5 (complete)

11. Performance and Area Analysis

10.1 Resources

```
Macro Statistics
# RAMs                                          : 5
 16x8-bit single-port Read Only RAM             : 1
 512x32-bit single-port RAM                     : 1
 512x32-bit single-port Read Only RAM           : 1
 64x1-bit single-port Read Only RAM             : 1
 8x8-bit single-port Read Only RAM              : 1
# Adders/Subtractors                            : 5
 16-bit adder                                   : 1
 3-bit adder                                    : 1
 32-bit adder                                   : 2
 32-bit addsub                                  : 1
# Registers                                     : 36
 1-bit register                                 : 1
 16-bit register                                : 1
 3-bit register                                 : 1
 32-bit register                                : 33
# Comparators                                   : 3
 16-bit comparator greater                      : 1
 32-bit comparator equal                        : 1
 32-bit comparator greater                      : 1
# Multiplexers                                  : 64
 1-bit 2-to-1 multiplexer                       : 27
 1-bit 8-to-1 multiplexer                       : 4
 16-bit 2-to-1 multiplexer                      : 1
 32-bit 2-to-1 multiplexer                      : 25
 32-bit 32-to-1 multiplexer                     : 5
 5-bit 2-to-1 multiplexer                       : 1
 6-bit 2-to-1 multiplexer                       : 1
```

8.2 Post- Synthesis Phase:

```
Slice Logic Utilization:
 Number of Slice Registers:            1053  out of  126800     0%
 Number of Slice LUTs:                 2067  out of  63400      3%
    Number used as Logic:              1811  out of  63400      2%
    Number used as Memory:              256  out of  19000      1%
       Number used as RAM:              256

Slice Logic Distribution:
 Number of LUT Flip Flop pairs used:   2719
   Number with an unused Flip Flop:    1666  out of   2719     61%
   Number with an unused LUT:           652  out of   2719     23%
   Number of fully used LUT-FF pairs:   401  out of   2719     14%
   Number of unique control sets:        36

IO Utilization:
 Number of IOs:                         32
 Number of bonded IOBs:                 32  out of    210     15%

Specific Feature Utilization:
 Number of BUFG/BUFGCTRLs:               2  out of     32      6%
```

```
 Minimum period: 12.122ns (Maximum Frequency: 82.494MHz)
 Minimum input arrival time before clock: 1.173ns
 Maximum output required time after clock: 5.572ns
 Maximum combinational path delay: 5.446ns
```

10.3 Post Route Phase

Slice Logic Utilization:				
Number of Slice Registers:	1,053	out of	126,800	1%
Number used as Flip Flops:	1,053			
Number used as Latches:	0			
Number used as Latch-thrus:	0			
Number used as AND/OR logics:	0			
Number of Slice LUTs:	2,037	out of	63,400	3%
Number used as logic:	1,779	out of	63,400	2%
Number using O6 output only:	1,721			
Number using O5 output only:	7			
Number using O5 and O6:	51			
Number used as ROM:	0			
Number used as Memory:	256	out of	19,000	1%
Number used as Dual Port RAM:	0			
Number used as Single Port RAM:	256			
Number using O6 output only:	256			
Number using O5 output only:	0			
Number using O5 and O6:	0			
Number used as Shift Register:	0			
Number used exclusively as route-thrus:	2			
Number with same-slice register load:	0			
Number with same-slice carry load:	2			
Number with other load:	0			
Slice Logic Distribution:				
Number of occupied Slices:	1,379	out of	15,850	8%
Number of LUT Flip Flop pairs used:	2,915			
Number with an unused Flip Flop:	1,863	out of	2,915	63%
Number with an unused LUT:	878	out of	2,915	30%
Number of fully used LUT-FF pairs:	174	out of	2,915	5%
Number of slice register sites lost to control set restrictions:	0	out of	126,800	0%

```
IO Utilization:
  Number of bonded IOBs:                          32 out of     210   15%
    Number of LOCed IOBs:                         32 out of      32  100%

Specific Feature Utilization:
  Number of RAMB36E1/FIFO36E1s:                    0 out of     135    0%
  Number of RAMB18E1/FIFO18E1s:                    0 out of     270    0%
  Number of BUFG/BUFGCTRLs:                        2 out of      32    6%
    Number used as BUFGs:                          2
    Number used as BUFGCTRLs:                      0
  Number of IDELAYE2/IDELAYE2_FINEDELAYs:          0 out of     300    0%
  Number of ILOGICE2/ILOGICE3/ISERDESE2s:          0 out of     300    0%
  Number of ODELAYE2/ODELAYE2_FINEDELAYs:          0
  Number of OLOGICE2/OLOGICE3/OSERDESE2s:          0 out of     300    0%
  Number of PHASER_IN/PHASER_IN_PHYs:              0 out of      24    0%
  Number of PHASER_OUT/PHASER_OUT_PHYs:            0 out of      24    0%
  Number of BSCANs:                                0 out of       4    0%
  Number of BUFHCEs:                               0 out of      96    0%
  Number of BUFRs:                                 0 out of      24    0%
  Number of CAPTUREs:                              0 out of       1    0%
  Number of DNA_PORTs:                             0 out of       1    0%
  Number of DSP48E1s:                              0 out of     240    0%
  Number of EFUSE_USRs:                            0 out of       1    0%
  Number of FRAME_ECCs:                            0 out of       1    0%
  Number of IBUFDS_GTE2s:                          0 out of       4    0%
  Number of ICAPs:                                 0 out of       2    0%
  Number of IDELAYCTRLs:                           0 out of       6    0%
  Number of IN_FIFOs:                              0 out of      24    0%
  Number of MMCME2_ADVs:                           0 out of       6    0%
  Number of OUT_FIFOs:                             0 out of      24    0%
  Number of PCIE_2_1s:                             0 out of       1    0%
  Number of PHASER_REFs:                           0 out of       6    0%
  Number of PHY_CONTROLs:                          0 out of       6    0%
```

Constraint	Check	Worst Case Slack	Best Case Achievable	Timing Errors	Timing Score
Autotimespec constraint for clock net chosenClk_BUFG	SETUP	N/A	31.309ns	N/A	0
	HOLD	0.362ns		0	0
Autotimespec constraint for clock net clk_BUFGP	SETUP	N/A	9.092ns	N/A	0
	HOLD	0.273ns		0	0

- The gates for area utilisation is denoted in the above screenshots
- Placement and Route phase is closer to the hardware than synthesis phase.
- Slice registers are staying constant at 1053 which means that the analysis done during synthesis phase was accurate.
- Slice LUTs are reducing after post placement and route because the LUTs are being used more effectively and thus the number is reducing from 2067 to 2037.
- The minimum period with the original clock is reducing from 12.122 ns to 9.092 ns. But to counter the worst-case scenario, we do not run the processor at 100 Mhz and thus we use a clock divider running the processor at 50 MHz. At the chosed clock the delay is much higher.
- Maximum speed of the processor (taking the worse scenario) =82.494 MHz

12. Description of Processor Interfaces

- Inputs-
  - Clear- Switch to reset PC to 0 and reset the register file from FPGA
  - Clock changer switch- to change the clock from FPGA clock to manual clock for viewing single stepping
  - Single Stepping switch- this provides the method to give the manual clock
  - Register selector-5 switches to select amongst the 32 registers from the register file.
  - Instruction Memory-Dynamic updating of the instruction memory is done by providing 8 bits input at a time through switches. Program Counter is pointed to that instruction in the instruction memory using a switch if we want to access it dynamically. We can update the instruction memory dynamically up to 5 instructions. **NOT VERIFIED!**
- Outputs-
  - Seven segment LED to see the values of register file values using selector switches

Inputs to the processor are provided via switches on the FPGA.

13. Sanity testing of Processor-51

13.1 Test bench for ALU

We wrote one test bench for verifying if the ALU was being implemented as desired. In this test bench, each operation was verified for 20 operands, giving 10 legit results for all 18 operations. 3 operations for both operands having a positive value, 3 operations for both operands having a negative value, 2 operations for operand 1 being positive and operand 2 being negative, and 2 operations for operand 1 being negative and operand 2 being positive.

13.2 Test Bench for Decoder

Another Test bench was for decoder to verify if all the flags were generated correctly. We provided various 32 bit instructions and checked for the flags.

13.3 Sample codes

We used sample programs from the project description and also another one which had all the instructions of the processor

13.4 RC5

We chose two values of plain text from Ron Rivest's original paper to get the desired output.

14. Assembler

The assembler we used for generating machine codes for instruction memory is written in Python 2.7 which takes input of assembly code in CSV format and gives the final output in machine code in Hexadecimal

References

1. The RC5 Encryption Algorithm, Ronald L. Rivest
   Link: http://people.csail.mit.edu/rivest/Rivest-rc5rev.pdf