**Description of Assembly Program**

Having created and implemented the microarchitecture of the processor, the next step was to write the RC5 algorithm in assembly.

The Assembly program makes the following assumptions about the data memory organization:

Data Memory [0]: Holds the value of A

Data Memory [1]: Holds the value of B

Data Memory [2-5]: Holds the value of User Key

Data Memory [6-31]: Holds the Initial value of S\_array (magic number followed by Pw + nQw).

Data Memory [32]: Holds user preference to perform encryption/decryption.

We have a load module interface outside the processor’s module which accepts inputs from the user using switches and stores it in the data memory of the processor in the above fashion.

The assembly program’s structure is as follows: The program first reads Data memory of [32] to decide if it should perform encryption or decryption. It then stores that value in R29.

Irrespective of that preference, the program first performs key expansion and stores the resulting S\_matrix back in Data Memory [6-31]. Once this is done, it checks R29 to decide if it should perform encryption or decryption of the value stored in Data Memory [0] and [1].

If the user picks decryption, it branches to the section of code where decryption is performed and stores the result in R0 and R1.

But if the user picks encryption, it performs encryption and stores the results in R30 and R31 AND in Data memory of [0] and [1]. Following this, the program will perform decryption (using the encrypted value we just stored in Data Memory of [0] and [1] too and stores the result in R0 and R1. This is to verify that encryption happened correctly and R0 and R1 should hold the original value of A and B and R30 and R31 will hold the encrypted result.

Since we had only a limited instruction set that didn’t have a rotate or XOR instruction (both of which are essential for RC5), we had to work around it using the available instructions.

**Rotation by multiple single bit shifts**

In order to rotate, we used two shifts. After creating a copy of the value to be rotated in 2 registers, we shift one left by 1 bit and the other right by 31 bits and added the two. This gives us left rotate by one bit. For right shift, we shift left by 31 bits and right by 1 bit and then add. Two get larger rotate, loop this multiple times. A code snippet it shown below:

**ROTATE\_LOOP:** SUB R9, R9, R9 \\ Clear R9

ADD R9, R0, R9 \\ Store number to shifted in R9 (from R0)

SUB R11, R11, R11 \\ Clear R11

ADD R11, R0, R11 \\ Store copy of R0 in R11 too

SHL R11, R11, 0x0001 \\ Shift R11 left by one bit

SHR R9, R9, 0x001F \\ Shift R9 right by 31 bits

ADD R9, R11, R0 \\ Add R9 and R11. Single bit rotate value stored in R0.

ADDI R10, R10, 0x0001 \\ Increment loop counter

BNE R10, R17, **ROTATE\_LOOP** \\ R17 contains rotate amount

**XOR using universal gates**

In order to perform XOR operation, we implemented the XOR logic ( B + A). To implement the negations, we use the NOR operation. The code snippet is shown below:

SUB R5, R5, R5 \\ CLEAR R5

ADD R1, R5, R5 \\ R5 HAS COPY OF R1 WHICH IS VALUE OF B

NOR R5, R5, R5 \\ R5 HAS B BAR

SUB R6, R6, R6 \\ CLEAR R6

ADD R0, R6, R6 \\ R6 HAS COPY OF R0 WHICH IS VALUE OF UPDATED A

NOR R6, R6, R6 \\ R6 HAS A BAR

NOR R5, R0, R9 \\ R9 HAS BBAR.A

NOR R6, R1, R5 \\ R5 NOW HAS B.ABAR

OR R9, R5, R9 \\ R9 NOW HAS ABAR.B + BBAR.A = A XOR B

**Conversion to Machine Language.**

The complete assembly code was 226 lines long. To convert it to machine language, we wrote a short python script which read the assembly from a file and converted it to the specific opcodes, calculated the branch addresses etc and stored the binary output in a separate text file. We put this array of binary values in the instruction memory of the processor.