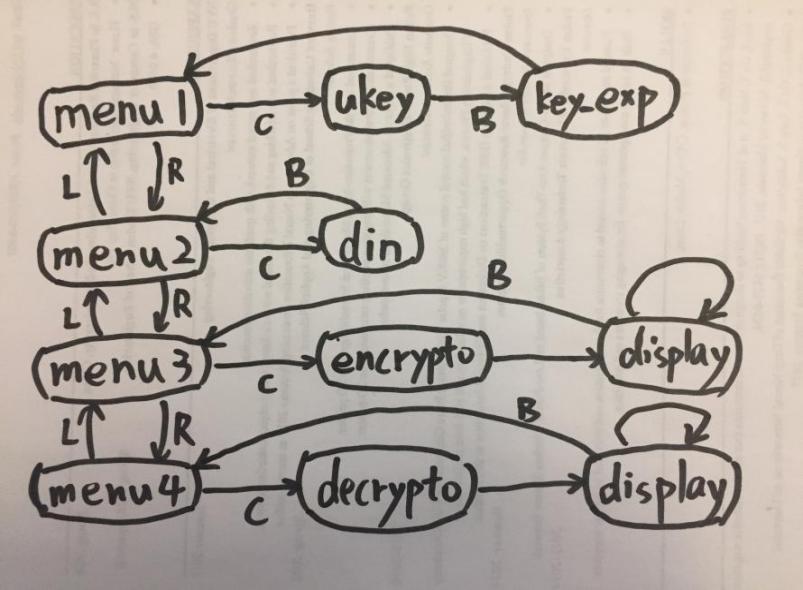
3. RC5 Implementation

3.1 Assmebly code

3.1.1 FSM diagram



This picture shows the diagram of our RC5 code. With the help of the input/output interface (modified ‘reg\_file’ modules), our program supports changing ‘ukey’ and ‘din’ on-the-fly. There are 4 buttons: LEFT(L), RIGHT(R), CONFIRM(C) and BACK(B). User can use these buttons to enter or exit different subprograms.

**UKEY**:

This is the subprogram for sending 128-bit ukey. The ukey will be stored into data memory [50 to 53] in little-endian.

**KEY\_EXP:**

This is the key expansion function. Round-key will be generated based on the ukey in the data memory. The expanded round-key will be stored into data memory [0 to 25].

**DIN:**

This is the subprogram for sending 64-bit input-text. The input-text will be stored into data memory [54 to 55] in little-endian.

**ENCRYPTION:**

This is the encryption function. 64-bit Cypher-text encrypted from input-text will be stored into data memory [30 to 31] in little-endian.

**DECRYPTION:**

This is the decryption function. 64-bit Plain-text decrypted from input-text will be stored into data memory [32 to 33] in little-endian.

**DISPLAY:**

This subprogram will show the output-text on the 7-segment display. User can use LEFT or RIGHT switch from higher or lower 32 bits of

Please check the source ‘rc5\_optimized.asm’ for more details. Our source code is well commented for reviewing.

3.1.2 Time complexity analysis

The critical part in the RC5 implementation is how fast we can run the key expansion, encryption and decryption in terms of cycles. So in this section we will briefly analyze the time complexity of rc5.

The running cycles of **key expansion** is **O(TL + TS + (Trot + Cexp)\*78 + C0)**, where TL is for L-Initialization, TS is for S-Initialization, Trot + Cexp is for one round of skey-Expansion, C0 is for other constant time operations. The running cycles of **encryption/decryption** is **O((Trot + Txor + C1)\*2\*12 + C0)**, where (Trot + Txor + C1)\*2 is for one round of encryption/decryption, C0 is for other constant time operations.

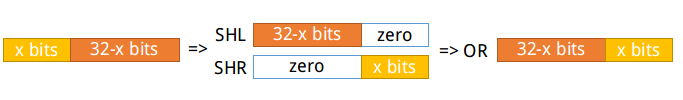
Notice that the constant factors 78 and 24 are relatively large, so reducing the cycles spend on rotation(Trot) and XOR(Txor) operation will be very helpful to reduce the total running cycles.

3.1.3 ROTATION optimization

In this project, we only have I-type shift instructions, which is a huge limitation.

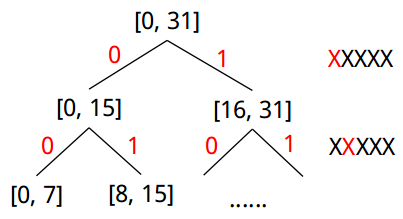
A trivial solution to implement left rotation using I-type shift is:

suppose the rotation amount[[1]](#footnote-0) is x, we can recursively shift left by 1 for x times. Then right shift by 1 for 32-x times. Finally combine them together using OR operation. This method will take at least 32\*3=96 cycles per rotation.



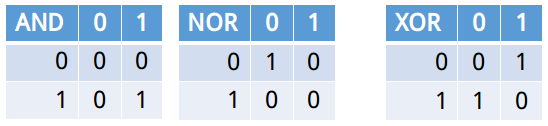
A smarter way is sequentially comparing the rotation amount with 0 to 31 and shifting the register by required amount directly. Because we don’t have instructions to compare a register with an immediate number, so each comparison will take 2 cycle. Thus, the running time per rotation in average is 2\*16+3=35 cycles.

An optimal way to do the rotation is using binary search, which takes only 13 cycles per rotation! We know that the rotation amount is a 5-bit number, so we can determine whether it’s in [0,15] or [16,31] by checking its 5th bit. Then use 4th bit to reduce the range again, so on and so forth. Picking 1 bit (AND) and branch if it’s 1 (BEQ) takes only 2 cycles. So the total running time is 2\*log(32)+3=13 cycles.



3.1.4 XOR optimization

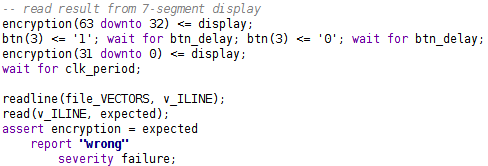
We don’t have XOR instruction, so we have to implement it using AND, OR, NOR. If we looking at the truth table of them, we can easily find an efficient solution: **A XOR B = (A NOR B) NOR (A AND B),** which takes only 3 cycles.



3.2 Testbench

To meet the requirement, we tested 1000 random cases for each of key\_expansion, encryption and decryption. We also tested another 10 random ukey, each with 100 cases of encryption and decrpytion. So there were in total 5000 cases tested in our testbench.

The random cases along with their expected result were generated from a python script (rc5.py), and stored in a text file for testbench to read. Assertion statement was used in the testbench for automatically checking.



3.3 Fuctional simulation

screenshots and explanation.

report number of cycles for key\_exp, enc, dec here.

3.4 Timing simulation

screenshots and explanation.

3.5 FPGA Implementation

1. The rotation amount is the lower 5 bits of the operand, so it’s in range from 0 to 31. [↑](#footnote-ref-0)