# RISC-V OTTER Extension

# Encryption Module and ISA

CPE 521- 01

Computer Systems

Spring 2023

Students: Maanav Patel, Nathan Jaggers,

Weston Keitz, Ethan Najmy

Project Due: 06/09/23

Instructor: Dr. Danowitz

**Overview**

We plan to create an encryption module on the Basys board that is implemented with the OTTER. Initially, we will implement this on the basic OTTER from 233, then if time permits, we will try to implement it on the pipelined OTTER from 333.

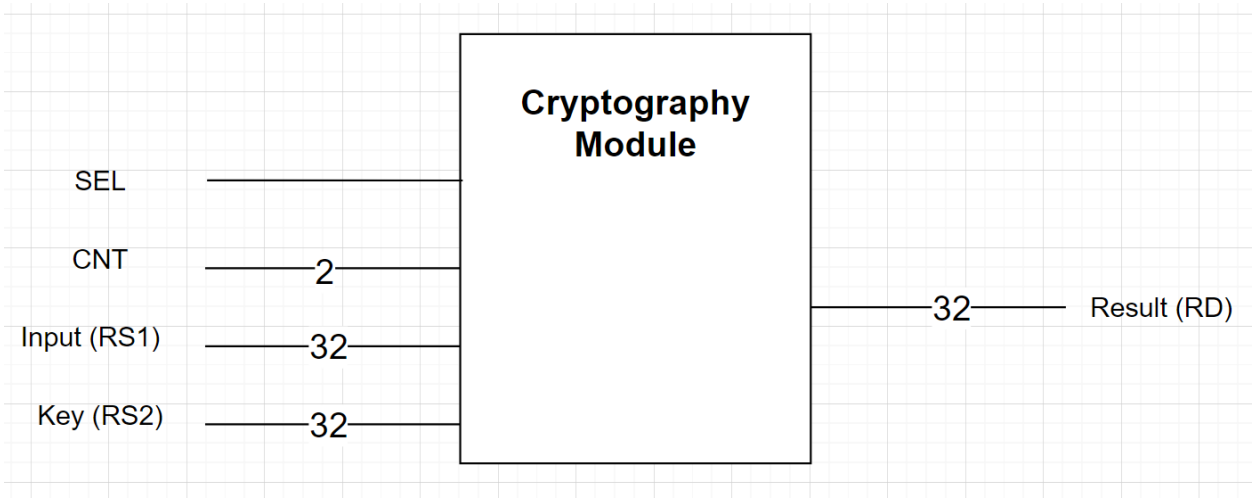


Figure 1 : Cryptography Module

The encryption module will be connected to the register file and will output an encryption key from one of the source registers. A symmetric encryption algorithm will be used, as an asymmetric encryption algorithm is not feasible since we are constrained locally to the Basys board.

Embedded within the black box diagram there will be a module for encryption and decryption. The select bit will control a MUX to the data paths to encrypt or decrypt. This will hopefully reduce the control logic needed when inserting the module into the OTTER. Additionally, this will simplify our testing and implementation process as we will be able to test encryption and decryption individually.

Our approach is to create the module and create a testbench to test functionality independent of the OTTER. Then we will insert it and add additional control logic and instructions to utilize the module.

**Extending RISC-V ISA for Encrypt and Decrypt Functions**

Adding encrypt and decrypt ability to the OTTER can be accomplished through hardware extensions or a software program. Besides implementing the correct hardware description of the cryptography module in Verilog and supporting custom control logic, custom compiler extensions need to be included. For our cryptography module, we are adding an encrypt and a decrypt instruction to extend the function of the instruction set architecture. The following steps are followed from the tutorial from Garrett O'neill [1].

We decided to use a custom opcode to characterize the two new instructions. Since an encrypt and decrypt instruction requires four rounds of computation at a cycle per round. This creates the need for a custom opcode to signal a counter in the execute state. The counter will increment from zero to three, then reset once the encryption/decryption rounds complete. We chose the corresponding custom opcode, 0x5B, that is supported as an extension in the RISC-V ISA [5]:

```
7'b1011011
```

The function register implemented in the base OTTER is to discern the proper ALU source selection for arithmetic and immediate operations. The three least significant bits of the function register primarily the control branch instructions. For our purposes, the function register will accomplish the choice between encrypt or decrypt as an input to the crypto module. The seven most significant bits of the function register have no impact on the module. We opted to set all to zero.

```
funct7: 7'b0000000
```

For the three least significant bits of the function register, we found two unused combinations to discern between an encrypt or decrypt instruction. This will avoid interferences with other funct3 calls. For the selector of the module, the only bit that switches modes is the least significant bit.

```
//encrypt funct3
funct3: 3'b010
//decrypt funct3
funct3: 3'b011
```

The instructions contain the same structure as an R-type instruction. The cryptography extensions require two source registers for the data and key and a destination register for the encrypted or decrypted signal.

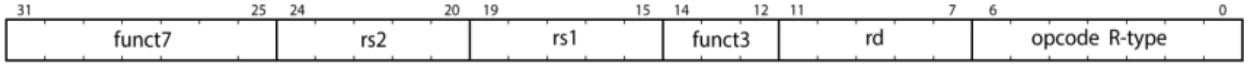


Figure 2. R-Type Instruction Format [3]

To add the instructions to the compiler, the ISA must recognize a mask and mask that identify the instruction.

Table 1.1 Match Define

| | funct7 | rs2 | rs1 | funct3 | rd | opcode |
|---|---|---|---|---|---|---|
| encrypt | 0000000 | 00000 | 00000 | 010 | 00000 | 0011100 |
| decrypt | 0000000 | 00000 | 00000 | 011 | 00000 | 0011100 |

Table 1.2 Mask Define

| | funct7 | rs2 | rs1 | funct3 | rd | opcode |
|---|---|---|---|---|---|---|
| encrypt | 1111111 | 00000 | 00000 | 111 | 00000 | 1111111 |
| decrypt | 1111111 | 00000 | 00000 | 111 | 00000 | 1111111 |

Define the following macros to the following header files:

```
/riscv-gnu-toolchain/binutils/include/opcode/riscv-opc.h
/riscv-gnu-toolchain/gdb/include/opcode/riscv-opc.h
```

```
//Opcode: 7'b0011100
//Function: 10'b0000000010
#define MATCH_ENCRYPT 0x201C
#define MASK_ENCRYPT 0xFE00707F
DECLARE_INSN(encrypt, MATCH_ENCRYPT, MASK_ENCRYPT)

//Opcode: 7'b0011100
//Function: 10'b0000000011
#define MATCH_DECRYPT 0x301C
#define MASK_DECRYPT 0xFE00707F
DECLARE_INSN(decrypt, MATCH_DECRYPT, MASK_DECRYPT)
```

Add the following instructions and aliases to the following files:

```
/riscv-gnu-toolchain/binutils/opcode/riscv-opc.c
/riscv-gnu-toolchain/gdb/opcode/riscv-opc.c
```

```
{"encrypt", 0, INSN_CLASS_I, "d,s,t", MATCH_ENCRYPT, MASK_ENCRYPT, match_opcode, 0},
{"decrypt", 0, INSN_CLASS_I, "d,s,t", MATCH_DECRYPT, MASK_DECRYPT, match_opcode, 0},
```

In the assembly source code, the encrypt and decrypt instructions follow hte same syntax convention as an R-type instruction.

```
encrypt x25, x25, x24
decrypt x25, x26, x27
```

```
add     rd,rs1,rs2
```

Figure 3. R-type Instruction Assembly Code [3]

The compiled machine code matches the original instruction format and the source/destination registers match. This indicates that the new instructions added to the RISC-V GNU toolchain are correct.

```
74: 018cacdb                encrypt     s9,s9,s8
94: 01bd3cdb                decrypt     s9,s10,s11
```

Table 2. Instruction Binary Decomposition and Implementation Instructions

| | funct7 | rs2 | rs1 | funct3 | rd | opcode |
|---|---|---|---|---|---|---|
| encrypt | 0000000 | 11000 | 11001 | 010 | 11001 | 1011011 |
| decrypt | 0000000 | 11011 | 11010 | 011 | 11001 | 1011011 |

If an additional source register to the instruction for an initialization vector is needed, the funct7 bits are all unused and can be replaced with a 5-bit register address value to indicate an RS3 address. The compiler settings would need to be tweaked to support this and an additional read port would need to be included from the register file.

---

**Implementing the Encryption/Decryption Module**

The encryption and decryption module will implement cipher block chaining (CBC). CBC is an encryption/decryption method where a sequence of bits is encrypted/decrypted with a cipher key and initialization vector. If we examine encryption, the general idea is that the previously encrypted block is used in conjunction with the key to encrypt the new plaintext block. The initialization vector is used to encrypt the first block of plaintext as there is no preceding block to use. In CBC, there can be any encryption algorithm used for the step between the plaintext and key, but typically the previous block of ciphertext is XOR'd with the current block of plaintext. This creates an easy way to decrypt the same block, as to undo an XOR operation, you simply use another XOR. Furthermore, we decided to use an XOR operation between the plaintext and the key itself for encryption, so essentially we are using a threeway XOR for both encryption and decryption, which ensures that we will be able to complete both operations in a reasonable amount of clock cycles.

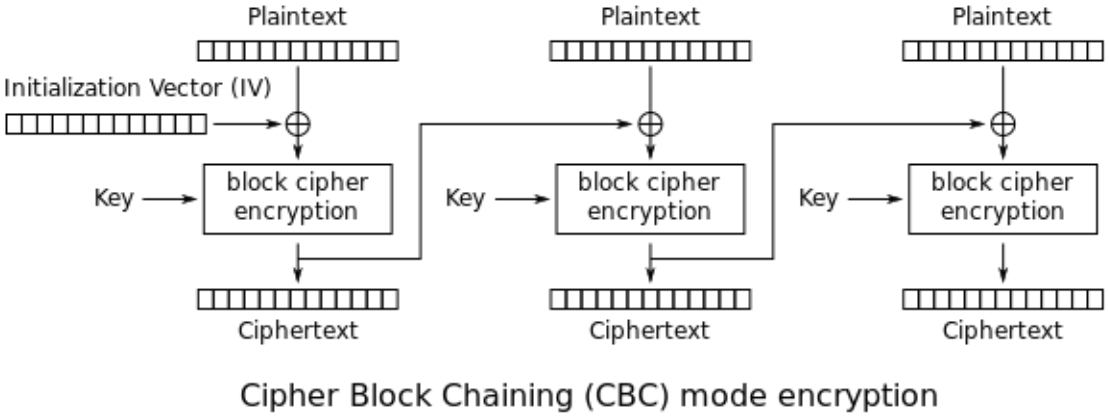


Figure 3: Cipher Block Chaining Encryption Diagram

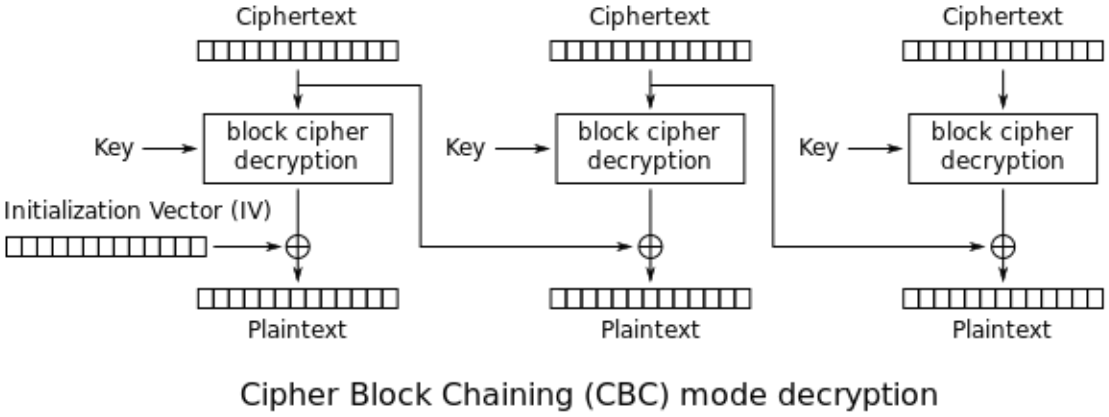


Figure 4: Cipher Block Chaining Decryption Diagram

```
module Cryptography_Module(
    input [31:0] data_in,
    input [31:0] key,
    input [1:0] cnt,
    input sel,
    output reg [31:0] result
    );

    reg iv = 8'b10011011;

    always @(cnt, data_in, key, sel)
    begin
        if(sel == 0)
        begin
            case(cnt)
                0: result = {24'b0, (iv ^ data_in[7:0] ^ key[7:0])};
                1: result |= {16'b0, (result[7:0] ^ data_in[15:8] ^ key[15:8]), 8'b0};
                2: result |= {8'b0, (result[15:8] ^ data_in[23:16] ^ key[23:16]), 16'b0};
                3: result |= {(result[23:16] ^ data_in[31:24] ^ key[31:24]), 24'b0};
```

```
                default: result = 0;
            endcase
        end
        else
        begin
            case(cnt)
                0: result = {24'b0, iv ^ data_in[7:0] ^ key[7:0]};
                1: result |= {16'b0, (data_in[7:0] ^ data_in[15:8] ^ key[15:8]), 8'b0};
                2: result |= {8'b0, (data_in[15:8] ^ data_in[23:16] ^ key[23:16]), 16'b0};
                3: result |= {(data_in[23:16] ^ data_in[31:24] ^ key[31:24]), 24'b0};
                default: result = 0;
            endcase
        end
    end
endmodule
```

Figure 5: Cryptography Module Verilog Code

The way the module is implemented is by splitting our logic into blocks, one for encryption and one for decryption, and we expect the cnt input to be incremented on each clock cycle as the FSM will remain in the execute state.

*OpCode*

We had to modify the OTTER and add an additional opcode that can be processed for encryption/decryption instructions. Below is that opcode:

```
ENCRY = 7'b1011011
```

*Counter Variable*

The counter variable is used to determine which state of encryption/decryption is currently being done. As mentioned, we split the data provided by the user into 4, 8-bit chunks, and apply the CBC algorithm on each chunk. This counter variable comes from the FSM and gets set with the following code:

```
EXECUTE: begin
    if(CU_OPCODE !=LOAD) begin
        state <= FETCH;
        if(CU_INT || CU_prevINT) state <= INTER;
    end else if (CU_OPCODE == ENCRY) begin
        if (crypto_count == 3) begin
            crypto_count = 0;
            state <= FETCH;
        end else begin
            crypto_count <= crypto_count + 1;
            state <= EXECUTE;
           end
```

```
        end else
           state <= WB;
        end
```

Above, when the FSM enters the FETCH state and the opcode is determined to match the ENCRY opcode, it begins the counter. We estimate that each cycle an instruction can be encrypted/decrypted so on each clock edge, we update the counter, therefore telling the crypto module to update and begin the next stage. Once all 4 8-bit chunks have been processed, the counter gets reset and the FSM will transition into FETCH state.

*Select Variable*

The select variable is used to determine encryption or decryption inside the crypto module. This variable is configured inside the decoder and is determined by the CU_FUNC3 opcode (IR[14:12]). The custom encrypt/decrypt instruction in RISC-V is set with the proper FUNC3 opcode. Below is the code that sets the select variable:

```
case(CU_FUNC3)
    3'b000: brn_cond = CU_BR_EQ;    //BEQ
    3'b001: brn_cond = ~CU_BR_EQ;   //BNE
    3'b100: brn_cond = CU_BR_LT;    //BLT
    3'b101: brn_cond = ~CU_BR_LT;   //BGE
    3'b110: brn_cond = CU_BR_LTU;   //BLTU
    3'b111: brn_cond = ~CU_BR_LTU;   //BGEU
    3'b010: cryptoSel = 0;          // ENCRYPT
    3'b011: cryptoSel = 1;          // DECRYPT
    default: begin brn_cond = 0; cryptoSel = 0;
endcase
```

This cryptoSel line is then sent over to the crypto-module and will determine how to go about processing the data.

*Control Signal Modifications*

Implementation of this required modifications to PCWRITE & REGWRITE control signals. When we are encrypting/decrypting, we do not want the PCWRITE to be enabled because we want the PC to stay at it's current value. This is because the crypto-module takes 4 cycles to complete. On the 4th cycle, PCWRITE is enabled again. REGWRITE is set to high when we are finished with the encryption, meaning the opcode is not ENCRY, or the count is 4. This is to prevent the data being written before it is fully encrypted or decrypted, which could lead to data hazards or improperly encrypted/decrypted data. Below are the control signal logic for each:

```
assign CU_PCWRITE = (state==1 && CU_OPCODE!=LOAD && ((CU_OPCODE != ENCRY)
|| (encryptCount == 3))) || (state==2 && CU_OPCODE==LOAD) ||(state==INTER);
```

```
assign CU_REGWRITE = (state == 2) || ((state == 1) && (CU_OPCODE != BRANCH
&& CU_OPCODE !=LOAD && CU_OPCODE !=STORE && ~MRET && ((CU_OPCODE!= ENCRY)
|| (encryptCount ==3))) );
```

*RF MUX Modification*

In order to save the encrypted/decrypted data to the RF, we needed to add another input to the MUX that feeds the data to be written into the RF. This meant we also had to modify the CU_RF_WR_SEL selector for the MUX. Below is the following code for that logic:

```
case(CU_OPCODE)
    JAL:     CU_RF_WR_SEL=0;
    JALR:    CU_RF_WR_SEL=0;
    LOAD:    CU_RF_WR_SEL=2;
    SYSTEM:  CU_RF_WR_SEL=1;
    ENCRY:   CU_RF_WR_SEL=4;
    default: CU_RF_WR_SEL=3;
endcase
```

This is simply selected based on the opcode that is sent into the decoder, which shows some of the inherent benefits of using a new, custom opcode for the new instructions instead of a R-type opcode. If we reused the R-type structure, we would have to add additional wires and logic to check for both opcode and func3 in the FSM, which also undermines the design philosophy of opcodes and func3 in general.

**Test and Results**

To test if our cryptographic module worked as we expected, we created a testbench in Vivado. This simulation took plaintext, encrypted it, and then decrypted it. We chose a memorable 32-bit hex phrase, 0xdeadbeef,  and a key, 0xc0ffeeee, to use for our encryption and decryption. The main indicator showing that it worked was that the plaintext we encrypted was the same plaintext we got after decrypting. The results of the simulation can be seen in the two figures below.

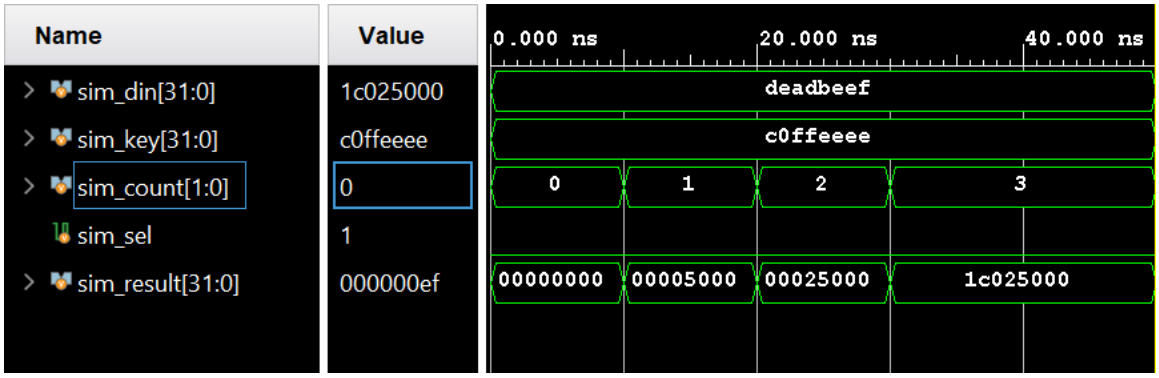


Figure 6: Testbench Results for Cryptography Module - Encryption

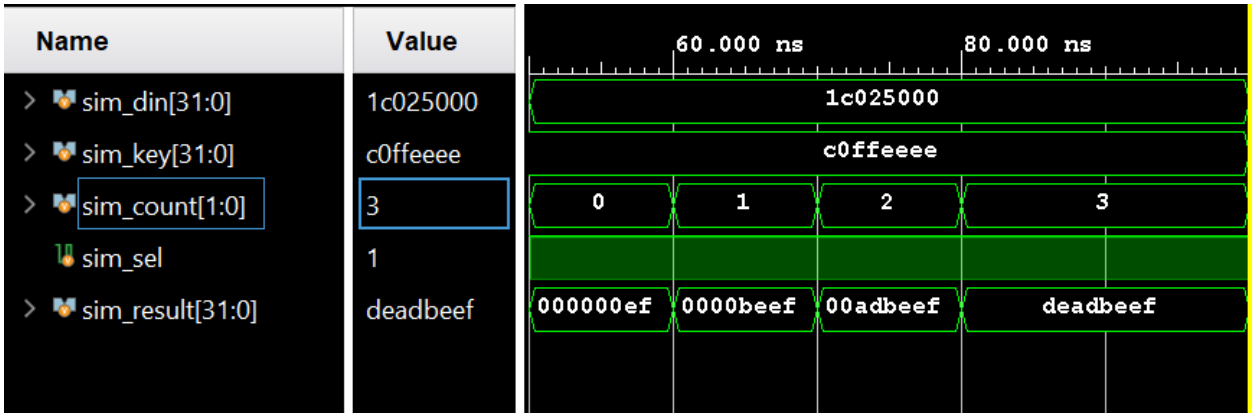


Figure 7: Testbench Results for Cryptography Module - Decryption

To use the Cryptography Module with the Otter, we created an Assembly program that would use our custom instructions that utilize our module. To start the encryption process we use the 12 switches on the right of the board to pick a number to encrypt. The module is capable of encrypting 32-bit numbers but with the limited resources we have on the Basys board, and for demonstration purposes, we only allow 12-bit numbers for input. Once a number has been selected, flipping the leftmost switch, SW15, allows us to input our key.

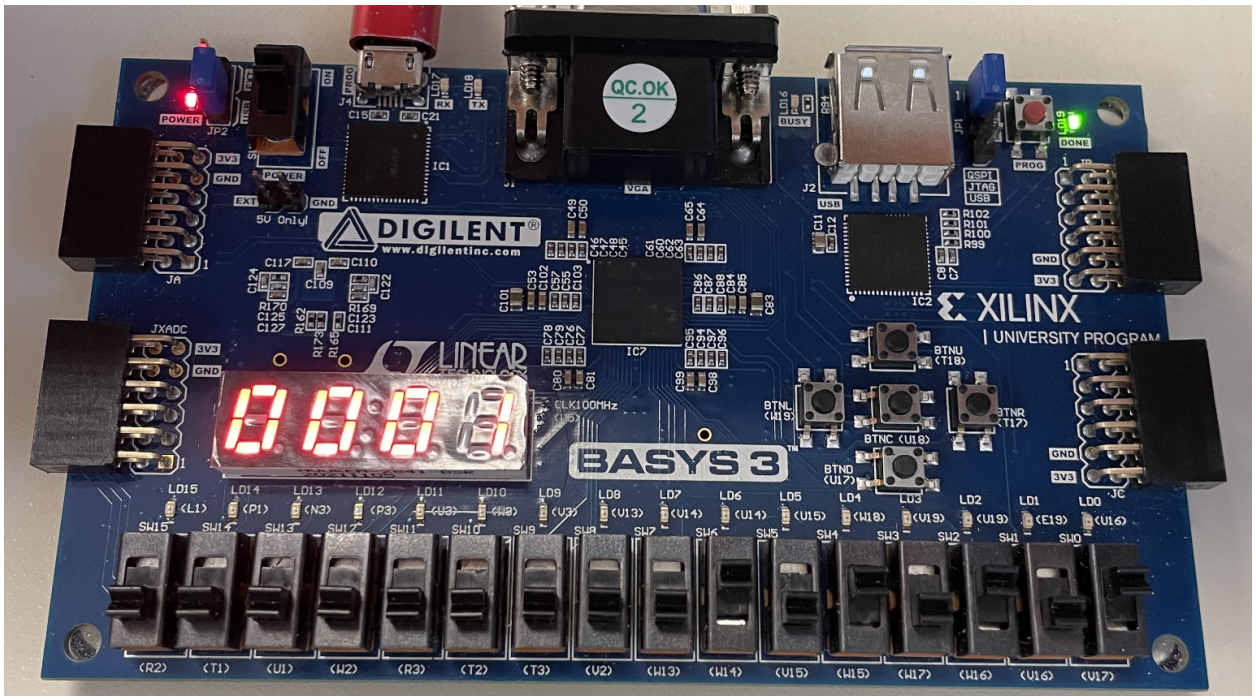


Figure 8: Otter with Cryptography Module Test - Inputting Data = 85

Similar to the limited 12-bit input for data, we are limited to a 12-bit input for the key. After a key has been selected the second to leftmost switch, SW14, can be flipped and that will encrypt the input. Although our inputs are both 12-bits, the full 32-bit encryption occurs.

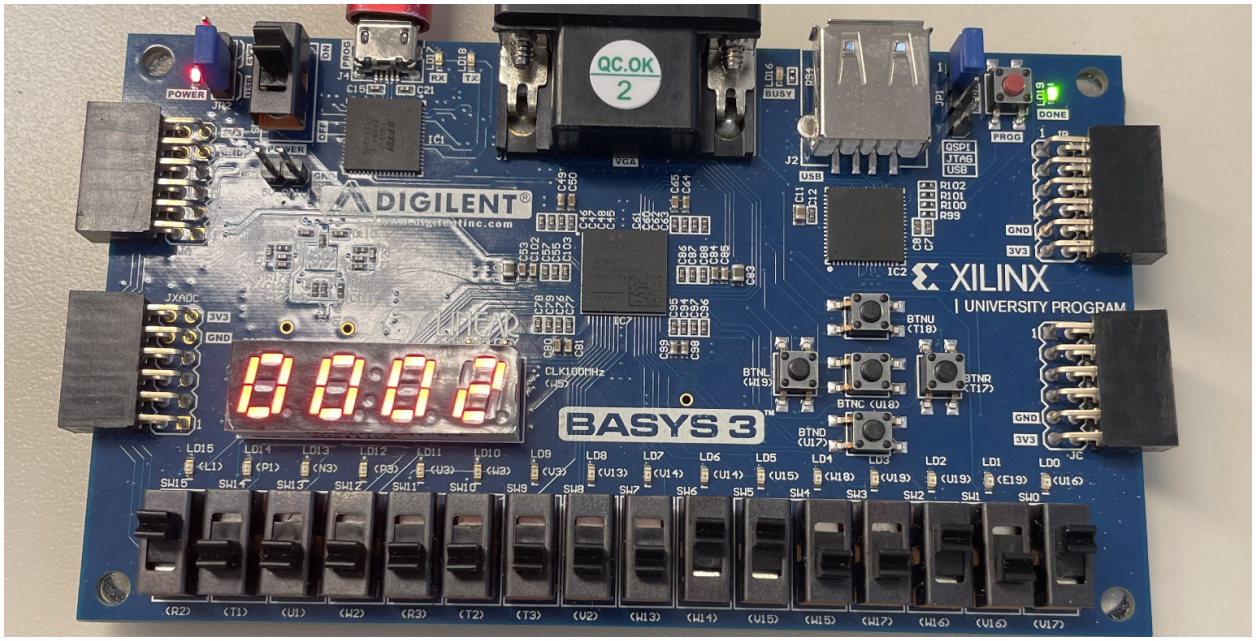


Figure 9: Otter with Cryptography Module Test - Inputting Key = 101

With SW14, and SW15 up, we have encrypted our data and we can see the results on the 7-segment display. Here we see the last 4 digits of our 32bit encrypted value.

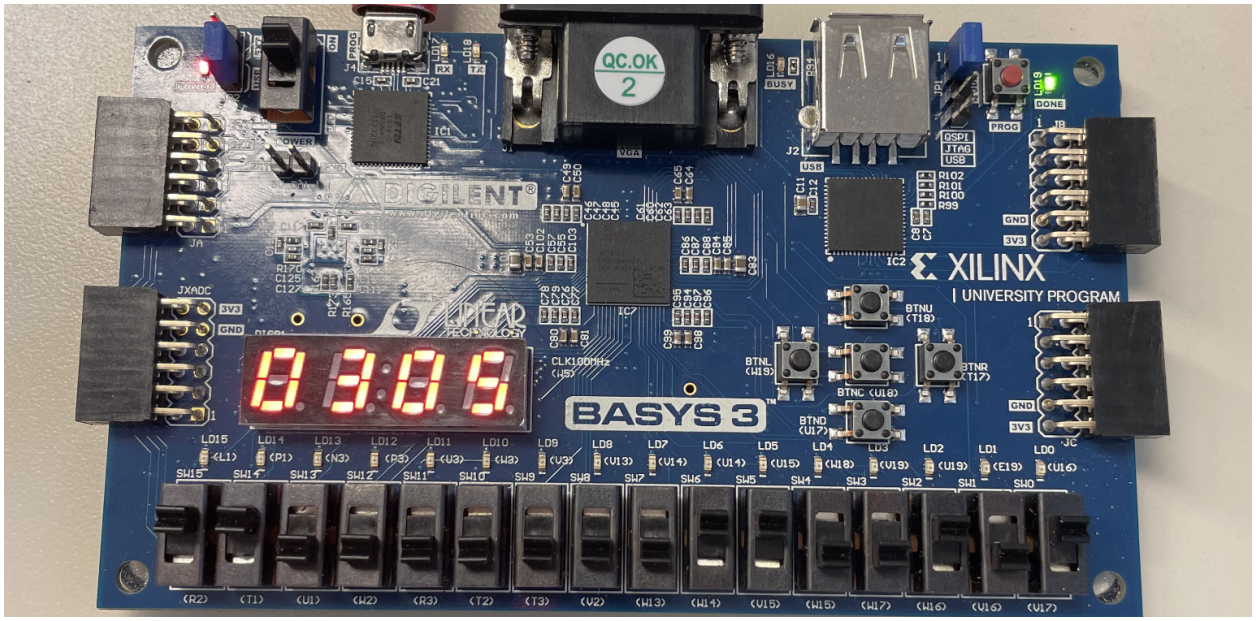


Figure 10: Otter with Cryptography Module Test - Encrypted Message = 305

To decrypt our input, we simply flip the third most switch, SW13. This reverses the encryption on our 32-bit value and shows the plaintext value on the 7-segment display.

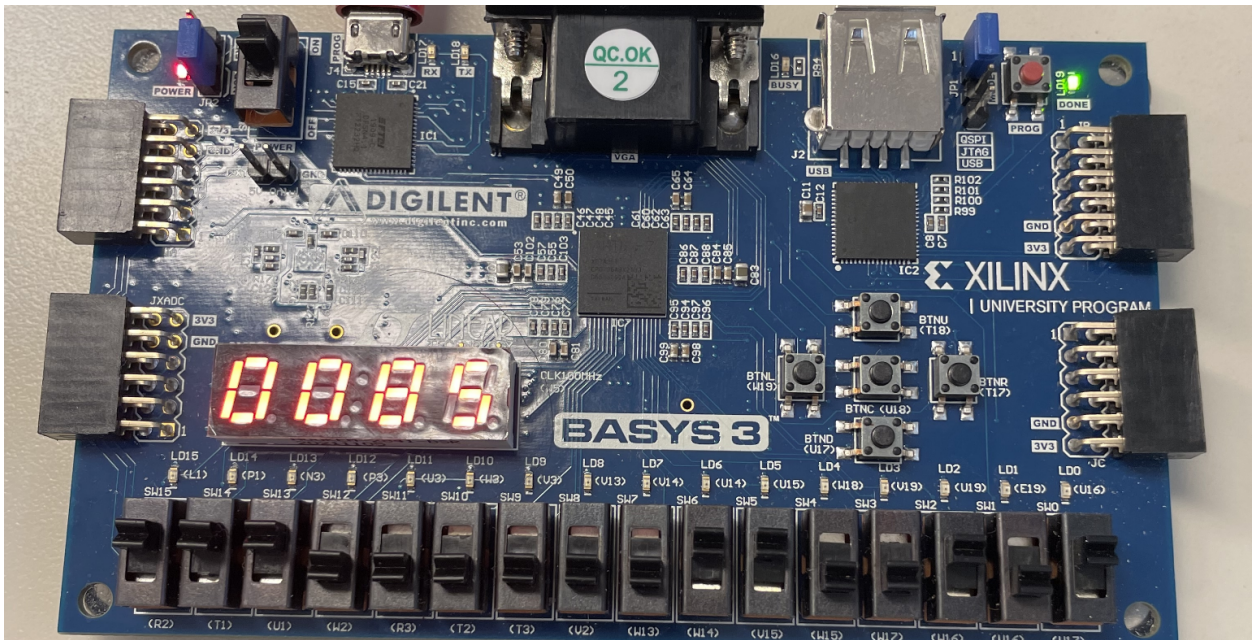


Figure 11: Otter with Cryptography Module Test - Decrypted Message = 85

**Conclusion**

Overall, this project was very successful and achieved it's goal. We were successfully able to implement a new instruction in RISC-V, and were successfully able to implement an encryption/decryption module in the OTTER that uses custom opcode to encrypt/decrypt a value entered by the user with a key provided by the user. The results in simulation show a successful implementation of the module on it's own - encrypting 'deadbeef' with the key 'c0ffeeee'. This gave an encrypted value of '1c025000' and we were successfully able to decrypt using the same key. We also were able to get a successful implementation working on the Basys board. On the board, we encrypted the value 85 with the key 101. When encrypted we got the value 305 and decrypting gave back our original number, 85. This project was a great success and in the future we could consider adding a more sophisticated encryption/decryption method such as AES, or also implementing the ability to have a custom initialization vector for the first stage of encryption and decryption.

---

**References**

[1] Garrett O'neill, "Adding Custom Instructions to RISC-V Compiler" [Online]. Available: https://github.com/geoneill12/Extending_RISCV/blob/main/Adding%20Custom%20Instructions%20To%20RISC-V%20Compiler.pdf [Accessed: 1-June-2023]

[2] Jonathan Skelly, "Viability and Implementation of a Vector Cryptography Extension for RISC-V" [Online]. Available: https://digitalcommons.calpoly.edu/theses/2573/ [Accessed: 6-June-2023]

[3] James Mealy, “The RISC-V OTTER Assembly Language Manual”

[4] Maanav Patel, Nathan Jaggers, Weston Keitz, Ethan Najmy, “GitHub - CPE521-Cryptography-Module”, https://github.com/enajmy/CPE-521-Cryptography-Module/tree/main

[5] H. Kim and T. Blaise, Implementing hardware extensions for multicore RISC-V gpus - github pages, https://carrv.github.io/2022/papers/CARRV2022_paper_11_Blaise.pdf (accessed May 17, 2023).