

**UNIVERSITA DEGLI STUDI DI NAPOLI FEDERICO II**  
**DEPARTMENT OF ELECTRICAL AND INFORMATION**  
**TECHNOLOGY ENGINEERING**

**EMBEDDED SYSTEMS FINAL PROJECT**

**INSTRUCTION SET EXTENSION OF RISC-V PROCESSOR**  
**FOR ASCON AND SHA-2 LIGHTWEIGHT CRYPTOGRAPHY**  
**ALGORITHMS**

**Yagiz YAGMUR**

**Matricola: 000123534**

# TABLE OF CONTENTS

TABLE OF CONTENTS .....	2
ABBREVIATIONS.....	4
LIST OF TABLES .....	5
LIST OF FIGURES.....	6
SUMMARY .....	7
1. SHA-256.....	8
1.1    Hashing.....	8
1.2    SHA-256 Algorithm Steps.....	8
2. ASCON .....	10
2.1    ASCON Permutation .....	10
2.2    ASCON Encryption Modes .....	11
2.3    ASCON Hashing .....	11
3. RISC – V .....	12
3.1    RISC-V ISA (RV32I) .....	12
3.2    RISC-V Registers .....	13
3.3    CV32E40S .....	13
4. SIMULATION SETUP .....	14
5. ASCON IMPLEMENTATION AND SIMULATION .....	18
5.1    Ascon-C Programs.....	18
5.2    Instruction Set Extension for ASCON.....	19
5.3    Modification of the Toolchain and the Simulator for the Custom Instructions.....	20
5.4    Simulation of ASCON Algorithms With and Without the Custom Instructions .....	23
6. SHA-256 IMPLEMENTATION AND SIMULATION .....	26
6.1    SHA-256 Algorithm in C .....	26
6.2    SHA-256 Extensions .....	26
7. RESULT ANALYSIS AND CONCLUSION.....	28
7.1    Effects of the Extensions .....	28
7.2    Effects of the Optimizations .....	28
7.3    Comparison Between ASCON Hash and SHA-256.....	28
7.4    Conclusion.....	29
8. FURTHER PRACTICE: HARDWARE IMPLEMENTATION .....	30
REFERENCES.....	32
APPENDICES.....	33
A        APPENDIX A.....	33

B	APPENDIX B .....	34
C	APPENDIX C.....	37
D	APPENDIX D.....	38
E	APPENDIX E .....	40

## **ABBREVIATIONS**

**ABI:** Application Binary Interface

**AEAD:** Authenticated Encryption and Decryption

**ALU:** Arithmetic Logic Unit

**ELF:** Executable and Linkable File

**FSM:** Finite State Machine

**GCC:** GNU Compiler Collection

**ISA:** Instruction Set Architecture

**LWC:** Lightweight Cryptography

**NIST:** National Institute of Standards and Technology

**PK:** Proxy Kernel

**RISC:** Reduced Instruction Set Computer

**RTL:** Register Transfer Level

**SHA:** Secure Hash Algorithm

## LIST OF TABLES

Table 5.1: opt32 AEAD simulation results .....	23
Table 5.2: opt32_lowsize AEAD simulation results .....	24
Table 5.3: opt32 hashing simulation results .....	24
Table 5.4: opt32_lowsize hashing simulation results.....	25
Table 6.1: Zknh Instruction Set Extension .....	26
Table 6.2: SHA-256 hashing simulation results .....	27

## LIST OF FIGURES

Figure 2.1: ASCON S-box (a) and linear layer (b) .....	10
Figure 2.2: ASCON encryption algorithm illustration .....	11
Figure 2.3: ASCON hashing algorithm illustration .....	11
Figure 3.1: RV32 instruction formats .....	12
Figure 3.2: RISC-V general purpose registers .....	13
Figure 4.1: AEAD simulation output .....	17
Figure 4.2: Parts from instruction_log.txt and stats.txt .....	17
Figure 5.1: main.c encryption/decryption test program .....	18
Figure 5.2: main.c hashing test program .....	19
Figure 5.3: riscv-gnu-toolchain/binutils/opcodes/riscv-opc.c modification .....	20
Figure 5.4: riscv-gnu-toolchain/binutils/include/opcode/riscv-opc.h modification .....	20
Figure 5.5: riscv-gnu-toolchain/gcc/gcc/config/riscv/riscv.md modifications .....	20
Figure 5.6: /disasm/disasm.cc , /riscv/riscv.mk.in, /riscv/encoding.h modifications .....	21
Figure 5.7: rot.h .....	21
Figure 5.8: roti.h .....	21
Figure 5.9: sbox.h and sboxi.h .....	22
Figure 5.10: Custom instructions seen in the assembly code .....	23
Figure 8.1: alu_opcode_e struct modification .....	30
Figure 8.2: ALU ResultMux modification .....	30
Figure 8.3: Custom instruction addition to the ALU .....	31
Figure 8.4: Sbox module schematic .....	31

## SUMMARY

In the era of computing, ensuring data security is crucial. Therefore, robust, and efficient lightweight cryptographic algorithms are often used in various applications. SHA-256, a widely recognized cryptographic hash function from the SHA-2 family and ASCON, a finalist in the NIST lightweight cryptography competition provides efficient and secure lightweight data encryption. For improving the performance of these algorithms, designing a processor specifically for encryption, which is high-cost and time consuming, or extending the instruction set of an existing processor is possible.

RISC-V, an open-source instruction set architecture (ISA) is used for implementation of these algorithms in this project. New instructions for accelerating the performance are introduced to the 32-bit instruction set and simulated to gather results between these two algorithms in terms of space, time, power, and overall performance.

# 1. SHA-256

SHA-2 is a family of cryptographic hash functions designed by the National Security Agency (NSA) and published by the National Institute of Standards and Technology (NIST) in 2001. SHA-256 is a member of the SHA-2 family and produces a 256-bit (32-byte) hash value. The SHA-256 algorithm ensures data integrity by producing a unique, fixed-size hash value from an arbitrary amount of input data. It is widely used in various security applications and protocols, including TLS, SSL, and Bitcoin.

## 1.1 Hashing

Hashing is the process of scrambling raw information to the extent that it cannot reproduce it back to its original form. It takes a piece of information and passes it through a function that performs mathematical operations on the plaintext, producing the output called digest. SHA-256 algorithm produces a 32-byte digest. By design, all hash functions are irreversible, meaning it is not possible to obtain the original value from the digest.

## 1.2 SHA-256 Algorithm Steps

**Padding:** Extra bits are added to the original message until the length is 64 bits short of a multiple of 512. The first padded bit is zero while the others are zero. The extra 64-bits are added, representing the length of the original input.

**Initialization:** Eight 32-bit words (constants) are used to initialize the hash value. These constants are derived from the first 32 bits of the fractional parts of the square roots of the first eight prime numbers. Also 64 different keys are stored in an array (K[63:0]).

**Message Schedule:** Input data is divided into 32-bit words. 48 extra words are added using the algorithm shown below and w[63:0] array called the message schedule is created.

```
for i = 16 to 63:
    s0 = (W[i-15] >> 7 | W[i-15] << 25) ^ (W[i-15] >> 18 | W[i-15] << 14) ^ (W[i-15] >> 3)
    s1 = (W[i-2] >> 17 | W[i-2] << 15) ^ (W[i-2] >> 19 | W[i-2] << 13) ^ (W[i-2] >> 10)
    W[i] = W[i-16] + s0 + W[i-7] + s1
```



**Compression Loops:** Variables a to h are set to current hash values. Then the compression loop is runs. This process repeats 64 times for each chunk.

```
for i = 0 to 63:
    S1 = (e >> 6 | e << 26) ^ (e >> 11 | e << 21) ^ (e >> 25 | e << 7)
    ch = (e & f) ^ ((~e) & g)
    temp1 = h + S1 + ch + K[i] + W[i]
    S0 = (a >> 2 | a << 30) ^ (a >> 13 | a << 19) ^ (a >> 22 | a << 10)
    maj = (a & b) ^ (a & c) ^ (b & c)
    temp2 = S0 + maj

    h = g
    g = f
    f = e
    e = d + temp1
    d = c
    c = b
    b = a
    a = temp1 + temp2
```

**Chunk Loops:** This step occurs if the input is bigger than 512 bits so we need to split it into 512-bits chunks. When one compression process is over, hash values change accordingly to their last value.

```
h0 = h0 + a
h1 = h1 + b
h2 = h2 + c
h3 = h3 + d
h4 = h4 + e
h5 = h5 + f
h6 = h6 + g
h7 = h7 + h
```

**Finalize:** Hash values from the last iteration are concatenated to produce the digest.

```
digest = h0 || h1 || h2 || h3 || h4 || h5 || h6 || h7
```

## 2. ASCON

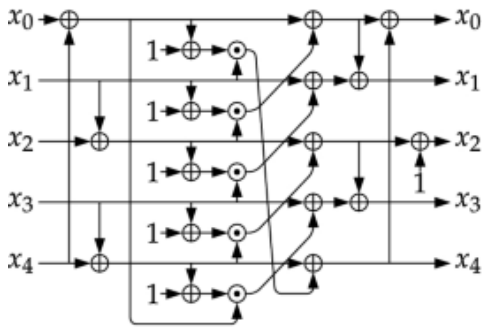
Ascon is a family of authenticated encryption and hashing algorithms designed for lightweight applications and ease of implementation, including countermeasures against side-channel attacks. Ascon's features include a unified approach to authenticated encryption and hashing with a single lightweight permutation and sponge-based modes of operation. It is also provably secure with keyed finalization, making it robust and reliable.

Ascon is efficient and versatile both on software and hardware implementations. It is optimized for fast performance, using a pipelinable, bit-sliced 5-bit S-box for 64-bit architectures. The algorithm is scalable for increased security or higher throughput and resists timing and side-channel attacks by avoiding table look-ups and additions. With a recommended key size and tag size of 128 bits, it provides minimal overhead, ensuring ciphertext length matches plaintext length.

### 2.1 ASCON Permutation

All Ascon family members use the same lightweight permutation. This permutation iteratively applies an SPN-based round transformation, consists of the following three steps which operate on a 320-bit state divided into 5 words  $x_0, x_1, x_2, x_3, x_4$  of 64 bits each:

- Addition of Round Constants: xors a round specific 1-byte constant to word  $x_2$ .
- Nonlinear Substitution Layer: applies a 5-bit S-box 64 times in parallel in a bit-sliced fashion (vertically, across words).
- Linear Diffusion Layer: xors different rotated copies of each word (horizontally, within each word).



(a) ASCON's 5-bit S-box  $\mathcal{S}(x)$

$$\begin{aligned} x_0 &\leftarrow \Sigma_0(x_0) = x_0 \oplus (x_0 \ggg 19) \oplus (x_0 \ggg 28) \\ x_1 &\leftarrow \Sigma_1(x_1) = x_1 \oplus (x_1 \ggg 61) \oplus (x_1 \ggg 39) \\ x_2 &\leftarrow \Sigma_2(x_2) = x_2 \oplus (x_2 \ggg 1) \oplus (x_2 \ggg 6) \\ x_3 &\leftarrow \Sigma_3(x_3) = x_3 \oplus (x_3 \ggg 10) \oplus (x_3 \ggg 17) \\ x_4 &\leftarrow \Sigma_4(x_4) = x_4 \oplus (x_4 \ggg 7) \oplus (x_4 \ggg 41) \end{aligned}$$

(b) ASCON's linear layer with 64-bit functions  $\Sigma_i(x_i)$

Figure 2.1: ASCON S-box (a) and linear layer (b)

## 2.2 ASCON Encryption Modes

Ascon uses a duplex sponge-based mode of operation. The encryption initializes the 320-bit state with the key and nonce. Then updates the state with the associated data, injects the plaintext and extracts the ciphertext. Finally, injects the key again and extracts the tag for authentication. After each injection, permutation is applied to the state, a stronger one with more rounds is used during initialization and finalization.

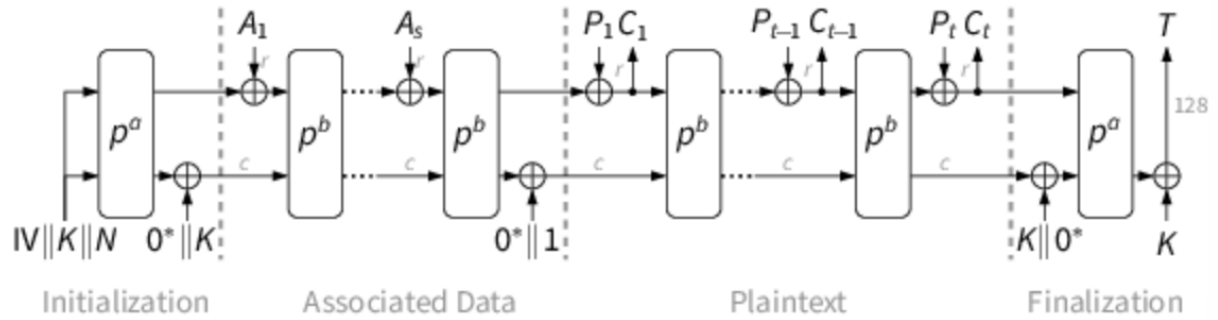


Figure 2.2: ASCON encryption algorithm illustration

## 2.3 ASCON Hashing

The hashing mode absorb the message and squeeze the hash value in 64-bit blocks. Permutation is applied in between every absorb or squeeze operation.

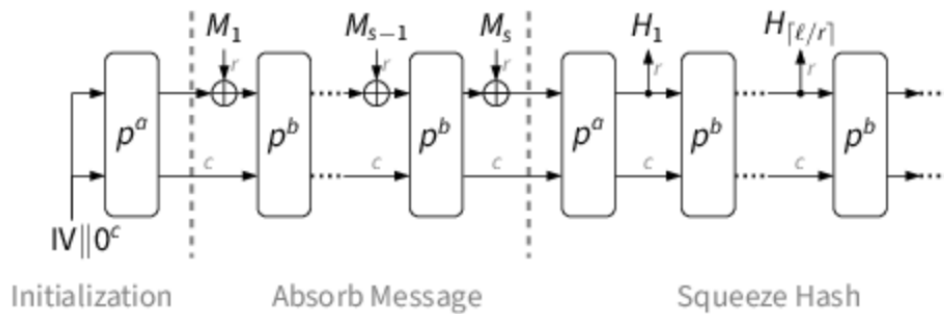


Figure 2.3: ASCON hashing algorithm illustration

### 3. RISC – V

RISC-V is an open-source instruction set architecture (ISA) that provides a simple and standardized framework for designing processors. Developed by the University of California, Berkeley, RISC-V uses the RISC (Reduced Instruction Set Computer) architecture for low complexity and energy, creating a highly optimized environment for embedded systems.

#### 3.1 RISC-V ISA (RV32I)

ISA (Instruction Set Architecture) defines a standard set of instructions that a processor can execute. It specifies how instructions are encoded, the format of data types, and the behavior of the processor when executing instructions. ISAs like RISC-V define the fundamental operations a processor can perform, influencing its performance, power efficiency, and compatibility with software applications.

RISC-V ISA supports both 32-bit (RV32I) and 64-bit (RV64I) base instruction sets, ensuring compatibility across different implementations. It has a base instruction set with six different base instruction formats, as seen on Figure 3.1. “rs1” and “rs2” are addresses of the source registers and the “rd” is the destination register address.

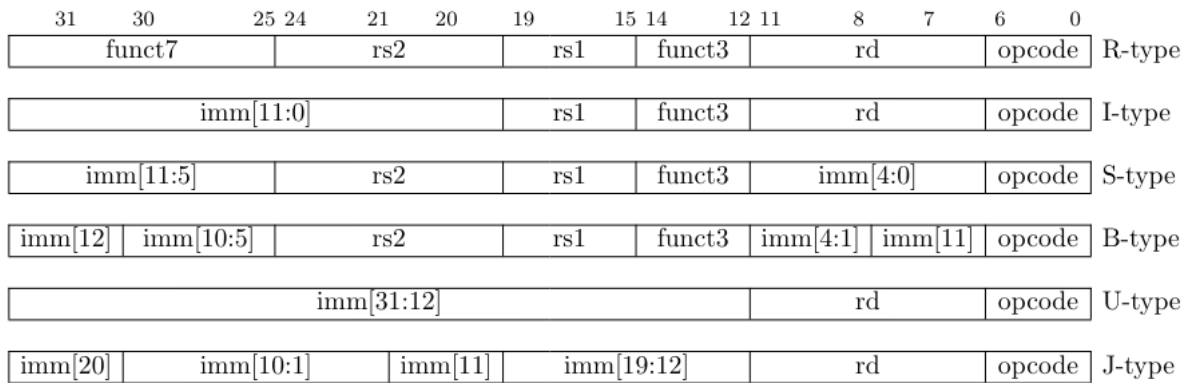


Figure 3.1: RV32 instruction formats

RV32I refers to the base instruction set of the RISC-V ISA for 32-bit processors. It includes fundamental instructions such as arithmetic operations (add, subtract), logical operations (and, or, xor), memory access (load, store), and control flow (branch, jump). RV32I forms the foundation for RISC-V processors, ensuring basic functionality and compatibility across different implementations while allowing customization through optional extensions. The base instruction set is listed on Appendix A.

RISC-V ISA has optional extensions for specific application (M for multiplication, A for atomic etc.) and can be extended with non-standard custom instructions for specific applications.

### 3.2 RISC-V Registers

RV32 variant of RISC-V has 32 x registers which have 32-bit wide. x0 is hard-wired to 0 and x1-x32 are the general-purpose registers, refer to Figure 3.2. The program counter (pc) register is an additional register that holds the address of the current instruction. Also, it has control and status registers (CSR) which provide information about the state and performance of the processor.

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller

Figure 3.2: RISC-V general purpose registers

### 3.3 CV32E40S

CV32E40S is a specific RISC-V processor core developed by the OpenHW Group. It is optimized for embedded applications, offering a balance of performance, power efficiency, and area utilization. The CV32E40S core is a 4-stage in-order 32-bit RISC-V processor that supports the RV32I and RV32E base instruction sets and can be customized with additional ISA extensions to enhance its functionality for specific application requirements.

## 4. SIMULATION SETUP

The following programs are used for the simulation:

- **riscv-gnu-toolchain:** A suite of tools including a compiler, assembler, and linker for generating .elf binaries from C and C++ source code targeting the RISC-V architecture.
- **Spike:** A RISC-V ISA (Instruction Set Architecture) simulator that executes and tests .elf binaries based on the RISC-V instruction set.
- **Proxy Kernel (pk):** A lightweight runtime environment that allows Spike to execute programs compiled with the RISC-V toolchain. It provides basic system calls and handles simple I/O operations.

First, the necessary dependencies are installed into the Linux environment (Ubuntu in this case) for it is to run these programs:

```
$ sudo apt-get install autoconf automake autotools-dev curl python3  
libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo  
gperf libtool patchutils bc zlib1g-dev libexpat-dev ninja-build  
libglib2.0-dev libpixman-1-dev
```

Now it is possible to start building the toolchain. “riscv-gnu-toolchain” repo is cloned from git and opened:

```
$ git clone https://github.com/riscv-collab/riscv-gnu-toolchain  
$ cd riscv-gnu-toolchain
```

Toolchain architecture is configured as rv32gc (extra instructions required in order to use the proxy kernel) and installed. “—version” command used for checking if the installation was successful:

```
$ ./configure --prefix=/home/yagyag12/riscv32-toolchain/rv32gc --with-  
arch=rv32gc --with-abi=ilp32  
make  
make install  
$ riscv32-unknown-elf-gcc --version
```

The created bin file is added to the path for using it outside of its directory:

```
$ export PATH=/home/yagyag12/riscv32-toolchain/rv32gc/bin:$PATH
```

GNU toolchain is successfully installed. Now ISA simulator Spike needs to get installed and built. Clone the git repository, create a directory inside called “build”, configure the file for it to work with the instruction set architecture and finally install and execute:

```
$ git clone https://github.com/riscv-software-src/riscv-isa-sim.git
cd riscv-isa-sim
mkdir build
cd build
../configure --prefix=/home/yagyag12/riscv32-toolchain/riscv-isa-sim/build --with-arch=rv32gc
make
$ sudo make install
$ ./spike
```

Again, it is added to the path:

```
$ export PATH=/home/yagyag12/riscv32-toolchain/riscv-isa-sim/build:$PATH
```

RISC-V Proxy Kernel is cloned, installed, and configured so that it uses the current toolchain as the host:

```
$ git clone https://github.com/riscv-software-src/riscv-pk.git
cd riscv-pk
mkdir build
cd build
../configure --prefix=/home/yagyag12/riscv32-toolchain/riscv-pk/build --host=riscv32-unknown-elf
make install
```

Adding to the path:

```
$ export PATH=/home/yagyag12/riscv32-toolchain/riscv-pk/build:$PATH
```

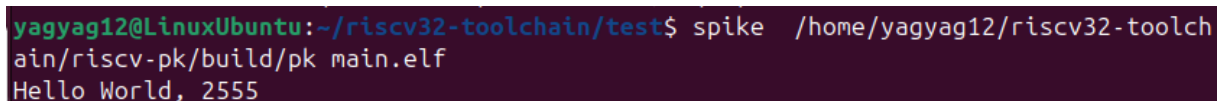
The simulation is ready to test software on ISA-level. A simple test code in C is created:

```
#include <stdint.h>
#include <stdio.h>
int main() {
    int a = 2222;
    int b = 333;
    int result = a + b;
    int *c = &result;
    printf("Hello World, %d\n", *c);
    return 0;
}
```

“riscv32-unknown-elf-gcc” command converts the .c file to .elf binary file. Spike with pk simulates the binary:

```
$ riscv32-unknown-elf-gcc -o main.elf main.c
$ spike pk main.elf
```

Output at the terminal:



```
yagyag12@LinuxUbuntu:~/riscv32-toolchain/test$ spike /home/yagyag12/riscv32-toolchain/riscv-pk/build/pk main.elf
Hello World, 2555
```

“-l” flag on spike command lists the registers and instructions. “-d” flag is for debug simulation, where each steps occur one by one and the flow is controlled by the user.

In order to have multiple variations of simulations on one run and listing and filing necessary information, a python script is created.

Below are the variables that is modified by the user:

- c\_source\_files: This array must include the main.c program alongside other .c programs inside files such as aead.c or hash.c for running the sim.
- optimization\_levels: This array includes which optimization runs will be on the simulation. -O0 flag makes no optimization, -O1 flag reduces code size and execution time for small functions, -O2 makes all supported optimizations that do not involve space-speed tradeoff, -O3 turns on all optimizations, -Os optimizes the code size.
- pk\_path: This is the directory where the pk is built, needed for the simulation command line.



Script compile and create folders for each optimization level. Then uses the installed toolchain to create binaries and assembly files for each optimization level.

After the binaries are ready, spike simulation occurs for each optimization level. Output of the program is written on the out.txt file while the log is saved inside instruction\_log.txt file. Also, the script calculates important simulation information including the total instruction count, instruction memory space and number of repetitions of each instruction executed and stores it inside stats.txt file. The full python code can be found in Appendix B.

After running a reference ASCON AEAD (explained next chapter) simulation using the python program, parts from the created files from the -O0 optimization level are shown below:

```
Original Message: Hello, World!
Associated Data: AD
Key: 31323334353637383930313233343536
Nonce: 31323334353637383930313200000000
Ciphertext: 11ec8facff00af4e27e135c42a1626a775eca646079a379e58d7f6cb3b
Decrypted Message: Hello, World!
```

Figure 4.1: AEAD simulation output

core	0:	0x00001000 (0x00000297)	auipc	t0, 0x0	
core	0:	0x00001004 (0x02028593)	addi	a1, t0, 32	
core	0:	0x00001008 (0xf1402573)	csrr	a0, mhartid	
core	0:	0x0000100c (0x0182a283)	lw	t0, 24(t0)	Total instruction count: 405366
core	0:	0x00001010 (0x00028067)	jr	t0	Instruction Memory Usage: 29188 bytes (28.5 kB, 0.03 MB)
core	0:	0x80000000 (0x1f80006f)	j	pc + 0x1f8	Instruction Opcode Frequencies (sorted by most used):
core	0:	0x8000001f8 (0x00000093)	li	ra, 0	lw: 43379
core	0:	0x8000001fc (0x000000113)	li	sp, 0	c.addi: 40730
core	0:	0x800000200 (0x000000193)	li	gp, 0	sw: 34733
core	0:	0x800000204 (0x000000213)	li	tp, 0	c.beqz: 31140
core	0:	0x800000208 (0x000000293)	li	t0, 0	c.lw: 30284
core	0:	0x80000020c (0x000000313)	li	t1, 0	bltu: 23115
core	0:	0x800000210 (0x000000393)	li	t2, 0	or: 22699
core	0:	0x800000214 (0x000000413)	li	s0, 0	c.mv: 16445
core	0:	0x800000218 (0x000000493)	li	s1, 0	
core	0:	0x80000021c (0x000000613)	li	a2, 0	
core	0:	0x800000220 (0x000000693)	li	a3, 0	

Figure 4.2: Parts from instruction\_log.txt and stats.txt

Note that this is an ISA simulation, not a processor-level, therefore hardware dependent variables such as total program space, program speed and power performance cannot be gathered. Instruction memory space only indicates how many different addresses used for instructions, not the total memory space of the programs.

## 5. ASCON IMPLEMENTATION AND SIMULATION

### 5.1 Ascon-C Programs

The official repository of Ascon has many variations for software implementation of the algorithm. Since this project is focused on 32-bit RISC-V architecture, the following programs for the Ascon128a and AsconHasha algorithms are simulated:

- opt32: Speed optimized 32-bit implementation
- opt32\_lowsize: Speed optimized 32-bit with less register use

To download the ascon-c repo:

```
$ git clone https://github.com/ascon/ascon-c
```

AEAD (Authenticated Encryption and Decryption) and hashing functions of ASCON are simulated separately. Python script logs information about encryption and decryption algorithms separately. Only the instructions occurring inside the encryption, decryption and hashing functions are calculated on stats.txt. Start and end flags are inserted to these functions.

**AEAD PROGRAM:** On main.c; key, nonce, ad and message values are set, then encryption and decryption functions are used, finally the message is controlled if it is encrypted and decrypted correctly. Also, a crypto\_aead.h header is created to use the functions on the aead.c on the main.c program.

```
int main() {
    unsigned char key[CRYPTO_KEYBYTES] = KEY;
    unsigned char nonce[CRYPTO_NPUBBYTES] = NONCE;
    unsigned char message[] = MESSAGE;
    unsigned char ad[] = ASSOCIATED_DATA;
    unsigned long long message_len = strlen(MESSAGE);
    unsigned long long ad_len = strlen(ASSOCIATED_DATA);

    unsigned char ciphertext[message_len + CRYPTO_ABYTES];
    unsigned long long ciphertext_len;

    unsigned char decrypted_message[message_len];
    unsigned long long decrypted_message_len;

    printf("Original Message: %s\n", message);
    printf("Associated Data: %s\n", ad);
    print_hex("Key", key, CRYPTO_KEYBYTES);
    print_hex("Nonce", nonce, CRYPTO_NPUBBYTES);

    // Encrypt the message
    if (crypto_aead_encrypt(ciphertext, &ciphertext_len, message, message_len, ad, ad_len, NULL, nonce, key) != 0) {
        printf("Encryption failed!\n");
        return -1;
    }

    print_hex("Ciphertext", ciphertext, ciphertext_len);

    // Decrypt the message
    if (crypto_aead_decrypt(decrypted_message, &decrypted_message_len, NULL, ciphertext, ciphertext_len, ad, ad_len, nonce, key) != 0) {
        printf("Decryption failed!\n");
        return -1;
    }

    printf("Decrypted Message: %s\n", (int)decrypted_message_len, decrypted_message);

    return 0;
}
```

Figure 5.1: main.c encryption/decryption test program

**HASH PROGRAM:** On main.c; a test input is hashed and stored. Also, crypto\_hash.h header is created for using the hash function from the hash.c program.

```
#include <stdio.h>
#include <string.h>
#include "crypto_hash.h"
#define TEST_INPUT "Hello, ASCON!"
#define HASH_OUTPUT_SIZE 32

int main() {
    // Input message to hash
    const unsigned char input[] = TEST_INPUT;
    unsigned long long input_len = strlen((const char*)input);

    // Buffer to store the hash output
    unsigned char output[HASH_OUTPUT_SIZE];

    // Call the ASCON hash function
    if (crypto_hash(output, input, input_len) != 0) {
        fprintf(stderr, "Hash function failed\n");
        return 1;
    }

    // Print the hash output
    printf("Input: %s\n", input);
    printf("Hash: ");
    for (size_t i = 0; i < HASH_OUTPUT_SIZE; ++i) {
        printf("%02x", output[i]);
    }
    printf("\n");

    return 0;
}
```

Figure 5.2: main.c hashing test program

## 5.2 Instruction Set Extension for ASCON

As explained before, ascon algorithm consists of three stages: constant addition, substitution (S-box) layer and linear diffusion layer. Constant addition is a very simple stage that the base instruction set easily handles. However, S-box function on substitution layer and rotation operations on the linear layer are complex algorithms that needs multiple instructions, meaning consumes more space and time. Creating and implementing custom instructions for s-box and rotation functions we can improve the core performance and decrease the code size.

**S-BOX:** Since s-box function uses 64-bit addresses, for 32-bit processor implementation, each word of the ascon-state (64x5 word array) is divided into even and odd 32-bit vectors. The first operand of the of the instruction, rs1, holds the address while the output operand, rd is hard-wired to zero. “sbox” for R-type and “sboxi” for I-type will be used.

**ROTATION:** The first operand, rs1, holds a 32-bit word. The second operand, rs2, is the number of rotations and the output operand, rd, is the rotated version of the word. “rot” for R-type and “roti” for I-type will be used.

### 5.3 Modification of the Toolchain and the Simulator for the Custom Instructions

In order to use custom instructions, they need to be recognizable by the compiler. Therefore, toolchain and simulator are modified and reinstalled following these steps:

Step 1: “riscv-opc.c” and “riscv-opc.h” files on GCC binutils are edited so that the assembler recognizes the custom instructions.

```
const struct riscv_opcode riscv_opcodes[] =
{
/* name, xlen, isa, operands, match, mask, match_func, pinfo. */
// ASCON
{"rot", 0, INSN_CLASS_I, "d,s,t", MATCH_ROT, MASK_ROT, match_opcode, 0},
{"roti", 0, INSN_CLASS_I, "d,s,j", MATCH_ROTI, MASK_ROTI, match_opcode, 0},
{"sbox", 0, INSN_CLASS_I, "d,s,t", MATCH_SBOX, MASK_SBOX, match_opcode, 0},
{"sboxi", 0, INSN_CLASS_I, "d,s,j", MATCH_SBOXI, MASK_SBOXI, match_opcode, 0},

```

Figure 5.3: riscv-gnu-toolchain/binutils/opcodes/riscv-opc.c modification

```
#define MATCH_ROT 0x30000033
#define MASK_ROT 0xfe00707f
#define MATCH_ROTI 0x50000033
#define MASK_ROTI 0xfe00707f
#define MATCH_SBOX 0x60000033
#define MASK_SBOX 0xfe00707f
#define MATCH_SBOXI 0x70000033
#define MASK_SBOXI 0xfe00707f

#endif /* RISCV_ENCODING_H */
#ifdef DECLARE_INSN
DECLARE_INSN(rot, MATCH_ROT, MASK_ROT)
DECLARE_INSN(roti, MATCH_ROTI, MASK_ROTI)
DECLARE_INSN(sbox, MATCH_SBOX, MASK_SBOX)
DECLARE_INSN(sboxi, MATCH_SBOXI, MASK_SBOXI)

```

Figure 5.4: riscv-gnu-toolchain/binutils/include/opcode/riscv-opc.h modification

Step 2: riscv\_rot and riscv\_sbox instruction patterns are inserted into GCC backend through the “riscv.md” file.

```
;; Custom instruction

;; ROTR32
(define_insn "riscv_rot"
  [(set(match_operand:SI 0 "register_operand" "=r,r")
    (unspec_volatile[(match_operand:SI 1 "register_operand" "r,r")
      (match_operand:SI 2 "arith_operand" "r,I")]
      UNSPECV_ROT))]
  ""
  "rot%i2\t%0,%1,%2"
)

;; SBOX
(define_insn "riscv_sbox"
  [(set(match_operand:SI 0 "register_operand" "=r,r")
    (unspec_volatile[(match_operand:SI 1 "register_operand" "r,r")
      (match_operand:SI 2 "arith_operand" "r,I")]
      UNSPECV_SBOX))]
  ""
  "sbox%i2\t%0,%1,%2"
)
```

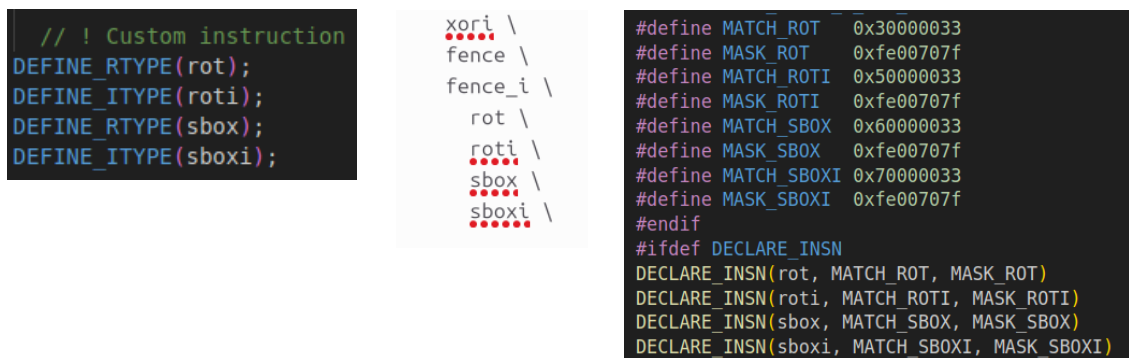
```
(define_c_enum "unspecv" [
  UNSPECV_ROT
  UNSPECV_SBOX
])
```

Figure 5.5: riscv-gnu-toolchain/gcc/gcc/config/riscv/riscv.md modifications

Step 3: GNU Toolchain is rebuilt, preferably onto a different directory. New instructions can be tested by a test code for checking if the build is correctly done.

```
$ ./configure --prefix=/home/yagyag12/isa-extension/rv32gca --with-arch=rv32gc --with-abi=ilp32
make
make install
$ riscv32-unknown-elf-gcc --version
```

Step 4: New instructions are added to “disasm.cc” and “riscv.mk.in” files on spike directory. Definitions and declarations are made on “encoding.h” file.



The figure displays three code snippets side-by-side, representing modifications to different files:

- Left snippet (disasm.cc):** Shows the definition of custom instruction types.
 

```
// ! Custom instruction
DEFINE_RTYPE(rot);
DEFINE_ITYPE(roti);
DEFINE_RTYPE(sbox);
DEFINE_ITYPE(sboxi);
```
- Middle snippet (riscv.mk.in):** Shows the addition of new instructions to the list of supported instructions.
 

```
xori \
fence \
fence_i \
rot \
roti \
sbox \
sboxi \
```
- Right snippet (encoding.h):** Shows the definition of match and mask constants, and the declaration of instruction matching functions.
 

```
#define MATCH_ROT 0x30000033
#define MASK_ROT 0xfe00707f
#define MATCH_ROTI 0x50000033
#define MASK_ROTI 0xfe00707f
#define MATCH_SBOX 0x60000033
#define MASK_SBOX 0xfe00707f
#define MATCH_SBOXI 0x70000033
#define MASK_SBOXI 0xfe00707f
#endif
#ifdef DECLARE_INSN
DECLARE_INSN(rot, MATCH_ROT, MASK_ROT)
DECLARE_INSN(roti, MATCH_ROTI, MASK_ROTI)
DECLARE_INSN(sbox, MATCH_SBOX, MASK_SBOX)
DECLARE_INSN(sboxi, MATCH_SBOXI, MASK_SBOXI)
```

Figure 5.6: /disasm/disasm.cc , /riscv/riscv.mk.in, /riscv/encoding.h modifications

Step 5: rot.h, roti.h, sbox.h and sboxi.h headers are created under “/riscv/insns/” directory for defining the functional behaviours of the custom instructions.

```
WRITE_RD((RS1 >> RS2) | (RS1 << (32 - RS2)));
```

Figure 5.7: rot.h

```
WRITE_RD((RS1 >> insn.i_imm()) | (RS1 << (32 - insn.i_imm())));
```

Figure 5.8: roti.h

```

// Load State
reg_t x0 = MMU.load<uint32_t>(RS1);
reg_t x1 = MMU.load<uint32_t>(RS1+4);
reg_t x2 = MMU.load<uint32_t>(RS1+8);
reg_t x3 = MMU.load<uint32_t>(RS1+12);
reg_t x4 = MMU.load<uint32_t>(RS1+16);

reg_t t0, t1, t2, t3, t4;

// S-Box
x0 ^= x4;
x4 ^= x3;
x2 ^= x1;
t0 = x0;
t4 = x4;
t3 = x3;
t1 = x1;
t2 = x2;
x0 = t0 ^ (~t1 & t2);
x2 = t2 ^ (~t3 & t4);
x4 = t4 ^ (~t0 & t1);
x1 = t1 ^ (~t2 & t3);
x3 = t3 ^ (~t4 & t0);
x1 ^= x0;
x3 ^= x2;
x0 ^= x4;

// Store State
MMU.store<uint32_t>(RS1, x0);
MMU.store<uint32_t>(RS1+4, x1);
MMU.store<uint32_t>(RS1+8, x2);
MMU.store<uint32_t>(RS1+12, x3);
MMU.store<uint32_t>(RS1+16, x4);
WRITE_RD(x0);

```

Figure 5.9: sbox.h and sboxi.h

Step 6: Now that the necessary modifications for the custom instructions are done, spike is reinstalled for simulation with the new custom instructions.

## 5.4 Simulation of ASCON Algorithms With and Without the Custom Instructions

For simulating the program with the custom instruction, round.h file is changed. The header used on the simulation with the custom instructions can be found on Appendix C.

<__BSS_END__+0xffff23554>		<__BSS_END__+0xffff20ed8>	
11858:	97a2	add	a5,a5,s0
1185a:	9307a783	lw	a5,-1744(a5)
1185e:	30f70733	rot	a4,a4,a5
11862:	77e5	lui	a5,0xfffff9
11864:	fd078793	addi	a5,a5,-48 # ffff8fd0
		1175e:	fd078793
		11762:	00878733
		11766:	4785
		11768:	60f70733
		1176c:	77e5
		1176e:	fd078793
		addi	a5,a5,-48
		add	a4,a5,s0
		li	a5,1
		sbox	a4,a4,a5
		lui	a5,0xfffff9
		addi	a5,a5,-48 # ffff8fd0

Figure 5.10: Custom instructions seen in the assembly code

Here are the simulation results of the algorithm variations (opt32, opt32\_lowsize) with and without the extension using the optimization flags for AEAD:

Table 5.1: opt32 AEAD simulation results

Number of Ins.	Base Instruction Set		Custom Instruction Set	
function/opt	encrypt	decrypt	encrypt	decrypt
-O0	253775	215023	242668	205701
-O1	35596	35636	24849	19751
-O2	40882	30658	24646	14399
-O3	40884	30660	17576	12447
-Os	30502	35671	19970	19992
Memory Space (kB)	Base Instruction Set		Custom Instruction Set	
function/opt	encrypt	decrypt	encrypt	decrypt
-O0	157.5	158.6	127.3	128.6
-O1	20.6	20.8	14.8	14.9
-O2	21.2	21.4	14.0	14.1
-O3	21.2	21.4	10.4	10.4
-Os	19.6	19.7	14.1	14.1

Table 5.2: opt32\_lowsize AEAD simulation results

Number of Ins.	Base Instruction Set		Custom Instruction Set	
function/opt	encrypt	decrypt	encrypt	decrypt
-O0	26658	26833	22688	22863
-O1	10461	10516	9952	10007
-O2	10431	10471	9708	9748
-O3	10431	10415	7943	7927
-Os	10310	10395	10441	10526

Memory Space (kB)	Base Instruction Set		Custom Instruction Set	
function/opt	encrypt	decrypt	encrypt	decrypt
-O0	14.9	15.3	13.9	14.2
-O1	6.2	6.3	6.0	6.1
-O2	5.4	5.5	5.1	5.2
-O3	5.4	5.7	5.1	5.4
-Os	5.0	5.2	4.9	5.0

Here are the simulation results of the algorithm variations (opt32, opt32\_lowsize) with and without the extension using optimization flags for hashing:

Table 5.3: opt32 hashing simulation results

Number of Ins.	-O0	-O1	-O2	-O3	-Os
Base Instruction Set	201843	21494	21380	21380	28489
Custom Instruction Set	196756	10223	10106	7543	10516

Memory Size (kB)	-O0	-O1	-O2	-O3	-Os
Base Instruction Set	117.1	17.9	17.5	17.5	17.5
Custom Instruction Set	99.8	11.1	10.9	7.9	10.8



Table 5.4: opt32\_lowsize hashing simulation results

Number of Ins.	-O0	-O1	-O2	-O3	-Os
Base Instruction Set	31921	11636	11752	11752	11509
Custom Instruction Set	26443	10937	10743	8324	11690

Memory Size (kB)	-O0	-O1	-O2	-O3	-Os
Base Instruction Set	9.2	4.8	4.5	4.5	4.3
Custom Instruction Set	8.2	4.6	4.3	4.2	4.2

## 6. SHA-256 IMPLEMENTATION AND SIMULATION

### 6.1 SHA-256 Algorithm in C

Pre-made C program for SHA-256 algorithm is used. This program is a very simple and easy to modify implementation of the algorithm. The repository is included on the references.

The program is simulated on RV32GC ISA using the python script.

### 6.2 SHA-256 Extensions

Unlike ASCON, SHA-2 has already had official instruction set extension called NIST Suite: Hash Function Instructions (Zknh).

Table 6.1: Zknh Instruction Set Extension

RV32	RV64	Mnemonic	Instruction
✓	✓	sha256sig0	SHA2-256 Sigma0 instruction
✓	✓	sha256sig1	SHA2-256 Sigma1 instruction
✓	✓	sha256sum0	SHA2-256 Sum0 instruction
✓	✓	sha256sum1	SHA2-256 Sum1 instruction
✓		sha512sig0h	SHA2-512 Sigma0 high (RV32)
✓		sha512sig0l	SHA2-512 Sigma0 low (RV32)
✓		sha512sig1h	SHA2-512 Sigma1 high (RV32)
✓		sha512sig1l	SHA2-512 Sigma1 low (RV32)
✓		sha512sum0r	SHA2-512 Sum0 (RV32)
✓		sha512sum1r	SHA2-512 Sum1 (RV32)
	✓	sha512sig0	SHA2-512 Sigma0 instruction (RV64)
	✓	sha512sig1	SHA2-512 Sigma1 instruction (RV64)
	✓	sha512sum0	SHA2-512 Sum0 instruction (RV64)
	✓	sha512sum1	SHA2-512 Sum1 instruction (RV64)

Spike and toolchain are configured and reinstalled including the zknh extension:

```
$ ./configure --prefix=/home/yagyag12/riscv32-toolchain/rv32gc_zknh --
with-arch=rv32gc_zknh --with-abi=ilp32
make
make install
$ riscv32-unknown-elf-gcc --version
```

```
$ mkdir build-sha
cd build-sha
../configure --prefix=/home/yagyag12/riscv32-toolchain/riscv-isa-
sim/build-sha --with-arch=rv32gc_zknh
make
$ sudo make install
$ ./spike
```

Python script for the SHA-256 ran using the new spike simulator and toolchain. Here are the simulation results with and without the instruction set extension:

Table 6.2: SHA-256 hashing simulation results

Number of Ins.	-O0	-O1	-O2	-O3	-Os
Base Instruction Set	16524	9780	9965	9413	9839
Custom Instruction Set	14396	8877	8804	8718	15554

Memory Size (kB)	-O0	-O1	-O2	-O3	-Os
Base Instruction Set	5.5	4.0	4.3	5.8	4.2
Custom Instruction Set	5.3	3.9	4.1	5.0	5.1

## **7. RESULT ANALYSIS AND CONCLUSION**

### **7.1 Effects of the Extensions**

Simulation results indicate that the implementation of custom instructions has a positive effect on program performance. The permutation rounds in ASCON and algorithm functions of SHA-256 require less instructions due to the instruction set extensions. These extensions reduce the number of instructions and the memory space allocated for instructions, allowing a more compact design and faster performance.

The rotation function (rot) in ASCON is relatively simple, involving only a few logical operations, while the SHA-256 sigma and sum functions are more complex, incorporating several logical operations. Despite its simplicity, rot is used extensively in ASCON permutations, making it beneficial. The most complex instruction extension is the non-linear S-box layer (sbox) in ASCON, which combines multiple logical operations. The high complexity of sbox significantly reduces the number of instructions in the program.

Overall, it is observed that custom instructions added to the ASCON (rot, sbox) have a bigger impact on performance compared to the Zknh extension on SHA-256.

### **7.2 Effects of the Optimizations**

When comparing other optimization flags to no optimization -O0, a drastic drop in number of instructions and memory space can be observed. This shows that the spike optimizations have a crucial impact on the code size and performance.

It is seen that -O2 and -O3 flags usually generate the same or very similar outcomes. This can be explained by that -O2 already reaches the maximum optimized value that the additions from -O3 flag either is not applicable in that specific program or have little effect.

From all the simulation results, it is evident that the -O3 optimization flag gives the optimal result most of the time, especially when combined with the custom instructions.

Note that on SHA-256 simulation with the -Os flag, an unexpected outcome occurs where the custom instructions increase the instruction size rather than decreasing it. This is probably because custom instructions are directly implemented by the user, same number in every optimization and does not depend on the toolchain or the simulator, restricts the compiler's ability to optimize the code so much that the free optimization without the added instructions results better.

### **7.3 Comparison Between ASCON Hash and SHA-256**

SHA-256 hashing algorithm has a smaller code size in C and requires less instructions than the ASCON Hash function, while the memory space is similar and optimization-dependent. For hashing only, SHA-256 is the optimal choice between the two considering the efficiency, simplicity, and ISA-level performance. It is also easier to implement and modify than ASCON hashing.

However, it is important to note that SHA-256 is only a hashing algorithm, whereas ASCON can perform encryption-decryption, authentication, and hashing. Therefore, ASCON shines in scenarios where comprehensive security and cryptographic functionality are required.

## **7.4 Conclusion**

The implementation of custom instructions significantly enhances the performance of cryptographic algorithms by reducing the number of instructions and memory usage. Optimization flags, particularly -O3, further improve performance and code size. SHA-256 is generally more efficient for hashing, while ASCON provides a broader range of cryptographic functions, making it suitable for more complex security needs.

## 8. FURTHER PRACTICE: HARDWARE IMPLEMENTATION

Note that the hardware implementation and simulation is not the topic of this project, so there will be no simulations on this chapter. Only a possible implementation of the custom instructions created onto the CV32E40S processor on RTL level using SystemVerilog.

There are multiple ways to implement ASCON and SHA-256 algorithms. There is a possibility to integrate ASCON and SHA-2 modules to the processor as peripherals. RTL modules for these algorithms can be found on GitHub. In that case, after the instruction is decoded, rs1 and rs2 values are transferred to the needed module using a bus with either UART or AXI communication. This option is more robust but not suitable for high-speed applications.

Rather than creating modules for each algorithm, another approach is adding the custom instructions directly to the ALU. New instructions' circuits and connections are inside the alu, just like the other operations, resulting in much faster and compact design.

RTL source files of CV32E40S are downloaded and built on Xilinx Vivado.

“cv32e40s\_pkg.sv” file contains the variables of the core. Custom instructions are appended to the ALU operations.

```
// Custom Extensions
ALU_ROT      = 6'b111110,
ALU_SBOX     = 6'b101101,
ALU_SHA256SUM0 = 6'b110110,
ALU_SHA256SUM1 = 6'b111010,
ALU_SHA256SIG0 = 6'b111011,
ALU_SHA256SIG1 = 6'b111111

} alu_opcode_e;
```

Figure 8.1: alu\_opcode\_e struct modification

Afterwards, custom instructions are included on result mux and their behaviors are created inside the “cv32e40s\_alu.sv” module.

```
ALU_ROT      : result_o = rot_result;
ALU_SBOX     : result_o = sbox_result;
ALU_SHA256SUM0 : result_o = sum0_result;
ALU_SHA256SUM1 : result_o = sum1_result;
ALU_SHA256SIG0 : result_o = sig0_result;
ALU_SHA256SIG1 : result_o = sig1_result;
default: ;
```

Figure 8.2: ALU ResultMux modification

Custom instructions are integrated to the ALU by simple assign operations, except for the sbx where a new module is created inside the ALU for realizing this operation.

```
////////// CUSTOM INSTRUCTIONS //////////

logic [31:0] rot_result;
logic [31:0] sbx_result;
logic [31:0] sum0_result;
logic [31:0] sum1_result;
logic [31:0] sig0_result;
logic [31:0] sig1_result;

assign rot_result = (operand_a_i >> operand_b_i) | (operand_a_i << (32 - operand_b_i));

sbox sbox_alu (
    .operand_a_i(operand_a_i),
    .result(sbx_result)
);

assign sum0_result = ((operand_a_i >> 2) | (operand_a_i << 20)) ^
    ((operand_a_i >> 13) | (operand_a_i << 19)) ^ ((operand_a_i >> 22) | (operand_a_i << 10));

assign sum1_result = ((operand_a_i >> 6) | (operand_a_i << 26)) ^
    ((operand_a_i >> 11) | (operand_a_i << 21)) ^ ((operand_a_i >> 25) | (operand_a_i << 7));

assign sig0_result = ((operand_a_i >> 7) | (operand_a_i << 25)) ^
    ((operand_a_i >> 18) | (operand_a_i << 14)) ^ (operand_a_i >> 3);

assign sig1_result = ((operand_a_i >> 17) | (operand_a_i << 15)) ^
    ((operand_a_i >> 19) | (operand_a_i << 13)) ^ (operand_a_i >> 10);
```

Figure 8.3: Custom instruction addition to the ALU

Sbox operation needs five 32-bit inputs but can only get it one at a time. So, an FSM is crafted where the module loads five words, executes the operation and transmits the result in five cycles, 11 states and 11 cycles per operation. ALU itself is a combinational circuit, meaning it is not sequential equipped with a clock or a reset. In order to control the sbx module, clock and reset signals of the decode-execute stage must be connected.

Sbox module code and modified ALU schematic can be found on Appendices D and E.

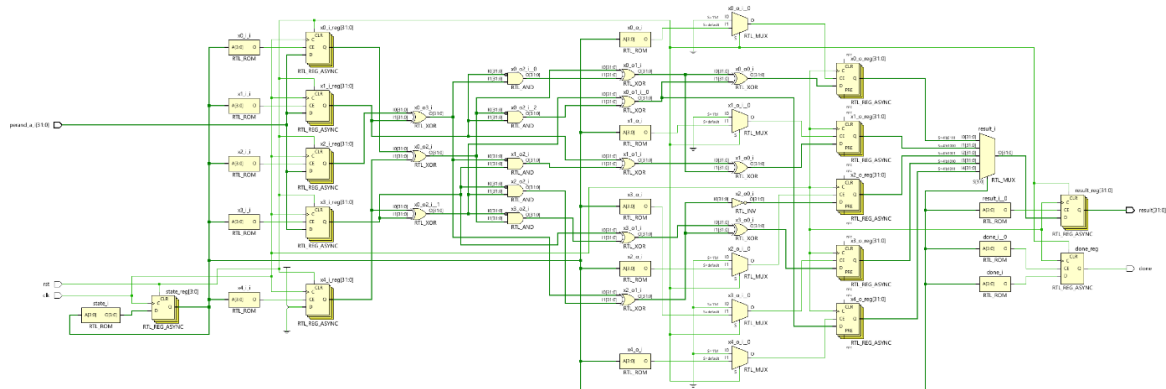


Figure 8.4: Sbox module schematic

## REFERENCES

1. Altınay, Ö., & Örs, B. (2021). Instruction extension of RV32I and GCC backend for Ascon lightweight cryptography algorithm.
2. ASCON C Implementation. (n.d.). Retrieved from <https://github.com/ascon/ascon-c>
3. B-Con. (n.d.). SHA256 C code. Retrieved from <https://github.com/B-Con/crypto-algorithms>
4. Dobraunig, C., Eichlseder, M., Mendel, F., & Schlaffer, M. (2019). Ascon v1.2: Submission to the NIST lightweight cryptography competition.
5. Eryilmaz, Y. (2022, January). Extending the instruction set of RISC-V processor for Ascon algorithm.
6. Jellema, L. (2019, July 10). Optimizing Ascon on RISC-V.
7. Joung, D. (2023, August). Design and simulate RISC-V processor using Verilog.
8. Maneesha, B., et al. (2024, April). Design of RISC-V processor using Verilog. International Journal of Research Publication and Reviews, 5(4).
9. OpenHW Group CV32E40S RISC-V IP. (n.d.). Retrieved from <https://github.com/openhwgroup/cv32e40s>
10. RISC-V GNU Toolchain. (n.d.). Retrieved from <https://github.com/riscv/riscv-gnu-toolchain>
11. RISC-V International. (2023). RISC-V cryptography extensions volume I: Scalar & entropy source instructions.
12. RISC-V Proxy Kernel. (n.d.). Retrieved from <https://github.com/riscv/riscv-pk>
13. Spike RISC-V ISA Simulator. (n.d.). Retrieved from <https://github.com/riscv/riscv-isa-sim>
14. Waterman, A., & Asanovic, K. (2017, May 7). The RISC-V instruction set manual, Volume I: User-level ISA.



# APPENDICES

## APPENDIX A: RV32I Base Instruction Set

RV32I Base Instruction Set							
imm[31:12]				rd	0110111	LUI	
imm[31:12]				rd	0010111	AUIPC	
imm[20 10:1 11 19:12]				rd	1101111	JAL	
imm[11:0]		rs1	000	rd	1100111	JALR	
imm[12 10:5]	rs2	rs1	000	imm[4:1 11]	1100011	BEQ	
imm[12 10:5]	rs2	rs1	001	imm[4:1 11]	1100011	BNE	
imm[12 10:5]	rs2	rs1	100	imm[4:1 11]	1100011	BLT	
imm[12 10:5]	rs2	rs1	101	imm[4:1 11]	1100011	BGE	
imm[12 10:5]	rs2	rs1	110	imm[4:1 11]	1100011	BLTU	
imm[12 10:5]	rs2	rs1	111	imm[4:1 11]	1100011	BGEU	
imm[11:0]		rs1	000	rd	0000011	LB	
imm[11:0]		rs1	001	rd	0000011	LH	
imm[11:0]		rs1	010	rd	0000011	LW	
imm[11:0]		rs1	100	rd	0000011	LBU	
imm[11:0]		rs1	101	rd	0000011	LHU	
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB	
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH	
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW	
imm[11:0]		rs1	000	rd	0010011	ADDI	
imm[11:0]		rs1	010	rd	0010011	SLTI	
imm[11:0]		rs1	011	rd	0010011	SLTIU	
imm[11:0]		rs1	100	rd	0010011	XORI	
imm[11:0]		rs1	110	rd	0010011	ORI	
imm[11:0]		rs1	111	rd	0010011	ANDI	
0000000	shamt	rs1	001	rd	0010011	SLLI	
0000000	shamt	rs1	101	rd	0010011	SRLI	
0100000	shamt	rs1	101	rd	0010011	SRAI	
0000000	rs2	rs1	000	rd	0110011	ADD	
0100000	rs2	rs1	000	rd	0110011	SUB	
0000000	rs2	rs1	001	rd	0110011	SLL	
0000000	rs2	rs1	010	rd	0110011	SLT	
0000000	rs2	rs1	011	rd	0110011	SLTU	
0000000	rs2	rs1	100	rd	0110011	XOR	
0000000	rs2	rs1	101	rd	0110011	SRL	
0100000	rs2	rs1	101	rd	0110011	SRA	
0000000	rs2	rs1	110	rd	0110011	OR	
0000000	rs2	rs1	111	rd	0110011	AND	
0000	pred	succ	00000	000	00000	0001111	FENCE
0000	0000	0000	00000	001	00000	0001111	FENCE.I
0000000000000			00000	000	00000	1110011	ECALL
0000000000001			00000	000	00000	1110011	EBREAK
csr			rs1	001	rd	1110011	CSR <sub>RW</sub>
csr			rs1	010	rd	1110011	CSR <sub>RS</sub>
csr			rs1	011	rd	1110011	CSR <sub>RC</sub>
csr			zimm	101	rd	1110011	CSR <sub>RWI</sub>
csr			zimm	110	rd	1110011	CSR <sub>RSI</sub>
csr			zimm	111	rd	1110011	CSR <sub>RCI</sub>

## APPENDIX B: run\_sim.py python file for SHA256 Hash simulation

```
import subprocess
import sys
from collections import defaultdict
import os

# Define the C source files and the output binaries
c_source_files = ["main.c", "hash.c", "permutations.c"]
optimization_levels = ["00", "01", "02", "03", "0s"]
output_folders = ["sim_00", "sim_01", "sim_02", "sim_03", "sim_0s"]
output_binaries = [f"{folder}/my_program" for folder in output_folders]

# Path to the proxy kernel (pk)
pk_path = "/home/yagyag12/riscv32-toolchain/riscv-pk/build/pk"

# Compile and create folders for each optimization level
for folder in output_folders:
    os.makedirs(folder, exist_ok=True)

for idx, opt_level in enumerate(optimization_levels):
    compile_command = ["riscv32-unknown-elf-gcc", f"-{opt_level}", "-o",
output_binaries[idx]] + c_source_files
    print(f"Compiling {c_source_files} with optimization {opt_level}...")

    try:
        subprocess.run(compile_command, check=True)
        print(f"Compilation with {opt_level} optimization successful.")
    except subprocess.CalledProcessError as e:
        print(f"Compilation failed with error: {e}", file=sys.stderr)
        sys.exit(1)

# Run the Spike simulator and capture statistics
spike_command_with_log = ["spike", "-l", pk_path, output_binaries[idx]]

print(f"Running the Spike simulator with {opt_level} optimization...")
```

```

    process = subprocess.Popen(spike_command_with_log, stdout=subprocess.PIPE,
stderr=subprocess.STDOUT, text=True)

    instruction_count = 0
    instruction_frequency = defaultdict(int)
    unique_instruction_addresses = set() # To track unique instruction memory addresses

    # Variables to track if we are inside the hash function
    inside_function = False

    # Open the instruction log file and output file
    with open(f"{output_folders[idx]}/instruction_log.txt", "w") as log_file,
open(f"{output_folders[idx]}/out.txt", "w") as out_file:
        for output in process.stdout:
            instruction = output.strip()
            if instruction.startswith("core"):
                log_file.write(instruction + "\n") # Always write to instruction log file

            if "FUNCTION_START" in instruction:
                inside_function = True
                continue

            if "FUNCTION_END" in instruction:
                inside_function = False
                continue

            if inside_function:
                # Handle instruction counting
                instruction_count += 1 # Increment instruction count for each line
                # Extract instruction memory address and opcode
                parts = instruction.split()
                if len(parts) >= 5:
                    address = parts[2] # Address is the third element in the line
                    opcode = parts[4] # Opcode is the fifth element in the line
                    unique_instruction_addresses.add(address)
                    instruction_frequency[opcode] += 1

            if (instruction.startswith("core") == 0):
                # Handle program output
                out_file.write(instruction + "\n")

```

```

# Calculate instruction memory usage in bytes and kilobytes
instruction_memory_bytes = len(unique_instruction_addresses) * 4 # Assuming 4 bytes
per instruction

instruction_memory_kb = instruction_memory_bytes / 1024
instruction_memory_mb = instruction_memory_kb / 1024

print(f"Spike simulation completed with {opt_level} optimization level.")

# Write the total instruction count, instruction frequencies, memory usage, and
estimated cycles to stats.txt
with open(f"{output_folders[idx]}/stats.txt", "w") as stats_file:
    stats_file.write(f"Total instruction count: {instruction_count}\n")
    stats_file.write(f"Instruction Memory Usage: {instruction_memory_bytes} bytes ")
    stats_file.write(f"({instruction_memory_kb:.1f} kB, {instruction_memory_mb:.2f}
MB)\n")
    stats_file.write(f"Instruction Opcode Frequencies (sorted by most used):\n")
    sorted_instructions = sorted(instruction_frequency.items(), key=lambda item:
item[1], reverse=True)
    for opcode, freq in sorted_instructions:
        stats_file.write(f"{opcode}: {freq}\n")
    print(f"Total instruction count: {instruction_count}. Stats have been logged to
stats.txt.")
    print(f"Program output has been written to {output_folders[idx]}/out.txt.")

```

## APPENDIX C: Modified round.h header

```
#ifndef ROUND_H_
#define ROUND_H_

#include "ascon.h"
#include "constants.h"
#include "forceinline.h"
#include "printstate.h"
#include "word.h"

forceinline uint32_t rot(unsigned int x, unsigned int n) {
    uint32_t result0;
    asm volatile("rot %[result0], %[value1], %[value2]\n\t":
        [result0] "=r" (result0) : [value1] "r" (x), [value2] "r" (n)
    );
    return result0;
}

forceinline void ROUND(ascon_state_t* s, uint64_t C) {
    /* round constant */
    s->x[2] ^= C;

    uint32_t even[5];
    uint32_t odd[5];
    for (int i = 0; i < 5; ++i) {
        even[i] = (uint32_t)(s->x[i] >> 32); // Extract even bits (higher 32 bits)
        odd[i] = (uint32_t)(s->x[i] & 0xFFFFFFFF); // Extract odd bits (lower 32 bits)
    }

    uint32_t t0_e, t1_e, t2_e, t3_e, t4_e; // temp variables (even)
    uint32_t t0_o, t1_o, t2_o, t3_o, t4_o; // temp variables (odd)

    uint32_t r0;
    asm volatile("sbox %[result0], %[value1], %[value2]\n\t":
        [result0] "=r" (r0) : [value1] "r" (&even[0]), [value2] "r" (1)
    );

    asm volatile("sbox %[result0], %[value1], %[value2]\n\t":
        [result0] "=r" (r0) : [value1] "r" (&odd[0]), [value2] "r" (1)
    );

    // linear layer
    t0_e = even[0] ^ rot(odd[0], 4);    t0_o = odd[0] ^ rot(even[0], 5);
    t1_e = even[1] ^ rot(even[1], 11); t1_o = odd[1] ^ rot(odd[1], 11);
    t2_e = even[2] ^ rot(odd[2], 2);    t2_o = odd[2] ^ rot(even[2], 3);
    t3_e = even[3] ^ rot(odd[3], 3);    t3_o = odd[3] ^ rot(even[3], 4);
    t4_e = even[4] ^ rot(even[4], 17);  t4_o = odd[4] ^ rot(odd[4], 17);

    even[0] ^= rot(t0_o, 9);            odd[0] ^= rot(t0_e, 10);
    even[1] ^= rot(t1_o, 19);           odd[1] ^= rot(t1_e, 20);
    even[2] ^= t2_o;                    odd[2] ^= rot(t2_e, 1);
    even[3] ^= rot(t3_e, 5);            odd[3] ^= rot(t3_o, 5);
    even[4] ^= rot(t4_o, 3);            odd[4] ^= rot(t4_e, 4);

    even[2] = ~even[2];                 odd[2] = ~odd[2];

    printstate(" round output", s);
}

forceinline void PROUNDS(ascon_state_t* s, int nr) {
    int i = START(nr);
    do {
        ROUND(s, RC(i));
        i += INC;
    } while (i != END);
}

#endif /* ROUND_H_ */
```

## APPENDIX D: sbbox.sv Module

```
1 | timescale 1ns / 1ps
2 | module sbbox(
3 |     input logic clk, rst,
4 |     input logic [31:0] operand_a_i,
5 |     output logic [31:0] result,
6 |     output logic done
7 | );
8 |     logic [31:0] x0_aff1, x0_chi, x0_aff2;
9 |     logic [31:0] x1_aff1, x1_chi, x1_aff2;
10 |    logic [31:0] x2_aff1, x2_chi, x2_aff2;
11 |    logic [31:0] x3_aff1, x3_chi, x3_aff2;
12 |    logic [31:0] x4_aff1, x4_chi, x4_aff2;
13 |    logic [31:0] x0_i, x1_i, x2_i, x3_i, x4_i;
14 |    logic [31:0] x0_o, x1_o, x2_o, x3_o, x4_o;
15 |
16 |    typedef enum logic [3:0] {
17 |        IDLE,
18 |        LOAD_X0,
19 |        LOAD_X1,
20 |        LOAD_X2,
21 |        LOAD_X3,
22 |        LOAD_X4,
23 |        SBOX,
24 |        STORE_X0,
25 |        STORE_X1,
26 |        STORE_X2,
27 |        STORE_X3,
28 |        STORE_X4
29 |    } state_t;
30 |    state_t state;
31 |
32 |    always_ff @(posedge clk or posedge rst) begin
33 |        if (rst) begin
34 |            state <= IDLE;
35 |        end else begin
36 |            case (state)
37 |                IDLE: state <= LOAD_X0;
38 |                LOAD_X0: state <= LOAD_X1;
39 |                LOAD_X1: state <= LOAD_X2;
40 |                LOAD_X2: state <= LOAD_X3;
```

```

41         LOAD_X3: state <= LOAD_X4;
42         LOAD_X4: state <= SBOX;
43         SBOX: state <= STORE_X0;
44         STORE_X0: state <= STORE_X1;
45         STORE_X1: state <= STORE_X2;
46         STORE_X2: state <= STORE_X3;
47         STORE_X3: state <= STORE_X4;
48         STORE_X4: state <= IDLE;
49         default: state <= IDLE;
50     endcase
51 end
52 end
53
54 always_ff @(posedge clk or posedge rst) begin
55     if (rst) begin
56         x0_i <= 32'b0;
57         x1_i <= 32'b0;
58         x2_i <= 32'b0;
59         x3_i <= 32'b0;
60         x4_i <= 32'b0;
61         result <= 32'b0;
62         done <= 32'b0;
63     end else begin
64         case (state)
65             IDLE: done <= 1'b0;
66             LOAD_X0: x0_i <= operand_a_i;
67             LOAD_X1: x1_i <= operand_a_i;
68             LOAD_X2: x2_i <= operand_a_i;
69             LOAD_X3: x3_i <= operand_a_i;
70             LOAD_X4: x4_i <= SBOX;
71             SBOX: begin
72                 x0_aff1 = x0_i ^ x4_i;
73                 x1_aff1 = x1_i;
74                 x2_aff1 = x2_i ^ x1_i;
75                 x3_aff1 = x3_i;
76                 x4_aff1 = x4_i ^ x3_i;
77
78                 x0_chi = x0_aff1 ^ ((~x1_aff1) & x2_aff1);
79                 x1_chi = x1_aff1 ^ ((~x2_aff1) & x3_aff1);
80                 x2_chi = x2_aff1 ^ ((~x3_aff1) & x4_aff1);
81                 x3_chi = x3_aff1 ^ ((~x4_aff1) & x0_aff1);
82                 x4_chi = x4_aff1 ^ ((~x0_aff1) & x1_aff1);
83
84                 x0_o = x0_chi ^ x4_chi;
85                 x1_o = x1_chi ^ x0_chi;
86                 x2_o = ~x2_chi;
87                 x3_o = x3_chi ^ x2_chi;
88                 x4_o = x4_chi;
89             end
90             STORE_X0: result <= x0_o;
91             STORE_X1: result <= x1_o;
92             STORE_X2: result <= x2_o;
93             STORE_X3: result <= x3_o;
94             STORE_X4: begin result <= x4_o; done <= 1'b1; end
95         endcase
96     end
97 end
98 endmodule

```

## APPENDIX E: Modified ALU RTL Schematic

