ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL

INSTRUCTION EXTENSION OF RV32I AND GCC BACK END FOR ASCON LIGHTWEIGHT CRYPTOGRAPHY ALGORITHM

M.Sc. THESIS

Özlem ALTINAY

Electronics and Communication Engineering Department

Electronics Engineering Programme

JULY 2021

ISTANBUL TECHNICAL UNIVERSITY ★ GRADUATE SCHOOL

INSTRUCTION EXTENSION OF RV32I AND GCC BACK END FOR ASCON LIGHTWEIGHT CRYPTOGRAPHY ALGORITHM

M.Sc. THESIS

Özlem ALTINAY

(504181224)

Electronics and Communication Engineering Department

Electronics Engineering Programme

Thesis Advisor: Prof. Dr. Sıddıka Berna Örs Yalçın

JULY 2021

İSTANBUL TEKNİK ÜNİVERSİTESİ ★ LİSANSÜSTÜ EĞİTİM ENSTİTÜSÜ

RISC-V KOMUT SETİ MİMARSİ VE GNU DERLEYİCİ KOLEKSİYONUNUN ASCON ŞİFRELEME ALGORİTMASI İÇİN GENİŞLETİLMESİ

YÜKSEK LİSANS TEZİ

Özlem ALTINAY

(504181224)

Elektronik ve Haberleşme Mühendisliği Anabilim Dalı

Elektronik Mühendisliği Programı

Tez Danışmanı: Prof. Dr. Sıddıka Berna Örs Yalçın

TEMMUZ 2021

Özlem ALTINAY, a M.Sc student of İTU Graduate School student ID 504181224 successfully defended the thesis/dissertation entitled "INSTRUCTION EXTENSION OF RV32I AND GCC BACK END FOR ASCON LIGHTWEIGHT CRYPTOGRAPHY ALGORITHM", which she prepared after fulfilling the requirements specified in the associated legislations, before the jury whose signatures are below.

Thesis Advisor : **Prof. Dr. Sıddıka Berna Örs YALÇIN**
İstanbul Technical University

Jury Members : **Assoc. Dr. Şerif BAHTİYAR**
İstanbul Technical University

Prof. Dr. Nizamettin AYDIN
Yıldız Technical University

Date of Submission : 29 June 2021

Date of Defense : 07 July 2021

FOREWORD

I would like to thank my project advisor, Prof. Dr. Sıddıka Berna Örs Yalçın for guiding me and sharing her experiences with me to accomplish this thesis.

I also want to thank my family and friends for their endless support to all my works.

July 2021 Özlem ALTINAY

(Electronics and Communication Engineer)

TABLE OF CONTENTS

Page

ABBREVIATIONS

ABI : Application Binary Interface
AEAD : Authenticated Encryption with Associated Data
AMD : Advanced Micro Devices
ARM : Advanced RISC Machines
ASIP : Application Specific Instruction Set Processor
CSR : Control and Status Register
ELF : Executable and Linkable File
FLEN : Width of a floating register in bits
FPGA : Field Programmable Gate Array
GCC : GNU Compiler Collection
GDB : GNU Debugger
GIMPLE : GNU Simple Representation
IoT : Internet of Things
IR : Intermediate Representation
ISA : Instruction Set Architecture
LWC : Lightweight Cryptography
MIPS : Microprocessor without Interlocked Pipeline Stages
NIST : National Institute of Standards and Technology
NTRU : Nth Degree Truncated Polynomial Ring Unit
OS : Operating System
PC : Personal Computer
Pk : Proxy Kernel
PULP : Parallel Ultra-Low-Power Processor
RAM : Random Access Memory
RFID : Radio Frequency Identification
RISC : Reduced Instruction Set Computer
RTL : Register Transfer Language
XLEN : Width of an integer register in bits

SYMBOLS

$\boldsymbol{A}$	: Associated data
$\boldsymbol{C}$	: Cipher text
$\boldsymbol{D}$	: Decryption function
$\boldsymbol{\mathcal{E}}$	: Encryption function
$\boldsymbol{f}$	: Bit-interleaving function
$\boldsymbol{H}$	: Hash
$\boldsymbol{IV}$	: Initialization vector
$\boldsymbol{K}$	: Secret key
$\boldsymbol{N}$	: Nonce
$\boldsymbol{M}$	: Message
$\boldsymbol{P}$	: Plain text
$\boldsymbol{p^a}$	: Permutation with a rounds
$\boldsymbol{r}$	: Rate
$\boldsymbol{T}$	: Authentication tag
$\boldsymbol{S}$	: The 320-bit state
$\boldsymbol{\mathcal{X}}$	: Hash function
$\parallel$	: Concatenation of bit strings
$\oplus$	: Bitwise XOR
$\odot$	: Bitwise AND
>>>	: Right-rotation circular shift
$\perp$	: Error
$	: Terminal command

LIST OF TABLES

Page

LIST OF FIGURES

Page

Page

INSTRUCTION EXTENSION OF RV32I AND GCC BACK END FOR ASCON LIGHTWEIGHT CRYPTOGRAPHY ALGORITHM

SUMMARY

Nowadays, the spread of internet of things applications leads more connected devices. The information shared between these devices might be secret and exposed to cyber-attacks. These embedded devices also have limited resources in terms of memory, power, area, etc. Lightweight cryptography is useful to provide security and privacy in resource constrained embedded devices.

Latency and memory consumption are the key elements in performance metrics for lightweight cryptography algorithm implementations. ASCON lightweight cryptography algorithm is one of the finalists in CEASAR competition that is organized by National Institute of Standards and Technology. ASCON provides authentication with associated data and hashing functionalities by using basic logical operations such as AND and XOR. It uses repeated rounds to achieve a certain security level and operates on 320-bit state. It operates with S-Boxes in substitution layer and right shift circular rotation in linear layer. ASCON also provides implementations for the 32-bit and 64-bit processors.

RISC-V is an open source instruction set architecture developed by University of California, Berkeley. RISC-V is easy to extend with custom instructions for specific applications such as cryptography, 5G, image processing, etc. In this study, special cryptographic non-standard RISC-V instructions are developed to execute some operations in the algorithm with these instructions. The aim of this procedure is to reduce the required number of clock cycles and instruction memory for the execution of the ASCON on RV32I based processors.

A profiling methodology is developed to choose the best special instruction for achieving the highest benefit in performance. Software implementation of ASCON is analyzed and profiled. An end-to-end test environment is formed by extending the GNU Compiler Collection and Spike RISC-V ISA Simulator for the special cryptographic instruction extensions of RISC-V processors. GCC instruction patterns and built-in functions are investigated with examples in GCC. Usage and purpose of the built-in functions and instruction patterns are explained with hands-on examples. New built-in functions and instruction patterns for the new instructions have been integrated into the GCC RISC-V back end.

Spike has been modified with the new instructions to run the program. The algorithm has been analyzed with the proposed instructions and different optimization flags. The performance improvements in terms of instruction memory and total number of clock cycles of the program are shown in the thesis.

RISC-V KOMUT SETİ MİMARSİ VE GNU DERLEYİCİ KOLEKSİYONUNUN ASCON ŞİFRELEME ALGORİTMASI İÇİN GENİŞLETİLMESİ

ÖZET

Günümüzde nesnelerin interneti uygulamalarının yaygınlaşması ile birbiri ile bağlantılı cihaz sayısı artmaktadır. Bu cihazlar arasında paylaşılan bilgiler gizli olabilir ve bu sebeple siber saldırılara açık olabilir. Bu gömülü cihazlar bellek, güç, alan vb. açısından sınırlı kaynaklara sahip olduğundan dolayı, üzerlerinde çalıştırılacak olan programların bu kaynakları etkin bir biçimde kullanması son derece kritiktir.

Hafif şifreleme algoritmaları, kaynak kısıtlaması olan bu gömülü cihazlarda güvenlik ve gizlilik sağlamak için kullanılır. Ulusal Standartlar ve Teknoloji Enstitüsü tarafından yazılan rapordaki kurallara göre bu algoritmaların nasıl tasarlanması gerektiği açıklanmıştır. Şifreleme anahtarının en az 112-bit olması gerekir. Hafif şifreleme algoritmalarının yüksek güvenlikli şifreleme yapabilmesi için sık tekrar eden turları mevcut olmalıdır. Ayrıca, hafif şifreleme algoritmalarının yazılım ve donanım uygulamalarının da kolayca gerçeklenebilir olması gerekmektedir. Algoritmanın çalışmasındaki gecikme ve bellek tüketimi, hafif şifreleme algoritmaları uygulamaları için performans ölçütlerindeki temel unsurlardandır.

ASCON hafif şifreleme algoritması, Ulusal Standartlar ve Teknoloji Enstitüsü tarafından düzenlenen CEASAR yarışmasındaki finalistlerden biridir. ASCON, yardımcı veriler ile doğrulama ve şifreleme çözümü sunar. ASCON'un, 16-bit, 32-bit ve 64-bit işlemciler için gerçeklenmiş yazılım uygulamaları da mevcuttur. 64-bit olmayan işlemciler için bit bölme metodu kullanarak şifreleme ve şifre çözme işlemlerini gerçekleştirir. Farklı uygulama alanlarında farklı güvenlik sunmak adına, çeşitli varyasyonları mevcuttur. Her bir varyant farklı tur sayılarına sahiptir ve uygulamaya özel olarak bu tur sayıları değiştirilebilir.

ASCON, VE ve VEYA gibi basit mantıksal işlemleri kullanarak çalışır. ASCON, belirli bir güvenlik düzeyine ulaşmak için tekrarlanan turları kullanır ve 320-bit durumunda çalışır. Yer değiştirme katmanında S-Box'ları ve lineer katmanda sağa kaydırmalı dairesel dönüşü kullanarak şifreleme ve şifre çözme yapar. ASCON'un şifreleme algoritması temel olarak 4 adımdan oluşur. İlk olarak 320-bit durum ilklendirilir. Ardından yardımcı veri işlenir. Üçüncü adımda ise metin işlenir ve şifrelenmiş metin üretilir. Son adımda doğrulma etiketi üretilir. ASCON'un şifre çözme algoritması da benzer adımlara sahiptir. Farklı olarak üçüncü adımda şifreli metin işlenerek düz metin üretilir.

RISC-V, açık kaynaklı bir komut seti mimarisidir. 32-bit, 64-bit ve 128-bit olmak üzere üç çeşidi vardır. Ayrıca, tam sayılar ve tam olmayan sayılar ile işlemleri destekler. Toplama ve çıkarma gibi temel işlemler, çapma, bölünemez ve vektör işlemleri gibi işlemler için ayrı komut kümelerinin olması RISC-V'ı modüler kılar. Az

sayıda komut içermesinden dolayı ise öğrenmesi ve uygulaması kolay bir komut seti mimarisidir. RISC-V komut seti mimarisi ile tasarlanmış işlemciler, şifreleme, 5G, görüntü işleme vb. gibi belirli uygulamalarda performansı arttırmak için özel komutlar ile genişletilebilir.

GNU Derleyici Koleksiyonu açık kaynaklı bir projedir. GCC derleyicisini, assembler, hata ayıklayıcı ve bağlayıcı gibi uygulamaları içerir. GCC, C, C++ gibi yüksek seviyeli programlama dillerini optimizasyon da yaparak işlemcilerin çalıştırabileceği düşük seviyeli makine diline derler.

GCC, ön-uç, orta-uç ve arka-uç olmak üzere 3 bölümden oluşmaktadır. Ön-uç desteklenen bir programlama dilinde yazılmış kaynak kodu ortak bir orta-seviye dile çevirerek programlama dilinden soyutlama sağlar. Ön-uç hedef komut seti mimarisinden bağımsız bir katmandır. Orta-uç ise program üzerinde çeşitli optimizasyonları gerçekleştir. Orta-uç programlama dili ve hedef komut seti mimarisinden bağımsız bir katmandır. Arka-uç ise bazı ek optimizasyonlar yapar ve hedef işlemciye uygun makine kodlarını üretir. Arka-uç katmanı programlama dilinden bağımsız fakat hedef komut seti mimarisine ise bağımlı bir katmandır.

GCC yeni komut seti mimarileri için genişletilebilir ve komut seti mimarileri için yeni komutlar GCC'ye eklenebilir. GNU Derleyici Koleksiyonu RISC-V komut seti mimarisi için en popüler derleyici koleksiyonudur.

Spike bir RISC-V komut seti mimarisi simülatörüdür. Linux tabanlı işletim sistemleri üzerinde komut satırından bir uygulama olarak çalıştırılır. Spike, RISC-V işlemciler için derlenmiş bir programı, gerçek bir RISC-V işlemcisi kullanmadan çalıştırmayı sağlar. RISC-V komutlarının fonksiyonel davranışlarını içerir ve RISC-V işlemcisini yazılımsal olarak modeller. RISC-V işlemcisi gibi davranır ve RISC-V makine kodlarını içeren bir programı çalıştırır. Ayrıca, Spike üzerinden programın hata ayıklaması yapılabilir, işlemci hafızasındaki ve saklayıcılardaki değerler anlık olarak gözlemlenebilir. Bunlara ek olarak Spike da açık kaynaklı bir proje olduğu için, yeni RISC-V komutları simülatöre eklenip çalıştırılabilir.

Bu çalışmada, performansta en yüksek faydayı elde etmek ve en iyi özel komutları seçmek için bir profilleme metodolojisi geliştirilmiştir. ASCON'un 32-bit işlemciler için yazılım uygulaması analiz edilmiş ve profillenmiştir. Çeşitli kod bloklarının başı ve sonunda çalıştırılan komut sayısı ve saat döngüsü sayısının farkı alınarak proffilleme yapılmıştır. Algoritmada en çok gecikemeye sebep olan ve en çok tekrar eden işlemler tespit edilerek, bu işlemler için yeni komutlar belirlenmiştir.

Saat döngüleri sayısını ve komut belleğini azaltmak için ASCON algoritmasına özel kriptografik standart olmayan 32-bit RISC-V komutları geliştirilmiştir. Yeni komular 32-bit temel komular kümesine eklenerek, RISC-V komut seti mimarisi, ASCON şifreleme algoritması için genişletilmiştir. Bu yeni komutlar yer değiştirme ve sağa kaydırmalı dairesel dönüş işlemleri için tasarlanmıştır.

RISC-V işlemcileri için ASCON'a özel şifreleme komutları GNU Derleyici Koleksiyonu ve Spike RISC-V komut seti mimarisi simülatörüne eklenerek uçtan uca bir test ortamı oluşturulmuştur. GNU Derleyici Koleksiyonu'ndaki komut kalıpları ve derleyicide tanımlanmış gömülü özel fonksiyonlar örneklerle incelenerek teorik alt yapısı anlatılmıştır. Bu fonksiyonların ve komut kalıplarının derleyicideki kullanımı ve kullanım amacı uygulamalı örneklerle açıklanmıştır. RISC-V için toplama işlemini gerçekleştiren komut kalıbı incelenerek, GCC RISC-V arka-ucunun toplama işlemi komutlarının üretilmesi için ne gibi bilgilere ihtiyacı olduğu belirlenmiştir. Kayan nokta aritmetiği destekleten RISC-V işlemcileri için GCC'de bulunan bir gömülü fonksiyon incelenmiş ve yeni bi gömülü fonksiyonun GCC RISC-V arka-ucuna hangi bilgiler ile nasıl eklenmesi gerektiği gösterilmiştir. Bu bilgiler ve örnekler ışığında, ASCON'a özel RISC-V komutları için yeni özel gömülü fonksiyonlar ve komut modelleri GCC RISC-V arka-ucuna entegre edilmiştir. ASCON için tasarlanmış yeni gömülü fonksiyonlar, algoritmanın yazılım uygulamasına entegre edilmiş ve bu fonksiyonlara tekabül eden komutlara derlenmiştir.

Spike simülatörü, yeni komutları içeren programı çalıştırmak için, bu komutlarla genişletilmiştir. Yer değiştirme ve sağa dairesel kaydırma işlemlerinin fonksiyonel davranışları simülatöre eklenmiştir. Spike simülatörü bütün RISC-V komutlarını 1 saat döngüsü olarak saydığı için, gerçek bir donanım üzerinde 1 saat döngüsünden fazla süren komutların simüle edilmesi performans analizi anlamında yanlış sonuç edilmesine sebep olabilir. Bu sebeple, simülatör eklenen yeni komutların program üzerindeki toplam saat döngüsüne etkisini doğru hesaplamak adına, simülatöre yeni bir algoritma entegre edilerek sağlıklı bir ölçüm yapmak hedeflenmiştir.

Algoritmanın uygulamasındaki gecikme ve kullanılan komut belleği, önerilen yeni komutlar ve farklı derleyici optimizasyon bayrakları ile tekrar profillenmiştir. ASCON hafif şifreleme algoritmasının yazılım uygulamasındaki programın komut hafızası azaltılmıştır. Şifreleme ve şifre çözme işlemleri hızlandırılmıştır ve iyileştirmeler sunulmuştur.

1. INTRODUCTION

The internet of Things (IoT) has spread rapidly in recent years to connect devices to each other [7]. It has become a common technology in several domains such as smart cities, factory automation, smart home, etc. Since IoT uses physical world information from sensors mostly, information in the systems might be confidential and communication lines between devices might be exposed to cyber-attacks. To overcome security issues, cryptography could be used for security and privacy.

Traditional cryptography is generally used in desktop systems such as servers and personal computers (PC) [8]. Since embedded devices such as sensor nodes, radio frequency identification (RFID) tags, smart cards in IoT have limited resources in terms of memory, power, and area, traditional cryptography is not suitable for embedded systems. To ensure security and privacy on resource constrained devices, a lighter version of traditional cryptography which is lightweight cryptography (LWC) has emerged. There is a trade-off between performance and resources which are needed to execute a certain task in embedded systems [8], [9], and latency and memory are one of the performance metrics for the implementation of LWC algorithms.

Several instruction set architecture (ISA) implementations used by processor producers such as Intel, ARM, AMD, etc. are proprietary because of commercial reasons and protected by intellectual property rights. However, by using an open-source ISA [13], anyone can design a processor freely without getting the license from an ISA provider.

Application-specific instruction set processor [18] (ASIP) allows the design of a processor for a specific domain and is used to accelerate the operation of the software. RISC-V [10] is an open-source ISA developed by the University of California, Berkeley. It is a minimalist and modular ISA and easy to extend with non-standard custom instructions to design an ASIP. RISC-V community offers open source tools such as compilers, kernels, simulators, and hypervisors, etc. to design and develop RISC-V based processors. Design of embedded controllers could take advantage of RISC-V ISA for low-power applications.

1.1 Purpose of Thesis

In this study, ASCON [1], an LWC algorithm, has been analyzed and profiled to improve the performance in terms of latency and instruction memory of the software implementation of the algorithm. The operations that take the most clock cycles of execution of the algorithm are determined by profiling. The new custom instructions are chosen to improve the performance. RISC-V ISA has been extended with custom instructions for special cryptographic operations in the algorithm.

GNU Compiler Collection (GCC) [11] has been extended in the back end with new built-in functions and instruction patterns of the custom instructions. Spike RISC-V ISA Simulator [5] has been used to run the binaries of software implementation of ASCON on a simulation environment. The simulator has been extended for the new instructions. The effect of the new instructions on the implementation of the algorithm in terms of latency and memory consumption have been investigated by extending GCC and Spike RISC-V ISA simulator and improvement results have been shown.

The basic workflow of this thesis to achieve the operations mentioned above has the following steps:

1. Installation of RISC-V GNU Toolchain and Spike RISC-V ISA Simulator.
2. Profiling of software implementation of ASCON.
3. Decide and design custom RISC-V instructions for ASCON.
4. Extending the GCC Binutils and GCC back end for the custom instructions and built-in functions.
5. Extending Spike RISC-V ISA simulator for the custom instructions.
6. Integrating GCC built-in functions to software implementation of ASCON
7. Re-profiling the software implementation with extended GCC and Spike and show improvement results.

1.2 Literature Review

There are various studies to extend RISC-V ISA with new instructions for different applications such as post-quantum cryptography, lightweight cryptography, IoT, 5G, etc. Implementations of the instruction extensions for RISC-V processors are realized on various RISC-V cores. Pure hardware implementation of several lightweight cryptography algorithms such as ASCON, AEGIS, etc. are achieved on different FPGA boards.

RISC-V has been extended for some operations such as hardware loops, bit manipulations, arithmetic operations, etc. in the PULP Platform for energy-efficient signal processing applications [32]. GCC built-ins are added to the RISC-V back end of GCC to generate the extended instructions for GAP8 IoT application processor [19].

RISC-V extensions are implemented for the NTRU Cryptosystem in [20] for post-quantum cryptography. Custom instructions are designed for array modulus, array addition and array equalization operations in the algorithms of the NTRU cryptosystem. The instructions are integrated into a 32-bit IBEX core which supports I, M and C extensions of RISC-V and performance of the algorithm with new instructions is increased by 32,3%.

RISC-V is extended with domain specific vector operations for lattice-based cryptography of post-quantum cryptography for in [24]. To achieve high throughput and low latency in lattice-based cryptography for 5G applications, number theoretic transform (NTT) is vectorized with new vector instructions and control and status registers. Implementation of vector extension is made with Chisel hardware construction language and integrated into 32-bit SCR1 IMC RISC-V core.

Bit manipulation instructions for population count, parity and bit swapping algorithms are added to RISC-V in [31]. The instructions are generated with inline assembly in source code by extending GCC Binutils. The experiments are realized on an ETISS simulator that is an ISA-independent instruction set simulator, and performance effects of the new instructions are investigated. New instructions are added to the IP-XACT model of PULPino and RTL code of the extended processor is generated for a FPGA and binaries of source code are executed. Effort of a workflow to design and simulate new instructions and extend the GCC compiler, ETISS simulator and IP-XACT is presented.

In [21], implementation of lightweight cryptography algorithms such as PRESENT, GIFT, etc. are accelerated by adding new RISC-V instructions for 4x4 S-Box, permutation, and matrix multiplication operations. SpinalHDL is used as high-level hardware description language and instructions are integrated into 32-bit VexRiscv IMC core. The extended instructions are added to GCC Binutils and generated by using the inline assembly method. Resource usage overhead for flip flops and lookup tables and gain factors of using new instructions in terms of instruction count are shown.

FPGA implementation of winning lightweight cryptography algorithm candidates of the CAESAR competition AEGIS, ASCON and DEOXYS-II are implemented in [22] by using ZedBoard. Reconfiguration time of switching between algorithms in run time is reduced and synthesis results of each algorithm are shown.

In [23] ASCON is used to encrypt images for IoT applications. The system is implemented on SOC PYNQ FPGA board. Throughput of the software implementation on ARM Cortex A9 and hardware implementation are compared, and results are shown that hardware implementation is 13,5 times better than software implementation.

1.3 Contribution of Thesis

There are several studies to implement ASCON on FPGA for various applications but to the best of author's knowledge, there is no RISC-V ISA extension for ASCON. The generation of extended instructions are made with inline assembly in most of the works by extending the GCC Binutils. However, in this thesis, new instruction patterns and GCC built-ins are integrated to GCC RISC-V back end for cryptographic instructions for ASCON. A simulation environment is designed for the new instructions with Spike RISC-V ISA Simulator to test improvements rather than a hardware platform.

The 320-bit state of ASCON is the challenging part of this study because S-Box operation requires five 64-bit parallel applications. Since the algorithm is executed on an RV32I based processor in this study, five 64-bit parallel applications must operate on 32-bit integer registers simultaneously. To achieve this operation, appropriate instructions are designed and tested with a simulation environment.

This thesis is organized as follows. In Section 2, a background information about RISC-V ISA specification, GNU Compiler Collection and Spike RISC-V ISA Simulator is given. Section 3 explains how to install RISC-V tools and construct a simulation environment. In Section 4, lightweight cryptography and ASCON are introduced. In Section 5, the profiling method of the algorithm is explained, and profiling results are shown. Section 6 and 7 describe how RISC-V GNU Toolchain and Spike have been extended for the new instructions. Section 8 shows the results of improvements. Section Conclusion concludes the study and explains the future work.

2. RISC-V

Instruction set architecture (ISA) specifies how a processor executes a program and gives an abstraction between software and hardware. An ISA specifies supported data types, registers, addressing modes, addressing memory and instruction formats. On the other hand, microarchitecture defines how ISA is implemented on the hardware. Thus, different processors could have the same ISA but have different microarchitectures. ARM, MIPS, and x86 are examples of ISAs and different microarchitectures could be designed using these ISAs. The abstraction hierarchy of computer systems is shown in Figure 2.1.

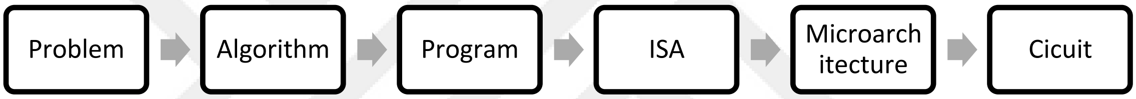


Figure 2.1: Abstraction hierarchy of computer systems.

Several ISAs used by such as Intel, ARM, AMD, etc. are private because of commercial reasons. Open-source ISAs such as RISCV, OpenRISC, and OpenSPARC are free to use and RISC-V belongs to a nonprofit foundation [13].

RISC-V ISA was started to develop by the University of California, Berkeley in 2010 [13]. RISC-V is a minimalist ISA meaning that its complexity is reduced compared to the other ISAs such as x86. RISC-V is a modular ISA, it has a base integer instruction set that never changes and optional extensions for specific applications such as multiplication, division, atomic, vector, and floating-point instructions. Since RISC-V is minimalist, it is useful for low energy with high-performance applications.

2.1 RISC-V ISA Specification

RISC-V has three base integer variants, RV32I, RV64I, and RV128I. RV32I has 32-bit, RV64I has 64-bit and RV128I has 128-bit address space [10]. RV32I variant of RISC-V has 32 x registers which are 32-bit wide. x0 is hard-wired to 0 and x1-x32 are general-purpose registers. Program counter (pc) register is an additional register that holds the address of the current instruction. In addition to the base integer set registers, RISC-V has 32 floating point registers and floating points status (fps) register. Registers of RISC-V are shown Figure 2.2. XLEN indicates the width of the integer register and FLEN indicates the width of the floating-point register of the RISC-V variant. For example, XLEN is 32 for RV32I.

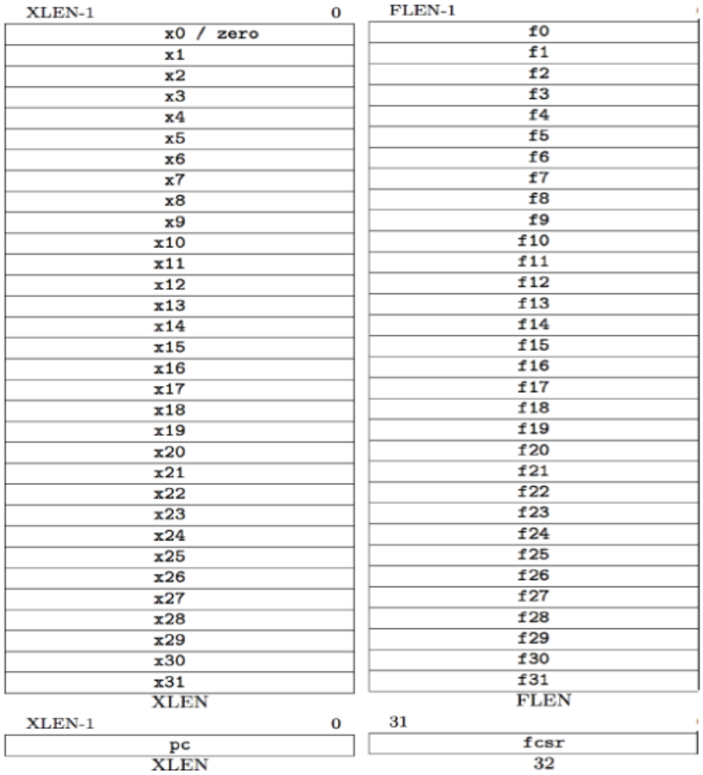


Figure 2.2: Register set of RISC-V [10].

RISC-V has four basic instruction formats, R, I, S, and U types and two additional formats B and J. All the instructions are fixed 32-bit wide. Base instruction formats are shown in Figure 2.3. *rs1* and *rs2* indicate source registers and *rd* indicates destination register of instruction. *Immediate* is used for constant values or offsets. *funct3* and *funct7* are the opcode parts and determined by GCC Binutils in Section 6.1.

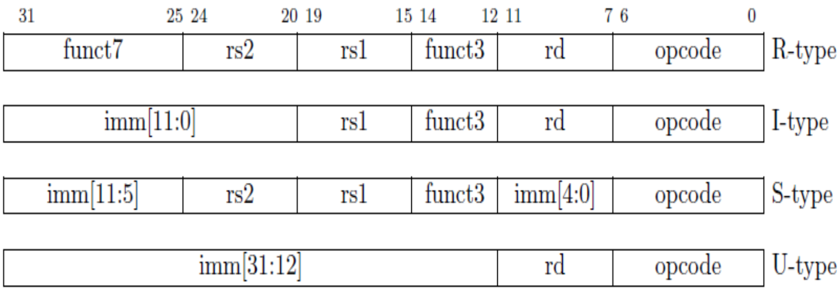


Figure 2.3: RISC-V base instruction formats [10].

RISC-V base integer instructions make basic integer arithmetic and logic computations such as addition, substruction, shift left/right, *AND*, *XOR*, *OR*, etc. It has integer register-register and integer immediate-register instructions. For example, the addition operation in RV32I is made with *add* instruction for register-register operation, *addi* for immediate-register operation. RISC-V also has control transfer instructions such as jump, two-register comparison. In RISC-V, memory is accessed by the only load and store instructions.

RISC-V has standard extensions in addition to the base integer set. It has an *M* extension for multiplication and division operations, *A* extension for atomic instructions. *F* extension is used for single-precision floating-point and *D* extension is used for double-precision floating-point operations. *C* extension has compressed instructions with 16-bit length. In addition to the standard extensions, RISC-V ISA can be extended with non-standard custom instructions for specific applications.

RISC-V ISA has control and status registers that provide information about the state and performance of the processor. The *cycle* register holds the count of the number of clock cycles executed by the processor. The *time* register counts the wall-clock time. The *instret* register counts the number of instructions executed by the processor. All three register's start time is an arbitrary time from the past. Control and status registers are accessed by control and status register instructions.

2.2 RISC-V Tools

The open-source RISC-V ecosystem has useful software resources for developing software for RISC-V based targets. These software resources include simulators such as Spike, QEMU, gem5 and MARSS-RISCV, toolchains such as Binutils and LLVM, debuggers such as GDB, compilers such as GCC, Clang/LLVM, bootloaders such as U-Boot, hypervisors, kernels such as Linux kernel and many more resources can be found in RISC-V official web page [26].

In this project, RISC-V GNU Toolchain [16] is used for optimizing, compiling, assembling, and linking software for RISC-V target. Execution of a program can be achieved with two options. The first option is running a program directly on a real RISC-V processor, the second option is running a program on a simulator. Because with the second option there is no need for effort to install a hardware environment, working with Spike ISA Simulator [5] is chosen for the test environment. An overview of this process is shown in Figure 2.4.

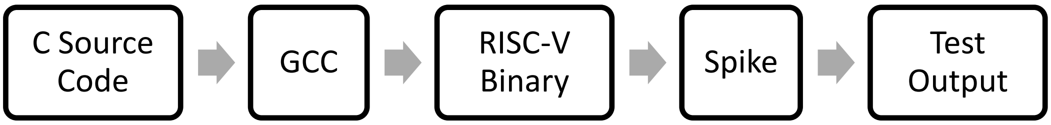


Figure 2.4: Overview of compilation and simulation process.

2.2.1 RISC-V GNU toolchain

RISC-V GNU toolchain has GCC cross compiler for C and C++ front ends, RISC-V back-end, and GDB. It also has Binutils which is a collection of binary tools such as assembler and linker. These open-source projects enable us to compile, optimize and debug C and C++ based software and create RISC-V executables.

GCC [11] is a compiler system created by GNU in 1984. GCC is one of the most significant components in GNU Toolchain which offers tools like a compiler, debugger, linker, assembler, etc. GCC supports high-level programming languages such as C, C++, Fortran, Ada, and Go [2]. Since GCC is an open-source project, it has an important role in developing software projects. GCC is the most popular compiler in software development.

GCC is a retargetable compiler meaning that it can generate code for different processors like ARM, RISC-V, MIPS [3]. Since GCC is portable, it can be modified and optimized for a specific processor.

Compilers convert high-level programming languages to a low-level machine-readable format which is assembly language. In the compilation process, the source code is preprocessed first. Preprocessing stage expands the source code according to preprocessor directives by separating comments and white spaces from source code. Compiler converts expanded code into assembly code. Checks and indicates syntax errors and warnings. After that, the assembler converts assembly code to object code. Generally, the source code is used together with libraries. These libraries are also compiled by the compiler as the source code and object files of libraries are created. Linker links all object files and creates an executable program. The compilation process of a C code is shown in Figure 2.5.

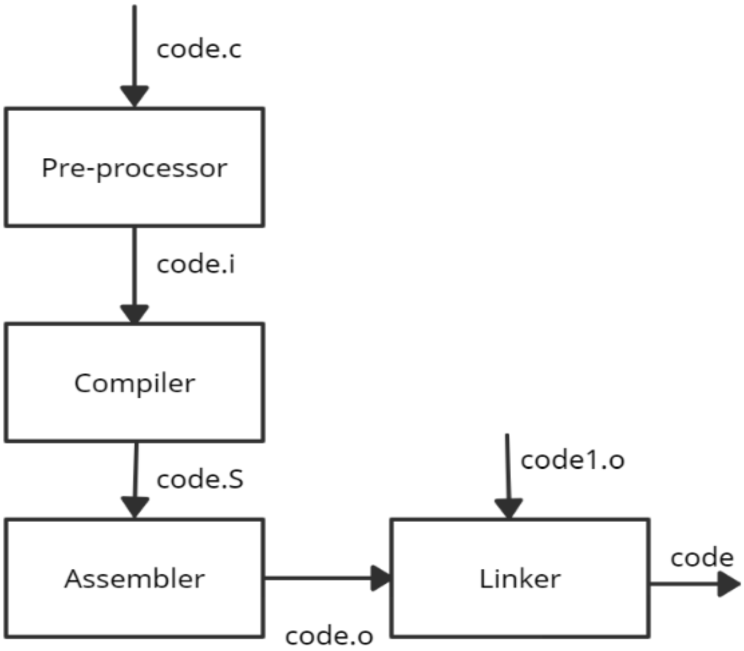


Figure 2.5: Compilation process of a C code.

The structure of GCC has three main parts, front end, middle end, and front end which is shown in Figure 2.6. The front end is a language-dependent and machine-independent part [2]. It scans and parses source code and makes lexical and syntactical analyses which are language dependent. GCC front end then creates language independent intermediate representation (IR) of the source code. In the front end, GCC uses GENERIC as IR to make abstraction from programming language used in source code and transition from front end to middle end.

The middle end is a language and machine-independent part [2]. In the middle end, GENERIC is converted to GIMPLE which is a tree-based IR. This transformation is called gimplification. GIMPLE is a simplified version of GENERIC, complex expressions in GENERIC are split into simpler expressions. GIMPLE is also language independent as GENERIC. Optimization is made in the middle end to accelerate the execution of the program and reduce the size of the executable. The input of the middle end is GENERIC, and output is register transfer language (RTL).

The back end takes RTL as input and produces machine code as output [2]. Some target dependent optimizations on RTL are made in the back end. The back end makes instruction scheduling and register allocation; thus, the back end is a machine-dependent part.

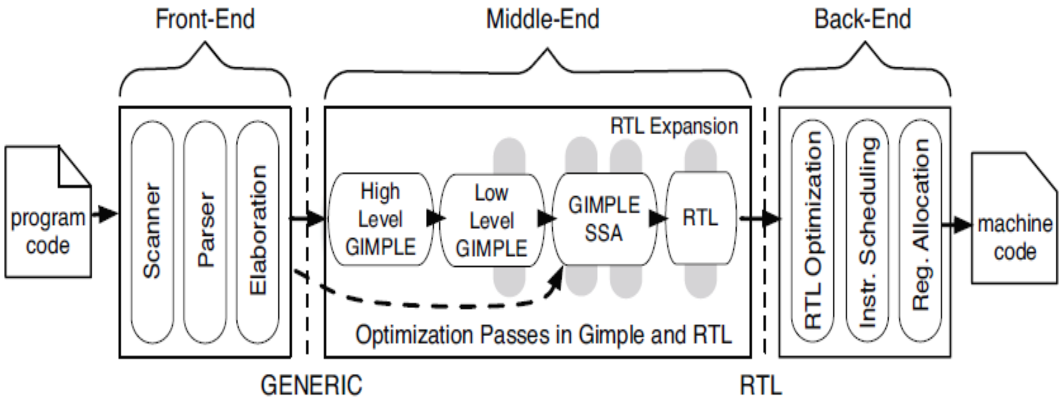


Figure 2.6: GCC compiler overview [12].

2.2.1.1 RTL and instruction patterns in GCC

Machine code for target ISA is generated from RTL representation in GCC back end with machine description file “<target.md>” under “riscv-gcc/gcc/config/<target>” directory. Instruction patterns map generated RTL to a machine code.

An instruction pattern is defined with *define_insn* expression [17]. *define_insn* has an optional *name*, has a *RTL template* which describes semantics of the instruction, a *condition* which is a C string and can be used optionally, an *output* statement which is a string or C code which returns a string and an optional *attribute* which adds extra information of instruction pattern. Structure of *define_insn* in a “<target.md>” file is shown in Figure 2.7.

```
(define_insn <name>-Optional
  RTL template
  C Condition
  Output
  Attribute-Optional
)
```

Figure 2.7: *define_insn* structure in a <target.md> file.

For example, defining the *addsi3* instruction pattern in “riscv.md” is shown in Figure 2.8. It describes an addition operation in RTL and maps it to one of the output *addw*, *addiw*, *add* or *addi* RISC-V machine instructions according to the target architecture and instruction operand. It has a condition which is an empty string and type and mode attributes which are defined on the top of riscv.md file. The parts of the *addsi3* instruction pattern is shown in Table 2.1.

```
(define_insn "addsi3"
  [(set (match_operand:SI          0 "register_operand" "=r,r")
  (plus:SI (match_operand:SI 1 "register_operand" " r,r")
      (match_operand:SI 2 "arith_operand"     " r,I")))]
  ""
  { return TARGET_64BIT ? "add%i2w\t%0,%1,%2" : "add%i2\t%0,%1,%2"; }
  [(set_attr "type" "arith")
   (set_attr "mode" "SI")])
```

Figure 2.8: *addsi3* instruction pattern in riscv.md file.

Table 2.1: Structure of *addsi3* instruction pattern.

Name	addsi3
RTL Template	[(set (match_operand:SI 0 "register_operand" "=r,r") (plus:SI (match_operand:SI 1 "register_operand" "r,r") (match_operand:SI 2 "arith_operand" "r,I")))]
Condition	""
Output	{ return TARGET_64BIT ? "add%i2w\t%0,%1,%2" : "add%i2\t%0,%1,%2"; }
Attribute	[(set_attr "type" "arith")(set_attr "mode" "SI")]

- *(set lval x)* is an RTL operator which represents the storing value of *x* into *lval*. In *addsi3* instruction pattern, it sets the result of *plus* operation into output operand.
- *(plus:m x y)* is an arithmetic expression which represents the *sum* of *x* and *y* with machine a mode *m* which is *SI* for the *addsi3* instruction pattern. *SI* is single integer mode for four-byte integer registers.
- *(match_operand:m n predicate constraint)* expression basically decides which operand in the RTL must be matched with which operand in the *addsi3* instruction pattern. *n* represents the operand number in the instruction pattern. 0 represents output operand, and 1 and 2 represent the input operands of *addsi3*.
- *Predicate* decides if the instruction pattern matches the RTL. *register_operand* and *arith_operand* are predicates which are defined in the “predicates.md” file in the RISC-V back end of GCC. *register_operand* is used for arithmetic operations. *arith_operand* covers the *register_operand* and constant integers.
- *Constraint* chooses the best register class which is allowed by the predicate. *r* represents a general register for *register_operand*. = is a constraint modifier and it

means that the instruction pattern writes to this operand a new value. *I* is a RISC-V specific constraint, it represents an I-type 12-bit signed immediate operand which is defined in "constraints.md" file in RISC-V back end of GCC. *arith_operand* allows *r* and *I* as register classes. Tree representation of RTL which defines the *addsi3* instruction pattern is shown in Figure 2.9.

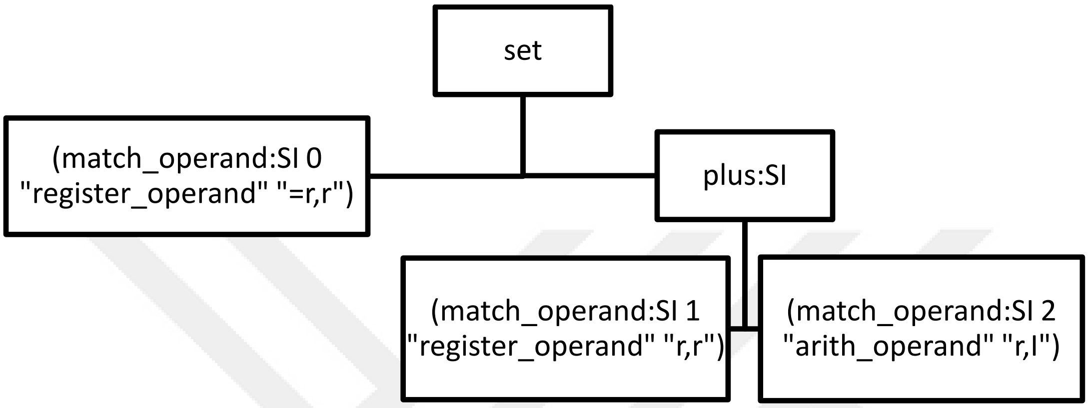


Figure 2.9: Tree representation of RTL template of *addsi3* instruction pattern.

- *Output template* is a string which describes which machine instruction will be produced for the instruction pattern. With *%* followed by *2i*, the compiler decides which instruction will be generated according to the register constraint of the operand whose operand number *n* is 2. For example, if *r* is chosen as constraint for the second input operand, *add* or *addw* is produced, if *I* is chosen *addi* or *addiw* is produced. C statement with a condition alternates the output of *addsi3*. If target is 64-bit, output will be *addw* or *addiw* else *add* or *addi* machine instruction. Output of *addsi3* instruction pattern according to the constraint and target architecture is shown in Table 2.2.

Table 2.2: Outputs of *addsi3* instruction pattern.

	Register class of the second input operand	
	r(register)	*I(immediate)*
Target is 64-bit	*addw*	*addiw*
Target is not 64-bit	*add*	*addi*

- *(set_attr name value-string)* is an attribute expression which sets the *name* attribute to *value-string* value. It classifies the addsi3 instruction pattern with *arith* type and *SI* mode.

2.2.1.2 Inline assembly and GCC built-ins

Inline assembly enables us to use assembly code in a program which has coded with a high-level programming language such as C [15]. Since target specific machine instructions can be generated directly from source code, performance of the program could be enhanced. Inline assembly method is used for compensating unavailable functionality of C language. Since GCC does not have any functionality to generate custom instructions from the C front end, an inline assembly method is used to generate these custom instructions.

An inline assembly statement for *add* RISC-V instruction is shown in Figure 2.10. *asm* keyword is used to create an inline assembly block within C code[28]. The assembler template is a string and refers to the instruction name, input, and output operands of the instruction. *result* is written into the output operand *rd*. Inputs of the addition operation, *x* and *y*, are written into input operands, *rs1* and *rs2*, respectively.

GCC may not produce the machine instruction in the *asm* statement because of optimizers. *volatile* keyword is used to tell the compiler that not to make any optimization on *asm* block and generate the instruction in the output.

```
int x,y,result;

asm volatile ("add %[rd], %[rs1], %[rs2]\n\t"  //Assembler Template
              :[rd]  "=r" (result)             //Output Operands
              :[rs1] "r" (x), [rs2] "r" (y));  //Input Operands
```

Figure 2.10: Inline assembly statement for *add* RISC-V instruction.

Inline assembly is generally used to generate a single instruction with multiple code lines [15]. Thus, using it in the codebase makes code more complex to read and difficult to understand. Also, many static code analyzers do not support inline assembly code blocks. Built-in functions can be used to overcome disadvantages of inline assembly.

GCC has built-in functions which also can cover inabilities of C language [14]. They can be compiled directly into a specific machine instruction; thus, built-in functions could be used for optimization and performance enhancement of the program as inline assembly. They are declared implicitly in GCC, so the compiler knows the built-in functions. Architecture independent built-in functions of GCC are useful for logical operations, atomic operations, memory allocation and many more. Architecture specific built-in functions could be used for generating target ISA specific instruction.

There are two types of built-in functions, standard library functions such as *malloc*, *str_cpy* and *printf* and not standard library functions which are also called *intrinsics*. GCC Intrinsic functions have *__builtin* prefix. RISC-V specific built-in functions in GCC have *__builtin_riscv* prefix.

Using built-in functions in codebase make programs more readable and offer a cleaner abstraction than inline assembly. However, it requires more effort and GCC internal knowledge to add a new built-in function to GCC.

2.2.1.3 GCC RISC-V built-in functions

In RISC-V back end of GCC, argument types and function types of RISC-V specific built-in functions are defined. It contains several macros and functions to create new built-in functions under "src/riscv/gcc/config/riscv/" directory. The primary files to construct a built-in function are "riscv-builtins.c" and "riscv-ftypes.def".

riscv_builtin_description struct holds information about a single RISC-V built-in function in the "riscv-builtins.c" file. It has an *icode* for corresponding instruction pattern name which is defined in the "riscv.md" file and *name* for name of the built-in function. *built-in type* specifies the type of the built-in. *prototype* corresponds to the prototype of the built-in which is defined in "riscv-ftypes.def" file. *avail* is an availability predicate for the built-in. *riscv_builtin_description* is shown in Figure 2.11.

```
struct riscv_builtin_description {
  /* The code of the main .md file instruction.  See riscv_builtin_type
     for more information.  */
  enum insn_code icode;

  /* The name of the built-in function.  */
  const char *name;

  /* Specifies how the function should be expanded.  */
  enum riscv_builtin_type builtin_type;

  /* The function's prototype.  */
  enum riscv_function_type prototype;

  /* Whether the function is available.  */
  unsigned int (*avail) (void);
};
```

Figure 2.11: *riscv_builtin_description* struct in riscv-builtins.c file.

If a built-in function corresponds directly to an instruction pattern in “riscv.md” file, built-in type is *RISCV_BUILTIN_DICRECT*, likewise but it has return type *void* it is *RISCV_BUILTIN_DICRECT_NO_TARGET*. *riscv_builtin_type* is shown in Figure 2.12.

```
enum riscv_builtin_type {
  /* The function corresponds directly to an .md pattern.  */
  RISCV_BUILTIN_DIRECT,

  /* Likewise, but with return type VOID.  */
  RISCV_BUILTIN_DIRECT_NO_TARGET
};
```

Figure 2.12: *riscv_builtin_type* in riscv-builtins.c file.

The *DIRECT_BUILTIN* macro contains the *RISCV_BUILTIN* macro which constructs a *riscv_builtin_description* with given arguments. These macros are shown in Figure 2.13. *DIRECT_BUILTIN* takes the name of instruction pattern, function type and availability predicate of the built-in as input arguments. It passes these arguments to *RISCV_BUILTIN* to define a built-in-in function.

```
#define RISCV_BUILTIN(INSN, NAME, BUILTIN_TYPE, FUNCTION_TYPE, AVAIL) \
  { CODE_FOR_riscv_ ## INSN, "__builtin_riscv_" NAME,      \
    BUILTIN_TYPE, FUNCTION_TYPE, riscv_builtin_avail_ ## AVAIL }

/* Define __builtin_riscv_<INSN>, which is a RISCV_BUILTIN_DIRECT function
   mapped to instruction CODE_FOR_riscv_<INSN>,  FUNCTION_TYPE and AVAIL
   are as for RISCV_BUILTIN.  */
#define DIRECT_BUILTIN(INSN, FUNCTION_TYPE, AVAIL)       \
  RISCV_BUILTIN (INSN, #INSN, RISCV_BUILTIN_DIRECT, FUNCTION_TYPE, AVAIL)
```

Figure 2.13: *RISCV_BUILTIN* and *DIRECT_BUILTIN* macros in riscv-builtins.c file.

For example, *frflags* is a pseudo instruction of RISC-V which reads the floating-point exception flags, *fflags*. RISC-V back end of GCC has a built-in function called *__builtin_riscv_frflags* which is mapped directly into RTL of *riscv_frflags* instruction pattern. Output of this instruction pattern is *frflags* machine instruction. The built-in function does not take any input argument and returns an unsigned integer. Definition of *__builtin_riscv_frflags* built-in function has the following items:

- A macro is defined to create an enumeration identifier for the function prototype in “riscv-builtins.c” as shown in Figure 2.14. In the *RISCV_FTYPE_NAME1(A, B)* macro, *A* represents the return of the built-in function and *B* represents the input argument of the built-in function.

```
#define RISCV_FTYPE_NAME1(A, B) RISCV_##A##_FTYPE_##B
```

Figure 2.14: Function type of *__builtin_riscv_frflags*.

```
#define RISCV_FTYPE_ATYPES1(A, B) \
  RISCV_ATYPE_##A, RISCV_ATYPE_##B
```

Figure 2.15: Argument types of prototype of *__builtin_riscv_frflags*.

- The input argument of the built-in function is *void* and it returns an unsigned integer. Thus, argument type codes of the built-in function are defined as in Figure 2.16.

```
#define RISCV_ATYPE_VOID void_type_node
#define RISCV_ATYPE_USI unsigned_intSI_type_node
```

Figure 2.16: Argument type-codes of *__builtin_riscv_frflags*.

- DEF_RISCV_FTYPE(NARGS, LIST) is defined for each function prototype in the"riscv-ftypes.def" file. NARGS is the number of input arguments of a built-in function, LIST contains the return-type code followed by the code for each argument type. Function type definition of the __builtin_riscv_frflags is shown in Figure 2.17. void input operand is counted as 1 for NARGS. USI and VOID correspond to return-type code and input-type code, respectively.

```
DEF_RISCV_FTYPE (1, (USI, VOID))
```

Figure 2.17: Function prototype *__builtin_riscv_frflags*.

- Built-in function *__builtin_riscv_frflags* is constructed in *riscv_builtins* array as shown in Figure 2.18. It is constructed with *DIRECT_BUILTIN* since its return type is not void.
- The name of the instruction pattern corresponding to *__builtin_riscv_frflags* is *riscv_frflags*. It is given as a parameter to the *INSN* argument without *riscv_* prefix because *RISCV_BUILTIN* inside *DIRECT_BUILTIN* adds this prefix automatically.
- Function type is given with *RISCV_USI_FTYPE_VOID* as defined in Figure 2.17.
- *hard_float* is the name of the availability predicate of the built-in function as shown in Figure 2.19. It returns *true* if the processor supports the float instruction

extension. Thus, this built-in function is available if the "-march" flag of the compiler contains the *F* extension of RISC-V.

```
static const struct riscv_builtin_description riscv_builtins[] = {
  DIRECT_BUILTIN (frflags, RISCV_USI_FTYPE_VOID, hard_float),
```

Figure 2.18: Construction of *__builtin_riscv_frflags*.

```
AVAIL (hard_float, TARGET_HARD_FLOAT)
```

Figure 2.19: Availability predicate of *__builtin_riscv_frflags*.

Usage of *__builtin_riscv_frflags* is shown in Figure 2.20. By dividing *c* by zero creates a dividing by zero (DZ) exception in the processor. This operation makes the fourth bit, DZ, in *fflags* true. The exception is read with *__builtin_riscv_frflags*. It has no input argument and returns the reading of floating-point exception flags. The returned value is 8 in decimal and 1000 in binary which corresponds to 1 for DZ bit in *fflags* [10].

```
#include "stdio.h"

int main()
{
  float a,b,c ;
  a = 5.5;
  b = 0;
  c = a/b;
  unsigned int ret_val = __builtin_riscv_frflags();
  printf("%d\n",ret_val);
}
```

Figure 2.20: Usage of *__builtin_riscv_frflags*.

2.2.2 Simulation environment

Spike [5] is an open source RISC-V ISA simulator which is offered by the RISC-V community. Spike executes statically linked RISC-V ELF binaries on a host environment. By using Spike, the ELF binaries that are compiled for a RISC-V processor can be executed without a RISC-V processor. It fetches, decodes, and executes RISC-V instructions by implementing the functional model of RISC-V cores as shown in Figure 2.21.

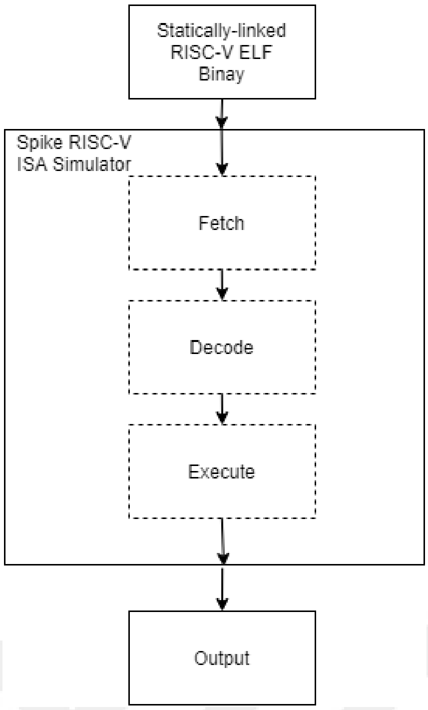


Figure 2.21: Execution of a RISC-V ELF binary on Spike.

Spike supports the 32-bit and 64-bit versions of RISC-V that have extensions such as I, M, A, F, D, and C, etc. Spike also can be extended to simulate new custom instructions by adding functional behavior, mask, and match opcodes of the new instructions to the simulator.

Spike is run from command line interface with many options such as processor count of target hardware, memory regions, starting on halt, privilege modes, and target ISA version and extensions.

Spike also has interactive debug mode to see the contents of a register or a memory location or execute instructions one-by-one. Since Spike acts like it is real hardware, it also supports debugging with GDB.

Pk [30] is an open source RISC-V proxy kernel and handles the system calls to the host computer. It can be built for RV32 and RV64 versions of RISC-V. Simulating an RISC-V ELF on Spike is realized with giving pk, options if needed, and ELF to be executed as input arguments.

3. BUILDING RISC-V TOOLS

Early in the project, RISC-V GNU Toolchain from official RISC-V sources was tried to install but it did not work due to incompatible versions and sub repositories. Thus, SiFive's open-source RISC-V GNU Toolchain called Freedom Tools [27] are preferred. The installed OS version and RISC-V repositories that will be modified in this project are shown in Table 3.1 with git branch names and the last git commits.

Table 3.1: Versions of the tools.

OS	*Ubuntu 18.04.5 LTS*	
Repository	Git Branch	Git Commit ID
riscv-gcc	*sifive-gcc-8.2.0*	*421d786e300f78054658d8892a26a3cdc4db3fcb*
riscv-binutils	*sifive-rvv-1.0.x-zfh-rvb*	*75d2236ce26a3048f52bbd5186602e27bd635e2b*
Spike	*master*	*032a68c3e94b0d315207a5a42251f836185b5d01*

3.1 Building RISC-V GNU Toolchain

Before the installation, prerequisites must be installed with the following commands.

```
$ sudo apt-get install cmake autoconf automake autotools-dev curl
libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex
texinfo gperf patchutils bc zlib1g-dev libexpat-dev libtool pkg-
config mingw-w64 mingw-w64-tools texlive zip python-dev gettext
libglib2.0-dev libpixman-1-dev swig ninja-build python3
$ sudo pip3 install meson
```

GCC Toolchain for 64-bit Ubuntu operating system is installed with the following commands.

```
$ git clone git@github.com:sifive/freedom-tools.git
$ cd freedom-tools
$ git submodule update --init --recursive
$ make ubuntu64-toolchain -j4
```

To run binaries without giving an installation path from bash, the installation path of RISC-V binaries should be added to the ".bashrc" file in the "/home/user" directory. With this modification, binaries can be used directly without giving any installation path. Setting the RISC-V environment and adding the installation path are shown in Figure 3.1.

```
# enable programmable completion features (you don't need to enable
# this, if it's already enabled in /etc/bash.bashrc and /etc/profile
# sources /etc/bash.bashrc).
if ! shopt -oq posix; then
  if [ -f /usr/share/bash-completion/bash_completion ]; then
    . /usr/share/bash-completion/bash_completion
  elif [ -f /etc/bash_completion ]; then
    . /etc/bash_completion
  fi
fi

export RISCV=$HOME/Desktop/freedom-tools/obj/x86_64-linux-ubuntu14/install/riscv64-unknown-elf-gcc-8.3.0-2020.04.1-x86_64-linux-ubuntu14
export PATH=$RISCV/bin:$PATH
```

Figure 3.1: Modification of .bashrc.

It can be verified that the RISC-V environment and installation path is correct with the following commands.

```
$ echo $RISCV
$ echo $PATH
```

When the installation is completed, GCC binaries with the name “riscv64-unknown-elf-*” can be found under the “$RISCV/bin/” directory. Unknown means that these binaries can be used for bare-metal hardware, instructions have full access to address space of the processor. A list of GCC binaries for RISC-V based targets is shown in Figure 3.2. *riscv64-unknown-elf-gcc* is the cross compiler to compile, assemble, and link C source files for RISC-V processors. It behaves similarly to the standard GCC; however, it produces binaries compiled into the RISC-V instruction set. *riscv64-unknown-elf-objdump* is used to display assembly code of object files.

```
elf2hex                          riscv64-unknown-elf-g++          riscv64-unknown-elf-gcov-tool       riscv64-unknown-elf-objdump
riscv64-unknown-elf-addr2line    riscv64-unknown-elf-gcc          riscv64-unknown-elf-gdb             riscv64-unknown-elf-ranlib
riscv64-unknown-elf-ar           riscv64-unknown-elf-gcc-8.2.0    riscv64-unknown-elf-gdb-add-index   riscv64-unknown-elf-readelf
riscv64-unknown-elf-as           riscv64-unknown-elf-gcc-ar       riscv64-unknown-elf-gprof           riscv64-unknown-elf-run
riscv64-unknown-elf-c++          riscv64-unknown-elf-gcc-nm       riscv64-unknown-elf-ld              riscv64-unknown-elf-size
riscv64-unknown-elf-c++filt      riscv64-unknown-elf-gcc-ranlib   riscv64-unknown-elf-ld.bfd          riscv64-unknown-elf-strings
riscv64-unknown-elf-cpp          riscv64-unknown-elf-gcov         riscv64-unknown-elf-nm              riscv64-unknown-elf-strip
riscv64-unknown-elf-elfedit      riscv64-unknown-elf-gcov-dump    riscv64-unknown-elf-objcopy         spike
```

Figure 3.2: List of GCC binaries after installation.

3.1.1 Compiling a hello world program

When the installation is completed, a C code can be compiled to a RISC-V ELF. A simple hello world application “hello.c” is shown in Figure 3.3.

```c
#include "stdio.h"

int main()
{
  printf("Hello World\n");
  return 0;
}
```

Figure 3.3: Simple hello.c program.

Compilation can be done with the following command.

```
$ riscv64-unknown-elf-gcc hello.c -o hello.out
```

A 64-bit ELF for RISC-V ISA is created with compilation. The type of the binary file can be seen with the following command.

```
$ file hello.out
```

Since the RISC-V variant in this project is RV32I, 32-bit ELF can be compiled with the following command.

```
$ riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 hello.c -o hello.out
```

- The arguments begin with "-m" are specific to the RISC-V architecture port in GCC. For the tool *riscv64-unknown-elf-gcc*, the "-march" argument selects architecture to target, controls which instructions and registers will be produced with compilation. For example, if the target processor supports just a 32-bit base integer set, it would be "-march=rv32i". Using "-march=rv32imafdc" means that the compiler can generate 32-bit *I*, *M*, *A*, *F*, *D*, and *C* instruction extensions, it can be used like "-march=rv32gc" alternatively. "g" represents *I*, *M*, *A*, *F*, and *D* instruction extensions.
- The "-mabi" argument selects the ABI to target, controls the calling convention and the layout of data in memory "-mabi=ilp32" means that int, long, and pointers are all 32-bits long, char is 8-bit and short is 16-bit. "-march=rv32imafdc -mabi=ilp32" means that hardware floating-point instructions can be generated, but no floating-point arguments will be passed in registers.
- "-o" argument specifies the output name of the executable. More GGC options for RISC-V can be found in [28].

If the type of hello.out is displayed, it is a 32-bit RISC-V ELF.

Since the program in Figure 3.4 is compiled for RISC-V based targets, its executable file has RISC-V machine instructions. Information about object files can be displayed using the *riscv64-unknown-elf-objdump* tool. This tool has arguments for specific options as the other GCC binaries. "-d" argument displays assembler contents of executable sections, "—visualize-jumps" argument is useful for visualize jumps in the program such as loops, "-M" is a switch for RISC-V specific disassembler options as mentioned above, “no-aliases” option disassembles only into canonical instructions, rather than into pseudo instructions.

Dump file hello.S which has assembly code of hello.out can be generated with the following command.

```
$ riscv64-unknown-elf-objdump -d --visualize-jumps -M no-aliases
hello.out > hello.S
```

Some RISC-V machine instructions such as *addi*, *jal*, *mv* and *lw* can be seen in the “hello.S” dump file. Main function in the assembly file is shown in Figure 3.4.

```
0001018c <main>:
   1018c: ff010113          addi  sp,sp,-16
   10190: 00112623          sw   ra,12(sp)
   10194: 00812423          sw   s0,8(sp)
   10198: 01010413          addi  s0,sp,16
   1019c: 000137b7          lui a5,0x13
   101a0: 66078513          addi  a0,a5,1632 # 13660 <__modsi3+0x30>
   101a4: 304000ef          jal ra,104a8 <puts>
   101a8: 00000793          addi  a5,zero,0
   101ac: 00078513          addi  a0,a5,0
   101b0: 00c12083          lw  ra,12(sp)
   101b4: 00812403          lw  s0,8(sp)
   101b8: 01010113          addi  sp,sp,16
   101bc: 00008067          jalr  zero,0(ra)
```

Figure 3.4: Assembly code of main function for RISC-V processors.

3.2 Installation of Spike and Pk

Before installation, the RISC-V environment variable must be set to the RISC-V tools install path as mentioned in Section 3.1.

Installation of Spike can be done with following commands.

```
$ git clone git@github.com:riscv/riscv-tools.git
$ cd riscv-tools/riscv-isa-sim
$ apt-get install device-tree-compiler
$ mkdir build
$ cd build
$ ../configure --prefix=$RISCV
$ make
$ sudo make install
```

Pk for RV32 can be installed with the following commands. It will install Pk for RV64 without specifying the “—with-arch” option.

```
$ cd riscv-tools/riscv-pk
$ mkdir build
$ cd build
```

```
$ ../configure --prefix=$RISCV --host=riscv64-unknown-elf –with-arch=rv32imafdc
$ make
$ sudo make install
```

3.2.1 Running hello world program on Spike

After installation of Spike and Pk, “hello.out”, compiled RISC-V ELF binary, can be executed with the following command. “-isa=RV32IMAFDC” selects the target architecture. It also takes pk and ELF which will be executed as the input arguments.

```
$ spike –isa=RV32IMAFDC $RISCV/riscv32-unknown-elf/bin/pk hello.out
```

4. LIGHTWEIGHT CRYPTOGRAPHY

Lightweight cryptography is a sub-field of conventional cryptography to ensure confidentiality of communication between devices that have limited resources [8]. National Institute of Standards and Technology (NIST) started to work on the standardization of LWC algorithms in 2013 for resource constraint devices. While conventional cryptography is mostly used in servers, desktops, and smartphones, for resource-constrained devices it is not suitable because of security, performance, and resource requirements. Since embedded systems have limited resources in terms of memory, power, and area, but it requires a certain performance level with low energy consumption, LWC is purposed to provide high security and privacy with a low cost for resource constraint devices such as sensor nodes, RFID tags, smart cards which can be seen in Table 4.1.

Table 4.1: Device spectrum of conventional and lightweight cryptography [8].

Conventional Cryptography	Lightweight Cryptography
Servers and Desktops	Embedded Systems
Tablets and Smartphones	RFID and Sensor Networks

There is a tradeoff between performance and resources which are needed to execute a certain task in embedded systems. Since LWC algorithms are designed for embedded systems, their performance metrics, and resources they consumed are related. For LWC, the performance metrics are memory requirements, gate area, power consumption, latency, and throughput.

During the execution of a program, it uses RAM to store runtime values, program code, and constant values such as S-Boxes and hardcoded round keys [9]. The gate area defines the physical area to implement the algorithm in terms of hardware. Count of the logic blocks on an FPGA can be used to describe the gate area. Since the devices are resource constrained, power consumption is one of the key elements. It describes how much power is needed to execute the algorithm. Latency is the amount of total time between the start and end of the operating algorithm. Throughput is the rate of producing output.

As shown in Figure 4.1, ciphers can be divided into two main categories, symmetric and asymmetric cipher [8]. In a symmetric cipher, the same private key is used to encrypt and decrypt the information however asymmetric cipher uses two keys, public key, and private key, for encryption and decryption respectively.

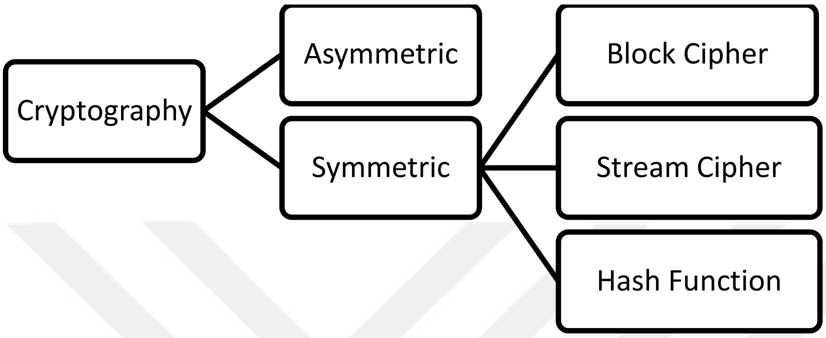


Figure 4.1: Classification of cryptography [7].

Block cipher has fixed size blocks for encryption and decryption [9]. It uses confusion and diffusion steps to manipulate the information. In block cipher, each round takes the data and the key and implements an encryption algorithm and generates ciphertext. The generation of the ciphertext process is shown in Figure 4.2. The significant parameters for a block cipher are the number of rounds, block size, and key size. Stream cipher uses only confusion steps. Hash function is a one-way function that maps information to fixed-size data. Because of the ease of achieving performance criteria of LWC, block ciphers are used for LWC.

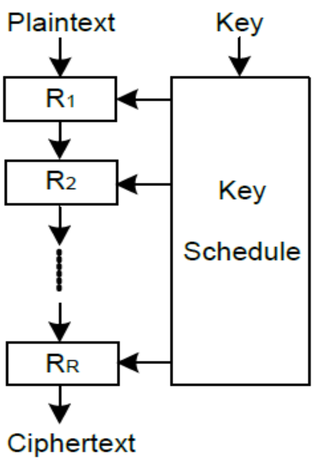


Figure 4.2: Block cipher algorithm [9].

According to the report of NIST, to increase the performance of a lightweight block cipher algorithm there are some lightweight design choices [8]. The key size must be larger than 112 bits. Rounds must be simple. It is recommended that rounds should use 4x4 S-Boxes to keep the gate area minimum. Simple linear layers such as bit permutations should be preferred as diffusion steps. While keeping rounds simple, rounds should be more iterative to achieve a certain security level. Complex key scheduling causes using more resources, therefore an LWC algorithm should use a simple key scheduler. While implementing an LWC algorithm, only needed functionalities should be considered not to waste limited resources of the device.

NIST also considers some design requirements which should be common to all LWC algorithms. To keep security strength, the key size must be at least 112 bits. The algorithm should be flexible, meaning that making changes on implementation, tuning parameters for a specific usage are easy to apply, and different versions of a single platform should be permitted. Encryption and decryption should share a common logic, this reduces the overhead for multiple functions. The size of ciphertext should not be larger than plaintext. The algorithm should be resistive against attacks such as side-channel and fault attacks.

Finally, the reason for the necessity of LWC, performance metrics, design choices, and design requirements are summarized in this section. A brief of these profile characteristics for LWC is shown in Figure 4.3.

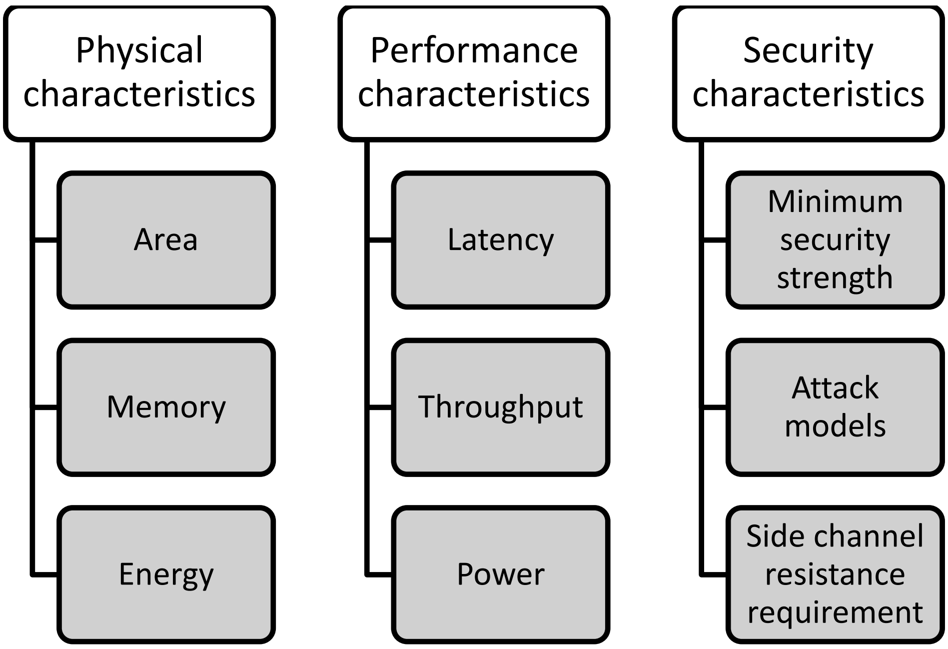


Figure 4.3: Profile characteristics of lightweight cryptography [8].

4.1 ASCON

NIST coordinates CEASAR competition for authenticated ciphers in secret-key cryptography [25]. CEASAR's portfolio has three use cases, lightweight applications for resource constrained environments, high-performance applications, and defense in depth. ASCON [1] is one of the finalists of LWC algorithms in Round 2 of CEASAR competition.

ASCON is a cipher suite that offers authenticated encryption with associated data (AEAD) and hashing functionality. The suite includes authenticated ciphers Ascon-128 and Ascon-128a, the hash function Ascon-Hash. ASCON uses 320-bit states with 64-bit words. To operate on 32-bit, 16-bit, or 8-bit systems, ASCON uses the bit-interleaving method. Each round contains a substitution layer and a linear layer. The number of rounds is changeable according to implementation to maximize security and robustness.

The authenticated encryption process $\mathcal{E}_{k,r,a,b}$ takes secret key *K* with *k* bits, a nonce *N*, an associated data *A* and a plaintext *P* as input as shown in equation 4.1. It produces a ciphertext *C* with an authentication tag *T* as output. Ciphertext has the same length as plaintext. Bit length *k* of the key is less than 160-bits. The data block size, rate, is represented with *r*, internal round numbers are represented with *a* and *b*.

$$\mathcal{E}_{k,r,a,b}(K,N,A,P) = (C,T) \quad (4.1)\ [1]$$

The decryption process $\mathcal{D}_{k,r,a,b}$ takes secret key *K* with *k* bits, a nonce *N*, an associated data *A*, a ciphertext *C*, and an authentication tag *T* as input as shown in equation 4.2. If the verification tag is correct, it produces a plaintext *P* else it produces $\perp$ as output.

$$\mathcal{D}_{k,r,a,b}(K,N,A,C,T) \epsilon \{P, \perp\} \quad (4.2)\ [1]$$

The hashing process $\mathcal{X}_{h,r,a}$ takes input message *M* with arbitrary length *l* and maps this input message to output *H* as shown in equation 4.3. *h* describes the output length; *r* is the data rate and *a* is the round number.

$$\mathcal{X}_{h,r,a}(M, l) = H \quad (4.3)\ [1]$$

The recommended parameter sets for encryption and hashing can be seen in Table 4.2 and Table 4.3, respectively.

Table 4.2: Recommended parameters for authenticated encryption in ASCON [1].

Name	Algorithms	Bit size of				Rounds	
		key	nonce	tag	data block	p^a	p^b
Ascon-128	$\mathcal{E}, \mathcal{D}_{128,64,12,6}$	128	128	128	64	12	6
Ascon-128a	$\mathcal{E}, \mathcal{D}_{128,128,12,8}$	128	128	128	128	12	8

Table 4.3: Recommended parameters for hashing algorithm in ASCON [1].

Name	Algorithm	Bit size of		Rounds
		hash	data block	p^a
Ascon-Hash	$X_{256,64,12}$ with $l = 256$	256	64	12

The ASCON cipher suite operates on a 320-bit state. The state is updated on rounds p^a and p^b. State S consists of two parts, S_r is the r-bit inner part, S_c c-bit outer part as shown in equation 4.4. r is the rate and $c = 320 - r$. State S is formed by five 64-bit words.

$$S = S_r \parallel S_c = x_0 \parallel x_1 \parallel x_2 \parallel x_3 \parallel x_4 \quad (4.4)\ [1]$$

4.1.1 Authenticated encryption

The operation of authenticated encryption and decryption has four stages: initialization, processing associated data, processing plain or ciphertext, and finalization. Mode of operation of ASCON is shown Figure 4.4.

The 320-bit initial state of ASCON is initialized by *IV*, secret key *K* with *k* bits, and nonce *N* with 128 bits as in equation 4.5. *IV* is specified by the variant of the algorithm, it is 64-bit for Ascon-128 and Ascon-128a. In the initialization process, α rounds transformation permutation p^a is applied to the initial state and *XOR*ed with the secret key *K* as shown equation 4.6.

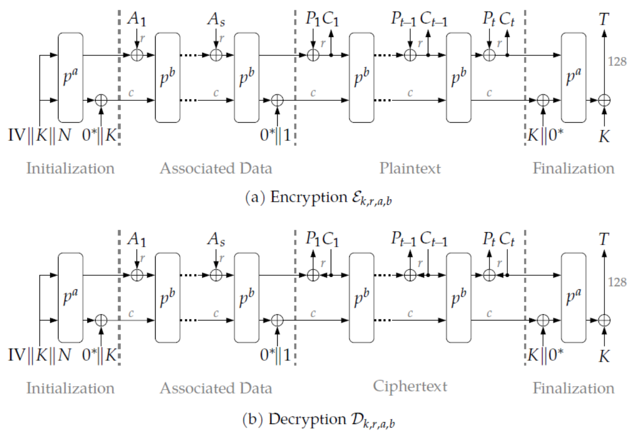


Figure 4.4: Mode of operation of ASCON [1].

$$S \leftarrow IV_{k,r,a,b} \parallel \mathrm{K} \parallel \mathrm{N} \qquad (4.5)\ [1]$$

$$S \leftarrow p^{a}(S) \oplus 0^{320-k} \parallel \mathrm{K} \qquad (4.6)\ [1]$$

After the initialization step, associated data is processed. It is divided into r-bit blocks and each r-bit block is *XOR*ed to the first r bits S_r of state *S*. After this, *b* rounds transformation permutation p^b is applied to S as in equation 4.7. In the end, state *S* is *XOR*ed to 1 as in equation 4.8.

$$S \leftarrow p^{b}((S_r \oplus A_i) \parallel S_c) \qquad (4.7)\ [1]$$

$$S \leftarrow S \oplus (0^{319} \parallel 1) \qquad (4.8)\ [1]$$

After deployment of associated data to the system, the plaintext is divided into r-bit blocks and *XOR*ed to the first r bits S_r of state S, and ciphertext is obtained as in equation 4.9. State S is updated with p^b using *b* rounds as in equation 4.10.

$$C_i \leftarrow S_r \oplus P_i \qquad (4.9)\ [1]$$

$$S \leftarrow p^{b}(C_i \oplus S_c) \qquad (4.10)\ [1]$$

For decryption, the ciphertext is divided into r-bit blocks and *XOR*ed to first the first r bits S_r of state S, and plaintext is obtained as in equation 4.11. The first r bits S_r of state S is replaced by ciphertext and b rounds round transformation applied to the state as in equation 4.12.

$$P_i \leftarrow S_r \oplus C_i \tag{4.11) [1}$$

$$S \leftarrow p^b(C_i \oplus S_c) \tag{4.12) [1}$$

In the finalization step, the secret key K is *XOR*ed to the state, and the state is transformed by p^a as in equation 4.13. The last 128 bits of the state *XOR*ed to the key is the tag T as in equation 4.14. Output of encryption is a tag T with ciphertext C and output of decryption is a tag T with a plaintext P.

$$S \leftarrow p^a(S \oplus (0^r \parallel \mathrm{K} \parallel 0^{c-k})) \tag{4.13) [1}$$

$$T \leftarrow S^{128} \oplus K^{128} \tag{4.14) [1}$$

To summarize all steps, authenticated encryption and decryption algorithms are shown in Figure 4.5.

Authenticated Encryption
$\mathcal{E}_{k,r,a,b}(K, N, A, P)$

Input: key $K \in \{0,1\}^k, k \leq 160$,
nonce $N \in \{0,1\}^{128}$,
associated data $A \in \{0,1\}^*$,
plaintext $P \in \{0,1\}^*$
Output: ciphertext $C \in \{0,1\}^{|P|}$,
tag $T \in \{0,1\}^{128}$

Initialization
$S \leftarrow \mathrm{IV}_{k,r,a,b} \parallel K \parallel N$
$S \leftarrow p^a(S) \oplus (0^{320-k} \parallel K)$
Processing Associated Data
if $|A| > 0$ **then**
$A_1 \ldots A_s \leftarrow r$-bit blocks of $A \| 1 \| 0^*$
for $i = 1, \ldots, s$ **do**
$S \leftarrow p^b((S_r \oplus A_i) \parallel S_c)$
$S \leftarrow S \oplus (0^{319} \parallel 1)$
Processing Plaintext
$P_1 \ldots P_t \leftarrow r$-bit blocks of $P \| 1 \| 0^*$
for $i = 1, \ldots, t-1$ **do**
$S_r \leftarrow S_r \oplus P_i$
$C_i \leftarrow S_r$
$S \leftarrow p^b(S)$
$S_r \leftarrow S_r \oplus P_t$
$\tilde{C}_t \leftarrow \lfloor S_r \rfloor_{|P| \bmod r}$
Finalization
$S \leftarrow p^a(S \oplus (0^r \parallel K \parallel 0^{320-r-k}))$
$T \leftarrow \lceil S \rceil^{128} \oplus \lceil K \rceil^{128}$
return $C_1 \parallel \ldots \parallel C_{t-1} \parallel \tilde{C}_t, T$

Verified Decryption
$\mathcal{D}_{k,r,a,b}(K, N, A, C, T)$

Input: key $K \in \{0,1\}^k, k \leq 160$,
nonce $N \in \{0,1\}^{128}$,
associated data $A \in \{0,1\}^*$,
ciphertext $C \in \{0,1\}^*$,
tag $T \in \{0,1\}^{128}$
Output: plaintext $P \in \{0,1\}^{|C|}$ or $\perp$

Initialization
$S \leftarrow \mathrm{IV}_{k,r,a,b} \parallel K \parallel N$
$S \leftarrow p^a(S) \oplus (0^{320-k} \parallel K)$
Processing Associated Data
if $|A| > 0$ **then**
$A_1 \ldots A_s \leftarrow r$-bit blocks of $A \| 1 \| 0^*$
for $i = 1, \ldots, s$ **do**
$S \leftarrow p^b((S_r \oplus A_i) \parallel S_c)$
$S \leftarrow S \oplus (0^{319} \parallel 1)$
Processing Ciphertext
$C_1 \ldots C_{t-1}\tilde{C}_t \leftarrow r$-bit blocks of $C, 0 \leq |\tilde{C}_t| < r$
for $i = 1, \ldots, t-1$ **do**
$P_i \leftarrow S_r \oplus C_i$
$S \leftarrow C_i \parallel S_c$
$S \leftarrow p^b(S)$
$\tilde{P}_t \leftarrow \lfloor S_r \rfloor_{|\tilde{C}_t|} \oplus \tilde{C}_t$
$S_r \leftarrow S_r \oplus (\tilde{P}_t \parallel 1 \parallel 0^*)$
Finalization
$S \leftarrow p^a(S \oplus (0^r \parallel K \parallel 0^{320-r-k}))$
$T^* \leftarrow \lceil S \rceil^{128} \oplus \lceil K \rceil^{128}$
if $T = T^*$ **return** $P_1 \parallel \ldots \parallel P_{t-1} \parallel \tilde{P}_t$
else return $\perp$

Figure 4.5: Authenticated encryption and decryption of ASCON [1].

4.1.2 Permutation

ASCON applies permutation p^a and p^b to the 320-bit state. The round transformation p consists of three layers, round constant addition p_c, substitution layer p_s, and linear layer p_l as in equation 4.15.

$$p = p_c \parallel p_s \parallel p_l \tag{4.15) [1}$$

- In the **round constant addition** layer, x_2 is *XOR*ed to the round constant c_r as in equation 4.16. Round constants are shown in Table 4.4.

$$x_2 \leftarrow x_2 \oplus c_r \tag{4.16) [1}$$

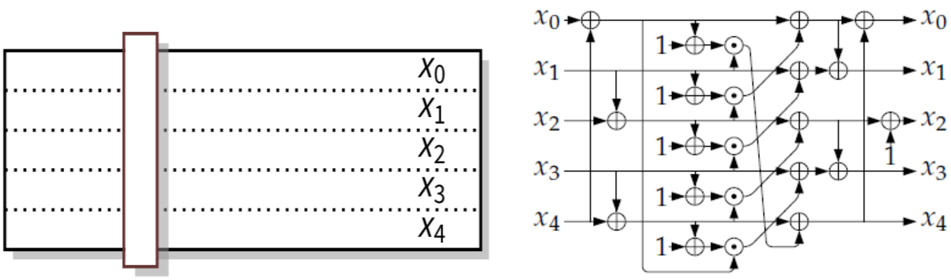
Table 4.4: The round constants for each round in ASCON [1].

p^{12}	p^8	p^6	Constant c_r	p^{12}	p^8	p^6	Constant c_r
0			00000000000000000f0	6	2	0	00000000000000000096
1			00000000000000000e1	7	3	1	00000000000000000087
2			00000000000000000d2	8	4	2	00000000000000000078
3			00000000000000000c3	9	5	3	00000000000000000069
4	0		00000000000000000b4	10	6	4	0000000000000000005a
5	1		00000000000000000a5	11	7	5	0000000000000000004b

- The non-linear **substitution layer** applies a transformation with 5-bit S-Boxes to the 64 parallel applications. The first bits of five 64-bit words x_1, x_2, x_3, x_4, and x_5 are transformed, then the second bits and so on. Lookup table of the 5-bit S-Box is shown in Table 4.5. Operation of S-Box with 5 parallel 64-bits words and representation of the logical operation of ASCON is shown in Fig 4.6. The logical operation of S-Box has *XOR* and *AND* operations.

Table 4.5: Lookup table of 5-bit S-Box of ASCON [1].

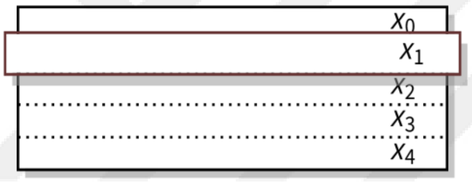
x	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
S(x)	4	b	1f	14	1a	15	9	2	1b	5	8	12	1d	3	6	1c
x	10	11	12	13	14	15	16	17	18	19	1a	1b	1c	1d	1e	1f
S(x)	1e	13	7	e	0	d	11	18	10	c	1	19	16	a	f	17



(a) (b)

Figure 4.6: (a) 5 parallel applications and (b) Logical description of S-Box.

- The **linear layer** operates on each 64-bit word individually by using a 64-bit linear function. The linear function uses shift left and *XOR* operations. Each 64-bit word is rotated with a different number of rotations. Separate 64-bit inputs of linear layer and mathematical definition of the linear function of ASCON is shown in Figure 4.7.



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

(a) (b)

Figure 4.7: (a) 5 separate inputs and (b) Mathematical function of linear layer.

5. PROFILING ASCON

Software implementation of AEAD and hashing algorithm of ASCON cipher suite can be found in the official repository [4]. The source code of the variants of ASCON such as Ascon128 and Ascon128av for 8-bit, 32-bit, and 64-bit systems are shared for public usage. In this project, version 'v1.2.1' of the Ascon-128av variant is profiled. Since RV32I is used in this project, implementation of the algorithm for 32-bit systems is profiled.

The profiling process has the following elements:

- Key with 16-byte
- Associated data with 24-byte
- Plain text with 16-byte
- Nonce with 16-byte
- RV32I base integer instruction set

5.1 Profiling Method

LWC uses highly repeated rounds to provide high security; thus, profiling is carried out by counting the clock cycles and number of instructions of the permutations in the encryption function with different compiler optimization flags. Only the encryption function is analyzed since encryption and decryption have the same logic. *ROUND* macro in the source code is formatted as a normal C function for profiling.

With the *-O0* optimization flag, the compiler makes no optimization on the code and creates a debuggable program [6]. *-O1* optimization flag reduces code size and execution time of the program for small functions. *-O2* makes all supported optimizations that do not involve a space-speed tradeoff. As compared to *-O0*, this option increases both compilation time and the performance of the generated code. *-O3* turns on all optimizations as *-O2* and turns on more optimization flags. *-Os* optimizes for size to reduce the code size.

The counting process is realized by reading the control and status registers of RISC-V which are *cycle* and *instret*. The used CSR pseudo instructions to read are *rdcycle* and *rdinstret*. Usage of instructions with the inline assembly method is shown in Figure 5.1. These functions are placed at the start and the end of program blocks to analyze. The difference between readings at the start and the end minus 1 gives the number of clock cycles and instructions according to the used macro.

```
#define read_cycle(cycle) asm volatile("rdcycle %0" : "=r" (cycle))
#define read_insn(insn)   asm volatile("rdinstret %0" : "=r" (insn))
```

Figure 5.1: Reading count of clock cycles and number of instructions.

CMake [29] is an open-source tool to build, test, and package software and produces make files for the project and supports cross-compilation. CMake is used as the build tool in this project.

To build the program, "CMakelists.txt" file is created and modified as in Figure 5.2. To use the cross-compiler *riscv-unknown-elf-gcc* for a RISC-V target, the RISC-V environment must be set to the installation path of RISC-V tools.

```
cmake_minimum_required(VERSION 2.8.9)

set(CMAKE_SYSTEM_NAME Generic)
set(CMAKE_SYSTEM_PROCESSOR riscv)
set(CMAKE_CROSSCOMPILING 1)
set(CMAKE_C_COMPILER "riscv64-unknown-elf-gcc")
set(CMAKE_CXX_COMPILER "riscv64-unknown-elf-g++")
set(CMAKE_ASM_COMPILER "riscv64-unknown-elf-gcc")

project (ascon)

set(CMAKE_C_FLAGS "-Os -march=rv32i -mabi=ilp32")

include_directories(include)
file(GLOB SOURCES "src/*.c")
add_executable(ascon ${SOURCES})
```

Figure 5.2: CMakelist.txt file for ASCON project.

5.2 Software Implementation of ASCON for 32-bit Processors

For 32-bit processors, ASCON interleaves a 64-bit word to two 32-bit half words, x_e and x_o, by using a bit-interleaving function, f . It stores even-numbered bits in one word and odd-numbered bits in another as shown in equation 5.1. The 320-bit state S consists of five 64-bit words as shown in equation 5.2. For 32-bit processors, even state, s_e, and odd state, s_o, consist of five even words and five odd words as shown in equation 5.3 and equation 5.4, respectively.

$$x_{i_e}, x_{i_o} = f(x_i), \qquad 0 \leq i \leq 4 \tag{5.1}$$

$$S \;=\; (x_0, x_1, x_2, x_3, x_4) \tag{5.2}$$

$$s_e \;=\; (x_{0_e}, x_{1_e}, x_{2_e}, x_{3_e}, x_{4_e}) \tag{5.3}$$

$$s_o \;=\; (x_{0_o}, x_{1_o}, x_{2_o}, x_{3_o}, x_{4_o}) \tag{5.4}$$

where S, s_e, s_o stands for state, even state, odd state, respectively.

In the software implementation of ASCON for 32-bit processors, the even and odd states, s_e and s_o, are arrays with five even and odd words, respectively. For 32-bit systems, S-Box layer operates on each half state, s_e and s_o, separately. The linear layer operates on each element of the array for even and odd states, that is even or odd word.

Round constants are interleaved into even and odd parts as in equation 5.5. Since round constants are known as in Table 4.4, in the software implementation of ASCON, they are used as pre-calculated constants rather than calculating in run time. Operation of the permutation layer with m rounds is shown in Figure 5.3.

$$c_e, c_o \;=\; f(c) \tag{5.5}$$

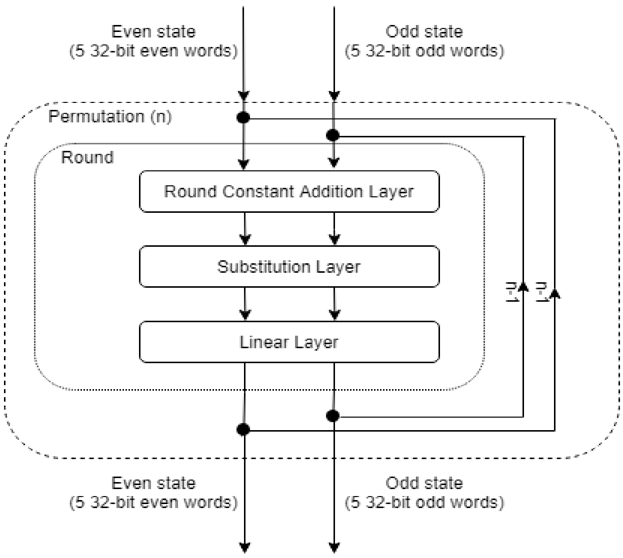


Figure 5.3: Overview of permutation of ASCON in software implementation.

16-byte key, *K*, and 16-byte nonce, *N*, are divided into two parts as in equation 5.6 and equation 5.7, respectively. Each part of key and nonce is interleaved into even and odd words as in equation 5.8 and equation 5.9, respectively.

$$K = (K_0 \,||K_1) \tag{5.6}$$

$$N = (N_0 \,||N_1) \tag{5.7}$$

$$K_{i_e}, K_{i_o} = f(K_i), \qquad 0 \leq i \leq 1 \tag{5.8}$$

$$N_{i_e}, N_{i_o} = f(N_i), \qquad 0 \leq i \leq 1 \tag{5.9}$$

Associated data, *A*, is divided with r-byte into *n* blocks as shown in equation 5.10. The rate *r* is 16-byte for Ascon-128av. Each block, A_i, is divided into two 8-byte parts as in equation 5.11. For implementation 32-bit processor, each 8-byte part is interleaved into even and odd parts, $A_{i_{j_e}}, A_{i_{j_o}}$, as shown in equation 5.12.

$$A = (A_0 \,||A_1 \ldots ||A_{n-1}) \tag{5.10}$$

$$A_i = \left(A_{i_0} \,||\, A_{i_1}\right), 0 \leq i \leq n-1 \tag{5.11}$$

$$A_{i_{j_e}}, A_{i_{j_o}} = f\left(A_{i_j}\right), 0 \leq j \leq 1 \tag{5.12}$$

Plain text, P, is also divided with r-bit into n blocks as shown in equation 5.13. Each block, P_i, is divided into two 8-byte parts as in equation 5.14. For implementation 32-bit processor, each block, P_i, is interleaved into even and odd parts, $P_{i_{j_e}}, P_{i_{j_o}}$, in equation 5.15.

$$P = (P_0 \,||P_1 \ldots ||P_{n-1}) \tag{5.13}$$

$$P_i = \left(P_{i_0} \,||\, P_{i_1}\right), 0 \leq i \leq n-1 \tag{5.14}$$

$$P_{i_{j_e}}, P_{i_{j_o}} = f\left(P_{i_j}\right), 0 \leq j \leq 1 \tag{5.15}$$

The end-to-end AEAD process of ASCON is shown in Figure 5.4 by combining all the equations explained above. Even without profiling, it could be inferred that permutations are the most processed part of the algorithm. As in equation 5.9 and 5.12, plain text and associated data is divided with 16-bytes. For each part, A_i and P_i, permutation with 8 and 12 rounds is applied to the 320-bit state. This leads to rounds taking the most clock cycles of the program.

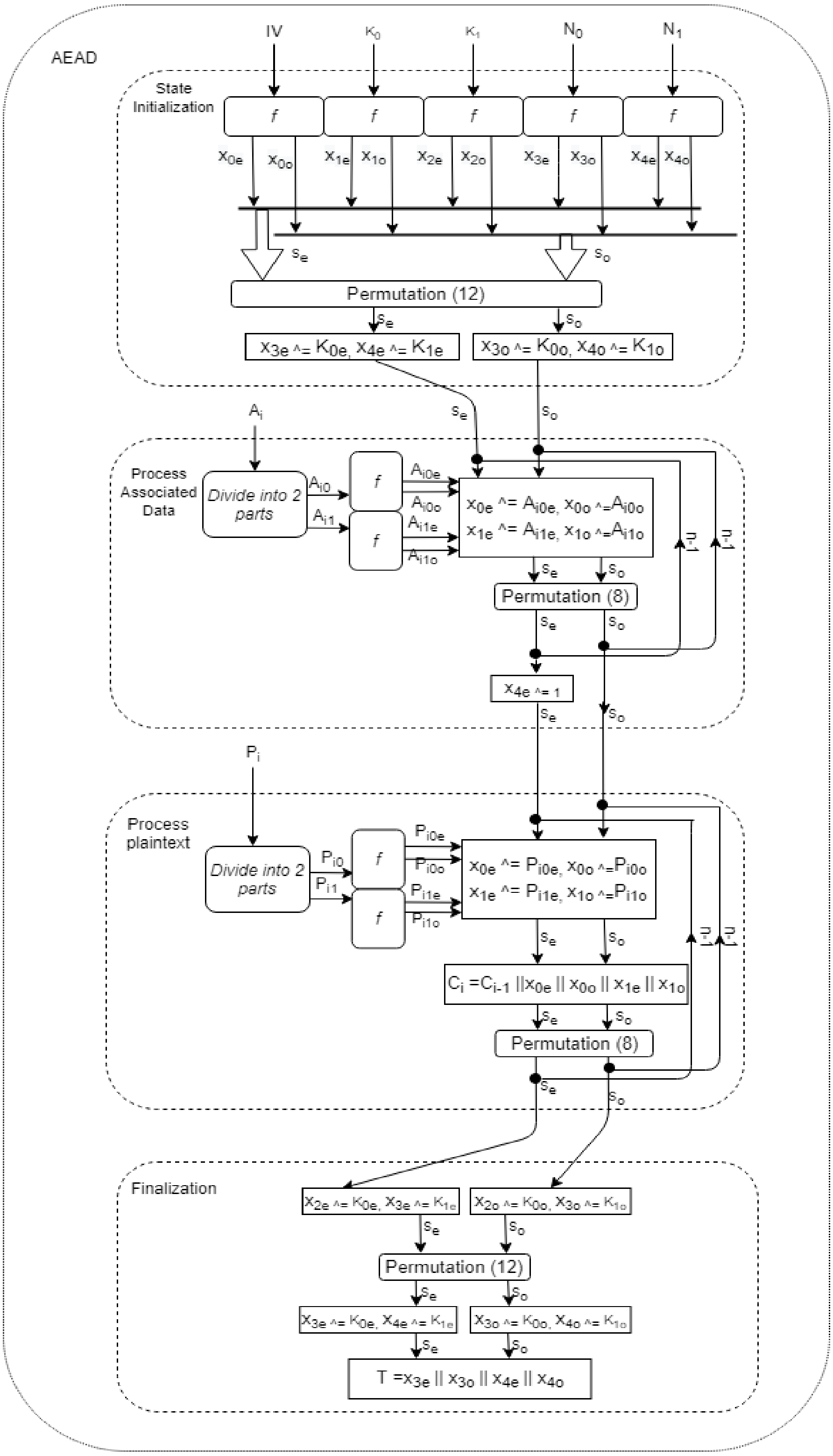


Figure 5.4: Overview of AEAD of ASCON in software implementation.

5.3 Profiling Results

The number of instructions and clock cycles, and code size of the executed encryption function are shown in Table 5.1. Comparing the other optimizations, *-O0* optimization has the greatest number of clock cycles and instructions, and largest code size. There is no significant difference between *-O1*, *-O2*, and *-O3*. *-Os* optimization has a smaller code size. RV32I based processors execute instructions in one clock cycle. Spike also counts all the number of clock cycles per instruction as one. Hence, the number of clock cycles and instructions are the same.

Table 5.1: Compilation results of authentication encryption on RV32I.

Optimization	Number of Instructions	Count of Clocks	Code Size
-Os	29138	29138	19536
-O3	22695	22695	21324
-O2	22694	22694	21336
-O1	22684	22684	20208
-O0	71877	71877	156876

Permutations have a dramatic majority in the algorithm in terms of clock cycles and instructions since they consist of 12 times and 8 times repeated rounds in Ascon-128av. It takes 37.1% of the program to realize permutations for *-O0* optimization level, 47% for *-O1*, *-O2*, and *-O3*, and 54.2% for *-Os* as shown in Table 5.2. Since the number of clock cycles per instruction are one, the instruction count is not shown explicitly.

Table 5.2: Number of clock cycles of permutations and encryption on RV32I.

Optimization	Clock count of Permutations	Total Clock Count	Percentage (%)
-Os	15789	29138	54,2
-O3	10656	22695	47
-O2	10656	22694	47
-O1	10656	22684	47
-O0	26694	71877	37,1

Each round has three layers, round constant addition, substitution layer, and linear layer. The percentage of the clock cycle of the layers in a round is shown in Table 5.3. Compared to round constant addition, the substitution layer and linear layer have the majority of one round for all optimization levels.

Table 5.3: Percentages of clock cycles of the for the layers in a round on RV32I.

Optimization	Round constant addition	Substitution layer	Linear layer
-Os	2,5%	45,2%	52,3%
-O3	2,5%	45,2%	52,3%
-O2	2,5%	45,2%	52,3%
-O1	2,5%	45,2%	52,3%
-O0	3,6%	48,7%	47,7%

5.4 RISC-V Custom Instructions for ASCON

A round operates on an unsigned 320-bit state with five 64-bit applications. Since integer registers are 32-bit in RV32, 32-bit implementation of ASCON interleaves a 64-bit application to two unsigned 32-bit applications as explained above.

Round constant addition is an *XOR* operation and RV32I has an instruction for it. Thus, extension for this operation does not affect the performance of execution of the algorithm. However, extensions for substitution and linear layer operations are a good choice to reduce the total number of clock cycles and instructions of the program.

5.4.1 S-Box

In the software implementation of ASCON, even and odd states are stored in arrays with even and odd words. Thus, one of the input operands of the S-Box instruction holds the address of the first element of the array. The S-Box instruction loads all the elements of the array which are contiguous five 32-bit words and makes substitution computations as shown in Figure 5.5. The instruction for the S-Box layer then stores the five 32-bit to their initial addresses. The output of the instruction is hard-wired to zero.

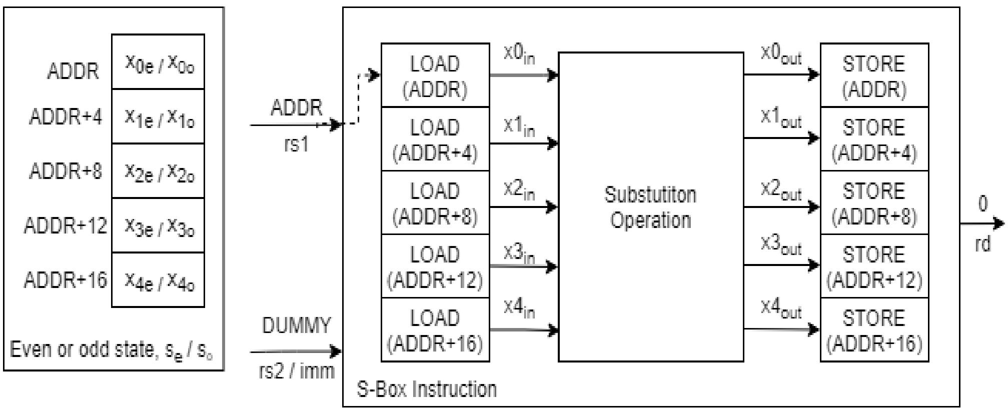


Figure 5.5: Overview of the S-Box instruction (ADDR=$ADDR_e$ or $ADDR_o$).

RISC-V has been extended with two types of instructions, R-type, and I-type in Figure 2.3. The first input operand of the S-Box instruction, *rs1*, is a register operand that holds an address. The second input operand, *rs2*, is a dummy operand and not used in any operation inside the instruction. It could be given directly as integers, as rvalue, in the source code. So, it could also be an *immediate* operand. Thus RISC-V has been extended for the S-Box layer with two versions of the instruction, R-type register-register and I-type register-immediate instruction. The output operand, *rd*, is hard-wired to zero. To follow the naming convention of RISC-V, extended instructions are called *sbox* for R-type and *sboxi* for I-type instruction.

5.4.2 Cyclic rotation

Since RV32I has no instruction for bit rotation operations, it can be extended with cyclic rotation instruction. The first input operand of the instruction, *rs1*, is a 32-bit even or odd word, x_e or x_o. The second input operand of the instruction, *rs2* or *immediate*, is the number of rotations. Rotated version of the word is written to the output operand of the instruction, *rd*. Overview of the cyclic rotation instruction is shown in Figure 5.6.

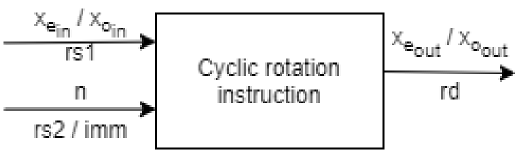


Figure 5.6: Overview of the cyclic rotation instruction.

In the software implementation of the linear layer of ASCON, each element of the array for even and odd states, that is even, or odd word is given as the first input operand of the cyclic rotation instruction. Since the number of rotations could be an rvalue in software implementation, two versions of cyclic rotation have been created as S-Box instruction. Instructions are called *rot* for R-type and *roti* for I-type instruction.

6. MODIFICATION OF RISC-V GNU TOOLCHAIN FOR ASCON CUSTOM INSTRUCTIONS

RISC-V ISA supports extended custom instructions and these instructions must be recognizable by the compiler. Compilers must be able to generate these kinds of instructions and create executable files. To realize this recognition, the toolchain is modified and reinstalled. There are two techniques to modify the compiler. The first technique is adding opcodes and some other information about the custom instruction to GCC Binutils and generating the custom instruction with an inline assembly method. The second option which also needs the first option modifications is adding a built-in function to GCC backend which will be compiled to a specific machine instruction, custom instruction in this case, automatically. Overview of these two techniques is shown in Figure 6.1. In this section, how to customize the toolchain for the custom instructions is explained and source codes of customized GCC can be found in the author's GitHub page[33] for public usage.

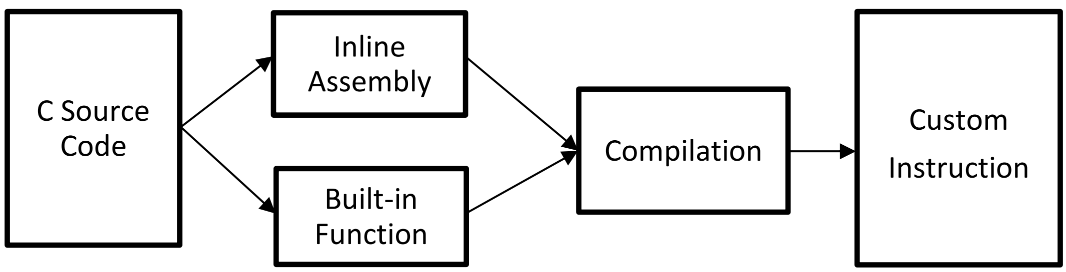


Figure 6.1: Overview of generating custom instructions.

6.1 Adding ASCON Custom Instructions to GCC Binutils

Modification of GCC Binutils for custom instructions can be achieved with modifying “riscv-binutils” source code under “freedom-tools”. The files which must be modified are “riscv-opc.h” under “riscv-binutils/include/opcode/” directory and “riscv-opc.c” under "riscv-binutils/opcodes/" directory as shown in Figure 6.2. By applying these modifications, assembler of GCC Binutils can assemble the custom instructions.

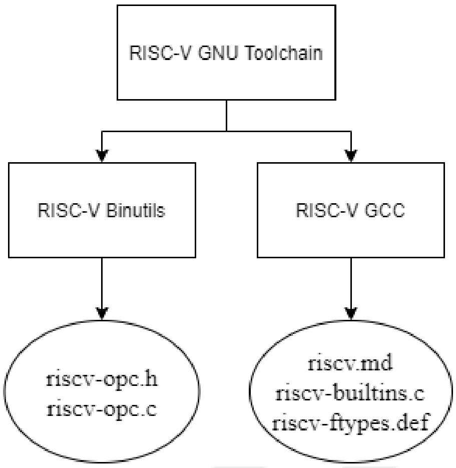


Figure 6.2: Modified files in RISC-V GNU Toolchain.

MATCH and *MASK* definitions and *DECLARE_INSN* macro of the new instructions are added to riscv-opc.h file. *MATCH* defines the numeric representation of instruction with zeros for registers and fills the *funct3* and *funct7* parts of an instruction in Figure 2.3. *MASK* is used to calculate *MATCH* value of the instruction. *MASK* and *MATCH* definitions of the custom instructions are shown in Figure 6.3.

The *DECLARE_INSN* macro takes the name, match, and mask values of the instruction. 4 new custom instructions which are *rot*, *roti*, *sbox* and *sboxi* were added with corresponding match and mask values as shown Figure 6.4

```
#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
```

Figure 6.3: Opcode of mask and match of new instructions in riscv-opc.h file.

```
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 6.4: New instruction declaration in riscv-opc.h file.

There is a *riscv_opcode* struct which holds information about a RISC-V instruction in "riscv.h" file under "include/opcode" directory. Every instruction has a *name*, *xlen requirement*, *instruction class*, *instruction arguments*, *match*, *mask opcode*, *match function* and description of instruction (*pinfo*). *riscv_opcode* structure is shown in Figure 6.5.

```
struct riscv_opcode
{
  /* The name of the instruction.  */
  const char *name;
  /* The requirement of xlen for the instruction, 0 if no requirement.  */
  unsigned xlen_requirement;
  /* Class to which this instruction belongs.  Used to decide whether or
     not this instruction is legal in the current -march context.  */
  enum riscv_insn_class insn_class;
  /* A string describing the arguments for this instruction.  */
  const char *args;
  /* The basic opcode for the instruction.  When assembling, this
     opcode is modified by the arguments to produce the actual opcode
     that is used.  If pinfo is INSN_MACRO, then this is 0.  */
  insn_t match;
  /* If pinfo is not INSN_MACRO, then this is a bit mask for the
     relevant portions of the opcode when disassembling.  If the
     actual opcode anded with the match field equals the opcode field,
     then we have found the correct instruction.  If pinfo is
     INSN_MACRO, then this field is the macro identifier.  */
  insn_t mask;
  /* A function to determine if a word corresponds to this instruction.
     Usually, this computes ((word & mask) == match).  If the constraints
     checking is disable, then most of the function should check only the
     basic encoding for the instruction.  */
  int (*match_func) (const struct riscv_opcode *op, insn_t word,
                     int constraints, const char **error);
  /* For a macro, this is INSN_MACRO.  Otherwise, it is a collection
     of bits describing the instruction, notably any relevant hazard
     information.  */
  unsigned long pinfo;
};
```

Figure 6.5: *riscv_opcode* structure in riscv.h file.

Information of new instructions are added to *riscv_opcode* struct array that holds the information about all RISC-V instructions in "riscv-opc.c" file. *Name* of the custom instructions in this project are *rot*, *roti*, *sbox* and *sboxi*. There is no *xlen requirement* for these instructions. They all are in base integer subset, so instruction class is *INSN_CLASS_I*. For R-type instructions, they have 3 arguments *d* for destination register, *s* for source register and *t* for target register. For I-type instructions, they have *j* for immediate register instead of target register *t*. Modification of "riscv-opc.c" file is shown in Figure 6.6.

```
{"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 6.6: Adding new instructions to riscv-opc.c file.

With the modifications above, the custom instructions can be generated by using inline assembly. Usage of generation of *rot* instruction with inline assembly is shown Figure 6.7.

```
int main()
{
  int result, a, b;

  asm volatile("rot %[result], %[value1], %[value2]\n\t":
      [result] "=r" (result) : [value1] "r" (a), [value2] "r" (b));
}
```

Figure 6.7: Usage of inline assembly for the *rot* instruction in source code.

It can be verified that *rot* instruction is generated by getting the assembly code of the executable. Content of this assembly file and *rot* instruction are shown in Figure 6.8.

```
0001018c <main>:
  1018c: fe010113          addi  sp,sp,-32
  10190: 00812e23          sw   s0,28(sp)
  10194: 02010413          addi  s0,sp,32
  10198: fec42783          lw   a5,-20(s0)
  1019c: fe842703          lw   a4,-24(s0)
  101a0: 30e787b3          rot a5,a5,a4
  101a4: fef42223          sw   a5,-28(s0)
  101a8: 00000793          addi  a5,zero,0
  101ac: 00078513          addi  a0,a5,0
  101b0: 01c12403          lw   s0,28(sp)
  101b4: 02010113          addi  sp,sp,32
  101b8: 00008067          jalr  zero,0(ra)
```

Figure 6.8: Generation of the *rot* instruction with inline assembly.

6.2 Adding Instruction Patterns of ASCON Custom Instructions to GCC Back end

To generate custom instructions with GCC built-ins, instruction patterns of the instructions and information about the new built-ins are added to GCC RISC-V back end as shown in Figure 6.2

6.2.1 Instruction patterns of RISC-V custom instructions

Instruction patterns of *riscv_rot* and *riscv_sbox* are added to “riscv.md” file as shown in Figure 6.9 and Figure 6.10. In spite the name of the instruction pattern is optional as stated in Section 2.2.1.1, to associate a built-in function with an instruction pattern, it must be named with *riscv_* prefix.

(unspec_volatile [operands ...] index) RTL expression is used to define a volatile pattern. Since custom instructions in this project are a machine specific instruction, its operation with its operands is machine specific. These kinds of instructions are considered as *volatile instructions* because they cannot be described with a functionality such as summation, division, etc. Using *unspec_volatile* expression leads the compiler to see the pattern as a black box and it does not make optimization. It knows only predicates and register constraints about the RTL representation of the instruction.

The instruction patterns have machine mode *SI*, *register_operand* and *arith_operand* as predicates and, *r* and *I* as constraints. Output of instruction pattern is a custom instruction which is *rot*, *roti*, *sbox* or *sboxi*.

UNSPECV_ROT and *UNSPECV_SBOX* in Figure 6.11 are the indexes and added to enumeration *unspecv* in “riscv.md”.

```
(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"
)
```

Figure 6.9: Instruction pattern of *riscv_rot*.

```
(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"
)
```

Figure 6.10: Instruction pattern of *riscv_sbox*.

```
(define_c_enum "unspecv" [

  UNSPECV_ROT
  UNSPECV_SBOX
```

Figure 6.11: Indexes of *riscv_rot* and *riscv_sbox* instruction patterns.

6.2.2 Creation of new built-in functions for RISC-V custom instructions

To create built-in functions in the GCC RISC-V back end, "riscv-builtins.c" and "riscv-ftypes.def" files needed to be modified under "src/riscv/gcc/config/riscv/" directory as shown in Figure 6.2

- Since custom instructions have 2 input arguments, built-in functions also have 2 two input arguments. The new macro is added as *RISCV_FTYPE_NAME2(A, B, C)* and *C* represents the second input argument.

```
#define RISCV_FTYPE_NAME2(A, B, C) RISCV_##A##_FTYPE_##B##_##C
```

Figure 6.12: Adding a built-in function type to riscv-builtins.c file.

- The first input operand of *sbox* and *sboxi* are an address which holds an unsigned 32-bit data, so it corresponds to an integer pointer as the first argument of the built-in function. The second input operand of S-Box custom instructions and all input operands of rotation instruction are unsigned 32-bit data. The integer pointer and unsigned 32-bit integer argument type codes for input arguments of built-in functions are defined as in Figure 6.13.

```
#define RISCV_ATYPE_USI unsigned_intSI_type_node
#define RISCV_ATYPE_INTPTR integer_ptr_type_node
```

Figure 6.13: Argument types in riscv-builtins.c file.

- After the argument types are defined, *RISCV_FTYPE_ATYPES2(A, B, C)* is added as in Figure 6.14.

```
#define RISCV_FTYPE_ATYPES2(A, B, C) \
  RISCV_ATYPE_##A, RISCV_ATYPE_##B, RISCV_ATYPE_##C
```

Figure 6.14: Argument types of function prototypes in riscv-builtins.c file.

- Function types for the built-in functions of custom instruction are shown in Figure 6.15. *DEF_RISCV_FTYPE(2, (USI, USI, USI))* corresponds to the built-in function of rotation instructions, *rot* and *roti*. It has two inputs as an unsigned 32-bit integer and returns an unsigned 32-bit integer.
- *DEF_RISCV_FTYPE(2, (USI, INTPTR, USI))* corresponds to the built-in function of S-Box instructions, *sbox* and *sboxi*. It has two inputs. First one is an integer pointer and the second one is an unsigned 32-bit integer and it returns an unsigned 32-bit integer.

```
DEF_RISCV_FTYPE (2, (USI, USI, USI))
DEF_RISCV_FTYPE (2, (USI, INTPTR, USI))
```

Figure 6.15: Function types in riscv-ftypes.def file.

- After all the modifications above, the last step is to add information of built-in functions to the *riscv_builtins* array which consist of the *riscv_builtin_description* struct for each RISC-V built-in function in the “riscv-builtins.c” file.
- Construction of the built-in functions which correspond to specific custom instructions are shown in Figure 6.16. They are added to *riscv_builtins* array with *DIRECT_BUILTIN* macro since return type is not void. The name of each instruction pattern is given as a parameter to the *INSN* argument without *riscv__* prefix.
- Function type is given with *RISCV_USI_FTYPE_USI_USI* for rotation instructions and *RISCV_USI_FTYPE_INTPTR_USI* for sbox instructions.
- *rv_base* is the availability predicate of the new built-in functions as shown in Figure 6.17. It always returns *true* meaning that *rot* and *sbox* built-in functions available for all RISC-V variants.

```
DIRECT_BUILTIN(rot,  RISCV_USI_FTYPE_USI_USI,    rv_base),
DIRECT_BUILTIN(sbox, RISCV_USI_FTYPE_INTPTR_USI, rv_base),
```

Figure 6.16: Adding new built-in functions to riscv-builtins.c file.

```
AVAIL (rv_base, 1)
```

Figure 6.17: Availability predicate of built-in functions.

With completing all the modifications in this section, GCC is reinstalled. *__builtin_riscv_* prefix is automatically added to the function name by the compiler. A simple code with built-in functions for the custom instructions is shown in Figure 6.18.

```
int main()
{
  uint32_t num, rot, rotated_num;
  rotated_num = __builtin_riscv_rot(num, rot);

  uint32_t ret, state[5], dummy;
  ret = __builtin_riscv_sbox(state, dummy);
}
```

Figure 6.18: Usage of new built-in functions in source code.

Custom instruction *sbox* from *__builtin_riscv_sbox* and *rot* from *__builtin_riscv_rot* is generated by compiling the code above. Assembly code of the *main* function is shown in Figure 6.19. Since the second input arguments are not given directly as integers meaning that it is not an *rvalue*, R-type instructions are generated from the corresponding instruction pattern of built-in.

```
0001018c <main>:
   1018c: fc010113          addi  sp,sp,-64
   10190: 02812e23          sw  s0,60(sp)
   10194: 04010413          addi  s0,sp,64
   10198: fec42783          lw  a5,-20(s0)
   1019c: fe842703          lw  a4,-24(s0)
   101a0: 30e787b3          rot a5,a5,a4
   101a4: fef42223          sw  a5,-28(s0)
   101a8: fc840793          addi  a5,s0,-56
   101ac: fe042703          lw  a4,-32(s0)
   101b0: 60e787b3          sbox  a5,a5,a4
   101b4: fcf42e23          sw  a5,-36(s0)
   101b8: 00000793          addi  a5,zero,0
   101bc: 00078513          addi  a0,a5,0
   101c0: 03c12403          lw  s0,60(sp)
   101c4: 04010113          addi  sp,sp,64
   101c8: 00008067          jalr  zero,0(ra)
```

Figure 6.19: Generation of custom instruction *rot* and *sbox* with built-in functions.

7. MODIFICATION OF SPIKE RISC-V ISA SIMULATOR FOR ASCON CUSTOM INSTRUCTIONS

The new custom instructions added to GCC cannot be executed by Spike. To realize this execution, Spike must be customized for these new custom instructions and clock cycles of them. The source codes of modified Spike can be found on the author's GitHub page[33].

7.1 Adding ASCON Custom Instructions to Spike

In Figure 7.1, the files which are modified are shown. After modification of these files, descriptions of functional behavior of these instructions are added to "/riscv/insns/<new_instruction_name.h>" for a single instruction. After this procedure, Spike must be reinstalled.

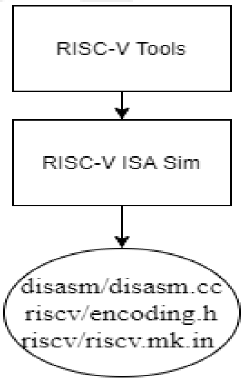


Figure 7.1: Modified files in Spike ISA Simulator for the new instructions.

- New instructions are defined with *DEFINE_RTYPE(<new_instruction_name>)* for R-type instructions and *DEFINE_ITYPE(<new_instruction_name>)* for I-type instructions. Defining custom instructions in "disasm/disasm.cc" is shown in Figure 7.2.

```
DEFINE_RTYPE(rot);
DEFINE_ITYPE(roti);
DEFINE_RTYPE(sbox);
DEFINE_ITYPE(sboxi);
```

Figure 7.2: Adding new instructions to disasm.cc file.

- Mask opcode, match opcode and declaration of instructions are added to the "/riscv/encoding.h" file as in Figure 7.3 and 7.4.

```
#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
```

Figure 7.3: Opcode mask and opcode match of new instructions in encoding.h file.

```
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 7.4: New instruction declaration in encoding.h file.

- Since custom instructions are RV32I, they are added to *riscv_insn_ext_i* list in "riscv/riscv.mk.in" as in Figure 7.5. Purpose of this modification is to generate source code of functional behavior for each custom instruction using its header file which will be created in the next step.

```
rot \
roti \
sbox \
sboxi \
```

Figure 7.5: Adding new instructions to riscv.mk.in file.

- After modification of the necessary files above, the next step is to define functional behaviors of the new instructions. "rot.h" and "roti.h" header files for rotation custom instructions are created under the "/riscv/insns/" directory as shown in Figure 7.6 and Figure 7.7. *RS1* and *RS2* are the registers which read the two source registers of *rot* instruction. *WRITE_RD* macro writes result to *rd*, the destination register of the instruction. Since *roti* is a register-immediate instruction, *insn.imm()* used to read the immediate part of the instruction as Figure 7.7.

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

Figure 7.6: Functional description of *rot* instruction in rot.h file.

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

Figure 7.7: Functional description of *roti* instruction roti.h file.

The functional behaviour of S-Box instructions is shown in Figure 7.8. Since the second operands which are *rs2* for *sbox* and *immediate* for *sboxi* have not any functionality, functional descriptions for two instructions are the same. *MMU.load_uint32(<address>)* is the functional behaviour of *load* instruction. It loads unsigned 32-bit data from <*address*> to a register. In operation of S-Box instructions, it gets the start address of state from source register *rs1* and loads elements of state which are even or odd words from contiguous 32-bit memory five times to registers. S-Box instruction then makes substition operation on the registers. After this operation, the instruction stores the processed words to the memory which is the same address in *load* operation for each word. The functional description of *store* instruction is *MMU.store_uint32(<address>, <data>)* that stores <*data*> to <*address*>. Finally, S-Box instructions write 0 to the destination register, *rd*.

```
//Load Even/Odd State
reg_t x0 = MMU.load_uint32(RS1);
reg_t x1 = MMU.load_uint32(RS1+4);
reg_t x2 = MMU.load_uint32(RS1+8);
reg_t x3 = MMU.load_uint32(RS1+12);
reg_t x4 = MMU.load_uint32(RS1+16);
reg_t t0, t1, t2, t3, t4;
//S-Box Operation
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 Even/Odd State
MMU.store_uint32(RS1,   x0);
MMU.store_uint32(RS1+4, x1);
MMU.store_uint32(RS1+8, x2);
MMU.store_uint32(RS1+12,x3);
MMU.store_uint32(RS1+16,x4);
WRITE_RD(0);
```

Figure 7.8: Functional description of *sbox* and *sboxi* instructions in sbox.h and sboxi.h file.

7.2 Integrating Clock Cycles of Custom Instructions into Spike

How many clock cycles it takes to execute an instruction depends on the micro-architecture. Spike counts clock cycles as 1 per instruction since it is not a cyclic accurate simulator. However, S-Box instruction operates with 5 cycles load from memory, 5 cycles store to memory, and 1 cycle substitution operation, 11 clock cycles in total. To see the effect of S-Box instruction on performance of the program, an algorithm is integrated into Spike to update the *CSR cycle* correctly.

The instructions which do not take one clock cycle are stored in the database with corresponding clock cycles. After an instruction is fetched from the pc, it is disassembled to get the name of the instruction. The instruction is searched in the database, if it is found, the corresponding clock cycle is taken from the database, else it is assumed as one and the obtained clock cycle added to the *CSR cycle*. The flow chart of the integrated algorithm for counting clock cycles is shown in Figure 7.9.

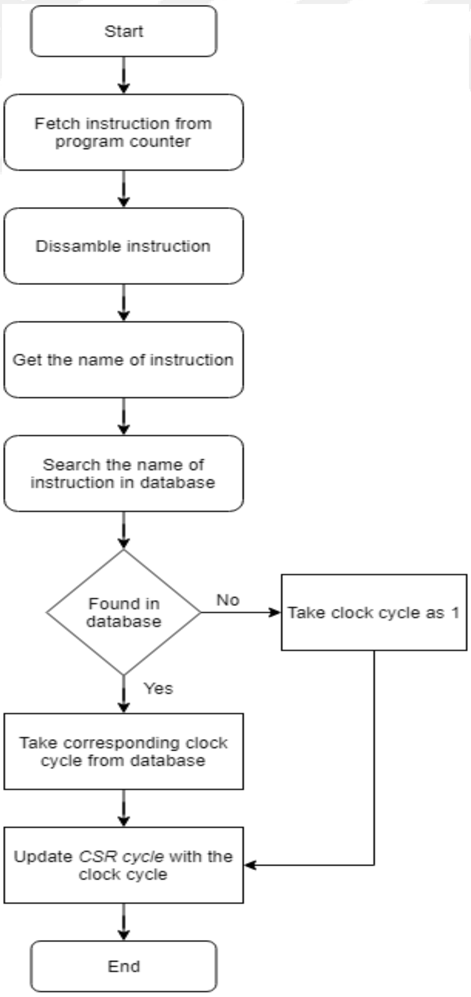


Figure 7.9: Flow chart of the integrated algorithm to Spike.

To realize this algorithm, Spike is modified with clock cycles of S-Box instructions. “execute.cc”, “processor.cc” and “processor.h” files under the “riscv-isa-sim/riscv” directory are modified. Modified files are shown in Figure 7.10.

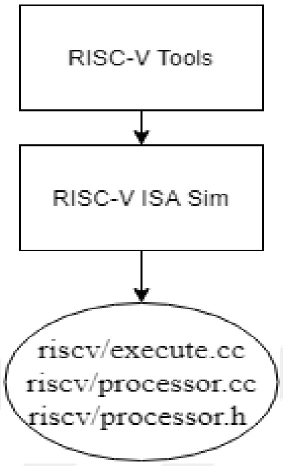


Figure 7.10: Modified files for clock cycles in Spike.

Clock cycles are read by getting the content of the *CSR cycle*. However, simulating this on Spike returns content of *CSR instret* instead of *cycle*. Thus, clock cycle and instruction counts are the same. To count clock cycles, cycle information is added to the state of the processor by introducing a new member *reg_t mcycle* to *struct state_t* in “processor.h” file. This new member is set to 0 in the *reset* method of *state*_t in “processor.cc” file as in Figure 7.11.

(a) (b)

Figure 7.11: (a) Adding *mcycle* to state of the processor. (b) Resetting *mcycle*.

A map data structure is defined to hold the clock cycles of instructions which do not take 1 clock cycle to execute. It maps the name of the instruction to the clock cycle of it. The map consists of <*instruction name, clock cycle*> pairs in "clock_cycles_map.h" file under "riscv-isa-sim/riscv" directory as in Figure 7.12. Since S-Box instructions, *sbox* and *sboxi*, take 11 clocks in total, these instructions are added to *clock_cycles* map. Rotation instruction is assumed as 1 clock cycle and not defined explicitly in data structure of clock cycles.

```
#pragma once

#include <map>
#include <string>

const std::map <std::string, uint64_t> clock_cycles = {
{"sbox", 11},
{"sboxi",11}
};
```

Figure 7.12: Data structure of clock cycles in clock_cycles_map.h file.

get_clock_cycle method is declared in *processor_t* class in the "processor.h" file to find the clock cycle of an instruction. The method basically takes instruction as the input argument. Inside the method, dissembling and getting names of the instruction operations are made. Instruction name is then searched in the *clock_cycles* map. If it is not found, method returns clock cycle as 1, else it returns the corresponding clock cycle. Function body of *get_clock_cycle* in "execute.cc file" is shown in Figure 7.13

```
#include "clock_cycles_map.h"

uint64_t processor_t::get_clock_cycle(const insn_t& insn)
{
  const disasm_insn_t* disasm_insn = disassembler->lookup(insn);
  const auto it = clock_cycles.find(disasm_insn->get_name());
  if(clock_cycles.end() == it)
  {
    return 1;
  }
  return it->second;
}
```

Figure 7.13: Function body of *get_clock_cycle* in execute.cc file.

Reading from the *CSR cycle* must return *mcycle*. To realize this operation, the *get_csr* method of *processor_*t class in the "processor.cc" file is modified as in Figure 7.14. This method takes a *CSR* and returns the data stored in the *CSR*.

```
switch (which)
{
  case CSR_FFLAGS:
    require_fp;
    if (!supports_extension('F'))
      break;
    ret(state.fflags);
  case CSR_FRM:
    require_fp;
    if (!supports_extension('F'))
      break;
    ret(state.frm);
  case CSR_FCSR:
    require_fp;
    if (!supports_extension('F'))
      break;
    ret((state.fflags << FSR_AEXC_SHIFT) | (state.frm << FSR_RD_SHIFT));
  case CSR_VCSR:
    require_vector_vs;
    if (!supports_extension('V'))
      break;
    ret((VU.vxsat << VCSR_VXSAT_SHIFT) | (VU.vxrm << VCSR_VXRM_SHIFT));
  case CSR_CYCLE:
  {
    if (!ctr_ok)
      goto throw_illegal;
    if (!ctr_v_ok)
      goto throw_virtual;
    ret(state.mcycle);
  }
```

Figure 7.14: Modifying *CSR cycle*.

In the *step* method of *processor*_t in the "execute.cc", instructions are fetched, decoded, and executed. After each execution of the fetched instruction, *CSR instret* is updated to hold the number of instructions. Clock cycles are calculated for the corresponding instruction by calling *get_clock_cycle* function after each execution and *mcycle* is updated with returned clock cycle (see Appendix A for the *step* function). With these modifications, Spike is now capable of counting clock cycles of S-Box instructions correctly.

8. RESULTS

8.1 Integration of Built-In Functions into Software Implementation of ASCON

The *ROUND* function in the software implementation of ASCON takes round constant as even and odd words and address of the even and odd state as shown in Figure 8.1. The even and odd states are arrays which store the even and odd words. To provide a more readable program, names of built-ins which are *__builtin_riscv_sbox* and *__builtin_riscv_rot* are defined with shorter words, *sbox* and *rot*, at the top of the code.

In the round constant addition layer, interleaved even and odd parts of x_2 which correspond to the third element of the even and odd state are *XOR*ed to the even and odd round constant separately.

In the S-Box layer, the built-in function is called twice, one for even state and one for odd state as in Figure 8.1. The return value of the built-in function for S-Box operation is omitted since it is not used in any operation. The first input parameter of each usage is the address of the corresponding state that is an even or odd state. The second input parameter is a dummy value and chosen as 1.

In the linear layer, the built-in function for rotation operation is called as shown in Figure 8.1. The first input parameter of the built-in even or odd word that is an element of the even or odd state. The second input parameter of the built-in is the rotation number. The return value of rotation is the rotated version of the first input parameter.

The second input operand of both built-ins is given as rvalue in the software; thus, I-type of instructions are generated in the assembly code. I-type or R-type versions of an instruction affects the total clock cycle of a program. By using rvalue in software implementation leads generation of I-type instruction. Since, I-type instruction has *immediate* operand, the value in it is known in the compile time; thus, it is not *loaded* to a register in run-time.

It can be inferred from Figure 8.2 that I-type *roti* instruction has 1 *load* instruction *(lw)* missing. Thus, I-type instructions decrease the total clock cycle and instruction count of a program.

```c
#define sbox  __builtin_riscv_sbox
#define rot   __builtin_riscv_rot
typedef unsigned int u32;
void ROUND(u32 C_e, u32 C_o, u32* s_e, u32* s_o)
{
  u32 temp0_e, temp1_e, temp2_e, temp3_e, temp4_e;
  u32 temp0_o, temp1_o, temp2_o, temp3_o, temp4_o;
  /* round constant */
  s_e[2] ^= C_e;                     s_o[2] ^= C_o;
  /* s-box layer */
  sbox(s_e, 1);
  sbox(s_o, 1);
  /*linear layer */
  temp0_e  = s_e[0] ^ rot(s_o[0], 4);   temp0_o  = s_o[0] ^ rot(s_e[0], 5);
  temp1_e  = s_e[1] ^ rot(s_e[1], 11);  temp1_o  = s_o[1] ^ rot(s_o[1], 11);
  temp2_e  = s_e[2] ^ rot(s_o[2], 2);   temp2_o  = s_o[2] ^ rot(s_e[2], 3);
  temp3_e  = s_e[3] ^ rot(s_o[3], 3);   temp3_o  = s_o[3] ^ rot(s_e[3], 4);
  temp4_e  = s_e[4] ^ rot(s_e[4], 17);  temp4_o  = s_o[4] ^ rot(s_o[4], 17);
  s_e[0] ^=  rot(temp0_o, 9);           s_o[0] ^= rot(temp0_e, 10);
  s_e[1] ^=  rot(temp1_o, 19);          s_o[1] ^= rot(temp1_e, 20);
  s_e[2] ^= temp2_o;                    s_o[2] ^= rot(temp2_e, 1);
  s_e[3] ^= rot(temp3_e, 5);            s_o[3] ^= rot(temp3_o, 5);
  s_e[4] ^= rot(temp4_o, 3);            s_o[4] ^= rot(temp4_e, 4);
  s_e[2] = ~s_e[2];                   s_o[2] = ~s_o[2];
}
```

Figure 8.1: Usage of the new built-ins in *ROUND* function of ASCON.

```c
int main()
{
uint32_t num,  rotated_num;
uint32_t rot = 5;
// 5 is rvalue
rotated_num = __builtin_riscv_rot(num, 5 );
// rot is lvalue
rotated_num = __builtin_riscv_rot(num, rot );
}
```

```
0001018c <main>:
  1018c: fe010113        addi  sp,sp,-32
  10190: 00812e23        sw  s0,28(sp)
  10194: 02010413        addi  s0,sp,32
  10198: 00500793        addi  a5,zero,5
  1019c: fef42623        sw  a5,-20(s0)
  101a0: fe842783        lw  a5,-24(s0)
  101a4: 505787b3        roti  a5,a5,1285
  101a8: fef42223        sw  a5,-28(s0)
  101ac: fe842783        lw  a5,-24(s0)
  101b0: fec42703        lw  a4,-20(s0)
  101b4: 30e787b3        rot a5,a5,a4
  101b8: fef42223        sw  a5,-28(s0)
  101bc: 00000793        addi  a5,zero,0
  101c0: 00078513        addi  a0,a5,0
  101c4: 01c12403        lw  s0,28(sp)
  101c8: 02010113        addi  sp,sp,32
  101cc: 00008067        jalr  zero,0(ra)
```

Figure 8.2: The difference between I-type and R-type of rotation instruction.

8.2 Improvement Results

After extension of RISC-V tools, the built-ins for the custom instructions for ASCON are integrated into software implementation of algorithm as in Figure 8.1. The profiling is made with the extended RISC-V GNU Toolchain and Spike.

As shown in Table 8.1, the number of clocks and instructions in the linear layer of ASCON is decreased 34,5% with *-Os*, *-O3*, *-O2* and *-O1* optimization levels and 16,9% with -O0 optimization level by rotation instruction, *roti*. Since rotation instruction takes 1 clock cycle to execute, the number of instructions and clocks are the same.

Table 8.1: Number of clock cycles and instructions in linear layer with/without new instructions.

Optimization	# of Clocks and # of Instructions		
	No new instruction	New Instruction	Reduction (%)
-Os	110	72	34,5
-O3	110	72	34,5
-O2	110	72	34,5
-O1	110	72	34,5
-O0	224	186	16,9

Number of clocks in the substitution layer of ASCON is decreased 75,5% with *-Os*, 77% with *-O3*, *-O2* and *-O1*, and 89% with -O0 optimization level by S-Box instruction, *sboxi*. Since S-Box instruction takes 11 clock cycles to execute, the number of instructions and clocks are not the same and shown explicitly. Reduction of the number of instructions in the substitution layer of ASCON is more than 95% for all optimization levels as shown in Table 8.2.

Table 8.2: Number of clock cycles and instructions in the substitution layer with/without new instructions.

Optimization	No new instruction	# of Clocks		# of Instructions	
		New Instruction	Reduction (%)	New Instruction	Reduction (%)
-Os	98	24	75,5	4	95,9
-O3	96	22	77	2	97,9
-O2	96	22	77	2	97,9
-O1	96	22	77	2	97,9
-O0	229	25	89	5	97,8

Profiling results also show that the performance of the *ROUND* function in the ASCON can be increased at least 50% in the worst-case considering the optimization levels. Number of clock cycles of a round is reduced with new instructions and execution of the function is accelerated 51,7% with *-O0* and 52,3% with the others, *-O1, -O2, -O3, -Os* as shown in Table 8.3. Without using the new instructions, the number of clock cycles and instructions are the same. Number of instructions of the program is reduced for the round function 55,9% with *-O0* optimization level, and 61,7% with the other levels by using new instructions.

Table 8.3: Number of clock cycles and instructions in a round with/without new instructions.

Optimization	No new instruction	# of Clocks		# of Instructions	
		New Instruction	Reduction (%)	New Instruction	Reduction (%)
-Os	212	101	52,3	81	61,7
-O3	212	101	52,3	81	61,7
-O2	212	101	52,3	81	61,7
-O1	212	101	52,3	81	61,7
-O0	468	226	51,7	206	55,9

Rounds in permutations are changeable according to the implementation and variant. For example, Round numbers are a=12 and b=8 for the Ascon-128av variant of ASCON. These round numbers could be chosen according to the implementation of the algorithm. The execution count of permutations also depends on the length of associated data and plain text. Thus, the more repeated rounds and longer associated data and plain-text lead to the more percentage share of the number of clock cycles and instructions in an encryption and decryption.

Table 8.4: Number of clock cycles and instructions in an encryption with/without new instructions.

Optimization	No new instruction	# of Clocks		# of Instructions	
		New Instruction	Reduction (%)	New Instruction	Reduction (%)
-Os	29138	24386	16,3	23425	19,6
-O3	22684	17931	20,9	16970	25,2
-O2	22684	17931	20,9	16970	25,2
-O1	22684	17931	20,9	16970	25,2
-O0	71877	60261	16,1	59300	17,5

The total number of clocks of the encryption implementation of ASCON is decreased by 16,1% with *-O0*, 20,9% for *-O1*, *-O2* and *-O3* and 16,3% with *-Os* as shown in Table 8.4. Reduction of number of instructions are 17,5% with *-O0* optimization level, 25,2% for *-O1*, *-O2* and *-O3* and 19,6% with *-Os*.

CONCLUSION

In this study, RISC-V I set instructions are extended with the basic operations of permutation of a lightweight cryptography algorithm ASCON. A methodology is proposed to profile most repetitive and clock cycle consuming operations in the ASCON lightweight cryptography algorithm. Spike RISC-V ISA simulator is used to execute RISC-V ELF binaries on a host environment.

By using the results of proposed profiling methodology, RISC-V ISA is extended for ASCON with two versions of instructions R-type, *sbox* and *rot*, and I-type, *sboxi* and *roti*. The reason for extending with two versions of instruction is that by using integers as rvalues in source is compiled into I-type instructions in the assembly code. The second input operand of an I-type instruction that is immediate is filled in compile time; thus, it has no load operation for it in run time This reduces the number of instructions and clock cycles of a program in case of using rvalue in source code.

To compile the high-level program into an assembly code with the new instructions, RISC-V GNU Toolchain is extended. Opcodes of mask and match values of new instructions are added to GCC Binutils. The new instructions are generated by using inline assembly in the source code. Assembler of GCC Binutils generates the new instructions from the assembly block in source code.

Inline assembly is not good enough in terms of abstraction and readability, because the developer must know the type and register types of the instruction to use it in the source code. The instructions can be also generated by using GCC built-in functions. They are compiled into a machine instruction directly and can present a better abstraction and more readable code. Thus, GCC RISC-V back end is extended with instruction patterns of the new instructions and built-in functions.

Effects of the new instructions on performance of the program are investigated on Spike RISC-V ISA Simulator by extending the simulator with the new instructions. Functional behaviors of the new instructions are added to the simulator and an application-specific processor is modeled and simulated in terms of executing an ELF.

Spike returns the executed instruction count when the number of clock cycles is read. So, an algorithm is integrated into Spike to measure the total clock cycles of executed instructions.

The experiment results show that extending RISC-V with ASCON specific custom instructions has a great effect in terms of total clock cycles and number of instructions of the program. A round in permutations is accelerated at least 51.7% and instruction memory is reduced 55.9% in worst-case considering optimization level of the compiler. Encryption execution of the Ascon128av variant is accelerated 16.1% and instruction memory is reduced 17.5% for the resource constraint devices.

In this study, compiled RISC-V ELF binaries are run on Spike simulator rather than real hardware. The future work will be to design the new instructions for RISC-V on hardware and run compiled ELF binaries directly on hardware to verify the improvements.

REFERENCES

[1] **C. Dobraunig, M. Eichlseder, F. Mendel and M. Schlaffer**, (2019). Ascon v1.2. Submission to the NIST Lightweight Cryptography competition, http://ascon.iaik.tugraz.at

[2] **D. Bokan, M. Dukic, M. Popovic and N. Cetic,** (2014). Adjustment of GCC compiler frontend for embedded processors, *2014 22nd Telecommunications Forum Telfor (TELFOR),* Belgrade, Serbia, 2014, pp. 983-986, doi: 10.1109/TELFOR.2014.7034571.

[3] **L. Ghica and N. Tapus,** (2015). Optimized retargetable compiler for embedded processors - GCC vs LLVM, *2015 IEEE International Conference on Intelligent Computer Communication and Processing (ICCP),* Cluj-Napoca, Romania, 2015, pp. 103-108, doi: 10.1109/ICCP.2015.7312613.

[4] **ASCON C Implementation**, https://github.com/ascon/ascon-c

[5] **Spike RISC-V ISA Simulator**, https://github.com/riscv/riscv-isa-sim

[6] **M. Poorhosseini, W. Nebel and K. Grüttner,** (2020). A Compiler Comparison in the RISC-V Ecosystem, *2020 International Conference on Omnilayer Intelligent Systems (COINS),* Barcelona, Spain, 2020, pp. 1-6, doi: 10.1109/COINS49042.2020.9191411.

[7] **V. Thakor, M. A. Razzaque, and M. Khandaker,** (2020). Lightweight Cryptography for IoT: A State-of-the-Art, 2020.

[8] **K. McKay, L. Bassham, M. Sonmez and N. Mouha,** (2017). Report on Lightweight Cryptography, *NIST Interagency/Internal Report (NISTIR),* National Institute of Standards and Technology, Gaithersburg, MD, 2017, https://doi.org/10.6028/NIST.IR.8114

[9] **B. J. Mohd and T. Hayajneh,** (2018). Lightweight Block Ciphers for IoT: Energy Optimization and Survivability Techniques, *in IEEE Access*, vol. 6, pp. 35966-35978, 2018, doi: 10.1109/ACCESS.2018.2848586.

[10] **A. Waterman, Y. Lee, D. Patterson, and K. Asanovic,** (2016). The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1, 2016.

[11] **GNU Compiler Collection (GCC),** https://gcc.gnu.org/

[12] **N. B. Jensen, P. Schleuniger, A. Hindborg, M. Walter and S. Karlsson,** (2015). Experiences with Compiler Support for Processors with Exposed Pipelines, *2015 IEEE International Parallel and Distributed Processing Symposium Workshop,* Hyderabad, India, 2015, pp. 137-143, doi: 10.1109/IPDPSW.2015.9.

[13] **D. Patterson,** (2018). 50 Years of computer architecture: From the mainframe CPU to the domain-specific tpu and the open RISC-V instruction set,

2018 IEEE International Solid - State Circuits Conference - (ISSCC), San Francisco, CA, USA, 2018, pp. 27-31, doi: 10.1109/ISSCC.2018.8310168.

[14] **M. Rigger, S. Marr, B. Adams, and H. Mössenböck,** (2019). Understanding GCC builtins to develop better tools, *In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2019).* Association for Computing Machinery, New York, NY, USA, 74–85. DOI:https://doi.org/10.1145/3338906.3338907

[15] **M. Rigger, S. Marr, S. Kell, D. Leopoldseder, and H. Mössenböck,** (2018). An Analysis of x86-64 Inline Assembly in C Programs, In Proceedings of the 14th ACM SIGPLAN/SIGOPS *International Conference on Virtual Execution Environments (VEE '18).* ACM, New York, NY, USA, 84–99. https://doi.org/10.1145/3186411.3186418

[16] **RISC-V GNU Toolchain,** https://github.com/riscv/riscv-gnu-toolchain

[17] **GCC Internals,** https://gcc.gnu.org/onlinedocs/gccint/

[18] **M. K. Jain, M. Balakrishnan and A. Kumar,** (2001). ASIP design methodologies: survey and issues, *VLSI Design 2001. Fourteenth International Conference on VLSI Design*, Bangalore, India, 2001, pp. 76-81, doi: 10.1109/ICVD.2001.902643.

[19] **PULP Platform,** https://github.com/pulp-platform/pulp-riscv-gcc

[20] **E. N. İsman, C. Topal, L. Akçay and B. Örs,** (2020). Instruction Extension of an Open Source RV32IMC Core for NTRU Cryptosystem, *2020 European Conference on Circuit Theory and Design (ECCTD),* Sofia, Bulgaria, 2020, pp. 1-5, doi: 10.1109/ECCTD49232.2020.9218372.

[21] **E. Tehrani, T. Graba, A. S. Merabet and J. L. Danger,** (2020). RISC-V Extension for Lightweight Cryptography, *2020 23rd Euromicro Conference on Digital System Design (DSD),* Kranj, Slovenia, 2020, pp. 222-228, doi: 10.1109/DSD51259.2020.00045.

[22] **A. Shisha, M. AboelMaged, R. Aboshabaan, A. Hassan, M. Fouad and M. A. A. El Ghany,** (2020). Efficient Hardware Implementation for IoT Security System, *2020 2nd Novel Intelligent and Leading Emerging Sciences Conference (NILES),* Giza, Egypt, 2020, pp. 204-207, doi: 10.1109/NILES50944.2020.9257939.

[23] **I. M. Shaher, M. Mahmoud, H. Ibrahim, M. Ali, and H. Mostafa,** (2020). Implementation of a Hardware Accelerator for a Real-time Encryption System, *2020 IEEE 63rd International Midwest Symposium on Circuits and Systems (MWSCAS),* Springfield, MA, USA, 2020, pp. 627-630, doi: 10.1109/MWSCAS48704.2020.9184446.

[24] **G. Xin et al.** (2020). VPQC: A Domain-Specific Vector Processor for Post-Quantum Cryptography Based on RISC-V Architecture, *in IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 8, pp. 2672-2684, Aug. 2020, doi: 10.1109/TCSI.2020.2983185.

[25] **NIST Cryptographic Competitions,** https://competitions.cr.yp.to/index.html

[26] **RISC-V Foundation Software Resources,** https://riscv.org/exchange/software/

[27] **SiFive's Freedom RISC-V Tools,** https://github.com/sifive/freedom-tools

[28] **Using the GNU Compiler Collection,** https://gcc.gnu.org/onlinedocs/gcc

[29] **CMAKE**, https://cmake.org/

[30] **RISC-V Proxy Kernel,** https://github.com/riscv/riscv-pk

[31] **S. Payvar, E. Pekkarinen, R. Stahl, D. Mueller-Gritschneder and T. D. Hämäläinen,** (2019). Instruction Extension of a RISC-V Processor Modeled with IP-XACT, *2019 IEEE Nordic Circuits and Systems Conference (NORCAS): NORCHIP and International Symposium of System-on-Chip (SoC)*, 2019, pp. 1-5, doi: 10.1109/NORCHIP.2019.8906975.

[32] **A. Traber, F. Zaruba, S. Stucki, A. Pullini, G. Haugou, E. Flamand, F. K. Gurkaynak, and L. Benini,** (2016). PULPino: A small single-core RISCV SoC, *in RISC-V Workshop*, 2016.

[33] **Modified RISC-V Tools,** https://github.com/oaltinay

APPENDICES

APPENDIX A: *step* method in Spike

APPENDIX A

```
void processor_t::step(size_t n)
{
  if (!state.debug_mode) {
    if (halt_request == HR_REGULAR) {
      enter_debug_mode(DCSR_CAUSE_DEBUGINT);
    } else if (halt_request == HR_GROUP) {
      enter_debug_mode(DCSR_CAUSE_GROUP);
    } // !!!The halt bit in DCSR is deprecated.
    else if (state.dcsr.halt) {
      enter_debug_mode(DCSR_CAUSE_HALT);
    }
  }

  while (n > 0) {
    size_t instret = 0;
    uint64_t cycle_total = 0;
    uint64_t cycle = 0;
    reg_t pc = state.pc;
    mmu_t* _mmu = mmu;

    #define advance_pc() \
     if (unlikely(invalid_pc(pc))) { \
       switch (pc) { \
         case PC_SERIALIZE_BEFORE: state.serialized =
 true; break; \
         case PC_SERIALIZE_AFTER: { cycle_total +=
 cycle; ++instret; break;}  \
         case PC_SERIALIZE_WFI: { cycle_total +=
 cycle; n = ++instret; break;} \
         default: abort(); \
       } \
       pc = state.pc; \
       break; \
     } else { \
       state.pc = pc; \
       instret++; \
       cycle_total += cycle; \
     }

    try
    {
      take_pending_interrupt();

      if (unlikely(slow_path()))
      {
        while (instret < n)
        {
          if (unlikely(!state.serialized && state.
single_step == state.STEP_STEPPED)) {
            state.single_step = state.STEP_NONE;
            if (!state.debug_mode) {
              enter_debug_mode(DCSR_CAUSE_STEP);
              // enter_debug_mode changed state.pc, s
o we can't just continue.
              break;
            }
          }

          if (unlikely(state.single_step == state.
STEP_STEPPING)) {
            state.single_step = state.STEP_STEPPED;
          }

          insn_fetch_t fetch = mmu->load_insn(pc);
          if (debug && !state.serialized)
            disasm(fetch.insn);
          pc = execute_insn(this, pc, fetch);
          cycle = get_clock_cycle(fetch.insn);
          advance_pc();
        }
      }
```

```
    else while (instret < n)
    {
      size_t idx = _mmu->icache_index(pc);
      auto ic_entry = _mmu->access_icache(pc);
      #define ICACHE_ACCESS(i) { \
        insn_fetch_t fetch = ic_entry->data; \
        pc = execute_insn(this, pc, fetch); \
        cycle = get_clock_cycle(fetch.insn); \
        ic_entry = ic_entry->next; \
        if (i == mmu_t::ICACHE_ENTRIES-1) break; \
        if (unlikely(ic_entry->tag != pc)) break; \
        if (unlikely(instret+1 == n)) break; \
        instret++; \
        state.pc = pc; \
        cycle_total += cycle; \
      }

      // This switch statement implements the modif
ied Duff's device as
      // explained above.
      switch (idx) {
        // "icache.h" is generated by the gen_icach
e script
        #include "icache.h"
      }

      advance_pc();
    }
  }
  catch(trap_t& t)
  {
    take_trap(t, pc);
    n = instret;

    if (unlikely(state.single_step == state.
STEP_STEPPED)) {
      state.single_step = state.STEP_NONE;
      enter_debug_mode(DCSR_CAUSE_STEP);
    }
  }
  catch (trigger_matched_t& t)
  {
    if (mmu->matched_trigger) {
      insn_fetch_t fetch = mmu->load_insn(pc);
      pc = execute_insn(this, pc, fetch);
      cycle = get_clock_cycle(fetch.insn);
      advance_pc();

      delete mmu->matched_trigger;
      mmu->matched_trigger = NULL;
    }
    switch (state.mcontrol[t.index].action) {
      case ACTION_DEBUG_MODE:
        enter_debug_mode(DCSR_CAUSE_HWBP);
        break;
      case ACTION_DEBUG_EXCEPTION: {
        insn_trap_t trap(CAUSE_BREAKPOINT, t.
address);
        take_trap(trap, pc);
        break;
      }
      default:
        abort();
    }
  }
  catch (wait_for_interrupt_t &t)
  {
    n = instret;
  }

  state.minstret += instret;
  n -= instret;
  state.mcycle += cycle_total;
}
```

Figure A.1: Updated version of *step* method in Spike[5].

CURRICULUM VITAE

Name Surname **:** Özlem ALTINAY

EDUCATION **:**

- **B.Sc.** **:** 2018, Yıldız Technical University, Faculty of Electrical and Electronics, Department of Electronics and Communication Engineering

PROFESSIONAL EXPERIENCE AND REWARDS:

- 2018 - 2019 Yaltes – Software Engineer.
- 2019 - 2020 Ford Otosan – Autonomous Vehicles Engineer.
- 2020 - Present Siemens – Software Engineer.