RISC-V INSTRUCTION SET EXTENSION ON BLOCKCHAIN APPLICATION

By

Cheong Kin Seng

A REPORT

SUBMITTED TO

Universiti Tunku Abdul Rahman

in partial fulfillment of the requirements

for the degree of

BACHELOR OF INFORMATION TECHNOLOGY (HONOURS) COMPUTER ENGINEERING

Faculty of Information and Communication Technology

(Kampar Campus)

JAN 2024

UNIVERSITI TUNKU ABDUL RAHMAN

REPORT STATUS DECLARATION FORM

Title: RISC-V INSTRUCTION SET EXTENSION ON BLOCKCHAIN APPLICATION

Academic Session: JAN 2024

I CHEONG KIN SENG

(CAPITAL LETTER)

declare that I allow this Final Year Project Report to be kept in Universiti Tunku Abdul Rahman Library subject to the regulations as follows:

1. The dissertation is a property of the Library.
2. The Library is allowed to make copies of this dissertation for academic purposes.

Verified by,

[illegible]	[illegible]
(Author's signature)	(Supervisor's signature)
Address:	
3, Lorong Kledang Timur 13,	
Taman Rasi, 31450 Menglembu,	Ooi Joo On
Perak.	Supervisor's name
Date: 25 APRIL 2024	**Date**: 26 APRIL 2024

Universiti Tunku Abdul Rahman			
Form Title : **Sample of Submission Sheet for FYP/Dissertation/Thesis**			
Form Number: **FM-IAD-004**	Rev No.: **0**	Effective Date: **21 JUNE 2011**	Page No.: **1 of 1**

FACULTY/INSTITUTE* OF INFORMATION AND COMMUNICATION TECHNOLOGY

UNIVERSITI TUNKU ABDUL RAHMAN

Date: 25 APRIL 2024

SUBMISSION OF FINAL YEAR PROJECT /DISSERTATION/THESIS

It is hereby certified that ***Cheong Kin Seng*** (ID No: ***20ACB03898***) has completed this final year project/ dissertation/ thesis* entitled "*RISC-V Instruction Set Extension on Blockchain Application*" under the supervision of Dr. Ooi Joo Onn (Supervisor) from the Department of Digital Economy Technology, Faculty/Institute* of Information and Communication Technology , and ____________ (Co-Supervisor)* from the Department of ____________, Faculty/Institute* of ____________.

I understand that University will upload softcopy of my final year project / dissertation/ thesis* in pdf format into UTAR Institutional Repository, which may be made accessible to UTAR community and public.

Yours truly,

[illegible]

(*Cheong Kin Seng*)

*Delete whichever not applicable

DECLARATION OF ORIGINALITY

I declare that this report entitled "**RISC-V INSTRUCTION SET EXTENSION ON BLOCKCHAIN APPLICATION**" is my own work except as cited in the references. The report has not been accepted for any degree and is not being submitted concurrently in candidature for any degree or other award.

Signature	:	[illegible]
Name	:	Cheong Kin Seng
Date	:	25 April 2024

ACKNOWLEDGEMENTS

I would like to express my sincere thanks and appreciation to my supervisor, Dr Ooi Joo Onn who has given me this bright opportunity to engage in a RISC-V architecture project. It is my first step to establish a career in IC design field with relate to the RISC-V architecture which is trending now. A million thanks to you. Besides, I would like to thank my moderator, Mr Lee Heng Yew, to take part in the review of my project, in order to provide me some useful insights and comments for me to improve on my project.

Finally, I must say thanks to my parents and my family for their love, support, and continuous encouragement throughout the course.

ABSTRACT

This project is an instruction set extension project based on RISC-V architecture for academic purpose. Blockchain technology was widely used today to keep records due to its high security and reliability. However, blockchain required a high computing power to function. Although there were numerous ways to improve the performance speed of blockchain technology in software implementations, hardware implementation of the blockchain algorithms was a more preferred choice due to the emerging open-source computer architecture, RISC-V. RISC-V was free and open license for anyone to customize their IC design. By adding new instruction extensions to the RISC-V cores, they could be specialized to run certain types of tasks. This would greatly shorten the instructions used by the algorithms and improved the execution time of the programmes. One of the most common cryptographic algorithms used in blockchain would be selected in this paper, typically djb2 hash algorithm. In this project, some instructions were proposed to execute the cryptographic algorithm in shorter clock cycles and shorter execution time. Towards the end of the project, the algorithms would be executed in a base RISC-V core and an extended RISC-V core using simulation tools to perform performance analysis. The simulation tool used in this project was Chipyard, which is a one-stop development tool for anything regarding RISC-V customization. One of the main components in Chipyard was Spike simulator, which was a software simulation tool in RISC-V standards to execute the software executable file and also output the hardware information used during the execution. Spike was used to run the compiled source codes in C/C++ language to determine the execution time and clock cycles used by the programme. A RISC-V GNU toolchain was installed for the compilation of the programme. The toolchain was also customised and extended with extra instructions to compile the programme. The compiled programme was simulated in Spike ISA simulator to test with the extension and without the extension. The results were presented at the end of the report.

TABLE OF CONTENTS

TITLE PAGE **i**

REPORT STATUS DECLARATION FORM **ii**

FYP THESIS SUBMISSION FORM **iii**

DECLARATION OF ORIGINALITY **iv**

ACKNOWLEDGEMENTS **v**

ABSTRACT **vi**

TABLE OF CONTENTS **vii**

LIST OF FIGURES **x**

LIST OF TABLES **xi**

LIST OF SYMBOLS **xii**

LIST OF ABBREVIATIONS **xiii**

CHAPTER 1 INTRODUCTION **1**

1.1 Problem Statement and Motivation 1

1.2 Objectives 2

1.3 Project Scope and Direction 3

1.4 Contributions 3

1.5 Report Organization 4

CHAPTER 2 LITERATURE REVIEW **5**

2.1 A RISC-V Processor with Area-Efficient Memristor-Based In-Memory Computing for Hash Algorithm in Blockchain Applications 5

2.2 Symmetric Cryptography on RISC-V: Performance Evaluation of Standardized Algorithms 6

2.3 RISC-V Instruction Set Extensions for Lightweight Symmetric Cryptography 6

2.4 VPQC: A Domain-Specific Vector Processor for Post-Quantum Cryptography Based on RISC-V Architecture 7

2.5 The design of scalar AES instruction set extensions for RISC-V 8

2.6 Critical Remarks of Previous Works 9

2.6.1 Strengths 9

2.6.2 Weaknesses 9

CHAPTER 3 SYSTEM MODEL (FOR RESEARCH-BASED PROJECT) 10

3.1 System Design Diagram/Equation 10

3.1.1 Description of DJB2 String Hash 10

3.1.2 Translation Hierarchy of C/C++ Programmes 11

3.1.3 System Architecture Diagram 12

3.2 RISC-V Architecture 13

CHAPTER 4 SYSTEM DESIGN 15

4.1 Design of Instruction Set Extension 15

4.1.1 Block Diagram of DJB2 String Hash Instruction Extension 15

4.1.2 Opcode of DJB2 String Hash Instruction Extension 15

4.1.3 Pseudocode of DJB2 String Hash Instruction Extension 17

4.2 Blockchain Application Source Codes 17

4.2.1 Assembly Codes of String Hash Function 18

4.3 Unit Test Programmes 19

4.3.1 Assembly Codes of Unit Test Programmes 22

CHAPTER 5 EXPERIMENT/SIMULATION 24

5.1 Hardware Setup 24

5.2 Software Setup 24

5.3 Setting and Configuration 24

5.4 System Operation (with Screenshot) 25

5.4.1 Source Code Compilation 25

5.4.2 Extending RISC-V GNU Toolchain 25

5.4.3 Modification of Spike ISA Simulator 27

5.5 Implementation Issues and Challenges 30

5.6 Concluding Remark 31

CHAPTER 6 SYSTEM EVALUATION AND DISCUSSION **32**

6.1 System Testing and Performance Metrics 32
6.2 Testing Setup and Result 32
6.3 Project Challenges 33
6.4 Objectives Evaluation 33
6.5 Concluding Remark 33

CHAPTER 7 CONCLUSION AND RECOMMENDATION **34**

7.1 Conclusion 34
7.2 Recommendation 34

REFERENCES **35**
APPENDIX **36**
WEEKLY LOG **49**
POSTER **54**
PLAGIARISM CHECK RESULT **55**
FYP2 CHECKLIST **58**

LIST OF FIGURES

Figure Number	Title	Page
Figure 1.5.1	Simplified view of the blockchain.	4
Figure 1.5.2	Cryptographic algorithms.	6
Figure 1.5.3	Cryptographic algorithms.	6
Figure 2.1.1	Modified RISC-V processor core with in-memory computing (IMC).	8
Figure 2.1.2	RV32I assembly code of SBOX table.	9
Figure 2.1.4	The architecture of VPQC.	11
Figure 2.1.5.1	An AES encryption round implemented using hardware-assisted T-tables.	11
Figure 2.1.5.2	Instruction mnemonics, and their mapping onto pseudo-code functions.	12
Figure 3.1.1	C implementation of djb2 string hash.	10
Figure 3.1.2	Translation Hierarchy for C/C++ Programmes	11
Figure 3.1.3	System Design Implementation Flowchart	12
Figure 3.2.2	RISC-V 32-bit instruction formats	13
Figure 4.1.1.1	Block Diagram of Instruction Set Extension	15
Figure 4.1.2.1	RISC-V Base Opcode Map	15
Figure 4.1.2.2	Opcodes of DJB2 String Hash Instruction	16
Figure 4.1.2.3	Opcode's Mask and Match Values of Custom Instruction	16
Figure 4.1.2.4	Format for the Declaration of Instruction	16
Figure 4.2	Code Snippet of hash.c	18
Figure 4.2.1	Assembly Codes of hash.c	19
Figure 4.3.1	Code Snippet of test_hash.c	20
Figure 4.3.2	Code Snippet of test_hash_asm.c	21

LIST OF TABLES

Table Number	Title	Page
Table 1.5.1	The structure of a block.	4
Table 1.5.2	The structure of a block header.	5
Table 3.1.1	Specifications of PC	16
Table 3.2.1.1	DWORD padding	17
Table 5.1.1	Specifications of PC	24
Table 5.3.1	Usage of Optimization Flags	25
Table 6.2.1	Simulation Results	32

LIST OF SYMBOLS

β	beta
Ω	Ohm (resistance)

LIST OF ABBREVIATIONS

PoW	Proof of Work
RISC-V	Reduced Instruction Set Computer (V)
ISA	Instruction Set Architecture
ISE	Instruction Set Extension
AES	Advanced Encryption Standard
SHA	Secure Hashing Algorithm
SM2	ShangMi 2
SM3	ShangMi 3
SM4	ShangMi 4
NIST	National Institute of Standards and Technology
HDL	Hardware Description Language
LWC	Light Weight Cryptography
LWE	Learning with Errors
RTL	Register-Transfer Logic
HDL	Hardware Description Language

Chapter 1
Introduction

1.1 Problem Statement and Motivation

Mining was the mechanism that underpinned the decentralized clearing house, by which transactions were validated and cleared [2]. However, on average, every 10 minutes [2] a new block was created with transactions that had taken place since the previous block, effectively adding these transactions to the blockchain. It was important to improve the efficiency of computing the cryptographic algorithms for the blockchain technology to scale economically and timely. One way to improve the performance of adding the new blocks to the chain was to have the computing chips designed to run specifically for the use of blockchain technology.

RISC-V was an instruction set architecture (ISA) which was developed from reduced instruction set computers (RISC) concepts. What set RISC-V apart from other ISA designs was that it was freely available under open-source licenses (Refer Appendix A-1 for its architecture). [4] There had been a recent surge of interest in RISC-V, with many companies starting to offer RISC-V hardware. It supported 32-bit, 64-bit, and 128-bit architectures. RISC-V's popularity stemmed from its ability to facilitate the development of specialized microprocessor designs. Unlike other architectures, RISC-V was flexible and modular, which allowed for tailored designs that were free from unnecessary features and capabilities that could negatively impact performance and energy usage. Additionally, RISC-V's open-source nature lowered its cost compared to proprietary RISC.

In October 2021, RISC-V published the volume 1 of the cryptography ISE that provided the developers a set of specialized instructions for cryptographic algorithms used in scalar calculations, namely the “Scalar & Entropy Source Instructions” [5]. In April 2023, RISC-V published the volume 2 of the cryptography ISE for the use in vector calculations, namely the “Vector Instructions” [6]. In this paper, a solution was proposed to extend the RISC-V ISA cryptography extension to enhance the performance of computing blockchain algorithms in terms of clock cycles and execution time.

The motivation of this project was that the blockchain technology was a decentralized ledger that ensured secure and transparent transaction records. The network was maintained by nodes that verified and appended transactions to the blockchain. However, with an increase in the number of transactions, the need for additional nodes increased, leading to potential delays and increased fees [7]. Enhancing the clock cycles of the blockchain system could improve transaction speed and decrease fees, thus making the technology more efficient and accessible to businesses and individuals.

1.2 Objectives

The aim of the paper was to propose a few specific hardware instructions to be added into the RISC-V ISA cryptography extension for the acceleration of some common cryptographic algorithms used in blockchain. There were a few objectives that should be achieved in this project.

Firstly, a functioning blockchain application with its cryptographic hash function should be built. A thorough study through the algorithm in RISC-V assembly codes should be studied **to design its instruction set extension for customization**.

Firstly, the proposed instructions should be able **to reduce the code size** when the algorithms were translated into assembly languages, thus decreasing the instructions per clock cycle when the processor was running the programmes. The code size will be compared between the algorithms when running the base RV32I or RV64I instruction set versus the one running in the base instruction set with the proposed instruction set.

Secondly, the proposed instructions were expected **to improve the performance of clock cycles and execution time** of the algorithms mentioned when running in the proposed solution than using the base RV32IMAFDC or RV64IMAFDC instruction set.

1.3 Project Scope and Direction

The scope of the project was to design a RISC-V ISE for the cryptographic algorithm of blockchain technology. This included with developing the blockchain application with cryptographic hash algorithm for design development, testing and verification. The selection of tools for this project was necessary to learn about the functionality and coverage for the success of this project. Besides, the RISC-V GNU Toolchain should be customized with additional instruction, as well as to ensure the source codes would be compiled with the custom instruction set. Then, a performance benchmark of the hash algorithm would be carried out to compare the performance when running in a base RISC-V machine and in a base RISC-V machine with the extended instruction set.

1.4 Contributions

The proposed RISC-V instruction set extension (ISE) aimed to improve the computing performance of certain computational tasks by providing a set of new instructions optimized for these tasks. By adding these instructions to the RISC-V ISA, the project would enable more efficient and faster execution of these tasks on RISC-V-based processors. Unlike existing solutions that rely on software-based optimizations or custom hardware accelerators, the approach in this paper leveraged the flexibility of the RISC-V ISA to add new instructions that could be executed directly by the processor. This approach not only simplified the design and implementation of the acceleration but also reduced the overhead of executing the task on the processor.

A thorough analysis of the existing RISC-V ISA was conducted and identified with the computational tasks that could benefit the most from the custom instructions. A set of new instructions for these tasks were designed and prototyped as well as their performance was evaluated on a simulated RISC-V processor. To evaluate the impact and contribution of the suggested ISE, a comprehensive set of benchmarks was planned to conduct to run the blockchain algorithms in the RISC-V simulator with the original instruction set and extended instructions. The results were expected to demonstrate a significant improvement in performance compared to existing solutions.

1.5 Report Organization

This report consists of seven chapters were written in this report. Chapter 1 was the project introduction; Chapter 2 was review of literature; Chapter 3 was describing the system model; Chapter 4 was outlining the system design; Chapter 5 was the detailed process of experiment and simulation; Chapter 6 reported the system testing and evaluation; and Chapter 7 concluded the project and suggestions for further improvements.

Chapter 2
Literature Review

2.1 "A RISC-V Processor with Area-Efficient Memristor-Based In-Memory Computing for Hash Algorithm in Blockchain Applications"

Xue *et al.* (2019) proposed the addition of a memristor-based in-memory computing (IMC) core on a RISC-V processor for the blockchain technology. Figure 2.1.1 [8] showed the additional memory module, i.e., the IMC module was installed in the CPU core along with with a modified IMC controller and operation module. The IMC-adapted instructions, which extended from the base RISC-V ISA were designed specifically for the Keccak hash algorithm. The authors were successful to prove that the addition of the IMC core in the RISC-V processor could save both execution time and power consumption tremendously by 70% as compared to the base processor without any extension.

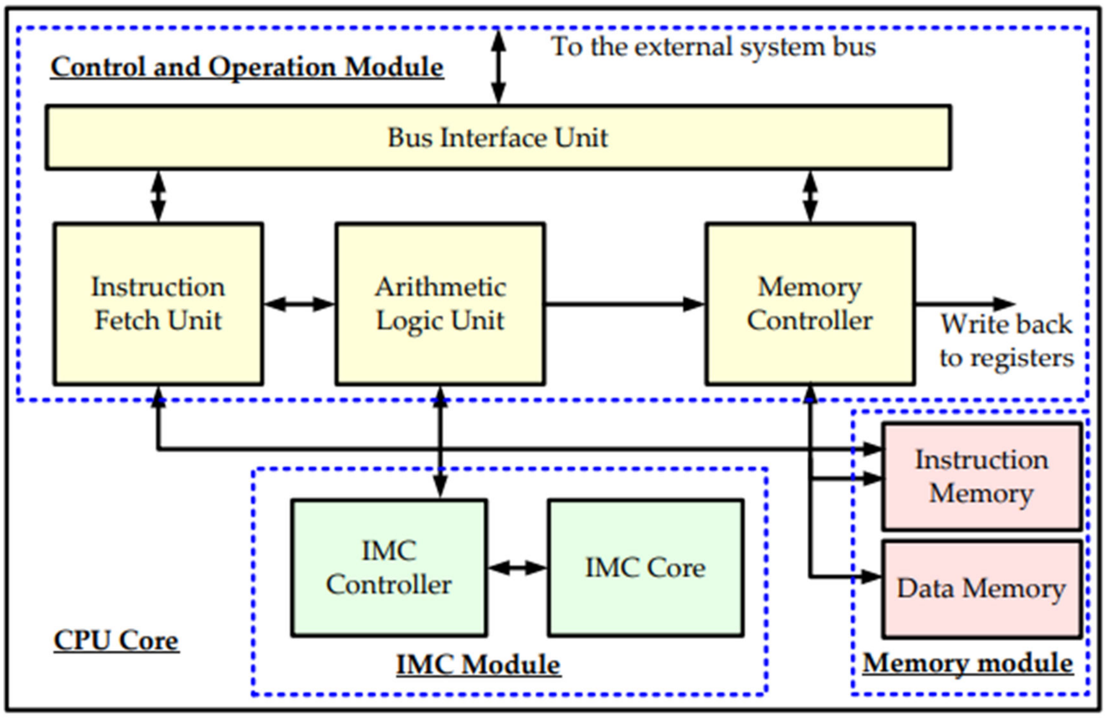
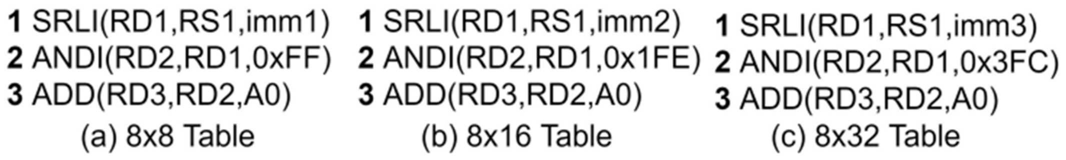


Figure 2.1.1 Custom in-memory computing (IMC) module.

2.2 “Symmetric Cryptography on RISC-V: Performance Evaluation of Standardized Algorithms”

Nisancı, G., Flikkema, P.G., and Yalçın, T. (2022) presented software-only algorithms with the RISC-V RV32I ISA. The performance of these algorithms was compared to the performance of a RISC-V processor with customized hardware design cryptographic execution. They implemented grev[i], shlf[i], and unshlf[i] in their work. The authors [9] showed that the software implementations with the RISC-V cryptography set extension provided significant improvements for the selected algorithms (AES, CAST-128, SEED-V1, CAMELLIA V1, SEED-V2) at an additional hardware cost of less than 9%. It is indeed a considerable amount of investment to upgrade the hardware to cater for the needs of the cryptographic algorithms. The authors also proposed a new instruction to accelerate the operations for calculation of memory address for 8-bit input SBOX table. Figure 2.1.2 showed the RV32I assembly code of the SBOX tables. Therefore, the authors used one instruction, instead of three to calculate the memory address for 8-bit input SBOX tables.



```
1 SRLI(RD1,RS1,imm1)   1 SRLI(RD1,RS1,imm2)    1 SRLI(RD1,RS1,imm3)
2 ANDI(RD2,RD1,0xFF)   2 ANDI(RD2,RD1,0x1FE)   2 ANDI(RD2,RD1,0x3FC)
3 ADD(RD3,RD2,A0)      3 ADD(RD3,RD2,A0)       3 ADD(RD3,RD2,A0)
    (a) 8x8 Table          (b) 8x16 Table          (c) 8x32 Table
```

Figure 2.1.2 RV32I assembly code of SBOX table.

2.3 “RISC-V Instruction Set Extensions for Lightweight Symmetric Cryptography”

Cheng H. *et al.* (2023) presented the design, implementation, and evaluation of each ISE for the ten LightWeight Cryptography (LWC) selected which were suitable for resource-constrained devices. The authors [10] developed the ISE designs for ten of the said algorithms by following a set of principled constraints. First, they followed the RISC-V design principles, which was the instructions with the three registers. Then, the authors implemented the designs with the RISC-V compliant Rocket core. When compared to software-only options, the authors observed that 1) ISEs had minimal additional overhead costs in hardware, 2) the ISEs can reduce execution time, which varies based on the algorithm, and 3) the ISEs allow for consistent execution time and a decrease in the size of the instruction set. varies based on the algorithm, and 3) the

ISEs allow for consistent execution time and a decrease in the size of the instruction set.

2.4 "VPQC: A Domain-Specific Vector Processor for Post-Quantum Cryptography Based on RISC-V Architecture"

Xin, G. *et al.* (2020) suggested the development of a configurable cryptographic processor, VPQC, for key encapsulation schemes, running both Ring-LWE and Module-LWE schemes, which included a vector co-processor in a basic RISC-V core in Figure 2.1.4. They explored the vectorization of number theoretic transform (NTT) and sampling algorithms and design a high-performance vector architecture using custom instructions that extended the RISC-V ISA. The proposed processor [11] exhibited significantly faster computation speeds for key encapsulation mechanisms (KEM) protocols compared to previous implementations, providing a high-speed PQC platform for security applications.

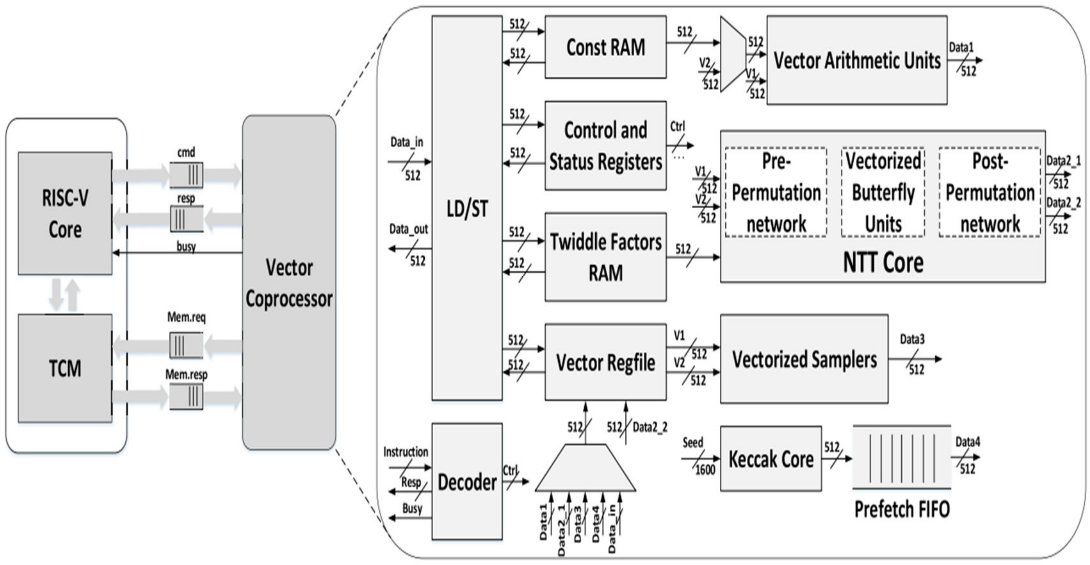


Figure 2.1.4 The architecture of VPQC.

2.5 "The design of scalar AES instruction set extensions for RISC-V"

Marshall *et al.* (2020) implemented and evaluated five ISE designs for AES on two different RISC-V complaint base microarchitectures. The authors found that the best design for AES on 32-bit cores was to use a hardware-assisted T-tables, which required only 20 instructions per round as shown in Figure 2.1.5.

```
lw              a0, 16(RK)      // Load Round Key
lw              a1, 20(RK)
lw              a2, 24(RK)
lw              a3, 28(RK)      // t0,t1,t2,t3 contains current round state.
saes.v3.encsm   a0, a0, t0, 0   // Next state for column 0.
saes.v3.encsm   a0, a0, t1, 1   // Current column 0 in t0.
saes.v3.encsm   a0, a0, t2, 2   // Next column 0 accumulates in a0
saes.v3.encsm   a0, a0, t3, 3
saes.v3.encsm   a1, a1, t1, 0   // Next state for column 1.
saes.v3.encsm   a1, a1, t2, 1
saes.v3.encsm   a1, a1, t3, 2
saes.v3.encsm   a1, a1, t0, 3
saes.v3.encsm   a2, a2, t2, 0   // Next state for column 2.
saes.v3.encsm   a2, a2, t3, 1
saes.v3.encsm   a2, a2, t0, 2
saes.v3.encsm   a2, a2, t1, 3
saes.v3.encsm   a3, a3, t3, 0   // Next state for column 3.
saes.v3.encsm   a3, a3, t0, 1
saes.v3.encsm   a3, a3, t1, 2
saes.v3.encsm   a3, a3, t2, 3   // a0,a1,a2,a3 contains new round state
```

Figure 2.1.5.1 An AES encryption round implemented using hardware-assisted T-tables.

For the AES on 64-bit cores, the best design option was to adopt a 64-bit data-path, where two columns were packed into a 64-bit word. Figure 2.1.5.2 showed the examples of the assembly instructions of the AES.

```
saes.v4.ks1      rd rs1 rcon  : v4.ks1(rd, rs1, rcon)
saes.v4.ks2      rd rs1 rs2   : v4.ks2(rd, rs1, rs2 )
saes.v4.imix     rd rs1       : v4.InvMix(rd, rs1)
saes.v4.encsm    rd rs1 rs2   : v4.Enc(rd, rs1, rs2, mix=1)
saes.v4.encs     rd rs1 rs2   : v4.Enc(rd, rs1, rs2, mix=0)
saes.v4.decsm    rd rs1 rs2   : v4.Dec(rd, rs1, rs2, mix=1)
saes.v4.decs     rd rs1 rs2   : v4.Dec(rd, rs1, rs2, mix=0)

v4.ks1(rd, rs1, enc_rcon):     // KeySchedule: SubBytes, Rotate, Round Const
    temp.32    = rs1.32[1]
    rcon       = 0x0
    if(enc_rcon != 0xA):
        temp.32 = ROTR32(temp.32, 8)
        rcon    = RoundConstants.8[enc_rcon]
    temp.8[i]  = AESSBox[temp.8[i]]   for i=0..3
    temp.8[0]  = temp.8[0] ^ rcon
    rd.64      = {temp.32, temp.32}

v4.ks2(rd, rs1, rs2):            // KeySchedule: XOR
    rd.32[0]   = rs1.32[1] ^ rs2.32[0]
    rd.32[1]   = rs1.32[1] ^ rs2.32[0] ^ rs2.32[1]

v4.Enc(rd, rs1, rs2, mix): // SubBytes, ShiftRows, MixColumns
    t1.128     = ShiftRows({rs2, rs1})
    t2.64      = t1.64[0]
    t3.8[i]    = AESSBox[t2.8[i]] for i=0..7
    rd.32[i]   = AESMixColumn(t3.32[i]) if mix else t3.32[i] for i=0..1

v4.Dec(rd, rs1, rs2, mix, hi): // InvSubBytes, InvShiftRows, InvMixColumns
    t1.128     = InvShiftRows(rs2 || rs1)
    t2.64      = t1.64[0]
    t3.8[i]    = AESInvSBox[t2.8[i]] for i=0..7
    rd.32[i]   = AESInvMixColumn(t3.32[i]) if mix else t3.32[i] for i=0..1

v4.InvMix(rd, rs1):              // Inverse MixColumns
    rd.32[i]   = AESInvMixColumn(rs1.32[i]) for i=0..1
```

Figure 2.1.6 AES pseudo-code functions.

Furthermore, the RISC-V bitmanip ISE could combine with either a hardware assisted T-tables or a 64-bit datapath to support AES-GCM, which the block ciphers took advantage of parallel processing.

2.2 Critical Remarks of Previous Works

2.2.1 Strengths

The strengths of the IMC core as proposed by Xue *et al.* [8] were the addition of the additional IMC module in the CPU core to assist in accelerating the execution time and clock cycles of the processes. The authors implemented additional memory in their solution that could reduce the execution time by 70% as compared to the base core without extension.

Nisancı, G., Flikkema, P.G., and Yalçın, T. implemented few custom instructions to reduce the execution time of the symmetric cryptographic algorithms by accelerating the calculation of memory address used by the algorithms. The solution was proven successful to speed up the execution time and clock cycles when running the algorithms with the custom instructions.

2.2.2 Weaknesses

However, these techniques also have several weaknesses. One of the studies did not cover on the diverse cryptographic algorithms found in blockchain application with the addition of the IMC core [8] in RISC-V processor as proposed by Xue *et al.*, although the authors achieved a remarkable performance to reduce the execution time and power consumption with limited area overhead in the execution of Keccak algorithm. It is also expensive for the chip manufacturers to design and integrate it with the RISC-V core.

Nisancı, G., Flikkema, P.G., and Yalçın, T. used loop unrolling [9] in the software implementations of cryptographic algorithms to produce their results, which increased the programme's execution speed. However, this practice is not reliable as the developers of the cryptographic algorithms would prefer using 'for' loop and 'while' loop in their approach to save time and reduce workload.

Xin *et al.* proposed the addition of a vector co-processor with the RISC-V base core to run the post-quantum cryptographic workloads. Although the computation speeds for the workloads were significantly improved, the processor core did not show versatility to support the other cryptographic algorithms. To implement a non-versatile processor core to operate for a limited number of functions, it would be very costly and non-economical, just like in [8].

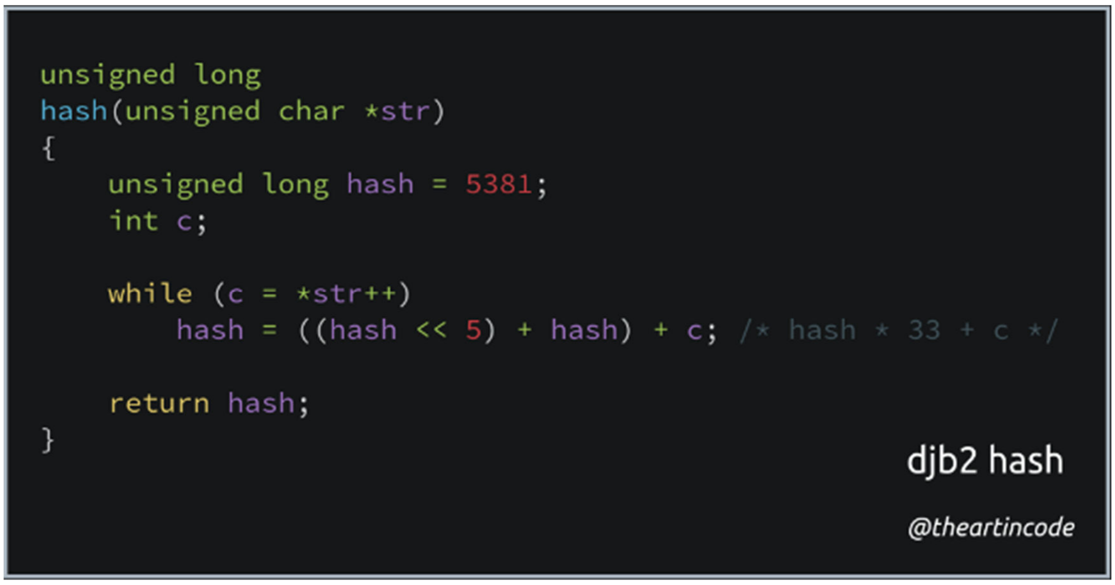
Chapter 3

System Model

3.1 System Design Diagram/Equation

3.1.1 Description of DJB2 String Hash

The cryptographic algorithm selected in the system design was djb2 string hash algorithm. A simple implementation of the djb2 in C was illustrated in Figure 3.1.1 [17].



```
unsigned long
hash(unsigned char *str)
{
    unsigned long hash = 5381;
    int c;

    while (c = *str++)
        hash = ((hash << 5) + hash) + c; /* hash * 33 + c */

    return hash;
}
```

Figure 3.1.1 C implementation of djb2 string hash.

The 32-bits prime number 5381 was assigned to the variable 'hash'. A full iteration of the character string 'str' was performed to each of its character with the djb2 string hash algorithm. The djb2 string hash algorithm would firstly perform a shift-left operation to the hash variable by 5, and then add it back by itself. This would also mean that the hash variable was multiplied by itself by a factor of 33. After that, the ASCII value of the current character was added to it. After the full iteration of the character string was being done, the function would return the final value of hash to the caller function.

3.1.2 Translation Hierarchy for C/C++ Programmes

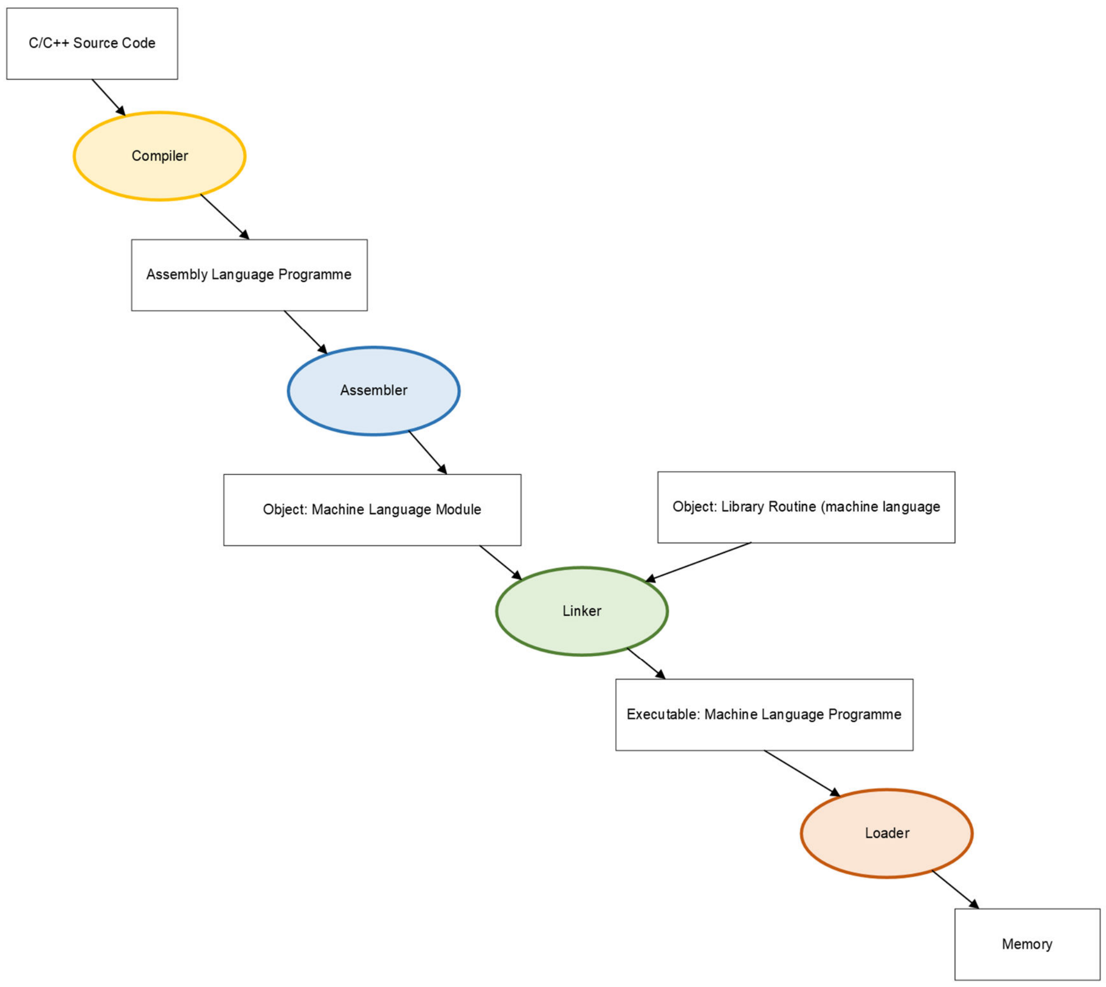

C/C++ Source Code
Compiler
Assembly Language Programme
Assembler
Object: Machine Language Module
Object: Library Routine (machine language
Linker
Executable: Machine Language Programme
Loader
Memory

Figure 3.1.2 Translation Hierarchy for C/C++ Programmes

In Figure 3.2.3 above, to be able to run in a RISC-V machine, C/C++ source codes were to be compiled with a RISC-V compatible compiler. An assembly language programme would be produced and passed to the assembler. Then, the assembler would generate an object file, which would be the machine language module. Another object file, which was the library routine in machine language, defined in the compiler toolchain, would be then processed together with the object file from the assembler into the linker. The linker would generate an executable file for the machine to execute the codes. The loader would read from the executable file and load the machine codes into the memory of the registers in the RISC-V machine.

3.1.3 System Architecture Diagram

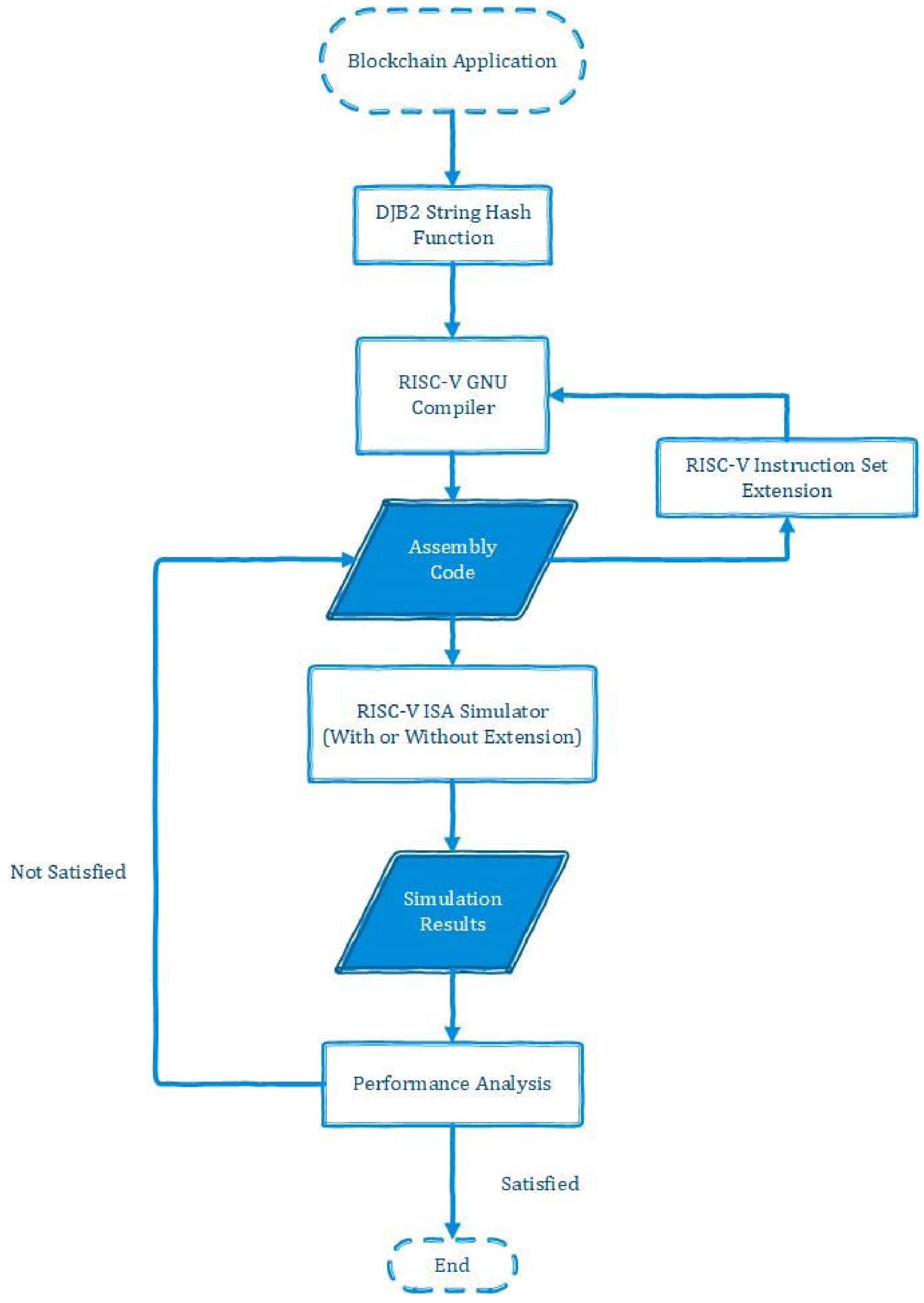


Figure 3.1.3 System Design Implementation Flowchart

The illustration of the system design implementation flow was shown in Figure 3.1.3. The design flow started with completing the source codes of the blockchain application. The development of the blockchain application should include the djb2 string hash function as describe in section 3.1.1. Then, the application would be compiled with the existing RISC-V GNU Toolchain. The compiler toolchain would produce an assembly file with all the RISC-V assembly instructions used in the programme execution. The assembly codes should be studied and analysed, so that the registers used were known. After that, the information would be used to design the custom instruction set extension. Then, the extension would be defined in the compiler toolchain and the toolchain should be rebuilt. The blockchain application should be compiled again and the assembly codes should include the customized instructions. The compiled files would be executable and a RISC-V ISA simulator, with or without the instruction set extension would execute the file. The simulation results would be generated, such as the clock cycles and the number of instructions executed. Then, the analysis would be performed on the results to determine the performance of the system.

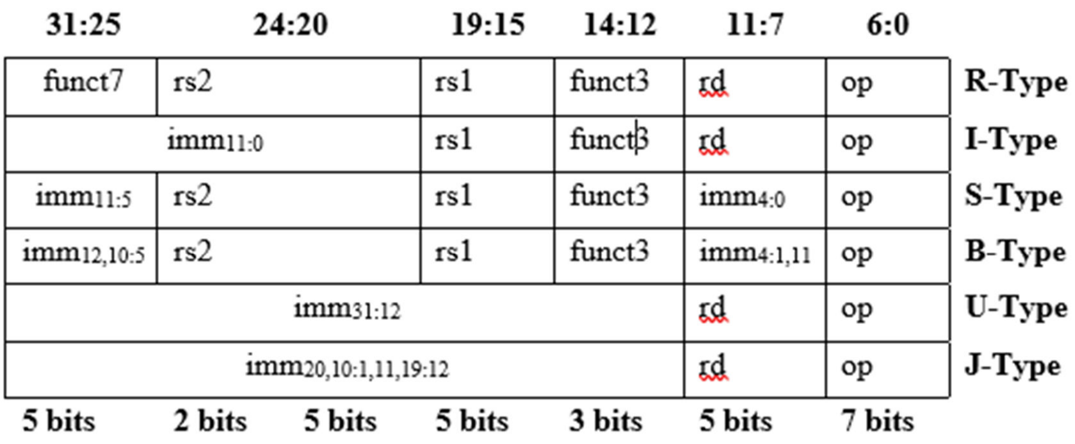
3.2 RISC-V Architecture

RISC-V is an open-source, royalty-free instruction set architecture (ISA) that defines the set of instructions that a computer processor can execute. It is designed to be simple, modular, and highly customizable.

It has a variety of standard extensions:

1) **RV32I/RV64I/RV128I**: The base integer instruction sets with 32, 64, or 128-bit data widths.
2) **M (Integer Multiplication and Division):** Adds instructions for integer multiplication and division.
3) **F (Single-Precision Floating-Point) and D (Double-Precision Floating-Point):** Extensions for floating-point arithmetic.
4) **C (Compressed):** Reduces instruction size by using 16-bit instructions for common operations.
5) **A (Atomic):** Supports atomic memory operations for concurrency control.
6) **V (Vector):** Introduces vector operations for SIMD (Single Instruction, Multiple Data) processing.

For a 32-bit RISC-V machine, it has the following structure for its instructions as shown in Figure 3.2.2. The programme's source codes were required to be translated to the corresponding machine code in 32-bit or 64-bit format for the programme to function correctly.



31:25	24:20	19:15	14:12	11:7	6:0	
funct7	rs2	rs1	funct3	rd	op	**R-Type**
$imm_{11:0}$		rs1	funct3	rd	op	**I-Type**
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	**S-Type**
$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	**B-Type**
$imm_{31:12}$				rd	op	**U-Type**
$imm_{20,10:1,11,19:12}$				rd	op	**J-Type**

5 bits 2 bits 5 bits 5 bits 3 bits 5 bits 7 bits

Figure 3.2.2 RISC-V 32-bit instruction formats

Chapter 4
System Design

4.1 Design of Instruction Set Extension

4.1.1 Block Diagram of DJB2 String Hash Instruction Extension

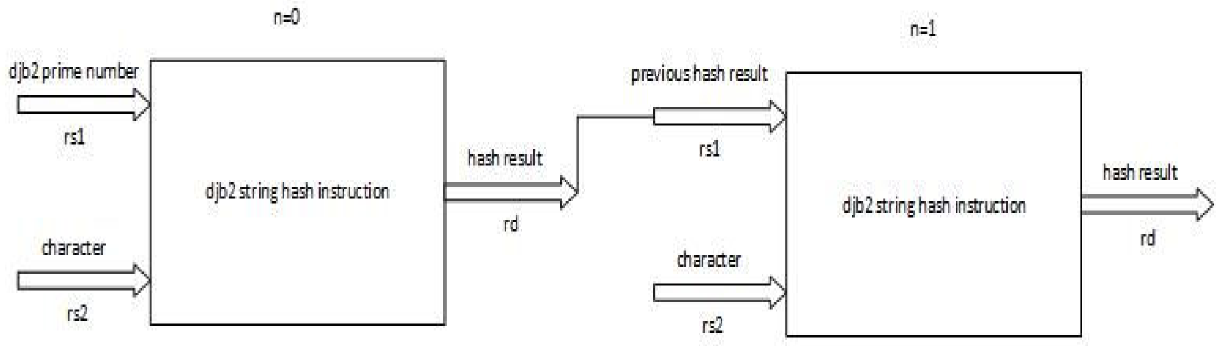
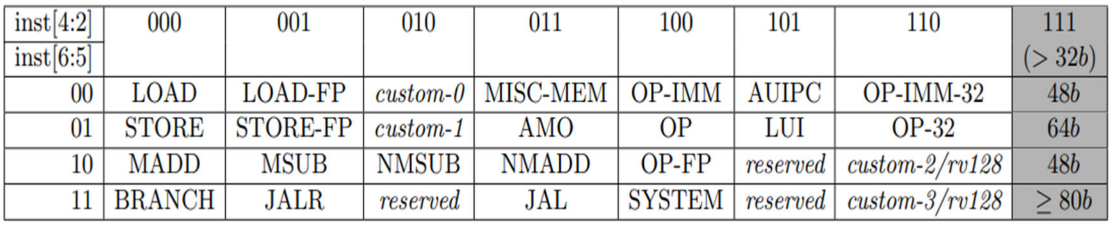


Figure 4.1.1.1 Block Diagram of Instruction Set Extension

As illustrated in the block diagram in Figure 4.1.1.1, the djb2 string hash instruction extension was accepting two inputs from *rs1* and *rs2* registers. The *rs1* stored the djb2 prime number as defined in the djb2 string hash function of the blockchain application in C. The *rs2* stored each ASCII value of the character from a character string. Then, the custom instruction would perform its behaviour to retrieve the value of the hash result, which would be stored into the *rd* register. In the next iteration of character, the hash result from the *rd* register would be loaded into the *rs1* register to perform its calculation again. The process would iterate itself for the next character in the character string until no character was found.

4.1.2 Opcode of DJB2 String Hash Instruction Extension

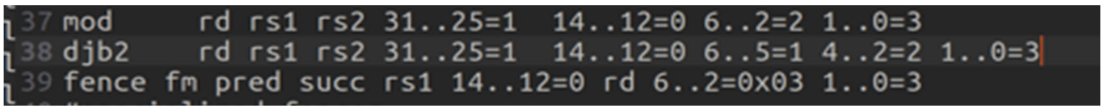


inst[4:2]	000	001	010	011	100	101	110	111
inst[6:5]								(> 32*b*)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48*b*
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	64*b*
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48*b*
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	≥ 80*b*

Figure 4.1.2.1 RISC-V Base Opcode Map

In Figure 4.1.2.1 [18], the opcodes labelled with ‘custom-0’, ‘custom-1’, ‘custom-2’, and ‘custom-3’ were available for the use of custom instruction extension in RISC-V. While there were three slots for reserved, it was better not to consume them as there would be overlapping of opcodes in the future. The ‘custom-1’ opcode was chosen for the custom extension of this project. The value for inst[6:5] was 01b and the value inst[4:2] was 010b, hence these two values would be taken into design the opcode of the custom instruction extension.

The opcode could be designed easily through a tool called 'riscv-opcodes' [19], which could be cloned from its GitHub repo. It would automatically generate the opcode for the RISC-V instruction after the custom instruction format was inserted into one of the files and inputting the terminal command 'make' to build the 'riscv-opcodes' project. As shown in Figure 4.1.2.2, the format of custom instruction 'djb2' was inserted into the *rv_i* file.

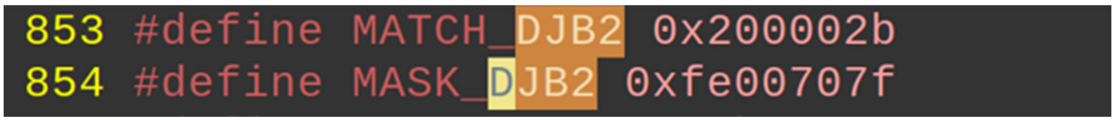


```
mod      rd rs1 rs2 31..25=1  14..12=0 6..2=2 1..0=3
djb2     rd rs1 rs2 31..25=1  14..12=0 6..5=1 4..2=2 1..0=3
fence fm pred succ rs1 14..12=0 rd 6..2=0x03 1..0=3
```

Figure 4.1.2.2 Opcodes of DJB2 String Hash Instruction

To breakdown the format, the 'funct7' from bit 31 to bit 25 has the value of 1, and the 'funct3' from bit 14 to bit 12 has the value of 0. The opcode from bit 6 to bit 0 was separately assigned for each bit to minimize the error of overlapping, i.e. bit 6 to bit 5 has the value of 1, bit 4 to bit 2 has the value of 2, and bit 1 to bit 0 has the value of 3. Finally, the other unassigned bits [24:20] are for rs2, bits [19:15] are for rs1, and bits [11:7] are for rd. The opcode of the instruction defined here was following the RTYPE instruction, which was shown in Figure 3.2.2, Section 3.2.

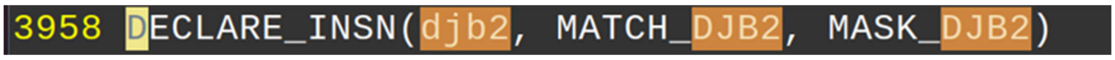
After the 'riscv-opcodes' tool was built, the representation of the opcode of the custom instruction would be generated in *encoding.out.h* file. The match and mask values of the custom instruction would be used in later steps to call the custom instruction.



```
#define MATCH_DJB2 0x200002b
#define MASK_DJB2 0xfe00707f
```

Figure 4.1.2.3 Opcode's Mask and Match Values of Custom Instruction

In the same *encoding.out.h* file, there was a line of code to declare the custom instruction in Figure 4.1.2.4, which will be used in the compiler.



```
DECLARE_INSN(djb2, MATCH_DJB2, MASK_DJB2)
```

Figure 4.1.2.4 Format for the Declaration of Instruction

4.1.3 Pseudocode of DJB2 String Hash Instruction Extension

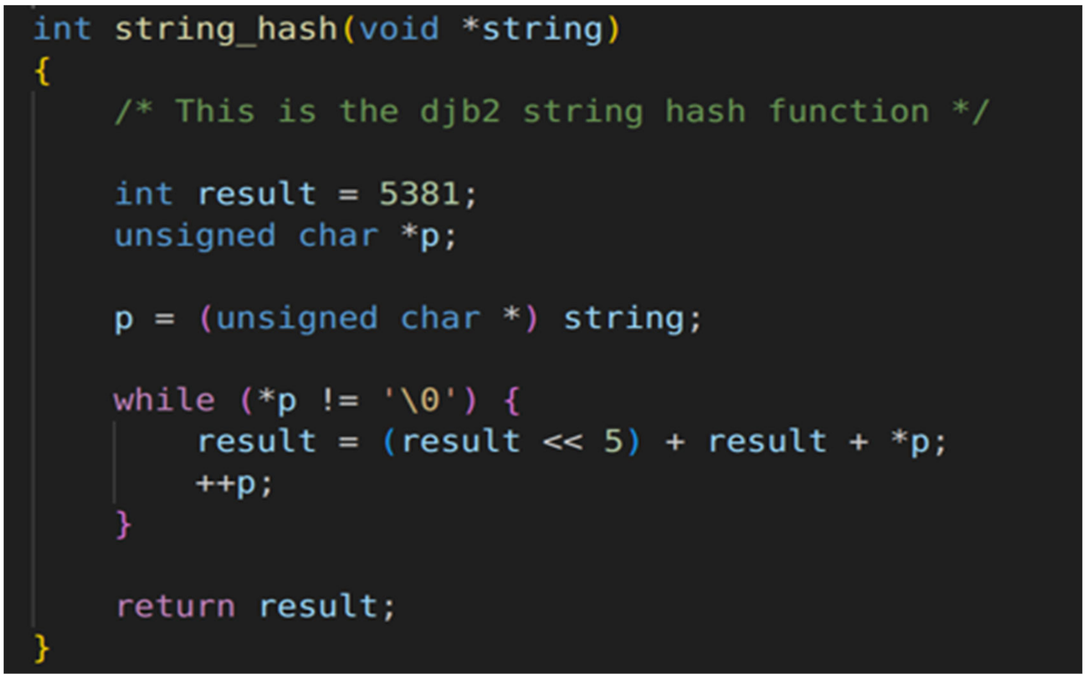
The behavioural design of the djb2 string hash instruction custom extension could be described in this way:

1. Assign the value of the prime number 5381 into *rs1* register if first iteration, else assign the value of *rd* into *rs1*.
2. Shift-left the value inside *rs1* by a factor of 5.
3. Add the shifted value with the value stored in *rs1*.
4. Add the value in *rs1* with the value in *rs2*.
5. Store the final value into *rd.*
6. Repeat Step 1 until all characters are completely hashed.

4.2 Blockchain Application Source Codes

The blockchain application source codes were taken from the GitHub repo 'A-Simple-Blockchain-Simulation-Using-C' [20]. The source codes consist of a *blockchain.c* which acts as the parent file to call the other children files, a *hash.c* file to generate the djb2 hash value for the strings, a *linkedList.c* file to link and chain the blocks together, and some other header files to construct the classes. The full source codes were included in Appendix C.

The *hash.c* file was the main file to make modifications for the project. The code snippet is shown in Figure 4.2. A string would be passed from the parent file *blockchain.c* into the function defined in *string_hash* to generate the djb2 string hash value. After all the characters from the string were executed with the string hash function, the final value of the hash result would be returned to the caller.

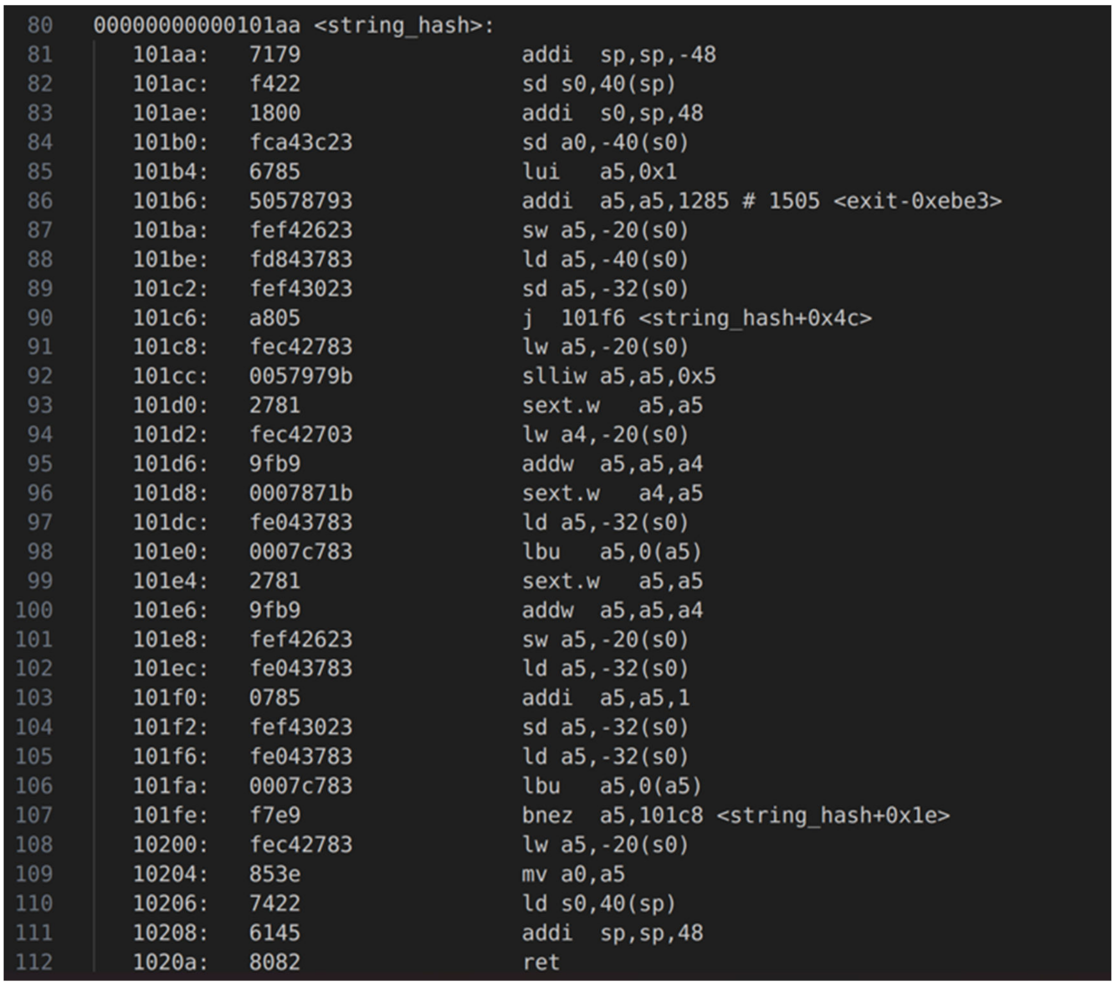


```c
int string_hash(void *string)
{
    /* This is the djb2 string hash function */

    int result = 5381;
    unsigned char *p;

    p = (unsigned char *) string;

    while (*p != '\0') {
        result = (result << 5) + result + *p;
        ++p;
    }

    return result;
}
```

Figure 4.2 Code Snippet of hash.c

4.2.1 Assembly Codes of String Hash Function

The assembly codes of the string hash function were as shown in Figure 4.2.1. The first column represented the program counter of the RISC-V machine, which would be incremented by 4 for every cycle. The second column represented the value of instruction code in hexadecimal format. The third column represented the human-readable instructions as decoded from the hexadecimal instructions from second column.

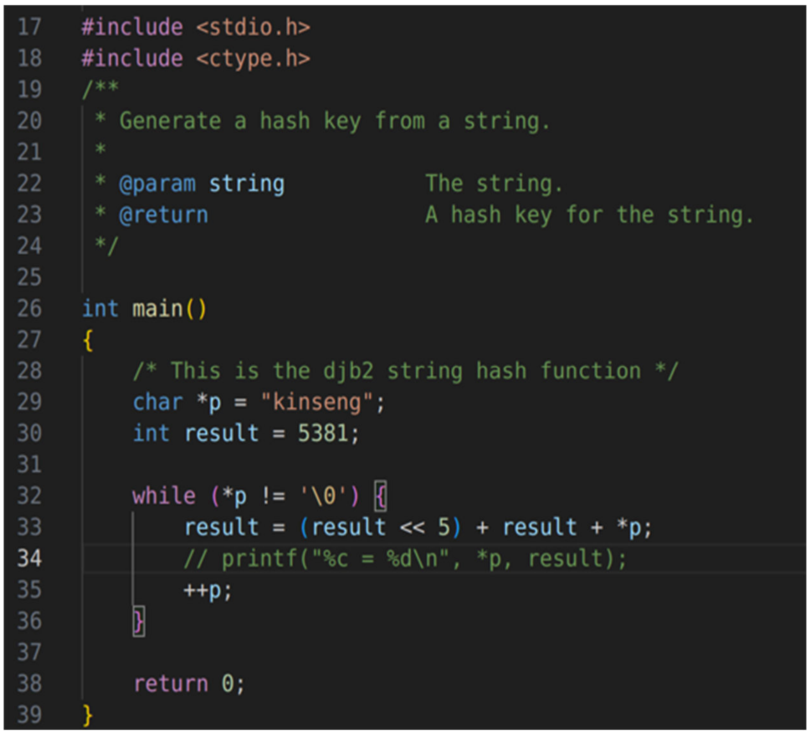


```
00000000000101aa <string_hash>:
   101aa:   7179                  addi  sp,sp,-48
   101ac:   f422                  sd s0,40(sp)
   101ae:   1800                  addi  s0,sp,48
   101b0:   fca43c23              sd a0,-40(s0)
   101b4:   6785                  lui   a5,0x1
   101b6:   50578793              addi  a5,a5,1285 # 1505 <exit-0xebe3>
   101ba:   fef42623              sw a5,-20(s0)
   101be:   fd843783              ld a5,-40(s0)
   101c2:   fef43023              sd a5,-32(s0)
   101c6:   a805                  j  101f6 <string_hash+0x4c>
   101c8:   fec42783              lw a5,-20(s0)
   101cc:   0057979b              slliw a5,a5,0x5
   101d0:   2781                  sext.w   a5,a5
   101d2:   fec42703              lw a4,-20(s0)
   101d6:   9fb9                  addw  a5,a5,a4
   101d8:   0007871b              sext.w   a4,a5
   101dc:   fe043783              ld a5,-32(s0)
   101e0:   0007c783              lbu   a5,0(a5)
   101e4:   2781                  sext.w   a5,a5
   101e6:   9fb9                  addw  a5,a5,a4
   101e8:   fef42623              sw a5,-20(s0)
   101ec:   fe043783              ld a5,-32(s0)
   101f0:   0785                  addi  a5,a5,1
   101f2:   fef43023              sd a5,-32(s0)
   101f6:   fe043783              ld a5,-32(s0)
   101fa:   0007c783              lbu   a5,0(a5)
   101fe:   f7e9                  bnez  a5,101c8 <string_hash+0x1e>
   10200:   fec42783              lw a5,-20(s0)
   10204:   853e                  mv a0,a5
   10206:   7422                  ld s0,40(sp)
   10208:   6145                  addi  sp,sp,48
   1020a:   8082                  ret
```

Figure 4.2.1 Assembly Codes of *hash.c*

4.3 Unit Test Programmes

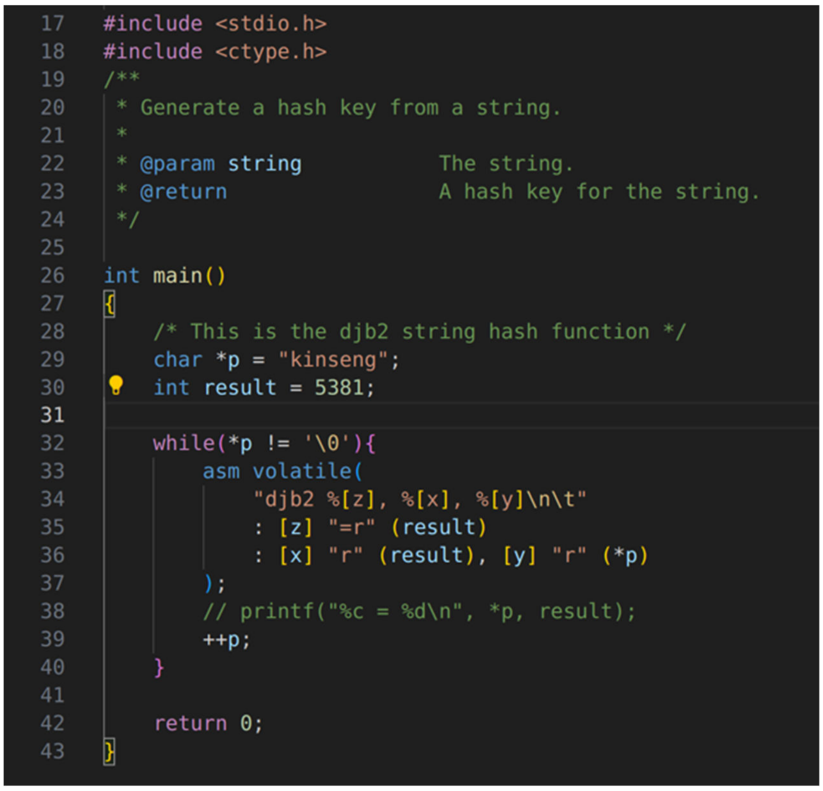
Two unit test programmes were written to test and validate the compiler without the custom instruction extension and with the extension. The first unit test was named *test_hash.c* and its purpose was to test the compiler without the custom instruction. The code snippet of *test_hash.c* was shown in Figure 4.3.1.



```c
#include <stdio.h>
#include <ctype.h>
/**
 * Generate a hash key from a string.
 *
 * @param string            The string.
 * @return                  A hash key for the string.
 */

int main()
{
    /* This is the djb2 string hash function */
    char *p = "kinseng";
    int result = 5381;

    while (*p != '\0') {
        result = (result << 5) + result + *p;
        // printf("%c = %d\n", *p, result);
        ++p;
    }

    return 0;
}
```

Figure 4.3.1 Code Snippet of *test_hash.c*

The second unit test was to test the compiler with the extended instruction and was named *test_hash_asm.c*. The source code was as shown in Figure 4.3.2. To directly call the custom instruction, the function *asm volatile( )* should be used. Then, write the instruction name with its register variables. In this case, "djb2 %[z], %[x], %[y]" would be equal to "djb2 rd, rs1, rs2". The line '[z] "=r" (result)' would represent that the write operation to *rd* register after the *result* was computed. The line '[x] "r" (result), [y] "r" (*p)' represented the value of *result* to be assigned to *rs1* register and the value of the current character **p* to be assigned to *rs2* register. The computing operation of the custom instruction would be done on back-end side of the RISC-V ISA Simulator, for which the Spike tool was chosen as the simulator in this project.



```c
#include <stdio.h>
#include <ctype.h>
/**
 * Generate a hash key from a string.
 *
 * @param string            The string.
 * @return                  A hash key for the string.
 */

int main()
{
    /* This is the djb2 string hash function */
    char *p = "kinseng";
    int result = 5381;

    while(*p != '\0'){
        asm volatile(
            "djb2 %[z], %[x], %[y]\n\t"
            : [z] "=r" (result)
            : [x] "r" (result), [y] "r" (*p)
        );
        // printf("%c = %d\n", *p, result);
        ++p;
    }

    return 0;
}
```

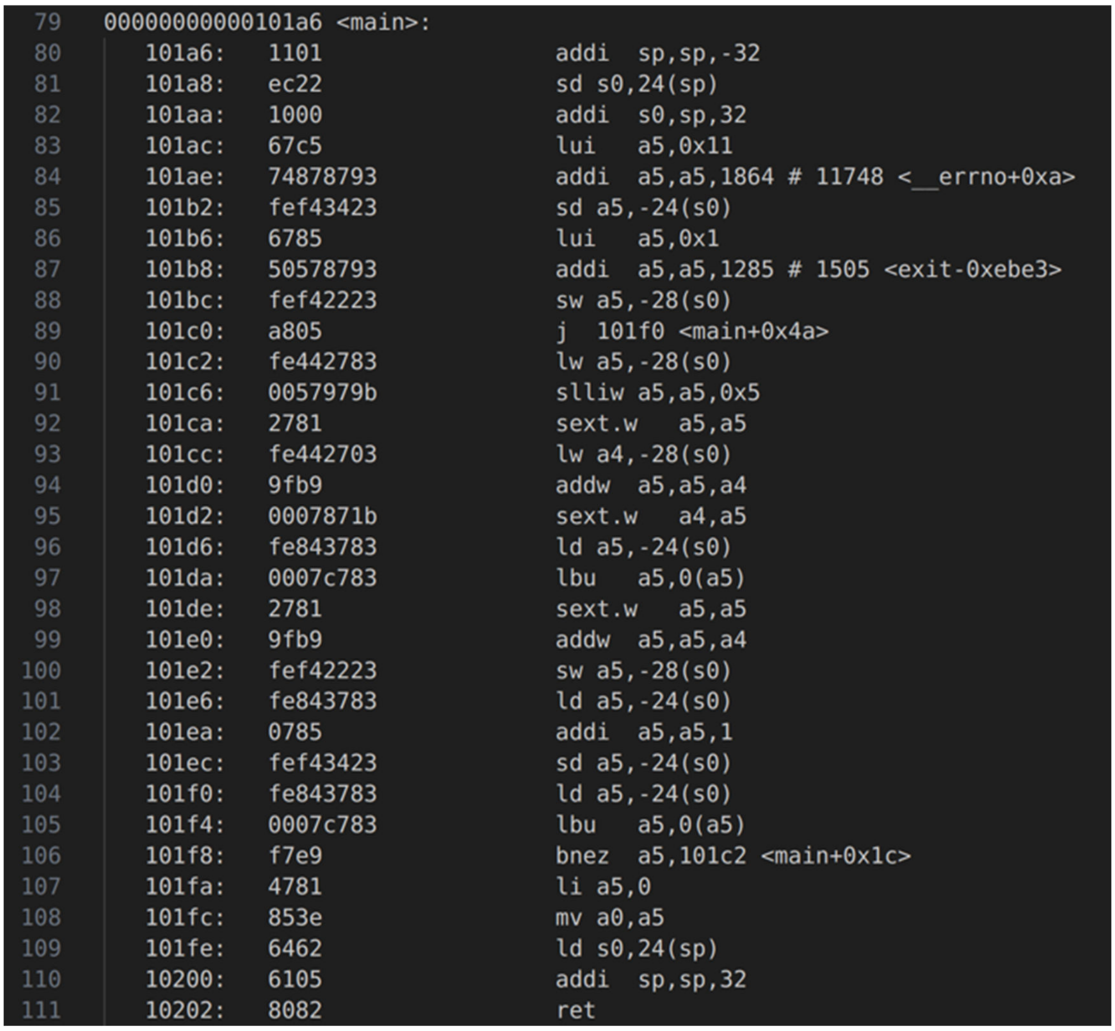
Figure 4.3.2 Code Snippet of *test_hash_asm.c*

4.3.1 Assembly Codes of Unit Test Programmes

After the installation of the RISC-V GNU Toolchain, the following command line could produce the assembly codes of the programme and output to a textfile.

```
Riscv64-unknown-elf-objdump -D filename > dumpfile.txt
```

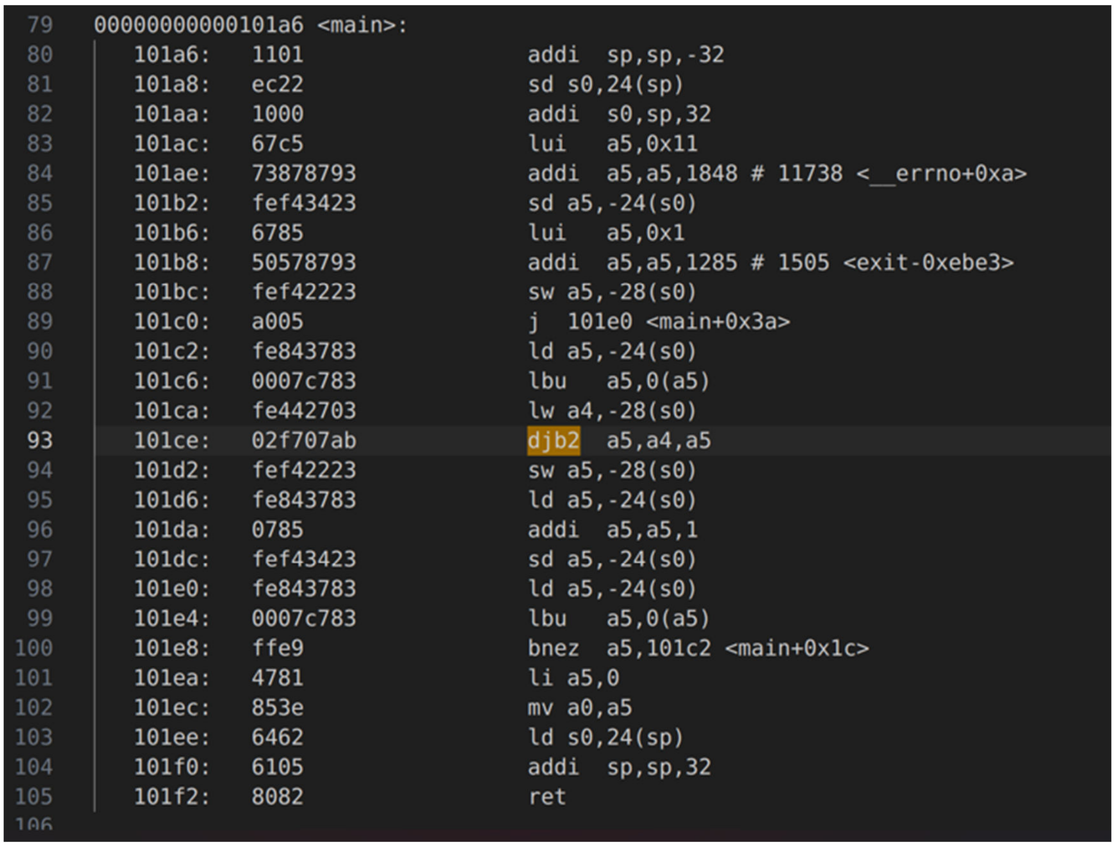
The assembly codes of the first unit test programme without the custom instruction were shown in Figure 4.3.1.1. The custom instruction of djb2 hash function was no where to be found in the assembly code of the test programme.



```
00000000000101a6 <main>:
   101a6:   1101                addi  sp,sp,-32
   101a8:   ec22                sd s0,24(sp)
   101aa:   1000                addi  s0,sp,32
   101ac:   67c5                lui   a5,0x11
   101ae:   74878793            addi  a5,a5,1864 # 11748 <__errno+0xa>
   101b2:   fef43423            sd a5,-24(s0)
   101b6:   6785                lui   a5,0x1
   101b8:   50578793            addi  a5,a5,1285 # 1505 <exit-0xebe3>
   101bc:   fef42223            sw a5,-28(s0)
   101c0:   a805                j  101f0 <main+0x4a>
   101c2:   fe442783            lw a5,-28(s0)
   101c6:   0057979b            slliw a5,a5,0x5
   101ca:   2781                sext.w   a5,a5
   101cc:   fe442703            lw a4,-28(s0)
   101d0:   9fb9                addw  a5,a5,a4
   101d2:   0007871b            sext.w   a4,a5
   101d6:   fe843783            ld a5,-24(s0)
   101da:   0007c783            lbu   a5,0(a5)
   101de:   2781                sext.w   a5,a5
   101e0:   9fb9                addw  a5,a5,a4
   101e2:   fef42223            sw a5,-28(s0)
   101e6:   fe843783            ld a5,-24(s0)
   101ea:   0785                addi  a5,a5,1
   101ec:   fef43423            sd a5,-24(s0)
   101f0:   fe843783            ld a5,-24(s0)
   101f4:   0007c783            lbu   a5,0(a5)
   101f8:   f7e9                bnez  a5,101c2 <main+0x1c>
   101fa:   4781                li a5,0
   101fc:   853e                mv a0,a5
   101fe:   6462                ld s0,24(sp)
   10200:   6105                addi  sp,sp,32
   10202:   8082                ret
```

Figure 4.3.1.1 Assembly Codes of Unit Test without Extension

In figure 4.3.1.2, the assembly codes of the second unit test programme with the custom instruction were extracted into the assembly dump file. The custom instruction djb2 was successfully called by the compiler, which would also mean the success of extending the compiler with the custom instruction.



```
00000000000101a6 <main>:
   101a6:	1101                	addi	sp,sp,-32
   101a8:	ec22                	sd	s0,24(sp)
   101aa:	1000                	addi	s0,sp,32
   101ac:	67c5                	lui	a5,0x11
   101ae:	73878793          	addi	a5,a5,1848 # 11738 <__errno+0xa>
   101b2:	fef43423          	sd	a5,-24(s0)
   101b6:	6785                	lui	a5,0x1
   101b8:	50578793          	addi	a5,a5,1285 # 1505 <exit-0xebe3>
   101bc:	fef42223          	sw	a5,-28(s0)
   101c0:	a005                	j	101e0 <main+0x3a>
   101c2:	fe843783          	ld	a5,-24(s0)
   101c6:	0007c783          	lbu	a5,0(a5)
   101ca:	fe442703          	lw	a4,-28(s0)
   101ce:	02f707ab          	djb2	a5,a4,a5
   101d2:	fef42223          	sw	a5,-28(s0)
   101d6:	fe843783          	ld	a5,-24(s0)
   101da:	0785                	addi	a5,a5,1
   101dc:	fef43423          	sd	a5,-24(s0)
   101e0:	fe843783          	ld	a5,-24(s0)
   101e4:	0007c783          	lbu	a5,0(a5)
   101e8:	ffe9                	bnez	a5,101c2 <main+0x1c>
   101ea:	4781                	li	a5,0
   101ec:	853e                	mv	a0,a5
   101ee:	6462                	ld	s0,24(sp)
   101f0:	6105                	addi	sp,sp,32
   101f2:	8082                	ret
```

Figure 4.3.1.2 Assembly Codes of Unit Test with Extension

Chapter 5
Experiment/Simulation

5.1 Hardware Setup

A desktop PC with Linux OS was setup. A computer issued for the process of building RISC-V simulation tools, RISC-V toolchains, and the blockchain application. The details of the PC specifications were as shown in Table 5.1.1

Table 5.1.1 Specifications of PC

Description	Specifications
Model	Asus B85-Pro Gamer
Processor	Intel Core i5-4400
Operating System	Ubuntu 22.04.3 LTS
Graphic	NVIDIA GeForce GTX 750TI 2GB GRAM
Memory	16GB DDR3 RAM
Storage	1TB SATA SSD

5.2 Software Setup

In this project, there are few software tools needed to be downloaded and installed in the PC:

1. Ubuntu 22.04.3 LTS
2. Chipyard: 1.10.0 (https://chipyard.readthedocs.io/en/stable/)
3. RISC-V GNU Toolchain (https://github.com/riscv-collab/riscv-gnu-toolchain/)
4. Spike RISC-V ISA Simulator from Chipyard (https://github.com/riscv-software-src/riscv-isa-sim/)
5. RISC-V Opcodes (https://github.com/riscv/riscv-opcodes)
6. Visual Studio Code

5.3 Setting and Configuration

There were several settings and configurations required to be done each time the source code was compiled. In this project, the optimization options were included during the compilation of the blockchain application source codes. The optimization options have different usages respectively and the results would also be affected. There were five optimization options used which were O0, O1, O2, O3, and Os. The usage of the optimization flags was described in Table 5.3.1.

Table 5.3.1 Usage of Optimization Flags

Optimization Flags	Usage
O0	• Compilation time is reduced. • No code optimization.
O1	• Code size is reduced. • Execution time for small functions is reduced.
O2	• More optimization. • Code size is reduced even more. • Execution time is improved.
O3	• Highest optimization. • Code size is reduced greatly. • Execution time is greatly improved.
Os	• Only reduce code size.

5.4 System Operation

5.4.1 Source Code Compilation

The blockchain application source codes in C could be compiled with the GCC compiler from the RISC-V GNU Toolchain. To compile the source codes, the gcc tool should be called in the terminal. The following commands were examples that were used:

```
riscv64-unknown-elf-gcc -O0 -c blockchain.c hash.c linkedList.c -o O0_main.riscv
riscv64-unknown-elf-gcc -O1 -c blockchain.c hash.c linkedList.c -o O1_main.riscv
riscv64-unknown-elf-gcc -O2 -c blockchain.c hash.c linkedList.c -o O2_main.riscv
riscv64-unknown-elf-gcc -O3 -c blockchain.c hash.c linkedList.c -o O3_main.riscv
riscv64-unknown-elf-gcc -O4 -c blockchain.c hash.c linkedList.c -o Os_main.riscv
```

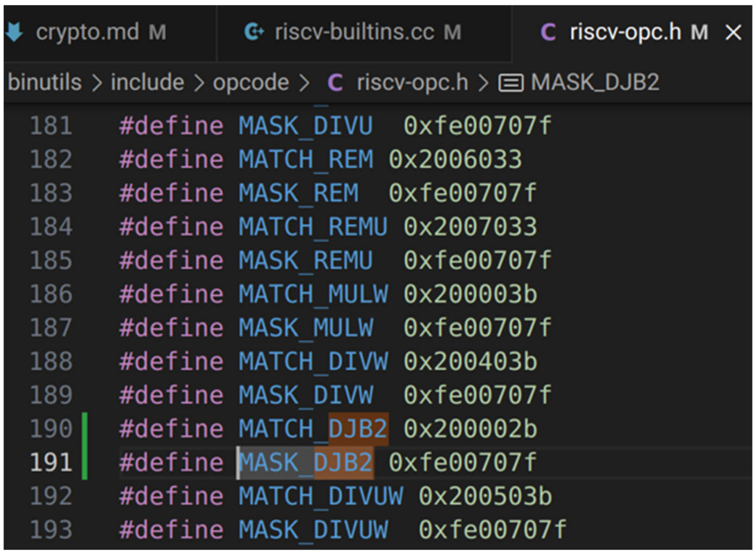
5.4.2 Extending RISC-V GNU Toolchain

RISC-V architecture was highly customizable and extensible, and the same would go for its RISC-V GNU compiler toolchain. The RISC-V GNU Toolchain was pre-installed if Chipyard was used in the project. Otherwise, the toolchain could be manually built and installed by referring to the official GitHub repo "riscv-gnu-toolchain" [21].

Since the source codes of blockchain application were written in C, only the GCC could compile all the codes into RISC-V executable. The customization and extension of the GCC compiler could be done in these few steps:

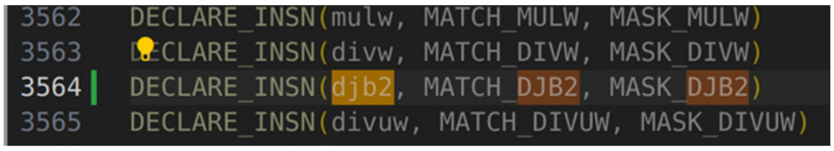
Step 1: Open the file *riscv-opc.h* located in riscv-gnu-toolchain/binutils/include/opcode/

Step 2: Add the match and mask values of the instruction, as well as the declaration of instruction into the file as shown in Figure 4.3.1 and Figure 4.3.2.



```
#define MASK_DIVU  0xfe00707f
#define MATCH_REM 0x2006033
#define MASK_REM  0xfe00707f
#define MATCH_REMU 0x2007033
#define MASK_REMU  0xfe00707f
#define MATCH_MULW 0x200003b
#define MASK_MULW  0xfe00707f
#define MATCH_DIVW 0x200403b
#define MASK_DIVW  0xfe00707f
#define MATCH_DJB2 0x200002b
#define MASK_DJB2 0xfe00707f
#define MATCH_DIVUW 0x200503b
#define MASK_DIVUW  0xfe00707f
```

Figure 4.3.1 Adding Match and Mask Values into *riscv-opc.h*

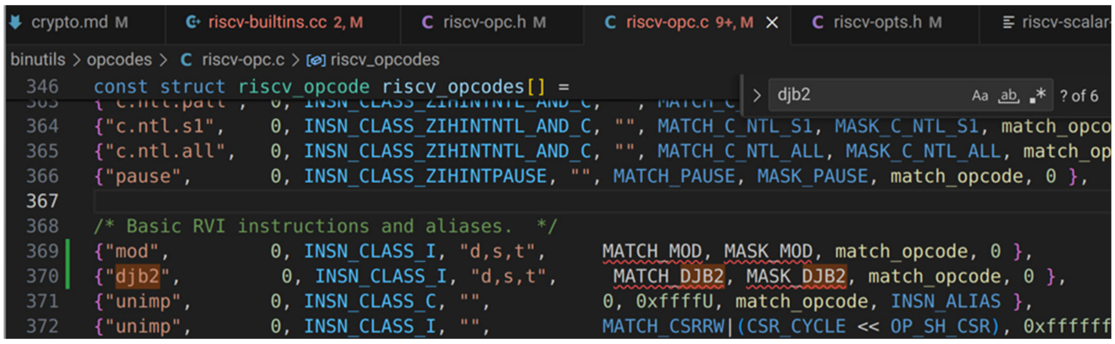


```
DECLARE_INSN(mulw, MATCH_MULW, MASK_MULW)
DECLARE_INSN(divw, MATCH_DIVW, MASK_DIVW)
DECLARE_INSN(djb2, MATCH_DJB2, MASK_DJB2)
DECLARE_INSN(divuw, MATCH_DIVUW, MASK_DIVUW)
```

Figure 4.3.2 Declaration of Instruction in *riscv-opc.h*

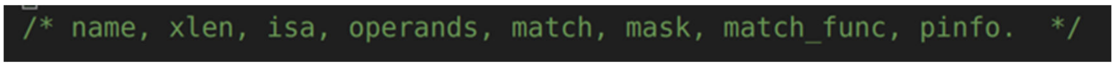
Step 3: Open the file *riscv-opc.c* located in riscv-gnu-toolchain/binutils/opcodes

Step 4: Add the line into the function ‘const struct riscv_opcode riscv_opcodes[]’ in the file as shown in Figure 4.3.3. The format of the declaration structure was described Figure 4.3.4. More explanations about each parameter required in the format could be found in Appendix D.



```
const struct riscv_opcode riscv_opcodes[] =
{"c.ntl.s1",    0, INSN_CLASS_ZIHINTNTL_AND_C, "", MATCH_C_NTL_S1, MASK_C_NTL_S1, match_opco
{"c.ntl.all",   0, INSN_CLASS_ZIHINTNTL_AND_C, "", MATCH_C_NTL_ALL, MASK_C_NTL_ALL, match_op
{"pause",       0, INSN_CLASS_ZIHINTPAUSE, "", MATCH_PAUSE, MASK_PAUSE, match_opcode, 0 },

/* Basic RVI instructions and aliases.  */
{"mod",         0, INSN_CLASS_I, "d,s,t",     MATCH_MOD, MASK_MOD, match_opcode, 0 },
{"djb2",         0, INSN_CLASS_I, "d,s,t",     MATCH_DJB2, MASK_DJB2, match_opcode, 0 },
{"unimp",       0, INSN_CLASS_C, "",          0, 0xffffU, match_opcode, INSN_ALIAS },
{"unimp",       0, INSN_CLASS_I, "",          MATCH_CSRRW|(CSR_CYCLE << OP_SH_CSR), 0xffffff
```

Figure 4.3.3 Defining Instruction Behaviour in *riscv-opc.c*



```
/* name, xlen, isa, operands, match, mask, match_func, pinfo.  */
```

Figure 4.3.4 Format of *riscv_opcodes* struct

Step 5: Rebuild the RISC-V GNU Toolchain using the following command.

```
./configure –prefix=$(RISCV) –with-cmodel=medany
sudo make clean
sudo make -j$(nproc)
```

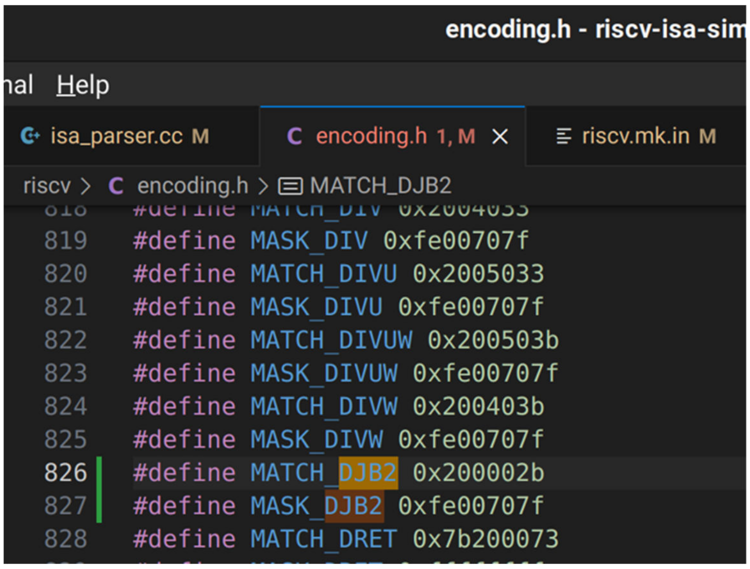
5.4.3 Modification of Spike ISA Simulator

Spike was a RISC-V ISA software simulator that could provide an emulation mimicking a RISC-V CPU machine. Spike was also highly customizable and extensible in which the developers could modify, add, and test new instructions.

In this project, the custom instruction set could be added to the Spike simulator in these few steps:

Step 1: Change directory to chipyard/toolchains/riscv-tools/riscv-isa-sim

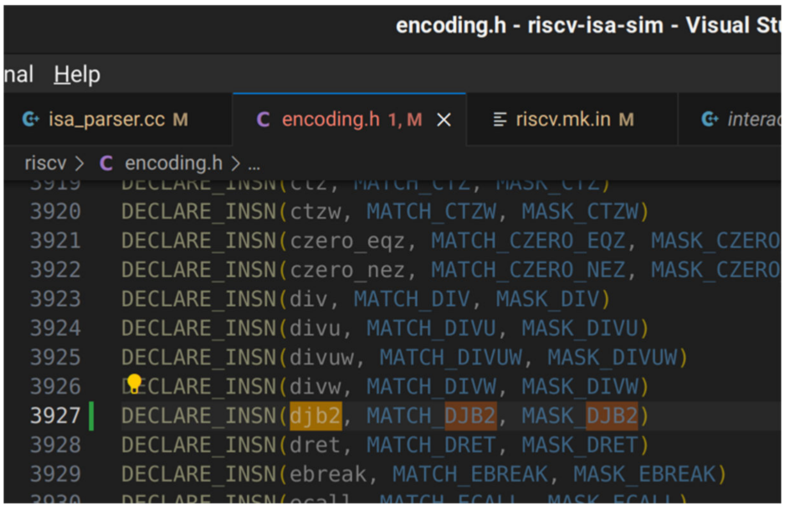
Step 2: Add match and mask values of custom instruction into riscv/encoding.h as shown in Figure 5.4.3.1



```
#define MASK_DIV 0xfe00707f
#define MATCH_DIVU 0x2005033
#define MASK_DIVU 0xfe00707f
#define MATCH_DIVUW 0x200503b
#define MASK_DIVUW 0xfe00707f
#define MATCH_DIVW 0x200403b
#define MASK_DIVW 0xfe00707f
#define MATCH_DJB2 0x200002b
#define MASK_DJB2 0xfe00707f
#define MATCH_DRET 0x7b200073
```

Figure 5.4.3.1 Adding Match and Mask Values of DJB2 Hash Instruction in *encoding.h*

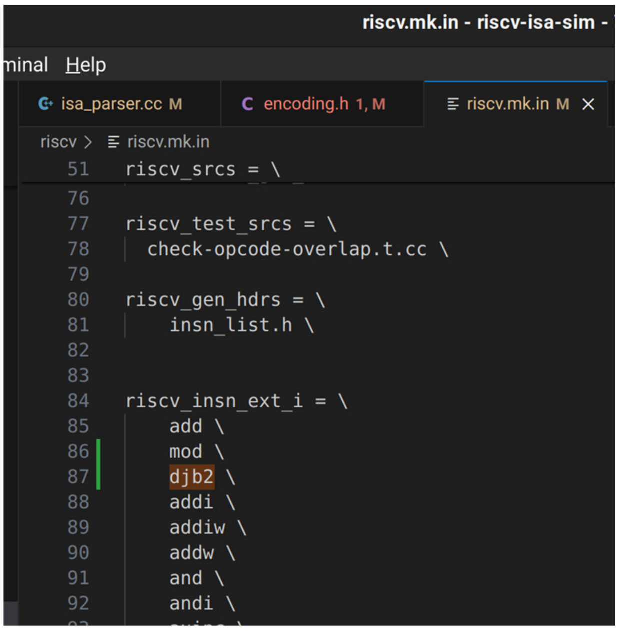
Step 3: Add the declaration of instruction in the same *encoding.h* file as shown in Figure 5.4.3.2.



```
DECLARE_INSN(ctzw, MATCH_CTZW, MASK_CTZW)
DECLARE_INSN(czero_eqz, MATCH_CZERO_EQZ, MASK_CZERO
DECLARE_INSN(czero_nez, MATCH_CZERO_NEZ, MASK_CZERO
DECLARE_INSN(div, MATCH_DIV, MASK_DIV)
DECLARE_INSN(divu, MATCH_DIVU, MASK_DIVU)
DECLARE_INSN(divuw, MATCH_DIVUW, MASK_DIVUW)
DECLARE_INSN(divw, MATCH_DIVW, MASK_DIVW)
DECLARE_INSN(djb2, MATCH_DJB2, MASK_DJB2)
DECLARE_INSN(dret, MATCH_DRET, MASK_DRET)
DECLARE_INSN(ebreak, MATCH_EBREAK, MASK_EBREAK)
```

Figure 5.4.3.2 Adding Declaration of DJB2 Hash Instruction in *encoding.h*

Step 4: Add the custom instruction into the array of “riscv_insn_ext_i” in *riscv.mk.in* file under the current directory as illustrated in Figure 5.4.3.3.

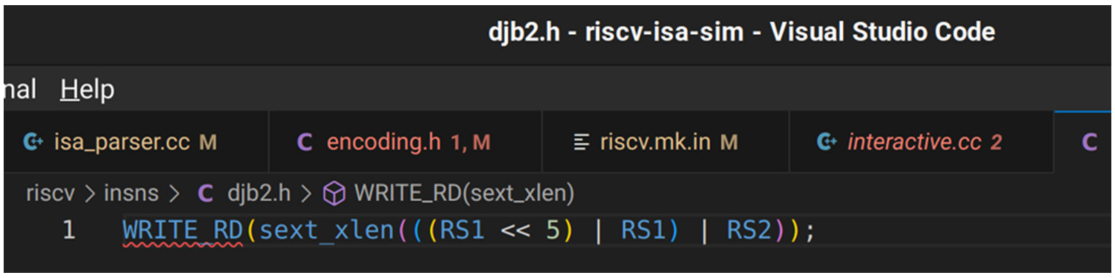


```
riscv_srcs = \

riscv_test_srcs = \
  check-opcode-overlap.t.cc \

riscv_gen_hdrs = \
    insn_list.h \


riscv_insn_ext_i = \
    add \
    mod \
    djb2 \
    addi \
    addiw \
    addw \
    and \
    andi \
```

Figure 5.4.3.3 Adding DJB2 Hash Instruction in *riscv.mk.in*

Step 5: Create a new header file with the name of the instruction in the directory *riscv/insn/*. Example: djb2.h

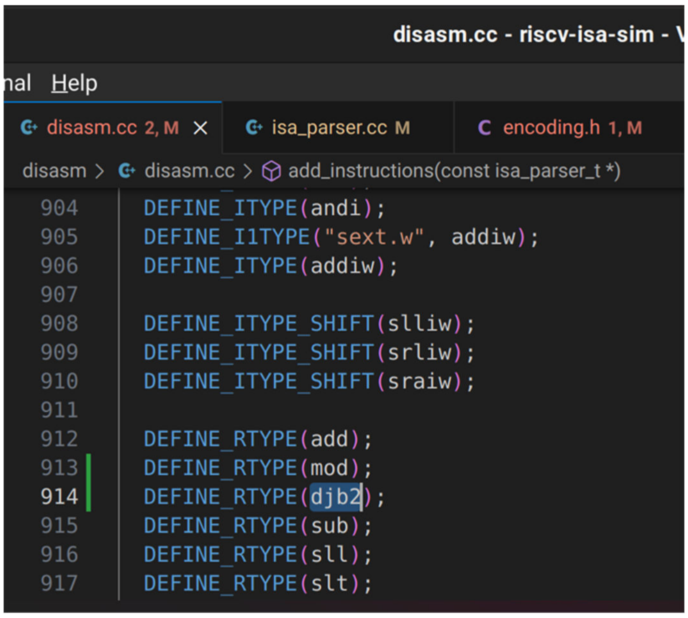
Step 6: Write the behaviour of the custom instruction in the header file that has just been created as shown in Figure 5.4.3.4.


djb2.h - riscv-isa-sim - Visual Studio Code
nal Help
isa_parser.cc M
encoding.h 1, M
riscv.mk.in M
interactive.cc 2
riscv > insns > djb2.h > WRITE_RD(sext_xlen)

```
WRITE_RD(sext_xlen(((RS1 << 5) | RS1) | RS2));
```

Figure 5.4.3.4 Functional Behaviour of Custom Instruction.

Step 7: Add the RISC-V format type of instruction in *riscv-isa-sim/disasm/disasm.cc* as seen in Figure 5.4.3.5.


disasm.cc - riscv-isa-sim - V
nal Help
disasm.cc 2, M
isa_parser.cc M
encoding.h 1, M
disasm > disasm.cc > add_instructions(const isa_parser_t *)

```
DEFINE_ITYPE(andi);
DEFINE_I1TYPE("sext.w", addiw);
DEFINE_ITYPE(addiw);

DEFINE_ITYPE_SHIFT(slliw);
DEFINE_ITYPE_SHIFT(srliw);
DEFINE_ITYPE_SHIFT(sraiw);

DEFINE_RTYPE(add);
DEFINE_RTYPE(mod);
DEFINE_RTYPE(djb2);
DEFINE_RTYPE(sub);
DEFINE_RTYPE(sll);
DEFINE_RTYPE(slt);
```

Figure 5.4.3.5 Defining the RISC-V Format Type of Custom Instruction

5.5 Implementation Issues and Challenges

During the installation of Chipyard, an issue might arise due to the absence of ‘guestmount’ module. It could be solved by entering the command ‘sudo apt-get install libguestfs-tools’. Nevertheless, the installation might take some time if the CPU and RAM are not powerful. The diskspace consumed for the project was around 10+ GB. A low performing CPU could also delay the time to rebuild the RISC-V GNU Toolchain and Spike ISA Simulator, as these tools contained a tremendous number of files.

Besides, the syntax of the code lines should be carefully typed and inserted as a lack of correct syntax could lead to the failure of rebuilding the customized tools for the research. The consequence of this would consume extra time to look for the errors and fix them.

Furthermore, the instruction extension should not be added into the RISC-V ISA profile ‘I’ as the extension profile was meant for integer operation. However, adding an extra instruction set extension class into the tools was tedious and complicated, as it required a lot more steps to complete the behaviour of the instructions and simulation.

5.6 Concluding Remark

In this project, the main tool used for simulation was Chipyard. Several tools were included when Chipyard was built, such as the RISC-V GNU Toolchain and Spike ISA Simulator. When compiling the source codes of the blockchain application, a few gcc optimization techniques were used, such as the -Os, -O0, -O1, -O2, and -O3. To extend the RISC-V GNU Toolchain such that it was able to compile the codes with custom instruction, some files are required to be modified. For instance, adding the opcodes in *riscv-opc.h* and declaring the instruction in *riscv-opc.c.* Finally, the customization of Spike ISA Simulator was required to simulate the execution of the software in the Spike device. To modify the simulator, quite a few steps are required to be completed. For example, creating a new header file for the custom instruction and defining its behaviour in the file, adding the opcode in *encoding.h*, adding the instruction in *riscv.mk.in*, and defining the format type of the instruction in *disasm.cc*.

Chapter 6
System Evaluation and Discussion

6.1 System Testing and Performance Metrics

A performance analysis of the Spike ISA simulator with custom instruction extension along with the simulator without any modification was conducted to determine the impact of the project. There were a number of information that the simulator could provide which could be taken as the performance metrics of this project. The performance metrics are the number of instructions executed and the program execution time.

The program execution time was a common metric to determine the time needed to execute the program in the CPU. The formula to calculate the program execution time was given as below:

$$Program\ Execution\ Time = \frac{(number\ of\ instructions) \times CPI}{clock\ ticks}$$

6.2 Testing Setup and Result

The blockchain application was compiled with different optimization flags as discussed in section 5.3. A total of 10 RISC-V executables were generated to perform the test. The cycles per instruction (CPI) was set to maintain at 1.00 in the simulator. To use the simulator to execute the programme and to generate the simulation result, the following command was entered:

```
spike pk -s <filename>
```

The result was collected and organized in Table 6.2.1. The evidence of the results was appended in Appendix E.

Table 6.2.1 Simulation Results

O Flags	Base (without extension)		With custom extension		Reduction in instructions (%)	Reduction in time (%)
	# Instructions	Execution Time (ms)	# Instructions	Execution Time (ms)		

-Os	41018	45.58	41013	45.57	0	0
-O0	55329	48.11	51855	47.14	-6.28	-2.02
-O1	40993	45.55	40413	44.90	-1.41	-1.43
-O2	41408	43.59	40828	42.98	-1.40	-1.40
-O3	41408	43.59	40828	42.98	-1.40	-1.40

Based on the data in Table 6.2.1, the blockchain application compiled with -Os flag had a very insignificant difference, which resulted in a 0% reduction in both instructions size and execution time. The application compiled with -O0 had the most significant impact, which resulted in -6.28% in instructions size and -2.02% in execution time. While the programme compiled with -O1 had a difference of -1.41% and -1.43% in both instruction sizes and execution time respectively. Finally, the -O2 and -O3 flags caused the programme to have a reduction of -1.40% in both instruction sizes and execution time.

6.3 Project Challenges

The Spike ISA simulator was a RISC-V instruction set simulator. The custom instruction sets were tested in the software simulator first, before developing it with the hardware changes. The Spike ISA simulator was not a cycle-accurate simulator after all. Therefore, the clock cycles and ticks generated from the simulator may not be accurate to act as a reference for the results. However, it showed the custom instruction extension showed performance improvement and was ready to be implemented with hardware changes.

6.4 Objectives Evaluation

The project was successful to achieve all the mentioned objectives in section 1.2. Firstly, a functioning instruction set extension was successfully designed and run in the simulator for the blockchain hash algorithm. Then, the code size was able to be reduced with the customization, which could be seen most obviously in the compiled application with -O0 optimization flag. Finally, the custom instruction set was able to reduce the clock cycles as well as the program execution time of the blockchain application.

6.5 Concluding Remark

The performance of the custom instruction was satisfied. It was able to improve the performance in both instruction code sizes and program execution time by at least 1.40%. However, the simulation results may not be as precise as it seemed as the Spike ISA simulator was not a cycle-accurate simulator, instead it was just a instruction set simulator to test the functionality of the custom instruction extension.

Chapter 7
Conclusion and Recommendation

7.1 Conclusion

Blockchain technology implementation in real-world applications had a challenge in which the time consumption to add a new block to its blockchains was high. To improve the time and cost efficiency of calculating its hash and thus speed up the addition of blocks in the blockchain, a custom instruction set was proposed as an extension to the RISC-V base ISA. RISC-V architecture was chosen in this project due to its open source nature, highly customizable, and availability of a large pool of community developers. In this project, a simple blockchain application was selected as a case study for its cryptographic algorithms. The hash function used was DJB2 string hash. The proposed instruction sets would be added to the RISC-V ISA to reduce the number of code sizes and to improve the performance of the program execution time of blockchain algorithm. Thus, the extended RISC-V CPU with the proposed extensions would be more efficient and effective in terms of execution time when running the blockchain application with DJB2 string hash. The impact of the custom instruction set extension on the performance was simulated in Spike ISA Simulator and its results are presented in Chapter 6.

7.2 Recommendation

The project could be further improved by modifying the hardware behaviours in the RTL design. A RISC-V hardware that was functioning and able to fit the custom instruction set could have a more concrete result and proof for the practicality and functionality of the instruction set extension.

REFERENCES

[1] Ledin J., “Blockchain and Bitcoin Mining Architectures,” in *Modern Computer Architecture and Organization,* 2nd ed. Birmingham, UK: Packt, 2022, pp. 395-419.

[2] Antonopoulos A.M., “The Blockchain,” in *Mastering Bitcoin,* 2nd ed. California, CA, USA: O’Reilly, 2017, pp. 195-211.

[3] Himelstein M. *et al*., “RISV-V Blockchain SIG Meeting 2022-Jan-24” in *RISV-V Blockchain SIG Biweekly Meeting*, Jan. 24, 2022. [Online]. Available: https://github.com/riscv-admin/blockchain/blob/main/MeetingMinutes/2022/RISC-V%20Blockchain%20SIG%20meeting%202022%20Jan24.pdf

[4] J. Jacobsen. “Let’s Make RISC-V Connected Systems Synonymous with Security” RISC-V.org. https://riscv.org/blog/2021/01/lets-make-risc-v-connected-systems-synonymous-with-security/ (Accessed Apr. 16, 2023)

[5] Zeh, A., Glew, A., Spinney, B., Marshall, B., Page, D., Atkins, D., Dockser, K., Saarinen, M.-J.O., Menhorn, N., Deutsch, L.P., et al. “RISC-V Cryptographic Extension Proposals Volume I: Scalar & Entropy Source Instructions Version v1.0.1”, 2022. Available online: https://github.com/riscv/riscv-crypto/releases/tag/v1.0.1-scalar (Accessed Apr. 19, 2023).

[6] Zeh, A., Glew, A., Spinney, B., Marshall, B., Page, D., Atkins, D., Dockser, K., Saarinen, M.J.O., Menhorn, N., Newell, R., et al. “RISC-V Cryptographic Extension Proposals Volume II: Vector Instructions.”, 2023. Available online: https://github.com/riscv/riscv-crypto/releases/tag/v20230407 (Accessed Apr. 19, 2023).

[7] IBM, “Benefits of blockchain.” IBM.com. https://www.ibm.com/topics/benefits-of-blockchain (Accessed Apr. 18, 2023)

[8] Xue, X., Wang, C., Liu, W., Lv, H., Wang, M., and Zeng, X., “A RISC-V processor with area-efficient memristor-based in-memory computing for hash algorithm in blockchain applications,” *Micromachines*, vol. 10, pp. 541, Aug. 2019, doi:10.3390/mi10080541

[9] Nisancı, G., Flikkema, P.G., and Yalçın, T., “Symmetric cryptography on RISC-V: Performance evaluation of standardized algorithms,” *Cryptography*, vol. 6, pp. 41, Aug. 2022, doi.org/10.3390/cryptography6030041

[10] Cheng, H., GroBschadl, J., Marshall, B., Page, D., and Pham, T., “RISC-V instruction set externsions for lightweight symmetric cryptography,” *IACR Transactions on Cryptographic Hardware and Embedded Systems,* vol. 2023, no. 1, pp. 193-237, Nov. 2022, doi:10.46586/tches.v2023.i1.193-237

[11] Xin, G., Han, J., Yin, T., Zhou, Y., Yang, J., Cheng, X., and Zeng, X., "VPQC: a domain-specific vector processor for post-quantum cryptography based on RISC-V architecture," *IEEE Transactions on Circuits and Systems-I: Regular Papers,* vol. 67, no. 8, pp. 2672-2684, Aug. 2020, doi:10.1109/TCSI.2020.2983185

[12] Marshall, B., Newell, G.R., Page, D., Saarinen, M.J.O., and Wolf, C., "The design of scalar AES instruction set extensions for RISC-V," *IACR Transactions on Cryptographic Hardware and Embedded Systems,* vol. 2021, no. 1, pp. 109-136, Dec. 2020, doi:10.4586/tches.v2021.i1.109-136

[13] Patterson, D.A., and Hennesy, J.L., "Instructions: language of the computer," in *Computer Organization and Design RISC-V: The hardware software interface, 2*nd ed. Cambridge, MA, USA: MK, 2021, ch. 2, pp. 68-128.

[14] Hu, B., Chen, Y., and Zeng, X., "An Agile Instruction Set Extension Method Based onthe RISC-V Processor," *2021 IEEE 4th International Conference on Electronics Technology*, 2021, doi:10.1109/ICET51757.2021.9450911

[15] Preneel, B., Dobbertin, H., and Bosselaers, A., "The Cryptographic Hash Function RIPEMD-160," in *CryptoBytes 3(2)*, pp. 9-14, 1997.

[16] Harris, S., and Harris, D.M., "Architecture: Machine Language," in *Digital Design and Computer Architecture RISC-V Edition*, Cambridge, MA, USA: MK, 2022, ch. 6, pp. 332-343.

[17] Stanis, F., "008 – djb2 hash" The Art in Code. https://theartincode.stanis.me/008-djb2/ (Accessed Apr. 24, 2024)

[18] RISC-V International, "The RISC-V Instruction Set Manual Volume I: Unprivileged ISA" riscv.org. https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf (Accessed Apr. 25, 2024)

[19] RISC-V International, "RISC-V Opcodes" riscv.org. https://github.com/riscv/riscv-opcodes (Accessed Apr. 25, 2024)

[20] Charvi, B., Chaitra, B., Ankitha, C., "A-Simple-Blockchain-Simulation-Using-C" https://github.com/Chaitra-Bhat383/A-Simple-Blockchain-Simulation-Using-C (Accessed Apr. 25, 2024)

[21] RISC-V Collab, "RISCV-GNU-TOOLCHAIN" https://github.com/riscv-collab/riscv-gnu-toolchain (Accessed Apr. 25, 2024)

[22] Altinay, O., Ors, B., "Instruction Extension of RV32I and GCC Back End for ASCON Lightweight Cryptography Algorithm", 2021.

APPENDIX
Appendix A

RISC-V 32-Bit Architecture

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Here is the meaning of each name of the fields in RISC-V instructions:

- *opcode*: Basic operation of the instruction, and this abbreviation is its traditional name.
- *rd:* The register destination operand. It gets the result of the operation.
- *funct3:* An additional opcode field.
- *rs1:* The first register source operand.
- *rs2:* The second register source operand.
- *funct7:* An additional opcode field.

Figure A1-1 RISC-V fields.

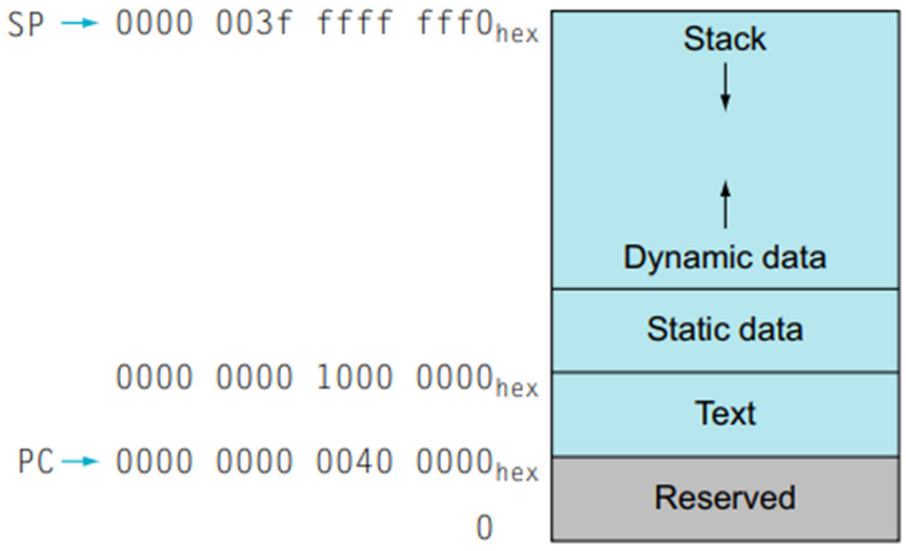


Figure A1-2 RISC-V memory allocation for programme and data.

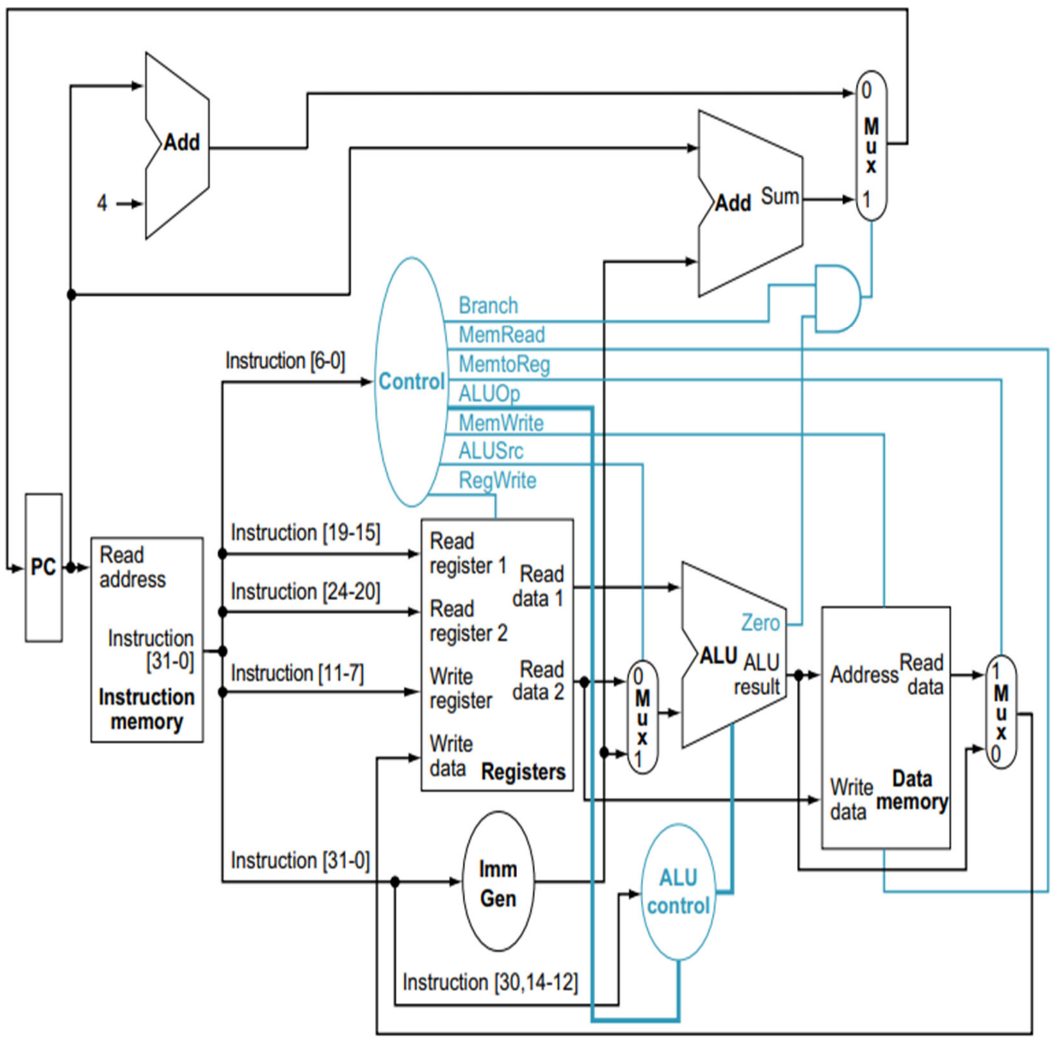


Figure A1-3 RISC-V simple datapath.

RISC-V 32-Bit Instruction Set Base and Extensions

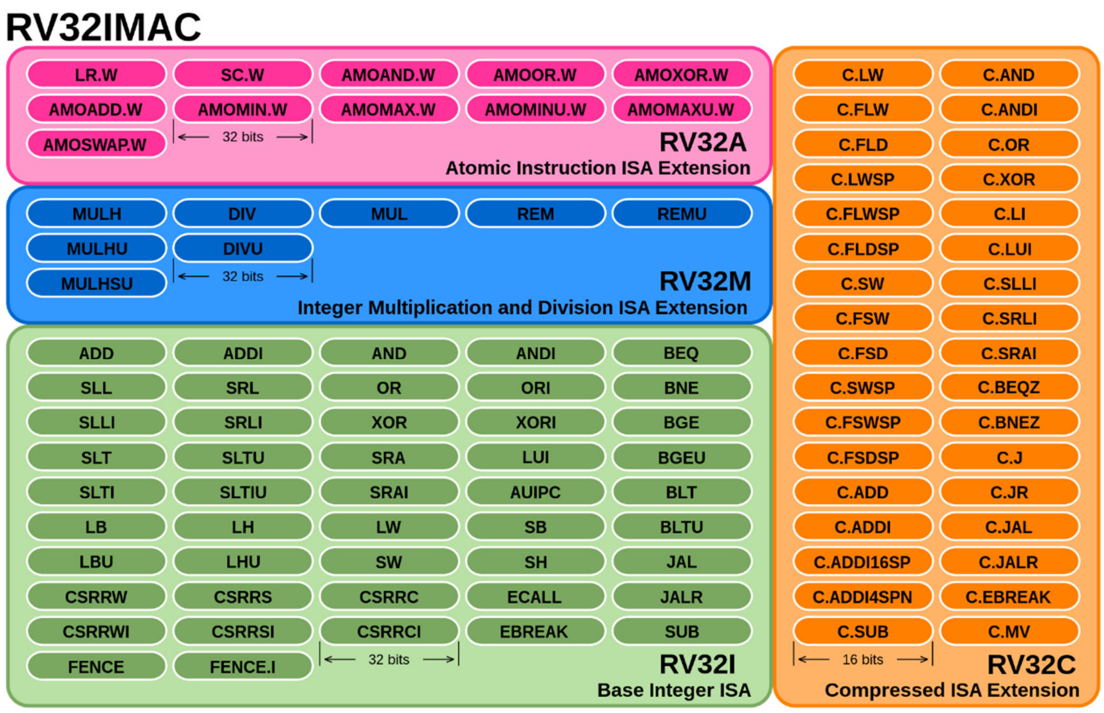
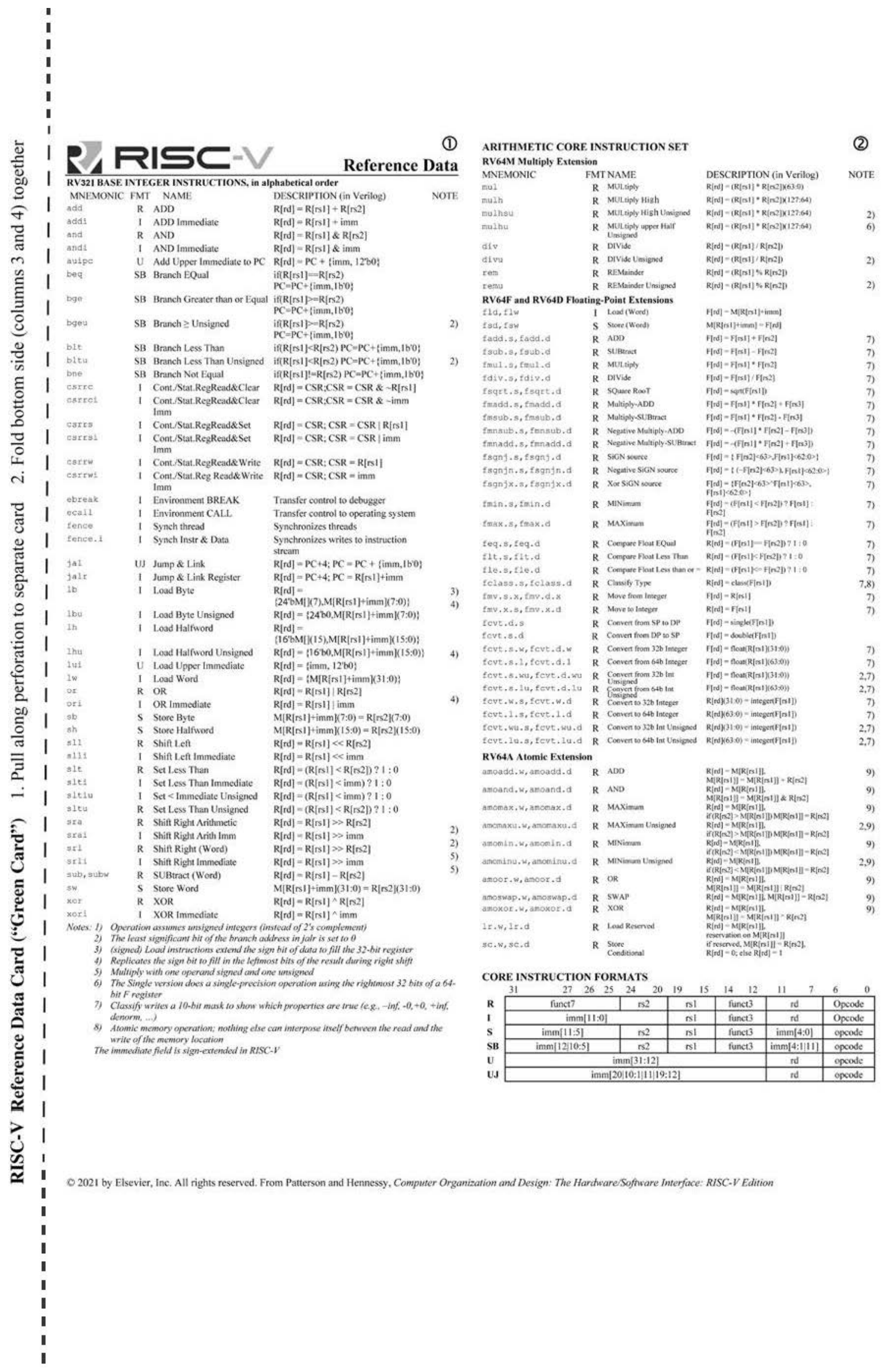


Figure A2-1 The modular instruction set of the RV32IMAC variant. This is a 32-bit CPU with the Base Integer ISA (RV32I) and the ISA extensions for Integer Multiplication and Division (RV32M), Atomic Instructions (RV32A), and Compressed Instructions (RV32C)

RISC-V Architecture Green Card



RISC-V Reference Data Card ("Green Card") 1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together

RISC-V Reference Data ①

RV32I BASE INTEGER INSTRUCTIONS, in alphabetical order

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
add	R	ADD	R[rd] = R[rs1] + R[rs2]	
addi	I	ADD Immediate	R[rd] = R[rs1] + imm	
and	R	AND	R[rd] = R[rs1] & R[rs2]	
andi	I	AND Immediate	R[rd] = R[rs1] & imm	
auipc	U	Add Upper Immediate to PC	R[rd] = PC + {imm, 12'b0}	
beq	SB	Branch EQual	if(R[rs1]==R[rs2) PC=PC+{imm,1b'0}	
bge	SB	Branch Greater than or Equal	if(R[rs1]>=R[rs2) PC=PC+{imm,1b'0}	
bgeu	SB	Branch ≥ Unsigned	if(R[rs1]>=R[rs2) PC=PC+{imm,1b'0}	2)
blt	SB	Branch Less Than	if(R[rs1]<R[rs2) PC=PC+{imm,1b'0}	
bltu	SB	Branch Less Than Unsigned	if(R[rs1]<R[rs2) PC=PC+{imm,1b'0}	2)
bne	SB	Branch Not Equal	if(R[rs1]!=R[rs2) PC=PC+{imm,1b'0}	
csrrc	I	Cont./Stat.RegRead&Clear	R[rd] = CSR;CSR = CSR & ~R[rs1]	
csrrci	I	Cont./Stat.RegRead&Clear Imm	R[rd] = CSR;CSR = CSR & ~imm	
csrrs	I	Cont./Stat.RegRead&Set	R[rd] = CSR; CSR = CSR | R[rs1]	
csrrsi	I	Cont./Stat.RegRead&Set Imm	R[rd] = CSR; CSR = CSR | imm	
csrrw	I	Cont./Stat.RegRead&Write	R[rd] = CSR; CSR = R[rs1]	
csrrwi	I	Cont./Stat.Reg Read&Write Imm	R[rd] = CSR; CSR = imm	
ebreak	I	Environment BREAK	Transfer control to debugger	
ecall	I	Environment CALL	Transfer control to operating system	
fence	I	Synch thread	Synchronizes threads	
fence.i	I	Synch Instr & Data	Synchronizes writes to instruction stream	
jal	UJ	Jump & Link	R[rd] = PC+4; PC = PC + {imm,1b'0}	
jalr	I	Jump & Link Register	R[rd] = PC+4; PC = R[rs1]+imm	
lb	I	Load Byte	R[rd] = {24'bM[](7),M[R[rs1]+imm](7:0)}	3) 4)
lbu	I	Load Byte Unsigned	R[rd] = {24'b0,M[R[rs1]+imm](7:0)}	
lh	I	Load Halfword	R[rd] = {16'bM[](15),M[R[rs1]+imm](15:0)}	
lhu	I	Load Halfword Unsigned	R[rd] = {16'b0,M[R[rs1]+imm](15:0)}	4)
lui	U	Load Upper Immediate	R[rd] = {imm, 12'b0}	
lw	I	Load Word	R[rd] = {M[R[rs1]+imm](31:0)}	
or	R	OR	R[rd] = R[rs1] | R[rs2]	
ori	I	OR Immediate	R[rd] = R[rs1] | imm	4)
sb	S	Store Byte	M[R[rs1]+imm](7:0) = R[rs2](7:0)	
sh	S	Store Halfword	M[R[rs1]+imm](15:0) = R[rs2](15:0)	
sll	R	Shift Left	R[rd] = R[rs1] << R[rs2]	
slli	I	Shift Left Immediate	R[rd] = R[rs1] << imm	
slt	R	Set Less Than	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	
slti	I	Set Less Than Immediate	R[rd] = (R[rs1] < imm) ? 1 : 0	
sltiu	I	Set < Immediate Unsigned	R[rd] = (R[rs1] < imm) ? 1 : 0	
sltu	R	Set Less Than Unsigned	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	
sra	R	Shift Right Arithmetic	R[rd] = R[rs1] >> R[rs2]	
srai	I	Shift Right Arith Imm	R[rd] = R[rs1] >> imm	2)
srl	R	Shift Right (Word)	R[rd] = R[rs1] >> R[rs2]	2)
srli	I	Shift Right Immediate	R[rd] = R[rs1] >> imm	5)
sub,subw	R	SUBtract (Word)	R[rd] = R[rs1] - R[rs2]	5)
sw	S	Store Word	M[R[rs1]+imm](31:0) = R[rs2](31:0)	
xor	R	XOR	R[rd] = R[rs1] ^ R[rs2]	
xori	I	XOR Immediate	R[rd] = R[rs1] ^ imm	

Notes: 1) Operation assumes unsigned integers (instead of 2's complement)
2) The least significant bit of the branch address in jalr is set to 0
3) (signed) Load instructions extend the sign bit of data to fill the 32-bit register
4) Replicates the sign bit to fill in the leftmost bits of the result during right shift
5) Multiply with one operand signed and one unsigned
6) The Single version does a single-precision operation using the rightmost 32 bits of a 64-bit F register
7) Classify writes a 10-bit mask to show which properties are true (e.g., -inf, -0,+0, +inf, denorm, ...)
8) Atomic memory operation; nothing else can interpose itself between the read and the write of the memory location
The immediate field is sign-extended in RISC-V

ARITHMETIC CORE INSTRUCTION SET ②

RV64M Multiply Extension

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
mul	R	MULtiply	R[rd] = (R[rs1] * R[rs2])(63:0)	
mulh	R	MULtiply High	R[rd] = (R[rs1] * R[rs2])(127:64)	
mulhsu	R	MULtiply High Unsigned	R[rd] = (R[rs1] * R[rs2])(127:64)	2)
mulhu	R	MULtiply upper Half Unsigned	R[rd] = (R[rs1] * R[rs2])(127:64)	6)
div	R	DIVide	R[rd] = (R[rs1] / R[rs2])	
divu	R	DIVide Unsigned	R[rd] = (R[rs1] / R[rs2])	2)
rem	R	REMainder	R[rd] = (R[rs1] % R[rs2])	
remu	R	REMainder Unsigned	R[rd] = (R[rs1] % R[rs2])	2)

RV64F and RV64D Floating-Point Extensions

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
fld,flw	I	Load (Word)	F[rd] = M[R[rs1]+imm]	
fsd,fsw	S	Store (Word)	M[R[rs1]+imm] = F[rd]	
fadd.s,fadd.d	R	ADD	F[rd] = F[rs1] + F[rs2]	7)
fsub.s,fsub.d	R	SUBtract	F[rd] = F[rs1] - F[rs2]	7)
fmul.s,fmul.d	R	MULtiply	F[rd] = F[rs1] * F[rs2]	7)
fdiv.s,fdiv.d	R	DIVide	F[rd] = F[rs1] / F[rs2]	7)
fsqrt.s,fsqrt.d	R	SQuare RooT	F[rd] = sqrt(F[rs1])	7)
fmadd.s,fmadd.d	R	Multiply-ADD	F[rd] = F[rs1] * F[rs2] + F[rs3]	7)
fmsub.s,fmsub.d	R	Multiply-SUBtract	F[rd] = F[rs1] * F[rs2] - F[rs3]	7)
fmnsub.s,fmnsub.d	R	Negative Multiply-ADD	F[rd] = -(F[rs1] * F[rs2] - F[rs3])	7)
fmnadd.s,fmnadd.d	R	Negative Multiply-SUBtract	F[rd] = -(F[rs1] * F[rs2] + F[rs3])	7)
fsgnj.s,fsgnj.d	R	SiGN source	F[rd] = { F[rs2]<63>,F[rs1]<62:0>}	7)
fsgnjn.s,fsgnjn.d	R	Negative SiGN source	F[rd] = { (~F[rs2]<63>), F[rs1]<62:0>}	7)
fsgnjx.s,fsgnjx.d	R	Xor SiGN source	F[rd] = {F[rs2]<63>^F[rs1]<63>, F[rs1]<62:0>}	7)
fmin.s,fmin.d	R	MINimum	F[rd] = (F[rs1] < F[rs2]) ? F[rs1] : F[rs2]	7)
fmax.s,fmax.d	R	MAXimum	F[rd] = (F[rs1] > F[rs2]) ? F[rs1] : F[rs2]	7)
feq.s,feq.d	R	Compare Float EQual	R[rd] = (F[rs1]== F[rs2]) ? 1 : 0	7)
flt.s,flt.d	R	Compare Float Less Than	R[rd] = (F[rs1]< F[rs2]) ? 1 : 0	7)
fle.s,fle.d	R	Compare Float Less than or =	R[rd] = (F[rs1]<= F[rs2]) ? 1 : 0	7)
fclass.s,fclass.d	R	Classify Type	R[rd] = class(F[rs1])	7,8)
fmv.s.x,fmv.d.x	R	Move from Integer	F[rd] = R[rs1]	7)
fmv.x.s,fmv.x.d	R	Move to Integer	R[rd] = F[rs1]	7)
fcvt.d.s	R	Convert from SP to DP	F[rd] = single(F[rs1])	
fcvt.s.d	R	Convert from DP to SP	F[rd] = double(F[rs1])	
fcvt.s.w,fcvt.d.w	R	Convert from 32b Integer	F[rd] = float(R[rs1](31:0))	7)
fcvt.s.l,fcvt.d.l	R	Convert from 64b Integer	F[rd] = float(R[rs1](63:0))	7)
fcvt.s.wu,fcvt.d.wu	R	Convert from 32b Int Unsigned	F[rd] = float(R[rs1](31:0))	2,7)
fcvt.s.lu,fcvt.d.lu	R	Convert from 64b Int Unsigned	F[rd] = float(R[rs1](63:0))	2,7)
fcvt.w.s,fcvt.w.d	R	Convert to 32b Integer	R[rd](31:0) = integer(F[rs1])	7)
fcvt.l.s,fcvt.l.d	R	Convert to 64b Integer	R[rd](63:0) = integer(F[rs1])	7)
fcvt.wu.s,fcvt.wu.d	R	Convert to 32b Int Unsigned	R[rd](31:0) = integer(F[rs1])	2,7)
fcvt.lu.s,fcvt.lu.d	R	Convert to 64b Int Unsigned	R[rd](63:0) = integer(F[rs1])	2,7)

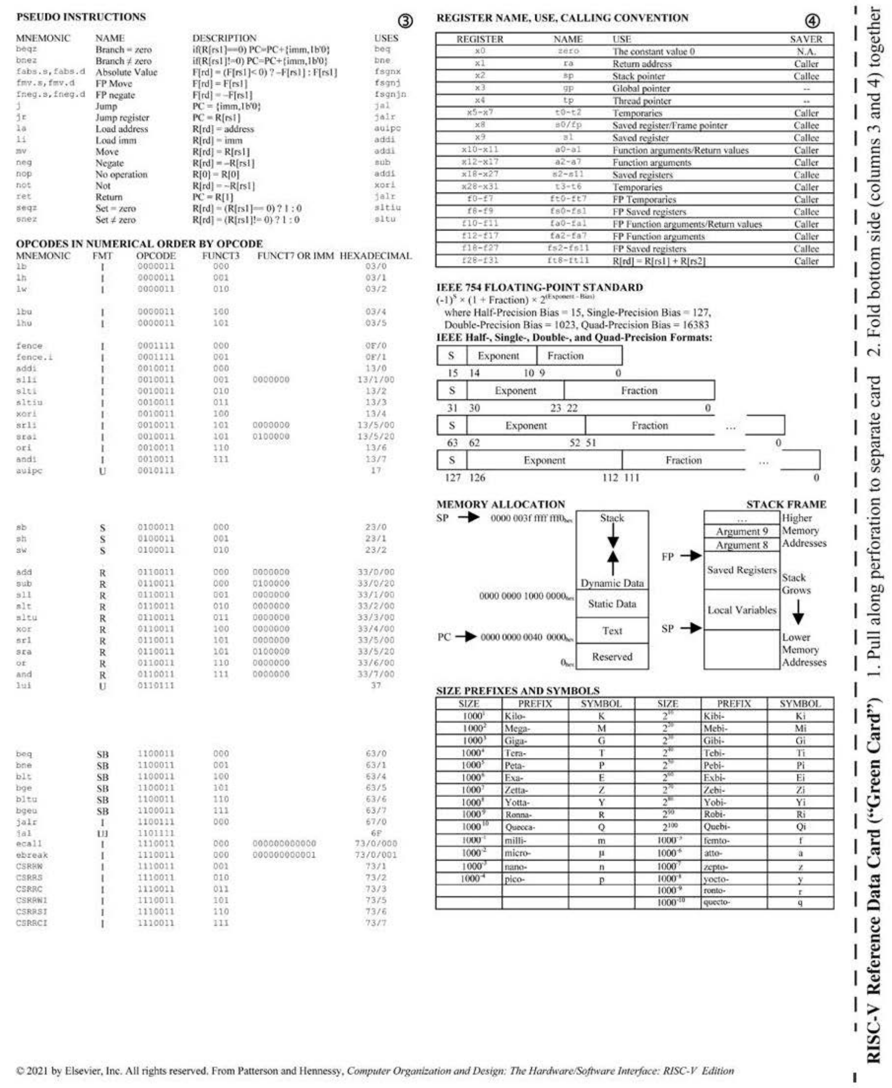
RV64A Atomic Extension

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
amoadd.w,amoadd.d	R	ADD	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] + R[rs2]	9)
amoand.w,amoand.d	R	AND	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] & R[rs2]	9)
amomax.w,amomax.d	R	MAXimum	R[rd] = M[R[rs1]], if (R[rs2] > M[R[rs1]]) M[R[rs1]] = R[rs2]	9)
amomaxu.w,amomaxu.d	R	MAXimum Unsigned	R[rd] = M[R[rs1]], if (R[rs2] > M[R[rs1]]) M[R[rs1]] = R[rs2]	2,9)
amomin.w,amomin.d	R	MINimum	R[rd] = M[R[rs1]], if (R[rs2] < M[R[rs1]]) M[R[rs1]] = R[rs2]	9)
amominu.w,amominu.d	R	MINimum Unsigned	R[rd] = M[R[rs1]], if (R[rs2] < M[R[rs1]]) M[R[rs1]] = R[rs2]	2,9)
amoor.w,amoor.d	R	OR	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] | R[rs2]	9)
amoswap.w,amoswap.d	R	SWAP	R[rd] = M[R[rs1]], M[R[rs1]] = R[rs2]	9)
amoxor.w,amoxor.d	R	XOR	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] ^ R[rs2]	9)
lr.w,lr.d	R	Load Reserved	R[rd] = M[R[rs1]], reservation on M[R[rs1]]	
sc.w,sc.d	R	Store Conditional	if reserved, M[R[rs1]] = R[rs2], R[rd] = 0; else R[rd] = 1	

CORE INSTRUCTION FORMATS

	31 27 26 25	24 20	19 15	14 12	11 7	6 0
R	funct7	rs2	rs1	funct3	rd	Opcode
I	imm[11:0]		rs1	funct3	rd	Opcode
S	imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]				rd	opcode
UJ	imm[20|10:1|11|19:12]				rd	opcode

Figure A3-1 RISC-V Architecture Green Card (1)



PSEUDO INSTRUCTIONS ③

MNEMONIC	NAME	DESCRIPTION	USES
beqz	Branch = zero	if(R[rs1]==0) PC=PC+{imm,1b'0}	beq
bnez	Branch ≠ zero	if(R[rs1]!=0) PC=PC+{imm,1b'0}	bne
fabs.s,fabs.d	Absolute Value	F[rd] = (F[rs1]< 0) ? –F[rs1] : F[rs1]	fsgnx
fmv.s,fmv.d	FP Move	F[rd] = F[rs1]	fsgnj
fneg.s,fneg.d	FP negate	F[rd] = –F[rs1]	fsgnjn
j	Jump	PC = {imm,1b'0}	jal
jr	Jump register	PC = R[rs1]	jalr
la	Load address	R[rd] = address	auipc
li	Load imm	R[rd] = imm	addi
mv	Move	R[rd] = R[rs1]	addi
neg	Negate	R[rd] = –R[rs1]	sub
nop	No operation	R[0] = R[0]	addi
not	Not	R[rd] = ~R[rs1]	xori
ret	Return	PC = R[1]	jalr
seqz	Set = zero	R[rd] = (R[rs1]== 0) ? 1 : 0	sltiu
snez	Set ≠ zero	R[rd] = (R[rs1]!= 0) ? 1 : 0	sltu

OPCODES IN NUMERICAL ORDER BY OPCODE

MNEMONIC	FMT	OPCODE	FUNCT3	FUNCT7 OR IMM	HEXADECIMAL
lb	I	0000011	000		03/0
lh	I	0000011	001		03/1
lw	I	0000011	010		03/2
lbu	I	0000011	100		03/4
lhu	I	0000011	101		03/5
fence	I	0001111	000		0F/0
fence.i	I	0001111	001		0F/1
addi	I	0010011	000		13/0
slli	I	0010011	001	0000000	13/1/00
slti	I	0010011	010		13/2
sltiu	I	0010011	011		13/3
xori	I	0010011	100		13/4
srli	I	0010011	101	0000000	13/5/00
srai	I	0010011	101	0100000	13/5/20
ori	I	0010011	110		13/6
andi	I	0010011	111		13/7
auipc	U	0010111			17
sb	S	0100011	000		23/0
sh	S	0100011	001		23/1
sw	S	0100011	010		23/2
add	R	0110011	000	0000000	33/0/00
sub	R	0110011	000	0100000	33/0/20
sll	R	0110011	001	0000000	33/1/00
slt	R	0110011	010	0000000	33/2/00
sltu	R	0110011	011	0000000	33/3/00
xor	R	0110011	100	0000000	33/4/00
srl	R	0110011	101	0000000	33/5/00
sra	R	0110011	101	0100000	33/5/20
or	R	0110011	110	0000000	33/6/00
and	R	0110011	111	0000000	33/7/00
lui	U	0110111			37
beq	SB	1100011	000		63/0
bne	SB	1100011	001		63/1
blt	SB	1100011	100		63/4
bge	SB	1100011	101		63/5
bltu	SB	1100011	110		63/6
bgeu	SB	1100011	111		63/7
jalr	I	1100111	000		67/0
jal	UJ	1101111			6F
ecall	I	1110011	000	000000000000	73/0/000
ebreak	I	1110011	000	000000000001	73/0/001
CSRRW	I	1110011	001		73/1
CSRRS	I	1110011	010		73/2
CSRRC	I	1110011	011		73/3
CSRRWI	I	1110011	101		73/5
CSRRSI	I	1110011	110		73/6
CSRRCI	I	1110011	111		73/7

REGISTER NAME, USE, CALLING CONVENTION ④

REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/Return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller
f0-f7	ft0-ft7	FP Temporaries	Caller
f8-f9	fs0-fs1	FP Saved registers	Callee
f10-f11	fa0-fa1	FP Function arguments/Return values	Caller
f12-f17	fa2-fa7	FP Function arguments	Caller
f18-f27	fs2-fs11	FP Saved registers	Callee
f28-f31	ft8-ft11	R[rd] = R[rs1] + R[rs2]	Caller

IEEE 754 FLOATING-POINT STANDARD

$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent - Bias})}$

where Half-Precision Bias = 15, Single-Precision Bias = 127,
Double-Precision Bias = 1023, Quad-Precision Bias = 16383

IEEE Half-, Single-, Double-, and Quad-Precision Formats:

S	Exponent	Fraction
15	14 ... 10	9 ... 0
31	30 ... 23	22 ... 0
63	62 ... 52	51 ... 0
127	126 ... 112	111 ... 0

MEMORY ALLOCATION

STACK FRAME

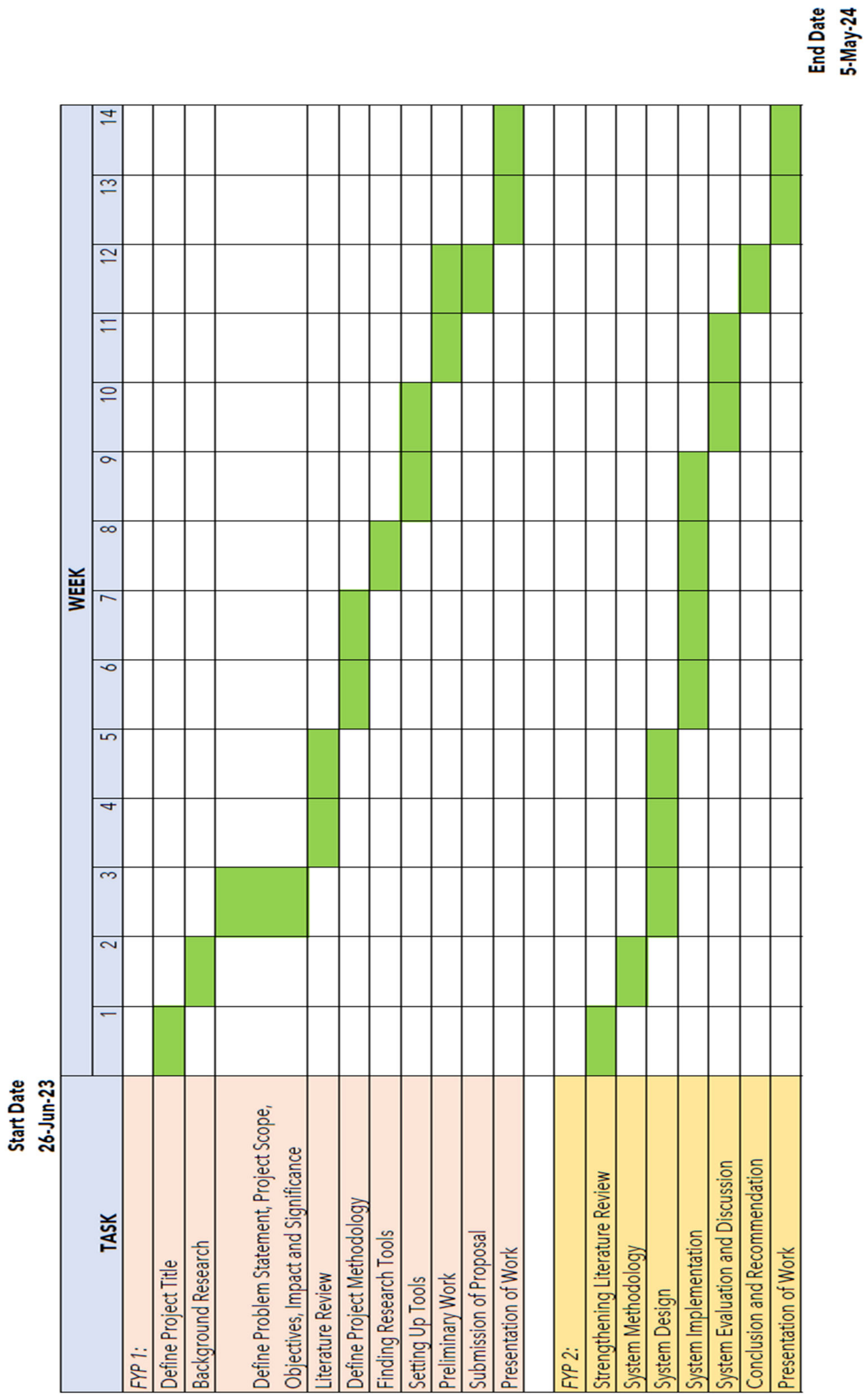
SIZE PREFIXES AND SYMBOLS

SIZE	PREFIX	SYMBOL	SIZE	PREFIX	SYMBOL
1000^1	Kilo-	K	2^{10}	Kibi-	Ki
1000^2	Mega-	M	2^{20}	Mebi-	Mi
1000^3	Giga-	G	2^{30}	Gibi-	Gi
1000^4	Tera-	T	2^{40}	Tebi-	Ti
1000^5	Peta-	P	2^{50}	Pebi-	Pi
1000^6	Exa-	E	2^{60}	Exbi-	Ei
1000^7	Zetta-	Z	2^{70}	Zebi-	Zi
1000^8	Yotta-	Y	2^{80}	Yobi-	Yi
1000^9	Ronna-	R	2^{90}	Robi-	Ri
1000^{10}	Quecca-	Q	2^{100}	Quebi-	Qi
1000^{-1}	milli-	m	1000^{-5}	femto-	f
1000^{-2}	micro-	μ	1000^{-6}	atto-	a
1000^{-3}	nano-	n	1000^{-7}	zepto-	z
1000^{-4}	pico-	p	1000^{-8}	yocto-	y
			1000^{-9}	ronto-	r
			1000^{-10}	quecto-	q

RISC-V Reference Data Card ("Green Card") 1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together

Figure A3-1 RISC-V Architecture Green Card (2)

Appendix B
Project Timeline



Start Date 26-Jun-23

End Date 5-May-24

TASK	WEEK 1	2	3	4	5	6	7	8	9	10	11	12	13	14
FYP 1:														
Define Project Title	🟩													
Background Research		🟩												
Define Problem Statement, Project Scope, Objectives, Impact and Significance			🟩											
Literature Review				🟩	🟩									
Define Project Methodology						🟩	🟩							
Finding Research Tools								🟩						
Setting Up Tools									🟩	🟩				
Preliminary Work											🟩	🟩		
Submission of Proposal												🟩		
Presentation of Work													🟩	🟩
FYP 2:														
Strengthening Literature Review	🟩													
System Methodology		🟩												
System Design			🟩	🟩	🟩									
System Implementation						🟩	🟩	🟩	🟩					
System Evaluation and Discussion										🟩	🟩			
Conclusion and Recommendation												🟩		
Presentation of Work													🟩	🟩

Figure B-1 Project Gantt Chart

Appendix C
Blockchain Application Source Codes

```
#include <ctype.h>
/**
 * Generate a hash key from a string.
 *
 * @param string          The string.
 * @return              A hash key for the string.
 */
int string_hash(void *string)
{
    /* This is the djb2 string hash function */

    int result = 5381;
    unsigned char *p;

    p = (unsigned char *) string;

    while (*p != '\0') {
        result = (result << 5) + result + *p;
        ++p;
    }

    return result;
}
```

Figure C-1 hash.c

```
#include <stdio.h>
#include <stdlib.h>

#include "linkedlist.h"

NODE * reverse(NODE * node) {
   NODE * temp;
   NODE * previous = NULL;
   while (node != NULL) {
      temp = node->next;
      node->next =        previous;
      previous = node;
      node = temp;
   }
   return previous;
}

void init(NODE** head) {
   *head = NULL;
}
```

```
NODE* add(NODE* node, DATA data) {
   NODE* temp = (NODE*) malloc(sizeof (NODE));
   if (temp == NULL) {
      exit(0); // no memory available
   }
   temp->data = data;
   temp->next = node;
   node = temp;
   return node;
}

void print_list(NODE* head) {
   head = reverse(head);
   NODE * temp;
   int indent = 0;
   printf("Print chain\n");
   printf("=========== \n");
   for (temp = head; temp; temp = temp->next, indent = indent+2)
   {
      printf("%*sPrevious hash\t%d\n", indent,"", temp->data.info.previous_block_hash);
      printf("%*sBlock hash\t%d\n", indent,"", temp->data.info.block_hash);
      printf("%*sTransaction\t%s\n", indent,"", temp->data.info.transactions);
      printf("%*s\n", indent, "");
   }

   printf("\r\n");
}

void add_at(NODE* node, DATA data) {
   NODE* temp = (NODE*) malloc(sizeof (NODE));
   if (temp == NULL) {
      exit(EXIT_FAILURE); // no memory available
   }
   temp->data = data;
   temp->next = node->next;
   node->next = temp;
}

void remove_node(NODE* head) {
   NODE* temp = (NODE*) malloc(sizeof (NODE));
   if (temp == NULL) {
      exit(EXIT_FAILURE); // no memory available
   }
   temp = head->next;
   head->next = head->next->next;
   free(temp);
}

NODE *free_list(NODE *head) {
    NODE *tmpPtr = head;
    NODE *followPtr;
    while (tmpPtr != NULL) {
        followPtr = tmpPtr;
        tmpPtr = tmpPtr->next;
        free(followPtr);
    }
    return NULL;
}
```

Figure C-2 linkedList.c

```
#include <stdio.h>
#include <search.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "block.h"
#include "linkedlist.h"

#define NVOTES 10

extern hash string_hash(void *string);

typedef enum party_code_t {GOOD_PARTY, MEDIOCRE_PARTY, EVIL_PARTY,
MAX_PARTIES} party_code;
char *party_name[MAX_PARTIES] = {"GOOD PARTY", "MEDIOCRE_PARTY",
"EVIL_PARTY"};

static party_code get_vote()
{
    int r = rand();
    return r%MAX_PARTIES;
}

void main(int argc, char const *argv[])
{
    srand(time(NULL));

    NODE *head;
    DATA genesis_element;
    init(&head);

    // First block is created manually with hash = 0
    transaction genesis_transactions = {party_name[get_vote()]};
    block_t genesis_block = {0, string_hash(genesis_transactions), genesis_transactions};
    genesis_element.info = genesis_block;
    head = add(head, genesis_element);
    // Now, we are going to submmit n random votes
    int i, previous_hash = genesis_element.info.previous_block_hash;
    transaction trans_list = (transaction) malloc(NVOTES * sizeof(char)*10);
    for(i=0;i<NVOTES;i++)
    {
        DATA *el = malloc(sizeof(DATA));
        block_t *b = malloc(sizeof(block_t));

        transaction t = {party_name[get_vote()]};
        strcat(trans_list, t);
        b->previous_block_hash = previous_hash;
        b->block_hash = string_hash(trans_list);
        b->transactions = t;
        el->info = *b;
        previous_hash = b->block_hash;
        head = add(head, *el);

    }

    print_list(head);

    return;
}
```

Figure C-3 blockchain.c

```
#ifndef BLOCK_H
#define BLOCK_H

typedef int hash;
typedef char *transaction;

typedef struct Block_T {
    hash previous_block_hash;
    hash block_hash;
    transaction transactions;
}block_t;

#endif //BLOCK_H
```

Figure C-4 block.h

```
#ifndef LINKEDLIST_H
#define LINKEDLIST_H

#include <stdio.h>
#include <stdlib.h>

#include "block.h"
```

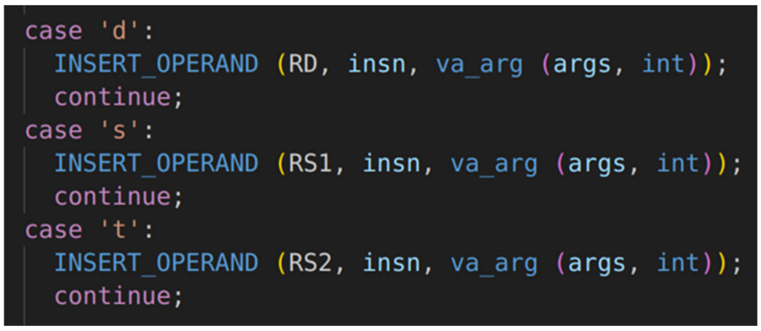
```
typedef struct {
   block_t info;
} DATA;

typedef struct node {
   DATA data;
   struct node* next;
} NODE;

void init(NODE** head);
NODE* add(NODE* node, DATA data);
void add_at(NODE* node, DATA data);
void print_list(NODE* head);
NODE * reverse(NODE * node);
void get_list_transactions(NODE* head, unsigned char *list_transactions);

#endif //LINKEDLIST_H
```

Figure C-5 linkedList.h

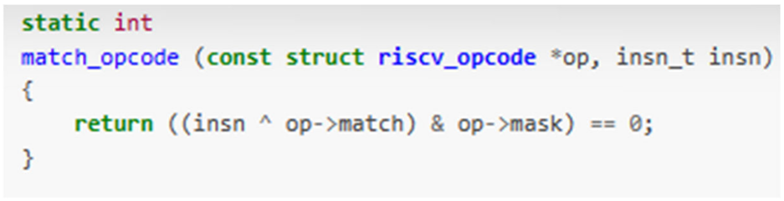
Appendix D
Format for Declaration of Instruction in riscv-opc.c

Table D-1 Format for Declaration of Instruction in *riscv-opc.c*

Parameter	Explanation
name	name of the instruction.
xlen	width of an integer register in bits. [32, 64, 0 (any)]
isa	ISA extension class name.

operands	defined in riscv-gnu-toolchain/binutils/gas/config/tc-riscv.c  `case 'd':`
	`  INSERT_OPERAND (RD, insn, va_arg (args, int));`
	`  continue;`
	`case 's':`
	`  INSERT_OPERAND (RS1, insn, va_arg (args, int));`
	`  continue;`
	`case 't':`
	`  INSERT_OPERAND (RS2, insn, va_arg (args, int));`
	`  continue;`

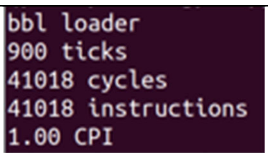
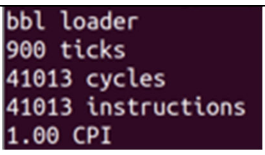
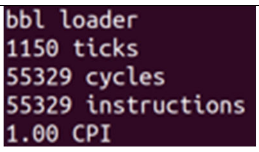
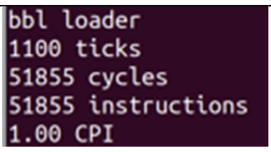
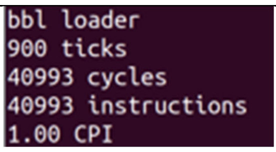
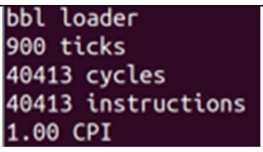
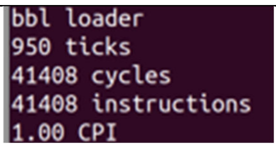
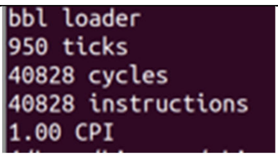
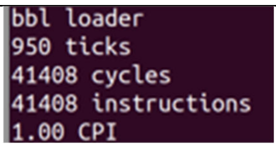
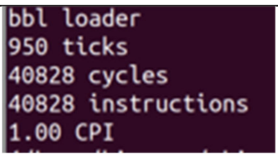
match	the match value.
mask	the mask value.

match_func	pointer to the function recovering funct7, funct3 and opcode fields of the instruction.  `static int`
	`match_opcode (const struct riscv_opcode *op, insn_t insn)`
	`{`
	`    return ((insn ^ op->match) & op->mask) == 0;`
	`}`

pinfo	this field is equal to 0 most of the time except for branch/jump instructions

Appendix E
Simulation Results

Table E-1 Simulation Results

Optimization	Without Custom Extension	With Custom Extension
-Os	 bbl loader 900 ticks 41018 cycles 41018 instructions 1.00 CPI	 bbl loader 900 ticks 41013 cycles 41013 instructions 1.00 CPI
-O0	 bbl loader 1150 ticks 55329 cycles 55329 instructions 1.00 CPI	 bbl loader 1100 ticks 51855 cycles 51855 instructions 1.00 CPI
-O1	 bbl loader 900 ticks 40993 cycles 40993 instructions 1.00 CPI	 bbl loader 900 ticks 40413 cycles 40413 instructions 1.00 CPI
-O2	 bbl loader 950 ticks 41408 cycles 41408 instructions 1.00 CPI	 bbl loader 950 ticks 40828 cycles 40828 instructions 1.00 CPI
-O3	 bbl loader 950 ticks 41408 cycles 41408 instructions 1.00 CPI	 bbl loader 950 ticks 40828 cycles 40828 instructions 1.00 CPI

FINAL YEAR PROJECT WEEKLY REPORT

(Project II)

Trimester, Year: Trimester 3, Year 3	**Study week no.: 2**
Student Name & ID: Cheong Kin Seng (20ACB03898)	
Supervisor: Dr Ooi Joo Onn	
Project Title: RISC-V INSTRUCTION SET EXTENSION ON BLOCKCHAIN APPLICATION	

1. WORK DONE

-

2. WORK TO BE DONE

Study the simulation tool and learn the usage of it.

3. PROBLEMS ENCOUNTERED

Too many bugs in the open-source tools.

4. SELF EVALUATION OF THE PROGRESS

Need to debug the tools on my own or ask the community for solutions

[illegible]

Supervisor's signature

[illegible]

Student's signature

FINAL YEAR PROJECT WEEKLY REPORT

(Project II)

Trimester, Year: Trimester 3, Year 3	**Study week no.: 4**
Student Name & ID: Cheong Kin Seng (20ACB03898)	
Supervisor: Dr Ooi Joo Onn	
Project Title: RISC-V INSTRUCTION SET EXTENSION ON BLOCKCHAIN APPLICATION	

1. WORK DONE Debugging the tools on my own.
2. WORK TO BE DONE Think of ways to carry out the objectives of my project.
3. PROBLEMS ENCOUNTERED Too difficult to figure out a solution as the complexity is too high and limited resources are available for reference.
4. SELF EVALUATION OF THE PROGRESS Think too much and it hindered my project progress.

[illegible]

Supervisor's signature

[illegible]

Student's signature

FINAL YEAR PROJECT WEEKLY REPORT

(Project II)

Trimester, Year: Trimester 3, Year 3	**Study week no.: 6**
Student Name & ID: Cheong Kin Seng (20ACB03898)	
Supervisor: Dr Ooi Joo Onn	
Project Title: RISC-V INSTRUCTION SET EXTENSION ON BLOCKCHAIN APPLICATION	

1. WORK DONE Reducing the scope of the project.
2. WORK TO BE DONE Refine the objectives to fit my abilities.
3. PROBLEMS ENCOUNTERED Project complexity is too hard. Worried to not able to finish it on time.
4. SELF EVALUATION OF THE PROGRESS Too much complaints and whining.

[illegible]
____
Supervisor's signature

[illegible]
____
Student's signature

FINAL YEAR PROJECT WEEKLY REPORT

(Project II)

Trimester, Year: Trimester 3, Year 3	**Study week no.: 8**
Student Name & ID: Cheong Kin Seng (20ACB03898)	
Supervisor: Dr Ooi Joo Onn	
Project Title: RISC-V INSTRUCTION SET EXTENSION ON BLOCKCHAIN APPLICATION	

1. WORK DONE
Refining the project objectives.

2. WORK TO BE DONE
Start working on writing parts of the reports.

3. PROBLEMS ENCOUNTERED
Stuck on few chapters as there was no clue how to continue writing them.

4. SELF EVALUATION OF THE PROGRESS
Need to calm down and think of a solution to progress further

[illegible]

Supervisor's signature

[illegible]

Student's signature

FINAL YEAR PROJECT WEEKLY REPORT

(Project II)

Trimester, Year: Trimester 3, Year 3	**Study week no.: 10**
Student Name & ID: Cheong Kin Seng (20ACB03898)	
Supervisor: Dr Ooi Joo Onn	
Project Title: RISC-V INSTRUCTION SET EXTENSION ON BLOCKCHAIN APPLICATION	

1. WORK DONE
Thought of ways to complete the project

2. WORK TO BE DONE
Finish the simulation, writing the report and complete project.

3. PROBLEMS ENCOUNTERED
Time was too tight to do everything at once.

4. SELF EVALUATION OF THE PROGRESS
Need to manage my time well for the progress.

[illegible]
Supervisor's signature

[illegible]
Student's signature

POSTER

RISC-V INSTRUCTION SET EXTENSION ON BLOCKCHAIN APPLICATION

Project Developer:
Cheong Kin Seng

Project Supervisor:
Dr Ooi Joo Onn

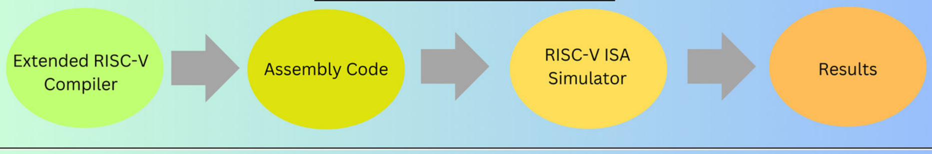
Introduction

DJB2 string hash algorithm is a common cryptographic algorithm used in Blockchain technology. A RISC-V instruction set extension is designed to accelerate the execution of the hash function.

OBJECTIVES

- To customize a RISC-V instruction set extension
- To enhance program execution time
- To reduce assembly code size

METHODOLOGY



SIMULATION RESULTS

O Flags	Base (without extension)		With custom extension		Reduction in instructions (%)	Reduction in time (%)
	# Instructions	Execution Time (ms)	# Instructions	Execution Time (ms)		
-Os	41018	45.58	41013	45.57	0	0
-O0	55329	48.11	51855	47.14	-6.28	-2.02
-O1	40993	45.55	40413	44.90	-1.41	-1.43
-O2	41408	43.59	40828	42.98	-1.40	-1.40

Lesser Memory, Faster Execution

CONCLUSION

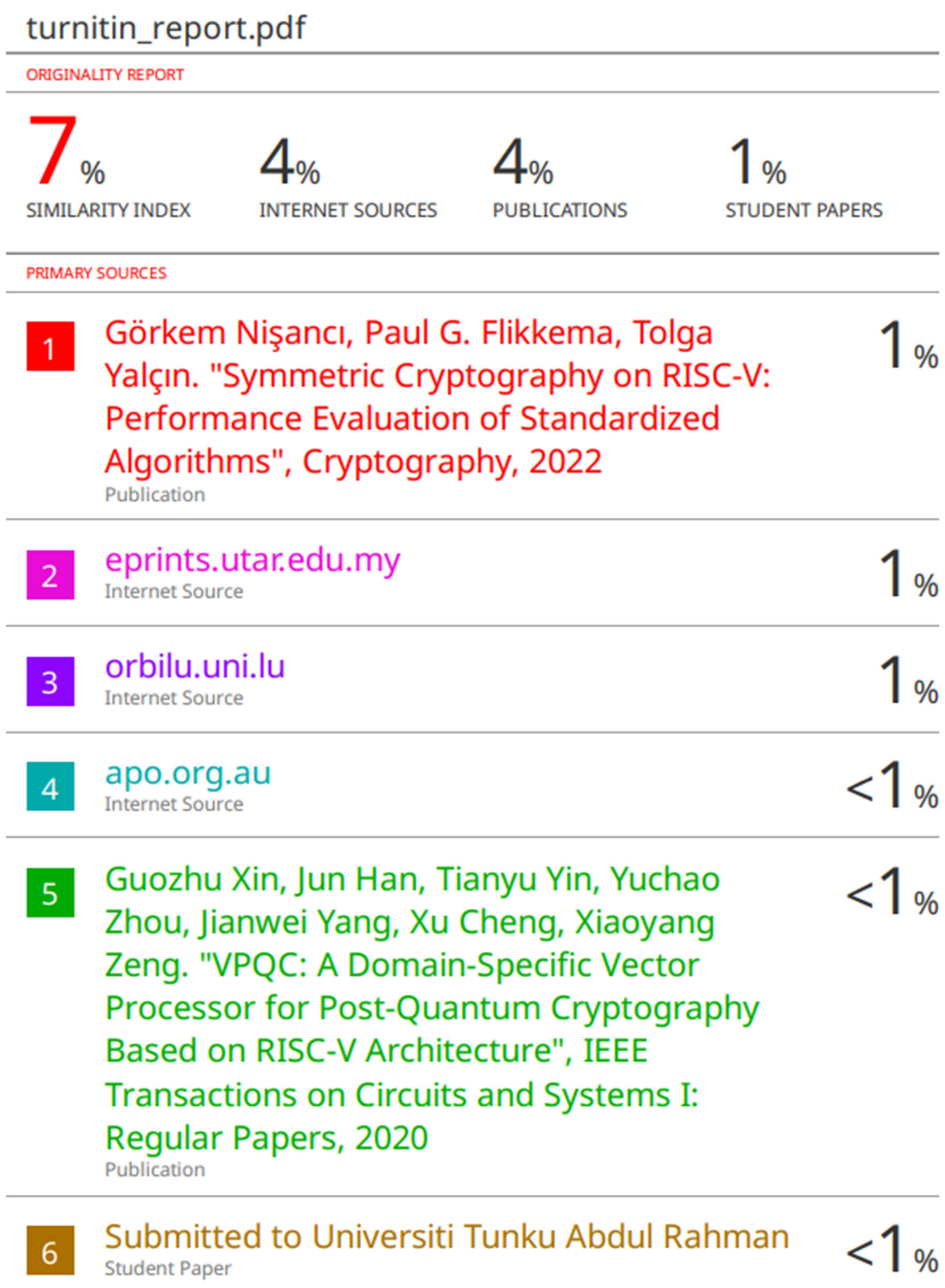
- Programs are able to successfully compile with the extension.
- Reduced number of instructions.
- Reduced program execution time.

UTAR
UNIVERSITI TUNKU ABDUL RAHMAN

Faculty of Information and Communication Technology

BACHELOR OF INFORMATION TECHNOLOGY (HONOURS)
COMPUTER ENGINEERING

PLAGIARISM CHECK RESULT



turnitin_report.pdf

ORIGINALITY REPORT

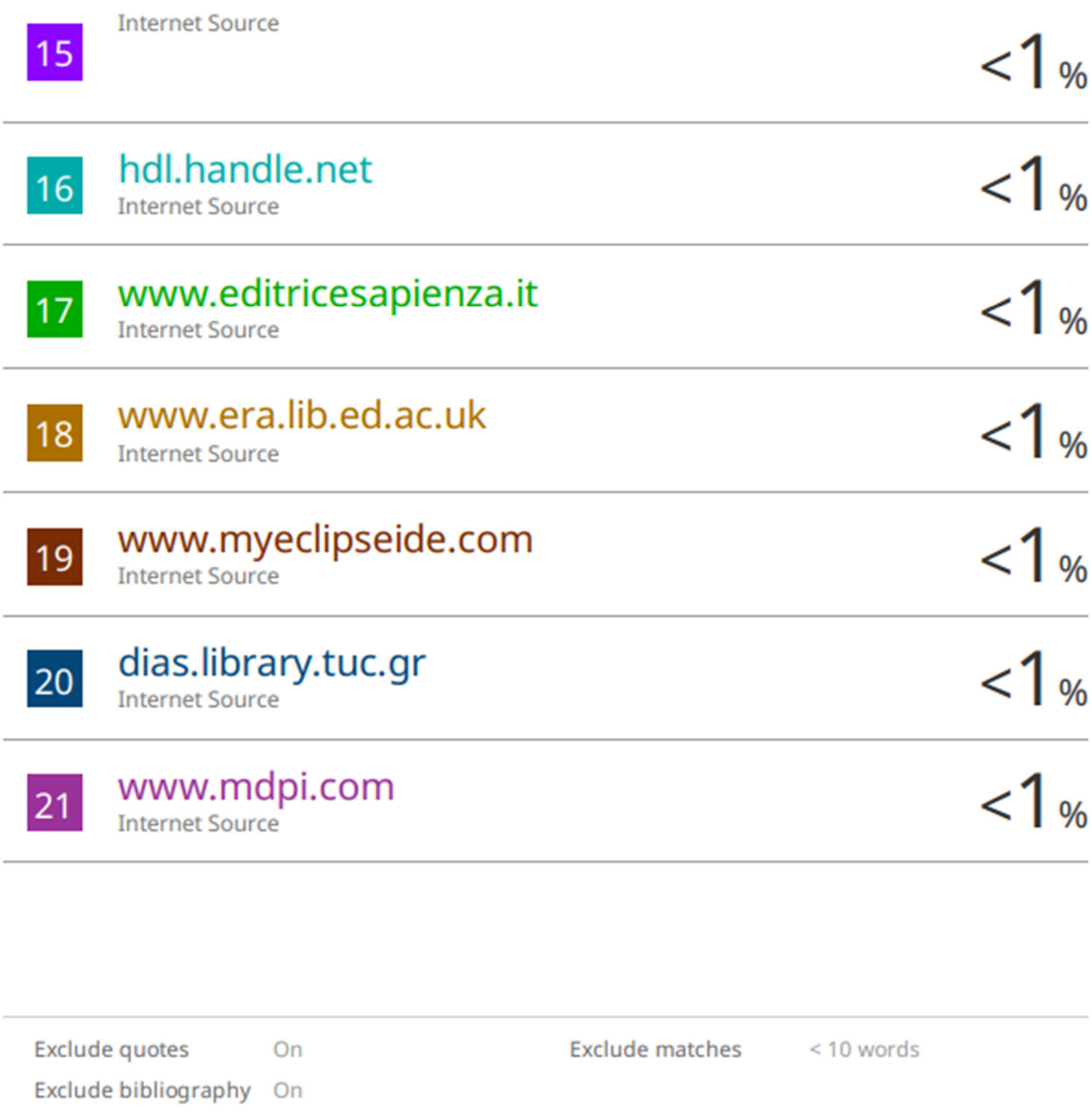
7%	4%	4%	1%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Görkem Nişancı, Paul G. Flikkema, Tolga Yalçın. "Symmetric Cryptography on RISC-V: Performance Evaluation of Standardized Algorithms", Cryptography, 2022 Publication	1%
2	eprints.utar.edu.my Internet Source	1%
3	orbilu.uni.lu Internet Source	1%
4	apo.org.au Internet Source	<1%
5	Guozhu Xin, Jun Han, Tianyu Yin, Yuchao Zhou, Jianwei Yang, Xu Cheng, Xiaoyang Zeng. "VPQC: A Domain-Specific Vector Processor for Post-Quantum Cryptography Based on RISC-V Architecture", IEEE Transactions on Circuits and Systems I: Regular Papers, 2020 Publication	<1%
6	Submitted to Universiti Tunku Abdul Rahman Student Paper	<1%

7	Marc L. Corliss, E. Christopher Lewis, Amir Roth. "The implementation and evaluation of dynamic code decompression using DISE", ACM Transactions on Embedded Computing Systems, 2005 Publication	<1 %
8	Ozlem Altinay, Berna Ors. "Instruction Extension of RV32I and GCC Back End for Ascon Lightweight Cryptography Algorithm", 2021 IEEE International Conference on Omni-Layer Intelligent Systems (COINS), 2021 Publication	<1 %
9	Submitted to Trinity College Dublin Student Paper	<1 %
10	img.electronicdesign.com Internet Source	<1 %
11	mdpi-res.com Internet Source	<1 %
12	www2.mdpi.com Internet Source	<1 %
13	"ICT Systems and Sustainability", Springer Science and Business Media LLC, 2023 Publication	<1 %
14	elib.dlr.de Internet Source	<1 %

s-space.snu.ac.kr



15	Internet Source	<1%
16	hdl.handle.net Internet Source	<1%
17	www.editricesapienza.it Internet Source	<1%
18	www.era.lib.ed.ac.uk Internet Source	<1%
19	www.myeclipseide.com Internet Source	<1%
20	dias.library.tuc.gr Internet Source	<1%
21	www.mdpi.com Internet Source	<1%

Exclude quotes	On	Exclude matches	< 10 words
Exclude bibliography	On		

Universiti Tunku Abdul Rahman			
Form Title : Supervisor's Comments on Originality Report Generated by Turnitin for Submission of Final Year Project Report (for Undergraduate Programmes)			
Form Number: FM-IAD-005	Rev No.: 0	Effective Date: 01/10/2013	Page No.: 1of 1

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Full Name(s) of Candidate(s)	Cheong Kin Seng
ID Number(s)	20ACB03898
Programme / Course	Bachelor of Information Technology (Honours) Computer Engineering
Title of Final Year Project	RISC-V INSTRUCTION SET EXTENSION ON BLOCKCHAIN APPLICATION

Similarity	**Supervisor's Comments (Compulsory if parameters of originality exceeds the limits approved by UTAR)**
Overall similarity index: 7 **%** **Similarity by source** Internet Sources: 4 % Publications: 4 % Student Papers: 1 %	ok
Number of individual sources listed of more than 3% similarity: 1	ok

Parameters of originality required and limits approved by UTAR are as Follows:
(i) **Overall similarity index is 20% and below, and**
(ii) **Matching of individual sources listed must be less than 3% each, and**
(iii) **Matching texts in continuous block must not exceed 8 words**
Note: Parameters (i) – (ii) shall exclude quotes, bibliography and text matches which are less than 8 words.

Note Supervisor/Candidate(s) is/are required to provide softcopy of full set of the originality report to Faculty/Institute

Based on the above results, I hereby declare that I am satisfied with the originality of the Final Year Project Report submitted by my student(s) as named above.

[illegible]
Signature of Supervisor

Name: OOI Joo On

Date: 26 April 2024

Signature of Co-Supervisor

Name: ______

Date: ______

UNIVERSITI TUNKU ABDUL RAHMAN

FACULTY OF INFORMATION & COMMUNICATION TECHNOLOGY
(KAMPAR CAMPUS)

CHECKLIST FOR FYP2 THESIS SUBMISSION

Student Id	20ACB03898
Student Name	Cheong Kin Seng
Supervisor Name	Dr. Ooi Joo Onn

TICK (√)	**DOCUMENT ITEMS** Your report must include all the items below. Put a tick on the left column after you have checked your report with respect to the corresponding item.
√	Title Page
√	Signed Report Status Declaration Form
√	Signed FYP Thesis Submission Form
√	Signed form of the Declaration of Originality
√	Acknowledgement
√	Abstract
√	Table of Contents
√	List of Figures (if applicable)
√	List of Tables (if applicable)
√	List of Symbols (if applicable)
√	List of Abbreviations (if applicable)
√	Chapters / Content
√	Bibliography (or References)
√	All references in bibliography are cited in the thesis, especially in the chapter of literature review
√	Appendices (if applicable)
√	Weekly Log
√	Poster
√	Signed Turnitin Report (Plagiarism Check Result - Form Number: FM-IAD-005)
√	I agree 5 marks will be deducted due to incorrect format, declare wrongly the ticked of these items, and/or any dispute happening for these items in this report.

*Include this form (checklist) in the thesis (Bind together as the last page)

I, the author, have checked and confirmed all the items listed in the table are included in my report.

[illegible]
____
(Signature of Student)
Date: 26 April 2024