INVESTIGATION OF NEW ARCHITECTURAL FEATURES TO SUPPORT PERFORMANCE IMPROVEMENT IN EMBEDDED PROCESSORS

A THESIS SUBMITTED TO

THE BOARD OF GRADUATE PROGRAMS
OF

MIDDLE EAST TECHNICAL UNIVERSITY, NORTHERN CYPRUS CAMPUS

BY

AHMAD OTHMAN

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR

THE DEGREE OF MASTER OF SCIENCE

IN ELECTRICAL AND ELECTRONICS ENGINEERING

AUGUST 2022

Approval of the Board of Graduate Programs

Prof. Dr. Cumali Sabah
Chairperson

I certify that this thesis satisfies all the requirements as a thesis for the degree of Master of Science.

Prof. Dr. Murat Fahrioğlu
Program Coordinator

This is to certify that we have read this thesis and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science .

Dr. Gürtaç Yemişcioğlu
Co-supervisor

Prof. Dr. Murat Fahrioğlu
Supervisor

Examining Committee Members

Prof. Dr. Mustafa Uyguroğlu EMU / EEE

Prof. Dr. Murat Fahrioğlu METU NCC / EEE

Dr. Gürtaç Yemişcioğlu METU NCC / EEE

Assoc. Prof. Dr. Enver Ever METU NCC / CNG

Asst. Prof. Dr. Hüseyin Sevay NEU / ISE

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Ahmad, Othman

Signature :

ABSTRACT

INVESTIGATION OF NEW ARCHITECTURAL FEATURES TO SUPPORT PERFORMANCE IMPROVEMENT IN EMBEDDED PROCESSORS

Othman, Ahmad
Master of Science, Electrical and Electronics Engineering Program
Supervisor: Prof. Dr. Murat Fahrioğlu
Co-Supervisor: Dr. Gürtaç Yemişcioğlu

August 2022, 62 pages

Recent advances in process automation, wireless sensor networks, and machine-to-machine (M2M) interfaces have caused embedded systems to be a blooming computing segment, with significant research focus on performance and energy efficiency. The embedded systems market witnessed enormous growth over the past decades and is foreknown to be boosted in the upcoming years. It has become harder to scale CMOS technologies compared to past and get performance and energy benefits through technology and circuits. Therefore, benefits that can be obtained through architectural optimizations are even more important than before. The focus of this thesis is identification and analysis of new architectural features to potentially improve performance in embedded open-source RISC-V integer processor. Energy efficiency is ultimately enhanced as well, when such enhancements are implemented with little additional power dissipation cost. The integer core of an open-source reference (Rocketchip) processor is enhanced through new fused instructions and hardware features. Statistical data collected on the common Dhrystone and Coremark benchmarks leads to the addition of five fused instructions to the RISC-V integer Instruction-Set-Architecture (ISA), and the Rocket Chip processor organization is modified to support the new instructions. Expected performance benefits are quantified as a result of these changes as compared to the reference Rocket Chip RISC-V processor. 4.4% and 6.1% reduction in instruction count is demonstrated on 500,000

iterations of Dhrystone and 5000 iterations of Coremark benchmarks, respectively, by leveraging a particular original set of fused instructions. This results in 3.8% and 5.4% reduction in execution time for Dhrystone and Coremark benchmarks, respectively.

Keywords: Processor performance, Macro-op fusion, Micro-op fusion, Integer benchmarks, Embedded systems

ÖZ

GÖMÜLÜ İŞLEMCİLERDE PERFORMANS İYİLEŞTİRMELERİNİ DESTEKLEMEK ÜZERE YENİ MİMARİ ÖZELLİKLERİN ARAŞTIRILMASI

Othman, Ahmad
Yüksek Lisans, Elektrik ve Elektronik Mühendisliği Programı
Tez Yöneticisi: Prof. Dr. Murat Fahrioğlu
Ortak Tez Yöneticisi: Dr. Gürtaç Yemişçioğlu

Ağustos 2022, 62 sayfa

Proses otomasyonu, kablosuz sensör ağları ve makineden makineye (M2M) arayüzlerdeki son gelişmeler, gömülü sistemlerin, performans ve enerji verimliliğine odaklanan önemli araştırmalarla gelişen bir bilgi işlem segmenti olmasına neden oldu. Gömülü sistemler pazarı, geçtiğimiz on yıllarda muazzam bir büyümeye tanık oldu ve önümüzdeki yıllarda daha da artacağı biliniyor. CMOS teknolojilerini eskiye göre ölçeklendirmek, teknoloji ve devreler aracılığıyla performans ve enerji avantajları elde etmek zorlaştı. Bu nedenle mimari optimizasyonlar ile elde edilebilecek faydalar eskisinden daha da önemlidir. Bu tezin odak noktası, gömülü açık kaynaklı RISC-V tamsayı işlemcisinde performansı potansiyel olarak geliştirmek için yeni mimari özelliklerin tanımlanması ve analizidir. Enerji verimliliği de, bu tür geliştirmeler çok az ek güç kaybı maliyeti ile uygulandığında eninde sonunda artırılır. Açık kaynaklı bir referans (Rocketchip) işlemcisinin tamsayı çekirdeği, yeni birleştirilmiş yönergeler ve donanım özellikleri ile geliştirilmiştir. Ortak Dhrystone ve Coremark kıyaslamalarında toplanan istatistiksel veriler, RISC-V tamsayılı Talimat-Set-Mimarisine (ISA) beş birleştirilmiş talimatın eklenmesine yol açar ve Rocket Chip işlemci organizasyonu, yeni talimatları desteklemek için değiştirilir. Beklenen performans faydaları, referans Rocket Chip RISC-V işlemci ile karşılaştırıldığında bu değişikliklerin bir sonucu olarak ölçülür. Talimat sayısında 4,4% ve 6,1%'lik azalma, belirli bir orijinal kaynaşmış talimat kümesinden yararlanılarak sırasıyla 500.000 Dhrystone yineleme ve

5000 yineleme Coremark kıyaslamada gösterilmiştir. Bu, Dhrystone ve Coremark kıyaslamaları için yürütme süresinde sırasıyla 3,8% ve 5,4% azalma ile sonuçlanır.

Anahtar Kelimeler: İşlemci performansı, Makro-operasyon füzyonu, Mikro-operasyon füzyonu, Tamsayı karşılaştırmaları, Gömülü Yazılım

To the special ones who gave me unconditional love all these years. I am thankful for your presence in my life. All the people in my life who left a blossom in my story and touched my heart, I dedicate this thesis to you.

ACKNOWLEDGMENTS

Nothing can express my gratitude to my two professors, Ali Muhtaroğlu and Gürtaç Yemişcioğlu, who gave countless hours of reading, supporting, reflecting, and above all, patience during the whole process. My deepest appreciation for your guidance and feedback throughout this project and for every insight you offered into this study. And last but not least, I would like to thank my parents, who have taught me that with care and hard work, there is nothing you can't do if you aspire to achieve.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

ABBREVIATIONS

MOP	**M**acro **OP**eration
SIMD	**S**ingle **I**nstruction **M**ultiple **D**ata
ALU	**A**rithmetic **L**ogic **U**nit
RISC	**R**educed **I**nstruction **S**et **C**omputer
CISC	**C**omplex **I**nstruction **S**et **C**omputer
VM	**V**irtual **M**achine
ISA	**I**nstruction **S**et **A**rchitecture
IPC	**I**nstruction **P**er **C**ycle
SoC	**S**ystem **o**n **C**hip
PC	**P**rogram **C**ounter

CHAPTER 1

INTRODUCTION

One of the main trends in computer architecture is the growth of computation speed over the years. Higher performance CPUs typically consume more power. However, a simple uni-processor that is optimized to satisfy the power and performance needed can be more useful in embedded applications. Embedded processors use less energy, smaller in size, and operate at lower frequencies than general-purpose processors. Most of the embedded system applications are integer in nature, which further supports low-energy operation.

An embedded system is built to do specific tasks. The more specific the processor, the more it can be optimized. Computer architects take advantage of this to increase embedded system performance and reliability, and reduce its cost, size, and energy. These include compiler optimization, architecture optimization, and hardware organization.

The main goal of this research is to explore incremental architectural and organizational enhancements through a particular technique called instruction fusion. Identification of effective fused instructions based on a given set of benchmarks associated with embedded system applications is investigated. This is followed by implementation of sample fusion enhancements on open-source RISC-V Rocket Chip processor to quantify performance benefits.

1.1 Motivation and Problem Definition

The Instruction Set Architecture (ISA) is a set of instructions the processor can understand and execute. Research on instruction set architectures shows

that ISAs have an important role in processor overall performance. Computer architecture community has divided into two fronts: While many argue the recent compiler and microarchitecture advances the ISA no longer has significant impact on processor performance, others still believe that the ISA still plays a significant role in processor performance [11].

RocketChip processor uses RISC instructions. The difference between RISC and CISC is that RISC instructions minimize the number of used standard instructions to make the ISA and corresponding hardware implementation as simple as possible. The main idea behind fused instructions is increasing the performance by decreasing the number of instructions executed by the processor. This can be observed through the processor performance equation:

$$\frac{seconds}{program} = \frac{seconds}{cycle} * \frac{cycles}{instruction} * \frac{instructions}{program}$$

Maintaining $\frac{cycles}{instruction}$, or CPI and $\frac{seconds}{cycle}$, or frequency, while reducing $\frac{instructions}{program}$ will boost processor performance. Customizing the RISC-V architecture to support a new set of fused instructions will target reduction in $\frac{instructions}{program}$. The proposed architectural design maintains CPI and frequency by adding a minimum set of logic blocks to the existing processor data-path to avoid changing frequency by taking advantage of the time wasted on existing memory and register file accesses in order to support new fused instructions.

1.1.1 Motivational Example

Let us consider a scenario where there are N instructions in a program and each instruction takes one clock cycle time, T on the average to execute on a given processor. The total processor execution time would then be N x T, which leads to a performance of 1/NT. If the average power dissipation per instruction is P, then the total energy dissipated by the program is NT x P. What if 20% of the instructions can be executed at the same time as other instructions through fusion of tasks due to the addition of few fused instructions and corresponding new hardware blocks? What if the hardware changes could be implemented with

only 5% increase in average power dissipation without affecting the worst case speedpath (i.e. maximum clock frequency)? Based on this conceptual scenario, the new execution time is 0.8 x NT, i.e. the performance improvement is 125%. In addition, the new energy dissipation for this task is 0.8NT x 1.05P = 0.84 x NTP, i.e. energy is reduced by 84%.

1.2 Proposed Methods and Models

One can appreciate the concept of instruction fusion better through an example. Consider the following example that implements a new Fused multiply-add instruction to replace all cases when the original multiply (MUL) instruction is immediately followed by an addition (ADD) performed on the same operand (X1):

```
// X4 = X1*X2+X4
MUL X1,X1,X2
ADD X4,X4,X1
```

The positive impact of the fused multiply-add is proportional to the number of times the original MUL-ADD instruction sequence occurs in the targeted code sets. Another example is the repeat-move instruction in x86, which copies data from one location to another. After identifying commonly executed micro-operations that are possible to fuse, the hardware dynamically fuses these in the decode stage prior to execution. After that, the fused operations are fed to the pipeline and executed as one operation. How can a processor execute two instructions as one? For example, there are many ways to handle fused multiply-add. One can add a new multiply-add logic unit to the processor datapath, add a second ALU to the pipeline stages that will be responsible for the addition after the first ALU completes the multiplication, or enhance the existing ALU to support the fused operation. The way fused instruction is handled in this research is by adding a second ALU to the memory stage, taking advantage of the memory delay to ensure no impact to worst case processor speedpath or maximum clock frequency. Furthermore, functional units are added to the

execution stage carefully to support the rest of fused instructions with minimum or no speedpath impact. A workflow diagram of this research is shown in figure 1.1.

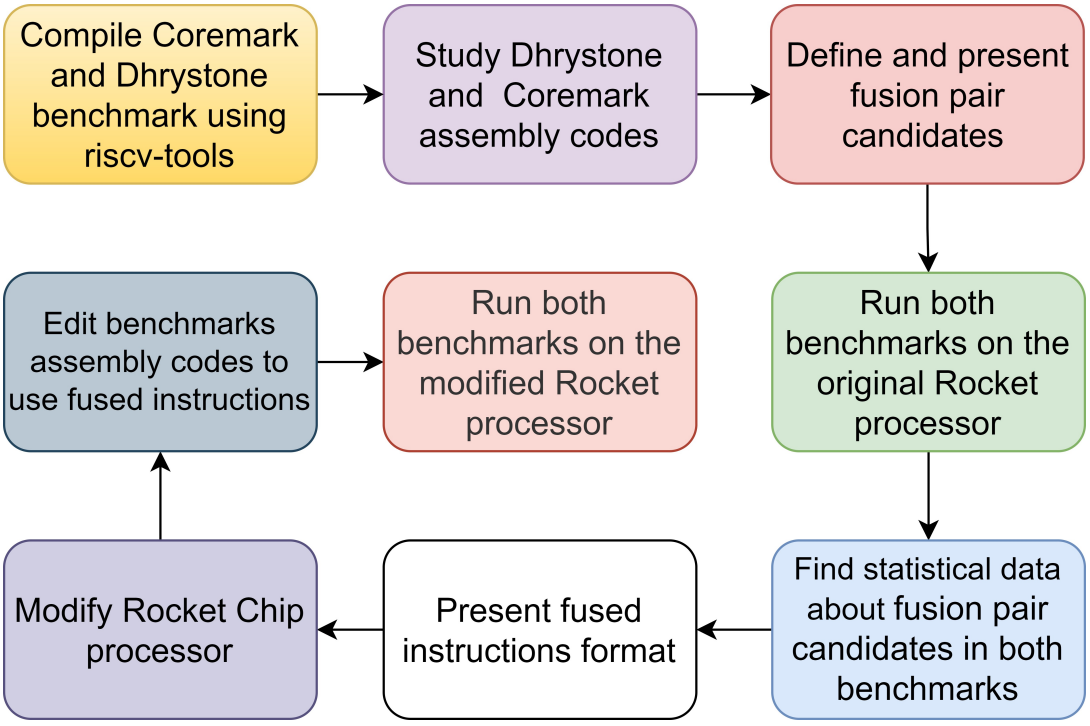

Figure 1.1 Thesis workflow.

1.3 Thesis Contribution

Our contributions are as follows:

- Identify pairs of instructions that are good fusion candidates based on Dhrystone and Coremark benchmarks running on Rocketchip RISC-V embedded processor;
- Propose and implement a 5-stage Rocket modified core design capable of executing five fusion pair candidates.
- Demonstrate performance improvement by running Dhrystone and Coremark benchmarks simulations on the proposed enhanced Rocketchip pro-

cessor model.

1.4 Thesis Outline

Chapter 2 provides background on processor evolution and segmentation to address embedded systems, summarizes relevant benchmarks, and reviews literature on recent RISC-V architecture and fused instruction benefits. Chipyard development environment for building and simulating RISC-V based processor models is also presented in Chapter 2. Analytical data to identify suitable fusing candidates, and resulting RISC-V ISA and Rocketchip hardware enhancements are discussed in Chapter 3. Chapter 4 summarizes results from Dhrystone and Coremark benchmarks simulations. Finally, Chapter 5 provides conclusions and future work.

CHAPTER 2

BACKGROUND AND LITERATURE

An embedded system can be defined as a system that is formed by tightly coupling software and hardware components, built to perform a specific task. It cannot typically be programmed by the end user, and forms a subsystem of a larger system. Embedded systems are everywhere, deeply ingrained into every corner of our life. Embedded systems can be found in education, healthcare, and other industries, forming smart modules in cars, surveillance cameras, planes, mobile phones, washing machines, defence equipment, and many other systems. The central component of any embedded system is an embedded processor providing it with computational processing capability. These embedded processors perform highly efficient limited tasks, with lower power consumption compared to general purpose processors. This chapter provide a snapshot of the most important aspects of the historical changes and current trends in embedded processors, different instruction sets, common embedded benchmarks, and the environment used in this research.

2.1 Processor evolution for embedded systems

Embedded system concept is old. In a sense, embedded systems introduce a framework for general purpose computer concept. Early electronic computing devices, such as Apollo Guidance Computer and Colossus Mark I and II computers, were closer to embedded systems in functionality than general purpose computers.

2.1.1 Early processors

The evolution of processors for embedded systems follows the evolution of microprocessors. After the emergence of microprocessors in the 1970s, embedded processors witnessed rapid developments and changes. In 1971, Intel launched the first microprocessor and named it Intel 4004, which could perform 4-bit arithmetic operations. Meanwhile, Texas Instrument (TI) followed Intel and introduced its first microprocessor in 1974, TMS1000, a 4-bit microprocessor with 1KB ROM and 32-byte RAM. A couple of years after Intel 4004, Intel followed it by Intel 8008 and 8080 microprocessors. Intel later updated the 8080 8-bit processor to Intel 8085 by increasing its instructions, input/output pins, and interrupts. At the same time, Motorola developed MC6800 8-bit microprocessor. These microprocessors were basically used as embedded applications rather than CPUs. Intel 8080 evolved into the famous Intel 8086 and Intel 8087 eventually.

The evolution in microprocessors continued during the 1980s and 1990s, resulting in 16-bit, 32-bit, and 64-bit CPU designs, and even 128-bit width in some specialized CPUs nowadays. Moreover, reduced instruction set computer (RISC) architecture has been introduced [12], providing hardware optimizations within the same resources compared to the multi-chip concept. Many processors adopted this idea, including UC Berkeley RISC I/II processors [13], MIPS R2000, Motorola 68020, Intel 80360DX, and Acorn RISC Machine (ARM). After the pipeline concept was introduced, the microprocessors market witnessed the addition of deeper pipeline stages, higher issue rates, multilevel caches, wider bandwidths, and more on-chip functional units [14]. In the 1990s, the microprocessor market was overtaken by superscalar processors, which increased the previous performance limits [15].

Afterward, in the mid-2000s, Companies like Intel and AMD stopped increasing their processor clock frequency due to power dissipation barriers. Later, the power density wall was overcome by parallel computing techniques by using more than one processor on a single chip [16]. Figure 2.1 shows the trends in the main characteristics of popular general-purpose CPUs over time.

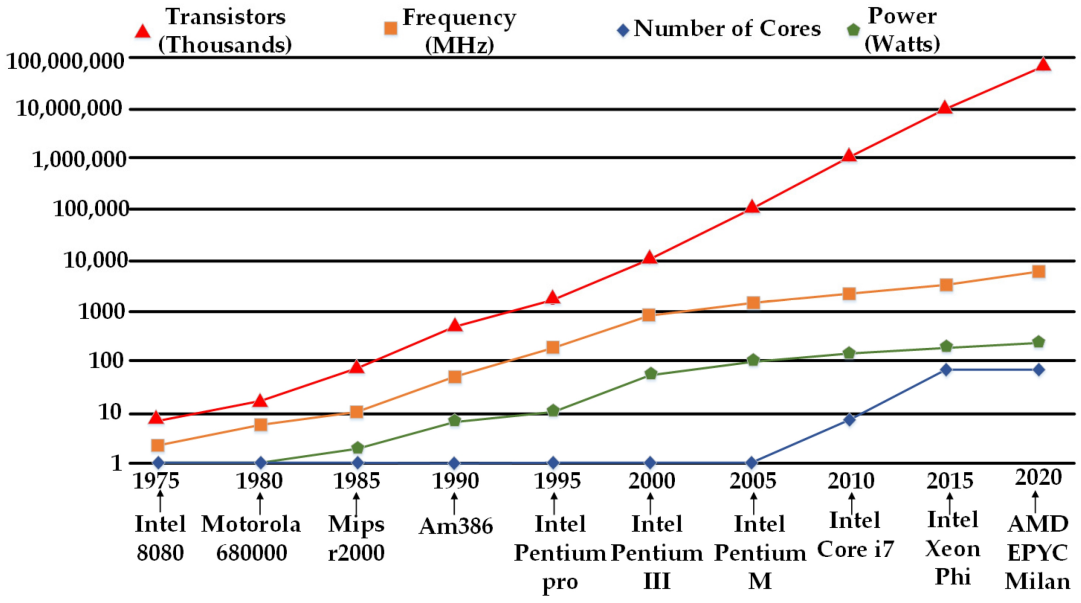

Figure 2.1 Performance evolution of general-purpose microprocessors [3].

2.1.2 Processor segmentation to support embedded systems

As computers became ubiquitous across personal, industrial, and scientific applications, the processor market became segmented to serve each category of users better. 8085 Intel processor released in 1977 was mainly targeted for embedded applications. In the 1980s, other segments, such as mobile, server, and desktop emerged.

2.1.3 Trends in embedded system processors

Lately, embedded processors got involved in almost every aspect of our life. In this section recent emerging trends in embedded processors [17] are reviewed.

ARM (Advanced RISC Machine)

From ARM1 to the latest ARM11 processors core, ARM architecture developed significantly over the years. ARM11 is a 32-bit RISC core with an 8-stages pipeline. ARM switched its architecture from Von Neumann to Harvard after

ARM 7.

- *ARM11*

Maximum frequency 335 MHz, 1.2 MIPS/MHz, and a 16X32 multiplier.

AVR

Embedded AVR core processors designed by Atmel used modified Harvard architecture design. Some of the AVR embedded processors used nowadays are:

- *32-bit AVR*

It has video and audio processing features and includes SIMD and DSP instructions. Its instruction set is similar to RISC cores.

- *XMEGA AVR –ATxmega*

Has internal ADC, 44-64-100 Pin package, and a 16-384 KB Program Memory.

- *Tiny AVR*

Limited peripheral set, 16KB Program Memory, and 6-32 Pin package.

- *Mega AVR-ATmega*

Large memory, ISA is extendable, extensive peripheral set, 28-100 Pin package, and a 512KB Program Memory.

- *Application specific AVR*

512KB Program Memory, 28-100 Pin package, LCD Controller, USB Controller, Advanced PWM, CAN.

PIC

Some of the PIC embedded processors used nowadays are:

- *Baseline core devices (12 bit)*

PIC10 series, as well as by some PIC16 and PIC12 devices. 12-bit wide code memory, 32-byte register file, Two-tiny-level deep call stack, 6-pin to 40-pin packages.

- *dsPIC16 and PIC 24-bit microcontrollers*

Hardware MAC (multiply–accumulate), Barrel shifting, Bit reversal, (16×16)-bit single-cycle multiplication and other DSP operations, Hardware divide assist (19 cycles for 16/32-bit divide), Hardware support for loop indexing, Direct memory access.

- *PIC32 32-bit microcontrollers*

The highest execution speed 80 MIPS Flash, memory: 512 kByte, One instruction/clock cycle execution, First cached processor, Allows execution from RAM, Full Speed Host/Dual Role, OTG USB capabilities, Full JTAG, 2-wire programming and debugging Real-time trace.

2.2 Embedded RISC-V open-source architecture

RISC-V is an open-source instruction set (ISA) developed and designed by UC Berkeley. Berkeley Architecture Group set two goals while developing RISC-V: i. Free, open-source ISA implementation, and ii. suitability for almost any computing device or computing segment.

2.2.1 RISC-V architectural features

RISC-V base ISA is designed to be simple, clean, and suitable for hardware implementation. Figure 2.2 below summarizes the base 32-bit RISC-V instruction format.

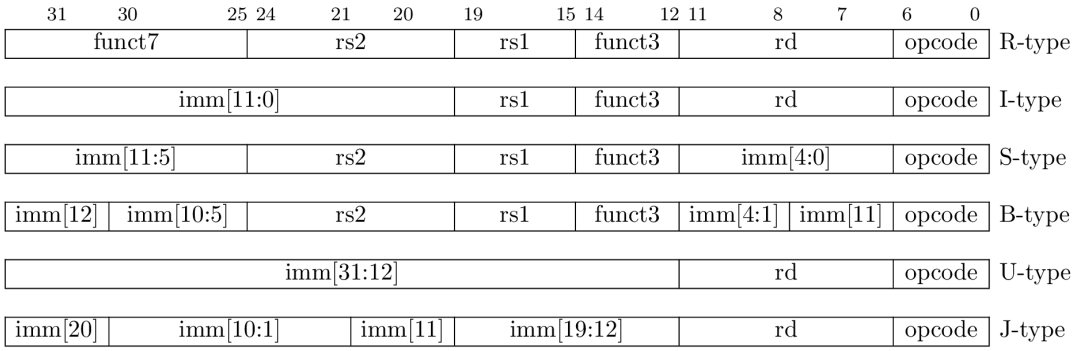

Figure 2.2 Base RISC-V instructions format [4].

RISC-V was built according to RISC design principles and has the following features and characteristics:

- Load-store architecture design;
- Simple, fixed base, stable, and standard extensions feature,
- Simple CPU multiplexers as a result of bit patterns,
- Fast sign extension, due to its designed fixed format,
- It supports a wide range of uses, from microcontrollers to personal computers to supercomputers,
- Flexible design, with the ability to add more encoding bits and predefined extensions for specialized CPUs.

RISC-V architecture is chosen for this research because of many factors. RISC-V is open-source; many academic studies and innovations use this baseline processor architecture. It has a simple ISA without special instructions, which means it is the best choice to study the effect of Instruction fusion. RISC-V simplifies multiplexers in a processor, and the immediate values are placed at a fixed location to speed up sign extension; such architecture makes it easier to modify to execute fused instruction with fewer functional units.

2.2.2 Rocket Chip organization

Rocket Chip is one of the SoC generators that Chipyard environment provides. The generator is written in Chisel language and contains many parts besides the CPU (Rocket core). Figure 2.3 shows an example of Rocket Chip instance [5]. Rocket Chip consists of parameterized Chisel RTL libraries that allow one to generate different SoC variants. The tile consists of rocket core, optional L1 caches(data and instructions), and page table walker. Rocket Chip caches communicate through what is called a tile bus to a simulated DRAM.

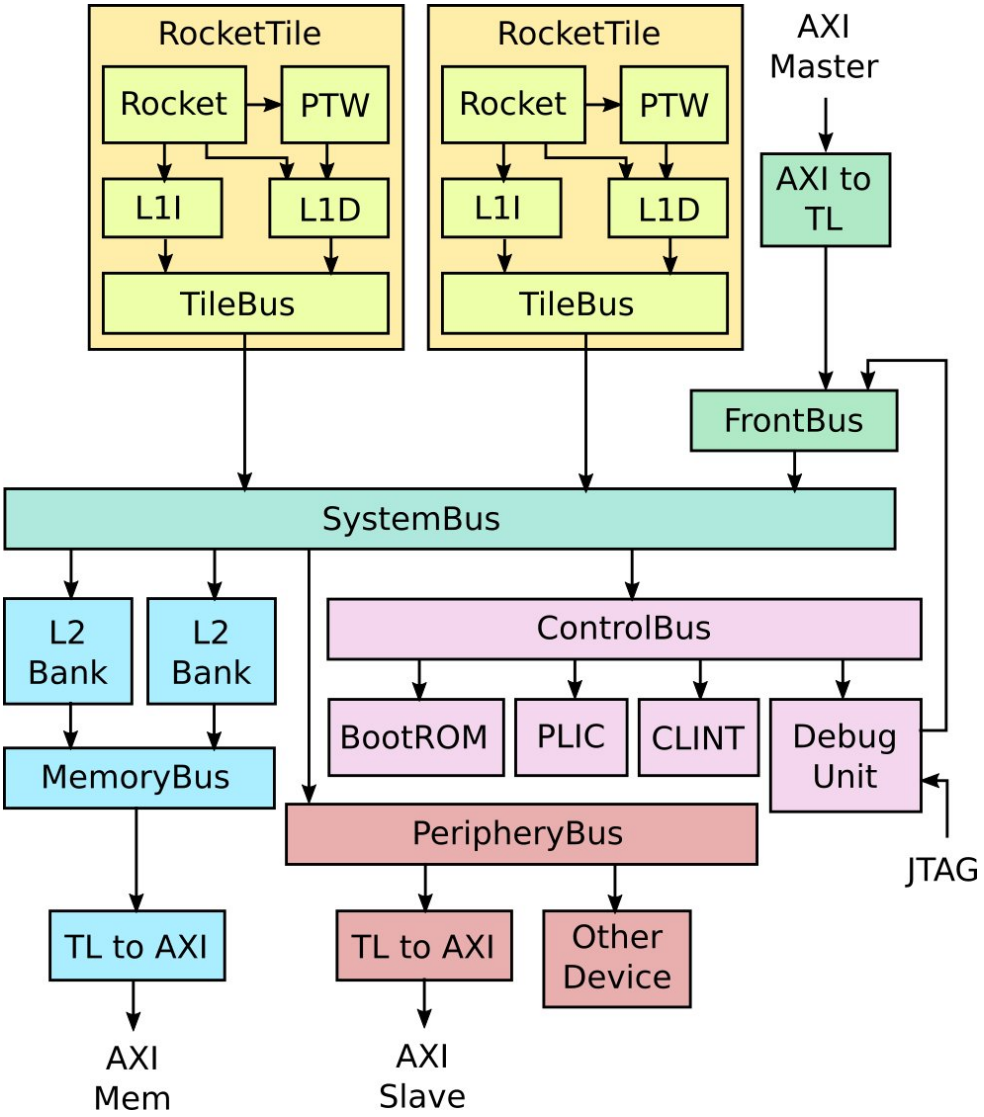

Figure 2.3 Rocket Chip system [5].

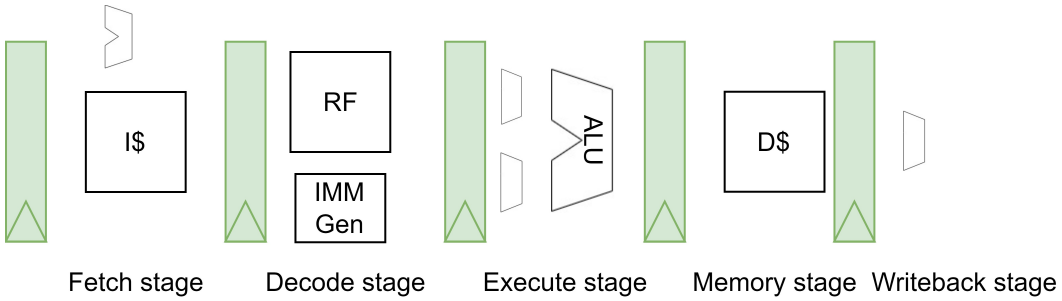

Figure 2.4 In-order Rocket Core pipeline [5].

This research uses Rocket Core as baseline, and does not explore any of the peripherals. Rocket Core is an in-order, scalar 5-stages RISC-V processor developed at University of California, Berkeley(Figure 2.4). Rocket core is also written in Chisel language and can support 32-bit or 64-bit RISC-V ISA with all the optional extensions (I, M, A, F, D, C). A detailed Rocket microarchitecture is shown in Figure 2.5.

There are several reasons why Rocket Chip is chosen for this research. The simplest version of Rocket core serves effectively as an embedded system processor using RISC-V ISA. It is compatible with RISC-V tools, and is written in chisel, a fast C emulator can be generated for debugging and simulating hardware designs. Rocket Core architecture design can be modified to execute fused instructions while minimizing the number of functional units added.

The Chisel C Emulator

Chisel can generate a fast C emulator from a scala source for debugging and simulating hardware designs. C emulator is capable of cycle-accurate simulation of the Rocket core. Using the **verbose** mode in the emulator prints debug information to the console each cycle.

The RISC-V Tools

The RISC-V toolchain is a standard GNU cross compiler used to compile, assemble, and link source files.

Spike

Spike is a software simulator widely used in processor development. Takes care of providing a complete hardware base system to run programs prepared for "bare metal", those programs are executables that run without an operating system, or it can be specified to use a kernel proxy [18] with protocol HTIF/FESVR transmission to be able to run programs that need a kernel to manage.

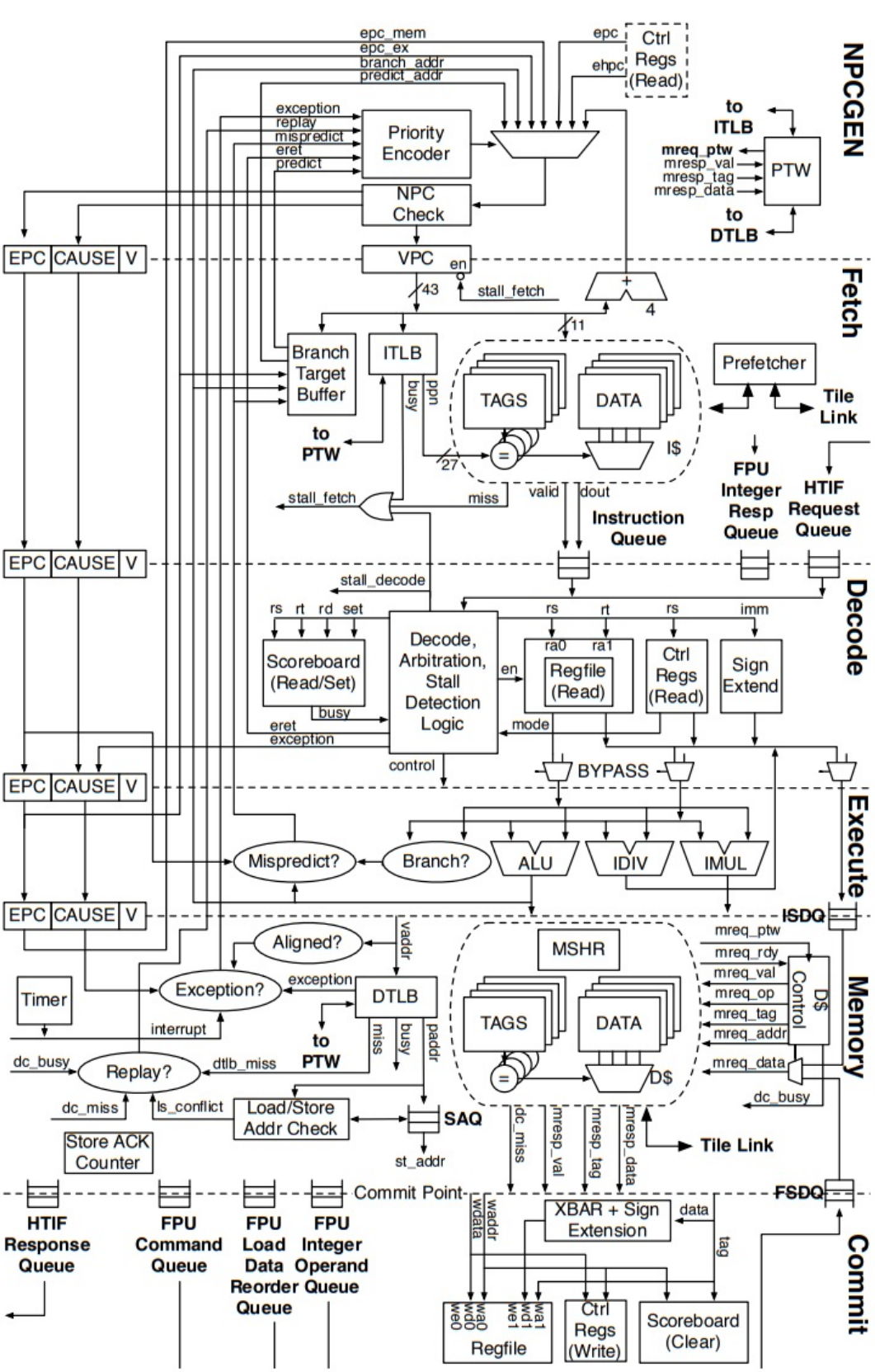

Figure 2.5 Rocket Core microarchitecture [6].

2.3 Instruction Sets

2.3.1 RISC vs. CISC instruction sets

CISC and RISC are the main two types of ISAs used in embedded systems. CISC ISA are more complex and tend to sacrifice the number of cycles per instructions to achieve a lower number of instructions per program. On the other hand, RISC focuses on making the instructions in the ISA as simple as possible in order to achieve a lower clock cycle value. Table2.1 shows the main differences between RISC and CISC characteristics and features.

Table 2.1 RISC vs CISC [1] [2].

RISC	CISC
Simple Instructions	Complex instructions
Have more general purposes registers	Have fewer general purposes registers
Load/Store instructions are independent	Load/Store instructions can be involved with other instructions
Large code size	Small code size
avoid any special instructions	Usually have special instructions
Simple binary encoding of instructions	Complex binary encoding of instructions
Fixed length instructions	Variable length instruction
Fewer addressing modes	More addressing modes
Examples: ARM, SPARC	Examples: IBM 370/168, VAX 11/780

Modern ISAs, including RISC-V, have converged to RISC. On the other hand, the approach has been revisited recently due to the potential benefit of having some instructions in the ISA that have higher complexity than the others in order to achieve better overall performance and energy efficiency. The study of instruction fusion is an example of such diversion from a "blind" RISC type design.

2.3.2 RISC-V instruction set for embedded systems

RISC-V offers an extension that supports embedded systems applications called RV32E Base Integer Instruction Set. RV32E is a reduced version of RV32I. It has 16 general-purpose registers and no counters, unlike RV32I which uses 32 general-purpose registers and mandate counters. RV32E can only extend with M, A, and C user-level standard extensions.

2.3.3 Fused instructions and benefits

Fused instructions or macro-op fusion have been applied with many variants and different microarchitectures on different types of processors, including superscalar processors, scalar processors, and VLIW. These methods vary and may involve specialized functional units to support a few fused instructions, microarchitecture pipeline changes to support instruction dynamic binary translation or designed microarchitecture with macro-op scheduling. Fusing instructions can be done within the pipeline after fetching the instructions using a hardware translator and scheduler or before feeding the instructions to the processors using software to fuse the dependent instructions into fused instructions that are supported by the processor. Most of the studies used the potential of parallelism, e.g., SIMD, to schedule and feed the fused instructions to the proposed microarchitecture processor. Fused instruction concept has only been applied on a small scale. The nature of instructions having strong dependencies is the key that makes fused instruction a strong optimization option. This section explores critical past studies on instruction fusion.

x86 Processor Implementations

The concept of fused instructions is not new and has been applied to many CISC processors before. The AMD k7/k8 micro microarchitecture [19] reduced the dynamic instruction count using internal Macro-Operations in the pipeline front-end. Another example is the Intel Pentium M microarchitecture [20], which fuses ALU operations with memory store/load instructions. An x86 processor MOP microarchitecture implementation is discussed in [7]. In that work, a CISC co-

design that uses fused instructions has been proposed and analyzed to increase performance. Considering the complexity, a fully transparent dynamic translation software was utilized to decompose CISC instructions into one or more RISC-style micro-ops and then fuse the dependent instructions after reordering them. The resulting macro-ops are fetched into the proposed microarchitecture as single instructions. A 3-input ALU and pipeline scheduling logic are used to implement the functionality of the fused instructions.

The research proposed new architectural features, which allowed the processor to translate CISC instructions into micro-ops. Then a two-level decoder was used to feed the control signals into the pipeline. A two-level decoder in Figure 2.6 provides two modes: Fused macro-op mode and x86-mode. x86-mode is activated when the program starts up, allowing the hardware to detect frequent dependent instructions or potential frequent code regions. The instructions are organized and translated using VM software into fused macro-ops and sent to a concealed code cache. Taking advantage of superscalar out-of-order processor, other instructions can be fetched while fused macro-ops are being prepared. When fused instruction is ready, the macro-op mode is turned on, and the first level of the decoder is bypassed. Moreover, while executing fused macro-ops from the cache, the first-level decode logic can be turned off to save energy.

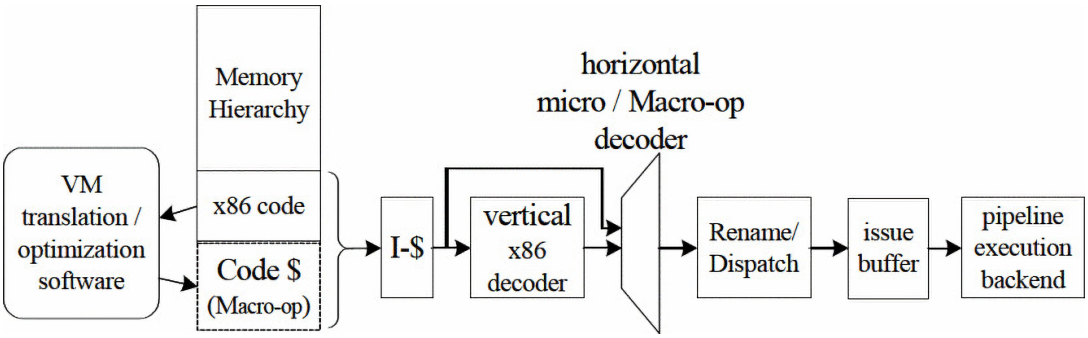

Figure 2.6 Overview of the proposed x86 design [7].

As programs run on this processor, the co-designed VM software will repeatedly switch between the two modes.

Fused macro-op instruction set is shown in Figure 2.7. This fusable ISA consists of RISC-like micro-ops with 16-bit and 32-bit formats. Using a format for fused instructions is not essential, but it provides more flexibility to the translator. As shown in the figure, the 32-bit format uses three register operands which utilize the three inputs of the added 3-input ALU, In comparison, the 16-bit format uses two operands where one of the operands is both a source and the destination.

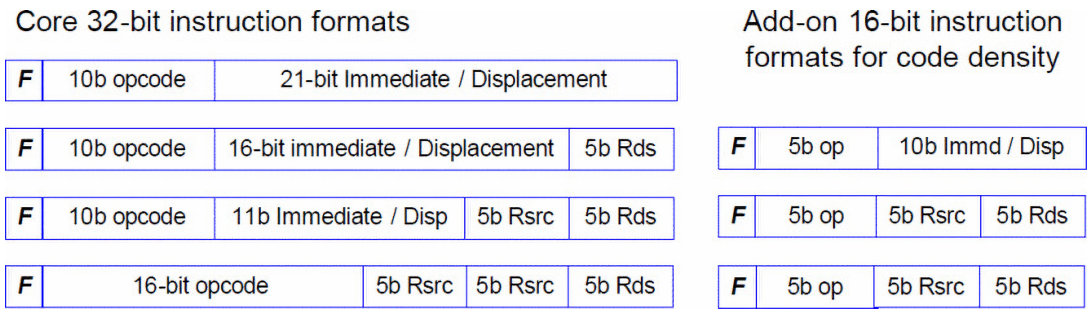

Figure 2.7 Formats for fusable micro-ops [7].

The co-designed microarchitecture and the conventional x86 processor have the same pipeline stages (Figure 2.8). The only difference here is that using the added two-way decoder and the 3-input ALU the processor can process fused macro-ops instructions in coarse-grained parallelism through the superscalar processor pipeline.

Since there are two modes of operation because of using the two-level decoders, there are two pipeline flows. The x86 mode pipeline flow is slightly different from fused macro-op mode as depicted in the figure. When an x86 code starts up, the pipeline works exactly as the conventional x86 superscalar processor.

The co-designed processor can recognize and process the fused macro-ops instructions through a special "fuse" bit. After pairing the dependent micro-ops and sending them in an adjacent way to the memory to create a macro-op code and fetch them, these instructions are fused. Two critical stages are considered in this design: Issue stage pipeline and ALU with result forwarding logic. As a result of the need to execute back-to-back instructions and the fact that there

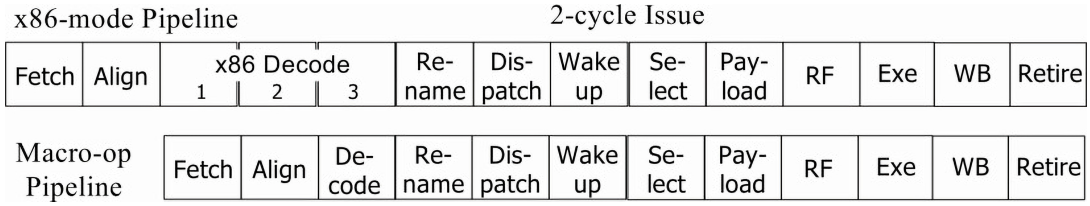

Figure 2.8 x86-mode and macro-op mode pipelines [7].

are only a few single-cycle result-register instructions, the processor is designed to have a two-stage instruction issue without any performance loss. ALU and result forwarding logic can use two cycles. The 3-input ALU can be thought of as two back-to-back ALUs, where the first ALU result is fed to the second ALU input while the other input of the second ALU is fed from the third operand. Each ALU has one cycle of latency. In reality, this design uses one 3-input ALU with two-cycle latency. There is no need for the second cycle in the regular x86 mode. The hardware is designed to take care of this, providing results like the conventional x86 processor design.

This research, measured performance through IPC with changing issue buffer size. Figure 2.9 shows the IPC scores for different configurations while changing the buffer size within the range of 16 to 64.

Four-wide co-designed x86 processor performs nearly 20% better than the baseline four-wide. The reason for that comes from the need to increase the issue buffer size results from using the macro-op execution engine and binary translation. In general, macro-op co-designed superscalar processor performed better than the conventional x86 superscalar processor using SPEC2000 benchmark.

In embedded systems, a uni-processor or a scalar processor often provides better efficiency than complex designs and is, therefore, the design of choice. Thus, the research area of this thesis does not directly overlap with the one presented in [7].

Macro-op For RISC-V

Instruction fusion requires the addition of specialized functional units. Many

mechanisms and microarchitectures are proposed to support different types of fused instructions. Many factors play a role in designing and implementing a processor that can run fused instructions or even translate instructions into other macro-op instructions. The following two discussions concern two papers that used instruction fusion on RISC-V ISA. The first paper presents the impact of fused instructions on different ISAs, focusing on RISC-V using a compiler and SPECINT2006 benchmark [8]. Meanwhile, the other modifies one stage RISC-V processor to support a single fused instruction.

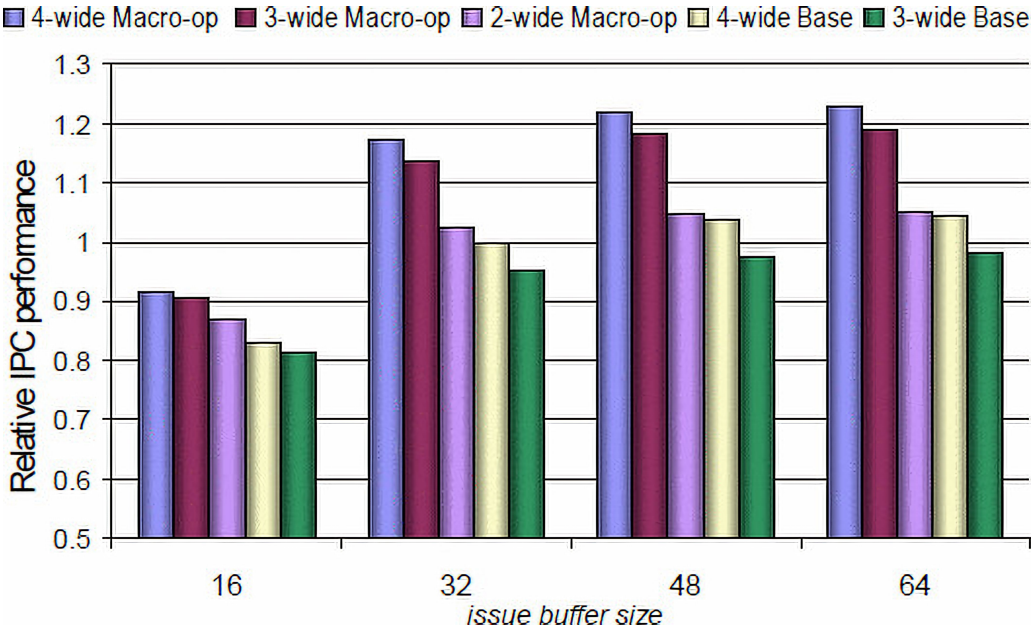

Figure 2.9 IPC performance comparison [7].

Avoiding ISA Bloat with Macro-Op Fusion for RISC-V

This paper studied the impact of adding fusion pair candidates on the dynamic instruction count for different ISAs. Using SPEC CINT2006 benchmark to compare the different ISA they show the reduction in dynamic instruction count after adding fused macro-ops. They cover a detailed explanation of the benchmark's behavior by presenting the most common loops in SPECInt 2006. In addition to that, they show eight fusion pair candidates and their effect on a processor from cost and complexity. The geometric mean of the instruction counts of different ISAs evaluated from SPECInt benchmarks normalizes to the conventional

x86-64 ISA is shown in Figure 2.10.

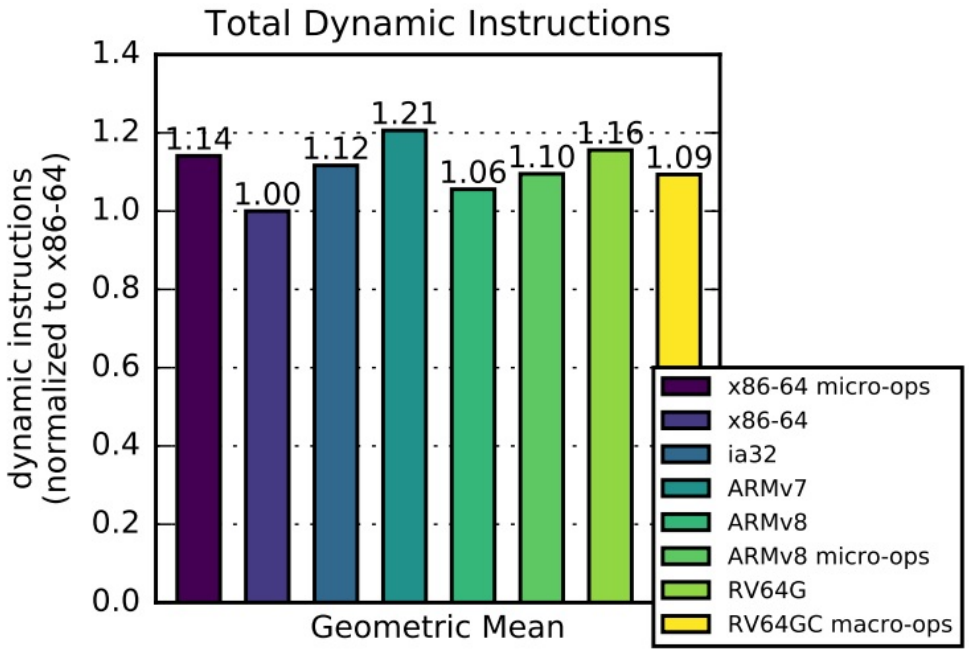

Figure 2.10 The geometric mean of the instruction counts of the twelve SPECInt benchmarks is shown for each of the ISAs, normalized to x86-64 [8].

This analysis showed that an RV64 processor could reduce its effective instruction count up to 5.4% when it supports macro-op fusion. However, the research also states that adding more complexity to the microarchitecture to support macro-op fusion may not be optimal, especially when the programs running on that processor are unknown and may result in less efficiency. Our research adds on top of this new proposed Fusion Pair Candidate and chooses these candidates in a way to reduce complexity and cost as much as possible. Additionally, modifying a processor (Rocket Chip) to support these instructions and studying the changes needed for these fused instructions on both hardware and software aspects is done.

SHA256-Efficient CPU Based on RISC-V Architecture

This analysis showed that an RV64 processor could reduce its effective instruc-

tion count up to 5.4% when it supports macro-op fusion. However, the research also states that adding more complexity to the microarchitecture to support macro-op fusion may not be optimal, especially when the programs running on that processor are unknown and may result in less efficiency. Our research adds on top of this new proposed Fusion Pair Candidate and chooses these candidates in a way to reduce complexity and cost as much as possible. Additionally, modifying a processor (Rocket Chip) to support these instructions and studying the changes needed for these fused instructions on both hardware and software aspects is done.

```
lw  ->  srliw ->  slliw ->  or ->  sext
```

The next step is the proposed instruction design. One instruction is designed to do all the above instructions called CURL. It is capable of loading data and shifting it right and left. Then, 'Or' the shifted left data with the sifted right data to get the result. Figure 2.11 shows the format of the customized added CURL instruction to RISC-V ISA.

Figure 2.11 Curl instruction format [9].

The design process explained in this paper can be presented as follows:

- Adding CURL instruction to the existing ISA by modifying the RISC-V compiler.
- Writing a C code to test the added fused macro-op instruction.
- Modify Spike and Gem5 simulators to define and recognize the functionality of the new fused instruction.

- Propose a co-design RV32-1-stage architecture capable of executing the new instruction.
- Simulate SHA256 program using Gem5 to see the impact of the fused macro-op.
- Hardware emulation using both C and Verilog to compare area, power, and performance between the modified processor and original RISCV-SODOR.

They modify the existing ALU to support the new instruction based on Figure 2.12. Overall, they managed to prove that their modified CPU is 2x times more efficient than the original RISCV-SODOR single-stage processor when it runs SHA256 F1.

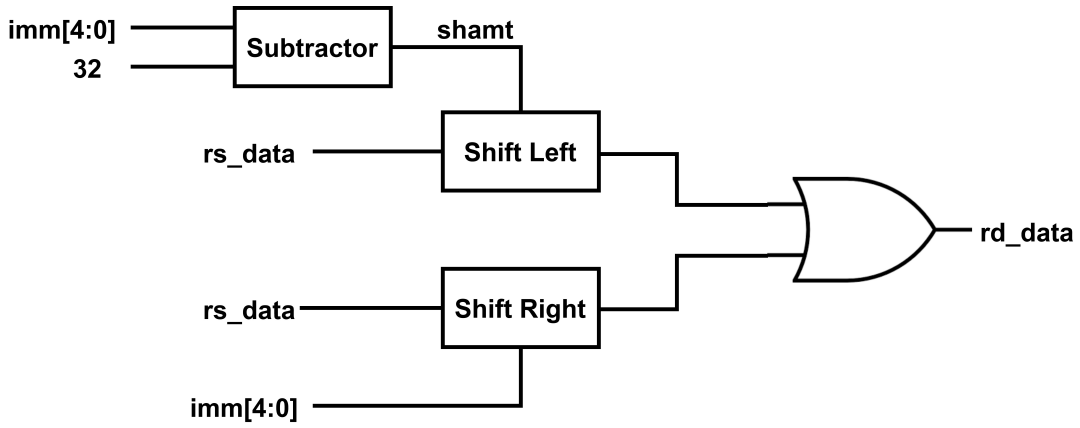

Figure 2.12 Curl Block Diagram [9].

Trace Cache

One possible way to take advantage of instruction fusion is through the instruction stream of a trace cache [21] [22] [23]. Taking advantage of the repeated loops, trace cache analyzes and collects them in what are called trace lines. These trace lines is called later on when the processor is executing the same instruction address. Instead of fetching the stream of instructions from the instruction cache, they are fetched from the trace cache using the address of the loop as an entry in the trace cache. A trace line can be created from three

branches with 16 instructions across them. A couple of studies proposed some dynamic optimization to perform on trace cache [22] [23]: instruction fusion, dead code elimination, and dependency collapsing. One of the disadvantages of using trace cache is that their performance depends on branch prediction. This disadvantage almost disappears in modern branch predictors with a high prediction rate. However, still having the recovery time when the loops end when the prediction is missed, all instructions fetched from the trace cache have to be flushed. Such optimization and others can make more sense when considering the processor's application.

Other studies used instruction fusion [24] [25] [26] [27]. "Macro-op scheduling: Relaxing scheduling loop constraints " [24] added a single cycle ALU with three operands beside the scheduler, allowing a superscalar out-of-order processor to fuse dependent instructions dynamically. One of the fused instructions they work on is combined instruction from control (branch) instruction and store address generation instruction. In addition, they used a large instruction window to overcome the gap between the processor core and the memory resulting in concealing memory latency. "Interlock Collapsing Alu's " [25] used what they call Interlock Collapsing ALUs to create "multi-operation instructions." Simply put, they created an ALU capable of doing two operations with three inputs. These operations are logical operations and 2's complement, such as:

```
//before using Interlock Collapsing Alu's
R1 = R1-R3
R2 = R2-R1
//After using Interlock Collapsing Alu's
R2 = R2-(R1-R3)
```

"Dataflow mini-graphs" [27] managed to achieve an overall processor performance gain of 2%, 6%,7%, and 12% on SPECint, CommBench, MiBench, and MediaBench, respectively, by using what they called "Dataflow mini-graphs" to fuse multiple instructions statically. Using "handles," the processor uses a chain of ALUs (3-stage non-collapsing ALU pipeline) with two reads and one write to amplify the pipeline's execution stage. After expanding the instruction for

execution by a dynamic instruction stream editor (DISE) through consulting a Mini-Graph Table, the ALU selects its input from any stage in the pipeline to achieve the required fusion. The research ended up with the best improvements using only two instruction mini-graphs.

"Static strands: safely collapsing dependence chains for increasing embedded power efficiency" [26] takes advantage of the dependence instruction chains to save energy in modern embedded systems. According to this research, the values generated within the strand feed only a single instruction and tend to be transient operands. These intermediate values can be bypassed and not written into the register file. The authors managed to obtain a 15% increment in the number of Instructions Per Cycle (IPC), which in return added energy reduction to the existing one from the issue logic (16-24% energy reduction), bypass logic (17-20% energy savings), and register file accesses (13-14% energy reduction); as a result of the values which are not written to the register file.

2.4 Common benchmarks for embedded systems

Dhrystone

It was created by Dr. Reinhold P.Weicker to measure the performance of a processor. Dhrystone consists of functions that focus on string handling. Because of the nature of this benchmark as an integer benchmark, it is considered a good embedded systems processor performance benchmark.

Coremark

Benchmark developed by EEMBC to test the functionality of the processor core (central processing units (CPUs) and microcontrollers (MCUs) used in embedded systems). It's designed to replace the antiquated Dhrystone benchmark. It contains the following algorithm: State machine (determine if an input stream contains valid numbers), which is repeated by a defined number of iterations, list processing (find and sort), which is done along with initializing data and getting the parameters, matrix manipulation (common matrix operations), and

CRC (cyclic redundancy check). After the iterations, the time is captured, and a CRC check is done to verify the results. Figure 3.7 illustrate Coremark workflow.

SPECint2006

Integer performance benchmark built to test the performance of modern server computer systems.

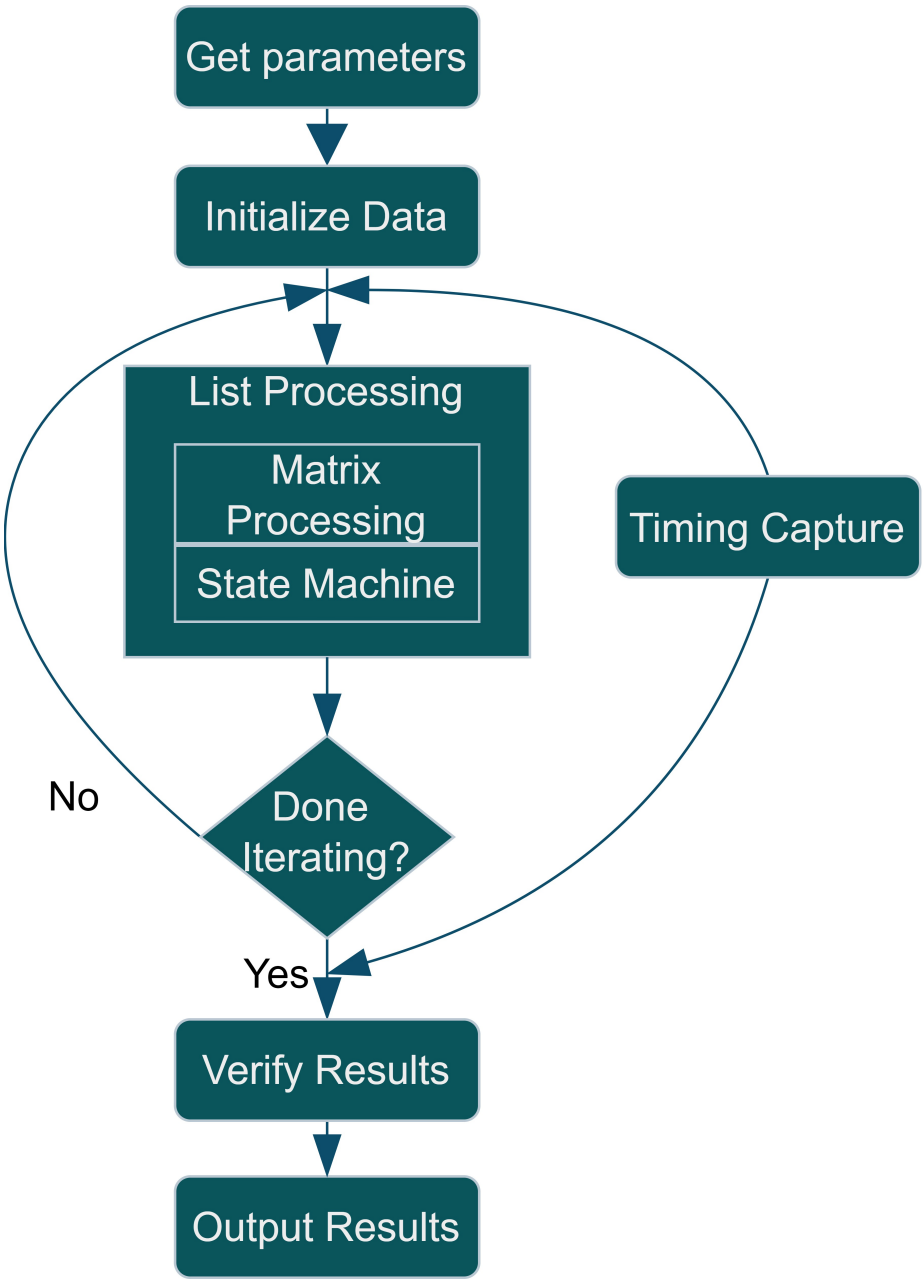

Figure 2.13 Coremark Workflow [10].

Studying a benchmark and finding the fused opportunities, then modifying the benchmark code to adopt fused instructions, is time-consuming. Dhrystone and Coremark are chosen as reference benchmarks to study the performance benefits.

2.5 Chipyard environment for processor architecture performance evaluation

2.5.1 Processor modeling

Chipyard is a framework that connects a collection of libraries and tools to provide a platform that uses commercial tools and open sources to develop SoCs. Chipyard framework provides different processor cores, such as BOOM and Rocket Core, in addition to other peripherals such as accelerators; Chipyard uses what is called Rocket Chip parameter system and Cake Pattern / Mixing to provide processor modeling that is easy to build and customize. Architects can easily build any customized SoC by using collection of parameters with predefined values to control generator architecture and having the ability to merge multiple systems components. Without going into too many details, the following example shows a small portion of a Chipyard default top that composes various traits into a fully-featured SoC.

```
class DigitalTop(implicit p: Parameters)
                        extends ChipyardSystem
  with testchipip.CanHavePeripheryCustomBootPin
  //Enables optional custom boot pin
  with testchipip.HasPeripheryBootAddrReg
  //Use programmable boot address register
  with testchipip.CanHaveTraceIO
  // Enables optionally adding trace IO
```

Chipyard provides many predefined configurations, such as BOOM and Rocket Chip configurations.

2.5.2 ISA compilation

Chipyard uses the riscv-tools software toolchain, which includes the RISC-V compiler and assembler, proxy kernel, the Berkeley Boot Loader (BBL), and functional ISA simulator (spike). Riscv-tools have a parameter that controls the ISA extensions. This parameter must be set according to the designed processor model architecture. For example, the floating point unit is optional in Rocket Chip configuration, and by default, it is used. Remove the floating point unit; the processor can no longer execute instructions from the "F" extension.

2.5.3 Benchmark execution and execution time estimation

Chipyard has two types of simulators RTL simulators and FPGA-accelerated simulators. Using the Verilator RTL simulator is the optimal choice as it is an open-source, free simulator. Compiling any design other than default using RTL simulator setting up the ***config*** parameter to one of the predefined configurations or a user-customized configuration is mandatory. After that, the User can run any code on the compiled design using riscv-tools or run the pre-packaged benchmarks.
Chipyard uses Verilator (Open-Source) and Synopsys VCS (License Required) as Software RTL Simulators. If the core is implemented correctly, it ought to be able to execute RISC-V binaries. Running the Verilator under debugging mode provides a cycle-accurate output file, including the number of cycles needed and the CPU clock frequency. Using the number of cycles that are required to finish a test or benchmark and the clock cycle of the implemented core, we can calculate the execution time using the equation:

$$CPU\ execution\ time = CPU\ clock\ cycles * Clock\ Cycle$$

CHAPTER 3

DESIGN AND ANALYSIS

This chapter discusses the design method and analysis of the proposed design. The differences are explained across macro-op fusion, micro-fusion, and instruction fusion. The fusion candidate selection process is described before presenting the fused instruction format. The last section in this chapter explores the changes to Rocket Chip organization and describes the implementation and simulation method.

3.1 Identification of fused candidates

Macro fusion, micro fusion, or instruction fusion all refer to the method of combining two or more instructions or operations into one instruction. Intel [28] describes micro-fusion as fusing multiple micro-ops from the same instruction into a single complex micro-op. In contrast, macro-fusion is done by merging multiple micro-ops from different instructions into a single complex micro-op. Instruction fusion, on the other hand, represents the idea of merging two or more instructions into a single instruction, for example, at compile time. Processors that support fusion generally have the same pipeline stages as the conventional processors. The fusion into one single fused macro-operation is done during the decoding stage or before it. This hardware optimization requires the instructions to be adjacent and have a dependency; otherwise, either a scheduler is needed, or the compiler needs to be enhanced to detect dependencies and rearrange the instructions to be adjacent without affecting the program's functionality. Fusion method targets reduction in the instruction count to achieve better performance. The following two hypothetical RISC-V codes demonstrate the potential benefit

of fusing instructions.

```
//un-fused
ADDI R1, R1, 16
loop:
ADD R5, R5, R6
ADD R8, R8, R5
ADDI R1, R1, -1
BNE R1, 0, loop
//Fused
ADDI R1, R1, 16
loop:
MADD R8, R8, R5 ,R6  //lets call this fused instruction MADD
//which is equal to R8 = R8+R5+R6
ADDI R1, R1, -1
BNE R1, 0, loop
```

The un-fused code has 65 instructions, while the fused code has 49 instructions. Therefore, the run time is reduced by 25% in this example.

Many fused instruction can be created. ALU instructions can be fused with other ALU instructions, control instructions, or memory instructions. Implementing more fused instructions leads to adding more complexity to the processor organization. Identification of fused instructions requires selection of benchmarks for the targeted processor segment. Dhrystone and Coremark integer benchmarks have been selected in this work for analysis.

In this section, fusion pair candidates are presented as described in [8]; Additional two idioms are explained which are repeated in Dhrystone benchmark. Statistical data is provided about each benchmark fusion pair candidate. Details statistical data shows the reduction in "effective" instruction count. The following idioms are good candidates for macro-op fusion.

- ***Load Effective Address***

This idiom computes the memory location effective address and sends it to a register.

```
// &(array[offset])
slli rd, rs1, {1,2,3}
add rd, rd, rs2
```

- ***Indexed Load***

This instruction enables the processor to load data from an address resulting from adding two registers.

```
// rd = array[offset]
add rd, rs1, rs2
ld rd, 0(rd)
```

- ***Clear Upper Word***

This idiom clears the upper word of a 64-bit register by zeroing the upper 32-bits.

```
// rd = rs1 & 0xffffffff
slli rd, rs1, 32
srli rd, rd, 32
```

- ***Load Immediate Idioms (LUI-based idioms)***

RISC-V ISA offers only a 12-bit immediate value. When we need to use larger values, we can use load upper immediate instruction to increase the immediate range up to 32 bits. There are two LUI idioms; the first idiom loads 32 bits to a register.

```
// rd = imm[31:0]
lui rd, imm[31:12]
addi rd, rd, imm[11:0]
```

The second idiom loads a value into the memory with an address range up to 32 bits.

```
// rd = *(imm[31:0])
lui rd, imm[31:12]
ld rd, imm[11:0](rd)
```

- ***AUIPC-based idioms***

Load global is an instruction that uses AUIPC to add an immediate to PC address, allowing the processor to access memory at arbitrary locations.

```
// ld rd, symbol[31:0]
auipc rd, symbol[31:12]
ld rd, symbol[11:0](rd)
```

After studying Dhrystone assembly code, other two AUIPC-based idioms were identified that are not mentioned in the literature. The first is AUIPC + ADDI, and the second is formed from the first idiom (AUIPC + ADDI) and load instruction. This idiom happens when we need a larger immediate value for load global.

```
// rd = imm[31:0] + PC
auipc rd, symbol[31:12]
addi rd, imm[11:0]
// ld rd, 0(imm[31:0] + PC)
auipc rd, symbol[31:12]
addi rd, imm[11:0]
ld rd,0(rd)
```

- ***Additional RISC-V Macro-op Fusion Pairs***

This section shows some addition fused instructions that have not been implemented in this research.

Load-pair/Store-pair

One of the famous idioms used by ARMv8 is load-pair/store-pair, which allow the processor to read/write up to 128-bits from/into two registers using one single instruction. This is possible to implement in RISC-V by fusing back-to-back loads/stores.

```
// ldpair rd1,rd2, [imm(rs1)]
ld rd1, imm(rs1)
ld rd2, imm+8(rs1)
```

Load-pair/Store-pair are not cheap, as they require adding extra wire-port to the register file and two read-ports to the memory. Furthermore, adding complex load/store units always leads the processor to suffer from additional complexity.

Wide Multiply/Divide & Remainder

Multiplication in RISC-V generates a product of size 2*xlen. Two separate instructions are needed to get the full 2*xlen of the product(MUL and MULH).

```
MULH[[S]U] rdh, rs1, rs2
MUL rdl, rs1, rs2
```

RISC-V user-level manual recommends fusing this idiom along with divide/remainder idiom.

```
DIV[U] rdq, rs1, rs2
REM[U] rdr, rs1, rs2
```

Post-indexed Memory Operations

This idiom performs a load/store from a memory address and increments the base memory address.

```
// ldia rd, imm(rs1)
ld rd, imm(rs1)
add rs1, rs1, 8
```

Post-indexed load operations require two write-ports, making it not profitable for all micro-architectures [20].

AUIPC+JALR

AUIPC can be used when jumping to address more than 1 MB in the distance.

```
auipc rd, symbol[31:12]
jalr rd
```

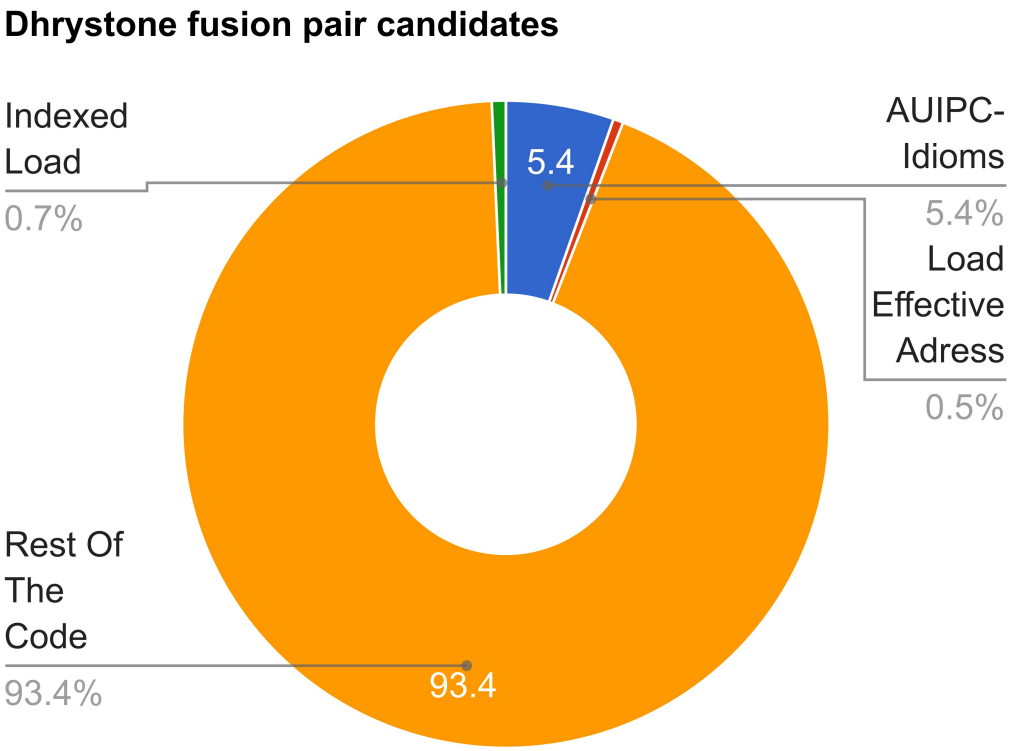

Figure 3.1 Dhrystone fusion pair candidates.

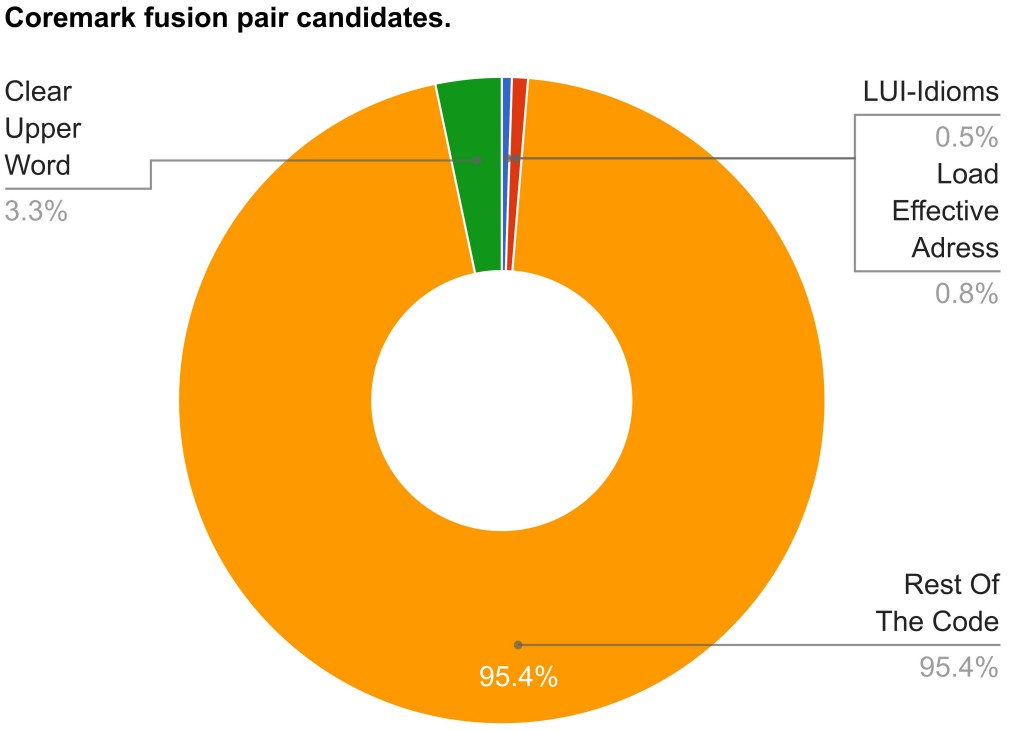

Figure 3.2 Coremark fusion pair candidates.

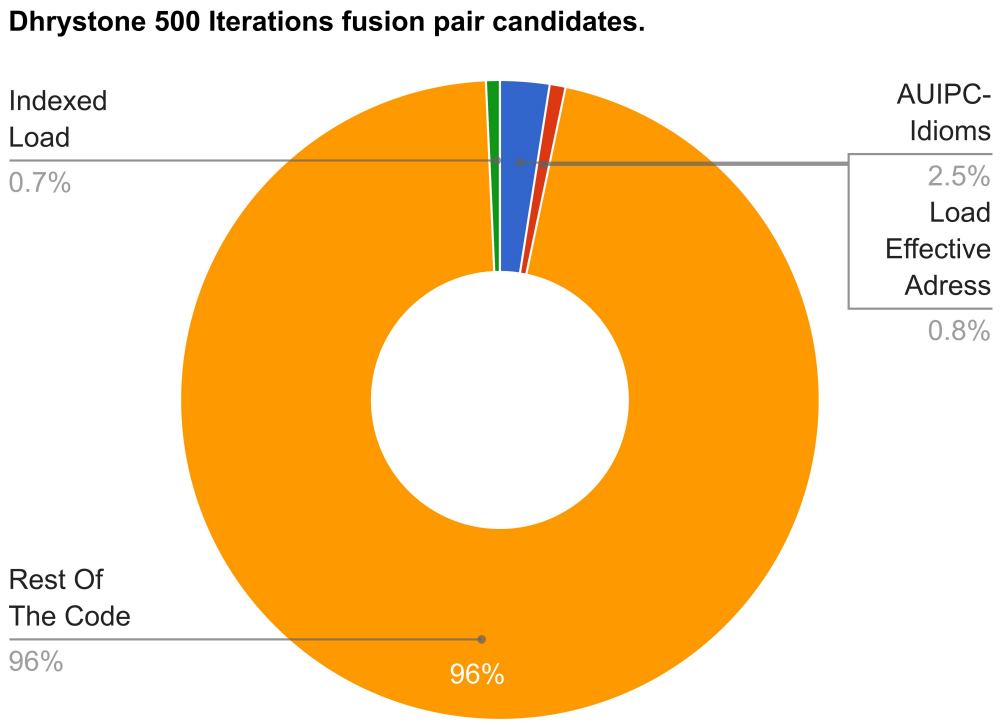

Figure 3.3 Dhrystone 500 iterations fusion pair candidates.

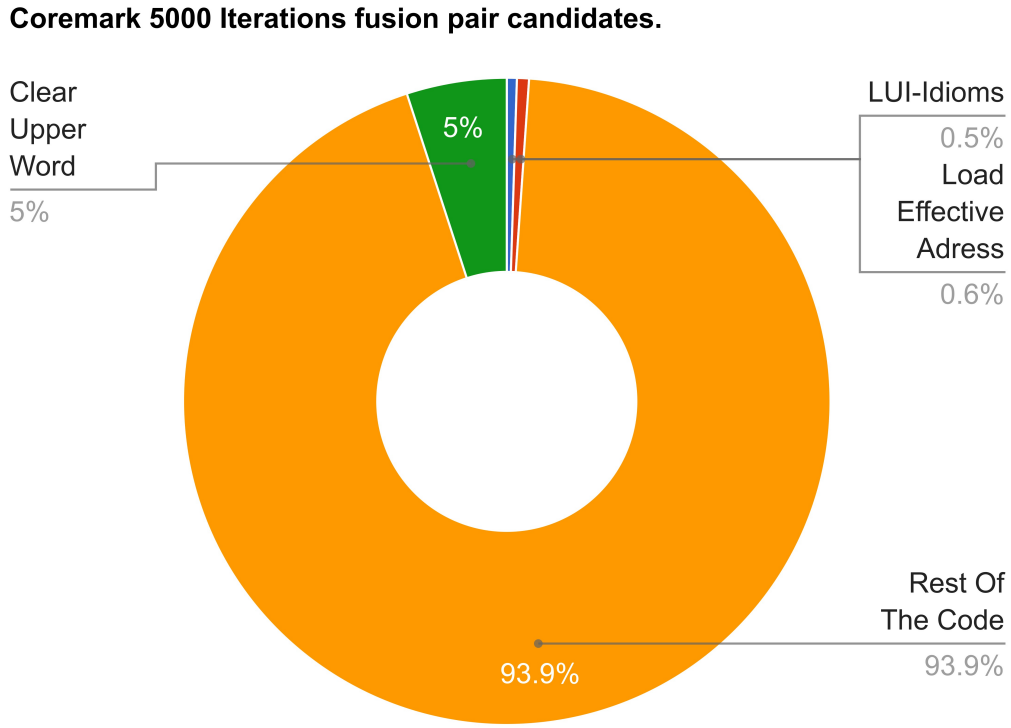

Figure 3.4 Coremark 5000 iterations fusion pair candidates.

Using riscv-tools, assembly codes for both Coremark and Dhrystone were obtained. After that, a python code is written to find the fusion pair candidates in each assembly code and collect statistical data. Figure 3.1 and figure 3.2 show the percentage of each fusion pair.

The next step is to run these benchmarks on the original Rocket Chip environment to collect data about the reduction in instruction count in each benchmark. Therefore, the Verilator is set to verbose mode, and a cycle-accurate output is obtained. Moreover, Dhrystone and Coremark were simulated under 500 and 5000 iterations, respectively. Afterward, a Python code is written to collect statistical data about each benchmark's instruction count reduction, as shown in figure 3.3 and figure 3.4.

It can be concluded that three idioms contribute to reducing instruction count in each benchmark. Based on that, five candidates are chosen to implement (Clear upper word, Load effective address, LUI-idioms, AUIPC-idioms, and Indexed load). Nevertheless, AUIPC idioms are implemented except (AUIPC+JALR). Section 3.3 demonstrates how fusion pair candidates are implemented in the hardware.

3.2 Fused Instruction Format

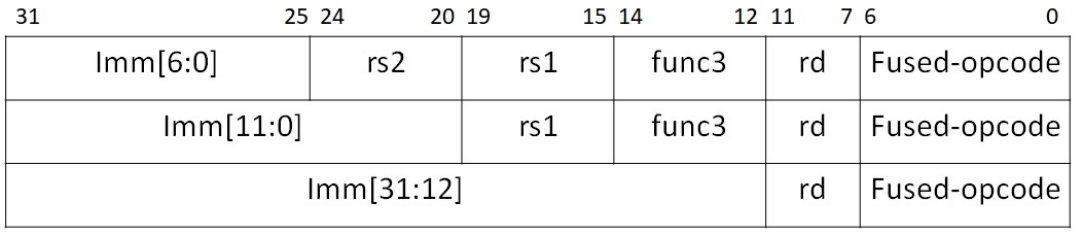

Figure 3.5 Fused Instruction Format.

Designing a customized instruction format must follow the general format rules of RISC-V ISA. Since Rocket core architecture propagates the instruction to all

the pipeline stages, using the same RISC-V format with different opcodes for the fused instructions is less complex and keeps the ISA clean and simple. Using a format for fused instructions is not essential. Still, it provides more flexibility to the translator or the compiler.

3.3 Changes to Rocket Chip Organization

This research focuses on integer extension ISA to support embedded systems applications; thus, we set riscv-tools to RV64IM. Rocket Chip offers many parameters that can be changed as knobs to customize the design, such as the number of floating-point pipeline stages, the cache parameters, and TLB sizes. RISC-V tools help implement a new processor without re-designing everything from scratch. Furthermore, since it allows different models to be connected without re-designing them again, it is easier to implement a new architecture at a low level of abstraction. A five-stage pipeline streams one instruction per cycle if there are no stalls or flushes. Suppose the processor fuses two instructions; the processor saves an extra cycle whenever it executes this fusion. Having a loop with fused instructions leads to an increase in the reduced number of cycles. Loops typically tend to iterate many times, potentially using such optimization techniques. Before going to the modified processor pipeline datapath, let us observe the original datapath of the Rocket Core architectural diagram (Figure 3.6).

Design Choices

Rocket core 5-stage executes instructions like any single-issue core. One instruction is issued each cycle, and each instrucioın needs five stages to complete. However, Rocket Chip supports branch prediction, TLB, and memory hierarchy. Therefore, the instruction fetch can be held whenever memory hierarchy misses, or branch misprediction occurs. Fusion pair candidates can be categorized into two types. The first type needs two arithmetic operations, and the second one needs 32-bit Immediate.

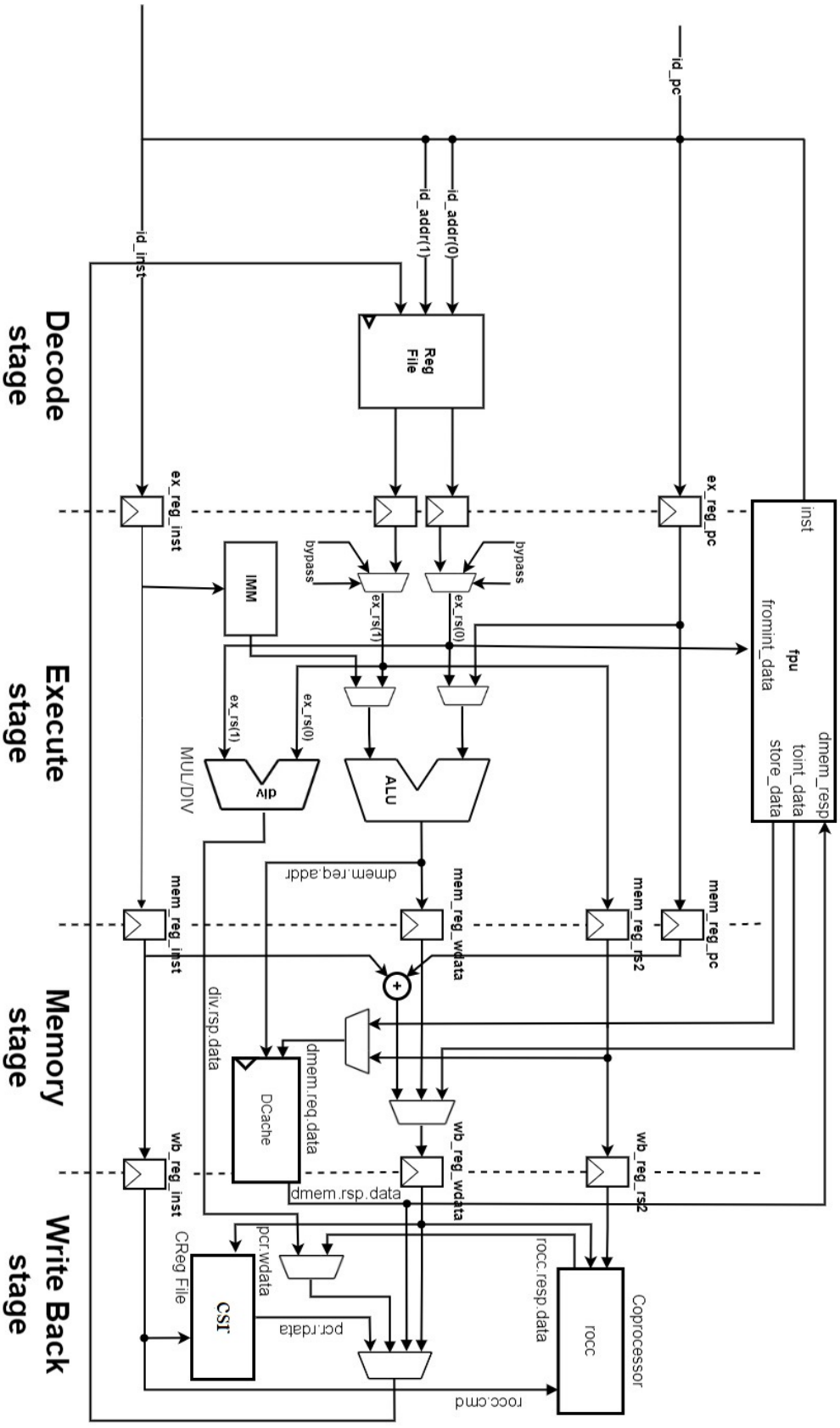

Figure 3.6 Rocket Core Architectural Diagram.

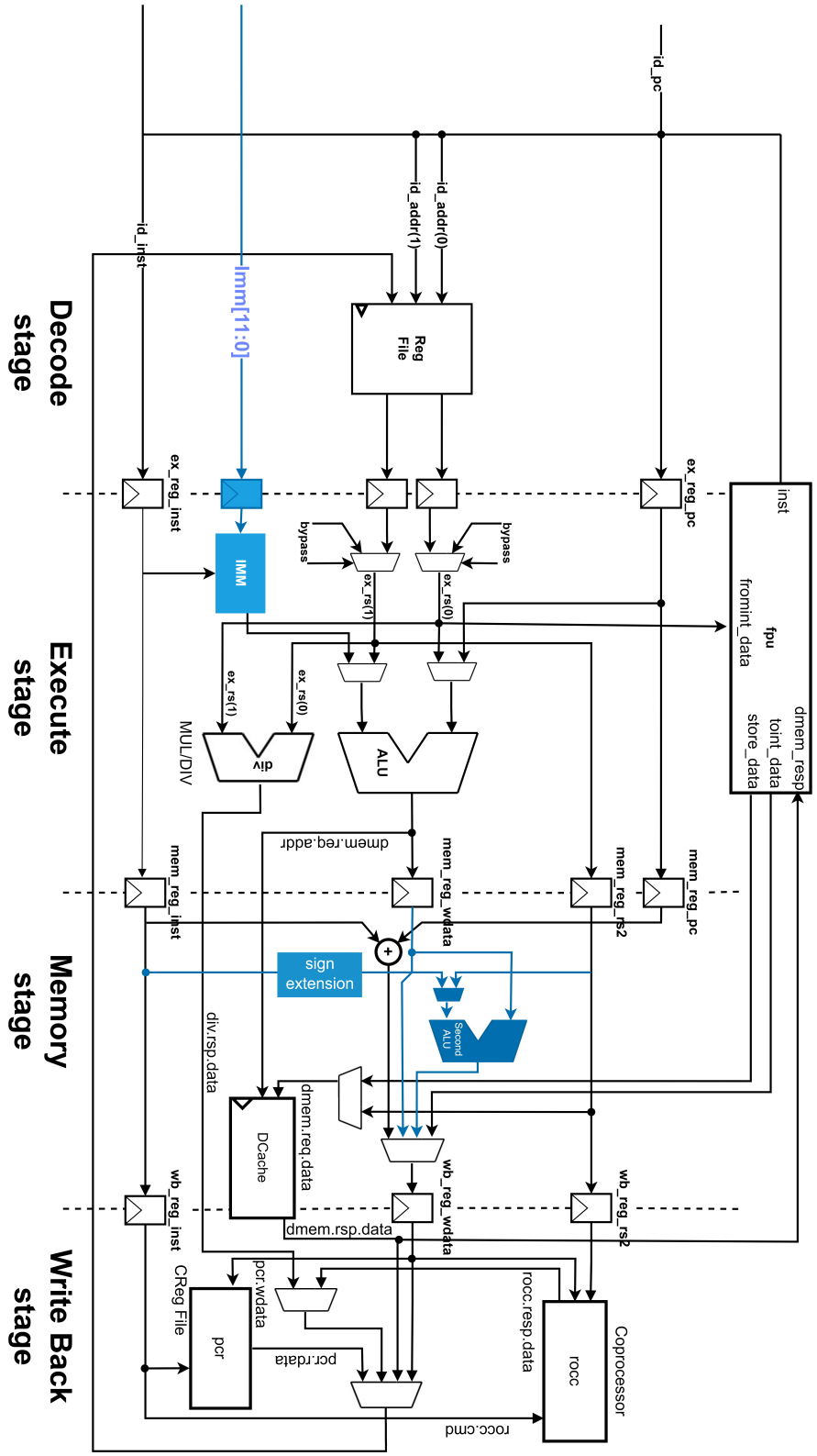

Figure 3.7 Modified Rocket Core Architectural Diagram.

The proposed work added to the reference Rocket Chip RISC-V processor a mux, second ALU, control fused signals, 12-bit execute stage register, and sign extension unit and modified the Immediate unit and the mux before the write-back register data to support executing fused instructions. Figure 3.7 illustrates the modified Rocket core architectural diagram.

The Immediate unit was modified to append the Immediate[11:0] with Immediate[31:12] to form the 32-bit immediate when needed; otherwise, it works as the original Immediate unit. Another way to feed the processor with the 32-bit Immediate would be through a 32-bit execute register to hold the Immediate value and a mux after the Immediate unit controlled by a fused control signal to select between 32-bit Immediate and the Immediate unit output. Nevertheless, this increases the number of added register bits.

The added Second ALU is used to perform the second arithmetic operation that the rest of the fusion pair candidates need. Moreover, the second ALU is controed by two muxes and fused control signals. The first mux selects between the second operand memory register and the sign-extended Immediate from the memory instruction register. Meanwhile, besides its function, the second mux was modified to work as a bypass unit for the second ALU. However, the second ALU is added to the memory stage instead of modifying the existing ALU to maintain the system's frequency and avoid adding latency to the execute stage by taking advantage of memory latency.

Fused control signals are generated by comparing the fused opcode with the one that resides in the instruction registers.

Rocketchip can be modified to fetch and decode up to two 4-byte instructions every cycle [8]. Controlling the processor with a fused control signal to fetch two instructions provides an opportunity to feed the pipeline with the 12-bit Immediate needed by some fusion pair candidates.

One of the critical factors in CPU design is the speedpath and its effect on performance. Hence, design choices were made to avoid affecting the speedpath. Therefore, considering all the possibilities of logic changes needed to implement

a processor capable of executing the fusion pair candidates, decisions were made to keep the Rocket Chip clock cycle time the same as the reference. It can be seen that the second ALU, mux, and sign extension unit were added in the memory stage to take advantage of memory latency and work in parallel. Moreover, since the memory latency is expected to be more than the latency of the speedpath of the added hardware, the speedpath of the memory stage will not change. Furthermore, modifying the Immediate unit to have an extra condition for fused instruction will not change the speedpath in the execute stage since the Multiplication/Division unit latency is more than the ALU latency resulting in compensating for the additional time needed by the modified Immediate unit. Lastly, the 12-bit Immediate in the execute stage is not affecting any speedpath. Overall, The modified Rocket clock cycle time did not change as a result of fused instruction changes.

- **Load Effective Address**

Figure 3.8 shows an overlay of how Load Effective Address is executed in the modified Rocket core.

Fetch stage

The instruction is fetched from the instruction cache, and the program counter **PC** increases by four.

Decode stage

The two operand registers values are loaded from the register file and sent to the execute stage registers.

Execute stage

The first ALU muxes select the first register and Immediate values. Next, The ALU shifts the value of the first register by a number of bits specified in the Immediate and sends the output to the write-data memory register. Furthermore, the second operand value is sent to the memory stage register.

Memory stage

The second ALU mux selects the second operand memory register. Then the second ALU adds the second operand value to the first ALU output. The write-data mux selects the output of the second ALU and sends it to the write-back write-data register.

Write-back stage

The second ALU output is stored in the destination register.

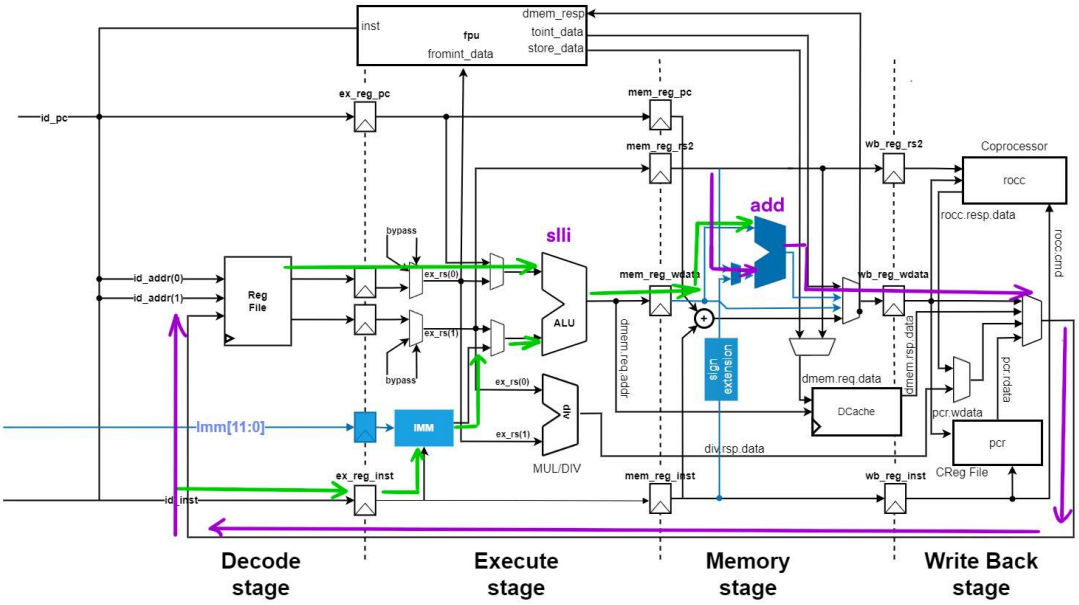

Figure 3.8 LEA overlay.

- **Indexed Load**

Figure 3.9 shows how Indexed Load is executed in the modified Rocket core.

Fetch stage

The instruction is fetched from the instruction cache, and the program counter **PC** increases by four.

Decode stage

The two operand registers values are loaded from the register file and sent to the execute stage registers.

Execute stage

The first ALU muxes select the first and second register values. Next, The first ALU adds the two values and sends the output to the memory stage write data register.

Memory stage

The First ALU output is sent to the data cache as a load address.

Write-back stage

The loaded data is stored in the destination register.

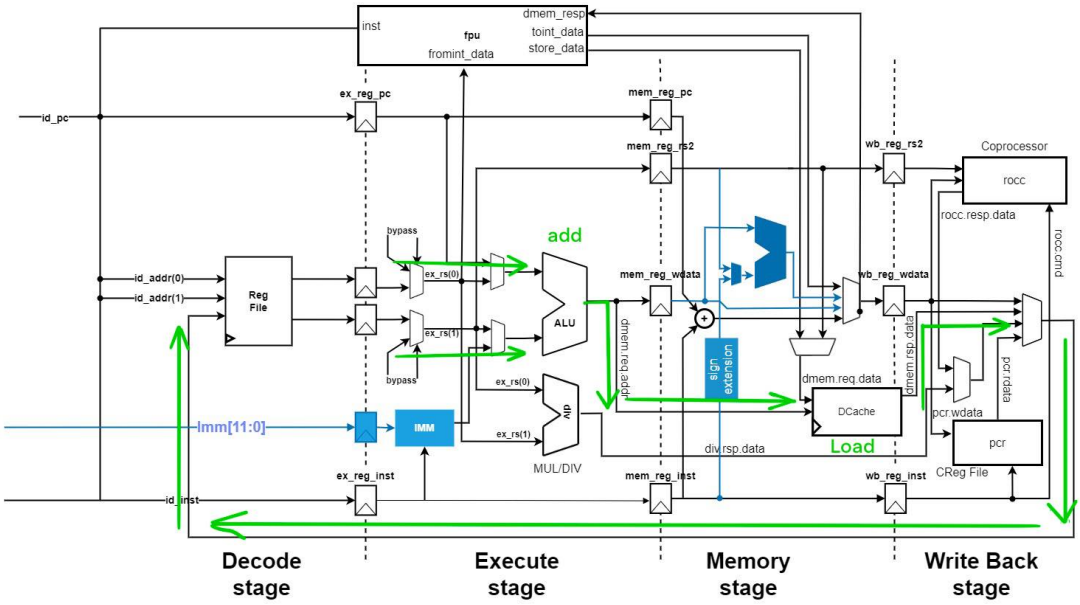

Figure 3.9 Indexed Load overlay.

- **Clear Upper Word**

Figure 3.10 shows how this instruction is executed in the modified Rocket core.

Fetch stage

The instruction is fetched from the instruction cache, and the program counter **PC** increases by four.

Decode stage

The two operand registers values are loaded from the register file and sent to the execute stage registers.

Execute stage

The first ALU muxes select the first register and Immediate values. Next, The ALU shifts the value of the first register left by a number of bits specified in the Immediate and sends the output to the write-data memory register.

Memory stage

The second ALU mux selects the Sign-extended Immediate. Then the second ALU shifts the value of the first ALU output right by a number of bits specified in the Immediate. The write-data mux selects the output of the second ALU and sends it to the write-back write-data register.

Write-back stage

The second ALU output is stored in the destination register.

- **Load Immediate Idioms (LUI-based idioms)**

Figure 3.11 and figure 3.12 shows how LUI idioms are executed in the modified Rocket core.

Fetch stage

The instruction is fetched from the instruction cache. However, a fused control signal is sent to the Instruction cache to fetch two instructions in the next cycle. Hence, the program counter **PC** increases by eight instead of four in the next cycle.

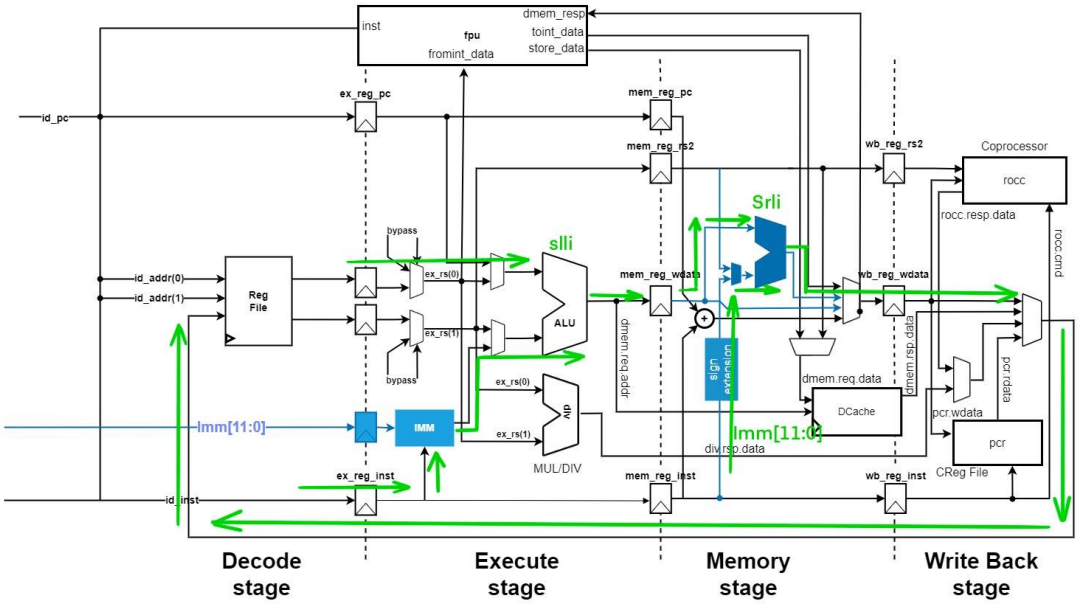

Figure 3.10 Clear Upper Word overlay.

Decode stage

One of the two fetched instructions holds the lower 12-bit Immediate value and is sent directly to the decode stage. Meanwhile, The two operand registers values are loaded from the register file and sent to the execute stage registers.

Execute stage

The Immediate unit appends the lower 12-bit Immediate with the upper 20-bits. Then the first ALU muxes select the first register and Immediate values. Followed by adding the Immediate value with register zero, and the output is sent to the write-data memory register.

Memory stage

The First ALU output is sent to the data cache as a load address for the LUI+Load idiom, whereas it is sent to the write-back write-data register for the LUI+addi idiom.

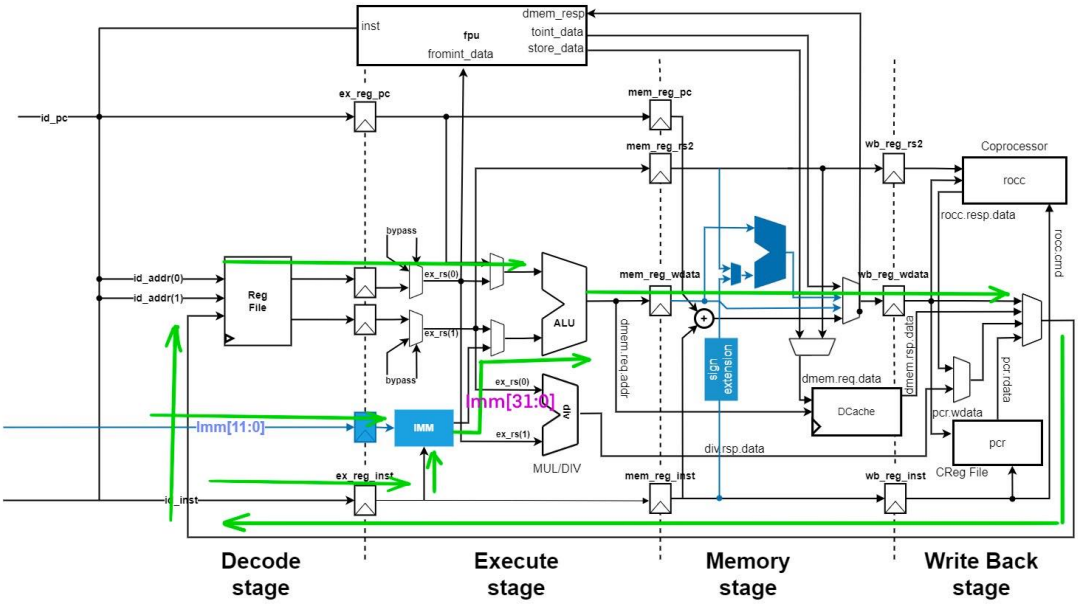

Figure 3.11 (LUI+ADD) overlay.

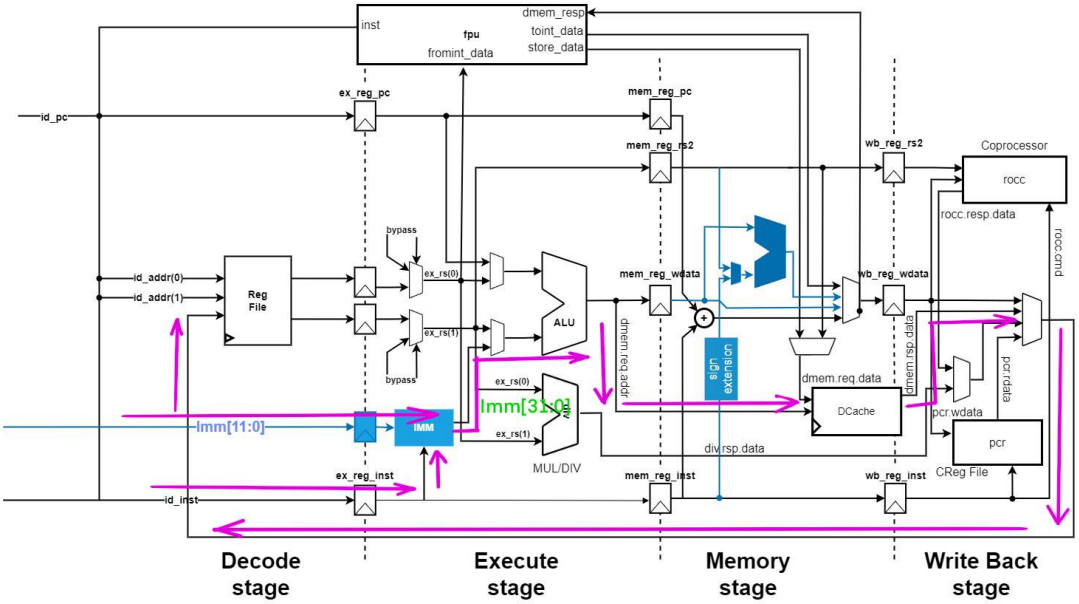

Figure 3.12 (LUI+Load) overlay.

Write-back stage

For the LUI+addi idiom, the write-back write-data register value is stored in the destination register. In contrast, the LUI+Load idiom stores the loaded data in the destination register.

- **AUIPC-based idioms**

Figure 3.13 and figure 3.14 demonstrate overlays on how AUIPC idioms are executed in the modified Rocket core. These idioms behave the same as LUI idioms with only one difference the Immediate is added to program counter **PC** instead of register zero.

Fetch stage

The instruction is fetched from the instruction cache. However, a fused control signal is sent to the Instruction cache to fetch two instructions in the next cycle. Hence, the program counter **PC** increases by eight instead of four in the next cycle.

Decode stage

One of the two fetched instructions holds the lower 12-bit Immediate value and is sent directly to the decode stage. Meanwhile, The two operand registers values are loaded from the register file and sent to the execute stage registers.

Execute stage

The Immediate unit appends the lower 12-bit Immediate with the upper 20-bits. Then the first ALU muxes select the program counter and Immediate values. Followed by adding the Immediate value with register zero, and the output is sent to the write-data memory register.

Memory stage

The First ALU output is sent to the data cache as a load address for the LUI+Load idiom, whereas it is sent to the write-back write-data register for the LUI+add idiom.

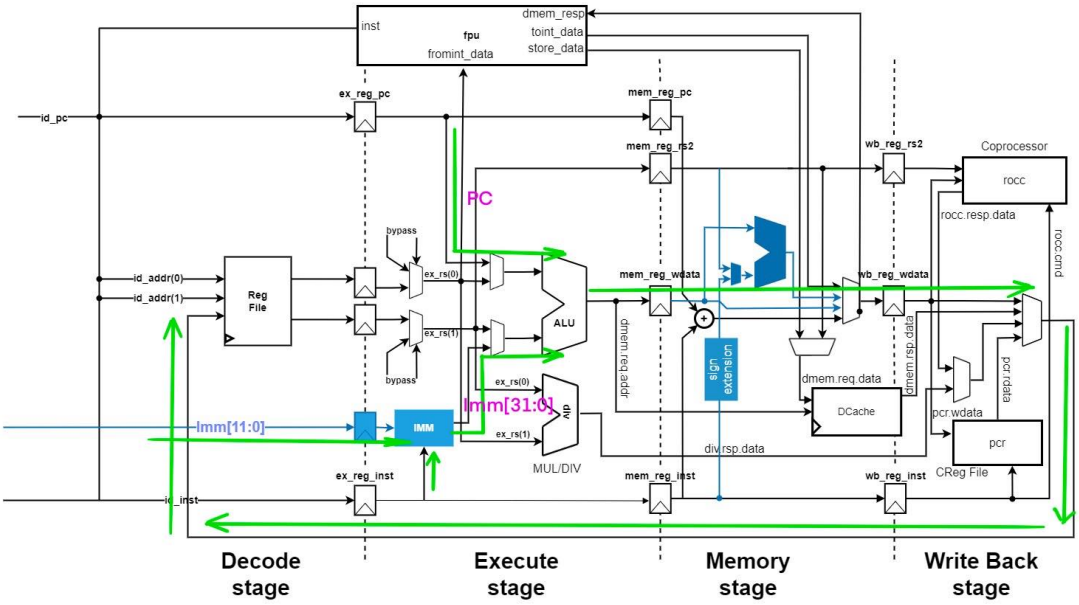

Figure 3.13 (AUIPC+ADD) overlay.

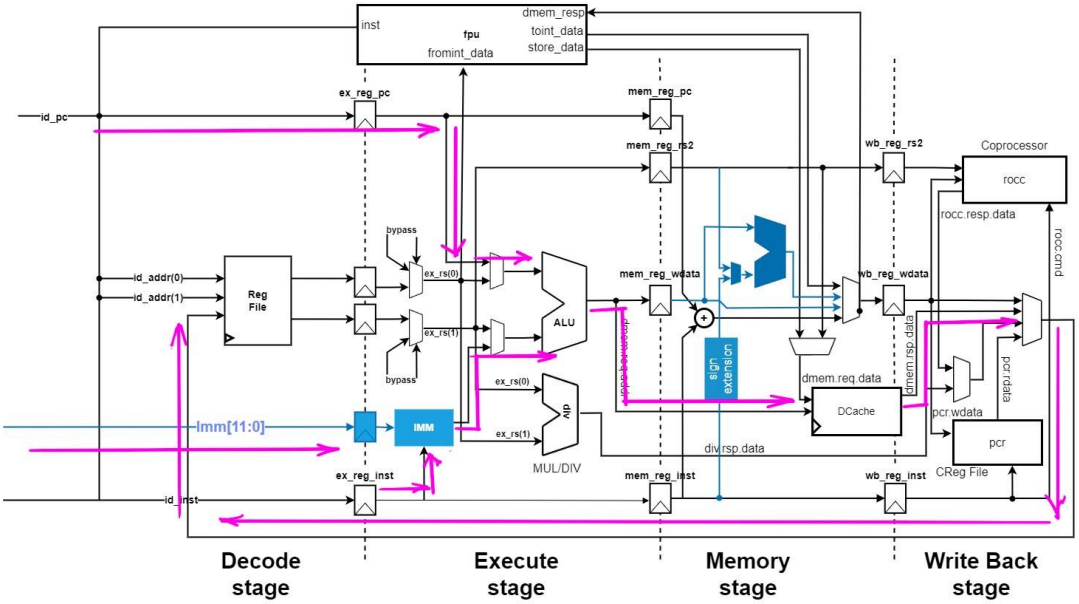

Figure 3.14 (AUIPC+Load) overlay.

Write-back stage

For the AUIPC+addi idiom, the write-back write-data register value is stored in the destination register. In contrast, the AUIPC+Load idiom stores the loaded data in the destination register.

Implementation and Simulation

Scala codes of the chipyard environment are modified to implement the proposed Rocket Chip design. Furthermore, benchmark codes are manually searched for fusion pairs candidates and replaced with the fused instructions. In contrast, the Verilator is used to simulate the modified binary files of the benchmarks. Moreover, as mentioned in the literature review, the Verilator output file is used to obtain execution time for both benchmarks. Modifying the Rocket Chip will result in a slight increment in power and area. However, the reduction in execution time will reduce the processor's energy.

CHAPTER 4

RESULTS AND DISCUSSIONS

Dhrystone and Coremark are run on the modified Rocket Chip to show the performance benefits of Macro-Op Fusion for RISC-V embedded processors. Dhrystone benchmark provides two scores to measure performance Microseconds and Dhrystones Per Second.

$$Microseconds = \frac{Runtime}{Number\ of\ iterations} * \frac{1000000}{frequency}$$

$$Dhrystones\ Per\ Second = \frac{frequency * Number\ of\ iterations}{Runtime}$$

First, Dhrystone is run with the default riscv-tools 500 iterations, and the results in table 4.1 are obtained. However, the reduction in execution time was not sufficiently noticeable. Hence, Dhrystone was run for more number of iterations table 4.2.

Table 4.1 Dhrystone 500 Iterations Results.

	Original Rocket Core	Modified Rocket Core
Dhrystones Per Second	2239	2349
Microseconds	446	425
Eff. Inst. Counts	234563	225180
Eff. Inst. Counts	234563	225180
Reduction in Eff. Inst. Counts = 4%		

Coremark is run with 5000 iterations. Using the two equations listed below and the Verilator clock cycle and CPU clock cycles reports, the reduction in execution time is obtained and summarized in table 4.3 and table 4.4.

$$CPU\ execution\ time = CPU\ clock\ cycles * Clock\ Cycle$$

Table 4.2 Dhrystone Results With Different Numbers Of Iterations.

Iterations	Original RC Inst. Counts	Modified RC Inst. Counts
10	16273	15948
100	56399	54594
400	190057	182645
500	234563	225180
10,000	390617185	373624397
500,000	390617185	373624397

$$Reduction\ in\ execution\ time = \frac{original\ time - modified\ time}{original\ time} * 100\%$$

Table 4.3 Reduction In Execution Time For Each Fusion Pair.

Fusion candidate	Coremark	Dhrystone
LEA	0.4%	0.67%
Indexed Load	——	0.93%
Clear Upper Word	4.5%	——
LUI-based idioms	0.5%	——
AUIPC-based idioms	——	2.2%

Table 4.4 Benchmarks Overall Performance Improvement.

Benchmark	Execution time reduction
Coremark	5.4%
Dhrystone	3.8%

PLEASE REWRITE THIS AS:

The new modified Rocket Core has increased the overall processor performance by decreasing the execution time by 5.4% and 3.8% for Coremark and Dhrystone benchmarks, respectively.

According to [29], Coremark runs under approximately 120mW for a Rocket Chip case study. In recent publications, the added functional units can be estimated to consume power between 0.5mW and 2.5mW. Based on these findings, the overall reduction in the energy dissipation of Coremark for 5000 iterations can be calculated as 5%.

As the number of iterations increases, the number of fused instructions executed increases, making the rest of the code residing outside the loops impact decrease

and vanish at some point.

These observations will result in a reduction in instruction count while the number of iterations increases.

All the previously proposed methods have done the fusion technique on super-scalar processors except [9], which they have applied on a 1-stage RISC-V-SODOR processor.

Macro-op fusion is not only a technique for high-performance super-scalar cores, but single-issue cores with no compressed ISA support like the RV64G 5-stage Rocket processor can also benefit [29]. "The renewed case for the reduced instruction set computer: Avoiding ISA bloat with macro-op fusion for RISC-V". [8] evaluates the SPECINT2006 benchmark performance using effective instruction count by running it on the SPIKE ISA simulator and provides a design proposal on adding macro-op fusion to the Berkeley Rocket in-order core.

This research has added fusion idiom (AUIPC+ADD), designed the Berkeley Rocket in-order core to support macro-op fusion, and studied the performance benefit using execution time on both Dhrystone and Coremark benchmarks. Compared to [9], this research has applied the fusion technique on a 5-stage RISC-V processor instead of a 1-stage processor and used five fused instructions instead of one. Table 4.5 Compares the proposed method to these two methods in the literature [8] [9].

Table 4.5 Proposed Work Vs Literature.

Research	Processor	Benchmark	Reduction in instruction count
Literature [9]	1-stage	SHA256	50%
Literature [8]	Compiler	SPECINT2006	5.4%
This research	5-stages	Coremark	6.1%
This research	5-stages	Dhrystone	4.4%

In conclusion, we have learned that analyzing the software level is essential in such optimization techniques and plays a significant role in the obtained results.

CHAPTER 5

CONCLUSION AND FUTURE WORK

Our modified processor proved that using instruction fusion as an optimization technique can increase the overall performance, especially for application-specific processors. In particular, the Rocket Chip processor that supports instruction fusion could see a 3.8% and 5.4% reduction in its execution time for Dhrystone and Coremark benchmarks, respectively. There are too many constraints and trade-offs in processor optimization techniques. Therefore, Instruction fusion may not be a good idea for a general-purpose processor because it may not be applicable for future applications, programs, and compilers since it can not be predicted that they will have sufficient fusion pairs candidates. However, the behavior of the ISAs and standard programs can assure instruction fusion opportunities even for general-purpose processors. In the end, We can say that instruction fusion is a powerful optimization technique if it is applied to a processor that has specific known applications, such as embedded processors.

5.1 Future Work

In the future, instruction fusion can be applied to encryption technologies to support future internet infrastructure after investigating its behavior. In addition, One of the leading technologies in computer architecture these days is accelerators. Specialized hardware accelerators can provide an alternative to harness a wider machine's potential and the abundance of transistors in a die. These accelerators have been introduced to support ISAs. By changing the design of accelerator hardware, such optimizations can be applicable under a reasonable design complexity and power budget. Two types of accelerators can use the

potential of macro-op fusion FU-based accelerators and Coarse-grained accelerators [30]. Moreover, an energy study can be done on the proposed modified design.

REFERENCES

[1] T. Jamil, "Risc versus cisc," *Ieee Potentials*, vol. 14, no. 3, pp. 13–16, 1995.

[2] A. D. George, "An overview of risc vs. cisc," in *Proceedings The Twenty-Second Southeastern Symposium on System Theory.* IEEE Computer Society, 1990, pp. 436–437.

[3] F. H. Khan, M. A. Pasha, and S. Masud, "Advancements in microprocessor architecture for ubiquitous ai—an overview on history, evolution, and upcoming challenges in ai implementation," *Micromachines*, vol. 12, no. 6, p. 665, 2021.

[4] A. Waterman, Y. Lee, D. Patterson, K. Asanovic, V. I. U. level Isa, A. Waterman, Y. Lee, and D. Patterson, "The risc-v instruction set manual," *Volume I: User-Level ISA', version*, vol. 2, 2014.

[5] B. A. Research, "Chipyard documentation," *URL: https://chipyard.readthedocs.io/en/stable/*.

[6] B. Keller, "Risc-v, spike, and the rocket core," *CS250 Laboratory*, vol. 2, 2013.

[7] S. Hu, I. Kim, M. H. Lipasti, and J. E. Smith, "An approach for implementing efficient superscalar cisc processors," in *The Twelfth International Symposium on High-Performance Computer Architecture, 2006.* IEEE, 2006, pp. 41–52.

[8] C. Celio, P. Dabbelt, D. A. Patterson, and K. Asanović, "The renewed case for the reduced instruction set computer: Avoiding isa bloat with macro-op fusion for risc-v," *arXiv preprint arXiv:1607.02318*, 2016.

[9] T.-Y. Wu, M.-K. Lin, and W.-K. Chang, "Sha256-efficient cpu based on risc-v architecture."

[10] S. Gal-On and M. Levy, "Exploring coremark a benchmark maximizing simplicity and efficacy," *The Embedded Microprocessor Benchmark Consortium*, 2012.

[11] A. Akram and L. Sawalha, "The impact of isas on performance," in *Workshop on Duplicating, Deconstructing and Debunking (WDDD) co-located with 44th International Symposium on Computer Architecture (ISCA), Toronto, Canada*, 2017.

[12] D. A. Patterson and D. R. Ditzel, "The case for the reduced instruction set computer," *ACM SIGARCH Computer Architecture News*, vol. 8, no. 6, pp. 25–33, 1980.

[13] D. A. Patterson and C. H. Sequin, "Risc i: A reduced instruction set vlsi computer," in *25 years of the international symposia on Computer architecture (selected papers)*, 1998, pp. 216–230.

[14] D. A. Patterson and J. L. Hennessy, "Computer organization and design: the hardware/software interface, (rev. ed. of: Computer organization and design/john l. hennessy, david a. patterson. 1998.)," 2012.

[15] M. W. Welker and O. A. Place, "Amd processor performance evaluation guide," *ADVANCED MICRO DEVICES One AMD Place*, 2003.

[16] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams *et al.*, "The landscape of parallel computing research: A view from berkeley," 2006.

[17] G. S. S. Gurvinder Singh, AditiTrivedi, "Emerging trends in embedded processors," *Journal of Engineering Research and Applications (IJERA) ISSN*, vol. 4, no. 5, pp. 77–80, May 2014.

[18] A. Waterman, Y. Lee *et al.*, "Spike, a risc-v isa simulator," *repository officiel: https://github. com/riscv/riscv-isa-sim*, 2011.

[19] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway, "The amd opteron processor for multiprocessor servers," *IEEE Micro*, vol. 23, no. 2, pp. 66–76, 2003.

[20] S. Gochman, "The intel pentium m processor: microarchitecture and performance," *Intel technology journal*, vol. 7, no. 2, 2003.

[21] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace cache: a low latency approach to high bandwidth instruction fetching," in *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29.* IEEE, 1996, pp. 24–34.

[22] W. Zhang, S. Checkoway, B. Calder, and D. M. Tullsen, "Dynamic code value specialization using the trace cache fill unit," in *2006 International Conference on Computer Design.* IEEE, 2006, pp. 10–16.

[23] D. H. Friendly, S. J. Patel, and Y. N. Patt, "Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors," in *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture.* IEEE, 1998, pp. 173–181.

[24] I. Kim and M. Lipasti, "Macro-op scheduling: Relaxing scheduling loop constraints," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.* IEEE, 2003, pp. 277–288.

[25] S. Vassiliadis, J. Phillips, and B. Blaner, "Interlock collapsing alu's," *IEEE Transactions on Computers*, vol. 42, no. 7, pp. 825–839, 1993.

[26] P. G. Sassone, D. S. Wills, and G. H. Loh, "Static strands: safely collapsing dependence chains for increasing embedded power efficiency," in *Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, 2005, pp. 127–136.

[27] A. Bracy, P. Prahlad, and A. Roth, "Dataflow mini-graphs: Amplifying superscalar capacity and bandwidth," in *37th International Symposium on Microarchitecture (MICRO-37'04).* IEEE, 2004, pp. 18–29.

[28] I. Coorporation, "Intel 64 and ia-32 architectures optimization reference manual, february 2022," *URL: https://software.intel.com/sites/default/files/managed/9e/bc/64-ia-32-architectures-optimization-manual.pdf.*

[29] D. Kim, A. Izraelevitz, C. Celio, H. Kim, B. Zimmer, Y. Lee, J. Bachrach, and K. Asanović, "Strober: Fast and accurate sample-based energy simulation for arbitrary rtl," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 128–139, 2016.

[30] A. Deb, J. M. Codina, A. Gonz *et al.*, "A co-designed hw/sw approach to general purpose program acceleration using a programmable functional unit," in *2011 15th Workshop on Interaction between Compilers and Computer Architectures*. IEEE, 2011, pp. 1–8.

[31] Open4Tech, "Memory layout of embedded c programs," *URL: https://open4tech.com/memory-layout-embedded-c-programs/*.

[32] A. Akram, "A study on the impact of instruction set architectures on processor's performance," 2017.

[33] A. S. PATIL and U. J. TUPE, "Recent trends in platforms of embedded systems."

[34] A. Deb, "Hw/sw mechanisms for instruction fusion, issue and commit in modern u-processors," 2012.

TEZ İZİN FORMU / THESIS PERMISSION FORM

PROGRAM / PROGRAM

Sürdürülebilir Çevre ve Enerji Sistemleri / Sustainable Environment and Energy Systems ☐

Siyaset Bilimi ve Uluslararası İlişkiler / Political Science and International Relations ☐

İngilizce Öğretmenliği / English Language Teaching ☐

Elektrik Elektronik Mühendisliği / Electrical and Electronics Engineering ☐

Bilgisayar Mühendisliği / Computer Engineering ☐

Makina Mühendisliği / Mechanical Engineering ☐

YAZARIN / AUTHOR
Soyadı / Surname : .....

Adı / Name : .....

Programı / Program : .....

TEZİN ADI / TITLE OF THE THESIS (**İngilizce** / English) : .....
.....
.....
.....

TEZİN TÜRÜ / DEGREE**:** **Yüksek Lisans** / Master ☐ **Doktora** / PhD ☐

1. **Tezin tamamı dünya çapında erişime açılacaktır. /** Release the entire work immediately for access worldwide. ☐

2. **Tez iki yıl süreyle erişime kapalı olacaktır.** / Secure the entire work for patent and/or proprietary purposes for a period of **two years. *** ☐

3. **Tez altı ay süreyle erişime kapalı olacaktır.** / Secure the entire work for period of **six months**. ***** ☐

Yazarın imzası / Author Signature ..... **Tarih** / Date .....

Tez Danışmanı / Thesis Advisor Full Name: ……..

Tez Danışmanı İmzası / Thesis Advisor Signature: ……..

Eş Danışmanı / Co-Advisor Full Name: ……..

Eş Danışmanı İmzası / Co-Advisor Signature: ……..

Program Koordinatörü / Program Coordinator Full Name: ……

Program Koordinatörü İmzası / Program Coordinator Signature: ……..