上海交通大學

SHANGHAI JIAO TONG UNIVERSITY

学士学位论文

BACHELOR'S THESIS

论文题目： RISC-V 处理器微架构的设计与优化

学生姓名： 史历、刘之远、孙逸秋、时尖、袁意超

学生学号： 517370910032, 517370910240, 517370910020, 517370910255, 517370910233

专　　业： 电子与计算机工程

指导教师： 钱炜慷

学 院 (系)： 密西根学院

上海交通大学
毕业设计（论文）学术诚信声明

本人郑重声明：所呈交的毕业设计（论文），是本人在导师的指导下，独立进行研究工作所取得的成果。除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的作品成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式标明。本人完全意识到本声明的法律结果由本人承担。

作者签名：史历 刘之远 孙遇秋 时尖 袁亮超

日期：2021 年 8 月 2 日

上海交通大学
毕业设计（论文）版权使用授权书

本毕业设计（论文）作者同意学校保留并向国家有关部门或机构送交论文的复印件和电子版，允许论文被查阅和借阅。本人授权上海交通大学可以将本毕业设计（论文）的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本毕业设计（论文）。

保密□，在___年解密后适用本授权书。

本论文属于

不保密 ☑。

（请在以上方框内打“√”）

作者签名：史丙 刘之远 孙遇秋 时尖 袁亮超　　指导教师签名：钱伟懿

日期：2021 年 8 月 2 日　　日期：2021 年 8 月 2 日

RISC-V 处理器微架构的设计与优化

摘要

在过去的几十年中，人们在不断研发更高性能的处理器。但是，由于物理及材料科学方面的限制，摩尔定律将在不久的将来达到瓶颈。这促使着工程师们在计算机体系结构领域进行新的尝试，以此来突破瓶颈。近年来，随着系统级芯片 (SoC) 技术的发展和特定领域应用的需求，在特定领域进行架构设计与优化变得越来越普遍。例如，因为嵌入式系统中的人工智能 (AI) 应用通常需要同时达到高性能、低功耗和低造价这三个要求，人工智能领域的嵌入式系统对现有的芯片体系提出了很大的挑战。在这个项目中，我们设计了一款基于 RISC-V 指令集的处理器。这款处理器支持四路超标量执行，并且具备指令动态规划能力。我们同时对该处理器进行了众多优化。例如，我们添加了近似计算单元，用以提升特定领域的计算性能和能耗比。最终，我们将这款 SoC 与周边设备一同放入模拟软件中进行测试。周边设备包括: 缓存，输入输出设备等。我们的处理器在性能、功耗和成本三方面达到了很好的平衡，并且在特定应用的嵌入式系统领域给用户提供了较好的体验。

关键词：计算机体系结构，系统级芯片，嵌入式系统

RISC-V SOC MICROARCHITECTURE DESIGN AND OPTIMIZATION

ABSTRACT

People have continuously pursued higher performance of processors in the past few decades. However, due to physical and material limitations, Moore's Law is reaching its boundary soon, forcing engineers to make new attempts in the field of computer architecture. Domain-specific architecture design and optimization is increasingly popular recently with emerging technology in systems on-chip (SoC) design as well as the need for domain-specific applications. One of the main challenges is raised from the domain of AI-oriented embedded systems, as AI applications in embedded systems usually require high performance, low power consumption and low cost at the same time. In this project, based on RISC-V instruction set architecture (ISA), we design a processor that supports 4-way superscalar execution and instruction dynamic scheduling. We apply further optimizations, e.g., approximate computing units, to enhance performance and energy efficiency for domain-specific applications. Finally, we set up the SoC and peripheral components including cache and I/O devices in software simulator tools. Our processor keeps a good performance-energy-cost balance and offers users good experience for specific applications in embedded systems.

Key words: Computer architecture, System on-chip, Embedded system

Contents

1 Introduction 1
2 Related Work 3
2.1 MIPS Processor with Classical 5-stage Pipeline 3
2.2 Rocket Core 3
2.3 Berkeley Out-of-Order Machine (BOOM) 3
2.4 Hummingbird E203 3
3 Customer Requirements & Engineering Specifications 4
3.1 Customer Requirements (CR) 4
3.1.1 General Requirements 4
3.1.2 Performance Requirements 6
3.1.3 Embedded System Requirements 7
3.1.4 Benchmark Competitions against CR 8
3.2 Engineering Specifications (ES) 8
3.2.1 Cross Correlation of ES 9
3.2.2 Benchmark Competitions against ES & Set Target for ES 9
3.2.3 Correlation from CR to ES 10
3.3 Quality Function Development (QFD) 11
4 Concept Generation 12
4.1 ISA 12
4.2 Microarchitecture 13
4.3 Verification & SoC Integration 14
4.3.1 Verification 14
4.3.2 SoC Integration 15
5 Concept Selection 15
5.1 ISA 15
5.2 Microarchitecture 17
5.3 Verification & SoC Integration 18
5.3.1 Verification 18
5.3.2 SoC Integration 19
6 Final Design Description & Analysis 20
6.1 Engineering Design Analysis 21
6.1.1 ISA 21
6.1.2 Microarchitecture 21
6.2 Design Description of ISA 21
6.2.1 Customized Instructions 21
6.3 Design Description of Microarchitecture 24
6.3.1 Instruction Fetch & Branch Prediction 24
6.3.2 Fetch Buffer 25

6.3.3 Instruction Decode 25
6.3.4 Register Renaming 26
6.3.5 Dispatch 29
6.3.6 Issue 29
6.3.7 Register File 32
6.3.8 Execution 32
6.3.9 Write Back 34
6.3.10 Commit 34
6.3.11 Cache 35
7 Implementation & Validation 36
7.1 Project Timeline 36
7.2 Implementing Plan 38
7.3 Validation Tool 39
7.3.1 Program layout in memory 39
7.3.2 IO libraries 39
7.3.3 Compilation Toolchains 40
7.3.4 Cycle-Accurate Verification Tool: Verilator 42
7.3.5 Trace-Accurate Verification Tool: Spike 44
7.3.6 Logic Synthesis & Implementation Tool: Xilinx Vivado 45
7.4 Validation of Engineering Specifications 45
7.4.1 Validation Environment 45
7.4.2 Validation Result 47
8 Discussion & Future Work 49
8.1 Load-Store Unit 49
8.1.1 Potential Problem Analysis 49
8.1.2 Current Solution & Future Work 50
8.2 Pipeline Optimization 51
8.2.1 Strengths of Our Pipeline Design 51
8.2.2 Potential Problem Analysis 51
8.2.3 Pipeline Optimization & Future Work 52
9 Conclusion 52
Acknowledgements 53
Appendix A Experiment Result 54
A.1 Verilator and Spike 54
A.2 Xilinx Vivado 54
Appendix B Bios 55
B.1 Li Shi 55
B.2 Zhiyuan Liu 55
B.3 Yiqiu Sun 56
B.4 Jian Shi 56
B.5 Yichao Yuan 57
References 58

Individual Contribution Reports 61

1 Introduction

During the past decade, people have tried to build machines with better computing performance. Frequency scaling was first used to enhance the performance of the system. Since clock frequency is the "heart rate" of a computer, frequency ramping has traditionally been the dominant force in the improvement of performance for commodity processors from the mid-1980s to about the end of 2004 ([1], Latif, 2014). However, increases in frequency also lead to increases in power consumption. It was also because of the power wall that Intel canceled its Tejas and Jayhawk processors in 2004 ([2], Vance, 2004).

Illustration 1–1 illustrates the improvement in clock frequency over time, measured in megahertz (MHz, millions of cycles per second). Clock-frequency improvements have been stalled and limited to about 3-4 gigahertz (GHz) due to power consumption nowadays ([3], Fuller et al., 2011).

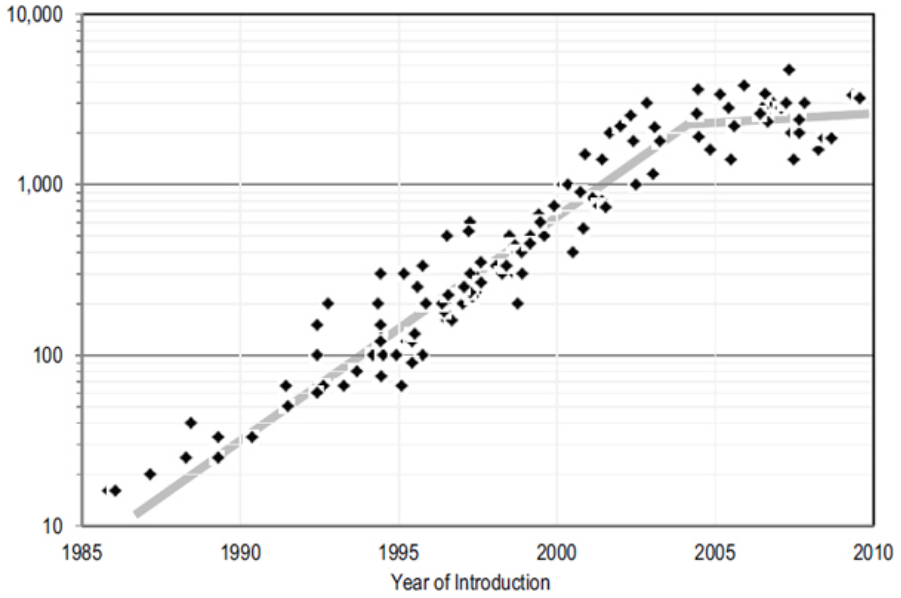


Illustration 1–1 Microprocessor clock frequency (MHz) over time (1985-2010) ([3], Fuller et al., 2011).

To continuously improve the computing performance without depending on Moore's Law, people start to apply parallel computing instead of sequential computing nowadays. Furthermore, products with parallel hardware and chip multi-processors have become a popular trend in the microprocessor industry ([3], Fuller et al., 2011).

In addition to parallel computing, people further exploit the potential of computer architecture by proposing new topics in different areas. Domain-specific architecture design and optimization are one of these hot topics due to the need for artificial intelligence (AI) applications. A groundbreaking example for domain-specific architecture is Google's tensor processing unit (TPU), which was

initially deployed in 2015 and now serves more than 1 billion users. Compared with contemporary CPU and GPU based on traditional technologies, it can run deep neural network (DNN) applications 15 to 30 times faster and has 30 to 80 times better energy efficiency ([4], Jouppi et al., 2018).

The biggest challenge in domain-specific architecture design is raised from the domain of the embedded system. Recently embedded systems start to adapt algorithms that require large computing power. For example, Tesla's automotive car needs to process millions of pixels in a short period and feed them to the AI algorithms ([5], Bouchard, 2019).

Traditional AI computing platforms like CPUs in embedded systems cannot provide adequate computing power. Not only the computation limits the design, but energy efficiency is also vital for some system. A smart camera usually can be supplied limited energy. A camera on the drone, for example, must save energy to improve battery life. However, smart embedded systems do need complicated operations to execute large applications with high power consumption. Finally, embedded systems are limited by cost. Generally, the engine controller under the hood of the car and the chip in the mobile phone needs to be below $3-5 ([3], Fuller et al., 2011). The chips in tennis shoes or greeting cards are even cheaper and the chip area is usually limited. Due to these reasons, some techniques like single instruction & multiple threads (SIMT) may not be appropriate for such a system. Therefore, the need for an embedded system calls for a novel design in computer architecture that can balance performance, energy efficiency, and cost for computation-intensive tasks.

There are some recent achievements in improving computer performance through optimizing computer architecture. In a study related to application-specific integrated circuit (ASIC) design, a high-accuracy approximate adder is introduced which can decrease the power and delay with low relative error([6], Hu et al., 2019: 370-388). Subsequently, a study introduces approximate operators into DNN hardware accelerators to reduce latency and energy consumption ([7], Mrazek et al., 2019). Another study applies approximate dividers and square root operators on their RISC-V design ([8], Li et al., 2017: 1-8). However, all these studies narrow their designs in some specific fields and the approximate adders and approximate operators actually can be used in a wider range of computing components. In this project, we propose an out-of-order processor design based on RISC-V instruction set architecture with approximate computing units to reach a general solution for processor design that can balance performance, energy, and cost.

In this report, a short introduction and related works will be given in chapter 1 and 2. We will discuss our customer requirements and engineering specifications in chapter 3. Then the process of concept generation and selection will be discussed in detail in chapters 4 and 5, followed by our final design of microarchitecture design and optimization in chapter 6. Implementation and validation will be discussed in chapter 7. Finally, our discussion, future work, and conclusion of this project will be

included in chapters 8 and 9.

2 Related Work

In the field of computer architecture, there exist several different processor designs to speed up computing from different perspectives.

2.1 MIPS Processor with Classical 5-stage Pipeline

One popular design is a MIPS processor with a classical 5-stage pipeline. This architecture, proposed by Stanford University, uses MIPS instruction set architecture (ISA) and a five-stage pipeline, which enables the processor to execute more than one instruction in one clock cycle. However, all instructions in the processor are executed in order with a single arithmetic logic unit (ALU) ([9], Hennessy et al., 2020). Therefore, the average Instruction per Cycle (IPC) is less than five, which is not adequately efficient.

2.2 Rocket Core

Another widely-used architecture is Rocket core, which is designed by UC Berkeley. Compared to a classical 5-stage MIPS processor, Rocket core implements a floating-point unit (FPU) and a co-processor, called "ROCC". The specialized FPU speeds up the floating-point computation. Meanwhile, the co-processor enables users to customize their instructions ([10], Asanović et al., 2016).

2.3 Berkeley Out-of-Order Machine (BOOM)

For further improvement, UC Berkeley releases an architecture for out-of-order execution, called "BOOM". The structure implements up to 10 pipeline stages. Furthermore, instructions are executed out-of-order in this processor ([11], Celio et al., 2017). Therefore, it is far more efficient than previous structures.

2.4 Hummingbird E203

Hummingbird E203 is one of the most popular open-source RISC-V SoC among the Chinese community and was designed by Zhenbo Hu's team in China ([12], Hu, 2018). It is designed for em-

bedded systems with extremely low power consumption and circuit area and is suitable for research purposes.

All architectures mentioned above do not have accelerators for machine learning or approximate computing. In comparison, our final product will have approximate computing units in the execution stage to speed up machine learning. We believe that with these accelerators, our architecture will perform better in certain situations.

3 Customer Requirements & Engineering Specifications

3.1 Customer Requirements (CR)

Since our project is experimental and research-oriented, there do not exist specific customers for our design. However, customer requirements are still necessary, as, after the completion of our project, potential embedded system customers may be interested. Meanwhile, mainstream CPU and SoC evaluation indices still apply to our design, e.g., core frequency, performance, energy efficiency, power consumption, etc. Briefly speaking, our target market is artificial intelligence hardware systems, and the consumers include users who want to run machine learning applications in their embedded systems.

We divide our customer requirements into three categories: general (G), performance (P), and embedded system (ES). For the general part, customers may require that the design should be compatible with general RISC-V applications, and attached with clear documentations. Target users of our design may include computer science researchers, embedded system users, or both. The former requires the performance part, e.g., good performance for machine learning applications and the latter requires the embedded system part, e.g., low power consumption and quick response.

3.1.1 General Requirements

The main concern for the part of the general requirements is to set up proper constraints so that we can achieve good compatibility as well as usability. This is determined by the characteristics of our target market and consumers.

The target market of our SoC is a sub-division of the traditional computing hardware market, which requires tight constraints on performance, cost, and energy efficiency. Thus, the general requirement should be a super-set of the current computing market. For example, microprocessors and computing system-based RISC-V ISA has a solid ecosystem and active community. Toolchains like `gcc`

compiler support, Linux kernel support, and verification support are well established ([13], RISC-V International, 2021). The market will welcome a new SoC which is fully compatible with the current RISC-V ecosystem because the cost to adopt our product will be low and they will need less labor and less time, comparing to the product that needs software part from scratch. Also, the competitive computing market requires good usability to win the business. The computing hardware market requires good documentation, as experienced engineers in this high standardized industry can be much more productive when the documentation follows professional protocols. This requirement can also be justified if we consider who is our target consumers. It is hard for startups or small size company to use our product because these companies generally cannot take the risk to adopt a new SoC. Our target consumers will be medium or large size, system-level companies who want to use our SoC to increase their current product's efficiency. These companies have strong low-level research and development capability, and their software or application may be architectural specific. Therefore, they will prefer a new SoC with minimum architectural change and requires little effort to deploy their applications. Compatibility and usability are therefore the main points in the General requirements part.

As our SoC is based on the RISC-V architecture, being compatible with the RISC-V architecture is the most important constraint to consider. RISC-V is a family of ISA, so selecting a proper RISC-V ISA subset is critical. RISC-V ISA includes a base integer ISA, like RISC-V 32I, and optional extensions to the base ISA. We want our SoC to be a stable target for RISC-V applications so that the application source code written in high-level languages like C/C++ can be correctly mapped to our SoC. To run general RISC-V applications, a RISC-V 32G is a reasonable target as suggested by the RISC-V specification ([14], Waterman et al., 2019), which includes:

1. RISC-V 32I base ISA
2. RISC-V M extension, for integer multiplication and division
3. RISC-V A extension, for atomic instructions
4. RISC-V F extension, for single-precision floating-point
5. RISC-V D extension, for double-precision floating-point

RISC-V 32G applications encode their instructions in 32 bits. However, due to the limited memory size of many embedded systems, we hope that the size of applications can be as small as possible. One of the solutions is to introduce the RISC-V C extension for compressed instructions, where instructions are encoded in 16 bits. If time permits, we should add the support for the RISC-V C extension to further optimize for embedded systems.

To enhance the usability of our SoC, clear and clean documentation is needed. Enough documentation should be written for each module, each subsystem, and the overall design. If time permits,

some performance analysis can also be added to the documentation so the consumers can better unleash the potential of our SoC.

Finally, due to the requirement of cost control in most embedded system designs, our SoC should be as inexpensive as possible.

General Requirements:

1. Is compatible with RISC-V applications (weight: 10)
2. Has good documentation for reference (weight: 6)
3. Is inexpensive (weight: 5)

3.1.2 Performance Requirements

While Gordon Moore predicted the growth in semiconductors, he also foresaw its end 50 years later. In recent days, even the company that uses Moore's law as a guideline has to slow its pace to build even smaller transistors. Therefore, our SoC faces a market that requires smart designs to achieve higher performance under the current technology node. The market will first need high performance for normal arithmetic operations. Although nowadays specific AI algorithms rely heavily on matrix operations, the normal arithmetic operations still take a great part in a program to do branching, jumping, indexing, etc. Thus, to achieve an overall high performance, we need to have high-performance normal arithmetic units and designs so that this part will not be a bottleneck. For the normal operations, the performance throughput equals to frequency times the Instructions executed per clock cycle (IPC). In order to achieve high throughput, we need both a high frequency and a high IPC. Designing more pipeline stages may help achieve higher clock frequency, but may result in higher plenty when there is a miss prediction. Dynamic scheduling is a technique to increase the throughput by enabling Out-of-Order execution when instructions have no dependency. This method extracts instruction-level parallelism and it is always combined with other techniques like renaming, speculative execution to maximize the outcome. Also, enough function units are needed so that the pipeline will not stall during the congestion at the execution stage. Our SoC should adopt these traditional techniques to have a high performance on the traditional normal arithmetic operations.

The AI application is a hot topic today. The performance of running AI applications is also demanded by our target market. One of the most significant properties of this application is that it is error-tolerant. During the inference, minor errors will not introduce much difference in the final result. During the training, some errors may even manually be introduced to increase the robustness of the algorithm. This makes approximate computing a potential way to achieve higher frequency by using

circuits with fewer elements.

Another property of AI application is that it is data-driven, which means it is usually memory-bounded. Our SoC needs a high-performance memory subsystem to make sure the pipeline for AI applications will not hungry for data and halt. By using memory hierarchy and introducing cache, we can exploit the temporary locality and spatial locality of memory access to increase the memory subsystem's performance. One of the most important parameters that determine the cache's performance is its size. Trade-offs between the cache size and other parameters like frequency need to be made so that the overall performance can be improved.

Before our design is fixed, we want to do some architecture exploration. Therefore, we want the parameters of some of the critical modules, like the reorder buffer size and cache size is configurable so that we can change them to evaluate performance.

Performance Requirements:

1. Runs fast for normal arithmetic operations (weight: 8)
2. Runs fast for machine learning applications (weight: 9)
3. Runs fast for memory-bound applications (weight: 7)
4. Can be configured with different parameters (weight: 2)

3.1.3 Embedded System Requirements

Although the warehouse-scale computing center still requires plenty of computation nowadays, the market has a trend to offload part of the computation to the edge because of security issues, latency issues, etc. Our SoC will be more competitive at the edge as the concept of AI on things has attracted lots of attention. Therefore, we require our SoC to satisfy some requirements for embedded systems.

The embedded system is often energy-bounded as the power source is usually a battery. High power consumption is not allowed because these scenarios often require long battery life. It can be evaluated by the number of operations processed within unit energy, so our SoC should have a limited number at this part. Some embedded systems also involve latency constraints. For AR/VR applications, this requirement is necessary as longer latency will cause discomfort. We hope our SoC can provide a low average response time to a request for service. The market also requires embedded SoC to have low cost, as the embedded system application is normally cost-sensitive. This can be evaluated by the usage of resources on the FPGA, like look-up tables (LUTs), block RAMs (BRAMs), and digital signal processors (DSPs). The embedded system also has to needs to connect to various types of peripherals, so have more I/O controllers can make our product more competitive on the market.

Embedded Requirements:

1. Saves power (weight: 6)
2. Responds quickly (weight: 4)
3. Low cost (weight: 3)
4. Has good support for multiple I/O devices (weight: 3)

3.1.4 Benchmark Competitions against CR

In this part, we quickly review some existing open-source solutions against customer requirements. This step gives us a better view of customer requirements as well as offers us some clues on how to design our own products based on existing solutions.

The evaluation is based on a 5-point scale, where 5 means "satisfies perfectly" and 1 means "doesn't satisfy at all". We select four open-source solutions to evaluate: MIPS processor with classical 5-stage pipeline, Rocket Core, Berkeley Out-of-Order Machine (BOOM), and Hummingbird E203. For instance, in terms of "is compatible to general RISC-V applications", we find that Rocket Core and BOOM have the best support, so we give them 5 points. Hummingbird E203 has limited support for RISC-V applications, so we give it 3 points. The MIPS processor with a classical 5-stage pipeline is not compatible with RISC-V applications, so we give it 1 point.

The detailed benchmark competitions against CR are shown in the HOQ chart (Illustration 3–1).

3.2 Engineering Specifications (ES)

Based on the previous customer requirements, we come up with the engineering specifications of our project, which represent the quantifiable measures to guide us to design the project properly that can meet customers' requirements. Table 3–1 shows our overall engineering specifications in our project design.

This should be a key step to "translate" customers' words into professional terms and specifications. For instance, when customers require that their RISC-V applications can run on our SoC, what we actually need is that our SoC should support RV32G ISA. In this part, we will explain how we check the cross-correlation of ES, examine the benchmark competitions against ES, set our own target for ES, and most importantly, correlate CR to ES.

Table 3–1 Engineering specifications.

Engineering Specification	Unit	Target Value
Support RV32G instruction set architecture (ISA)	-	Yes
Core frequency on FPGA test platform	MHz	100
Number of pipeline stages	-	9
Instructions executed per clock cycle (IPC)	-	0.5
Support instruction dynamic scheduling	-	Yes
Typical total cache size	KB	32
Number of function units	-	6
Average response time to a request for service	ms	10
Usage of look-up tables (LUT) on FPGA	k	120
Usage of block RAM (BRAM) on FPGA	-	50
Usage of digital signal processor (DSP) on FPGA	-	30
Power consumption on target FPGA test platform	W	5
Operations processed within unit energy.	MOp/J	25
Number of flexibly-configured modules	-	10
Number of I/O device types	-	3
User guide and programmers manual	-	Yes

3.2.1 Cross Correlation of ES

In this step, we check the cross-correlation of ES and reveal some internal connections between each specification we just made in the previous step. For example, when we increase the number of function units, the usage of hardware resources also increases. When we want to support instruction dynamic scheduling to support out-of-order instruction execution, we can greatly improve the number of instructions executed per clock cycle, which is a significant mark of performance improvement.

This step also gives us clear guidance on design trade-offs. The most evident contradiction is that the SoC we want to design can never achieve low cost, high performance, low power consumption at the same time, which exactly corresponds to our three different parts of customer requirements. For example, if we want to improve the performance, possible solutions include increasing the frequency, increasing the number of function units, etc., none of which is good for saving power and reducing cost.

The detailed cross-correlation of ES is given in the HOQ chart (Illustration 3–1).

3.2.2 Benchmark Competitions against ES & Set Target for ES

It is important to perform some benchmark on some existing solutions in terms of our engineering specifications, which enables us to understand better about drawbacks of existing solutions as well

as our design targets in the future.

This process sometimes requires "reverse engineering" for commercial products, but should be much easier for open-source products. In our survey, except that Hummingbird E203 is partially open-source, all the other three processors are completely open-source, and we should be able to access the data conveniently. However, we need to pay attention that some data is still difficult to access, e.g., the average response time to a request for service. There are some reasons for this kind of lack of data. First, even though the processor design is open-source, the synthesis tool or the test suites are not open-source, leading to the difficulty to reproduce the test results. Second, even though the test processes are completely public, the results may depend on the specific hardware testing platform, which may be expensive or inaccessible. Third, some of the designs can be flexibly configured, and there is no standard or typical settings, as a result of which the test results under different settings can not be referred to or compared directly.

In this step, we try our best to explore the data of these existing solutions. Some data can be easily accessed, e.g., the number of pipeline stages or whether instruction dynamic scheduling is supported, while some other data cannot. ([15], Dörflinger et al., 2021) provides us with many useful indicators to evaluate a design and meanwhile gives us lots of data of both Rocket Core and BOOM, including usage of hardware resources, etc.

Based on the data of existing solutions, we come up with our target values for ES. For example, as our aim is to provide a good experience for machine learning application users, performance is crucial to some specific applications, and hence we decide to support instruction dynamic scheduling and set a relatively high value of core frequency of our design. During the process of deciding our own targets, we mainly refer to the data of BOOM, but also look at the data of the other three solutions.

The detailed benchmark competitions against ES and our targets for ES are shown in the HOQ chart (Illustration 3–1).

3.2.3 Correlation from CR to ES

In this step, we not only make our engineering specifications from scratch but also check whether all the customer requirements can be covered in our specifications. While the following paragraphs give a short explanation of how we transform CR into ES.

1. **General part**: As customers require that our SoC should be compatible with general RISC-V applications, we require that our SoC should support RV32G ISA. As customers need good and clear documentation, we require that we need to deliver the SoC user guide and programmer

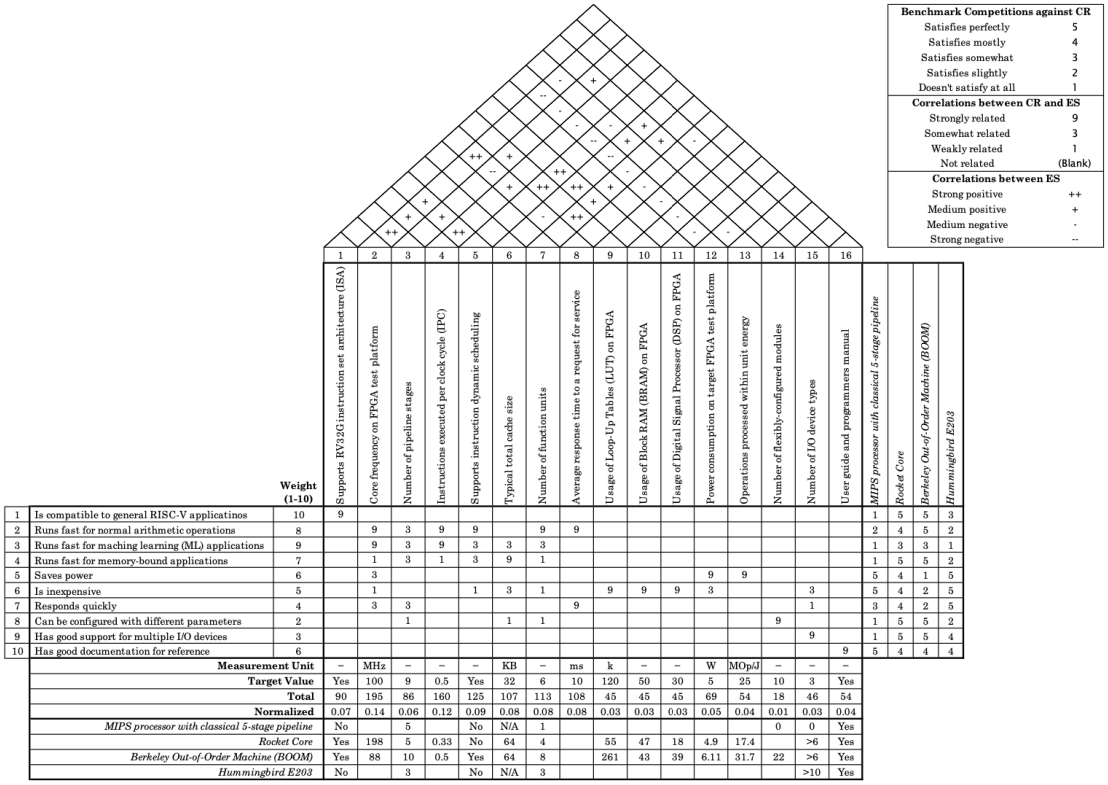
manual. To control the cost, we need to control our circuit size by limiting our usage of hardware resources, e.g., usage of look-up tables (LUT), digital signal processors (DSP), and block RAMs (BRAM).

2. **Performance part**: We want to optimize our SoC for three different customer scenarios: normal arithmetic operations, machine learning applications, and memory-bound applications. They share some common characteristics, e.g., they all require that our SoC should support instruction dynamic scheduling and as high core frequency as possible. Meanwhile, they differ in some specific aspects. For instance, as normal arithmetic applications and machine learning applications are mostly compute-bound, they will benefit more from the improvement in core frequency, instructions executed per clock cycle, and the number of function units, while memory-bound applications will benefit more from the improvement in total cache size, as the cache is used to accelerate the access to memory.
3. **Embedded system part**: Customers require that our SoC works well in embedded systems. To save power, we need to monitor and control the power consumption and energy efficiency of our SoC. To achieve a quick response, we need to measure the average response time to a request for service. To configure our SoC with different parameters easily, we need to increase the number of flexibly configured modules in our digital design. To have good support for multiple I/O devices, we need to support multiple kinds of I/O types and test on various I/O devices.

3.3 Quality Function Development (QFD)

We develop our House of Quality (HOQ) chart with the QFD method. The HOQ chart is shown in Illustration 3–1.

The main steps of the QFD method have been expanded in the previous sections. Finally, we are able to examine our design priority according to the “Total” and “Normalized” fields. The top 5 engineering specifications we need to consider are core frequency, instructions executed per clock cycle, support for instruction dynamic scheduling, number of function units, and average response time. The result generally matches our expectation that performance is still among the most important factors in embedded system SoC design. Meanwhile, some specifications like the number of



Benchmark Competitions against CR	
Satisfies perfectly	5
Satisfies mostly	4
Satisfies somewhat	3
Satisfies slightly	2
Doesn't satisfy at all	1
Correlations between CR and ES	
Strongly related	9
Somewhat related	3
Weakly related	1
Not related	(Blank)
Correlations between ES	
Strong positive	++
Medium positive	+
Medium negative	-
Strong negative	--

		Weight (1-10)	1 Supports RV32G instruction set architecture (ISA)	2 Core frequency on FPGA test platform	3 Number of pipeline stages	4 Instructions executed per clock cycle (IPC)	5 Supports instruction dynamic scheduling	6 Typical total cache size	7 Number of function units	8 Average response time to a request for service	9 Usage of Loop-Up Tables (LUT) on FPGA	10 Usage of Block RAM (BRAM) on FPGA	11 Usage of Digital Signal Processor (DSP) on FPGA	12 Power consumption on target FPGA test platform	13 Operations processed within unit energy	14 Number of flexibly-configured modules	15 Number of I/O device types	16 User guide and programmers manual	*MIPS processor with classical 5-stage pipeline*	*Rocket Core*	*Berkeley Out-of-Order Machine (BOOM)*	*Hummingbird E203*
1	Is compatible to general RISC-V applicatinos	10	9																1	5	5	3
2	Runs fast for normal arithmetic operations	8		9	3	9	9		9	9									2	4	5	2
3	Runs fast for maching learning (ML) applications	9		9	3	9	3	3											1	3	3	1
4	Runs fast for memory-bound applications	7		1	3	1	3	9	1										1	5	5	2
5	Saves power	6		3										9	9				5	4	1	5
6	Is inexpensive	5		1			1	3	1		9	9	9	3			3		5	4	2	5
7	Responds quickly	4		3	3					9							1		3	4	2	5
8	Can be configured with different parameters	2			1			1	1							9			1	5	5	2
9	Has good support for multiple I/O devices	3															9		1	5	5	4
10	Has good documentation for reference	6																9	5	4	4	4
	Measurement Unit		–	MHz	–	–	–	KB	–	ms	k	–	–	W	MOp/J	–	–	–				
	Target Value		Yes	100	9	0.5	Yes	32	6	10	120	50	30	5	25	10	3	Yes				
	Total		90	195	86	160	125	107	113	108	45	45	45	69	54	18	46	54				
	Normalized		0.07	0.14	0.06	0.12	0.09	0.08	0.08	0.08	0.03	0.03	0.03	0.05	0.04	0.01	0.03	0.04				
	MIPS processor with classical 5-stage pipeline		No		5		No	N/A	1							0	0	Yes				
	Rocket Core		Yes	198	5	0.33	No	64	4		55	47	18	4.9	17.4		>6	Yes				
	Berkeley Out-of-Order Machine (BOOM)		Yes	88	10	0.5	Yes	64	8		261	43	39	6.11	31.7	22	>6	Yes				
	Hummingbird E203		No		3		No	N/A	3								>10	Yes				

Illustration 3–1 House of Quality (HOQ) chart.

flexibly configured modules are relatively not as important as previous ones.

4 Concept Generation

4.1 ISA

An instruction set architecture (ISA) describes the capabilities of a processor. The ISA specifies how instructions should be formatted so that the control unit can correctly interpret them, and the meaning of those instructions in terms of their semantics.

Besides, an ISA also defines the supported data types, the registers, the hardware support for managing main memory, fundamental features, and the input/output model of a processor. As such, an ISA represents an interface between programmer and an abstract processor rather than an exact specification of how the processor should be implemented ([16], Page, 2009).

Therefore, to design our personal processor, we must choose an ISA that satisfies our design requirement. A good ISA can help us achieve our design specifications easily.

An ISA may be classified in a number of different ways. A common classification is by architectural complexity. A complex instruction set computer (CISC) contains many specialized instructions,

some of which may only be rarely used in practical programs. A reduced instruction set computer (RISC) simplifies the processor by efficiently implementing only the instructions that are frequently used in programs, while the less common operations are implemented as subroutines, having their resulting additional processor execution time offset by infrequent use ([17], Chen et al., 2015).

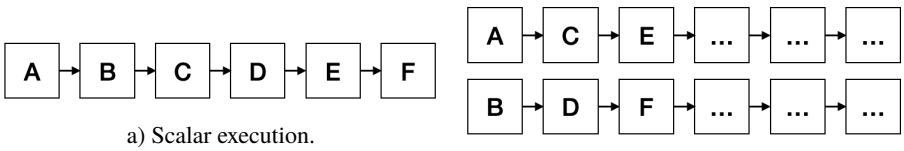
We will make our decision among all these ISAs.

4.2 Microarchitecture

Comparing to ISA, which describes the overall architecture of a computer system that is usually visible to programmers, microarchitecture usually describes the way that how a specific processor design implements the ISA. Namely, to some extent the concepts of ISA and microarchitecture are orthogonal, i.e., one ISA can be implemented by many processors, while one microarchitecture can support different ISAs after necessary modifications. Unlike ISA, usually microarchitecture is invisible to programmers. For instance, in our out-of-order microarchitecture design, although the instructions are dynamically scheduled and executed, from the perspective of the programmer, the instructions seem to be executed sequentially and follow the programmers' intention.

Therefore, a good microarchitecture design is crucial to a successful processor product. Even though different processors share the same ISA, they may differ in microarchitecture designed for different kinds of markets, e.g., high-performance computing or embedded systems.

In terms of the number of instructions that can be executed at the same time, we have scalar design and superscalar design. Fig. 4–1 demonstrates the difference between the two kinds of design. While the scalar design is able to execute one instruction at a time, the superscalar design supports multiple channels to execute the instructions. Theoretically, for example, a 2-way superscalar design can double the performance. However, in reality, due to many external factors, e.g., data and control dependency, the speed-up ratio should be between 1 and 2.



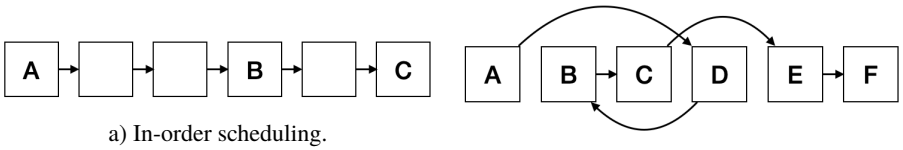
a) Scalar execution.

b) 2-way superscalar execution.

Illustration 4–1 Scalar vs. superscalar processor design.

In terms of instruction scheduling, we have in-order scheduling and out-of-order scheduling, or so-called instruction dynamic scheduling. Fig. 4–2 shows the differences between two ways of instruction scheduling mechanisms. For example, instruction A requires 3 clock cycles to complete, but

instruction B depends on the result of A. In the in-order design, B must wait until A is completed, but in the out-of-order design, we can execute instruction D (that is independent of A) prior to B, so that the performance is further improved. Most of the modern processors for high-performance computing are based on out-of-order scheduling design.



a) In-order scheduling.

b) Out-of-order scheduling.

Illustration 4–2 In-order vs. out-of-order instruction scheduling mechanism.

4.3 Verification & SoC Integration

4.3.1 Verification

The aim of verification is to make sure that the design meets the system requirements and specifications. Normal approaches to verify design include the following three ways ([18], Stroud et al., 2009):

1. **Logic simulation**. The design is verified by performing a cycle-accurate simulation. The detailed timing and functionality information can be check to see correctness.
2. **Functional verification**. The design is checked against functional models, which describe the correct behavior of the design. A behavior level simulation is performed to check the correctness of the design while timing details are ignored.
3. **Formal verification**. This method includes property checking and equivalence checking.

We emphasize performing functional verification as well as logic simulation. We cannot perform formal verification because normally this requires commercial software support. Due to our budget constraints, we do not use this to check our model. For the logic simulation part, the verification process will be based on the logs and waveform, i.e., we check the logs and waveform to check our design's timing and behaviors. Although by logic simulation we can know whether our design is identical to our expectation, they may not necessarily meet the RISC-V specification. We will use another verified RISC-V simulator as a comparison to check the architectural correctness of our processor.

4.3.2 SoC Integration

A single CPU can do few things as it cannot perform IO. It can only run hardwired programs from specific locations. Although we are using many tools to do the evaluation, this still makes the whole development process inconvenient. We perform an SoC Integration aiming at providing our SoC IO capabilities, making it possible to dynamically loading the program and output its computation result to the host machine.

The SoC will integrate at least some memory devices. The whole simulation system will provide the interface for the simulation host machine to communicate and verify the design under test.

5 Concept Selection

5.1 ISA

There exist several popular instruction set architectures (ISAs), such as x86, MIPS, ARMv7, and RISC-V. As we mentioned before, they can be classified by their architectural complexity, e.g., memory addressing modes and instruction length. Since complex instruction set computer (CISC) ISAs are usually not open-source and too complicated for our project, we choose among reduced instruction set computer (RISC) ISAs. Following are three widely-used RISC ISAs.

1. **MIPS:** MIPS is a RISC ISA developed by MIPS Computer Systems. It is widely used in industry and also computer architecture courses for university students. Most of the group members are familiar with MIPS Instructions and have development experience with it.

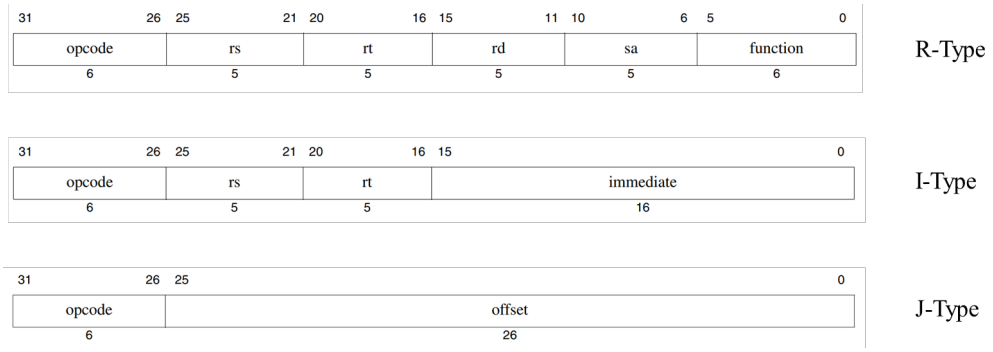


Illustration 5–1 MIPS instruction format([19], MIPS Inc., 2014).

However, with years of development, the design space of MIPS has been exploited, which will be an obstacle to further research. For example, originally branch delay slots are designed to reduce the cost of the branch misprediction. However, it introduces unnecessary design complexity and does not always result in performance improvement. Illustration 5–1 shows the general MIPS

instruction format.

2. **ARMv7:** ARM is a family of RISC ISA developed by Arm Ltd. Among them, ARMv7 is a classic structure applied in many areas. Due to its low cost, low power consumption, and low heat generation rate, ARMv7 is an ideal choice for light, portable, battery-powered devices, such as smartphones, Internet of things (IoT), and other kinds of embedded systems.

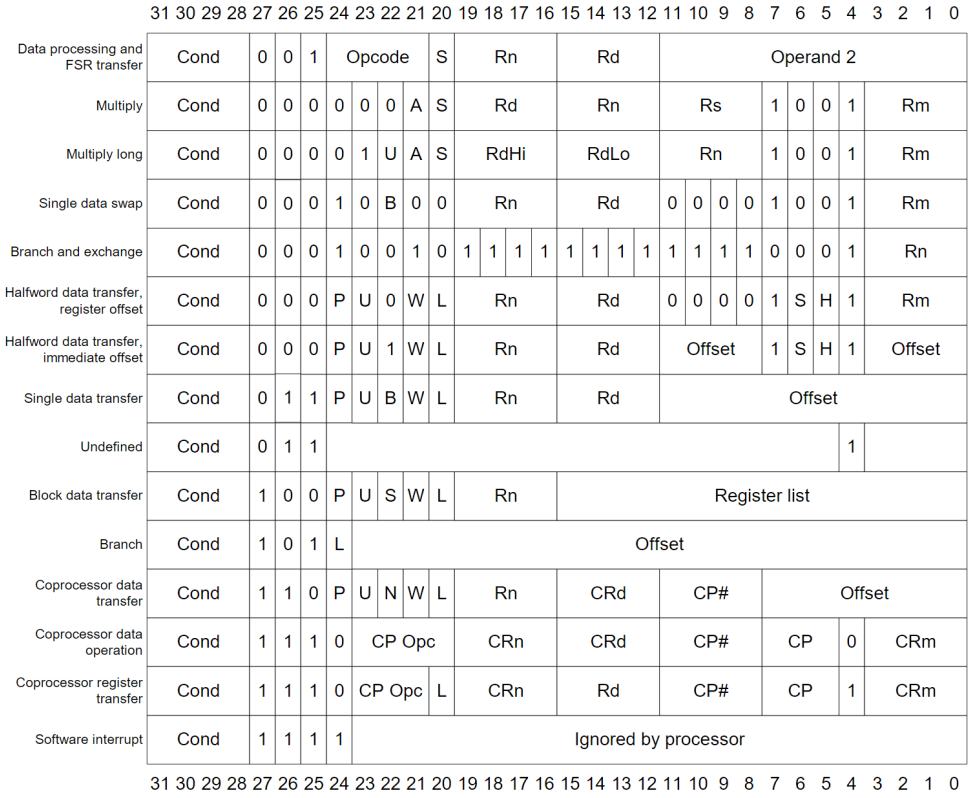


Illustration 5–2 ARM instruction format([20], ARM Limited., 2021).

However, ARM does not allow users to custom their instructions without permission, i.e., we don't have design space to support our own customized instructions. Furthermore, to support more features, there are many advanced instructions and techniques in the ISA specifications, which means that it is difficult for us to implement in a short period. Illustration 5–2 shows the instruction format for ARMv7 ISA.

3. **RISC-V:** RISC-V is an open standard RISC ISA. Unlike most other ISA designs, the RISC-V ISA is provided under an open-source license, i.e, it is completely free to use. Besides, RISC-V has been an emerging ISA during the past decade, which was initially introduced in 2010. It is simple but powerful for embedded system design. Therefore, it becomes our ideal ISA to implement in our processor design. Illustration 5–3 shows the instruction format for RISC-V ISA.

   However, compared to ARMv7, RISC-V supports fewer instructions, resulting in limited functions.

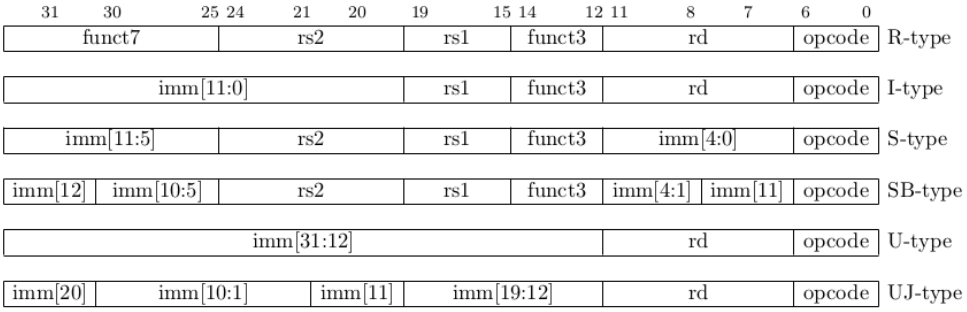


Illustration 5–3 RISC-V instruction format([14], Waterman et al., 2019).

5.2 Microarchitecture

As mentioned in the previous section, microarchitecture determines the performance, power consumption, and also cost of a processor and is crucial to the success of a processor. In this section, we use the decision matrix method to select our microarchitecture design based on customer requirements and engineering specifications. Illustration 5–4 shows the decision matrix of microarchitecture design.

Design Criterion	Weight Factor	Unit	Instruction Parallelism				Instruction Scheduling			
			Scalar		Superscalar		In-order		Out-of-order	
			Score	Rating	Score	Rating	Score	Rating	Score	Rating
Performance	0.45	%	5	2.25	8	3.6	3	1.35	10	4.5
Hardware Complexity	0.20	%	9	1.8	6	1.2	9	1.8	3	0.6
Cost	0.15	%	8	1.2	5	0.75	9	1.35	5	0.75
Flexibility	0.20	%	6	1.2	9	1.8	5	1	9	1.8
Total				6.45		**7.35**		5.5		**7.65**

Illustration 5–4 Decision matrix of microarchitecture design.

In the decision matrix, we systematically evaluate our selection on two concepts, i.e., instruction parallelism and instruction scheduling, based on multiple design criteria, including performance, hardware complexity, cost, and flexibility. Based on customer requirements and engineering specifications, we give each criterion a weight factor. As our processor is mainly used for accelerate computing-intensive tasks, performance is the most important evaluation standard, followed by hardware complexity, flexibility, and cost. Then, we assess each concept based on a 10-point scale, multiplying by the weight factor and get the rating for each criterion. Finally, we sum the ratings and get the total score of a certain concept design.

For example, in terms of instruction scheduling, in-order design results in good hardware complexity and low cost, but the stalls at different kinds of instruction dependency severely restrict the performance of an in-order processor. On the contrary, the out-of-order design may be complicated to implement in hardware, but results in great performance improvement.

Finally, according to the decision matrix, we choose to implement superscalar and out-of-order features in our processor design.

5.3 Verification & SoC Integration

5.3.1 Verification

(1) Logic Simulation

Logic simulation is the process to verify the correctness of our processor design. We need a logic simulator that is free to use, so we have the following options:

1. **Vivado Webpack Xsim**. Vivado Webpack Xsim is the simulator used in Vivado integrated development environment (IDE). It supports most SystemVerilog features for design and verification. It provides users a good experience if they use Vivado IDE to develop and debug, but it also means that users have less flexibility.
2. **Icarus Verilog**. It is an open-source Verilog / SystemVerilog logic simulation tool, which converts the Verilog / SystemVerilog source code to an intermediate form and then performs behavioral simulation. It supports both SystemVerilog design language and verification language, but the performance for simulation is relatively low.
3. **Verilator**. It is another open-source Verilog / SystemVerilog simulator, which converts Verilog / SystemVerilog models into C/C++ models, and uses C++ as the verification language. Although it does not support the verification language of SystemVerilog, the performance for simulation is satisfying.

There are other commercial software tools for this task. However, all of them require an expensive commercial license.

Our final decision is to use both Vivado Webpack Xsim and Verilator. We use Vivado Webpack as it is convenient to debug considerably small designs in the Vivado IDE. We choose Verilator to simulate the integrated SoC and use it to build our simulation environment. C++ is a much more powerful language than SystemVerilog, and it brings productivity when we build the simulation

system. We do not choose Icarus Verilog due to both its poor performance and its difficulty to integrate with the other parts in the simulation system.

(2) Functional Verification

The key part of the functional verification is to choose a proper simulator / emulator as a comparison model. Followings are some popular choices:

1. **Gem5**. Gem5 is an open-source computer architecture simulator used in both academia and industry. It is a very complete but complicated simulator, which provides support for multiple ISAs, including ARM, MIPS, X86, etc. It also provides multiple models for microarchitecture and memory system simulation.
2. **Qemu**. Qemu is a popular emulator, which dynamically translates the target binary to the host machine code. Therefore, the user can execute RISC-V programs on an X86 or ARM machine. This emulator can perform not only user-level simulation but also system-level simulation.
3. **Spike**. Spike is a lightweight simulator, which implements the RISC-V ISA specifications. It is often considered a golden model for RISC-V ISA.

In this project, we focus on the RISC-V ISA. Also, we use the simulator only to verify the functional correctness of our design, so we do not need a complicated simulator like gem5. Qemu is an emulator so it cannot provide adequate information about program execution trace, which is a critical part to compare to verify our design. Therefore, our final decision is to use Spike simulator. Spike is also easier to modify and integrate into our simulation system.

5.3.2 SoC Integration

There exists a vast design space for SoC integration, mainly depending on the customer requirements. Our SoC integration and simulation system should be able to complete the following tasks conveniently and efficiently:

1. The process of loading and executing the program.
2. The process of reading inputs and saving outputs.
3. The process of printing messages.

The SoC integration scheme and the whole simulation system integration scheme are thus referred to Spike simulator. Although Spike is an architectural level simulator, its internal structure includes an SoC as well as a well-designed, lightweight simulation environment.

Illustration 5–5 is a high-level overview of the simulation environment configuration. This illustration can serve as a description of the system for both our Verilator-based simulation system as well as the Spike simulation model, although the internal details are totally different. The SoC integrates the CPU core with peripheral devices like RAM and ROM through a system-level bus. The simulation environment lies on a host machine, which communicates with the SoC through a pure-software frontend server. The frontend server can perform tasks like loading programs into the memory as well as transmitting some requests from the CPU core to the host machine. The program running on the CPU can communicate with the frontend server at the system level through global variables in the program. The frontend server will link to those global variables when it loads the program into the RAM. By writing values to and reading values from the global variables, the program can perform tasks like opening a file, writing to a file, etc.

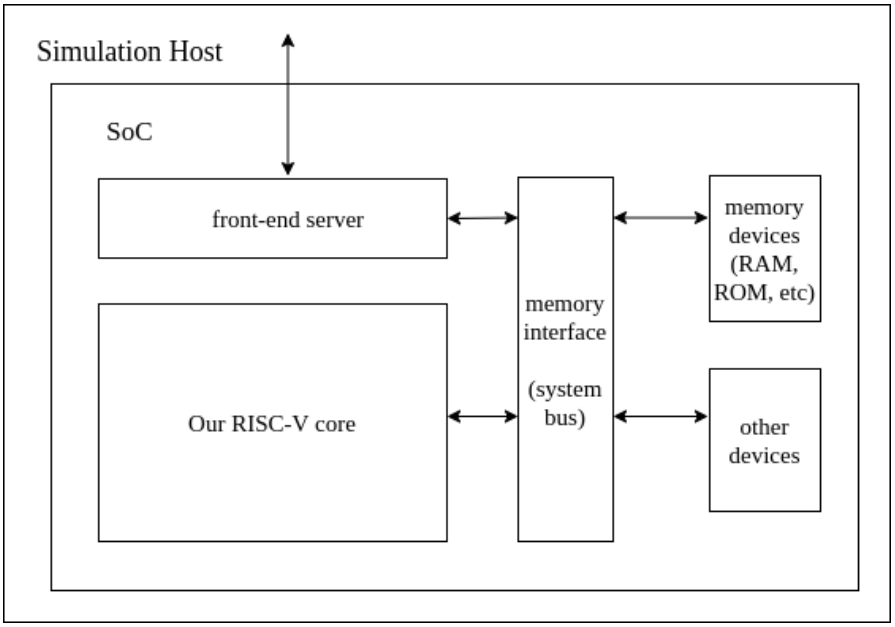


Illustration 5–5 SoC and simulation system integration.

6 Final Design Description & Analysis

In this chapter, we will review our selected concepts in detail, including engineering design analysis, design description, and analysis of ISA and microarchitecture for each pipeline stage.

6.1 Engineering Design Analysis

6.1.1 ISA

As we mentioned in the previous chapter, we select our instruction set architecture (ISA) among three choices: MIPS, ARMv7, and RISC-V. Their advantages and disadvantages are listed in section 5.1.

After discussion, we decide to choose RISC-V ISA as our design standard.

Meanwhile, since our final product implements approximate computing units to accelerate machine learning and neural network computation, customized ISA is necessary for us. RISC-V supports customized instructions and offers abundant design space compared to the other two choices.

Besides, we need to balance our workload and supported features. RISC-V supports many predefined instructions, with acceptable complexity. After evaluation, we can complete our processor design and implement RISC-V ISA in a short period.

6.1.2 Microarchitecture

We have used the decision matrix to determine the final design of our processor microarchitecture design. Specifically, we determine to implement superscalar and out-of-order features in our design, matching most of the customer requirements. For example, in the aspect of performance, our superscalar out-of-order design provides good performance, especially in those computing-intensive tasks. It matches the engineering specifications we set earlier, including supporting instruction dynamic scheduling, etc., and provides a good performance-power-cost balance.

6.2 Design Description of ISA

Based on RISC-V basic ISA, we further add some customized instructions to perform approximate floating-point computations.

6.2.1 Customized Instructions

For floating-point execution units, especially in our design, we support both accurate computation and approximate computation simultaneously. The advantage of approximate computation over ac-

curate floating-point computation is that it is fast and consumes less power. For embedded systems that require high performance, low power consumption, and relatively low computation accuracy, approximate computation units are preferred. To support using our approximate floating-point computing units in ordinary C programs, we have added customized instructions to RISC-V F standard extension and D standard extension. These two standard extensions of RISC-V are used to support operations for single-precision floating-point and double-precision floating-point operations respectively.

(1) Floating-Point Register State

To support the RISC-V F standard extension instruction set and D extension instruction set, 32 floating-point registers are added, shown in Illustration 6–1, named `f0 -f31`, each `FLEN` bits wide with a floating-point control and status register `fcsr` handling the operating mode and exception status. For RISC-V standard F extension only, `FLEN = 32`. If both F extension and D extension are supported, `FLEN = 64` such that floating-point registers can hold either 32-bit or 64-bit floating-point values ([14], Waterman et al., 2019).

(2) Floating-Point Customized Instructions

Similar to the standard floating-point arithmetic instructions operating on one or two source operands, our customized approximate computation instructions are encoded in the R-type format with the OP-FP major opcode ([14], Waterman et al., 2019). `FMULA.S` or `FMULA.D` performs the approximate single-precision or double-precision floating-point multiplication of `rs1` times `rs2` respectively and writes the result back to `rd`. `FDIVA.S` or `FDIVA.D` performs the approximate single-precision or double-precision floating-point division of `rs1` by `rs2` and writes the result back to `rd`.

The encoding of the 2-bit floating-point format field `fmt` is shown in Illustration 6–2. For all instructions in the F extension, `fmt` is set to `S(00)`, and for all instructions in the D extension, `fmt` is set to `D(01)` ([14], Waterman et al., 2019).

The rounding mode of floating-point operations can be selected through `rm` field. The encoding for `rm` field is shown in Illustration 6–3 ([14], Waterman et al., 2019).

Finally, our customized approximate computation instructions are listed in Illustration 6–4. The formats should be `fmula.s/fdiva.s/fmula.d/fdiva.d rd rs1 rs2`.

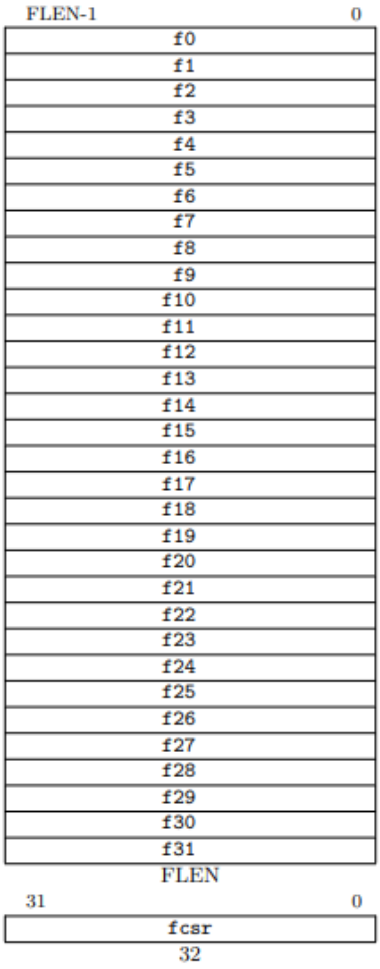


Illustration 6–1 RISC-V F register state ([14], Waterman et al., 2019).

fmt field	Mnemonic	Meaning
00	S	32-bit single-precision
01	D	64-bit double-precision
10	H	16-bit half-precision
11	Q	128-bit quad-precision

Illustration 6–2 Format field encoding ([14], Waterman et al., 2019).

Rounding Mode	Mnemonic	Meaning
000	RNE	Round to Nearest, ties to Even
001	RTZ	Round towards Zero
010	RDN	Round Down (towards $-\infty$)
011	RUP	Round Up (towards $+\infty$)
100	RMM	Round to Nearest, ties to Max Magnitude
101		*Invalid. Reserved for future use.*
110		*Invalid. Reserved for future use.*
111	DYN	In instruction's *rm* field, selects dynamic rounding mode; In Rounding Mode register, *Invalid.*

Illustration 6–3 Rounding mode encoding ([14], Waterman et al., 2019).

	31-27	26-25	24-20	19-15	14-12	11-7	6-2	1-0
fmula.s	00110	00	rs2	rs1	rm	rd	10100	11
fdiva.s	00111	00	rs2	rs1	rm	rd	10100	11
fmula.d	00110	01	rs2	rs1	rm	rd	10100	11
fdiva.d	00111	01	rs2	rs1	rm	rd	10100	11

Illustration 6–4 Customized approximated computation instructions.

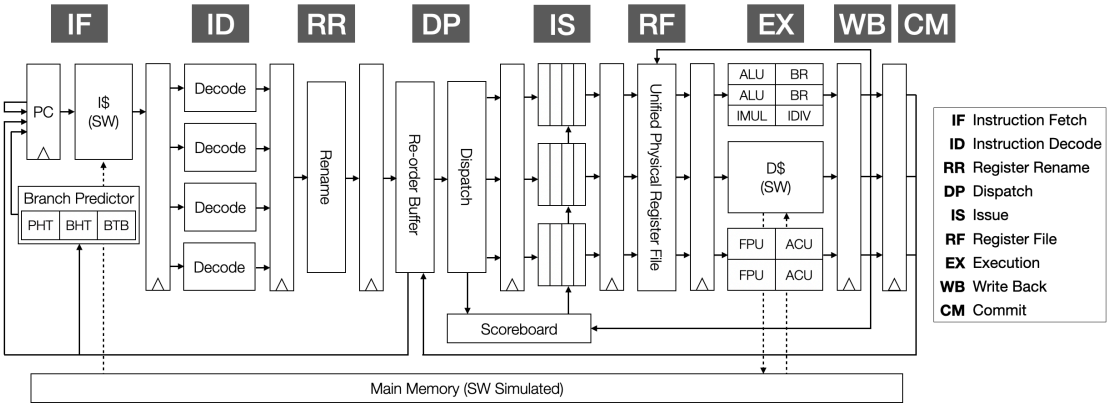


Illustration 6–5 Simplified pipeline diagram.

6.3 Design Description of Microarchitecture

In this part, we will explain the design details of each microarchitecture pipeline stage. A simplified pipeline design diagram is shown in Illustration 6–5. Our processor supports 4-way superscalar execution and instruction dynamic scheduling, dividing into two parts: frontend and backend. In the frontend, there are 4 stages: instruction fetch (IF), instruction decode (ID), register renaming (RR), dispatch (DP). In the backend, there are 5 stages: issue (IS), register file (RF), execution (EX), write back (WB), and commit (CM).

6.3.1 Instruction Fetch & Branch Prediction

The instruction fetch stage is the first stage of the whole pipeline shown in Illustration 6–5. It reads instructions from memory and sends them to the subsequent stages. The branch predictor is also implemented in this stage. As is shown in Illustration 6–6, the branch predictor consists of both direction predictor and branch target buffer (BTB). The direction predictor implements a 2-bit saturation counter to represent the possibility of the branch is taken: 00 - strongly not taken, 01 - weakly not taken, 10 - weakly taken, 11 - strongly taken. We predict the next program counter (PC) address based on the prediction feedback from the re-order buffer (ROB) and the prediction output of the branch predictor.

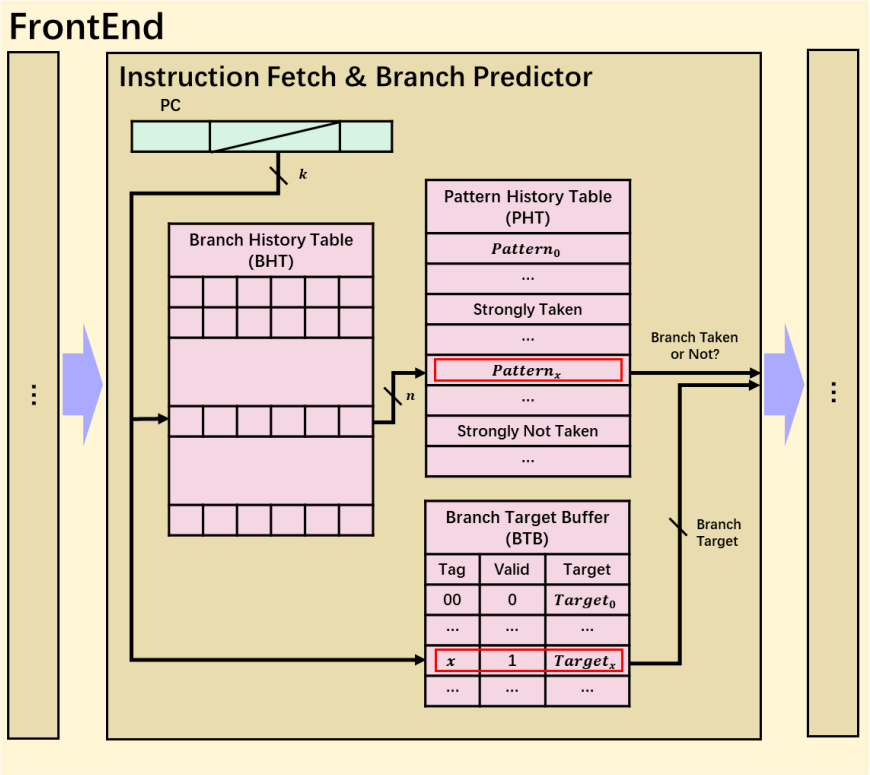


Illustration 6–6 Branch predictor diagram.

6.3.2 Fetch Buffer

The fetch buffer serves as a buffer to store the fetched instructions if they can not be dispatched at once due to the congestion of later stages. The buffer is implemented as a first-in-first-out (FIFO) queue to ensure an in-order dispatch of instructions. It is not used in the final design, but it is useful if we want to further support RISC-V C extension, as the output of this stage is always 4 instructions but the input from the previous stage may be at most 8 instructions.

6.3.3 Instruction Decode

In this part, we decode the input instructions into micro-operations (`uops`). We implement four instruction decoders to decode four instructions simultaneously, shown in Illustration 6–7.

RISC-V instructions can be classified by their formats. Illustration 5–3 shows all instruction types. The decoders first sort input instructions according to the format table. Then, the decoders analyze information in each part of the instruction and translate them into micro-operations that can be processed by the following stages.

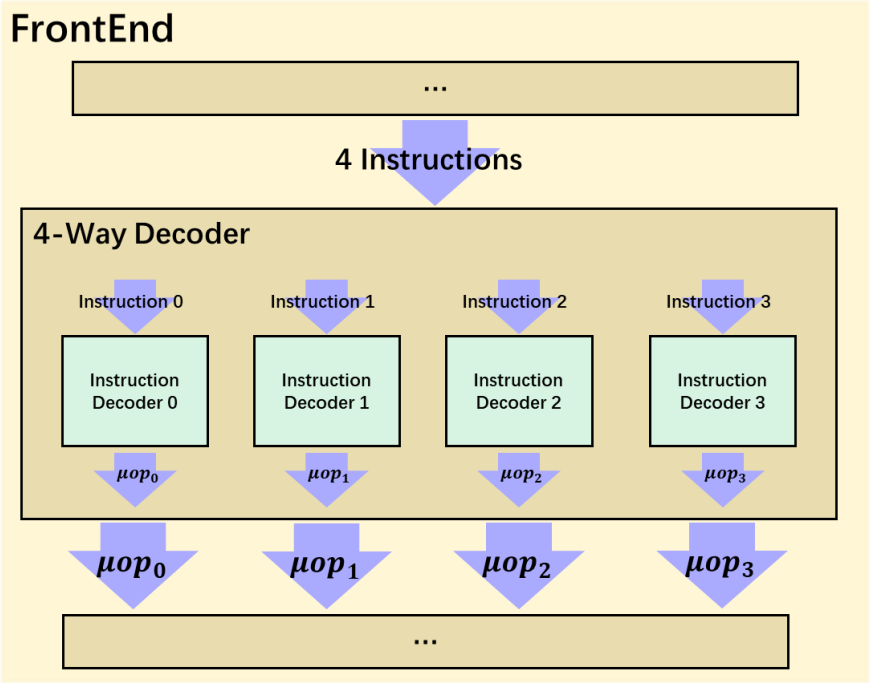


Illustration 6–7 4-way instruction decoder.

Micro operations are designed as a data structure, including source architectural registers, destination registers, used immediate and operation code. Therefore, other stages can fetch needed information conveniently from micro-operations.

6.3.4 Register Renaming

In this part, we rename architectural registers and assign physical registers to them. Besides, the renaming part should also mark physical registers in the retiring instructions as free.

The main purpose of register renaming is to break anti-dependency (write after read, WAR) and output dependency (write after write, WAW) between architectural registers. Therefore, instructions with these dependencies can be pipelined together in the execution stage, which further improves the performance of the processor. Illustration 6–8 shows how to break WAR and WAW in this stage. On the other hand, the renaming part should protect and record the true dependency (read after write, RAW) so that instructions can read and get the right source register. For the zero architectural register, i.e., `x0`, the renaming part should protect it from being assigned to other physical registers.

Our renaming stage is based on an "explicit renaming" out-of-order core design, also called a "physical register file" design. A physical register file (PRF) with 128 entries, containing many more registers than the ISA dictates (32 entries for integer and 32 entries for floating-point), holds both the committed architectural register state and speculative register state. The mapping relations be-

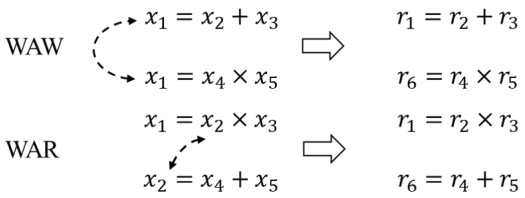


Illustration 6–8 Write after read (WAR) and write after write (WAW) dependency.

tween architectural registers and physical registers are recorded in the mapping table, also called rename allocation table (RAT). A retirement rename allocation table (rRAT) containing the information is utilized to recover the committed state. When a group of operands in the instructions are renamed, their register specifiers are explicitly updated to point to physical registers located in the PRF. Illustration 6–9 shows the rename logic between two instructions and their architectural registers. Since our 4-way rename stage can process 4 instructions simultaneously, the real renaming logic is far more complex than the given example.

Since we use a unified PRF in the register file stage, integer architectural registers and floating-point architectural registers share the same register "pool" and addressing structure. For this reason, we rename their physical registers together and do not consider their differences in this stage.

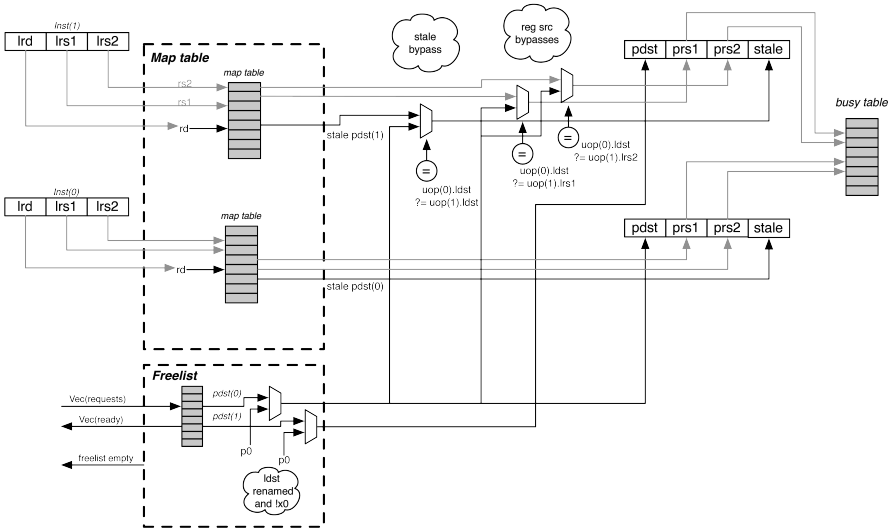


Illustration 6–9 Register renaming logic([11], Celio et al., 2017).

We implement a free list to record the usage of physical registers. When a physical register is assigned to an architecture register, it will be marked as "busy" in the free list. On the other hand, when it is retired by the retired instructions, it will be marked as "free". Illustration 6–10 shows the relation between mapping table, free list, and rRAT.

To resolve data hazards, we apply a conservative strategy. When an architectural register x_a is reassigned to a new physical register r_b, its previous physical register r_a is recorded. Cycles later, when

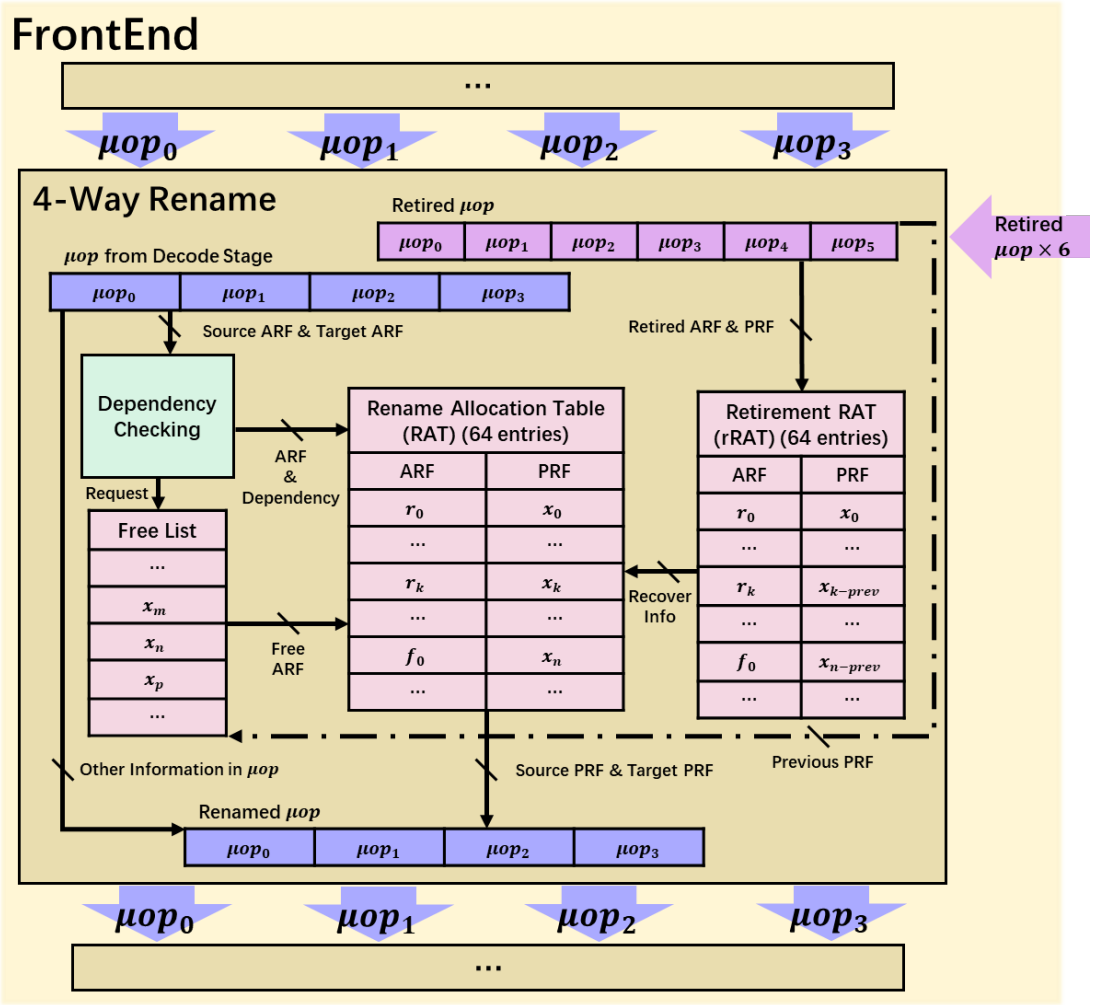


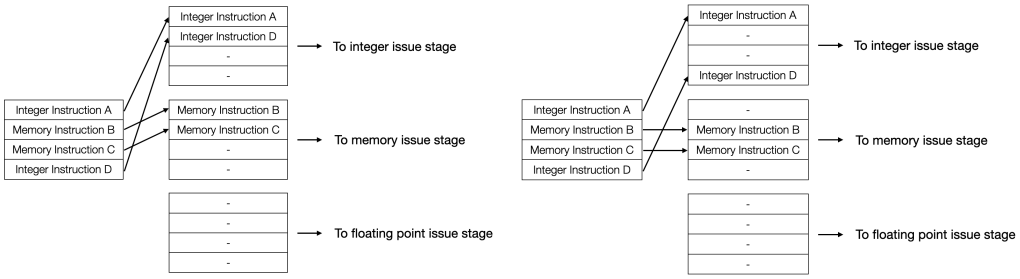
Illustration 6–10 Rename stage with free list, mapping table and rRAT.

the architectural register x_a is retired by its instruction, its previous physical register r_a is freed or assigned to another architecture register. At that time, the free list or the mapping table is updated by the new mapping relation. Besides, rRAT is also refreshed by the mapping relation between architectural register x_a and its new physical register r_b so that the rename stage can quickly recover from misprediction, interrupts, or exceptions.

When a branch misprediction occurs, i.e., when we want to recover the processor states, the mapping table and the free list will recover their mapping relation and free stages from rRAT. The recovery process takes only one clock cycle, which means that our rename stage can quickly respond to branch mispredictions, interrupts, or exceptions. This feature can improve our instructions per cycle (IPC) in programs with many branch mispredictions, interrupts, or exceptions.

6.3.5 Dispatch

Dispatch is the last stage in the frontend and also the last in-order stage. In this part, we write the instructions from the previous stage into the re-order buffer (ROB) the backend and then dispatch these instructions to the corresponding issue units according to the `iq_code` field in the uops. We support dispatching 4 instructions at one time. To reduce the logic complexity of input selection in the issue stage, we design our dispatch stage based on collapsing logic, shown in Illustration 6–11a. The instructions will always be dispatched to the top space in the inputs of the backend without any bubbles.



a) Collapsing dispatch logic. b) Non-collapsing dispatch logic.

Illustration 6–11 Two different dispatch logic design.

On the contrary, a non-collapsing dispatcher is shown in Illustration 6–11b. The logic is simpler, but it will further complicate the input selection logic and increase the delay in the issue stage, especially for memory instructions, which usually require strict memory access to keep the consistency.

6.3.6 Issue

In this part, we accept the input of instructions from the frontend, track the data and structural dependency between instructions, select part of the ready instructions and issue them to the register file and the execution units. This is also the critical point between in-order and out-of-order stages in our microarchitecture design and is the key component to implement an out-of-order scheduling algorithm.

In our design, we implement 3 issue units: integer issue unit, memory issue unit, and floating-point issue unit. All 3 issue units can accept four instructions at one time, but the outputs differ, dependent on the design of execution units.

(1) Issue Unit for Integer & Floating-Point

We design the input selection logic to assign the input instructions to free space in the issue units. The logic of instruction assignment is simple. We just put the instructions in the free space from top to bottom. Meanwhile, we design the output selection logic to issue instructions whose dependency is ready for the subsequent pipeline stages. The output width depends on the execution width. In our design, we can issue at most 3 integer instructions or 2 floating-point instructions simultaneously. Among all the ready instructions, we also select from top to bottom. A simple example of an issue unit for integer instructions is shown in Illustration 6–12.

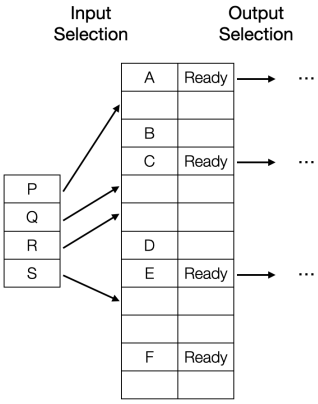


Illustration 6–12 Example of issue logic for integer instructions (L for load, S for store).

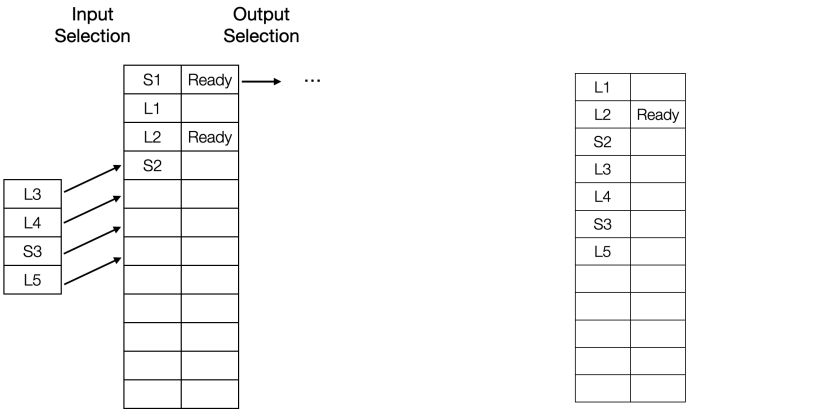
(2) Issue Unit for Memory Access

Due to the special characteristics of memory access instructions, the selection logic of the issue unit for memory access is different from the logic of the other two issue units. The main principle is to keep the memory consistency, which is a big challenge for out-of-order execution design. It is easy to track data dependency in registers, but it is much more difficult to track the dependency in memory. For example, assuming that the same memory address is stored in the registers `x1` and `x2`, for the case that we first store the data to `[x4]` and then load the data from `[x3]`, we cannot directly determine whether there exists potential dependency between the two instructions.

Thus, in our processor design, we use a relatively conservative way to achieve memory consistency. We consider each store instruction as a barrier, i.e., while load instructions can be executed out-of-order, we can issue the store instructions only when it is at the top of our issue unit. As a result, similar to the dispatch unit, we design a collapsing structure in the issue unit for memory access. Each time one instruction is issued, all the instructions are “compressed” to the top, and the input instructions are added to the tail of the unit. For the output selection logic, if there are remaining store

instructions, only the store instruction at the top or the load instructions before any store instructions can be issued.

A simple example is shown in Illustration 6–13. At the top of the issue unit is a store instruction, which will be issued in this clock cycle. Then all the remaining instructions are pushed to the top, and new instructions are added below.



a) Example of issue unit logic: stage 1.

b) Example of issue unit logic: stage 2.

Illustration 6–13 Example of issue unit logic for memory access.

(3) Scoreboard

Although register renaming may resolve anti-dependency and output dependency, it cannot resolve the true dependency, i.e., read-after-write (RAW) dependency, which is usually the crucial bottleneck to further improve instruction-level parallelism. In our design, the scoreboard is mainly used for tracking data dependency between instructions and sending wake-up signals to the issue units.

The width of the scoreboard matches the depth of our physical register file. First, it accepts input from the dispatch stage, which is the last in-order stage, and sets the busy bits in the scoreboard. Second, it accepts input from the write back stage, which clears the busy bits in the scoreboard. Finally, it accepts the query requests from the issue units and returns whether the requiring source registers are ready or not.

Illustration 6–14 shows an example of how the scoreboard helps to track RAW dependency between two instructions. Here the second instruction depends on the result of the first instruction. First, when the two instructions enter the dispatch stage, set both `X` and `Y` as busy, marking them as in-flight instructions (Illustration 6–14a). Second, the first instruction is issued to the later stages of the pipeline (Illustration 6–14b). Third, when the first instruction is completed and writes back to the register file, set `X` as not busy (Illustration 6–14c). At last, as `X` is not busy anymore, the second

instruction checks the scoreboard, finding that it is also ready to issue, and we issue it to the register file and corresponding execution units (Illustration 6–14d).

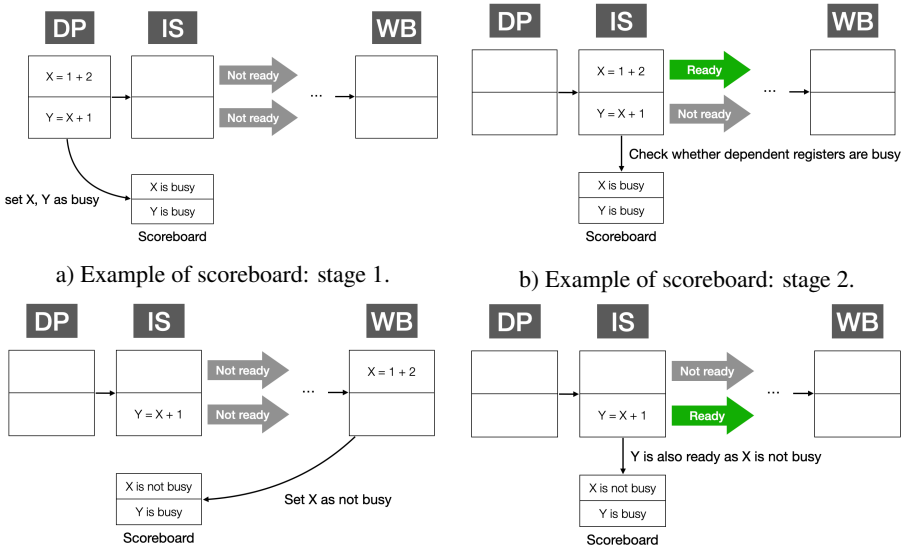


Illustration 6–14 RAW dependency tracking with scoreboard.

6.3.7 Register File

As mentioned in the register rename part, our processor resolves anti-dependency and output dependency based on the physical register file (PRF) design. In our design, there is only one unified PRF with 128 entries. Bypassing logic is implemented to handle the case that source operands are identical with destination operands.

Compared to other methods of register renaming, there are several advantages of using PRF. There is no need to move the data frequently, as the data always stays in the PRF. We don't need to add selectors for instructions to determine which register file to use. Furthermore, It is also easy to recover from branch mispredictions, which helps to further explore the instruction-level parallelism and improve the performance.

6.3.8 Execution

An execution pipe is a set of function units. In our microarchitecture design, we have 6 execution pipes in total (3 for integer, 1 for memory access, and 2 for floating-point), shown in Illustration 6–15.

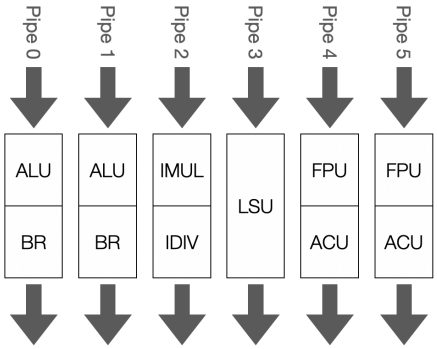


Illustration 6–15 Execution pipes.

(1) Execution Units for Integer

There are 3 pipes for integer instructions. Pipe 0 and 1 encapsulate arithmetic logic units (ALU), which perform normal integer arithmetic operations except multiplication and division, and branch units, while pipe 2 is used for integer multiplication and division. ALU and branch units require one clock cycle delay, while integer multiplication and division require more than one clock cycle delay and they are blocking. Instead of using commercial intellectual property (IP) cores, we implement the integer multiplier and divider by ourselves based on open-source designs.

(2) Execution Units for Memory Access

Pipe 3 is for memory access, i.e., load and store instructions, and it is blocking. When a load or store instruction enters this pipe, the processor will send the address, write-enable signal, data to be written, and its size (if applicable) to the memory, and receive the data and valid signal from the memory.

(3) Execution Units for Floating-Point

There are 2 pipes for floating-point instructions. Both pipe 4 and 5 encapsulate floating-point units (FPU) and approximate computing units (ACU). Each pipe consists of two function units, FPU for general floating-point computation and ACU for approximate computation.

6.3.9 Write Back

When an instruction is completed, the result will be written back to the register file and clear the corresponding busy bit in the scoreboard if the destination operand is valid.

6.3.10 Commit

The commit stage is the last stage in the processor to handle instructions, consisting of a re-order buffer (ROB) that tracks the state of all in-flight instructions in the pipeline. Besides, the ROB also records the order of instructions before they are dispatched into issue queues. The ROB is a first-in-first-out (FIFO) queue.

When an instruction is renamed and put into the ROB, it will be marked as “busy” in the ROB. On the other hand, when it enters the commit stage, it informs the ROB and is marked “not busy”. Once the “head” of the ROB is no longer “busy”, the instruction is retired, and the architectural state of the processor is updated. Illustration 6–16 shows the structure of our ROB design.

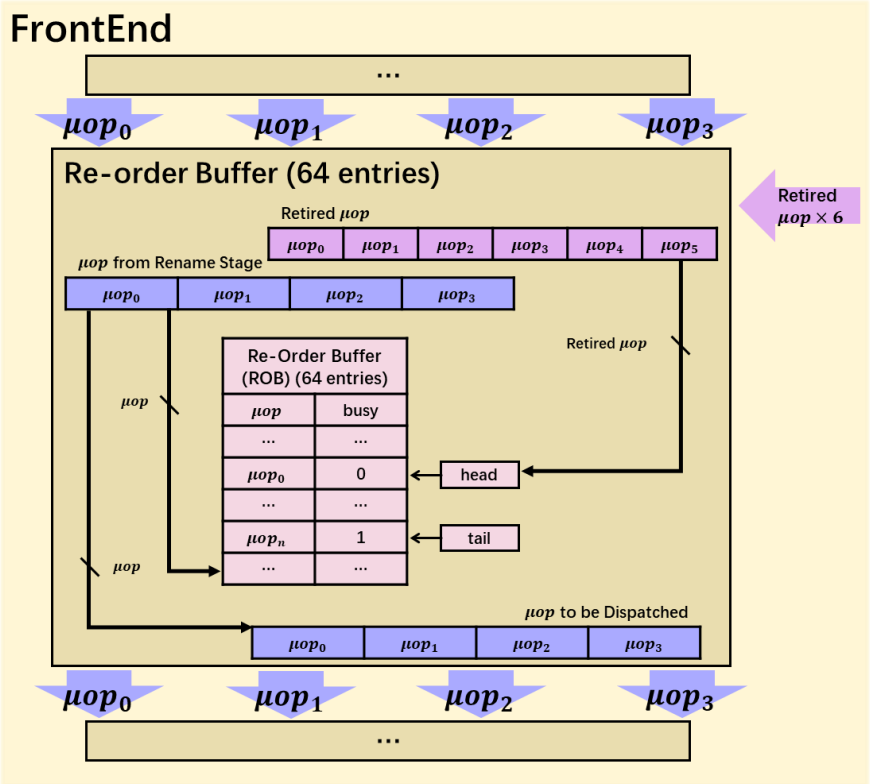


Illustration 6–16 Re-order buffer (ROB).

Since we have 4-way renaming and 6-way execution, the ROB can accept 4 instructions input and pop 6 instructions simultaneously. Some special techniques need to be applied in the final circuit

implementation, like interleaving, to satisfy the requirement.

Branch mis-predictions interrupts or exceptions are handled when they are at the commit head. When mispredictions, interrupts, or exceptions happen, their following instructions are marked as illegal and their previous instructions are all retired normally. The ROB should be cleared and flushed then, and send the "recover" signal to all the previous stages in the pipeline.

6.3.11 Cache

Both hardware cache and software memory systems are modeled, during the first half and the second half of this project. However, in the end, the software one is used due to other constraints.

(1) Hardware cache model

Both instruction cache and data cache are modeled. Each cache has a 64 KB size in total (16-word cacheline and 1024 cachelines). The placement policy is the direct-mapped method, and the writing policy is the write-back method with write allocation. Parameters including the number of cachelines, the size of the cacheline, and, as a result, the total size of the cache are configurable. Both caches share a similar design as demonstrated in Illustration 6–17. The cache main pipeline accepts requests from and gives results back to the processor core, based on a state machine. If there is a cache miss, the main pipeline will send an allocation request to the allocating unit. The allocating unit will allocate data from `AXI_slave` port (on the FPGA, it will be a DRAM controller) through an AXI interconnect bus, which takes several clock cycles depending on AXI bus configurations. If the victimized cacheline is dirty, the cache will issue a write back request to the write back unit, which takes several clock cycles to finish the write back request.

(2) Software memory model

Due to resource limitations and other constraints, a software-based simulated memory model is used in our project, shown in Illustration 6–18. The main memory is modeled by the C++ abstract class `IdeaMemory`, which is implemented by `BucketMemory` class, which implements a hash map to model the memory, and each entry of the hash map holds a reference to a block of memory with 1K bytes. The processor access the memory through two wrappers `DMem` and `IMem`, which provide clean interfaces. The memory model is integrated into our Verilator simulator.

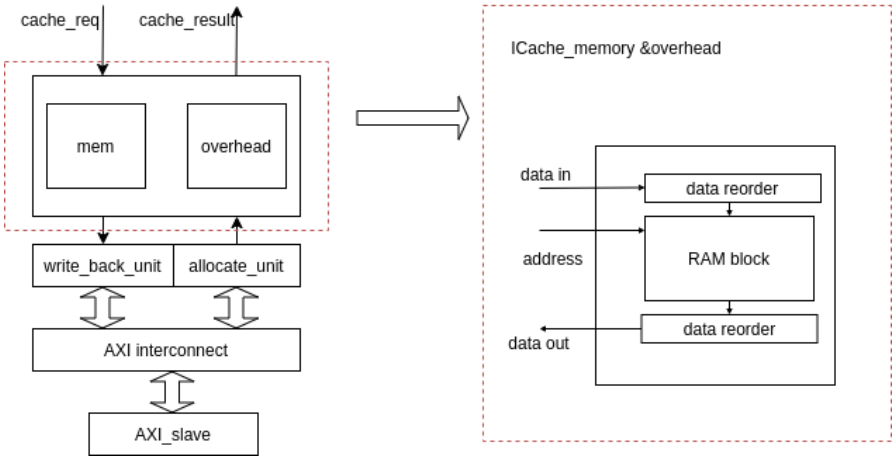


Illustration 6–17 Hardware cache model.

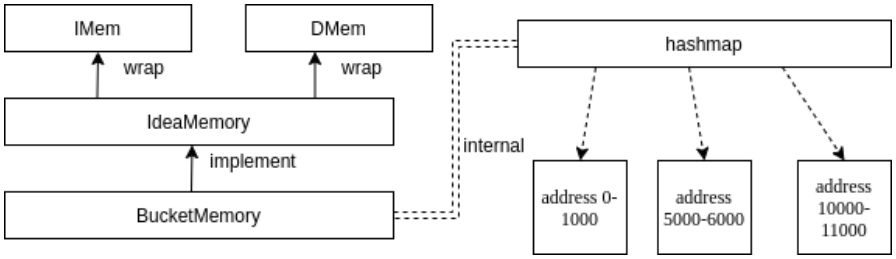


Illustration 6–18 Software memory model.

7 Implementation & Validation

7.1 Project Timeline

Based on the project specification, the overall workload can be divided into three milestones. We arrange the timeline of each milestone according to the deadline of three design reviews and the final prototype review. Meanwhile, we classify our work into five categories: **Memory and Compilers, Computation units, Pipeline components, Testing** and **Technical Communications**. **Memory and Compilers** is about simulating the DRAM memory on Verilator and export the RISC-V assembly code from our customized compiler to the memory simulator. **Computation units** covers integer arithmetic logic unit (ALU), multiplier, divider, floating-point unit (FPU), and approximate computing units. **Pipeline components** includes every core component like mapping table, re-order buffer, and issue queues. **Testing** is basically to verify the correctness of every module we implement. **Technical Communications** involves every presentation, report, and other broadcast media we need to make for design reviews, advisor meetings, and the final expo. Illustration 7–1 shows a simplified Gantt chart of the project. The testing part should be conducted throughout the develop-

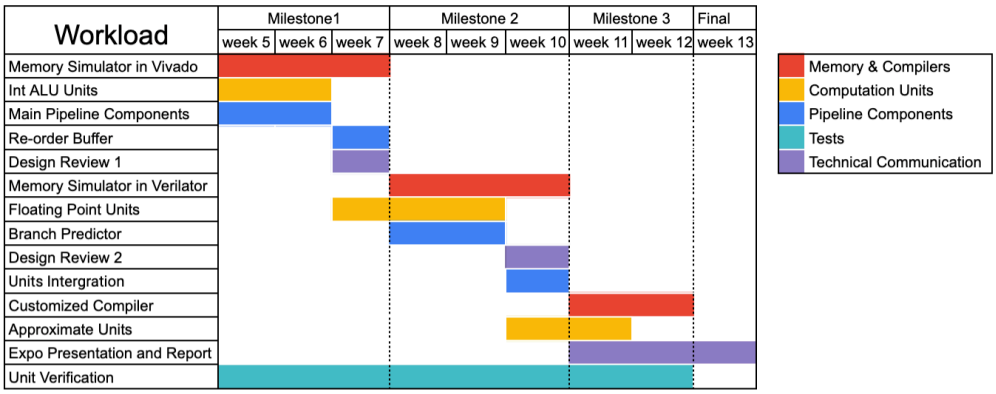


Illustration 7–1 Project Gantt chart.

ing process. The rest four workloads will be distributed evenly into every milestone to make sure we can always balance our pace.

From the chart, we can observe that Week 2-5 is not included because what we have mainly done is preparation. We familiarize ourselves with the memory blocks on the FPGA board and start to write the main components in the RISC-V out-of-order pipeline.

1. By milestone 1 (week 5 - week 7), we finished most parts of a naive RISC-V out-of-order core such as issue queues, re-order buffer, etc.illustr
2. By milestone 1.5 (week 7 - week 8), we finished the simulation of a RISC-V out-of-order core to make sure the whole core runs without any syntax errors.
3. By milestone 2 (week 8 - week 10), we completed extra components like approximate units and customized compilers.
4. By milestone 3 (week 10 - week 12), we will have a complete flow to run an image processing C program on our synthesized RISC-V core.

We assign different workloads to different teammates based on their respective expertise. Jian Shi and Yichao Yuan will be in charge of memory-related units. Yiqiu Sun will be responsible for computation units. Li Shi and Zhiyuan Liu will be responsible for pipeline-related units. Details will be discussed in each group member's individual report.

Our initial goal is to build our whole design on the FPGA board. However, after synthesizing some parts of our modules on Vivado, we found that our available FPGA board is inadequate to build such large circuits. There are too many unavoidable I/O ports in our design. Therefore, we have switched our plan to run software simulation of our design through Verilator. At this point, every workload can be done on software. In conclusion, the overall project is expected to be finished without any funding.

7.2 Implementing Plan

The whole implementation involves different levels of the computer architecture hierarchy, with ranges from upper-level compiler design to lower-level circuit synthesis. Illustration 7–2 shows a flow chart of our implementing plan.

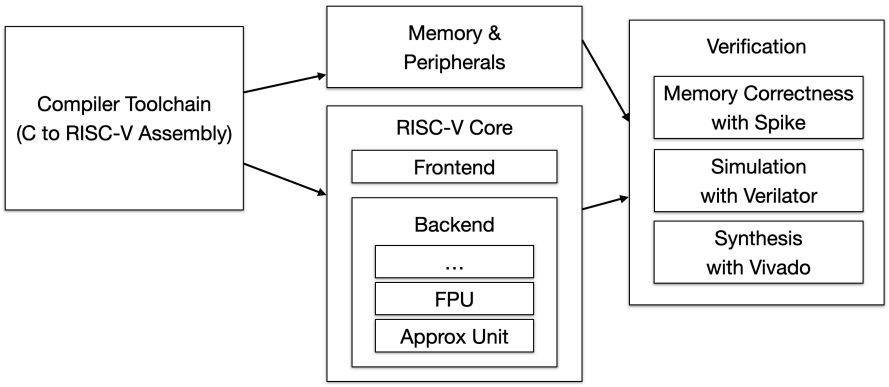


Illustration 7–2 Implementing plan diagram.

To make our processor "understand" those operations assigned, we will first build a compiler to "translate" higher-level programming language like C language into assembly code, which can be processed by the processor. Meanwhile, we also need to customize the compiler to convert floating-point operations into approximate computing operations instead of accurate computing operations. The detailed implementation is explained in section 6.2.1.

After the program is translated to assembly code, it is ready to be transported into the memory and run on the RISC-V core that we are going to build. The memory core is a simulator of the DRAM, which is always on a separate chip other than the CPU chip in a real case. The RISC-V core will be written in SystemVerilog, which is a hardware description language used to design and test electronic systems. The detailed design of our RISC-V core is demonstrated in section 6.1.2.

After the completion of the memory system and RISC-V processor core, we need to perform several levels of verification towards our designed circuit because a functional-correct design might not be practical for real circuit synthesis. The whole verification includes memory correctness, simulation correctness, and synthesis correctness. We use software tools such as Spike, Verilator, and Vivado. The details will be discussed in the following sections.

7.3 Validation Tool

We introduce the validation part through a top-down approach: we will first introduce the software stack, then the simulator architecture. We validate our design by comparing the simulation of our processor model and a golden RISC-V reference model, Spike. The software needs to compile the source code into binary that can utilize the approximate computing units in the processor model as well as run on these two targets.

7.3.1 Program layout in memory

We need a clear picture of the program layouts in the simulator and Spike so that we can control the IO behavior of the program, like referring to where to fetch instructions, data, and output results. The layout of the program and the physical address space in Spike are shown in Illustration 7–3. The left-hand side shows how the program will be loaded into our verification environment. `data` and `bss` segments are placed starting from memory address `0x10000000` and the text segment is placed starting from memory address `0x80000000`. The stack is set to start from memory address `0x20000000`, growing downwards.

The right-hand side is the devices related to the physical address in Spike. Our modified Spike simulator as a DRAM device starting from memory address `0x10000000`, where all the segments and the stack locate. There are some other devices in Spike, like ROM, CLINT, and MMIO. However, none of them will be used in this project so they will not be introduced.

Our simulator using Verilator includes an idea memory space, which means each physical address corresponds to a piece of memory in the host machine.

7.3.2 IO libraries

On a regular Linux operating system, some special operations like opening a file and printing a string require system calls. Similarly, in Spike simulator, the running program performs such operations through special methods. We provide several functions to encapsulate the IO operations. The implementation of the library has a lot to do with Spike architecture, which will be introduced later. Following is a list of functions we provide.

1. `tohost_exit()` function takes an integer as the exit code and terminates the simulation.
2. `tohost_printstr()` function prints the given string.

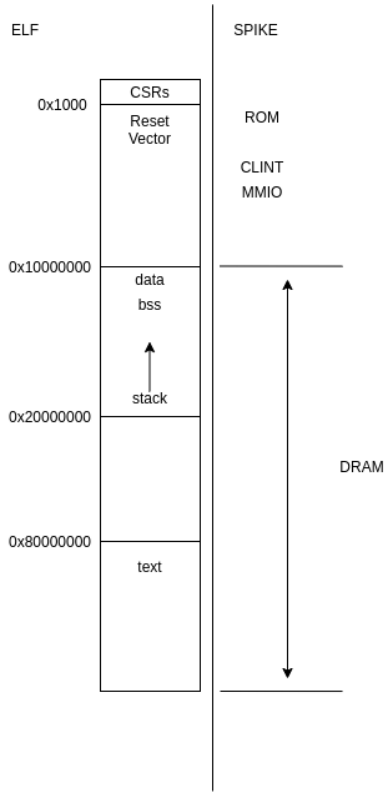


Illustration 7–3 Address space configuration in ELF format and Spike simulator.

3. `tohost_open()` function takes the filename and flags as arguments, and returns a file descriptor
4. `tohost_close()` function closes the file corresponding to the given file descriptor.
5. `tohost_write()` function writes data in the buffer to the file corresponding to the given file descriptor.
6. `tohost_read()` function reads data from the file corresponding to the given file descriptor to the buffer.

These functions are similar to the system calls in system programming. The functions configures the registers properly using the convention with our frontend server. Users can use `tohost_open()`, `tohost_close()`, `tohost_write()` and `tohost_read()` to perform I/O operations with a file. For example, an imaging program can open a file to read the inputs and write the output to a file.

7.3.3 Compilation Toolchains

We also need to modify the compilation toolchains accordingly, so that the compiler can issue and disassemble our customized instructions. RISC-V has its own compilation toolchains and provides

multiple types of cross-compilers listed below

- `riscv32/64-unknown-elf-gcc`
- `riscv32/64-unknown-linux-gnu-gcc`
- `riscv64-multilib-elf-gcc`
- `riscv64-linux-multilib-elf-gcc`
- `riscv-none-embed-gcc`

The suffix `32` and `64` points out whether to support 32-bit RISC-V architecture or 64-bit RISC-V architecture while the cross-compiler with the `multilib` suffix can support both architectures at the same time. The cross-compiler with the `linux` suffix uses `glibc` a dynamic link library, as C runtime library, while the one without the `linux` suffix uses the static link library `newlib` as C runtime library. The last cross-compiler is designed for bare-metal embedded system.

In our project, we propose an SoC design for embedded systems. For embedded systems, we should use the static link library (`newlib`) as C runtime library. Our project only supports 32-bit RISC-V architecture now, but to be better compatible with 64-bit architecture in the future, we choose the `multilib` cross-compiler. In summary, we choose `riscv64-multilib-elf-gcc` as our target cross-compiler and modify it to support our customized instructions.

We also use `riscv-opcodes` tool to get the "mask and match" of our customized instructions. Our cross-compiler will use the "mask and match" to identify our customized instructions. Take `FMULA.S` as an example, by parsing opcodes, we can get `#define MATCH_FMULA_S 0x30000053`, `#define MASK_FMULA_S 0xfe00007f`. After changing the corresponding files in `riscv-binutils` and `riscv-gdb`, our cross-compiler supports assembling and disassembling customized instructions and our customized instructions can be invoked through C inline assembly code.

The compilation is managed by `GNU Make` tool. The whole compilation process is demonstrated in Illustration 7–4.

The library includes the startup code and IO libraries, which are compiled into object code in advance. Since we do not have an OS environment, we cannot use the default startup code, so we write the startup code using assembly code, which does nothing more than setting up the stack and jumping to `main` function. The IO libraries provide methods to open files, write files, etc. The program sources are written in C and are compiled into object files for linking. We use a link script to control the program layout. The text segment is placed starting from memory address `0x80000000` and the data segments, including sections like `.data`, `.bss`, etc., are placed starting from memory address `0x10000000`. The linker will then output a Unix ELF file for execution.

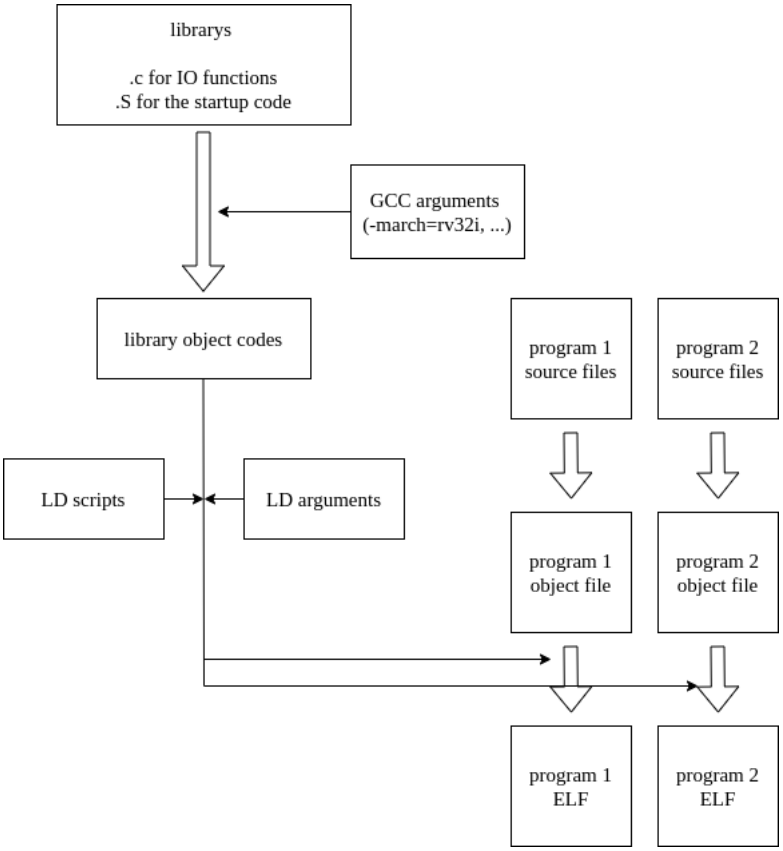


Illustration 7–4 Compilation toolchains setup.

7.3.4 Cycle-Accurate Verification Tool: Verilator

Verilator is a program that can convert register transfer level (RTL) models described in Verilog and SystemVerilog into C++ source files, in which the RTL models are converted into classes. This process is called "verilating" and the generated models are called "verilated" models. The generated files can then be linked to the C++ simulation environment to perform further simulation. Users can write their own C++ "wrapper" files, which instantiate the generated models. By setting values to the variables that correspond to the signals in the HDL model, the simulation is performed. During the simulation, Verilator can generate useful information like log files as well as the waveform. The whole program is compiled by a normal C++ compiler and converted into an executable. Users can execute the executable to start the simulation. We use Verilator to perform logic simulation. The signal information is printed and the waveform is dumped into log files so that we can validate the correctness of our design.

The Verilator C++ wrapper files are integrated into the whole verification system. The verilated CPU model will have access to an ideal memory modeled by C++. The verification system will load the program into the idea memory based on the information in the ELF file. The wrapper file will also take responsibility to handle IO operations like printing and saving files.

The detailed architecture of this verification environment is demonstrated in Illustration 7–5.

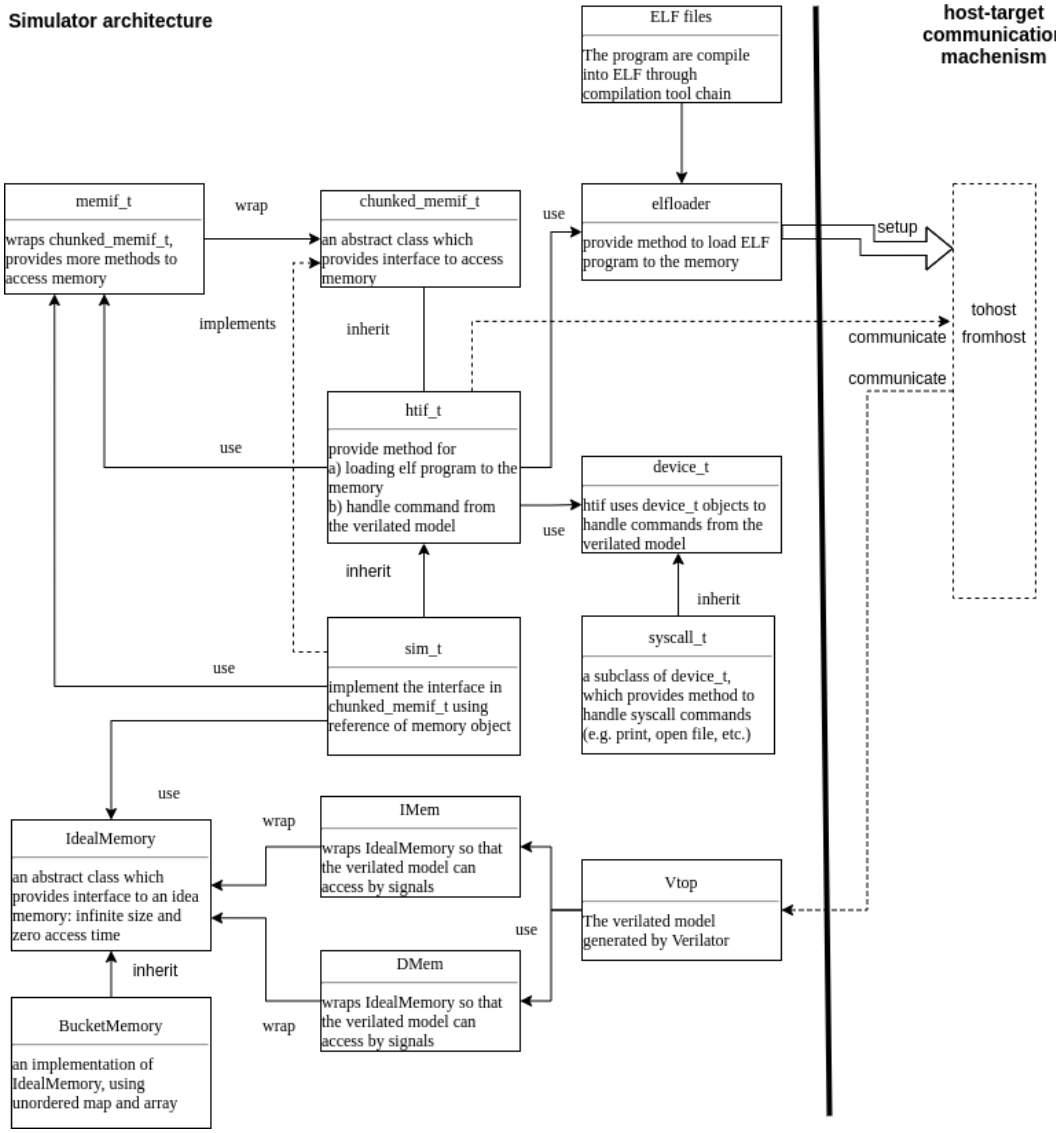


Illustration 7–5 Verification environment based on Verilated model

This architecture reuses codes from Spike simulator. The `chunked_mem_if` is an abstract class that provides interfaces to access the memory by other components in the simulator. The `mem_if` class is a wrapper for the `chunked_mem_if`, which gives more convenient interfaces to access the memory. The `htif_if` inherits from the `chunked_mem_if` class, which provides the methods to load the ELF files into the memory and handle communication with the running software. The `sim_t` inherits from `htif_if`, which use the reference to an object of `IdeaMemory` to implements the interface in `chunked_mem_if`. The `IdeaMemory` class is an abstract class, which is implemented by `BucketMemory`. The Verilated model uses two wrappers, `DMem` and `IMem`, to access the memory. They provide convenient interfaces for the Verilated Model.

The Verilated model, or more precisely the running program on the verilated model, communicates

with each other through two variables in the memory, `tohost` and `fromhost`. These two variables should be defined in the program and they will have records in the ELF file. When the `htif_t` loads the program, it will track the position of these two variables. The Verialted Model can write a value, which is called a command, in the `tohost` variable. The `htif_t` will monitor the variable and process the command using `device_t` classes. The result will be written in the `fromhost` valuable. The software can wait on the `fromhost` variable to finish the communication.

7.3.5 Trace-Accurate Verification Tool: Spike

Spike is a "golden" model of RISC-V which provides a trace-accurate model for the RISC-V specification. By using Spike, we can know the exact behavior of the correct processor. By comparing the result of the spike and the instruction retired from our Verilator model. If they are identical, then we can be sure we implement the specification correctly.

The front-end architecture, the way it loads the program and handles the communication between targets, has been introduced in the previous section because we build our Verilator environment using Spike's architecture.

The original Spike (v1.0.0) has limited IO support for `XLEN = 32` RISC-V architecture. Under the RV64 architecture, it is possible to use the spike's hardware-target interface to print characters on the screen by setting the command's device to 1. However, this interface is based on a 64-bit register. Under the RV32 architecture, there is no way to do atomic 64 bits write so the high 32 bits will always be 0. As a result, we will have no access to the device to print.

In this project, we proposed an additional system call `sys_printstr` which uses the same system call procedure to evoke like any other system call in spike. The new system call occupies the number 2012 and takes the pointer to string as an argument. This system call will print the string onto the standard output.

Another change we made on the original spike is to change its memory mapping. the original memory map maps the DRAM to start from memory address `0x80000000`, which limits the way we write the linker script. We changed the memory mapping so that most of the physical memory has a meaning.

To validate the result of the architecture simulator and the instructions retired from our Verilator model, Spike should also support our customized instructions. Section 6.2.1 describes these instructions and our modification on ISA in detail.

To support our customized instructions in Spike, we add two instructions, `MASK` and `MATCH`, in `riscv-isa-sim/riscv/encoding.h`. Besides, we add those instructions in their corresponding extensions in `riscv-isa-sim/riscv/riscv.mk.in`. An approximate computation algorithm written in C language is used to describe the functional behavior of our approximate computing units, which is added to the folder `riscv-isa-sim/riscv/insns` so that Spike can simulate our design simultaneously. To support disassembling customized instructions in Spike, corresponding instruction descriptions are added in `riscv-isa-sim/disasm/disasm.cc`.

7.3.6 Logic Synthesis & Implementation Tool: Xilinx Vivado

Xilinx Vivado is a software electronic design automation (EDA) tool produced by Xilinx for the synthesis and analysis of hardware designs. With Xilinx Vivado, we can measure the usage of resources on FPGA after synthesis and implementation. Besides, we also use Xilinx Vivado to test the core frequency on FPGA.

Xilinx Vivado can also return the ideal power consumption on FPGA after proper settings. However, this value is just a result given by the simulator, not from a real FPGA board. Therefore, there might exist errors or biases.

7.4 Validation of Engineering Specifications

During the whole validation process, most of our engineering specifications will be tested. Part of engineering specifications require advanced test methods or more general tests and the overall validation results will be demonstrated in the final report.

7.4.1 Validation Environment

(1) System Dependency

The whole system is tested on Ubuntu 20.04.2 LTS (kernel version 5.8.0-63-generic) with the following software dependencies:

1. Verilator v4.205
2. `riscv64-unknown-elf-gcc` (GCC) 10.2.0

3. `GNU ld`(`GNU Binutils`) 2.36.1
4. `GNU Make` 4.2.1
5. Vivado v2020.2 (64-bit)

(2) Validation Guide

The whole system is managed by `GNU Make`. The default target for make is to run a simulation based on verilated model.

```
make SIM_TARGET=<simulator framework> SIMULATOR_PROG=<program>
```

The option `SIM_TARGET` selects a test framework to run. Target `sim_main` runs a raw test, in which the input program should be a raw binary. This is used for early-stage debug and development. The suggested target is `sim_main2`. This target provides a spike-like simulator, in which the input program should be an elf. The framework will capture the entry point and enable utilities like system call.

The option `SIMULATOR_PROG` selects a program to run on the simulator. The program is compiled by the flow in `spike-software`. Our environment supports the user add their program for test and debug. User can write their C program in the directory spike-software/progs. Each C program must define a `reg_t tohost`, a `reg_t fromhost` and a `reg_t tohost_cmd[8]` to enable syscall utility. All variables need to be initialized to 0. The make will compile the software before simulation. Or, you can compile manually by `make make-spike`.

To see the result from the Spike reference model, use the following command.

```
make sim-spike SPIKE_PROG=<program>
```

The `<program>` here should be the same program being used in the above step. The make will invoke the `spike` the perform simulation. The user can add a `-l` option to the spike if they want to see the trace of execution. The user can compare the output from the verilated model and the spike to debug.

The operation on Xilinx Vivado depends on the folder you install Vivado Tool chain. Therefore, we assume that the user knows how to open Vivado Console. All following commands are run and operated in Vivado Console.

Besides, the measure results from Vivado is closely related to the model of chosen FPGA board. Take our program as an example, we choose `xczu19eg-ffve1924-2-i` as the motherboard of our

processor. The user can create the project and assign the FPGA model by `create_project RIA . -part xczu19eg-ffve1924-2-i`.

7.4.2 Validation Result

Table 7–1 Engineering specification validation (↑ high is better / ↓ lower is better).

Engineering Specification	Unit	Target	Actual	Comment
Support RV32G instruction set architecture (ISA)	-	Yes	Partial	RV32IM
Core frequency on FPGA test platform	MHz	100 ↑	74.88	Synthesis result
Number of pipeline stages	-	9	9	
Instructions executed per clock cycle (IPC)	-	0.5 ↑	0.601	Simulation result
Support instruction dynamic scheduling	-	Yes	Yes	
Typical total cache size	KB	32 ↑	128 / ideal	Ideal software model
Number of function units	-	6 ↑	6	
Average response time to a request for service	ms	10 ↓	ideal	Ideal software model
Usage of look-up tables (LUT) on FPGA	k	120 ↓	101.7	Synthesis result
Usage of block RAM (BRAM) on FPGA	-	50 ↓	0	Synthesis result
Usage of digital signal processor (DSP) on FPGA	-	30 ↓	8	Synthesis result
Power consumption on target FPGA test platform	W	5 ↓	2.63	Synthesis result
Operations processed within unit energy.	MOp/J	25 ↑	17.11	Synthesis result
Number of flexibly-configured modules	-	10 ↑	13	
Number of I/O device types	-	3 ↑	3	File r/w & standard output
User guide and programmers manual	-	Yes	Yes	

The validation result of engineering specifications is shown in Illustration 7–1.

Part of engineering specifications can be validated manually by checking the source code of the RTL design model or the corresponding supplementary materials of our processor:

1. Number of pipeline stages: We check the original design diagram of our processor (Illustration 6–5) and count the number of pipeline stages.
2. Number of function units: We check the design and count the number of function units.
3. Number of flexibly-configured modules: We check the source code and count the number of flexibly-configured modules, including fetch buffer, register renaming unit, issue units, etc.
4. User guide and programmers' manual: We check the manual supplementary materials of our processor.

Part of engineering specifications can be validated by software validation tools, including Verilator and Spike:

1. Support RV32G instruction set architecture (ISA): We cross-compile C programs into binaries compatible with RV32G ISA and run them on our processor.
2. Instructions executed per clock cycle (IPC): We run benchmark programs on our processor and acquire the data of both the number of instructions and clock cycles to execute the programs. We then derive the average value of IPC of our processor.

3. Support instruction dynamic scheduling: We check Verilator simulation logs and prove that the instruction dynamic scheduling algorithm works well.
4. Typical total cache size: We manually set the cache size in Verilator simulation model.
5. Average response time to a request for service: We count the clock cycles of some response programs. Based on core frequency on the FPGA test platform, we then derive the average response time.
6. Number of I/O device types: We count the virtual I/O devices in our software model.

Part of engineering specifications can be validated by Xilinx Vivado EDA tool:

1. Core frequency on FPGA test platform: During the process of synthesis and implementation, Vivado performs static timing analysis (STA) on our RTL model and gives the data of critical delay in nanoseconds. We then derive the core frequency of our processor on the FPGA test platform.
2. Usage of look-up tables (LUT), block RAM (BRAM), and digital signal processor (DSP) on FPGA: We check Vivado synthesis report and acquire related data of resource usage on the FPGA platform.
3. Power consumption on target FPGA test platform: We check Vivado synthesis and implementation report and acquire the dynamic power of our design.
4. Operations processed within unit energy: Based on specific benchmark programs, the value of core frequency, and power consumption, we estimate the value of operations processed within unit energy. The formula to calculate is

$$\text{Operations processed within unit energy} = \frac{\text{IPC} \times \text{Core Frequency}}{\text{Power consumption per second}}$$

The correctness of the CPU is validated using the test suit describe above. We performed the test on

1. a simple program written in assembly to test whether the CPU can correctly conduct simple instructions like arithmetic, jump, branching etc.
2. some simple C programs which do not include any system calls. These programs are tested to check whether the CPU can run the simplest real program as well as whether the whole validation flow works correctly.
3. some C programs that can do real works using system calls. An image processing program is compiled and tested.

We also do a comparison between accurate computing and approximate computing. Table 7–2 lists the results of the comparison and the error between them.

An image processing program is also executed on Spike simulator for comparison and the processed

Table 7–2 Accurate computing vs. approximate computing.

Equation	Accurate	Approximate	Relative Error
6.023345 / 2.044555	2.946042	2.945312	-0.02%
8.9342 / 1.56	5.727052	5.730469	0.05%
8.9342 × 1.56	13.937352	13.937500	0.001%
6.023345 × 2.044555	12.315060	12.320312	0.04%

images are shown in Illustration 7–6. By comparing the processed images, we can see that the results are very similar. Approximate computing loses a small part of the accuracy, but at the same time it also improves the computational efficiency, and the result is acceptable.

a) Accurate computing results. b) Approximate computing results.

Illustration 7–6 Accurate computing results vs. approximate computing results.

8 Discussion & Future Work

8.1 Load-Store Unit

8.1.1 Potential Problem Analysis

The most significant characteristic of an out-of-order core is that the order of executed instructions is random, including memory access instructions. With the help of ROB, we can ensure that the results of all the integer and floating-point instructions are correct. However, without a specifically designed load-store unit, we cannot ensure that the content in the memory is consistent. For example, consider the following two instructions.

```
sw x1, 0(x2); lw x3, 0(x4)
```

If the addresses stored in `x2` and `x4` are identical, there exists potential memory data dependency, i.e., memory alias, under which circumstance the two instructions must be executed sequentially. Thus, we need a load-store unit to ensure that the memory consistency is kept. As mentioned in the issue logic for memory access instructions, currently our strategy is to consider all store instructions

as barriers, i.e., we don't care about the order of load instructions between two store instructions, but the store instructions must be executed sequentially.

However, we still need to consider the case of branch mispredictions. If a branch misprediction occurs, speculative load instructions will not influence the memory consistency, but speculative store instructions will. For example, a store instruction following a branch misprediction will write the wrong value to the memory, which is irrevocable.

8.1.2 Current Solution & Future Work

To resolve this conflict, a load-store unit is necessary to keep memory consistency. Currently, we design a simple store buffer simulated in Verilator. As it is implemented in software, there is no limitation in terms of size and delay. When speculative store instructions are executed in the execution stage, the processor will send a store request to the store buffer. Only when these store instructions commit, the store buffer will write the data to corresponding memory addresses. If a branch misprediction occurs and sends the recover signal, the content of the store buffer will be immediately flushed. Furthermore, when load instructions are executed, we will first check whether there exist store instructions that share the same address with load instructions. If so, we directly return the data from the store buffer. Otherwise, we load the data from the main memory as usual. Fig. 8–1 shows a simplified diagram of store buffer in our memory access system.

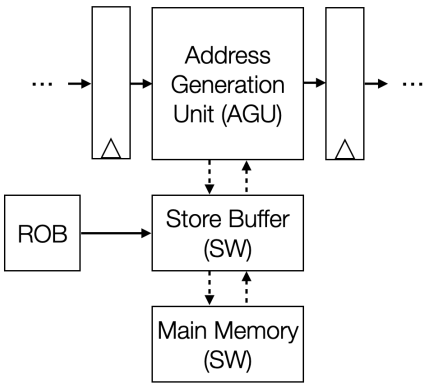


Illustration 8–1 Current solution of store buffer.

In the future, we may implement the store buffer in the RTL design. We may also further improve the performance of memory access by introducing more complicated load-store units, e.g., miss status handling register (MSHR).

8.2 Pipeline Optimization

8.2.1 Strengths of Our Pipeline Design

The traditional 5-stage in-order pipeline has been widely used in industry and academia for a long time. However, there are still some drawbacks. For example, due to the data, control, and structural hazards, “bubbles”, or “nops” (no operation) may appear in the pipeline, which may reduce the processor performance. When the processor is communicating with the memory and waiting for the data, it will stall until the data is loaded from memory or stored into memory. Furthermore, as the number of stages is relatively small, the critical path in each stage may be long, leading to a high delay and low core frequency.

In our project, our out-of-order processor successfully overcomes the aforementioned drawbacks. When there exist data, control, or structural hazards, due to the advantage of the instruction dynamic scheduling algorithm, subsequent instructions can fill the pipeline and avoid the potential pipeline “bubbles”, resulting in higher performance. Also, the load and store instructions can be executed partially out-of-order, so the efficiency of memory access is also improved. Furthermore, as the number of stages is larger (9 stages in our design), the critical delay can be theoretically reduced, resulting in a higher core frequency.

8.2.2 Potential Problem Analysis

We designed our out-of-order 9-stage processor based on several designs such as the ”BOOM” design mentioned in Section 2.3. However, there is still some room for optimizations in microarchitecture. Two stages that perform independent operations can be combined. The stage that takes a longer time than others should be split to shorten the critical path.

In the old design, decoding and renaming are separated into two stages. This means we have to wait for at least two cycles to dispatch decoded instructions (one in the register renaming stage, and one in the dispatch stage). In addition, in the issue stage, we can only figure out the physical register number needed, instead of the actual value, and we have to wait for another cycle to read the value from the physical register file in the register file (RF) stage. The same problem applies to the write back (WB) and commit (CM) stages. After an instruction finishes execution, it will first write back its calculated value. We have to wait for at least one more cycle to commit this instruction.

Meanwhile, the Dispatch stage can be split. The re-order buffer in this stage can remain the same while the dispatch can be combined to the following issue stage.

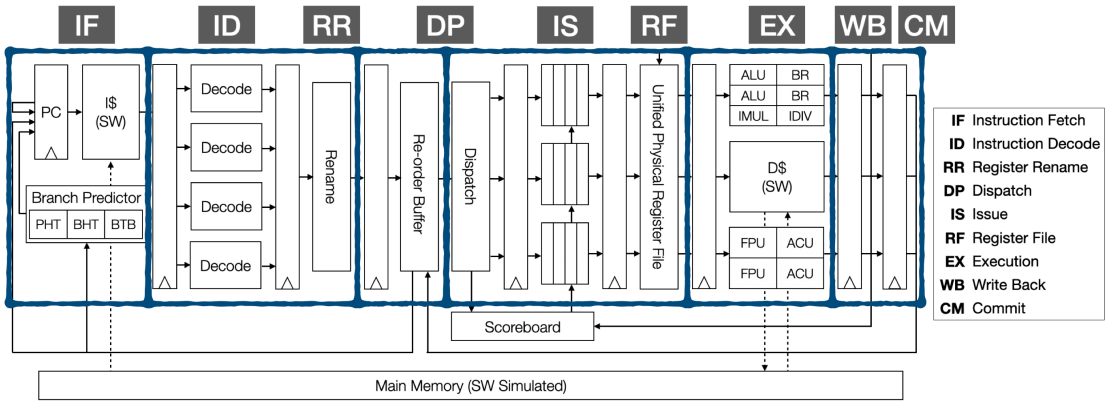


Illustration 8–2 Further optimized pipeline design.

8.2.3 Pipeline Optimization & Future Work

According to our discussion in the previous part, a tentative new pipeline division is shown in Illustration 8–2. The blue box represents the new six stages based on the old ten stages. Since it is just an optimization, it won't affect the correctness of the current design. Our current design is still synthesizable and functionally correct. In the future, we will apply those optimizations and analyze the performance gained from the changes.

9 Conclusion

In this project, we design a processor based on RISC-V ISA and simulate the SoC with software tools. Our processor supports 4-way superscalar execution and instruction dynamic scheduling, which keeps a good performance-energy-cost balance and involves complex algorithm and logic design. As the project is mainly for research and education purposes, it offers users a good base to explore computer architecture design and optimization.

Acknowledgements

Throughout the capstone design project of "RISC-V SoC Microarchitecture Design and optimization", we have received a great deal of support and assistance.

We greatly appreciate our instructor, Dr. Weikang Qian from UM-SJTU Joint Institute at Shanghai Jiao Tong University. As the instructor of our project, he listens to our weekly report and gives many constructive suggestions. Besides, he provides much support for us, not only in the technical part but also in terms of our future career planning, etc. His insightful feedback pushes us to sharpen our thinking and brings our work to a higher level.

We greatly thank Dr. Jigang Wu, Dr. Chengbin Ma, Dr. Peisen Huang, Dr. Gang Zheng, Dr. An Zou, and Dr. Mingjian Li from UM-SJTU Joint Institute at Shanghai Jiao Tong University, who are the course instructors of *VE450 Major Design Experience* at Joint Institute. We also greatly thank the whole teaching assistant team of this course. They teach us many concepts of the engineering design process and project management, which helps us explore the engineering problems and needs of the concept selection and final solutions.

In addition, we would like to express our greatest gratitude to our parents. No matter what kind of difficulties we are facing, they are always there, providing firm and unconditional support for us. We would like to thank our friends and schoolmates at Joint Institute for their support in our academic endeavors during the past four years.

Thanks to Ziqiao Ma and his capstone design project teammates from UM-SJTU Joint Institute for providing the LaTeX template for this thesis document.

Appendix A Experiment Result

A.1 Verilator and Spike

Table A–1 Measured experiment result from Verilator and Spike.

Mearsured Terms	Unit	Measured Results
Clock cycles used for verification samples	-	2266
Retired instructions	-	1363
Typical total cache size	KB	128 / ideal
Average response time to a request for service	ms	ideal
Number of I/O device types	-	3

A.2 Xilinx Vivado

Table A–2 Measured experiment result from Xilinx Vivado.

Mearsured Terms	Unit	Measured Results
Core frequency on FPGA test platform	MHz	74.88
Usage of look-up tables (LUT) on FPGA	k	101.7
Usage of block RAM (BRAM) on FPGA	-	0
Usage of digital signal processor (DSP) on FPGA	-	8
Power consumption on target FPGA test platform	W	2.63

Appendix B Bios

B.1 Li Shi

Major Electrical & Computer Engineering
Student ID 517370910032
SJTU Email shili2017@sjtu.edu.cn
Personal Email shili2048@gmail.com
Phone +86-14782059981

I am Li Shi, a senior undergraduate major in ECE at JI. I like studying photography, watching sci-fi movies, eating Chinese food (especially Sichuan cuisine), and sleeping. Also, I enjoy playing with kittens and puppies.

Before the capstone design project, I have been working on the High-Level Synthesis project instructed by Prof. Weikang Qian at JI for about 2 years, during which I accumulated some experience in digital design, software engineering, compiler design, etc. Meanwhile, I once worked as an intern at Apple, where I participated in developing a CPU simulator and studied computer architecture by myself. My interests include digital design, embedded systems, computer architecture, etc. In 2022, I will start my journey in M.S. in ECE program at Carnegie Mellon University.

B.2 Zhiyuan Liu

Major Electrical & Computer Engineering
Student ID 517370910240
SJTU Email angelinaliu@sjtu.edu.cn
Personal Email angelinaliuzy@gmail.com
Phone +86-13761505538

My name is Zhiyuan Liu, a senior student in JI major in ECE. I like drawing, photography, playing video games, and basketball. Besides, I like to assemble computers by myself. Before the capstone design project, I studied *VE270 Introduction to Logic Design* and *VE370 Intro to Computer Organization* in JI, and I also learned *EECS470 Computer Architecture* by myself. I am very interested in computer architecture. I once worked as an intern at Microsoft and I participated in feature development for products there. This fall, I will become a regular employee of Microsoft.

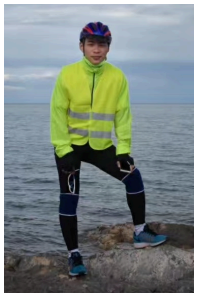
B.3 Yiqiu Sun

Major Electrical & Computer Engineering
Student ID 517370910020
SJTU Email susansun@sjtu.edu.cn
Personal Email susan.yiqiu.sun@gmail.com
Phone +86-18321098021

My name is Yiqiu Sun. I major in ECE at JI and CE at University of Michigan. Interested in Computer Architecture and new computing techniques like approximate computing, I worked with Prof. John Hayes on stochastic computing and with Prof. Trevor Mudge on re-configurable accelerators. Meanwhile, my course experience of EECS 470 Computer Architecture at Michigan would help me better contribute to this project. This fall, I will pursue my Ph.D. degree in computer science at University of Illinois at Urbana-Champaign. With a research focus on In-Memory Computing, I would like to figure out a better communication scheme between processor and memory in the future.

Besides academics, I am interested in cooking and hiking. I also adopted two cats. They are Gomez and Kent. These two fluffy creatures are my best life companions.

B.4 Jian Shi



Major Electrical & Computer Engineering
Student ID 517370910255
SJTU Email timeshi@sjtu.edu.cn
Personal Email shi965966503@hotmail.com
Phone +86-15000028420

I am Jian Shi, a senior student major in ECE at JI. I like taking photos, analyzing video games, and swimming. Also, I am a full-stack developer. I build and maintain an online server by myself. Starting from choosing hardware components in the server, to the daemon service written by myself, I learn a lot from this experience. Currently, the server has become my computing center. Whenever I have a task that can be processed or maintained by computer or network, I will write a program and throw it to my server. By the way, the test platform for this project is also based on my server.

This fall, I will start my Ph.D. degree at JI under Prof. Weikang Qian's guidance. I think this project will help a lot in my future Ph.D. study.

B.5 Yichao Yuan

Major	Electrical & Computer Engineering
Student ID	517370910233
SJTU Email	yyc19990826@sjtu.edu.cn
Personal Email	yichao.yuan.99@gmail.com
Phone	+86-13671502927

My name is Yichao Yuan. I am a senior student in JI major in ECE. I like movies, graphics, and various technology topics. I enjoy writing software and building hardware and the beauty of their interaction always amazes me. Before the capstone project, I have a solid knowledge of each abstraction layer of hardware. In JI, VE270, and VE370 introduce basics about logic design and computer organization. In 2020 spring, I took CS152 and EECS151 at UC, Berkeley, which gave me advanced knowledge in computing architecture and digital design.

This fall, I will be an ECE M.S. student at University of Michigan, Ann Arbor.

References

[1] LATIF K. Parallelism Via Concurrency at Multiple Levels[Z]. 2014. arXiv: 1406.0184 [cs.PF].

[2] VANCE A. Intel says Adios to Tejas and Jayhawk chips[EB/OL]. 2004. https://www.theregister.com/2004/05/07/intel_kills_tejas/.

[3] FULLER S H, MILLETT L I. The Future of Computing Performance: Game Over or Next Level?[M]. [S.l.]: National Academy Press, 2011.

[4] JOUPPI N P, YOUNG C, PATIL N, et al. A Domain-Specific Architecture for Deep Neural Networks[J/OL]. Commun. ACM, 2018, 61(9): 50-59. https://doi.org/10.1145/3154484. DOI: 10.1145/3154484.

[5] BOUCHARD Y. How Tesla is Using Artificial Intelligence and Big Data[EB/OL]. 2019. https://www.techiexpert.com/how-tesla-is-using-artificial-intelligence-and-big-data/.

[6] HU J, LI Z, YANG M, et al. A High-Accuracy Approximate Adder With Correct Sign Calculation[J/OL]. Integration, 2019, 65: 370-388. https://www.sciencedirect.com/science/article/pii/S0167926017302742. DOI: https://doi.org/10.1016/j.vlsi.2017.09.003.

[7] MRAZEK V, VASICEK Z, SEKANINA L, et al. ALWANN: Automatic Layer-Wise Approximation of Deep Neural Network Accelerators without Retraining[J/OL]. 2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD), 2019. http://dx.doi.org/10.1109/ICCAD45719.2019.8942068. DOI: 10.1109/iccad45719.2019.8942068.

[8] LI L, GAUTSCHI M, BENINI L. Approximate DIV and SQRT instructions for the RISC-V ISA: An efficiency vs. accuracy analysis[C]//2017 27th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS). [S.l. : s.n.], 2017: 1-8. DOI: 10.1109/PATMOS.2017.8106987.

[9] HENNESSY J L, PATTERSON D A. Computer Organization and Design MIPS Edition: The Hardware/Software Interface (Sixth Edition)[M]. [S.l.]: Elsevier Science, 2020.

[10] ASANOVIĆ K, et al. The Rocket Chip Generator[R/OL]. 2016. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[11] CELIO C, CHIU P F, NIKOLIC B, et al. BOOM v2: An Open-source Out-of-Order RISC-V Core[R/OL]. 2017. https://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-157.html.

[12] HU Z. Teach You How to Design a CPU Hand by Hand: RISC-V Processors[M]. [S.l.]: People's Posts, 2018.

[13] RISC-V International. RISC-V Exchange: Available Software[EB/OL]. 2021. https://riscv.org/exchange/software/.

[14] WATERMAN A, ASANOVIĆ K. The RISC-V Instruction Set Manual Volume I: User-Level ISA, Document Version 20191213[M/OL]. [S.l. : s.n.], 2019. github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf.

[15] DÖRFLINGER A, et al. A Comparative Survey of Open-Source Application-Class RISC-V Processor Implementations[C/OL]//CF '21: Proceedings of the 18th ACM International

Conference on Computing Frontiers. New York, NY, USA: Association for Computing Machinery, 2021: 12-20. doi.org/10.1145/3457388.3458657. DOI: 10.1145/3457388.3458657.

[16] PAGE D. Practical Introduction to Computer Architecture[M]. [S.l.]: Springer London, 2009.

[17] CHEN C, NOVICK G, SHIMANO K. RISC Architecture: RISC vs. CISC[R/OL]. 2015. https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/.

[18] STROUD C E, WANG L T, CHANG Y W. CHAPTER 1 - Introduction[M/OL]. Ed. by WANG L T, CHANG Y W, CHENG K T. Boston: Morgan Kaufmann, 2009: 1-38. https://www.sciencedirect.com/science/article/pii/B9780123743640500084. DOI: https://doi.org/10.1016/B978-0-12-374364-0.50008-4.

[19] MIPS Inc. MIPS® Architecture For Programmers[M/OL]. [S.l. : s.n.], 2014. https://www.mips.com/?do-download=introduction-to-the-mips32-architecture-v6-01.

[20] ARM Limited. ARM7 Technical Reference Manual[M/OL]. [S.l. : s.n.], 2021. https://developer.arm.com/documentation/ddi0210/c/.

上海交通大学 毕业设计（学士学位论文）
单独工作报告

SHANGHAI JIAO TONG UNIVERSITY

CAPSTONE DESIGN (BACHELOR'S THESIS)

INDIVIDUAL CONTRIBUTION REPORT

Student Name： 史历

Student ID Number： 517370910032

Major: 电子与计算机工程

Li Shi's Individual Contribution Report

In our capstone design of "RISC-V SoC Microarchitecture Design and Optimization", based on RISC-V instructions set architecture (ISA), we design a processor, apply some advanced optimizations, and set up the SoC in software simulator tools. During this process, each team members contribute to the processor design and optimization.

To design a processor based on RISC-V ISA requires the efforts from not only the hardware part, but also from the software part. In our team, Jian Shi, Yiqiu Sun and I are hardware engineers, while Yichao Yuan and Zhiyuan Liu are software engineers. We hold the group meetings twice a week and co-design the hardware circuit design and software debugging workflow to build our RISC-V processor. Each member contributes to the processor design equally, and here is each member's contribution to the project.

1. Jian Shi is responsible for designing the hardware frontend, including register renaming table, free list, re-order buffer. He also designs the floating-point units in the execution units in the backend. Besides, during the preparation stage of this project, he is partially responsible for the survey of open-source RISC-V core and SoC and reads some academic articles. His workload is about 15 hours per week.
2. Yiqiu Sun is responsible for designing and integrating the overall microarchitecture. She provides technical support for the team and designs the instruction fetch unit and branch predictor. Meanwhile, she helps other team members review the code, raises many constructive suggestions, and improves the overall design quality. Her workload is about 15 hours per week.
3. Yichao Yuan is responsible for software simulation and validation. He helps the team to validate the processor design in software design tools, including Xilinx Vivado, Verilator, Spike, etc. Besides, although not implemented in the final design, he explores the instruction and data cache design and AXI bus protocol for the SoC. His workload is about 15 hours per week.
4. Zhiyuan Liu is responsible for compilation workflow. She builds and modifies compiler toolchains, which is important for embedded system construction. She also adds our custom RISC-V instructions for approximate computing to the compiler so that users can use these custom instructions in their C programs. Her workload is about 15 hours per week.
5. I am responsible for hardware backend design, including instruction dispatch, issue units, scoreboard, physical register file and part of execution units. At the same time, I also participate in integrating the microarchitecture and debugging. My workload is about 15 hours per week.

As mentioned in the previous part, I am the hardware engineer in our team, and my job is mainly

concentrated on register transfer level (RTL) design and optimization.

In terms of the technical part, it is my second time to design a CPU core with hardware design language, and it is also the first time that I participate in designing a large processor core that supports superscalar execution and instruction dynamic scheduling. During this process, I have learned many basic concepts in computer architecture, digital design, and embedded systems. During the preparation stage, I investigate many open-source RISC-V cores and compare them in terms of performance, cost, and energy efficiency. After the project starts, I deep dive into the hardware design part.

The microarchitecture design part is one of the core parts in our project and requires many concept selections and trade-off in the detailed designs. For example, in terms of the design of instruction dispatch stage, we have two options: collapsing-style dispatch and non-collapsing-style dispatch. The former means that we need to compress the incoming instructions and send to the issue stage, while the latter just selects the instructions and marks the unnecessary instruction as invalid. After carefully comparing the two design styles, I choose to implement the first one, which is more complicated, but can reduce the logic complexity in the issue stage. Later in the issue and register file stage, we also have two options: whether we should first read from the register file and then pass the values to the issue stage, or first issue the instructions and then read from the register file. The former may lead to higher efficiency, but we need to store all the source operand values in the issue units and frequently pass these values. On the contrary, the latter can save such unnecessary operations, save some power, and fits the customer requirements in embedded systems. Thus, after this process of analysis and referring to other open-source RISC-V cores, I choose to implement the latter design. We need to carefully compare the designs and choose the best fit for our processor.

In terms of the engineering project management part, I also learn a lot during the process of this capstone project. We apply many useful project management tools, including Feishu docs, which is useful to draw Gantt chart and track each group member’s schedule, and Git, which helps the team to manage the project repository as well as review the source code. For example, in the Feishu docs page of “VE450 project management”, I record my progress and plan on a weekly basis, which will be automatically shown in the generated Gantt chart. In the group meeting, I report my weekly progress and share my hardware design or workflow with my team members. In the Git repository, I create my own working branches, including `Li/IssueQueue`, `Li/Dispatch`, `Li/ExecutionUnit`, etc., so that other people can easily check my progress and personal contribution to the project.

In terms of the technical communication part, I actively participate in the preparation for each design review and final expo. In design review 1, I am responsible for present our customer requirements and engineering specifications. In design review 2, I demonstrate our first version of processor design.

In design review 3, I present our final microarchitecture design which consists of 9 pipeline stages in the frontend and backend. In the final expo, I participate in the oral defense and JI design expo. I try my best to not only work hard on the project itself, but also present our project to the audience so that people can clearly understand what we have done and how they can utilize our processor to do great things.

In brief, as one of the hardware engineers in the team, I actively contribute to the project of "RISC-V SoC Microarchitecture Design and Optimization". We work as a team and design a complicated RISC-V processor.

上海交通大学 毕业设计（学士学位论文）
单独工作报告

SHANGHAI JIAO TONG UNIVERSITY

CAPSTONE DESIGN (BACHELOR'S THESIS)

INDIVIDUAL CONTRIBUTION REPORT

Student Name： 刘之远

Student ID Number： 517370910240

Major: 电子与计算机工程

Zhiyuan Liu's Individual Contribution Report

Our capstone design topic is "RISC-V SoC Microarchitecture Design and Optimization" and in this project, we propose a processor design that can support 4-way superscalar execution and instruction dynamic scheduling together with some advanced optimizations like approximate floating-point computing units to solve challenges in AI-oriented embedded system which requires the CPU to be energy-efficient, inexpensive and fast at same time. After designing our processor, we also set up the SoC and peripheral components like cache and I/O devices in software simulator tools to test and validate our design. Our project can be divided into hardware part and software part. Li Shi, Jian Shi and Yiqiu Sun are mainly hardware engineers in our team while Yichao Yuan and I are software engineers. We hold our group meetings two to three times a week and there are also some individual meetings. Besides, we have meetings with our instructor every two weeks to report our project progress and explain some technical issues. Everyone has done their best for our design and everyone has contributed to our project equally, the following is our division of work.

1. Li Shi: He is responsible for most of backend designs like instruction dispatch, instruction issue, register file and execution units for integer and memory access. He also helps debug and integrate FP components. His workload is about 15 hours per week.
2. Jian Shi: He is responsible for hardware frontend designs such as free list, register renaming table, reorder buffer and execution units for floating point. At the same time, during the preparation phase of our project, he also did a lot of research on other RISC-V cores. His workload is about 15 hours per week.
3. Yiqiu Sun: She is responsible for designing the branch predictor and instruction fetch unit and helps integrate and design our overall microarchitecture. She is also our technical support and helps review our code and proposes many constructive ideas about our design. Her workload is about 15 hours per week.
4. Yichao Yuan: He is responsible for software simulation and validation part. He helps to replicate the Spike-based model to our verilator-based model, so that our CPU core can be better compared with the Spike model, and help simulation and validation. He also explores cache design and AXI bus protocol for our project at the preparation stage. His workload is about 15 hours per week.
5. Zhiyuan Liu: I am responsible for the compilation workflow and I also do parts of the instruction fetch at the beginning of our project with the help of Yiqiu Sun. I investigate and study the structure of LLVM compiler and GCC compiler and try to modify and rebuild the compiler toolchains such that customized instructions for approximate computing units can be issued and disassembled by our new compiler toolchains. I also integrate the approximate computing functions into Spike to help validate our core. My workload is about 15 hours per week.

As mentioned in the previous part, I am the software engineer in our team, and my job is mainly concentrated on compilation workflow and Spike validation part. But at the beginning of the project, I am also responsible for part of the hardware frontend design - instruction fetch. I have had the experience of RTL design, but I have not been in touch with SystemVerilog before, so it is painful when I first start writing SystemVerilog for our project. In the beginning, I just treat it purely as an ordinary software language, but in the process of writing and with the help of our team members, I can now roughly understand how to use SystemVerilog to describe the behavior of the hardware. With the help of Yiqiu Sun, we complete the instruction fetch part. In the middle and late part of our project, I focus on the compilation workflow. In this process, I compare the architecture of the LLVM compiler with that of GCC, and get an in-depth understanding of the compilation process. Between LLVM and GCC compiler, after comparing their structure, I select GCC compiler. Among many versions of RISC-V GCC compiler, I finally choose riscv64-multilib-elf-gcc as our target compiler because the design goal of our CPU is for embedded system, so we must use static link library, and we want our design to be better compatible with 64-bit RISC-V architecture in the future. At the same time, I also study how to modify and rebuild the compiler so that it can issue and disassemble our customized instructions for approximate computing, how to add our customized instruction to Spike, and how to describe the functional behavior of our customized instruction so that Spike can also run approximate computation for floating point.

I also learn a lot about how to manage engineering project in our capstone design.

Many useful project management tools are used in our project such as Feishu and Git. We use Feishu to arrange our group meetings and use Feishu docs to track each team member’s progress and schedule. We use Git to manage our code. For example, we have a Feishu docs named “VE450 project management”. In that docs, we record the division of work for everyone and update our progress in time.

In terms of the technical communication part, I actively participate in each presentation and our final expo. In our presentations, I am responsible for introducing the background of our project to the audience, the actual problems that our project solved, and the innovations of our project. I also make a technical report and presentation on compilation workflow to our instructor.

In conclusion, as one of the software engineers in our team, I try my best to contribute to our capstone design. We work as a team and everyone in our group tries their best to make the project better.

上海交通大学 毕业设计（学士学位论文）
单独工作报告

SHANGHAI JIAO TONG UNIVERSITY

CAPSTONE DESIGN (BACHELOR'S THESIS)

INDIVIDUAL CONTRIBUTION REPORT

Student Name： 孙逸秋

Student ID Number： 517370910020

Major: 电子与计算机工程

Yiqiu Sun's Individual Contribution Report

Our capstone design is "RISC-V SoC Microarchitecture Design and Optimization". We design a processor based on RISC-V instructions set architecture (ISA). We simulate and verify the SoC in software simulator tools and further apply some advanced optimizations. This is a project that involves both hardware and software. All team members contribute to the processor design and optimization based on their expertise.

In our team, Jian Shi, Li Shi and I are hardware engineers, while Yichao Yuan and Zhiyuan Liu are software engineers. We hold the group meetings twice a week and co-design the hardware circuit design and software debugging workflow to build our RISC-V processor. Each member contributes to the processor design equally, and here is each member' s contribution to the project.

1. During the preparation stage of this project, Jian Shi is partially responsible for the literature review for open-source RISC-V core and SoC. Jian Shi is responsible for designing the core hardware for Out-of-order pipeline, including register renaming table, free list, re-order buffer. He also works actively on the design of floating-point units (both accurate and approximate) in the execution stage. His workload is about 15 hours per week.
2. Yichao Yuan is responsible for software simulation and validation. He is in charge of validating the processor design on various software design tools, such Xilinx Vivado, Verilator, Spike, etc. For every verification tool, he always designs a detailed plan. He also explores the instruction and data cache design and AXI bus protocol for the SoC although it is not implemented in the final design due to time. His workload is about 15 hours per week.
3. Zhiyuan Liu is responsible for compilers. She builds and modifies compiler toolchains, which is important for embedded system construction. She also customizes the compiler by adding a new RISC-V instruction for approximate computing so that users can use these custom instructions in their C programs. Her workload is about 15 hours per week.
4. Li Shi is responsible for hardware backend design. This includes instruction dispatch, issue units, scoreboard, physical register file and part of execution units. At the same time, he also actively participates in microarchitecture integration and debugging. His workload is about 15 hours per week.
5. I am responsible for designing and integrating the overall microarchitecture. Meanwhile, I provide technical support for the team and design the instruction fetch module, fetch buffer module and branch predictor module. Besides that, I implement the int multiplier and divider for the execution stage. I also help other team members review the code by raising many constructive suggestions thus, improving the overall design quality. My workload is about 15 hours per week.

Throughout this project, I am mainly in charge of the hardware. Before the start of the project, I already know the whole project is very challenging because I have taken EECS 470: Computer Architecture in University of Michigan, which is a similar but less interesting work than this capstone design. My previous course experience does help me better understand the goal and smoothen the concept selection process. But I have to be honest that the benefits are limited as our capstone design is more professional and advanced. For example, although I have designed a branch predictor before, when I am designing the new branch predictor for this capstone, I find new optimization points and there are still many new sophisticated details for me to consider. I keep reminding myself that "Practice makes perfect" and "Devils are in the details". Only by continuously optimizing the microarchitecture details can we achieve an overall best performance. Besides interacting with modules that I am familiar with, I also learn some new designs such as approximate computing. Those new designs are a perfect example of what engineering is. Engineering is not about finding a universally accepted truth, but about striking a balance between different considerations and applying optimizations in a specific scenario.

In terms of the technical communication part, I actively participate in the preparation for each design review and final expo. In design review 1, I am responsible for setting the timeline and overall workflow. In design review 2, I partially discuss our concept selection process. In design review 3, I present our final implement plan. In my bi-week meeting with advisor, I give a presentation about the branch predictor that we built. I also actively participate in our promotional video making process. I will participate in the oral defense and JI design expo in the final expo. In general, my goal is to not only design a high-performance processor, but also make more people understand what we have done and how they can utilize our processor to do great things in different fields. We also would like to inspire more and more young people to join us in the research of computer architecture.

This project also familiarizes me with various software that are necessary to remote-working. Due to COVID-19, I decide to stay in the US and participate the project remotely. I manage to pull through many difficulties that come with this choice, including time-zone difference, poor internet connection and schedule conflict.

I also learned a lot from my teammates. They are both professional and respectful. We meet frequently and have many meaningful debates. I always enjoy hearing different voices and discussing the pros and cons with them. We have a great atmosphere and strong team spirits.

In brief, as one of the hardware engineers in the team, I actively contribute to the project of "RISC-V SoC Microarchitecture Design and Optimization". We work as a team and design a complicated RISC-V processor. Although there is definitely some room to improve in the future, we have done our best to design and verify in such a short time.

上海交通大学 毕业设计（学士学位论文）
单独工作报告

SHANGHAI JIAO TONG UNIVERSITY

CAPSTONE DESIGN (BACHELOR'S THESIS)

INDIVIDUAL CONTRIBUTION REPORT

Student Name： 时尖

Student ID Number： 517370910255

Major: 电子与计算机工程

Jian Shi's Individual Contribution Report

Our capstone project is about "RISC-V SoC Microarchitecture Design and Optimization". In this project, we first design an out-of-order RISC-V processor supporting 4-way superscalar execution and instruction dynamic scheduling. What's more, we also try to optimize the SoC with approximate computing units. During the project, all team members make their own contribution to our design and we are satisfied about our final product.

To design and optimize a processor based on RISC-V ISA, we need a lot of efforts, from the hardware part, to the software part. In our group, Li Shi, Yiqiu Sun and I are responsible for hardware engineering, while Yichao Yuan and Zhiyuan Liu are responsible for software engineering. Every week, we hold the group meetings twice, on Wednesday and Saturday. The software part and the hardware part work together to build our own RISC-V processor. We roughly make the same contribution to the processor design. Each member's contribution to the project is as follows.

1. Li Shi focuses on hardware backend design, including instruction dispatch, issue units, scoreboard, physical register file and part of execution units. On the other hand, he also takes part in integrating the microarchitecture and debugging. His workload is roughly 15 hours per week.
2. Yiqiu Sun focuses on designing and integrating the overall microarchitecture. She provides technical support for the team and designs the instruction fetch unit and branch predictor. What's more, she helps other team members review the code, raises many constructive suggestions, and improves the overall design quality. Her workload is roughly 15 hours per week.
3. Yichao Yuan focuses on software simulation and validation. He helps the team to validate the processor design in software design tools, including Xilinx Vivado, Verilator, Spike, etc. Besides, although not implemented in the final design, he explores the instruction and data cache design and AXI bus protocol for the SoC. His workload is roughly 15 hours per week.
4. Zhiyuan Liu is focuses on compilation workflow. She builds and modifies compiler toolchains, which is important for embedded system construction. She also adds our custom RISC-V instructions for approximate computing to the compiler so that users can use these custom instructions in their C programs. Her workload is roughly 15 hours per week.
5. I focus on designing the hardware frontend, including register renaming table, free list, re-order buffer. I also design the floating-point units in the execution units in the backend. Besides, during the preparation stage of this project, I am partially responsible for the survey of open-source RISC-V core and SoC and read some academic articles. My workload is roughly 15 hours per week.

As previously mentioned, I am the hardware engineer in our team, and my job mainly focuses on register transfer level (RTL) design and optimization.

In terms of the technical part, it is my first time to design and test a System-on Chip (SoC), which means that I need to learn a lot of concepts in computer architecture, embedded systems and logic circuit design. On the other hand, in my four-year university study, I only have experience in a five-stage MIPS CPU design with in-order execution using Verilog language. Therefore, it is also my first time to take part in design for a processor core that supports execution and instruction dynamic scheduling with SystemVerilog language. For this reason, I studied a lot in digital design, and circuit simplification. In the evaluation stage, I made a survey for open-source RISC-V cores and SoC. Besides, to find design suitable for our SoC, I read many academic articles about approximate computing, floating point units and FPGA architecture.

One of the main parts in our project is the microarchitecture design, which requires us to select between concepts and make a balance in many aspects. For instance, the register renaming table needs to be recovered when mis-prediction or exceptions happen. However, there exist two structures to recover it accurately: checking point and retirement rename allocation table (rRAT). The former means that the processor should check the table when a branch instruction is renamed, while the latter just update the rRAT when an instruction is retired. In comparison, the method based on checking point requires a larger circuit but can provide faster recover speed. We first implement the first one in our processor. However, the final verification results show that a larger circuit leads to timing violation in FPGA implement. Therefore, I replace it with the rRAT one.

In terms of the technical support part, I also learn a lot from this project. We use my central server for circuit verification and implementation. The central server is much more powerful than our personal computers and can handle many computing tasks, such as compiling and logic synthesis. I am responsible for maintaining the server and provide in-time technique support. For example, I deploy the virtual network console (VNC) on the server so that we will share the same graphical user interface (GUI) during the project. Besides, I also teach my group mates to use Xilinx Vivado as a tool for logic synthesis and implementation. To share and manage our source code, we use Git and create an organization on GitHub. In `UMJI-VE450-21SU/Ria` repository on GitHub, I have my own working branches, including `Jian/Rename`, `Jian/ROB`, `Jian/FloatingPoint`, etc.

In terms of the technical communication part, we roughly have the same workload for each design review, final report and final expo. In design review 1, I do the literature search and take them as a benchmark for our design. In design review 2, I introduce the concept of instruction set architecture (ISA) and make a comparison between ARMv7 ISA and RISC-V 32G ISA. In design review 3, I compare our measured value with the target value in engineering specification. I also point out

our design oversights in the engineering specification part. In the final expo, I take part in the oral presentation and JI design expo. Besides, I am responsible for the promotion video cutting. I devote myself to present an easy-to-understand and amazing project to the audience so that they can be inspired by our design and give us useful response for further improvement.

All in all, as a hardware engineer in this project, I have exploited my potential and make my own contribution to the project of "RISC-V SoC Microarchitecture Design and Optimization". We cooperate as a team and design a complex processor with RISC-V ISA.

上海交通大学 毕业设计（学士学位论文）
单独工作报告

SHANGHAI JIAO TONG UNIVERSITY

CAPSTONE DESIGN (BACHELOR'S THESIS)

INDIVIDUAL CONTRIBUTION REPORT

Student Name： 袁意超

Student ID Number： 517370910233

Major: 电子与计算机工程

Yichao Yuan's Individual Contribution Report

Our capstone design is "RISC-V SoC Microarchitecture Design and Optimization". In this capstone project, we designed a CPU based on RISC-V instruction set architecture (ISA). We investigate and use some advanced optimization techniques and build a software environment for verification and testing. During this process, each team members contribute to the processor design and optimization.

Designing a processor based on RISC-V ISA needs joint efforts from both the hardware part, which is about the design itself, and the software part, which helps verification and testing. In our team, Li Shi, Yiqiu Sun and I are hardware engineers, while Yichao Yuan and Zhiyuan Liu are software engineers. We hold the group meetings twice a week and co-design the hardware circuit design and software debugging workflow to build our RISC-V processor. Each member contributes to the processor design equally, and here is each member' s contribution to the project.

1. Li Shi is responsible for hardware backend design, including instruction dispatch, issue units, scoreboard, physical register file and part of execution units. At the same time, he also participates in integrating the microarchitecture and debugging. His workload is about 15 hours per week.
2. Yiqiu Sun is responsible for designing and integrating the overall microarchitecture. She provides technical support for the team and designs the instruction fetch unit and branch predictor. Meanwhile, she helps other team members review the code, raises many constructive suggestions, and improves the overall design quality. Her workload is about 15 hours per week.
3. Jian Shi is responsible for designing the hardware frontend, including register renaming table, free list, re-order buffer. He also designs the floating-point units in the execution units in the backend. Besides, during the preparation stage of this project, He is partially responsible for the survey of open-source RISC-V core and SoC and read some academic articles. His workload is about 15 hours per week.
4. Zhiyuan Liu is responsible for compilation workflow. She builds and modifies compiler toolchains, which is important for embedded system construction. She also adds our custom RISC-V instructions for approximate computing to the compiler so that users can use these custom instructions in their C programs. Her workload is about 15 hours per week.
5. I am responsible for software simulation and validation. I help the team to validate the processor design in software design tools, including Xilinx Vivado, Verilator, Spike, etc. Besides, although not implemented in the final design, I explore the instruction and data cache design and AXI bus protocol for the SoC. His workload is about 15 hours per week.

As previously mentioned, I am the software engineer in our team, and my job mainly focuses on

building simulation and validation environment.

In terms of the technical part, it is my first time to get involved in the design of such a large digital system, and it is also the first time I participate in a design that needs effort from software level, architectural level and hardware level. During my undergraduate study, I have experience in all these levels, however, bringing them together and build a system from scratch is still challenging. I review a lot of concepts I learnt before and study in-depth about simulators and FPGA. During the first half of this project, I investigate building memory system with FPGAs. During the second half of this project, I focus on building an integrated simulation and validation system to verify the hardware design.

How to provide an environment for the computation core is an important point to considered in this system. The computation core needs at least some memory systems and IO system to be tested and verified. As this project aims at doing microarchitecture optimization, two domains can be considered: hardware implementation and software simulation. The former part provides very good performance and more solid result; However, it demands resources and provides little flexibility. In comparison, performing a software simulation is easy to implement, require nearly no resources and can be easily configured. I investigate the hardware implementation option first. However, the synthesize result shows it occupies too much resource and it is the bottleneck of the system. To better serve the overall goal of optimizing microarchitecture, I then turn to the software simulation path, and the system ties the software verification environment with the computation core successfully. I build the verification environment’s frontend server, which provides memory interfaces and IO interfaces, by building our own model as well as adapting part of the source code from the Spike simulator. I tune the compilation process so that the generated executable can be correctly loaded and executed by the models. I write libraries for this simulation system and implement some programs for test and demonstration using these libraries. I help organize part of the overall flow by writing makefiles. The final system boosts the verification cycle and demonstrates positive result of our CPU design.

In terms of the technique support part, I also learn a lot from this project. The project is organized by both Feishu and Git. The Feishu tracks members progress and help resolved the dependency among tasks. I report my progress on this system steadily to make sure I can cooperate well with my teammates. The Git holds repository of the codes of this project. During the first half of this project, I mainly working on the branch `Yuan/memory`, working on implementing the memory subsystem of this project on FPGA. After we turn to software simulation, we start to working on `Test/T1` and `Test/verilator` branch to work out the software stack of this project. This method makes code review possible and enhance cooperation.

In terms of the technical communication part, I participate in each design review, final report and final expo. In the design review 1, I am responsible for reviewing the problem of this project. In the design review 2, I am responsible for conducting the concept selection between static scheduling and dynamic scheduling. In the design review 3, I present the demo of the project. In the final expo, I take part in the oral presentation and JI design expo. I actively prepare the technical communication so that we can introduce our audience with this amazing project, and make sure that they can understand our design with ease as well as provide us valuable responses.

All in all, as a software engineer in this project, I contribute to the project of "RISC-V SoC Microarchitecture Design and Optimization" actively a lot and benefit from this experience a lot. As a team, we cooperate together and design a complex RISC-V processor with optimization.