RVSingle: A general purpose power efficient RISC-V for FPGAs

YuYang Shen
Master of Science Thesis in Electrical Engineering

RVSingle: A general purpose power efficient RISC-V for FPGAs

YuYang Shen
LiTH-ISY-EX--23/5596--SE

Supervisor: Kent Palmkvist
ISY, Linköpings universitet

Examiner: Mark Vesterbacka
ISY, Linköpings universitet

Division of Integrated Circuits and Systems
Department of Electrical Engineering
Linköping University
SE-581 83 Linköping, Sweden

Copyright © 2023 YuYang Shen
Abstract

With the increasing need for low-cost, power-efficient computing units, RISC-V as an open-standard Instruction Set Architecture (ISA) is becoming more and more popular in the industry. There are multiple open-source RISC-V soft processors like cva6, VEGA, NOEL-V and more. But those processors have a common problem in that they can only be implemented onto a specific FPGA development platform. This thesis introduces a new processor design with compatibility in mind so that it will not be limited to a certain development platform but can be used on multiple different platforms as far as they meet the basic requirements. This processor is a single-stage processor without any pipeline implemented. The processor is used to evaluate the power efficiency of the architecture and has a unique feature to enable or disable the RISC-V Compressed (RVC) instruction subset to understand its impact on power-efficient. It is simple in architecture but still has the full capability for the RV64IC instruction set. Because of it uses RISC-V architecture, in the future, this processor can be easily expanded to adopt more RISC-V instruction subsets.
Acknowledgments

I would like to express my gratitude to all those who have provided their assistance and support, enabling the successful completion of my Master's Thesis project.

First of all, I would like to express my deep appreciation to my examiner and supervisor, Mark Vesterbacka and Kent Palmkvist. Their continuous support, encouragement, and invaluable professional insights were instrumental in the successful realization of my project. Their consistent guidance and frequent communication transformed the challenges of this project into an enjoyable and rewarding experience.

Equally, I am indebted to my fellow Master's students, whose assistance and companionship proved invaluable throughout this journey. They provided not only academic support but also much-needed camaraderie, helping me maintain focus on my research and persevere through periods of doubt and difficulty. Their contributions may not be directly evident in the research, but they have been undoubtedly indispensable.

Last but not least, I would express my sincere gratitude to my family members. Without their firm support, constant encouragement and love, my study and research wouldn't be possible. They are the ones who made this come true and enabled me to do my Master's here.

Once again, I would love to thank everyone who helped me complete this research. This journey, enriched by your support and assistance, has been invaluable to me and will forever remain a cherished part of my memories.

Linköping, June 2023
YuYang Shen
# Contents

Notation ix

1 Introduction 1
  1.1 Motivation .................................................. 1
  1.2 Aim ......................................................... 2
  1.3 Research questions ........................................ 2
  1.4 Delimitations .............................................. 3

2 Theory 5
  2.1 RISC-V ...................................................... 5
    2.1.1 An overview of RISC-V architecture ................. 5
    2.1.2 The RV64 Extension .................................. 6
    2.1.3 The Compressed Extension ........................... 7
    2.1.4 Instruction Format .................................. 8
  2.2 RISC-V Toolchain ......................................... 10
    2.2.1 RISC-V GNU Compiler Toolchain ..................... 10
    2.2.2 RISC-V Proxy Kernel ................................ 11
    2.2.3 Spike .................................................. 12
    2.2.4 Ripes ................................................ 12
  2.3 Hardware .................................................. 14

3 Method 17
  3.1 Pre-Study ............................................... 17
  3.2 Implementation .......................................... 18
    3.2.1 Instruction Fetch .................................... 18
    3.2.2 Decoder ............................................... 18
    3.2.3 CSG .................................................... 19
    3.2.4 ALU ................................................... 20
    3.2.5 Memory and RF ....................................... 20
    3.2.6 I/O ..................................................... 20
  3.3 Verification .............................................. 21
    3.3.1 Software Verification ................................. 22
    3.3.2 Hardware Verification ................................. 22
### Abbreviations

<table>
<thead>
<tr>
<th>Shortening</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computer</td>
</tr>
<tr>
<td>CISC</td>
<td>Complex Instruction Set Computer</td>
</tr>
<tr>
<td>ISA</td>
<td>Instruction Set Architecture</td>
</tr>
<tr>
<td>MCU</td>
<td>Micro-Controller Units</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Units</td>
</tr>
<tr>
<td>IOT</td>
<td>Internet of Things</td>
</tr>
<tr>
<td>OS</td>
<td>Operation System</td>
</tr>
<tr>
<td>VM</td>
<td>Virtual Machines</td>
</tr>
<tr>
<td>PFC</td>
<td>Program Flow Controller</td>
</tr>
<tr>
<td>PC</td>
<td>Program Counter</td>
</tr>
<tr>
<td>ELF</td>
<td>Executable and Linkable Format</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>VSRTL</td>
<td>Visual Simulation of Register Transfer Logic</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Arrays</td>
</tr>
<tr>
<td>IC</td>
<td>Integrated Circuits</td>
</tr>
<tr>
<td>LUT</td>
<td>Look-Up-Table</td>
</tr>
<tr>
<td>FF</td>
<td>Flip-Flop</td>
</tr>
<tr>
<td>BLE</td>
<td>Basic Logic Element</td>
</tr>
<tr>
<td>ALE</td>
<td>Adaptive Logic Element</td>
</tr>
<tr>
<td>IOB</td>
<td>Input/Output Block</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Progress</td>
</tr>
<tr>
<td>FIR</td>
<td>Finite Impulse Response</td>
</tr>
<tr>
<td>FFT</td>
<td>Fast Fourier Transform</td>
</tr>
<tr>
<td>ALU</td>
<td>Arithmetic Logic Unit</td>
</tr>
<tr>
<td>MACC</td>
<td>Multiply and Accumulate</td>
</tr>
<tr>
<td>PLL</td>
<td>Phase-Locked Loop</td>
</tr>
<tr>
<td>DLL</td>
<td>Delay-Locked Loop</td>
</tr>
<tr>
<td>VCO</td>
<td>Voltage-Controlled Oscillator</td>
</tr>
<tr>
<td>JTAG</td>
<td>Joint Test Action Group</td>
</tr>
<tr>
<td>HDL</td>
<td>Hardware Description Language</td>
</tr>
<tr>
<td>IM</td>
<td>Instruction Memory</td>
</tr>
<tr>
<td>CSG</td>
<td>Control Signal Generator</td>
</tr>
<tr>
<td>DM</td>
<td>Data Memory</td>
</tr>
<tr>
<td>RF</td>
<td>Register File</td>
</tr>
<tr>
<td>GPIO</td>
<td>General Purpose Input and Output</td>
</tr>
<tr>
<td>MSB</td>
<td>Most Significant Bits</td>
</tr>
<tr>
<td>IP</td>
<td>Intellectual Property</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Design Environment</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 Motivation

Micro-Controller Units (MCU) and Central Processing Units (CPU) are the heart of modern electronics and Industrial automation. Not only do they play a big role in modern devices, but they also are the key factor for how powerful devices are. The development in technology makes them smaller and more efficient which makes it possible to fit them into almost everything. When those devices connect to the internet to let people be able to remotely control them, they are called the Internet of Things (IoT). They make people’s life smarter and easier [1]. The thriving of the IoT industry not only enlarges the need for MCU or even CPU but also makes engineers need to take power efficiency very seriously as those devices tend to be powered by limited power sources like button batteries or even use energy harvesting as the power source for the device [2]. Besides, most IoT devices have the Always-On characteristic which makes power-efficiency becomes crucial for energy saving and green energy [3].

To achieve low power consumption with reasonable performance, Reduced Instruction Set Computer (RISC) architecture is often used instead of Complex Instruction Set Computer (CISC) architecture [4]. However most of these ISA in RISC are closed-source and require a license to use them. Not only is the license fee quite high but also if a developer needs to further improve the architecture they need to pay more for that or they may not even be possible [5]. IoT devices normally only execute a limited set of operations so to maximise their power efficiency, the core architecture needs to be modified to meet the requirement. Besides that as those devices often have similar operations or even a fixed data path, having custom instructions to accelerate those operations can not only enhance the performance but also be more power efficient. In order to easily study and customise the architecture or extend the ISA, a modern open-source ISA will
be needed for that purpose.

RISC-V is a relatively new and fast-developing ISA. It is developed at the University of California, Berkeley. It has several attractive features. One of them is its openness, it is open-sourced which means no need for a license to use it. Not only that, thanks to its modular design, developers can easily custom and improve the instruction set and architecture as they wish. It also easy to implement it to an FPGA to verification and RISC-V has a full test and verification suite for developers to use. Besides hardware, RISC-V also has GCC cross compiler for software engineers to develop software without the need to care about how the real hardware work. In the meantime, with those key features, the RISC-V ISA also has comparable performance compared to X86 or ARM [6]. The RISC-V has 32-bit and 64-bit sub-ISA, but most open-source cores are built upon the 32-bits architecture, while the current 64-bits cores are more close-sourced and for commercialisation.

One more obstacle for using the current available open-source RISC-V cores like CVA-6[7], RiscyOO[8] and NutShell all have limited FPGA development board support. That to say, if someone wants to implement these cores to an FPGA board and run it in real hardware, his/her options are limited to what the project provides. That makes it more difficult and perhaps more expensive for one to test and evaluate these RISC-V cores. Thus having a general-purpose power-efficient 64-bit RISC-V core for FPGAs will be beneficial to developers.

1.2 Aim

This thesis aims to propose a general-purpose RISC-V core with 64-bit architecture and RV64IC instruction set for FPGAs. Without any dedicated hardware or 3rd party IP needed, this makes the developer having a modern core easy to test, evaluate and even improve or customize this core for special usage.

1.3 Research questions

This is the section where the research questions are listed. Those questions will be answered later in this report.

1. How to design a RISC-V core without the need for any dedicated hardware or 3rd party IPs?

2. Can the designed core fit different FPGA development boards? Even from different manufacturers like Intel and AMD.

3. How does the Compressed instruction set (the "C" instruction set in RV64IC) affect the power efficiency? How much difference in power consumption with and without it?
1.4 Delimitations

This paper mainly focuses on the RV64IC instruction set. As the RISC-V ISA has the modular feature this means the final design will only support the 32-bit and 64-bit "Integer" instruction set and "Compressed" instruction set. This still meets the RISC-V standard as the "Integer" instruction set which is the only required instruction set. However, this means the final core won't be able to run any desktop Operation System (OS). This paper isn't aiming to compare the performance and efficiency of desktop-level software and will focus more on MCU-level comparison. Also, the instruction set will be limited to the unprivileged level and its specification as we won't run any OS and don't need the privileged level functions.

The final design doesn't contain any pipeline, which is for simplifying the architecture and also minimizes the need for FPGA resources. Besides, this paper assumes the developer has a way to load the program into the instruction memory. The final design will always execute from the instruction memory address 0x0000000000000000. For compatibility reasons, the instruction memory is set default to 64 KB, and the data memory is set to 32 KB. However, those sizes can be tweaked by setting parameters. The actual availability of space for those memories may be limited because of the FPGA development board used.

Due to the limitation of the available hardware, this paper will only verify the design with Intel and AMD FPGA development boards or platforms. It is easy for viewers to verify/adopt the design onto their own FPGA board.

For easier comparison, the results and discussion section will only focus on the Intel platform which in this case is the DE-10 nano development board. The other platforms, like AMD boards, shows a similar pattern of those results and outcomes.
2.1 RISC-V

The RISC-V is a relatively new ISA designed initially mainly for educational usage, but later, people found out it has the potential to be used in other applications. With the establishment of the RISC-V Foundation, to become an open-source, license-free and open architecture for industrial usage. RISC-V is actually a set of ISAs. It does not only includes 32-bit architecture but also supports the compressed 16-bit instructions and extended instructions for 64-bit and, in the future, will have the 128-bit architecture as an official standard. Starting from the beginning, direct implementation on hardware is supported, which means it is not designed to be just a simulation or binary compilation [9].

For beginning a newcomer ISA, RISC-V used a lot from other ISAs. First, its standard is designed to be simple to prevent giving new users a sharp learning curve. But developers still have the ability to introduce complex architectures or features like Out-Of-Order(OoO), pipeline or decoupled. It is also not designed to be implemented with particular hardware technology like FPGA, ASIC or type-out ICs [9].

2.1.1 An overview of RISC-V architecture

As mentioned above, the RISC-V is a set of bundled ISAs. It is defined as a base integer ISA which must be present in the design, along with some optional extensions for extended usage. There are mainly two officially standardized integer base ISAs: the RV32I and the RV64I. The 128-bit base integer vision is still in the draft version, so it doesn’t count as standardized yet. The base integer ISA adopted a lot of good parts from other ISA:s and learned a lot of lessons from them also. But they don’t have branch delay slots and can decode instructions with variable lengths, making them more flexible for encoding instructions. In
addition, base ISAs are carefully designed so that only a minimal set of instructions is required to provide a reasonable target for compilers, assemblers and linkers. That minimalist design also makes the RISC-V ISA a convenient one to build a toolchain framework and customization the ISA as developers need [9].

To allow quick extension of the RISC-V functionality, there are also multiple extensions for designers to choose from. For example, the "Zifencei" extension supports Instruction-Fetch Fence, the "M" standard adds support for Integer Multiplication and Division, the "F" standard extension allows users to do single-precision floating and the "D" extension for double-precision [9].

There is a bundled ISA suite called the "G" extension. It represents the combination of a base integer ISA (RV32I or RV64I) and several selected standard extensions (IMAFD, Zicsr and Zifencei) together as a "general-purpose" ISA. This is the minimum requirement for a RISC-V core to run any desktop OS. In addition, this bundle of ISA can sometimes also have the Compressed ("C") extension for better performance and efficiency. Besides this ISA suite, other extensions, such as multimedia or encryption and decryption, will be more application-specific [9].

There are two different specifications of the RISC-V ISA, the unprivileged specification [9] and the privileged specification [10]. The privileged version is more for complex OS-level usage like supervisor and hypervisor for Virtual Machines (VM). It covers things like Machine-Level ISA, supervisor-level ISA and Hypervisor extension. In this research, we will only use the unprivileged level instructions [10].

Besides those standardized ISAs, there are also some newly suggested or work-in-progress ISAs called draft ISAs. The RV128I is one of them. Those ISAs haven't been "frozen" yet. That means they are not their final version and can change in the future. Developers can design their core with those extensions but must remember that the specification can change later and might make their product incompatible. Therefore, the design in this research only uses the ratified instruction set and extensions.

2.1.2 The RV64 Extension

The RV32I and RV64I architectures are characterized by the width of their integer register and their related address space size by the number of those integer registers. That means RV32I and RV64I have 32-bit and 64-bit address spaces and the corresponding width of the integer register. One caveat is that the RV32E ISA, which is a subset of the RV32I, also has 32-bit wide registers but only 16 in total. That ISA is designed for 32-bit small embedded MCUs. RISC-V uses two's-complement to encode and decode to represent signed integer values. Unlike other ISA designs like MIPS, the RV32I is not a strict subset of RV64I. The idea is that the user can completely ditch 32-bit support or fully use the instruction encoding space, which otherwise needs to be reserved for the extension. The downside of that decision is that additional hardware design will be required to identify different base ISAs. But with the complete design, which naturally needs to capture illegal instructions and identify differences in addressing, means that
additional hardware is required anyway. RISC-V mitigated that issue by a very carefully designed instruction structure which makes identifying and supporting different base ISAs relatively low cost [9].

In the specification of the RV64I, some additional instructions are provided to work with 32-bit values for compatibility. These instructions are indicated by adding a ‘W’ suffix to the end of the opcode. These "W" instructions always ignore the upper 32bits of their inputs, which can be immediate or register context and output a 64-bit value by sign-extending the result 32-bit value [9].

In short, the RV64I base Integer Instruction set widens the integer registers and user address space to 64 bits. All values will be sign-extended to 64 bits. Note here it means that the architecture supports the maximum user memory addressing 64 bits, but that doesn’t mean it needs to have that big of memory [9].

2.1.3 The Compressed Extension

The compressed instruction-set extension is named the "C" extension. It is designed to reduce static and dynamic code size by using 16-bit instruction encoding instead of 32-bit [9].

According to the RISC-V specification, a regular program can have 50%-60% instructions which the RVC instructions can compress. This will lead to a 25%-30% reduction in code size [9].

The C extension is compatible with any base ISAs (RV32I, RV64I and RV128I), and the RISC-V standard uses the term "RVC" to cover any of these. Not only is the RVC compatible with them, but it can also be freely intermixed with the standard 32-bit instructions. The standard instructions will now be able to align at a 16-bit boundary. Also, with the C extension, the instruction-address-misaligned exceptions will not arise. This can also allow significantly better code density. [9]

One caveat is that although most RVC instruction encodings are standard across different base instruction sets, as shown in Figure 2.1, some are used for other functions based on the base instruction set [9].

![RVC opcode map](image)

**Figure 2.1: RVC opcode map[9]**

RVC has a very interesting feature: each RVC instruction can expand into a single normal 32-bit instruction in either the base ISA (RV32I/E, RV64I or RV128I) or the F and D standard extensions. This leads to two main benefits:
• When designing the hardware, this feature allows for easier decoding, simplifying verification and minimising modifications to existing microarchitectures.

• Compilers and software developers can be unaware of the RVC extension and leave the conversion to the assembler and linker, making coding and compiling easier. But if the compiler has the compression-awareness feature, the results will generally be better.

It is important to remember that the C extension is not designed to be a stand-alone ISA. Instead, it must be used with a base ISA (RV32I, RV64I or RV128I) [9].

2.1.4 Instruction Format

The base RISC-V ISA has a standard 32-bit instruction length and must be aligned by 32-bit. But as mentioned in Section 2.1.1, the RISC-V ISA have the ability to support extensions which has various instruction length. This enables the ISA to have 16-bit instruction decoding, meaning the smallest alignment can be 16-bit. This is used for the "C" extension mentioned in Section 2.1.3. The "C" extension also has the most complex instruction encoding, which is shown in Figure 2.2 while other RISC-V instructions have more uniform and simple encoding [9].

![Figure 2.2: RVC instruction format][9]

For other instructions, the RISC-V specification states that it keeps the source registers (rs1 and rs2) and the destination (rd) register always at the same position in all instruction formats in order to simplify decoding. Besides the CSR instructions, all other instructions which have immediates will always sign-extend those immediates so that the hardware complexity can be reduced. All those instructions are well-aligned as shown in Figure 2.3. The RISC-V organisation uses several letters to represent different instruction formats, such as "R", "I", "S" and "U". These instruction formats are the base instruction formats, which means they will cover most of the instructions defined in RISC-V ISA. The "R-type" represent the register operations like adding two register values into the destination
The "I-type" is called the immediate type and covers operations including register and immediates. It is also used for LOAD instructions. The "S-type" is used for instructions which store value. Lastly, the "U-type" is used for instructions that need 20-bit immediates. Besides those base instructions, two more formats (B/J) are shown in Figure 2.4 based on S-type and U-type, respectively. Those are commonly used for branch and jump instructions. Altogether those formats cover all encodings RISC-V ISA uses, even for extensions [9].

![Figure 2.3: RISC-V base instruction formats][9]

![Figure 2.4: RISC-V B/J instruction formats compared to S/U][9]

Do mentioned above, the RISC-V ISA instructions are very well-structured. This not only make it way easier to design the decoder and even the whole microarchitecture but also saves a lot of the hardware to run the core [9].
As mentioned in Section 2.1.2, the RISC-V ISA requires the identification of the base architecture for a given instruction. This is achieved by the base instruction-length encoding, which encodes the base instruction-length into the opcode as shown in Figure 2.5. Note that some of these encodings are still at the draft stage and only the 16-bit and 32-bit are frozen at this time. In this way, when decoding instructions, the desired instruction length can be decoded at once. In this way, mixing different base instruction-length is possible as the instruction itself will indicate its length, enabling Program Flow Controller (PFC) to easy set the Program Counter (PC) to the correct next instruction’s address.

2.2 RISC-V Toolchain

To use the final design, users need to run their code on the processor. This requires a suite of software to develop and simulate.

This section introduces the RISC-V GNU Compiler Toolchain, RISC-V Proxy Kernel and Boot Loader, Spike and Ripes as the toolchain for developing, simulating and verifying the processor design and software design.

2.2.1 RISC-V GNU Compiler Toolchain

The RISC-V GNU Compiler toolchain [11] is the cross-compiler for RISC-V. It has a full set of toolchains for two different targets: a generic ELF which uses Newlib (a C standard library aiming to support embedded devices) and a more sophisticated Linux-ELF which uses Glibc (the standard C library for GNU).

This toolchain contains several components:

- GCC for RISC-V: A compiler which is extended to support the RISC-V ISA and is able to compile C, C++ and other supported languages.

- GNU Binutils for RISC-V: A set of tools for the RISC-V binary file. This mainly includes an assembler, a linker, binary processing tools and a binary file viewer.
2.2 RISC-V Toolchain

- Newlib/Glibc: As mentioned above, this is the C standard library ported to RISC-V. Depending on the toolchain target, it will use Newlib or Glibc for Embedded systems or desktop OS respectively.

- GNU Debugger: This is the debugger ported to RISC-V ISA. But we need to notice here that it needs to be debugged under the RISC-V environment. This can be achieved by the RISC-V Proxy Kernel and Boot Loader mentioned below.

The toolchain is mainly used for generating the Executable and Linkable Format (ELF) file for the usage of simulators or the actual processor. The correct RISC-V ISA must be set in order to generate the ELF file and the toolchain correctly. That means the toolchain needs to be set to the desired processor instruction set and cannot have unimplied extensions configured. The full process can be found in Figure 2.6. The process is in detail:

- Write the source code: This is the code you would like to be compiled to run on the RISC-V processor. It can be in C, C++ or any other supported language.

- Preprocess the source code: In this step, the preprocessor walks through the input source code and processes through any preprocessor directive (like "#define" or "#include").

- Compile the source code: Now the compiler compiles the code into the assembly code. This means the compiler with translate the high-level programming language into the low-level assembly language, which is related to RISC-V instructions and architecture. Different RISC-V ISA and extensions will result in different assembly code.

- Assemble the assembly code: The assembler will take place and translate the assembly code generated previously into machine code and ready for the linker to use. Those files are called object files.

- Link the object files: Once all the object files are generated, the linker will combine them into a single executable file. That is the ELF file the processor can use.

![Figure 2.6: Process to generate ELF file](image)

2.2.2 RISC-V Proxy Kernel

The RISC-V Proxy Kernel (pk) [12] is designed to be a lightweight application runtime for hosting statically-linked RISC-V ELF binaries. This is named as a
proxy kernel because it is designed only to provide a strained RISC-V implementation which only has a limited input and output (I/O) capability and thus needs to proxying I/O related system calls to the host OS.

This software needs to have the RISC-V GNU Compiler Toolchain pre-installed and will use that software suite to build itself. It is useful for debugging software without connecting to the development board with the RISC-V core. It is also a necessary component for Spike and the RISC-V's GDB.

2.2.3 Spike

Spike is a RISC-V ISA simulator [13]. It implements a functional model for one or more RISC-V extensions. Spike supports multiple extensions, including the 32-bit and 64-bit base instruction set (RV32I and RV64I), the "E" extension, the "G" instructions suite, the "C" extension, and extensions inside the privilege level specification. There are several bit-manipulation extensions also introduced, some of which are not a part of the RISC-V specification. Thus, they will be disabled by default. Users can enable them by setting the build configuration.

The aim of this simulator is mainly to behave like the RISC-V processor in hardware in order to simulate instruction sets and work along with the RISC-V Proxy Kernel to simulate and debug software. The Spike simulator also allows users to introduce their custom instructions for the case they have customized the ISA.

2.2.4 Ripes

Ripes is a computer architecture simulator with a Graphical User Interface (GUI) built-in and it is also an assembly code editor. Ripes is built specifically for RISC-V ISA [14, 15].

It is not only suitable for teaching users the basic of computer architecture but also useful for simulating RISC-V software with certain RISC-V ISA. This is achieved by focusing on visualization-assisted simulation which in Ripes, it brings the datapath of the processor, the registers, the cache accessing and memory into graphical interfaces and even has live updating when simulating a program for users to have a more intuitive idea of what the processor is doing and what state each components are [15]. This allows for both easier learning and also for better simulating and debugging. The latter helps a lot when designing and verifying the processor.

Another outstanding feature is the Ripes support for simulating the pipelined architecture in an animated way so that it can clearly show not only the current stage the processor is executing but also the full pipeline stage and their related utilization of different components inside the processor [15]. A five-stage pipelined processor is shown in Figure 2.7.

Ripes also allows users to set the clock speed so that the processor can run at a fixed frequency. If the user needs to simulate a high frequency, Ripes also can disable the update of the visual datapath to achieve the desired frequency at the host machine [15]. This makes it useful not only for education and learning
2.2 RISC-V Toolchain

Figure 2.7: Ripes simulating a five-stage pipelined processor[13]

purposes but also to simulate the program and let the user have a visual picture of how various components like cache, register and memory will behave during the execution.

The visualization of Ripes makes it more intuitive to developers while the built-in assembler, disassembler and the ability to utilize an external compiler makes Ripes also suitable for simulating small and big programs, as shown in Figure 2.8. Those even make swift development possible on the RISC-V platform.

For aiding hardware development for RISC-V, Ripes also can be useful. When developing Ripes, developers also keep the simulation of hardware behaviour in mind. [15]. With the help of the Visual Simulation of Register Transfer Logic (VS-RTL), hardware developers can have a visual interface to view how the hardware will behave and how signals should be and how they transfer in-between components inside the processor. As Ripes developers mention, the overall goal for VSRTL is to make Ripes an environment for visual simulating and experimenting with the RTL-level processor core.

To sum up, Ripes is a useful tool for understanding the RISC-V ISA and having RTL level simulation for programs on different processors with different ISA extensions. It enables users to have a visual interface for datapath, cache accessing, memory and I/O.
2.3 Hardware

The hardware used in this research will mainly be Field Programmable Gate Arrays (FPGA) development boards. FPGA differs from other Integrated Circuits (IC) or Application Specific Integrated Circuits (ASICs) by its capability of reprogram on the field to change the purpose and behaviour of the chip [16]. This changing usage feature makes the FPGA an ideal platform for developers to develop hardware and make prototypes for later manufacture [17].

FPGA normally have several components:

- Logic clusters/Adaptive LUT
- Routing
- I/O block
- DSP block
- Embedded Memory
- PLL and DLL
- Configuration Chain

The basic component inside is called Look-Up-Table (LUT), which is an array or associative array. It is used to replace a fixed computation result with the array indexing operation to customise the data path. With the development of FPGA, people bundled LUT with some other logic elements like a Flip-Flop (FF),
2.3 Hardware

A selector, carry logic and others to form a Basic Logic Element (BLE). Multiple BLEs connect to each other to form a logic cluster [18].

In recent years, modern FPGAs adopted a new technique called Adaptive LUT. This is a feature that allows the larger LUTs split into smaller LUTs. For example, an adaptive logic element (ALE) can operate as a 6-input LUT or two 5-input LUTs depending on the usage. This feature helps to increase the flexibility of the FPGA and increase the area efficiency as the utilization ratio increases for those LUTs have a large number of inputs [18, 19].

The routing is the internal interconnection for the FPGA. It can be classified into four different types: the full-connecting type, the one-dimensional array type, the island-style (two-dimensional array type) and the hierarchical type [18, 20].

The Input/Output block (IOB), as its name indicates, is the part of the hardware which handles inputs and outputs. These blocks don’t have logic inside but have several special features like pull-up and pull-down resistors, output buffers and so on. These IOBs can also have different output standards or be optimized for special prototypes. Recently they can even have different voltage ranges, and their number has increased to the need for I/O banks[21, 22].

With the development of FPGA, people found out these FPGA boards are more and more used in Digital Signal Processing (DSP), for example, Finite Impulse Response (FIR) filter and Fast Fourier Transform (FFT), as those operations require a lot of multiplication operations, which require a lot of components to achieve when using the standard LUTs and still have poor performance due to the complex interconnection. So it makes sense to implement dedicated hardware for the multiplication operations, which are called DSP blocks [23].

The DSP block normally contains an accumulator, an Arithmetic Logic Unit (ALU), a multiplier and some other components. The DSP block is designed to accelerate the multiply and accumulate (MACC) operation,[18] which is represented in the formula below:

\[ Y \leftarrow A \ast B + Y \]  \hspace{1cm} (2.1)

To enable FPGAs to store data, memory is required. However, in the early FPGA era, they were based on LUTs and flip-flops. Only the flip-flops have the ability to store data, thus it is the only element for users to use as memory in early FPGAs. The problem is, if a user wants to achieve a big memory size, the resource requirement is extremely high. This makes storing large amounts of data nearly impossible for FPGA chips themselves. People try to use external memories to solve this problem. However, the connection between external memory and the FPGA chip quickly becomes the bottleneck for performance. Because of those issues, the onboard memory, which is known as Embedded Memory, is pre-built inside the FPGA chip[16, 18].

As FPGAs normally operate under different frequencies, the clock signal onboard will need to be able to change as demanded. This is achieved by either Phase-Locked Loop (PLL) or Delay-Locked Loop (DLL). The PLL mainly relays on the Voltage-Controlled Oscillator (VCO) whose oscillation frequency can be controlled by the input voltage [18]. This gives users the ability to change the
frequency as they want. The DLL on the other hand, mainly uses several delay elements to vary the frequency, but the end result is the same: let the user have the ability to change the frequency.

The configure chain is used to load the design into the actual FPGA board. Normally the hardware design will be converted into bitstream or configuration data. Both of them include information like truth tables for each LUT and routing info for the chip, and so on to setup the FPGA board. In most cases, FPGAs use the Joint Test Action Group (JTAG) interface for configuring and debugging. Figure 2.9 shows the configure chain of the DE10-nano board.
This chapter describes the method of designing the desired RISC-V processor. It includes a section for pre-studying which is Section 3.1. Section 3.2 describes the coding and implementation process for the processor. Section 3.3 focuses on the verification process for the core. At last, Section 3.4 mainly states methods used for measurement.

### 3.1 Pre-Study

In the beginning, we considered which base instruction sets to use and decided to pick RV64I as we wanted a capable and modern processor. Also, it is interesting to see how power-efficient the 64-bit architecture can be. But for backward compatibility, the RV32I ISA is also included for 32-bit support. We considered different extensions and also the "G" extension suite. But at last, we decided to only add the "C" extension for the main purpose of this research and also as we want to set a good foundation for the possibility and easiness for future extending this design.

For the architecture of the processor, for the same reason mentioned above, this research will have a single-stage, no-pipeline architecture. The benefit of this design choice is it makes the architecture simple and easier to understand and tweak. The downside, on the other hand, is that the performance of the chip will be limited by the long datapath/critical path. So the clock frequency needs to be carefully controlled to let the whole datapath works in one clock cycle.

The design takes some inspiration from the Ripes software’s single pipeline architecture [15], which is mentioned in Section 2.2.4. It is also shown in Figure 3.1. Need to note that the design has its own improvements and modifications so that the final architecture is not the same as the architecture shown in Figure 3.1.
3.2 Implementation

The designed processor uses SystemVerilog as the Hardware Description Language (HDL). It is built with the Program Counter (PC), Instruction Memory (IM), Decoder, Control Signal Generator (CSG), Arithmetic Logic Unit (ALU), Data Memory (DM) and several Multiplexers.

In the flowing sections, Section 3.2.1 describes the design for the PC and the way to fetch instructions from the IM, Section 3.2.2 describes the design of the instruction decoder, Section 3.2.3 discuss the detail of the CSG, in Section 3.2.4 the design of ALU and its peripherals are described, IM, DM and RF are mentioned in Section 3.2.5 and at the end, Section 3.2.6 shows how the input and output are assigned.

3.2.1 Instruction Fetch

In part for instruction fetching, the main component will be the PC register, multiplexers and an ALU. The PC ALU is designed to increase the program address, it takes the current PC value as one input, and the other input will be either 0x02 or 0x04. The PC ALU’s control signal is hard-wired to 0x0000, which operates the add operation. This will generate the "PC=PC+2" or "PC=PC+4" for 16-bit or 32-bit instructions to be fetched as the next instruction. The PC register takes input from either the output from the PC ALU or the result from the main ALU. In that way, the execution can be either sequential or jump to a certain address.

To compensate for the fact that the IM normally need one clock cycle to read the instruction, the address chosen to be the next PC value will send to the IM, so that the IM can output the correct instruction for the use of other components.

3.2.2 Decoder

The Decoder, as its name indicates, decodes the instruction fed from the IM.
In the design, as mentioned in Section 2.1.2, we will identify the base instruction set of the instruction and this will be an output for the CSG to use. One caveat here is that some of the 64-bit instructions, mainly those ones with the "*W" suffix at the end, share the same base length encoding as the 32-bit instructions they share the same control signal as their 32-bit counterparts. This means it's fine to identify them as 32-bit instructions as far as they are decoded correctly.

The Decoder need to decode instructions according to the RISC-V specifications. [9]. The ISA need to decode will be the RV32I, RV64I and RV64C. In order to make designing and the architecture simpler, the opcode, Source Register1 (rs1), Source Register2 (rs2), Destination Register (rd) and immediates are all sign-extended to the uniform width. That applies to both the base instruction set and the "C" extension. In the final design, the opcode will be 7-bit while the rs1, rs2, rd be 5-bit and immediates are 21-bit.

The Decoder also generate some additional function indicators named "funct2", "funct3" and "funct7". This can also be found in the specification. [9]. The number in these indicators' names implies the length of them. "Funct2" is mostly used in 16-bit Compressed RVC ISA, while the "funct3" and "funct7" is mostly used in the base instruction set. Note that the "funct6" mentioned in the specification [9] is zero-extended to "funct7" and "funct4" is identified to other components by a special flag to distinguish instructions.

The correct decoding is essential to make the processor work as intended. It is also crucial because its output will be used in the CSG.

In addition to those functions, to compensate for memories need one clock cycle to read and write. The Decoder will detect memory operations and actively instruct the whole processor to stall one cycle. That means all components of the processor will hold their value and wait for the memory data to be read or written.

### 3.2.3 CSG

CSG, the Control Signal Generator, is used to generate all the control signals. For example, the selection signal for all multiplexers, the control signal for the ALU and the write enable signal for the register file and DM.

The CSG takes inputs from the Decoder's outputs. By referring to the RISC-V specification [9] the CSG uses those signals to identify what should be the desired behaviour and sets control signals based on that. For example, if the instruction is decoded as a 16-bit instruction and is not a jump instruction, then the CSG will set the PC adder Multiplexer to output 0x02. So that the PC ALU will output PC+2 to correctly indicate the next sequence instruction.

The CSG is also used to handle the branch instructions. Inside the CSG, some additional logic is used to compare the rs1 and rs2 from the output of the RF and make branch decisions based on that.

With that functionality, the additional branch decision component is not needed thus simplifying the design.
3.2.4 ALU

The ALU is used for performing the arithmetic operation alongside the logic operations like bit-shifting and logic calculations.

For easier understanding and to simplify the design, the ALU design will be reused for both the main ALU and the PC ALU. The only difference is, as mentioned in Section 3.2.1, the PC ALU is hard-wired so that it will always execute the add operation.

3.2.5 Memory and RF

In this section, we will discuss the design for the memory and RF. Those two components share quite some similarities, so that's why we put them into one section to discuss.

The Register File (RF) is used for storing registers and need to be able to read and write them. As the specification [9] defines, the RV64 architecture should have 64-bit registers. It has a total of 32 registers, and except the first register, which is called x0, all others can be used as general-purpose registers. The x0 register is always hard-coded to 0 and cannot be changed.

The RF takes three 5-bit inputs for rs1, rs2 and rd, respectively, and only when the write-enable signal is set, the data from the data input port will write into the corresponding register. The RF has two 64-bit outputs and will always output the value of rs1 and rs2.

With the purpose of having a general RISC-V processor for various FPGAs, the dedicated memory can not be used as they will differ from platform to platform. However, thanks to the modern FPGA toolchain, they can automatically use the embedded memories if possible. So we need to design in a way that fits those memories.

The final memory design is synchronised with the system clock and has a write-enable signal. The other inputs are the address port and the port for the input data. The timing for the memory will be one clock cycle to read and one clock cycle to write. The IM and DM are almost the same, the differences are the width of the address port and data ports.

This makes the memory fits perfectly into the DE10 board, the Quartus tool will Synthesis it into the M10K memory blocks.

3.2.6 I/O

In order to have some output from the processor, to obtain data for both normal usage and debugging/verifying purposes. Some I/O interface is needed. In this case, we created eight General Purpose Input and Output (GPIO). Those GPIOs are connected to a specific register: x31. It can show the value of the eight Most Significant Bits (MSB) in that register. On the FPGA board, they are connected to LEDs for easier observation.
3.3 Verification

To verify the designed processor is working, a verifying process is needed. This is achieved by using the ModelSim software and some test program like the one shown in Figure 3.2. To better test the processor, the test program is separated into RV64I and RV64C. This is because the assembly will be quite different for them and for the purpose of covering as many cases as possible. The test program for the RVC ISA is shown in Figure 3.3. Here hardware debugging is also used to better verify the processor is working.

**Figure 3.2:** A simple test program

**Figure 3.3:** The test program for RVC ISA
3.3.1 Software Verification

Ripes is again used as a reference to cross-check that the processor’s behaviour is correct. It is used to observe the state of registers, the flow of the program and the internal datapath details like control signals and internal values. The comparison is shown in Figure 3.4. By comparing results in Ripes and Multisim, we can verify that the design is behaving as expected.

![Figure 3.4: Comparison with Ripes](image)

3.3.2 Hardware Verification

As mentioned earlier, in addition to Ripes and Modelsim, hardware verification is also implemented in order to make sure that the processor is working.

This is achieved by implementing either the signal tap module for Intel FPGA boards or the debugger hub for AMD FPGA boards. Note here that the processor created in this research can work across platforms. In order to verify it’s working on the FPGA and simplify the verifying process, the platform-specific debugging block is used. However, they are for verification only and won’t change the property that processor is still cross-platform.

As shown in those figures, the Intel implementation uses its Dedicated Debugging Intellectual Property (IP) and sends out the result through the JTAG interface. While AMD’s way is to generate totally dedicated hardware IP for debugging and outputting.

As the hardware debugging needs to implement additional components into the FPGA, it needs some extra resources. This includes not only LUTs but can also need block memories and others. This will be shown in Chapter 4 and will be discussed in the conclusion chapter.

In our case we are using DE10-nano for Intel platform and Perf-V for AMD platform. The FPGA chip on the DE10-nano is 5CSEBA6U23I7 which has 41910
logic blocks, 5662720 bits onboard memory and a speed grade of 7. For Pref-V, the FPGA chip is XA7A50T which has 52160 logic cells, 2700000 bits onboard memory and a speed grade of -11.

3.4 Measurement

3.4.1 Power Consumption

For power consumption, we use the power analysis tools inside the FPGA Integrated Design Environment (IDE). The tools are Quartus and Vivado for Intel and AMD respectively. With the help of generating switching rates and back-annotated delay estimates from Modelsim with Quartus, we chose the Quartus Power analysis tool as the main data source and used Vivado as a reference.

Settings for the Quartus Power Analysis tool are:

- Environment temperature: 0-100°C
- Ambient Temperature: 25°C
- Cooling Solution: No heat sink with 200 LFpM airflow

3.4.2 Resource Utilization

For resource utilization, we rely on reports generated from Quartus and Vivado. They can generate separate reports for both post-synthesis and post-implementation. The post-implementation shows the actual resource usage on the actual FPGA board. The amount of resources required can change from post-synthesis to post-implementation as the implementation operation will optimize the design with the direction of speed or space. In this case, the choice of balance is made so the tool will keep both speed and space in mind.
This chapter shows the results of the designed processor. Results from both RV64I and RV64IC architecture are shown here. In this chapter, we discuss the results of the work in different aspects including the model compiling, signals inside the hardware and power consumption of the model.

4.1 Code Result

The code has been open-sourced and can be found in LiU’s Gitlab page [24].

4.2 Compiling Result

In this section, resource usage for both RV64I and RV64IC architecture is shown. There is another section shows the resource usage with the hardware debugger probing the reset signal, instructions signals, GPIO output signals and the PC address signals.

Here we will use the report from the Intel platform which is generated with Intel Quartus IDE software as the main data source. This is for the purpose of simplifying the report and making it more readable and easier for comparison.

One can easily run the flow of his/her EDA tool to see the resource usage on his/her FPGA development board.

4.2.1 Post-Implementation

Here shows the post-synthesis report for the resource usage regarding RV64I and RV64IC architecture shown in Figure 4.1. The next one, Figure 4.2 and Figure 4.3 shows the resource usage for individual entities.
Figure 4.1: Overall Resource utilization for RV64I (Left) vs RV64IC (Right)

Figure 4.2: Resource utilization of each entities for RV64I

Figure 4.3: Resource utilization of each entities for RV64IC
4.2.2 Resource Utilization with Debugger

Figure 4.4 highlights the additional resource requirement for the hardware debugger. Note that in order to expose the desired signals for debugging, there needs some additional circuits for that purpose.

![Resource Utilization by Entity Table]

\textit{Figure 4.4: Resource needed for hardware debugger}

4.3 Hardware Runtime Signals

Figure 4.5 shows the usage of the hardware debugger to observe runtime signals on the actual FPGA development board.
4.4 Power Consumption

In this section, power consumption for both RV64I and RV64IC is shown. Below, Figure 4.6 shows the summary of power consumption. The Figure 4.7 shows the power consumption of different types of blocks. Figure 4.8 shows the power dissipation of each design hierarchy. The last one, Figure 4.9 shows details of the confidence level of the power consumption analysis.

Figure 4.6: Power Consumption Summary for RV64I (Left) vs RV64IC (Right)
4.4 Power Consumption

<table>
<thead>
<tr>
<th>Block Type</th>
<th>Total Thermal Power by Block Type</th>
<th>Block Thermal Dynamic Power</th>
<th>Block Thermal Static Power</th>
<th>Routing Thermal Dynamic Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>27.49 mW</td>
<td>27.49 mW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>304.98 mW</td>
<td>304.98 mW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2.17 mW</td>
<td>2.17 mW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.33 mW</td>
<td>0.33 mW</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 4.7:** Power Consumption of each blocks for RV64I (Up) vs RV64IC (Down)

<table>
<thead>
<tr>
<th>Block Type</th>
<th>Total Thermal Power by Block Type</th>
<th>Block Thermal Dynamic Power</th>
<th>Block Thermal Static Power</th>
<th>Routing Thermal Dynamic Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20.53 mW</td>
<td>20.53 mW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>366.73 mW</td>
<td>366.73 mW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1.50 mW</td>
<td>1.50 mW</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0.53 mW</td>
<td>0.53 mW</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 4.8:** Power Consumption of each hierarchy for RV64I (Up) vs RV64IC (Down)
4.5 Program Compiling Result

In this section, the compilation result for both RV64I and RV64IC ISA is shown in Table 4.1

<table>
<thead>
<tr>
<th>Program</th>
<th>RV64I</th>
<th>RV64IC</th>
<th>Space Saved</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program A</td>
<td>272</td>
<td>194</td>
<td>28.6%</td>
</tr>
<tr>
<td>Program B</td>
<td>192</td>
<td>118</td>
<td>38.5%</td>
</tr>
</tbody>
</table>

Table 4.1: Code section space usage (bytes)
5.1 Results

The results from the design are quite interesting.

Starting from the resource usage for the RV64I and RV64IC. From Figure 4.1, it is easy to observe that both architectures need almost the same amount of hardware resources as the RV64IC. However if one takes a closer look at Figure 4.2 and Figure 4.3 for the hardware consumption of each entity, one can see that without the support of the Compressed extension, the Decoder uses relatively fewer hardware resources. However other components like circuits for ALU and the one for the RF need more resources, and this can be because of the lack of the replacement of the Compressed extension, which makes data flow inside the processor become wider.

With similar area usage in mind, the power consumption comparison becomes very interesting, and from there, we can see the power consumption drops from 2.81 W to 2.36 W. More important is that from the summary shown in Figure 4.6, almost all of that improvement comes from the dynamic power dissipation, which tells us that improvement is actually from the use of the Compressed extension. This can also be obtained from Figure 4.7, showing that most gains come from the combinational cell.

One side note here is that the power used by the register cell is slightly higher in RV64IC architecture compared to the RV64I architecture, with a closer look into the power consumption analysis for hierarchies shown in Figure 4.8 and the instruction specification [9], we believe it is because the Compressed instructions are more complex and their decoding process is more complicated. However saving resources from ALU compensates for that additional power consumption and makes it more power efficient.

Apart from the improvement of the FPGA design, the end product of compil-
Discussion

Running the same program with different architecture's toolchain shows an improvement in the size of the code. As shown in Table 4.1 the code section space usage for two same programs can be saved up to 38.5% with the help of the Compressed extension. This matches the statement inside the ISA specification [9] that the Compressed extension can reduce the size of the code by 25% to 30%. This can help a lot for processors which don’t have large instruction storage.

From the hardware resource usage summary, it is clear that the use of LUT is not as much as the block memory. This opens up a lot of potential for future extension and enables the processor to have more functionalities or improvements.

5.2 Method

In this section, the method used in this research is discussed. Then, mainly the processor’s implementation and the verification and measurement methods will be addressed.

5.2.1 Implementation

There is room for improvement and discussion regarding the processor’s design.

As mentioned in Section 3.2.3, the decision of making a jump or not is computed and controlled by the CSG. This requires an additional ALU or subtraction unit to compare the rs1 and rs2 values. And it uses combination operations to take or discard the jump operation, making the CSG design more complex. There is another possible way to handle this by letting the CSG only generates the control signal, using the main ALU to compare values and having one additional branch decision component to decide if take a jump or not and use the ALU for calculating the PC address to add the offset to the current PC address. However as one can notice here, that method also requires additional components and changing the PC ALU design may introduce problems for the processor like creating combinational loops or preventing the processor from performing operation in one clock cycle.

Readers might notice that there is no I/O interface other than the GPIO in the design, and there is no way to load different programs into the FPGA board in the runtime. One possible solution is implementing the Universal asynchronous receiver-transmitter (UART) to send and receive data between the processor and other devices like another computer. A control signal can be added to control the enabling of the write operation for the instruction memory or even design a specific data format to transmit the address and instructions. For data output, an additional memory dedicated for UART so that a pointer pointing to its address will put data into the UART buffer for sending it out.

As mentioned previously in the Section 3.2.2, the Decoder detects if an operation is memory related and will stall the processor if it is. This is to compensate that memories need one clock cycle to read and write. However this stalling operation worse the performance of the processor. The main problem here is to preserve the correct register index to read or write to as it is a single-stage proces-
The work in a wider context

5.2 Verification and Measurement

For verification, using a single test program instead of using Modelsim's coverage function for verification can lead to not testing all the conditions or ignoring some corner cases. Although the test program covers all the instructions of RV32I, RV64I and RV64C ISAs, using Modelsim's coverage tool can make the simulation verification more intuitive and easier to identify corner cases and see the coverage of different entities.

For hardware verification, there is more to do like measure power consumption with hardware, other than the lack of I/O mentioned above. The limit of hardware resources also makes observing all signals more difficult, especially if one needs to see the output from memories. It will require not only LUTs but also memories, as shown in Figure 4.4, in order to capture and output signals. This also limits the resolution of the data capturing.

Although the power consumption analysis from the FPGA IDE software can provide different options for different use cases like room temperature, operation temperature and so on, a real-world power consumption measurement can still make the report more convincing and fits more to the daily usage scenarios. However, one should notice that the processor’s power consumption is not as high as it is in the mV range, so to see the differences between different architectures can be quite challenging and may need high-precision measurement tools.

One other shortcoming here is that the program running on the processor may not create the most intensive workload for the processor, so the power consumption might not be the highest case. This can be improved by using some test benches like Embench [25] and Coremark [26]. However one also needs to notice that those test benches must be ported to the designed architecture. They usually only have been ported to RV32G or RV64G architecture.

5.3 The work in a wider context

As mentioned earlier, the proposed processor in this research is quite simple compared to other RV64 cores like cva6 [7], The Berkeley Out-of-Order RISC-V Processor (BOOM) [27] and the RiscyOO [8] but it still is a full implementation of RV64IC architecture. As also mentioned above, the usage of the logic resources onboard the FPGA chip is very small which means it has a lot of potential to expend in the future. It can also learn for those cores' design and have more sophisticated and complex architecture improvements like pipeline and maybe even Out-of-Order execution (OoOE).
This processor can also be used in scenarios with high energy efficiency ratio requirements like IoT devices as it not only has reduced power consumption compared to the RV64I architecture, but also it is still capable of 64-bit instructions and have wider registers compared to the 32-bit architecture. It can also be extended to have custom instruction extensions to optimize certain operations or data paths to achieve even better power efficiency.
Conclusion

In this research, we introduced a single-stage power-efficient general-purpose RISV core in RV64IC architecture. The aim of it is to prevent using any 3rd party IPs and have only SystemVerilog code, try to make it easy to understand and be able to extend it in the future. Also, compare it with the Compressed extension to understand how much improvement we gain from that extension.

The processor is designed with some special design considerations like the reuse of ALUs, the CSG and the stall operation to avoid any Data Hazard introduced because of the memory. Also, the PC bypassing to compensate for the same delay of the instruction memory.

The processor can successfully run on both Intel and AMD’s FPGA development board. It is tested with DE-10 nano and Perf-V development board. The final clock speed for the processor is 33 MHz due to the single-stage architecture but function wise it shows no problem.

From the results, the hardware resources usage is shown, and the improvement of power consumption with the help of the Compressed extension is also demonstrated from different aspects. Also, in that section, we show differences in code size for different architectures which shows how the Compressed extension can be more efficient in another way.

This research does not only provides a RISC-V processor in a modern 64-bit architecture with the Compressed instructions extension but also expands it into a full RV64GC core and with this running operating systems becomes possible. As it is simple and easy to understand, it should be quick to get hands-on and enable one to learn RISC-V architecture and explore what he/she can do with it. In the future, this processor can add support for other RISC-V instruction extensions like the Atom instructions, Integer Multiplication and Division and others. It can be fitted with custom instructions to optimize specific operations.
Appendix


