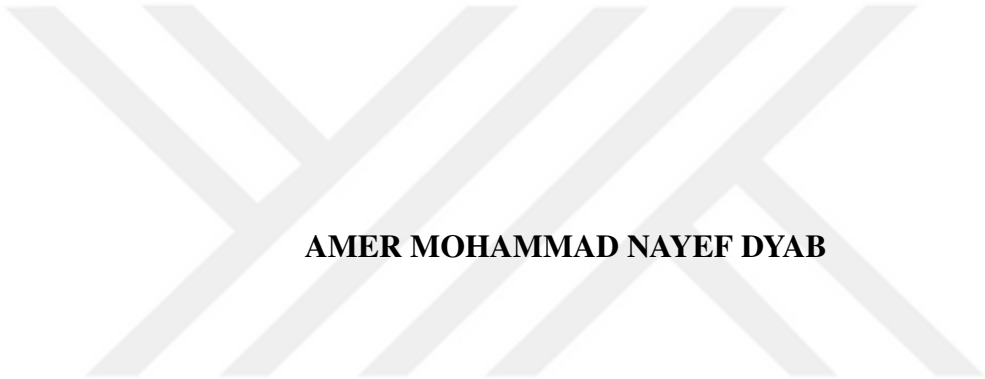


**DESIGNING CUSTOM RISC-V BASED SOCS:
FROM INTEGRATION TO ISA OPTIMIZATION**



AMER MOHAMMAD NAYEF DYAB

ÖZYEĞİN UNIVERSITY

JANUARY, 2025

DESIGNING CUSTOM RISC-V BASED SOCS: FROM INTEGRATION TO ISA OPTIMIZATION

by
Amer Mohammad Nayef Dyab

Thesis
Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science

in
Electrical and Electronics Engineering

Advisor: Prof. H. Fatih Uğurdağ

Graduate School of Science and Engineering
Özyeğin University
İstanbul

January, 2025

DESIGNING CUSTOM RISC-V BASED SOCS: FROM INTEGRATION TO ISA OPTIMIZATION

Approved by:

Prof. H. Fatih Uğurdağ, Advisor,
Department of Electrical and Electronics
Engineering
Özyeğin University

Assoc. Prof. Cenk Demiroğlu
Department of Electrical and Electronics
Engineering
Özyeğin University

Asst. Prof. Onur Demir
Department of Computer Engineering
Yeditepe University

Approval Date: 27 December, 2024

*To the soul of my late father, to my supportive family in Jordan, and in
honor of our resilient people in Palestine and Lebanon.*



DECLARATION OF ORIGINALITY

I hereby declare that I am the sole author of this thesis and that this is the true copy of my thesis, including the final revisions, approved by my thesis committee. All data and information have been obtained, produced, and presented in accordance with the rules of research ethics and principles of academic honesty. As required by these rules, to the best of my knowledge I have acknowledged ideas, thoughts, and any copyrighted material in accordance with the standard referencing rules. I certify that any part of this thesis has not been submitted for a degree or diploma in another educational institution.

Amer Mohammad Nayef Dyab

ABSTRACT

Soft processor cores, particularly open-source processor architectures like the RISC-V, have extended the diversity and flexibility of SoC designs, tailoring the hardware design to the software needs, instead of the opposite. These developments enable unprecedented flexibility in both hardware and software aspects, motivating the exploration of creating highly-customizable SoC integrations that can adapt to a diverse range of application requirements. The first part of this thesis begins by implementing the concept of a flexible and customizable SoC prototype, the "OzU SoC", tailored to the meet requirements of an electric metering application. The philosophy of this development work is to quickly adapt the design to its specific application needs, rather than relying solely on software optimizations on general-purpose designs. This effort culminates in implementing a prototype of the OzU SoC as a soft Intellectual Property (IP) demonstrated on a Field Programmable Gate Array (FPGA). The second part of this thesis focuses on optimizing a key challenge in modern SoC design: physical area reduction. On-chip memory, a significant factor in determining the size of modern SoC design, is the main interest. In this part, the Novel Variable-Length Inclusive Instruction Set Architecture (NOVLI-ISA) is proposed as a compatible extension to the 32-bit base integer RISC-V ISA, known as RV32I. This part includes the design, implementation, and evaluation of the proposal, with the primary goal of machine code size reduction without compromising complexity, functionality, or flexibility. This thesis demonstrates key aspects of modern customizable SoC development, contributing to the RISC-V community with the proposed extension.

Keywords: RISC-V, System-on-Chip, Instruction Set Architecture, Machine Code Optimization

ÖZET

Açık-kaynak işlemci mimarileri, özellikle RISC-V gibi sentezlenebilir işlemci çekirdekleri, donanım tasarımını yazılım ihtiyaçlarına göre uyarlayarak, Yongada Sistem (SoC) tasarımlarının çeşitliliğini ve esnekliğini artırmıştır. Bu gelişmeler, donanım ve yazılım alanlarında eş benzeri görülmemiş bir esneklik sağlamak ve geniş bir uygulama yelpazesinin gereksinimlerine uyum sağlayabilecek, yüksek düzeyde özelleştirilebilir SoC entegrasyonlarının oluşturulmasını araştırmayı teşvik etmektedir. Bu tezin ilk bölümü, bir elektrik sayaç uygulamasının gereksinimlerini karşılamak üzere tasarlanmış esnek ve özelleştirilebilir bir SoC prototipi olan "OzU SoC" konseptinin uygulanması ile başlar. Bu geliştirme çalışmasının temel felsefesi, genel amaçlı tasarımlarda yalnızca yazılım optimizasyonlarına güvenmek yerine, tasarımı hızlı bir şekilde belirli bir uygulamanın ihtiyaçlarına uyarlamaktır. Bu çabanın sonunda OzU SoC, Sahada Programlanabilir Kapı Dizisi (FPGA) üzerinde gerçekleştirilmiş, sentezlenebilir bir Fikri Mülkiyet (IP) bloğu olarak prototiplenmiştir. Tezin ikinci bölümü ise modern SoC tasarımındaki temel bir zorluğun, fiziksel alanın minimize edilmesine odaklanır. Modern SoC tasarımının boyutunda önemli bir faktör olan yongadaki bellek, bu bölümün ana odağıdır. Bu bölümde, RISC-V ISA'nın 32-bit tamsayı versiyonu olan RV32I'ye uyumlu bir genişletme olarak Yenilikçi Değişken Uzunluklu Kapsayıcı Komut Seti Mimarisi (NOVLI-ISA) önerilmektedir. Bu bölüm, önerinin tasarımı, uygulanması ve değerlendirilmesini kapsar ve temel amacı, makine kodu boyutunu karmaşıklık, işlevsellik veya esneklikten ödün vermeden azaltmaktır. Bu tez, modern özelleştirilebilir SoC geliştirme alanındaki temel unsurları göstermekte ve önerilen genişletme ile RISC-V topluluğuna katkıda bulunmaktadır.

Anahtar Kelimeler: RISC-V, Yongada Sistem, Komut Seti Mimarisi, Makine Kodu Optimizasyonu

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my advisor, Prof. H. Fatih Uğurdağ, for his invaluable mentorship, assistance, and unwavering support since we started working on this research and until its presentation. His decades-long academic and industry expertise, accompanied by his transformational leadership, have been instrumental in shaping my motivation towards bringing this work to fruition and considering further career directions. I am truly fortunate to have had the opportunity to work with and learn from such an inspiring, experienced mentor.

I would like to extend my sincere thanks to the executives and staff of Synopsys Armenia CJSC, led by Dr. Yervant Zorian, Chief Architect and Fellow at Synopsys and President of Synopsys Armenia, who have been key co-organizers of the IEEE 20th East-West Design & Test Symposium (EWDTS), where I had the privilege to present part of the work of this thesis, the NOVLI-ISA. The feedback, insights, and discussions during the conference provided valuable perspectives on my work and the semiconductor industry overall. Their professionalism and hospitality created an excellent environment for intellectual exchange, networking, and collaboration.

Finally, I am grateful to my professors, administrative staff, and other faculty members at the Graduate School of Science and Engineering (GSSE) of Özyeğin University, who have assisted and supported me in various ways during my academic journey. I would especially like to extend my gratitude to Dr. Evşen Yanmaz, Associate Professor at Electrical and Electronics Engineering Department, for her help and feedback in finalizing this document, and Ms. Gizem Bakır, an Administrative Supervisor at the GSSE, for her remarkable dedication in assisting graduate students in her technical domain.

TABLE OF CONTENTS

DECLARATION OF ORIGINALITY	iv
ABSTRACT	v
ÖZET	vi
ACKNOWLEDGEMENTS	vii
LIST OF TABLES	xi
LIST OF FIGURES	xiii
LIST OF ACRONYMS AND ABBREVIATIONS	xiv
1 INTRODUCTION	1
1.1 Modern SoC Design and RISC-V Architecture	1
1.1.1 Emergence of RISC-V	6
1.1.2 Challenges in SoC Design	9
1.2 Research Objectives and Contributions	14
1.2.1 Integrating a Prototype Microcontroller SoC	14
1.2.2 Optimizing Machine Code Efficiency	15
1.2.3 Exploring Areas of Automation	16
1.3 Structure of the Thesis Report	17
2 INTEGRATION OF CUSTOMIZED SOCS	18
2.1 Examples of Customized SoCs	18
2.2 SoC Design and Development Workflow	22
2.3 Designing the OzU SoC Prototype	30
2.3.1 User Requirements	30
2.3.2 System Architecture	31
2.3.3 Memory Organization and Dataflow	35
2.3.4 Integrating Peripherals and Auxiliary Units	37
2.3.5 Customizability and Scalability	45
2.4 Implementation and Testing	46

2.4.1	Verilog Implementation	47
2.4.2	Machine Code Generation	48
2.4.3	Simulation and Testing	49
2.5	Prototyping and Realization	50
2.5.1	FPGA Prototyping	50
2.5.2	Limitations and Further Development	52
2.6	Possible Areas of Automation	53
2.6.1	Automated RTL Customization	53
2.6.2	Automated Code and Memory Generation	53
2.6.3	Automated Testing and Verification	54
3	ISA OPTIMIZATION	56
3.1	The Necessity for Code Optimization	56
3.2	Machine Code Optimization in the Literature	57
3.2.1	The Ratified RV32C Extension	59
3.3	Design Philosophy of the NOVLI-ISA	61
3.3.1	Compatibility with Standard Memories	61
3.3.2	Optimization for Embedded Applications	61
3.3.3	Maintaining the Simplicity of RISC-V	62
3.3.4	Inclusiveness, Enhanced Functionality, and Efficiency	62
3.3.5	Minimal Altering of Existing RV32I/E Implementations	63
3.4	NOVLI-ISA Formats and Functions	63
3.4.1	8-bit Micro Instructions	64
3.4.2	Unary Register Operations Instructions	65
3.4.3	Register-Register Operations Instructions	65
3.4.4	Register-Immediate Operations Instructions	66
3.4.5	Load Immediate / Upper Immediate Instruction	66
3.4.6	Memory Operations Instructions	66
3.4.7	Branching Operations Instructions	68
3.4.8	Jump / Jump and Link Instruction	68
3.4.9	Jump / Jump and Link (Register) Instruction	69

3.4.10	Instructions Left Without Optimization	70
3.5	Implementation of the NOVLI-ISA	70
3.5.1	Hardware and Software Requirements	70
3.5.2	Designing the RV32IN Prototype.....	75
3.5.3	Coding, Simulation and Testing	77
3.6	Feasibility Evaluation Results	78
3.7	Integrating the RV32IN Prototype within the OzU SoC	82
3.8	Future Research for the NOVLI-ISA.....	82
4	CONCLUSIONS	84
4.1	OzU SoC as a Testbed for Future Development	84
4.2	Contributing the NOVLI-ISA to the RISC-V Community	88
4.3	Future Research Directions	90
4.4	Summary of the Thesis	91
	REFERENCES	92
	APPENDICES.....	98
APPENDIX A	— RV32I/E ISA CARD.....	99
APPENDIX B	— RV32C EXTENSION CARD	100
APPENDIX C	— OTHER RV32I EXTENSIONS	101
APPENDIX D	— RV32I PSEUDO INSTRUCTIONS.....	102
APPENDIX E	— RV32I AND F REGISTER FILES.....	103
APPENDIX F	— STM32F40XXX SOC DIAGRAM	104
APPENDIX G	— PICORV32 PARAMETERIZATION	105
APPENDIX H	— AXI4-LITE TRANSACTIONS TIMING	110
APPENDIX I	— OZU SOC DETAILED DIAGRAM	111
APPENDIX J	— OZU SOC PLANNED ADDRESS SPACE	112
APPENDIX K	— PIN FUNCTION CONTROLLER.....	114
APPENDIX L	— NOVLI-ISA BLUEPRINT	117

LIST OF TABLES

Table 2.1:	Original User Requirements for OzU SoC Pins.	31
Table 2.2:	AXI4-Lite Write Address (AW) Channel Signals.	34
Table 2.3:	AXI4-Lite Write Data (W) Channel Signals.	34
Table 2.4:	AXI4-Lite Write Response (B) Channel Signals.	34
Table 2.5:	AXI4-Lite Read Address (AR) Channel Signals.	34
Table 2.6:	AXI4-Lite Read Data (R) Channel Signals.	35
Table 2.7:	Memory Organization of the OzU SoC.	36
Table 2.8:	Integrated OzU SoC Components Excluding Memories.	38
Table 2.9:	Unimplemented Components of the OzU SoC.	39
Table 2.10:	GPIO Module Registers.	40
Table 2.11:	TCP Module Registers.	41
Table 2.12:	UART Controller Registers.	42
Table 2.13:	I2C Transceiver Registers.	43
Table 2.14:	SPI Controller Registers.	43
Table 2.15:	Interrupt Controller Registers.	44
Table 3.1:	Instruction Subsets of the NOVLI-ISA.	64
Table 3.2:	8-bit Micro Instructions Format.	64
Table 3.3:	Unary Register Operations Instructions Format.	65
Table 3.4:	Register-Register Operations Instructions Format.	65
Table 3.5:	Register-Immediate Operations Instructions Format.	66
Table 3.6:	Load Immediate / Upper Immediate Instruction Format.	67
Table 3.7:	Memory Operations Instructions Format.	67
Table 3.8:	Branching Operations Instructions Format.	68

Table 3.9: Jump / Jump and Link Instructions Format.	69
Table 3.10: Jump / Jump and Link (Register) Instructions Format.	69
Table 3.11: NOVLI-ISA Optimization Rules.	79



LIST OF FIGURES

Figure 1.1: Block Diagram of a Sample SoC Design.	2
Figure 1.2: Silicon Die Image of HiSilicon’s Kirin 980 Mobile SoC.	4
Figure 1.3: An IC Package Cutout Showing Silicon Die Connection to Pins.	5
Figure 1.4: Key Members of the RISC-V Foundation.	6
Figure 1.5: ISA Decomposition of the RV32IMAC Architecture.	7
Figure 2.1: Block Diagram of YosysHQ PicoSoC.	19
Figure 2.2: Block Diagram of Raven SoC.	20
Figure 2.3: Interconnects Architecture in STM32F40xxx MCUs.	24
Figure 2.4: Memory-Mapped Space Organization in SAM D21 MCUs.	26
Figure 2.5: Simplified Block Diagram of the OzU SoC.	32
Figure 2.6: Section of the Verilog-Coded STR Memory in OzU SoC.	35
Figure 2.7: Waveform Simulation Views of the Final OzU SoC Prototype.	55
Figure 3.1: ISA Card of the RV32C Extension.	60
Figure 3.2: Memory-Aligned Instructions Cases.	71
Figure 3.3: Example Case of a Fragmented Instruction.	73
Figure 3.4: Fetch Unit State Transitions.	74
Figure 3.5: Block Diagram of the RV32IN Prototype.	76
Figure 3.6: RV32IN Simulation Handling Fragmented Store Instructions.	77
Figure 3.7: Script Output of Optimizing a Single Input Program.	80
Figure 3.8: Script Analytics Output for the Overall Batch.	80
Figure 3.9: Code Size Reduction Percentages of the Evaluation Batch.	81

LIST OF ACRONYMS AND ABBREVIATIONS

ADC	Analog-to-Digital Converter
AI	Artificial Intelligence
ALU	Arithmetic Logic Unit
AMBA	Advanced Microcontroller Bus Architecture
APB	Advanced Peripheral Bus
ATE	Automated Test Equipment
AXI	Advanced Extensible Interconnect
CCM	Core-Coupled Memory
CPU	Central Processing Unit
CSV	Comma-Separated Values
DAC	Digital-to-Analog Converter
DMA	Direct Memory Access
DRC	Design Rule Checking
DSP	Digital Signal Processor
DSPI	Dual Serial Peripheral Interface
DU	Decode Unit
EEPROM	Electrically-Erasable Programmable ROM
eSPI	Enhanced Serial Peripheral Interface
EU	Execution Unit
FFT	Fast Fourier Transform

FIFO	First In, First Out
FPGA	Field-Programmable Gate Array
FSM	Finite-State Machine
FSMC	Flexible Static Memory Controller
FU	Fetch Unit
GCC	GNU Compiler Collection
GDB	GNU Debugger
GPIO	General Purpose Input/Output
GPU	Graphics Processing Unit
HIL	Hardware-In-the-Loop
HLS	High-Level Synthesis
I2C	Inter-Integrated Circuit
I2S	Inter-Integrated Circuit Sound
IC	Integrated Circuit
IDE	Integrated Development Environment
INTC	Interrupt Controller
I/O	Input/Output
IoT	Internet of Things
IP	Intellectual Property
IRQ	Interrupt Request
ISA	Instruction Set Architecture
ISC	Internet Systems Consortium

JTAG	Joint Test Action Group
LUT	Look Up Table
LVS	Layout-Versus-Schematic
MCU	Microcontroller Unit
NOP	No Operation
NOVLI-ISA	Novel Optimized Variable-Length Inclusive ISA
NVIC	Nested Vectored Interrupt Controller
OTP	One-Time Programmable
OzU	Özyeğin University
PC	Program Counter
PCB	Printed Circuit Board
PCPI	Plug-in Custom Peripheral Interface
PFCN	Pin Functions Controller
PMU	Power Management Unit
PPA	Performance, Power, and Area
PPAC	Performance, Power, Area, and Cost
PULP	Parallel Ultra-Low Power
PWM	Pulse-Width Modulation
QFP	Quad Flat Package
QSPI	Quad Serial Peripheral Interface
RC	Resistor-Capacitor
RTC	Real-Time Clock

RTL	Register-Transfer Level
RV32E	32-bit Embedded RISC-V Architecture
RV32I	32-bit Base Integer RISC-V Architecture
RVFI	RISC-V Formal Interface
RX	Receiver
SDIO	Secure Digital Input Output
SDK	Software Development Kit
SoC	System-on-Chip
SPI	Serial Peripheral Interface
SPROM	Serially-Programmable Read-Only Memory
SRAM	Static Random Access Memory
SWD	Serial Wire Debug
TX	Transmitter
UART	Universal Asynchronous Receiver-Transmitter
VLSI	Very-Large-Scale Integration
WDT	Watchdog Timer
XCVR	Transceiver
XIP	eXecute-In-Place

1. INTRODUCTION

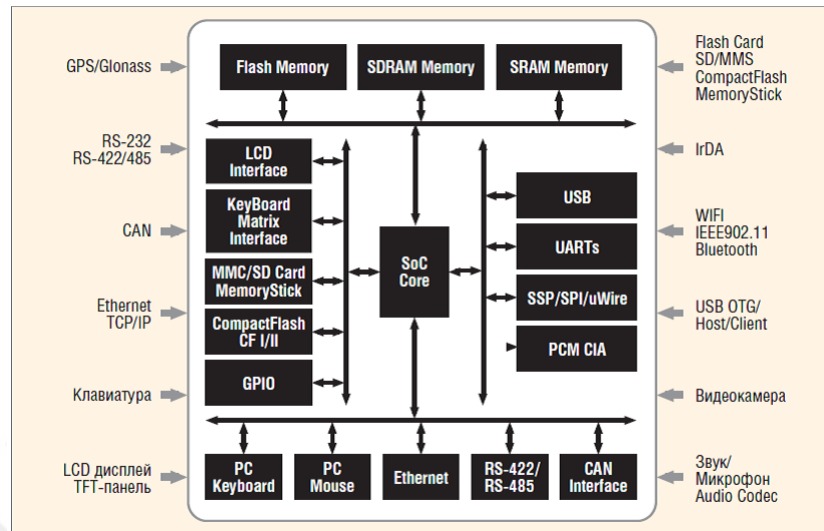
This chapter introduces the technical domain of the thesis, highlights the associated engineering challenges, and concludes with the research objectives and contributions. The chapter reviews the general fundamentals of modern System-on-Chip (SoC) design, introduces the RISC-V Central Processing Unit (CPU) architecture and its significance in the future of computing, highlighting the key engineering challenges in modern SoC design and development. The research objectives and contributions of the thesis are then introduced, identifying key goals and expected outcomes. Finally, the outline and content of the document are explained to the reader.

1.1 Modern SoC Design and RISC-V Architecture

SoC design has emerged as the backbone of modern electronics, driving innovation in several evolving industries such as mobile devices, embedded systems, communication systems, automotive and industrial automation, and consumer electronics [1]. Through the integration of multiple functionalities onto a single silicon chip, SoCs offer compact, high-performance, and energy-efficient solutions for diverse applications. The complexity of SoC development lies in balancing functionality, performance, power efficiency, and cost while meeting the specific user requirements of target applications.

A System-on-Chip is a single Integrated Circuit (IC) chip that integrates all the essential components of a computing system, including processors, memory, Input/Output (I/O) controllers, and other peripherals [2]. Compared to earlier computing designs, where components are distributed across multiple chips on a Printed Circuit Board (PCB), a SoC unifies these elements into a smaller and more compact structure. This reduces the

Figure 1.1: *Block Diagram of a Sample SoC Design.*



physical size of the device, latencies and faults, power consumption, and other undesirable phenomena observed in PCB-based designs: parasitic electromagnetic effects¹, for example. SoCs often include specialized units in the context of their end applications; for example, Graphics Processing Unit (GPU), Power Management Unit (PMU), Digital Signal Processor (DSP), and other sensor-specific controller units are common SoC components in mobile device applications [3]. Figure 1.1 [4] shows a sample block diagram of a typical SoC chip for mobile applications.

SoC design and development is a multidisciplinary engineering process that requires innovation, precision, and collaboration. The process begins with architectural definition, which lays the foundation for all subsequent activities. This involves determining the processor architecture, such as RISC-V², ARM³, or a custom design, as well as selecting the required components aforementioned. A modular design approach is often

¹Electromagnetic Interference (EMI) and Electromagnetic Fields (EMF) are growing concerns in modern PCB design for computing devices. Higher clock speeds and miniaturization increase the risk of interference and crosstalk, which can disrupt device performance and raise compliance issues.

²Official website of the RISC-V Foundation: www.riscv.org

³Official website of ARM Ltd.: www.arm.com

employed, reusing standardized Intellectual Property (IP) blocks for typical functionalities, while custom-designing other modules for application-specific tasks. The process requires careful planning and iterative refinements to meet the functional and economical requirements of the target application.

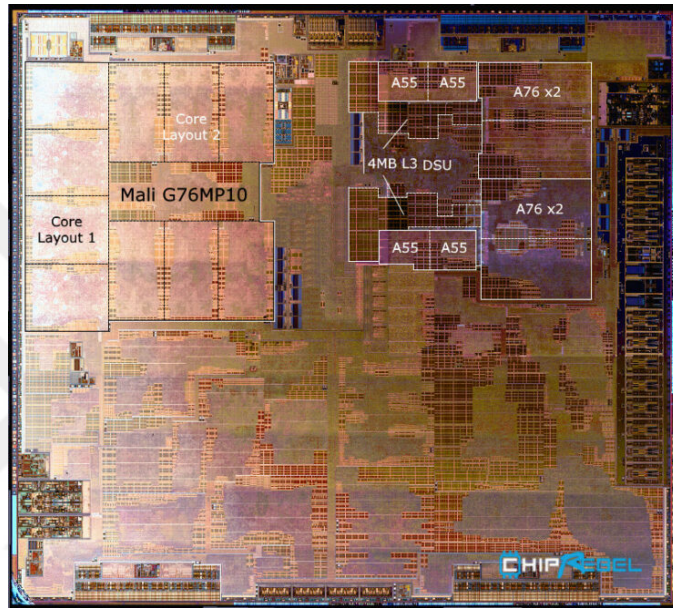
Although SoC development requires expertise across multiple technical domains, digital design is a primary focus, where engineers describe and verify logic circuits at the Register-Transfer Level (RTL) [5]. Analog and mixed-signal design ensures that analog components, such as power regulators and sensors, integrate effectively with digital systems [6]. Embedded software engineering is another critical domain, responsible for developing the software and firmware that enable the SoC to interface with operating systems and applications [7]. Physical design, typically falling under the Very-Large-Scale Integration (VLSI) category, converts the logical design into a manufacturable layout, optimizing the physical placement and routing of components to achieve Performance, Power, and Area (PPA) or Performance, Power, Area, and Cost (PPAC) targets [8]. Meanwhile, verification and validation engineers test the design to confirm its functionality and performance, while power and thermal engineers work to ensure the SoC operates efficiently within its planned energy and thermal limitations [9].

Designing and modeling the behavior of the SoC is implemented using a Hardware Description Language (HDL), such as Verilog or VHDL⁴. Afterward, dedicated software tools are used to create synthesizable RTL designs. Once the design is complete, the simulation, verification, and prototyping phases begin. During the design and simulation phase, the focus is on validating the functional correctness of the functional and RTL design against the intended architecture and specifications. Simulation tools are used to evaluate the behavior of the design under various conditions, covering functional

⁴HDL text languages are used to describe the structure or behavior of digital circuits. Verilog is simpler and C-like, making it popular in industry, while VHDL is more verbose and strongly typed, often favored for its rigor in academia and safety-critical applications.

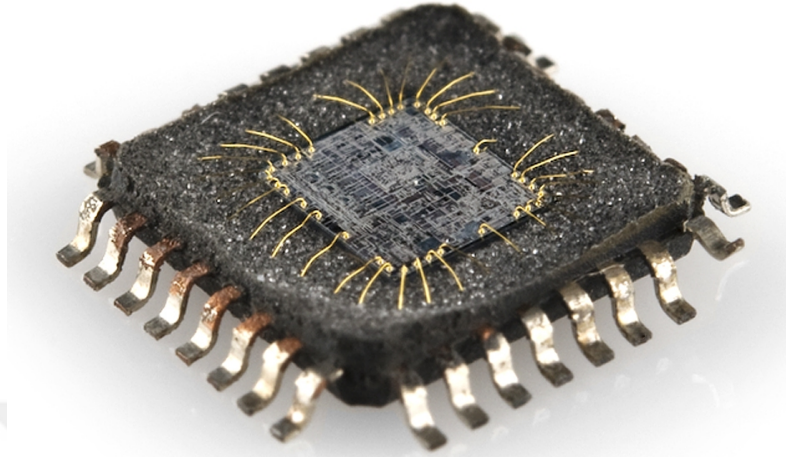
verification, edge case testing, and stress scenarios. Formal verification techniques are also applied to mathematically prove the functionality of critical modules and interfaces, ensuring compliance with design requirements [10]. This phase targets issues such as logic errors, interface mismatches, and clocking.

Figure 1.2: *Silicon Die Image of HiSilicon's Kirin 980 Mobile SoC.*



Prototyping, often conducted using Field-Programmable Gate Array (FPGA) platforms, provides a cost-effective and reiterative method to evaluate the SoC under near-real-world conditions, identifying critical functional issues before implementation. [11]. In the FPGA prototyping phase, the design is tested in a hardware environment, enabling early validation and malfunction detection. Emulation techniques may also complement FPGA prototyping, allowing for the execution of large-scale test scenarios that may be impractical to simulate or perform on hardware. Prototyping and emulation help verify system-level functionality, assess performance metrics such as throughput and latency, and evaluate interactions between hardware and software components. Any observed issues, such as hardware bottlenecks or power inefficiencies, can be refined iteratively.

Figure 1.3: *An IC Package Cutout Showing Silicon Die Connection to Pins.*



After system validation, the design enters the physical design and fabrication phase, where the logical design is translated into a physical IC layout, consisting of semiconductor, conductive, and insulation materials. Layout, routing, and packaging are optimized for manufacturability, and the final design is sent for silicon fabrication, resulting in silicon dies ready for packaging as IC chips. Figure 1.2 [12] shows the silicon die of a SoC chip, the HiSilicon Kirin 980⁵. The silicon fabrication phase introduces post-silicon validation, where Automated Test Equipment (ATE) is used to perform electrical tests, timing analysis, and parametric checks [13]. The manufactured chips undergo rigorous tests to ensure compliance with design specifications, industry standards, and reliability under operating conditions. Only after successful testing of the dies and final IC chips, the SoCs are ready to be integrated into their intended systems and deployed for use [14]. Figure 1.3 [15] shows an IC package cutout as the final SoC product.

⁵More information about the SoC can be found at www.hisilicon.com/en/products/Kirin/Kirin-flagship-chips/Kirin-980

1.1.1 Emergence of RISC-V

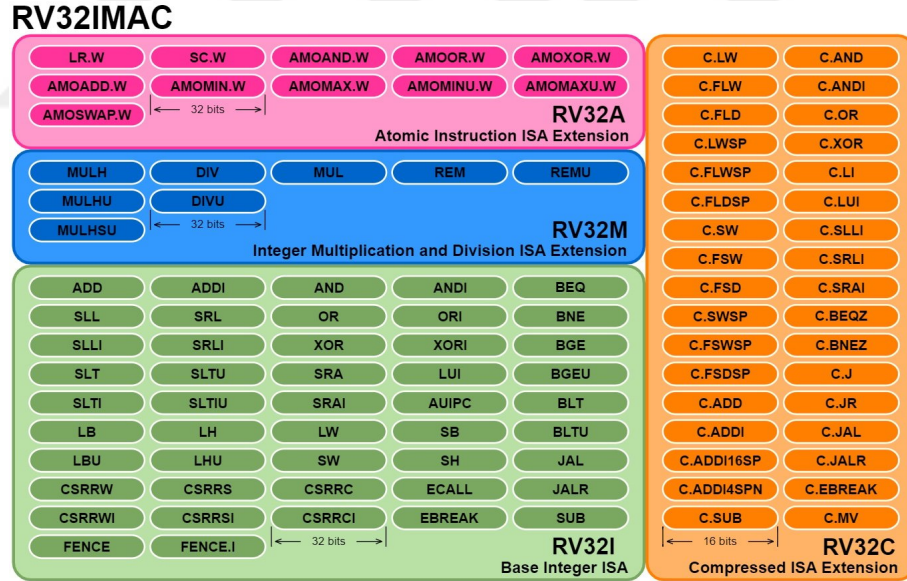
The RISC-V architecture has emerged as a significant competitor in the foreseeable future in modern computer design and SoC applications. The open architecture has revolutionized the SoC design industry, technically and economically [16]. The open source, highly modular, and highly flexible nature of RISC-V has made it an attractive alternative to proprietary architectures. The RISC-V architecture represents a unique development in processor designs, offering an open-standard and modular Instruction Set Architecture (ISA) that has gained widespread adoption across various computing domains. The simplicity and flexibility of RISC-V, stemming from a simple load-store design, encompasses 32-bit, 64-bit, and 128-bit address-space architectures. This flexibility enables its use in a broad spectrum of applications, ranging from lightweight, single-core embedded devices to complex, high-performance multi-core systems. The reader may refer to [17] for more detailed information on the RISC-V computer applications. The key members of the RISC-V Foundation can be seen in Figure 1.4 [18].

Figure 1.4: Key Members of the RISC-V Foundation.



Compared to traditional ISAs, which are often tightly coupled with specific physical hardware designs, the RISC-V ISA is a pure instruction set definition. This level of abstraction, combined with its open-hardware nature, provides CPU and SoC designers a high degree of freedom and flexibility to optimize their hardware implementations tailored to their particular use cases. The 32-bit Base Integer RISC-V Architecture (RV32I) ISA, amended as Appendix A, serves as the foundational base of the RISC-V architecture, where the suffix letter "I" stands for "Integer". The RV32I is designed for general-purpose computing, consisting of a basic set of 38 little-endian instructions for arithmetic, logical operations, memory access, and flow control. This fundamental architecture is an ideal starting point for new designers in various fields of application. Detailed definition of the RISC-V architecture can be found in its ISA manual [19].

Figure 1.5: ISA Decomposition of the RV32IMAC Architecture.



Moreover, the modularity of the RISC-V ISA allows other extensions to be implemented on top of the base ISA. For the RV32I base ISA, the C extension stands for the 16-bit compressed instructions extension, detailed in [20] and amended as Appendix B. The M extension stands for the multiplication set, the F and D extensions stand for

floating point instructions, and A for atomic instructions⁶. These extensions, amended as Appendix C, enhance the functionality of the base ISA for different computational scenarios. These extensions can be combined; for example, in an RV32IMC configuration, the CPU supports integer, multiplication, and compressed sets instructions. The 32-bit Embedded RISC-V Architecture (RV32E) ISA is a reduced architecture of RV32I for embedded applications. In RV32E, the number of general purpose registers is reduced to 16, instead of the 32 registers available in RV32I (see Appendix E). According to [19], this reduction enables 25% smaller and more power-efficient cores, while maintaining compatibility with the broader RISC-V environment. The RV32IMAC configuration shown in Figure 1.5 [21] demonstrates a combination of these modular elements of the RV32 ISA definition.

The RISC-V architecture has become an appealing choice for SoC development due to its discussed merits. RISC-V enables designers to develop a cohesive range of applications using overlapping workflows and resources. RISC-V's open standard also reduces development costs and accelerates innovation, as developers can modify or extend the ISA without licensing restrictions. Designers often combine pre-existing RISC-V cores with memory controllers, interconnects, and peripherals, leveraging a robust ecosystem of open-source and commercial tools for design, simulation, and synthesis.

RISC-V-based SoCs have found rapid applications in areas such as Internet of Things (IoT), the automotive industry, and Artificial Intelligence (AI) acceleration, where customization and resource efficiency are critical. The RISC-V architecture has gained significant traction in the global semiconductor market, with several commercial products showcasing its adaptability and potential. These implementations highlight the ability of RISC-V to address diverse application needs, demonstrating its increasing prominence in

⁶In RISC-V, atomic instructions work by performing a read-modify-write operation in a single step, ensuring no other thread or processor can access the memory being modified until the operation is complete.

the SoC landscape. The latest examples of commercially available RISC-V based SoCs, to the date of writing this document, include Raspberry Pi's hybrid RP2350 MCU⁷ and SiFive's Performance P550 core⁸.

1.1.2 Challenges in SoC Design

The design of modern SoC architectures is a complex process that involves integrating multiple components and making them work together seamlessly. Although this integration enables high performance and energy efficiency, it also presents several engineering challenges. These challenges must be addressed to ensure the functionality, reliability, and manufacturability of the SoC product. This is especially applicable in domains like embedded systems and IoT where PPAC aspects matter the most.

1.1.2.1 Balancing Requirements and Constraints

Balancing user requirements with resource constraints is a central challenge in SoC design. Modern SoCs must meet a wide range of end applications, from power-efficient embedded devices to high-performance computing frameworks, necessitating careful balancing and trade-offs in computational performance, memory capacity, and power consumption. In addition, resource allocation is often constrained by physical limitations such as die area, thermal dissipation, and manufacturing costs. Meeting user demands for high performance and rich feature sets, while adhering to these constraints and limitations, requires careful planning and innovative decisions.

⁷Official documentation: www.raspberrypi.com/products/rp2350

⁸Official documentation: www.sifive.com/cores/performance-p550

1.1.2.2 Designing, Obtaining, and Integrating Components

The selection and integration of SoC components and IP blocks represents a significant challenge. With the increasing complexity of SoCs, designers must rely on third-party or open-source IP cores for components. However, ensuring compatibility among IPs, particularly when sourced from multiple vendors, is often a challenge. Differences in standards, interfaces, and protocols often require extensive adaptation, which can introduce delays and errors into the design process. In addition, the quality and documentation of IPs vary widely and open-source solutions sometimes lack robust testing and certification compared to commercial offerings. This requires rigorous verification and customization efforts, further complicating the integration process.

1.1.2.3 Verification, Synthesizability, and Manufacturability

Verification of the functionality of the overall integrated system is a demanding process; exhaustive testing of SoC functionality through simulation and hardware prototyping is essential to identify and resolve design bugs. With the increasing complexity and heterogeneity of SoC designs, the verification process has become a significant bottleneck in the development cycle and is becoming a key research interest for automation. Further reading on verification challenges can be found in [22]. Ensuring synthesizability and manufacturability is equally demanding; SoCs are built up using multi-sourced IPs that must eventually boil down into a single, homogeneous physical IC structure. Achieving synthesizability and manufacturability across all components and their interfaces requires careful design practices, post-synthesis verification, and strict adherence to hardware constraints. Advancing an SoC design into the realization phase requires extensive verification processes, where automation is often used.

1.1.2.4 Reducing Size and Complexity

Reducing the physical footprint of synthesized SoC designs is a primary goal in modern electronics, particularly for applications that demand compact, energy-efficient, and cost-effective solutions. Achieving a minimal design footprint involves optimizing all major components in both the design and synthesis phases, including CPU cores, memory interconnects, peripherals, and other components. Among these elements, on-chip memory structures often prove to be a significant factor, impacting both the size and power consumption of the SoC chip. Memories typically occupy a substantial portion of the chip size, making their optimization critical to reducing the overall size of the SoC product. Further reading can be found in [23].

Efficient on-chip memory organization is key to this effort. Systems must balance memory density with performance and cost considerations, tailoring solutions to specific workloads and architectural constraints. Hierarchical memory architectures, which combine small and fast caches with larger and slower main memory, are frequently used to balance access speed and area efficiency. Advanced techniques, such as selective memory banking, shared memory regions, and multi-port memory interfaces, further contribute to reducing memory-related overheads while maintaining performance [23]. Other non-hardware tools include developing more efficient ISA definitions to reduce memory requirements. This challenge in particular, observed in the first part of the research in Chapter II, has been a critical motivating factor for establishing the second part of the research in Chapter III.

1.1.2.5 Data and Timing Integrity

When different components and interfaces are integrated into an SoC chip, ensuring data and timing integrity is also a key challenge in modern SoC design. The increasing complexity of SoC designs and the use of smaller manufacturing processes can lead

to data and timing integrity hazards. Multiple clocks must work together seamlessly in a modern SoC design to ensure proper functionality of the different components and proper transactions of data. Timing integrity ensures that signals within the SoC reach their destinations within the allowed delay and deformation margins, enabling reliable operation at the intended system clock speed. In modern SoC designs, when multiple clock domains, complex time-dependent modules, and precise signal processing components are integrated, achieving precise timing becomes increasingly difficult. Factors such as signal delays, clock mismatches, and timing instability can cause errors, data loss, or degrade performance. These issues are even worsened by the physical variations introduced by the manufacturing process at smaller technology scales, adding further uncertainty to signal propagation and timing integrity, making timing analysis even more challenging. The work in [24] demonstrates a case of verification of the integrity of the reset signal.

1.1.2.6 Power and Energy Efficiency

An increasingly emerging challenge in modern computing systems, including SoC designs, is balancing power consumption and performance [25]. In portable applications, such as mobile and node devices, minimizing power consumption is critical in order to extend battery life, but this must be achieved without compromising computing performance. The rise of battery-powered devices, such as smartphones, smart "things" and IoT nodes, has further highlighted the demand for efficient energy management, the survey [26] highlights energy management in mobile devices. In addition, the growing global concern about energy sustainability and efficiency calls for designs that also reduce the overall carbon footprint of electronic systems. Different technical practices are used in modern SoC design to achieve these goals.

1.1.2.7 Scalability and Customization

Scalability and customization are becoming an advantage and a complication of SoC design. Modular ISAs, such as RISC-V, provide flexibility in adapting designs to specific applications while maintaining compatibility with future upgrades. However, designing systems that remain scalable and customizable, where components can be easily integrated or replaced without disrupting the overall system, presents challenges. Ensuring such modularity, reusability, and proper interfacing requires careful architectural planning and future forecasting. Scalable interconnects play a key role in supporting the addition or removal of components and functionalities. Additionally, the quality of the RTL design itself of individual modules, through parameterization and proper documentation, plays an important role in the modifiability of different SoC components.

1.1.2.8 Software Toolchains

Robust SoC development introduces challenges in establishing reliable toolchains for tasks such as design, synthesis, implementation, testing, and firmware development. Toolchains for proprietary architectures, like ARM, are mature and widely supported, but for many emerging architectures, toolchains are mostly still evolving, lacking such reliability and support; designers often face issues such as incomplete or incompatible tools, often requiring additional workload and manual intervention. Automation is an ever-evolving assisting tool in managing this growing complexity: automated tools, including modern AI and machine learning technologies, help reduce workloads and human-caused errors. However, their effectiveness often relies on their compatibility with the chosen architecture. Companies such as Synopsys⁹ have invested heavily in AI-driven and automated SoC workflows, as demonstrated in [27].

⁹Synopsys, Inc. is a global leader in EDA and semiconductor IP, offering tools for chip design, verification, and testing, as well as a broad portfolio of processor, interface, and memory IP solutions.

1.2 Research Objectives and Contributions

This thesis explores the planning, design, integration, and prototyping of a RISC-V-based MCU SoC for embedded applications, with a focus on investigating and optimizing at least one key aspect of the engineering challenges previously identified. The research is conducted in two successive stages: the first emphasizes the overall integration of the SoC, providing a comprehensive view of its design and functionality, while the second phase concentrates on optimizing a specific challenge arising from the initial implementation, which is size reduction.

1.2.1 *Integrating a Prototype Microcontroller SoC*

The first objective of this research is to design and prototype the Özyeğin University (OzU) SoC, or "OzU SoC", a minimal functional Microcontroller Unit (MCU) SoC prototype tailored to meet specific embedded application requirements for Özyeğin University's Electrical and Electronics Engineering Department, particularly for use in electrical metering applications. The OzU SoC is a lightweight single core MCU, consisting of a set of essential SoC components: General Purpose Input/Output (GPIO) ports, an Analog-to-Digital Converter (ADC) module, Timer/Counter modules with Pulse-Width Modulation (PWM) generators, and a set of basic serial communication interfaces: Universal Asynchronous Receiver-Transmitter (UART), Inter-Integrated Circuit (I2C), and Serial Peripheral Interface (SPI). The OzU SoC is planned with on-chip program and data memories, constructing a complete minimal SoC prototype, alongside other auxiliary functional and control units.

The OzU SoC represents a research contribution as a RISC-V-based SoC prototype designed specifically to address the demands of a real-life embedded application. The design serves as a configurable and extendable testbed for exploring further architectural improvements, performance evaluation, and possible areas of design automation.

The final deliverables of the OzU SoC is its customizable RTL design, demonstrated with limitations through FPGA prototyping. The course of this development demonstrates the practical projection of several methodologies and challenges discussed in the previous section onto an application case, as well as bridging the concepts between the theoretical and practical perspectives of SoC design and development.

1.2.2 Optimizing Machine Code Efficiency

Following the development of the OzU SoC, as an RTL design and an FPGA prototype, as the first part of this research, it became evident that one of the most significant challenges observed is minimizing the size of on-chip memories to reduce the physical size of the SoC, as an FPGA prototype or an ASIC chip. Previously detailed in 1.1.2.4, memories occupy a substantial portion of the physical chip area, directly contributing to the size, cost, and power usage of the target application device. With the ongoing shortage of silicon since the COVID-19 pandemic [28], and the increasing demand for efficient utilization of global silicon reserves, optimizing this aspect became a critical focus and a secondary objective of this investigation.

Consequently, the Novel Optimized Variable-Length Inclusive ISA (NOVLI-ISA), an optimized RV32I ISA extension that focuses on reducing the size of machine code, became a key research contribution of this thesis. The NOVLI-ISA is designed, implemented, and tested to address the challenges encountered with memory size limitations in resource-constrained SoC development and applications. As a compatible extension to the base RV32I/E ISA, the NOVLI-ISA introduces a full definition of variable-length instructions, designed to reduce machine code storage requirements and increase its representation efficiency, without compromising functionality or performance. The extension is designed based on key guidelines inherited from the RISC-V design philosophy, such

as simplicity, modularity, compatibility, flexibility, and the introduction of minimal additional complexities to both hardware and software components of SoC design. The NOVLI-ISA was presented as a conference paper, evaluated through theoretical analysis, practical implementation, and application evaluation. This contribution not only demonstrates the feasibility of architectural-level optimizations in reducing memory size requirements but also highlights the potential of customizable ISAs to enhance the efficiency and adaptability of modern SoC designs, opening further research directions and possible optimizations from both the hardware and software perspectives.

1.2.3 Exploring Areas of Automation

Several workflow aspects of this research highlight the need for automation in SoC design, development, and validation workflows, facilitating workload, reducing possible human-made errors, and enhancing overall efficiency. The general streamlining of the OzU SoC design workflow, its requirements of customizability and scalability, added to the process of optimized machine code generation and evaluation encountered in the NOVLI-ISA, pinpoint areas where automation can be used. Addressing these key areas where automation is applicable, the thesis highlights, proposes, and implements automated processes in both parts by identifying algorithmic, time-consuming, and error-prone tasks. These key areas where automation is applicable can be summarized in HDL generation and customization, machine code generation and evaluation, verification, and feasibility and performance evaluation. In the first part, OzU SoC, automation is implemented for HDL generation and proposed for customizing HDL designs from the base prototype, while in the second part, NOVLI-ISA, automation is used to generate batches of machine code and evaluate several metrics of them, while other methods are also proposed for further automation.

1.3 Structure of the Thesis Report

This document is organized into four chapters. This chapter introduced the technical scope and background of the thesis, providing an overall technical context for the following context, without delving into the micro-technical details, concluding with the research objectives and contributions. Chapter II discusses the first part of this research: the development, implementation, and results of the OzU SoC, covering a comprehensive review of related literature and industry contributions, as well as introducing the technical methodologies involved where they have been used. Chapter III, the second part of this thesis, similarly addresses the design, implementation, and evaluation of the NOVLI-ISA, similar work in the literature, and the fine details of its definition, prototyping, testing, and evaluation. Finally, the outcomes, contributions, and possible future research directions of the OzU SoC and NOVLI-ISA are summarized in Chapter IV, concluding and summarizing the work presented in this thesis.

2. INTEGRATION OF CUSTOMIZED SOCS

This chapter, concerning the OzU SoC as the first part of the research, first provides a review of relevant academic and industry projects in the domain of SoC integration in embedded applications, particularly based on the RV32 architecture as the architecture of interest. The chapter then explains some essential methodologies of the SoC design and development workflow, particularly what is used in the development of the OzU SoC. The context then delves into the technical details of implementing the OzU SoC, starting from user requirements to final implementation and testing of the prototype, concluding with potential areas of implementing workflow automation. Additionally, the final deliverables of the OzU Soc are summarized in Chapter IV.

2.1 Examples of Customized SoCs

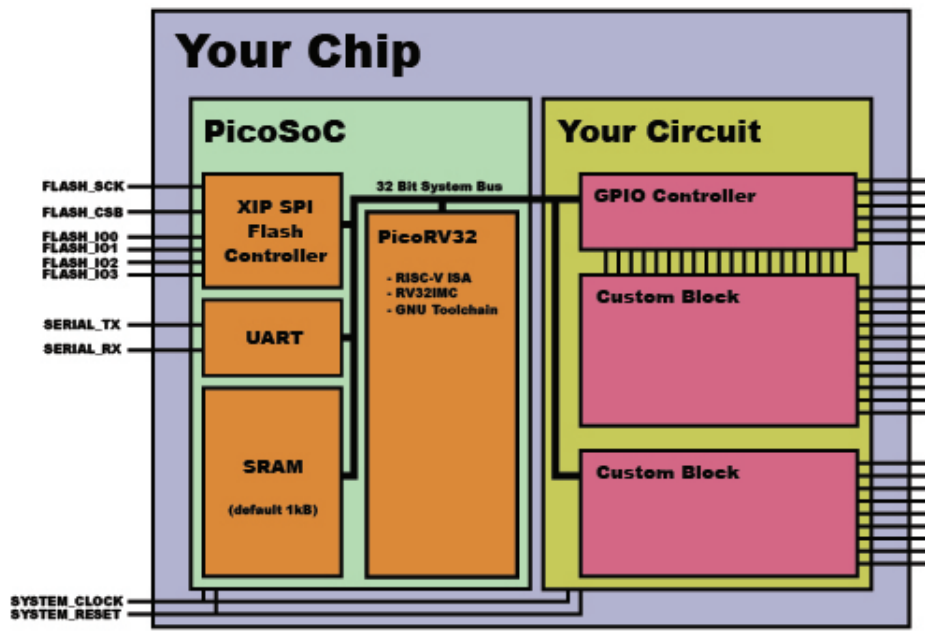
The RV32 architecture, with its common ISA extensions discussed in Chapter I, provides an ideal foundation for creating compact, efficient, and highly customizable SoC designs. In the following, notable examples of RV32-based SoC designs are reviewed, highlighting their architectures, applications, and contributions to the field.

PicoRV32 [29], developed by YosysHQ GmbH¹, an open source EDA company [30], is a configurable and size-efficient CPU core that implements the RV32IMC/E architecture. The PicoRV32 has customization options to adapt it as RV32E, RV32I, RV32IC, RV32IM, or RV32IMC. Designed for performance, while achieving efficient use of hardware resources, it achieves high frequencies, up to 700 MHz on Xilinx FPGAs, while occupying as few as 750-2000 Look Up Table (LUT)'s, depending on configuration. It is particularly suited for auxiliary roles in FPGA and ASIC designs, offering features such as optional interrupt handling, a customizable coprocessor interface, and compatibility with

¹Official website: www.yosyshq.com

AXI4-Lite or Wishbone interconnect standards. Licensed under the permissive Internet Systems Consortium (ISC) license², it provides an integration-friendly CPU architecture in open-source-based SoC designs, offering performance, flexibility, and configurability. Gowin PicoRV32 [31] is a commercial soft processor core based on PicoRV32; it extends the CPU with additional serial interfaces and cache memories.

Figure 2.1: Block Diagram of YosysHQ PicoSoC.

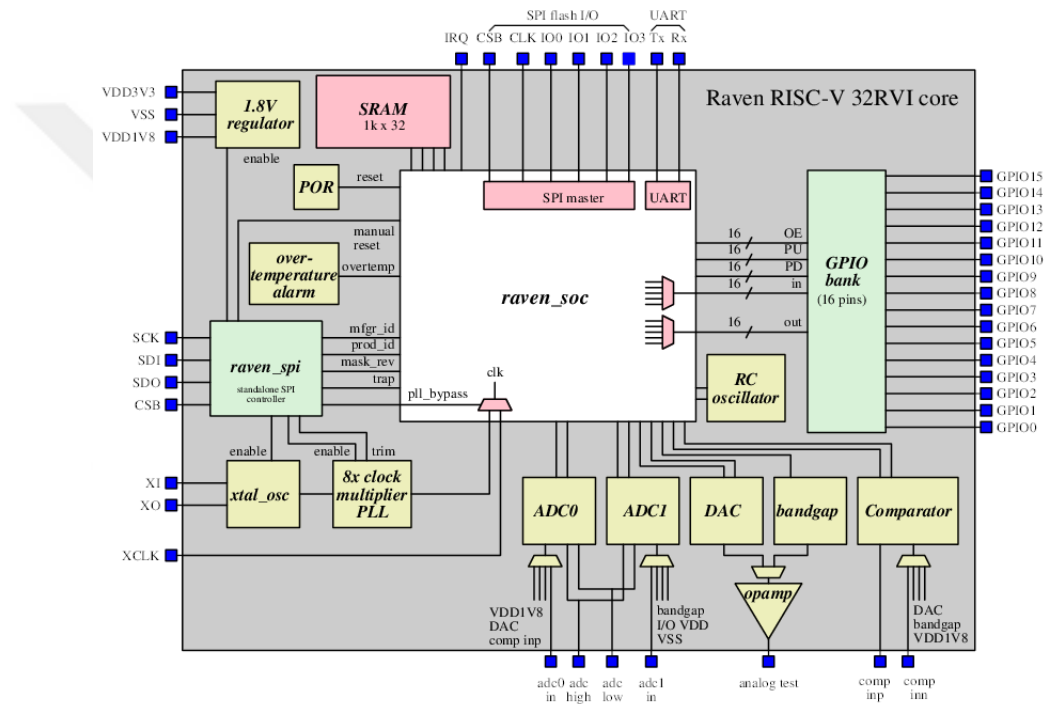


The PicoSoC [32] is an extendable, minimalist single core SoC design built around the PicoRV32 core. It is designed to enable the integration and prototyping of essential SoC components using PicoRV32 as a core. The PicoSoC features a memory-mapped, eXecute-In-Place (XIP) SPI flash interface that allows for direct execution of operational program from an external flash memory, significantly reducing the prototype footprint in resource-constrained environments. Additionally, the PicoSoC has a single-bank data

²The ISC license is a permissive free software license published by the Internet Software Consortium, now called Internet Systems Consortium (ISC). It is functionally equivalent to the simplified BSD and MIT licenses, but without language deemed unnecessary following the Berne Convention

SRAM and a single UART interface, in addition to its compatibility to receive for more user integrations. Figure 2.1 shows the block diagram provided in [32], showing possible additional user integrations into the PicoSoC in the "Your Circuit" section. The design emphasizes modularity and ease of customization, making it a foundational design to be used as a starting point for developing application-specific SoCs based on the PicoRV32.

Figure 2.2: Block Diagram of Raven SoC.



A fabrication-proven implementation of the PicoSoC is Raven [33]. Developed by Efabless [34], Raven as shown in Figure 2.2 from [33], integrates the PicoRV32 core with an SRAM, a programmable clock, GPIO port, ADCs, a Digital-to-Analog Converter (DAC), a comparator, and other additional hardware-specific auxiliary units. It also includes an Resistor-Capacitor (RC) oscillator, power regulator, and an on-chip temperature sensor. Raven provides a foundation for a manufacturable PicoSoC design, providing the hardware design with its relevant simulation and firmware development tools.

A practical example of implementing the RV32 architecture for application-specific SoC designs is the piXo project by Omar Chadib [35], a PicoSoC-based SoC for image processing on FPGA platforms. Focused on approximate computing, it uses fewer computational resources to perform image processing tasks, such as approximate multiplication, with results displayed on a monitor via a VGA controller.

RISC-V architectures are also common to find as multi-core implementations, due to their flexibility and modularity. The Parallel Ultra-Low Power (PULP) [36] is an open source multi-core computing platform, developed collaboratively by ETH Zurich and the University of Bologna. It is tailored for edge-node IoT applications, offering advanced data processing capabilities for sensors like cameras, accelerometers, and microphones. PULP incorporates a tightly coupled processing cluster, autonomous I/O subsystems, and a scalable memory architecture, supporting high-efficiency hardware accelerators. The platform supports interfaces such as SPI, I2C, and Inter-Integrated Circuit Sound (I2S) and is complemented by a comprehensive Software Development Kit (SDK) and simulation environment, facilitating design, testing, and customization for diverse applications. The reader may refer to [37] for a detailed academic review of the PULP platform.

The SiFive Freedom repository [38] serves as a further comprehensive reference for practical SoC integration based on the RISC-V architecture. The repository provides RTL designs for the Freedom E300 and U500 platforms. The Freedom E300 platform demonstrates an embedded SoC that integrates the SiFive E31 core, which implements the RV32IMAC architecture, targeting applications requiring lightweight and efficient computing [39]. It provides a foundation SoC platform to map onto Digilent's Arty FPGA [40], enabling developers to generate Verilog files and configuration memory files that support the integration of boot ROM and custom peripherals. The Freedom U500 platform represents a more advanced SoC implementation designed for high-performance applications, based on SiFive's U54-MC core that implements RV64GC architecture [41].

The U500 integrates with the Xilinx VC707 FPGA kit [42] and supports bootable Linux images, illustrating how RISC-V can scale for complex systems. SiFive³ is a leading commercial player in the RISC-V ecosystem, offering a wide range of platforms and products that implement the RISC-V architecture.

2.2 SoC Design and Development Workflow

The development of a SoC prototype, briefly reviewed in Chapter I, involves a systematic approach that integrates multiple subsystems within a unified architecture, considering different technical and economical aspects. From receiving the user requirements, functional planning is the starting point in SoC design. In this phase, system functionalities are broken down into discrete, manageable components and blocks such as CPU cores, memory hierarchies, and peripheral interfaces. High-level modeling tools such as MATLAB-based Simulink⁴ or SystemC⁵ can sometimes be used to simulate and validate architectural choices before RTL design begins, but not necessarily for smaller, simpler SoC designs. Functional partitioning ensures that the system is functionally complete and ready for further detailed design stages.

The CPU, single-core or multi-core, is the central component of a SoC for general purpose computing⁶. It serves as the primary computational engine, performing the main operations and coordinating interactions between all system components. Its selection is the first step in SoC design, as it is the main influence of the SoC's functionality and PPAC constraints. The tightly coupled components of the CPU, such as registers and the Core-Coupled Memory (CCM), play a vital role in optimizing the CPU's functionality

³Official website: www.sifive.com

⁴Simulink enables model-based design for SoC development, supporting simulation, verification, and hardware-software co-design.

⁵SystemC extends C++ with libraries for hardware modeling, enabling concurrent execution, timing, and event-driven simulation essential for system-level design.

⁶Other SoCs may be application-specific and have no CPUs, such as those implementing GPUs, or those used for radio communication or signal processing.

and should be planned properly. The CPU and its closely integrated elements define the processing capabilities and possible applications of the SoC, establishing the foundation for the design steps that follow. For multi-core systems, CPU planning involves determining the appropriate balance between shared and private resources, ensuring efficient inter-core communication, and minimizing contention in shared memory or interconnects. Multi-core architectures can significantly enhance parallelism and scalability, but they present several challenges in consideration of workload partitioning and synchronization mechanisms to achieve optimal performance and PPAC balance.

The next step is to standardize on-chip communication protocols, also called interconnects or buses, as well as to determine the dataflow architecture, i.e., Harvard or Von Neumann⁷. Examples of on-chip interconnect protocols are ARM's Advanced Microcontroller Bus Architecture (AMBA), Advanced Extensible Interconnect (AXI), and Advanced Peripheral Bus (APB)⁸. Other examples are OpenCore's⁹ Wishbone, SiFive's TileLink, and OpenCAPI¹⁰. Other vendor-specific protocols exist but can only be used within their manufacturer's IP range. Choosing the interconnect in the SoC design is a critical decision, as it directly impacts the system's performance, reliability, power efficiency, scalability, and communication integrity and latency. The decision is influenced by factors such as bandwidth and latency requirements, potential future integrations, licensing, and the nature of data for the planned functionalities. Figure 2.3 [43], full diagram in Appendix F, shows some interconnects implemented in STMicroelectronics STM32F40xxx MCU chips, which demonstrates a multi-bus example.

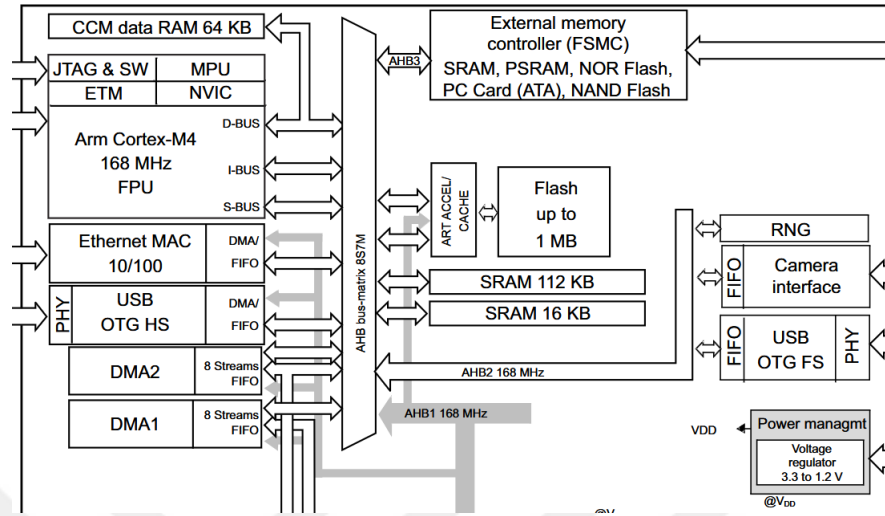
⁷The first separates program and data interconnects, while the latter uses a unified dataflow interconnect.

⁸More details on AMBA 4 family at www.arm.com/architecture/system-architectures/amba/amba-4

⁹OpenCores is an open-source platform providing a repository of reusable IP cores for hardware design, official website is www.opencores.org

¹⁰Developed by IBM, supported by the the OpenCAPI Consortium, OpenCAPI (Open Coherent Accelerator Processor Interface) is an open standard designed to enable high-speed, coherent communication between processors, accelerators, and memory, enhancing performance and flexibility in modern computing architectures.

Figure 2.3: *Interconnects Architecture in STM32F40xxx MCUs.*



Determining the interconnect and dataflow architecture, memory planning comes next, ensuring that area resources are optimally allocated for optimal system performance and efficiency. ROMs are typically partitioned into sections for different purposes: a partition is reserved as vendor-programmable, One-Time Programmable (OTP) memory, which could contain self-test or startup routines to verify or initialize hardware components during power-up. Another partition could be assigned for a user-programmable bootloader code to initialize the system and load the operating environment. An example of the usage of OTP and bootloader memories can be found in [44]. Finally, the largest segment of the ROM contains the main operational program, or firmware, which is a user-programmable partition. Data memories, typically Static Random Access Memory (SRAM), are carefully sized and placed for high-speed temporary storage of application data, supplemented with cache memories if optimal. Persistent storage memories

like Flash and Electrically-Erasable Programmable ROM (EEPROM)¹¹ can also be implemented to store configuration settings or persistent data, somewhat equivalent to data drives on personal computers.

Moreover, external memory interfaces have become popular in modern SoC designs, with the ever-increasing size requirements of program and data storage. High-speed interfaces like SPI, Dual Serial Peripheral Interface (DSPI), Quad Serial Peripheral Interface (QSPI), and Enhanced Serial Peripheral Interface (eSPI) are typical for connecting external memories used for both firmware and data storage, typically a flash. Other examples of more advanced external memory interfaces are: Flexible Static Memory Controller (FSMC)¹², Secure Digital Input Output (SDIO)¹³, and parallel interfaces are also used for higher bandwidth needs, such as connecting external high-performance RAMs, and video playback or recording memories. I2C is commonly used for low-speed connections, such as for external EEPROMs or small configuration memories.

Having the CPU and memory architecture planned and interfaced, composing the solid core and the minimal computing environment of a SoC, and integrating other SoC components with this core, mainly peripherals, is what comes next. The peripherals enable the CPU to interact with external devices and communication channels. This integration involves connecting peripherals through standard or dedicated interfaces that allow efficient data exchange, typically via memory-mapped or port-mapped communication. In load-store architectures like RISC-V, peripherals are commonly accessed through memory-mapped I/O, where peripheral registers are assigned specific memory addresses

¹¹Flash memory erases and writes data in blocks, making it suitable for high-capacity storage, while EEPROM operates at the byte level, ideal for frequent updates to small data sets. Flash offers larger capacity but slower individual updates, whereas EEPROM is optimized for precision writes at the cost of storage size. [45]

¹²The FSMC interface in STM32 microcontrollers supports high-speed interfacing with external memories like SRAM, NOR Flash, and LCDs. For detailed specifications, the reader can refer to the STMicroelectronics AN2784 application note.

¹³The SDIO interface enables high-speed data transfer with Secure Digital (SD) peripherals, such as memory cards and wireless modules.

within the CPU's address space, and handled like a memory space. The CPU interacts with these peripherals by executing standard load and store instructions, using the same interconnects and instruction set designed to access regular memory. This unified approach simplifies the CPU design and enhances scalability and flexibility, since no special instructions are needed for operating peripherals. Alternatively, autonomous Direct Memory Access (DMA) controllers can be implemented to transfer data between peripherals and memory without involving the CPU for each transaction. DMA is particularly used in applications that require high-throughput data transfers, such as audio or video streaming, as it reduces CPU overhead and allows the processor to focus on other tasks while data transfer occurs autonomously [46]. When integrating peripherals, it is crucial to consider factors such as licensing, compatibility with interconnects and architecture, and reliability. Figure 2.4 [47] shows a practical case of memory-mapped addresses in Microchip's SAM D21 MCUs, showing both on-chip memories and peripheral space.

Figure 2.4: *Memory-Mapped Space Organization in SAM D21 MCUs.*

Table 10-1. SAM D21 Physical Memory Map⁽¹⁾⁽¹⁾

Memory	Start address	Size					
		SAMD21x18	SAMD21x17	SAMD21x16	SAMD21x15	SAMD21x16L	SAMD21x15L
Internal Flash	0x00000000	256 Kbytes	128 Kbytes	64 Kbytes	32 Kbytes	64 Kbytes	32 Kbytes
Internal RWWEE Emulation section ⁽²⁾	0x00400000	-	4 Kbytes	2 Kbytes	1 Kbytes	2 Kbytes	1 Kbytes
Internal SRAM	0x20000000	32 Kbytes	16 Kbytes	8 Kbytes	4 Kbytes	8 Kbytes	4 Kbytes
Peripheral Bridge A	0x40000000	64 Kbytes	64 Kbytes	64 Kbytes	64 Kbytes	64 Kbytes	64 Kbytes
Peripheral Bridge B	0x41000000	64 Kbytes	64 Kbytes	64 Kbytes	64 Kbytes	64 Kbytes	64 Kbytes
Peripheral Bridge C	0x42000000	64 Kbytes	64 Kbytes	64 Kbytes	64 Kbytes	64 Kbytes	64 Kbytes
IOBUS	0x60000000	0.5 Kbytes	0.5 Kbytes	0.5 Kbytes	0.5 Kbytes	0.5 Kbytes	0.5 Kbytes

Finally, additional auxiliary units should be implemented to enhance the functionality, reliability, and security of a SoC. These units provide essential control and support

functions for system management, monitoring, and protection, ensuring that the SoC operates correctly, efficiently, safely, and securely in various scenarios. For example, a Watchdog Timer (WDT) is commonly included to monitor the health of the system and recover from software failures by resetting the processor if it does not respond within a predefined interval; an example application can be found in the application note [48]. Components like the Nested Vectored Interrupt Controller (NVIC) are employed to manage complex multi-interrupt systems, allowing prioritized and efficient handling. Security modules are also becoming essential with the emergence of cyber threats in the context of hardware environments. These include features such as hardware-based cryptographic accelerators, secure boot mechanisms, and tamper detection. PMUs may be added to optimize energy consumption by dynamically adjusting voltage and frequency. Other auxiliary units, such as clock generators, Real-Time Clock (RTC), and debug interfaces, further contribute to the SoC's functionality, robustness, and serviceability.

Once the concept and architectural design of the SoC have been outlined, the design process begins by translating these plans into a structured technical workflow. The first step involves writing RTL code using an HDL like Verilog. This code defines the hardware behavior and structure, in terms of sequential logic, combinational circuits, and registers. Verification of this RTL design is performed through behavioral simulations, where test cases validate the functionality of each module by performing a dedicated testing machine code, usually embedded in sample Verilog-implemented OTP ROMs or loaded from files. Once the functionalities are thoroughly tested, the design moves to synthesis, where the HDL code is translated into a gate-level netlist¹⁴. This step prepares the design for the implementation phase, which involves placing and routing the netlist to create a physical layout of implementing FPGA resources that satisfies predefined timing,

¹⁴In the context of FPGA, a netlist defines the circuit's components and connections, enabling design mapping, placement, and routing onto the FPGA structure.

resource utilization, and performance constraints. In this thesis, Xilinx Vivado¹⁵ and Verilog HDL are used to design and implement the OzU SoC and the NOVLI-ISA. Vivado provides a comprehensive suite of tools that support the entire workflow, from RTL development to FPGA programming. During synthesis, the Verilog code is converted into a netlist compatible with the target FPGA, ensuring that it meets its resources constraints. In the implementation phase, Vivado performs placement and routing to optimize the design for efficient operation on the FPGA, under the aforementioned predefined performance constraints. Finally, a bit stream file is generated for programming the FPGA, enabling the SoC to be tested in near-real-world conditions.

Although HDL remains the main tool in SoC design, alternative methods have emerged to enhance flexibility and speed up development. High-Level Synthesis (HLS) allows developers to describe SoC functionality using high-level languages like C, C++, or SystemC, which are automatically converted to RTL by tools such as Xilinx Vitis HLS. Graphical design tools such as Simulink and LabVIEW FPGA enable block diagram-based designs that are particularly effective for applications like signal processing. IP-based design uses pre-verified IP blocks that are integrated using tools like Vivado IP Integrator, reducing development time and complexity. The reader may refer to [49] for more details of the latest methodologies in SoC design.

Developing and implementing the SoC's firmware, typically written in low-level languages like C or assembly, involves the use of specialized software toolchains, including compilers, assemblers, linkers, and debuggers, to generate machine code optimized for the target SoC architecture. Programming tools such as flash programmers or in-circuit debuggers, e.g., Joint Test Action Group (JTAG) or Serial Wire Debug (SWD) interfaces¹⁶, are used to load the firmware into the SoC program memory and monitor

¹⁵Xilinx Vivado, a leading FPGA design suite, is now part of AMD's portfolio following its acquisition of Xilinx in 2022.

¹⁶JTAG and SWD are interfaces for debugging and programming embedded systems. JTAG supports

its operation within the CPU's environment. If present, firmware components are developed and tested alongside the hardware using simulators or Hardware-In-the-Loop (HIL) setups to verify functionality and ensure reliable operation. Firmware bridges the gap between hardware and higher-level software, such as operating systems or applications, by providing essential services such as bootloading and peripheral drivers.

FPGAs are widely used for prototyping SoCs because of their affordable cost, scalability, and ability to sufficiently emulate the target environment of a SoC design. FPGAs enable designers to implement the RTL description of the SoC and test its operation within a hardware environment, allowing early identification of design flaws, bottlenecks or integration issues. FPGA development boards can accommodate a wide range of peripherals, memory interfaces, and custom logic, making them ideal for verifying functionality in application-specific scenarios. High-performance FPGAs, such as those from Xilinx or Intel, provide the resources necessary to prototype even complex multi-core or heterogeneous SoCs¹⁷. In addition to FPGA prototyping, other methods for SoC validation¹⁸ include software emulation, virtual prototyping using SystemC for system-level simulations, and rapid prototyping with off-the-shelf development boards like Raspberry Pi for proof-of-concept designs. Hybrid approaches, which combine FPGAs and virtual platforms, balance cost and accuracy. These methods can be complementary to FPGA prototyping based on project needs, available resources, and development stages.

After successful prototyping and testing, the SoC design undergoes additional refinement and preparation for its realization as an ASIC product. This phase involves

testing and multi-core debugging, while SWD is a simpler, two-wire option for ARM Cortex-M-based devices.

¹⁷A heterogeneous SoC integrates a complex and diverse set of processing units, such as CPUs, GPUs, DSPs, and dedicated accelerators, onto a single chip.

¹⁸In the context of SoC and digital systems design, verification ensures the design is implemented correctly according to specifications, focusing on functional correctness. Validation, on the other hand, confirms that the final system meets intended requirements, often involving testing on actual hardware to assess real-world performance.

converting the validated RTL design into a final layout suitable for silicon fabrication, a process that includes detailed physical design steps such as placement, routing, and timing analysis. Power optimization, Design Rule Checking (DRC), and Layout-Versus-Schematic (LVS) verification ensure that the design meets both functional and manufacturability standards. The design is then sent to a semiconductor foundry for fabrication, where the chip is manufactured on silicon wafers using advanced lithography techniques. Post-fabrication steps include packaging, testing, and validation under real-world conditions to confirm functionality and performance. Following successful validation, the ASIC is prepared for mass production and deployment into its target application. Although these stages are critical to the life cycle of an SoC, the realization of the ASIC and its deployment are beyond the scope of this thesis, which focuses on the design, implementation and validation of the SoC as an RTL design, ending at the FPGA stage.

2.3 Designing the OzU SoC Prototype

Having established the general workflow for SoC design and explored various relevant methodologies, this process is projected onto the development of the OzU SoC prototype, the first part of this research, with its specifications briefed in Chapter I. This section goes through the step-by-step design process of designing and implementing the OzU SoC: from user requirements to simulation and prototyping.

2.3.1 User Requirements

The OzU SoC is a dedicated lightweight, single-core MCU, designed for embedded applications, intended for use in electrical power metering applications in particular. The OzU SoC is specified by its end user as 46 functional and electrical pins, serving basic functions and power requirements for the intended application. The detailed user requirements and specifications that guided the design process are provided in Table 2.1.

Table 2.1: *Original User Requirements for OzU SoC Pins.*

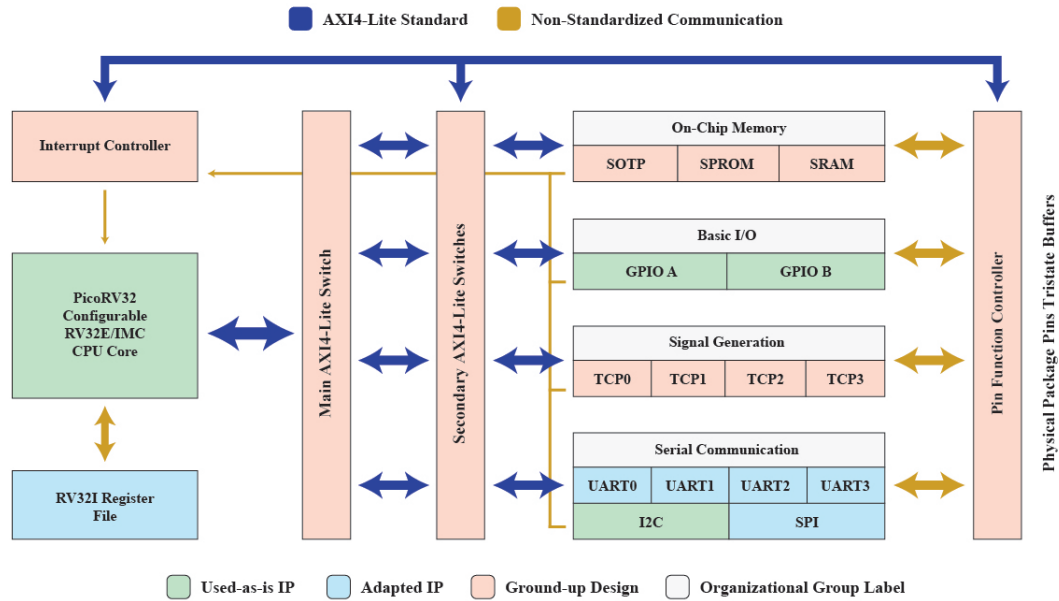
Function	Number of Pins	Description
GPIO	12	General purpose digital I/O
CS	10	Chip Select outputs for SPI
ADC	6	6-channel analog input pins
PWM	4	Generated from Timer/Counter modules
UART	6	3 RX and TX pairs
I2C	2	SCL and SDA pins
SPI	3	SCK, MOSI and MISO pins
Clock	1	Clock input
Power	2	VDD and VSS

2.3.2 System Architecture

As a minimalist design with a basic set of peripherals, and an MCU intended for use in embedded environments where PPAC constraints balance is important, designing the OzU SoC as a single-core SoC is sufficient. YosysHQ’s PicoRV32 and PicoSoC projects, reviewed earlier, surfaced as the top candidates for the base architectural design for the OzU SoC. These implementations have been extended and integrated in several other mentioned works, proving their versatility, flexibility, reconfigurability, and robustness across different applications, added to their open-source licensing advantage. The PicoRV32, which serves as the core of OzU SoC, is highly configurable, offering compatibility with the RV32C and M ISA extensions, or a reduction to the embedded RV32E architecture. The PicoRV32 offers its configurability through Verilog parameterization, detailed in Appendix G as obtained from [29]. It offers Plug-in Custom Peripheral Interface (PCPI)-based arithmetic co-processors extensions, interrupt handling, trace data, memory addressing, RISC-V Formal Interface (RVFI) debugging, and other supplementary Arithmetic Logic Unit (ALU) functions. Parameterization, in Verilog, allows designers to define flexible hardware modules by specifying parameters that can be adjusted

during instantiation, such as enabling or disabling features, passing values, or naming objects. Parameterization, supplemented with other Verilog design techniques, enables the creation of reusable, configurable, and scalable designs tailored to varying requirements. These techniques in Verilog 2001 are explained in detail in [50]. Little modifications have been made to the original PicoRV32 IP in the integration process, such as fixing a minor bug in Interrupt Request (IRQ) handling, and revising the provided RV32I register file design. The PicoRV32 offers three interconnect protocol options: AXI4-Lite, Wishbone, and a simple non-standardized memory interface.

Figure 2.5: *Simplified Block Diagram of the OzU SoC.*



It has been decided to use the AXI4-Lite as the main interconnect of the OzU SoC. The AXI4-Lite protocol is a version of the full ARM AXI protocol specification. It is designed to facilitate simple and efficient communication between a master and one or more slaves, which suits the OzU SoC's single-core, DMA-less environment. The AXI4-Lite features a five-channel communication structure with a reduced set of signals

compared to the full AXI protocol, balancing low-bandwidth performance, reliability, and complexity constraints. The simplicity of AXI4-Lite ensures ease of implementation, minimal resource utilization, and compatibility with a wide range of IP cores, which is essential in customized SoC integration. The AXI4-Lite protocol, as part of the AMBA specification by ARM, is an open standard and free to use, making it an accessible choice for developers to integrate into customized SoC designs without licensing constraints. Given the lightweight dataflow requirements of the OzU SoC, the design inherits the Von Neumann architecture from PicoRV32, which implements a single 32-bit AXI4-Lite bus for all integrated components, controlled by the CPU as the communication master. A simplified block diagram of the OzU SoC is shown in Figure 2.5, while a more detailed diagram is amended as Appendix I.

The AXI4-Lite's five distinct channels are: Write Address (AW-channel), Write Data (W-channel), Write Response (B-channel), Read Address (AR-channel), and Read Data (R-channel), ensuring a reliable, handshake-based requests and transfers of data, addressing all possible faults and mismatches. For write transactions, i.e. store instructions, the master first places an address onto the AW channel, indicating its validity and transfer parameters, as well as the write data, its validity, and its parameters on the W channel. Once the write is complete, the slave responds on the B channel, acknowledging a proper write, and optionally responding with a code indicating the status of the write operation. Meanwhile, for read transactions, i.e. load instructions, the master similarly implements the AR and R channels; however, without the necessity of a response from the slave. Detailed description of each signal in the AXI4-Lite protocol is shown in Tables 2.2 to 2.6, while a timing diagram, obtained by simulating the OzU SoC, is shown in Appendix H. Further details about the protocol can be found in the ARM technical report in [51].

Table 2.2: AXI4-Lite Write Address (AW) Channel Signals.

Signal	Direction	Description
AWVALID	M to S	Indicates that the master has provided a valid address
AWREADY	S to M	Indicates that the slave is ready to accept the address
AWADDR	M to S	Specifies the address of the transaction
AWPROT	M to S	Protection Type: specifies the transaction attributes

Table 2.3: AXI4-Lite Write Data (W) Channel Signals.

Signal	Direction	Description
WVALID	M to S	Indicates that the master has provided valid data
WREADY	S to M	Indicates that the slave is ready to accept data
WDATA	M to S	Contains the data to be written to the slave
WSTRB	M to S	Write Strobes, indicates which byte lanes are valid

Table 2.4: AXI4-Lite Write Response (B) Channel Signals.

Signal	Direction	Description
BVALID	S to M	Indicates that the slave is providing a valid response
BREADY	M to S	Indicates that the master has accepted the response
BRESP	S to M	Contains the status of the write transaction

Table 2.5: AXI4-Lite Read Address (AR) Channel Signals.

Signal	Direction	Description
ARVALID	M to S	Indicates that the master has provided a valid address
ARREADY	S to M	Indicates that the slave is ready to accept the address
ARADDR	M to S	Specifies the address of the transaction
ARPROT	M to S	Protection Type: specifies the transaction attributes

Table 2.6: AXI4-Lite Read Data (R) Channel Signals.

Signal	Direction	Description
RVALID	S to M	Indicates that the slave has provided valid data
RREADY	M to S	Indicates that the master is ready to accept data
RDATA	S to M	Contains the data being returned from the slave
RRESP	S to M	Contains the status of the read transaction

2.3.3 Memory Organization and Dataflow

The OzU SoC encompasses three memories on the chip: a System OTP ROM (SOTP), a Serially-Programmable Read-Only Memory (SPROM), and a data SRAM. The starting address of the SOTP memory is the reset vector of the SoC, where the operation starts under normal operation conditions. The SOTP is a Verilog-hard-coded memory, implemented as simple read-only lookup tables, as shown in Figure 2.6. It is partitioned into two size-configurable sectors, the first having a self-test routine that ensures the SoC's core and different components are functional upon startup, designated the Self-test ROM (STR). The other sector, designated the Bootloader ROM (BLR), is designed to be used for any possible essential bootloader code, redundancy, or possible alternative structure. The STR also serves as the memory for simulating and verifying the SoC's RTL design. A simple bus decoder in the SOTP ensures proper switching between the STR and BLR identical interfaces according to the requested read address.

Figure 2.6: Section of the Verilog-Coded STR Memory in OzU SoC.

```
// ----- Block 0 ----- //
// ----- RESET and ISR ----- //

// Control Settings

// Disable All Interrupts

32'h0000_0032: str_instr <= 32'h010021b7; // 0050 lui gp, 0x1002 Load INTC Base Address
32'h0000_0033: str_instr <= 32'h0001a423; // 0051 sw x0, 0x8(gp) Mask All Interrupts
```

Storing the main operating firmware of the SoC, the SPROM serves as the main program memory. It is implemented as an n-size array of 32-bit readable/writeable cells in Verilog, compatible with the AXI4-Lite interconnect. The SPROM can be interfaced to one of the OzU SoC's existing serial peripherals, i.e., SPI, I2C, or UART, or to a dedicated SWD or JTAG interface for programming. Due to limitations of hardware resources, this typical method was not implemented; instead, the SPROM loads the program from a local computer file before synthesis, implementation, and bit stream generation. The program is stored as a .hex file, in simple hexadecimal RV32I/E machine code format.

Table 2.7: *Memory Organization of the OzU SoC.*

Memory	Use	Max Size	Programmable
STR SOTP	Self-test and startup routine	128 KB	Verilog-coded
BLR SOTP	Bootloader or initialization	128 KB	
SPROM	Main operational firmware	768 KB	Initialization or serially
SRAM	Processing data memory	256 KB	N/A

Similarly to SPROM, the data SRAM is a simple, single-bank size configurable memory implemented as R/W cells in Verilog. However, contrary to the SoC's ROMs, which are designed to store 32-bit instructions, the cells are implemented in size of 8 bits for each, to enable the proper functionality of byte-sized and halfword-sized load and store instructions of the RV32I ISA. The memory content is not initialized with any valid data upon bit stream generation. Due to their simplicity, all of the mentioned memories are implemented from the ground up, without using any obtained IPs. All memories, listed in Table 2.7, are designed to seamlessly interface with the AXI4-Lite interconnect. However, as mentioned, switching between the STR and BLR sectors is done using a simpler local decoder. Furthermore, to fulfill modern SoC design practices, a specific address range is reserved for future dedicated external memory interface integrations. The full planning of the address space of OzU SoC is amended as Appendix J.

It is evident that having multiple AXI4-Lite slaves, not limited to the mentioned memories, requires the design of dedicated switching mechanisms. Therefore, a configurable, generic and reusable 1-to-6 simple combinational AXI4-Lite switch has been designed, also from the ground up. The switch connects and disconnects the five channels to one of up to six connected slaves, in addition to asserting or deasserting an enable signal to each, according to the given address from the master. This switch is reused in a two-level hierarchy: the first level groups similar components of the SoC under one switch, for example, memories, and the second level bridges the connection to the particular addressed component, as seen in Figure 2.5 and Appendix I. Similar to memory, all other integrated peripherals discussed next are grouped and can be accessed through the two-level switching hierarchy of the SoC.

2.3.4 Integrating Peripherals and Auxiliary Units

As discussed in Chapter I, finding, designing, and integrating peripherals is a fairly challenging task that requires careful interfacing, standardization, and testing of integrated components. The development of OzU SoC's peripherals ecosystem is a combination of integrating obtained open-source IPs as they are (Category A in Table 2.8), integrating these other IPs through customized adapter interfaces (Category B in Table 2.8), and designing them from the ground up (Category C in Table 2.8). The same table also shows the grouping of these components, reflecting the usage of AXI switches, as previously discussed. The addressable SoC space is grouped into five groups: Memories, Basic I/O, Signal Generation, Serial Interfaces, and Control Units. The finally implemented peripherals set of the OzU SoC does not follow the exact original user specification, rather fulfills and extends them without compromising the lightweight size requirements. The only exception is the ADC peripheral, which was not implemented, as it cannot be implemented on FPGA chips without using additional external hardware.

Table 2.8: *Integrated OzU SoC Components Excluding Memories.*

AXI Group	Component	Description	IP Source
Basic I/O	GPIO0	Configurable 32-bit GPIO port	Category A
	GPIO1		
Signal Generation	TCP0	Timer/Counter/PWM module	Category C
	TCP1		
	TCP2		
	TCP3		
Serial Interfaces	UART0	Static UART XCVR	Category B
	UART1		
	UART2		
	UART3		
	I2C	Configurable I2C master XCVR	Category A
	SPI	SPI/DSPI/QSPI master XCVR	Category B
Control Units	INTC	Simple interrupt controller	Category C
	PFCN	Package pin function controller	

Although the established OzU SoC prototype managed to achieve most of the functionalities required by the end user, the prototype still skipped the implementation of some planned modules for different reasons. The main reason behind skipping these implementations is the availability of sufficient external hardware resources beyond the FPGA development board, complexity concerns, or the components serving non-essential features that do not affect normal operations. While Table 2.8 details the peripherals set implemented in the OzU SoC, Table 2.9 shows the rest of the components which were not implemented in the prototype for various reasons, but remain listed to complete a full realization of the OzU SoC. Following, a detailed description of each of these, implemented and unimplemented, components is discussed, highlighting their features, sources, operation, and integration mechanism to the SoC's core structure.

Table 2.9: *Unimplemented Components of the OzU SoC.*

Planned Component	Reason(s) for Not Implementing
6-Channel ADC	Absence of external hardware resources
CCM SRAM	Optional feature
External Memory Interfaces	Optional feature / Availability of HW
Multiple-Clock Controller	Reduction of complexity and timing issues
Reset Controller	Not necessary for FPGA prototyping
RVFI and JTAG Interfaces	Reduction of complexity and area
SWD Programming Interface	Volatility of FPGA storage / Available HW
Trap Handler	Optional feature / Can be handled by INTC
Watchdog Timer	Optional feature

2.3.4.1 GPIO Ports

The OzU SoC prototype has two identical 32-bit GPIO ports. This GPIO port unit is an MIT-licensed IP designed by Spencer Chang, obtained from GitHub [52]. This reusable design is a versatile and robust configurable GPIO port, which is fully compatible with AXI interfaces. Parameterization of this Verilog module allows determining the number of GPIO pins, the AXI bus address width, the base address of the peripheral, and the next pointer address for the next component. The design offers input, output, and direction lines for the number of pins assigned in the module parameters. Unifying these three groups of wires as physical bidirectional pins can be achieved by using supplementary external modules in the SoC. In the OzU SoC, the dedicated PFCN module maps these functions to physical pins as programmed in the firmware. This GPIO module has a comprehensive set of registers to access data and configure functionality. Table 2.10 lists these registers, their offset from the base address of the module, their read/write type, and their functions. As explained previously, these registers can be accessed using load and store instructions in the RV32I/E ISA.

Table 2.10: *GPIO Module Registers.*

Register	Offset (h)	Read/Write	Description
ID	0x00	Read only	Identifies the GPIO peripheral ID
HDL Revision	0x04	Read only	Identifies HDL revision of module
Next Pointer	0x08	Read only	Identifies next peripheral address
Software Reset	0x10	Write only	Performs a software reset of the port
GPIO Info	0x20	Read only	Stores the no. of GPIO pins available
DDR	0x24	Read/Write	Data direction configuration
DOUT	0x28	Read/Write	GPIO output data
DIN	0x2C	Read only	GPIO data in
REIN	0x30	Read/Write	Rising edge interrupt Enables
FEIN	0x34	Read/Write	Falling edge interrupt Enables
INTE	0x38	Read/Write	Master interrupt enables
INTS	0x3C	Read/Write	Interrupt status and clearing

2.3.4.2 Timer/Counter/PWM (TCP) Modules

The OzU SoC prototype implements four identical Timer/Counter modules, with configurable two-channel PWM output for each. Due to their simplicity, these modules are designed from the ground up. All four TCP modules are combined under a parent module that interfaces with the CPU using one AXI4-Lite interface. This is done to reduce the complexity and area of the design. Each single TCP module consists of a generic 32-bit Timer/Counter module with its corresponding registers, and two independent PWM signal generators that derive their timing from the parent module. This configuration provides the OzU SoC prototype with eight semi-independent PWM outputs, which is sufficient for most applications in its scope. Each TCP module has an independent set of control registers. All registers are 32-bit sized for simplicity and best utilization. Additionally, each TCP module has two independent Duty Cycle registers corresponding to each PWM generator. By default, TCP0 is configured as a system up-time timer, unless re-configured. The registers of any identical TCP module are listed in Table 2.11.

Table 2.11: *TCP Module Registers.*

Register	Offset (h)	Read/Write	Description
Status	0x00	Read only	Indicates operational status flags
Control	0x04	Read/Write	Controls the operation of module
Counter	0x08	Read only	Live access of current counter value
Offset	0x0C	Read/Write	Starting offset value upon start
Period	0x10	Read/Write	Finishing value to stop counter at
Division Scalar	0x14	Read/Write	Division prescalar for counting
PWM DC (A)	0x18	Read/Write	Duty cycle for PWM generator (A)
PWM DC (B)	0x1C	Read/Write	Duty cycle for PWM generator (B)

2.3.4.3 UART Modules

Similarly, the SoC implements four identical UART modules, grouped under one parent module that acts as a unified AXI4-Lite slave interface. The UART peripherals are based on simple hardware transceiver models, obtained from GitHub [53]. The IP of the original transceiver provides the simplest and minimal functionality of a UART Transceiver (XCVR), consisting of three files: the Transmitter (TX), Receiver (RX), and the parent transceiver module. These original designs are byte-based and static-configurable through Verilog parameterization; therefore, additional custom hardware is required to integrate them properly into the SoC's environment. A custom UART controller module is designed to wrap the original transceivers, having control registers, and dedicated Finite-State Machine (FSM) to handle automatic buffered transmission and reception of a complete word (4 bytes) per single transaction. This addition introduces little complexity, while allowing for a more autonomous exchange of data to and from each UART interface, requiring less execution of instructions, less AXI transactions, and less interrupt handling by the SoC's other modules, as compared to byte-based transactions. The CPU simply stores a 32-bit word of data into the controller's TX register and enables transmission in its control register, which in turn starts buffering it to low-level hardware

autonomously. Likewise, data reception is buffered into a 32-bit word register, which only interrupts the CPU once full if a read and clear operation is not requested earlier. The four transceivers, wrapped in four independent controllers, are all grouped under the parent module's single AXI slave interface as one peripheral. The registers of a single UART controller module are detailed in Table 2.12. It is worth mentioning that the Clock Frequency and Baudrate registers, as well as some control functions, are not functional in the most recent OzU SoC prototype, as they require extensive design modifications of the original TX and RX IPs.

Table 2.12: *UART Controller Registers.*

Register	Offset (h)	Read/Write	Description
Status	0x00	Read only	Indicates operational status flags
Control	0x04	Read/Write	Controls the operation of module
Clock Frequency	0x08	Read/Write	Stores the UART clock frequency
Baudrate	0x0C	Read/Write	Stores the baud rate of TX and RX
TX Word	0x10	Read/Write	Word to be buffered to TX hardware
RX Word	0x14	Read/Write	Buffered word from RX hardware

2.3.4.4 I2C Master Transceiver

Another thoroughly designed module of the OzU SoC is the I2C transceiver. Obtained from GitHub [54] and integrated as is, it is similar to the GPIO module in terms of offering robust configurability through parameterization and native compatibility as an AXI slave. The design includes optional, configurable n-level First In, First Out (FIFO) buffers for control, transmission, and reception. Verilog parameterization includes setting prescales for baud rates, as well as enabling and setting the size for command, write, and read FIFOs. The four registers of the I2C transceiver module are listed in Table 2.13.

Table 2.13: *I2C Transceiver Registers.*

Register	Offset (h)	Read/Write	Description
Status	0x00	Read/Write	Indicates operational status flags
Command	0x04	Read/Write	Controls the operation of module
Data	0x08	Read/Write	I2C read or write data
Prescalar	0x0C	Read/Write	Baud rate prescalar

2.3.4.5 SPI Transceiver

The original user requirements of the OzU SoC specify a single SPI transceiver to be implemented. Similarly to the UART, the transceiver is based on an IP hardware IP obtained [55]. The obtained transceiver provides a functionally tested and configurable module that can operate as an SPI, a DSPI, or a QSPI, which made it an appealing choice for customizability and scalability requirements. However, like the UART IPs obtained, the original design is byte-based and is not designed for AXI4-Lite interconnect. Consequently, an additional controller and AXI interface were designed to interface it with the SoC. Similarly, the SPI controller implements FSMs to buffer word transactions in both directions autonomously. The SPI controller module registers are listed in Table 2.14, which is similar to that of a UART controller.

Table 2.14: *SPI Controller Registers.*

Register	Offset (h)	Read/Write	Description
Status	0x00	Read only	Indicates operational status flags
Control	0x04	Read/Write	Controls the operation of module
TX Word	0x10	Read/Write	Word to be buffered to TX hardware
RX Word	0x14	Read/Write	Buffered word from RX hardware

2.3.4.6 Interrupt Controller (INTC)

A simple interrupt controller has been designed from the ground up to supplement the PicoRV32 limited single-interrupt input. It collects interrupts from up to 32 modules in the SoC in a 32-bit register, including most of the aforementioned integrated components. Before being reduced into a single-bit IRQ signal to the CPU, through a unary OR function, another configurable 32-bit register controls what components may trigger a final interrupt to the CPU, which is known as interrupt masking. The control communication with this module, from the CPU, is achieved through a parent control units module that is integrated as an AXI slave component. Table 2.15 lists the INTC registers.

Table 2.15: *Interrupt Controller Registers.*

Register	Offset (h)	Read/Write	Description
Master Interrupts	0x00	Read only	CPU-triggering interrupts
Non-Masked Interrupts	0x04	Read only	Non-masked interrupts
Interrupts Mask	0x08	Read/Write	Individual interrupts masking

2.3.4.7 Pin Functions Controller (PFCN)

Finally, a key module of the OzU SoC, designed from the ground up, is the Pin Functions Controller. This module selectively maps the native functional I/O of the SoC's different peripherals to the physical pins of the planned IC package, achieving a lower, standardized number of pins of the end-product package. This practice of pin function re-configuration is seen in many commercial deployment examples like the above-mentioned STM32 and SAM D21 MCUs. In most typical use cases of similar microcontroller SoCs, not all native peripheral I/Os are used at the same time, allowing this pin sharing mechanism. The OzU SoC can be realized as a quad, at least 64-pin, package, such as the Quad

Flat Package (QFP) package¹⁹. The PFCN module maps the SoC's components' native I/Os to eight 32-bit pin ports, designated from A to H. Each of these ports has a corresponding 32-bit control register that determines the functions of the physical pins. The PFCN has a default safe configuration through parameterization, until being reconfigured by the CPU. Like all other modules in the OzU SoC, the CPU can interact with this module through load and store instructions for setup, through the parent control units module, alongside the INTC. Further details about the particular mechanism of configuring the ports' pin functions and the module registers can be found in Appendix K.

2.3.5 Customizability and Scalability

The OzU SoC has been designed for customizability and scalability in the first place. From the beginning, PicoRV32 with its configurability features was selected to fulfill this requirement at the core of the SoC. The PicoRV32's parameterization, in Appendix G, does not only decide the base architecture of the CPU, but also affects the overall PPA aspects of the SoC. Extending that, memories of the OzU SoC have their own parameterization to assign their start addresses, end addresses, and most importantly their sizes. In fact, all OzU SoC modules, regardless of their level in the hierarchy, implement parameterization to assign their start and end addresses. Moreover, these parameters are priority-inherited from each following parent module in the SoC's hierarchy, allowing for a seamless, overall configuration of the SoC through the top module. The designed AXI4-Lite switch in particular is a good example of this flexibility. When reused, i.e. instantiated, it is not necessary to implement it with its full number of six slaves but for any other lower number of connected slaves, reducing complexity as much as possible.

¹⁹Quad packages, such as QFP and QFN (Quad Flat No-Lead), are widely used in modern MCUs packaging, due to their balance of performance, size, and ease of integration. These packages offer a high pin count for complex functionality while maintaining a compact form factor, making them ideal for space-constrained applications. Figure 1.3 in Chapter I is an example of a small-size QFP-8 package.

Combined modules, that is, TCP, UART and SPI blocks, implement automatic assignment of the addresses of their individual unit registers, using the inherited assigned base address to the parent AXI-compatible interface. These practices significantly reduce manual modifications of Verilog files, which could easily result in human-caused errors in synthesis and runtime. Additionally, the OzU SoC implements a master Verilog header file to customize the SoC's functions through directives and parameterization. This, supplemented with proper documentation, enables potential co-designers to reconfigure the SoC's functionality in a simplified way, without reviewing all of its source Verilog.

Finally, on the micro level of an end-child component module, local parameters are implemented to make the HDL understandable, reconfigurable, and debuggable, especially when it comes to representing and understanding the waveform simulation. This nature of customizability planned in the OzU SoC serves as a testbed for exploring further design automation opportunities using high-level programming languages, particularly in generating customized versions of the OzU SoC from a base Verilog prototype. This could enable the end user to generate versions of the SoC seamlessly, through a dedicated GUI interface, without going into the trouble of modifying Verilog files manually.

2.4 Implementation and Testing

As originally planned and due to the limited hardware resources, the OzU SoC is implemented targeting a front-end SoC design IP as the main development outcome. Therefore, its development scope is focused on its architectural planning and establishing an RTL blueprint design that includes coding, simulation, and verification. This front-end scope ends with basic FPGA prototyping and testing of the OzU SoC. Although most of the work orbits within the basic RTL design methodologies, other tools are involved, particularly in the coding and prototyping process.

2.4.1 Verilog Implementation

The entire OzU SoC is implemented as a Verilog HDL project in the Xilinx Vivado suite. Xilinx Vivado is a comprehensive design suite for digital system design and development, including SoC and FPGA-based designs. It offers tools for hardware modeling, simulation, synthesis, implementation, and prototyping of digital designs. In addition to being a comprehensive Integrated Development Environment (IDE) for writing HDL codes, it provides an optimized integrated environment for high performance designs with advanced features such as block-based IP reuse, system modeling, constraints analysis, and hardware management. Vivado has hardware and software tools to enable designers to work seamlessly with the outer environment, such as data exchange with computer files and direct programming of supported FPGA development boards.

In addition to the integrated open source modules in the OzU SoC, custom-designed modules were written maintaining consistency, systematism, readability, and modifiability. Depending on their nature, custom modules are designed as purely sequential logic modules, purely combinational logic modules, or modules adhering to the standard RTL Verilog implementation, consisting of both sequential and combinational logic blocks. Though the PicoRV32 CPU is implemented as a single-file implementation, the rest of the OzU SoC is well-modularized in separate-file implementations. Although this practice complicates the process of interfacing at the beginning of development, it ensures future flexibility, reusability, and scalability, as well as localizing debugging, isolating faults, and facilitating modifications. As mentioned above, the OzU SoC Verilog model invests heavily in the use of parameterization, Verilog directives, and headers to achieve the best possible customizability and reconfigurability. The ultimate goal of these practices is to implement future automation tools for seamless configuration and customizability.

2.4.2 Machine Code Generation

RISC-V machine code generation and programming the SoC, either through Verilog-embedded LUTs or through files, have proven to be an unexpectedly nontrivial task. The GNU Toolchain for RISC-V is the only known official method for generating RISC-V machine code. It is an open-source suite of software tools, including GNU Compiler Collection (GCC), Binutils assembler and linker, GNU Debugger (GDB), and libraries such as glibc or Newlib, tailored for the RISC-V architecture. The RISC-V's GNU toolchain, officially published on GitHub [56], operates primarily on Linux platforms and supports the complete process of translating high-level source code into executable RISC-V machine code. After installing the toolchain, developers write code in languages like C or C++ and use terminal line commands for compilation.

Although the GNU toolchain is mature, benefiting from years of development and support, its ease of use and effectiveness can vary, as it often requires familiarity with command line tools, Linux environment, and reconfiguration for non-standard setups. For automated generation of machine codes, developers must establish supplementary toolchains, like makefiles²⁰ or customized scripts to achieve one-click compilations. Known issues of the GNU toolchain include limited support for experimental or custom RISC-V extensions, and occasional bugs in less commonly used configurations. Some of these issues can be seen in the "Issues" section in [56]. Alternative methods of generating RISC-V machine code include proprietary tools, such as those of SiFive or LLVM-based toolchains²¹, which offer similar functionality but present different workflows or platforms, tailored specifically to the target product.

²⁰A Makefile in Linux is a script used to manage the compilation and build process of software projects. It defines rules and dependencies, specifying how source files should be compiled and linked, automating repetitive tasks, and ensuring efficient project builds.

²¹LLVM-based toolchains are modular development tools built on the Low-Level Virtual Machine (LLVM) framework, offering front-ends like Clang for high-level languages and backends for optimized code generation across architectures.

Considering the small code requirements to operate, test, and verify the functions of the OzU SoC prototype, alternative, simpler methods were used. Having to install the GNU toolchain on a virtual Linux machine for such a small code, given that Vivado is operated on Windows, introduces more complication than facilitation to the OzU SoC workflow. Therefore, for the OzU SoC, the source codes have been written in C and compiled into the RV32 assembly using an on-line compiler²². From that or directly, using another online assembler application²³, the assembly code can be assembled into a standard hexadecimal-represented RV32I machine code. The assembler application also offers the ability to add a linker script if necessary. The resulting hexadecimal machine code is then hard-coded in the SOTP Verilog itself with proper addressing or loaded to the SPROM from a .hex file before synthesis and implementation.

2.4.3 Simulation and Testing

Vivado provides a comprehensive GUI waveform viewer for simulating the designed model behaviorally, post-synthesis, or post-implementation, as well as the possibility to export the numerical results to console, files, or verify them. The OzU SoC prototype has been simulated, tested, and verified incrementally; starting from the CPU-memory integration, and all the way to completion, with each new integration being followed with the necessary coding, simulation, and verification. Eventually, these codes combined together in the SoC's OTP STR memory, forming a minimal demonstration and self-test routine of the SoC's different function. A persistent-configuration waveform view is organized, showing different operational signals of the SoC in a modular organization relevant to the SoC's hierarchy. The Verilog-coded self-test code in the STR starts by initializing default interrupt and pin function states, proceeding to testing the CPU and

²²The compiler used is Compiler Explorer that can be accessed online on www.godbolt.org

²³The assembler used is Online RISC-V Assembler, which can be accessed on riscvasm.lucasteske.dev

registers data integrity, and from that to memory and peripherals testing. The routine then jumps to the start address of the BLR sector, which in turn jumps to the SPROM, the latter storing a sample operational code loaded from a C-compiled .hex file. Figure 2.7 shows some key waveform views of this test, aligned within the same timing window: (a) shows enable signals of different switches and SoC components, indicating their operation at any given time, (b) shows the changing contents of the CPU registers, and (c) shows the end results on the eight physical pin ports.

2.5 Prototyping and Realization

The midpoint of a SoC's development process from concept to realization can be considered to be the FPGA prototyping phase, where the SoC design is simulated as a hardware structure under near-real-world conditions. Although an FPGA prototype does not represent the final physical semiconductor structure of a SoC, rather imitates it using a reconfigurable digital circuit, it still presents the SoC design to the real-life environment and ensures the design meets its PPA and timing constraints. The FPGA prototype allows for an evaluation of the proper functionality, area utilization, power consumption, and timing, as well as enabling interfacing with external real-life devices. After sufficient functional testing of the SoC prototype on FPGA, the development can move to the detailed backend aspects of the ASIC chip design.

2.5.1 *FPGA Prototyping*

The OzU SoC has been successfully prototyped on a Digilent Basys 3 FPGA development board. The Basys 3 development board is a small-to-mid-range development

platform, well suited for small projects made among beginners and educational environments²⁴. The Basys 3 is supported for plug-and-program in Vivado, ensuring a seamless synthesis, programming, and testing process without using any additional software tools. The Basys 3 is focused on simplicity and compactness, supporting designs ranging from basic digital circuits to small-sized projects. However, such a range of FPGA boards is often insufficient for complex applications requiring significant computational resources and connectivity, such as SoC designs.

The Basys 3 is built around the Xilinx Artix-7 FPGA series, specifically the XC7A35T-1CPG236C model. This mid-range, general-purpose FPGA belongs to the 28nm Artix-7 series, which is known to offer an optimal balance between performance, power efficiency, and cost. The XC7A35T provides 33,280 logic cells, 90 DSP slices, and 1,800 kb of block RAM, making it suitable for implementing digital logic, signal processing, and small-scale embedded systems. The Basys 3 board integrates this FPGA chip within a range of onboard essential components and external connectivity, using an on-board 100 MHz crystal oscillator as the primary clock source. It features 16 slide switches and 5 push buttons for user inputs, 16 LEDs and a 4-digit seven-segment display for output visualization, as well as external ports for extending functionality with external hardware. The board also has a USB-JTAG interface and a USB-UART bridge for programming and debugging. For more information, the reader may refer to [57].

For FPGA prototyping, a reduced version of the final, software-simulated OzU SoC prototype is customized, leveraging the design customizability, and considering the Artix-7's limited resources. Minimal modifications have been made, such as eliminating serial peripherals and other functions that would not be tested due to the absence of external hardware, and extending software delays to be human-noticeable. The reduction

²⁴More information about the Basys 3 can be found on www.digilent.com/reference/programmable-logic/basys-3/start

of functionality reserves more resources for the on-chip SPROM storing the main operational program. The SoC's program, after performing a self-test routine, operates the GPIO peripheral, blinking the LEDs on the Basys 3 using a parameterized software delay routine.

2.5.2 Limitations and Further Development

Given the limited resources available on the Artix-7 and Basys 3 board, as well as the absence of external hardware devices to interact with the SoC prototype²⁵. The FPGA prototype was barely enough to demonstrate the integrity of the SoC's core, its interconnects, their interaction with peripherals, as well as proper synthesis and timing. Although a solid proof-of-concept of the proper architecture and functionality of the OzU SoC, the FPGA prototype provides a limited test for validation.

Given that, further development should be done using more sophisticated FPGAs, such as the Kintex-7 series, to accommodate sufficient on-chip resources. Additionally, it is highly recommended to further implement external memory chips to store larger operational programs, interfaced using the SoC's SPI or I2C peripherals, or dedicated interfaces, as well as establishing a dedicated hardware testing framework for all peripherals in the OzU SoC. Such a framework requires customizing other development boards, or MCU development boards, to imitate outer-world signals and communications. Finally, it is also reasonable to extend the OzU SoC with a JTAG interface or enable the RVFI interface to accurately monitor and debug CPU operations while operating on an FPGA. The end scope of a complete FPGA prototype should be implementing, testing, and validating the complete, hardware, and software, version of the OzU SoC completely within its target environment. The development can then proceed safely to the ASIC phase.

²⁵External hardware could be an external flash memory as program ROM, debuggers, environment-imitating and communication MCUs, as well as ADC hardware with real-life signals.

2.6 Possible Areas of Automation

Developing and testing the OzU SoC until the FPGA prototype phase, it became evident that automation could play a crucial role in streamlining and optimizing its design, reconfigurability, and operation. Some areas of the development workflow where automation can be leveraged are identified for further development. High-level programming languages, such as Python or Java, are common means to implement such automation.

2.6.1 Automated RTL Customization

Although the OzU SoC RTL model is well parameterized and implements directives for reconfiguration and extension, using these features manually is still a detailed, time-consuming, and error-prone process, due to the many details involved. It is hypothesized that this base prototype design of the OzU SoC, after extensive revision and completion, can be used with a dedicated GUI-based program to generate autonomously customized versions. Using the extensive parameterization implemented in the base RTL model, such a hypothesized program can regenerate custom versions tailored to specific requirements and specifications. In addition, this tool should provide automated simulation, testing, and verification of these generated versions.

2.6.2 Automated Code and Memory Generation

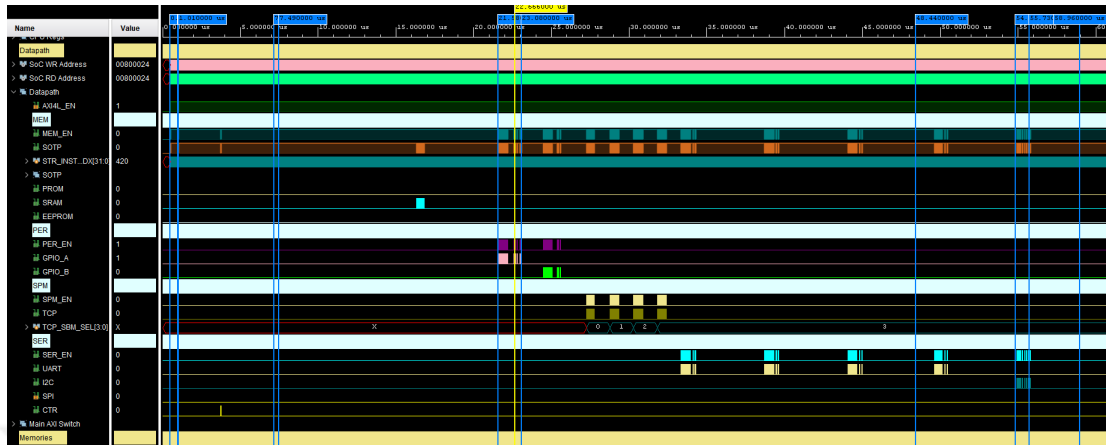
As coding turned out to be a nontrivial task as explained, dedicated scripts should be developed to compile source, operational codes of the SoC into OzU SoC-specific machine code. This implies considering the specific architecture of the OzU SoC at the compiling stage, providing peripheral drivers if necessary, and formatting the machine code properly to be placed in either of the available program memories. Such tools are typical in commercial MCU products, including products based on the RISC-V architecture: vendors often provide IDEs, SDKs, and other software tailored to their particular

products. Even for OzU SoC as a testing prototype, the most trivial application in automated machine code generation is the autonomous conversion of the simple hexadecimal RISC-V machine code into dedicated Verilog-hardwired LUT structures, forming SOTP memory sectors as previously shown in Figure 2.6. This proposed automated generation of STOP modules eliminates the need for manual readdressing and commenting of their Verilog codes, significantly reducing the development and testing efforts, and eliminating human-caused errors.

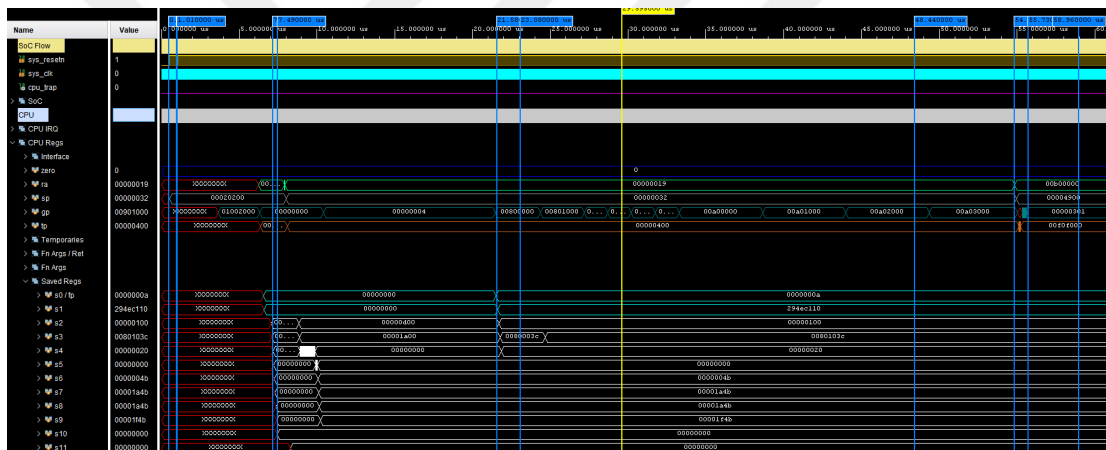
2.6.3 Automated Testing and Verification

As the simulation results and waveforms in Vivado can be exported to data files, it is best for larger programs that require extensive verifications to be done through dedicated automated verification tools. This implies writing high-level scripts to emulate the correct behavior of the SoC, comparing the provided results from simulation to its predicted ideal behavior, and pinpointing any arising result mismatches seamlessly in long programs and large simulation. Establishing such tools is typical, especially for larger SoC designs that require extensive testing of their functionalities under several testing scenarios. The same automation concept can be applied to hardware in the FPGA prototyping phase, where external MCUs or FPGAs can be programmed to interface, interact, and verify the SoC autonomously, providing a more advanced level of verification under near-real-world conditions and operational scenarios. Furthermore, AI could be used to write large batches of testing scenarios and verify them automatically. As previously explained, the use of automation does not stop at the prototyping phase but becomes even more sophisticated as the SoC development advances into the realization phases.

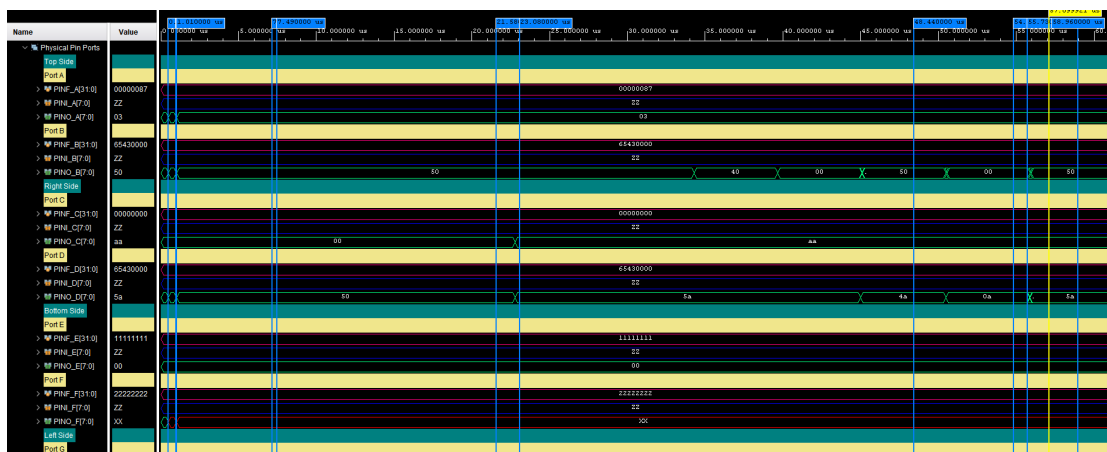
Figure 2.7: *Waveform Simulation Views of the Final OzU SoC Prototype.*



(a) Components Enable Signals.



(b) CPU Registers Content.



(c) Physical Pin Ports.

3. ISA OPTIMIZATION

This chapter presents the NOVLI-ISA, a proposed compatible extension to the standard RV32I/E ISA, with a focus on reducing machine code size as the primary goal, without compromising functionality or representation efficiency. The chapter starts with discussing the motivation behind reducing machine code size, reviewing relevant works in the literature. Then, the design philosophy of the NOVLI-ISA is explained, following with the technical details of its implementation, results, and feasibility evaluation.

3.1 The Necessity for Code Optimization

In general, optimizing the efficiency and storage requirements of both codes and user data have become essential in modern computer engineering. Previously discussed reasons like the global silicon shortage that followed the COVID-19 pandemic, coupled with the rapidly increasing global demand for electronic products, have focused more research efforts towards the efficiency of silicon usage and efficient power consumption. Code and data optimization and compression allow for better utilization of computing and energy resources, achieving faster execution times, and reducing the system's physical size. These optimizations reflect mainly on both on-chip and external memories, which typically occupy the largest area of a SoC chip or environment, as detailed in Chapters I and II, translating to more or less silicon usage. The OzU SoC prototype proved to be no exception, where the limitations in available FPGA resources have forced a significant scale-down of its main operational program memory, the SPROM. This in turn limited the testing of the prototype to running small sample programs in the absence of external memories and relying solely on on-chip memory, necessitating costly hardware upgrades to enable more comprehensive testing. These limitations would be indeed inherited in any further realization phase, such as affecting the PPAC aspects of an ASIC realization.

3.2 Machine Code Optimization in the Literature

For their discussed necessity, several technical methods have been developed for reducing the size of both machine codes and user data. Nevertheless, the technical scope of this thesis is only concerned with machine code. Machine codes can be either optimized through compression techniques, or optimizing ISA definitions and extensions.

One of the common and simple approaches to compress codes is to use dictionary-based compression techniques as in the work presented in [58]. In dictionary-based techniques, frequently occurring instruction sequences are replaced with shorter codes in program memory. The CPU executes these preprogrammed sequences in runtime, using dedicated LUTs or other specialized hardware. Another practice of code compression involves the application of entropy coding, which is more used in data compression. A widely used application of entropy coding is Huffman coding or arithmetic coding [59], which is based on the conversion of instructions into shorter codes based on their statistical frequency of occurrence. This results in the most commonly occurring instructions being represented in shorter codes, compared to those with less occurrence. This method requires a detailed analysis of typical operational programs to determine the frequency of occurrence, which could be either an application-specific or a general case analysis.

Compressed machine codes can be generated by either encoding standard codes into compressed codes, directly compiling high-level source codes into an optimized assembly, or a combination of both. When compressing a standard assembly code to its compressed equivalent, the operation can be performed either statically or dynamically. Static compression as in [60], is done at compile time, producing a compressed code that is decompressed at runtime. Dynamic compression as in [61], occurs during program execution, which allows for better adaptation to runtime conditions. The decompression of compressed codes is performed by the internal modules of the CPU or by an additional adapted decompressor hardware, as implemented in [62].

The merits and need for machine code compression and optimized ISAs have been previously explained. However, code optimization and compression also present several challenges; the process of compressing and decompressing instructions introduces additional complexities in both hardware and software toolchain. In a design that implements code compression, simplicity, compatibility, and flexibility are essential to ensure that the compression and decompression process does not introduce significant performance overhead and hardware complexities.

There have been several proposals to improve the RISC-V machine code density using different methods, and from different perspectives of optimization, including the ratified RV32C extension. The comprehensive report from UC Berkeley [63] outlines a detailed approach to improve code density in RISC-V architectures, including specific optimizations for load/store, register operations, and more. The report focuses mainly on the fixed-length 16-bit encoding method, as in the RV32C extension. The approach followed in the RV32C extension is to find a reduced fixed-size 16-bit ISA, which is explained in detail in the following context.

In the work presented in [64], instruction fusion is proposed for the RV64G ISA and its standard extensions. Instruction fusion is a method that combines smaller operations into single complex operations, reducing latency and resource utilization in exchange for more hardware complexity. From another perspective of optimization, the practical work in [65] proposes an application-specific ISA extension. This work aims to improve the processing of the Fast Fourier Transform (FFT) executed in the RV32I architecture. A set of both hardware and software optimizations are proposed in [66] to improve the performance and code density of IoT applications using the RV32-based CV32E40P core. The work proposes a new ISA with focus on pop/push operations to reduce the code size, in addition to the other hardware and software toolchain optimizations.

The work in [67] is similar to the proposed NOVLI-ISA, where a specialized RISC architecture, CCRP, is designed to execute compressed codes directly. An instruction-level, variable-length compression method is implemented, with a real-time decompression engine integrated into the processor, allowing for a code size reduction of 30-50%. Finally, the work presented in [68] introduces a tool that employs Dead Function Elimination and Common Instructions Outliner techniques to optimize assembly code, achieving a code size reduction of up to 6% in the RISC-V ISA. These optimizations are applied at the assembly level, mostly independently of the software toolchain.

3.2.1 *The Ratified RV32C Extension*

The RV32C extension is composed of 27 fixed-length 16-bit instructions organized in 8 formats as shown in Figure 3.1 [69], also amended as Appendix B for clarity. The C extension aims to decrease memory usage and improve code density, particularly for embedded systems with limited memory resources. The C extension reduces the size of machine codes by providing a shorter, half-length representation of some standard and pseudo instructions in the base RV32I definition, trading off some of their full functionality for shorter size in some formats. For example, the register-immediate instructions in the standard RV32I ISA define three instruction operands: a source register, a destination register, and the immediate literal. However, in the C extension, the source and destination registers must necessarily be the same register, and the immediate bit size is reduced. In addition of that functional reduction, other instructions in the extension are designed around exploiting frequent operations, such as operations affecting the Stack Pointer register (see Appendix E for RV32 register file details). The extension also introduces some pseudo instructions natively, such as the *c.nop* (Compressed No Operation), equivalent to any write to the Zero register if a standard RV32I instruction is used. It also introduces a new instruction, *c.mv*, to "move", i.e. copy, the content of a register to another.

Figure 3.1: ISA Card of the RV32C Extension.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0					
funct4				rd/rs1				rs2				op				CR-type				
funct3			imm	rd/rs1				imm				op				CI-type				
funct3			imm					rs2				op				CSS-type				
funct3			imm					rd'				op				CIW-type				
funct3			imm	rs1'			imm	rd'				op				CL-type				
funct3			imm	rd'/rs1'			imm	rs2'				op				CS-type				
funct3			imm	rs1'			imm				op				CB-type					
funct3			offset										op				CJ-type			

Inst	Name	FMT	OP	Funct	Description
c.lwsp	Load Word from SP	CI	10	010	lw rd, (4*imm)(sp)
c.swsp	Store Word to SP	CSS	10	110	sw rs2, (4*imm)(sp)
c.lw	Load Word	CL	00	010	lw rd', (4*imm)(rs1')
c.sw	Store Word	CS	00	110	sw rs1', (4*imm)(rs2')
c.j	Jump	CJ	01	101	jal x0, 2*offset
c.jal	Jump And Link	CJ	01	001	jal ra, 2*offset
c.jr	Jump Reg	CR	10	1000	jalr x0, rs1, 0
c.jalr	Jump And Link Reg	CR	10	1001	jalr ra, rs1, 0
c.beqz	Branch == 0	CB	01	110	beq rs', x0, 2*imm
c.bnez	Branch != 0	CB	01	111	bne rs', x0, 2*imm
c.li	Load Immediate	CI	01	010	addi rd, x0, imm
c.lui	Load Upper Imm	CI	01	011	lui rd, imm
c.addi	ADD Immediate	CI	01	000	addi rd, rd, imm
c.addi16sp	ADD Imm * 16 to SP	CI	01	011	addi sp, sp, 16*imm
c.addi4spn	ADD Imm * 4 + SP	CIW	00	000	addi rd', sp, 4*imm
c.slli	Shift Left Logical Imm	CI	10	000	slli rd, rd, imm
c.srli	Shift Right Logical Imm	CB	01	100x00	srli rd', rd', imm
c.srai	Shift Right Arith Imm	CB	01	100x01	srai rd', rd', imm
c.andi	AND Imm	CB	01	100x10	andi rd', rd', imm
c.mv	MoVe	CR	10	1000	add rd, x0, rs2
c.add	ADD	CR	10	1001	add rd, rd, rs2
c.and	AND	CS	01	10001111	and rd', rd', rs2'
c.or	OR	CS	01	10001110	or rd', rd', rs2'
c.xor	XOR	CS	01	10001101	xor rd', rd', rs2'
c.sub	SUB	CS	01	10001100	sub rd', rd', rs2'
c.nop	No OPeration	CI	01	000	addi x0, x0, 0
c.ebreak	Environment BREAK	CR	10	1001	ebreak

The design of the RV32C extension enables compatible integration on top of the standard RV32I ISA, allowing a mix of RV32C and RV32I instructions within the same program. The RV32I set instructions are identified by their first two least significant bits (LSB) always being '11'. On the other hand, RV32C instructions start with '00', '01' or '10' as the first two LSBs. Although the extension is an efficient and simple method to compress standard machine codes, increasing their density, it should be obviously noted that the extension is not all-inclusive of all possible operations in the standard RV32I ISA. The C extension has shown to typically save 25 to 30% of the storage space requirement for a given program [70].

3.3 Design Philosophy of the NOVLI-ISA

The NOVLI-ISA is designed to be inclusive, efficient, and fully backward compatible as an extension to the standard RV32I ISA. However, it cannot be used alongside the C extension, rather as a comprehensive replacement of it, as it shares the first two LSBs with the RV32C instructions. In discussion of its results at the end of this chapter, the proposed ISA has proven to offer size reduction metrics similar to those of the C extension, offering far more inclusive functionality, making it a candidate future replacement to the ratified C extension. The following are the design guidelines and considerations that shaped the design and implementation of the NOVLI-ISA.

3.3.1 *Compatibility with Standard Memories*

The proposed ISA introduces variable-length instructions in discrete sizes of N multiples of bytes, ensuring compatibility with standard memory organizations whether they store and deliver data as bytes or words. This practice of discretized length is established to avoid any infeasible complications in hardware and program flow control. The NOVLI-ISA encompasses fixed and variable-length 8-bit, 16-bit, 24-bit, and 32-bit instructions. To ensure optimal utilization of the instruction size, some subsets can take either one of two formats: short or long, depending on the size of given operands.

3.3.2 *Optimization for Embedded Applications*

Since area constraints matter the most in embedded applications, the NOVLI-ISA leverages the RV32E architecture to introduce embedded-specific instructions. In an RV32E environment, 4 bits are sufficient to address any of the 16 defined registers, instead of 5 bits for the RV32I. This single vacant bit, though seems insignificant, has proven to be useful in establishing a subset of 8-bit, micro instructions. These compact instructions are mostly useful in embedded applications.

3.3.3 *Maintaining the Simplicity of RISC-V*

The proposed ISA inherits the simplicity and straightforwardness of the RV32 ISA definition. It implements a leading fixed-size 4-bit opcode for all instructions, which simplifies the decoding hardware. The opcode is supplemented by other function bits to determine the exact function of the instruction, similar to *funct3* and *funct7* in the original RV32I ISA. Moreover, a single, fixed-position bit is used to determine the instruction length, short or long, for variable-length subsets. The function and register addressing bits are positioned consistently in the same-size instruction formats. This practice, as in the opcode, simplifies the complexity of the hardware when synthesizing the decoding hardware. As in the RV32C, all NOVLI-ISA opcodes start with either '00', '01' or '10' as the first two LSBs, permitting mixing with standard RV32I instructions.

3.3.4 *Inclusiveness, Enhanced Functionality, and Efficiency*

NOVLI-ISA emphasizes an enhanced all-inclusive scope of functionality, motivated by the original RV32I ISA definition and its RV32C extension, which lacks some key functionalities. Moreover, the ISA extends the RV32I functionality with instructions for performing typical, frequently-occurring operations, especially in embedded environments, e.g., register increments and decrements. The NOVLI-ISA introduces new functionalities: a hardware-based No Operation (NOP) or short wait instruction, instructions for unary register operations, embedded-specific instructions, and other arithmetic operations derived from the pseudo instructions of the RV32I ISA. Designed for efficient utilization of instruction bits and eventually memory usage, the NOVLI-ISA increases the information density of instructions when size reduction is not possible, compared to the standard RV32I ISA. For example, the NOVLI-ISA can load a register with a 21-bit immediate literal in a single 32-bit instruction, compared to 12 bits at once in the standard ISA's I-type instructions. This enables a single-instruction loading of an RV32's 32-bit

register with unsigned values of up to 2,097,152, compared to only 4096 using standard RV32I instructions. The NOVLI-ISA instructions are generally highly versatile, offering more sub functionalities compared to original RV32I definition, and certainly, compared to the reduced RV32C extension. Although this extensive functionality might sound like too many hardware complications at first, the NOVLI-ISA is nevertheless carefully designed to maintain the simplicity of the RISC-V, as well as being seamlessly flexible to be reduced or extended. This flexibility enables tailoring supporting CPU designs for specific applications, as well as adhering to the designated PPA constraints seamlessly.

3.3.5 *Minimal Altering of Existing RV32I/E Implementations*

The NOVLI-ISA is designed to introduce modular, as possibly minimal modifications to any existing or future RV32I/E architecture implementation. The discretization of instruction lengths in N-multiples of bytes, simple opcode organization, and consistent bits positioning, are all practices that translate into hardware simplicity, modularity, and compatibility. In the following context, the NOVLI-ISA demonstrates the systematization and efficiency of its consequent, CPU-containable, hardware modifications, without affecting any outer SoC structures, i.e., memory and interconnects.

3.4 NOVLI-ISA Formats and Functions

The NOVLI-ISA proposes the subsets of instructions shown in Table 3.1 and detailed afterwards. The definition of the NOVLI-ISA is versatile and flexible, where individual functionalities within each subset can be eliminated to satisfy tight area requirements, or extending in the reserved space for application-specific tailoring. In the following, the NOVLI-ISA subsets with their functions and formats are generally explained, while the fully-detailed blueprint of the NOVLI-ISA is amended as Appendix L.

Table 3.1: *Instruction Subsets of the NOVLI-ISA.*

Instructions Subset	Instruction Length
Micro / RV32E-Specific Instructions	Fixed 8-bit
Unary Register Operations	Fixed 16-bit
Register-Register Operations	Fixed 24-bit
Register-Immediate Operations	Variable 24/32-bit
Load Immediate / Upper Immediate	Variable 16/32-bit
Memory Operations	Variable 24/32-bit
Branching Operations	Variable 24/32-bit
Jump / Jump and Link	Variable 16/32-bit
Jump / Jump and Link (Register)	Variable 24/32-bit

3.4.1 8-bit Micro Instructions

This is the smallest set of the NOVLI-ISA consisting of four 8-bit instructions: an instruction that can be used in both RV32I and RV32E environments, and three instructions that are specifically designed for the RV32E architecture, the latter being operational on registers x0 to x15 only. These three RV32E-specific instructions can still be implemented in an RV32I architecture, but only affecting the RV32E register range. The four instructions are as follows: No Operation / Short Wait (NOP/SWAIT), Register Reset to Zero (RGRST), Register Increment by One (RGINC), and Register Decrement by One (RGDEC). The subset format shown in Table 3.2 consists of two bits for the subset opcode, two bits for determining the instruction (function), and four bits for the instruction operand, that is, the wait parameter or the target RV32E register address.

Table 3.2: *8-bit Micro Instructions Format.*

Bits	7:4	3:2	1:0
Function	Instruction parameter	Instruction function	Subset opcode

3.4.2 Unary Register Operations Instructions

This is a fixed-size 16-bit subset that is not defined in the original RV32I ISA. It is introduced to perform arithmetic and logical operations on a single (target) register, *rt*, with its format shown in Table 3.3. These instructions are applicable where the two source registers *rs1* and *rs2* are the same as the destination register *rd*, in the RV32I ISA. When these three are the same, the operation is unary. In the NOVL-ISA, a basic set of unary register operations is defined, including resets to zero or common nonzero values, increments and decrements by a value from 1 to 4, complements (1's or 2's complement), as well as left and right shifts by any unsigned 4-bit value. Using these instructions where applicable reduces the original instruction size requirement to half.

Table 3.3: *Unary Register Operations Instructions Format.*

Bits	15:12	11:10	9:5	4	3:0
Function	I. function	Sub function [1:0]	<i>rt</i> address	Sub funct. [2]	S. opcode

3.4.3 Register-Register Operations Instructions

This subset offers the exact functionality of the standard RV32I R-type instructions, but comes in a reduced fixed size of 24 bits, as shown in Table 3.4. Unlike the C extension, these instructions offer full addressing of *rs1*, *rs2*, and *rd*. This optimization is always applicable, eliminating 8 bits or 25% of the standard instruction size.

Table 3.4: *Register-Register Operations Instructions Format.*

Bits	23:20	19:15	14:10	9:5	4	3:0
Function	I. function	<i>rs2</i> address	<i>rs1</i> address	<i>rd</i> address	N/A	S. opcode

3.4.4 Register-Immediate Operations Instructions

This subset offers the same functionality as the equivalent standard subset with an added function (see Appendix L). The subset has a variable size, 24 bit or 32 bit long, format as detailed in Table 3.5. It optimizes code density by utilizing 14 bits for the immediate literal *imm* in its long form, instead of 12 bits in the standard format. In its short form for passing a small immediate, the immediate literal is 6-bit long.

Table 3.5: *Register-Immediate Operations Instructions Format.*

24-bit (Short) Format						
Bits	23:21	20:15	14:10	9:5	4	3:0
Function	I. function	<i>imm</i> (short)	<i>rs1</i> address	<i>rd</i> address	24/32-bit	S. opcode
32-bit (Long) Format						
Bits	31:28	27:15	14:10	9:5	4	3:0
Function	I. function	<i>imm</i> (long)	<i>rs1</i> address	<i>rd</i> address	24/32-bit	S. opcode

3.4.5 Load Immediate / Upper Immediate Instruction

This single variable-length, 16-bit or 32-bit long instruction serves register loading of both immediates and upper immediates (see *lui* in Appendix A). In its long form, the passed (lower or upper) immediate size is 21 bits, improving the representation density, compared to its standard equivalents which can only pass 12-bit-long immediates or 20-bit-long upper immediates. In its short form, the instruction reduces the standard instruction sizes to half, utilizing five bits to pass smaller immediate values. Both formats are detailed in Table 3.6.

3.4.6 Memory Operations Instructions

This subset encompasses all possible load and store operations of the standard ISA in one opcode and format. The subset has a variable-length, 24-bit or 32-bit long, formats,

Table 3.6: *Load Immediate / Upper Immediate Instruction Format.*

16-bit (Short) Format					
Bits	15	14:10	9:5	4	3:0
Function	I. function (<i>li/lui</i>)	<i>imm</i> (short)	<i>rd</i> address	16/32-bit	S. opcode
32-bit (Long) Format					
Bits	31	30:10	9:5	4	3:0
Function	I. function (<i>li/lui</i>)	<i>imm</i> (long)	<i>rd</i> address	16/32-bit	S. opcode

depending on the required addressing offset size. In Table 3.7, *rba* refers to the "Base Address Register" that holds the base address of the memory transaction, determining the final address when the offset is added. In loads, the content of the addressable memory location is loaded to *rd*, while in stores, the content of *rs2* is stored in the target memory location, as defined in the standard RV32I. The format can either pass a small address offset of 6 bits or a larger one of 14 bits, compared to the standard ISA that can pass 12 bits for loads and 13 bits for stores. Optionally, the format can be modified to provide additional 3 offset bits, in exchange of limiting the registers that can be used as base address register to a predefined range of special-purpose registers. This subset exploits the frequent use of small offsets in accessing peripheral registers in the load-store, RISC-V applications: OzU SoC as an example. This approach proved to be the most frequently occurring optimization in the NOVLI-ISA evaluation, detailed in the following.

Table 3.7: *Memory Operations Instructions Format.*

24-bit (Short) Format						
Bits	23:21	20:15	14:10	9:5	4	3:0
Function	I. function	Offset [5:0]	<i>rba</i> addr.	<i>rs2/rd</i> addr.	24/32-bit	S. opcode
32-bit (Long) Format						
Bits	31:29	28:15	14:10	9:5	4	3:0
Function	I. function	Offset [13:0]	<i>rba</i> addr.	<i>rs2/rd</i> addr.	24/32-bit	S. opcode

3.4.7 Branching Operations Instructions

This subset maintains the original branching functionality of the original RV32I definition (B-type in Appendix A). It is similar to memory operations in exploiting the variance in addressing offset sizes, formats shown in Table 3.8.

Table 3.8: *Branching Operations Instructions Format.*

24-bit (Short) Format						
Bits	23:21	20:15	14:10	9:5	4	3:0
Function	I. function	Offset [5:0]	<i>rs2</i> address	<i>rs1</i> addr.	24/32-bit	S. opcode
32-bit (Long) Format						
Bits	31:29	28:15	14:10	9:5	4	3:0
Function	I. function	Offset [13:0]	<i>rs2</i> address	<i>rs1</i> addr.	24/32-bit	S. opcode

3.4.8 Jump / Jump and Link Instruction

This single instruction format performs unconditional jumps and jumps linking the return address to a register *rl*, i.e. storing the return address. Using an instruction-passed literal as an added offset to the current Program Counter (PC), forming a jump address, this format combines the functionalities of native *jal* and pseudo *j* instructions in the standard RV32I definition. This format is highly dynamic, not only providing a variable-length format of 16 or 32 bits, but also utilizing the 5 bits for addressing the link register as additional offset bits when linking is not required, i.e. as a *j* instruction. In comparison, the equivalent pseudo *j* instruction in RV32I is just a representation-inefficient 32-bit *jal* instruction that links to the hardwired zero register, x0. The short format achieves a 10-bit offset without linking or a 5-bit offset with linking, where the long format achieves a 16-bit offset without linking or 11-bit offset with linking, as shown in Table 3.9.

Table 3.9: *Jump / Jump and Link Instructions Format.*

16-bit (Short) Format					
Bits	15	14:10	9:5	4	3:0
Function	I. function (<i>j/jl</i>)	Offset [4:0]	<i>rl</i> or offset [9:5]	16/32-bit	S. opcode
32-bit (Long) Format					
Bits	31	30:10	9:5	4	3:0
Function	I. function (<i>j/jl</i>)	Offset [10:0]	<i>rl</i> or offset [15:11]	16/32-bit	S. opcode

3.4.9 Jump / Jump and Link (Register) Instruction

Similarly, this format performs jumping and jumping and linking, but using a value stored in a register, added to the instruction-passed offset. Equivalently, this format combines the native *jalr* and pseudo *jr* instructions in the standard definition. The format implements the same dynamic mechanism of utilizing link register address bits as additional offset bits when linking is not required, i.e., *jr* instruction. The only difference is that this format can come in sizes of 24 bits or 32 bits, as shown in Table 3.10. The short format achieves a 13-bit offset without linking or 8-bit offset with linking, while the long format achieves a 21-bit offset without linking or 16-bit offset with linking.

Table 3.10: *Jump / Jump and Link (Register) Instructions Format.*

24-bit (Short) Format						
Bits	23	22:15	14:10	9:5	4	3:0
Function	Func.	Offset [7:0]	<i>rba</i> addr.	<i>rl</i> or offset [12:8]	24/32-bit	Opcode
32-bit (Long) Format						
Bits	31	30:15	14:10	9:5	4	3:0
Function	Func.	Offset [15:0]	<i>rba</i> addr.	<i>rl</i> or offset [20:16]	24/32-bit	Opcode

3.4.10 Instructions Left Without Optimization

From the standard RV32I definition, the instructions *auipc* (add upper immediate to PC), *ecall*, and *ebreak* (environment call and break) do not have equivalents in the NOVLI-ISA and must be used as is. This is because these instructions are fairly less frequently used instructions. Therefore, their impact on storage size can be neglected. These instructions can be used as they are, as mixing RV32I and NOVLI-ISA instructions was considered in the first place.

3.5 Implementation of the NOVLI-ISA

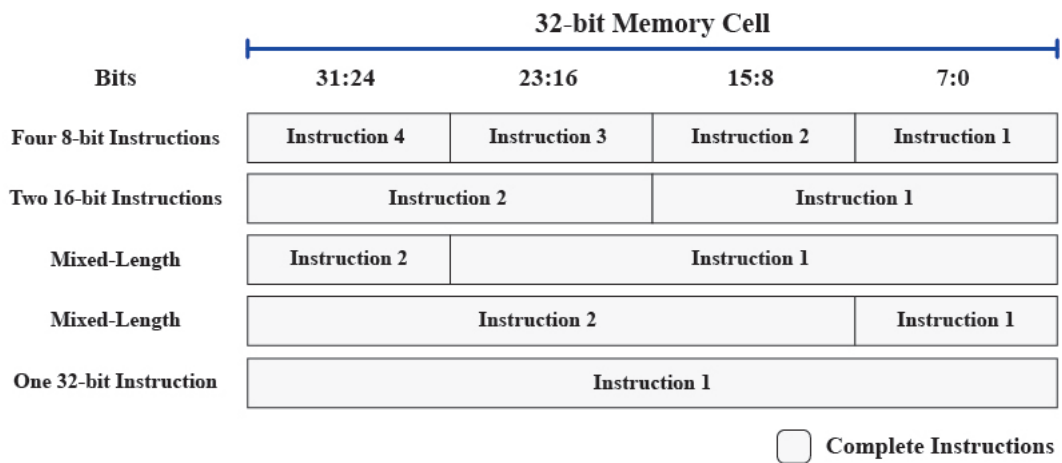
The NOVLI-ISA is designed considering a minimal amount of modifications and complications in both hardware and software requirements, while maintaining the simple, modular, and flexible nature of the RISC-V architecture. In the following, hardware and software requirements to realize possible CPU implementations supporting the NOVLI-ISA are highlighted, followed by a discussion of the implementation made in this thesis.

3.5.1 Hardware and Software Requirements

Designed with compatibility in mind, the NOVLI-ISA leaves the program memory and its interface intact without any changes; a supporting CPU can be interfaced to any standard program memory whether it stores and delivers code in bytes or words. The drastic changes in a CPU architecture supporting the NOVLI-ISA would be mostly made to the Fetch Unit (FU), which ensures the proper fetching of the variable-length instructions from memory, whether the transactions occur as bytes or words. Other significant modifications are expected in the Decode Unit (DU), which would act as the decompressor (or converter) of the NOVLI-ISA instructions to the standard RV32I format. The DU would convert the NOVLI-ISA variable-length instructions to standard 32-bit instructions before passing them to the following Execution Unit (EU). The EU would have no 'knowledge'

of the origin of the passed instructions, standard or optimized. The fetching and parsing of variable-length instructions requires the introduction of additional FSM parameters into the FU. Depending on the memory interface, one approach is to fetch program in words, deconstructing and reconstructing these words as valid instructions, and another is to fetch the program as bytes and combine them to construct valid instructions. Thus, the guideline on designing the ISA lengths in N-multiples of bytes. In the sample CPU implementation made to test the hypothesized NOVLI-ISA hardware requirements, designated RV32IN and detailed in 3.5.2, the machine code is fetched in words, which presents a finite set of three cases for extracting valid instructions:

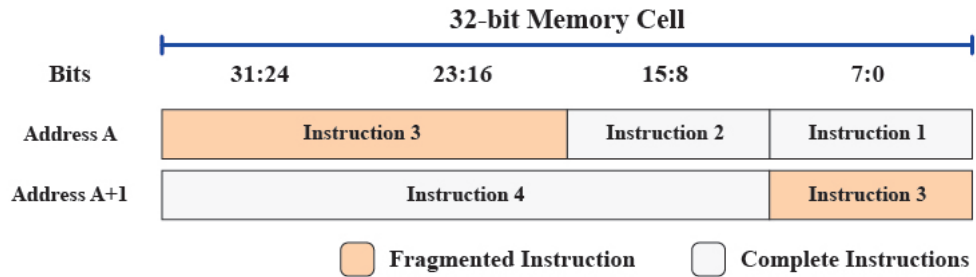
Figure 3.2: *Memory-Aligned Instructions Cases.*



1. This case represents the normal fetching process of a standard RV32I/E CPU, only mentioned for inclusion and reference. In this case, the fetched word is a single, complete, standard or optimized 32-bit instruction, as illustrated in the last row of Figure 3.2. The fetching and execution of the instruction in this case occurs normally as in a standard RV32I/E CPU, without inducing further FU states. Standard or optimized, the instruction is then decoded in the DU.

2. The fetched word has a number of complete shorter instructions, i.e., four 8-bit instructions, two 16-bit instructions, or a 24-bit instruction with an 8-bit instruction. All of these possibilities are illustrated in Figure 3.2, excluding the last example (corresponding to Case 1). If any of these possibilities occur, the FU's FSM holds the fetched word in a parsing state until all valid instructions are extracted and forwarded to the DU, before going back to normal fetching states. Both discussed cases, 1 and 2, are ideal, not necessarily occurring all of the time "naturally". However, if desired to be the only occurring cases "artificially", in order to reduce the FU's FSM complexity, a dedicated assembler can be designed to fill any word gaps with 8-bit NOP instructions, ensuring that memory words contain only complete instructions. This option eliminates any occurrence of Case 3, which adds more complications to the FU's FSM, but may also compromise code precision in timing applications, and slightly increase power consumption.
3. The word fetched has an incomplete fragment of a 16-bit, 24-bit, or 32-bit instruction that must be completed by fetching the word in the next memory address. In this case, the FU's FSM holds the fragment after forwarding any complete instructions found in the fetched word to the DU. Holding the fragment, the FSM then initiates a new fetch to complete it and forward it to the DU, under an asserted flag indicating a pending completion. Once that is done, the flag is deasserted and the FSM repeats the process handling one of the three possible cases. This case the naturally-occurring case if instructions are "lined up" one after another in memory words. Figure 3.3 shows an example of a memory word that contains two complete 8-bit instructions followed by a fragment of a 24-bit instruction. The latter's other fragment is found in the next memory address, followed by a complete 24-bit instruction.

Figure 3.3: *Example Case of a Fragmented Instruction.*



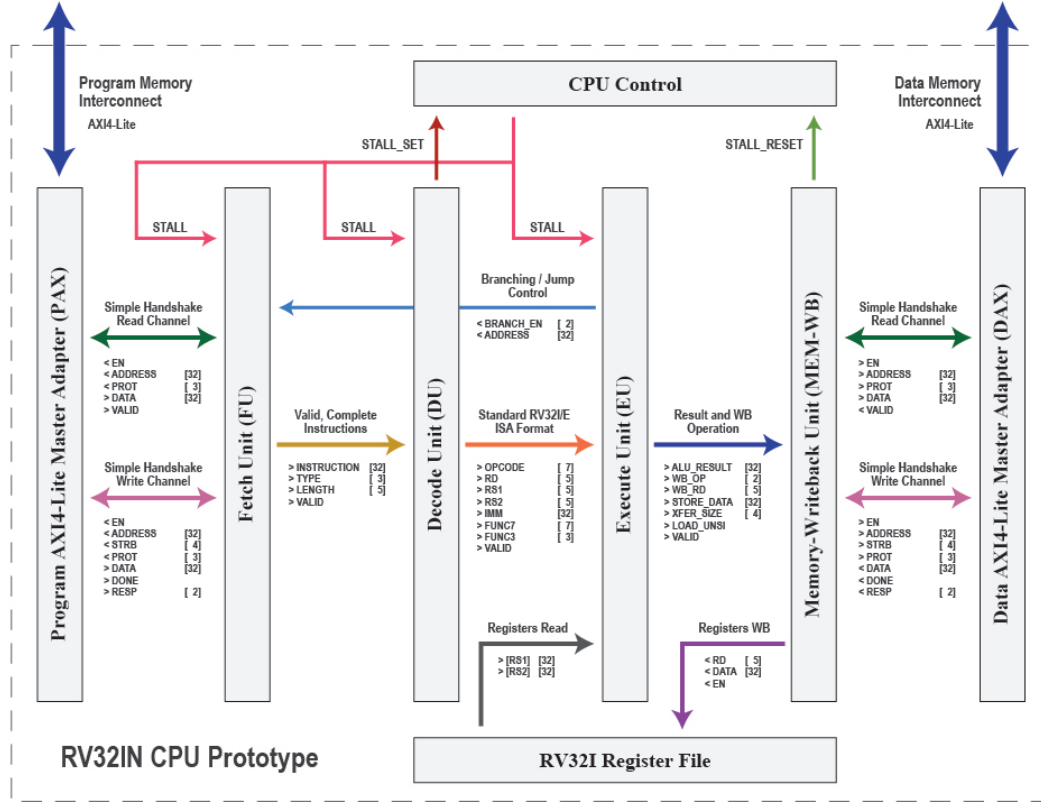
In the prototyped RV32IN, with its implementation discussion left for 3.5.2, the FU has four FSM states: FETCH, WAIT, PARSE and STALL. In the FETCH state, the FU determines the next fetch address, depending on the presence of a stall, branch, or jump signal. If no flow control signals are present, the state updates the fetching address to the next memory word address normally, and asserts the fetch enable signal in the next clock cycle, moving to the WAIT state. In WAIT, which stands for waiting for memory response, the FU remains in the state until a valid fetch is received, while still monitoring the presence of any flow control signals. If a valid fetch is received, the fetch enable is deasserted in the next cycle, and the FSM moves to the PARSE state. PARSE is where the words are handled, tracking the number of bytes passed (of instructions), held (as a fragmentation), and pending (to complete the fragmentation), as well as the mentioned flag to determine if a completion fetch is needed. The PARSE state extracts valid instructions from words and passes them to DU, updating the tracking parameters in the process. Depending on these variables and the obtained opcodes, as well as any flow control signals, the next state is determined. Finally, STALL state is just a simple state that moves the address pointer backwards in the presence of a pipeline stall, as the word would be flushed and has to be fetched again. Figure 3.4 shows a focused waveform of the prototyped RV32IN's FU, showing these details in a case of fragmentation.

2. A simpler approach is to develop a dedicated software tool to convert standard machine code into mixed machine code, utilizing NOVLI-ISA instructions where applicable, based on algorithmic application of predefined rules. This approach is implemented in Section 3.6 as part of the evaluation of NOVLI-ISA.
3. In rare practical cases, such as in the context of precision timing applications or RTL testing, RV32 assembly codes are written manually. NOVLI-ISA instructions can be used where applicable; however, this still dictates developing an assembler tool to convert these codes to hexadecimal and binary format, organizing them properly in memory. The organization part of variable-length instructions has proven to be a tedious task when done manually, requiring the development of automation tools with rules of alignment and handling fragmentation and memory gaps.

3.5.2 *Designing the RV32IN Prototype*

The RV32IN is a prototype soft-core RV32 CPU that supports the execution of NOVLI-ISA instructions. The I suffix in its designation represents the RV32I ISA base integer, and the N represents the support for the proposed NOVLI-ISA. The RV32IN has been implemented using the same tools as used for the OzU SoC: Verilog and Vivado. This sample CPU prototype is implemented to demonstrate the functionality and performance of the proposed ISA and its hardware requirements. The design has been developed in an incremental approach, where it was first implemented as a standard RV32I CPU, ensuring a successful run of a pre-written, pre-tested code: the OzU SoC's self-test routine. The RV32IN is integrated within a sample SoC environment reused from the OzU SoC, consisting of the SOTP, SRAM, and one GPIO port. Contrary to the OzU SoC with PicoRV32 at its core, the RV32IN and its sample SoC environment are organized in Harvard architecture, having separate interconnects for program and data communications. The signal-detailed block diagram in 3.5 shows the exact architecture of the RV32IN.

Figure 3.5: Block Diagram of the RV32IN Prototype.

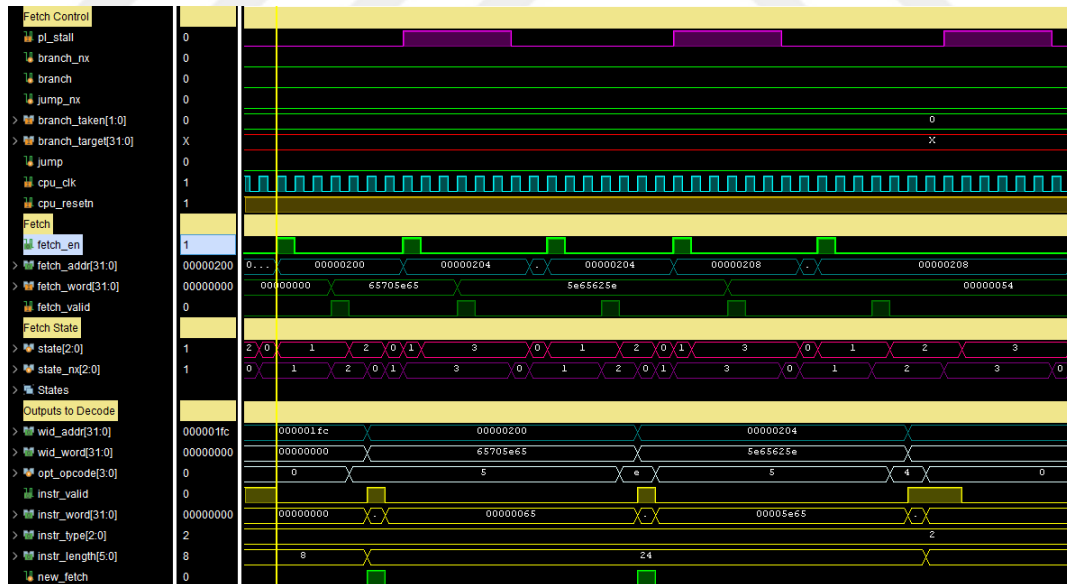


The RV32IN CPU prototype is a modular multi-cycle, four-stage pipelined CPU. The pipeline stages are: Fetch, Decode, Execute and Memory-Writeback (MEM-WB), supplemented by a basic control unit to handle pipeline stalls occurring while waiting for memory responses. The FU controls the program fetching from the program memory, handling NOVL-ISA as explained in 3.5.1 in detail. The DU handles the decoding of both valid standard and NOVL-ISA instructions from the FU, forwarding their end parameters to the EU in standard RV32I format. The EU performs the ALU operation and determines the writeback operation type, as well as generating branching and jump signals to the FU. Finally, the Memory-Writeback handles writebacks to either the register file or the data memory. Although the DU and EU are simple one-cycle units, the FU and MEM-WB stages can take several cycles to complete a task, depending on the memory response.

3.5.3 Coding, Simulation and Testing

It should be evident that the NOVL-ISA has no existing compilers or assembler tools; therefore, the testing program of the RV32IN prototype has been developed using primitive makeshift tools, combining spread sheets and manual assembly writing and embedding it in the SOTP's Verilog. Developing the small program needed to test the RV32IN prototype did not utilize automation; instead, it focused on testing and verifying the NOVL-ISA formats one by one, focusing on each bit of them while refining the ISA definition and the hardware in the process. Automation has been left to evaluate the feasibility of the proposed ISA in 3.6. Figure 3.6 shows a successful test of the RV32IN prototype, mainly performing three store instructions, of which one is fragmented. The figure also shows the pipeline stall signal generated by the Memory-Writeback Unit while waiting for a valid data memory response on the AXI interconnect.

Figure 3.6: RV32IN Simulation Handling Fragmented Store Instructions.



3.6 Feasibility Evaluation Results

The centric motivation for developing the NOVLI-ISA is to reduce the memory requirements to store machine codes. This is achieved through the ISA’s variable-length formats and its improved utilization of bits. This centric motivation is also the primary metric in evaluating the feasibility of the proposal. Secondary evaluation metrics should also be investigated eventually, such as the added complexity to support the NOVLI-ISA, and its consequences on the PPA aspects of compatible hardware. Detailed in the following, typical machine code size reduction resulting of utilizing the NOVLI-ISA can be evaluated through converting standard RV32I machine codes into their mixed, RV32I and NOVLI-ISA equivalents, observing size reduction. However, the secondary metrics of evaluation in terms of added hardware complexity and its consequences on the PPA aspects are left for future research, especially that such an evaluation may require advancing into some realization stages. Such extended evaluation requires completing and testing two exact-match, fully functional RV32I and RV32IN prototypes and comparing their PPA metrics under different operational scenarios. Although almost certainly theoretically proven to reduce machine code size, the only question that remains about the NOVLI-ISA is the reduction it can typically achieve. Serving as evaluation codes, a set of RV32I machine codes have been obtained from GitHub¹ [71, 72, 73, 74, 75], added to another set of typical-usage MCU programs², and standard benchmarks³, which have been written in C and automatically compiled to RV32I assembly codes. A dedicated optimizer script, written in Python, is developed to batch-convert these standard machine codes into their mixed, RV32I and NOVLI-ISA, assembly equivalents. Converting the batch, the program produces fine evaluation outputs on the instruction, program, and batch levels.

¹The codes obtained from these five repositories are 39 small programs in total.

²The written programs perform typical operations in MCU environments, such as peripherals and communications setup, sensor data acquisition, controlling robotic systems and more.

³Examples on these benchmarks used are Dhrystone, SPEC CPU2006, MiBench, TinyBench and more.

Table 3.11: *NOVLI-ISA Optimization Rules.*

Optimization Rule	Condition	Reduction
No Operation	Arithmetic instruction (R/I) writing to x0	-24 bits
RV32E Reg. Reset to Zero	Arithmetic inst. (R/I) writing 0 to x1-x15	-24 bits
RV32E Reg. Increment	<i>addi</i> — $\text{imm} = 1$ and $\text{rs} = \text{rd}$ (x1-x15)	-24 bits
RV32E Reg. Decrement	<i>addi</i> — $\text{imm} = -1$ and $\text{rs} = \text{rd}$ (x1-x15)	-24 bits
Unary Reg. Operation	Any R-type — $\text{rd} = \text{rs1} = \text{rs2}$	-16 bits
Reg.-Reg. Operation	Any R-type instruction	-8 bits
Small Immediate	Any register-immediate — $\text{imm} \leq 5\text{bits}$	-8 bits
Small Mem. Offset	Any load or store — $\text{offset} \leq 6\text{bits}$	-8 bits
Small Branch. Offset	Branching instruction — $\text{offset} \leq 6\text{bits}$	-8 bits
Small J Target	<i>jal</i> — $\text{rd} = \text{x0}$ and $\text{offset} \leq 10\text{bits}$	-16 bits
Small JAL Target	<i>jal</i> — $\text{offset} \leq 5\text{bits}$	-16 bits
Small JR Target	<i>jalr</i> — $\text{rd} = \text{x0}$ and $\text{offset} \leq 8\text{bits}$	-8 bits
Small JALR Target	<i>jalr</i> — $\text{offset} \leq 13\text{bits}$	-8 bits

The optimizer is a comprehensive multifunctional Python script. It first takes an input batch of text files containing standard RV32I hexadecimal machine codes, then disassembles them back to standard assembly. From that, it attempts to apply a set of programmed NOVLI-ISA optimization rules to each instruction, starting from the most efficient solution. Some optimizations may be less efficient than others; for example, the 'NOP' rule, which detects any write to the x0 register, has the highest priority. This rule produces the best size reduction of -24 bits per instruction, resulting in an 8-bit NOP/SWAIT instruction. For each input instruction, an output instruction is generated, either from the NOVLI-ISA or the standard RV32I ISA if optimization is not possible. Table 3.11 shows the rules and conditions of their application, ordered according to their priority of application, highest priority first. Finally, a Comma-Separated Values (CSV) file is constructed for each input program. This CSV file, instance shown in Figure 3.7, includes the input (column C), the standard disassembly (column D), the optimization rule (column F), the NOVLI-ISA reassembly (column I) and size reduction (column G).

Figure 3.7: Script Output of Optimizing a Single Input Program.

	A	B	C	D	E	F	G	H	I
1	No.	Address	Std. Instr. (Hex)	Disassembly (Stc)	Std. Size (bits)	Optimization Rule	Reduction (bits)	NOLVI-ISA Instr.	Opt. Size (bits)
2	1	0x4	ff010113	addi x2, x2, -16	32	Small Immediate I-Type (<5 bits)	-8	addi x2, x2, -16	24
3	2	0x8	00112623	sw x1, 12(x2)	32	Small Memory Offset (<6 bits)	-8	sw x1, 12(x2)	24
4	3	0xc	00812423	sw x8, 8(x2)	32	Small Memory Offset (<6 bits)	-8	sw x8, 8(x2)	24
5	4	0x10	01010413	addi x8, x2, 16	32	No Optimization	0	addi x8, x2, 16	32
6	5	0x14	01c000ef	jal x1, 28	32	No Optimization	0	jal x1, 28	32
7	6	0x18	00000793	addi x15, x0, 0	32	Small Immediate I-Type (<5 bits)	-8	addi x15, x0, 0	24
8	7	0x1c	00078513	addi x10, x15, 0	32	Small Immediate I-Type (<5 bits)	-8	addi x10, x15, 0	24
9	8	0x20	00c12083	lw x1, 12(x2)	32	Small Memory Offset (<6 bits)	-8	lw x1, 12(x2)	24
10	9	0x24	00812403	lw x8, 8(x2)	32	Small Memory Offset (<6 bits)	-8	lw x8, 8(x2)	24
11	10	0x28	01010113	addi x2, x2, 16	32	No Optimization	0	addi x2, x2, 16	32
12	11	0x2c	00008067	jalr x0, 0(x1)	32	Short JR Target (<8 bits)	-8	jr 0(x1)	24
13	12	0x30	fe010113	addi x2, x2, -32	32	No Optimization	0	addi x2, x2, -32	32
14	13	0x34	00112e23	sw x1, 28(x2)	32	Small Memory Offset (<6 bits)	-8	sw x1, 28(x2)	24
15	14	0x38	00812c23	sw x8, 24(x2)	32	Small Memory Offset (<6 bits)	-8	sw x8, 24(x2)	24
16	15	0x3c	02010413	addi x8, x2, 32	32	No Optimization	0	addi x8, x2, 32	32
17	16	0x40	fe042623	sw x0, -20(x8)	32	Small Memory Offset (<6 bits)	-8	sw x0, -20(x8)	24
18	17	0x44	0c00006f	jal x0, 192	32	Short J Target (<10 bits)	-16	j 192	16
19	18	0x48	000007b7	lui x15, 0	32	No Optimization	0	lui x15, 0	32
20	19	0x4c	00200713	addi x14, x0, 2	32	Small Immediate I-Type (<5 bits)	-8	addi x14, x0, 2	24
21	20	0x50	00e7a423	sw x14, 8(x15)	32	Small Memory Offset (<6 bits)	-8	sw x14, 8(x15)	24
22	21	0x54	000007b7	lui x15, 0	32	No Optimization	0	lui x15, 0	32
23	22	0x58	00300713	addi x14, x0, 3	32	Small Immediate I-Type (<5 bits)	-8	addi x14, x0, 3	24
24	23	0x5c	00e7a623	sw x14, 12(x15)	32	Small Memory Offset (<6 bits)	-8	sw x14, 12(x15)	24
25	24	0x60	000007b7	lui x15, 0	32	No Optimization	0	lui x15, 0	32

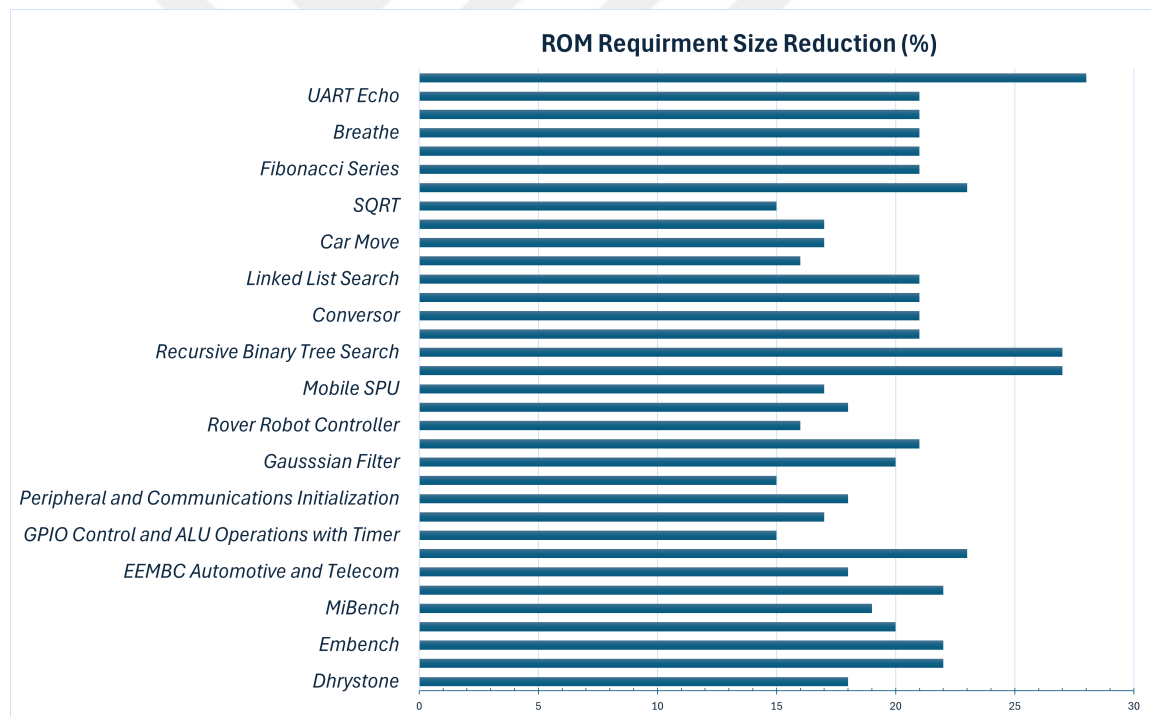
Moreover, analytics CSV file outputs are generated for both individual input programs and the entire batch. Figure 3.8 shows the overall analytics file, post-formatted in Excel. These files contain additional useful analysis of the input batch and optimizations, including instruction frequency-of-occurrence analysis in the input programs, and counting the application of optimization rules. These secondary analytics outputs could be useful for further statistically based improvements of the NOVLI-ISA definition.

Figure 3.8: Script Analytics Output for the Overall Batch.

Location	Car Move	Print Image	SQRT	Testbench for Decoder	Fibonacci Series	Blink	Breathe	Shp2	UART Echo	Find Multiple	Total	Average	
257	69	69	109	48	37	37	37	37	37	32	5.808	0.171	Total Instructions (x1000)
1028	276	276	436	192	148	148	148	148	148	128	22.688	0.667	Original Size (KB)
-167	-48	-48	-66	-45	-32	-32	-32	-32	-32	-36	-4.552	-0.134	Total Optimization (KB)
861	228	228	370	147	116	116	116	116	116	92	18.136	0.533	Optimized Size (KB)
16	17	17	15	23	21	21	21	21	21	28		28.000	Optimization (%)
rep1-07 Location	rep1-08 Car Move	rep1-09 Print Image	rep1-10 SQRT	rep2 Testbench for Decoder	rep3 Fibonacci Series	rep4-01 Blink	rep4-02 Breathe	rep4-03 Shp2	rep4-04 UART Echo	rep5 Find Multiple	Total	Average	Average Percentage
105	27	27	46	13	16	16	16	16	16	7	474595	61	10.0449%
0	6	6	0	7	3	3	3	3	3	0	472540	19	10.0014%
42	9	9	22	4	9	9	9	9	9	7	945427	19	20.0101%
0	3	3	0	10	3	3	3	3	3	0	472037	24	9.9907%
17	7	7	8	3	3	3	3	3	3	2	472319	7	9.9967%
0	5	5	0	0	0	0	0	0	0	0	471440	6	9.9781%
0	4	4	0	2	0	0	0	0	0	0	59	2	0.0012%
7	0	0	6	2	1	1	1	1	1	8	471409	6	9.9774%
7	1	1	3	2	0	0	0	0	0	0	53	2	0.0011%
0	0	0	0	0	2	2	2	2	2	3	52	2	0.0011%
													addi
													sw
													jal
													lw
													jalr
													lui
													stli
													add
													bge
													bne

Applying the optimizer script to the previously detailed batch of evaluation programs, the overall optimization of these codes has resulted in an average reduction in size of 20%, with a minimum reduction of 15% and a maximum of 28%, individual results shown in Figure 3.9. Secondary analytics have shown that the most frequently applied optimization rule is 'small memory offset' for load and store operations. This rule accounts for approximately 20% of all applied rules, including the 'No optimization possible' rule. Other contributing rules in size reductions are the short-format *j*, *jr*, branching, and register-immediate instructions, as well as the reduced fixed-size register-register format, accounting for 50% of size reductions together.

Figure 3.9: *Code Size Reduction Percentages of the Evaluation Batch.*



3.7 Integrating the RV32IN Prototype within the OzU SoC

Designed to be modular and scalable, the OzU SoC can be modified to have the RV32IN prototype at its core. Memories and peripherals remain the same with no changes, while the only reconfiguration required would be in the AXI switches' connectivity to the CPU, as the PicoRV32 is designed for Von Neumann architecture, while the RV32IN is designed for Harvard. The NOVLI-ISA in turn is designed for compatibility; therefore, the RV32IN does not require any additional hardware in the SoC's environment.

3.8 Future Research for the NOVLI-ISA

Both the NOVLI-ISA and its supporting RV32IN implementation are prototype concept demonstrators. Designed to be flexible and modifiable, the NOVLI-ISA guides several directions for future improvements and further research. Future refinements of the NOVLI-ISA may include the following:

- Designing a comprehensive and robust configurable software toolchain for compiling source C/C++ codes into mixed, RV32I and NOVLI-ISA, assembly and machine codes. This toolchain should also include a dedicated assembler and linker tool that can handle automatic memory placement of the variable length of the NOVLI-ISA, considering flexible support for the hardware designers' preference in handling the memory organization and fetching cases discussed in 3.5.1.
- This toolchain can also be extended to generate NOVLI-ISA codes from other RV32 extensions, particularly from the noninclusive RV32C instructions.
- Improving the utilization of the NOVLI-ISA by designing a dedicated optimizing compiler that can track the register values and detect data dependencies, also as detailed in 3.5.1. Such a compiler is expected to increase size reduction significantly, although careful considerations should be taken to mitigate data integrity hazards.

- Further qualitative and quantitative analysis and evaluation of RV32I machine codes and their NOVLI-ISA equivalents could result in more refinements of the ISA definition and possible alternative ways of prioritizing rules. Such analysis and evaluation may include evaluating more programs, categorizing them per application, and gathering more statistics about the frequency of occurrence of instructions.
- Based on the mentioned extended analysis and evaluation, establishing dynamic reconfigurability mechanisms of the NOVLI-ISA base definition for application-specific implementations could be feasible, which could further increase the reduction in machine code size in their particular application domains.
- Comprehensive analysis of the PPA consequences on compatible hardware resulting from supporting the NOVLI-ISA which, as detailed in the introduction of 3.6.

4. CONCLUSIONS

This concluding chapter of the thesis report summarizes the final results of the research, comparing them against the primary research objectives and technical scope, and pointing out further future research directions. Drawing a conclusion to this research, a concise summary of the work outcomes is underscored, highlighting what has been achieved and what should be done next, for both the OzU SoC and the NOVLI-ISA.

4.1 OzU SoC as a Testbed for Future Development

Implementing the OzU SoC as a soft IP and an FPGA prototype, its development demonstrates a practical case of integrating a small- to medium-scale user-specified single-core microcontroller SoC design. The development of the OzU SoC provides essential technical insights for working with open-source hardware architectures and third-party IPs, focusing on the RV32 architecture. This development reveals the detailed workflow of conceptualizing a microcontroller SoC's design and planning its architecture, starting with the selection of the CPU at its core, and into the organization of memory hierarchy and planning on-chip interconnects. Going further into integration, the development exemplifies several means of integrating other SoC components, mainly peripherals, such as obtaining and adapting them or custom-designing them from the ground up. Documenting the process, the development explains several essential theoretical and practical methodologies associated with SoC design, implementation, and realization, covering both the hardware and software aspects. The design of the OzU SoC also introduces some innovative features and emphasizes automation. Autonomous buffering in serial transceivers is an example, as well as implementing the Pin Functions Controller to optimize the utilization of physical package resources. Automation is used or suggested in some areas, particularly for machine code generation and RTL customizability.

The final outcome of the OzU SoC is a customizable, reconfigurable, and scalable soft RTL design, demonstrated with limitations as an FPGA prototype. Such limitations in prototyping emerged mainly due to logistical reasons: the available FPGA development board and its resource limitations, as well as the availability of external hardware for full functional testing and debugging. Although an overall functional and synthesizable RTL design is available, the established OzU SoC design is subject to further testing, refinements, and improvements to be considered for advancing into realization. The deliverables of the OzU SoC development are summarized as follows:

1. The RTL design written in Verilog and deliverable as a Xilinx Vivado project. The project files are modular, parameterized, and well documented for future modifications or improvements. Depending on the preference of the IP owners, Özyeğin University, this prototype design can be published on GitHub or similar open-source hardware platforms for community contributions, or as an academic reference. The design is subject to further micro-functions completion, revisions, and refinements. It can be considered a first-Alpha version in software life cycle terms.
2. A reduced, FPGA-specific version of the full OzU SoC RTL design is particularly reconfigured to be prototypable on Digilent's Basys 3 development boards, operating a mid-range Artix-7 series FPGA. This version runs a sample SPROM-stored operational GPIO program, validating the SoC's core architecture, functionality, timing and data integrity, and synthesizability. A more extended version can be prototyped if adequate hardware is available, derived from the full RTL design.

There are known technical issues in the final OzU SoC design, and others that may remain unknown until a complete refinement and prototyping of the design is possible. Known issues and suggested refinements and improvements are summarized as follows:

1. In consequence of the discussed available hardware limitations, the ADC requirement for the OzU SoC was never implemented due to the absence of external hardware and testing environment. A dummy unit could have been made to emulate an ADC acquiring random data, but this implementation would have served no value for the workflow or results, aside from redundancy and non-useful area utilization. The unimplemented components, some of which are optional, are previously listed in Table 2.9 in Chapter II with the main reasons why they were not implemented.
2. Although successfully adapted and simulated, both the SPI and UART transceivers can be upgraded to more advanced and robust versions, similar to the more comprehensive GPIO and I2C IPs obtained. The IP of the obtained UART transceiver lacks runtime configuration capabilities, while the SPI transceiver proved to be not fully synthesized. Moreover, the DSPI and QSPI functions that made the obtained SPI IP an appealing choice were never tested in the final Alpha version. Upgrading the UART may be optional; although static configuration is a performance setback and a non-standard practical implementation, the fact that the OzU SoC has four independent UART transceivers and that it is intended for application-specific design customization could make it a feature. The current static-configuration UART implementation still offers an advantage of less hardware complexity; thus less power and area utilization. The four transceivers can be preconfigured with typical-usage configurations, or they could be configured for the exact requirements of the end application of the OzU SoC. However, the SPI transceiver has to be revised or replaced, as the synthesizability requirement is indispensable for prototyping and realization.

3. The present non-vectorized, non-nested interrupt controller is a simple hardware unit that collects all interrupt signals from the SoC's components to generate a one-bit master interrupt to the CPU, providing individual interrupt masking. In the future, a fully functional NVIC unit could be implemented, which may require modifications to the PicoRV32 IP.
4. Although the self-test routine operates all components of the OzU SoC, it only performs operations without checking their results. It is more of a connectivity check execution program than a functionality check program. This code should be revised to check for correct execution and data integrity, where applicable, indicating faults and altering the program flow accordingly.
5. Due to their elimination in the FPGA prototype version for the logistical reasons discussed above, all serial interfaces have not yet been mapped through the PFCN unit. This is more of a thing made on purpose than a design bug. Due to the well-organized structure of the design, these can be easily completed later upon the availability of a more capable FPGA platform.
6. Optionally, as detailed in Chapter II, many other units can be integrated on-demand within the OzU SoC's scalable environment, such as external memory controllers, debugging interfaces, clock controllers, cryptographic elements, etc.

In summary, with its core architecture and most of its key functions completed and tested, the OzU SoC serves as an initial and scalable prototype for further development, testing, and later realization. Other issues may remain unknown until a more extensive testing of the SoC is made through both behavioral simulation and FPGA prototyping.

4.2 Contributing the NOVLI-ISA to the RISC-V Community

Supplementing OzU SoC, committing to the SoC optimization part following integration, this thesis introduced, implemented, tested, and evaluated NOVLI-ISA: a compatible optimizing extension to the base RV32I ISA. The primary objective of developing the NOVLI-ISA is to reduce the machine code size, reflecting on its physical storage requirements. The proposed extension introduces variable-length instruction formats with all-inclusive and flexible functionalities, of which some are tailored for operations in the embedded RV32E architecture. The NOVLI-ISA inherits simplicity, modularity, and flexibility practices from the design philosophy of the RISC-V architecture.

Demonstrating the applicability and functionality of the NOVLI-ISA, a supporting prototype RV32 CPU core is developed, designated the RV32IN. It is shown that achieving a compatible RV32 CPU requires limited and modular modifications, which can be layered over a standard RV32I implementation. The main hardware modifications to support the NOVLI-ISA affect the fetch and decode stages, and optionally, the execution. The RV32IN prototype is successfully tested within a minimal SoC environment similar to the OzU SoC, running the standard RV32I, then the NOVLI-ISA machine code. The RV32IN prototype is synthesizable and written in standard Verilog RTL practices. The core can be integrated seamlessly within the OzU SoC's prototype with no additional SoC components required, replacing the PicoRV32 at the core.

To evaluate the feasibility of NOVLI-ISA in terms of size reduction, a dedicated script program is implemented to convert batches of standard RV32I machine code programs into their NOVLI-ISA equivalents. The program outputs detailed analytics about size reduction, as well as other optimization and statistical data. Promising to offer a similar typical size reduction to the officially ratified RV32C extension upon evaluation, the NOVLI-ISA could be a competitive alternative addressing the functional noninclusion of the RV32C extension definition.

The NOVLI-ISA proposal provides an intellectual contribution to the open RISC-V community, opening further directions for the research and investigation of variable length instructions in RISC-V. The NOVLI-ISA deliverables are summarized as follows:

1. The ISA definition blueprint of the NOVLI-ISA, detailing its formats, instructions, and their exact functionalities, which is appended as Appendix L.
2. The RV32IN within its minimal SoC environment is a similar deliverable to the OzU SoC design, also implemented in Verilog as a Xilinx Vivado project. This prototype serves as a testbed for future refinement and testing, eventually testing the NOVLI-ISA fully integrated within the OzU SoC design.
3. The general guidelines for storing and handling the variable length instructions of NOVLI-ISA, detailed in 3.5.1, are essential insights for hardware designers to be able to design compatible hardware. Although the detailed focus is on fetching design from word-organized memories, the general concept still applies for byte-based memories, leaving alternative implementation methods open to designers.
4. The optimizer script designed to convert standard RV32I assembly codes to their equivalents in the NOVLI-ISA, and analyze them, is an essential asset in the NOVLI-ISA environments enabling automated generation and evaluation of NOVLI-ISA codes. This modular program could be seamlessly extended to generate optimized codes in hexadecimal format and produce them as Verilog-coded memories.

There are no known technical issues in the theory of the NOVLI-ISA definition. Nevertheless, as stated, the RV32IN is subject to full completion and more extensive testing. The work in Chapter III of conceptualizing, designing, implementing, and testing the NOVLI-ISA is all submitted in a conference paper, accepted at the 20th East-West Design and Test Symposium (EWDTS).

4.3 Future Research Directions

The combined nature of the research conducted in this thesis reveals several aspects of SoC integration and optimization from both the macro- and micro-perspectives. These different aspects of the development, as well as its results, highlight further directions for future research and investigation:

1. Given the ultimate hypothesized goal of achieving a highly customizable and autonomously reconfigurable design of the OzU SoC, further research should be conducted to establish a highly flexible hardware and software framework to fulfill this goal. This includes refining the current OzU SoC prototype into an even more flexible version, ensuring fail-safe mechanisms to prevent design mismatches caused of reconfigurability. From that, a dedicated software tool can be developed to rapid-customize and verify application-tailored versions, as discussed in 2.6.1.
2. It is possible to explore the merits of an alternative, customizable multi-core exclusively-RV32E architecture for the OzU SoC instead of its current single-core RV32IMC architecture. This could possibly extend its flexibility and reconfigurability while introducing hardware simplicity and homogeneity. Multi-core architectures exhibit lots of merits and flexibility, though adding some design challenges of their own.
3. As detailed in 3.8, the NOVLI-ISA with its hardware and software tools can be further improved, following a more extensive development and testing of the RV32IN prototype, combined with performing more analytics on larger numbers of machine codes. The ISA could possibly have flexible application-specific versions that could drastically improve size reduction in their domains. Most importantly, it is fundamental to complete the software toolchain discussed in Section 3.8, which enables automatic coding of the NOVLI-ISA and promises more size reduction.

4.4 Summary of the Thesis

This thesis presents a dual research in the field of SoC integration and optimization based on the RISC-V architecture. Approaching the research from both theoretical and practical perspectives, the first part of the thesis focuses on the design, implementation, and prototyping of the OzU SoC, a scalable and customizable RV32I-based microcontroller SoC design. In this part, the thesis reveals the foundational aspects of SoC design while addressing key development challenges encountered in the process. Despite resulting in a prototype in need of further refinement, the OzU SoC serves as a testbed for future improvements and research, particularly in the context of design and customizability automation. Following further development, the OzU SoC can be realized and deployed on the availability of necessary resources. The second part of the thesis addressing ISA optimization introduces the NOVLI-ISA: a compatible, all-inclusive flexible extension to the standard RV32I ISA. Designed to reduce machine code size as the primary goal, the NOVLI-ISA demonstrates its applicability through the development and testing of a prototype processor, the RV32IN. To demonstrate its feasibility in size reduction, an optimization and analysis software tool is developed, providing valuable insight for future improvements. The results of the NOVLI-ISA are published and accepted as a peer-reviewed publication, further emphasizing its academic and industry significance. In conclusion, the thesis combines theory and literature with the practical engineering perspective to demonstrate the integration and optimization of microcontroller SoC designs based on the RISC-V architecture, revealing the fundamental methodologies and challenges in modern SoC design. The deliverables of this thesis include a functional soft IP SoC design, and a novel feasibility-proven ISA extension to the RV32I ISA. Future work should focus on improving customizability and automation, as well as refining and improving deliverables, ensuring that this research continues to contribute to both the academic and industry community of SoC and RISC-V designers.

REFERENCES

- [1] S. Chandrasekaran, “System-on-Chip (SoC) design: Paving the path to integrated innovation,” August 2023. <https://medium.com/%40chandrasyah999/system-on-chip-soc-design-paving-the-path-to-integrated-innovation-428150cfebef> [Accessed December 2024].
- [2] S. B. Furber, *ARM System-on-Chip Architecture*. Addison-Wesley, 2nd ed., 2000.
- [3] R. Triggs, “What is an SoC? Everything you need to know about smartphone chipsets,” May 2023. <https://androidauthority.com/what-is-an-soc-smartphone-chipsets-explained-1051600> [Accessed December 2024].
- [4] Promwad Electronics Design House, “Modern System-on-Chip: Features and applications,” 2024. <https://promwad.com/publications/modern-system-on-chip> [Accessed: December 2024].
- [5] Semiconductor Engineering, “RTL (Register Transfer Level),” 2023. https://semiengineering.com/knowledge_centers/eda-design/definitions/register-transfer-level [Accessed December 2024].
- [6] C. Morrison, “Simplifying analog and mixed-signal design integration,” 2024. <https://agileanalog.com/blog-posts/simplifying-analog-and-mixed-signal-design-integration> [Accessed December 2024].
- [7] A. A. Jerraya, S. Yoo, D. Verkest and N. Wehn, *Embedded Software for SoC*. Springer, 2003.
- [8] A. Chopde and A. M. Kulkarni, “Physical design: Methodologies and developments,” *arXiv*, vol. 2409, pp. 1–15, September 2024.
- [9] B. Kapoor, J. Marc Edwards, S. Hemmady et al., “Tutorial: SoC power management verification and testing issues,” in *Proc. International Workshop on Microprocessor Test and Verification (MTV)*, pp. 67–72, 2008.
- [10] M. Girish, G. Gopakumar and D. S. Divya, “Formal and simulation verification: Comparing and contrasting the two verification approaches,” in *Proc. International Conference on Advances in Computing, Communication, Embedded and Secure Systems (ACCESS)*, pp. 41–44, 2021.
- [11] J. Stahl, “FPGA prototyping powers the SoC design verification process,” 2021. <https://synopsys.com/blogs/chip-design/fpga-prototyping-powers-soc-process.html> [Accessed December 2024].
- [12] R. Zafar, “Exynos 9820 die image shows dedicated L3 cache and upgraded design,” 2019. <https://wccfttech.com/exynos-9820-die-shot> [Accessed December 2024].

- [13] B. Daniel, “What is Automatic Test Equipment (ATE)?,” 2021. <https://trentonsystems.com/en-us/resource-hub/blog/automatic-test-equipment-overview> [Accessed December 2024].
- [14] V. P. Sampath, “Post-silicon validation methodology in SoC (part 1 of 2),” 2018. <https://electronicsforu.com/electronics-projects/electronics-design-guides/post-silicon-validation-methodology> [Accessed December 2024].
- [15] SparkFun Electronics, “Integrated circuits,” 2024. <https://learn.sparkfun.com/tutorials/integrated-circuits/inside-the-ic> [Accessed December 2024].
- [16] S. Sharma, “RISC-V architecture: A comprehensive guide to the open-source ISA,” 2024. <https://wevolver.com/article/risc-v-architecture> [Accessed December 2024].
- [17] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann, 2017.
- [18] T. Simon, “Open source RISC-V ISA brings a new wrinkle to the processor market,” 2017. <https://semiwiki.com/ip/sifive/7089-open-source-risc-v-isa-brings-a-new-wrinkle-to-the-processor-market> [Accessed December 2024].
- [19] A. Waterman and K. Asanović, “The RISC-V instruction set manual, volume i: Unprivileged ISA, version 20191213,” technical manual, SiFive and UC Berkeley, 2019.
- [20] A. Waterman, Y. Lee, D. A. Patterson and K. Asanović, “The RISC-V compressed instruction set manual, version 1.9,” Technical Manual UCB/EECS-2015-209, UC Berkeley, 2015.
- [21] E. Corpeño, “An introduction to RISC-V—Understanding RISC’s open ISA,” 2022. <https://allaboutcircuits.com/technical-articles/introductions-to-risc-v-instruction-set-understanding-this-open-instruction-set-architecture> [Accessed December 2024].
- [22] W. Chen, S. Ray, J. Bhadra et al., “Challenges and trends in modern SoC design verification,” *IEEE Design Test*, vol. 34, pp. 7–22, 2017.
- [23] Y.-L. S. Lin, “SoC memory system design,” in *Essential Issues in SOC Design: Designing Complex Systems-on-Chip*, pp. 73–118, Springer, 2006.
- [24] C. Kwok, P. Viswanathan and P. Yeung, “Addressing the challenges of reset verification in SoC designs,” in *Proc. Design and Verification Conference (DVCon)*, pp. 1–9, 2015.
- [25] D. E. Lackey, P.S. Zuchowski, T.R. Bednar et al., “Managing power and performance for System-on-Chip designs using voltage islands,” in *Proc. IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pp. 195–202, 2002.

- [26] S. Pasricha, R. Ayoub, M. Kishinevsky et al., “A survey on energy management for mobile and IoT devices,” Survey 2168-2356, Colorado State University, 2020.
- [27] Synopsys, Inc., “Synopsys.ai – full stack, AI-driven EDA suite,” Brochure 11/23/23.CS1244151798, Synopsys, Inc., 2023.
- [28] Semiconductor Review, “The global semiconductor shortage: Causes, effects, and uncertain future,” 2023. <https://semiconductorreview.com/news/the-global-semiconductor-shortage-causes-effects-and-uncertain-future-nwid-643.html> [Accessed December 2024].
- [29] YosysHQ GmbH, “PicoRV32: A size-optimized RISC-V CPU core,” 2024. <https://github.com/YosysHQ/picorv32> [Accessed December 2024].
- [30] YosysHQ GmbH, “About YosysHQ,” 2024. <https://yosyshq.com/about> [Accessed December 2024].
- [31] “Gowin PicoRV32 hardware design,” Reference Manual IPUG914-1.7E, Gowin Semiconductor Corp., June 2024.
- [32] YosysHQ GmbH, “PicoSoC: A simple example System-on-Chip (SoC) using PicoRV32,” 2024. <https://github.com/YosysHQ/picorv32/blob/main/picosoc> [Accessed December 2024].
- [33] Efabless Corporation, “Raven PicoRV32: PicoRV32 integrated with raven open-source ASIC platform,” 2024. <https://github.com/efabless/raven-picorv32> [Accessed December 2024].
- [34] Efabless Corporation, “About Efabless Corporation,” 2024. <https://efabless.com/company> [Accessed December 2024].
- [35] O. Chadib, “PicoSoC image application: A PicoRV32-based image processing system,” 2024. https://github.com/omarchadib/picosoc_image_application [Accessed December 2024].
- [36] PULP Platform Team, “PULP: Parallel Ultra-Low Power Platform,” 2024. <https://github.com/pulp-platform/pulp> [Accessed December 2024].
- [37] A. Pullini, D. Rossi, I. Loi et al., “Mr.wolf: An energy-precision scalable parallel ultra low power SoC for IoT edge processing,” *IEEE Journal of Solid-State Circuits*, vol. 54, pp. 1970–1981, 2019.
- [38] SiFive, Inc., “Freedom: RISC-V SoC RTL repository.” <https://github.com/sifive/freedom> [Accessed December 2024].
- [39] “SiFive Freedom E300 platform datasheet,” Datasheet 1.0, SiFive, Inc., 2016.
- [40] “Arty FPGA board reference manual,” Datasheet Rev. C, Digilent, Inc., 2017.

- [41] “SiFive Freedom U500 platform datasheet,” Datasheet 1.0, SiFive, Inc., July 2016.
- [42] “Getting started with the Virtex-7 FPGA VC707 evaluation kit,” Datasheet 1.4.1, Xilinx Inc., 2015.
- [43] “STM32F407VG microcontroller datasheet,” Datasheet DM00037051, STMicroelectronics NV, 2024.
- [44] Ch. Peng and H. Janakiraman, “Understanding and using bootloaders in embedded system design,” Technical Manual SPRAAQ3, Texas Instruments Incorporated, 2007.
- [45] EFY Bureau, “Difference between flash memory and EEPROM.” <https://electronicsforu.com/technology-trends/learn-electronics/eprom-difference-flash-memory> [Accessed December 2024].
- [46] ITU Online, “What is Direct Memory Access (DMA).” <https://ituonline.com/tech-definitions/what-is-direct-memory-access-dma> [Accessed December 2024].
- [47] “SAM D21/DA1 family data sheet,” Datasheet DS40001882H, Microchip Technology, Inc., 2024.
- [48] “Using the XMEGA watchdog timer,” Application Note AN2654, Microchip Technology, Inc., 2018.
- [49] M. Cirstea, K. Benkrid, A. Dinu et al., “Digital electronic System-on-Chip design: Methodologies, tools, evolution, and trends,” *Micromachines*, vol. 15, no. 2, p. 247, 2024.
- [50] C. E. Cummings, “New Verilog-2001 techniques for creating parameterized models,” Technical Reference HDLCON 2002, Sunburst Design, Inc., 2005.
- [51] “AMBA AXI and ACE protocol specification,” Protocol Specification ARM IHI 0022D (ID102711), ARM Ltd., 2011.
- [52] S. Chang, “Verilog GPIO.” <https://github.com/schang412/verilog-gpio> [Accessed December 2024].
- [53] C. Capen, “UART serial transceiver written in Verilog HDL.” <https://github.com/ccapen/Verilog-UART> [Accessed December 2024].
- [54] A. Forencich, “Verilog I2C interface.” <https://github.com/alexforencich/verilog-i2c> [Accessed December 2024].
- [55] Abdul Wadood, “SPI master.” https://github.com/abdulwadood97/spi_master [Accessed December 2024].

- [56] RISC-V Collaboration, “RISC-V GNU compiler toolchain.” <https://github.com/riscv-collab/riscv-gnu-toolchain> [Accessed December 2024].
- [57] “Basys 3 FPGA board reference manual,” Reference Manual 502-183, Digilent, Inc., 2019.
- [58] S. W. Seong and P. Mishra, “A bitmask-based code compression technique for embedded systems,” in *Proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 251–254, 2006.
- [59] D. A. Huffman, “A method for the construction of minimum redundancy codes,” pp. 1098–1101, 1952.
- [60] M. Ros and P. Sutton, “A Hamming distance based VLIW/EPIC code compression technique,” in *Proc. International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*, pp. 132–139, 2004.
- [61] C. H. Lin, Y. Xie and W. Wolf, “LZW-based code compression for VLIW embedded systems,” in *Proc. Design, Automation and Test in Europe Conference (DAT)*, pp. 76–81, 2004.
- [62] G. B. Hwang, K. N. Cho, C. Y. Han et al., “Lossless decompression accelerator for embedded processor with GUI,” *Micromachines*, vol. 12, no. 2, p. 145, 2021.
- [63] P. Li, “Reduce static code size and improve RISC-V compression,” Technical Report UCB/EECS-2019-46, UC Berkeley, 2019.
- [64] S. Singh, A. Perais, A. Jimborean and A. Ros, “Exploring instruction fusion opportunities in general purpose processors,” in *Proc. IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, 2022.
- [65] “Extending RISC-V ISA with a custom instruction set extension,” technical paper, Codasip GmbH, December 2019.
- [66] M. Perotti, P. Davide, G. Tagliavini et al., “HW/SW approaches for RISC-V code size reduction,” *CARRV 2020*, pp. 1–7, 2020.
- [67] A. Wolfe and A. Chanin, “Executing compressed programs on an embedded RISC architecture,” in *Proc. Annual International Symposium on Computer Architecture (ISCA)*, pp. 80–88, 1992.
- [68] J. Liu, W. Gao, H. Liang et al., “Towards a universal and portable assembly code size reduction: A case study of RISC-V ISA,” *CCF Transactions on High Performance Computing*, vol. 6, no. 3, pp. 263–273, 2024.
- [69] A. Shriram, “RISC-V green card.” https://cs.sfu.ca/~ashriram/Courses/CS295/assets/notebooks/RISCV/RISCV_CARD.pdf. [Accessed December 2024].

- [70] A. Waterman, “Improving energy efficiency and reducing code size with RISC-V compressed,” M.S. thesis, UC Berkeley, 2011.
- [71] G. Horn, “rv32-assembly-tasks,” 2024. <https://github.com/GuilhermeHorn2/RV32-assembly-tasks> [Accessed December 2024].
- [72] ac618, “rv32i-decoder,” 2024. https://github.com/ac618/rv32i_decoder [Accessed December 2024].
- [73] A. Efi, “Fibonacci-in-rv32i,” 2024. <https://github.com/AmirEfi/Fibonacci-in-RV32I> [Accessed December 2024].
- [74] betteros, “tutorial-rv32-examples,” 2024. https://github.com/betteros/tut_rv32_examples [Accessed December 2024].
- [75] A. Efi, “Find-multiples-by-rv32i,” 2024. <https://github.com/AmirEfi/Find-Multiples-by-RV32I> [Accessed December 2024].



APPENDICES

APPENDIX A. RV32I/E ISA CARD

Core Instruction Formats

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

RV32I Base Integer Instructions

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ^ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	zero-extends
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ^ imm	
ori	OR Immediate	I	0010011	0x6		rd = rs1 imm	
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]	
srlr	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]	
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	zero-extends
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]	
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]	
sw	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]	
beq	Branch ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch ≥	B	1100011	0x5		if(rs1 ≥ rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch ≥ (U)	B	1100011	0x7		if(rs1 ≥ rs2) PC += imm	zero-extends
jal	Jump And Link	J	1101111			rd = PC+4; PC += imm	
jalr	Jump And Link Reg	I	1101111	0x0		rd = PC+4; PC = rs1 + imm	
lui	Load Upper Imm	U	0110111			rd = imm << 12	
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm << 12)	
ecall	Environment Call	I	1110011	0x0	imm=0x0	Transfer control to OS	
ebreak	Environment Break	I	1110011	0x0	imm=0x1	Transfer control to debugger	

APPENDIX B. RV32C EXTENSION CARD

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
funct4				rd/rs1				rs2				op		CR-type		
funct3		imm		rd/rs1				imm				op		CI-type		
funct3		imm						rs2				op		CSS-type		
funct3		imm								rd'		op		CIW-type		
funct3		imm			rs1'			imm		rd'		op		CL-type		
funct3		imm			rd'/rs1'			imm		rs2'		op		CS-type		
funct3		imm			rs1'			imm				op		CB-type		
funct3		offset										op		CJ-type		

Inst	Name	FMT	OP	Funct	Description
c.lwsp	Load Word from SP	CI	10	010	lw rd, (4*imm)(sp)
c.swsp	Store Word to SP	CSS	10	110	sw rs2, (4*imm)(sp)
c.lw	Load Word	CL	00	010	lw rd', (4*imm)(rs1')
c.sw	Store Word	CS	00	110	sw rs1', (4*imm)(rs2')
c.j	Jump	CJ	01	101	jal x0, 2*offset
c.jal	Jump And Link	CJ	01	001	jal ra, 2*offset
c.jr	Jump Reg	CR	10	1000	jalr x0, rs1, 0
c.jalr	Jump And Link Reg	CR	10	1001	jalr ra, rs1, 0
c.beqz	Branch == 0	CB	01	110	beq rs', x0, 2*imm
c.bnez	Branch != 0	CB	01	111	bne rs', x0, 2*imm
c.li	Load Immediate	CI	01	010	addi rd, x0, imm
c.lui	Load Upper Imm	CI	01	011	lui rd, imm
c.addi	ADD Immediate	CI	01	000	addi rd, rd, imm
c.addi16sp	ADD Imm * 16 to SP	CI	01	011	addi sp, sp, 16*imm
c.addi4spn	ADD Imm * 4 + SP	CIW	00	000	addi rd', sp, 4*imm
c.slli	Shift Left Logical Imm	CI	10	000	slli rd, rd, imm
c.srli	Shift Right Logical Imm	CB	01	100x00	srli rd', rd', imm
c.srai	Shift Right Arith Imm	CB	01	100x01	srai rd', rd', imm
c.andi	AND Imm	CB	01	100x10	andi rd', rd', imm
c.mv	MoVe	CR	10	1000	add rd, x0, rs2
c.add	ADD	CR	10	1001	add rd, rd, rs2
c.and	AND	CS	01	10001111	and rd', rd', rs2'
c.or	OR	CS	01	10001110	or rd', rd', rs2'
c.xor	XOR	CS	01	10001101	xor rd', rd', rs2'
c.sub	SUB	CS	01	10001100	sub rd', rd', rs2'
c.nop	No OPeration	CI	01	000	addi x0, x0, 0
c.ebreak	Environment BREAK	CR	10	1001	ebreak

APPENDIX C. OTHER RV32I EXTENSIONS

RV32M Multiply Extension

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)
mul	MUL	R	0110011	0x0	0x01	rd = (rs1 * rs2)[31:0]
mulh	MUL High	R	0110011	0x1	0x01	rd = (rs1 * rs2)[63:32]
mulhsu	MUL High (S) (U)	R	0110011	0x2	0x01	rd = (rs1 * rs2)[63:32]
mulu	MUL High (U)	R	0110011	0x3	0x01	rd = (rs1 * rs2)[63:32]
div	DIV	R	0110011	0x4	0x01	rd = rs1 / rs2
divu	DIV (U)	R	0110011	0x5	0x01	rd = rs1 / rs2
rem	Remainder	R	0110011	0x6	0x01	rd = rs1 % rs2
remu	Remainder (U)	R	0110011	0x7	0x01	rd = rs1 % rs2

RV32A Atomic Extension

31		27	26	25	24	20 19		15 14		12 11		7 6		0	
funct5		aq	rl	rs2		rs1		funct3		rd		opcode			
5		1	1	5		5		3		5		7			
Inst	Name	FMT	Opcode	funct3	funct5	Description (C)									
lr.w	Load Reserved	R	0101111	0x2	0x02	rd = M[rs1], reserve M[rs1]									
sc.w	Store Conditional	R	0101111	0x2	0x03	if (reserved) { M[rs1] = rs2; rd = 0 } else { rd = 1 }									
amoswap.w	Atomic Swap	R	0101111	0x2	0x01	rd = M[rs1]; swap(rd, rs2); M[rs1] = rd									
amoadd.w	Atomic ADD	R	0101111	0x2	0x00	rd = M[rs1] + rs2; M[rs1] = rd									
amoand.w	Atomic AND	R	0101111	0x2	0x0C	rd = M[rs1] & rs2; M[rs1] = rd									
amoor.w	Atomic OR	R	0101111	0x2	0x0A	rd = M[rs1] rs2; M[rs1] = rd									
amoxor.w	Atomic XOR	R	0101111	0x2	0x04	rd = M[rs1] ^ rs2; M[rs1] = rd									
amomax.w	Atomic MAX	R	0101111	0x2	0x14	rd = max(M[rs1], rs2); M[rs1] = rd									
amomin.w	Atomic MIN	R	0101111	0x2	0x10	rd = min(M[rs1], rs2); M[rs1] = rd									

RV32F / D Floating-Point Extensions

Inst	Name	FMT	Opcode	funct3	funct5	Description (C)
flw	Flt Load Word	*				rd = M[rs1 + imm]
fsw	Flt Store Word	*				M[rs1 + imm] = rs2
fmadd.s	Flt Fused Mul-Add	*				rd = rs1 * rs2 + rs3
fmsub.s	Flt Fused Mul-Sub	*				rd = rs1 * rs2 - rs3
fnmadd.s	Flt Neg Fused Mul-Add	*				rd = -rs1 * rs2 + rs3
fnmsub.s	Flt Neg Fused Mul-Sub	*				rd = -rs1 * rs2 - rs3
fadd.s	Flt Add	*				rd = rs1 + rs2
fsub.s	Flt Sub	*				rd = rs1 - rs2
fmul.s	Flt Mul	*				rd = rs1 * rs2
fdiv.s	Flt Div	*				rd = rs1 / rs2
fsqrt.s	Flt Square Root	*				rd = sqrt(rs1)
fsgnj.s	Flt Sign Injection	*				rd = abs(rs1) * sgn(rs2)
fsgnjn.s	Flt Sign Neg Injection	*				rd = abs(rs1) * -sgn(rs2)
fsgnjx.s	Flt Sign Xor Injection	*				rd = rs1 * sgn(rs2)
fmin.s	Flt Minimum	*				rd = min(rs1, rs2)
fmax.s	Flt Maximum	*				rd = max(rs1, rs2)
fcvt.s.w	Flt Conv from Sign Int	*				rd = (float) rs1
fcvt.s.wu	Flt Conv from Uns Int	*				rd = (float) rs1
fcvt.w.s	Flt Convert to Int	*				rd = (int32_t) rs1
fcvt.wu.s	Flt Convert to Int	*				rd = (uint32_t) rs1
fmv.x.w	Move Float to Int	*				rd = ((int*) &rs1)
fmv.w.x	Move Int to Float	*				rd = *((float*) &rs1)
feq.s	Float Equality	*				rd = (rs1 == rs2) ? 1 : 0
flt.s	Float Less Than	*				rd = (rs1 < rs2) ? 1 : 0
fle.s	Float Less / Equal	*				rd = (rs1 <= rs2) ? 1 : 0
fclass.s	Float Classify	*				rd = 0..9

APPENDIX D. RV32I PSEUDO INSTRUCTIONS

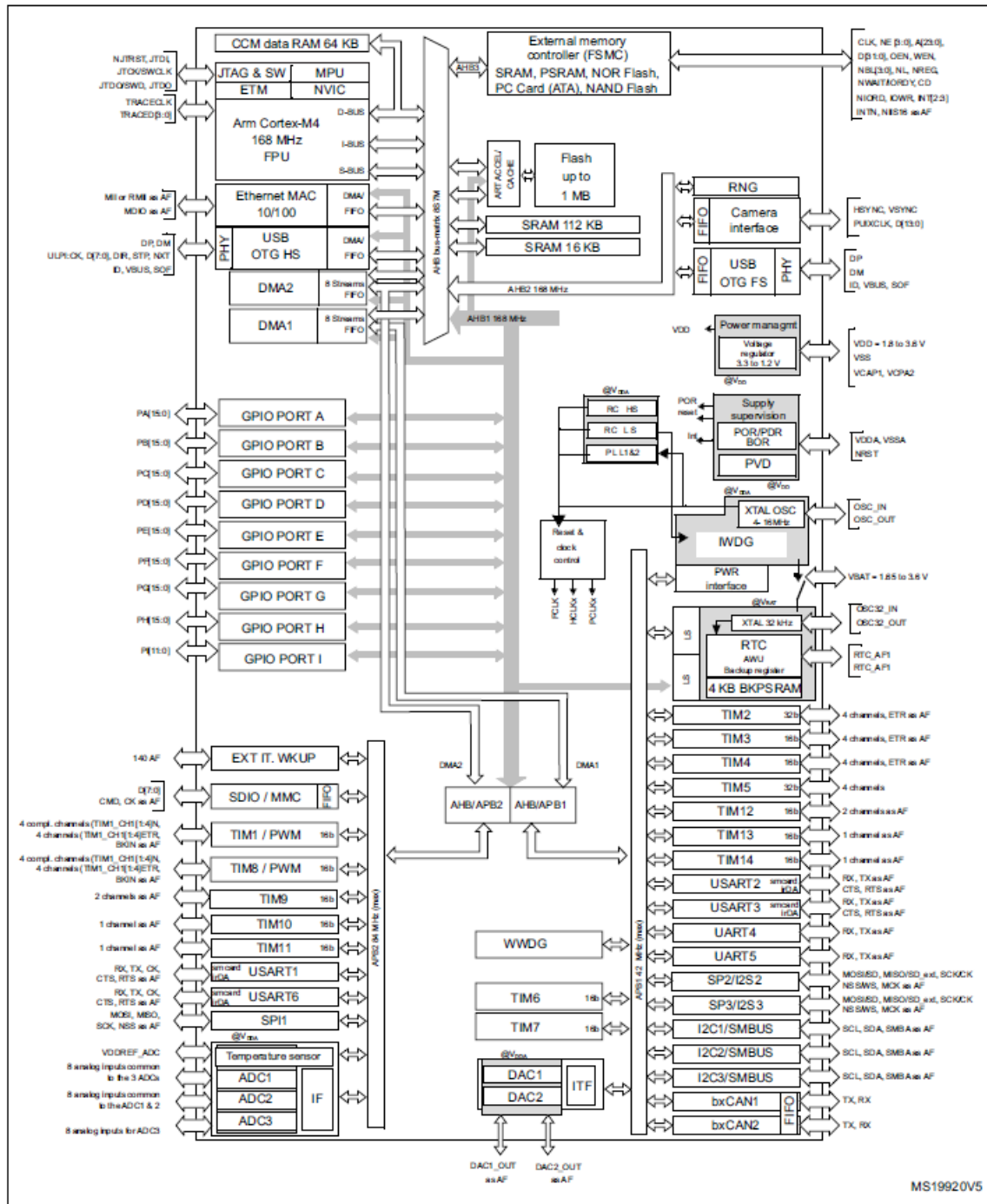
Pseudo Instructions

Pseudoinstruction	Base Instruction(s)	Meaning
la rd, symbol	auipc rd, symbol[31:12] addi rd, rd, symbol[11:0]	Load address
l{b h w d} rd, symbol	auipc rd, symbol[31:12] l{b h w d} rd, symbol[11:0](rd)	Load global
s{b h w d} rd, symbol, rt	auipc rt, symbol[31:12] s{b h w d} rd, symbol[11:0](rt)	Store global
fl{w d} rd, symbol, rt	auipc rt, symbol[31:12] fl{w d} rd, symbol[11:0](rt)	Floating-point load global
fs{w d} rd, symbol, rt	auipc rt, symbol[31:12] fs{w d} rd, symbol[11:0](rt)	Floating-point store global
nop	addi x0, x0, 0	No operation
li rd, immediate	<i>Myriad sequences</i>	Load immediate
mv rd, rs	addi rd, rs, 0	Copy register
not rd, rs	xori rd, rs, -1	One's complement
neg rd, rs	sub rd, x0, rs	Two's complement
negw rd, rs	subw rd, x0, rs	Two's complement word
sext.w rd, rs	addiw rd, rs, 0	Sign extend word
seqz rd, rs	sltiu rd, rs, 1	Set if = zero
snez rd, rs	sltu rd, x0, rs	Set if $\neq$ zero
sltz rd, rs	slt rd, rs, x0	Set if < zero
sgtz rd, rs	slt rd, x0, rs	Set if > zero
fmv.s rd, rs	fsgnj.s rd, rs, rs	Copy single-precision register
fabs.s rd, rs	fsgnjx.s rd, rs, rs	Single-precision absolute value
fneg.s rd, rs	fsgnjn.s rd, rs, rs	Single-precision negate
fmv.d rd, rs	fsgnj.d rd, rs, rs	Copy double-precision register
fabs.d rd, rs	fsgnjx.d rd, rs, rs	Double-precision absolute value
fneg.d rd, rs	fsgnjd.d rd, rs, rs	Double-precision negate
beqz rs, offset	beq rs, x0, offset	Branch if = zero
bnez rs, offset	bne rs, x0, offset	Branch if $\neq$ zero
blez rs, offset	bge x0, rs, offset	Branch if $\leq$ zero
bgez rs, offset	bge rs, x0, offset	Branch if $\geq$ zero
bltz rs, offset	blt rs, x0, offset	Branch if < zero
bgtz rs, offset	blt x0, rs, offset	Branch if > zero
bgt rs, rt, offset	blt rt, rs, offset	Branch if >
ble rs, rt, offset	bge rt, rs, offset	Branch if $\leq$
bgtu rs, rt, offset	bltu rt, rs, offset	Branch if >, unsigned
bleu rs, rt, offset	bgeu rt, rs, offset	Branch if $\leq$, unsigned
j offset	jal x0, offset	Jump
jal offset	jal x1, offset	Jump and link
jr rs	jalr x0, rs, 0	Jump register
jalr rs	jalr x1, rs, 0	Jump and link register
ret	jalr x0, x1, 0	Return from subroutine
call offset	auipc x1, offset[31:12] jalr x1, x1, offset[11:0]	Call far-away subroutine
tail offset	auipc x6, offset[31:12] jalr x0, x6, offset[11:0]	Tail call far-away subroutine
fence	fence iorw, iorw	Fence on all memory and I/O

APPENDIX E. RV32I AND F REGISTER FILES

Register	ABI Name	Description	Saver
x0	zero	Zero constant	—
x1	ra	Return address	Callee
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5-x7	t0-t2	Temporaries	Caller
x8	s0 / fp	Saved / frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Fn args/return values	Caller
x12-x17	a2-a7	Fn args	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller
f0-7	ft0-7	FP temporaries	Caller
f8-9	fs0-1	FP saved registers	Callee
f10-11	fa0-1	FP args/return values	Caller
f12-17	fa2-7	FP args	Caller
f18-27	fs2-11	FP saved registers	Callee
f28-31	ft8-11	FP temporaries	Caller

APPENDIX F. STM32F40XXX SOC DIAGRAM



1. The camera interface and ethernet are available only on STM32F407xx devices.

APPENDIX G. PICORV32 PARAMETERIZATION

These Verilog module parameters can be used to configure the PicoRV32 core.

ENABLE_COUNTERS (default = 1)

This parameter enables support for the RDCYCLE[H], RDTIME[H], and RDINSTRET[H] instructions. This instructions will cause a hardware trap (like any other unsupported instruction) if ENABLE_COUNTERS is set to zero.

Note: Strictly speaking the RDCYCLE[H], RDTIME[H], and RDINSTRET[H] instructions are not optional for an RV32I core. But chances are they are not going to be missed after the application code has been debugged and profiled. This instructions are optional for an RV32E core.

ENABLE_COUNTERS64 (default = 1)

This parameter enables support for the RDCYCLEH, RDTIMEH, and RDINSTRETH instructions. If this parameter is set to 0, and ENABLE_COUNTERS is set to 1, then only the RDCYCLE, RDTIME, and RDINSTRET instructions are available.

ENABLE_REGS_16_31 (default = 1)

This parameter enables support for registers the x16..x31. The RV32E ISA excludes these registers. However, the RV32E ISA spec requires a hardware trap for when code tries to access these registers. This is not implemented in PicoRV32.

ENABLE_REGS_DUALPORT (default = 1)

The register file can be implemented with two or one read ports. A dual ported register file improves performance a bit, but can also increase the size of the core.

LATCHED_MEM_RDATA (default = 0)

Set this to 1 if the mem_rdata is kept stable by the external circuit after a transaction. In the default configuration, the PicoRV32 core only expects the mem_rdata input to be valid in the cycle with mem_valid && mem_ready and latches the value internally.

This parameter is only available for the picorv32 core. In the picorv32_axi and picorv32_wb core this is implicitly set to 0.

TWO_STAGE_SHIFT (default = 1)

By default shift operations are performed in two stages: first shifts in units of 4 bits and then shifts in units of 1 bit. This speeds up shift operations, but adds additional hardware. Set this parameter to 0 to disable the two-stage shift to further reduce the size of the core.

BARREL_SHIFTER (default = 0)

By default, shift operations are performed by successively shifting by a small amount (see TWO_STAGE_SHIFT above). With this option set, a barrel shifter is used instead.

TWO_CYCLE_COMPARE (default = 0)

This relaxes the longest data path a bit by adding an additional FF stage at the cost of adding an additional clock cycle delay to the conditional branch instructions.

Note: Enabling this parameter will be most effective when retiming (aka "register balancing") is enabled in the synthesis flow.

TWO_CYCLE_ALU (default = 0)

This adds an additional FF stage to the ALU data path, improving timing at the cost of an additional clock cycle for all instructions that use the ALU.

Note: Enabling this parameter will be most effective when retiming (aka "register balancing") is enabled in the synthesis flow.

COMPRESSED_ISA (default = 0)

This enables support for the RISC-V Compressed Instruction Set.

CATCH_MISALIGN (default = 1)

Set this to 0 to disable the circuitry for catching misaligned memory accesses.

CATCH_ILLINSN (default = 1)

Set this to 0 to disable the circuitry for catching illegal instructions.

The core will still trap on EBREAK instructions with this option set to 0. With IRQs enabled, an EBREAK normally triggers an IRQ 1. With this option set to 0, an EBREAK will trap the processor without triggering an interrupt.

ENABLE_PCPI (default = 0)

Set this to 1 to enable the external Pico Co-Processor Interface (PCPI). The external interface is not required for the internal PCPI cores, such as `picorv32_pcp_i_mul`.

ENABLE_MUL (default = 0)

This parameter internally enables PCPI and instantiates the `picorv32_pcp_i_mul` core that implements the `MUL[H[SU—U]]` instructions. The external PCPI interface only becomes functional when `ENABLE_PCPI` is set as well.

ENABLE_FAST_MUL (default = 0)

This parameter internally enables PCPI and instantiates the `picorv32_pcp_i_fast_mul` core that implements the `MUL[H[SU—U]]` instructions. The external PCPI interface only becomes functional when `ENABLE_PCPI` is set as well.

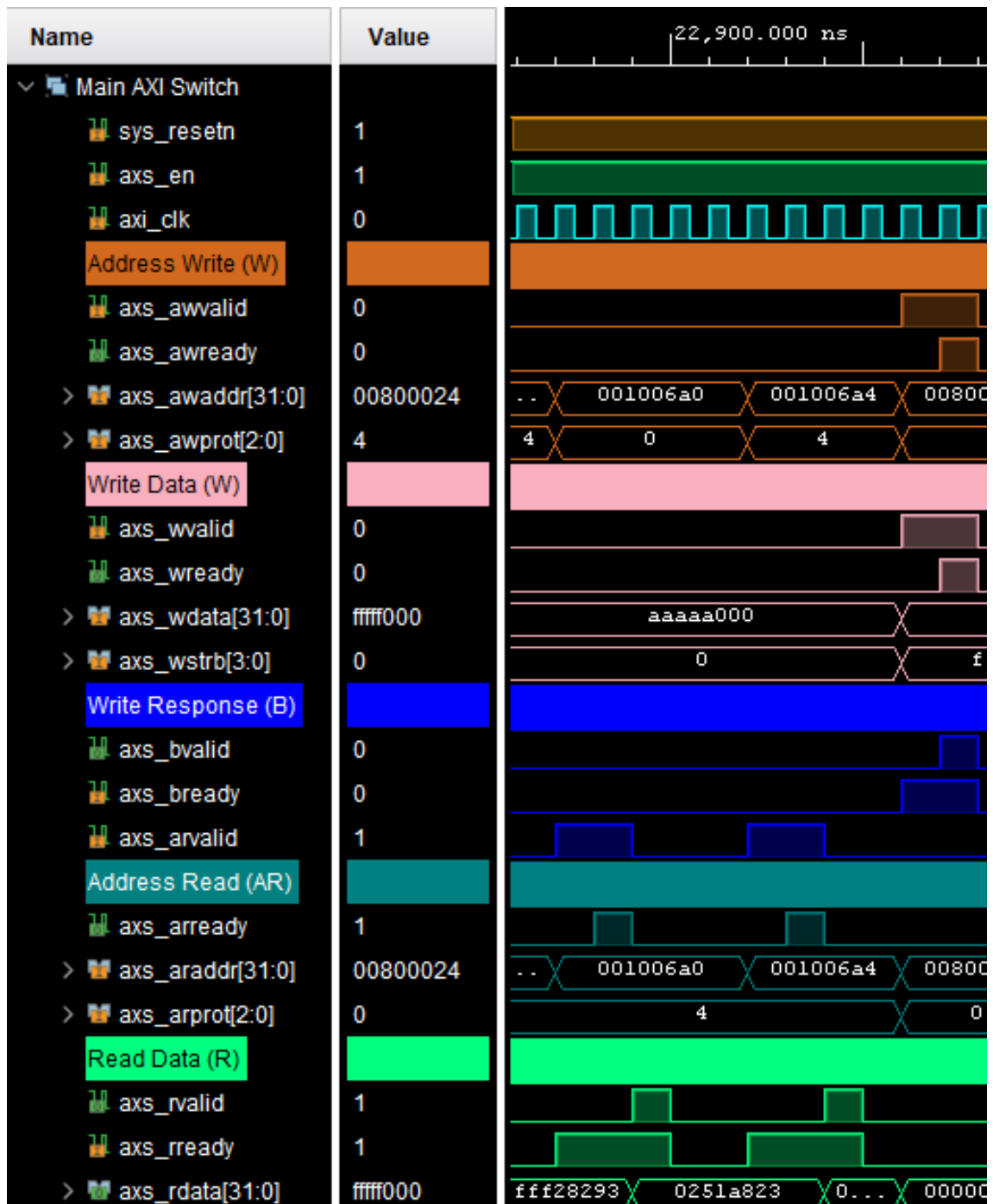
If both `ENABLE_MUL` and `ENABLE_FAST_MUL` are set then the `ENABLE_MUL` setting will be ignored and the fast multiplier core will be instantiated.

`ENABLE_DIV (default = 0)`

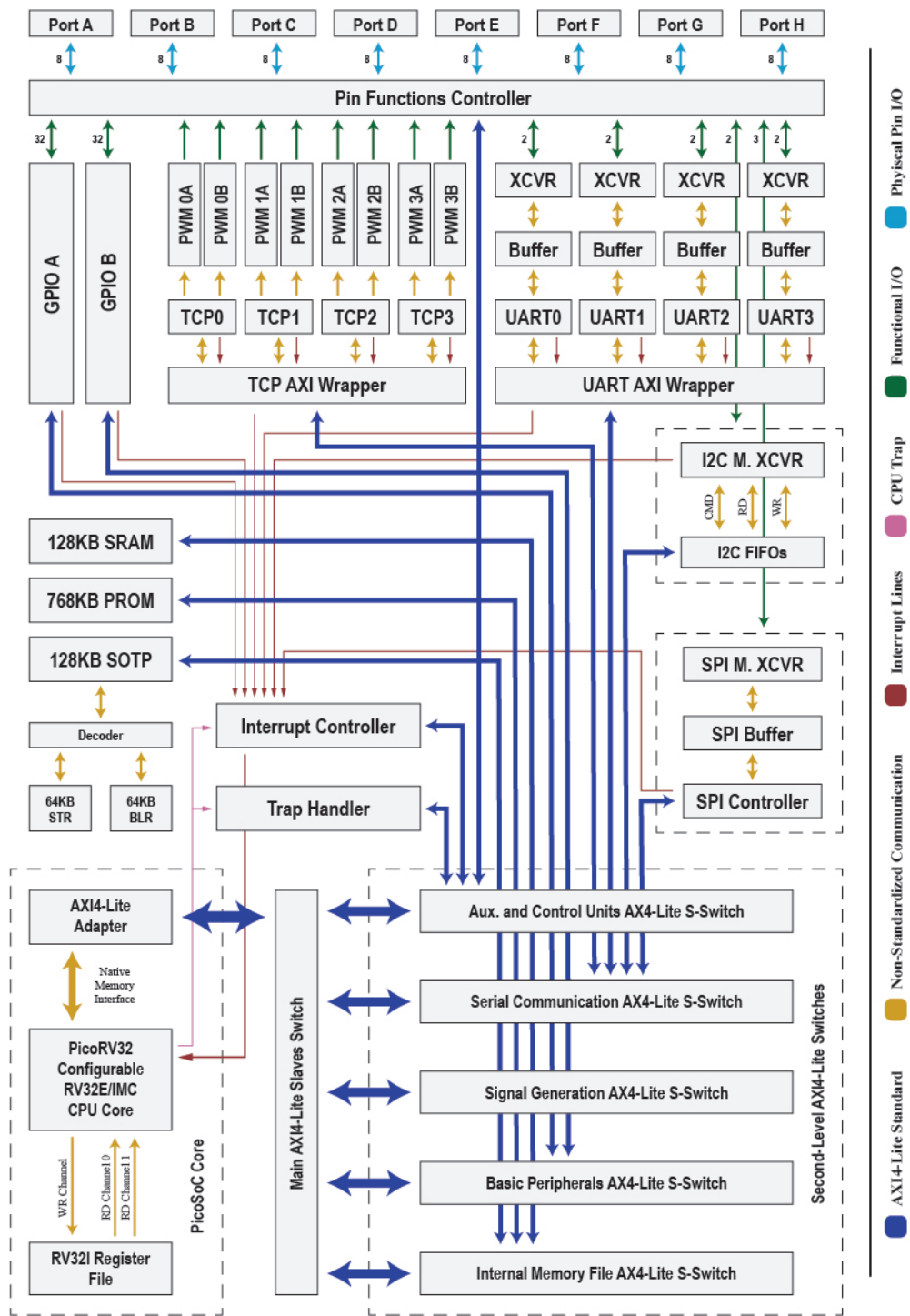
This parameter internally enables PCPI and instantiates the `picorv32_pcp-div` core that implements the `DIV[U]`/`REM[U]` instructions. The external PCPI interface only becomes functional when `ENABLE_PCPI` is set as well.



APPENDIX H. AXI4-LITE TRANSACTIONS TIMING



APPENDIX I. OZU SOC DETAILED DIAGRAM



APPENDIX J. OZU SOC PLANNED ADDRESS SPACE

Components Group		<i>Internal Memory (8 MB Max.)</i>		
Component	Implemented	Start Addr. [h]	End Addr. [h]	Max Size
CCM SRAM	No	0x00000000	0x0001FFFF	128 KB
Main SRAM	Yes	0x00020000	0x0005FFFF	256 KB
Reserved	N/A	0x00060000	0x000FFFFFFF	640 KB
SOTP ROM (STR)	Yes	0x00100000	0x0011FFFF	128 KB
SOTP ROM (BLR)	Yes	0x00120000	0x0013FFFF	128 KB
Main PROM	Yes	0x00140000	0x001FFFFFFF	768 KB
Reserved	N/A	0x00200000	0x007FFFFFFF	6144 KB
Components Group		<i>Basic I/O Peripherals</i>		
Component	Implemented	Start Addr. [h]	End Addr. [h]	Max Size
GPIO A	Yes	0x00800000	0x008003FF	N/A
GPIO B	Yes	0x00800400	0x008007FF	N/A
Reserved	N/A	0x00800800	0x0080FFFF	N/A
ADC	No	0x00810000	0x008103FF	N/A
Reserved	N/A	0x00810400	0x008FFFFFFF	N/A
Components Group		<i>Signal Generation / Processing</i>		
Component	Implemented	Start Addr. [h]	End Addr. [h]	Max Size
TCP0	Yes	0x00900000	0x009000FF	N/A
TCP1	Yes	0x00900100	0x009001FF	N/A
TCP2	Yes	0x00900200	0x009002FF	N/A
TCP3	Yes	0x00900300	0x009003FF	N/A
Reserved	N/A	0x00900400	0x009FFFFFFF	N/A

Components Group		Serial Communication Interfaces		
Component	Implemented	Start Addr. [h]	End Addr. [h]	Max Size
UART0	Yes	0x00A00000	0x00A000FF	N/A
UART1	Yes	0x00A00100	0x00A001FF	N/A
UART2	Yes	0x00A00200	0x00A002FF	N/A
UART3	Yes	0x00A00300	0x00A003FF	N/A
Reserved	N/A	0x00A00400	0x00AFFFFFFF	N/A
I2C	Yes	0x00B00000	0x00B003FF	N/A
Reserved	N/A	0x00B00400	0x00BFFFFFFF	N/A
SPI	Yes	0x00C00000	0x00C003FF	N/A
Reserved	N/A	0x00C00400	0x00CFFFFFFF	N/A
Components Group		Extensions and Debug Interfaces		
Component	Implemented	Start Addr. [h]	End Addr. [h]	Max Size
Reserved	N/A	0x00D00000	0x00DFFFFFFF	N/A
Components Group		Control and Auxiliary Units		
CPU Controller	Yes	0x01000000	0x010003FF	N/A
Reserved	N/A	0x01000400	0x011FFFFFFF	N/A
Components Group		Native External Memory (Up to 16 MB)		
Direct Addressing	N/A	0x01200000	0x021FFFFFFF	N/A
Components Group		None-Planned Address Space		
Reserved	N/A	0x02200000	0xFFFFFFFF	N/A

APPENDIX K. PIN FUNCTION CONTROLLER

The OzU SoC PFCN module, designed from the ground up, implements a sophisticated and specific mechanism to determine the function of each pin in the physical IC package, namely each pin in each pin port from A to H, totaling 64 pins. The PFCN module has one 32-bit function control register (PINF) corresponding to each of the 8-pin ports (PORT). This allocates 4 control bits to each pin in each port, to determine its function as one of up to 16 different functions, for example, `PINF_A[3:0]` controls the function of A0 pin, `PINF_A[7:4]` controls the function of A1 pin, etc.

To ensure meeting a reasonable PPA balance, not all ports can be mapped to any functionality found in the SoC, as this requires more than 4 control bits per each pin, dictating more numbers of registers; therefore, increasing complexity and area. This resulted in some functions being able to be mapped to some specific ports only, while others can be mapped to any port. While this approach might seem limiting, it actually fulfills most of the practical usage scenarios of the end-product, as it is inspired from MCUs found in the commercial market. The table in the end of appendix shows the exact details of these assignments. The following notes should be taken into consideration for the table to make functional sense to the reader:

1. Fault-proof mechanisms for mapping input and bidirectional lines, for example, ADC, RXL, and SDA, should be implemented to prevent multiple input hazards. These mechanisms can be implemented on the hardware level, such as priority mapping in the PFCN itself or using an auxiliary settings validation module, or the mechanisms could also be implemented on the software level, ensuring the compiler would detect and prevent such multiple input hazards.

2. On the hardware finalization level, the GPIO ports should have independent bidirectional and tri-state hardware modules to handle proper data direction flow.
3. The OzU SoC has 8 PWM outputs in total, starting from 0A and ending with 3B (refer to Appendix I). The PWM channels are mapped in order with respect to the port pins order. For example, PWM-0A maps exclusively to pin 0 on any port, 1A to pin 2, and 3B to pin 7. The same can be said for ADC inputs, e.g., ADC [5] maps to PORT_X [5], though considering note number 1.
4. SPI-DIOX pins vary depending on the configured mode, i.e., SPI, DSPI, or QSPI. Careful considerations should be taken in the PFCN to detect and prevent any misconfigurations that result in multiple input faults.

<i>Package Pins</i>	Pins [15:0]	Pins [31:16]	Pins [47:32]	Pins [63:48]
<i>Affected Ports</i>	Ports B — A	Ports D — C	Ports F — E	Ports H — G
<i>4-bit Funct.</i>	Mapped Function(s) to Affected Above			
0x0	GPIO A [15:0]	GPIO A [31:16]	GPIO B [15:0]	GPIO B [31:16]
0x1	ADC Input [7:0] from any First-Assigned Pin in any Port			
0x2	PWM Output [7:0] on any First-Assigned Pin in any Port			
0x3	UART0-TXL	UART2-TXL	UART0-TXL	UART2-TXL
0x4	UART0-RXL	UART2-RXL	UART0-RXL	UART2-RXL
0x5	UART1-TXL	UART3-TXL	UART1-TXL	UART3-TXL
0x6	UART1-RXL	UART3-RXL	UART1-RXL	UART3-RXL
0x7	I2C-SCL (Serial Clock) Output on any Pin			
0x8	I2C-SDA (Serial Data) Line on any Pin			
0x9	Reserved	SPI-SCK	Reserved	SPI-SCK
0xA	Reserved	SPI-DIO0	Reserved	SPI-DIO0
0xB	Reserved	SPI-DIO1	Reserved	SPI-DIO1
0xC	Reserved	SPI-DIO2	Reserved	SPI-DIO2
0xD	Reserved	SPI-DIO3	Reserved	SPI-DIO3
0xE	CPU Trap Output on any Pin			
0xF	Reserved			

APPENDIX L. NOVLI-ISA BLUEPRINT

8-bit Micro Instructions (Fixed-size 8-bit)					
7:4		3:2		1:0	Bits
parameter		function		opcode	Function
Subset Opcode		2'b00 (0x0)			
Instruction	Assembly	Function	Operation		Operand
NOP/WAIT	nop swait [delay]	2'b00	No operation for one cycle Enable built-in T/C passing the delay parameter		Zero ('b0000) Non-zero Delay Parameter*
RGRST	rgrst [rt]	2'b01	Reset rt to Zero		Target Register Address rt ∈ {x0 .. x15} (RV32E Registers)
RGINC	rginc [rt]	2'b10	Increment rt by 1		
RGDEC	rgdec [rt]	2'b11	Decrement rt by 1		

* The delay resulting from the given parameter is subject to factors like CPU clock, and pre-configured scalar for the T/C unit

Unary Register Operations Instructions (Fixed-size 16-bit)					
15:12	11:10	9:5	4	3:0	Bits
function	sub function [1:0]	rt	sub function [2]	opcode	Function
Subset Opcode		4'b0001 (0x1)			
Instruction	Assembly	Function	Operation	Sub Function	Operation
URGRST	<i>urgrst [rt, reset_val]</i>	4'h0	Unary Register Reset to Value	3'h0 3'h1 3'h2 3'h3 3'h4 to 4'h7	Reset to Zero Reset to 0x1 Reset to 0xffff Reset to 0xffff_ffff Reserved for Extension
URGINC	<i>urginc [rt, inc_val]</i>	4'h1	Unary Register Increment	Increment Value = sub function + 1	
URGDEC	<i>urgdec [rt, dec_val]</i>	4'h2	Unary Register Decrement	Decrement Value = sub function + 1	
URGCMP	<i>urgcmp [rt, cmp_type]</i>	4'h3	Unary Register Increment	3'h0 3'h1	1's Complement (NOT) 2's Complement (Negative)
Reserved		4'h5 4'h6	Possibly to Load a Register with Scientific Constants or Frequently Used Literals		
URGSLL	<i>urgsll [rt, shift_val]</i>	4'h7 4'h8 4'h9	Unary Register Left Shift	Shift Value = sub function + 1 Shift Value = sub function + 5 Shift Value = sub function + 9	
URGSRL	<i>urgsrl [rt, shift_val]</i>	4'h7 4'h8 4'h9	Unary Register Logical Right Shift	Shift Value = sub function + 1 Shift Value = sub function + 5 Shift Value = sub function + 9	
URGSRA	<i>urgsra [rt, shift_val]</i>	4'h7 4'h8 4'h9	Unary Register Arithmetic Right Shift	Shift Value = sub function + 1 Shift Value = sub function + 5 Shift Value = sub function + 9	

Register-Register Operations Instructions (Fixed-size 24-bit)						
23:20	19:15	14:10	9:5	4	3:0	Bits
<i>function</i>	<i>rs2</i>	<i>rs1</i>	<i>rd</i>	<i>N/A</i>	<i>opcode</i>	<i>Function</i>
Subset Opcode			4'b1001 (0x9)			
<i>Instruction</i>	<i>Assembly</i>	<i>Function</i>		<i>Operation</i>		
ADD SUB XOR OR AND SLL SRL SRA SLT SLTU	Same assembly formats as in the standard RV32I/E ISA	4'h0 4'h1 4'h2 4'h3 4'h4 4'h5 4'h6 4'h7 4'h8 4'h9		Same functionalities as in the standard RV32I/E ISA		
Reserved		4'ha to 4'hf		Reserved for possible extension of operations		

Register-Immediate Operations Instructions (Variable-size 24-bit or 32-bit)						
24-bit Format						
23:20	19:15	14:10	9:5	4	3:0	Bits
<i>function</i>	<i>imm (short)</i>	<i>rs1</i>	<i>rd</i>	<i>24/32-bit</i>	<i>opcode</i>	<i>Function</i>
32-bit Format						
31:28	27:15	14:10	9:5	4	3:0	Bits
<i>function</i>	<i>imm (long)</i>	<i>rs1</i>	<i>rd</i>	<i>24/32-bit</i>	<i>opcode</i>	<i>Function</i>
Subset Opcode			4'b1101 (0xD)			
<i>Instruction</i>	<i>Assembly</i>	<i>Function</i>		<i>Operation</i>		
ADDI SUBI XORI ORI ANDI SLLI SRLI SRAI SLTI SLTIU	Same assembly formats as in the standard RV32I/E ISA, except the introduction of subi which is a new instruction: <i>subi [rd, rs1, imm]</i>	4'h0 4'h1 4'h2 4'h3 4'h4 4'h5 4'h6 4'h7 4'h8 4'h9		Same functionalities as in the standard RV32I/E ISA, except the introduction of subi which is a new instruction: SUBI $\rightarrow$ $rd = rs1 - imm$		
Reserved		4'ha to 4'hf		Reserved for possible extension of operations		

Load Immediate / Upper Immediate Instruction (Variable-size 16-bit or 32-bit)					
16-bit Format					
15	14:10	9:5	4	3:0	Bits
<i>function</i> $I = 0 \mid UI = 1$	<i>immediate (short)</i>	<i>rd</i>	<i>16/32-bit</i>	<i>opcode</i> <i>4'b0010 (0x2)</i>	<i>Function</i>
32-bit Format					
31	30:10	9:5	4	3:0	Bits
<i>function</i> $I = 0 \mid UI = 1$	<i>immediate (long)</i>	<i>rd</i>	<i>16/32-bit</i>	<i>opcode</i> <i>4'b0010 (0x2)</i>	<i>Function</i>

Memory Operations Instructions (Variable-size 24-bit or 32-bit)						
24-bit Format						
23:20	19:15	14:10	9:5	4	3:0	Bits
<i>function</i>	<i>offset (short)</i>	<i>rba *</i>	<i>rs2 / rd</i>	<i>24/32-bit</i>	<i>opcode</i>	<i>Function</i>
32-bit Format						
31:29	28:15	14:10	9:5	4	3:0	Bits
<i>function</i>	<i>offset (long)</i>	<i>rba *</i>	<i>rs2 / rd</i>	<i>24/32-bit</i>	<i>opcode</i>	<i>Function</i>
Subset Opcode			4'b0101 (0x5)			
<i>Instruction</i>	<i>Assembly</i>	<i>Function</i>	<i>Operation</i>			
LB LH LW LBU LHU SB SH SW	Same assembly formats as in the standard RV32I/E ISA * <i>rba</i> is the "Base Address Register" <i>rba == rs1</i> in standard RV32I/E ISA	4'h0 4'h1 4'h2 4'h3 4'h4 4'h5 4'h6 4'h7	Same functionalities as in the standard RV32I/E ISA as in the standard RV32I/E ISA * <i>rba</i> is the "Base Address Register" <i>rba == rs1</i> in standard RV32I/E ISA			

Branching Operations Instructions (Variable-size 24-bit or 32-bit)						
24-bit Format						
23:20	19:15	14:10	9:5	4	3:0	Bits
<i>function</i>	<i>offset (short)</i>	<i>rs2</i>	<i>rs1</i>	<i>24/32-bit</i>	<i>opcode</i>	<i>Function</i>
32-bit Format						
31:29	28:15	14:10	9:5	4	3:0	Bits
<i>function</i>	<i>offset (long)</i>	<i>rs2</i>	<i>rs1</i>	<i>24/32-bit</i>	<i>opcode</i>	<i>Function</i>
Subset Opcode			4'b1110 (0xE)			
<i>Instruction</i>	<i>Assembly</i>	<i>Function</i>	<i>Operation</i>			
BEQ BNE BLT BGE BLTU BGEU	Same assembly formats as in the standard RV32I/E ISA * <i>rba</i> is the "Base Address Register" <i>rba == rs1</i> in standard RV32I/E ISA	4'h0 4'h1 4'h2 4'h3 4'h4 4'h5	Same functionalities as in the standard RV32I/E ISA as in the standard RV32I/E ISA * <i>rba</i> is the "Base Address Register" <i>rba == rs1</i> in standard RV32I/E ISA			

Jump / Jump And Link (Variable-size 16-bit or 32-bit)					
16-bit Format					
15	14:10	9:5	4	3:0	Bits
<i>function</i> $J = 0 \quad \quad JAL = 1$	<i>offset (short)</i>	<i>rl or offset [9:5]</i>	<i>16/32-bit</i>	<i>opcode</i> $4'b0110 \text{ (0x6)}$	<i>Function</i>
32-bit Format					
31	30:10	9:5	4	3:0	Bits
<i>function</i> $J = 0 \quad \quad JAL = 1$	<i>offset (long)</i>	<i>rl or offset [25:21]</i>	<i>16/32-bit</i>	<i>opcode</i> $4'b0110 \text{ (0x6)}$	<i>Function</i>

Jump Register / Jump And Link Register (Variable-size 24-bit or 32-bit)						
24-bit Format						
23	22:15	14:10	9:5	4	3:0	Bits
<i>function</i> $JR = 0 \quad \quad JALR = 1$	<i>offset (short)</i>	<i>rba</i>	<i>rl or offset [12:8]</i>	<i>24/32-bit</i>	<i>opcode</i> $4'b1010 \text{ (0x6)}$	<i>Function</i>
32-bit Format						
31	30:15	14:10	9:5	4	3:0	Bits
<i>function</i> $JR = 0 \quad \quad JALR = 1$	<i>offset (long)</i>	<i>rba</i>	<i>rl or offset [20:16]</i>	<i>24/32-bit</i>	<i>opcode</i> $4'b1010 \text{ (0x6)}$	<i>Function</i>