8-2-2023

seL4 on RISC-V - Developing High Assurance Platforms with Modular Open-Source Architectures

Michael A. Doran Jr
Grand Valley State University

seL4 on RISC-V

Developing High Assurance Platforms with Modular Open-Source Architectures

Michael A. Doran

A Thesis Submitted to the Graduate Faculty of

GRAND VALLEY STATE UNIVERSITY

In

Partial Fulfillment of the Requirements for the Degree of

Master of Science in Engineering Electrical & Computer Engineering

School of Engineering

August 2023

seL4 on RISC-V – Developing High Assurance Platforms With Modular Open-Source Architectures

A Thesis Presented by

MICHAEL A. DORAN

Approved as to style and content by:

[illegible] 07/20/2023

Dr. Nabeeh Kandalaft (Committee Chair) Date

[illegible] 07-20-23

Dr. Chirag Parikh (Committee Member) Date

[illegible] 07-20-23

Dr. Byron Devries (Committee Member) Date

[illegible]

Dean of the College

July 20, 2023

Date

[illegible]

Dean of The Graduate School

8/2/2023

Date

DEDICATION

This work is dedicated to my loving wife whose unwavering support allowed me the time and motivation needed to complete this body of work.

ACKNOWLEDGMENTS

I would like to sincerely acknowledge Dr. Nabeeh Kandalaft. Dr. Kandalaft's patience and guidance throughout this process has been invaluable to my development as a student and future academic. I would also like to acknowledge, DornerWorks, without the opportunity presented by DornerWorks I never would have pursued research in RISC-V. The entire DornerWork's team that worked on this effort alongside of me made this body of work possible. Finally, I would like to acknowledge the work of Bruno Sa who initially completed the port of the Rocketchip to the ZCU104 as that served as the basis of my research for porting seL4/RISC-V to the ZCU102.

ABSTRACT

Virtualization is now becoming an industry standard for modern embedded systems. Modern embedded systems can now support multiple applications on a single hardware platform while meeting power and cost requirements. Virtualization on an embedded system is achieved through the design of the hardware-software interface. Instruction set architecture, ISA, defines the hardware-software interface for an embedded system. At the hardware level the ISA, provides extensions to support virtualization.

In addition to an ISA that supports hypervisor extensions it is equally important to provide a hypervisor completely capable of exploiting the benefits of virtualization for securing modern embedded systems. Currently there does not exist a commercial hardware design that leverages the RISC-V ISA hypervisor extension co-designed with an open-source microkernel.

This research describes an implementation of the seL4 open-source microkernel with the latest version of the RISC-V hypervisor extension (H-extension v0.6.1) specification in a Rocket chip soft core. The combination of open ISA, open-source OS and open-source hardware enables hardware and software co-design for securing embedded applications.

The implication of this research provides a meaningful evaluation of RISC-V with the seL4 open-source microkernel by providing an open-source hardware implementation on a Zynq Ultrascale+ MPSoC ZCU102 to assist the RISC-V community towards implementation and evaluation of hypervisor technology such as seL4.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

Virtualization has enabled embedded systems to meet the demands of market pressure to minimize size, weight, power, and cost (SWaP-C). Powerful multicore architectures provide the ability to isolate virtual machines (VMs) that perform safety-critical functions concurrent to non-safety critical functions [1].

This thesis presents the first public implementation of a virtualized system targeting a quad-core RISC-V softcore utilizing the seL4 microkernel. The softcore was designed using open-source tools to generate a bitstream that executes on the FPGA fabric of a Xilinx ZCU102 ZynqMP Ultrascale+ development board.

The reference architecture evaluated in this thesis consists of executing a single guest virtual machine executing a feature rich operating system, Linux. Performance and benchmarking were collected to compare the Rocketchip softcore vs the ARM Cortex A-53 hardcore.

1.1 Problem Statement

Despite the logical CPU and memory isolation provided by existing hypervisor layers, there are several challenges and difficulties in proving strong isolation, due to the reciprocal interference caused by micro-architectural resources (e.g., last-level caches, interconnects, and memory controllers) shared among virtual machines (VM) [2, 3]. Currently Intel and ARM are working on implementations of microarchitectures to bolster strong isolation mechanism, however, these architectures are proprietary [4, 5] RISC-V is an open-source instruction set architecture (ISA). As part of the RISC-V privileged architecture specification, hardware virtualization support is specified through the hypervisor extension (H-extension) [6]. The H-extension specification used in this design is currently at version 0.6.1. To date, the H-extension has achieved function completeness with KVM and Xvisor in QEMU as well as one public implementation as a softcore on a ZCU104 development kit [5]. The goal of this thesis is to provide the first virtualized reference design utilizing the secure seL4 microkernel targeted for a RISC-V softcore running on a ZCU102.

1.2 Scope

This thesis presents a reference architecture implementing two microarchitectures of equal caliber: a quadcore RISC-V Rocketchip and a quadcore Cortex A-53 ARM processor. Each implementation will contain a single virtual machine running the feature rich operating system (OS) Linux. Benchmarking is performed on both implementations to gain a comparison of performance.

1.3 Layout of Thesis

The following section will provide background information in the form of a literature review for kernels, virtualization, seL4, formal methods, and RISC-V. This background information will provide the baseline information needed to conceptualize the reference architecture implemented. The third section will cover details of the implementation on both the RISC-V softcore and the ARM Cortex A-53; it also will outline the tools used to generate the bitstream for the RISC-V softcore. The fourth section will cover the tests performed to evaluate performance of the reference design for both architectures. Performance metrics are measured from each guest operating system during real time. The final section discusses future work, improvements, and a summary of results.

CHAPTER 2: LITERATURE REVIEW

2.1 RISC-V

RISC-V is an open-source instruction set architecture (ISA) that originated from the University of California Berkely [6, 7]. As indicated by the name, the ISA was developed on Reduced Instruction Set Computer principles. This ISA seeks to provide a base ISA and optional application-specific instructions to support software engineers with the ability to add additional instructions to meet project specific requirements [7]. The open-source nature of the ISA is what ultimately allows hardware and software engineers the flexibility to create custom instructions as compared to the closed-source counterparts ARM and x86. Over the last decade RISC-V has made significant strides in becoming a universal ISA that supports a variety of processor types from embedded SoCs to high performance server platforms [2, 3]. This survey provides an overview of the state-of-art for RISC-V and the development ecosystem.

2.1.1 Base ISA

The RISC-V ISA extends four primary integer variants that include: RV32I (for 32bit), RV32E (for 32-bit with 16 registers), RV64I (for 64-bit), and RV128I (for 128-bit) [9]. These base instructions provide the necessary requirements to bootstrap a processor and execute a minimal operating system. Data stored in memory utilizes the little-endian system and represents signed integer values with the two's complement.

Functionality with the RISC-V ISA is expanded by leveraging the extension modules [9]. For example, the M module provides multiplication and division instructions. Table 1 below provides an overview of each ISA extension module.

Table 1: RISC-V Extension Modules

RISC-V ISA Extension	Description
M	Enables multiplication and division
A	Enables atomic instructions
F	Enables floating point instructions
D	Enables double precision floating point instructions
G	Enables modules M, A, F, D.
Q	Quad precision floating-point instructions
L	Decimal floating-point instructions
C	Compressed instructions
B	Bit manipulation instructions
J	Dynamically translated languages support
T	Transactional memory support
P	Packed-single Instruction Multiple Data
V	Vector operation instructions
N	User-level interrupt support
H	Hypervisor support
S	Supervisor level instructions

The extensions modules can be enabled during compile time to extend the requisite functionality needed by the design. A system designer has the complete ability to leverage additional custom instructions to add to this list so long as the hardware architecture can support it.

2.1.2 Instruction Format

For this thesis, the rest of this overview will focus on the RV64I since that is what is applicable for this study. There are six formats for RV64I in the RISC-V architecture. They include [9]:

- R-Type (Register Type): This format is used for arithmetic and logical operations that involve two source registers and one destination register. The R-Type instruction format has the following fields:
    - Opcode (6 bits): Specifies the operation to be performed.
    - rd (5 bits): Destination Register
    - funct3 (3 bits): Function code that further specifies the operation.
    - rs1 (5 bits): Source register 1.
    - rs2 (5 bits): source register 2.
    - funct7 (7 bits): Additional function code.

- I-Type (Immediate type): This format is used for instructions that involve an immediate value and one source/destination register. The I-Type instruction format has the following fields:
    - Opcode (6 bits): Specifies the operation to be performed.
    - rd (5 bits): Destination Register
    - funct3 (3 bits): Function code that further specifies the operation.
    - rs1 (5 bits): Source Register 1.
    - imm [11:0]: Immediate value.

- S-Type (Store Type): This format is used for storing data from a register to memory. The S-Type instruction format has the following fields:
    - Opcode (6 bits): Specifies the operation to be performed.
    - imm [11:5] (7 bits): immediate value (offset) used for memory addressing
    - funct3 (3 bits): function code that further specifies the operation.
    - rs1 (5 bits): Source register.
    - rs2 (5 bits): Register containing the data to be stored.
    - imm [4:0] (5 bits): immediate value (offset) used for memory addressing.

- B-Type (Branch Type): this format is used for conditional branching. instructions. The B-Type instruction format has the following fields:
    - Opcode (6 bits): Specifies the operation to be performed.
    - imm [12|10:5] (7 bits): immediate value (offset) used for memory addressing
    - funct3 (3 bits): function code that further specifies the operation.
    - rs1 (5 bits): Source register.
    - rs2 (5 bits): Register containing the data to be stored.
    - imm [4:1|11] (5 bits): immediate value (offset) used for memory addressing.

- U-Type (Upper Immediate Type): This format is used for instructions with wider immediate values. The U-Type instruction format has the following fields:
    - Opcode (6 bits): Specifies the operation to be performed.
    - rd (5 bits): Destination register.
    - imm [31:12] (20 bits): Upper immediate value.

- J-Type (Jump Type): This format is used for unconditional branching instructions. The J-Type instruction format has the following fields:
    - Opcode (6 bits): Specifies the operation to be performed.

- rd (5 bits): Destination register.
- imm [20|10:1|19:12] (21 bits): Immediate value (offset) used for jumping.

The primary instruction format listed above can be extended by a hardware/software engineer by introducing additional formats and instructions. The next section focuses on the RISC-V privilege level concept.

2.1.3 Privilege Level

The RISC-V privilege level model allows the operating system the ability to change its execution privilege status. RISC-V supports three privilege levels of execution, any attempt of software to perform an operation outside of its designated privilege level is not permitted [6]. This allows the protection of software stacks from executing unauthorized instructions. If a piece of software were to attempt to execute an instruction outside of its privilege level, then a system exception would occur which would cause the underlying kernel or operating system to intervene.

The highest privilege level in RISC-V is the Machine-Mode (M-Mode). This level is inherently trusted as it executes closest to the hardware. This access provides an operating system with the ability to manage and configure all the hardware resources of a platform. The next privilege level includes the supervisor-Mode (S-Mode). S-Mode is intended for OS usage. Finally, User-Mode (U-Mode) is typically used for applications to execute within the User Space of the system. This would be the formal or business logic that executes on an embedded system or performance computer.

In recent years RISC-V has extended the use of three additional modes: Hypervisor-Supervisor-Mode, Virtual-User-Mode, and Virtual-Supervisor-Mode [6]. These additional modes support the virtualization of platforms built with the RISC-V ISA. The next section covers details of the RISC-V Virtualization extensions.

2.1.3.1 RISC-V Virtualization Extensions

Very similar to the ARMv8 [10, 11, 12], the RISC-V virtualization extensions are a set of hardware features and instructions that are designed to support efficient and secure virtualization within the RISC-V architecture [6, 10] . These extensions provide the same capabilities as mentioned with the ARMv8 to enable VMMs for running multiple guest VMs [13]. The RISC-V privilege model is employed to implement the virtualization extensions. A hardware acceleration thread, or hart, contains all the state information mandated by the RISC-V ISA: program counter (PC) and registers that track the virtualized extensions and respective execution mode [9].

In the RISC-V privilege model the modes of privilege include U, S, M, VU, VS [6]. Table 2 below provides a brief overview and description of each of these modes.

Table 2: RISC-V Virtualization Privilege Model

Virtualization Mode	**Nominal Privilege**	**Abbreviation**	**Name**	**Two-Stage Translation**
0	**U**	**U-Mode**	**User mode**	**Off**
0	**S**	**HS-Mode**	**Hypervisor-extended supervisor mode**	**Off**
0	**M**	**M-Mode**	**Machine mode**	**Off**
1	**U**	**VU-Mode**	**Virtual user mode**	**On**
1	**S**	**VS-Mode**	**Virtual-supervisor mode**	**On**

The current virtualization mode, denoted V indicates whether the hart is currently executing in a guest. When V=1, the hart is either in virtual VS-mode, or in VU-mode atop a guest OS running in VS-mode. When V=0, the hart is either in M-mode, in HS-mode, or in U-mode atop an OIS running in HS-mode. The virtualization mode also indicates whether two-stage address translation is active (V=1) or inactive (V=0).

The hypervisor extensions add a stage of address translation, from guest physical addresses to supervisor physical addresses, to virtualize the memory and memory mapped I/O subsystems for a guest operating system. In this case HS-mode will act the same as S-mode, but with additional instructions and controls status registers (CSRs) that control the new stage of address translation and support hosting a guest OS in VS-mode.

An OS or hypervisor running in HS-mode uses the supervisor CSRs to interact with the exception, interrupt, and address-translation subsystems. Additional CSRs are provided to HS-mode, but not to VS-mode to manage two-stage address translation and to control the behavior of VS-mode guest.

2.1.4 Registers

RISC-V utilizes 32 registers: x0-0x31 [9]. These registers specifically address the Application Binary Interface, return address and stack pointer, temporary values, and persistent values, and function arguments, and return value. Table 3 below provides a description of the RISC-V registers and gives a summary of their purpose.

Table 3: RISC-V Registers

Register	Description
Zero	Always hardwired and always holds the value 0 to simplify the ISA
ra	Holds return address
sp	Holds stack pointer
t0-t6	Holds temporary values that are not guaranteed to persist after a function call
s0-s11	Hold persistent values across function calls
a0-a1	Hold the first two arguments of a function and return value
a2-a7	Holds any remaining arguments

2.1.4 Control and Status Register

The Control and Status Registers (CSRs) are the RISC-V system registers that operate and control the platform's current state [6]. CSRs can be read or written to by specific CSR instructions and are reserved for M-Mode execution. A RISC-V implementation, however, may contain additional CSRs that are accessible to additional privilege levels. If a different privilege level attempts to access a CSR and does not have authorization, then an illegal instruction exception is thrown.

2.1.5 Exceptions and Interrupts

The RISC-V ISA utilizes a trap model for exceptions and interrupts. During runtime if an exception or an interrupt occurs the processor provides a mechanism that invokes an unscheduled procedure call to a random address [9].

Traps are separated into two categories: synchronous and asynchronous. Synchronous traps are exceptions resulting from an instruction execution. For example, a synchronous trap may occur if software attempts to access an invalid memory address. An asynchronous trap is an interrupt and is considered an external event that occur asynchronously to the instruction stream. RISC-V considers three sources of interrupts: software, timer, and external interrupts. Software generated interrupts give software engineers the ability to interrupt the CPU. Timer interrupts occur when a hardware thread (HART) time comparator meets the interrupt condition set by the software engineer. External interrupts are asserted by the platform level controller (PLIC) and can be the result of external hardware used to interface with the platform.

The rest of this section provides an overview of development frameworks used to design RISC-V based SoCs.

2.1.6 Rocketchip Softcore

The Rocketchip softcore is an open-source RISC-V implementation that serves as a versatile and customizable platform for designing SoC solutions [14, 15]. This review aims to provide an overview of the features, and applications of the Rocketchip softcore. The Rocketchip provides a system designer with the ability to configure and customize a RISC-V based softcore with specific requirements [15]. Because of the open-source nature of the Rocketchip a designer can configure system features such as cache size, number of cores, ISA extensions; it is also flexible in that it can integrate with other IP blocks to create complete SoC solutions. The Rocketchip can integrate custom IP blocks such as memory controllers, I/O interfaces, accelerators, and peripherals. An example of a Rocketchip design is given below in Figure 1. The example demonstrates a dual core Rocket system where each core comes equipped with a page table walker, L1 instruction cache, and data cache.

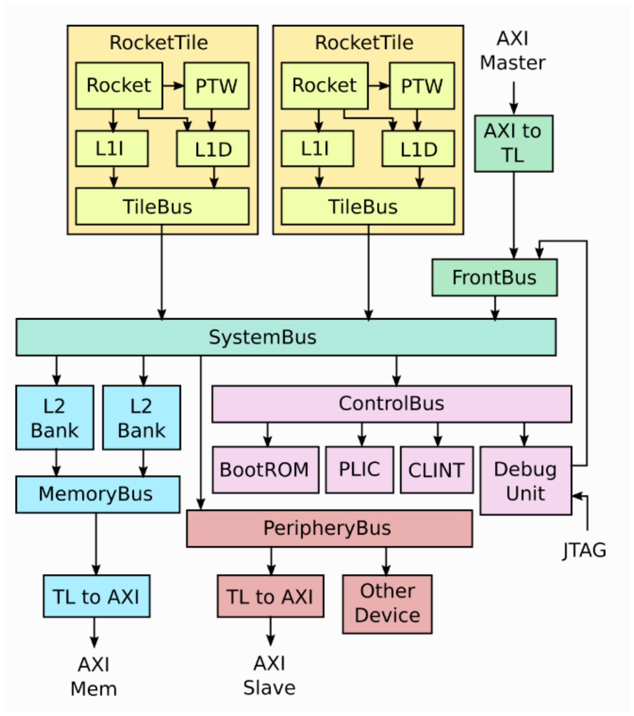


Figure 1: Example Rocketchip Design Generated from Chipyard[14]

Each Rocket tile interfaces to the system bus that extends the L2 cache banks and are interconnected to the memory bus. The memory bus in this design is responsible for interfacing with the DRAM controller via the AXI converter. Memory mapped IO devices on the control bus provide control for the platform level interrupt controller, controller level interrupts, BootROM, etc.

This Rocketchip design is just one example of how a system designer can implement an open source SoC and develop a RISC-V based design. The process of taking this concept and generating synthesizable Verilog to test on hardware involved using the open-source framework known as Chipyard. The following section provides details on Chipyard and how it is used to create designs such as the one presented in Figure 1.

2.1.7 Chipyard

Chipyard is a framework for designing and evaluating full-system hardware. It is composed of a collection of tools and libraries designed to provide an integration between open-source and commercial tools for the development of SoCs [16, 17]. The framework consists of register transfer language (RTL) core generators for RISC-V processor cores.

Chipyard relies on software tooling such as the construction hardware in Scala embedded language (Chisel) [16]. Chisel is a hardware description language that uses the Scala semantics to describe digital electronics and circuits. The goal of Chisel is to facilitate advanced circuit generation and design reuse for both ASIC and FGPA digital logic designs.

Hardware construction primitives are written in the Scala programming language and provides the ability to architect and generate complex parameterizable systems [18]. Chisel can be used to generate synthesizable Verilog with the Flexible Internal Representation for RTL tool (FIRRTL) [19]. FIRRTL is an intermediate representation for digital circuits design as a platform for writing circuit level transformations. The FIRRTL compiler can analyze a code written in Chisel to produce valid Verilog.

2.2 Kernel

The kernel is the minimum amount of software that resides in memory to facilitate full control over the CPU, Memory, and all hardware peripherals [20]. The kernel arbitrates data flow from computer hardware to the applications executing in real time. Applications perform read and write operations on hardware via system level calls that are handled by the kernel [20, 21]. Figure 2 below illustrates the interface between the hardware of a system to the application layer.

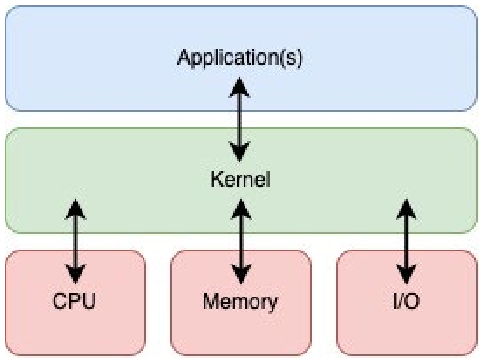


Figure 2: Kernel Interface to Hardware and Software[22]

From Figure 2 the flow of data flows from the hardware layer through the kernel to the application layer.

There are two common kernel architectures the monolithic kernel and microkernel [20, 21, 22]. The work conducted in this thesis research focuses specifically on microkernel architecture and the following section provides an overview into what a microkernel is.

2.2.1 Microkernel

Microkernels are a prominent design choice for embedded system development [23]. This review provides insight into the design principles, advantages, and challenges associated with implementing a microkernel for an embedded system design.

In recent years microkernels have gained a significant amount of momentum in the field of operating system design [1, ,13, 24]. Microkernel design, as the name suggests, revolve around the idea that the kernel should provide the minimum functionality to bootstrap the system and delegate services that manage hardware resources to user-space components.

Microkernels strive for minimalizing its software footprint by only implementing essential functions within the kernel. A kernel's core functionality can be thought of as: scheduling, inter-process communication, and memory management. Keeping the minimalization aspect of microkernel design intact results in an easier to understand codebase that is easier to verify and maintain.

Part of what makes microkernel design a good fit for embedded system development is modularity. Designing an embedded system with a microkernel requires that the system is structured and broken into separate independent component. Each separate component handles specific system services. For example, a file system component would be solely responsible for interfacing with non-volatile storage hardware within a system. The kernel would provide the necessary scheduling, IPC, and memory management that allows the file system component to efficiently perform its intended tasks. Other examples of independent components for an embedded system architecture could include device drivers for interfacing with digital sensors, network protocols to provide network connectivity and many more. Providing clear separate between the kernel and user space components allows for easier debugging, maintenance, and extensibility of the system.

When a user-space component requires communication between components the microkernel provides mechanisms in the form of IPC. Communication typically takes place in the form of message passing by exchanging requests, responses, and notifications. This loose coupling enables individual components the ability to integrate into a larger functional role while maintaining isolation that enables fault tolerant design. Figure 3 below illustrates a high-level concept of microkernel design for an embedded system.

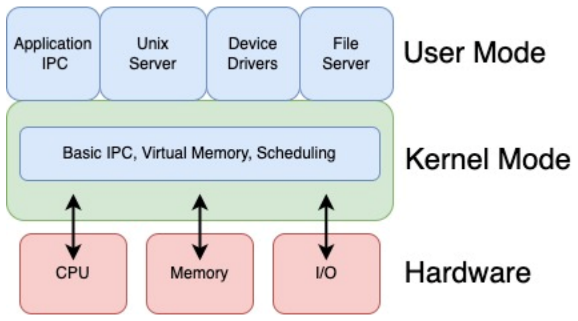


Figure 3: Example Microkernel Architecture

In the illustration provided from Figure 3 above this example contains at the base a CPU, Memory, and I/O hardware peripherals. The microkernel executes as the next layer above this hardware and provides the basic mechanisms for IPC, memory management, and scheduling. To enable complete functionality of the hardware peripherals the user-space components execute on top of the kernel to orchestrate system functionality such as file access, I/O access, etc.

In addition to modularity, microkernel design also provides the ability for a system designer to employ policy-driven design and protect address spaces. Policy driven design enforces decisions about resource allocation, scheduling, and security to be made outside of the kernel. User-space components would be designed to account for the specific policy configuration of the system and execute in concurrency with the rest of the user-space components to ensure that policies are adhered to. To ensure proper isolation microkernels can also employ protected address spaces. This adds an additional layer of security and fault tolerance design as it prevents direct access to memory from a component that may otherwise have unauthorized access. Both policy-driven design and protection of memory addresses are two ways that a microkernel can bolster the security posture of a system.

An example of a microkernel that employs these design principles discussed is the seL4 microkernel [1, 13, 24]. The seL4 microkernel is considered the world's most secure and efficient microkernel [13, 25]. The next section provides an overview and details about what makes this microkernel so unique.

2.2.2 seL4

seL4 is a third-generation microkernel that is based on the L4 family of microkernels [13, 25, 26]. It was influenced by the EROS projects and features abstraction for virtual address spaces, thread scheduling, and IPC. seL4 is the first OS kernel with a machine-checked functional correctness proof at the source code level [8, 13, 25, 26]. The formal proof of correctness ensures that the source code compiled and intended to execute on a system is indeed "defect-free", making seL4 a strong candidate for building safety-critical and secure systems.

The seL4 proof chain consists of a model that includes properties for confidentiality, integrity, and availability. These properties feed into the abstract model which define the parameters of what the system is capable of. This abstract model is an obfuscated version of the C implementation which results in the machine-checked binary code that is intended to execute on a target platform. Figure 4 below illustrates seL4's proof chain.

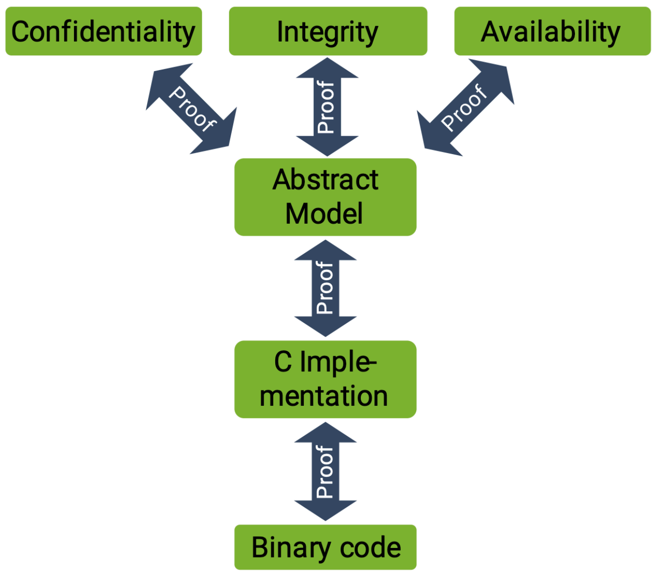


Figure 4: sel4 Proof Chain[25, 26]

The core of seL4's verification is the functional correctness proof, which claims that the C implementation is free of defects. The formal specification of the kernel's functionality is expressed in a mathematical language called higher order logic (HOL). The HOL specification is represented by the abstract model in Figure 4 above. The C-implementation is then a refinement of the abstract model, meaning that the possible behaviors of the C code are a subset of those allowed by the abstract model. Kernel behavior is expressed by the abstracted specification, thus preventing the kernel from behaving in ways that are not allowed by the specification. This allows a kernel such as seL4 to shield itself from attacks such as stack smashing, null pointer dereference, and any code injection or control flow-hijacking [25, 26].

As a part of the formally verified proof seL4 also provides a way to additionally verify that the executable binary produced by the compiler. This is an additional security step that prevents malicious compilers from building in Trojans and back doors to the OS [25, 26]. Figure 5 below illustrates the translation validation proof chain.

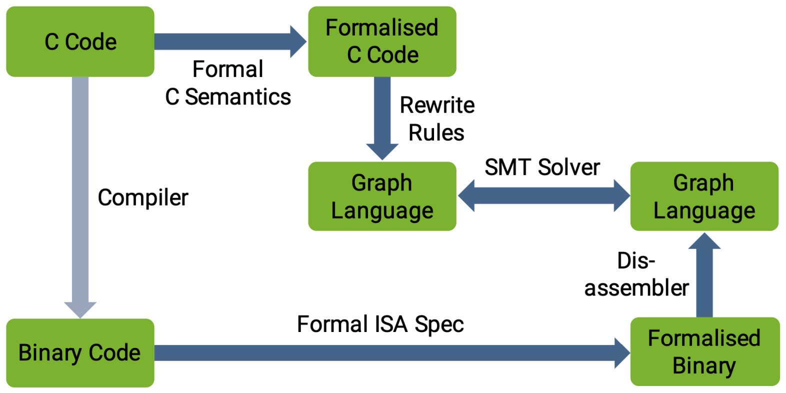


Figure 5: Translation Validation Proof Chain[25, 26]

The proof chain illustrated in Figure 5 is an automatic process that happens in multiple stages. A formal model of the processor's ISA formalizes the binary in the theorem prover. The formalized ISA feeds the dissembler, written as a HOL theorem, to translate the low-level representation in a graph language that represents the control flow. The formalized C program is then translated into the same graph language which allows for comparison of two programs to assess for equivalent representation [25, 26].

As mentioned previously, seL4 implements security properties into the abstract model to bolster the security posture of a system designed with seL4. This allows seL4 to integrate policy-driven design that allows an architect the ability to prevent an unauthorized read/writes to sensitive data, modifications, and unauthorized use of resources [25, 26]. In addition to a policy-driven design seL4 comes with a unique mechanism known as capabilities. Capability-based access control for authorization is what sets seL4 apart from other L4 kernels. Think of a capability as a pointer with access rights. Figure 6 below illustrates a high-level example of a capability.

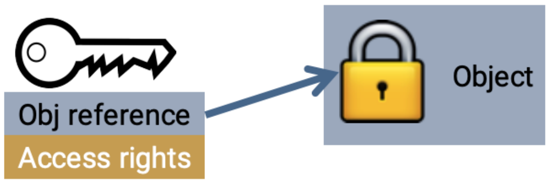


Figure 6: Capability Representation[25, 26]

seL4 defines three kinds of capabilities:

- Capabilities that control access to kernel objects such as thread control blocks (TCBs)
- Capabilities that control access to abstract resources such as interrupt control
- Untyped capabilities that are responsible for ranges and allocations.

Capabilities, again, provide that level of fine-grained control of a system for a designer to implement a policy-driven approach to embedded system design.

In addition to being a microkernel, seL4 is also a hypervisor. It is completely capable of executing virtual machines that support feature rich operating systems such as Linux [25, 26]. When executing as a hypervisor, the seL4 microkernel executes in hypervisor mode. To support virtualization of a guest virtual machine (VM), the kernel deploys what is known as a virtual machine monitor (VMM) to assist with handling system calls and events from the guest VM. The VMM executes in what is known as user mode. Finally, the guest VM executes in what is known as guest kernel mode in the context of seL4. Figure 7 below illustrates a high-level example of what a virtualized architecture deployed on seL4 may look like.

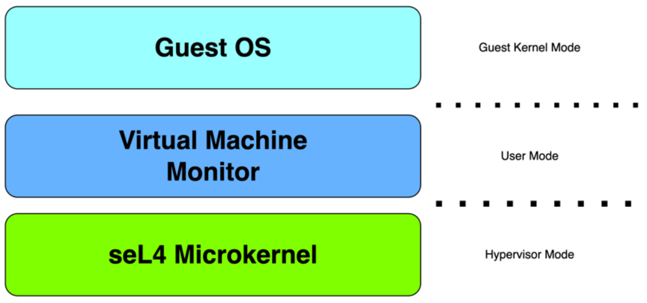


Figure 7: seL4 Virtualization Example

Virtualized design is vital to meeting SWaP-C requirements for safety-critical systems. By employing the hypervisor capabilities for seL4 a system designer can now account for microkernel design, security principles, and SWaP-C.

To summarize, seL4 is a shining example of how microkernel design can be leveraged with virtualization and formal methods to design and develop next generation embedded system for a variety of applications within. The formal proof of correctness provides a solid foundation to achieve a design that can adhere to fault-tolerance, real-time, and security requirements. Sections 2.2.3 and 2.2.4 will follow up on specific details of what virtualization and formal methods are to provide more context.

2.2.3 Virtualization

Virtualization refers to the creation of virtual instances or environments that exist on a single physical platform [27, 28, 29]. This type of abstraction allows for the distribution of physical resources to multiple compute domains; resources such as processors, memory, and storage. This section provides a literature review of virtualization and its key concepts including hypervisors, virtual machines, and containers.

Traditional hypervisors, such as those used for cloud/server computing (VMWare ESXi and Microsoft Hyper-V) provide full hardware abstraction and enable the simultaneous execution of multiple operating systems. In recent years, containerization employed through technology such as Docker and Kubernetes have emerged as a lightweight mechanism for providing virtualization within a host operating system [30].

There are three common virtualization architectures', full-virtualization, para-virtualization, and containerization [27]. Full virtualization emulates all hardware peripherals on a platform which allow an unmodified guest operating system to run. Para-virtualization requires modification to the guest operating system. Often these modification address performance/efficiency requirements not met by full virtualization. Finally, containerization allows for the creation of lightweight isolated environments within a guest operating system.

Virtualization has a broad range of applications across different domains [28, 29, 30]. The most common example includes server virtualization. In this scenario multiple virtual machines can be hosted on a single physical server. This is how services such as Amazon Web Services (AWS), Microsoft's Azure, and Google Cloud provision instances for their customers to manage and utilize. Another example of virtualization is more common to software development and engineering, Desktop virtualization. This example delivers virtual desktops to developers that allow them to include a variety of tools and software needed for engineering purposes. Network virtualization is the process by which virtual local area networks can be provisioned and used. In recent years there has been a focus to address security concerns related to the isolation of virtual instances.

Virtualization has transformed the way computer systems are designed, deployed, and managed [1]. Specific embedded system design virtualization has allowed designers to consolidate multiple systems into one platform to meet SWaP-C requirements. As virtualization continues embedded system design will be able to address concerns of security, resource contention, and performance.

2.2.3.1 Hypervisor

A Hypervisor, more formally known as a virtual machine monitor (VMM), is a layer of software that enables the creation and management of virtual machines (VMs) [24, 32]. A VMM will abstract the underlying hardware resources and provide the capability to execute multiple OS or RTOS environments [25, 32].

There are two main types of hypervisors: Type-1 Hypervisor (Bare-metal) and Type-2 Hypervisor (Hosted Hypervisor) [31, 32]. A Type-1 hypervisor runs directly on the physical hardware without the need for an underlying firmware or operating system to manage it. Type-1 hypervisors offer high performance and scalability making them suitable for embedded system design. A Type-2 hypervisor runs on top of a host operating system and relies on the host to manage hardware resources. The host OS also provides virtualization capabilities to the Type-2 hypervisor via software. An example of a Type-2 hypervisor would be VMWare Workstation or Oracle's Virtual Box which are often used for desktop virtualization, development, and testing environments. Ongoing research is being performed to assess the usability for a DevSecOps pipeline for a containerized environment executing within an embedded system deploying a Type-1 hypervisor.

The key features and functionalities of a hypervisor include:

- Virtual Machine Management - Provides mechanisms to allocate hardware resources such as CPU, memory, I/O, etc.
- Resource Isolation – Ensures resource isolation between VMs to ensure they run independently and securely.
- Hardware Abstraction – Hypervisors need to abstract the underlying physical

hardware and present the resources to guest VMs so that they can function without interference.

- Performance – Hypervisors employ techniques such as paravirtualization, hardware-assisted virtualization, and memory/page sharing to optimize the performance of VMs and minimize overhead caused by virtualization.

Hypervisors have been instrumental in enabling SWaP-C requirements for embedded devices.

2.2.3.2 Virtual Machine

A VM is a software emulation of a physical system [32]. Typically, a VM will execute a form an OS or RTOS alongside of multiple VMs. Each VM executes as an independent entity with access to a system physical resource. VMs abstract the underlying physical hardware to provide a virtual representation of hardware components such as processors, memory, I/O, etc. This provides the ability for the OS to function as if it were executing on the bare-metal system. In this paradigm VMs offer strong isolation between concurrent instances running on the same physical machine. Each VM operates within its own virtualized environment, with its own dedicated resources and independent execution space. The isolation ensures that actives or issues within one virtual machine does not interfere with others.

2.2.4 Formal Methods

Formal methods are the intersection of computer science and mathematics. This field involves utilizing mathematical techniques for the specification, development, analysis, and verification of software and hardware systems [33]. Over the years formal methods have evolved to approach a wide variety of problems in the commercial and defense sectors [34, 28, 29]. This review explores the advancements in formal methods by providing an overview of the tools involved in model checking, theorem proving, abstract interpretation, and symbolic execution.

Advancements in formal methods has allowed this technique to see continued industry adoption. For example, the automotive industry has embraced formal methods as a tool to enable autonomous driving [29]. The broad list of applications includes developments in domains such as aerospace, automotive, medical devices, and cybersecurity. These advancements include model checking techniques that can automatically verify system properties against formal specifical. Theorem proving is also another element of formal methods that has advanced through the years [35]. Theorem proving is the process in which correctness is proved using logical deduction rules. Abstract interpretation provides a framework for approximating system behavior. Bug detection is made possible by symbolic execution which explores program paths.

Despite the progress that has been made, several limitations adopt wide adopting of formal methods. The main challenge is the scalability of formal verification techniques to handle large and complex systems. Traditional engineering is a costly process, and adding the process of specifying verification techniques would add to the overhead of a typical engineering process. Even if formal methods were to be utilized the talent pool for providing such expertise is very limited [36].

Presently research and development are being done that focus on developing scalable and automated tools for formal methods [36, 24]. The objective of these tools is to simplify modeling and verification. AI/ML is also being explored to develop models that will help with the efficiency of formal analysis. Finally, research and development for domain specific tools is also being explored to allow traditional engineers the ability to utilize formal methods with low overhead.

Undoubtedly, formal methods can be a valuable tool for high assurance system design. This review has provided an overview of some of the advancements, applications, and challenges associated with formal methods. While progress has been made, the tooling and expertise is not readily available to allow traditional engineers to utilize formal methods in high assurance system design specific to their domain.

2.2.5 Reference Design

The reference designed implemented in this thesis uses the formally verified microkernel seL4 as a Type 1 hypervisor. The hardware peripherals passthrough are the system timer and UART device of the ZCU102 development kit. The VMM maps the memory space for the VM along with the hardware devices: system timer, and UART. Traditionally this type of paradigm causes system overhead and performance hits, however, for this evaluation the performance hits are negligible with respect to evaluation for the reference design implemented on a RISC-V Rocketchip. If this were a real time system, then the real time performance would need to be evaluated.

CHAPTER 3: IMPLEMENTATION

This section covers the details surrounding the design and tooling needed to implement seL4 on RISC-V. It starts with an overview of the reference design implemented in this effort and continues with a discussion of the design tooling for compiling both hardware and software components.

This section also provides an overview of the design decisions and introduces the design decisions and provides a brief overview of the boot process for each stage of testing.

3.1 Reference Design

The reference architecture for this design can conceptually be thought of as three separate components: FPGA softcore design, seL4 microkernel, and guest OS. The FPGA softcore design is the Rocketchip. seL4 microkernel is effectively the OS/hypervisor of the architecture. Linux is the guest operating system (OS) that represents the virtual machine (VM). Figure 8 below illustrates each of these components.

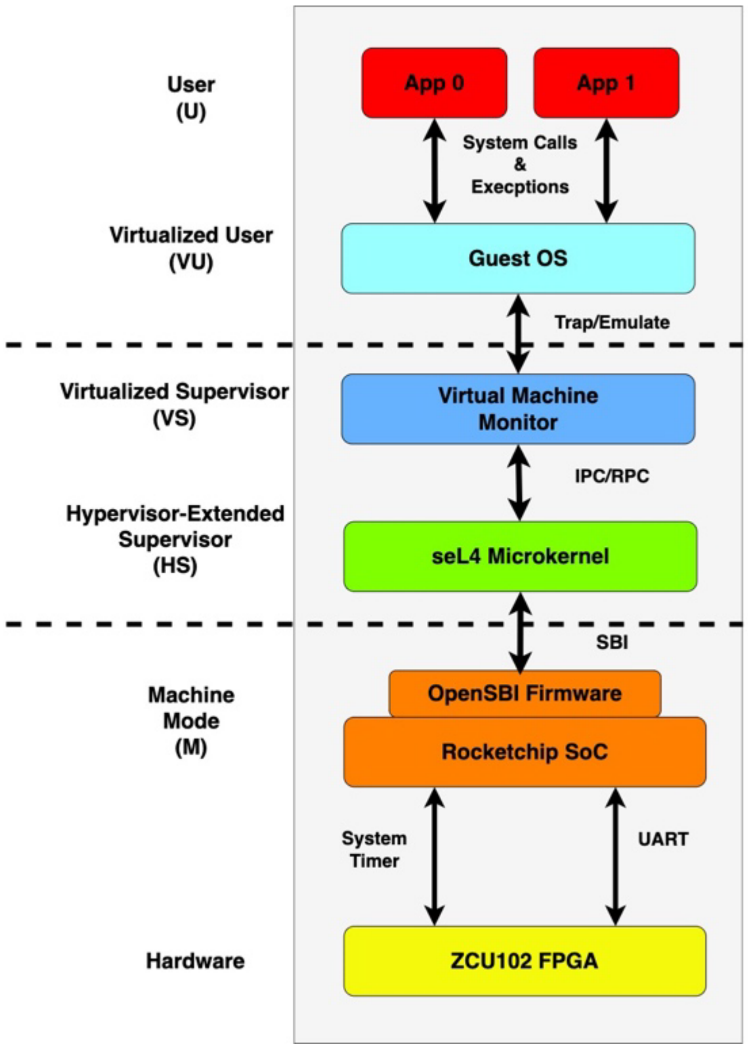


Figure 8: Reference Architecture

The first component comprises of the Rocketchip softcore executing in the FPGA fabric of ZCU102. The ZCU102 is a part of the Xilinx ZynqMP Ultrascale+ architecture which is notably equipped with an AXI interconnect that allows for the interaction of the Processing System (PS) with the Programmable Logic (PL) [37]. From a development perspective this interconnect provides a system designer with the ability to load, execute, and troubleshoot PL designs such as the Rocketchip. OpenSBI is the firmware element that assists with bootstrapping the system during boot time [38]; it is loaded into DDR and executes in M mode. This first stage bootloader is responsible for configuring the PLIC and UART, and System Timer peripherals prior to relinquishing execution to seL4.

Once OpenSBI has configured the Rocketchip softcore it loads and executes the seL4 microkernel. The seL4 microkernel configures and bootstraps the Rocketchip resources to setup for loading and executing the VMM. Once the VMM starts execution it has the responsibility of providing memory mapping for the guest VM to load and execute properly. As the guest VM loads and executes any system calls and interrupts are trapped and emulated by the VM to achieve virtualization.

3.2 Development Environments

This effort utilized a variety of development environments to achieve an seL4 software image that would boot on the Rocketchip softcore. All tools used in this effort are open source except for the Vivado tool suite [39]. Vivado is utilized for FPGA design for the Xilinx ZCU102 development kit. Vivado was selected mainly since Xilinx provides all the necessary tooling for synthesis, bitstream generation, and troubleshooting for the ZCU102 development kit.

3.2.1 seL4

The first step in porting seL4 to the RISC-V architecture was to compile the source code using the RISC-V toolchain [40, 41, 42]. There are two approaches to doing this. The first approach is to install all the necessary tooling for RISC-V and seL4 on a host development environment. The second approach involves utilizing a docker container released and maintained by the seL4 community [40]. For this effort the latter approach was utilized as it provided less overhead time getting a development environment setup.

3.2.2 Linux

Linux was also cross-compiled using the RISC-V toolchain and a framework known as

Buildroot [43]. Buildroot is a simple and efficient tool to generate embedded Linux images. It allows for configurability for toolchains to cross-compile a Linux image for a target platform. It also provides tooling to generate complete root filesystems with default tooling. For example, benchmark utilities such as cyclic test can be added by configuring Buildroot.

3.2.3 Chipyard

Chipyard is a framework for designing and evaluating full-system hardware. It is composed of a collection of open-source tools and libraries [44, 45]. A system developer can use Chipyard to integrate between commercial or open IP blocks for FPGA. Chipyard allows for the configuration of system design with options for selecting processor cores, accelerators, and system components.

3.2.4 FPGA

The Vivado tool suite v2022.2 was used as the integrated development environment (IDE) for FPGA synthesis. To integrate the Rocketchip design on the Xilinx ZCU102 it required a license to synthesize for the ZCU102.

3.3 General Design Decisions

To reduce development overhead it was decided that a single guest VM would be deployed for this effort. In addition, it was also decided that the Rocketchip would be a single core operating at 300MHz as this was the upper limit of the PL clock. By keeping the development overhead low this effort was completed and provides the basis for expanding the Rocketchip to multicore, and for a multi guest deployment scheme. More details of the seL4 and Rocketchip design are provided in Section 3.3 and 3.4 respectively.

3.3 seL4 Design

The seL4 software design is made up of several elements: the seL4 kernel, fileserver, VMM image, and guest image. The kernel is responsible for bootstrapping the system by loading the fileserver and VMM. The fileserver in this case is holds the guest VM image, Linux, in a CPIO format. During the boot process the VMM will interface with the fileserver to load and execute Linux as a guest VM.

3.4 Rocketchip Design

To emulate a cyber physical system representative of a real-time safety critical system the Rocketchip developed for this effort was a 4-core design with a memory controller, cache-controller, interrupt-controller, MMIO-port, system clock, and a ROM component. Additionally, as a part of the RISC-V architecture there is a system for handling interrupts, and core local interrupts. The Rocketchip device tree, describing the SoC, is available in Appendix A.

The system described in Table 4 is the minimal design needed to be representative of an SoC capable of deploying a SWaP-C system. The nodes expressed in the device tree from Appendix A are mapped to addresses within the system. Table 4 provides the Rocketchip address map.

Table 4: Rocketchip Address Map

```
Generated Address Map
            3000 -     4000 ARWX error-device@3000
           10000 -    20000 R X  rom@10000
         2000000 -  2010000 ARW   clint@2000000
         2010000 -  2011000 ARW   cache-controller@2010000
         c000000 - 10000000 ARW   interrupt-controller@c000000
        40000000 - 80000000 ARWXC memory@40000000
        ff000000 - 100000000  RWX  mmio-port-axi4@ff000000
```

The address map of the Rocketchip design for this effort includes the hardware peripherals (i.e., System Clock and UART) mapped between 0xFF000000 – 0x100000000. The system is equipped with 1GB of RAM with starting address 0x40000000 – 0x80000000. The platform-level interrupt controller (PLIC) provides the centralized interrupt prioritization and routes shared platform-level interrupts among multiple harts is mapped to address 0xC000000. The cache-controller is the hardware block responsible for managing the memory and is mapped to address 0x2010000. The core local interrupt controller is mapped to address 0x2010000. The error-device responsible for tracking bus errors of the Rocketchip design is mapped to 0x3000. Finally, the Rocketchip ROM which contains initial instructions to bootstrap the system during a power on reset (POR) event is mapped to address 0x10000. The assembly code synthesized into the ROM at address 0x10000 is included in Table 5.

Table 5: ROM Assembly Code

```
.section .text.start, "ax", @progbits
.globl _start
_start:
  csrwi 0x7c1, 0 // disable chicken bits
  li s0, DRAM_BASE
  csrr a0, mhartid
  la a1, _dtb
  jr s0

.section .text.hang, "ax", @progbits
.globl _hang
_hang:
  csrwi 0x7c1, 0 // disable chicken bits
  li s0, DRAM_BASE
  csrr a0, mhartid
  la a1, _dtb
  jr s0
1:
  wfi
  j 1b

.section .rodata.dtb, "a", @progbits
.globl _dtb
.align 5, 0
_dtb:
    .word 0
    .word 0
```

The assembly code provided in the table above is used to bootstrap the system during a POR event. This code is responsible for reading the first instruction at DRAM_BASE and executing subsequent instructions from there. The FPGA design utilized a combination of the generated Rocketchip softcore and Xilinx IP for interfacing with the PS, AXI bus, PS Reset, and UART. From Figure 9 (Appendix B) the Rocketchip design sources the clock and reset signals from the ZynqMP Ultrascale+ IP Core. The two AXI interconnect cores are responsible for fetching/writing to the mem_axi and mmio_axi bus for the Rocketchip design. Resets for the design are driven by the Processor system reset. For UART communication the Concat core is used to tie to the interrupt signal of the Rocketchip design.

3.5 Build Procedures

The work performed in this thesis is admittedly hard to keep track of. It is the culmination of multiple projects spanning across multiple frameworks. This section attempts to sort out a procedure to recreate the Rocketchip softcore for the ZCU102, Linux kernel, and seL4 monolithic image. This section also provides the procedure for loading the design with the resulting software images for test repeatability.

3.5.1 Build Dependencies

A Linux host environment was used for this effort. Specifically, Ubuntu v22.04. This section provides the command needed to satisfy the host environment dependencies needed for recreating this work. The command provided in Table 6 below will install the host dependencies.

Table 6: Host Dependency Install Command

```
sudo apt install build-essential bison flex git libssl-dev ninja-build \
   u-boot-tools pandoc libslirp-dev pkg-config libglib2.0-dev libpixman-1-dev \
   gettext-base curl xterm cmake python3-pip default-jre

pip3 install pykwalify packaging pyelftools
```

Once the host tools are installed the version of each tool can be verified. Table 7 below provides the exact version number of each host dependency installed for this effort.

Table 7: Host Dependency Versions

Dependency	Version
gcc	11.3.0
bison	3.8.2
Flex	2.6.4
Git	2.34.1
Libssl-dev	3.0.2
Ninja-build	1.10.1
u-boot-tools	2023.01
pandoc	2.9.2.1
Libslirp-dev	4.7.0.1
Pkg-config	0.29.2
Libglib2.0-dev	2.64.6
Libpixman-1-dev	0.42.2-1
Gettext-base	0.21
Curl	7.81.90
Xterm	3.72
Cmake	3.22.1
Python3-pip	22.0.2
Java	11.0.19
Pykwalify	1.8.0
Packaging	23.1
Pyelftools	0.29

3.5.1 RISC-V Toolchain

The RISC-V toolchain v12.2.0 was used for this effort. This section provides the commands needed to extract the source, configure, and build the toolchain at that specific revision. For compiling embedded software code, such as seL4 the "riscv64-unknown-elf-gcc is used.

Table 8: Build Commands for riscv64-unknwon-elf- Toolchain

```
git clone https://github.com/riscv/riscv-gnu-toolchain
./configure --prefix=/opt/riscv
make
```

To build the Linux kernel the "riscv64-unknown-linux" toolchain will be needed as well. The following command will build the "riscv64-uknown-linux" toolchain assuming the "riscv64-gnu-toolchain" project has already been cloned and configured just as above:

Table 9: Build Command for riscv64-uknown-linux Toolchain

```
make linux
```

The "prefix" argument will install the resulting compiled toolchains into the "/opt/riscv" directory of the Linux host environment. A user can now add the toolchain to the system PATH environment variable by executing the following command:

Table 10: Command to add Toolchain to PATH

```
export PATH=$PATH:/opt/riscv/bin
export CROSS_COMPILE=riscv64-unknown-elf-
```

3.5.2 Rocketchip

The Rocketchip port was largely based on existing work that was available for the ZCU104. The ZCU104 port was modified slightly to generate the Verilog to port into Vivado for the ZCU102. This section aims to provide a build procedure and highlight those changes to recreate the Rocketchip softcore for the ZCU102.

The first step in building the Rocketchip softcore is to download the source code for the ZCU102 port. This can be done by executing the following command at the Linux command line:

Table 11: Git Clone Command for BAO Hypervisor Project

```
git clone https://github.com/bao-project/bao-demos -b rocket
cd bao-demos
```

Once the project has been cloned environment variables can be created to ease the process of generating the Rocketchip softcore. Table 12 provides the necessary commands needed to add the requisite environment variables.

Table 12: Environment Setup Commands

```
export PLATFORM=rocket-fpga-zcu102
export ARCH=riscv
export BAO_DEMOS=$(realpath .)
export BAO_DEMOS_WRKDIR=$BAO_DEMOS/wrkdir
export BAO_DEMOS_WRKDIR_SRC=$BAO_DEMOS_WRKDIR/srcs
export BAO_DEMOS_WRKDIR_BIN=$BAO_DEMOS_WRKDIR/bin
export BAO_DEMOS_WRKDIR_PLAT=$BAO_DEMOS_WRKDIR/imgs/$PLATFORM
export BAO_DEMOS_WRKDIR_IMGS=$BAO_DEMOS_WRKDIR_PLAT/$DEMO
mkdir -p $BAO_DEMOS_WRKDIR
mkdir -p $BAO_DEMOS_WRKDIR_SRC
mkdir -p $BAO_DEMOS_WRKDIR_BIN
mkdir -p $BAO_DEMOS_WRKDIR_IMGS
cp -R ./platforms/rocket-fpga-zcu104 ./platforms/rocket-fpga-zcu102
```

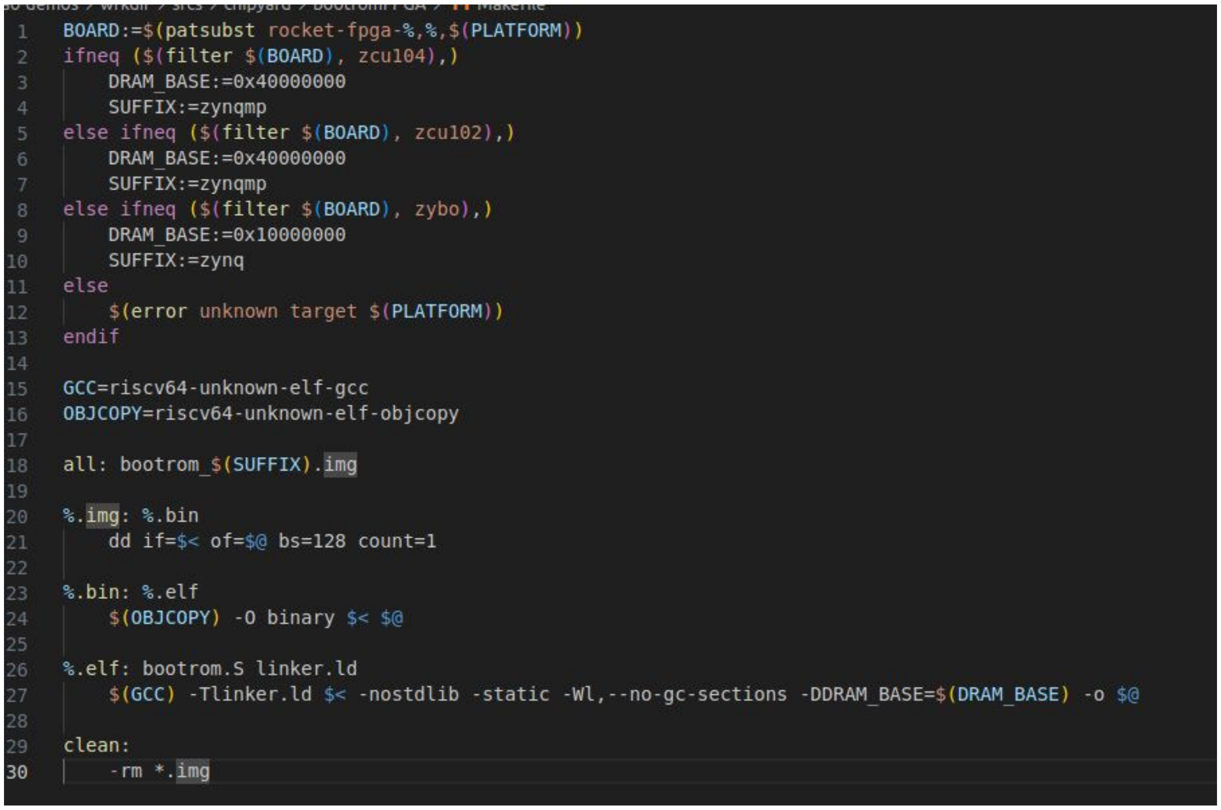
The first change to this repository that is specific to the ZCU102 port is copying over the requisite platform definition from the folder "rocket-fpga-zcu104". The next step involves cloning the Chipyard/Rocketchip repos.

Table 13: Clone the Chipyard/Rocketchip Repos

```
export BAO_DEMOS_CHIPYARD=$BAO_DEMOS_WRKDIR_SRC/chipyard
export BAO_DEMOS_ROCKETCHIP=$BAO_DEMOS_CHIPYARD/generators/rocket-chip
git clone https://github.com/ucb-bar/chipyard.git $BAO_DEMOS_CHIPYARD
cd $BAO_DEMOS_CHIPYARD
git checkout 64632c8
./scripts/init-submodules-no-riscv-tools.sh
git apply $BAO_DEMOS/platforms/$PLATFORM/patches/0001-add-rocket-hyp-fpga-support.patch
git -C generators/boom apply $BAO_DEMOS/platforms/$PLATFORM/patches/0001-boom-add-
```

```
usehyp-option.patch
git -C generators/ariane apply $BAO_DEMOS/platforms/$PLATFORM/patches/0001-ariane-add-
usehyp-option.patch
cd $BAO_DEMOS_ROCKETCHIP
git remote add hyp https://github.com/josecm/rocket-chip.git
git fetch hyp
git checkout hyp
```

The next change specific to the ZCU102 port involves updating the Makefile specific for the FPGA BootROM that will be built into the Rocketchip softcore design. Following the environment variables setup from the above instructions the BootROM project is located at "wrkdir/chipyard/bootromFPGA/Makefile". Figure 10 below summarizes the change needed to build the BootROM for the ZCU102.



```
1   BOARD:=$(patsubst rocket-fpga-%,%,$(PLATFORM))
2   ifneq ($(filter $(BOARD), zcu104),)
3       DRAM_BASE:=0x40000000
4       SUFFIX:=zynqmp
5   else ifneq ($(filter $(BOARD), zcu102),)
6       DRAM_BASE:=0x40000000
7       SUFFIX:=zynqmp
8   else ifneq ($(filter $(BOARD), zybo),)
9       DRAM_BASE:=0x10000000
10      SUFFIX:=zynq
11  else
12      $(error unknown target $(PLATFORM))
13  endif
14
15  GCC=riscv64-unknown-elf-gcc
16  OBJCOPY=riscv64-unknown-elf-objcopy
17
18  all: bootrom_$(SUFFIX).img
19
20  %.img: %.bin
21      dd if=$< of=$@ bs=128 count=1
22
23  %.bin: %.elf
24      $(OBJCOPY) -O binary $< $@
25
26  %.elf: bootrom.S linker.ld
27      $(GCC) -Tlinker.ld $< -nostdlib -static -Wl,--no-gc-sections -DDRAM_BASE=$(DRAM_BASE) -o $@
28
29  clean:
30      -rm *.img
```

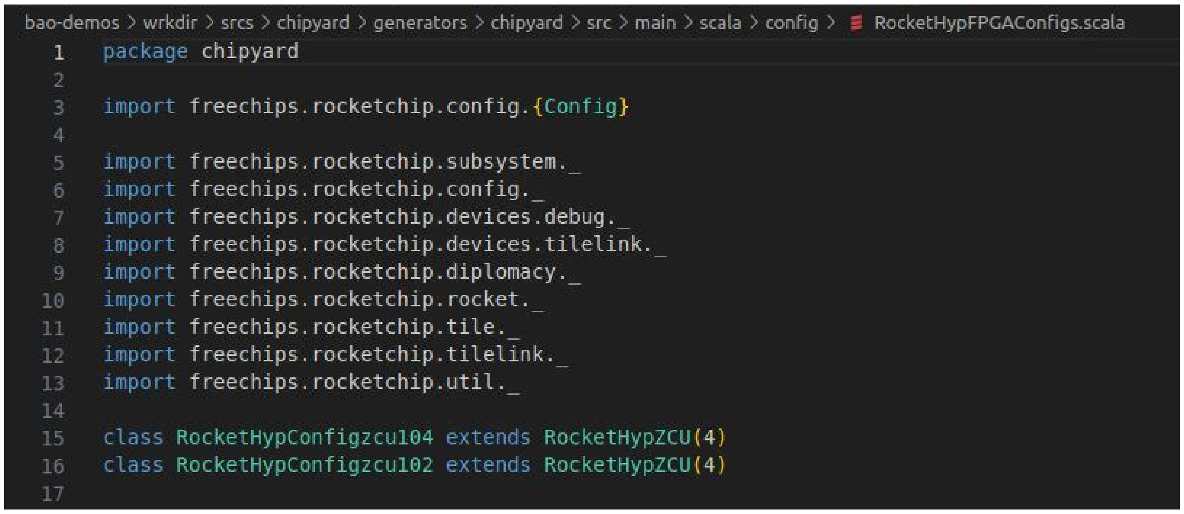
Figure 9: BootROM Makefile Modification for ZCU102

The specific change for the ZCU102 includes lines 5 to 7. The change is a conditional statement that utilizes the PLATFORM environment variable which was previously set to "rocket-fpga-zcu102". Once this change has been implemented the BootROM can be generated by executing the following command:

Table 14: Build Command for BootROM

make -C $BAO_DEMOS_CHIPYARD/bootromFPGA

The resulting file generated from the above command is "bootrom_zynqmp.img". This binary will be utilized later one during the generation of the Rocketchip softcore. The final modification for the ZCU102 includes updating the Scala definition for the Rocketchip that adds the hypervisor extensions. Figure 11 summarizes the change needed to build the Rocketchip with the hypervisor extension.



bao-demos > wrkdir > srcs > chipyard > generators > chipyard > src > main > scala > config > RocketHypFPGAConfigs.scala

```
package chipyard

import freechips.rocketchip.config.{Config}

import freechips.rocketchip.subsystem._
import freechips.rocketchip.config._
import freechips.rocketchip.devices.debug._
import freechips.rocketchip.devices.tilelink._
import freechips.rocketchip.diplomacy._
import freechips.rocketchip.rocket._
import freechips.rocketchip.tile._
import freechips.rocketchip.tilelink._
import freechips.rocketchip.util._

class RocketHypConfigzcu104 extends RocketHypZCU(4)
class RocketHypConfigzcu102 extends RocketHypZCU(4)
```

Figure 10: Scala Modification for ZCU102

At this point the Verilog can be generated by executing the command included in Table 15 below.

Table 15: Command to Generate Rocketchip Verilog

```
export BAO_DEMOS_ROCKET_CONFIG=RocketHypConfig$(echo $PLATFORM | awk '{split($0, A,"-"); print A[length(A)]}')

make -C $BAO_DEMOS_CHIPYARD/sims/vcs verilog SUB_PROJECT=rocket \
   CONFIG=$BAO_DEMOS_ROCKET_CONFIG
```

Once the resulting Verilog has been generated the files can be imported into Vivado and a new project can be created. The following commands will generate the project and build the design using Vivado.

Table 16: Build Commands for Vivado

```
export VIVADO_CORES=$(nproc)
export BAO_DEMOS_VIVADO_SCRIPTS=$BAO_DEMOS/platforms/$PLATFORM/scripts
vivado -nolog -nojournal -mode batch -source $BAO_DEMOS_VIVADO_SCRIPTS/create_ip.tcl
vivado -nolog -nojournal -mode batch -source
$BAO_DEMOS_VIVADO_SCRIPTS/create_design.tcl
vivado -nolog -nojournal -mode batch -source $BAO_DEMOS_VIVADO_SCRIPTS/build.tcl
```

The result of the build commands should include a new Vivado project located at "wrkdir/imgs/rocket-fpga-zcu102". This step also provides a bitstream that can be programmed into the PL of the ZCU102. However, before the softcore will properly execute with a software payload loaded into DDR RAM the bootchain of the PS must properly configure the ZCU102. The easiest way to do this at boot time is to create a first stage bootloader (FSBL) based on the PL design in Vivado.

Vivado provides a mechanism to import a hardware design into Vitis. Vitis is another Xilinx tool provided to create board support package projects (BSP) and applications from. The process of creating a BSP based on a Vivado design includes exporting the hardware design and setting up a new FSBL project in Vitis. Once the FSBL project has been built a new boot image (BOOT.BIN) can be created by using the following bootable image format (BIF) configuration:

Table 17: Bootable Image Format (BIF) Configuration

```
the_ROM_image:
{
        [bootloader, destination_cpu=a53-0] ./workspace/rocket_fsbl/Debug/rocket_fsbl.elf
        [pmufw_image] pmufw.elf
        [destination_device=pl]. /rocket.bit
        [destination_cpu=a53-0, exception_level=el-3, trustzone] bl31.elf
        [destination_cpu=a53-0, load=0x00100000] system.dtb
        [destination_cpu=a53-0, exception_level=el-2] u-boot.elf
}
```

The above configuration in Table 17 provides the necessary information to build a boot image capable of booting the ZCU102 and configuring it such that the custom FSBL and Rocketchip softcore can now boot a software payload properly. The first entry is the custom Vitis application based on the BSP provided by the Vivado design. The third entry is the bitstream, Rokcetchip, that will be programmed into the PL. The rest of the software images are prebuilt binaries from the Xilinx website for the v2022.2 release. Once the BIF file has been configured a new boot image can be built with the following command:

Table 18: Bootgen Command to Generate Boot Image

```
bootgen -image bootgen.bif -arch zynqmp -o BOOT.BIN
```

The resulting BOOT.BIN file can be loaded onto the boot partition of an SD card. The ZCU102 can be placed into SD card boot mode (Shown in Figure 12 below) and once the board is powered on the PS will be configured from the FSBL and the PL will be programmed with the Rocketchip softcore awaiting a valid payload to be stored in at address 0x40000000.

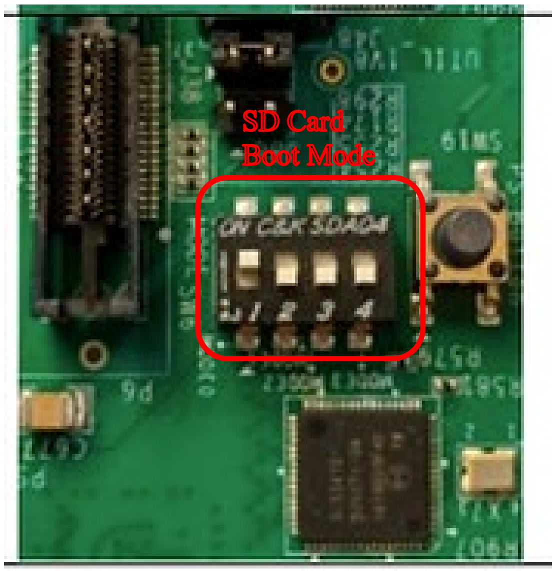


Figure 11: SD Card Boot Mode for ZCU102

3.5.3 Linux

Buildroot was used to build the Linux kernel for this effort. Specifically, the Linux kernel v5.11 was used. This section aims to provide the steps involved in downloading, configuring, and building the source for the Linux kernel. Assuming all the environment variables from the previous section are set the following commands will setup and download the Linux source code:

Table 19: Download and Configure Linux Commands

```
# Configure environment variable
export BAO_DEMOS_LINUX=$BAO_DEMOS/guests/linux
export BAO_DEMOS_LINUX_REPO=https://github.com/torvalds/linux.git
export BAO_DEMOS_LINUX_VERSION=v5.11
export BAO_DEMOS_LINUX_SRC=$BAO_DEMOS_WRKDIR_SRC/linux-
$BAO_DEMOS_LINUX_VERSION

# Download the Linux Source
git clone $BAO_DEMOS_LINUX_REPO $BAO_DEMOS_LINUX_SRC\
   --depth 1 --branch $BAO_DEMOS_LINUX_VERSION
cd $BAO_DEMOS_LINUX_SRC

# Point to target architecture platform specific config to be used by buildroot
export BAO_DEMOS_LINUX_CFG_FRAG=$(ls $BAO_DEMOS_LINUX/configs/base.config\
   $BAO_DEMOS_LINUX/configs/$ARCH.config\
   $BAO_DEMOS_LINUX/configs/$PLATFORM.config 2> /dev/null)
```

With the Linux source code downloaded the Buildroot source code can now be downloaded and configured. Table 20 below includes the commands to download and configure Buildroot.

Table 20: Download and Configure Buildroot

```
# Setup Buildroot environment variables
export BAO_DEMOS_BUILDROOT=$BAO_DEMOS_WRKDIR_SRC/\
buildroot-$ARCH-$BAO_DEMOS_LINUX_VERSION
export BAO_DEMOS_BUILDROOT_DEFCFG=$BAO_DEMOS_LINUX/buildroot/$ARCH.config
export LINUX_OVERRIDE_SRCDIR=$BAO_DEMOS_LINUX_SR


# clone buildroot
git clone https://github.com/buildroot/buildroot.git $BAO_DEMOS_BUILDROOT\
    --depth 1 --branch 2020.11.3
cd $BAO_DEMOS_BUILDROOT


# Build Linux
make defconfig BR2_DEFCONFIG=$BAO_DEMOS_BUILDROOT_DEFCFG
make linux-reconfigure all
```

The result of the above commands in Table 20 should be a Linux kernel image located in "output/images/". This image is the guest VM image used to boot in the reference architecture. The next section covers how to incorporate this Linux image as a guest VM in a monolithic seL4 image.

3.5.4 seL4

The seL4 community provides a Docker container that includes all the necessary dependencies needed to perform seL4 design. This section provides an overview of downloading the seL4 Docker container, configuring an seL4 project for RISC-V, and compiling that project. The command provided in Table 21 below demonstrates how to build the seL4 Docker container.

Table 21: seL4 Docker Image Build Command

```
git clone https://github.com/seL4/seL4-CAmkES-L4v-dockerfiles.git
cd seL4-CAmkES-L4v-dockerfiles
make user
```

Once "make user" completes the Docker environment will be activated. The next step is to download, configure and build seL4test. The commands are provided below for seL4test.

Table 22: Build Command for seL4test

```
repo init -u https://github.com/seL4/sel4test-manifest.git
repo sync
mkdir build
../init-build.sh -DPLATFOR=Rocketchip-zcu102 -DRISCV=64
```

seL4test is used to verify the port of seL4 to a RISC-V platform. This program can be loaded to DDR RAM address 0x40000000 for the Rocketchip softcore to fetch and execute. The next part of this section demonstrates how to build the seL4 VMM for loading and executing a guest VMM. The sel4-riscv-vmm source code can be downloaded using the following command:

Table 23: Download sel4-riscv-vmm Source Code

```
repo init -u https://github.com/SEL4PROJ/sel4-riscv-vmm-manifest.git -m master.xml
repo sync
```

Configuration and building of the seL4 monolithic image can be performed with the following command:

Table 24: seL4 Monolithic Build Command

```
mkdir build
cd build
../init-build.sh -DPLATFORM=Rocketchip -DElfloaderImage=elf -DRISCV64=TRUE
ninja
```

The resulting ELF image is in the directory “build/elfloader-tool” as the file “elfloader”. The file needs to be converted to a binary format which can be done with the “riscv64-unknown-elf-objcopy” tool. The process to copy the ELF format to a BIN format can be achieved by executing the following:

Table 25: Objcopy Command

```
riscv64-unknown-elf-objcopy -O binary elfloader payload
```

"payload" is what is loaded into DDR RAM for the Rocketchip to boot during a POR event. The next section provides an overview on how to program the PL and load the "payload" file in DDR RAM.

3.5.5 Execute seL4 on Rocketchip

The final step in the build procedure is to load software into DDR RAM and program the PL with the bitstream. This section provides the necessary commands using the Xilinx tool "xsct" to program the PL and load a payload file into DDR RAM.

Table 26: Commands to Flash Bitstream and Load Payload

```
xsct
Connect
Target -set -filter {name =~ "PSU"}
#program bitstream
fpga -no-revision-check -f rocket.bit
# Load software into DDR RAM
dow -data /path/to/payload 0x40000000
```

This final section concludes the build procedures used in this effort to successfully load and execute seL4 on the Rocketchip softcore for evaluation. The next section provides an overview of the tests performed to verify the Rocketchip and to gather performance/benchmarks of seL4 on RISC-V.

CHAPTER 4: RESULTS & DISCUSSION

4.1 Verification Tests

The tests covered in this effort aimed to validate the proof-of-concept of a virtualized architecture with the RISC-V H-extensions. The benefits of these are that the results indicate whether the reference architecture is functional or not. This process initially began with validating the Rocketchip design in the FPGA fabric of the ZCU102. Once the Rocketchip was validated the port of the seL4 port was validated with the seL4Test project. Upon successful validation of the seL4 test suite the cross-compiled Linux kernel was validated on the Rocketchip. Once each individual element has been validated the reference design was deployed and benchmarking tests were performed.

4.1.1 Rocketchip Verification

To verify the Rocketchip used for this effort was validate the first step was to perform an initial synthesis of the FPGA design. A successful synthesis prepares the design for bitstream generation. The bitstream is what ultimately loads the design on to the FPGA for the ZCU102, therefore, a successful synthesis is a good indication of whether the design is valid for the intended hardware. Vivado v2022.2 also provides feedback as to size constraints of the target platform. If the design were beyond the capacity of the FPGA fabric, then the synthesis process would error.

The second step of verification was to validate that during a POR event the ROM in the design is attempting to access instructions at 0x40000000. This required modification to the design to setup an experiment that allowed a user to reset the Rocketchip processor at will. To achieve this a pushbutton peripheral was tied to the “external_reset” line and a system integrated logic analyzer ILA, was tied to the “mem_axi” bus of the design. This experiment would force a reset condition and capture data being fetched on the “mem_axi” bus at the intended address of 0x40000000. Figure 13, see appendix C, provides a diagram of the modification implemented for this experiment. The test harness adds two additional elements to the FPGA design a “reset” signal that is tied to “SW20” of the ZCU102, and a “System ILA” which ties to the “mem_axi” bus of “rocket_0”. The reset signal force a power on reset (POR) scenario that allows the System ILA to capture the instructions transmitted on the “mem_axi” bus.

4.1.2 seL4 Test Suite

The seL4 test project aims to test seL4 and some of its user libraries on many different targets to indicate a successful port. seL4test is a testing framework that runs test suites on target hardware. It is used to test both kernel and user code on a target platform. For this effort the seL4 test suite was used to successfully validate the port of seL4 to the Rocketchip.

4.1.3 seL4 Hypervisor and VM

The final stage of testing is to load and execute the reference architecture discussed in this thesis work. This process involves compiling the monolithic seL4 image that includes all the software elements to boot the reference design. The monolithic image is loaded into a specified address in RAM that allows the Rocketchip ROM to fetch the image and boot seL4 and subsequent software components.

4.2 Verification Test Results

Verification occurred in two stages: validate a successful and perform visual observation of the Rocketchip softcore fetching data properly from DDR RAM. The first stage was a matter of performing a synthesis and bitstream generation within Vivado. The Vivado tooling provides information such as resource usage, timing summaries, and estimated power output of the Rocketchip design. The Vivado tooling also provides an indication if the design is going to be valid for the ZCU102, for example, if the design were to consume to many resources, then the synthesis process would have failed.

The second stage of Rocketchip verification was to ensure that the design was fetching instructions from the intended address at DDR RAM and operating as intended. To perform this level of verification a System ILA was used by adding it to the design in Vivado and a trigger scenario was setup to capture real time data being fetched by the Rocketchip from DDR RAM. More details on these two tests are provided in the rest of this section.

4.2.1 Rocketchip Verification

This section covers verification and validation of the Rocketchip softcore. It covers the resource utilization as well as integrated logic analyzer (ILA) captures. The ILA capture the results of loading firmware to RAM address 0x40000000.

4.2.2.1 Resource Utilization

Upon successful synthesis Vivado provided resource utilization of the FPGA design. Table 26 below provides a summary of the resources utilized by this design.

Table 27: Resource Utilization

Component	CLB LUTs	CLB Registers	CARRY8	F7 Muxes	F8 Muxes	Block RAM	DSPs	Global Clock Buffers	PS9
Axi_interconnect_0	0	0	0	0	0	0	0	0	0
Axi_interconnect_1	0	0	0	0	0	0	0	0	0
Rocket_0	126359	69365	2114	4795	284	202	60	0	0
Rst_ps8_099M	19	40	0	0	0	0	0	0	0
Xlcocat_0	0	0	0	0	0	0	0	0	0
Zynqmp_ultra_ps_e_0	264	0	0	0	0	0	0	1	1
Total	126642	69405	2114	4795	284	202	60	1	1

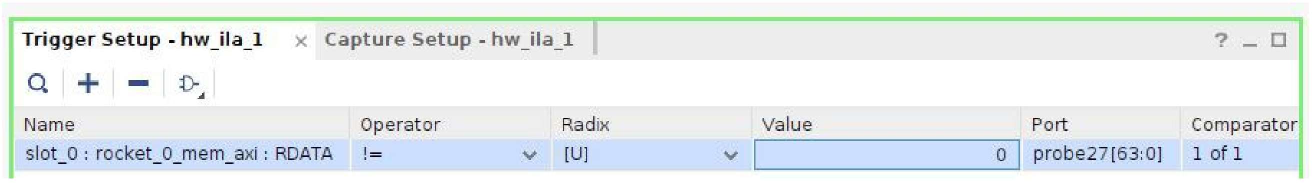
From inspection the Rocket_0 component is utilizing the most resources. Naturally this would be the case considering the softcore design implements a representative SoC for this effort. Table 23 provides specific resource utilization from the Rocket_0 softcore.

Table 28: Rocketchip Resource Usage

Component	CLB LUTs	CLB Registers	CARRY8	F7 Muxes	F8 Muxes	Block RAM	DSPs	Global Clock Buffers	PS9
Tile	28490	16081	496	1150	57	20	15	0	0
Tile_2	28480	16082	496	1150	57	20	15	0	0
Tile_1	28456	16081	496	1150	57	20	15	0	0
Tile_3	28454	16082	496	1150	57	20	15	0	0
L2 Cache	7101	2961	10	53	6	122	0	0	0
System Bus	3036	645	0	11	0	0	0	0	0
Periphery Bus	1264	503	8	0	0	0	0	0	0
BootROM	587	0	0	109	50	0	0	0	0
CLINT	196	324	24	1	0	0	0	0	0
PLIC	143	116	0	0	0	0	0	0	0
Memory Bus	98	291	0	21	0	0	0	0	0
Total	97851	69166	2026	4795	284	202	60	0	0

4.2.2.2 ILA Capture

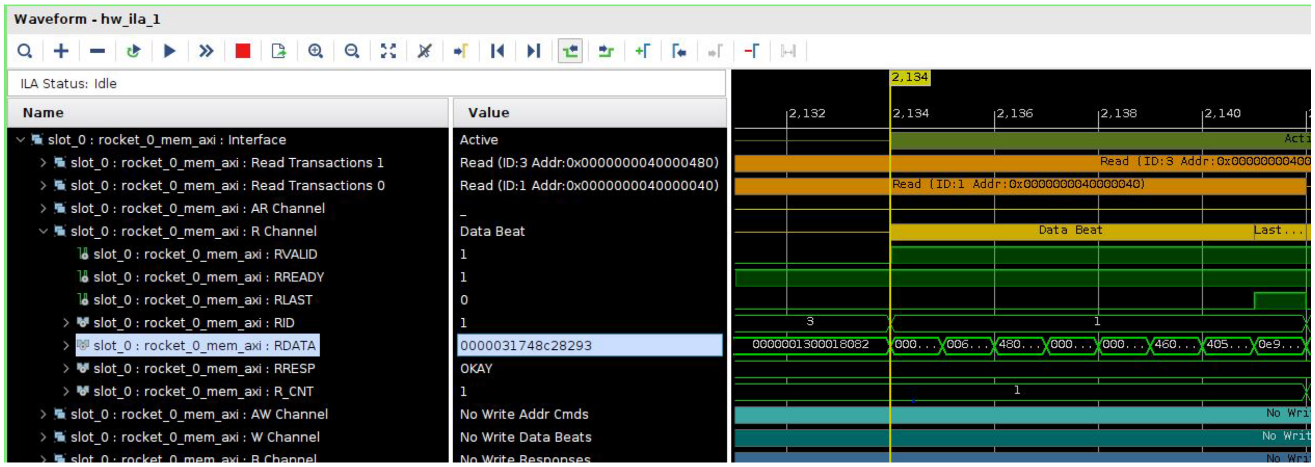
The ILA capture played an important role in verifying the Rocketchip design as it allowed for real time capture of data being transmitted on the "mem_axi" bus for the softcore. The trigger case in this setup involved capture of the RDATA on the AXI bus. The ILA probe was configured to trigger on data currently being transmitted from RAM to the softcore via the AXI bus. Figure 14 below provides the capture setup for the ILA core.



Name	Operator	Radix	Value	Port	Comparator
slot_0 : rocket_0_mem_axi : RDATA	!=	[U]	0	probe27[63:0]	1 of 1

Figure 12: System ILA Probe Condition

The probe condition from Figure 14 relies on the value currently transmitted on the AXI bus to be not equal to 0. If the condition were to be satisfied, then a resulting capture of 4096 bits would be stored in the capture of the host machine. Figure 15 below provides an example verification case where the softcore was pulled out of reset by using SW20.


Waveform - hw_ila_1

ILA Status: Idle

Name	Value
slot_0 : rocket_0_mem_axi : Interface	Active
slot_0 : rocket_0_mem_axi : Read Transactions 1	Read (ID:3 Addr:0x0000000040000480)
slot_0 : rocket_0_mem_axi : Read Transactions 0	Read (ID:1 Addr:0x0000000040000040)
slot_0 : rocket_0_mem_axi : AR Channel	_
slot_0 : rocket_0_mem_axi : R Channel	Data Beat
slot_0 : rocket_0_mem_axi : RVALID	1
slot_0 : rocket_0_mem_axi : RREADY	1
slot_0 : rocket_0_mem_axi : RLAST	0
slot_0 : rocket_0_mem_axi : RID	1
slot_0 : rocket_0_mem_axi : RDATA	0000031748c28293
slot_0 : rocket_0_mem_axi : RRESP	OKAY
slot_0 : rocket_0_mem_axi : R_CNT	1
slot_0 : rocket_0_mem_axi : AW Channel	No Write Addr Cmds
slot_0 : rocket_0_mem_axi : W Channel	No Write Data Beats
slot_0 : rocket_0_mem_axi : B Channel	No Write Responses

Figure 13: Rocketchip Verification Capture Beat 1

In this case the value read on the RDATA line of "mem_axi" is 0x0031748C28293. This is the first 64-bit instruction of the payload written to RAM address 0x40000000. During a POR event, the softcore BootROM is configured to read from that RAM address and execute RISC-V 64-bit instructions. Figure 15 demonstrates the first beat, or 8 instructions, of the payload written to address 0x40000000. The process demonstrated in Figure 15 above is showcasing the Rocketchip softcore specifying a read from RAM location 0x40000040, where the instructions fetched are the hexadecimal representation of the software loaded in RAM. A hex dump of the payload stored at that address verifies the softcore is reading the correct instructions to execute to intended program. Figure 16 below provides the hex dump of the software stored at RAM location 0x40000040.

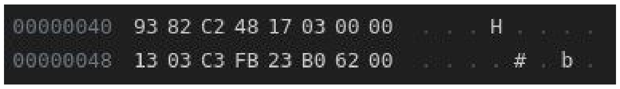
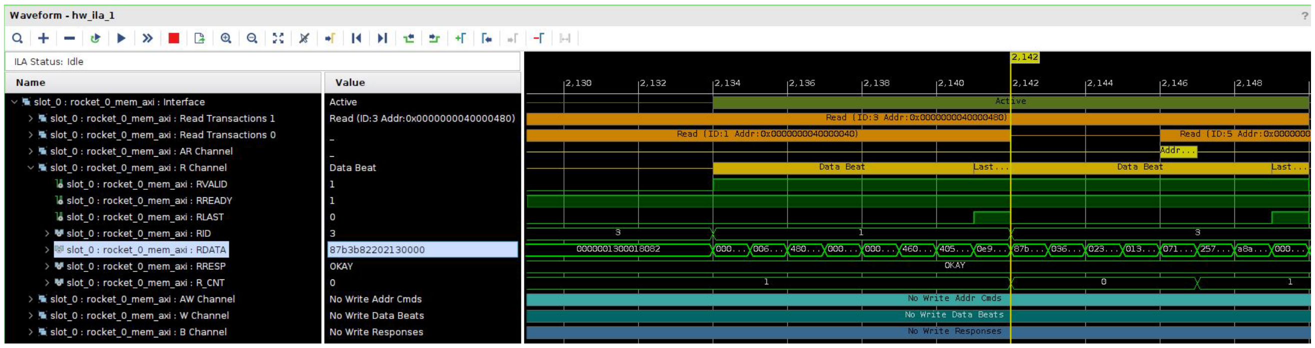

```
00000040  93 82 C2 48 17 03 00 00   . . . . H . . . .
00000048  13 03 C3 FB 23 B0 62 00   . . . . . # . b .
```

Figure 14: Hexadecimal Dump seL4 Payload Offset 0x40

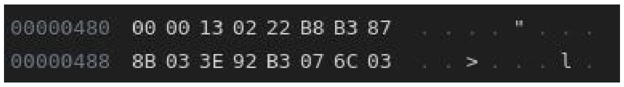
The hex dump provided in Figure 16 above confirms that the data read from the first beat matches the data of the software payload stored at that RAM location. To further verify the process a second beat was captured and confirmed in the same manner as beat one. Figure 17 below demonstrates the ILA capture of beat two during a POR event.



Name	Value
slot_0 : rocket_0_mem_axi : Interface	Active
slot_0 : rocket_0_mem_axi : Read Transactions 1	Read (ID:3 Addr:0x0000000040000480)
slot_0 : rocket_0_mem_axi : Read Transactions 0	_
slot_0 : rocket_0_mem_axi : AR Channel	_
slot_0 : rocket_0_mem_axi : R Channel	Data Beat
slot_0 : rocket_0_mem_axi : RVALID	1
slot_0 : rocket_0_mem_axi : RREADY	1
slot_0 : rocket_0_mem_axi : RLAST	0
slot_0 : rocket_0_mem_axi : RID	3
slot_0 : rocket_0_mem_axi : RDATA	87b3b82202130000
slot_0 : rocket_0_mem_axi : RRESP	OKAY
slot_0 : rocket_0_mem_axi : R_CNT	0
slot_0 : rocket_0_mem_axi : AW Channel	No Write Addr Cmds
slot_0 : rocket_0_mem_axi : W Channel	No Write Data Beats
slot_0 : rocket_0_mem_axi : B Channel	No Write Responses

Figure 15: Rocketchip Verification Capture Beat 2

The data captured from the ILA shows that beat two also includes 8 RISC-V instructions encoded in a hexadecimal format. The instructions from this beat were read from RAM address 0x400000480 and start off with the hexadecimal value of 0x87b3b82202130000. A hex dump of the software payload indicates that the instruction read from RAM matches that of the payload written to RAM starting at address 0x40000000. Figure 18 below demonstrates the hex dump at location 0x400000480.



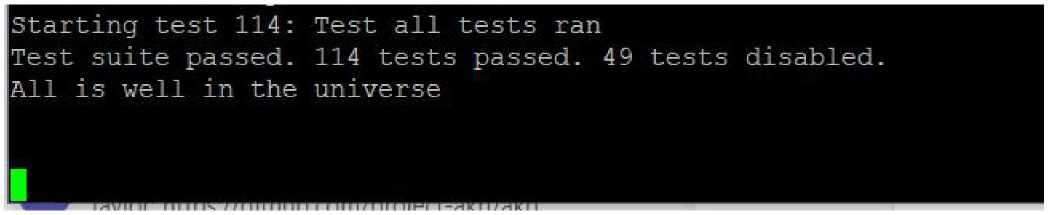
```
00000480  00 00 13 02 22 B8 B3 87  . . . . " . . .
00000488  8B 03 3E 92 B3 07 6C 03  . . > . . . l .
```

Figure 16: Hexadecimal Dump seL4 Payload Offset 0x480

From Figure 15 the instructions read from RAM address 0x40000480 match the payload at that same offset. The instructions are equivalent and stored in little-endian format. The entire payload technically could be validated this way; however, the results are expected to remain the same. Additionally, verification also falls in the form of observing the intended program execute; in this case the intended program used for verification was the seL4test payload. Results from the seL4 test suite are included in the next section.

4.2.2 seL4 Test Suite

The results of the seL4test indicate that the Rocketchip softcore can load a payload stored at RAM address 0x40000000 during a POR event and start execution. Execution starts with bootstrapping the system with SBI and then loading and executing seL4test. Each test in the suite is intended to assess the platform port. For this effort the seL4 kernel API was evaluated on the Rocketchip. The results indicate that all 114 applicable tests passed and that the port of seL4 to the Rocketchip is valid. Figure 19 below provides validation of the seL4 test suite.



```
Starting test 114: Test all tests ran
Test suite passed. 114 tests passed. 49 tests disabled.
All is well in the universe
```

Figure 17: seL4test Verification

4.3.3 seL4 Hypervisor and VM

The next step in verification of the softcore design is to run load and execute the seL4 payload that contains a guest VM.

4.3 Guest VM Performance Tests

The Linux kernel comes equipped with several performance and benchmarking tools to profile the OS on an embedded platform. Performance typically involves load and stress testing to evaluate real-time performance of an embedded system. For this effort load testing was performed using the Dhrystone program, and stress testing utilized stress-ng.

Dhrystone is a general performance benchmark test used to measure and compare general processor performance against different platforms. The goal of performance testing is to determine a benchmark to compare the Rocketchip softcore against a Quad Core ARMv8 processor. Dhrystone was used specifically because it was an easy element to build into the initramfs for this effort. Admittedly Dhrystone is not representative of a real time system, however, for this effort it is sufficient to capture basic performance results to benchmark against an ARMv8 system.

Stress-ng is a tool to load and stress a system. Stress-ng was designed to exercise various physical subsystem of a computer as well as the OS and interfaces. Stress-ng provides a variety of CPU specific stress tests that exercise floating point, integer, and bit manipulation control.

4.4 Guest VM Performance Results

This section provides an overview of the performance/benchmarking metrics captured from Dhrystone and stress-ng test runs. As mentioned previously, benchmarking compared the Rocketchip softcore design against an ARMv8 hardcore executing at 1.5 GHz on the ZCU102. The first set of results were from Dhrystone runs captured on the Rocketchip. These results are included in Table 28.

Table 29: Rocketchip Dhrystone Results

Number of Runs	Microsecond/Dhrystone	Dhrystone/Second
1000000	8.6	115740.7
10000000	8.6	116090.1
100000000	8.6	116076.6

The test results indicate Dhrystone runs on the Rocketchip softcore executes every 8.6 microseconds, this results in ~116076.6 Dhrystone runs per second. The results from Table 28 also indicate that performance is consistent when increasing the number of runs from 1000000 – 100000000. The same Dhrystone test were executed on the ARMv8, and the results are included in Table 29.

Table 30: Dhrystone Results on ARMv8

Number of Runs	Microsecond/Dhrystone	Dhrystone/Second
1000000	0.2	5889291.5
10000000	0.2	5817160.0
100000000	0.2	5906674.5

Results from Dhrystone runs on the ARMv8 indicate that it takes 0.2 microseconds to execute one Dhrystone and that the platform can execute ~5800000 Dhrystones per second. The speed at which the ARMv8 platform executes through the Dhrystone runs compared to the Rocketchip softcore is orders of magnitude faster. This result is expected considering the ARMv8 is a hardcore design, furthermore, the ZCU102 is equipped with the Cortex A-53 which is intended to be used for proof-of-concept for production ready systems.

The next set of tests involved stressing the softcore and ARMv8 platform using stress-ng. Initial stress tests were performed on the Rocketchip softcore using parameters that executed 4 workers on the CPU, 2 workers on IO, and 1 worker exercising 128 Mbytes at a time for memory. Each stress test was executed with a 10 second timeout. Table 30 below provides an overview of the commands used to stress both the Rocketchip and ARMv8.

Table 31: Stress Test

```
stress-ng –cpu 4 –io 2 –vm 1 –vm-bytes 128M –timeout 10s –metrics-brief
```

The results from the initial stress test for the Rocketchip is included in Table 31 below. The results summarize “bogus operations” executing on the CPU, IO, and VM memory in this case. Results are categorized as time in seconds that it took to execute in real-time, user time, and system time.

Table 32: Rocketchip Stress Test Results

Stressor	Bogo Ops	Real Time (s)	Usr time (s)	Sys Time (s)	Bogo ops/s (real time)	Bogo ops/s (usr+sys time)
Cpu	4	23.35	18.90	0.00	0.17	0.21
Io	4456	9.98	0.00	2.84	446.30	1569.01
Vm	0	10.28	1.13	.34	0.00	0.00

Starting with the CPU, the results form Table 31 show that the CPU was able to execute through 4 bogus operations in 23.35 seconds of real time. The total time consumed by the bogus operations occurred in 18.90 seconds of User time and did not register a single measurement in system time. For IO the system was able to execute 4456 bogus operations in 9.6 seconds. The VM memory category was unable to execute a single operation of 128Mbytes of memory in the 10 second timeout. The same stress test was executed on the ARMv8 Cortex A-53 and results are included in Table 32 below.

Table 33: ARMv8 Stress Tests Results

Stressor	Bogo Ops	Real Time (s)	Usr time (s)	Sys Time (s)	Bogo ops/s (real time)	Bogo ops/s (usr+sys time)
Cpu	208	10.89	6.55	0.00	19.09	31.76
Io	119753	10.01	0.23	0.23	11967.63	42166.56
Vm	8192	10.50	1.32	.14	780.17	5610.96

Once again, the ARMv8 hardcore outperformed the Rocketchip softcore with respect to the stress test. Starting with the CPU category the ARMv8 platform was able to execute 208 bogus operations in 10.89 seconds of real time. This results in 19.09 operations per second while under load. For the IO category the ARMv8 platform was able to execute through 119753 bogus operations in 10.01 seconds. Finally, the VM category resulted in 8192 bogus operations executing with a 128 Mbyte load.

4.5 Summary

Initial verification results were extremely satisfactory. The purpose of executing verification tests was to determine if the H-extensions were developed enough to support a virtualized guest VM executing on top of seL4. What was observed from the generation of the bitstream, and synthesis was that the design was compatible with the resources available on the ZCU102 and that the Rocketchip softcore was able to be programmed into the PL of the ZCU102. Once programmed a System ILA could then capture the instructions fetched from DDR RAM from the softcore. Further verification tests confirmed that the port of seL4 to the Rocketchip was valid, hence the successfully seL4test suite execution, and lastly the verification of booting an operation/functional guest VM would indicate that verification of seL4 on RISC-V was satisfied.

Benchmarking/performance tests went as expected as well. Performance of the softcore was not expected to outpace that of the ARMv8 Cortex A-53. Naturally the production design of the hardcore (ARMv8) would outperform the softcore because this system is entirely ready to provide the ability to create proof-of-concept designs for production ready applications. The ARMv8 hardcore is much more mature than the Rocketchip softcore. What the benchmarking does however provide is the ability to understand the benchmarks in place for a RISC-V hardcore design that operates closer to the 1.5 GHz range.

CHAPTER 5: FUTURE WORK

5.1 Design Improvements

This effort provided a baseline implementation for the reference architecture to evaluate seL4 on hardware. This work certainly makes a great starting point to extend this research into multiple areas to improve the design and collect data. One challenge encountered by this design was the inability to debug the monolithic seL4 kernel during the boot process. One design improvement that would allow for easier debugging would be the addition of a JTAG core that would allow a system designer to process software one instruction at a time. In industry development kits come with a JTAG interface and RISC-V on the Rocketchip would make troubleshooting in real time easier. Typical JTAG interfaces allow for a full system dump of the platform registers and give the ability to perform typical debug operations (step, pause, break, reset, etc.).

The main purpose of designing an embedded system is to interact with the analog world via a digital sensor. This is indeed the case with systems that measure light, sound, speed, etc. Most digital sensors interface with an SoC via a serial bus (I2C, SPI, etc.). The Rocketchip design implemented in this effort could become more representative of a real product if it had the ability to interface with a serial bus. The Vivado design suite comes equipped with an AXI Quad SPI IP core that would allow a system designer to interface a sensor that utilized the SPI protocol to the Rocketchip. That hardware peripheral could then be mapped utilizing the MMIO component of the Rocketchip and give an application developer the means to write software that interfaces with a digital sensor.

The internet of things (IoT) is entirely built up of edge compute devices such as embedded SoCs. Connectivity is enabled by the physical network hardware that allows a device to establish a TCP/UDP connection with a server to send and receive data. The ZCU102 development kit can support an Ethernet stack in the PL. The AXI Ethernet core implements an Ethernet MAC that supports 1000BASE-X and SGMII PHY interfaces. This capability would provide a system designer to develop and evaluate connected processes and applications using the virtualized reference architecture developed because of this effort.

One major benefit of an open-source ISA is that a system designer can implement custom instructions to perform dedicated actions/processes. A design improvement for this effort could entail the addition of custom instructions that can perform actions such as bootstrapping the system during POR to enabling encryption/decryption on the fly for inherently secure processing.

CHAPTER 6: CONCLUSION

6.1 Summary of Work

This body of work demonstrates a working reference architecture of seL4 on RISC-V ported to a Quadcore Rocketchip complete with H-extensions v0.6 that executes in the FGPA fabric of the ZCU102 development kit. The H-extension allows for virtualization of a guest VM which was assessed for performance and benchmarked against a Quad Core ARMv8 processor that exists on the same ZCU102 development kit. The Rocketchip was developed using the Chipyard framework for Verilog generation. Once the Verilog was generated the FPGA design was ported into the Vivado design suite to generate a bitstream for the ZCU102 development kit. To verify the Rocketchip softcore a dummy software payload compiled with OpenSBI and loaded into DDR RAM of the ZCU102. A system ILA was used to capture the Rocketchip fetching instructions from DDR RAM during a POR event.

After verification of the Rocketchip softcore the seL4 monolithic image that contains OpenSBI, seL4, VMM, and guest VM were loaded into DDR RAM and used to boot on the Rocketchip softcore. This effort concludes that the reference design can boot a virtualized guest VM executing on seL4. Booting the virtualized architecture on the Rocketchip allowed for performance and benchmarking analysis. Benchmark and performance metrics were captured from the guest OS, Linux, and used to compare to the same reference architecture executing on an ARMv8 platform also on the ZCU102.

6.2 Summary of Results

The main conclusion to draw form this effort is that there is now an open-source implementation of seL4 on RISC-V available. The Rocketchip port is a completely viable softcore SoC that can be leveraged to advance seL4 design and development with for future implementations until a hardcore variant of the RISC-V architecture with H-extensions is released for production. Continuation of research using this implementation could investigate the possibility of expanding the number for virtualized guest VMs that are deployed to adding additional hardware peripherals. More interestingly is the ability for this research to serve as a basis for further formal verification of seL4 on RISC-V for a completely formalized virtual system.

APPENDICES

Appendix A – Rocket Chip Device Tree

Table 1: Rocketchip Device Tree

```
/dts-v1/;

/ {
	#address-cells = <1>;
	#size-cells = <1>;
	compatible = "freechips,rocketchip-unknown-dev";
	model = "freechips,rocketchip-unknown";
	L23: cpus {
		#address-cells = <1>;
		chipyard #size-cells = <0>;
				L7: cpu@0 {
						clock-frequency = <0>;
						compatible = "sifive,rocket0", "riscv";
						d-cache-block-size = <64>;
						d-cache-sets = <64>;
						d-cache-size = <16384>;
						d-tlb-sets = <1>;
						d-tlb-size = <32>;
						device_type = "cpu";
						hardware-exec-breakpoint-count = <1>;
						i-cache-block-size = <64>;
						i-cache-sets = <64>;
						i-cache-size = <16384>;
						i-tlb-sets = <1>;
						i-tlb-size = <32>;
						mmu-type = "riscv,sv41";
						next-level-cache = <&L2>;
						reg = <0x0>;
						riscv,isa = "rv64imafdch";
						riscv,pmpregions = <8>;
						status = "okay";
						timebase-frequency = <1000000>;
						tlb-split;
						L5: interrupt-controller {
								#interrupt-cells = <1>;
								compatible = "riscv,cpu-intc";
								interrupt-controller;
						};
				};
				L10: cpu@1 {
						clock-frequency = <0>;
						compatible = "sifive,rocket0", "riscv";
						d-cache-block-size = <64>;
						d-cache-sets = <64>;
						d-cache-size = <16384>;
						d-tlb-sets = <1>;
						d-tlb-size = <32>;
						device_type = "cpu";
						hardware-exec-breakpoint-count = <1>;
						i-cache-block-size = <64>;
						i-cache-sets = <64>;
						i-cache-size = <16384>;
						i-tlb-sets = <1>;
						i-tlb-size = <32>;
						mmu-type = "riscv,sv41";
```

```
			next-level-cache = <&L2>;
			reg = <0x1>;
			riscv,isa = "rv64imafdch";
			riscv,pmpregions = <8>;
			status = "okay";
			timebase-frequency = <1000000>;
			tlb-split;
			L8: interrupt-controller {
				#interrupt-cells = <1>;
				compatible = "riscv,cpu-intc";
				interrupt-controller;
			};
		};
		L13: cpu@2 {
			clock-frequency = <0>;
			compatible = "sifive,rocket0", "riscv";
			d-cache-block-size = <64>;
			d-cache-sets = <64>;
			d-cache-size = <16384>;
			d-tlb-sets = <1>;
			d-tlb-size = <32>;
			device_type = "cpu";
			hardware-exec-breakpoint-count = <1>;
			i-cache-block-size = <64>;
			i-cache-sets = <64>;
			i-cache-size = <16384>;
			i-tlb-sets = <1>;
			i-tlb-size = <32>;
			mmu-type = "riscv,sv41";
			next-level-cache = <&L2>;
			reg = <0x2>;
			riscv,isa = "rv64imafdch";
			riscv,pmpregions = <8>;
			status = "okay";
			timebase-frequency = <1000000>;
			tlb-split;
			L11: interrupt-controller {
				#interrupt-cells = <1>;
				compatible = "riscv,cpu-intc";
				interrupt-controller;
			};
		};
		L16: cpu@3 {
			clock-frequency = <0>;
			compatible = "sifive,rocket0", "riscv";
			d-cache-block-size = <64>;
			d-cache-sets = <64>;
			d-cache-size = <16384>;
			d-tlb-sets = <1>;
			d-tlb-size = <32>;
			device_type = "cpu";
			hardware-exec-breakpoint-count = <1>;
			i-cache-block-size = <64>;
			i-cache-sets = <64>;
			i-cache-size = <16384>;
			i-tlb-sets = <1>;
			i-tlb-size = <32>;
			mmu-type = "riscv,sv41";
			next-level-cache = <&L2>;
			reg = <0x3>;
			riscv,isa = "rv64imafdch";
			riscv,pmpregions = <8>;
```

```
				status = "okay";
				timebase-frequency = <1000000>;
				tlb-split;
				L14: interrupt-controller {
					#interrupt-cells = <1>;
					compatible = "riscv,cpu-intc";
					interrupt-controller;
				};
			};
		};
		L18: memory@40000000 {
			device_type = "memory";
			reg = <0x40000000 0x40000000>;
		};
		L22: soc {
			#address-cells = <1>;
			#size-cells = <1>;
			compatible = "freechips,rocketchip-unknown-soc", "simple-bus";
			ranges;
			L2: cache-controller@2010000 {
				cache-block-size = <64>;
				cache-level = <2>;
				cache-sets = <1024>;
				cache-size = <524288>;
				cache-unified;
				compatible = "sifive,inclusivecache0", "cache";
				next-level-cache = <&L18>;
				reg = <0x2010000 0x1000>;
				reg-names = "control";
				sifive,mshr-count = <7>;
			};
			L4: clint@2000000 {
				compatible = "riscv,clint0";
				interrupts-extended = <&L5 3 &L5 7 &L8 3 &L8 7 &L11 3 &L11 7 &L14 3 &L14 7>;
				reg = <0x2000000 0x10000>;
				reg-names = "control";
			};
			L1: error-device@3000 {
				compatible = "sifive,error0";
				reg = <0x3000 0x1000>;
			};
			L17: external-interrupts {
				interrupt-parent = <&L3>;
				interrupts = <1 2>;
			};
			L3: interrupt-controller@c000000 {
				#interrupt-cells = <1>;
				compatible = "riscv,plic0";
				interrupt-controller;
				interrupts-extended = <&L5 11 &L5 9 &L8 11 &L8 9 &L11 11 &L11 9 &L14 11 &L14 9>;
				reg = <0xc000000 0x4000000>;
				reg-names = "control";
				riscv,max-priority = <3>;
				riscv,ndev = <2>;
			};
			L19: mmio-port-axi4@ff000000 {
				#address-cells = <1>;
				#size-cells = <1>;
				compatible = "simple-bus";
				ranges = <0xff000000 0xff000000 0x1000000>;
			};
			L20: rom@10000 {
				compatible = "sifive,rom0";
				reg = <0x10000 0x10000>;
				reg-names = "mem";
			};
			L0: subsystem_pbus_clock {
				#clock-cells = <0>;
				clock-frequency = <100000000>;
				clock-output-names = "subsystem_pbus_clock";
				compatible = "fixed-clock";
			};
		};
};
```

APPENDIX B – Rocketchip FPGA Design

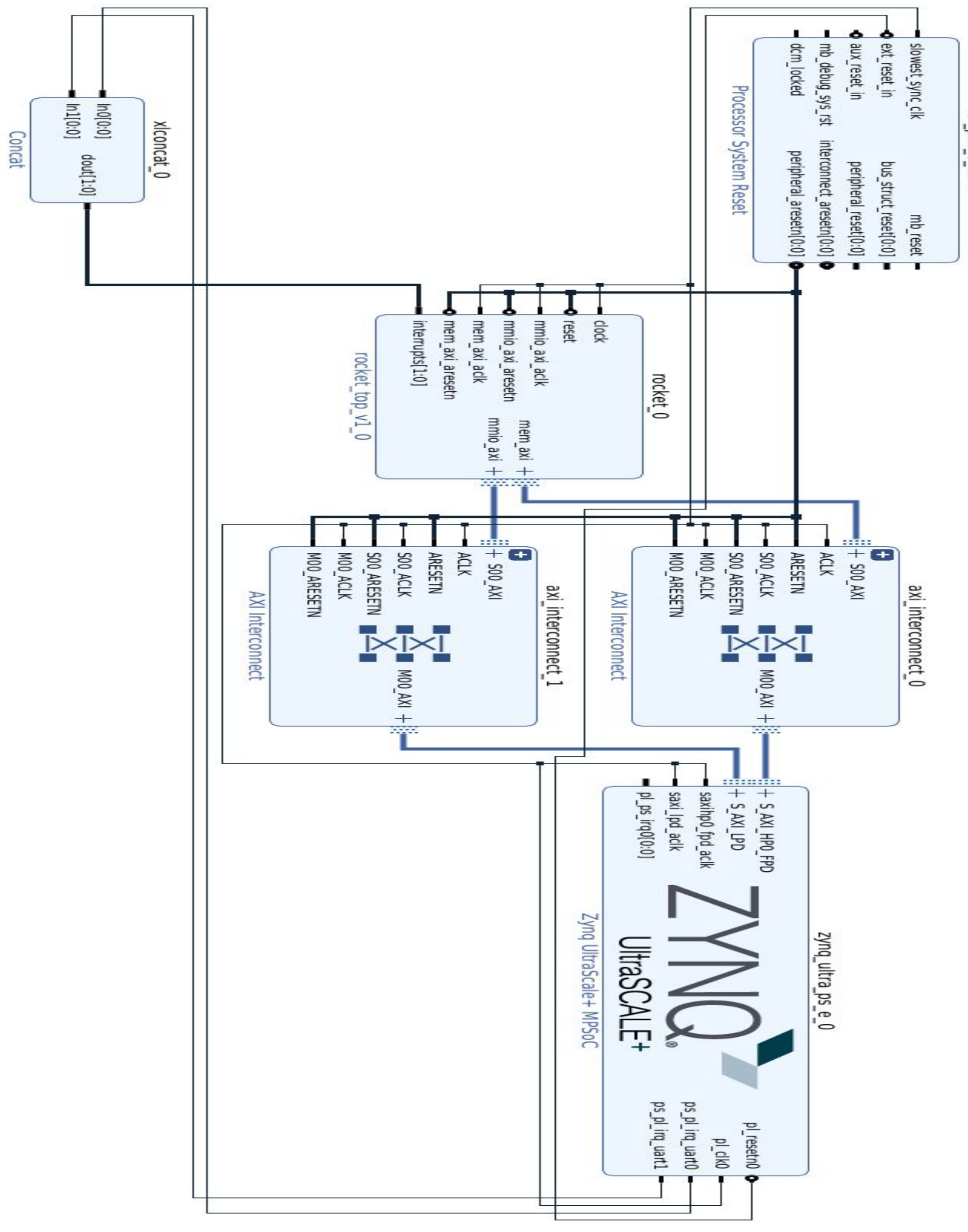


Figure 18: Rocketchip FPGA Design

APPENDIX C – Rocket Chip Test Harness

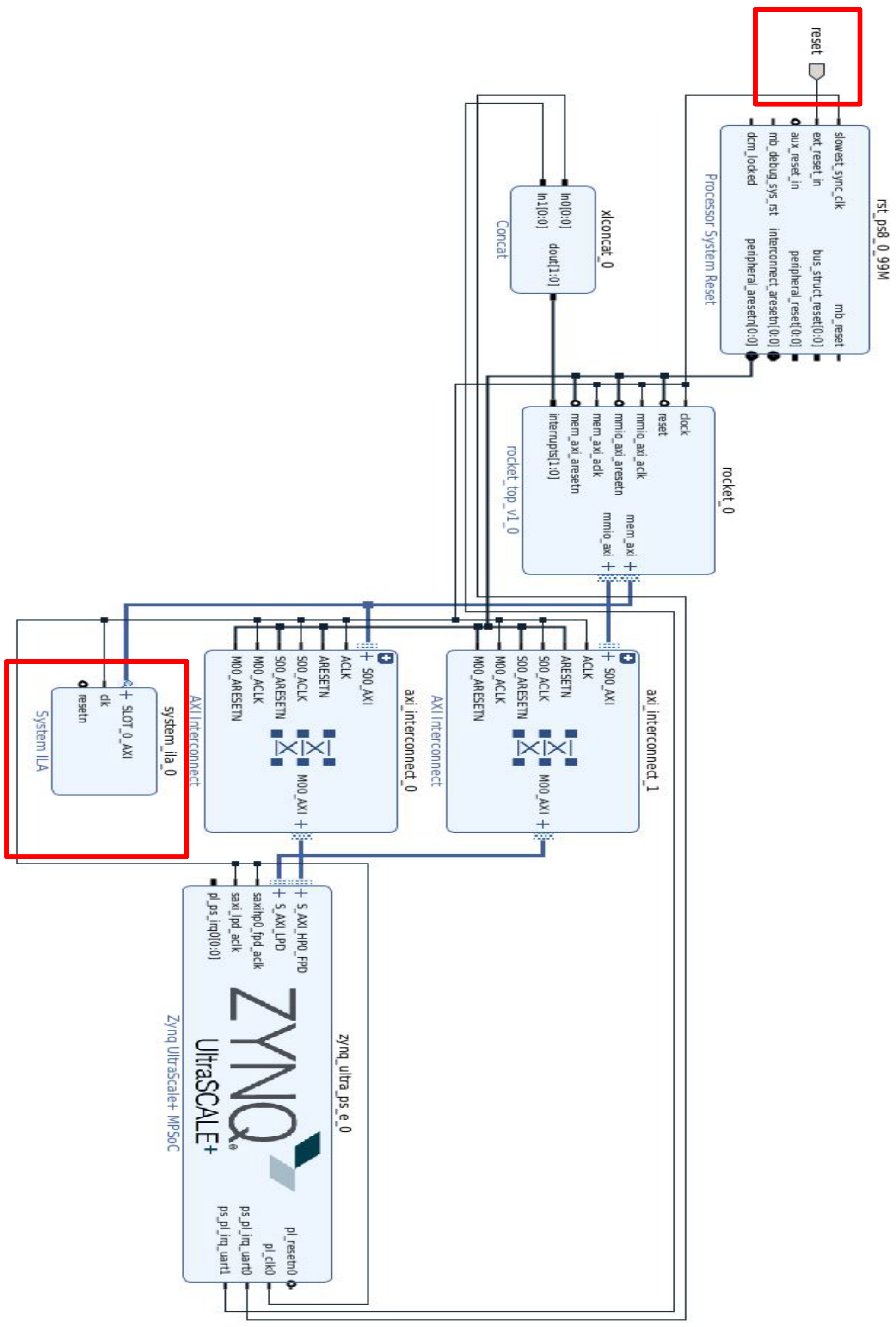


Figure 19: Rocketchip Test Harness

BIBLIOGRAPHY

[1] G. Heiser, “Virtualizing embedded systems - Why bother?,” in Proc. 48th ACM/EDAC/IEEE Des. Autom. Conf., 2011, pp. 901–905.

[2] J. Martins, A. Tavares, M. Solieri, M. Bertogna, and S. Pinto, “Bao: A lightweight static partitioning hypervisor for modern multicore embedded systems,” in Proc. Workshop Next Gener. Real-Time Embedded Syst., 2020, vol. 77, pp. 3:1–3:14.

[3] T. Kloda, M. Solieri, R. Mancuso, N. Capodieci, P. Valente, and M. Bertogna, “Deterministic memory hierarchy and virtualization for modern multi-core embedded systems,” in Proc. IEEE RealTime Embedded Technol. Appl. Symp., 2019, pp. 1–14.

[4] C. Dall and J. Nieh, “KVM/ARM: The design and implementation of the linux ARM hypervisor,” in Proc. 19th Int. Conf. Architectural Support Program. Lang. Oper. Syst., 2014, pp. 333–348.

[5] B. Sá, J. Martins and S. Pinto, "A First Look at RISC-V Virtualization From an Embedded Systems Perspective," in IEEE Transactions on Computers, vol. 71, no. 9, pp. 2177-2190, 1 Sept. 2022, doi: 10.1109/TC.2021.3124320.

[6] A. Waterman, K. Asanović, and J. Hauser, SiFive Inc., CS Division, EECS Department of California, Berkely “The RISC-V instruction set manual volume II: Privileged architecture, document version 20211203,” RISC-V Foundation, 2021.

[7] K. Asanović and D. A. Patterson, “Instruction sets should be free: The case for RISC-V,” Univ. California, Berkeley, Berkeley, CA, Tech. Rep. UCB/EECS-2014–146, 2014.

[8] G. Klein et al., “SeL4: Formal verification of an OS kernel,” in Proc. SIGOPS 22nd Symp. Oper. Syst. Princ., 2009, pp. 207–220.

[9] A. Waterman, K. Asanović, and J. Hauser, SiFive Inc., CS Division, EECS Department of California, Berkely “The RISC-V Instruction Set Manual Volume I: Unprivileged ISA, document version 20191213,” RISC-V Foundation, 2019.

[10] Arm Ltd., “Arm Architecture Reference Manual Supplement for A-profile architecture,” 2023. [Online]. Available: https://developer.arm.com/documentation/ddi0487/latest/

[11] C. Dall and J. Nieh, “KVM/ARM: The design and implementation of the linux ARM hypervisor,” in Proc. 19th Int. Conf. Architectural Support Program. Lang. Oper. Syst., 2014, pp. 333–348.

[12] Arm Ltd., “Isolation using virtualization in the Secure world Secure world software architecture on Armv8.4,” 2018. [Online]. Available: https://armkeil.blob.core.windows.net/developer/Files/pdf/Isolation%20using%20virtualization%20in%20the%20Secure%20World_Whitepaper.pdf
[13] G. Klein et al., “SeL4: Formal verification of an OS kernel,” in Proc. SIGOPS 22nd Symp. Oper. Syst. Princ., 2009, pp. 207–220.

[14] Asanović K, Avizienis R, Bachrach J, et al. The Rocket Chip Generator [J]. EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, 2016.

[15] SiFive Inc. 2017. SiFive TileLink Specification. Technical Report. SiFive, Inc., https://static.dev.sifive.com/docs/tilelink/tilelink-spec-1.7-draft.pdf

[16] K. Asanovic, R. Avizienis, J. Bachrach ´ et al., "The rocket chip generator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.pdf

[17] Y. Lee, A. Waterman, H. Cook et al., "An agile approach to building risc-v microprocessors," IEEE Micro, vol. 36, no. 2, pp. 8–20, Mar 2016.

[18] The Scala Programming Language. (n.d.). Retrieved July 14, 2023, from https://www.scala-lang.org/

[19] Patrick S. Li, Adam M. Izraelevitz and Jonathan Bachrach., "Specification for the FIRRTL Language," EECS Department University of California Berkeley, Tech. Rep. UCB/EECS-2016-9, Feb 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-9.pdf

[20] Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). Operating System Concepts (10th ed.). Wiley. (Chapter 2: Operating System Structures)

[21] Tanenbaum, A. S., & Bos, H. (2014). Modern Operating Systems (4th ed.). Pearson. (Chapter 2: Operating System Structures)

[22] Love, R. (2010). Linux Kernel Development (3rd ed.). Addison-Wesley Professional.

[23] Tanenbaum, A. S., & Van Steen, M. (2006). Distributed Systems: Principles and Paradigms. Prentice Hall.

[24] Heiser, G. "The Role of the L4 Microkernel in Mobile and Embedded Systems." Proceedings of the 4th International Conference on Mobile Technology, Applications, and Systems. [Online]. Available: https://www.researchgate.net/publication/4374042_Design_of_the_L4_Microkernel_Based_Lightweight_Mobile_Middleware_for_Mobile_Phone

[25] Trustworthy Systems Team, Data61, 2021, "seL4 Reference Manual v12.1.0," Retrieved July 14, 2023, from https://sel4.systems/Info/Docs/seL4-manual-latest.pdf

[26] Heiser, G. 2020, "The seL4 Microkernel An Introduction," https://sel4.systems/About/seL4-whitepaper.pdf

[27] Smith, J. E., & Nair, R. (2005). Virtual Machines: Versatile Platforms for Systems and Processes. Morgan Kaufmann.

[28] J. Millwood, R. VanVossen, L. Elliott, "Performance Impacts from the seL4 Hypervisor", In Proceedings of the Ground Vehicle Systems Engineering and Technology Symposium (GVSETS), NDIA, Novi, MI, Aug. 13-15, 2020.

[29] R. VanVossen, J. Millwood, C. Guikema, L. Elliott, J. Roach, "The seL4 Microkernel – A Robust, Resilient, and Open-Source Foundation for Ground Vehicle Electronics Architecture", In Proceedings of the Ground Vehicle Systems Engineering and Technology Symposium (GVSETS), NDIA, Novi, MI, Aug. 13-15, 2019.

[30] T. Prins, “Containerization in Trusted Computing.” In Proceedings of the Ground Vehicle Systems Enginnering and Technology Symposium (GVSETS), NDIA, Novi, MI, Aug. 16-18 2022.

[31] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., ... & Warfield, A. (2003). Xen and the Art of Virtualization. ACM SIGOPS Operating Systems Review, 37(5), 164-177.

[32] M. Rosenblum and T. Garfinkel, "Virtual machine monitors: current technology and future trends," in Computer, vol. 38, no. 5, pp. 39-47, May 2005, doi: 10.1109/MC.2005.176.

[33] Leveson, N. (2012). Engineering a Safer World: Systems Thinking Applied to Safety. MIT Press.

[34] Khoo, T. P., & Sun, J. (2018). The Miles Before Formal Methods - A Case Study on Modeling and Analyzing a Passenger Lift System (Vol. 11232). Springer International Publishing. https://doi.org/10.1007/978-3-030-02450-5_4

[35] Huth, M., & Ryan, M. (2004). Logic in Computer Science: Modelling and Reasoning about Systems. Cambridge University Press.

[36] Heitmeyer, C. (2018). “Formal Methods for Modeling and Analyzing Requirements: A Survey.” IEEE Transactions on Software Engineering, 44(6), 579-607. Available: https://ieeexplore.ieee.org/xpl/RecentIssue.jsp?punumber=32

[37] Xilinx ZynqMP Ultrascale+. (n.d.). Retrieved July 14, 2023, from https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html

[38] OpenSBI. (n.d.). Retrieved July 14, 2023, from https://github.com/riscv-software-src/opensbi

[39] Vivado. (n.d.). Retrieved July 14, 2023, from https://www.xilinx.com/support/university/vivado.html

[40] seL4-CAmkES-L4v-dockerfiles. (n.d.). Retrieved July 14, 2023, from https://github.com/seL4/seL4-CAmkES-L4v-dockerfiles

[41] sel4test. (n.d.). Retrieved July 14, 2023, from https://github.com/seL4/sel4test

[42] sel4_riscv_vmm. (n.d.). Retrieved July 14, 2023, from https://github.com/SEL4PROJ/sel4_riscv_vmm

[43] buildroot. (n.d.). Retrieved July 14, 2023, from https://github.com/buildroot/buildroot

[44] chipyard. (n.d.). Retrieved July 14, 2023, from https://github.com/ucb-bar/chipyard.git

[45] bao-demos. (n.d.). Retrieved July 14, 2023, from https://github.com/bao-project/bao-demos/tree/rocket