1. Introduction

In the world of computer architecture, the RISC-V (Reduced Instruction Set Computer - Five) architecture has emerged as a ground breaking open standard that offers both flexibility and simplicity in processor design. RISC-V's open-source nature has spurred innovation and democratized access to cutting-edge hardware development, making it an exciting area of study and experimentation for engineers and researchers alike.

Creating a custom RISC-V emulator provides a unique opportunity to gain in-depth understanding of how RISC-V processors operate at a fundamental level. Emulation is a powerful tool that allows developers to simulate the behaviour of a RISC-V CPU in a controlled software environment, facilitating testing, debugging, and development without needing actual hardware.

In this project, we will delve into the process of developing a custom RISC-V emulator using C and Python. The combination of C and Python leverages the strengths of both languages: C for its performance and low-level hardware interaction, and Python for its simplicity and ease of integration. By building an emulator from scratch, you will gain hands-on experience with CPU architecture, instruction sets, and the intricacies of processor emulation.

Our approach will be structured as follows:

1. **Understanding RISC-V Architecture**: We will begin by exploring the core concepts of the RISC-V instruction set architecture (ISA), including its register set, instruction formats, and execution model.
2. **Designing the Emulator**: The architecture of our emulator will be outlined, focusing on the key components such as the fetch-decode-execute cycle, instruction handling, and state management.
3. **Implementation in C**: We will implement the core of the emulator in C, focusing on performance-critical components and direct manipulation of hardware-like operations.
4. **Integration with Python**: Python will be used for higher-level functionalities, including user interfaces, testing frameworks, and scripting capabilities.
5. **Testing and Validation**: Comprehensive testing strategies will be employed to ensure the accuracy and reliability of the emulator.

By the end of this project, you will have a robust RISC-V emulator capable of running RISC-V programs, providing valuable insights into processor design and software development. Whether you are a student of computer architecture, a software engineer, or a hobbyist, this endeavour will enhance your understanding of both hardware and software systems in the context of modern computing.

1.1 Background

A] RISC-V Overview:

- **History and evolution of RISC-V**

ISA that could be used by anyone with minimal restriction. The development of the RISC-V standard began in 2010 when researchers at the University of California, Berkeley created a simple, yet powerful.

It was released in 2015 as a free and open-standard ISA that allows anyone to design, manufacture and sell processors based on the RISC-V specification — **without royalties or license fees**. Now managed by the RISC-V Foundation.

- **Comparison with other ISAs (e.g., x86, ARM)**

1] ISA Comparison

ISAs can be broadly categorized into two types: Open ISA and Closed ISA

Closed ISAs, like ARM, are proprietary and tightly controlled by specific companies (Arm Holdings here), offering established reliability and compatibility but limiting customization.

Open ISAs, exemplified by RISC-V, are community-driven and provide greater flexibility for customization, fostering innovation and adaptation to specific needs.

2]Performance

In the performance comparison between RISC-V and ARM, ARM's consistent iteration, comprehensive ecosystem, and wide range of options give it a notable performance advantage.

However, RISC-V's modular nature and customization potential hold promise for specific use cases. The ongoing efforts of RISC-V proponents to narrow the performance gap will be a crucial factor in determining how well RISC-V can match ARM's established performance standards in the future.

3] Power Efficiency

ARM's refined power management techniques and specialized cores give it a major advantage

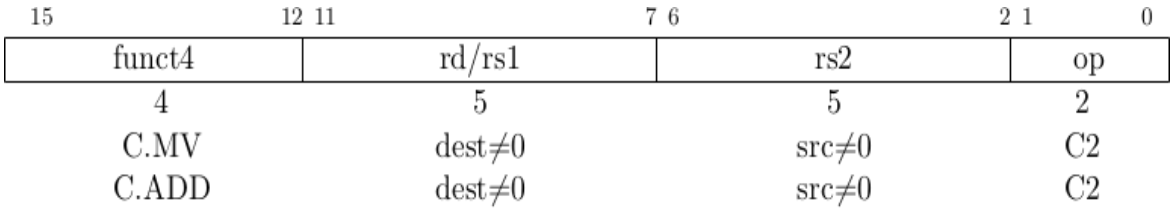
in power efficiency.

While RISC-V holds promise due to its customization potential, its open nature requires a more extensive investment of time and resources to fully harness its energy-saving capabilities.

4] Compressed Instruction set

Compared to ARM's Thumb instruction set, RISC-V also supports a compressed instruction set extension called RV32C (or RV64C for 64-bit), which provides 16-bit compressed instructions that can be mixed with the standard 32-bit instructions.

This feature helps reduce code size and improve energy efficiency, making RISC-V particularly suitable for embedded systems and low-power applications.



5]Modularity & Extensibility

One of the defining characteristics of RISC-V is its modularity and extensibility.

The ISA is designed to be easily extended with custom instructions and coprocessors, allowing for tailored implementations that meet specific application requirements.

This flexibility is achieved through a modular design, where the base ISA can be combined with optional standard extensions, such as the M extension for integer multiplication and division, the A extension for atomic operations, and the F and D extensions for single- and double-precision floating-point arithmetic.

Name	Extension
M	Integer Multiply/Divide
A	Atomic Instructions
F	Single-precision FP
D	Double-precision FP
G	General-purpose (= IMAFD)
Q	Quad-precision FP
C	Compressed Instructions

Standard extensions in RISCV: Standard RISC encoding in a fixed 32- bit instruction format.The "C" extension (compressed extension) offers shorter 16-bit versions of common 32-bit

RISC-V instructions (can be intermixed with 32-bit instructions).

- **Advantages**
    1. **Flexibility:** RISC-V offers a unique set of features that allow users to customize and optimize both software and hardware for specific use cases, resulting in faster development cycles and better design tradeoffs for performance, power and area.
    2. **Control:** This open ISA provides designers and developers with greater control over their computing environments, allowing them to fine-tune their systems without relying on third parties or incurring additional license fees associated with proprietary architectures.
    3. **Visibility:** The open-standard nature of RISC-V also means that developers have more visibility into the codebase, making it easier to understand the roadmap and identify potential security risks before they become an issue.
- **Project Motivation:**

    1. **Exploration of New Architectures**: RISC-V is an open standard, which means it offers a playground for exploring new ideas in processor design and optimization. Building an emulator can help you experiment with different architectural features and enhancements.
    2. **Prototyping**: If you're working on new ideas for RISC-V-based processors, an emulator can serve as a testing ground before moving to hardware implementation.

1.2 Objective

1]Simulate the RISC-V ISA (Instruction Set Architecture):

- **Objective**: Develop an emulator that accurately mimics the RISC-V instruction set, enabling users to execute and test RISC-V programs in a virtual environment.
- **Details**: This involves implementing the RISC-V ISA specifications, including various instruction formats, addressing modes, and exception handling.

2] Support a Subset of RISC-V Instructions:

- **Objective**: Focus on supporting a selected subset of the RISC-V instruction set to ensure the emulator can effectively handle the core functionality needed for development and testing.

- **Details**: Prioritize the most commonly used instructions and features, such as integer operations, control flow, and basic memory operations. Expand support as needed based on user requirements and project scope.

3] Enable Debugging and Performance Analysis of RISC-V Programs:

- **Objective**: Provide tools and features within the emulator for debugging and analyzing the performance of RISC-V programs.
- **Details**: Incorporate functionalities such as step-by-step execution, breakpoints, register and memory inspection, performance profiling, and tracing. This will help developers identify issues and optimize their code effectively.

2. Literature Review

For a long time, the Instruction Set Architecture (ISA) has been the firm contract between software and hardware. This firm contract plays an important role by decoupling the development of software from hardware micro-architectural features, enabling both to evolve independently. Nonetheless, it also condemns the ISA to become larger, more cluttered and inefficient as new instructions are incorporated over the years and deprecated instructions are left untouched to keep legacy compatibility. In this work we propose Open ISA, a flexible ISA that enables both the software and the hardware to evolve independently and discuss how Open ISA 1.0 was designed to enable efficient Open ISA software emulation on alien ISAs, which is key to free the user from hardware lock-ins. Our results show that software compiled to Open ISA can be latter emulated on x86 and ARM processors with very little overhead achieving near native performance, under 10% for the majority

of programs. [1]

IA-32 execution layer (IA-32 EL) is a new technology that executes IA-32 applications on Intel Itanium processor family systems. Currently, support for IA-32 applications on Itanium-based platforms is achieved using hardware circuitry on the Itanium processors. This capability will be enhanced with IA-32 EL - software that will ship with Itanium-based operating systems and will convert IA-32 instructions into Itanium instructions via dynamic translation. In this paper, we describe aspects of the IA-32 execution layer technology, including the general two-phase translation architecture and the usage of a single translator for multiple operating systems. The paper provides details of some of the technical challenges such as precise exception, emulation of FP, MMX, and Intel streaming SIMD extension instructions, and misalignment handling. Finally, the paper presents some performance results. [2]

We present the internals of QEMU, a fast machine emulator using an original portable dynamic translator. It emulates several CPUs (x86, PowerPC, ARM and Sparc) on several hosts (x86, PowerPC, ARM, Sparc, Alpha and MIPS). QEMU supports full system emulation in which a complete and

unmodified operating system is run in a virtual machine and Linux user mode emulation where a Linux process compiled for one target CPU can be run on another CPU.[3]

3. System Design

3.1Architecture

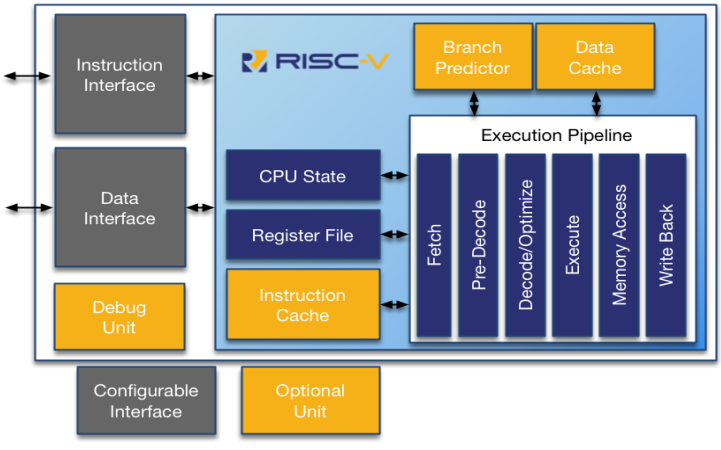
A] About RISCV

Utilize STM32F407VGt6 microcontroller as the central processing unit. Connect temperature sensor, speed sensor, and NO2 sensor to the STM32 board for data acquisition. Utilize the on board accelero meter of the STM32 board to capture x, y, z axis data. Ensure proper connectivity and configuration of all hardware components for seamless data acquisition.

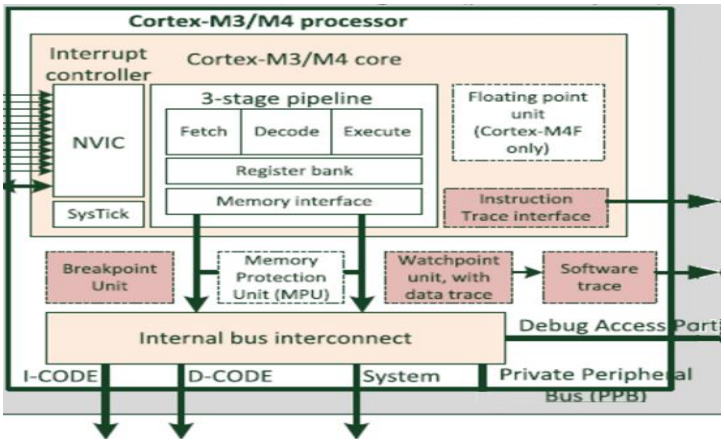
RISC-V has grown rapidly due to its flexibility and cost-saving benefits because it can be an ideal way to customize computing environments without the high costs of proprietary ISA's.

These advantages are quickly being recognized across industries, and businesses have been taking advantage of RISC-V cores for all kinds of applications including

1. Artificial intelligence (AI) image sensors,
2. Security management,
3. AI computing,
4. Machine control systems for 5G networks, and
5. More sophisticated storage, graphics and machine learning applications.



(a)



(b)

Fig 3.1: RISCV Architecture (fig a) and ARM Architecture (fig b)

The RISC-V architecture is based on the RISC principles (as compared to CISC), which emphasize a small, simple, and efficient instruction set. The key architectural features of RISC-V include

- load-store architecture
- fixed-length 32-bit instruction format
- small number of general-purpose registers
- RISC-V supports various integer instruction set extensions, such as RV32I (32-bit), RV64I (64-bit), and RV128I (128-bit), which define the base integer instruction set for different address space sizes.
- RISC-V utilizes little-endian byte ordering within the memory system, implying that the smallest significant byte of multi-byte data is stored at the lowest memory address.

1] RISC V Modes: Privilege levels & Virtual Memory

- The RISC-V Privileged Architecture Specification defines three privilege levels:
    1. machine mode (M-mode),
    2. supervisor mode (S-mode),
    3. user mode (U-mode).
- These privilege levels provide a mechanism for isolating the operating system kernel, hypervisors, and user applications, ensuring system security and stability.
- RISC-V also supports a virtual memory system based on a multi-level page table scheme, enabling efficient memory management and protection.
- Machine mode is the highest privileged mode and the only required mode
    - Flexibility allows for a range of targeted implementations from simple MCUs to high-performance Application Processors
    - Example for Simple bare metal application Machine mode is enough and it’s the default and mandatory mode, for an isolation boundary between the application and more direct hardware access, M and U mode may both be supported.
    - A robust system, such as a server or desktop machine will support M, S, and U as the Supervisor mode will bring the benefits of Virtualization and Hypervisor called Hypervisor-extended Supervisor (HS)
- Machine, Hypervisor, Supervisor modes each have Control and Status Registers (CSRs)

RISC-V Modes

Level	Name	Abbr.
0	User/Application	U
1	Supervisor	S
2	Hypervisor	HS
3	Machine	M

RISC-V Modes		
Level	Name	Abbr.
0	User/Application	U
1	Supervisor	S
2	Hypervisor	HS
3	Machine	M

Table 3.1 RISCV Mode

2] Integration and Data Flow

Data Flow:

Instructions are fetched from memory and sent to the Instruction Decoder.The Instruction Decoder interprets the instructions and passes them to the Execution Unit. The Execution Unit performs the specified operations, accessing and updating the Register File and interacting with the Memory Management component as needed. I/O Handling components facilitate interactions between the emulator and external environments or simulated devices.

Control Flow:

The Control Unit within the CPU Model directs the flow of instructions and coordinates between different components. It manages the progression of the program counter, handles branching and control flow instructions, and ensures that the emulator's state is updated correctly throughout execution.This architecture outlines the core components and their interactions in a RISC-V emulator, providing a framework for simulating a RISC-V processor and executing programs accurately.

B]Overall Design:

1. **CPU Model**:
    - **Function**: Represents the simulated RISC-V processor, encompassing its core functionalities and state.
    - **Details**: The CPU model includes:
        - **Registers**: A set of general-purpose and special-purpose registers (e.g., program counter, status registers) that hold intermediate values and control information.
        - **Program Counter (PC)**: Keeps track of the address of the next instruction to be executed.

- **Control Unit**: Directs the operation of the CPU model by managing the flow of instructions and handling control flow changes.

2. **Instruction Decoder**:
   - **Function**: Interprets the fetched instructions and translates them into a format suitable for execution.
   - **Details**:
     - **Instruction Fetch**: Retrieves instructions from memory based on the current program counter.
     - **Instruction Decoding**: Breaks down the fetched instruction into its opcode, operands, and immediate values. Identifies the type of instruction (e.g., arithmetic, branch) and prepares it for the execution unit.
     - **Instruction Queue**: May include a queue to hold instructions that are waiting to be decoded or executed.
3. **Execution Unit**:
   - **Function**: Performs the actual computations and operations specified by the decoded instructions.
   - **Details**:
     - **Arithmetic and Logic Unit (ALU)**: Executes arithmetic operations (e.g., addition, subtraction) and logic operations (e.g., AND, OR).
     - **Load/Store Unit (LSU)**: Handles memory operations, such as loading data from and storing data to memory.
     - **Branch Unit**: Manages branch instructions and alters the program counter as needed to handle control flow changes.
     - **Floating-Point Unit (Optional)**: If the emulator supports floating-point instructions, this unit performs floating-point operations.
4. **Memory Management**:
   - **Function**: Manages access to the emulator's memory, including handling address translation and memory protection.
   - **Details**:
     - **Memory Interface**: Provides the interface between the execution units and the simulated memory space.
     - **Address Translation**: If virtual memory is supported, translates virtual addresses to physical addresses.

        - **Memory Access**: Handles read and write operations, ensuring data consistency and managing memory access permissions.

5. **I/O Handling**:
    - **Function**: Manages input and output operations for the emulator, allowing interaction with external environments.
    - **Details**:
        - **Input/Output Ports**: Provides mechanisms for simulating I/O operations, such as reading from or writing to simulated devices or external interfaces.
        - **Device Simulation**: May include models for various peripherals or devices that interact with the RISC-V processor, such as timers, serial ports, or displays.
        - **Interrupt Handling**: Manages interrupts and exceptions, ensuring that the emulator correctly responds to and processes these events.

3.2Instruction Set Support

A] Instruction Set Coverage:

When developing a RISC-V emulator, the specific instruction set you support will dictate the functionality and scope of the emulator. Here's a detailed breakdown of the various RISC-V instruction sets that an emulator might include, from the base integer instructions to optional extensions:

1. RV32I Base Integer Instructions

This is the core set of instructions for RISC-V 32-bit integer operations. It includes:

- **Arithmetic Instructions**:
    - ADD (addition)
    - SUB (subtraction)
    - MUL (multiplication)
    - DIV (division)
    - REM (remainder)
- **Logic Instructions**:
    - AND (bitwise and)
    - OR (bitwise or)

  - XOR (bitwise xor)
  - SLL (shift left logical)
  - SRL (shift right logical)
  - SRA (shift right arithmetic)
- **Comparison Instructions**:
  - SLT (set less than)
  - SLTU (set less than unsigned)
- **Immediate Instructions**:
  - ADDI (add immediate)
  - ANDI (and immediate)
  - ORI (or immediate)
  - XORI (xor immediate)
- **Load/Store Instructions**:
  - LB (load byte)
  - LH (load halfword)
  - LW (load word)
  - SB (store byte)
  - SH (store halfword)
  - SW (store word)
- **Control Flow Instructions**:
  - BEQ (branch if equal)
  - BNE (branch if not equal)
  - BLT (branch if less than)
  - BGE (branch if greater than or equal)
  - JAL (jump and link)
  - JALR (jump and link register)
  - AUIPC (add upper immediate to PC)

2. RV64I (64-bit Extension)

This extends RV32I to support 64-bit operations and instructions:

- **64-bit Integer Instructions**:
  - All RV32I instructions adapted for 64-bit operations.
  - ADDW, SUBW (32-bit operations with 64-bit support)

- LWU (load word with unsigned extension)
- LD (load doubleword)
- SD (store doubleword)

3. RV32M (Multiplication and Division Extension)

Adds support for efficient multiplication and division operations:

- **Multiplication Instructions**:
  - MUL (multiplication)
  - MULH (high half multiplication)
  - MULHSU (high half multiplication with unsigned)
  - MULHU (high half multiplication with unsigned)
- **Division Instructions**:
  - DIV (division)
  - DIVU (unsigned division)
  - REM (remainder)
  - REMU (unsigned remainder)

4. RV32A (Atomic Operations Extension)

Provides support for atomic operations, crucial for multi-threaded programming:

- **Atomic Instructions**:
  - LR.W (load reserved)
  - SC.W (store conditional)
  - AMOSWAP.W (atomic swap)
  - AMOADD.W (atomic add)
  - AMOXOR.W (atomic xor)
  - AMOAND.W (atomic and)
  - AMOOR.W (atomic or)
  - AMOMIN.W (atomic min)
  - AMOMAX.W (atomic max)
  - AMOMINU.W (atomic min unsigned)
  - AMOMAXU.W (atomic max unsigned)

5. RV32F (Single-Precision Floating-Point Extension)

Adds single-precision floating-point arithmetic support:

- **Single-Precision Floating-Point Instructions**:
  - FADD.S (float add)
  - FSUB.S (float subtract)
  - FMUL.S (float multiply)
  - FDIV.S (float divide)
  - FSQRT.S (float square root)
  - FSGNJ.S (float sign inject)
  - FMIN.S (float min)
  - FMAX.S (float max)

6. RV32D (Double-Precision Floating-Point Extension)

Extends RV32F to support double-precision floating-point arithmetic:

- **Double-Precision Floating-Point Instructions**:
  - FADD.D (double-precision float add)
  - FSUB.D (double-precision float subtract)
  - FMUL.D (double-precision float multiply)
  - FDIV.D (double-precision float divide)
  - FSQRT.D (double-precision float square root)

7. RV32C (Compressed Instructions Extension)

Provides a set of 16-bit compressed instructions to reduce code size:

- **Compressed Instructions**:
  - C.ADDI (compressed add immediate)
  - C.LW (compressed load word)
  - C.SW (compressed store word)
  - C.BEQZ (compressed branch if equal to zero)
  - C.JAL (compressed jump and link)

8. RV32B (Bit-Manipulation Extension)

Adds instructions for efficient bit manipulation:

- **Bit-Manipulation Instructions**:
  - ROL (rotate left)
  - ROR (rotate right)
  - BSET (bit set)
  - BCLR (bit clear)
  - BINV (bit invert)
  - BEXT (bit extract)

9. RV64M, RV64A, RV64F, RV64D (64-bit Extensions)

These extensions are the 64-bit versions of their 32-bit counterparts, supporting 64-bit integers, floating-point operations, atomic operations, and compressed instructions.

10. RV64C (Compressed Instructions for 64-bit)

Extends RV32C to support compressed instructions in the 64-bit architecture.

11. RV64B (Bit-Manipulation for 64-bit)

Supports bit manipulation instructions for 64-bit architecture.Implementing these instruction sets and extensions in a RISC-V emulator allows for comprehensive simulation and testing of various RISC-V applications and helps ensure compatibility with a wide range of RISC-V hardware and software. The choice of which extensions to support depends on the goals and requirements of your emulator project.

3.3 Development Tools and Environment

1] Programming Languages: The languages used (e.g., C++, Python).

2] Development Environment:

a. RISC-V Toolchain:

GCC Toolchain: The RISC-V GCC toolchain includes a cross-compiler that generates

code for RISC-V. You can obtain it from RISC-V GitHub repositories or install pre-built binaries from RISC-V's official site.

Clang/LLVM: An alternative to GCC, available from LLVM's official site.

b. Emulator/Simulator: Emulation refers to the ability of one system (the emulator) to mimic another (the target), allowing software designed for the target to run on the emulator seamlessly. In the context of RISC-V, emulators enable the execution of RISC-V binaries on architectures such as x86 or ARM.

c. Debugger:

GDB: The GNU Debugger, often used with the RISC-V GCC toolchain. Ensure you use the RISC-V version for compatibility.

3] Types of Emulation

- **Full-system Emulation**

Full-system emulation replicates an entire hardware environment, enabling an OS and applications to run as if on actual hardware. This approach leverages virtualization to create a sandboxed environment.

- **User-mode Emulation**

Focused on running specific user-level applications, user-mode emulators provide performance benefits by avoiding the overhead of replicating a full system and allowing more targeted testing.

4] Techniques for Emulation

Emulators can be built using various techniques, including binary translation (converting machine code from one ISA to another), interpretation (executing instructions one at a time), and hybrid methods that combine both strategies.

5] Overview of Existing RISC-V Emulators

There are several types of emulators and simulators for RISC-V, catering to different needs such as software development, hardware design, and education. Here are some notable ones:

5.1 QEMU

Description: QEMU is a popular open-source emulator that supports RISC-V among many other architectures. It is used for running operating systems and applications in a virtualized environment.

Use Cases: Software development, testing, and debugging.

5.2. Spike

Description: Spike is the RISC-V instruction set simulator and is often used as a reference model. It's a simple, cycle-accurate simulator for running RISC-V programs.

Use Cases: Verification, validation, and software development.

5.3. OVP (Open Virtual Platforms)

Description: OVP provides fast, cycle-accurate models of RISC-V processors and systems. It is useful for system-level simulation and hardware/software co-design.

Use Cases: System design, performance analysis, and software development.

5.4. RISC-V Proxy Kernel (PK)

Description: This is a minimal operating system kernel provided for testing and developing RISC-V applications. It helps in simulating the operating environment.

Use Cases: Application testing, OS development, and research.

5.5. RISC-V Simulator (RISCV-SIM)

Description: A lightweight simulator that focuses on RISC-V instruction execution. It provides basic functionality for executing RISC-V binaries.

Use Cases: Basic execution and educational purposes.

5.6. Renode

Description: Renode is a simulation framework that allows for modeling and simulating entire embedded systems. It supports RISC-V and is used for system-level testing and development.

Use Cases: System-level testing, embedded systems development, and hardware/software co-simulation.

5.7. Gem5

Description: Gem5 is a modular and flexible simulator that supports a wide range of architectures, including RISC-V. It can simulate full systems and processors.

Use Cases: Computer architecture research, system performance evaluation, and detailed simulation.

5.8. RISC-V Virtual Platform

Description: Various virtual platforms, often provided by specific vendors, offer detailed and customizable RISC-V processor and system simulations.

Use Cases: Hardware design, verification, and software development.

5.9. FESVR (Frontend Server for RISC-V)

Description: FESVR is often used in conjunction with other simulators like Spike or QEMU to provide additional features and interfaces for RISC-V.

Use Cases: Enhancing simulation capabilities, interfacing with other tools.

Each emulator or simulator serves different purposes and comes with its own set of features, so the choice of which to use often depends on the specific requirements of your project or research.

4. Implementation Details

4.1 Creating a file system for Building a linux distribution

a) The Role of a FileSystem:

- The data is organized and can be easily located.
- The data can be easily retrieved at any later point in time.
- The integrity of the data is preserved.
- Data can be categorized under different labels.

b) The Linux File System

- The *Nix (Unix or Linux) file system is a hierarchical directory structure
- The structure resembles an upside down tree
- Directories are collections of files and other directories. The structure is recursive with many levels.
- Every directory has a parent except for the root directory. Many directories have sub-directories.
- Unlike Windows, with multiple drives and multiple file systems, the Linux system only has ONE unified file system.
- The Linux Standard Base (LSB) specifies the structure of a Linux file system.
- Supports 256-character filenames.
- All command line entries are case sensitive.
- Use the slash(/) rather than the backslash(\) that we use in Windows.

c) Various directories in Linux FS

- /bin, /sbin : executable files necessary to manage and run the Linux system.
- /boot : bootloader files, required to boot the system
- /dev : special files used to represent the various hardware devices in the system. E,g, hda, hdb ...
- /etc : text-based configuration files used by the system as well as services running on the system.
- /home : subdirectories that serve as home directories for each user account.

- /lib : code libraries used by programs in /bin and kernel modules (stored in the modules subdirectory)
- /mnt , /media : used by some Linux distributions (such as Fedora or Red Hat) to mount external devices.
- /root : root user's home directory. Notice that it is located separately from the home directories for other users in /home.
- /srv : subdirectories where services running on the system (such as httpd and ftpd) save their files
- /sys : information about the system hardware.
- /usr : application files used on the system are saved in various subdirectories.
- /var : variety of variable data, including the system log files

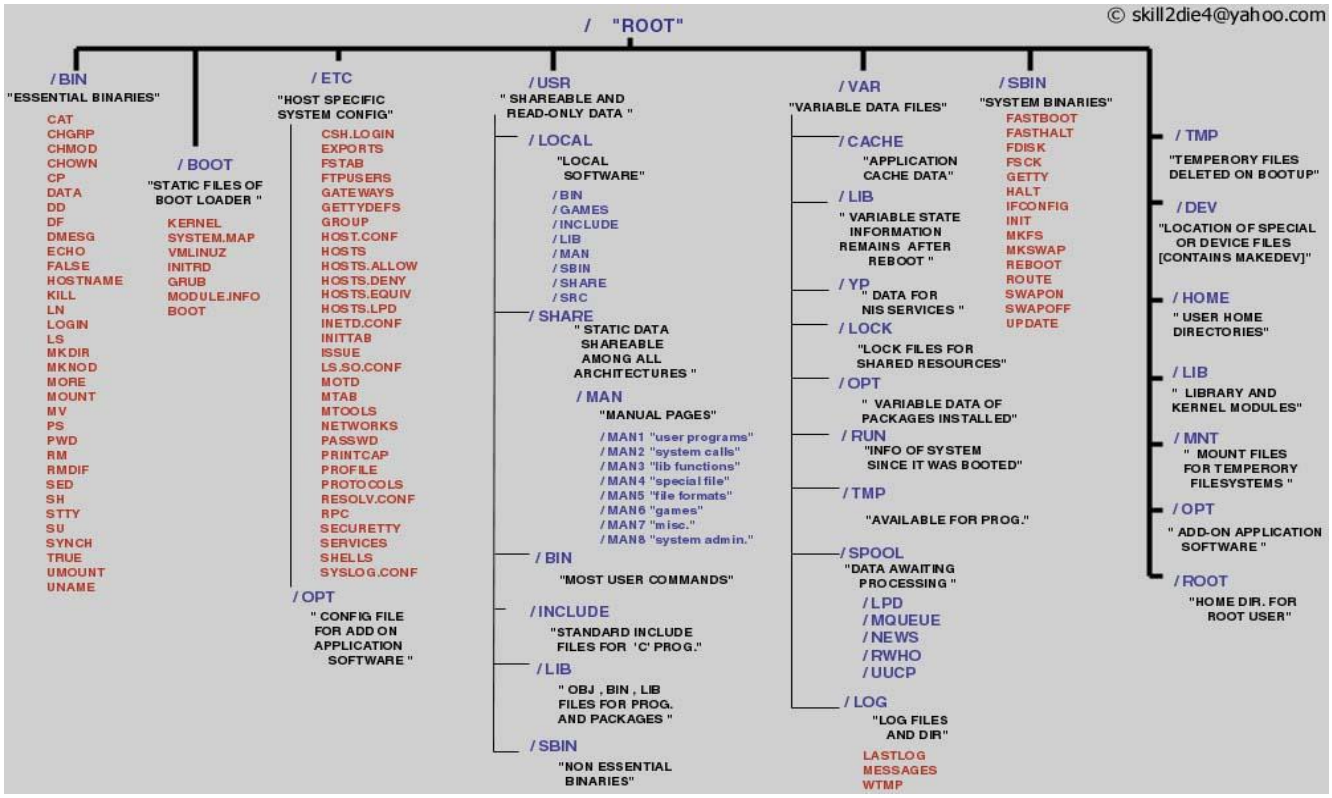


Fig 4.1: Typical linux file system view

d) Types of file system

- Linux supports many (almost 100) different file system types - most common choices - the ext family (such as ext2, ext3 and **ext4**), XFS, ZFS, JFS and btrfs.(Please note that Windows supports only two types of File Systems – FAT for the very old systems and NTFS)
- The ext4 or fourth extended file system is the default file system for many popular Linux distributions.

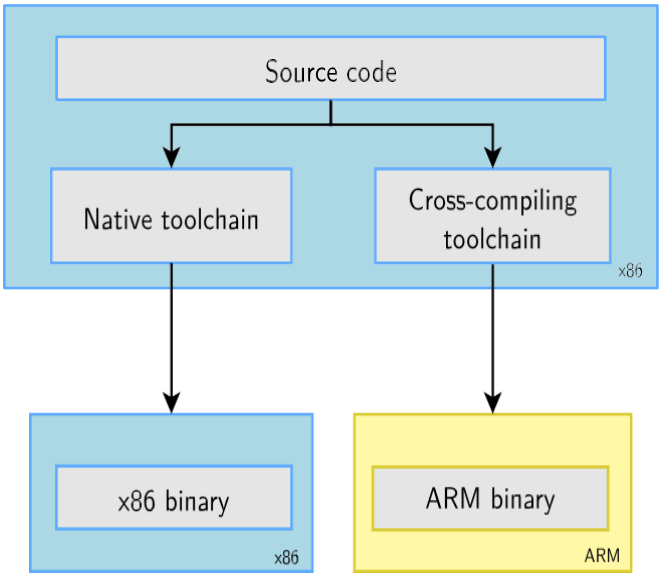
- It has the journaling feature - A journaling file system logs changes to a file (usually a circular log in a dedicated area) before committing them to the main file system. Such file systems are less likely to become corrupted in the event of power failure or system crash.

e) Links

- Multiple names can point to same inode
- The inode keeps track of how many links
- If a file gets deleted, the inode's link-count gets decremented by the kernel
- File is deallocated if link-count reaches 0
- This type of linkage is called a 'hard' link
- Hard links may exist only within a single FS
- Hard links cannot point to directories (cycles)

4.2 Building a cross complier toolchain

- The usual development tools available on a GNU/Linux workstation is a **native toolchain**
- This toolchain runs on your workstation and generates code for your workstation, usually x86
- For embedded system development, it is usually impossible or not interesting to use a native toolchain
    - The target is too restricted in terms of storage and/or memory
    - The target is very slow compared to your workstation
    - You may not want to install all development tools on your target.
- Therefore, **cross-compiling toolchains** are generally used. They run on your work station but generate code for your target.
- Three machines must be distinguished when discussing toolchain creation
    - The **build** machine, where the toolchain is built.
    - The **host** machine, where the toolchain will be executed.
    - The **target** machine, where the binaries created by the toolchain are executed.



Execution
Machine

Fig 4.2: Toolchain

a) Components of gcc toolchains

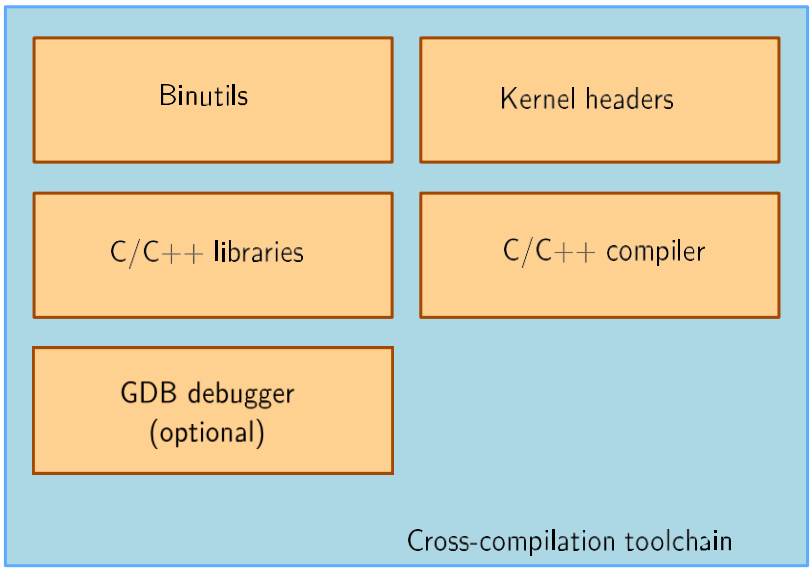


Fig 4.3: Components of toolchain

1) Binutils

Binutils is a collection of binary utilities used in software development, primarily for tasks related to the manipulation and management of binary files, such as executables and object files. It is often used in conjunction with GNU Compiler Collection (GCC) and other development tools. The suite includes several key components:

1. **as (the assembler)**: Converts assembly language code into machine code, producing object files that can be linked to create executable programs.
2. **ld (the linker)**: Links object files together to create executable programs or shared libraries. It resolves references between object files and handles relocation.
3. **objdump**: Displays information about object files, such as their structure, contents, and disassembly of machine code.

4. **readelf**: Displays detailed information about ELF (Executable and Linkable Format) files, which are commonly used for executables and shared libraries on Unix-like systems.
5. **strip**: Removes symbols and debugging information from object files and executables, reducing their size.
6. **nm**: Lists symbols from object files, allowing developers to see which symbols are defined and used.
7. **ar**: Creates, modifies, and extracts from archives (collections of object files), commonly used for static libraries.
8. **ranlib**: Generates an index to the contents of an archive, making it easier for linkers to find symbols.
9. **objcopy**: Copies and translates object files, allowing for transformations such as changing formats or stripping data.

binutils is crucial for low-level programming and systems development, enabling developers to manage and inspect the binary components of their software. It is widely used on Unix-like systems and is an essential part of many development toolchains.

2) Kernel headers

- The C library and compiled programs needs to interact with the kernel
    - Available system calls and their numbers
    - Constant definitions
    - Data structures, etc.
- Therefore, compiling the C library requires kernel headers, and many applications also require them.
- Available in <linux/...>and <asm/...>and a few other directories corresponding to the ones visible in include/uapi/and in arch/<arch>/include/uapi in the kernel sources
- The kernel headers are extracted from the kernel sources using the headers_install kernel Makefile target.
- The kernel to user space ABI is **backward compatible**
- ABI = *Application Binary Interface* - It's about binary compatibility
- Kernel developers are doing their best to **never** break existing programs when the kernel is upgraded. Otherwise, users would stick to older kernels, which would be badfor everyone.
- Hence, binaries generated with a toolchain using kernel headers older than the running kernel

will work without problem, but won't be able to use the new system calls, data structures, etc.

- Binaries generated with a toolchain using kernel headers newer than the running kernel might work only if they don't use the recent features, otherwise they will break.
- What to remember: updating your kernel shouldn't break your programs; it's usually fine to keep an old toolchain as long as it works fine for your project.

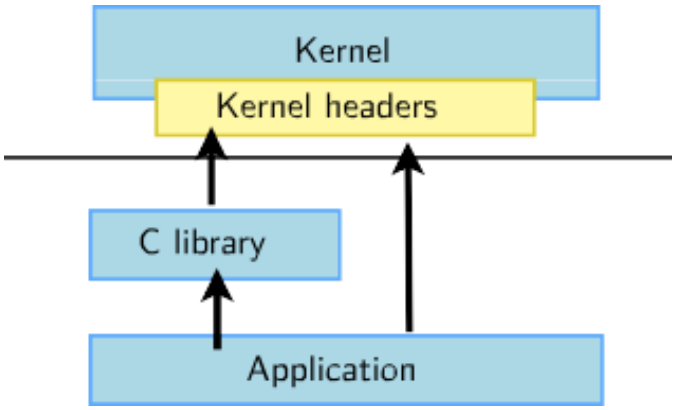


Fig 4.4 Kernel header

3) C/C++ compiler

- GCC: GNU Compiler Collection, the famous free software compiler
- https://gcc.gnu.org/
- Can compile C, C++, Ada, Fortran, Java, Objective-C, Objective-C++, Go, etc. Can generate code for a large number of CPU architectures, including x86, ARM, RISC-V, and many others.
- Available under the GPL license, libraries under the GPL with linking exception.
- Alternative: Clang / LLVM compiler (https://clang.llvm.org/) getting increasingly popular and able to compile most programs (license: MIT/BSD type). It can offer better optimizations and make errors easier to interpret.
- GCC: GNU Compiler Collection, the famous free software compiler can be downloaded from websitehttps://gcc.gnu.org/
- Can compile C, C++, Ada, Fortran, Java, Objective-C, Objective-C++, Go, etc. Can generate code for a large number of CPU architectures, including x86, ARM, RISC-V, and many others.
- Available under the GPL license, libraries under the GPL with linking exception.
- Alternative: Clang / LLVM compiler (https://clang.llvm.org/) getting increasingly popular

and able to compile most programs (license: MIT/BSD type). It can offer better optimizations and make errors easier to interpret.

4) C complier

- TheClibraryisanessentialcomponentofa Linuxsystem.
  - Interfacebetweentheapplicationsandthekernel
  - Providesthewell-knownstandardCAPItoeaseapplicationdevelopment
- SeveralClibrariesareavailable:*glibc*,*uClibc*, *musl*,*klibc*,*newlib*...
- ThechoiceoftheClibrarymustbemadeat cross-compilingtoolchaingenerationtime,as theGCCcompileriscompiledagainstaspecific.

b) glibc

- License:LGPL
- ClibraryfromtheGNUproject
- Designedforperformance,standardscomplianceand portability
- FoundonallGNU/Linuxhostsystems
- Ofcourse,activelymaintained
- By default, quite big for small embedded systems. On armv7hf, version 2.31: libc: 1.5 MB, libm: 432 KB, source:https://toolchains.bootlin.com
- But some features not needed in embedded systems can beconfiguredout(mergedfromtheold*eglibc*project).

4.3 Busybox

Busy Box is a highly versatile and efficient software suite that consolidates numerous Unix utilities into a single executable. It's designed to be lightweight and modular, making it especially suitable for embedded systems, minimalist environments, and recovery environments. Here's a more detailed look at its features, design, and usage:

a) Core Features

1. Unified Executable: Busy Box combines a wide range of command-line tools into a single binary. This design helps reduce the number of individual files on the file system and
minimizes the overall disk space required.

2. Customizable Configuration: Busy Box is highly configurable through its build system. During compilation, users can select which utilities to include and configure their options. This allows for creating a tailored Busy Box binary that only includes the necessary tools, further optimizing resource usage.

3. Multiple Utilities: It provides a wide range of utilities, including but not limited to:

- Shell Utilities: ash (Almquist shell), sh (Bourne shell)
- File Utilities: ls, cp, mv, rm, cat, touch
- Text Processing: grep, sed, awk, cut, sort, uniq
- System Utilities: ps, top, df, du, kill, chmod, chown
- Networking Tools: ifconfig, ping, wget, telnet

The exact set of available utilities depends on how Busy Box is configured during compilation.

4. Lightweight Design: Busy Box is designed to be as small as possible. This lightweight nature is achieved by reusing code and providing a single binary with many functionalities rather than individual binaries for each tool.

5. Embedded System Use: It's commonly used in embedded Linux systems, initial ramdisks (initrd), and rescue environments where saving space and minimizing dependencies is crucial.

b) Implementation Details

1. Source Code: Busy Box is written in C and is open-source. It is licensed under the GNU General Public License (GPL), which allows users to modify and redistribute the code.

2. Build System: Busy Box uses a configuration script (usually make menuconfig, make xconfig, or similar) that enables users to select which utilities and features to include. This configuration is then used to build a customized Busy Box binary.

3. Linking: The tools provided by Busy Box are implemented using the same binary but with different functionality based on the arguments provided. This approach allows Busy Boxto act as a Swiss army knife of Unix commands.

4. Compatibility: Busy Box aims to be compatible with standard Unix utilities, though it may not always provide all features found in the GNU core utilities or other full-featured implementations. It is designed to be a functional subset of these utilities.

5. Usage: Busy Box can be used interactively through its shell (ash) or executed directly from scripts. It is commonly used in system initialization scripts and as part of the root file system in embedded systems.

c) Typical Use Cases

1. Embedded Systems:Ideal for devices with limited storage and memory, such as routers, IoT devices, andappliances.

2. Initial Ramdisk (Initrd):Used in Linux boot processes to provide essential tools for the initial stages of systemstartup.

3. Recovery Environments: Employed in rescue systems for troubleshooting and system repair due to its minimalfootprint.

4. Minimalist Environments:Suitable for creating small, single-purpose Linux distributions or environments whereonly basic functionality is required.

d) BusyBox on RISC-V

1. Purpose:BusyBox provides a minimal set of Unix utilities that are critical for managing andoperating a system. When used on RISC-V systems, it delivers essential functionalitiesneeded for system initialization, configuration, and management in a compact andefficient manner.

2. Cross-Compilation:To run BusyBox on a RISC-V system, it needs to be cross-compiled for the RISC-Varchitecture. This process involves configuring and building BusyBox using a toolchainthat targets RISC-V.

3. Steps to Build BusyBox for RISC-V:

- Set Up a RISC-V Toolchain:You need a cross-compilation toolchain that targets RISC-V. Thistypically includes the RISC-V GCC compiler and other related tools. Pre-built toolchains can be downloaded, or you can build one from source.
- Obtain BusyBox Source Code: Download the source code for BusyBox from its official website orrepository.

e) Configure BusyBox:

1. Configure BusyBox using the make menuconfig command or similarconfiguration tool. Ensure that you select the appropriate settings foryour RISC-V target. You can include or exclude specific utilities based onyour needs.

f) Build BusyBox:

1. Compile BusyBox using the cross-compiler. The typical build command ismake ARCH=riscv CROSS_COMPILE=riscv64-unknown-linux-nu- (replace riscv64-unknown-linux-gnu- with your specifictoolchain prefix).

g) Deploy BusyBox:

After building, you will have a BusyBox binary that can be copied to yourRISC-V system. It can be used in the root filesystem, initramfs, ordirectly on the RISC-V device.

1. Integration: BusyBox can be used in various scenarios on a RISC-V system:
    - Root Filesystem: Integrate BusyBox into a Linux root filesystem for RISC-V.
    - Initramfs: Include BusyBox in an initramfs image for initial system boot.
    - Embedded Systems: Use BusyBox in embedded RISC-V systems where space isconstrained.

2. Customization:BusyBox is highly customizable, so you can include only the utilities you need for yourRISC-V system. This is particularly useful in embedded systems where minimizingfootprint and resource usage is crucial.

3. Benefits of Using BusyBox on RISC-V

1. Space Efficiency:BusyBox helps reduce the overall size of the root filesystem, which is beneficial forsystems with limited storage.

2. Simplicity:Provides a single, unified interface for multiple Unix utilities, simplifying systemmanagement.

3. Flexibility:The ability to customize which utilities are included allows for a tailored system thatmeets specific needs without unnecessary overhead.

4. Resource Management:Helps manage resources efficiently, which is crucial in embedded and minimalisticenvironments.

5. Example Use Case: Embedded Linux on RISC-V: For an embedded Linux system running on a RISC-V processor,BusyBox can serve as the core set of utilities for basic system operations, initialization scripts,and maintenance tasks. It provides essential command-line tools while keeping the systemlightweight.In summary, using BusyBox on RISC-V systems involves cross-compiling the software to matchthe architecture and integrating it into your system environment. This setup is ideal formaintaining aminimalistic and efficient operating system on RISC-V hardware.

4.4 Compiling linux for RISCV

Compiling the Linux kernel for the RISC-V architecture involves several steps, including configuring the build environment, configuring the kernel, and building the kernel. Here's a step-by-step guide to get you started:

A) Prerequisite:

1. **Install Required Packages:** Ensure you have the necessary development tools and libraries installed. On a Debian-based system, you can use:

   bash
   Copy code
   sudo apt-get update
   sudo apt-get install build-essential bison flex libssl-dev libncurses5-dev libncursesw5-dev

2. **Install RISC-V Toolchain:** You need a RISC-V cross-compiler. You can download pre-built toolchains or build one from source. To install a pre-built toolchain:

   bash
   Copy code
   sudo apt-get install gcc-riscv64-linux-gnu

   Alternatively, you can build the toolchain from source. Detailed instructions can be found in the RISC-V GitHub repository.

B) Download the Kernel Source

1. Get the Linux Kernel Source: You can download the latest Linux kernel source from Kernel.org or clone the repository:

bash
Copy code
git clone https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
cd linux

2. Configure the Kernel

2.1 Configure Kernel for RISC-V: There are pre-configured files for different RISC-V configurations in the kernel source tree. For a basic configuration, you can use:

bash
Copy code
make ARCH=riscvdefconfig

This will set up a default RISC-V configuration. If you want to customize the configuration, use:

bash
Copy code
make ARCH=riscvmenuconfig

This will launch a menu-based configuration tool.

2.2 Build the Kernel

a) Compile the Kernel: Use the RISC-V cross-compiler to build the kernel:

```
bash
Copy code
make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu- -j$(nproc)
```

Here, CROSS_COMPILE specifies the prefix for the cross-compiler (adjust based on your toolchain), and -j$(nproc) utilizes all available CPU cores to speed up the build process.

b) Build Modules (if needed): If your configuration includes kernel modules, build them with:

```
bash
Copy code
make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu- modules
```

2.3 Deploy the Kernel

a) Install the Kernel: After building, you need to deploy the kernel. The kernel image will typically be located in arch/riscv/boot (e.g., Image or vmlinux). You'll also need a device tree blob (DTB) and initramfs if required.

b) Run the Kernel: If you are using a RISC-V emulator or simulator like QEMU, you can run the kernel with:

```
bash
Copy code
qemu-system-riscv64 -machine virt -nographic -kernel arch/riscv/boot/Image -append "root=/dev/vdaro" -drive file=rootfs.ext4,format=raw
```

Replace rootfs.ext4 with your root filesystem image if necessary.

c) Debugging and Testing

1. **Debugging:**If you encounter issues, you can use tools like gdb and QEMU's debugging features to troubleshoot.

2. **Testing:**Make sure to test the kernel on real hardware or emulators to ensure it functions as expected.

By following these steps, you should be able to compile and run a Linux kernel for the RISC-V architecture. For more detailed information and troubleshooting, refer to the official RISC-V documentation, and the Linux kernel documentation.

4.5 Preparing RISCV Image

Preparing a RISC-V kernel image involves creating a bootable kernel that can be used with emulators or actual RISC-V hardware. Here's a detailed guide on how to prepare a RISC-V kernel image for deployment:

a) Compile the Kernel: Ensure you have followed the previous steps to compile the Linux kernel for RISC-V and build.

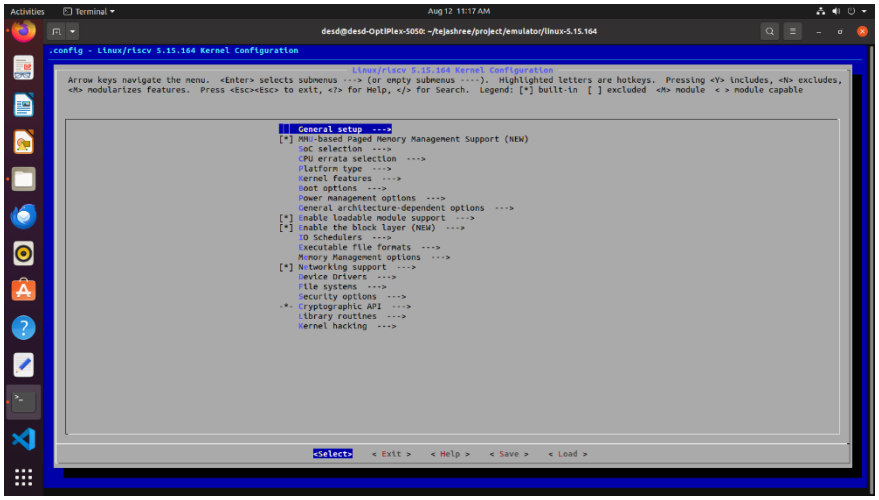


Fig 4.5 Configuration of Linux

b) Prepare the Kernel Image

After compiling, the kernel image will be located in the arch/riscv/boot directory, typically named Image or vmlinux. *Kernel Image Types*

- **Image**: This is the compressed kernel image used by the bootloader or emulator.
- **vmlinux**: This is an uncompressed kernel binary that may also be used in some scenarios.

Note: For most scenarios, Image is preferred as it's a compressed, bootable format.

c) Create a Root Filesystem

If you're testing on an emulator or real hardware, you'll need a root filesystem. You can create a

minimal root filesystem using tools like debootstrap or by using pre-built images.Replace focal with the version of Ubuntu or your desired distribution.Or download a pre-built root filesystem from resources such as riscv.org or various Linux distributions.

d) Prepare the Device Tree Blob (DTB)

The Device Tree Blob describes the hardware layout and is necessary for the kernel to boot correctly on emulators and some hardware.

- **Obtain or Generate DTB:**

  You can use the dtc tool to compile a device tree source (DTS) file into a DTB file. For most emulators, a generic DTB file is available:

  dtc -I dts -O dtb -o your-device-tree.dtb your-device-tree.dts

In many cases, pre-built DTB files are provided with the kernel source or by the hardware vendor.

- **Create an Initramfs (Optional)**

An initramfs is a filesystem that gets loaded into memory and is used as the root filesystem during boot. It's useful for including drivers and tools needed to mount the real root filesystem.

Create Initramfs:You can create an initramfs using tools like initramfs-tools or by manually assembling it:

```
mkdirinitramfs
cd initramfs
mkdir -p bin sbinetc proc sys
cp /bin/busybox bin/
ln -s bin/busybox bin/sh
```

Then, create a cpio archive:

```
find . | cpio -H newc -o > ../initramfs.cpio
```

You can compress it using gzip:

```
gzipinitramfs.cpio
```

e) Run the Kernel with an Emulator

For testing, you can use QEMU to emulate a RISC-V environment:

```
qemu-system-riscv64 -machine virt -nographic \
-kernel arch/riscv/boot/Image \
 -append "root=/dev/vdaro console=ttyS0" \
```

```
-drive file=rootfs.ext4,format=raw \
-device virtio-blk-device,drive=hd \
-drive file=initramfs.cpio.gz,format=raw,if=none,id=initramfs \
-device virtio-blk-device,drive=initramfs
```

Adjust the -drive and -device parameters according to your setup.

d) Run on Actual Hardware (Optional)

For real hardware, you'll need to follow the specific procedures for flashing the kernel image and bootloader to the device. This usually involves:

- **Building or obtaining a bootloader** (e.g., U-Boot) that supports RISC-V.
- **Flashing the kernel image, DTB, and root filesystem** to the appropriate storage device or memory.

Refer to the documentation of your RISC-V hardware for specific instructions on how to flash and boot the kernel.

4.6 Booting image in QEMU

Booting a Linux kernel image in QEMU for RISC-V involves a series of steps to configure and run the emulator with the appropriate kernel, device tree blob (DTB), and root filesystem. Here's a comprehensive guide to help you through the process:

a) Prepare Your Environment

Make sure you have QEMU installed. If it's not installed, you can typically install it via package managers. On Debian-based systems:

```
sudo apt-get update
sudo apt-get install qemuqemu-system-misc
```

b) Prepare the Kernel Image

Ensure you have a RISC-V kernel image ready. This is typically located in the arch/riscv/boot directory of your kernel source and named Image.

c) Prepare the Device Tree Blob (DTB)

You need a Device Tree Blob (DTB) file to describe the hardware configuration. This file can often be found or generated from the kernel source. For QEMU, you might use a generic DTB file:

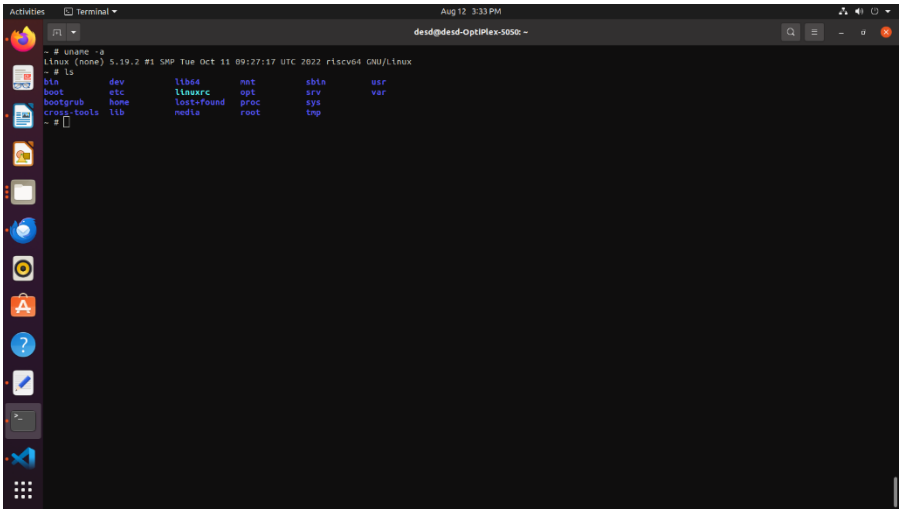
- **Example DTB File:**

  You might find a DTB file in the kernel source tree, or you can use a default one like virt.dtb for a virt machine.

d) Prepare a Root Filesystem

Create or download a root filesystem image. If you don't have one, you can create a minimal root filesystem using tools like debootstrap or obtain a pre-built one.

- **Example Root Filesystem Image:**



You can use a root filesystem image (e.g., rootfs, ext4) that is compatible with RISC-V.

Fig 4.6: RISCV Root File System

e) Prepare an Initramfs (Optional)

An initramfs is a root filesystem that gets loaded into memory at boot time. Create one if needed:

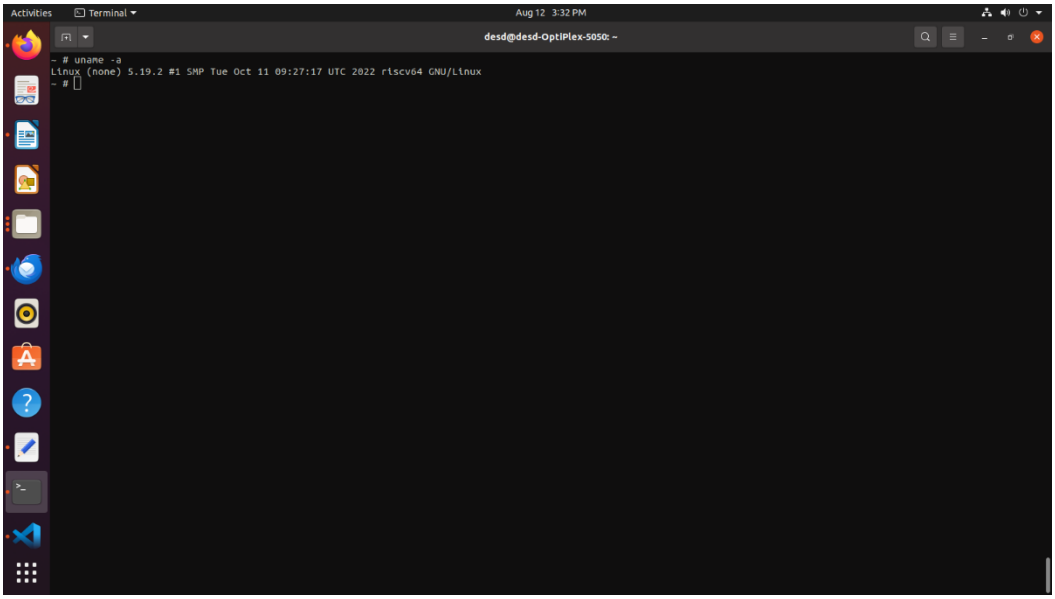
mkdirinitramfs

```
cd initramfs

mkdir -p bin sbinetc proc sys

cp /bin/busybox bin/

ln -s bin/busybox bin/sh

find . | cpio -H newc -o > ../initramfs.cpio

gzip ../initramfs.cpio
```

f) Run QEMU with RISC-V

Use QEMU to boot the kernel with the appropriate parameters. Here's a typical command for running QEMU with a RISC-V kernel:

```
qemu-system-riscv64 \

-machine virt \

-nographic \

-kernel arch/riscv/boot/Image \

-device virtio-blk-device,drive=hd \

-drive file=rootfs.ext4,format=raw \

-append "root=/dev/vdaro console=ttyS0" \

-device virtio-blk-device,drive=initramfs \

-drive file=initramfs.cpio.gz,format=raw,if=none,id=initramfs \
```



-device virtio-blk-device,drive=initramfs

Fig 4.7 Details of RISCV

Explanation of Parameters:

- -machine virt: Specifies the virtual machine type.
- -nographic: Disables graphical output; use terminal output only.
- -kernel arch/riscv/boot/Image: Specifies the kernel image.
- -device virtio-blk-device,drive=hd: Configures a virtual block device for the root filesystem.
- -drive file=rootfs.ext4,format=raw: Specifies the root filesystem image.
- -append "root=/dev/vdaro console=ttyS0": Passes kernel parameters, including specifying the root filesystem and console.
- -device virtio-blk-device,drive=initramfs: Adds a virtual block device for the initramfs.
- -drive file=initramfs.cpio.gz,format=raw,if=none,id=initramfs: Specifies the initramfs image.

g) Alternative Commands

Depending on your setup and the components you have, you might need to adjust the command. Here's an example command without initramfs:

qemu-system-riscv64 \

```
-machine virt \

-nographic \

-kernel arch/riscv/boot/Image \

-device virtio-blk-device,drive=hd \

-drive file=rootfs.ext4,format=raw \

-append "root=/dev/vdaro console=ttyS0" \

-device virtio-blk-device,drive=initramfs \

-drive file=initramfs.cpio.gz,format=raw,if=none,id=initramfs
```

h) Debugging and Monitoring

- **QEMU Console**: You'll get kernel boot messages and a login prompt in your terminal if everything is configured correctly.
- **Logs**: Check QEMU logs and console output for any errors or issues during boot.

i) **Additional Tips**

- Ensure that the kernel image and root filesystem are compatible with each other.
- Check the QEMU documentation for any additional parameters or features you might need for specific use cases.

By following these steps, you should be able to boot a RISC-V kernel image in QEMU successfully.

5. Conclusion

Developing a custom RISC-V emulator in C and Python is a complex but rewarding endeavor that offers insights into both low-level hardware operations and high-level software design. By carefully planning the architecture, implementing core components efficiently, leveraging Python for scripting and control, and rigorously testing the emulator, you can create a powerful tool for experimenting with and understanding the RISC-V architecture. Engaging with the community and maintaining thorough documentation will further enhance the emulator's utility and longevity.

- **Real Hardware:** Best for accurate performance metrics and application deployment, providing the highest execution speed.
- **QEMU:** Suitable for general development and testing, offering a compromise between speed and functionality.
- **Spike:** Ideal for debugging and verifying RISC-V implementations with high accuracy but lower speed.
- **OVP and Other Specific Emulators:** Useful for specialized scenarios, with performance varying based on their design goals.

Choosing the right tool depends on your specific needs—whether you prioritize speed, accuracy, or a balance of both. For development and early-stage testing, emulators are invaluable, but for final performance evaluation and deployment, real hardware remains essential.

6. References

1. Auler, R., Borin, E.: The case for flexible ISAs: unleashing hardware and software. In: SBAC-PAD 2017 (2017).
2. Baraz, L., et al.: IA-32 execution layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium®-based systems. In: 36th MICRO (2003).
3. Bellard, F.: QEMU, a fast and portable dynamic translator. In: USENIX, FREENIX Track(2005).