Date of presentation: 14/06/23

DUAL CORE SECURE BOOT

Boris Bukchin, Sergey Mashkin

Supervised by: Avi Mendelson

Contents

[**1. Abstract** 2](#_Toc137232607)

[**2. Introduction** 2](#_Toc137232608)

[**2.1. OpenTitan** 2](#_Toc137232609)

[2.1.1. Links 2](#_Toc137232610)

[2.1.2. About 2](#_Toc137232611)

[2.1.3. HW architecture overview 3](#_Toc137232612)

[**2.2. Ibex core** 4](#_Toc137232613)

[2.2.1. Links 4](#_Toc137232614)

[2.2.2. About 4](#_Toc137232615)

[2.2.3. HW Architecture overview 4](#_Toc137232616)

[**2.3. Secure Boot** 6](#_Toc137232617)

[2.3.1. ROM 7](#_Toc137232618)

[**2.4. Fault injection attacks** 9](#_Toc137232619)

[**2.5. Dual core secure boot** 9](#_Toc137232620)

[**2.6. Verilator & Bazel** 9](#_Toc137232621)

[**3. Hardware Modifications** 9](#_Toc137232622)

[**4. Software Modifications** 10](#_Toc137232623)

[**5. Validation** 11](#_Toc137232624)

[**6. Conclusions** 11](#_Toc137232625)

[**7. Future Research** 11](#_Toc137232626)

[**8. Appendix** 11](#_Toc137232627)

# **1. Abstract**

Modern secure multi core systems use single core to run boot sequence, while rest of the cores sleep and wait for the single core to finish boot. The boot sequence code is checked against cryptographic signature to validate it’s authenticity on dedicated hardware before executed on the core. This method may be vulnerable to fault injection attacks because the system isn’t validated after initial cryptographic validation. In this project we implement a system that boots on two cores simultaneously, both cores executing simultaneously the same code and validating each other’s outputs, in case of mismatch in the main core’s output the shadow core restoring all the main core’s registers to continue correct execution. We used OpenTitan Root of Trust source code with RISC-V ibex cores. In the project we showcasing the additions made to implement the dual core secure boot and its validation.

# **2. Introduction**

## **2.1. OpenTitan**

### 2.1.1. Links

Website: <https://opentitan.org/>

GitHub: <https://github.com/lowrisc/opentitan>

Documentation: <https://opentitan.org/book/hw/index.html#top>

Getting started guide: <https://opentitan.org/guides/getting_started/index.html#top>

### 2.1.2. About

OpenTitan is an open-source project that aims to develop a transparent, high-quality, and secure root of trust (RoT) for computer systems. It focuses on designing and implementing silicon RoT, which is responsible for providing a secure foundation for the entire system, ensuring its integrity and protecting against various security threats.

The OpenTitan project is a collaborative effort involving several industry leaders and organizations, including Google, lowRISC and others. The primary goal of OpenTitan is to create a trustworthy and verifiable RoT that can be used in a wide range of applications, including servers, data centers, storage devices, networking equipment, and more. By providing an open-source implementation, OpenTitan aims to foster transparency, accountability, and security in the design and manufacturing of silicon chips.

Key features and objectives of OpenTitan include:

1. Transparency: OpenTitan aims to make the design and implementation details of the RoT open and accessible, enabling thorough security analysis and scrutiny by the community.
2. Security: OpenTitan focuses on providing a strong foundation for system security, protecting against a wide range of attacks, including hardware and software tampering, side-channel attacks, and supply chain threats.
3. Collaboration: OpenTitan encourages collaboration between industry, academia, and the open-source community to foster innovation, share best practices, and collectively improve the security of computer systems.

### 2.1.3. HW architecture overview

OpenTitan uses a single RISC-V Ibex core connected with TL-UL Crossbar to:

1. Memory modules like ROM and flash.
2. Dedicated hardware to execute security features like keys, AES, HMAC, KMAC, etc.
3. Peripherals to clocks, power management, UART, etc.

Key units in OpenTitan:

1. Cryptographic Engines: OpenTitan incorporates cryptographic engines that perform cryptographic operations such as encryption, decryption, hashing, and digital signatures. These engines are crucial for ensuring secure communication, data integrity, and authentication.
2. True Random Number Generators (TRNGs): OpenTitan utilizes TRNGs to generate random numbers for cryptographic purposes. TRNGs rely on unpredictable physical processes to generate high-quality random data, which is essential for secure cryptographic operations.
3. Key Management Units: OpenTitan includes key management units responsible for securely generating, storing, and managing cryptographic keys. These units protect the confidentiality and integrity of cryptographic keys, ensuring they are securely used and protected throughout the system.
4. Secure Memories: OpenTitan incorporates secure memories that are resistant to various attacks, such as side-channel attacks and physical tampering. These memories store sensitive data, cryptographic keys, and security-related information.

Notice that different domains require different clock frequencies to ensure high performance while saving power.

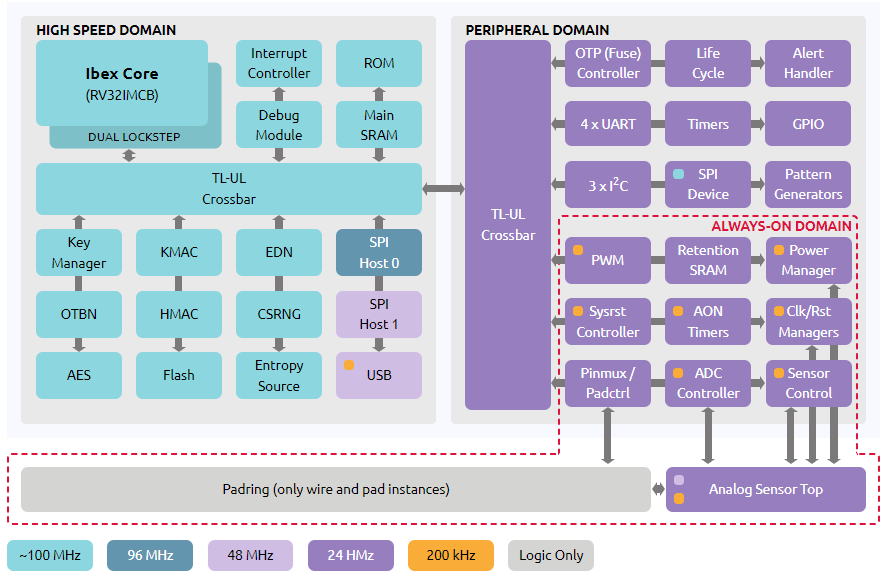


Figure 1 OpenTitan HW overview

## **2.2. Ibex core**

### 2.2.1. Links

Documentation: [https://ibex-core.readthedocs.io/en/latest/index.html](https://ibex-core.readthedocs.io/en/latest/index.html%20)

GitHub: <https://github.com/lowRISC/ibex/tree/master>

### 2.2.2. About

Ibex Core, also known as Ibex, is an open-source, high-performance, and small-footprint processor core designed specifically for RISC-V architecture. RISC-V is an instruction set architecture (ISA) that is becoming increasingly popular in the industry due to its open nature, simplicity, and flexibility.

Ibex Core was developed by researchers at the University of Cambridge and the University of Edinburgh. Being an open-source project, Ibex Core encourages community contributions and allows for customization and optimization by developers. It has gained attention from both academic and industry communities, and it serves as a foundation for further research and development in the RISC-V ecosystem.

### 2.2.3. HW Architecture overview

Ibex is a pipeline type core including two stages:

1. Instruction Fetch (IF): Fetches instructions from memory via a prefetch buffer, capable of fetching 1 instruction per cycle if the instruction side memory system allows.
2. Instruction Decode and Execute (ID/EX): Decodes fetched instruction and immediately executes it, register read and write all occurs in this stage. Multi-cycle instructions will stall this stage until they are complete.

All instructions require two cycles minimum to pass down the pipeline. One cycle in the IF stage and one in the ID/EX stage. Not all instructions can complete in the ID/EX stage in one cycle so will stall there until they complete. This means the maximum IPC (Instructions per Cycle) Ibex can achieve is 1 when multi-cycle instructions aren’t used.

Key units in Ibex:

1. Fetch Unit: responsible for fetching instructions from the instruction memory based on the program counter (PC). It includes an instruction cache to reduce memory access latency and improve performance.
2. Decode Unit: decodes the fetched instructions, determining their types, operands, and execution requirements. It extracts the necessary information to control subsequent stages of the pipeline.
3. Register File: The register file is a set of registers that stores data during instruction execution. Ibex Core includes multiple general-purpose registers, which are used for temporary storage and passing data between instructions.
4. Execution Units: Ibex Core consists of multiple execution units, each capable of performing specific types of operations. These units may include an arithmetic logic unit (ALU) for basic arithmetic and logical operations, a multiplier for multiplication operations, and a divider for division operations. The number and types of execution units may vary depending on the configuration and extensions chosen.

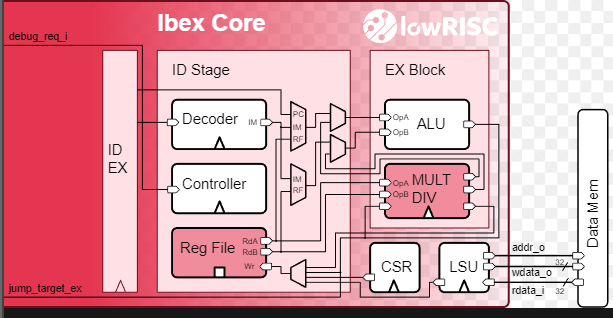


Figure 2 Ibex HW overview

#### CSR

Ibex implements a set of Control and Status Registers (CSRs), these registers provide control and status information for the processor or system. And typically used to configure various aspects of the processor's behavior, monitor its operation, and facilitate communication with the underlying system.

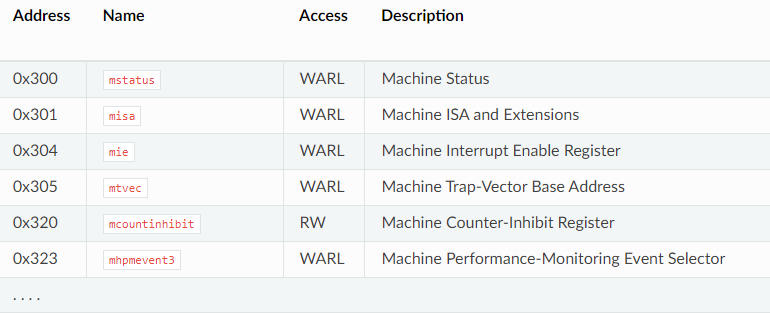


Table 1 Ibex CSRs

#### MMIO

To implement additional hardware accelerators or for software-hardware interaction Memory Mapped I/O technique can be used, each module gets a piece of the memory space. Which can be read from / written to by CPU. A specific write to this memory can activate the device or configure some specific use case.

#### Lockstep

Ibex supports a configuration option that instantiates a second copy of the core logic, referred to as the shadow core. The shadow core executes using a delayed version of all inputs supplied to the main core. All outputs of the shadow core are compared against a delayed version of the outputs of the main core. Any mismatch between the two sets of outputs will trigger an internal major alert.

This feature enables some protection against Fault injection attacks (Section 2.4). In our project we improved this feature to include protection on the register file and allow restoration of the core after detecting a mismatch.

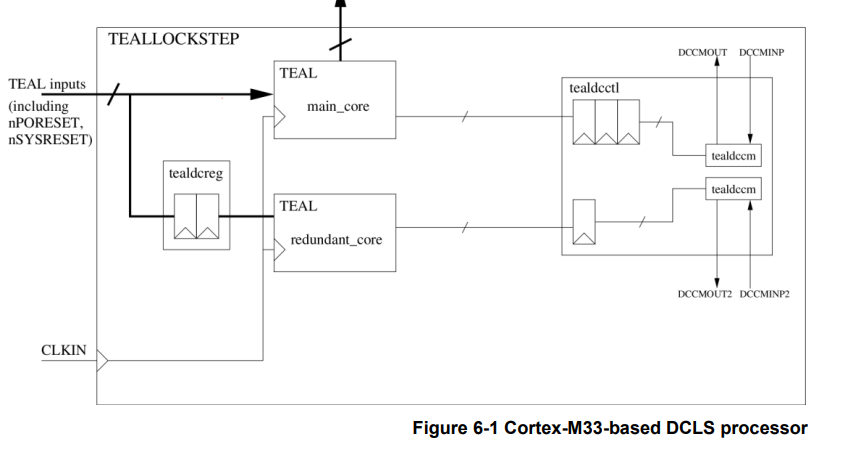


Figure 3 Lockstep Scheme

#### ePMP

Enhanced Physical Memory Protection unit, implements region-based memory access checking in-accordance with the RISC-V Privileged Specification. Addresses from the instruction fetch unit and load-store unit are passed to the PMP module for checking, and the output of the PMP check is used to gate the external request. To maintain consistency with external errors, the instruction fetch unit and load-store unit progress with their request as if it was granted externally. The PMP error is registered and consumed by the core when the data would have been consumed.

## **2.3. Secure Boot**

The basic guarantee of secure boot on OpenTitan is that no unauthorized code will be executed before the boot process reaches the device owner’s code. All executed code must be cryptographically signed by either the owner of the OpenTitan device or the (trusted) entity that originally set up the device at manufacturing time (the “Silicon Creator”).

Additionally, the secure boot procedure restricts certain stages to the Silicon Creator, so that even the current device owner can’t change them. Therefore, if the device changes owners, the new owner only has to trust the Silicon Creator, not the previous owner(s).

[Boot stages:](https://opentitan.org/book/doc/security/specs/secure_boot/index.html?highlight=secure%20boot#boot-stages)

* ROM: Metal ROM, sometimes known as ROM or Boot ROM.
* ROM\_EXT: ROM Extension. Stored in flash and signed by the Silicon Creator.
* BL0: Bootloader. Signed by the Silicon Owner.
* Kernel: Post-bootloader code. Signed by the Silicon Owner.

The diagram below summarizes the specific steps involved in the secure boot process:

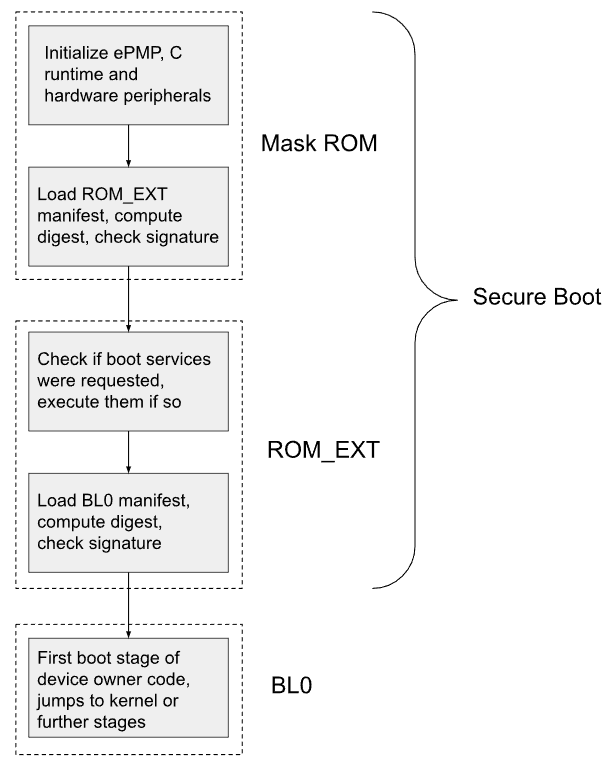


Figure 4 Secure boot process

**Notice:** in our project we focus on ROM part of the boot sequence.

### 2.3.1. ROM

The first stage of secure boot is called “ROM”. ROM is a region of read-only memory that cannot be updated at all after an OpenTitan device is manufactured. For that reason, the ROM is kept as simple as possible; it does some minimal setup, authenticates the next stage (ROM\_EXT), and jumps there.

At first when chip is powered on, a rom checker scheme is running to validate cryptographic signatures on ROM code. Later in normal mode the ROM address is requested to be executed on core. We’re interested in the normal mode of work after the code was validated. In normal mode ROM access is requested over the system TL-UL bus. These come in through the TL-UL SRAM adapter.

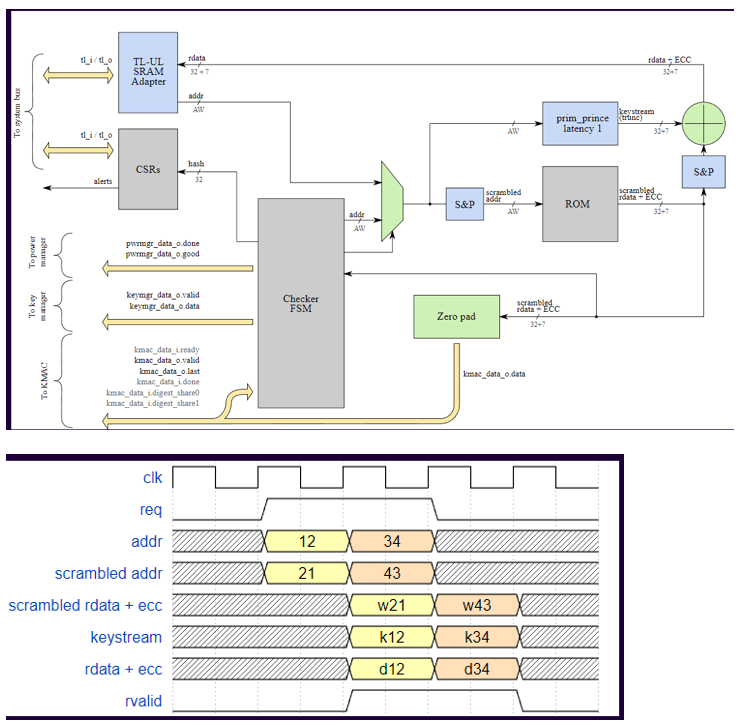


Figure 5 ROM instruction read.

On boot, the ROM code does the following:

1. Initialize any additional ePMP settings beyond what is set by hardware at startup.
2. Load the “manifest” for ROM\_EXT, which includes the start and end address of ROM\_EXT code, a cryptographic signature, a public key modulus, and “selector bits” for hardware information.
3. Read usage constraints from hardware according to the selector bits from the manifest.
4. Compute the SHA2-256 digest of the selected usage constraints concatenated with the contents of memory between the ROM\_EXT start and end address given in the manifest.
5. Check the signature from the manifest against the digest and the selected Silicon Creator public key.

## **2.4. Fault injection attacks**

## **2.5. Dual core secure boot**

## **2.6. Verilator & Bazel**

# **3. Hardware Modifications**

1. Modules

This project is based on open-source silicon root of trust “OpenTitan”, which based on RISC-V architecture and Ibex core design. The whole system is large and very complex, therefore in this documentation listed only modifications that concerns dual core secure boot system, links to full documentation may be found in this article in section 3.

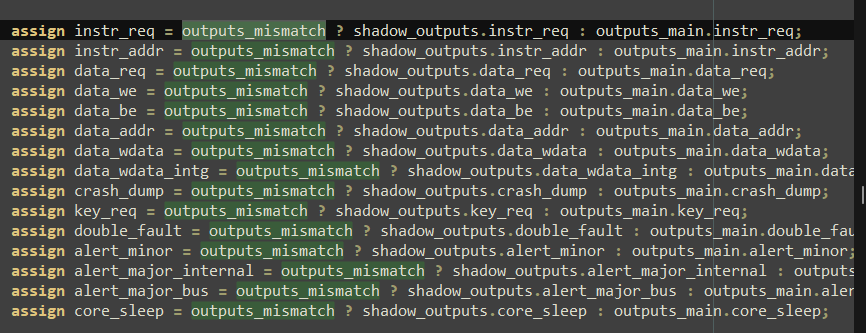
* 1. boot\_identifier.sv

The main goal of this module is to indicate start of boot address and send an activation signal to the “shadow” core. After finishing the boot sequence, the module sends de-activation signal to the “shadow” core. Currently this module is obsolete, since it was determined that boot address is constant and its address is at range of \_\_\_\_\_\_\_\_\_\_ and doesn’t change from run to run, so instead of using this module, activation of the “shadow” core is bounded to the current instruction address.

* 1. rv\_core\_ibex.sv

This is the top wrapper of ibex core which includes Ibex RISC-V core + optionally EMC instruction sets. Instruction and data bus are 32 bit wide. In this module we instantiated main core and shadow core and connected both outputs buses to the comparator, which enables whenever it gets compare\_command signal and current instruction address is at boot range. If there is a mismatch in cores outputs, comparator output “output\_mismatch” raises and (under assuming that the main core is the one corrupted) causes shadow core’s output, temperately become a main core output, until main core recovers from the failure.





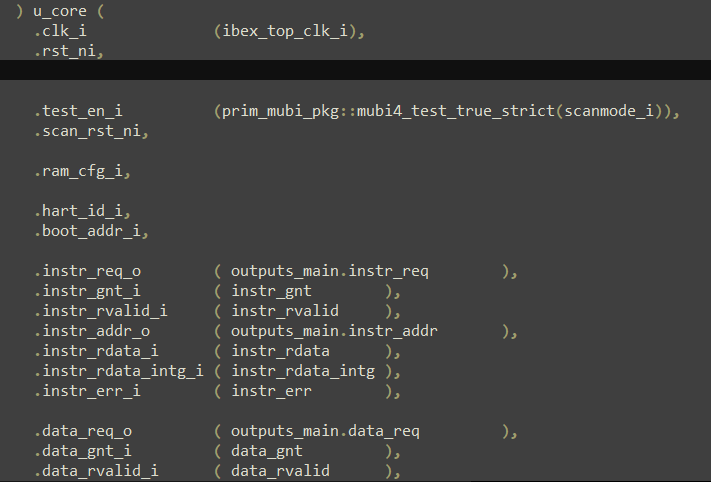


Figure 6: Main core instantiation



Figure 7: Boot address range

# **4. Software Modifications**

# **5. Validation**

# **6. Conclusions**

# **7. Future Research**

# **8. Appendix**