DUAL CORE SECURE BOOT

BORIS BUKCHIN, SERGEY MASHKIN
SUPERVISED BY: AVI MENDELSON

Date of presentation: 14/06/23

Contents

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 its 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 outputs the shadow core restores 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. Overview

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.

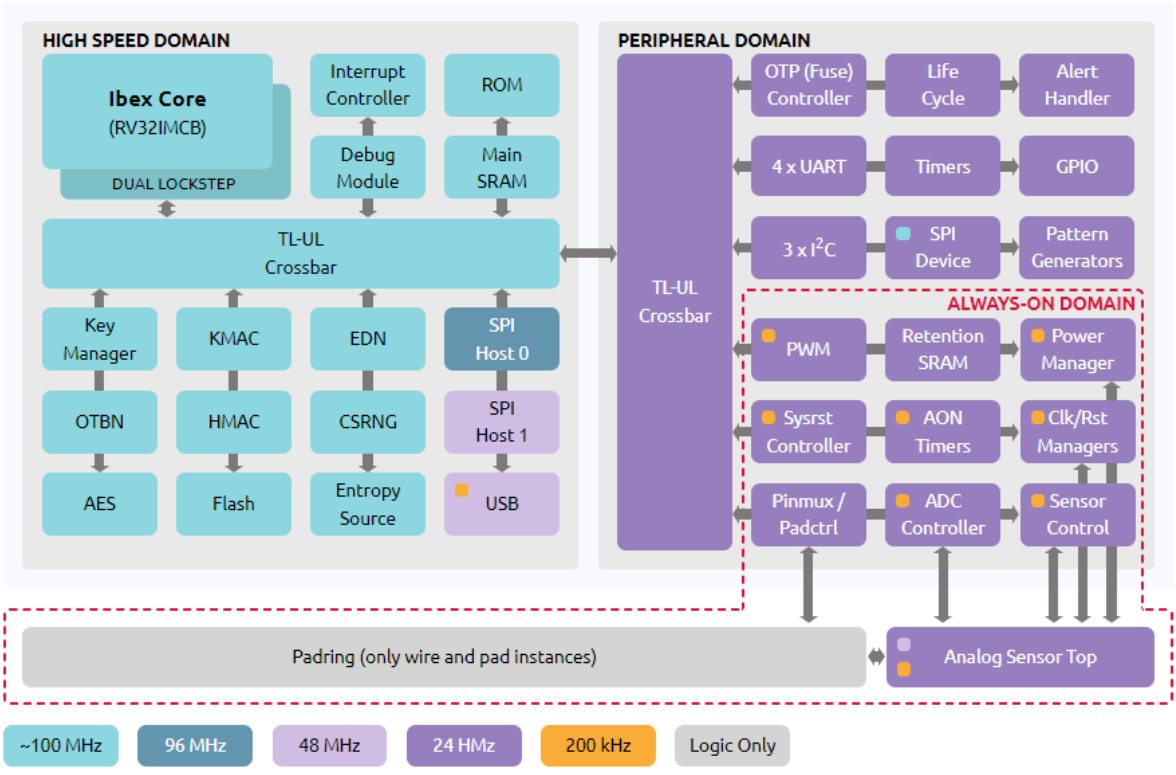


Figure 1 OpenTitan HW overview

2.2. Ibex core

2.2.1. Links

Documentation: https://ibex-core.readthedocs.io/en/latest/index.html

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

2.2.2. Overview

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 be completed 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.

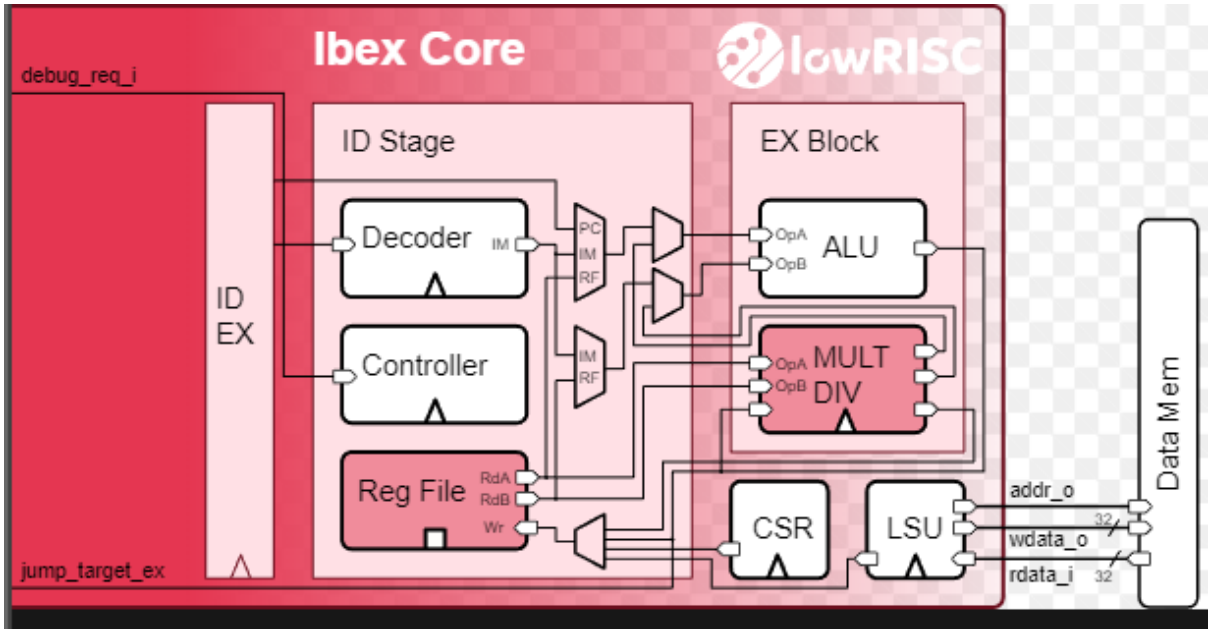


Figure 2 Ibex HW overview

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.

Address	Name	Access	Description
0x300	`mstatus`	WARL	Machine Status
0x301	`misa`	WARL	Machine ISA and Extensions
0x304	`mie`	WARL	Machine Interrupt Enable Register
0x305	`mtvec`	WARL	Machine Trap-Vector Base Address
0x320	`mcountinhibit`	RW	Machine Counter-Inhibit Register
0x323	`mhpmevent3`	WARL	Machine Performance-Monitoring Event Selector
....			

Table 1 Ibex CSRs

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.

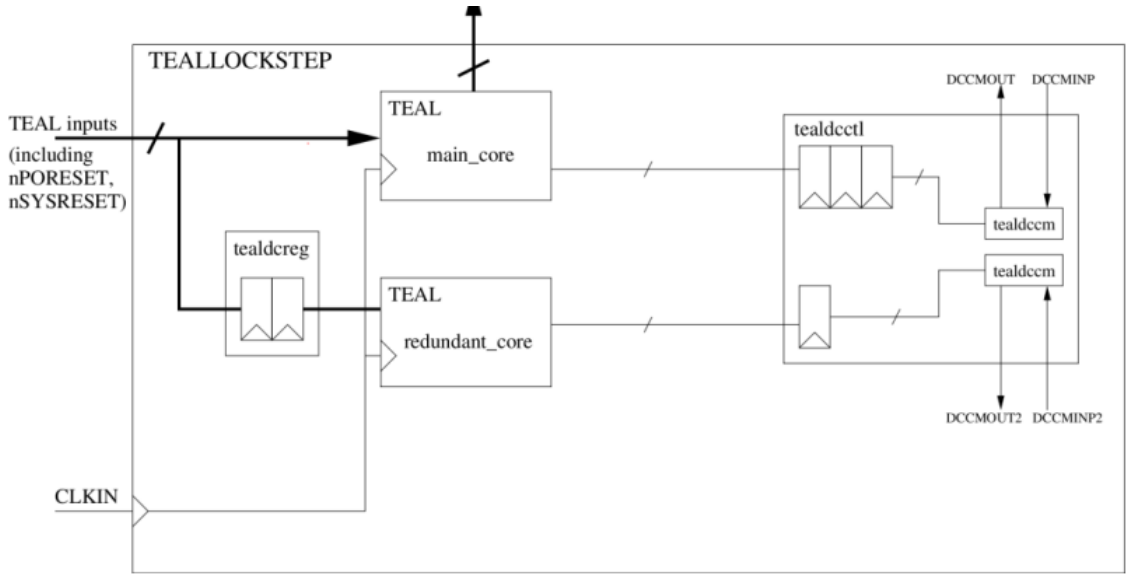


Figure 6-1 Cortex-M33-based DCLS processor

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:

- 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:

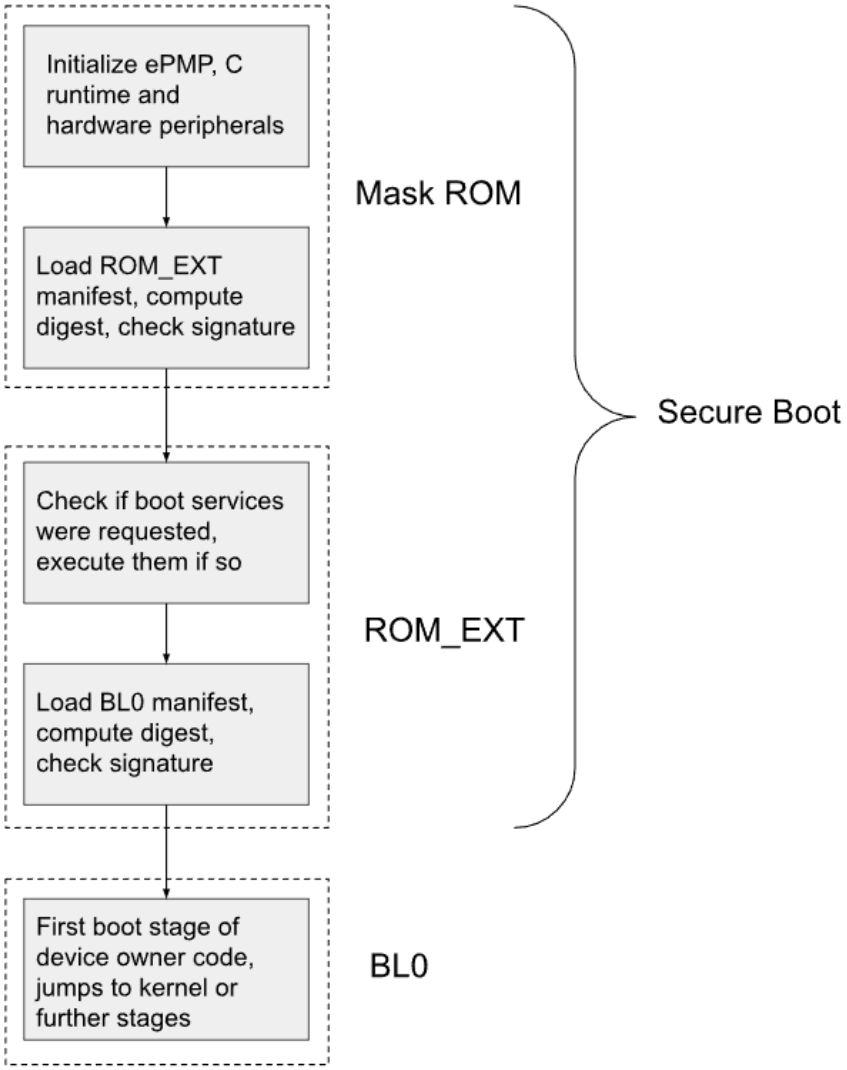


Figure 4 Secure boot process

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.

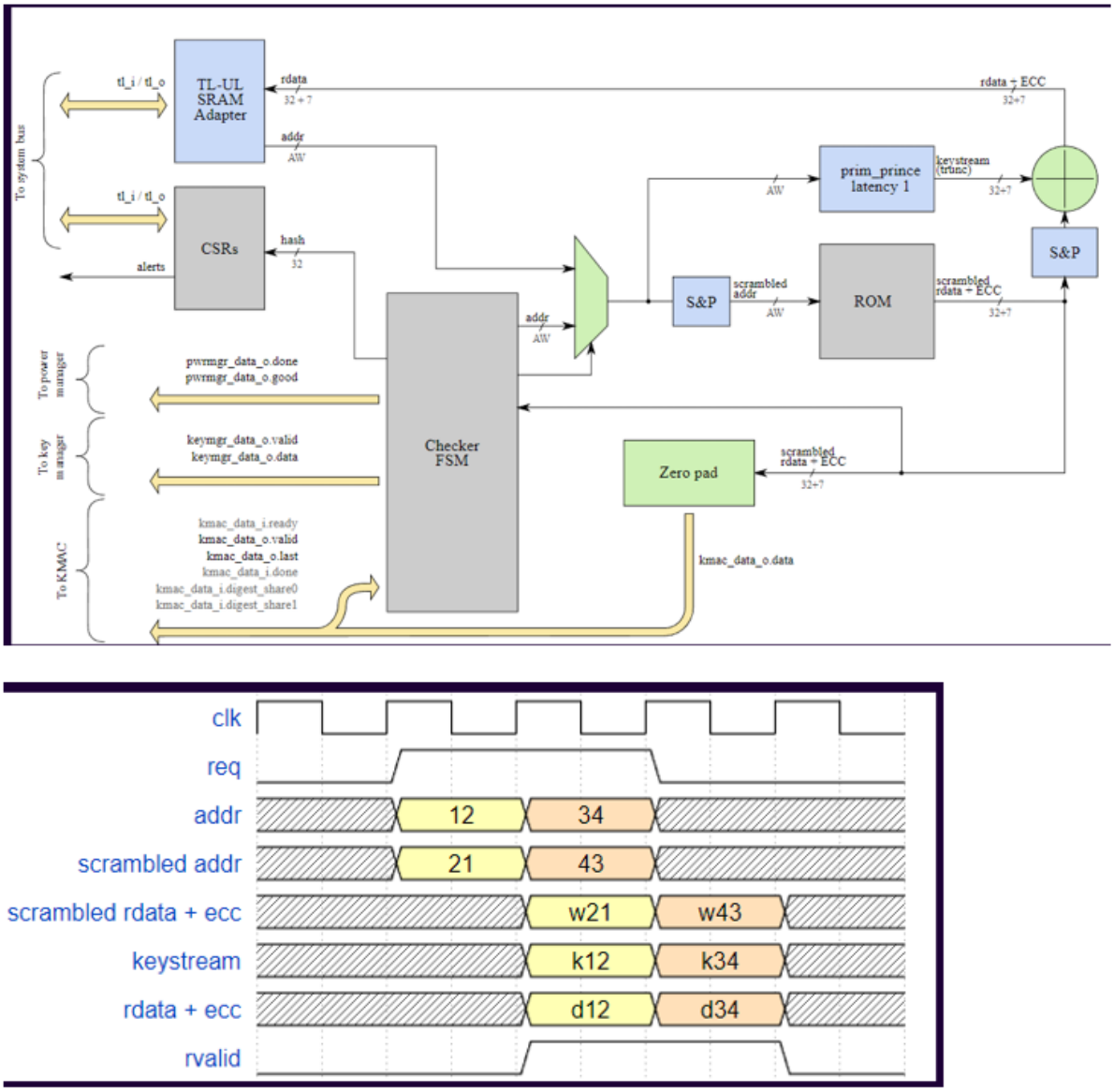


Figure 5 ROM instruction read.

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

2.4.1. Overview

Hardware fault injection attacks, target the hardware components of a system. These attacks involve intentionally inducing faults or errors in the physical hardware to exploit vulnerabilities and compromise the security of the system. Hardware fault injection attacks typically involve manipulating the electrical or physical properties of the targeted hardware components to disrupt their normal operation. By injecting faults into the hardware, attackers aim to cause the system to behave unexpectedly, which can lead to security breaches or unauthorized access to sensitive information.

There are various techniques used in hardware fault injection attacks, including:

1. Power Fault Injection: manipulating the power supply to the targeted hardware component by injecting voltage spikes, power glitches, or power supply disruptions. This can cause the system to enter an undefined state or bypass security mechanisms.
2. Clock fault Injection: Manipulating the clock signals provided to the targeted hardware component, leading to timing errors or synchronization issues. This can disrupt the normal execution flow and potentially compromise the security of the system.
3. Electromagnetic Fault Injection: Inducing electromagnetic interference or radiation to disrupt the operation of the hardware component. This can result in data corruption, memory bit flips, or other unexpected behaviors.
4. Temperature Fault Injection: Manipulating the temperature of the hardware component to induce faults or errors. Temperature variations can affect the electrical properties of the hardware and lead to security vulnerabilities.

2.4.2. Examples

Fault Injection attacks propose a real threat to real life systems, especially in boot sequence as it has the highest level of privilege and is not protected by software security components. Examples for such attacks are researched in many papers such as:

1. Laser-Induced Fault Injection on Smartphone bypassing the secure boot, where they show it was possible to build an attack scenario allowing to malevolently get the highest level of privilege within the mobile phone: https://ieeexplore.ieee.org/abstract/document/8167709
2. Controlling PC on ARM using fault injection, where they target the initial boot stages to introduce an ARM specific fault injection attack strategy for exploiting embedded systems where externally controlled data is loaded in the program counter (PC) register of the processor. This allows an attacker to control the target's execution flow which eventually will lead to arbitrary code execution on the target: https://ieeexplore.ieee.org/abstract/document/7774479

2.4.3. Mitigations

To mitigate the risks associated with hardware fault injection attacks, several common mitigations can be employed. Here are a few examples:

1. Redundancy and Error Correction Codes.
2. Error Detection Mechanisms.
3. Physical security measures, such as tamper-resistant packaging, epoxy coatings, or secure enclosures.
4. And more.

2.5. Dual core secure boot

2.5.1. Overview

As a mitigation for Fault Injection Attacks, in this project we introduce Dual Core Secure Boot. The idea is that two cores will execute ROM code simultaneously and validate each other's outputs. The main core is responsible to memory writes, while the shadow core gets the same inputs as the min core and its output only validates the main core's output. The assumption is that attacking two cores simultaneously is much more challenging for the attacker than attacking only one core.

The output comparator is activated by a specific write to CSR register for a limited amount of clock cycles. During which the two cores will validate each other's outputs. In case of mismatch detection, the shadow core is responsible for memory writes for a few cycles, while restoring the main core's registers to correct values. We assume in this project that the shadow core is always right, and main core is vulnerable to attacks.

The main advantages of dual core secure boot:

1. Dual Core Secure Boot provides protection against register fault injection attacks, while in lockstep the registers are not duplicated.
2. Ability to restore the system to correct behavior after an attack was detected.
3. Ability to control with software to disable/activate the validation for specific parts of code.
4. In multi core systems our method provides an efficient usage of resources to provide security.

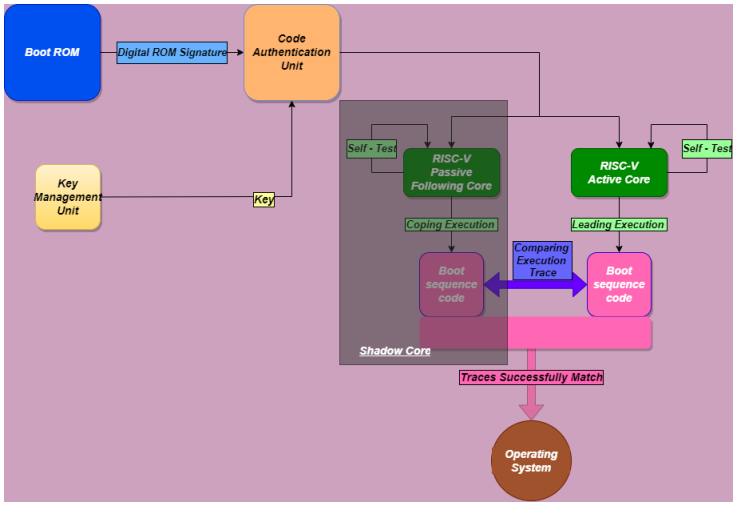


Figure 6 Dual core boot block diagram

2.5.2. High level Hardware Description

Main modules:

1. Shadow core: We create another instantiation of the Ibex core to implement the shadow core. The core will get its inputs from same buffer as the main core and its output routed to output comparator.
2. Compare command: We implement a command from software to initiate comparisons between outputs using writes to CSR.
3. Ibex register files: We implement a bus to override main core's registers with shadow registers in case of output mismatch.

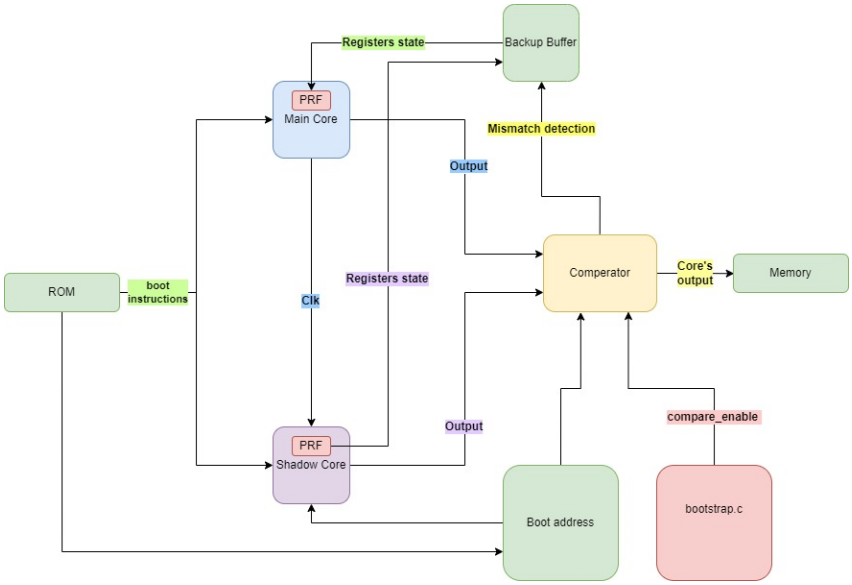


Figure 7 High level HW description

2.6. Development Tools

2.6.1. Verilator

For Design Validation (DV) purposes we use in our project “Verilator” tool. Verilator is an open-source software tool used for the simulation and analysis of hardware description language (HDL) designs. It is specifically designed for simulating and verifying designs written in the Verilog hardware description language (HDL) and the SystemVerilog hardware modeling language.

This tool was selected because of these features:

1. Speed: Verilator is known for its fast simulation speed. By generating an optimized C++ model, it can achieve significantly faster simulation times compared to traditional simulators, making it suitable for large and complex designs.
2. Open-source: Verilator is an open-source tool, meaning that its source code is freely available.
3. Cycle-accurate simulation: Verilator provides cycle-accurate simulation, meaning that it models the behavior of the hardware at each clock cycle. This level of accuracy is essential for verifying timing-dependent aspects of the design.
4. Support for SystemVerilog: Verilator supports a subset of the SystemVerilog language, enabling users to simulate designs that utilize advanced features provided by SystemVerilog.

Using this tool allowed us accurately debug and validate our design and is essential in hardware development. Installation instructions are described in the appendix section of this article.

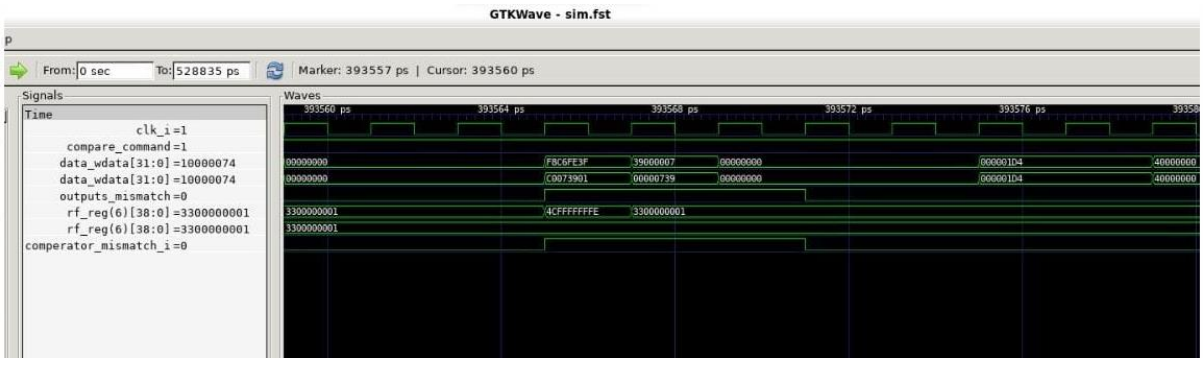


Figure 8 Verilator/GTKWave example

2.6.2. Bazel

Bazel is a build system and build tool developed by Google. It is used in various software projects, including the OpenTitan project. Bazel provides a scalable and efficient infrastructure for building, testing, and deploying software across different platforms and languages.

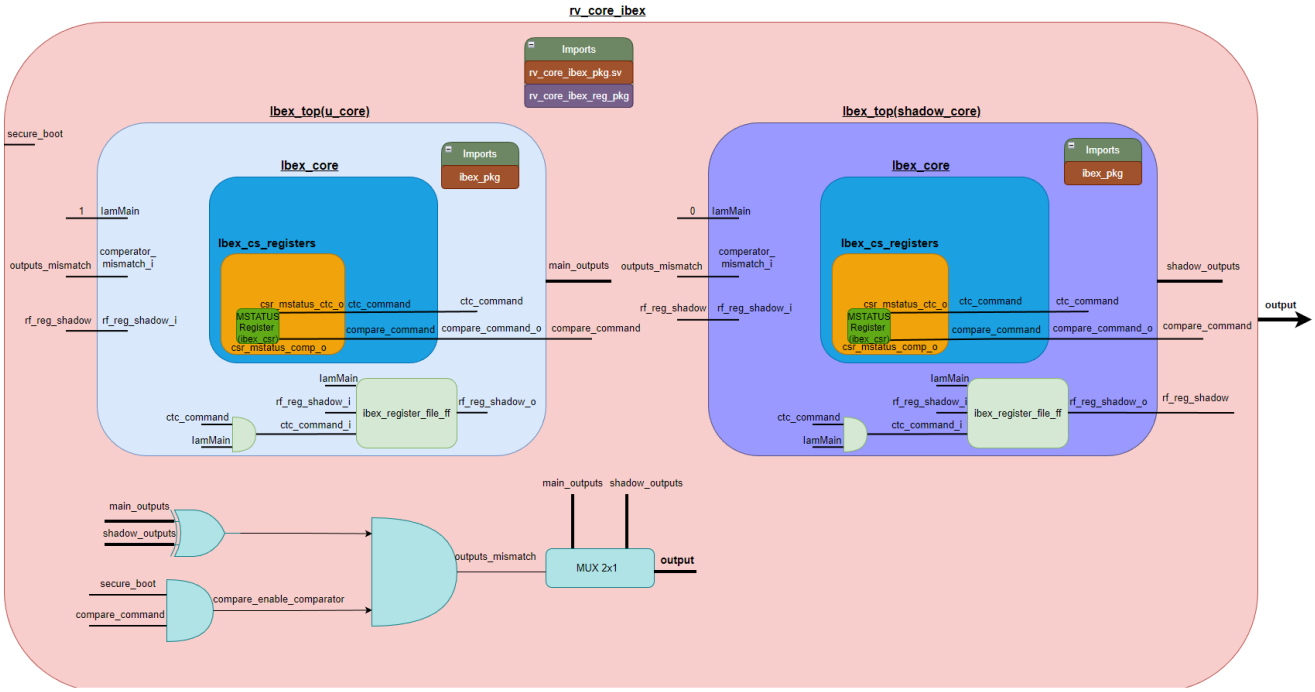
Bazel is used as the build system to manage the compilation, testing, and deployment of the OpenTitan hardware and software components. It provides a consistent and scalable approach to building complex systems, ensuring that all the dependencies are resolved correctly and that the build process is efficient and reliable.

Installation instructions and some tips are provided in the appendix section of this article.

3. Hardware Modifications

Important note:
This project is built upon the open-source silicon root of trust called "OpenTitan," which utilizes the RISC-V architecture and Ibex core design. Given the extensive size and complexity of the entire system, this documentation focuses solely on the modifications relevant to the dual-core secure boot system.

TOP view of hardware modifications:



3.1. rv_core_ibex.sv

This module serves as the top wrapper for the Ibex core, incorporating the Ibex RISC-V core along with optional EMC instruction sets. The instruction and data bus widths are set to 32 bits. The primary function of this module is to enable the dual-core boot system and facilitate communication between the two cores.

To initiate the modifications, the first step involved instantiating the inner wrappers, namely "ibex_top," for both the main and shadow cores (refer to Figure 9). Our specific changes begin by setting the “secure_boot” signal to HIGH when the current instruction address (“instr_addr”) reaches the boot address section (as illustrated in Figure 12).

When the “secure_boot” signal rises, it triggers the activation of the shadow core (“u_shadow_core”), which previously operated with a constant clock signal of 0. To initiate the comparison of the cores' outputs, the “compare_command” signal must be set to HIGH. The “compare_command” signal is connected to the output of the “ibex_top” module (“compare_command_o”) of the main core (“u_core”). Once both the “compare_command” and “secure_boot” signals are set to HIGH, our comparator becomes active (compare_enable_comparator), and continuous monitoring of the cores' outputs begins (as shown in Figure 10).

Whenever the comparator is enabled and detects a mismatch between the outputs of the two cores, the mismatch signal (outputs_mismatch) is set to HIGH (as depicted in Figure 10). A detected mismatch signifies that the output of the main core has been compromised, and it will require a certain amount of time to recover and resume execution. In the meantime, the output of the shadow core will be used as the main output, ensuring an uninterrupted boot sequence. Once the main core has fully recovered, it will regain control, and its output will be utilized for further execution until the next mismatch occurs (as shown in Figure 11).

```
) u_core (
  .clk_i                (ibex_top_clk_i),
  .rst_ni,

  .test_en_i            (prim_mubi_pkg::mubi4_test_true_strict(scanmode_i)),
  .scan_rst_ni,

  .ram_cfg_i,

  .hart_id_i,
  .boot_addr_i,

  .instr_req_o          ( outputs_main.instr_req        ),
  .instr_gnt_i          ( instr_gnt         ),
  .instr_rvalid_i       ( instr_rvalid      ),
  .instr_addr_o         ( outputs_main.instr_addr       ),
  .instr_rdata_i        ( instr_rdata       ),
  .instr_rdata_intg_i   ( instr_rdata_intg  ),
  .instr_err_i          ( instr_err         ),

  .data_req_o           ( outputs_main.data_req         ),
  .data_gnt_i           ( data_gnt          ),
  .data_rvalid_i        ( data_rvalid       ),
```

```
) u_shadow_core (
  .clk_i                (ibex_top_clk_i & secure_boot),
  .rst_ni,


  .test_en_i            (prim_mubi_pkg::mubi4_test_true_strict(inputs_shadow.scanmode_i)),
  .scan_rst_ni,

  .ram_cfg_i,

  .hart_id_i,
  .boot_addr_i,

  .instr_req_o          ( shadow_outputs.instr_req        ),
  .instr_gnt_i          ( inputs_shadow.instr_gnt),
  .instr_rvalid_i       ( inputs_shadow.instr_rvalid),
  .instr_addr_o         ( shadow_outputs.instr_addr     ),
  .instr_rdata_i        ( inputs_shadow.instr_rdata     ),
  .instr_rdata_intg_i   ( inputs_shadow.instr_rdata_intg),
  .instr_err_i          ( inputs_shadow.instr_err       ),

  .data_req_o           ( shadow_outputs.data_req         ),
  .data_gnt_i           ( inputs_shadow.data_gnt ),
  .data_rvalid_i        ( inputs_shadow.data_rvalid      ),
```

Figure 9: Ibex_top cores instantiation for both main and shadow core

```
assign compare_enable_comparator = secure_boot && compare_command;
assign outputs_mismatch = (compare_enable_comparator & (shadow_outputs != outputs_main));
```

Figure 10: Comparator and mismatch detect activation

```
assign instr_req = outputs_mismatch ? shadow_outputs.instr_req : outputs_main.instr_req;
assign instr_addr = outputs_mismatch ? shadow_outputs.instr_addr : outputs_main.instr_addr;
assign data_req = outputs_mismatch ? shadow_outputs.data_req : outputs_main.data_req;
assign data_we = outputs_mismatch ? shadow_outputs.data_we : outputs_main.data_we;
assign data_be = outputs_mismatch ? shadow_outputs.data_be : outputs_main.data_be;
assign data_addr = outputs_mismatch ? shadow_outputs.data_addr : outputs_main.data_addr;
assign data_wdata = outputs_mismatch ? shadow_outputs.data_wdata : outputs_main.data_wdata;
assign data_wdata_intg = outputs_mismatch ? shadow_outputs.data_wdata_intg : outputs_main.data
assign crash_dump = outputs_mismatch ? shadow_outputs.crash_dump : outputs_main.crash_dump;
assign key_req = outputs_mismatch ? shadow_outputs.key_req : outputs_main.key_req;
assign double_fault = outputs_mismatch ? shadow_outputs.double_fault : outputs_main.double_fau
assign alert_minor = outputs_mismatch ? shadow_outputs.alert_minor : outputs_main.alert_minor;
assign alert_major_internal = outputs_mismatch ? shadow_outputs.alert_major_internal : outputs
assign alert_major_bus = outputs_mismatch ? shadow_outputs.alert_major_bus : outputs_main.aler
assign core_sleep = outputs_mismatch ? shadow_outputs.core_sleep : outputs_main.core_sleep;
```

Figure 11: Switching to shadow core in case of a mismatch.

```
// Snoop to get the secure boot true signal back to:
logic secure_boot;
assign secure_boot = ( instr_addr >= 32'h00008000 && instr_addr <= 32'h0000AFFF );
// instantiation of shadow
```

Figure 12: Boot address range

3.2. rv_core_ibex_pkg.sv

This module serves as an import for the "rv_core_ibex" module and provides various struct implementations that are utilized by "rv_core_ibex." As part of our modifications to the boot sequence, the inclusion of an additional core necessitates the definition of inputs and outputs. To fulfill this requirement, we have established the appropriate type declarations within this module, namely "core_outputs_t" and "core_inputs_t." These declarations enable seamless communication and data exchange between the different components involved in the boot process.

3.3. Ibex_top.sv

The Ibex_top.sv module serves as a crucial connector between the rv_core_ibex wrapper, ibex_core (the main core), and ibex_register_file_ff (the register file). Its main purpose is to establish the necessary connections for seamless communication and coordination among these components.

Ibex_top provides the following connections and functionalities:

Mismatch Signal (comperator_mismatch): This signal indicates any detected mismatches between the outputs of the cores. It is connected to the register file, allowing appropriate handling of mismatches.

Main Core Identification Signal (IamMain): The IamMain signal is essential for distinguishing between the main and shadow cores in the OpenTitan architecture. It helps identify the active core and enables different behaviors to be implemented for each core.

CTC* Signal (ctc_command): The CTC* signal is connected to the register file and facilitates specific commands and operations related to the register file. The register file accepts the CTC signal only when it receives the main core identification signal, ensuring that mismatches can occur exclusively in the main core.

Input and Output Shadow Core Registers (rf_reg_shadow_i, rf_reg_shadow_o): Ibex_top module passes the input and output values of the shadow core registers through the module. This enables the integration and communication of these registers within the overall system.

Additionally, it's worth noting that the compare and CTC signals originate from the main core itself, further emphasizing the central role of the main core in Ibex_top.

By incorporating these connections and functionalities, Ibex_top facilitates efficient communication and coordination between the rv_core_ibex wrapper, the main core, and the register file. It enables the implementation of distinct behaviors for the main and shadow cores within the OpenTitan architecture, ensuring proper functionality and allowing for mismatches to occur exclusively in the main core.

3.4. Ibex_core.sv

Top level module of the ibex RISC-V core that plays a fundamental role in constructing the core itself within the system. Its primary purpose is to assemble and integrate the various components and sub-modules that form the core's architecture. In line with our modifications, we have implemented the connection of the CTC and compare signals to the core's control and status registers file, known as ibex_cs_registers.sv. This enhancement enables effective communication and control between the core and the registers, further enriching the core's functionality and expanding its capabilities within the system.

CTC – "Check the Checker", a signal that causes a forced mismatch between main and shadow cores for testing and debug.

3.5. Ibex_cs_registers.sv

Top module responsible for managing the control and status registers. It serves as a hub for gathering and transferring input and output signals between the registers and the cores.

In our modifications, we specifically focused on enhancing the control panel of the MSTATUS register. These modifications enable efficient handling of the CTC and Compare_Command registers by implementing the necessary read and write logic.

Additionally, to introduce unique logic behavior to the MSTATUS register, the ibex_cs_registers module sends a HIGH signal specifically to the ibex_csr submodule (mstatus_en_i), while all other sets of registers receive a LOW signal (You can observe an example in Figure 13). This approach allows us to alter the logic of the MSTATUS register without impacting any other register sets.

It's important to note that when writing to the registers, the process takes two clock cycles. During the first cycle, the writing command is detected, and the hardware executes the required commands. However, the updated values are reflected in the registers only during the subsequent cycle.

```
assign csr_mstatus_mie_o   = mstatus_q.mie;
assign csr_mstatus_tw_o    = mstatus_q.tw;
assign csr_mstatus_ctc_o = mstatus_q.ctc;
assign csr_mstatus_comp_o = mstatus_q.comp;
assign debug_single_step_o = dcsr_q.step;
assign debug_ebreakm_o     = dcsr_q.ebreakm;
assign debug_ebreaku_o     = dcsr_q.ebreaku;

// Qualify incoming interrupt requests in mip CSR with mie CSR for controller and to re-enable
// clock upon WFI (must be purely combinational).
assign irqs_o        = mip & mie_q;
assign irq_pending_o = |irqs_o;

////////////////////////
// CSR instantiations //
////////////////////////s

// MSTATUS

localparam status_t MSTATUS_RST_VAL = '{mie:  1'b0,
                                       comp: 1'b0,
                                       ctc: 1'b0,
                                       mpie: 1'b1,
                                       mpp:  PRIV_LVL_U,
                                       mprv: 1'b0,
                                       tw:   1'b0};

ibex_csr #(
  .Width     ($bits(status_t)),
  .ShadowCopy(ShadowCSR),
  .ResetValue({MSTATUS_RST_VAL})
) u_mstatus_csr (
  .clk_i     (clk_i),
  .rst_ni    (rst_ni),
  .wr_data_i ({mstatus_d}),
  .wr_en_i   (mstatus_en),
  .mstatus_en_i(1'b1),
  .rd_data_o (mstatus_q),
  .rd_error_o(mstatus_err)
);
```

Figure 13: CSR instantiations in ibex_cs_registers module.

3.6. Ibex_csr.sv

Ibex_csr.sv serves as the main control module for a single register set, specifically describing the behavior of the registers. In our modifications, we introduced a signal that distinguishes the MSTATUS register from the others.

The primary goal of this modification was to ensure that the CTC command remains active for a single cycle, regardless of the user's actions. This was crucial because as long as the CTC signal is high, the main core will consistently produce a mismatch. However, writing a '0' to the CTC bit of the MSTATUS register through software would take a significant number of cycles. Therefore, we implemented this functionality in hardware to effectively write to the CTC bit as a pulse, triggering a mismatch but without interfering with the mismatch handling process.

When the MSTATUS register detects the CTC command, it forcefully writes a LOW signal to the CTC bit. Since the writing process takes two cycles, the CTC bit will transition from HIGH to LOW a cycle after it has been set to HIGH. (The logic behind this behavior can be observed in Figure 14).

```
module ibex_csr #(
  parameter int unsigned    Width      = 32,
  parameter bit             ShadowCopy = 1'b0,
  parameter bit [Width-1:0] ResetValue = '0
 ) (
  input  logic             clk_i,
  input  logic             rst_ni,

  input  logic [Width-1:0] wr_data_i,
  input  logic             wr_en_i,
  input  logic             mstatus_en_i,
  output logic [Width-1:0] rd_data_o,

  output logic             rd_error_o
);
  logic ctc_pulse;
  logic [Width-1:0] rdata_q;

  always_ff @(posedge clk_i or negedge rst_ni) begin
    if (!rst_ni) begin
      rdata_q <= ResetValue;
      ctc_pulse <= 1'b0;
    end else if (wr_en_i)  begin
      if (ctc_pulse) begin
        ctc_pulse <= 1'b0;
        rdata_q <= wr_data_i & 32'b11111111111111111111111111011111;
      end else begin
        rdata_q <= wr_data_i;
        if ((wr_data_i[5]) && (mstatus_en_i)) begin
          ctc_pulse <= 1'b1;
        end
      end
    end else if (ctc_pulse) begin
        ctc_pulse <= 1'b0;
        rdata_q <= wr_data_i & 32'b11111111111111111111111111011111;
    end
  end

  assign rd_data_o = rdata_q;

  if (ShadowCopy) begin : gen_shadow
    logic [Width-1:0] shadow_q;

    always_ff @(posedge clk_i or negedge rst_ni) begin
      if (!rst_ni) begin
        shadow_q <= ~ResetValue;
      end else if (wr_en_i) begin
        shadow_q <= ~wr_data_i;
      end
    end
```

Figure 14: Logic of Status and Control registers.

3.7. ibex_register_file_ff.sv

The module serves as the top-level module for the RISC-V register file, which is implemented using flip flops. This module provides the necessary logic for handling mismatches by redirecting the shadow core's registers to the main core (as depicted in Figure 15).

Additionally, we have introduced logic to create deliberate mismatches whenever the CTC command is active. This functionality serves debugging and testing purposes. The logic flow is as follows:

a. When the CTC signal (ctc_active) is HIGH, all the bits in the main registers of the main core are flipped. This flip operation takes effect on the next clock cycle, resulting in a mismatch between the main and shadow cores.
b. When the mismatch signal (comperator_mismatch_i) is HIGH, the values of the shadow core's registers overwrite the values in the main core's registers. This ensures that the main core adopts the values of the shadow core in the event of a detected mismatch.

c. Shadow core's register values pass through core's wrappers to the main core, in order to handle mismatch case.

In summary, the ibex_register_file_ff.sv module handles mismatches by redirecting registers to restore correct boot sequence. And facilitates deliberate mismatches through CTC command activation, providing valuable debug and testing capabilities.

```
// No flops for R0 as it's hard-wired to 0
logic IamShadow = ~IamMain;
for (genvar i = 1; i < NUM_WORDS; i++) begin : g_rf_flops
  logic [DataWidth-1:0] rf_reg_q [NUM_WORDS];

  always_ff @(posedge clk_i or negedge rst_ni) begin
    if (!rst_ni) begin
      rf_reg_q[i] <= WordZeroVal;
    end else if (ctc_active) begin
      rf_reg_q[i] <= ~(rf_reg_q[i]);
    end else if (IamMain & comperator_mismatch_i) begin
      rf_reg_q[i] <= rf_reg_shadow_i[i];
    end else if (we_a_dec[i]) begin
      rf_reg_q[i] <= wdata_a_i;
    end
  end
  // Write to main registers write data or destroy data for CTC.
  assign rf_reg[i] = rf_reg_q[i];
  assign rf_reg_shadow_o[i] = rf_reg[i] & {39{IamShadow}};
end
```

Figure 15: Flip flop register file logic of CTC and mismatch handling.

3.8. ibex_pkg.sv

The ibex_pkg.sv module serves as a package that provides constants used by Ibex and other hardware modules. It plays a crucial role in setting the addresses and bit distribution within the CSR (Control and Status Registers) of Ibex.

During the implementation of our modifications, we encountered the challenge of finding available registers to accommodate our additional signals. Since the default registers in Ibex were already assigned specific purposes, we needed to identify registers with unused bits. After consulting the Ibex documentation, we identified the MSTATUS register as a suitable candidate. This register had several unused bits and supported user mode.

To incorporate our modifications, we allocated the fourth bit of the MSTATUS register to the Compare command and the fifth bit to the CTC command (refer to Figure 16 and Figure 17). In order to activate these commands, the user needs to set the respective bits to '1'. To achieve this, the user can write a value that raises only the fourth and/or fifth bit, such as 0x10, 0x20, or 0x30.

By leveraging the available bits in the MSTATUS register and configuring them for our desired commands, we were able to implement our modifications effectively.

Machine Status (mstatus)

CSR Address: `0x300`

Reset Value: `0x0000_1800`

Bit#	R/W	Description
21	RW	**TW:** Timeout Wait (WFI executed in User Mode will trap to Machine Mode).
17	RW	**MPRV:** Modify Privilege (Loads and stores use MPP for privilege checking).
12:11	RW	**MPP:** Machine Previous Privilege mode.
7	RW	**Previous Interrupt Enable (MPIE)**, i.e., before entering exception handling.
3	RW	**Interrupt Enable (MIE):** If set to 1'b1, interrupts are globally enabled.

Figure 16: Default MSTATUS register format.

```
// CSR status bits
parameter int unsigned CSR_MSTATUS_MIE_BIT      = 3;
parameter int unsigned CSR_MSTATUS_MPIE_BIT     = 7;
parameter int unsigned CSR_MSTATUS_MPP_BIT_LOW  = 11;
parameter int unsigned CSR_MSTATUS_MPP_BIT_HIGH = 12;
parameter int unsigned CSR_MSTATUS_MPRV_BIT     = 17;
parameter int unsigned CSR_MSTATUS_TW_BIT       = 21;
parameter int unsigned CSR_MSTATUS_COMPARE_COMMAND  = 4;
parameter int unsigned CSR_MSTATUS_SHADOW_CTC       = 5;
```

Figure 17: Modified MSTATUS register.

3.9. boot_identifier.sv – not in use

The boot_identifier.sv module serves the purpose of indicating the start of the boot address and sending an activation signal to the "shadow" core. Its primary function is to coordinate the activation and deactivation of the "shadow" core based on the boot sequence.

However, it should be noted that this module is currently not in use and considered obsolete. This determination was made during the project development when it was discovered that the boot address remains constant and falls within the range of 32'h00008000 to 32'h0000AFFF. The boot address does not change from one run to another.

As a result, instead of relying on the boot_identifier.sv module, the activation of the "shadow" core is now tied to the current instruction address. This change in approach eliminates the need for the boot_identifier.sv module, as the activation of the "shadow" core can be determined based on the instruction being executed rather than relying on a separate module.

4. Software Modifications

4.1. COMPARE_CSR

To enable output comparison between the two cores at specific areas of the code we use a special CSR bit. This bit can be configured from software using the “CSR_WRITE” function. When this bit is high, comparisons an restore systems are enabled and securing the code from error injections.

Register name	Bits	Description	Permissions
CSR_REG_MSTATUS	[4:4]	Enable output comparator	R/W

Table 2 COMPARE_CSR config

4.2. Compare command

To use the compare command code that is in risk of being attacked by fault injection is wrapped by the “CSR_WRITE” functions to the COMAPRE bit. We turn the bit high in the beginning of the code to provide security, later to save power when the code is no longer at risk, we turn COMPARE bit down using “CSR_WRITE” function.

For example, we chose the bootstrap.c of the ROM code. If it is known that bootstrap() function is vulnerable to fault injection attacks, one might try to defend using dual-core secure boot:

```
rom_error_t bootstrap(void) {
  CSR_READ(CSR_REG_MSTATUS, &existing_value);
  write_value = existing_value | 16 ; // existing_value | 0000010000(binary) - compare command
  CSR_WRITE(CSR_REG_MSTATUS, write_value);
  hardened_bool_t requested = bootstrap_requested();
  if (launder32(requested) != kHardenedBoolTrue) {
    return kErrorBootstrapNotRequested;
  }
  HARDENED_CHECK_EQ(requested, kHardenedBoolTrue);

  spi_device_init();
  CSR_READ(CSR_REG_MSTATUS, &existing_value);
  write_value = existing_value & -17 ; // existing_value | 1111111111 1111111111 1111111011 11(binary) - compare command
  CSR_WRITE(CSR_REG_MSTATUS, write_value);
```

Figure 18 bootstrap.c usage example

Notice: before CSR_WRITE we use CSR_READ to keep the original values of the MSTATUS register unmodified except for the desired bit.

Figure 9 Compare command enable waveform

In the waveform we can see that part of the boot ROM execution is protected by dual core – compare_enable_comparator is high, while other parts that are not protected to improve performance.

5. Validation

5.1. OpenTitan Validation environment

In OpenTitan project to provide high-quality hardware validation plays a crucial role. With the OpenTitan project we got various functional tests ready for validation. These tests are written in C language and are easy to run and debug using Verilator and Bazel. The OpenTitan project validation suite contains Formal validation, Security validation and Functional validation. But for our project we focus only on **Functional validation.**

Key notes about our usage:

1. Notice how ROM source code that is provided for testing is different than the ROM source code used in testing. Therefor when testing the feature changes have to be applied to the testing ROM.
2. Most of our validation was done in opentitan\sw\device\silicon_creator\rom\bootstrap.c This file is used in ROM boot sequence in any test therefore easy to use.
3. To validate that the new hardware doesn't break the system, we can run any test that uses boot sequence and check that boot is completed without errors.

5.1.1. Results

As we can see, multiple tests that run boot sequence passed without problems:

```
I00001 test_rom.c:235] Test ROM complete, jumping to flash (addr: 20000480)!
I00000 ottf_main.c:130] Running sw/device/tests/aes_smoketest.c
I00001 ottf_main.c:101] Finished sw/device/tests/aes_smoketest.c
I00002 status.c:28] PASS!
```

Figure 20 Functional test result

5.1. Error Injection

For proper validation of system's output mismatch detection and recovery mechanisms we use error injection technique or CTC (check the checker). The idea is to inject an error – bit flips in main core's registers to cause a wrong output in the main core. We look at the waveforms and check that the output mismatch was detected and that the registers recovery is working. In addition, we check that functional tests are passing as before.

5.1.1. CTC CSR

To inject the errors in hardware we use a special CSR bit. This bit can be configured from software using the "CSR_WRITE" function. When this bit is high, bit flip occurs in all main core's registers and forcing errors in output. Notice how for CTC to work, both compare command and CTC should be high.

Register name	Bits	Description	Permissions
CSR_REG_MSTATUS	[5:5]	Inject error in main core	R/W

Table 3 CTC_CSR config

To use CTC logic, we first need to read the existing value of MSTAUS CSR, then we want to turn the bits of compare command **and** for CTC for validation.

```
CSR_READ(CSR_REG_MSTATUS, &existing_value);
base_printf("\t##### Second Read MSTATUS register (compare comand) = %x #####\n", existing_value);
write_value = existing_value | 48 ; // existing_value | 00000110000(binary) - ctc command
CSR_WRITE(CSR_REG_MSTATUS, write_value);
```

Figure 21 Code example of CTC usage

To produce errors, we insert errors to main core's registers, this is achieved by inserting new hardware to register files to support validation. In the code we see that one cycle after CTC command is received in register file, all the registers are injected with errors.

Select	Write data	Cores supported
reset	32'b0	both
ctc_active	Error injection	Only in main core
Comperator mismatch	Data from shadow core	Only in main core
Write to reg	Wdata	both

Table 4 MUX description

```
always_ff @(posedge clk_i or negedge rst_ni) begin
  if (!rst_ni) begin
    rf_reg_q[i] <= WordZeroVal;
  end else if (ctc_active) begin
    rf_reg_q[i] <= ~(rf_reg_q[i]);
  end else if (IamMain & comperator_mismatch_i) begin
    rf_reg_q[i] <= rf_reg_shadow_i[i];
  end else if (we_a_dec[i]) begin
    rf_reg_q[i] <= wdata_a_i;
  end
end
```

Figure 22 MUX for CTC – ibex_register_file_ff.sv

5.1.2. Results

In the waveform we can observe how:

1. CTC signal rise
2. Register 27 value in main core changes form correct value: 2A00000… to faulty value: 55FFFFF…
3. Wrong register values causing output mismatch between the two cores, the mismatch is detected by comparator.
4. Restore system pushes the correct register values from the shadow core to the main core.
5. Outputs from both cores return to being equal.

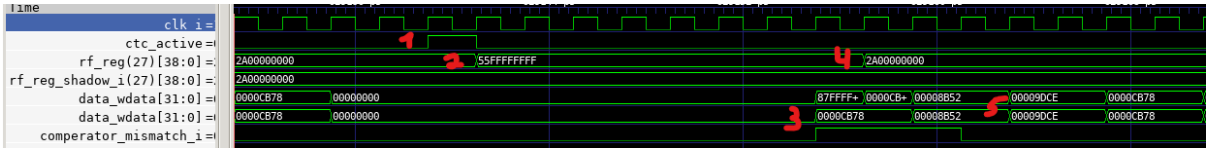


Figure 23 Simulation results

6. Conclusion

In this project we developed a system that leverages two cores within the boot sequence. The primary core retains its role in executing the boot program, while a shadow core is introduced to validate the execution of the main core during runtime.

During boot program execution outputs are compared between both cores. If a mismatch occurs, the shadow core continues the normal program execution. Shadow core also restores main core to correct state by overriding main core's register file with shadow's trusted register file. When the main core is restored with correct values, it continues boot execution.

For the implementation we used OpenTitan RoT project, which utilizes the Ibex RISC-V core. We used CSRs for interaction with software, implementing two commands: Compare command – to activate the comparisons with shadow core. CTC command – induces an error for debug purposes. We used Ibex's register files to enable main core restoration in case of mismatch detection. Finally, we used OpenTitan infrastructure for system integration and testing.

7. Future Research

For future research we plan to implement the system on an FPGA and evaluate the shadow core's performance in terms of power and area. In addition, we plan to Close timing and area problems that may be discovered on FPGA.

Additional work can be done to improve the system such as introducing a mechanism to decide which core (shadow/main) is correct in cases of mismatch. Furthermore, skew in timing between both core's executions can be used as an additional security feature.

8. Appendix

8.1. How to use

There are multiple ways to use OpenTitan software including simulation, FPGA and DV (commercial RTL simulators, such as VCS and Xcelium). Our project is simulation oriented, therefore we will explain setup step for this method only, for other methods please check OpenTitan documentation.

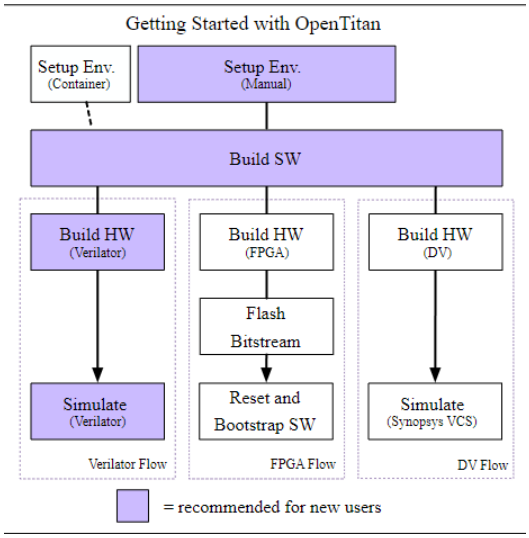


Figure 104 Different options for OpenTitan use.

Step 0: Check System requirements

OpenTitan installation requires Linux, preferably Ubuntu 20.04 LTS. At least 7GB available RAM (requires for Verilator simulations) and 50 GB space on disk (preferably 32GB of RAM and 512 GB of disk storage). In addition to download OpenTitan or our project, you will need a Github account.

If you encounter any problem during the installation, please go to the official OpenTitan installation manual.

Step 1: Ubuntu Installation.

Go to Ubuntu official website or alternatively open "Microsoft store" (on windows). Find Ubuntu 20.04 LTS, download and install it. After successful installation, run following commands in terminal:

```
sudo apt update

sudo apt upgrade
```

This will update Ubuntu software.

Step 2: Clone the OpenTitan repository.

```
git clone https://github.com/planetofwar/SecureBoot.git
```

Note: throughout the manual, you will be required to create an environmental variables by using "export" and "alias" commands, copy those commands to "~/.bashrc" and every time after launching Ubuntu, run "*source ~/.bshrc*" , this will save time.

```
export REPO_TOP=$PWD/SecureBoot
```

Step 3: Install dependencies using the package manager.

```
cd $REPO_TOP
sed '/^#/d' ./apt-requirements.txt | xargs sudo apt install -y
```

Step 4: Install Python libraries needed.

```
cd ..
python3 -m pip install --user -U pip "setuptools<66.0.0"
export PATH=~/.local/bin:$PATH
cd $REPO_TOP
pip3 install --user -r python-requirements.txt
```

Step 5: Install the lowRISC RISC-V toolchain.

```
cd $REPO_TOP
sudo ./util/get-toolchain.py
```

Step 6: Install Verilator

```
cd ..
sudo add-apt-repository -y ppa:ubuntu-toolchain-r/test
sudo apt install -y gcc-11
sudo apt install -y g++-11
export VERILATOR_VERSION=4.210
git clone https://github.com/verilator/verilator.git
cd verilator
git checkout v$VERILATOR_VERSION
autoconf
CC=gcc-11 CXX=g++-11 ./configure --prefix=/tools/verilator/$VERILATOR_VERSION
CC=gcc-11 CXX=g++-11 make
sudo CC=gcc-11 CXX=g++-11 make install
```

The make step can take up to 20 minutes.

```
export PATH=/tools/verilator/$VERILATOR_VERSION/bin:$PATH

verilator --version
```

If you receive “Verilator 4.210 2021-07-07 rev v4.210 (mod)”, verilator installation was successful.

Step 7: Installing Bazel (automatic way).

```
alias bazel="$REPO_TOP/bazelisk.sh"

cd $REPO_TOP

bazel test --test_output=streamed --disk_cache=~/bazel_cache
//sw/device/tests:uart_smoketest_sim_verilator
```

The last command will install the correct version of bazel, build, and run a single test with Verilator.

For other tests use the last command with a different test name but keep “_sim verilator” at the end.

Step 8: Using GTKwave, to observe test run wave form.

```
Cd

 ~/.cache/bazel/_bazel_smashkin/<random number created by
bazel>/execroot/lowrisc_opentitan/bazel-out/k8-
fastbuild/bin/sw/device/tests/aes_smoketest_sim_verilator.runfiles/lowrisc_opentitan

gtkwave sim.fst &
```

8.2. Bibliography

Opentitan:

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

Ibex:

https://ibex-core.readthedocs.io/en/latest/index.html

Secure boot:

https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-secure-boot

Fault injection:

https://ieeexplore.ieee.org/abstract/document/7774479
https://ieeexplore.ieee.org/abstract/document/8167709

Cryptographic signatures:

https://en.wikipedia.org/wiki/Digital_signature