DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

An in-hardware cycle-accurate benchmarking tool for security critical operations.

Julian Pritzi

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

An in-hardware cycle-accurate benchmarking tool for security critical operations.

Ein hardwarebasiertes, zyklengenaues Benchmarking-Tool für sicherheitskritische Vorgänge.

Author:	Julian Pritzi
Supervisor:	Prof. Dr.-Ing. Pramod Bhatotia
Advisor:	M.Sc. Harshavardhan Unnibhavi
Submission Date:	15.08.2022

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.08.2022 Julian Pritzi

Acknowledgments

I would like to thank Harshavardhan Unnibhavi for guiding me throughout the project and the whole process of writing this thesis.
Likewise, I am also grateful to Prof. Dr.-Ing. Pramod Bhatotia for the opportunity to work on this thesis and the welcoming inclusion at the chair.
I also thank all the staff, including Sophia Adelmeier, for their help with all the formalities and for providing a nice work environment.

I would also like to give special thanks to my family, in particular my parents and my sister, for their unconditional support. I am immensely grateful for the fact that I can always count on my parents, without them I would not be able to perform my studies.
Lastly, I thank all my friends and especially Jonas and Clemens for their emotional support and for keeping me motivated.

Abstract

Openly developed trusted computing platforms, like opentitan, provide a way to bootstrap security into a system, that can be inspected and audited by independent parties to guarantee the correctness of the platform itself. While security is the main focus of those projects, they also have an impact on the overall performance.

In that regard, this thesis introduces the design and implementation of a benchmarking tool running directly on hardware and performing cycle accurate measurements.

The tool is split into two mostly independent components linked by a single common library defining a communication protocol between the two. The first component is a bare metal application running on the hardware under test, where it sets up a runtime environment including a benchmarking suite before acting as a server application that is ready to receive benchmarking requests to be processed. The second component is an application running on a host PC that is connected to the hardware under test through a UART connection. It is responsible for interacting with the bare metal application to generate the results by reading and writing files on the host PC.

This thesis further provides an implementation for both components including a library that is targeted at the opentitan platform with the possibility to benchmark its accelerators in tasks related to hashing, random number generation, and aes encryption and decryption.

Running the benchmarks on the opentitan platform shows that the total computation time for sha2 and sha3 hashing, for an input of 256 bytes is less than 1174 and 1091 cycles respectively. Secondly, the platform was able to provide random numbers of 128 bits in length in 150 cycles. Finally, for encryption and decryption of 5x 128 bit blocks, using the AES encryption algorithms, the computation time is 491 cycles.

For both the hashing and the encryption this thesis shows that the CPU is a considerable bottleneck by not being able to read and write to and from the cryptographic accelerators fast enough, to fully utilize its potential. In addition, some accelerators may require the CPU to wait for a flag to be set which can cause significant overhead if performed frequently.

This thesis then concludes that given the above results, performance optimizations for the analyzed version of opentitan are best targeted at the CPU, in particular the load, store and branch instructions.

Contents

1 Introduction

Security is a vital part of almost all modern applications. While a major part of security research analyzes flaws in existing systems that can be abused, it is harder, maybe even impossible to prove the opposite, that an application is completely secure. Ignoring everything else, simply guaranteeing that some code is actually doing what it claims is hard. There is always a lower layer, be it some APIs, syscalls or the individual CPU instructions, that have to be trusted to actually do what they claim to do, nothing more, nothing less.

One attempt to battle this is trusted platform modules [15] (TPMs), specialized hardware components that use a chain of trust to guarantee their integrity. This reduces the implicitly trusted components of a system and provides a certificate that a given platform is implemented in accordance with the manufacturer. Because most of today's hardware development is closed, the manufacturer must be trusted to document all interfaces and capabilities properly.

The fact that there exist multiple accounts of this not being the case [8, 22, 12, 20, 7], suggests that current systems may be flawed.

While hardware is still mainly developed in closed form, developments in recent years bring the philosophy of open source software ever closer to hardware development. One of the main benefits is the fact that independent parties are allowed to inspect and audit the hardware. The transparency allows users of the hardware to confirm that it works as expected. One openly developed hardware project in this field is opentitan [27]. While the security of opentitan is already greatly covered [24, 11, 37, 23] this thesis focuses on its performance aspects. In particular the performance of different components of which the opentitan platform is composed.

While a simulation of individual parts allows performing a detailed analysis of a component under test, this ignores the potential performance impact that the interactions between components can have. Although it is possible to scale the simulation up to include more components and potentially the entire system, this comes with a big increase in complexity and simulation time. The correctness of the result also depends on the correctness of the simulator, any bugs or wrong assumptions made by the simulator can lead to corrupted results.

There is, therefore, value in performing benchmarks as close to the final real-world use case as possible and that entails running on actual hardware in the conditions that are desired for the benchmark.

Therefore, this thesis introduces a benchmarking tool that runs on the platform under test and measures the cycles for arbitrary operations. These can include code running on the CPU or interactions with accelerators and other peripherals.

As part of this thesis, the open source project opentitan is benchmarked for different security operations that guarantee the following properties:

1. Confidentiality: Data should only be accessed by authorized entities. Hence, all communication over all channels has to prevent unauthorized third parties from deciphering the exchanged messages. If an untrusted channel is used for communication, then confidentiality is guaranteed using encryption. Sufficiently strong encryption guarantees the confidentiality of the transmitted data at the cost of causing performance overheads for the sender as well as the receiver in the form of encryption and decryption times.

2. Integrity: Unexpected data modifications or transmission faults have to be detectable, this is achievable by introducing redundancy which can be used to verify the integrity of the data. A common form of redundancy is hash digests. Hash functions ensure that a small change in the input data causes a drastic change in the output, thereby making it hard for attackers to modify a message in a way that the hash digest remains the same. This also makes it unlikely that simple transmission faults produce an altered message with the same hash digest. One trade-off is that computing a hash value for verification can impact the overall performance, especially if checking large amounts of data for their integrity.

3. Authenticity: It has to be possible to verify that data has not been altered since it is generated by a specific entity. One standard way to do this is by using cryptographic signatures. Signatures use asymmetric encryption to sign a piece of data using the private key. At a later point using the public key of the signing entity, it is possible to verify the signature. These signing and verification operations have an impact on performance, especially on low-power devices.

4. Freshness: It has to be possible to check that a received message is fresh, in particular, to prevent replay attacks. When communicating over an untrusted channel an attacker can record and replay any messages. To prevent this it should be possible to use ephemeral data that changes for every request. A potential performance overhead lies in the generation of such, often random data.

The opentitan project provides functionality to achieve these properties in the form of dedicated hardware or custom software libraries. These are benchmarked using the custom benchmarking tool with the results listed in chapter 6.

The evaluation of these benchmarks shows that load, store and branching CPU operations limit the performance of the faster accelerators like the hashing and encryption accelerators. This holds true especially when the accelerator implements an internal queue or supports 2 stage pipelining. While in a purely sequential execution the cycles of every step add up to the overall performance, this is not true for such asynchronous implementations, where the slowest sequence determines the overall performance exclusively.

Thus this thesis concludes that to improve the overall performance of those security operations it is necessary to reduce the cycles that load, store and branch operations take or modify the accelerators to support another way to access the required information.

2 Overview

2.1 Design Goals

The benchmarking tool adheres to the following design goals:

1. Flexibility: The whole system, especially the suite should be flexible and easy to adapt to new platforms, additional cores and accelerators. This decision led to the *platform abstraction* to describe the target platform the suite is running on and using compile-time flags to compile for different platforms. The setup should make it easy to introduce new platforms which potentially share accelerators with existing platforms. This should allow for the benchmarking suite to be run on any RISC-Vplatform with minimal changes.
2. Accuracy: Since the tool potentially measures functions or accelerators that only take a few cycles to execute, it is necessary to reduce the number of cycles spent on working with abstractions that a high-level language may introduce in order to make the application more general. An example of this in Rust is using dynamic dispatch to realize polymorphism, which for more accurate results should be avoided between cycle measurements as much as possible. This is because dynamic dispatch is realized as a pointer to the actual data structure, and accessing it involves resolving pointers which can take up some extra cycles.
3. Reliability: When developing low-level code on a bare metal target invalid pointers or accesses to the bus can have cascading effects that potentially alter measurements in unexpected ways. Increased reliability of the code is guaranteed by Rust's memory safety, in addition, any unsafe functions have a documentation comment explaining the assumptions they make.

2.2 System Overview

The benchmarking setup contains two major physical components. On one side the field programmable gate array (FPGA) board which realizes the platform under test, this thesis uses a NexysVideo board as the FPGA. On the other side, there is the host PC that manages the FPGA and controls the benchmarking process. The two components are connected by a UART channel, on the FPGA. This may be realized via dedicated UART pins or a micro USB port. The host PC is expected to run a Linux operating system where, if set up correctly, the connection is accessible as a TTY Port.

The benchmarking tool is split among three major components:

1. The benchmarking suite which is running on the target platform to be benchmarked. It contains all the functionality to perform benchmarks of individual components. The role of the suite is similar to a server, it starts up and awaits messages that are sent over the UART in an endless loop. Every message is processed individually by performing their associated benchmarks or control actions and then replied to with a response message.

2. The benchmarking CLI (Command Line Interface) is running on a host PC. It is responsible for establishing a valid TTY connection and communicating with the benchmarking suite. The CLI reads files describing the benchmarks that should be performed and converts them to messages that are sent to the suite over the UART. The CLI then captures any responses received from the suite and writes the results back to files on disk on the host.

3. The *common library* is responsible for defining the valid messages that can be sent over the UART communication channel. This library also includes functions for serializing and deserializing of all the messages sent between the CLI and the suite. Both the suite and the CLI integrate this library and directly use the features provided so that any change to the common library immediately affects both the suite and the CLI.

2.3 System Workflow

This thesis additionally describes the setup of the target platform. In the specific case of the opentitan project, this means downloading the project, synthesizing the Verilog files and producing a bitstream for the target FPGA. The bitstream also has to be flashed onto the FPGA before the system is ready for software to run on it. This step may have to be done multiple times for all build configurations of the platform that is benchmarked. In the opentitan case, this would mean that if the impact of masking of cores will be measured then both an unmasked and a masked version of the project have to be synthesized into a bitstream and flashed separately.

Figure 2.1 shows the high-level workflow of the benchmarking tool. In particular, the suite has to be potentially rebuilt with a custom feature set depending on the

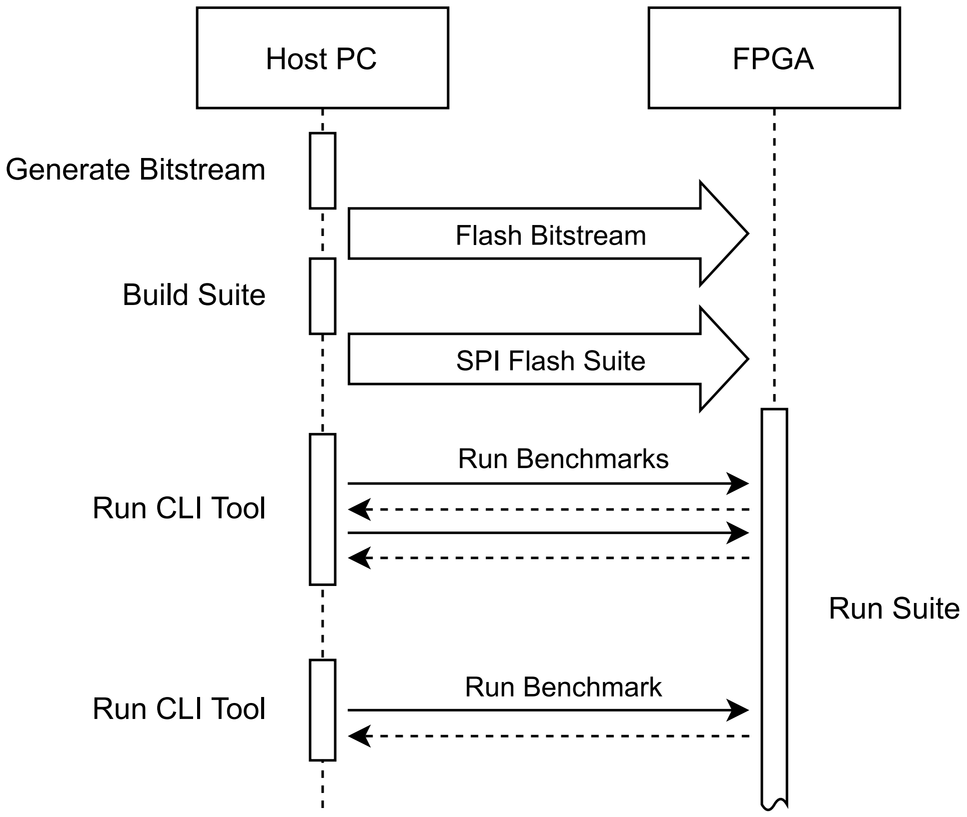


Figure 2.1: Workflow showing the individual steps of the benchmarking tool.

bitstream that was flashed. After flashing the suite the workflow remains the same independent of platform and bitstream since it is the job of the suite to provide a uniform interface to the CLI. This means that the suite decides at compile time which module implementations are used and has also the possibility to return an empty benchmark when no benchmarking result could be obtained if a module is not supported.

The flow of information can be seen on Figure 2.2. The CLI Tool is responsible for reading and writing to files on the host system and uses the common library for the communication over the system's TTY port to the suite. The suite itself contains the communication module, which allows for reading and writing over the UART connection. The common library, which is used for the serialization of the messages. The benchmarking functionality, which includes different functions and datasets that realize a benchmark and control the platform's modules as well as perform the required measurements to send the results back to the CLI Tool.

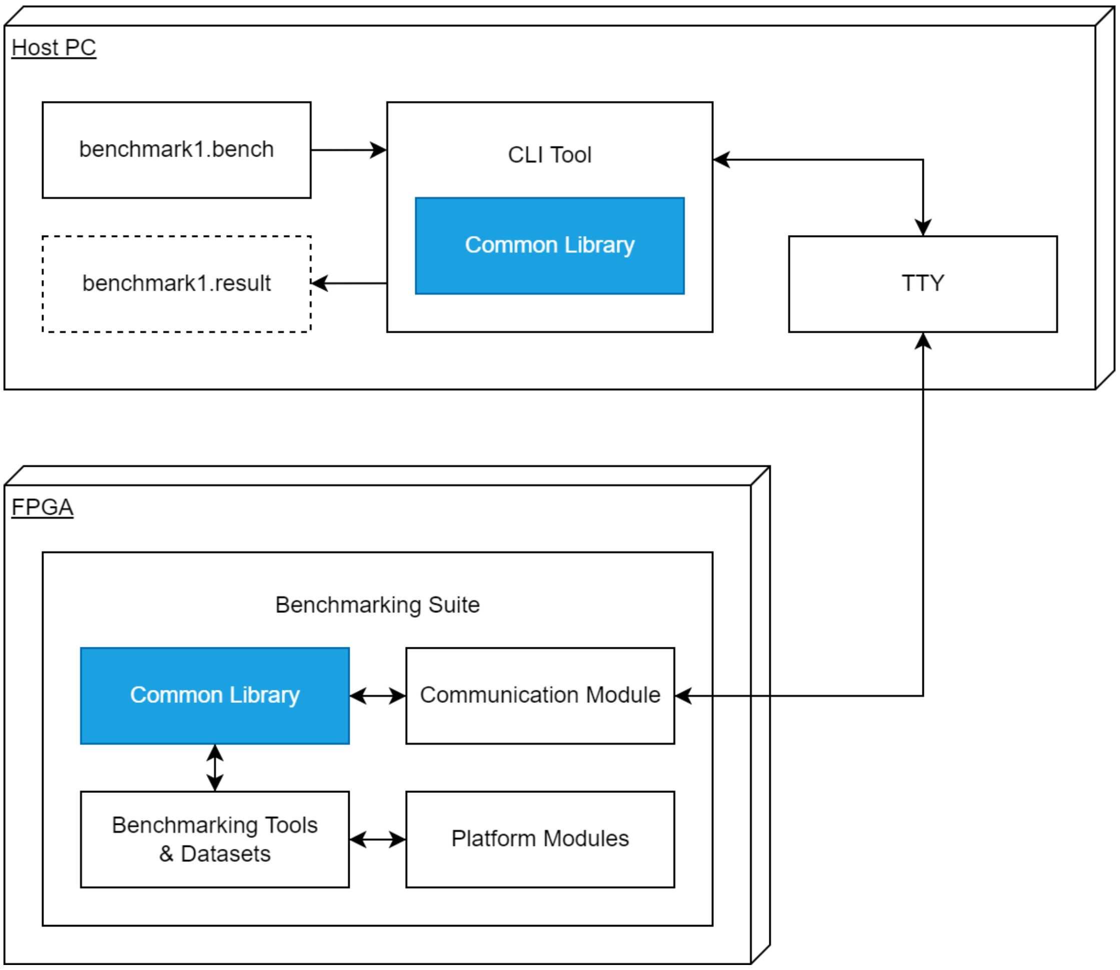


Figure 2.2: Information flow of the different components

3 Background

This thesis uses a series of different tools and projects to develop, build and run the benchmarking tool. The project underlying this thesis is written in Rust and makes use of cross-compilation as well as bare metal development features of the language. Dependency management for the software packages, called "crates", is done with the Rust tool Cargo.

As a platform under test, the opentitan project provides Verilog files that describe the hardware IP and can be used to program an FPGA. This allows for the quick and flexible realization of hardware components.

The common interface between the software written in Rust and the hardware is the RISC-V instruction set architecture (ISA). Thus in order to benchmark the platform to the best degree using a software tool running on the platform the RISC-V ISA has to be understood to achieve measurements that are as accurate as possible.

Reproducibility of all artifacts and results of this thesis is guaranteed by using nix-shell to provide a common development environment, including scripts that automate the process.

3.1 Bare Metal Development

The Rust management tool Rustup allows for easy installation of several toolchains that can be used to cross-compile Rust code. An important factor for this to work is that the Rust project has sufficient support for the target platform. After installing the desired toolchain and specifying it when compiling, Rust is able to cross-compile without any additional changes needed.

Not all target platforms of Rust have equal levels of support as listed in [39]. In particular, many bare metal targets most likely do not have support for the Rust standard library. This is partly because the standard library depends on functionality provided by an operating system, which among other things includes a way to dynamically allocate memory.

Because a lot of the standard libraries' data structures would only need a way to allocate memory, the standard library contains the alloc crate which, given an implementation for an allocation function, provides this set of data structures. The alloc crate is part of the standard library but can also be included separately on bare metal targets as long as an allocation implementation is provided.

Also relevant for bare metal Rust development is the startup or bootstrapping code that sets the platform up so that the Rust code can run. This usually includes setting up the stack, which is necessary for the function call structure. After this, it is possible to call the first Rust function that usually continues the startup by clearing the block starting symbol (bss) section, which is part of the code that may contain static uninitialized variables and is expected to be zeroed. Lastly, the runtime has to initialize the data section by loading its contents from the potentially write-protected memory where the code is stored to a location that allows for reads and writes. For this Rust's embedded devices working group provides crates that implement this functionality for different target platforms. Examples for such crates include cortex-m [4] for ARM devices, riscv-rt [31] for general RISC-V targets or msp430-rt [25] for MSP430 microcontrollers.

3.2 Opentitan

Opentitan [27] is the first open-source silicon root of trust (RoT). The project is managed by the lowRISC foundation and is backed by multiple companies like Google and Western Digital. As the name might suggest it is the open source extension of Google's internally developed Titan Chip. The opentitan project is a combination of hardware and software components which together compose a Root of Trust (RoT) platform.

3.2.1 Hardware IP cores

The hardware IP cores of the project are mainly written in SystemVerilog. They are structured into individual subfolders of the repository, each being mostly self contained. During synthesis, individual cores are connected through a TileLink Uncached Lightweight (TLUL) [35] bus to form the opentitan platform.

For interoperability, all IP cores of the opentitan project conform to a custom standard called "Comportable Design". The Comportable Design standard is defined by the opentitan project itself and mainly regulates the interface of the cores. To conform to this standard an IP core is required to have the following:

TLUL bus client connection, that allows the IP core to connect to the common bus so that other cores have a uniform way to interface with it. **Register Interface**: these registers are used for control and communication by both other cores and software. **Clock Dependency**: this specifies the clock that is used to synchronize any device interfaces.

Additional interfaces can optionally be implemented.

3.2.2 Earl Grey

The Earl Grey platform is a predefined combination of IP cores connected through a bus, it also includes a CPU and memory devices. This platform can be synthesized for the supported FPGAs and works "out of the box" as a general purpose RV32 computer. The registers of the included IP cores are mapped in memory (MMIO) and can therefore be accessed by software running on the CPU.

The earl grey platform at the time of writing this thesis only officially supports the ChipWhisperer 310 FPGA board. In particular, the support for the NexysVideo board was dropped. For this thesis, since it was not possible to acquire a ChipWhisperer board the NexysVideo board was used instead. This results in this thesis using the last tagged release of opentitan that supports the NexysVideo board, which is earlgrey_silver_v5 (June 2021). There are substantial differences between that version and the current opentitan repository. Some of the changes that are relevant to the benchmarking tool are the differences in register offsets, the lack of address translation and changes in the project structure.

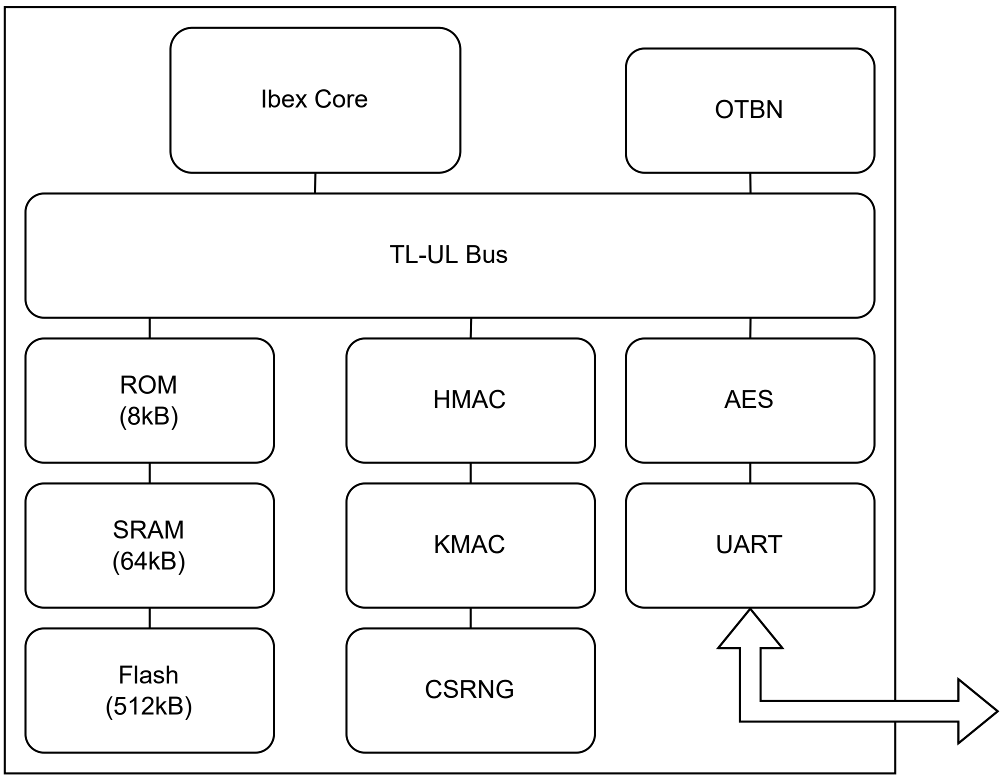


Figure 3.1: Overview of the earl grey platform (version earlgrey_silver_v5)

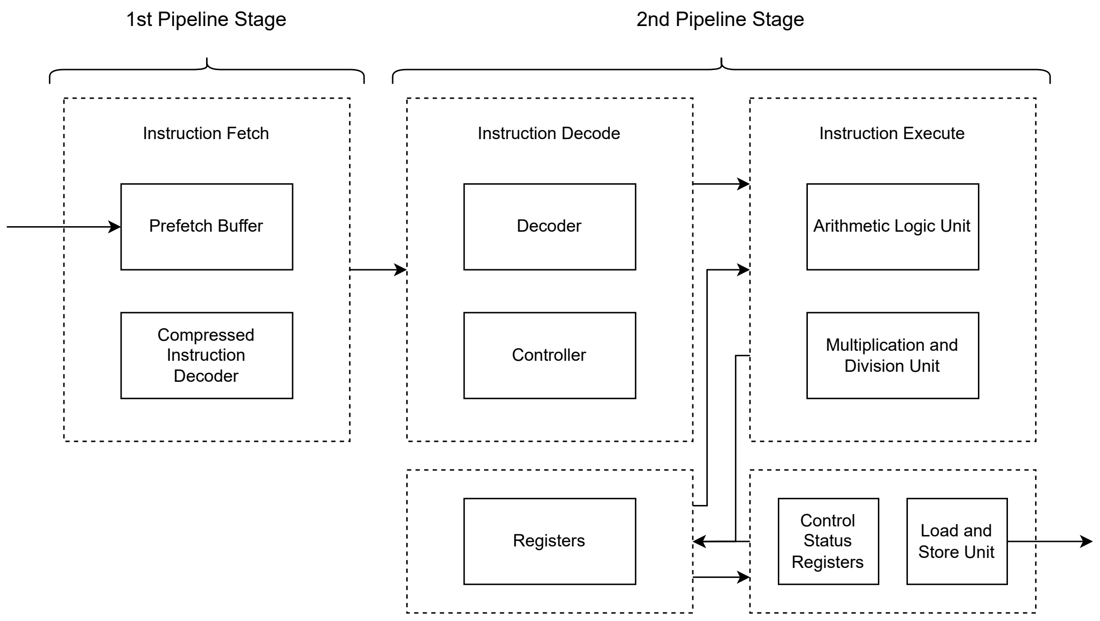


Figure 3.2: Simplified diagram of the Ibex cores 2 stage pipeline.

Ibex Core

The Ibex Core [17] is an open source general purpose, 32bit RISC processor, developed in SystemVerilog. Although not directly part of the opentitan project and instead included as a dependency, it is also managed by the lowRISC foundation (similar to OpenTitan). The ISA is RV32IMC, notably, this means the core does not support atomic instructions. While these are not necessary because of the single hardware thread (called hart in RISC-V), the linked_list_allocator library [21] that is part of the benchmarking tool requires the presence of atomic instructions to support the default feature set.

The version of Ibex that is present in the earlgrey_silver_v5 version of opentitan has a single hardware thread (hart) executing instructions in order in a two-stage pipeline, as shown in Figure 3.2. The first stage is responsible for fetching the instructions while the second stage decodes and executes the instruction.

The Ibex core can execute one instruction in one cycle in the ideal case. Some operations take up multiple cycles as explained in [16]. Table 3.1 is a list of a selected number of instructions and the number of cycles they take on the configuration of the Ibex core that is present in the earlgrey_silver_v5 version of opentitan.

Instruction Type	Cycles
Integer Computation	1
Control Status Register Access	1
Load or Store Operation	2 - N
Branch (Not-Taken)	1
Branch (Taken)	3 - N

Table 3.1: Numbers of cycles for instructions on the Ibex core, adapted from [16].

Uart

Opentitan's uart is an I/O IP core that implements general uart functionality. It has an external interface and both a sending and a receiving FIFO queue for communication over it. The uart is configured as 8N1, which stands for 8 data bits, no parity bit and 1 stop bit. The baud rate of the uart is set using the NCO field of the control register, the NCO value is computed using both the baud rate and the frequency of the peripheral clock.

Hmac

The hmac IP core is used for message authentication code (MAC) generation. It supports 256bit keys and uses SHA2 for computing the hash over some input data. The core can optionally be configured to only compute the SHA2 hash digest. Input data is inserted in a 16 x 32bit message queue.

KMAC

Like hmac, the kmac core is used to generate MACs. Instead of SHA2 it uses SHA3 as its hashing algorithm. Since SHA2 is susceptible to length extension attacks [3] due to its underlying Merkle–Damgård construction, SHA3 serves as a successor that addresses this issue. For the process of selecting the SHA3 hash function, a competition was held which was won by the Keccak hash function [9]. The default Keccak function is expected to be slower than SHA2 when implemented in software [5], but when implemented as accelerators Keccak is expected to outperform SHA2 [19].

The kmac core supports a number of different hash modes: SHA3-224, 256, 384, 512 as well as SHAKE-128, 256 and also cSHAKE-128, 256. Key lengths can also vary with support for 128, 192, 256, 384 or 512 bits. Input data is inserted in a 9 x 64bit message queue. Additionally, the IP core's Keccak permutations are masked in order to protect against 1st order side channel attacks (SCA), which are attacks that only analyze a single source of side channel information eg. differential power analysis. Higher order SCAs use multiple sources simultaneously instead, by varying the measurement techniques and temporal offsets [18]. To protect against nth order SCAs using masking, the sensitive information is split into $n+1$ shares [14]. This means data is internally represented in two shares which when combined using the xor operation produce the actual result. This separation makes a power analysis side channel attack (SCA) of the IP core more costly as described in [13]. Masking of the core can optionally be disabled using a flag inside a configuration file of the opentitan project. While improving security against SCAs masking also negatively impacts the performance [14], it can therefore be beneficial to disable masking if security is not required or other security measures sufficiently protect against SCAs.

Aes

The aes core is an accelerator for aes en/decryption. It natively supports the following modes of operation: Electronic Codebook (ECB), Cipher Block Chaining (CBC), Cipher Feedback (CFB), Output Feedback (OFB) and Counter (CTR) mode. The core supports the key lengths 128, 192 and 256 bits, which can either be supplied by writing to the registers of the core or by sideloading them from the keymanager. Like the hmac core, the aes core is also masked against 1st order SCAs by default but this masking can again be disabled in the core's configuration files of the opentitan repository. When disabling masking the default S-Box does not work anymore so an alternative to the default implementation has to be selected, two alternatives are already included in the project and can be selected: Canright [2] or a lookup table (LUT) based S-Box. The aes module is operated either manually or automatically, the former requires writing to a trigger register for the core to start computing. The latter automatically starts the computation once new data is written to the input register.

CSRNG

The cryptographically secure random number generator (CSRNG) core, is used to generate random data from an initial seed. The core can optionally load the seed from an entropy source which allows for true random number generation (RNG). Communication with the core differs for hardware and software, the entropy distribution network (EDN) core was specifically designed to make it easier for hardware to work with the CSRNG core. Software can use the core's registers to communicate using a special protocol.

The protocol consists of exchanging 32bit message blocks. The first block is always a header message, containing information about the command to perform, the number of blocks that belong to this header, a list of flags and depending on the command the number of data blocks the CSRNG core should generate.

Opentitan Big Number Accelerator

The Opentitan Big Number accelerator (OTBN) is a cryptographic coprocessor specialized for computations involving big numbers that usually do not fit in a 32bit or 64bit register. In particular, the core includes 32 x 256bit data registers while operating using regular 32bit registers for control. These wide registers are useful for computations like asymmetric encryption where key sizes tend to be large. To operate the OTBN, custom assembly code has to be written for it. This assembly code uses an adaptation of the RV32 ISA, implementing a subset of the instructions with some additional ones for controlling the wide data registers. This custom code has to be loaded into special code registers during runtime by the platform's main CPU. The core can then be triggered to execute whatever instructions were previously inserted. OTBN has direct access to the csrng core through a special control status register (CSR).

3.2.3 Software

The opentitan project provides software implementing secure boot as defined by the project:

The boot process can be divided into three stages MASK_ROM, ROM_EXT, BL0. Each stage reads the manifest of the next stage, the manifest is a collection of metadata information about the following code, it includes the entry point of the executable, a signature used for secure boot and a list of parameters that are used for the generation of the signature. After reading the manifest the next boot stage is verified by computing a hash and checking the signature before enabling the execution of the next stage and delegating execution.

Because the MASK_ROM is in ROM and can therefore not be changed its tasks are kept minimal: first, it sets up a C runtime (CRT) by disabling interrupts and clearing registers, setting up the stack and preparing the memory. In the next step, the secure boot code written in C is executed, which performs the verification and jumps to the next boot stage as described earlier. The next stage is ROM_EXT, which is located in Flash instead and therefore allows firmware updates. The ROM_EXT may perform different boot services, like attesting the device state or managing the keys used during the secure boot process, before jumping to the BL0 stage, which is the first stage to run user defined code.

For the benchmarking suite secure boot was not relevant. For this the opentitan project also provides a test ROM, that can be used. It only reads the entry point field of the manifest and does not perform any validation before jumping to it.

The opentitan project also provides crypto libraries that use OTBN to accelerate their computations. One such library is an implementation of ECDSA [10] signing and signature verification using the P256 curve. Similarly there also exists a library for verifying RSA-3072 signatures. Unfortunately, these libraries are not present in the earlgrey_silver_v5 version of opentitan.

3.2.4 Verification

All opentitan IP cores are verified either through dynamic simulations with functional tests or through formal property verification (FPV). Dynamic simulations are performed conforming to the Universal Verification Methodology 1.2 standard [40] in combination with SystemVerilog based verification [6]. Verification is part of the opentitan hardware development stages, these stages mark milestones that have to be completed until an IP core is considered to be signed off. These stages are listed in [29], in particular for the verification part the following stages are defined:

- V0: Initial Work, includes writing a testplan and deciding on the testing methodology.
- V1: Under Test, requires a completed testplan and a partially completed testing or formal verification setup.
- V2: Testing Complete, requires a completed testing or formal verification setup, with all high-priority bugs addressed.
- V2S: Security Countermeasures Verified, requires all tests for security countermeasures to be present.
- V3: Verification Complete, requires all bugs to be addressed and a 100% coverage of tests or verifications.

At the time of the earlgrey_silver_v5 version of opentitan, most cryptographic accelerator cores like aes, csrng, otbn and kmac where in stage V1. In the most recent versions, some cores like aes, otbn and kmac progressed to stage V2, but neither are signed off yet.

3.3 RISC-V

RISC-V [41] is an open RISC ISA consisting of specifications for 32bit and 64bit systems. In addition to the size of the registers, the supported extensions are a defining factor for a RISC-V architecture. Extensions are usually abbreviated by a single letter and listed when specifying the system, a selection of extensions is listed in Table 3.2. The extensions I, M, A, F, D, Zicsr, Zifencei, which are explained in Table 3.2, are commonly grouped into a single symbol G if all are present, to signal that they are considered to provide a general purpose ISA.

Symbol	Description
I	Integer base instruction set
M	Integer multiplication and division instructions
A	Atomic instructions
F	32bit floating point numbers instructions
D	64bit floating point numbers instructions
Zicsr	Control and status register instructions
Zifencei	Instruction-Fetch fence instruction
C	Compressed instructions
B	Bit manipulation instructions

Table 3.2: A selection of RISC-Vextensions.

The RISC-V ISA specifies three different privilege modes in which code can run, these modes are stored in a control status register (csr) at runtime. Privilege modes are changed using traps, which when caused are usually handled by a trap handler running in a higher privileged mode. The supported modes are, sorted by decreasing privilege:

1. Machine mode (M), the machine mode is the only required privilege level.
2. Supervisor mode (S), intended for operating systems code and only supported if user mode is also supported.
3. User mode (U), intended for application level code.

Like the current privilege level, other information like the time, an instruction counter, the version of the ISA, the id of the hardware thread and memory protection regions are stored in CSRs. Useful for benchmarking is the set of performance counter CSRs that automatically increment on certain conditions. The majority of these counters are defined by the CPU implementation and have to be manually enabled, but two are defined in the RISC-V standard: the cycle and instruction counter CSRs. The cycle CSR in particular is important for this thesis but because the values inside the counters are defined to be 64bit this leads to some caveats when running on the 32bit Ibex core that this thesis targets. The resulting limitations are discussed in the rest of this section.

RISC-V cycle CSR allows to read the current cycle count of the CPU, this feature is fundamental for the benchmarking tool. The single drawback arises due to the size of the registers, because the cycle is stored as a 64bit value, the cycle CSR is split into a cycle_lo and cycle_hi register on 32bit architectures, containing the lower and upper 32bits respectively. To read the full cycle count additional checks have to be made because the lower register could overflow while the higher one is read. Therefore the ISA specification suggests code similar to the one in Figure 3.3 for reading the correct cycle count. This snipped has to execute 8 instructions in the worst case and 4 instructions in the best.

In particular, if the cycles of some code are to be measured using the cycle CSR and the provided snippet by computing the difference of two cycle reads the following has to be considered:

In the first readout after reading the final `mcycle` register two instructions are executed:

1. Reading `mcycleh`
2. A branch which is not taken.

Additionally in the second readout before reading the final `mcycle` there are two possible scenarios:

1. One `mcycleh` read instruction is executed prior.
2. A full readout loop plus one `mcycleh` read instruction is executed prior.

This results in any difference measured using the provided snippet having the best case overhead of 3 CSR reads and 1 branch not taken and the worst case overhead of 6 CSR reads, 1 branch not taken and 1 branch taken.

When using Table 3.1 to get a concrete number of cycles of overhead this results in the best case, an overhead of 4 cycles and in the worst case, an overhead of more than 10 cycles.

```
1: csrr t0, mcycleh
csrr t1, mcycle
csrr t2, mcycleh
bne t0, t2, 1b
```

Figure 3.3: Assembly code to read the cycle control status register adapted from [41].

4 Design

4.1 Suite

The suite is intentionally designed to be highly flexible and this is achieved using different abstractions for the underlying hardware. The two most important abstractions are *modules* and *platforms*. A platform is the abstract representation of the hardware the suite is running on. It provides the suite with module implementations as a way for the suite to interact with the platform specific components. Because such components may be shared by different platforms, platforms mainly combine a set of components and map them to memory addresses for MMIO or provide them with platform specific configuration values like clock frequencies.

4.1.1 Platform

The purpose of the platform abstraction is to allow the suite to run and interface with any supported hardware by simply combining a set of module implementations. It should especially facilitate adding new or altering existing platforms without rewriting the whole suite. Individual platforms are isolated so that the code of one platform does not have any effect on the other platforms. A platform has to decide which module implementations should be used for sha2 hashing, sha3 hashing, AES and RNG, it is also possible to not support any of those modules. The only required module implementation is the communication module. Platform selection is done through cargo feature selection at compile time.

Each platform additionally has a linker script describing the memory layout of the platform and allowing for prepending platform specific data or startup code before the suite. The following are the currently supported platforms of the suite.

Qemu Virt

The qemu virt platform is a minimal platform with only support for a communication module. This platform was mainly used at the beginning of the development to quickly check the correctness of the runtime, asserting that the abstractions work, as well as checking platform agnostic functionality like heap allocations and test execution.

Earl Grey

The Earl Grey platform is the main target of this work and corresponds to the opentitan chip with the same name. The platform combines implementations and information like memory addresses for all supported modules. Because the opentitan boot sequence requires the existence of a manifest containing metadata, the platform also needs to prepend this information to the code of the suite. This is done using a custom section in the linker script and the inclusion of an assembly file containing the required manifest data.

4.1.2 Modules

A module is a description of an interface to some functionality. This functionality can then be implemented by different components of the platform the suite is running on. Most functionality is specific to the type of module but all modules have initialization code that should be run exactly once at the startup of the suite to set up any requirements of the module's implementation.

Communication Modules

Communication modules are a combination of the byte read and the format string write modules. Anything that can both read and write bytes can be used as a communication module. The communication module of a platform represents the primary way the suite should communicate with the CLI tool. The communication module is therefore used to send and receive the serialized strings representing the messages that are sent between CLI tool and suite. This module is not intended to be benchmarked. The communication module serves for low-level transmission. On a higher level, the common library is responsible for defining the structure of the messages exchanged. The only limitations set by the communication module are that every protocol that uses it is using ASCII characters and is line based.

Both the qemu virt and the earl grey platform provide implementations for this module, each one using their respective uart.

Hashing Modules

The hashing module is used for performing general hashing, it is up to the implementation to determine what hashing algorithm is used as long as the digest is 256bits in size. The module defines several functions listed in Figure 4.1, which implementations of the module have to realize. The following is the expected usage of the hashing module:

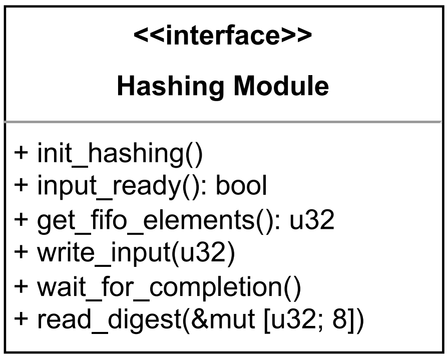


Figure 4.1: Hashing module modeled as an UML interface diagram.

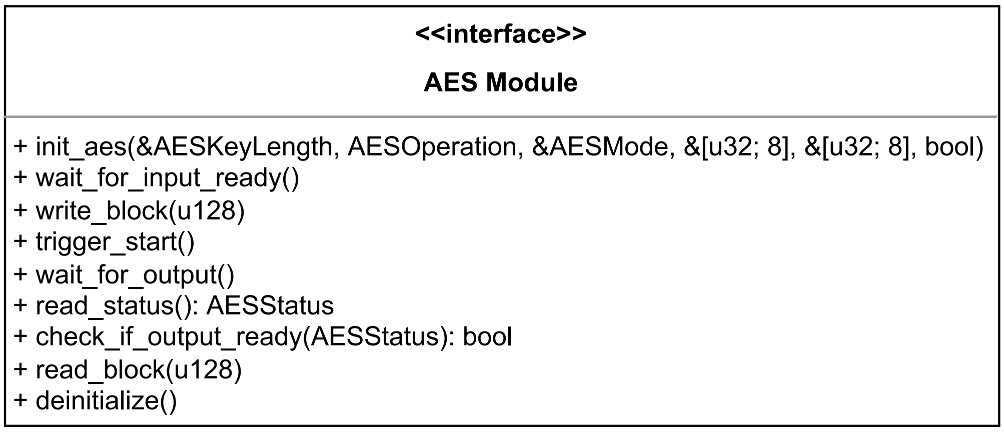
1. The `init_hashing` function is responsible for resetting and setting up the hashing module for a new hash computation and has to be called before starting a new hash computation.
2. Before starting to insert data to compute its hash, it is first necessary to ensure that the module is ready for input. This can be done in two ways:
   - Using `input_ready` which returns true if more input can be inserted.
   - If the internal queue size of the module implementation is known, by calling `get_fifo_elements` and comparing it to the maximum size of the queue.
3. Once it is determined that the module is ready for more input, the `write_input` function can be used to input the next 32bits of data.
4. Steps 2-3 can be repeated until all data is provided to the hashing module.
5. To start and wait for the computation of the final digest the `wait_for_completion` function has to be called once.
6. Once ready, the hash digest can be read into a buffer using the `read_digest` function.

The earl grey platform provides an implementation for this module in form of the hmac or kmac IP core. It is possible to obtain a reference to these implementations by calling `platform::current().get_sha2_module()` for the hmac implementation or `platform::current().get_sha3_module()` for the kmac implementation. The references implement this interface and can hence be used by utilizing the functions defined by it.

AES Modules

The aes module can operate in a number of different aes encryption and decryption modes. The list of modes it operates in is shown in Figure 4.2. The following is the expected usage of the aes module:

1. The process of encrypting or decrypting data using the aes module always starts by calling `init_aes` with the required configuration information. This includes specifying the key length by choosing from 128bit, 192bit and 256bit; declaring whether the module will be used for encryption or decryption; selecting an aes mode from Table 4.1 including an IV if required; providing the encryption key in two shares and lastly specifying whether the module will be operated manually or should start encryption automatically.
2. It should then be asserted that the module is ready to receive the next block using:
    - `wait_for_input_ready`, which returns once the module is ready for more input.
    - Alternatively, a non-blocking way to check for the output to be ready is by first reading the status of the module using `read_status` and then checking if the status signals that the output is ready using `check_if_output_ready`. The separation into two functions is deliberate to minimize the cycles spent reading the status, as it might be desirable to check the status at a later time after reading it.
3. Once ready, `write_block` is used to write the next 128bit block.
4. A module in automatic mode will now start the en/decryption of the block, in manual mode, this has to be triggered by calling the `trigger_start` function.
5. To wait for the module to finish the en/decryption of the block the function `wait_for_output` should be called, once the function returns this implicitly also signals that new input can be provided to the module.
6. Once the output is ready it can be obtained using the `read_block` function. These steps can be repeated until all blocks are processed.
7. To encrypt or decrypt multiple blocks steps 2-6 can be repeated.
8. When finished the module should be reset by calling `deinitialize`.



```
+ init_aes(&AESKeyLength, AESOperation, &AESMode, &[u32; 8], &[u32; 8], bool)
+ wait_for_input_ready()
+ write_block(u128)
+ trigger_start()
+ wait_for_output()
+ read_status(): AESStatus
+ check_if_output_ready(AESStatus): bool
+ read_block(u128)
+ deinitialize()
```

Figure 4.2: AES module modeled as an UML interface diagram.

AES Mode	IV	Description
Electronic Code Book (ECB)	no	Blocks are en/decrypted indepedetly
Chiper Block Chaining (CBC)	yes	Chiphertext of previous block is xor'ed with plaintext of next block before encrypting
Cipher Feedback (CFB)	yes	Ciphertext of previous block is encrypted and then xor'ed with plaintext of next block to generate chiphertext
Output Feedback (OFB)	yes	IV is encrypted repeatedly and xor'ed with plaintext to genereate cipertext
Counter (CTR)	yes	$\mathrm{IV} + n$ is encrypted and xor'ed with plaintext to create the nth ciphertext

Table 4.1: List of modes that the AES module supports.

RNG Modules

The rng module can be used to generate pseudo or true random numbers it has the simplest interface of all the introduced modules as can be seen in Figure 4.3. The following is the expected usage of the rng module:

1. The `init_rng` function is called prior to the first generation and whenever the seed should change, it accepts an optional slice of seed material. If present the module uses the provided seed material to seed the pseudo random number generation, the module may only use the first n elements of the slice where n

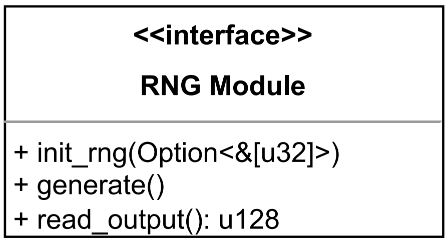


Figure 4.3: RNG module modeled as an UML interface diagram.

depends on the maximum seed size of the rng implementation. If no seed vector is provided the rng module should instead produce a truly random number. This is different from an empty vector which corresponds to the seed material 0.

2. After initialization the modules `generate` function can be called to generate 128bits of random data.
3. The `read_output` function can be used after the `generate` function terminated to read the generated bits
4. Steps 2-3 can be repeated for as long as new random material is required.

4.2 CLI Tool

The CLI tool is the counterpart to the suite and runs on any host that wants to perform benchmarks. The CLI tool is just an example of how an application could look like, that interacts with the benchmarking suite, as any program using the common library can be used to control the suite.

When connecting the FPGA running opentitan and the suite to the host PC it is possible to connect and interface to opentitans UART via a TTY port exposed on the host PC. The CLI tool requires a path to this TTY port as an argument, in addition to the port, a list of benchmark description files is passed.

For every file passed to the CLI, the tool first establishes that a suite is running on the other side of the TTY port and then exchanges messages to perform the requested benchmark. For this, it uses the common library to ensure all message types and their serialized forms are synchronized between the CLI and the suite.

4.3 Common Library

The common library is used to serialize and deserialize all the messages sent between the CLI tool and the Suite running on the FPGA. To accomplish this all valid messages are defined in a respective enum, one for CLI to Suite communication and one for Suite to CLI communication.

Using enums is a deliberate design choice because of Rust's exhaustive pattern matching, this allows adding new types of messages to the common library easily because the CLI and Suite are required by the compiler to take them into account when performing a match statement.

The serialize and deserialize functions are also defined in the common crate which allows to easily change the format in which messages are sent. To avoid duplication these functions are defined on type aliases that are set using a compile time feature, allowing the outgoing and incoming messages to correspond to the correct enum depending on whether the library is included in the suite or CLI.

The message types that have to be supported by the CLI and suite are listed together with a short description in Table 4.2.

Message Type	Direction	Description
GetStatus	CLI → Suite	Request the current status of the suite
Done	CLI → Suite	Signal that the CLI is done sending requests and that the suite should respond with a status update
Suspend	CLI → Suite	Request the suite to permanently suspend execution
Benchmark	CLI → Suite	Contains benchmarking information, requesting the suite to perform the benchmark and reply with the result
Status	Suite → CLI	Contains status information of the suite
Error	Suite → CLI	Signals an error occured while handling a request, contains error information
BenchmarkResults	Suite → CLI	Response to a benchmark request, optionally contains the results of the benchmark if the suite was able to perform it

Table 4.2: List of message types defined in the common crate.

5 Implementation

5.1 Suite

5.1.1 Booting

The majority of the booting process is managed by riscv-rt crate. The crate depends on a linker script for it to correctly set up the memory layout on the target device, it clears the bss section and initializes the data section by copying the contents from the small data (sdata) section, which is only used to store the bytes that should be used to initialize the data section. This is required because sdata lies in potentially read-only memory regions, while the data section is required to be writable. When stripping the compiled elf to a binary file for loading onto the FPGA, it is important to specify that the sdata section should be included, otherwise the suite will fail to run.

Additionally, when running on the earl grey platform the standard riscv-rt crate does not work out of the box. The reason for this is the manifest that has to be prepended to the rest of the suite's code. This is done by altering the linker script used by riscv-rt to include a new section called manifest that lies before any other section. To populate the manifest section with the required information for the test rom an assembly file is globally included through the Rust code.

5.1.2 Runtime

The runtime of the suite is the first thing that is set up after riscv-rt delegates control to the suite. The runtime first configures the heap to allow for the allocation of dynamically sized memory, this is required for easier handling of control data like the messages sent and received over the communication module. To use a custom allocator the 'alloc' crate of Rust is used, usually part of Rust's standard library, this crate depends on the implementation of an allocation function and in turn provides a set of dynamically allocated data structures. The function, that the alloc crate depends on, receives a memory layout description as an argument and has to return a pointer to the space where data conforming to the provided layout can be stored.

There also exist crates that given a memory range, provide logic for allocation and freeing of memory using different memory data structures like the `buddy_system` [1], `linked_list` [21] or `simple_chunk` [36] allocators. For the runtime of the suite the allocator provided by the `linked_list` [21] crate was used. A problem was the crate's dependency on atomic operations for it to provide the full functionality required by the alloc crate.

This makes sense for multi core architectures because memory allocations have to be synchronized across harts in order to prevent undesired memory sharing between harts. But since the target platform for the suite does not have support for atomic instructions the included crate had to be adapted to work on systems without them. This does not produce any problems because the suite is running on a single core processor with interrupts disabled, thus it is impossible for the hart to unexpectedly change execution in the middle of an allocation or deallocation.

After the setup of the heap, the runtime initializes the platform's communication module, this is important because functionality like the custom panic handler or test results try to output information using the communication module. As a fail-safe, when panicking the custom handler also tries to initialize the communication module if it is uninitialized.

Lastly, the runtime initializes all other modules that are present on the system. For easier debugging and convenience the runtime also provides print and println macros that are adaptations of the ones present in Rust's standard library. Functionality like formatting is therefore fully supported.

5.1.3 Main Loop

The main loop of the suite reads a line using the communication module and deserializes it using the common crate. It then matches the message in order to decide how to proceed. Rust's requirement for exhaustive matches ensures that the suite is always up to date with all the valid messages that may be interchanged, otherwise the suite fails to build.

For a typical benchmark, a list of datasets is stored that can be used as input for the modules to benchmark them. As part of the message, one of the datasets gets selected and the benchmark is performed. The datasets include information for input, configuration and expected output. All outputs from modules have to be checked somehow to guarantee that no unwanted compiler optimization takes place, where possible the output is therefore checked for correctness using an assertion.

5.1.4 Testing

Testing functionality was made possible following [42]. It works by annotating functions that should be run as tests. The Rust feature `custom_test_frameworks` [38] then collects those functions and passes them to a testing function for execution. By implementing a custom trait for all function pointers it is possible to output information like the currently running test, before executing the function's body. Additionally using conditional compilation an alternative panic handler is used to signal the failure of a test.

While this suffices for the suite's testing needs, this approach has the drawback that the whole test run is aborted once the first test fails. Hence, the order in which the tests are run might be significant.

As an example: In an earlier version of the suite the test for seeded random number generation was run before the test for heap allocations. Because the wrong memory region was provided to the allocator, heap allocations failed and so did the test of the rng module because it depended on them. From the test output alone it looked like the rng module was causing the issues because the test run is aborted after its test failed.

This was easily fixed by reordering the test, in particular running the runtime tests before any others. The decision to keep aborting after the first failed test was made because unwinding and resetting the testing suite after a panic might result in an unstable state. As part of this thesis, it was therefore deemed easier to keep the ordering of these tests in mind instead of putting work into recovering from panics during testing.

Since not all platforms include all modules, it is also not possible to perform all tests on every platform. Instead of not including a test in the output which could lead to the impression that everything works, instead a macro was implemented to mark a test as skipped. This macro simply sets a global flag that is read after running the test and depending on whether it is set or not the test was either passed or skipped.

5.1.5 Opentitan Uart

The opentitan uart module serves both as a driver for the uart IP core of the earl grey platform and as an implementation of the communication module. The module upon creation has to be provided with a base address, a baud rate and the frequency of the peripheral clock (f_{pclk}). This information is required during initialization, where it is used to compute the value for the numerically controlled oscillator (NCO).

$$NCO = 2^{20} \cdot f_{baud} \div f_{pclk}$$

During initialization, the NCO is provided to the IP core through the control register (CTRL) together with the flags to enable receiving and sending over the uart. After resetting the core's fifo queues, it is ready for operation.

Sending is as simple as writing to the "write data" (WDATA) mmio register to append a byte to the sending queue, it is only necessary to first check that the queue is not full by reading the FIFO_STATUS register. If the queue is full the current implementation of the suite just spins until the uart has sent the next byte. Similarly for receiving data the number of elements in the receiving queue can be queried and if data is present it can be read from the "read data" (RDATA) mmio register.

5.1.6 Opentitan Hmac

The opentitan hmac module controls the hmac IP core of the earl grey platform. While the core supports computing both SHA2 hashes and MACs, only the hashing functionality is used by the suite. The hmac module has to be initialized before every hashing process to enable the cores SHA2 functionality and disable the MAC process. Before providing data to the core the hashing process is started by writing a trigger flag to the command register. Then as long as the 16 x 32bit queue is not full data can be provided by writing to the message register. By default, data provided to the message register is read in little-endian byte order, but this can be configured through a flag during the initialization of the hashing process. The status register can be read to get the number of elements currently in the queue as well as a flag that is only set if the queue is full, this allows for implementing the `input_ready` and `get_fifo_elements` functions. After writing the last block of data a trigger flag has to be written to the CMD register for the core to finish processing the input. The suite then polls the interrupt state register to check if the core has finished its computation. Once done the hash digest can be read from the 8 digest registers. Each individual digest register is in little-endian by default and can be configured to be big-endian during the initialization of the hashing process.

5.1.7 Opentitan Kmac

The opentitan kmac module controls the kmac IP core of the earl grey platform. The core supports multiple hashing modes as well as message authentication code generation but the implementation only makes use of the SHA3 hashing. During the initialization, the module implementation configures the core to only compute the hash before starting the core by writing to the command register. Like for the hmac module the status register can be read to get the number of elements currently in the queue as well as the queue full flag for implementing the `input_ready` and `get_fifo_elements` functions. Waiting for the completion of the hash computation works similar to the hmac module by first telling the core to process the input through the command register before polling the status register to check whether the core reached the SHA3_SQUEEZE state. After reading the digest from the corresponding register one last command has to be sent to the core to deinitialize it.

5.1.8 Opentitan Csrng

The opentitan csrng module interfaces with the csrng IP core to provide true and pseudo random number generation. During the runtime initialization, the csrng core is enabled for software control. The csrng IP core is controlled through a custom message protocol. A message is a 32bit value and sent to the core by writing to the COMMAND_REQUEST register, as long as the command ready flag is set. The core can either be initialized by a software seed or through an entropy source. The csrng module accommodates this by accepting an optional vector of seed material. If no seed material is passed during initialization the entropy source is used for seeding the core.

The initialization process when using the entropy source is as follows: First, the "uninstantiate" message is encoded in a header and then sent to the core to reset it to a valid state independent of it's previous state. Then the instantiate message is encoded together with the flags to seed using an entropy source.

When initializing with seed material the first "uninstantiate" message is the same. The following instantiate message has the software seeded flag set instead. Additionally, the length of the seed (in messages) is also included. After the initialization message, the specified number of up to 12 messages can be sent as seed material.

Once initialized the core accepts any number of generation requests. Such a request encodes the generation command and the number of 128bit blocks that should be generated. After sending the message the GENBITS_VLD register is polled until the core signals it finished generating, and then the blocks can be read from the 32bit GENBITS register. For every block, the register has to be read 4 times.

5.1.9 Opentitan AES

The opentitan aes module is responsible for interfacing with the aes IP core. It supports a number of different configurations with different supported modes and key lengths. The configurations are set during the initialization for a new encryption or decryption.

The initialization code serializes the mode (ECB, CBC, CFB, OFB, CTR), the key length (128, 192, 256) and the operation (encrypt, decrypt) into a 32bit value that is written to the shadowed control register. Because the control register is shadowed, which is a security measure against fault injection attacks by keeping multiple copies of the register contents [30], to change the content two write operations have to be performed for the register to fully update. The different arguments to the initialization function are provided as enums, some containing additional initialization vector information.

After writing the configuration, the key is written in two shares to the core using the KEY_SHARE0_0 - KEY_SHARE0_7 and KEY_SHARE1_0 - KEY_SHARE1_7 registers. The actual key is obtained by XORing the two shares, while this is required for the masked aes cores. Independent of the key size it is always necessary to write to all 8 32bit registers, if the actual key is smaller than 256bit the rest is ignored. Once the key is processed an initialization vector (IV) can be supplied if the encryption mode requires it, this is done by writing the 128bit IV block to the registers IV_0 - IV_3.

During manual operation of the aes module, it is then possible to write a single 128bit block of data to the cores DATA_IN_0 - DATA_IN_3 registers before triggering the en- or decryption by writing a start flag to the TRIGGER register. The status register can then be polled for output valid flag. Once the flag is set the DATA_OUT_0 - DATA_OUT_3 registers can be read for the result.

The automatic mode instead immediately starts computation once a block is received on the DATA_IN registers, this results in overall better performance but less granularity when trying to benchmark the time the aes module takes to en-/decrypt one block. The automatic mode also allows for treating the aes input as a queue with depth 2: Once the first block is written to the core the suite can await for the input registers to become ready again, which will happen before the output registers contain the valid output of the 1st block. Once ready, the 2nd block can be written to the registers before reading the output of the 1st block.

5.2 CLI Tool

After starting, the CLI Tool parses the arguments and for each of the passed benchmarking files starts a connection over TTY and performs the described benchmark, writing the result back to a file.

To establish a connection to the suite the CLI tool uses the serialport [34] crate communicating over the TTY to send a status request, once the suite responds the actual communication starts. The CLI reads the benchmark file line by line and tries to parse each line into a message. Empty Lines or lines starting with a # are regarded as comments and copied without changing to the output file. Every valid line containing a message gets instead replaced by the reply sent by the suite. When reaching the end of the file the CLI tool sends a message to signal it is done and reads lines for as long as the suite needs to reply with a status message signaling its completion.

BenchmarkInfo	Possible results	Additional information provided
AESDataSet	AESPerBlock or AESTotal	Specifies a dataset and a combination of en/decryption and either total or per-block benchmarking
RNGDataSet	RNGTotalSeeded	Specifies a dataset
RNGTrueRandom	RNGTotalTrueRandom	Specifies number of blocks to generate after seeding by entropy source
HashDataSet	SHA2Total or SHA3Total	Specifies a dataset and if sha2 or sha3 should be used
MicroBenchmarks	MicroBenchmarks	-

Table 5.1: The different types of benchmarks that can be requested by the CLI

5.3 Common Library

The common library defines the list of messages that can be sent between the CLI tool and the suite, all defined message types can be found in Table 4.2. These messages are defined as enums with optional tuples of values to store additional information. Table 5.1 lists all types of benchmarks that are currently implemented in the common library.

Because the library is also part of the suite it has to work in no-std environments with nothing but the alloc crate as a dependency, reducing the number of libraries that can be used for serialization and deserialization.

Serialization and deserialization is implemented using `serde` [32] and `serde_json` [33] library both of which support the no-std environment and were chosen because of ease of use. The `serde_json` library is comparatively large in size and not the most efficient way to serialize the small set of messages supported. Yet it allowed for fast development and was part of a noncritical part in terms of performance. If the resulting binary would grow too large or communication would become too slow this is one part that should be easy to optimize by providing an alternative implementation for the serialization and deserialization functions.

6 Evaluation

Next, we describe the manner in which all modules of the benchmarking tool are evaluated. For each module the following questions were analyzed:

1. What is the number of cycles a module takes for a specific configuration?
2. What is the contribution of different operations to the total cycle count?
3. What is the bottleneck for each module in terms of performance?

6.1 Results

The following benchmarks were all performed on the NexysVideo FPGA [26] with an opentitan bitstream generated from the opentitan_earlgrey_v5 version of the project. The results provide an overview of the cycles a higher-level application would spend on the processes.

As already explained in the RISC-Vbackground the method used to measure cycles introduces an overhead of at least five cycles this has a lot of impact on the tighter measurements, especially for the aes benchmarks. The numbers presented here are the raw numbers i.e. the direct difference in the cycle measurements without any adjustments made for the overheads. This decision was made due to the inconsistent overhead seen in the microbenchmarks, where the cycles between two immediate readouts were between 4 and 6.

It is also worth noting that as part of the benchmarks it is often necessary to wait for a flag to be set. This can either be done by having the core cause an interrupt or by busy waiting. These benchmarks are all performed by busy waiting and thus continuously reading, comparing and jumping if the flag is not set. This decision was made because interrupts have generally been disabled for the whole suite and to save cycles for handling interrupts.

A typical busy wait loop usually consists of the following steps: 1 load operation, 1 integer operation 1 conditional jump taken. When using Table 3.1 this results in a loop cycle taking at least 6 cycles. This considerably impacts the granularity at which a flag can be detected to be set. As an example suppose the flag is set the cycle after the load operation finished it will take at least 8 more cycles (2x integer operations, 1x jump taken, 1x load operation, 1x jump not taken) until the busy loop successfully exits.

6.1.1 Hashing

The first benchmark for the hashing modules was done using a dataset containing 2048bits or 256bytes of data in total. This data was then hashed using both the hmac and the kmac module. Both implementations internally make use of a message queue resulting in the computation cycles being the sum of the cycles spent inserting all the input data and the cycles spent waiting for the final data block to be processed.

After running the first benchmark, the results of which are in Table 6.1, it was first noted that the results do not seem all that different. The initialization time is the same which is expected since both implementations write to two control registers to start the hashing process. The insertion of the input data into the respective queues differs by 60 cycles between the hmac and the kmac implementation, which would correspond to less than 1 cycle per block assuming they are equally distributed. A bigger difference is present in the final step of computation, waiting for the digest to be computed, the kmac implementation is 55% faster taking 87 cycles less than the hmac one. Considering the whole computation time hmac took 1848 cycles while kmac took 1701 cycles and thus being 8% faster.

To better understand the result, the implementation was first modified to count the number of times it had to wait because the queue was full. It turned out that neither SHA2 nor SHA3 had to wait at any time because the queue was full. This could be due to the queue being emptied faster than new input can be provided.

To test this hypothesis it was necessary to provide input as fast as possible while measuring the elements in the queue. For this, the busy wait loop was removed and replaced by a single readout of the queues element number.

This effectively removes the check of whether the module is ready for new input in order to provide input as fast as possible. To determine if the input was ready retroactively this thesis looks at the number of elements in the queue. For both SHA2 and SHA3 the queue is never filled with more than 1 element during this process. Considering that the total queue size of hmac and kmac are 16 and 18 (9 x 64bit) respectively, the queues are consistently underfull.

Table 6.2 shows the results of hashing 2048bits of data without any logic to assert that the module is ready for more input. The resulting digest is correct. Given the previous information about the underfull queues for both hmac and kmac, this thesis argues that removing these checks is safe to do for this specific benchmark. As can be seen in Table 6.2 the cycles spent inserting the data into the queues were significantly

Step	Cycles (H)	Cycles (K)
Initialization	21	20
Input Data	1694	1634
Await Digest	154	67
Read Digest	74	81

Table 6.1: Benchmark results of hashing modules
H - Using sha2 and hmac, K - Using sha3 and kmac

Step	Cycles (H)	Cycles (K)
Initialization	27	27
Input Data	911	913
Await Digest	159	70
Read Digest	77	81

Table 6.2: Tighter Benchmark results of hashing modules by removing the checks of whether the queue is already full
H - Using sha2 and hmac, K - Using sha3 and kmac

reduced by more than 40% for both hmac and kmac. Since for both implementations, the input data section only contains memory transfer operations they are expected to be equal.

The benchmarks suggest that waiting for the control flag is a considerable bottleneck, making up 40% of the insertion time. Secondly because during the benchmarks the internal queues of hmac and kmac are never full, the CPUs store operations are not fast enough to fill the queue. Therefore it can be concluded that improvements to the CPU's store instruction as well as the instructions used during the busy wait for the flag can significantly improve the overall performance.

6.1.2 Random number generation

The benchmark was performed for two configurations of the csrng module. The first one uses a software provided seed of maximum supported length, which is 384bits and the second one uses a seed provided entirely by the entropy source. Both configurations then generated 3 x 128bit of random data.

The result of the benchmark can be seen in Figure 6.2. As expected there is a big difference in the initialization step of the random number generation between the two configurations. This is the result of the 12 * 32bit values having to be sent to the IP core.

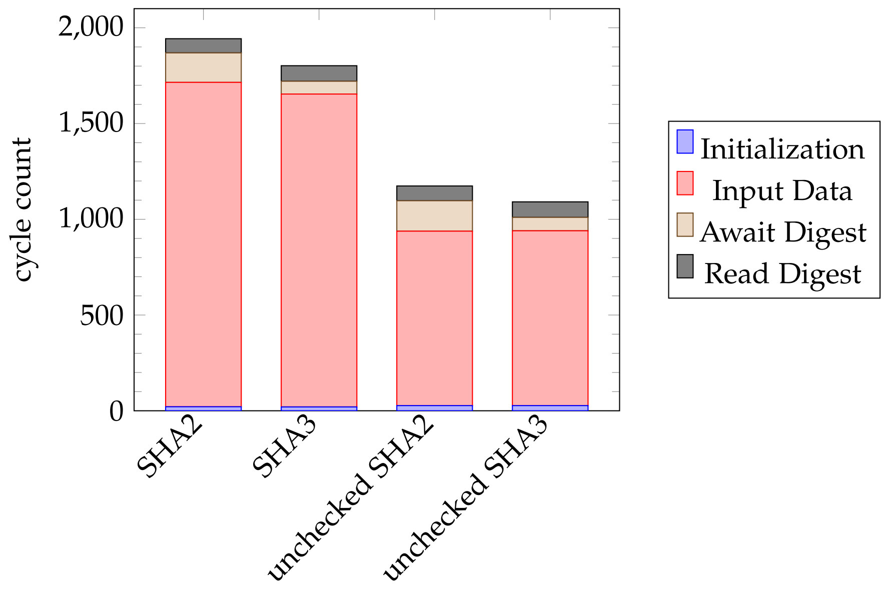


Figure 6.1: Cycles measured for 2048bits of input data for both SHA2 and SHA3 hashing implementations. Unchecked refers to implementations that do not check whether the internal queue is full.

An interesting point is that the generation of the first block takes considerably longer for the entropy seeded implementation. This is likely the case because the initialization is not finished after the 55 cycles. Because of how the random number module is operated through messages the initialization step returns once the last message is sent. Because before sending any message, it is required to wait for a flag to be set. This flag is likely not set by the time the first generation request is made, resulting in a longer wait.

To check this theory a second benchmark was performed that inserts an artificial wait time between the initialization and first generation step which then had the same cycle count as the other ones. This means that it takes only 55 cycles to perform the initialization steps of entropy seeded rng but the core takes around 50 additional cycles to fully initialize.

The only check for the validity of the generated data is that it is consistent for seeded generation. No other criteria were used to determine if the data read suffices as cryptographically secure random numbers.

The random number generation logic of the IP cores itself is a potential bottleneck when performing a lot of random number generations. In special cases, if there

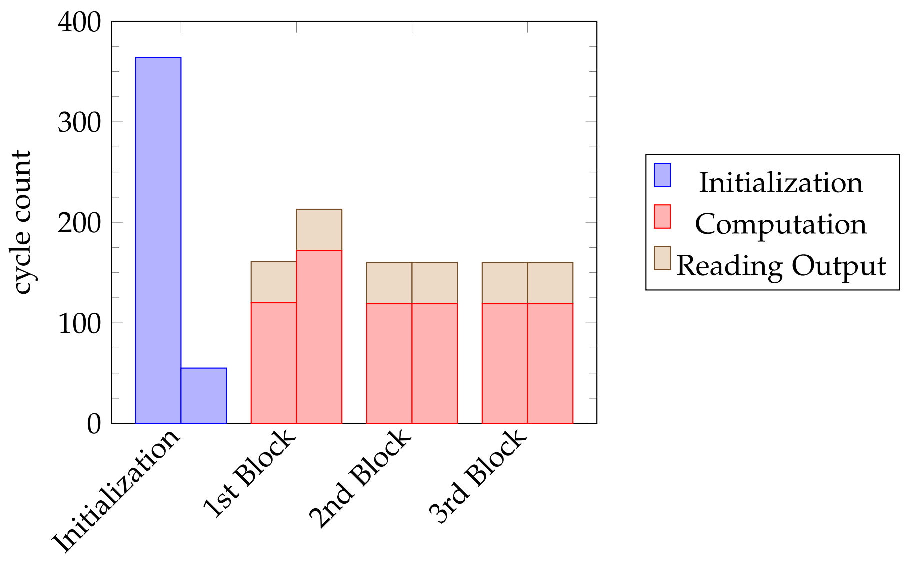


Figure 6.2: Cycles measured for generating 3x 128bit blocks.
With 384bit seed left and entropy seeded right.

are many reseeding operations, for example when using the csrng module as a key derivation function, the transmission of the seed material to the csrng module could also become a bottleneck.

6.1.3 AES

The initial implementation of the benchmarking procedure for the AES core only included the automatic mode, making use of the cores 2 stage pipeline to write the next input block before the prior block finished computation. As a dataset, a message consisting of 5 x 128bit blocks was used, with the benchmark results displayed in Figure 6.3.

The fact that the initialization times match up is expected because regardless of actual key size, all 8 key registers have to be written to. The computation results were unexpected at first because according to the opentitan documentation [28] the different key lengths should have an effect on the encryption time, yet the cycle count perfectly matches up for the unmasked implementation.

To get a more detailed result on the actual computation time a benchmark in manual operation mode was added, while having a larger expected total cycle count this allows for measuring more accurately which step takes up what amount of cycles. The cycles

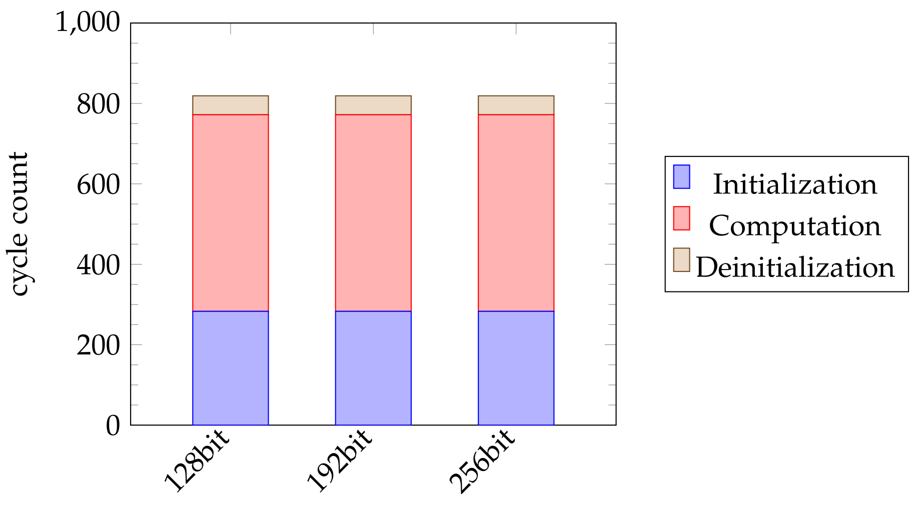


Figure 6.3: Cycles measured for encrypting 5 x 128bit blocks using unmasked AES.

measured for encrypting a single 128bit block can be seen in Figure 6.4. The read and write operations of the different key sizes again match up since all operate on 128bit blocks, what can be seen now are the differences in the computation time.

To understand why a difference in the cycle count can be seen in manual mode but not in the total cycle count of the automatic count Figure 6.5 and Figure 6.6 can be considered. The diagrams depict the execution without pipelining on the top row and the two-stage pipelined execution on the bottom row for two different potential realities. The orange and blue blocks correspond to cycles spent by the CPU writing and reading blocks to the accelerator respectively. While the grey blocks correspond to the cycles the accelerator spends encrypting/decrypting a given block. The numbers signal what block is processed.

Figure 6.5 shows a reality where the accelerator's computation time is greater than the sum of the read and write operations. If this is the case then the accelerator is the bottleneck when using a two-stage pipeline and automatic execution. This leads to the total cycle count for the pipelined result changing with small deviations in the accelerator's computation time. Since this does not hold for the benchmarks of the opentitan aes core this is likely not the case for this core.

Figure 6.6 shows the case where the accelerators computation time is dominated by the sum of the read and write operations. This leads to the CPU being the bottleneck when pipelining, in particular, changes in the computation time of the accelerator are only noticed once the computation cycles get large enough to dominate the read and write operations. This is likely the case for the opentitan aes core since the observations

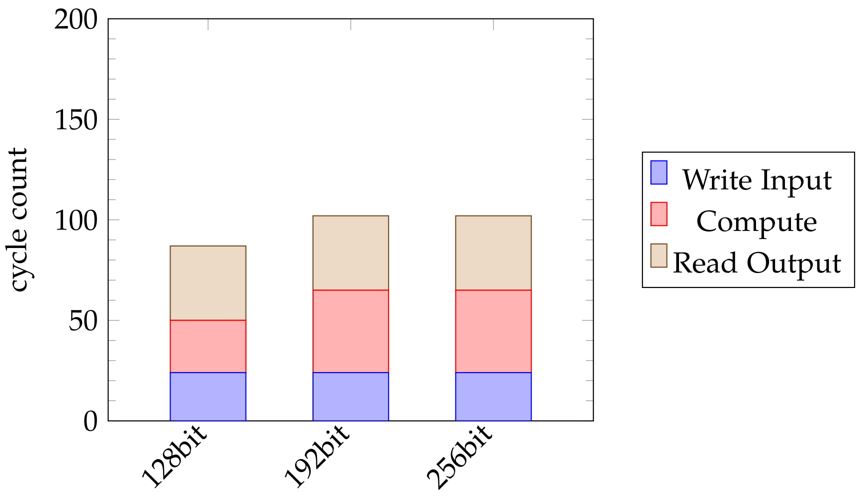


Figure 6.4: Cycles measured for a single 128bit block unmasked AES encryption.

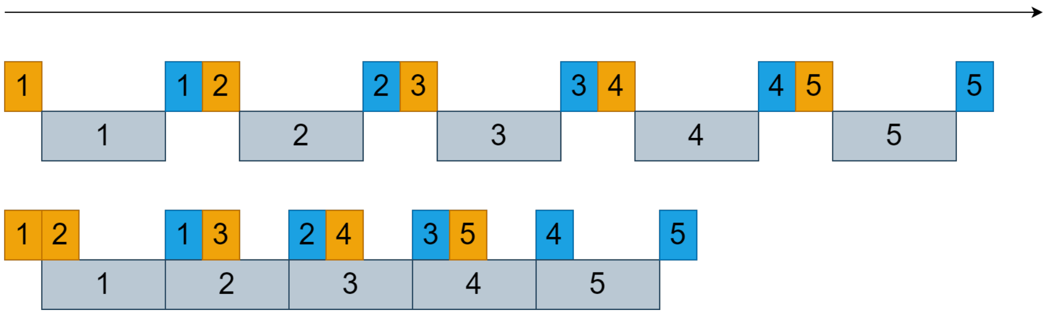


Figure 6.5: Usage of the aes core with dominant computation
Top: no pipelining, Bottom: 2 stage pipelining

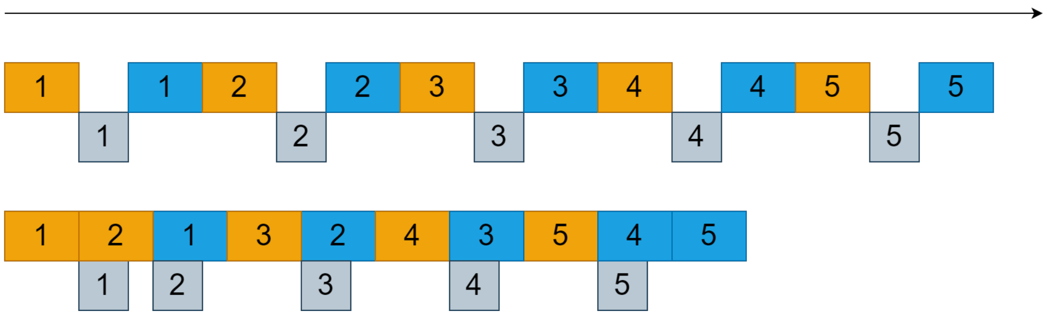


Figure 6.6: Usage of the aes core with dominant read-writes
Top: no pipelining, Bottom: 2 stage pipelining

Step	Cycles	Description
Initialization	306	Initializes by setting mode, key length, key, iv
Write Block	24	writes a single 128bit block in the aes cores input register
Compute Block	41	triggers the start of the computation and waits until completion
Read Block	37	reads a single 128bit block from the aes cores output register
Deinitalization	44	Clears any sensitive information from the aes core

Table 6.3: Benchmark results of the unmasked AES core with 256bit key length

match up.

It appears that the aes core is only taking a few cycles for the encryption and decryption of a single block especially in the unmasked case, this allows evaluation of the impact of busy wait loops as described earlier. During the benchmarking process, it was observed that inserting `nop` instructions before entering the busy wait loop potentially decreased the overall waiting time, this is because of a better timed flag readout.

In an attempt to minimize the overhead of the flag check a benchmark was created that uses `nop` instructions to precisely time the readout of the status register containing the desired flag. This was done by decreasing the number of instructions inserted until the readout had the flag not set. This had to be done for every configuration of the AES module that could have an impact on the computation time.

The computation time of a single block using this measurement method can be seen in Figure 6.7 for an unmasked aes implementation. The figure shows that this timed readout can have a significant impact, reducing the cycles for the 192bit key length by more than 40%.

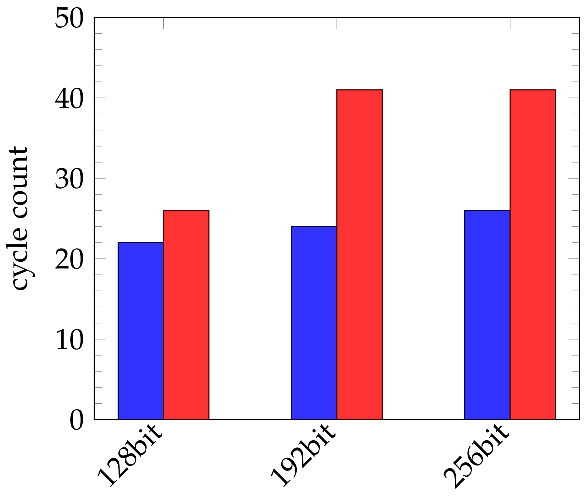


Figure 6.7: Cycles measured for a single unmasked 128bit block AES encryption. With custom waiting left and ordinary waiting right.

7 Summary and Conclusion

The effort to reduce the number of components of a system, that require trust, gives way to a handful of small critical components that are easier to verify than a whole system. Using a project like opentitan to realize these components in the form of open hardware allows to reinforce this trust by being able to inspect and audit, not only the software running on it but also the hardware underneath.

Open hardware in this form, with its fundamentally transparent design, is not enough to make it successful. In addition, any such projects have to be competitive with the current industry standards and one area allowing for comparisons is performance. It is in the interest of the open hardware projects to provide the best performance possible for all its modules to be competitive.

To further the performance improvements of the opentitan project this thesis analyzed a subset of its security functionality. By measuring their performance and breaking it down into different operations, this thesis identified the critical operations that act as a bottleneck to the overall performance. By doing so this thesis gives insight into what areas of the project should receive more attention to increase the performance and thus the overall competitiveness of the opentitan project.

In order to perform this analysis, this thesis introduced the design of a benchmarking tool running directly on the hardware and performing cycle accurate benchmarks. By doing so the opentitan project was tested under real-world conditions with all components that potentially affect the performance taken into account. This thesis explained how such a tool can be designed so that it can easily be adapted to new targets by splitting it into two mostly independent components. It showed how a minimal communication protocol can be defined and enforced by linking those two components through a single common library.

The introduced design was then realized through the implementation of the benchmarking tool that is published on GitHub under the following address:
`https://github.com/TUM-DSE/cycle-accurate-hardware-benchmark-tool`
The implementation is able to benchmark hashing, random number generation, and aes encryption and decryption on the opentitan platform and can easily be extended to support other platforms.

This tool was then used to perform and evaluate a number of different benchmarks on the opentitan platform. The results, that are listed in chapter 6, show that the CPU is a considerable bottleneck for certain opentitan accelerators. This is especially the case for accelerators that require a lot of interaction to manage, including a lot of input or output data. Further, it was shown that busy wait loops for flags can produce a significant overhead if the flag is read at an inconvenient time.

These findings can be supported by recent development efforts in the opentitan project. The current version uses a different configuration of the Ibex core that implements three-stage pipelining to reduce the cycles spent on instructions that await a response. By introducing this additional writeback stage it is possible to reduce the cycles required for a load operation by one. Additionally, the new Ibex core allows for the addition of a specialized branch target arithmetic logical unit (ALU) to compute the branch condition and the jump destination in one cycle reducing the number of cycles spent for branches taken, by one cycle. Both of these changes directly reduce the cycles of a busy wait loop allowing for more accurate waiting.

8 Future Work

The scope of this thesis was initially bigger, including both an in-depth benchmarking of the OTBN and a comparison between the masked and unmasked implementations of the kmac as well as the aes cores. Unfortunately, issues during the development process made these goals unachievable under the conditions and the time frame of this thesis.

Initial development was performed with verilator running the latest opentitan version as the target under the assumption that the tool should be able to run on the actual FPGA with only a small number of changes. Unfortunately, the only available FPGA was the, by opentitan no longer supported, NexysVideo board. The opentitan version that was so far used for development was not able to run on this board. After a failed attempt to backport the new opentitan version to the NexysVideo board the decision was made to switch to an older version. This thesis was then set to work with the last tagged release on GitHub that still supports the NexysVideo board. This is the opentitan_earlgrey_v5 version. In hindsight, it would have potentially been better to use a more recent version, even if it is not tagged as a release, since there was substantial development since the opentitan_earlgrey_v5 version.

Using the opentitan_earlgrey_v5 version allowed for opentitan to run on the NexysVideo board but even after porting the code introduced a number of problems. The OTBN libraries (ECDSA & RSA) were missing with existing ones being incompatible due to changes to the hardware IP. An attempt to write a custom module for the OTBN core, to manually control it and backport the existing ECDSA and RSA libraries failed because any access to the core, even its control and status registers resulted in a bus crash.

This gives potential future work in the form of including the OTBN module in the benchmarking tool to perform measurements for ECDSA and RSA operations. In particular, given the results of this thesis, it is worth comparing the OTBN accelerated crypto libraries to ones running solely on the CPU. The potential impact of the read/write operations required to program the OTBN and operate it may add a significant overhead. Taking into account that the OTBN may has to be reprogrammed multiple times to perform different operations this could lead to an overhead of multiple hundred cycles.

In addition, it is also worth comparing the masked and unmasked implementation of the kmac and aes cores, in particular, to see whether the overhead introduced by the CPU operations results in a nearly identical performance in practice even though the underlying IP cores have vastly different cycle counts. Unfortunately, the generation of opentitan_earlgrey_v5 bitstreams that implement these masked IP cores failed for this thesis but should be possible and achievable as future work.

The fact that this thesis used an older project version to accommodate for the FPGA restriction also provides a basis for future improvements by porting the benchmarking tool to the most up to date version, which at the time of writing only supports the ChipWhisperer CW310 board. This allows taking into account the changes to the Ibex core with its new three-stage pipeline and the branch target ALU.

Porting the tool should at least require updating the memory layout files and adapting the manifest to the new specification. Additional work will also be required in updating the existing opentitan modules since the register layout of certain cores has changed. In that regard, it would be a worthwhile future addition to integrate opentitans auto-generated c header files that serve as a mmio address specifications. That way the benchmarking tool would always use the addresses that are defined in the auto-generated file.

List of Figures

List of Tables

Bibliography

[1] *buddy_system_allocator*. https://github.com/rcore-os/buddy_system_allocator. Accessed: 2022-07-24.

[2] D. Canright. "A very compact rijndael S-box." In: (2005). DOI: 10.21236/ada434781.

[3] J.-S. Coron, Y. Dodis, C. Malinaud, and P. Puniya. "Merkle-Damgård Revisited: How to Construct a Hash Function." In: *Advances in Cryptology – CRYPTO 2005*. Ed. by V. Shoup. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 430–448.

[4] *cortex-m*. https://github.com/rust-embedded/cortex-m. Accessed: 2022-07-24.

[5] R. K. Dahal, J. Bhatta, and T. N. Dhamala. "Performance analysis of SHA-2 and SHA-3 finalists." In: *International Journal on Cryptography and Information Security (IJCIS)* 3.3 (2013), pp. 720–730.

[6] *Design Verification Methodology within OpenTitan*. https://docs.opentitan.org/doc/ug/dv_methodology/. Accessed: 2022-07-28.

[7] C. Domas. "Breaking the x86 ISA." In: *Black Hat* (2017).

[8] C. Domas. "Hardware backdoors in x86 CPUs." In: *Black Hat* (2018), pp. 1–14.

[9] M. J. Dworkin. "SHA-3 standard: Permutation-based hash and extendable-output functions." In: (2015). DOI: 10.6028/nist.fips.202.

[10] *ECDSA with NIST P-256*. https://github.com/lowRISC/opentitan/tree/master/sw/device/lib/crypto/ecdsa_p256. Accessed: 2022-07-28.

[11] R. N. Fjeldsø and S. S. Nielsen. "Detecting Possible Timing Attack Vulnerabilities in OpenTitan Big Number Accelerator Programs." In: ().

[12] A. Fogh and D. Gruss. "Using Undocumented CPU Behavior to See Into Kernel Mode and Break KASLR in the Process." In: *Blackhat USA* (2016).

[13] H. Gross, S. Mangard, and T. Korak. "Domain-oriented masking." In: *Proceedings of the 2016 ACM Workshop on Theory of Implementation Security* (2016). DOI: 10.1145/2996366.2996426.

[14] H. Gross, S. Mangard, and T. Korak. *Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order*. Cryptology ePrint Archive, Paper 2016/486. https://eprint.iacr.org/2016/486. 2016.

[15] T. C. Group. *TPM 2.0 Library*. https://trustedcomputinggroup.org/resource/tpm-library-specification/. Accessed: 2022-08-08.

[16] *Ibex Reference Guide: Pipeline Details*. https://ibex-core.readthedocs.io/en/latest/03_reference/pipeline_details.html. Accessed: 2022-07-23.

[17] *Ibex RISC-V Core*. https://github.com/lowRISC/ibex. Accessed: 2022-07-23.

[18] *Introduction to Differential Power Analysis and Related Attacks*. https://www.rambus.com/wp-content/uploads/2015/08/DPATechInfo.pdf. Accessed: 2022-07-28.

[19] Y. Jararweh, H. Tawalbeh, A. Moh'd, et al. "Hardware performance evaluation of SHA-3 candidate algorithms." In: (2012).

[20] X. Li, Z. Wu, Q. Wei, and H. Wu. "UISFuzz: An Efficient Fuzzing Method for CPU Undocumented Instruction Searching." In: *IEEE Access* 7 (2019), pp. 149224–149236. DOI: 10.1109/ACCESS.2019.2946444.

[21] *linked_list_allocator*. https://github.com/rust-osdev/linked-list-allocator. Accessed: 2022-07-24.

[22] M. G. Mark Ermolov Dmitry Sklyarov. *UNDOCUMENTED X86 INSTRUCTIONS TO CONTROL THE CPU AT THE MICROARCHITECTURE LEVEL IN MODERN INTEL PROCESSORS*. https://github.com/chip-red-pill/udbgInstr/raw/main/paper/undocumented_x86_insts_for_uarch_control.pdf. Accessed: 2022-08-08.

[23] B. Møller, M. Pedersen, and T. Bøgedal. "Formally Verifying Security Properties for OpenTitan Boot Code with Uppaal." In: *To Appear. MA thesis. AAU* (2021).

[24] B. H. Møller, J. G. Søndergaard, K. S. Jensen, M. W. Pedersen, T. W. Bøgedal, A. Christensen, D. B. Poulsen, K. G. Larsen, R. R. Hansen, T. R. Jensen, et al. "Preliminary Security Analysis, Formalisation, and Verification of OpenTitan Secure Boot Code." In: *Nordic Conference on Secure IT Systems*. Springer. 2021, pp. 192–211.

[25] *msp430-rt*. https://github.com/rust-embedded/msp430-rt. Accessed: 2022-07-24.

[26] *Nexys Video Reference Manual*. https://digilent.com/reference/programmable-logic/nexys-video/reference-manual. Accessed: 2022-08-04.

[27] *OpenTitan*. https://github.com/lowRISC/opentitan. Accessed: 2022-07-25.

[28] *OpenTitan Documentation*. https://docs.opentitan.org/. Accessed: 2022-07-24.

[29] *OpenTitan Hardware Development Stages*. https://docs.opentitan.org/doc/project/development_stages/. Accessed: 2022-07-28.

[30] *OpenTitan: Register Tool*. https://docs.opentitan.org/doc/rm/register_tool/. Accessed: 2022-07-28.

[31] *riscv-rt*. https://github.com/rust-embedded/riscv-rt. Accessed: 2022-07-24.

[32] *Serde*. https://github.com/serde-rs/serde. Accessed: 2022-07-24.

[33] *Serde JSON*. https://github.com/serde-rs/json. Accessed: 2022-07-24.

[34] *serialport-rs*. https://github.com/serialport/serialport-rs. Accessed: 2022-07-25.

[35] *SiFive TileLink Specification*. https://static.dev.sifive.com/docs/tilelink/tilelink-spec-1.7-draft.pdf. Accessed: 2022-07-25.

[36] *Simple Chunk Allocator*. https://github.com/phip1611/simple-chunk-allocator. Accessed: 2022-07-24.

[37] J. Søndergaard and K. Jensen. "Formally Verifying the Correctness and Safety of OpenTitan Boot Code using CBMC." In: *To Appear. MA thesis. AAU* (2021).

[38] *The Rust Unstable Book: custom_test_frameworks*. https://doc.rust-lang.org/beta/unstable-book/language-features/custom-test-frameworks.html. Accessed: 2022-07-24.

[39] *The rustc book: Platform Support*. https://doc.rust-lang.org/nightly/rustc/platform-support.html. Accessed: 2022-07-24.

[40] *Universal Verification Methodology (UVM) 1.2 Class Reference*. https://www.accellera.org/images/downloads/standards/uvm/UVM_Class_Reference_Manual_1.2.pdf. Accessed: 2022-07-28.

[41] A. Waterman and K. Asanovi. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*. RISC-V Foundation, Dec. 2019.

[42] *Writing an OS in Rust | Testing*. https://os.phil-opp.com/testing/. Accessed: 2022-07-24.