Processor Design Project
Intermediate Report

Project Title: CESE4040 – Processor Design Project
Group Members: Zubair Al-Zubi, Hashim Karim, Amin Aynan, Oussama Seddouki
Student IDs: 5342007, 4811518, 5344948, 46
Course Name: Processor Design Project
Instructors: Motta Taouil, Georgi Gaydadjiev, Carlo Galuzzi, Stephan Wong, Mahmood Naderan-Tahan
Submission Date: May 2, 2025

Contents

1 Introduction **2**

2 Background **3**
2.1 RISC-V and Extensibility 3
2.2 RISCY Core 3
2.3 AES and Its Hardware Implementations 3
2.4 Cryptography Extensions in Existing ISAs 4
2.5 Compiler Optimizations for Cryptographic Workloads 4

3 Hypothesis **5**

4 Plan **7**

5 Individual Contribution **9**

6 Conclusion **10**

Appendices **12**

1. Introduction

The aim of the Processor Design Project is to address a common problem in modern computing: how to significantly increase the performance of AES (Advanced Encryption Standard) encryption such that it does not overly tax an embedded system's resources. AES is everywhere, from everyday smartphone apps to large-scale data centers and yet it can still take a considerable amount of time and energy when handled purely by software. This project aims to bridge this gap by customizing the RISCY core, which is based on the RISC-V ISA, with specialized AES instructions. Subsequently, the project will also extend the LLVM compiler to leverage these new instructions, ultimately delivering a faster and more efficient way to run AES workloads.

During the initial phase (Weeks 1 and 2), our focus is on setting up and verifying the baseline system. This involves taking the unmodified RISCY core, synthesizing it onto an FPGA board using Vivado, and running a straightforward AES software implementation to confirm that everything works as expected. Establishing this baseline is crucial: if the baseline system is not reliable, any enhancements we add later such as custom AES instructions or compiler optimization would be harder to verify and measure accurately.

Beyond just ensuring that the system works, Phase 1 also includes a short literature review to see how AES has been accelerated before, either through hardware or compiler techniques. This helps us to pinpoint proven strategies, as well as areas that could still benefit from fresh ideas. Once we understand what is already out there, we can form well reasoned hypotheses about how new AES instructions in hardware and loop-unrolling in the compiler might reduce cycle counts, improve throughput, or lower overall resource usage.

By the end of the first phase, we will have a clear picture of our baseline performance and a plan for the next steps. The intermediate report will summarize our initial setup, discuss what we learned from the literature, and map out how we intend to implement and evaluate our hardware instructions and compiler tweaks. From there, we will be ready to jump into Phase 2 (Weeks 3 to 8), where we'll put our plan into action adding specialized AES instructions to the RISCY core, adjusting the LLVM toolchain to make the most of them, and carefully measuring the gains.

2. Background

This section provides the necessary technical background on RISC-V, the RISCY core, AES encryption, and relevant compiler techniques to help understand the design choices made in this project.

2.1 RISC-V and Extensibility

RISC-V is an open-source, modular instruction set architecture (ISA) that has gained considerable traction in both academia and industry. Its defining feature is its extensibility: developers are free to define custom instruction set extensions tailored to specific applications, without being restricted by proprietary licenses [1]. This makes RISC-V especially attractive for domain-specific acceleration, such as cryptographic processing. Unlike conventional ISAs such as x86 or ARM, RISC-V allows implementers to strike a customized balance between performance, area, and energy efficiency. In particular, custom extensions for cryptography can deliver substantial speedups while maintaining a relatively small hardware footprint.

2.2 RISCY Core

In this project, we use the RISCY processor, a lightweight, open-source four-stage RISC-V processor specifically optimized for energy-efficient computing and digital signal processing (DSP) applications. Developed as a part of the Parallel Ultra-Low-Power (PULP) platform initiative, RISCY fully implements the base 32-bit integer instruction set (RV32I), alongside the multiplication instructions from the RV32M extension. It also integrates numerous custom instruction set extensions, collectively referred to as RV32IMFC, which significantly enhance its DSP capabilities. RISCY provides a straightforward pipeline structure that is particularly suitable for experimenting with custom ISA extensions. Its modular and accessible architecture simplifies modifications at the RTL level, enabling researchers and engineers to quickly prototype and evaluate new instructions. RISCY's simplicity and compatibility with standard FPGA toolchains make it an excellent candidate for integrating domain-specific accelerators, such as the AES extensions explored in this work.

2.3 AES and Its Hardware Implementations

The Advanced Encryption Standard (AES), introduced by NIST in 2001, is the most widely adopted symmetric key encryption algorithm today [2]. It operates on 128 bit data blocks and performs a sequence of well-defined transformations in multiple rounds namely SubBytes, ShiftRows, MixColumns, and AddRoundKey. When implemented in software, each of these transformations requires numerous instructions and intermediate memory accesses. As a result, software only AES can become a performance bottleneck on embedded processors.

To address these challenges, hardware-accelerated implementations of AES have become common. These include designs that embed optimized S-boxes in lookup tables, apply

pipelining across rounds, or unroll part of the round logic to reduce the number of cycles per block. Fully pipelined architectures can achieve high throughput but often require significant FPGA or ASIC resources. In contrast, iterative or semi-unrolled architectures offer a practical compromise between area efficiency and performance, making them suitable for mid-range embedded platforms such as the one used in this project [3].

2.4 Cryptography Extensions in Existing ISAs

To improve the performance of cryptographic workloads, commercial ISAs such as Intel's x86 and ARMv8 have introduced dedicated instruction set extensions like AES-NI and the ARMv8 Cryptography Extensions [4, 5]. These allow AES round functions to be executed as single instructions, significantly reducing the instruction count per block and improving both speed and energy efficiency.

However, such instructions are not always fully utilized by compilers. Without explicit support in the compiler backend, the instructions may remain inaccessible or require manual use through intrinsics or inline assembly. This lack of integration limits their potential and highlights the need for tight coupling between hardware design and compiler infrastructure. In contrast, RISC-V's openness and support for custom instruction integration make it well-suited for truly compiler-aware cryptographic acceleration.

2.5 Compiler Optimizations for Cryptographic Workloads

Compiler techniques are critical in realizing the full potential of hardware acceleration, especially for compute-intensive tasks like AES encryption. One of the most effective techniques in this context is loop unrolling. By statically replicating loop bodies, loop unrolling reduces the number of branches and exposes greater instruction-level parallelism. This enables better pipelining and scheduling in both hardware and compiler backends.

For AES, which involves ten nearly identical rounds of operations, full loop unrolling simplifies the control flow and allows the compiler to insert optimized instructions without the overhead of branching or loop counters. LLVM in particular supports extensive backend customization and transformation passes, making it a strong candidate for introducing target-specific AES instructions in a fully integrated way [6].

Other techniques such as bit slicing, aggressive function inlining, and prefetching have also been studied, though their benefits are often architecture-dependent. In our case, we focus on combining loop unrolling with custom instruction emission to achieve a high-performance, low-overhead AES implementation that is tightly coupled with the capabilities of the RISCY core.

3. Hypothesis

Modern embedded processors spend far more time and energy on moving data than on the arithmetic itself. For AES encryption on the five-stage RISCY core, the dominant cost arises when the round state and keys are repeatedly spilled to, and refetched from, the L1 data-cache. Our central hypothesis is therefore: **If the entire AES working set can be retained in a handful of dedicated on-chip registers, execution latency and dynamic energy will fall dramatically—without pushing the core outside its strict area and timing budgets.** The intuition is straightforward. A single 32-bit register read completes in one cycle and dissipates roughly a picojoule, whereas an SRAM hit stretches over several cycles and consumes an order of magnitude more energy. Each extra register that prevents a load or store thus removes a pipeline bubble and avoids a costly memory toggle. Because AES performs ten almost identical rounds, even modest savings per round compound into double-digit gains at the block level.

To act on that insight we propose a minimal-footprint design:

- Four 32-bit "crypto-state" registers hold the 128-bit data block,
- Four further registers cache the current round key
- Lifetimes are overlapped so that temporaries are overwritten as soon as they become dead.

By keeping the live set below eight physical registers we avoid enlarging the global register file, limit additional decode wiring, and retain the RISCY core's 100 MHz timing closure on the PYNQ-Z2 FPGA.

These architectural choices lead to three testable sub-hypotheses:

Hypothesis 1 (Latency): Collapsing each round into a single macro-operation and eliminating memory traffic will cut the cycles-per-block by at least an order of magnitude relative to the scalar baseline.

Hypothesis 2 (Energy): Replacing 400-plus cache accesses with register transfers will lower dynamic energy per encrypted block by $\geq 40\%$.

Hypothesis 3 (Register Pressure and Memory Footprint): We hypothesise that the complete AES encryption kernel can be executed with no more than eight live physical registers, zero compiler-generated spills, and a combined code + constant footprint not exceeding 512 bytes. Every avoided spill eliminates a data cache round trip, reinforcing the latency reduction promised in H1 and the energy savings in Hypothesis2. We will confirm this hypothesis by inspecting the generated assembly to verify that no lw/sw spill instructions are emitted, analysing the register-allocation logs to ensure the peak live range count never exceeds eight, and using standard binary size tools to check that the kernel (text plus read only data) remains under the 512-byte target. Exceeding any of these limits appearance of spills, more than eight live registers, or a footprint above 512 bytes will falsify Hypothesis 3 and trigger a redesign of our unrolling factor, temporary variable mapping, or S-box implementation.

While custom AES registers and decode paths promise major speed-and-energy gains, they must not compromise the microcontroller-class budget that makes RISCY attractive.

We therefore assert that the added "crypto" register segment, widened decode logic, and single-cycle AES execution slice will enlarge the synthesized core by no more than 15% (in LUTs or μm^2) and reduce the post-place-and-route maximum clock frequency by no more than 5 % on the PYNQ-Z2 FPGA. Success means the design still fits within tight silicon-area and timing margins typical of low-power IoT devices; failure—area growth beyond 1.15× or fMAX drop below 0.95× baseline—would invalidate the trade-off, forcing a rethink of register count, porting, or clock-gating strategy.

Hypothesis 4 (Security): Employing dedicated registers for AES instructions will enhance security by preventing register spilling and limiting exposure of sensitive data in memory, thereby reducing the risk of memory-based leakage.

To evaluate this hypotheses, we will benchmark our design against the software-only baseline by comparing execution cycles, area utilization, compiler output, and other metrics. We will validate functionality through simulation, measure performance on the FPGA, and analyze LLVM IR and assembly code to confirm that our instructions are being used correctly and effectively.

4. Plan

To successfully implement, validate, and evaluate our proposed AES hardware and compiler enhancements, we have divided the project into five distinct work packages. Each package targets a core aspect of the system, from RTL-level design to high-level benchmarking and documentation.

The first work package focuses on modifying the RISCY RTL by integrating two custom AES instructions: `aes32esi` and `aes32esmi`. This involves extending the decode and execute stages of the pipeline, and ensuring that results are properly written back to the register file.

In parallel, the second work package concerns the compiler. We will extend the LLVM backend to support automatic emission of these instructions. This includes writing a custom instruction pattern in the selection DAG and developing a loop unrolling pass tailored to AES round transformations.

The third and fourth packages involve verification. First, we will run simulations and out-of-context (OOC) synthesis in Vivado to ensure that the modified hardware is functionally correct and meets timing requirements. Second, we will deploy the system on the PYNQ-Z2 FPGA to collect runtime performance data. Benchmark results will include cycle counts, area utilization, and power estimates where available.

The final package covers the integration and reporting phase. In this step, we will consolidate all results, verify consistency across tools, and finalize the documentation and evaluation needed for the final report and presentation.

Timeline and Milestones

A detailed overview of the project timeline is provided in Figure 1, which outlines the distribution of tasks across the project weeks. This Gantt chart captures major milestones, including the RTL and LLVM implementation deadlines, simulation phases, and FPGA benchmarking efforts.

Team Responsibilities

Each team member will contribute to all aspects of the system. Rather than splitting development tasks in parallel, we chose to collaborate closely throughout the project's six-week timeframe. We believe this approach maximizes productivity, enabling collective problem-solving and reducing the overhead of individually becoming familiar with the entire system to start development.

Baseline Comparison Strategy

To evaluate the effectiveness of our enhancements, we will compare our final implementation to the original RISCY processor running a software-only AES encryption. Key metrics for this comparison include:

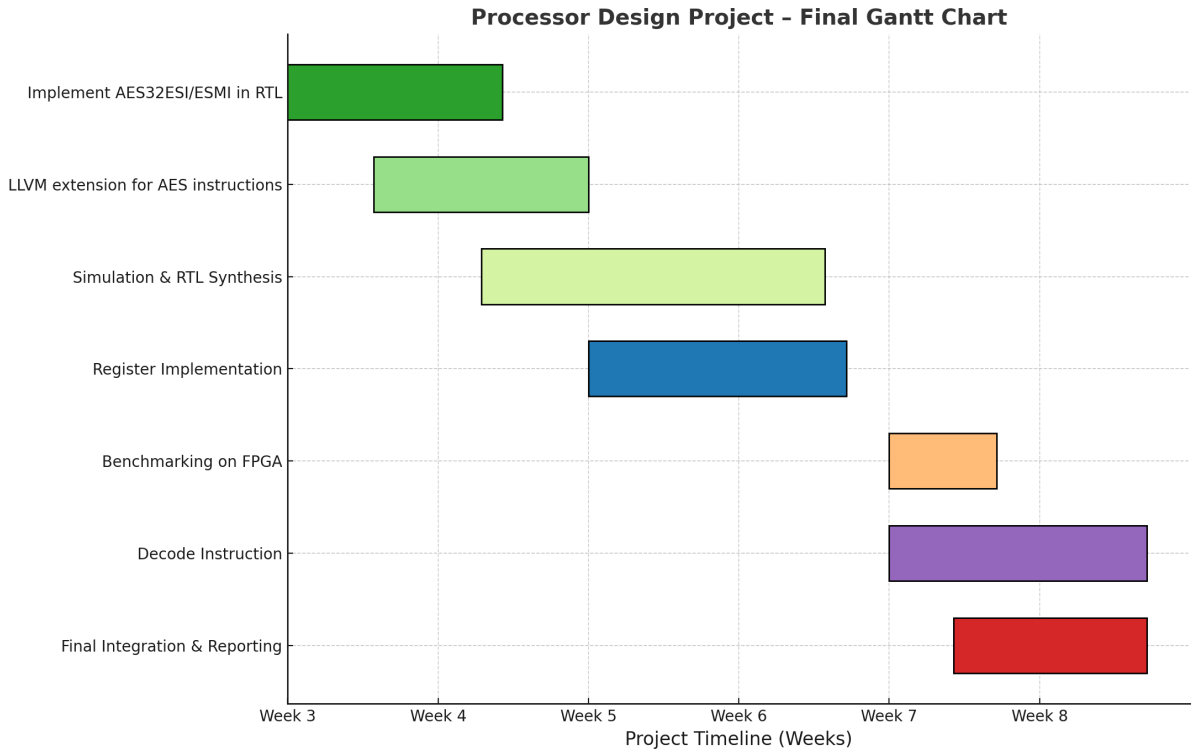

Figure 1: Updated Gantt Chart – PDP Phase 2 Work Packages

- Total number of execution cycles per AES block
- Area utilization based on Vivado post-synthesis reports
- Estimated power usage and switching activity from synthesis tools
- Memory footprint during AES instructions

This comparison will help quantify the trade-offs between performance, resource usage, and implementation complexity.

5. Individual Contribution

All team members contributed to the initial software setup, including attempts at configuring the RISC-V toolchain locally and ultimately migrating to the remote server environment. The team also worked on doing background/literature review about the AES encryption algorithm as well as researching supporting references and related literature related to RISC-V hardware accelerators. Furthermore, the team investigated the baseline system as well as writing the intermediate report.

6. Conclusion

In this intermediate report, we have laid the foundation for our Processor Design Project by analyzing the limitations of the current RISCY-based AES implementation and proposing a tightly integrated hardware–software co-design strategy. Our hypothesis is that by extending the RISCY core with two dedicated AES instructions and enabling their use through compiler-driven loop unrolling in LLVM, we can significantly improve the performance, efficiency, and scalability of AES encryption on an embedded platform.

We began by validating the baseline system, performing literature research, and identifying architectural and compiler-level bottlenecks. Based on this analysis, we developed a clear project plan that includes RTL modifications, LLVM backend extensions, FPGA benchmarking, and a final evaluation framework. Team responsibilities have been distributed to align with each member's strengths, and a structured timeline has been established to ensure steady progress.

The next phase will involve implementing the proposed AES instructions in hardware, modifying the LLVM toolchain to emit them automatically, and benchmarking the system to assess performance gains. We believe that the insights gathered during this first phase—together with the groundwork we have laid—will enable us to deliver a solution that is both technically sound and demonstrably effective.

Ultimately, this project is not only about optimizing AES encryption on RISC-V; it is also about learning how to design systems where hardware and software collaborate seamlessly. The methods, tools, and principles we apply here will be valuable well beyond this project, in future embedded systems work and advanced processor design.

References

[1] A. Waterman, Y. Lee, D. Patterson, and K. Asanović, "The risc-v instruction set manual, volume i: User-level isa," vol. 1, no. 2, pp. 1–80, 2019.

[2] National Institute of Standards and Technology (NIST), "Fips-197: Advanced encryption standard (aes)." Federal Information Processing Standards Publication, 2001.

[3] J. Zambreno, D. Nguyen, and A. Choudhary, "Exploring area/delay tradeoffs in an aes fpga implementation," in *Proceedings of the International Conference on Field-Programmable Logic and Applications (FPL)*, vol. 3203 of *Lecture Notes in Computer Science*, pp. 575–585, Springer, 2004.

[4] Intel Corporation, "Intel® advanced encryption standard (aes) instructions set – white paper." Intel Technical Documentation, 2010.

[5] ARM Limited, "Armv8 cryptography extensions technical overview." ARM White Paper, 2016.

[6] LLVM Project, "Llvm compiler infrastructure documentation: Loop unrolling pass," 2022. Accessed: 2024-04-30.

Appendix A: Jupyter Notebook Output

May 2, 2025

1 Base Riscy Notebook

This notebook is supposed to exemplify how to set up and launch the execution of your core in the FPGA.

It is recommended to make a duplicate of this notebook that you can edit and change all the paths/names from `base_riscy` to `riscy` so you can always keep the initial version of both the overlay and the control of the core in a working state.

1.1 Initialization of the notebook

A series of packages need to be imported, as well as defining a few variables and functions

[1]:
```
from pynq import Overlay
from pynq import PL
from pynq import MMIO

zynq_system = Overlay("/home/xilinx/jupyter_notebooks/riscy/overlays/
 ↪riscv_wrapper.bit", download=False)

ins_mem       = MMIO(0x40000000, 0x8000)
data_mem      = MMIO(0x42000000, 0x8000)
riscv_control = MMIO(0x40008000, 0x1000)
reg_bank      = MMIO(0x40009000, 0x1000)

#Register Bank Addresses:
end_of_test_addr = 0x0 #only bit 0 out of 32.
exec_clk_cycles = 0x4

#Data mem addresses:
end_seq_addr = 0x2000
c_result_check_addr = 0x2004
expected_result_addr = 0x2030
calculated_result_addr = 0x2040

def parse_coe_file(file_path):
    data_values = []
    with open(file_path, 'r') as file:
        lines = file.readlines()
```

```
        start_parsing = False

        for line in lines:
            line = line.strip()

            if start_parsing:
                values = line.split(',')
                data_values.extend([v.strip() for v in values if v.strip()])

            if "memory_initialization_vector=" in line:
                start_parsing = True
                line = line.split("=")[1]  # Get the values after '='
                values = line.split(',')
                data_values.extend([v.strip() for v in values if v.strip()])
                print(data_values)

    return data_values

def parse_and_process_file(filename, write_func):
    data_values = parse_coe_file(filename)
    count = 0
    offset = 0x0

    #print(f"Length of memory file: {len(data_values)} x 32-bits")
    while count < len(data_values):
        write_func(offset, int(data_values[count], 16))
        #print(f"Write {hex(offset)}: {hex(int(data_values[count], 16))}")
        offset += 0x4
        count += 1
```

```
/usr/local/lib/python3.6/dist-packages/pynq/pl_server/device.py:594:
UserWarning: Users will not get PARAMETERS / REGISTERS information through TCL
files. HWH file is recommended.
  warnings.warn(message, UserWarning)
```

1.1.1 Downloading the FPGA image (bitstream)

Now we will download the bistream into the PL of the FPGA:

[2]:
```
zynq_system.download()
```

1.2 Control of the RISCY Core

Two top level pins of the RISCY core are connected to a control block (reboot_riscv_0 in the bd).

This block has a base address of `0x40008000`, but the relative offset of the 32-bit register controlling the connected riscv instance pins is `0x10`. This 32-bit register is connected as follows:

- bit 4 = fetch_enable port

- bit 0 = reboot port

The default value of the control register is: `0x00000000`.

1.2.1 Rebooting the RISCY Core

We will start by asserting high and deasserting the register bit that is connected to the reboot pin of the riscv core:

[3]:
```
data = riscv_control.read(0x10)
print(f"Read: {hex(data)}")
riscv_control.write(0x10, 0x00000001)
data = riscv_control.read(0x10)
print(f"Read: {hex(data)}")
riscv_control.write(0x10, 0x00000000)
data = riscv_control.read(0x10)
print(f"Read: {hex(data)}")
```

```
Read: 0x0
Read: 0x1
Read: 0x0
```

1.2.2 Programming the instruction memory

Now that the riscv core is rebooted and stalling we can proceed to write our desired initialization sequences in the instruction memory (blk_mem_gen_1):

[4]:
```
parse_and_process_file("/home/xilinx/jupyter_notebooks/riscy/mem_files/code.
↪coe", ins_mem.write_reg)
```

```
['00000013']
```

[5]:
```
# We can also read the instruction memory:
data=0x0
addr=0x0
for i in range(20):
    data = ins_mem.read(addr)
    print(f"Address is: {hex(addr)}; Value is: {hex(data)}.")
    addr += 0x4
```

```
Address is: 0x0; Value is: 0x13.
Address is: 0x4; Value is: 0x13.
Address is: 0x8; Value is: 0x13.
Address is: 0xc; Value is: 0x13.
Address is: 0x10; Value is: 0x13.
Address is: 0x14; Value is: 0x13.
Address is: 0x18; Value is: 0x13.
Address is: 0x1c; Value is: 0x13.
Address is: 0x20; Value is: 0x13.
Address is: 0x24; Value is: 0x13.
```

```
Address is: 0x28; Value is: 0x13.
Address is: 0x2c; Value is: 0x13.
Address is: 0x30; Value is: 0x13.
Address is: 0x34; Value is: 0x13.
Address is: 0x38; Value is: 0x13.
Address is: 0x3c; Value is: 0x13.
Address is: 0x40; Value is: 0x13.
Address is: 0x44; Value is: 0x13.
Address is: 0x48; Value is: 0x13.
Address is: 0x4c; Value is: 0x13.
```

1.2.3 Programming the data memory

Same as before with the data memory (blk_mem_gen_2 in the bd):

[6]:
```
parse_and_process_file("/home/xilinx/jupyter_notebooks/riscy/mem_files/data.
↪coe", data_mem.write_reg)
```

```
['7B777C63']
```

[7]:
```
# We can also read the data memory:
data=0x0
addr=0x0
for i in range(20):
    data = data_mem.read(addr)
    print(f"Address is: {hex(addr)}; Value is: {hex(data)}.")
    addr += 0x4
```

```
Address is: 0x0; Value is: 0x7b777c63.
Address is: 0x4; Value is: 0xc56f6bf2.
Address is: 0x8; Value is: 0x2b670130.
Address is: 0xc; Value is: 0x76abd7fe.
Address is: 0x10; Value is: 0x7dc982ca.
Address is: 0x14; Value is: 0xf04759fa.
Address is: 0x18; Value is: 0xafa2d4ad.
Address is: 0x1c; Value is: 0xc072a49c.
Address is: 0x20; Value is: 0x2693fdb7.
Address is: 0x24; Value is: 0xccf73f36.
Address is: 0x28; Value is: 0xf1e5a534.
Address is: 0x2c; Value is: 0x1531d871.
Address is: 0x30; Value is: 0xc323c704.
Address is: 0x34; Value is: 0x9a059618.
Address is: 0x38; Value is: 0xe2801207.
Address is: 0x3c; Value is: 0x75b227eb.
Address is: 0x40; Value is: 0x1a2c8309.
Address is: 0x44; Value is: 0xa05a6e1b.
Address is: 0x48; Value is: 0xb3d63b52.
Address is: 0x4c; Value is: 0x842fe329.
```

1.2.4 Starting the execution of the RISCY Core

Now we can proceed to assert high to the fetch enable pin of the RISCY core:

[8]:
```
data = riscv_control.read(0x10)
print(f"Read: {hex(data)}")
riscv_control.write(0x10, 0x0000_0010)
data = riscv_control.read(0x10)
print(f"Read: {hex(data)}")
```

```
Read: 0x0
Read: 0x10
```

1.2.5 Check the results of the execution of the core

First we check that the core finished execution. Then we check expected and calculated results by the core. Core also compares them and saves a sequence to indicate if the match or not. Check numnber of clk cycles it took to execute the C code.

[9]:
```
data = reg_bank.read(end_of_test_addr)
if data == 1:
    print("Test execution finished!")
else:
    print("Test execution DID NOT finish!")
```

```
Test execution finished!
```

[10]:
```
for i in range(4):
    data = data_mem.read(expected_result_addr + (i * 4))
    print(f"[{i}] Expected result: {hex(data)}")
```

```
[0] Expected result: 0xfba50914
[1] Expected result: 0x714bf41f
[2] Expected result: 0x2e25aabe
[3] Expected result: 0xaaf9080f
```

[11]:
```
for i in range(4):
    data = data_mem.read(calculated_result_addr + (i * 4))
    print(f"[{i}] Calculated result: {hex(data)}")
```

```
[0] Calculated result: 0xfba50914
[1] Calculated result: 0x714bf41f
[2] Calculated result: 0x2e25aabe
[3] Calculated result: 0xaaf9080f
```

[12]:
```
data = data_mem.read(c_result_check_addr)
print(f"C code result check: {hex(data)}")
print(f"0xcafebabe = match; 0xbaaaaaad = no match.")
```

```
C code result check: 0xcafebabe
0xcafebabe = match; 0xbaaaaaad = no match.
```

[13]:
```
data = reg_bank.read(exec_clk_cycles)
print(f"It took: {data} clk cycles.")
```

```
It took: 161441 clk cycles.
```