Benchmarking a RISC-V CPU using CoreMark

Mitu Raj, chipmunklogic.com
May-2025

Contents

Setting the stage

Some of you may already know that I have been blogging a lot on the Pequeno RISC-V CPU at chipmunklogic.com, which I designed from scratch in RTL. The Pequeno is a 5-stage pipelined in-order RV32I processor designed to strike a balance between area and performance.

After completing the design and validation, the first question, which left me pondering was: how good is the processor I have designed? Is it "good enough" for what it was designed for? Did it meet the performance goal? How does my CPU compare with others? Where are the bottlenecks hidden? What can I improve about the pipeline of the CPU?

This whitepaper is a result of the above questions that I asked myself. In summary, this whitepaper discusses the journey of benchmarking a RISC-V processor.

1. Performance of a CPU

In the context of a CPU designed in RTL, performance has two key aspects: timing performance and computational performance. Timing performance is the maximum achievable clock frequency of operation on targeted technology (ASIC/FPGA). But, in the context of a CPU, "performance" usually refers to computational performance that reflects how efficiently it executes instructions. CPI (Cycles Per Instruction) is an important metric to measure the computational performance. For a single-issue in-order processor, the design goal is to achieve a CPI close to 1. However, this is a theoretical ideal value. Bottlenecks push this to values higher than 1. The practical design goal is to approach the value of 1 as closely as possible.

The Pequeno (v1.0) achieved a timing performance of ≈ 115 MHz max. operational clock frequency on the Artix-7 FPGA (-1 speed grade)—pretty satisfactory for an RV32I processor. But the real question in my mind was: how does the CPI trend in the Pequeno under realistic applications? How does it compare to other CPUs out there? That's when I came across the concept of **CPU benchmarking**.

What is CPU benchmarking?

CPU benchmarking is the process of measuring a processor's performance by running a set of standardized programs designed to stress different aspects of the CPU, such as arithmetic computational ability, data handling with memory, and flow of control. The goal of benchmarking is to quantify how efficiently the CPU executes real-world application programs.

"Processor architects make different trade-offs between several design factors, such as the clock frequency, core pipeline, degree of out-of-order execution, number of execution units, cache organization and size, memory hierarchy, etc. The performance of a processor is determined by the microarchitecture bottlenecks (design trade-offs) and how they are exercised." [1]

Benchmark suites compile standardized programs to evaluate CPU performance. Once the benchmark runs, it produces a performance score that can be compared with results from other CPUs to gauge relative performance.

2. Benchmarking RISC-V CPUs

RISC-V is an open-source and extensible ISA. Benchmarking a RISC-V CPU not only validates compliance with the ISA but also helps identify bottlenecks in the design. It helps to validate performance improvements we are trying to achieve with optimizations and ISA extensions added at different stages of development.

CoreMark and Dhrystone

CoreMark and **Dhrystone** are two of the popular industry-standard CPU benchmarks designed to measure the computational performance of processors, especially in embedded systems. Dhrystone has been around for a while (since 1984), being the most popular benchmark. Dhrystone is a simple synthetic benchmark that evaluates integer performance using a mix of string handling, assignments, and control structures. It reports results in *DMIPS* (*Dhrystone MIPS*).

CoreMark® is a relatively new benchmark (since 2009).

"EEMBC's CoreMark® is a benchmark that measures the performance of microcontrollers (MCUs) and central processing units (CPUs) used in embedded systems. Replacing the antiquated Dhrystone benchmark, Coremark contains implementations of the following algorithms: list processing (find and sort), matrix manipulation (common matrix operations), state machine (determine if an input stream contains valid numbers), and CRC (cyclic redundancy check). It is designed to run on devices from 8-bit microcontrollers to 64-bit microprocessors." [2]

CoreMark was created to overcome the shortcomings of Dhrystone, such as susceptibility to compiler optimizations, lack of floating-point operations, and other limitations. CoreMark measures CPU performance using a fixed workload of linked lists, matrix operations, state machines, and CRC calculations, and reports results in *iterations per second* (*CoreMark/MHz*). CoreMark focuses mainly on CPU core performance (ALU and control logic). It doesn't deeply stress-test the memory subsystem, caches, or bus performance. Also, it normalizes the benchmark score to *per MHz*. This makes sense, as CPUs with higher clock rates typically achieve higher benchmark scores, but CPUs with lower clock rates can still outperform in terms of CPI. Hence, the score is normalized with clock frequency.

3. Our Goal – CoreMark!

I hope that's enough theoretical background to kick-start things…. Let's dive into the real challenge! Now that the pros of CoreMark relative to Dhrystone have been understood, let's explore the process of benchmarking a RISC-V CPU with CoreMark.

Our goal is to take a RISC-V processor (a physical silicon chip or an RTL implementation), port and build CoreMark, run it on the bare-metal on the CPU, and evaluate its benchmark score.

To analyse the raw CPU performance, let's assume that no cache is present and all memory accesses are designed to be single-cycle access at Instruction and Data RAMs in the test platform.

Disclaimer

In writing this paper, I used the Pequeno as the target to perform the benchmark. While the process described in this document is largely generic, certain steps may be tailored or biased toward the specifics of the Pequeno architecture. But you will still get the gist of it and be able to apply the idea to your own processor.

4. CoreMark process in a nutshell

1. Port CoreMark to the target platform. Target platform is the SoC (System-On-Chip) built around the processor.
2. Compile and build the ported CoreMark.
3. Generate the binaries and load them onto the Instruction & Data RAMs.
4. Boot the CPU with the binary and execute the CoreMark.
5. Log the results and print them.

5. Requirements in the Processor subsystem/SoC

To perform CoreMark, the processor sub-system/SoC should support:

- A free-running counter – This is also called a **performance counter**. This free-running 32-bit counter must be running on the same clock as the CPU. It is used by the benchmark kernel to accurately time the CPU performance. The counter should be a memory-mapped peripheral, so that software can access it.
- Serial interface like UART – to print benchmark results. The UART should be a memory-mapped peripheral.
- Data & Instruction RAM – to load the binaries of the CoreMark program.

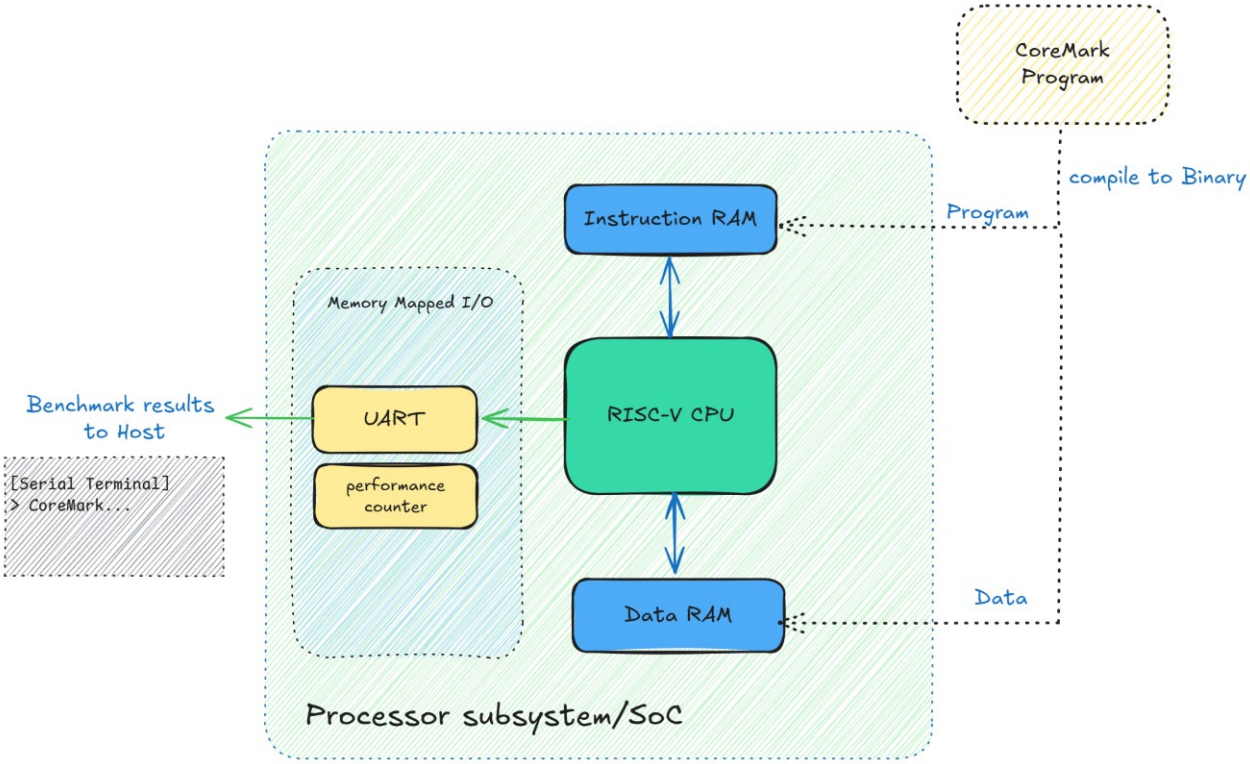


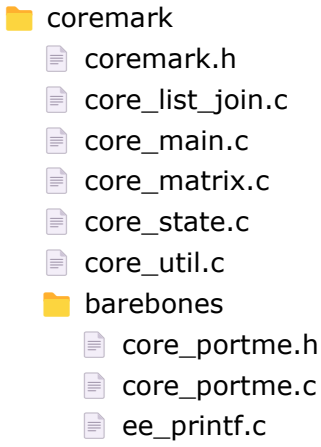
Figure 1: Building CoreMark for a RISC-V SoC

6. CoreMark benchmark suite

The official CoreMark® benchmark suite is available in the EEMBC GitHub repo[3]. Since we are targeting bare-metal applications, the benchmark suite needs to be tuned and modified to cross-compile and run on the target processor. This process is known as **porting**. In this paper, we will target a CPU with RV32I architecture (32-bit RISC-V CPU processor with Integer base instructions set).

To begin, follow these steps:

1. Clone the repo to your local.
2. For bare-metal application, only following directories/files are relevant to us. The rest, you can ignore (unless, you want to dig into Linux porting 😊)



File Name	Description
coremark/	The root directory
coremark.h	The main header file that defines CoreMark-specific macros, function prototypes, benchmark configuration etc.
core_list_join.c	Benchmark kernel for linked list operations. It stresses pointer operations, memory access
core_main.c	The main C code where main() is defined. This will initialize the benchmark, run it, and collect the results to dump.
core_matrix.c	Benchmark kernel for matrix operations. It stresses integer computations.
core_state.c	Benchmark kernel for FSM processing. It stresses control flow.
core_util.c	Contains utility/helper functions for timing, CRC, etc.
coremark/barebones/	Contains the source codes that should be ported to the target platform.
barebones/core_portme.h	Defines platform-specific macros, function prototypes, platform configuration etc.
barebones/core_portme.c	Implements all platform-specific functions.
barebones/ee_printf.c	Contains platform-specific printf() implementation to print the benchmark results.

Table 1: CoreMark directory structure

For more information on the files or directory structure of the benchmark suite, refer to the CoreMark documentation inside *coremark/docs/*

Source files to be ported

In the directory tree of CoreMark, only the files inside the *barebones* directory need to be modified. These are the files you need to port to the architecture of your processor. The rest of the files have no platform dependency, and hence no need to be modified.

7. Configuring CoreMark

Setting up the target platform environment

CoreMark parameters in the header file, *core_portme.h* are configured to be compatible with the target platform environment. The below table describes what is typically defined for an RV32I processor like the Pequeno in *core_portme.h*.

core_portme.h		
MACRO	**VALUE**	**Comments/Justification**
HAS_FLOAT	0	RV32I has no floating point support.
HAS_TIME_H	0	<time.h> std lib function implementation NOT available.
USE_CLOCK	0	<time.h> std lib function implementation NOT available.
HAS_STDIO	0	Bare-metal doesn't support standard I/O functions.
HAS_PRINTF	0	Bare-metal doesn't support standard I/O functions like printf()
COMPILER_VERSION	DON'T CARE	We will use a custom Makefile.
COMPILER_FLAGS	DON'T CARE	We will use a custom Makefile.
MEM_LOCATION	DON'T'CARE	We will use a custom Makefile.
SEED_METHOD	SEED_VOLATILE	Seed is generated using volatile variables.
MEM_METHOD	MEM_STACK	Data is allocated on stack, no dynamic allocation like heap.
MULTI_THREAD	1	Target is a single core processor; one thread per core.
MAIN_HAS_NO_ARGC	1	main() doesn't support arguments.
MAIN_HAS_NO_RETURN	0	main() returns an int value = 0.

Table 2: Configuration macros in core_portme.h

Implementing the timing function

This is probably the most important part of the porting process. You must implement a timing function in the CoreMark. So, what is a timing function, and what is its significance? To understand that, we should understand the CoreMark execution flow.

CoreMark execution is broadly divided into two phases. The untimed part, and the timed part. Below is a summary of the basic CoreMark execution flow in a processor.

1. The processor executes the startup code. // Untimed part, the CoreMark begins only after this step…
2. CoreMark initializes components defined outside main(). // Untimed part
3. CoreMark enters the benchmark loop within main(), executing the benchmark algorithms. // Timed part
4. CoreMark completes the benchmark, and logs the results. // Untimed part

The timed part is the most critical section of CoreMark. It consists of benchmark algorithms that the CPU executes for a specified number of iterations. CoreMark evaluates CPU performance by measuring the execution time, reporting both the number of clock cycles and the actual elapsed real time. This finally translates to the CoreMark score of the CPU.

To measure the timed part of CoreMark, a cycle-accurate timing function must be defined. This is implemented in the barebones_clock() function in *core_portme.c*.

This is a two-step process.

1. Access the performance counter with the respective memory-mapped address.
2. Read the counter inside barebones_clock() and return its current value.

Initializing UART

After the benchmark run, the results are logged in memory. These results can be viewed by reading them from memory and "printing" them. In this context, "printing" essentially means sending the data stream to a standard output. Since we are working in a bare-metal environment, without OS, there is no standard output available. Therefore, the processor sub-system should support a serial interface such as UART. Once the benchmark run is complete, the processor can "print" the results by sending the output to a host system via the UART. The host system can display the results using a serial terminal application like PuTTY or similar tools.

To enable printing data over the serial interface, the UART should be initialized on boot. This routine can be implemented in the portable_init() function in *core_portme.c*.

In the CoreMark database of the Pequeno, I implemented the UART init function in ee_printf.c, which implements all functions related to printing. This function is called in core_portme.c. This was done for better code maintainability. Check out the file core_portme.c in the GitHub repo of the Pequeno[4].

Implementing the print function

Once the benchmark run is complete, the results are printed using the ee_printf() function inside main(). This function must be implemented by the user. As we discussed earlier, the processor can output the results to the host via UART. Therefore, the ee_printf() function should be implemented to access the UART and send the output stream out through its serial interface.

Check out the file ee_printf.c in the GitHub repo of the Pequeno[4].

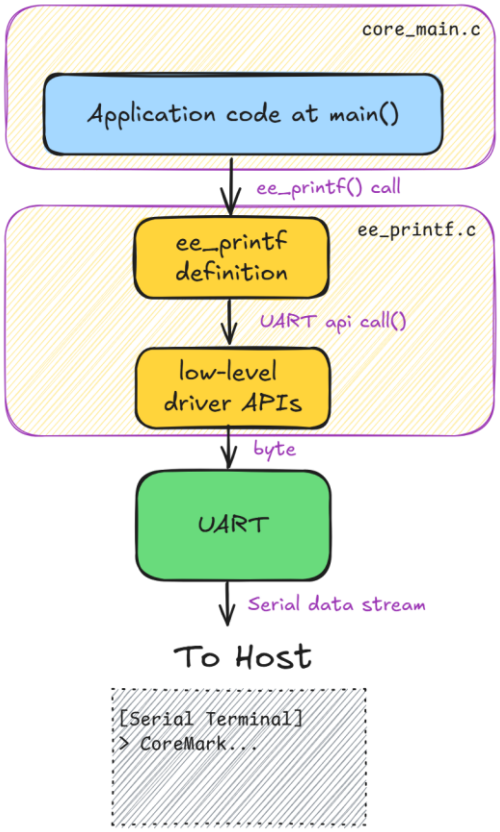


Figure 2: Printing results with UART

8. Adding a startup code

Apart from the CoreMark files, you may need to add a few of your own source files. For a bare-metal application, at minimum, a startup code is required. Since there is no OS running in the processor, this startup code would be responsible for initializing the system on boot, before main() is called. Startup code is typically written in assembly.

In Pequeno, the startup code does the following functions:

1. Initialize the stack pointer.
2. Zero the BSS section.
3. Zero the general-purpose registers (not mandatory though…).
4. Jump to main()
5. Enter a NOP infinite loop after returning from main().

Check out the startup code in the GitHub repo of the Pequeno[4].

9. Compiling and building CoreMark

We are almost there! Two more additional and important files are required to compile and build CoreMark: Linker script and Makefile.

Linker script

Linker script specifies the memory layout to map the various segments/sections of the benchmark code. It also specifies the target architecture, and output ELF format.

In embedded applications, the memory layout is typically divided between Instruction RAM and Data RAM. The *text* segment is always mapped to Instruction RAM. The *data*, *bss* segments are mapped to Data RAM. The *rodata* segment may be mapped to either Instruction RAM or Data RAM, depending on the capability of the target architecture. In case of the Pequeno, *rodata* is mapped to Data RAM.

Startup code must be mapped at the beginning of the *text* segment. And the base address of the *text* segment should match the Reset vector PC of the processor. This ensures the correct boot sequence: execution of startup routine, followed by a handoff to main().

Check out the Linker script in the GitHub repo of the Pequeno[4].

Makefile

Makefile is used to automate the compilation of CoreMark and dump the binaries. For the Pequeno, I used a custom Makefile instead of the default one in the CoreMark repo. The compiler toolchain must be properly configured in the Makefile (RISC-V GCC). All the source files of CoreMark, including the startup code, should be compiled and linked with the linker script. The outputs of the build process are: ELF file, Instruction and Data binaries.

Two CoreMark parameters are typically passed to the compiler during the build.

Parameter	Description
ITERATIONS	No. of CoreMark iterations to be run
CLOCKS_PER_SEC	Core clock frequency in Hz

Table 3: CoreMark parameters passed to Compiler

For an RV32I processor targeting baremetal applications, the following compiler and linker flags are typically used.

Flag	Type	Description
-g	Compiler	Generates debug symbols (to debug with GDB etc.)
-O3	Compiler	Highest optimization, typical for CoreMark.
-march=RV32I	Compiler	Target architecture = RV32I
-mabi=ilp32	Compiler	Target ABI = ILP32 i.e., integer, long, pointer = 32-bit
-ffreestanding	Compiler	Free-standing baremetal environment with no OS and standard libraries.

-fno-builtin	Compiler	Disables using builtin functions like *memcpy*. Instead, forces an explicit function call assuming a custom implementation exists.
-fno-stack-protector	Compiler	Disables stack smashing protection.
-fno-zero-initialized-in-bss	Compiler	Prevents moving zero-initialized globals to *bss* segment. Keeps them in *data* segment.
-mstrict-align	Compiler	Forces aligned memory accesses only.
-static	Linker	Forces static linking.
-nostartfiles	Linker	Prevents linking crt0/builtin startup files.

Table 4: Compiler & Linker flags

Another important point to keep in mind is that the compiler generates call to standard SW based division and multiplication routines because RV32I architecture lacks hardware multiplier and divider. The linker needs to resolve these references by linking against the appropriate standard libraries (such as libgcc) that implement these routines. These library paths should be passed in the linker flags.

Check out the Makefile in the GitHub repo of the Pequeno[4].

10. Interpreting benchmark results

The figure below shows a typical benchmark result obtained after running CoreMark. A successful run prints "Correct operation validated", along with some key performance metrics.

```
mituraj@chipbook:~/chipmunklogic/github/pequeno_riscv/scripts$ . opserial.sh /dev/ttyUSB0 115200
[OPSERIAL]: Serial port /dev/ttyUSB0 opened successfully...
[OPSERIAL]: Reading from serial port:
Starting benchmark on Pequeno with COREMARK 1.0...
2K performance run parameters for coremark.
CoreMark Size    : 666
Total ticks      : 563158780
Total time (secs): 46
Iterations/Sec   : 8
Iterations       : 400
Compiler version : GCC14.2.0
Compiler flags   :
Memory location  : STACK
seedcrc          : 0xe9f5
[0]crclist       : 0xe714
[0]crcmatrix     : 0x1fd7
[0]crcstate      : 0x8e3a
[0]crcfinal      : 0x25b5
Correct operation validated. See README.md for run and reporting rules.
```

Figure 3: CoreMark results on the Pequeno

CoreMark score can be calculated as:

$$CoreMark\ score = \frac{Iterations\ per\ sec}{Core\ clock\ (in\ MHz)}\ CoreMark/MHz$$

11. Troubleshooting

CoreMark must run for at least 10 seconds to generate valid benchmark results. In case of CoreMark fails to run and give outputs as expected, consider the following checks:

1. The No. of iterations is sufficient for the given clock frequency to run the CoreMark for 10 seconds.
2. Verify the linker script and ensure all the sections are aligned and placed correctly.
3. Verify that the startup code is mapped at the beginning of the *text* section, and that the base address of the *text* section matches the Reset vector PC of the processor.
4. Check compiler flags for correctness and compatibility with the target.
5. Check whether the stack pointer is properly initialized, with enough space allocated to prevent stack overflow.
6. Verify the ee_printf() function implementation by running standalone program.
7. Add debug prints at key points in *core_main.c* to trace program flow and see where it gets hung. Please note that these debug prints must be later disabled for the actual benchmark run, if it's in the "timed part" of the code.
8. Worst-case scenario for RTL-implemented processors: If everything else fails, consider that CoreMark acts as a functional ISA validator. Run a comprehensive regression suite to check whether your processor implementation is functionally correct and compliant with the ISA.

Wrapping up

When I first planned to do benchmarking of the Pequeno, I couldn't find a clear, single resource explaining how to benchmark an RTL-designed RISC-V CPU with CoreMark. This held me back for a while, until I began exploring multiple resources and codebases and gradually understood the "behind-the-scenes". What I've shared here is built on weeks of reading, trial & error, and learning by doing. I hope this white paper provides a useful starting point for everyone exploring the same path.

References

[1] "An Introduction to CPU Performance Benchmarks and How This Applies to the Home Market" – ARM, Nov 2021.

[2] EEMBC, the official website of CoreMark®: https://www.eembc.org/coremark/

[3] EEMBC CoreMark GitHub repo: https://github.com/eembc/coremark

[4] Pequeno RISC-V CPU: https://github.com/iammituraj/pequeno_riscv, a 32-bit RV32I processor from chipmunklogic.com