RISCY: A Custom RISC-V SoC

Panagiotis Nanousis, Konstantinos Oikonomopoulos, Konstantin Ziou, Dimosthenis Zempilas
Department of Electrical and Computer Engineering, University of Thessaly, Volos, Greece

***Abstract*—RISC-V is an open and extensible instruction set architecture (ISA) that enables rapid experimentation in CPU and System-on-Chip (SoC) design [1]. This work presents RISC-Y, a custom RV32IM soft-core and accompanying SoC developed as a flexible platform for education, research, and hardware–software co-design. The processor implements a six-stage pipeline, machine-mode Control and Status Registers (CSRs), interrupts, and trap handling mechanisms. The SoC integrates a memory-mapped peripheral architecture and an instruction cache to enhance performance. To address the target FPGA's limited on-chip memory, RISC-Y incorporates a 32-bit SDRAM subsystem and a framebuffer controller via a custom arbiter. A two-stage bootloader, a lightweight filesystem, and a full ELF loader provide a robust firmware environment. The proposed environment provided sufficient performance for the SoC to handle computationally demanding workloads, ranging from games and text renderers to full RTOS implementations like FreeRTOS. While many RISC-V cores exist, few open-source projects bridge the gap between RTL, firmware, and OS implementation on resource-constrained FPGAs.**

The hardware and software source code is publicly available at https://github.com/Nanousis/RiscY.

I. Introduction

This work presents RISC-Y, a custom RV32IM RISC-V System-on-Chip implemented on the Tang Nano 20K FPGA, designed to explore lightweight CPU microarchitecture, memory hierarchy integration, and hardware–software co-design. The system features a six-stage pipeline, a memory-mapped peripheral architecture, an instruction cache, and support for machine-mode CSRs, interrupts, and traps. To overcome the limited on-chip memory, RISC-Y integrates a 32-bit wide SDRAM subsystem with arbitration between the CPU, instruction cache, and framebuffer, enabling larger programs and high-resolution graphics through a dedicated framebuffer controller and Pixel Processing Unit (PPU). A two-stage bootloader, custom filesystem, and a full ELF loader provide a robust software environment, complemented by a high-performance Rust-based emulator for verification and cycle-accurate tracing. The SoC runs a suite of demonstration applications, including games and graphical workloads. Benchmarking with Dhrystone shows 0.87 DMIPS/MHz when executing from BRAM and 0.34 DMIPS/MHz when using SDRAM with caching, highlighting the impact of external memory latency and motivating future memory hierarchy development.

This system not only supports practical applications but also serves as an educational tool for understanding CPU design and firmware development. The following sections present the hardware architecture and the associated system software and development environment.

II. Hardware Architecture

A. RISC-V Baseline Microarchitecture

The RISC-Y SoC (Figure 1) is built on the Tang Nano 20k FPGA. It features an RV32IM-based core with a six-stage pipeline: Instruction Fetch 1 (IF1), IF2, Decode, Execute, Memory, and Write-back. This split-fetch design helps reduce pipeline hazards and keeps memory operations synchronized. On the control side, the CPU supports Machine Mode, using a subset of CSRs to handle interrupts, traps, and system-level tasks.

The SoC features a memory-mapped architecture with a centralized bus interconnect that routes memory and I/O accesses efficiently. An instruction cache (direct-mapped, 256 up to 1024 lines with 64 bytes cache lines) was implemented that enhances performance by reducing the need for repeated memory fetches and allows for the inclusion of a Dynamic RAM (DRAM).

To enable real-world interaction, the SoC integrates several memory-mapped peripherals:

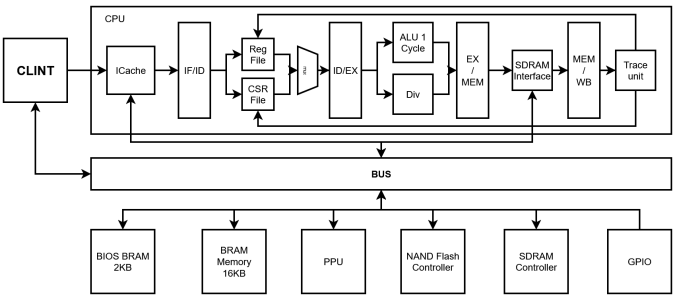

Fig. 1. The RISC-Y SoC diagram

- **Flash Memory Controller:** Allows storage and retrieval of software applications from external Serial Peripheral Interface (SPI) Flash.
- **Pixel Processing Unit (PPU):** Handles HDMI/VGA output with support for text and sprite rendering.
- **Interrupt Controller:** Manages timer, software, and external interrupts.
- **Framebuffer Controller**: A framebuffer module that fetches pixel lines directly from the DRAM for high resolution image rendering and graphics.

B. RISC-V M Extension

The complete RV32M extension was implemented to provide the capability for hardware multiplication and division operations for signed and unsigned 32-bit integers. (Figure 2).

Since a lot of application software depends on these functions, integration was necessary to avoid performance bottlenecks. Without the extension, the software implementation is inefficient, substituting a single hardware instruction with hundreds of software instructions. This overhead caused visible delays in graphics and calculation-intensive programs. Integrated DSPs within the ALU handle multiplication; meanwhile, an adapted Restoring Division algorithm resolves division into quotients and remainders. The division operation is structured into primary stages. First, the logic calculates the minimum number of required operations based on the dividend's leading zeros. Second, it performs bit shifting to bypass unnecessary steps. Finally, the system executes the iterative Restoring Division algorithm. Additionally, in compliance with the RISC-V ISA, a trap signal is raised if the divisor is zero to prevent undefined behavior.

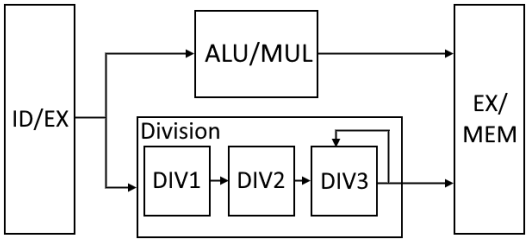

Fig. 2. The execution stage split between one cycle Instructions and Divisions that take multiple cycles.

C. Memory Hierarchy

The system utilizes the Tang Nano 20k's Micron 64 Mb SDRAM (32-bit interface). Although specified for 135 MHz, empirical testing confirmed stable operation at a maximum of 108 MHz, resulting in a measured throughput of 200 MB/s. A robust SDRAM implementation was prioritized to support the memory requirements of larger software applications.

The SDRAM was controlled by the High Speed SDRAM Controller IP from Gowin (Figure 3), meaning that we had to manage the Clock Domain Crossing and synchronization between the SDRAM that runs at 27MHz and the SDRAM at 108MHz.

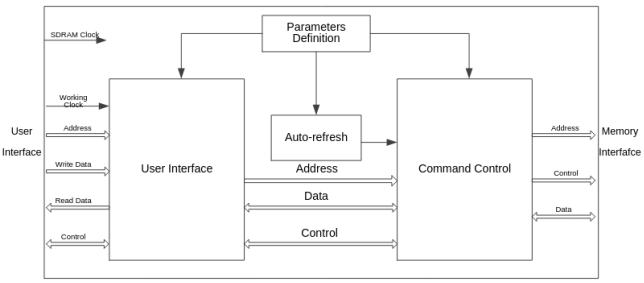

Fig. 3. The block diagram of the Gowin HS SDRAM Controller[2]

To handle memory operations that require multiple clock cycles, the CPU was designed to incorporate a stalling mechanism. A memory-ready signal was added to the pipeline to indicate when the memory is not yet available. This signal prevents the CPU from progressing to the next instruction until the memory operation is complete, ensuring proper synchronization between the processor and memory.

An SDRAM arbitration module acts as the traffic controller for system memory, prioritizing the Framebuffer over the Instruction Cache and CPU data. This prioritization is critical to maintain display stability, as video rendering demands consume the majority of the available bandwidth.

The interface had two modes: either single memory access or burst mode up to 16 words. This meant that the cache and the framebuffer both utilized bursting to maximize the bandwidth of the memory. On the other hand, because there was no data cache, the CPU could access only one item at a time.

The interface also managed the refreshing mechanism of the SDRAM instead of enabling the autorefresh feature of the controller. This was done to have better control on when the memory needed to be refreshed. In general a refresh was issued every $64\mu s$.

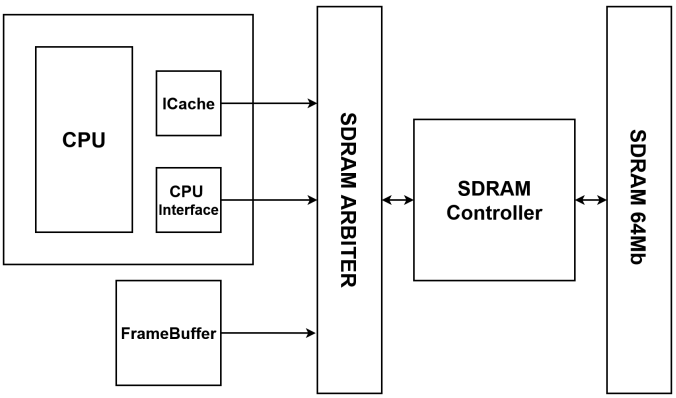

Fig. 4. The block diagram of the SDRAM interface interacting with the CPU and the Framebuffer

1) Instruction Cache: The primary motivation for incorporating an Instruction Cache stemmed from the need to bridge the gap between the CPU's processing speed and the latency of fetching instructions from main memory. Without a cache, every instruction fetch would require accessing Main Memory, which considerably hinders the CPU performance.

To enhance the overall performance and scalability of the architecture, both an Instruction Cache and a Data Cache should be implemented. The Instruction Cache was prioritized as it offered a more significant performance impact.

2) Cache Design: Due to the storage limitations of the FPGA, a direct-mapped instruction cache was designed, with 256 up to 1024 cache lines where each line contained 16 instructions. Being a direct-mapped cache, the memory address is divided into index bits and tag bits. With 1024 cache lines,

we allocate 10 bits for the index, while the remaining bits serve as the tag. One cache line is 16 instructions wide in order to accommodate a full cache line fill with a single burst request from the SDRAM. This provided better bandwidth utilization.

3) Trap Handling and Interrupt Management: To support firmware execution and robust interrupt management, a Core Local Interrupt Controller (CLIC) and a trap handler were introduced. The CLIC module manages external interrupts and handles traps that occur within the CPU pipeline. The trap handler processes these events, ensuring appropriate responses and maintaining the system's reliability. These features are essential for seamless hardware-software integration and efficient exception handling.

D. Interrupt Controller and Control and Status Registers

In the RISC-V architecture, Control and Status Registers (CSRs) are special-purpose registers used to manage and configure the operation of the processor. They enable control over various system-level features such as interrupts, exception handling, performance monitoring, and privilege management. CSRs are accessed using the CSRRW, CSRRS, and CSRRC instructions, which allow reading, writing, and bitwise modification of the registers, respectively. Additionally, immediate variants of these instructions (CSRRWI, CSRRSI, and CSRRCI) simplify operations using constant values. Each CSR is uniquely identified by a 12-bit address, and their accessibility depends on the current privilege level (user, supervisor, or machine).

1) Zicsr extension: The Zicsr extension in RISC-V is a standard extension that provides instructions to interact with Control and Status Registers (CSRs). It is part of the base integer instruction set and is essential for enabling system-level functionality in the RISC-V architecture. The Zicsr extension defines six primary instructions for CSR access and manipulation. Please note that all of these instructions are considered atomic and should not be preempted in the middle of the execution. The instructions are:

- **CSRRW** (CSR Read and Write): Reads the value of a CSR into a general-purpose register while writing a new value to the CSR.
- **CSRRS** (CSR Read and Set): Reads the value of a CSR and sets specific bits using a bitwise OR operation with a specified value.
- **CSRRC** (CSR Read and Clear): Reads the value of a CSR and clears specific bits using a bitwise AND operation with the NOT of the specified mask value.
- **CSRRWI** (CSR Read and Write Immediate): Similar to CSRRW but uses an immediate value for the write operation.
- **CSRRSI** (CSR Read and Set Immediate): Similar to CSRRS but uses an immediate value for the set operation.
- **CSRRCI** (CSR Read and Clear Immediate): Similar to CSRRC but uses an immediate value for the clear operation.

The Zicsr extension is particularly important in systems requiring privilege management, interrupt handling, and performance monitoring, as it provides the essential interface to configure and query the processor's CSRs. It is a prerequisite for many RISC-V profiles and extensions, including the Supervisor mode (S-mode) and Machine mode (M-mode). Processors without system-level functionality, such as those designed for simple embedded applications, may omit this extension if CSRs are not needed.

2) Machine Status (mstatus) Register: The *mstatus* register is a 32-bit read/write CSR formatted as shown in Figure 5 for RV32. It is a critical component of the RISC-V privileged architecture, as it keeps track of and controls the hart's current operating state. The register contains fields for managing interrupt enablement, exception handling, and privilege level transitions.

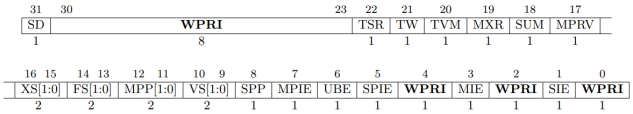

Fig. 5. The *mstatus* register bit layout.

For the purposes of this project, only the functions relevant to interrupts were implemented. The privileged mode bits can be implemented in the future by adding the functionality of an MMU. Hence the most relevant fields in the *mstatus* register are:

- **Machine Interrupt Enable (MIE):** This bit globally enables interrupts for the system. When the MIE bit is set, the processor can handle interrupts that are enabled in the *mie* CSR. If an interrupt occurs, the processor transitions to the trap handler, and the Program Counter (PC) is updated to the value specified in the *mtvec* CSR, which will be discussed later.
- **Machine Previous Interrupt Enable (MPIE):** This bit stores the previous value of the MIE bit before entering a trap. It allows the processor to restore the interrupt enablement state upon returning from a trap handler, ensuring correct operation during privilege transitions.

It is also worth noting that the MBE bit can also be implemented in the future to allow the user to control the endianness of the non-instruction fetch memory accesses.

3) Machine Trap Vector (MTVEC) Register: The mtvec register is a 32-bit WARL read/write register that holds trap vector configuration, consisting of a vector base address (BASE) and a vector mode (MODE). It was implemented to give the CPU the correct location to jump to in case of an interrupt.

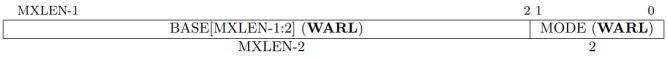

Fig. 6. The *mtvec* register bit layout.

E. Flash Binary File System

To support efficient loading and management of resources within the flash memory, we implemented a minimal file system. This abstraction layer enables the system to access files directly by their names:

Component	Size	Description
Magic Key	8 bytes	Ensures filesystem integrity ("RISCY.FS")
Number of Files	1 byte	Specifies the total number of files
File Entries (repeated for each file)		
File Name	Variable	Null-terminated string (`\0` terminated)
Start Address	4 bytes	Start address of the file in flash
End Address	4 bytes	End address of the file in flash
Thumbnail Sprite	256 bytes	A thumbnail sprite for the file
File Data Section		
File Binaries	Variable	Raw binary contents of the files

F. FPGA Implementation

To build the bitstream for the FPGAs, the Gowin EDA was used [3]. The Gowin EDA is a lightweight FPGA development environment that performs synthesis, placement, and routing. Compared to more feature-rich EDA such as Vivado, it lacks advanced tools like behavioral/timing simulation and extensive reference designs. However, functional verification was conducted using Icarus Verilog and a custom RISC-V Emulator (section IV-A) to ensure correctness before synthesis.

Despite its limitations, Gowin EDA offers a notable advantage in terms of speed. Bitstream generation for the Tang Nano 20K is significantly faster than other FPGA tools, typically completing in under a minute.

The following table summarizes the resource utilization of the RISC-Y SoC across different display configurations:

TABLE I
LUT UTILIZATION FOR DIFFERENT DISPLAY CONFIGURATIONS.

Display Configuration	Look-Up Table (LUT) Usage
HDMI (without objects)	`7,934 LUTs`
VGA (without objects)	`7,119 LUTs`
HDMI (with 20 max objects)	`9,934 LUTs`
VGA (with 20 max objects)	`9,199 LUTs`

The HDMI implementation requires approximately 800 more LUTs than VGA due to the added complexity of DVI encoding. The Tang Nano 20K and Tang Primer 25K (another FPGA by Sipeed that was used) handle the design efficiently, with bitstream generation taking less than a minute

The table below presents the maximum achievable clock frequencies for each FPGA:

TABLE II
MAXIMUM FREQUENCY ESTIMATES FOR EACH FPGA.

FPGA	Maximum Frequency (F_{max})
Tang Nano 20K	`27.9 MHz` (27 MHz oscillator)
Tang Primer 25K	`36.1 MHz` (27 MHz oscillator)

Among the tested FPGAs, the Tang Nano 20K demonstrated the highest stability, achieving a maximum frequency F_{max} that is exactly the same as its oscillator frequency. The Tang Primer 25K is reported slightly lower F_{max} values than its respective crystal oscillators, but operates reliably in practice.

III. SYSTEM SOFTWARE

A. FreeRTOS

FreeRTOS is a lightweight, open-source real-time operating system kernel widely used in constrained embedded systems. It provides preemptive multitasking, mutexes, semaphores, and software timers while maintaining a small footprint (≈ 24 KiB code). This minimal resource footprint makes it ideal for our system.

To streamline FreeRTOS validation, we adapted the RISC-Y SoC to match QEMU's[4] default hardware configuration and memory map. Specifically, we remapped on-chip RAM from `0x1000_0000` to QEMU's standard `0x8000_0000` base address and implemented an NS16550-compatible UART at the same I/O address QEMU uses. With these changes, the unmodified FreeRTOS binary runs identically under QEMU and on our physical hardware, allowing early software debugging and functional verification in the emulator before hardware bring-up.

B. Bootloader

A bootloader is a minimal piece of firmware that executes immediately after reset to initialize essential hardware and load the main application. In the RISC-Y SoC, a two-stage bootloader is employed to allow the system to boot and give the user access to the loaded applications.

1) First Stage Bootloader: The system utilizes a two-stage boot process starting at address 0x0. The First-Stage bootloader is embedded in a 4 KB synthesized ROM and bridges the gap between hardware and software. It initializes the flash interface and loads the larger Second Stage bootloader from off-chip storage into BRAM. This separation ensures that boot logic updates do not necessitate a full FPGA re-synthesis.

2) Second Stage Bootloader: Once loaded into the 16 KB BRAM, the second stage bootloader provides richer services:

Memory Tester. A simple March-C pattern is written to write and read back from each word of RAM to detect stuck-at and addressing faults. Any failure halts further boot and light the screen purple with an error message.

Program Selector (Figure 7). A lightweight menu is rendered on the screen, listing available programs stored in flash metadata. The user navigates the interface, selecting the desired program; selection triggers the ELF loader which communicates information and faults via UART.

ELF Loader. The Executable and Linkable Format (ELF) is the standard binary container for Unix-like systems and is used by FreeRTOS toolchains on RISC-V. An ELF image is composed of three primary parts:

- **ELF Header:** The first 52 bytes (for 32-bit) identify the file as ELF (magic number 0x7F 'E' 'L' 'F'), specify endianness, target architecture, file type (e.g. executable, relocatable), and the entry point address (`e_entry`).

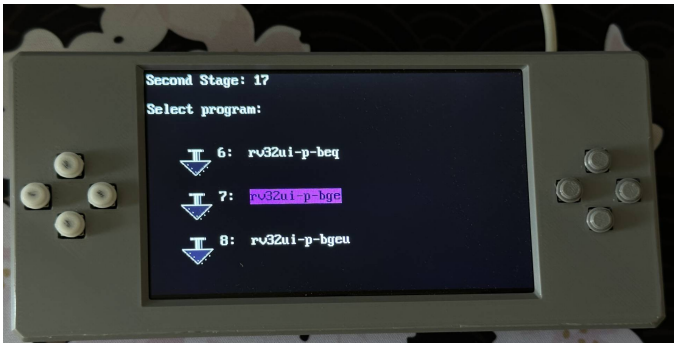

Fig. 7. The program selector allowing the user to select a RISC-V test

- **Program Header Table (Segments):** Each program header (one per loadable segment) describes how to map a region of the file into memory. There are a total of `e_phnum` segments. Key fields include:
  - `p_type`: segment type (e.g. `PT_LOAD`)
  - `p_offset` / `p_vaddr`: file offset for the section headers and the physical address to write them to.
  - `p_filesz` / `p_memsz`: sizes in file and in memory
- **Section Header Table (Sections):** Sections provide finer-grained metadata (e.g. `.text`, `.data`, `.bss`, `.rodata`). Sections are loaded to the appropriate addresses in memory based on the `p_vaddr`.

During boot, the second stage bootloader performs the following steps:

1) Read the ELF header to verify the magic and extract `e_entry` and program header table offset.
2) Iterate over `PT_LOAD` entries, copying `p_filesz` bytes from flash (at `p_offset`) to the target address `p_vaddr`, then zero the remaining `p_memsz`–`p_filesz` bytes (typically the `.bss` section).
3) After all segments are relocated, set the stack pointer and jump to `e_entry`.

This mechanism ensures that code (`.text`), initialized data (`.data`), and zeroed memory (`.bss`) are correctly laid out in RAM before execution begins.

IV. Application Software

A few tools for better application support were also developed along with applications to showcase the features of RISC-Y.

A. Emulator

An RV32I emulator was developed in Rust to verify the correctness of programs before flashing them onto the FPGA. The emulator's main goal was to allow for instruction tracing to cross-verify the CPU's output. The emulator can also turn off logging for faster execution times. The emulator can run 200MI/s with logging turned off and 45MI/s with logging as it is limited by disk I/O latency. Furthermore the emulator also supports both the VGA-Textmode rendering and Framebuffer emulation like the CPU.

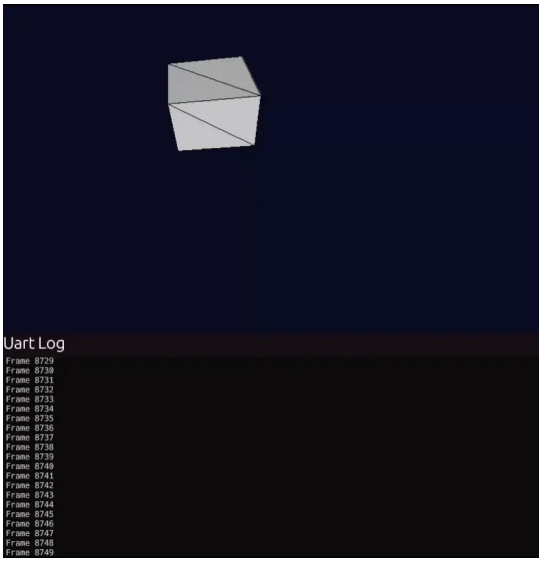

Fig. 8. The emulator running a rendering algorithm and showing the result on the framebuffer.

B. Instruction Tracing

In order to visualize the performance bottlenecks of the CPU, an instruction trace viewer was implemented that takes an execution trace from the CPU and renders it as an execution pipeline (Figure 9).

C. Games

The games developed for the RISC-Y system, include Tetris, Snake and a Pokémon clone. These applications stress-tested the SoC, helping to identify and resolve boundary-case logic errors.

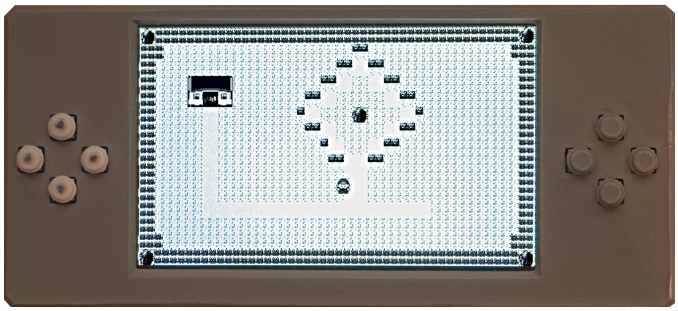

Fig. 10. Pokemon clone running on the Tang Nano 20k

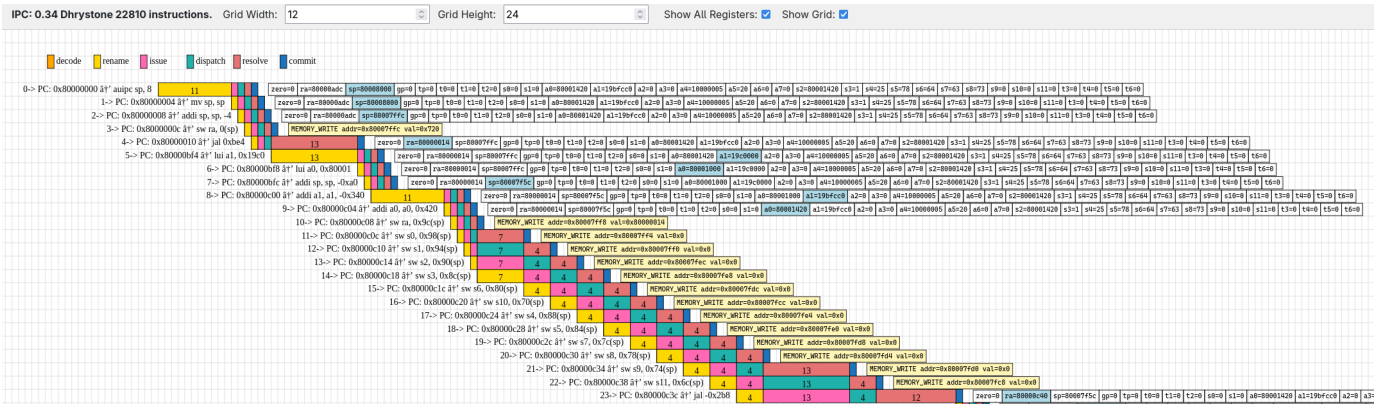

Fig. 9. CPU trace of the Dhrystone benchmarking program

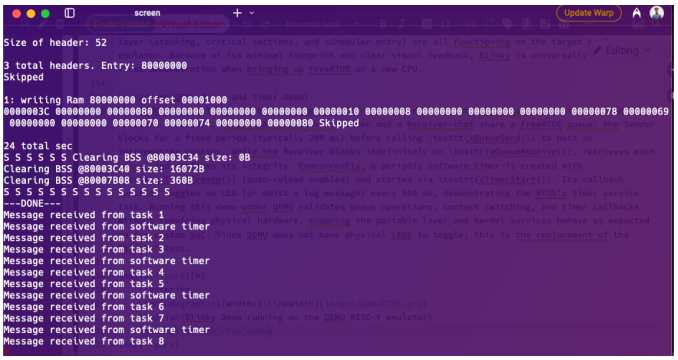

Fig. 11. Blinky Demo running on the real hardware and printing with UART

V. Performance Analysis

We utilized the Dhrystone benchmark to measure the processor's efficiency [5]. The resulting execution traces highlighted performance limitations, guiding our subsequent improvements to the CPU architecture.

A. Dhrystone results

Two tests were run on Dhrystone. One where we used only 32KB of BRAMS as main memory instead of the SDRAM and another where we used the SDRAM along with the instruction cache.

TABLE III
Dhrystone Benchmark Results

Configuration	DMIPS/MHz
RISC-Y with BRAMs	0.87
RISC-Y with SDRAM	0.34

While SDRAM latency currently restricts peak throughput, necessitating a data cache for future improvements, the core demonstrates high efficiency. At 0.87 DMIPS/MHz, it outperforms the similarly complex AMD 80386 (0.4375 MIPS/MHz [6]) by a significant margin.

VI. Conclusion

This work presented RISC-Y, a custom RISC-V System-on-Chip designed to explore CPU microarchitecture, memory hierarchy designs, and hardware–software co-development on resource-constrained FPGAs. Our design demonstrated that a small FPGA can incorporate a full system that can accommodate multiple use cases. From real-time Operating Systems like FreeRTOS, to games and more. With performance comparable to a microcontroller and the flexibility of the FPGA, the SoC can be expanded in numerous ways. Our design can serve as both an educational platform, not only for students to learn computer and peripheral architectures, but also for future research, thanks to its simple and modular design.

References

[1] Andrew Waterman and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.2 and Volume II: Privileged Architecture, Version 1.10*. RISC-V Foundation. 2017. URL: https://riscv.org/specifications/.

[2] Gowin. *Gowin SDRAM Controller IP*. 2025. URL: https://cdn.gowinsemi.com.cn/IPUG279E.pdf.

[3] GOWIN Semiconductor Corp. *Gowin Software User Guide*. SUG100. Available: https://www.gowinsemi.com. Guangdong Gowin Semiconductor Corporation. 2025.

[4] QEMU team. *Quick Emulator (QEMU)*. n.d. URL: https://www.qemu.org/.

[5] Reinhold P. Weicker. "Dhrystone: A synthetic systems programming benchmark". In: *Communications of the ACM* 27.10 (1984), pp. 1013–1030.

[6] Roy Longbottom. *Dhrystone Benchmark Results On PCs*. 2017. URL: http://www.roylongbottom.org.uk/dhrystone%20results.htm.