RISCY: A Custom RISC-V SoC

1st Panagiotis Nanousis 03599
2nd Konstantinos Varakliotis 3460
University Of Thessaly

***Abstract*—This project presents the development of a complete system built around a RISC-V soft-core, enabling users to execute applications on an FPGA. The primary objective was to create a physical platform for testing various architectural concepts, integrating seamless communication between hardware and software components.**

The project addressed key challenges in both architectural and software domains. These included implementing an interrupt controller and designing firmware that effectively leverages it for enhanced software support. Beyond its technical contributions, this work provides a valuable resource for understanding CPU architecture and firmware development. It establishes a robust foundation for reliable embedded applications, supports broader peripheral integration, and serves as a tool for education and experimentation in CPU and system design.

I. Introduction

RISC-V is an open-source instruction set architecture (ISA) celebrated for its simplicity, extensibility, and modularity. Its adaptability has made it a preferred choice for developing custom embedded systems and experimental architectures.

Despite RISC-V's flexibility, practical systems that bridge hardware and software for testing architectural concepts remain limited. Existing solutions are often complex with little to no documentation on how to integrate new components on the core. This project addresses this problem by creating a simple core that can be easily understood and modified for the needs of the user.

The primary objectives of this work are to design and implement a RISC-V soft-core, integrate peripherals, and develop robust firmware support. These contributions aim to enhance hardware-software interaction and demonstrate the feasibility of deploying reliable embedded applications using RISC-V.

This system not only supports practical applications but also serves as an educational tool for understanding CPU design and firmware development.

II. Terminology

This section defines key terminology used throughout the paper to ensure clarity and consistency.

A. RISC-V Terminology

- **Hart (Hardware Thread)**: A hardware thread (hart) refers to an independently executing RISC-V core within a processor. In a single-core CPU, there is only one hart, whereas multi-core processors contain multiple harts that execute instructions in parallel.
- **RV32I**: The base integer instruction set for 32-bit RISC-V processors, which includes essential arithmetic, logical, load/store, and control flow instructions.
- **CSR (Control and Status Register)**: Special-purpose registers used for system control, interrupt management, and privilege level management.
- **Trap**: A mechanism used to handle exceptions and interrupts, transferring control to a predefined software handler.
- **M-Mode (Machine Mode)**: The highest privilege level in RISC-V, allowing direct control over system hardware, including memory protection and interrupts.

B. Processor Architecture Terminology

- **Pipeline**: A technique used in CPU design where instruction execution is divided into multiple stages (e.g., Fetch, Decode, Execute, Memory, and Write-back) to improve performance.
- **RAW (Read After Write) Hazard**: A data dependency issue where an instruction tries to read a register before a previous instruction has written to it.

C. Interrupt and Exception Handling

- **Interrupt**: An event triggered by hardware or software that temporarily halts the CPU's execution to handle a higher-priority task.
- **Exception**: An unexpected condition (e.g., illegal instruction, memory access violation) that requires special handling by the CPU.

D. Memory and Peripheral Terminology

- **BRAM (Block RAM)**: On-chip memory available in FPGAs, used for storing data and instructions.
- **SPI Flash**: A type of non-volatile memory used for storing firmware and bootloader code.
- **CLINT (Core Local Interrupt Controller)**: A module responsible for handling timer and software interrupts.

III. The CPU

A. The original architecture

The project started as a continuation of the project from the fellow students Apostolos Giannousas, Thanasis Kastoras and Ioanna Panagou. The original architecture was a 5 stage pipelined core that supported only RV32I (integer) instructions and asyncrhonous memory. The need for synchronous memory was the first aspect that was adress in order to be able to use FPGA's BRAMS to handle memory and not waste valuable register space. Another problem that arose was the need for cpu stalls when waiting to access memory. Hence before working on the firmware we had to change the architecture to fix the above issues.

B. Architectural changes

Changing the architecture was an important part of fixing the above issues and allow us to have a functional processor.

1) Loads and Stores: Executing load and store instructions was the easiest to make first since we could change when the cpu sent a memory request. Instead of waiting for the memory stage to ask for an address, the execution stage sends the control signals and the memory stage acts as an intermediate step to wait for the memory to arrive. The read instruction was changed in order to send the read enable signal to the memory from the execution stage. Thus the memory worked as an intermidiate pipeline between the execution and the memory stage. The write enable signal was sent on the memory stage creating a problem where the cpu could have both read enable and write enable at the same time. To counteract this we checked if Read after Write (RAW) was on the same memory address. If that was the case, the memory would forward the write data to the read data. If not, the cpu had to stall after the identification stage in order to allow the write to finish. This way, there was no data hazzard that could happen on the data memory.

2) Instruction Fetch: The instruction Fetch stage was a bit harder to change since there was no way of knowing what the program counter would be in advance. This is why one more stage was added when fetching instructions. This meant that the program counter would be sent to memory on IF1 stage and we would have our instruction on the IF2 stage. This allows for synchronization between the IF2 stage with the rest of the pipeline, minding carefully bubbles and stalls. Hence the memory would act as a intermediate pipeline between the IF1 stage and the IF2 stage before being sent to the IF/ID pipeline.

3) CPU Stalls: 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.

4) Control and Status Registers (CSRs): To enable software-level control and monitoring of the CPU, additional Control and Status Registers (CSRs) were implemented based on the RISC-V specifications. These registers provide critical information about the CPU's state and allow software to configure various functionalities, such as interrupt masks and architectural extensions. This addition enhances the system's flexibility and programmability.

5) 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.

IV. Instruction Cache

A. Motivation

In order to improve the performance and capability of extension of the entire architecture, both and Instruction Cache and a Data Cache should be implemented. At this stage, the Instruction Cache is prioritized since it affects more the performance of the architecture, while being the more complicated one to incorporate with the rest of the CPU. At this point, the Main Memory of the architecture consists solely of the FPGA's BRAMS, however, in the future, a RAM should be implemented so that there is larger and more flexible storage. If this is the case, the instructuion cache implementation should provide a very fast way for the instructions to interact with the CPU without comprimising the architecture.

B. Cache Implementation

Due to the limitations of the FPGA's storage capabilities, we opted for a direct-mapped cache consisting of 128 cache lines of 4 bytes each line (one 32-bit instruction). Since it is a direct-mapped cache, we split the address into the index bits and the tag bits. For 128 cache lines, we must use 8 bits for the index, while the remaining bits are used for the tag. If we require specific bytes from the cache, there is an implemented mechanism from the the CPU architecture that, in essence, stores the entire address in the cache apart from 2 bits called *byte select vector* that chooses the specific bytes from the memories.

Furthermore, in order to improve the distribution of the addresses in the cache, we utilize a hashing mechanism that instead of using the last 8 bits of the address to calculate the cache index, we calculate the XOR result of the first 8 and the next 8, so that they are more evenly distributed and less tags are fitted in the same index. As far as the implementation is concerned, we valid bits in order to swiftly determine whether a cache line is used. If we require reading from the cache, there is an output *hit* that signals the cache controller if there is a hit or not.

C. Cache Controller

A crucial part in the integration of a cache in the RISC-V CPU architecture is the interaction between the different memories and the actual CPU. The Cache Controller is responsible for controlling the traffic of data and instruction between the CPU, the Cache and the Main Memory. Inside the Cache Controller, we instantiate the modules for the Instruction Cache and the Main Memory. The Finite State Machine (FSM) that controls the flow of the controller comprises of states for the Cache, the Main Memory, a state to write the instruction to the Cache and a Wait state that stalls for one cycle.

The Cache State, behaves as the Idle state of the FSM, due to the fact that it is the first state in the Cache Controller that determines the flow of the FSM. In this state, we should have received the signal *hit* from the Cache that pinpoints whether or not the PC that the CPU has requested exists in the Cache. If it does, we do not need to inquire about it for the Main Memory and, as far as the Cache is concerned, we

should not stall and we should proceed with the next PC. However, if there is a need for data from the Main Memory, indicated by the write/read enable signals, we stall the CPU and we proceed with the Main Memory State. Of course, if the PC is not found in the Cache, we must proceed to the Main Memory State anyways in order to retrieve the corresponding instruction and, therefore, we must stall the CPU regardless of the need for data.

In this state of the project, the Instruction Cache works only in behavioural simulation, however, it is planned to put it inside the BRAM so that it can be tested in actual hardware.

In the Main Memory State, the first variable we must check is the *ready* signal from the Main Memory that indicates whether or not the Main Memory is busy from reading. If this is the case, we loop back to the same state. When the Main Memory is finished reading, the Instruction is retrieved from the Main Memory, however, since we have decided to implement a Write-Allocate Cache, when the instruction is received we must write it back to the Cache. The FSM is informed to move to the *Write to Cache* State by the hit signal being zero. If there is a need to write or read data from the Main Memory, the read data is saved in a register and we must disable the Main Memory's capability for reading and writing until the next time it is needed.

If we have read the instruction, we must devote one cycle to write to the instruction cache so that the instruction exists in the cache the next time it is read. There is a variable called *instr enable* that acts like a safeguard to prevent the special case of a branch, we will address this problem and its solution in a later section. If the instructions are enabled, we can move on to the Wait State.

Finally, the Wait State, acts as an intermediate state between the end of the Cache Controller and its beginning to address the data sychronization issues that will be delved into later.

D. Incorporating into the CPU

One of the most challenging aspects of the project was to integrate the completed cache controller with the rest of the CPU architecture. Since the CPU architecture has been proven to work very well, we decided to make as little changes as possible to the rest of the architecture in order to solve the different issues that have risen since the incorporation of the Cache. The most prominent issue that we faced, was the synchronization between the CPU and the Cache. Having decided not to make alterations to the CPU architecture, we meddled with the timing of the outputs of the Cache Controller in order to ensure both the data and thee instructions are being sent correctly.

1) PC Timing: For the cache to be effective, in the case of a hit, it must deliver the instruction in the next possible cycle. So, when the new PC is received, in the next cycle it should be decided if the CPU will stall or not. This means that, when the PC arrives, in the next cycle the resulting hit signal is received and in the next cycle the CPU will have stalled or not. To accomplish this, the release of the stall cycle is timed to be exactly where it needs to be so that, as far as the CPU is concerned, the transition between the instruction is, in the ideal scenario, seamless.

2) Consecutive Hits: Another issue that was exposed through the testing of the CPU and the changes made to fit the timing of the PC was a missed instruction after consecutive cache hits. Because we achieved a seamless transition between the Cache and the CPU by changing the PC earlier, this caused the PC to change early and an instruction to be missed To resolve this, instead of the Main Memory having the same PC as the Cache, we gave it one PC earlier than the current one so that when the PC is changed and it is time to read the instruction from the memory, it will be correct.

3) Data Output Synchronization: One more issue of the same nature was the synchronization of the data outputs. Again, the decision to not change the CPU architecture made it essential to change when the Data is given to the CPU inside the FSM to fit with the pipeline. So, even though the data is available in the Main Memory State, it is not given in this state and it is instead held inside a register and it is given in the next state. For the same reason, it is a essential to create the Wait state so that the pipeline is correctly synchronized with the cache controller.

4) Branches: Lastly, there was a problem regarding the branch instructions. The CPU stall signal is mostly handled by the cache controller so that it can solve the PC timing problems as discussed earlier. However, the CPU when a branch instruction is detected the stall signal is overridden for the PC to move to its new location. In the Cache Controller timeline, this unpredictable PC change happens during the write instruction state and to avoid any issues, the control stall module of the CPU sends the *instr enable* signal to the cache to alert it of this change and, in turn, it does not proceed with the FSM, and waits until the signal is inactive (one extra stall cycle) for the FSM to continue.

V. Interrupt Controller

A. 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).

CSR Address [11:10]	[9:8]	[7:4]	Hex	Use and Accessibility
Unprivileged and User-Level CSRs				
00	00	XXXX	0x000–0x0FF	Standard read/write
01	00	XXXX	0x400–0x4FF	Standard read/write
10	00	XXXX	0x800–0x8FF	Custom read/write
11	00	0XXX	0xC00–0xC7F	Standard read-only
11	00	10XX	0xC80–0xCBF	Standard read-only
11	00	11XX	0xCC0–0xCFF	Custom read-only
Supervisor-Level CSRs				
00	01	XXXX	0x100–0x1FF	Standard read/write
01	01	0XXX	0x500–0x57F	Standard read/write
01	01	10XX	0x580–0x5BF	Standard read/write
01	01	11XX	0x5C0–0x5FF	Custom read/write
10	01	0XXX	0x900–0x97F	Standard read/write
10	01	10XX	0x980–0x9BF	Standard read/write
10	01	11XX	0x9C0–0x9FF	Custom read/write
11	01	0XXX	0xD00–0xD7F	Standard read-only
11	01	10XX	0xD80–0xDBF	Standard read-only
11	01	11XX	0xDC0–0xDFF	Custom read-only
Hypervisor and VS CSRs				
00	10	XXXX	0x200–0x2FF	Standard read/write
01	10	0XXX	0x600–0x67F	Standard read/write
01	10	10XX	0x680–0x6BF	Standard read/write
01	10	11XX	0x6C0–0x6FF	Custom read/write
10	10	0XXX	0xA00–0xA7F	Standard read/write
10	10	10XX	0xA80–0xABF	Standard read/write
10	10	11XX	0xAC0–0xAFF	Custom read/write
11	10	0XXX	0xE00–0xE7F	Standard read-only
11	10	10XX	0xE80–0xEBF	Standard read-only
11	10	11XX	0xEC0–0xEFF	Custom read-only
Machine-Level CSRs				
00	11	XXXX	0x300–0x3FF	Standard read/write
01	11	0XXX	0x700–0x77F	Standard read/write
01	11	100X	0x780–0x79F	Standard read/write
01	11	1010	0x7A0–0x7AF	Standard read/write debug CSRs
01	11	1011	0x7B0–0x7BF	Debug-mode-only CSRs
01	11	11XX	0x7C0–0x7FF	Custom read/write
10	11	0XXX	0xB00–0xB7F	Standard read/write
10	11	10XX	0xB80–0xBBF	Standard read/write
10	11	11XX	0xBC0–0xBFF	Custom read/write
11	11	0XXX	0xF00–0xF7F	Standard read-only
11	11	10XX	0xF80–0xFBF	Standard read-only
11	11	11XX	0xFC0–0xFFF	Custom read-only

Fig. 1. Allocation of RISC-V CSR address ranges.

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 stopped 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 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) Implementation of CSR Registers: The implementation of CSR registers required the development of a dedicated module to manage their storage, as well as to handle read and write operations. Additionally, the CPU had to be modified to accommodate different register types, extending beyond the standard integer registers (‘x0‘–‘x31‘). A key challenge in this process was mitigating potential data hazards, such as Write-After-Read (WAR) and Read-After-Read (RAR), which could occur when CSR values are read and subsequently used as source registers in later instructions.

To address this, an additional internal register was introduced within the pipeline to track the type of register being executed at each stage. This enhancement ensures the core can seamlessly handle various register types, including CSRs, without conflict. Furthermore, by updating the Control Forward module, data dependencies involving CSRs are resolved efficiently, allowing the pipeline to operate without stalling.

This modular design also simplifies the integration of new register types. For instance, implementing floating-point registers (‘f0‘–‘f31‘) becomes more straightforward with minimal adjustments to the existing framework. By establishing this flexible foundation, the system is better equipped to support future extensions and advanced functionality.

The ALU of the CPU was leveraged to implement the CSR instructions with minimal modifications. Two additional states were introduced to the ALU module to support the ‘CSRRC‘ and ‘CSRRW‘ instructions.

For the ‘CSRRC‘ instruction, a state was added to enable the ALU to perform a bitwise AND operation between the CSR value and the inverse of the source register value. This ensures that the specified bits are cleared in the CSR. For the ‘CSRRW‘ instruction, a straightforward passthrough state was implemented, allowing the ALU to read the value of the selected CSR while simultaneously writing the value of the source register to it.

These minimal changes to the ALU streamline the handling of CSR instructions, maintaining efficiency and simplicity while expanding the CPU's functionality to accommodate system-level operations.

B. Important Control and Status Registers

The privilege mode manual mentions that only a handful of control and status registers need to be implemented in order to handle interrupts. The specification mentions that some register may need to be;

- **Reserved Writes Preserve Values, Reads Ignore Values (WPRI)**: Some whole read/write fields are reserved for future use.
- **Write/Read Only Legal Values (WLRL)**: Some read-/write CSR fields specify behavior for only a subset of possible bit encodings, with other bit encodings reserved. Software should not write anything other than legal values to such a field, and should not assume a read will return a legal value unless the last write was of a legal value, or the register has not been written since another operation (e.g., reset) set the register to a legal value.
- **Write Any Values, Reads Legal Values (WARL)**: Some read/write CSR fields are only defined for a subset of bit encodings, but allow any value to be written while guaranteeing to return a legal value whenever read. Assuming that writing the CSR has no other side effects, the range of supported values can be determined by attempting to write a desired setting then reading to see if the value was retained.

This is not yet implemented although it should be a simple change on the write part of the register file.

Below are the registers that were implemented:

1) Machine Status (mstatus) Register[1, pp. 34-43]: The *mstatus* register is a 32-bit read/write CSR formatted as shown in Figure 2 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.

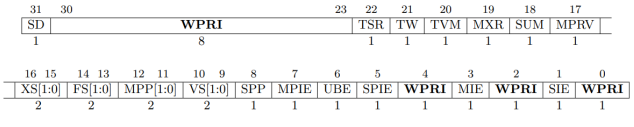

Fig. 2. The *mstatus* register bit layout.

For the purposes of this project, only the functions relevant to interrupts were implemented. The priviledge 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.

2) Machine Trap Vector (MTVEC) Register[1, pp. 43]: 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).

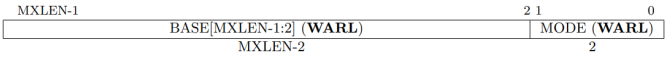

Fig. 3. The *mtvec* register bit layout.

Value	Name	Description
0	Direct	All exceptions set `pc` to BASE.
1	Vectored	Asynchronous interrupts set `pc` to BASE+4×cause.
≥2	—	*Reserved*

Fig. 4. Encoding of the mtvec MODE

The value in the BASE field must always be aligned on a 4-byte boundary, and the MODE setting may impose additional alignment constraints on the value in the BASE field. In the context of this project, the two least significant bits should always be 0, to indicate that the traps are all handled by one trap vector. This is done for simplicity in the software implementation of the trap handler. Hence all traps cause the pc to be set to the address in the BASE field.

3) Machine Interrupt Registers (mip and mie)[1, pp. 46-48]: The mip register is a 32-bit read/write register containing information on pending interrupts, while mie is the corresponding 32-bit read/write register containing interrupt enable bits Interrupt cause number i (as reported in CSR mcause) corresponds with bit i in both mip and mie. Bits 15:0 are allocated to standard interrupt causes only, while bits 16 and above are designated for platform or custom use.

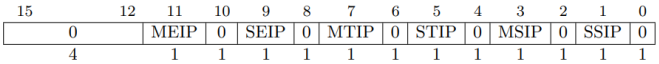

Fig. 5. Encoding of the mie and mpi CSR registers

For an interrupt i to be handled both of the following need to be true; The MIE bit of the mstatus register needs to be set, and the corresponding i bit needs to be set on the mie CSR. If both are true, then the core needs to handle the interrupt as soon as possible after returning from an x-RET instruction. The mip CSR is a read only register that informs the hart that an interrupt i is pending.

Bits mip.MEIP and mie.MEIE are the interrupt-pending and interrupt-enable bits for external interrupts. MEIP is read-only in mip, and is set and cleared by the platform-specific interrupt controller that was developed.

Bits mip.MTIP and mie.MTIE are the interrupt-pending and interrupt-enable bits for machine timer interrupts. MTIP is

read-only in mip, and is cleared by writing to the memory-mapped machine-mode timer compare register.

Bits mip.MSIP and mie.MSIE are the interrupt-pending and interrupt-enable bits for software interrupts. MSIP is read-only in mip, and is written by accesses to memory-mapped control registers, which are used by remote harts to provide machine-level interprocessor interrupts. A hart can write its own MSIP bit using the same memory-mapped control register. Since the system has only one hart, the delivery of machine-level interprocessor interrupts through external interrupts (MEI) instead, hence the mip.MSIP and mie.MSIE are both read-only zeros.

Since supervisor priviledge mode is not yet implemented, all S-x interrupts are always set to zero.

4) Machine Cause Register (mcause)[1, pp. 52-54]: The mcause register is a 32-bit read-write register formatted as shown in Figure 6 When a trap or interrupt is taken, mcause is written with a code indicating the event that caused the trap.

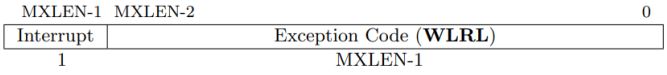

Fig. 6. Encoding of the mcause CSR register

The Interrupt bit in the mcause register is set if the trap was caused by an interrupt. The Exception Code field contains a code identifying the last exception or interrupt. Figure 7 lists the possible machine-level exception codes.

Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2	*Reserved*
1	3	Machine software interrupt
1	4	*Reserved*
1	5	Supervisor timer interrupt
1	6	*Reserved*
1	7	Machine timer interrupt
1	8	*Reserved*
1	9	Supervisor external interrupt
1	10	*Reserved*
1	11	Machine external interrupt
1	12–15	*Reserved*
1	≥16	*Designated for platform use*
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	Environment call from U-mode
0	9	Environment call from S-mode
0	10	*Reserved*
0	11	Environment call from M-mode
0	12	Instruction page fault
0	13	Load page fault
0	14	*Reserved*
0	15	Store/AMO page fault
0	16–23	*Reserved*
0	24–31	*Designated for custom use*
0	32–47	*Reserved*
0	48–63	*Designated for custom use*
0	≥64	*Reserved*

Fig. 7. Machine cause register (mcause) values after trap.

By reading the mcause register, the software trap handler can know what caused the trap and act accordingly.

5) Machine Exception Program Counter (mepc)[1, pp. 51]: The *mepc* register is used when the CPU begins handling a trap. Upon the occurrence of an interrupt or exception, the current program counter (PC) is saved into the *mepc* register. Specifically, in the case of an exception, the address of the instruction that caused the trap is written to *mepc*.

This mechanism allows the trap handler to determine how to proceed after handling the exception. Depending on the type of trap and the implementation, the trap handler can choose to:

- Re-execute the instruction that caused the trap.
- Skip to the next instruction in the program.
- Take alternative actions, such as reporting an error or halting execution.

6) Machine Scratch Register (mscratch)[1, pp. 51]: The *mscratch* register is a general-purpose, read/write CSR intended for use by machine-mode software. Unlike many other CSRs, *mscratch* does not have a predefined function within the RISC-V privileged architecture, allowing software developers to utilize it as temporary storage for context switching or other machine-mode operations. For example, during a trap or interrupt, machine-mode code can use *mscratch* to save and restore values from other registers without requiring additional memory accesses.

Since *mscratch* is not automatically used or modified by hardware, it is entirely under the control of machine-mode software, providing flexibility for various custom use cases. Its implementation ensures that critical operations, such as trap handling, can be performed efficiently and with minimal overhead.

C. Environment Calls

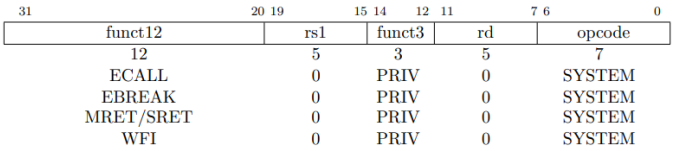

Fig. 8. The environment call instructions.

The instructions `ecall`, `ebreak`, and `mret` serve critical roles in managing control flow and privilege transitions. The `ecall` (Environment Call) instruction is used to trigger a trap to the operating system or firmware, allowing a lower privilege mode (e.g., user mode) to request services from a higher privilege mode (e.g., supervisor or machine mode). This mechanism is essential for implementing system calls. The `ebreak` (Environment Break) instruction is primarily used for debugging purposes, as it triggers a breakpoint trap, enabling a debugger to inspect the program state. Lastly, the `mret` (Machine-mode Return) instruction is used to return from a trap or interrupt handler in machine mode. It restores the processor state based on the values in the `mstatus`

and `mepc` registers, resuming execution at the interrupted program's address.

A further Wait For Interrupt (WFI) instruction can also be implemented for a halt to stall until an interrupt is received. This can be usefull for lower power consumption, especially in multicore systems.

D. Trap Handler

Whenever an interrupt occurs, hardware will automatically save and restore important registers. The following steps are complete as an interrupt handler is entered.

- Save pc to mepc
- Save mie to mstatus.mpie
- Set pc to interrupt handler address, based on mode of operation
- Disable interrupts by setting mstatus.mie=0

When interrupting a pipelined CPU, it is critical to ensure that the interrupt does not alter the control flow of instructions that have already been issued and executed. To prevent data or control hazards, the trap handler follows a strict process when handling an interrupt. First, it halts the issuance of new instructions and injects bubbles into the ID stage, allowing the pipeline to flush naturally. The handler waits until the last issued instruction passes the write-back (WB) stage before proceeding.

To maintain program consistency, the program counter (PC) saved in the `mepc` register must reflect the control flow of the executed instructions. This means that if a jump or branch is taken during the pipeline flush, the `mepc` must store the destination address of the jump or branch. This ensures that after the `mret` instruction, the processor can resume program execution seamlessly.

To implement this, a validity flag is added to each pipeline stage, ensuring that each stage is either valid or invalid. The program counter to be written to `mepc` is derived from the latest valid instruction in the pipeline. This mechanism avoids issues when an interrupt occurs in the middle of a branch instruction, especially if the processor is simultaneously waiting for memory operations.

For exceptions, modifications were made to the control stall module to ensure proper handling. If an exception arises in the decode stage, before taking the trap, the CPU needs to make sure that no earlier branches or jumps are taken. In that case, the instruction that causes the exception needs to be flushed. If not,t he instruction causing the exception is guaranteed to execute fully, and no premature jumps occur before it. Additionally, before signaling the trap handler to initiate the trap process, the CPU ensures that it is not stalled due to memory operations. This careful synchronization prevents issues and ensures that the trap process begins only under safe conditions.

At this point control is handed over to software where the interrupt processing begins. At the end of the interrupt handler, the mret instruction will do the following.

- Restore mepc to pc
- Restore mstatus.mpie to mie

E. External Interrupt Controller

An external interrupt controller needs to be implemented to handle both timer interrupts and I/O interrupts. This project follows the specification that were proposed on the SiFive Interrupt Cookbook[2].

In this project a Core Local Interruptor (CLINT) was developed to handle timer interrupts, external interrupts and software interrupts. This houses a 64 bit free rolling cpu timer (mtime), a 64 bit register (mtimecmp) that is compared to the mtime. If mtime is greater than mtimecmp, then an interrupt is routed to the trap handler of the cpu. It also houses the msip register that is used for software interrupts.

These registers are memory mapped on the location 0x02000000-0x02f00000 as per the SiFive specifications. The actual location for each memory access is:

- **msip**: 0x02000000
- **mtime**: 0x0200BFF8-0200BFFF
- **mtimecmp**: 0x02004000-0202004007

When the CPU is reset, the mtimecmp needs to be set to *0xFFFFFFFF_FFFFFFFF*. This way for the interrupt to occur it would take 5820 years for an 100Mhz cpu. Hence it's considered cleared. When a timer interrupt is handled by the software trap handler and it wants to clear it, the timecmp is set to that value as well.

```
    la   t0, trap_vector      # Address of our trap handler
    csrw mtvec, t0

    # 2. Enable global interrupts in mstatus.MIE
    li   t1, 0x8              # Bit 3 = MIE
    csrs mstatus, t1

    # 3. Enable software interrupts in mie (mie.MSIP = bit
        3)
    li   t1, 0x8
    csrs mie, t1

    # 4. Enable external interrupts in mie (mie.MEIE = bit
        11)
    li   t1, 0x800
    csrs mie, t1

    # 5. Enable machine timer interrupt in mie (mie.MTIMER
        = bit 7)
    li   t1, 0x80
    csrs mie, t1

    call main            # Call main function
```

Listing 1. Assembly for enabling timer interrupts

```
trap_vector:
    # Save context on the stack
    addi sp, sp, -144
    ... # Save caller-saved registers
    # Load trap information into function arguments
    csrr a0, mcause # Load mcause into a0 (first argument)
    csrr a1, mepc   # Load mepc into a1 (second argument)
    csrr a2, mtval  # Load mtval into a2 (third argument)
    # Call the C trap handler
    call c_trap_handler
    # Restore context from the stack
    ... #Load saved registers
    addi sp, sp, 144         # Restore stack pointer
    # Return from trap
    mret
```

Listing 2. Assembly for enabling timer interrupts

```c
void c_trap_handler(uintptr_t mcause, uintptr_t mepc,
     uintptr_t mtval) {
    if ((mcause & 0x80000000) != 0) {
        if(mcause == 0x80000007){
            // Timer interrupt
            set_timer_interrupt(ONETENTH);
            renderFrame();
        }
} else {
        // Exception
        switch (mcause) {
            case 2:  // Illegal instruction
            printfSCR(SCREEN_WIDTH * 14, 15, "Illegal
                instruction!\n");
            break;
            case 3:  // Breakpoint (ebreak)
            printfSCR(SCREEN_WIDTH * 14, 15, "Breakpoint (
                ebreak)!\n");
            while(getButtonDown()==0);
            break;
            case 8:  // Environment call from U-mode
            printfSCR(SCREEN_WIDTH * 14, 15, "System call
                from U-mode!\n");
            break;
            case 11:  // Environment call from S-mode
            printfSCR(SCREEN_WIDTH * 14, 15, "System call
                from M-mode!\n");
            while(getButtonDown()==0);
            break;
            default:
            printfSCR(SCREEN_WIDTH * 14, 15, "Unhandled
                exception!\n");
            break;
        }

        // Skip the faulting instruction
        if ((*(uint16_t *)mepc & 0x3) == 0x3) {
            // Regular 32-bit instruction
            mepc += 4;
        } else {
            // Compressed 16-bit instruction
            mepc += 2;
        }
        // Skip to the next instruction
        asm volatile("csrw mepc, %0" : : "r"(mepc));
        printfSCR(SCREEN_WIDTH * 11, 15, "new PC: %x!\n",
             mepc);
    }
}
```

Listing 3. Example of a trap handler that handles timer-based frame rendering

VI. Peripherals

All peripherals are memory mapped and are routed to the cpu via the bus. The primary function of the bus is to facilitate communication between the CPU and various peripherals by arbitrating memory loads and stores. The memory address space is divided into distinct memory-mapped regions. Each region is assigned to a specific device or function, such as RAM, ROM, I/O devices, or control registers, enabling efficient access and resource management. Memory-mapped I/O (MMIO) allows peripherals to be accessed using standard load and store instructions, simplifying the design and integration of hardware components. The bus ensures that memory access requests are properly routed to their designated regions, resolving potential conflicts through arbitration mechanisms. This structured approach not only optimizes performance but also enhances modularity and scalability in system design.

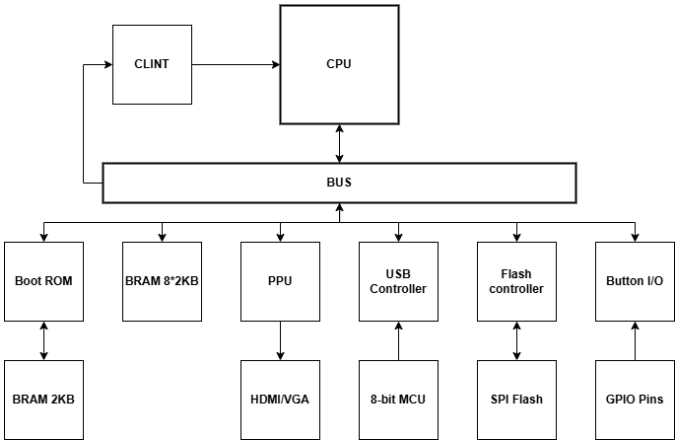

Fig. 9. The RISC-Y SoC diagram

The main peripherals developed for this project are as follows:

- **BRAM Memories**: The internal FPGA Block RAMs (BRAMs) are utilized to store both the boot code and application data, providing high-speed memory access for running programs and storing temporary data.
- **USB Keyboard**: The USB keyboard is managed through a dedicated USB controller, which handles all USB interrupts and provides a register that reflects the currently pressed key, allowing the user to read its value efficiently.
- **Flash Controller**: The flash controller interfaces with the FPGA's SPI Flash, enabling data storage and retrieval operations for non-volatile memory.
- **Pixel Processing Unit (PPU)**: The PPU is responsible for managing video data output. It supports rendering for both HDMI and 40-pin LCD displays, enabling both VGA Text mode and sprite/tile map rendering.

A. USB Controller

The USB controller is designed as a simple state machine that efficiently caches the keys currently pressed on a USB 1.0 keyboard. It incorporates an 8-bit microcontroller developed by *nand2mario*, which is available on GitHub[3]. This microcontroller is responsible for handling the low-level USB communication, significantly simplifying the overall system design. Instead of developing a dedicated software driver to manage USB protocol details, the 8-bit CPU offloads this workload from the main CPU. As a result, the USB controller provides a streamlined interface that exposes only the essential information required for basic input operations, such as the currently pressed keys. This approach reduces complexity, enhances modularity, and ensures efficient handling of USB interrupts and input events, allowing the main processor to focus on higher-level tasks without being burdened by low-level USB communication.

B. Flash Controller

The flash controller serves as a critical intermediary between the FPGA and the P25Q32SH SPI Flash chip, facilitating efficient read and write operations. It is designed to manage

the complexities of the Serial Peripheral Interface (SPI) protocol, enabling seamless communication with the external flash memory. The controller features a write-only flash address register, a write data buffer, a read data buffer, a read/write enable register and a command finished signal. These components work together to buffer data and simplify interactions with the SPI flash memory.

To execute a memory operation, the flash controller translates the requested command into the appropriate sequence of SPI instructions, including commands for page programming, sector erasing, and data reading. By offloading these protocol-specific tasks, the controller abstracts away the intricacies of SPI communication, allowing the main CPU to interact with the flash memory using simple, high-level commands. This design ensures efficient memory operations, reduces system latency, and supports robust data storage for applications requiring persistent memory. Additionally, the modular architecture of the flash controller makes it adaptable to other SPI-compatible flash devices with minimal modifications.

C. Pixel Processing Unit (PPU)

To provide meaningful output from the FPGA, a display-based communication method was chosen instead of the conventional UART text output. This decision enhances interactivity and usability, making the final result more engaging.

The FPGAs used for this project include the Tang Nano 9K/20K and the Tang Primer 25K from Sipeed. The Tang Nano boards feature a 40-pin LCD connector, while the Tang Primer 25K supports HDMI output. To achieve consistent rendering across both display types, a Pixel Processing Unit (PPU) was developed. The PPU processes screen coordinates (x, y) and generates an RGB color output accordingly.

The PPU architecture draws inspiration from the Game Boy's PPU[4] and the IBM VGA Text Mode[5]. This design choice allows for future integration with a Linux terminal, which traditionally relies on hardware text mode[6]. The PPU stores a tilemap containing either the first 128 ASCII characters or 128 custom-stored sprites. It utilizes two BRAMs (2×2KB) for the tilemap: one storing pointers to ASCII characters or sprites, and another—referred to as attribute memory—containing color information. The tilemap consists of 64 columns and 32 rows. Depending on the display resolution, some rows and columns may not be rendered but could be utilized for future enhancements, such as screen scrolling.

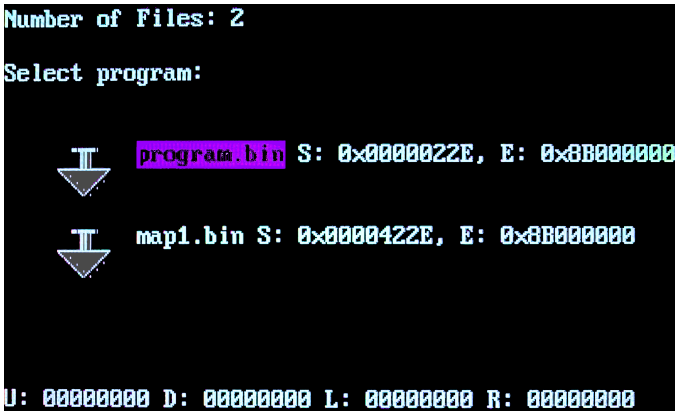

Fig. 10. Example of the screen displaying VGA Text Mode with custom sprites.

1) VGA Text Mode: The VGA Text Mode represents each character using 16 bits: the first 8 bits store the ASCII code, while the remaining 8 bits define the foreground and background colors.

Attribute								Character							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Blink[n 1]	Background color			Foreground color[n. 2][n. 3]				Code point							

Fig. 11. Bit layout of each character in Text Mode [7].

To render ASCII characters, an 8×16 font set containing 128 characters is stored in a ROM BRAM. The screen coordinates are divided using bit shifting to determine the current tile position:

```
currentCharacter = {ycursor[9:5], xcursor_next[9:4]};
```

Listing 4. Tile position calculation in Verilog

During each clock cycle, both BRAMs are accessed using the current tile's address. Since BRAM access introduces a one-cycle delay, this results in a slight pixel offset. A future improvement could involve adding a read buffer to prefetch the required data and align it with the display pipeline. However, for text mode, this delay does not significantly affect the output, so further optimization was deferred.

Once the appropriate ASCII character is retrieved from the tilemap, a pixel-by-pixel check determines whether a font pixel should be rendered in the foreground or background color. The color assignment follows a predefined palette:

```
if (charMem[{yPos[3:0], xPos[2:0]}] == 1'b1) begin
    case (dataOutAttr[3:0])
        4'b0000: begin r = 0; g = 0; b = 0; end  // Black
        4'b0001: begin r = 127; g = 0; b = 0; end // Red
        ...
    endcase
end
else begin
    case (dataOutAttr[6:4])
        3'b000: begin r = 0; g = 0; b = 0; end  // Black
        3'b001: begin r = 255; g = 0; b = 0; end // Red
        ...
    endcase
end
```

Listing 5. Pixel rendering logic based on font data

2) Sprite Tiles: Sprite rendering consists of two key components: *tiles* and *objects*. Tiles extend beyond the standard ASCII character set, utilizing the remaining 128 character slots to store additional graphical elements that can be mapped directly onto the screen. Objects, on the other hand, are independent graphical entities that can move freely across the screen without being constrained to predefined tile positions.

Custom sprites are stored in dedicated BRAMs, with a total capacity of 64 sprites distributed across four 2KB BRAMs (4×2KB). Similar to the data and attribute memories, these sprite memories are memory-mapped to the CPU, allowing them to be modified at any time. This is made possible by the *dual-port* nature of the BRAMs, which permits simultaneous read and write operations from different sources. Additionally, the BRAMs support differing clock domains, enabling communication between the screen refresh clock and the CPU clock.

To allow multiple colors per pixel, sprites are encoded using a 2-bit color format, as illustrated in Figure 12. Tile-mapped sprites are rendered using the same methodology as text-mode characters. However, a key difference arises due to memory access latency. Since retrieving a sprite requires first fetching the sprite pointer and then loading the corresponding sprite data, there is an inherent two-cycle delay. This results in a visible artifact when accessing different sprites in succession.

As discussed earlier, this issue can be mitigated by implementing a read buffer that preloads sprite data ahead of time, reducing latency-induced glitches.

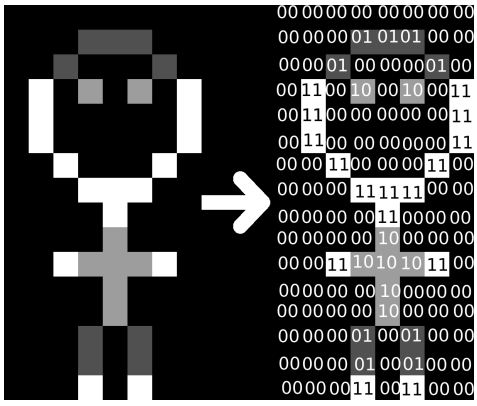

Fig. 12. Two bit encoding of an image with 4 total colors.

3) Object Sprites: Object sprites are independent graphical elements that can move freely across the screen without being constrained to a predefined tile grid. Each object is represented by a 32-bit memory register, which encodes information about the sprite's position, sprite ID, and enable status. This structure allows for potential future extensions, such as flipping the sprite horizontally or vertically and color inversion, enabling more advanced graphical effects.

Bit Range	Field	Description
31	`enable`	Enables or disables the sprite
24–16	`x`	X-coordinate of the sprite (9 bits)
15–9	`spriteID`	Sprite identifier (7 bits)
8–0	`yPos`	Y-coordinate of the sprite (9 bits)

4) Object Sprites: Object sprites are independent graphical elements that can move freely across the screen without being constrained to a predefined tile grid. Each object is represented by a 32-bit memory register, which encodes information about the sprite's position, sprite ID, and enable status. This structure allows for potential future extensions, such as flipping the sprite horizontally or vertically and color inversion, enabling more advanced graphical effects.

Bit Range	Field	Description
31	`enable`	Enables or disables the sprite
24–16	`x`	X-coordinate of the sprite (9 bits)
15–9	`spriteID`	Sprite identifier (7 bits)
8–0	`yPos`	Y-coordinate of the sprite (9 bits)

Unlike tile-mapped sprites, object sprites present additional challenges due to their dynamic nature. Since they require access to the same memory resources as tile-mapped sprites, rendering them directly on the fly is impossible. To address this, a row buffer is implemented, which is populated during each HBlank period with the sprite line data that will be rendered in the next scanline.

5) HBlank and Object Rendering: HBlank (Horizontal Blanking Interval) is the period between the completion of rendering a scanline and the start of the next scanline [8]. During this interval, no pixels are actively drawn to the display, allowing the system to perform background operations such as updating sprite data, buffering graphical elements, and synchronizing memory operations without causing visual artifacts.

The number of maximum objects that can be rendered per scanline is directly constrained by the available HBlank time. For the screens used in this system, approximately 20 objects per scanline can be processed within the HBlank period.

After the row buffer is filled, each sprite is rendered according to its X and Y position, offset by the current scanline. This ensures smooth movement and positioning of objects across the display.

6) Handling Clock Domain Crossings: One of the major challenges in integrating object sprites with different display interfaces, such as HDMI (110 MHz) and the CPU clock (50 MHz), is avoiding timing and setup violations caused by mismatched clock domains. Since write signals from the CPU could arrive at unpredictable times relative to the HDMI pixel clock, synchronization issues could occur, leading to data corruption due to timing violations.

To mitigate this issue, FIFO (First-In-First-Out) read/write buffers are used to safely transfer sprite attributes across clock domains [9, pp. 92-97]. These buffers ensure that sprite data is properly synchronized before being written to the sprite attribute registers, maintaining stability and preventing visual artifacts.

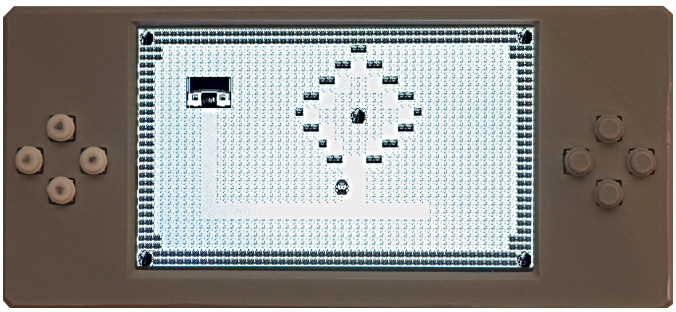

Fig. 13. A sprite based game running on the Tang Nano 20k.

VII. GCC Toolchain

A crucial part of the implementation is enabling software to be written in a high-level language instead of RISC-V assembly. This simplifies development, allowing for more complex programs to be written with greater ease. C was selected as the primary language for software development due to its simplicity and familiarity. Other languages, such as Rust, could also be used, as they support cross-compilation to RISC-V.

The GNU C Compiler (GCC) was chosen for its straightforward setup and extensive support for cross-compilation. While LLVM is another viable alternative—especially for integrating custom instructions—GCC was preferred for its robust toolchain and existing support for bare-metal RISC-V development.

A. Compiling for Bare-Metal RISC-V

By either installing a precompiled binary of riscv64-gcc or building it from source, any C file can be cross-compiled into RISC-V assembly. By default, GCC compiles programs as Linux executables, expecting them to be linked with an operating system to provide standard input and output functionalities. However, when running code in a bare-metal environment (without an OS), standard libraries such as stdio are unavailable unless manually implemented.

To support bare-metal execution, GCC allows custom linker scripts to define how code and data should be placed in memory. Instead of producing an OS-dependent binary, the toolchain can generate an Executable and Linkable Format (ELF) file targeting ‘riscv64-unknown-elf‘, which does not rely on an operating system [10].

B. ELF Files and Bootstrapping

Executing a .elf file requires a kernel-like software component to parse the ELF header and correctly map the program sections into memory. Since the system does not include an OS, an ELF file alone is insufficient for directly booting the CPU. Instead, the necessary binary instructions must be extracted and formatted for execution.

A custom stdio-like library was implemented to provide essential functions such as ‘printf‘, ‘memcpy‘, ‘putch‘, and basic arithmetic operations like modulo and division. Additionally, a custom linker script was developed to ensure that compiled programs are correctly mapped to the designated memory regions.

C. Memory Layout and Linker Script

The following linker script defines how code, data, and stack memory are allocated in RAM:

```
MEMORY
{
    RAM (wx) : ORIGIN = 0x00400000, LENGTH = 16K
}

/* Define stack size */
_stack_size = 0x800; /* 2KB stack size */

/* Place sections into memory */
SECTIONS
{
    . = ORIGIN(RAM);

    /* .text section (code) and .rodata (read-only data) */
    .text : {
        *(.text)           /* Include all .text sections */
        *(.rodata)         /* Include all .rodata sections
             */
        _etext = .;        /* Mark end of .text and .rodata
              section */
    } > RAM

    /* Align to 4 bytes to ensure proper alignment for the
         .data section */
    . = ALIGN(4);

    /* .data section (initialized data) */
    .data : {
        _sdata = .;        /* Start of .data section */
        *(.data)           /* Include all .data sections */
        _edata = .;        /* End of .data section */
    } > RAM AT> RAM

    /* Align to 4 bytes for .bss */
    . = ALIGN(4);

    /* .bss section (uninitialized data) */
    .bss : {
        _sbss = .;         /* Start of .bss section */
        *(.bss COMMON)     /* Include .bss and COMMON
             sections */
        _ebss = .;         /* End of .bss section */
    } > RAM

    /* Stack at the top of RAM */
    _stack_end = ORIGIN(RAM) + LENGTH(RAM);      /* Top of
         RAM */
    _stack_top = _stack_end;                     /* Start of
          stack */
    _stack_bottom = _stack_top - _stack_size;    /* Bottom
         of stack */
}

/* Assembly entry point label */
ENTRY(_start)
```

Listing 6. Linker script for RAM-based programs

This linker script ensures that:

- The program starts execution with a startup assembly file that initializes CSRs and the stack pointer, before jumping to ‘int main()‘.
- The .data section does not overlap with the .bss (uninitialized data) section or the stack.
- Proper alignment is maintained for correct memory access.

D. Extracting the Binary from ELF

To obtain an executable binary from the compiled .elf file, the objdump and objcopy tool from the GNU Binutils package

is used. ‘objcopy‘ extracts the assembly code and ,with the help of some bash scripts, formats it and pads it into a hex file and a .bin file, with the instructions stored at the beginning of the file. This ensures that the CPU can begin execution from the correct memory address.

```
riscv64-unknown-elf-objdump -O binary program.elf < program
    .hex
riscv64-unknown-elf-objdump -D program.elf > program.asm
```

Listing 7. Extracting assembly from an ELF file using objdump and objcopy

With a Makefile, the C program is compiled and converted into both a .hex file and a .bin file. As mentioned earlier, the CPU utilizes two types of program memory: Boot ROM and Program RAM.

The Boot ROM is embedded directly into the FPGA bitstream and initialized at configuration time. To achieve this, a hex file is required to pre-load the BRAMs with the necessary program data during FPGA synthesis.

For Program RAM, a binary file (.bin) is generated, which is later merged with other system files and flashed onto the FPGA's built-in flash memory. The next section is going to explain how the programs are dynamically loaded from the flash to the RAM to be run.

E. Flash Binary File System

A crucial aspect of utilizing programmable flash memory is the ability to load and manage files effectively. To achieve this, a simple file system is implemented, enabling file access based on their names. The structure of the file system is as follows:

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. Bootloader

With this file system in mind a basic bootloader was made that reads the filesystem's files and allows the user to select which application to load into the Program Memory RAM. The bootloader traverses the file system reading each file and displays to the user the available applications to run. The user can navigate the programs and select one by utilizing the buttons or USB arrow keys. After selecting an application, the bootloader copies the binary data from flash to the RAM. After that it creates a function pointer to the start location of the RAM and calls the start function.

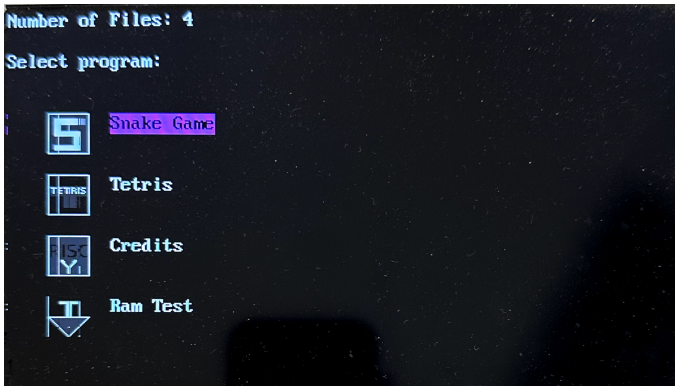

Fig. 14. The program select screen of the bootloader

VIII. Developed Tools

Several tools were developed to facilitate various aspects of software development and debugging.

A. Program Merger

The *Program Merger* is a Python-based application that provides a graphical user interface (GUI) for adding applications and files to generate the required binary file for the bootloader. This tool streamlines the process of organizing and preparing software components for execution.

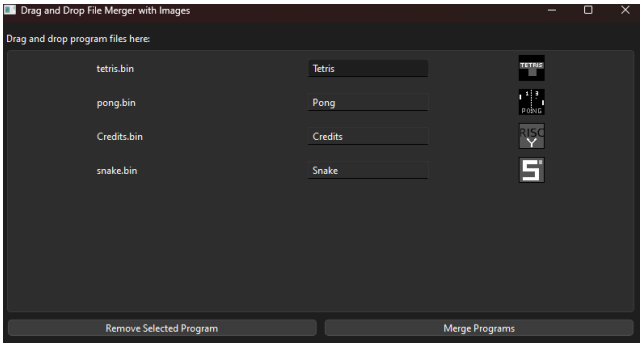

Fig. 15. The Program Merger with sample programs.

B. Emulator

An RV32I emulator was developed in JavaScript to verify the correctness of programs before flashing them onto the FPGA. Prior to integrating flash memory, this emulator significantly reduced development time by eliminating the need to generate and upload bitstreams for each test. Additionally, it served as a verification tool for the CPU implementation—matching outputs between the emulator and the CPU confirmed correct execution.

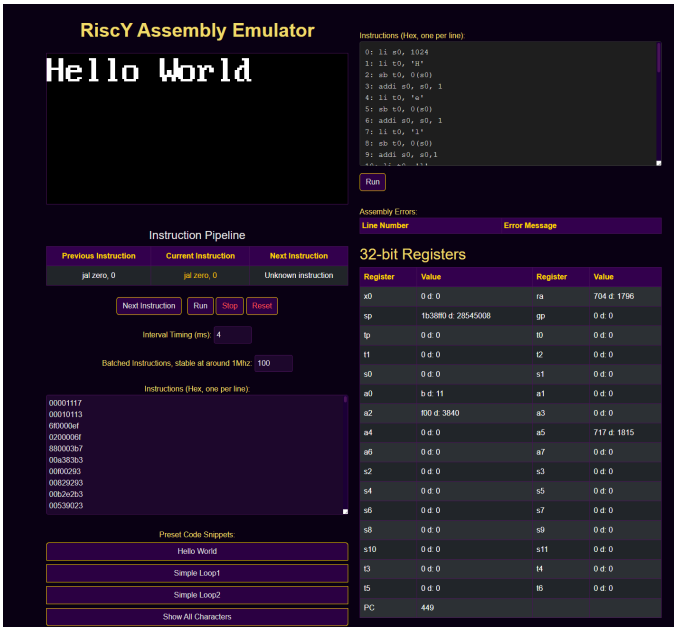

Fig. 16. The emulator running a "Hello World" program.

The emulator includes the following features:

- **VGA Text Mode Canvas:** Simulates text-based display output.
- **Hex File Input:** Loads compiled programs directly from hex files.
- **Register Table:** Displays real-time register values and updates.
- **Execution Speed Control:** Supports step-by-step execution and timed instruction execution.
- **Program Decompilation:** Provides assembly-level insight into compiled programs.

C. Map creator for fun.

For fun, a map creator was also created in Javascript to facilitate map creation for tile based games. It can be loaded as a file in flash and contains both the sprite data and the id for each sprite in a 52x14 map.

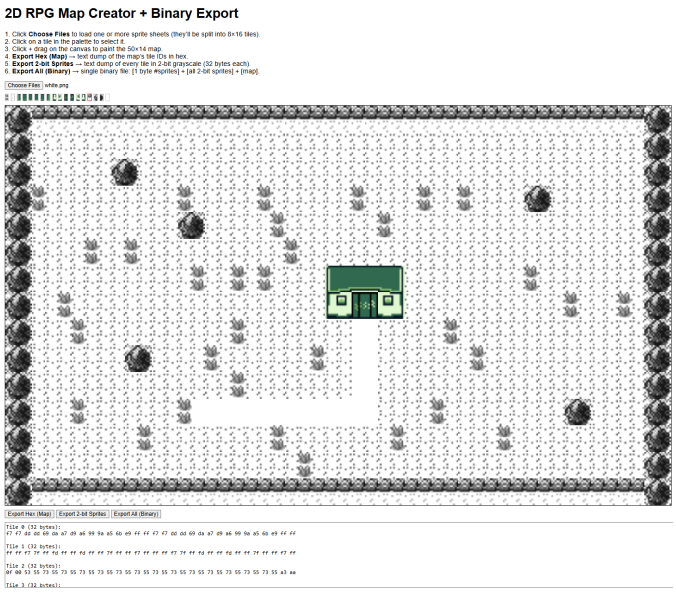

Fig. 17. A map based on the Pokemon Red Sprites.

IX. Running on the FPGA

The RISC-Y SoC was implemented on Sipeed's Tang Nano 9K, Tang Nano 20K, and Tang Primer 25K FPGA boards. These boards require the use of the Gowin IDE to generate the FPGA bitstream.

A. Gowin IDE

The Gowin IDE is a lightweight FPGA development environment that performs synthesis, placement, and routing. Compared to more feature-rich IDEs such as Vivado, it lacks advanced tools like behavioral/timing simulation and extensive reference designs. However, functional verification was conducted using Icarus Verilog to ensure correctness before synthesis.

Despite its limitations, Gowin IDE offers a notable advantage in terms of speed. Bitstream generation for the Tang Nano 20K and Tang Primer 25K is significantly faster than other FPGA tools, typically completing in under a minute.

B. Area and Timing Analysis

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. While the design technically fits within the 9K

LUTs available on the Tang Nano 9K, placement and routing often fail due to congestion. In contrast, the Tang Nano 20K and Tang Primer 25K handle the design efficiently, with bitstream generation taking less than a minute, compared to nearly 20 minutes on the Nano 9K.

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 9K	`25.9 MHz` (27 MHz oscillator)
Tang Nano 20K	`36.1 MHz` (27 MHz oscillator)
Tang Primer 25K	`40.3 MHz` (Stable at 50 MHz)

Among the tested FPGAs, the Tang Nano 20K demonstrated the highest stability, achieving an F_{max} that is 40% higher than its oscillator frequency. The Tang Nano 9K and Tang Primer 25K reported slightly lower F_{max} values than their respective crystal oscillators, but both operated reliably in practice.

C. Encryption controller

A fun project was creating an encrption core for the SoC. It demonstrated the performance gains of accelerating an algorithm from software to hardware with minimal area utilization. The encryption algorithm used was the Simon Cipher.

1) Simon and Speck: Simon and Speck are lightweight block ciphers [11] developed by the National Security Agency (NSA) to address the needs of resource-constrained environments, such as embedded systems and IoT devices. Both algorithms are designed to provide strong security with minimal computational overhead, making them ideal for hardware and software implementations.

Simon is a block cipher optimized for hardware implementations. It uses a simple round function that relies on bitwise operations, such as AND, XOR, and rotations, to achieve high efficiency and low resource usage. Simon supports a range of block sizes (e.g., 64, 96, and 128 bits) and corresponding key sizes, allowing for flexibility in its security and performance trade-offs. Its low gate count and minimal power consumption make it particularly suitable for FPGAs and ASICs.

Speck, on the other hand, is optimized for software implementations. It employs a Feistel network structure with addition modulo 2^n, XOR, and rotation operations. Like Simon, Speck supports various block and key sizes, making it adaptable to different applications. Speck is known for its speed and simplicity, making it well-suited for devices with limited processing power.

While both algorithms are highly efficient, Simon is often preferred for hardware-centric designs due to its lower complexity and resource requirements. Speck, with its optimization for software, finds applications in systems where hardware acceleration is not available.

Initial attempts to standardize Simon and Speck failed to achieve the super-majority required by the International Organization for Standardization (ISO). Expert delegates from several countries, including Germany, Japan, and Israel, raised concerns that the NSA's efforts to standardize these ciphers might stem from knowledge of exploitable weaknesses. These concerns were influenced by the NSA's previous promotion of the compromised Dual_EC_DRBG [**Schneier2007**] algorithm and partial evidence suggesting potential weaknesses in Simon and Speck. Despite this, the NSA cited over 70 security analysis papers from leading cryptographers supporting the algorithms' security and affirmed that no cryptanalytic techniques exist to exploit Simon or Speck.

Despite this, they remain strong candidates for lightweight encryption, especially in environments where traditional algorithms like AES are too resource-intensive. For this project, Simon was chosen due to its hardware-friendly characteristics and ability to meet the stringent resource constraints of an FPGA with 25k LUTs and a RISC-V core.

D. Performance Evaluation

To test the performance of the algorithms in a CPU environment, we utilized pre-existing, verified implementations to ensure correctness. These tests were conducted on two different platforms:

- **Custom RISC-V SoC:** Operating at 50 MHz on the Tan Primer 25k.
- **Raspberry Pi Zero 2W:** Featuring an ARM Cortex-A53 CPU clocked at 1 GHz, this platform provided a comparison point for the algorithms under a more powerful processing environment.

These tests provided insights into the relative efficiency, throughput, and resource requirements of each algorithm when implemented in software on different architectures. By combining these results with FPGA-specific metrics, a comprehensive performance profile was established for the Simon core.

TABLE III
PERFORMANCE COMPARISON OF ENCRYPTION AND DECRYPTION (IN CYCLES)

Platform	Encryption (cycles)	Decryption (cycles)
Raspberry Pi (AES -Os)	8489	13542
Raspberry Pi (AES -O3)	4635	8489
Raspberry Pi (SPECK -Os)	2187	1979
Raspberry Pi (SPECK -O3)	990	885
Raspberry Pi (Simon -Os)	2708	1354
Raspberry Pi (Simon -O3)	1563	990
RISC-V (AES -Os)	8188	11620
RISC-V (AES -O3)	2988	5726
RISC-V (SPECK -Os)	6344	8931
RISC-V (SPECK -O3)	1132	1278
RISC-V (Simon -Os)	5674	2341
RISC-V (Simon -O3)	1801	1393
Encryption Core	44	44-82

As shown in Figure 18, the encryption core demonstrates remarkable performance advantages over software implementations. Specifically, it achieves speeds approximately 40 times faster than software-based encryption on the custom RISC-V processor and 1.25 times faster than an ARM Cortex-A53 processor, despite the ARM core running at 20 times the clock frequency.

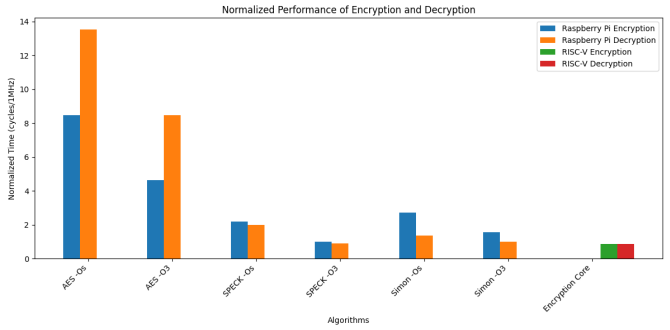

Fig. 18. Comparisons of times between ARM and Custom Core

These results highlight the substantial performance benefits of utilizing an Softcore CPU with the FPGA fabric to accelerate algorithms like encryption. It is also interesting to note that the core can run up to 150 Mhz stably, which would increase the speedup by 3 times!

Fig. 19. Encryption core running on the Tang Primer 25k with the RISCY SoC

X. Future Considerations

While the current implementation of the RISC-Y SoC successfully demonstrates a functional RISC-V-based system, several improvements and extensions can be explored in future iterations. These enhancements span across architectural optimizations, software support, and system integration.

A. Architectural Enhancements

- **Branch Prediction:** Implementing branch prediction mechanisms could minimize pipeline stalls, increasing overall CPU efficiency.
- **Floating-Point Unit (FPU):** Extending the core with floating-point instructions (RV32F/RV32D) would enable applications requiring numerical computations.
- **Multiplication and division:** Extending the core with multiplication and division instructions (RV32M) would enable applications requiring numerical computations.
- **Priviledge Modes Support:** The foundation has been layed out to implement Superuser (S-mode) and User (U-mode) modes. Some changes are going to be needed in order to delegate interrupts and trap M/S-mode instructions for lower level priviledges.
- **MMU and Virtual Memory Support:** Adding a Memory Management Unit (MMU) would allow support for privilege levels and operating system compatibility. With the help of the trap handler, virtualization should be easier to implement, trapping all memory accesses in user mode outside of the memory table.

B. Software and Toolchain Improvements

- **Operating System Support:** Porting a lightweight RTOS or custom OS like Zephyr or RTOS would enable multitasking and broader application support.
- **Expanded Compiler Support:** While the current implementation relies on GCC, integration with LLVM and Rust toolchains could enhance software portability.

C. Peripheral and FPGA Enhancements

- **More Efficient PPU Rendering:** Optimizing the Pixel Processing Unit (PPU) with better memory access or even VRAM for complete Frame buffered rendering.
- **DMA Controller:** Implementing a Direct Memory Access (DMA) controller would improve memory transfer efficiency, particularly for graphics and USB operations.
- **SDRAM Support:** Extending the system's memory with the built in SDRAM would allow running applications that require a lot more memory. This would sky rocket the software possibilities for the SoC.
- **Multi-Core Expansion:** Extending the system with multiple harts would allow parallel execution, improving computational throughput.

D. FPGA-Specific Optimizations

- **Alternative FPGA Implementations:** Exploring implementations on Xilinx's Zedboard and Virtex FPGAs to allow for bigger implementation with more memory and bigger caches.
- **Gowin Embedded Oscilator Debugging and Tracing:** Adding support for the Gowin Analyzer Osciloscope debugging would streamline architecture development and profiling.

References

[1] RISC-V Foundation. *The RISC-V Privileged Instruction Set Manual*. Version 20211203, for RISC-V Privileged Architecture Version 1.12. RISC-V Foundation. 2024. URL: https://www.scs.stanford.edu/~zyedidia/docs/riscv/riscv-privileged.pdf.

[2] Inc. SiFive. *SiFive Interrupt Cookbook V1.2*. 2019. URL: https://www.starfivetech.com/uploads/sifive-interrupt-cookbook-v1p2.pdf.

[3] nand2mario. *USB HID Host*. 2025. URL: https://github.com/nand2mario/usb_hid_host/tree/main.

[4] hacktix. *The Gameboy Emulator Development Guide*. 2021. URL: https://hacktix.github.io/GBEDG/ppu/.

[5] Charles Petzold. "Triple Standard: Three New Video Modes from IBM". In: *PC Magazine* (1987). URL: https://books.google.com/books?id=LRBokcwLB70C&pg=PA131.

[6] kernel.org. *The Framebuffer Console*. n.d. URL: https://www.kernel.org/doc/Documentation/fb/fbcon.txt.

[7] Wikipedia contributors. *VGA text mode*. n.d. URL: https://en.wikipedia.org/wiki/VGA_text_mode.

[8] Wikipedia contributors. *Vertical blanking interlval*. n.d. URL: https://en.wikipedia.org/wiki/Horizontal_blanking_interval.

[9] Steve Kilts. *Advanced FPGA Design: Architecture, Implementation, and Optimization*. Wiley-Interscience, 2007. ISBN: 978-0-470-05437-6.

[10] Wikipedia contributors. *Executable and Linkable Format*. n.d. URL: https://en.wikipedia.org/wiki/Executable_and_Linkable_Format.

[11] Ray Beaulieu et al. *The Simon and Speck Families of Lightweight Block Ciphers*. 2016. URL: https://eprint.iacr.org/2013/404.