FACULTY OF APPLIED SCIENCES
UNIVERSITY
OF WEST BOHEMIA

DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Master's Thesis

ARMv6 Processor Emulator for Raspberry Pi Environment Emulation

Jakub Šilhavý

PILSEN, CZECH REPUBLIC

2023

FACULTY OF APPLIED SCIENCES
UNIVERSITY
OF WEST BOHEMIA

DEPARTMENT OF
COMPUTER SCIENCE
AND ENGINEERING

Master's Thesis

ARMv6 Processor Emulator for Raspberry Pi Environment Emulation

Bc. Jakub Šilhavý

Thesis advisor
Ing. Martin Úbl

PILSEN, CZECH REPUBLIC 2023

Citation in the bibliography/reference list:
ŠILHAVÝ, Jakub. *ARMv6 Processor Emulator for Raspberry Pi Environment Emulation*. Pilsen, Czech Republic, 2023. Master's Thesis. University of West Bohemia, Faculty of Applied Sciences, Department of Computer Science and Engineering. Thesis advisor Ing. Martin Úbl.

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Akademický rok: 2023/2024

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Jakub ŠILHAVÝ**
Osobní číslo:	**A21N0072P**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Softwarové inženýrství**
Téma práce:	**Emulátor ARMv6 procesoru pro emulaci prostředí Raspberry Pi**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování

1. Learn about the ARMv6 architecture, the BCM2835 microcontroller and the Raspberry Pi Zero platform.
2. Analyze the software resources available to emulate this environment.
3. Design an emulator that allows emulation of a subset of the instruction set of the ARM1176JZF--S processor and basic peripheral support for the BCM2835 microcontroller.
4. Implement this emulator in C++ using modern standards.
5. Test the implemented solution on a set of basic tasks and a selected minimalist operating system that supports this platform, evaluate the solution.

Rozsah diplomové práce:	**doporuč. 50 s. původního textu**
Rozsah grafických prací:	**dle potřeby**
Forma zpracování diplomové práce:	**tištěná/elektronická**
Jazyk zpracování:	**Angličtina**

Seznam doporučené literatury:

dodá vedoucí diplomové práce

Vedoucí diplomové práce:	**Ing. Martin Úbl** Katedra informatiky a výpočetní techniky
Datum zadání diplomové práce:	**8. září 2023**
Termín odevzdání diplomové práce:	**16. května 2024**

Západočeská univerzita v Plzni
Fakulta aplikovaných věd
L.S.

Doc. Ing. Miloš Železný, Ph.D.
děkan

Doc. Ing. Přemysl Brada, MSc., Ph.D.
vedoucí katedry

V Plzni dne 11. října 2023

Declaration

I hereby declare that this Master's Thesis is completely my own work and that I used only the cited sources, literature, and other resources. This thesis has not been used to obtain another or the same academic degree.

I acknowledge that my thesis is subject to the rights and obligations arising from Act No. 121/2000 Coll., the Copyright Act as amended, in particular the fact that the University of West Bohemia has the right to conclude a licence agreement for the use of this thesis as a school work pursuant to Section 60(1) of the Copyright Act.

V Plzni, on 10 September 2023

. . . . . .

Jakub Šilhavý

Abstract

This paper examines the potential of emulating a Raspberry Pi Zero, a selected example of one of the most widely adopted architectures for embedded systems — the ARM architecture. The initial chapters delve into a general introduction to the ARM architecture, highlighting its profound significance evidenced by billions of electronic that leverage it. Transitioning to the second part, the thesis addresses the benefits of utilizing an ARM emulator, delineating overall requirements, and reviewing existing methodologies.

The second part centers on the development of a custom Raspberry Pi Zero emulator whose functionality is systematically tested using a set of examples pertinent to operating system development. The thesis concludes with an objective evaluation of the emulator's performance, identifying its key benefits and suggesting areas for further enhancements.

Abstrakt

Diplomová práce zkoumá potenciál emulace Raspberry Pi Zero, který reprezentuje jednu z nejvíce rozšířených architektur pro vestavěné systémy — architekturu ARM. Úvodní kapitoly se věnují obecnému seznámení se s ARM architekturou, s důrazem na její význam, který je demonstrován více než miliardou elektronických zařízení, které ji využívají. Dále se práce zaměřuje na výhody využívání ARM emulátoru, stanovení celkových požadavků a analýzu existujicích možnosti řešení.

Druhá část textu se soustředí na vývoj samotného Raspberry Pi Zero emulátoru, jehož funkčnost je systematicky testována pomocí sady příkladů vztahujících se k vývoji operačních systémů. Práce je zakončena objektivním hodnocením výkonu emulátoru, identifikací jeho klíčových výhod a navrhováním oblastí pro další vylepšení.

Keywords

ARM • Processor • Emulator • Raspberry Pi

Contents

Introduction 1

Computer Architectures

2

A computer architecture can be generally considered a low-level principal design of a computer system that defines the interaction of its components that work alongside to accomplish a given task. While there are a number of different computer architectures, some of the most well-known architectures may include Intel, MIPS, ARM, or the increasingly popular RISC-V architecture.

The internal workings of a computer architecture can be explained through the concept known as instruction set architecture, often abbreviated as `ISA`, which provides a comprehensive understanding of computer's operation capabilities. It also provides insights into how the user can interact with the central processing unit (CPU) based on the specific types of instructions it supports.

2.1 Classification of Computer Architectures

Computer architectures can be categorized by various aspects, such as how they handle data, the addressing modes they provide, how they organize memory, what register set they feature [1], or the number of operands in an instruction. All these criteria must be taken into consideration when selecting the appropriate architecture for a given application, as each one comes with its own set of advantages and disadvantages. The following sections delve into different types of computer architectures, explaining their fundamental principles along with their pros and cons.

[1] Although registers are the fastest type of memory, from an operating system perspective, featuring an extensive number of registers can also pose a disadvantage, as it increases the time required to perform a context switch, which might be critical for real-time applications.

2.1.1 Von Neumann Architecture vs Harvard Architecture

As far as memory access is concerned, there are two major architectural ideas that serve as the foundation for other computer architectures: the Von Neumann architecture and the Harvard architecture. While they are distinguished by various factors, including their cost, intended usage environment, and more, their primary differentiation lies in how they organize and access memory.

2.1.1.1 Von Neumann Architecture

The Von Neumann architecture, designed by the mathematician and physicist John von Neumann in 1945, is what can be found in modern personal computers. As illustrated in Figure 2.1, it features a **single memory space for both code and data**, necessitating only a single address and data bus, which could pose the risk of unintentionally overwriting the program's instructions, potentially leading to a CPU crash. On the other hand, its design is considerably cheaper compared to the Harvard architecture, as it requires less physical space.

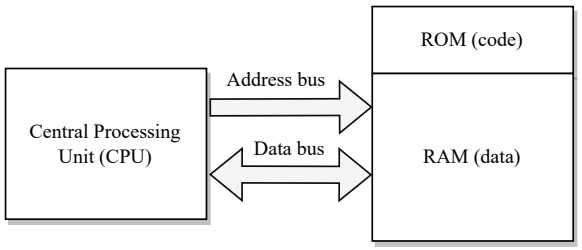

Figure 2.1: Von Neumann architecture

Another potential drawback arises from the fact that data and instructions share the same bus, even though fetching instructions occurs more frequently than data transfers. Consequently, the shared data bus may represent a performance bottleneck within the entire system.

2.1.1.2 Harvard Architecture

On the other hand, the Harvard architecture employs **distinct memory modules for code and data**, requiring twice the number of bus lines compared to the Von Neumann architecture. This design is typically preferred in scenarios where the performance benefits outweigh the additional costs, such as in microcontrollers, and field programmable gate arrays, also known as FPGA boards.

Taking advantage of two separate memory modules, it can effectively mitigate resource conflicts and enhance its performance through parallelism and separate memory caching.

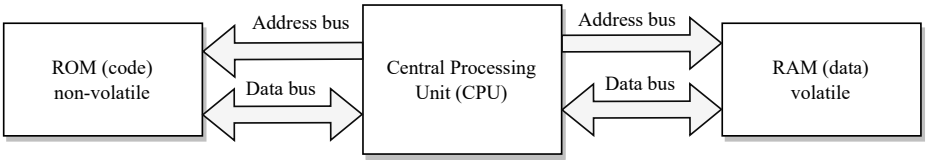

Figure 2.2: Harvard architecture

2.1.2 Instruction Set Architectures

Instruction set architectures can also by categorized by how they handle operands in terms of interacting with memory. As noted previously, each approach may be beneficial for different types of applications.

2.1.2.1 Stack Architecture

The stack architecture, shown in figure 2.3, does not rely on memory to retrieve operands. Instead, the CPU maintains an internal stack, which serves as storage for operands. An operation is executed in a *last-in-first-out* (LIFO) fashion, where two operands are popped off the stack, and the operation's result is consequently pushed back onto the stack.

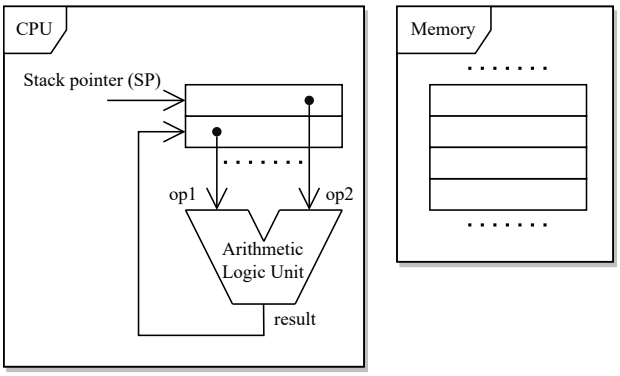

Figure 2.3: Stack architecture[2]

A distinct advantage of this architecture is that it makes compiler implementation considerably easier compared to other types of architectures. However, the main drawback is the stack, which introduces a bottleneck that hinders any potential parallelization.

2.1.2.2 Accumulator Architecture

Similar to the stack architecture, the accumulator architecture imposes minimal hardware requirements. It substitutes the stack with a single register called the ac-

[2]This architectural design is employed, for example, by the Java Virtual Machine.

cumulator, which serves both as an input operand and as storage for the operation's result. Same as the previous architecture, the accumulator represents a bottleneck. Furthermore, it leads to high memory traffic, as every two-operand instruction requires retrieving the second operand from memory.

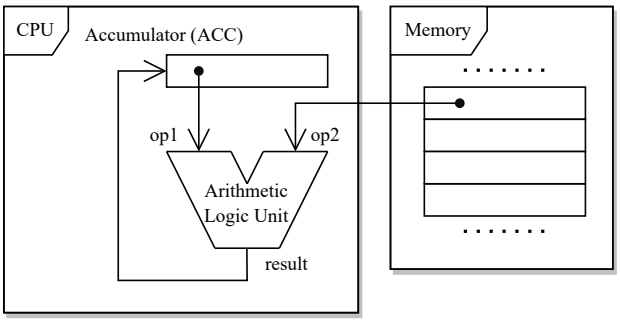

Figure 2.4: Accumulator architecture

2.1.2.3 Load-Store Architecture

The load-store architecture does not utilize memory during arithmetic-logic operations. Instead, **it restricts memory access to a specific pair of instructions: the load and store instructions**. These instructions serve as an interface for reading and writing data to memory. Consequently, when an operation needs to be executed, the CPU must ensure that all operands are stored in individually addressable memory banks, known as CPU registers, before employing the arithmetic-logic unit to proceed with the operation.

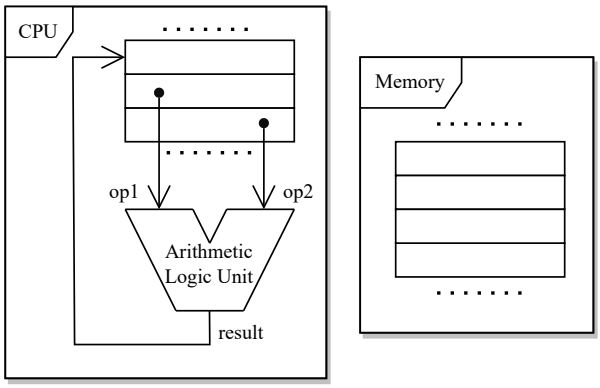

Figure 2.5: Load-store architecture

Figure 2.6 illustrates the process of retrieving operands from memory, carrying out the necessary operation, and then storing the resulting data back in the main memory.

Load-store architectures frequently use fixed-length instructions, providing the potential for more efficient pipelining [3], which can result in improved performance. However, a notable drawback of this architecture is its strong reliance on a sophisticated compiler, which may not be as straightforward to implement as in the architectures mentioned previously.

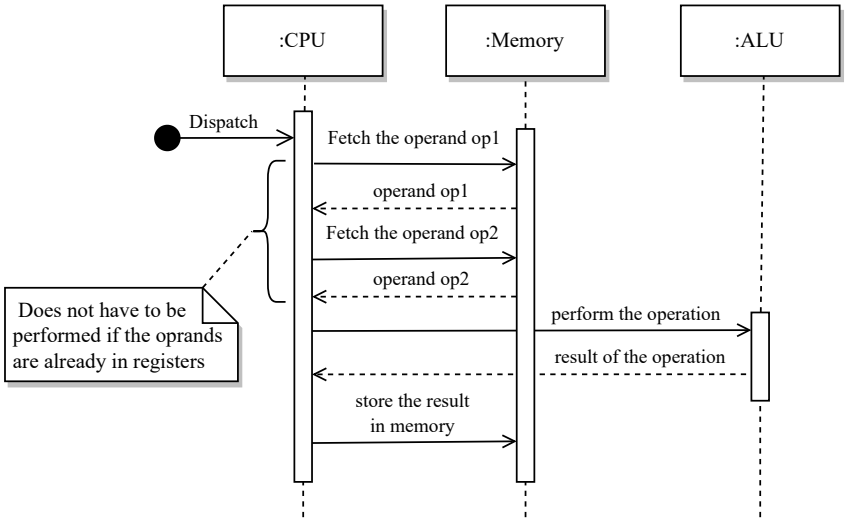

Figure 2.6: Load-store sequence diagram

2.1.3 Reduced vs Complex Instruction Set Computer

`RISC` (Reduced Instruction Set Computer) and `CISC` (Complex Instruction Set Computer) are two prominent `CPU` architectures that have gained widespread adoption over time. They employ distinct approaches in utilizing their respective instruction sets to align with the hardware implementation [1].

2.1.3.1 Complex Instruction Set Computer

The `CISC` architecture places an **emphasis on the underlying hardware**, leading to the development of complex multi-clock instructions that can vary in length. This length variation may introduce complications in pipelining, which can have an adverse effect on performance. However, implementing custom instructions tailored to specific hardware specification result in smaller code sizes, as they take a more concrete and less abstract approach to achieve the desired functionality. An example of this type of architecture is the `x86-64` Intel processor.

[3] Pipelining is a low-level parallelization technique that involves breaking down the processing of an instruction into multiple stages that can be executed simultaneously.

2.1.3.2 Reduced Instruction Set Computer

RISC, on the other hand, **focuses on the software aspect**, resulting in fewer single-clock instructions of a fixed-size, which facilitate easier pipeline processing. Another significant aspect is that RISC does not perform operations directly on memory. Instead, it adheres to the principle illustrated previously in figure 2.6. Another important consideration in this architecture is that having more general and, perhaps, simpler instructions can potentially increase the size of the final binary. This is because achieving a certain functionality may require the use of multiple instructions, unlike the CISC architecture, which typically accomplishes the same goal with a single instruction. A typical example of this architecture is the ARM architecture, which is described more in detail in the following chapter.

There are other types of computer architectures that can be categorized by various aspects. The objective of this chapter was to provide the reader with introductory insights into some of the key factors that define a computer architecture. For more comprehensive information, readers can refer to the book series *Computer Organization and Design: The Hardware/Software Interface* [2].

ARM Architecture 3

3.1 History

ARM was officially founded as a company in November 1990 under the name Advanced RISC Machines Ltd. It originated as a joint venture involving companies such as Arcon Computers, Apple Computer, and VLSI Technology. Its early indications of future potential became evident with the Nokia 6110 GSM mobile, which experienced a remarkable surge in popularity after its release in 1998. Currently, it is estimated that over 99% of the world's smartphones are built on ARM technology.

Throughout the 2000s, ARM's sustained success enabled it to evolve beyond smartphones and become arguably the most widely used processor architecture. Nowadays, ARM technology can be found across a broad spectrum of embedded devices, ranging from sensors and low-power microcontrollers to supercomputers and real-time mission-critical systems [3].

Figure 3.1: Devices leveraging ARM technology

An interesting aspect of ARM is that it does not engage in silicon manufacturing. Instead, it preserves the architecture as intellectual property, outsourcing the implementation to its closely aligned silicon partners, who are part of the so-called connected community. This ARM surrounding community forms a global network

of companies that collaborate by sharing expertise, providing support services, offering design consulting, and supplying tools for creating ARM-powered solutions.

3.2 Processor Cores

ARM classifies its processors into three major groups, which makes it a viable choice for a wide range of applications. Figure 3.2 displays these categories in ascending order based on their capabilities.

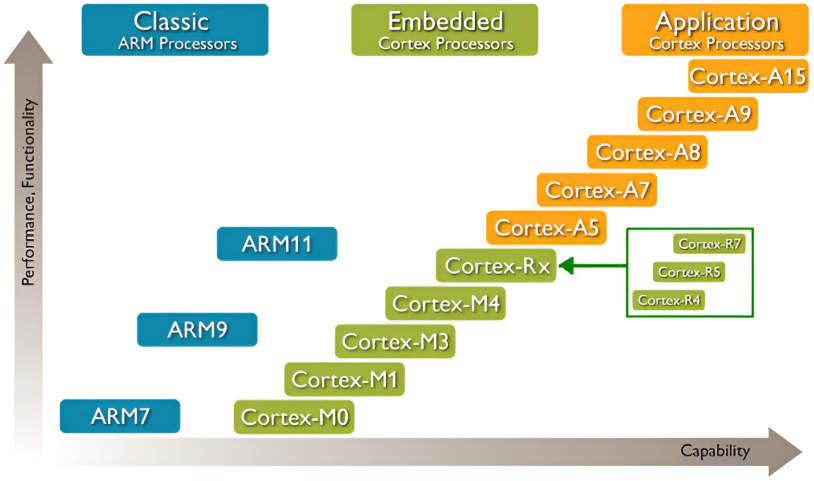

Figure 3.2: ARM processor roadmap

3.2.1 Classic ARM Processors

The Classic ARM Processors represent the company's initial line of processors. A typical example within this category is the `ARM7TDMI-S` CPU core, which was widely embraced by the cellphone industry. Over time, ARM has introduced additional Cortex families, each tailored for their intended domain of applications.

3.2.2 Embedded Cortex Processors

Generally speaking, the Cortex-M family is utilized in low-cost microcontrollers, which can be commonly found embedded in Internet of Things (`IoT`) devices such as home automation systems, wearables[1], or smart locks.

Usually, the ARM Cortex-M family simplifies or modifies certain features, resulting in slight deviations from the traditional ARM architecture. These

[1]Wearable technology refers to any type of smart devices designed to be worn, such as smartwatches, smart glasses, etc.

modifications may involve CPU modes, the exception model, or bank registers, all of which are further discussed in the following sections.

Another line of processors targeted for the embedded world is the Cortex-R family, known for delivering high performance and throughput while upholding precise timing properties and minimizing interrupt latency. This characteristic makes it a suitable core for domains with time-constrained requirements, including automotive systems, medical devices, and aerospace and defense systems.

3.2.3 Application Cortex Processors

The Cortex-A line of processors is designed for applications that require a general-purpose platform operating system, which is commonly used in laptops and personal computers. As a result, they integrate an extended instruction set to improve multimedia processing, along with an advanced memory management system that ensures a seamless human-machine interaction experience.

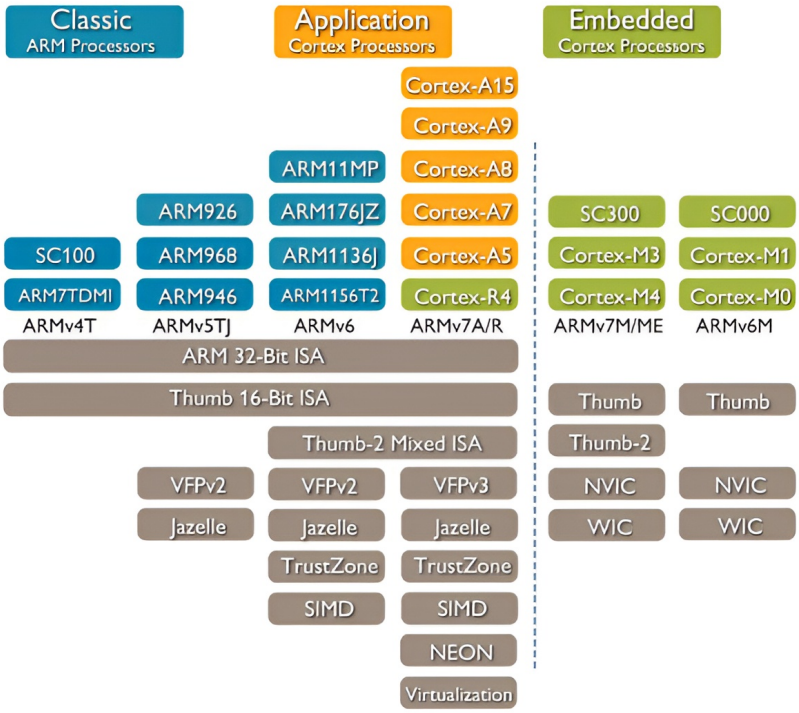

Figure 3.3: Features of different ARM processor cores

Figure 3.3 illustrates the ongoing evolution of capability features across various CPU cores.

3.2.3.1 ARMv6

For instance, it can be observed that ARMv6 incorporates the *Thumb* instruction set version 2, which is runtime interchangeable with the *ARM* instruction set. Additionally, it includes support for floating-point operations version 2, Jazelle for execution of `Java bytecode`, TrustZone for secure data storage, and Single Instruction/Multiple Data instructions, often referred to as `SIMD` instructions.

3.3 Instruction Set Architecture

ARM is a load-store `RISC` architecture that supports two major sets of fixed-size instructions: the standard 32-bit ARM instruction set and the reduced 16-bit *Thumb* instruction set. The following sections delve into some of the key characteristics of the Classic line of ARM processors. For a more comprehensive understanding, the reader is encouraged to refer to the official ARM Architecture Reference Manual [4].

3.3.1 Central Processing Unit Modes

There are a total of seven modes in which an ARM `CPU` can operate, each represented by a unique 5-bit number stored in the *Current Program Status Register*. The `CPU` can switch to one of these modes either implicitly, such as when an interrupt occurs, or explicitly by the programmer, for example, when switching the current `CPU` context. All modes except for the *User* mode are privileged, meaning that when the `CPU` is in the *User* mode, the execution of certain instructions might be restricted [2].

Table 3.1: List of ARM `CPU` modes

CPU mode	Description
User	Normal program execution
FIQ	Supports a high-speed data transfer or channel process
IRQ	Used for general-purpose interrupt handling
Supervisor	A protected mode for the operating system
Abort	Implements virtual memory and/or memory protection
Undefined	Supports software emulation of hardware coprocessors
System	Runs privileged operating system tasks

[2] In the context of operating systems, a non-privileged mode is typically used for the execution of user programs, while the kernel operates in a privileged mode.

It is worth noting that Cortex-M utilizes only two CPU modes: *Thread* mode, an unprivileged mode designed for executing application code, and *Handler* mode, a privileged mode intended for handling exceptions.

3.3.2 Exception Model

When an exception or interrupt occurs, the CPU sets the program counter register to the address associated with that exception, known as the interrupt vector, and switches to the corresponding CPU mode, which can be found in Table 3.2. The interrupt vector represents a fixed address in RAM where the CPU will continue its execution. Therefore, during the system initialization, these memory locations are typically filled with branch instructions to direct the execution to the corresponding exception handlers.

As illustrated in figure 3.4, this address region, also known as the interrupt vector table, can be found located either in the lower part or the upper part of the virtual address space, depending on the current setting stored in the *System Control Coprocessor*.

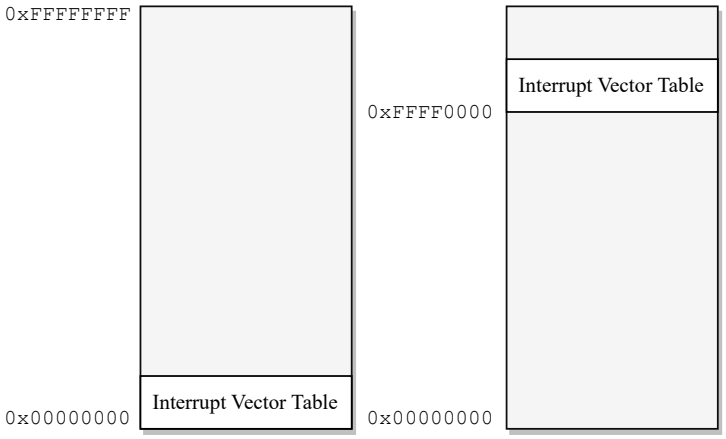

Figure 3.4: Possible locations of the interrupt vector table in RAM

Table 3.2: List of ARM CPU exceptions

Exception	CPU mode	Normal address	High address
Reset	*Supervisor*	0x00000000	0xFFFF0000
Undefined instruction	*Undefined*	0x00000004	0xFFFF0004
Software interrupt	*Supervisor*	0x00000008	0xFFFF0008
Prefetch abort	*Abort*	0x0000000C	0xFFFF000C

(table continues on the next page)

Table 3.2 *(continued from the previous page)*

Exception	CPU mode	Normal address	High address
Data abort	*Abort*	`0x00000010`	`0xFFFF0010`
Interrupt	*IRQ*	`0x00000018`	`0xFFFF0018`
Fast interrupt	*FIQ*	`0x0000001C`	`0xFFFF001C`

Similar to the previously mentioned CPU modes, the Cortex-M family employs a slightly different exception model.

3.3.3 Registers

ARM offers 16 general-purpose 32-bit registers labeled `r0` through `r15`, three of which serve special functions, which are listed in Table 3.3. The programmer is free to use the remaining registers as needed.

However, specific calling conventions were established to ensure a systematic use of these registers. For instance, registers `r0-r3` are used as argument values passed into a subroutine, while return values are typically stored in registers `r0-r1`. Further details on calling conventions can be found in Chapter 6 of the ARM's Procedure Call Standard Manual [5].

Table 3.3: List of special function ARM registers

Index	Name	Description
13	SP	*Stack pointer* - holds the address of the top of the stack
14	LR	*Link register* - holds the return address (when calling a function, it is not pushed onto the stack)
15	PC	*Program counter* - holds the address of the instruction to be executed

3.3.3.1 Banked Registers

A distinctive feature of ARM is its utilization of so-called *bank* registers. In this concept, each CPU mode has its own unique set of registers that are automatically loaded whenever the current CPU mode changes.

The more bank registers are utilized, the faster the switch into the corresponding CPU mode is, as there is no need to preserve the current state by storing all registers

onto the stack. As a result, this concept is extensively employed by the *FIQ* mode for ensuring fast interrupt handling, hence the mode's name.

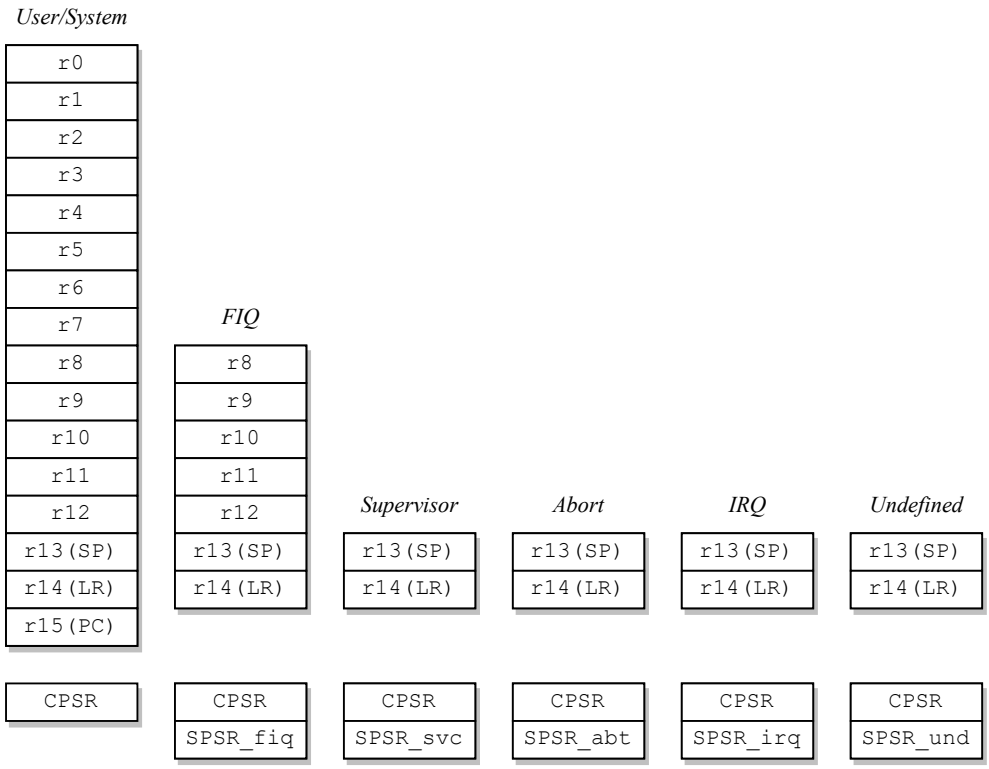

Figure 3.5: Bank registers of different CPU modes

When the CPU is executing 16-bit *Thumb* instructions, only the first 8 registers (`r0-r7`) can be addressed directly. These registers are sometimes referred to as the *Thumb State Low Registers*.

As far as Cortex-M is concerned, since there are only two CPU modes, the banking scheme applies only to the SP register.

3.3.3.2 Control Registers

Current Program Status Register.

The *Current Program Status Register* (CPSR) preserves the current state of the CPU along with some additional information.

To modify the control register, its content must first be transferred into one of the general-purpose registers using the `MRS` instruction, as direct modification is not allowed. Subsequently, after the necessary modifications have been made, it can be transferred back from the general-purpose register using the `MSR` instruction.

This 32-bit register is segmented into four sections, as illustrated in figure 3.6. The initial letters of the section names can function as a bit mask when using the

MRS or MSR instruction to prevent unintentional modifications of bits that are not intended to be changed.

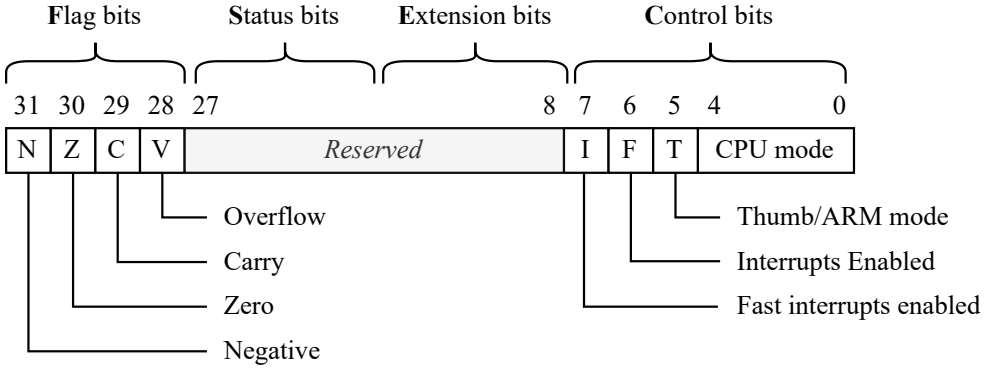

Figure 3.6: Current Program Status Register[3]

Saved Program Status Register.

The *Saved Program Status Register*, or SPSR, serves as a copy of the CPSR register utilized during CPU mode switching. When transitioning to a mode that contains an SPSR register, which is shown in figure 3.5, the contents of the CPSR register of the current mode are copied into the SPSR of the new mode. This process enables the restoration of the original state when reverting to the original CPU mode.

3.3.4 ARM Instructions

As mentioned previously, all ARM instructions are 32-bits in length. The most significant four bits of each instruction form a condition filed that constrains its execution. The CPU assesses the current state of flag bits in the CPSR register, and depending on the condition field, the instruction is either skipped or executed. From the programmer's standpoint, this can be accomplished by suffixing the instruction name by the desired condition code, as listed in Table 3.4.

In contrast to other architectures, the flag bits are not automatically set upon the execution of an instruction. Instead, it is up to the programmer to choose whether to update them by adding an `'S'` to the end of the instruction name. For instance, `ADD` does not update the flags, whereas `ADDS` does [4].

[3]The reserved area contains additional information and control bits including the result of `SIMD` instructions, the status of the Jazelle bit, and endianness, which can also be changed at runtime.

[4]This concept does not apply to instructions like `CMP`, which, as its sole purpose suggests, implicitly sets the flags.

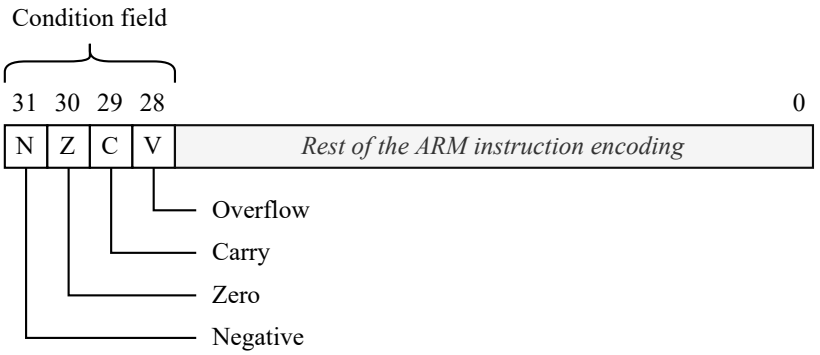

Figure 3.7: Condition field of an ARM instruction

Table 3.4: List of ARM instruction condition codes

Code	Flags tested	Meaning
EQ	Z==1	Equal
NE	Z==0	Not equal
CS or HS	C==1	Unsigned higher or same (or carry set)
CC or LO	C==0	Unsigned lower (or carry clear)
MI	N==1	Negative (mnemonic - „minus“)
PL	N==0	Positive or zero (mnemonic - „plus“)
VS	V==1	Signed overflow (mnemonic - „V set“)
VC	V==0	No signed overflow (mnemonic - „V clear“)
HI	(C==1) && (Z==0)	Unsigned higher
LS	(C==0) || (Z==1)	Unsigned lower or same
GE	N==V	Signed greater than or equal
LT	N!=V	Signed less than
GT	(Z==0) && (N==V)	Signed greater than
LE	(Z==1) || (N!=V)	Signed less than or equal
AL	Not tested	Always executed (suffix is omitted)

In general, ARM instructions can be classified into several different categories based on their purposes, such as data processing instructions, data transfer instructions, branch instructions, or coprocessor instructions, which are designed for interacting with external CPU coprocessors. Detailed ARMv6 instruction encodings can be found, for instance, in the B2 ARM Appendix document [18].

3.4 Coprocessors

3.4.1 System Control Coprocessor

3.4.2 Single-precision Floating-point Coprocessor

Design of a Raspberry Pi Zero Emulator — 4

When designing a complex software system, it is important to take into consideration deciding factors such as the intended usage environment, interaction methods, system dependencies, preferred technologies, or different components constructing the final application. Addressing these questions early on enhances the likelihood of its successful completion as well as its long-term maintainability.

This chapter outlines the key design decisions made within the implementation process of the **ZeroMate emulator**, which is the chosen name for the project [1].

4.1 Input

The emulator necessitates a single input file in the `ELF` format, simplifying the booting process. In this process, the *stage 1 bootloader*, residing in `ROM`, reads the contents of the `SD` card and then initiates and delegates control to the `GPU`, which subsequently resets the `CPU` and loads the kernel into `RAM`.

This file is further referred to as the **kernel** since the emulator was designed within the context of operating systems development. Nevertheless, the input file can fundamentally represent any application intended for execution on Raspberry Pi Zero. Figure 4.2 illustrates the general process of building an `ELF` file, which contains all essential data and information required for code emulation.

4.1.1 Executable and Linkage Format

`ELF` stands for *Executable and Linkage Format* [6], and it is one of the most commonly used formats for executable files, especially on Unix-like systems. There are a number of other representations used in embedded development. For instance, the *Motorola S-Record format*, or `SREC` for short, is often used for programming non-volatile types of memory, such as flash or `EEPROM`.

[1] It combines the word *Zero*, as in Raspberry Pi Zero, and *Mate*, which in this case is used as a synonym for a friend or „buddy“.

In terms of this project, the key advantage of ELF over SREC is that ELF is **used for both linkage and execution**. Therefore, if the kernel is compiled with debug symbols turned on [2], the symbol table stored in the final ELF file can be used during the parsing process, which is discussed in section 4.3.2, to provide the user with function names as they were used in the source code, which should improve the readability of the final disassembly. SREC, on the other hand, is **only used for execution**. Therefore, it can be viewed as highly compressed as it comprises only the necessary information for uploading firmware onto a microcontroller 4.1, which is commonly referred to as flashing.

It can be concluded that ELF provides more information that can be useful when reconstructing the original source code. Hence, it is used as the supported format for the input files. It is worth mentioning that this choice does not have as much impact on the core functionality as it does on visual aspects, which is discussed more in detail in section 4.6.

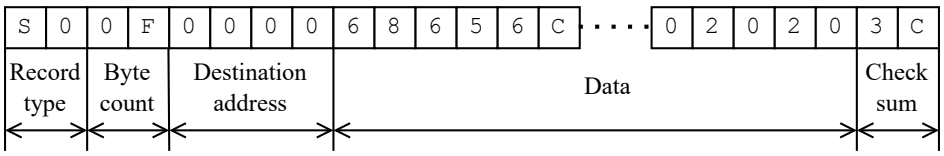

Figure 4.1: Single SREC record (16-bit addressing)

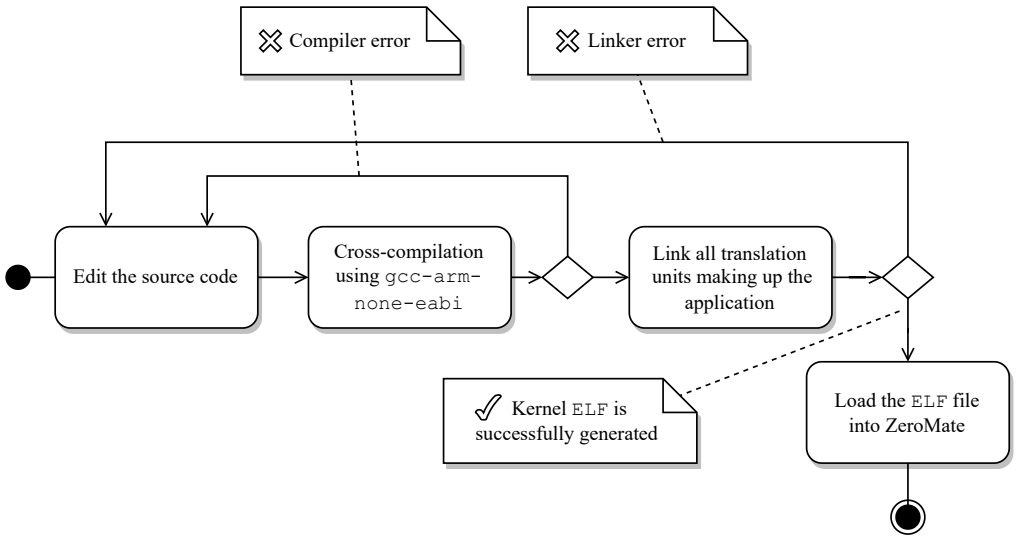

Figure 4.2: Process of building an ELF file (input for the emulator)[3]

[2] In the case of the `gcc-arm-none-eabi` compiler, the `-g -O0` flags should be used.

[3] Cross-compiling is a process where the source code targets a different platform than the one it is compiled on.

4.2 User's Interaction

As shown in the deployment diagram in figure 4.4, the emulator was **designed to run as a native desktop application on Windows, Linux, and MacOS operating systems**. It places a strong emphasis on visualization, serving as a debugging tool to assist with troubleshooting embedded applications targeting Raspberry Pi Zero.

The primary interaction with the system, from the user's perspective, is visualized in figure 4.3, where the user is provided with an interface that allows them to load an input file as well as to control the state of the emulation.

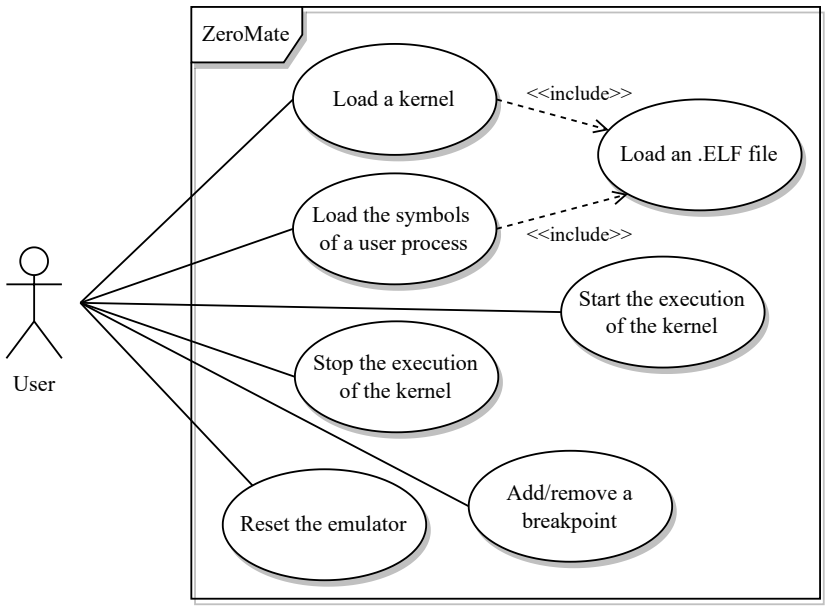

Figure 4.3: Primary use-cases of the ZeroMate emulator

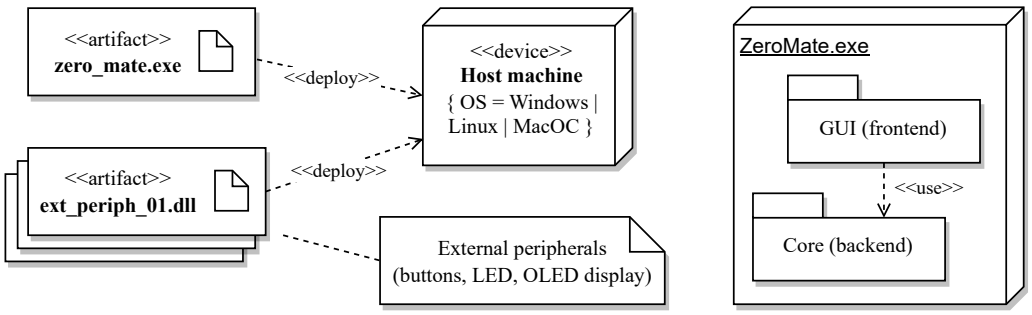

Figure 4.4: Deployment diagram of the ZeroMate emulator

The main application is designed as a **two-tier architecture**. In this arrangement, the top layer, which is the `GUI`, serves the dual purpose of visualizing data and acting as the primary user interface. The following chapters delve into the architectural structure of the core of the emulator.

4.3 Core Components

There are a number of different components working alongside to achieve a thorough emulation of a given kernel. Among these components, the `ARM1176JZF_S` component, which represents the `CPU` itself, may arguably stand out as the most complex one due to its encapsulation of various sub-components, including the `CPU` context, `ALU`, `MMU`, `ISA` decoder, and more. The role of every component will be examined further in the following sections.

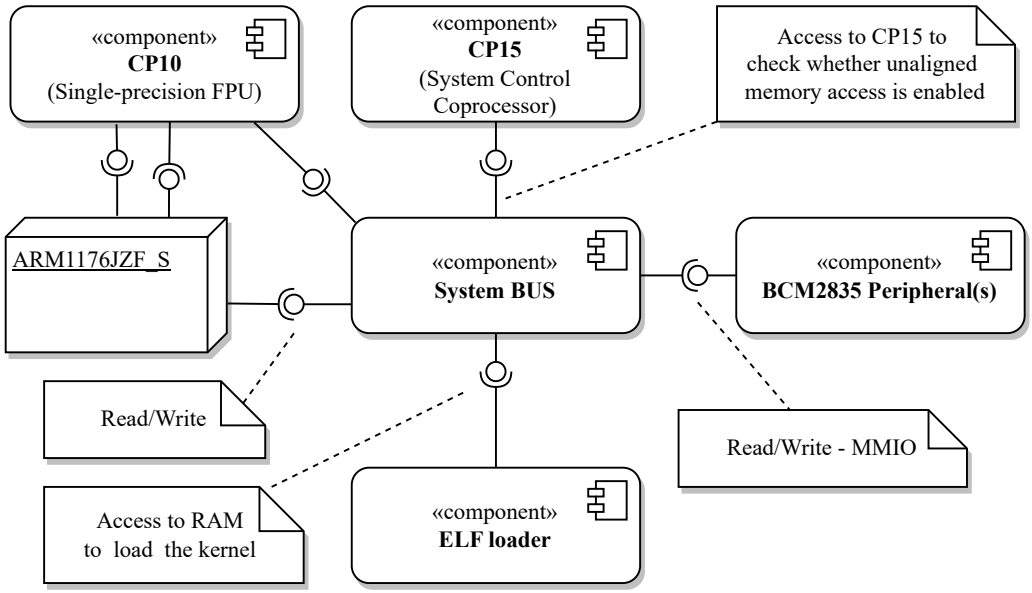

Figure 4.5: Core components of the ZeroMate emulator

Figure 4.5 illustrates the fundamental interactions among the core components. It can be observed that the majority of the components communicate with one another via the system bus [4]. For example, when the `CPU` executes a load/store instruction, it propagates the target address to the system bus, and the system bus then forwards the request to the corresponding peripheral associated with that address.

[4] What ZeroMate denotes as the system bus is typically regarded as the primary `CPU` bus.

4.3.1 System Bus

As mentioned previously, the system bus serves as an **intermediate unified interface** for accessing the `RAM` or any of the `BCM2835` memory-mapped peripherals [7]. Each peripheral that is meant to be mapped into the address space must implement the same interface, so the bus can forward the read/write request independently of the peripheral's implementation (see section 4.3.3). All actions associated with the request itself are then handled internally within the target peripheral.

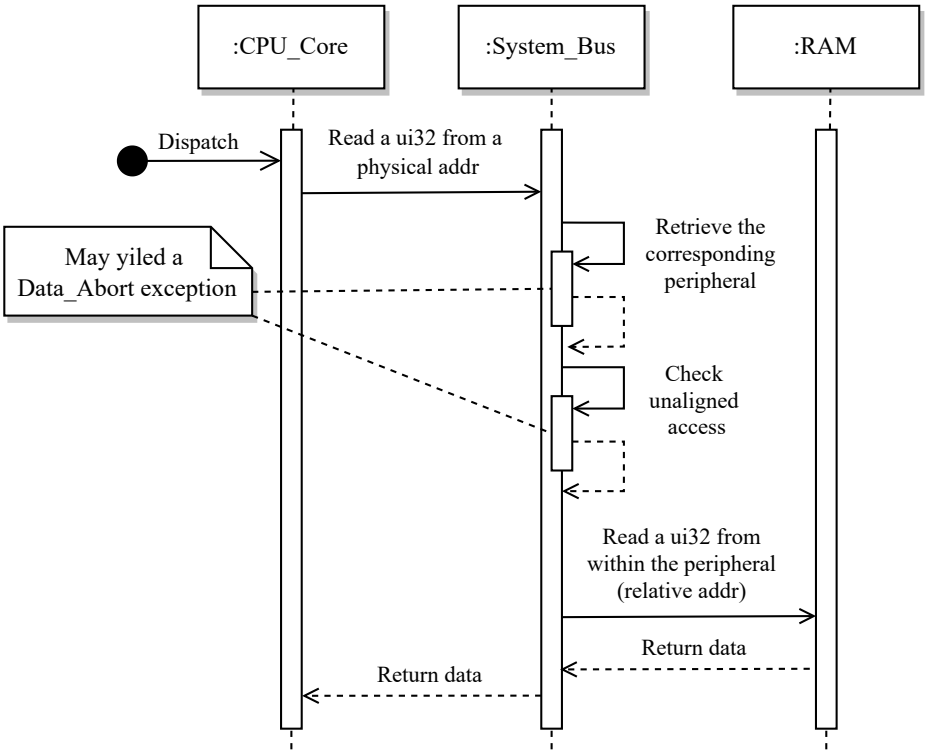

Figure 4.6: Example of a read/write data request issued by the `CPU`

As shown in figure 4.6, there are two internal steps the system bus carries out before proceeding with the request. First, the bus needs to determine what peripheral should the request be forwarded to. Secondly, it checks whether unaligned memory access is taking place or not.

> In reality, the main system bus does not manage peripherals the same way it does in ZeroMate. It only serves as a medium for connecting different types of memory-mapped devices. Nevertheless, from an architectural point of view, it is a reasonable place for implementing common validity checks as it plays the role of a single point of access to all memory-mapped peripherals.

Additionally, the system bus ensures that the peripheral receives an **address relative to its location in the address space** [5]. In other words, it does not have any knowledge about its location on the bus, which is desired, as it decreases coupling and increases cohesion between the two components [8].

Source code 4.1: System bus interface for `I/O` operations

```
class CBus final {
public:
    template<typename Type>
    void Write(std::uint32_t addr, Type value);

    template<typename Type>
    [[nodiscard]] Type Read(std::uint32_t addr);
};
```

It can be argued that permitting the reading or writing of a general data type may diverge from real hardware specifications, as the system bus is typically of a fixed size, e.g. 32 bits. This simplification was made for convenience reasons when accessing predefined data structures in the `RAM`, such as the page table(s).

4.3.1.1 Managing Peripherals

The system bus component maintains a collection of references to all memory-mapped input-output devices, further referred to as `MMIOs`. Whenever a peripheral needs to be attached to the bus, it is inserted into the appropriate position within the collection. This ensures that the entire collection remains sorted in ascending order based on the starting addresses of the peripherals. This property enables the use of a binary search algorithm, resulting in faster lookup times, particularly in $O(log_2 n)$ time complexity [9], which is crucial for improving the overall speed of the emulation.

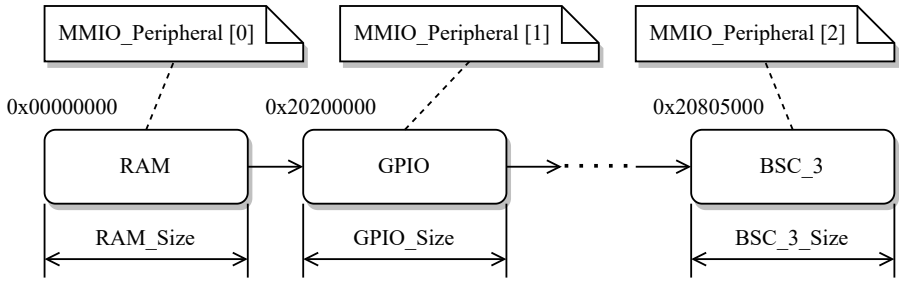

Figure 4.7: Collection of memory-mapped peripherals

[5] The relative address is calculated as the address contained in the `R/W` request issued by the `CPU` minus the address of the peripheral on the system bus.

According to statistics, on average, **load-store instructions account for more than 50% of all instructions in an x86 application** [10]. Although this research applies to a different architecture, it is reasonable to conclude that optimizing peripheral access efficiency might be crucial for emulation speed.

When connecting a peripheral, the bus must also ensure that there is no overlap between two peripherals and that they all fit within the address space, which, on a `32-bit` architecture, spans out to `4GB`.

4.3.1.2 Unaligned Memory Access

Unaligned memory access occurs when the **CPU attempts to read or write data from an address that is not divisible by the word size**. For example, reading 4 bytes from address `0x00000011` triggers unaligned access as the address is not `word`-aligned [6]. Nevertheless, this behavior can optionally be disabled, for example, for compatibility reasons, in the system control coprocessor `CP15` using the following sequence of instructions.

Source code 4.2: Enabling unaligned access in `CP15`

```
1 mrc p15, #0, r0, c1, c0, #0   ;@ Copy ctrl reg of CP15 to R0
2 orr r0,  #0x400000            ;@ Set bit 22 in R0
3 mcr p15, #0, r0, c1, c0, #0   ;@ Update CP15
```

[6] A `word` is a fixed-size number of bits that the `CPU` can process as a single unit. In the case of `ARM1176JZF_S`, one `word` equivocates to 4 bytes.

4.3.2 Executable and Linkage Format File Loader

The main objective of this module is to **parse an input ELF** file and copy the `.text` section, `word` by `word`, into `RAM`, as specified by the linker script. Additionally, it performs code disassembly, which is a process of reconstructing the source code from machine code. This allows the user to observe individual instructions in a more user-friendly way as they are executed.

For visualization purposes, the component also features the capability to parse an `ELF` file without copying its data into memory, which can be useful for viewing user processes that are compiled separately from the kernel itself. During this process, the **ELF loader also demangles all symbols** found in the input file [7], thereby presenting the user with function names that have not undergone modification by the compiler for its internal purposes.

Source code 4.3: Example of symbol demangling

```
1 Demangle("_ZNSt6vectorIiSaIiEE9push_backERKi") =
2 "std::vector<int,std::allocator<int>>::push_back(int␣const&)"
```

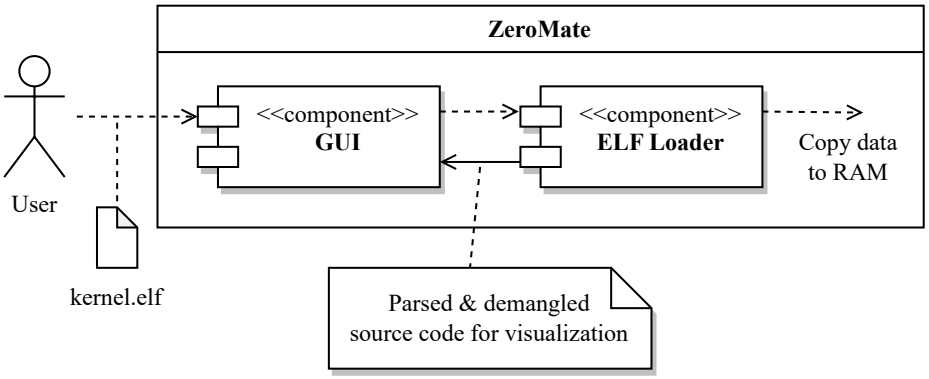

Figure 4.8: Loading an input `ELF` file (kernel)

[7] Demangling is a process of transforming `C/C++` `ABI` identifiers (like `RTTI` symbols) into the original `C/C++` source [11].

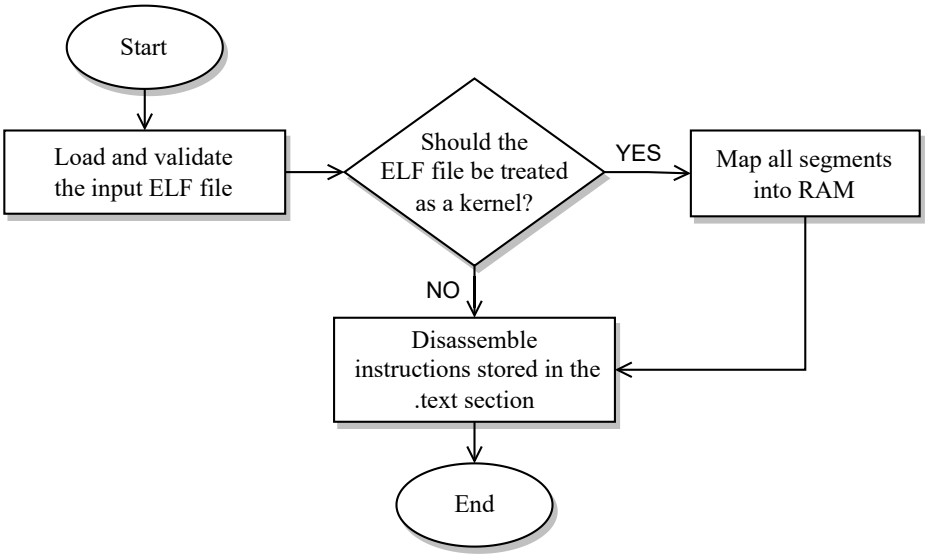

Figure 4.9: Internal logic of the ELF Loader component

It is important to emphasize that ZeroMate does not perform the tasks of parsing an ELF file and demangling symbols all by itself. Instead, it utilizes two external open-source libraries, *ELFIO* [12] and *Demumble* [13], to accomplish these functions.

4.3.3 BCM2835 Peripherals

ZeroMate distinguishes between two types of peripherals; those directly integrated with the microcontroller, such as RAM, ARM timer, or the interrupt controller, and those referred to as external peripherals, which are externally connected to another internal peripheral, the GPIO pins, forming a cascading set of connected peripherals. Examples of external peripherals may include buttons, switches, LEDs, displays, or keyboards. The following sections focus on the internal peripherals of Raspberry Pi Zero.

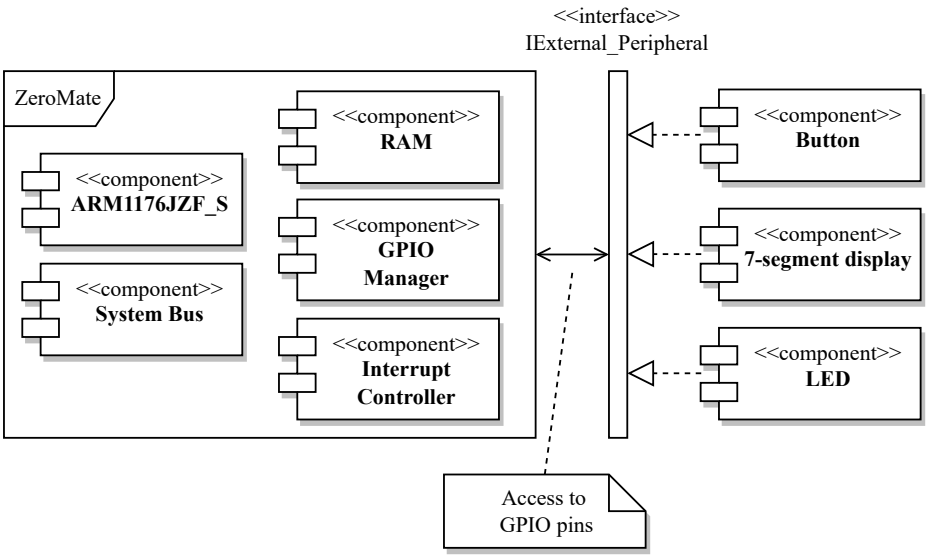

Figure 4.10: Internal vs External peripherals

In section 4.3.1, it is explained that the system bus manages a collection of references to all peripherals that are mapped into the address space. Using a general interface, the bus does not need to be concerned about how each peripheral functions internally. It simply forwards a `R/W` request initiated by the `CPU` to the corresponding peripheral.

Every `BCM2835` peripheral encapsulates a set of registers, whose functions are detailed in the manual [7] [8]. By reading from or writing to these registers, the internal state of the peripheral can be modified, which is specific for each peripheral.

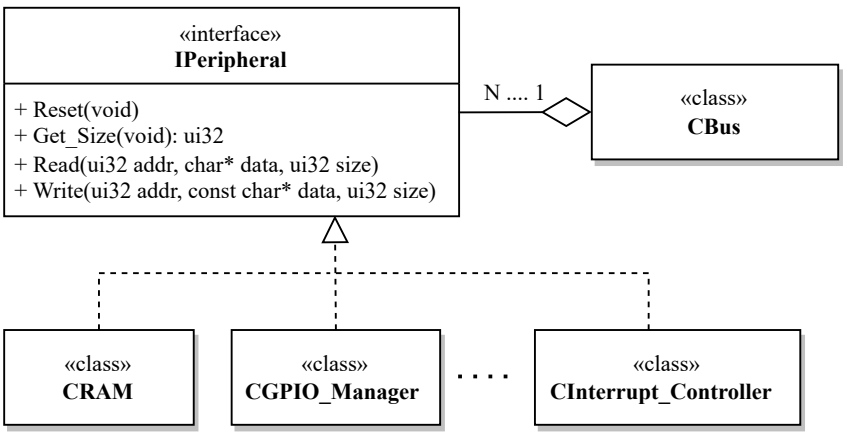

Figure 4.11: Hierarchy of internal peripherals

The `Get_Size()` method is used primarily for detecting bus collisions when

[8] The `BCM2835` manual is known to contain several typographical errors. As a result, the community surrounding it published a list of these errors along with their respective corrections [14].

mapping peripherals into the address space, which was mentioned previously in section 4.3.1.1.

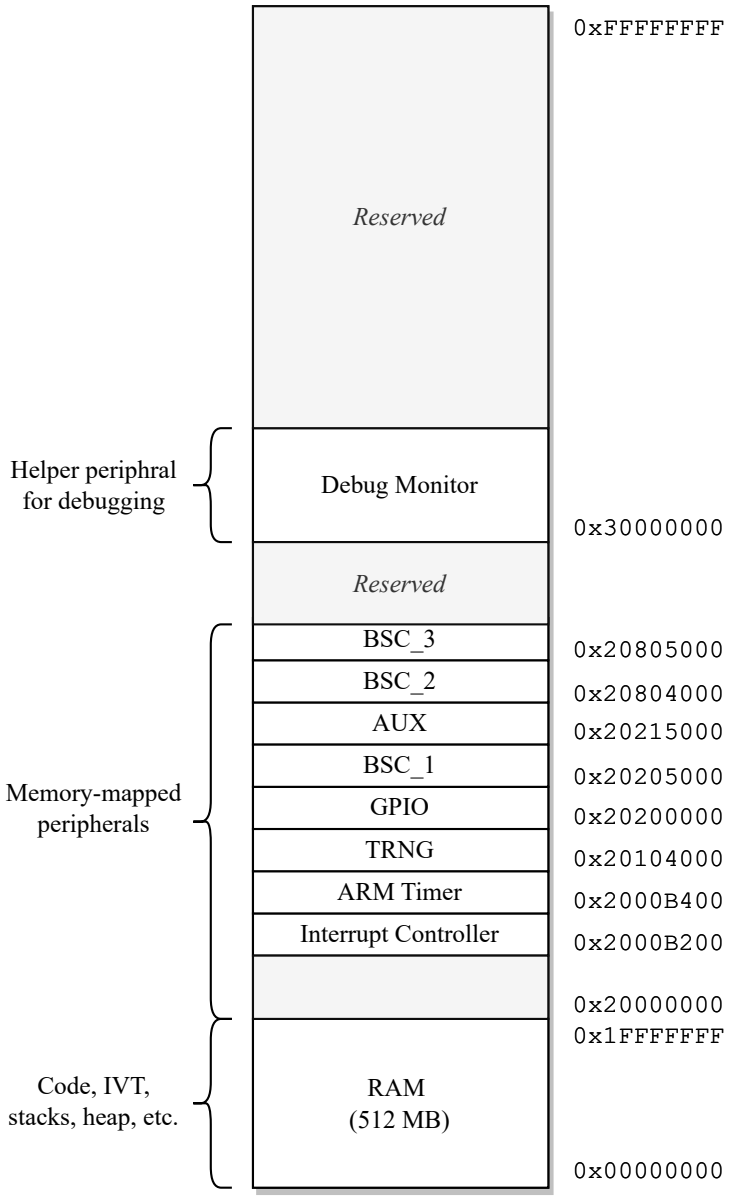

Figure 4.12: BCM2835 physical memory layout emulated by ZeroMate

It can be noticed that **ZeroMate does not account for all BCM2835 peripherals** since emulating every single one in its entirety would pose a significant complexity. Consequently, ZeroMate focuses emulation efforts on the most frequently utilized peripherals, such as the ARM timer, GPIO, IC, and others.

Nonetheless, the system's overall design is structured to allow for a smooth integration of additional peripherals in the future if needed. The following sections

explain the fundamental emulation principles of each of the peripherals listed in figure 4.12.

4.3.3.1 Memory-mapped Registers

As mentioned previously, each `BCM2835` peripheral encapsulates a set of registers, through which the user can interact with the peripheral. From an implementation perspective, these registers can be represented as a fixed-size array of `32-bit` integers, which are addressed relative to the peripheral's base address.

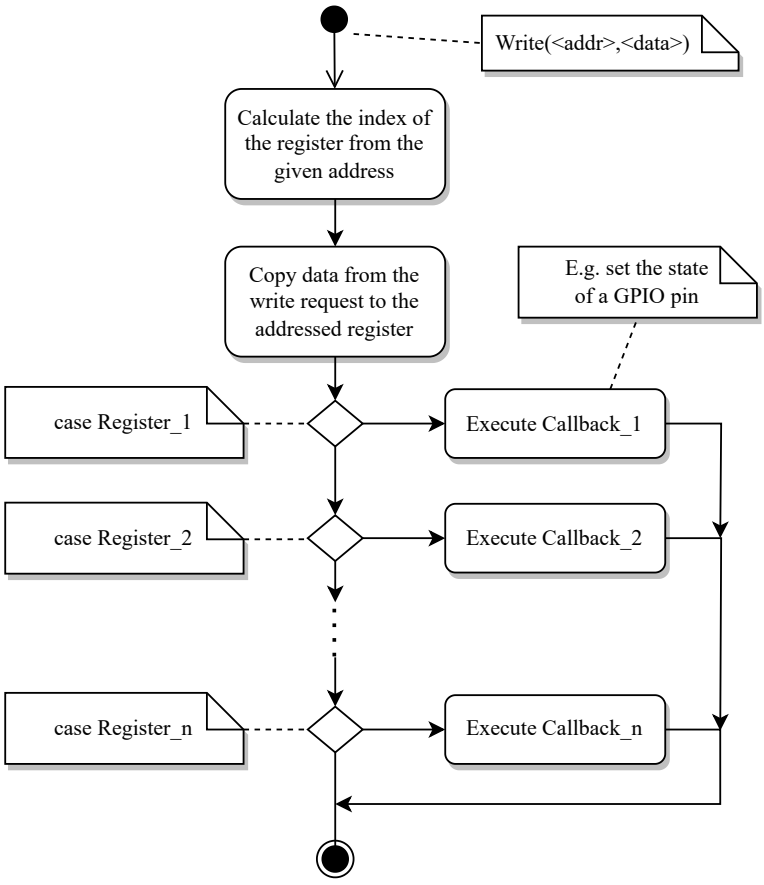

Figure 4.13: Writing to a peripheral's register[9]

Whenever a read/write request is sent through the bus, the peripheral identifies the addressed register, and the execution is then dispatched to the corresponding callback function, which carries out the necessary actions associated with that

[9] Reading works in a similar way. Optionally, there might be some prior actions taken before the value of the register is copied from the peripheral, such as inserting a random number into the `data` register when utilizing `TRGN` 4.3.3.5.

specific register. Generally, this approach can be applied to the vast majority of memory-mapped peripherals.

4.3.3.2 System Clock Listener

Optionally, each peripheral can implement the `ISystem_Clock_Listener` interface, which allows it to register with the CPU as a system clock listener. Whenever an instruction is executed, the CPU notifies all of its system clock listeners of how many CPU cycles it took to execute the instruction, allowing them to update themselves accordingly. Examples of such listeners may include the ARM Timer 4.3.3.6 or the AUX 4.3.3.9 and BSC 4.3.3.10 peripherals, which encapsulate time-based hardware communication functions.

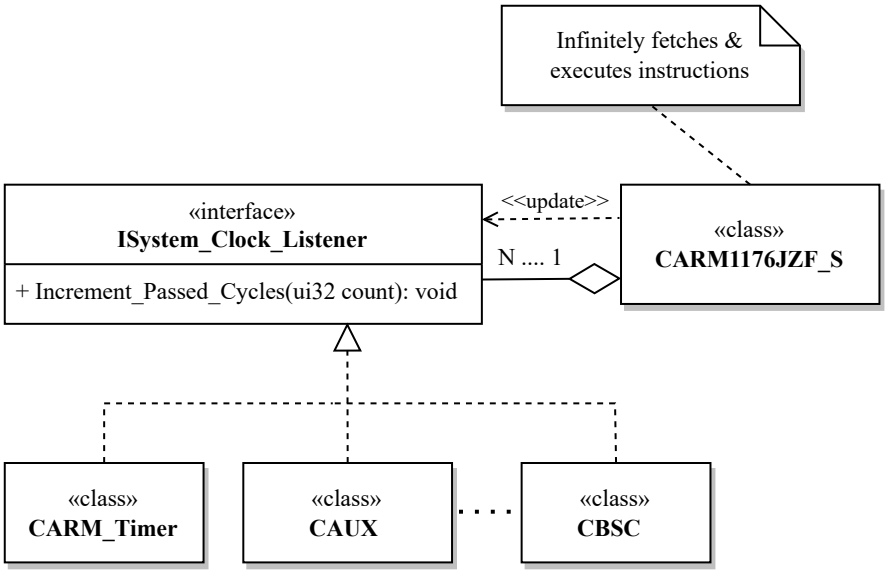

Figure 4.14: `ISystem_Clock_Listener` interface

It is important to mention that **updating a system clock listener is, from the emulated CPU's perspective, a blocking operation**. Therefore, the peripheral's callback function should avoid any unnecessary actions that might further prevent the CPU from executing the next instruction. Alternatively, updating system clock listeners could be performed asynchronously within a separate thread. However, this approach would introduce additional concurrency-related challenges that would require thorough consideration.

4.3.3.3 Random Access Memory

From an oversimplified perspective, a computer consists of two essential components; the CPU and memory. One key parameter used to classify various types of memory is their ability to retain data even after the power supply is shut down. Raspberry Pi Zero is equipped with an SD card slot, which houses non-volatile [10] memory for storing the kernel image. However, from ZeroMate's point of view, this type of memory is implicitly provided by the host machine.

The board is also featured with 512MB of RAM, which functions as volatile memory for executing the kernel code. It accommodates runtime-critical sections such as the stacks [11], heap, page tables, or the interrupt vector table, often referred to as the IVT.

The implementation of RAM is straightforward since it can be represented as an array of bytes as shown in the figure 4.15 below. However, the downside of this approach is that it immediately takes up 512 MB of the host's RAM, which may become an issue on older computers with limited resources.

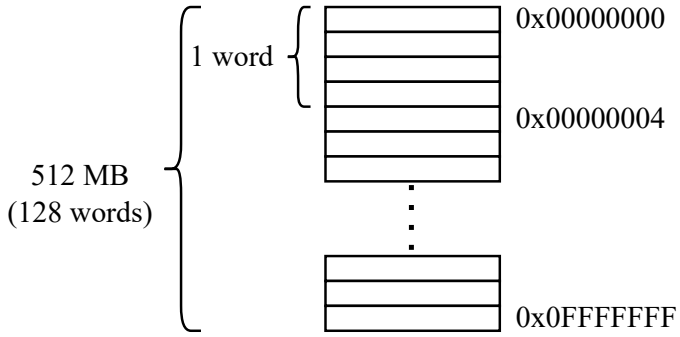

Figure 4.15: RAM implementation as a continuous piece of memory

A more effective approach would involve dynamically allocating fragmented pieces of memory as they are being addressed by the CPU. However, the author would argue that such an implementation would be algorithmically more complex, which could lead to distracting errors when implementing memory-related instruction, especially in the early stages of development. As a result, it was classified as a *nice-to-have* feature that would be worth addressing in the future once the emulator has been thoroughly QA-tested.

[10] Non-volatile memory is capable of persisting data even after the supply voltage is turned off.
[11] ARM1176JZF_S uses a different set of registers for each CPU mode.

4.3.3.4 Debug Monitor

The debug monitor plays the role of a memory-mapped output device for displaying `8-bit` character-based information.

> The component is not included in Raspberry Pi Zero itself; its presence serves solely for debugging purposes during the development of ZeroMate.

The ZeroMate project also includes a basic driver of the peripheral that the user can seamlessly integrate into their build system. This allows them to use **„print-like" functions** they might be familiar with from high-level programming languages, which may result in easier troubleshooting and resolving errors.

Source code 4.4: Demonstration of the use of the debug monitor

```
#include "monitor.h"

int main() {
    bool flag = false;
    unsigned int my_var = 155;

    sMonitor << "Hello␣World\n";
    sMonitor << "myVar␣=␣" << my_var << '\n';
    sMonitor << "flag␣=␣" << flag << '\n';

    return 0;
}
```

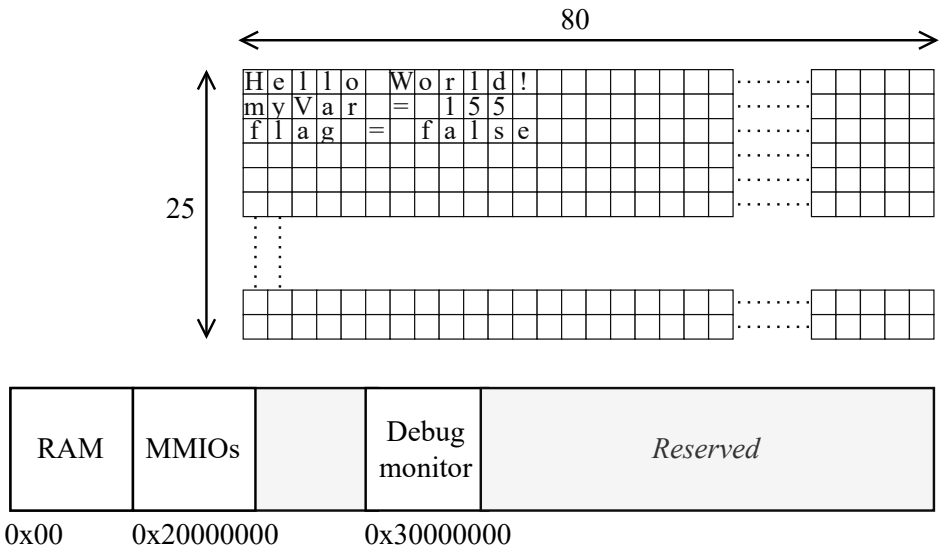

Figure 4.16: Memory-mapped debug monitor

To attain the same capabilities in practice, the user would need to utilize a form of serial communication, such as `UART`, through which they could transmit characters to an external device [12]. This is commonly achieved by running software like *PuTTY* [15] on the user's computer.

As shown in figure 4.16, the debug monitor is mapped to an unoccupied address `0x30000000`. It is structured as a flat memory region, which is managed by the driver the user code interacts with. The size of the monitor was chosen to be `80x25 8-bit` characters [13].

4.3.3.5 True Random Number Generator

The `TRGN` peripheral is an integrated `32-bit` hardware random number generator. Although it is not documented in the official `BCM2835` manual [7], its existence can be confirmed, for instance, by examining the implementation in the GNU/Linux kernel [16].

For simplification purposes, ZeroMate primarily focuses on providing random numbers while omitting more advanced features such as configuring the generator's speed, generating interrupts, or the warm-up count. The warm-up count refers to the process of generating and immediately discarding a set of random numbers before the initialization is completed [14].

From the user's code perspective, the process of **retrieving a random number consists of two steps**, which are displayed in figure 4.17.

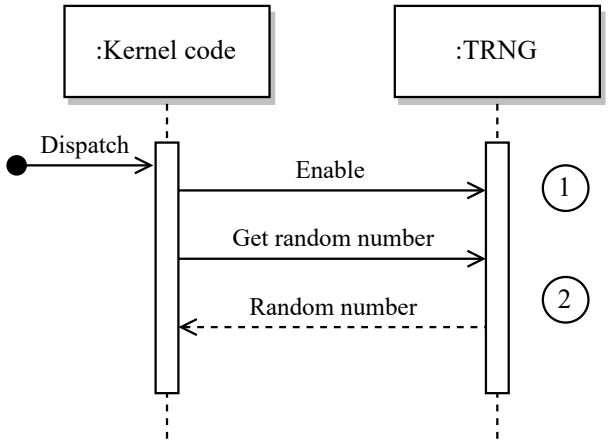

Figure 4.17: Reading random numbers from the `TRNG` peripheral

[12] While there are alternative methods to achieve the same functionality, this approach is among the most common ones.

[13] From an implementation point of view, it could vary in size as long as it does not overlap with other memory regions.

[14] The initial values are „less random".

Enabling TRNG.

The TRNG peripheral is enabled by setting bit 0 of the **control register** to 1. If implemented, this action would also trigger the processing of „warming up“, which was mentioned previously.

Reading random numbers.

First, the user should check the availability of random numbers in the TRNG's queue by examining the most significant 8 bits of the **status register**. If this number is 0, they should wait until the generator accumulates a sufficient amount of entropy to generate a random number. When data is ready, reading from the **data register** will retrieve a random number from the queue.

> ZeroMate can almost instantly generate a random number using a pseudo-random number generator. As a result, when reading the most significant 8 bits of the status register, the user will consistently receive the value 1, meaning they can read random numbers without delay.

Utilizing a pseudo-random number generator, such as an *LCG* [17] or *Mersenne-Twister*, **can greatly improve the performance of the emulation**. Depending on the implementation, accessing a true random number generator via the host operating system may have a detrimental impact on overall speed, as it may continually gather entropy from user inputs, like key presses or cursor movements. This can potentially lead to a blocking operation if there is currently insufficient entropy available.

4.3.3.6 ARM Timer

One of the most frequently used functions of the ARM timer is to periodically trigger interrupts, whether it is for toggling an LED, generating a PWM signal, or switching the current CPU context, which is an integral part of any preemptive OS scheduler.

As shown in figure 4.18, there are two data/control paths through which the timer can be interacted with. The first path, when the timer is treated as a memory-mapped peripheral, serves the purpose of reading from and writing to its internal registers in order to configure its desired functionality. This may involve steps such as setting up the prescaler, enabling interrupts, or defining the initial threshold value. The other path is used implicitly by the CPU to notify the peripheral about how many CPU cycles it took to execute the last instruction. The ARM timer then leverages the prescaler to divide the input frequency, as the main CPU frequency may not always be suitable for the given task.

As far as ZeroMate is concerned, all time-related functionalities, such as the ARM timer, UART, or I²C, are for synchronization purposes, inherently derived from the CPU's clock.

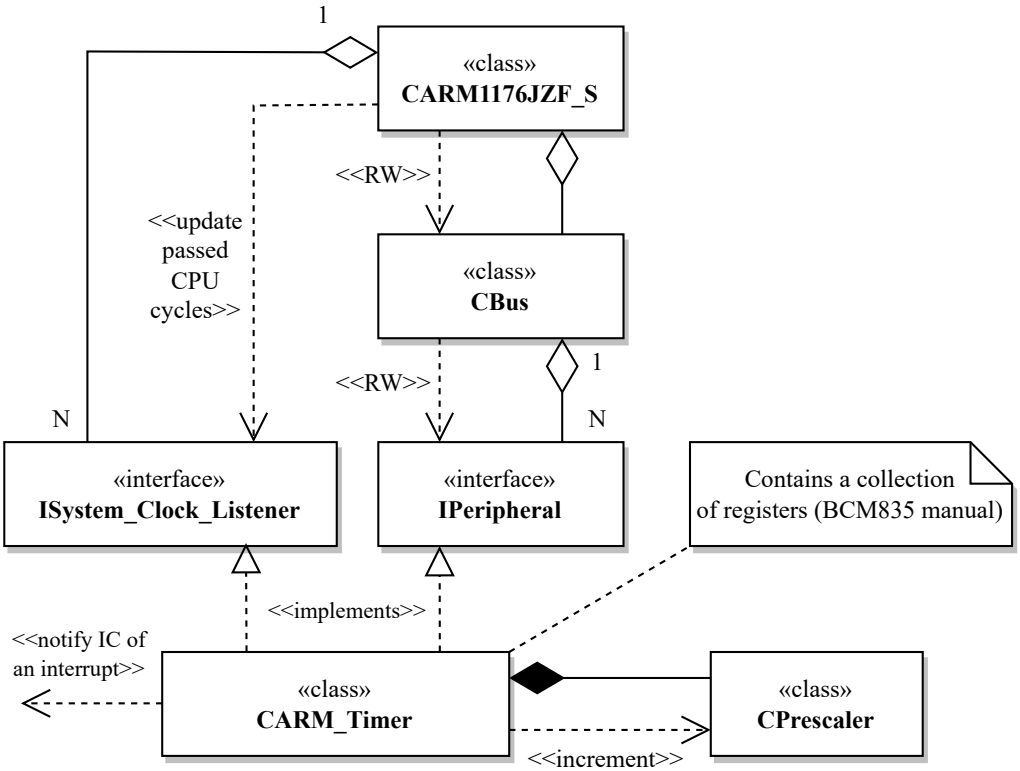

Figure 4.18: Context of the ARM timer component

As stated previously, the purpose of the prescaler is to divide the CPU's frequency by a factor of 1, 16, or 256, which ultimately affects the timer's period - how rapidly the **value register** counts down to zero. Additionally, the timer's period can be adjusted by modifying the value in the **load register**, which serves to re-initialize the value register whenever it reaches zero. If enabled, with each such event, the timer will trigger an interrupt. This concept is visualized in figure 4.19.

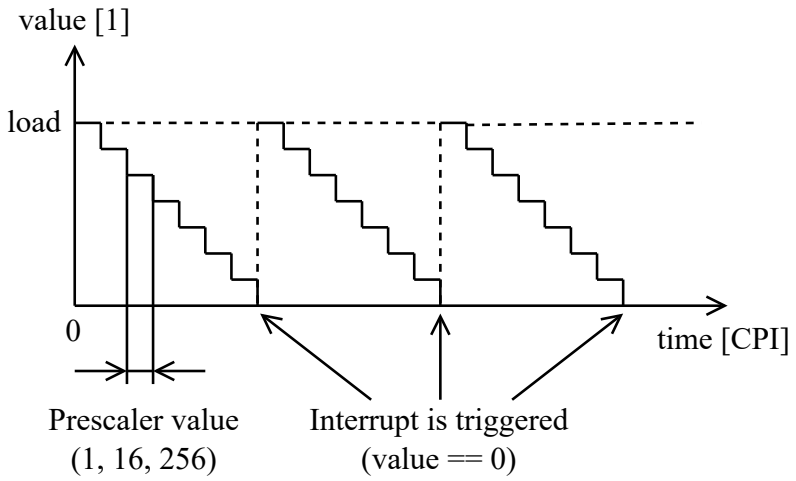

Figure 4.19: Content of the value register of the ARM timer over time

4.3.3.7 General Purpose Input/Output

The GPIO manager is a peripheral that manages the **54 general-purpose input-output pins** (GPIO) that are available to the programmer. These GPIO pins also function as a connecting interface for all external devices, which is shown in figure 4.10. Each pin is represented as a separate class encapsulating its current state, which consists of the pin's function, a list of enabled interrupts, an indication of any pending interrupts, and its current state.

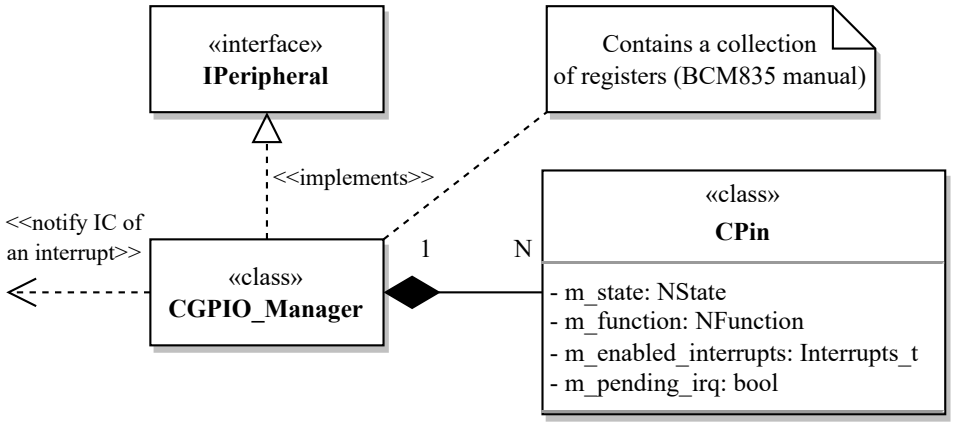

Figure 4.20: Structure of the GPIO manager[15]

All GPIO pin functions, as well as interrupt types, can be found explained in the BCM2835 manual [7].

The pin's function restricts the way the pin can be interacted with. For instance, when the user tries to read from an output pin, they will be prompted with a warning

[15]The CPin class also exposes a set of functions that allow the caller to access its private members.

message indicating that their action is inconsistent with the current pin configuration.

> ZeroMate does not provide support for analog pins. Therefore, all GPIO pins mentioned in this document are regarded as digital pins, with only two possible states, HIGH and LOW.

Emulation of latches.

Some of the GPIO registers work, on a hardware level, as latches, which may not be as intuitive from a software emulation point of view. An example of this principle would be the GPSETx and GPCLRx registers, which respectively set an output pin to a logical one and zero. These registers are stateless, meaning they do not retain their previous state internally. One approach to emulate such behavior is to reset the register to its default value after a write operation has been performed.

Source code 4.5: SW emulation of a HW latch register

```
1 void Set_Pin_High(std::uint32_t& gpset0)
2 {
3     // Iterate over all bits of gpset0.
4     // If gpset0[i]==1, then set the
5     // corresponding pin to HIGH.
6
7     gpset0 = 0; // SW latch emulation
8 }
```

Detecting Interrupts.

Whenever the state of a pin changes, a series of checks is performed to determine whether an interrupt has occurred. One of the most commonly used types of interrupts is triggered by a change in the logical value of a specific pin, either transitioning from HIGH to LOW or vice versa. The types of interrupt to be detected for each pin can be specified in the corresponding registers. When an interrupt is detected, it is reported to the interrupt controller, which can then initiate further actions, which is captured in figure 4.20.

4.3.3.8 Interrupt Controller

The interrupt controller serves as the primary interface for managing all peripherals that can generate interrupts. Through the interrupt controller, users can enable or disable various interrupt sources, including GPIO pins, the ARM timer, and the UART peripheral. **When an interrupt is signaled to the interrupt controller, it checks whether the source is enabled; otherwise, the event is disregarded**.

From the CPU's point of view, after an instruction is executed, it checks with the interrupt controller to ascertain the existence of any pending interrupts. If the interrupt controller has a record of a pending interrupt, and global interrupts are enabled, the CPU proceeds to throw an IRQ exception.

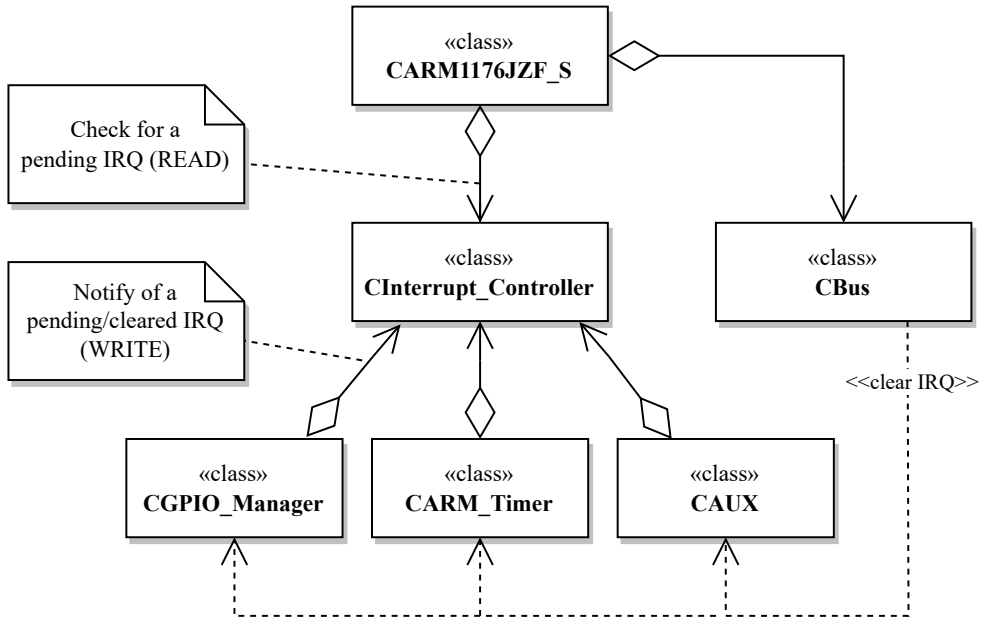

Figure 4.21: Context of the interrupt controller

In terms of design, the interrupt controller encapsulates an associative storage that pairs each IRQ source [16] with its associated metadata, indicating whether it is enabled and if there is a pending interrupt. From the CPU's perspective, this storage is read-only, as its only purpose is to check for any pending interrupts. The contents of this storage are modified by the peripherals themselves, either when they generate an interrupt or clear a pending interrupt.

The ARM1176JZF_S processor also features so-called fast interrupts, or FIQ for short. However, ZeroMate does not offer support for it as it primarily focuses on fundamental principles rather than more advanced features.

[16] You can find a comprehensive list of all available IRQ sources in the BCM2835 datasheet [7].

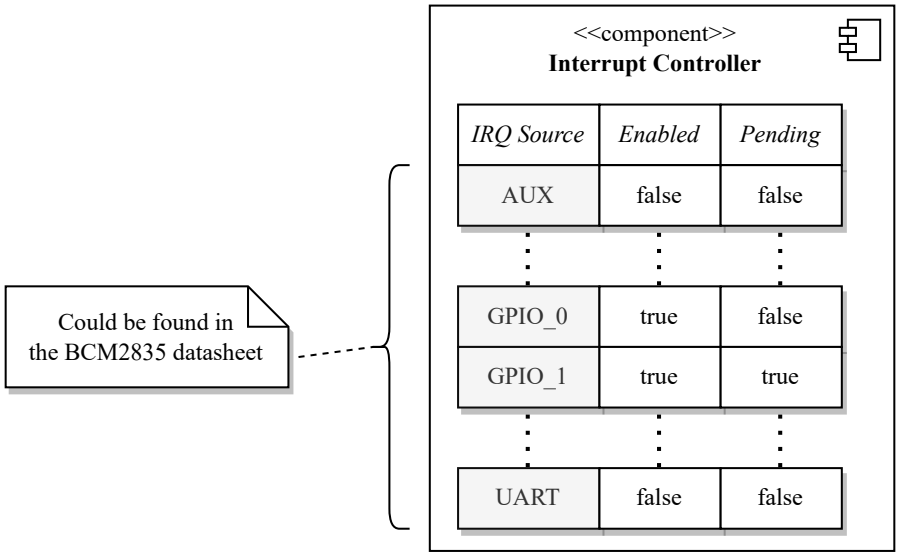

Figure 4.22: Encapsulated information about `IRQ` sources

It is worth mentioning that the interrupt controller of the `BCM2835` microcontroller distinguishes between two types of interrupt sources: Basic `IRQ` and `IRQ`, both of which are listed in the `BCM2835` datasheet [7]. Nevertheless, from a design perspective, the underlying principles remain the same.

4.3.3.9 Auxiliaries

The auxiliary peripheral comprises three distinct peripherals: `Mini_UART`, `SPI_0`, and `SPI_1`. Among these, only `Mini_UART` is currently supported by ZeroMate.

There are two primary registers shared among all auxiliary peripherals: the `enable` register, responsible for activating the respective peripheral, and the `IRQ` register, which signals pending interrupts. The remaining registers are specific to each peripheral, as depicted in the figure below.

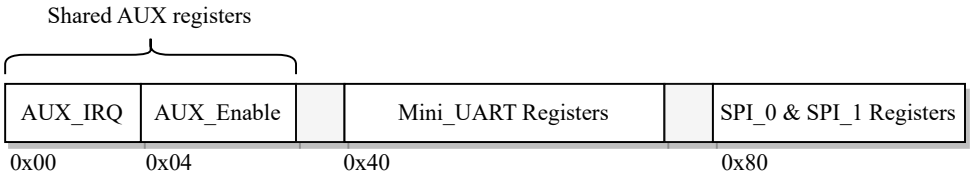

Figure 4.23: Registers of the `AUX` peripheral

Whenever a read/write request is received, using the technique described in section 4.3.3.1, the `AUX` class can efficiently redirect the execution to the relevant auxiliary peripheral, which will subsequently handle the request internally.

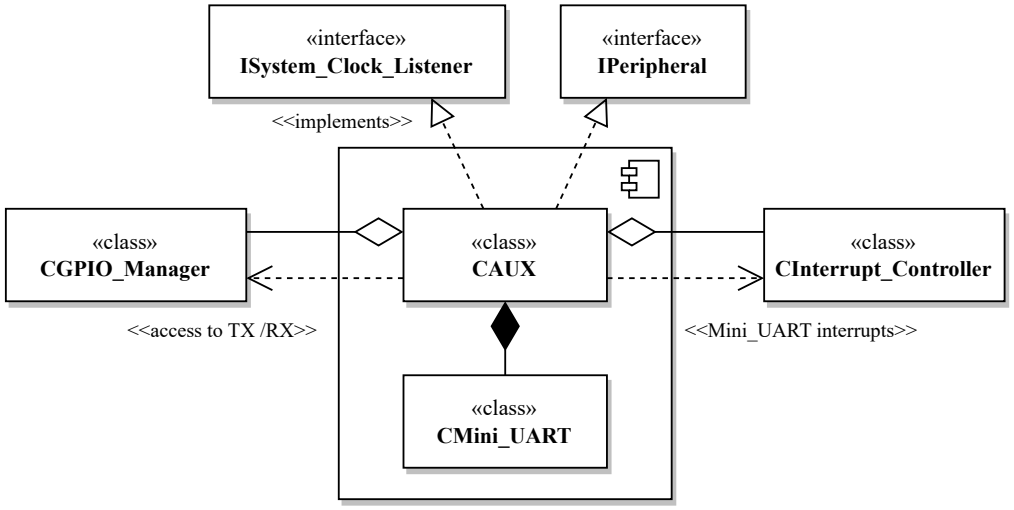

Figure 4.24: Structure of the AUX peripheral

Figure 4.24 shows the internal structure and dependencies of the AUX peripheral. It can be noticed that it implements the ISystem_Clock_Listener interface 4.3.3.2, which allows it to synchronize with the rest of the system.

Mini_UART.

UART, short Universal Asynchronous Receiver-Transmitter, inherently operates as asynchronous communication, which means there is no explicit synchronization between the two devices. Since these devices may have different clock speeds, they must adjust their frequencies to establish a common speed known as the baudrate, which expresses how many bits can be received/transmitted per second.

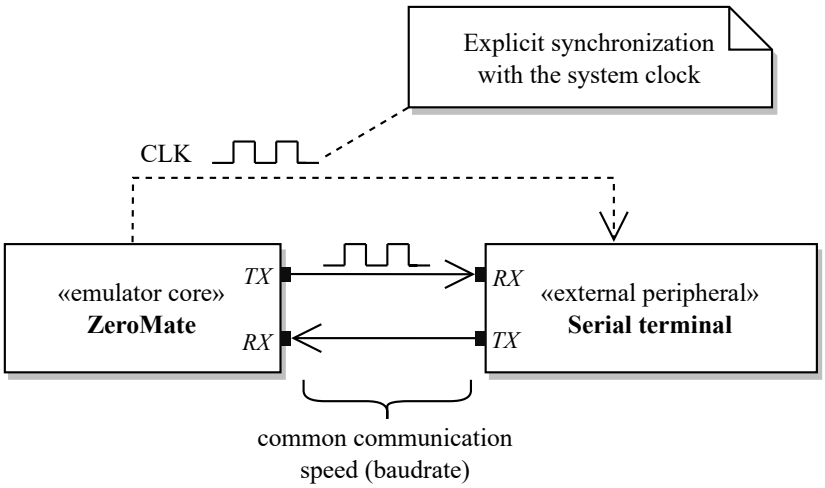

Figure 4.25: UART communication with an external peripheral

The BCM2835 microcontroller does not incorporate a full version of UART. Instead, it supports a simplified version known as Mini_UART, which omits some of

the extended features. As a result, the user is only able to modify the baudrate and the number of data bits transferred within a single frame, which can be either seven or eight. The remaining parameters, such as parity and the number of stop bits, are fixed according to the datasheet [7].

In ZeroMate, all external peripherals are provided read-only access to the system clock, enabling them to synchronize themselves if required. This approach contradicts the fundamental principles of asynchronous communication since it introduces a form of synchronization. However, this design decision was made to enhance the emulation's reliability while still enabling users to utilize `Mini_UART` as if they were interacting with real hardware.

The implementation of `Mini_UART` communication can be accomplished through a state machine driven by a pre-divided system clock as shown in figure 4.27. When a predefined number of `CPU` cycles have passed, the state machine updates itself to transmit and receive the next bit of the current data frame, which can be seen in figure 4.26 [17].

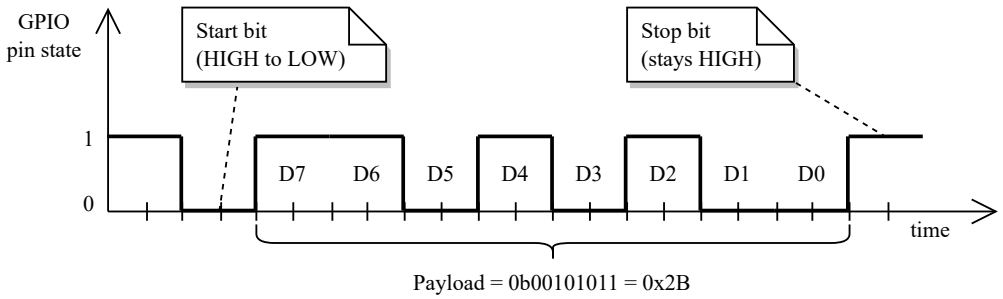

Figure 4.26: Example of a `Mini_UART` data frame with 8 bits of data

Receiving data can be implemented in a similar manner. Instead of setting the value of the `TX` pin, the state machine reads the value of the `RX`, which has been set previously by the external peripheral.

[17] Transmitting a single bit can be done by setting the corresponding `TX` pin to the desired value. Further details on what `GPIO` pins are designated for `Mini_UART` can be found in the datasheet [7].

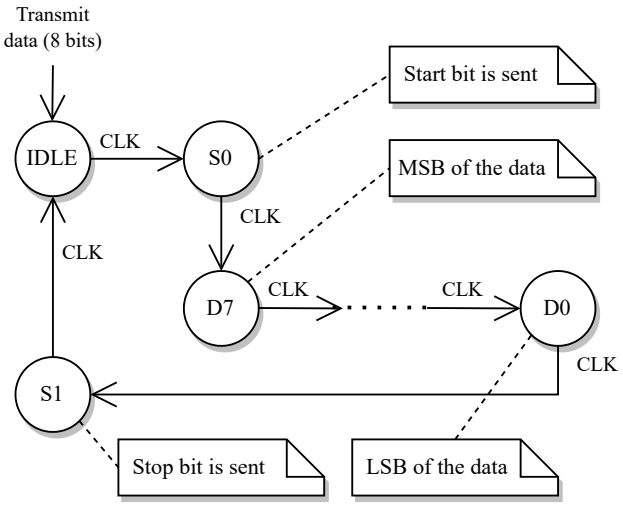

Figure 4.27: `Mini_UART` state machine (transmitting 8 bits)

4.3.3.10 Broadcom Serial Controller

The Broadcom serial controller, often abbreviated as BSC, is a peripheral device that allows the user to communicate with external devices, typically sensors, using the I^2C protocol, which is a synchronous serial communication standard. I^2C utilizes two GPIO pins: SDA for data transmission and SCL for synchronization. Raspberry Pi Zero is equipped with three of these devices that can be found mapped to their respective addresses in figure 4.12.

From a design point of view, the emulation of an I^2C bus is nearly identical to UART, with the primary distinctions being the frame structure and the synchronous transmission of individual bits using an additional GPIO pin instead of the emulated CPU's clock.

While there are various configurations of the I^2C communication protocol [a], ZeroMate exclusively supports only one, which is presented in figure 4.28. As stated previously, ZeroMate emphasizes the emulation of fundamental principles, rather than attempting to implement every conceivable configuration, which would only add unnecessary complexity without delivering any added value.

[a] They vary in voltage levels for representing logical 1 and 0, as well as in how they define the start and stop bits using the SDA and SCL signals.

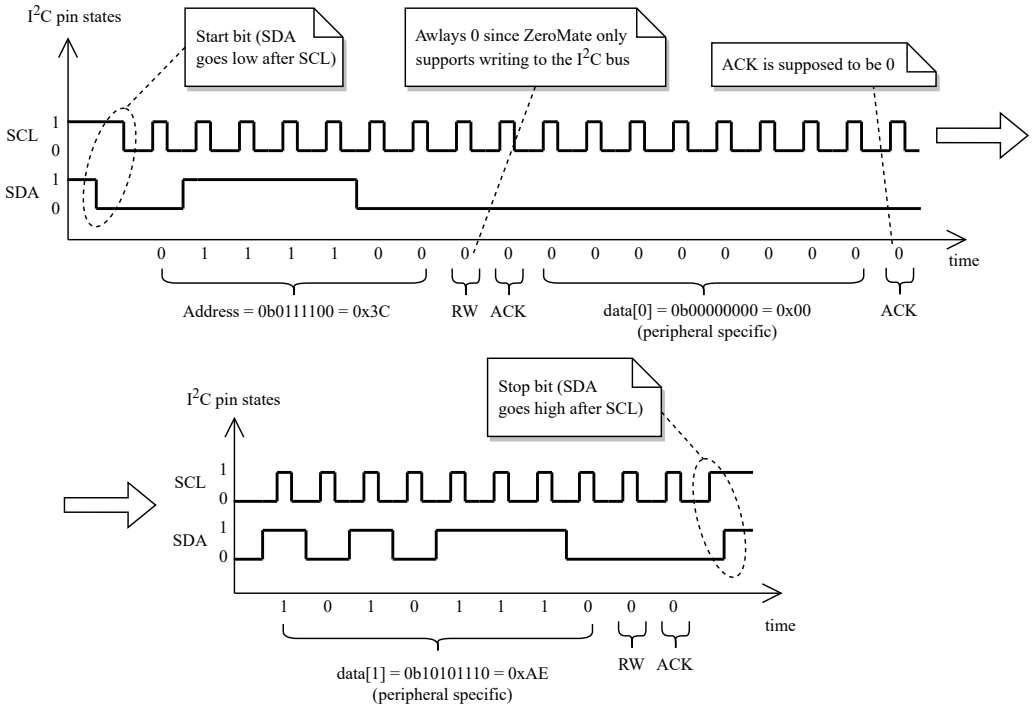

Figure 4.28: Structure of an I²C frame

The ZeroMate emulator comes with several standalone external peripherals that employ the same communication interface. Users can connect these peripherals to the system based on their preferences using a configuration file, which is further discussed in section 4.4.

4.3.4 ARM1176JZF_S

All the previously mentioned peripherals are not self-sufficient in operation; they require external control. This is where `ARM1176JZF_S`, which serves as the central processing unit in the `BCM2835` microcontroller, comes into play, as it executes individual `32-bit ARM` instructions that may utilize these peripherals in various ways.

In terms of architecture, the `ARM1176JZF_S` component is divided into several tightly integrated building blocks, as shown in figure 4.29, to maintain a structured and organized design. The subsequent sections detail how each of these subperipherals contributes to the overall emulation of the `ARM1176JZF_S` processor.

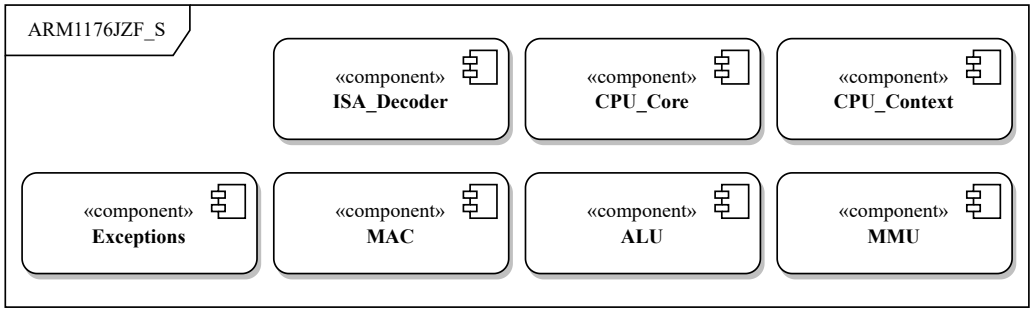

Figure 4.29: Internal components of the `ARM1176JZF_S` processor

4.3.4.1 Central Processing Unit Context

The CPU context functions as an encapsulation of the current state of the CPU, including details such as the current contents of the registers and the active CPU mode. It is designed to provide an interface for accessing registers, enabling transitions between different modes, and offering other utility functions that abstract the underlying low-level logic, such as reading the state of individual bits of the control register.

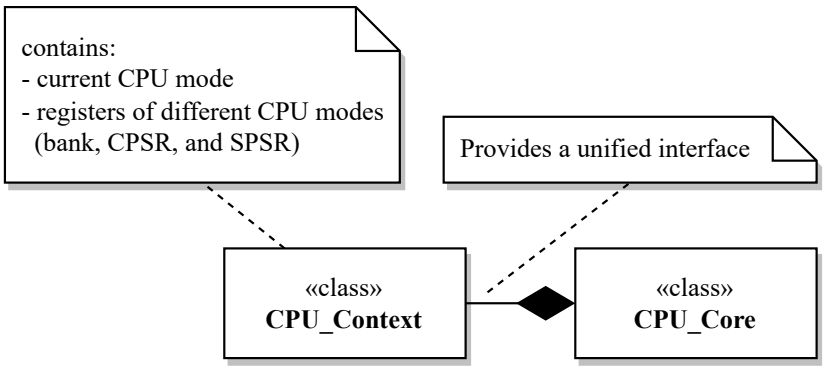

Figure 4.30: Relationship between the CPU context and the CPU core

Bank Registers.

One of the presented challenges involves implementing so-called *bank* registers, where each CPU mode possesses its distinct subset of registers that are automatically loaded when the mode changes. Additional information regarding this topic is available in section A2.3 of the official ARM Architecture Reference Manual [4].

As illustrated in listing 4.6, one approach to address this challenge involves creating a lookup table for all bank registers within each CPU mode, effectively replacing the ones used in the User/System mode, which are otherwise used by default.

Source code 4.6: Retrieving a CPU register

```
uint32_t& Get_Register(uint32_t idx, NCPU_Mode mode)
{
    // Check if a banked register should be returned
    if (m_banked_regs.at(mode).contains(idx))
        return m_banked_regs[mode][idx];

    // A non-banked register is being addressed
    return m_banked_regs[NCPU_Mode::System][idx];
}
```

Control Registers.

The current program state register and the saved program state registers, commonly known as the CPSR and SPSR registers, are implemented in a similar way, with each CPU mode having its designated pair of these registers.

4.3.4.2 Instruction Set Architecture Decoder

The primary role of the instruction set architecture decoder, abbreviated as the ISA decoder, is to analyze a 32-bit value and ascertain the type of the ARM instruction it represents, so it could be treated and decoded accordingly. This task must be executed with the utmost efficiency, as it is repetitively performed for each instruction the CPU executes.

In the case of the emulated CPU, the ISA decoder functions as a „black box“ offering a single-function interface, as demonstrated in listing 4.7 below.

Source code 4.7: Interface of the ISA decoder

```
// Returns the type of a given 32-bit ARM instruction
[[nodiscard]] static CInstruction::NType
Get_Instruction_Type(uint32_t instruction) noexcept;
```

The typical and sole use-case of this interface is further illustrated in the sequence diagram shown in Figure 4.31. In this diagram, the CPU fetches the next instruction from RAM and employs the ISA decoder to determine its type, enabling it to proceed with the execution.

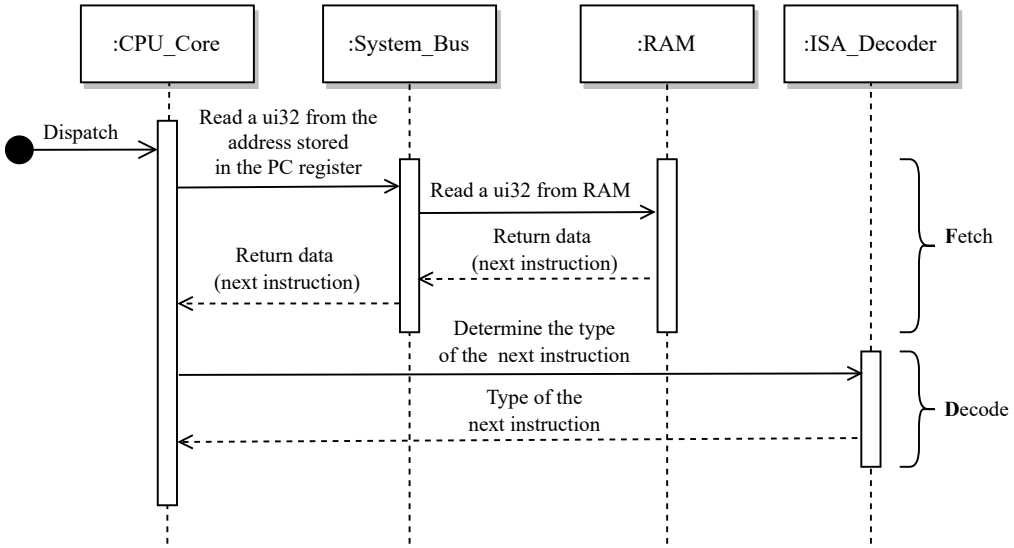

Figure 4.31: Use of the `ISA` decoder by the `CPU`

Internally, the `ISA` decoder maintains a look-up table of instructions masks that are sequentially applied to the given `32-bit` value until the operation's result matches the expected value associated with the current mask. Each mask serves the purpose of zeroing out the variable bits specific to the instruction, leaving only the known bits in place, the expected value, which is then used to unambiguously determine the type of instruction.

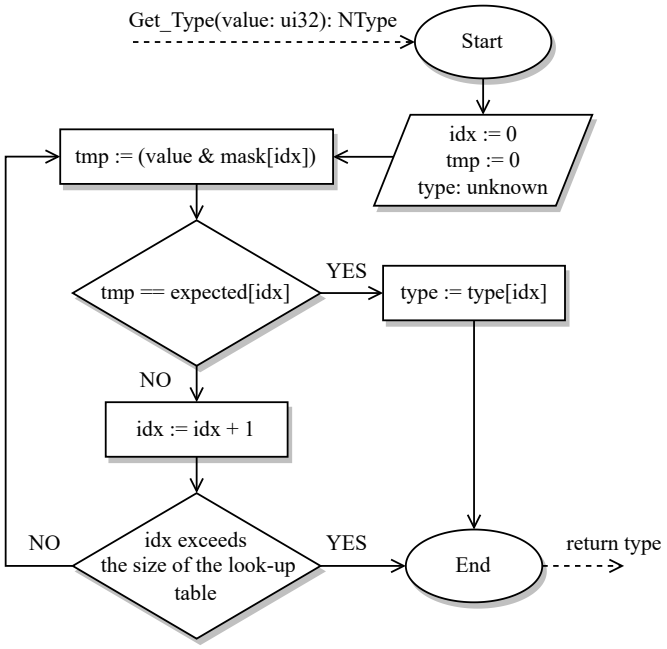

Figure 4.32: Algorithm for decoding `ARM` instructions

Figure 4.32 shows the algorithm for decoding `ARM` instructions, which, in the

worst-case scenario, operates with time complexity of $O(n)$, where n represents the size of the look-up table. **To ensure unambiguous decoding, it is essential to test the bit masks in a specific order, starting with the most restrictive one and proceeding to the least restrictive one, based on the number of bits set within each mask**. Otherwise, there might be a risk of incorrectly identifying the instruction type, inevitably leading to unexpected behavior.

Once an instruction is classified, the `32-bit` value can be encapsulated within its corresponding class representation, providing an interface to access the specific fields relevant to that instruction. The contents of the look-up table can be constructed using resources such as the B2 Appendix authored by Andrew Sloss and Chris Wright, which provides ARM instruction encodings [18].

4.3.4.3 Exceptions

Exceptions can originate from various components within the ZeroMate emulator. These exceptions may arise from factors such as the absence of an addressed page, unaligned memory access, or the execution of a privileged instruction in a non-privileged CPU mode. ZeroMate handles ARM CPU exceptions as runtime errors on the host machine. As a result, the emulated CPU core must be prepared for the possibility that the currently executing instruction may abruptly trigger an exception that must be properly handled. Figure 4.33 shows the hierarchy of CPU exceptions, where each class may include additional information pertaining to the exception, such as a descriptive message, the address at which it occurred, or the address of the vector associated with that specific exception.

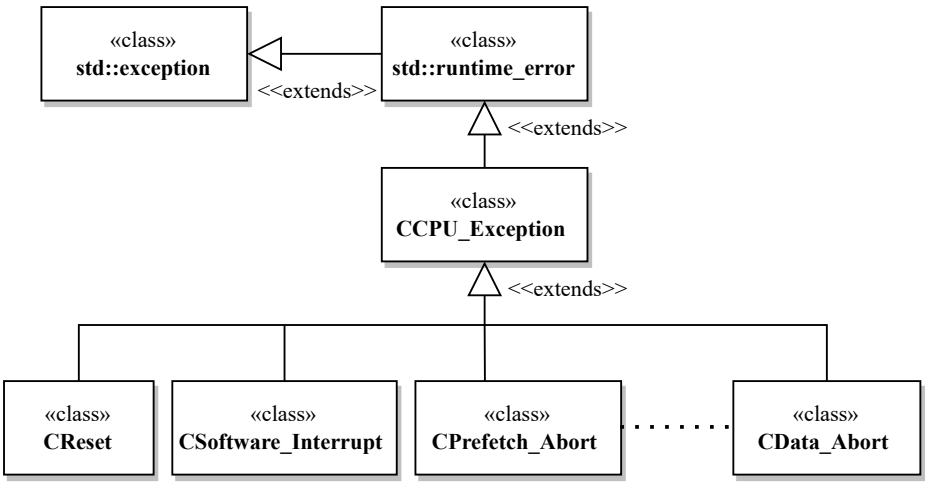

Figure 4.33: Hierarchy of the CPU exceptions

4.3.4.4 Arithmetic-Logic Unit

As the name suggests, the arithmetic-logic unit, also known as the ALU, is responsible for carrying out **arithmetic and logical operations**, such as addition, subtraction, comparison, etc.

From a design standpoint, the ALU can be envisioned as a set of collaborative functions concealed behind a single function interface, which is made accessible to the central processing unit as shown in figure 4.34.

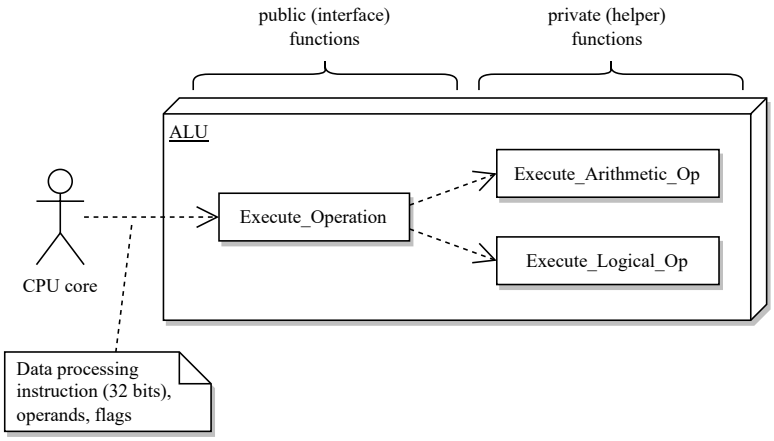

Figure 4.34: Architectural structure of the ALU

As illustrated in the figure above, when the ALU is employed, the CPU must provide both the current state of its flags and the data processing instruction currently in execution, which is internally analyzed by the ALU to determine the specific type of operation to be carried out.

MAC Unit.

The ARM1176JZF_S also incorporates a module referred to as MAC, specifically designed for performing a variety of **multiplication operations**. These operations can involve the multiplication of either signed or unsigned 32-bit integers, signed or unsigned 64-bit integers, or multiplication with addition, where a third value is added to the result of the multiplication. All instruction encodings can be found in the B2 Appendix document [18]. From an implementation perspective, it can be integrated in a manner similar to the ALU.

4.3.4.5 Memory Management Unit

If enabled, the memory management unit, often referred to as the MMU, comes into play just before the CPU sends a read/write request to the system bus. **Its primary function is to convert a 32-bit virtual address into a corresponding 32-bit**

physical address based on the information stored in the corresponding page table, allowing the user to reorganize the address space to their needs. Furthermore, it performs a series of checks, the failure of which could result in a MMU abort [18].

1. In the initial step, the MMU retrieves the page associated with the address from the first-level page table to ascertain its type. This type indicates whether it references a nested second-level page table or a physical frame. In the latter case, the MMU also confirms the page's presence in RAM, as it might have been swapped out, for example, to a file.

2. When addressing a physical frame, the MMU checks access privileges to the targeted frame based on the current mode of the CPU in order to ensure that no security policy is being violated.

If all checks have been successfully passed, the MMU proceeds to convert the virtual address into a corresponding physical one. More detailed information about how the MMU operates can be found in chapter 6 of the ARM1176JZF_S Technical Reference Manual [19].

The ZeroMate emulator lacks support for nested page tables, which inevitably introduces certain limitations in the emulation. As a result, it utilizes a first-level page table that spans over the entire address space.

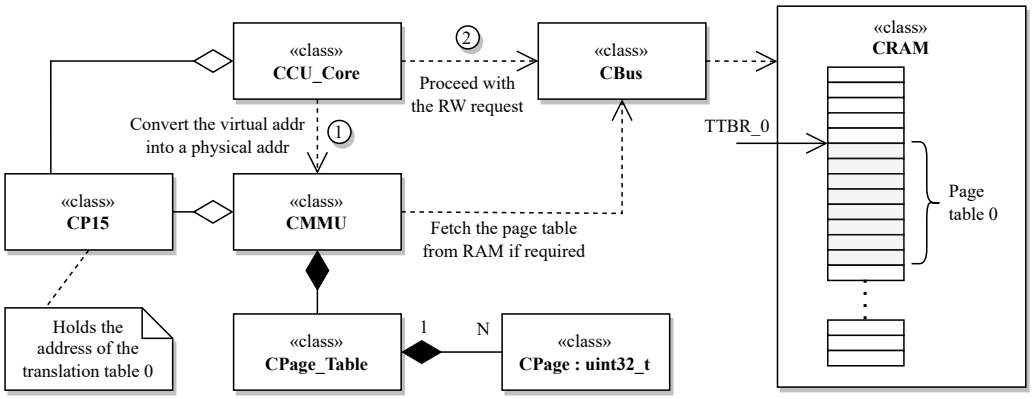

Figure 4.35: Utilization of the MMU when processing a RW request

Placing a strong emphasis on the Memory Management Unit (MMU), Figure 4.35 illustrates the essential steps to be taken when executing a read/write instruction. Whenever the address of page table 0 changes in the TTBR_0 register, the MMU retrieves the entire page table from RAM and stores it within its private data structure.

[18] Users of the x86 architecture might already be acquainted with the concept of a *page fault*, which, in this context, can be thought of in a similar way.

This approach eliminates the need to repeatedly read it for each memory access, which would otherwise negatively impact the overall performance of the emulation. Once a physical address is obtained, the CPU can then proceed with the request.

Implementation.

The implementation of a first-level page table can be tacked as an array of 4096 classes called `CPage`[19]. These classes serve as an abstraction for accessing individual fields of individual pages, where each page is represented as a `32-bit` value.

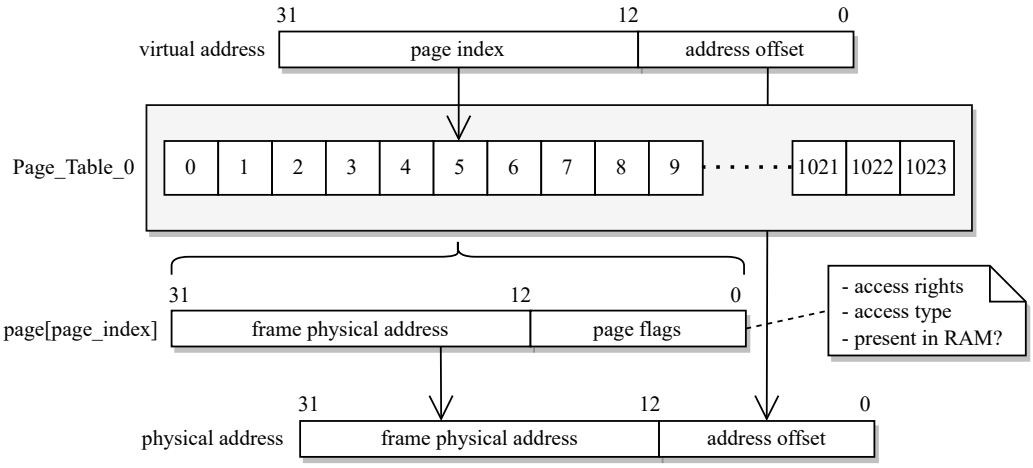

Figure 4.36: Structure of the first-level page table

Translation Lookaside Buffer.

Once a virtual address undergoes the process of translation, it is stored, along with its corresponding physical address, in a look-up table [20]. In a real system, this table is known as the Translation Lookaside Buffer, or TLB for short. The next time the same address is used, it doesn't have to go through the translation process again, which enhances performance. When a request is made to invalidate the TLB through coprocessor 15, ZeroMate clears this associative data structure to prevent invalid addressing across various virtual address spaces.

[19] Assuming the page granularity is `1MB`, covering the entire `4GB` address space requires 4096 page table entries.

[20] It can be implemented, for instance, as an `std::unordered_map<uint32_t, uint32_t>`

4.3.4.6 Central Processing Unit Core

The central processing unit utilizes the functionality of all previously mentioned components. In terms of design, it is composed of multiple private member functions that are invoked based on the type of instruction currently being executed. These functions can be seen as microprograms, as they handle the specific operations associated with the current instruction.

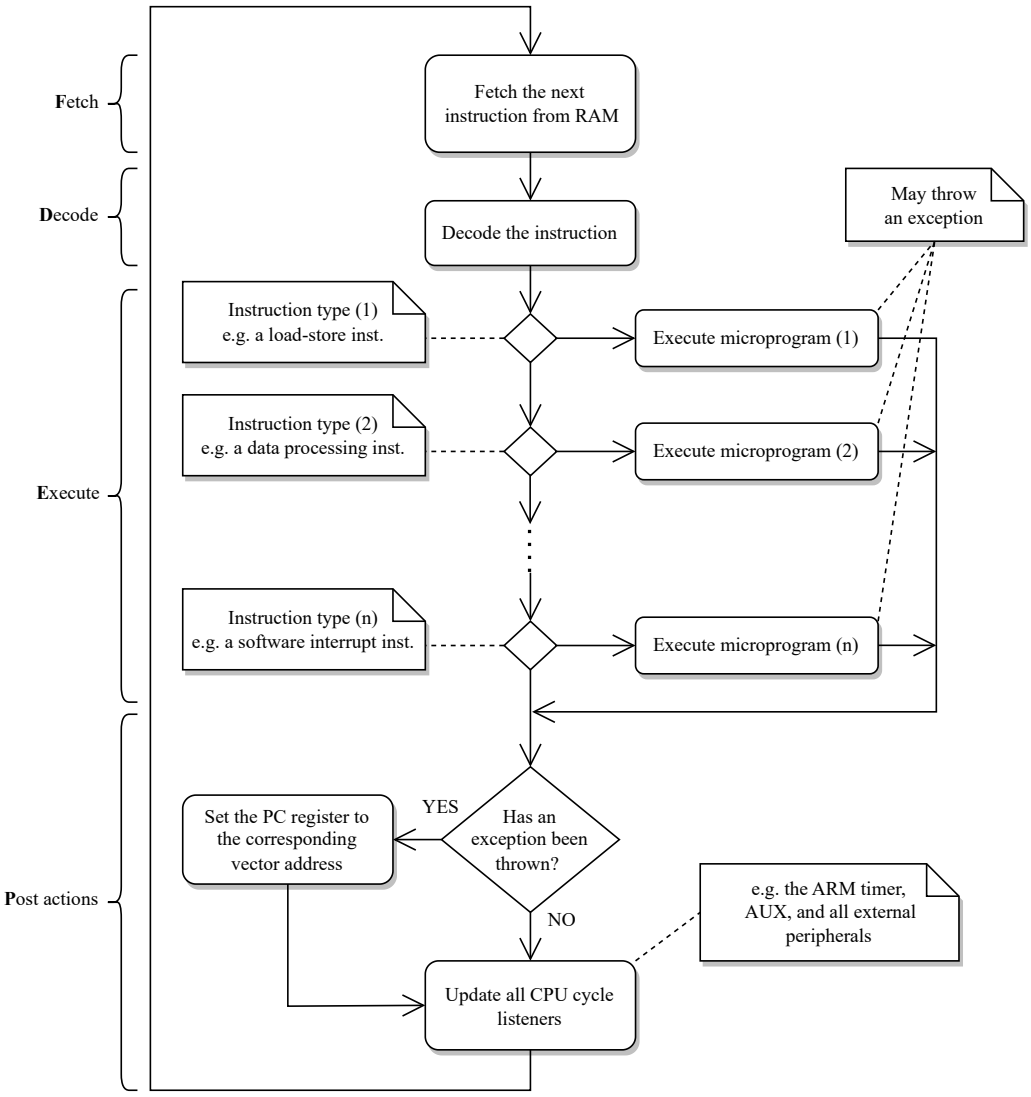

Figure 4.37: Execution loop of the emulated CPU

As shown in figure 4.37, the microprograms are called within an infinite loop known as the execution loop, which fetches the next instruction from RAM, decodes it using the ISA decoder, and directs the execution to the appropriate microprogram responsible for carrying out the necessary steps to execute it. These steps

may include tasks such as reading data from the stack, modifying register contents, utilizing the arithmetic-logic unit, and more.

It is worth noting that a real system is significantly more complex than how ZeroMate implements it. However, the author would contend that any form of emulation involves trade-offs that result in the omission of certain details.

Catching Exceptions.

As described in section 4.3.4.3, the CPU must handle any exceptions that may arise during the execution of the current instruction. When an exception occurs, the CPU switches to the corresponding mode, saves the return address in the link register, as if it were invoking a function call, and sets the address of the next instruction to the value stored at the corresponding offset in the interrupt vector table.

Updating CPU clock listeners.

As discussed in section 4.3.3.2, prior to advancing to the next instruction, the CPU informs all peripherals subscribed as system clock listeners about the number of CPU cycles required to execute the previous instruction.

While it is true that each instruction may require a varying number of CPU cycles for execution, in the current implementation, ZeroMate empirically averages this number to 8, which presents a potential area for future improvement. Users should be aware that the emulation speed does not match the speed of real hardware, and as a result, they may need to adjust timings in their firmware accordingly.

4.3.5 Coprocessors

ZeroMate takes into account two coprocessors, the design and functionality of which are described in the following two sections. Although their capabilities may be somewhat limited compared to what a real system offers, they serve the purpose of demonstrating the fundamental principles of interactions that can also be applied in practice.

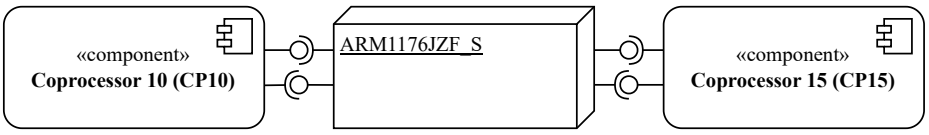

Figure 4.38: Component diagram of the CPU and its coprocessors

The CPU communicates with individual coprocessors through three types of instructions: *data transfer*, *register transfer*, and *data operation*. When the emulated

CPU detects a coprocessor instruction, it does not perform any further analysis but promptly delegates it to the appropriate coprocessor, determined by the ID encoded within the instruction itself. The coprocessor then internally performs decoding and execution of the instruction using technique similar to that used by the emulated CPU.

Although the ZeroMate emulator currently accommodates only two coprocessors, its overall design allows for a seamless integration of additional ones in the future, should the need arise. As showin in figure 4.39, the CPU maintains a collection of available coprocessors, all of which implement the same interface, through which they are controlled by the CPU.

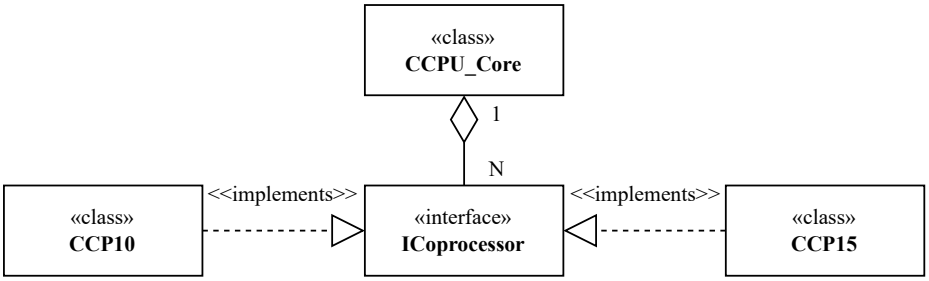

Figure 4.39: Coprocessor hierarchy

4.3.5.1 Coprocessor 15

The system control coprocessor (CP15) comprises a tree-like hierarchy of 32-bit registers that are used to enable a variety of additional features. These features include caching policy, branch prediction, unaligned memory access, setting up access to other coprocessors, and many more [21]. As far as ZeroMate is concerned, its emulation efforts are directed primarily towards the following registers.

Table 4.1: List of emulated CP15 registers

Primary register	Secondary register	Index	Description
C1	C0	0	Control Register (see table 4.2)
		2	Coprocessor Access Control Register
C2	C0	0	Translation Table Base Register 0 (TBBR_0)

(table continues on the next page)

[21] The list of all registers along with their functions can be found detailed in chapter 3 of the ARM1176JZF_S Technical Reference Manual [19].

Table 4.1 *(continued from the previous page)*

Primary register	Secondary register	Index	Description
		1	Translation Table Base Register 1 (TBBR_1)
		2	Translation Table Base Control
C8	C7	0	Invalidate unified TLB unlocked entries

Other frequently used CP15 registers are implemented for the purpose of completeness, even though modifying them has no effect. If a user's firmware attempts to write to an unimplemented register, a warning message will be displayed, indicating this specific functionality is beyond the emulator's capabilities.

Table 4.2: List of emulated flags of the CP15 control registers

Bit position	Description
0	Enables the MMU
13	Determines the location of exception vectors (0x00000000 vs 0xFFFF0000)
22	Enables unaligned data access operations

From an emulation perspective, CP15 serves as an organized repository of information that is queried from within other components of the application, so they can, if required, carry out their designated actions accordingly.

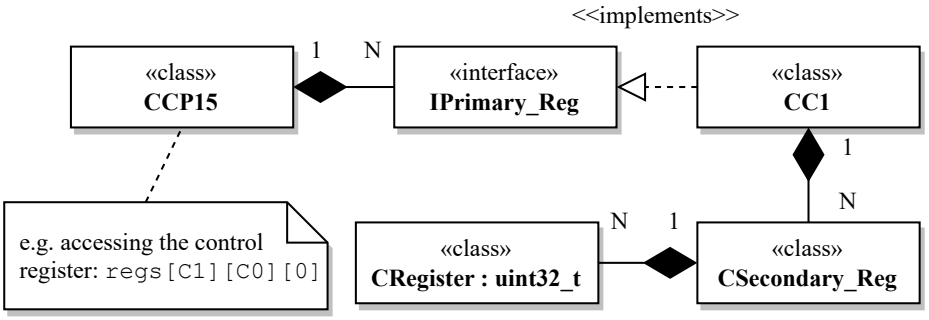

Figure 4.40: Design of the coprocessor 15 register hierarchy (uses primary register C1 as an example)

4.3.5.2 Coprocessor 10

Coprocessor 10 allows users to work with single-precision (32-bit) floating-point numbers. The design of this coprocessor closely resembles that of the CPU, as illustrated in figure 4.37. In this design, each instruction is first decoded and consequently executed within a private member function, which can be considered a microprogram.

While ZeroMate supports most common vector floating-point instructions of version 2 (VFPv2 [a]), including addition, subtraction, multiplication, division, and square root, it simplifies its implementation by omitting support for various rounding modes and floating-point exceptions.

[a] All VFP instruction can be found listed in chapter C3 of the ARM1176JZF_S Technical Reference Manual [19].

In terms of design, the floating-point unit, commonly denoted as the FPU, consists of an array of 32 internal registers, each 32 bits in size. Despite representing floating-point numbers, their underlining data type remains `uint32_t`, enabling seamless data exchange between the CPU and the FPU.

However, it is essential to emphasize that all operations must be executed as floating-point operations. As a result, the `32-bit` number can be encapsulated within a class that overrides its math operators, effectively hiding implementation details from the caller. Whenever an operation needs to be carried out, the register internally performs the following steps.

1. For all operands, convert the raw `32-bit` value into a `float` using the `IEEE 754` floating-point representation [20]. This conversion can be accomplished through the helper member function shown in source code 4.8.
2. Carry out the floating-point operation.
3. Store the result back as a `uint32_t` using the same technique demonstrated in source code 4.8.

Source code 4.8: Conversion between `uint32_t` and `float`

```
1  template<typename Type>
2  [[nodiscard]] Type Get_Value_As() const
3  {
4    return std::bit_cast<Type>(m_value); // since C++20
5  }
```

4.4 External Peripherals

All the modules described previously form the core of the emulator, which is compiled as a standalone application. Furthermore, the **ZeroMate emulator offers a public single-header interface that enables the implementation of third-party external peripherals**, which can be compiled independently of the toolchain used for the core itself. In other words, users can choose their preferred programming language for developing an external peripheral, with the condition that they implement the interface and compile it as a shared library [22].

Using a configuration file, these external peripherals can be then loaded by the core at runtime, enabling the user to create a custom circuit of external peripherals. Compiling as a shared library offers the advantage of creating multiple instances of the same peripheral. For instance, the user can employ multiple instances of `led.dll` to assemble a traffic light system, which can then be controlled by their custom firmware.

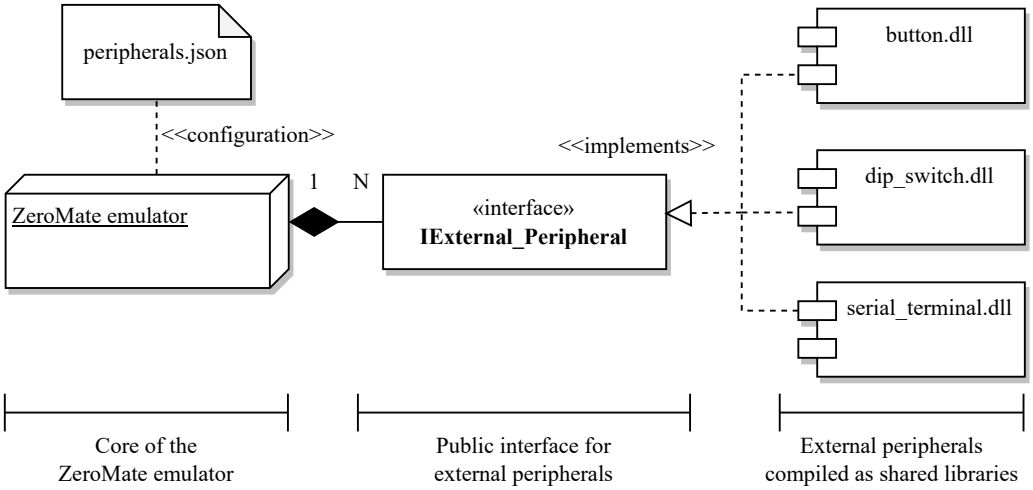

Figure 4.41: External peripheral interface

4.4.1 External Peripheral Interface

Upon construction, **every external peripheral is given access to the `GPIO` pins**, as it is their primary way of interaction with the core of the emulator. The interface also mandates that they implement a `Get_GPIO_Subscription` function, responsible for returning a set of `GPIO` pins they wish to subscribe to [23]. Whenever the state

[22] The format of a shared library is `.dll` on Windows and `.so` on Linux, respectively.

[23] ZeroMate allows for connecting multiple external peripherals to the same `GPIO` pins, effectively creating a parallel connection.

of a GPIO changes, the emulator iterates through all external peripherals, examining their GPIO subscriptions, and duly informing them of the change. Additionally, they receive information about the number of CPU cycles it took to execute the last instruction, the same way internal peripherals do, effectively serving as an emulated replacement for an independent system clock.

> Certain external peripherals, such as a serial terminal, come with their own system clock. However, in ZeroMate, all timing functions derived from the emulated CPU frequency.

Optionally, **they are provided access to the logging system**, which proves to be invaluable for debugging or providing insights into their current state. External peripherals are not obliged to implement a graphical user interface; nevertheless, they are implicitly provided with a context that they can utilize to render themselves within each frame.

4.4.2 Configuration

Upon startup, ZeroMate attempts to read a single configuration file in the JSON format that details the connections of external peripherals. Table 4.3 outlines the obligatory fields that must be provided for every peripheral. ZeroMate handles the construction and management of all external peripherals within its dedicated address space on the host machine.

Table 4.3: Information stored in `peripherals.json`

Field	Example	Description
`name`	`"7-segment Display"`	Unique name associated with the peripheral
`connection`	`[ 2, 3, 4 ]`	Set of GPIO pins the peripheral is connected to
`lib_dir`	`"peripherals"`	Directory where the shared library is held
`lib_name`	`"seven_seg_display"`	Name of the shared library

Figure 4.42 provides a visual representation of a custom connection that connects two 7-segment displays in parallel, having them simultaneously display the same piece of information.

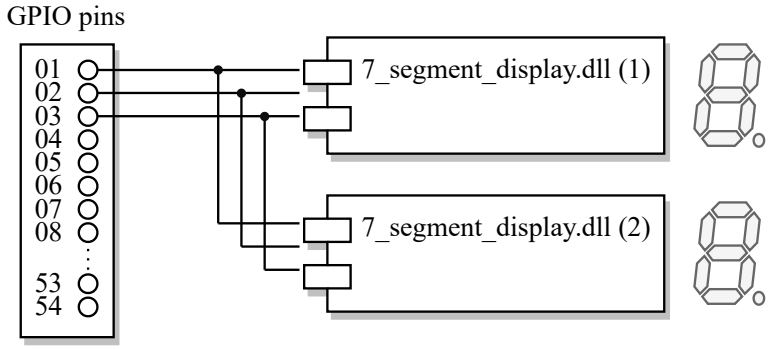

Figure 4.42: Illustration of connection of two parallel 7-segment displays

4.4.3 Examples of External Peripherals

ZeroMate is shipped with several external peripherals out of the box, most of which mirror the ones found on the `DPP-01` board [21], which was designed for educational purposes within the `KIV/OS` class at the University of West Bohemia.

Table 4.4: List of emulated external `DPP-01` peripherals

Peripheral	Additional information
`button.dll`	-
`dip_switch.dll`	-
`led.dll`	Features multiple color support
`7_seg_display.dll`	7-segment display controlled via a shift register
`ssd1306_oled.dll`	`OLED` display controlled via the I^2C protocol

Table 4.5: List of additional external peripherals

Peripheral	Additional information
`serial_terminal.dll`	Used for the `UART` communication
`logic_analyzer.dll`	Visualization of `GPIO` pins' state over time

4.4.3.1 Serial Terminal

The serial terminal serves as the **counterpart in full-duplex `UART` communication**, enabling the user to communicate externally with the firmware running in the emulator. In practice, this is exemplified by programs such as *PuTTY* [15], which establish communication with the development board through the host's operating system. From a design perspective, it employs the same algorithmic techniques as those described in section 4.3.3.9.

4.4.3.2 Logic Analyzer

The logic analyzer works as a **read-only observer of the current state of the `GPIO` pins** it is connected to. It graphically represents their state over time, allowing the user to visually debug individual frames of various low-level communication protocols, such `UART` or `I`^{2}`C`.

Depending on the specific use-case, the logic analyzer can either periodically sample the state of the `GPIO` pins [24], or it can sample them whenever there is a change in the state of any of the monitored pins. This feature can be especially valuable when observing synchronous types of communications.

4.5 Logging System

The ZeroMate project implements a custom logging system, which can be utilized both within the emulator's core and by the external peripherals. From a design perspective, the logging system component serves as a central access point, acting as a proxy to forward log messages to individual endpoint loggers. This design enables the implementation of multiple loggers for various purposes, such as logging to a file, console, or graphical user interface.

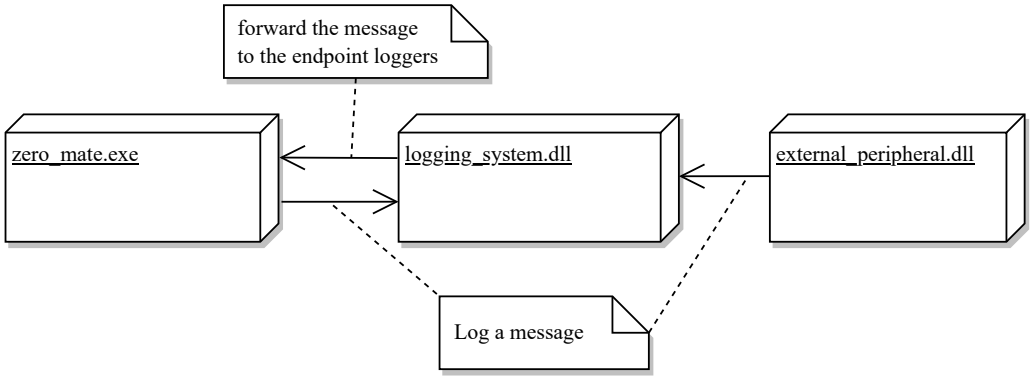

Figure 4.43: Utilization of the logging system throughout the project

[24] The frequency is derived from the emulated CPU clock.

The logging system is compiled separately as a shared library, making it usable across all project targets. When it receives a message, it applies a uniform formatting and forwards it to all endpoint loggers.

The `ILogger` interface requires that all endpoint loggers implement the *Debug*, *Info*, *Warning*, and *Error* callback functions, which are commonly found in the majority of logging systems. Moreover, each endpoint logger can have a different logging level [25], allowing them to filter out messages that do not meet their specific criteria.

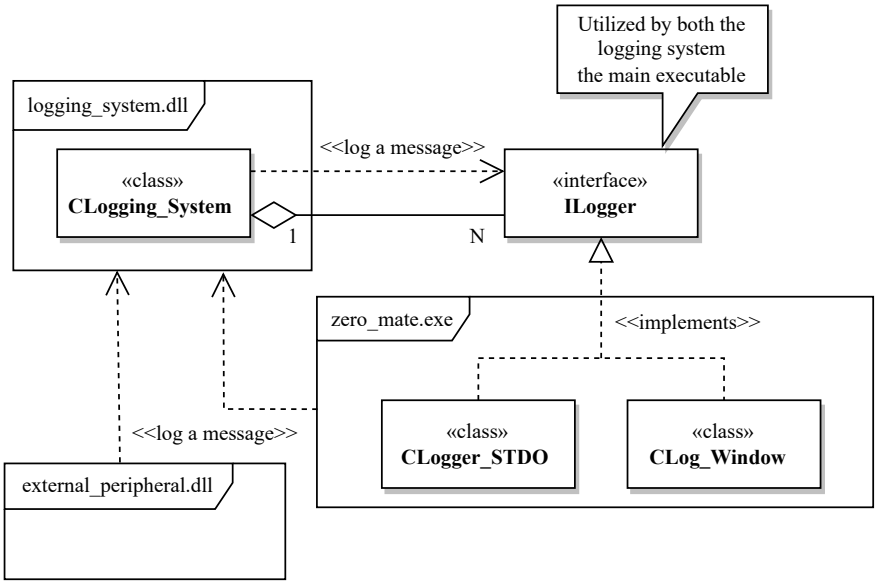

Figure 4.44: Hierarchy of endpoint loggers utilized by the logging system

4.6 User Interface

ZeroMate places a strong emphasis on the visual aspects of emulation, aiming to assist the user in debugging and troubleshooting potential bugs or issues. It serves as a frontier interface, through which the user can interact with the emulator and view the current system's state. ZeroMate consists of a set of hierarchically structured windows, each displaying information about distinct aspects of the emulator's core.

Figure 4.45 shows the architectural structure of all `GUI` windows, with each window being periodically rendered within each frame.

[25] For instance, when the logging level is set to *Warning*, the loger will only accept messages classified as *Warning* or *Error*.

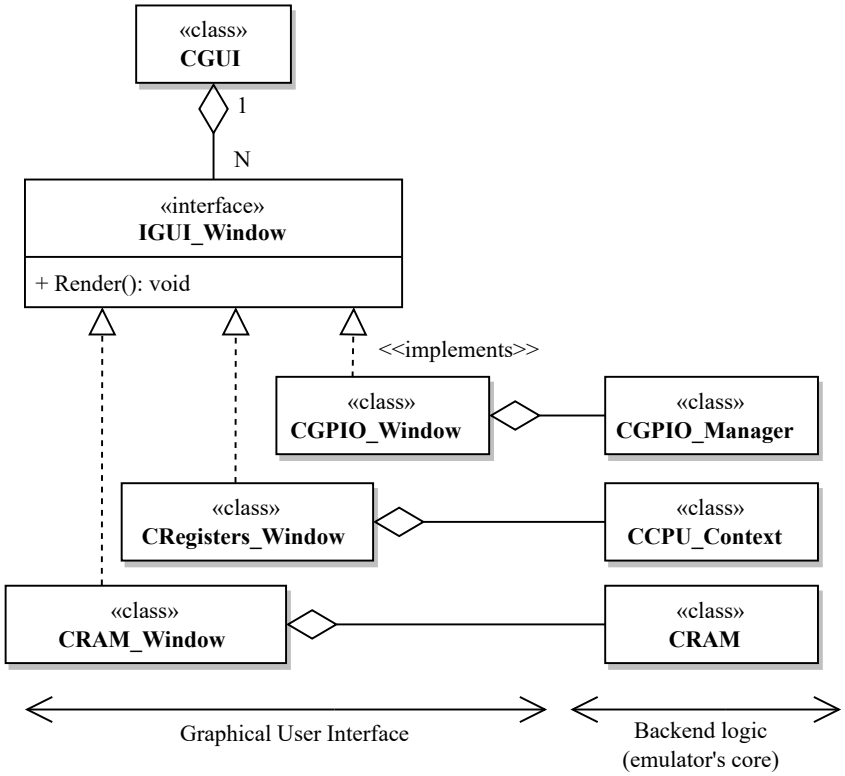

Figure 4.45: Structure of GUI windows

From figure 4.45, it can be observed that every class windows has access to the part of the emulator's core whose state is supposed to visualize. This imposes a one-direction dependency, allowing the core to be potentially used with different GUI libraries in the future, should the needed arise.

In general, all windows can be categorized into three main types: *component windows*, *utility windows*, and the *primary source code window*. The component windows are designed to display the current state of specific components, such as the CPU context, RAM, CPU registers, coprocessors, and others. Utility windows aim to enhance the user experience. This category includes windows like the logging window, the top bar menu, and the control window, which enable users to control the state of execution. Lastly, the primary source code window is arguably of utmost importance, as it presents the disassembled source code organized into different functions. It provides the user with information about the current line of execution and it allows them to set breakpoints at their desired locations.

Further design as well as implementation details regarding both the core and GUI components can be found in the documentation generated from the source code. This documentation is accessible online through GitHub Pages [22].

Development Process 5

5.1 Technologies

Testing 6

User Manual 7

Conclusion 8

Bibliography

1. JAMIL, T. *RISC versus CISC*. Institute of Electrical and Electronics Engineers, 1995. Available also from: https://ieeexplore.ieee.org/abstract/document/464688.
2. A., Patterson David. *Computer Organization and Design Arm Edition: The Hardware Software Interface*. 2016. ISBN 978-0128017333.
3. TEAM, Arm Editorial. *The Official History of Arm*. 2023. Available also from: https://newsroom.arm.com/arm-official-history.
4. *ARM Architecture Reference Manual*. ARM. Available also from: https://documentation-service.arm.com/static/5f8dacc8f86e16515cdb865a?token=.
5. ARM. *Procedure Call Standard for the Arm® Architecture*. 2023. Available also from: https://github.com/ARM-software/abi-aa/releases/download/2023Q3/aapcs32.pdf.
6. COMMITTEE, TIS. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*. Linux Foundation, 1995. Available also from: https://refspecs.linuxfoundation.org/elf/elf.pdf.
7. BROADCOM. *BCM2835 ARM Peripherals*. Broadcom Corporation, 2012. Available also from: https://datasheets.raspberrypi.com/bcm2835/bcm2835-peripherals.pdf.
8. COPLIEN, James O. *Multi-Paradigm Design for C++*. Taschen, 1998. ISBN 978-0201824674.
9. BALOGUN, Ghaniyyat Bolanle. *A Comparative Analysis of the Efficiencies of Binary and Linear Search Algorithms*. Department of Computer Science, University of Ilorin, Ilorin, Nigeria., 2020. Available also from: https://afrjcict.net/wp-content/uploads/2020/03/Vol13No1Mar20pap3journalformatpagenumb.pdf.
10. KANKOWSKI, Peter. *x86 Machine Code Statistics*. 2008. Available also from: https://www.strchr.com/x86_machine_code_statistics.

11. *The GNU C++ Library - Chapter 28. Demangling*. The GNU Compiler Collection, 2008. Available also from: https://gcc.gnu.org/onlinedocs/libstdc++/manual/ext_demangling.html.

12. LAMIKHOV-CENTER, Serge. *ELFIO - ELF (Executable and Linkable Format) reader and producer implemented as a header-only C++ library*. Available also from: https://elfio.sourceforge.net.

13. WEBER, Nico. *Demumble - A better c++filt and a better undname.exe, in one binary*. Available also from: https://github.com/nico/demumble.

14. *BCM2835 datasheet errata*. Embedded Linux Wiki. Available also from: https://elinux.org/BCM2835_datasheet_errata.

15. TATHAM, Simon. *PuTTY - free implementation of SSH and Telnet for Windows and Unix platforms, along with an xterm terminal emulator*. Available also from: https://www.putty.org.

16. TORVALDS, Linus. *Linux kernel*. Available also from: https://github.com/torvalds/linux/blob/master/drivers/char/hw_random/bcm2835-rng.c.

17. BHATTACHARJEE, Kamalika; DAS, Sukanta. *A search for good pseudo-random number generators: Survey and empirical studies*. 2022. Available also from: https://www.sciencedirect.com/science/article/pii/S1574013722000144.

18. SLOSS, Andrew; WRIGHT, Chris. *ARM and Thumb Instruction Encodings (B2 APPENDIX)*. ARM. Available also from: https://gab.wallawalla.edu/~curt.nelson/cptr380/textbook/advanced%20material/Appendix_B2.pdf.

19. *ARM1176JZF-S Technical Reference Manual*. ARM, 2009. Available also from: https://developer.arm.com/documentation/ddi0301/h.

20. *754-2019 - IEEE Standard for Floating-Point Arithmetic*. Institute of Electrical and Electronics Engineers, 2019. Available also from: https://ieeexplore.ieee.org/document/8766229.

21. ÚBL, Martin. *KIV-DPP-01 - Expansion board documentation*. University of West Bohemia, 2021. Available also from: https://home.zcu.cz/~ublm/files/os/kiv-dpp-01-en.pdf.

22. ŠILHAVÝ, Jakub. *ZeroMate Doxygen documentation*. 2023. Available also from: https://silhavyj.github.io/ZeroMate.

List of Abbreviations

SIMD - Single Instruction/Multiple Data
IoT - Internet of Things
ARM - Advanced RISC Machines
RAM - Random Access Memory
ROM - Read-Only Memory
ELF - Executable Linkage Format
GPU - Graphics Processing Unit
SREC - Motorola S-record (file format)
GUI - Graphical User Interface
EEPROM - Electrically Erasable Programmable Read-only Memory
CPU - Central Processing Unit
ALU - Arithmetic-Logic Unit
MAC - MAC Unit (performs multiply-accumulate operations)
MMU - Memory Management Unit
ISA - Instruction Set Architecture
MMIO - Memory-mapped Input-Output device
I/O - Input-Output
R/W - Read-Write
ABI - Application Binary Interface
RTTI - Run-Time Type Information
GPIO - General Purpose Input/Output
LED - Light-emitting diode
SD - Secure Digital
IVT - Interrupt Vector table
QA - Quality Assurance
UART - Universal Asynchronous Receiver/Transmitter
TRNG - True Random Number Generator
LCG - Linear Congruential Generator
OS - Operating System
I^2C - Inter-Integrated Circuit
CPI - Cycles Per Instruction

`IC` - Interrupt Controller
`PWM` - Pulse Width Modulation
`IRQ` - Interrupt Request
`FIQ` - Fast Interrupt Request
`AUX` - Auxiliary
`MSB` - Most Significant Bit
`LSB` - Least Significant Bit
`TX` - Transmit
`RX` - Receive
`BSC` - Broadcom Serial Controller
`CPSR` - Current Program Status Register
`SPSR` - Saved Program Status Register
`TLB` - Translation Lookaside Buffer
`TTBR` - Translation Table Base Register
`CP10` - Coprocessor 10
`CP15` - Coprocessor 15
`VFP` - Vector Floating-Point
`FPU` - Floating-Point Unit
`OLED` - Organic Light-Emitting Diode
`STDO` - Standard Output
`SP` - Stack Pointer
`RISC` - Reduced Instruction Set Computer
`CISC` - Complex Instruction Set Computer
`ACC` - Accumulator
`LIFO` - Last-In-First-Out
`FPGA` - Field Programmable Gate Array

List of Figures

List of Tables

List of Listings

Attachments 9