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

2024

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 2024

Citation in the bibliography/reference list:
ŠILHAVÝ, Jakub. *ARMv6 Processor Emulator for Raspberry Pi Environment Emulation*. Pilsen, Czech Republic, 2024. 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**

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 14 May 2024

. . . . . .

Jakub Šilhavý

Abstract

This paper examines the potential of emulating 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 devices 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 se zabývá zkoumáním potenciálu emulace Raspberry Pi Zero, který reprezentuje jednu z nejrozšířenějších architektur pro vestavěné systémy - architekturu ARM. Úvodní kapitoly přinášejí obecné seznámení s ARM architekturou, jejíž významnost je demonstrována více než miliardou elektronických zařízení, které ji využívají. Dále se práce zaměřuje na výhody využívání ARM emulátoru, stanovení celkových požadavků a analýzu existujících možností 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 Zero

Contents

1 Introduction

Over the past several years, ARM [1], as a computer architecture, has garnered popularity across a wide array of applications. Its usage spans from low-power solutions and affordable microcontrollers to real-time applications and safety-critical systems that include medical devices, automotive technology, and aviation. Furthermore, ARM is extensively utilized in personal computers and the cell phone industry. Presently, it is estimated that ARM powers over 99% of the world's smartphones [1]. Covering such a diverse range of applications, ARM has tailored its processor cores into various groups known as families. Notable among these is ARMv6, employed in devices like Raspberry Pi Zero. This version introduced innovative technologies such as *TrustZone*, *Jazelle*, and *Single Instruction/Multiple Data* instructions to the Classic line of ARM processor cores.

Emulating such a widely adopted architecture can assist in illustrating concepts of computer organization and operating system principles. In addition to educational purposes, the use of an ARM emulator can prove beneficial in the software development process, particularly when immediate access to a development board may not be feasible. Furthermore, it provides a safety net by enabling developers to experiment with potentially risky code without concerns about damaging real hardware. As a software tool, it could also be integrated into continuous-integration automated testing to identify potential bugs before moving on to testing on actual hardware, which may not be readily available.

This thesis delves into the fundamental principals of the ARM architecture, exploring and evaluating existing emulation solutions tailored for an embedded environment. The primary goal is to design, implement, and rigorously test a comprehensive and extensible Raspberry Pi Zero emulator capable of emulating KIV-RTOS [2] — a real-time operating system developed for educational purposes at the University of West Bohemia.

[1] The term ARM is commonly recognized as an acronym, initially representing *Acorn RISC Machine* before being redefined as *Advanced RISC Machine*.

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 widely recognized may include Intel, MIPS, ARM, or the increasingly popular RISC-V architecture [3].

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, the `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 support, how they organize memory, what register set they feature [1], or the number of operands expected in an instruction [4]. 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 ideologies along with their pros and cons.

2.1.1 Basic Classification

As far as memory access is concerned, there are two major architectural ideas that serve as the foundation for nearly all computer architectures: the Von Neumann

[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.

architecture and the Harvard architecture. While they are distinguished by various factors, including their cost, intended usage environment, and more, their primary distinction lies in how they organize and access memory [3].

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 [5]. 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 system crash. On the other hand, its design is considerably cheaper compared to the Harvard architecture, as it requires less physical space on the chip.

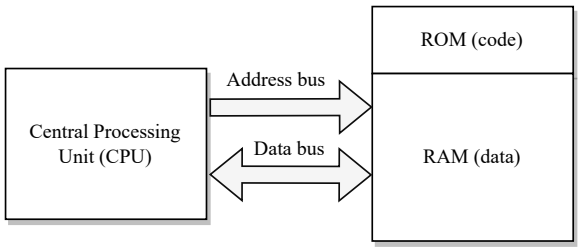


Figure 2.1: Von Neumann architecture

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

As shown in Figure 2.2, the Harvard architecture employs **distinct memory modules for code and data**, requiring twice the number of bus lines compared to the Von Neumann architecture [3]. 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.

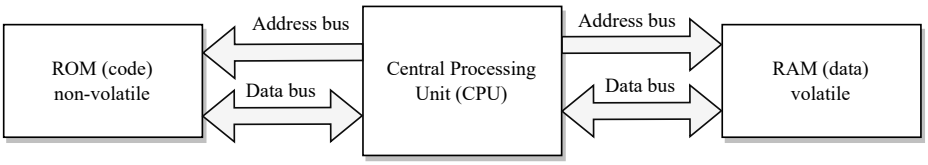


Figure 2.2: Harvard architecture

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

2.1.2 Instruction Set Architectures

Instruction set architectures can also by categorized by how they handle operands in terms of their interaction 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 the main 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 first popped off the stack, and the operation's result is consequently pushed back onto it [4].

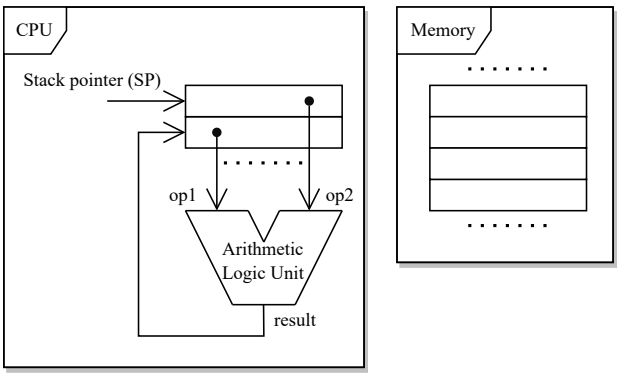


Figure 2.3: Stack architecture

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 itself, which introduces a bottleneck that hinders any potential parallelization. This architectural design is employed, for example, by the Java Virtual Machine.

2.1.2.2 Accumulator Architecture

Similar to the stack architecture, the accumulator architecture, illustrated in Figure 2.4, imposes minimal hardware requirements. It substitutes the stack with a single register called the accumulator, which serves both as an input operand and as storage for the operation's result. Same as the previous architecture, the accumulator represents a potential bottleneck. Furthermore, it leads to high memory

traffic, as every two-operand instruction requires the second operand to be retrieved from memory. Some processor architectures are equipped with more than one accumulator. For instance, the MOS Technology 6502 microprocessor utilized in the Apple II computer featured not only a primary accumulator but also two index registers that functioned as additional accumulators [4].

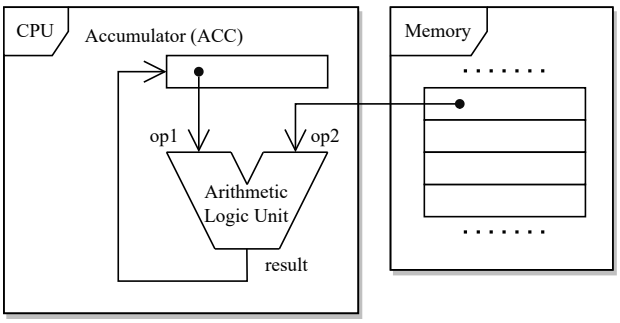


Figure 2.4: Accumulator architecture

2.1.2.3 Load-Store Architecture

Examining Figure 2.5, it can be observed that 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**, which serve as an interface for reading and writing data to memory. As a result, 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, or ALU, to proceed with the operation [4].

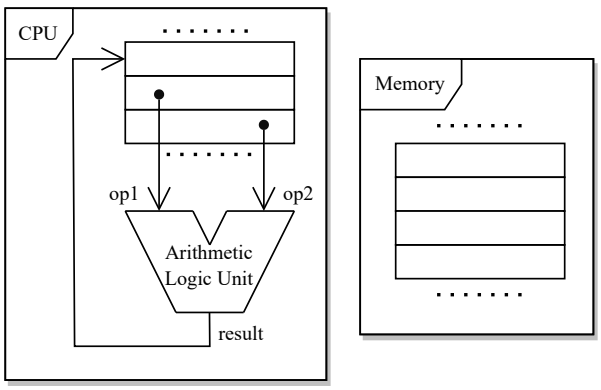


Figure 2.5: Load-store architecture

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

Load-store architectures frequently use fixed-length instructions, providing the potential for more efficient pipelining [2], which can result in improved performance [6]. 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.

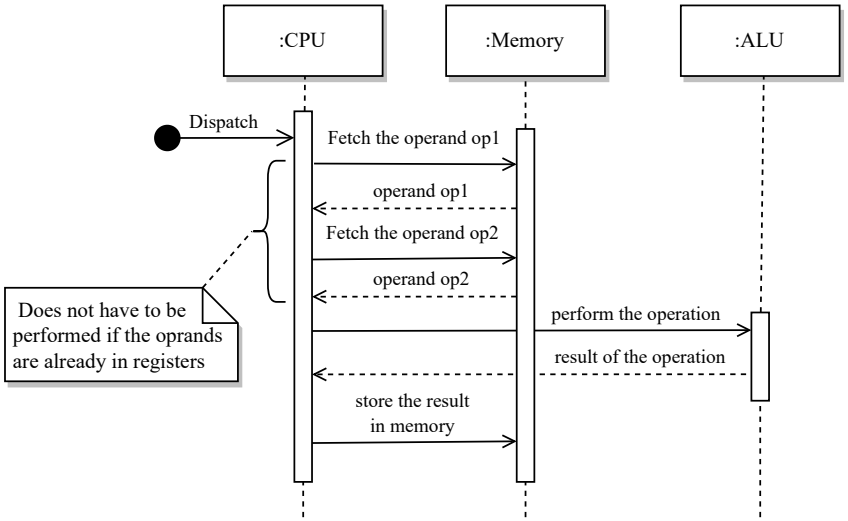


Figure 2.6: Load-store sequence diagram

2.1.3 Instruction Set Classification

`RISC`, which stands for *Reduced Instruction Set Computer*, and `CISC`, or *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 [7].

2.1.3.1 Complex Instruction Set Computer

The `CISC` architecture places an **emphasis on the underlying hardware**, leading to the development of complex, usually 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 results 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.

[2] 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. As depicted in Figure 2.7 below, 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 due to the fact that, in RISC, achieving a specific function might involve using multiple instructions, whereas the CISC architecture usually accomplishes the same goal with a single instruction. An example of this architecture is the ARM architecture, which is described more in detail in Chapter 3.

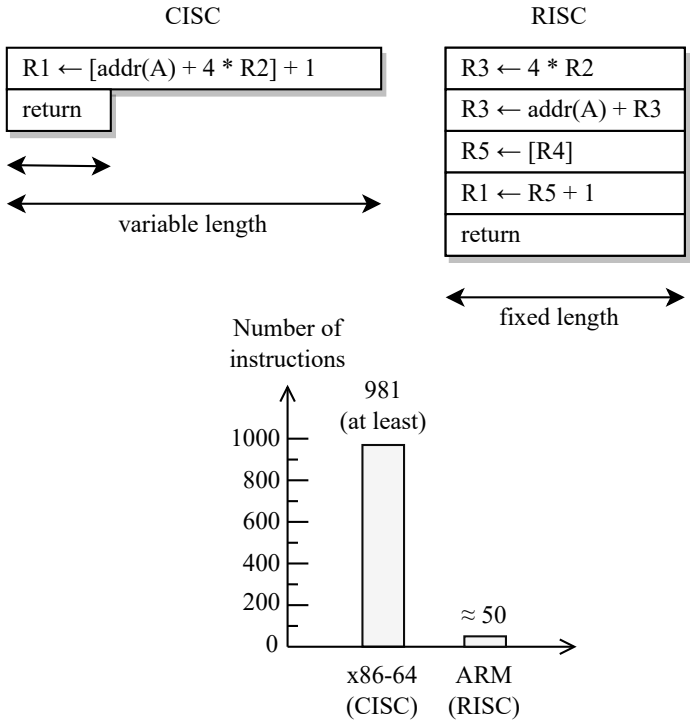


Figure 2.7: Comparison between the RISC and CISC architectures [8]

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, the reader can refer, for instance, to the book series *Computer Organization and Design: The Hardware/Software Interface* [9].

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 [1].

Throughout the 2000s, ARM's sustained success enabled it to evolve beyond smartphones and become **arguably the most widely used processor architecture**. Nowadays, as illustrated in Figure 3.1, 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 [1].

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.

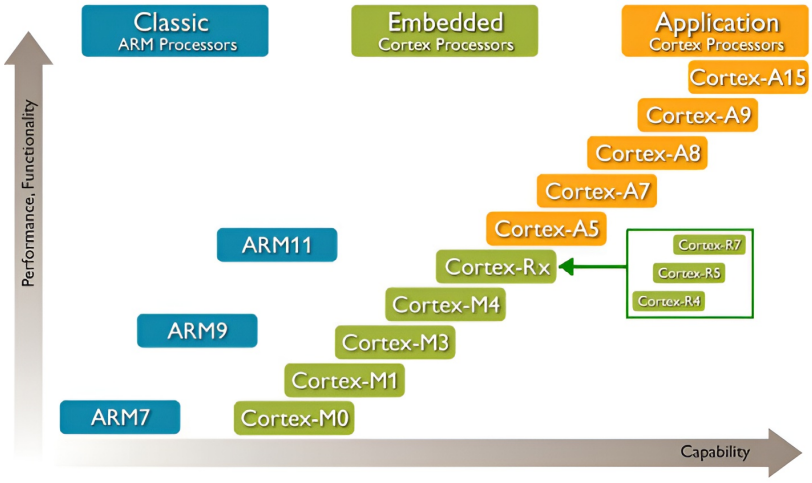


Figure 3.2: ARM processor roadmap

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 [10]. 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, or `IoT`, devices such as home automation systems, wearables[1], or smart locks.

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

Usually, the ARM Cortex-M family simplifies or modifies certain features, resulting in slight deviations from the traditional ARM architecture. These modifications may involve different 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, aerospace, defense, and real-time 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 to ensure a seamless human-machine interaction experience.

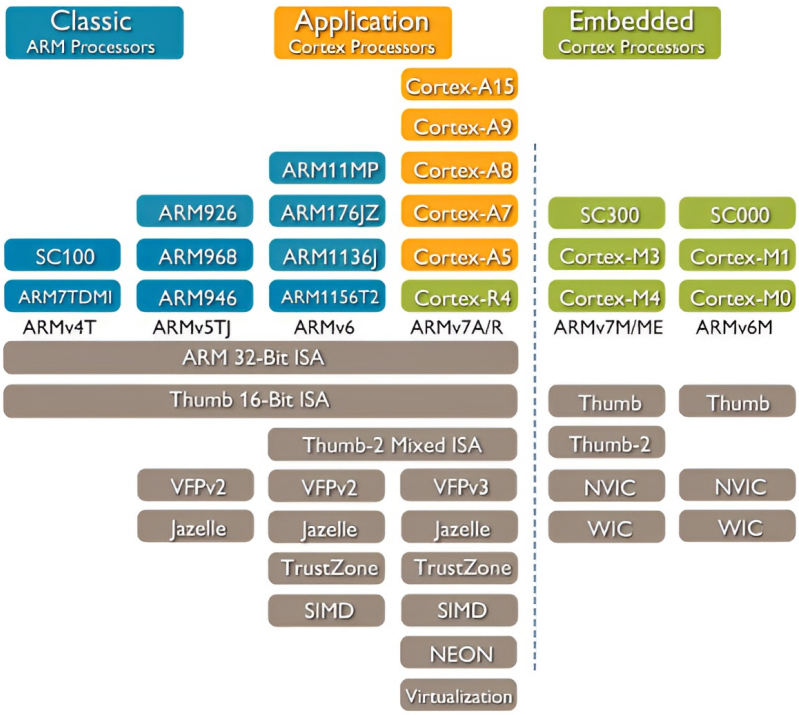


Figure 3.3: Features of different ARM processor cores

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

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

As mentioned in Section 2.1, **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 [11].

3.3.1 Central Processing Unit Modes

There are a total of **seven modes in which a modern 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 as intended 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 unprivileged mode, the execution of certain instructions might be restricted [2]. Table 3.1 summarizes all available CPU modes.

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 co-processors
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*, or PC for short, 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 redirects 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, or IVT, 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 Co-processor*. In practice, the lower part of the address space is usually reserved for user processes, while the kernel is remapped to the upper part, hence the option to relocate the IVT as well [3].

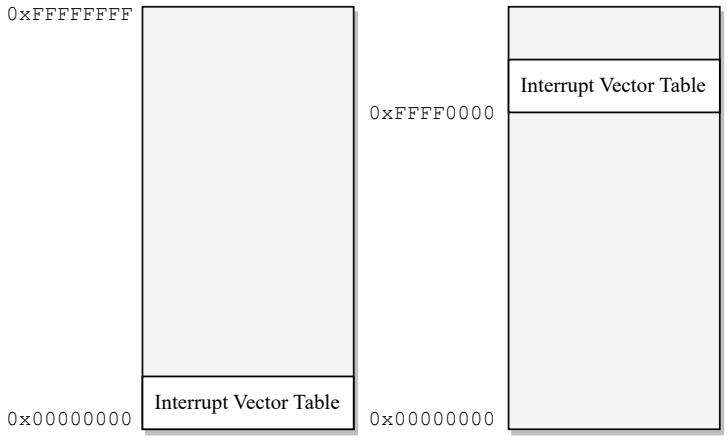


Figure 3.4: Possible locations of the interrupt vector table in RAM

Table 3.2 provides a list of addresses for individual vectors located in both the lower and upper portions of the address space.

[3] In the ARM architecture, the lower portion of the address space is typically controlled by *Translation Table 0*, with its address is stored in the TTBR0 register. Simultaneously, the kernel section is managed in parallel by *Translation Table 1*, whose address is located in the TTBR1 register. From the viewpoint of an operating system, the presence of two distinct page tables provides the advantage of avoiding the need to replicate kernel pages for each individual process created.

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`
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, specifically tailored for microcontroller applications.

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 [12].

Table 3.3: List of special function ARM registers

Index	Mnemonic	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, **the majority of CPU modes have their 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 displays distinct bank registers for each CPU mode. Notably, *User*, the unprivileged mode, and *System*, a privileged mode, utilize the same set of underlying registers, which serves as a "meeting point" between the kernel and user space when, for instance, running system tasks or handling system calls.

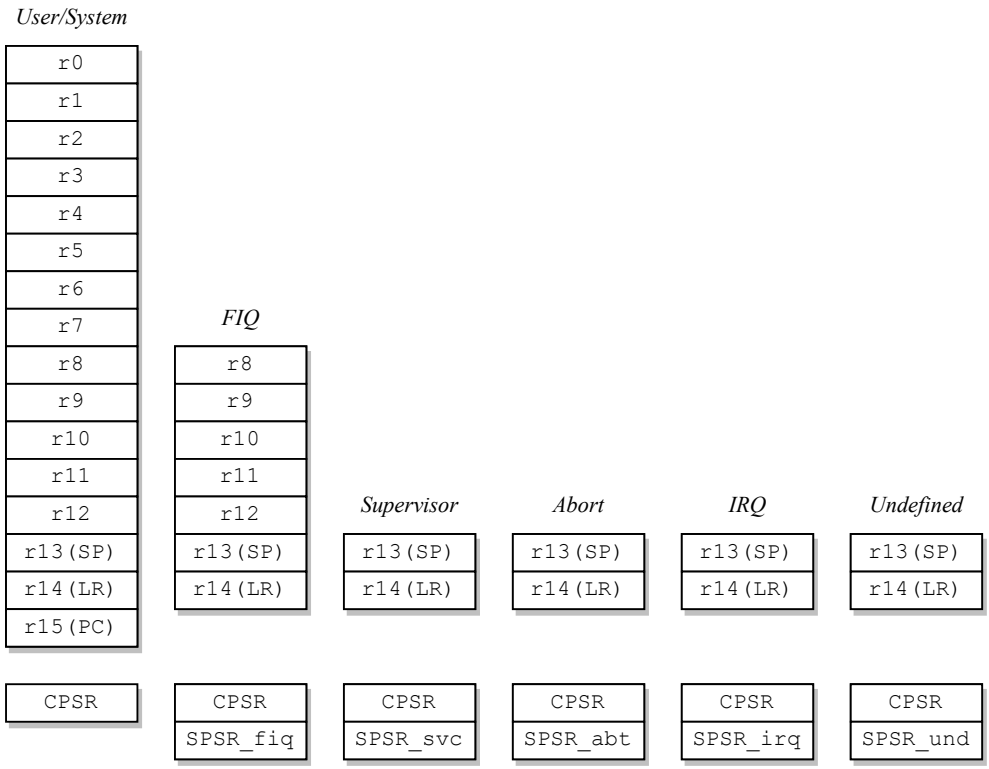


Figure 3.5: Bank registers of different CPU modes

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

In general, control registers are **specialized registers within a processor that are used to manage system configuration**, memory, interrupts, and power. They enable software to interact with and configure hardware features, such as system-wide settings and power-saving mechanisms.

Current Program Status Register

The CPSR register, which stands for *Current Program Status Register*, **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 its direct modification is not permitted. 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 (F, S, E, and C) can function as a bit mask when using the MRS or MSR instruction to prevent unintentional modifications of bits that are not meant to be changed.

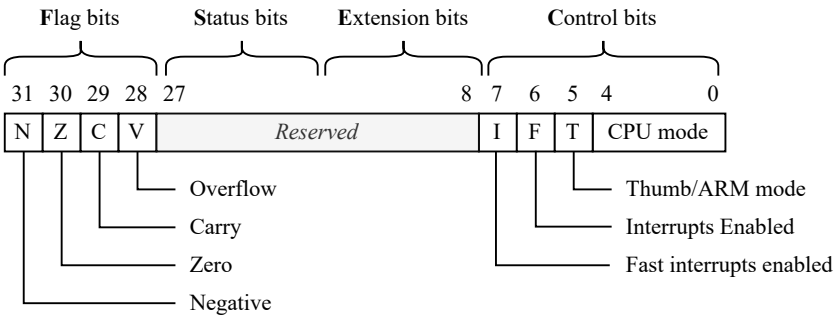


Figure 3.6: Current Program Status Register[4]

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, as shown in Figure 3.5, the contents of the CPSR register of the current mode are copied into the SPSR register of the target mode.

[4]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.

This process allows a future restoration of the original state when reverting to the original CPU mode.

3.3.4 ARM Instructions

As mentioned previously, **all instructions in the *ARM* instruction set are 32-bits in length**. As shown in Figure 3.7, the most significant four bits of each instruction form a condition filed that constrains its execution. The CPU assesses the current state of the 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 appending an `'S'` to the instruction name. For instance, the `ADD` instruction does not update the flags, whereas `ADDS` does [5].

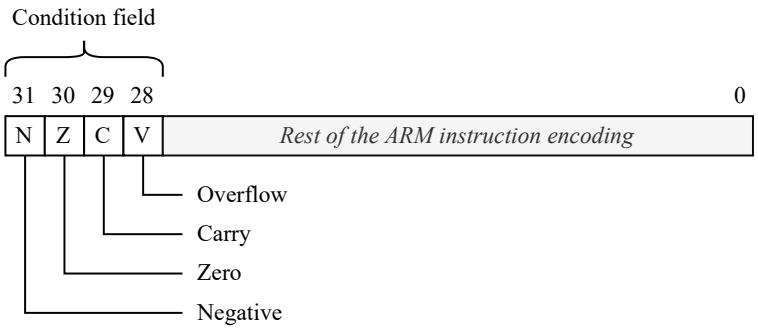


Figure 3.7: Condition field of an ARM instruction

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

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 *co-processor instructions*, which are designed for interacting with external CPU co-processors. Detailed ARMv6 instruction encodings can be found, for instance, in the B2 ARM Appendix document [13].

3.4 Co-processors

As showcased in Figure 3.8, a co-processor, as its name implies, **operates alongside the main CPU to further extend its functionality**. Examples of co-processors may include a floating-point unit or the system control co-processor. Each co-processor is addressed by a unique 4-bit number [6], resulting in up to 16 different co-processors that can be attached to an ARM CPU.

As listed in Table 3.5, the **CPU interacts with co-processors via three designated instructions**, enabling it to offload specific functions to more specialized hardware suitable for the given task, such as when multiplying two floating-point numbers.

[6]The co-processor ID is encoded in every co-processor instruction.

Table 3.5: List of ARM co-processor instructions

Instruction	Description
Coprocessor Data Operation (CDP)	Signals a co-processor to perform an internal operation
Coprocessor Data Transfer (LDC, STC)	Loads or stores a subset of co-processor's registers directly to memory
Coprocessor Register Transfer (MRC, MCR)	Communicates information directly between the CPU and a co-processor

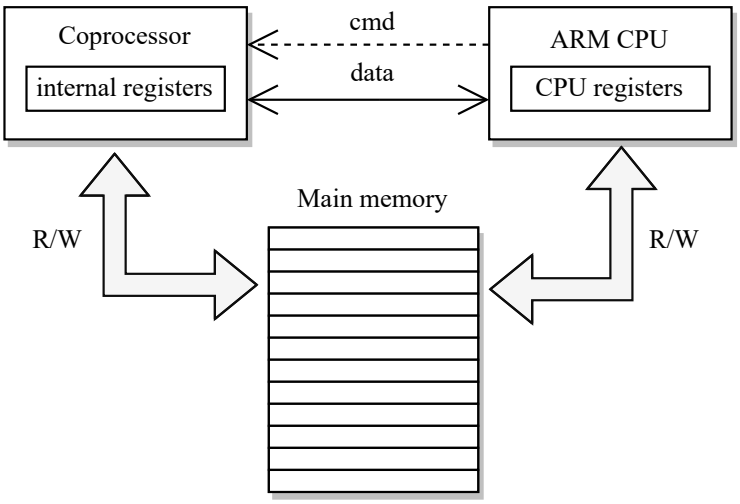


Figure 3.8: Context of an ARM co-processor

3.4.1 System Control Co-processor

The system control co-processor, or CP15, incorporates a tree-like structure of additional registers that are utilized for **configuring various settings within the overall system**. For instance, the users can configure branch prediction, caching, unaligned-memory access, paging, and so on. Moreover, via CP15, users can enable or disable other co-processors and define their CPU mode access rights.

3.4.2 Floating-point Unit

The floating-point unit, also referred to as the FPU, is represented by two distinct co-processors, CP10 (single-precision) and CP11 (double-precision). As illustrated in Figure 3.9, this distinction occurs solely at an interface level, meaning that they

share the same underlining set of registers, which are treated differently based on the leveraged co-processor. The primary purpose of the FPU is to extend the capability of an ARM CPU by **incorporating floating-point arithmetic instructions**, such as *addition*, *subtraction*, *multiplication*, *division*, and *square root* operations.

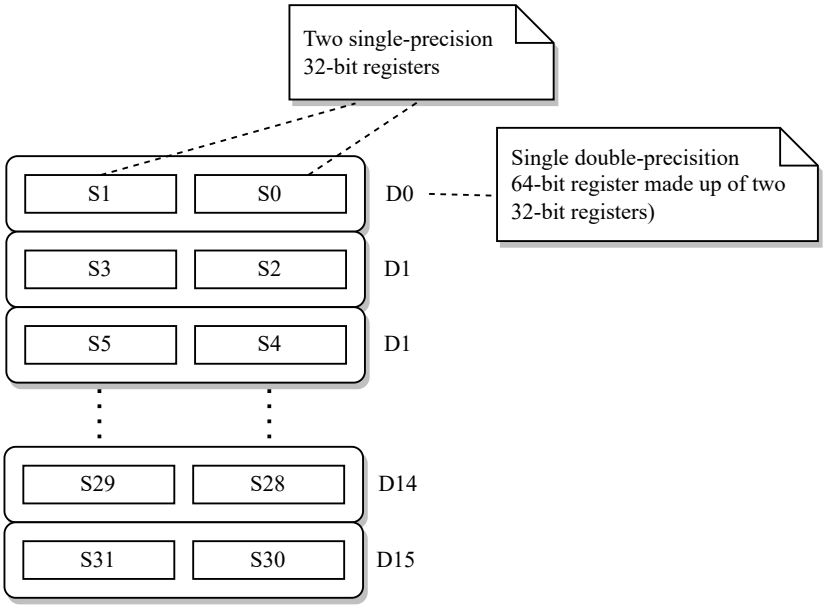


Figure 3.9: Floating-point registers

Additionally, the FPU also provides its own *Status Control Register*, called FPSCR, which contains the current state of flags. Whenever a comparison instruction is executed, it becomes essential to transfer the FPU flags to the CPU, so they can be used to determine the execution of the next instruction [7].

Finally, the user can optionally set different rounding modes and exceptions via the *Exception Control Register* named FPEXC. This register is also employed to activate the FPU. As a result, it must be initially enabled on a co-processor level (in CP15) and subsequently in the FPEXC register as well.

[7] From the programmer's perspective, this process is typically automated by the compiler, which implicitly inserts a subsequent MRC instruction.

Raspberry Pi Zero 4

Raspberry Pi Zero is a system on chip, or SoC, which is a design that integrates various components into a single chip, such as a central and graphics processing unit, memory interfaces, and input-output devices. As far as Raspberry Pi Zero is concerned, it is equipped with 512 `MB` of `RAM`, an `SD` card holder, which is utilized during the booting process, a Mini `HDMI` adapter, a micro `USB` port, and the `BCM2835` microcontroller, which is powered by the `ARM1176JZF_S` processor [14]. Figure 4.1 labels the main visual components of the board.

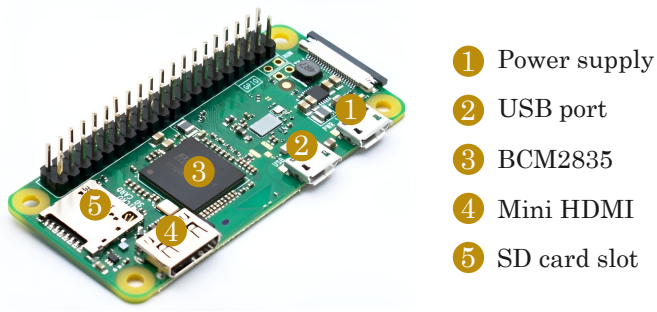


Figure 4.1: Raspberry Pi Zero board

In contrast to other Raspberry Pi boards, such as Raspberry Pi 4, the Zero model **aims to be a compact, cost-effective, and minimalist system**. Nonetheless, it effectively fulfills its role in illustrating fundamental principles shared by more sophisticated boards. This makes it particularly valuable for smaller non-computationally intensive projects or educational purposes. Consequently, it had been chosen as an example for implementing an ARM emulator.

4.1 `ARM1176JZF_S`

`ARM1176JZF_S` is a **single-core 800 `MHz` processor that implements the ARM11 ARM architecture version 6**. It supports both the *ARM* and *Thumb* instruction sets, incorporates Jazelle technology for the direct execution of `Java bytecodes`, and

includes a set of SIMD instructions designed to operate on 16-bit or 8-bit data values within 32-bit registers. Table 4.1 provides a breakdown of the individual letters and numbers found in the processor's name.

Table 4.1: Breakdown of the processor's name (ARM1176JZF_S)

Section	Meaning
ARM11	Indicates that the processor belongs to the ARM11 family
76	Model identifier within the ARM11 family
J	Presence of Jazelle technology (excution of Java bytecodes)
Z	Support for *Thumb-2* technology (mix of 16/32-bit instructions)
F	Support for floating-point operations
S	"Secure" - presence of security features (e.g. TrustZone)

While the majority of its functionality aligns with the principles described in previous Chapter 3, for a more in-depth understanding, the reader can refer to the reference manual [14].

4.2 Microcontroller BCM2835

BCM2835, a microcontroller made by the Broadcom company, **incorporates a range of memory-mapped peripherals that are made accessible to the ARM processor**. These peripherals include timers, an interrupt controller, General-Purpose Input-Output pins (GPIO), a Universal Serial Bus interface (USB), a Direct Memory Access controller (DMA), an I^2C interface, a Serial Peripheral Interface (SPI), and the Universal Asynchronous Receiver-Transmitter, also known as the UART interface [1].

All peripherals, along with their memory-mapped registers, can be found detailed in the BCM2835 ARM Peripherals datasheet [15].

However, it has been observed to contain a number of misleading pieces of information, primarily related to typing errors. Consequently, a separate document has been generated, explicitly enumerating known errors found across various chapters [16].

[1] The BCM2835 microcontroller also includes a couple of peripherals intended for use by the GPU. However, due to complexity reasons, they were not taken into consideration in this paper.

4.2.1 Memory-mapped peripherals

From the programmer's perspective, memory-mapped peripherals are **devices that can be found located on a predefined known addresses in memory** [2]. For instance, according to the datasheet, the `GPIO` controller can be found located at `0x20200000`. This concept is present in one form or another in most modern CPUs.

In addition to the ARM's Memory Management Unit, or `MMU`, the `BCM2835` microcontroller is equipped with a second coarse-grained `MMU` responsible for mapping ARM physical addresses to system bus addresses.

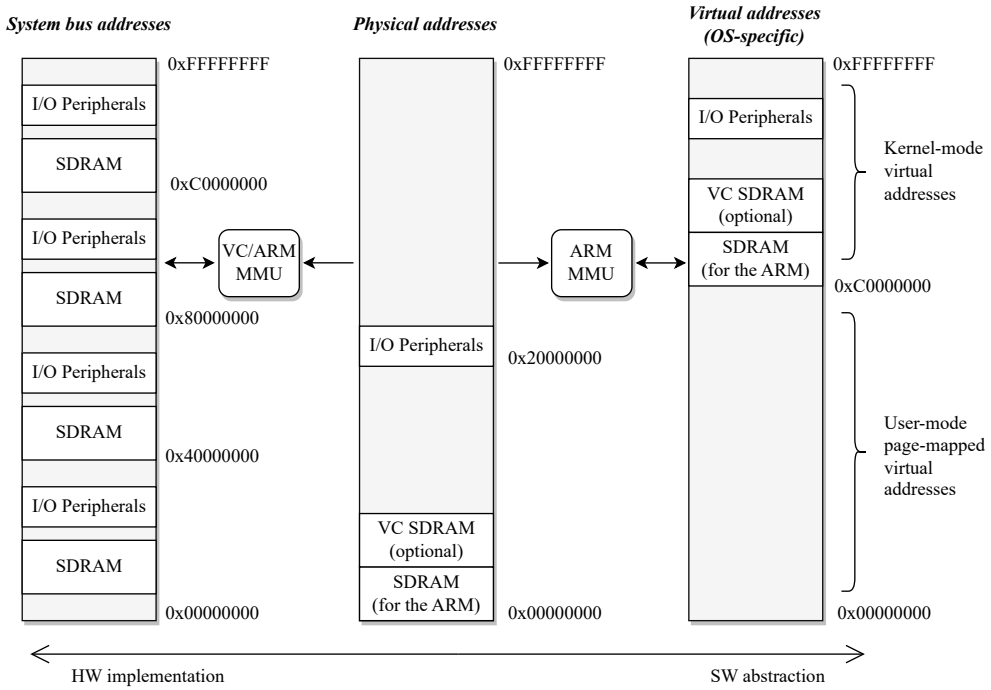


Figure 4.2: `BCM2835` address translation processes[3]

As shown in Figure 4.2, if the ARM `MMU` is employed, the first stage in a memory read-write operation involves translating the virtual address to a physical address based on the page table hierarchy set up by the kernel. Subsequently, once a physical address is obtained, it is mapped to a bus address that corresponds to the particular chip involved in the operation. This mapping occurs due to the fact that memory, such as flash or Synchronous Dynamic Random Access Memory, or `SDRAM`, may often be composed of multiple chips, each potentially implementing different caching policies.

[2] In this context, the term "memory" refers to the entire address space rather than the physical memory.

[3] As an example, Figure 4.2 illustrates the virtual address space utilized in `GNU/Linux`.

Raspberry Pi Zero Emulation

5

Emulating Raspberry Pi Zero on a different platform, such as x86, can offer several benefits and serve various purposes.

For instance, as noted previously, it can be a powerful **tool for educational purposes**, allowing students and beginners to learn about the ARM architecture in a visual and user-friendly way. Furthermore, it can be valuable for illustrating concepts like operating system principles and embedded system development [1]. The author suggests that **it could find applications in a continuous-integration development process**, where it might be utilized for executing an initial set of unit and regression tests to identify early bugs before carrying out testing on real hardware, which may not always be readily available. Examples of applicable domains may include medical devices, automotive, and other safety-related systems. Another potential benefit is that it implicitly provides a level of isolation, which **allows developers to experiment with potentially risky code** without worrying about damaging their board.

All these reasons are further accentuated by ARM's omnipresence, as it is arguably one of the most widely adopted architectures worldwide.

[1] If modularized, it can also be extended by other external modules, allowing the user to create a fully-customized system.

5.1 Terminology

The following terms are sometimes used interchangeably. However, they fundamentally differ in the purposes they serve. Therefore, it is essential to explain them in order to establish a common understanding of their respective roles.

5.1.1 Simulation

In general, a simulation is a computer program designed to replicate a specific process or sequence of events, with the goal of gaining a thorough understanding of its behavior, especially in cases where a comprehensive analytical mathematical solution may be unknown. Simulations are typically parameterized, allowing them to be re-run with different input parameters to investigate how different changes in these parameters impact the observed values. They are commonly employed in various scenarios, such as modeling traffic in a city, simulating disease outbreaks, or predicting weather conditions [17].

5.1.2 Emulation

In emulation, each instruction of architecture *A*, an emulated architecture, is replaced with one or more instructions of architecture *B*, the underlying architecture [18]. This typically results in an implicit slowdown in performance as it requires more operations to achieve the same functionality. The primary advantage of an emulator is that it enables the execution of a program that is not originally written for the underlying architecture. Some popular examples of emulators include DOSBox [19], which can be used for playing retro video games on a modern computer, and QEMU [20], a full-emulation system capable of running various architectures.

5.1.3 Virtualization

Unlike emulation, virtualization allows the execution of a guest application, possibly written for a different operating system, directly on the underlying hardware [21]. As shown in Figure 5.1, this is achieved through a *hypervisor*, installed on top of the host operating system, which manages the execution of multiple distinct host machines that remain "invisible" to each other. Examples of such hypervisors include Xen, KVM, and Microsoft's Hyper-V. These days, virtualization plays a vital role in cloud-based solutions, as it enables dynamic allocation of virtual machines that can be utilized for various purposes, such as web servers, load balancers, computing units, and more. There are also other types of virtualization techniques, such as *paravirtualization* or *full virtualization*. In the case of further interest in this topic,

the reader is encouraged to seek other sources of information, such as the *Hardware and Software Support for Virtualization* book authored by Edouard Bugnion, et al. [22].

Another popular technology that belongs to the emulation/virtualization category is Docker. However, it adopts a slightly different approach by primarily emulating software rather than the underlying hardware.

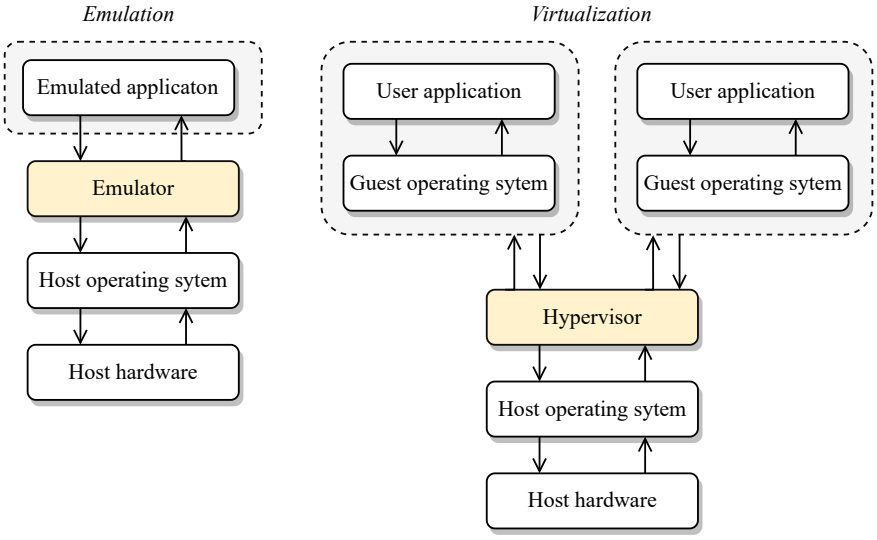


Figure 5.1: Emulation vs. virtualization

5.2 Existing Solutions

There are several available solutions that can serve as ARM emulators. However, it becomes evident that not all of them may be entirely suitable for the comprehensive and configurable emulation of an embedded environment. The following sections provide a review of some existing ARM emulators.

5.2.1 CPUlator

CPUlator is a sophisticated online emulator and debugger that offers **support not only for ARMv7 but also for MIPS and Nios II architectures** [23]. It is specifically designed as a tool for learning assembly-language programming and gaining insight into computer organization. As a **web browser-based application**, it can be accessed from any device, making it an **excellent starting point for beginners**, as it does not require any installation. This accessibility is particularly advantageous for those who might be deterred by the complexities of installing a custom build of QEMU, for example.

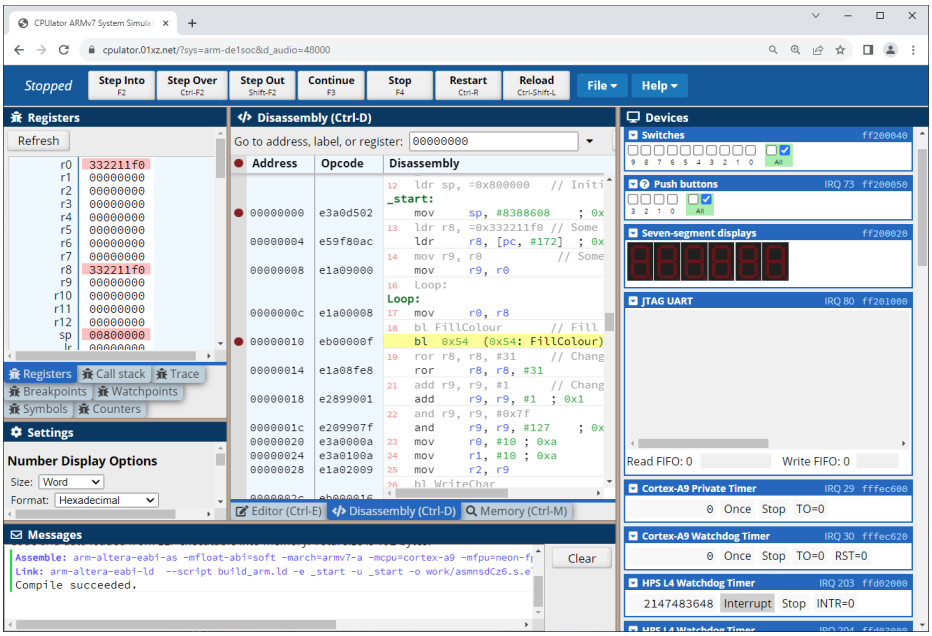


Figure 5.2: CPUlator

Its core is written in C++ and compiled into WebAssembly, resulting in a user-friendly modern-looking interface that offers various debugging features such as breakpoints, single step, reverse step, step over function, step out of a function, modifying registers, showing the stack call, and more. The overall user interface can be seen in Figure 5.2. An input file [2] can either be loaded from the user's computer or entirely written and compiled using its built-in editor. In addition to the built-in compiler, it allows the user to directly upload an already built binary file.

According to its documentation, CPUlator boasts an outstanding emulation speed of 13 mega-instructions per second. As far as ARM is concerned, it features a 4 GB flat memory model with a maximum usable memory of 2042 MB. It also incorporates several input/output devices including a Video Graphics Array, or VGA, buttons, watchdog timers, seven-segment displays, and more. However, the integration of custom external peripherals would be a welcomed feature.

Despite its support for floating-point numbers, CPUlator **lacks support for a memory-management unit**, making it impossible to leverage the full 4 GB address space. Similarly, it **restricts the user from switching the default CPU mode**, rendering it unsuitable for demonstrating most of the operating system principles. Nevertheless, it successfully fulfills its intended purpose as an excellent tool for learning the ARM assembly language.

[2] It supports both the C programming language and ARMv7 assembly.

5.2.2 ARMSim#

Another tool designed primarily for educational purposes is the ARMSim# emulator, developed at the University of Victoria in Canada [24]. This **desktop application, created using the .NET technology**, allows users to observe and debug the execution of ARM assembly programs on a system utilizing the ARM7TDMI processor [3], which implements the **ARMv4 architecture**. The user interface can be seen in Figure 5.3. Similar to CPUlator, it supports debugging features such as stepping through the source code and viewing the current contents of both the CPU registers and memory. According to the CPUlator's comparison chart, ARMSim# is claimed to be capable of executing up to 3 mega-instructions per second [23].

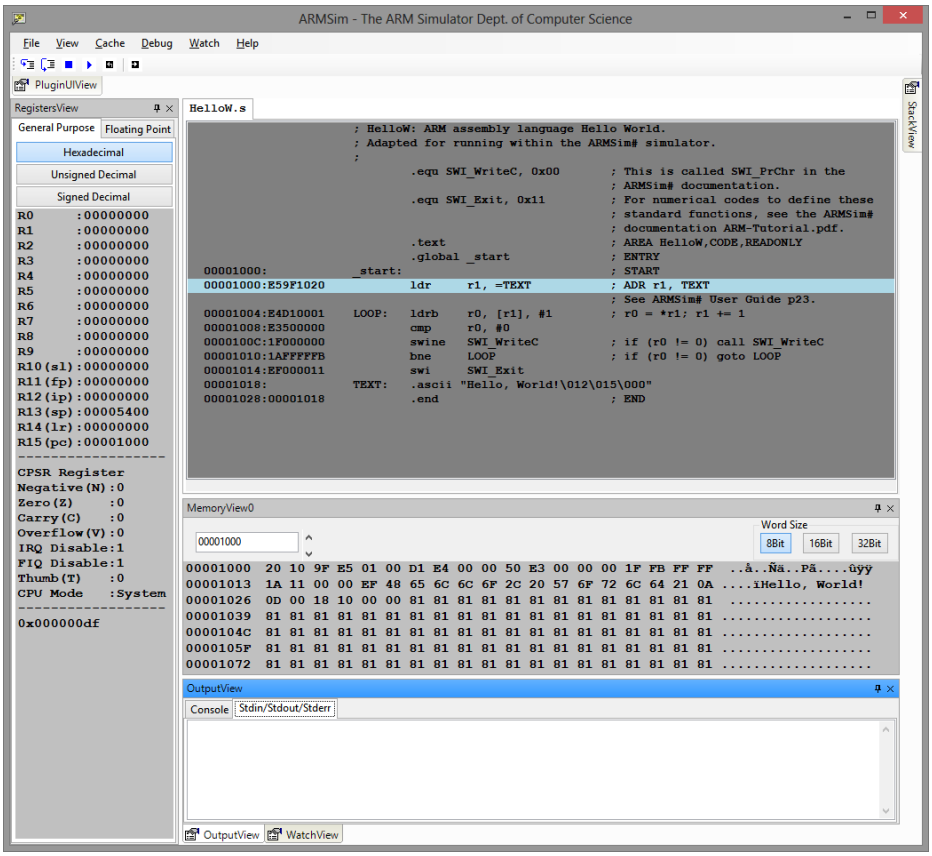


Figure 5.3: ARMSim#

The emulator **features so-called plugins that further extend its functionality in a modular fashion**. Examples of these plugins may include *SWIInstructions*, which encapsulates a variety of system calls, such as reading and writing to a file, or *EmbestBoard*, which allows the core to interact with external peripherals such as

[3] As mentioned previously in Chapter 3, this processor was widely adopted by the cell phone industry. However, it is no longer licensed by ARM and may be considered outdated.

a keyboard, segment display, or a 40x15 display — all of which are also interacted with via system calls.

ARMSim# also supports vector floating-point instructions; however, similar to CPUlator, it **lacks support for a memory-management unit**. It can be concluded that while it finds application in learning the ARM assembly language, it does not provide features for advanced system-level programming, like developing an operating system, **nor does it support the concept of memory-mapped peripherals**, which can be found in the majority of modern embedded devices.

5.2.3 QEMU

QEMU stands out as a powerful **cross-platform full-emulation system**, written in C, with the capability to emulate operating systems across various architectures, including x86, MIPS, ARM, and PowerPC [20]. While it may not offer the same visual appeal as the previous examples, its versatility surpasses them. QEMU can operate in both graphics mode, if supported by the guest operating system, and terminal-only mode. In terms of debugging, **it allows the attachment of the GNU debugger**, a tool heavily utilized in Linux-based operating systems.

However, when attempting to emulate the Raspberry Pi Zero environment, **QEMU does not provide a plug-and-play solution upon installation**. By default, it expects an ARM-based system to boot up from address `0x00010000`. In contrast, the Raspberry Pi Zero's *first stage bootloader* necessitates the kernel's reset vector to be situated at address `0x00008000`.

Consequently, users face two options. They can either relocate their kernel to the required address or modify QEMU's source code to change the kernel load address and create a custom build. It is important to mention that the second approach may pose certain challenges for beginners. Conveniently, the entire process of running a custom image built for Raspberry Pi Zero is detailed in the KIV/OS - dodatek A document [25].

While QEMU does support most of the features that the previous emulators lack, its versatility may also come with certain disadvantages when emulating the Raspberry Pi Zero environment. Specifically, there are reported **issues related to the system timer**, hindering the implementation of a preemptive scheduler in the custom operating system. Furthermore, it lacks a straightforward solution for attaching external hardware peripherals. In the case of Raspberry Pi, it can be inferred that QEMU might not be the optimal choice for emulating such an embedded environment, given its primary focus on emulating more general-purpose operating systems.

5.3 General Requirements

Reviewing existing solutions renders a set of general requirements for an ideal emulator designed to emulate the Raspberry Pi Zero environment.

Preciseness

It is evident that the emulator should identically mimic a real Raspberry Pi Zero board. This is arguably the most crucial requirement to fulfill, as failing to do so could result in unexpected behavior. Ideally, if the user compiles and runs their application inside the emulator, they should be presented the with same output as if it was executing on real hardware. However, this may sometimes be challenging to achieve, especially considering time constraints mentioned previously in Section 5.1.2.

Effectiveness

In terms of performance, a slowdown in emulation speed is nearly inevitable. Nevertheless, the emulator should leverage modern technologies to mitigate these effects. Consequently, the use of certain programming languages like Python might be ruled out, as they could adversely impact overall performance, which is discussed further in Section 7.1. Moreover, the emulator should strive to consume only a reasonable amount of the system's resources.

User-friendliness

The final emulator should introduce users to embedded development principles through a visual and user-friendly interface [4]. They should have the capability to perform all the previously mentioned actions, such as setting breakpoints and viewing the contents memory and registers, without the necessity of connecting an external debugger. This is particularly crucial for beginners, as they should have access to these features effortlessly upon installation. Another essential requirement is that users should not be restricted to the use of the emulator on a single platform, such as Windows.

Extensibility

In the embedded world, microcontrollers are frequently programmed to communicate with external devices, including motors, displays, sensors, or even other microcontrollers. Therefore, the emulator should facilitate the integration of external devices that can be developed independently of the emulator itself, thereby enabling other developers to contribute to the project. This functionality would empower users to configure a real-world scenario tailored to their specific application.

[4] In practice, this can be measured, for instance, through surveys.

Design of a Raspberry Pi Zero Emulator — 6

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].

6.1 Input

The emulator necessitates a single input file in the `ELF` format, simplifying the Raspberry Pi's 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 6.2 illustrates the general process of building an `ELF` file, which contains all essential data and information required for code emulation.

6.1.1 Executable and Linkage Format

`ELF` stands for *Executable and Linkage Format* [26], 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`. The structure of an `SREC` record is visualized in Figure 6.1.

[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, the symbol table stored in the final ELF file can be used during the parsing process, which is discussed in Section 6.3.2, 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, 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 6.6.

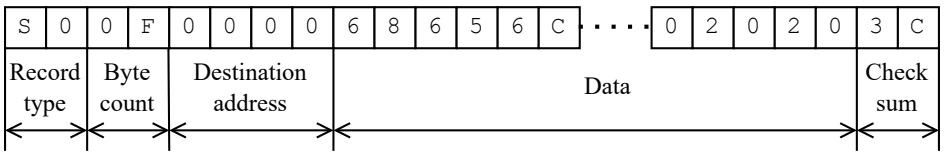


Figure 6.1: Single SREC record (16-bit addressing)

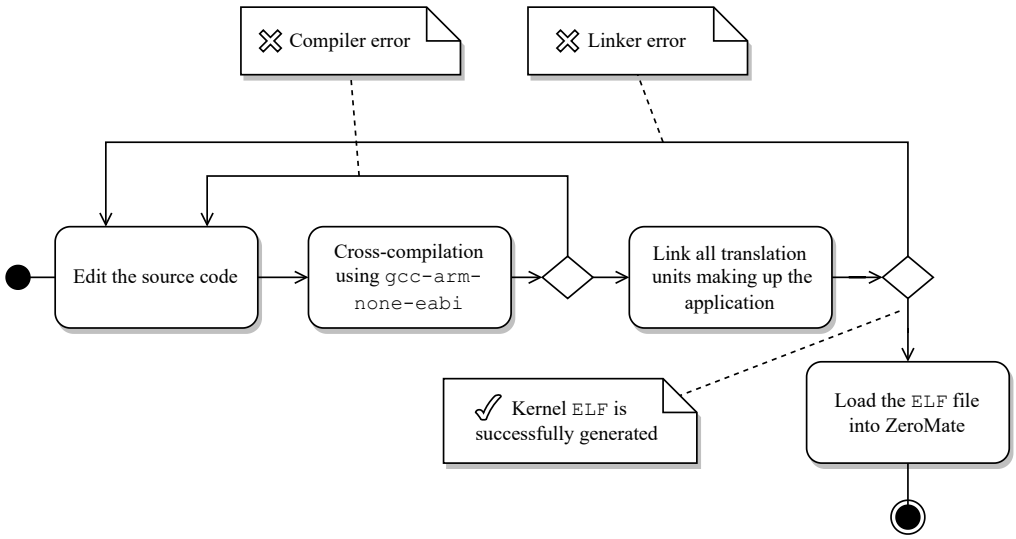


Figure 6.2: Process of building an ELF file (input for the emulator)[2]

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

6.2 User's Interaction

As shown in the deployment diagram in Figure 6.4, 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 6.3, 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.

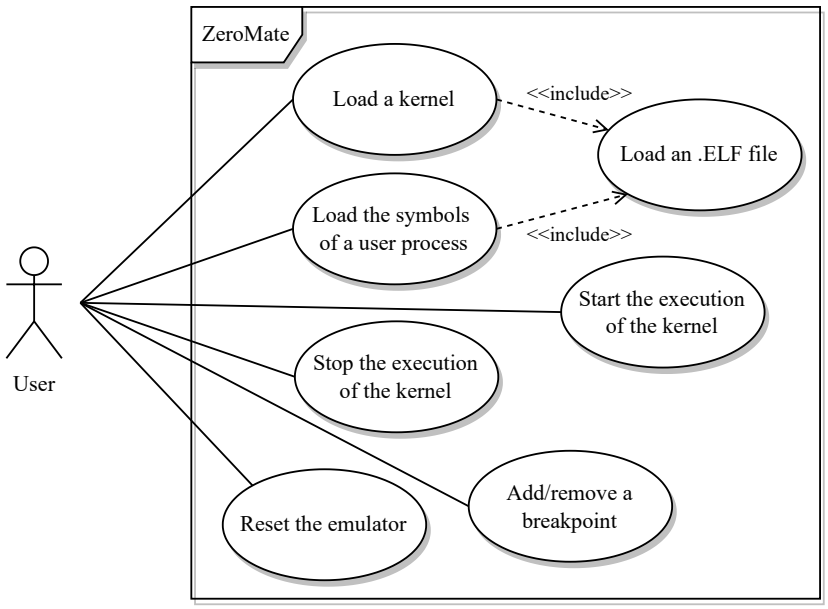


Figure 6.3: Primary use-cases of the ZeroMate emulator

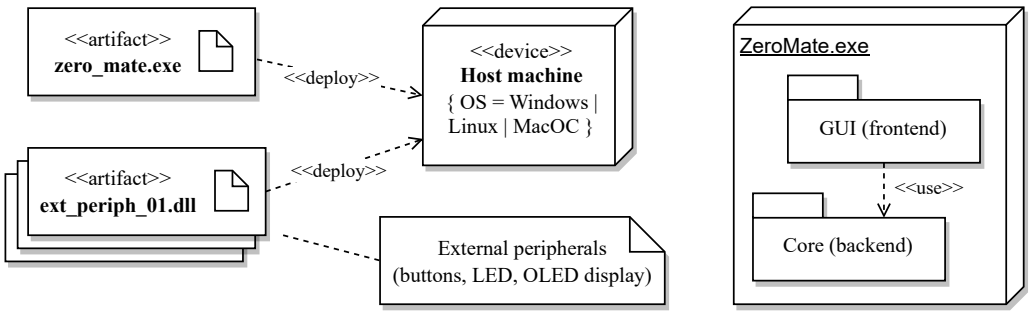


Figure 6.4: Deployment diagram of the ZeroMate emulator

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 part of the emulator.

6.3 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.

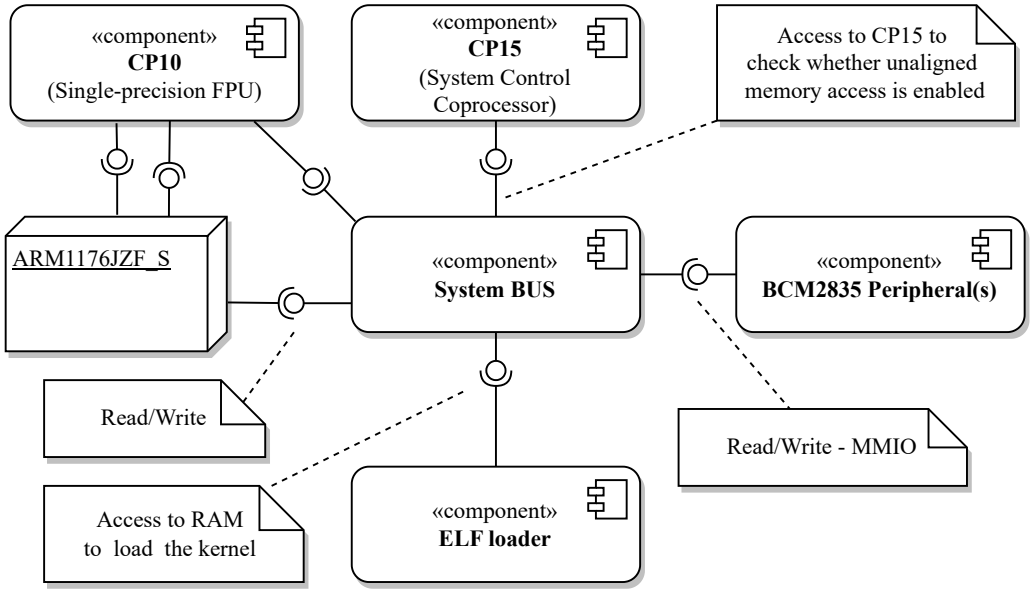


Figure 6.5: Core components of the ZeroMate emulator

Figure 6.5 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 [3]. 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.

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

6.3.1 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 [15]. The primary functions this interface provides can be seen in Listing 6.1. 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 6.3.3). All actions associated with the request itself are then handled internally within the target peripheral.

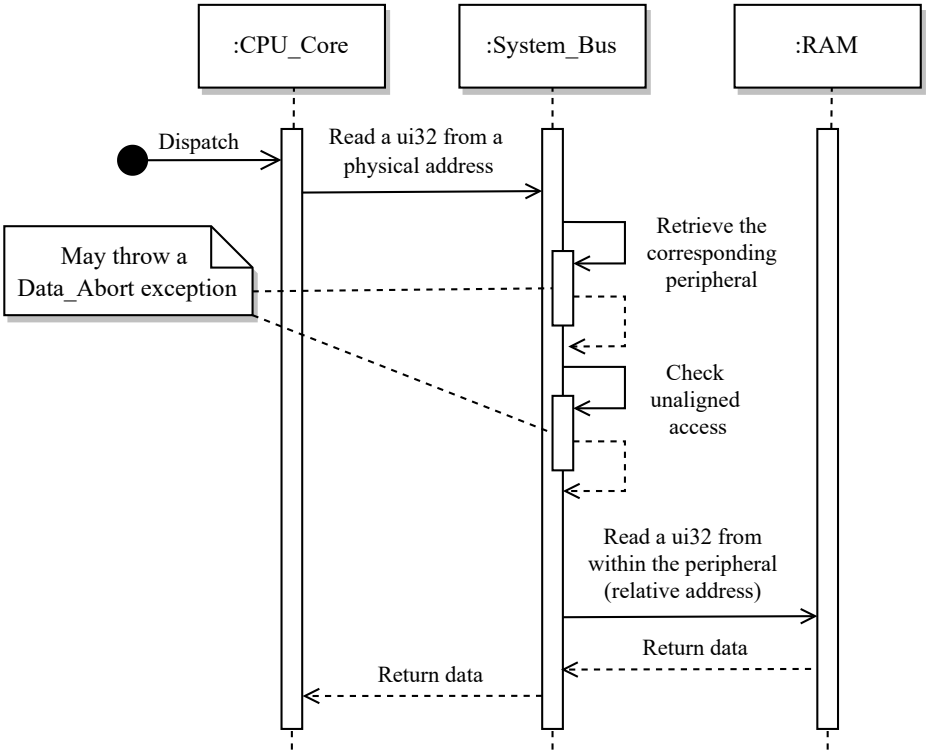


Figure 6.6: Example of a read/write data request issued by the `CPU`

As shown in Figure 6.6, there are two internal steps the system bus carries out before proceeding with the request. First, it 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** [4]. 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 [27].

Source code 6.1: 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).

6.3.1.1 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, which is visualized in Figure 6.7. 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 $O(\log n)$ time complexity [28], which is crucial for improving the overall speed of the emulation.

According to statistics, on average, **load-store instructions account for more than 50% of all instructions in an x86 application** [29]. 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] 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.

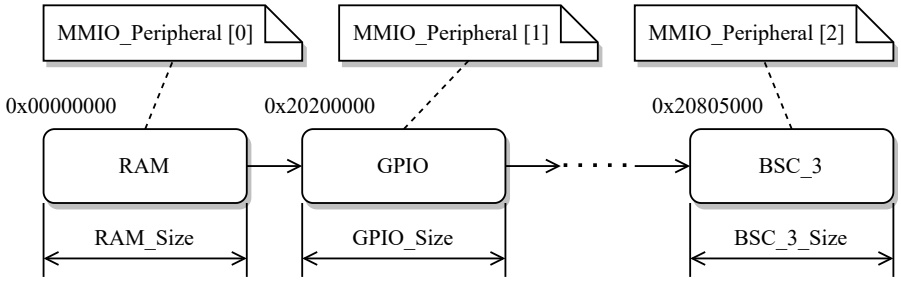


Figure 6.7: Collection of memory-mapped peripherals

6.3.1.2 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 [5]. Nevertheless, this behavior can optionally be disabled, for example, for compatibility reasons, in the *System Control Co-processor* (`CP15`) using the sequence of ARM instructions shown in Listing 6.2.

Source code 6.2: 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
```

[5] 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.

6.3.2 Executable and Linkage Format File Loader

The main purpose of this module is to **parse an input ELF** file and copy the `.text` section, which contains source code instructions, `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 [6], thereby presenting the user with function names that have not undergone modification by the compiler for its internal purposes. An example illustrating symbol demangling is demonstrated in Listing 6.3.

Source code 6.3: Example of symbol demangling

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

Figures 6.8 and 6.9 illustrate the steps and components involved in loading and parsing an input `ELF` file.

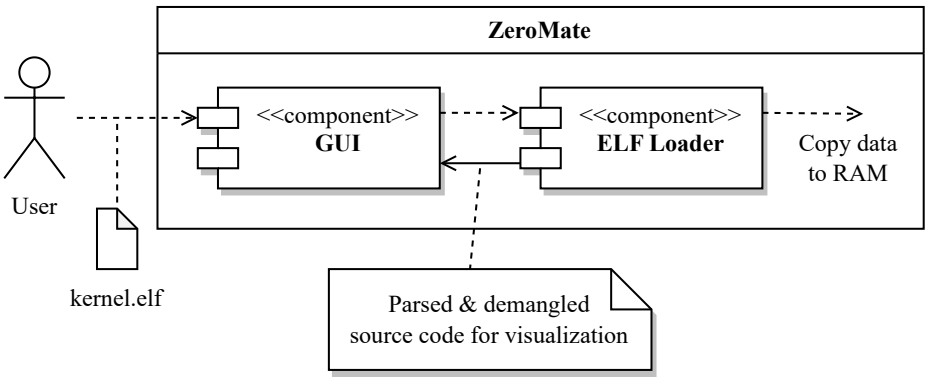


Figure 6.8: Loading an input `ELF` file (kernel)

[6] Demangling is a process of transforming `C/C++` `ABI` identifiers into the original `C/C++` source [30].

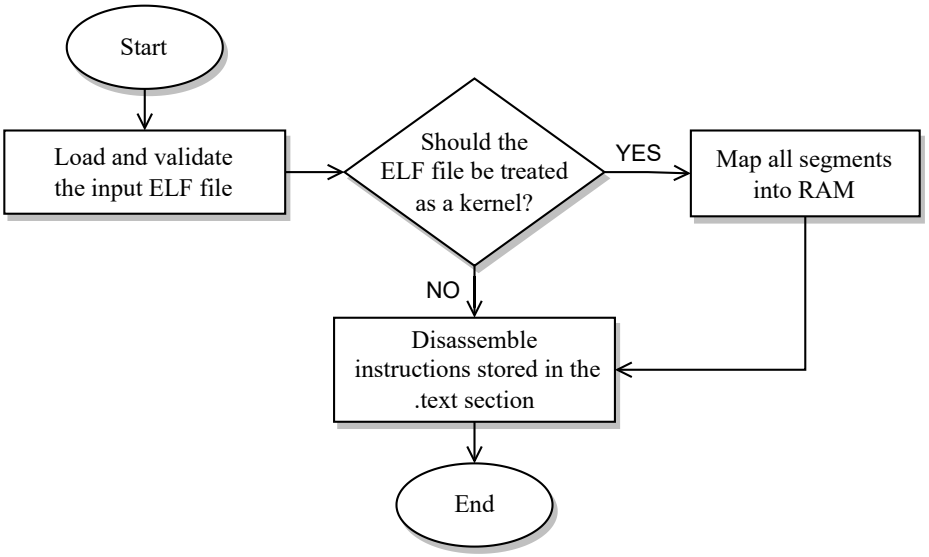


Figure 6.9: Internal logic of the ELF Loader component

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* [31] and *Demumble* [32], to accomplish these functions.

6.3.3 BCM2835 Peripherals

ZeroMate distinguishes between two types of peripherals - those directly integrated within 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.

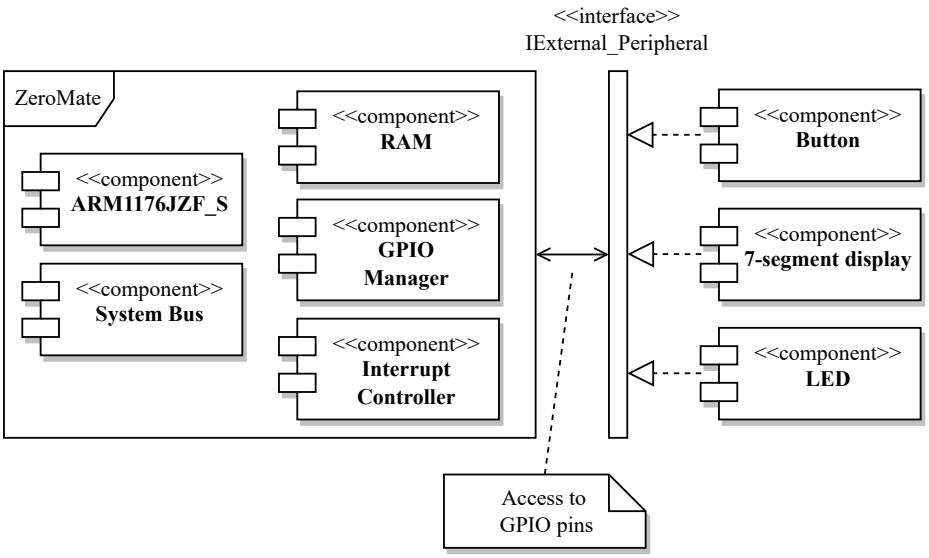


Figure 6.10: Internal vs External peripherals

In Section 6.3.1, 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. In a proxy-like fashion, 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 `BCM2835` manual [7] [15]. By reading from or writing to these registers, the internal state of the peripheral can be modified, which is specific for each peripheral.

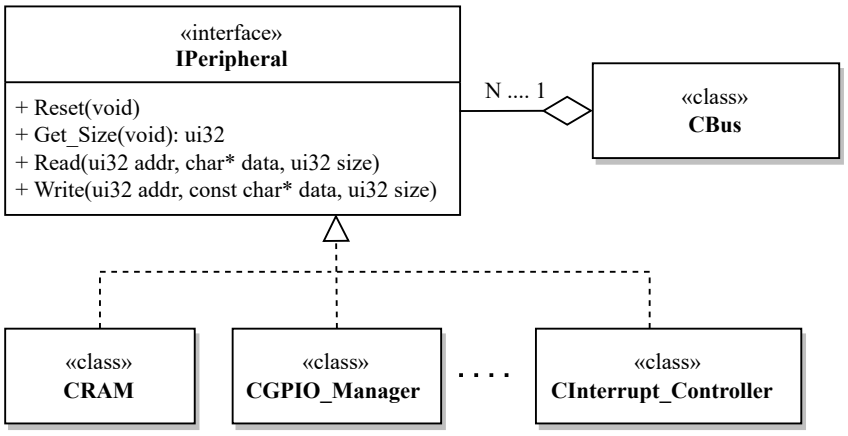


Figure 6.11: Hierarchy of internal peripherals

[7] As mentioned previously, 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 [16].

The `Get_Size()` method shown in Figure 6.11 is used primarily for detecting bus collisions when mapping peripherals into the address space, which was mentioned previously in Section 6.3.1.1.

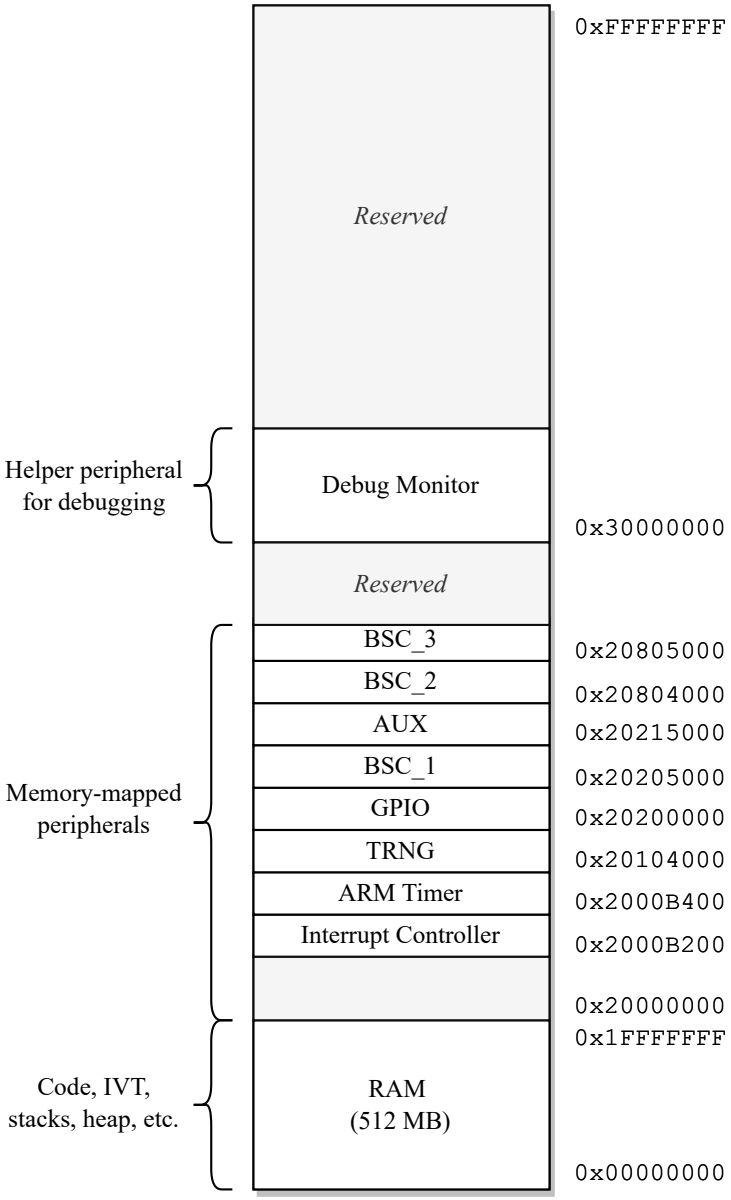


Figure 6.12: `BCM2835` physical memory layout emulated by ZeroMate

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 its emulation efforts on the most frequently utilized peripherals, such as the `ARM` timer, `GPIO`, interrupt controller, 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 6.12.

6.3.3.1 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.

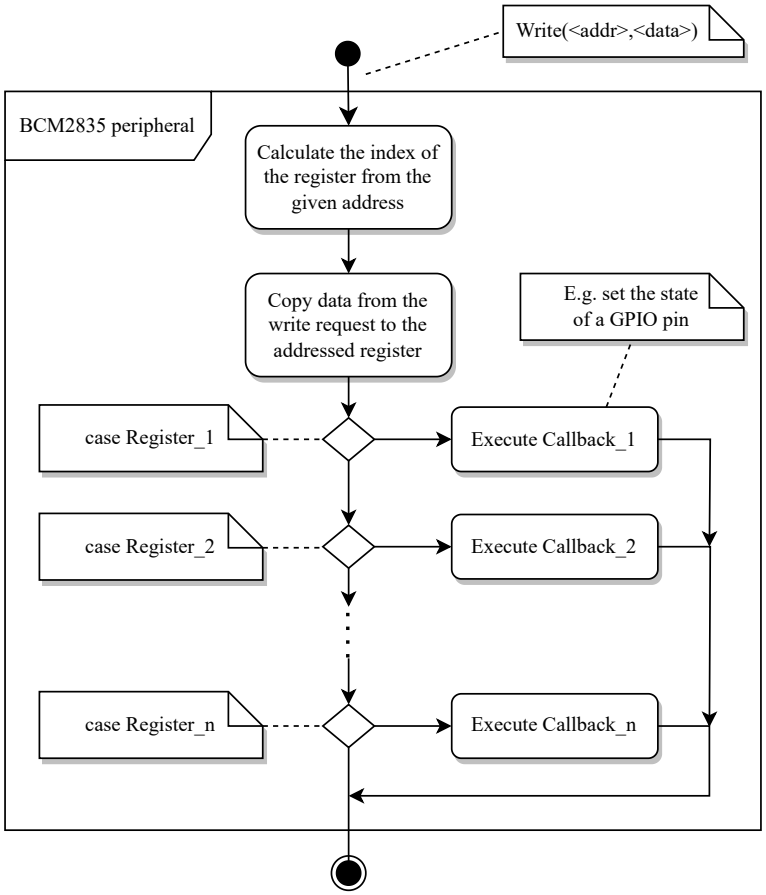


Figure 6.13: Writing to a peripheral's register[8]

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 specific register. Generally, this approach, which is shown in Figure 6.13, can be applied to the vast majority of memory-mapped peripherals.

[8] Reading works in a similar way. Optionally, there might be additional prior actions taken before a value is read from a peripheral's register, such as inserting a random number into the `data` register when utilizing `TRGN`, which is described in Section 6.3.3.5.

6.3.3.2 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. This concept is visualized in Figure 6.14. Whenever an instruction is executed, the CPU notifies all of the system clock listeners of how many CPU cycles it took to execute the instruction, allowing them to update themselves accordingly. Further information on how ZeroMate emulates the execution time of individual ARM instructions can be found in Section 6.3.4.6.

Examples of such listeners may include the ARM Timer or the AUX and BSC peripherals, which encapsulate time-based hardware communication functions. These peripherals are described in Sections 6.3.3.6, 6.3.3.9, and 6.3.3.10, respectively.

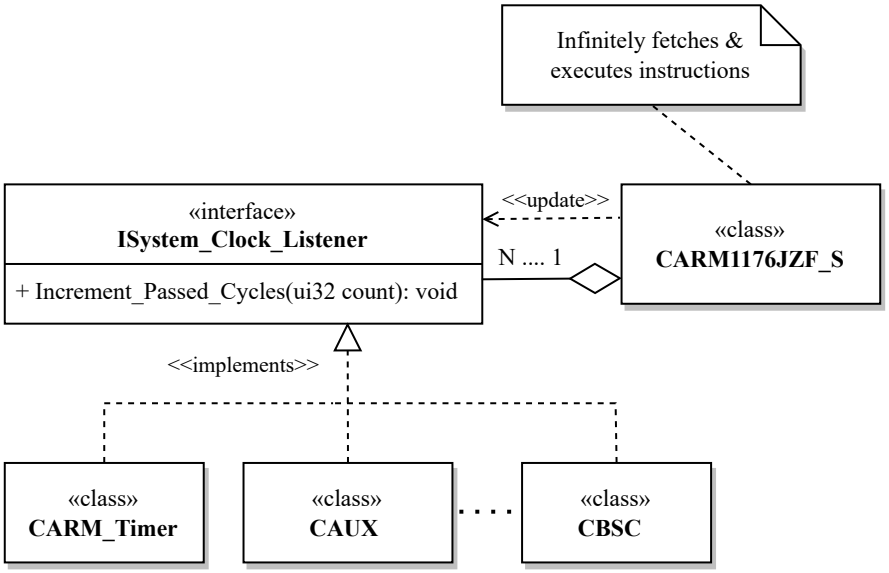


Figure 6.14: `ISystem_Clock_Listener` interface

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.

6.3.3.3 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 [9] 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 [10], 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 Figure 6.15 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.

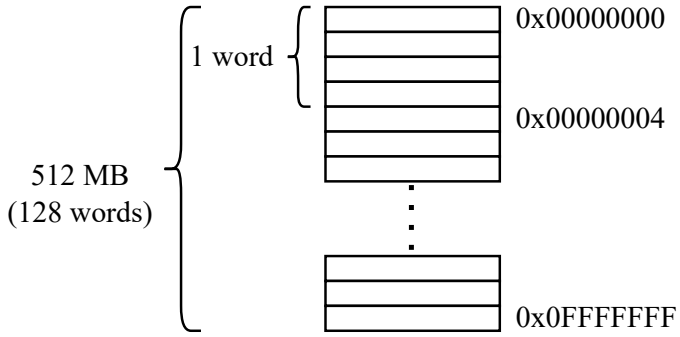


Figure 6.15: RAM implementation as a continuous piece of memory

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.

[9] Non-volatile memory is capable of persisting data even after the supply voltage is turned off.
[10] As described in Section 3.3.3.1, ARM uses a different set of registers for each CPU mode.

6.3.3.4 Debug Monitor

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

> This 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 for the debug monitor that the user can seamlessly integrate into their build system. As showcased in Listing 6.4, 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 6.4: 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;
}
```

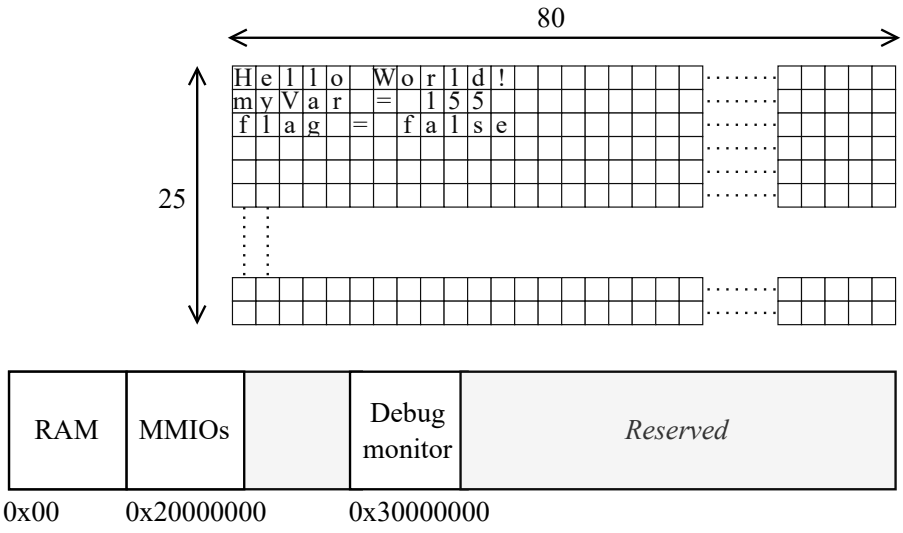


Figure 6.16: Memory-mapped debug monitor

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 [11]. This is commonly achieved by running software like *PuTTY* [33] on the user's computer.

As shown in Figure 6.16, 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. Inspired by the standard terminal text mode, the size of the monitor was chosen to be `80x25 8-bit` characters [12].

6.3.3.5 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 [15], its existence can be confirmed, for instance, by examining the implementation in the GNU/Linux kernel [34].

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 [13].

From the user's code perspective, the process of **retrieving a random number consists of two steps**, which are displayed in Figure 6.17.

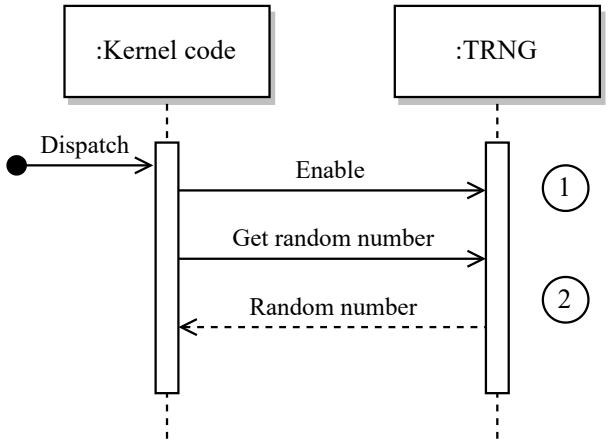


Figure 6.17: Reading random numbers from the `TRNG` peripheral

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

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

[13] 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, which could potentially be another area for future improvements.

Utilizing a pseudo-random number generator, such as an *LCG* [35] 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.

6.3.3.6 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 6.18, 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 emulated CPU's clock.

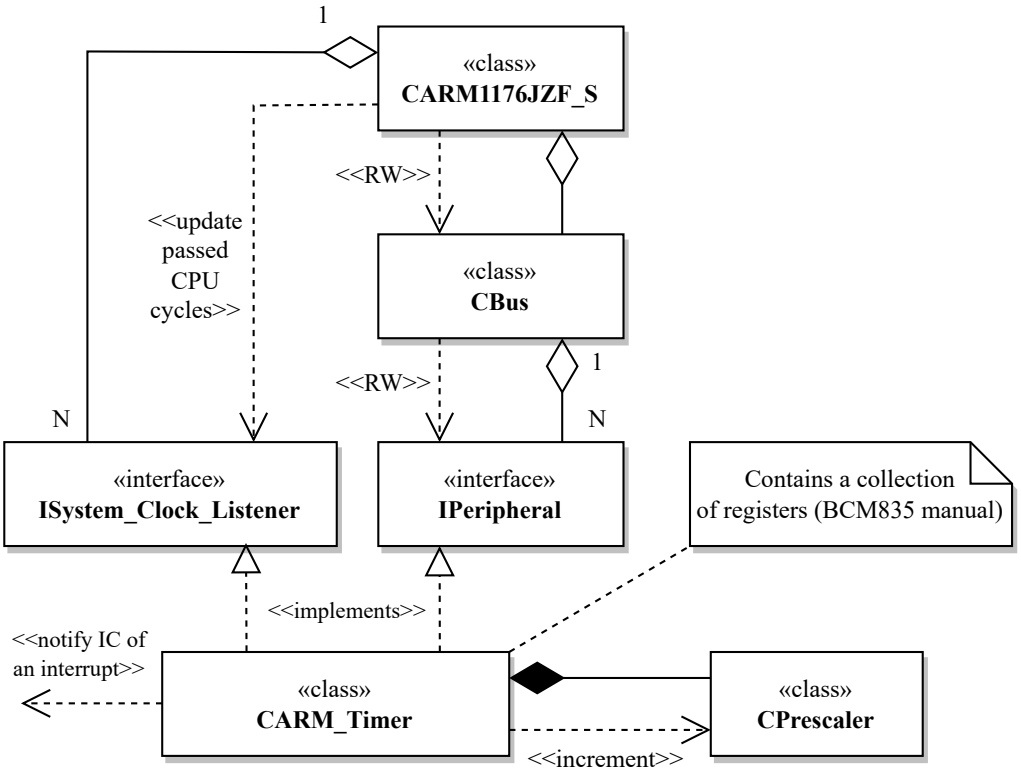


Figure 6.18: Context of the ARM timer component

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 6.19.

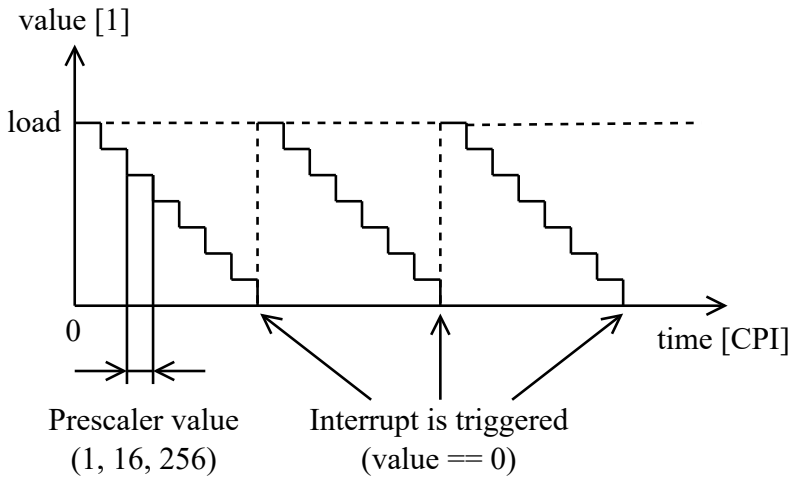


Figure 6.19: Content of the value register of the ARM timer over time

6.3.3.7 General Purpose Input/Output

The GPIO manager is a peripheral that manages the **54 general-purpose input-output pins** that are available to the programmer. These GPIO pins also function as a connecting interface for all external devices, which is shown in Figure 6.10. Each pin can be 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.

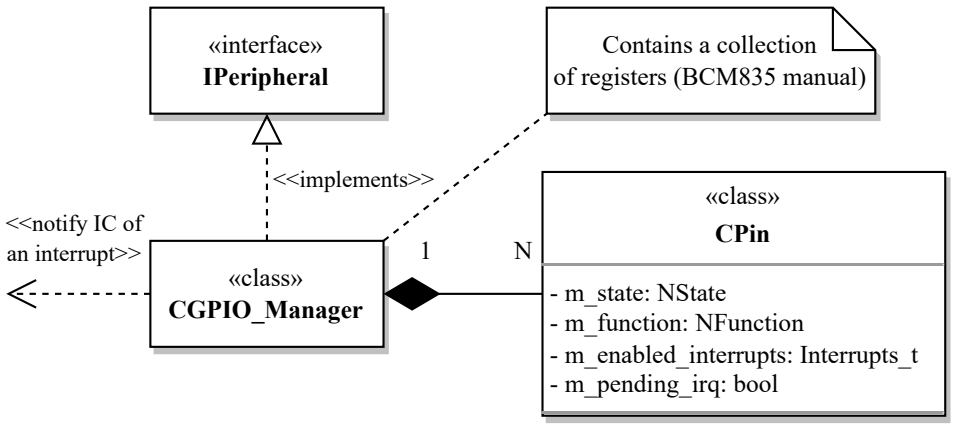


Figure 6.20: Structure of the GPIO manager[14]

All GPIO pin functions, as well as interrupt types, can be found explained in the BCM2835 manual [15].

[14]The CPin class also exposes a set of public functions (getters/setters) that allow the caller to access its private data members.

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 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`. To accommodate analog signals, the `GPIO` controller would need to be extended by an additional piece of information indicating whether a pin is digital or analog to either hold a `bool` or `signed integer` value respectively [a].

[a] In reality, the `GPIO` controller routes analog pins to the `ADC`, or *Analogue to Digital Converter*, which is another internal peripheral of Raspberry Pi Zero.

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. As shown in Listing 6.5, one approach to emulate such behavior is to reset the register to its default value after a write operation has been performed.

Source code 6.5: `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 6.20.

6.3.3.8 Interrupt Controller

As shown in Figure 6.21, 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 discarded**. From the `CPU`'s point of view, after an instruction has been 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.

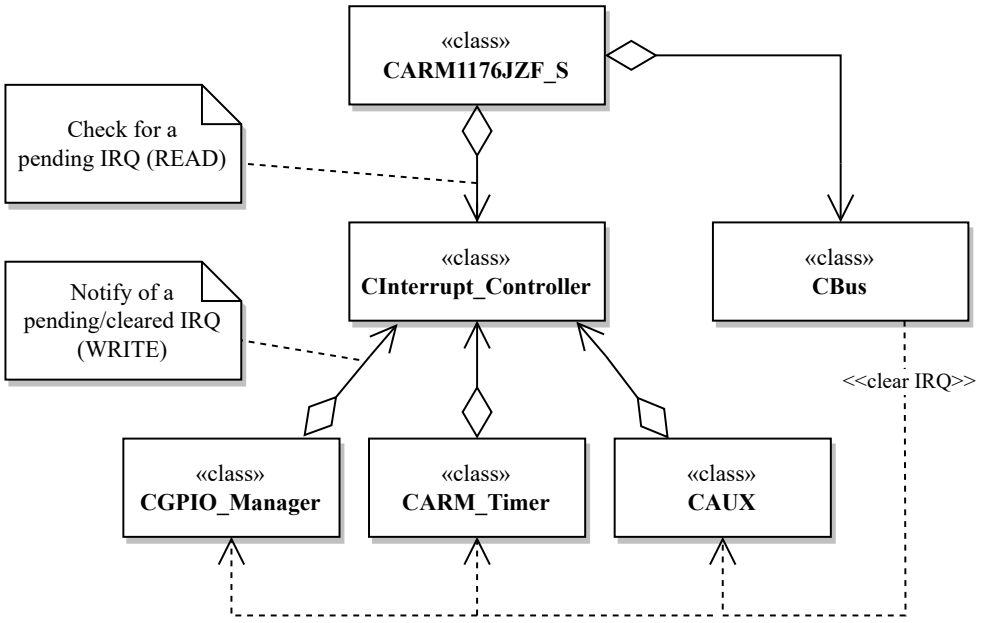


Figure 6.21: Context of the interrupt controller

In terms of design, the interrupt controller encapsulates an associative storage that pairs each `IRQ` source [15] with its associated metadata, indicating whether it is enabled and if there is a pending interrupt. From the `CPU`'s perspective, this storage, which is shown in Figure 6.22, is read-only, as its sole 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.

As noted in Chapter 3, an ARM 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.

[15] The reader can find a list of all available `IRQ` sources in the `BCM2835` datasheet [15].

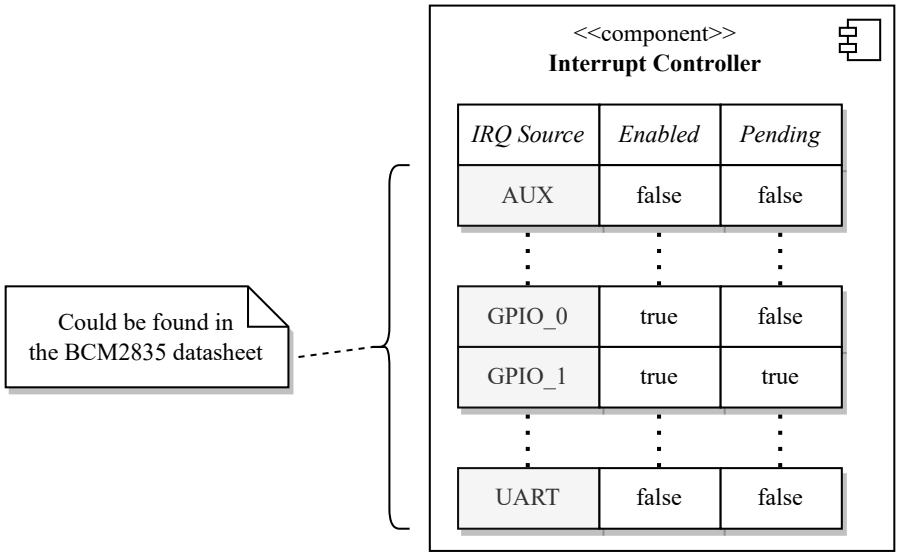


Figure 6.22: Encapsulated information about `IRQ` sources

It is worth mentioning that the interrupt controller of the `BCM2835` microcontroller distinguishes between two types of interrupt sources - *Basic `IRQ`*, which are ARM-specific, and *`IRQ`*, which are shared between the `GPU` and the `CPU`. Both types can be found listed in the `BCM2835` datasheet [15]. Nevertheless, from a design perspective, the underlying principles remain the same.

6.3.3.9 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 Figure 6.23.

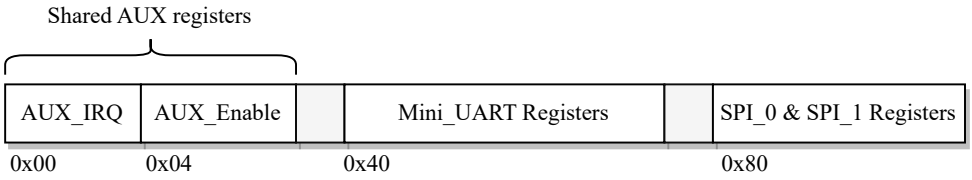


Figure 6.23: Registers of the `AUX` peripheral

Whenever a read/write request is received, using the technique described in Section 6.3.3.1, the `AUX` class can efficiently redirect the execution to the relevant auxiliary peripheral, which will subsequently handle the request internally.

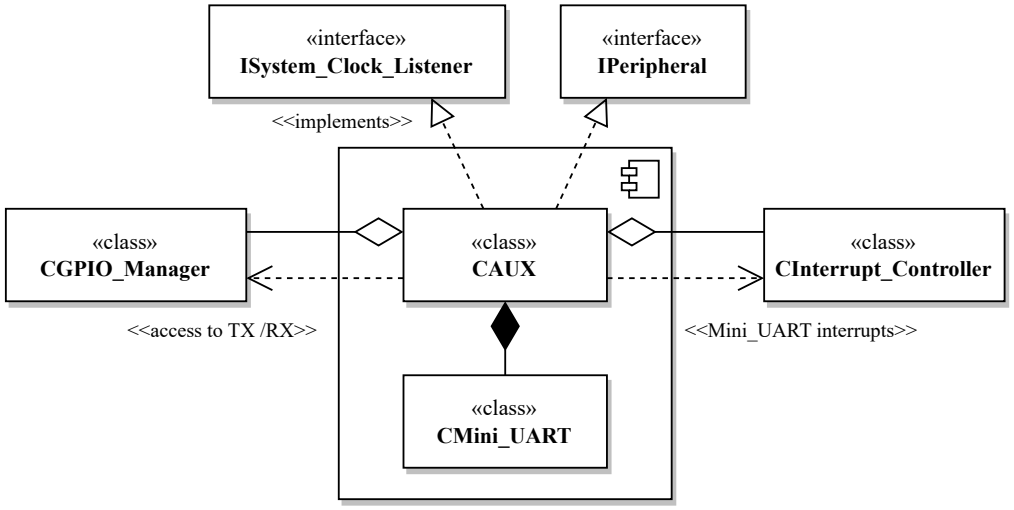


Figure 6.24: Structure of the AUX peripheral

Figure 6.24 shows the internal structure and dependencies of the AUX peripheral. It can be noticed that it implements the ISystem_Clock_Listener interface, whose purpose is described in Section 6.3.3.2, which allows it to synchronize with the rest of the system.

Mini_UART

UART, which stands for *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.

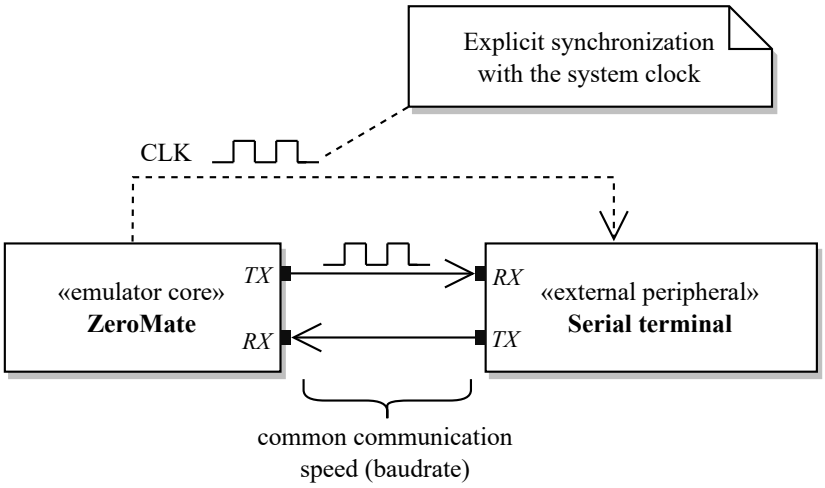


Figure 6.25: UART communication with an external peripheral

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 [15].

In ZeroMate, all external peripherals are provided read-only access to the system clock, enabling them to synchronize themselves if required. This approach, depicted in Figure 6.25, contradicts the fundamental principles of asynchronous communication since it introduces a form of synchronization. However, this design choice 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 6.27. 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 6.26 [16].

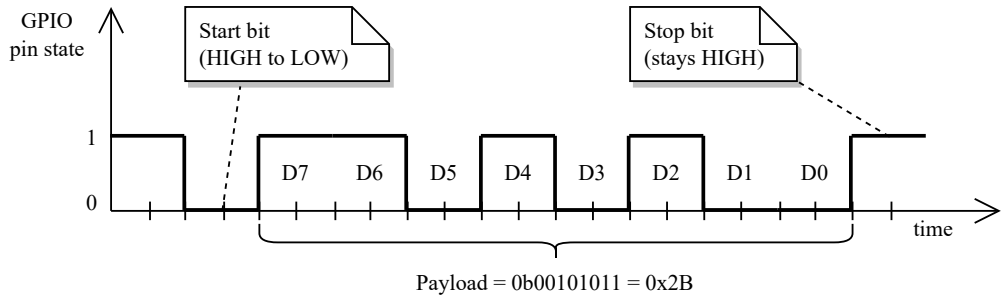


Figure 6.26: Example of a Mini_UART data frame with 8 bits of data

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 pin, which has been set previously by an external peripheral.

[16] 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 [15].

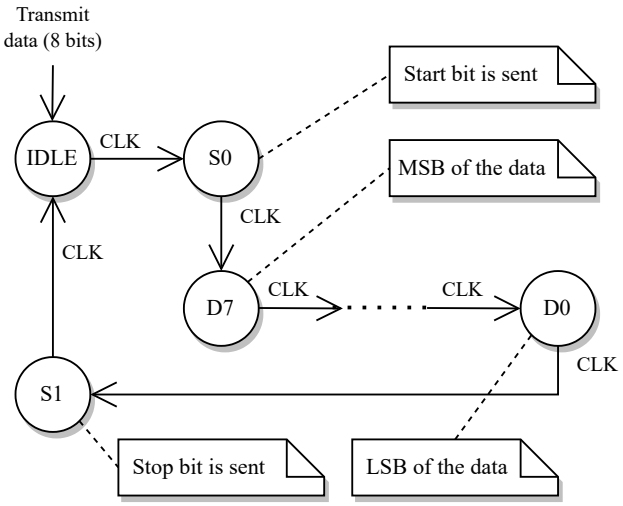


Figure 6.27: `Mini_UART` state machine (transmitting 8 bits)

6.3.3.10 Broadcom Serial Controller

The Broadcom serial controller, often abbreviated as BSC, is a peripheral device that allows the user to communicate with external devices, usually 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 6.12.

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 clock.

While there are various configurations of the I^2C communication protocol [a], ZeroMate exclusively supports only one, which is presented in Figure 6.28. 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.

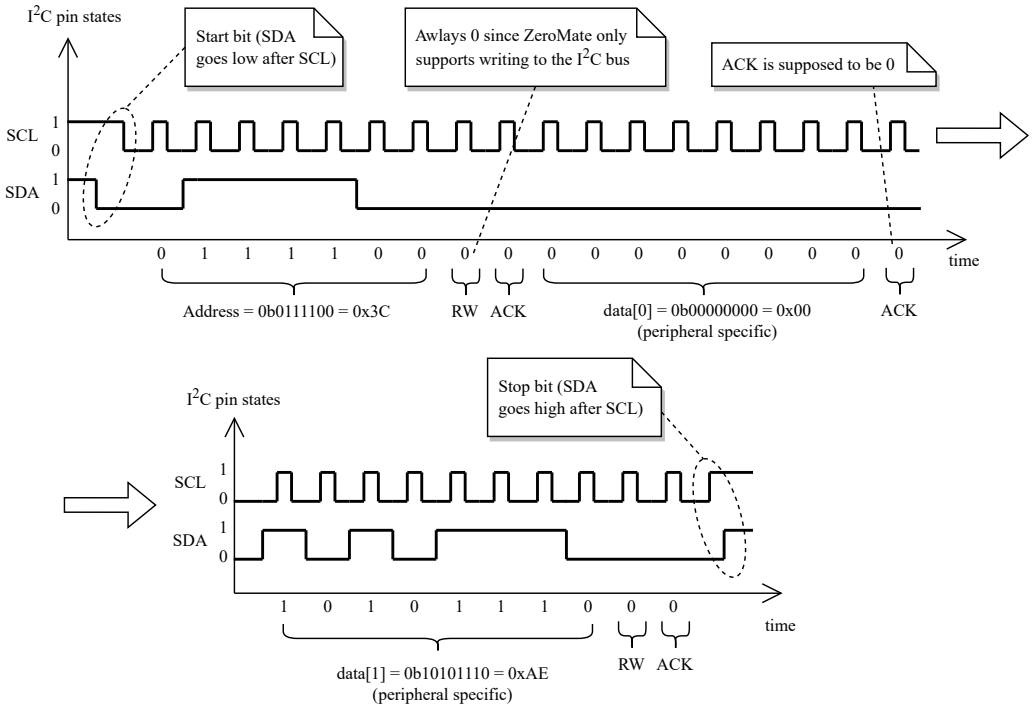


Figure 6.28: Structure of an I²C frame

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 6.4.

6.3.4 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 6.29, 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.

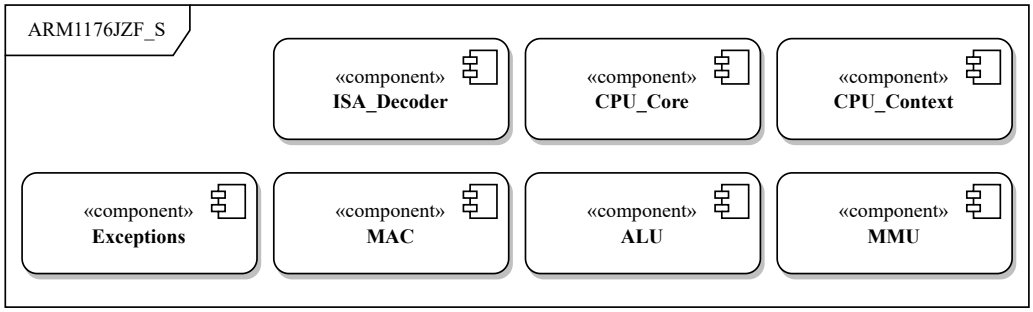


Figure 6.29: Internal components of the `ARM1176JZF_S` processor

6.3.4.1 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 *Current Program Status Register*, which are shown in Figure 3.6. From a design standpoint, its relationship with the CPU core is captured in Figure 6.30.

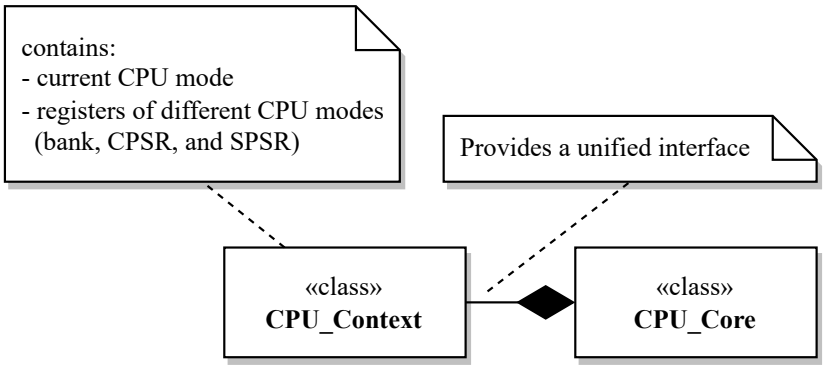


Figure 6.30: Relationship between the CPU context and the CPU core

Bank Registers

One of the presented challenges involves implementing so-called *bank* registers, where, as mentioned in Section 3.3.3.1, 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 [11].

As illustrated in Listing 6.6, 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 6.6: 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 Status Register* and the *Saved Program Status 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.

6.3.4.2 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 6.7 below.

Source code 6.7: 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 6.31. 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.

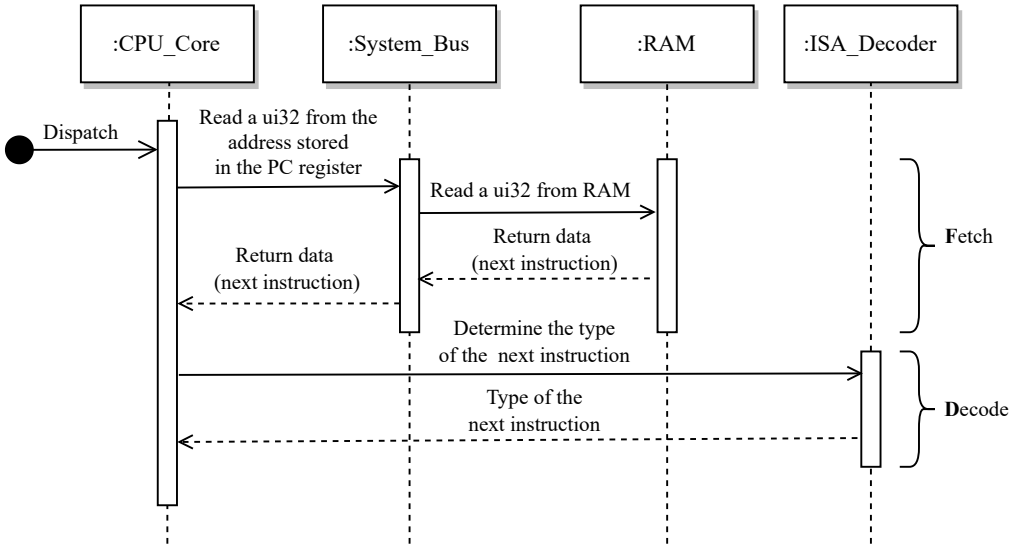


Figure 6.31: Use of the `ISA` decoder by the `CPU`

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.

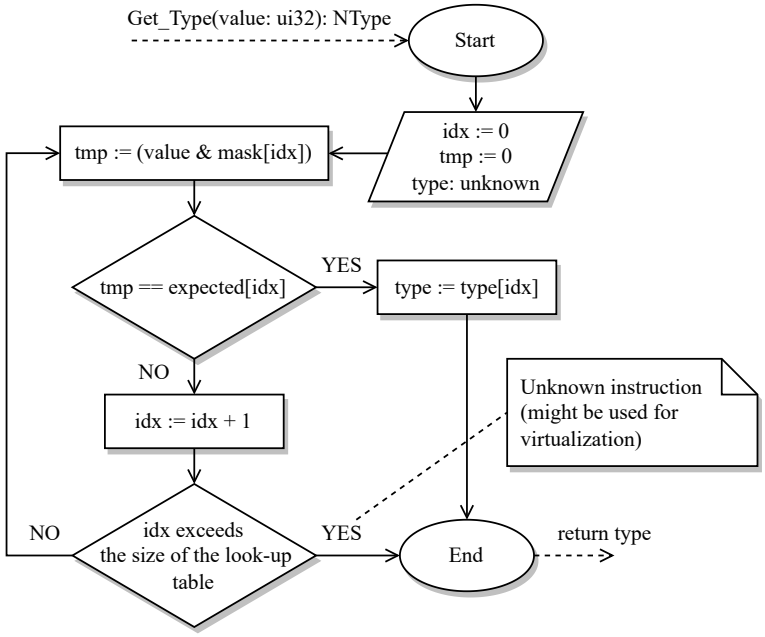


Figure 6.32: Algorithm for decoding `ARM` instructions

Figure 6.32 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 has been 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 [13].

6.3.4.3 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 the unprivileged 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 6.33 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, which is shown in Table 3.2.

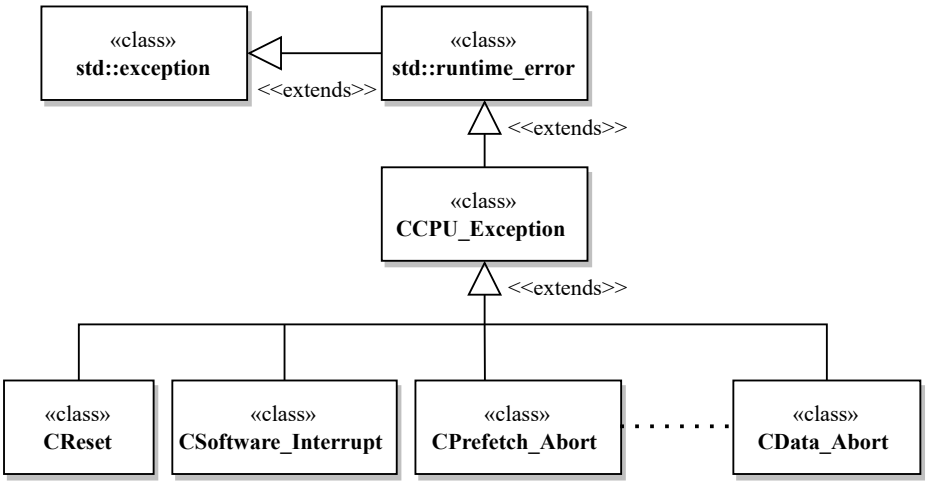


Figure 6.33: Hierarchy of the CPU exceptions

6.3.4.4 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 6.34.

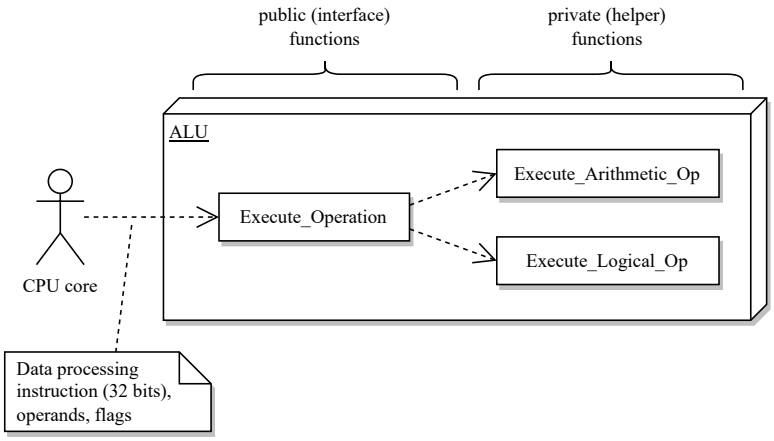


Figure 6.34: Architectural structure of the ALU

As illustrated in Figure 6.34 above, when the ALU is employed, the CPU must provide the current state of its flags as well as the data processing instruction currently in execution, which is then 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 a multiplication. All instruction encodings can be found in the B2 Appendix document [13]. From an implementation perspective, it can be integrated in a manner similar to the ALU.

6.3.4.5 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. As shown in Figure 4.2, **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 [17].

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 of the target frame based on the current mode of the CPU in order to ensure that no security policy is being violated.

If all checks have successfully passed, the MMU proceeds to convert the virtual address into a corresponding physical address. More detailed information about how the MMU operates can be found in Chapter 6 of the ARM1176JZF_S Technical Reference Manual [14].

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.

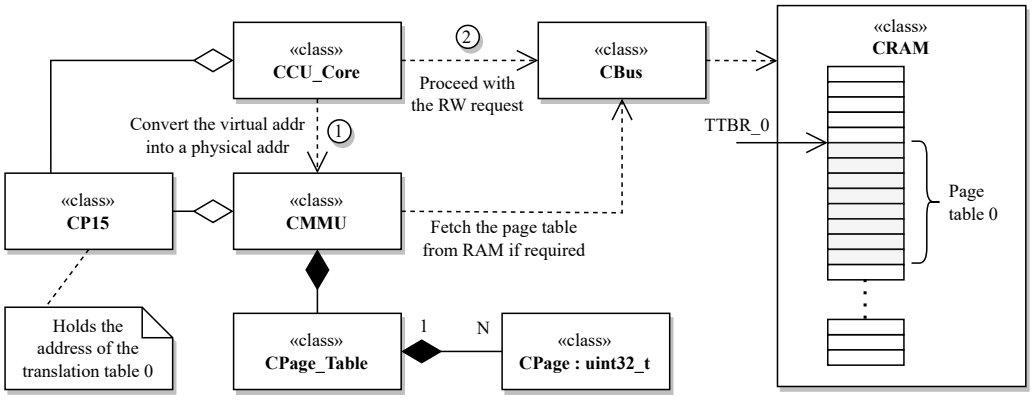


Figure 6.35: Utilization of the MMU when processing a RW request

Placing an emphasis on the Memory Management Unit, Figure 6.35 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

[17] 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.

entire page table from RAM and stores it within its private data structure. This approach eliminates the need to repeatedly read it with 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

As shown in Figure 6.36, the implementation of a first-level page table can be tacked as an array of 4096 classes called CPage[18]. These classes serve as an abstraction for accessing individual fields of individual pages, where each page is represented as a 32-bit value.

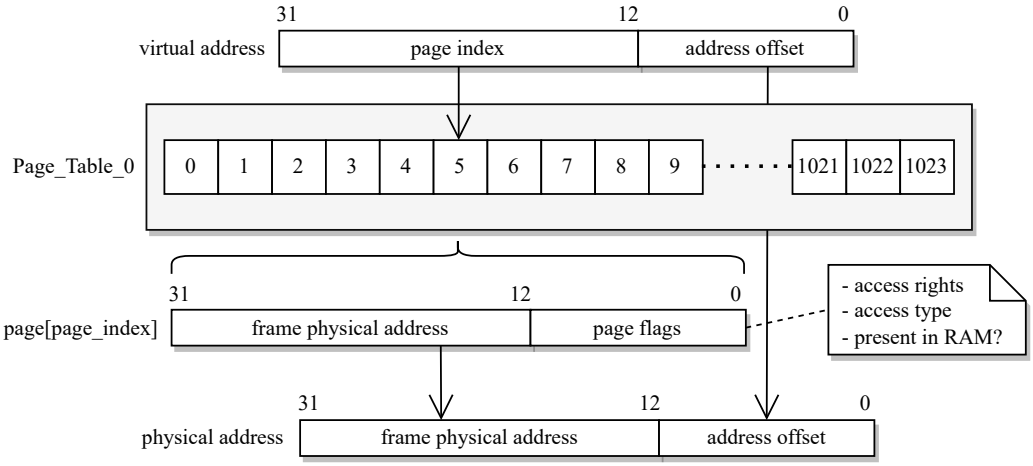


Figure 6.36: Structure of the first-level page table

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 [19]. 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 does not have to go through the translation process again, which enhances performance. When a request is made to invalidate the TLB through the *System Control Co-processor*, ZeroMate clears this associative data structure to prevent invalid addressing across various virtual address spaces.

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

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

6.3.4.6 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.

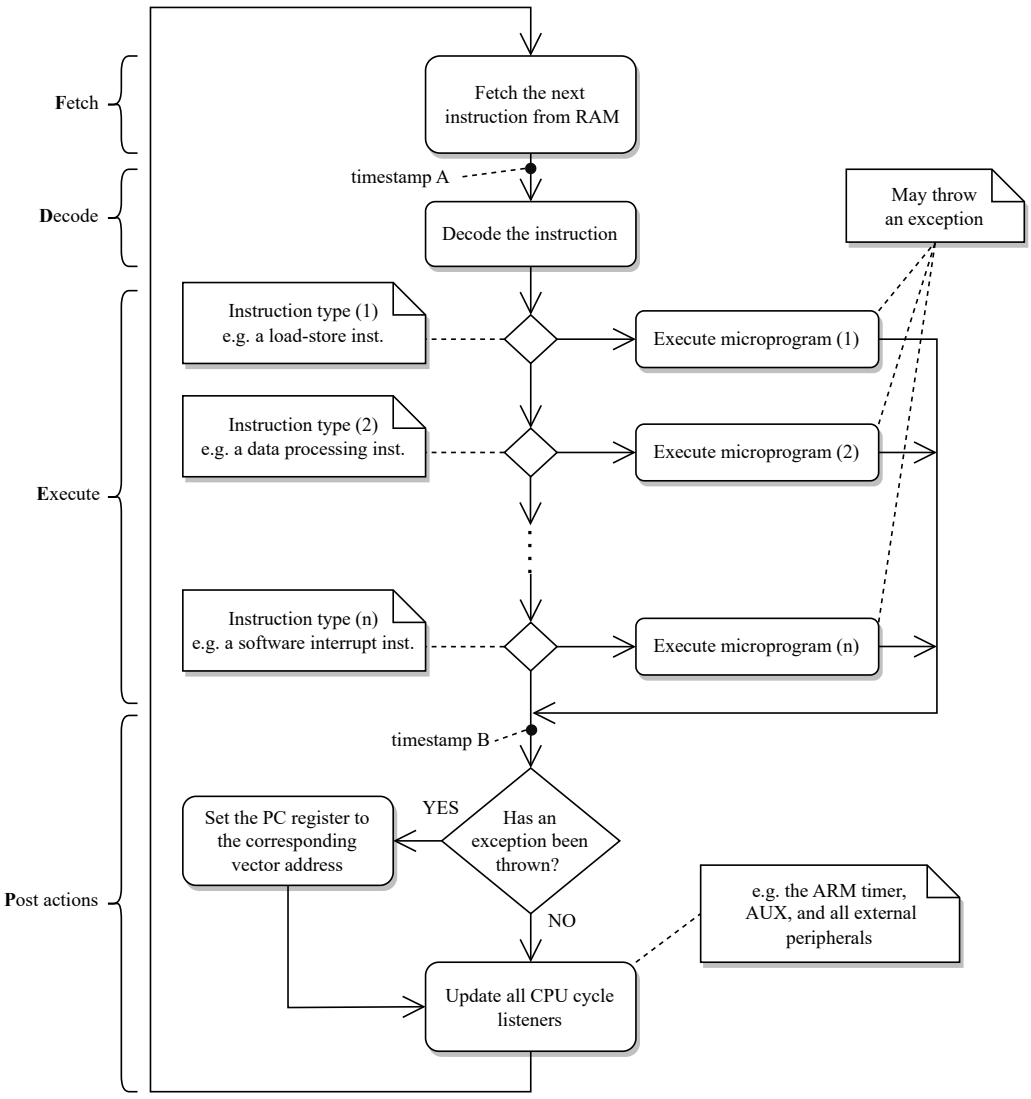


Figure 6.37: Execution loop of the emulated CPU[20]

As shown in Figure 6.37, the microprograms are called within an infinite loop known as the execution loop, which fetches the next instruction from RAM, decodes

[20]The timestamp points serve as markers for performance measurement, which is further discussed in Chapter 9.

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 6.3.4.3, 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 was 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 6.3.3.2, prior to moving on to the next instruction, the CPU informs all peripherals subscribed as system clock listeners about the number of CPU cycles required to execute the last 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.

6.3.5 Co-processors

As illustrated in Figure 6.38, **ZeroMate takes into account two co-processors**, 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.

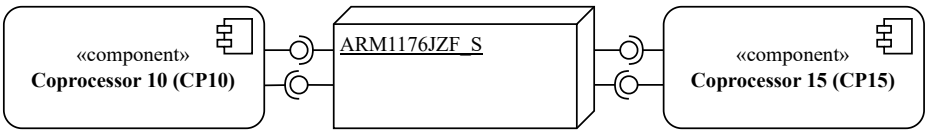


Figure 6.38: Component diagram of the CPU and its co-processors

As shown in Table 3.5, the CPU communicates with individual co-processors through three types of instructions - *data transfer*, *register transfer*, and *data operation*. When the emulated CPU detects a co-processor instruction, it does not perform any further analysis but promptly delegates it to the appropriate co-processor, determined by the ID encoded within the instruction itself. The co-processor then internally performs decoding and execution of the instruction using a technique similar to that used by the emulated CPU.

Although the ZeroMate emulator currently accommodates only two co-processors, its overall design allows for a seamless integration of additional co-processors in the future, should the need arise. As shown in Figure 6.39, the CPU maintains a collection of available co-processors, all of which implement the same interface, through which they are controlled by the CPU.

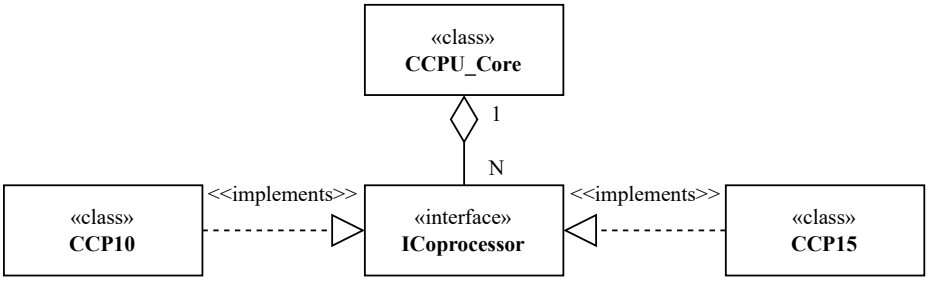


Figure 6.39: Co-processor hierarchy

6.3.5.1 Co-processor 15

As stated previously, the *System Control Co-processor* 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 co-processors, and so on [21]. As far as ZeroMate is concerned, its emulation efforts are directed primarily towards the registers listed in Table 6.1.

[21] The list of all registers along with their functions can be found detailed in Chapter 3 of the ARM1176JZF_S Technical Reference Manual [14].

Table 6.1: List of emulated CP15 registers

Primary register	Secondary register	Index	Description
C1	C0	0	Control Register (see Table 6.2)
		2	Co-processor Access Control Register
C2	C0	0	Translation Table Base Register 0 (TTBR_0)
		1	Translation Table Base Register 1 (TTBR_1)
		2	Translation Table Base Control
C8	C7	0	Invalidate unified TLB unlocked entries

Other frequently used CP15 registers are implemented solely 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 current capabilities.

Table 6.2: List of emulated flags of the CP15 control register

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, adjust their designated actions accordingly. This hierarchical structure is visualized in Figure 6.40.

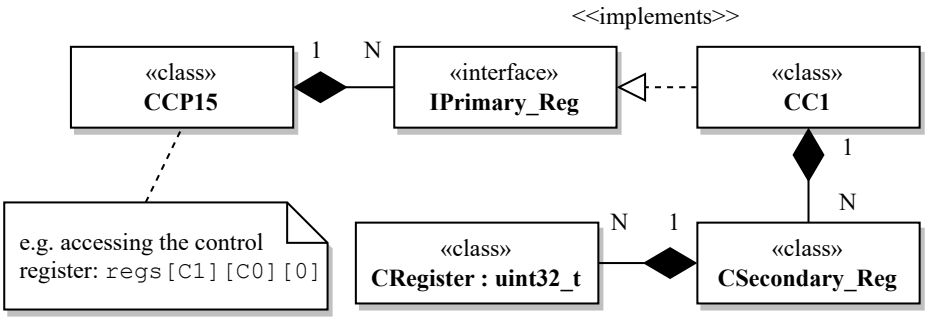


Figure 6.40: Design of the co-processor 15 register hierarchy (uses primary register C1 as an example)

6.3.5.2 Co-processor 10

Co-processor 10 allows users to work with single-precision (32-bit) floating-point numbers. The design of this co-processor closely resembles that of the CPU, as illustrated in Figure 6.37. 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 ARM Architecture Reference Manual [11].

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 [36]. This conversion can be accomplished through the helper member function shown in Listing 6.8.
2. Carry out the floating-point operation.
3. Store the result back as a `uint32_t` using the same technique demonstrated in Listing 6.8.

Source code 6.8: Conversion between `uint32_t` and `float`

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

6.4 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]. This concept is illustrated in Figure 6.41.

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.

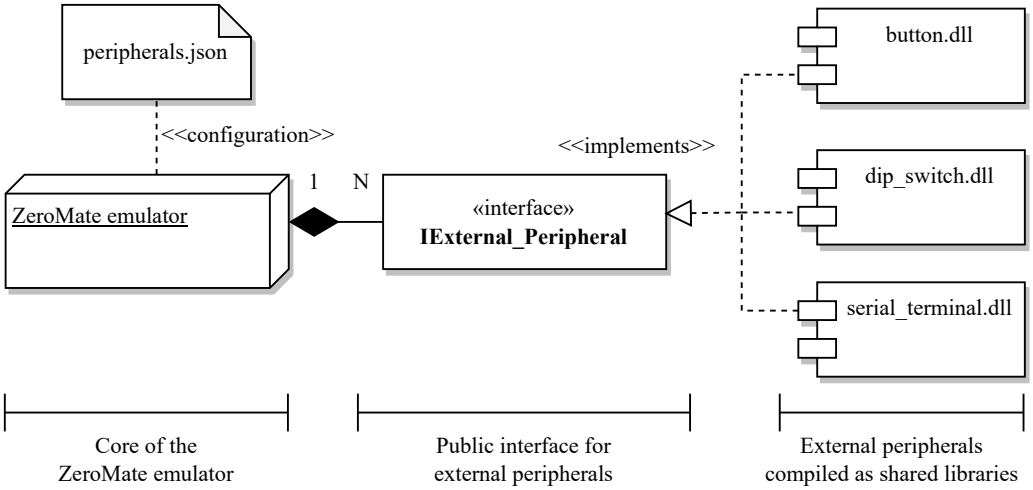


Figure 6.41: External peripheral interface

6.4.1 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].

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

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

Whenever the state of a GPIO changes, the emulator iterates over 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 are 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.

6.4.2 Configuration

Upon startup, ZeroMate attempts to read a single configuration file in the JSON format that details the connections of external peripherals. Table 6.3 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 6.3: 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 6.42 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.

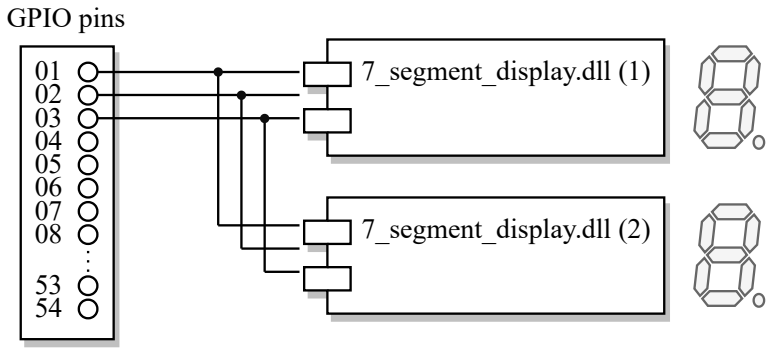


Figure 6.42: Illustration of connection of two parallel 7-segment displays

6.4.3 Examples of External Peripherals

ZeroMate comes pre-packaged with several external peripherals, many of which mirror those found on the `DPP-01` board [37], which was designed for educational purposes within the KIV/OS class taught at the University of West Bohemia. Table 6.4 enumerates the external peripherals developed alongside the emulator.

Table 6.4: 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
`serial_terminal.dll`	Used for the `UART` communication
`logic_analyzer.dll`	Visualization of `GPIO` pins' state over time

6.4.3.1 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* [33], 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 6.3.3.9.

6.4.3.2 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 the previously mentioned UART or I^2C.

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, which can prove to be valuable when observing synchronous types of communications.

6.5 Logging System

The ZeroMate project implements a custom logging system, which can be utilized both within the emulator's core and 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, which is visualized in Figure 6.43, enables the implementation of multiple loggers for various purposes, such as logging to a file, console, or graphical user interface.

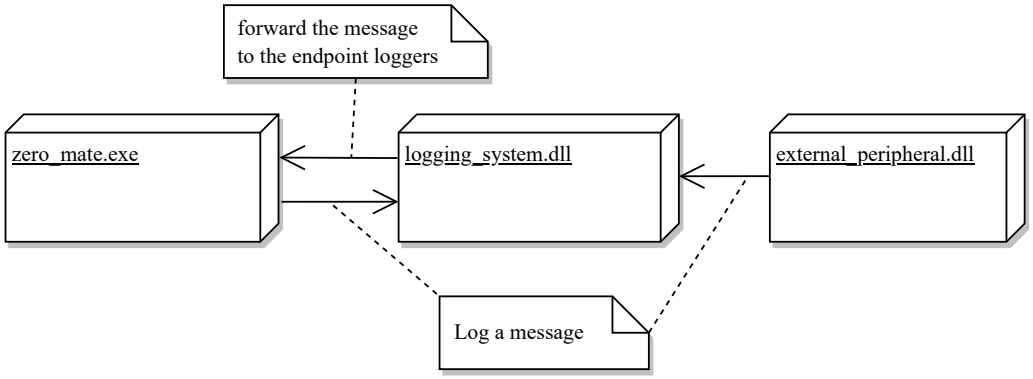


Figure 6.43: Utilization of the logging system throughout the project

[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, shown in Figure 6.44, 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.

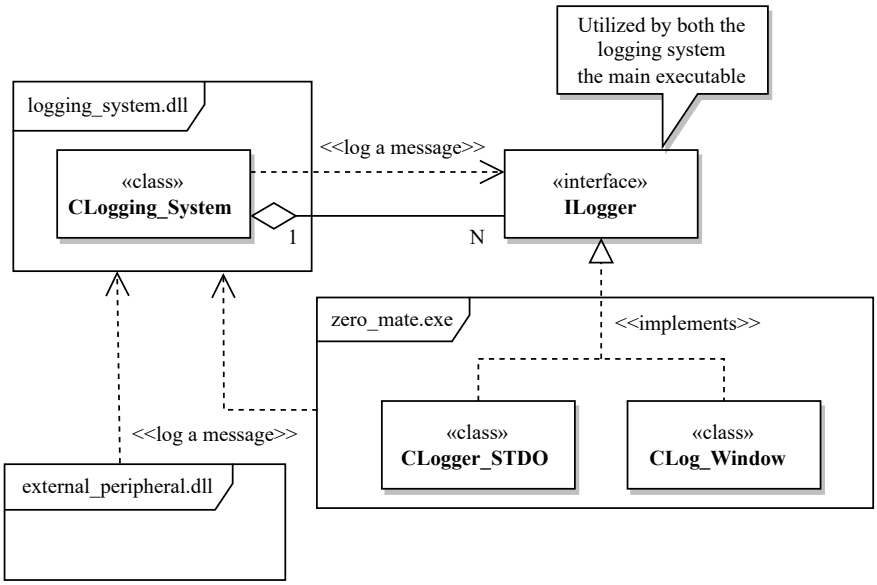


Figure 6.44: Hierarchy of endpoint loggers utilized by the logging system

6.6 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. It consists of a set of hierarchically structured windows, each displaying information about distinct aspects of the emulator's core.

Figure 6.45 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*.

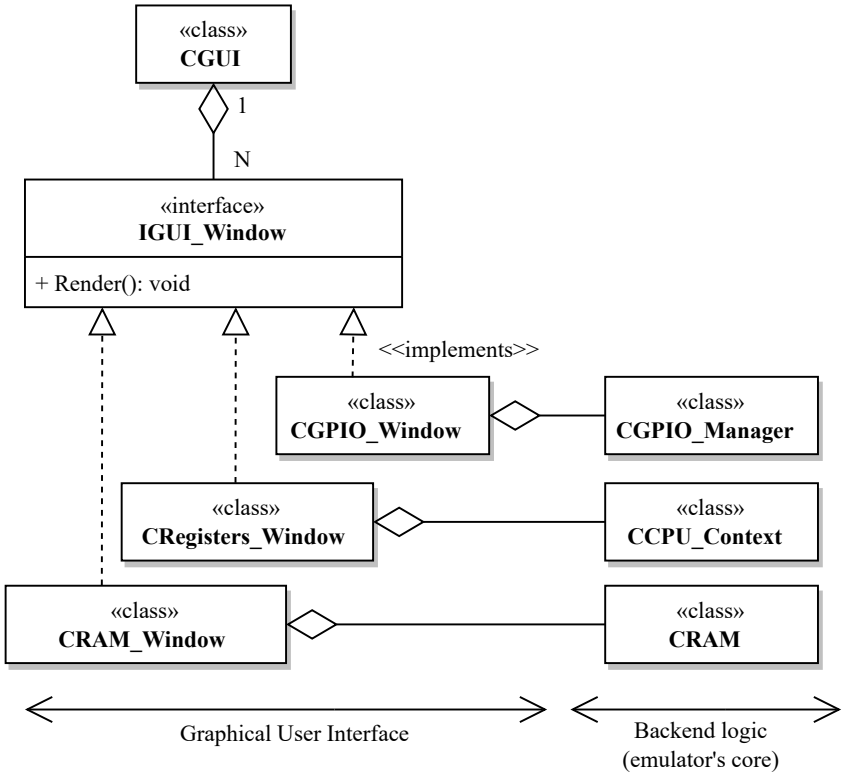


Figure 6.45: Structure of GUI windows

From Figure 6.45, 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, co-processors, 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 and 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` [38]. Several screenshots of the final emulator, including external peripherals, can be viewed in Figures 12.2, 12.3, and 12.4, which are included in the attachments.

Development Process 7

The development process aimed to establish an environment and employ techniques for both creating and testing a Raspberry Pi Zero emulator that aligns with the design concepts outlined in Chapter 6.

7.1 Programming Language

As emphasized earlier in Section 5.3, selecting the appropriate programming language is crucial since the emulator's speed is a critical factor. As a result, the use of languages such as `Java` [1] or `Python` [2] may not be suitable for this project, as they introduce an additional layer of runtime abstraction, which typically results in slower execution.

For these reasons, **C++, with its modern standards, stands out as a strong candidate, as it offers high-level abstraction, commonly found in most modern programming languages, while maintaining low-level efficiency**. One disadvantaged of choosing C++ is that it does not come with a package manager, which may render handling third-party dependencies more challenging. Nevertheless, the author believes that its speed and portability outweighs its potential disadvantages.

To mitigate common pitfalls in C++, the source code is written following the guidelines outlined in the *C++ Best Practices* book, authored by Jason Turner [39].

7.2 Project Setup

The entire project is maintained on `GitHub` [40], which is the world's largest hosting platform for open source projects. It serves as the primary platform for reporting issues, suggesting new features, and accepting public contributions.

[1] In the case of `Java`, it does not run directly on the CPU; instead, its execution is carried out through the `Java Virtual Machine`, often abbreviated as `JVM`.

[2] `Python` is an interpreted programming language, meaning it is not compiled down to machine code. It relies on its runtime interpreter to carry out the execution, which similar to `Java`, may adversely impact its performance.

The overall structure of the ZeroMate project is detailed in Figure 12.1 in the attachments.

7.2.1 Third-party Libraries

Using `Git submodules`, upon cloning the project, **all third-party dependencies are automatically fetched from `GitHub`**, eliminating the need for their manual installation. In particular, ZeroMate takes advantage of the cross-platform libraries listed in Tables 7.1, 7.2, 7.3, and 7.4.

Table 7.1: List of third-party backend dependencies

Library	Description/purpose
`capstone`	Disassembly framework for binary analysis
`demumble`	Demangling compiler symbols (based on `LLVM`)
`dylib`	Loading dynamic libraries (Windows, Linux, and MacOS)
`elfio`	Parsing `ELF` files
`fmt`	Modern text formatting library
`json`	Parsing `JSON` files
`magic_enum`	Static reflection for `C++` enumerations

Table 7.2: List of third-party frontend dependencies

Library	Description/purpose
`IconFontCppHeaders`	Set of header files and classes for using icons
`glew`	OpenGL extension library
`glfw`	`API` for creating windows
`stb`	Window icon handling
`imgui`	Immediate mode `GUI` library for `C++`
`imgui-filebrowser`	File browser implementation for imgui
`imgui_club`	Hex editor & memory viewer for imgui
`implot`	Immediate mode plotting library

Regarding the graphical user interface, another considered approach was to utilize frameworks and libraries like `Qt`, `GTK`, or `wxWidgets`.

However, the author believes that for rapid prototyping and experimentation with graphical user interfaces, an immediate mode library such as `imgui` proves to be more flexible.

When selecting a third-party library, it was crucial to ensure that it has cross-platform support, ensuring that the emulator would not be confined to a single operating system.

Table 7.3: List of third-party testing dependencies

Library	Description/purpose
`googletest`	C++ testing and mocking framework

Table 7.4: List of third-party documentation dependencies

Library	Description/purpose
`doxygen-awesome-css`	CSS theme for Doxygen HTML-documentation

7.2.2 Build System

ZeroMate employs `CMake` as its primary build system [41], allowing it to be developed in various modern integrated development environments, or `IDE`s, such as CLion, VSCode, and Microsoft Visual Studio.

Moreover, it comes with a `CMakePresets.json` file, which can be utilized to preconfigure `CMake` for different generators and compilers [3]. `CMake` also handles the building and linking of all third-party dependencies, ensuring a smooth project setup.

7.2.3 Continuous Integration

Using `GitHub Actions`, **ZeroMate leverages its custom set of `CI` pipeline configurations** that are automatically triggered whenever a change is integrated into the project. The primary goal is to automatically verify that the project successfully

[3] This file includes configurations for generators such as `Unix Makefiles`, `Ninja`, and `Visual Studio 17 2022`, along with compilers like `gcc`, `clang`, and `cl`.

builds and runs on all three major operating systems. The result of a pipeline run is automatically reported and reflected in the top-level `README.md` file, providing users with an overview of the current state of the project.

7.2.4 Testing

Following a successful build of the project, the next step of **each pipeline involves running a set of regression tests** to confirm that they all still pass. Additionally, the pipeline automatically uploads the result to the Codecov website, presenting users with a detailed overview of the overall project code coverage [42].

7.2.5 Documentation

ZeroMate also features a dedicated pipeline configuration to **automatically generate a Doxygen documentation** directly from the source code, which is then automatically uploaded to `GitHub Pages` [38]. Not only does it allow users to browse through individual functions, classes, and namespaces, but it also provides insight into various dependencies through automatically generated class diagrams.

7.3 Branching Strategy

The ZeroMate development follows the branching strategy illustrated in Figure 7.1. In this approach, individual features, such as the implementation of different components, are developed in their designated *feature branches*. These branches are consistently merged into the *development branch*, which, when stable enough, is then merged into the *main branch*. Once a reasonable number of features have been incorporated into the main branch, a new release is published.

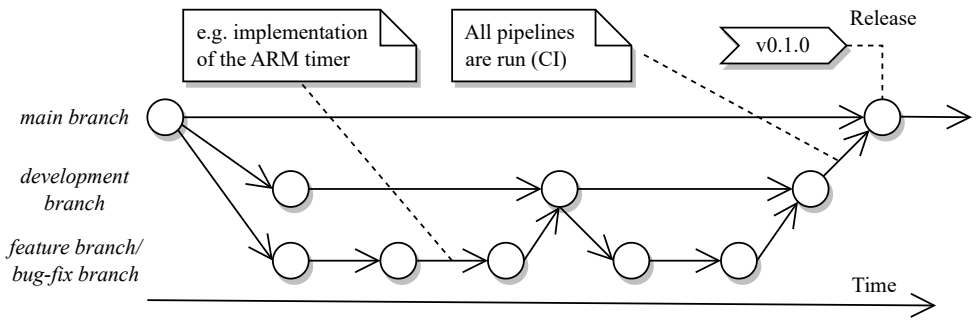


Figure 7.1: Branching strategy of the ZeroMate project[4]

In addition to release notes, each release includes the source code and a pre-built executable for both Linux and Windows, along with their respective `SHA256` checksums, providing a means to verify their authenticity.

7.3.1 Versioning

Each release is tagged with its version in the `vX.Y.Z` format. The value of `X` increments whenever there is a change in the public interface, such as modifications to the `IExternal_Peripheral` interface, which serves as the access point for all external peripherals. The value of `Y` increases whenever a new feature is introduced, and `Z` increases whenever a bug is fixed.

[4]There is also an additional branch that runs in parallel with the *main branch*, which is used by `GitHub Pages` to host the Doxygen documentation [38].

Testing 8

As illustrated in Figure 8.1, the **testing strategy for this project was segmented into various tiers based on granularity**. The core of the emulator underwent rigorous testing through an extensive suite of unit tests, addressing its fundamental yet crucial functionalities that other parts of the system heavily rely on. Functional testing integrated different components of the emulator working alongside to execute specific tasks, such as blinking an LED using a timer interrupt or scheduling processes running in userspace. The final phase of testing, system testing, was conducted by students enrolled in the KIV/OS class to assess its overall usability in practice.

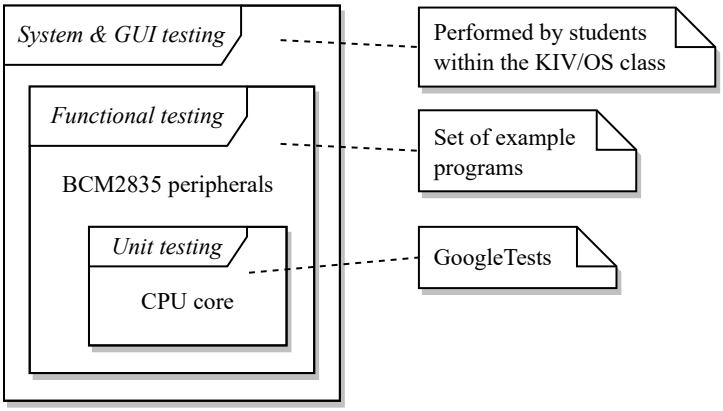


Figure 8.1: Testing strategy leveraged in the ZeroMate project

8.1 Unit Testing

The core of the emulator was developed with unit testing in mind, meaning they were utilized from the very beginning of the project development [1]. Their primary purpose is to **test the correct execution of individual ARM instructions**, including setting the CPU flags, which are shown in Figure 3.6, switching between CPU

[1] This strategy is sometimes referred to as test-driven development.

modes, and accessing the main memory. Additionally, they are used to test out the correct execution of floating-point instructions, the emulation of which is described in Section 6.3.5.2.

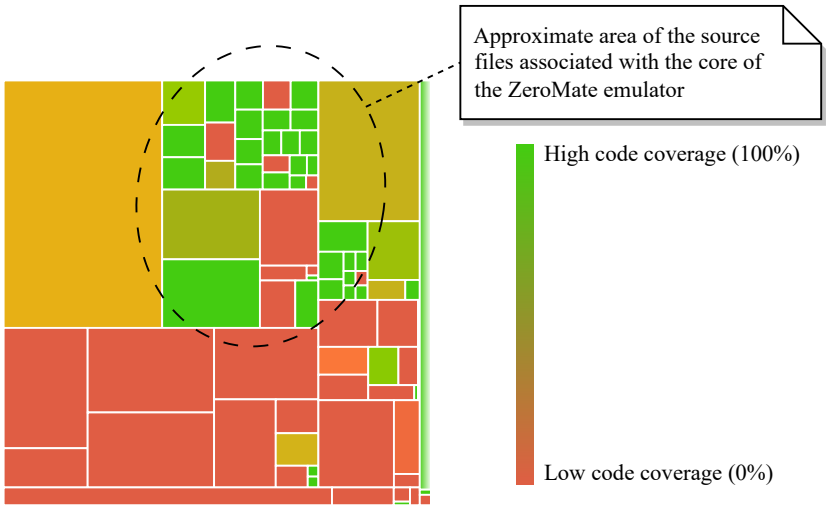


Figure 8.2: Code coverage of individual source files (exported from Codecov [42])

Each block shown in Figure 8.2 corresponds to a single `.cpp/.hpp` file of the project. The size and color of each block signify the number of statements and the coverage, respectively. Notably, smaller files, often indicative of individual ARM instructions, exhibit higher test coverage compared to larger files, which are typically associated with peripheral implementations. **In total, over 500 unit tests cover approximately 78% of the emulator's core**.

As discussed in Section 7.2.3, unit tests are also used as regression tests in the `CI` pipelines, ensuring that new changes do not break previously functioning code.

8.2 Functional Testing

Functional testing aimed at testing the `BCM2835` peripherals that work in conjunction with the emulator's core. **This level of testing was carried out through a series of example scenarios that progressively increase in complexity**. These scenarios were specifically designed for operating system development, beginning with fundamental tasks such as blinking an `LED`, advancing to interrupt handling, utilizing different `BCM2835` peripherals, and ultimately, managing the execution of different processes running in userspace [2].

[2] Each example was cross-compiled and linked to a single `ELF` file, which could then be loaded directly into the emulator as discussed in Section 6.1.

The final example used to assess the emulator's functionality was the KIV-RTOS project [2], which is a real-time operating system developed for educational purposes at the University of West Bohemia.

> The emulator ships with all test examples, allowing users to experiment with them prior to creating their own applications. Each example is accompanied by a brief description and a concise demo in the `gif` format, demonstrating the exact functionality as well as its intended usage.

There are a total of 21 such examples that can be found in the ZeroMate project, some of which are briefly described further below.

01 - factorial_recursive

This example marks a transition from unit testing to functional testing. Its primary objective is to test function calls, implicitly utilizing the stack along with several essential instructions. The kernel recursively calculates 7! and subsequently halts the `CPU`. The outcome of this operation, 5040, is located in register `r0`, which is employed to store the return value from a function.

08 - LED_toggle_sos_signal

This example utilizes `GPIO` pin `47`, the built-in `LED`, to emit the `SOS` signal in an infinite loop. It employs an active sleep function to introduce a delay before changing the pin's state [3].

11 - timer_and_GPIO_interrupt

This example illustrates the utilization of multiple interrupts originating simultaneously from various sources. One interrupt occurs periodically from the ARM timer, while the second one originates from a `GPIO` pin, such as an external button. The timer interrupt toggles an `LED` connected to pin `47`, and the `GPIO` interrupt toggles an `LED` connected to pin `48`.

13 - context_switch_monitor

This examples showcases the use of context switching, which an essential part of any operating system with a preemptive scheduler [4]. The kernel creates four processes that take turns utilizing the `CPU` time. Whenever it is time to switch to

[3] An active sleep function is a method that suspends ongoing execution by continuously looping, essentially "not doing anything", until the awaited event occurs or the specified time elapses. In certain scenarios, this approach can be more efficient than using interrupts, especially when the sleep period is anticipated to be relatively short.

[4] In an operating system, preemption refers to temporarily interrupting a currently executing task, such as when a high-priority task needs to be executed, with the intention of resuming it at a later time.

another process, as determined by the ARM timer, the CPU stores the context of the current process and loads the context of the next one. Each process is assigned a unique number, which it continuously prints out to the debug monitor, which is discussed in Section 6.3.3.4.

16 - paging_userspace

This example demonstrates the management of virtual address spaces for user-space processes. During system initialization, the kernel bootstraps the following user processes, which are subsequently scheduled in a preemptive manner. (1) An idle process that performs no active tasks but yields control. (2) Process 1, which continuously writes digits 0 through 9 to a 7-segment display. (3) Process 2, which leverages the random number generator (TRNG) along with the debug monitor to print out random numbers.

19 - I2C

This example demonstrates the use of an SSD1306 OLED display that is controlled over I^2C. There are two userspace processes bootstrapped within this example. (1) A "dummy" process that periodically blinks an LED. (2) A process that utilizes TRNG to generate random numbers, serving as indices for an array of string messages. These messages are then sent over I^2C to be displayed on the SSD1306 OLED display.

20 - UART_game

The objective of this example is to showcase bidirectional interrupt-driven UART communication. This is achieved through a simple number guessing game played between the kernel and the user. In the game, the user thinks of a number, and the kernel guesses the number by inquiring whether it is greater than a specific value. The user interacts with the kernel by responding with 'y' (yes) or 'n' (no) to each question posed.

8.3 System Testing

After the emulator was developed, it was used as an educational tool within the KIV/OS class. Students, effectively playing the roles of end users, were encouraged to use it during practical sessions and as a debugging tool for their final semester projects.

The objective was to assess the usefulness and intuitiveness of the emulator across various aspects, including the installation process, navigation through the GUI, and the execution of their applications. To collect comprehensive feedback, students were given a survey that included various questions regarding the overall

usage of the emulator. However, due to the limited number of participants, the results lack statistical interpretability. One student included the following in their overall evaluation.

"Very usefull for debugging not computionally intensive projects / functions, the best feature being the ability to view RAM and disassembled code. Testing code with tons of instructions or loops with many iterations would be very impractical, since the emulator runs slow for that (but it's an emulator, this is kinda expected I guess). The main downside was the not very well functioning UART emulation. These two problems were the reason I left development for the emulator as well as real hardware and continued working only on the device itself (used UART logging for debugging)."

Regarding the reported issues with the `UART` interface, a comprehensive analysis was conducted, revealing that the root cause of the problem stemmed from the firmware itself and not from within ZeroMate, as evidenced by its manifestation on a `HW` board as well.

Performance Evaluation

9

The significance of the emulator's speed has been emphasized repeatedly throughout the thesis. As a result, the following experiment was conducted to assess the emulator's capability in terms of its emulation speed.

9.1 Experiment Parameters

As an input for this experiment, the KIV-RTOS operating system was chosen since it was the targeted system for emulation [2]. It was considered complex enough to offer a general understanding of the emulator's capabilities. However, it is essential to acknowledge that the results may vary depending on what application is used, as each may differ in the quantity and types of individual instructions.

During the run of KIV-RTOS, the **time taken to emulate each instruction was continuously measured**, starting from the point where the instruction is being decoded to the point when its execution is completed. These points are marked in Figure 6.37 as *timestamp A* and *timestamp B*. The goal was to focus solely on the emulation of individual ARM instructions, excluding other actions such as updating peripherals or checking for pending interrupts [1].

The experiment was conducted on the Lenovo ThinkPad P50 laptop, running the Windows 10 operating system. The laptop is equipped with an Intel(R) Core(TM) i7-6820HQ `CPU` running at `2.70GHz` and `16GB` of `RAM`. In terms of `C++`, the `std::chrono::high_resolution_clock` class was employed for measuring execution speed in nanoseconds [43].

[1] As all peripherals are updated synchronously, an inefficient implementation of a peripheral can potentially compromise overall performance.

9.2 Emulation Speed

Throughout the run, a total of **426,922,064 instructions were executed**. The average emulation speed for each instruction is illustrated in Figure 9.1. Taking into account both the frequency of individual instruction types and their respective average execution times, the overall performance was calculated to be **4.84 mega instructions per second** using the following formula[2].

$$\text{Emulation speed} = \frac{10^3}{\frac{1}{t}\sum_{i=1}^{n}(o_i * s_i)}[Minst/s],$$

where t represents the total number of instructions, n is the number of different instruction types, o_i is the absolute number of occurrences of instruction i, and s_i denotes its average execution time in nanoseconds.

Instruction type	Relative occurrence [%]	Emulation speed [Minst/s]
Software interrupt	0.014	0.100
Block data transfer	0.436	0.833
Store return state	0.033	1.282
Return from exception	0.033	2.041
Co-processor register transfer	0.001	2.500
Single data transfer	**41.705**	**3.846**
Halfword data transfer	0.020	3.846
Multiply long	< 0.001	3.846
Change processor state	0.033	4.000
Data processing	**41.061**	**5.556**
Program status register transfer	0.007	5.882
Extend	5.388	8.333
Count leading zeros	< 0.001	8.333
Multiply	0.003	9.091
Branch	**9.940**	**10.000**
Signed multiply	0.002	10.000
Branch and exchange	1.214	16.949
NOP	0.100	31.250

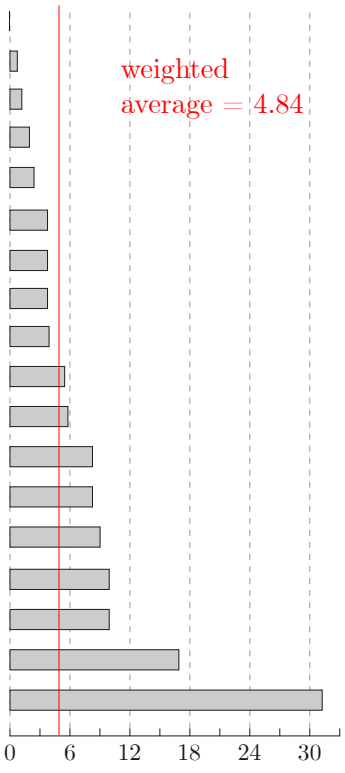


Figure 9.1: ZeroMate's emulation performance of KIV-RTOS

[2]This process was repeated multiple times, consistently yielding results in the range of 4 to 5 mega instructions per second.

Figure 9.1 also highlights the top 3 most executed instructions - *Single data transfer*, *Data processing*, and *Branch*.

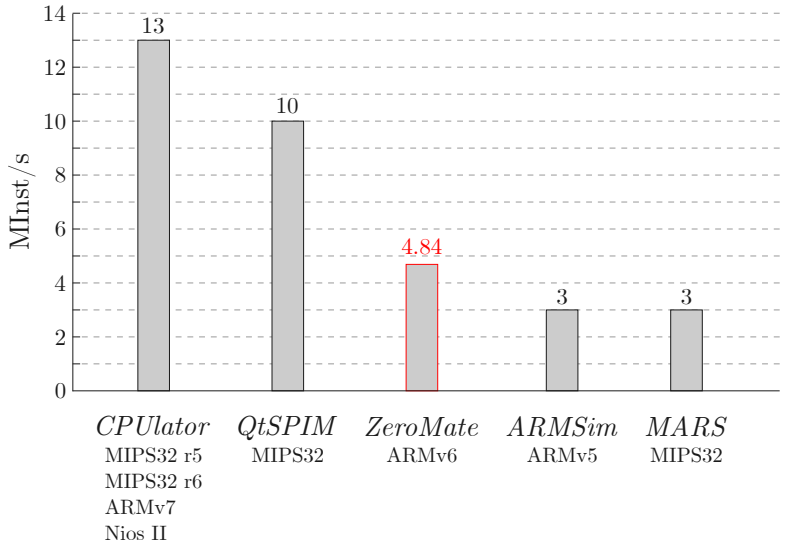


Figure 9.2: Performance comparison of different emulators

Figure 9.2 shows the performance comparison of ZeroMate to other emulators along with a list of architectures they support. This comparative analysis is sourced from the CPUlator website [23]. It is noteworthy that among these emulators, **ZeroMate distinguishes itself as the only emulator supporting an `MMU`**. When activated, the `MMU` is inevitably utilized in every memory access operation, leading to a notable reduction in the overall emulation speed.

However, it is crucial to note that the source lacks information on how these statistics were obtained[a]. Therefore, the data presented in Figure 9.2 is included merely as an intriguing observation and should be approached with caution, as it may not be entirely reliable for drawing definitive conclusions.

[a] Measurements might have been taken on different machines with varying parameters. Additionally, emulators may not universally support the same features, which could impact their overall performance.

9.3 Performance-affecting Instructions

To accurately assess the overall impact of different instruction types on emulation performance, both the emulation speed and the frequency of occurrences must be considered. For instance, in Figure 9.1, it can be observed that the *Software interrupt* instruction exhibits the poorest emulation performance, meaning it takes a significant amount of time to emulate. However, its execution frequency is lower

compared to other instructions, mitigating its substantial negative impact on overall performance.

As far as KIV-RTOS is concerned, Figure 9.3 enumerates the top 8 instructions that affect the overall emulation speed. This analysis could potentially serve as an indication of which instruction types should be prioritized for further optimization.

Instruction type	**Impact on performace [avg. execution time [s] * absolute count]**
Single data transfer	46.293
Data processing	31.553
Branch	4.244
Extend	2.760
Block data transfer	2.232
Software interrupt	0.595
Branch and exchange	0.306
Store return state	0.110

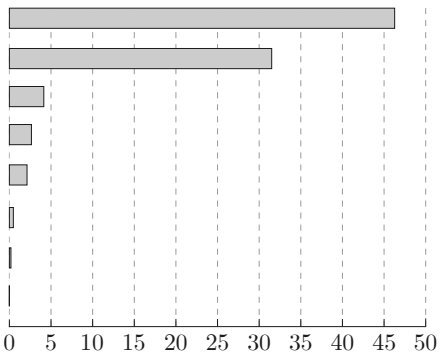


Figure 9.3: Top 8 most performance-affecting instructions[3]

It can be noticed that both the *Single data transfer* and *Block data transfer* instructions exert a significant impact on overall performance, which is unsurprising, given that ARM, as mentioned earlier, operates on a load-store architecture, which requires both operands to be present in registers, leading to increased memory traffic. This finding aligns with the statement made in Section 6.3.1 - *"It is reasonable to conclude that optimizing peripheral access efficiency might be crucial for emulation speed."*

[3]The data collected for this experiment is presented in Table 12.2, which can be found in the attachments.

Potential Improvements

10

Implementing a complete Raspberry Pi Zero emulator is a complex task that poses numerous challenges. Throughout the thesis, some of ZeroMate's limitations were highlighted and suggested as future improvements. From a critique yet objective point of view, this chapter lists out some of the limitations and ideas for further enhancements.

10.1 Current Limitations

RAM Implementation

The existing `RAM` implementation employs a continuous array of bytes. However, since not all of `RAM` is immediately utilized, a more resource-friendly approach would involve dynamically allocating individual blocks of memory as they are addressed by the emulated `CPU`.

Implementation of CPU Exceptions

ZeroMate implements `CPU` exceptions as `std::runtime_error`, which can potentially overlap with genuine exceptions that may arise at runtime due to a bug in the source code. Consequently, the emulated `CPU` may face difficulty distinguishing between an actual exception and an emulated `CPU` exception.

Support for a Second-level Page Table

The current implementation of ZeroMate only permits the use of a first-level page table, with its entries pointing directly to physical frames. This limitation hinders users from further managing the address space through nested paging.

Implementation of All ARM Instructions

There are still a few ARMv6 instructions that are yet to be implemented, such as the `SETEND` instruction, responsible for changing the current endianness, or the `PLD` instruction, designed to minimize cache-miss latency by preloading data into

cache before it is accessed. However, the majority of commonly used instructions have already been fully implemented and rigorously tested, as evidenced by the successful emulation of KIV-RTOS [2].

Asynchronous Update of Peripherals

In the current implementation, all peripherals are updated synchronously immediately after an instruction is executed, temporarily halting the execution of the next instruction. A more effective approach, mirroring real hardware, would involve updating them asynchronously in a separate thread. This way, the `CPU` could continue its execution without any delay.

Thorough & Accurate Peripheral Implementation

It has been noted that the implementation of certain `BCM2835` peripherals was simplified to illustrate a general approach of how they can be used, rather than delving into every single piece functionality they are capable of. Consequently, some of the more advanced features might be omitted or deviate from the real hardware specifications. Additionally, it is important to mention that ZeroMate does not currently support all of the `BCM2835` peripherals.

Code Disassembly

As highlighted by the students, despite being segmented into distinct function blocks, navigating through the current program disassembly may become challenging as the codebase grows in size.

10.2 Future Enhancements

Enhancing the functionality of the system also requires addressing the issues previously outlined in Section 10.1.

`CLI` Mode

ZeroMate functions as a `GUI` application. Nevertheless, there could be an advantage in introducing a `CLI` mode, specifically tailored for deployment in continuous integration environments as a testing utility for embedded applications.

Socket Communication

A notable improvement could entail the integration of inter-process communication, or `IPC`, facilitating bidirectional communication between multiple instances of the emulator, each effectively functioning as independent computing unit. This feature would empower users to emulate a distributed environment and could poten-

tially serve as a practical platform for showcasing a variety of distributed computing techniques, such as the farmer-worker scheme or MapReduce.

WebAssembly

Taking inspiration from CPUlator [23], the author believes that it would be a welcomed addition to introduce the capability of compiling the application into WebAssembly, which would enable users to access it directly from a browser without the need for any installation.

Further Modularization

Currently, all the `BCM2835` peripherals are an integral part of the final executable. It would be advantageous if they were compiled as separate dynamic libraries, which would enable them to be replaced if needed, allowing users to configure their own system. For instance, users could upgrade their `RAM`, replace timers, and so on. Moreover, developing them independently as separate projects, akin to external peripherals, facilitates seamless integration of new peripherals into the system without the need to recompile the core itself.

Support for Analog Pins

Currently, ZeroMate supports only digital pins. However, its applications would be more extensive if the emulator also supported both analog and digital inputs.

Conclusion 11

The goal of this thesis was to explore and evaluate potential approaches to emulate Raspberry Pi Zero, representing one of the most widely embraced architectures for embedded devices. Beyond educational purposes, the compelling reasons for this undertaking included compensating for a physical board, which may not always be readily available, and experimenting with software without concerns about damaging real hardware.

To provide context, the paper initially explored the general concept and classification of various computer architectures before delving into some of the distinctive characteristics specific to the ARM architecture itself. The thesis then proceeded to scrutinize existing solutions that could potentially emulate KIV-RTOS [2], an educational real-time operating system, which eventually served as the primary source for testing. However, the study ultimately concluded that these solutions might not be suitable for such a purpose, primarily due to their limited support for some of the more advanced system-related features, such as paging, which is indispensable in the development of an operating system.

As a result, the paper shifted its focus to an abstract design of a custom modular Raspberry Pi Zero emulator with the aim to address these limitations and provide a user-friendly interface that would enable seamless debugging of embedded applications targeting the Raspberry Pi Zero platform. One of the design goals was to enable communication with third-party external peripherals that could be developed independently of the core itself, allowing users to emulate an environment with custom hardware components, such as screens, sensors, and actuators. Aligning with the design decisions, the emulator was consequently developed using modern standards of the C++ programming language. In addition to rigorous unit testing, the entire system underwent functional testing using a set of examples pertained to operating system development and practical use by students enrolled in the KIV/OS class.

The thesis concludes with a performance evaluation, proposing areas for future improvements and suggesting ideas for further enhancements, such as employing socket communication to potentially emulate distributed applications.

Bibliography

1. TEAM, Arm Editorial. *The Official History of Arm* [online]. 2023. [visited on 2023-09-25]. Available from: `https://newsroom.arm.com/arm-official-history`.
2. ÚBL, Martin. *KIV-RTOS - An educational operating system for bare-metal Raspberry Pi Zero W (BCM2835-based board)* [online]. University of West Bohemia. [visited on 2023-09-27]. Available from: `https://github.com/MartinUbl/KIV-RTOS`.
3. NOERGAARD, T. *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers*. Elsevier Science, 2005. ISBN 978-0750677929.
4. GILREATH, W.F.; LAPLANTE, P.A. *Computer Architecture: A Minimalist Perspective*. Springer US, 2012. ISBN 978-1461502371.
5. CRAGON, Harvey G. *Computer Architecture and Implementation*. 2000. ISBN 978-0521651684.
6. FURBER, S.B. *ARM System-on-chip Architecture*. Addison-Wesley, 2000. ISBN 978-0201675191.
7. JAMIL, T. *RISC versus CISC* [online]. Institute of Electrical and Electronics Engineers, 1995 [visited on 2023-11-02]. Available from: `https://ieeexplore.ieee.org/abstract/document/464688`.
8. KINGATUA, Amos. *ARM Processor vs X86: Choosing the Right Architecture for Your PC* [online]. 2023. [visited on 2023-11-02]. Available from: `https://electronicsandict.com/arm-processor-vs-x86-choosing-the-right-architecture-for-your-pc`.
9. A., Patterson David. *Computer Organization and Design Arm Edition: The Hardware Software Interface*. Morgan Kaufmann, 2016. ISBN 978-0128017333.
10. SEGARS, Simon. *Arm Partners Have Shipped 200 Billion Chips* [online]. 2021. [visited on 2023-09-26]. Available from: `https://newsroom.arm.com/200bn-arm-chips`.

11. *ARM Architecture Reference Manual* [online]. ARM. [visited on 2023-10-15]. Available from: https://documentation-service.arm.com/static/5f8dacc8f86e16515cdb865a?token=.

12. ARM. *Procedure Call Standard for the Arm® Architecture* [online]. 2023. [visited on 2023-11-16]. Available from: https://github.com/ARM-software/abi-aa/releases/download/2023Q3/aapcs32.pdf.

13. SLOSS, Andrew; WRIGHT, Chris. *ARM and Thumb Instruction Encodings (B2 APPENDIX)* [online]. ARM. [visited on 2023-09-28]. Available from: https://gab.wallawalla.edu/~curt.nelson/cptr380/textbook/advanced%20material/Appendix_B2.pdf.

14. *ARM1176JZF-S Technical Reference Manual* [online]. ARM, 2009. [visited on 2023-09-28]. Available from: https://developer.arm.com/documentation/ddi0301/h.

15. BROADCOM. *BCM2835 ARM Peripherals* [online]. Broadcom Corporation, 2012. [visited on 2023-12-03]. Available from: https://datasheets.raspberrypi.com/bcm2835/bcm2835-peripherals.pdf.

16. *BCM2835 datasheet errata* [online]. Embedded Linux Wiki. [visited on 2023-12-03]. Available from: https://elinux.org/BCM2835_datasheet_errata.

17. DURÁN, Juan Manuel. *Computer Simulations in Science and Engineering: Concepts - Practices - Perspectives*. Springer International Publishing, 2018. ISBN 978-3319908823.

18. STEVENS, Kenneth. *The Emulation User's Guide*. Lulu.com, 2008. ISBN 978-1435753730.

19. VEENSTRA, Peter; SJOERD; FRÖSSMAN, Tommy; WOHLERS, Ulf. *DOSBox* [online]. [visited on 2023-12-16]. Available from: https://www.dosbox.com.

20. DEVELOPERS, The QEMU Project. *QEMU - A generic and open source machine emulator and virtualizer* [online]. [visited on 2023-12-16]. Available from: https://www.qemu.org.

21. PORTNOY, Matthew. *Virtualization Essentials 3rd edition*. Wiley, 2023. ISBN 978-1394181575.

22. BUGNION, E.; NIEH, J.; TSAFRIR, D. *Hardware and Software Support for Virtualization*. Springer International Publishing, 2022. ISBN 978-3031017537.

23. *CPUlator Computer System Simulator* [online]. [visited on 2023-12-16]. Available from: https://cpulator.01xz.net.

24. HORSPOOL, R. Nigel; LYONS, W. D.; SERRA, M. *ARMSim#* [online]. Department of Computer Science, University of Victoria. [visited on 2023-12-17]. Available from: `https://webhome.cs.uvic.ca/~nigelh/ARMSim-V2.1/index.html`.

25. ÚBL, Martin. *KIV/OS - dodatek A - qemu* [online]. University of West Bohemia. [visited on 2023-11-07]. Available from: `https://home.zcu.cz/~ublm/files/os/practicals/0a_kivrtos_cz.pdf`.

26. COMMITTEE, TIS. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification* [online]. Linux Foundation, 1995. [visited on 2023-08-26]. Available from: `https://refspecs.linuxfoundation.org/elf/elf.pdf`.

27. COPLIEN, James O. *Multi-Paradigm Design for C++*. Addison-Wesley, 1999. ISBN 978-0201824674.

28. BALOGUN, Ghaniyyat Bolanle. *A Comparative Analysis of the Efficiencies of Binary and Linear Search Algorithms* [online]. Department of Computer Science, University of Ilorin, Ilorin, Nigeria., 2020. [visited on 2024-01-02]. Available from: `https://afrjcict.net/wp-content/uploads/2020/03/Vol13No1Mar20pap3journalformatpagenumb.pdf`.

29. KANKOWSKI, Peter. *x86 Machine Code Statistics* [online]. 2008. [visited on 2023-10-27]. Available from: `https://www.strchr.com/x86_machine_code_statistics`.

30. *The GNU C++ Library - Chapter 28. Demangling* [online]. The GNU Compiler Collection, 2008. [visited on 2023-10-01]. Available from: `https://gcc.gnu.org/onlinedocs/libstdc++/manual/ext_demangling.html`.

31. LAMIKHOV-CENTER, Serge. *ELFIO - ELF (Executable and Linkable Format) reader and producer implemented as a header-only C++ library* [online]. [visited on 2023-08-26]. Available from: `https://elfio.sourceforge.net`.

32. WEBER, Nico. *Demumble - A better c++filt and a better undname.exe, in one binary* [online]. [visited on 2023-08-28]. Available from: `https://github.com/nico/demumble`.

33. TATHAM, Simon. *PuTTY - free implementation of SSH and Telnet for Windows and Unix platforms, along with an xterm terminal emulator* [online]. [visited on 2023-10-13]. Available from: `https://www.putty.org`.

34. TORVALDS, Linus. *Linux kernel* [online]. [visited on 2023-12-05]. Available from: `https://github.com/torvalds/linux/blob/master/drivers/char/hw_random/bcm2835-rng.c`.

35. BHATTACHARJEE, Kamalika; DAS, Sukanta. *A search for good pseudo-random number generators: Survey and empirical studies* [online]. 2022. [visited on 2023-12-05]. Available from: https://www.sciencedirect.com/science/article/pii/S1574013722000144.

36. *754-2019 - IEEE Standard for Floating-Point Arithmetic* [online]. Institute of Electrical and Electronics Engineers, 2019. [visited on 2023-11-05]. Available from: https://ieeexplore.ieee.org/document/8766229.

37. ÚBL, Martin. *KIV-DPP-01 - Expansion board documentation* [online]. University of West Bohemia, 2021. [visited on 2023-08-28]. Available from: https://home.zcu.cz/~ublm/files/os/kiv-dpp-01-en.pdf.

38. ŠILHAVÝ, Jakub. *ZeroMate Doxygen documentation* [online]. 2023. [visited on 2024-02-16]. Available from: https://silhavyj.github.io/ZeroMate.

39. TURNER, Jason. *C++ Best Practices 2nd edition*. 2022. ISBN 979-8822105607.

40. ŠILHAVÝ, Jakub. *ZeroMate* [online]. 2023. [visited on 2024-02-16]. Available from: https://github.com/silhavyj/ZeroMate.

41. *CMake* [online]. [visited on 2024-01-07]. Available from: https://cmake.org.

42. ŠILHAVÝ, Jakub. *ZeroMate Codecov report* [online]. 2023. [visited on 2024-02-16]. Available from: https://app.codecov.io/gh/silhavyj/ZeroMate.

43. *std::chrono::high_resolution_clock* [online]. [visited on 2024-01-04]. Available from: https://en.cppreference.com/w/cpp/chrono/high_resolution_clock.

List of Abbreviations

A

ABI - Application Binary Interface
ACC - Accumulator
ADC - Analogue to Digital Converter
ALU - Arithmetic-Logic Unit
API - Application Programming Interface
ARM - Advanced RISC Machines
AUX - Auxiliary

B

BSC - Broadcom Serial Controller

C

CISC - Complex Instruction Set Computer
CI - Continuous Integration
CLI - Command Line Interface
CP10 - Co-processor 10
CP15 - Co-processor 15
CPI - Cycles Per Instruction
CPSR - Current Program Status Register
CPU - Central Processing Unit

E

EEPROM - Electrically Erasable Programmable Read-only Memory
ELF - Executable Linkage Format

F

FIQ - Fast Interrupt Request
FPGA - Field Programmable Gate Array
FPU - Floating-Point Unit

G

GPIO - General Purpose Input/Output
GPU - Graphics Processing Unit
GUI - Graphical User Interface

H

HDMI - High-Definition Multimedia Interface

I

I/O - Input-Output
IC - Interrupt Controller
IDE - Integrated Development Environment
IPC - Inter-Process Communication
IRQ - Interrupt Request
ISA - Instruction Set Architecture
IVT - Interrupt Vector table
I^2C - Inter-Integrated Circuit
IoT - Internet of Things

J

JVM - Java Virtual Machine

L

LCG - Linear Congruential Generator
LED - Light-emitting diode
LIFO - Last-In-First-Out
LSB - Least Significant Bit

M

MAC - MAC Unit (performs multiply-accumulate operations)
MMIO - Memory-mapped Input-Output device
MMU - Memory Management Unit
MSB - Most Significant Bit

O

OLED - Organic Light-Emitting Diode
OS - Operating System

P

PWM - Pulse Width Modulation

Q

QA - Quality Assurance

R

R/W - Read-Write
RAM - Random Access Memory
RISC - Reduced Instruction Set Computer
ROM - Read-Only Memory
RX - Receive

S

SDRAM - Synchronous Dynamic Random Access Memory
SD - Secure Digital
SHA - Secure Hash Algorithm
SIMD - Single Instruction/Multiple Data
SOS - Save Our Souls
SPI - Serial Peripheral Interface
SPSR - Saved Program Status Register
SP - Stack Pointer
SREC - Motorola S-record (file format)
STDO - Standard Output
SoC - System on Chip

T

TLB - Translation Lookaside Buffer
TRNG - True Random Number Generator
TTBR - Translation Table Base Register
TX - Transmit

U

UART - Universal Asynchronous Receiver/Transmitter
USB - Universal Serial Bus

V

VFP - Vector Floating-Point
VGA - Video Graphics Array

List of Figures

List of Tables

List of Listings

Attachments 12

12.1 Building the Project

The user has the option to either download a pre-built executable for both Linux and Windows from the latest release [40], or they can attempt to build the emulator themselves from the source code.

12.1.1 Dependencies

If they choose to build the application themselves, they are required to have the tools listed in Table 12.1 already installed on their system. The remaining third-party dependencies are then handled automatically by `CMake`.

Table 12.1: List of required dependencies for a successful build of ZeroMate

Tool	**Purpose**
`Git`	Cloning the repository from `GitHub`
`CMake`	Configuring and building the project
`C/C++` compiler	e.g. `Clang` that will be employed during the building process
Build system of choice	e.g. `Ninja`, `Unix Makefiles`, `MSVC`
`OpenGL`	Utilized by the ZeroMate's user interface (likely already installed on the user's machine)

12.1.2 Build Steps

12.1.2.1 Clone

Once all mandatory dependencies have been installed, the user can pull down the project from `GitHub` using the command provided in Listing 12.1.

Listing 12.1: Cloning the project repository

```
silhavyj@my-pc:$ git clone --recursive
https://github.com/silhavyj/ZeroMate.git
```

> It is important to use the `--recursive` option, which automatically clones all submodules used by the project.

12.1.2.2 Configuration

Once cloned and in the directory, using the command showcased in Listing 12.2, the user can inspect a predefined set of `CMake` configurations they can use to configure the project. Nevertheless, they are not obliged to adhere strictly to these configuration sets. Users have the flexibility to customize the `CMake` project according to their specific needs.

Listing 12.2: Listing of all `CMake` presets

```
silhavyj@my-pc:ZeroMate$ cmake --list-presets
Available configure presets:

"unix_makefiles_gcc_release"
"unix_makefiles_gcc_debug"
"unix_makefiles_gcc_code_coverage"
"unix_makefiles_clang_release"
"unix_makefiles_clang_debug"
"unix_makefiles_clang_code_coverage"
"ninja_clang_release"
"ninja_clang_code_coverage"
"ninja_clang_debug"
"ninja_gcc_release"
"ninja_gcc_debug"
"ninja_gcc_code_coverage"
"msvc"
silhavyj@my-pc:ZeroMate$
```

After the user has chosen their preferred combination of build system, compiler, and build type, they can configure the project by executing the command shown in Listing 12.3. In this instance, the `ninja_clang_release` preset is used as an example.

Listing 12.3: Configuring the project using a CMake preset

```
silhavyj@my-pc:ZeroMate$ cmake --preset ninja_clang_release
```

12.1.2.3 Build

After a successful completion of the previous command, the user can build the project by executing the final command as demonstrated in Listing 12.4. It is important to note that the name of the preset must match the one used during the project configuration. The `--parallel` option is employed to accelerate the entire process.

Listing 12.4: Building the project using a CMake preset

```
silhavyj@my-pc:ZeroMate$ cmake --build --parallel --preset
ninja_clang_release
```

Upon a successful build of the project, the user can find all libraries along with the main executable in the `output` folder located in the root folder of the project structure, whose contents can be seen in Figure 12.1. This `output` folder is what can be found in individual GitHub releases if the user opts not to go through the process of compiling the emulator themselves.

12.2 Project Structure

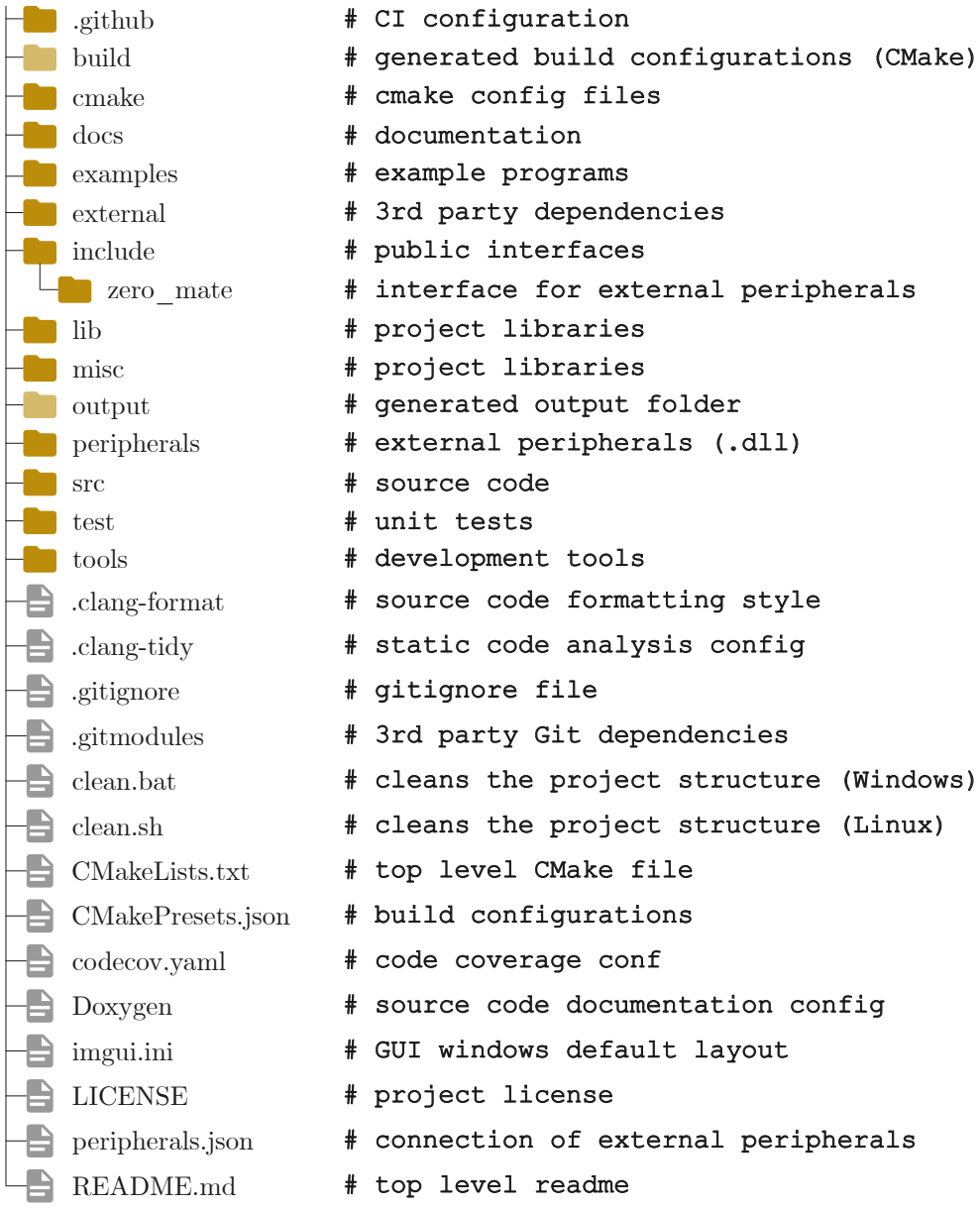


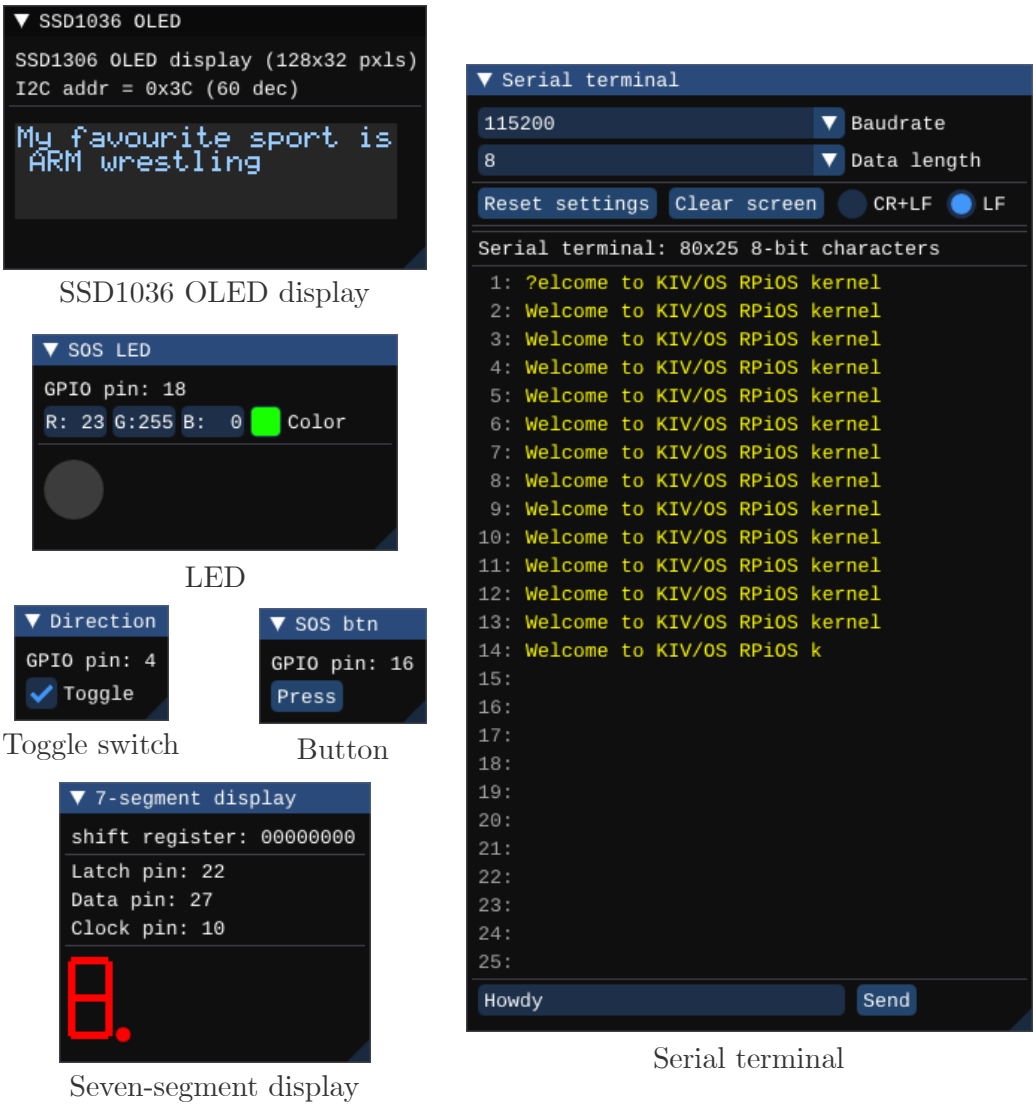
Figure 12.1: ZeroMate project structure

12.3 Execution Speed Raw Data

Table 12.2: Execution speed raw data

Instruction type	Avg. execution time [ns]	Count
Program status register transfer	170	30976
Single data transfer	260	178049029
Branch and exchange	59	5184115
Multiply	110	13659
Branch	100	42436124
Data processing	180	175296849
Block data transfer	1200	1859703
Signed multiply	100	6816
Co-processor register transfer	400	3176
NOP	32	471397
Store return state	780	141139
Halfword data transfer	260	83910
Extend	120	23003265
Change processor state	250	141146
Return from exception	490	141138
Software interrupt	10000	59524
Multiply long	260	82
Count leading zeros	120	16

12.4 Screenshots



SSD1036 OLED display

LED

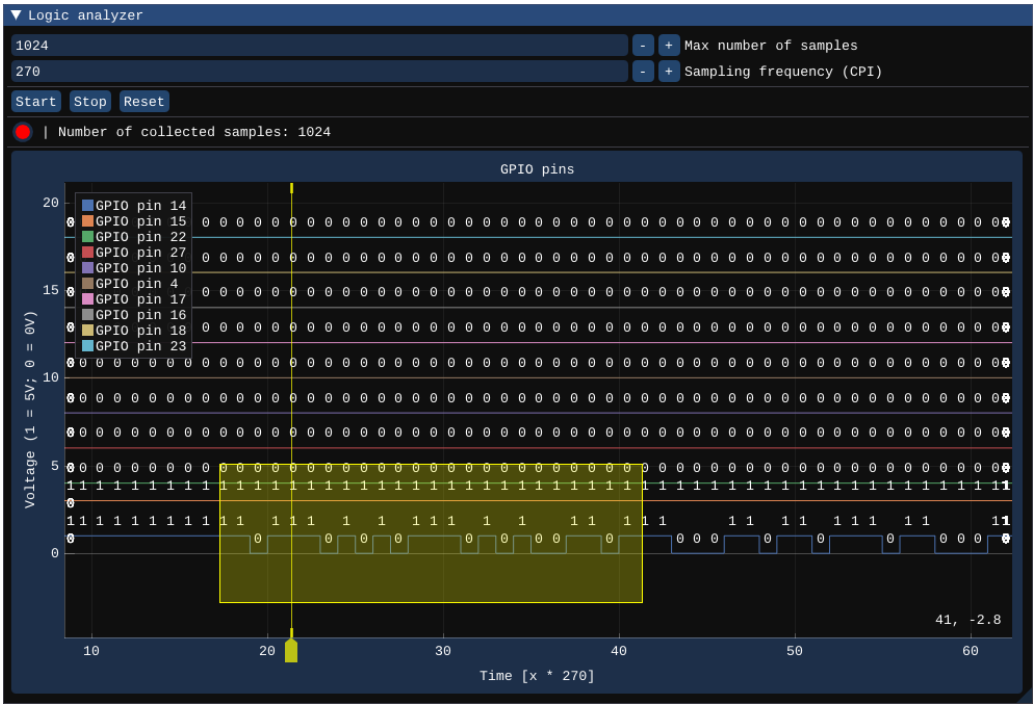
Toggle switch

Button

Seven-segment display

Serial terminal

Figure 12.2: External peripherals (1)



Logic analyzer

Figure 12.3: External peripherals (2)

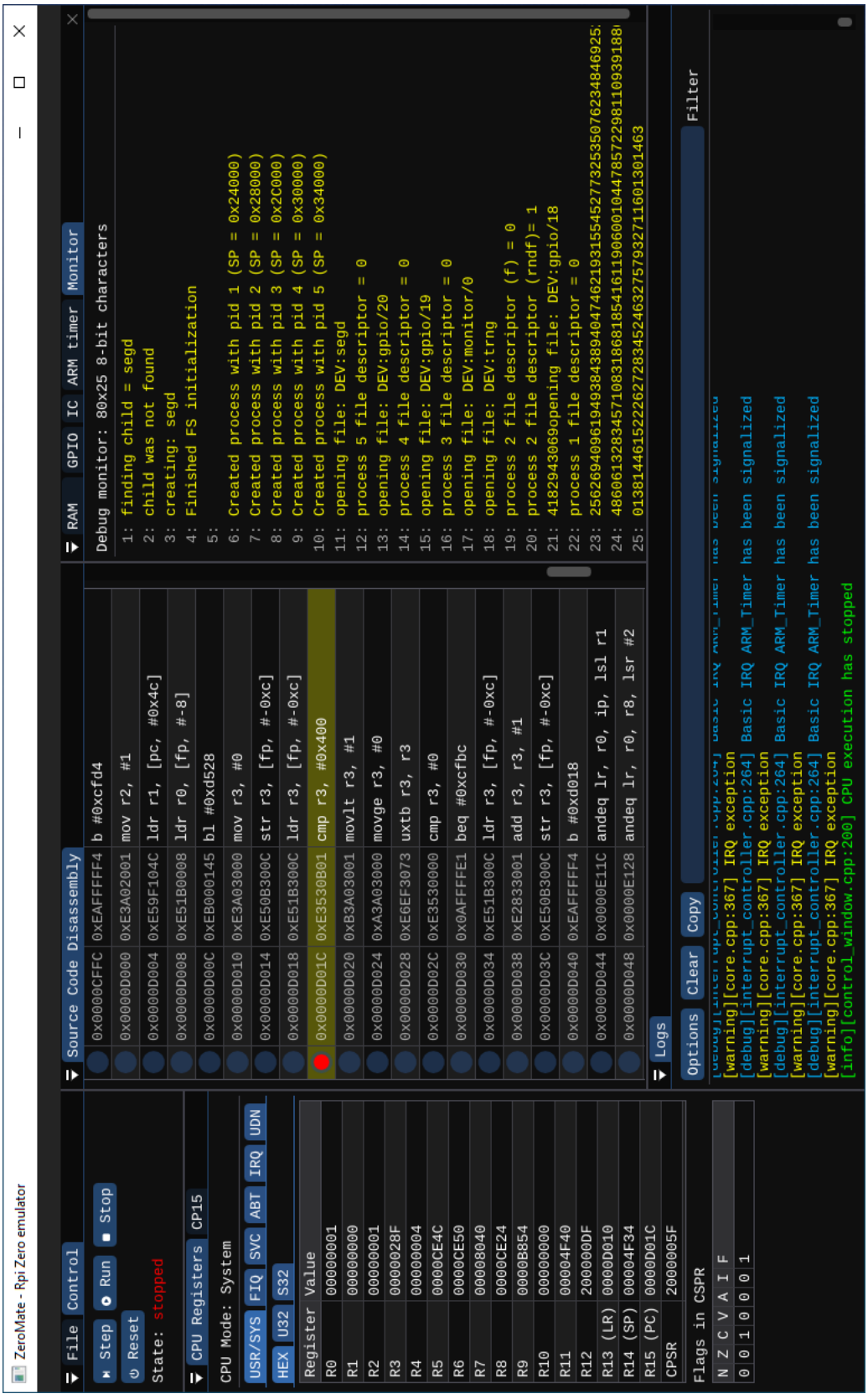


Figure 12.4: ZeroMate emulator