FAKULTA APLIKOVANÝCH VĚD
ZÁPADOČESKÉ UNIVERZITY
V PLZNI

KATEDRA INFORMATIKY
A VÝPOČETNÍ TECHNIKY

Bakalářská práce

Modulární emulátor platformy RISC-V pro výukové účely

Jonáš Dufek

PLZEŇ

2024

FAKULTA APLIKOVANÝCH VĚD
ZÁPADOČESKÉ UNIVERZITY
V PLZNI

KATEDRA INFORMATIKY
A VÝPOČETNÍ TECHNIKY

Bakalářská práce

Modulární emulátor platformy RISC-V pro výukové účely

Jonáš Dufek

Vedoucí práce
Ing. Martin Úbl

PLZEŇ **2024**

© Jonáš Dufek, 2024.

Všechna práva vyhrazena. Žádná část tohoto dokumentu nesmí být reprodukována ani rozšiřována jakoukoli formou, elektronicky či mechanicky, fotokopírováním, nahráváním nebo jiným způsobem, nebo uložena v systému pro ukládání a vyhledávání informací bez písemného souhlasu držitelů autorských práv.

Citace v seznamu literatury:

DUFEK, Jonáš. *Modulární emulátor platformy RISC-V pro výukové účely*. Plzeň, 2024. Bakalářská práce. Západočeská univerzita v Plzni, Fakulta aplikovaných věd, Katedra informatiky a výpočetní techniky. Vedoucí práce Ing. Martin Úbl.

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

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Jonáš DUFEK**
Osobní číslo:	**A21B0111P**
Studijní program:	**B0613A140015 Informatika a výpočetní technika**
Specializace:	**Informatika**
Téma práce:	**Modulární emulátor platformy RISC-V pro výukové účely**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování

1. Seznamte se s architekturou RISC-V a nejrozšířenějšími implementacemi její instrukční sady.
2. Analyzujte dostupná řešení pro emulaci a ladění programů pro tuto architekturu.
3. Navrhněte nástroj, který umožní emulovat platformu RISC-V s možností definovat a konfigurovat periferie odděleně od jeho jádra a provádět úkony související s laděním.
4. Implementujte tento nástroj jako modulární, umožněte vytvářet periferie definované v dynamicky linkovaných knihovnách.
5. Otestujte řešení na sadě standardních úloh a zhodnoťte dosažené výsledky.

Rozsah bakalářské práce: **doporuč. 30 s. původního textu**
Rozsah grafických prací: **dle potřeby**
Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

Dodá vedoucí bakalářské práce.

Vedoucí bakalářské práce: **Ing. Martin Úbl**
Katedra informatiky a výpočetní techniky

Datum zadání bakalářské práce: **2. října 2023**
Termín odevzdání bakalářské práce: **2. května 2024**

L.S.

Doc. Ing. Miloš Železný, Ph.D.
děkan

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

V Plzni dne 25. října 2023

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného akademického titulu.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Západočeská univerzita v Plzni má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Plzni dne 1. května 2024

. . . .

Jonáš Dufek

V textu jsou použity názvy produktů, technologií, služeb, aplikací, společností apod., které mohou být ochrannými známkami nebo registrovanými ochrannými známkami příslušných vlastníků.

Abstrakt

Tato práce se zabývá tvorbou modulárního emulátoru pro platformu RISC-V, který je zároveň možné použít pro výukové účely.

V první části práce je provedena analýza architektury RISC-V z komerčního a technického hlediska, dále jsou zde na teoretické úrovni popsány různé techniky emulace. Následně je věnován prostor analýze současných knihoven pro emulaci instrukční sady RISC-V a knihoven pro tvorbu uživatelského rozhraní.

Druhá část práce se týká především analýzy staré verze tohoto emulátoru, a popisem vytvořeného software.

Práce je zakončena sadou testovacích programů, které jsou otestovány zároveň na emulátoru a reálném hardwarovém zařízení. Důraz je zde kladen především na zhodnocení věrohodnosti a spolehlivosti emulace.

Abstract

This thesis deals with the development of a modular emulator for the RISC-V platform, which can also be used for educational purposes.

In the first part of the thesis, the RISC-V architecture is analyzed from a commercial and technical point of view, and various emulation techniques are described at a theoretical level. Subsequently, space is devoted to an analysis of current libraries for emulation of the RISC-V instruction set and libraries for the creation of graphical user interfaces.

The second part of the thesis is mainly concerned with the analysis of the old version of this emulator, and with the description of the developed software.

The thesis concludes with a set of test programs that are tested simultaneously on the emulator and on a real hardware device. The emphasis here is mainly on evaluating the plausibility and reliability of the emulation.

Klíčová slova

softwarová emulace • emulátor • RISC-V • počítačová architektura • Qt Quick • multiplatformní uživatelské rozhraní

Poděkování

Na tomto místě bych rád poděkoval vedoucímu práce, Ing. Martinu Úblovi, za cenné rady, věcné připomínky a vstřícnost při konzultacích a vypracování bakalářské práce.

Jonáš Dufek,
autor práce
(květen 2024)

Obsah

1 **Úvod** 3

2 **Platforma RISC-V** 5
- 2.1 Základní přehled a význam . . . 5
- 2.2 Technická struktura . . . 6

3 **Emulace** 9
- 3.1 Úvod do tématu . . . 9
- 3.2 Hardwarová emulace . . . 10
- 3.3 Softwarová emulace . . . 12
  - 3.3.1 Procesorová jednotka . . . 12
  - 3.3.2 Paměťový systém . . . 14
  - 3.3.3 Vstupně-výstupní zařízení . . . 16
  - 3.3.4 Emulovaný program . . . 16
  - 3.3.5 Nástroje pro ladění . . . 17
- 3.4 Knihovny pro emulaci instrukční sady . . . 18
  - 3.4.1 mini-rv32ima . . . 18
  - 3.4.2 rv32emu . . . 19
  - 3.4.3 libriscv . . . 20
  - 3.4.4 Souhrn . . . 21

4 **Uživatelské rozhraní** 23
- 4.1 GTK . . . 23
- 4.2 Nuklear . . . 24
- 4.3 webview . . . 26
- 4.4 Qt Widgets . . . 27
- 4.5 Qt Quick . . . 28
- 4.6 Souhrn . . . 29

5 **Výchozí implementace a použité technologie** 31
- 5.1 Zhodnocení výchozí implementace . . . 31

5.1.1 Dekompozice 31
5.1.2 Systém pro sestavení programu 32
5.1.3 Architektura 33
5.1.4 Periferní zařízení 35
5.1.5 Uživatelské rozhraní 35
5.2 Zvolené technologie 37

6 Realizovaná implementace **39**
6.1 Adresářová struktura a CMake 40
6.1.1 Specifika spojená s CMake 42
6.2 Architektura 42
6.3 Emulační jednotka 44
6.4 Periférie a dynamické knihovny 46
6.5 Uživatelské rozhraní 48
6.5.1 Konfigurační soubory 48
6.5.2 QML soubory 49
6.5.3 Modely a dynamické komponenty 50
6.6 Ostatní 50
6.6.1 Logger 51
6.6.2 Events 51

7 Testování a výsledky **53**
7.1 Jednotkové testy 53
7.2 Funkční testy 55
7.2.1 Použité zařízení 55
7.2.2 Testovací programy 55
7.2.3 Kompilace 56
7.2.4 Spuštění 57
7.3 Dosažené výsledky 60

8 Závěr **61**

A Uživatelská příručka **63**

Bibliografie **67**

Seznam obrázků **71**

Seznam tabulek **73**

Seznam výpisů **75**

Úvod 1

V současné době existuje z pohledu programátora velké množství podpůrných nástrojů, které lze při vývoji software aplikovat. Typický scénář je použití ***integrovaného vývojového prostředí***, které uživateli poskytne editor zdrojového kódu, detekci syntaktických chyb, nástroje pro ladění, kompilaci, testování software apod.

Pro softwarového vývojáře je toto dostačující řešení v případě, že hardwarová platforma použita pro vývoj se shoduje s platformou, na které bude výsledný software nasazen.

Problém nastává ve chvíli, kdy vývoj na zařízení samotném není možný. Často se jedná například o ***vložená zařízení*** nebo zařízení typu ***IoT***, které vyžadují program nejprve vytvořit na běžném počítači, a pak až ho nahrát do jejich interní paměti. Tento proces může být časově náročný, nebo i nemožný, v případě že programátor dané fyzické zařízení nevlastní.

Vhodným řešením je použití ***emulačního software***, s pomocí kterého je programátor schopný na vlastním počítači emulovat procesorovou architekturu a periférie cílového zařízení.

Cílem této práce je vytvořit modulární emulátor vývojové desky založené na architektuře **RISC-V**, který by se zároveň mohl prakticky uplatnit při výuce předmětu KIV/OS.

Jedním z důvodů vzniku této práce byly problémy stávajících open source řešení, u kterých často nebyla podpora periferních zařízení na požadované úrovni, a případnému doprogramování této funkcionality bránila jejich komplexnost.

Protože se jedná o software, který bude přímo používán koncovým uživatelem, bude se tato práce zaměřovat z velké části na tvorbu přehledného uživatelského rozhraní, které by mělo podporovat většinu dnes používaných operačních systémů.

Dalším cílem této práce je modularita, která by měla zajistit jednoduchost případných budoucích rozšíření a úprav.

Při realizaci lze částečně vycházet z již hotové semestrální práce zpracované v rámci předmětu KIV/ZSWI (akademický rok 22/23), která slouží jako základ tohoto tématu.

Teoretická část této práce se bude zabývat analýzou platformy **RISC-V**, poté bude zpracován přehled dostupných knihoven, prostředků pro tvorbu grafického rozhraní a samotného emulátoru.

Dále bude provedena analýza výchozí implementace programu, v rámci níž budou identifikovány a popsány nedostatky, které je třeba odstranit.

V realizační části bude představena finální implementace, včetně odůvodnění toho, jaké technologie, nástroje nebo knihovny byly pro implementaci zvoleny.

Na konec bude připraveno několik standardních úloh, na kterých bude hotový software otestován. Celkově zde bude zhodnocena funkčnost a reálná použitelnost výsledného software.

Platforma RISC-V 2

2.1 Základní přehled a význam

Instrukční sada RISC-V byla publikována v roce 2014 na Kalifornské univerzitě v Berkeley [1], jedná se již o pátou iteraci instrukční sady vyvíjené touto univerzitou.

Mezi hlavní cíle RISC-V patří plná otevřenost instrukčního souboru, to znamená, že vývoj procesorů pro RISC-V není zatížen žádnými licenčními poplatky. To je značná výhoda oproti jiným dnes používaným instrukčním souborům, jako například **ARM** nebo **MIPS**. [2, s. 1]

Samotné RISC-V procesory open source být nemusí, tedy firma vyvíjející procesor s instrukčním souborem RISC-V není povinna zveřejnit jeho interní architekturu. Přesto jsou ale k dispozici open source implementace RISC-V jader, viz například **OpenHW Group CORE-V**. [3]

O vývoj RISC-V se v současné době stará nezisková organizace RISC-V International, jejímiž členy jsou například Google, Intel, Nvidia, Qualcomm, Huawei, a další. [4]

O rozšířenosti RISC-V svědčí například data výše zmíněné firmy Qualcomm, která uvádí, že od roku 2019 bylo pro jejich účely vyrobeno něco přes 650 milionů RISC-V jader. Jako hlavní benefity RISC-V vidí Qualcomm jeho flexibilitu a otevřenost, která umožňuje návrhářům hardware přidávat vlastní instrukce a optimalizační techniky, pomocí kterých lze procesory specializovat pro potřeby umělé inteligence, strojového učení, 5G síťových systémů, IoT zařízení, apod. [5]

Dalším úspěchem pro RISC-V je jeho využití v Číně, kde RISC-V slouží jako příležitost zbavení se závislosti na licenčních podmínkách původem západních architektur, jako jsou **x86**, **ARM** nebo **PowerPC**. Tento krok zároveň umožňuje čínským institucím obejít americké sankce v oblasti polovodičů, které byly na Čínu uvaleny na konci roku 2023. [6]

Největším uživatelem RISC-V v Číně je společnost Alibaba, která se v letech 2021 - 2023 společně s čínskou univerzitou podílela na vývoji serveru disponující 3072 RISC-V jader. Tento systém představuje první komerční implementaci „RISC-V výpočetního clusteru“ v cloudovém prostředí. [7]

Dalším poskytovatelem RISC-V cloudových služeb je od února roku 2024 francouzská firma Scaleway. V době psaní tohoto textu je cena za pronájem jednoho Scaleway serveru 15,99€ měsíčně, v ceně je zahrnuta 100 Mbit/s síťová karta, IPv4, IPv6 veřejná adresa a 128 GB úložiště. Samotný server pak obsahuje 64 bitový procesor o 4 jádrech, společně s 16 GB DDR4 RAM. Ačkoliv je tento server relativně cenově dostupný, je mu namítáno použití eMMc úložiště, které je pomalejší a méně spolehlivé než alternativní řešení. [8]

Kromě běžných procesorů, si instrukční sada RISC-V našla uplatnění i v jiných oblastech ať už hardware, nebo software. Jedním příkladem jsou akcelerátory umělé inteligence a strojového učení od firmy Tenstorrent, které rozšiřují základní instrukční sadu o 256 bitové vektorové instrukce. Dle dostupných informací by tyto akcelerátory měly být vyrobeny na novém 2 nm výrobním procesu, jako plánovaný začátek sériové výroby je uveden rok 2027. [9]

V oblasti software existuje několik projektů využívajících instrukční sadu RISC-V jako formu ***bajtkódu***, tzn. přenositelného kódu, který lze za pomoci virtuálního stroje spustit na jakékoliv podporované platformě. Princip fungování je stejný jako u virtuálního stroje programovacího jazyka **Java**, nebo **WebAssembly**. Projekty **RVScript** a **PolkaVM** uvádějí nízkou latenci a nízké paměťové nároky jako hlavní výhody oproti existujícím řešením. [10][11][12]

2.2 Technická struktura

Jako většina procesorů typu RISC, je RISC-V tzv. ***load-store*** architekturou [13], to znamená, že přistupovat do hlavní paměti lze pouze pomocí instrukcí ***load*** a ***store***. Tento přístup zjednodušuje návrh procesorů na hardwarové úrovni (např. pipelining[1]).

Pro paměťový systém byla zvolena ***little-endian*** endianita, návrháři architektury se pro tento krok rozhodli, protože většina dnešních systémů (**x86**, **ARM**) využívá právě ***little-endian*** endianitu. [14, s. 21]

Základní podoba RISC-V má 32 registrů, z toho je registr `x0` konstantně nastaven na nulu a programátor ho nemůže nijak modifikovat. Tento nulový registr usnadňuje například nastavení registrů na nulu, nebo porovnávání nějakého registru s nulou. Registr `PC` taktéž nelze modifikovat přímo, tímto rozhodnutím byla zlepšena predikce skoků a zvýšena účinnost pipeliningu. [14, s. 18]

RISC-V je vyvíjen jako modulární instrukční soubor, každý RISC-V procesor obsahuje instrukční sadu **RV32I** nebo **RV64I**, které poskytují základní celočíselné operace. Podle potřeby jsou pak přidávány další instrukční sady jako například **RV32A**

[1]Pipelining je technika umožňující procesoru paralelně zpracovávat více instrukcí na jednou. Každá instrukce je rozdělena na několik sekvenčních kroků, jednotlivé „stupně“ pipeline pak tyto kroky provádějí.

(atomické instrukce), **RV32M** (instrukce dělení a násobení), **RV32F** (instrukce pro operace s čísly s plovoucí desetinnou tečkou), apod.

Existují i další instrukční sady jako třeba **RV128I**, pro procesory s 128 bitovým adresováním, **RV32V** pro vektorové instrukce, **RV32E** pro účely embedded systémů, **RV32B** pro bitové manipulace, apod. Všechny tyto instrukční sady jsou ale stále v režimu aktivního vývoje, RISC-V Foundation tudíž negarantuje jejich neměnnost a připouští budoucí změny.

Oproti tomu všechny instrukční sady uváděné v tabulce 2.1 jsou označovány jako „frozen", což značí že jejich vývoj byl ukončen a již nikdy by nemělo dojít k jejich změně. To je užitečné zejména pro programátory kompilátorů nebo operačních systémů, kteří se mohou zaměřit právě na tyto instrukční sady, čímž je zaručena budoucí kompatibilita s novým hardware. [14, s. 5]

Sada **RV32I** tvoří jádro celé architektury, kromě celočíselných operací jako `add` (sčítání) nebo `sub` (odečítání), obsahuje i instrukce pro řízení toku programu `jal` (volání podprogramu), `beq` (skok v případě rovnosti), `bne` (skok v případě nerovnosti), nebo instrukce pro přístup do paměti `lw` (načtení slova), `sw` (uložení slova).

Hlavní instrukce přidané sadou **RV32M** jsou `mul` (násobení), `div` (dělení) a `rem` (zbytek), zbylých 5 instrukcí v této sadě jsou varianty těchto instrukcí pro ***znaménková*** (signed) a ***bez-znaménková*** (unsigned) čísla.

RV32A poskytuje atomické instrukce pro přístup do paměti, lze pomocí nich zaručit konzistenci paměti na úrovni hardware. Jsou tedy užitečným nástrojem pro vícevláknové programování.

Sady **RV32F** a **RV32D** zavádějí operace pro práci s čísly s plovoucí desetinnou tečkou podle standardu IEEE 754. Rozdíl mezi těmito dvěma sadami je pouze v šířce jejich argumentu, jinak jsou jejich instrukce totožné. Pro práci s těmito čísly je navíc zavedeno dalších 32 samostatných registrů (`f0` až `f31`).

RV32C poskytuje 16 bitové verze nejpoužívanějších instrukcí z ostatních instrukčních sad. Oproti standardním 32 bitovým instrukcím tak přináší možnost ušetření paměti.

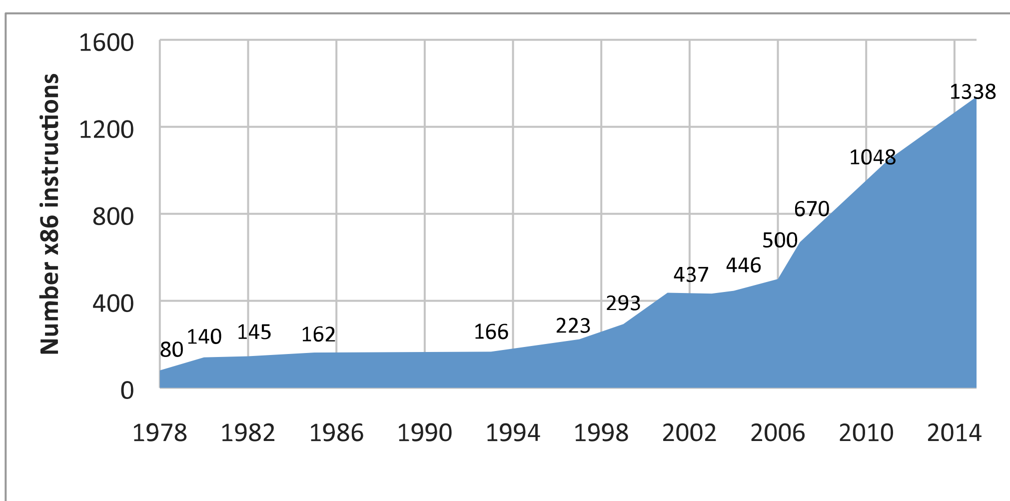
Jako poslední je v tabulce 2.1 zmíněno **RV32G**. **RV32G** není samostatnou instrukční sadou, jedná se pouze o souhrnné označení pro sady `I, M, A, F, D`. Přesto je ale důležitým pojmem, protože se často objevuje ať už v dokumentačních materiálech, nebo v názvech různých RISC-V procesorů.

Výhodou této modularizace (rozdělení na množství menších vzájemně nezávislých instrukčních sad), je vyhnutí se „inkrementálnímu" přístupu, který byl problémem starších architektur, jako například **x86**.

Typická inkrementální architektura typu **x86** vyžadovala plnou zpětnou kompatibilitu, tedy všechny nové procesory musely kompletně implementovat všechny instrukce procesorů starších. To způsobilo, že instrukční soubor x86 se časem rozrostl z původních 80 instrukcí v roce 1978 až na 1338 instrukcí v roce 2014. [14, s. 3]

Označení	Popis	Počet instrukcí
I	Základní celočíselné operace	49
M	Násobení a dělení	8
A	Atomické instrukce	11
F	Single-Precision Floating-Point	25
D	Double-Precision Floating-Point	25
C	„Komprimované“ instrukce	36
G	„General“, zahrnuje I, M, A, F, D	—

Tabulka 2.1: Základní rozšiřující instrukční sady RISC-V, typicky uváděné ve formátu RV[32/64][písmena rozšíření]



Obrázek 2.1: Ilustrace inkrementálního přístupu x86, zdroj: [14, s. 3]

Řada těchto instrukcí je dnes již zastaralých, například **x86** instrukce pro BCD kód, přesto ale musí být implementovány každým **x86** procesorem. To způsobuje ztrátu efektivity a snižuje přehlednost samotného instrukčního souboru.

Méně instrukcí mimo jiné také znamená menší plochu čipu, tím lze při výrobě ušetřit místo na křemíkovém waferu, a dosáhnout tak nižší ceny. [15]

Emulace 3

3.1 Úvod do tématu

Emulátor je program (nebo zařízení) které má za cíl simulovat určitou počítačovou platformu nebo architekturu.

První emulátory byly vyvinuty s rozvojem výpočetní techniky na konci 60. let minulého století. V korporátní sféře emulátory jako první začala používat firma IBM. [16] Zde emulátory sloužily jako nástroj zpětné kompatibility, který umožnil novějším počítačům spouštět programy původně napsané pro počítače starších architektur. Díky tomuto kroku byly ušetřeny nemalé finanční prostředky, které by jinak musely být vynaloženy na modifikaci nekompatibilních programů.

K vývoji emulátorů v 60. a 70. letech přispěla také oblast raketového inženýrství. Například pro řídící počítač D-17B použit v americké mezikontinentální balistické raketě Minuteman I byl v roce 1972 vytvořen softwarový emulátor v jazyce Fortran. [17] Toto je rozdíl oproti již zmiňovaným emulátorům IBM, které byly implementovány hardwarově na úrovni mikrokódu.

V dnešní době jsou emulátory často využívány například pro emulaci herních konzolí, nebo mobilních zařízení, hlavní využití ale stále zůstává v oblasti vývoje software.

Z technického hlediska lze interní strukturu každého emulátoru rozdělit do tří základních částí:

- Procesorová jednotka (datové cesty, ALU, registry, ...)
- Paměť
- Vstupně-výstupní zařízení

Jednotlivé techniky emulace se od sebe odlišují v rychlosti, přesnosti, dostupnosti nástrojů pro ladění, a podobně. V následujících podkapitolách budou představeny klady a zápory dvou dnes nejpoužívanějších emulačních technik, tzn. hardwarové a softwarové emulace.

Tento text se dále bude zaměřovat pouze na emulaci procesorů typu **RISC** (s redukovanou instrukční sadou). Emulace procesorů tohoto typu je podstatně jednodušší než u procesorů **CISC** (s komplexní instrukční sadou). O tom svědčí i fakt, že s věrohodnou emulací dnes nejpoužívanější **CISC** architektury **X86** měli do nedávné doby problémy například i velké softwarové projekty typu QEMU, Valgrind, Simics, atd. [18]

3.2 Hardwarová emulace

Hardwarová emulace má za cíl buď kompletně replikovat výchozí systém na úrovni digitální logiky, nebo se jeho funkčnosti co nejvíce přiblížit.

Pro realizaci této metody emulace je nejdříve třeba navrhnout samotný digitální obvod v některém z jazyků pro popis hardware (např. VHDL, Verilog), nebo použít některou z již existujících open source implementací procesorů RISC-V.

Tento obvod je pak možné nahrát na FPGA (programovatelné hradlové pole). [19] FPGA lze běžně zakoupit jako součást vývojové desky (viz obr. 3.1), která kromě samotného integrovaného obvodu zahrnuje i paměť, konektory, tlačítka, displej, apod.

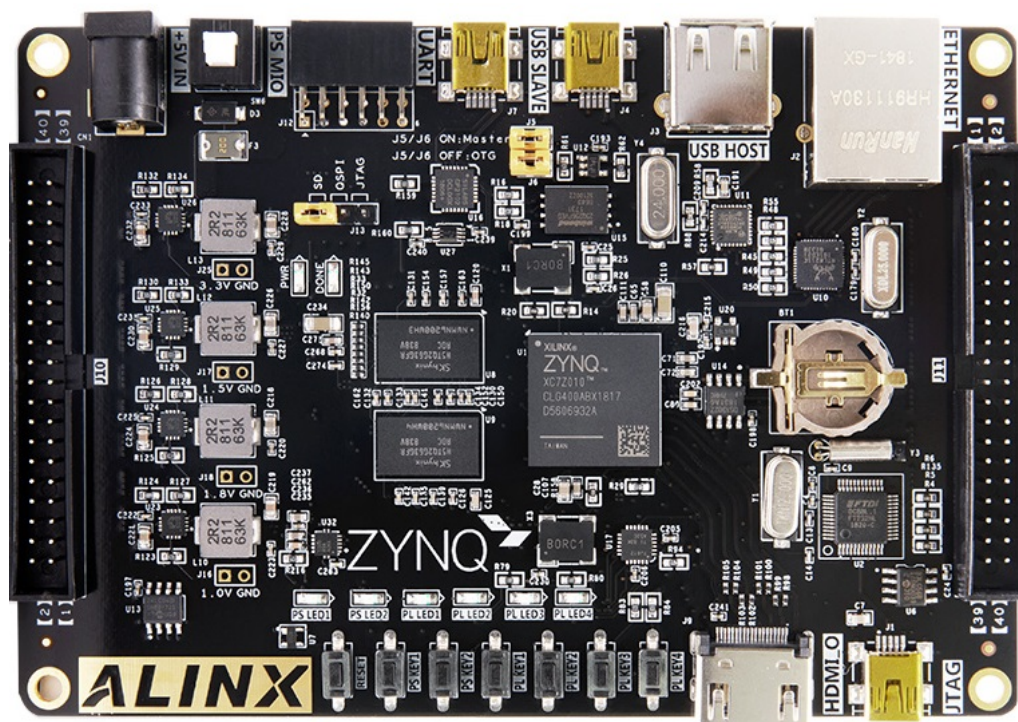
FPGA je možné kdykoliv přeprogramovat a využít pro jiný obvod, můžeme tak aplikovat iterativní vývojový cyklus podobně jako u software. [20]

Použití FPGA je obzvlášť výhodné, pokud se snažíme emulovat historický systém, který již fyzicky neexistuje nebo není k dispozici. V tomto případě lze k FPGA připojit i hardwarová periferní zařízení původního systému, bez nutnosti tyto zařízení upravovat nebo je simulovat softwarově.

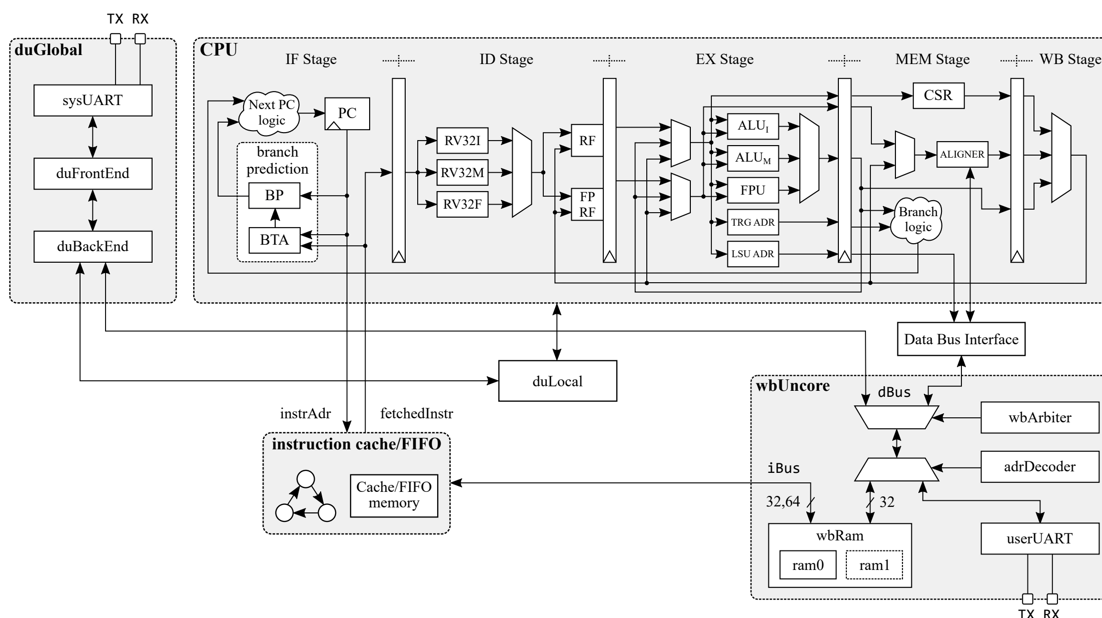
Mezi nevýhody FPGA patří obtížnější a časově náročnější ladění. Můžeme využít logický analyzátor nebo ladící prostředky poskytované přímo vývojovou deskou (např. rozhraní JTAG). Další možností je integrování ladících prostředků do vyvíjeného digitálního obvodu, příklad tohoto přístupu můžeme vidět na obr. 3.2.

Nevýhodou je také vyšší pořizovací cena většiny FPGA vývojových desek. To znamená, že použití FPGA pro jednorázovou emulaci levných mikrokontrolérů a dalších embedded zařízení nedává smysl.

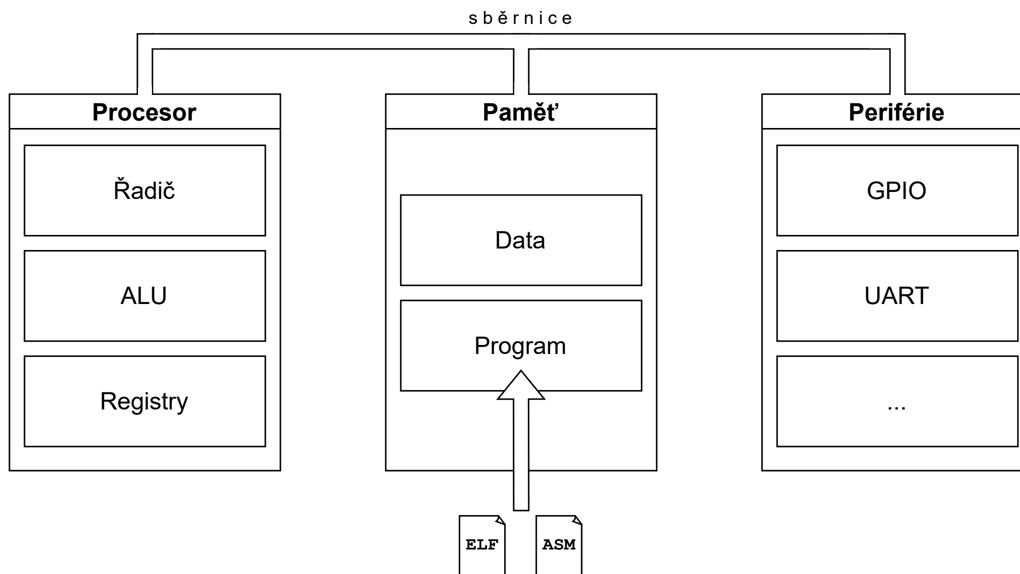
Digitální obvod je možné také simulovat softwarově pomocí logického simulátoru. Z kategorie open source software je to například Logisim. [21] Logický simulátor tak lze použít pro prototypování obvodu, co bude následně nahrán na FPGA nebo fyzicky vyroben. Samozřejmostí ale je, že softwarová simulace bude mnohonásobně pomalejší než reálný obvod nebo FPGA.



Obrázek 3.1: Ukázka FPGA vývojové desky (ALINX AX7020), zdroj: [22]



Obrázek 3.2: Ilustrace RISC-V procesoru typu RV32IMF natrhnutého pro účely FPGA. V procesorové části (CPU) si lze povšimnout pěti-úrovňového pipeliningu, typického pro procesory typu RISC. Pro účely ladění je zde zahrnut blok duGlobal, se kterým lze komunikovat pomocí integrovaného UART rozhraní. Zdroj: [23]



Obrázek 3.3: Diagram ukazující vysokoúrovňovou strukturu softwarového emulátoru. Nutným vstupem emulátoru je program, který je před začátkem emulace třeba načíst do paměti (o tento krok se na reálném hardware stará bootloader).

3.3 Softwarová emulace

V této podkapitole bude představen souhrn technik, které lze uplatnit při programování softwarového emulátoru. Vzájemný vztah komponent probíraných v následujícím textu je pak popsán na obrázku 3.3.

3.3.1 Procesorová jednotka

Nejdůležitější částí emulátoru je procesorová jednotka, jejímž vstupem je strojová instrukce (v binární podobě), úkolem procesoru je instrukci dekódovat, a na základě toho pak provést požadovanou činnost. Při každém cyklu procesor modifikuje svůj interní stav (registry), a případně může také zapsat nebo přečíst hodnotu z paměti.

Datová struktura vyjadřující interní stav procesoru by měla obsahovat obecné registry, které je nejvhodnější vyjádřit jako pole. Klíčový je pak registr PC (Program Counter), který udává adresu následující instrukce. PC musí být nutně modifikován při každém procesorovém cyklu, buď přičtením délky aktuální instrukce, nebo nastavením na jinou hodnotu (v případě skoku). Jako další může tato struktura obsahovat například flag registry, a další specifika dané architektury.

Samotnou činnost procesoru (zpracovávání instrukcí), lze implementovat následujícími třemi způsoby [24]:

- Interpretace

- Statická rekompilace
- Dynamická rekompilace (kombinace předchozích dvou technik)

3.3.1.1 Interpretace

Nejjednodušší z těchto tří metod je interpretace. Ve smyčce sekvenčně načítáme instrukce z paměti a provádíme je. Implementace tak víceméně kopíruje jednotlivé úrovně pipleline reálného procesoru (viz obr. 3.2), při softwarové implementaci se můžeme vyhnout některým detailům hardwarové implementace, jako je predikce skoků, cache, apod.

Musíme tedy implementovat alespoň funkce ilustrované na obrázku 3.4.

Funkce **instruction_fetch** načte instrukci z paměti a inkrementuje program counter.

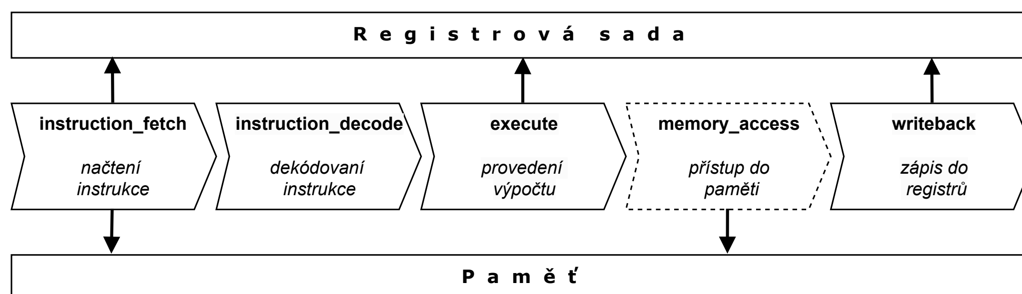
Načtená instrukce je předán funkci **instruction_decode**, která z ní dekóduje opkód (operační kód), který slouží jako identifikátor typu instrukce. Typ instrukce můžeme mít definovaný například jako `enum` (výčet).

Instrukce, společně s jejím typem, je předána funkci **execute**, která z instrukce na základě jejího typu získá všechny potřebné argumenty a instrukci poté provede. Tuto funkci lze implementovat pomocí příkazu `switch`, do kterého vložíme `case` pro každý typ instrukce.

Funkce **memory_accesss** Provede buď čtení nebo zápis do paměti, adresu přebírá od funkce **execute**. Funkce je přeskočena, pokud není potřeba přistupovat do paměti.

Jako poslední pak funkce **writeback** zapíše konečný výsledek[1] výpočtu do registrů.

[1] „konečný výsledek" zde znamená hodnotu vypočtenou ve funkci **execute**, pokud nebyla provedena funkce **memory_accesss**. V opačném případě je to hodnota načtená z paměti funkcí **memory_access**.



Obrázek 3.4: Diagram popisující návaznost funkcí, pomocí kterých je interpretována instrukce.

3.3.1.2 Statická rekompilace

Interpretace původní program nijak nemění, pouze přijme strojový kód a interpretuje ho. Oproti tomu cílem statické rekompilace je program převést do instrukční sady našeho systému, program tak lze spustit stejně jako jakýkoliv jiný, a je tím eliminována ztráta výkonu spojená s interpretací.

Nevýhodou ale je, že statickou rekompilaci nelze provést, pokud program za běhu modifikuje sám sebe.

3.3.1.3 Dynamická rekompilace

Je kombinací interpretace a statické rekompilace. Části programu jsou za běhu rekompilovány do instrukční sady našeho počítače, tím je teoreticky možné emulovat i programy co modifikují samy sebe.

Z teoretického hlediska je tato metoda nejvýhodnější, prakticky je ale těžké ji správně implementovat.

3.3.2 Paměťový systém

V této podkapitole budou představeny dva nejběžnější způsoby organizace paměťového systému v softwarovém emulátoru. Obecně platí, že emulace paměti je snadnější než implementace na reálném hardware, protože o specifika spojená s přístupem k fyzické hardwarové paměti se již stará hostitelský operační systém ve kterém emulátor běží.

3.3.2.1 Lineární model

Nejprimitivnější implementace paměti je lineární model, který je tvořen jedním polem. Délka pole a velikost jeho prvků nám pak udává celkovou kapacitu paměti. Nejpraktičtější je velikost jednoho prvku pole nastavit jako 1 bajt, v tom případě bude maximální velikost rovna 4,2 GB, pokud námi zvolený programovací jazyk reprezentuje index pole jako 32 bitovou hodnotu. Tuto limitaci jde obejít buď použitím 64 bitové hodnoty pro indexaci pole (pokud to umožňuje programovací jazyk), nebo použitím větších hodnot pro velikost jednoho prvku pole (například 32 bitů nebo 64 bitů).

Procesorové jednotce pak můžeme rovnou předat ukazatel na toto pole, kdy adresní prostor je definován indexací pole, tzn. první adresa je rovna nule a poslední je rovna velikosti pole.

Přímý přístup do paměti pomocí ukazatele je sice nejjednodušším řešením, pro jakýkoliv složitější emulátor ale není vhodný, protože:

- Není nijak ošetřena ochrana paměti (zápis na index vyšší než je kapacita pole)

- Nelze nijak zajistit bázové adresování (první adresa je vždy nula, stejně jako index pole)

- Nelze implementovat paměťově mapovaná periferní zařízení

Řešením je zavedení dvou pomocných funkcí **read_memory** a **write_memory**, které poskytují jednotné rozhraní pro přístup k paměti a plní tak stejnou roli jako sběrnice a řadič paměti v hardwarové implementaci.

Tyto funkce obalují přístup k poli s obsahem paměti. Lze tak implementovat ochranu paměti, bázové adresování a paměťově mapované periférie. Tímto krokem rozdělíme adresní prostor na virtuální adresy a „reálné" adresy. Funkce tak vždy buď převede virtuální adresu na adresu reálnou (index pole) a provede očekávanou operaci, nebo informuje uživatele o chybě, například pomocí výjimky.

3.3.2.2 **Stránkování**

Stránkování na rozdíl od lineárního modelu neukládá všechna data sekvenčně v jednom poli, ale rozděluje paměťový prostor na oblasti o předem dané velikosti tzv. ***stránky***.

Hardwarová implementace stránkování se skládá z tabulky stránek, paměťového řadiče, TLB cache, a fyzické paměti.

V softwarové implementaci toto můžeme zjednodušit, nýbrž nepřistupujeme přímo do fyzické paměti, tak můžeme využít například datové struktury slovník, kde klíčem je identifikátor stránky (např. její nejnižší adresa) a hodnotou je ukazatel na obsah stránky (implementovaný jako pole).

Přístup do paměti pak lze zajistit pomocí funkcí **read_memory** a **write_memory**, které pomocí virtuální adresy modifikují nebo přečtou obsah stránky ze slovníku.

Stránkování má hned několik výhod oproti lineárnímu modelu. Velikost paměti již není dána počtem prvků jednoho pole, ale jako počet stránek vynásoben jejich velikostí, problém indexace pole omezující maximální velikost paměti je tak vyřešen.

Použití stránek také umožňuje lepší správu a ochranu paměti. Některé stránky mohou být nastaveny jako ***read-only*** (jen pro čtení). Stránkovaný paměťový prostor nemusí být spojitý, můžeme tak mít spojitou skupinu stránek od adresy `0x000000` do `0x00020000`, a pak například pro účel paměťově mapovaných periférií přidat stránku na adresu `0x01960000`, bez toho aniž bychom museli rozsah adres `0x00020000` až `0x01960000` alokovat a vyplnit nulami, jako v případě lineárního modelu.

Při softwarové implementaci stránkování nemusíme řešit odkládání stránek na disk, v případě potřeby to za nás udělá hostitelský operační systém.

3.3.3 Vstupně-výstupní zařízení

Vstupně-výstupní zařízení (tzv. ***I/O zařízení***) jsou základním prostředkem umožňující interakci uživatele se systémem. Tato zařízení jsou k počítači připojena přes vybraný typ sběrnice (PCI, USB, SATA, I2C, ...), která slouží jako komunikační kanál.

Existují dva základní přístupy, jak komunikaci s I/O zařízením realizovat, první z nich je komunikace pomocí speciálních ***I/O instrukcí***, tyto instrukce jsou například `in` a `out` v instrukcím souboru **x86**.

Druhý způsob jsou paměťově mapované periférie (tzv. ***Memory-mapped I/O***), pro komunikaci s periferiemi se pak používá vyhrazená oblast v hlavní paměti. Tento způsob využívá například **RISC-V** a další instrukční sady typu RISC, z toho důvodu, že pro komunikaci není třeba zavádět nové instrukce.

Samotné I/O zařízení si může uchovávat svůj interní stav, při softwarové emulaci je tedy vhodné ho reprezentovat jako objekt nebo strukturu. Zařízení může být implementováno jako „speciální případ", který přes ukazatele přímo přistupuje k jednotlivým komponentám emulátoru. Lepším řešením je ale zařízení oddělit od interní struktury emulátoru pomocí vhodného rozhraní, ve formě například abstraktní třídy, ze které třídy jednotlivých zařízení dědí.

Po definici jednotného rozhraní je možné k emulátoru „připojit" jakékoliv zařízení, co toto rozhraní splňuje, stejně jako u reálné hardwarové implementace.

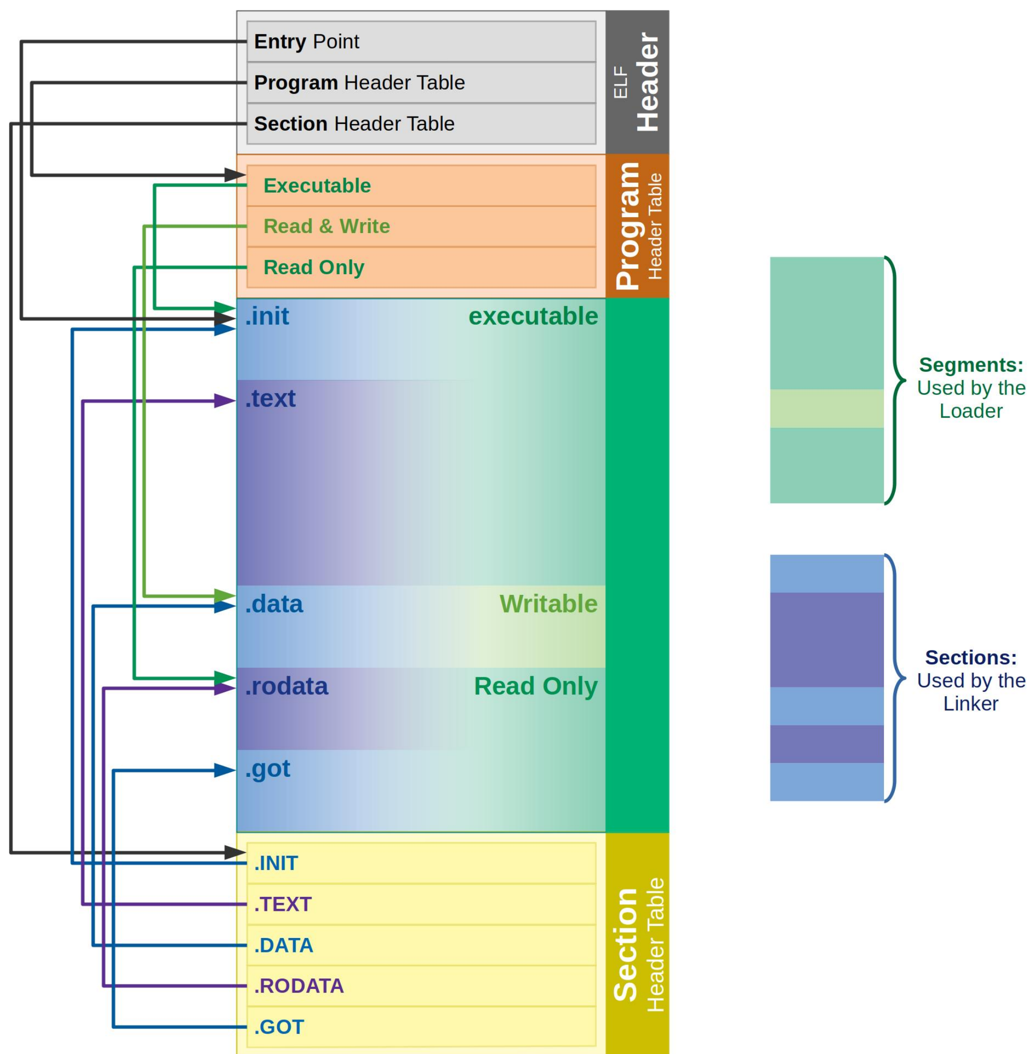
Ať už je zařízení paměťově mapované, nebo používá I/O instrukce, je nutné nějakým způsobem zajistit, aby emulátor byl informován o změně interního stavu zařízení. Při použití paměťově mapovaných periférií je třeba zachytit zápisy / čtení z jejich adresního prostoru ve funkcích **read_memory** a **write_memory** (viz sekce 3.3.2). Pokud používáme I/O instrukce, musíme zavést nějaký systém přerušení nebo událostí, pomocí kterého bude emulátor notifikován o změně stavu.

3.3.4 Emulovaný program

Pro praktické použití emulátoru je potřeba ho nejdříve nějakým způsobem inicializovat, tedy umístit program do jeho paměti a nastavit registry.

Program může být emulátoru poskytnut v podobě strojového kódu (nebo assembly kódu, který je interně přeložen na strojový kód). V tomto případě si program musí sám definovat všechny své datové a kódové sekce (`.init`, `.text`, `.data`, ...). Nutné je také poskytnout emulátoru ***entry point*** (vstupní bod), od kterého se program začne vykonávat, emulátor ho nastaví jako počáteční hodnotu registru PC.

Dalším způsobem je použití ***ELF souboru*** (Executable and Linkable Format). Tento soubor obsahuje informace o různých sekcích programu, které jsou pomocí ***bootloaderu*** (zavaděče) načteny do paměti. Emulátor tak musí provést syntaktickou analýzu ELF souboru, načíst jeho sekce na určená místa v paměti a jako poslední nastavit registr PC.



Obrázek 3.5: Struktura ELF souboru, zdroj: [25]

ELF soubor lze vytvořit například pomocí **GCC** kompilátoru pro danou platformu. Struktura tohoto souboru je znázorněna na obrázku 3.5.

3.3.5 Nástroje pro ladění

Pro implementaci tzv. ***debuggeru*** je zapotřebí modifikovat instrukční cyklus popsaný v sekci 3.3.1. Krokování programu po jednotlivých instrukcích zajistíme tak, že na začátku cyklu vždy čekáme, dokud uživatel manuálně nezmáčkne tlačítko „pokračovat“. ***Breakpointy*** lze implementovat podobně, v seznamu si uchováváme adresy všech breakpointů, když v instrukčním cyklu na některou z těchto adres narazíme, tak cyklus pozastavíme.

Výpis registrů a paměti je jednoduchý, jelikož ukazatele na jejich datové struk-

tury máme vždy k dispozici. Obsah paměti je vhodné vypisovat po částech, nebo použít grafický komponent s podporou postupného načítání.

Některé emulátory navíc poskytují i možnost připojení externích debuggerů (např. **GDB**). V takovém případě musí emulátor implementovat rozhraní dle dokumentace zvoleného debuggeru.

3.4 Knihovny pro emulaci instrukční sady

V této podkapitole prakticky uplatníme znalosti získané v předešlé části textu na zhodnocení a analýzu knihoven, které bychom mohly použít pro implementaci „procesorové“ části našeho emulátoru.

Všechny knihovny zmiňované níže poskytují permisivní *open-source* licence, které nám umožňují jejich použití v našem software bez jakýchkoliv právních rizik.

3.4.1 mini-rv32ima

- Dostupné z: github.com/cnlohr/mini-rv32ima

Jde o minimální implementaci instrukčních souborů I, M, A ve 32 bitové verzi. Celá knihovna je napsána v jazyce C a obsažena v jednom hlavičkovém souboru.

I přesto, že se jedná o opravdu minimalistickou knihovnu (zhruba 400 řádek kódu), dokáže spustit některé verze operačního systému GNU/Linux.

Knihovna neobsahuje žádné nástroje pro správu paměti a implementuje pouze jednoduchý lineární paměťový model, tedy vše je ve stejném adresním prostoru a nedochází zde k využití stránkování nebo segmentace.

Autor knihovny zmiňuje částečnou podporu paměťově mapovaných I/O zařízení, s tím, že většinu funkcionality si musí uživatel knihovny doprogramovat ručně.

Knihovna je ve vývoji zhruba od konce roku 2022, to ji dělá ze všech 3 knihoven v této kapitole nejnovější.

Výhody

- jednoduchost, pouze jeden soubor, nezávisí na žádných knihovnách třetích stran
- potenciál pro vlastní rozšíření

Nevýhody

- oproti ostatním knihovnám je vývoj méně aktivní
- knihovna obsahuje pouze instrukční sady I, M, A ve 32 bitové verzi, implementací dalších by se kód výrazně rozrostl

- pouze lineární paměťový model, implementace stránkování nebo segmentace by opět vyžadovalo výrazný zásah do kódu knihovny

- jedná se o takzvanou „header-only" knihovnu, musí se tedy znovu kompilovat pro každý soubor co ji používá, to by hlavně při rozšíření o další funkcionalitu způsobilo pomalou kompilaci

3.4.2 rv32emu

- Dostupné z: github.com/sysprog21/rv32emu

Je knihovna s podporou emulace instrukčních souborů I, M, A, C, F ve 32 bitové verzi. Knihovna je napsána kompletně v jazyce C99.

Oproti knihovně mini-rv32ima se již nejedná o „header-only" implementaci, knihovna je dekomponována do celkem 16 hlavičkových souborů. Zároveň je tato knihovna závislá na dvou externích knihovnách.

Knihovna pro správu paměti využívá stránkování, navíc obsahuje ELF loader, který umožňuje načíst zkompilovaný program ve formátu ELF do paměťového prostoru emulátoru.

Knihovna obsahuje základní podporu pro paměťově mapovaná I/O zařízení v podobě tzv. callback funkcí. Chybí zde ale jakýkoliv popis nebo návod na použití, uživatel knihovny si tedy musí sám podle zdrojových souborů knihovny zjistit, jak tyto I/O callback funkce použít.

Z hlediska nástrojů pro ladění knihovna poskytuje možnost připojit se a krokovat ELF soubor pomocí GDB protokolu. Pro aktivaci GDB je potřeba provést sestavení knihovny s přepínačem *ENABLE_GDBSTUB=1*, schopnost využít GDB debugger není součástí výchozího sestavení.

Mezi další užitečné nástroje této knihovny patří například možnost výpisu registrů do JSON souboru, nebo výpis histogramu instrukcí, které byly za doby běhu emulátoru provedeny.

Knihovna je vyvíjena od roku 2020, to ji po libriscv děla druhou nejstarší knihovnou v této kapitole. Vývoj knihovny se zdá být aktivní, za poslední rok vždy alespoň jeden git příspěvek každý měsíc.

Součástí repozitáře jsou i jednotkové testy a benchmarky porovnávající výkon na sadě testovacích programů.

Výhody

- množství ladících nástrojů

- aktivní vývoj

- stránkovaná paměť
- jednotkové testy a testovací programy

Nevýhody

- zdrojové soubory nejsou správně dekomponovány do složek, vše včetně hlavičkových souborů je ve složce ./src
- podpora pouze 32 bitové verze instrukčního souboru, oproti mini-rv32ima přidává pouze C a F instrukce
- Závislost na externích knihovnách

3.4.3 libriscv

- Dostupné z: github.com/fwsGonzo/libriscv

Knihovna podporuje emulaci instrukčních sad RV32GCB, RV64GCB, a RV128G. Autor uvádí také podporu instrukční sady V (vektorové instrukce), která ale v současné době v knihovně ještě není plně implementována.

Z toho vyplývá, že oproti ostatním v této kapitole zmiňovaným knihovnám podporuje libriscv téměř všechny dnes běžně používané RISC-V instrukční sady včetně jejich 32 bit, 64 bit a 128 bit implementací.

Knihovna je implementována v jazyce C++ a využívá moderních konstrukcí C++20. Z hlediska rozsáhlosti a množství zdrojových souborů se jedná o relativně velkou knihovnu. Všechny hlavičkové a implementační soubory se nacházejí ve složce /lib/libriscv/, soubory jsou zde dekomponovány do několika vnořených podsložek.

Výhodou libriscv je použití jazyka C++ a objektového paradigmatu, to oproti knihovnám v jazyce C usnadňuje práci uživateli knihovny, který si pouze vytvoří objekt třídy *Machine* a volá nad ním metody. Třída *Machine* umožňuje načíst program ve formátu ELF, spustit ho, ladit, apod.

Libriscv obsahuje podporu pro zachycení zápisu a čtení paměti pomocí callback funkcí. Ukázky použití této funkcionality jsou obsaženy v jednotkových testech knihovny a popsány autorem v repozitáři knihovny.

Jelikož knihovna pro správu paměti používá stránkování, je implementace paměťově mapovaných I/O zařízení složitější, nicméně to lze očekávat u každé knihovny, kromě těch co používají pouze lineární paměťový model.

Jak již bylo zmíněno, třída *Machine* obsahuje podporu debuggeru. Ladění může probíhat buď pomocí připojeného GDB debuggeru, nebo pomocí manuálního krokování prostřednictvím metod třídy, pomocí těchto metod si může uživatel knihovny vhodně naprogramovat vlastní ladící rozhraní.

Libriscv je vyvíjen již od roku 2019. Stále probíhá aktivní vývoj a nové verze vycházejí přibližně každé 3 měsíce.

Knihovna pokrývá velkou část svého kódu jednotkovými testy, zároveň jsou k dispozici testovací programy, a výsledky různých benchmarků.

Dokumentace knihovny je kvalitní, obsahuje ukázky kódu a návod k použití. Skládá se z několika textových souborů a příspěvků v github „issue“ systému.

Výhody

- podpora velkého množství instrukčních sad, a to v 32 bit, 64 bit, nebo 128 bit variantě
- kvalitní dokumentace, ukázkové programy a unit testy
- využití moderního C++
- aktivní vývoj již od roku 2019

Nevýhody

- rozsáhlost knihovny způsobuje menši přehlednost a sníženou možnost vlastní modifikace

3.4.4 Souhrn

Na závěr této kapitoly této kapitoly krátce shrneme funkce všech tří knihoven pro emulaci instrukční sady, představených v předchozí části textu.

Nejnovější z těchto knihoven je *mini-rv32ima*, neobsahuje žádné pokročilé funkce, jedná se o minimalistickou „header only“ knihovnu.

Knihovna *rv32emu* oproti knihovně *mini-rv32ima* podporuje větší množství instrukčních sad a stránkovanou paměť. Její hlavní nevýhodou je závislost na dvou externích knihovnách.

Největší počet instrukčních sad poskytuje knihovna *libriscv*, zároveň také jako jediná podporuje emulaci 64 bitové a 128 bitové architektury. Výhodou je stránkovaná paměť, použití jazyka C++ a nezávislost na externích knihovnách. Oproti předchozím knihovnám se tedy *libriscv* zdá být nejlepší volbou.

Souhrnné porovnání všech jmenovaných knihoven je obsaženo v tabulce 3.1.

	mini-rv32ima	rv32emu	libriscv
Instrukční sady	IMA	IMACF	GCB
Architektura	32 bit	32 bit	32 bit, 64 bit, 128 bit
Jazyk	C	C	C++
Paměť	Lineární	Stránkovaná	Stránkovaná
Externí závisloti	Ne	Ano	Ne
„Header only“	Ano	Ne	Ne
Vývoj od roku	2022	2020	2019

Tabulka 3.1: Porovnání knihoven pro emulaci instrukční sady.

Uživatelské rozhraní 4

V této kapitole budou představeny čtyři knihovny, které bychom mohly použít pro tvorbu grafického rozhraní našeho emulátoru. Všechny z těchto knihoven jsou open source, u některých (například Qt) mohou ale licenční podmínky do určité míry omezovat jejich komerční použití.

4.1 GTK

- Dostupné z: www.gtk.org

GTK, neboli *GIMP ToolKit* je knihovna pro tvorbu multiplatfrmních uživatelských rozhraní, oficiálně podporuje GNU/Linux, MS Windows a MacOS.

Jak již název knihovny napovídá, GTK vzniklo pro potřeby grafického editoru GIMP. Vývoj GTK začal v roce 1998, původní autor knihovny jako důvod vzniku uvádí potřebu nahradit zastaralou knihovnu Motif, která byla původně používána ranými verzemi programu GIMP.

První verze GTK byla napsána v čistém C, tato verze byla prakticky použita pouze v již zmiňovaném programu GIMP. Vývoj knihovny postupem času přešel pod GNOME Project, knihovna byla v této době přepsána s využitím objektového paradigmatu *GObject*. Nejnovější verzí GTK je dnes GTK4, která oproti GTK3 a GTK2 obsahuje navíc například plnou podporu Unicode, podporu monitorů s vysokým rozlišením, možnost hardwarové akcelerace, gesta pro dotyková zařízení, apod.

Jelikož GTK vyvíjí GNOME Project, je dnes GTK známe hlavně z Linuxu, kde ho používají například desktopová prostřední GNOME, MATE, XFCE, Cinnamon. Mezi známé desktopové aplikace používající GTK patří například prohlížeč Firefox, LibreOffice, nebo video transkodér Handbrake.

Původní verze GTK byla dostupná pouze v C, dnes ale podporuje množství další jazyků, jako například C++, JavaScript, Python, Rust, Perl, Vala, atd. Samostatný projekt vycházející z GTK je GTK#, který umožňuje tvorbu GTK aplikací pomocí Microsoft .NET frameworku.

Uživatel knihovny má v nejnovější verzi GTK dostupných zhruba 200 *widgetů*.[1]

Z hlediska grafické architektury přešlo GTK postupně od používaní grafických prvků z operačního systému k ručnímu vykreslování vlastních prvků. Některé verze GTK mohou podporovat oba přístupy.

Použit GTK v jazyce C++ je podobné knihovně Qt Widgets (viz sekce 4.4). Nejprve je třeba GTK aplikaci inicializovat ve funkci *main* pomocí volání *Gtk::Application::create*, jednotlivá okna jsou pak reprezentována třídami, které dědí od *Gtk::Window*. Pro správu události používá GTK „signály", u každého widgetu si lze zaregistrovat vlastní *signal handler* metodu, která bude danou událost zpracovávat.

Grafické nástroje pro tvorbu GTK rozhraní jsou méně vyspělé než například ty co jsou dostupné pro Microsoft platformu .NET. Oficiální nástroj vyvíjený pod GNOME Project je *Glade Interface Designer*, poskytuje pouze základní *drag and drop* s možností měnit atributy prvků. Poslední verze nástroje Glade byla vydána v roce 2022.

Výhody

- stabilitní a časem ověřená knihovna
- velké množství ukázkového kódu a dokumentace

Nevýhody

- zaměřuje se především na Linux, podpora na jiných platformách může být horší
- téměř 30 let stará knihovna, podpora moderních vývojových postupů je tedy horší
- stylování grafických prvků je poměrně neprůhledné a mezi různými verzemi GTK se liší

4.2 Nuklear

- Dostupné z: github.com/Immediate-Mode-UI/Nuklear

Nuklear je knihovna umožňující tvorbu takzvaných *immediate mode* grafických rozhraní. *Immediate mode* grafické knihovny fungují tak, že si neuchovávají žádný interní stav, a okno je tak znovu vykreslováno v každém snímku. Tento přístup je odlišný od většiny moderních knihoven, které fungují na základě toho, že si interně

[1] Widgety jsou elementární grafické prvky jako tlačítka, textová pole, atd.

uchovávají stav uživatelského rozhraní, a znovu ho vykreslí jen v případě, když dojde k nějaké změně.

Nuklear je napsaný v čistém ANSI C, není závislý na žádných knihovnách třetích stran, dle autora není nutná závislost ani na standardní knihovně. Celá knihovna je tvořena zhruba 40 zdrojovými soubory o 18 tisících řádkách.

Knihovna je kompatibilní s programovacími jazyky C, C++, Java, Golang, Rust, Lua, Python, atd.

Nuklear není přímo závislý na žádném operačním systému, jako backend využívá různé grafické knihovny jako například DirectX, OpenGL, nebo SFML.

Benefitem tohoto přístupu je, že Nuklear je možné použít na jakémkoliv operačním systému, pro který je k dispozici alespoň jedna z podporovaných grafických knihoven.

Na druhou stranu tento přístup může být i nevýhodou, v případě kdy použitá grafická knihovna nepodporuje hardwarovou akceleraci (ve smyslu toho, že rozhraní bude vykreslováno grafickou kartou), může dojít ke značnému vytížení procesoru, který kvůli výše zmiňované *immediate mode* architektuře musí neustále okno znovu vykreslovat, i když nedochází k žádné změně.

Nuklear je v současné době využíván především na různých embedded systémech, pro vývoj jednoduchých her a na tvorbu grafických rozhraní určených pro interní použití.

Knihovnu je možné použít v C++ jako tzv. *single-header library* (vše je zahrnuto v jednom hlavičkovém souboru).

Jelikož je knihovna napsána v C, samotné prvky rozhraní se definují s použitím C struktur a neexistuje zde žádný objektový přistup. Často jsou zde využity obyčejné ukazatele (tzv. raw pointers) a další prostředky, které není doporučeno používat v moderním C++ kódu.

Jedná o minimalistickou open source knihovnu, která je čistě deklarativní bez použití jakéhokoliv značkovacího jazyka pro definici prvků uživatelského rozhraní, tudíž pro Nuklear není dostupný žádný „drag and drop" vývojový nástroj, jako je tomu například u Qt nebo GTK.

Z hlediska aktivity vývoje je Nuklear vyvíjen již od roku 2015, vývoj je stále velmi aktivní s několika příspěvky každý měsíc.

Výhody

- Pouze jeden hlavičkový soubor, jednoduché linkování a sestavení knihovny
- Relativně jednoduchý způsob definice grafických prvků
- Nezávislost na operačním systému

Nevýhody

- Knihovna není objektově orientovaná, použití v objektových jazycích jako C++ je tak horší
- Horší efektivita, v závislosti na typu backendu může nadměrně zatěžovat procesor
- Podpora zařízení s vysokým rozlišením nebo dotykových zařízení je horší než u velkých knihoven typu Qt nebo GTK

4.3 webview

- Dostupné z: github.com/webview/webview

Webview je knihovna pro tvorbu desktopových uživatelských rozhraní pomocí frontend webových technologií (HTML / JS / CSS).

Webview podporuje operační systémy GNU/Linux, MS Windows a MacOS. Na každém z těchto systémů používá pro interpretaci HTML, JS a CSS jiný backend, WebKitGTK pro Linux, WebKit pro MacOS a Microsoft Edge WebView2 pro Windows. Tento přístup byl poprvé popularizován knihovnou Electron, která na rozdíl od webview používá jako svůj backend na každé platformě Chromium.

Knihovna je napsaná v C++, ale zaručuje kompatibilitu s množstvím dalších jazyků jako například C#, Java, Haskell, Kotlin, Node.js, Pascal, Python, PHP, Ruby, Swift, atd.

Mezi hlavní výhody této knihovny patří explicitní separace kódu a logiky uživatelského rozhraní od zbytku aplikace. Struktura uživatelského rozhraní je definována pomocí HTML, vzhled pomocí CSS a logika pomocí Javascriptu. Webview poskytuje nástroje pro volání C++ funkcí z Javascript kódu, tím je zajištěna komunikace mezi uživatelským rozhraním a kódem naší aplikace.

Další výhodou je že o vykreslování a stylování grafických prvků se plně stará backend knihovny, to zajišťuje stabilitu a konzistentní vzhled.

Nevýhodou tohoto přístupu je, že kvůli použití webových technologií nemusí poskytovat stejnou responzibilitu jako běžné typy uživatelských rozhraní. Využití paměti bude také vyšší, protože o vykreslování se stará upravený backend webového prohlížeče.

Knihovna je ve vývoji od roku 2017, vývoj je stále aktivní s několika příspěvky v posledním měsíci.

Výhody

- jednoduché použití

- separace kódu uživatelského rozhraní od zbytku aplikace
- oproti knihovně Electron, která funguje na podobném principu, nepřibaluje do výsledného programu Chromium, velikost spustitelných souborů je tak mnohem nižší

Nevýhody

- vyšší využití paměti
- horší reponzibilita
- knihovna je vyvíjena jednotlivci, budoucí podpora tak není zaručena

4.4 Qt Widgets

- Dostupné z: www.qt.io

Qt Widgets je jedna z knihoven spadající pod Qt Framework. Qt framework se zaměřuje především na uživatelská rozhraní, pro jejich tvorbu poskytuje knihovny Qt Widgets a Qt Quick. Kromě toho obsahuje i další knihovny například pro testování, pro připojení k SQL databázi nebo pro práci se sítí.

Qt oficiálně podporuje GNU/Linux, MS Windows, MacOS, Android, iOS, různé embedded platformy, a nově i WebAssembly.

Qt Framework je ve vývoji od roku 1991, postupem času přešel od proprietární licence na open source licenci. Existuje i možnost komerční licence, která navíc poskytuje rozšířenou uživatelskou podporu a opravy chyb i po ukončení oficiální podpory dané verze Qt. Vývoj Qt momentálně spadá pod *The Qt Company*.

Qt Widgets se primárně zaměřuje na C++, dnes je ale dostupný i v dalších jazycích jako například Python, Node.js, Rust, C#, atd.

Qt Widgets se od Qt Quick liší primárně v tom, že grafické prvky si knihovna nevykresluje sama, ale využívá grafické prvky které poskytuje operační systém. To způsobuje určitou nekonzistenci mezi platformami, v novějších verzích Qt lze tento problém vyřešit pomocí stylování.

Qt Widgets je známé především z Linuxových desktop prostředí KDE Plasma, LXQt, nebo Lumina. Mezi známý komerční software vyvíjený pomocí Qt Widgets patří například Adobe Photoshop Elements, Autodesk, Google Earth desktop nebo Teamviewer.

Je k dispozici vývojový nástroj Qt Designer, který umožňuje vytvářet grafická rozhraní pomocí „drag and drop", měnit atributy prvků, apod. Jako další je k dispozici Qt Creator, který slouží jako hlavní prostředí pro vývoj Qt aplikací, poskytuje nástroje pro testování, kompilaci, apod.

Výhody

- stabilní a časem prověřená knihovna
- velké množství ukázkového kódu, dokumentace
- Qt Designer a vývojové prostředí Qt Creator

Nevýhody

- horší vzhled
- nevynucuje explicitní separaci kódu uživatelského rozhraní a samotné aplikace
- horší podpora zařízení s vysokým rozlišením

4.5 Qt Quick

- Dostupné z: www.qt.io

Qt Quick je druhá knihovna pro tvorbu uživatelských rozhraní dostupná pod Qt frameworkem. Je novější než Qt Widgets, první verze byla vydána v roce 2010.

Podpora operačních systémů je stejná jako u Qt Widgets, nemusí ale podporovat všechny programovací jazyky jako Qt Widgets. Samozřejmá je podpora C++, dalším z jazyků co stabilně podporuje Qt Quick je Python. Podporu těchto dvou jazyků zaručuje přímo *The Qt Company*.

Qt Quick na rozdíl od Qt Widgets obsahuje skriptovací jazyk QML. QML je jazyk inspirovaný jazykem Javascript a CSS. Pomocí QML je možné kompletně definovat strukturu, chování a vzhled uživatelského rozhraní. Tímto je zajištěno kompletní oddělení kódu uživatelského rozhraní od kódu aplikace, komunikace se zbytkem aplikace probíhá pomocí tzv. callback funkcí u událostí grafických prvků.

Qt Quick nepoužívá grafické prvky z operačního systému, ale vykresluje je manuálně pomocí QML engine, který za běhu interpretuje uživatelem definované QML skripty.

Existuje vývojový nástroj *Qt Quick Designer*, který umožňuje tvorbu Qt Quick rozhraní pomocí „drag and drop" přístupu.

Výhody

- explicitní separace kódu uživatelského rozhraní od zbytku aplikace
- moderní přístup k tvorbě uživatelských rozhraní

Nevýhody

- interpretovaný skriptovací jazyk

4.6 Souhrn

V rámci této sekce krátce porovnáme všech 5 knihoven pro tvorbu uživatelského rozhraní, které byly představeny v předešlých částech této kapitoly.

Mezi knihovny, které používají tzv. „widgety", patří *GTK* a *Qt Widgets*. Obě tyto knihovny jsou relativně staré (první verze vydány v 90. letech minulého století), a tak často nemusí plně podporovat „moderní" funkce jako tmavý režim, pokročilé stylování, škálování na monitorech s vysokým rozlišením, apod.

Jediným zmiňovaným zástupcem tzv. *Immediate Mode* grafické knihovny je knihovna *Nuklear*. Výhoda *Immediate Mode* přístupu je, že knihovna si nemusí uchovávat žádný interní stav okna, neboť celé okno znovu vykresluje v každém snímku. Tento přístup může být na druhou stranu i nevýhodou, v situacích kdy daný systém nepodporuje hardwarovou akceleraci, může dojít k nadměrnému zatížení procesoru.

Mezi knihovny využívající interpretovaný skriptovací jazyk patří *webview* a *Qt Quick*.

Knihovna *webview* je založena na webových technologiích (HTML, Javascript, CSS). Vykreslování rozhraní zajišťuje pomocí backendu (např. WebKit), který je poskytnut operačním systémem. Mezi nevýhody této knihovny patří vyšší využití paměti nebo horší doba odezvy.

Qt Quick používá jazyk *QML*, který byl vyvinut speciálně pro tvorbu uživatelských rozhraní. Stejně jako knihovna *webview* tak zaručuje explicitní separaci kódu uživatelského rozhraní od zbytku aplikace. Oproti knihovně *webview* si ale své rozhraní vykresluje sama, a tak je zaručen jednotný vzhled na všech podporovaných operačních systémech. *Qt Quick* podporuje sylování všech grafických prvků, dále podporuje zařízení s vysokým rozlišením nebo tmavý režim. Na základě těchto kritérií, lze knihovnu *Qt Quick* považovat za jednu z nejlepších knihoven pro tvorbu moderních uživatelských rozhraní.

Výchozí implementace a použité technologie

5

V této kapitole postupně zhodnotíme výchozí implementaci zpracovanou v rámci semestrální práce (více informací viz kapitola 1), budou zde popsány nedostatky a špatná rozhodnutí původní verze, ze kterých se můžeme poučit a vyhnout se jim při tvorbě našeho software (viz kapitola 6).

Dále zde bude vytvořen seznam a odůvodnění technologií, které byly vybrány pro implementaci v kapitole 6. Vycházíme zde ze znalostí získaných v kapitolách 4 a 3.

5.1 Zhodnocení výchozí implementace

Všechny zdrojové soubory projektu vypracovaného v rámci semestrální práce jsou poskytnuty jako příloha tohoto dokumentu.

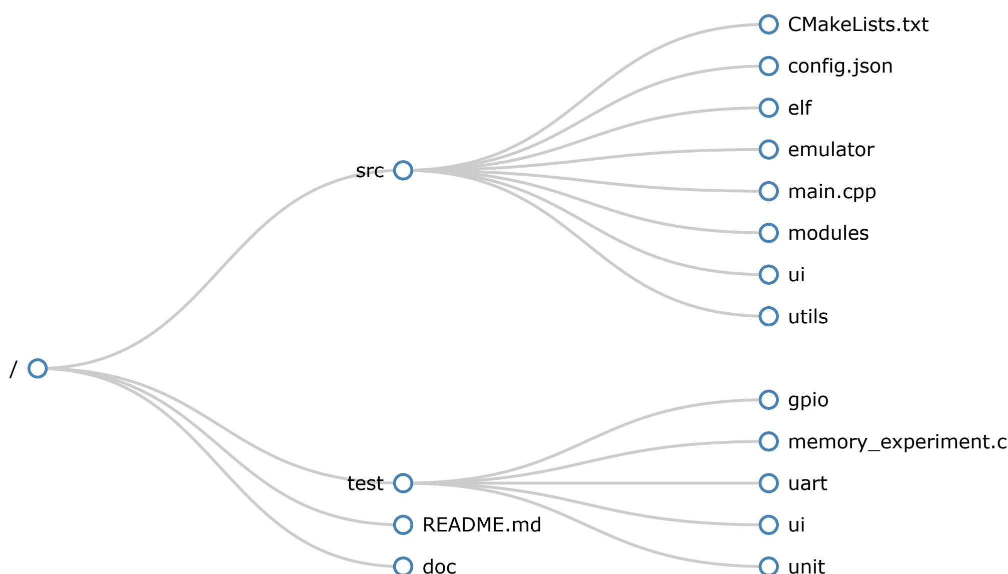
5.1.1 Dekompozice

Kořenový adresář projektu byl dekomponován do tří složek: `./src`, `./test` a `./doc`, podrobnou strukturu včetně vnořených složek a souborů můžeme vidět na obrázku 5.1.

Složky `./doc` a `./test` obsahovaly podpůrný materiál, který není součástí zkompilovaného programu. Tento postup tak byl správný, až na to, že složka `./test` neobsahovala jen jednotkové testy, ale i zdrojové soubory pro programy, které byly spouštěny uvnitř emulátoru. Zkompilované verze těchto programů se pak nacházely ve složce `./src/elf`.

Tyto soubory se tak nacházely na neobvyklých místech, čímž byla adresářová struktura projektu znepřehledněna.

Jelikož je tento projekt napsán v jazyce **C++**, tak dalším problémem byla separace implementačních a hlavičkových souborů. Složka `./src` sice byla rozdělena



Obrázek 5.1: Adresářová struktura výchozí implementace, zobrazeno do hloubky 2 od kořene (tzn. vnořené položky hloubky 3 a více nejsou zobrazeny)

na několik podsložek, v každé z nich ale byly implementační i hlavičkové soubory umístěny společně. Pro programátora je toto rozdělení nepříjemné a snižuje efektivitu práce, protože nejčastěji se snaží hledat právě soubory implementační, a ne hlavičkové.

Jako poslední byly zavádějící i názvy některých složek a souborů. To platilo například pro složku `./src/modules`, která obsahovala soubory týkající se periferních zařízení. Jmenný prostor (`namespace`) použit ve všech souborech unitř této složky se taktéž nazýval „modules“, oproti tomu ve stejné složce existovala například třída `PeripheralDevice`, jejíž instance pak byly vytvářeny v souboru `./src/ui/peripherals/PeripheralsWidget.cpp`. Pojmenování je tak značně matoucí, programátor by si mohl myslet, že mezi termíny „module“ a „peripheral“ existuje nějaký rozdíl, ve skutečnosti tomu tak není, oba termíny totiž popisují výhradně soubory zabývající se periferními zařízeními.

5.1.2 Systém pro sestavení programu

Sestavení programu bylo zajištěno pomocí open source software pro automatizaci překladu **CMake**. Celý projekt obsahoval tři `CMakeLists.txt` soubory, dva ve složce `./test`, a jeden ve složce `./src`.

Existovaly tak tři samostatné CMake projekty (každý soubor si definoval svůj vlastní). Tyto projekty byly na sobě plně nezávislé, tzn. každý z nich si vytvářel

vlastní cache složku a načítal se samostatně, to výrazně zpomalovalo celý vývojový proces. Pro testy by stačil pouze jeden projekt, a uvnitř něj pak testovací sady definovat jako tzv. ***CMake targety***, nebo lépe, mít jeden CMake projekt pro celou aplikaci, a jednotlivé části pak fragmentovat do targetů.

Nejrozsáhlejší CMake soubor `./src/CMakeLists.txt` definoval instrukce pro linkování, překlad všech knihoven a zdrojových souborů aplikace. Neexistovala zde žádná dekompozice na více menších CMake souborů v podsložkách `./src/`, jak tomu typicky bývá zvykem. Nevhodné je ale už i samotné umístění tohoto `CMakeList.txt` souboru, měl by být spíše umístěn v kořenovém adresáři, protože umístění do kořenového adresáře umožní tento soubor automaticky detekovat vývojovým prostředím jako **Microsoft Visual Studio** nebo **CLion**.

Protože projekt používal externí knihovny jako například **Qt**, bylo nutné nějakým způsobem předat CMake cestu k těmto knihovnám, pokud to programátor neudělal, tak kompilace selhala. Aby byla kompilace úspěšná, tak musel programátor ručně zadávat proměnné jako *CMAKE_PREFIX_PATH* apod. buď do příkazové řádky, nebo do uživatelského rozhraní vývojového prostředí. Tento proces může být časově náročný a provádí se jinak v závislosti na tom, jaké vývojové prostředí programátor používá. Správným řešením tohoto problému je vytvoření souboru `CMakePresets.json`, který na jednom místě uchovává všechny CMake proměnné (o tomto více v kapitole 6).

Další, co projekt postrádal byl třeba CMake skript pro instalaci aplikace, nebo skript pro automatické vygenerování programové dokumentace.

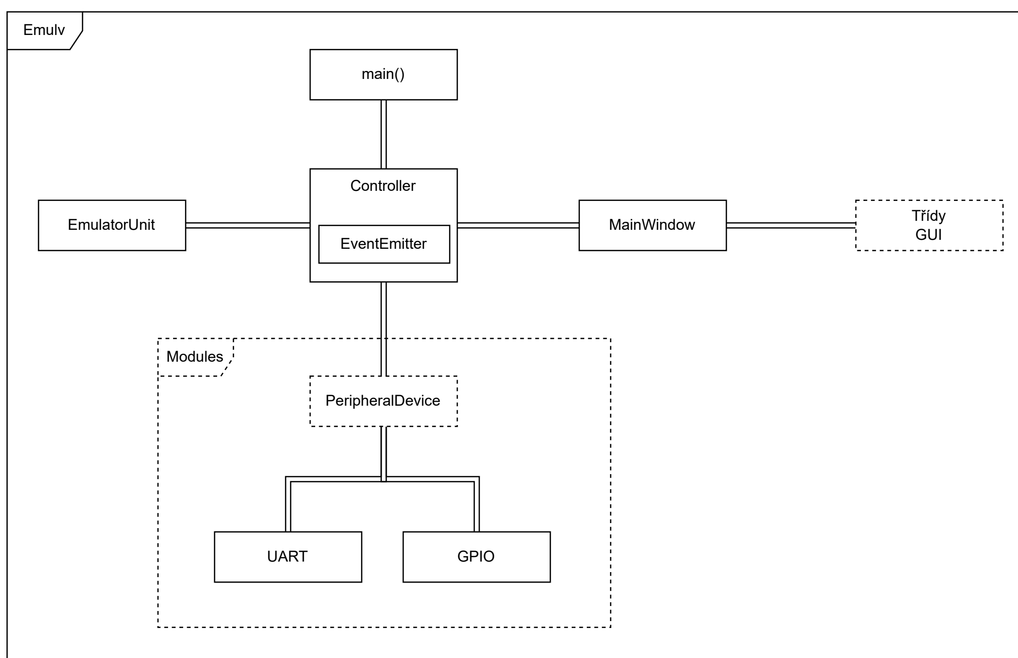
5.1.3 Architektura

Architektura původní aplikace je popsána na obrázku 5.2, jak můžeme vidět, skládá se ze tří hlavních částí: `EmulatorUnit`, `Controller` a `MainWindow`.

Třída `EmulatorUnit` obsahuje všechny funkce spojené s emulací procesoru, třída `MainWindow` je hlavní třídou uživatelského rozhraní a třída `Controller` zajišťuje komunikaci mezi těmito částmi.

Koncepčně toto rozdělení dává smysl, prakticky se ale ukázalo, že zajistit komunikaci mezi různými částmi aplikace není jednoduchým úkolem. Nejprimitivnější způsob by bylo si uchovávat v každé třídě referenci na všechny třídy se kterými chceme komunikovat, to ale dobrým řešením není, každá z těchto tříd je vytvářena v jiný čas, a tak bychom museli vytvořit nějaký systém, pomocí kterého bychom postupně každé z nich předávali potřebné reference, v aplikaci by tak vznikl dlouhý řetěz vzájemných závislostí.

Proto byla vytvořena třída `EventEmitter`, která slouží jako jednotný komunikační kanál pro události vyvolané třídami. `EventEmitter` poskytuje dvě metody: `on(eventName, listener)` a `emit(eventName, data)`. Pomocí metody



Obrázek 5.2: Architektura výchozí implementace, funkce `main` po spuštění vytvoří instanci třídy `Controller`, `Controller` inicializuje všechny ostatní třídy a zprostředkovává mezi nimi komunikaci. `Controller` si vytvoří instanci třídy `EventEmitter`, referenci na tento objekt předává periferiím, a třídám `EmulatorUnit`, `MainWindow`. `PeripheralDevice` je abstraktní třída, ze které dědí UART a GPIO.

`on(eventName, listener)` je možné si zaregistrovat „listener" funkci, která bude spuštěna pokaždé, když nastane událost s názvem `eventName`. Pomocí metody `emit(eventName, data)` je pak možné vyvolat událost `eventName` s parametrem `data`.

Ačkoliv se tento přístup zdá být lepší, tak stále neodstraňuje problém předávání reference, která musí být explicitně předána všem třídám, které chtějí `EventEmitter` použít.

Druhým nedostatkem tohoto systému bylo předávání dat událostí, každá z událostí může přenášet jiná data, a tak nelze tento parametr pevně nastavit na jeden datový typ. Například událost zmáčknutí tlačítka žádná další data přenášet nepotřebuje, vše je jasné již z jejího názvu, oproti tomu třeba událost změny stavu GPIO pinu potřebuje přenášet data jako číslo pinu, aktuální režim (*vstup / výstup*), apod.

Nakonec bylo rozhodnuto, že datovým typem parametru `data` bude abstraktní třída `AbstractEvent`, každá událost pak má vlastní třídu, která dědí z `AbstractEvent`, a popisuje data přenášená danou událostí. Tím však vzniklo dalších 10 souborů, které nepřináší žádnou novou funkcionalitu a jsou tak víceméně zbytečné.

Řešení problémů spojených se třídu `EventEmitter` je dále popsáno v kapitole 6.

5.1.4 Periferní zařízení

Jak bylo popsáno v předchozí podkapitole (obr. 5.2), tak projekt obsahuje abstraktní třídu `PeripheralDevice`, která slouží jako jednotné rozhraní pro všechny periferní zařízení.

Proces inicializace a použití periferních zařízení byl následující:

1. Uživatel načte konfigurační soubor
2. Podle konfiguračního souboru se uvnitř třídy `Controller` vytvoří objekty periferních zařízení
3. Zařízením je předána reference na `EventEmitter`
4. Třída `Controller` předá referenci na seznam zařízení třídě `EmulatorUnit`, `EmulatorUnit` tak nad zařízeními může provádět všechny funkce abstraktní třídy `PeripheralDevice`
5. Zařízení šíří své události pomocí `EventEmitteru`, třídy GUI tyto události zachycují, a podle toho upravují uživatelské rozhraní

Periferní zařízení jsou tak závislé na `EventEmitteru` a uživatelském rozhraní, které je definované odděleně od jejich implementace. K tomu, abychom mohli `EventEmitter` použít, musíme navíc ještě pro každou událost těchto zařízení definovat potomka třídy `AbstractEvent`.

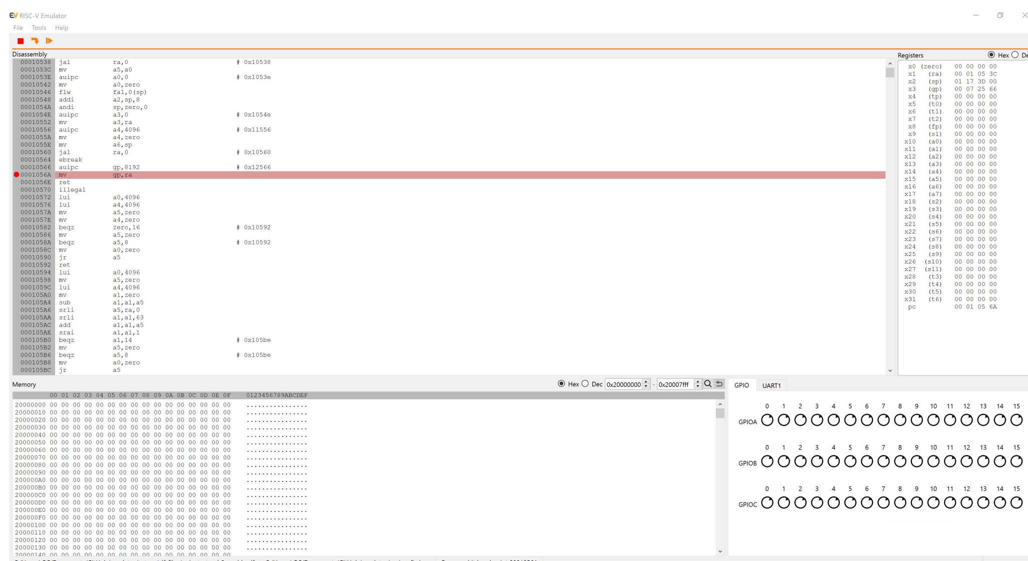
To znamená, že tato koncepce neumožňuje téměř žádnou modularizaci. Pro definování nového periferního zařízení se neobejde bez znatelných uprav několika částí této aplikace.

5.1.5 Uživatelské rozhraní

Uživatelské rozhraní této verze programu bylo zhotoveno v knihovně **Qt Widgets**.

Neexistuje zde žádná separace logiky uživatelského rozhraní od logiky „backendu“, celé rozhraní je implementováno imperativně v jazyce C++ a rozděleno zhruba do 25 souborů. Často se zde lze setkat s tím, že jeden soubor obsahuje například algoritmus pro zpracování dat, a zároveň i sekce kódu, co ovlivňují pouze uživatelské rozhraní. Nejsou zde tedy splněny základní nároky na udržitelnost a čitelnost zdrojového kódu.

Struktura rozhraní je definována v metodě `MainWindow::SetupUI()`, často se zde pracuje s ukazateli na různé interní objekty Qt Widgets, ukazatele na widgety vytvořené v již zmiňované metodě `SetupUI()` jsou pak uloženy jako atributy třídy `MainWindow`. Definice tohoto rozhraní je tedy velmi komplikovaná ať už z logického, nebo syntaktického hlediska (viz výpis 5.1), například pro to, aby bylo do rozhraní přidáno nové tlačítko, se musí změnit několik metod a atributů v různých souborech.



Obrázek 5.3: Ukázka uživatelského rozhraní původní implementace

Mezi další problémy tohoto rozhraní patřil například nekonzistentní vzhled na různých operačních systémech, nebo špatné vykreslování na zařízeních s vysokým rozlišením.

Zdrojový kód 5.1: Ukázka nadměrného počtu ukazatelů ve třídě `MainWindow`

```
class MainWindow : public QMainWindow {
Q_OBJECT
public:
QToolButton *btn_terminate_{}, *btn_continue_{}, *btn_run_{}, *btn_debug_{}, *btn_step_{};
QFrame *running_indicator_{}, *debug_indicator_{};
QSplitter *main_splitter_{}, *top_splitter_{}, *bot_splitter_{};
QLabel *lbl_file_{}, *lbl_config_{}, *lbl_program_status_{};

DisassemblyWidget *disassembly_widget_{};
RegistersWidget *registers_widget_{};
MemoryWidget *memory_widget_{};
PeripheralsTabWidget *peripherals_tab_widget_{};

explicit MainWindow(QWidget *parent = nullptr, Controller *controller = nullptr);
~MainWindow() override;
```

5.2 Zvolené technologie

Na základě dosavadních poznatků se nyní můžeme rozhodnout o tom, jaké technologie zvolit pro implementaci emulačního software, který bude zpracován v rámci této bakalářské práce.

Procesorová část emulátoru bude implementována pomocí knihovny **libriscv** (viz sekce 3.4.3), jedná se o časem prověřenou knihovnu, která oproti jiným knihovnám zmíněným v kapitole 3, podporuje největší počet instrukčních sad, objektové paradigma v jazyce C++, možnost použití paměťově mapovaných periferních zařízení, apod. Pro účely této bakalářské práce je tedy plně dostačující, a není třeba ji oproti výchozí implementaci měnit.

Pro uživatelské rozhraní byla zvolena knihovna **Qt Quick**, oproti nevyhovující knihovně **Qt Widgets**, která byla použita ve výchozí implementaci, je knihovna Qt Quick modernější, a podporuje většinu dnes žádaných funkcionalit jako například tmavý režim, „plovoucí" okna, podporu zařízení s vysokým rozlišením, jednotný vzhled napříč operačními systémy, apod. Další z důvodů pro toto rozhodnutí je také jazyk QML, který explicitně odděluje definici struktury a logiky uživatelského rozhraní od zbytku aplikace. Qt Quick se vyhýbá problémům spojeným s knihovnami používající technologii „webview", jako je nadměrná velikost spustitelných souborů nebo vysoké využití paměti. Qt Quick zároveň není „malou" knihovnou která by byla vyvíjena jednotlivci, a tak je zaručeno, že bude i do budoucna podporována. Qt Quick se tedy v současné době jeví jako jedna z nejlepších knihoven pro vývoj multiplatformních aplikací.

Jako systém pro sestavení programu bude i nadále používán **CMake**, který je v open source komunitě de facto standardem pro vývoj multiplatformních C++ aplikací. Pro testování bude použita knihovna **Google Test** [26], pro generování programové dokumentace bude využito knihovny **Doxygen** [27], systém CMake má pro obě tyto knihovny integrovanou podporu.

Mezi další knihovny využívané v této práci patří knihovna **spdlog** [28], poskytující logovací výpisy, **riscv-disassembler** [29] pro účely zobrazení *disassembly* programu v uživatelském rozhraní, knihovna **JSON for Modern C++** [30] používaná pro načtení konfiguračního souboru emulátoru, a knihovna **qwindowkit** [31] umožňující úpravy standardního Qt Quick okna.

Realizovaná implementace — 6

V této kapitole bude představen emulační software, který byl vyvinut pro účely této bakalářské práce. Nejprve budou popsány adresáře a architektura tohoto programu, následně budou popsány nejdůležitější třídy a metody. Cílem této kapitoly je popis čistě z programátorského hlediska, návod k použití psaný z pohledu koncového uživatele je dostupný v příloze A.

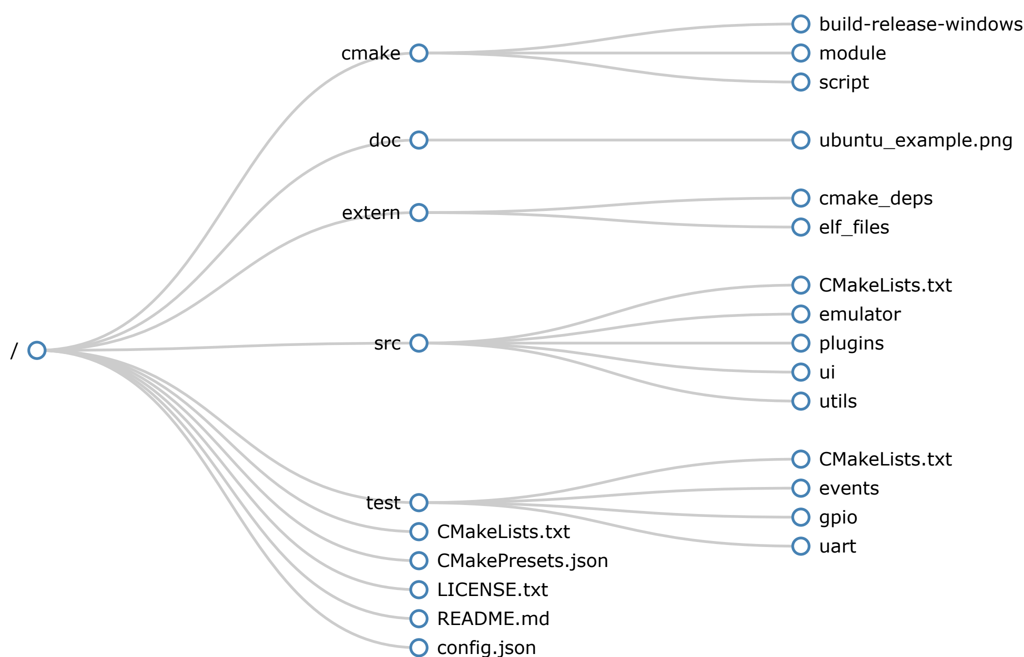
Pro lepší orientaci v následujícím textu zde uvedeme seznam funkcí, které vyvinutý software poskytuje:

- Emulační jednotka
  - Podpora instrukčních sad `RV32GCB`, `RV64GCB` (knihovna libriscv)
  - Podpora paměťově mapovaných periferních zařízení
  - Vstupní program je dodán ve formě ELF souboru
  - Vestavěný validátor ELF souborů
  - Stránkovaná paměť
  - Přístup k ladícím funkcím: krokování, breakpointy, výpis registrů, výpis paměti
- Periferní zařízení
  - Aktuálně jsou k dipozici GPIO a UART
  - Možnost vytvoření vlastních periferních zařízení
  - Všechna periferní zařízení jsou implementována jako dynamické knihovny a k emulátoru připojena za běhu
  - Dynamická knihovna si sama implementuje a ovládá svůj grafický komponent, který je pak připojen k uživatelskému rozhraní
  - Informace o dynamických knihovnách a paměťovém prostoru periférií jsou načteny z konfiguračního souboru ve formátu `JSON`

- Uživatelské rozhraní
  - Implementováno v knihovně Qt Quick, hardwarově akcelerováno pomocí knihovny OpenGL
  - Plná podpora zařízení s vysokým rozlišením, všechny ikony jsou ve formátu `SVG`
  - Možnost načíst program z ELF souboru, pro tento účel je k dispozici dialogové okno a dvě tlačítka v levé části okna
  - Okno obsahuje nástroje pro ladění programu: tlačítka „Debug", „Continue", „Step", poté tlačítko „Memory", pomocí kterého je možné zobrazit výpis paměti a tlačítko „Registers", které umožňuje zobrazení registrů procesoru
  - Textová oblast uprostřed okna, zobrazující *disassembly* načteného programu, s možností vkládat breakpointy a exportovat obsah do textového souboru
  - Okno je možné zobrazit buď v tmavém nebo světlém barevném motivu

6.1 Adresářová struktura a CMake

Nová struktura projektu, která odstraňuje nedostatky původní verze, byla vytvořena na základě doporučení obsažených v knize *Modern CMake for C++* [32], adresářová



Obrázek 6.1: Adresářová struktura projektu (zobrazeno do hloubky 2 od kořene)

struktura projektu je zobrazena na obrázku 6.1, hlavní 4 složky projektu jsou:

- `cmake`
- `extern`
- `src`
- `test`

Složka `./cmake` obsahuje všechny pomocné soubory a adresáře, které jsou využívány systémem CMake.

Uvnitř složky `./src/cmake/module` se nacházejí CMake „moduly“, které je možné importovat pomocí příkazu `include()` do jakéhokoliv `CMakeLists.txt` souboru. Jedním z těchto souborů je například `./src/cmake/module/GetLibrisv.cmake`, který obsahuje kód pro stažení knihovny libriscv. Tento postup zabraňuje výskytu duplikovaného kódu uvnitř `CMakeLists.txt` souborů, a zároveň tyto soubory zpřehledňuje.

Další soubory týkající se sestavení aplikace se nacházejí uvnitř složky `./src/cmake/script`, jedná se o pomocné skripty, které nejsou přímo importovány příkazem `include()`.

Do složky `./cmake` jsou také umisťovány zkompilované soubory aplikace a CMake cache, tyto složky mají různé názvy podle typu sestavení, například `build-release-windows`, `build-release-linux`, `build-debug-windows`, apod.

Složka `./extern` obsahuje všechny *externí knihovny* a další zdrojové soubory, co nejsou součástí projektu. Externí knihovny stažené pomocí CMake budou umístěny do složky `./extern/cmake_deps`, normálně by tyto knihovny byly umístěny do stejné složky jako soubory přeloženého programu (např. `build-release-windows`). Výhodou umístění do složky `./extern/cmake_deps` tedy je to, že složku s přeloženým programem můžeme smazat bez toho, aniž bychom museli znovu stahovat všechny externí knihovny.

Ve složce `./src` se nacházejí všechny zdrojové soubory projektu. Soubory jsou organizovány do několika podsložek, tam kde to bylo vhodné, byly vytvořeny složky `include`, do těchto složek jsou umisťovány hlavičkové soubory, například tedy hlavičkový soubor náležící k souboru `./src/ui/EmulvApi.cpp` se je umístěn v `./src/ui/include/EmulvApi.h`. Tento krok výrazně zpřehlednil strukturu projektu, hlavičkové a implementační soubory jsou nyní od sebe explicitně odděleny.

Jako poslední se zde nachází složka `./test`, která obsahuje jednotkové testy tříd tohoto programu.

6.1.1 Specifika spojená s CMake

Všechny `CMakeLists.txt` soubory jsou nyní uspořádány hierarchicky. Tedy každá „logická část“ programu má nyní svůj `CMakeLists.txt` soubor. To znamená, že `CMakeLists.txt` nejvyšší úrovně (v kořenové složce) pouze „volá“ CMake soubory ve vnořených složkách pomocí příkazu `add_subdirectory()`. Každý `CMakeLists.txt` si pak pro své zdrojové soubory definuje tzv. ***CMake target***[1]. Tímto přístupem jsme přesunuly instrukce pro překlad co nejblíže samotným zdrojovým souborům, místo toho aby vše bylo v jednom velkém CMakeLists.txt souboru, jsou nyní instrukce pro překlad ve stejné složce jako dotčené zdrojové soubory.

Dalším užitečným nástrojem je soubor `CMakePresets.json` v kořenovém adresáři. Pomocí tohoto souboru lze definovat proměnné a parametry CMake, které by jinak musely být manuálně zadávány do příkazové řádky. Programátor pak může do příkazové řádky zadat jen `cmake --preset <jméno>` pro zvolení daného *presetu*. Ukázku můžeme vidět ve výpisu 6.1.

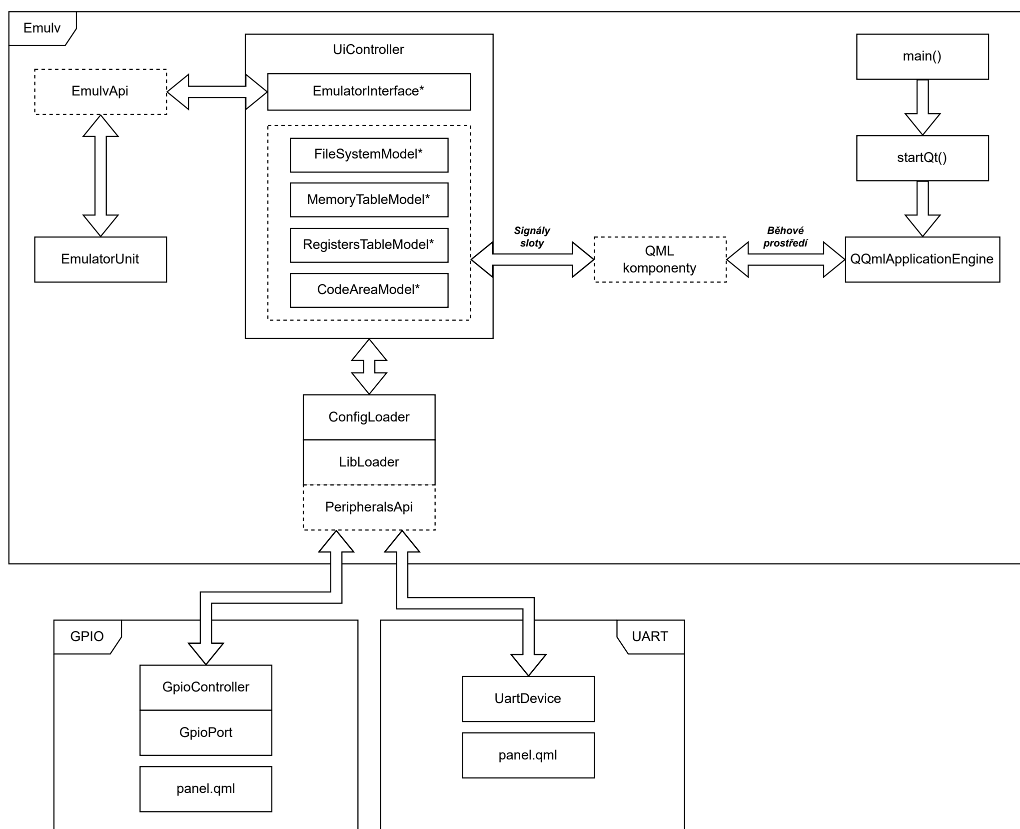
Zdrojový kód 6.1: Definice CMake *presetu* „Debug-Windows“

```
1  {
2  "name": "Debug-Windows",
3  "displayName": "Debug for Windows",
4  "description": "Basic debugging build for Windows. Remember to change the compiler paths in the CMakePresets.json file.",
5  "generator": "Ninja",
6  "binaryDir": "\${sourceDir}/cmake/build-debug-windows",
7  "cacheVariables": {
8  "CMAKE_BUILD_TYPE": "Debug",
9  "CMAKE_PREFIX_PATH": "C:/Qt/6.6.2/mingw_64",
10 "CMAKE_C_COMPILER": "C:/Qt/Tools/mingw1120_64/bin/gcc.exe",
11 "CMAKE_CXX_COMPILER": "C:/Qt/Tools/mingw1120_64/bin/g++.exe",
12 "CMAKE_CXX_FLAGS_INIT": "-DQT_QML_DEBUG"
13 }
```

6.2 Architektura

Architekturu této aplikace (viz obr. 6.2), můžeme rozdělit na tři části: `QML`, `ConfigLoader` a `EmulatorUnit`. Každá z těchto částí se liší v tom, jakým způsobem komunikuje se třídou `UiController`.

[1] Zjednodušeně řečeno, target je samostatně kompilovatelnou jednotkou, jejímž výstupem může být buď spustitelný soubor, nebo *object file*.



Obrázek 6.2: Architektura výsledné aplikace. Je zde zobrazen samotný emulátor (Emulv) a dvě dynamické knihovny (GPIO a UART). Jádrem emulátoru je třídá UiController, která propojuje jednotlivé části aplikace. Funkce `main()` v tomto případě pouze volá funkci `startQt()`, která inicializuje uživatelské rozhraní

Třída `UiController` dědí ze třídy `QObject` a zároveň je nastavena jako `QML_SINGLETON`, to znamená, že její instance je automaticky vytvořena objektem `QQmlApplicationEngine`, který je inicializován v rámci funkce `main()`. Díky tomu je možné uvnitř třídy `UiController` používat *sloty* a *signály* poskytované knihovnou Qt, a další metody integrace s `QML` rozhraním. `UiController` řeší čistě jen komunikaci a přenos dat, nikoliv definici struktury uživatelského rozhraní, ta je plně definovaná uvnitř `QML` souborů a všechny inicializační kroky jsou provedeny běhovým prostředím `QML`.

`UiController` si uchovává ukazatele (ve formě C++ chytrých ukazatelů), na třídu `EmulatorInterface` a na modely, pomocí kterých jsou poskytována data `QML` komponentům. S QML komponenty a instancemi modelů komunikuje `UiController` pomocí *signálů* a *slotů*. Zjednodušeně řečeno, je *slot* funkce, kterou lze propojit s určitým *signálem*, po vyvolání signálu pak budou spuštěny všechny sloty, které jsou s ním propojeny. Mezi výhody tohoto systému patří například to, že signály

lze bezpečně použít ve vícevláknové aplikaci (Qt zajistí, že do slotu bude vždy přistupovat jen jedno vlákno na jednou), dále je pak pomocí systému signálů a slotů možné také odstranit vzájemné závislosti mezi třídami[2].

Pomocí statické třídy `ConfigLoader`, je `UiController` schopen načíst konfigurační `JSON` soubor[3], na základě kterého, třída `ConfigLoader` načte dynamické knihovny a vytvoří ukazatele na jejich objekty. Třída `UiController` je tak plně odstíněna od procesu načítání dynamických knihoven, třída `ConfigLoader` ji pouze předá seznam ukazatelů na objekty načtených knihoven.

Komunikace `UiControlleru` s třídou `EmulatorUnit` je zajištěna prostřednictvím třídy `EmulatorInterface`, která implementuje rozhraní `EmulvApi`. Cílem tohoto rozhraní bylo oddělit interní strukturu `EmulatorUnit` od zbytku aplikace. Například metoda `EmulatorInterface::configureEmulator` ve skutečnosti volá 4 různé metody třídy `EmulatorUnit`, díky přítomnosti tohoto rozhraní se tedy kód uvnitř `UiControlleru` výrazně zjednoduší.

Problémem ale zůstává notifikovaní třídy `UiController` o událostech třídy `EmulatorUnit`. Pro tento účel je zde stále využíván `EventEmitter`, který již není zobrazen na obrázku 6.2, protože nově existuje statický `GlobalEmitter` a tak není potřeba mezi třídami předávat jeho referenci. Použití `EventEmitteru` je kompromisem, jelikož je `EventEmitter` nyní používán jen pro účely třídy `EmulatorUnit`, mohly bychom ho odstranit a nahradit třeba systémem *slotů* a *signálů* z knihovny Qt. Na druhou stranu, tím, že využíváme `EventEmitter` je třída `EmulatorUnit` plně nezávislá na knihovně Qt, to může být výhodou pro případné budoucí verze této aplikace.

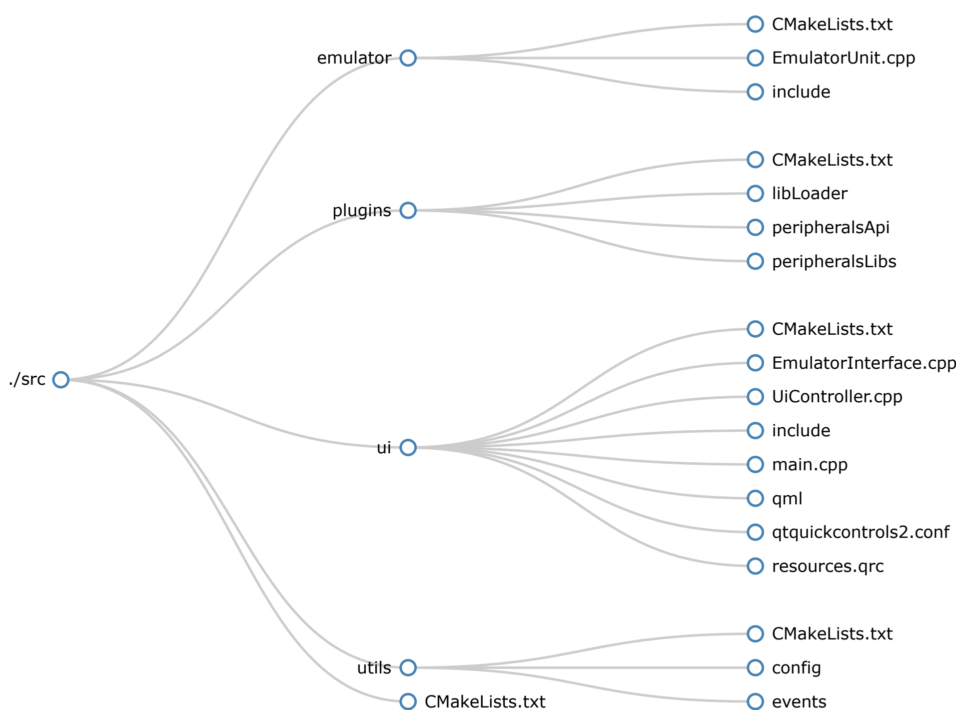
6.3 Emulační jednotka

Třída `EmulatorUnit` implementuje všechny metody potřebné pro emulaci procesoru a paměťového systému. Třída může být v různých stavech podle toho, jakou činnost právě emulátor vykonává, o změně stavu informuje pomocí globálního `EventEmitteru`.

`EmulatorUnit` je závislý pouze na `EventEmitteru` a knihovně libriscv, teoreticky ho tedy lze použít i bez grafického rozhraní. Nejdůležitější metodou této třídy je `LoadElfFile`, která umožňuje načíst program ve formátu ELF, metoda provede validaci ELF souboru, pokud je soubor neplatný, tak je vyhozena výjimka. Následně

[2] např. `FileSystemModel` by tak bylo možné použít i s jinou třídou než `UiController`, bez nutnosti ji jakkoliv modifikovat, protože jediné, co pro komunikace se třídou `UiController` používá, jsou signály a sloty

[3] příklad tohoto souboru se nachází v `./src/config.json`, lze ho načíst pomocí uživatelského rozhraní (viz sekce 6.5)



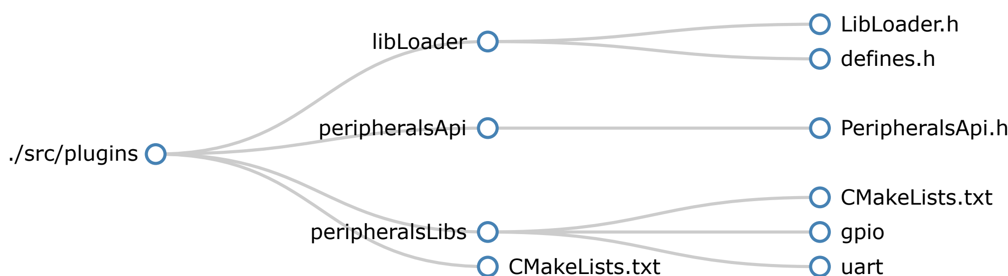
Obrázek 6.3: Pohled do složky `./src` (zobrazeno do hloubky 2)

je program možné spustit pomocí metody `Execute`, po ukončení programu je pak možné získat stav registrů pomocí metody `GetRegisters`.

Program je možné spustit také v ladícím („debug“) režimu prostřednictím metody `Debug`. V tomto případě je program pozastaven, když instrukční cyklus narazí na breakpoint, pokračovat lze buď krokováním (metoda `DebugStep`) nebo obnovením běhu (metoda `DebugContinue`). Pro práci s breakpointy jsou zde připraveny metody `AddBreakpoint`, `RemoveBreakpoint` a `ClearBreakpoints`. Při krokování lze kdykoliv získat obsah registrů nebo paměti (metoda `GetMemoryPages`). *Disassembly* načteného programu lze získat pomocí metody `Disassemble`.

Periferní zařízení lze připojit pomocí metody `RegisterPeripherals`, `EmulatorUnit` si interně zaregistruje jejich adresní prostor pomocí metod `PageTrapHandler_`, `SetupMemoryTraps_`, `GetRealDevice_` a `MapDeviceToPage_`.

Z původní implementace byly převzaty základní metody pro inicializaci `EmulatorUnit` a metody týkající se „debug“ režimu. Metody pro obsluhu periferních zařízení a paměti musely být modifikovány, jelikož jejich původní verze byly nevyhovující.



Obrázek 6.4: Pohled do složky `./src/plugins` (zobrazeno do hloubky 2)

6.4 Periférie a dynamické knihovny

Pro účely dynamicky linkovaných knihoven a periferních zařízení byla vytvořena složka `./src/plugins` (viz obr. 6.4). Soubory jsou zde rozděleny do tří podsložek: `libLoader`, `peripheralsApi` a `peripheralsLib`.

Složka `libLoader` obsahuje šablonovou třídu `LibLoader`, která dokáže za běhu načíst a zkonstruovat objekt typu `T`[4] z dynamické knihovny. Tato třída tedy umožňuje relativně snadnou modularizaci emulátoru, bez toho, aniž bychom museli program znovu kompilovat lze načíst jakékoliv periferní zařízení definované v dynamické knihovně.

Třída `LibLoader` se skládá ze tří hlavních metod: `openLib`, `getInstance` a `closeLib`. Cesta k dynamické knihovně je třídě předána pomocí konstruktoru, metoda `openLib` tuto knihovnu otevře a uloží na ni ukazatel do atributu `_handle`. Metoda `getInstance` vytvoří instanci třídy `T`, obsažené uvnitř dynamické knihovny, návratovou hodnotou je chytrý ukazatel na tento objekt. Metoda `closeLib` knihovnu uzavře a odstraní ji z adresního prostoru procesu.

Soubor `defines.h` je využíván třídou `LibLoader` pro zajištění kompatibility se systémy MS Windows a GNU/Linux. Oba operační systémy používají pro práci s dynamickými knihovnami jiné funkce standardní knihovny (např. `LoadLibrary` a `dlopen`), řešením tedy je využití direktiv preprocesoru `#define`, `#if`, `#elif` apod. pro zvolení správného systémového volání pro náš operační systém v době kompilace. Ukázka viz výpis 6.2.

Zdrojový kód 6.2: Ukázka definování maker preprocesoru na základě typu operačního systému. Tyto makra jsou používána třídou `LibLoader`. Pro stručnost byla vynechána část pro Linux, která definuje stejná makra, jen s linuxovou verzí příslušných systémových volání.

```
1    #if defined(WIN32)
```

[4] `T` je zde šablonový parametr třídy, v našem případě je `T` vždy instance třídy implementující `PeripheralsApi`

```
#include "Windows.h"
#include <system_error>
using HANDLE_TYPE = HMODULE;

#define my_dlopen(path) LoadLibraryA(path)
#define my_dlsym(handle, symbol) GetProcAddress(handle,
symbol)
#define my_dlclose(handle) FreeLibrary(handle)
#define has_dlclose_failed(return_value) return_value ==
0

#define error_msg std::string("Dll␣error:␣") + std::
system_category().message(GetLastError())
#elif defined(__linux__)

...

#else
#error "Unsupported␣platform"
#endif
```

Soubor `PeripheralsApi.h` definuje abstraktní třídu `PeripheralsApi`, která dědí od třídy `QObject`. Tato třída je jednotným rozhraním pro třídy periferních zařízení definovaných v dynamických knihovnách, zároveň díky dědičnosti z `QObject` je možné v této třídě použít *signály*, *sloty* a další prostředky knihovny Qt pro obsluhu grafického rozhraní spojeného s daným periferním zařízením. Třída obsahuje metody pro zápis a čtení z interní paměti periferního zařízení (tyto metody jsou volány při uvnitř třídy `EmulatorUnit`), dále je zde přítomna metoda `getQML`, která vrací `QML` soubor definující strukturu uživatelského rozhraní periferie.

Ve složce `peripheralsLib` se nacházejí dynamické knihovny `gpio` a `uart`. Tyto knihovny jsou kompilovatelné nezávisle na tomto projektu, jediné, co potřebují mít k dispozici, je soubor `PeripheralsApi.h` a knihovnu Qt nainstalovanou na počítači provádějící kompilaci.

Pro pochopení, jak tyto dynamické knihovny fungují, se můžeme podívat například na knihovnu `uart`. Klíčový je zde soubor `CMakeLists.txt`, který kromě instrukcí pro překlad a linkování, obsahuje i příkaz `qt_add_resources()`, který nám zpřístupní prostředky ze složky `resources`, tyto prostředky lze následně použít v QML a C++ kódu. Soubory nacházející se uvnitř podsložky `resources` jsou `panel.qml` s definicí rozhraní (je vracen metodou getQML) a ikona `send-icon.svg`, která je interně využívána souborem `panel.qml`. Reference na instanci třídy `UartDevice` (definované v souboru `uart.cpp`), je vložena do souboru `panel.qml`, když je po načtení knihovny tento panel přidáván do uživatelského rozhraní emulátoru třídou `UiController`. Panel se třídou `UartDevice` komunikuje výhradně pomocí

signálů a *slotů*.

V této části aplikace nebyl použit téměř žádný kód z původní implementace (kromě některých interních struktur v knihovnách GPIO a UART), jelikož přístup aplikovaný v původní verzi nesplňoval požadavky na použití dynamických knihoven.

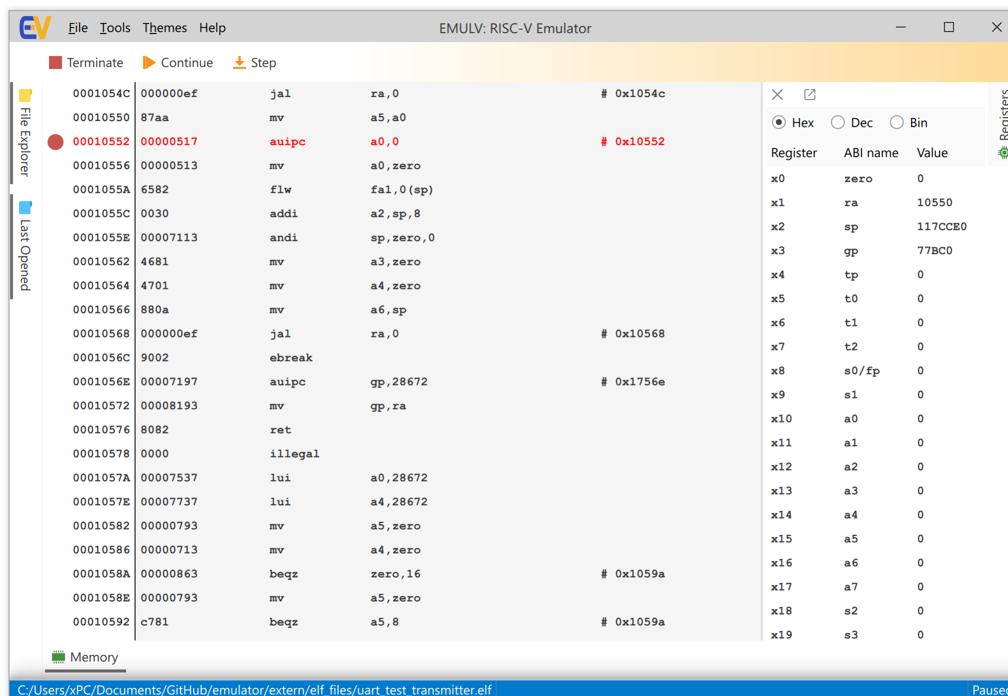
6.5 Uživatelské rozhraní

V této části textu budou představeny implementační detaily uživatelského rozhraní emulátoru, pro ilustraci je zde obrázek 6.5 s výslednou podobou tohoto rozhraní. Souborovou strukturu části aplikace zabávající se uživatelským rozhraním můžeme vidět na obrázku 6.6, podstatná je zde složka `./src/ui/qml`, která obsahuje všechny zdrojové soubory týkající se logiky a struktury uživatelského rozhraní.

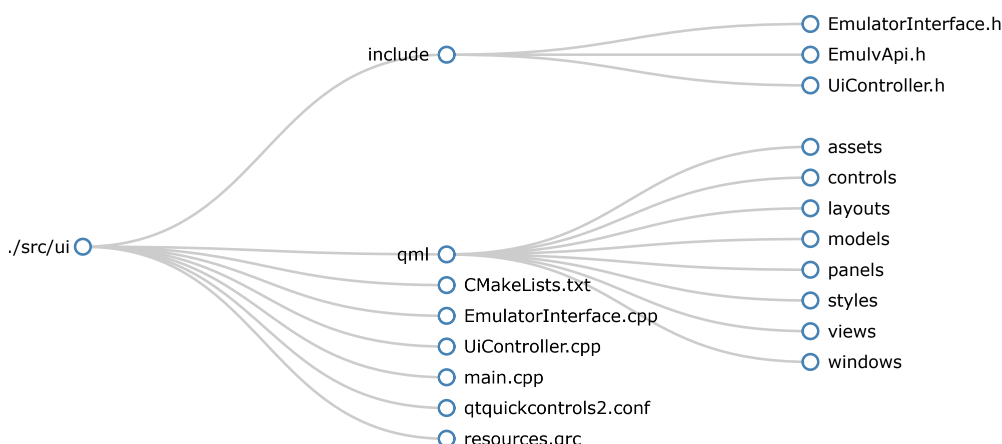
6.5.1 Konfigurační soubory

Ve složce `./src/ui` se nacházejí dva konfigurační soubory, `qtquickcontrols2.conf` a `resources.qrc`.

První z těchto souborů, `qtquickcontrols2.conf`, je konfigurační soubor ovlivňující vzhled prvků z modulu `Qt Quick Controls`, součástí tohoto modulu jsou na-



Obrázek 6.5: Ukázka uživatelského rozhraní programu



Obrázek 6.6: Pohled do složky `./src/ui` (zobrazeno do hloubky 2)

příklad tlačítka, textová pole a další komponenty používané napříč celým grafickým rozhraním. Mezi parametry nastavované tímto souborem patří styl aplikace, nastavený na *Fusion*, tento styl je dokumentací knihovny Qt doporučován pro desktopové aplikace, jako další je zde nastavena velikost fontu pomocí parametru *Font\PixelSize*.

Druhý soubor, `resources.qrc` obsahuje seznam všech „prostředků“ využívaných v této aplikaci, jedná se výhradně o ikony a obrázky ze složky `./src/ui/qml/assets`. Tento soubor je nutný k tomu, aby knihovna Qt všechny tyto tzv. prostředky našla a zpřístupnila uvnitř ať už QML nebo C++ kódu.[5]

6.5.2 QML soubory

Ve složce `./src/ui/qml` jsou soubory dekomponovány do podsložek dle jejich významu v rámci rozhraní, obsah složek je definován následovně:

- assets - všechny ikony a obrázky
- controls - samostatné ovládací prvky (tlačítka, apod.)
- layouts - prvky zajištující „pozicování“ jiných prvků
- models - datové modely
- panels - panel je skupina logicky souvisejících prvků
- styles - všechny soubory týkající se stylu rozhraní

[5]Stejného efektu by šlo dosáhnout i pomocí vyjmenování prostředků v `CMakeLists.txt` souboru a použitím příkazu `qt_add_resources()`, stejně jako tomu je v externích knihovnách probíraných v předešlé sekci. Samostatný soubor je zde ale kvůli velkému počtu definovaných prostředků lepší volbou.

- views - tzv. „pohledy“, tedy konkrétní okna, které aplikace může zobrazit
- windows - abstraktní okna, ze kterých můžeme vyjít při tvorbě „pohledů“

Nejdůležitějším QML prvkem je soubor `./src/ui/qml/views/Main.qml`, uvnitř kterého, je definována struktura a chování hlavního okna aplikace. Okno používá tzv. „side panely“, které je možné za běhu vytvořit pomocí funkce `addSideBarItem`. Tyto panely se zobrazují buď v levé, pravé, nebo dolní části okna, patří mezi ně například *File Explorer*, *Last Opened*, *Memory*, *Registers* (viz obr. 6.5) nebo panely periferních zařízení, které jsou vytvářeny za běhu aplikace, tento systém je tak pro jejich funkčnost nutný. Mezi užitečné funkce „side panelů“ patří například možnost je zobrazit uvnitř samostatného „plovoucího“ okna.

Soubor `Main.qml` je složen pomocí prvků ze složek `controls`, `layouts` a `panels`, tím byla dosáhnuta relativně dobrá přehlednost a udržitelnost tohoto kódu. Grafická podoba tohoto okna je definována v souboru `/windows/BorderlessWindow.qml`, ze kterého `Main.qml` vychází.

V neposlední řadě by bylo dobré zmínit i soubor `/styles/Colors.qml`, který definuje barvy použité pro tmavý a světly motiv této aplikace. Obecně použitelné barvy jsou zde definované v polích `themes.dark` a `temes.light`, barvy, které jsou automaticky aplikované na všechny prvky modulu `Qt Quick Controls` jsou definované v objektech `darkPalette` a `lightPalette`.

6.5.3 Modely a dynamické komponenty

Napříč celým rozhraním byla snaha použít tzv. „dynamické komponenty“, které ze svého modelu načítají jen ta data, která je aktuálně nutné zobrazit, tento krok podstatně zlepšuje rychlost a odezvu rozhraní.

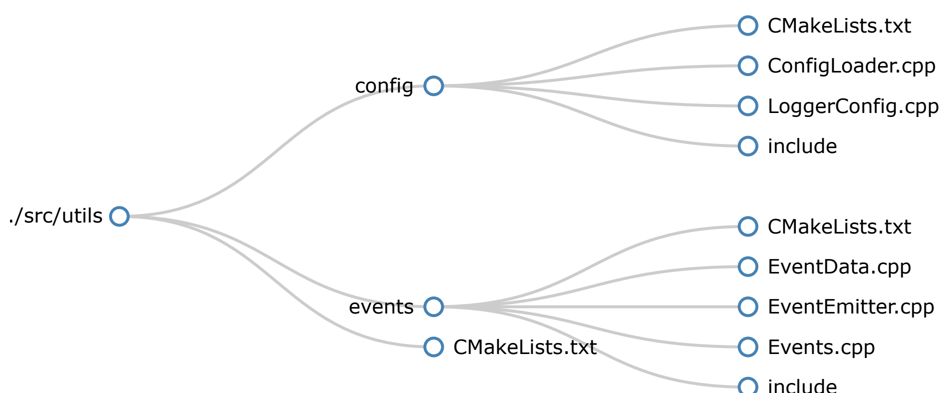
Komponenty zobrazující paměť, registry a textové pole uprostřed okna zobrazující *disassembly* programu jsou implementovány jako tabulka, tedy `TableView` na straně QML, a modelem odvozeným od třídy `QAbstractTableModel` na straně C++.

Komponent zobrazující souborový systém je implementován jako strom, tzn. `TreeView` v QML a na straně C++ speciálním modelem `QFileSystemModel` který je poskytován knihovnou Qt pro účely zobrazování souborového systému.

Komponenty si svůj model vyžádají od třídy `UiControler`, která pro tento účel obsahuje příslušné *get* metody.

6.6 Ostatní

Na konec této kapitoly krátce představíme zbývající dvě části aplikace, které byly již několikrát zmiňovány v předchozím textu. Konkrétně se tedy jedná o `Logger` a třídy spojené s `EventEmitterem`.



Obrázek 6.7: Pohled do složky `./src/utils` (zobrazeno do hloubky 2)

6.6.1 Logger

Knihovnu `spdlog`, kterou používáme v této aplikaci pro ladící výpisy, je při spuštění programu třeba nakonfigurovat, to zajišťuje soubor `./src/utils/config/LoggerConfig.cpp`.

Součástí tohoto souboru jsou dvě funkce, `canWriteLog` a `setupLogger`. Funkce `setupLogger` je po spuštění programu volána uvnitř funkce `main`, tato funkce nastavuje úroveň výpisů (`debug`, `trace`, ...) a výstupní kanály.

Aktuálně je vypisování výstupu nastaveno zároveň do konzole a textového souboru. Funkce `canWriteLog` je zde použita pro ověření, zda máme oprávnění zapisovat logovací soubor na disk, pokud jsou práva nedostatečná, tak je výpis nastaven pouze do konzole.

6.6.2 Events

Jmenný prostor `EventsLib` obsahuje celkem dvě třídy, `EventData` a `EventEmitter`.

Jako další se zde nachází soubor `Events.cpp`, který sdružuje statické funkce spojené s použitím `EventEmitteru`. Buď je možné si zaregistrovat vlastní instanci `EventEmitteru` pomocí funkce `registerEmitter` a později si ho vyzvednou prostřednictvím `getEmitter`, nebo využít funkcí `globalEmit` a `globalOn`, které jsou prováděny nad statickým „globálním“ `EventEmitterem`.

Třída `EventData` je základní datovou jednotkou všech událostí. Skládá se ze slovníku, který je typovaný na `(std::string key, std::any value)`, díky využití `std::any` tedy může obsahovat hodnotu jakéhokoliv datového typu. Tím byl vyřešen problém s definicí nové třídy pro každý druh události.

Testování a výsledky 7

7.1 Jednotkové testy

Součástí tohoto projektu jsou celkem tři různé sady jednotkových testů (viz obr. 7.1). Všechny jednotkové testy jsou implementovány pomocí knihovny **Google Test**, tyto testy je možné spustit prostřednictvím nástroje *CTest*, který je součástí systému *CMake*. Detailní popis příkazů nutných pro spuštění testů viz výpis 7.1.

Výpis 7.1: Příkazy pro spuštění jednotkových testů. První příkaz inicializuje CMake dle „presetu" `Debug-Windows`. Druhý příkaz zkompiluje všechny soubory projektu. Třetí příkaz spustí všechny jednotkové testy v „Debug" režimu.

```
1 C:\Users\pc\emulator>cmake --preset Debug-Windows
2 C:\Users\pc\emulator>cmake --build --preset Debug-Windows
3 C:\Users\pc\emulator>ctest -C Debug --test-dir
    ./cmake/build-debug-windows/test
```

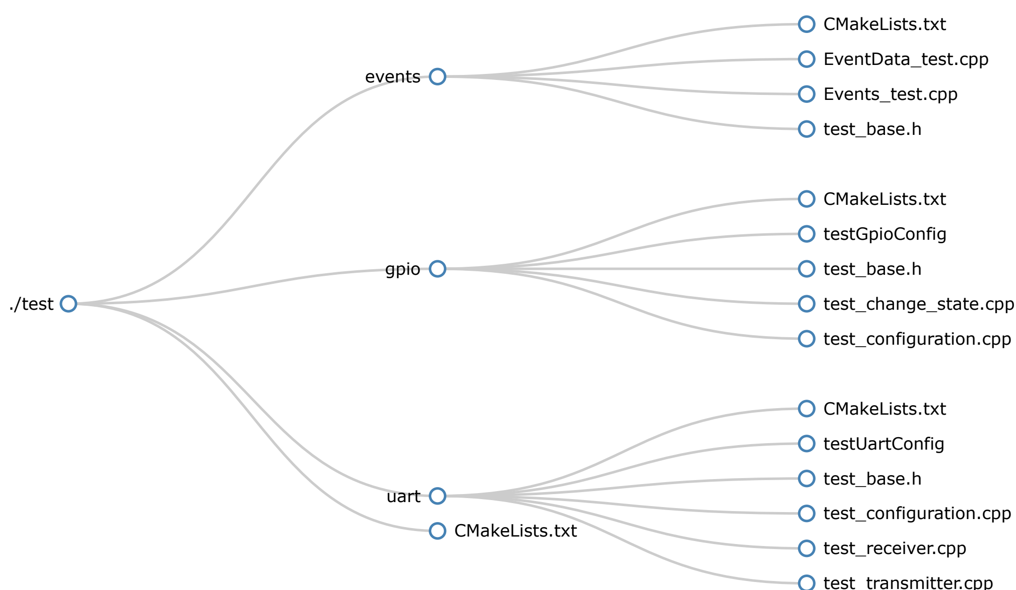
Testy ze složky `events` lze spustit okamžitě po kompilaci projektu, situace je ale komplikovanější s testy `gpio` a `uart`.

Testovací sady `gpio` a `uart` nutně potřebují ve svém adresáři[1] mít k dispozici dynamické knihovny `libgpio.dll` a `libuart.dll`. Obě knihovny jsou zároveň závislé na knihovně `Qt6Core.dll`[2], a tak je nutné, aby se i tato knihovna nacházela ve stejné složce.

Pomocí CMake skriptů bylo zajištěno automatické kopírování souborů `libuart.dll`, `libgpio.dll` a `Qt6Core.dll` do příslušných složek při kompilaci programu. Může se ale stát, že skript tyto soubory nenajde, v tom případě je nutné je zkopírovat manuálně.

[1] Pro gpio `./cmake/build-debug-windows/test/gpio`, pro uart `./cmake/build-debug-windows/test/gpio`

[2] Na operačním systému MS Windows jsou ve skutečnosti knihovny závislé na `Qt6Core.dll`, `libgcc_s_seh-1.dll`, `KERNEL32.dll`, `msvcrt.dll` a `libstdc++-6.dll`. Dle provedených pokusů jsou ale všechny tyto knihovny kromě `Qt6Core.dll` nalezeny automaticky



Obrázek 7.1: Pohled do složky `./test` (zobrazeno do hloubky 2)

Events

Testy pro soubory ze jmenného prostoru `EventsLib` jsou rozděleny do dvou souborů, `EventData_test.cpp` a `Events_test.cpp`.

Soubor `EventData_test.cpp` obsahuje testy týkající se datové struktury `EventData`, je zde testováno vložení dat, odstranění dat, apod.

V souboru `Events_test.cpp` je testována třída `EventEmitter` a funkce ze soubrou Events.cpp.

Tyto testy mají celkově 100% pokrytí řádek kódu souborů patřící do jmenného prostoru `EventsLib`.

GPIO

Test dynamické knihovny `gpio` se skládá ze souborů `test_configuration.cpp` a `test_change_state.cpp`.

Každý test si načte dynamickou knihovnu `gpio`, jak již bylo vysvětleno na začátku této kapitoly, všechny soubory knihovny musí nutně být dostupné v testovacím adresáři. Samotné testování pak probíhá vůči rozhraní `PeripheralsApi`, pomocí funkcí `ReadWord` a `WriteWord`.

Testy v souboru `test_change_state.cpp` ověřují, zda je možné zapsat a číst data ze všech přístupných interních registrů periferie `gpio`.

V souboru `test_configuration.cpp` je pak otestována možnost změnit logickou úroveň jednotlivých `gpio` pinů.

UART

Stejně jako u knihovny gpio, je test dynamické knihovny uart prováděn nad rozhraním PeripheralsApi.

Testy jsou zde rozděleny celkem do tří souborů: test_configuration.cpp, test_receiver.cpp a test_transmitter.cpp.

Soubor test_configuration.cpp obsahuje testy ověřující funkčnost stavových bitů uvnitř UART registrů, jmenovitě UEN, WL, TEN, REN a STB.

Soubor test_receiver.cpp ověřuje, zda je skrze UART možné přijmout zprávu. Oproti tomu soubor test_transmitter.cpp obsahuje testy ověřující, zda je možné skrze UART odeslat zprávu.

7.2 Funkční testy

Cílem této sekce je ověřit věrohodnost a funkčnost emulátoru na několika testovacích programech. Výstup emulátoru zde bude porovnán s výstupem reálného RISC-V zařízení.

7.2.1 Použité zařízení

Všechny testovací programy uvedené v této kapitole byly spuštěny na vývojové desce *Sipeed Longan Nano*, viz obrázek 7.2.

Jedná se o vývojovou desku založenou na 32 bitovém RISC-V procesoru GD32VF103CBT6, deska také obsahuje 128 Kb paměti Flash a 32 Kb paměti RAM.

Mezi hlavní periferie této desky patří UART, GPIO a jednopalcový LCD displej. Deska je napájena buď prostřednictvím USB-C nebo skrze 3.3V pin, který se nachází vedle RX0 a TX0 pinů pro UART.

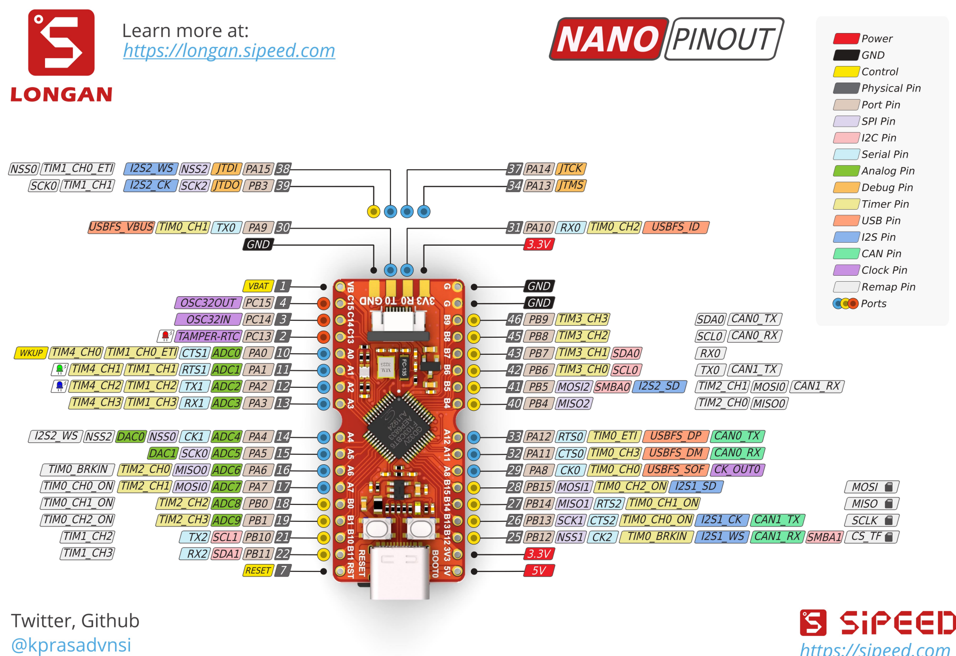
Pro komunikace se zařízením bylo využito CP2101 USB to UART mostu, který lze zároveň použít i pro nahrání firmware.

7.2.2 Testovací programy

Pro tvorbu testovacích programů bylo využito open source repozitáře *GD32VF103_templates* [34], ze kterého byly využity především soubory ze složky ./common, které zajišťují počáteční inicializaci zařízení, nutnou pro spuštění programu.

Výhodou tohoto repozitáře je, že nepoužívá žádné knihovny nebo frameworky, a tak máme absolutní kontrolu nad tím, jakým způsobem budou periferie použity.

Seznam testovacích programů je uveden v tabulce 7.1. Zdrojové soubory těchto programů jsou k dispozici jako příloha této bakalářské práce.



Obrázek 7.2: Diagram vývojové desky *Sipeed Longan Nano*, zdroj: [33]

Název	Periferie	Popis
led_hello	GPIO	Blikání LED
led_toggle	GPIO + UART	Ovládání GPIO pomocí UART vstupu
uart_hello	UART	Výpis hello zprávy
uart_echo	UART	Kopírování vstupních znaků na výstup
uart_fibonacci	UART	Výpočet Fibonacciho čísla

Tabulka 7.1: Seznam testovacích programů

7.2.3 Kompilace

Pro kompilaci testovacích programů je třeba nejdříve zkompilovat a nainstalovat kompilátor `GCC` pro platformu `RV32GC`, jehož zdrojový kód je dostupný z repozitáře *riscv-gnu-toolchain* [35]. Repozitář obsahuje návod popisující, jak kompilátor přeložit na systému GNU/Linux.

Před překladem je důležité spustit konfigurační skript s parametry `--with-arch=rv32gc` a `--with-abi=ilp32`. Bez těchto parametrů bude kompilátor nekompatibilní se zařízením *Sipeed Longan Nano*, a programy tak nepůjde spustit.

Ve chvíli, kdy máme funkční kompilátor, můžeme ve složce libovolného testovacího programu použít příkaz `make`, který program přeloží dle souboru `Makefile`.

Všechny `Makefile` soubory našich testovacích programů očekávají dostupnost

příkazů `riscv32-unknown-elf-gcc`, `riscv32-unknown-elf-objcopy` a `riscv32-unknown-elf-size`. To zajistíme přidáním adresáře obsahující nainstalovaný *riscv-gnu-toolchain* do proměnné `PATH`.

Výstupem kompilace za pomoci `Makefile` souboru jsou v našem případě vždy soubory **main.elf** a **main.bin**. `ELF` soubor můžeme spustit v emulátoru[3], zatímco `bin` soubor je určen pro fyzické zařízení.

Zkompilované verze testovacích programů jsou zároveň k dispozici ve složce emulátoru `./extern/elf_files`.

7.2.4 Spuštění

Programy byly na zařízení nahrány pomocí python nástroje *STM32Loader* [36], pro sériovou komunikaci se zařízení pak bylo využito volně dostupného programu *Termite* [37].

led_hello

Tento program opakovaně mění stav tří `GPIO` pinů, které jsou na zařízení *Sipeed Longan Nano* připojeny k RGB diodě. Způsobuje tak barevné blikání diody.

Program využívá aktivního čekání, které je implementováno ve funkci `delay_cycles`. Na reálném zařízení by bylo možné místo aktivního čekání použít hardwarový časovač, tento časovač však není implementován emulátorem[4], pro účely testování je tak aktivní čekání vhodnou volbou.

V emulátoru lze pozorovat změnu stavu těchto tří `GPIO pinů` (viz obrázek 7.3). Ukázka na fyzickém zařízení viz obrázek 7.4.

led_toggle

Program `led_toggle` cyklicky čeká na vstup od uživatele, na základě kterého pak aktivuje nebo deaktivuje jednu z barev RGB diody.

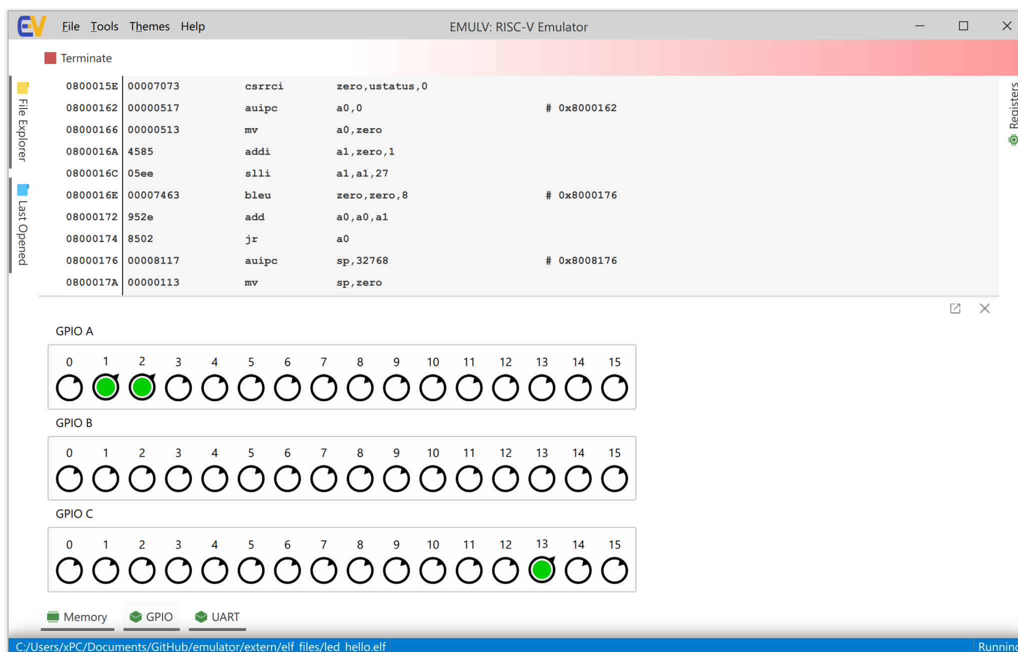
Výstup tohoto programu můžeme vidět ve výpisu 7.2. Při spuštění programu je nejprve vypsána zpráva s instrukcemi, poté je uživatel vyzván k zadání vstupu.

Důležité je, že program vstupní příkaz provede až v té chvíli, když obdrží symbol konce řádky.

V emulátoru lze výpis pozorovat v okně `UART`, změna stavu se pak projeví v okně `GPIO` (viz obrázek 7.3).

[3]Teoreticky lze pro účely fyzického zařízení použít i `ELF` soubor, pokud tuto možnost podporuje program, který používáme pro nahrání firmware na zařízení. `Bin` soubor je pouze prostým obrazem paměti, typicky je tak pro tento účel využíván místo `ELF` souboru.

[4]Bylo by ho možné implementovat jako dynamickou knihovnu, podobně jako `GPIO` nebo `UART`.



Obrázek 7.3: Ukázka aktivních `GPIO` pinů v rozhraní emulátoru

Obrázek 7.4: *Sipeed Longan Nano* se zeleně svítící RGB diodou v dolní části zařízení

Výpis 7.2: Ukázka programu `led_toggle`, uživatel zadává čísla 1, 2 nebo 3. Program vypisuje, jaká barva byla aktivována. V případě, že uživatel zadá neplatný vstup, je vypsána chybová hláška „Wrong input.“, jak můžeme v tomto výpisu vidět na řádce 9.

```
LED TOGGLE
Input one of the following numbers:
1 = Toggle RED led
2 = Toggle GREEN led
3 = Toggle BLUE led

Toggling GREEN...
Toggling RED...
Wrong input.
Toggling RED...
Toggling BLUE...
```

uart_hello

Program `uart_hello` obsahuje minimální funkcionalitu nutnou pro vypsání zprávy „Hello UART“.

Adresy registrů paměťově mapovaných periferních zařízení jsou definovány v horní části souboru pomocí direktiv preprocesoru `#define`. Tyto registry jsou pak nastaveny na výchozí hodnoty pomocí funkcí `prepare_device` a `initialize_uart`.

Následně je možné použít funkci `send_uart`, která odešle parametrem předaný řetězec.

uart_echo

Tento program rozšiřuje `hello_uart` o funkci `uart_receive`, která přijímá znaky ze vstupu. Činnost tohoto programu spočívá v cyklickém čekání na vstup uživatele, který je obratem vypsán na výstup.

uart_fibonacci

Program `uart_fibonacci` je ze všech programů testujících `uart` tím nejrozsáhlejším. Podobně jako program `toggle_led` vždy čeká na vstupní řetězec zakončený symbolem konce řádky, pro tuto činnost zde byla vytvořena funkce `uart_getline`.

Následně je řetězec převeden na číslo pomocí funkce `strtoi`, a je proveden výpočet Fibonacciho čísla pomocí funkce `fib`.

Výsledek je pak vypsán ve formátu `F(<vstup>) = <výstup>`, pokud byl vstupní řetězec neplatný, tak je `<vstup>` i `<výstup>` roven nule.

Ukázkový výstup tohoto programu můžeme vidět ve výpisu 7.3.

Výpis 7.3: Ukázka programu `uart_fibonacci`, uživatel zadává číslo, v případě neplatného vstupu je vypsána nula

```
FIBONACCI NUMBER GENERATOR
Please enter a number...
F(12) = 144
F(33) = 3524578
F(6) = 8
F(20) = 6765
F(0) = 0
```

7.3 Dosažené výsledky

U všech testovaných programů bylo dosaženo stejného chování jak v emulátoru, tak na reálném zařízení.

Při testování byly odhaleny chyby knihovny *libriscv* a programů v repozitáři *GD32VF103_templates*.

Knihovna *libriscv* nebyla schopna spustit program obsahující instrukci `CSRRCI`, při provádění této instrukce program vyhazoval výjimku. Později se ukázalo, že chyba byla přímo v tom, jak knihovna tuto instrukci zpracovávala. Autor knihovny byl na tuto chybu upozorněn a dne 17.4.2024 [38] ji opravil.

Při použití aktuálně nejnovější verze kompilátoru `GCC` pro platformu `RISC-V` nastávala pro některé programy v repozitáři *GD32VF103_templates* chyba linkeru: `relocation truncated to fit: R_RISCV_JAL against symbol 'reset_handler'` [39]. Tento problém byl vyřešen odstraněním řádky `LFLAGS += -Wl,--gc-sections` ze všech `Makefile` souborů.

Závěr 8

Cílem této práce bylo vytvořit modulární emulátor platformy RISC-V, který by zároveň byl vhodný i pro výukové účely.

Modularita byla zajištěna prostřednictvím dynamicky linkovaných knihoven, pomocí kterých je možné k emulátoru připojit libovolné paměťově mapované periferní zařízení, které implementuje stanovené rozhraní.

Grafické rozhraní emulátoru bylo navrženo tak, aby bylo uživatelsky přívětivé a zároveň zobrazovalo všechna podstatná data emulovaného systému. Uživatel tak může v reálném čase sledovat činnost emulovaného procesoru a periferií, čímž bylo dosáhnuto požadovaného „výukového“ efektu.

V první povlovně práce byla proveden analýza platformy RISC-V z komerčního a technického hlediska, dále bylo na teoretické úrovni popsáno téma emulace. V neposlední řade byly prozkoumány a zhodnoceny některé ze současných knihoven pro emulaci instrukční sady RISC-V, společně s knihovnami pro tvorbu grafického rozhraní.

Druhá polovina této práce se zabývala analýzou výchozí implementace tohoto programu, poté následoval popis výsledného software a testování.

Všechny cíle práce byly tedy do uspokojivé míry splněny. Za hlavní úspěch práce by se dala považovat skutečnost, že emulátor dokáže věrohodně emulovat periferie GPIO a UART vývojové desky *Sipeed Longan Nano*. Při testování nebyly zaznamenány žádné situace, kdy by se emulátor choval jinak, než reálné zařízení.

V průběhu práce byla odhalena některá úskalí spojená s architekturou RISC-V, jako například větší míra výskytu chyb v běžně používaných knihovnách pro tuto platformu, nebo nižší dostupnost vývojových nástrojů, které si často uživatel musí sám přeložit.

Největší potenciál pro budoucí rozšíření této práce je v oblasti periferních zařízení, současný stav tohoto software umožňuje relativně jednoduchou implementaci mnoha dalších zařízení jako například displejů, časovačů, apod.

Uživatelská příručka A

Kompilace programu

Pro úspěšnou kompilaci programu je potřeba mít nainstalovanou knihovnu **Qt6.6** (nebo novější verzi). Po nainstalování Qt je třeba upravit proměnné uvnitř souboru `CMakePresets.json`, který se nachází v kořenovém adresáři projektu.

Důležité je uvnitř souboru `CMakePresets.json` specifikovat cestu ke knihovně Qt v proměnné `CMAKE_PREFIX_PATH`, poté proměnné specifikující cestu ke kompilátorům: `CMAKE_C_COMPILER` a `CMAKE_CXX_COMPILER`.

Samotný překlad je pak možné provést pomocí příkazů uvedených ve výpisu A.1.

Výpis A.1: Příkazy pro překlad programu. Poslední příkaz `windeployqt` do adresáře zkopíruje všechny dynamické knihovny, na kterých je aplikace závislá

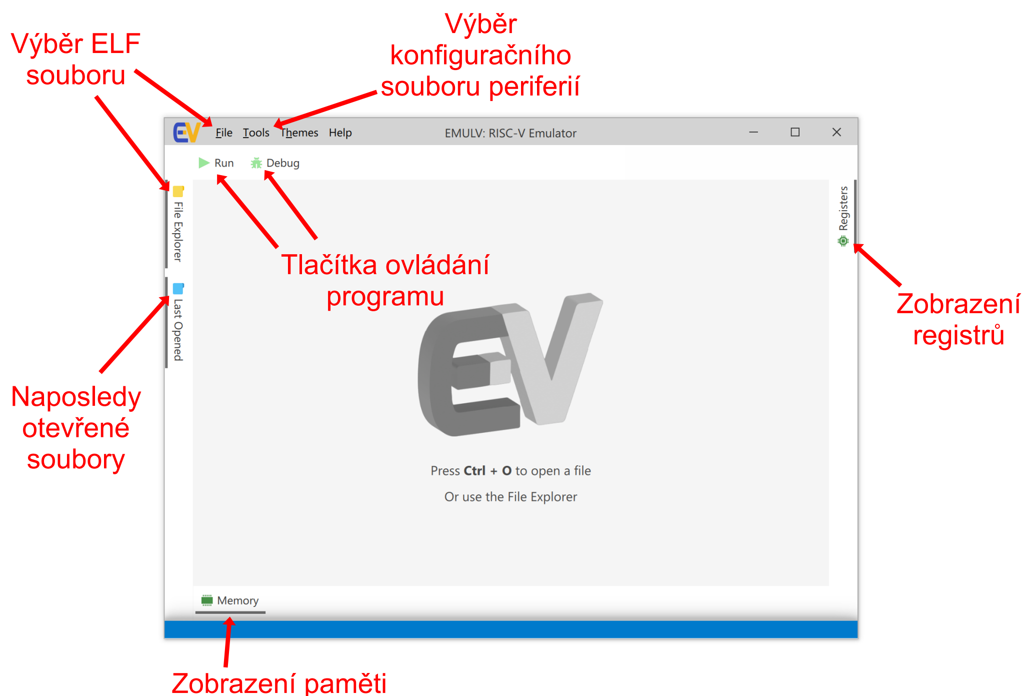
```
1 C:\Users\pc\emulator>cmake --preset Debug-Windows
2 C:\Users\pc\emulator>cmake --build --preset Debug-Windows
3 C:\Users\pc\emulator>windeployqt
    "./cmake/build-debug-windows/bin/ui/Emulv.exe" --qmldir
    "./src/ui/qml/"
```

Vytvoření NSIS instalátoru

Při kompilaci dle CMake *presetu* `Release-Windows` se automaticky vytvoří všechny soubory nutné pro sestavení NSIS intalátoru.

Instalátor sestavíme provedením příkazu `cpack` v adresáři `./cmake/build-release-windows`.

Sestavený NSIS instalátor je k dispozici jako příloha této bakalářské práce.



Obrázek A.1: Popis ovládacích prvků emulátoru

Generování Doxygen dokumentace

Po inicializaci CMake projektu je možné programovou dokumentaci vygenerovat pomocí príkazu `cmake --build --preset Debug-Windows --target doxygen`. Dokumentace pak bude dostupná ve složce `./cmake/build-debug-windows/docs`.

Spuštění programu uvnitř emulátoru

Nejprve je třeba načíst `ELF` soubor programu. To je možné buď prostřednictvím tlačítka **File** a **Open ELF...** nebo pomocí tlačítka **File Explorer** v levém horním okraji okna. Po načtení je možné program spustit buď ve standardním nebo ladícím režimu. Detailní popis tlačítek je znázorněn na obrázku A.1.

Připojení periferií

Knihovny periferií je doporučeno umístit do stejné složky, jako spustitelný soubor emulátoru. Tím je zaručeno, že budou mít k dispozici všechny potřebné dynamické knihovny.

Jako další je potřeba načíst konfigurační soubor pomocí tlačítek **Tools** a **Select Configuration**. Konfigurační soubor, který je součástí přílohy této bakalářské práce, předpokládá knihovny pojmenované `libgpio.dll` a `libuart.dll` umístěné ve stejném adresáři, jako spustitelný soubor `Emulv.exe`.

Bibliografie

1. ASANOVIĆ, Krste; PATTERSON, David A. Instruction sets should be free: The case for risc-v. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146*. 2014.
2. WATERMAN, Andrew; ASANOVI, Krste. *The RISC-V instruction set manual*. [B.r.]. Dostupné také z: `https://riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf`.
3. *OpenHW Group* [online]. OpenHW Group, [b.r.]. [cit. 2024-01-14]. Dostupné z: `https://www.openhwgroup.org/`.
4. *RISC-V foundation* [online]. RISC-V International, [b.r.]. [cit. 2023-12-20]. Dostupné z: `https://riscv.org/members/`.
5. *What is RISC-V, and why we're unlocking its potential* [online]. [B.r.]. [cit. 2023-12-18]. Dostupné z: `https://www.qualcomm.com/news/onq/2023/09/what-is-risc-v-and-why-were-unlocking-its-potential`.
6. BAPTISTA, Eduardo. China bets on open-source chips as US export controls mount. *Reuters* [online]. 2024 [cit. 2024-03-28]. Dostupné z: `https://www.reuters.com/technology/china-bets-open-source-chips-us-export-controls-mount-2024-02-05/`.
7. EADLINE, Doug. *China Deploys Massive RISC-V Server in Commercial Cloud* [online]. 2023. [cit. 2023-12-18]. Dostupné z: `https://www.hpcwire.com/2023/11/08/china-deploys-massive-risc-v-server-in-commercial-cloud/`.
8. WILLIAMS, Wayne. 'World's first': AWS rival launches bare metal servers based on Chinese RISC-V CPU and costs only cents per hour to run [online]. 2024 [cit. 2024-04-03]. Dostupné z: `https://www.techradar.com/pro/worlds-first-aws-rival-launches-bare-metal-servers-based-on-chinese-risc-v-cpu-and-costs-only-cents-per-hour-to-run-but-will-it-live-to-regret-using-emmc-storage`.

9. SHILOV, Anton. Tenstorrent licenses RISC-V CPU IP to build 2nm AI accelerator for Edge. *AnandTech* [online]. 2024 [cit. 2024-03-29]. Dostupné z: `https://www.anandtech.com/show/21281/tenstorrent-licenses-risc-vcpu-ip-to-build-2nm-edge-ai-accelerator`.
10. FWSGONZO. *GitHub - fwsGonzo/rvscript: Fast RISC-V-based scripting backend for game engines* [online]. [B.r.]. [cit. 2024-04-09]. Dostupné z: `https://github.com/fwsGonzo/rvscript`.
11. KOUTE. *GitHub - koute/polkavm: A fast and secure RISC-V based virtual machine* [online]. [B.r.]. [cit. 2024-04-09]. Dostupné z: `https://github.com/koute/polkavm`.
12. *PolkaVM* [online]. 2023. [cit. 2024-04-09]. Dostupné z: `https://forum.polkadot.network/t/announcing-polkavm-a-new-risc-v-based-vm-for-smart-contracts-and-possibly-more/3811`.
13. PYEATT, Larry D.; UGHETTA, William. Chapter 3 - Load/store and branch instructions. In: PYEATT, Larry D.; UGHETTA, William (ed.). *ARM 64-Bit Assembly Language*. Newnes, 2020, s. 53–81. ISBN 978-0-12-819221-4. Dostupné z DOI: `https://doi.org/10.1016/B978-0-12-819221-4.00010-9`.
14. PATTERSON, David A; WATERMAN, Andrew. *The RISC-V reader: an open architecture atlas*. Strawberry Canyon Llc, 2018.
15. LEACHMAN, Robert C.; DING, Shengwei; CHIEN, Chen-Fu. Economic Efficiency Analysis of Wafer Fabrication. *IEEE Transactions on Automation Science and Engineering*. 2007, roč. 4, č. 4, s. 501–512. Dostupné z DOI: `10.1109/TASE.2007.906142`.
16. PUGH, Emerson W. *Building IBM*. 2009. Dostupné z DOI: `10.7551/mitpress/1687.001.0001`.
17. CHATTERTON, Bruce. Software Simulation of the Minuteman D17B Computer. 1972. Dostupné také z: `https://apps.dtic.mil/sti/citations/AD0742965`.
18. MARTIGNONI, Lorenzo; PALEARI, Roberto; ROGLIA, Giampaolo Fresi; BRUSCHI, Danilo. Testing CPU emulators. In: *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*. Chicago, IL, USA: Association for Computing Machinery, 2009, s. 261–272. ISSTA '09. ISBN 9781605583389. Dostupné z DOI: `10.1145/1572272.1572303`.

19. HUNG, William N.N.; SUN, Richard. Challenges in Large FPGA-based Logic Emulation Systems. In: *Proceedings of the 2018 International Symposium on Physical Design*. Monterey, California, USA: Association for Computing Machinery, 2018, s. 26–33. ISPD '18. ISBN 9781450356268. Dostupné z DOI: `10.1145/3177540.3177542`.
20. YANG, Haigang; ZHANG, Jia; SUN, Jingru; YU, Le. Review of advanced FPGA architectures and technologies. *Journal of Electronics (China)/Journal of Electronics. China*. 2014, roč. 31, č. 5, s. 371–393. Dostupné z DOI: `10.1007/s11767-014-4090-x`.
21. *Logisim* [online]. [B.r.]. [cit. 2024-04-12]. Dostupné z: `http://www.cburch.com/logisim/`.
22. *ALINX AX7020 FPGA* [online]. [B.r.]. [cit. 2024-04-12]. Dostupné z: `https://www.xilinx.com/products/boards-and-kits/1-t9ddos.html`.
23. SCOTTI, Giovanni; ZONI, Davide. A Fresh View on the Microarchitectural Design of FPGA-Based RISC CPUs in the IoT Era. *Journal of Low Power Electronics and Applications*. 2019, roč. 9, č. 1. ISSN 2079-9268. Dostupné z DOI: `10.3390/jlpea9010009`.
24. FAYZULLIN, Marat. *How To Write a Computer Emulator* [online]. 2000. [cit. 2024-04-17]. Dostupné z: `http://fms.komkon.org/EMUL8/HOWTO.html`.
25. *238P Operative Systems, University of California, Irvine - The ELF Format* [online]. [B.r.]. [cit. 2024-04-14]. Dostupné z: `https://ics.uci.edu/~aburtsev/238P/hw/hw3-elf/hw3-elf.html#0`.
26. *Google Test* [online]. [B.r.]. [cit. 2024-04-20]. Dostupné z: `https://github.com/google/googletest`.
27. *Doxygen* [online]. [B.r.]. [cit. 2024-04-20]. Dostupné z: `https://www.doxygen.nl/`.
28. *spdlog* [online]. [B.r.]. [cit. 2024-04-20]. Dostupné z: `https://github.com/gabime/spdlog`.
29. *riscv-disassembler* [online]. [B.r.]. [cit. 2024-04-20]. Dostupné z: `https://github.com/michaeljclark/riscv-disassembler`.
30. *JSON for Modern C++* [online]. [B.r.]. [cit. 2024-04-20]. Dostupné z: `https://github.com/nlohmann/json`.
31. *qwindowkit* [online]. [B.r.]. [cit. 2024-04-20]. Dostupné z: `https://github.com/stdware/qwindowkit`.
32. SWIDZINSKI, Rafal. *Modern CMake for C++*. 2022. ISBN 1801070059.

33. SIPEED. *Introduction to Longan Nano* [online]. [B.r.]. [cit. 2024-04-19]. Dostupné z: https://longan.sipeed.com/en/.
34. WRANSOHOFF. *GitHub - WRansohoff/GD32VF103_templates* [online]. [B.r.]. [cit. 2024-04-19]. Dostupné z: https://github.com/WRansohoff/GD32VF103_templates.
35. RISCV-COLLAB. *GitHub - riscv-collab/riscv-gnu-toolchain: GNU toolchain for RISC-V, including GCC* [online]. [B.r.]. [cit. 2024-04-19]. Dostupné z: https://github.com/riscv-collab/riscv-gnu-toolchain.
36. *SMT32loader* [online]. 2023. [cit. 2024-04-20]. Dostupné z: https://pypi.org/project/stm32loader/.
37. RIEMERSMA, Thiadmer. *Termite: a simple RS232 terminal* [online]. [B.r.]. [cit. 2024-04-20]. Dostupné z: https://www.compuphase.com/software_termite.htm.
38. FWSGONZO. *Add missing on_unhandled_csr for CSRRCI (fwsGonzo/libriscv@06f2baf)* [online]. [B.r.]. [cit. 2024-04-20]. Dostupné z: https://github.com/fwsGonzo/libriscv/commit/06f2baf14aa98eb1a5ddc48a463af0bf1f6974e0.
39. WRANSOHOFF. *Linker error - Issue #1 (WRansohoff/GD32VF103_templates)* [online]. [B.r.]. [cit. 2024-04-20]. Dostupné z: https://github.com/WRansohoff/GD32VF103_templates/issues/1.

Seznam obrázků

2.1 Ilustrace inkrementálního přístupu x86, zdroj: [14, s. 3] 8

3.1 Ukázka FPGA vývojové desky (ALINX AX7020), zdroj: [22] 11

3.2 Ilustrace RISC-V procesoru typu RV32IMF natrhnutého pro účely FPGA. V procesorové části (CPU) si lze povšimnout pěti-úrovňového pipeliningu, typického pro procesory typu RISC. Pro účely ladění je zde zahrnut blok duGlobal, se kterým lze komunikovat pomocí integrovaného UART rozhraní. Zdroj: [23] 11

3.3 Diagram ukazující vysokoúrovňovou strukturu softwarového emulátoru. Nutným vstupem emulátoru je program, který je před začátkem emulace třeba načíst do paměti (o tento krok se na reálném hardware stará bootloader). 12

3.4 Diagram popisující návaznost funkcí, pomocí kterých je interpretována instrukce. 13

3.5 Struktura ELF souboru, zdroj: [25] 17

5.1 Adresářová struktura výchozí implementace, zobrazeno do hloubky 2 od kořene (tzn. vnořené položky hloubky 3 a více nejsou zobrazeny) . 32

5.2 Architektura výchozí implementace, funkce `main` po spuštění vytvoří instanci třídy `Controller`, `Controller` inicializuje všechny ostatní třídy a zprostředkovává mezi nimi komunikaci. `Controller` si vytvoří instanci třídy `EventEmitter`, referenci na tento objekt předává periferiím, a třídám `EmulatorUnit`, `MainWindow`. `PeripheralDevice` je abstraktní třída, ze které dědí UART a GPIO. 34

5.3 Ukázka uživatelského rozhraní původní implementace 36

6.1 Adresářová struktura projektu (zobrazeno do hloubky 2 od kořene) . . 40

6.2 Architektura výsledné aplikace. Je zde zobrazen samotný emulátor (Emulv) a dvě dynamické knihovny (GPIO a UART). Jádrem emulátoru je třídá UiController, která propojuje jednotlivé části aplikace. Funkce `main()` v tomto případě pouze volá funkci `startQt()`, která inicializuje uživatelské rozhraní 43
6.3 Pohled do složky `./src` (zobrazeno do hloubky 2) 45
6.4 Pohled do složky `./src/plugins` (zobrazeno do hloubky 2) 46
6.5 Ukázka uživatelského rozhraní programu 48
6.6 Pohled do složky `./src/ui` (zobrazeno do hloubky 2) 49
6.7 Pohled do složky `./src/utils` (zobrazeno do hloubky 2) 51

7.1 Pohled do složky `./test` (zobrazeno do hloubky 2) 54
7.2 Diagram vývojové desky *Sipeed Longan Nano*, zdroj: [33] 56
7.3 Ukázka aktivních `GPIO` pinů v rozhraní emulátoru 58
7.4 *Sipeed Longan Nano* se zeleně svítící RGB diodou v dolní části zařízení 58

A.1 Popis ovládacích prvků emulátoru 64

Seznam tabulek

2.1 Základní rozšiřující instrukční sady RISC-V, typicky uváděné ve formátu RV[32/64][písmena rozšíření] 8

3.1 Porovnání knihoven pro emulaci instrukční sady. 22

7.1 Seznam testovacích programů 56

Seznam výpisů

5.1 Ukázka nadměrného počtu ukazatelů ve třídě `MainWindow` 36
6.1 Definice CMake *presetu* „Debug-Windows" 42
6.2 Ukázka definování maker preprocesoru na základě typu operačního systému. Tyto makra jsou používána třídou `LibLoader`. Pro stručnost byla vynechána část pro Linux, která definuje stejná makra, jen s linuxovou verzí příslušných systémových volání. 46
7.1 Příkazy pro spuštění jednotkových testů. První příkaz inicializuje CMake dle „presetu" `Debug-Windows`. Druhý příkaz zkompiluje všechny soubory projektu. Třetí příkaz spustí všechny jednotkové testy v „Debug" režimu. 53
7.2 Ukázka programu `led_toggle`, uživatel zadává čísla 1, 2 nebo 3. Program vypisuje, jaká barva byla aktivována. V případě, že uživatel zadá neplatný vstup, je vypsána chybová hláška „Wrong input.", jak můžeme v tomto výpisu vidět na řádce 9. 59
7.3 Ukázka programu `uart_fibonacci`, uživatel zadává číslo, v případě neplatného vstupu je vypsána nula 60
A.1 Příkazy pro překlad programu. Poslední příkaz `windeployqt` do adresáře zkopíruje všechny dynamické knihovny, na kterých je aplikace závislá 63