VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

EMULÁTOR HERNÍ KONZOLE PLAYSTATION 1 S VYŠŠÍM RENDEROVACÍM ROZLIŠENÍM

PLAYSTATION 1 EMULATOR WITH HIGHER RENDERING RESOLUTION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. FILIP STUPKA**
AUTHOR

VEDOUCÍ PRÁCE — **doc. Ing. JIŘÍ JAROŠ, Ph.D.**
SUPERVISOR

BRNO 2024

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání diplomové práce

157543

Ústav:	Ústav počítačových systémů (UPSY)
Student:	**Stupka Filip, Bc.**
Program:	Informační technologie a umělá inteligence
Specializace:	Počítačová grafika a interakce
Název:	**Emulátor herní konzole Playstation 1 s vyšším renderovacím rozlišením**
Kategorie:	Počítačová architektura
Akademický rok:	2023/24

Zadání:

1. Seznamte se s architekturou herní konzole Playstation 1. Zaměřte se na jednotlivé hardwarové komponenty a jejich propojení.
2. Prostudujte technologie pro emulaci herních konzolí na PC.
3. Navrhněte strukturu softwarového emulátoru této konzole.
4. Navržené řešení implementujte.
5. Otestujte implementované řešení na několika testovacích úlohách.
6. Navržené řešení zdokumentujte a diskutujte jeho přínos pro herní komunitu.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce:	**Jaroš Jiří, doc. Ing., Ph.D.**
Vedoucí ústavu:	Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání:	1.11.2023
Termín pro odevzdání:	17.5.2024
Datum schválení:	30.10.2023

Abstrakt

Tato práce se zabývá zvýšením vizuální věrnosti v emulaci systému PlayStation 1 zavedením pokročilých vykreslovacích technik pro výstup ve vyšším rozlišení. Studie začíná důkladnou analýzou původní hardwarové architektury a identifikací komponent ovlivňujících grafický výstup. Výzkum se zaměřuje na vývoj vlastního emulátoru a zabývá se problémy spojenými se zvýšením rozlišení grafiky při zachování kompatibility se stávajícím herním softwarem. Výsledky poskytují cenné poznatky pro nadšence retro her a výzkumné pracovníky a ukazují úspěšnou implementaci emulátoru PlayStation 1 s vyšším rozlišením vykreslování pro zachování odkazu klasických her.

Abstract

This thesis investigates the augmentation of visual fidelity in PlayStation 1 emulation by implementing advanced rendering techniques for higher resolution output. The study begins with a thorough analysis of the original hardware architecture, identifying components impacting graphical output. Focusing on developing a custom emulator, the research addresses challenges associated with upscaling graphics while maintaining compatibility with existing game software. The findings provide valuable insights for retro gaming enthusiasts and researchers, showcasing the successful implementation of a PlayStation 1 emulator with higher rendering resolution to preserve the legacy of classic games.

Klíčová slova

emulátor, PlayStation, Sony, simulátor, grafika, rozlišení

Keywords

emulator, PlayStation, Sony, simulator, graphics, resolution

Citace

STUPKA, Filip. *Emulátor herní konzole Playstation 1 s vyšším renderovacím rozlišením.* Brno, 2024. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce doc. Ing. Jiří Jaroš, Ph.D.

Emulátor herní konzole Playstation 1 s vyšším renderovacím rozlišením

Prohlášení

Prohlašuji, že jsem tuto semestrální práci vypracoval samostatně pod vedením pana doc. Ing. Jiřím Jarošem, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Filip Stupka
13. května 2024

Poděkování

Děkuji svému vedoucímu doc. Ing. Jiřímu Jarošovi, Ph.D. za vedení této práce a věcné připomínky. Dále děkuji skupině nadšenců pro emulaci *PlayStation* konzole za nemalou pomoc v beznadějných situacích. Hlavní dík pak patří Martinu "nocash" Korthovi, bez jeho cenné práce by tento projekt nebyl možný.

Obsah

Seznam obrázků

Kapitola 1

Úvod

Videoherní průmysl, ačkoliv je stále relativně mladý, představuje jednu z nejvýdělečnějších a nejdůležitějších forem médií v moderní době. Nejenže se stále více integruje do kultury, ale již v roce 2018 byl tento průmysl schopen vygenerovat 134,9 miliardy dolarů [5] a toto číslo každým rokem stále roste.

S tímto růstem a úspěchem však přichází určitá forma amnézie. Většina firem, které se zabývají tvorbou a publikováním videoher, se soustředí pouze na moderní systémy. Ty hry, které byly v minulosti odpovědné za vývoj a růst tohoto průmyslu, rychle mizí z dějin lidské kultury, podobně jako pára nad hrncem.

1.1 Dostupnost her

V roce 2023 byla provedena studie [8] společností *Video Game History Foundation*, zaměřující se na dostupnost starých videoher na moderních systémech a to, zda-li je vůbec možné zakoupit staré videohry oficiálně a legální cestou.

Výsledek tohoto výzkumu byl skličující. Z 4000 vytipovaných videoher vydaných před rokem 2010 je **87%** nedostupných [8] a není možné je získat oficiální legální cestou (studie se zabývala pouze obchody v USA). Článek tvrdí, že existuje několik důvodů proč je historie videoherního průmyslu v takto zanedbaném stavu.

Jedním z důvodů jsou technické problémy, kde portování videohry z jednoho systému na jiný může být netriviální záležitost a vyžaduje alokaci finančních a lidských zdrojů s nejistým výdělkem. Dalším a důležitějším faktorem nedostupnosti videoher je problém licenčních práv, kde samotný zákon ztěžuje dostupnost a samozřejmě také distribuční prostředky mohou v tomto problému hrát roli.

I přesto, jak je tento problém široce pochopen, není mu v širších kruzích přisuzována téměř žádná váha.

1.2 Emulátory

Většina způsobů jak získat možnost zahrát si historické videohry bohužel hraničí se zákonem. Člověk je nucen nelegálně stahovat software z internetových stránek, které jsou dedikované na nelegální sdílení licencovaného obsahu.

Ovšem to je možné pouze tehdy, pokud daná videohra byla vytvořena pro hardwarový systém, který daný člověk již vlastní. Jestliže byla videohra publikována pouze na platformě, která se již 20 let neprodává a není již nijak oficiálně podporována, má člověk smůlu. To

platí i tehdy, když si člověk oficiálně hru zakoupil, ale nemá dostupný hardware, který by hru dokázal spustit.

Problém nedostupnosti samotných her je obtížně řešitelný, avšak problém nedostupnosti historického hardwaru je na tom podstatně lépe. Ačkoliv herní konzole jsou speciálně navrženy a tedy patentovány, nic veřejnosti nebrání vytvořit softwarový simulátor, který se snaží co nejpřesněji napodobit chování hardwaru herních konzolí a zároveň umožnit určitou videohru si zahrát, jak ukázal soudní spor mezi firmami Sony Computer Entertainment a Connectix Corporation [1]. Implementace emulátoru retro herní konzole **PlayStation 1** je náplní této práce.

Kapitola 2

PlayStation

V roce 1995 se společnost *Sony Computer Entertainment* rozhodla vydat herní konzoli *PlayStation*. Toto rozhodnutí bylo reakcí na rozpadlý vztah s firmou *Nintendo*, se kterou měla *Sony* spolupracovat na vytvoření nové herní konzole, jak můžeme vidět v obrázku 2.2, umožňující využívání jak *CD*, tak i kazet jako úložné médium. Díky ambicím ze strany *Sony* a nepřátelství vůči *Nintendu* se *PlayStation* konzole odlišovala v mnoha ohledech od svých konkurentů, a právě tyto změny vyvolaly významný ohlas a zajistily této konzoli úspěch, viz obrázek 2.1.

V současné době je *PlayStation* šestou nejlépe prodávanou herní konzolí[1], čímž se stala velmi populárním systémem. Konzole obsahovala následující hardware[2]:

- **CPU** - MIPS R3000A 33.8688MHz
- **RAM** - 2MiB EDO DRAM
- **Geometry Transformation Engine (GTE)** - Akcelerátor lineární algebry
- **Motion Decoder (MDEC)** - Dekodér JPEG obrázků
- **GPU** - 32bit Sony GPU, 1MB VRAM
- **SPU** - 16bit Sony SPU
- **CD-ROM**

2.1 NTSC/PAL a bezpečnost

V roce *1995* byla hlavní televizní technologií stále *Cathode Ray Tube (CRT)*, přičemž existovaly dva standardy, které specifikovaly enkódování a zobrazování barev na těchto analogových zařízeních. Tyto standardy byly pojmenovány **NTSC** a **PAL** přičemž to, s jakým standardem se člověk mohl setkat, záviselo na jeho geografické poloze. *PlayStation* konzole samozřejmě musela s těmito rozdíly počítat a rozlišovala celkem tři různé regiony:

[1] Seznam nejlépe prodávaných videoherních konzolí: https://en.wikipedia.org/wiki/List_of_best-selling_game_consoles

[2] Technická specifikace PlayStation konzole: https://en.wikipedia.org/wiki/PlayStation_technical_specifications

Obrázek 2.1: První pokus společnosti *Sony* proniknout na herní trh byl velmi úspěšný. Celkem se prodalo přibližně 102,49 milionu kusů. Pro srovnání, *Nintendo 64* od společnosti *Nintendo* se prodalo pouze 32,93 milionu

- **USA** - NTSC
- **Japonsko** - PAL
- **Evropa** - PAL

Tyto dva standardy definovaly vertikální rozlišení, snímkovou frekvenci a také verzi *BIOSu*, který konzole obsahovala[3]. Aby se *Sony* vyhnulo licenčním problémům a předešlo pirátství, regionálně uzavřelo každou konzoli tak, že každá konzole měla ve svém *BIOSu* regionální podpis. Poté na každém *CD* obsahujícím hru byla vypálena jedna ze tří regionálních značek. Tento speciální segment se nacházel velmi blízko středového otvoru *CD* a konvenční *CD-ROM* čtečky nebyly schopné tento region číst ani do něj cokoliv vypalovat.

Při bootování hry byl tento speciální segment přečten a porovnán s regionálním kódem *BIOSu*, a pokud se tyto podpisy neshodovaly, konzole odmítla hru načíst.

Tento způsob ochrany byl velmi prostý, ale účinný. Speciální vypalovače vlastnila pouze *Sony*, a tedy oficiálně licencované *CD* disky nebylo možné zreplikovat. Nicméně tato forma ochrany nebyla perfektní a relativně brzy byly objeveny způsoby, jak tento systém obejít.

Prvním způsobem bylo vytvoření hardwarového čipu, který člověk nainstaloval do konzole[4]. Tento čip pak figuroval jako prostředník mezi *CD-ROM* a *BIOSem* a kdykoliv byl speciální region kód *BIOSem* vyžádán, čip zaslal falešná data s nimiž byl *BIOS* spokojen. Nicméně jisté hry, jako například *Spyro 2*, dokázaly detekovat prezenci tohoto čipu a znemožnit hráči v postupu ve hře, nebo mu tento postup potají zásadně znepříjemnily. Toto vítězství na straně *Sony* však mělo krátkou životnost, jelikož vývojáři čipu vytvořili tzv.: *Stealth mod-chip*, který se po zaslání falešných informací uměl deaktivovat.

Druhým způsobem obejití ochrany byla technika nazvaná *Disc Swapping*. Tento exploit byl založen na faktu, že *CD-ROM* mechanika používala fyzický spínač, pomocí kterého detekovala, zda je poklop zavřen či otevřen. Uživatel mohl tento spínač, například pomocí kusu plastu nebo složeného papíru, uměle ovládat a donutit tak *CD-ROM* mechaniku si myslet, že poklop je uzavřen. Uzavření poklopu byl signál pro konzoli, aby začala načítat

[3]NTSC/PAL rozdíly [4]: https://psx-spx.consoledev.net/graphicsprocessingunitgpu/#vertical-video-timings

[4]Stealth chip: https://www.r43ds.org/products/PS1-Modchip-Playstation-Stealth-Mod-chip.html

Obrázek 2.2: Nedávno se objevil prototyp konzole, která je důkazem vztahu mezi společnostmi *Sony* a *Nintendo*. Konzole fungovala jako hybrid, do kterého bylo možné nahrávat hry z *CD* nebo kazet. Nakonec tato konzole nebyla nikdy vydána, protože *Sony* chtělo kontrolovat licencování *CD* verzí her a *Nintendo* to tento požadavek zamítlo [2].

hru a provést kontrolu regionálního kódu. Jakmile kontrola byla provedena, uživatel se stále otevřeným poklopem oficiální *CD* rychle vyměnil za neoficiální *CD*, a tím byl exploit hotov. Konzole si pak myslela, že obsahuje legitimní *CD*, což ale nebyla pravda.

Konečné naložení s piráctvím pak *Sony* řešilo na softwarové úrovni tak, že do *sub-kanálů*[5] disku zapisovala speciální klíče, které tehdejší běžné *CD-ROM* mechaniky nedokázaly replikovat. Poté za běhu hra spustila několik kryptografických kontrol, jako například *CRC* kontrolní sumu nad svými daty, přičemž pokud na disku chyběly *sub-kanály* s klíči, hra poté stejně jako v předchozích situacích odmítla hru spustit, nebo například v případě hry *Spyro 3*, hra hráči dovolila pokračovat do určitého bodu a pak mu bez varování smazala uložený postup na paměťové kartě. V tomto případě hry musely být individuálně analyzovány a pro každou hru bylo nutné vytvořit záplaty pro vypnutí těchto kontrol, aby hra byla spustitelná na zkopírovaném disku.

2.2 Existující emulátory

Popularita této konzole je samozřejmě patrná z mnoha hledisek. Nejenže v roce 2020 *Sony* vydalo pátou generaci své herní konzole, ale dodnes existuje dedikovaná komunita nadšenců, kteří tuto konzoli dokázali rozebrat do posledního šroubku [4]. Existují rozsáhlé dokumenty, které popisují tuto konzoli do každého detailu, a díky tomu existuje celá řada emulátorů. To, co také činí tento systém populárním v těchto kruzích je, že na tehdejší dobu tato konzole měla z velké části modularní design. Herní systémy té a předchozí doby měly mnoho komponent, které byly doslova *natvrdo zadrátovány* a neposkytovaly žádnou flexibilitu. I přesto, že *PlayStation* do této kategorie hardwarového designu částečně spadá, celý systém je navržen spíše jako moderní stolní počítač.

[5]CD-ROM subchannels: https://en.wikipedia.org/wiki/Compact_Disc_subcode

V dnešní době emulátory nejen poskytují velmi přesnou emulaci *PlayStation* systému, ale také nabízejí nespočet vylepšení oproti původnímu systému, které zpříjemňují jeho použití. Například emulátor *DuckStation*[6] umožňuje uložit současný stav celé konzole (*Save State*) a vytvořit tak snímek v čase, který člověk může později obnovit. Tato vlastnost se hodí v obtížných videohrách, ve kterých se ukládá postup zřídka, a tedy umožňuje hráči si diktovat uložení postupu dle vlastního uvážení.

Další vymožeností, kterou disponuje emulátor *PCSX ReARMed*[7], je vlastní implementace *High-Level Emulation (HLE) BIOSu*. *BIOS* je nedílnou součástí každé *PlayStation* konzole. Stará se jednak o inicializaci hardwaru, ale také figuruje jako velmi jednoduchý operační systém, který videohry, stavěné na tomto systému, mohou používat pro usnadněný přístup k hardwaru. Tento fakt nutí každého uživatele emulátoru *BIOS* extrahovat z původní konzole nebo jej získat nelegálními prostředky. Emulátor *PCSX ReARMed* se pokusil celý *BIOS* dekompilovat a implementovat ho přímo do vlastního systému.

Většina emulátorů také nabízí možnost *Just-in-Time (JIT)* kompilace *PlayStation* procesoru buď na *x86_64* či *arm* architekturu. Transformace instrukcí *PlayStation* procesoru na nativní procesor za běhu má za následek velké zisky v oblasti výkonu. Emulátor *PCSX ReARMed* jde ještě dál a určité hardwarové komponenty *PlayStation* systému implementuje pomocí ručně psaného *arm assembly* kódu, což umožňuje získat nemálo výkonu zejména u mobilních zařízení, u kterých dominuje *arm* architektura.

Velmi impozantním projektem je také *MiSTer FPGA PSX*[8], což je práce snažící se implementovat *PlayStation* emulátor na *FPGA* čipu.

2.3 Renderovací rozlišení

Obrázek 2.3: *Dolphin* emulátor umí zvýšit interní renderovací rozlišení, což může vylepšit herní zážitek

Nepochybně je ekosystém okolo konzole *PlayStation* velmi vyzrálý. Mít možnost si bez větších problémů užít videohry z 90. let je velmi uspokojivé z hlediska zachování videoherní

[6]Duckstation github repozitář: https://github.com/stenzek/duckstation
[7]PCSX ReARMed github repozitář: https://github.com/notaz/pcsx_rearmed
[8]MiSTer FPGE PSX github repozitář: https://github.com/MiSTer-devel/PSX_MiSTer

historie. Přestože existuje několik vyspělých emulátorů s mnoha vychytávkami, žádný z nich se nesoustředí na vylepšení grafické prezentace, jako například možnost změnit velikost interního renderovacího rozlišení[9], viz obrázek 2.3. Tento nápad není originální a lze ho nalézt v emulátorech různých konzolí jako jsou například:

- **Citra** - emulátor *Nintendo 3DS*[10]
- **Dolphin** - emulátor *Nintendo Gamecube/Wii*[11]
- **PCSX2** - emulátor *Sony PlayStation 2*[12]

Tato schopnost emulátoru je možná díky tomu, že systémy jako *Nintendo Gamecube* či *PlayStation 2* mají podobnou grafickou pipeline jako moderní systémy, a tedy není tak složité modifikovat emulátor tak, aby vykresloval do většího prostoru. *PlayStation* má velmi netradiční zpracování grafiky (např.: chybějící *z-buffer*, žádná perspektivní korekce, ...) a vykreslování ve vyšším rozlišení je tedy složitější, protože nemálo her na těchto netradičních hardwarových funkcích napevno závisí. Tato práce se tedy nezabývá pouze implementací emulátoru, ale také zvyšováním renderovacího rozlišení *GPU* komponenty systému.

[9] Ukázka změny rozlišení: https://www.youtube.com/watch?v=LSIYalc1D6Y
[10] Citra github repozitář: https://web.archive.org/web/20240304191755/https://github.com/citra-emu/citra
[11] Dolphin github repozitář: https://github.com/dolphin-emu/dolphin
[12] PCSX2 github repozitář: https://github.com/PCSX2/pcsx2

Kapitola 3

Architektura

Emulátor jakéhokoliv hardwarového systému je ve své podstatě velmi komplexní, proto je třeba správně rozvrhnout celkovou architekturu tohoto projektu. Za prvé, je nezbytné myslet na udržitelnost kódu a schopnost jej bez větších problémů rozšiřovat. Za druhé, komplexita kódu by měla být rozdělena do jednotlivých tříd. Emulátor také potřebuje přístup k oknu a musí být schopen vykreslovat obsah grafické paměti na obrazovku.

3.1 Nástroje/Techniky

Pro tyto účely a požadavky jsem zvolil *C++* jako jazyk pro implementaci. *Objektivně orientovaný* přístup k designu projektu a *polymorfismus* tohoto jazyka jsou pro tento projekt ideální.

Vzhledem k tomu, že celkový systém se skládá z několika objektů, které na sobě vzájemně závisí a vyžadují referenční ukazatele, je nutno myslet na správné definice jednotlivých objektů a správnou správu paměti. V tomto případě lze využít *Dopředné deklarace* všech komponent, aby si dané komponenty mohly lehce vytvářet reference na komponenty jiné. Pro správu paměti jsem využil *RAII* přístup, zkombinovaný s filozofií *Heap-only initialization*. To znamená, že všechny objekty jsou spravovány přes *smart* ukazatele ze standardní knihovny *C++* (*shared_ptr, unique_ptr, weak_ptr*), přičemž takovýto objekt programátor nemůže zkonstruovat na zásobníku, protože konstruktor tohoto objektu je privátní. Místo toho programátor musí zavolat speciální statickou funkci *create*, která se postará o alokaci a inicializaci objektu na haldě a vrací *smart* ukazatel na tento objekt. Zjednodušenou implementaci této myšlenky můžeme vidět v obrázku 3.1.

3.2 Virtuální hardwarová komponenta

PlayStation ve svém hardwarovém designu připomíná velice obyčejný počítač, kterému byly odstraněny přebytečné komponenty. *Sony*, na rozdíl od svých oponentů, vytvořilo tuto konzoli z relativně dobře dokumentovaných čipů již existujících počítačů. Samozřejmě *PlayStation* obsahuje i patentované, na míru udělané součástky, které se snaží ulehčit práci procesoru.

Tyto hardwarové čipy sdílejí velmi podobné rozhraní. To je dáno tím, že interakce mezi těmito komponentami jsou založeny na *Memory-mapped I/O*. To znamená, že sběrnice systému má uniformní paměťové rozložení, přičemž jednotlivé komponenty se mapují do specifických paměťových intervalů. Jednotlivé komponenty pak mohou do těchto intervalů

```cpp
#include <iostream>
#include <memory>

class RAIIManagedObject
{
public:

    /**
     * api for creating object
     */
    static std::shared_ptr<RAIIManagedObject> create(int foo)
    {
        return std::shared_ptr<RAIIManagedObject>(new RAIIManagedObject(foo));
    }

    /**
     * delete implicit move and copy constructors
     */
    RAIIManagedObject(const RAIIManagedObject&) = delete;
    RAIIManagedObject(const RAIIManagedObject&&) = delete;
    RAIIManagedObject operator=(const RAIIManagedObject&) = delete;
    RAIIManagedObject operator=(const RAIIManagedObject&&) = delete;

private:

    /**
     * only be able to construct the object from within the class
     */
    explicit RAIIManagedObject(int foo):
        m_foo(foo) {}

    int m_foo;
};


int main()
{
    auto object = RAIIManagedObject::create(5);
}
```

Obrázek 3.1: Zjednodušený pohled na *RAII, heap-only initialization* filozofii. *C++* má bohužel spoustu možností jak vytvořit a spravovat objekt, přičemž ne každá z nich je optimální či správná. Omezení vytváření objektů v projektu unifikuje paměťovou správu a odpovědnost jednotlivých objektů.

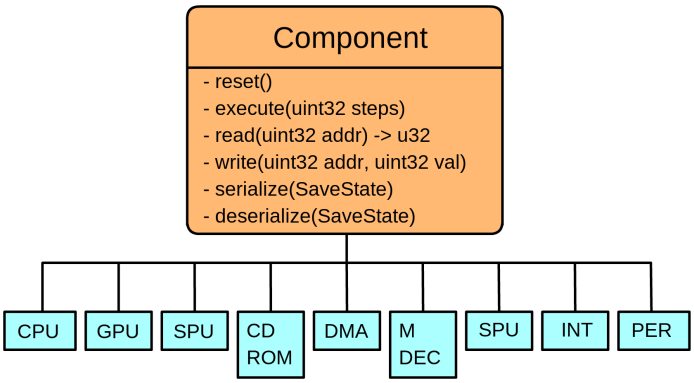
zapisovat nebo z nich číst a sběrnice pak rozdistribuuje tyto přístupy daným komponentám. Díky tomuto faktu, v emulátoru musí mít každá virtuální hardwarová komponenta následující schopnosti:

- *Reset/Inicializace*
- *Čtení* z komponenty
- *Zápis* do komponenty
- Provedení *jednotky práce*
- *Serializace*
- *Deserializace*

Pomocí dědičnosti a virtuálních metod v *C++* můžeme specifikovat virtuální třídu, která bude sloužit jako základ pro všechny hlavní hardwarové komponenty, viz obrázek 3.2.

3.3 Ukládání stavu emulátoru

Aby se emulátor snadněji debugoval, bylo nutné vytvořit systém ukládání a načítání stavu emulátoru na disk. Díky tomu se dají přeskočit sáhodlouhé inicializační rutiny a úvody v



Obrázek 3.2: Rozhraní, které každá virtuální hardwarová komponenta v emulátoru musí respektovat a implementovat.

testovaných hrách. Každá komponenta se tedy musí umět převést na sérii bytů a z těchto samých bytů se znovu zrekonstruovat.

Tuto zodpovědnost přebírá objekt *SaveState*, který dokáže serializovat/deserializovat všechny primitiva jednotlivých komponent. Emulátor při ukládání/načítání svého stavu tento objekt stromovitě předá postupně všem komponentám. Jednotlivé komponenty vloží svoje atomy do *SaveState* objektu při serializaci, nebo je načtou v předem daném pořadí při deserializaci.

3.4 Návrh architektury

PlayStation se skládá z několika hlavních hardwarových komponent. Patří sem [4]:

- Sběrnice
- Central Processing Unit **(CPU)**
- Graphics Processing Unit **(GPU)**
- Sound Processing Unit **(SPU)**
- 3 Časovače/Hodiny
- Ovladač přerušení
- Ovladač přímého přístupu do paměti **(DMA)**
- Dekodér makrobloku **(MDEC)**
- CD-ROM

Každá z těchto komponent je klíčová pro správné fungování emulátoru jako celku. Zjednodušený návrh propojení je popsán v následujícím obrázku B.1, přičemž detailní návrh komponent, schopností a s kým mohou komunikovat je pak popsán v obrázku B.2.

V návrhu jsou také zohledněny podřadné komponenty, které nemohou fungovat samostatně. To se týká například komponenty *Geometry Transformation Engine (GTE)*, což je koprocesor specializující se na práci s lineární algebrou. Jelikož k této komponentě lze

přistoupit pouze skrz *CPU* pomocí speciální instrukce, nelze tuto komponentu chápat jako samostatný celek.

Každá komponenta je následně propojena se sběrnicí kvůli *Memory-mapped I/O*. Existují však i přímá propojení bez sběrnice jako prostředníka. To je hlavně díky *DMA* komponentě, která zajišťuje rychlý přenos dat, aniž by se *CPU* muselo o tento přenos starat. Další přímá propojení jsou určena pro správu přerušení. Vzhledem k tomu, že přerušení může nastat skoro v každé komponentě, je nutné toto přerušení přenášet do *Ovladače výjimek*, který následně upraví stav *CPU* a přerušení se zpracuje jako výjimka.

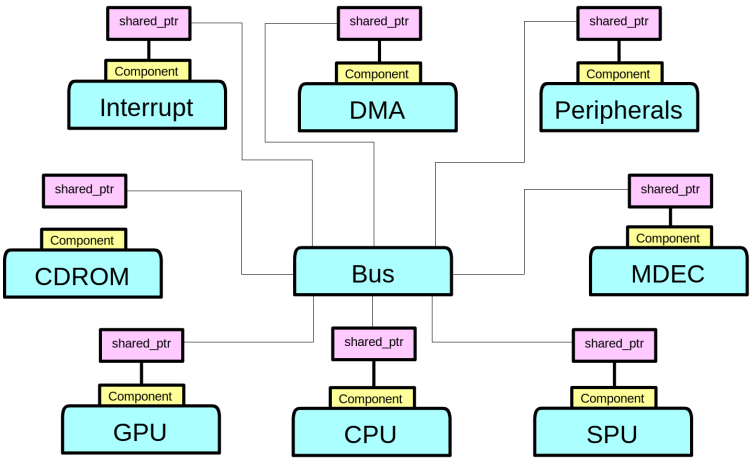
Kapitola 4

Implementace hardwarových komponent

4.1 Sběrnice/Jádro emulátoru

Sběrnice představuje do jisté míry nervový systém celého emulátoru, neboť nejen propojuje hlavní hardwarové komponenty, ale musí se také starat o jejich paměťovou správu. Díky tomuto faktoru sběrnice v emulátoru vytváří všechny komponenty a řeší jejich vzájemné závislosti, čímž vytváří pomyslné jádro emulátoru.

Jelikož komponenty mohou být vzájemně sdíleny, bylo nutné zvolit vhodnou strukturu pro správu jednotlivých objektů. Pro tyto účely jsem se rozhodl použít *shared_ptr* ze standardní knihovny *C++*. Nejenže bude paměť spravována automaticky, ale lze velice jednoduše vytvořit duplicitní reference na tentýž objekt, viz obrázek 4.1.



Obrázek 4.1: Sběrnice pomocí *shared_ptr* struktury může vlastnit jednotlivé komponenty, ale také umožňuje sdílení mezi těmito komponentami.

Další odpovědností sběrnice je správná distribuce čtení a zápisů jednotlivým komponentám. Všechny tyto paměťové operace závisejí na *Memory-mapped I/O*. *Memory-mapped I/O*

mapuje lineární 32-bitový paměťový prostor na segmenty, skrz které může sběrnice ovládat jednotlivé komponenty[1]. Jakákoliv komponenta v emulátoru může zavolat 2 funkce sběrnice pro interakci s ostatními komponentami: *dispatch_read* a *dispatch_write*. Obě funkce na základě šířky dat a paměťové mapy popsané v tabulce 4.1 rozhodnou, které komponentě tato data zaslat, či ze které komponenty data získat.

Tabulka 4.1: Paměťová mapa

Název	**Lokace**	**Velikost**
RAM	0x00000000	2 MiB + 3x mirrors
Expansion	0x1F000000	1 MiB
Scratchpad	0x1F800000	1 KiB
Ovladač paměti	0x1F801000	36 B
Periférie	0x1F801040	16 B
Serial	0x1F801050	16 B
Ovladač RAM	0x1F801060	4 B
Ovladač přerušení	0x1F801070	8 B
DMA	0x1F801080	128 B
DotClock Časovač	0x1F801100	16 B
HBlank Časovač	0x1F801110	16 B
SystemClock/8 Časovač	0x1F801120	16 B
CDROM	0x1F801800	4 B
GPU	0x1F801810	8 B
MDEC	0x1F801820	8 B
SPU	0x1F801C00	1 KiB
I/O porty	0x1F802000	8 KiB
BIOS	0x1FC00000	512 KiB
Ovladač cache	0x1FFE0130	4 B

V systému je několik paměťových regionů, které slouží pro ukládání generických dat. Jsou to: *RAM*, *Expansion*, *Scratchpad (nebo taky známá jako fast RAM, či data cache)*, *Video RAM* (vlastněno *GPU*) a *Sound RAM* (vlastněno *SPU*). K těmto paměťovým regionům lze přistupovat s různou šířkou dat (8, 16 a 32 bitové jednotky). Pomocí *STL* knihovny *C++* byla implementována třída *MemoryRegion*, pomocí níž jsou všechny tyto emulované paměťové regiony realizovány. Třída řeší správné adresování na základě různé datové šířky a také zajišťuje korektní kontroly mezí.

Ačkoliv standardní knihovna *C++* má způsob, jak spravovat statické pole pomocí třídy *array*, ošetření přístupů k tomuto poli s různými datovými šířkami by vyžadovalo extrahování ukazatele na alokované pole pomocí funkce *array::data*, čímž bychom ztratili kontrolu mezí, protože třída *array* vyžaduje specifikaci typu uložené jednotky. Proto bylo nutné implementovat zvláštní třídu pro tyto speciální účely.

Jelikož sběrnice má 32-bitovou šířku, mohlo by se zdát, že stačí pouze na základě paměťové mapy vzít danou adresu a rozdistribuovat ji příslušné komponentě. Nicméně sběrnice tohoto systému má 2 zvláštnosti: Správa virtuální paměti popsané v sekci [4.1.1] a přímý zápis do vyrovnávací paměti instrukcí popsané v sekci [4.1.2].

[1]Paměťová mapa PlayStation [4]: https://psx-spx.consoledev.net/iomap/

4.1.1 Virtuální paměť

Jak bylo naznačeno, *PlayStation* je do jisté míry zjednodušená verze stolního počítače a existují určité artefakty a ostatky, které zajišťovaly plnou funkčnost stolního počítače. Jedním z takovýchto artefaktů je virtuální paměť. I přesto, že procesor má podporu *Translation Lookaside Bufferu (TLB)* pro efektivní správu virtuální paměti, *PlayStation* jej vůbec nevyužívá a všechny přístupy do paměti jsou v zásadě absolutní. To, co zůstalo z virtualizace paměti, je maskování všech adres, které přijdou na sběrnici a na základě velikosti zapisovaných či čtených dat probíhá maskování odlišně. Nejdříve se u každého přístupu zahodí vrchní 3 bity. Poté u půl-slova (16 bitů) se zahodí spodní 1 bit a u slova (32 bitů) se zahodí spodní 2 bity. Zahození spodních bitů souvisí se zarovnáním do paměti[2].

4.1.2 Vyrovnávací paměť instrukcí

Druhá zvláštnost, na kterou je třeba myslet při distribuci čtení a zápisu, je vyrovnávací paměť instrukcí uvnitř *CPU*. Tato vyrovnávací paměť slouží pro rychlé získávání instrukcí z hlavní paměti a teoreticky by nikdy neměla nastat situace, kdy ve vyrovnávací paměti bude něco jiného, než co je obsaženo v hlavní paměti. Ovšem to není u tohoto systému vždy pravda.

CPU má speciální stav, který se dá programově nastavit a který takzvaně *izoluje vyrovnávací paměť instrukcí*. Pokud se *CPU* nachází v tomto stavu, pak každé čtení za účelem získat instrukce (*fetch* fáze *CPU*) nikdy nebude přecházet do hlavní paměti, ale do této vyrovnávací paměti a **každý** zápis bude modifikovat vyrovnávací paměť namísto hlavní paměti. *CPU* tedy jinými slovy poskytuje přímý přístup do této vyrovnávací paměti. Pokud toto není správně ošetřeno a simulováno, *PlayStation* nebude schopen nastartovat[3], protože *BIOS* se v prvé řadě snaží vyčistit tuto vyrovnávací paměť tak, že ji vyplní samými nulami. Nicméně pokud zápisy nejsou filtrovány, *BIOS* přepíše nulami svůj vlastní *shell* program a tedy zničí svoji zaváděcí fázi.

Toto chování je reflektováno ve funkci *dispatch_write* uvnitř sběrnicového modulu *Bus*, kde se kontroluje stav *CPU* a filtrují se zápisy.

4.1.3 Časování

Každá hardwarová komponenta pracuje v reálném čase paralelně a nezávisle. Situaci dále zhoršuje fakt, že každá komponenta má odlišnou frekvenci hodin[4]:

- **CPU** - 33.868800 MHz
- **GPU** - 53.222400 MHz
- **SPU** - 44.100 KHz
- **DotClock Timer** - 4.980705 MHz (Průměr)
- **HBlank Timer** - 9923 Hz (NTSC), 9943 Hz (PAL)
- **System Clock/8 Timer** - 4.233600 MHz

[2] Přístup do paměti [4]: https://psx-spx.consoledev.net/memorymap/

[3] Koprocesor 0, registr 12, bit 16 [4]: https://psx-spx.consoledev.net/cpuspecifications/#cop0r12-sr-system-status-register-rw

[4] Časování [4]: https://psx-spx.consoledev.net/timers/, https://psx-spx.consoledev.net/graphicsprocessingunitgpu/#gpu-timings

Tento problém je řešen tak, že emulátor simuluje každou komponentu sekvenčně a zvlášť v malých kvantech. Množství hodinových cyklů alokovaných pro danou komponentu reflektuje předchozí list hodnot. Tento přístup ovšem může způsobit časovou dilataci mezi jednotlivými komponentami a vést k narušení jejich synchronizace. Při volbě časového kvanta ho nesmíme zvolit příliš malé, protože by došlo k nesprávnému zaokrouhlení na základě časovací tabulky a nepřesnost časování by byla větší. Zároveň však kvantum nesmíme zvolit příliš velké, protože by se komponenty nemusely dostat včas ke slovu. Dvě komponenty, pro které je synchronizace nejdůležitější, jsou *CPU* a *GPU*. Protože je *GPU* o $\frac{11}{7}$ rychlejší, zvolil jsem tedy časovou konstantu 105. Výsledek formule $105 \times \frac{11}{7} = 165$ je celé číslo a není tedy potřeba řešit zlomkovou část časování.

4.1.4 Načítání her/programů

Při inicializaci jádra emulátoru musí uživatelské rozhraní specifikovat cestu k obrazu disku hry, nebo cestu k *PSX Executable* souboru. V obou případech se musí upravit vnitřní stav emulátoru, aby byl schopen dodaný program spustit.

U **načítání disku** jádro přijímá speciální formát souboru obsahující nejenom data hry, ale také sub-kanály. Odkaz na obraz hry (Obsah není přečten, pouze se otevře soubor ve čtecím módu) je předán *CD-ROM*, která hru analyzuje a rozdělí ji do stop (tento emulátor nepodporuje více stop v jedné hře, jako jsou například audio stopy, čili *CD-ROM* vytvoří pouze jednu stopu). Poté je při vyžádání dat z *CD-ROM* mechaniky soubor v určitém místě přečten jako 2352-bytový sektor.

Tabulka 4.2: Formát PSX Executable

Název	**Typ**	**Velikost**
PSX magic	uint8	8
Výplň/Neznámé	uint8	8
Program Counter	uint32le	1
Global Pointer (R28)	uint32le	1
Pozice *TEXT* Sekce	uint32le	1
Velikost *TEXT* Sekce	uint32le	1
Pozice *DATA* Sekce	uint32le	1
Velikost *DATA* Sekce	uint32le	1
Pozice *BSS* Sekce	uint32le	1
Velikost *BSS* Sekce	uint32le	1
Stack Pointer Base	uint32le	1
Stack Pointer Offset	uint32le	1
Výplň/Neznámé	uint8	20
Sony Podpis	uint8	1972
TEXT Sekce	uint32le	...
DATA Sekce	uint32le	...
BSS Sekce	uint32le	...

Při načítání **PSX Executable** musí jádro nejdříve zpracovat hlavičku spustitelného souboru. Obsah této hlavičky je popsán v tabulce 4.2[5]. Z ní jádro extrahuje počáteční hodnoty *CPU* registrů a vyplní *RAM* paměť instrukcemi programu v *TEXT* Sekci. Aby se program správně spustil, musí se nejdřív provést inicializační rutina, kterou provádí *BIOS*. Jakmile *BIOS* začne spouštět svůj *shell*, je třeba *CPU* ručně přerušit, nastavit registry na jejich iniciální hodnoty tak, jak jsou obsaženy v hlavičce *PSX Executable*, přepsat *RAM* paměť a posléze znovu povolit simulaci systému. Pro tyto účely lze využít *Breakpointy*, kde přerušíme chod systému, jakmile *CPU* dosáhne adresy **0x8003'0000**, což indikuje počátek *shell* programu.

4.2 Centrální procesorová jednotka (CPU)

Pokud je sběrnice nervovým systémem, pak *CPU* je mozkem *PlayStation* systému. Jde o *MIPS R3000A* 32-bitový *RISC* procesor. Jeho architektura je založena na *MIPS I* redukované instrukční sadě. *CPU* má celkem 5 fází, ve kterých se může jedna instrukce nacházet a které CPU provádí paralelně v nepřetržité smyčce:

- **Fetch (IF)** fáze - získání následující instrukce z paměti *RAM*.
- **Decode (ID)** fáze - dekódování instrukce a zjištění následující operace.
- **Execute (EX)** fáze - *CPU* provede dekódovanou instrukci (aritmetickou či logickou operaci).
- **Memory Access (MEM)** fáze - pokud instrukce přistupuje do paměti, data jsou pomocí sběrnice přečtena či zapsána do paměti *RAM*.
- **Write Back(WB)** fáze - Výsledek instrukce je zapsán do souboru registrů.

Emulátor tuto linku částečně respektuje, ale neprovádí ji paralelně. **IF** fáze je implementovaná jako samostatná jednotka, ovšem fáze **ID**, **EX**, **MEM** a **WB** jsou prováděny naráz, přičemž u fází **MEM** a **WB** pak navíc je nutná režie pro zpoždění načítaných a ukládaných hodnot pro simulaci paralelního zpracování jednotlivých fází. Tato režie je popsána níže v sekci [4.2.2]. Transformace toku instrukcí v emulátoru na funkční kód je prováděna pomocí jednoduché interpretace. Emulátor uchovává tabulku ukazatelů na funkce, přičemž operační kód instrukce pak specifikuje index do této tabulky ukazatelů, kde indexovaná funkce pak implementuje funkcionalitu dané instrukce.

4.2.1 Vnitřní stav *CPU*

Pro ukládání mezivýsledků a pro obecné zpracování logiky, *CPU* obsahuje celkem **32** 32-bitových registrů, plus **2** 32-bitové registry, které jsou specializované pro práci s násobícími a dělícími instrukcemi [3]. *Nultý* registr je speciální tím, že při čtení vrací vždy nulu a jakýkoliv zápis do něj je ignorován.

Vzhledem k tomu, že *nultý* registr má speciální chování a navíc registry vykazují fenomén zpoždění načítání hodnot, viz sekci [4.2.2], bylo nutné vytvořit v emulátoru speciální logiku pro načítání hodnoty do registru. Tato funkcionalita je implementována ve funkci *CPU::set_register*.

[5] formát PSX executable [4]: https://psx-spx.consoledev.net/cdromfileformats/#filenameexe-general-purpose-executable

4.2.2 Zpoždění načítání hodnot

MIPS R3000A, jako každý *RISC* procesor, vykazuje zvláštní jev při načítání hodnot do registrů. Pokud se snažíme načíst hodnotu z paměti *RAM* do procesoru, zpoždění způsobené načítáním hodnoty má za následek opožděné nastavení registru na přečtenou hodnotu o jeden procesorový takt [3]. Toto je způsobeno samotným návrhem *RISC* architektury a je nutné tuto situaci ošetřit.

V emulátoru je tento fenomén simulován pomocí dvou registrových přihrádek, které fungují jako fronta. Kdykoliv chce *CPU* přečíst hodnotu z paměti *RAM*, místo přímého nastavení registru, přečtená hodnota spolu s indexem výsledného registru v prvním taktu je vložena do fronty a po druhém taktu se modifikuje registrové pole. V sekci [4.2.1] bylo řečeno, že nultý registr při zápisu ignoruje hodnotu a při čtení vždy vrací hodnotu nula. Díky tomu lze prázdnou přihrádku reprezentovat jako zapisování do nultého registru.

Je nutné také pamatovat na situaci, kdy je ve frontě připravená hodnota, ale mezitím přijde jiná instrukce, která stejný registr modifikuje. V takovém případě je nutné frontu vyčistit, aby se výsledný registr neobsahoval špatnou hodnotu. Příklad opožděného načítání můžeme vidět v obrázku 4.2.

	IF	ID	EX	MEM	WB	r1
1. LW r1 [r2]	1.					0
2. ADDIU r1 r1 4	2.	1.				0
3. NOP	3.	2.	1.			0
4. NOP	4.	3.	2.	1.		0
5. NOP	5.	4.	3.	2.	1.	8
6. NOP	6.	5.	4.	3.	2.	4

Obrázek 4.2: Příklad programu a vizualizace opoždění načtení hodnoty z paměti *RAM*. Od toku programu bychom očekávali, že z adresy *0* se přečte 32-bitová hodnota (dejme tomu hodnota *8*) a poté se k ní přičte *4*, přičemž ve výsledku v registru *r1* by měla být výsledná hodnota *12*. Protože je ovšem přečtení z paměti opožděno, *ADDIU* instrukce počítá se špatnou hodnotou. Řešením by bylo vložit mezi instrukce *1.* a *2. NOP*, nebo jinou užitečnou nezávislou instrukci.

4.2.3 Zpoždění skoku

Podobně jako opožděné načítání hodnot do registru, design 5-stupňové architektury má na svědomí ještě jeden problém. Tento artefakt se vyskytuje u všech skokových instrukcí a má za následek že bez ohledu na to, zda-li se v případě podmíněných skokových instrukcí skočí nebo ne, následující instrukce po skokové instrukci se vždy provede [3].

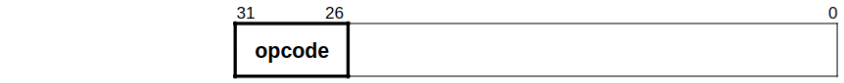
To je způsobeno tím, že následující instrukce je již načtena a dekódována uvnitř *CPU*. Kompilátory té doby byly dobře seznámeny s tímto fenoménem a ve většině případů vyplnily instrukci za skokem prázdnou instrukcí. Z tohoto důvodu je každý skok v emulátoru zpožděn o jeden takt.

4.2.4 Zpracování breakpointů

Pro snazší debugování při vývoji hry, *CPU* podporovalo softwarové a hardwarové *breakpointy*. Jelikož breakpointy indikovaly buď špatný tok programu, či špatnou práci s hardwarovou komponentou, v emulátoru je zpracování breakpointů dosti zjednodušené a nereflektuje jejich reálně zpracování, jak by je zpracoval původní systém. *CPU* v emulátoru uchovává množinu adres pomocí třídy *set* ze standardní knihovny *C++*, kde pokud programový čítač narazí na některou z nich, celý emulátor je pozastaven a počká, až přijde potvrzení breakpointu od jádra emulátoru. Breakpoint je posléze smazán z množiny adres a *CPU* pokračuje dále v provádění programu.

Breakpoint je v celém emulátoru využit pouze jednou a to při načítání *PSX Executable* souboru.

4.2.5 Instrukční sada



Instructions encoded by opcode field.

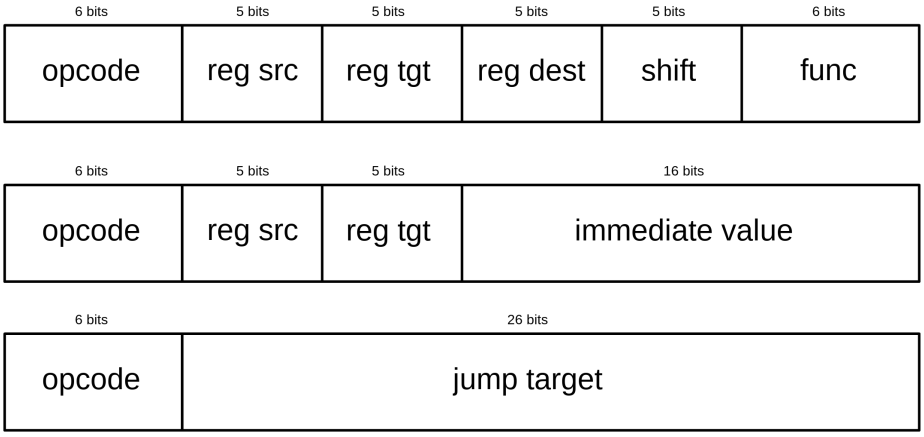
opcode bits 28..26 / bits 31..29	000	001	010	011	100	101	110	111
0 000	*SPECIAL* δ	*REGIMM* δ	J	JAL	BEQ	BNE	BLEZ	BGTZ
1 001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2 010	*COP0* δ,π	*COP1* δ,π	*COP2* δ,π	*COP3* δ,π,κ	*	*	*	*
3 011	*	*	*	*	*	*	*	*
4 100	LB	LH	LWL	LW	LBU	LHU	LWR	*
5 101	SB	SH	SWL	SW	*	*	SWR	*
6 110	*	LWC1 π	LWC2 π	LWC3 π,κ	*	*	*	*
7 111	*	SWC1 π	SWC2 π	SWC3 π,κ	*	*	*	*

Instructions encoded by function field when opcode field = SPECIAL.

function bits 2..0 / bits 5..3	000	001	010	011	100	101	110	111
0 000	SLL	*	SRL	SRA	SLLV	*	SRLV	SRAV
1 001	JR	JALR	*	*	SYSCALL	BREAK	*	*
2 010	MFHI	MTHI	MFLO	MTLO	*	*	*	*
3 011	MULT	MULTU	DIV	DIVU	*	*	*	*
4 100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5 101	*	*	SLT	SLTU	*	*	*	*
6 110	*	*	*	*	*	*	*	*
7 111	*	*	*	*	*	*	*	*

Obrázek 4.3: Kompletní mapa všech **69** instrukcí podle *MIPS* specifikace [7]. V nespecifickém pořadí, procesor má instrukce pro ovládaní toku programu, aritmetické instrukce, logické instrukce a instrukce pracující s pamětí.

Instrukční sada *CPU*, založená na architektuře *MIPS I*, obsahuje relativně malý počet instrukcí. Tato sada zahrnuje *40* základních instrukcí a *29* rozšířených instrukcí, což dohromady činí **69** instrukcí celkem. Mapa instrukcí je podchycena v obrázku 4.3. Tyto instrukce jsou navrženy tak, aby každá z nich zabírala jeden procesorový takt, čímž je udr-



Obrázek 4.4: *MIPS* má celkem 3 způsoby, jak zakódovat instrukci do 32 bitů.

žována neustále plněná 5-ti úrovňová linka. Díky této filozofii mají instrukce velmi málo zodpovědností a jsou ve své podstatě velmi jednoduché.

Aby se předešlo složitému dekódování instrukcí, jak tomu je například u architektury *x86_64*, kde každá instrukce může mít proměnlivou délku, *MIPS I* definuje každou instrukci jako 32-bitové slovo. Horních 6 bitů pak definuje kódování specifické instrukce. Proto lze každou instrukci rozdělit do zhruba tří tříd, viz obrázek 4.4. *MIPS* Instrukce v emulátoru je implementována jako sjednocení bitových polí. V *C++* lze využít anonymních struktur a sjednocení, kde jednotlivé bitové alokace se mohou překrývat a tedy umožňuje emulátoru snadno instrukce interpretovat.

4.2.6 Koprocesory

MIPS R3000A má ve své instrukční sadě podporu pro celkem 4 různé koprocesory, avšak *PlayStation* využívá pouze 2 z nich[6]. Koprocesorové instrukce jsou všestranné a lze za ně substituovat jakýkoliv čip, kromě **koprocesoru 1**, který je dedikován pro práci s čísly s plovoucí desetinnou čárkou [7] (není přítomen v *PlayStation* konzoli).

Koprocesor 0 - Správce výjimek

Koprocesor 0 slouží ke správě výjimek a přerušení. Koprocesor obsahuje nejen informace o typu výjimky nebo o tom, kdo způsobil přerušení, ale také logiku pro zpracování výjimky. Procesor při vyhození výjimky uloží současný stav a jeho kontrolní tok je přenesen na rutinu, která se stará o obsluhu výjimky. Pro správné ukládání stavu při vyhození výjimky je nutné myslet na situaci, kdy se procesor nachází ve stavu opožděného skoku. Pokud bychom informaci o provedení skoku neuložili při vyhození výjimky, po zpracování výjimky a obnovení původního stavu procesoru by se původní skok nemusel provést. To znamená, že při každém procesorovém taktu je nutné ukládat informaci o skokové prodlevě a zda-li se skok provede.

[6] CPU specifikace [4]: https://psx-spx.consoledev.net/cpuspecifications/

Koprocesor 2 - Geometry transformation engine (GTE)

Koprocesor 2 pak zpřístupňuje komponentu *Geometry Transformation Engine (GTE)*, což je hardware specializovaný pro rychlou práci s lineární algebrou. Tento koprocesor pracuje se dvěma základními datovými primitivy: 3D vektory (16/32-bitový atom) a 3x3 matice (16-bitový atom). Vektory mohou být interpretovány jako body v prostoru nebo jako barvy a pro každý typ má *GTE* dedikované příkazy. Funkce *GTE* jsou velmi všestranné, zahrnují rychlé násobení matice s vektorem, normalizaci barev nebo vektoru a interpolaci. Tento koprocesor je nepochybně klíčový pro rychlé zpracování geometrie ve hře a efektivnější vykreslení 3D scény.

Hlavní schopností *GTE* je operace implementovaná ve funkci *multiply_and_translate*. Tato operace implementuje násobení 3D matice a 3D vektoru, přičemž výsledek násobení může být také posunut o 3D *translate* vektor. Tyto výpočty jsou dělány čistě na hostovacím *CPU*.

4.3 Grafická procesorová jednotka (GPU)

Grafická jednotka systému je speciálně navržený čip pro hardwarovou podporu rasterizace geometrických primitiv[7]. *GPU* pracuje na frekvenci 53.222400 MHz, což znamená, že je o $\frac{11}{7}$ rychlejší než frekvence *CPU*. Všechna komunikace s *GPU* probíhá přes čtyři registry:

- **GP0** (pouze zápis) - registr pro odesílání rasterizačních příkazů a posílání geometrických dat. Různé grafické příkazy mohou mít různou délku (počet zapsaných 32-bitových slov) a jsou spravovány přes komunikační frontu.
- **GP1** (pouze zápis) - registr pro modifikaci stavu *GPU* (například: reset, nastavení rasterizačního okna či potvrzení přerušení).
- **GPUREAD** (pouze čtení) - registr pro čtení *VRAM* paměti nebo pro čtení speciálních registrů.
- **GPUSTAT** (pouze čtení) - registr pro čtení celkového stavu *GPU*.

4.3.1 Video RAM (VRAM)

GPU má také vlastní paměť nazývanou *Video RAM (VRAM)*. Paměť je rozdělena do 512 řádků o 1024 16-bitových slovech. Celková kapacita paměti *VRAM* tedy činí: $1024 \times 512 \times 2 = 1048576B = 1MiB$[8].

VRAM paměť slouží pouze k ukládání textur, palet pro indexované textury (*CLUT*, viz sekci [4.3.3]) a výsledného *framebufferu*. Jakákoli data o kreslených primitivech (pozice vrcholů či texturovacích souřadnicích) musí být předána přes registr **GP0** a jsou okamžitě vykreslena a tedy tyto informace *VRAM* nemůže obsahovat.

GPU má potom 2 různé módy interpretace *VRAM framebufferu* týkající se barevné hloubky. Při vykreslování na obrazovku může být *VRAM* interpretována jako sekvence 16-bitových barev (kde nevýznamnější bit slouží jako maska a zbytek jsou 5-bitové RGB hodnoty), nebo 24-bitových barev (8-bitové RGB hodnoty). Oba módy mají svoje pravidla a omezení. Ačkoliv *GPU* může rasterizovat primitiva v obou módech, obecně ve většině

[7] GPU specifikace [4]: https://psx-spx.consoledev.net/graphicsprocessingunitgpu/

[8] GPU VRAM [4]: https://psx-spx.consoledev.net/graphicsprocessingunitgpu/#gpu-video-memory-vram

her platilo, že pokud se renderovala 3D scéna využíval se 16-bitový mód, protože pak ve *VRAM* paměti zbylo více místa pro textury. 24-bitový mód se pak většinou používal pouze u přehrávání videí, kde se ve *VRAM* paměti nemohlo ukládat v zásadě nic jiného než *framebuffer*, protože 24-bitový *framebuffer* při *dvojitém bufferingu*[9] mohl zabírat přes polovinu celé *VRAM* paměti (v závislosti na velikosti videa).

4.3.2 Geometrická primitiva

GPU dokáže vykreslit 3 geometrická primitiva:

- Osově zarovnaný obdélník
- Čára/Polyčára
- Trojúhelník/Čtyřúhelník

Všechna tato primitiva lze kreslit pomocí **GP0** registru[10], přičemž je nutné zapsat správný počet argumentů do tohoto registru na základě prvního 32-bitového slova příkazu. Počet argumentů daného primitiva závisí především na různých vlastnostech daného primitiva. Aby emulátor získal korektně všechny argumenty, emulátor v *GPU* interně uchovává příchozí data ve frontě, přičemž podle daného příkazu očekává určitý počet argumentů. Po provedení **GP0** příkazu se tato fronta vypláchne a čeká se na další příkaz. Ačkoliv ne všechny primitiva mohou mít všechny vlastnosti, *GPU* dokáže rasterizovat jednotlivá primitiva následujícími způsoby[11]:

- **Texturovací koordináty** - Primitivum má k sobě přiřazenou texturu a každý vrchol má asociované texturovací koordináty.
- **Průhlednost** - Na základě předchozího obsahu ve *VRAM* paměti dokáže *GPU* mixovat barvy ve čtyřech různých módech:
  - $half\text{-}each \Rightarrow result = \frac{background+source}{2}$
  - $additive \Rightarrow result = background + source$
  - $subtractive \Rightarrow result = background - source$
  - $\frac{additive}{4} \Rightarrow result = background + \frac{source}{4}$
- **Gouraudovo stínování** - Tento atribut je svým názvem trochu zavádějící, neboť gouraudovo stínování v moderní informatice souvisí spíše s výpočtem *per-vertex* osvětlení. V tomto případě jde ovšem pouze o zapnutí interpolace atributů mezi vrcholy primitiva.

Tyto atributy jsou při komunikaci přes **GP0** registr nashromážděny a použity v emulátoru při rasterizaci primitiva, viz sekci [4.3.5].

[9] Double-buffering: https://en.wikipedia.org/wiki/Multiple_buffering#Double_buffering_in_computer_graphics

[10] GPU GP0 registr [4]: https://psx-spx.consoledev.net/graphicsprocessingunitgpu/#gpu-other-commands

[11] GPU Semi-transparency [4]: https://psx-spx.consoledev.net/graphicsprocessingunitgpu/#semi-transparency

4.3.3 Color Lookup Table (CLUT) vyrovnávací paměť

V drtivé většině případů, kdy dané geometrické primitivum má asociovanou texturu, pixely této textury jsou indexy do specifické palety barev, neboli *Color Lookup Table (CLUT)*, která je uložena spolu s texturou uvnitř *VRAM*. Díky tomu mají textury relativně mnohem menší velikost, než kdybychom ukládali texturu s absolutními hodnotami pixelů, což za své doby bylo bezpodmínečně nutné, protože nakonec *VRAM* má pouze *1 MiB* místa. *CLUT* je spolu s texturou uložena ve *VRAM* jako lineární sekvence bytů.

Pokud ovšem rasterizujeme kupříkladu texturovaný obdélník, šance, že se *CLUT* bude radikálně měnit je blízko nule. Jelikož každé čtení z *VRAM* stojí GPU cykly, *GPU* má vyrovnávací paměť speciálně dedikovanou pro *CLUT*, přičemž jakmile započne rasterizace texturovaného primitiva, *GPU* nejdříve zkontroluje jaký *CLUT* region dané primitivum vyžaduje podle příslušných *CLUT* koordinátů. Pokud se tyto koordináty odlišují od posledních použitých *CLUT* koordinátů, nebo se liší barevná hloubka od poslední použité, *CLUT* vyrovnávací paměť se vypláchne a naplní se novými *CLUT* daty.

Při čtení texturových dat se pak exkluzivně používá *CLUT* vyrovnávací paměť. To ovšem platí jen u specifické barevné hloubky pixelu. *CLUT* vyrovnávací paměť se používá pouze u *4-bitové* a *8-bitové* barevné hloubky. *16-bitová* hloubka pak indikuje absolutní ukládání barev v textuře bez *CLUT*.

4.3.4 Barvy

Jak bylo zmíněno, *GPU* dokáže pracovat s 24-bitovými barvami, které obsahují 3 základní barevné komponenty (červená, zelená a modrá). Každému z těchto komponent je alokováno 8 bitů a dále dokáže pracovat s 15-bitovými barvami, kde každé komponentě je přiřazeno pouze 5 bitů.

Pokud programátor chtěl detaily 24-bitové hloubky, ale nechtěl přepnout *VRAM* do 24-bitového módu a mít tak méně prostoru pro kreslení, *GPU* poskytovalo hardwarovou funkcionalitu přesně pro tyto účely. Zatímco 15-bitová barva (plus maskovací bit) se pouze zkopíruje do paměti *VRAM*, 24-bitová barva může být za pomocí techniky *dithering*[12] rozdistribuována do kreslícího okolí, čímž může ošálit lidské oko a získat tak lepší barevnou hloubku.

Paměť *VRAM* v 16-bitovém módu v každém fragmentu také ukládá extra 1 bit, který figuruje jako kreslící maska. Pokud je nejvyšší bit v 16-bitové barvě nastaven na 1, pak tento fragment bude ignorován a nic se do něj nevykreslí.

4.3.5 Rasterizace primitiv

U každého ze 3 geometrických primitiv, které *GPU* dokáže vykreslit, je nutno zvolit správný algoritmus pro jeho rasterizaci. U rasterizace osově zarovnaného obdélníku stačí zjistit maximum a minimum ve 2D prostoru. Tento omezený prostor může být následně vyplněn.

Rasterizace čáry vyžaduje sofistikovanější přístup, čímž je **Digital Differential Analyzer (DDA) algoritmus**[13], který na základě výpočtu sklonu čáry dokáže vyplnit fragmenty mezi dvěma body ve 2D prostoru. Algoritmus *DDA* zjednodušeně funguje následujícím způsobem:

[12] GPU dithering [4]: https://psx-spx.consoledev.net/graphicsprocessingunitgpu/#24bit-rgb-to-15bit-rgb-dithering-enabled-in-texpage-attribute

[13] DDA algoritmus: https://en.wikipedia.org/wiki/Digital_differential_analyzer_(graphics_algorithm)

1. spočítáme rozdíl $d = line_{end} - line_{start}$
2. na základě kvadrantu prohodíme x a y souřadnice rozdílu d i souřadnic čáry $line_{start}$ a $line_{end}$
3. vypočítáme sklon čáry jako $k = d.y/d.x$
4. ve smyčce od $line_{start}.x$ do $line_{end}.x$ vyplňujeme pixely, přičemž y souřadnici při každé iteraci navyšujeme o hodnotu k

Pro urychlení algoritmu lze použít *fixed-point* aritmetiku, kde při každé adresaci fragmentu operace pracuje pouze s celými čísly.

Trojúhelníková rasterizace se řadí mezi vyplňovací rasterizační algoritmy. Pro jeho implementaci jsem zvolil základní variantu **Pinedova algoritmu** [6]. Jeho podstata závisí na rozdělení jednotlivých hran trojúhelníku na poloroviny a u každého fragmentu zjišťovat jeho polohu v závislosti na všech polorovinách daného trojúhelníku. Zjednodušený algoritmus, který dokáže trojúhelník rasterizovat vypadá zhruba takto:

1. Spočítáme ohraničující obdélník:

$$topleft = min(tri_a, tri_b, tri_c)$$

$$bottomright = max(tri_a, tri_b, tri_c)$$

2. Spočítáme jednotlivé hrany trojúhelníku jako vektory:

$$\vec{delta}_{ba} = tri_b - tri_a$$

$$\vec{delta}_{cb} = tri_c - tri_b$$

$$\vec{delta}_{ac} = tri_a - tri_c$$

3. Pro každý fragment ohraničujícího obdélníku zjistíme relativní polohu daného fragmentu vůči třem hraničním polorovinám pomocí 2D vektorového součinu:

$$planepos_{cb} = \vec{delta}_{cb} \times (\vec{fragment}_{pos} - tri_b)$$

$$planepos_{ac} = \vec{delta}_{ac} \times (\vec{fragment}_{pos} - tri_c)$$

$$planepos_{ba} = \vec{delta}_{ba} \times (\vec{fragment}_{pos} - tri_a)$$

4. Znaménka jednotlivých relativních pozicí pak určují, na které straně poloroviny se daný fragment nachází. Pokud všechny relativní pozice mají stejné znaménko, znamená to, že se fragment vyskytuje uvnitř trojúhelníku:

$$sign(planepos_{cb}) = sign(planepos_{ac}) = sign(planepos_{ba})$$

Ačkoliv tento přístup je správný, kalkulace vektorového součinu pro každý fragment je zbytečná práce. Místo toho, vektorový součin lze vypočítat pouze pro první fragment, a při každé změně fragmentu stačí pouze přičíst *delty*. To znamená, že místo 6 násobení a 3

odčítání pro každou složku fragmentu pro výpočet vektorového součinu, nám stačí pouze 3 sčítání.

Při zpracování trojúhelníku a přiřazování atributů k jednotlivým vrcholům, je nutno myslet na orientaci trojúhelníku (pořadí vrcholů ve směru/proti směru hodinových ručiček). Ačkoliv *GPU* pracuje čistě s protisměrném pořadí hodinových ručiček, data přicházející do *GPU* nejsou nutně v tomto pořadí.

Pokud *GPU* v emulátoru dostane data v opačném pořadí, interně tuto chybu detekuje pomocí vektorového součinu dvou hran trojúhelníku. Na základně znaménka poté jednotlivé vrcholy prohodí pro správné protisměrné pořadí hodinových ručiček.

U rasterizace čáry/polyčáry a trojúhelníku, rasterizace reálného *GPU* využívá *fixed-point* aritmetiky, což má za následek, že vykreslené polygony mají tendenci se "chvět", protože jakmile jsou polygony transformovány pomocí *GTE*, výsledné vrcholy mají celočíselné koordináty. Bohužel existují konfliktní informace o tom, kolik bitů je alokováno pro celočíselnou část a kolik pro zlomkovou část a ve které části rasterizace se bere pouze celočíselná část koordinátu. V tomto bodě byla nutná experimentace, přičemž nejvěrohodnější výsledky byly dosaženy s 32-bitovou *fixed-point* hodnotou s 12-bitovou zlomkovou částí.

4.3.6 Zvýšení rozlišení

Zvýšení rozlišení se týká především *GPU* a rasterizace jednotlivých primitiv. *GPU* ve svém interním stavu ukládá mimo jiné meze *framebufferu*. Tyto meze pak figurují při rasterizaci, kde jakýkoliv pokus kreslit mimo tyto meze bude ignorován. Zároveň tyto meze určují, odkud ve *VRAM* paměti se budou brát data pro zasílání do *CRT* monitoru.

Pro tyto účely je nutné spravovat separátní *High resolution VRAM* (*Hi-res VRAM*) paměť, která podle nastavení bude mít násobek velikosti současné reálné *VRAM* paměti uvnitř *GPU*.

U rasterizaci primitiv je pak nutné zachytit a správně přepočítat jejich pozice, stejně jako upravit jejich omezující vlastnosti. Největší potíž však je při adresaci textur a jejich palet. Při každém zápisu textury a indexu barvy je nutné koordináty přemapovat o současný násobek velikosti *Hi-res VRAM*. Tento problém je vyřešen tak, že při používání *Hi-res VRAM* kdykoliv rasterizace vyžaduje barvu textury, rasterizér sáhne do původní *VRAM* a výsledek je rozdistribuován do okolí *Hi-res VRAM*. To znamená, že při vykreslování ve vyšším rozlišení se musí kreslit do obou *VRAM* i *Hi-res VRAM* současně.

4.4 Jednotka přímého přístupu do paměti (DMA)

Jelikož *CPU* má hodinovou rychlost 33.8688 MHz, jakýkoliv přenos dat je nesmírně pomalý. Tento fakt je pouze umocněn v situaci, kdy program chce přenést data z paměti *RAM* do jiné hardwarové komponenty. Pokud bychom měli jednoduchou smyčku pro kopírování dat s prokládanou inkrementací indexu do paměti (abychom vyplnili zpoždění čtení z paměti *RAM* popsané v sekci [4.2]), dostaneme 4 instrukce pro kopírování a 3 instrukce pro správu cyklu, viz obrázek 4.5.

Z teoretického hlediska to znamená, že ideální rychlost přenosu činí $\frac{33.8688}{4+3} \times 4 = 19.3536 MB/s$. Kvůli této nevýhodě obsahuje *PlayStation* komponentu *Direct Memory Access (DMA)*, která slouží výhradně k velmi rychlému přenosu dat mezi pamětí *RAM* a hardwarovými komponentami. *DMA* má celkem 7 různých kanálů, přičemž každý kanál specifikuje komunikující komponenty[14].

[14] DMA [4]: https://psx-spx.consoledev.net/dmachannels/

```
.loop:            ; loop label
LW r1 [r2]        ; Load 32-bit value into r1 from address r2
ADDIU r2 r2 4     ; Increment r2 by 4 to move to the next 32-bit read value
SW r1 [r3]        ; Store 32-bit value back to ram to address r3
ADDIU r3 r3 4     ; Increment r3 by 4 to move to the next 32-bit write value
ADDI r4 -1        ; Update remaining words to copy
BNE r4 r0 .loop   ; If there are still things to copy, jump back to loop
NOP               ; Filler, which will get executed regardless if jump is made or not
```

Obrázek 4.5: Pomocí 7 instrukcí a správným prokládáním lze docílit maximální rychlosti při kopírování dat. Každý cyklus přenese 4 byty.

- **DMA Kanál 0. - MDECIN** - Z *RAM* do *MDEC* vstupu
- **DMA Kanál 1. - MDECOUT** - Z *MDEC* výstupu do *RAM*
- **DMA Kanál 2. - GPU** - Mezi *RAM* a *GPU*
- **DMA Kanál 3. - CDROM** - Z *CD-ROM* do *RAM*
- **DMA Kanál 4. - SPU** - Mezi *RAM* a *SPU*
- **DMA Kanál 5. - PIO** - Mezi *RAM* a *Expansion Port*
- **DMA Kanál 6. - OTC** - Mezi *RAM* a *GPU Ordering Table*

DMA také poskytuje celkem 3 různé módy přenosu:

- **Word Copy** - Jde o rychlý přenos lineární sekvence 32-bitových hodnot. Maximálně se může přenést až *65536* 32-bitových hodnot.
- **Block Copy** - Přenáší se bloky o uživatelem definované délce. Po dokončení přenosu jednoho bloku se pak čeká na připravenosti specifikované komponenty. Tento mód se především používá při přenosu dat z a do *MDEC* komponenty.
- **Linked List Copy** - Přenášená data se dělí na hlavičku a tělo. Hlavička obsahuje délku těla a adresu další hlavičky. Celá datová struktura je pak řetězec hlaviček a těl. Tento mód je hlavně využit při zasílání renderovacích příkazů do *GPU*.

Každý mód přenosu poskytuje velmi rychlé přenosy, protože *DMA* využívá *DRAM Hyper Page* mód, což umožňuje *DMA* přistupovat k *DRAM* řádkům v zásadě v jednom procesorovém cyklu. Tento přístup má minimální režii, která způsobí, že pro každých 17 cyklů se přečte 16 32-bitových slov. To znamená, že teoretické maximum přenosové rychlosti činí $33.8688 \times 4 \times \frac{16}{17} = 127.5060 MB/s$.

Při *DMA* přenosu má *CPU* přísná pravidla ohledně přístupu do paměti. Pokud probíhá přenos, *CPU* může přistupovat k vyrovnávacím pamětem a ke svým dvěma koprocesorům. Jakmile se *CPU* pokusí přistoupit do paměti, je jeho chod pozastaven, dokud *DMA* přenos není dokončen.

Díky tomuto faktu lze v emulátoru jakoukoliv *DMA* synchronizaci obejít tím, že se celý přenos provede najednou a *CPU* je pozastaveno. Výjimkou je *MDEC*, kde pro simulaci synchronizace tato komponenta obsahuje speciální příznak, který je modifikován při

začátku, či při dokončení dekódování, a který indikuje připravenost přenosu dat z a do *MDEC* komponenty.

DMA umožňuje pomocí speciálního registru definovat priority jednotlivých kanálů. To znamená, že se některé *DMA* kanály dostanou ke slovu dříve než ostatní. Toto chování je v emulátoru implementováno pomocí prioritní fronty (*priority_queue* z *C++* standardní knihovny). Při každém emulačním kvantu *DMA* ovladač extrahuje jednotlivé priority a zkonstruuje prioritní frontu, pomocí níž jsou jednotlivé kanály zpracované ve správném pořadí.

4.5 Ovladač přerušení/výjimek

Aby *CPU* mělo přehled o různých událostech, jako je například dokončení *DMA* přenosu, *PlayStation* má 2 úzce svázané hardwarové komponenty: *Ovladač přerušení* a *Ovladač výjimek*. *Ovladač přerušení* je propojen v podstatě se všemi ostatními komponentami, od kterých je schopen přijímat požadavky na přerušení. Tento čip také obsahuje stavový registr, ze kterého lze zjistit kdo přerušení způsobil.

To ale není dostatečné pro pozastavení chodu procesoru. *Ovladač přerušení* je napojen na *koprocesor 0 CPU*, tedy *Ovladač výjimek*, přičemž přerušení je pouze zvláštní typ výjimky.

BIOS následně disponuje předem definovanými vektory adres, na jejichž základě se rozhoduje kam přesměrovat řízení procesoru a následně zpracovat výjimku.

4.6 Časovače

PlayStation má 3 různé druhy časových zdrojů, plus jeden časový zdroj pro *CPU*. Jelikož se tyto 3 hardwarové složky příliš neliší, v emulátoru jsem využil *STL* knihovnu *C++* k instanciaci každého ze tří zdrojů.

První zdroj se nazývá **Dot Clock**. Tento zdroj souvisí s renderovacím módem *GPU* komponenty. *Dot* v tomto kontextu reprezentuje jeden vykreslený fragment (tzn. ne pixel na obrazovce. Fragment může mít větší či menší velikost než pixel.) a tento zdroj tedy počítá počet vykreslených fragmentů, přičemž rychlost závisí na vertikálním rozlišení *GPU*.

Druhý zdroj také souvisí s *GPU* a jeho název je **Horizontal Blank Clock**. Pokaždé, když *GPU* dokončí kreslení jednoho řádku *Framebufferu*, tento zdroj je inkrementován o jedničku. Rychlost závisí na regionu konzole a verzi *BIOSu* (*NTSC*/*PAL*).

Třetí zdroj je **System Clock**, který odráží zdroj hodin *CPU*, přičemž je ale podělen osmi. Každý z těchto časovačů má také schopnost vyvolat přerušení, pokud dosáhne určité hodnoty. Všechny tyto časovače kromě svých specifických zdrojů mají také zdroj hodin *CPU* a jsou schopny svůj zdroj měnit[15]. Časovače pak ve hrách byly využívány pro různé účely, jako například:

- Semínko pro generátor náhodných čísel
- Speciální (per-scanline) vizuální efekty

[15] Časovače [4]: https://psx-spx.consoledev.net/timers/

4.7 Dekodér makrobloku (MDEC)

Tento hardwarový čip je jedním z hlavních komponent, která učinila *PlayStation* velmi populární konzolí. Jelikož se *Sony* rozhodlo použít *CD* jako hlavní úložiště pro hry, každá hra měla na tu dobu velký prostor. V porovnání se svým konkurentem, *Nintendo 64*, který mohl ukládat maximálně *64 MB*, *PlayStation* disponoval až *600 MB*, přičemž hra mohla obsahovat i více disků a hráč mohl během hry měnit disky podle postupu ve hře.

Takový prostor bylo třeba využít a *Sony* se rozhodlo pro full-motion video. *MDEC* je čip speciálně věnovaný na dekódování patentovaného formátu videa. Hlavní myšlenkou tohoto formátu byla *JPEG* komprese obrázku. *JPEG* používá několik kompresních technik, které efektivně kombinuje dohromady. Obrázek je rozdělen na *makrobloky* o velikosti 8x8 nebo 16x16 pixelů. Tyto bloky jsou následně převedeny do *YCbCr* barevného formátu a *CbCr* složky jsou zploštěny na polovinu, přičemž složka intenzity *Y* je zachována v plném rozlišení. Toto je možné, protože lidské oko je citlivější na intenzitu světla než na samotnou diferenci barev, aby člověk ve tmě byl schopen alespoň rozlišit tvary. Poté je provedena *Discrete Cosine Transform (DCT)*, frekvenční analýza a přebytečné frekvence jsou odstraněny pomocí kvantizační tabulky. Výsledný makroblok je přeorganizován *Zig Zag* vzorem, aby potom následující technika *Run-Length* byla schopna zakódovat daný makroblok co nejefektivněji. *JPEG* pak využívá *Huffmanova* kódování pro vytvoření výsledného datového toku. Dekódování probíhá velmi podobným způsobem, ale obráceným směrem, viz obrázek 4.6[16].

Formát snímku videa v *PlayStationu* je velice podobný *JPEGu*, s výjimkou toho, že *PlayStation* nepoužívá *Huffmanovo* kódování. To má za následek mnohonásobně rychlejší dekódování videa, ale samozřejmě uložený video soubor má pak mnohonásobně větší velikost.

Při přesunu dat, *MDEC* má speciální příznak pomocí kterého indikuje stav dekódování (připravenost vstupu a výstupu).

4.8 CD-ROM mechanika

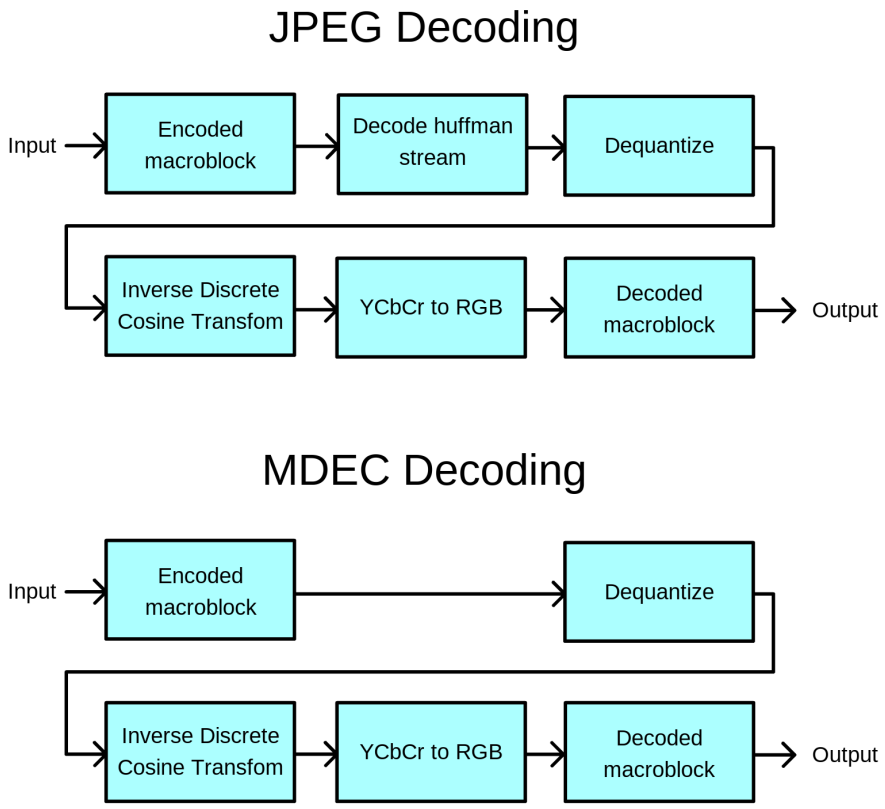
Sony se rozhodlo použít pro herní úložiště *Compact Disc Extended Architecture (CD-XA)* a standard ISO 9660. *BIOS* je zodpovědný za analýzu ISO 9660 souborového systému a poskytuje hrám k jednotlivým souborům přístup, čili emulátor se o tuto část nemusí starat.

CPU může komunikovat s *CD-ROM* mechanikou jednak přes registry a příkazy, ale navíc *CD-ROM* má také tři datové fronty, pomocí nichž se přenášejí data a dotazy na přerušení. Interně *CD-ROM* disponuje celkem *37* příkazy, pomocí kterých lze manipulovat se čtecím motorem.

Pro správný přístup k *CD* je nutné správně adresovat toto médium. Jelikož je *CD-ROM* spíše chápán jako úložiště pro audio či video média, disk je adresován pomocí stop. Každá stopa se dá reprezentovat indexem nebo formátem *minuty (MM):sekundy (SS):zlomky (FF)*. Každý disk má maximálně *74* minut, přičemž v každé minutě je *60* sekund a každá sekunda obsahuje *75* zlomků. S touto znalostí se pak tento formát dá převést na *Logical Block Addressing (LBA)*, což je schopnost adresovat *CD* jako lineární sekvenci bytů.

Hry také měly možnost být prodávány ne na jednom, ale na několika discích. *CD-ROM* má pak schopnost výměny disku bez resetování konzole. Pomocí vypnutí motoru lze *CD-ROM* mechaniku pozastavit, otevřít poklop, vyměnit disk a program po uzavření poklopu

[16] MDEC [4]: https://psx-spx.consoledev.net/macroblockdecodermdec/



Obrázek 4.6: *MDEC* má velmi podobnou strukturu jako *JPEG* až na vynechaný krok Huffmanova dekódování.

může bezproblémově pokračovat[17]. Tato funkcionalita ovšem bohužel není v emulátoru implementována a tedy některé hry uživatel nemá schopnost dokončit.

4.9 Periférie

Ačkoliv samotná konzole má vnitřně několik typů periférií, jako jsou I/O a debugovací porty, veřejně dostupná verze konzole má celkem 4 porty, se kterými běžný uživatel může interagovat. Z těchto portů slouží 2 na vsunutí *MemoryCard*, což figurovalo jako malé úložiště pro uchování postupu ve hrách. Zbývající 2 porty pak fungují jako vstupy pro herní ovladače, přičemž existují celkem 3 oficiálně podporované typy: *Digital*, *Analog* a *Mouse*.

Hry mohou individuálně číst z těchto portů a umožňují tak uživateli s konzolí interagovat. Veškerá komunikace mezi *CPU* a periferiemi probíhá pomocí *Serial I/O (SIO)*[18]. *PlayStation* měl podporu pro nemálo typů herních ovladačů[19], ovšem drtivá většina her z *PlayStation* knihovny dokázala používat pouze standardní digitální ovladač. Čili v emulátoru je implementovaná komunikace pouze pro tento typ ovladače.

[17] CD-ROM mechanika [4]: https://psx-spx.consoledev.net/cdromdrive/
[18] Ovladače a paměťové karty [4]: https://psx-spx.consoledev.net/controllersandmemorycards/
[19] List podporovaných ovladačů [4]: https://psx-spx.consoledev.net/controllersandmemorycards/

Ovladač je v emulátoru implementován jako jednoduchý automat, obsahující stavovou proměnou, která udává, ve které části komunikace se ovladač s *BIOSem* nachází. Na základě předem určené výměně bytové sekvence[20] si *CPU* a ovladač vymění synchronizační data, přičemž se na konci této komunikační sekvence předají stavy jednotlivých tlačítek.

4.10 Zvuková procesorová jednotka (SPU)

Zvuková část *PlayStation* systému je relativně nejsložitější hardwarová komponenta. Bohužel, tuto komponentu jsem nebyl schopen přivést do funkčního stavu. I přesto emulátor musí do jisté míry, alespoň částečně, tuto komponentu simulovat.

4.10.1 Zvuková RAM (SRAM)

Zvukový čip není pouze zodpovědný za zpracování vzorkovaného signálu, ale je také schopen vzorky generovat nebo nad tokem vzorků aplikovat speciální efekty. Některé zvukové efekty, jako například ozvěna, ovšem vyžadují paměťový prostor. Pro implementaci ozvěny je nutno si pamatovat předchozí zvukové vzorky, a proto *SPU* má samostatnou paměť *Sound RAM* (*SRAM*). Tato paměť má kapacitu *512KB*, a je zpřístupněna pouze přes registry *SPU*.

Pro zprovoznění některých her, implementace této paměti byla bezpodmínečně nutná, neboť určité přenosy do *SRAM* paměti se provádí přes *DMA*. To znamená, že některé hry čekají na přerušení indikující dokončení přenosu, a pokud toto přerušení nepřijde, logika hry se zasekne.

4.10.2 Voice

SPU má celkem *24 Voice* kanálů. *Voice* v tomto kontextu funguje přibližně jako hudební nástroj v *midi* formátu. Do jednotlivého *Voice* kanálu lze nahrát vzorek hudebního nástroje a poté mu nastavovat výšku tónu. Díky tomu se ušetří nemálo prostoru, protože hudbu lze pak ukládat jako noty spolu s krátkým vzorkem jednotlivých hudebních nástrojů. *Voice* také má schopnost generovat pseudo-náhodný šum, což je užitečné například při generování zvuku bubnů, místo toho aby byl uložen jako tok *ADPCM* vzorků.

4.10.3 ADPCM

Obzvlášť při přehrávání full-motion videa, *Voice* nemusí mít dostatečnou flexibilitu, pro reprodukci dostatečně detailní zvukové stopy. Pro tyto účely *SPU* poskytuje dekódování a dekompresi *Adaptive differential pulse-code modulation (ADPCM)*[21].

[20] Komunikace s ovladačem [4]: https://psx-spx.consoledev.net/controllersandmemorycards/#controllers-communication-sequence

[21] ADPCM: https://en.wikipedia.org/wiki/Adaptive_differential_pulse-code_modulation

Kapitola 5

Uživatelské rozhraní

Vzhledem k tomu, že jádro emulátoru je architektonicky navrženo jako samostatná kompilační jednotka, lze relativně jednoduše zkonstruovat hostovací grafické rozhraní pro systém, který podporuje *C++20* standard, a který je odpovědný za vizualizaci a interakci s interním stavem emulátoru.

Pro tyto účely bylo nutné správně navrhnout celkovou architekturu uživatelského grafického rozhraní tak, aby co nejméně ovlivňovala rychlost emulace jádra *PlayStation* sytému a aby také co nejrychleji dokázala předávat informace o hostovacím vstupu jádru. Jakožto cílové a testovací hostovací systémy jsem zvolil *Windows* a *Linux*.

5.1 Použité technologie

Podobně jako u jádra emulátoru, celková filozofie tohoto modulu stavěla na pilířích *objektově orientovaného přístupu*, zkombinovanou s *RAII* technologií, viz obrázek 3.1. Jelikož kompilace závisí na dvou naprosto odlišných hostovacích systémech, bylo nutné zvolit multiplatformní knihovny, které se starají o správu okna a umožňují snadné zobrazovaní pole pixelů na obrazovku.

Pro samotnou správu okna jsem zvolil knihovnu *SDL2*, která nejenom umožňuje vykreslit obsah *framebufferu* jádra systému, ale dokáže zpracovat události zaslané z operačního systému, včetně práce s klávesovým vstupem a herních ovladačů.

Nicméně knihovna *SDL2* neposkytuje žádné uživatelské grafické rozhraní. To je nucen vývojář napsat sám a tedy bylo nutné hledat nadstavbu nad *SDL2* knihovnou. Ta byla nalezena v knihovně *Dear ImGui*, která nejenom poskytuje uniformní grafické rozhraní, obsahující širokou škálu interaktivních objektů jako menu, tlačítka a vyskakující modální okna, jak můžeme vidět například v aplikaci Ship of Harkinian 5.1[1], ale celková filozofie této knihovny staví na *immediate UI* filozofii, která zásadně zjednodušuje vývoj, prototypování a celkovou správu grafického uživatelského rozhraní.

5.2 Architektura grafického rozhraní

Celková architektura grafického uživatelského rozhraní je rozděleno na 3 části:

[1]Ship of Harkinian github: https://github.com/will-pickett/ship-of-harkinian

Obrázek 5.1: *Dear ImGui* je velmi populární knihovna a poskytuje velmi jednoduchý způsob, jak zkonstruovat *GUI*. V tomto obrázku můžeme vidět použití *Menu* lišty, plovoucího okna, záložky, tlačítka, *Drop-down* menu či přepínačů.

- **App** - objekt spravující okno, vykreslování, události z operačního systému a běh emulátoru.
- **Menu** - objekt spravující menu, tlačítka, *GUI* a celkové ovládání jádra emulátoru.
- **Input** - objekt starající se nejenom o zasílání vstupu z hostovacího systému do jádra emulátoru, ale také zajišťuje přemapování jednotlivých tlačítek systému.

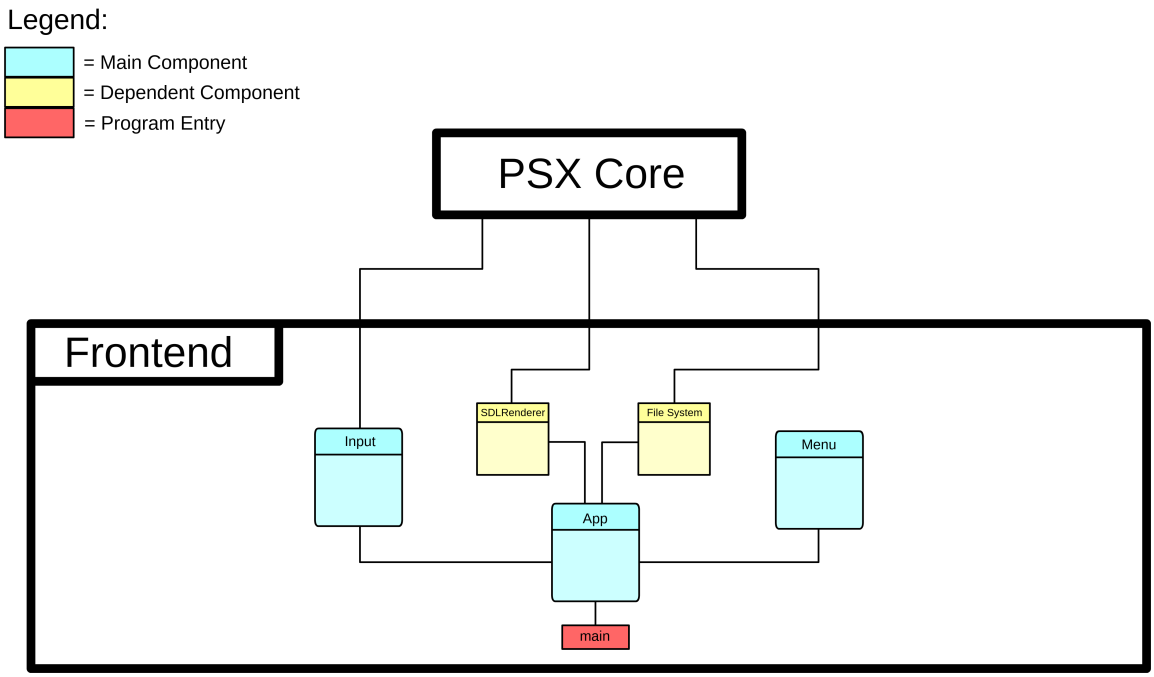
App objekt je implementován jako *singleton* a figuruje jako vstupní bod do programu, viz obrázek 5.2. Detailnější propojení pak můžeme vidět v obrázku 5.3.

5.2.1 Vláknová architektura

Pokud bychom chtěli emulovat *PlayStation* systém v jednom vlákně, bezpochyby by to nebyl zásadní programovací problém. Popis fungování jedno-vláknové architektury je zachycen v obrázku 5.4.

Nicméně narazíme u tohoto návrhu na nepříjemnou věc. Zatímco se například vykresluje *framebuffer*, nebo když se zpracovávají události, či pokud zpracováváme menu, samotná emulace systému stojí. Jelikož je nám každý procesorový takt drahý, je třeba přesunout emulační smyčku do separátního vlákna, jak můžeme vidět v obrázku 5.5.

Aby toto bylo možné, hlavní vlákno a emulační vlákno si musí předávat informace o svém stavu a tudíž je nutné vytvořit komunikační a synchronizační sekvence, ve kterých tyto dvě



Obrázek 5.2: Vizualizace zapojení komponent uživatelského rozhraní. Jádro emulátoru komunikuje s uživatelským rozhraním a vzájemně si předávají zprávy.

vlákna mohou bez problému komunikovat a navzájem se nezpomalovat. Tato dvou-vláknová architektura rozlišuje:

- **Renderovací** vlákno
- **Emulační** vlákno

Emulační vlákno se snaží co nejrychleji simulovat celý systém, dokud systém nevykreslí jeden snímek a nevystoupí do stavu *VBlank*[2]. To je signál pro *Renderovací* vlákno, aby se probudilo a vyžádalo od *Emulačního* vlákna obsah paměti *VRAM* a informace o tom, jak *VRAM* vykreslit (např.: barevná hloubka *VRAM*, ořezávací okno *VRAM*, či zda-li je display zapnut).

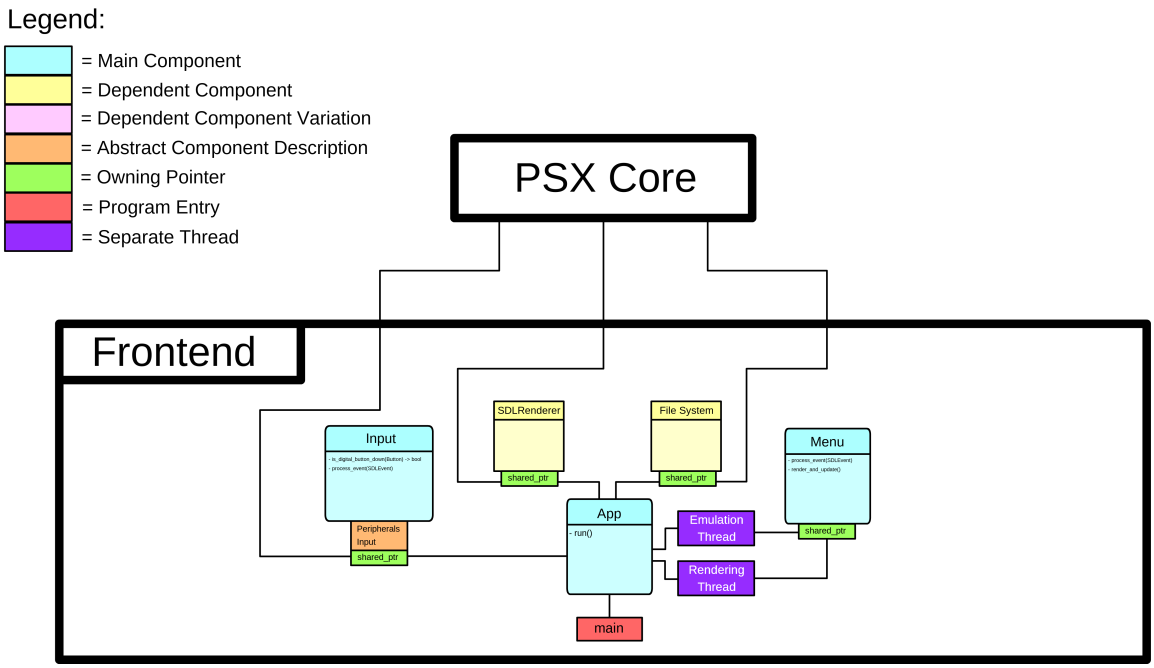
Pro indikaci, zda-li nastal *VBlank* a následné probuzení *Renderovacího* vlákna je použita třída *conditional_variable* ze standardní knihovny *C++*. Pokud probíhá kopírovaní *VRAM* z *Emulačního* vlákna do *Renderovacího* vlákna, celá procedura je ochráněna třídami *mutex* a *scoped_lock*, které zajišťují exkluzivní přístup do kritické sekce.

Technicky vzato, exkluzivní kopírování *VRAM* paměti mezi vlákny není bezpodmínečně nutné, ovšem zaplatíme za to tím, že výsledný obraz bude vykazovat známky *screen tearing*[3], kde část vykreslené *VRAM* je z přechozího snímku a část ze snímku současného, vytvářející artefakt, který může lidské oko rozptylovat.

5.2.2 Tlačítkový vstup

Aby jádro emulátoru dokázalo přijímat vstup z hostovací klávesnice, či herního ovladače, musí nutně vystavit uživatelskému rozhraní rozhraní, které je nutné implementovat na

[2]Vertical blanking interval: https://en.wikipedia.org/wiki/Vertical_blanking_interval
[3]Screen tearing: https://en.wikipedia.org/wiki/Screen_tearing



Obrázek 5.3: Aby uživatelské rozhraní mohlo zasílat informace o tlačítkovém vstupu, musí třída *Input* podědit rozhraní *PeripheralsInput* z jádra emulátoru.

základě hostovacího systému. To znamená, že jádro emulátoru obsahuje čistě virtuální třídu, která je vyžadována při konstrukci emulátoru, a která vyžaduje implementaci funkce *is_digital_button_down(DigitalButton)*, kde detail implementace samozřejmě závisí na hostujícím systému.

Jakmile uvnitř jádra započne *Serial IO* komunikace s herním ovladačem, jádro vyžádá od uživatelského rozhraní nový stav stisknutých kláves, či tlačítek ovladače. Tento přístup by měl minimalizovat artefakt známý jako *input lag*. Moderní systémy pracují na principu zpracování událostí. V kombinaci hodinové komunikace s *USB* periferiemi a navíc samotné zpracování vstupu uvnitř *BIOSu* emulátoru, prodleva, která vznikne touto delegací má nemalou hodnotu. U testovaných her, tato prodleva činila 2-3 snímky.

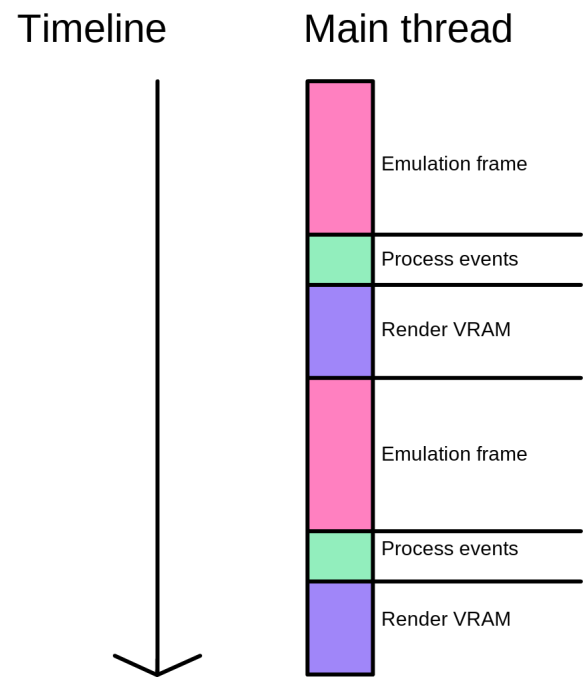
Tento neduh je v emulačních kruzích dobře znám a existují složité procedury, který tento artefakt minimalizují (Například technika Run Ahead[4], implementovaná v emulačním frontendu RetroArch).

5.3 Zobrazení obsahu VRAM

Ačkoliv jsme architekturu výměny obsahu *VRAM* zajistili synchronizačními objekty, samotné zobrazování obsahu *VRAM* není snadná záležitost. Jak bylo zmíněno, *GPU* má 2 módy, jak interpretovat obsah *VRAM* paměti:

- sekvence **15-bitových** barev
- sekvence **24-bitových** barev

[4]RetroArch Run Ahead: https://docs.libretro.com/guides/runahead/



Obrázek 5.4: Jakmile se musí zpracovávat události z operačního systému, nebo se rendruje obsah paměti *VRAM*, emulace stojí.

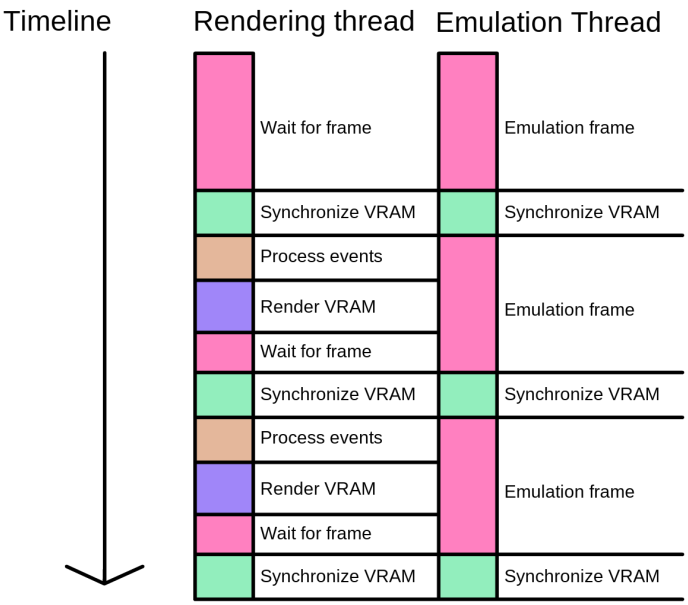
Jelikož v *SDL2* knihovně neexistuje způsob, jak měnit formát textury za běhu programu, je tedy nutno spravovat 2 textury se specifickými formáty, do kterých emulátor kopíruje obsah *VRAM*, a které se vykreslují do okna. Situaci nám taky komplikuje ten fakt, že *GPU* specifikuje ořezávací obdélník, pomocí něhož na obrazovce zobrazuje pouze specifickou část *VRAM* paměti. Tento ořezávací obdélník závisí na interních registrech *GPU* a je netriviální jej vypočítat.

5.4 Grafické uživatelské rozhraní (GUI)

Jak bylo zmíněno, grafická část uživatelského rozhraní je řešena pomocí knihovny *Dear ImGui*, přičemž interakce s vnitřním stavem emulátoru je realizována přes *Menu lištu*, která ve výchozím stavu je přítomna v horní části okna. Přes toto *menu* má uživatel možnost pozměnit a ovládat emulátor, viz obrázek 5.6.

5.4.1 Rozložení GUI

Celkové *GUI* je navrženo pro minimální obstrukci okna. *Menu lišta* má celkem 4 položky:



Obrázek 5.5: I přesto, že dvou-vláknová architektura má určitou režii při komunikaci a výměně dat, emulace již není brzděna správou uživatelského rozhraní.

- **State** - Položka pro reset, načítání a ukládání stavu emulátoru.
- **Controls** - Položka pro mapování hostových tlačítek na tlačítka digitálního ovladače *PlayStation*.
- **Debug** - Položka pro užitečné debugovací funkce jako je vizualizace celé *VRAM* či určení rychlosti emulace.
- **Hide** - Položka pro skrytí *menu lišty*. Skryté *menu* může být obnoveno pomocí klávesy *ESC*.

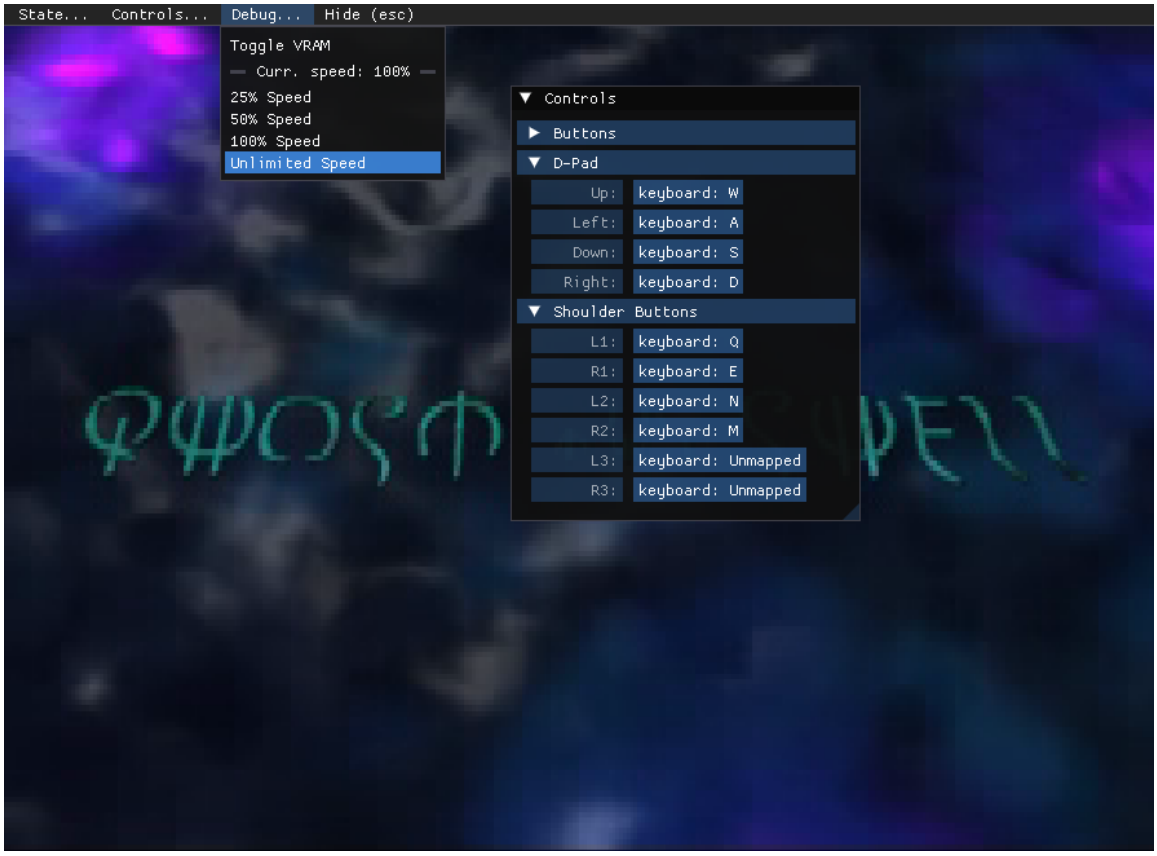
State

Pro ovládání stavu emulátoru, uživatel má několik možností jak s emulátorem interagovat. V prvém případě uživatel může resetovat stav konzole, kde systém spustí inicializační rutinu *BIOSu* a začne se spouštět hra od samého začátku.

Bohužel, v současném stavu emulátor nepodporuje ukládání postupu ve hře pomocí *MemoryCard* periférie. Tento nedostatek je řešen tak, že uživatel má možnost uložit celý stav emulátoru do souboru hostovacího systému. Tím pádem je postup uložen i bez podpory *MemoryCard* periférie. Stejně jako uložení, emulátor podporuje načítání uloženého stavu.

Controls

Emulátor podporuje základní rozložení digitálního *PlayStation* ovladače na klávesnici, přičemž počáteční rozložení mapování tlačítek je podchyceno v tabulce 5.1. Jelikož uživatel může mít specifickou klávesnici, či různé klávesové rozložení, je nutná implementace přemapování tlačítek. Tato odpovědnost je řešena právě ve třídě *Input*, kde se používá *unordered_map* ze standardní knihovny *C++* pro správu jednotlivých přemapování.



Obrázek 5.6: *GUI* emulátoru

U každého mapování uživatel může kliknout na specifickou klávesu kterou chce přemapovat. To spustí modální okno které odposlouchává klávesový vstup, přičemž první stisknutá klávesa po spuštění tohoto okna se bere jako mapovací klávesa.

Debug

Aby člověk mohl pohlédnout za oponu a měl větší kontrolu nad chodem emulátoru, v menu existuje *Debug* položka, ve které si uživatel může zobrazit obsah celé *VRAM* a má tak možnost vidět co se děje mimo vykreslený *framebuffer*. Stejně tak může uživatel nastavit rychlost chodu celého emulátoru a nastavovat změnu rozlišení.

5.4.2 Prevence dark patterns

Ač chtěně či nechtěně, může se stát že *GUI* je architektonicky navržena tak, že části rozhraní dokáží svést uživatele k akcím které provést vůbec nechtěl (tzv.: *dark patterns/anti patterns*[5]). Když bylo *GUI* tohoto emulátoru testováno, ukázalo se, že uživatel při ukládání stavu emulátoru mohl omylem kliknout na tlačítko *Reset*, které bez upozornění systém obnovilo do původního stavu, přičemž vymazal jakýkoliv postup ve hře, který se uživatel zrovna snažil uložit.

[5]Dark pattern: https://en.wikipedia.org/wiki/Dark_pattern
Anti-pattern: https://en.wikipedia.org/wiki/Anti-pattern

Tabulka 5.1: Základní mapa tlačítek

PlayStation ovladač	Klávesnice
Select	L
Start	K
D-pad Up	W
D-pad Right	D
D-pad down	S
D-pad left	A
L2	N
R2	M
L1	Q
R1	E
Triangle	Arrow Up
Circle	Arrow Right
Cross	Arrow Down
Square	Arrow Left

Z tohoto důvodu je třeba řešit prevence a záchranné sítě, které tomuto neduhu předejdou. V tomto případě *Reset* tlačítko bylo odděleno oddělovací čárou, klávesová zkratka pro reset systému byla odstraněna a při kliknutí na tlačítko *Reset* se zobrazí modální okno, které se ptá na konfirmaci uživatelské akce, viz obrázek 5.7.

Obrázek 5.7: Ačkoliv *Reset* tlačítko je nebezpečně blízko *Save State* a *Load State* tlačítek, po jeho stisknutí vyskočí konfirmační okno, které jakýkoliv překlik neguje.

5.5 Spuštění programu

Emulátor očekává své spuštění přes příkazovou řádku, kde mu uživatel musí specifikovat externí data, se kterými pak emulátor dokáže pracovat. Pro minimální chod emulátoru je nutné předat cestu k *BIOS* souboru v hostovacím systému jakožto první argument programu. Bez *BIOSu* nebude emulátor schopen fungovat. Emulátor byl testován na dvou verzích *PlayStation BIOSu*:

- (md5sum: *924e392ed05558ffdb115408c263dccf*) *PSX BIOS* verze *SCPH1001*
- (md5sum: *c53ca5908936d412331790f4426c6c33*) modifikovaný *PSX BIOS* pro *PSP*

Bohužel, jedná se o licencovaný software, a tedy *BIOS* nemůže být distribuován spolu s emulátorem. Pro získání *BIOSu* musí člověk vlastnit originální konzoli a *BIOS* z ní extrahovat. Tudíž ani stránky jako *google*, či *archive.org* nepomůžou.

Ačkoliv emulátor pouze s *BIOSem* poběží, je nutné specifikovat programu cestu k obrazu hry v hostovacím systému, pro získání plného využití tohoto softwaru. V současné době, tento emulátor podporuje pouze bitové kopie sektorů disku s jednou stopou. Nicméně některé hry obsahovaly na disku více stop (ve většině případů se oddělovaly datové stopy a audio stopy). Tyto typy her nejsou podporovány a ačkoliv lze načíst pouze datová stopa více-stopového disku, může se stát že hra vyžádá data z jiných stop a tedy ukončí chod emulátoru. Cestu k bitové kopii jedno-stopového *PlayStation* disku je nutno specifikovat jako druhý argument programu.

Po specifikaci argumentů emulátor automaticky načte *BIOS* i *Disk* a okamžitě začne systém emulovat.

Kapitola 6

BIOS

6.1 Odrazový bod

Aby člověk byl schopen vytvořit emulátor jakéhokoliv systému, je vždy potřeba mít určitý odrazový bod, od kterého lze emulátor začít implementovat a který současně slouží i jako test korektnosti jeho fungování. Ve většině případů jde o zaváděcí program, který inicializuje daný systém. V případě *PlayStationu* se jedná o *Basic Input/Output System (BIOS)*, který je zabudován do speciální *ROM* paměti každé *PlayStation* konzole. Existuje několik verzí *BIOSu*. Hlavní dělení probíhá podle regionálních/verzích základních desek, kde se rozlišují tři hlavní verze *BIOSu*: *Americký (SCEA)*, *Japonský (SCEI)* a *Evropský (SCEE)*. *Sony* také vytvořilo modifikované verze *BIOSů*, které používalo v konzolích dalších generací pro hladší emulaci *PlayStation* systému (například systém PlayStation Portable obsahuje upravený *BIOS*, který přeskakuje úvodní bootovací animaci).

Tento *BIOS* lze využít pro strukturovanou implementaci celého emulátoru, kde je nejprve vytvořena paměťová struktura emulátoru podle paměťové mapy a poté jsou postupně spouštěny instrukce *BIOSu*. Jakmile emulátor narazí na instrukci, která zahrnuje funkcionalitu či přístup do ještě neimplementované hardwarové komponenty, emulátor signalizuje chybu a chybějící funkcionalitu je třeba buď plně implementovat, nebo ji utlumit, v závislosti na její závažnosti a schopnosti systému fungovat bez ní.

6.2 Funkce

Ačkoliv se tomuto zaváděcímu programu v *PlayStation* komunitě říká *BIOS*, jedná se v podstatě o velmi odlehčený operační systém, který poskytuje systémová volání pro usnadněný přístup k hardwaru.

Uživatel mohl spustit *PlayStation* konzoli i bez vložené hry do *CD-ROM* čtečky a byl uvítán úvodní obrazovkou *BIOSu*, tj. *shellem*. V tomto menu měl uživatel možnost spravovat, kopírovat a mazat uložený postup her, pokud byla do konzole připojena speciální paměťová karta. Uživatel také mohl v tomto menu přehrávat *Audio CD*.

Hlavní funkcionalitu však *BIOS* poskytuje prostřednictvím speciálních instrukcí systémových volání nebo speciálních tabulek rutin umístěných na začátku paměti *RAM*. Skrze toto *API*, *BIOS* nabízí řadu funkcí, jako jsou například přístup k souborovému systému

CD-ROM, správa paměti, výpis ladění, ale také zpracování výjimek, manipulace s řetězci, přístup k *GPU*, *SPU*, *GTE* a mnoho dalších[1].

6.3 Fáze BIOSu

6.3.1 Zaváděcí fáze

Ačkoliv se jednotlivé verze *BIOSu* liší ve své funkcionalitě, všechny následují velmi podobný zaváděcí proces. Při zapnutí/resetu konzole se *Program Counter* procesoru nastaví na hodnotu **0xBFC0'0000**, což je začátek adresy, kde je uložen *BIOS*. V tomto bodě je uložena resetovací logika. V prvé řadě jsou registry *CPU* vyčištěny a poté začne inicializovat hardware.

Nejdříve se nastaví registry *paměťového ovladače*. Tento kus hardwaru je diskutabilně do jisté míry zbytkem z reálného počítače, obsahující informace o velikostech jednotlivých pamětí (*RAM*, *BIOS*, *Scratchpad*, ...).

Dále se odizoluje *vyrovnávací paměť instrukcí*, což způsobí, že všechny zápisy se převedou místo na sběrnici do této *vyrovnávací paměti*. *BIOS* poté tuto paměť vyčistí nulami.

V další fázi *BIOS* přistupuje k *ovladači vyrovnávací paměti* a zapne instrukční a datovou vyrovnávací paměť. Tato komponenta slouží jako globální vypínač jednotlivých vyrovnávacích pamětí (*d-cache, i-cache, scratchpad*), ovšem podobně jako *paměťový ovladač*, jde spíše o formalitu, neboť do tohoto ovladače se dále už nepřistupuje.

BIOS pak zresetuje **koprocesor 0** (ovladač výjimek) tak, že všechny jeho registry nastaví na nulu a utlumí všechny kanály *SPU*.

Pokud *BIOS* běžel na vývojářské verzi konzole, programátor mohl využít *Expansion Port (PIO)*, což do jisté míry je *GPIO* použitelné pro debugování vývoje hry. Aby *PIO* mohlo být použito, zařízení na druhé straně muselo zaslat speciální ASCII řetězec: *"Licensed by Sony Computer Entertainment Inc."* kvůli verifikaci.

6.3.2 Jádro BIOSu

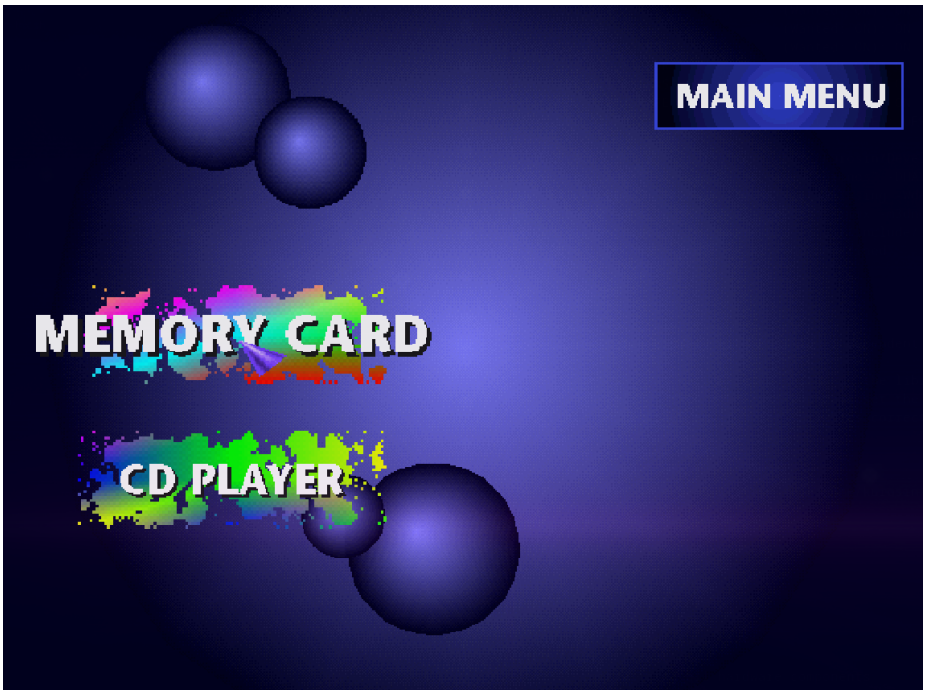
Po inicializačním resetu hardwaru je do paměti zkopírován obraz jádra a spuštěn. Jádro zpřístupní svou funkcionalitu tím, že vyplní speciální rutinní tabulky na začátku paměti *RAM*, spustí *shell* a zobrazí úvodní animaci, jak můžeme vidět v obrázku 6.1.

Po dokončení této animace je zkontrolována *CD-ROM* mechanika, zda-li je její poklop uzavřen a zda-li mechanika obsahuje *CD*. Pokud jsou tyto podmínky splněny, *BIOS* začne analyzovat souborový systém *CD* a hledat hlavní spustitelný soubor. Pokud je přítomen, je posléze spuštěn.

Pokud v *CD-ROM* mechanice nic není, *BIOS* spustí *shell* (viz obrázek 6.2) a dále se již s ničím nadále nezabývá.

[1] PCSX ReARMed HLE BIOS: https://github.com/notaz/pcsx_rearmed/blob/master/libpcsxcore/psxbios.c

Obrázek 6.1: Úvodní obrazovka indikující správnou inicializaci systému.



Obrázek 6.2: Úspěšné nastartování *BIOS shellu*.

Kapitola 7

Testování

Emulátor jakéhokoliv systému který je implementován na základě kusých informací a reverzním inženýrství bude vždy mít určité nedostatky, nepřesnosti a chyby. Ovšem cílem emulátoru není precizně napodobit chování emulovaného systému, nýbrž získat *drop-in replacement* za původní systém, kde v případě *PlayStation* konzole znamená kompatibilitu a hratelnost oficiálně publikovaných her.

Bohužel, přístup k celé *PlayStation* knihovně není pro jednotlivce dost dobře možný. Ty hry, které byly testovány na tomto emulátoru byly zapůjčeny od blízkých přátel. Samozřejmě testování jednotlivých her je velmi časově náročná záležitost, neboť na *PlayStation* systém vycházely hry, které vyžadovaly až stovky hodin investice.

7.1 Kompatibilita dostupných her

Z celkem 8 testovaných her, 5 z nich bylo *hratelných*. To znamená, že hráč se dostal přes úvodní intra do hlavní herní smyčky a měl možnost ovládat to, co se děje na obrazovce (to ovšem neznamená, že hra běžela perfektně, ani že byla dohratelná). Zbytek, který *hratelný* nebyl většinou závisel na specifických nedokončených implementací jednotlivých hardwarových komponent (především *CDROM*, *SPU* a *GTE*). Pro demonstraci jsem zvolil 3 různé dostupné hry, které byly *hratelné*.

7.1.1 Skullmonkeys

Tato hra, při testování byla *hratelná* od začátku až do konce. Jde o zástupce 2D *PlayStation* her a tedy testovala schopnosti emulátoru správně vykreslovat texturované a obarvené obdélníky, viz obrázky 7.2 a 7.7. Hra ovšem neběžela bez problémů. Její koncept je budován na základě skrolovaných úrovní a v určitých částech hra načítala stavební bloky úrovně příliš pozdě, jak můžeme vidět v obrázku 7.3. Co tento nedostatek způsobilo není známé, ovšem s nejvyšší pravděpodobností jde o špatně načasované přerušení, nebo hře je poskytována špatná hodnota ohledně pozice kamery (možná špatná znaménková interpretace registru v *GPU* nebo *GTE*).

7.1.2 Final Fantasy VII

Tato hra byla bohužel příliš dlouhá[1] na otestování všech kapabilit, nicméně zhruba 2-hodinová seance běžela bez větších problémů. Tato hra kombinuje 3D modely s 2D pozadím

[1] https://howlongtobeat.com/game/3521

dekódované pomocí čipu *MDEC* vytvářející iluzi 3D scény. Výjimkou jsou bitevní scény, které plně využívají 3D schopností *PlayStation* systému, viz obrázky 7.4 a 7.8. Ukázal se ovšem minoritní problém při vykreslování polygonů, kde tato hra při vykreslování 2D pozadí využívá zelenou masku, pomocí níž filtruje 3D modely, aby se správně scéna zobrazila. Tato zelená maska by teoreticky neměla být nikdy vidět, ovšem může se stát že se sem tam nějaký ten pixel nepřesně vykreslí, jak si můžeme všimnout v obrázku 7.6. Tento nedostatek může být způsoben špatnou interpolací atributů vrcholů trojúhelníku, nebo je špatně logika *fixed-point* aritmetiky kterou *GPU* rasterizace používá.

7.1.3 Ghost in the Shell

Tato hra používá širokou škálu *GTE* příkazů a tedy řádně tuto hardwarovou komponentu otestovala. Až na výjimečné chyby v texturování trojúhelníků a občasné zaseknutí emulátoru, tato hra běžela relativně dobře, viz obrázky 7.5 a 7.9.

7.2 Měření/Potenciální optimalizace

Bezpochyby přesnost emulace může mít zásadní vliv na hratelnost jednotlivých her. Ovšem pokud se emulace vleče, celkový požitek ze hry může být tímto faktorem narušen. Měření rychlosti bylo provedeno jako násobek odchylky od ideálního běhu emulátoru. To znamená, že pokud konzole vyžadovala *50* snímků za sekundu, ale emulátor při plné rychlosti dokázal simulovat *100* snímků za sekundu, výsledek měření by byl *2*, protože emulátor dokáže simulovat systém s dvojnásobnou rychlostí. Výsledek měření je zachycen v grafu 7.1.

Vzhledem k tomu, že celková architektura tohoto emulátoru byla navržena spíše pro modularitu a celkovou čitelnost zdrojového kódu, rychlostně tento emulátor poněkud tápe. Ovšem existují hlavní místa, ve kterých by se dalo získat nemálo procesorových cyklů popsané v sekcích [7.2.1] a [7.2.2].

7.2.1 Potenciální optimalizace CPU

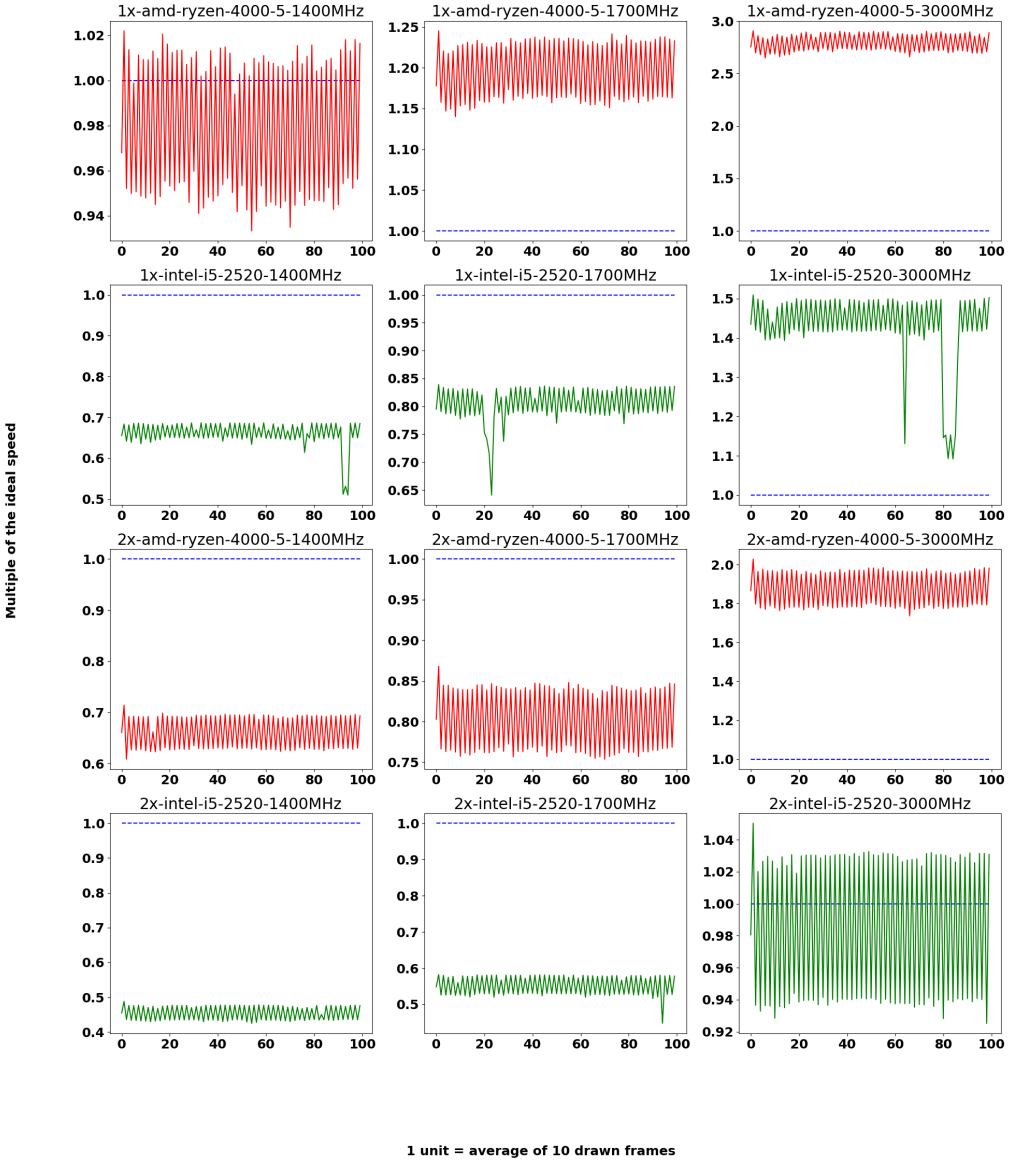
CPU modul v současné době je řešen čistě jako interpretace proudu *MIPS* instrukcí. Ovšem existuje rychlejší a mnohem složitější varianta jak transformovat jednu instrukční architekturu na architekturu jinou. Tím samozřejmě myslím koncept *Just In Time (JIT)* kompilace, či *rekompilátor*, který by za běhu dokázal identifikovat bloky instrukcí a transformovat je na nativní sekvenci instrukcí hostovacího systému. V našem případě by se jednalo o *MIPS R3000A* na *x86_64 rekompilátor*. Výsledný transformovaný blok by se pak nativně zavolal a tím by se dalo získat nemálo procesorových taktů.

Ačkoliv jednoduchý *JIT* modul by nebylo příliš složité na implementaci, tento koncept vyžaduje hlubokou znalost obou instrukčních sad a celkově se jedná o velmi časově náročnou záležitost. *JIT* taky znamená poteciální risk co se bezpečnosti týče, protože *JIT* vyžaduje vytváření spustitelného bloku v paměti za běhu, který přímo byte po bytu modifikujeme.

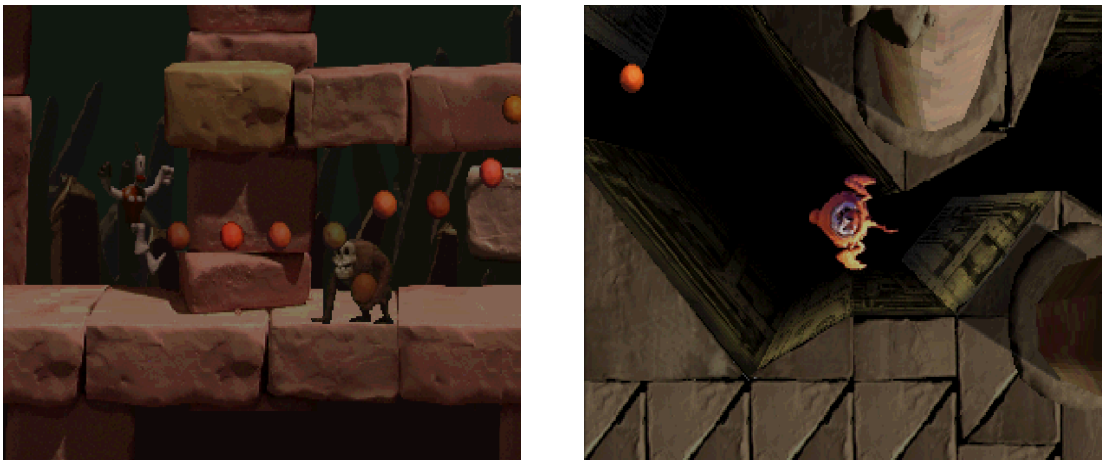
7.2.2 Potenciální optimalizace GPU

V *GPU* probíhá veškerá rasterizace softwarově pomocí hostovacího *CPU*. Potenciální zrychlení by se dalo nalézt v hardwarové rasterizaci pritiv. Je ovšem na pováženou jak moc by hardwarová rasterizace v základním režimu pomohla, protože většina 3D scén, které se *PlayStation* hry snaží vykreslit, jsou vesměs velmi malé trojúhelníky a jejich počet se pohybuje kolem spodních tisíců (nejnáročnější z testovaných her byla *Ghost in the Shell*, kde za běhu

hry bylo naměřeno až 3000 vykreslených trojúhelníků za jeden snímek). S takto malou pracovní jednotkou není jisté, zda-li samotná práce s hostovacím *GPU* (především přenos dat z a do hostovacího *GPU*) by nezabíralo více času, než softwarové renderování. Tato spekulace samozřejmě padá při renderování ve vyšším rozlišení, kde hardwarový renderer by naprosto jistě zrychlil celý chod emulátoru.



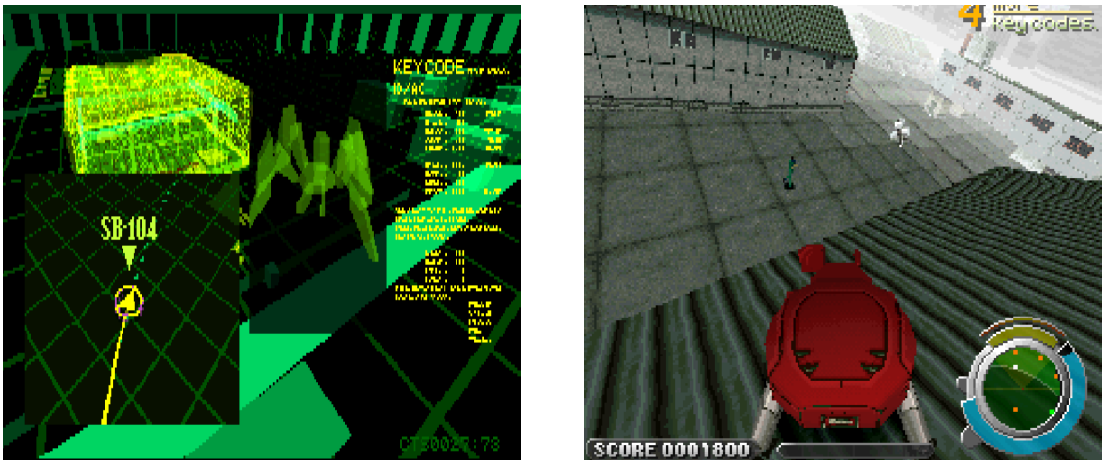
Obrázek 7.1: Rychlost emulace především závisí na schopnostech hostovacího procesoru. Testovány byly dva různé procesory (**AMD 4000 5 - červený graf** a **intel i5 2520 - zelený graf**). V každém z grafu je také přerušovaná **ideální čára**, která slouží jako referenční bod ideálního běhu emulátoru. Pokud graf je nad touto přerušovanou čárou, znamená to, že hostovací procesor dokáže *PlayStation* systém simulovat rychleji. Stejně tak pokud graf je pod přerušovanou čárou, emulátor nestíhá simulovat systém dostatečně rychle. Graf tedy zobrazuje násobek ideální rychlosti vzhledem k indexu naměřeného snímku (horizontální osa), přičemž prefix názvu grafu určuje, při jakém násobku rozlišení bylo měření provedeno (**1x** a **2x**). Oba procesory byly postupně omezeny vzhledem k jejich hodinové rychlosti. U tohoto porovnání procesorů si můžeme všimnout, že čistě rychlost hodin není všechno. **AMD 4000 5** disponuje *3MiB L2 cache* a *8 MiB L3 cache*, kdežto **intel i5 2520** má pouze *512 KiB L2 cache* a *3 MiB L3 cache*. Naměřená data se nacházejí v přílohách A.1 a A.4



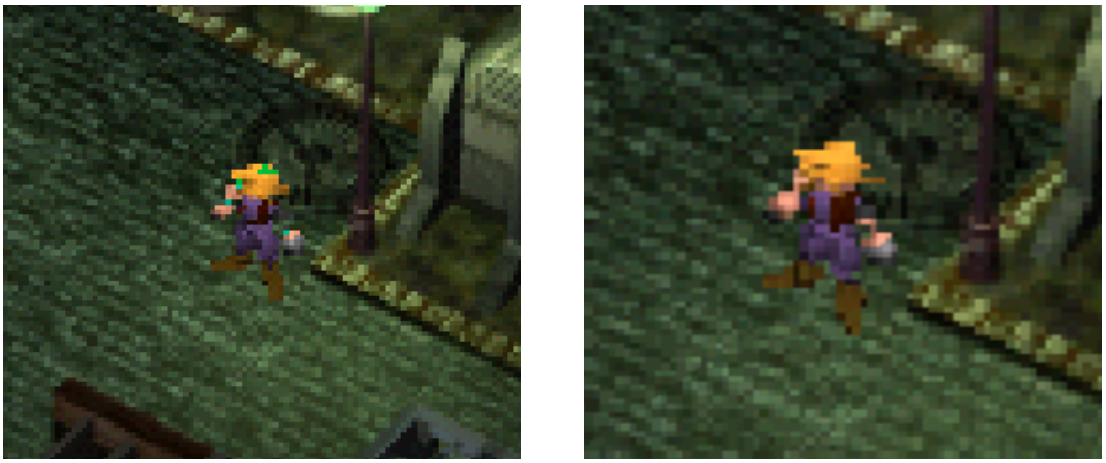
Obrázek 7.2: Ačkoliv *Skullmonkeys* je dominantně 2D skákačka, hra obsahuje určité úrovně, v nichž utilizuje 3D schopnosti *PlayStation* systému.

Obrázek 7.3: Vlevo si můžeme všimnou správně vykreslené úrovně (emulátor *SwanStation*) a napravo je současný stav mého emulátoru, kde chybí části úrovně.

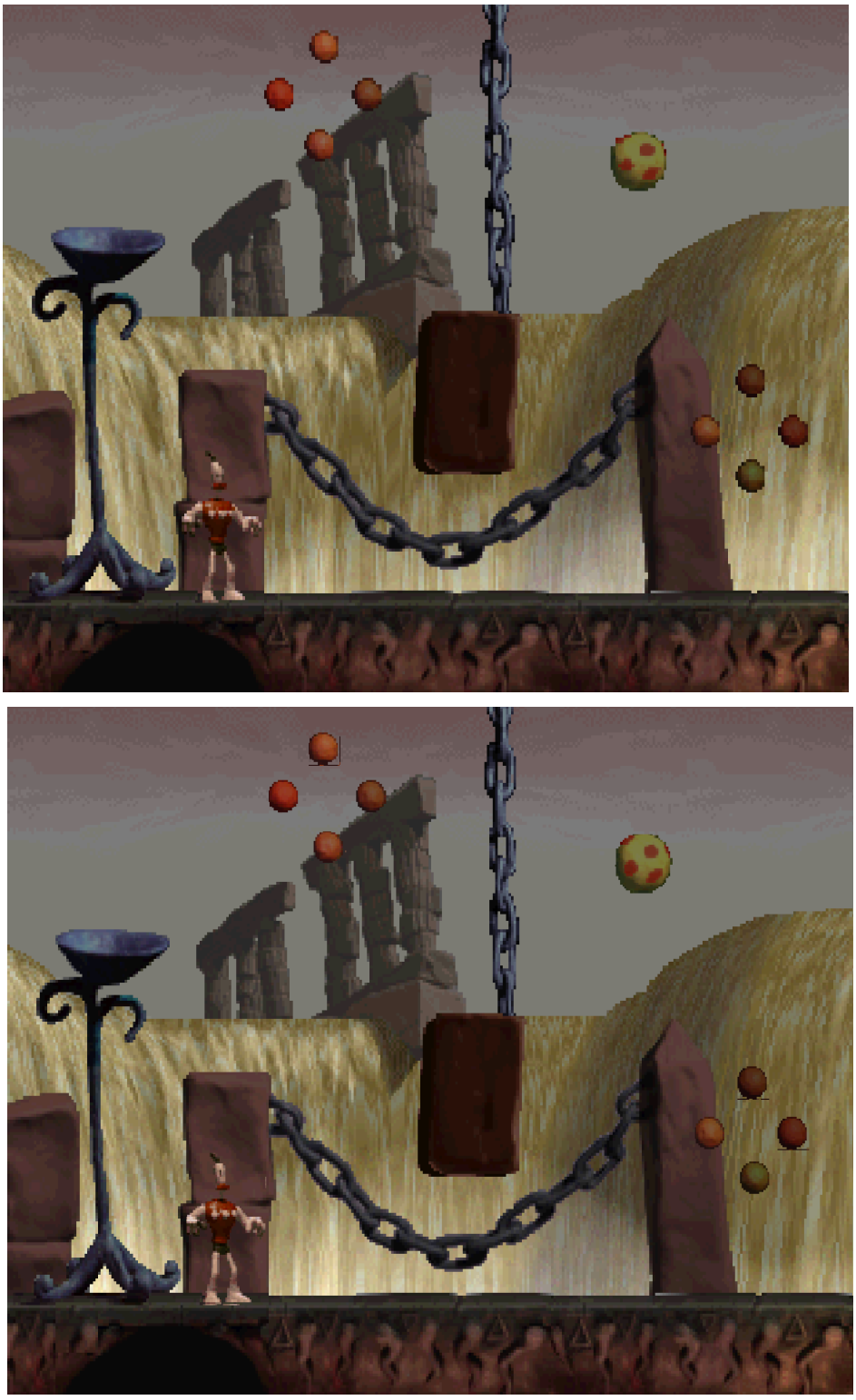
Obrázek 7.4: Vlevo si můžeme všimnou kombinace předem vyrendrovaného 2D pozadí, které v kombinaci s 3D modely dokázaly vytvořit solidní iluzi 3D prostředí. Vpravo pak vidíme plně 3D vykreslenou dynamickou scénu.



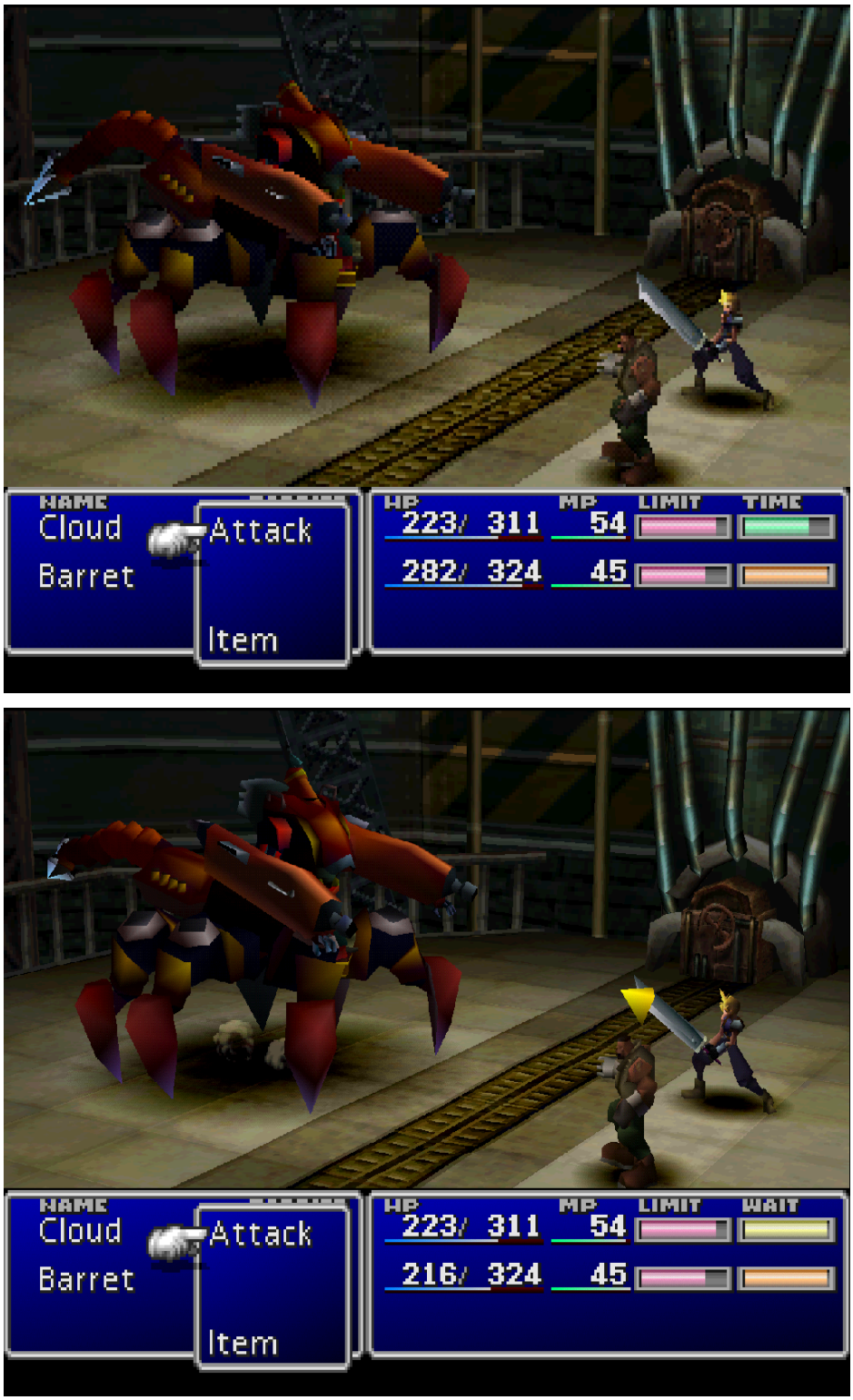
Obrázek 7.5: Briefing mise, který vidíme v levém obrázku, vyžaduje dobré pokrytí většiny příkazů *GTE* komponenty. Stejně tak hlavní herní smyčka, v obrázku v pravo, provádí netriviální transformace scény.



Obrázek 7.6: Z neznámeho důvodu, příliš malé trojúhelníky se v mém emulátoru nevykreslují korektně, nebo jde o chybu při interpolaci barev vrcholů. Tento artefakt je patrný v levém obrázku, kde zelená maska, použitá pro správné vykreslení 2D pozadí, se "propíjí"do 3D modelů.



Obrázek 7.7: Při zvýšení rozlišení se textury úrovně příliš nezmění, ale můžeme si povšimnout změny vykreslení dynamických objektů. V této scéně jsou objekty, jako například hlavní postavička, zmenšeny. Pokud ale zvýšíme rozlišení, dynamické objekty získají zpět svůj detail.



Obrázek 7.8: Ačkoliv při změně rozlišení textury menu zůstávají stále hranaté, samotná scéna získává úplně nový nádech.

Obrázek 7.9: Při změně rozlišení se stává, že obdélníková primitiva mají malý problém s adresováním textur. Radar v pravém dolním rohu obrazovky obsahuje viditelné *jizvy*. Můžeme si také všimnout, že změna rozlišení nám snadněji dovolí rozpoznávat vzdálené objekty, jako například číslo 3 na dveřích budovy v levé části obrázku.

Kapitola 8

Závěr

Ačkoliv tento emulátor nemá ani vzdáleně 100% kompatibilitu s celou *PlayStation* knihovnou, rozhodně jsem rád za to, že se mi podařilo dostat emulátor do stavu, kde nejenom tento software je schopen nastartovat originální *PlayStation* hry, ale pár z nich bylo i relativně hratelných. V tomto projektu šlo hlavně o to prozkoumat komerční a velmi populární systém a pokusit se zpříjemnit jeho používání pro komunitu herních nadšenců milující staré hry, ať už kvůli zvědavosti, nebo kvůli nostalgii.

Implementace takto složitého systému vyžaduje přehlédnutí určitých specifických chování hardwaru, přeskočení logiky fungování komponentu, nebo dokonce přeskočení celistvé komponenty. Čili nebudu zastírat, že tu a tam chybí cihla či trám. Tento nedostatek je způsoben tím, že mohu čerpat pouze z dokumentů, které jsou výsledkem mnohaleté práce reverzního inženýrství mnoha lidí a ačkoliv se jedná o monumentální úsilí, nakonec nikdo (krom *Sony*) nemá přístup k celému originálnímu hardwarovému návrhu *PlayStation* konzole.

Následující výčet popisuje, jak schopné jsou hlavní hardwarové komponenty tohoto emulátoru a poskytuje hrubý procentuální odhad dokončení:

- **CPU 91%** - Chybí řádné zpracování výjimek a CPU nepodporuje hardwarové breakpointy
- **GPU 80%** - Ačkoliv samotná funkcionalita je plně pokryta (kromě kreslení polyčar), *GPU* má spoustu chyb a nepřesností (fixed-point aritmetika, či nepřesná rasterizace polygonu).
- **MDEC 65%** - *MDEC* v emulátoru podporuje pouze 16-bitovou a 24-bitovou barevnou hloubku, ale originální hardware umožňuje dekódovat 4-bitovou a 8-bitovou barevnou hloubku (*grayscale*).
- **CDROM 42%** - Z celkových **38** příkazů, **20** z nich je implementováno a na zbylých **18** emulátor padá. *CDROM* také dokáže pouze načítat disky s 1 stopou.
- **GTE 70%** - Ze **23** příkazů, **17** z nich funguje. Nicméně je to komponenta, ze které pramenilo nejvíce chyb a implementačních problémů.
- **DMA 80%** - Přenosy mezi komponentami fungují, ale synchronizace je oproti reálnému hardwaru velmi zjednodušená.
- **Časovače 95%** - U časovače typu *DotClock* je jeho rychlost aproximovaná, jinak by vše mělo fungovat.

- **Periférie 50%** - Simulace digitálního ovladače funguje, ale emulátor nemá podporu pro paměťové karty pro uložení postupu ve hře.
- **SPU 5%** - *SPU* implementuje pouze svůj hardwarový interface aby ostatní komponenty s ním mohly komunikovat, ale interně se v něm neděje prakticky nic.

Samotná změna rozlišení, až na pár artefaktů ohledně adresování textur, funguje relativně dobře. U změny rozlišení je hlavní problém ten, že zásadně zpomaluje chod emulátoru jak bylo popsáno v naměřených datech 7.1. Nejrychlejší z testovaných strojů zvládal s rezervou emulovat systém s **2x** násobkem rozlišení, ale při **4x** násobku již nebyl schopen dosáhnout požadované rychlosti. Jak bylo zmíněno v sekci [7.2.2], řešením tohoto zpomalení by bylo přesunout vykreslování na hostovací *GPU*.

Literatura

[1] *Sony Computer Entertainment v. Connectix Corp., 203 F.3d 596 (9th Cir. 2000)* [online]. 2000. Dostupné z: https://casetext.com/case/sony-computer-entertainment-v-connectix-corp-2.

[2] *Nintendo PlayStation prototype - PlayStation Wikipedia* [online]. 2020. Dostupné z: https://playstation.fandom.com/wiki/Nintendo_PlayStation.

[3] Integrated Device Technology, I. *IDT R30xx Family Software Reference Manual* [online]. 1994. Dostupné z: https://hitmen.c02.at/files/docs/psx/3467.pdf.

[4] Korth, M. 'nocash'. *PlayStation Specifications* [online]. 2022. Dostupné z: https://psx-spx.consoledev.net/.

[5] newzoo.com. *Key Numbers, Archived from https://newzoo.com/key-numbers/* [online]. 2019. Dostupné z: https://web.archive.org/web/20190509014637/https://newzoo.com/key-numbers/.

[6] Pineda, J. *A Parallel Algorithm for Polygon Rasterization* [online]. Apollo Computer Inc., 1988. Dostupné z: https://www.cs.drexel.edu/~deb39/Classes/Papers/comp175-06-pineda.pdf.

[7] Price, C. *MIPS IV Instruction Set* [online]. 1995. Dostupné z: https://www.cs.cmu.edu/afs/cs/academic/class/15740-f97/public/doc/mips-isa.pdf.

[8] Salvador, P. *Survey of the Video Game Reissue Market in the United States (1.1)* [online]. 2023. Dostupné z: https://doi.org/10.5281/zenodo.8161056.

Příloha A

Naměřená výkonnostní data

Tabulka A.1: **AMD Ryzen 4000 5**. Procesor byl omezen na 3 hodinové rychlosti (sloupce), přičemž jednotlivé řádky reprezentují průměrný násobek ideální rychlosti 100 naměřených snímků (tedy bylo naměřeno celkem 1000 konzistentních snímku specifické hry a okno po 10 snímcích bylo zprůměrováno). Pokračování tabulky A.2

1x/1.4GHz	**2x/1.4GHz**	**1x/1.7GHz**	**2x/1.7MHz**	**1x/3GHz**	**2x/3MHz**
0.967953	0.660651	1.177842	0.802855	2.753748	1.866711
1.021956	0.714269	1.245324	0.867937	2.907587	2.028165
0.952063	0.608470	1.157768	0.766193	2.698310	1.796814
1.013590	0.691830	1.220589	0.844670	2.861357	1.964868
0.949956	0.626657	1.146902	0.763004	2.679373	1.776648
0.998788	0.692615	1.217571	0.844600	2.841532	1.976229
0.950634	0.626210	1.149305	0.764787	2.649055	1.770648
1.011210	0.691721	1.220672	0.841403	2.829500	1.968275
0.948727	0.627635	1.140178	0.762465	2.685867	1.787347
1.009227	0.691611	1.227207	0.840232	2.852648	1.968768
0.947913	0.624947	1.153232	0.757096	2.686585	1.776579
1.009133	0.691848	1.228513	0.839240	2.872848	1.962725
0.949992	0.623523	1.154841	0.761236	2.660109	1.763244
1.012998	0.662046	1.231141	0.839299	2.858463	1.974491
0.944989	0.622910	1.147994	0.758766	2.675943	1.771073
1.009707	0.692042	1.228346	0.839692	2.846368	1.966736
0.948496	0.625481	1.150522	0.762029	2.680474	1.780018
1.020564	0.698591	1.233510	0.845116	2.875341	1.974169
0.955350	0.630879	1.159687	0.765969	2.705843	1.781338
1.016089	0.692792	1.229790	0.845268	2.861546	1.967983
0.953212	0.626627	1.157796	0.761806	2.718002	1.780613
1.012300	0.692556	1.225741	0.838726	2.885216	1.950384
0.951137	0.626784	1.158032	0.761489	2.713928	1.766997
1.013305	0.692123	1.225749	0.846733	2.895792	1.964867
0.954652	0.628600	1.164545	0.762959	2.726558	1.779830
1.013413	0.691397	1.230657	0.843462	2.875053	1.956245
0.955127	0.627639	1.163608	0.767903	2.730391	1.781067
1.007080	0.690659	1.231025	0.842135	2.845124	1.948782
0.945962	0.627589	1.156600	0.763230	2.707818	1.768893
1.010141	0.691136	1.225642	0.840501	2.891845	1.967905
0.959790	0.634844	1.173308	0.772402	2.739660	1.789174
1.012774	0.694421	1.235905	0.840162	2.890630	1.958250
0.941036	0.629470	1.160307	0.756637	2.723393	1.785323
1.001889	0.693560	1.232101	0.841870	2.891902	1.962635
0.943416	0.628478	1.165127	0.763130	2.708328	1.776906
1.003956	0.694900	1.230676	0.838805	2.885710	1.960541
0.948227	0.628819	1.162590	0.762819	2.735948	1.779839
1.013143	0.693586	1.235372	0.841782	2.904402	1.974533
0.946525	0.629481	1.168492	0.765998	2.738970	1.782431
1.006079	0.693265	1.235775	0.838810	2.895302	1.959135
0.948796	0.630495	1.162529	0.768382	2.735157	1.780691

Tabulka A.2: **AMD Ryzen 4000 5**. Pokračování tabulky A.3

1x/1.4GHz	2x/1.4GHz	1x/1.7GHz	2x/1.7MHz	1x/3GHz	2x/3MHz
1.014328	0.693983	1.237621	0.847253	2.892178	1.972136
0.953787	0.628826	1.164285	0.762520	2.732571	1.783905
1.014806	0.696550	1.234170	0.846648	2.868323	1.970332
0.956611	0.630253	1.166207	0.769802	2.725551	1.782815
1.012033	0.694975	1.237662	0.844254	2.898626	1.972057
0.950294	0.632380	1.164547	0.756961	2.731532	1.779316
0.993995	0.695041	1.233714	0.843916	2.894082	1.970537
0.941791	0.629128	1.157075	0.764911	2.728946	1.797333
1.002849	0.694999	1.235865	0.840646	2.891623	1.982617
0.953391	0.629880	1.163253	0.761694	2.722710	1.803316
1.007728	0.693358	1.236065	0.834506	2.902842	1.982162
0.942626	0.630388	1.163983	0.762922	2.730648	1.793742
1.002000	0.695992	1.232990	0.840201	2.891229	1.976936
0.933302	0.628536	1.163409	0.770808	2.736006	1.785235
1.009791	0.695957	1.236435	0.848073	2.899871	1.984652
0.941929	0.630641	1.167811	0.769895	2.734105	1.784323
1.007856	0.686512	1.235751	0.840594	2.900186	1.966735
0.953287	0.626911	1.163943	0.761610	2.726532	1.782491
1.010908	0.695153	1.234478	0.845754	2.850332	1.969029
0.944254	0.627254	1.156130	0.762630	2.711149	1.775234
1.008185	0.694801	1.232340	0.841005	2.881186	1.963256
0.946039	0.627757	1.158614	0.760605	2.703842	1.783541
1.007541	0.693260	1.228071	0.839275	2.880050	1.967225
0.944738	0.626835	1.154384	0.757040	2.697456	1.774669
1.006160	0.688294	1.232170	0.834484	2.872949	1.963367
0.943533	0.623836	1.154939	0.754709	2.660142	1.736704
1.006431	0.691562	1.229461	0.828565	2.872303	1.964661
0.946468	0.626684	1.159355	0.756633	2.713552	1.768796
1.004435	0.689071	1.227291	0.838136	2.868566	1.962128
0.934921	0.624844	1.153880	0.753633	2.699735	1.774327
1.008477	0.689401	1.230704	0.836536	2.880236	1.966218
0.944659	0.625872	1.150912	0.756944	2.705485	1.778557
1.015397	0.694705	1.241444	0.844660	2.903413	1.977742
0.950761	0.630638	1.165042	0.758740	2.715742	1.791366
1.007359	0.693223	1.233846	0.843092	2.880240	1.952200
0.944579	0.627403	1.160964	0.761233	2.721445	1.775152
1.010641	0.695013	1.229316	0.841263	2.891097	1.965340
0.947246	0.629885	1.158331	0.766671	2.718438	1.784515
1.015650	0.692720	1.239584	0.839329	2.898660	1.969109
0.946619	0.627706	1.161185	0.761933	2.723480	1.774710
1.004146	0.694073	1.234127	0.837957	2.881136	1.955137
0.946752	0.627905	1.164081	0.762125	2.698138	1.774320
1.005840	0.693069	1.230594	0.838697	2.879157	1.959552
0.946325	0.625101	1.157356	0.756876	2.703806	1.781619

Tabulka A.3: **AMD Ryzen 4000 5**.

1x/1.4GHz	**2x/1.4GHz**	**1x/1.7GHz**	**2x/1.7MHz**	**1x/3GHz**	**2x/3MHz**
1.008899	0.692432	1.232953	0.841765	2.878013	1.950519
0.952455	0.629579	1.165142	0.762871	2.720674	1.780712
1.009791	0.694312	1.232626	0.834862	2.881935	1.956572
0.942759	0.628761	1.159043	0.762604	2.717130	1.782168
1.004613	0.692710	1.231671	0.840767	2.886271	1.964042
0.944875	0.631079	1.161551	0.764721	2.710477	1.783279
1.013529	0.695030	1.234849	0.842502	2.897147	1.966990
0.954270	0.629325	1.164827	0.764606	2.695506	1.793237
1.018024	0.693532	1.237851	0.840485	2.858340	1.979354
0.956625	0.629511	1.164275	0.766778	2.694798	1.797358
1.013077	0.693832	1.237927	0.842670	2.877039	1.973729
0.952169	0.627651	1.163873	0.767042	2.693091	1.792998
1.018078	0.696044	1.235108	0.847078	2.844889	1.983881
0.956209	0.630240	1.163153	0.768392	2.709724	1.793017
1.016369	0.693181	1.233023	0.846147	2.889254	1.981691

Tabulka A.4: **Intel i5 2520**. Procesor byl omezen na 3 hodinové rychlosti (sloupce), přičemž jednotlivé řádky reprezentují průměrný násobek ideální rychlosti 100 naměřených snímků (tedy bylo naměřeno celkem 1000 konzistentních snímku specifické hry a okno po 10 snímcích bylo zprůměrováno). Pokračování tabulky A.5

1x/1.4GHz	2x/1.4GHz	1x/1.7GHz	2x/1.7MHz	1x/3GHz	2x/3MHz
0.655273	0.454683	0.795157	0.548547	1.434706	0.980536
0.683726	0.488089	0.839157	0.582340	1.509348	1.050314
0.641811	0.436267	0.790519	0.525791	1.419877	0.936589
0.681576	0.476474	0.833991	0.580165	1.498776	1.020135
0.638984	0.433638	0.786628	0.525719	1.414686	0.932876
0.685392	0.475978	0.831502	0.574748	1.495392	1.026580
0.649344	0.433303	0.787209	0.526461	1.394714	0.936325
0.685921	0.477102	0.832138	0.577407	1.473665	1.029583
0.635652	0.431858	0.783851	0.524666	1.395248	0.936004
0.685154	0.475448	0.827906	0.560239	1.440413	1.026692
0.646601	0.431531	0.777453	0.525573	1.398668	0.928503
0.683128	0.470788	0.831443	0.578677	1.478543	1.022013
0.638645	0.429772	0.781060	0.524011	1.400219	0.935716
0.682870	0.473997	0.831219	0.577584	1.488241	1.029347
0.644370	0.431556	0.784476	0.520098	1.393303	0.935004
0.680888	0.468052	0.831552	0.577193	1.492435	1.023903
0.645353	0.432664	0.783018	0.526869	1.410299	0.937545
0.685205	0.474624	0.826400	0.581488	1.488719	1.030657
0.651709	0.435752	0.785407	0.530457	1.420636	0.943905
0.680046	0.475559	0.820479	0.579492	1.500238	1.019065
0.650415	0.434071	0.752909	0.529131	1.417208	0.938579
0.681644	0.477120	0.741728	0.580750	1.495146	1.029904
0.650511	0.434683	0.715668	0.529571	1.400238	0.939542
0.685131	0.476387	0.640797	0.580741	1.498681	1.030380
0.650571	0.435494	0.779751	0.519795	1.418142	0.939718
0.685456	0.476292	0.832266	0.580857	1.498149	1.030429
0.649010	0.434497	0.788545	0.529289	1.416607	0.939612
0.676332	0.476674	0.816672	0.567529	1.495711	1.030313
0.650121	0.435022	0.737253	0.525021	1.416910	0.936886
0.686493	0.470448	0.818027	0.580848	1.496875	1.028657
0.652188	0.435122	0.784727	0.530456	1.415484	0.941379
0.669511	0.473661	0.832576	0.580225	1.494753	1.030146
0.650381	0.433867	0.788396	0.528983	1.415243	0.936642
0.686862	0.473612	0.834614	0.580466	1.496939	1.029834
0.648732	0.429787	0.791603	0.526708	1.415177	0.940144
0.684949	0.477071	0.836047	0.580443	1.499680	1.030595
0.649732	0.434427	0.787357	0.530038	1.419447	0.938522
0.685849	0.476896	0.833103	0.572328	1.497756	1.030763
0.651100	0.434471	0.790977	0.530968	1.419590	0.940348
0.684771	0.476458	0.833666	0.580602	1.476317	1.030820
0.651248	0.433157	0.789950	0.530401	1.420123	0.940205

Tabulka A.5: **Intel i5 2520**. Pokračování tabulky A.6

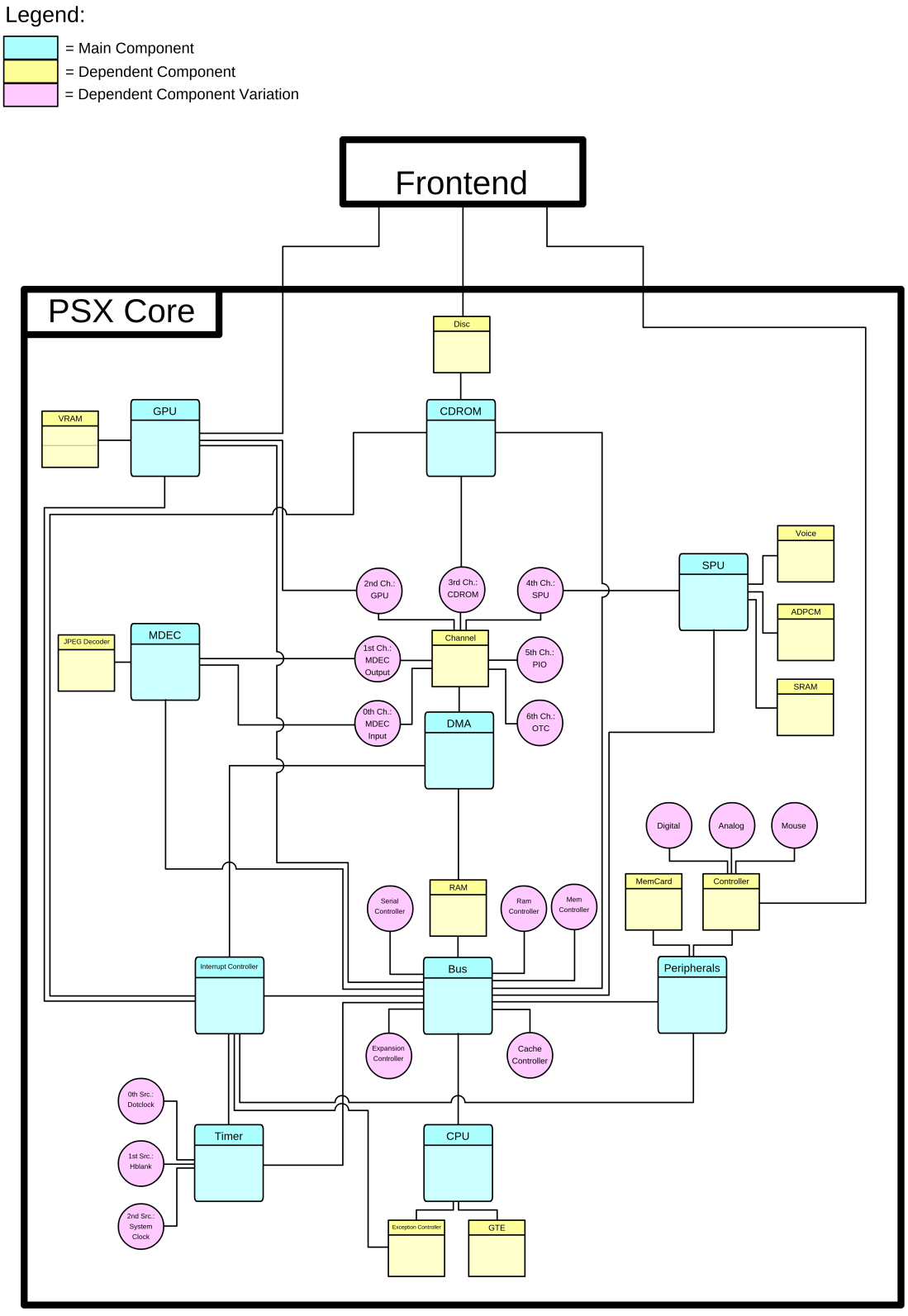
1x/1.4GHz	2x/1.4GHz	1x/1.7GHz	2x/1.7MHz	1x/3GHz	2x/3MHz
0.686070	0.477551	0.814369	0.581015	1.497881	1.029635
0.641728	0.430302	0.789631	0.529621	1.415023	0.940108
0.674466	0.477665	0.834900	0.581576	1.496832	1.031383
0.651952	0.431518	0.791685	0.523629	1.417715	0.940720
0.686646	0.477079	0.836594	0.581557	1.494895	1.029731
0.649303	0.434700	0.783342	0.530525	1.416478	0.940957
0.685725	0.477604	0.834937	0.580476	1.489723	1.032096
0.649781	0.435547	0.789823	0.529715	1.417288	0.940532
0.686115	0.477322	0.834461	0.580592	1.497260	1.032538
0.646389	0.434644	0.769893	0.529379	1.417851	0.940995
0.685433	0.476846	0.832239	0.580177	1.496651	1.030566
0.649680	0.428990	0.789793	0.528902	1.419579	0.940094
0.685206	0.476001	0.834247	0.581126	1.497588	1.030161
0.633904	0.425060	0.790992	0.529534	1.416920	0.937730
0.686114	0.477501	0.829032	0.574758	1.483223	1.031725
0.650558	0.427959	0.790711	0.529744	1.421458	0.940308
0.675486	0.478232	0.834713	0.580995	1.499280	1.031302
0.651433	0.435508	0.790024	0.529632	1.419071	0.938033
0.686863	0.477139	0.810363	0.572763	1.499380	1.031017
0.648759	0.433572	0.787849	0.527761	1.412421	0.937489
0.669279	0.477278	0.832776	0.580411	1.493694	1.029616
0.648545	0.434109	0.788463	0.519885	1.410213	0.937880
0.684120	0.476590	0.834309	0.579086	1.483545	1.026285
0.647771	0.434369	0.783780	0.525217	1.130580	0.937450
0.676276	0.476879	0.831687	0.579056	1.491655	1.027230
0.647146	0.432818	0.786864	0.527088	1.407238	0.937704
0.683580	0.475828	0.830427	0.577560	1.494506	1.027673
0.646933	0.431993	0.785595	0.527081	1.409572	0.935344
0.667140	0.471730	0.828095	0.578669	1.490695	1.028021
0.644746	0.430050	0.783755	0.525617	1.405030	0.937023
0.682515	0.474206	0.828993	0.577553	1.484294	1.023609
0.645038	0.433121	0.783796	0.524210	1.394532	0.936808
0.676927	0.470721	0.827072	0.579690	1.494046	1.031385
0.650079	0.435700	0.791957	0.527946	1.420464	0.941165
0.686001	0.467797	0.835424	0.579538	1.496651	1.031962
0.614336	0.433138	0.787060	0.527959	1.414427	0.935819
0.660199	0.470737	0.831741	0.567343	1.496420	1.030616
0.649516	0.435389	0.768972	0.527505	1.418126	0.939837
0.686542	0.467786	0.836123	0.576086	1.498010	1.030746
0.640281	0.435654	0.789161	0.528803	1.146021	0.928240
0.686388	0.476569	0.834430	0.579853	1.152805	1.030195
0.649948	0.434566	0.786115	0.526130	1.092740	0.939747
0.684967	0.450539	0.831361	0.578979	1.153030	1.028802
0.647586	0.435036	0.786853	0.525627	1.091954	0.938722

Tabulka A.6: **Intel i5 2520**.

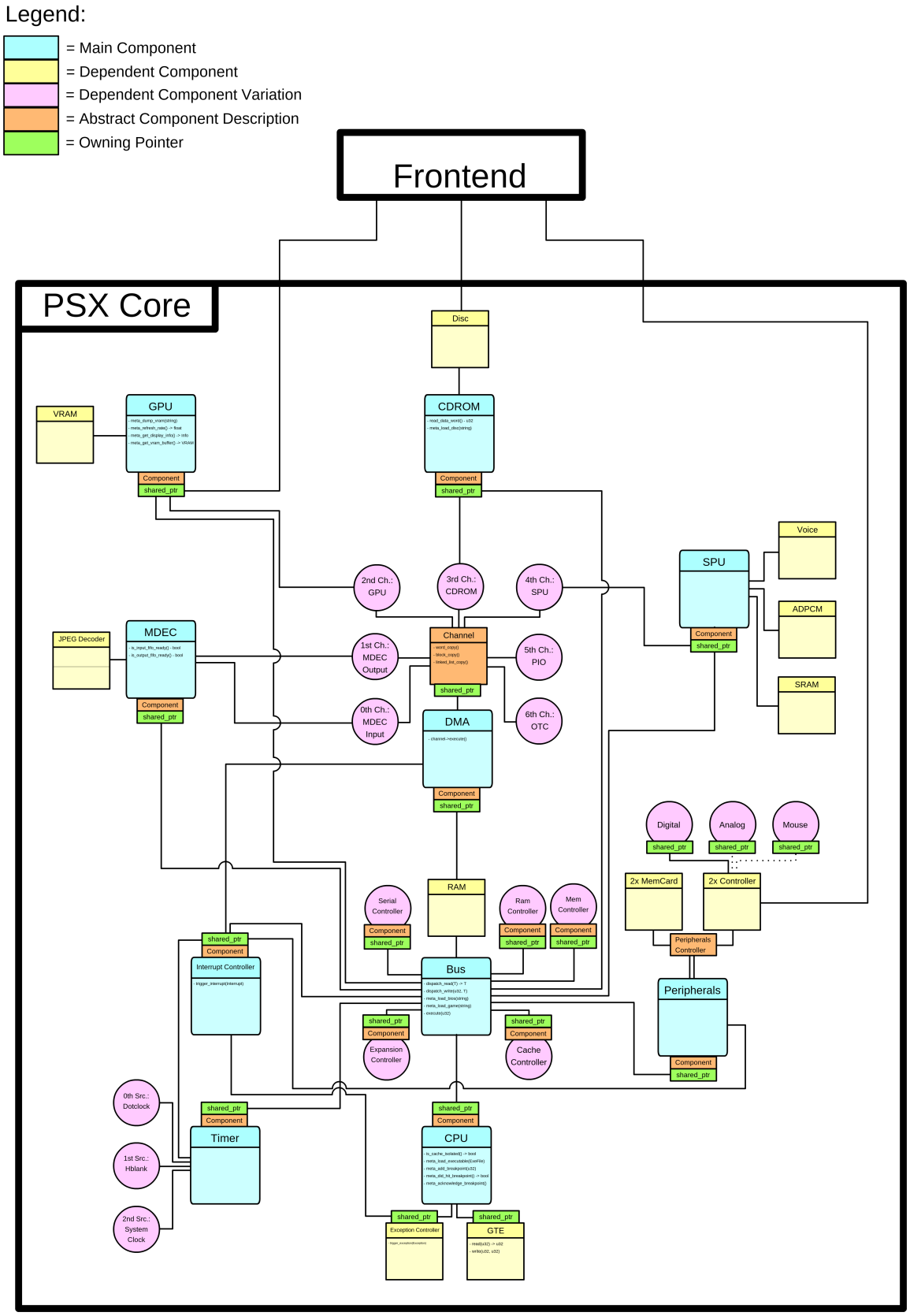
1x/1.4GHz	**2x/1.4GHz**	**1x/1.7GHz**	**2x/1.7MHz**	**1x/3GHz**	**2x/3MHz**
0.684924	0.475867	0.831667	0.579858	1.152271	1.026067
0.650745	0.434397	0.789479	0.528190	1.352606	0.940549
0.685788	0.477040	0.830606	0.571646	1.495003	1.030714
0.647193	0.433610	0.788363	0.528757	1.415690	0.939950
0.686008	0.474921	0.834585	0.576507	1.494838	1.029132
0.649939	0.435008	0.791660	0.516165	1.417299	0.940795
0.676648	0.476670	0.835552	0.579224	1.495929	1.031726
0.511705	0.435659	0.787232	0.520700	1.417702	0.938767
0.531551	0.473757	0.834933	0.580740	1.498218	1.025664
0.509763	0.436310	0.790882	0.447710	1.417072	0.940797
0.685801	0.476756	0.835603	0.578738	1.475480	1.031466
0.650594	0.435538	0.789377	0.527564	1.417521	0.940095
0.686499	0.477396	0.835363	0.579964	1.500624	1.031441
0.649651	0.435353	0.792504	0.527575	1.422035	0.925232
0.685112	0.476044	0.835925	0.577984	1.502216	1.030698

Příloha B

Návrh architektury



Obrázek B.1: Kompletní návrh zapojení nejen hardwarových komponent emulátoru, ale i nadstavby nutné pro zpřístupnění stavu konzole.



Obrázek B.2: Povšimněme si, že každá komponenta dědí z generické virtuální třídy *Component* 3.2, která je spravována přes *smart* ukazatel. To pak umožňuje jednoduše sdílet ten samý objekt mezi vícero komponentami.