FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Zadání bakalářské práce

Název:	Emulátor konzole Nintendo Entertainment System
Student:	Ondřej Golasowski
Vedoucí:	Ing. Stanislav Jeřábek
Studijní program:	Informatika
Obor / specializace:	Počítačové inženýrství
Katedra:	Katedra číslicového návrhu
Platnost zadání:	do konce letního semestru 2023/2024

Pokyny pro vypracování

Navrhněte a realizujte softwarový emulátor herní konzole Nintendo Entertainment System. Zmapujte existující implementace, dostupné dokumentace a katalogové listy. Na základě rešerše navrhněte vhodné technologie a způsob řešení emulace. Využijte OOP a jednotlivé komponenty implementujte jako třídy. Během práce berte v potaz vhodnost budoucího využití ve výuce.

V analýze i následné implementaci se zaměřte na:
– procesor 6502: Implementujte standardní i nestandardní instrukce a obsluhu přerušení. Třídu navrhněte univerzálně tak, aby se dala využít i v jiných emulátorech systémů používajících stejný procesor.
– grafický čip 2C02: renderování pozadí i popředí
– rozšíření procesoru 2A03: zpracování zvuku (APU), implementace syntezátorů a jejich synchronizace
– periferie: herní ovladače
– rozhraní emulátoru: Emulátor by měl poskytovat přehledné grafické rozhraní poskytující nejen uživatelské, ale i vývojářské rozhraní vhodné také pro užití ve výuce (zobrazení stavu komponent, obsahů registrů a pamětí, atd.).

V rámci testování zkontrolujte věrnost emulace pomocí testovacích obrazů ROM. Výsledky srovnejte s reálným systémem.

Elektronicky schválil/a prof. Ing. Hana Kubátová, CSc. dne 24. října 2022 v Praze.

Bakalářská práce

EMULÁTOR KONZOLE NINTENDO ENTERTAINMENT SYSTEM

Ondřej Golasowski

Fakulta informačních technologií
Katedra číslicového návrhu
Vedoucí: Ing. Stanislav Jeřábek
10. května 2023

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2023 Ondřej Golasowski. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení, je nezbytný souhlas autora.

Odkaz na tuto práci: Golasowski Ondřej. *Emulátor konzole Nintendo Entertainment System.* Bakalářská práce. České vysoké učení technické v Praze, Fakulta informačních technologií, 2023.

Obsah

Poděkování **ix**

Prohlášení **x**

Abstrakt **xi**

Seznam zkratek **xii**

Úvod **1**

1 Představení problematiky **3**
1.1 Emulace . . . 3
1.1.1 Způsoby emulace . . . 3
1.2 Klíčové hardwarové principy . . . 5
1.2.1 Komunikace na sběrnici . . . 5
1.2.2 Přerušení . . . 6
1.2.3 Paměti a adresace . . . 6
1.2.4 Přímý přístup do paměti . . . 7
1.2.5 Analogové video . . . 7
1.2.6 Testování hardwaru . . . 7
1.3 Další principy . . . 8
1.3.1 Přehrávání zvuku . . . 8
1.3.2 Synchronizace vláken . . . 8

2 Analýza **11**
2.1 Zdroje informací . . . 11
2.2 Nintendo Entertainment System . . . 11
2.2.1 Hlavní sběrnice . . . 12
2.2.2 Grafická sběrnice . . . 13
2.3 Procesor 6502 . . . 13
2.3.1 Historie . . . 14
2.3.2 Interakce procesoru s prostředím . . . 14
2.3.3 Architektura . . . 15
2.3.4 Specifika Ricoh 2A03 . . . 25
2.4 Grafický čip 2C02 . . . 26
2.4.1 Paměti a data . . . 26
2.4.2 Vnější rozhraní . . . 30
2.4.3 Pracovní registry . . . 30
2.4.4 Inicializace po spuštění . . . 32
2.4.5 Renderování . . . 32
2.5 Game Pak . . . 34
2.5.1 Mapper . . . 36
2.5.2 Formát souboru kopie kazety . . . 37
2.6 Periferie . . . 37

2.6.1 Herní ovladače . . . 38
2.7 Zvukový syntezátor Audio Processing Unit . . . 39
2.7.1 Společné komponenty . . . 39
2.7.2 Generátor pulzů . . . 40
2.7.3 Generátor šumu . . . 41
2.8 Existující řešení . . . 42

3 Návrh 43
3.1 Výběr technologií . . . 43
3.1.1 Programovací jazyk . . . 43
3.1.2 Kolekce vývojových nástrojů . . . 44
3.1.3 Správa zdrojového kódu . . . 44
3.1.4 Dokumentace . . . 44
3.1.5 Testování . . . 45
3.2 Emulační platforma . . . 45
3.2.1 Stanovení požadavků . . . 45
3.2.2 Porovnání s původními návrhy . . . 46
3.2.3 Návrh platformy 2.0 . . . 48
3.3 Emulace konzole . . . 53
3.3.1 Základní komponenty . . . 53
3.3.2 Procesor 6502 . . . 53
3.3.3 Grafický čip 2C02 . . . 54
3.3.4 Kazeta a mappery . . . 54
3.3.5 Zvukový syntezátor APU . . . 55
3.3.6 Herní ovladače . . . 55
3.3.7 Integrace do platformy . . . 55

4 Implementace 57
4.1 Struktura projektu . . . 57
4.2 Emulační platforma . . . 57
4.2.1 Komponenty . . . 59
4.2.2 Práce se zvukem . . . 59
4.2.3 Komunikace komponent . . . 61
4.2.4 Systémy . . . 62
4.2.5 Platforma . . . 64
4.3 Emulované komponenty . . . 64
4.3.1 Univerzální komponenty . . . 64
4.3.2 Komponenty NES . . . 66

5 Testování 75
5.1 Testování univerzálních součástí . . . 75
5.2 Testování komponent konzole . . . 75
5.2.1 Testování procesoru 6502 . . . 75
5.2.2 Testování grafického čipu . . . 80
5.2.3 Testování APU . . . 80
5.2.4 Další komponenty . . . 81
5.3 Shrnutí . . . 81

6 Navazující práce **83**
6.1 Rozšíření emulátoru konzole . . . 83
6.2 Rozšíření emulační platformy . . . 83
6.2.1 Rozšíření funkcionality . . . 83
6.2.2 Implementace komponent . . . 84
6.3 Vývoj dalších emulátorů . . . 84

Závěr **85**

A Diagramy ke konzoli NES **87**

B Knihovna ImInputBinder **91**

C Klony NES **93**

Bibliografie **95**

Obsah přiloženého média **99**

Seznam obrázků

1.1 Úrovně abstrakce softwaru. 4
1.2 Úrovně abstrakce hardwaru. 5

2.1 Obálka hardwarového manuálu rodiny komponent MCS6500 (sken archive.6502.org). 12
2.2 Konzole Nintendo Entertainment System (foto © 2016, Evan-Amos). 12
2.3 Hlavní sběrnice konzole NES. 13
2.4 Grafická sběrnice konzole NES. 13
2.5 Časování čtení procesoru. 15
2.6 Struktura instrukce implikovaného adresního režimu s příkladem instrukce Clear Interrupt Disable Bit (CLI). 17
2.7 Struktura instrukce okamžitého adresního režimu s příkladem instrukce AND Memory with Accumulator (AND). 17
2.8 Struktura instrukce absolutního adresního režimu s příkladem instrukce Load Accumulator (LDA). 17
2.9 Struktura instrukce adresního režimu nulté stránky s příkladem instrukce Store Accumulator (STA). 18
2.10 Struktura instrukce relativního adresování s příkladem instrukce Branch On Carry Set (BCS). 18
2.11 Struktura instrukce indexovaného-nepřímého režimu s příkladem instrukce STA. 19
2.12 Struktura instrukce nepřímého-indexovaného režimu s příkladem instrukce STA. 20
2.13 Struktura instrukce JMP využívající nepřímý absolutní adresní režim s příkladem skoku na adresu $FA55. 20
2.14 Instrukční cyklus procesoru. 21
2.15 Detekce přerušení NMI. 24
2.16 Převážení přerušení BRK přerušením NMI. 25
2.17 Část paletové paměti RAM s ukázkou hodnot barev z množiny dostupné pro čip 2C02. 27
2.18 Struktura pattern table čipu PPU. 28
2.19 Ukázka složení částí dlaždic ve výsledný obraz za použití palety pozadí 3 z obrázku 2.17. 29
2.20 Proces výběru pixelu k vykreslení. 35
2.21 Standardní ovladač konzole NES (foto © 2016 Evan-Amos). 38
2.22 Vzhled průběhů signálu pulzního kanálu APU. 41
2.23 Zpětnovazební registr generátoru šumu. 41

3.1 Úrovně abstrakce práce s grafikou. 47
3.2 Komunikace komponent pomocí tříd Port a Connector. 50
3.3 Tok zvukových vzorků v platformě 2.0. 52
3.4 Integrace NES do platformy 2.0. 56

4.1 Popis struktury projektu. 58
4.2 Schéma toku zvukových dat ze 3 zdrojů. 61
4.3 Porovnání surového čtvercového signálu a jeho aproximace Fourierovou řadou. . . 73

5.1 Vydání testů pro procesor 6502. 78
5.2 Ukázka chybného výstupu generátoru šumu. 81

A.1 Diagram procesu renderování u čipu 2C02 (PPU). Vytvořila komunita Nesdev.org [53]. 88
A.2 Přehledové hardwarové schéma konzole NES. Obrázek „NES-001 Console“ vytvořil schenkzoola [54] a zveřejnil pod licencí CC BY 4.0. 89

C.1 Klon „Super Megason IV“ obsahující kopii periferie NES Zapper. Foto © 2012 kevro, reddit.com. 93
C.2 Klon „Terminator 2“ firmy „Ending-Man“. Součástí balení jsou i pirátské kopie softwaru. Foto © 2021 Tempoulker, reddit.com. 94
C.3 Klon „Dendy Junior“, populární v Rusku. Foto © 2012 Nzeemin, wikimedia.org, CC BY-SA 3.0 [55]. 94

Seznam tabulek

2.1 Adresní prostor CPU. 14
2.2 Registry procesoru 6502. 15
2.3 Popis příznakového registru procesoru 6502. 16
2.4 Ukázka záznamů v PLA procesoru 6502 [22]. 22
2.5 Ukázka popisu instrukcí logických operací. 23
2.6 Stav registrů 6502 po restartu. 23
2.7 Vektory přerušení a resetu. 24
2.8 Adresní prostor vlastní sběrnice PPU. 26
2.9 Procesorem přístupné registry čipu PPU. Označení „-“ znamená, že se používá celý bajt pro jedinou hodnotu. 31
2.10 Pracovní registry pro práci s videopamětí. 32
2.11 Vnitřní operace PPU během viditelných obrazových řádků. 33
2.12 Mapování kazety do rozsahu CPU a PPU. 35
2.13 Adresy a popis vnitřních registrů MMC1. 37
2.14 Formát hlavičky iNES. 37
2.15 Funkce vstupních pinů portů konzole. 38
2.16 Registry APU. 39

Seznam výpisů kódu

3.1 Zobrazení stavů registru „V“ PPU na platformě 1.0. 47
3.2 Tvorba Package z Elementů. 48
4.1 Příklad konfigurace grafického rozhraní pro komponentu emulující procesor. . . . 60
4.2 Implementace rozhraní pro přístup ke zvukovým datům. 61

4.3 Definování konektorů různých typů. 63
4.4 Propojení komponent v systému. 64
4.5 Definice čtecí a zápisové logiky sběrnice využívané například procesorem. 65
4.6 Definice počátečního stavu emulované paměti. 66
4.7 Záznam instrukční tabulky procesoru 6502. 66
4.8 Ukázka implementace absolutního adresního režimu 6502 a instrukce ADC. . . . 68
4.9 Implementace přímého přístupu do paměti v čipu 2A03. 69
4.10 Úryvek implementace rozhraní čipu 2C02. 70
4.11 Dedukce formátu kopie kazety. 71
4.12 Ukázka čtení PRG ROM v mapperu. 72
5.1 Příklad konfigurace testu pro procesor 6502. 76
5.2 Kompilace testů v automatickém sestavení. 76
5.3 Upravený procesor 6502 v Google Test. 77
5.4 Řádek signalizující úspěch v listingu testu 6502. 78
5.5 Testovací cyklus procesoru 6502 v Google Test. 78
5.6 Výňatek z kódu testu signalizující chybu. 79
5.7 Oprava chybné implementace IRQ. 79

Děkuji všem hardwarovým vývojářům i nadšencům za to, co dělají. Od kalkulaček došlo k velkému posunu a je mi ctí prezentovat jeden z hardwarových milníků, který představuje procesor 6502 i celá konzole Nintendo Entertainment System.

Děkuji panu doktorovi Martinovi Novotnému, který mi ukázal, že i přes velkou sofistikaci současných systémů je stále nejen možné, ale i důležité se zabývat vývojem počítačů až na úrovni hradel.

Velký dík poté patří panu inženýrovi Stanislavovi Jeřábkovi, jenž se ujal vedení práce a umožnil tak její vznik. Děkuji mu za věcné poznámky k projektu jako celku i za připomínky k textu bakalářské práce.

Děkuji komunitě nesdev.org, která i dnes stále aktualizuje a přetváří dokumentaci ke konzoli NES, poskytla mi mnoho užitečných faktů, pozorování a testů.

A nakonec, děkuji i Vám, čtenáři, za čtení tohoto textu a zájem o problematiku emulace v souvislosti nejen se vzděláváním.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, avšak pouze k nevýdělečným účelům. Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 10. května 2023 . . . .

Abstrakt

Bakalářská práce se zabývá problematikou emulace v souvislosti s výukou principů počítačových architektur a jejich hardwaru. Na konkrétním příkladě zábavního systému Nintendo Entertainment System ukazuje celý proces vývoje softwarového emulátoru od pochopení základních principů a nutnou analýzu emulovaného systému přes návrh vhodného řešení na základě zjištěných informací až po samotnou implementaci v jazyce C++ a testování výsledku. Snahou implementace je být co nejsrozumitelnější. Součástí řešení je i univerzální emulační platforma, která je dále využitelná pro jiné projekty mající za cíl vytvořit emulátor počítačového systému. Pro motivaci dalších zájemců o problematiku je v poslední kapitole obsažen výčet funkcionalit, které je možné doimplementovat. Byla vytvořena i podrobná dokumentace, aby se zvýšila přístupnost a srozumitelnost projektu nejen pro zájemce o rozšíření platformy, ale i pro zájemce o vytvoření vlastního emulátoru s použitím vytvořené platformy.

Klíčová slova softwarová emulace, Nintendo Entertainment System, vzdělávání, počítačové architektury, hardware, C++, emulační platforma

Abstract

The bachelor's thesis is focused on the problematics of software emulation in the context of teaching the principles of computer architectures and associated hardware. There is a whole emulator development process presented in an example of a particular computer system, which is the Nintendo Entertainment System. The process consists of understanding the basic principles behind software emulation, analysis of the emulated system, design of the solution based on discovered information, and finally, the implementation of the emulator including testing. The goal of the implementation is to be as comprehensible as possible. The project also includes a universal platform for emulator development. To motivate other students (or hobbyists) interested in the topic, there is a list of possible extensions of the project in the last chapter of the thesis. Detailed documentation was created to make the project more accessible for emulator developers and potential project contributors.

Keywords software emulation, Nintendo Entertainment System, education, computer architectures, hardware, C++, emulation platform

Seznam zkratek

A	akumulátor
ADH	address high
ADL	address low
APU	Audio Processing Unit
ASCII	American Standard Code for Information Interchange
ASIC	application specific integrated circuit
CPU	central processing unit
DMA	direct memory access
DUT	design under test
FC	Family Computer
GUI	graphical user interface
HBL	horizontal blanking
I/O	input/output
IRQ	interrupt request
ISA	instruction set architecture
JSA	jazyk symbolických adres
LCD	liquid crystal display
NES	Nintendo Entertainment System
NMI	non-maskable interrupt
NTSC	National Television System Committee
OAM	object attribute memory
OOP	objektově orientované programování
OZ	operační znak
PAL	phase alternating line
PC	program counter
PCM	pulse code modulation
PLA	programmable logic array
PPU	Picture Processing Unit
RAM	random access memory
ROM	read-only memory
S	stack pointer
VBL	vertical blanking
VRAM	video random access memory
URISC	ultimate reduced instruction set computer
USE	Universal System Emulator (platforma 2.0)

Úvod

„We can only see a short distance ahead, but we can see plenty there that needs to be done.“

Computing Machinery and Intelligence
Alan Turing

Technologický pokrok je nezadržitelný. Nové poznatky umožňují rychlý vývoj sofistikovaného technického vybavení výpočetních číslicových elektronických strojů — sálových, domácích i mobilních univerzálních i specializovaných počítačů, vestavěných řídicích systémů a dalších zařízení.

Spolu s technologickým pokrokem přichází mnoho nových informací. V mnoha oblastech však obecné principy zůstávají podobné, ne-li stejné. Přirozeně se k demonstraci principů nabízí využít jednoduššího systému. Takový systém můžeme získat vytvořením modelu aktuálních složitých systémů, což se prakticky využívá například v systémech reálného času [1]. Jinou variantou řešení se zabývá tento text; využitím historického systému.

Historické systémy, podobně jako ty moderní, vychází z teoretických matematických konceptů (například programovatelný počítač vycházející z Turingova stroje [2]). Zároveň jsou poměrně jednoduché, protože vznikaly s technologickými omezeními. Není tedy potřeba vytvářet abstraktní model, ale demonstrovat principy na existujícím systému, což může zvýšit atraktivitu i užitečnost předávaných informací.

Jedním ze způsobů, jak takový systém přiblížit jakémukoliv zájemci o problematiku, je přenést jej do softwaru, který bude možné spustit na běžně dostupných počítačích. Jednou z možností je takzvaná emulace; výsledný software je nazýván emulátor.

Cílem bakalářské práce je vytvořit emulátor historické herní konzole *Nintendo Entertainment System* (NES). Ježto je kladen důraz na využití ve výuce, je nutnou součástí návrhu vývoj univerzální platformy, která takový systém zvládne nejen emulovat, ale zároveň zobrazovat informace o vnitřním stavu systému, jakožto i umožnit jednoduché modifikace a přidávání funkcionalit.

Implementační část, hotová emulační platforma, je pouze dílčí výsledek práce. Samotný vývoj emulovaných komponent přináší mnoho zajímavých problémů k řešení, proto je namístě tento proces důkladně dokumentovat a vytvořit tak příklad pro čtenáře, kteří by chtěli příkladnou implementaci rozšířit, případně na platformě vyvinout emulátor jiného systému. Dílčím cílem práce je tedy seznámit čtenáře s vývojem a motivovat jej ještě důkladněji zkoumat prezentované principy.

Práce je členěna na několik hlavních částí:

Představení problematiky V této kapitole je představena problematika emulace v teoretické rovině — definují se potřebné pojmy. Kapitola popisuje jednak emulaci obecně, jednak témata, se kterými se může vývojář emulátoru setkat a měl by je proto chápat v širších souvislostech.

Analýza Analytická část, stěžejní kapitola práce, se věnuje analýze konkrétního emulovaného systému. Jelikož neexistuje jeden dokument, který obsahuje vše potřebné, je nutné informace seskupit a podat přívětivou formou. Mimo jiné tato kapitola slouží jako ukázka čtenáři, kde a jak by mohl hledat informace pro případný svůj vlastní vývoj libovolného emulovaného systému.

Návrh Po zpracování potřebné teorie i konkrétních faktů týkající se konzole je možné vytvořit vhodné řešení. V kapitole je možné nalézt diskusi k samotnému vývojovému procesu, který je nezbytný k rychlému a pohodlnému vývoji, dále pak úvahy nad způsoby vytvoření univerzální emulační platformy a nakonec i způsob integrace konzole NES do navrženého systému.

Implementace Implementační kapitola je popisem procesu tvorby emulační platformy a emulovaných komponent. Ukazuje konkrétní problémy, které byly řešeny a nabádá čtenáře k souběžnému průzkumu zdrojového kódu.

Testování Předposlední kapitola je zaměřena na testování projektu. Popisuje nejen průběžné testování aplikace, ale i porovnání věrnosti s reálným systémem pomocí testovacích programů.

Navazující práce Poslední kapitola je věnována popsání funkcionalit, které mohou být implementovány v dalších verzích emulační platformy. Slouží především jako pobídka dalším čtenářům-vývojářům, kteří by měli zájem projekt rozšířit ať už v rámci sebevzdělávání nebo v rámci vlastní závěrečné práce.

▶ Poznámka (Terminologie). Jelikož je bakalářská práce zaměřena na vzdělávací využití, kombinuje odbornost s populárně-naučným formátem. Počítá se s faktem, že čtenář se v oboru informačních technologií již pohybuje. Ačkoliv se v obrozeneckém duchu používají české (nebo počeštěné) výrazy, vyskytují se i anglické termíny tam, kde je to běžné a v dané situaci lepší volbou. Nemělo by tedy například čtenáře zaskočit, že se občas *programové vybavení* počítače označuje výrazem *software* a *technické vybavení* počítače jako *hardware*.

▶ Poznámka (Značení). V textu se často používají čísla v šestnáctkové soustavě, poněvadž úsporně reprezentují například paměťové adresy. Šestnáctková čísla se značí předponou amerického dolaru: $. Desítková čísla jsou uvedena bez předpony.

Dále se používají různé zkratky; jsou uvedeny v seznamu zkratek. Neobvyklé zkratky se před jejich použitím v textu vysvětlují.

V textu se taktéž vyskytují paměťové diagramy, to jest znázornění relevantních fragmentů paměti. Takové diagramy mají bajty vždy řazeny dle pořadí v paměti zleva doprava. Jsou-li však v těchto diagramech jednotlivé bajty rozepsány dvojkově, jsou bity (číslice) uspořádány od nejvyšší váhy po nejnižší tak, jako je zvykem u zápisu čísel pozičních číselných soustav.

Kapitola 1
Představení problematiky

„There's no sense in being precise when you don't even know what you're talking about.“

— John von Neumann

1.1 Emulace

V úvodu byl použit pojem emulátor. Pro začátek je tedy vhodné tento pojem oficiálně zavést.

▶ **Definice 1.1** (Emulátor)**.** *Emulátor je software, který umožňuje běh počítačových programů na jiné platformě, než pro kterou byly původně vytvořeny.* [3]

▶ Poznámka 1.2 (Emulovatelnost). Zabývat se vytvářením emulátoru má smysl, jelikož lze pro každý software vytvořit příslušný emulátor. Lze se odkázat na Churchovu-Turingovu tezi, ze které vyplývá, že ke každému algoritmu existuje ekvivalentní Turingův stroj.

Dle jiné definice pod pojem emulátor spadá i hardwarové řešení emulátoru. Tímto se však práce nezabývá, proto bude dále brána v potaz jen již uvedená softwarová emulace.

Emulace se od podobného pojmu, *simulace*, liší především tím, že se na emulátoru spouští originální programové vybavení emulovaného systému. Nedochází tedy k napodobení funkce, ale celého hardwaru tak, aby byl schopný věrně interpretovat původní program. V případě této práce se jedná o interpretaci instrukcí původně obsažených v paměti ROM kazety.

▶ **Příklad 1.3.** V kontextu herních konzolí lze uvést rozdíl na následujícím příkladu. Simulace by napodobila vzhled a chování každé jednotlivé hry. Například simulátor příruční herní konzole s vestavěnou hrou *Tetris* by byla nová implementace hry bez ohledu na hardware, který byl v konzoli použit. Emulátor by naopak nebral žádný ohled na jakýkoliv software, ale snažil by se věrně napodobit hardwarové vybavení konzole tak, aby bylo možné kopii softwaru (hru) spustit beze změn. Takto je možné provozovat na emulátoru jakékoliv programové vybavení kompatibilní s daným hardwarem. [4]

1.1.1 Způsoby emulace

Emulaci je možné dělit dle úrovně, na které emulátor pracuje, což je úzce spjato s teorií počítačových architektur. Na úvod je vhodné se zamyslet nad programováním fyzického

počítačového systému, což poskytne přehled o dostupných zdrojích informací pro vývoj emulátoru. Tato podkapitola tedy odpoví na dvě otázky:

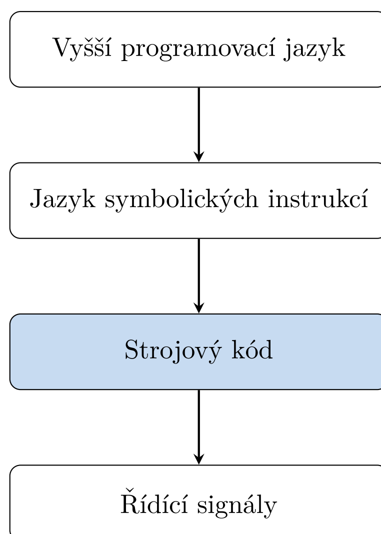
1. V jaké formě bude spouštěný software?
2. Na jaké úrovni se tento software zpracuje?

Běžné počítačové systémy odpovídají teoretickému modelu programovatelného počítače.

▶ **Definice 1.4** (Programovatelný počítač)**.** *Programovatelný počítač je takový počítač, jehož chování lze změnit výměnou programu sestávajícího z instrukcí uložených v paměti.* [5]

Komponentou počítače, která je zodpovědná za řízení, je většinou *procesor*. Proto má smysl se nejdříve zamýšlet nad úrovní abstrakce procesoru, jelikož je to právě ta komponenta, která bude zpracovávat programy a řídit komponenty ostatní.

Instrukce bývají v paměti číslicových počítačů reprezentovány jako strojový kód, který většinou vzniká překladem z jazyka vyšší abstrakce (například JSA) [6], což ilustruje diagram 1.1. Jelikož je strojový kód nativní způsob zpracování instrukcí a zároveň se v této formě běžně distribuuje software, emulátor by měl pracovat právě s touto reprezentací. Tím se získala odpověď na první otázku.



Obrázek 1.1 Úrovně abstrakce softwaru.

Druhá otázka se již zabývá přiřazením smyslu jednotlivým instrukcím. Množina podporovaných instrukcí včetně dalších potřebných informací (především o způsobu reprezentace a ukládání dat) je součástí architektury procesoru (ISA) [6].

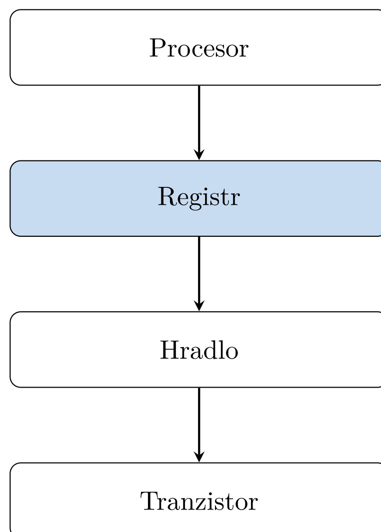
Z výše uvedeného vyplývá, že pro zpracování instrukcí tak, jako to dělal původní hardware, stačí jen přesně napodobit chování jednotlivých instrukcí dle popisu architektury, bez dalšího zamýšlení se, jak je procesor konkrétně implementován. To představuje nejvyšší úroveň abstrakce.

Některé programy se však občas spoléhají na nedokumentované chování procesorů, kde je již nutné pracovat na nižší úrovni. Dle prof. Kubátové [6] se jedná o úroveň předávání dat mezi registry, na které pracuje i emulátor bakalářské práce.

Existují ještě dvě nižší úrovně, úroveň logických hradel a úroveň tranzistorů [6]. Tyto dvě úrovně již však vyžadují znalost konkrétní hardwarové implementace, která bývá obchodním tajemstvím. Výhodou je, že ve své podstatě nevyžaduje vůbec znalost o funkci procesoru jako takovém a zároveň nejvěrněji implementuje jeho funkčnost. Velkými nevýhodami jsou obtížnost, častá absence potřebných informací a při implementaci v softwaru i velká náročnost

na prostředky, jelikož se emuluje každý řídicí signál, tedy nejnižší úroveň řízení dle diagramu 1.1. Zájemce o tuto úroveň lze odkázat na projekt Visual6502 [7].

Všechny úrovně shrnuje diagram 1.2, kde je zvýrazněna úroveň používaná v této práci.



Obrázek 1.2 Úrovně abstrakce hardwaru.

1.2 Klíčové hardwarové principy

Tato podkapitola popisuje vybrané principy z oblasti počítačového technického vybavení, především za účelem uvedení do témat dále rozebíraných a pro dovysvětlení terminologie.

1.2.1 Komunikace na sběrnici

▶ **Definice 1.5** (Sběrnice)**.** *„Je to skupina vodičů, k nimž jsou připojeny vstupy a výstupy jednotlivých jednotek. (. . .) Vodiče lze rozdělit na adresové, datové, řídicí a stavové.“* [8]

Sběrnice je způsobem propojení více komponent tak, aby spolu mohly komunikovat. Tyto komponenty se dělí na řízené (volané, jsou cílem komunikace) a řídicí (volající, jsou zdrojem komunikace). Přenos dat může probíhat tak, že se nejprve adresou identifikuje volaná komponenta, nastaví se pomocí řídicího signálu směr přenosu (čtení, zápis) a na datových signálech se vystaví přenášené informace. V případě čtení vystavuje informace volaná komponenta, v případě zápisu volající. [8]

V nejjednodušším případě existuje na sběrnici jen jediná řídicí komponenta, poté není potřeba řešit přidělování (arbitraci) sběrnice, což je i případ sběrnic v konzoli NES.

▶ Poznámka 1.6 (Terminologie sběrnic). Sběrnice je dle definice skupinou vodičů. Může se tedy takto označit nejen celá skupina složená z adresových, datových i řídicích, ale i jednotlivé podskupiny. V textu se vyskytují označení jako adresní sběrnice, tím je myšlena část sběrnice obsahující pouze adresní vodiče.

1.2.2 Přerušení

▶ **Definice 1.7** (Přerušení)**.** *„Přerušení spočívá v tom, že se přestane provádět původní sekvence instrukcí a že se začne provádět jiná sekvence, nazývána rutina přerušení, která začíná na určené adrese; ta se nazývá přerušovací adresa.“* [8]

Zjednodušeně řečeno lze tvrdit, že přerušení je podprogram, který může být vyvolán i jinak, než programově. Z toho vyplývá i fakt, že může být vyvoláno téměř v jakékoliv fázi zpracování programu. Při signalizaci, že došlo k přerušení, se uloží aktuální kontext procesoru (například stavy příznaků), hodnota programového čítače se nastaví na hodnotu přerušovací adresy a pokračuje se ve vykonávání programu z této nové adresy. Většinou je nutné brát v potaz i příčinu přerušení (jeho zdroj), na příčině totiž může být závislá hodnota přerušovací adresy. [8] Konkrétně procesor 6502 použitý v NES má tyto adresy dvě (tři, pakliže je počítána i adresa pro reset).

▶ Poznámka 1.8 (Terminologie přerušovací adresy). Umístění přerušovací adresy v paměti je v manuálech procesoru 6502 [9] označováno jako vektor přerušení. Uvedená terminologie je použita i v této práci.

Existuje-li v systému více přerušení, mohou být stanoveny jejich priority. Dále může být nastaveno ignorování některých přerušení. To může být provedeno stanovenou hodnotou v příslušných registrech, taková hodnota je pak označena jako maska přerušení. [8]

V případě NES se jedná o dva typy přerušení. Je možné vyvolat přerušení programově (instrukce BRK) i pomocí přerušovacích vstupů procesoru (IRQ, NMI). Konkrétní aplikace je popsána v sekci 2.3.3.10.

1.2.3 Paměti a adresace

Pamětí jsou takové komponenty, které umožňují ukládat informace v různých formách. V případě této práce tak budou označována média skládající se z paměťových míst, do kterých lze ukládat položky o stejných velikostech (nejčastěji půjde o osmibitovou hodnotu). Mohou být různých typů; hlavní členění bude spočívat v trvanlivosti uložených informací: paměti permanentní (nepřepisovatelné: read-only memory, ROM) a paměti měnitelné (přepisovatelné: RWM či ROM, read/write memory, respektive random access memory) [8].

Pro přístup do paměti ke konkrétním položkám se používá jejich adresa, která může být různě široká. Běžně jde o nezáporné číslo. Množina možných adres je označována jako adresní prostor [8]. Součástí adresního prostoru nemusí být jen paměť, ale díky sběrnici i další komponenty adresované stejnými vodiči. Takto je možné adresy přiřadit i periferiím, které se pak označují jako periferie mapované do paměti.

Adresa může být členěna na několik částí, což používá i manuál k procesoru 6502. V architektuře 6502 je adresa 16bitová. Je členěna na dvě části po 8 bitech. Horní část dělí paměti na 256 segmentů o velikosti 256 bajtů označovaných jako paměťová stránka, v rámci kterých se adresuje pomocí dolní částí adresy. [9]

V širším kontextu NES je adresa dělena na části o různých jiných velikostech. Horní část vždy slouží k adresaci paměťových segmentů, dolní část k adresaci v rámci segmentu. V takovém obecném případě se segmenty označují jako banky, což je hojně používáno v souvislosti s manipulací s adresním prostorem.

Adresní prostor již ze své definice omezuje množství položek, ke kterým je možné přistupovat. Vhodnou manipulací s adresami však lze adresní prostor rozšířit. Chceme-li například adresovat více než 2^{16} záznamů, můžeme pomocí registrů přidat libovolné množství pomyslných adresních vodičů, čímž adresní prostor rozšíříme. Tato technika je hojně používána u takzvaných mapperů, které do adresního prostoru vystaví konfigurovatelné registry sloužící k přepínání částí paměti, které se objeví ve viditelném adresním prostoru komponent. Více o mapperech v kapitole 2.5.1.

1.2.4 Přímý přístup do paměti

Existuje-li v systému více komponent, může se vyskytnout potřeba přemisťovat větší množství dat z jedné komponenty do druhé. To je možno provést buďto bajt po bajtu procesorem, nebo lze využít techniky přímého přístupu do paměti (DMA, direct memory access). V takovém případě přenos provádí specializovaný řadič DMA. [9]

V případě NES k přenosu dochází přímo na hlavní sběrnici, tudíž je pro uvolnění sběrnice procesor po dobu běhu DMA pozastaven, potřebná data jsou přenesena a poté je běh procesoru obnoven, více v části 2.3.4.

1.2.5 Analogové video

Grafický čip použitý v NES generuje přímo analogový kompozitní signál, proto je nutné porozumět alespoň struktuře, v jaké dochází k vykreslování na televizoru.

Vše vychází z principu vykreslování obrazu televizory typu Cathode-Ray Tube. Ty obsahovaly elektronové dělo, které postupně zleva shora po řádcích (označované jako scanline) vykreslovalo obraz po bodech. Dělo se mezi řádky muselo vracet zpět doleva. V ten moment signál reprezentoval barvu označovanou jako „černější než černá" a tento interval je označen jako horizontal blanking (HBL). Kompozitní video standardu NTSC obsahuje celkem 525 obrazových řádků, z toho viditelných je 480. Zbytek je prostor pro navrácení děla zpět vlevo nahoru, což je nazýváno vertical blanking interval (VBL). V případě vysílání analogové televize byla tato „mezera" využita například pro teletext či skryté titulky. [10]

U NES se interval VBL využíval pro nerušenou práci s grafickým čipem a grafickou sběrnicí, viz sekce 2.4.5.1.

▶ Poznámka 1.9 (Prokládání a NES). Běžně se u NTSC využívala technika nazývaná prokládání, kdy se během jednoho vykreslování zobrazil jen takzvaný půlsnímek v sudých řádcích, poté další půlsnímek v lichých řádcích, což dohromady tvořilo obraz o vyšším rozlišení. U NTSC je vykresleno celkem 50 půlsnímků za sekundu (tato hodnota se označuje jako vertikální obnovovací frekvence), to odpovídá 25 celým snímkům za sekundu (což se označuje jako snímková frekvence). Tomu odpovídá digitální rozlišení označované jako 480i (i je zkratka pro interlaced). [10]

NES pracuje pouze s jedním půlsnímkem, čímž se vychyluje ze standardu. Dvěma stejnými půlsnímky tak je vytvořen obraz o nižším rozlišení, v digitálním světě tomu odpovídá rozlišení 240p (p znamená progressive; technika progresivního skenování). [11]

1.2.6 Testování hardwaru

Pro testování hardwaru existuje mnoho metod. Jelikož je hardware v bakalářské práci softwarovým modelem, omezuje se testování na softwarové metody. Jednou z takových metod používaných i pro testování reálného hardwaru jsou *testovací programy.* Ty fungují tak, že postupně provádí operace a ověřují, zdali přinesly očekávaný výsledek. Správnost výsledku je odvozena od popisu v dokumentaci či patentech, popřípadě od analýz fyzického hardwaru.

1.2.6.1 Testovací programy

Testovací programy používané v bakalářské práci jsou pouze ve formě strojového kódu přímo zpracovatelném procesorem.

Testovací programy umožňují získávat informace o průběhu různými metodami. Jedná-li se o test spouštěný na celém systému i s uživatelským rozhraním, provádí se ovládání testu a monitoring přes toto rozhraní.

Často ale není vhodné (nebo ani možné) s testy interagovat takto přímočaře. Spouští-li se test automatizovaně nebo na systému bez uživatelského rozhraní, je nutné přistoupit k jinému řešení.

Testy se spouští většinou vhodným nastavením programového čítače, popřípadě stačí, když je programový čítač nastaven na hodnotu vektoru resetu, což se v případě 6502 děje automaticky (viz sekce 2.3.3.9).

Monitorování je možné dvěma způsoby. Jednodušší je přímočaré sledováním předem určeného místa v paměti (některé testy kromě číselných hodnot do paměti přímo vypisují textový stav v ASCII). Používá-li test tuto metodu, často stačí zjistit, co jaká chybová hláška znamená a podle toho provést opravy. Složitější varianta je v případě použití zacyklení, přezdívané jako TRAP. TRAP se dá implementovat tak, že instrukce skáče na svou vlastní adresu. Je-li dostupný monitoring programového čítače procesoru, pozná se TRAP tak, že nedochází ke změnám programového čítače. TRAP každopádně pouze informuje, kde došlo k chybě ve strojovém kódu, což se rozebere dále.

Aby bylo možné výsledky testu používající TRAP interpretovat, je třeba mít dostupný zdrojový kód programu v JSA a zároveň i *listing*.

▶ **Definice 1.10** (Listing)**.** *Listing, neboli výpis, je soubor generovaný assemblerem. Obsahuje původní zdrojový kód, kde je navíc každá instrukce doplněna o adresu, na které se v paměti nachází, a také odpovídající strojový kód, do kterého byla přeložena.* [12]

Pomocí listingu lze tedy zjistit, jakému řádku ve zdrojovém kódu odpovídá adresa, kde došlo k zacyklení (TRAP). Analýza chyb poté spočívá v procházení nejbližšího kódu. Často jsou součástí kódu i komentáře, které popisují očekávané chování a důvod, proč byl TRAP vyvolán. V naprosté většině případů je ale nutné rozumět JSA dané instrukční sady a zároveň i správně interpretovat program.

1.3 Další principy

1.3.1 Přehrávání zvuku

Jelikož se v práci počítá i s použitím zvukového výstupu počítače, je zahrnut stručný přehled o používaných termínech a principech. V analytické části byla vybrána knihovna miniaudio, proto je výklad zaměřen na souvislost se zmíněnou knihovnou a následuje stručný výtah z dokumentace ke knihovně [13].

V nejobecnější rovině spočívá přehrávání zvuku v pravidelném zasílání dat v pevně daném formátu do ovladače audio periferií. Jelikož by nemělo docházet k přerušením během přehrávání (způsobující nepříjemné „chrastění“), musí být tok zvukových dat souvislý. O data si pravidelně žádá ovladač prostřednictvím „callbacku“, což je obyčejná C funkce [13].

Zvuková data jsou složená z takzvaných rámců (frames), které se skládají ze vzorků (samples). Počet vzorků v rámci určuje počet audio kanálů (pro stereo jsou to dva kanály — levý a pravý). Počet rámců za sekundu určuje vzorkovací frekvence. Callback je ovladačem volán během jedné sekundy vícekrát. Ovladač si vždy požádá o nějaké množství rámců, což je předáno argumentem. Úkolem aplikace je tedy dodat požadované množství rámců, pro stereo série vzorků levý, pravý, levý, pravý. . . [13]

1.3.2 Synchronizace vláken

Vykonávání více úloh prostřednictvím vláken je v práci použito pouze v souvislosti se zpracováním zvuku, kdy volání callbacku (viz sekce 1.3.1) probíhá v jiném vlákně než vše ostatní. Aby nedocházelo k souběžnému přístupu k jednomu zdroji, je nutné zvuková data předávat zvláštním způsobem.

Problém předávání zvukových dat ovladači odpovídá klasické synchronizační úloze producent-konzument. Tato úloha spočívá v tom, že existuje několik zdrojů dat vytvářející data paralelně a několik zpracovatelů zpracovávající tato data také paralelně. Zdroje i zpracovatelé tedy sdílejí

předávací místo informací, tím je většinou sdílená paměť. Problém nastane tehdy, když je buffer prázdný, nebo plný. Aby nemusela vlákna čekat, uspávají se. [14]

Takový obecný případ je v rámci zpracování audia zjednodušen na jednoho producenta a jednoho konzumenta, kde navíc uspávání nemusí být ideálním řešením. Alternativu proto nabízí přímo miniaudio ve formě kruhového bufferu, který při naplnění či úplném vyprázdnění bufferu umožňuje posouvat ukazatele do bufferu, takže není nutné vlákna uspávat. Navíc používá atomické operace, nikoliv zámky, proto je buffer efektivní i pro velice časté přístupy (lze tedy přidávat zvukové rámce po jednom). [13]

Kapitola 2
Analýza

„Kowalski, Analysis.“

Penguins of Madagascar
Skipper

2.1 Zdroje informací

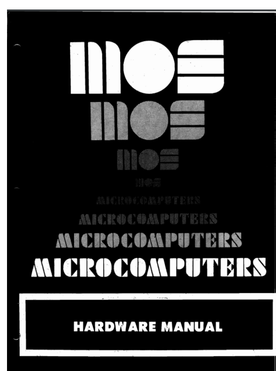
Hardware, není-li open-source, nebývá dokumentován do větší míry, nežli je třeba k vytváření softwaru pro danou platformu. Jinak tomu není ani po vypršení patentu. Přestože veškeré patenty konzole Nintendo Entertainment System již vypršely [15], společnost Nintendo nevydala (ani nemá důvod vydat) kompletní hardwarový manuál. V takovém případě je nutné tyto informace získat jinou formou. Nabízí se časově náročná metoda reverzního inženýrství. Díky popularitě a stáří systému NES již ale vzniklo mnoho komunitní dokumentace, na niž se lze odkazovat a při vývoji emulátoru není potřeba mít k dispozici reálný systém.

Velkým komunitním zdrojem je organizace nesdev.org zabývající se neoficiálním vývojem softwaru (nazýváno „homebrew“) a emulátorů. Tento zdroj je důležitý především pro vývoj softwarového modelu proprietárních komponent specifických pro NES — grafického čipu, zvukového syntezátoru, paměťového rozhraní pro ROM a dalších.

Komponentou, která byla využívána i v jiných systémech, je (kromě základních součástek jako posuvné registry) procesor 2A03. Jelikož je klonem procesoru 6502 od firmy MOS, existuje mnoho dokumentace od ISA až po popis na úrovni hardwaru — viz publikace [9], jejíž obálka je na obrázku 2.1. V této publikaci je popsán nejen hardwarový princip, ale i filozofie za jednotlivými návrhovými rozhodnutími.

2.2 Nintendo Entertainment System

Konzole Nintendo Entertainment System, často zkracována jako NES, je osmibitový zábavní počítačový systém firmy Nintendo, který byl vydán nejprve v Japonsku jako Family Computer (FC, „Famicom“). První verze je ukázaná na obrázku 2.2. V České republice je tento systém znám především díky mnoha klonům („televizní hry na žlutých kazetkách“), které byly levnější a dostupnější než oficiální systém. Tyto klony používaly kopie původního hardwaru, poté se objevily hardwarové emulátory založené na ASIC, které celou konzoli zmenšily do jednoho čipu (proto přezdívány NES-on-a-chip). Tato kapitola má za úkol popsat především technické specifikace systému — zájemce o podrobnou historii NES lze odkázat na Wikipedii [16], [17], kde je mnoho



Obrázek 2.1 Obálka hardwarového manuálu rodiny komponent MCS6500 (sken archive.6502.org).

odkazů na čínské klony konzole (často s groteskními názvy: „Terminator 2 Super Design“ od firmy „Ending-Man“, „Dr. Boy“, „Fun Time Home Computer: The New System“). Pro ilustraci jsou fotografie některých klonů součástí přílohy C. Další informace o klonech jsou v článku [18].

Obrázek 2.2 Konzole Nintendo Entertainment System (foto © 2016, Evan-Amos).

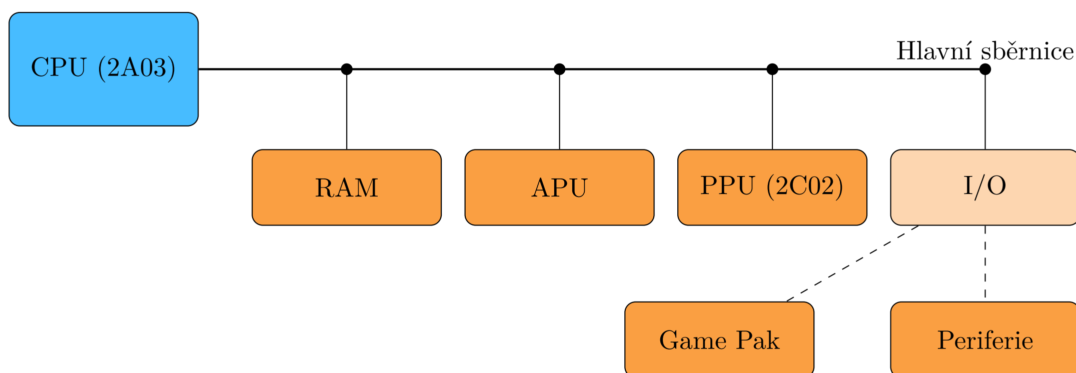
Systém se skládá z několika hlavních komponent, které spolu komunikují pomocí dvou sběrnic. Nejprve jsou představeny sběrnice a k nim připojené komponenty, jež jsou později popsány podrobněji.

2.2.1 Hlavní sběrnice

Hlavní sběrnice, kterou ilustruje obrázek 2.3, je adresována 16 bity a přenáší 8 datových bitů. Komunikaci na hlavní sběrnici řídí pouze procesor *2A03*, který obsahuje i řadič přímého přístupu do paměti (DMA). Tudíž neexistuje (a ani není potřebná) žádná forma arbitrace.

Procesor má k dispozici 2 kB paměti RAM. Přímo v procesoru se nachází čip pro generování zvuku, přezdívaný jako Audio Processing Unit (APU). Na sběrnici se dále nachází grafický čip *2C02*, označovaný jako Picture Processing Unit (PPU). Jako poslední je na sběrnici několik vstupně-výstupních (I/O) rozhraní: slot pro paměťové médium typu kazeta (cartridge), obchodně označovaná jako Game Pak, pomocí níž se distribuoval veškerý software pro konzoli, a porty pro periferie, především herní ovladače.

▶ Poznámka 2.1 (APU a hlavní sběrnice). APU, ačkoliv je součástí procesorového čipu, také komunikuje na hlavní sběrnici. Proto je na obrázku 2.3 uveden jako další zařízení na sběrnici.



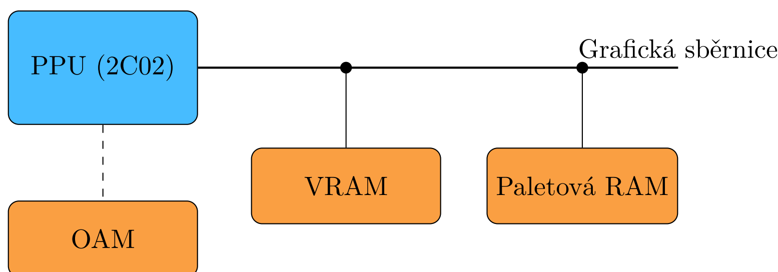
Obrázek 2.3 Hlavní sběrnice konzole NES.

2.2.2 Grafická sběrnice

Systém obsahuje i vedlejší sběrnici adresovanou 14 bity (celkem tedy 16 kB adresovatelného prostoru), kde komunikaci řídí PPU. Schematicky je znázorněna na obrázku 2.4. Tato sběrnice je zcela oddělena od hlavní. Sběrnice obsahuje paměť video RAM (VRAM, také označována jako CIRAM) o kapacitě 2 kB. Do paměťového prostoru je dále mapována RAM obsahující barevnou paletu. V systému existuje ještě paměť OAM (Object Attribute Memory), která obsahuje seznam spritů a informace potřebné k jejich zobrazování. Ta ale není připojena ke sběrnici, nýbrž přímo k čipu PPU.

▶ **Definice 2.2** (Sprite)**.** *Sprite (počeštěně sprajt) je dvourozměrný obrázek, který bývá integrován do větších scén. Termín pochází z dob, kdy se zvlášť vykreslovalo pozadí a právě sprity, které často představovaly herní postavičky a asociované předměty (zbraně, náboje a další). To je případ i konzole NES, která pro sprity měla dedikovanou paměť OAM.*

▶ Poznámka 2.3 (Přístup CPU na grafickou sběrnici). Přestože není procesor přímo připojený na grafickou sběrnici, může na ní nepřímo komunikovat přes registry PPU ($2006 a $2007), které jsou mapovány na hlavní sběrnici (a tím i do adresního prostoru CPU). Tyto registry se používají i pro přístup během DMA.



Obrázek 2.4 Grafická sběrnice konzole NES.

2.3 Procesor 6502

Základem NES je klon procesoru *6502*, označený jako *2A03*. Tato podkapitola popisuje původní variantu procesoru a specifika klonu jsou popsány v samostatné podkapitole 2.3.4.

2.3.1 Historie

Procesor 6502 navrhla firma MOS Technology v roce 1975. Na procesoru pracoval tým, který původně navrhoval mikroprocesor Motorola 6800. Čip 6502 vznikl jako levnější a rychlejší alternativa procesoru od Motoroly pod vedením Chucka Peddla [19], která zachovává hardwarovou kompatibilitu. Cílem bylo umožnit využití i v projektech, kde by jinak byla levnější diskrétní logika [9]. Ve své podstatě jde o aplikaci programovatelného počítače (definovaného v kapitole 1) v praxi — funkce zařízení lze změnit pouze výměnou programu, což byl i jediný požadavek při přechodu z konkurenční Motoroly; upravit program pro ISA 6502.

2.3.2 Interakce procesoru s prostředím

Na úvod, jak se píše v hardwarovém manuálu [9], je vhodné se zabývat tím, v jakém prostředí procesor bude pracovat a jak s ním bude interagovat. Prostředím je myšlen systém, který bude procesorem řízen. Procesor 6502 s okolím komunikuje pomocí jedné systémové sběrnice, která obsahuje 16 adresních vodičů, 8 datových vodičů a signál R/W, který signalizuje zdroj dat vzhledem k procesoru. Logická jednička (napětí větší než 2,4 V) signalizuje čtení procesorem, logická nula pak zápis procesorem. Okolí procesoru je tvořeno několika komponentami, ty jsou součástí adresního prostoru CPU, který je znázorněn v tabulce 2.1.

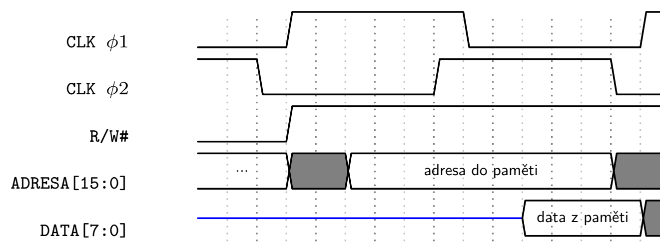
Adresní prostor je v terminologii produktové řady MCS650X rozdělen na stránky, což představuje rozsah adresovatelný jedním bajtem (256 adres). Index v rámci stránky zajišťuje spodní bajt, index stránky poté horní bajt. Z toho vyplývá, že stránek je také 256. Znalost tohoto faktu je klíčová pro pochopení důvodu existence speciálního zero-page adresního režimu, který je vysvětlen v podkapitole 2.3.3.2.

Tabulka 2.1 Adresní prostor CPU.

Adresní rozsah	Zařízení
$0000–$07FF	RAM
$0800–$1FFF	zrcadlo $0000–$07FF
$2000–$2007	registry PPU
$2008–$3FFF	zrcadlo $2000–$2007
$4000–$4017	registry APU a I/O
$4018–$401F	nepoužíváno
$4020–$FFFF	Game Pak

Komunikace je řízena dvoufázovými systémovými hodinami. V první fázi se vystaví adresa na sběrnici (předstih), v druhé fázi dochází k přenosu dat. Z těchto dvou fází se pak skládá celý procesorový cyklus [9]. Příklad komunikace směrem k procesoru (čtení) je znázorněn na obrázku 2.5. Dle dokumentace je pro 1MHz hodiny garantováno, že adresa bude stabilní 300 ns po náběžné hraně první fáze; naopak je požadováno, aby data byla platná alespoň 100 ns před sestupnou hranou druhé fáze hodin.

Kromě systémové sběrnice existuje další způsob komunikace, a to je přerušení. Všechny procesory v produktové řadě MCS650X obsahují celkem tři vstupy reprezentující různá přerušení: RST, IRQ a NMI.



Obrázek 2.5 Časování čtení procesoru.

2.3.3 Architektura

Po diskusi vnější komunikace procesoru je vhodné se zabývat jeho architekturou, a nejen instrukční sadou, ale i implementačními detaily tam, kde je to nutné. Architekturou se zabývá především softwarový manuál [20].

Procesor 6502 je osmibitový, jelikož pracuje se slovem o velikosti osmi bitů. ISA procesoru 6502 je střadačově orientovaná, pracovní registr je totiž právě pouze střadač (akumulátor). ISA definuje adresu jako 16bitové číslo ve formátu little-endian, jako první je tedy vždy uváděn nejméně významný bajt (LSB). ISA také definuje několik adresních režimů, z toho jeden speciální, zero-page režim, který slouží jako částečná náhrada absence více registrů a efektivně činí z první paměťové stránky pomyslnou zápisníkovou paměť.

Nejprve je potřeba analyzovat datovou cestu pro pochopení, jaký hardware mají jednotlivé instrukce k dispozici. Dále adresní režimy, jelikož struktura a délka instrukcí a instrukční cyklus s nimi pevně souvisí.

▶ Poznámka 2.4 (Značení adresních bajtů). Jelikož je adresa uváděna po bajtech a obsahuje právě dva bajty, bylo zavedeno v rámci příruček firmy MOS označení ADL (address low) pro nejméně významný (nižší) bajt adresy a ADH (address high) pro nejvíce významný (vyšší) bajt. Tohoto značení se drží i bakalářská práce.

2.3.3.1 Datová cesta

Procesor 6502 obsahuje ve své datové cestě několik registrů. V tabulce 2.2 je uveden popis registrů s velikostmi, často používanými zkratkami a popisem obsahu.

Tabulka 2.2 Registry procesoru 6502.

Registr	Zkratka	Velikost (bit)	Obsah
programový čítač	PC	16	adresa instrukce ke zpracování
střadač (akumulátor)	A	8	zpracovávané hodnoty
ukazatel zásobníku	S	8	adresa vrcholu zásobníku
indexovací registr X	X	8	adresní offset
indexovací registr Y	Y	8	adresní offset
registr příznaků	P	8	výsledky provedení ALU operací a stav CPU

Střadač v ISA 6502 má podobný účel jako v jiné střadačové architektuře. Jedná se o jediný univerzální registr, kde všechny operace (kromě načítání a ukládání, což zvládají i indexovací registry) musí být prováděny přes zásobník. Střadač je implicitním úložným místem pro výsledky operací.

Zásobník architektury 6502 je fixován na adresách $0100–$01FF. Jeho kapacita je tedy 256 bajtů a roste odshora dolů. Se zásobníkem se manipuluje pomocí dedikovaných instrukcí, je možné na zásobník uložit střadač nebo obsah příznakového registru. Adresa vrcholu zásobníku je v registru S.

Indexovací registry slouží primárně jako zdroj offsetu pro adresní režimy s indexací. Fungují jako čítač, existují instrukce pro jejich inkrementaci (INX, INY), dekrementaci (DEX, DEY) a porovnávání hodnot s hodnotou v paměti (CPX, CPY). Jelikož se ale jejich hodnota načítá z paměti a lze do paměti i uložit, mohou sloužit jako programem využitelné pomocné registry.

Příznakový registr obsahuje 7 využívaných příznaků. Jejich popis je v tabulce 2.3. Index označuje pořadí bitu (zprava), do kterého se daný příznak ukládá při použití instrukce PHP. Příznakem, který ve fyzickém registru není implementován, je B. Tento příznak je viditelný pouze při operaci přenosu registru příznaků do zásobníku a jeho hodnota záleží na tom, která operace přenos do zásobníku vyvolala. Přenos je vyvolán dvěma způsoby:

- softwarově (instrukce BRK a PHP): hodnota B je 1,
- hardwarově (přerušení): hodnota B je 0.

Tím, že registr B nemá hardwarovou reprezentaci, je jeho hodnota ignorována při navrácení příznaků ze zásobníků.

Tabulka 2.3 Popis příznakového registru procesoru 6502.

Index	Příznak	Popis
0	C	Operace vygenerovala přenos.
1	Z	Zpracovávaná hodnota je nulová.
2	I	Maska přerušení (hodnota 1: přerušení maskováno).
3	D	Režim BCD (hodnota 1: režim je aktivní).
4	B	Příznak „break“. Neexistuje fyzicky.
5	-	Nepoužito.
6	V	Operace vyvolala přetečení.
7	N	Zpracovávaná hodnota je záporná (má-li sedmý bit má hodnotu 1).

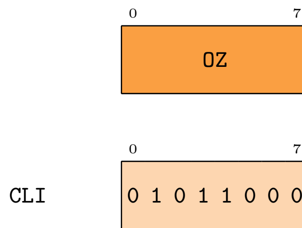
Procesor dále obsahuje pomocné registry pro dočasné ukládání paměti a dat (programově nepřístupné; používané při přístupu na sběrnici). Nedílnou součástí je pak také aritmeticko-logická jednotka, ve které probíhají nejen konkrétní výpočty požadované instrukcemi, ale i pomocné výpočty například pro zjištění absolutní adresy při vyhodnocování skoků.

2.3.3.2 Základní adresní režimy

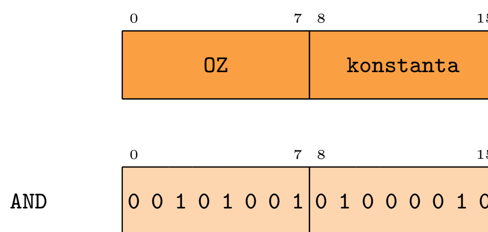
Základní adresní režimy pracují pouze s pevnými adresními hodnotami. Patří mezi ně implikovaný režim, okamžitý režim, absolutní režim, režim nulté stránky a relativní režim.

Nejjednodušší adresní režim se skládá pouze z operačního znaku (OZ, anglicky opcode), který jednoznačně identifikuje příslušnou instrukci. Sama instrukce implikuje, s jakými daty se bude pracovat, proto je tento režim nazván *implikovaný* a taková instrukce má vždy 1 bajt. Struktura a příklad instrukce je znázorněn na obrázku 2.6.

Další adresní režim pracuje s konstantní hodnotou, která je uváděná ihned za operačním znakem. To znamená, že se zpracovávaná hodnota nemusí načítat z paměti pomocí adresy, ale nachází se přímo ve zpracovávaném kódu. Označuje se jako *okamžitý* (immediate) a instrukci tak tvoří dva bajty. Příklad je uveden na obrázku 2.7; instrukce AND provede logický součin hodnoty v akumulátoru s konstantou $42.

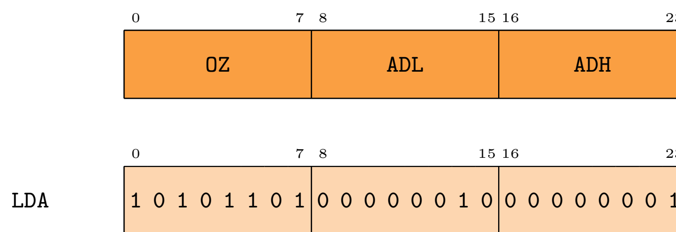


Obrázek 2.6 Struktura instrukce implikovaného adresního režimu s příkladem instrukce Clear Interrupt Disable Bit (CLI).



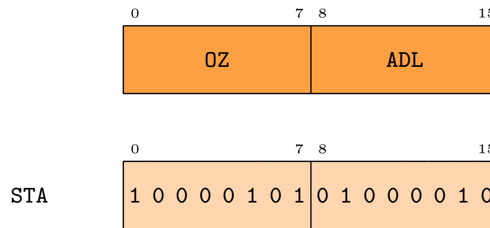
Obrázek 2.7 Struktura instrukce okamžitého adresního režimu s příkladem instrukce AND Memory with Accumulator (AND).

Adresní režim, který již pracuje s hodnotami adres, se označuje jako *absolutní*. Součástí instrukce v tomto režimu je přímá hodnota adresy, kde se nachází kýžená data. Instrukce je tedy 3bajtová. Formát instrukce i s příkladem je na obrázku 2.8; demonstrovaná instrukce načte do akumulátoru hodnotu ze zařízení v adresním prostoru procesoru na adrese $102.



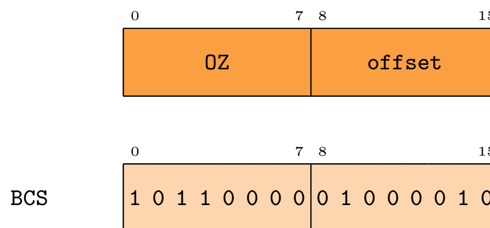
Obrázek 2.8 Struktura instrukce absolutního adresního režimu s příkladem instrukce Load Accumulator (LDA).

Absence univerzálních registrů je částečně suplována existencí adresního režimu *nulté stránky* (zero-page). Adresa, která je uvedena za OZ, je jednobajtová. V tomto režimu je možné indexovat pouze v rámci jedné stránky, a to té první (při indexování od nuly nulté, adresní rozsah $0000–$00FF). Zato však tyto instrukce zabírají dva bajty v instrukční paměti a jejich zpracování je rychlejší. Na nultou paměťovou stránku je tak možné nahlížet jako na formu ručně spravované cache — v programátorském manuálu řady MSC6500 [20] je zdůrazněno, že je program možné optimalizovat přesunem nejčastěji používaných hodnot právě do nulté stránky. Formát je znázorněn na obrázku 2.9; instrukce Store Accumulator načte do akumulátoru bajt z první paměťové stránky s offsetem $42.



Obrázek 2.9 Struktura instrukce adresního režimu nulté stránky s příkladem instrukce Store Accumulator (STA).

Relativní adresování je používáno výlučně instrukcemi větvení. Obsahuje pouze jeden adresní bajt, který reprezentuje offset v dvojkovém doplňku. Vyhodnotí-li se podmínka skoku kladně a skok se tedy provádí, je hodnota offsetu přičítána k adrese následující instrukce (než se podmínka skoku vyhodnotí, nachází se již programový čítač na další adrese). V JSA obecně není nutné uvádět offset explicitně, uvádí se konkrétní adresa, nebo návěští; pomocí těchto údajů je assembler schopen výsledný offset dopočítat. Režim je demonstrován na instrukci Branch On Carry Set na obrázku 2.10 s offsetem $42.



Obrázek 2.10 Struktura instrukce relativního adresování s příkladem instrukce Branch On Carry Set (BCS).

2.3.3.3 Adresní režimy s indexací

Složitější adresní režimy přinášejí další možnosti přístupu k datům v paměti. Do této chvíle byly uvedeny pouze takové režimy, které disponují pouze pevně stanovenou adresou. Často je však nutné adresy měnit, nebo vytvářet zcela dynamicky. Takový typ adres označuje manuál [20] jako počítané adresy. Pro práci s počítanými adresami obsahuje ISA speciální adresní režimy využívající indexovací registry: absolutní režim s indexací a režim nulté stránky s indexací.

▶ **Příklad 2.5** (Kopírování souvislých dat bez indexace)**.** Jedním ze základních řídicích struktur programovacích jazyků jsou cykly, které mohou posloužit jako nástroj pro práci s bloky dat. Typickým příkladem nechť je kopírování dat z jednoho paměťového místa na jiné. Ve střadačové architektuře se provádí načtením do střadače a uložením.

Byla-li by implementace provedena pouze za použití pevných adres, musela by se pro každé paměťové místo uvést instrukce načtení i instrukce zápisu. Vytváří-li se adresy dynamicky, je možné provést operace v cyklu. Ačkoliv existuje způsob úpravy pevných adres za běhu programu pomocí techniky samomodifikujícího se kódu (viz stranu 72 manuálu [20]), představují adresní režimy s indexací elegantnější alternativu nevyžadující přepisovatelnou instrukční paměť.

Nechť je jako první uveden *absolutní adresní režim s indexací*. Tento režim přidává absolutnímu režimu možnost přičíst k původní adrese i offset z registru X, nebo Y; dle zvoleného

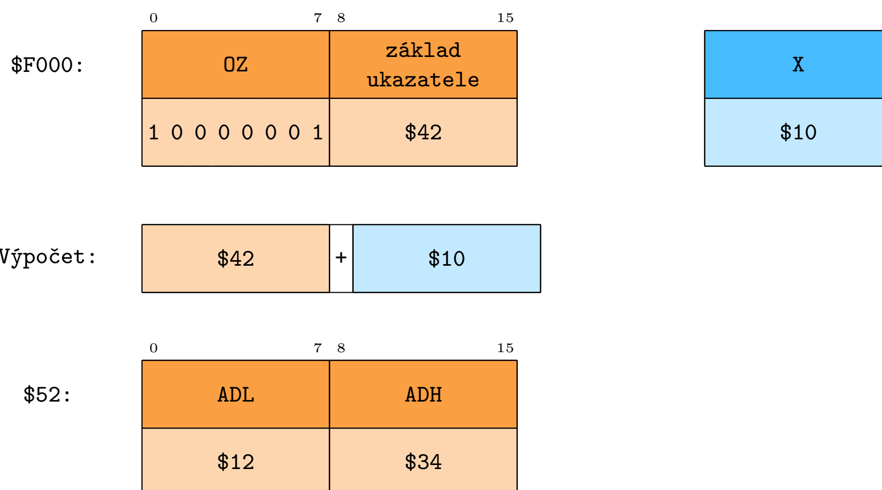
registru se tak jedná o dva různé adresní režimy. Adresa pevně určená instrukcí je nazývaná jako základová (base). Výsledná adresa je vypočtena jednoduše součtem základové adresy a offsetu. Struktura je stejná jako u standardního absolutního režimu na obrázku 2.8.

Podobně jako u standardních režimů existuje možnost pracovat pouze s nultou stránkou. K tomu účelu slouží *režim nulté stránky s indexací*. Struktura instrukce je opět stejná jako na obrázku 2.9. V tomto režimu nedochází k překročení paměťových stránek při přičtení indexu, horní bajt je ignorován a výpočty tak efektivně probíhají v modulu $100. Výpočet tedy probíhá jako: $(\text{zakladova_adresa} + \text{offset}) \mod 256$. Kromě instrukcí LDX a STX, kdy jsou k dispozici oba indexovací registry k výběru, je tento režim použitelný pouze s registrem X.

2.3.3.4 Nepřímé adresování

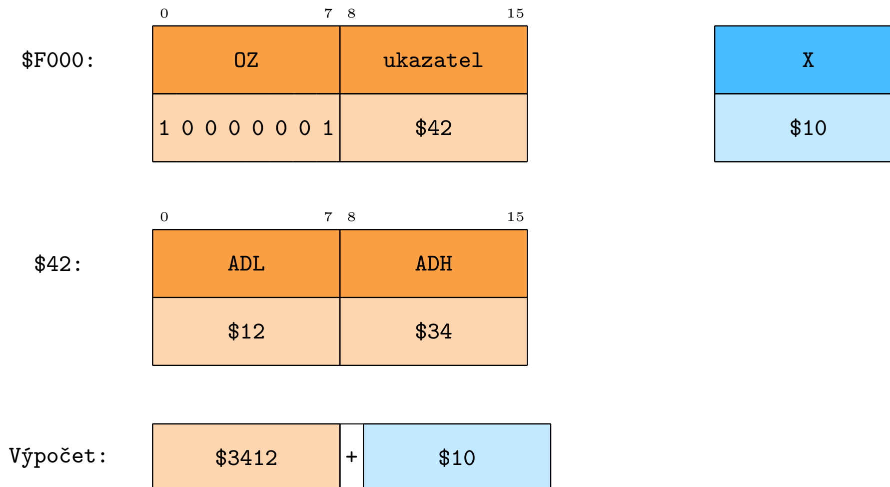
V architektuře 6502 existuje ještě *nepřímé adresování*, které umožňuje pracovat s ukazateli namísto s přímou hodnotou adresy. Takové režimy jsou dva, indexovaný-nepřímý a nepřímý-indexovaný. Názvy režimů jsou odvozené podle pořadí, ve kterém se přičítá index.

První zmíněný režim, *indexovaný-nepřímý*, pracuje s jednobajtovou adresou následovanou po operačním znaku. Tato adresa je základ ukazatele do nulté stránky. K základu se přičte hodnota indexovacího registru X. Součet probíhá opět v modulu 256, vyšší bajt je totiž zahazován. Vznikne tak výsledný ukazatel, který směřuje na místo nacházející se v nulté stránce, které obsahuje první bajt kýžené adresy. Ta je opět uspořádána ve formátu nižší bajt a vyšší bajt. Příklad na obrázku 2.11 ukazuje variantu instrukce STA v indexovaném-nepřímém režimu. Samotný OZ se základem ukazatele je umístěn v instrukční pamětí na adrese $F000. K základu $42 je přičten obsah registru X $10. Výsledkem je adresa $52, na které se již nachází 16bitová konečná adresa. Instrukce STA v popsaném příkladě tedy uloží hodnotu akumulátoru až na adresu zjištěnou v posledním kroku: $3412.



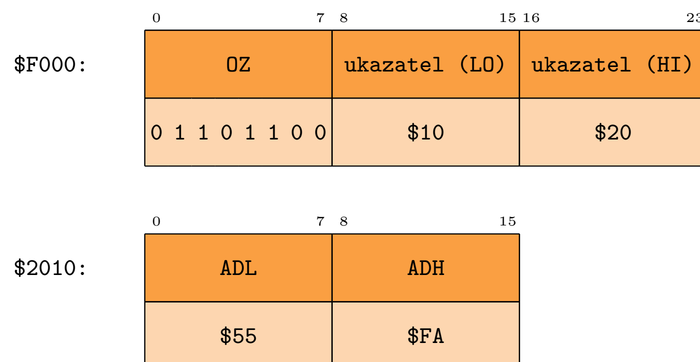
Obrázek 2.11 Struktura instrukce indexovaného-nepřímého režimu s příkladem instrukce STA.

Druhý režim, *nepřímý-indexovaný*, funguje podobně, ale přičítání indexu probíhá až v druhém kroku a používá se registr Y. Po operačním znaku následuje hodnota ukazatele, která se již nemění. Ukazuje do nulté stránky, kde se nachází dvojbajtový základ konečné adresy. K základu se přičte hodnota registru Y. Vznikne tak konečná adresa, se kterou může daná instrukce dále pracovat. Příklad opět na instrukci STA je uveden na obrázku 2.12. Tentokrát bude hodnota akumulátoru uložena na adresu $3422.



Obrázek 2.12 Struktura instrukce nepřímého-indexovaného režimu s příkladem instrukce STA.

Existuje ještě jeden nepřímý režim, označovaný jako *nepřímý absolutní*. Tento režim je použit pouze instrukcí skoku (JMP). Princip je podobný jako u zmíněných nepřímých režimů s tím rozdílem, že nedochází k přičítání indexu. Instrukce se skládá z operačního znaku a dvou adresních bajtů, které fungují jako 16bitový ukazatel na výslednou adresu, na kterou se má skočit. V příkladu na obrázku 2.13 instrukce JMP skočí na adresu uloženou na adrese $2010, jejíž hodnota je $FA55 — skok bude tedy proveden na adresu $FA55.

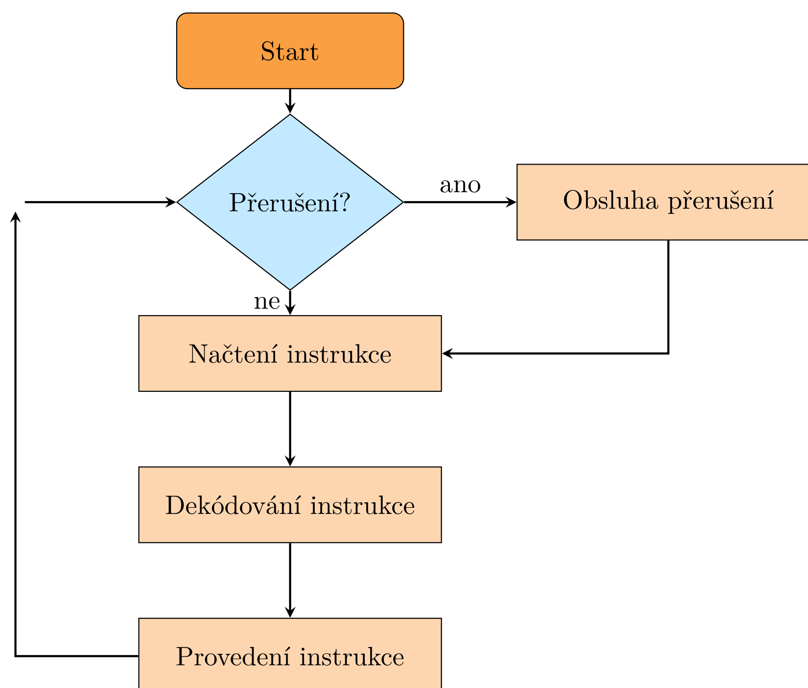


Obrázek 2.13 Struktura instrukce JMP využívající nepřímý absolutní adresní režim s příkladem skoku na adresu $FA55.

▶ **Příklad 2.6** (Využití nepřímého adresování)**.** Nepřímé adresování má mnoho různých využití. U indexovaného-nepřímého se jedná především o práci se seznamem adres. Tento seznam může uchovávat například adresy tlačítek herních ovladačů, ze kterých se periodicky vyčítá stav. Nepřímý-indexovaný pak může sloužit k modifikaci chování volaného podprogramu. Úpravou hodnoty registru Y bude podprogram pracovat s jiným offsetem v odkazované paměti.

2.3.3.5 Instrukční cyklus

Instrukční cyklus odpovídá standardnímu cyklu procesoru, jako uvádí prof. Kubátová v [6]; fetch (načtení instrukce), decode (dekódování instrukce), execute (provedení instrukce). Dekódování je věnována zvláštní pozornost v podkapitole 2.3.3.6. Standardně je součástí běhu navíc počáteční nastavení při spuštění a ošetření příčiny přerušení; právě tomuto je věnována zvláštní pozornost v podkapitolách 2.3.3.9 a 2.3.3.10.



Obrázek 2.14 Instrukční cyklus procesoru.

Přesné dodržení instrukčního cyklu nemusí být u emulace nutné, dokonce bývá zvykem velkou část cyklu zjednodušit pro dosažení většího výkonu tak, že se pracuje pouze na úrovni instrukcí. Jelikož je ale v systému komponent více, je nutné se zabývat alespoň délkou trvání jednotlivých instrukcí, aby nedocházelo ke ztrátě synchronizace.

Délka zpracování instrukce je vázána především na její typ a podtyp — pohybuje se od 2 do 7 (8 v případě nedokumentovaných instrukcí) strojových cyklů. Instrukce totiž existují ve více variantách v závislosti na použitém adresním režimu. Každý podtyp je jednoznačně identifikován operačním znakem. To znamená, že pro stejnou instrukci existuje více různých operačních znaků, lišících se pouze v adresním režimu. Je tedy možné délku zpracování odvodit pouze na základě operačního znaku.

Existují však výjimky. U některých instrukcí pracujících s pamětí může dojít k překročení paměťové stránky, což způsobí provedení jednoho strojového cyklu navíc. To se týká adresních režimů pracujících s indexy, konkrétně absolutního režimu s indexací pro registry X i Y. Týká se to i nepřímého-indexovaného režimu, ten totiž také přičítá hodnotu registru k 16bitové adrese s možností překročení stránky. Indexovaný-nepřímý režim již nedovoluje překročení paměťové stránky (horní bajt je vždy ignorován a k přenosu do vyššího řádu tak nedochází); jej se to již tedy netýká.

Poslední výjimkou jsou instrukce větvení. Dojde-li ke kladnému vyhodnocení podmínky, musí se vykonat jeden strojový cyklus navíc; to je způsobeno nutností přičíst offset k programovému čítači. Dojde-li navíc k překročení paměťové stránky, musí se kvůli přičtení přenosu provést

ještě další strojový cyklus. Abych to shrnul, je-li proveden skok, který navíc překračuje stránku, provedou se navíc celkem dva strojové cykly.

► Poznámka 2.7 (6502 a pipelining). Procesor 6502 zároveň provádí více činností — operace na vnitřní sběrnici provádí souběžně s operacemi na vnější sběrnici. Zatímco na vnější sběrnici se připravuje operační znak a příslušné adresy, na vnitřní probíhají výpočty (inkrementace programového čítače, operace na aritmeticko-logické jednotce a další). Díky pipeliningu je možné instrukce zpracovat v méně cyklech. Pro účely emulace to však není relevantní a zájemci si mohou o problematice více přečíst v manuálu [20].

Podrobný rozpis vykonávaných úkonů při zpracování instrukcí je s přesností na strojové cykly uveden v příloze A hardwarového manuálu [9].

2.3.3.6 Dekódování instrukcí

Každá instrukce je jednoznačně identifikována včetně příslušného adresního režimu pomocí operačního znaku. V případě 6502 je tento operační znak vždy 8bitový. Každá instrukce je prováděna v několika strojových cyklech. Index strojového cyklu je uchováván v čítači a v dokumentaci je označován písmem T s indexem (například T1 pro první cyklus). Dvojice operační znak a index jsou dekódovány v programovatelném logickém poli (Programmable Logic Array, PLA). Právě použití PLA způsobilo existenci takzvaných neoficiálních instrukcí, viz podkapitola 2.3.3.8.

PLA slouží k efektivní implementaci kombinační logiky. Oproti klasické paměti PLA nevyžaduje záznam pro každou možnou adresu, výstupy jsou řízeny hradly AND a OR, které implementují logickou funkci v disjunktivní normální formě [21]. Takto je možné efektivně zakódovat funkci obsahující hodnoty „don't care“, aniž by muselo dojít k duplikaci řádků. Právě díky hodnotám don't care může jeden záznam reagovat na více různých vstupů, v případě řadiče 6502 jsou vstupem operační kódy. Mají-li se stejná operace vykonat pro více operačních kódu, je to implementováno právě takto.

Příklad několika záznamů je uveden v tabulce 2.4, kde například operační znak $AC ve strojovém cyklu T3 způsobí načtení hodnoty ze sběrnice do registru Y. Operační znak $AF však vlivem hodnot don't care způsobí načtení do dvou registrů: A a X. Operační znaky s více funkcemi jsou neoficiální a jejich bližší popis je v podkapitole 2.3.3.8.

Tabulka 2.4 Ukázka záznamů v PLA procesoru 6502 [22].

Maska	Cyklus	Popis
`10101100`	T3	Načti do registru Y.
`101011X1`	T3	Načti do registru A.
`1010111X`	T3	Načti do registru X.

2.3.3.7 Instrukční sada

Instrukční sadu procesoru je možné rozdělit do několika skupin dle jejich účelu. Každá instrukce se pak dělí dle adresních režimů, které používá; to určuje i délku vykonávání ve strojových cyklech a také říká, zdali dojde při překročení paměťové stránky k vykonání dalšího strojového cyklu. Nakonec je u každé instrukce známo, jaké příznaky procesoru nastavuje. Instrukce má pro použití v JSA symbolické textové označení, které se označuje jako mnemonika (mnemonic); příklady takových označení již byly uvedeny u adresních režimů, například STA pro Store Accumulator. Instrukční sada je popsána nejen v oficiálních manuálech, ale i v mnoha přehledných dokumentech, které jsou k dispozici na webu; tato kapitola je tedy věnována analýze formátu takových dokumentů a ukázce pár instrukcí.

Popis instrukcí většinou bývá ve formě tabulky. První tabulkou je přehled operačních znaků, kde řádky odpovídají vyšším čtyřem bitům OZ a sloupce nižším bitům. Další tabulka je poté věnována popisům jednotlivých instrukcí. Příkladem je tabulka 2.5, ve které je mnemonika instrukce, seznam příznaků a stručný popis.

Tabulka 2.5 Ukázka popisu instrukcí logických operací.

Instrukce	N	Z	C	I	D	V	Popis
AND	✓	✓	-	-	-	-	Provede logický součin střadače a hodnoty z paměti.
EOR	✓	✓	-	-	-	-	Provede úplnou disjunkci (XOR) střadače a hodnoty z paměti.
ORA	✓	✓	-	-	-	-	Provede disjunkci střadače a hodnoty z paměti.

2.3.3.8 Rozšířená instrukční sada

Kromě oficiálních instrukcí, tedy takových, které byly popsány v oficiálních manuálech od výrobce, existuje i „skrytá“ množina instrukcí, označována jako neoficiální nebo nelegální. Tyto instrukce nebyly brány v potaz při návrhu a jejich existence je důsledkem principu, který byl použit pro dekódování instrukcí (dekódování probíhá v druhém kroku instrukčního cyklu, viz podkapitola 2.3.3.5). Nejprve je tedy vhodné pochopit, jak funguje dekódování instrukcí u procesoru 6502, což vysvětluje podkapitola 2.3.3.6.

Některé instrukce jsou prostou kombinací více instrukcí. Tyto instrukce bývají stabilní a používají je i některé oficiální hry. Například hra *Disney's Aladdin* [23] z roku 1994 používá instrukci SLO, která kombinuje ASL (aritmetický posuv vlevo) a ORA (disjunkce střadače s paměťovou hodnotou). Dále je často používána instrukce LAX, která načte hodnotu do akumulátoru i do registru X. Příklad, jak taková instrukce funguje, je popsán v podkapitole 2.3.3.6.

Další instrukce nastavují příznaky zvláštním způsobem, popřípadě jsou nestabilní (výsledek je dán fyzikálními jevy, například teplotou čipu). Poslední skupina instrukcí zastaví instrukční cyklus a je vyžadován restart. Tyto instrukce jsou označovány jako JAM, KIL, HLT. Zastavení instrukčního cyklu je způsobeno nenastavením čítače taktů zpět na první index, procesor se tak zacyklí.

2.3.3.9 Počáteční stav

Má-li být emulace věrohodná, je třeba ctít stav po restartu zařízení. Tabulka 2.6 popisuje obsah registrů po restartu. Nejdůležitější je hodnota programového čítače, bez korektního nastavení jeho hodnoty se začne program vykonávat z nesprávné adresy.

Po spuštění dojde k inicializaci o 7 procesorových cyklech, nakonec se provede instrukce skoku (JMP) na adresu nacházející se ve vektoru resetu $FFFC–$FFFD.

Tabulka 2.6 Stav registrů 6502 po restartu.

Registr	Hodnota
P	$34
A	0
X	0
Y	0
PC	dle vektoru na adresách $FFFC–$FFFD

2.3.3.10 Obsluha přerušení

Přerušení se dá použít mimo jiné pro synchronizaci systémových komponent. Toho využívá například PPU pro oznámení procesoru, že bylo dosaženo konce zobrazitelné oblasti obrazovky. Proto je podrobná analýza obsluh přerušení důležitá. Procesor 6502 disponuje dvěma typy přerušení: NMI a IRQ (BRK). Tabulka 2.7 ukazuje vektory jednotlivých přerušení, pro úplnost je uveden i vektor resetu vyvolatelného pinem RST.

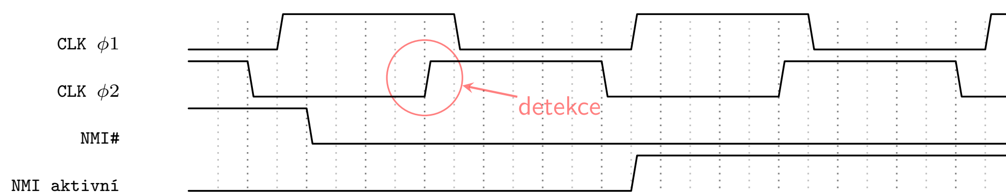
Tabulka 2.7 Vektory přerušení a resetu.

Přerušení	Vektor
NMI	\$FFFA–\$FFFB
IRQ, BRK	\$FFFE–\$FFFF
Reset	\$FFFC–\$FFFD

Přerušení se liší pouze způsobem vyvolání, postup po aktivaci vnitřního signálu je poté stejný. Vždy se dokončí právě probíhající instrukce, pak se na zásobník zálohuje hodnota registrů PC a P, dojde ke změně hodnoty PC na adresu nacházející ve vektoru přerušení pro daný typ a nakonec se nastaví příznak masky přerušení. Poté se začne zpracovávat první instrukce rutiny přerušení. Z přerušení se vystoupí instrukcí RTI (return from interrupt), která obnoví uložené hodnoty ze zásobníku a navrátí se k původnímu toku programu.

NMI

Přerušení NMI (non-maskable interrupt) je nemaskovatelné a vyvolává se stejnojmenným pinem. Časový diagram detekce je na obrázku 2.15. K detekci dochází v každé druhé fázi strojového cyklu pomocí hranového detektoru. Signál NMI je aktivní v logické nule, tudíž k detekci dochází, pakliže byla hodnota v minulém cyklu 1 a současná je 0. Dojde-li k detekci, je v první fázi následujícího strojového cyklu procesoru aktivován vnitřní signálu, v diagramu označen jako *NMI aktivní*. Tento signál zůstává aktivní do doby, než dojde ke zpracování [24].



Obrázek 2.15 Detekce přerušení NMI.

IRQ

Přerušení IRQ (interrupt request) se dá vyvolat buďto hardwarově stejnojmenným pinem, nebo softwarově instrukcí BRK.

Hardwarová implementace je zajištěna úrovňovým detektorem a vyvolání probíhá stejně jako u NMI s tím rozdílem, že vnitřní signál zůstává aktivní pouze v cyklu, ve kterém byl vyvolán. Poté dojde k další kontrole stavu vnějšího signálu NMI a není-li v logické nule, vnitřní signál se deaktivuje [24].

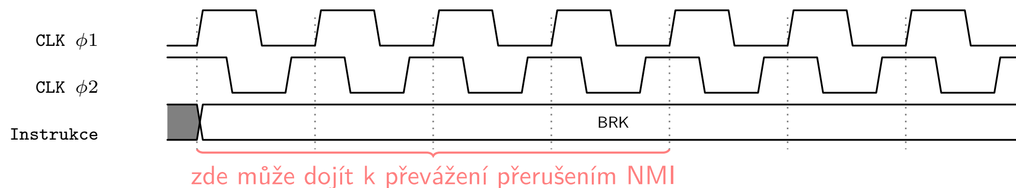
Softwarové přerušení pomocí instrukce BRK provede podobný sled kroků jako při vyvolání hardwarovém s tím rozdílem, že není automaticky nastaven příznak maska přerušení (I).

Zvláštnosti

Reakce na přerušení a jeho zpracování je jedním z příkladů, kde se není možné vyhnout emulaci na úrovni strojových cyklů.

První zvláštností je zpožděná reakce IRQ u instrukcí manipulujících s maskou přerušení (NMI se to netýká; to je vyvoláno vždy nehledě na masku přerušení). Dvojcyklové instrukce nejprve vyhodnotí stav přerušení a poté mění masku. Instrukce pro vymazání masky (CLI) a přepis příznakového registru hodnotou ze zásobníku (PLP) tedy potenciální čekající přerušení odsunou až do další instrukce. Naopak instrukce RTI nejprve smaže masku a poté kontroluje, zdali není očekávané přerušení IRQ; může tedy k opětovnému skoku do rutiny přerušení dojít už po dokončení RTI [24].

Další zvláštností je převážení přerušení (komunitně označováno jako hijacking; únos), které je dáno pseudo-prioritou přerušení: NMI, IRQ, BRK. Dojde-li při prvních čtyřech cyklech zpracovávání instrukce BRK k vyvolání NMI, bude PC nastaven na hodnotu vektoru NMI, namísto očekávaného vektoru BRK (který je společný s IRQ). Graficky je chování znázorněno na obrázku 2.16. Stejně může dojít ke změně vektoru při souběhu IRQ a NMI. Podobné chování vykazuje i souběh IRQ a BRK, ale vzhledem ke stejnému použitému vektoru se to na zpracování programu nikterak neprojevuje, což tento souběh činí irelevantním pro emulační účely [24].



Obrázek 2.16 Převážení přerušení BRK přerušením NMI.

2.3.4 Specifika Ricoh 2A03

Aby se zabránilo porušení patentu, v hardwaru se přerušilo několik cest tak, aby se učinil desítkový režim procesoru nefunkční. Zároveň byl procesor doplněn o zvukový syntezátor, který je popisován v podkapitole 2.7, a řadič DMA.

Řadič DMA existuje v 2A03 pouze pro dva účely. Prvním je kopírování obrazových dat pro sprajty využívané komponentou PPU, druhým je kopírování surových zvukových dat komponentou APU. Jelikož bude v emulátoru implementován pouze první typ, následuje jen popis této funkcionality.

Podtyp DMA pro kopírování sprajtů se označuje jako OAM DMA. Spouští se zápisem do paměťově mapovaného registru procesoru na adrese $4014. Do registru se zapíše číslo paměťové stránky, která se poté celá překopíruje (256 bajtů) do vnitřní paměti PPU přes registr PPU $2004 (OAM DATA). Tento zápis trvá 513 cyklů (v případě jiného zarovnání s procesorem 514, to je ale náhodné po spuštění, tudíž irelevantní) [25].

▶ Poznámka 2.8 (Důvod existence OAM DMA). Data pro sprajty je možné zapisovat i procesorovým přístupem do registru OAM DATA, což je dobře využitelné pro obnovu části paměti. Avšak vzhledem k tomu, že na přepisování paměti OAM je jen omezený čas (doba prodlevy mezi snímky; vertical blanking), bylo nutné implementovat rychlejší alternativu pro případ nutnosti přepsání celé paměti OAM.

2.4 Grafický čip 2C02

Nedílnou součástí herní konzole je grafický výstup, tu zajišťuje čip 2C02, přezdívaný jako Picture Processing Unit (PPU). 2C02 je grafickým čipem, který ve své podstatě operuje vždy nad surovými daty, pouze s nimi umožňuje programátorovi efektivně manipulovat — jak pozadí, tak sprajty jsou hardwarově renderovány. Tím, že PPU generuje přímo kompozitní signál, je průběh renderování pevně svázán s rozlišením i použitým televizním standardem. Každý hodinový takt odpovídá jednomu obrazovému bodu na televizní obrazovce.

Existují dvě varianty specifické pro konkrétní trhy v závislosti na typu výstupního signálu — NTSC a PAL. Americká verze konzole (NES), kterou se tato práce zabývá, obsahuje variantu pro standard NTSC. Ačkoliv původní hardware oproti jiným konzolím generoval přímo tento analogový signál [11], není třeba se tímto při tvorbě emulátoru zabývat — pro věrohodnost emulace stačí pouze dodržet rozlišení a časování a generovat obraz v hodnotách RGB, které se i jednoduše zobrazují na monitoru. Případné převedení do analogového signálu lze udělat dodatečně, což se používá ve formě efektů pro „zvěrohodnění“ zážitku z hraní na emulátoru.

Jedná o velice rozsáhlý čip. Analýzu je tedy třeba provést v několika částech. Nejprve se zjistí, s jakými daty čip pracuje, poté jak se tento čip ovládá (rozhraní, které je dostupné procesoru) a nakonec i to, jak s dostupnými daty čip pracuje.

2.4.1 Paměti a data

Jak již bylo zmíněno v úvodu, čip PPU je připojen do dvou sběrnic. Jednak je připojen do hlavní systémové sběrnice, pomocí které procesor 6502 s čipem PPU komunikuje, což je probíráno v podkapitole 2.4.2. Dále je připojen do své vlastní sběrnice, na kterou je připojena i část kazety Game Pak (viz tabulka 2.12 na straně 35). Tabulka 2.8 ukazuje rozdělení adresního prostoru vlastní sběrnice čipu PPU včetně částí zařízení, které jsou běžně v příslušných rozsazích dostupné. Kromě zvenku přístupných pamětí obsahuje PPU ještě mnoho pracovních registrů, které jsou popsány v sekci 2.4.3.

Tabulka 2.8 Adresní prostor vlastní sběrnice PPU.

Rozsah	Popis	Standardně připojeno
\$0000–\$0FFF	pattern table 0	paměť CHR ROM/RAM kazety
\$1000–\$1FFF	pattern table 1	paměť CHR ROM/RAM kazety
\$2000–\$23FF	nametable 0	vestavěná videopaměť PPU / vlastní paměť kazety
\$2400–\$27FF	nametable 1	vestavěná videopaměť PPU / vlastní paměť kazety
\$2800–\$2BFF	nametable 2	vestavěná videopaměť PPU / vlastní paměť kazety
\$2C00–\$2FFF	nametable 3	vestavěná videopaměť PPU / vlastní paměť kazety
\$3000–\$3EFF	zrcadla rozsahu \$2000–\$2EFF	viz zrcadlené oblasti
\$3F00–\$3F1F	paletová RAM	vestavěná paměť PPU
\$3F20–\$3FFF	zrcadla rozsahu \$3F00–\$3F1F	viz zrcadlené oblasti

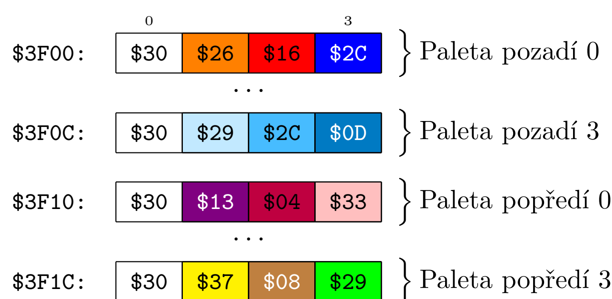
Nakonec je čip PPU připojen zvlášť ještě k paměti OAM obsahující sprajty, viz podkapitola 2.4.1.4.

2.4.1.1 Paletová RAM

Před další diskusí je třeba popsat, jak funguje práce s barvami v konzoli NES. Konzole jako taková má pevně danou množinu barev, kterou může používat. Množina je dána verzí čipu v závislosti na tom, jestli se generuje přímo kompozitní signál, nebo RGB signál, což používaly především arkádové skříňové automaty. Palety lze vygenerovat v závislosti na různých nastaveních například nástrojem od Joela Yliluomy [26].

Třebaže je tato množina barev už tak omezená, není možné ji používat v jednu chvíli celou. Množina aktivních barev je uložena v RAM integrované přímo v čipu PPU dostupná na sběrnici na adresách $3F00–$3F1F. Tato paměť obsahuje čtyři palety pro pozadí a další čtyři pro sprajty. Úplně první barva ($3F00) je také nazývána jako univerzální barva. Každá paleta obsahuje hodnotu tří barev ze zmiňované množiny a dále jednu barvu, která je zrcadlem univerzální barvy pozadí. Obrázek 2.17 ukazuje vnitřní uspořádání paletové paměti s ukázkou možného obsahu. Čtenář si ráčí všimnout, že první barva palety je vždy stejná, totiž taková, která je obsažena na adrese $3F00. Jedná se o zrcadlo tohoto paměťového místa, a tudíž první barvu lze zvolit pouze pro všechny palety stejnou; proto také označení univerzální barva.

▶ Poznámka 2.9 (Přepis barev pozadí). Existuje způsob, jakým lze přinutit čip PPU k přepsání prvních barev z palet pozadí namísto zrcadlení univerzální barvy. Jedná se však o nestandardní postup.

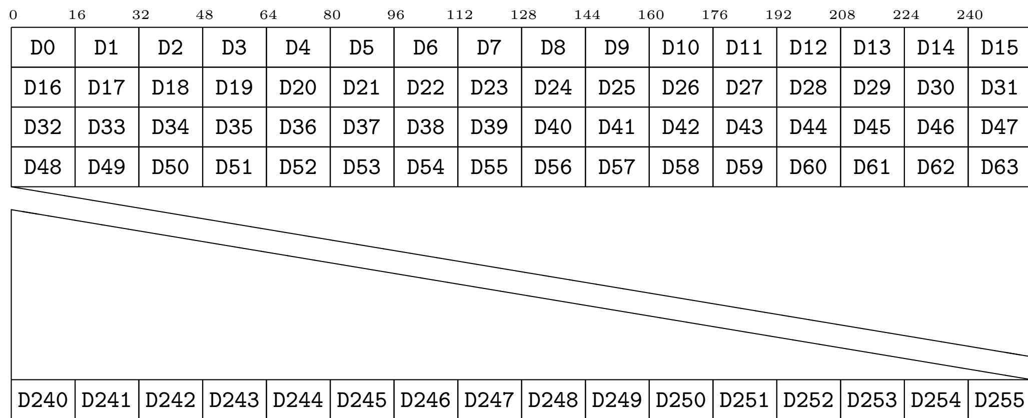


Obrázek 2.17 Část paletové paměti RAM s ukázkou hodnot barev z množiny dostupné pro čip 2C02.

2.4.1.2 Pattern table

Pattern table (volně přeložitelné jako tabulka tvarů) je část paměti typicky obsažená v kazetě ve dvou po sobě jdoucích tabulkách. Zde se ukládají tvary, ze kterých se poté skládá výsledný obraz, ať už v pozadí, nebo ve formě sprajtu. Každá pattern table obsahuje 256 dlaždic. Diagram jedné pattern table je k nahlédnutí na obrázku 2.18, kde jednotlivé dlaždice jsou označeny písmenem „D“ následované indexem dlaždice.

Každá dlaždice je reprezentována dvěma v paměti po sobě jdoucími částmi. Každá část má 8 bajtů, dohromady má tedy dlaždice 16 bajtů. Každý bajt jedné části představuje řádek obrázku; každý bit poté představuje část hodnoty pixelu. Poněvadž se dlaždice skládá ze dvou částí, je pro každý pixel definována dvoubitová hodnota. Dolní bit této hodnoty reprezentuje bit z první části; horní bit pak ten z druhé části. Hodnota pixelu se používá pro výběr barvy z aktivní palety — celkem jsou k dispozici čtyři různé barvy, z toho první barva je pro všechny palety stejná. Princip skládání hodnot pixelu je znázorněn na obrázku 2.19, kde je ukázán i potenciální výsledný obrázek, pokud by se používala paleta pozadí 3 z obrázku 2.17.



Obrázek 2.18 Struktura pattern table čipu PPU.

2.4.1.3 Nametable

Nametable je označení pro část videopaměti, která reprezentuje všechna data potřebná pro vykreslení pozadí na jedné obrazovce (256×240 pixelů). Skládá se ze dvou částí:

- indexy do pattern table (960 bajtů),
- tabulka atributů (64 bajtů).

Celkem tedy jedna nametable zabírá přesně 1 KiB.

První 960bajtová část definuje, z jakých dlaždic se bude skládat obraz na pozadí. Každý bajt je indexem do pattern table. Přeneseně řečeno, nametable slouží tedy vyskládání obrazu dlaždicemi z pattern table. Jak již bylo zmíněno, dlaždice v pattern table je velká 8×8 pixelů, tudíž každý bajt v nametable řídí oblast velikostně odpovídající dlaždici. Pro jeden řádek je tedy potřeba $\frac{256}{8} = 32$ bajtů a takových řádků bude $\frac{240}{8} = 30$

▶ Poznámka 2.10 (Výběr pattern table). Pomocí bajtu lze indexovat v rámci jedné pattern table, ale již nelze vybrat, se kterou se bude pracovat (celkem jsou v systému dvě). Výběr pattern table pro pozadí se provádí v regitru PPUCTRL přístupném procesoru.

V této chvíli je tedy jasné, jaká dlaždice se kam umístí a jak vypadá — každý pixel může nabývat celkem čtyř různých hodnot, tedy čtyř různých barev. Číslo palety, ze které se barvy získávají, určuje právě druhá část nametable; tabulka atributů.

Tabulka atributů se nachází vždy na konci nametable. Každý bajt tabulky určuje indexy palety pozadí pro oblast o velikosti 32×32 pixelů, což odpovídá celkem 16 dlaždicím. Tato oblast je rozdělena na čtyři kvadranty, kdy jsou pro každý kvadrant přiděleny dva bity z příslušného bajtu. Dvěma bity lze přesně indexovat všechny čtyři dostupné palety pro pozadí. Z toho všeho vyplývá, že je celá scéna představovaná nametable rozdělena na čtverce obsahující 2×2 dlaždice, které vždy sdílejí stejnou paletu.

Pořadí indexů v bajtů přísluší jednotlivým kvadrantům v tomto pořadí od nejvyšších dvou bitů: dolní pravý, dolní levý, horní pravý, horní levý.

Ukázka je k dispozici na stránce [27].

2.4.1.4 Object Attribute Memory

Object Attribute Memory, běžně zkracována jako OAM, je zcela oddělená paměť čipu PPU dedikovaná pouze pro ukládání sprajtů. Nachází se mimo obě sběrnice systému, přímo k této paměti může přistupovat pouze PPU, nebo nepřímo přes registry OAMADDR a OAMDATA

část 1 (dolní bity)

0							7
0	1	1	0	0	1	1	0
1	1	0	1	1	1	1	1
1	0	1	1	1	1	1	1
1	1	1	1	1	1	0	1
0	1	1	1	1	0	1	0
0	0	1	1	1	1	0	0
0	0	0	1	1	0	0	0

+

část 2 (horní bity)

0							7
0	1	1	0	0	1	1	0
1	0	1	1	1	1	1	1
1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1
0	1	1	1	1	1	1	0
0	0	1	1	1	1	0	0
0	0	0	1	1	0	0	0

= složené dvoubitové indexy do palety

0	3	3	0	0	3	3	0
3	1	2	3	3	3	3	3
3	2	3	3	3	3	3	3
3	3	3	3	3	3	2	3
0	3	3	3	3	2	3	0
0	0	3	3	3	3	0	0
0	0	0	3	3	0	0	0

výsledný obraz

Obrázek 2.19 Ukázka složení částí dlaždic ve výsledný obraz za použití palety pozadí 3 z obrázku 2.17.

procesor. Paměť OAM má kapacitu 256 bajtů (což odpovídá i velikosti jedné paměťové stránky). Informace příslušící jednomu sprajtu zabírají 4 bajty; tudíž kapacita odpovídá uložení 64 sprajtů.

Struktura sprajtu není nikterak složitá. První bajt je souřadnice Y (řádek); druhý slouží k výběru dlaždice z pattern table, která bude reprezentovat sprajt; třetí obsahuje atributy, kromě výběru indexu palety pro sprajty obsahuje také informace o překlopení dlaždice a prioritu sprajtu; čtvrtý obsahuje souřadnici X (sloupec). Další informace jsou přehledně uvedeny na Nesdev [28].

▶ Poznámka 2.11 (Existence dvou OAM). Kromě výše popsané OAM, označované jako primární, existuje i další, takzvaná sekundární. Má poloviční kapacitu a používá se pouze interně pro ukládání sprajtů patřící na daný obrazový řádek. Programátor k ní tedy nemá přístup a využívá se jen a pouze během procesu renderování.

2.4.2 Vnější rozhraní

PPU je řízeno pomocí osmi registrů mapovaných do paměťového prostoru procesoru. Tyto registry jsou zrcadleny po širokém adresním rozsahu, jak bylo uvedeno v úvodu. Tabulka 2.9 poskytuje přehled a stručný popis. Pro podrobnější informace je možné čtenáře opět odkázat na [25].

2.4.3 Pracovní registry

PPU obsahuje také dvě skupiny pracovních registrů sloužící k průběžnému ukládání vyhodnocovaných a následně vykreslovaných dat, ale i pro přístup na grafickou sběrnici zvenčí. První skupina souvisí s přístupem do videopaměti, což je prováděno nejen programátorem, ale i čipem PPU samotným. Druhá skupina slouží k vykreslování sprajtů.

2.4.3.1 Pozadí a práce s videopamětí

Data pro vykreslování pozadí jsou uložena ve videopaměti (VRAM), se kterou kromě programátora při zápisu do této paměti pracuje i PPU při vykreslování obrazu na obrazovku televizoru. Pro adresaci v rámci paměti se používají dva 15bitové registry, tyto registry jsou sdíleny pro obě aktivity; jak modifikace programátorem, tak čtení samotným čipem. Tabulka 2.10 ukazuje všechny registry používané pro indexaci ve videopaměti a jejich abstrahovaný význam v souvislosti s posuvem souřadnic na obrazovce.

Registr T je přístupný pomocí registrů $2000 (PPUCTRL), $2005 (PPUSCROLL) a $2006 (PPUADDR). Slouží k úpravě programátorem, nepoužívá se pro samotnou indexaci. Do registru PPUCTRL stačí zapsat jednou a přenáší se do T pouze index nametable. Do zbývajících dvou registrů se musí zapsat dvakrát, oba sdílejí stejný přepínač pořadí (W) a oba slouží k úpravě posuvů. Registr PPUSCROLL je zaměřený na úpravu před začátkem vykreslování snímku pro nastavení posuvu v rámci videopaměti (zjednodušeně řečeno: tímto se nastavuje okno do videopaměti, které je viditelné na obrazovce, díky principu zrcadlení paměti je takto možné zobrazit části více nametable zároveň). Registr PPUADDR je poté zaměřen na práci se surovými daty vkládané do paměti programátorem pomocí registru PPUDATA, proto adresu do dočasného registru T vkládá v jiném pořadí a navíc při druhém zápisu rovnou data aktualizuje i v aktivním registru V — počítá se s tím, že programátor bude do videopaměti přistupovat okamžitě.

Registr V již slouží pro přímou indexaci ve videopaměti. Horizontální posuvy jsou aktualizovány registrem T buďto okamžitě při druhém zápisu do registru PPUADDR, nebo je aktualizován průběžně v každém obrazovém bodu 257 každého obrazového řádku. Dále je hodnota posuvu souřadnice Y průběžně aktualizována v obrazovém bodu 256 každého řádku. Vertikální posuvy jsou aktualizovány registrem T poté na konci intervalu VBL v bodech 280–304. Nakonec je hodnota V inkrementována dle nastavení v PPUCTRL vždy při čtení, nebo zápisu do PPUDATA.

Další informace, příklady a podrobnosti lze nalézt na příslušné stránce Nesdev [29].

Tabulka 2.9 Procesorem přístupné registry čipu PPU. Označení „-“ znamená, že se používá celý bajt pro jedinou hodnotu.

Označení	Adresa	Přístup	Bity	Popis bitů
PPUCTRL	$2000	zápis	`VPHB SINN`	`V`: povolení generování NMI
				`P`: režim PPU (0: slave, 1: master)
				`H`: velikost sprajtů (0: 8×8, 1: 8×16)
				`B`: výběr pattern table pro pozadí (0: první, 1: druhá)
				`S`: výběr pattern table pro sprajty (0: první, 1: druhá)
				`I`: způsob inkrementace indexu do videopaměti (0: přičíst 1, 1: přičíst 32)
				`NN`: základ pro index do videopaměti (0: $2000, 1: $2400, 2: $2800, 3: $2C00)
PPUMASK	$2001	zápis	`BGRs bMmG`	`BGR`: zvýraznění barev (jednobitová hodnota pro modrou, zelenou a červenou, 1 zvýraznění aktivuje)
				`s`: povolení sprajtů
				`b`: povolení pozadí
				`M`: 1: zobrazit sprajty v levých 8 pixelech obrazovky, 0: skrýt
				`m`: 1: zobrazit pozadí v levých 8 pixelech obrazovky, 0: skrýt
				`G`: 1: renderovat ve stupních šedi, 0: renderovat barevně
PPUSTATUS	$2002	čtení	`VSO- ----`	`V`: 1: probíhá VBL
				`S`: 1: proběhla kolize sprajtu 0
				`O`: 1: došlo k přetečení sprajtů
OAMADDR	$2003	zápis	-	aktuální adresa v OAM; běžně se nastaví na $00 a poté se použije DMA
OAMDATA	$2004	čtení i zápis	-	čtení dat z OAM, nebo zápis na adresu zvolenou pomocí OAMADDR; zápis autoinkrementuje adresu
PPUSCROLL	$2005	zápis ($2\times$)	-	zápis aktivní adresy na grafické sběrnici (uzpůsobené pro práci s plynulým posuvem)
PPUADDR	$2006	zápis ($2\times$)	-	zápis aktivní adresy na grafické sběrnici
PPUDATA	$2007	čtení i zápis	-	čtení i zápis dat na grafickou sběrnici

Tabulka 2.10 Pracovní registry pro práci s videopamětí.

Označení	Popis	Bity	Popis bitů
V	aktivní adresa do VRAM	`yyy NNYY YYYX XXXX`	`y`: jemný posuv souřadnice Y
			`N`: index nametable ve VRAM
			`Y`: hrubý posuv souřadnice Y
			`X`: hrubý posuv souřadnice X
T	dočasná adresa do VRAM	`yyy NNYY YYYX XXXX`	Stejné jako u registru V.
X	jemný posuv souřadnice X	`xxx`	-
W	indikace pořadí zápisu	`w`	Hodnota 0 pro první, hodnota 1 pro druhý zápis.

Kromě pracovních registrů jsou v čipu obsaženy i posuvné registry, sloužící k ukládání pixelů a metadat aktuálně vykreslovaných. Jsou to dva 16bitové registry pro vzory zkopírované z pattern table a dva 8bitové pro ukládání souvisejících atributů.

2.4.3.2 Sprajty

Pro práci se sprajty kromě hlavní a vedlejší OAM existuje i osm párů 8bitových posuvných registrů uchovávající vzory z pattern table pro 8 sprajtů vykreslovaných na aktivním obrazovém řádku. Dalších osm klopných obvodů uchovává pro tyto sprajty atributy a osm čítačů uchovává horizontální pozice (souřadnice X).

2.4.4 Inicializace po spuštění

Všechny registry u PPU je možné po spuštění vynulovat. Dočasné paměti jako OAM a paletová RAM nemají definovaný stav po spuštění; od programátora se očekává, že tuto paměť korektně inicializuje.

2.4.5 Renderování

Proces renderování obrazu je pevně spjat s jeho průběžným vykreslováním na obrazovku. Po načtení veškerých potřebných dat do pamětí a nastavení čipu pomocí registrů je důležité se zabývat tím, jak čip s dodanými daty pracuje. Přehledný, komunitou vytvořený, diagram celého procesu včetně operací na pozadí je součástí přílohy A.

2.4.5.1 Pozadí

Proces vykreslování pozadí probíhá průběžně během celého televizního snímku. Sekvence kroků se liší dle obrazového řádku. Jelikož jeden hodinový takt odpovídá přesně jednomu obrazovému bodu výstupního signálu, jsou operace PPU pevně svázány s rozlišením obrazu. Název pro časový úsek odpovídající jednomu obrazovému bodu v rámci řádku je *cyklus*.

Na viditelných řádcích (0–239) se provádí sled vnitřních (negenerující výstup) i vnějších (generující výstup) operací. Sled vnitřních operací je uvedený v tabulce 2.11. Pro načtení příslušného

řádku jedné dlaždice z pattern table (dle indexu v nametable) jsou třeba 4 přístupy do paměti, v tabulce jsou rozepsány pro cykly 1–8. Po provedení přístupů potřebných pro jednu dlaždici dojde k naplnění vnitřních posuvných registrů, které mohou generovat samotný výstup na obrazovku, což už se týká vnějších operací. K těm dochází v cyklech 4–256, kdy se generuje obrazový bod za použití bitů z naplněných posuvných registrů. To, jaký konkrétní bit se vybere, závisí na hodnotě vnitřního registru uchovávajícího jemný posuv souřadnice X. Registry vykonají posuv a pokračuje další cyklus. Před výstupem na obrazovku se ještě vybírá mezi pixelem pozadí a popředí, tomu je věnována sekce 2.4.5.3.

Tabulka 2.11 Vnitřní operace PPU během viditelných obrazových řádků.

Cyklus	Operace
0	Žádná.
1–2	Načtení bajtu z nametable; vyhodnocování sprajtů.
3–4	Načtení bajtu z tabulky příznaků; vyhodnocování sprajtů.
5–6	Načtení bajtu z první (spodní) části dlaždice z pattern table; vyhodnocování sprajtů.
7–8	Načtení bajtu z druhé (horní) části dlaždice z pattern table; vyhodnocování sprajtů.
9–256	Stejná posloupnost jako pro cykly 1–8.
257–320	Proces načítání dat pro sprajty.
321–336	Načtení prvních dvou dlaždic pro následující řádek (stejně jako v cyklech 1–8).
337–340	Načítání bajtů z nametable (účel pro PPU je neznámý, některé mappery, například MMC5, tyto přístupy detekují za účelem počítání obrazových řádků).

Po viditelných řádcích následuje řádek 240 (post-renderovací). Od řádku 240 až do předposledního PPU nepřistupuje PPU do videopaměti, avšak příznak intervalu vertikální prodlevy mezi snímky (VBL) je nastaven až v prvním cyklu řádku 241, v ten moment je vyvoláno i NMI. Od této chvíle je tedy programátor informován o možnosti bezpečně přistupovat do videopaměti a může její obsah libovolně upravovat až do řádku 260.

Poslední řádek, označovaný jako 261 (pre-renderovací), provádí stejné paměťové operace, jako se dějí u viditelných, akorát nedochází k vykreslování na obrazovku. Zde se připravují vnitřní registry obsahem patřící na první viditelný řádek.

2.4.5.2 Popředí

Vykreslování sprajtů, tedy obsahu popředí, je u PPU zařízeno hardwarově. Nemusí být tedy vloženy softwarem do průběžně vykreslované videopaměti, stačí správně zavést obrazová data do paměti OAM a nakonfigurovat hardwarové vykreslování.

Na každém řádku nejprve probíhá proces vyhodnocování, kdy se rozhoduje, které sprajty budou vykresleny na řádku následujícím:

1. Inicializace vedlejší OAM hodnotou $FF.
2. Přečtení hlavní OAM a vybrání prvních osm sprajtů, které mají být vykresleny na řádku (dle souřadnice Y, která je obsažena v prvním bajtu každého sprajtu).
3. Kontrola, jestli není v hlavní OAM definováno více než osm aktivních sprajtů na jednom řádku (obsahuje hardwarovou chybu).
4. Inicializace vykreslovací pipeline.

Samotné vykreslování probíhá souběžně s pipeline řešící pozadí. V každém cyklu se dekrementují čítače uchovávající horizontální souřadnice sprajtů pro daný řádek. Je-li nějaký čítač nulový, začne se sprajt postupně vykreslovat pixel po pixelu — dojde k postupnému vysouvání dat z posuvných registrů příslušící danému sprajtu. Je-li aktivních více sprajtů, vybere se ten, který má na výstupu zrovna neprůhledný pixel (nenulová hodnota) a zároveň má nejmenší index v paměti, a pošle se do multiplexeru, kde se vyhodnotí, jestli se vykreslí pixel sprajtu, nebo pozadí, čemuž se věnuje sekce 2.4.5.3.

▶ Poznámka 2.12 (Chyba v inkrementaci). Chyba v kontrole popisovaná v bodu 3 se projevuje chybnou inkrementací. První sprajt po nalezení osmi aktivních sprajtů se vyhodnotí správně, poté ale autoinkrementace způsobí vyhodnocování jiných bajtů namísto toho uchovávající souřadnici Y. Jedná se právě o jednu z chyb, kterou je také nutné emulovat v případě, že je třeba spouštět software, který se na tyto chyby spoléhá (což je ale velice vzácné; většina programů se na špatně definované či chybné chování nespoléhala).

Podrobnější popis je k dispozici na stránce [30].

2.4.5.3 Výběr pixelu k vykreslení (priority)

Těsně před vykreslením se pixel pozadí a pixel popředí (sprajtu) účastní rozhodovacího procesu. Pixel pozadí je vždycky ten, který byl zrovna vysunut z posuvných registrů pozadí. Pixel popředí je pixel patřící takovému sprajtu, který má nejvyšší prioritu dle nejnižšího indexu a zároveň je neprůhledný-nenulový (jinak se zvolí další sprajt, po vyčerpání všech osmi sprajtů jde na výstup nulová hodnota sprajtu). Nebere se ohled na nastavenou prioritu v atributech sprajtu.

▶ Poznámka 2.13 (Zanedbání priorit a z toho plynoucí zvláštnosti). Jak bylo uvedeno, do rozhodování vstupuje pixel sprajtu, který je neprůhledný a má nejnižší index, i když má tento sprajt v atributech nastaveno, že prioritu má pozadí. Může se tedy stát, že přestože v seznamu osmi aktivních sprajtů pro obrazový řádek existuje prioritní sprajt, dojde stejně k vykreslení pozadí, protože se zkrátka priorita sprajtu v moment vstupu do rozhodovacího procesu nebere v potaz.

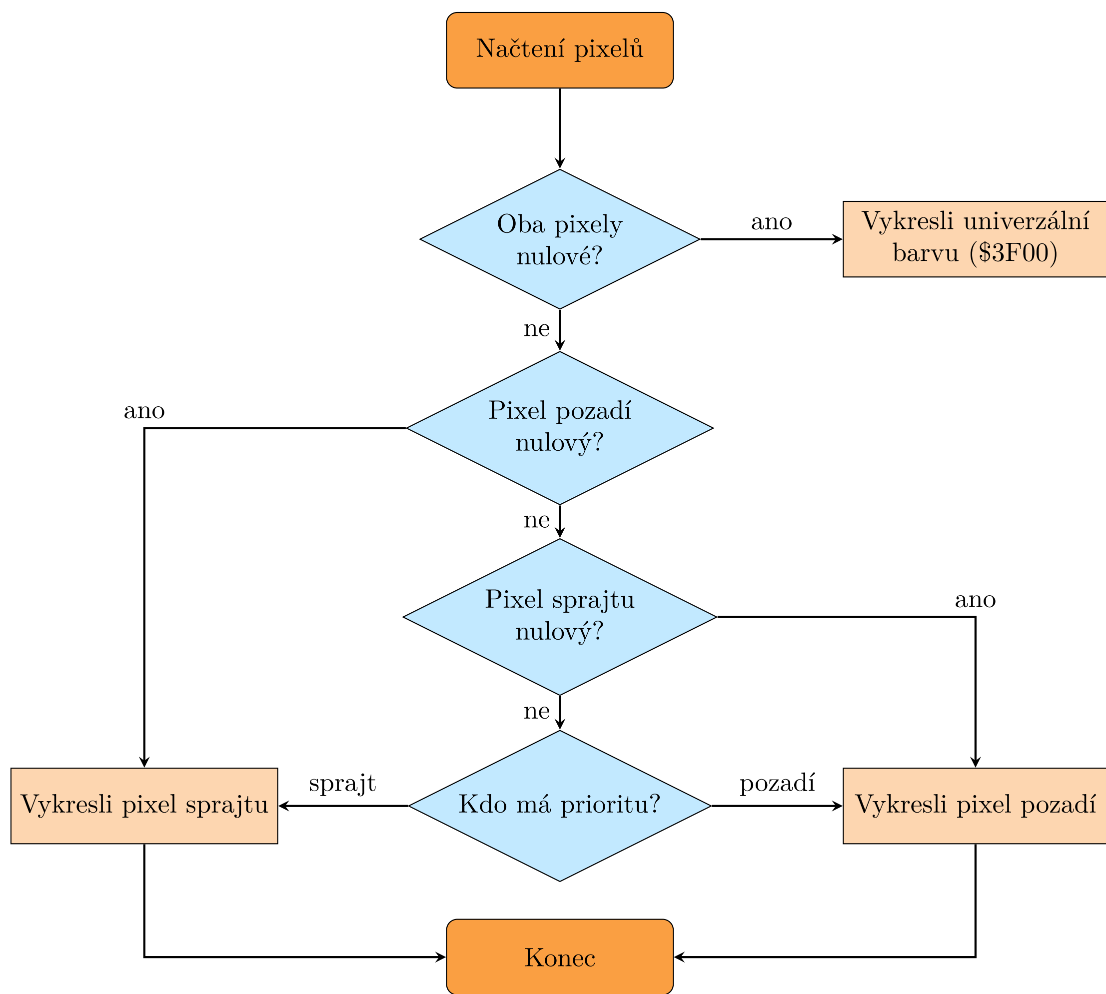
Samotný rozhodovací proces spočívá v několika krocích, kdy se nejprve kontroluje „nulovost" pixelu. Nulová hodnota pixelu totiž indexuje v paletě univerzální barvu pozadí, což je bráno jako transparentní pixel. Není-li žádný z pixelů nulový, vykreslí se pixel dle nastavené priority v atributech sprajtu. Tento proces je graficky znázorněn na obrázku 2.20.

2.5 Game Pak

Hlavním a jediným nativně podporovaným formátem distribuce softwaru pro konzoli NES byly kazety (cartridge) označované jako Game Pak. Tyto kazety nebyly pouze standardní pamětí typu ROM, obsahovaly i často velmi sofistikované obvody především pro mapování paměťových segmentů označované jako mappery, které umožňovaly obejít některé hardwarové limitace konzole. Pro účely emulace je také nutné se zabývat formátem souboru, v jakém se distribuují kopie těchto kazet; bylo totiž nutné uchovat kromě obsahu paměti i informaci o přídavných obvodech na kazetě.

Každá kazeta obsahuje paměť pro program, PRG ROM, volitelně pracovní paměť pro program, PRG RAM. Tyto jsou mapovány do rozsahu procesoru. Dále obsahuje buďto paměť pouze pro čtení s grafickými podklady (CHR ROM), nebo pracovní paměť (CHR RAM), do které se grafické podklady nahrály za běhu. Tyto paměti jsou mapovány do rozsahu PPU. Standardní mapování kazety do rozsahu CPU a PPU je uvedeno v tabulce 2.12. Toto mapování také určuje velikost okna, tedy část paměti, která může být najednou viditelná pro CPU či PPU, pakliže mapper umožňuje přepínat paměťové banky.

U přepisovatelných pamětí není pevně definována výchozí hodnota; korektně naprogramovaný software se na výchozí hodnoty nespoléhá. U emulátorů se však často volí hodnota \$00 či \$FF.



Obrázek 2.20 Proces výběru pixelu k vykreslení.

Tabulka 2.12 Mapování kazety do rozsahu CPU a PPU.

Rozsah	Mapování
CPU	
$6000–$7FFF	PRG RAM (s možností zálohování baterií)
$8000–$BFFF	PRG ROM banka 0
$C000–$FFFF	PRG ROM banka 1 (případně zrcadlo 0)
PPU	
$0000–$0FFF	CHR ROM/RAM (pattern table 0)
$1000–$1FFF	CHR ROM/RAM (pattern table 1)
$2000–$2FFF	(volitelně) vlastní videopaměť

2.5.1 Mapper

Každá kazeta obsahovala řídicí desku, která jednotlivé části paměti umisťovala do paměťových prostorů procesoru a PPU. Jelikož byl výchozí paměťový rozsah nedostačující pro komplexnější hry, umožňovaly sofistikovanější mappery přepínat mezi jednotlivými bankami paměti, což se označuje jako *bankswitching*. Tyto desky byly mnoha typů, jednoduché byly složeny z diskrétních komponent (deska NROM), složitější obsahovaly zákaznické obvody (ASIC), například deska MMC1 a její podtypy (SKROM, SLROM, SNROM. . .).

Další funkcí, kterou desky v kazetách měly, bylo určení, jakým způsobem se bude číst z videopaměti. Jednak bylo možné použít vestavěnou videopaměť PPU a určit, jak se má zrcadlit (pomocí speciálních výstupů CIRAM na kazetě), jednak bylo možné tuto paměť zcela obejít a poskytnout vlastní videopaměť, což umožnilo převzít celou kontrolu nad mapováním. Japonská verze konzole (Famicom) navíc vedla zvukový výstup nejprve přes kazety, tudíž jim umožnila přimíchat další audiokanály pomocí dedikovaných zvukových procesorů [31].

Aby bylo možné desky jednoznačně identifikovat, bylo komunitou vytvořeno označení mapper a deskám začaly být přiřazovány indexy. Mapper může zahrnovat více desek stejného podtypu, které se liší například jen velikostí pamětí. Pro účely emulace tedy dává smysl zabývat se vyhověním specifikace mapperu.

2.5.1.1 Mapper 000: NROM

První přidělený mapper má index 000 a jedná se o oficiální sadu desek společnosti Nintendo, která byla využívána pro jejich vlastní hry i pro hry třetích stran. Jsou to desky NROM, HROM, RROM, RTROM, SROM, STROM. Tyto desky nenabízí žádné možnosti přepínání bank. Obsahují jednu, nebo dvě banky PRG ROM, některé desky i až jednu banku PRG RAM. CHR ROM/RAM je také nepřepínatelná a pevně mapována [32].

Desky používají vestavěnou videopaměť PPU, nepoužívá ani žádné mechanismy změny zrcadlení, toto je nastaveno v hardwaru zaletováním příslušných pinů. Mapper tudíž umožňuje adresovat maximálně 32 KiB paměti programu, 32 KiB pracovní paměti programu a 8 KiB CHR ROM/RAM.

2.5.1.2 Mapper 001: MMC1

Druhý přidělený mapper je také oficiální sada desek Nintenda: SKROM, SLROM, SNROM, SOROM, SUROM, SXROM, SZROM, 2ME a další. Již umožňují přepínat paměťové banky pamětí PRG i CHR. Vše se konfiguruje přes jediný sériový port, který je dostupný procesorem na adresách $8000–$FFFF. Pozorný čtenář může namítnout, že tento rozsah koliduje s mapováním pamětí PRG ROM; ano, je to tak a je to možné díky tomu, že čtení je mapováno do pamětí a zápis do sériového portu [33].

Proces konfigurace mapperu probíhá následujícím způsobem. Procesor zapíše do jednoho z definovaných adresních rozsahů 5 po sobě jdoucích bajtů, které se postupně přenášejí do posuvného registru. Adresa, která byla použita při zápisu posledního bajtu poté rozhodne, do kterého vnitřního registru se posuvný registr zkopíruje. Tyto vnitřní registry jsou čtyři a mají různé funkce, viz tabulka 2.13. Struktura jednotlivých vnitřních registrů je závislá na konkrétním typu desky a je uvedena například ve [33]. Zapisovaná hodnota je jeden bajt, kde funkci mají pouze dva bity; nejvyšší a nejnižší. Zapsáním 1 do nejvyššího bitu se resetuje posuvný registr a do řídicího registru se zapíše hodnota $0C. Nejnižší bit je poté hodnota k zapsání do posuvného registru. Hodnota k zapsání se do sériového portu posílá od nejnižšího bitu po nejvyšší [33].

Díky využití přepínaní bank umožňuje mapper 001 oproti mapperu 000 adresovat až 512 KiB paměti PRG ROM, 32 KiB paměti PRG RAM a 128 KiB paměti CHR ROM/RAM.

Tabulka 2.13 Adresy a popis vnitřních registrů MMC1.

Adresa	Název	Popis
\$8000–\$9FFF	řízení	Nastavení mapperu, režim zrcadlení videopaměti a režim přepínání bank.
\$A000–\$BFFF	index banky v CHR ROM/RAM 0	Nastavení banky, která bude dostupná v adresním rozsahu pattern table 0.
\$C000–\$DFFF	index banky v CHR ROM/RAM 1	Nastavení banky, která bude dostupná v adresním rozsahu pattern table 1.
\$E000–\$FFFF	index banky v PRG ROM	Nastavení banky, která bude dostupná v adresním rozsahu paměti programu CPU.

2.5.2 Formát souboru kopie kazety

První emulátory znamenaly také nutnost vytvoření jednotného formátu, pomocí kterého se budou ukládat obsahy a konfigurace kazet. Prvním formátem, dodnes hojně používaným, je iNES.

Formát iNES se skládá z 16bajtové hlavičky. Následuje volitelný „trainer“, což byly pomocné informace pro rané emulátory, o velikosti 512 B. Poté je uložen jeden nebo více 16KiB bloků PRG ROM a volitelně jeden nebo více 8KiB bloků CHR ROM. Dále mohou být součástí dodatečná data systému PlayChoice, čímž se bakalářská práce nezabývá.

Formát hlavičky je uveden v tabulce 2.14. Jednotlivá struktura je opět velmi dobře dokumentována, proto je pro další informace možné nahlédnout do komunitní dokumentace Nesdev [34].

Tabulka 2.14 Formát hlavičky iNES.

Offset (B)	Obsah
0–3	Konstanta \$4E \$45 \$53 \$1A.
4	Počet bloků PRG ROM.
5	Počet bloků CHR ROM (0 znamená použití RAM).
6	Index mapperu (spodní čtyři bajty), typ zrcadlení, přítomnost baterie, trainer.
7	Index mapperu (horní čtyři bajty), typ PlayChoice.
8–10	Zřídka používané či neoficiální rozšíření.
11–15	Výplň.

2.6 Periferie

Základními periferiemi, které byly dodávány s konzolí, jsou dva herní ovladače (každý pro jednoho hráče). Všechny periferie komunikují přes stejný typ portu. Kromě 5V napájení a hodin obsahuje port tři datové piny označované jako D0, D3 a D4, kde pro ovladače je použit pouze D0.

► Poznámka 2.14. Mimo herních ovladačů existovalo mnoho dalších periferií, hodně známou je například Zapper, což je model zbraně používaný ve hře Duck Hunt. Později se objevily i pokročilejší periferie, například klávesnice pro Family BASIC, Famicom 3D brýle, interaktivní

robot R.O.B., disketová jednotka Family Disk System [35], nebo dokonce síťová karta (pouze pro Famicom) s vlastních procesorem, která sloužila pro přístup k bankovnictví, sázecím programům a dalším [36]. Pokročilejší periferie se připojovaly expanzním portem, který byl k dispozici pouze u japonské verze konzole Family Computer.

2.6.1 Herní ovladače

Ovladače obsahují celkem 8 tlačítek, jak ukazuje ilustrační obrázek 2.21. Výstup z tlačítek je udržován v logické jedničce pomocí rezistoru. Jejich zmáčknutím dojde ke spojení se zemí (GND, 0 V) a na výstupu je tak logická nula.

Obrázek 2.21 Standardní ovladač konzole NES (foto © 2016 Evan-Amos).

Periferie jsou mapovány do paměti; čtení i zápis tedy probíhají stejným způsobem, jako do každého jiného registru. První ovladač je mapován na adresu $4016, druhý na sousední $4017.

Při čtení jsou k dispozici všechny tři piny, které jsou umístěny do jednotlivých bitů dle svého označení, tedy D0 na nejnižším bitu a tak dále. Tabulka 2.15 ukazuje, jakou mají piny funkci.

Tabulka 2.15 Funkce vstupních pinů portů konzole.

Pin	Funkce
D0	Sériový výstup stavů tlačítek ovladače.
D3	Výstup světelného senzoru periferie Zapper.
D4	Stav spouště periferie Zapper.

Stav tlačítek se vyčítá z jediného bitu: D0. Jedná se o sériový výstup osmibitového paralelně-sériového posuvného registru a to, co je na jeho výstupu, záleží na stavu ovládacího klopného obvodu, který je k dispozici pouze na nejnižším bitu adresy $4016 ($4017 je pouze pro čtení). Logickou jedničkou se aktivuje obnova stavu tlačítek do vnitřních registrů a na výstupu je neustále stav jediného tlačítka (A). Přivedením nuly je pak možné postupně vyčíst hodnotu stisknutých tlačítek v následujícím pořadí: A, B, Select, Start, Up, Down, Left, Right. Jelikož stisk tlačítka přivede na výstup logickou nulu, dochází k dodatečné inverzi hodnoty tak, aby stisknuté tlačítko vracelo logickou jedničku a puštěné logickou nulu [37].

Po přečtení tlačítek vrací výstup logickou jedničku. To je způsobeno tím, že na sériový vstup posuvného registru (který se používá pro řetězení více takových posuvných registrů) je přivedeno nízké napětí (logická nula), což po dodatečné inverzi odpovídá logické jedničce. Pro další čtení je nutné opět obnovit stavy tlačítek [37].

Standardní postup při čtení by se tedy dal shrnout takto:

1. Zapsání logické jedničky do bitu 0 na adrese $4016.
2. Zapsání logické nuly do stejného místa.
3. Postupné vyčítání osmi bitů ze sériového portu na adrese dle požadovaného ovladače ($4016 či $4017).

2.7 Zvukový syntezátor Audio Processing Unit

Součástí procesoru Ricoh 2A03 je také zvukový syntezátor, přezdívaný jako Audio Processing Unit (APU).

APU obsahuje celkem čtyři konfigurovatelné syntezátory; dva generující pulzy, jeden generující trojúhelníky a jednu jednotku generující šum. APU také umožňuje přehrávat krátké audiozáznamy ve formátu rozdílové PCM (pulzně-kódové modulaci). Pro analýzu a implementaci byly zvoleny dva typy generátorů: pulzní a šumový. APU se ovládá pomocí registrů přístupných procesoru. Přehled vybraných registrů relevantních k bakalářské práci je uveden v tabulce 2.16.

Tabulka 2.16 Registry APU.

Adresa	Funkce
$4000–$4003	řízení generátoru pulzů 1
$4004–$4007	řízení generátoru pulzů 2
$400C–$400F	řízení generátoru šumu
$4015	stav APU
$4017	čítač rámců

2.7.1 Společné komponenty

2.7.1.1 Čítač rámců

Základem APU je takzvaný čítač rámců, které generuje hodinový signál pro všechny kanály. Navíc je tato jednotka schopná generovat přerušení s frekvencí 60 Hz. Samotný čítač je řízen hodinami CPU dělenými dvěma, tudíž každý druhý cyklus procesoru je poslán hodinový cyklus i do APU. Čítač lze nastavit registrem $4017, který má následující strukturu: `MI------`. Je-li bit `M` roven 0, APU pracuje ve čtyřkrokovém režimu, jinak pracuje v pětikrokovém režimu. Bit `I` je maskou přerušení; je-li tento bit nastaven, poté nebude přerušení vyvoláváno. [38]

Čítač pracuje ve dvou režimech. Režim říká, v kolika krocích bude APU pracovat a co budou tyto kroky zahrnovat. Může pracovat buďto v režimu sestávajícím ze čtyř, nebo z pěti kroků. Krok je vykonán zhruba každých 3728 APU cyklů. Ve čtyřkrokovém řežimu je při každém kroku poslán hodinový signál do první skupiny komponent (generátor obálky a čítač trojúhelníkového generátoru); každý druhý krok je poté poslán hodinový signál do druhé skupiny komponent (čítač délky a regulátor period). V pětikrokovém režimu je hodinový signál do první skupiny zaslán ve všech krocích kromě čtvrtého, do druhé skupiny ve druhém kroku a v kroku pátém. [38]

Podrobnější informace o fungování čítače lze nalézt v [38].

2.7.1.2 Mix

Míchání všech výstupů probíhá v mixu. Každý kanál má vlastní digitálně-analogový převodník, výsledkem je pak nelineární míchací schéma odhadnutelné například následujícím výpočtem převzatým z [39].

$$výstup = suma_pulz + suma_tnd$$

kde jsou jednotlivé složky vyjádřitelné jako

$$suma_pulz = \frac{95,88}{\frac{8128}{pulz1 + pulz2} + 100}$$

$$suma_tnd = \frac{159,79}{\frac{1}{\frac{trojúhelník}{8227} + \frac{šum}{12241} + \frac{pcm}{22638}} + 100}$$

2.7.2 Generátor pulzů

APU obsahuje celkem dva generátory pulzů. Oba se povolují zápisem hodnoty 1 do příslušných bitů ve společném registru $4015; pro první kanál se jedná o nejnižší bit a pro druhý kanál druhý nejnižší bit. Každý z generátorů se skládá z pěti komponent:

- generátor obálky signálu (viz [40]),
- regulátor periody (označován jako sweep unit),
- 11bitový časovač řízený hodinami CPU dělenými dvěma (tudíž o poloviční frekvenci),
- sekvencer (generátor průběhu) pracující v osmi krocích,
- čítač délky (viz [41]).

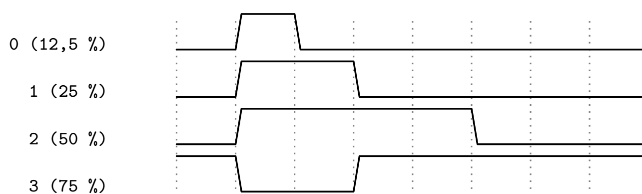
Regulátory periody existují ve dvou jednotkách, každý pulzní kanál má svou jednotku. Ovládají se registry $4001 pro první kanál a $4005 pro druhý. Oba mají stejnou strukturu: `EPPPNSSS`, kde `E` je povolení regulátoru (1 znamená povoleno), `PPP` je perioda děličky regulátoru, `N` určuje, zdali se se perioda přičítá (hodnota 0), nebo odčítá (hodnota 1), `SSS` je počet bitů k posuvu. Je-li regulátor periody aktivní, upravuje hodnotu hlavního časovače v těchto krocích:

1. Načtení hodnoty 11bitového časovače a posuv této hodnoty vpravo dle nastavení (bity `SSS`).
2. Je-li bit `N` roven 1, poté je hodnota negována (první kanál narozdíl od druhého pracuje s jedničkovým doplňkem, tudíž je hodnota ještě o 1 menší).
3. Vypočtená (a volitelně negovaná) hodnota je poté přičtena k hlavnímu časovači.

Regulátor periody má za jistých podmínek schopnost celý kanál vyřadit, což je včetně dalších podrobností fungování dokumentováno na stránce [42].

Třetí komponenta, 11bitový čítač, slouží k taktování sekvenceru. Jeho hodnota je upravována pomocí dvou registrů: $4002 ($4006 pro druhý kanál) slouží k nastavení výchozích spodních osmi bitů časovače; spodní tři bity $4003 ($4007 pro druhý kanál) poté nastavují výchozí horní tři bity časovače. 11bitová výchozí hodnota je perioda časovače. Každý APU hodinový takt je hodnota čítače dekrementována, po dosažení nuly se do čítače načte nakonfigurovaná výchozí hodnota a sekvencer je posunut o 1 krok vpřed. [43]

Sekvencer slouží ke generování průběhu signálu. APU obsahuje vnitřní tabulku o čtyřech řádcích, kdy každý řádek reprezentuje jeden průběh signálu lišící se ve střídě. Jednotlivé hodnoty jsou omezeny pouze na 1 (maximální amplituda) a 0 (nulová amplituda — ztlumení). Index řádku (a tedy hodnotu střídy) lze nakonfigurovat pomocí registru dvou nejvyšších bitů registru $4000 pro první, respektive $4004 pro druhý kanál, což přesně odpovídá čtyřem možným hodnotám [43]. Obrázek 2.22 ukazuje vzhled jednotlivých průběhů dle nakonfigurovaného indexu i s odpovídající hodnotou střídy v procentech.



Obrázek 2.22 Vzhled průběhů signálu pulzního kanálu APU.

▶ Poznámka 2.15 (Průběh signálu se 75% střídou). Signál odpovídající indexu 3 na obrázku 2.22 je oproti ostatním signálům posunut. To je způsobeno tím, že sekvencer je sice inicializován na nultý krok, ale kroky počítá dolů, v tabulce se tedy pohybuje opačným směrem a signál je i takto zvláštně generován [43].

Sekvencer přivádí hodnotu vytvořenou generátorem obálky do mixu pouze tehdy, jsou-li splněny všechny následující podmínky:

- aktuální hodnota kroku sekvenceru je 1,
- regulátor periody neztlumuje signál,
- hodnota čítače délky je vyšší než 0,
- hodnota 11bitového hlavního časovače je méně než 8.

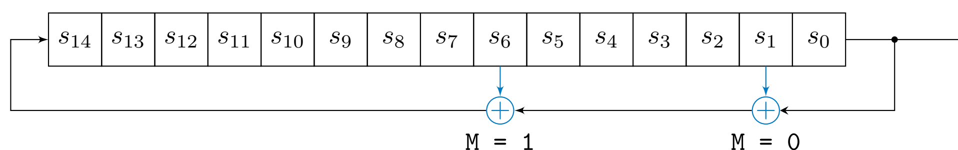
2.7.3 Generátor šumu

Generátor šumu je mnohem jednodušší jednotkou nežli generátor pulzních signálů. Ve své podstatě jde o pseudonáhodný generátor tónů. Skládá se z následujících komponent:

1. generátor obálky signálu (viz [40]),
2. 11bitový hlavní časovač,
3. posuvný registr s lineární zpětnou vazbou,
4. čítač délky (viz [41]).

Jediné dvě konfigurovatelné hodnoty specifické pro generátor šumu jsou nastavení režimu a výchozí hodnoty časovače, obojí pomocí registru $400E s následující strukturou: `M---PPPP`. `M` je režim rozhodující o bitu jdoucím do zpětné vazby, `PPPP` volba výchozí hodnoty (periody) hlavního časovače [44]. Konkrétní hodnota je určena dle tabulky, ta je k nahlédnutí například v [44].

Hlavní časovač funguje podobně jako u pulzního kanálu, po dosažení nulové hodnoty dojde k taktování tentokrát zpětnovazebního registru. Struktura registru je na obrázku 2.23. Je-li nastaven bit M na 1, poté jsou zpětnovazební bity 0 a 6, jinak bity 0 a 1 [44].



Obrázek 2.23 Zpětnovazební registr generátoru šumu.

Generátor šumu přivádí hodnotu vytvořenou generátorem obálky do mixu pouze tehdy, jsou-li splněny všechny tyto podmínky:

- aktuální hodnota nejnižšího bitu posuvného registru je 0,
- hodnota čítače délky je vyšší než 0.

2.8 Existující řešení

Emulátory konzole NES vznikaly již pro platformy dnes nepoužívané, například MS-DOS. Emulátorů NES vzniklo mnoho: pro různé platformy, s různými funkcemi a různými cíli. Přehledný seznam existujících implementací lze nalézt v [45]. Emulátory lze rozdělit do tří skupin.

Precizní výkonné emulátory: cílí především na co nejvěrohodnější napodobení NES a zároveň nízkou náročnost na hardware. Tento typ emulátorů vznikal od samého začátku snah především proto, aby bylo možné spouštět původní programové vybavení pro NES (hlavně zábavní software) na novějších platformách co nejpřesněji s ohledem na původní fungování na NES. Tomu odpovídá i struktura kódu emulátoru. Využívají se různé triky, aby byla emulace věrohodná, ale zároveň nepříliš náročná, což vede k nepřehlednému kódu (nejasné pojmenování funkcí a proměnných, mnoho binárních operací bez dalších popisů. . .). Tím, že kód nebyl psán s ohledem na modularitu, je často i většina komponent silně provázaných. Komponenty nekomunikují přes abstrakci sběrnic a signálních vodičů, ale bývá využívána ještě o stupeň vyšší abstrakce. Dnes již tyto emulátory obsahují kromě prostředí pro samotné hraní i například debugger a jsou používány nejen hráči, ale díky své věrohodnosti i jako simulátory pro vývojáře domácího softwaru pro NES (takzvaný homebrew vývoj). Pro účely bakalářské práce však vzhledem ke stylu napsaného kódu mohou posloužit pouze jako praktické ověření fungování konkrétních her a porovnání s implementací v práci. Velmi věrohodným emulátorem je například fceux dostupný na GitHubu [46].

Experimentální emulátory: projekty (většinou) jednotlivců, kteří se chtějí dozvědět více o fungování NES, popřípadě vytvořit emulátor nestandardním způsobem. Takových emulátorů vzniklo a stále vzniká velké množství. Často jde o nedokončené projekty, kde bylo cílem vytvořit jen prototyp či koncept. Takové emulátory se běžně nepoužívají ani pro hraní her, ani pro vývoj. Narozdíl od první jmenované skupiny již bývá kód přehledný, jsou totiž psány bez vyšších nároků na výkon, naopak s vyššími nároky na přehlednost, mohou být tedy zajímavou inspirací i co se kódu týče. Příkladem je projekt olcNES dostupný na GitHubu [47].

Komerční emulátory. Tyto vznikly pouze pro distribuci konkrétních her na nové platformy přímo společností Nintendo (například acNES pro Game Boy a GameCube, Virtual Console pro Wii, Wii U a 3DS). Jedná se o proprietární aplikace a nebyl pro ně zveřejněn zdrojový kód. Nemá tedy smysl je dále studovat, jelikož jde pouze o komerční zábavní využití.

Bakalářská práce cílí výhradně do druhé jmenované skupiny; experimentální. Nemá smysl se pokoušet vyvíjet nový co nejpřesnější emulátor, takový zájem je mnohem lepší aplikovat příspěvkem do již existujících projektů. Ve skupině experimentálních emulátorů stále chyběl takový, který by se pokusil NES implementovat tak, že struktura kódu odpovídá i reálné implementaci (hlavně co se modularity komponent a komunikace mezi nimi týče). Navíc samotná existence experimentálních emulátorů ukazuje, že implementace emulátoru může být zajímavou příležitostí k prozkoumání prakticky používané počítačové architektury a vzdělání se v této oblasti. Příkladem je emulátor olcNES, kdy autor vytvořil i sadu videí, kde vysvětluje zajímavé principy fungování konzole. Bylo by tedy v zájmu širší skupiny co nejvíce takový experimentální vývoj podpořit a zpřístupnit tak, aby se potenciální vývojáři již mohli soustředit výhradně na implementaci emulovaného systému a jeho komponent. Na základě toho byla do bakalářské práce zakomponována další myšlenka, a to vytvořit univerzální emulační platformu, která zařídí vše, co je sice k běhu emulátoru potřebné, ale přímo se k vývoji emulovaných komponent nevztahuje, a až na této platformě vytvořit emulátor konzole NES.

Kapitola 3
Návrh

„Simplicity is prerequisite for reliability.“

EDSGER W. DIJKSTRA

Při analýze byly získány potřebné informace o tom, z čeho se NES skládá a jak tyto komponenty fungují. Po analyzování systému je třeba navrhnout, jak jej prakticky implementovat. Vzhledem k tomu, že je cílem projektu být co nejsrozumitelnější, je nutné brát v potaz nejen návrh implementace samotného systému, ale i emulační platformy, na které emulovaný systém poběží.

Nejprve je třeba navrhnout technologie, které se pro implementaci využijí. Dále je nutné navrhnout univerzální rozšiřitelnou platformu. Nakonec se za pomocí určených technologií a platformy navrhne, jak implementovat samotnou konzoli NES.

Kapitola slouží jako přehled návrhových rozhodnutí, která vyvstala během řešení praktické části, a jejich odůvodnění.

3.1 Výběr technologií

Na začátku vývoje je třeba vybrat správné nástroje tak, aby byl vývoj co nejefektivnější a nejpohodlnější. Kromě jmenovaných požadavků je nutné brát v potaz také externí požadavky, které stanovuje jednak zadání, jednak výsledky samotného bádání v analytické části.

3.1.1 Programovací jazyk

Zadání požaduje, aby byl emulátor implementován s využitím principů objektově orientovaného programování. Vhodnost k využití ve výuce pak znamená, že je třeba využít rozšířený jazyk, nebo takový jazyk, jehož syntaxe je běžným jazykům podobná. Nepřímo také vyplývá, že by mělo být možné výsledný kód spouštět na co největším množství platforem tak, aby byl emulátor, jakožto vzdělávací pomůcka, snadno dostupný. Nakonec je nutné se při výběru zaměřit na to, že se jedná o implementaci počítačového systému. Zvolený jazyk by tedy neměl poskytovat příliš velkou abstrakci nad počítačovým hardwarem — to by mohlo způsobit odstínění od vysvětlovaných principů.

Zvoleným jazykem pro implementaci je C++ ve verzi C++20. Splňuje totiž veškeré požadavky dané zadáním i ze zadání vyplývajících:

- podpora paradigmatu OOP,

- náklonnost k systémovému programování,
- umožňuje výběr míry abstrakce programátorem,
- podpora mnoha platforem,
- syntaxe podobná jiným rozšířeným C-like jazykům.

3.1.2 Kolekce vývojových nástrojů

▶ **Definice 3.1** (Toolchain)**.** *Toolchain je anglický termín pro kolekci nástrojů využívaných při vývoji. Typicky se jedná o soubor prostředí pro vývoj: textový editor pro psaní zdrojového kódu, kompilátor pro překlad do strojového kódu, linter pro kontrolu syntaktických chyb (dnes existují nástroje i pro hledání různých sémantických chyb, například nástroj clang-tidy), debugger.*

Použití editoru se týká jen programátora, není tedy důležité toto rozhodnutí stanovit před začátkem projektu; nebude mít žádný vliv na výsledný projekt. Je však vhodné vyžadovat editor nabízející zvýrazňování syntaxe a snadnou integrací s dalšími částmi toolchainu, ať už se jedná o pouhý editor, nebo celé integrované vývojové rozhraní (IDE). Pro účely projektu byl mezi dalšími možnostmi jako Visual Studio Code a Qt Creator zvolen CLion (integrace s CMake, přehledný grafický debugger, vestavěný linter).

Při návrhu toolchainu je brána v potaz jednoduchost použití tak, aby šlo pokud možno vše zařídit automaticky: od stažení závislostí, přes kompilaci, testování i generování dokumentace. Zároveň je z hlediska přístupnosti vhodné, aby šel program zkompilovat na více než jedné platformě za pomocí stejného nástroje. Pro popis postupu kompilace se v C/C++ projektech často používá Makefile; ten však není možné jednoduše použít na jiných platformách (například Windows). Pro abstrakci nad nástroji jako Make existuje CMake. Ten se jeví jako mnohem vhodnější řešení, jelikož umí vše vyžadované včetně generování Makefile, projektu pro Visual Studio, kompilování pro WebAssembly a další. Bude tedy zvolen CMake.

3.1.3 Správa zdrojového kódu

Pro udržitelný vývoj většího projektu je vždy dobré mít přehled nad změnami, potažmo jednotlivými verzemi projektu. Možností verzování je několik. Nejjednodušší je prosté ukládání do několika složek v souborovém systému, což ale brzy způsobí chaos a špatně se synchronizují kolizní změny v případě využití synchronizované složky. Standardem je dnes používání systémů správy verzí. Existují dvě možnosti: centralizovaná a distribuovaná. Pro tento projekt byla zvolena distribuovaná, poněvadž umožňuje pracovat off-line a téměř vždy existují alespoň dvě celé kopie repozitáře (lokální a serverová). Konkrétním nástrojem byl zvolen Git, jelikož je standardem a nabízí veškeré očekávané funkcionality distribuovaného systému pro správu verzí: lokální práci s repozitářem, práce s verzemi a vývojovými větvemi, snadná komunikace se serverem, navíc i jednoduché podepisování změň pomocí asymetrických kryptografických nástrojů.

Aby byl projekt v rámci edukativnosti co nejdostupnější, je nutné použít veřejně přístupné úložiště i s možnostmi spolupráce a přijímání změň od dobrovolníků. Velice známou platformou je *GitHub*, který zdarma nabízí nejen umístění repozitáře, ale i různé nástroje pro týmovou spolupráci, například trasování chyb a úkolů. Vhod přijde i možnost přímo na GitHubu jednoduše provozovat on-line dokumentaci.

3.1.4 Dokumentace

Dokumentace je nedílnou součástí projektu. Jednou z možností, jak takovou dokumentaci tvořit, je průběžně vyplňovat externí textový dokument s popisem kódu. Tato možnost je vhodná jen pro velice primitivní programy. Je možné totiž využít možností psaní dokumentace přímo v kódu.

Využívá se funkcionality komentářů, které jsou obohaceny o možnosti vyplňování dalších údajů o jednotlivých funkcích i třídách; takovým komentářům se říká dokumentační. Z takových komentářů lze pak jednoduše dokumentaci vygenerovat; proto je zvolena tato možnost.

Pro konkrétní zvolený programovací jazyk, C++, existuje nástroj *Doxygen*. Ten za pomocí speciální syntaxe v komentářích vygeneruje například i popisy parametrů funkcí, návratových hodnot, popřípadě možných vyvolaných výjimek. Takto vygenerovanou dokumentace již je možné číst lokálně i na serveru. Výstup programu Doxygen lze však dále zpracovat a vytvořit přehlednější dokumentaci, což je velice důležité, pakliže je požadováno využití ve výuce. Pomocí nástrojů *Breathe* a *Exhale* dojde k převedení do formátu podporovaného nástrojem *Sphinx*, který umožňuje přidávat další dokumentační stránky ve formátu *reStructuredText* a použití přehledných šablon (například šablona „ReadTheDocs“ používaná pro mnoho open-source projektů, jmenovitě Admesh, Zoneminder a další). Celý proces generování dokumentace lze automatizovat nástrojem CMake, což ještě zjednoduší průběžné úpravy dokumentace. Z toho důvodu bude implementována také automatizace.

3.1.5 Testování

Aby byl zajištěn soulad s požadavky, je nutné program testovat. Úkolem této části je stanovit požadavky na testy a na tom základu navrhnout vhodné metodiky a nástroje.

Ačkoliv je možné testování provést až na konci, je vhodné *testovat průběžně*, aby bylo možné chyby diagnostikovat *izolovaně*. Mohlo by totiž dojít k situaci, kdy se chyba v jedné komponentě projeví v jiné, což přináší velmi těžce diagnostikovatelné problémy. Není však reálné provádět průběžné testy ručně, cílem tedy je navrhnout *automatické testy*.

Spouštění testů v základní formě vyžaduje akci programátora, což přináší riziko lidského faktoru v podobě zapomenutí, především u jednoduchých změn. Existují však nástroje, které umí i spouštění testů provádět automaticky. Je rozumné se zaměřit především na *testování před přijetím změn* — nedovolit sloučit změny do hlavní vývojové větve v repozitáři před tím, než se otestuje, zdali změny nevnesly nové chyby.

Specifikem bakalářské práce je fakt, že implementuje emulaci systému, od které se očekává, že bude věrně napodobovat emulovaný systém. Dalším požadavkem je proto možnost *spouštět testy ve formě strojového kódu* pro danou platformu s možností automatického vyhodnocení. Pro archaické systémy vzniklo již mnoho testů, které mají za úkol porovnat funkci s reálným systémem. Sofistikované testy umožňují vybrat druh výstupu, kdy součástí bývá i zápis výsledků do vybraného místa v paměti.

Testy se dají provádět ručně i automaticky. Jelikož je požadováno průběžné testování, je nejvhodnější co nejvíce testů automatizovat. To vyžaduje nástroj pro spouštění a řízení testů. Součástí nástroje CMake je CTest, který je využit i v této práci. Pro samotné psaní testů již není třeba další komponenty, avšak využití dalších nástrojů může testování dále zjednodušit a standardizovat jejich strukturu. Pro účely této práce byl zvolen Google Test. Umožňuje testovat konkrétní hodnoty, ověřovat vyvolané výjimky i vytvářet znovupoužitelné struktury pro testování jednotným způsobem; díky tomu budou testy přehlednější. Výhodou je integrace Google Test se systémem CMake, proto je možné po úvodním nastavení testy vytvářet jako jednotlivé zdrojové soubory bez dalších úprav.

3.2 Emulační platforma

3.2.1 Stanovení požadavků

Aby byl projekt co nejuniverzálnější, bude v rámci projektu vyvinuta platforma zjednodušující vývoj libovolného emulátoru historického systému. Na základě důkladné analýzy konzole NES je možné vytvořit seznam požadavků pro takovou platformu, což poslouží jako reference pro návrh.

Klíčovým požadavkem pevně souvisejícím s principem fungování číslicového hardwaru je modularita, což představuje dělení na *nezávislé komponenty*. Hardware vždy tvoří více nezávislých komponent. V případě NES se i ve vyšší úrovni abstrakce jedná třeba o procesor, grafický čip a kazetu. Samotné dělení na komponenty bude představovat implementace v třídách v rámci OOP, avšak je rozumné vyžadovat, aby měly komponenty stejné komunikační rozhraní tak, aby se daly univerzálně propojovat.

První požadavek přirozeně vytváří další: po navržení jednotného rozhraní pro komponenty musí být navržen *způsob vzájemné komunikace* a to ideálně takový, který odpovídá skutečné implementaci a zároveň je rozumně efektivní.

Třetím požadavkem je vytvoření *abstrakce celého systému*, v jehož rámci se budou komponenty propojovat. Očekává se, že emulátor bude nabízet více různých systémů, proto je namístě vyžadovat jednotné rozhraní pro všechny systémy. Takové rozhraní by mělo obsahovat možnost řízení běhu systému přirozeně pomocí hodinového signálu; zároveň i možnost, jak získávat různé informace o systému.

Jakmile bude v emulátoru existovat množina komponent jako součást systému, je možné uvažovat o *monitorování vnitřního stavu*. Bude nutné vytvořit jednoduchý způsob, jakým je možné pracovat s celým systémem i jednotlivými komponentami až na úroveň registrů a obsahu paměti.

Kromě vnitřního stavu je přirozené očekávat, že bude emulátor nabízet i běžně externě přístupné rozhraní, tedy reprezentaci *vnějšího stavu systému*. Pro zachování univerzálnosti je nutné vytvořit univerzální řešení umožňující reprezentovat obrazový výstup (v případě NES se jedná o výstup herní grafiky z čipu PPU na televizor; v případě jiných soudobých systémů může jít o výstup terminálu), zvukový výstup (u NES je to zajištěno čipem APU; jinde, například u Commodore 64 čipem 6581 SID, u Atari čipem POKEY . . .) a uživatelský vstup (herní ovladače u NES, klávesnice u domácích mikropočítačů).

Pro shrnutí následuje seznam zjištěných požadavků:

- dělení na nezávislé komponenty,
- způsob komunikace mezi komponentami,
- abstrakce celého systému zastřešujícího komponenty,
- monitorování vnitřního stavu komponent,
- reprezentace vnějšího stavu systému.

3.2.2 Porovnání s původními návrhy

Požadavky přímo nevyplývající ze zadání se objevovaly postupně při vývoji několika prvních verzí emulační platformy; až posléze vznikl ucelený seznam popsaný v části 3.2.1. Pro úplnost a pochopení požadavků je vhodné připomenout nedostatky původních verzí.

3.2.2.1 Platforma 1.0

První platforma vznikla čistě pro účely experimentování. Hlavní cíl bylo rychle vytvořit funkční prostředí, ve kterém je možné implementovat emulovaný hardware. Velkou výhodou první platformy tedy bylo to, že byla implementována velice rychle.

Rozhraní komponent bylo různé a systém byl ve své podstatě reprezentován třídou pro sběrnici. Tato vlastnost se projevila okamžitě při nutnosti testovat procesor 6502 zvlášť v omezeném systému; bylo nutné buď duplikovat kód, nebo se kód stával rychle nepřehledným. I z toho důvodů byla jako požadavek pro další platformy stanovena především univerzální modularita.

Další velice nepříjemnou vlastností plynoucí z absence modularity byla neexistence univerzálního způsobu prezentace stavu systému; vše bylo řešeno vystavením všech registrů a pamětí

```
ImGui::Text("V nameY: %u", m_bus.getPPU().m_internalRegisters.v.bits.nameY);
ImGui::Text("V nameX: %u", m_bus.getPPU().m_internalRegisters.v.bits.nameX);
ImGui::Text("V fineY: %u", m_bus.getPPU().m_internalRegisters.v.bits.fineY);
ImGui::Text("V coY: %u",   m_bus.getPPU().m_internalRegisters.v.bits.coarseY);
ImGui::Text("V coX: %u",   m_bus.getPPU().m_internalRegisters.v.bits.coarseX);
```

Výpis kódu 3.1 Zobrazení stavů registru „V“ PPU na platformě 1.0.

přímo, nebo pomocí getterů. Často tedy docházelo k porušení zapouzdření. Příklad takového špatného přístupu je v ukázce kódu 3.1.

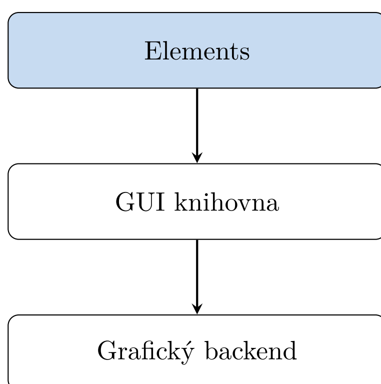
Poslední velkou nevýhodou této platformy byla vázanost na jeden video i audio backend. To znemožňovalo jednoduchou výměnu těchto komponent a snadnou přenositelnost na jiné platformy. I tato nevýhoda byla eliminována v poslední verzi platformy.

Další nevýhody plynuly spíše pro uživatele. Jednou z nich byla například i absence systému dokování oken, což vždy vyústilo v nepřehledné uživatelské rozhraní. Velice špatně byl řešen i zvukový výstup, samotná emulace probíhala ve stejném vlákně jako callback pro zvukový ovladač, což vedlo k trháním a dalším zvukovým artefaktům.

3.2.2.2 Platforma 1.5

Mezikrok mezi platformami byl spíše prostorem pro vyzkoušení různých technik abstrakce grafického rozhraní. Jako první vznikla knihovna Elements, umožňující definovat soubor datových struktur k zobrazení bez závislosti nejen na grafickém backendu, ale i na celé grafické knihovně. Projekt je dostupný na `https://github.com/andreondra/elements`.

Myšlenkou bylo vytvořit třetí vrstvu abstrakce, jak ukazuje diagram 3.1. Důvod takového uvažování vznikl z prosté myšlenky. První platforma řešila zobrazování vnitřního stavu komponent externě, tudíž do nich musela nějakým způsobem zasahovat (například přes gettery). To způsobilo vazbu mezi více třídami; zobrazování bylo vázáno na vnitřní implementaci a nabízené rozhraní dané komponenty, které nebylo jednotné. Bylo navrženo, že by komponenta měla být zodpovědná sama za zobrazování vnitřních hodnot tak, aby na tyto informace nemuselo být vázáno okolí. Komponenta by pouze zveřejnila nutné rozhraní pro komunikaci. Aby však komponenta nebyla vázána na konkrétní grafickou knihovnu, byla vytvořena idea další abstrakce.



Obrázek 3.1 Úrovně abstrakce práce s grafikou.

Úkolem knihovny Elements bylo poskytnout takové prostředky, pomocí kterých by programátor v dané komponentě definoval, jaké proměnné a v jaké formě se mají zobrazovat v grafickém rozhraní (otázka *Co?*). Vše ostatní by řešila knihovna (otázka *Jak?*). Třída představující

```
Elements::Package pkg("Test package", Elements::Types::Dock::LEFT);
pkg.addElements({
        new Elements::Text("Example text received via getter:"),
        new Elements::Text(getCurrentInstruction()),
        new Elements::Number("Number 8", &number8),
        new Elements::Number("Number 16", &number16),
        new Elements::Number("Number 32", &number32),
        new Elements::Number("Number Int", &numberInt),
        new Elements::Number("Number 64", &number64),
        new Elements::Separator(),
        new Elements::Number("Number dbl", &numberDbl),
        new Elements::Separator(),
        new Elements::Bool("Check me!", &exampleBoolean),
        new Elements::Separator(),
        new Elements::Memory("Example Mem", memory, sizeof(memory), 0, false),
        new Elements::Separator(),
        new Elements::Text("This is a beginning of custom rendering!"),
        new Elements::Display("Display Example", pixeldata, true),
        new Elements::Text("This is the end of custom rendering!")
});
```

Výpis kódu 3.2 Tvorba Package z Elementů.

abstrahovanou proměnnou a další datové struktury byla označena jako Element. Každá část dat, která měla být reprezentována jakýmkoliv způsobem, by byla Element, ať už obsah registru, obsah paměti nebo surový grafický výstup. Tyto Elementy by byly strukturovány do balíčků označované jako Package. Takto bylo možné z libovolné komponenty pomocí společného rozhraní *getElements* získat abstraktní definici uživatelského rozhraní nezávislého na aktuální vnitřní reprezentaci.

Elementy umožňovaly získávat data nejen přímo, ale i pomocí funkcí, bylo tedy možné hodnoty před zobrazením ještě upravit. Příkladem tvorby balíčku je kód ve výpisu 3.2. Takový kód je prakticky spustitelný, v repozitáři projektu je součástí příkladu v souboru `main.cpp`.

Definice proměnných a dalších struktur k zobrazení měla probíhat ve veřejné metodě třídy komponenty. Vznikly dva návrhy, jak by se definice mohla řešit: *rekurzivně* a *registrací*.

Rekurzivní metoda počítala s tím, že by funkce getElements vždy vrátila balíček, který by se začlenil do balíčku nadřazené komponenty. Takto by i vznikla stromová struktura reprezentující hierarchii komponent. Registrační metoda by pak jako argument obdržela kontext, do kterého by vložila potřebné informace o datech k zobrazení bez nutnosti zanoření.

Nakonec se ukázalo, že spousta dalších funkcionalit knihovny Elements by už přímo musela souviset s emulací (například Elementy pro řízení časování); také vývoj abstrakce nad všemi potřebnými typy zobrazovaných dat by zabral příliš mnoho času. Od vývoje knihovny bylo ustoupeno ve prospěch zcela nové platformy.

3.2.3 Návrh platformy 2.0

Úkolem platformy druhé verze bylo již vytvořit univerzální multiplatformní prostředí pro vývoj emulátorů; takto vznikla verze 2.0 označována jako Universal System Emulator. Průběžným vývojem se objevilo mnoho problémů a jejich řešení, což vyústilo v celistvý seznam požadavků uvedený v 3.2.1.

Celou platformu bude představovat třída *Emulator*. Součástí bude rozhraní pro přepínání systémů, jejich řízení a také bude zastřešovat veškerá potřebná uživatelská nastavení. Zvýšenou péči v návrhové části vyžaduje především systém pro nastavování preferencí. Jako první bude implementována možnost nastavit klávesy pro jednotlivé vstupy emulovaných systémů. Takové rozšíření pro knihovnu Dear ImGui neexistuje, je tedy nutné jej naprogramovat. Jelikož je požadavek nastavování kláves využitelný i v jiných projektech, bude toto rozšíření vyvinuto zvlášť a do projektu přidáno jako závislost. Jelikož se vývoj této knihovny (rozšíření) přímo netýká hlavního textu, je popsán v příloze B.

3.2.3.1 Modularita: třída Component

První požadavek hovoří o modularitě a univerzálním rozhraní. Bude tedy implementována abstraktní třída *Component*, která zastřeší veškeré komponenty obsažené v systému, ať už to je složitější prvek jako procesor, nebo jednodušší prvky: paměti, sběrnice. Komponenty by měly poskytovat následující funkcionality:

- práce s piny (porty) na čipu — (pouze) pomocí nich komponenta komunikuje s okolím,
- práce s metadaty — každá komponenta má svůj název,
- inicializace — navrácení komponenty do stavu po spuštění (restart),
- vytvoření rozhraní pro monitoring vnitřních i vnějších stavů — bude čistě v režii komponenty, aby se nemusely interní informace vystavovat ven,
- rozhraní pro zvukový výstup.

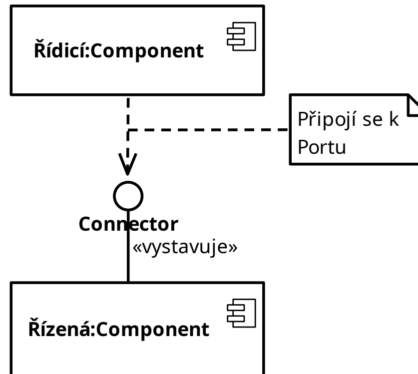
Jediný problém, který při návrhu nastal, byla prostá otázka: co když není nějaká z výše uvedených funkcionalit ve skutečném hardwaru k dispozici? Možností bylo specializovat komponenty dle typu zařízení (pasivní: paměť, aktivní: procesor . . .). Toto řešení bylo rychle zavrženo, anžto by popíralo myšlenku zcela unifikovaného rozhraní. Výsledné řešení je prozaické: zkrátka dojde k navrácení „prázdných“ hodnot (odpovídá to realitě, absence funkcionality totiž odpovídá prázdnotě). Konkrétní definice prázdnoty je implementačně závislá: prázdný kontejner, vyvolání výjimky, vrácení hodnoty přiřazené prázdnotě. Například prázdný kontejner může být vrácen tam, kde se vždy vrací skupina hodnot (seznam grafických prvků), vyvolání výjimky může být vhodné tam, kde by běh aplikace již neměl smysl (programátor zvolí neexistující port), speciální hodnota reprezentující prázdnotu může být navrácena u jednoduchých typů (například je-li zvuk reprezentován párem čísel, absence zvuku odpovídá nule).

Díky existenci jednotného rozhraní bude možné s komponentami pracovat hromadně; například zobrazování rozhraní bude spočívat v zavolání stejné funkce napříč všemi komponentami v jednom kontejneru.

3.2.3.2 Komunikace: třídy Port a Connector

Při návrhu komunikace mezi komponentami musí být brána v potaz i reálná implementace. Lze vyjít z hardwarového schématu konzole, které je obsahem příloh jako obrázek A.2. Na obrázku je patrné, že komponenty obsahují jednak adresní (A), datové (D) a řídicí (R/W) signály tvořící sběrnici, jednak jednotlivé signály (NMI, IRQ, INT. . .). Při komunikaci na sběrnici je vždy nějaký prvek, který je řídicí a přes sběrnici (nebo přímo) přistupuje ke komponentám. Popsané chování lze emulovat tak, že řízená komponenta vystaví rozhraní, kterým ji lze ovládat, a řídicí komponenta toto rozhraní využije. Pro řízenou komponentu bude vytvořena třída *Connector* a pro řídicí komponentu třída *Port*. Řízená komponenta tedy nabízí několik konektorů, které si může řídicí třída připojit do svého portu, jehož prostřednictvím bude s řízenou komponentou komunikovat. Využitím rozhraní odděleného od konkrétních komponent je možné komponenty

propojovat libovolně, což bude důležité v budoucích fázích projektu, kdy bude možné komponenty skládat do systému v grafickém rozhraní. Bude tedy nutné mít možnost dostupné rozhraní enumerovat, což je díky takto univerzálnímu návrhu možné. Grafická reprezentace komunikace komponent je na obrázku 3.2.



Obrázek 3.2 Komunikace komponent pomocí tříd Port a Connector.

Otázkou, kterou je třeba již v návrhové fázi vyřešit, je zdali zůstane úroveň abstrakce u jednotlivých signálu i v případě rozhraní pro sběrnici. Komunikaci je možné řešit stejně jako u reálného hardwaru — vystavení adresy a příslušného řídicího stavu, poté přenos dat. Ukázalo se však, že ačkoliv by přístup věrně napodobil realitu, přinesl by zbytečné komplikace, přestože by výsledek byl stejný, jako u abstraktnějších řešení. Případný zájemce o vývoj vlastní komponenty by mohl být odrazen komplikovaností realistického přístupu. Výsledné řešení je kompromisem: ponechat možnost řízení na úrovni jednotlivých signálu, což by mohly navíc využít jednoduché signály (přerušení, hodinový signál); zároveň však i zavést možnost, jak přenést všechny tři komunikační informace najednou (adresu, data i řízení).

Sdružení komunikace přináší i otázku, jak reprezentovat šířku komunikačního kanálu. Adresní sběrnice může být u různých systémů různě široká, stejně tak i datová. V prvotních fázích bylo uvažováno, že šířka může být reprezentována pomocí šablon (templates), které C++ nabízí; adresa i data by tak byla omezena zcela nativně datovým typem (například `uint8_t`). Ukázalo se však, že mnohem lepší alternativou je použít dostatečně široký datový typ pro historické systémy (32 bitů) a samotné ořezávání nastavit konstruktorem. Důvody jsou dva: obsahuje-li třída šablony, je nutné vše specifikovat v hlavičkovém souboru (nepřehledné), navíc existují i šířky sběrnic, pro které neexistují nativní datové typy (například adresní sběrnice pro komunikaci PPU s kazetou je široká 14 bitů).

3.2.3.3 Abstrakce systému: třída System

Po vymyšlení reprezentace komponent a jejich vzájemné komunikace je možné navrhnout abstrakci systému. Systém by měl poskytnout pro komponenty místo (stejně, jako jsou například v NES v krabičce na desce plošných spojů). To bude vyřešeno jednoduchým kontejnerem. Mělo by být možné pomocí systému řídit celou emulaci až na úrovni jednotlivých cyklů. Bude tedy dedikována metoda pro zaslání hodinového taktu do všech komponent.

Dále by měla existovat metoda umožňující běh v reálném čase, která bude taktovací metodu využívat. Není možné běžící vlákno uspávat na časový úsek odpovídající prodlevě mezi jednotlivými takty o frekvenci v jednotkách megahertz, proto je vhodné takových taktů provést více. To, kolik taktů je nutné provést, záleží na četnosti volání zodpovědné metody. Nabízí se jednoduchý výpočet: $\text{počet_volání} = \frac{f_{\text{taktovací}}}{f_{\text{volání}}}$.

Díky tomu, že komponenty budou součástí jednoho kontejneru, je vhodné vytvořit i metody, které automaticky zařídí přenos všech grafických i zvukových informací.

3.2.3.4 Reprezentace vnitřních i vnějších stavů

Poslední dva požadavky spolu úzce souvisí. Pro reprezentaci grafických dat bude vybrána vhodná grafická knihovna. Možnosti jsou dle návrhových vzorů dvě: *immediate mode* a *retained mode*. Dle prvního návrhového vzoru vykreslování probíhá přímo v místě volání příslušných funkcí grafické knihovny. Dle druhého je tato zodpovědnost přenechána grafické knihovně. Požadavkem je jednoduchost a možnost přímo reprezentovat požadované údaje, což naplňuje první návrhový vzor.

Nejpoužívanější a nejlépe podporovanou immediate mode knihovnou je Dear ImGui. Nabízí jednotné rozhraní nezávislé na subsystému řešící spolupráci s konkrétní platformou (renderovací backend). Je možné vybrat mnoho backendů (OpenGL, SDL2, DirectX a další). Platforma 1.0 používala jako backend knihovnu SDL2. Znalost konkrétního backendu přinesla široké možnosti optimalizace (práce přímo s texturami), avšak i určitá omezení v podobě nemožnosti kompilovat pro platformy nepodporované tímto backendem.

Rozumnějším řešením je umožnit tyto backendy vybírat dynamicky a pokud možno bez nutného zásahu programátora. Existuje projekt Hello ImGui, který nabízí celý sestavovací skript postavený na CMake a zároveň i funkce zajišťující potřebné inicializace ovladačů, což přinese další zjednodušení i flexibilitu. Pro bakalářskou práci bude zvoleno rozšíření označené jako ImGui Bundle, které navíc obsahuje například funkcionality pro výběr souborů.

Díky použití knihovny Dear ImGui je možné přenechat vykreslování na každé komponentě, kdy každá funkce reprezentuje jedno okno. Aby mohly komponenty zobrazovat oken více, bude zodpovědná funkce vracet kontejner obsahující renderovací rutiny. C++ pro ukládání function-like objektů nabízí `std::function`. Objekty této třídy lze navíc ukládat do kontejnerů, což přesně odpovídá požadované funkcionalitě. Vrátí-li každá komponenta kolekci vykreslovacích funkcí, může systém tyto kolekce sloučit, zpracovat a předat k vykreslení.

Po návrh konkrétních vykreslovacích funkcí je nutné se zamyslet nad přístupem k zobrazovaným datům. Vlastností immediate mode návrhového vzoru je to, že při vykreslování se přímo přistupuje k datům. Není tedy možné bez dalších úprav zároveň vykreslovat a emulovat. Existují tři možnosti:

1. Nechat běžet vykreslování i emulaci v jednom vlákně.
2. Provádět aktivity paralelně a přístup k datům chránit synchronizačním mechanismem (například mutex).
3. Provádět aktivity paralelně a data mezi vlákny synchronizovat.

Paralelní přístup může lákat vyšším výkonem; mohl by však přinést zbytečně vyšší složitost platformy, přestože by neměl opodstatnění. Druhá možnost se ukázala jako zcela zbytečná, jelikož by téměř vždycky mohlo běžet jen jedno vlákno, tudíž by tento přístup odpovídal výkonu první varianty. To plyne z faktu, že jeden hodinový cyklus může ovlivnit vnitřní stav všech komponent, tudíž vykreslovací vlákno by stejně muselo čekat. Třetí varianta by počítala s tím, že by emulované komponenty vždy vkládaly vzorek svých vnitřních stavů do fronty, ze které by si je vybíralo vykreslovací vlákno. To odpovídá synchronizačnímu problému producent-konzument. Další problém by však nastal při požadavku na úpravu vnitřních stavů: již by muselo dojít k zasílání i druhým směrem a musela by se tudíž vyvinout abstraktní knihovna podobná Elements popsané v sekci 3.2.2.2, což se ukázalo jako neekonomické. Nakonec byl zvolen první přístup, který dovoluje ve vykreslovacích funkcích specifikovat zobrazení požadovaných dat přímo a tudíž velice jednoduše. Například stav příznaku procesoru: `ImGui::Checkbox("C", &m_registers.status.c)`.

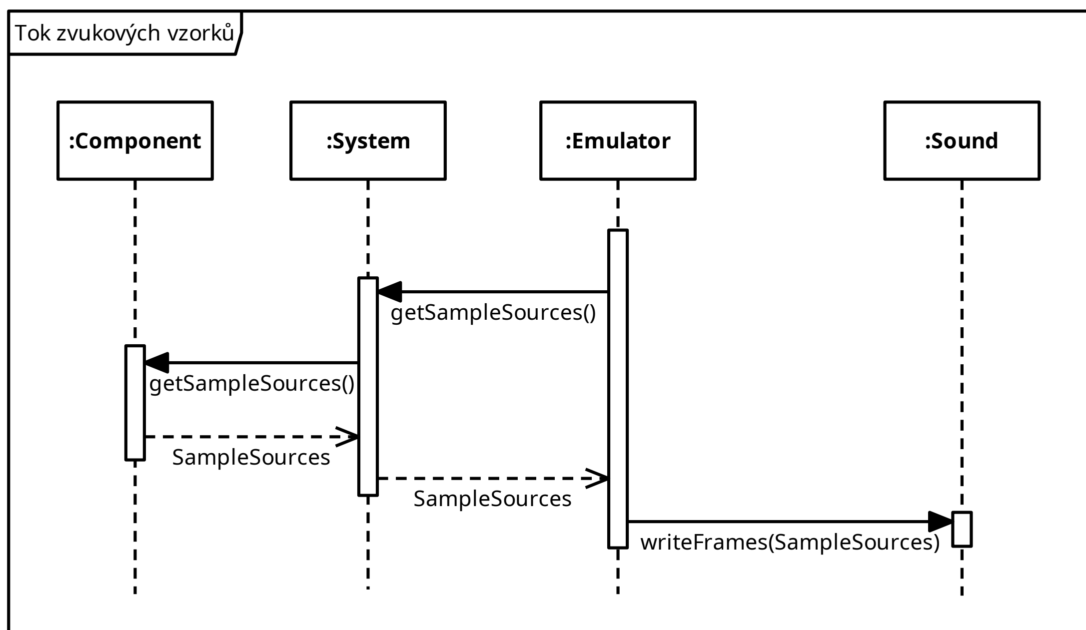
3.2.3.5 Zvukový výstup

Posledním problémem k vyřešení je zvukový výstup. Zvukové zařízení (čip, karta) vyžaduje neustálý tok nových zvukových dat, který je vždy vyzvednut takzvaným callbackem běžícím

ve zvláštním vlákně. Dojde-li k přerušení, vznikají různé nepříjemné vedlejší efekty (chrčení, zrnění), které jsou uživatelem zaznamenatelné citelněji, nežli občasné zpomalení uživatelského rozhraní. Jedná se tedy o reprezentaci zpracování v reálném čase, kde deadline je při vyprázdnění vyrovnávací paměti obsahující vzorky k přehrání.

Prvním úkolem bylo nalezení vhodné knihovny. Mezi požadavky patřila podpora více platforem (GNU/Linux, Windows, macOS, Emscripten), vestavěné filtry (dolní propust) a vestavěné možnosti synchronizace (vyrovnávací paměť). Mezi kandidáty byly knihovny libsoundio, ffaudio, Maximilian, libnyquist a miniaudio. Adeptem naplňujícím všechny zmíněné požadavky je pouze miniaudio.

Platforma by měla umožnit pracovat se zvukovým výstupem co nejjednodušeji. Veškerý zvukový výstup bude řešit zvláštní třída, která by měla miniaudio inicializovat, zaslat další vzorky k přehrání a zvuk spouštět i zastavovat. Od komponent bude vyžadováno jen velice primitivní rozhraní odpovídající reálnému stavu: *analogová* hodnota reprezentující amplitudu na výstupním portu. Veškeré časování bude zařízeno platformou, která se bude starat i o vyrovnávací paměť tak, aby nedošlo k trhání zvuku. Očekávaný tok vzorků audia je znázorněn na diagramu 3.3. Takový mechanismus nabízí přímo knihovna miniaudio ve formě kruhového bufferu, který umí pracovat s jedním producentem (emulační vlákno) a jedním konzumentem (audio vlákno). Zde se opět ukazuje praktický výskyt synchronizačního problému producent-konzument, avšak je nutné oproti práci s daty k vykreslení aplikovat synchronizační mechanismy.



Obrázek 3.3 Tok zvukových vzorků v platformě 2.0.

3.2.3.6 Shrnutí návrhu

Celkově se tedy platforma bude skládat ze:

- třídy Emulator, zastřešující celý projekt včetně možnosti nastavení kláves (knihovna ImInputBinder),
- třídy System, reprezentující systém, ve kterém se komponenty propojí pomocí tříd Port a Connector,

- tříd Port a Connector, reprezentující rozhraní řídicí, respektive řízené komponenty,
- třídy Component, reprezentující jednu komponentu, obsahující metody pro propojování portů a konektorů a pro export grafických i zvukových vzorků,
- třídy Sound abstrahující veškeré nepříjemné aspekty práce se zvukem (inicializace, správa vyrovnávací paměti, přehrávání).

3.3 Emulace konzole

Emulační platforma samotná je jen nástrojem pro vývoj emulátorů — nyní je třeba navrhnout jak implementovat konkrétní komponenty. Všechny komponenty budou potomky třídy Component pro dodržení myšlenky univerzálního rozhraní napříč celým emulátorem. Většina práce bude přenesení analýz, ale některým částem je třeba věnovat zvláštní pozornost, což je popsáno v následujícím textu.

3.3.1 Základní komponenty

Základní komponenty je záhodno navrhnout univerzálně. Součástí NES je hlavní paměť a dvě sběrnice.

3.3.1.1 Paměť

Paměť je možné reprezentovat velice jednoduše: jedná se typicky o blok bajtů za sebou, což lze reprezentovat polem. Paměť může reagovat na různé adresní rozsahy a může být různě velká. Součástí konstruktoru tedy budou informace o velikosti a o rozsahu. Díky existenci univerzálního rozhraní Port a Connector stačí pouze do příslušných čtecích a zapisovacích funkcí správně nastavit přístupy dle adresního rozsahu a nastavené velikosti.

3.3.1.2 Sběrnice

Sběrnice bude pouhým prostředníkem mezi komponentami. Stejně jako u skutečného hardwaru bude zápis na nějakou adresu vystaven na celou sběrnici — to, jestli komponenty zareagují, je již na jejich vnitřní implementaci (například paměť reaguje jen na určitý rozsah, ale to sběrnici nemusí zajímat). Podobně jako u zápisu i adresa čtení bude vystavena na celou sběrnici. Může dojít ke konfliktům (reakci více komponent). Řešení konfliktů je ale otázka arbitračních mechanismů, ne sběrnice; bude tedy stačit, když se vezme v potaz reakce první komponenty a sběrnice s arbitrací nebude reprezentována touto komponentou (u NES arbitrace není třeba).

3.3.2 Procesor 6502

U návrhu procesoru je nejdůležitější vymyslet, jakým způsobem se budou dekódovat instrukce. Ostatní bude již přenesením analýz do implementace.

Dekódování bude otázka dvou kroků: správně připravit data dle adresního režimu a provést nad těmito daty příslušné operace. Instrukce včetně adresního režimu je jednoznačně reprezentována operačním kódem. Je tedy možné každému operačnímu kódu přiřadit jeden adresní režim a jednu operaci. Adresní režimy jsou pro všechny instrukce společné, je tedy možné je implementovat zvlášť, nezávisle na operacích. Tím se ušetří mnoho práce, jelikož většina instrukcí existuje v několika adresních režimech.

Operační znak je u 6502 jednobajtový. Je možné jej dekódovat několika způsoby:

1. sada podmínek (`if-else`): to by přineslo dlouhý a nepřehledný kód,

2. přepínač (`switch`): je to lepší varianta než předchozí, stále ale bude nutné vytvořit mnoho řádků,

3. vyhledávací tabulka: takto je řešeno i dekódování v procesoru, operační znak by mohl být pouhým indexem to takové tabulky, navíc lze tabulku implementovat přehledně tak, aby byla stejná, jako diagramy v dokumentacích,

4. postupné dekódování: to je možné u takových instrukčních sad, kde mají operační kódy nějaké schéma (například procesor CHIP-8). ISA procesoru 6502 má sice nějaké schéma instrukcí, existují však výjimky, které by se stejně musely implementovat ručně, což by dekódování zpomalilo a kód znepřehlednilo.

Pro účely implementace procesoru 6502 v tomto projektu bude zvolena možnost 3., jelikož je rychlá, přehledná i věrně napodobující fungování původního hardwaru.

NES používá mírně upravenou verzi 6502 — 2A03. Ta zároveň obsahuje zvukový syntezátor. Pro zajištění modularity bude zvukový syntezátor implementován ve zvláštní třídě a 2A03 bude pouze potomkem 6502 s upravenými vlastnostmi.

3.3.3 Grafický čip 2C02

Grafický čip, neboli PPU, může být implementován na více úrovních abstrakce, což bude mít nejvyšší dopad na vykreslování pozadí.

Tím, že nametable ve videopaměti přímo reprezentuje část pozadí, může být tato část renderování vyřešena prostým vykreslením obsahu nametable. To však je možné pouze pro jednoduché hry nevyužívající posuv (scrolling). Aby bylo možné hrát více her, bude nutné vykreslování řešit po cyklech přesně tak, jak je popsáno v analytické části. Pro projekt byla zvolena druhá možnost: sice je náročnější jak na implementaci, tak na výkon procesoru, avšak je věrohodnější a bližší k hardwarovému vzoru.

Další otázkou je reprezentace vnitřních pamětí grafického čipu. OAM i paletová paměť mohou být součástí PPU, jelikož je přímo používá pouze PPU (procesor musí přistupovat přes registry). Avšak videopaměť může být jak součástí PPU, tak kazety; záleží na tom, jaká konkrétní deska (mapper) byla použita. První myšlenkou je ponechat videopaměť v PPU a vystavit další rozhraní mezi PPU a kazetou, pomocí které bude kazeta PPU informovat, zdali se má vestavěná videopaměť používat a jaký typ zrcadlení se má zvolit. Tím, že se ale kvůli zjednodušení sloučily adresní i datové vodiče (viz část 3.2.3.2), je vhodnější najít alternativní řešení. Tím je pevná integrace videopaměti do kazety — stejně o způsobu jejího používání vždy rozhoduje kazeta. Ačkoliv se takto přijde o soulad s hardwarovou reprezentací konzole, věrohodnost emulace to neovlivní, tudíž je možné tuto variantu zvolit.

3.3.4 Kazeta a mappery

Kazeta jako taková je pouze obalem pro paměti a řídicí desku, která rozhoduje o mapování těchto pamětí (mapper). Takto to může být reprezentováno i v softwaru. Třída *Gamepak* (pojmenovaná po obchodním označení kazet pro NES) bude sloužit k načtení souborů obsahujících potřebná data (ve formátu iNES) a vybrání správného mapperu dle zjištěných metadat v hlavičce souboru.

Reprezentace pamětí bude velice jednoduchá. Jelikož paměti nejsou přístupné jinde než v rámci kazety, není třeba vytvářet podsběrnici a reprezentovat je jako samostatné komponenty.

Mapper již bude třeba reprezentovat sofistikovaněji. Pro práci s mapperem je třeba jednotné rozhraní. Konkrétní implementace pak bude záviset na jednotlivých typech. Protože je v C++ možné rozhraní implementovat pouze pomocí dědičnosti, budou jednotlivé mappery potomkem třídy *Mapper*. To, jaký konkrétní mapper je schován za rozhraním, bude známo jen při inicializaci, což ale stačí, jelikož vstupní parametry je třeba předat pouze při konstrukci mapperu. Dále tedy

může být mapper zapouzdřen a veškerá interakce bude probíhat přes jednotné rozhraní (čtení, zápis), což odpovídá realitě.

3.3.5 Zvukový syntezátor APU

Zvukový syntezátor je vlastně skupinou registrů, které říkají, jak se má generovat zvuk. Je tedy nutné se zamyslet, na jaké úrovni abstrakce bude emulace probíhat.

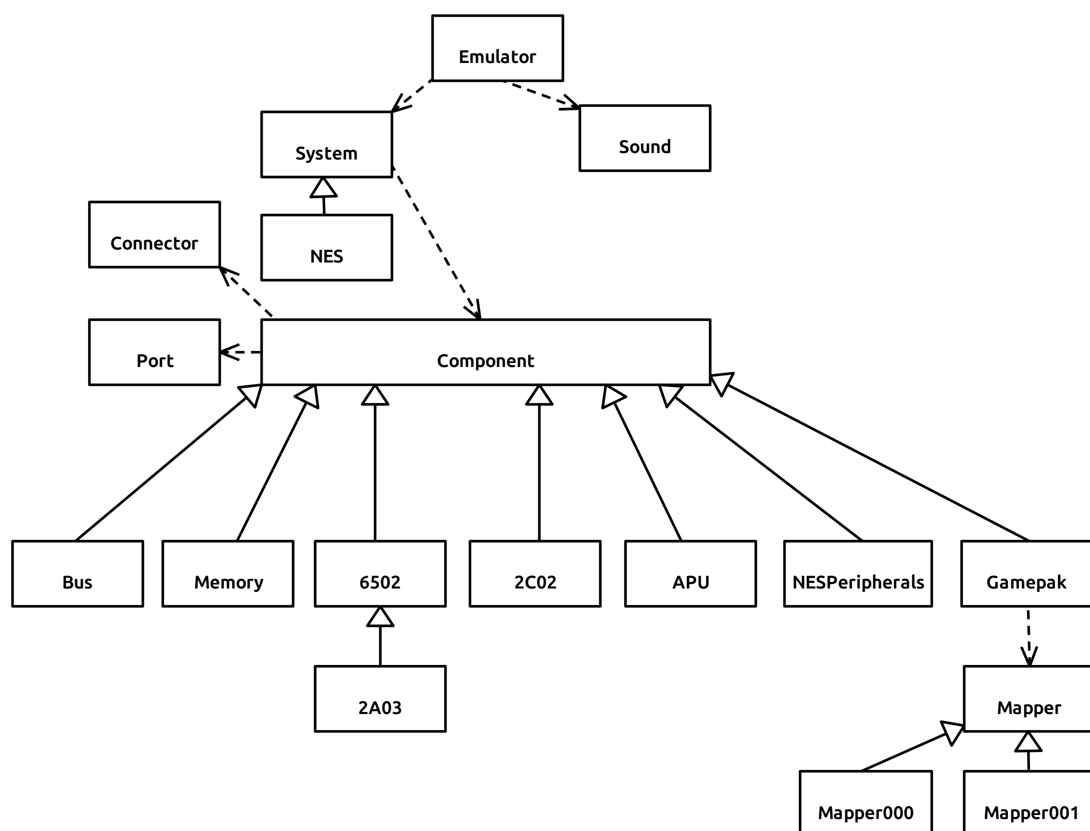
Je možné pouze využít parametry a zvuk generovat jiným způsobem. Existují dva návrhy, které budou vysvětleny na jednom kanálu — generátoru pulzů. Tento kanál generuje čtvercový signál. Posílal-li by se čistý čtverec na výstup, mohlo by dojít k nepříjemným artefaktům, které jsou způsobeny příliš rychlými změnami signálu (čtverec jsou skoky mezi žádným signálem a signálem o maximální amplitudě). Proto se nabízí tento signál aproximovat Fourierovou řadou pomocí sčítání sinusových signálů, vznikne tak mnohem hladší signál. Druhou možností je využít filtrovacích mechanismů, které nabízí zvolená knihovna miniaudio. Využije-li se filtr typu dolní propusti, výsledek bude stejný, ale méně výpočetně náročný. Navíc se takto věrně napodobí chování původního hardwaru: byl generován surový signál, který byl přes sadu filtrů dopraven až do reproduktorů televizoru.

3.3.6 Herní ovladače

Vzhledem k jednoduchosti herních ovladačů není nutné vytvářet speciální návrh. Jde pouze o sadu registrů. Jediné, co je nutné zajistit, je integrace s knihovnou ImInputBinder, která zajistí přenos úhozů kláves do komponenty. Tím, že knihovna pracuje s jednoduchým callbackem, jedná se pouze o vrácení callbacku přes systém až do emulátoru.

3.3.7 Integrace do platformy

Pro přehled je na konec uveden diagram zachycující konkrétní realizaci emulátoru NES na platformě 2.0.



Obrázek 3.4 Integrace NES do platformy 2.0.

Kapitola 4
Implementace

„If you love what you do and are willing to do what it takes, it's within your reach.“

STEVE WOZNIAK

Po analýze a navržení následuje praktická část, kdy se projekt implementuje pomocí vybraných technologií. Úkolem kapitoly implementace je provést strukturou projektu a poukázat na zajímavé řešené problémy.

Praktická část probíhala v několika fázích po dobu 2 let. Během prvních fází byly postupně vytvořeny emulované komponenty, v dalších byla vytvářena emulační platforma v několika verzích (které jsou porovnány v návrhové kapitole v sekci 3.2). Poslední verze emulační platformy, nazvaná jako Universal System Emulator, byla vytvářena posledního půlroku studia na fakultě.

Projekt je podrobně dokumentován. V případě většího zájmu o implementační detaily je důrazně doporučeno paralelně studovat zdrojový kód; je obohacen o mnoho dokumentačních komentářů, z nichž lze snadno vygenerovat dokumentaci ve formátu kompatibilním s webovými prohlížeči (HTML dokument).

▶ Poznámka 4.1 (Zápis signatur metod). U některých metod a funkcí jsou vynechány signatury (značeno trojtečkou), pakliže to není v textu relevantní a je důležitý pouze název. Patřičné signatury i s popisky lze nastudovat ve zdrojovém kódu.

4.1 Struktura projektu

Projekt je k dispozici na veřejném repozitáři na platformě GitHub [48]. Součástí repozitáře je celá platforma, implementace NES i dokumentace a veškeré pomocné nástroje. Obrázek 4.1 strukturu projektu znázorňuje graficky.

4.2 Emulační platforma

Princip fungování emulační platformy bude rozebrán od té nejnižší úrovně — komponenty — až po celou emulační platformu.

- .github ... skripty pro kontinuální sestavování na platformě GitHub Actions
- cmake_src ... další pomocné skripty nástroje CMake
- docs ... zdrojový kód dokumentace
- graphics ... různá pomocná grafika (loga a další)
- include ... C++ hlavičkové soubory odpovídající zdrojovým souborům
- src ... zdrojový kód projektu
  - components ... zdrojové kódy jednotlivých komponent
    - Gamepak
      - Gamepak.cpp ... emulace kazety NES
      - Mapper.cpp ... abstrakce mapperů
      - Mapper000.cpp ... emulace mapperu 0
      - Mapper001.cpp ... emulace mapperu 1
    - 2A03.cpp ... emulace čipu 2A03
    - 2C02.cpp ... emulace grafického čipu 2C02 (PPU)
    - 6502.cpp ... emulace procesoru 6502
    - APU.cpp ... emulace audio čipu APU
    - Bus.cpp ... univerzální emulace sběrnice
    - Memory.cpp ... univerzální emulace přepisovatelné paměti
    - NESPeripherals.cpp ... emulace periferií NES
    - Trigger.cpp ... konvertor mezi typy signálů
  - systems ... zdrojové kódy systémů
  - Component.cpp ... abstrakce komponenty
  - Connector.cpp ... abstrakce konektoru
  - Emulator.cpp ... hlavní třída zastřešující celou platformu
  - Port.cpp ... abstrakce portu
  - Sound.cpp ... rozhraní nad zvukovým ovladačem
  - System.cpp ... abstrakce systému
  - Tools.cpp ... pomocné funkce
- tests ... testy projektu
  - testfiles ... pomocné testovací soubory
  - unit ... testy jednotlivých komponent
- .gitignore ... seznam ignorovaných souborů systémem git
- CMakeLists.txt ... soubor projektu systému CMake
- COPYING ... licence GPL-3.0
- README ... základní informace k projektu
- main.cpp ... hlavní kompilační jednotka projektu

Obrázek 4.1 Popis struktury projektu.

4.2.1 Komponenty

Každá komponenta, která má být součástí platformy, musí splňovat rozhraní definované abstraktní třídou `Component`, což se v C++ provede snadno principem dědění: `class mojeKomponenta : public Component`. Úkolem této abstraktní třídy je poskytnout komponentě vše potřebné pro začlenění do libovolného systému i pro samotný běh emulace a debugging. Pro komunikaci s ostatními komponentami jsou připraveny následující metody:

- `connect(...)`: připojení řízené komponenty,
- `disconnect(...)`: odpojení řízené komponenty,
- `getConnector(...)`: získání rozhraní pro jinou řídicí komponentu.

Dále pro zjednodušení komunikace s uživatelem slouží tyto metody:

- `getGUIs()`: získání seznamu popisů grafického rozhraní debuggerů,
- `getSoundSampleSources()`: získání seznamu funkcí pro načítání zvukových vzorků,
- `getInputs()`: získání seznamu podporovaných uživatelských vstupů (klávesnice, ovladače).

Jsou připraveny i další metody pro sjednocení životního cyklu komponent. Takovými funkcemi jsou `init()` pro uvedení komponenty do stavu po přivedení napájecího napětí (hard reset, nebo také studený start) a `initRequested()` jakožto možnost komponenty požádat o inicializaci celého systému (například po výměně ROM je nutné obnovit programový čítač procesoru, což se například u 6502 standardně děje při resetu).

Vývojář komponenty má za úkol implementovat funkce `getGUIs(), init()` a volitelně další. Většinou je také nutné definovat vnější rozhraní, což se provádí v konstruktoru a příklad je uveden v dokumentaci k projektu.

Nejdůležitějším přínosem platformy je jednoduchost tvorby grafického rozhraní a zvukového výstupu. Nejprve je demonstrován proces tvorby rozhraní na zjednodušené komponentě ve výpisu kódu 4.1. Vše, co je potřebné definovat, je obsaženo v jediné funkci, v rámci níž je možné definovat libovolné množství lambda funkcí reprezentujících vykreslovaná okna. Tyto funkce jsou pak předány v kontejneru typu vektor i s metadaty (kategorie, název okna, popřípadě preferovaný dok, do kterého se má okno umístit).

Mnohem jednodušší je předávání zvuku k přehrání. Tím, že veškerou rutinní práci odvede již implementovaná specializovaná třída `Sound`, stačí v rámci funkce `getSoundSampleSources()` předat dvouvzorkový rámec (jeden vzorek pro levý a jeden pro pravý reproduktor) reprezentující amplitudu, která by byla v reálné komponentě na analogovém výstupu. Pro většinu emulovaných zvukových čipů je to tedy otázka předání hodnot z již hotové funkce generující zvuk (popřípadě i převedení formátu, nejedná-li se o výstup v plovoucí desetinné čárce v rozsahu $[-1, 1]$). Ukázka implementované funkce pro čip APU je obsažena ve výpisu kódu 4.2.

4.2.2 Práce se zvukem

Pro co nejjednodušší zpřístupnění zvukového výstupu byla vytvořena třída `Sound`. Tato třída má velice jednoduché rozhraní: `start()` a `stop()` pro spuštění či zastavení přehrávání dodaných zvukových rámců a `writeFrames()` pro dodání rámců k přehrání. Je zcela abstrahováno to, jaký zvukový ovladač se ve skutečnosti používá, jak dochází ke zpracování jednotlivých zvukových rámců i jak probíhá komunikace s audio callbackem (a tedy dodávání zpracovaných rámců) — to vše zajistí platforma tak, aby byla vývojáři emulátoru zpříjemněna a zjednodušena práce.

Třídu `Sound` spravuje platforma (třída `Emulator`). Při nahrání emulovaného systému dojde k vytvoření grafu reprezentujícího tok zvukových vzorků při jejich zpracování. Počátek grafu je tvořen vstupními body, jejichž počet závisí na počtu zdrojů zvuku. Vstupní body se integrují

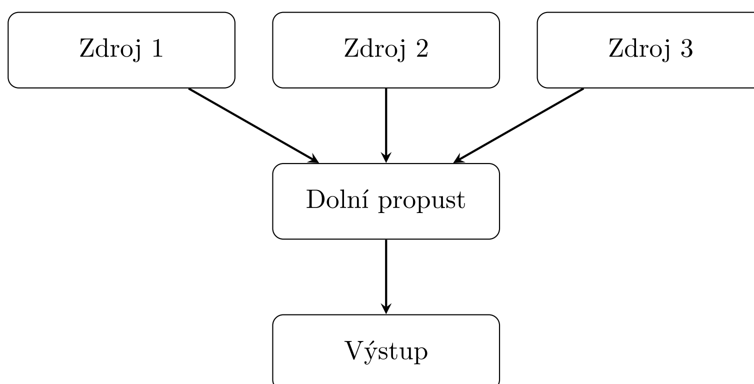
```
std::vector<EmulatorWindow> MujProcesor::getGUIs() {

  std::function<void(void)> debugger = [this](){

    // Definice obsahu okna.
    // ===========================================================
    ImGui::SeparatorText("Current instruction");
    ImGui::Text("Mnemonic: %s",         m_instruction.mnemonic);
    ImGui::Text("Cycles: %u",           m_cycles);
    ImGui::Text("Size: %u B",           m_instruction.instrLen);
    ImGui::Text("Address mode: %s",     getAddressMode().c_str());
    ImGui::Text("Remaining cycles: %u", m_cycles);

    ImGui::SeparatorText("Registers");
    ImGui::InputScalar("PC", ImGuiDataType_U16, &m_registers.pc,
    nullptr, nullptr, "%x", ImGuiInputTextFlags_CharsHexadecimal);
    // ...zkraceno...
    ImGui::InputScalar("Y", ImGuiDataType_U8, &m_registers.y,
    nullptr, nullptr, "%x", ImGuiInputTextFlags_CharsHexadecimal);

    ImGui::SeparatorText("Status flags");
    ImGui::Checkbox("C", &m_registers.status.c);
    // ...zkraceno...
    ImGui::SameLine();
    ImGui::Checkbox("N", &m_registers.status.n);

    ImGui::SeparatorText("Interrupt vectors");
    ImGui::Text("NMI at: 0x%x",     VECTOR_NMI);
    ImGui::Text("RESET at: 0x%x",   VECTOR_RST);
    ImGui::Text("IRQ/BRK at: 0x%x", VECTOR_IRQ);
    ImGui::SeparatorText("Interrupt status");
  };

  // Definice seznamu oken.
  return {
    EmulatorWindow{
      .category =    m_deviceName,
      .title =       "Debugger",
      .id    =       getDeviceID(),
      .dock  =       DockSpace::LEFT,
      .guiFunction = debugger
    }
  };
}
```

Výpis kódu 4.1 Příklad konfigurace grafického rozhraní pro komponentu emulující procesor.

```
SoundSampleSources APU::getSoundSampleSources() {

  return {
    [&](){

      float sample =  output();
      // Mono vystup, proto jsou oba vzorky stejne.
      SoundStereoFrame frame{sample, sample};
      return frame;
    }
  };
}
```

Výpis kódu 4.2 Implementace rozhraní pro přístup ke zvukovým datům.

zcela automaticky bez zásahu vývojáře systému, pakliže správně implementoval funkci pro získání seznamu zdrojů zvuku. Všechny vstupní body jsou smíchány a předány dalšímu uzlu, což je filtr typu dolní propust. Ten je použit proto, aby byly omezeny příliš velké změny amplitud, které vedou k nepříjemnému praskání na výstupu (v reproduktorech či sluchátkách). Nakonec je smíchaný a filtrovaný výsledný zvuk přiveden na výstup. Graficky je tento tok znázorněn na obrázku 4.2.



Obrázek 4.2 Schéma toku zvukových dat ze 3 zdrojů.

Jak lze pozorovat na implementaci mechanismu zpracování zvuku, je jen na vývojáři emulovaných komponent, potažmo systému, kolik zdrojů zvuku platformě poskytne. Je možné vše smíchat vlastním způsobem a klidně za celý systém předat jen jeden zdroj, nebo předat několik zdrojů za každou komponentu a přenechat veškeré míchaní na platformě. Samozřejmě je možné nepředávat žádný zdroj, přehrávání zvuku tak nebude k dispozici.

4.2.3 Komunikace komponent

Jak bylo popsáno v návrhové části, komunikace komponent probíhá pomocí tříd `Port` a `Connector`. Nejprve je ukázáno, jak se jednotlivé třídy používají, poté je vysvětlen princip jejich implementace.

Chce-li komponenta zveřejnit port, pomocí nějž by řídila ostatní komponenty, vybere typ portu a přidá jej jako datovou položku třídy. Poté v konstruktoru přidá port do seznamu zveřejněných portů: `m_ports["nazev_portu"] = &m_mujPort`. Dle vybraného typu portu jsou

pak k dispozici různá rozhraní. Typ `DataPort` je emulací kombinace datových, adresních a řídicích vodičů, umožňuje komunikovat pomocí metod `uint32_t read(uint32_t address)` pro přečtení dat z vybrané adresy a `void write(uint32_t address, uint32_t data)` pro zápis dat na zvolenou adresu. Typ `SignalPort` je poté emulace surových signálů. Umožňuje nastavit logickou úroveň na pomyslném vodiči pomocí `void set(bool active)`, popřípadě pouze zaslat jeden pomyslný signální pulz (změnu logické hodnoty; je na připojené komponentě, jak jej interpretuje) pomocí `void send()`. Nastavení logické úrovně je použitelné pro komunikaci s úrovňovými detektory (u 6502 port IRQ), poslání signálu je poté užitečné pro hranové detektory (u 6502 port NMI). Do portu je možné připojit konektor, takové rozhraní mají oba typy portů společné: `void connect(std::weak_ptr<Connector> connector)`. Port po připojení konektoru zkontroluje, zdali je v konektoru definováno požadované rozhraní a pokud ano, je možné pomocí portu komunikovat s připojenou komponentou přes její konektor.

Chce-li být komponenta řízena jinou komponentou, musí naopak zveřejnit konektor, toť jest typ třídy `Connector`. Konektory jsou v komponentách ukládány do kontejneru typu `std::map`. Deklarace i definice probíhá v konstruktoru. Výpis kódu 4.3 ukazuje, jak lze nadefinovat jak typ konektoru poskytující datové rozhraní, tak typ konektoru poskytující signální rozhraní; prezentovaný kód by byl uveden v konstruktoru.

▶ Poznámka 4.2 (Důvod použití chytrých ukazatelů). Jak bylo popsáno výše, port je vždy lokální datová položka komponenty, kdežto konektory se předávají napříč komponentami. Bylo tedy nutné vymyslet, jak se budou komponenty předávat. Myšlenka surového ukazatele byla v rámci dodržování principů bezpečné práce s pamětí zavržena. Předává-li se totiž surový ukazatel, nemusí být příliš zřejmé, kdo spravuje ukazovaný objekt. Dobrou volbou není ani `std::shared_ptr`, jelikož je sémanticky vlastníkem pouze komponenta, které konektor patří.

Nejideálnější volbou by byl „chytrý“ pozorující (observing) ukazatel. Je to vlastně jen sémantický obal nad surovým ukazatelem, který však jasně říká, že se pracuje s ukazatelem na objekt spravovaný někým jiným. Ten je však v C++ zatím součástí plánovaných rozšíření, tudíž ve jmenném prostoru `std::experimental`, což není zcela vhodné používat ve stabilním kódu [49]. V projektu byl proto použit pro předávání dalším komponentám `std::weak_ptr`, jakožto kompromisní (ale stabilní) řešení.

4.2.4 Systémy

Po vyvinutí komponent je možné začít tvořit komplexní systémy využívající tyto komponenty. K tomuto účelu slouží třída `System` zajišťující jednotné rozhraní i jednotný přístup k definování emulovaných systémů.

Systém obsahuje metody pro postupování emulací po různě velkých úsecích, například po jednotlivých taktech hlavních hodin lze postupovat pomocí `doClocks()`. Je zajištěno i rozhraní pro běh v reálném čase: `doRun(...)`. Tyto metody musí implementovat vývojář systému.

Systém obsahuje mnoho předpřipravených metod. Týká se to především práce se společným rozhraním komponent. Jsou to metody pro získání definic grafického rozhraní všech komponent (`getGUIs()`), získání všech zdrojů zvuku (`getSampleSources()`), definice všech dostupných uživatelských vstupů (`getInputs()`), popřípadě `init()` pro skupinovou inicializaci všech komponent. Aby měl systém povědomí o komponentách v systému, je nutné tyto komponenty přidat do připraveného kontejneru typu vektor s identifikátorem `m_components`. Poté je již funkční výchozí implementace zmiňovaných metod.

▶ Poznámka 4.3 (Grafický editor systému). V současné verzi platformy se počítá s vytvářením systému pouze vyvinutím dalších tříd, které se poté ručně zakomponují do platformy přidáním do nabídky v hlavní třídě `Emulator`. Návrh komponent proběhl však velice pečlivě tak, aby bylo možné do budoucna implementovat třídu reprezentující univerzální systém, který bude implementovatelný grafickým rozhraním. Počítá se s využitím editoru grafů, pomocí nějž půjde skládat systém jak na úrovni komplexních komponent (procesor, grafický čip), tak na úrovni hradel.

```
// Konektor poskytujici signalni rozhrani emulujici hranovy detektor.
  m_connectors["CLK"] = std::make_shared<Connector>(SignalInterface{
  .send = [this](){
    CLK();
  }
});

// Konektor poskytujici signalni rozhrani emulujici urovnovy detektor.
 m_connectors["IRQ"] = std::make_shared<Connector>(SignalInterface{
  .set = [this](bool active) {
    IRQ(active);
  }
});

// Kontektor slouzici pro pripojeni na hlavni sbernici.
// V prikladu jsou k dispozici dva registry na dvou adresach,
// z toho druhy je pouze pro cteni.
m_connectors["BUS"] = std::make_shared<Connector>(DataInterface{

  .read = [&](uint32_t address, uint32_t & buffer) {
    if(address == 0x4016) {
      buffer = m_register1;
      return true;
    } else if(address == 0x4017) {
      buffer = m_register2;
      return true;
    } else {
      return false;
    }
  },

  .write = [&](uint32_t address, uint32_t data) {
    if(address == 0x4016){
      m_register1 = data;
    }

    // (Registr 2 pouze pro cteni, neni potreba zadny kod.)
  }
});
```

Výpis kódu 4.3 Definování konektorů různých typů.

```
// Definice nazvu systemu.
m_systemName = "Bare 6502";

// Pripojeni RAM ke sbernici.
m_bus.connect("slot 1", m_RAM.getConnector("data"));
// Pripojeni sbernice k procesoru.
m_cpu.connect("mainBus", m_bus.getConnector("master"));

// Pridani komponent do metadat tak,
// aby byly funkcni predimplementovane metody.
m_components.push_back(&m_bus);
m_components.push_back(&m_RAM);
m_components.push_back(&m_cpu);
```

Výpis kódu 4.4 Propojení komponent v systému.

Pro implementaci funkčního systému musí vývojář také definovat propojení komponent, vhodné je definovat i název systému. To se provádí v konstruktoru systému. Díky univerzálnímu návrhu je to velice jednoduché. Příklad propojení ukazuje výpis kódu 4.4, v němž je procesor řídicí prvek komunikace na sběrnici (master) a RAM figuruje na sběrnici jako ukázka řízeného zařízení (slave).

4.2.5 Platforma

Platforma je reprezentována třídou `Emulator`. Tato třída definuje společné grafické rozhraní (správa systémů, nastavení přiřazení kláves a další). Zajišťuje také načítání systémů a předávání zvukových dat třídě `Sound` a grafických dat knihovně Dear ImGui.

4.3 Emulované komponenty

Po dokončení emulační platformy je možné implementovat jednotlivé komponenty a integrovat je s platformou. V rámci bakalářské práce byly vyvinuty jednak komponenty univerzální, jednak komponenty specifické pro konzoli NES.

4.3.1 Univerzální komponenty

Aby mohl procesor komunikovat s více komponentami, je potřebná *sběrnice*. Koncept sběrnice je použitelný ve více systémech, proto je záhodno sběrnici navrhnout univerzálně. Téměř veškerá „logika“ sběrnice spočívá v komunikaci, proto je také většina kódu v definici konektoru. Ukázkou toho, jak se dá emulovat univerzální čtecí a zápisová logika sběrnice pro řídicí zařízení na platformě 2.0 (USE) je znázorněna ve výpisu kódu 4.5.

Podobně jako sběrnice je i *paměť* komponentou, kterou je možné využít i v rámci jednoho systému hned několikrát, z čehož vyplývá, že je vhodné ji také implementovat univerzálně. Stejně jako v případě sběrnice se většina logiky nachází právě v definici konektoru, neboli rozhraní pro zařízení, které bude k paměti přistupovat. Zde již stojí za zmínku fakt, že k paměti lze pomocí definovaného konektoru přistupovat buď přímo, tedy tak, že se přímo k paměti připojí například procesor, nebo nepřímo přes sběrnici, pakliže má procesor ve svém adresním prostoru více zařízení, což je případ procesoru 6502 v konzoli NES. Zkrátka univerzálnost návrhu umožňuje

```
m_connectors["master"] = std::make_shared<Connector>(

 DataInterface {

   .read = [&](uint32_t address, uint32_t & buffer) {

     // Pokus o cteni na vsech zarizenich.
     // Zaznamena se pouze odpoved prvniho zarizeni
     // (primitivni arbitracni mechanismus).
     for(auto & device : m_devices) {
       if(device.readConfirmed(address & m_addrMask, buffer)) {
         buffer &= m_dataMask;
         return true;
       }
     }

     return false;
   },

   .write = [&](uint32_t address, uint32_t data) {

     // Provedeni zapisu na vsechna zarizeni.
     for(auto & device : m_devices)
       device.write(address & m_addrMask, data & m_dataMask);
   }
 }
);
```

Výpis kódu 4.5 Definice čtecí a zápisové logiky sběrnice využívané například procesorem.

```
void Memory::init() {
  std::fill(m_data.begin(), m_data.end(), m_defaultValue);
}
```

Výpis kódu 4.6 Definice počátečního stavu emulované paměti.

```
typedef struct {
  // Oznaceni OZ.
  char mnemonic[4];
  // Ukazatel na adresni rezim.
  uint8_t (MOS6502::*addrMode)();
  // Ukazatel na funkci nalezici instrukci.
  uint8_t (MOS6502::*instrCode)();
  // Delka instrukce v bajtech.
  uint8_t instrLen;
  // Pocet cyklu potrebnych k vykonani instrukce.
  uint8_t cycles;
} instruction_t;
```

Výpis kódu 4.7 Záznam instrukční tabulky procesoru 6502.

s komponentami pracovat modulárně tak, jako by se jednalo o komponenty skutečné. Aby nebyla demonstrována jen tvorba konektorů, kterou lze prostudovat ve zdrojovém kódu práce, je v případě paměti ukázán způsob, jakým se definuje počáteční stav komponenty pomocí metody `init()` ve výpisu kódu 4.6.

Kromě paměti a sběrnice byla implementována ještě další komponenta, `Trigger`, která slouží jako převodník mezi adresním a signálním konektorem. Využití je například v případě, kdy potřebujeme mapovat signální vstup do paměti. V reálném systému se to například používalo pro testování reakcí na přerušení procesoru 6502, kdy se vstupy pro přerušení (IRQ a NMI) mapovaly do adresního rozsahu procesoru tak, aby kód mohl zápisem do předem daných adres uměle-programově přerušení vyvolat (viz sekce 5.2.1). Implementace tohoto převodníku spočívá jen v definici rozhraní konektoru v konstruktoru, což lze v případě zájmu podrobněji prostudovat ve zdrojovém kódu.

4.3.2 Komponenty NES

4.3.2.1 Procesor 6502 a varianta 2A03

První implementovanou komponentou byl *procesor 6502*. Před započetím implementace instrukcí bylo nutné implementovat způsob dekódování instrukcí. Jak bylo vybráno v návrhové části, vhodnou možností je vyhledávací tabulka. Při implementaci záznamu vyhledávací tabulky bylo zváženo vše, co je třeba uchovávat spolu s instrukcí. Jak bylo zjištěno v analýze, instrukce existuje ve více variantách dle adresního režimu, má své zkrácené označení a zároveň může její zpracování trvat různě dlouho, což je měřitelné ve strojových cyklech. Strukturu záznamu ukazuje výpis kódu 4.7, kde byla pro jednoduchost přidána ještě délka instrukce. Celou tabulku lze nalézt ve zdrojovém kódu.

Instrukce i adresní režimy procesoru 6502 byly implementovány jako třídní metody. Příklad absolutního adresního režimu a implementace instrukce pro sčítání s přenosem (add with carry, ADC) je ve výpisu kódu 4.8. Dále bylo nutné implementovat stav po spuštění (opět přetížením metody `init()`) a dále obsluhu přerušení tak, aby věrně napodobila skutečné chování proce-

soru 6502, jelikož je klíčová ke správné synchronizaci komponent. Zpracování přerušení probíhá v několika krocích; protože je dosti rozsáhlé, je zde zevrubně popsáno včetně názvů relevantních metod. Opět je doporučeno nahlédnout do zdrojového kódu v případě zájmu o hlubší pochopení.

Při obdržení požadavku na přerušení (obvykle pomocí relevantního konektoru) dojde k nastavení vnitřního signálu pomocí metod dle typu přerušení (`NMI()`, `IRQ(bool active)`). Během vykonávání strojového cyklu ve funkci `CLK()` se vyhodnotí priorita přerušení a podle toho je naplánováno další vykonávání: buď se vykoná standardně další instrukce, nebo se provede přerušení. Je-li naplánováno přerušení, zavolá se metoda `irqHandler()`, nebo `nmiHandler()`, ve kterých se provede stejný sled kroků jako u skutečného procesoru (záloha kontextu na zásobník, přepsání programového čítače hodnotou vektoru přerušení) a další vykonávanou instrukcí je pak první instrukce rutiny obsluhy přerušení.

Na základě analýz bylo zjištěno, že se v NES používá upravená verze 6502 s označením 2A03 od firmy Ricoh. Bylo využito principu dědění a byla vytvořena třída `RP2A03`, kde bylo upraveno a doplněno chování původního procesoru. Největším rozdílem (kromě syntezátoru APU, který byl pro přehlednost implementován v jiné třídě) je přítomnost řadiče přímého přístupu do paměti. Tento řadič je dostupný pro použití programem pomocí jediného registru mapovaného do paměti, proto stačilo pouze přidat další konektor. Implementace konektoru (a tedy i celé logiky DMA) je znázorněno ve výpisu kódu 4.9.

4.3.2.2 Grafický čip 2C02

Nejkomplexnější komponenta celého systému je grafický čip přezdívaný PPU implementován jako třída `R2C02`. Implementace se dá rozdělit do logických celků metod dle jejich účelu:

- Definice rozhraní. 2C02 má několik registrů přístupných procesoru. Toto rozhraní se standardně definuje v konstruktoru pomocí konektorů.
- Implementace vnitřních operací. Během vykreslování i mimo něj provádí PPU mnoho vnitřních operací, tyto jsou reprezentovány příslušnými metodami, které odpovídají operacím popsaných v analýze.
- Implementace strojového cyklu. Volání metod příslušejících vnitřním operacím se provádí v metodě `clock()`, která dle aktuálního obrazového řádku provede sled vnitřních operací.
- Implementace pomocných metod. Ty slouží především k vykreslení obsahu vnitřních pamětí s aplikovanými transformacemi (například vykreslení videopaměti za použití aktivní palety), nebo pro práci s vnitřními paměťmi PPU, například paměti OAM a paletová paměť.
- Implementace emulačních metod. Jako u každé komponenty je nutné implementovat její výchozí stav v metodě `init()` a grafické rozhraní v metodě `getGUIs()`.

Rozhraní přesně odpovídá analyzované množině registrů. Úryvek z implementace rozhraní je ve výpisu 4.10, kde je ukázána emulace registru PPUSTATUS pro čtení a registru PPUSCROLL pro zápis.

Vnitřní operace byly rozděleny tak, aby se daly volat odděleně dle aktivního strojového cyklu. Těmito operacemi jsou například `verticalIncrement()`, `verticalTransfer()`, `fetchNT()` a `fetchAT()`. Jedná se především o průběžné aktualizace hodnot registrů (posuvy), načítání indexů z videopaměti, načítání příslušných dlaždic dle indexů z pattern table a nakonec načítání atributů náležících dlaždicím. To vše je postupně ukládáno do posuvných registrů, se kterými se pracuje pomocí metod `feedShifters()`, `shiftShifters()`. Implementace těchto metod je zřejmá z kódu. Zde není uváděna, jelikož se jedná o rutinní práci s registry a pamětí.

Strojový cyklus je pak místem, kde se veškeré operace provádějí. V rámci emulace je evidována aktuální pozice s ohledem na obrazové řádky a body. Dle těchto hodnot je vybrán sled vnitřních operací a také, odpovídá-li aktuální pozice zobrazitelnému úseku NTSC signálu (viz sekce 1.2.5), je vykreslován obraz. Vykreslování je řešeno jednodušší variantou, tedy variantou

```
// Absolutni adresni rezim.
uint8_t MOS6502::ABS(){
  m_addrAbs = m_mainBus.read(m_registers.pc) |
              (m_mainBus.read(m_registers.pc + 1) << 8);
  m_registers.pc += 2;
  return 0;
}

// Instrukce add with carry (scitani obsahu pameti, akumulatoru a priznaku C).
uint8_t MOS6502::ADC(){

  uint8_t memoryValue = m_mainBus.read(m_addrAbs);

  bool memoryNegative = (memoryValue & 0x80) == 0x80;
  bool accNegative    = (m_registers.acc & 0x80) == 0x80;

  uint16_t result = (uint16_t)m_registers.acc +
                    (uint16_t)memoryValue +
                    (uint16_t)m_registers.status.c;

  m_registers.status.c = (result & 0x100) == 0x100;
  result &= 0xFF;
  m_registers.status.z = result == 0x0;
  m_registers.status.n = (result & 0x80) == 0x80;

  if(memoryNegative != accNegative)
    m_registers.status.v = 0;
  else
    m_registers.status.v = memoryNegative == accNegative &&
                           memoryNegative != m_registers.status.n;

  m_registers.acc = result;

  return 1;
}
```

Výpis kódu 4.8 Ukázka implementace absolutního adresního režimu 6502 a instrukce ADC.

```
m_connectors["OAMDMA"] = std::make_shared<Connector>( DataInterface {

  // Registr OAMDMA je pouze pro zapis.
  .read = [&](uint32_t address, uint32_t & buffer) {
    return false;
  },

  .write = [&](uint32_t address, uint32_t data) {
    // Zapisem do 0x4014 se zkopiruje obsah pametove stranky s indexem
    // dle zapsane hodnoty do pameti OAM cipu PPU.
    // Napriklad zapisem 0xAA do registru 0x4014 se
    // prekopiruje rozsah 0xAA00-0xAAFF.
    if(address == 0x4014) {
      for(int index = 0; index <= 0xFF; index++) {
        m_mainBus.write(0x2004, m_mainBus.read(((data & 0xFF) << 8) | index));
      }
      m_cycles += 513;
    }
  }
});
```

Výpis kódu 4.9 Implementace přímého přístupu do paměti v čipu 2A03.

přímého vykreslování RGB hodnot, které se ukládají do dvourozměrného pole reprezentujícího obrazovku. Jelikož se vykreslování bitmapy z dvourozměrného pole provádí na více místech aplikace (například i v debuggeru, kde se vykresluje obsah pattern table), byla navržena pomocná funkce `renderScalableBitmap(...)`. Prvním atributem funkce je bitmapa, druhým atributem je zvětšení. Bitmapa je předávána jako vektor, aby nemusela být předávána zvlášť i velikost — tím by se riskovalo, že vývojář zvolí špatnou hodnotu rozměru a dojde k nezamýšlenému přístupu do irelevantní části paměti.

Mezi pomocné metody se řadí například metoda `applyPixelEffects(...)` sloužící k dodatečným úpravám barvy pixelu, kdy se používá saturovaná aritmetika: přičtení již hodnotu nezvýší více než je limit datového typu, podobně odčítání dovolí snížit hodnotu maximálně do nuly.

Nakonec bylo nutné implementovat rozhraní pro platformu. Opět byla implementována metoda `init()` pro definování počátečního stavu, což jsou v případě PPU stavy registrů, a metoda `getGUIs()`.

4.3.2.3 Kazeta a mappery

Další implementovanou komponentou byla kazeta, potažmo mappery, které jsou součástí těchto kazet. Jako reprezentace kazety byla vytvořena třída `Gamepak`. Tato třída je standardní komponentou systému. Její hlavní zodpovědností je načíst soubor s kopií kazety a správně interpretovat metadata, na jejichž základě je vybrán odpovídající mapper a načtena data (program a grafická data). Načítání se provádí pomocí grafického rozhraní, které třída definuje standardně v metodě `getGUIs`. Aby bylo rozhraní pro výběr souboru jednotné napříč platformami, byla vybrána knihovna ImGuiFileDialog. Knihovna vytvoří dialogové okno, ve kterém uživatel vybere požadovaný soubor. Aplikace se pokusí soubor otevřít, podaří-li se předá se otevřený stream metodě `load(std::ifstream & ifs)`, která již používá standardní C++ funkcionalitu pro práci se soubory.

```
// Vynatek ze cteci metody konektoru PPU.
switch(address) {
  // ...zkraceno...
  case 0x0002:
    // PPU ma zvlastni chovani, paklize se cte stavovy registr
    // jeden strojovy cyklus pred NMI, nebo primo ve strojovem
    // cyklu, ve kterem je vyvolavano NMI.
      if(m_scanline == 241){

      // Jeden cyklus pred vyvolanim NMI je priznak VBL navracen
      // jako false (0) a NMI nebude vyvolano.
      if(m_clock == 0){
        m_registers.ppustatus.bits.vBlank = 0;
        m_blockNMI = true;
      }

      // Cteni primo v cyklu, kde je bezne vyvolano NMI, vrati
      // sice stav priznaku VBL true (1), ale NMI take neni vyvolano.
      else if(m_clock == 1){
        m_registers.ppustatus.bits.vBlank = 1;
        m_blockNMI = true;
      }
    }
    buffer = (m_registers.ppustatus.data & 0xE0) | (m_dataBuffer & 0x1F);
    m_registers.ppustatus.bits.vBlank = 0;
    m_internalRegisters.w = false;
    break;
  // ...zkraceno...
}

// Vynatek ze zapisove metody konektoru PPU.
switch(address) {
  // ...zkraceno...
  case 0x0005:
    if(!m_internalRegisters.w) {
      m_internalRegisters.t.bits.coarseX = (data & 0xF8) >> 3;
      m_internalRegisters.x = data & 0x7;
      m_internalRegisters.w = true;
    } else {
      m_internalRegisters.t.bits.fineY = data & 0x7;
      m_internalRegisters.t.bits.coarseY = (data & 0xF8) >> 3;
      m_internalRegisters.w = false;
    }
    break;
  // ...zkraceno...
}
```

Výpis kódu 4.10 Úryvek implementace rozhraní čipu 2C02.

```
// Prvni dedukce probiha na zaklade dvou bitu v priznaku 7.
uint8_t formatFlag = (flags7 & 0x0C);

// Format NES2.0 ma tuto hodnotu pevne urcenou konstantou 2.
if(formatFlag == 0x08){
  m_params.fileFormat = fileFormat_t::NES20;

// Tato hodnota neni definovana v zadnem modernim formatu,
// jedna se tedy nejspis o zastaraly format.
} else if(formatFlag == 0x04) {
  m_params.fileFormat = fileFormat_t::ARCHAICINES;

// Format iNES musi mit hodnotu nulovou a navic musi byt
// dalsi priznaky prazdne.
} else if (
  formatFlag == 0x00 &&
  0 == flags12     &&
  0 == flags13     &&
  0 == flags14     &&
  0 == flags15
){
  m_params.fileFormat = fileFormat_t::INES;

// Neni-li nejaka podminka splnena, jedna se nejspis o zastaraly
// format vznikly v pocatcich emulace NES; nema pevnou strukturu hlavicky.
} else {
  m_params.fileFormat = fileFormat_t::ARCHAICINES;
}
```

Výpis kódu 4.11 Dedukce formátu kopie kazety.

Načítání probíhá dle struktury popsané v analytické části (sekce 2.5.2). Složitější bylo určit, o jakou verzi formátu souboru se jedná. Jelikož pro tuto informaci nebylo v hlavičce vyhrazeno zvláštní místo, je proces identifikace formátu založen na odhadech dle obsahu hlavičky. Výňatek funkce `load(...)` zachycující algoritmus pro zjištění formátu souboru zachycuje výpis kódu 4.11.

Mappery již nejsou implementovány jako samostatné komponenty, jelikož se nedají využít jinde (jsou pevně svázány s kazetou), tudíž by taková implementace byla zbytečná. Jelikož mají mappery vždy stejné rozhraní (jsou mapovány do rozsahu procesoru a PPU vždy na stejné místo), je možné vytvořit abstraktní třídu, ze které budou konkrétní typy mapperů dědit. Jak bylo rozhodnuto v návrhové části, obsluhu vestavěné grafické paměti bude mít na starosti přímo kazeta. Je tedy možné tento kód vložit do nadřazené třídy, jelikož jej mohou používat všechny mappery. Dále mapper obsahuje pouze abstraktní metody:

- `cpuRead(...)` a `cpuWrite(...)` pro emulaci rozhraní pro procesor,
- `ppuRead(...)` a `ppuWrite(...)` pro emulaci rozhraní pro PPU,
- `drawGUI()` pro vykreslení debuggeru.

Mapper sice nebude zvláštní komponentou, ale také bude vyžadovat vykreslení rozhraní pro debugging, především v tom případě, obsahuje-li další registry pro přepínání paměťových bank, což většina mapperů obsahuje. Proto je součástí i abstraktní metoda `drawGUI()`, kterou volá

```
// Vynatek z mapperu 0; cteni celeho rozsahu pro PRG ROM.
if(addr >= 0x8000 && addr <= 0xFFFF) {
  data = m_PRGROM[addr & (m_PRGROM.size() - 1)];
  return true;
}

// Vynatek z mapperu 1; cteni z prvni banky PRG ROM.
if(addr >= 0x8000 && addr <= 0xBFFF) {

  addr &= 0x3FFF;
  switch(m_registers.PRGMode) {
    // Aktivni prepinani obou bank.
    case PRGMode_t::SWITCH_BOTH0:
    [[fallthrough]];
    case PRGMode_t::SWITCH_BOTH1:
    data = m_PRGROM[addr | ((m_registers.PRGROMSelect & 0x1E) << 14)];
    break;

    // Aktivni prepinani pouze nizsi banky.
    case PRGMode_t::FIX_LOW_SWITCH_HIGH:
    data = m_PRGROM[addr];
    break;

    // Aktivni prepinani pouze vyssi banky.
    case PRGMode_t::SWITCH_LOW_FIX_HIGH:
    data = m_PRGROM[addr | (m_registers.PRGROMSelect << 14)];
    break;
  }

  return true;
}
```

Výpis kódu 4.12 Ukázka čtení PRG ROM v mapperu.

kazeta ve své vlastní vykreslovací funkci. Funkce pro čtení a zápis jsou poté použity kazetou při definici konektorů (viz zdrojový kód pro kazetu v souboru `Gamepak.h`).

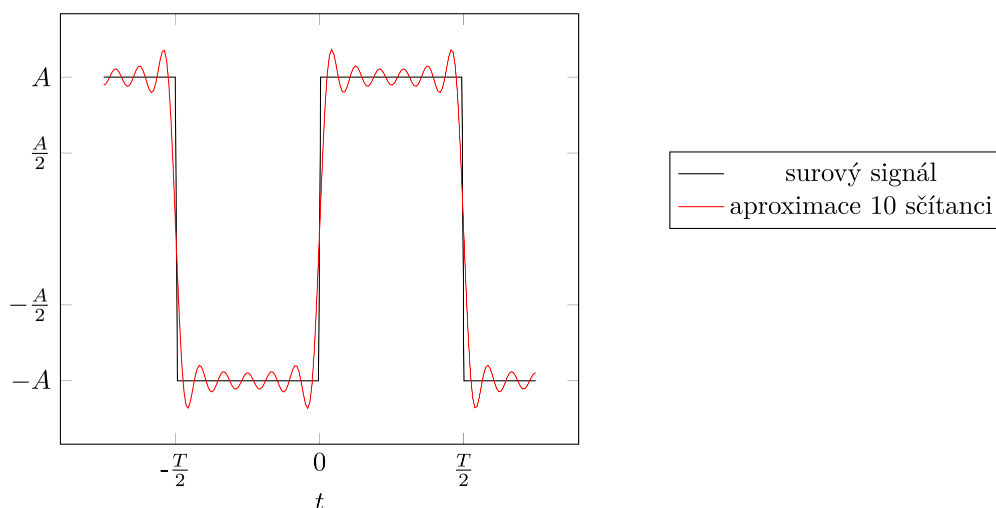
Implementace mapperu 0 byla přímočará, šlo jen o správné mapování obsahu PRG ROM a CHR ROM. Složitější byla implementace mapperu 1, kdy bylo nutné implementovat emulaci přepínání paměťových bank a zároveň i správné mapování dle nastaveného přepnutí. Porovnání funkce čtení PRG ROM procesorem v mapperu 0 a v mapperu 1 ukazuje výpis kódu 4.12.

4.3.2.4 Další komponenty

Dalšími implementovanými komponentami byly APU a herní ovladače.

APU bylo implementováno dle analýz a návrhu. Důležitá byla především správná implementace registrů v rozhraní pro procesor. Byly implementovány dva druhy kanálů. Zajímavý problém k řešení nastal při implementaci výstupu pulzního kanálu APU. V první verzi platformy nebyl k dispozici filtr typu dolní propusti, proto nešlo přímo použít surový generovaný signál. Místo toho byla pulzní vlna aproximována za použití Fourierovy řady pro 10 sčítanců. Porovnání lze vidět na obrázku 4.3. Použitý vzorec je následující (f frekvence, d střída, t čas):

$$vzorek(t) = \frac{2 \cdot amplituda}{\pi} \sum_{n=1}^{10} \frac{1}{n} \sin(\pi n d) \cos(2\pi f t n)$$



Obrázek 4.3 Porovnání surového čtvercového signálu a jeho aproximace Fourierovou řadou.

► Poznámka 4.4 (Přebuzení v aproximaci). Na obrázku 4.3 lze vidět, že při aproximaci dochází k překročení amplitudy, dochází poté ke zkreslení signálu (přebuzení). Tomu lze zabránit vynásobením konstantou z intervalu $(0, 1)$ a snížením tak výsledné amplitudy, popřípadě lze použít funkce zvukové knihovny, které přebuzení zabrání a do zvukového ovladače pošlou regulovaný signál.

Do této aproximace musely být doplněny parametry vlny, ty jsou ale díky architektuře APU jednoduše zjistitelné. Střída je konfigurována přímo registrem. Čas i frekvence se dají implementovat dvěma způsoby. Jedna možnost je taková, že bude frekvence jednoduše zjištěna pomocí vzorce, kde f_{apu} je frekvence hodin APU (pro NTSC zhruba 1,79 MHz) a t je nastavená výchozí hodnota hlavního časovače pulzního kanálu (perioda je tudíž o jedničku vyšší):

$$f = \frac{f_{apu}}{8 \cdot (t + 1)}$$

Tento vzorec vychází z faktu, že sekvencér je osmikrokový, časovač tedy musí osmkrát posunout sekvencér pro vygenerování jedné periody signálu. Časovač je dekrementován každý hodinový takt APU a na nule posouvá sekvencér o krok vpřed, tudíž musí být časovač dekrementován až na nulu (a nastaven zpět na hodnotu periody) celkem 8krát. Z toho vyplývá, že se vlastně jedná o děličku frekvence APU, výpočet je pak přímočarý.

Toto řešení bylo funkční, ale ukázalo se, že je zcela nevhodné pro signál o měnící se frekvenci; při změně frekvence se objevily příliš velké skoky, které způsobovaly stejný problém, jako surový čtvercový signál. Řešením by bylo buď měnit frekvenci postupně, nebo elegantněji použít fázový akumulátor. Akumulátor uchovává hodnotu, která reprezentuje čas (aktuální pozici v signálu). K této hodnotě je přičítán zlomek periody dle aktuální frekvence, tudíž při její změně dojde k plynulému navázání signálu. Nakonec však bylo díky existenci dolní propusti zvoleno řešení mnohem jednodušší, které umožňovalo přesnou emulaci chování skutečného čipu.

Poslední komponentou k emulaci byly herní ovladače. Tyto ovladače jsou v případě konzole NES dva shodné pro každého hráče. Jedná se o přímočarou implementaci posuvných registrů. Jediný problém, který byl během implementace řešen, byl fakt, že je nutné po přečtení všech

tlačítek vracet hodnotu logické jedničky. Není-li výstup posuvného registru takto vyřešen, některé hry nelze spustit. Například hra „Super Mario Bros.“ přestane reagovat na veškeré vstupy již v hlavní nabídce. Může se tedy jednat o jistou formu kontroly, že software běží na legitimním systému.

Kapitola 5

Testování

Implementaci je důležité náležitě otestovat, aby se ověřilo, zdali neobsahuje chyby a zdali naplňuje očekávání. Některé testy lze provádět automaticky, jiné nikoliv. Tato kapitola zběžně provádí průběhem testování a rozebírá zajímavé situace, ke kterým během testování došlo.

5.1 Testování univerzálních součástí

Jelikož se za pomocí univerzálních součástí staví celý systém, je nutné potenciální chyby eliminovat již na této úrovni; jinak by nebylo možné ověřit, zdali je chyba způsobena komponentou, nebo komunikací mezi komponentami.

Testy univerzálních komponent lze i díky jejich malé složitosti provádět automaticky jednoduchými testy integrovanými s platformou Google Test a s nástrojem CTest, tudíž je lze spouštět například při každém commitu do vzdáleného repozitáře na GitHubu a jednoduše ověřovat, zdali daný commit nechtěně nezpůsobil chybu v jiné funkci.

Takto jsou testovány veškeré univerzální komponenty i pomocné třídy: `Bus`, `Memory`, `Trigger` i `Connector`. Ukázku testovacích rutin lze nalézt v příslušných složkách ve zdrojovém kódu.

5.2 Testování komponent konzole

5.2.1 Testování procesoru 6502

5.2.1.1 Příprava testů

Emulovaný procesor 6502 byl testován existujícími testy. Základní implementace byla testována pomocí sady testů od Klause Dormanna [50]. Testy jsou ve formě JSA, kde je možné nakonfigurovat základní parametry testu. Tento test používá pro vyhodnocování výsledků pasti (viz sekce 1.2.6.1), tedy zacyklení v případě dokončení testu (úspěšného i neúspěšného).

Jelikož byl v návrhu stanoven požadavek automatizace testů, je nutné vytvořit pipeline, která testy sestaví a publikuje tak, aby je bylo možné stáhnout a spustit zcela bez zásahu člověka.

Pro účely vývoje emulátoru byl vytvořen fork originálního repozitáře s kódy na adrese `https://github.com/andreondra/use-tests-6502-65C02`. V tomto repozitáři byly testy nakonfigurovány dle potřeb a vytvořen skript pro automatické sestavování nástrojem GitHub Actions.

Test pro ověření funkce přerušení vyžaduje mapování pinů NMI a IRQ do paměti tak, aby je bylo možné ovládat programově. Součástí nastavení je tedy adresa mapování, způsob řízení a umístění signálů dle bitů. Ve výpisu 5.1 jsou čtyři konfigurované položky.

```
I_port    = $bffc   ; Adresa mapovani do pameti.
I_drive   = 0       ; 0 = prime rizeni, 1 = otevreny kolektor.
IRQ_bit   = 0       ; Cislo bitu prirazene IRQ.
NMI_bit   = 1       ; Cislo bitu prirazene NMI (-1, neni-li k dispozici).
```

Výpis kódu 5.1 Příklad konfigurace testu pro procesor 6502.

Po patřičném nastavení testů je možné vytvořit konfiguraci automatického sestavení. Přímo v repozitáři je vytvořen soubor `build-release.yaml` v adresáři `.github/workflows`. Tato konfigurace při každé změně spustí virtuální stroj, zkompiluje zdrojové kódy a publikuje je v novém vydání. Samotná kompilace probíhá pomocí assembleru as65, který je pro jednoduchost obsažen přímo v repozitáři (licence to umožňuje). Výpis 5.2 ukazuje výňatek, ve kterém je spouštěn kompilátor i s popisem použitých přepínačů. Velice důležitý je přepínač `-l`, který vygeneruje listing (viz kapitola 1.2.6.1). Pomocí něj je možné určit, v jaké části programu došlo k zacyklení, a tedy jestli byl test úspěšný či nikoliv a proč.

```
jobs:
    build-and-release:
        runs-on: ubuntu-latest
        steps:
          # ...
          # Pouzite prepinace:
          # -lw = vygeneruje se siroky listing
          # -m  = vypisi se makra
          # -t  = vygeneruje se tabulka symbolu
          - name: Assemble the sources
            run: |
              as65/as65 -l -mwt 6502_functional_test.a65
              as65/as65 -l -mwt 6502_decimal_test.a65
              as65/as65 -l -mwt 6502_interrupt_test.a65
```

Výpis kódu 5.2 Kompilace testů v automatickém sestavení.

Výsledkem každé změny je soubor spustitelných programů ve formě strojového kódu a odpovídající výpisy, jak ukazuje obrázek 5.1.

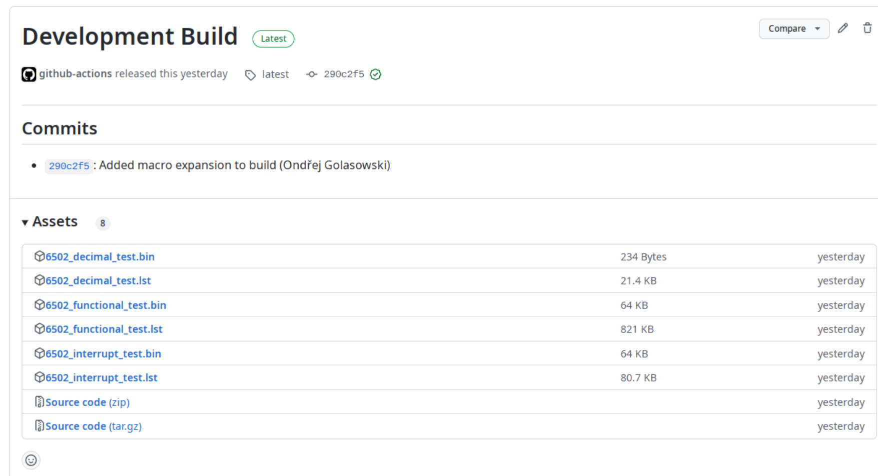
5.2.1.2 Integrace s Google Test

Nakonfigurované a sestavené testy je nyní možné spouštět ručně. Pro automatické spouštění je nutné vytvořit testovací funkci pro platformu Google Test. Aby byly dodržovány zásady dobrého testování, je vhodné test vytvořit za použití existující komponenty, do které se nebudou přidávat žádné funkcionality pouze pro běh testu. Případné změny lze totiž provést přímo v testu a to za použití dědičnosti. Ve výpisu 5.3 je znázorněna upravená třída procesoru. Je použit název DUT (Design Under Test), který se používá například při tvorbě testů ve VHDL. Důležitou úpravou je přidání rozhraní pro úpravu programového čítače, což je nutné pro spuštění testu a kontroly stavu testu.

Samotný test pak probíhá v jednoduchém cyklu, kde podmínkou pro opuštění je opakovaná hodnota programového čítače. Ověření výsledku probíhá porovnáním poslední hodnoty programového čítače. Dle listingu vygenerovaného assemblerem lze zjistit, že úspěch je signalizován

```
class DUT : public MOS6502 {
public:
        void step() {
                while(!instrFinished()) {
                        CLK();
                }

                CLK();
        }
        uint16_t getPC() {
                return m_registers.pc;
        }

        void setPC(uint16_t val) {
                m_registers.pc = val;
        }

        void triggerNMI() {
                NMI();
        }

        void setIRQ(bool active) {
                IRQ(active);
        }
};
```

Výpis kódu 5.3 Upravený procesor 6502 v Google Test.



Obrázek 5.1 Vydání testů pro procesor 6502.

zacyklením na adrese `$06e5`, což ukazuje výňatek ve výpisu 5.4. Celý testovací cyklus s ověřením výsledku se nachází ve výpisu 5.5.

```
                                  success          ; Navesti makra uspechu.
06e5 : 4ce506             >       jmp *            ; Test byl dokoncen uspesne.
```

Výpis kódu 5.4 Řádek signalizující úspěch v listingu testu 6502.

```
do {
        prevPC = cpu.getPC();
        cpu.step();
} while(prevPC != cpu.getPC());

EXPECT_EQ(prevPC, ADR_SUCCESS)
        << "The test failed on trap at address 0x"
        << std::hex << prevPC;
```

Výpis kódu 5.5 Testovací cyklus procesoru 6502 v Google Test.

5.2.1.3 Příklad testování

Jedním z problémů, které se v průběhu vývoje objevily, byl problém s časováním přerušení. Byla proto vytvořena verze, která lépe odpovídá skutečnému chování. Kvůli tomu ale přestal fungovat test přerušení: `6502_interrupt_test.bin`. Tato část popisuje, jak se dá podobná chyba diagnostikovat a opravit.

Test se zastavil na adrese `$6b2`. Dle listingu tato adresa odpovídá sekci, kde se testují obě přerušení, NMI a IRQ. Část způsobující chybu je vyobrazena ve výpisu 5.6. Nejprve se nastaví vyvolání obou typů přerušení instrukcí `sta I_port`, přičemž IRQ bylo maskováno (pro stručnost není proces maskování uveden). Čeká se na provedení obslužné rutiny NMI a otestuje se, zdali

proběhlo. Poté se povolí přerušení IRQ instrukcí `cli`. Nyní by měla proběhnout rutina IRQ, k tomu však nedojde a test se zastaví.

```
; Testovani IRQ a NMI s maskovanim preruseni.
; ...

0699 : 8dfcbf          >        sta I_port      ; Vyvolani preruseni.

069c : e8                       inx
069d : e8                       inx
069e : e8                       inx
069f : ad0302                   lda I_src       ; Probehlo preruseni?
                                trap_ne
06a2 : d0fe            >        bne *           ; Pokud ne, skonci test.

06a4 : a200                     ldx #0

06a6 : a902                     lda #2          ; Testovani IRQ
06a8 : 8d0302                   sta I_src
06ab : 58                       cli             ; Povoleni IRQ.
06ac : e8                       inx
06ad : e8                       inx
06ae : e8                       inx
06af : ad0302                   lda I_src       ; Probehlo preruseni?
                                trap_ne
06b2 : d0fe            >        bne *           ; Pokud ne, skonci test.
```

Výpis kódu 5.6 Výňatek z kódu testu signalizující chybu.

Již na základě této množiny informací lze chybu nalézt. Test nastaví zpětnovazební registr tak, aby byla vyvolána obě přerušení již při prvním testu, kdy se ověřuje NMI. V dalším testu (ověření IRQ) již pouze povolí přerušení a hodnotu ve zpětnovazebním registru nemění.

Skutečný procesor 6502 každé přerušení detekuje jiným způsobem. Přerušení IRQ detekuje úrovňový detektor. Při logické nule se zaznamená požadavek na přerušení a po dokončení stávající instrukce se ověří, zdali přerušení není maskováno. Pokud je, provede se další instrukce obvyklým způsobem a požadavek se zahodí. Tento požadavek je ale znovu zaznamenán, pokud je signál stále aktivní (tedy na logické nule). Emulovaný procesor vyvolal při zápisu do zpětnovazebního registru pouze jeden požadavek, choval se tedy jako hranový detektor, což je v rozporu se skutečným procesorem, a proto toto chování test vyhodnotil jako chybné. Hranový detektor je použit pouze u NMI.

Stačí tedy funkci odpovídající pinu IRQ implementovat tak, aby se dal nastavovat jeho stav (aktivní a neaktivní), nikoliv pouze vyvolávat signál, jako je to u NMI. Klíčovou část kódu ukazuje výpis 5.7.

```
void MOS6502::IRQ(bool active){
        m_irq = active;
}
```

Výpis kódu 5.7 Oprava chybné implementace IRQ.

▶ Poznámka 5.1 (Funkčnost původní implementace). A proč tedy původní implementace fungovala? To bylo způsobeno jinou chybou, kdy se požadavek na přerušení ukládal, pokud bylo přerušení maskováno. To ale také neodpovídá skutečnému procesoru.

5.2.2 Testování grafického čipu

Čip 2C02 je nejkomplexnějším čipem celé konzole, byl testován v několika fázích, především implementováním různých funkcionalit pro zobrazování obsahů paměti a porovnávání s popisem fungování zjištěným v analytické části.

Jako první bylo nutné ověřit, jestli je správně mapován obsah grafické paměti. Pro tyto účely byly vytvořeny funkce, které renderují obsah pattern table s možností přepínat paletu, která je použita. Takto byla odhalena i první chyba, která překvapivě nebyla způsobena chybným přístupem do pattern table, ale špatně implementovanou pamětí palet. Nemá-li totiž program zajištěný korektní přístup do této paměti, zůstane paměť prázdná, a tudíž i pattern table bude vykreslena jako jednolitý čtverec. Po opravě chyb následuje ověření, že je obsah pattern table správně vykreslován. To je možné porovnáním s funkčními emulátory, které obsahují debugger, popřípadě existují i vyobrazení pattern table v různých diskusích a článcích.

Dalším krokem byla implementace zobrazování videopaměti. U jednoduchých her jako „Donkey Kong“, které nepoužívají zrcadlení, stačí vykreslit první polovinu paměti tak, jak je uložena — není potřeba pečlivě provádět sekvenci kroků popisovanou v části 2.4.5.1. To, jestli je obsah paměti správně vykreslen, lze velice jednoduše ověřit opět několika způsoby: spuštěním softwaru na jiném emulátoru, na skutečné konzoli, nebo prostým porovnáním screenshotů z fór a diskusí (což je však nejméně věrohodný zdroj).

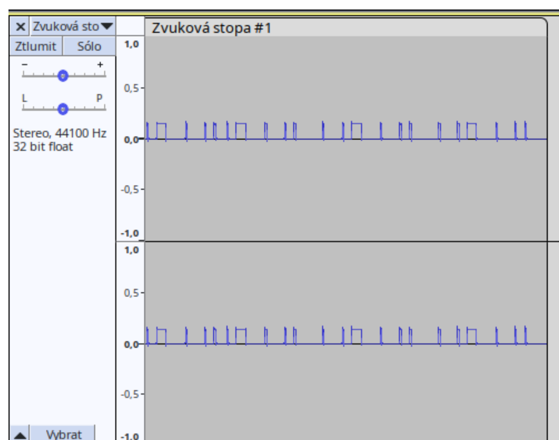
Podobným způsobem byla vždy implementována další funkcionalita a byla testována a opravována tak dlouho, dokud nebyla funkční. Po implementaci všech základních funkcí čipu je možné zkoušet spouštět různé hry a zjišťovat, zdali se chovají dle očekávání. Zde však existuje riziko, že samotná hra obsahuje neočekávané chování; některé hry například chybným přístupem do paměti způsobily poškození obrazu, popřípadě zobrazení nesmyslných textů. Například hra „Ghostbusters“ načítá závěrečný text ze špatné pattern table, tudíž dojde k zobrazení prázdné obrazovky [51]. Je jasné, že cílem softwarové emulace je především to, aby správně běželo původní programové vybavení, tudíž testování spouštěním tohoto softwaru je validní. Ovšem důkladnější ladění je možné jen při použití softwaru, jehož chování je přesně definováno. Takovou roli zastávají testy vytvořené komunitou, která vyvinula testovací programy, definovala jejich chování na reálném systému a zveřejnila je. Velká část používaných testů pochází z Nesdev [52].

5.2.3 Testování APU

Komponentu APU je složité testovat automatizovanými testy, jelikož je zvukový výstup ovlivněn vlastnostmi použitého filtru. Ruční testy lze provádět pomocí nahrávání zvukového výstupu aplikace a analýzy výsledného průběhu signálu. Nástroj, který lze pro tyto účely použít, je Audacity.

V průběhu vývoje APU byl zjištěn problém s výstupem zvuku. Generovaný zvuk obsahoval nepříjemné praskání. První krokem bylo zjistit, který kanál je zdrojem problému, tudíž z výsledného mixu nejprve odstranit pulzní kanál jeho jednoduchým odstraněním z mixovacího výpočtu a ponecháním pouze pulzního kanálu. Bylo nutné vybrat software, který určitě používá kanál pro generování šumu; takovým je například hra „Super Mario Bros.“. Po spuštění softwaru šlo stále slyšet praskání; problémový kanál tak byl identifikován. Nyní je možné nahrát výstup aplikace a podrobně prozkoumat průběh výstupního signálu. Průběh vyobrazený aplikací Audacity je znázorněn na obrázku 5.2.

Lze vidět, že zjištěný průběh zdaleka nepřipomíná šum. Ze znalosti principu fungování generátoru šumu lze usoudit, že problém může být způsoben posuvným registrem; je totiž komponentou, která rozhoduje, zdali bude na výstup puštěna hodnota obálky signálu.



Obrázek 5.2 Ukázka chybného výstupu generátoru šumu.

Nejprve bylo zkontrolováno, jestli byla správně implementována zpětná vazba, což se potvrdilo. Dále bylo zkontrolováno, zdali dochází ke korektní inicializaci. Chyba byla právě zde; registr byl inicializován na hodnotu 0, přitom dle [44]: „On power-up, the shift register is loaded with the value 1.“

Opravením výchozí hodnoty na 1 byl problém vyřešen. Výstup již zněl tak, jako zní ze skutečné konzole či jiných emulátorů.

5.2.4 Další komponenty

Testování dalších komponent probíhalo obdobně, jako bylo již popsáno výše. Herní ovladače byly otestovány reakcemi softwaru, takto byla například objevena chyba v implementaci posuvných registrů zmíněná v sekci 4.3.2.4, kdy se hra „Super Mario Bros.“ spoléhala na jistou hodnotu po přečtení celého registru.

Kazeta a mappery poté byly testovány automaticky pomocí testů na platformě Google Test, stejně jako společné komponenty. Velice důkladně bylo otestováno fungování přepínání bank, protože i malá chyba může způsobit, že se kód začne číst z jiné adresy a celý program přestane fungovat, stejně závažná je i chybná implementace zrcadlení videopaměti, což v případě tohoto projektu také zajišťuje kazeta, tudíž je zrcadlící funkcionalita také testována v rámci těchto testů.

5.3 Shrnutí

Jak je zřejmé z předchozích odstavců, bez průběžného testování by nebylo možné emulátor vyvinout. I přes to však zbývá mnoho prostoru pro vylepšení testů, především pro zvukový a grafický čip. Pro grafický čip je možné testy provádět tak, že se pořídí snímek obrazovky, vytvoří se kontrolní součet a ten se porovná se „zlatým standardem“, neboli s výstupem emulátoru, který je určitě implementován správně. Takové testy budou užitečné především pro budoucí implementátory emulátorů, kteří by si chtěli jednoduše vyzkoušet, jestli je jejich emulátor správně fungující.

Kapitola 6

Navazující práce

Bakalářská práce otevřela mnoho témat, se kterými lze dále pracovat a na které lze navázat další prací, což stručně popisuje tato kapitola. Navazovat lze nejen na vývoj emulátoru NES, ale i emulační platformy, popřípadě lze emulační platformu využít pro vývoj jiných emulátorů.

6.1 Rozšíření emulátoru konzole

Emulovaná konzole NES v projektu sice již obsahuje vše, co je potřeba pro spouštění jednodušších i složitějších her, avšak vzhledem ke komplexitě systému je mnoho prostoru pro rozšíření a zvýšení věrohodnosti emulace.

Přímočaře lze rozšířit množství emulovaných mapperů. Tím, že pro NES vzniklo mnoho softwaru, vznikla také spousta mapperů použitých třeba jen v jednotkách kazet. Mnoho mapperů vzniklo i pro podporu kazet s více hrami (takzvané multicarts).

Dále lze jednoznačně rozšířit počet emulovaných zvukových kanálů čipu APU. Ještě zbývá implementovat emulaci generátoru trojúhelníkového signálu a kanálu pro přehrávání surových zvukových dat.

Hotové emulované součásti NES je také možné ještě vylepšit, například lze implementovat další hardwarová specifika grafického čipu, aby bylo možné spouštět hry simulující trojrozměrný prostor (například „Ferrari Grand Prix Challenge“).

6.2 Rozšíření emulační platformy

Dalším výsledkem práce je emulační platforma Universal System Emulator. Platforma je zveřejněna v repozitáři na GitHubu [48]. Rozšiřování projektu se dá rozdělit do dvou různých částí: rozšíření platformy a rozšiřování balíku podporovaných komponent a systémů.

6.2.1 Rozšíření funkcionality

Samotná platforma má několik chybějících funkcionalit, které jsou využitelné napříč různými emulátory, například zaznamenávání historie stavu komponent a jeho ukládání. Další funkcionality vhodné k implementaci se týkají především přenositelnosti emulátoru: zajištění správného fungování zvuku na všech platformách a nastavení automatické kompilace pro webové prohlížeče (WebAssembly, nástroj Emscripten).

Rozsáhlým plánovaným rozšířením je grafický editor systémů. Rozhraní komponenty bylo již v rámci bakalářské práce navrhováno tak, aby bylo možné do platformy přidat způsob, jakým

s komponentami pracovat v grafickém editoru a skládat je takto do funkčního systému. Tím, že platforma nabízí možnosti komunikace mezi komponentami až na úrovni digitálních signálů, nabízí se mnoho různých zajímavých využití potenciálního grafického editoru. Implementují-li se do platformy hradla, bude možné sestavovat logické obvody. Samozřejmě bude možné propojovat i stávající komponenty. Do obvodu bude tedy jednoduché zapojit například i procesor 6502.

Bude-li implementován grafický editor, bude vhodné prozkoumat další využití tohoto editoru ve výuce. Jedním z návrhů je implementace komponent potřebných k sestavení mikroarchitektury založené na RISC-V ISA. Takto by bylo možné RISC-V kompatibilní procesor sestavit graficky a sledovat průběhy signálů mezi jednotlivými komponentami, což bude zajímavé především pro účely výuky počítačových architektur. V rámci dodatečné rešerše bylo zjištěno, že existují projekty schopné překládat komponenty popsané v jazyce Verilog do jazyka C/C++ (například *v2c* a *Verilator*). Vytvořilo-li by se rozhraní mezi existujícími metodami propojení komponent a rozhraním generovaným těmito nástroji, bude možné do platformy přímo importovat komponenty popsané v jazyce pro popis hardwaru.

S výše uvedeným editorem systémů také souvisí možnost vytvářet a upravovat kód v jazyce symbolických adres, který by byl na systémech spouštěn. Díky tomu bude moct uživatel napsat kód, který se přeloží do spustitelného strojového kódu přímo na emulované komponentě, kde uživatel uvidí, jak dochází ke zpracování jím napsaného programu.

6.2.2 Implementace komponent

Tím, že je platforma navržena univerzálně, lze libovolně přidávat další emulované komponenty. Mezi navrženými komponentami, které by byly zajímavé na implementaci, je například jednoduchý 16 × 2 LCD mapovaný do paměti. Tento displej by bylo možné připojit například k procesoru, který by na displej mohl vypisovat přímo z programu.

Dalším návrhem je implementace procesoru CHIP-8, což je jednoduchý procesor obsahující i primitivní způsob vykreslování grafiky. Je možné implementovat i například koncept procesoru URISC (ultimate reduced instruction set computer, procesor s ISA obsahující jedinou instrukci), což je užitečné jako demonstrace primitivního, ale univerzálního počítače.

6.3 Vývoj dalších emulátorů

Emulační platforma je vhodná jako základ pro využití dalšími projekty, které mají za cíl vytvořit celý emulátor nějakého systému. Díky projektu tito vývojáři nebudou muset přemýšlet, jak vytvořit grafický debugger, jak implementovat zvuk, nebo pro jaký operační systém vyvíjet. Platforma Universal System Emulator totiž vše zmíněné zařizuje sama a navíc je plánována podpora všech hlavních operačních systémů — Linux, Windows, macOS, Android a všech ostatních, na kterých běží moderní webový prohlížeč (Firefox, Safari a prohlížeče založené na projektu Chromium).

Vývojář emulátoru se tak bude již od začátku vývoje emulátoru zabývat pouze vývojem jednotlivých komponent a jejich propojováním do fungujícího systému, což ušetří mnoho času a umožní vývojáři se soustředit na části vývoje spjaté s hardwarem.

Aby se vývoj emulátorů co nejvíce usnadnil, byla vytvořena podrobná dokumentace obsahující popis rozhraní generovaný z komentářů v kódu emulátoru a také podrobný návod k vývoji vlastních komponent a systému. Tuto dokumentaci si mohou vývojáři buďto vygenerovat sami pomocí návodu v repozitáři [48], nebo mohou využít průběžně automaticky generovanou dokumentaci na GitHub Pages, na níž je odkazováno také v repozitáři projektu [48].

Emulační platformu Universal System Emulator nemusí využít jenom hobby programátoři, kteří se chtějí o svém oblíbeném systému dozvědět více vývojem emulátoru, ale například i studenti, kteří takto chtějí bádat v rámci své závěrečné práce; ať už maturitní nebo bakalářské.

Závěr

Primárním cílem práce bylo vytvořit funkční softwarový emulátor konzole Nintendo Entertainment System. Cíl zahrnoval využití ve výuce, což vyžadovalo vývoj přehledného grafického rozhraní. Primární cíl byl v průběhu 2 let vývoje bakalářského projektu značně rozšířen o cíle sekundární, kdy nejdůležitějším bylo vytvoření univerzální platformy pro vývoj emulátorů.

Pro dosažení primárního cíle byla nezbytná *důkladná analýza*. Za pomocí hardwarových manuálů a komunitní dokumentace byl vytvořen přehled principů stojících za fungováním jednotlivých komponent konzole NES. Výsledky analýzy byly popsány v textu bakalářské práce, což z textu vytvořilo užitečný přehled pro potenciální zájemce nejen o vývoj vlastního emulátoru NES, ale i o pochopení principu fungování velmi rozšířené herní konzole. Podrobně bylo popsáno fungování procesoru 6502 včetně jeho varianty 2A03: zpracování instrukcí, práce s pamětí i obsluha přerušení. Byly vysvětleny způsoby vykreslování grafiky čipem 2C02 i principy vnitřních procesů čipu doplněné o přehledné ilustrace — od uchovávání grafických informací v paměti přes vyhodnocení až po samotné zobrazení na obrazovce. Analyzovány a popsány byly také kazety pro distribuci softwaru včetně obvodů pro mapování paměti. Jako poslední byly popsány periferie a čip pro zpracování a přehrávání zvuku. V poslední podkapitole byla shrnuta existující řešení emulace konzole.

Následovala *návrhová část*. V této části bylo na základě výsledků analýzy navrženo optimální řešení. Po výběru programovacího jazyka jakožto i vhodných nástrojů pro vývoj, správu zdrojového kódu, dokumentace a provádění testování byla navrhována platforma pro vývoj emulátoru. Návrh emulační platformy zabral značnou část vývojového procesu. Musel být totiž navržen vhodný stupeň abstrakce i způsob reprezentace komponent a jejich propojení. Byly navrženy a otestovány různé varianty platformy; všechny jsou popsány v textu. Nakonec byla zvolena varianta univerzální platformy využitelné pro další emulační projekty, jelikož existuje mnoho zájemců o tvorbu vlastního emulátoru. Samotná platforma je tak dobře použitelná pro vzdělávací účely, například jako základ maturitní i bakalářské práce. Po navržení platformy již byla navržena implementace konkrétních komponent NES. Byly například vybírány vhodné způsoby dekódování operačních znaků ISA 6502, úroveň abstrakce emulace grafického čipu i způsoby vytváření zvukových vzorků v emulaci zvukového čipu. Tyto úvahy mohou vzhledem ke své obecné platnosti pomoci dalším vývojářům emulátorů.

Po analýze a navržení následovala samotná *implementace vybraného řešení*. Nejprve byly implementovány emulované komponenty NES na raných verzích emulační platformy. V průběhu implementace byly řešeny různé problémy, které jsou popsané v implementační části. Mezi takovými problémy byl například návrh záznamů dekódovací tabulky, rozdělení zodpovědností metodám v implementaci třídy emulující grafický čip nebo konkrétní implementace zvukového výstupu v závislosti na platformě. Bylo totiž nutné vyřešit, zdali posílat surový zvukový výstup generátorů pulzů, nebo tento výstup aproximovat pomocí Fourierových řad. Později byla postupně vytvářena univerzální platforma, ze které vznikl projekt Universal System Emulator do-

stupný na GitHubu. Do tohoto projektu byly emulované komponenty plně integrovány, vznikl tak i referenční systém projektu. V rámci sekundárních cílů byl implementován doplněk grafické knihovny používané v projektu. Tento doplněk přidává chybějící funkcionalitu mapování uživatelských vstupů na akce a je k dispozici jako open-source na platformě GitHub pod názvem ImInputBinder.

Důležitá část vývoje, *testování*, probíhala průběžně. Byly vyzkoušeny různé varianty testování od ručního porovnávání výsledků s reálným systémem až po plně automatizované testy. V příslušné kapitole byly popsány i zajímavé problémy, které testování odhalilo, a způsoby řešení těchto problémů. Pro účely testování procesoru 6502 vznikl další vedlejší projekt, který zajistil automatické kompilace a vydávání testů ve strojovém kódu tak, aby je bylo možné definovat jako závislost projektu v nástroji CMake.

V poslední kapitole byly popsány *navazující práce*. Bakalářská práce otevřela různá témata, na které je možné navázat, což primárně spočívá v dalším rozšíření emulace NES. Lze doplnit emulaci dalších obvodů pro mapování paměti, zvýšit věrohodnost emulace grafického čipu, doplnit emulované periferie a další. Dále se jedná o využití emulační platformy nejen pro vývoj emulátorů, ale například i pro výuku počítačových architektur pomocí demonstrace mikroarchitektur RISC-V přímo na platformě. Všechna další využití jsou podpořena podrobnou dokumentací, která je dostupná i ve webové verzi.

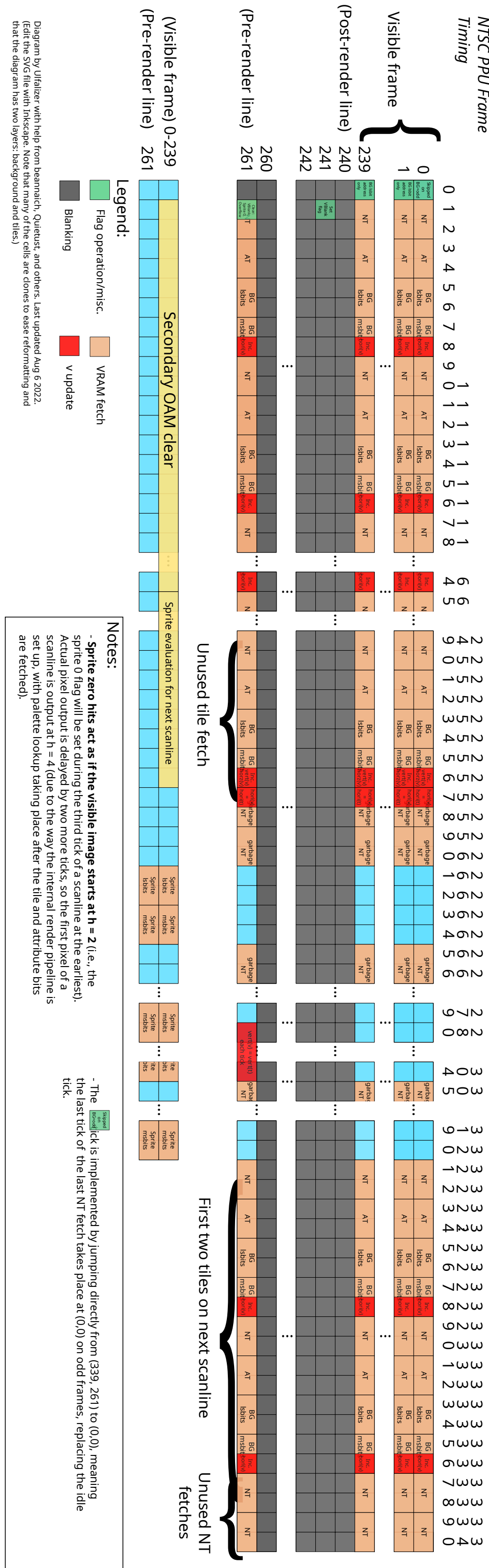
Bakalářská práce tedy splnila primární cíl definovaný v zadání. Emulátor NES je funkční, je schopen spouštět jednoduché i složitější programy pro originální konzoli, obsahuje přehledné rozhraní a byl otestován a srovnán s fungováním původního hardwaru. Primární cíle zadání byly doplněny o množství sekundárních cílů majících za úkol přinést alternativní metody výuky počítačových architektur, které byly také splněny dle očekávání.

Příloha A

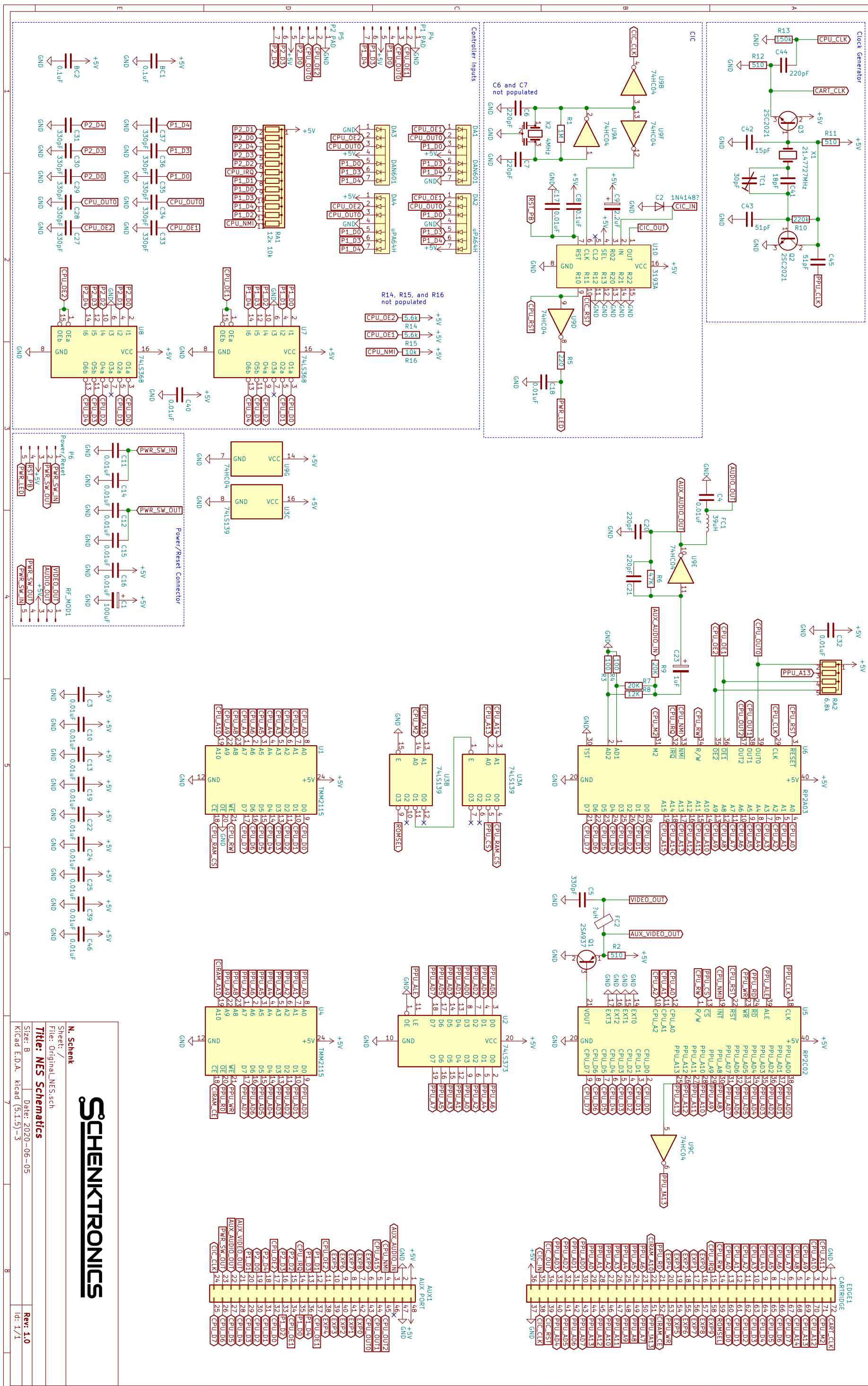
Diagramy ke konzoli NES

Tato příloha obsahuje dodatečné diagramy zjednodušující pochopení principu fungování konzole NES.

Diagram na obrázku A.1 ukazuje proces vykreslování v čipu PPU včetně operací na pozadí. V digitální verzi jde o vektorovou grafiku, tudíž je možné (a doporučené) obrázek libovolně přiblížit. Pro čtenáře tištěné bakalářské práce je připravena webová verze [53].



Obrázek A.1 Diagram procesu renderování u čipu 2C02 (PPU). Vytvořila komunita Nesdev.org [53].



Obrázek A.2 Přehledové hardwarové schéma konzole NES. Obrázek „NES-001 Console“ vytvořil schenkzoola [54] a zveřejnil pod licencí CC BY 4.0.

Příloha B
Knihovna ImInputBinder

Knihovna ImInputBinder vznikla během vývoje projektu Universal System Emulator jako rozšíření knihovny Dear ImGui. Slouží jako centrální bod odchytávání uživatelského vstupu přes klávesnici a herní periferie. Kromě samotného zpracování vstupu a vyvolání akce umožňuje také změnit přiřazení kláves a to ukládat do souboru. Návod k použití i samotná knihovna je k dispozici ve veřejném repozitáři: `https://github.com/andreondra/ImInputBinder`, zde následuje jen meditace nad návrhovými rozhodnutími, které byly během vývoje učiněny.

První otázkou byla míra dodržování návrhových principů knihovny Dear ImGui. Běžně jsou komponenty navrženy tak, že není nutná žádná reprezentace stavu, tedy není třeba vytvářet žádné instance tříd; stačí pouze zavolat statické metody. Ukázalo se, že aby byla komponenta konfigurovatelná, minimální vnitřní stav musí existovat, a to seznam akcí. Tento stav totiž vyžaduje nejen metoda pro vykreslení obrazovky nastavení přiřazení kláves, ale i metoda pro aktualizaci stavů stisku kláves a případné volání callbacků. I kdyby byly principy dodrženy a veškeré metody byly statické, musel by uživatel pomyslný vnitřní stav udržovat mimo a dodávat jej metodám při každém volání, což je mnohem horší varianta z hlediska výkonu, přehlednosti i OOP principů (zapouzdřenost).

Druhou otázkou byl způsob ukládání konfigurace. Uvažovalo se nad čistě textovým formátem, pomocí něj by se snadno a přenositelně reprezentovala čísla, avšak by mohl nastat problém u konců řádků (znaky carriage return a line feed), což každá platforma řeší jinak. Nakonec byl zvolen vlastní (binární) formát, který většinu hodnot ukládá jako ASCII text.

Příloha C

Klony NES

Za dobu existence konzole NES vzniklo mnoho klonů především její japonské verze Family Computer (Famicom). Tyto klony u nás byly dosti rozšířeny na přelomu tisíciletí [18]. V této příloze je pro zajímavost uvedeno několik fotografií těchto klonů. Právě díky klonům se systém NES proslavil u nás a dostal se takto i do mého povědomí.

Obrázek C.1 Klon „Super Megason IV“ obsahující kopii periferie NES Zapper. Foto © 2012 kevro, reddit.com.

■ **Obrázek C.2** Klon „Terminator 2“ firmy „Ending-Man“. Součástí balení jsou i pirátské kopie softwaru. Foto © 2021 Tempoulker, reddit.com.

■ **Obrázek C.3** Klon „Dendy Junior“, populární v Rusku. Foto © 2012 Nzeemin, wikimedia.org, CC BY-SA 3.0 [55].

Bibliografie

1. KUBÁTOVÁ, Hana. *Systémy reálného času: Realita, návrhová omezení. Globální čas.* Czech Technical University in Prague, 2019.
2. TEUSCHER, Christof. *Alan Turing: Life and legacy of a great thinker*. Vyd. 1. Berlin, Germany: Springer, 2003. ISBN 978-3-540-20020-8.
3. NASH, Henry. The design and development of a software emulator. In: *Digest of Papers* [online]. San Francisco, CA, USA: Institute of Electrical a Electronics Engineers, 1989 [cit. 2023-05-04]. ISBN 0-8186-1909-0. Dostupné z DOI: `10.1109/CMPCON.1989.301943`.
4. FULBER-GARCIA, Vinicius. *Differences Between Simulation and Emulation* [online]. Baeldung, 2022-12-16. [cit. 2023-03-12]. Dostupné z: `https://www.baeldung.com/cs/simulation-vs-emulation`.
5. ALLISON, Joanne. *Stored-program Computers* [online]. The University of Manchester, 1997-09-12. [cit. 2023-05-04]. Dostupné z: `https://web.archive.org/web/20110927012816/http://www.computer50.org/mark1/stored.html`.
6. KUBÁTOVÁ, Hana. *Struktura a architektura počítačů s řešenými příklady*. Vyd. 2. Praha: ČVUT, 2018. ISBN 978-80-01-06410-8.
7. JAMES, Greg; SILVERMAN, Barry; SILVERMAN, Brian. *Visualizing a Classic CPU in action: The 6502* [online]. 2010-07-27. [cit. 2023-03-13]. Dostupné z: `http://www.visual6502.org/docs/6502_in_action_14_web.pdf`.
8. PLUHÁČEK, Alois. *Projektování logiky počítačů*. Dotisk vyd. 1. Praha: České vysoké učení technické, 1995. ISBN 80-01-00813-4.
9. MOS TECHNOLOGY, Inc. *MCS6500 Microcomputer Family Hardware Manual* [online]. Vyd. 2. 1976. [cit. 2023-04-10]. Dostupné z: `http://archive.6502.org/books/mcs6500_family_hardware_manual.pdf`.
10. POYNTON, Charles. *Digital Video and HD: Algorithms and Interfaces*. Vyd. 2. Waltham (Massachusetts): Morgan Kaufmann, 2012. ISBN 978-0-12-391926-7.
11. NESDEV. *NTSC video* [online]. 2023-04-12. Ver. 105 [cit. 2023-04-18]. Dostupné z: `https://www.nesdev.org/w/index.php?title=NTSC_video&oldid=21035`.
12. PLANTZ, Robert G. *Introduction to Computer Organization: ARM Assembly Language Using the Raspberry Pi* [online]. Sonoma State University, 2021-06-15. [cit. 2023-03-20]. Dostupné z: `https://bob.cs.sonoma.edu/IntroCompOrg-RPi/frontmatter-1.html`.
13. REID, David. *Miniaudio Programming Manual* [online]. 2023. Ver. v0.11.14 [cit. 2023-04-25]. Dostupné z: `https://miniaud.io/docs/manual/index.html`.

14. TANENBAUM, Andrew S.; BOS, Herbert. *Modern Operating Systems*. Vyd. 4. Amsterdam: Pearson Education, 2015. ISBN 978-0-13-359162-0.
15. NESDEV. *Patents* [online]. 2009-11-15. Ver. 1 [cit. 2023-03-14]. Dostupné z: https://www.nesdev.org/w/index.php?title=Patents&oldid=10382.
16. WIKIPEDIA CONTRIBUTORS. *Nintendo Entertainment System — Wikipedia, The Free Encyclopedia*. 2023. Dostupné také z: https://en.wikipedia.org/w/index.php?title=Nintendo_Entertainment_System&oldid=1140676771. [Online; accessed 13-March-2023].
17. WIKIPEDIA CONTRIBUTORS. *Famiclone — Wikipedia, The Free Encyclopedia*. 2023. Dostupné také z: https://en.wikipedia.org/w/index.php?title=Famiclone&oldid=1132305316. [Online; accessed 13-March-2023].
18. ŠVÁRA, Ondřej. Dejte mi jedno Polystation, pane stánkař. *Hrej* [online]. 2009 [cit. 2023-03-13]. Dostupné z: https://hrej.cz/article/dejte-mi-jedno-polystation-pane-stankar.
19. FAIRBAIRN, Doug; DIAMOND, Stephen. *Oral History of Chuck Peddle* [dokumentární film]. Mountain View, CA, 2019 [cit. 2023-04-10]. Dostupné z: https://www.youtube.com/watch?v=enHF9lMseP8.
20. MOS TECHNOLOGY, Inc. *MCS6500 Microcomputer Family Programming Manual* [online]. Vyd. 2. 1976. [cit. 2023-04-11]. Dostupné z: http://archive.6502.org/books/mcs6500_family_programming_manual.pdf.
21. KAMBAYASHI, Yahiko. Logic Design of Programmable Logic Arrays. *IEEE Transactions on Computers* [online]. 1979, roč. C-28, č. 9 [cit. 2023-05-04]. Dostupné z DOI: 10.1109/TC.1979.1675428.
22. STEIL, Michael. *How MOS 6502 Illegal Opcodes really work* [online]. 2008-07-29. [cit. 2023-03-16]. Dostupné z: https://www.pagetable.com/?p=39.
23. NESDEV. *CPU unofficial opcodes* [online]. 2023-03-18. Ver. 44 [cit. 2023-03-16]. Dostupné z: https://www.nesdev.org/w/index.php?title=CPU_unofficial_opcodes&oldid=20812.
24. NESDEV. *CPU interrupts* [online]. 2021-02-16. Ver. 53 [cit. 2023-04-17]. Dostupné z: https://www.nesdev.org/w/index.php?title=CPU_interrupts&oldid=1329.
25. NESDEV. *PPU registers* [online]. 2023-03-31. Ver. 177 [cit. 2023-04-17]. Dostupné z: https://www.nesdev.org/w/index.php?title=PPU_registers&oldid=20980.
26. YLILUOMA, Joel. *NTSC NES palette generator* [online]. 2013. [cit. 2023-05-05]. Dostupné z: https://bisqwit.iki.fi/utils/nespalette.php.
27. NESDEV. *PPU attribute tables* [online]. 2023-03-18. Ver. 25 [cit. 2023-04-19]. Dostupné z: https://www.nesdev.org/w/index.php?title=PPU_attribute_tables&oldid=20795.
28. NESDEV. *PPU OAM* [online]. 2023-04-13. Ver. 89 [cit. 2023-04-19]. Dostupné z: https://www.nesdev.org/w/index.php?title=PPU_OAM&action=info.
29. NESDEV. *PPU scrolling* [online]. 2023-04-08. Ver. 62 [cit. 2023-04-20]. Dostupné z: https://www.nesdev.org/w/index.php?title=PPU_scrolling&oldid=21007.
30. NESDEV. *PPU sprite evaluation* [online]. 2022-11-05. Ver. 46 [cit. 2023-04-20]. Dostupné z: https://www.nesdev.org/w/index.php?title=PPU_sprite_evaluation&oldid=20004.
31. NESDEV. *Mapper* [online]. 2023-01-18. Ver. 213 [cit. 2023-04-17]. Dostupné z: https://www.nesdev.org/w/index.php?title=Mapper&oldid=20240.
32. NESDEV. *NROM* [online]. 2020-02-16. Ver. 32 [cit. 2023-04-17]. Dostupné z: https://www.nesdev.org/w/index.php?title=NROM&oldid=8673.
33. NESDEV. *MMC1* [online]. 2022-10-14. Ver. 73 [cit. 2023-04-17]. Dostupné z: https://www.nesdev.org/w/index.php?title=MMC1&oldid=19949.

34. NESDEV. *INES* [online]. 2023-01-18. Ver. 65 [cit. 2023-04-17]. Dostupné z: https://www.nesdev.org/w/index.php?title=INES&oldid=20243.

35. NESDEV. *Input devices* [online]. 2022-10-06. Ver. 46 [cit. 2023-04-17]. Dostupné z: https://www.nesdev.org/w/index.php?title=Input_devices&oldid=19920.

36. TAKANO, Masaharu. How the Famicom Was Born – Part 10. *Nikkei Electronics* [online]. 1995 [cit. 2023-05-04]. Dostupné z: https://glitterberri.com/developing-the-famicom-modem/.

37. NESDEV. *Standard controller* [online]. 2023-01-28. Ver. 71 [cit. 2023-04-17]. Dostupné z: https://www.nesdev.org/w/index.php?title=Standard_controller&oldid=20308.

38. NESDEV. *APU Frame Counter* [online]. 2022-07-02. Ver. 32 [cit. 2023-04-27]. Dostupné z: https://www.nesdev.org/w/index.php?title=APU_Frame_Counter&oldid=19540.

39. NESDEV. *APU Mixer* [online]. 2021-12-13. Ver. 14 [cit. 2023-04-27]. Dostupné z: https://www.nesdev.org/w/index.php?title=APU_Mixer&oldid=18963.

40. NESDEV. *APU Envelope* [online]. 2020-09-23. Ver. 16 [cit. 2023-04-28]. Dostupné z: https://www.nesdev.org/w/index.php?title=APU_Envelope&oldid=418.

41. NESDEV. *APU Length Counter* [online]. 2013-06-12. Ver. 30 [cit. 2023-04-28]. Dostupné z: https://www.nesdev.org/w/index.php?title=APU_Length_Counter&oldid=476.

42. NESDEV. *APU Sweep* [online]. 2022-06-12. Ver. 32 [cit. 2023-04-27]. Dostupné z: https://www.nesdev.org/w/index.php?title=APU_Sweep&oldid=19486.

43. NESDEV. *APU Pulse* [online]. 2022-10-06. Ver. 38 [cit. 2023-04-27]. Dostupné z: https://www.nesdev.org/w/index.php?title=APU_Pulse&oldid=19922.

44. NESDEV. *APU Noise* [online]. 2020-10-11. Ver. 20 [cit. 2023-04-27]. Dostupné z: https://www.nesdev.org/w/index.php?title=APU_Noise&oldid=519.

45. NESDEV. *Emulators* [online]. 2020-09-23. Ver. 226 [cit. 2023-02-20]. Dostupné z: https://www.nesdev.org/w/index.php?title=Emulators&oldid=20434.

46. MJBUDD77; ZEROMUS; G0ME3 et al. *fceux* [online]. 2023. [cit. 2023-05-05]. Dostupné z: https://github.com/TASEmulators/fceux.

47. JAVIDX9. *olcNES* [online]. 2023. [cit. 2023-05-05]. Dostupné z: https://github.com/OneLoneCoder/olcNES.

48. GOLASOWSKI, Ondřej. *Universal System Emulator* [online]. 2023. [cit. 2023-05-05]. Dostupné z: https://github.com/andreondra/use.

49. BROWN, Walter E. *A Proposal for the World's Dumbest Smart Pointer, v3* [online]. WG21 N3840. International Organization for Standardization, 2014 [cit. 2023-04-30]. Dostupné z: https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3840.pdf. Součást projektu JTC1.22.32 Programming Language C++.

50. DORMANN, Klaus. *6502 65C02 Functional Tests* [online]. 2020. [cit. 2023-05-05]. Dostupné z: https://github.com/Klaus2m5/6502_65C02_functional_tests.

51. NESDEV. *Game bugs* [online]. 2023-01-11. Ver. 122 [cit. 2023-05-04]. Dostupné z: https://www.nesdev.org/w/index.php?title=Game_bugs&oldid=20212.

52. NESDEV. *Emulator tests* [online]. 2023-03-17. Ver. 129 [cit. 2023-05-04]. Dostupné z: https://www.nesdev.org/w/index.php?title=Emulator_tests&oldid=20642.

53. NESDEV. *Ppu.svg* [online]. 2022-10-22. Ver. 11 [cit. 2023-05-05]. Dostupné z: https://www.nesdev.org/w/index.php?title=File:Ppu.svg&oldid=19964.

54. SCHENKZOOLA. *NES-001 Console* [online]. 2020. [cit. 2023-05-05]. Dostupné z: https://github.com/schenkzoola/NES/tree/main/NES-001%20Console.

55. NZEEMIN. *Dendy Junior with cart and joypads* [online]. 2012. [cit. 2023-05-05]. Dostupné z: https://commons.wikimedia.org/w/index.php?curid=114087394.

Obsah přiloženého média

- readme.txt ..... stručný popis obsahu média
- implementace
  - use-main.zip ..... kopie repozitáře s implementací projektu
- text
  - bp-main.zip ..... kopie repozitáře se sazbou textu ve formátu LaTeX
  - thesis.pdf ..... text práce ve formátu PDF