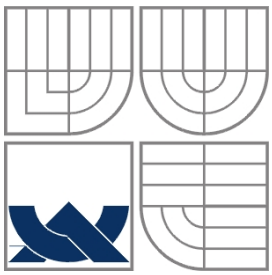
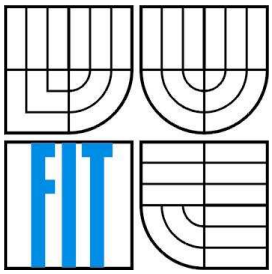


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

EMULÁTOR JEDNODUCHÉHO PROCESORU

EMULATOR OF SIMPLE PROCESSOR

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — PETR KUZNÍK
AUTHOR

VEDOUCÍ PRÁCE — Ing. JAKUB KŘOUSTEK
SUPERVISOR

BRNO 2010

Abstrakt

Emulátor bude navržen jako obecný, schopný emulovat různorodé architektury. Jednotlivé architektury budou v samostatných modulech implementovaných jako dynamicky linkované dll knihovny. Hlavním cílem je dosáhnout právě obecnosti emulátoru a navrhnout jeho strukturu takovým způsobem, aby bylo možné jednoduše přidávat nové architektury a s použitím již implementovaných abstrakcí tyto architektury vytvářet. Hlavní implementovanou architekturou bude Commodore 64, předchůdce dnešních osobních počítačů, používaný v 80. letech hlavně v USA.

Abstract

Emulator will be designed as generic emulator. It should be capable of emulating versatile architectures. Each architecture will be stored in separate module implemented as dynamically linked dll libraries. Main goal is for the emulator to be generic and design its structure in a way, so that it would be possible to easily add new architecture modules and design these modules with already implemented abstractions. Primarily implemented architecture will be Commodore 64. It’s a personal computer used mainly in USA during 1980s.

Klíčová slova

emulátor, procesor, instrukční sada, grafický výstup, Commodore 64, asembler

Keywords

emulator, processor, instruction set, graphic output, Commodore 64, assembler

Citace

Kuzník Petr: Emulátor jednoduchého procesoru, bakalářská práce, Brno, FIT VUT v Brně, 2010

Emulátor jednoduchého procesoru

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Jakuba Křoustka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Petr Kuzník
16.5.2010

Poděkování

Za pomoc děkuji vedoucímu práce Ing. Jakubu Křoustkovi.

Obsah

1 Úvod

Emulace je proces, při kterém je jeden systém napodobován druhým tak, aby jeho chování odpovídalo prvnímu systému. S tímto termínem je velmi úzce spojen pojem simulace. Dříve se pojem emulace omezoval spíše na hardware, který byl zkonstruován za účelem napodobit jiný hardware. Na softwarové úrovni se mluvilo o simulaci. Dnes se tyto pojmy stále více přibližují a emulací je dnes běžně rozuměn program napodobující jiný program nebo nějaký hardware. Emulace je pokus imitovat vnitřní návrh zařízení. Simulace je pokus imitovat funkce zařízení [1]. Emulovat je možné v podstatě jakékoli zařízení, které obsahuje mikroprocesor (např. kalkulačky, procesory, herní konzole a podobně).

Například ve Free On-Line Dictionary Of Computing je emulace definována jako: „Jeden systém pracuje naprosto stejně jako jiný, když ne nutně se stejnou rychlostí. Typickým příkladem by byla emulace jednoho počítače (programu na něm běžícím) jiným. Emulaci je možné použít k náhradě systému, zatímco simulaci je možné použít k analýze a vytvoření předpovědí o systému.“

Úkolem práce bude navrhnout vlastní emulátor. Tento emulátor by měl mít povahu obecného emulátoru, který bude schopný emulovat různorodé architektury. Jako součást práce bude implementována jedna architektura a to osobní počítač Commodore 64 používaný zejména v 80. letech v USA. Každá nová architektura bude implementována tak, že bude odvozena z již implementovaného abstraktního emulátoru. Tento abstraktní emulátor bude mít implementovanou velkou část chování společného pro většinu architektur a hlavně logiku spojenou s chodem emulace. V nové architektuře bude nutné doimplementovat pouze chovaní specifické pro danou architekturu. Protože vnitřní návrh architektur může být velmi odlišný a návrh abstraktního emulátoru nemusí vyhovovat požadavkům některých architektur je i tato abstrakce navržena tak aby bylo možné její chování jednoduše upravit

2 Emulátory

2.1 Využití emulátorů

Vývoj aplikací a jejich testování bývá obtížná činnost. To platí zejména pro jednoduché architektury. Na běžných (osobních) počítačích má programátor podporu operačních systémů, ty významně zjednodušují práci s počítačem samotným a rovněž poskytují spoustu funkcí, které usnadňují implementaci a redukují množství chyb v programech. Dále jsou k dispozici kvalitní vývojová rozhraní, která dále výrazně zjednodušují práci a umožňují ladění vyvíjených aplikací a krokování zdrojových kódů po jednotlivých instrukcích.

Pro velké množství jednoduchých procesorů toto neplatí, neběží na nich žádný operační systém, nebo jen velice primitivní, a i při využití vývojových prostředí pro tyto jednoduché architektury jsou možnosti ladění velmi omezené. Pro tyto účely může být velmi vhodné použít emulátor architektury, pro kterou je daná aplikace zamýšlena.

Emulátor může umožnit jednodušší nalezení chyb například díky výpisu obsahu paměti apod. Pokud emulátor podporuje např. krokování aplikace po instrukcích, programátor má další možnost přesněji sledovat svou aplikaci při práci což opět umožní lepší a rychlejší opravení případných chyb.

Další oblast, kde emulátory hrají nezastupitelnou roli je obor zvaný “Digital preservation” neboli ochrana digitálních dokumentů. Oblast multimédií a digitálních informací se velmi dynamicky vyvíjí. A je zde hrozba tzv. “Digital Dark Age”, kdy informace uložené před několika desítkami let nemusí být čitelné z důvodu absence zařízení, které jsou je schopny přečíst. Toto platí i pro zdrojové kódy programů a aplikace, které za několik let po svém vývoji již nemusí být spustitelné, protože architektury pro tyto aplikace už nejsou provozuschopné. Ideálním řešením v tomto případě jsou emulátory. Ty umožní spuštění staré aplikace na jiné architektuře, která je běžně používána [2].

2.2 Typy Emulátorů

Emulátory můžeme rozdělit na dva základní typy podle způsobu emulace. Prvním z nich je Interpretovaný emulátor. Z těchto dvou typů je univerzálnější ale pomalejší. Při tomto způsobu emulace dochází ke čtení instrukcí přímo z paměti, tak jak jsou uloženy v emulované architektuře a následnému vykonání. Druhý typ je rekompilovaný emulátor. V tomto případě je snaha převést program z původní podoby do nové, která je poté vykonávána. Transformovaný kód je pak jednoduší vykonávat, dokonce může být možné nový kód rovnou spustit bez použití emulátoru. S tímto typem

emulace jsou ovšem spojeny problémy, protože nemusí být vždy možné kód převést, problém je zejména se samo-modifikující se kódem [1].

Při návrhu jádra emulátoru bylo zvažováno použití mezikódu pro vykonávání instrukcí ale tato varianta nebyla nakonec použita. Mezikód by pravděpodobně urychlil vykonávání instrukcí a umožnil by další optimalizace spojené s během emulovaných programů. Nicméně pokud by návrh mezikódu nevyhovoval nějaké nově implementované architektuře znamenalo by to pravděpodobně velký problém při vývoji. Nicméně celkový návrh emulátoru je založen na možnosti upravit každou jeho část, proto by tento problém měl být řešitelný.

Větší problém při použití mezikódu by znamenal samo-modifikující se kód. V případě, že je emulován program obsahující takový kód stává se použití mezikódu velmi obtížné. Problém představuje skutečnost, že mezikód, který byl vygenerován před začátkem vykonávání programu nemusí být již několik instrukcí dále platný. Bylo by tedy nutné neustále kontrolovat aktuálnost vygenerovaného mezikódu. To by představovalo nutnost kontrolovat každou instrukcí zapisující do paměti a v případě, že tato instrukce mění programovou oblast upravit mezikód do nové podoby. Což by vyžadovalo poměrně velkou režii, jak samotné generování nové verze mezikódu za běhu, tak i kontrolování programových adres na jejich změny a tyto adresy nemusí být v paměti pohromadě, ale můžou být rozděleny na desítky bloků.

V dnešní obě nemusí být použití samo-modifikující se kódu tak časté, protože dnešní procesory mají většinou k dispozici dostatečné množství paměti a poměrně kvalitní instrukční sadu. Protože samo-modifikující se kód dělá program hůře čitelným a méně srozumitelným je snaha se tomuto programování vyhnout. Starší procesory ovšem mívaly s pamětí problém a použití samo-modifikující se kód je častější. Běžně je takto v programech upravována adresa skokových instrukcí. Nebo je programový kód kopírován na adresy, které mají optimalizovaný přístup do paměti a jeho vykonávání zde může být rychlejší.

2.3 Metody použité ve vyvíjeném emulátoru

Emulátor má pracovat jako obecný, schopný emulovat množství rozdílných architektur. Pří návrhu emulátoru pro jednu architekturu je možné provést velké množství optimalizací, které tuto architekturu zohledňují, které vycházejí ze specifických vlastností této architektury. Při návrhu obecného emulátoru ale zavedení nějaké optimalizace může znamenat problém při implementaci jiné architektury, jejíž vnitřní struktura je odlišná a daná optimalizace může znemožnit správné fungování některé její části. Proto generické část emulátoru neobsahuje, žádné optimalizace, které by omezovali vnitřní strukturu procesoru nebo jeho instrukční sadu. Návrh je založen na izolovaných blocích, ty

jsou navrženy co nejobecněji a v případě nutnosti je možné vytvořit novou implementaci těchto bloků nebo jejich částí.

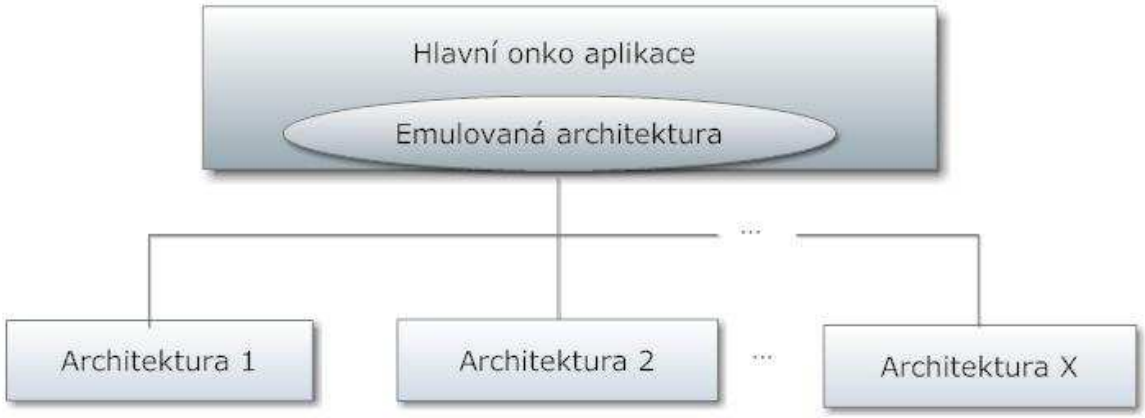
Emulátor není navržen s ohledem na maximální rychlost, efektivnost nebo s ohledem na nízkou paměťovou náročnost. Všechny části návrhu se podřizují generičnosti. Je kladen důraz na to, aby byl vnitřní návrh emulátoru přehledný a vývoj dalších architektur byl s použitím již definovaných abstraktních šablon co nejjednodušší.

Protože je emulátor zamyšlen pro jednoduché, často starší procesory, mezikód není použit, aby nevznikaly problémy se samo-modifikující se kódem. Vzhledem k zaměření emulátoru na různorodost emulovaných architektur a programů na nich běžících, jsou rychlost a další výhody mezikódu druhořadé. Při emulaci je tedy vykonávaný kód načítán přímo z paměti, čímž nevznikají problémy se samo-modifikující se kódem a emulátor blíže reprezentuje skutečnou strukturu emulovaného procesoru. Jedná se tedy o interpretovaný emulátor.

3 Návrh Emulátoru

Emulátor je rozdělen do několika izolovaných návrhových celků zastupujících hlavní části emulované architektury, jako jsou procesor, paměť nebo grafický výstup popřípadě některé další. Všechny tyto části jsou obsaženy v hlavním bloku emulátoru, který je zodpovědný za chod emulátoru a koordinaci těchto částí. Tento blok zastupuje celou emulovanou architekturu a představuje modul ve formě dll knihovny, který je možno dynamicky připojit k aplikaci. Uživatel poté z dostupných modulů vybírá konkrétní architekturu, kterou si přeje připojit do emulátoru.

Emulátor je zamýšlen pro emulování více různorodých architektur a tomu je i podřízen celý jeho návrh. K vývoji modulů dalších architektur je vhodné použít abstraktních tříd zastupujících jednotlivé časti nové architektury (paměť, procesor …). Tyto abstraktní třídy již implementují značnou část chovaní těchto prvků. Abstraktní prvky a komunikace mezi nimi je navržena tak aby byla co nejjednodušší a nejuniverzálnější a dané moduly na sobě byly vnitřně naprosto nezávislé. Tím je možné kombinovat prvky, které mohly být původně zamýšleny pro jiné architektury, ale jejich chování odpovídá danému účelu, a byly vytvořeny na sobě naprosto nezávisle. I proto jsou při vývoji často použity konstrukty jazyka C#, které jsou pomalejší a výpočetně náročnější než jejich alternativy(pro komunikaci mezi moduly by mohli být použity reference modulů a volání metod místo událostí) ale umožňují vývoj nezávislejších a obecnějších součástí aplikace.



Obr. 1 Hlavní aplikace načítá z externích modulů emulované architektury. Uživatel si poté může vybrat z nabídky dostupných architektur.

3.1 Jádro emulátoru

Tento modul zastřešující jednu emulovanou architekturu. Modul řídí emulátor na poměrně vysoké úrovni abstrakce. Je zde abstrahována instrukční sada, struktura paměti, i forma výstupu. Chod emulátoru je řízen na úrovni vykonávání instrukcí, přitom není známo jakou má instrukce podobu a jak její provedení změní stav emulátoru. Toto je definováno v částech emulátoru implementujících chování paměti a zejména procesoru, popřípadě některých dalších. V jádře emulátoru bude také implementována celá problematika ladění a krokování aplikace. Při implementaci nové architektury je samozřejmě možné zasáhnout a upravit tuto vlastnost emulátoru pokud je potřeba dosáhnout nějakých speciálních vlastností. Ale základní implementace by měla dostačovat pro všechny běžné potřeby krokování a ladění aplikace. Základní implementace umožňuje spuštění emulátoru, krokování po jednotlivých instrukcích, získání stavu procesoru a jeho registrů, a zobrazení libovolné části paměti. Forma zobrazení stavu procesoru není přesně definovaná, protože každý procesor má specifickou sadu registrů a příznaků stavového registru. Je tedy definována univerzální podoba předávání stavu procesoru, ta bude zmíněna později.

Tento blok je rovněž zodpovědný za vytvoření všech svých částí, jejich řádně propojení a spuštění jejich inicializací. Při implementaci vlastní architektury je nutno dodržet danou konvenci inicializace aby nedocházelo k propojování nebo inicializaci ještě nevytvořených částí zejména v případě, kdy jednotlivé části ještě provádějí inicializaci vůči sobě (grafický výstup například může potřebovat nastavit v paměti úsek, ze kterého přijímá data jdoucí na výstup).

3.2 Procesor

Vlastnosti jednotlivých procesorů, jejich instrukční sady, stavové příznaky můžou být do značné míry rozdílné. I sémantika některých instrukcí může být rozdílná. Tento blok je navržen s ohledem na tyto skutečnosti. Implementace opět poskytuje základní chování společné pro většinu procesorů. Podporuje jak trvalé vykonávání instrukcí, tak krokování po jednotlivých instrukcích. Rovněž poskytuje prostředky pro zveřejňování stavu procesoru a nastavení stavových příznaků.

Protože rozdíly mezi procesory mohou být poměrně velké je při implementaci vlastního procesoru nutné definovat mnohem větší část chování pro tento blok než je tomu například u paměťového bloku. Je nutné nadefinovat registry procesoru, stavové příznaky a jejich chování. Rovněž je nutné definovat instrukční sadu a implementovat chování jednotlivých instrukcí. Právě kvůli rozdílnosti procesorů a jejich instrukčních sad, i délky instrukcí se mohou lišit nejen u různých procesorů ale i jednotlivé instrukce na jednom procesoru mohou mít rozdílnou délku a procesory mohou poskytovat několik adresových módu, by bylo velmi obtížné zavést nějaký ucelený formát implementace. Snaha zavést nějaký takový formát, který by již obsahoval předdefinované vlastnosti, pro všechny procesory by pravděpodobně vedla k velké složitosti této třídy a při vytváření

procesorového modelu by byla spíše matoucí. Nebo by implementace tímto způsobem byla výpočetně náročná.

Vykonávání instrukcí je proto rozděleno do několika metod, jako je identifikování instrukce, přečtení parametrů instrukce a podobně, které poskytují jistou šablonu, která tuto činnost zpřehlední a usnadní její implementaci, nicméně přesná podoba implementace je v režii autora aby bylo možné zachovat flexibilitu vývoje pro různorodé procesory.

Pro urychlení a usnadnění vývoje modulu procesoru jsou k dispozici statické metody, které implementují některé běžné chování procesorů, jako je nastavení stavových příznaků pro přetečení, bitový přenos nebo implementují chování běžných instrukcí, které je z velké části neměnné u většiny procesorů. Tyto metody je možné použít pokud odpovídají chování cílové architektury a není tedy nutné vytvářet vlastní implementace.

Protože značná část procesorů obsahuje zásobník, v modulu je implementován i on. Je možné nastavit počáteční adresu jeho velikost i směr růstu zásobníku. Před použitím je třeba zásobník inicializovat.

3.3 Paměť

Paměťový modul reprezentuje fyzickou paměť emulované architektury. Modul má jednoduché rozhraní, které umožňuje výběr a uložení dat z adresy, popřípadě bloku dat. Vzhledem k tomu, že jednotlivé bloky emulátory jsou na sobě nezávislé musejí mít bloky možnost zjistit změnu hodnoty na určitých adresách. Například grafický výstup musí mít možnost zjistit změnu na adresách, definujících výstup na obrazovku. K tomuto účelu slouží tzv. Memory trigger (paměťový spouštěč). Každý Memory trigger je definován adresami, které sleduje a cílem, kterému je oznámena změna. V případě jakékoli změny na sledovaných adresách, je tato skutečnost oznámena cílovému objektu Memory triggeru. Tento objekt pak může na danou změnu zareagovat.

Paměťový modul má dva základní modely pro organizaci paměti v emulované architektuře. Jednodušší z nich se nazývá PlainMemory, druhý je Multi Layer Memory. Oba tyto modely vycházejí z třídy MemoryBase, ta implementuje obsluhu Memory triggerů.

3.3.1 PlainMemroy model

Obsahuje jednoduchý úsek paměti, ke kterému modul umožňuje přístup. Model nepodporuje žádné složité uspořádání paměti. Počáteční adresa je nula a je možné definovat pouze velikost paměti.

3.3.2 Mutli Layer Memory

Model je složitější než Plain Memory model. Tento model umožňuje mapovat více vrstev paměti přes sebe. Je tak možné mít více paměťových bloků na stejných adresách (často např. střídání ROM a RAM pamětí). Tento paměťový model je složen z jednotlivých paměťových bloků, které mají svoji počáteční adresu a velikost. Každý blok je identifikován pomocí ID bloku a může mít vlastnost ReadOnly („jen pro čtení“), díky tomu je možné zachytit pokus o zápis do nepovolených oblastí jako např. ROM paměť. V tomto modelu se jednotlivé Memory triggery nevztahují k celému paměťovému modulu, ale k jednotlivým blokům. Je tak zaručeno, že při změně na adrese, kterou Memory trigger sleduje dojde k jeho aktivaci pouze pokud je do paměti mapováno zařízení, pro které trigger změny sleduje. O mapování bloků se stará tzv. Block Mapper. Jde o metodu v paměťové třídě. Tato metoda musí být definována autorem, čímž je umožněna variabilita způsobu mapování. Jako klíč pro mapování bloků je použito jeho ID.

3.3.3 Memory Trigger

Memory trigger je abstrakce ulehčující komunikaci mezi pamětí a ostatními moduly emulátoru. Slouží k uvědomění modulů o změnách na paměťových adresách, které daný modul potřebuje sledovat a vědět o jejich změnách. Pomocí tohoto nástroje lze sledovat změny jak jedné adresy tak na větším paměťovém úseku. Každý Memory trigger musí mít vždy definovány tři základní atributy, adresu, popřípadě začátek adres, a koncovou adresu, na které má být citlivý. Poté musí obsahat cíl, který bude uvědomen v případě změny.
Každý memory trigger je většinou vytvářen v modulu který si přeje reagovat na změny paměti. Tento modul vytvoří instanci memory triggeru a vyplní počáteční a koncovou adresu citlivosti, cíl každého triggeru je delegát jazyka C# ukazující na metodu daného modulu. Tato metoda je volána v případě aktivace triggeru. Instance může být ještě doplněna o popis, který zlepšuje přehlednost v případě použití většího počtu triggerů.

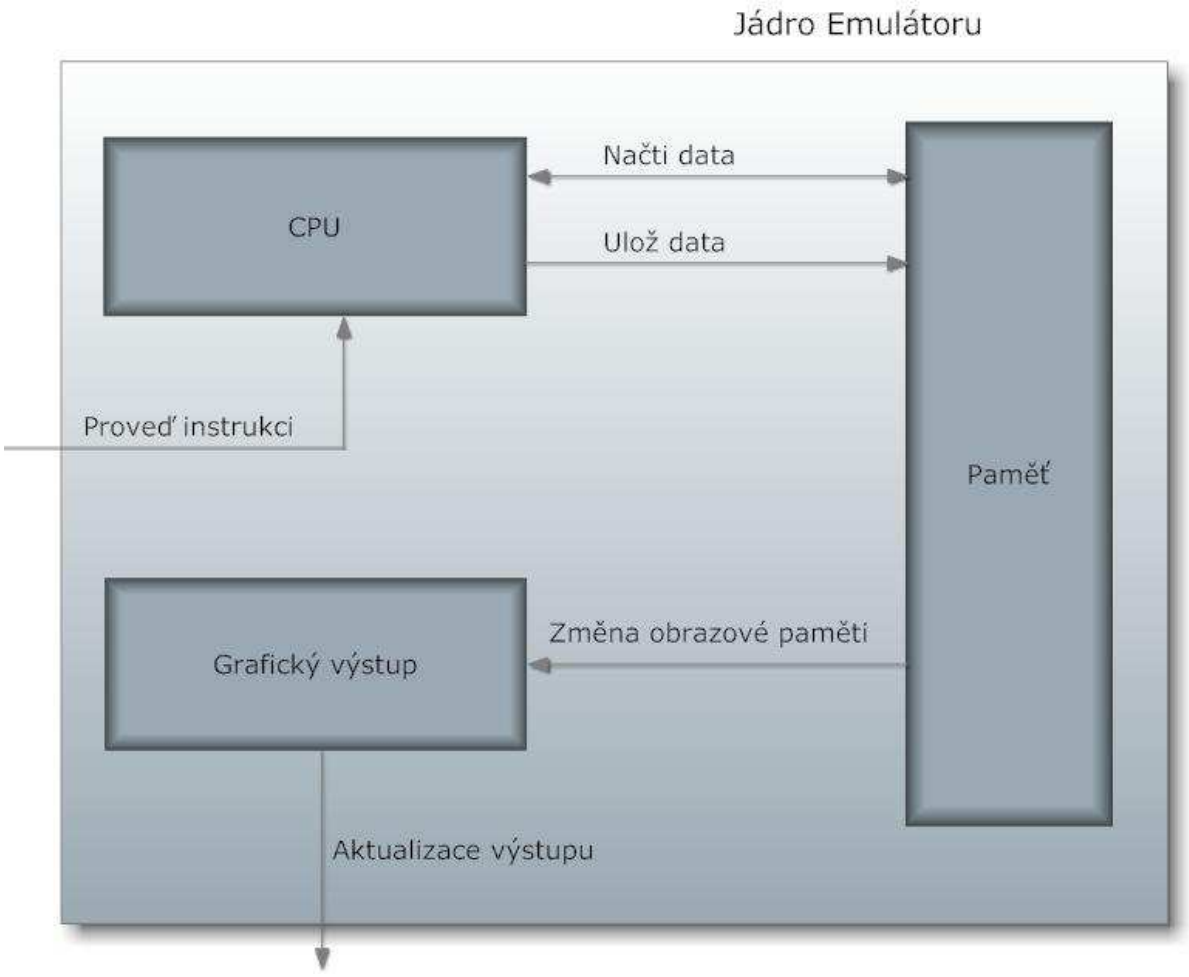
3.4 Text Graphic Output

Modul zajišťuje grafický výstup emulátoru jako grafického displeje v textovém módu. Poskytuje základní prostředky pro vykreslování grafiky založené na znakových primitivách. Je definován několika základními parametry jako jsou rozlišení ve znacích, rozlišení jednoho znaku a podobně. Modul má přístup k paměťovému prostoru, který bere jako zdroj dat pro vykreslování výstupu. V paměťovém modulu definuje oblast, na kterou je upozorněn v případě změny některé adresy a aktualizuje výstup emulátoru. Kromě odkazů na již zmíněnou obrazovou paměť, ta je vykreslována na výstup, obsahuje modul ještě znakovou paměť, ta definuje podobu znaků. Pokud není žádná

přiřazena předpokládá modul implicitní znakovou sadu (znaky ASCII). Při vytvoření je tomuto modulu předán odkaz na vykreslovanou plochu.

Podobně jako procesory se grafické čipy můžou od sebe velmi lišit. Nepředpokládá se proto použití toho modulu v jeho základní podobě pro složitější grafické čipy. Ale předpokládá se převzatí implementace a úprava nekompatibilních částí podle specifikace konkrétního grafického čipu.

Proto tento modul poskytuje ve své základní podobě několik primitivních vykreslovacích metod jako jsou vykreslení jednoho bytu, nebo znaku a podobně. Které by mely být univerzální pro většinu čipů. Přesné použití těchto metod a jejich parametry jako např. barva už záleží na konkrétní implementaci.



Obr 2. Schéma spolupráce jednotlivých modulů při vykonávaní instrukce. Jádrem emulátoru je po procesoru vyžádáno vykonání instrukce. V případě, že instrukce pracuje s pamětí, je zpaměti vyzvednuta nebo do paměti zapsána hodnota na požadovanou adresu. Pokud je daná adresa součástí obrazové paměti, je to oznámeno modulu grafického čipu a ten aktualizuje výstup.

4 Prostředí emulátoru

Uživatelské rozhraní emulátoru je složeno ze dvou hlavních částí. První částí je Hlavní okno emulátoru. Toto okno obsahuje pouze grafický výstup emulátoru a menu. Při spuštění programu okno vybízí uživatele na vybrání emulované architektury. Toto okno dále zajišťuje uživatelský vstup. Klávesy, zmačknuté v tomto okně jsou předávány emulovanému modulu. Jinak toto okno neposkytuje žádnou interakci s uživatelem. Druhým podstatným oknem je Status/Control window. Stavové/Kontrolní okno. Zde je možné provádět krokování aplikace a přesněji ovládat její průběh, tak i sledovat stav emulovaného zařízení.

4.1 Status/Control Window

Status Windows slouží k zobrazování stavu emulátoru. Je zde zobrazován obsah registrů procesoru, stavové příznaky, ladící informace. Je možné nahlížet do paměti a sledovat zdrojový kód. Okno umožňuje také nastavit hodnoty registrů procesoru.

Z tohoto okna je možné spustit simulaci nebo provést jednu instrukci. Při každém zastavení emulátoru, provedení instrukce nebo zastavení na break pointu (pokud program dosáhne adresy označené break pointem emulace je zastavena), je aktualizován stav procesoru, je znovu načten náhled do paměti, pokud je zobrazen, a je aktualizován náhled na vykonávaný kód.
Celé okno Status/Control Windows je rozděleno na čtyři hlavní části.

4.1.1 Status Widget (stavové okno)

V tomto okně je možné zobrazovat a upravovat stav procesoru. Okno je použito k zobrazení zejména registrů a stavových příznaků procesoru, popřípadě některých dalších informací. Vzhledem k různorodosti procesoru nemá toto okno přesně definovanou strukturu a je dodáváno v modulu s emulovanou architekturou. Jeho implementaci provádí autor modulu. K vyzvednutí informací z procesoru slouží metoda GetStatus() třídy EmulatorShellBase. Více o implementaci toho okna bude uvedeno dále.

4.1.2 Debug Windows (ladící okno)

Toto okno zobrazuje ladící informace týkající se běhu emulátoru a vykonávání instrukcí rovněž je zde umožněno přidávat a odebírat break pointy. Debug window zobrazuje operační kód a adresu poslední vykonané instrukce a kód a adresu příští instrukce. Poslední položkou je adresa nejbližšího příštího break pointu.

4.1.3 Code/MemoryView (Náhled na kód a paměť)

Okno obsahuje jeden TabControl, který může zobrazovat buď náhled na paměť nebo zobrazovat vykonávaný kód.

V náhledu na paměť je obsah paměti zobrazován po skupinách 16 bytů. Je možno si vybrat počáteční adresu a velikost bloku paměti, který bude zobrazen. Data jsou reprezentována v šestnáctkové soustavě. Hranice zobrazované paměti jsou vždy zarovnány na 16 bytů, dolů pro první adresu a nahoru pro délku zobrazovaného bloku.

V náhledu na vykonávaný kód jsou zobrazovány instrukce procesoru na jednotlivých adresách. Pro každou instrukci je zobrazeno její jméno, parametry, adresa a její kód v bytové podobě.
Je zvýrazněna instrukce, která bude vykonána jako další a jsou zvýrazněny i break pointy na jednotlivých adresách. V náhledu je možné break pointy i odstraňovat a přidávat. Podobně jako u paměti je možné zadat rozsah adres, na kterých bude zdrojový kód zobrazen. Opět formou první adresa, a délka paměťového bloku.

Pro vykonávaný program často nemusí být k dispozici zdrojový kód, protože program je načten v binární podobě nebo kód mohl být změněn programem samotným. Je zobrazovaný kód přímo dissasemblován z paměti. V případě, že je uživatelem požadováno zobrazit kód na adrese, která neobsahuje platný kód, disasemblování pokračuje i při nalezení neplatné instrukce a snaží se platný kód najít.

4.1.4 Ovládací prvky emulátoru

Tato část obsahuje ovládací prvky emulátoru. Tlačítka pro spuštění emulátoru, jeden krok emulátoru (provede jednu instrukci) a načtení dat do paměti.
Tlačítko Step – Emulátor provede jednu instrukci
Tlačítko Run – Spustí emulátor, změna tlačítka na stop
Tlačítko Stop – Korektně přeruší emulaci.
Tlačítko Stop + Ctrl – dojde k násilnému přerušení emulace, není vhodné používat, může způsobit nekorektní stav emulátoru.
Tlačítko Load – Umožňuje uložit do paměti data.

5 Implementace základních modulů

Emulátor bude implementován v jazyce C#. Jedná se o objektově orientovaný jazyk, vyvinutý společností Microsoft používající platformu .NET. Jazyk se může zdát méně vhodný pro implementaci emulátoru při porovnání s jazyky nižší úrovně, třeba i jazyky jako např. C++, zejména z důvodu absence pointerů, které se dají s výhodou použít při operacích nad malými skupinami bytů.

Nicméně emulátor je zamýšlet pro emulaci starších nebo jednodušších, tedy výrazně pomalejších procesorů a architektur. Kde případná vyšší režie nízko-úrovňových operací nebude znamenat zásadní problém. Pokud by bylo potřeba pro nějakou architekturu optimalizovat operace na nižší úrovni, je možnost zapouzdřit tyto operace do jiného modulu a implementovat metody těchto operací např. v C++/CLI.

Jazyk má velkou výhodu oproti jiným i v tom, že pro něj existuje velice kvalitní vývojové prostředí a to Microsoft Visual Studio. Vzhledem k tomu, že se předpokládá, že pro emulátor může být vyvíjeno více modulů pro emulaci různých architektur, přítomnost kvalitního vývojového prostředí je velkou výhodou zejména pro další potenciální vývojáře.

Implementace bude rozdělena do několika provázaných celků. Tyto celky budou reprezentovat jednotlivé části emulované architektury, které je možné od sebe oddělit a provázat je poměrně jednoduchým způsobem komunikace(displej, paměť apod.). Každý takový celek bude reprezentován třídou jazyka C#, která bude mít implementovány charakteristické vlastnosti pro daný celek. Toto umožní další použití těchto celků u podobných architektur s malými změnami specifickými pro danou architekturu.

Vhodné by bylo vytvořit jakýsi balík základních celků, jejichž úpravami bude možné implementovat emulátor pro novou architekturu. V některých jednodušších architekturách pak změn v základních blocích může být minimum a vývoj emulátoru je velmi rychlý a pohodlný.

5.1 Jádro Emulátoru

Jádro emulátoru je implementováno jako abstraktní třída s názvem EmulatorShellBase. Zveřejňuje několik základních metod pro komunikaci s emulátorem jako je vykonání instrukce, spuštění emulace, získání a nastavení stavu procesoru. Dále emulátor poskytuje možnost uložit do paměti zdrojové soubory aplikací. Při vývoji modulu pro novou architekturu je základním požadavkem zdědění této třídy. A inicializace jednotlivých součástí vhodnými implementacemi a jejich vzájemné propojení.

Implementace EmulatorShellBase umožňuje vykonávání instrukcí i souvislý běh emulátoru vykonávat synchronně nebo asynchronně, v samostatném vlákně. Tato vlastnost není tak důležitá při krokování aplikace po jednotlivých instrukcích ale pro souvislé vykonávání instrukcí je vhodné používat asynchronní přístup. Asynchronní metody mají prefix Async před svým jménem. AsyncRun() zahájí asynchronní vykonávání instrukcí a AsyncExecute() asynchronně provede jednu instrukci. Po započetí vykonávání instrukcí EmlatorShellBase vyvolá událost ExecutionStartedEvent.

V případě běhu může být emulátor kdykoli zastaven, k tomu slouží metoda StopEmulation(). V případě, že nastala chyba a emulátor není možné zastavit je k dispozici metoda ForceAbortExecution(), tato metoda by neměla být použita, pokud to není nezbytně nutné, protože není zaručeno, že právě vykonávaná operace bude korektně ukončena. Emulátor se tím může dostat to nekorektního stavu. Po ukončení vykonávání emulátor vyvolá událost ExecutionSuspendedEvent.

5.2 Implementace Procesoru

Procesor je opět implementován jako abstraktní třída. Tato obsahuje implementaci pro vykonávání jednotlivých instrukcí. Zveřejňuje pouze metody pro vykonání instrukcí a získání a nastavení stavu procesoru. Samotné vykonávání instrukcí je rozděleno do několika soukromých metod, ty je nutné pro novou architekturu implementovat. Základem je metoda ExecuteInstruction, ta podle registru Program Counter vyzvedne kód následující instrukce a najde pro ni implementaci. Následně jsou přečteny parametry instrukce, pokud jsou potřeba. Po provedení instrukce je inkrementován registr Program Counter a následně může být vykonána další instrukce.

Třída obsahuje pouze dva registry a to právě Program Counter register, udávající adresu právě vykonávané instrukce a registr StackPointer, který ukazuje na první prázdnou pozici zásobníku. Další registry jsou doplněny v implementaci odvozených tříd podle parametrů procesoru.

CPUBase obsahuje již implementovaný zásobník, tato implementace by měla být dostačující pro většinu procesorů. Před použitím je třeba nastavit počáteční adresu zásobníku, jeho velikost a směr růstu pomocí inicializační metody InitializeStack.

5.3 Implementace Paměťového modulu

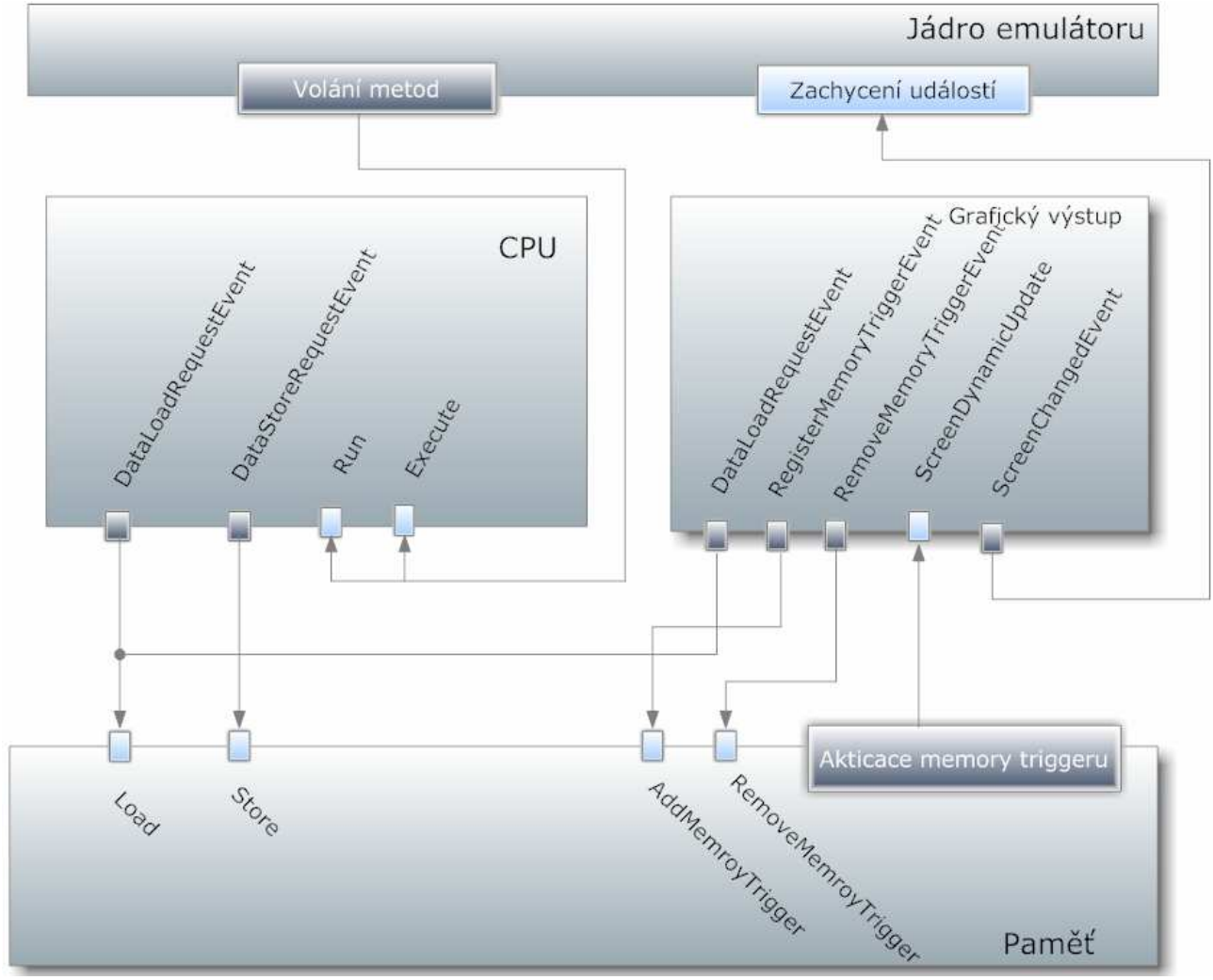
Paměťový modul je implementován jako třída MultiLayerMemoryBase obsahující seznam jednotlivých paměťových jednotek. Každá z těchto paměťových jednotek obsahuje počáteční adresu, její velikost, bytové pole reprezentující data a seznam memory triggerů. Při přístupu k datům se

paměťový prostor jeví jako souvislý. I v případě přístupu k většímu objemu dat, kdy se požadovaný adresový prostor může nalézat ve více paměťových blocích. Přístup k jednotlivým blokům je mapován pomocí metody BlockMapper. Tuto metodu je nutné implementovat podle způsobu mapování paměti v cílové architektuře.

5.4 Implementace grafického výstupu

Modul grafického výstupu je implementován jako třída TextGraphicOutput. Po vytvoření je modulu nutné předat bitmapu do které bude výstup vykreslován. Třída obsahuje metody pro řízení vykreslování znaků na tento grafický výstup. Implementace je vytvořena tak aby bylo možné vykreslit celou obrazovku nebo reagovat na změny, které nastanou pouze na jenom bytu, a nebylo nutné vždy aktualizovat celý výstup. Při každé změně obrazové paměti je tedy volána metoda, která aktualizuje výstup na místě odpovídajícím pozici dané adresy na obrazovce.

Třída samotná přímo neobsahuje metody pro zápis na výstup na úrovni nastavování jednotlivých pixelů. Tyto metody jsou umístěny ve statické třídě QEmulatorDrawing. Tyto metody je pak možno s výhodou použít při vývoji odvozených modulů.



Obr 3 Schéma propojení jednotlivých bloků emulátoru. Jsou zobrazeny pouze nejdůležitější události a metody jednotlivých bloků.

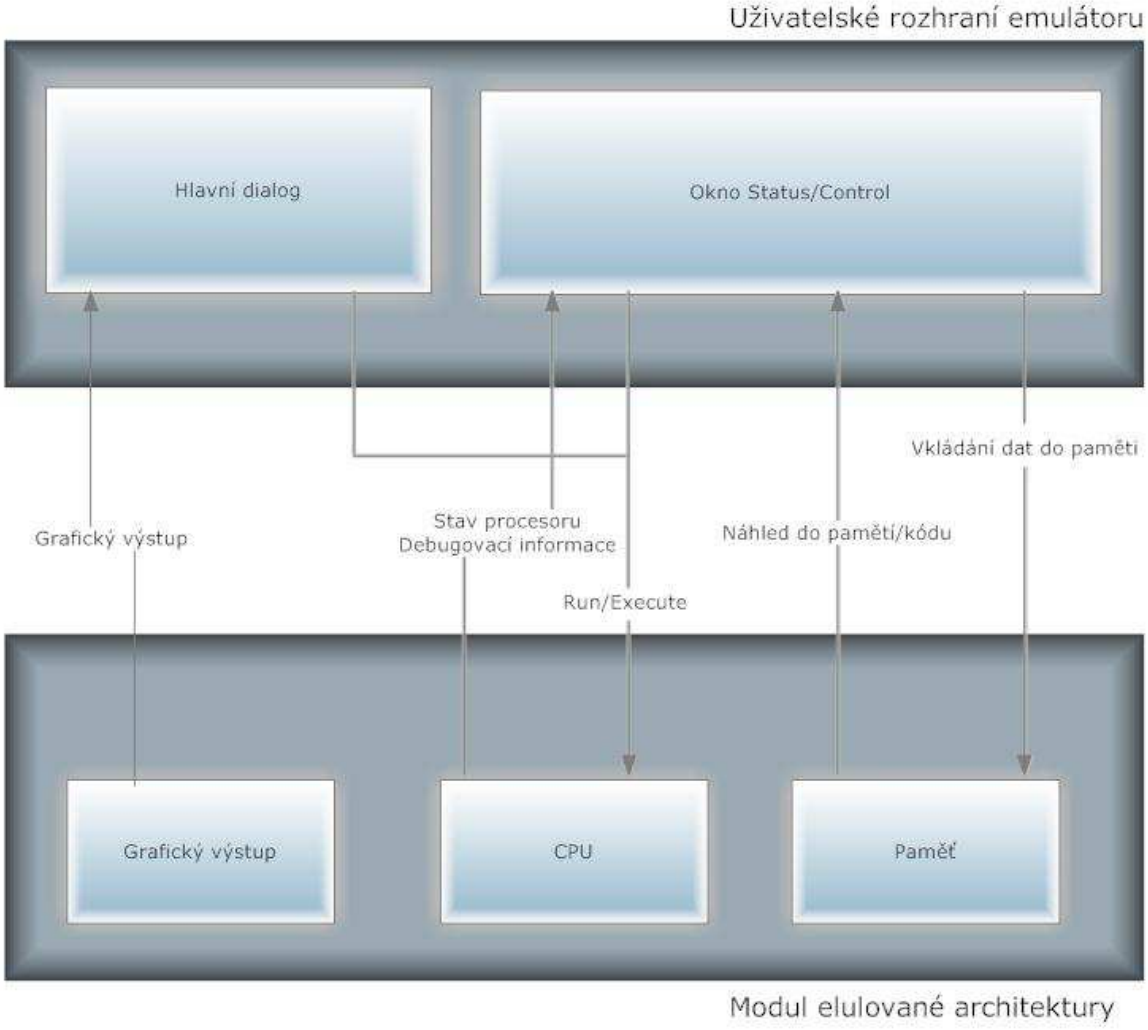
5.5 Další vlastnosti emulátoru

Emulátor poskytuje přístup ke stavu emulátoru a nastavení break pointů a debugování. Přístup ke stavu emulátoru je definován přes interface IStatusControl. Tento interface slouží k získání hodnot registrů procesoru a stavových příznaků, tak k jejich nastavování. Přes tento interface je rovněž možné manipulovat s break pointy. Interface deklaruje metody pro přidání nebo odebrání break pointu na libovolné adrese. Interface rovněž umožňuje přepnout break point na adrese, pokud na dané adrese break point je, je odstraněn, pokud není, je přidán.

Pro zjišťování stavu procesoru a jeho nastavování slouží v IStatusControl dvě metody a to SetStatus() a GetStatus(), tyto metody používají k přenosu stavu třídu Dictionary, jako název

položky(registr nebo stavový příznak) typ string a datový typ long, díky tomu není nijak omezeno jakou vnitřní podobu má mít procesor a je možné předat stav procesoru s libovolným počtem regisrtů a jejich jmény.

IStatusControl interface rovněž deklaruje metody pro získání debugovacích informací z procesoru. Tyto informace jsou v základní podobě neměnné a jsou předávny v rámci struktury DebugInformation. Tato struktura obsahuje informace o poslední vykonané instrukci, následující instrukci, jejich adresy, je zde i informace o pozici nejbližšího break pointu.



Obr 4. Propojení prvků grafického uživatelského rozhraní s jednotlivými logickými celky modulu emulátoru. Jde pouze o ilustrativní schéma. Veškerá komunikace uživatelského rozhraní s emulátorem prochází přes jádro emulátoru (třída EmulatorShellBase).

6 Implementace nových modulů s využitím abstraktních modulů emulátoru

Všechny moduly jsou hlavní aplikaci předávány jako dll knihovny. Každý takový modul musí obsahovat minimálně jádro dané emulované architektury reprezentované třídou odvozenou od EmulatorShellBase. Dále pak může obsahovat další komponenty jako je vlastní assembler a stavové okno procesoru, popřípadě i další komponenty. Podobně jako jádro emulátoru je nutné vytvořit ostatní součásti architektury jako je paměť, procesor atd. odvozením jejich tříd.

6.1 Implementace modulu procesoru

Každý procesorový modul musí vycházet z abstraktní třídy CPUBase. Nová implementace musí být třída zděděná z této abstrakce. Instance této třídy bude vykonávat instrukce procesoru. Kvůli různorodosti instrukčních sad a procesorů musí uživatel nadefinovat instrukční sadu procesoru jeho registry a jeho další prostředky, které budou emulovány.

CPUBase definuje několik stěžejních metod, jež spolupracují na provedení instrukce. Tyto metody je třeba vhodně naimplementovat tak aby jejich chování odpovídalo požadovanému chování.

6.1.1 Metoda ExecuteInstructionHandler

Jde o nejdůležitější metodu této třídy. Metoda slouží k identifikaci a vykonání instrukce. Implementace metody musí podle registru PC (Program Counter) vyzvednout z paměti operační kód instrukce, která se má vykonat a provést tuto instrukci. Metoda nastavuje stavové příznaky podle sémantiky instrukce a provádí výběry a uložení z a do paměti aritmetické operace a další úkony spojené s vykonáním instrukce. Samotné provedení instrukce jako je přístup do paměti, nebo např. aritmetické operace je vhodné implementovat do dalších funkcí, což činí metodu přehlednější.

Při implementaci je možné použít již implementované statické metody pro nastavování běžných stavových příznaků jako je carry flag nebo overflow flag. Nebo statické metody pro vykonání běžných instrukcí, pokud odpovídají chování cílového procesoru.

Při čtení instrukce z paměti tato metoda používá další metodu této třídy FinalizeFetch, tato metoda čte případné parametry instrukce a vrací cíl instrukce. Metoda FinalizeFetch ve své implementaci volá metodu FinalizeFetchHandler, toto je opět metoda, kterou je nutné implementovat dle specifikace procesoru a jeho instrukční sady.

6.1.2 Metoda FinalizeFetchHandler

Metoda je použita při čtení instrukce z paměti. Metoda přijímá jako parametr adresu, na které začíná parametr právě vykonávané instrukce a adresový mód. Adresová mód slouží k určení způsobu uložení parametru instrukce. U některých procesorů může být adresový mód stejný pro všechny instrukce, jiné procesory můžou obsahovat množství adresových módů pro každou instrukci, a proto je nutné, aby metoda věděla o způsobu adresování. První parametr (adresa parametru instrukce) je vstupně výstupní parametr, ve kterém je po ukončení instrukce vrácena cílová adresa instrukce, adresa se kterou instrukce pracuje. Návratová hodnota instrukce udává délku instrukce, tedy o kolik se bude inkrementovat PC registr po vykonání instrukce.

6.1.3 Další součásti implementace

Dále je nutné implementovat metody SetStatus a GetStatus. Tyto metody slouží k nastavení a získání stavu procesoru. Metody přijímají resp. vrací (metoda GetStatus vrací jako návratovou hodnotu, metoda SetStatus přijímá jako parametr) třídu Dictionary<string, long>, kde typ string zastupuje jméno registru a typ long jeho hodnotu. Vzhledem k různorodosti procesorů nejsou na předávání stavu procesoru kladeny žádné další nároky. Je na autorovy modulu, aby Dictionary vyplnil daty z procesorových registrů.

6.2 Implementace paměťového modulu

Vzhledem k povaze paměťového modulu není nutné, oproti modulu procesoru, doplňovat velké množství kódu. Pokud je paměťový model architektury velmi jednoduchý je možné přímo použít třídu MemoryBase bez nutnosti vytvářet vlastní. Pro reprezentaci složitější paměti je vhodné využít modelu MultiLayerMemoryBase, který je schopen reprezentovat složitější vícevrstvou paměť, kdy na jednu adresu může být mapováno více fyzických pamětí. V tomto případě podobně jako procesor, musí být modul reprezentován třídou dědící abstraktní paměťovou třídu MultiLayerMemoryBase.

- Paměť typu MemoryBase

Pokud vlastnosti MemoryBase dostačují pro reprezentaci paměti cílové architektury, je vhodné tuto třídu použít. Při jejím vytváření je pouze nutné specifikovat počáteční adresu a její velikost.

- Paměť typu MultiLayerMemoryBase

Při použití toho paměťového modulu je třeba vytvořit třídu dědící MultiLayerMemoryBase. V konstruktoru této třídy se pak vytvářejí jednotlivé paměťové bloky, ze kterých je paměť složena. Každý paměťový blok je instance třídy MemoryUnitBase. Pro každý blok je definována jeho počáteční adresa a velikost. Mapování bloku zajišťuje metoda BlockMapper, tu je nutné implementovat.

- Metoda BlockMapper

Tato metoda zajišťuje mapování bloků do paměti. Způsob mapování se liší podle implementované architektury proto je metoda implementována autorem modulu. Jako parametr je jí předána adresa, na kterou je přistupováno a metoda vrací blok (instanci MemoryUnitBase), který by měl být mapován na danou adresu. Vyzvednutí nebo zapsání dat už zajišťuje implementace základní třídy.

.

6.3 Implementace textového grafického výstupu

Pro reprezentaci jednoduchého grafického výstupu může být dostačující použití třídy TextGraphicOutput. Ovšem pro vytvoření složitějších výstupů je nutné vytvořit novou třídu dědící TextGraphicOutput a doimplementovat specifické chovaní daného čipu. Základní třída poskytuje několik atributů pro definování parametrů displeje, jako jsou rozlišení ve znacích, rozlišení jednoho znaku, dále je zde definována poloha obrazové a znakové paměti.

Obrazová paměť (Screen Memory) určuje kód znaku, který má být vykreslen na pozici odpovídající adrese v paměti. Znaková paměť (Charakter Memroy) určuje podobu znaku. Pokud není definována znaková paměť jsou použity ASCII znaky a základní font definovaný ve třídě. Pro vykreslování výstupu slouží dvě metody. Metoda Draw vykreslí celou obrazovou plochu. Metoda DynamicScreenUpdate vykreslí pouze znak odpovídající adrese v obrazové paměti.

Třída dále obsahuje několik pomocných metod, které usnadní implementaci odvozené třídy. Tyto třídy přímo neimplementují nastavování jednotlivých bodů, ale používají k tomu metody definované ve statické třídě QEmulatorDrawing.

- Metoda GetCharPosFormAddress

Metoda je používána k nalezení pozice znaku odpovídajícího adrese. Metoda vychází z počáteční adresy obrazové paměti a počtu znaků na řádku a v sloupci. Předpokládá se, že obraz je v paměti uložen po řádcích, tedy že adresa o jedna větší reprezentuje znak vlevo.

- Metoda DrawCharacter

Metoda vykreslí jeden znak na výstup. Metoda používá implementaci v QEmulatorDrawing.

- Metoda GetCharacter

Metoda získá definici znaku na základě jeho kódu. Kód znaku násobený jeho velikostí je použit jako index do znakové paměti.

- Metoda DrawGenericCharacter

Vykreslí ASCII znak na danou pozici. Požije aktuální barvu znaku a pozadí.

- Metoda GetColor

Metoda vrací barvu podle kódu barvy. Tato metoda by měla být implementována podle specifikace grafického čipu. V základní implementaci vrací černou barvu pro hodnotu nula, jinak vrací bílou barvu.

6.4 Implementace jádra emulátoru

Každé jádro emulátoru musí byt implementováno třídou dědící abstraktní třídu EmulatorShellBase. Tato nová třída reprezentuje celou emulovanou architekturu a komunikuje s uživatelským rozhraním. Implementace musí vytvořit instance všech modulů architektury, provést jejich inicializaci a zajistit jejich vzájemné propojení. Většina těchto operací by měla být provedena v konstruktoru nebo v inicializačních metodách. Propojování jednotlivých modulů je založeno na přiřazování odpovídajících metod na vyvolání událostí.

Přestože je většina metod EmulatorShellBase deklarována jako protected virtual a je možné přetížit jejich implementaci, nová třída by se měla omezit na vytvoření jednotlivých modulů a jejich propojení v inicializačních metodách. Jiné úpravy této třídy by měly být nutné jen v extrémních případech.

Inicializace modulů je prováděna ve třech metodách definovaných v EmulatorShellBase, tyto metody jsou volány z konstruktoru a proto by nová implementace třídy měla zajistit zavolání konstruktoru bázové třídy. Při implementaci vlastního konstruktoru a vlastního vytváření modulů je nutné dodržet toto pořadí volání funkcí.

- Metoda CreateModules

V metodě by měly být vytvořeny instance jednotlivých logických celků emulátoru, jako je paměť, procesor atd.

- Metoda ConnectModules

V této metodě by měly být moduly vzájemně propojeny spojením jejich událostí s metodami. V bázové třídě již je obsažena implementace této metody ustavující základní komunikaci jednotlivých modulů. Je propojen procesor s pamětí, požadavek na uložení a vyzvednutí dat. Dále je vytvořeno spojení paměti a grafického výstupu, jejich datová komunikace a komunikace pro vytváření Memory triggerů.

- Metoda InitializeModules

Metoda provede inicializaci jednotlivých bloků. Nad každým modulem je zavolána metoda Initialize, Tuto metodu má definovanou každý logický blok a metoda zajistí inicializace modulů vůči sobě. Dojde k předání MemoryTriggerů a jiných nastavení.

- Metoda DataSetups

Jako čtvrtá je k dispozici tato metoda, kterou ale není nutné implementovat. Slouží k dalšímu nastavení jednotlivých bloků, po jejich vzájemné inicializaci. Při volání této metody už jsou jednotlivé moduly plně funkční, inicializované a jejich vzájemná komunikace již nemá žádná omezení. Je možné zde uložit data do emulátoru nebo provést inicializaci celého emulátoru jako celku.

7 Assembler

Implementace assembleru a disassembleru je součástí dynamické knihovny dll, která obsahuje modul emulované architektury ale není přímo součástí tohoto modulu. Proto implementace těchto součástí není nezbytně nutná při vývoji nového modulu. Nicméně debugování aplikací se v tomto případě stává velmi obtížné, protože není možné zobrazit vykonávaný kód v podobě, která je dobře čitelná.

I zde je snaha vytvořit jistou obecnou abstrakci, který bude základem pro assemblery různých architektur. Tato abstrakce by ale měla umožňovat aby si jednotlivé assemblery mohli udržet svůj specifický formát zápisu, který se může lišit. Touto abstrakcí je třída AssemblerBase. Ta obsahuje popis jednotlivých instrukcí a jejich parametry, a má informace o formátu jejich zápisu. Rovněž obsahuje metody pro asemblování a disasemblování.

Každá Instrukce je definovaná pomocí třídy InstructionTemplate. Zde je operační kód instrukce, její jméno a další atributy nutné k překladu do/z binární podoby instrukce. Aby různé implementace assemblerů mohly mít různý formát je přesná podoba instrukce definována atributem Format, ten obsahuje formátovací řetězec. Podle tohoto atributu dochází k překladu instrukce. Formátovací řetězec je složen z fragmentů, reprezentujících jednotlivé části instrukce. Fragment má většinou délku jednoho znaku, ale není to podmínkou. Jednotlivé fragmenty jsou odděleny znakem „|“. Formátovací řetězec má dva pevně definované fragmenty, těmi jsou parametry instrukce (většinou představují adresu, se kterou instrukce pracuje). Těmito fragmenty jsou písmena B a L doplněné o číslici, reprezentující délku parametru instrukce v bytech. Fragment B předpokládá uložení parametru ve formátu big endian, L jako little endian. Použitím formátovacího řetězce pro definici formátu instrukce je možné zobecnit převod instrukcí z a do binární podoby a mít velkou část implementace společnou pro různé moduly asemblerů.

8 Commodore 64

Technické specifikace C64 v této kapitole jsou převzaty z [4].

Commodore 64 je jedním z nejúspěšnějších osobních počítačů všech dob, o tom svědčí i obrovské množství aplikací a her, které pro něj byly vyvinuty. Z tohoto důvodu bude právě Commodore 64 emulovanou architekturou. Jedná se o 8 bitový osobní počítač představený v lednu roku 1982. Velkou výhodou C64 oproti konkurenci v tehdejší době byla možnost připojení k běžným domácím televizním obrazovkám bez jakýchkoli úprav a velmi kvalitní grafika.

8.1 Konkurenční architektury

ZX Spektrum

Jedná se o 8 bitový osobní počítač, hlavní konkurent C64. Byl používat hlavně ve Velké Británii, oproti tomu C64 byl používán hlavně v USA. Jednou z nevýhod ZX bylo horší řešení grafiky což se projevovalo hlavně ve hrách a jiných graficky náročných aplikacích. Zde měl C64 výhodu hlavně kvůli a multi-color módu a používání Sprite objektů (popsáno dále). Disponoval 16 nebo 48KB RAM paměti, CPU běželo na 3,5MHz.

BBC Micro

8 bitový osobní počítač. Používán hlavně ve Velké Británii pro školní účely. Byl odolnější než například C64 nebo ZX právě kvůli jeho plánovanému nasazení ve školách. První modely měly 16-24 KB RAM paměti, procesor běžel na frekvenci 4MHz.

Amstrad CPC

Zamýšlen jako přímí konkurent C64. Byl dodáván s vlastním monitorem. Disponoval 64 nebo 128 KB RAM paměti, frekvence CPU byla 4MHz.

8.2 Architektura C64

Z dnešního pohledu už je C64 nevyhovující pro jakékoli běžné použití. Nicméně v 80. letech byly možnosti C64 velice zajímavé.

C64 obsahuje vestavěný interpret jazyka BASIC, ten je hlavním uživatelským komunikačním prostředkem a umožňuje jednodušší vývoj aplikací než strojový jazyk a dělá zdrojový kód aplikací čitelnější. Adresový prostor má velikost 64kB, ne celý je ovšem k dispozici pro uživatele vždy.

Obrazovka C64 je ve výchozím stavu textový mód o 80 znacích na jednom řádku, tímto jsou omezeny veškeré textové řetězce tak i příkazy jazyka BASIC a počtem řádků je omezena i maximální délka podprogramů.

Zajímavou oblastí je grafika. C64 nedisponuje žádným grafickým módem, umožňujícím pracovat přímo s jednotlivými body obrazovky, který je běžný u dnešních počítačů, ale má několik znakových módů s možností definovat vlastní znaky, čímž je tento nedostatek do jisté míry odstraněn. Nezvyklým grafickým prvkem jsou tzv. SPRITEs. Těmto grafickým objektům je možné definovat libovolný vzhled a mají schopnost se volně pohybovat po obrazovce, což je velice vhodné zejména při vývoji her.

Commodore 64 disponuje i zvukovým čipem disponuje 6581 "SID", což byl jeden z nejlepších zvukových čipů té doby.

8.3 Soubor instrukcí C64

C64 je vybaven procesorem 6510 firmy MOS Technology. Jde o 8 bitový procesor schopný adresovat 64kB paměti. Instrukční sada obsahuje 54 instrukcí. Je velmi podobná sadám dnešních procesorů, v tomto směru nenajdeme mnoho rozdílů. Instrukce se dají rozdělit do několika skupin stejně jako u moderních procesorů.

Aritmetické instrukce – SBC(odčítání), ADC(sčítání), …

Logické instrukce – EOR(exklusive OR), ROL(bitová rotace), …

Instrukce skoků – BEQ(skok při shodě), ….

Operace s pamětí – MOV(přesun), ….

Podrobněji jsou instrukce procesoru 6510 popsány v přiloženém souboru

„C64 instruction set.txt“.

8.4 Paměťový prostor C64

Paměť je v C64 adresována 2 byty. To znamená, že adresovatelný prostor má 65536 (adres 2^{16}). Celý adresový prostor ale není dostupný pro uživatele.

Přehled adresového prostoru C64

Adresa 0 a 1	Na těchto adresách jsou mapovány registry procesoru 6510
Adresy 2 až 1023	Paměť používaná operačním systémem
Adresy 1024 až 2039	Paměť je mapována na výstup obrazovky. Každý byte odpovídá jednomu znaku na displeji 40x25, to je 1000 znaků/bytů
Adresy 2040 až 2047	Ukazatele na paměť s popisy SPRITE objektů
Adresy 2048 až 40959	Uživatelský paměťový prostor. Zde jsou ukládány programy v jazyce BASIC nebo strojovém jazyce
Adresy 40960 až 49151 8kB	užívaných interpretem jazyka BASIC
Adresy 53248 až 53294	Adresy používány grafickým čipem VIC-II
Adresy 54272 až 55295	SID registry (Sound Interface Device)
Adresy 55296 až 56296	RAM barev
Adresy 56320 až 57343	I/O registry

Adresy 57344 až 65535 8kB pro operační systém Kernal. Obsahuje rutiny pro obsluhu uživatelských vstupů, výstupu na obrazovku a podobně.

8.4.1 Adresní módy a indexování

Protože adresní prostor má 2^{16} adres, adresy jsou 16 bitové. Každá adresa je uložena na 2 bytech a při adresování paměti je adresa zadávána v pořadí vyšší byte, nižší byte (např. LDA $00FF). Horní byte je proto možné považovat za „stránku“ o velikosti 256 bytů.

- Indexování

Pro indexování jsou k dispozici 2 registry X a Y. K použití indexoví stačí přidat X nebo Y registr jako další parametr instrukce.

- Zero page (nultá stránka)

V tomto módu C64 automaticky předpokládá, že je přistupováno k a datům na nulté stránce a proto je se zadává pouze nižší byte adresy (např. LDA $FF – poslední byte nulté stránky).

- Nepřímé indexování

Pro nepřímé indexování je možné použít pouze Y registr. Při použití nepřímého adresování musí být skutečná adresa adresovaná nepřímo v nulté stránce. Adresa musí být uložena tak, že v místě adresovaném z instrukce je nižší polovina adresy a na adrese o jedna vyšší je vyšší polovina adresy.

Předpokládejme: $05 = #$20 $06 = #$2E Y = $01

Pak při zavolání instrukce LDA(05) bude skutečná adresa 2E20 + 1

- Indexování nepřímo

Toto adresování používá pouze X registr. Je velmi podobné nepřímému adresování, ale hodnota registru X je přičtena k oběma částem nepřímé adresy.

- Zásobník

Zásobník C64 je na první stránce paměti, tj. na adresách $0100 až $01FF. Začíná na nejvyšší adrese a SP (ukazatel na vrchol) je dekrementován s každým vložením nové informace. SP ukazuje vždy na první volnou pozici.

8.4.2 I/O port a data direction register procesoru 6510

Na adrese 1 se nachází I/O port procesoru 6510. Tento port řídí mapování ROM do adresového prostoru

0. bit	LORAM	výstup	řídí mapování pro adresy $A000 až $BFFF
1. bit	HIRAM	výstup	řídí mapování pro adresy $E000 až $FFFF
2. bit	CHAREN	výstup	řídí mapování pro adresy $D000 až $DFFF
3. bit		výstup	zápis na pásku
4. bit		vstup	chování pásky
5. bit		výstup	řízení motoru pásky

Na adrese 0 se nachází data direction registr pro tento port, ten udává směr toku dat.

Správné hodnoty pro bity:	0	1	2	3	4	5
	1	1	1	1	0	1

Kde 1 značí výstup a 0 vstup. C64 po startu automaticky nastaví tyto hodnoty.

LORAM odstraňuje z adresovatelného prostoru 8kB používaných interpretem jazyka BASIC a nahrazuje je pamětí RAM. Proto je nevhodné tento paměťový segment odstraňovat, jestliže jej uživatel používá. Standardně je bit nastaven na 1 a na adresách $A000 až $BFFFF je ROM s interpretem jazyka BASIC.

HIRAM odstraňuje z paměti 8kB vyhrazených pro operační systém Kernal. Standardně je Kernal v adresovém prostoru a tento bit je nastaven na 1. Při nastavení na 0 je používána paměť RAM místo ROM s rutinami Kernalu.

CHAREN mapuje do paměti 4kB ROM paměti, používané generátorem znaků. Standardně je tato hodnota na 1 a ROM znaků je nedostupná. Do paměti jsou mapovány vstupně výstupní zařízení.

Možnosti mapování paměti:

E000-FFFF	RAM nebo 8kB KERNAL ROM
D000-DFFF	4kB I/O nebo RAM nebo CHAR. ROM
C000-CFFF	4K RAM
A000-BFFF	8kB BASIC ROM nebo RAM nebo ROM PLUG-IN
8000-9FFF	ROM PLUG-IN nebo 8kB RAM
4000-7FFF	16kB RAM
0000-3FFF	16kB RAM

8.5 Grafika na Commodore 64

Commodore 64 má několik grafických módů. Díky nimž je možné dosáhnout poměrně velké flexibility při práci s grafikou i textem. Pro zobrazení grafických prvků je rezervováno 1000B paměti, C64 má 1000 pozic na obrazovce na kterých je možné zobrazit nějaký znak. V každém z grafických módů se s danou pamětí pracuje jinak. Je zde i grafický mód, ve kterém je snaha přistupovat k obrazovce jako k jednotlivým bodům, jak je tomu běžné dnes, ale práce v tomto módu je přece jen složitější, než je tomu dnes kvůli uspořádání dat.

8.5.1 Znakový mód

Tento mód používá pro zobrazování 2 oddělené paměťové bloky. Jeden pro definici znaků na jednotlivých pozicích obrazovky a druhý definuje barvy těchto bloků. Může být použit ještě třetí blok popisující sadu znaků.

1.Color Memory – Pozice toho bloku je neměnná. Leží na adresách $D800 až $DBE7.

2. Screen Memory – Pozici toho bloku je možné měnit na adrese $D018. Horní 4 bity označují polohu tohoto bloku, dolní adresují sadu znaků. Jsou zde uloženy znaky, které budou uloženy na jednotlivých pozicích displeje.

3. Character Memory – Tento blok definuje sadu znaků. Z této paměti získává VIC-II (Video Interface Chip) informace o vzhledu každého znaku.

8.5.1.1 Standardní znakový mód

V tomto módu jsou použity všechny tři paměťové bloky. Screen Memory obsahuje znaky pro každé pole na displeji, Color Memory obsahuje barvy pro tyto bloky. Znak je definován jako matice bitů 8x8, každé pole tedy obsahuje 64 bitů. Jak by mely být tyto bity nastaveny je uloženo v Character Memory. Jestliže je bit nastaven na 1, použije se k jeho vybarvení barva definovaná v Color Memory, jestliže je nastaven na 0 použije se barva pozadí. Nevýhoda tohoto módu je v tom, že každá pozice může mít pouze 2 barvy, např. ve hrách může být tato skutečnost vážný problém, protože např. neumožní ani protnutí 2 různobarevných čar na pozadí jiné bary.

8.5.1.2 Multi-Color mód

Tento mód řeší problém 2 barev na pozici jednoho znaku. Umožňuje každému znaku mít až 4 barvy, ale za cenu nižšího horizontálního rozlišení. Multi-Color mód je možné nastavit pro každý znak jednotlivě. Nastavením 3. bity na odpovídajícím místě v Color Memory je daný znak vykreslen v tomto módu. Barva pro daný bod znaku je určována bity, ty nejsou brány jednotlivě, ale tvoří páry. Na těchto 2 bitech jsou možné 4 kombinace, což koresponduje se 4 možnými barvami pro znak. Bod opět může nabývat barvy znaku a barvy pozadí, ale ještě dalších dvou barev, které jsou definovány v registrech VIC-II čipu na adresách $D022 a $D023.

8.5.2 Bitmapový mód

V bitmapovém módu je možno kontrolovat přímo jednotlivé body. Body ale nejsou v paměti přímo za sebou, ale podobně jako u znakových módů jsou seskupeny. Opět dostáváme matice 8x8 a k tomu je nutné přihlížet při přístupu k jednotlivým bitům. I zde je možné si vybrat jestli potřebujeme upřednostnit vyšší rozlišení nebo více barev.

8.5.2.1 Standardní mód (vysoké rozlišení)

Zde je opět možnost dát bodům jednu ze 2 barev pro matici 8x8. V tomto módu můžeme pracovat s rozlišením 320x200.

8.5.2.2 Multi-Color mód

Podobně jako multi-Color mód se znaky umožňuje použít 4 různé barvy v oblasti matice 8x8. Opět je sníženo rozlišení aby bylo možno rozšířit informaci o barvě každého bodu. Podobně jako u znaků je barva řízena kombinací párů bodů. Dodatečné barvy jsou opět definovány v registrech VIC-II čipu.

8.5.3 Skrolování

C64 umožňuje plynulé skrolování obrazu. C64 je schopný posunout celý obraz o jeden bod na libovolnou stranu. VIC-II obsahuje tzv. scroll register, který se stará o posunutí na danou stranu. Je ale nutné aby bylo v programu zajištěno naplnění okrajových oblastí, které se odkrývají.

8.5.4 Sprites

Sprite jsou grafické objekty nezávislé na zbytku scény. Tyto objekty jsou spravovány přímo VIC-II čipem. Těchto objektů může čip spravovat až 8 najednou. Sprite lze považovat za jistý druh uživatelsky definovaného znaku. Každý sprite má svůj vzhled (podobně jako definovatelný znak), může mít grafický mód nezávislý na zbytku displeje a dalších spritech, vlastní barvy.

Sprite jsou definovány maticí 24x21 bodů a je možné je zvětšovat vertikálně horizontálně nebo obojí na dvojnásobnou velikost. Pro každý sprite je možné zajistit detekci kolizí s jinými sprity a s pozadím.

Na každý sprite je definovaný na 64 bytech. Na každý sprite se VIC-II čip odkazuje pomocí registrů Sprite Pointer #0 až #7.

9 Závěr

Při návrhu emulátoru byl zvolen interpretovaný způsob emulace. Tato volba se ukázala jako poměrně vhodná, emulátor nemá žádné potíže se samo-modifikující se kódem a rovněž blíže reprezentuje skutečnému návrhu emulované architektury. I celkový návrh emulátoru se ukázal být poměrně dobrý. Bylo dosaženo požadované nezávislosti jednotlivých bloků, kdy změna v jednom z bloků nevyžaduje žádné úpravy v ostatních blocích. I v případě podstatnějších změn během procesu vývoje emulátoru návrh obstál a úpravy neznamenaly nutnost přepracovávat větší část implementace. Proto by mělo být možné upravit funkčnost emulátoru, při nutnosti dosáhnout specifické funkčnosti, bez větších problémů. I abstrakce jednotlivých bloků jsou navrženy poměrně dobře a bylo nutné implementovat pouze chováním specifické pro architekturu C64.

Větší problém znamenala samotná implementace C64 z hlediska jeho funkčnosti. Commodore 64 má velmi širokou paletu příslušenství, v popisu architektury C64 byly míněny jen nejdůležitější součásti této architektury. Současná implementace modulu je poměrně daleko od plné funkčnosti. Velmi dobře a podrobně je implementována část procesoru a paměti, včetně přerušení, všech adresových módů. Funkční je i grafický výstup ten ovšem nemá plně implementované všechny zobrazovací módy. Problémem v této části bylo často nalezení přesné dokumentace pro periferie a popis rutin operačního systému, které s nimi komunikují.

Při navazování na tento projekt je možné pokračovat ve vývoji dalších modulů. Rovněž je zde možnost dokončit implementaci periferií pro primárně implementovanou architekturu C64. Po stránce generičnosti emulátoru by bylo vhodné rozšířit fond statických metod pro vykonávání běžných instrukcí nebo vytvořit specializované metody pro efektivní práci s daty na úrovni registrů a bitových operací. Dále je možné se zaměřit na vývoj několika více specializovaných abstrakcí, které by dále usnadňovaly implementaci nových modulů.

Literatura

[1] Fayzullin, M.: Computer Emulation Resources. [online]. Dostupné na URL: <http://fms.komkon.org/EMUL8/>

[2] Lohman B., Van der Hoeven J., Rose C., Kiers**.:** Projekt Dioscuri. [online]. Dostupné na URL: <http://dioscuri.sourceforge.net/>

[3] National library of netherlands [online]. Emulation. Dostupné z WWW: <http://www.kb.nl/hrd/dd/dd_projecten/projecten_emulatie-en.html>.

[4] Commodore 64 Programmer's Reference Guide [online]. Dostupné z WWW: <http://www.devili.iki.fi/Computers/Commodore/C64/Programmers_Reference/>.

Seznam příloh

Příloha 1. CD

- Zdrojové texty aplikace jako solution vývojového prostředí Microsoft Visual Studio
- Instalační soubor aplikace
- Soubor „C64_instruction_set.txt“ obsahující popis instrukční sady C64
- Zpráva ve formátu .pdf a .doc