VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

EMULACE CPU PRO VÝUKU ASEMBLERŮ

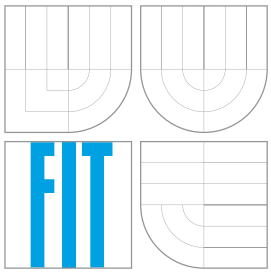
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

LUKÁŠ CHARVÁT

BRNO 2009

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

EMULACE CPU PRO VÝUKU ASEMBLERŮ

EMULATION OF CPU ARCHITECTURES FOR ASSEMBLY LANGUAGES COURSE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — LUKÁŠ CHARVÁT
AUTHOR

VEDOUCÍ PRÁCE — Ing. ALEŠ SMRČKA
SUPERVISOR

BRNO 2009

Abstrakt

Práce řeší tvobu emulátoru počítačové architektury se záměrem pro použití při vyúce asemblerů. Zatímco většina dnešních emulátorů je zavislá na specifické architektuře, tato práce popisuje přístup, jak vytvořit emulátor, který by umožnoval uživatelům jednoduše vytvářet vlastní architektury, provádět nad nimi operace a zobrazovat jejich aktuální stav.

Abstract

The bachlors's thesis discusses the design of a CPU architecture emulator aimed to assembly languages course. While most of nowadays emulators are architecture specific, this document describes an approach to create an emulator allowing users to easily set up their own architecture, to perform operations upon it, and to display its current state.

Klíčová slova

Emulátor, cpu, asembler, jazyk, gramatika, krokování kódu.

Keywords

Emulator, cpu, assembly, language, grammar, debugging.

Citace

Lukáš Charvát: Emulace CPU pro výuku asemblerů, bakalářská práce, Brno, FIT VUT v Brně, 2009

Emulace CPU pro výuku asemblerů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Aleše Smrčky.

.....
Lukáš Charvát
19. května 2009

Poděkování

Tímto bych chtěl poděkovat panu Ing. Aleši Smrčkovi za jeho podporu i užitečné a cenné rady při tvorbě práce.

Obsah

Kapitola 1

Úvod

Emulace slouží k imitaci funkčnosti jiného systému, ať už modifikací hardwaru nebo softwaru, který umožní imitujícímu systému přijímat stejná data, spouštět stejné programy a dosahovat stejných výsledků jako imitovaný systém [7]. Emulace se liší od simulace tím, že oproti simulaci si neklade za cíl získávání nových znalostí o systému, ale pouze daný systém imituje. Tyto vlastnosti nachází své uplatnění hlavně v boji proti stárnutí hardware, které je způsobeno nástupem nových technologií. Emulace umožňuje vytvořit na současné platformě původní prostředí umožňující běh aplikace, jež byla vytvořena pro architekturu již zastaralou. I když náklady na vývoj emulátoru mohou být vysoké a je nutná dokonalá znalost emulovaného systému, v jistých případech by mohly být několikrát převýšeny prostředky nutnými k migraci aplikace samotné na architekturu současnou. Jiným oborem využívajícím emulaci je vývoj software pro zabudované systémy. Emulace v tomto oboru umožňuje spouštět aplikace pro zatím ještě fyzicky neexistující hardware, a tím dovoluje jejich současný vývoj.

Další možností pro uplatnění emulace může být i výuka. Cílem této bakalářské práce je vytvořit emulátor umožňující začátečníkům snadný vstup do problematiky programování ve strojovém kódu různých platforem. Důraz by měl být kladen na jednoduchost použití. Hlavním zaměřením práce není nasazení v průmyslu, ale v experimentálním prostředí, jež je pro akademickou půdu typické.

V první kapitole bakalářské práce je rozebrán aktuální stav na poli emulátorů pomocí rozborů vlastností některých z nich. Druhá kapitola je souhrnem požadavků na nově vyvíjenou aplikaci. Třetí kapitola se věnuje návrhu této aplikace. Ve čtvrté kapitole je popsána implementace návrhu popsaného v kapitole třetí. Jsou zde rozebrány některé klíčové detaily implementace. V kapitole páté jsou provedeny testy výsledné aplikace. Šestá kapitola je závěrem této práce. Shrnuje dosažené výsledky a provádí nástin možného budoucího rozvoje aplikace.

Kapitola 2

Přehled architektur a emulátorů, specifikace požadavků

Na následujících řádcích jsou nejdříve shrnuty vlastnosti současných počítačových architektur. Dále jsou diskutovány aplikace související s emulací procesorů a na základě jejich analýzy jsou specifikovány požadavky na nově vyvíjený emulátor.

2.1 Současné architektury

Tento oddíl pojednává o vybraných současných počítačových architekturách. Popis architektur se především věnuje různým specifickým vlastnostem architektur.

2.1.1 Architektura Intel x86

Architetura Intel [5] stála na počátku počítačové revuluce a podle posledních měření použití výpočetní síly ve světě je dnes nejpreferovanější počítačovou architekturou. Současná instrukční sada architektury, která se vyznačuje proměnnou délkou instrukce, je v podstatě souborem rozšíření aplikovaných na jednoduchou osmibitovou architekturu 8080/8085, která stála u zrodu architektury. Dnešní procesory architektury Intel jsou tak díky své zpětné kompatibilitě stále schopny vykonávat kód napsaný pro tuto dnes již zastaralou architekturu. Jádrem architektury je osm registrů pro všeobecné použití, šest segmentových registrů, registr s příznaky a čítač instrukcí. Zmíněné registry vytvářejí základní výpočetní prostředí, ve kterém lze pracovat se souborem instrukcí pro všeobecné použití. Tyto instrukce provádějí základní matematické operace s celými čísly na osmi, šestnácti a třiceti dvou bitech, řídí tok programu a adresují paměť. Slova jsou u architektury ukládána do paměti ve formě little-endian [9].

2.1.2 Java Virtual Machine

Java Virtual Machine (JVM) [6] je modul virtuálního stroje určený ke spouštění počítačových programů a skriptů vytvořených převážně v jazyce Java. Úkolem tohoto modulu je zpracovat tzv. mezikód, v Javě označovaný jako Java bytecode. Programy jednou zkompilované do tohoto mezikódu již není nutné upravovat pro konkrétní počítačové architektury, o přenositelnost programů se zde stará implementace JVM na konkrétní platformě [8].

2.2 Současná emulace procesorů

Tento oddíl postupně rozebírá vlastnosti některých zástupců ze současného velkého množství emulátorů. U každého diskutovaného emulátoru je provedena jeho analýza, která je následně shrnuta v několika bodech.

2.2.1 PearPC

PearPC [3] je emulátor počítače typu PowerPC umožňující běh aplikací i (operačních systémů) pro tuto platformu.

- Zaměření: Volně dostupný a dobře přenositelný emulátor architektury PowerPC.
- Emulované prvky: PowerPC CPU, PCI-Bridge, diskový řadič, síťová karta, NVRAM, klávesnice a myš.
- Modifikovatelnost architektury: Emulovaná architektura je modifikovatelná pomocí nastavení v konfiguračních souborech.
- Výhody: Rychlost, snadná přenositelnost mezi operačními systémy (aplikace je napsaná v programovacím jazyce C++).
- Nevýhody: K pochopení možností nastavení v konfiguračních souborech je nutné delší studium dokumentace.

2.2.2 ZX32

Emulátor ZX32 [2] je zástupcem z široké rodiny podobných nástrojů pro emulaci počítače Sinclair ZX Spektrum. Pro tento typ počítače bylo napsáno velké množství programů (především her), cílem emulátoru je tedy snaha o jejich zachování.

- Zaměření: Volně dostupný emulátor počítače Sinclair ZX Spektrum
- Modifikovatelnost architektury: Není možná.
- Výhody: Schopnost emulovat hned několik verzí tohoto počítače.
- Nevýhody: Výčet emulovaných verzí nelze nijak rozšiřovat, aplikace je dostupná pouze pro operační systém Windows.

2.2.3 MAME – Multiple Arcade Machine Emulator

Hlavním cílem projektu MAME [4] je vytvářet imitaci vnitřního chování emulované herní konzole. Projekt je určen jak pro vzdělávací účely, tak pro účely zachovaní starých her. Mnohé z nich jsou odsouzeny k zániku, jakmile hardware, na kterém běží, přestane pracovat.

- Zaměření: Volně dostupný emulátor zastaralých herních konzolí.
- Modifikovatelnost architektury: Implementováno rozhraní pro přidávání nových architektur herních konzolí.
- Výhody: Rychlost a přenositelnost (emulátor je napsán v jazyce C), možnost přidávat vlastní architektury.
- Nevýhody: Pokud se uživatel chce podílet na vytváření nových prvků, je nutná hlubší studie emulátoru a rozhraní, na jehož základě se nové komponenty vytvářejí.

2.3 Specifikace požadavků

Analýzou zástupců současných emulátorů bylo zjištěno, že většina z nich postrádá možnost měnit emulovanou architekturu nebo její instrukční sadu. Práce s nimi bývá také přísliš složitá na to, aby je šlo použít jako výukový prostředek. Z toho vyplývají základní požadavky na navrhovanou práci (dané především jejím zaměřením):

- jednoduchost – emulátor by měl být lehce pochopitelný bez nutnosti číst příručku nebo návod
- schopnost emulovat co největší množství architektur – schopnost emulátoru emulovat současné architektury a adaptovat se na platformy nově příchozí
- přívětivost pro začátečníky – emulátor by měl být schopen používat i uživatel bez hlubokých znalostí problematiky programovaní v asemblerech

Ze základních požadavků můžeme vytvořit požadavky na další vlastnosti výsledného programu. Lze je rozdělit do dvou kategorií. První kategorií jsou požadavky na výkonné jádro emulátoru, druhou kategorií jsou pak nároky na grafické rozhraní.

2.3.1 Požadavky na výkonné jádro

- schopnost dynamicky vytvářet typické prvky a struktury počítačových architektur – nutné pro reprezentaci emulované architektury a jejího stavu (definovaného stavem prvků architektury)
- schopnost dynamicky rušit prvky architektury – dynamičnost vytváření a rušení je vyžadována některými architekturami (zejména virtuálními)
- možnost modifikovat stav prvků architektury i částí těchto prvků
  - přesuny hodnot
  - matematické operace sčítání, odčítání, násobení, dělení
  - logické operace and, or, xor, logické posuvy vlevo a vpravo
- možnost spojovat sled operací do funkcí – často používané sledy operací by měly být uspořadatelné do celků, aby je nebylo nutné pokaždé vypisovat zvlášť
- prvky vznikající v emulovaném systému nesmí být ovlivněny architekturami stávajícími (např. bitovou šířkou) – musí být umožněna emulace např. 256bitové architektury
- schopnost seskupování prvků pro zobrazení
- schopnost definovat logické vazby mezi prvky – z výukových účelů, studentovi mohou být zobrazeny vztahy mezi prvky architektury, jako např. vrchol zásobníku
- možnost zařazení vlastního modulu pro interpretaci instrukcí – možnost změny instrukční sady emulované architektury

2.3.2 Požadavky na grafické rozhraní

- možnost psaní kódu v asembleru emulované architektury
- možnost krokování kódu (angl. debugging)
- možnost tvorby bodů přerušení (angl. breakpoints)
- zobrazení aktuálního stavu emulované architektury
- přenositelnost mezi různými druhy operačních systémů
- schopnost zobrazovat vytvořené skupiny prvků
- schopnost zobrazovat logické vazby mezi prvky

Kapitola 3

Návrh řešení

S ohledem na požadavky kladené na systém můžeme návrh projektu rozdělit na pět částí. V první z nich jsou rozebrány společné vlastnosti architektur a na jejich základě je proveden návrh tříd, jež budou reprezentovat emulovanou architekturu a umožňovat provádět nad ní operace. Následuje návrh tříd reprezentujících jednotlivé funkční celky emulátoru. V kapitole věnující se návrhu jazyka pro popis architektury je pak vystavěna gramatika jazyka. Dále je navržen vzhled a schopnosti grafického uživatelského rozhraní. Poslední část návrhu se zaměřuje na celkový princip činnosti emulátoru.

3.1 Návrh tříd pro reprezentaci architektury

Pro dosažení schopnosti emulovat co nejvíce počítačových architektur musí být emulátor schopen reprezentovat jejich základní prvky. Z pohledu identifikace daných prvků je možné rozdělit je na tři druhy:

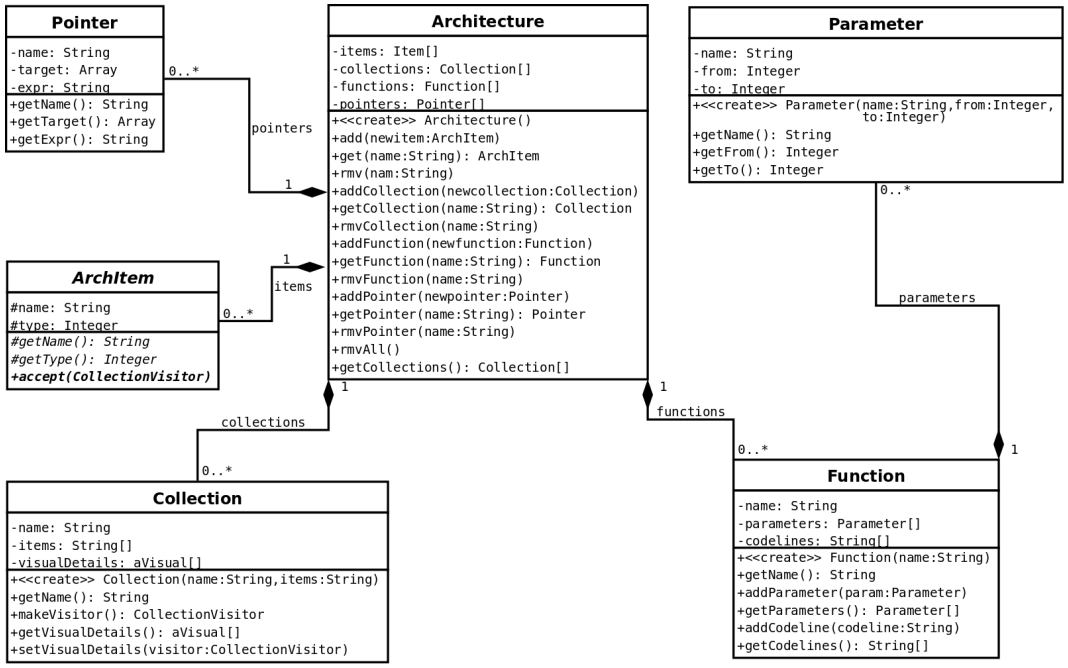
- jedinečné, v architektuře jednoznačně identifikované
- skupiny prvků, kde jsou jednotlivé prvky identifikované názvem a počtem prvků, k jednotlivým prvkům se pak přistupuje pomocí indexace
- symbolické, které se v architektuře fyzicky nevyskytují, ale jsou součástí jiného prvku

Dále musí být emulátor schopen provádět operace nad těmito prvky. Jsou jimi především:

- operace přiřazení
- matematické operace (sčítání, odčítání, násobení, dělení)
- logické operace (and, or, xor, logický posuv vlevo a vpravo)
- spojení sledů výše uvedených operací do funkcí

Z hlediska přípravy prvků architektury pro zobrazení a jejich logických vazeb, musí emulátor umožňovat vytvářet následující:

- kolekce, tj. skupiny prvků, jež pod jedním názvem spojují více prvků architektury a připravují je pro zobrazení



Obrázek 3.1: UML diagram tříd – detail třídy `Architecture`

- ukazatele, které umožňují definovat logické vazby mezi prvky, např. zvýrazňovat vrchol zásobníku v paměti

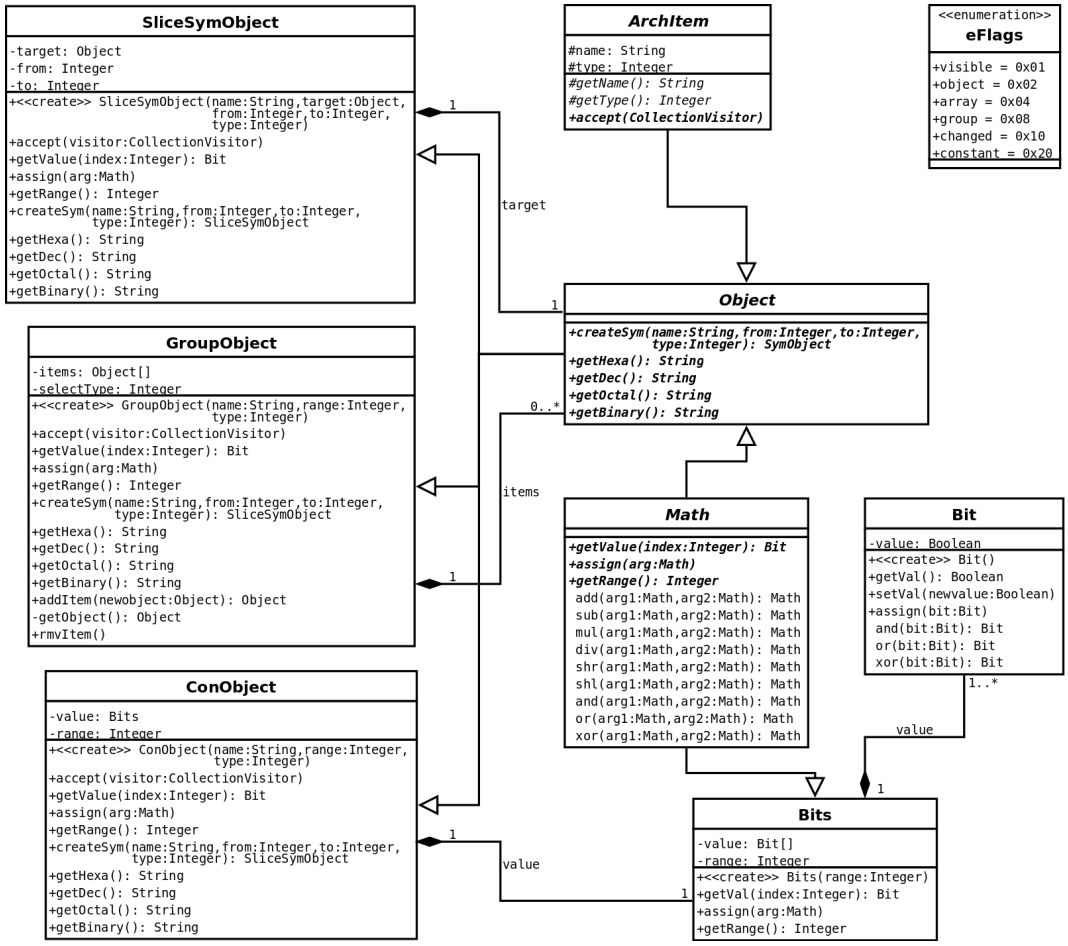
Emulátor bude napsán s využitím metod objektově orientovaného programování. Třídy obsažené v návrhu reflektují konstrukce, které mohou v emulátoru vzniknout.

Architecture Třída se vyskytuje v celém programu pouze v jediné instanci. Jedná se o třídu spravující prvky architektury. Umožňuje přidávat, získávat a odstraňovat prvky, funkce, ukazatele a kolekce architektury. Třída a její nejbližší vazby na třídy další je zachycena na obrázku 3.1.

eFlags Výčtový typ umožňující nastavení vlastností prvků, tj. jejich typ (jedinečný, skupina prvků), rozsah platnosti (aktuálně vykonávaný příkaz, aktuálně vykonávaná funkce, platnost je trvalá po celou dobu běhu interpretu), příznak změny prvku od posledního zobrazení a příznak konstanty.

ArchItem Abstraktní třída definující rozhraní, pod kterým se bude přistupovat k prvkům architektury. Definuje také dva základní atributy všech prvků architektury – název a typ. Název je pro každý prvek v architektuře unikátní. Typ určuje vlastnosti prvku definované pomocí výčtového typu `eFlags`. Třída též nařizuje zděděným prvkům implementovat funkci pro obsloužení objektu třídy `CollectionVisitor`, který je využíván třídou `Collection` k zisku informací pro zobrazení o daném prvku architektury.

Math Třída definující rozhraní prvků schopných matematických operací. Umožňuje provádět základní matetické operace, tj. operace sčítání, odčítání, násobení, dělení, logické operace and, or, xor a bitové posuvy vlevo a vpravo. Veškeré výše zmíněné operace



Obrázek 3.2: UML diagram tříd – detail třídy `Object`

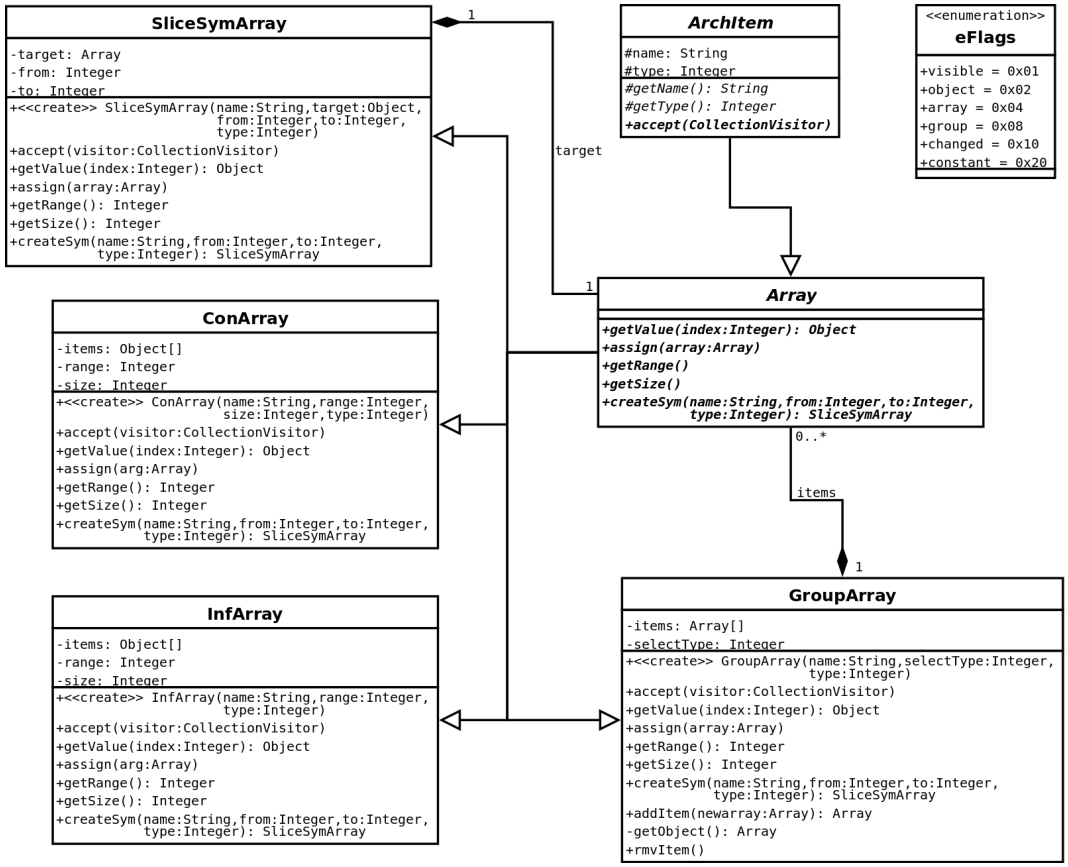
musí být implementovány na bitové úrovni (u zděděných tříd musí být doplněny metody popisující načtení bitu z určitého indexu a operace přiřazení) z důvodu zajištění nezávislosti na architektuře, nad níž emulátor běží.

Bit Základní třída z hlediska reprezentace hodnot používaná v emulátoru. Jelikož má být emulátor univerzální a schopný pracovat s libovolnou architekturou, tedy i s architekturami pracujícími s dnes netypickým počtem bitů, musí být každý bit reprezentován zvlášť. Třída také zapouzdřuje základní operace nad bity, tzn. logické operace and, or a xor.

Bits Třída zapouzdřující sled bitů (reprezentuje hodnoty v architektuře) a základní operace nad nimi dané rodičovskou třídou `Math`.

Object Abstraktní třída definující rozhraní k jedinečným prvkům. Je zděděna od tříd `ArchItem` a `Math`. Zděděné třídy musí být také schopny exportu hodnoty v ní uložené do různých kódování. Třída a její nejbližší vazby na třídy další je zachycena na obrázku 3.2.

ConObject Třída představuje jedinečný prvek v architektuře (např. registr). Je zděděna od třídy `Object` a doplňuje její abstraktní metody.



Obrázek 3.3: UML diagram tříd – detail třídy `Array`

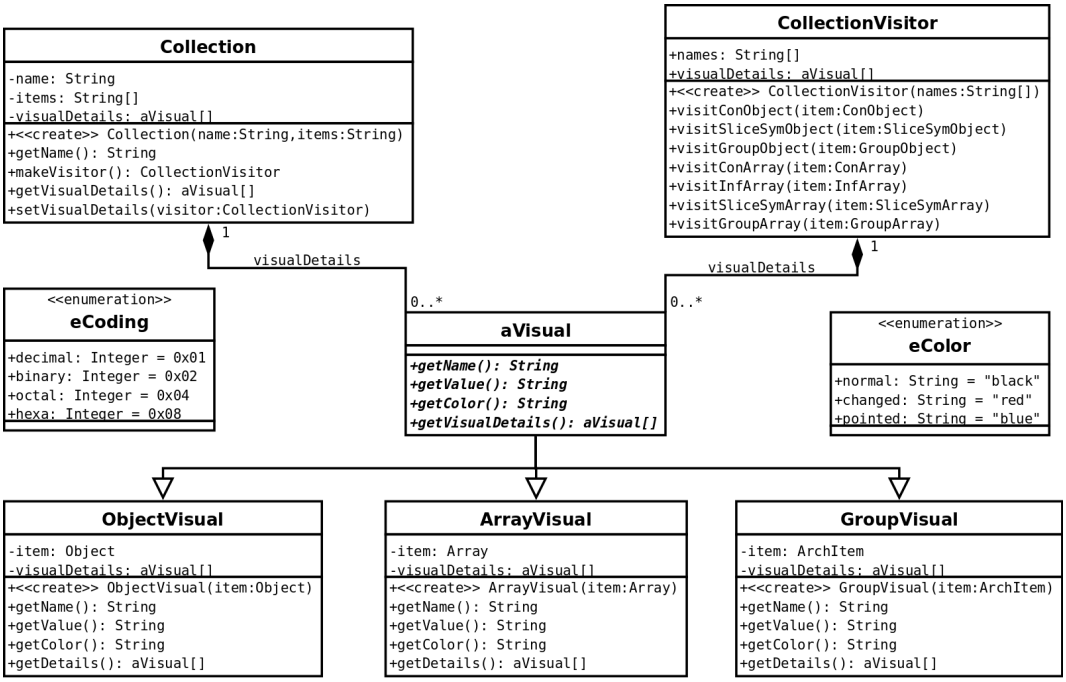
SliceSymObject Třída představující symbolický prvek v architektuře vytvořený z jedinečného prvku pomocí řezu. Transformuje tak operace definované v třídě `Object` pouze na daný rozsah hodnot (bitů) prvku, jehož část symbolický prvek představuje.

GroupObject Třída umožňující zapouzdření více jedinečných prvků pod stejným názvem, kdy je při použití tohoto názvu použit prvek, jehož způsob výběru je určen při definici skupiny (možnosti jsou LIFO, FIFO, RANDOM).

Array Abstraktní třída představující rozhraní skupinám prvků s přístupem pomocí indexu. Je zděděna od třídy `ArchItem` a nařizuje třídám zděděným implementovat metody obstarávající získání objektu z určitého indexu, operaci přiřazení, získání bitové šířky a získání velikosti skupiny (počtu jejích prvků). Třída a její nejbližší vazby na třídy další je zachycena na obrázku 3.3.

ConArray Třída představující skupinu prvků s přístupem pomocí indexu v architektuře (např. paměť). Je zděděna od třídy `Array` a doplňuje její abstraktní metody.

InfArray Třída představující skupinu prvků s přístupem pomocí indexu s neomezeným rozsahem (např. halda v JVM). Tento typ pole lze samozřejmě použít jen u architektur virtuálních. Je zděděna od třídy `Array` a doplňuje její abstraktní metody.



Obrázek 3.4: UML diagram tříd – detail třídy `Collection`

SliceSymArray Třída představující symbolickou skupinu prvků s přístupem pomocí indexu. Transformuje tak operace definované v třídě `Array` pouze na daný rozsah prvků skupiny, jejíž část symbolická skupina představuje.

GroupArray Třída umožňující zapouzdření více skupin prvků s přístupem pomocí indexu pod stejným názvem, kdy je při použití tohoto názvu použit prvek, jehož způsob výběru je určen při definici skupiny (možnosti jsou LIFO, FIFO, RANDOM).

Function Třída představující funkci definovanou v architektuře. Slouží pro uložení parametrů funkce a jednotlivých řádků s kódem funkce.

Parameter Třída představující parametr funkce. Slouží pro uložení názvu parametru funkce a rozsahu řezu, jež se má na daný prvek aplikovat při volání funkce. Parametry jsou při volání funkce předávány odkazem, aby se jejich modifikace uvnitř funkce projevila i po zpracování funkce.

Pointer Třída reprezentující ukazatel umožňující znázornění logických vazeb. Ukazatel je definován svým názvem, skupinou prvků, do které ukazuje, a výrazem, jež je u všech definovaných ukazatelů v architektuře přepočítáván při každém zpracování příkazu interpretu architektury.

Collection Třída umožňující seskupování prvků pro jejich zobrazení v grafickém uživatelském prostředí. Třída a její nejbližší vazby je zachycena na obrázku 3.4.

eColor Výčtový typ určující barvu zobrazení daného prvku v grafickém uživatelském rozhraní.

eCoding Výčtový typ určující kódování zobrazení daného prvku v grafickém uživatelském rozhraní.

CollectionVisitor Třída zajišťující sběr informací o prvcích architektury pro grafické zobrazení.

aVisual Abstraktní třída definující rozhraní pro třídy uchovávající data (jméno, hodnota v daném kódování a barva zobrazení) určená pro grafické uživatelské rozhraní.

ObjectVisual Třída uchovávající data pro zobrazení o jedinečném prvku architektury.

ArrayVisual Třída uchovávající data pro zobrazení o skupině prvků s přístupem pomocí indexu.

GroupVisual Třída uchovávající data pro zobrazení o zapouzdřených skupinách prvků.

Na obrázku 3.5 je možné vidět UML diagram shrnující závislosti mezi výše navrženými třídami. Z důvodu zajištění lepší čitelnosti neobsahuje nákres členské proměnné a názvy metod.

3.2 Návrh tříd pro moduly emulátoru

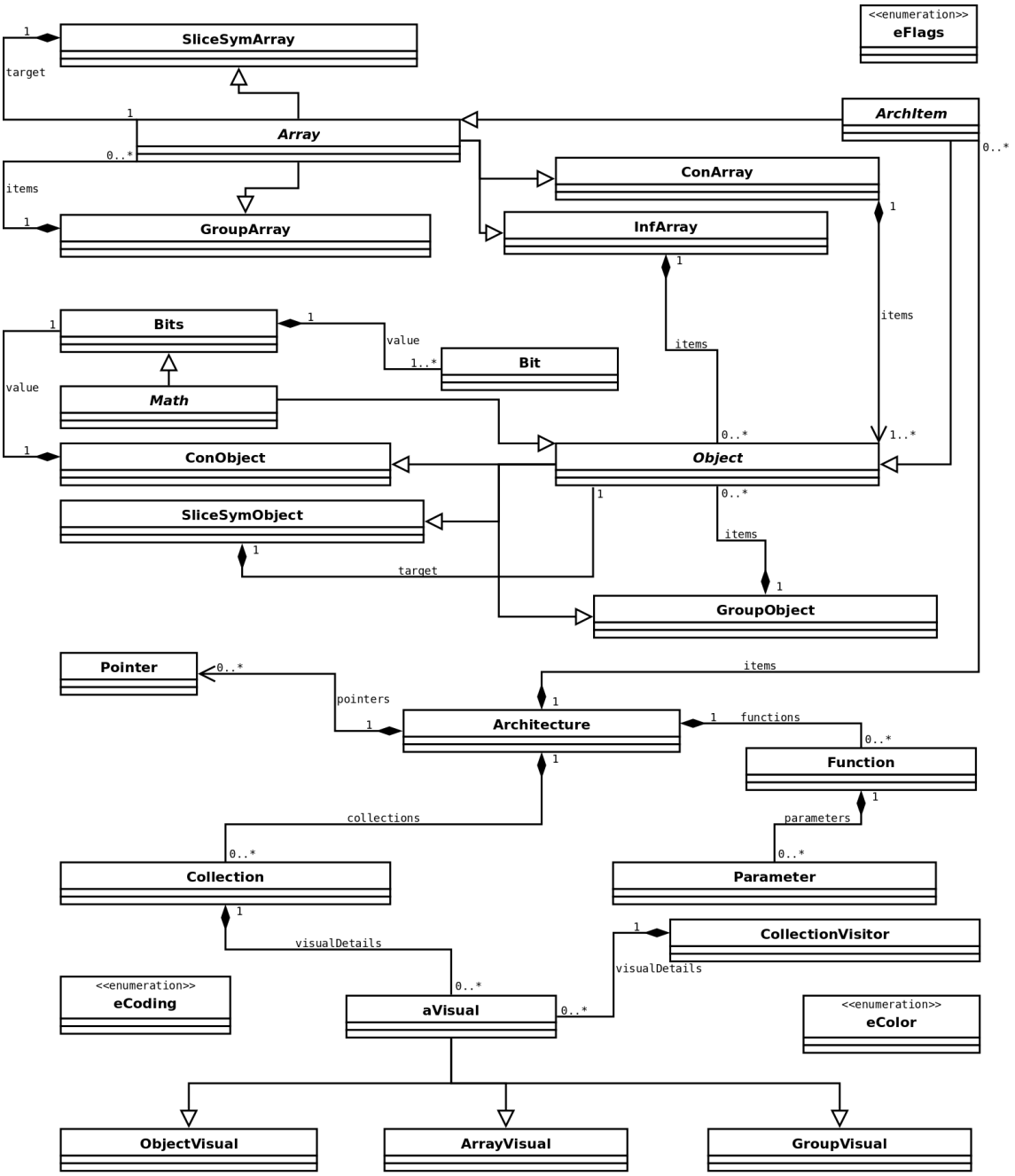
Na obrázku 3.6 je proveden návrh tříd pro jednotlivé hlavní moduly emulátoru. Tyto třídy oddělují výkonné jádro emulátoru od tříd souvisejících s grafickým uživatelským rozhraním a zároveň zajišťují jejich vzájemnou komunikaci. U navržených tříd jsou z důvodu přehlednosti vynechány některé metody a členské proměnné. Z návrhu jsou též vynechány třídy grafického uživatelského rozhraní.

ArchitectureInterpreter Třída zapouzdřuje interpret architektury. Obsahuje funkce zajišťující zpracování kódu v interpretu architektury a vykonání potřebných změn v architektuře, uvedení architektury do výchozího stavu a také funkci, která slouží k načtení prvků určených pro zobrazení v grafickém uživatelském prostředí.

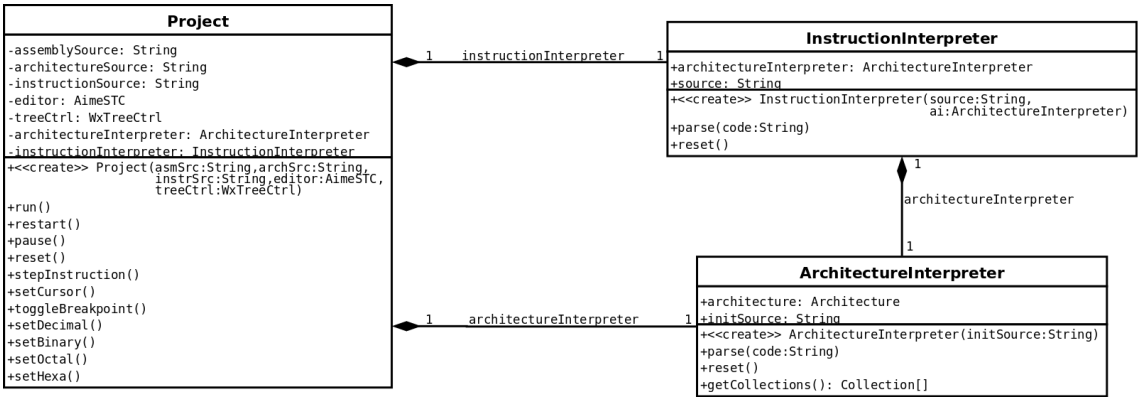
InstructionInterpreter Třída vytváří rozhraní pro interpret instrukcí emulované architektury. Jeho úkolem je rozložit instrukci na sled příkazů jazyka pro popis architektury a postupně je předávat interpretu architektury. Rozklad instrukce asembleru je učiněn na základě souboru s popisem instrukcí. Interpret instrukcí není součástí řešení této práce. Pro korektní činnost emulátoru je nutné jej doplnit a implementovat zde navržené metody.

Project Třída tvoří mezičlánek mezi výkonným jádrem emulátoru a grafickým uživatelským rozhraním. Důvodem zavedení této třídy je zajištění menší vzájemné závislosti mezi hlavními částmi emulátoru. Ve třídě `Project` je též uložena cesta k souborům, jež jsou nutné k emulaci konkrétní architektury, tj. souboru s kódem v asembleru emulované architektury, souboru s počátečním stavem architektury a souboru s popisem instrukční sady.

Díky modulárnímu přístupu k částem emulátoru je umožněna bezproblémová možnost náhrady jednotlivých modulů za jiné bez zásahu do modulů ostatních, např. jádra emulátoru za jeho novější verzi bez nutnosti zásahu do struktury uživatelského rozhraní.



Obrázek 3.5: UML diagram tříd – souhrn navržených tříd



Obrázek 3.6: UML digram tříd pro moduly emulátoru

3.3 Jazyk pro popis architektury

Výše navržené třídy představující prvky, které se mohou v architektuře vyskytovat, je nutné jistým způsobem spravovat. Musí být umožněno definovat jak emulovanou architekturu, tak i operace nad ní. Tímto prostředkem je jazyk pro popis architektury. Vzhledem k předpokládanému nasazení projektu jako výukového prvku a z toho vyplývajícího požavku na jednoduchost byl vytvořen vlastní jazyk pro popis emulované architektury.

Jazyk je vytvořen s důrazem na univerzálnost použití, přehlednost a podobnost s některými již existujícími jazyky, aby se znalí uživatelé nemuseli přizpůsobovat zcela novému prostředí. Pro tento účel byl navržen jazyk inspirovaný jazykem Python, nabízející stručné a jednoduché vyjadřování.

Příkazy jazyka lze rozdělit do tří kategorií:

1. definiční, vytvářející nové prvky nebo funkce architektury
2. rušící, které mažou prvky nebo funkce architektury
3. výkonné, provádějící operace nad prvky

Vstupním bodem gramatiky jazyka zapsané v Backus-Naurově formě je nonterminál `<stmt>`, který se dále bude moci dále rozvinout v příkaz jazyka pro popis architektury.

```
<stmt> ::= <object-def>       // definiční
         | <symbol-def>       // definiční
         | <array-def>        // definiční
         | <infarray-def>     // definiční
         | <pointer-def>      // definiční
         | <collection-def>   // definiční
         | <group-def>        // definiční
         | <func-def>         // definiční
         | <item-del>         // rušící
         | <assignement>      // výkonný
         | <func-call>        // výkonný
```

Definice jedinečného prvku `<object-def>` se skládá z klíčového slova `object`, identifikátoru `<ID>` nově vznikajícího jedinečného prvku, znaku rovná se a vyjádření, jaká je bitová šířka objektu `<bits-expr>`. Definice je ukončena středníkem.

```
<object-def> ::= 'object' <ID> '=' <bits-expr> ';'
```

Bitová šířka `<bits-expr>` je vyjádřena pomocí klíčového slova `bits` a rozsahu pomocí počáteční a koncové hodnoty, jež jsou odděleny dvojtečkou a uzavřeny v hranatých závorkách. Výsledná bitová šířka je dána rozdílem mezi počáteční a koncovou hodnotou.

```
<bits-expr> ::= 'bits' '[' <INT> ':' <INT> ']'
```

Definice symbolických prvků `<symbol-def>` se skládá z klíčového slova `symbol`, identifikátoru `<ID>` nově vznikajícího symbolického prvku a znaku rovná se. Následuje již existující prvek architektury `<arch-item>`, na který bude symbolický prvek odkazovat. Příkaz ukončuje středník.

```
<symbol-def> ::= 'symbol' <ID> '=' <arch-item> ';'
```

Prvkem architektury `<arch-item>` může být fyzický prvek jako celek (určený identifikátorem) nebo část jiného prvku architektury. Tato část je učena pomocí tzv. řezu `<range>`. Řezů lze na jeden prvek architektury aplikovat i více.

```
<arch-item> ::= <ID>
              | <arch-item> <range>
```

Samotný řez `<range>`, vždy uzavřený v hranatých závorkách, je určen buďto pomocí jedné hodnoty výrazu `<expr>`, nebo pomocí rozsahu hodnot oddělených středníkem.

```
<range> ::= '[' <expr> ']'
          | '[' <expr> ':' <expr> ']'
```

Definice skupiny prvků s přístupem pomocí indexu `<array-def>` se bude skládat z klíčového slova `array<>`, kde v lomených závorkách bude vyjádřena bitová šířka `<bits-expr>` jednotlivých prvků pole. Dále se definice skládá z identifikátoru `<ID>`, počtu prvků pole `<INT>` a ukončujícího středníku.

```
<array-def> ::= 'array' '<' <bits-expr> '>' <ID> '(' <INT> ')' ';'
```

Pro dosažení schopnosti emulovat co největší množství architektur musí být též jazyk schopen definovat skupiny prvků s přístupem pomocí indexu, které mají nekonečnou velikost `<infarray-def>` (některé virtuální architektury ji vyžadují, např. Java Virtual Machine). Definice tvořena klíčovým slovem `infarray<>`, kde v lomených závorkách bude vyjádřena bitová šířka `<bits-expr>` jednotlivých prvků pole. Dále se definice skládá z identifikátoru `<ID>` této skupiny prvků s přístupem pomocí indexu a ukončujícího středníku.

```
<infarray-def> ::= 'infarray' '<' <bits-expr> '>' <ID> ';'
```

Definice ukazatele (angl. pointer) `<pointer-def>` je tvořena klíčovým slovem `pointer<>`, kde se v lomených závorkách nachází identifikátor skupiny prvků s přístupem pomocí indexu `<ID>`, v jejímž rámci bude ukazatel odkazovat. Dále se bude definice skládat z identifikátoru ukazatele `<ID>`, znaku rovná se a výrazu `<expr>`, jehož pomocí se vypočítá hodnota indexu, na který bude do skupiny prvků ukazováno. Ukazatel je následně využíván grafickým prostředím pro názornější zobrazení vazeb mezi prvky.

```
<pointer-def> ::= 'pointer' '<' <ID> '>' <ID> '=' <expr> ';'
```

Definice kolekce `<collection-def>` se bude skládat z klíčového slova `collection`, identifikátoru kolekce `<ID>`, znaku rovná se a seznamu (uzavřeného do složených závorek) `<item-list-opt>` identifikátorů prvků kolekce oddělených čárkami. Příkaz je ukončen středníkem. Podobně jako ukazatel má i kolekce význam pro názorné zobrazení pro uživatele, sdružuje prvky, které k sobě jistým způsobem patří a uživatel si je pak bude moci v grafickém prostředí zobrazit dle jeho libosti.

```
<collection-def> ::= 'collection' <ID> '=' '{' <item-list-opt> '}' ';'

<item-list-opt> ::= ''
                  | <item-list> <ID>

<item-list> ::= ''
              | <item-list> <ID> ','
```

Skupiny slouží explicitnímu určení, jak bude nakládáno s prvky, jež vystupují v architektuře pod stejným názvem. Implicitně je k prvkům přistupováno, jako by se jednalo o zásobník (přístup LIFO), tzn. při užití prvku s tímto názvem se přistoupí k poslednímu vytvořenému. Pomocí příkazu definice skupiny `<group-def>` je možné tento výběr modifikovat. Definice skupiny (angl. group) jedinečných prvků je tvořena klíčovým slovem `groupobject<>`, kde se v lomených závorkách nachází typ skupiny `<group-type>`, identifikátorem skupiny `<ID>` (shodný s názvy prvků jež budou do skupiny řazeny) a středníkem ukončujícím příkaz. Obdobným způsobem je tvořena i definice pro skupiny prvků s přístupem pomocí indexu.

```
<group-def> ::= 'groupobject' '<' <group-type> '>' <ID> ';'
              | 'grouparray' '<' <group-type> '>' <ID> ';'
```

Typ přístupu k prvkům ve skupině `<group-type>` může být:

- zásobníkový (LIFO)
- jako ve frontě (FIFO)
- náhodný

```
<group-type> ::= 'LILO' | 'LIFO' | 'RANDOM'
```

Definice funkce `<func-def>` se skládá z hlavičky `<func-head>` a těla `<func-body>`. Hlavička je oddělena od těla pomocí dvojtečky a tělo ukončeno klíčovým slovem `pass`.

```
<func-def> ::= <func-head> ':' <func-body> 'pass'
```

Hlavičku funkce `<func-head>` tvoří klíčové slovo `def`, identifikátor funkce `<ID>` a seznam parametrů `<param-list-opt>` uzavřený v závorkách.

```
<func-head> ::= 'def' <ID> '(' <param-list-opt> ')'

<param-list-opt> ::= ''
                   | <param-list> <ID> <range>

<param-list> ::= ''
               | <param-list> <ID> <range> ','
```

Tělo funkce `<func-body>` tvoří jednotlivé příkazy funkce. Chování těchto příkazů je ekvivalentní s příkazy zadávanými mimo funkci.

```
<func-body> ::= ''
              | <func-body> <func-stat>

<func-stat> ::= <func-object-def>
              | <func-symbol-def>
              | <func-array-def>
              | <func-infarray-def>
              | <func-collection-def>
              | <func-item-del>
              | <func-assignment>
              | <func-func-call>
```

Příkaz zrušení prvku nebo funkce `<item-del>` je definován pomocí klíčového slova `del`, identifikátoru rušeného prvku `<ID>` a středníku ukončujícího příkaz.

```
<item-del>: 'del' <ID> ';'
```

Prvním výkonným příkazem jazyka je přiřazení (angl. assignement) `<assignment>` skládající se z L-hodnoty a R-hodnoty. L-hodnota je tvořena obecným prvkem architektury `<arch-item>` definovaným výše. To umožňuje přiřazovat hodnotu nejen fyzickým prvkům jako celku, ale i jejich řezům. R-hodnotu pak tvoří obecný syntakticky a sémanticky správný výraz `<expr>`.

```
<assignment> ::= <arch-item> '=' <expr> ';'
```

Výrazem jazyka pro popis architektury `<expr>` může být obecný prvek architektury `<arch-item>`, celé číslo `<INT>`, další výraz uzavřený v závorkách nebo výrazy, na které je aplikována některá matematická nebo logická operace. U jednotlivých možností přechodů je naznačena jejich priorita.

```
<expr>  ::= <arch-item>              // pri = 6
          | <INT>                    // pri = 6
          | '(' <expr> ')'           // pri = 5
          | <expr> '/' <expr>        // pri = 3
          | <expr> '*' <expr>        // pri = 3
          | <expr> '+' <expr>        // pri = 2
```

```
| <expr> '-' <expr>        // pri = 2
| <expr> '&' <expr>        // pri = 1
| <expr> '|' <expr>        // pri = 1
| <expr> '^' <expr>        // pri = 1
| <expr> '<<' <expr>       // pri = 0
| <expr> '>>' <expr>       // pri = 0
```

Druhým výkonným příkazem jazyka pro popis architektury je volání funkce `<func-call>`. Skládá se z identifikátoru funkce `<ID>` a parametrů funkce `<call-param-list-opt>` uzavřených do závorek. Příkaz je ukončen středníkem.

```
<func-call> ::= <ID> '(' <call-param-list-opt> ')' ';'

<call-param-list-opt> ::= ''
                        | <call-param-list> <expr>

<call-param-list> ::= ''
                    | <call-param-list> <expr> ','
```

Celý návrh gramatiky i s příkladem popisu architektury je uveden v dodatku A, resp. v dodatku B. Cílem návrhu jazyka pro popis architektury nebylo vytvořit jazyk s plnou výpočetní silou, důraz byl kladen zejména na intuitivnost a jednoduchost použití. Zvýšení výpočetní síly emulátoru jako celku může být dosaženo zařazením vhodného interpretu instrukcí.

3.4 Grafické uživatelské prostředí

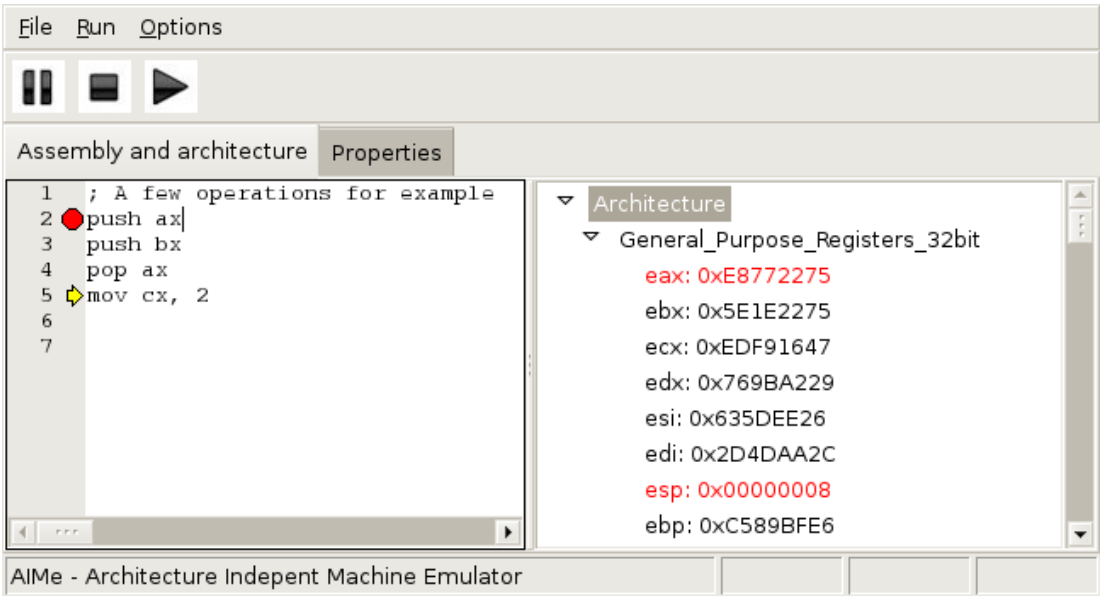
Hlavními vlastnostmi grafického uživatelského prostředí (GUI) jsou možnost komfortního psaní kódu v asembleru emulované architektury a zobrazování aktuálního stavu architektury.

Část určená pro psaní kódu disponuje následujícími vlastnostmi:

- Zobrazení a možnost pohodlné editace kódu v asembleru emulované architektury
  - Zobrazení čísel řádků
  - Možnost vyjmutí, kopírování a vložení kódu nebo jeho části
  - Zobrazení aktuálního řádku a sloupce
- Možností krokovat kód v asembleru a zobrazovat aktuální pozici v kódu
- Možností definovat a zobrazovat body přerušení

Druhá část, jež je určená zobrazení architektury, má následující vlastnosti:

- Užití hierarchické struktury — stromu
- Zobrazení jedinečných a symbolických prvků
  - Zobrazení názvu a hodnoty
  - Možnost měnit kódování užité pro zobrazení hodnoty (hexadecimální, desetinné, binární)



Obrázek 3.7: Návrh grafického uživatelského rozhraní

- Zobrazení skupin prvků s přístupem pomocí indexu
  - Zobrazení názvu a počtu prvků ve skupině
  - Interakcí (např. poklepáním myší) lze zobrazit detailnější popis
- Zvýraznění změněných prvků
- Zvýraznění logických vazeb mezi prvky

Výsledný grafický návrh vytvořený dle výše uvedených požadavků je možné vidět na obrázku 3.7.

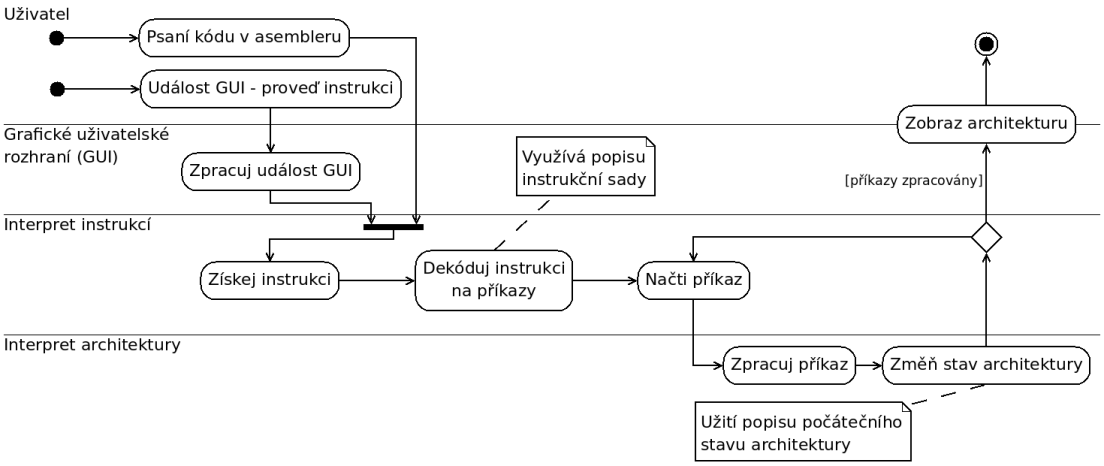
3.5 Princip činnosti emulátoru

Interpret jazyka pro popis architektury je navržen jako dynamický — změny v architektuře se projevují po jednotlivých příkazech. Na obrázku 3.8 je naznačena činnost celého emulátoru při zpracování jednoho příkazu napsaného v asembleru emulované architektury.

Princip činnosti:

1. Načtení počáteční konfigurace architektury
2. Uživatel píše kód v asembleru emulované architektury
3. Uživatel zvolil možnost „proveď instrukci“ pomocí nabídky v grafickém uživatelském rozhraní
4. Grafické uživatelské rozhraní zpracuje událost
5. Řízení je předáno interpretu instrukcí, jehož úkolem je rozložit instrukci na sled příkazů jazyka pro popis architektury

6. Příkazy pro popis architektury jsou postupně zpracovávány interpretem architektury
7. Po zpracování posledního příkazu je změněný stav architektury prostřednictvím grafického uživatelského rozhraní opět zobrazen uživateli



Obrázek 3.8: UML diagram aktivit popisující činnost emulátoru

Kapitola 4

Implementace emulátoru

Pro implementaci aplikace byl použit jazyk Python. Výhodou tohoto interpretovaného programovacího jazyka je vyšší abstrakce při programovaní, která umožňuje lepší čitelnost při ladění a rychlejší pochopení kódu novými vývojáři. Implementace probíhala z velké části přesně podle výše uvedeného návrhu. V této kapitole jsou probrány pouze některé klíčové vlastnosti implementace a rozložení jednotlivých částí aplikace do modulů.

4.1 Syntaktický analyzátor

K implementaci syntaktického analyzátoru pro zpracování jazyka pro popis architektury byl zvolen nástroj PLY [1]. PLY je nástroj napsaný v jazyce Python, jež implementuje hojně užívané programy LEX a YACC sloužící pro tvorbu překladačů. Důvodem k tomuto postupu byl opět rychlejší vývoj aplikace, než v případě vytváření překladače vlastního.

Nástroj PLY dovoluje jednoduše přepsat gramatiku navrženou v Backus-Naurově formě do programového kódu v podobě LALR(1) gramatiky a každému přechodu gramatiky dovoluje přidat jeho sémantiku, v tomto případě příslušnou operaci nad stavem architektury.

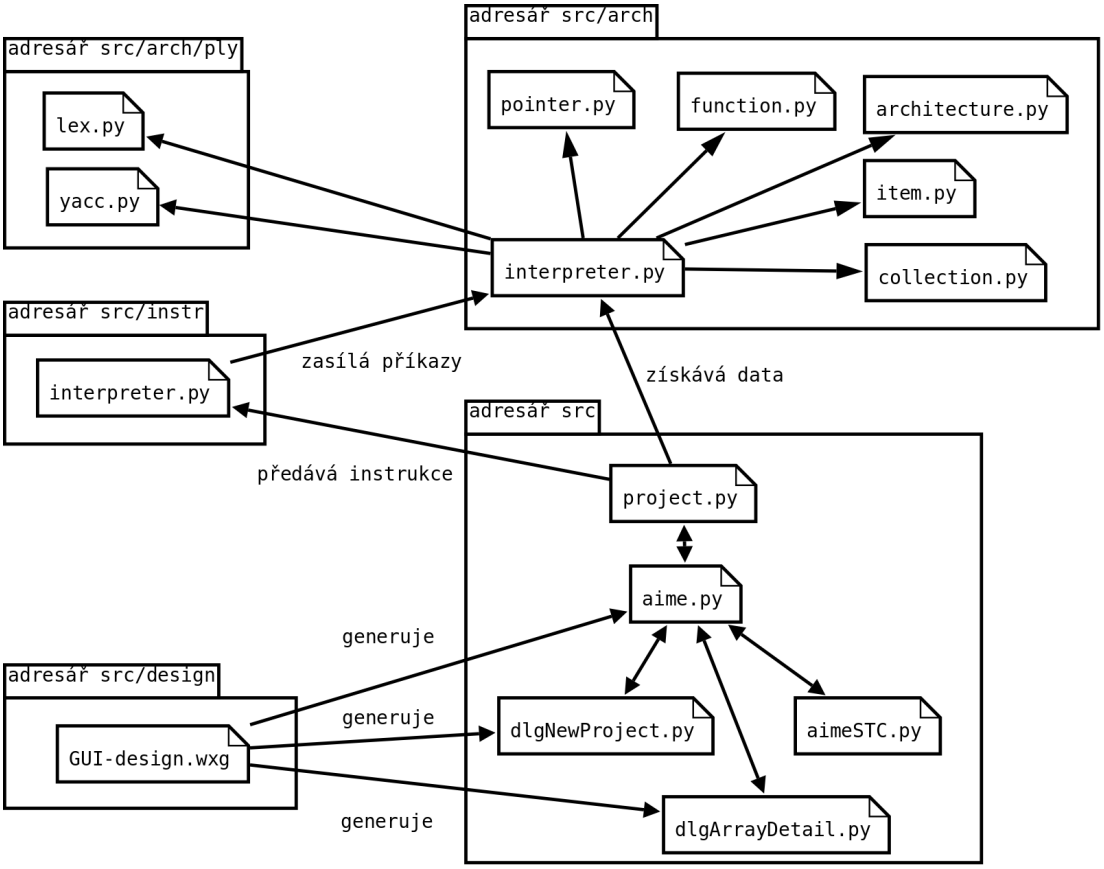
4.2 Grafické uživatelské rozhraní

Pro grafické uživatelské rozhraní byla zvolena grafická nadstavba wxPython, patřící do rodiny wxWidgets. Jejím použitím byla zabezpečena dobrá přenositelnost mezi různými operačními systémy. Rozmístění grafických prvků do formulářů a dialogových oken bylo vytvořeno pomocí programu wxGlade.

4.3 Rozložení do modulů

Na obrázku 4.1 je zobrazeno rozložení projektu do modulů a jejich vzájemná spolupráce. Implementace výkonného jádra aplikace (syntaktického analyzátoru) je uložena v souboru `interpreter.py` v adresáři `src/arch`. Využívá přitom soubory `architecture.py`, `item.py`, `collection.py`, `function.py` a `pointer.py` obsahující třídy reflektující konstrukce, které mohou vzniknout v jazyce pro popis architektury, a soubory `lex.py` a `yacc.py` v adresáři `src/arch/ply`, které obsahují implementaci nástroje PLY.

Třída `Project` představující mezičlánek mezi výkonným jádrem emulátoru a grafickým uživatelským rozhraním je uložena v souboru `project.py` v adresáři `src`.



Obrázek 4.1: Rozložení projektu do modulů

Implementace grafického uživatelského rozhraní je uložena v několika souborech. Soubor `aime.py` obsahuje zdrojový kód hlavního okna aplikace a je vstupním bodem aplikace. Soubory `dlgNewProject.py` představuje dialogové okno pro vytvoření nového projektu. Soubor `aimeSTC.py` obsahuje implementaci ovládacího prvku umožňujícího psaní kódu v asembleru tj. editoru kódu. Chování tohoto prvku grafického rozhraní je implementováno zvlášť z důvodu jeho komplexního ovládání. V souboru `dlgArrayDetail.py` je kód dialogového okna zobrazujího detailnější pohled na skupiny prvků s přístupem pomocí indexu. Soubory `aime.py`, `dlgNewProject.py`, `aimeSTC.py` a `dlgArrayDetail.py` jsou uloženy v adresáři `src`.

Návrh rozložení grafických prvků obsažených v souborech `aime.py`, `dlgNewProject.py` a `dlgArrayDetail.py` je definován v souboru `GUI-design.wxg` v adresáři `src/design`. Tento soubor a s ním i grafické rozhraní aplikace je možné editovat programem wxGlade.

Kapitola 5

Testy implementace

Aby bylo možné ověřit správnost emulátoru, byla jeho implementace rozšířena o testovací rozhraní. Způsob ovládání tohoto rozhraní je probrán v první části této kapitoly. Druhá část obsahuje informace související s testováním projektu.

5.1 Testovací rozhraní

Do testovacího rozhraní je možné vstoupit, pokud je jako vstupní bod do aplikace použit soubor `interpreter.py` z adresáře `src/arch`. Aplikace si nejdříve vyžádá zadání cesty k souboru, který definuje počáteční stav architektury. Při nezadání žádné cesty je použita architektura prázdná. Aplikace nyní očekává zadavání příkazů jazyka pro popis architektury na standardní vstup. Po každém vykonaném příkazu je zobrazen stav architektury, aby bylo možné určit, zda příkaz provedl očekávané změny v architektuře. Ukončení činnosti testovacího rozhraní se provádí zadáním znaku konce souboru `EOF`.

5.1.1 Příklad testu

- Jméno: Vytvoření jedinečného prvku v prázdné architektuře.
- Popis: Pomocí příkazu `object` jazyka pro popis architektury bude vytvořen jedinečný prvek s názvem `ACC` (např. akumulátor) o bitové šířce 8bitů.
- Vstup: Prázdná architektura, příkaz: `object acc = bits[0:7];`
- Očekávaný výstup: Ve výpisu všech prvků v sekci `Items` existuje pouze jediný prvek třídy `ConObject` o bitové šířce 8bitů s názvem `ACC`.

```
INFO: Entering testing/debugging mode.
    Please input path to file with the architecture initial state
    or leave blank for none.

File>
INFO: No initial state file. No initial state loaded.

=== Current items in architecture ===
--- Items:
--- Collections:
```

```
--- Functions:
--- Pointers:

AIMe> object ACC = bits[0:7];
DEBUG: Creating object: 'ACC'

=== Current items in architecture ===
--- Items:
ACC, flags: 0x3, class: item.ConObject, val: 01110110
--- Collections:
--- Functions:
--- Pointers:
```

- Úspěšnost: PASS

5.2 Testování emulátoru

V rámci testování projektu byla provedena série testů zaměřená na klíčové vlastnosti emulátoru. Část z těchto testů je uvedena v dodatku D.

Kapitola 6

Závěr

Cílem této práce bylo navrhnout a implementovat emulátor umožňující začátečníkům snadný vstup do problematiky programování ve strojovém kódu různých platforem. Byly probrány některé současné projekty věnující se emulaci procesorů. Na základě jejich analýzy byly vytvořeny požadavky na nově vyvíjený emulátor. Byly navrženy třídy reprezentující typické prvky v architekturách a navržen jazyk pro popis architektury emulovaného stroje. Při tomto návrhu byl kladen důraz na univerzálnost výsledného řešení. V navrženém jazyce pro popis architektur byly popsány architektury Intel x86 a částečně JVM, které byly v emulátoru úspěšně otestovány. Na popisu dalších architektur se pracuje. Dále bylo navrženo pro začátečníky vhodné grafické uživatelské prostředí. Výsledná implementace byla otestována souborem testů. Tímto způsobem vznikl nástroj schopný emulace většiny dnešních a vzhledem ke své univerzálnosti i budoucích počítačových architektur. Pokročilejším uživatelům by měla aplikace také poskytnout možnost pro návrh architektur vlastních. V rámci tvorby projektu byly vytvořeny i webové stránky [10] dokumentující projekt a jeho aktuální vývoj.

6.1 Možné pokračování projektu

Do budoucna by bylo zajímavé rozšířit emulátor tak, aby umožňoval nejen emulaci instrukcí zapsaných v asembleru, ale i přímo zdrojových binárních souborů. Bylo by nutné hlubší studium instrukčních sad různých architektur i organizace jejich paměti.

Další možností je vytvoření přehlednější grafického znázornění, než je hierarchický strom. Architektura by mohla být například vykreslována pomocí grafických funkcí na plochu pro toto zobrazení určeného okna a logické vazby mezi jejími prvky by byly znázorněny lepším způsobem.

Literatura

[1] Beazley, D.: PLY (Python Lex-Yacc) [online]. `http://www.dabeaz.com/ply/`, 2009-02-06 [cit. 2009-05-01].

[2] Kapartzianis, V.: ZX32 – ZX Spectrum Emulator [online]. `http://www.geocities.com/SiliconValley/Bay/9932/`, 2000-02-16 [cit. 2009-04-04].

[3] Kolektiv autorů: PearPC – Emulátor architektury PowerPC [online]. `http://pearpc.sourceforge.net/`, 2005-12-20 [cit. 2009-04-20].

[4] Kolektiv autorů: MAME – Multiple Arcade Machine Emulator [online]. `http://mamedev.org/`, 2009-03-14 [cit. 2009-04-15].

[5] Kolektiv autorů: Intel 64 and IA-32 Architectures Software Developer's Manuals, Vol. 1 Basic Architecture [online]. `http://download.intel.com/design/processor/manuals/253665.pdf`, 2009-03 [cit. 2009-05-10].

[6] Lindholm, T., Yellin, F.: The Java Virtual Machine Specification, Second Edition [online]. `http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html`, 1999, iSBN 978-0201432947.

[7] Pickett, Joseph P. a kolektiv (editor): *The American Heritage Dictionary of the English Language. Fourth edition.* Boston: Houghton Mifflin Company, 2000, iSBN 0-395-82517-2.

[8] Wikipedie, otevřená encyklopedie: Java Virtual Machine [online]. `http://cs.wikipedia.org/wiki/Java_Virtual_Machine`, 2009-05-15 [cit. 2009-05-16].

[9] Wikipedie, otevřená encyklopedie: Intel x86 [online]. `http://en.wikipedia.org/wiki/X86`, 2009-05-16 [cit. 2009-05-16].

[10] WWW stránky: Stránky projektu [online]. `http://www.stud.fit.vutbr.cz/~xcharv03/projects/IBP`, 2009-05-14 [cit. 2009-05-16].

Dodatek A

Gramatika pro popis architektury

Zde uvedený návrh gramatiky je také uložen na CD v souboru `grammar.txt` ve složce `doc`.

```
/// --- Příkaz ---
<stmt>  ::= <object-def>
          | <symbol-def>
          | <array-def>
          | <infarray-def>
          | <pointer-def>
          | <collection-def>
          | <group-def>
          | <func-def>
          | <item-del>
          | <assignement>
          | <func-call>

/// --- Jedinečné prvky ---
<object-def> ::= 'object' <ID> '=' <bits-expr> ';'
<bits-expr> ::= 'bits' '[' <INT> ':' <INT> ']'

/// --- Symbolické prvky ---
<symbol-def> ::= 'symbol' <ID> '=' <arch-item> ';'
<arch-item> ::= <ID>
          | <arch-item> <range>
<range> ::= '[' <expr> ']'
          | '[' <expr> ':' <expr> ']'

/// --- Skupiny prvků s˜přístupem pomocí indexu ---
<array-def> ::= 'array' '<' <bits-expr> '>' <ID> '(' <INT> ')' ';'
<infarray-def> ::= 'infarray' '<' <bits-expr> '>' <ID> ';'

/// --- Ukazatele ---
<pointer-def> ::= 'pointer' '<' <ID> '>' <ID> '=' <expr> ';'

/// --- Kolekce ---
<collection-def> ::= 'collection' <ID> '=' '{' <item-list-opt> '}' ';'
```

```
<item-list-opt> ::= ''
                  | <item-list> <ID>
<item-list> ::= ''
              | <item-list> <ID> ','

/// --- Skupiny ---
<group-def> ::= 'groupobject' '<' <group-type> '>' <ID> ';'
              | 'grouparray' '<' <group-type> '>' <ID> ';'
<group-type> ::= 'LILO' | 'FIFO' | 'RANDOM'

/// --- Rušení prvku ---
<item-del> ::= 'del' <ID> ';'

/// --- Funkce ---
<func-def> ::= <func-head> ':' <func-body> 'pass'

// --- Hlavička funkce ---
func-head ::= 'def' <ID> '(' <param-list-opt> ')'
<param-list-opt> ::= ''
                   | <param-list> <ID> <range>
<param-list> ::= ''
               | <param-list> <ID> <range> ','

// --- Tělo funkce ---
<func-body> ::= ''
              | <func-body> <func-stat>
<func-stat> ::= <func-object-def>
              | <func-symbol-def>
              | <func-array-def>
              | <func-infarray-def>
              | <func-collection-def>
              | <func-item-del>
              | <func-assignment>
              | <func-func-call>

/// --- Operace přiřazení ---
<assignement> ::= <arch-item> '=' <expr> ';'

/// --- Volání funkce ---
<func-call> ::= <ID> '(' <call-param-list-opt> ')' ';'
<call-param-list-opt> ::= ''
                        | <call-param-list> <expr>
<call-param-list> ::= ''
                    | <call-param-list> <expr> ','

/// --- Výrazy ---
<expr>  ::= <arch-item>                  // pri = 6
          | <INT>                    // pri = 6
```

```
| '(' <expr> ')'          // pri = 5
| <expr> '/' <expr>       // pri = 3
| <expr> '*' <expr>       // pri = 3
| <expr> '+' <expr>       // pri = 2
| <expr> '-' <expr>       // pri = 2
| <expr> '&' <expr>       // pri = 1
| <expr> '|' <expr>       // pri = 1
| <expr> '^' <expr>       // pri = 1
| <expr> '<<' <expr>      // pri = 0
| <expr> '>>' <expr>      // pri = 0
```

Dodatek B

Příklad popisu architektury

Uvedený příklad popisu architektury, konkrétně architektury Intel x86, představuje část souboru uloženého na CD v souboru `intelx86.arch` ve složce `examples`.

```
// Basic 32-bit Intel Architecture
// --- General-Purpose Registers ---
// Accumulator for operands and results data
object eax = bits[0:31];
symbol ax = eax[0:15];
symbol al = ax[0:7];
symbol ah = ax[8:15];

// Pointer to data in the DS segment
object ebx = bits[0:31];
symbol bx = ebx[0:15];
symbol bl = bx[0:7];
symbol bh = bx[8:15];

// Counter for string and loop operations
object ecx = bits[0:31];
symbol cx = ecx[0:15];
symbol cl = cx[0:7];
symbol ch = cx[8:15];

// I/O pointer
object edx = bits[0:31];
symbol dx = edx[0:15];
symbol dl = dx[0:7];
symbol dh = dx[8:15];

// Pointer to data in the segment pointed to by the DS register; source
// pointer for string operations
object esi = bits[0:31];
symbol si = esi[0:15];

// Pointer to data (or destination) in the segment pointed to by the ES
```

```
// register; destination pointer for string operations
object edi = bits[0:31];
symbol di = edi[0:15];

// Stack pointer (in the SS segment
object esp = bits[0:31];
symbol sp = esp[0:15];

// Pointer to data on the stack (in the SS segment)
object ebp = bits[0:31];
symbol bp = ebp[0:15];

collection GeneralPurposeRegisters32bit = {eax,ebx,ecx,edx,esi,edi,ebp,esp};

// --- Segment Registers ---
object cs = bits[0:15];
object ds = bits[0:15];
object ss = bits[0:15];
object es = bits[0:15];
object fs = bits[0:15];
object gs = bits[0:15];
collection SegmentRegisters = {cs,ds,ss,es,fs,gs};

// --- Program Status and Control Register ---
object eflags = bits[0:31];
collection ProgramStatusAndControlRegister = {eflags};

// --- Instruction Pointer ---
object eip = bits[0:31];
symbol ip = eip[0:15];
collection InstructionPointer = {eip};

// --- Memory ---
array<bits[0:7]> memory(1048576);
collection Memory = {memory};

// --- Common operations ---
def stbyte(address[0:15], value[0:7]):
  memory[address] = value[0:7];
pass;

def ldbyte(address[0:15], value[0:7]):
  value[0:7] = memory[address];
pass;

// ...
```

Dodatek C

Instalace a příklad použití

Tento dodatek obsahuje návod k instalaci implementové aplikace a za pomoci příkladu naznačuje způsob jejího možného využití.

C.1 Instalace

Zdrojový kód aplikace je uložen na přiloženém CD v adresáři `src`. Ke spuštění aplikace je nutné mít nainstalovaný interpret jazyka Python (minimálně ve verzi 2.5.x). Jeho instalace pro 32 a 64bitové systémy Windows jsou k dispozi na CD v adresáři `install`. V prostředí UNIX/Linux je interpret jazyka Python většinou součástí instalace samotného operačního systému. Pokud tomu tak není, je možné interpret stáhnout z webové adresy `http://www.python.org/download/`.

Dále je nutná instalace grafické nadstavby wxPython (testováno na verzi 2.8.8.0). Instalace pro systémy Windows je opět uložena na CD v adresáři `install`, v prostředí UNIX/Linux je možné nadstavbu stáhnout a nainstalovat pomocí informací dostupných na webové stránce `http://www.wxpython.org/download.php`.

Projekt byl testován v prostředí operačního systému Linux (distribuce Ubuntu 8.10 - the Intrepid, 64bit), Windows Vista 64bit a Windows XP 32bit.

C.2 Příklad použití

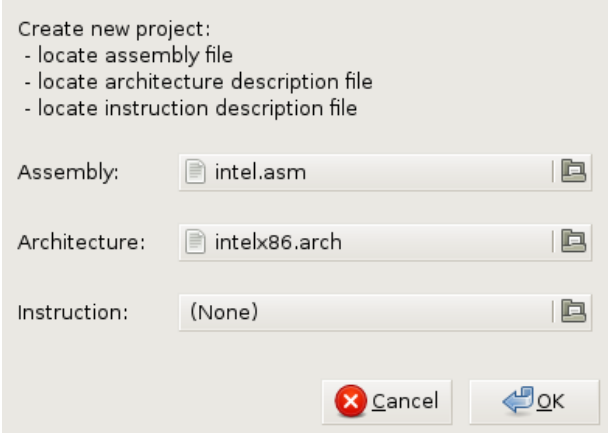
V následujících několika krocích je proveden nástin možného využití emulátoru.

C.2.1 Spuštění aplikace

Spuštění emulátoru z adresáře se provádí z prostředí příkazové řádky zadáním příkazu `python aime.py` zadaným v adresáři `src`. Po jeho zadání by se mělo zobrazit hlavní okno programu.

C.2.2 Vytvoření projektu

Vytvoření nového projektu se provádí vybráním položky `Nový` z nabídky `Soubor`. Objeví se dialog založení nového projektu zobrazený na obrázku C.1. V tomto dialogu je nutné vybrat soubor s asemblerem emulované architektury (standardně přípona `*.asm`), soubor s popisem počátečního stavu architektury (standardně přípona `*.arch`) a nakonec soubor



Obrázek C.1: Dialog založení nového projektu

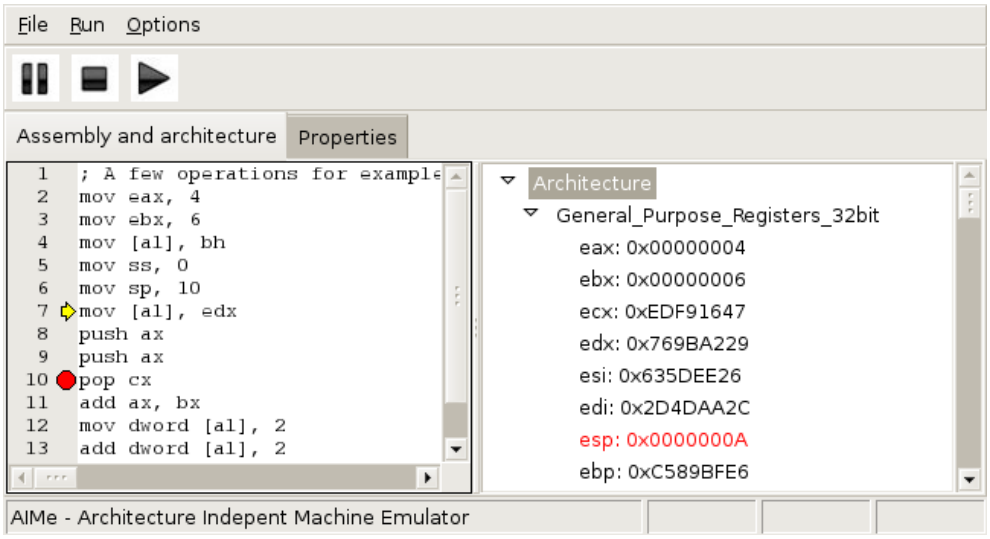
s popisem instrukční sady architektury (standardně přípona `*.instr`). Jelikož návrh interpretu instrukční sady nebyl součástí zadání tohoto projektu a očekává se použití interpretu dodaného uživatelem, nemá tato volba prozatím smysl. Pro demonstrační účely je spolu s řešením tohoto projektu dodán jednoduchý překladač instrukcí z architektury Intel x86 do příkazů jazyka pro popis architektury. Jeho možnosti jsou velice omezené (implementován je pouze překlad instrukcí `mov`, `push`, `pop`, `add` a `inc`) a nebyl dostatečně testován. V rámci tohoto příkladu je vybán jako soubor `intel.asm` ze složky `examples` jako zdrojový kód asembleru a soubor `intelx86.arch` ze stejné složky jako počáteční popis stavu architektury. Po potvrzení voleb je emulátor připraven krokovat kód v asembleru.

C.2.3 Hlavní okno programu

Hlavní okno programu zobrazené na obrázku C.2 se skládá ze dvou částí. Levá je určena pro prohlížení a editaci kódu v asembleru, zatímco pravá slouží k zobrazení aktuálního stavu architektury (v tomto případě Intel x86). Krokování kódu můžeme začít výběrem položky `Krokuj instrukci` z nabídky `Spustit` (klávesová zkratka `F8`). Bude zobrazen žlutý ukazatel reprezentující pozici v rámci kódu. Nyní krokováním dalších intrukcí můžeme sledovat jejich dopad na architekturu, poslední instrukcí změněné prvky architektuře jsou zvýrazňovány červenou barvou. Modrá barva značí důležité logické vazby mezi prvky (v tomto případě vrchol zásobníku). K rychlejšímu průchodu více intrukcemi je možné využít bodů přerušení (reprezentovány čeveným kruhem). Bod přerušení se vytváří výběrem položky `Vytvořit bod přerušení` z nabídky `Spustit` (klávesová zkratka `F9`). Pokud je při procházení automatickém procházení kódu, které je spouštěno pomocí tlačítka „Přehrát“ z panelu nástrojů, dosaženo bodu přerušení, aplikace zobrazí dialogové okno a automatické krokování je pozastaveno. Pozastavení můžeme dosáhnout i stiskem tlačítka „Pauza“ z téhož panelu nástrojů. K nastavení pozice v kódu na aktuální řádek je možné využít položku `Nastavit kurzor` z nabídky `Spustit` (klávesová zkratka `F7`).

C.2.4 Nabídka možnosti a záložka vlastnosti

Nabídka možnosti umožňuje uživateli vyběr způsobu zobrazení hodnot prvků obsažených v architektuře. Nabízená kódování jsou hexadecimální, binární, osmičkové a desetinné.



Obrázek C.2: Práce v hlavním okně programu

Až do této doby se příklad odehrával na záložce `Architektura a kód v asembleru`. Záložka `Vlastnosti` je druhou nabízenou. Lze v ní procházet obsah souborů, které byly zvoleny při vytváření nového projektu — soubor s popisem počátečního stavu a soubor s popisem intrukční sady architektury. Oba soubory mohou být pomocí příslušných tlačítek znovunahrány do příslušných interpretů.

C.2.5 Uložení a ukončení činnosti

K uložení činnosti provedené v emulátoru je možné použít nabídku `Soubor`.

Výběrem položky `Uložit` je celý projekt uložen, tzn. jsou uloženy pouze přístupové cesty k třem hlavním souborům — kód v asembleru, počáteční stav architektury a popis intrukční sady. Projekt může být načten znovu výběrem položky `Open` ze stejné nabídky.

Položka `Uložit kód asembleru` uloží pouze aktuální obsah editačního okna. Obsah může být posléze načten pomocí položky `Otevřít kód v asembleru` ze stejné nabídky.

Dodatek D

Testování implementace

Tento dodatek je zaměřen na provedení několika testů výsledné implementace projektu. Do testovacího rozhraní je možné vstoupit použitím souboru `interpreter.py` ze složky `src/arch` jako vstupního bodu programu, tj. zadáním příkazu `python interpreter.py` v prostředí příkazové řádky z uvedeného adresáře.

D.1 Symbolické prvky

- Jméno: Vytvoření a práce se symbolickým prvkem.
- Popis: Pomocí příkazu `symbol` jazyka pro popis architektury bude vytvořen symbolický prvek `AL` o bitové šířce 8bitů, který bude v rozsahu nultého až sedmého bitu částí jedinečného prvku s názvem `AX` o bitové šířce 16bitů. Pomocí operace přiřazení bude ověřena funkčnost symbolického prvku.
- Vstup: Architektura obsahující jedinečný prvek s názvem `AX` o bitové šířce 16bitů, příkazy: `symbol AL = AX[0:7];` a `AL = 0;`
- Očekávaný výstup: Ve výpisu všech prvků v sekci `Items` existují dva prvky, jeden třídy `ConObject` o bitové šířce 16bitů s názvem `AX` a druhý třídy `SliceSymObject` s názvem `AL` o bitové šířce 8bitů. Hodnota nultého až sedmého bitu obou prvků je stejná ve všech částech testu.

```
... výstup zkrácen ...

=== Current state of architecture ===
--- Items:
AX, flags: 0x3, class: item.ConObject, val: 0011111000101000
--- Collections:
--- Functions:
--- Pointers:

AIMe> symbol AL = AX[0:7];
DEBUG: Creating symbol: 'AL'

=== Current state of architecture ===
--- Items:
```

```
AX, flags: 0x3, class: item.ConObject, val: 0011111000101000
AL, flags: 0x3, class: item.SliceSymObject, val: 00101000
--- Collections:
--- Functions:
--- Pointers:

AIMe> AL = 0;

=== Current state of architecture ===
--- Items:
AX, flags: 0x13, class: item.ConObject, val: 0011111000000000
AL, flags: 0x13, class: item.SliceSymObject, val: 00000000

... výstup zkrácen ...
```

- Úspěšnost: PASS

D.2 Práce s výrazy

- Jméno: Práce s výrazy, priorita operátorů.
- Popis: Do jedinečného prvku je přiřazena hodnota výrazu složeného z matematických operátorů tak, aby bylo možné ověřit správnost priorit operátorů.
- Vstup: Architektura obsahující jedinečný prvek s názvem `ACC` o bitové šířce 8bitů, příkaz: `ACC = 4 + (ACC ^ ACC) * 2 >> 3 & 2;`
- Očekávaný výstup: Ve výpisu všech prvků v sekci `Items` existuje jeden prvek třídy `ConObject` o bitové šířce 8bitů s názvem `ACC` s binární hodnotou 00000001.

```
... výstup zkrácen ...

=== Current state of architecture ===
--- Items:
ACC, flags: 0x3, class: item.ConObject, val: 01010100
--- Collections:
--- Functions:
--- Pointers:

AIMe> ACC = 4 + (ACC ^ ACC) * 2 >> 3 & 2;

=== Current state of architecture ===
--- Items:
ACC, flags: 0x13, class: item.ConObject, val: 00000001

... výstup zkrácen ...
```

- Úspěšnost: PASS

D.3 Vytváření prvků

- Jméno: Vytváření prvků pomocí struktury `bits[]`.
- Popis: Pomocí příkazu `array` jazyka pro popis architektury budou vytvořeny dvě skupiny prvků s přístupem pomocí indexu s různými rozsahy uvedenými ve strukturách `bits[]`. Bitová šířka nově vzniklých prvků by měla být rovna rozdílu počáteční a koncové hodnoty rozsahu ve struktuře `bits[]`.
- Vstup: Prázdná architektura, příkazy: `array<bits[0:7]> mem1(3);` a `array<bits[1:8]> mem2(3);`
- Očekávaný výstup: Ve výpisu všech prvků v sekci `Items` existují dva prvky třídy `ConArray` s názvem `mem1`, resp. `mem2` o bitové šířce 8bitů a velikosti 3 prvky.

```
... výstup zkrácen ...

AIMe> array<bits[0:7]> mem1(3);
DEBUG: Creating array: 'mem1' size: '3'

=== Current state of architecture ===
--- Items:
mem1, flags: 0x5, class: item.ConArray:
  #mem1#0, flags: 0x2, class: item.ConObject, val: 01010010
  #mem1#1, flags: 0x2, class: item.ConObject, val: 01100111
  #mem1#2, flags: 0x2, class: item.ConObject, val: 10100110

--- Collections:
--- Functions:
--- Pointers:

AIMe> array<bits[1:8]> mem2(3);
DEBUG: Creating array: 'mem2' size: '3'

=== Current state of architecture ===
--- Items:
mem1, flags: 0x5, class: item.ConArray:
  #mem1#0, flags: 0x2, class: item.ConObject, val: 01010010
  #mem1#1, flags: 0x2, class: item.ConObject, val: 01100111
  #mem1#2, flags: 0x2, class: item.ConObject, val: 10100110

mem2, flags: 0x5, class: item.ConArray:
  #mem2#0, flags: 0x2, class: item.ConObject, val: 10110010
  #mem2#1, flags: 0x2, class: item.ConObject, val: 10000000
  #mem2#2, flags: 0x2, class: item.ConObject, val: 11010110

... výstup zkrácen ...
```

- Úspěšnost: PASS

Dodatek E

Obsah CD

Kořenový adresář CD obsahuje čtyři podadresáře:

- `install` – instalace interpretu jazyka Python a jeho grafické nadstavby wxPython pro operační systémy Windows 64/32bit
- `src` – zdrojové kódy aplikace související s grafickým rozhraním, vstupní bod programu (soubor `aime.py`), obsahuje další podadresáře:
  - `arch` – zdrojové kódy aplikace související s interpretem architektury
  - `instr` – zdrojové kódy aplikace související s interpretem instrukcí
  - `design` – soubory umožňující editaci grafického uživatelského rozhraní v programu wxGlade
- `doc` – text této technické zprávy, programová dokumentace a soubor s gramatikou jazyka pro popis architektury
- `examples` – soubory vhodné pro demostraci možností aplikace