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

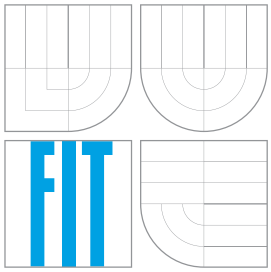
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. LUKÁŠ CHARVÁT

BRNO 2011

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

A CPU EMULATOR FOR COURSE OF ASSEMBLY LANGUAGES

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. LUKÁŠ CHARVÁT
AUTHOR

VEDOUCÍ PRÁCE — Ing. ALEŠ SMRČKA, Ph.D.
SUPERVISOR

BRNO 2011

Abstrakt

Práce řeší tvorbu emulátoru počítačové architektury a její instrukční sady se záměrem pro použití při výuce asemblerů. Zatímco většina dnešních emulátorů je závislá na specifické architektuře, tato práce popisuje přístup, jak vytvořit emulátor vhodný pro použití ve výuce a pro snadnější pochopení asemblerů. Emulátor se neomezuje pouze na jeden typ procesoru, ale umožňuje uživatelům jednoduše definovat vlastní architektury spolu s jejich instrukčními sadami za účelem možnosti provádět nad nimi operace a především názorně zobrazovat aktuální stav.

Abstract

The master thesis discusses the design of an emulator of a CPU architecture instruction set aimed at assembly languages course. While most of nowadays emulators are architecture specific, the emulator proposed in master thesis aims at education and better understanding of assembly languages. The emulator is not limited to a single CPU, but it easily allows defining a purpose-specific architecture and instruction set in order to perform operations upon it and to display its current state.

Klíčová slova

Emulace, asembler, instrukční sada.

Keywords

Emulation, assembly language, instruction set.

Citace

Lukáš Charvát: Emulace CPU pro výuku asemblerů, diplomová práce, Brno, FIT VUT v Brně, 2011

Emulace CPU pro výuku asemblerů

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Aleše Smrčky, Ph.D.

.....
Lukáš Charvát
24. května 2011

Poděkování

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

Obsah

1 Úvod **3**

2 Přehled emulátorů **4**
2.1 MAME - Multiple Arcade Machine Emulator 4
2.2 QEMU 5
2.3 PearPC 5
2.4 Aspectrum - Another Spectrum emulator 6

3 Přehled vybraných procesorů **7**
3.1 Intel x86 7
3.2 ARM11 8
3.3 Freescale MC9S08JM60 9
3.4 PowerPC 10
3.5 Java Virtual Machine 10

4 Specifikace požadavků emulátoru **12**
4.1 Požadavky na interpret architektury 12
4.2 Požadavky na interpret instrukcí 13
4.3 Požadavky na grafické uživatelské rozhraní 13

5 Návrh řešení **15**
5.1 Reprezentace a popis architektury 16
5.2 Reprezentace a popis instrukční sady 21
5.3 Reprezentace a popis grafického výstupu 24
5.4 Návrh řídících struktur emulátoru 27
5.5 Grafické uživatelské rozhraní 30
5.6 Rozdělení návrhových tříd do vrstev 33
5.7 Princip činnosti emulátoru 34

6 Implemetace emulátoru **42**
6.1 Syntaktický analyzátor 42
6.2 Grafické uživatelské rozhraní 42
6.3 Rozhraní příkazové řádky 44
6.4 Testování emulátoru 45
6.4.1 Jednotkové testy 45
6.4.2 Integrační testy 46

7 Závěr **47**
7.1 Možné pokračování projektu 47

A Navržené gramatiky **50**
A.1 Gramatika jazyka pro popis architektury 50
A.2 Gramatika jazyka pro popis instrukční sady 51
A.3 Gramatika jazyka pro popis grafického zobrazení 52

B Instalace a příklad použití **53**
B.1 Instalace 53
B.2 Příklad použití 53
B.2.1 Spuštění aplikace 53
B.2.2 Vytvoření projektu 53
B.2.3 Hlavní okno programu 54
B.2.4 Možnosti zobrazení a správa popisu architektury 54
B.2.5 Uložení a ukončení činnosti 54

C Obsah CD **56**

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 [22]. Své uplatnění nachází v případech, kdy nástup nových technologií (např. v hardware) již neumožňuje spustit starší aplikace. Použitím emulátoru je možné ušetřit nemalé náklady, které by mohly být spojené s převodem dané aplikace na technologii novou. Postup ovšem vyžaduje dokonalou znalost emulované architektury a častým problémem bývá i pomalejší odezva aplikace běžící v emulátoru. Dále emulace nachází své uplatnění v oboru hardware/software codesign, který se zaměřuje na souběžný vývoj aplikací a architektur zpravidla pro zabudované systémy. V tomto případě, kdy se klade za cíl získávat nové poznatky o vytvářeném systému, je rozšířen pojem simulace. Simulace cílového, ještě fyzicky neexistujícího, hardware v tomto případě umožňuje spouštění cílových aplikací zároveň s vývojem hardware. Tomuto problému se věnuje např. projekt Lissom [19]. Emulace může nalézt uplatnění i při ladění software, a to až na úrovni strojového kódu.

Cílem diplomové práce je využít poslední uvedené vlastnosti a vytvořit aplikaci schopnou názorně zobrazovat stav procesoru emulovaného na úrovni jazyka symbolických instrukcí spolu se schopností snadno definovat strukturu architektury procesoru a jeho instrukční sadu. Výsledek práce by měl sloužit k nasazení zejména ve výukovém, ale i experimentálním prostředí.

V druhé kapitole jsou shrnuty některé ze současných nástrojů sloužících k emulaci architektur. Náplní kapitoly třetí je rozbor některých ze současných procesorů. Čtvrtá kapitola definuje požadavky na jednotlivé části nově vyvíjeného emulátoru. V páté kapitole je proveden návrh řešení s ohledem na specifikaci uvedenou v kapitole čtvrté. V poslední kapitole jsou pak shrnuty dosažené výsledky.

Kapitola 2

Přehled emulátorů

Následující odstavce se zabývají vlastnostmi vybraných současných emulátorů. Záměrně jsou voleny emulátory s neproprietárním kódem a s dostupnou dokumentací. V podobě přehledu jsou shrnuty vlastnosti, výhody a nevýhody jednotlivých emulátorů s ohledem na téma práce.

2.1 MAME - Multiple Arcade Machine Emulator

MAME je emulátorem vyvinutým s cílem vytvořit hardwarové podmínky starých arkádových herních systémů. Záměrem vývoje je zachování starých počítačových her dřív, než budou ztraceny nebo zapomenuty. Emulátor se snaží o kompletní napodobení vnitřní funkčnosti arkádových herních strojů, možnost hrát staré hry je autory považována za „vedlejší efekt“ [18].

MAME umožňuje uživateli sestavit emulovanou architekturu podle jeho potřeb na velice nízké úrovni abstrakce. Nejdříve je nutné nadefinovat chování prvků architektury (např. CPU, DMA, řadič přerušení, paměť, grafický akcelerátor, vstupně/výstupní zařízení). K usnadnění této činnosti MAME nabízí řadu předdefinovaných maker. Takto definované prvky jsou pak zaregistrovány do simulační části emulátoru, jenž se stará o jejich paralelní emulaci.

Při emulaci procesoru je zpravidla vytvořena struktura reprezentující fyzické části procesoru – zejména registry. Popis chování jednotlivých instrukcí procesoru se provádí mapováním jednotlivých operačních kódů procesoru na funkce jazyka C, které operují nad dříve definovanou fyzickou reprezentací procesoru. Postup je demonstrován na následujícím úseku kódu.

```
OP(op, 3c)
{
    z80->A = INC(z80, z80->A);
}
/* ... výstup zkrácen ... */
inline unsigned int INC(z80_state *z80, unsigned int value)
{
    unsigned int res = value + 1;
    z80->F = (z80->F & CF) | SZHV_inc[res];
    return res;
}
```

V první části makro OP mapuje operační kód `3C` reprezentující instrukci „inkrementuj registr A“ na kód jazyka C. Struktura z80 představuje architekturu (přesněji stav) procesoru Z80. Mapování využívá pomocnou funkci INC (druhá část kódu) zapouzdřující operaci inkrementace, která předpokládá stav architektury a inkrementovanou hodnotu jako argumenty na svém vstupu. Její kód nejdříve inkrementuje předanou hodnotu, poté odpovídajícím způsobem nastaví příznakový registr F a navrátí inkrementovanou hodnotu. Vzhledem ke své obecnosti může být funkce znovupoužita pro inkrementaci ostatních registrů.

Podobným způsobem řeší emulaci i jiné emulátory, u nichž je hlavním cílem dosáhnout co možná nejvyšší rychlosti zpracování emulovaného kódu.

Výhody:

- rychlost, přenositelnost (jazyk C++),
- implementováno rozhraní pro přidávání nových architektur herních konzolí.

Nevýhody:

- nutná hlubší studie emulátoru a rozhraní, na jehož základě se nové komponenty vytvářejí,
- velice nízká úroveň abstrakce,
- časová náročnost na pochopení souvislostí a na studium dokumentace.

2.2 QEMU

QEMU umí emulovat architektury x86, Sparc, ARM a PowerPC. Může pracovat ve dvou režimech. Plná emulace umožňuje přímo spustit virtuální počítač a provozovat na něm libovolný operační systém. Z pohledu běžícího software je pak vidět pouze virtuální hardware emulovaný pomocí QEMU. Uživatelská emulace umožňuje spouštět aplikace určené pro jiný procesor bez abstraktní hardwarové vrstvy. Režim využívá dynamického překladu operačních kódů emulovaného architektury na kódy platformy, na níž emulátor běží.

Výhody:

- přenositelnost (jazyk C++).

Nevýhody:

- nemožnost měnit emulovanou architekturu.

2.3 PearPC

PearPC [15] je emulátor počítače typu PowerPC umožňující běh aplikací i (operačních systémů) pro tuto platformu. V současném stádiu vývoje však umožňuje pouze běh operačního systému MacOS. Emulátor pracuje na podobném principu jako zástupce předcházející, tj. na bázi přepisu jednotlivých operačních kódů procesoru PowerPC na funkce jazyka C/C++.

Výhody:

- přenositelnost (jazyk C++),
- architektura je částečně modifikovatelná pomocí nastavení v konfiguračních souborech.

Nevýhody:

- pochopení možností nastavení vyžaduje delší studium dokumentace,
- časté neočekávané ukončení aplikace.

2.4 Aspectrum - Another Spectrum emulator

Open-source projekt Aspectrum [8] je vyvíjený za účelem vytvoření emulátoru počítače ZX Spectrum, schopného pracovat téměř na všech současných architekturách. Tato přenositelnost je zaručena implementací emulátoru v čistém kódu jazyka C. Emulátor po kompilaci pracuje v systému MS-DOS, Windows, Linux a MacOS X. Postup emulace zůstává zachován stejný i v tomto případě pouze s rozdílem v použití čistého jazyka C, čímž by měla být podle autorů emulátoru zaručena ještě vyšší rychlost emulace než v případě použití jiných jazyků.

Výhody:

- přenositelnost (jazyk C),
- rychlost.

Nevýhody:

- neimplementuje veškeré operační kódy, což vede k občasným pádům aplikace.

Kapitola 3

Přehled vybraných procesorů

V jednotlivých následujících sekcích jsou shrnuty některé ze současných procesorů. U každého je proveden rozbor jeho architektury, instrukční sada a způsob adresování paměti. Výběr je stručný, nicméně zahrnuje architektury jednoduché (Freescale rodiny HCS08) i virtuální (Java Virtual Machine).

3.1 Intel x86

Současná instrukční sada architektury, která stála na počátku počítačové revoluce, se vyznačuje proměnnou délkou instrukce a je v podstatě souborem rozšíření aplikovaných na jednoduchou osmibitovou architekturu 8080/8085. Prvním z procesorů této řady je Intel 8086 [13]. Jedná se o 16bitový procesor vyráběný od roku 1978 s následující architekturou [16]:

- 8 registrů pro všeobecné využití AX, BX, CX, DX, SI, DI, SP, BP – 16bitové,
- segmentové registry CS, DS, SS, ES – 16bitové,
- registr příznaků FLAGS – 16bitový,
- programový čítač IP – 16bitový.

S příchodem procesoru 80386 byla architektura rozšířena na 32bitů (se zachováním zpětné kompatibility):

- 8 registrů pro všeobecné využití EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP – 32bitové,
- segmentové registry CS, DS, SS, ES, FS, GS – 16bitové,
- registr příznaků EFLAGS – 32bitový,
- programový čítač EIP – 32bitový.

V roce 2003 firma AMD rozšířila instrukční sadu na 64bitů (opět se zachováním zpětné kompatibility):

- 16 registrů pro všeobecné využití RAX, RBX, RCX, RDX, RSI, RDI, RSP, RBP, R8-R15 – 64bitové,

- segmentové registry CS, DS, SS, ES, FS, GS – 16bitové,
- registr příznaků RFLAGS – 64bitový,
- programový čítač RIP – 64bitový.

Instrukce vykonávající aritmetické a logické operace mohou volně používat registrovou sadu procesoru. Některé jsou ovšem určeny pouze pro práci s určitým registrem. Registr AX reprezentuje střádač. BX se používá především jako ukazatel na data v některých adresovacích módech. CX je čítač, jenž je využíván instrukcemi reprezentujícími cykly. DX je datový registr a lze jej používat pro adresování vstupně-výstupních portů.

Aby mohl původní 16bitový mikroprocesor používat 20bitové adresy, je každá adresa rozdělena na dvě části – segment a offset, které tvoří tzv. logickou adresu. Segmentová část adresy musí být vždy uložena v některém ze segmentových registrů, offsetová část může být v některém z registrů BX, SI, DI, SP, BP nebo IP anebo může být zadána jako konstanta v programu. Z logické adresy se fyzická vytváří tak, že se nejprve posune hodnota segmentu o čtyři bity vlevo (odpovídá násobení šestnácti) a k takto vzniklému číslu se přičte offset. Tím vznikne 20bitová adresa ukazující na konkrétní místo v paměti. Důsledkem adresování je paměť rozdělena na jednotlivé bloky o velikosti 64 kB (segmenty). Segmentová část adresy po vynásobení šestnácti ukazuje na začátek segmentu a offset je pozice v segmentu vzhledem k jeho začátku. V současné době, kdy jsou k dispozici 32 a 64bitové procesory, segmentace ztrácí svůj původní význam. Registry v těchto procesorech jsou již schopné adresovat 2^{32} B resp. 2^{64} B. paměti a lze tedy použít lineární adresování [17].

3.2 ARM11

Architektura ARM [2] umožňuje široké spektrum využití. Jedná se o vedoucí architekturu v mnoha segmentech trhu. Představuje jádro většiny současně vyráběných chytrých mobilních telefonů. Je také široce využívána koncovými uživateli v domácnostech a vestavěných zařízeních. Jednoduchost procesorů ARM vede k velmi malým výsledným implementacím, které mohou mít velice nízkou spotřebu energie. Instrukční sada na bázi RISC procesorů ARM zahrnuje:

- velký soubor registrů,
- operace load a store pro přístup do paměti a žádné přímé operace s hodnotami v paměti,
- jednoduché adresování, při kterém jsou všechny adresy pro instrukce typu load a store získávány pouze z obsahu instrukčních polí,
- zpětnou kompatibilitu se všemi generacemi ARM procesorů.

Dále ARM architektura nabízí některé další klíčové prvky pro zvýšení hustoty výsledného kódu a zvýšení výkonu. Tato vylepšení (popsaná níže) umožňují procesorům ARM dosahovat vyrovnaného poměru výkonu, velikosti kódu, spotřeby a prostoru na čipu. Programovací model procesoru disponuje šestnácti všeobecnými registry a jedním (dvěma v privilegovaných módech) stavovými registry přístupnými kdykoliv za běhu procesoru.

Programovací model (příklad procesoru ARM1176JZ-S [4]):

- 13 registrů pro obecné využití R0-R12 – 32bitové,

- ukazatel zásobníku (stack pointer) R13 – 32bitový,
- registr odkazů (link register) R14 – 32bitový,
- programový čítač (program counter) R15 – 32bitový,
- stavový registr (current program status register) CPSR – 32bitový.

Registry R0-R12 určené pro všeobecné použití slouží k uložení hodnot nebo adres do paměti. Registers R13, R14, R15 mají dále uvedené specifické funkce. Registr R13 je používán jako ukazatel zásobníku, který je přepínaný (angl. banked) pro módy zpracování chyb. To znamená, že obsluha chyby může využívat zásobník rozdílný od toho, při jehož používání se objevila chyba. V mnoha případech lze registr R13 použít i jako všeobecný. Registr odkazů R14 se používá při práci s podprogramy. Do registru R14 se ukládá návratová adresa z podprogramu při volání instrukcí BL or BLX (Branch with Link). V ostatních případech může být registr opět použit jako všeobecný. Registr R15 uchovává aktuální hodnotu programového čítače. Procesory ARM pracují s organizací paměti Little-Endian i Big-Endian. Od procesorů architektury ARMv3 je podporován 32bitový lineární adresový prostor (tj. 2^{32} B paměti) [4].

Procesor disponuje instrukcemi typickými pro architekturu RISC. Navíc každá instrukce obsahuje podmínku, která rozhoduje, zda bude daná instrukce procesorem vykonána, či nikoliv. Tato vlastnost redukuje blokování instrukční pipeline a umožňuje vytvářet souvislý kód bez velkého množství skoků. Hodnota druhého operandu (pokud je uložen v registru) může být logicky posunuta pomocí integrovaného válcového posunovače (angl. barrel shifter) bez dopadu na rychlost provedení instrukce (více viz. [3]). Pro optimalizaci poměru cena/výkon a hustoty kódu je od procesorů architektury ARMv4T [1] uvedené na trh roce 2001 k dispozici i instrukční rozšíření Thumb (u novějších procesorů i Thumb-2). Jedná se o 16bitové instrukce pokrývající většinu nejpoužívanějších 32bitových instrukcí (ovšem bez možnosti nastavovat podmínku pro vykonání instrukce). V průběhu vykonávání programu jsou tyto instrukce transparentně dekódovány zpět na 32bitové. Výsledkem je zvýšená hustota výsledného kódu a mírný nárůst výkonnosti [3].

3.3 Freescale MC9S08JM60

Procesory série MC9S08JM60 jsou členy cenově dostupné a výkonné rodiny HCS08 8bitových mikrokontrolérů. Všechny procesory v této rodině používají vylepšené jádro HCS08 vycházející z jádra HC08 a jsou dostupné s velkým množstvím modulů, velikostí pamětí a typů paměti. I když tato řada přináší nové instrukce, je stále plně zpětně kompatibilní s kódem napsaným pro jejího předchůdce. Instrukční sada je přizpůsobena střadačové architektuře procesoru. Instrukce mají variabilní délku provádění. V mikrokontrolerech HCS08 celá paměť spolu se stavovými, řídícími a vstupně-výstupními porty sdílejí jeden 64 kB lineární adresový prostor, a tak 16bitová adresa může unikátně identifikovat jakoukoliv pozici v paměti. Toto uspořádání také znamená, že instrukce, které modifikují proměnné v RAM, mohou být použity i k přístupu k vstupně výstupním portům, řídícím registrům nebo nevolatilnímu programovému prostoru. [9].

Programovací model:

- akumulátor (A) – 8bitový,
- indexový registr (index register) – 16bitový, dělený na dvě 8bitové části H:X,

- ukazatel zásobníku (stack pointer) SP – 16bitový,
- programový čítač (program counter) PC – 16bitový,
- registr podmínek (condition code register) CCR – 8bitový.

Akumulátor A je určen k všeobecnému použití. Mikrokontroler jej využívá pro uložení operandů a výsledků aritmetických a jiných dalších operací. Indexový registr SP se používá k adresování 64 kB paměti. Tvoří ho spojení dvou osmibitových registrů H a X. Ukazatel zásobníku SP ukazuje na první volnou pozici na zásobníku. Vybrané instrukce mohou využít ukazatele zásobníku jako dalšího indexového registru. Programový čítač PC obsahuje adresu příští instrukce nebo operandu. Standardně se PC automaticky inkrementuje, při instrukcích skoků, volání podprogramu, nebo obsluze přerušení PC je nastaven podle kontextu na adresu odlišnou. Po resetu je PC naplněn resetovacím vektorem uloženém v paměti na adrese 0xFFFE a 0xFFFF. Registr podmínek CCR obsahuje masku přerušení a pět příznaků indikujících výsledek poslední provedené instrukce [10].

3.4 PowerPC

Podle [7] je PowerPC pokračováním superskalární architektury Power firmy IBM. Je společným úsilím firem IBM, Apple a Motorola zjednodušit architekturu jejího předchůdce, rozšířit instrukční slovo na 64 bitů a umístit celý systém na jeden čip (původně 9 čipů). Přestože PowerPC architektura byla původně určena pro využití v osobních počítačích (počítače Macintosh), nakonec se stala oblíbenou ve vestavěných, na výkon zaměřených zařízeních. Své využití našla např. v herních konzolích (Microsoft Xbox 360, Sony Playstation 3) a jiných vestavěných zařízeních.

Architektura:

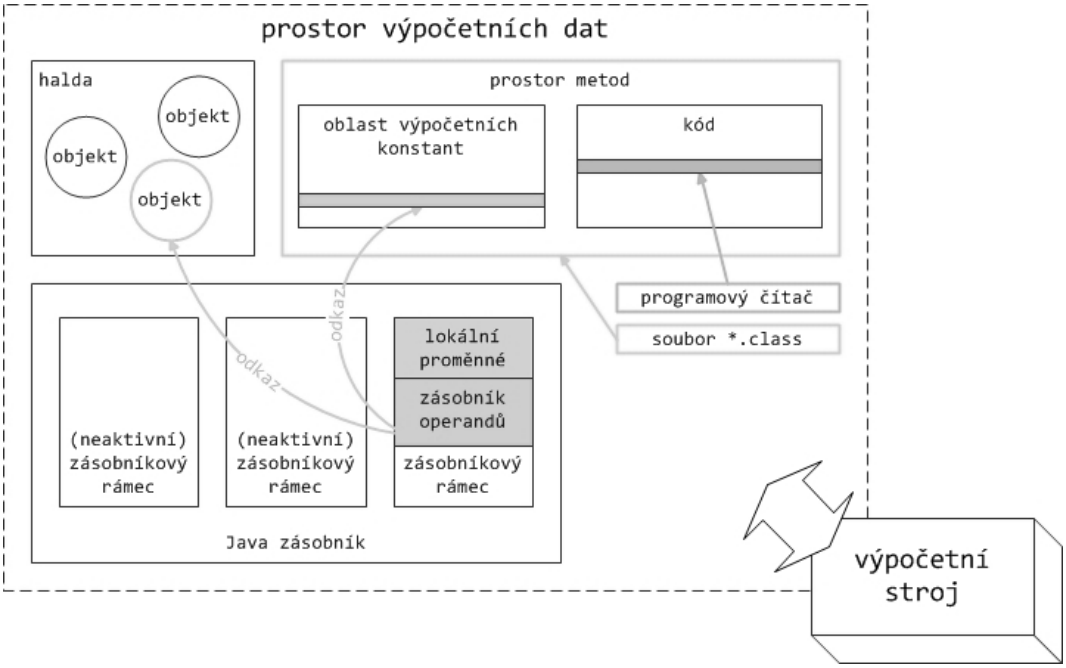
- registry pro všeobecné použití (GPR0-GPR31) – 64bitové,
- registr podmínek – 32bitový,
- registr odkazů – 64bitový,
- programový čítač – 64bitový.

Procesor disponuje instrukcemi typickými pro architekturu RISC. U většiny aritmetických a logických instrukcí je navíc možné definovat, zda nastavují registr podmínek. Procesor PowerPC pracuje standardně s organizací paměti Big-Endian, ale nabízí i možnost přechodu do režimu Little-Endian. PowerPC pracuje se 64bitovým lineárním adresovým prostorem (tj. 2^{64} B paměti).

3.5 Java Virtual Machine

Java Virtual Machine (JVM) [20] je virtuální výpočetní stroj 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.

Jelikož se jedná o virtuální architekturu, její specifikace pouze nastiňuje prvky abstraktního výpočetního stroje. Správná implementace JVM požaduje pouze schopnost číst



Obrázek 3.1: Schéma architektury JVM.

soubory typu `*.class` a vykonávat operace, jež jsou v nich uvedené. Implementační detaily nejsou součástí specifikace Java virtual machine a vyžadují nutnou tvořivost tvůrců konkrétního virtuálního stroje.

Každá instance JVM (zobrazená na obrázku 3.1) obsahuje paměťové prostory nazývané oblast metod (angl. method area) a halda (angl. heap). Jakmile virtuální stroj načte soubor typu `*.class`, zpracuje z něj data v binární formě a uloží jej do prostoru oblasti metod. Při běhu programu virtuální stroj ukládá jednotlivé programové instance na haldu.

Každé výpočetní vlákno, které vznikne v rámci instance Java virtual machine, dostane přiřazen programový čítač (registr PC) a zásobník (Java stack). V průběhu vykonávání kódu pak programový čítač vždy odkazuje na instrukci, která má být vykonána jako další. Zásobník slouží k uchování stavu aktuálních volání metod. Stav metody zahrnuje parametry, se kterými byla funkce zavolána (angl. local variables), a mezivýpočty (uložené na operačním zásobníku, angl. operand stack).

Zásobník se skládá z tzv. zásobníkových rámců (angl. stack frames). Zásobníkový rámec obsahuje vždy stav jednoho volání metody. Ve chvíli volání metody JVM vytvoří nový zásobníkový rámec na zásobníku. Jakmile je metoda dokončena, virtuální stroj odstraní její zásobníkový rámec.

JVM neobsahuje žádné registry pro uložení hodnot mezivýpočtů. Místo toho instrukční sada využívá část aktuálního zásobníkového rámce nazývanou operační zásobník jako místo pro uložení mezivýpočtů. Tento koncept byl zvolen z důvodů zachování kompaktní instrukční sady JVM a možnosti nasadit Java architekturu na výpočetní zařízení s málo registry pro obecné využití.

Kapitola 4

Specifikace požadavků emulátoru

Analýzou zástupců současných emulátorů bylo zjištěno, že většina z nich nabízí pouze omezenou možnost měnit emulovanou architekturu nebo její instrukční sadu. Práce s nimi bývá také příliš složitá na to, aby je bylo možné po krátké chvíli začít používat. To eliminuje možnost jejich použití jako výukového prostředku. Z toho vyplývají základní požadavky práce (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 architektur počítačů nebo programování v asemblerech.

Na rozdíl od stávajících nástrojů nově vyvíjený emulátor předpokládá užívání dvěma typy uživatelů. Prvním typem jsou uživatelé definující architekturu a instrukční sadu emulovaného procesoru a způsob zobrazení architektury (např. vyučující). Typem druhým jsou uživatelé, kteří emulují programy a sledují vnitřní stav procesoru (např. studenti). Požadavky lze shrnout a tématicky rozdělit do tří následujících sekcí. První se věnuje požadavkům na interpret architektury, následuje druhá s definicí nutných schopností interpretu instrukcí a závěrečná třetí s požadavky na uživatelské rozhraní.

4.1 Požadavky na interpret architektury

Z nutnosti reprezentovat emulovanou architekturu a její stav (definovaný stavem prvků architektury) vyplývají následující požadavky na interpret architektury:

- schopnost dynamicky vytvářet typické prvky a struktury počítačových architektur,
- schopnost dynamicky rušit prvky architektury – dynamičnost vytváření a rušení je vyžadována virtuálními architekturami,
- prvky vznikající v emulovaném systému nesmí být ovlivněny architekturami stávajícími (např. bitovou šířkou),

- schopnost modelovat fyzické závislosti mezi prvky architektury – zejména takové, při kterých změna jednoho prvku implikuje změnu stavu v prvku jiném.

Fyzickými závislostmi v posledním z výše uvedených požadavků jsou míněny případy, v nichž jeden prvek představuje část prvku jiného, nebo naopak ty, ve kterých je větší prvek složen z menších nezávislých částí.

4.2 Požadavky na interpret instrukcí

Interpret instrukcí musí především poskytovat základní elementární operace umožňující popis chování jednotlivých instrukcí emulovaného procesoru na instrukční úrovni:

- možnost vytvářet pomocné proměnné pro potřeby interpretu instrukcí,
- provádět přesuny hodnot mezi prvky architektury a proměnnými,
- matematické operace (sčítání, odčítání, násobení, dělení, dělení modulo),
- logické operace (and, or, xor, not),
- bitové posuvy,
- operace porovnání (menší než, větší než, je rovno),

Výčet operací není záměrně rozsáhlý, aby bylo možné se s ním poměrně rychle seznámit. Pro popis chování instrukce je dále nezbytné realizovat mechanismus zajišťující možnost definovat platné operandy a jejich případné textové přeformátování z platformě specifického zápisu (jako bývá např. adresace operační paměti) do podoby, která může být dále zpracována v rámci vyvíjeného emulátoru. Dalšími požadavky na interpret instrukcí jsou:

- spojování sledu operací do procedur – často používané sledy operací by měly být uspořadatelné do celků, aby je nebylo nutné pokaždé vypisovat zvlášť,
- větvení toku výpočtu pomocí podmínek,
- provádět sled operací v cyklu (s podmínkou i bez ní).

Uvedené požadavky z větší části pokrývají operace používané v pseudokódu pro popis chování instrukcí, který můžeme nalézt v referenčních příručkách dodávaných k procesorům (viz např. [16], [17]).

4.3 Požadavky na grafické uživatelské rozhraní

Na uživatelské rozhraní jsou z důvodu zaměření práce kladeny odlišné nároky, než jak je tomu u ostatních emulátorů. Vyžaduje se sloučení editace kódu spolu se zobrazením aktuálního stavu architektury. Z pohledu práce s kódem v jazyku symbolických instrukcí se jedná zejména o možnost:

- psaní kódu v jazyku symbolických instrukcí emulované architektury,
- krokování kódu (angl. debugging),

- tvorby bodů přerušení (angl. breakpoints).

Z hlediska zobrazení architektury musí výsledná aplikace umožňovat:

- zobrazení aktuálního stavu emulované architektury,
- spojování prvků do skupin pro zobrazení – prvky společného typu nebo podílející se na podobné funkci je třeba mít možnost zobrazit společně,
- zobrazení vytvořených skupin prvků,
- definici logických vazeb mezi prvky – vhodné jak pro začátečníky zásluhou možnosti vidět provázání jednotlivých elementů architektury z hlediska sémantiky, tak i pro zkušené uživatele pro usnadnění orientace např. při ladění kódu,
- zobrazení logických vazeb mezi prvky.

Grafické uživatelské rozhraní by mělo respektovat výše zmíněné typy uživatelů a poskytovat odpovídající pohled na problematiku emulace. Dále by se nemělo vázat pouze na jedinou platformu, nýbrž by mělo umožňovat přenositelnost mezi různými druhy operačních systémů.

Kapitola 5

Návrh řešení

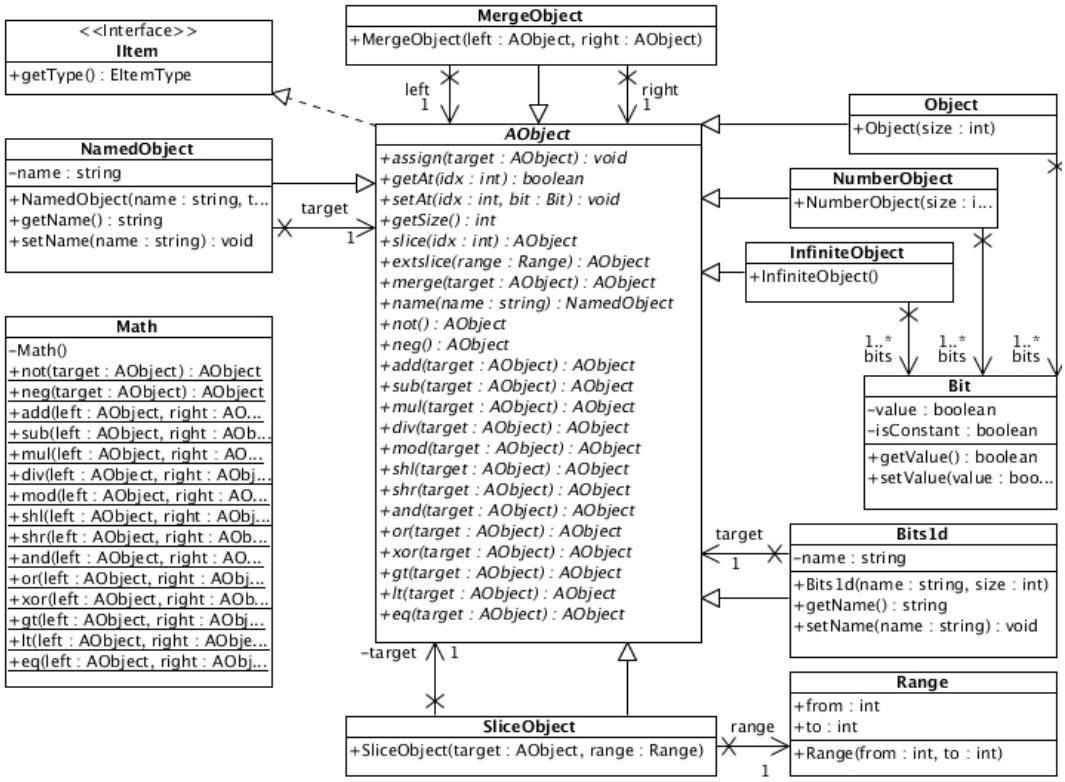
S ohledem na požadavky kladené na systém můžeme návrh projektu rozdělit na sedm částí. V první sekci této kapitoly je proveden návrh tříd nutných k reprezentaci architektury a definován jazyk pro její popis. Podobně je učiněno v sekci druhé, třetí a čtvrté pro instrukční sadu, grafický výstup, resp. řídící struktury. Pátá sekce se zabývá návrhem uživatelského rozhraní. Šestá sekce shrnuje provedený návrh do vrstev. Poslední část návrhu se zaměřuje na celkový princip činnosti emulátoru.

Emulátor bude napsán s využitím metod objektově orientovaného programování. Třídy obsažené v návrhu reflektují konstrukce, které mohou v emulátoru vzniknout. Návrh je proveden pomocí jazyka UML verze 2. Navržené jazyky jsou vytvořeny s důrazem na univerzálnost použití, přehlednost a podobnost s některými existujícími jazyky, aby se znalí uživatelé nemuseli přizpůsobovat zcela novému prostředí. Pro tento účel byly navrženy jazyky inspirované jazykem Python, nabízející stručné a jednoduché vyjadřování. Příkazy u navržených jazyků jsou rozděleny do čtyř skupin (je vždy uvedena spolu s definicí příkazu):

- *definiční* – vytvářející nové prvky,
- *rušící* – odstraňující prvky dříve definované prvky,
- *řídící* – modifikující programový tok,
- *výkonné* – provádějící operace nad již definovanými položkami.

Použité symboly pro popis navržených jazyků mají stejný význam jako u zápisu ENBF (Extended Backus-Naur Form) [14], která je prostředkem pro definici bezkontextových gramatik. Jedná se o rozšíření klasické Backus-Naurovy formy, z nichž mezi nejdůležitější patří:

- opakování **{** ... **}** – obsah ohraničený složenými závorkami se může libovolně opakovat a může být i vynechán,
- volba [...] – forma uzavřená závorkách může, ale nemusí být přítomna,
- povinné uvození (’ ... ’ nebo ” ... ”) terminálních symbolů,
- možnost vynechání ostrých závorek < ... > u nonterminálů.



Obrázek 5.1: Diagram tříd reprezentující jedinečné prvky architektury.

5.1 Reprezentace a popis architektury

Pro dosažení schopnosti emulovat co nejvíce počítačových architektur musí být emulátor schopen reprezentovat základní prvky vyskytující se ve většině z nich. Z provedené analýzy vyplývá, že tyto prvky lze rozdělit z pohledu identifikace na tři typy:

- *jedinečné*, v architektuře jednoznačně identifikované názvem a bitovou šířkou (např. registr),
- *kolekce* (pole) prvků s přístupem pomocí indexu – identifikované názvem, bitovou šířkou a počtem prvků v kolekci (např. paměť),
- *symbolické*, které se v architektuře fyzicky nevyskytují, ale jsou součástí jiného prvku (tzv. *řezy*, angl. slices).

Emulátor musí být schopen provádět operace čtení a přiřazení (zápis) nejen nad prvkem architektury jako celkem, ale i nad jejich částmi, tzv. řezy. Z typů prvků a operací nad nimi lze vytvořit následující návrh tříd reprezentující architekturu emulovaného procesoru.

Obrázky 5.1 a 5.2 formou UML diagramu tříd zachycují třídy představující prvky architektury a operace nad nimi.

`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 (potenciálně až nekonečným), musí být každý bit reprezentován zvlášť.

`AObject` Abstraktní třída pro veškeré jedinečné prvky. Definuje metody pro získání hodnoty bitu na požadovaném indexu (operace `getAt()`), jeho nastavení (operace `setAt()`) a pro přiřazení hodnoty z jiného objektu třídy `AObject` (operace `assign()`). Operace `slice()`, resp. `extslice()` vytvoří z aktuálního prvku řez na základě zadaného indexu, resp. indexů. Metoda `name()` slouží k pojmenování aktuálního prvku. Dále třída definuje základní matematické a logické operace.

`Object` Třída, jejíž objekty představují jedinečné prvky v architektuře. Je zděděná od třídy `AObject`.

`SliceObject` Třída představující symbolický prvek v architektuře vytvořený z jedinečného prvku pomocí řezu. Transformuje tak operace definované ve třídě `AObject` pouze na daný rozsah bitů prvku, jehož část symbolický prvek představuje.

`MergeObject` Třída představující symbolický prvek v architektuře vytvořený spojením dvou jedinečných prvků. Transformuje operace definované ve třídě `AObject` do rozsahu bitů prvků, jejichž spojení symbolický prvek představuje.

`InfiniteObject` Třída představující prvek architektury s nekonečným počtem bitů. Dědí od třídy `AObject`.

`NamedObject` Třída představuje pojmenované prvky v architektuře. Pojmenovaným prvkem může být libovolný objekt zděděný od třídy `AObject` kromě objektů třídy `Bits1d` a `NumberObject`.

`Bits1d` Třída představující pomocnou proměnnou pro výpočet. Objekty této třídy mají stejné vlastnosti jako třídy `Object` – nemohou být ovšem pojmenovány, a tím ani zobrazeny v uživatelském prostředí.

`NumberObject` Třída zapouzdřující anonymní sled bitů, který může být např. produktem mezivýpočtů. Tento objekt nemůže být zobrazen v uživatelském prostředí.

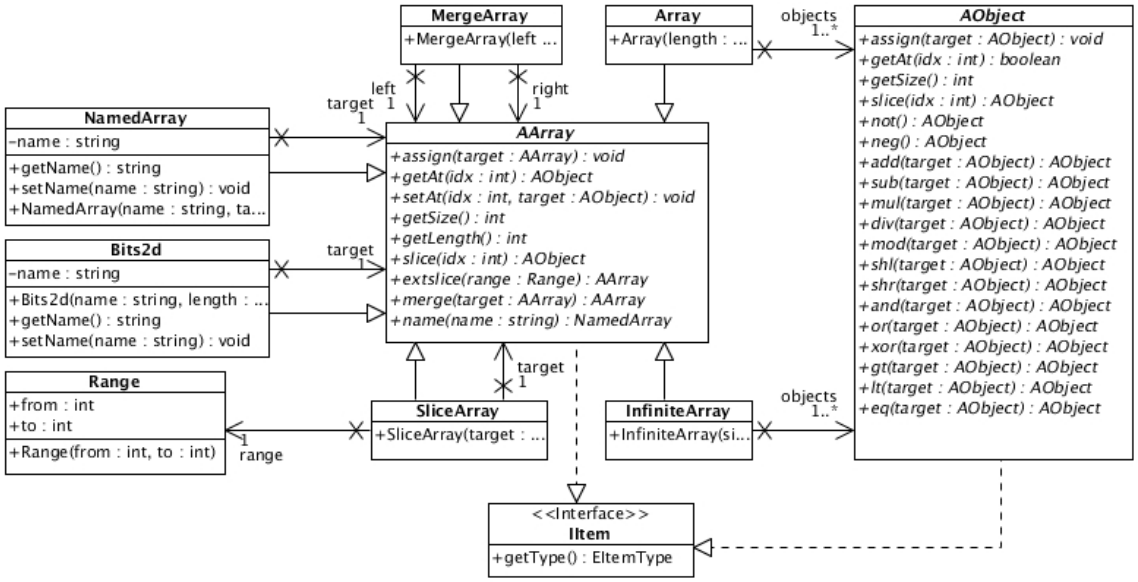
`Math` Třída implementující matematické operace nad objekty zděděnými od třídy `AObject`.

`AArray` Abstraktní třída představující kolekce prvků s přístupem pomocí indexu. Definuje metody pro získání prvku na požadovaném indexu (operace `getAt()`), jeho nastavení (operace `setAt()`) a pro přiřazení hodnoty z jiného objektu třídy `AArray` (operace `assign()`). Operace `slice()`, resp. `extslice()` vytvoří z aktuálního prvku řez na základě zadaného indexu, resp. indexů. Metoda `name()` slouží k pojmenování aktuálního prvku.

`Array` Třída, jejíž objekty představují kolekce prvků s přístupem pomocí indexu. Je zděděná od třídy `AArray`.

`SliceArray` Třída představující symbolický prvek v architektuře vytvořený z kolekce prvků pomocí řezu. Transformuje tak operace definované ve třídě `AArray` pouze na daný rozsah prvků kolekce, jejíž část symbolický prvek představuje.

`MergeArray` Třída představující symbolický prvek v architektuře vytvořený spojením dvou kolekcí prvků. Transformuje operace definované ve třídě `AArray` do rozsahu prvků kolekcí, jejichž spojení symbolický prvek představuje.



Obrázek 5.2: Diagram tříd reprezentující kolekce prvků s přístupem pomocí indexu.

`InfiniteArray` Třída představující kolekci s nekonečným počtem prvků. Dědí od třídy `AArray`.

`NamedArray` Třída představuje pojmenovanou kolekci prvků v architektuře (např. paměť). Pojmenovaným prvkem může být libovolný objekt zděděný od třídy `AArray` kromě objektů třídy `Bits2d`.

`Bits2d` Třída představující pomocnou kolekci prvků pro výpočet. Objekty této třídy mají stejné vlastnosti jako třídy `Array` – nemohou být ovšem pojmenovány, a tím ani zobrazeny v uživatelském prostředí.

Pro manipulaci s objekty těchto tříd byl také navržen následující jazyk. Vstupním bodem gramatiky jazyka je nonterminál `simple-stmt` představující jednoduchý (tedy nesložený) příkaz, který se dále rozvíjí v další příkazy jazyka pro popis zobrazení.

```
simple-stmt ::= object-def      # definiční
              | array-def       # definiční
              | symbol-def      # definiční
              | assignment      # výkonný
              | item-del        # rušící
```

Definice jedinečného prvku `object-def` se skládá z klíčového slova `object`, identifikátoru `ID` nově vznikajícího jedinečného prvku, znaku rovná se a vyjádření velikosti objektu v bitech `bits-1d` vyjádřeného klíčovým slovem `bits` a bitovým rozsahem `bits-range`. Lexém `ID` reprezentující identifikátor popisuje regulární výraz `[a-zA-Z_][a-zA-Z0-9_]*`.

```
object-def ::= 'object' ID '=' bits-1d
bits-1d ::= 'bits' bits-range
```

Definice rozsahu `bits-range` určená celým číslem, které je reprezentováno formou výrazu `expression` (viz. strana 20). Není-li výstup výrazu převeditelný na celé číslo, jedná se o sémantickou chybu. Zahrnuta je i možnost rozsahu nekonečného (klíčové slovo `inf`), který může být vyžadován některými abstraktně definovanými specifikacemi u virtuálních architektur.

```
bits-range ::= '[' expression ']'
             | '[' 'inf' ']'
```

Definice kolekce prvků s přístupem pomocí indexu `array-def` se bude skládat z klíčového slova `array`, identifikátoru `ID` a počtu prvků pole spolu s jejich velikostí v bitech `bits-2d`.

```
array-def ::= 'array' ID '=' bits-2d
bits-2d ::= 'bits' bits-range bits-range
```

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 s možnou aplikací operace řezu nebo spojení (i opakovaně), na který bude symbolický prvek odkazovat. V gramatice je taková konstrukce označena jako `target`.

```
symbol-def ::= 'symbol' ID '=' target
target ::= ID | slicing | merging
```

Řez `slicing` může být aplikován buď v jednoduché formě `simple-slicing`, při níž dochází k vyčlenění jednoho prvku na zadaném indexu (tj. v případě jedinečných prvků bude vybrán bit, u kolekcí prvek kolekce), nebo formě rozšířené `extended-slicing`, při níž bude proveden řez od počáteční po konečnou hodnotu. Indexace prvků začíná číslem nula. Hodnoty určující rozsah řezu mohou být libovolné výrazy jazyka `expression` (viz. strana 20) reprezentující celé číslo, v opačném případě se jedná o sémantickou chybu.

```
slicing ::= simple-slicing | extended-slicing
simple-slicing ::= target '[' expression ']'
extended-slicing ::= target '[' expression ':' expression ']'
```

Operace spojení `merging` vyjádřená operátorem tečka, vyžaduje jako své operandy dvě konstrukce typu `target`. Tyto prvky musejí být vždy stejného typu, tj. musí se jednat buď o dva jedinečné prvky, nebo o dvě kolekce s přístupem pomocí indexu. Jiná možnost je považována za sémantickou chybu. Z hlediska priority má operace spojení menší prioritu než operace řezu.

```
merging ::= target '.' target
```

Výkonným příkazem jazyka je přiřazení (angl. assignement) `assignment` skládající se z L-hodnoty a R-hodnoty. L-hodnota je tvořena obecným prvkem `target` definovaným výše. To umožňuje přiřazovat hodnotu nejen fyzickým prvkům jako celku, ale i jejich řezům či spojením. R-hodnotu pak tvoří obecný výraz `expression`. Přiřazovat lze prvky stejného typu (jedinečné prvky, či kolekce).

```
assignment ::= target '=' expression
```

Příkaz zrušení `item-del` je definován pomocí klíčového slova `del`, určení typu mazaného prvku `del-type` a identifikátoru rušeného prvku `ID`.

```
item-del ::= 'del' del-type ID
del-type ::= 'object' | 'array' | 'symbol' | 'var' | 'operand' | 'alias' |
           | 'procedure' | 'instruction' | 'group' | 'pointer'
```

Výrazem jazyka pro popis architektury `expression` může být obecný prvek `target`, celé číslo reprezentované ve formě lexému `INT`, které může být zapsáno v soustavě:

- desítkové – regulární výraz `[1-9][0-9]*`,
- šestnáctkové – regulární výraz `0x[0-9A-F]+`,
- osmičkové – regulární výraz `0[0-7]+`,
- dvojkové – regulární výraz `0b[01]+`.

Celé číslo o bitové šířce 8bitů také reprezentuje lexém `CHAR` zapsaný jako ASCII znak uzavřený mezi znaky apostrof. Výrazem také může být lexém `STRING`, který představuje sled ASCII znaků uzavřených mezi znaky uvozovka. Tato konstrukce je v emulátoru považována za kolekci prvků s přístupem pomocí indexu, ve které se na jednotlivých indexech nacházejí odpovídají znaky řetězce zaznamenané jako celá čísla o bitové šířce 8bitů.

Výrazem jazyka může být také dotaz na získání bitové šířky prvku skládající se z klíčového slova `sizeof` a v závorkách uzavřeném výrazu vracejícím prvek, na jehož šířku je dotazováno. Pro zjištění délky kolekce prvků s přístupem pomocí indexu slouží výraz jazyka skládající se z klíčového slova `len` a v závorkách uzavřeném výrazu vracejícím kolekci. Znaménkové rozšíření se skládá z klíčového slova `signext` a v závorkách uzavřeném výrazu, který bude rozšířen. Dále může být výrazem jazyka jiný výraz uzavřený v závorkách nebo výrazy, na které je aplikována některá matematická nebo logická operace. Aplikace bude implicitně pracovat v doplňkovém kódu. V případě použití jiného kódu je uživateli umožněno definovat si vlastní procedury (viz. strana 24) realizující výpočty v požadovaném kódování.

U jednotlivých možností přechodů je naznačena jejich priorita. Vzhledem k zaměření projektu pro výuku jsou logické operace and, xor, or a not definovány redundantně i pomocí textových ekvivalentů.

```
expression ::= target                       # pri = 9
             | INT                          # pri = 9
             | CHAR                         # pri = 9
             | STRING                       # pri = 9
             | sizeof '(' expression ')'    # pri = 9
             | len '('expression ')'        # pri = 9
             | signext '('expression ')'    # pri = 9
             | '(' expression ')'           # pri = 8
             | '-' expression               # pri = 7
             | '~' expression               # pri = 7
             | 'not' expression             # pri = 7
             | expression '/'   expression  # pri = 6
             | expression '%'   expression  # pri = 6
```

```
| expression '*'   expression   # pri = 6
| expression '+'   expression   # pri = 5
| expression '-'   expression   # pri = 5
| expression '<<'  expression   # pri = 4
| expression '>>'  expression   # pri = 4
| expression '&'   expression   # pri = 3
| expression 'and' expression   # pri = 3
| expression '^'   expression   # pri = 3
| expression 'xor' expression   # pri = 3
| expression '|'   expression   # pri = 2
| expression 'or'  expression   # pri = 2
| expression '>'   expression   # pri = 1
| expression '<'   expression   # pri = 1
| expression '=='  expression   # pri = 0
```

Cílem návrhu gramatiky jazyka pro popis architektury nebylo vytvořit jazyk disponující všemi prostředky současných programovacích jazyků, nýbrž jazyk schopný jednoduše a přehledně spravovat prvky emulované architektury.

5.2 Reprezentace a popis instrukční sady

Dalším z požadavků daných zaměřením práce je možnost popsat instrukční sadu emulovaného procesoru. Pro jeho splnění byly navrženy třídy reprezentující instrukční sadu a jazyk umožňující její popis. Návrh tříd je zachycen na obrázku 5.3.

`Procedure` Třída představující definovanou proceduru. Slouží pro uložení parametrů procedury a jejích jednotlivých řádků s kódem funkce.

`Instruction` Třída, jejíž objekty udržují popis chování instrukce. Slouží pro uložení argumentů a bloku kódu s popisem chování.

`Parameter` Třída představující parametr procedury nebo instrukce. Slouží pro uložení názvu parametru.

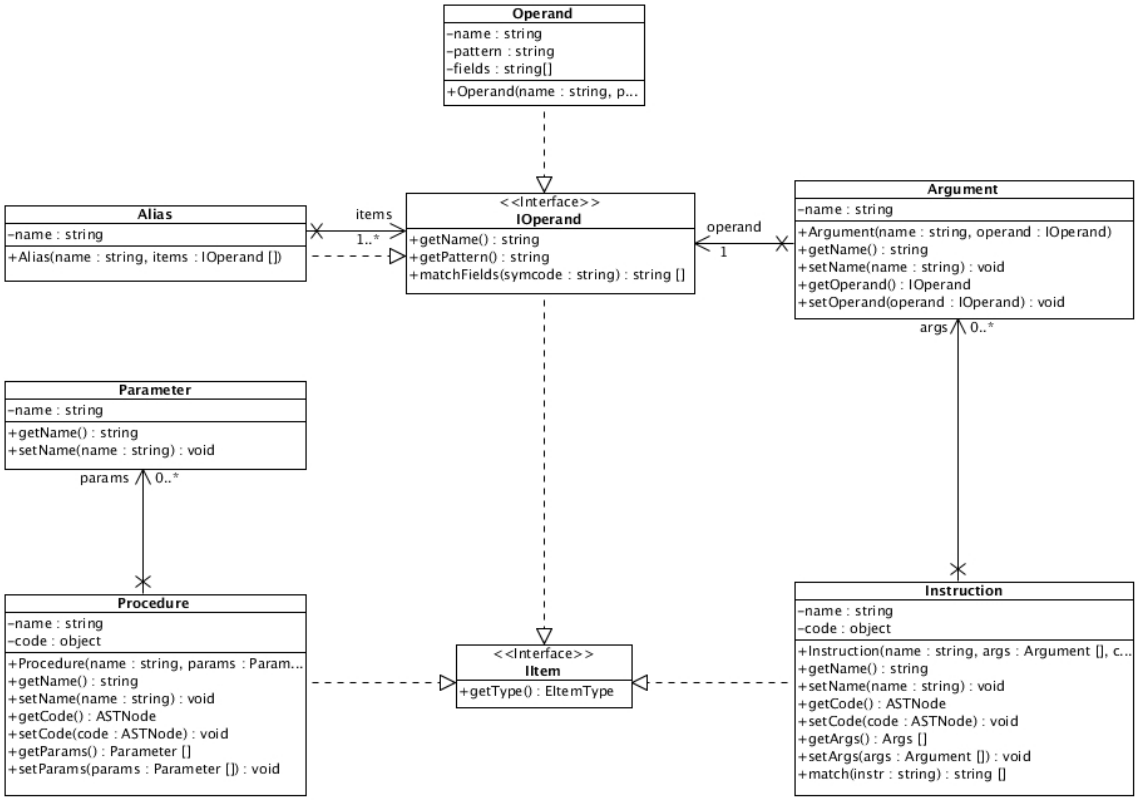
`Argument` Třída představující argument instrukce. Uchovává informace o názvu argumentu a operandu.

`IOperand` Rozhraní společné pro třídy `Operand` a `Alias`. Představuje obecný typ operandu instrukce.

`Operand` Třída sloužící k uchování informace o typu operandu instrukce.

`Alias` Třída umožňuje seskupovat podobné typy operandů nebo jiné objekty třídy alias pod jedním názvem.

Příkazem jazyka instrukční sady může být jednoduchý příkaz definovaný nonterminálem `simple-stmt` a `body-simple-stmt` nebo příkaz složený reprezentovaný nonterminály `compound-stmt` a `body-compound-stmt`. Účelem tohoto rozdělení je zabránění tvorby jistých, z gramatiky zřejmých programových konstrukcí v tělech funkcí a popisu chování instrukcí. U jednotlivých příkazů je uveden i jejich druh.



Obrázek 5.3: UML diagram tříd – třídy reprezentující instrukční sadu.

```
simple-stmt ::= variable-def              # definiční
              | exec-proc                 # výkonný
body-simple-stmt ::= operand-def          # definiční
                   | alias-def            # definiční
                   | exec-instr           # výkonný
compound-stmt ::= if-stmt                 # řídící
                | while-stmt              # řídící
body-compound-stmt ::= instruction-def    # definiční
                     | procedure-def      # definiční
```

Definice proměnné `variable-def` se skládá z klíčového slova `var`, identifikátoru `ID` nově vznikající proměnné, znaku rovná se a vyjádření velikosti proměnné v bitech `bits`. U vícerozměrných proměnných je podobně jako u kolekcí vyžadována i délka dimenze (viz. strana 18).

```
variable-def ::= 'var' ID '=' bits
bits ::= bits-1d | bits-2d
```

Definice operandu `operand-def` se skládá z klíčového slova `operand`, jeho identifikátoru `ID`, znaku rovná se a ve složených závorkách uzavřeného přepisovacího páru. Operandy se uplatňují při výběru popisu chování instrukce. Plní funkci filtru argumentů instrukce – instrukce může být interpretována pouze tehdy, pokud veškeré její argumenty odpovídají

danému vzoru `pattern` definovaného pomocí regulárního výrazu reprezentovaného lexémem `STRING`. Zároveň lze z hodnoty nalezeného vzoru odvodit hodnoty polí `field` operandu, které je možné následně použít v kódu popisu chování instrukce. Při odvození hodnoty lze používat možnosti poskytované operacemi s regulárními výrazy tak, jak byly zavedeny u jazyka Perl, tj. včetně vytváření skupin pomocí závorek a jejich zpřístupnění pomocí znaku zpětné lomítko následované číslem skupiny.

```
operand-def ::= 'operand' ID '=' '{' translate-pair '}'
translate-pair ::= pattern ':' '[' field { ',' field } ']'
pattern ::= STRING
field ::= STRING
```

Možné použití demonstruje následující příklad definice adresování operandu v zásobníkovém segmentu u architektury Intel x86, který je implicitním segmentem při adresování pomocí ukazatele zásobníku SP. U této architektury se pro vyjádření adresování operandů v paměti v jazyce symbolických instrukcí používá hranatých závorek.

```
operand MOFFS_SS = { ''\[(SP)\]'' : [ ''\1'' ] }
```

Definice skupiny operandů `alias-def` se skládá z klíčového slova `alias`, identifikátoru `ID` nově vznikající skupiny, znaku rovná se a ve složených závorkách uzavřeného seznamu identifikátorů operandů. Příkaz `alias-def` umožňuje seskupovat podobné operandy pod stejným názvem.

```
alias-def ::= 'alias' ID '=' '[' alias-item { ',' alias-item } ']'
alias-item ::= ID
```

Složený řídící příkaz reprezentující podmíněné vykonání kódu `if-stmt` se skládá z klíčového slova `if`, podmínky ve formě výrazu `expression`, dvojtečky a bloku příkazů představujícího zanořenou programovou větev. Příkaz může obsahovat nepovinnou `else-stmt` větev.

```
if-stmt ::= 'if' expression ':' suite [ else-stmt ]
else-stmt ::= 'else' ':' suite
```

Blokem příkazů `suite` může být jeden příkaz `suite-stmt` ukončený znakem nového řádku `NEWLINE` nebo sled příkazů, který začíná novým řádkem `NEWLINE` a je uzavřen změnami v odsazení ve formě bílých znaků (`INDENT` a `DEDENT`).

```
suite ::= suite-stmt NEWLINE
        | NEWLINE INDENT suite-stmt { suite-stmt } DEDENT
suite-stmt ::= simple-stmt NEWLINE
             | compound-stmt
```

Složený řídící příkaz reprezentující cyklus `while-stmt` se skládá z klíčového slova `while`, podmínky ve formě výrazu `expression`, dvojtečky a bloku příkazů `suite`.

```
while-stmt ::= 'while' expression ':' suite
```

Definice popisu chování instrukce `instruction-def` se skládá z hlavičky a těla. Hlavičku tvoří klíčové slovo `instruction`, identifikátor procedury `ID` a seznam argumentů uzavřený v závorkách. Tělo tvoří blok příkazů. Argumenty vždy představují dvojici operand a parametr s podmínkou, že blok kódu reprezentující chování instrukce může být vykonán pouze tehdy, pokud parametry odpovídají restrikcím definovaným u operandů. Parametry jsou implicitně předávány odkazem, aby se jejich modifikace uvnitř promítla i mimo tělo popisu chování instrukce. Interpretace chování instrukce je podrobněji rozvedena na straně 38.

```
instruction-def ::=
        'instruction' ID '(' [ arg-pair { ',' arg-pair } ] ')' ':' suite
arg-pair ::= operand param
operand ::= ID
param ::= ID
```

Výkonný příkaz interpretující instrukci `exec-instr` se skládá z klíčového slova `exec` a textového řetězce `STRING` s kódem instrukce v jazyce symbolických instrukcí, jež má být interpretována.

```
exec-instr ::= 'exec' STRING
```

Definice procedury `procedure-def` se skládá z hlavičky a těla. Hlavičku tvoří klíčové slovo `procedure`, identifikátor procedury `ID` a seznam parametrů uzavřený v závorkách. Podobně jako v předchozím případě je tělo tvořeno blokem příkazů a parametry jsou předávány odkazem.

```
procedure-def ::= 'procedure' ID '(' [ param { ',' param } ] ')' ':' suite
```

Výkonný příkaz volání procedury `exec-proc` se skládá z identifikátoru procedury `ID` a seznamem parametrů `par` uzavřeným v závorkách. Parametrem instrukce může být libovolný výraz `expression`.

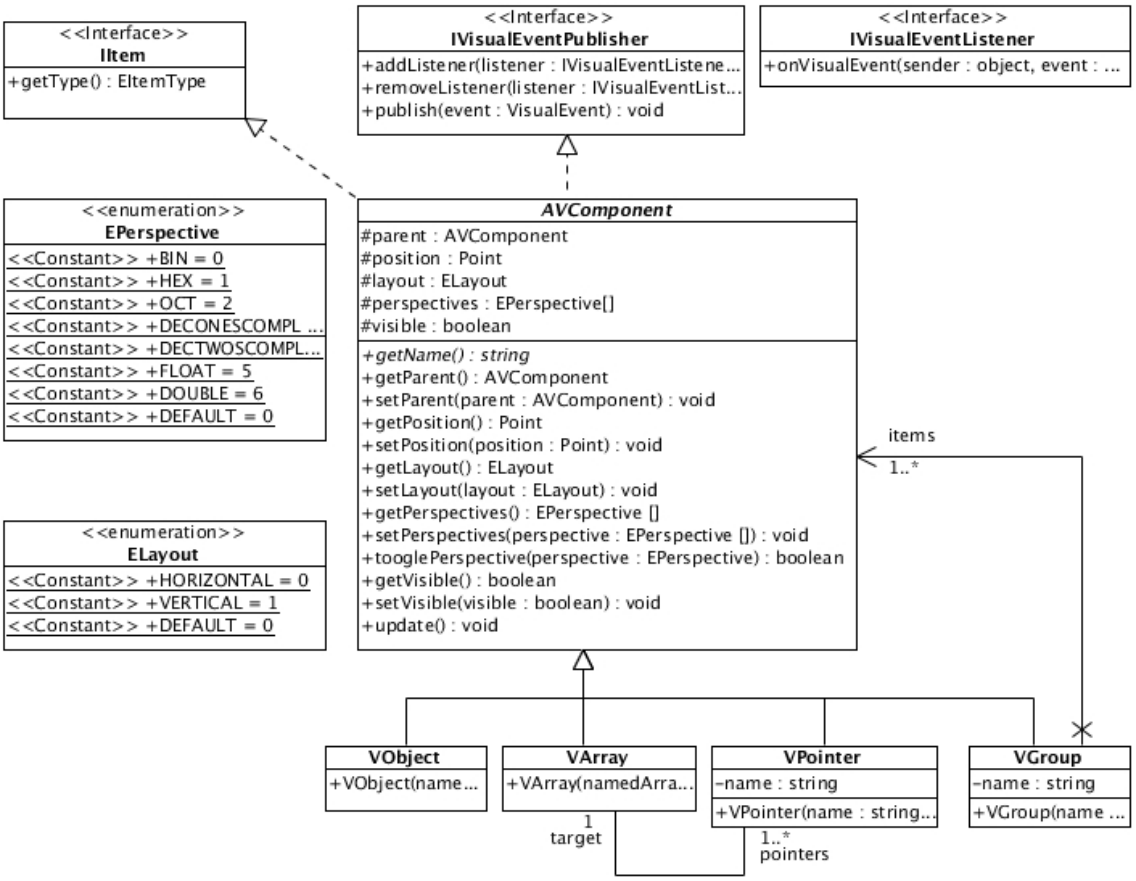
```
exec-proc ::= ID '(' [ par { ',' par } ] ')'
par ::= expression
```

5.3 Reprezentace a popis grafického výstupu

Data uchovaná v objektech reprezentujících prvky emulované architektury potřebují být zobrazena v rámci grafického výstupu aplikace. K tomu je třeba definovat třídy zajišťující jejich vykreslení (viz. obrázek 5.4). Při návrhu musí být brán ohled na to, aby nevznikla závislost tříd reprezentujících samotné prvky architektury na toolkitu grafického uživatelského rozhraní. Níže uvedené třídy jsou tedy mezistupněm mezi samotnou grafickou podobou prvku (závislou na konkrétní použité knihovně pro zobrazení) a modelem architektury, který byl navržen v předcházejících sekcích.

`AVComponent` Abstraktní třída definující rozhraní pro grafickou reprezentaci prvků. Uchovává informaci o rodičovském prvku, pozici, rozložení, pohledů a barvě grafické reprezentace prvku.

`VObject` Třída dědící ze třídy `AVComponent`, která představuje zobrazení jedinečného prvku architektury.



Obrázek 5.4: Diagram tříd reprezentující grafické zobrazení.

`VArray` Třída dědící ze třídy `AVComponent`, která představuje zobrazení kolekce prvků s přístupem pomocí indexu. Objekty třídy udržují seznam všech svých ukazatelů.

`VPointer` Třída dědící ze třídy `AVComponent`, která zajišťuje zobrazení ukazatele do kolekce prvků.

`VGroup` Třída představující zobrazení skupiny prvků. Skupina se skládá z dalších grafických prvků, které musejí být vykresleny spolu se skupinou. Je na ni tedy možné pohlížet jako na atomický prvek, a proto je zde s výhodou použit návrhový vzor *Composite* (viz. [11]).

`EPerspective` Výčtový typ určující pohled na data. Může nabývat hodnot:

- `BIN` – data ve dvojkové soustavě,
- `OCT` – data v osmičkové soustavě,
- `HEX` – data v šestnáctkové soustavě,
- `TWOSCOMPL` – data v desítkové soustavě, dvojkový doplňek,
- `ONESCOMPL` – data v desítkové soustavě, jedničkový doplňek,
- `FLOAT` – data jako desetinné číslo s jednoduchou přesností dle IEEE 754 (viz. [12]),

- `DOUBLE` – data jako desetinné číslo s dvojitou přesností dle IEEE 754,
- `ASCII` – data jako znak ASCII.

`ELayout` Výčtový typ určující preferovanou orientaci prvku, může nabývat hodnot `HORIZONTAL` nebo `VERTICAL`.

`EColor` Výčtový typ definicí palety vybraných barev.

Určení toho, jaké prvky se mají zobrazovat, může být zajištěno jazykem pro zobrazení. Ten by měl vycházet ze zvyklostí vytvořených v předcházejících odstavcích. V jazyku pro popis zobrazení není třeba používat složené příkazy. Počátečním nonterminálem je tedy `simple-stmt`, který se dále rozvíjí v další příkazy jazyka.

```
simple-stmt ::= group-def         # definiční
              | pointer-def       # definiční
              | show-stmt         # výkonný
              | hide-stmt         # výkonný
```

Definice skupiny `group-def` se bude skládat z klíčového slova `group`, identifikátoru skupiny `ID`, znaku rovná se a v závorkách uzavřeného seznamu identifikátorů `ID` prvků skupiny navzájem oddělených čárkami.

```
group-def ::= 'group' ID '=' '[' group-item { ',' group-item } ']'
group-item ::= ID
```

Definice ukazatele (angl. pointer) `pointer-def` je tvořena klíčovým slovem `pointer`, za kterým se v lomených závorkách nachází identifikátor kolekce 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 `expression`, s jehož pomocí se vypočítá hodnota indexu, na který bude do kolekce prvků ukazováno.

```
pointer-def ::= 'pointer' '<' ID '>' ID '=' expression
```

Příkaz pro zobrazení daného prvku `show-stmt` se skládá z klíčového slova `show` a identifikátoru grafického prvku `ID` a volitelných parametrů zobrazení `at`, `layout`, `perspectives` a `color`. Slouží k zobrazení daného prvku na výstupu grafického uživatelského rozhraní. Identifikátorem může být jakýkoliv z pojmenovaných prvků architektury (reprezentované instancemi tříd `NamedObject` či `NamedArray`), skupina nebo ukazatel (oba definované pomocí příkazů jazyka pro zobrazení). Způsob zobrazení může být určen pomocí volitelných parametrů.

```
show-stmt ::= 'show' ID [ at ] [ layout ] [ perspectives ] [ color ]
```

Volitelný parametr určení pozice prvku `at` příkazu `show-stmt` určuje pozici prvku (levý horní roh) při vykreslení na kreslící plochu v grafickém uživatelském rozhraní. Parametr se skládá z klíčového slova `at` a dvou výrazů `expression` reprezentujících celé číslo oddělených čárkou.

```
at ::= 'at' expression ',' expression
```

Volitelný parametr rozložení prvku `layout` příkazu `show-stmt` určuje preferovanou orientaci prvku – horizontální (klíčové slovo `horizontal`) nebo vertikální (klíčové slovo `vertical`). Parametr se skládá z klíčového slova `layout` a klíčového slova definujícího orientaci.

```
layout ::= 'layout' layout-type
layout-type ::= 'horizontal' | 'vertical'
```

Volitelný parametr určující způsob zobrazení prvku `perspectives` u příkazu `show-stmt` se skládá z klíčového slova `perspectives` a v hranatých závorkách uzavřeného seznamu pohledů na zobrazovaný prvek. Pohled je určen klíčovým slovem `bin` (binární kódování), `oct` (osmičkové kódování), `hex` (šestnáctkové kódování), `dec`, které je jiným možným zápisem pro `twoscompl` (desítkové kódování – dvojkový doplněk), `onescompl` (desítkové kódování – jedničkový doplněk), `float`, `double` (desetinné číslo s jednoduchou, resp. dvojitou přesností dle normy IEEE 754), nebo `ascii` (zobrazení jako znak ASCII).

```
perspectives ::= 'perspectives' '[' perspective { ',' perspective } ']'
perspective ::= 'bin' | 'oct' | 'hex' | 'dec' | 'twoscompl' | 'onescompl'
              | 'float' | 'double' | 'ascii'
```

Volitelný parametr nastavení barvy `color` příkazu `show-stmt` se skládá z klíčového slova `color` a textového řetezce `STRING` s názvem barvy dle standardu CSS (viz. [23]).

```
color ::= 'color' STRING
```

Příkaz `hide-stmt` skládající se z klíčového slova `hide` a identifikátoru `ID` má opačnou funkci v porovnání s příkazem `show-stmt`. Slouží ke skrytí zobrazení na grafickém výstupu.

```
hide-stmt ::= 'hide' ID
```

Navržený jazyk umožňuje dynamicky měnit stav a způsob zobrazení architektury v rámci grafického uživatelského rozhraní emulátoru.

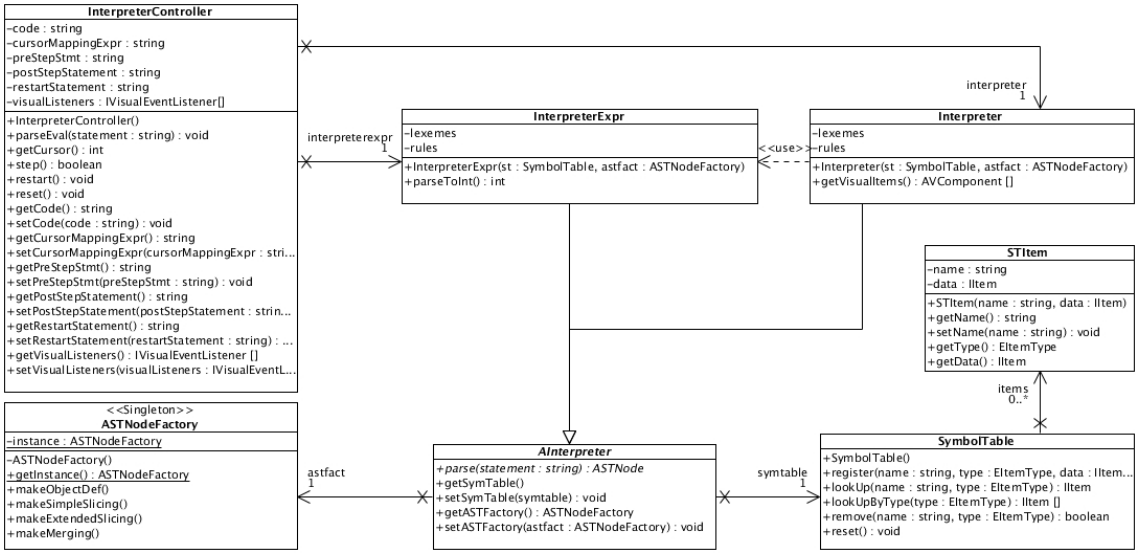
5.4 Návrh řídících struktur emulátoru

Emulátor bude řízen pomocí interpretu. K tomu je potřebné definovat třídy, jejichž objekty reprezentují interpret a jeho podpůrné struktury (návrh na obrázku 5.5).

`AInterpreter` Abstraktní třída interpretu definující operace pro překlad daného řetězce do podoby abstraktního syntaktického stromu a operace pro nastavení a získání tabulky symbolů a továrny (angl. factory) na vytváření prvků abstraktního syntaktického stromu, které jsou interpretem využívány.

`InterpreterExpr` Interpret realizující překlad výrazů `expression` (viz. 5.1). Tato část byla navržena odděleně jako samostatný podsystém hlavního interpretu tak, aby byl použitelný samostatně i v jiných částech aplikace.

`Interpreter` Třída představující hlavní interpret zajišťující zpracování kódu zapsaného ve výše navržených jazycích a vykonání potřebných změn v architektuře, instrukční sadě nebo zobrazení. Zapouzdřuje činnosti lexikálního a syntaktického analyzátoru. Při tom spolupracuje s tabulkou symbolů.



Obrázek 5.5: Digram tříd řídících struktur emulátoru.

`InterpreterController` Objekt této třídy představuje řadič interpretu zajišťující ovládání emulace. Je také primárně užívaným prvkem návrhu pro komunikaci s uživatelským rozhraním. Obsahuje následující metody implementující řízení emulátoru:

- `step()` – vykonání kroku emulace,
- `restart()` – restart emulace,
- `reset()` – uvedení do výchozího stavu.

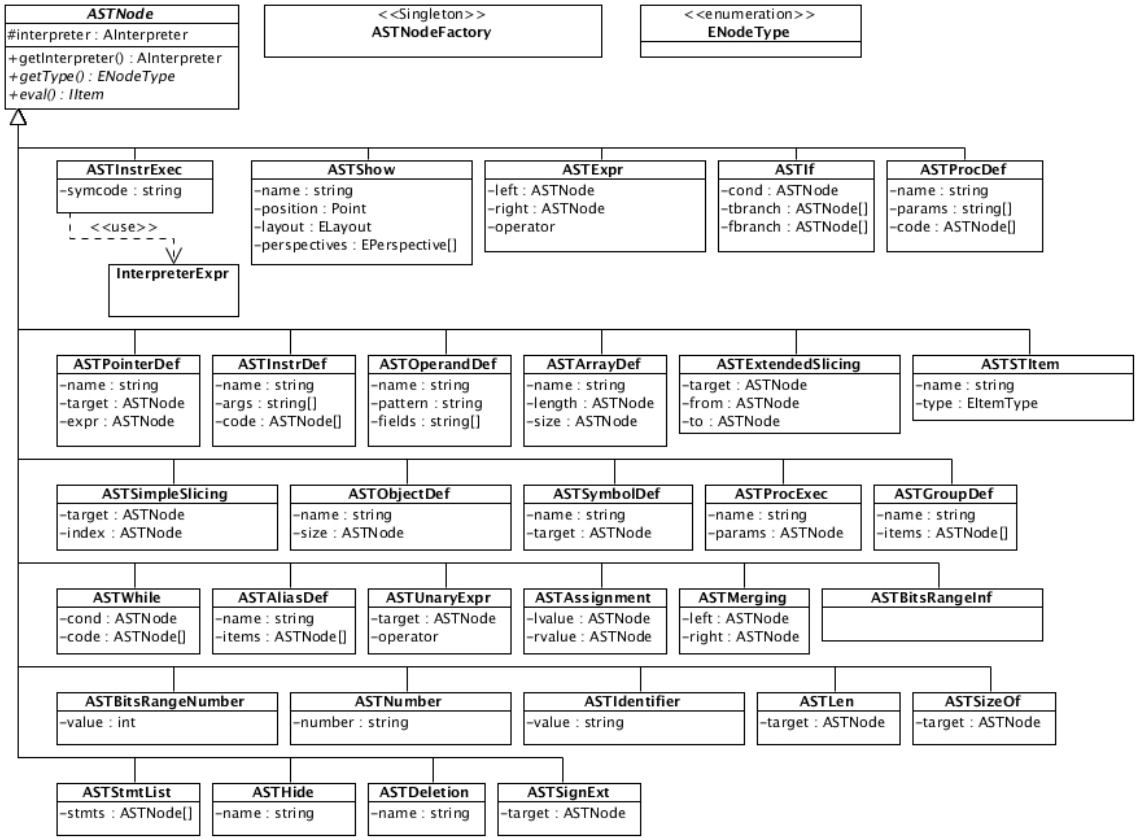
Řadič interpretu uchovává řádky s kódem zapsaným v jazyce symbolických instrukcí emulované architektury. Jelikož součástí řešení není disasembler, je v rámci návrhu uvažováno zjednodušení, při kterém veškeré instrukce mají stejnou jednotkovou velikost (narozdíl od reality, kde instrukce mají obecně různou délku). Ze stejného důvodu také nelze ve strukturách představujících paměť zobrazovat operační kódy instrukcí a naopak. Řadič také umožňuje nastavit výraz ve výše navrženém jazyce pro popis architektury, který bude použit pro výpočet pozice programového kurzoru emulované architektury (např. za pomoci objektu, který představuje registr PC). Podobně lze nastavit příkazy vykonávané před a po zpracování kroku emulace (vhodné např. pro automatickou inkrementaci programového čítače) nebo při restartu emulátoru.

Definice prvků architektury, její instrukční sada i způsob grafického zobrazení jsou uloženy v tabulce symbolů a s ní spolupracujících strukturách.

`IItem` Rozhraní, které musí implementovat všechny objekty modelu architektury. Slouží k určení typu prvku architektury.

`EItemType` Výčtový typ, určující typ konkrétního prvku modelu architektury. Využívá se také pro vyhledávání v tabulce symbolů.

`SymbolTable` Třída představuje tabulku symbolů spravovanou interpretem sloužící k uchování informací o pojmenovaných prvcích. Obsahuje operace umožňující přidávání



Obrázek 5.6: Diagram tříd abstraktního syntaktického stromu.

nových položek `register()`, jejich vyhledávání `lookUp()` a rušení `remove()`, resp. `reset()`.

`STItem` Třída reprezentující položku tabulky symbolů, v jejichž atributech je uložen název položky a reference na příslušná data.

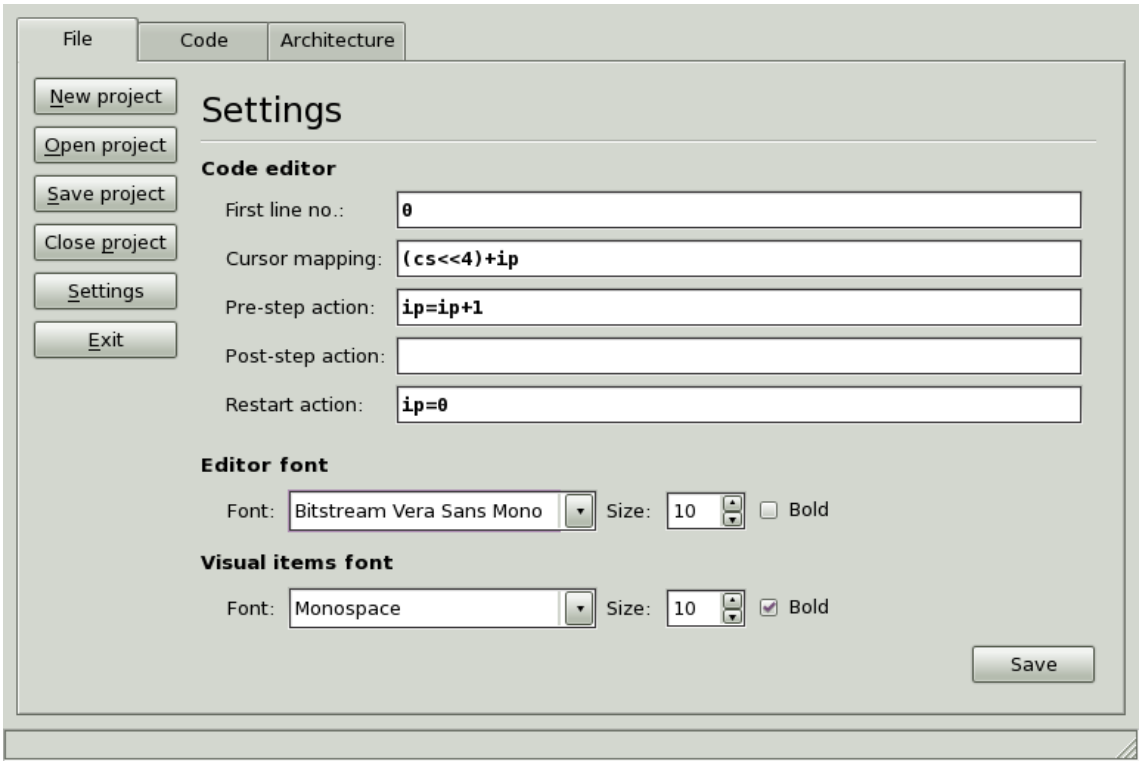
Při syntaktické analýze interpret vytváří abstraktní syntaktický strom. V rámci návrhu byly vytvořeny následující třídy umožňující vytváření a reprezentaci této abstraktní datové struktury (obrázek 5.6).

`ASTNode` Třída reprezentující uzel abstraktního syntaktického stromu. Jedná se o abstraktní třídu, ze které jsou odvozeny všechny ostatní uzly. Každá zděděná třída musí implementovat operace pro určení typu uzlu `getType()` a vykonání sémantického významu uzlu `eval()`.

`ASTNodeFactory` Třída představující továrnu pro tvorbu uzlů abstraktního syntaktického stromu. Při návrhu byla využita kombinace návrhových vzorů *Simple Factory* a *Singleton*. U třídy nejsou pro přehlednost zapsány veškeré metody.

`ENodeType` Výčtový typ sloužící pro určení typu uzlu abstraktního syntaktického stromu.

`AST*****` Všechny třídy, jejichž jméno začíná prefixem AST, představují uzly abstraktního syntaktického stromu představující a danou konkrétní gramatickou konstrukci.



Obrázek 5.7: Grafické uživatelské rozhraní – záložka *File*.

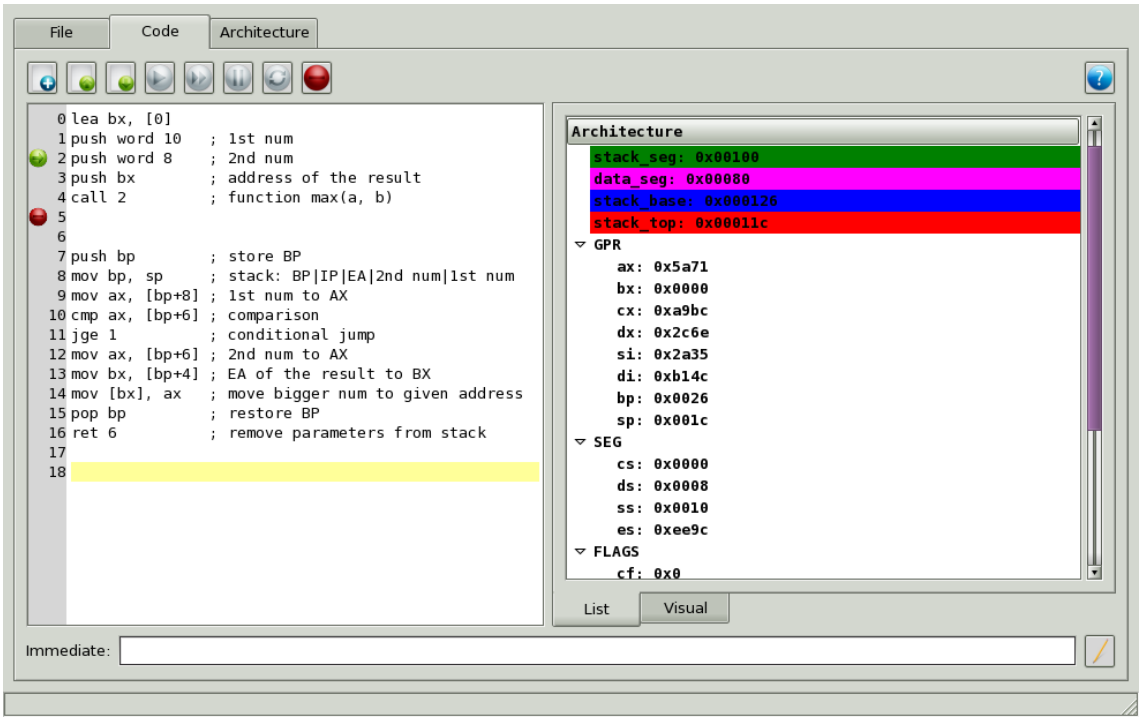
5.5 Grafické uživatelské rozhraní

Hlavním požadavkem na vlastnosti grafického uživatelského rozhraní (zkr. GUI), které představuje most mezi výkonným jádrem emulátoru a uživatelem, je spojení komfortního psaní kódu v asembleru emulované architektury a zobrazení jejího aktuálního stavu.

Z důvodů přehlednosti budou pohledy na zpracovávanou architekturu rozděleny pomocí systému záložek (viz obrázek 5.7):

- *File* sloužící k uložení a otevření souborů souvisejících s aktuálně emulovanou architekturou,
- *Code* představující hlavní pracovní pohled pro uživatele emulujícího symbolické instrukce. Slouží k zobrazení kódu v jazyku symbolických instrukcí emulované architektury a jejího stavu pomocí grafické reprezentace,
- *Architecture*, která je dovolující definovat prvky, instrukce a aktuální zobrazení pomocí příslušných jazyků pro jejich popis. Je primárním pohledem pro uživatele definujícího architekturu.

Vzhled záložky *File* je možné vidět na obrázku 5.7. Levá část umožňuje vytvořit, otevřít, uložit nebo zavřít *projekt*, což je programová struktura i společné pojmenování souborů souvisejících s právě emulovanou architekturou. Třída `Project` udržuje informaci o názvu projektu, aktuálním souboru s definicí architektury a souboru s kódem v jazyce symbolických instrukcí emulované architektury. Dále uchovává nastavení projektu jako nastavení

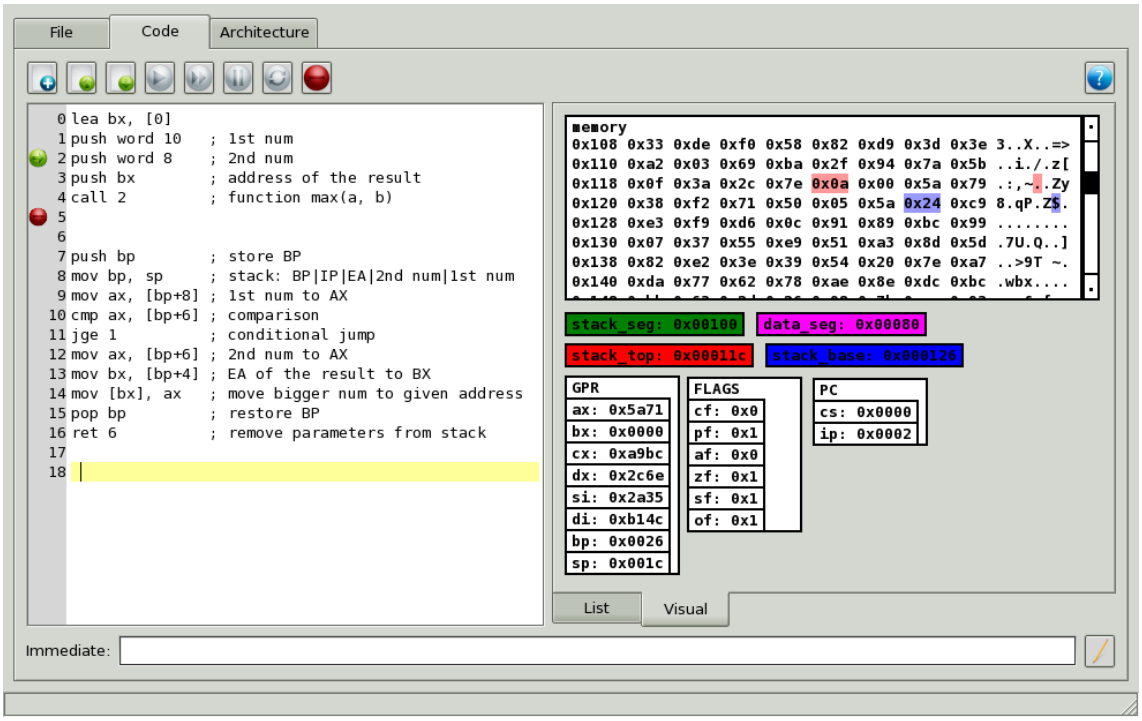


Obrázek 5.8: Zobrazení architektury pomocí stromové struktury.

mapování programového kurzoru (aktuální instrukce) na grafické rozhraní a definice akcí, které mají být v interpretu vykonány při restartování nebo kroku simulace (odpovídají tedy hodnotám udržovaným řadičem interpretu viz. strana 28). Třída také pomocí metod `loadXML()` a `storeXML()` umožňuje načítání, resp. uložení informací o projektu do textové formy ve formátu `XML`. Pro usnadnění práce s emulátorem při prezentacích (např. při výuce) poskytuje záložka *File* u písem použitých na grafických prvcích a v editorech kódu na záložkách *Code* a *Architecture* možnost nastavit jejich řez, velikost a váhu.

Návrh vzhledu základního pracovního okna aplikace s aktivní záložkou *Code* zobrazuje obrázek 5.8. V horní části se nachází panel umožňující spravovat aktuálně otevřený soubor s kódem v jazyce symbolických instrukcí emulované architektury a zobrazení stručné nápovědy k dané záložce. Panel dále obsahuje následující tlačítka pro ovládaní emulace:

- spuštění emulace – po stisku emulátor ve sledu emuluje instrukce jednu za druhou, běh může být přerušen stiskem tlačítka pro zastavení emulace nebo přístupem na bod přerušení,
- krok emulace – vykonána je pouze instrukce na aktuální pozici programového kurzoru,
- zastavení emulace – vedoucí k zastavení běžící emulace,
- znovuspuštění emulace – emulátor je uveden do výchozího stavu daného příslušným nastavením na záložce *File*,
- vložení bodu přerušení – umístí bod přerušení na řádek programu s aktivním textovým kurzorem.

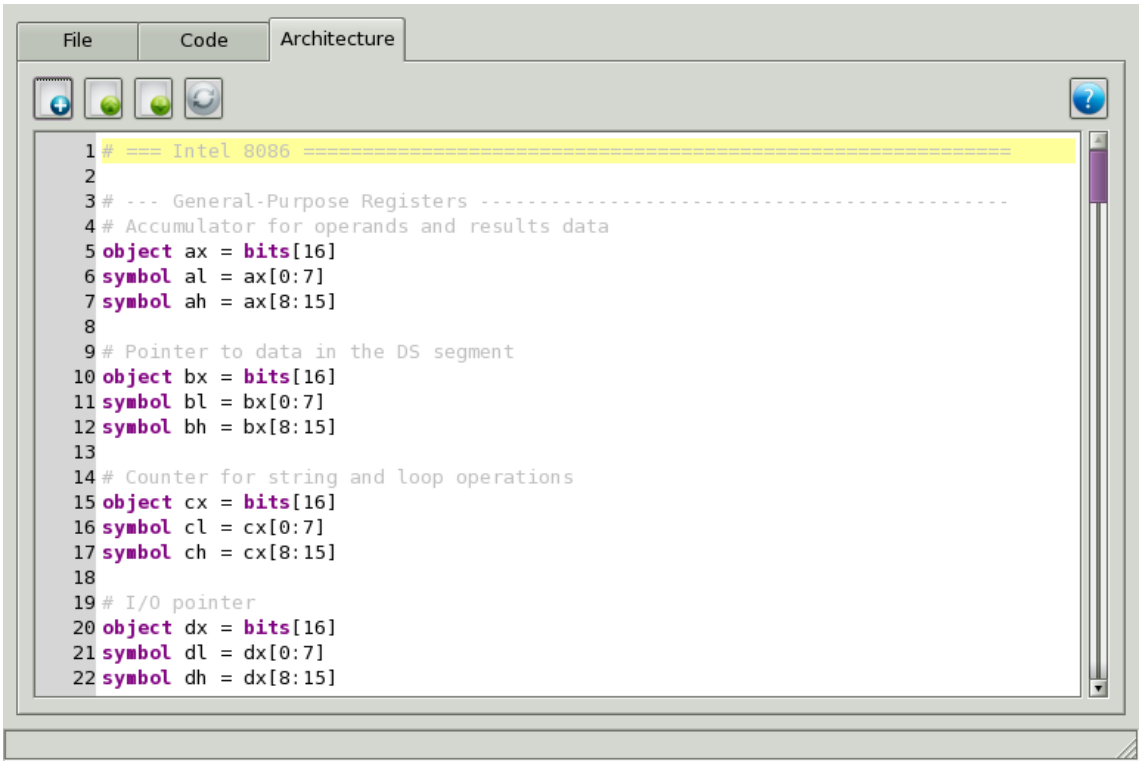


Obrázek 5.9: Zobrazení architektury pomocí grafické reprezentace.

Levá část slouží k zápisu kódu a vyznačení aktuální pozice programového kurzoru spolu s body přerušení (způsob řádkování odpovídá popisu uvedeném při definici řadiče interpretu na straně 28), pravá část zobrazuje stav architektury v požadovaném formátu (v tomto případě pomocí stromové struktury). Zobrazení stavu lze modifikovat pomocí ovládacích prvků okna nebo přímou editací kódu v jazyce pro popis zobrazení. Pro větší přehlednost je možné přepnout formát zobrazení i do plně grafické reprezentace ukázané na obrázku 5.9. Obrázek demonstruje zobrazení grafické skupiny jedinečných prvků (skupiny `GPR`, `SEG` a `FLAGS`) a kolekce prvků (`memory`) s ukazateli (`stack_top`, `stack_base`, `data_seg` a `stack_seg`). Počáteční rozmístění prvků na ploše bez určených souřadnic přidělených uživatelem definujícím zobrazení architektury je automatizováno. Zobrazené prvky mohou být libovolně pozičně uspořádány. Vyvolání kontextové nabídky (pravým tlačítkem myši) vede na zobrazení dalších možností nastavení grafického prvku reflektující možnosti nastavení tříd definujících rozhraní pro grafickou reprezentaci prvků (viz. strana 24). Spodní část okna umožňuje zapsat a vykonat příkazy jazyka pro definici architektury bez zbytečného zdržení, které by způsobilo přepínání mezi záložkami *Code* a *Architecure*.

Záložka *Architecture* zachycená na obrázku 5.10 slouží primárně pro zápis definici architektury. K tomuto účelu slouží textový editor kódu vybavený zvýrazněním syntaxe. V panelu nacházejícím se v horní části záložky je kromě možnosti zobrazit stručnou nápovědu a spravovat aktuálně otevřený soubor s definicí architektury uživateli též umožněno uvést architekturu do výchozího stavu definovaného popisem.

Vedle plně grafického výstupu by uživatelské rozhraní mělo poskytovat i přístup pomocí prostředí příkazové řádky, které nabízí lepší možnosti dávkového zpracování ať už pro účely testování při vývoji aplikace, nebo při jejím běžném nasazení v praxi.



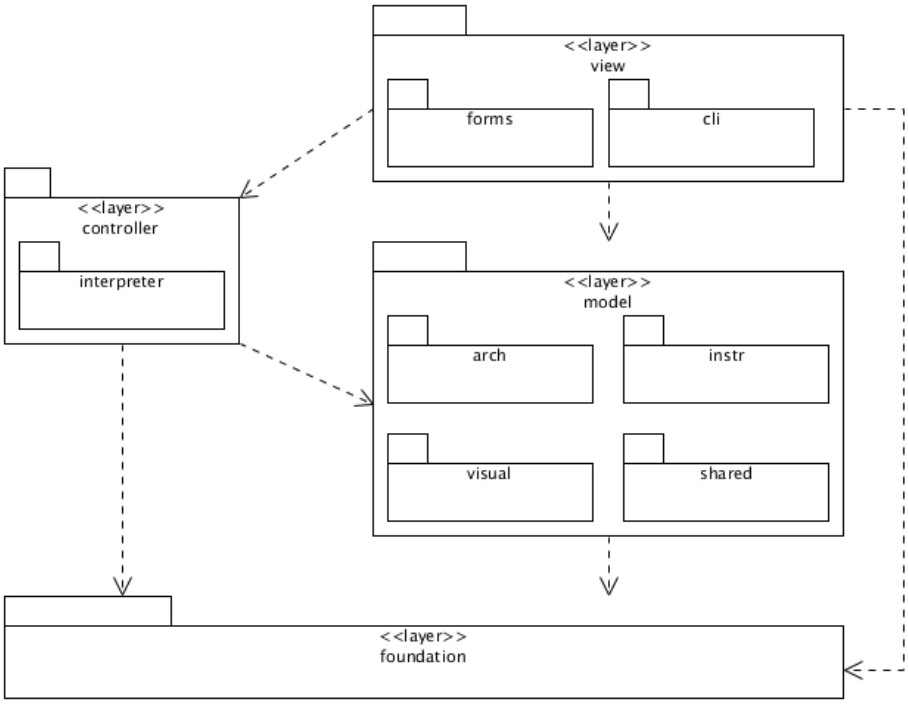
Obrázek 5.10: Grafické uživatelské rozhraní – záložka *Architecture*.

5.6 Rozdělení návrhových tříd do vrstev

Třídy navržené v předchozích kapitolách lze rozdělit s použitím architektonického stylu *MVC (Model-View-Controller)* do třech logicky ucelených vrstev `model`, `view` a `controller`. Pro abstraktní struktury, které jsou znovu použitelné i v projektech jiného zaměření a které jsou využívané napříč všemi uvedenými vrstvami, byla navíc zavedena čtvrtá vrstva `foundation`. Výsledné rozčlenění je možné vidět na obrázku 5.11.

Vrstva `model` obsahuje třídy související s prvky a instrukční sadou emulované architektury. Je členěna na následující balíčky:

- `arch` – obsahující třídy `AObject`, `Object`, `NamedObject`, `SliceObject`, `MergeObject`, `InfiniteObject`, `NumberObject`, `Bits1d`, `AArray`, `Array`, `NamedArray`, `SliceArray`, `MergeArray`, `InfiniteArray`, `Bits2d` a `Bit`, které představují prvky emulované architektury,
- `instr` – složený ze tříd `Procedure`, `Instruction`, `Parameter`, `Argument`, `Operand` a `Alias` reprezentující instrukční sadu architektury,
- `visual` – obsahující třídy `AVComponent`, `VObject`, `VArray`, `VPointer` a `VGroup`, které zajišťují grafickou reprezentaci prvků architektury,
- `shared` – tvořený třídami a rozhraními společně využívanými ve výše vyjmenovaných balíčcích `IItem`, `EItemType` a `Convert`.



Obrázek 5.11: Rozdělení návrhu aplikace do vrstev.

Vrstva `view` je zodpovědná za zobrazení oken uživatelského rozhraní a prezentaci stavu architektury. Skládá se z následujících balíčků:

- `forms` – složený ze tříd reprezentujících formuláře grafického uživatelského rozhraní,
- `cli` – tvořeného třídami pro potřeby prostředí příkazové řádky.

Obsahy balíčků `forms` a `cli` jsou vázány na použité knihovny uživatelského rozhraní při implementaci, a proto jsou uvedeny v příslušné kapitole (strana 42).

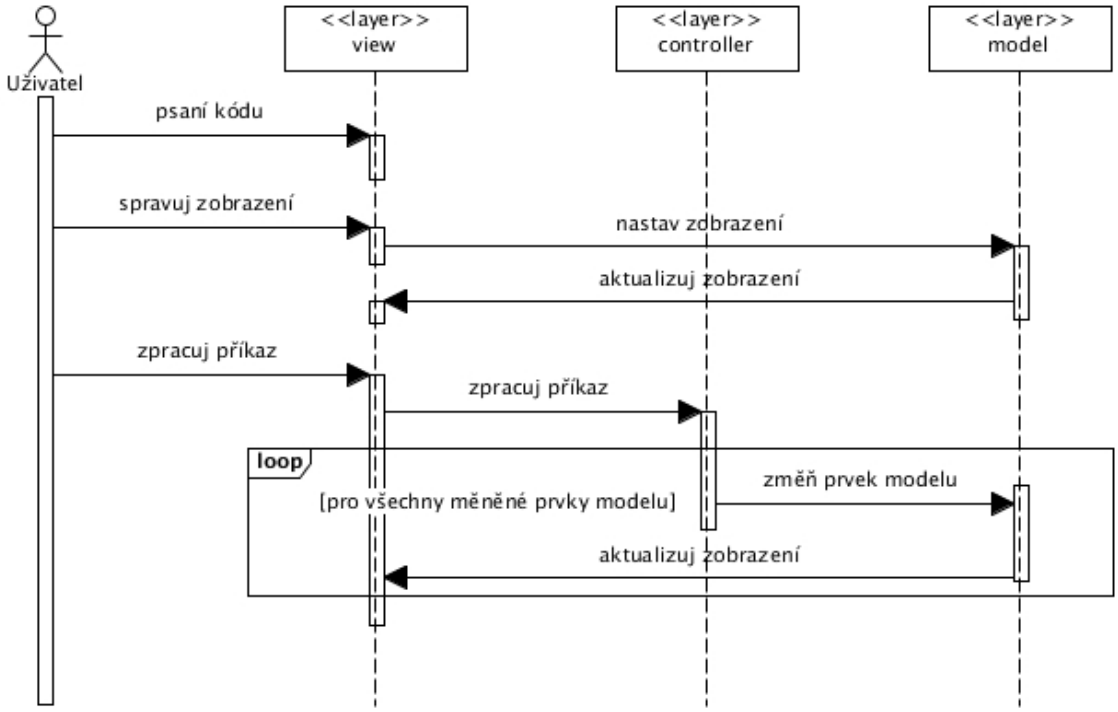
Vrstva `controller` zodpovídá za zpracování a reakce na události (typicky akce uživatele) a může měnit stav jak objektů vrstvy `model`, tak i `view`. Vrstva obsahuje balíčky:

- `interpreter` – obsahující třídy `AInterpreter`, `Interpreter`, `InterpreterExpr`, `SymbolTable` a `STItem`,
- `ast` – složený ze tříd `ASTNodeFactory`, `ASTNodeType`, `ASTNode` a dalších tříd, které odpovídají jednotlivým uzlům abstraktního syntaktického stromu.

5.7 Princip činnosti emulátoru

Emulátor bude řízen pomocí interpretu, který je navržen jako dynamický – změny se projevují po jednotlivých příkazech příslušného jazyka. Obrázek 5.12 zobrazuje interakce v emulátoru na úrovni vrstev pro nejčastější případy užití:

- *psaní kódu* – uživatel píše kód v jazyku symbolických instrukcí emulované architektury,



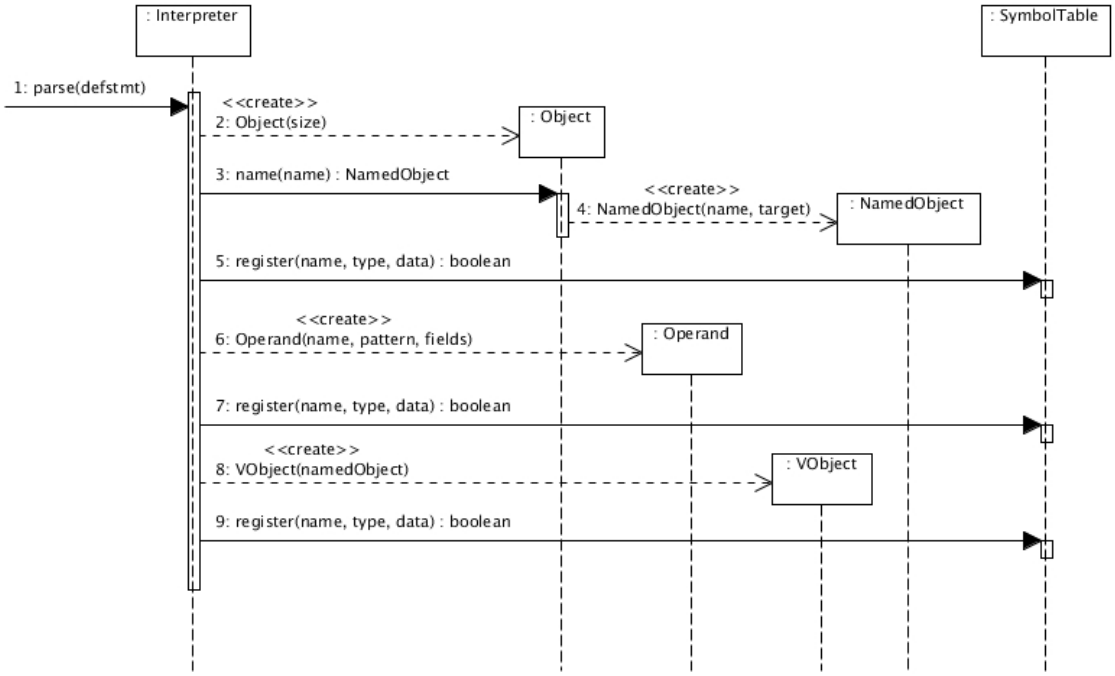
Obrázek 5.12: Interakce v emulátoru na úrovni vrstev.

- *spravuj zobrazení* – uživatel mění grafický pohled (např. forma zobrazení, použitá soustava) na architekturu, nemění však její stav,
- *zpracuj příkaz* – vede na volání interpretu, který provede modifikace nad architekturou nebo její instrukční sadou.

Případ užití *psaní kódu* vede na interakci pouze mezi uživatelem emulátoru a vrstvou uživatelského rozhraní. Případ *spravuj zobrazení* způsobuje komunikaci mezi uživatelským prostředím a grafickou reprezentací spravovaného prvku (některá z podtříd třídy `AVComponent`, viz. strana 24).

Případ *zpracuj příkaz* vede na komunikaci mezi všemi vrstvami, návrhy a činnosti při něm prováděné jsou detailněji rozebrány pomocí následujících diagramů sekvence. Činnosti je možné rozdělit podle typu příkazu, který je vyvolal na *definiční*, *rušící*, *výkonné* a *řídící*. Průběh zpracování příkazů stejného typu je podobný, a v následujících odstavcích je tedy prezentován na vybraných zástupcích.

Při zpracování definičního příkazu objekt třídy `Interpreter` představující interpret provede lexikální a syntaktickou analýzu příkazu a přitom úzce spolupracuje s objektem třídy `SymbolTable`. Princip činnosti je pro názornost popsán diagramem sekvence (obrázek 5.13) na příkladu, ve kterém je definován nový jedinečný prvek architektury. Diagram nezobrazuje z důvodu zachování přehlednosti proces lexikální a syntaktické analýzy, a tedy ani tvorbu abstraktního syntaktického stromu. Zobrazena jsou pouze volání metod objektů, ke kterým by příslušná syntaktická analýza vedla po zavolání metody `eval()` na kořenový uzel abstraktního syntaktického stromu. (Toto zjednodušení bude aplikováno i v následujících diagramech.) Podobným způsobem probíhá i definice ostatních struktur, které mohou v emulátoru vzniknout.

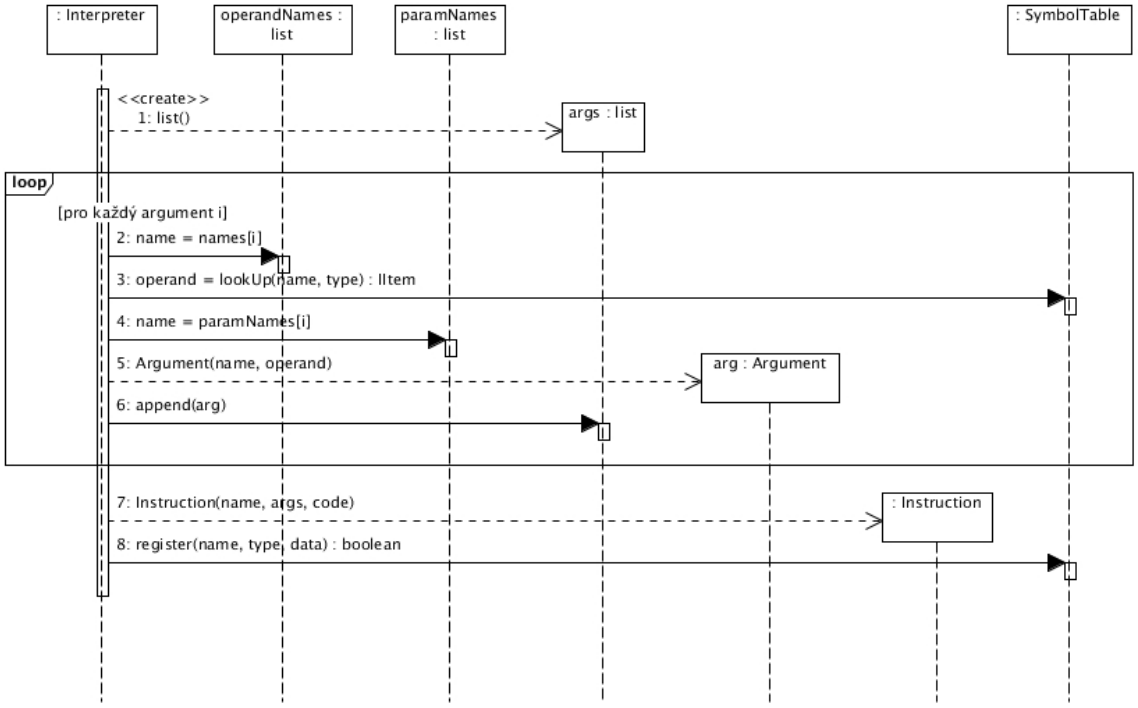


Obrázek 5.13: Příklad zpracování definičního příkazu.

1. Interpret je volán za účelem zpracování definičního příkazu.
2. Interpret vytváří instanci objektu třídy `Object`, představující jedinečný prvek.
3. Zasláním zprávy `name()` na nově vytvořený objekt dochází k vytvoření pojmenovaného objektu architektury `NamedObject`.
4. Pojmenovaný objekt je přidán do tabulky symbolů `SymbolTable` voláním metody `register()`.
5. Na základě jména objektu je vytvořen operand `Operand`.
6. Operand je přidán do tabulky symbolů `SymbolTable` voláním metody `register()`.
7. Vytvoření grafické reprezentace objektu `VObject`.
8. Grafická reprezentace je přidána do tabulky symbolů `SymbolTable` voláním metody `register()`.

Zvláštním případem definičního příkazu je definice instrukce, jejíž průběh znázorňuje sekvenční diagram na obrázku 5.14. Pro zkrácení zápisu následující popis činnosti předpokládá, že v průběhu syntaktické analýzy bylo již zjištěno jméno instrukce, názvy typů operandů a parametrů (uložené v příslušných seznamech `list`) i kód realizující chování instrukce.

1. Interpret vytváří pomocný seznam pro uložení objektů třídy `Argument`.
2. Interpret načítá jméno typu operandu na příslušném indexu.



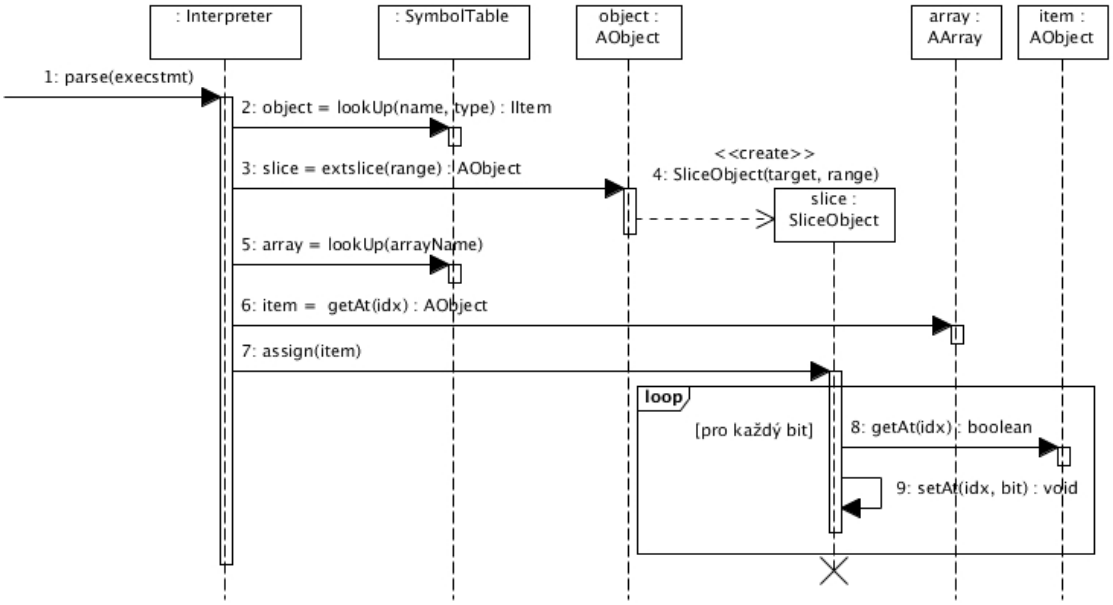
Obrázek 5.14: Příklad zpracování definičního příkazu.

3. Typ operandu je vyhledán v tabulce symbolů `SymbolTable`.
4. Interpret načítá jméno parametru na příslušném indexu.
5. Dochází k vytvoření nového argumentu instrukce `Argument`.
6. Nově vzniklý objekt je přidán do pomocného seznamu.
7. Po vytvoření všech argumentů je vytvořen objekt třídy `Instruction`.
8. Instrukce je přidána do tabulky symbolů `SymbolTable` voláním metody `register()`.

Zpracování rušícího příkazu vede pouze na komunikaci mezi interpretem a tabulkou symbolů (objekty tříd `Interpreter`, resp. `SymbolTable`). Interpret volá metodu `remove()` objektu tabulky symbolů, která odstraní požadovanou položku ze seznamu prvků. Pokud se v tabulce symbolů `SymbolTable` vyskytoval i operand odvozený od rušené položky je tento operand vymazán taktéž. Podobně je v případě existence odstraněna i grafická reprezentace rušené položky.

Při zpracování výkonného příkazu dochází k interakcím mezi interpretem a tabulkou symbolů, a také mezi objekty obsaženými ve vrstvě `model`. Princip činnosti je opět pro názornost popsán na příkladu (obrázek 5.15), ve kterém je do části jedinečného prvku architektury (získané pomocí operace řezu) přiřazena hodnota prvku z kolekce prvků na daném indexu.

1. Interpret je volán za účelem zpracování výkonného příkazu.
2. Interpret provádí vyhledání jedinečného prvku v tabulce symbolů.



Obrázek 5.15: Příklad zpracování výkonného příkazu.

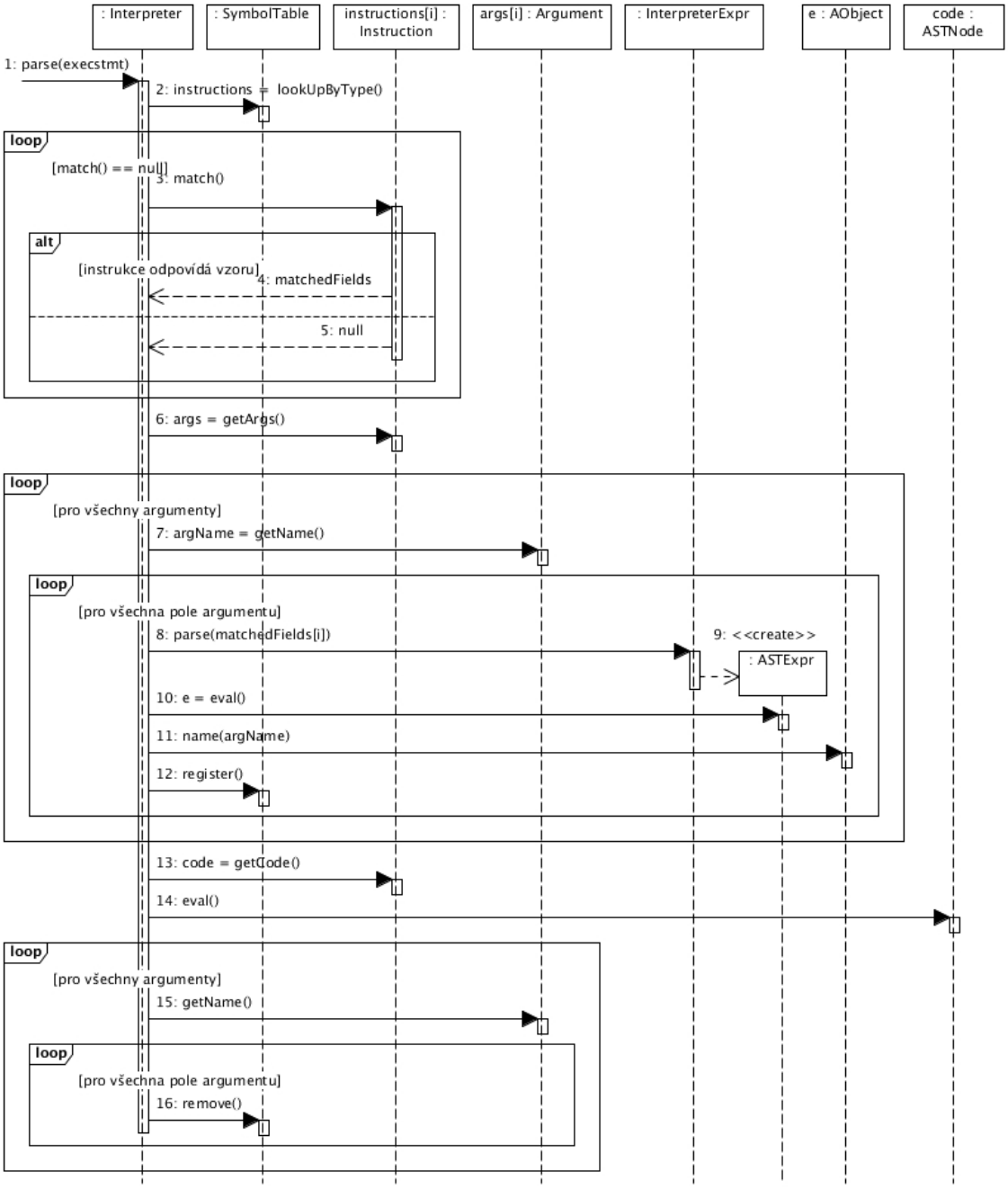
3. Na jedinečný prvek je aplikována operace řezu.
4. Je vytvořen dočasný anonymní prvek představující řez.
5. Interpret vyhledá kolekci prvků v tabulce symbolů.
6. Z kolekce je vybrán prvek na daném indexu.
7. Interpret volá metodu vedoucí na přiřazení hodnoty prvku kolekce do prvku reprezentujícího řez.
8. Načtení bitu ze zdrojového operandu operace přiřazení.
9. Uložení bitu do cílového operandu operace přiřazení.

Zvláštním případem výkonného příkazu se jeví interpretace instrukce. Princip činnosti pro názornost naznačuje sekvenční diagram na obrázku 5.16. Tato činnost využívá při vyhledávání instrukce informace zadané při definici instrukce. Na základě názvu instrukce a vzoru operandů (definice operandů viz. strana 23) je sestaven regulární výraz `match` ve tvaru, který je možné popsat následujícím způsobem:

```
match ::= sep instr-name sep [ op sep { ',' sep op sep } ]
```

Část výrazu `instr-name` značí název instrukce, `sep` představuje separátor ve formě znaků mezera nebo tabulátor a `op` reprezentuje odpovídající vzor operandu. Uvedená forma byla zvolena tak, aby odpovídala způsobu zápisu instrukcí ve většině jazyků symbolických instrukcí.

1. Interpret je volán za účelem interpretace instrukce.

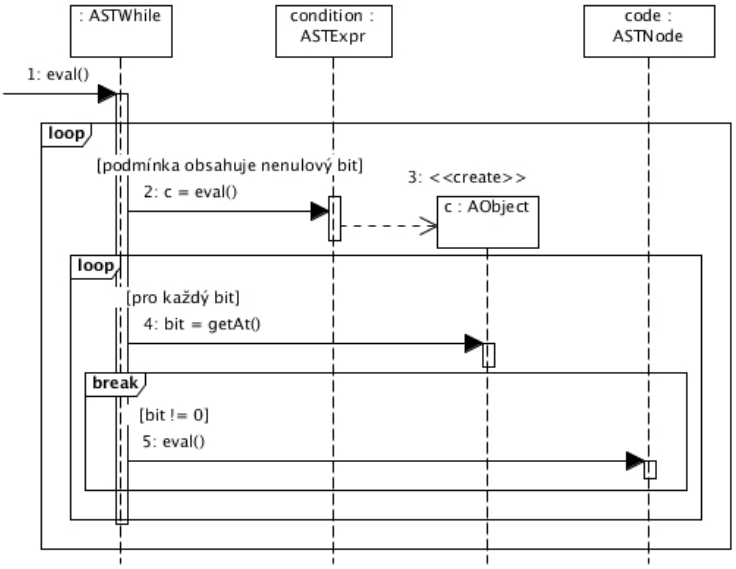


Obrázek 5.16: Příklad interpretace instrukce.

2. Interpret voláním metody `lookUpByType()` provádí vyhledání veškerých aktuálně definovaných instrukcí v tabulce symbolů.
3. Pro každou instrukci byl při její definici na základě typu a počtu operandů vytvořen regulární výraz. Aktuálně prováděná instrukce je s tímto výrazem porovnána v metodě `match()`.
4. Pokud tímto regulárním výrazem lze generovat řetězec představující instrukci, jež má být interpretována, jsou na základě typů argumentů definovaných pomocí příkazu `operand-def` (viz. strana 23) naplněna a vrácena pole argumentu.
5. V případě, že interpretovaná instrukce neodpovídá vzoru danému regulárnímu výrazu se výše uvedený postup opakuje.
6. Interpret volá metodu `getArgs()` načítající pole argumentů.
7. Pro každý argument je metodou `getName()` nejdříve načteno jméno.
8. Hodnota pole je předána k vyhodnocení interpretu výrazů `InterpreterExpr`.
9. Interpret výrazů na základě hodnoty pole vytváří nový abstraktní syntaktický strom a navrátí jeho kořenový uzel `ASTNode()`.
10. Kořen abstraktního syntaktického stromu je vyhodnocen voláním metody `eval()`.
11. Výsledek vyhodnocení, který musí být třídy `AObject` nebo `AArray`, se pojmenuje voláním metody `name()` jménem aktuálního argumentu.
12. Nově vzniklý pojmenovaný prvek je přidán do tabulky symbolů voláním její metody `register()` a je přístupný v době interpretace kódu instrukce.
13. Interpret načítá kód popisu chování instrukce.
14. Kód instrukce je vyhodnocen.
15. Po vykonání kódu musí být dočasně vytvořené položky zrušeny. Nejdříve je načteno metodou `getName()` jméno argumentu.
16. Položka s daným jménem je metodou `remove()` odstraněna z tabulky symbolů.

U posledního typu příkazu – příkazu řídícího – je nejdříve vyhodnocen výraz podmínky. Jeho vyhodnocení může být provedeno, pouze když je výstupem výrazu podmínky objekt třídy `AObject`. Podmínka je platná, pokud výsledek jejího vyhodnocení obsahuje nenulovou hodnotu, v opačném případě je podmínka považována za neplatnou. Na základě platnosti podmínky je poté dále zpracován příslušný blok příkazů. Na příkladu na obrázku 5.17 je naznačena činnost na příkladu smyčky while.

1. Uzel smyčky while je interpretem volán za účelem vyhodnocení.
2. Vyhodnocení podmínky smyčky.
3. Vytvoření nového objektu třídy `AObject` s výsledkem vyhodnocení podmínky smyčky.
4. Bity objektu představujícího výsledek vyhodnocení podmínky smyčky jsou testovány na nenulovou hodnotu.



Obrázek 5.17: Příklad interpretace řídícího příkazu.

5. Je-li hodnota nenulová, podmínka je platná a vyhodnocuje se uzel abstraktního syntaktického stromu představující kód smyčky. Výše uvedený postup se opakuje od bodu 2. V případě nulové hodnoty je výpočet v rámci uzlu smyčky while ukončen.

Kapitola 6

Implemetace emulátoru

Pro implementaci aplikace byl použit jazyk Python. Výhodou tohoto interpretovaného programovacího jazyka je vyšší abstrakce při programování, 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.

6.1 Syntaktický analyzátor

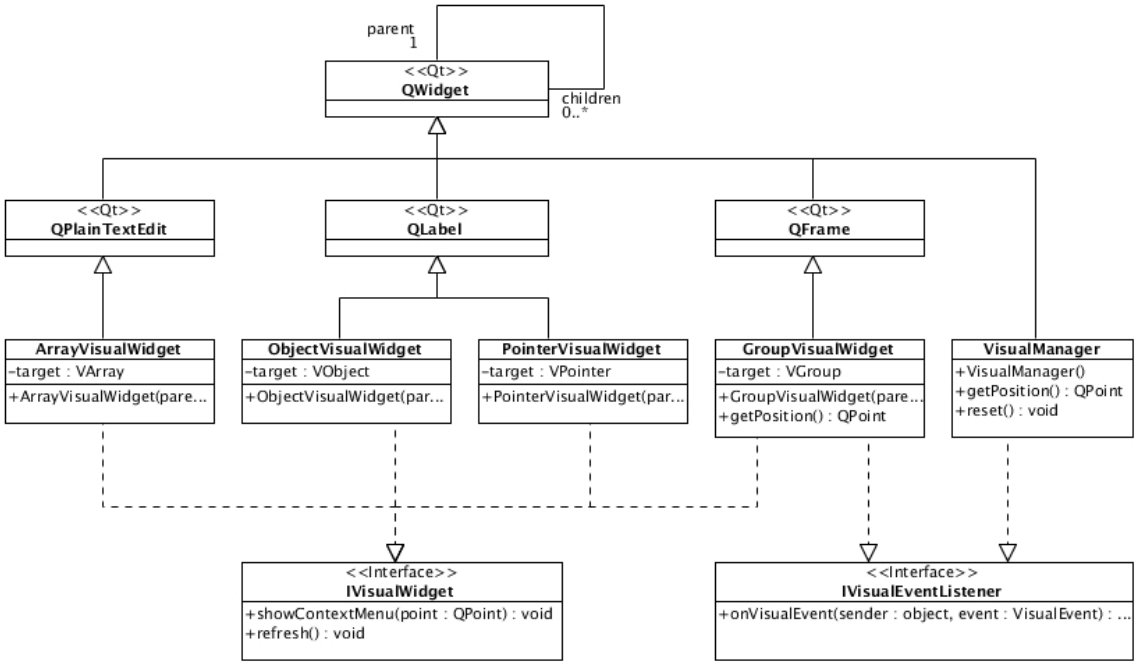
K implementaci syntaktického analyzátoru pro zpracování navržených jazyků pro popis byl zvolen nástroj PLY [5]. PLY je nástroj napsaný v jazyce Python, jenž implementuje hojně užívané programy LEX a YACC sloužící pro tvorbu překladačů. Důvodem k tomuto postupu byl opět rychlejší vývoj aplikace než v 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 vytvoření příslušného uzlu abstraktního syntaktického stromu.

6.2 Grafické uživatelské rozhraní

Pro grafické uživatelské rozhraní byla zvolena grafická nadstavba PyQt4 využívající grafickou knihovnu Qt4 [21]. Jejím použitím byla zabezpečena dobrá přenositelnost výsledné aplikace mezi různými operačními systémy.

Rozmístění grafických prvků do formulářů a dialogových oken podle návrhu uvedeného v předchozí kapitole bylo vytvořeno pomocí programu Qt Designer. Nástroj ukládá rozložení grafických prvků do souboru typu XML s příponou `*.ui`. Pomocí nástroje `pyuic4`, který je součástí instalace PyQt4, lze grafické rozhraní navržené v nástroji Qt Designer převést na kód v jazyce Python. Vzniklý kód ovšem zajišťuje pouze rozložení grafických prvků v rámci okna aplikace, logiku okna je třeba programovat odděleně (a to i z důvodu ponechání možnosti efektivně měnit rozložení prvků grafického uživatelského rozhraní). Grafické uživatelské rozhraní vyvíjené aplikace je uloženo v modulech `MainWindow.py`, `DialogAsmHelp.py` a `DialogArchHelp.py`, jeho logika potom v modulu `MainWindowCtrl.py`. Oba výše zmíněné moduly se nacházejí v balíčku `view.form`, který náleží do vrstvy View návrhu MVC.

Pro splnění požadavků uvedených v návrhu grafického uživatelské rozhraní byly v rámci implementace vytvořeny upravené verze grafických prvků (angl. widgets) použité grafické knihovny Qt. Jejich návrh využívá dědičnosti a přidává prvkům nové funkce.



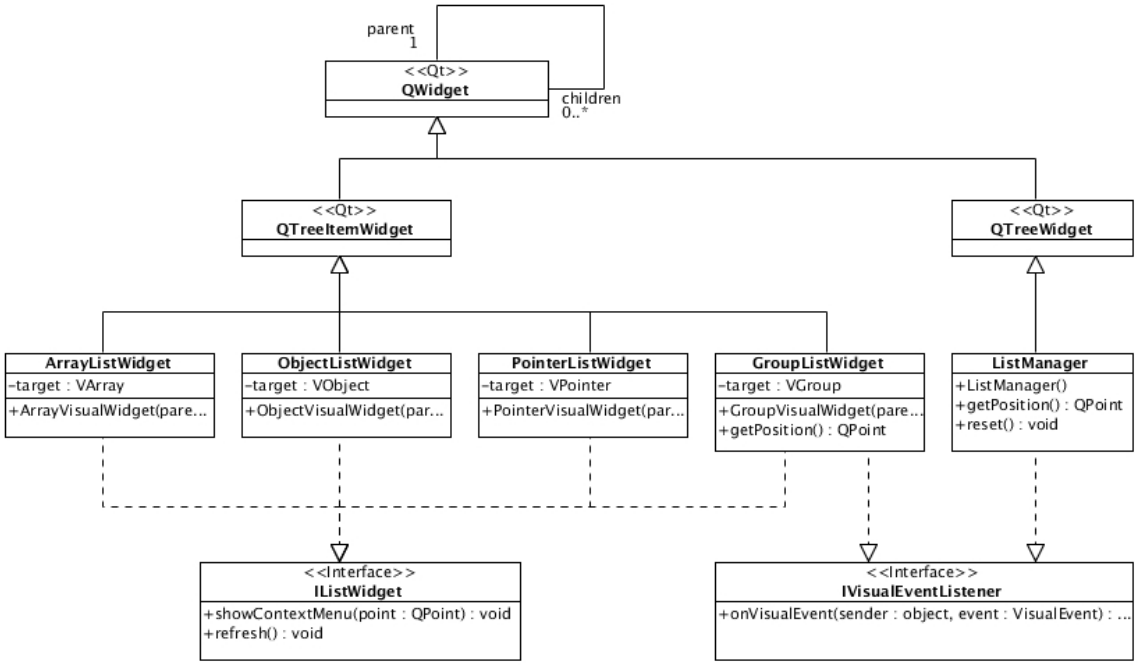
Obrázek 6.1: Třídy grafického zobrazení architektury.

Editor kódu Grafický prvek využívá jako svoji bázi třídu `QPlainTextEdit`. V rámci specializace byl nalevo od editačního prostoru přidán pruh zobrazující číslování řádků a grafické značky (použitelné např. pro zvýraznění aktuální pozice programového kurzoru a bodů přerušení). Dále bylo přidáno zvýraznění aktuálně editovaného řádku pomocí podbarvení. Upravený prvek je uložen v modulu `CodeEditor.py` v balíčku `view.form`.

Grafické zobrazení architektury Plně grafické zobrazení architektury je realizováno objektem třídy `VisualManager` a spoluprací dalších objektů tříd obsažených v balíčku `view.form.widget.visual` zobrazených na obrázku 6.1. Všechny použité prvky vycházejí ze třídy `QWidget`, která je bázovou třídou všech objektů uživatelského rozhraní. Realizace rozhraní `IVisualEventLister` u třídy `VisualManager`, které je součástí návrhového vzoru *Observer*, umožňuje předávat informace o změnách grafické podoby prvků bez nežádoucí závislosti nižších vrstev architektonického návrhu na vrstvách vyšších. Dále třída poskytuje metodu pro získání pozice `getposition()` pro vykreslení prvků bez předem definovaného umístění. O samotné zobrazení stavu jednotlivých prvků architektury se starají třídy:

- `ObjectVisualWidget` – určená pro zobrazení jedinečných prvků,
- `ArrayVisualWidget` – sloužící pro vizuální znázornění kolekcí prvků,
- `PointerVisualWidget` – sloužící pro zobrazení *logických vazeb*,
- `GroupVisualWidget` – používaná pro znázornění *skupin prvků*.

Výše uvedené objekty také realizují rozhraní `IVisualWidget`, které je zavazuje implementovat metodu pro překreslení (`refresh()`) a pro zobrazení kontextového menu (`showContextMenu()`).



Obrázek 6.2: Třídy stromového zobrazení architektury.

Stromové zobrazení architektury Stromové zobrazení architektury je realizováno objektem třídy `ListManager` a spoluprací dalších objektů tříd obsažených v balíčku `view.form.widget.list` zobrazených na obrázku 6.2. Všechny použité prvky vycházejí ze třídy `QWidget`, která je bázovou třídou všech objektů uživatelského rozhraní. Realizace rozhraní `IVisualEventLister` u třídy `ListManager` plní stejnou úlohu jako u předcházejícího grafického prvku. O zobrazení stavu jednotlivých prvků architektury se starají třídy:

- `ObjectListWidget` – určená pro zobrazení jedinečných prvků,
- `ArrayListWidget` – sloužící pro znázornění kolekcí prvků,
- `PointerListWidget` – sloužící pro zobrazení *logických vazeb*,
- `GroupListWidget` – používaná pro znázornění *skupin prvků*.

Výše uvedené objekty také realizují rozhraní `IListWidget`, které je zavazuje implementovat metodu pro překreslení (`refresh()`) a pro zobrazení kontextového menu (`showContextMenu()`).

6.3 Rozhraní příkazové řádky

Zejména kvůli možnosti ověřit správnou činnost emulátoru, obsahuje implementace rozhraní přístup pomocí prostředí příkazové řádky (angl. command line interface – CLI). Do prostředí příkazové řádky se vstupuje zadáním parametru `-cli` při spuštění emulátoru. Volitelně lze pak pomocí parametrů `-asm` a `-adl` následovaných cestou k souboru nahrát kód v jazyce symbolických instrukcí emulované architektury, resp. popis architektury. Emulátor poté

očekává zadávání příkazů jazyka pro popis architektury na standardní vstup, nebo interních příkazů CLI. Mezi interní příkazy, které odlišuje od příkazů jazyka pro popis architektury prefix ve formě znaku dvojtečka, patří:

- `:show` – příkaz vypisují informace na základě parametru, který následuje za ním. Parametr `c` vede na výpis aktuálně zpracovávaného kódu v jazyce symbolických instrukcí emulované architektury. Parametr `v` vede na výpis prvků zobrazených pomocí příkazu `show` (viz. strana 26) jazyka pro popis architektury.
- `:load` – příkaz pro načtení souboru s kódem v jazyce symbolických instrukcí emulované architektury nebo popisem architektury. Určení typu načítaného souboru realizuje parametr `asm` či `adl` následovaný cestou k danému souboru.
- `:set` – příkaz pro nastavení řadiče interpretu (viz. strana 28). Řadič výraz výpočet pozice programového kurzoru (parametr `mapping`), příkazy vykonávané před (parametr `prestep`) a po (parametr `poststep`) zpracování kroku emulace a při restartu emulátoru (parametr `restart`).
- `:step` – příkaz pro provedení kroku emulace.
- `:run` – příkaz opakovaně provádějící kroky emulace dokud se programový kurzor pohybuje v rámci kódu.
- `:restart` – příkaz pro provedení restartu emulace.
- `:reset` – příkaz pro provedení uvedení emulátoru do výchozího stavu.

Ukončení činnosti testovacího rozhraní se provádí zadáním znaku konce souboru EOF.

6.4 Testování emulátoru

Pro ověření správnosti implementace emulátoru a jeho komponent byly použity dva typy testů, které více popisují následující sekce. První se zabývá realizací jednotkových testů (angl. unit test). Druhá sekce pojednává o použitých integračních testech. Projekt byl testován v prostředí operačního systému Linux (distribuce Fedora 14, 64bit a Ubuntu 10.10, 32bit) a Windows XP, 32bit.

6.4.1 Jednotkové testy

Jednotkové testy byly v průběhu implementace vytvářeny ke každému modulu, který představuje implementaci jedné třídy obsažené v návrhu. Pro tento typ testů byl použit testovací systém interpretu jazyka Python nazývaný PyUnit. Použití tohoto nástroje se velmi podobá nástroji JUnit známého z prostředí jazyka Java.

Jednotkové testy se zaměřují zejména na ověření správné funkčnosti jednotlivých metod příslušné třídy, bez spolupráce s ostatními prvky implementace. Každý balíček výsledné implementace obsahuje modul s implementací jednotkových testů tříd, které jsou v daném balíčku zahrnuty. Spuštění programu obsaženém v souboru `aimeUnitTests.py`, který se nachází v adresáři `src`, pak provede veškeré implementované jednotkové testy zároveň.

6.4.2 Integrační testy

Integrační testy ověřují splnění specifikovaných požadavků jako celku. Dohromady testují hlavní položky, které byly předmětem návrhu a následné implementace, testována je tedy zejména komunikace mezi jednotlivými spolupracujícími skupinami prvků.

Integrační test se skládal z vytvoření architektury a programů v daném jazyce symbolických instrukcí s předem daným výsledkem. Následně bylo s výhodou použito CLI rozhraní pro automatické ověření korespondence výsledků. Příklad jednoduchého testu je uveden níže. Další pokročilejší testy jsou uloženy v adresáři `test`.

Příklad testu

- Jméno: Definice jedinečného prvku a instrukce.
- Popis: Pomocí příkazu object jazyka pro popis architektury bude vytvořen jedinečný prvek s názvem `acc` (např. akumulátor) o bitové šířce 8bitů. Pomocí příkazu `instruction` bude vytvořena instrukce `zero` (bez parametrů) zajišťující nastavení obsahu prvku `acc` na nulovou hodnotu.
- Vstup: Soubor `test.adl` s definicí architektury a soubor `test.asm` s programem v jazyku symbolických instrukcí.

```
# Soubor test.adl
object acc = bits[8]
instruction zero():
    acc = acc xor acc

show acc
```

```
; Soubor test.asm
zero     ; vynuluje akumulátor
```

- Očekávaný výstup: Výpis všech zobrazených prvků provedený pomocí interního příkazu CLI `:show v` obsahuje pouze jediný prvek s názvem `acc` a nulovou hodnotou.
- Úspěšnost: PASS

Kapitola 7

Závěr

Cílem této práce bylo navrhnout emulátor umožňující začátečníkům snadný vstup do problematiky programování ve strojovém kódu různých platforem. Byly probrány některé současné projekty věnující se emulaci procesorů (konkrétně MAME, PearPC, Aspectrum a QEMU). Dále bylo analyzováno několik ze současných procesorů jako Intel x86, ARM11, Freescale rodiny HCS08, PowerPC a virtuální architektura Java Virtual Machine. Na základě toho byly vytvořeny požadavky na nově vyvíjený emulátor, a posléze vytvořen návrh řešení zahrnující návrh tříd pro reprezentaci typických prvků obsažených v architekturách a jejich instrukčních sadách. Nad nimi pak byl vystavěn návrh jazyků pro popis architektury a instrukční sady emulovaného stroje. Nakonec bylo navrženo grafické uživatelské prostředí. V průběhu návrhu byl kladen důraz na univerzálnost výsledného řešení a zejména na možnost jeho budoucího nasazení ve výukovém prostředí. Za zmínku také stojí možnost dynamického přidávání a odebírání prvků emulované architektury. Výsledek implementovaný v jazyce Python využívá PLY pro syntaktickou analýzu navržených jazyků a grafický toolkit Qt pro realizaci uživatelského rozhraní. Implementace je dostupná na webových stránkách projektu (viz. [6]), kde je kromě zdrojových kódů, návodu k instalaci a návrhů jednotlivých gramatik, možné nalézt i příklady použití výsledné aplikace.

7.1 Možné pokračování projektu

Do budoucna by bylo zajímavé rozšířit emulátor o funkce, které by upozorňovaly uživatele na nejčastější začátečnické chyby při programování v jazyku strojových instrukcí. Bylo by možné např. nastavit chráněné oblasti v paměti proti přepisu a při porušení těchto omezení uživatele vhodně informovat. Další možností je rozšířit implementaci o emulaci paralelně běžících procesů. Uživatel by se pak mohl blíže seznámit s problematikou, která paralelní zpracování programů doprovází. Dále se nabízí možnost vytvoření rozhraní pro integraci disasembleru pro emulovanou architekturu. To by umožnilo např. označovat úseky v operační paměti a dynamicky je převádět na do jazyka symbolických instrukcí.

Literatura

[1] ARM Ltd.: The ARMv4T Architecture [online]. http://infocenter.arm.com/help/topic/com.arm.doc.dvi0025b/CEGFFCBF.html, 2010 [cit. 2010-11-15].

[2] ARM Ltd.: ARM11 Processor Family [online]. http://www.arm.com/products/processors/classic/arm11/, 2010 [cit. 2010-11-25].

[3] ARM Ltd.: Instruction Set Architectures [online]. http://www.arm.com/products/processors/technologies/instruction-set-architectures.php, 2010 [cit. 2010-11-25].

[4] ARM Ltd.: ARM1176JZ-S Technical Reference Manual [online]. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0333h/ch02s09s01.html, 2010 [cit. 2010-11-28].

[5] Beazley, D.: PLY: Python Lex Yacc [online]. http://www.dabeaz.com/ply/, 2011-02-17 [cit. 2011-04-15].

[6] Charvát, L.: Stránky projektu AIME [online]. http://www.assembla.com/space/aime/, 2011-05-05 [cit. 2011-05-05].

[7] Dvořák, V.; Drábek, V.: *Architektura procesorů*. Brno: VUTIUM, 1999, ISBN 80-214-1458-8.

[8] Fernandez, A. A.: Aspectrum – Another Spectrum Emulator [online]. http://aspectrum.sourceforge.net/, 2004 [cit. 2010-11-15].

[9] Freescale Semiconductor: HC08 Family [online]. http://www.freescale.com/webapp/sps/site/overview.jsp?nodeId=01624684497663, 2010 [cit. 2010-10-28].

[10] Freescale Semiconductor: MC9S08JM60 Series Data Sheet [online]. http://www.freescale.com/files/microcontrollers/doc/data_sheet/MC9S08JM60.pdf, 2011 [cit. 2010-03-22].

[11] Gamma, E.; Helm, R.; Johnson, R.; aj.: *Design Patterns*. Addison-Wesley Professional, 1994, ISBN 0-201-63361-2.

[12] IEEE: IEEE 754: Standard for Binary Floating-Point Arithmetic [online]. http://grouper.ieee.org/groups/754, 2006-11-06 [cit. 2011-04-12].

[13] Intel Corporation: Intel Product Information [online]. http://ark.intel.com/, 2009-05-16 [cit. 2009-05-16].

[14] International Organization for Standardization: Extended BNF [online]. http://standards.iso.org/ittf/PubliclyAvailableStandards/s026153_ISO_IEC_14977_1996(E).zip, 1996-12-15 [cit. 2011-04-12].

[15] Kolektiv autorů: PearPC – Emulátor architektury PowerPC [online]. http://pearpc.sourceforge.net/, 2005-12-20 [cit. 2010-11-20].

[16] 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. 2010-12-27].

[17] Kolektiv autorů: Intel 64 and IA-32 Architectures Software Developer's Manuals, Vol. 2 Instruction Set Reference [online]. http://download.intel.com/design/processor/manuals/253667.pdf, 2009-03 [cit. 2010-12-27].

[18] Kolektiv autorů: MAME – Multiple Arcade Machine Emulator [online]. http://mamedev.org/, 2010-12-23 [cit. 2010-12-27].

[19] Kolektiv autorů: Výzkumný projekt Lissom [online]. http://www.fit.vutbr.cz/research/groups/lissom/index.html, 2010 [cit. 2011-01-08].

[20] Lindholm, T., Yellin, F.: *The Java Virtual Machine Specification, Second Edition.* Prentice Hall PTR, 1999, ISBN 978-0201432947.

[21] Nokia Corporation: Qt framework [online]. http://qt.nokia.com/, 2011 [cit. 2011-04-15].

[22] 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.

[23] W3C: CSS Color Module [online]. http://www.w3.org/TR/css3-color, 2010-10-28 [cit. 2011-04-08].

Dodatek A

Navržené gramatiky

Uvedené návrhy gramatik je také možné nalézt přiloženém na CD v souboru `grammar.txt` ve složce `doc`.

A.1 Gramatika jazyka pro popis architektury

```
simple-stmt ::= object-def      # definiční
              | array-def       # definiční
              | symbol-def      # definiční
              | assignment      # výkonný
              | item-del        # rušící

object-def ::= 'object' ID '=' bits-1d
bits-1d ::= 'bits' bits-range
bits-range ::= '[' expression ']'
             | '[' 'inf' ']'

array-def ::= 'array' ID '=' bits-2d
bits-2d ::= 'bits' bits-range bits-range

symbol-def ::= 'symbol' ID '=' target
target ::= ID | slicing | merging
slicing ::= simple-slicing | extended-slicing
simple-slicing ::= target '[' expression ']'
extended-slicing ::= target '[' expression ':' expression ']'
merging ::= target '.' target

assignment ::= target '=' expression

item-del ::= 'del' del-type ID
del-type ::= 'object' | 'array' | 'symbol' | 'var' | 'operand' | 'alias' |
           | 'procedure' | 'instruction' | 'group' | 'pointer'

expression ::= target                       # pri = 9
             | INT                          # pri = 9
```

```
              | CHAR                          # pri = 9
              | STRING                        # pri = 9
              | sizeof '(' expression ')'     # pri = 9
              | len '('expression ')'         # pri = 9
              | signext '('expression ')'     # pri = 9
              | '(' expression ')'            # pri = 8
              | '-' expression                # pri = 7
              | '~' expression                # pri = 7
              | 'not' expression              # pri = 7
              | expression '/'   expression   # pri = 6
              | expression '%'   expression   # pri = 6
              | expression '*'   expression   # pri = 6
              | expression '+'   expression   # pri = 5
              | expression '-'   expression   # pri = 5
              | expression '<<'  expression   # pri = 4
              | expression '>>'  expression   # pri = 4
              | expression '&'   expression   # pri = 3
              | expression 'and' expression   # pri = 3
              | expression '^'   expression   # pri = 3
              | expression 'xor' expression   # pri = 3
              | expression '|'   expression   # pri = 2
              | expression 'or'  expression   # pri = 2
              | expression '>'   expression   # pri = 1
              | expression '<'   expression   # pri = 1
              | expression '=='  expression   # pri = 0
```

A.2 Gramatika jazyka pro popis instrukční sady

```
simple-stmt ::= variable-def              # definiční
              | exec-proc                 # výkonný
body-simple-stmt ::= operand-def          # definiční
                   | alias-def            # definiční
                   | exec-instr           # výkonný
compound-stmt ::= if-stmt                 # řídící
                | while-stmt              # řídící
body-compound-stmt ::= instruction-def    # definiční
                     | procedure-def      # definiční

variable-def ::= 'var' ID '=' bits
bits ::= bits-1d | bits-2d

operand-def ::= 'operand' ID '=' '{' translate-pair '}'
translate-pair ::= pattern ':' '[' field { ',' field } ']'
pattern ::= STRING
field ::= STRING

alias-def ::= 'alias' ID '=' '[' alias-item { ',' alias-item } ']'
alias-item ::= ID
```

```
exec-instr ::= 'exec' STRING

if-stmt ::= 'if' expression ':' suite [ else-stmt ]
else-stmt ::= 'else' ':' suite

suite ::= suite-stmt NEWLINE
        | NEWLINE INDENT suite-stmt { suite-stmt } DEDENT
suite-stmt ::= simple-stmt NEWLINE
             | compound-stmt

while-stmt ::= 'while' expression ':' suite

instruction-def ::=
        'instruction' ID '(' [ arg-pair { ',' arg-pair } ] ')' ':' suite
arg-pair ::= operand param
operand ::= ID
param ::= ID

procedure-def ::= 'procedure' ID '(' [ param { ',' param } ] ')' ':' suite
exec-proc ::= ID '(' [ par { ',' par } ] ')'
par ::= expression
```

A.3 Gramatika jazyka pro popis grafického zobrazení

```
simple-stmt ::= group-def         # definiční
              | pointer-def       # definiční
              | show-stmt         # výkonný
              | hide-stmt         # výkonný

group-def ::= 'group' ID '=' '[' group-item { ',' group-item } ']'
group-item ::= ID

pointer-def ::= 'pointer' '<' ID '>' ID '=' expression

show-stmt ::= 'show' ID [ at ] [ layout ] [ perspectives ] [ color ]
at ::= 'at' expression ',' expression
layout ::= 'layout' layout-type
layout-type ::= 'horizontal' | 'vertical'
perspectives ::= 'perspectives' '[' perspective { ',' perspective } ']'
perspective ::= 'bin' | 'oct' | 'hex' | 'dec' | 'twoscompl' | 'onescompl'
              | 'float' | 'double' | 'ascii'
color ::= 'color' STRING

hide-stmt ::= 'hide' ID
```

Dodatek B

Instalace a příklad použití

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

B.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 2.6.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 samotné distribuce operačního systému. Pokud tomu tak není, je možné interpret stáhnout z webové adresy `http://www.python.org/download/`.

Dále je nutná instalace grafické knihovny Qt (minimálně verze 4.7.2), nadstavby PyQt4 (testováno na verzi 4.8.3) a SIP generátoru vazeb jazyka Python na C++ knihovny. Instalace pro systémy Windows jsou opět uloženy na CD v adresáři `install`. Z důvodu komplikovanosti instalace na systému Windows je ve stejnojmenném adresáři připojena výsledná aplikace přímo ve spustitelné binární formě (ZIP archiv) nebo ve formě instalačního balíčku MSI. V prostředí UNIX/Linux je možné grafickou knihovnu, nadstavbu a generátor vazeb zpravidla nainstalovat pomocí správce balíčků dané distribuce systému nebo pomocí informací dostupných na webových stránkách `http://qt.nokia.com`, resp. `http://www.riverbankcomputing.co.uk/software/pyqt`.

B.2 Příklad použití

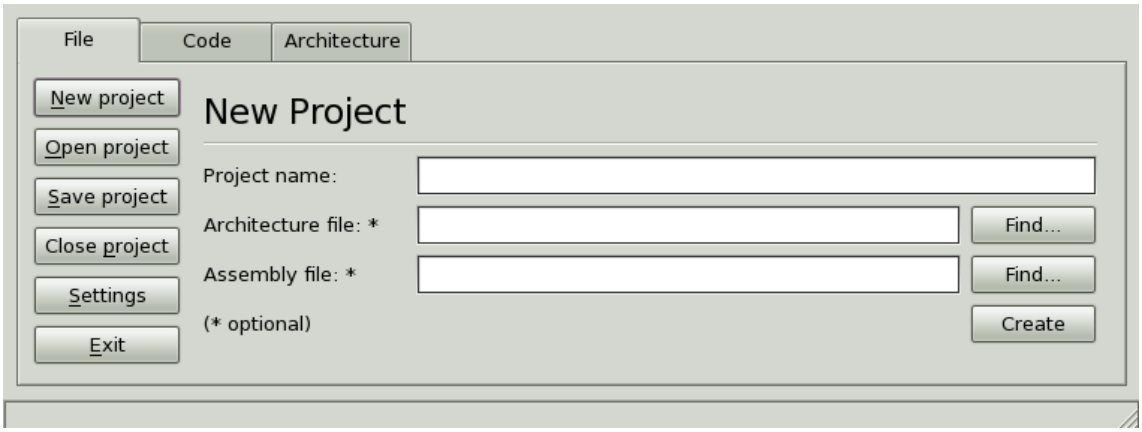
V následujících několika krocích je proveden nástin možného využití emulátoru.

B.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.

B.2.2 Vytvoření projektu

Vytvoření nového projektu se provádí výběrem položky *New project* na záložce *File*. V pravé části okna se objeví možnost založení nového projektu zobrazen na obrázku B.1. V tomto



Obrázek B.1: Dialog založení nového projektu.

dialogu je nutné vybrat soubor s asemblerem emulované architektury (standardně přípona `*.asm`), soubor s popisem architektury (standardně přípona `*.adf`) a nakonec soubor s popisem instrukční sady architektury. Po potvrzení voleb je emulátor připraven pracovat s kódem v jazyce symbolických instrukcí emulovaného procesoru.

B.2.3 Hlavní okno programu

Hlavní okno programu zobrazené na obrázku B.2 se skládá ze dvou částí. Levá je určena pro prohlížení a editaci kódu v jazyce symbolických instrukcí, zatímco pravá slouží k zobrazení aktuálního stavu architektury (v tomto případě Intel x86). Krokování kódu můžeme začít stiskem tlačítka *Step intruction* panelu nacházejícího se v horní části záložky (klávesová zkratka F6). Zelený ukazatel zobrazuje aktuální pozici v rámci programu. Nyní krokováním dalších instrukcí můžeme sledovat jejich dopad na architekturu. Logické vazby mezi prvky se značí barevně (v tomto případě vrchol zásobníku). K rychlejšímu průchodu více intrukcemi je možné využít bodů přerušení (reprezentovány čeveným kruhem). Bod přerušení se vytváří stiskem tlačítka *Toogle breakpoint* (klávesová zkratka F9). Pokud je při automatickém procházení kódu, které je spouštěno pomocí tlačítka *Run* (klávesová zkratka F5) z panelu nástrojů, dosaženo bodu přerušení, běh emulace se zastaví. Pozastavení můžeme dosáhnout i stiskem tlačítka *Stop* (klávesová zkratka F7) z téhož panelu nástrojů. K nastavení restartování emulace je možné využít tlačítko *Restart* (klávesová zkratka F8).

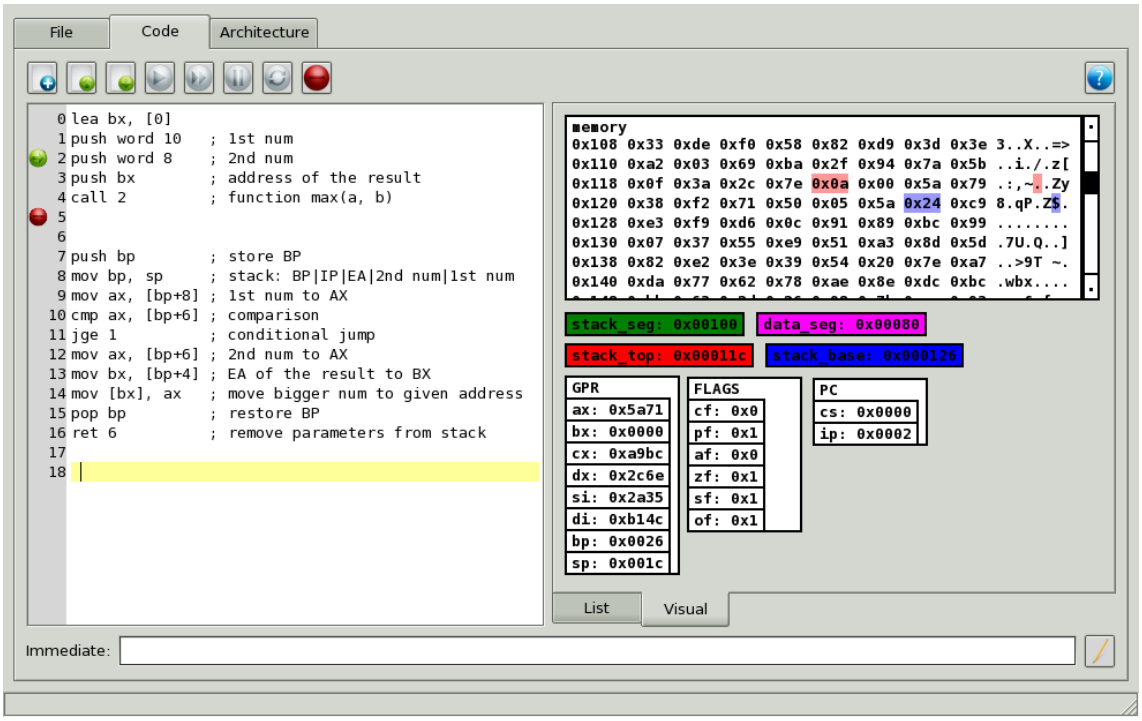
B.2.4 Možnosti zobrazení a správa popisu architektury

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

Záložka *Architecture* umožňuje editaci souboru s popisem architektury. Soubor může být pomocí tlačítka *Reload* (klávesová zkratka F5) znovunahrán do příslušných interpretů.

B.2.5 Uložení a ukončení činnosti

K uložení činnosti provedené v emulátoru je možné použít záložku *File*. Výběrem položky *Save project* je celý projekt uložen, tzn. jsou uloženy pouze přístupové cesty k souborům, se



Obrázek B.2: Práce v hlavním okně programu.

kterými se pracovalo — kód v asembleru, popis architektury spolu s nastavením v záložce settings. Projekt může být načten znovu výběrem položky *Open project*.

Dodatek C

Obsah CD

Kořenový adresář CD obsahuje čtyři podadresáře:

- `install` – Instalace interpretu jazyka Python, knihovny Qt4, grafické nadstavby PyQt4 a generátoru vazeb SIP pro operační systémy Windows 64/32bit.
- `src` – Zdrojové kódy aplikace, vstupní bod programu (soubor `aime.py`) a výstupní soubory z programu Qt Designer pro generování grafického uživatelského rozhraní (`aime.ui`, `DialogAsmHelp.ui`, `DialogArchHelp.ui`), obsahuje další podadresáře:
  - `foundation` – Zdrojové kódy abstraktních datových struktur.
  - `model` – Zdrojové kódy aplikace související s vrstvou `model` návrhu aplikace.
  - `view` – Soubory související s vrstvou `view` návrhu aplikace, grafické uživatelské rozhraní.
  - `controller` – Soubory související s vrstvou `controller` návrhu aplikace, grafické uživatelské rozhraní.
  - `resources` – Další zdroje využívané emulátorem, zejména obrázky použité na ovládacích prvcích grafického rozhraní.
- `doc` – Text této technické zprávy, programová dokumentace a soubor s gramatikou jazyka pro popis architektury.
- `cpu` – Soubory s popisem procesorů, vhodné pro demonstraci možností aplikace.
- `test` – Soubory související s testováním implementace.