VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

SIMULÁTOR ASEMBLERU X86 PRO VÝUKU

X86 ASSEMBLER SIMULATOR FOR EDUCATION

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — **ANDREJ HEŠTERA**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. FILIP ORSÁGH, Ph.D.**
SUPERVISOR

BRNO 2019

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2018/2019

Zadání bakalářské práce

Student: **Heštera Andrej**
Program: Informační technologie
Název: **Simulátor asembleru x86 pro výuku**
x86 Assembler Simulator for Education
Kategorie: Překladače

Zadání:

1. Prostudujte problematiku simulátorů procesorů a seznamte se detailně s 32bitovým strojovým jazykem procesorů z rodiny x86.
2. Navrhněte aplikaci, která bude simulovat a vhodně demonstrovat funkci vybrané množiny instrukcí 32bitového procesoru x86 pro účely předmětu ISU - Programování na strojové úrovni.
3. Navržené řešení implementujte v programovacím jazyce Java.
4. Výslednou aplikaci otestujte z hlediska funkčnosti a zhodnoťte přínos pro studenty (například formou testování aplikace studenty s následnou anketou).

Literatura:

- DUNTEMANN, Jeff. *Assembly language step-by-step: programming with Linux*. 3rd ed. Indianapolis: Wiley, 2009. ISBN 978-0-470-49702-9

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Orság Filip, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 15. května 2019
Datum schválení: 1. listopadu 2018

Abstrakt

Cieľom tejto práce je nadobudnúť potrebné znalosti analýzou architektúry inštrukčnej sady x86 a jazyka symbolických inštrukcií pre navrhnutie a implementovanie simulačného prostriedku v objektovo orientovanom jazyku Java SE8.

Ten používateľovi umožní vytvárať kód založený na konvenciách a syntaxe z prostredia Netwide Assembler a následne daný kód simulovať na virtuálnej reprezentácii – simulačnom modele, ktorý napodobňuje chovanie procesora z rodiny architektúry x86.

Výsledkom by malo byť prehĺbenie znalostí používateľa o principiálnej funkcionalite vykonávaného strojového kódu a to, ako mení stav procesora bez potreby takýto kód kompilovať špeciálnym spôsobom za účelom spustenia cez Debugger, či nutnosťou disponovať fyzickým systémom implementujúcim architektúru x86.

Abstract

Point of this thesis is gain knowledge base of x86 Instruction Set Architecture and x86 assembly language through analysis. Based on this knowledge, design and implement simulation environment in object oriented programming language Java SE8.

This environment will give user option to create code based on conventions and syntax of Netwide Assembler and simulate created code on virtual representation – simulation model, which will imitate behavior of processor implementing instruction set architecture x86.

The result of using this environment should be new knowledge for user about basic function of machine code execution and how this execution alters state of processor, without the need to specially compile created code for use in Debugger and having physical system implementing architecture x86.

Kľúčové slová

Simulátor, Simulácia, Modelovanie, Java, Objektovo orientované programovanie, Netwide Assembler, NASM, asembler, asembler x86, architektúra x86

Keywords

Simulator, Simulation, Modeling, Java, Object oriented programming, Netwide Assembler, NASM, assembler, assembler x86, architecture x86

Citácia

HEŠTERA, Andrej. *Simulátor asembleru x86 pro výuku.* Brno, 2019. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Filip Orságh, Ph.D.

Simulátor asembleru x86 pro výuku

Prehlásenie

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

. . . .
Andrej Heštera
4. mája 2019

Poďakovanie

Chcem poďakovať doktorovi Orságovi za jeho trpezlivosť a zdroje, ktoré mi poskytol v rámci akademického pôsobenia na fakulte.

Obsah

1 Úvod **3**

2 Teoretický základ **5**
2.1 Architektúra procesorov rodiny x86 s ohľadom na tridsaťdva (32) bitovú architektúru . . . 5
2.1.1 Register . . . 6
2.1.2 Pamäťový model a Zásobník . . . 10
2.1.3 Interpretácia dát . . . 12
2.1.4 Spôsob adresovania dát . . . 14
2.1.5 Inštrukcie inštrukčnej sady procesorov rodiny x86 . . . 15
2.2 Jazyk symbolických inštrukcií a Netwide Assembler . . . 16
2.2.1 Vysoko a nízko úrovňové jazyky, strojový kód . . . 16
2.2.2 Netwide Assembler . . . 17
2.3 Modelovanie a Simulácia . . . 18

3 Návrh **19**
3.1 Dátový model . . . 19
3.1.1 Register a Registrový súbor . . . 19
3.1.2 Pamäť a Pamäťový súbor . . . 20
3.1.3 Príznaky a Príznakový súbor . . . 20
3.1.4 Strojový súbor . . . 21
3.2 Interpretácia vstupu . . . 21
3.2.1 Štruktúra spracovania používateľského vstupu . . . 22
3.2.2 Lexikálna a syntaktická analýza . . . 22
3.2.3 Továrne inštrukcií . . . 24
3.2.4 Vnútorná reprezentácia programu . . . 25
3.2.5 Abstrakcia jazyka NASM . . . 26
3.3 Vykonávanie programu . . . 27
3.4 Zobrazovanie dát . . . 29
3.4.1 Editor . . . 29
3.4.2 Simulátor . . . 29

4 Implementácia **32**
4.1 Implementácia modulu strojového súboru . . . 32
4.1.1 PseudoRegister32 a RegisterFile . . . 33
4.1.2 Pamäťový súbor a Mapa prístupov . . . 35
4.1.3 Príznakový súbor a Príznakový bit . . . 36
4.1.4 Inštrukčný ukazovateľ . . . 36

4.1.5 Inicializácia strojového súboru 36
4.2 Implementácia modulu interpretácie 37
4.2.1 Objekt DataBundle 38
4.2.2 Továrne inštrukcií a Hlavná továreň 39
4.2.3 Inštrukcia a Operandy 40
4.2.4 Interpretácia pseudo–inštrukcií a Továreň pamäte 42
4.2.5 Vnútorná reprezentácia programu 43
4.3 Implementácia modulu vykonávania 44
4.3.1 Vstupno Výstupný manažér a zabudované procedúry 45
4.4 Implementácia modulu používateľského rozhrania 46
4.4.1 Obrazovka Editoru 46
4.4.2 Obrazovka Simulátoru 46
4.4.3 Inšpektor dátového modelu 47

5 Testovanie **50**
5.1 Komponentové testovanie 50
5.2 Exploratívne testovanie 51
5.3 Anonymné testovanie dotazníkom 51
5.4 Vyhodnotenie testovania 52

6 Záver **54**

Literatúra **55**

A Obsah priloženého pamäťového média **56**

B Lexikálne a syntaktické pravidlá jazyka *Netwide Assembler* **57**
B.1 Lexikálne pravidlá 57
B.1.1 Numerické hodnoty 57
B.1.2 Symboly a návestia 59
B.1.3 Kontrolné symboly 59
B.1.4 Rezervované identifikátory 59
B.2 Syntaktické pravidlá 59

C Implementované inštrukcie **61**
C.1 Dátové inštrukcie 61
C.2 Zásobníkové inštrukcie 62
C.3 Aritmetické inštrukcie 62
C.4 Posuvné a rotujúce inštrukcie 65
C.5 Logické inštrukcie 68
C.6 Porovnávacie inštrukcie 69
C.7 Riadiace a skokové inštrukcie 70
C.7.1 Rodina inštrukcií `Jcc` 71

D Implementované procedúry **74**
D.1 Procedúry pre používateľský vstup 74
D.2 Procedúry pre výstup skalárnej hodnoty 75
D.3 Procedúry pre výstup vektorovej hodnoty 75

Kapitola 1

Úvod

Cieľom tejto práce je predstaviť čitateľovi detailnú analýzu prostredia programovania v jazyku symbolických inštrukcií pre procesory rodiny x86. Na základe tejto analýzy (s prihliadnutím ku modelovaniu a simulácii) bude následne navrhnutý a implementovaný systém, ktorý umožní používateľom skúmať vykonávanie programu (napísanom v jazyku *Netwide Assembler*) v simulovanom prostredí.

Dôvodom pre túto prácu bolo zjednodušenie pochopenia tejto problematiky, nakoľko je nízko-úrovňové programovanie kriticky významné hneď z niekoľkých uhľov pohľadu:

1. *Optimalizácia.* Ak je programátor schopný akceptovať to, že program napísaný v ľubovoľne sofistikovanom programovacom jazyku je v konečnom dôsledku interpretovaný na postupnosť strojových inštrukcií, bude schopný do svojho kódu vnášať optimalizačné prvky, či pristupovať k samotnému návrhu s ohľadom na pamäťovú/časovú náročnosť.

2. *Bezpečnosť.* Už programovanie v jazyku *C* nabáda k disciplíne pri práci s pamäťou a dátami. Programovanie s extrémne nízkou abstrakciou ukazuje ako môžu mať neošetrené, hazardné stavy drvivý dopad na integritu dát, či stabilitu samotného systému, ktorého prostriedky program využíva.

3. *Kontrola.* Programy vytvorené priamo v jazyku cieľovej architektúry umožňujú programátorovi prevziať takmer dokonalú kontrolu nad systémom. V prípade, ak sa operuje v prostredí, kde neexistuje operačný systém zabezpečujúci riadenie prístupov k prostriedkom (napríklad vstavané systémy s mikrokontrolérom) je táto kontrola totálna. Programátorovi otvárajú možnosti pre optimalizáciu a zvýšenie bezpečnosti návrhu bez potreby spoliehať sa na kompilátor z vyššieho programovacieho jazyka.

Programovanie v jazyku symbolických inštrukcií však prináša svoje úskalia, nakoľko odstraňuje pohodlné štruktúry vyšších programovacích jazykov, čitateľnosť a transparentnosť kódu. To môže pre programátora, ktorý s touto problematikou predtým neprišiel do styku, predstavovať výraznú prekážku. Aplikačné riešenie, ktoré je výsledkom tejto práce, umožňuje tieto problémy premostiť.

Aplikácia vytvorená vo vyššom programovacom jazyku, využívajúc *Netwide Assembler* ako model pre vstupný kód, toto riešenie umožňuje rýchlo a efektívne vykonávať experimenty. Prostriedky tohoto riešenia sú však obmedzené na základné inštrukcie a riadiace štruktúry, nakoľko by plnohodnotná simulácia asembleru pre procesory rodiny x86 predstavovala výzvu prevyšujúcu rozsah tejto práce.

Cieľom je prekonať prvotné problémy s nastavovaním prostredia (prípravok vyžaduje len virtuálny stroj jazyka Java) a umožňuje sledovať zmeny nad modelom predstavujúcim prostriedky procesora a operačnú pamäť. Toto má význam, nakoľko dosiahnutie obdobného riešenia s využitím, napríklad nástroja `GDB`, vyžaduje špeciálnu formu kompilácie a prirodzene, fyzický systém obsahujúci procesor rodiny x86.

Tento prípravok však nemá nahradiť reálne experimenty a učenie s fyzickým systémom. Má predstavovať prostredie pre rýchlu demonštráciu základných rysov systému a chovania pre potreby vzdelávania a úvodu do problematiky programovania v nízko úrovňových programovacích jazykoch.

Kapitola 2

Teoretický základ

Cieľom tejto kapitoly je analýza existujúceho systému (ako systému fyzického, teda využívajúceho architektúry x86, tak aj teoretických systémov, ako je štruktúra programovacieho jazyka Netwide Assembler), ktorý bude nutné modelovať a simuláciou umožniť vykonávanie používateľského programu nad týmto modelom. Kombinovaným výstupom tejto kapitoly teda bude vedomostná báza, ktorá poslúži ako základ pre návrh aplikačného riešenia.

Pre úspešnú analýzu je nutné prejsť tromi základnými celkami a to:

1. Architektúra procesorov x86 a ich prostriedkov
2. Programovací jazyk symbolických inštrukcii Netwide Assembler (NASM) a programovanie na strojovej úrovni
3. Modelovanie a Simulácia

Kombinácia týchto troch celkov predstavuje znalosti potrebné pre navrhnutie aplikačného riešenia. Vzhľadom k objemu informácií týkajúcich sa týchto oblastí bude súčasťou kapitoly aj zvolenie vhodnej abstrakcie, teda vyňatia kritických súčastí systému a zdôvodnenia toho, ako bol model pozmenený voči reálnemu systému.

2.1 Architektúra procesorov rodiny x86 s ohľadom na tridsaťdva (32) bitovú architektúru

Architektúra, alebo inštrukčná sada (anglicky tiež aj *Instruction set*) predstavuje abstrakciu reálneho, fyzického systému (takáto aplikácia architektúry je tiež známa aj ako *implementácia*). Abstrakcia obsahuje opis náležitostí fyzického systému tak, ako k nim môže pristupovať (používať) programátor a výpočet dostupných prostriedkov, ktoré je možné v rámci vytvárania programu pre cieľovú architektúru využiť. Napriek tomu, že sa môžu konkrétne hardvérové realizácie líšiť, základne jadro opisujúce chovanie, inštrukcie, interpretácie a prácu s dátami je vďaka tejto abstrakcii zhodné.

Analýza architektúry teda bude z hľadiska tejto práce primárne zameraná na:

- Register
- Pamäťový model
- Interpretácia dát

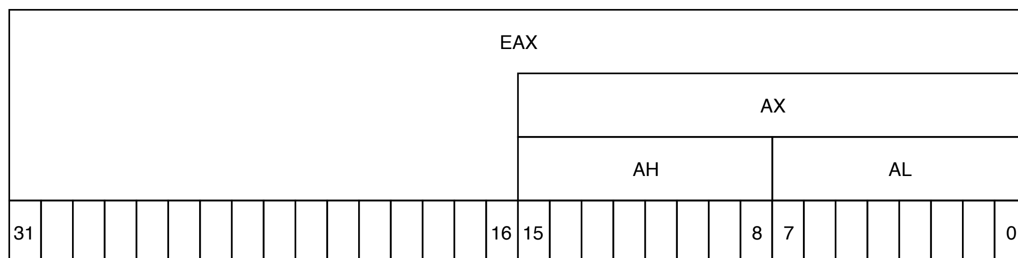
- Spôsob adresovania
- Inštrukcie a Vykonávanie

2.1.1 Register

Register je primárnou súčasťou systému a ako taká umožňuje na najnižšej úrovni manipuláciu dát, ktoré reprezentujú hodnotnú informáciu. Táto manipulácia predstavuje ukladanie, načítanie a transformáciu dát, ktorá je spojená s chovaním systému. Základná funkcia registra je teda štruktúra, ktorá uchováva dáta vo forme binárneho vektora (pričom dĺžka tohto vektora je určená konkrétnou implementáciou registra), k týmto dátam je možné pristupovať pomocou jednoznačného identifikátora registra.

Je vhodné zmieniť, že napriek tomu, že systém obsahuje veľké množstvo jednoznačných identifikátorov, niektoré z týchto identifikátorov sa odkazujú na jednu hardvérovú inštanciu registra, avšak na rozličné rozsahy binárneho vektora obsiahnutom v tomto komponente.

Registre sú z hľadiska architektúry x86 využívané nielen pre všeobecnú prácu s užitočnými dátami (ako napríklad aritmetické, logické operácie a iné), ale taktiež pre reprezentáciu a kontrolu systému (príznakový register), zdroj kalkulácie adries (segmentové registre), riadenie prístupov do pamäte pre potreby obsluhy zásobníka (zásobníkový ukazovateľ na vrchol a bázu) a riadenie toku programu (inštrukčný čítač).



Obr. 2.1: Reprezentácia zdieľaných bitových šírok asociovaných s pomenovaním konkrétnych šírok. V obrázku je najmenej významný bit umiestnený vpravo.

Architektúra x86 (s ohľadom na tridsaťdva (32) bitovú architektúru) poskytuje registre veľkosti tridsaťdva (32) bitov. Ako bolo zmienené vyššie, tieto registre je však možné pristupovať s odlišnou *zrnitosťou*. Akumulátor, teda register EAX, ktorý má rozsah tridsaťdva (32) bitov je možné pristupovať aj ako AX (teda šestnásť (16) bitová reprezentácia), alebo AH, či AL, ktoré predstavujú horný a dolný bajt (kde bajt predstavuje osmicu bitov). Je dôležité zmieniť, že nie všetky registre poskytujú túto možnosť *zrnitosti* prístupov k rôznym rozsahom. Toto chovanie je demonštrované obrázkom[2.1].

Všeobecné registre

Všeobecné registre predstavujú registre, ktoré sú primárne využívané programátorom pre potreby modelovania a realizácie chovanie programu. Architektúra poskytuje osem (8) všeobecných registrov, ktoré sú poskytnuté pre voľné využitie, avšak z hľadiska návrhu systému

sa ku konkrétnym registrom viažu špecifické spôsoby využitia. Pre opis registrov je použitá šestnásť (16) bitový spôsob pomenovania[3, zv. 1, kap. 3, sek. 4.1].

- `AX` : Akumulátor
- `BX` : Bázový register
- `CX` : Počítadlo
- `DX` : Dátový register
- `SI` : Ukazovateľ na zdroj
- `DI` : Ukazovateľ na cieľ
- `BP` : Ukazovateľ na bázu zásobníka
- `SP` : Ukazovateľ na vrchol zásobníka

Prípady špecifického využitia primárne vychádzajú zo spôsobu, akým architektúra definuje chovanie jednotlivých inštrukcií poskytovaných v rámci svojej inštrukčnej sady (a taktiež rozdielmi, ktoré vychádzajú z abstrakcie jazyka symbolických inštrukcií a reálneho strojového kódu, opísané nižšie). Z tohto hľadiska je teda možné definovať jednotlivé špecifikácie použitia registrov ako:

- `Akumulátor` : Primárne využívaný ako univerzálny register, register indikujúci návratovú hodnotu funkcie/programu či zdroj pre aritmetické operácie.
- `Počítadlo` : Register, ktorý je využívaný pre kontrolu programových slučiek a cyklického chovania určitých inštrukcií. Tento register teda poskytuje funkcionalitu iterátora.
- `Dátový register` : Využitie tohto registra je primárne spojené s aritmetickými operáciami (ako napríklad násobenie, či delenie), ktorých výsledky môžu presiahnuť originálne rozsahy vstupných operandov (z hľadiska binárnej reprezentácie hodnôt), alebo sú výsledkom operácie dve hodnoty (v prípade celočíselného delenia výsledok a zvyšok po delení).
- `Ukazovatele na zdroj a cieľ` : Registre využité v rámci operácií pracujúcich s ukazovateľmi na dáta v pamäti. Tieto registre poskytujú na rozdiel od vyššie zmienených len prístup k tridsaťdva (32) a šestnásť (16) bitovým rozsahom.
- `Ukazovatele na Bázu a vrchol zásobníka` : Táto dvojica registrov poskytuje zdroj adries pre inštrukcie obsluhujúce prístupy do abstraktnej dátovej štruktúry realizovanej v pamäti systému, ktorá slúži ako zásobník. Rovnako ako predošlá dvojica ukazovateľov, aj tieto umožňujú len tridsaťdva (32) a šestnásť (16) bitové prístupy.

Všeobecné registre predstavujú kritickú funkcionalitu architektúry x86 a preto bude ich funkcionalita čo najpresnejšie modelovaná.

Segmentové registre

Segmentové registre veľkosti šestnásť (16) bitov slúžia procesoru pre riadenie prístupov do pamäte[5, kap. 4, str. 91]. Na základe zvoleného modelu pamäte tieto registre slúžia k výpočtu reálnej adresy v sprístupnenej pamäti. Vzhľadom k vývoju v oblasti nízko-úrovňového programovania je však ich užívanie na ústupe a nahradzované riadením pamäte, ktoré programátorovi neumožňuje manipulovať s obsahom týchto registrov (viac o tejto problematike v sekcii zaoberajúcou sa analýzou Pamäťového modelu).

V súčasnej generácii operačných systémov je obsah segmentových registrov jednotný, ukazujúci na jednu konkrétnu adresu. Špeciálnym prípadom je práca s viac vláknovými procesmi, kedy sa špecifický segmentový register (*FS* a *GS*) používa pre ukladanie adresy náležiacej konkrétnemu vláknu.

Architektúra x86 definuje tieto segmentové registre:

- *CS* : Kódový Segment
- *DS* : Dátový Segment
- *SS* : Zásobníkový segment
- *ES* : Extra dátový segment
- *FS* : Extra (F nasleduje po E) dátový segment
- *GS* : Extra (G nasleduje po F) dátový segment

Vzhľadom k ich využívaniu v zvolenom spôsobe, akým bude modelovaná pamäť budú teda tieto registre opomenuté.

Príznakový register

Príznakový register je súčasťou systému, ktorého chovanie je nadstavbou klasického všeobecného registra a je kriticky významný pri riadení vykonávania a vyhodnocovaní rozhodovacej logiky programu[3, zs. 1, kap. 3, sek. 4.3]. Príznakový register obsahuje bity, ktorých hodnota nadobúda dvoch stavov:

- 0, teda nepravda, FALSE
- 1, teda pravda, TRUE

Primárne sú tieto bity nastavované vykonávaním programu, konkrétne špecifickými inštrukciami, ktoré majú možnosť ovplyvňovať stav konkrétnej hodnoty, to reprezentuje situácie, ktoré počas vykonávania nastali. Tieto bity sú potom kľúčové pri vyhodnocovaní výsledkov operácii a na to nadväzujúce riadenie toku a chovania programu[5, kap. 12, str. 212]. Obsah registra je taktiež možné nastaviť pomocou špeciálnych inštrukcií hodnotou definovanou programátorom (alebo využiť inštrukcií pre zmenu jedného konkrétneho príznaku), avšak nie je možné obsah registra meniť priamo (napríklad inštrukciou *MOV*).

Príznak samotný je definovaný ako dvojica špecifického indexu bitu v registry a kľúčovým označením.

Príznakový register inštrukčnej sady x86, architektúry 32 bitov obsahuje nižšie opísané príznaky. Sú označené trojicou index bitu, triviálne označenie/kľúč a názov. V prípade ak je kľúč XX, predstavuje nepomenovaný index.

- 00 : CF : Carry Flag
- 01 : XX : Tento bit je vždy nastavený na 1
- 02 : PF : Parity Flag
- 03 : XX : Tento bit je vždy nastavený na 0
- 04 : AF : Adjust/Auxiliary Flag
- 05 : XX : Tento bit je vždy nastavený na 0
- 06 : ZF : Zero Flag
- 07 : SF : Sign Flag
- 08 : TF : Trap Flag
- 09 : IF : Interrupt Flag
- 10 : DF : Direction Flag
- 11 : OF : Overflow Flag
- 12–13 : IOPL : I/O Privilege Flag
- 14 : NT : Nested Task Flag
- 15 : XX : Tento bit je vždy nastavený na 0
- 16 : RF : Resume Flag
- 17 : VM : Virtual–8086 Mode
- 18 : AC : Alignment Check
- 19 : VIF : Virtual Interrupt Flag
- 20 : VIP : Virtual Interrupt Pending Flag
- 21 : ID :Identification Flag
- 22–31 : XX : Tento bit je vždy nastavený na 0

Tieto príznaky je možné rozdeliť do dvoch kategórií na základe ich významu a to:

- *Stavové* CF, PF, AF, ZF, SF a OF
- *Systémové a Riadiace* IOPL, NT, RF, VM, AC, VIF, VIP, ID, TF, IF, a DF

Rovnako ako pri všeobecných registroch, príznakový register je kriticky dôležitý pre pochopenie funkcie fyzického systému a tak bude nutné vytvoriť jeho hodnoverný model. Pre potreby špecifického modelu však budú primárne modelované príznaky *CF*, *PF*, *ZF*, *SF* a *OF*, nakoľko tieto sú využívané na riadenie chovania programu definovaného programátorom a reflektovanie stavu systému po vykonaní operácií úzko súvisiacich s dátami a to:

- *Carry Flag*, teda príznak prenosu do vyššieho rádu, alebo príznak výpôžičky z vyššieho rádu pri aritmetických operáciach. Carry Flag je indikátorom aritmetickej chyby v operáciach, ktoré pracujú s hodnotami so znamienkom. Taktiež je tento príznak využívaných pri operáciach, ktoré súvisia s manipuláciu dát rotáciou, posunom a násobením.[4]
- *Parity Flag*, teda príznak binárnej parity indikuje, či je je počet kladných bitov vo výsledku párny (ak je 1), alebo nepárny (ak je 0).
- *Zero Flag*, teda nulový príznak, ktorý indikuje, či je výsledkom operácie nula (0).
- *Sign Flag*, teda príznak znamienka výsledku operácie, ktorý indikuje stav najviac významného (teda najvyššieho) bitu operácie. Tento bit je v doplnkovom kóde jednoznačným identifikátorom toho, či je číslo kladné, alebo záporné.
- *Overflow Flag*, teda príznak pretečenia, ktorý indikuje, že výsledok operácie presiahol rozsah miesta pre uloženie výsledku a tým pádom je hodnota takto uložená neplatná, alebo nesprávna z hľadiska aritmetiky bez znamienka.[4]

Inštrukčný čítač

Inštrukčný čítač, je registrom, ktorý v sebe uchováva hodnotu ukazujúcu na inštrukciu, čo sa má v nasledujúcom cykle vykonať. Z tohoto hľadiska teda tento register slúži ako primárny kontrolný prvok riadenia programu, nakoľko transformácie nad informáciami uchovávanie v tomto registry ovplyvňujú tok programu.[3, zv. 1, kap. 3, sek. 5]

Pre potreby aplikačného riešenia je nutné modelovať toto chovanie, avšak s adekvátnou abstrakciou, nakoľko cieľom nie je korektne a hodnoverne zobraziť dekódovanie a spracovanie nasledujúcej inštrukcie, ale len zohľadniť dopad zmien nad týmto registrom. Napríklad zmena vyvolaná skokovou inštrukciou, či vyvolaním procedúry.

2.1.2 Pamäťový model a Zásobník

Architektúra poskytuje prístup k prostriedku, ktorý je definovaný ako pamäť. Z hľadiska významu táto pamäť predstavuje blok informácií, ktorý je na základe kontextu spracovávaný ako dáta pre operácie, alebo ako inštrukcie vykonávaného programu. Prístup k pamäti je realizovaný samotným procesorom, ktorý poskytuje rozhranie služieb prístupu pre operačný systém a program.[7, kap. 2, sekcia 2.3]

Takto realizovaná pamäť predstavuje sekvenciu za sebou nasledujúcich bajtov, z ktorých každý je adresovateľný unikátnou adresou.

Z hľadiska programov obsluhovaných procesorom je však pamäť v súčasnosti používaná vo forme jej *virtuálnej reprezentácie*[3, zv. 1, kap. 3, sek. 3.1], teda transparentnej abstrakcie *fyzickej pamäte*, ktorá je reprezentovaná ako kontinuálny blok užitočnej pamäte. Procesor samotný zabezpečuje túto virtuálizáciu pomocou využívania *pamäťového modelu reálnych adries* a *stránkovania*[3, zv. 1, kap. 3, sek. 3.2]. Architektúra taktiež disponuje dvomi ďalšími modelmi pamäte, konkrétne *modelom plochého rozloženia pamäte* a *segmentovaným modelom pamäte*.[5, kap. 4, str. 101]

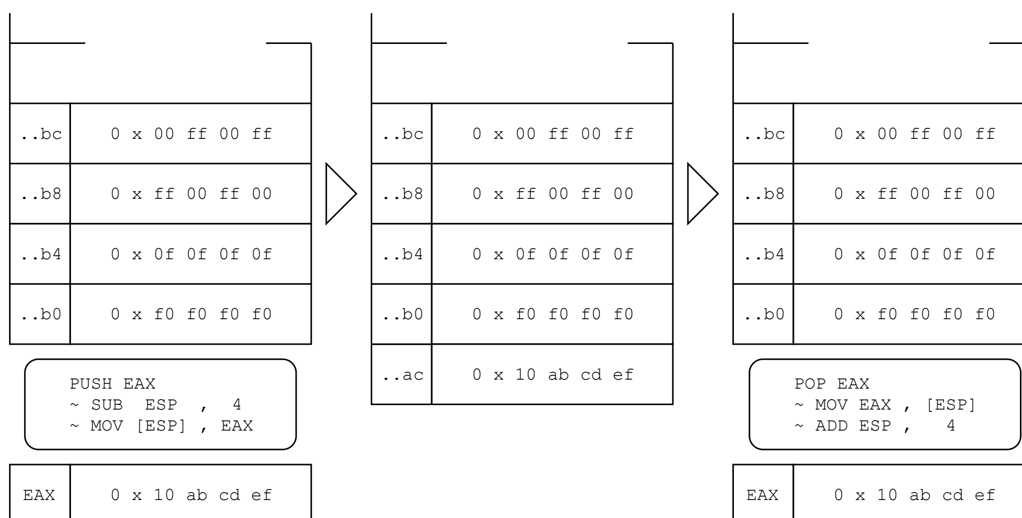
Táto technológia umožňuje programom zaistenie konzistentného, bezpečného a spoľahlivého prístupu do pamäte. Ako bolo vyššie spomenuté, toto je docielené pomocou systému stránkovania, kedy procesor na základe požiadaviek programu o prístup do pamäte načítava z fyzickej pamäte stránky a vystavuje ich pre potreby načítania a ukladania dát

bez akejkoľvek réžie (teda potreby priameho riadenia hardvéru programátorom) na strane používateľom vytvoreného programu.[3, zv. 1, kap. 3, sek. 3.1]

Z tohto dôvodu bude preto aj v rámci simulácie pamäť modelovaná ako jej virtuálna reprezentácia, teda lineárny adresovateľný blok pamäte.

Zásobník

Zásobník je abstraktnou štruktúrou realizovanou pomocou vnútorných prostriedkov architektúry x86, ktorá sa nachádza uložená v pamäti. Tento systém vychádza z predpisu chovania svojej abstraktnej dátovej štruktúry, teda umožňuje na základe implementovaných inštrukcií pomocou chovania *LIFO* (*Last In First Out*, Posledný Dnu Prvý Von) umožniť prístup k dátam uloženým v zásobníku[[7, kap. 5, sek. 1, str. 140]. Chovanie zásobníku pre operácie *PUSH* (vloženie novej informácie) a *POP* (vybratie poslednej uloženej informácie) je demonštrované na obrázku[2.2].



Obr. 2.2: Postupnosť vloženia a vybratia prvku zo zásobníka.

Zásobník, ako komponent poskytovaný architektúrou programátorovi, je kriticky dôležitý pre používateľom definované riadenie chovania programu, primárne pri využívaní zásobníka ako nástroja na odovzdávanie parametrov pri volaní procedúr. Mimo toho je taktiež zásobník použitý ako zdroj návratovej adresy pri vyvolaní konkrétnej procedúry (inštrukcie *CALL* a *RET*).

Operácia zásobníku je realizovaná prostredníctvom dvoch množín inštrukcií a to inštrukcií pre ukladanie dát na zásobník (rodina *PUSH*) a inštrukcií pre extrakciu dát (rodina *POP*). Správa prístupov na zásobník je podporovaná dvojicou registrov, konkrétne registrov *SP* a *BP*, kedy register *SP* slúži ako zdroj adresy na vrchol zásobníku v pamäti a *BP*, ktorý programátor môže využiť pre definovanie zásobníkového rámca (teda bázy, či dna zásobníku). Z hľadiska pamäťovej organizácie je vhodné zmieniť, že zásobník v pamäti *rastie* smerom *dole*, teda adresy vrcholu zásobníka sú každým uložením nových dát znižované o šírku dát (tridsaťdva (32), alebo šestnásť (16) bitov). Tieto veľkosti sú definované v architektúre.

2.1.3 Interpretácia dát

Inštrukčná sada a architektúra definuje akým spôsobom systém spracováva dáta. Vo svojom základe sú všetky takéto informácie chápané ako binárne vektory rôznych dĺžok, ktoré systém na základe požiadaviek programu interpretuje ako numerické hodnoty. Z tohto dôvodu architektúra predstavuje dátové typy, teda spôsoby, akým je možné zakódovať rôzne formy dát ako vyššie zmienené binárne reprezentácie[[3, zv. 1, kap. 4, sek. 2].

Prvou definíciou sú veľkosti, ktoré je možné aplikovať na dáta. Pre modelovanú architektúru sú definované tri základné veľkosti a to *BYTE* (8 bitov – 1 bajt), *WORD* (16 bitov – 2 bajty) a *DOUBLEWORD* (32 bitov – 4 bajty).[2, kap. 3]

Spôsob interpretácie binárneho vektoru špecifickej dĺžky ako konkrétnej numerickej hodnoty je podmienené existenciou definície dátového typu. Pre potreby simulácie budú spracované tri základné dátové typy a to:

- Celé čísla bez znamienka
- Celé čísla so znamienkom
- Reťazce symbolov (string)

Ostatné dátové typy, ako čísla s pohyblivou desatinnou čiarkou, BCD (*Binary–coded decimal*, teda binárne kódované desiatkové číslo), celočíselné vektory a iné, definované v technickej dokumentácii, budú v rámci tejto simulácie opomenuté.

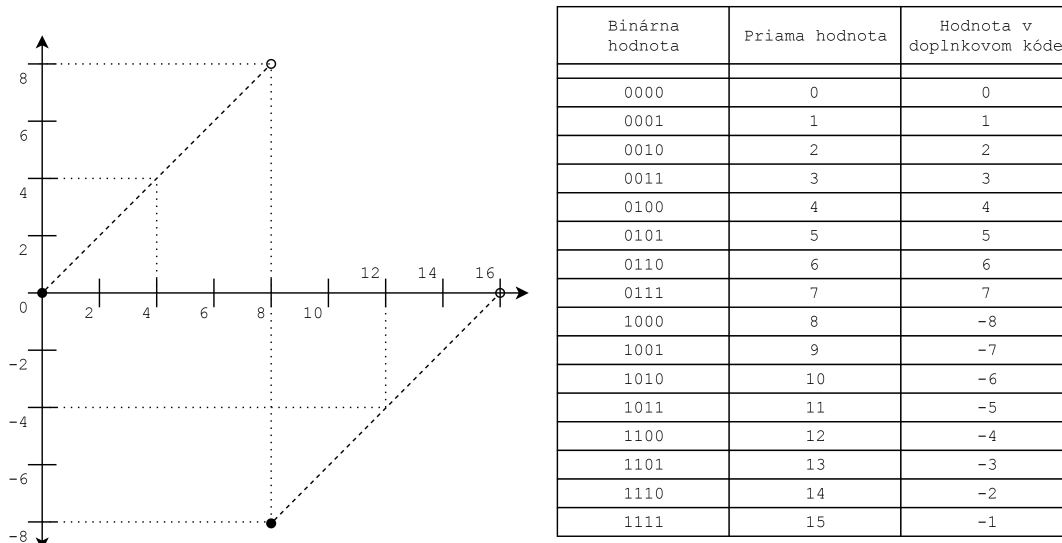
Doplnkový kód

Doplnkový kód, alebo tiež kódovanie dvojkového doplnku, je spôsob, akým systém dokáže mapovať na súbor priamych hodnôt hodnoty so znamienkom. Demonštrácia tohoto mapovania je viditeľná na obrázku[2.3]. Takto zvolený spôsob umožňuje zachovanie konzistencie vybraných matematických operácií, ktorých chovanie teda nevyžaduje špecifickú úpravu spôsobu, akým je výpočet vykonávaný. Tento návrh taktiež vytvára možnosť výskytu aritmetickej chyby, kedy užitočná reprezentácia hodnoty výsledku presiahne rozsah operácie. Nakoľko nie je možné určiť, či programátor zamýšľal použiť priamu, či doplnkovú reprezentáciu, systém disponuje indikátormi, ktoré indikujú vznik potencionálnej chyby (špecificky *Carry Flag* pre operácie so znamienkom a *Overflow Flag*, pre operácie bez znamienka).

Z ďalších vlastností tohto spôsobu kódovania je to, že ak poznáme veľkosť hodnoty, tak vieme vždy a jednoznačne určiť, či je hodnota kladná, alebo záporná podľa hodnoty najviac významného bitu (v klasickej reprezentácii binárnej hodnoty teda najľavejší/najvyšší bit). Ak je tento bit jedna, hodnota je záporná, ak je bit nula, hodnota je kladná.

Algoritmus kódovania[6] priamej hodnoty do doplnkového kódu je nasledovný (požiadavkou na úspešné zakódovanie je znalosť veľkosti výsledku, tu je demonštrácia pre 8 bitovú hodnotu):

1. Cieľom je binárna reprezentácia čísla *-13* doplnkovým kódom.
2. Hodnota *13* ako binárny vektor: *0000_1101*.
3. Invertovanie hodnoty bitov: *1111_0010*.
4. K hodnote je pripočítaná jednotka: *1111_0011*.
5. Výsledná reprezentácia čísla *-13* je *1111_0011*.



Binárna hodnota	Priama hodnota	Hodnota v doplnkovom kóde
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	-8
1001	9	-7
1010	10	-6
1011	11	-5
1100	12	-4
1101	13	-3
1110	14	-2
1111	15	-1

Obr. 2.3: Korelácia priamej hodnoty a hodnoty odvodenej pomocou doplnkového kódu.

Algoritmus určenia celočíselnej hodnoty čísla zakódovaného doplnkovým kódom (požiadavkou na úspešné zakódovanie je znalosť veľkosti výsledku, tu je demonštrácia pre 8 bitovú hodnotu):

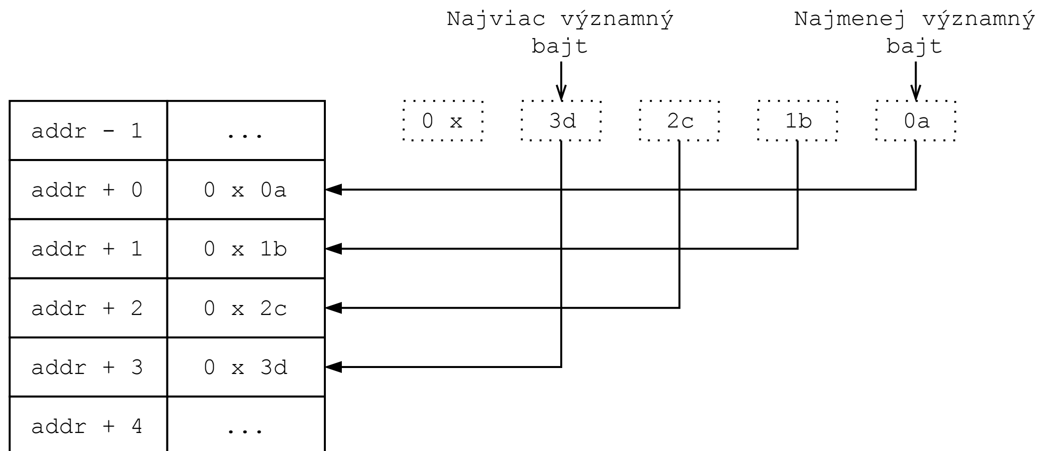
1. Cieľom je celočíselná hodnota so znamienkom bitového vektoru *1111_1001*.
2. Ak bitový vektor začína nulou, vykonáme len priame prevedenie z dvojkovej bázy do desiatkovej, nakoľko je číslo kladné.
3. Ak bitový vektor začína jednotkou, invertuje sa: *0000_0110*.
4. K hodnote je pripočítaná jednotka: *0000_0111*.
5. Výsledná hodnota binárnej reprezentácie je -7.

Little–Endian

Vzhľadom k práci s dátami uloženými v pamäti využíva architektúra x86 spôsob, ktorý sa nazýva *Little–Endian*. Vo svojej podstate ide o zmenu poradia slabík pri ukladaní do pamäte z dôvodu, že na cieľovej adrese sa vždy ako prvá nachádza najmenej významná slabika (na rozdiel od *Big–Endian*, kde je to najviac významná slabika).

Významom tohto je zaistenie, že pri zmene veľkostí operácií pre načítanie z pamäte bude zachované, že sa hodnoty budú zhodovať v najmenej významnej slabike. Tým pádom nedôjde k strate užitočnej informácie, ak obslužný kód pracuje v rozmedzí šestnástich (16) bitov, zatiaľ čo adresu na zdroj a zdrojové dáta vytvoril kód využívajúci tridsaťdva (32) bitové operácie.

Toto chovanie je demonštrované na obrázku[2.4].



Obr. 2.4: Organizácia slabík pomocou *Little–Endian*.

2.1.4 Spôsob adresovania dát

Architektúra x86 opisuje spôsoby toho, ako môže byť pristupované k dátam, ktoré slúžia ako zdroje informácii pre inštrukciu, ktorá je vykonávaná systémom. V základnom princípe môžu byť štyri (4) zdroje dát, ktoré definujeme ako *operandy.*[3, zv. 1, kap. 3, sek. 7] Konkrétne sú to tieto:

- Okamžitá (*Immediate*) hodnota, ktorá je súčasťou inštrukcie samotnej. Napr.: *14*
- Register, kedy je hodnota uložená v registry. Napr.: *EAX*
- Pamäť, kedy je hodnota uložená na konkrétnej adrese v pamäti. Napr.: *[symbol_adresy]*
- Vstupno–výstupný port.

S ohľadom na dostupné prostriedky simulácie budú pre návrh vybrané len prvé tri spôsoby adresovania dát, teda operandy pre okamžitú, registrovú a pamäťovú hodnotu.

Na základe architektúry x86[3, zv. 1, kap. 3, sek. 5], je adresovanie do pamäte realizované dvomi spôsobmi a to priamym deklarovaním adresy a nepriamym opísaním adresy prostredníctvom ofsetu voči špecifickej adrese, čo je možné urobiť ako kombináciu štyroch premenných:

- Odsadenie (*Displacement*)
- Báza (*Base*)
- Index (*Index*)
- Mierka (*Scale factor*)

Nepriamu adresu je teda možné interpretovať ako rovnicu:

$$Báza + (Index * Mierka) + Odsadenie$$

kde:

- *Báza*, registrová hodnota, slúži ako základ adresy
- *Index*, registrová hodnota, slúži ako ukazovateľ na konkrétny prvok, napríklad pri prechádzaní poľom
- *Mierka*, okamžitá hodnota, 1, 2, alebo 4, ktorá sa násobí s indexom pre korektnú úpravu odsadenia od počiatočnej adresy, napríklad pri prechádzaní poľom, slúži pre zohľadnenie veľkosti elementu poľa
- *Odsadenie*, 8, 16, alebo 32 bitová hodnota, táto slúži ako okamžitá hodnota odsadenia voči originálnej adrese

Na základe tejto definície je možné teda voliť niekoľko kombinácii, z ktorých všetky sú kalkulované ako vyššie zmienená rovnica, pričom ak chýba niektorá z hodnôt, je automaticky dosadená implicitná hodnota (špeciálnou výnimkou je mierka, ktorá musí byť použitá v kombinácii s indexom). Klasické spôsoby kalkulácie adresy sú:

- *Odsadenie* : Predstavuje priame, nevypočítané odsadenie voči začiatku pamäti a býva zakódované priamo v samotnej strojovej inštrukcii. Tiež aj označované ako absolútna, alebo statická adresa, použitá k prístupu staticky alokovaných skalárnych hodnôt.
- *Báza* : Nakoľko je adresa uložená v registry, jej hodnota je tým pádom dynamická a môže sa v priebehu vykonávania programu meniť.
- $Báza + Odsadenie$: Využité v špecifických prípadoch a to k pristúpeniu špecifického prvku poľa, alebo k špecifickej položke štrukturovaných dát v pamäti.
- $(Index * Mierka) + Odsadenie$: Efektívny spôsob k dynamickému prístupu ku konkrétnemu prvku staticky definovaného poľa v pamäti, kde mierka predstavuje veľkosť prvku v bajtoch a index poradie prvku.
- $Báza + Index + Odsadenie$: Spôsob, kedy je adresovanie realizované prostredníctvom dvoch registrov, využitie pre prechod dvojrozmerným poľom (kde odsadenie predstavuje začiatok poľa), alebo prechodu poľom záznamov (kde odsadenie umožňuje prístup ku konkrétnej položke určitého záznamu).
- $Báza + (Index * Mierka) + Odsadenie$: Plný výraz je použitý k efektívnemu prechodu dvojrozmerným poľom, ak sú jeho prvky veľkosti dva (2), alebo štyri (4) bajty.

S ohľadom na význam tohto v programovaní (prechádzanie poľom, pristupovanie k záznamom a pod.) je teda nutné v rámci návrhu simulácie umožniť používateľovi takto definovať prístupy do pamäte.

2.1.5 Inštrukcie inštrukčnej sady procesorov rodiny x86

Inštrukcie predstavujú primárny spôsob na opis chovania programu. Samotná inštrukcia predstavuje operáciu, ktorú je procesor schopný vykonať nad poskytnutými dátami a výsledok tejto operácie. Z tohto hľadiska je teda možné povedať, že inštrukcia sa skladá z definície vstupných dát, alebo operandov inštrukcie, z jednoznačného identifikátora inštrukcie, z opisu chovania inštrukcie a zo spôsobu, ako je naložené s výsledkom operácie.

Proces vykonania inštrukcie strojového kódu pozostáva zo štyroch (4) základných krokov a to:

1. *FETCH* : získanie nasledujúcej inštrukcie pomocou adresy uloženej v inštrukčnom čítači. Táto inštrukcia je následne uložená do inštrukčného registra. Ukončenie tohto kroku spôsobí nastavenie inštrukčného čítača na adresu nasledujúcej inštrukcie po aktuálne načítanej.
2. *DECODE* : dekódovanie numerickej hodnoty reprezentujúcu inštrukciu pomocou dekóderu. Súčasťou tohto kroku je aj načítanie efektívnej adresy, ak inštrukcia vyžaduje priamy, alebo nepriamy prístup do pamäte. Tieto dáta budú teda načítané do pamäte procesoru z hlavnej pamäte.
3. *EXECUTE* : vykonanie chovania inštrukcie, ktoré je získané jej dekódovaním. Toto predstavuje užitočnú operáciu, kedy procesor vykoná transformácie dát. Tieto zmeny sa môžu týkať ako hodnôt obsiahnutých v registroch a pamäti, tak aj ovplyvnenia samotného chovania systému (napríklad zmena adresy nasledujúcej inštrukcie v prípade skoku).
4. *REPEAT* : návrat do kroku jedna, kedy sa celá sekvencia krokov opakuje.

Tento proces bude modelovaný vo výslednom aplikačnom riešení len povrchne, z dôvodu, že programátor nemôže tento proces ovplyvniť inak, než použitím existujúcich inštrukcií. Bude teda reprezentovaný algoritmom, ktorý modeluje toto chovanie nad vytvoreným modelom reprezentujúci fyzický systém.

2.2 Jazyk symbolických inštrukcií a Netwide Assembler

Jazyk symbolických inštrukcií, tiež aj označovaný ako jazyk symbolických adries, predstavuje extrémne nízku abstrakciu strojového jazyka formalizovaného tak, aby bol čitateľný človekom. K prekladu takto vytvoreného kódu slúži špecifická forma kompilátoru, nazývaná asembler, ktorého úlohou je preložiť jazyk symbolických inštrukcií do strojového kódu cieľového fyzického systému implementujúceho inštrukčnú sadu.

Vzťah medzi symbolickými inštrukciami a strojovými inštrukciami je v základe izomorfný, nakoľko jedna symbolická inštrukcia sa na základe kombinácie svojich operandov (teda zdrojov dát, s ktorými pracuje), dá mapovať na väčšie množstvo konkrétnych inštrukcií zanesených v strojovom jazyku. Ako príklad sa dá uviesť inštrukcia *ADD*.

Inštrukcia *ADD* je považovaná za jednu symbolickú inštrukciu, ktorá umožňuje rôzne kombinácie operandov. Jej reálna reprezentácia v strojovom jazyku je možná štrnástimi (14) operačnými kódmi (teda špecifikácia chovania inštrukcie a toho, aké zdroje má použiť).

2.2.1 Vysoko a nízko úrovňové jazyky, strojový kód

Jazyk symbolických inštrukcií je nízko-úrovňový jazyk, ktorého obsah, dostupné konštrukcie a prostriedky sú prísne určené cieľovou architektúrou, pre ktorú bol tento jazyk vytvorený. Ako bolo spomenuté vyššie, nízko-úrovňový jazyk poskytuje extrémne nízku abstrakciu, ktorej významom je primárne poskytnutie človekom čitateľného kódu.

Takto vytvorený jazyk taktiež poskytuje aj konštrukcie, prípadne špeciálne pseudo–inštrukcie, ktorých významom je napríklad deklarovať požiadavky na pamäť, aby programátor nebol nútený vykonávať manuálne kalkulácie a deklarácie pamäte a pamäťových adries (čo s pomocou týchto konštrukcií vykoná samotný asembler).

Vysoko-úrovňové jazyky poskytujú nezávislosť na cieľovom systéme a vyššiu abstrakciu chovania do ľahšie čitateľných konštrukcií (ako napríklad jazyk C), pod podmienkou, že

existuje kompilátor zacielený na špecifickú architektúru. Významom tohoto je zjednodušenie programovania.

2.2.2 Netwide Assembler

Netwide Assembler (ďalej len NASM), predstavuje balík nástrojov a pravidiel, ktoré umožňujú programátorovi na cieľovej architektúre realizovať kontrolu a preklad zdrojových súborov obsahujúcich program napísaný v jazyku NASM.

Základná štruktúra programu vytvoreného podľa pravidiel NASM, môže byť definovaná od najmenšieho vnútorného prvku a tým je *výraz*. Takéto výrazy predstavujú jednotlivé riadky kódu, ktorý môže predstavovať užitočné inštrukcie riadiace chovanie programu, alebo pseudo–inštrukcie, ktorých úlohou je napríklad definícia pamäte. Takáto štruktúra je vždy ukončená symbolom konca riadku. Výraz je (vzhľadom na dokumentáciu NASM[2, kap. 3, sek. 1] definovaný ako:

```
label: instruction operands ; comment \n
```

kde:

- `label:` predstavuje symbolické označenie konkrétnej adresy v kóde, na ktoré je možné sa odkazovať prostredníctvom pamäťového ukazovateľa.
- `instruction` je jednoznačným identifikátorom inštrukcie podporovanej zvolenou architektúrou.
- `operands` sa označuje pole operandov, ktoré identifikujú zdroje informácii pre vykonávanie inštrukcie.
- `; comment` identifikuje komentár, teda text ignorovaný asemblerom, ktorý siaha od symbolu bodkočiarky („;“) až po koniec riadku.

Takto definované výrazy sú ďalej zaobalené pomocou špeciálnej direktívy *SECTION* (ekvivalentná direktíve *SEGMENT*), ktoré identifikuje pre potreby prekladu, kde budú dané výrazy v reálnej binárnej reprezentácii preložené.

Vzhľadom k realizácii simulačného prostriedku a objeme informácii bude teda primárne modelovaný jazyk ako postupnosti výrazov obsahujúcich ako inštrukcie, tak pseudo–inštrukcie s pevne definovanými segmentmi/sekciami pre kód a dáta.

Jazyk NASM opisuje štruktúru pre vyjadrenie vzťahu inštrukcie a dát, konkrétne definuje akým spôsobom je možné pre inštrukciu zadať operandy. Sú to tieto spôsoby:

- *Okamžitá hodnota* : Numerická hodnota, ktorá je známa v okamžiku písania kódu (môže byť aj skalárny algebraický výraz).
- *Register* : Kľúč označujúci zdrojový register, ktorého obsah bude použitý ako hodnota pre operáciu.
- *Pamäťový prístup* : Výraz, často algebraický, začína a končí hranatými zátvorkami („[“ a „]“) a výsledkom výrazu je pamäťová adresa s dátami.
- *Symbolická hodnota* : Hodnota, ktorá je známa až v momente prekladu.

Pre hlbšiu analýzu (s ohľadom na lexikálne a syntaktické pravidlá jazyka NASM[2, kap. 3] viď. prílohu[B].

2.3 Modelovanie a Simulácia

Modelovanie a Simulácie predstavuje vedný odbor, ktorého úlohou je prezentovať vedomostnú bázu, postupy a nástroje pre úspešné modelovanie fyzického systému a vykonávanie simulačných experimentov nad takýmto modelom za účelom prehĺbenia znalostí o systéme.

Základným prvkom tejto práce je vytvorenie simulátora naprogramovaného vo vyššom programovacom jazyku, ktorý umožní vykonávať simulačné experimenty s jazykom symbolických inštrukcií pre procesory rodiny x86 (s ohľadom na tridsaťdva (32) bitovú architektúru), bez toho, aby používateľ musel disponovať systémom s takýmito prostriedkami (konkrétne procesor rodiny x86). Úlohou tejto simulácie je taktiež umožniť, pomocou vhodne navrhnutých prostriedkov, používateľovi identifikovať procesy odohrávajúce sa počas vykonávania samotných inštrukcií (teda viditeľnosť zmien dát v dátovom modely).

Pre potreby navrhnutia simulátora je nutné identifikovať a navrhnúť dátový model, ktorý bude vychádzať zo znalostí nadobudnutých analýzou fyzického systému zahrnutou v tejto kapitole.

Základom pre simulátor bude diskrétna simulácia, teda simulačné experimenty budú vykonáva v krokoch, v ktorých je definovaný stav dátového modelu. Pre potreby aplikačného riešenia budú teda tieto kroky určené ako vykonanie singulárnej inštrukcie a stav dátového modelu je teda možné analyzovať len medzi týmito krokmi.

Simulačný experiment sa bude skladať z dvoch základných krokov a to:

1. Príprava simulačného experimentu, teda definovanie vstupného programu a overenie jeho správnosti oproti pravidlám jazyka NASM

2. Vykonávanie simulačného experimentu, teda analýza stavov systému medzi vykonaním jednotlivých inštrukcií (alebo bloku inštrukcií)

Kritickým významom simulácie je aj schopnosť vidieť a pochopiť zmeny, ktoré sa odohrávajú nad modelom simulovaného systému, k tomuto poslúži teda vhodne zvolený spôsob reprezentácie dát obsiahnutých v modele.

Kapitola 3

Návrh

Návrh simulačného prostriedku bude založený na vedomostnej báze nadobudnutej predošlou analýzou reálneho fyzického systému a zvolenej abstrakcii. Mimo tohto je nutné rozlíšiť základné prvky takto navrhovaného aplikačného riešenia. Tento program, či simulačný prostriedok bude teda obsahovať základné moduly, ktoré budú svojou interakciou vytvárať cieľové chovanie, teda umožnia používateľovi navrhnúť, otestovať, spustiť a simulovať vykonávanie jednoduchého programu napísaného v jazyku Netwide Assembler.

Základnými modulmi teda budú:

- Dátový model predstavujúci abstrakciu fyzického systému
- Prekladač používateľského kódu do vnútornej reprezentácie programu
- Vnútorná reprezentácia programu
- Vykonávač vnútornej reprezentácie programu
- Zobrazovač zmien nad dátovým modelom systému

Pre návrh týchto modulov bude použité objektovo orientovaný prístup, nakoľko umožňuje navrhovať reálne súčasti implementácie ako modely fyzického systému a definovať nielen ich dáta, ale aj chovanie v čo najvhodnejšej reprezentácii voči tomu, ako sa tieto súčasti chovajú vo fyzickom systéme.

3.1 Dátový model

Dátový model bude navrhnutý na základe analýzy, ktorá definovala nosné prvky fyzického systému. Týmito prvkami sú charakteristické súčasti definované v rámci architektúry ako registre, pamäť a iné. Tieto modely umožnia demonštráciu chovania systému s dostatočnou abstrakciou, aby postačili k nadobudnutiu nových znalostí zo simulácie.

3.1.1 Register a Registrový súbor

Register ako komponent je kriticky dôležitý pre definovanie chovania, nakoľko je základný kameň pre drvivú väčšinu operácií, ktorých je procesor schopný. Vychádzajúc z analýzy, modelovanie procesoru bude čo najhodnovernejšie modelovaniu reálneho komponentu.

Model registra teda bude predstavovať súčasť systému, ktorá obsahuje spôsob ako ukladať, uchovávať a vystavovať dáta obsiahnuté v jej vnútornej premennej. Taktiež bude tento

model schopný na základe požiadavky z vyššej úrovne systému identifikovať pomocou prístupového kľúča špecifický rozsah dát zo svojej vnútornej reprezentácie a adekvátne ich vystaviť.

Takto vytvorený komponent teda bude schopný na základe kľúča identifikovať, či daný kľúč náleží do jeho množiny identifikátorov, vystaviť reprezentáciu hodnoty zviazanú s daným kľúčom a určiť k akej veľkosti vnútornej binárnej reprezentácie hodnoty sa daný kľúč vzťahuje.

Základná štruktúra modelu registra sa bude skladať z vnútornej premennej udržiavajúcej hodnotu, množiny kľúčov, z ktorých každý predstavuje určitý rozsah z binárnej reprezentácie vnútorného vektoru a mechanizmov, ktoré na základe kľúča vyberú z tejto vnútornej hodnoty určité dáta a predstavia ich.

Takto vytvorené registre sa budú pre prehľadnosť a organizáciu zlučovať do modelu, ktorý reprezentuje dostupné všeobecné registre a obsluhuje prístupy k nim. Tento systém obsluhy taktiež zabezpečí vnútornou kontrolou, že systém nebude disponovať dvomi registrami s rovnakým prístupovým kľúčom.

3.1.2 Pamäť a Pamäťový súbor

Pamäť bude s ohľadom na abstrakciu reprezentovaná ako systém disponujúci poľom pamäťových buniek (veľkosti jedného (1) bajtu), ku ktorým bude možné pristupovať pomocou použitia adresy konkrétnej bunky. Z hľadiska modelovania musí taktiež tento systém umožniť organizáciu dát pomocou systému *Little–Endian*, kedy budú na základe požiadavky na veľkosť prístupu, tieto dáta správne načítané, zorganizované a vystavené.

Pre potreby kontroly chovania programu a možnej identifikácie problémov vzniknutých zaneseným sémantickej chyby do kódu, bude dátový model reprezentácie pamäte taktiež disponovať subsystémom pre validáciu prístupov do pamäte. Na základe deklarácie nárokov na pamäťový priestor (ktorú vykoná používateľ), model umožní vykonávanému programu pristupovať k tejto pamäti. Prístup k iným ako povoleným častiam pamäťového modelu bude teda považovaný za ilegálny a vyvolá chybu.

Takto vytvorený komponent teda bude schopný na základe dvojice adresa a veľkosť poskytnúť dáta z pamäte (ktorá je organizovaná formou *Little–Endian* ako priamu hodnotu. Mimo nálezu a vystavenia dát taktiež vykoná kontrolu, či je program oprávnený k daným dátam pristupovať.

Základná štruktúra modelu pamäte sa bude skladať z vnútorného poľa typu `byte`, mechanizmov pre preklad *Little–Endian* hodnoty do priamej hodnoty, overenie legálnosti prístupu ku konkrétnej adrese (nielen k špecifickej bunke, ale k požadovanému rozsahu) a podsystém, ktorý bude udržovať mapu legálnych prístupov prostredníctvom poľa typu `byte`. Táto kontrola bude realizovaná pomocou asociácie konkrétnych bitov s bunkami hlavnej pamäte (pre zníženie pamäťovej náročnosti).

3.1.3 Príznaky a Príznakový súbor

Vzhľadom k tomu, že príznakový register v fyzickom systéme je reálny register, ktorého konkrétne bity predstavujú príznaky, model príznakového registra bude v základe vnútorná premenná udržiavajúca hodnoty jednotlivých bitov ako binárny vektor. Pre potreby prístupov konkrétnych hodnôt teda bude nutné vytvoriť mechanizmus, ktorý umožní asociovať konkrétny bit s kľúčom označujúcim príznak.

K tomuto bude slúžiť systém dvojíc, ktoré sa budú skladať z indexu bitu a kľúča a príznakový model bude túto množinu dvojíc používať k riadeniu prístupov k hodnotám

bitov. Toto umožní nastavovať bity, testovať ich hodnotu a taktiež predstaviť príznakový register ako hodnotu reprezentovanú binárnym vektorom.

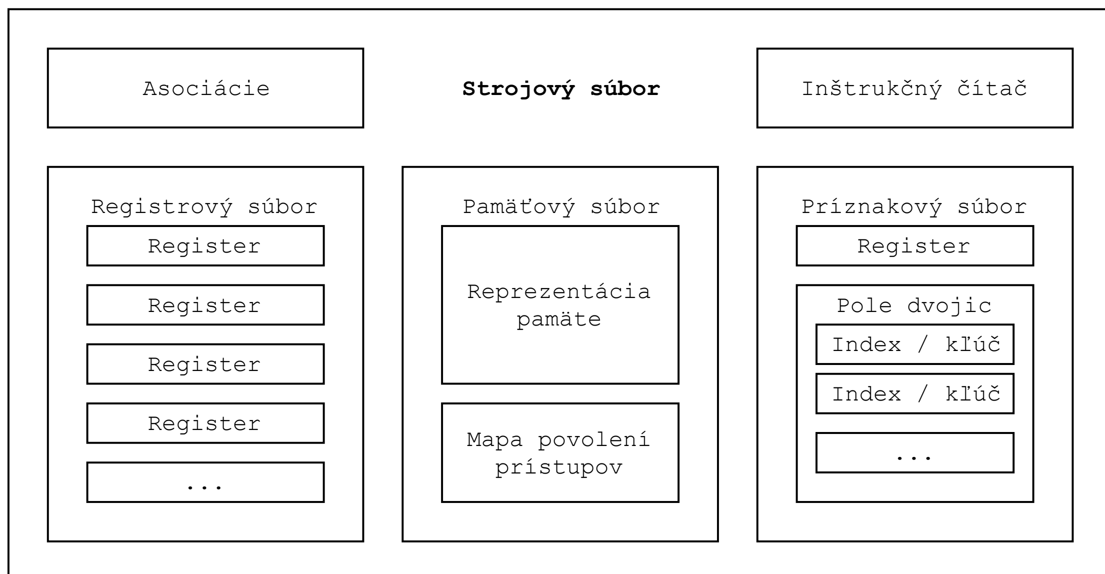
3.1.4 Strojový súbor

Strojový súbor predstavujú najvyššiu úroveň hierarchie dátového modelu. Tento systém bude agregovať registrový, pamäťový a príznakový súbor a poskytovať kontrolované prístupy k dátam v nich uloženým.

Takto vytvorený strojový súbor bude taktiež poskytovať abstrakciu inštrukčného ukazovateľa, ktorý bude realizovaný ako jednoduchá vnútorná premenná ukazujúca na aktuálne vykonávanú inštrukciu obsiahnutú vo vnútornej reprezentácii programu.

Ako poslednou súčasťou bude mechanizmus na reštart, teda re–inicializáciu obsahu pamäťového súboru, pre potreby vykonávania opakovaných experimentov nad dátami bez nutnosti vytvárať novú reprezentáciu strojového súboru.

Pre prehľadné zobrazenie je štruktúra Strojového súboru a jeho podsystémov demonštrovaná diagramom[2.1].



Obr. 3.1: Triviálne zobrazenie štruktúry navrhovanej reprezentácie stroja.

3.2 Interpretácia vstupu

Interpretácia vstupu predstavuje ďalší komplexný problém tohto aplikačného riešenia. Je nutné nielen vhodne spracovať používateľský vstup (ktorý je definovaný reťazcami znakov, teda surovým textom), ale taktiež zvoliť správny spôsob interpretácie používateľského programu do formy vnútornej reprezentácie, pričom používateľský program by mal byť spustiteľným programom z prostredia Netwide Assembly (NASM).

Z tohoto dôvodu bol zvolený prístup, ktorý umožňuje vstupný text spracovať s využitím znalostí lexikálnych, syntaktických a sémantických pravidiel založených na množine pravidiel jazyka NASM.

Vzhľadom k možnostiam, ktoré umožňujú pravidlá zanesené v špecifikácii NASM, bola zvolená vhodná forma abstrakcie a obmedzení na používateľský vstupný kód tak, aby výsledný kód bol stále platným kódom pre asembler NASM, no jeho interpretácia bola zjednodušená. Konkrétne zmeny nad pravidlami sú zmienené ako v prílohe o analýze jazyka NASM [B], tak nižšie v tejto kapitole.

3.2.1 Štruktúra spracovania používateľského vstupu

V základnom návrhu bol vstup od používateľa rozdelený na dva prúdy, či zdrojové reťazce. Prvým je reťazec obsahujúci užitočný kód, teda inštrukcie. Druhým zdrojom sú používateľom definované nároky a požiadavky na pamäť, ktorá bude sprístupnená programu v dátovom modele.

Takto definované zdroje informácii budú podrobené lexikálnej analýze, ktorá vyprodukuje dve polia tokenov, teda identifikovaných výrazov definovaných lexikálnymi pravidlami. Následne budú tokeny spracované interpretačnou logikou, ktorá zoradí tokeny do zmysluplných výrazov založených na syntaktických pravidlách jazyka NASM (teda budú tokeny rozdelené na postupnosti podľa tokenov indikujúcich koniec riadku).

Spracovanie pamäťových deklarácii je založené na princípe, že každý neprázdny riadok obsahuje buď symbolické označenie párované s konkrétnou adresou v pamäti (teda reťazec zodpovedajúci lexikálnemu pravidlu symbolu nasledované dvojbodkou), deklaráciu pamäte pomocou pseudo–inštrukcie[2, kap. 3, sek. 2] (inicializovaná a nie inicializovaná deklarácia, teda rodiny `RESx` a `Dx` pseudo–inštrukcie), alebo ich kombináciu.

Ich postupným spracovaním bude vybudovaná tabuľka symbolov, ktorá poskytne informácie pre preklad používateľského užitočného kódu.

Následne bude obdobne spracovaný zdroj obsahujúci samotnú programovú logiku. V tomto prípade bude taktiež rozdelenie postupnosti tokenov do výrazových celkov, načo systém vykoná prvý prechod kódom pre identifikovanie a budovanie mapy návestí, ktoré slúžia skokovým inštrukciám. Výsledkom prvého prechodu je postupnosť výrazov, ktoré sú jednoznačne identifikované (teda či sa jedná o výraz obsahujúci návestie, alebo výraz obsahujúci inštrukciu).

Druhým prechodom je vykonaná samotná interpretácia inštrukcií, kedy bude použité špeciálnych objektov k prekladom konkrétnych inštrukcií do vnútornej reprezentácie programu. Takto rozdelené spracovanie umožňuje rozširovanie existujúceho kódu o nové inštrukcie bez potreby hlbších zásahov do primárneho spracovania vstupného kódu programu. Podmienkou je, aby nové inštrukcie využívali lexikálne pravidlá aplikované už existujúcim systémom.

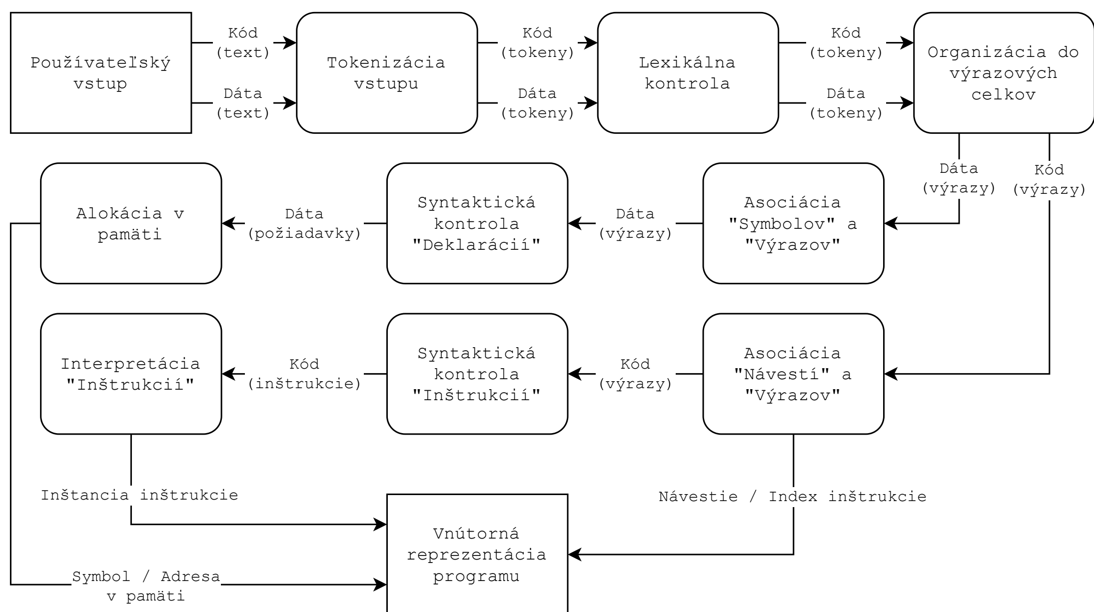
V závere bude vnútorná reprezentácia podrobená kontrole na existenciu požadovaných prvkov (existujúce návestie `main:`) a bude jej nastavená počiatočná adresa pre štart programu.

Postup tohto prekladu je demonštrovaný diagramom pre prehľadnosť[3.2].

3.2.2 Lexikálna a syntaktická analýza

Lexikálna analýza

Lexikálna analýza je realizovaná prostredníctvom konečného stavového automatu, ktorý spracováva vstup po znakoch a identifikuje definované tokeny. Táto analýza sa skladá z dvoch základných krokov, nakoľko primárne podľa lexikálnych pravidiel identifikuje len tri typy tokenov a to:



Obr. 3.2: Postupnosť interpretácie používateľského vstupu.

- Jednoznakové riadiace symboly (symbol konca riadku, aritmetické operátory, atď.)
- Numerické hodnoty
- Symboly

V nasledujúcom kroku je na základe prostriedkov systému vytvorený súbor slovníkov, ktoré umožňujú identifikovať, či konkrétny symbol náleží do niektorej z definovaných množín (registrový kľúč, definícia veľkosti, identifikátor inštrukcie) a metadáta tokenu sú náležite upravené. Táto štruktúra závislostí je zobrazená na diagrame[3.3].

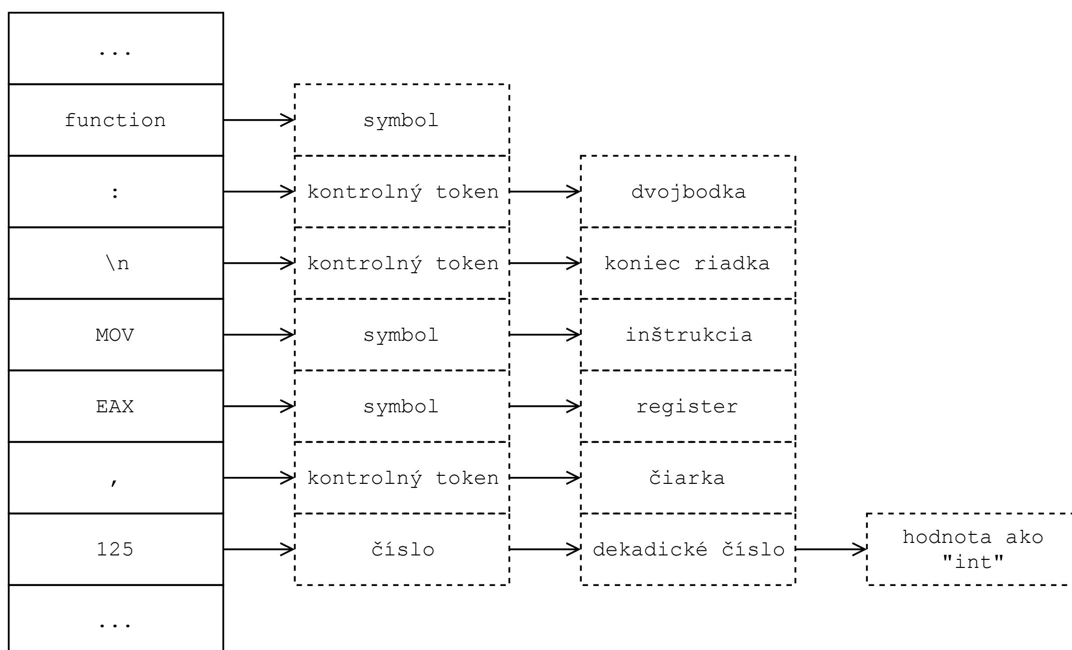
Takto spracovaný vstup je následne posunutý syntaktickej analýze.

Syntaktická analýza

Syntaktická analýza predstavuje z hľadiska spracovania tri kroky.

1. Rozdelenie tokenov na výrazy
2. Kontrola výrazov (prvý cyklus analýzy)
3. Kontrola inštrukčných výrazov (druhý cyklus analýzy)

Prvým krokom je rozdelenie poľa tokenov na výrazy (teda koncom riadka oddelenými celkami). Príklad štruktúry takto spracovaných výrazov je možné vidieť na obrázku[3.4]. V rámci tohto kroku sú taktiež výrazy podrobené analýze, ktorá sa snaží identifikovať, či všetky celky obsahujú legálny počiatočný token. Výraz môže z hľadiska zvoleného jazyka začať iba symbolom návestia, alebo identifikátorom inštrukcie. V prípade, ak výraz obsahuje kombináciu, čo je legálne z hľadiska prekladu, bude výraz rozdelený na dva samostatné celky.



Obr. 3.3: Spôsob, akým sú interpretované lexikálne dáta.

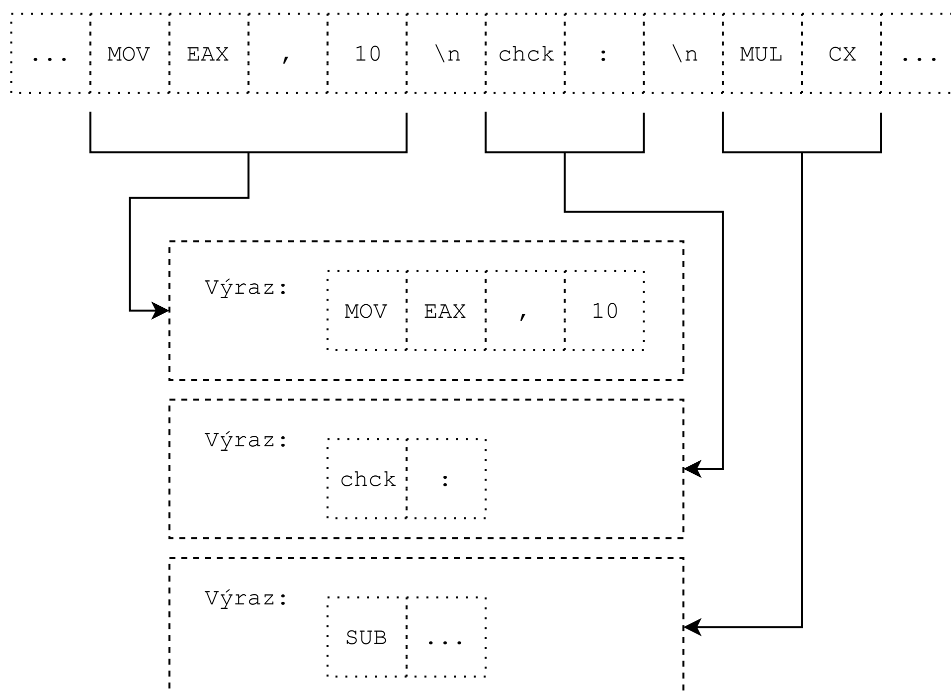
Druhým krokom je vybudovanie mapy návestí, čo je doplnenie tabuľky symbolov o symbolické hodnoty ukazujúce do kódu na konkrétne inštrukcie. V rámci tohto kroku je taktiež realizované párovanie globálnych a lokálnych návestí, overenie správnosti týchto návestí (či nedošlo k znovu–použitiu, teda redeklarácií návestia a nedošlo ku kolízii medzi označením návestia a označením symbolickej adresy v pamäti dát).

Tretím a posledným krokom je postupné spracovanie výrazov obsahujúcich inštrukcie. Toto je realizované pomocou delegácie zodpovednosti za overenie syntaxe a interpretáciu inštrukcie továrenským objektom[8, kap. 1]. Tie budú schopné identifikovať, či výraz náleží im a následne ho podrobia overeniu na základe ich vlastnej, internej logiky. Výsledkom takejto interpretácie je teda objekt patriaci do vnútornej reprezentácie programu (reprezentácia singulárnej inštrukcie), alebo výnimka indikujúca chybu.

3.2.3 Továrne inštrukcií

Továrne inštrukcií slúžia ako spôsob zjednodušenia spracovania samotných výrazov obsahujúcich inštrukcie. Vzhľadom k veľkému objemu rôznych dostupných kombinácií typov operandov si teda každá továreň vykoná syntaktickú analýzu a overenie po vlastnej osi. Vďaka tomuto je možné simulátor rozširovať o nové inštrukcie bez nutnosti hlbších zásahov do primárneho spracovania používateľského kódu programu.

Samotné inštrukčné továrne sú agregované v štruktúre reprezentujúcu hlavnú továreň. Tá na základe požiadavkov interpretačnej logiky poskytuje továrenské objekty, ktoré náležia inštrukčnému identifikátoru. Vzhľadom k tomu, že spracovanie operandov je prenositeľné medzi inštrukciami, tieto továrne disponujú balíkom funkcionalít, ktoré sú univerzálne použiteľné. Pre prípad, ak by inštrukcia vyžadovala špecifickú implementáciu spracovania, je



Obr. 3.4: Príklad rozloženia poľa tokenov na výrazy.

v rámci návrhu umožnené túto funkcionalitu realizovať (teda implementovať špeciálnym obslužným kódom).

3.2.4 Vnútorná reprezentácia programu

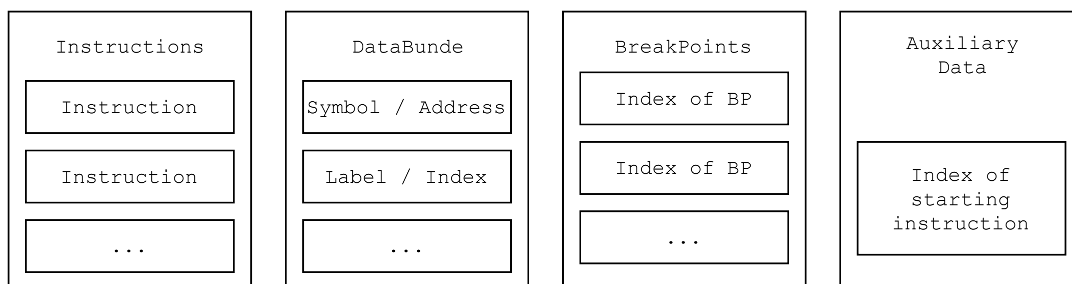
Výsledkom interpretácie korektného používateľského vstupu je vnútorná reprezentácia programu. Tento program disponuje niekoľkými základnými informáciami potrebnými pre vykonanie simulačného experimentu. V základe reprezentácia obsahuje:

- Pole s objektami reprezentujúce inštrukcie a ich operandy
- Pole s objektami reprezentujúce vzťahy symbolických a reálnych hodnôt
- Počiatočnú adresu pre štart programu

Táto štruktúra je taktiež demonštrovaná diagramom[3.5].

Základným prvkom programu je reprezentácia inštrukcie, čo je objekt, ktorý pomocou vnútorných metód definuje chovanie inštrukcie, teda zmeny vykonávané nad dátovými modelom. Inštrukcia taktiež môže obsahovať operandy, teda objekty, ktoré slúžia ako zdroje dát pre chovanie inštrukcie. Tieto operandy môžu predstavovať statický zdroj informácii (konštanta, či okamžitá hodnota), alebo súčasti dátového modelu, ktorých hodnoty môžu byť dynamické počas vykonávania programu (odkaz na register, prípadne pamäťový priestor).

Zdieľané chovanie operandov je možnosť transformovať dáta (v prípade, ak to je sémanticky zmysluplné, teda napríklad nie je možné ukladať hodnotu do konštanty) a metódu,



Obr. 3.5: Triviálna reprezentácia štruktúry interpretovaného programu.

ktorá definuje veľkosť operandu. V rámci návrhu sú definované štyri základné typy operandov a to:

`IMM` : Okamžitá hodnota, numerická konštanta

`REG` : Odkaz na register obsiahnutý v dátovom modele

`MEM` : Odkaz do pamäťového priestoru

`SYM` : Okamžitá hodnota známa až počas prekladu, ktorá predstavuje adresu v pamäti, fixnej veľkosti štyri (4) bajty

Operandy okamžitej hodnoty a prístupov do pamäte umožňujú využitie algebraických konštrukcií. Operand okamžitej hodnoty umožňuje opísať túto hodnotu rovnicou, ktorá obsahuje statické, teda skalárne hodnoty známe v dobe prekladu. Operand prístupu do pamäte umožňuje využiť aritmetických operácií a kombinácie konštánt, registrov a symbolických hodnôt pre definovanie prístupu do pamäte v rámci vykonávania programu.

Vyhodnotenie týchto algebraických výrazov je realizované pomocou algoritmu *Shunting–yard*, rozšíreného pre unárne operácie.

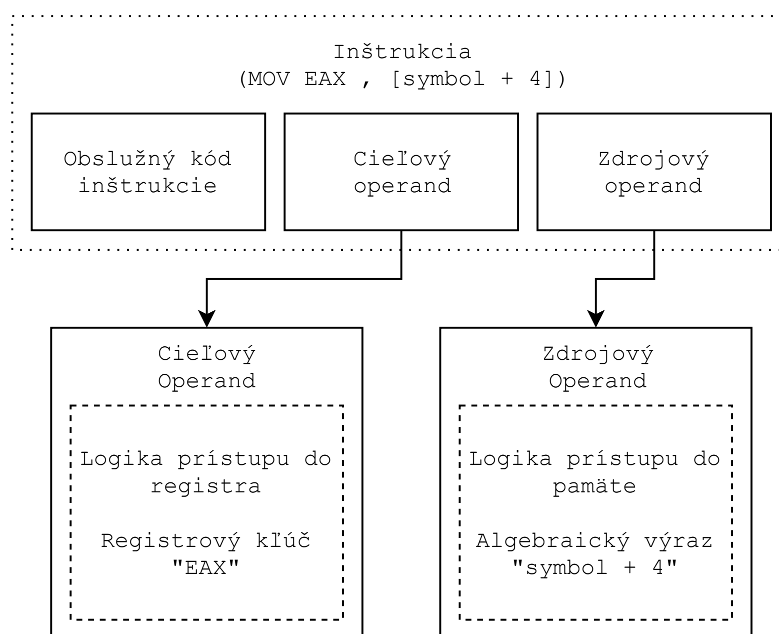
Takto štrukturovaný objekt je spárovaný so špecifickým modelom fyzického systému a umožňuje pomocou systému vykonávania programu realizovať simulačný experiment. Teda predstavuje základnú jednotku nesúcu a opisujúcu operáciu nad dátami.

Štruktúru takto navrhnutého objektu je možné vidieť na diagrame[3.6].

3.2.5 Abstrakcia jazyka NASM

V rámci návrhu modelu vstupného kódu, teda abstrakcie možností, ktoré poskytuje prostredie NASM boli zvolené isté úpravy existujúcich pravidiel pre vstupný kód. Novo vzniknutá forma kódu je stále legálna v rámci prekladu originálneho kódu a vychádza zo snahy o čo najjednoduchšiu formu kódu s ohľadom na obmedzenia navrhnutého systému prekladu. Konkrétne ide o:

- Každý používateľom navrhnutý program začína prvou inštrukciou za návestím `main:`
- Koniec programu nastane dosiahnutím volania inštrukcie `RET` nad prázdnym zásobníkom



Obr. 3.6: Vnútorná štruktúra inštrukcie.

- Symbolické pomenovania miest v pamäti a návestí v kóde musia byť zakončené znakom dvojbodky
- Inštrukcie pracujúce nad zásobníkom poskytujú len tridsaťdva (32) bitové operácie
- Ukazovatele do pamäte sú fixne veľkosti tridsaťdva (32) bitov

3.3 Vykonávanie programu

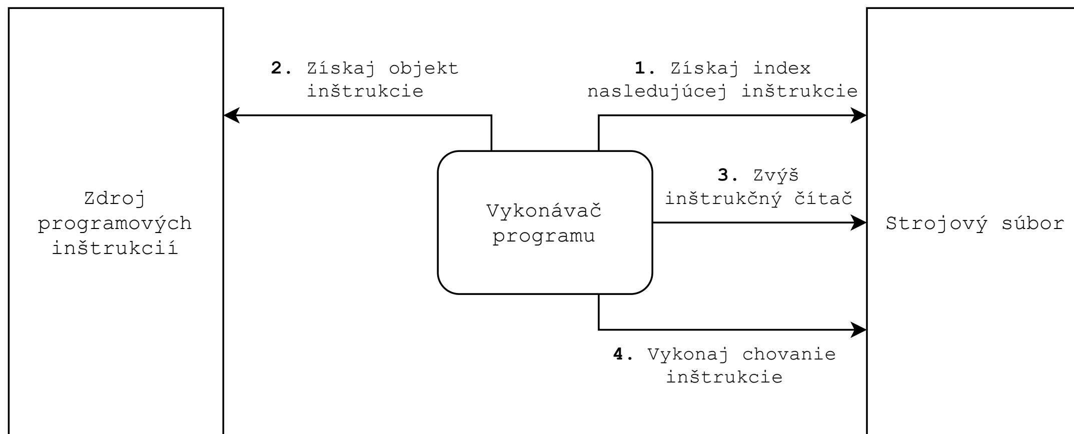
Vykonávanie programu nad modelom dát je simulačným experimentom. Pre úspešné začatie simulácie je potrebné dodať vnútornú reprezentáciu programu a dátový model zastúpený strojovým súborom popísaným vyššie. Takto pripravené prostriedky sú následne podrobené vykonaniu simulačného experimentu.

Simulačný experiment umožňuje používateľovi prostredníctvom grafickej reprezentácie stavu dátového modelu skúmať, aký malo vykonanie inštrukcie dopad na simulovaný systém. V rámci tohoto vykonávania je teda nutné, aby systém nasledoval postup, akým pracuje reálny systém, teda:

1. Podľa hodnoty inštrukčného čítača (ukazovateľa) načíta inštrukciu uloženú v poli inštrukcií vo vnútornej reprezentácii programu.
2. Inkrementuje inštrukčný čítač (ukazovateľ) o jedna (1), nakoľko táto hodnota predstavuje index v poli a nie reálny ukazovateľ do pamäte programu.
3. Vykoná inštrukciu, teda aktivuje jej chovanie. Toto môže skončiť úspechom, teda simulácia prejde do nasledujúceho kroku, alebo systém indikuje vhodným spôsobom chybu.

4. Späť na krok 1.

Je možné si všimnúť, že tento algoritmus je podobný opisu algoritmu v predošlej kapitole, akým fyzický systém realizuje vykonávanie inštrukcií v skutočnom prostredí. Tento algoritmus je demonštrovaný diagramom na obrázku[3.7].



Obr. 3.7: Algoritmus vykonávania simulovaného programu.

Ukončenie simulácie je možné zo strany používateľa v ktoromkoľvek legálnom bode (teda medzi jednotlivými vykonávaniami inštrukcií, nakoľko je simulácia diskrétna a teda nie je definovaný stav medzi týmito bodmi). Simulácia taktiež môže dosiahnuť ukončenia na základe vlastných obmedzení a to:

A. Vykonávač programu narazil na volanie inštrukcie `RET`, pričom je zásobník prázdny. Táto situácia je definovaná ako dosiahnutie konca hlavného tela programu.

B. Vykonávač vykonal „N“ krokov, čo je vnútorne nastavená hodnota. Dosiahnutie tohto počtu krokov indikuje pravdepodobne nekonečný cyklus a vykonávač ukončuje simuláciu.

C. Vykonávač programu indikoval počas vykonania inštrukcie chybu, v takomto prípade je simulačný experiment zastavený.

Z hľadiska simulácie je teda používateľ schopný vykonať niekoľko aktivít so samotným simulačným experimentom, ktorý ovplyvňuje priamo vykonávanie programu a to:

A. Vykonať jeden „krok“, čiže vykoná jednu inštrukciu.

B. Vykonať „beh“ od aktuálnej pozície inštrukčného ukazovateľa, teda vykoná program celý.

C. Vykonať všetky inštrukcie, pokým nedosiahne konca programu, alebo symbolu indikujúceho pozastavenie vykonávania, teda „break point“.

D. Reštartovať experiment, teda inicializovať dáta v registroch a pamäti na nulové (prípadne počiatočné) hodnoty a nastaviť inštrukčný ukazovateľ na začiatok programu definovaný vo vnútornej reprezentácii programu.

Jeden z vyššie zmienených spôsobov ukončenia simulácie je indikácia chyby počas vykonávania inštrukcie, alebo načítania novej inštrukcie. Je nežiaduce, aby experiment pokračoval, nakoľko táto chyba môže do experimentu zaniesť nepredvídané chovanie a situácie. Za tieto chyby môžeme považovať chybné použitie aritmetických inštrukcií (delenie nulou, extrémne rozsahy dát pri násobení, a pod.), alebo prístupy do ilegálnych sekcií pamäte (vrátane podtečenia a pretečenia zásobníka, či použitia ilegálnej adresy pri vyvolaní inštrukcie `RET`).

3.4 Zobrazovanie dát

Základným kameňom tejto práce je adekvátne sprostredkovanie informácii používateľom. V prípade ak používateľ bude používať tento simulačný prípravok, musí byť schopný prehľadne identifikovať problémy a situácie, ktoré môžu vzniknúť ako počas tvorenia programu, tak aj informácie vychádzajúce z vykonávania simulačného experimentu.

K tomuto bude slúžiť grafické používateľské rozhranie, ktoré bude z hľadiska použitia rozdelené do dvoch celkov:

A. Editor

B. Simulátor

3.4.1 Editor

Editor je obrazovka určená k príprave simulačného experimentu. V tomto rozhraní používateľ môže pracovať s programom ako textom, vytvárať nové konštrukcie, či upravovať existujúce. Toto rozhranie taktiež poskytuje informácie o interpretácii a prípadných chybách, na ktoré používateľa upozorní chybovým hlásením a číslom riadku, kde k chybe došlo.

Mimo týchto poskytuje toto rozhranie taktiež prístup k súborovému systému, kde môže používateľ otvárať uložené programy, alebo načítať existujúce príklady riešení. V prípade, ak je program interpretovateľný, môže používateľ pristúpiť k vykonávaniu simulácie.

Návrh rozloženia obrazovky Editora je možné vidieť na obrázku[3.8].

3.4.2 Simulátor

Simulátor sa skladá zo štyroch základných celkov a to:

A. Reprezentácia programu s indikáciou inštrukcie, ktorá bude vykonaná v nasledujúcom kroku

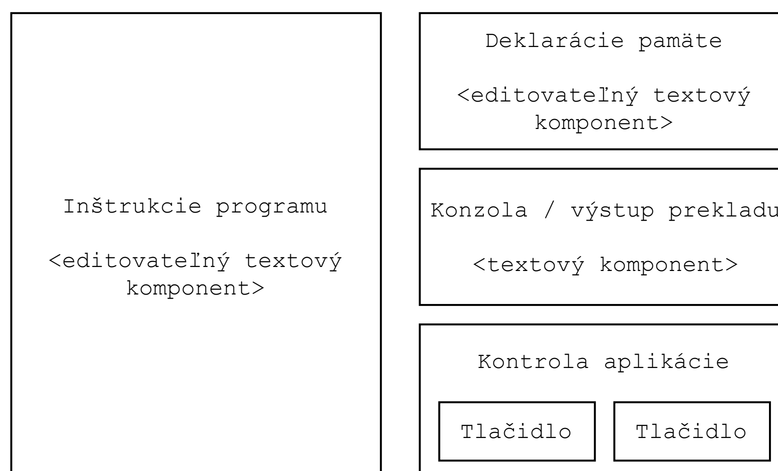
B. Kontrolný panel simulácie, ktorý umožňuje používateľovi ovládať simuláciu

C. Inšpekčný panel strojového súboru, teda dátového modelu

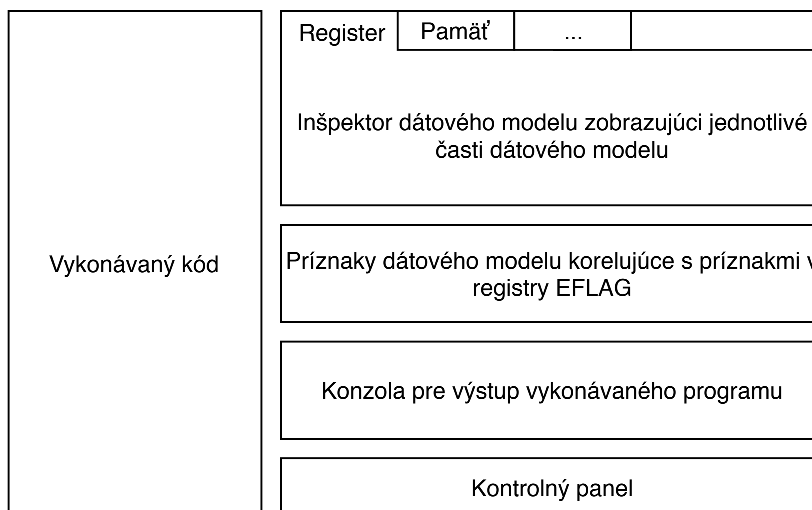
D. Konzola indikujúca stav a slúžiaca ako štandardný výstup

Z týchto štyroch (ktorých návrh rozloženia je možné vidieť na obrázku[3.9]) je najvýznamnejším Inšpekčný panel, teda spôsob, akým je používateľovi počas simulačného experimentu zobrazovaný stav dátového modelu.

Tento inšpektor poskytuje niekoľko základných pohľadov na dátový model, konkrétne ide o zobrazenie všetkých kľúčových prvkov systému, ktoré sa dynamicky menia na základe zmien vykonaných inštrukciou. Ako pádne významné inšpekčné zobrazenia boli zvolené:



Obr. 3.8: Návrh rozhrania pre obrazovku editoru.



Obr. 3.9: Návrh rozhrania pre obrazovku editoru.

A. Inšpektor obsahu registrov

B. Zobrazenie pamäte ako sekvencie bajtov, teda ako *Big–Endian*

C. Abstrakcie pamäte ako poľa numerických hodnôt (s využitím *Little–Endian* dekódovania)

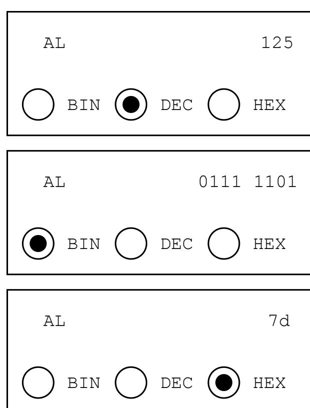
D. Zobrazenie zásobníka a jeho hodnôt v kontexte s obsahov registrov zviazaných so zásobníkom (*SP* a *BP*)

E. Zobrazenie ľubovoľného, premenlivého aspektu systému (formou operandu inštrukcie), teda obsah registra, alebo pamäte ako singulárneho výrazu

F. Zobrazenie stavu príznakov

Špecifickou súčasťou inšpekčnej sekcie je zobrazovač stavu príznakov, ktoré sú viditeľné stále (navrhované rozloženie obrazovky ráta s možnosťou prepínať medzi jednotlivými inšpektormi).

Vzhľadom k variabilite, akou je možné interpretovať numerické dáta, systém zobrazenia by mal disponovať nástrojmi pre dynamické zobrazenie dát počas vykonávania simulácie v želanej forme. Príkladom takéhoto chovania môže byť sekvencia demonštrovaná na diagrame[3.10].



Obr. 3.10: Postupnosť, ako by mal používateľ byť schopný meniť zobrazované dáta.

Kapitola 4

Implementácia

Táto kapitola pojednáva o realizácii opísaného návrhu formou aplikačného riešenia. Dôraz bude kladený na špecifické aspekty implementácie, ako aj spomenutie všeobecných, celo–aplikačne zvolených riešení.

Jazykom zvoleným pre implementáciu návrhu je jazyk *Java SE8*, pričom bude využitých len natívne obsiahnutých knižníc a prostriedkov, aby sa zabezpečila čo najširšia kompatibilita so systémami, na ktorých bude aplikácia používaná.

Vzhľadom na objektovú podstatu aplikácie (teda primárny koncept objektovo orientovaného programovania: objekty komunikujúce správami) bol ako univerzálny prostriedok pre zdieľanie dát zvolený dátový typ `int`, ktorý je v prostredí Java definovaný ako štyri (4) bajty široký dátový typ implementujúci kódovanie numerických hodnôt prostredníctvom dvojkového doplnku.

Ako nástroj pre návrh a implementáciu grafického používateľského rozhrania bol zvolený *toolkit* Swing, nakoľko poskytuje jendoduché komponenty určené pre reprezentáciu dát formou textu.

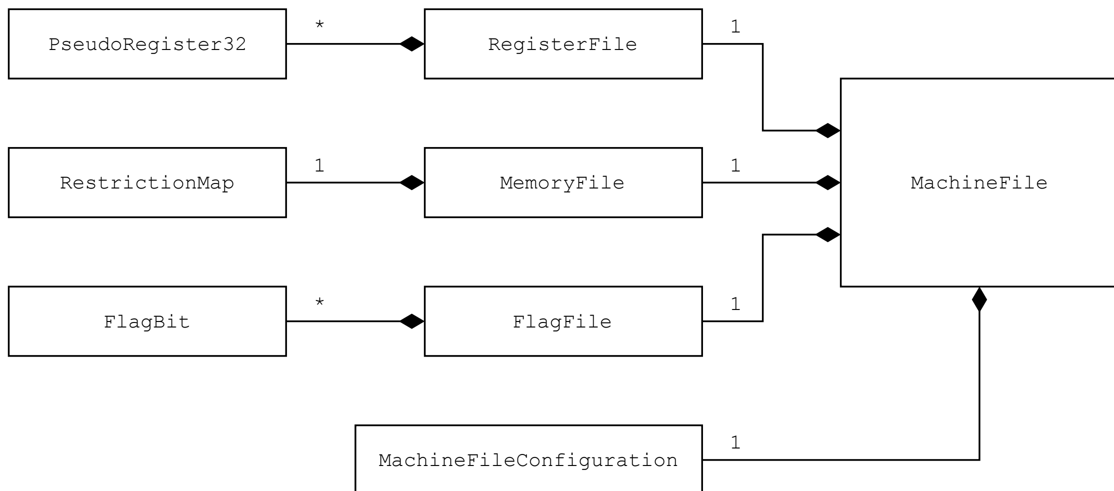
Založené na rozdelení zmienenom v návrhu, systém je programovaný modulárne, teda aj implementácia bude riešená sériou modulov, ktoré vzájomne spolupracujú a tým vytvárajú funkcionalitu aplikácie. Týmito modulmi sú:

- Modul strojového súboru
- Modul interpretácie
- Modul vykonávania
- Modul používateľského rozhrania

Vzhľadom k všeobecným informáciám o aplikácii je vhodné zmieniť, že ako návrhový vzor pre realizáciu bol zvolený *Model–View–Controller*, ktorý poslúžil ako základ pre architektúru riešenia. Jednotlivé systémy spolu komunikujú vhodne zvolenými rozhraniami, ktoré sú spomenuté nižšie.

4.1 Implementácia modulu strojového súboru

Strojový súbor predstavuje primárny dátový model, nad ktorým je vykonávaný program, teda simulačný experiment. Z hľadiska návrhu v sebe tento strojový súbor agreguje objekty



Obr. 4.1: Zjednodušený diagram tried demonštrujúci vzťahy objektov.

predstavujúce jednotlivé súčasti systému ako sú registre, pamäť, príznakový súbor a inštrukčný čítač. K týmto súčastiam taktiež poskytuje kontrolovaný prístup. Štruktúru tohto strojového súboru je možné demonštrovať obrázkom[4.1].

4.1.1 PseudoRegister32 a RegisterFile

Tieto dve triedy sú primárnou reprezentáciou dostupných všeobecných registrov pre použitie programátorom. Objekt `PseudoRegister32` modeluje chovanie konkrétneho registra a jeho inštancie tým pádom predstavujú konkrétne registre identifikované svojimi prístupovými kľúčmi.

Z hľadiska implementácie bol spôsob prístupu k dátam realizovaný pomocou poľa typu `byte`, ktoré má veľkosť fixne nastavenú na štyri (4) a vybavené obslužným mechanizmom.

K nastavovaniu hodnôt tohoto vnútorného poľa slúžila časť mechanizmu, ktorá spracovala vstupné hodnoty definované ako numerickú hodnotu typu `int`. K tejto obsluhe sa viažu štyri (4) metódy a to:

- `public void setFull (int value)`
- `public void setHalf (int value)`
- `public void setLow (int value)`
- `public void setHigh (int value)`

Ako už napovedajú názvy, každá metóda zodpovedala za nastavovanie určitého rozsahu vnútorného poľa, ktoré zodpovedá prístupu k rozsahom v rámci fyzickej verzie registra, teda plný tridsaťdva (32) bitový rozsah (napr.: `EAX`), polovičný šestnásť (16) bitový rozsah (napr.: `AX`), dolný bajt polovičného rozsahu (napr.: `AL`) a horný bajt polovičného rozsahu (napr.: `AH`).

Získavanie hodnôt z registra je realizované druhou časťou mechanizmu, ktorá je analogická k ich nastavovaniu, avšak je rozšírená o detekciu hodnoty najvyššieho bitu požadovaného rozsahu, aby objekt zabezpečil *bit padding* najviac významného bitu. Toto chovanie

vykoná rozšírenie najviac významného (najvyššieho) bitu z hodnoty, ktorej rozsah je menší ako tridsaťdva (32) bitov do plnej šírky dátového typu návratovej hodnoty (teda dátový typ `int`). Význam tejto operácie je pre potreby korektného vykonávania aritmetických operácií. Výsledok je potom pole typu `byte` zaobalené prostredníctvom využitia prostriedkov obsiahnutých v knižnici `NIO` (*New IO*) a takto je vytvorený dátový typ `int` ako návratová hodnota.

K tomu slúžia tieto metódy:

- `public int getFull ( )`
- `public int getHalf ( )`
- `public int getLow ( )`
- `public int getHigh ( )`

Druhu súčasťou objektu je systém asociácie kľúčov s rozsahmi vnútorného poľa, čo je realizované prostredníctvom vnútorných položiek objektu typu `String`. Tieto udržiavajú konkrétne označenie určitého rozsahu. Tieto položky sú:

- `String fullKey`
- `String halfKey`
- `String lowKey`
- `String highKey`

Mechanizmus je následne doplnený dvojicou metód určených k vyvolaniu špecifických obslužných metód na základe poskytnutého kľúča a to:

- `public int getValue ( String key )`
- `public void setValue ( int value , String key )`

Tieto metódy na základe kľúča identifikujú ktorá z interných metód prístupu k dátam má byť použitá. Je vhodné zmieniť, že kľúč môže taktiež mať hodnotu `null`, čo modeluje chovanie, kedy isté registre neposkytujú prístup k danému rozsahu (napríklad register *SP*, ktorý poskytuje prístup len k svojej šestnásť (16) a tridsaťdva (32) bitovej reprezentácii).

Pre potreby použitia taktiež objekt poskytuje skupinu metód, ktoré slúžia ako kontrola a to metódy pre identifikáciu, či kľúč náleží danému objektu a spôsob, ktorý určuje veľkosť asociovaného rozsahu v bajtoch. Tieto metódy slúžia primárne k identifikácii konkrétnej inštancie triedy a použité ako overovací mechanizmus v rámci interpretácie.

Následne sú tieto objekty agregované v objekte označenom ako RegisterFile, či registrový súbor, ktorého úlohou je zabezpečiť kontrolovaný prístup k existujúcim registrom v rámci dátového modelu, kedy jeden objekt zaobaľuje využitie metód skupiny objektov uložených v kolekcii.

4.1.2 Pamäťový súbor a Mapa prístupov

Pamäťový súbor je objekt, ktorý reprezentuje pamäť systému a jeho súčasťou je ďalší objekt, ktorý slúži k overeniu legality pamäťových prístupov. Takto vytvorený objekt predstavuje samostatný celok reprezentujúci pamäť systému (podobne ako singulárna inštancia registra predstavuje jeden register), tým pádom je zastúpený v celkovej štruktúre strojového súboru len raz.

Hlavnou súčasťou objektu predstavujúceho pamäť je pole typu `byte`, ktorého veľkosť predstavuje veľkosť adresovateľného priestoru. V súčasnej implementácii je pole obmedzené maximálnou veľkosťou poľa definovaného v rámci prostredia virtuálneho stroja jazyka Java.

Mapa povolení, ktorá kontroluje prístupy je taktiež realizovaná pomocou vnútorného poľa typu `byte`. Vnútorný mechanizmus je však upravený tak, že jednotlivé bity tohto poľa sú asociované s bunkami hlavnej pamäte a ich nastavením na jedna (1) sa odomyká možnosť použiť adresu. Veľkosť tohto poľa je určená veľkosťou hlavnej pamäte vzťahom:

$$sizeOfRestrictionMap = (sizeOfMemory/8) + 1$$

Mapa povolení poskytuje metódy, ktoré umožňujú odomykať a uzamykať použitie adries, identifikovať, či je daná adresa (prípadne rozsah adries)voľná pre použitie, alebo naopak, túto adresu v pamäti nie je možné využiť. Tieto metódy slúžia nielen ku kontrole prístupov do pamäte, ale taktiež predstavujú nástroje pre preklad, kedy je mapa upravovaná na základe alokačných požiadaviek definovaných v používateľskom kóde. Táto alokácia využíva algoritmu *First Fit*, kedy sekvenčne prechádza pamäťou pokiaľ nenájde blok adekvátnej veľkosti, ktorý nie je asociovaný s inou, predošlou požiadavkou na pamäťový rozsah v poli.

Pomocou tohto objektu je implementované chovanie hlavnej pamäte, ktorá poskytuje sadu prístupových metód pre ukladanie a načítanie hodnôt z pamäte. Tieto metódy požadujú nielen adresu, z ktorej budú dáta načítané, ale aj veľkosť dát (v implementácii sú definované veľkosti prístupov ako jeden (1), dva (2) a štyri (4) bajty).

Táto veľkosť je potrebná pre realizáciu obojsmerného prekladu hodnôt pomocou kódovania *Little–Endian*. Tieto hodnoty sú následne pomocou vnútorného poľa typu `byte` kódované do adekvátneho tvaru a na základe smeru prístupu sú buď uložené do hlavnej pamäte, alebo zaobalené pomocou funkcionality knižnice `NIO` a navrátené objektu volajúcemu tieto metódy. Prístupové metódy sú deklarované týmto spôsobom:

- `public int loadFromMemory ( int address, int size )`
- `public void storeToMemory ( int address, int size, int value )`

Tieto metódy taktiež generujú výnimku `MachineFileException`, ktorá slúži ako indikátor ilegálneho prístupu do pamäte počas vykonávania programu.

Pre potreby implementácie možnosti navrátenia strojového súboru do pôvodného stavu pred vykonaním programu, bol objekt reprezentujúci pamäť rozšírený o možnosť vytvoriť *snímku* obsahu pamäte. Vzhľadom k tomu, že pamäť je možné počas interpretácie inicializovať, ak používateľ využil pseudo–inštrukcií k tomu určených, prosté vynulovanie obsahu pamäte môže poškodiť tieto dáta. Z toho dôvodu je vytvorená snímka stavu pamäte po interpretácii. Takto vytvorený záznam je neskôr, na základe požiadavky na reštart simulačného experimentu, použitý k obnoveniu hodnôt v pamäti.

4.1.3 Príznakový súbor a Príznakový bit

Príznakový súbor je implementovaný na základe triedy `PseudoRegister32`, kedy modeluje svoj vnútorný obsah pomocou poľa typu `byte` s fixnou veľkosťou štyri (4), ktorá umožňuje interpretovať príznakový súbor ako priamu hodnotu dátového typu `int` a tiež nastaviť obsah registra príznakov pomocou tejto funkcionality.

Tento objekt je taktiež rozšírený o kolekciu objektov dvojíc, ktorá slúži ako identifikátor asociácie medzi indexom a kľúčom príznaku a obslužnými metódami, ktoré umožňujú kontrolovanú a korektnú inicializáciu (obmedzenia kladené na absenciu duplicitných kľúčov, či indexy mimo rozsahu vnútornej reprezentácie binárneho vektoru), získavanie a nastavovanie hodnôt jednotlivých bitov pomocou prístupových kľúčov.

Ako dátový typ reprezentujúci hodnotu bitu bol v implementácii zvolený typ `boolean`, teda dvoj–stavová hodnota, ktorá môže byť `true`, alebo `false`.

Prístup k týmto hodnotám je teda realizovaný pomocou metód, ktoré na základe použitého kľúču vystavia hodnotu bitu ako `boolean`, alebo naopak, umožnia hodnotu tohto bitu zmeniť. Táto manipulácia je realizovaná pomocou logických bitových operácií a kalkulácie pozície konkrétneho bitu prostredníctvom indexu asociovaného s kľúčom. Metódy sú deklarované ako:

- `public boolean getFlagValue ( String key )`
- `public void setFlagValue ( boolean value, String key )`

Tieto metódy zaobaľujú funkcionalitu, ktorá na základe indexu viažucého sa ku kľúču identifikujú konkrétnu položku vnútorného poľa typu `byte` a následne vykonajú logickú operáciu, ktorej výsledkom je získanie hodnoty konkrétneho bitu, alebo nastavenie tohto bitu.

4.1.4 Inštrukčný ukazovateľ

Inštrukčný ukazovateľ je implementovaný ako jednoduchá premenná typu `int`. Obsah tejto premennej bude udržiavať indexy objektov inštrukcií uložených v kolekcii, ktorá je súčasťou vnútornej reprezentácie programu. Strojový súbor poskytuje metódy na transformáciu hodnoty, ako je nastavenie novej hodnoty (v prípade ak inštrukcia vykonáva skok v kóde, alebo v stave na začiatku simulačného experimentu), alebo metódu na zvýšenie ukazovateľa o jedna (ktorá je vyvolávaná v rámci obsluhy systému na vykonávanie programu).

4.1.5 Inicializácia strojového súboru

Súčasťou strojového súboru je statická metóda, ktorá umožňuje inicializáciu pomocou špeciálneho objektu definujúceho obsah dátového modelu, teda konkrétne inštancie objektov a hodnoty, na základe ktorých budú vytvorené prostriedky pre simuláciu.

Vytvorenie tohto objektu a metódy bolo podmienené komplexným procesom nastavovania nového objektu a možnosti vytvárania variabilného dátového modelu, kedy je nutné zohľadniť vzťahy medzi jednotlivými vnútornými komponentmi. Tento objekt, ktorý je označený ako *konfigurácia* definuje všetky aspekty novo vytvorenej reprezentácie strojového súboru. Taktiež slúži pre potreby vykonávania simulácie, kde vystupuje ako zdroj legálnych adries jednotlivých blokov pamäte (definuje začiatočnú adresu oblasti pre dáta a oblasti, v ktorej je realizovaný zásobník).

Mimo týchto adries sprístupňuje dátovému modelu informácie o tom, ktorý z registrov slúži pre obsluhu zásobníka (nakoľko návrh umožňuje vytvoriť ľubovoľne pomenované registre a následne tieto registre použiť pre tieto operácie).

Ako bolo spomenuté vyššie, sekundárnou úlohou tejto konfigurácie je poskytnutie počiatočných adries v pamäti a rozsahy pamäťových blokov, ktoré slúžia k ukladaniu dát programu, alebo pre zásobník. K tomuto bolo použité jednoduchého modelu, kedy konfigurácia deklaruje charakteristiky pamäte a to:

A. Veľkosť jedného bloku pamäte

B. Počet blokov, ktoré predstavujú pamäť

C. Identifikátor, ktorý z blokov obsahuje počiatok pamäte dát

D. Počet kontinuálnych blokov pridelených pamäti dát

E. Identifikátor, ktorý z blokov obsahuje počiatok pamäte zásobníku

F. Počet kontinuálnych blokov pridelených pamäti zásobníka

S týmto je aj spojená validácia, či nedochádza k prekrývaní pamäťových rozsahov. Vďaka týmto dátam je potom konfigurácia schopná vykalkulovať veľkosti pamäťových prostredí, prvé a posledné legálne adresy pre prístupy.

4.2 Implementácia modulu interpretácie

Interpretačný modul predstavuje ďalší samostatne pracujúci celok, ktorý je zodpovedný za vytvorenie vnútornej reprezentácie používateľského programu založenej na vstupnom kóde.

Systém interpretácie je rozdelený na:

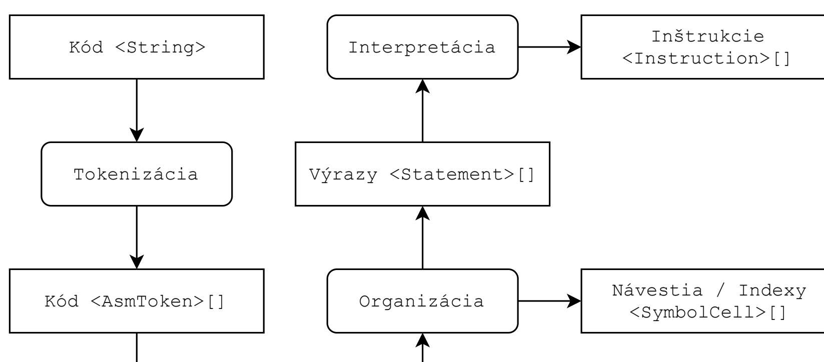
1. Všeobecná kontrola vstupov (či je vstup prázdny)
2. Rozdelenie vstupu na tokeny pomocou tokenizácie
3. Postupné spracovanie prúdu tokenov kódu programu a dátovej deklarácie na výrazy
4. Spracovanie výrazov dátovej deklarácie za účelom vytvorenia tabuľky symbolov
5. Prvý cyklus spracovania výrazov kódu programu za účelom kontroly a rozšírenia tabuľky symbolov o návestia
6. Druhý cyklus spracovania výrazov kódu programu za účelom vytvorenia objektov inštrukcií

Interpretačný systém používa doplnkových objektov, ktoré sú poskytnuté statickými metódami dátového modelu a abstraktnej továrne za účelom slovníkovej identifikácie špecifických tokenov. Ako bolo spomenuté v návrhu, lexikálna analýza identifikuje len numerické hodnoty, reťazce, špeciálne riadiace znaky a symboly. Množina tokenov identifikovaná ako symboly je následným porovnaním so slovníkovými objektami dodatočne identifikovaná, ak obsahuje niektorý z rezervovaných symbolov (identifikátory registrov, inštrukcií a špecifikácie veľkostí).

Objekt reprezentujúci výraz (`Statement`) je objekt, ktorý slúži ako obaľovacia štruktúra udržiavajúca tokeny viazané k špecifickému riadku.

Dvojnásobný prechod vstupných údajov zabezpečuje, že pri interpretácii konkrétnej inštrukcie má jej továrenský objekt k dispozícii všetky potrebné dáta pre to, aby mohol byť jej obsah vyhodnotený ako syntakticky a čiastočne sémanticky správny (použitie deklarovaných symbolov v algebraických výrazoch, existujúce návestia pre skokové inštrukcie).

Postupnosť tejto transformácie je možné vidieť na diagrame[4.2].



Obr. 4.2: Postupná transformácia dát postupom interpretácie.

V prípade, ak počas interpretácie nastane chyba (lexikálneho, syntaktického, alebo sémantického charakteru), interpretačný modul disponuje výnimkou `InterpretationError`, ktorá indikuje vzniknutú chybu a na základe stavu spracovania a zdroja chyby identifikuje, kde v používateľskom kóde táto chyba nastala (dvojica určujúca zdrojový riadok chyby a opis chyby).

Vzhľadom ku spôsobu interpretácie bol systém navrhnutý tak, aby bol schopný sa v rámci prechodu zotaviť z identifikovanej chyby a dokončiť interpretáciu. Aj keď kód nebude možné spustiť, používateľ bude oboznámený so všetkými chybami, ktoré bolo možné v rozsahu tohto behu interpretácie identifikovať.

Tu je nutné zmieniť, že v prípade, ak ku chybe došlo počas prvého prechodu, teda zlyhala lexikálna interpretácia tokenu, alebo výraz začína ilegálnym tokenom (nezačína symbolom návestie, identifikátorom inštrukcie), systém nepokračuje do druhého cyklu. Tým pádom nie sú odhalené chyby, ktoré by boli obsiahnuté v rámci analýzy jednotlivých výrazov špecifickými továrňami v druhom cykle a budú zobrazené až po odstránení chyby a zahájenia druhého, precízneho prechodu kódom.

Výstupom interpretácie je modifikovaný strojový súbor (o legálne prístupnú pamäť deklarovanú používateľom, ktorá môže byť inicializovaná) a vnútorná reprezentácia programu, ktorá disponuje špeciálnou položkou identifikujúcu index prvej inštrukcie a tabuľku symbolov.

4.2.1 Objekt DataBundle

Objekt `DataBundle` slúži ako zdroj dát pre program, ktoré sú známe až v momente interpretácie. Objekt DataBudndle je významný pre samotný proces interpretácie, nakoľko slúži ako tabuľka symbolov, ktoré definuje programátor a môže ich tým pádom využívať v rámci aplikačnej logiky sebou navrhnutého programu. Nakoľko je rozsah možností poskytovaný

simulačným prostriedkom relatívne úzky, objekt `DataBundle` poskytuje pole objektov, ktoré je lineárne prehľadávané.

Jednoznačným identifikátorom záznamu v takto implementovanej tabuľke symbolov je názov symbolu. K tomuto symbolu sa následne viažu položky využité logikou vykonávania programu a to:

- Hodnota, ktorá môže reprezentovať adresu v pamäti, alebo index inštrukcie vo vnútornej reprezentácii programu
- Identifikátor, či je daný symbol symbolickou adresou v pamäti, alebo návestím

Tento objekt sa potom stane súčasťou vnútornej reprezentácie programu a slúži k vykonávaniu simulačného experimentu ako jeden z dvoch primárnych zdrojov dát. Druhým je vyššie spomenutý objekt strojového súboru.

V rámci interpretácie objekt obsahuje špeciálnu položku, ktorá udržiava kontext globálneho návestia. Význam tohoto je pre prípad, ak by továrenská metóda interpretovala skokovú inštrukciu, ktorá by využívala lokálneho návestia. Vzhľadom k tomu, že si továrenské objekty nie sú vedomé kontextu inštrukcie (nakoľko interpretujú len jeden výraz), DataBundle zabezpečuje dodanie týchto informácii.

Pre potreby vykonávania taktiež objekt DataBundle poskytuje špeciálne pole obsahujúce dátové typy `int`, ktoré predstavujú indexy inštrukcií v kóde obsahujúce *break point*. Týmto inštrukciám predchádza špeciálny príkaz `#B` alebo `#break` a pre simulačný experiment tieto dáta predstavujú index konkrétnej inštrukcie, kedy má dôjsť k zastaveniu výkonu programu (ak tak určil používateľ vyvolaním adekvátnej funkcionality vykonávača kódu).

4.2.2 Továrne inštrukcií a Hlavná továreň

Každá interpretovateľná inštrukcia má vlastný továrenský objekt, ktorý zabezpečuje jej konkrétnu interpretáciu. K tomu slúži na tento účel definované rozhranie `Factory`, ktoré poskytuje podporné metódy:

- `public String getMyIdentifier ()`
- `public boolean isMyStatement ( Statement statement )`

Významom týchto metód je zabezpečiť interpretáciu, kde prvá z metód poskytuje identifikátor inštrukcie (potrebné pre tokenizáciu) a druhá metóda jednoznačne identifikuje, či daný výraz je interpretovateľný týmto konkrétnym objektom (teda, či identifikátor inštrukcie náleží tomuto objektu).

Následne objekt implementujúci rozhranie `Factory` poskytuje metódu, ktorá v prípade úspešnej interpretácie vráti objekt inštrukcie:

- `Instruction createInstruction(Statement s, DataBundle d, MachineFile m)`

Táto metóda používa ako vstupné parametre nielen samotný výraz, ale dva dodatočné objekty, z ktorých jeden reprezentuje tabuľku symbolov (a návestí), označený ako `DataBudndle` a reprezentáciu strojového súboru `MachineFile`. S využitím týchto parametrov je objekt schopný zabezpečiť korektnú interpretáciu inštrukcie, alebo v prípade chyby (či varovania) vystaví k tomu určenú výnimku `InterpretationError` a `InterpretationWarning`.

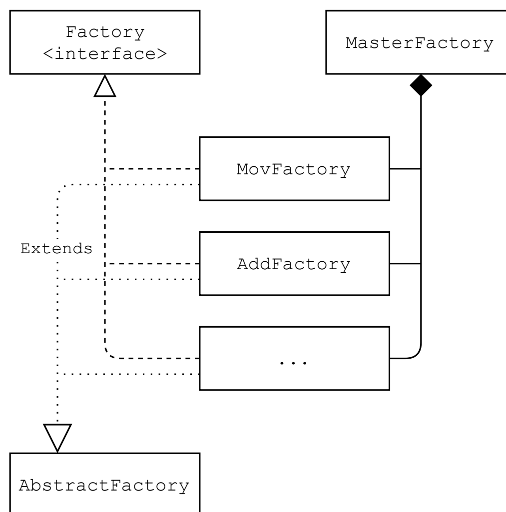
InterpretationWarning je špecifická situácia, kedy je kód interpretovateľný a spustiteľný, no môže vyvolať počas vykonávania chybový stav (špecificky sa jedná o špecifikáciu veľkosti pre register, ktoré je z hľadiska NASM ignorované a v prípade pamäťových prístupov môže spôsobiť prístup do ilegálnej, nepridelenej pamäti). Keďže je táto výnimka vyvolaná a tým pádom metóda nevráti štandardným spôsobom interpretovaný objekt inštrukcie, inštancia výnimky je rozšírená o položku, ktorá inštrukciu obsahuje.

Pre potreby zjednodušenia interpretácie bola vytvorená abstraktná trieda `AbstractFactory`, ktorá poskytuje všeobecné, na špecifickej inštrukcii nezávislé, metódy spracovania vstupného výrazu (teda objektu obsahujúceho postupnosť tokenov viazaných na špecifický riadok). K metódam tejto abstraktnej továrne pristupujú konkrétne továrenské objekty prostredníctvom dedenia.

Tieto metódy slúžia primárne k spracovaniu základného tela inštrukcie, ako napríklad rozloženie výrazu na identifikátor a dvojrozmerné pole tokenov, kde vonkajší index určuje operand a vnútorná sekvencia zase tokeny tohto operandu. K tomu sa taktiež viažu metódy pre vytvorenie vnútornej reprezentácie operandov z dostupných tokenov.

Továrenské objekty sú agregované v objekte hlavnej továrne, ktorá zabezpečuje prístup k adekvátnej továrni a taktiež poskytuje slovník inštrukčných identifikátorov lexikálnej analýze.

Štruktúru je možné vidieť na diagrame[4.3].



Obr. 4.3: Zjednodušený diagram tried zobrazujúci vzťahy hlavnej továrne a špecifických tovární.

4.2.3 Inštrukcia a Operandy

Produktom spracovania výrazu továrňou je objekt inštrukcie. Inštrukcia je definovaná rozhraním `Instruction`, ktoré poskytuje základnú metódu, ktorú musí objekt inštrukcie implementovať, aby ju bolo možné korektne vykonať v rámci simulačného experimentu:

- `public void execute ( MachineFile m, DataBundle d )`

Táto metóda je vyvolávaná v rámci vykonávania simulačného experimentu a jej definícia predstavuje opis chovania inštrukcie. Pre zachovanie kontroly nad prístupom sú teda ako parametre poskytnuté objekty, ktoré slúžia ako zdroje informácii potrebné pre vykonanie inštrukcie. V tomto prípade slúži objekt `DataBundle` ako zdroj reálnych hodnôt symbolických hodnôt (adresy v pamäti deklarované pomocou pseudo–inštrukcií a indexy inštrukcií viazané k návestiam). Objekt MachineFile zase slúži ako zdroj hodnôt uložených v dátovom modele (hodnoty registrov, pamäte a príznakov). Konkrétny výpočet implementovaných inštrukcií je možné nájsť v prílohe[C].

Pre zjednodušenie implementácie chovania inštrukcie bola vytvorená abstraktná trieda `AbstractInstruction` (analogická s `AbstractFactory`), ktorá poskytovala funkcie všeobecne využiteľné pri práci s dátami (napr.: kontrola najviac a najmenej významného bitu vychádzajúc z veľkosti operandu, maskovanie hodnôt ak nie je želaný textitbit padding a iné).

Táto metóda taktiež umožňuje vystaviť výnimku `InstructionExecutionError`, ktorá umožňuje identifikovať chybu vykonávania programu (špecifické chyby aritmetických operácií, alebo nelegálne prístupy do pamäte).

Objekt inštrukcie môže využívať objektov reprezentujúcich operandy. Tieto operandy sú objekty implementujúce rozhranie `Operand`, ktoré poskytuje metódy pre načítanie a uloženie hodnoty, či detekciu veľkosti a typu operandu. Samotné implementácie operandov poskytujú štyri základne zdroje informácii, ktoré sa v rámci návrhu vyskytujú v systéme a to:

A. Operand registra

B. Operand pamäte

C. Operand okamžitej hodnoty, alebo konštanty

D. Operand symbolickej hodnoty

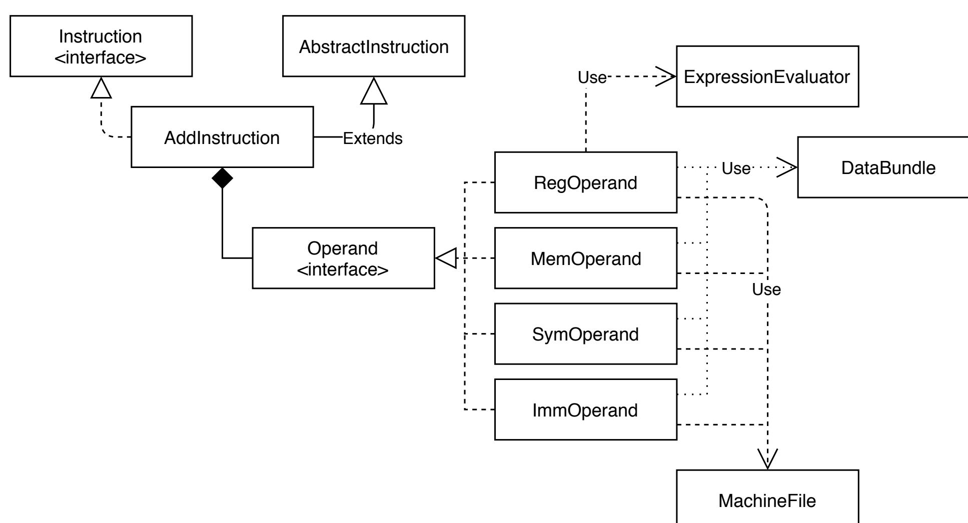
Realizácia získavania týchto hodnôt je riešená poskytnutím zdrojov informácii (objekty DataBundle pre informácie týkajúce sa programu a MachineFile pre informácie týkajúce sa modelovaného systému). Objekty ďalej slúžia ako adaptéry, nakoľko na základe svojej implementovanej logiky zabezpečia poskytnutie hodnoty štandardným spôsobom (teda ako dátový typ `int`).

Štruktúru toho ako pracuje tento subsystém a vzťahy medzi jednotlivými objektmi je demonštrovaný diagramom[4.4].

Významom takto vytvoreného riešenia je odľahčenie samotnej implementácie inštrukcie, nakoľko poskytnuté objekty operandov automatizujú prístup k dátam a vykonávajú vyhodnotenie a zachytenie prípadných chybových stavov (čo je realizované prostredníctvom mechanizmu výnimky `InstructionExecutionError`).

Špecifickým prípadom je operand zastrešujúci prístup do pamäte a operand obsahujúci okamžitú hodnotu, nakoľko tieto môžu obsahovať algebraické výrazy. Pre potreby analýzy a vyhodnotenia tohto objektu bol teda implementovaný vnorený systém, ktorý pracuje analogicky k hlavnému interpretéru. Konvertuje existujúce tokeny do špeciálnej formy aritmetických tokenov a umožňuje overenie ich správnosti. Tento test zahŕňa overenie syntaxe matematického výrazu, korektné párovanie zátvoriek a identifikácie, či obsahuje legálne matematické operandy.

Legálnymi matematickými operandmi sú operandy obsahujúce okamžité hodnoty, teda konštanty, numerické hodnoty. V prípade, ak je operand prístupom do pamäte (teda sa



Obr. 4.4: Zjednodušený diagram tried demonštrujúci vzťahy objektov inštrukcie.

začína tokenom asociovaným s „[“ a končí tokenom asociovaným s „]“), môže obsahovať operandy, ktorých hodnota sa môže dynamicky meniť počas vykonávania programu (registre), alebo hodnoty, ktoré sú známe až v momente interpretácie (symbolické hodnoty adries).

Analogicky k procesu, akým sú vyhodnocované algebraické výrazy v jazyku NASM, aj interpretér kladie obmedzenia vzťahy operácií a operandov. Delenie je umožnené len v prípade, ak sa netýka dynamicky sa meniacej hodnoty (symbol, alebo register), teda je tento vnorený výraz možné vyhodnotiť v momente prekladu programu.

Vyhodnotenie algebraických výrazov je realizované pomocou algoritmu *Shunting–yard* s rozšírením pre unárne operátory, využívajúci dvojice zásobníkov, jeden pre hodnoty, druhý pre operandy.

4.2.4 Interpretácia pseudo–inštrukcií a Továreň pamäte

Špecifickým prípadom interpretácie je spracovanie používateľských deklarácii na pamäť ich programu. Tieto informácie sú analogicky spracované podobne ako hlavný používateľský program, no namiesto vytvárania objektov inštrukcií dochádza na základe interpretácie výrazov k zmenám objektu reprezentujúceho pamäť systému.

Tieto výrazy sa skladajú z dvoch rodín pseudo–inštrukcií a to:

`RESx (RESB, RESW a RESD)`, ktoré na základe operandu deklarujú veľkosť požadovanej pamäte v bajtoch

`Dc (DB, DW a DD)`, ktoré deklarujú obsah pamäte (kde je veľkosť požadovanej pamäte realizovaná počtom operandov)

Spracovanie tohto vstupu je analogické so spracovaním kódu, kedy sa buduje mapa návestí (v tomto prípade symbolických hodnôt adries v pamäti dát) a na základe spracovania požiadavky na pamäť je jej pridelená hodnota, teda reálna adresa v reprezentácii pamäte.

Tieto hodnoty generujú interpretáciou požiadavky na pamäťový súbor, ktoré továreň pamäte interpretuje ako požiadavky na súvislý blok pamäte konkrétnej veľkosti. Pamäťový súbor následne obstará alokáciu (v prípade nedostatku pamäte vyvolá chybu interpretácie) a navráti adresu začiatku bloku. Súčasťou tejto alokácie je aj úprava prístupových informácii v mape legálnych prístupov obsiahnutých v pamäťovom súbore. Následne pamäťová továreň asociuje symbol s touto adresou v objekte `DataBundle`.

V prípade ak je požiadavka na inicializovanú pamäť, pamäťová továreň obstará inicializáciu hodnôt v pamäti spracovaním operandov pseudo–inštrukcie rodiny `Dx`. Tieto operandy musia byť dátami, ktoré je možné interpretovať ako okamžité, numerické hodnoty.

Inicializácia pamäte prebieha spôsobom, kedy je (na základe povahy použitej pseudo–inštrukcie) zvolená veľkosť elementu a následne je sekvencia operandov uložená ako pole hodnôt v pamäti. Ak je, napríklad, použitá inštrukcia `DW` a nasleduje za ňou postupnosť numerických hodnôt, každá z týchto hodnôt je mapovaná do pamäte ako element poľa veľkosti dvoch (2) bajtov.

Špeciálnym prípadom je spracovanie reťazcov znakov, ktoré sú vždy ukladané do pamäte sekvenčne (po bajtoch) a pamäť je na základe veľkosti viažúcej sa k pseudo–inštrukcii, zarovnaná na násobok veľkosti. Napríklad, ak je použitý výraz:

```
oneChar: DD 'a'
```

V pamäti bude vykonaná alokácia veľkosti štyroch (4) bajtov a na prvom bajte v pamäti bude uložená numerická reprezentácia symbolu „a“.

4.2.5 Vnútorná reprezentácia programu

V prípade úspešnej interpretácie poskytne modul vnútornú reprezentáciu programu, ktorá slúži spolu so strojovým súborom použitým počas interpretácie ako základ pre simulačný experiment. Tento objekt disponuje trojicou položiek:

A. Objekt `DataBundle`, ktorý slúži ako tabuľka symbolov

B. Pole objektov inštrukcií

C. Počiatočná adresa programu

Táto sekcia slúži primárne ako zhrnutie základných informácií o vnútornej reprezentácii programu z predošlých sekcií, ktorá sa v rámci procesu interpretácie odkazovali na tento objekt.

DataBundle je špecifickým objektom slúžiacim ako tabuľka symbolov deklarovaných používateľom a definovaných interpretáciou. Jedná sa o kolekciu lineárne zoradených trojíc dát, ktoré obsahujú reťazec identifikujúci symbol, hodnotu dátového typu `int`, ktorá udržiava hodnotu pridelenú počas interpretácie (adresu v pamäti, alebo index inštrukcie nasledujúcej za návestím) a identifikátor, či je daný symbol adresou v pamäti, alebo indexom inštrukcie (dátového typu `boolean`).

DataBundle taktiež obsahuje špeciálne pole, ktoré indikuje, či je možné konkrétny index viažúci sa k inštrukcii použiť ako *break–point*, teda bod, kedy je vykonávanie programu zastavené, ak obslužné metódy pre vykonávanie používajú tejto funkcionality.

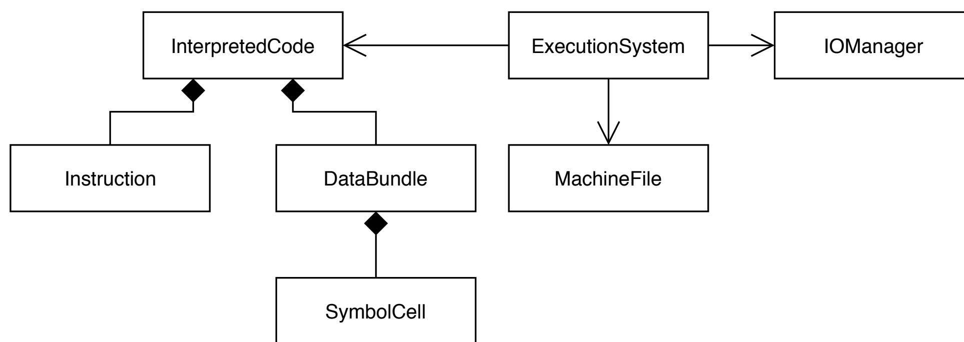
Pole objektov inštrukcií je objekt typu `ArrayList`, ktoré udržiava referencie na inštancie konkrétnych inštrukcií a poskytuje tak vykonávaciemu modulu k týmto objektom prístup

pomocou hodnoty inštrukčného ukazovateľa obsiahnutého v reprezentácii strojového súboru.

Počiatočná adresa programu je prvá vykonateľná inštrukcia, ktorá nasleduje globálne návestie `main:`. Táto adresa je odvodená z tabuľky symbolov v rámci záverečnej fázy interpretácie, kedy prebieha príprava objektu vnútornej reprezentácie programu.

4.3 Implementácia modulu vykonávania

Modul vykonávania je objekt zodpovedný za exekúciu vnútornej reprezentácie programu, pričom spôsob, akým je tento simulačný experiment vykonaný je riadené používateľom. Exekučný systém požaduje v rámci svojej inicializácie prípravu na simulačný experiment, teda mu musia byť poskytnuté objekty reprezentujúce používateľský program a objekt reprezentujúci strojový súbor. Tieto vzťahy je možné vidieť na diagrame[4.5].



Obr. 4.5: Zjednodušený diagram tried demonštrujúci vzťahy objektov vykonávajúcich simuláciu.

Na základe tejto inicializácie dôjde k príprave simulačného experimentu (nastavenie počiatočnej adresy a zaistenie *snímky* pamäte strojového súboru pre potreby reštartu experimentu). Následne, na základe požiadaviek používateľa zaisťuje tento modul vykonávanie programu, teda simulačného experimentu a to pomocou týchto metód:

- `public void executeStep ()`
- `public void executeRun ()`
- `public void executeContinue ()`
- `public void restartExecutor ()`

Ako naznačujú názvy metód, tieto umožňujú používateľovi vykonať jeden krok, teda jednu inštrukciu (`Step`), vykonať beh programu do konca od aktuálne vykonávanej inštrukcie (`Run`), vykonať beh programu od aktuálne vykonávanej inštrukcie po najbližšie prerušenie, alebo koniec programu (`Continue`), alebo reštartovať simulačný experiment.

Metódy, ktoré riadia vykonávanie programu (teda všetky vyššie zmienené mimo metódy `restarExecutor`), môžu vystaviť výnimku `ExecutionTerminationException`. Táto výnimka vzniká v prípade, ak program dospel ku koncu (vyvolanie inštrukcie `RET` nad

prázdnym zásobníkom), nastala počas vykonávania chyba, alebo vykonávanie programu prekročilo maximálny povolený počet krokov.

Vykonanie jedného kroku simulácie je základnou funkcionalitou, ktorá je následne využitá ostatnými metódami (teda beh programu je opakované vyvolávanie kroku simulácie, až pokým nie je dosiahnuté situácie ukončenia simulačného experimentu).

Algoritmus vykonania programu je nasledovný:

1. Overenie, či obsah inštrukčného ukazovateľa je platným indexom poľa inštrukcií
2. Načítanie inštrukcie, ktorá sa má vykonať (inštrukcia na indexe rovnom obsahu inštrukčného ukazovateľa)
3. Zvýšenie inštrukčného ukazovateľa o jedna (1)
4. Overenie, či súčasný počet krokov nepresiahol maximálny počet krokov
5. Vykonanie inštrukcie
6. Overenie, či obsah inštrukčného ukazovateľa je platným indexom poľa inštrukcií (overenie navyše potrebné pre zachovanie stability používateľského rozhrania)

Implementácia tohto vykonávacieho systému nerozlišuje medzi korektným a nekorektným ukončením simulačného experimentu, nakoľko výsledkom oboch stavov je odopretie vykonávania ďalších zmien nad dátovým modelom a pre oba prípady je nutné simulačný experiment reštartovať, alebo ukončiť návratom do editora kódu.

Riadenie toho, či simulácia skončí, alebo nie je dosiahnuté vystavením výnimky `ExecutionTerminationException` kontroléru zahrnutom v používateľskom rozhraní.

4.3.1 Vstupno Výstupný manažér a zabudované procedúry

Aplikácia bola v neskorších fázach implementácie rozšírená o vstupno výstupné operácie. Tieto umožnili používateľovi, pomocou vyvolania k tomu ručených procedúr, interakciu s dátovým modelom programu počas vykonávania simulačného experimentu. Taktiež slúžili pre výpis dát a informácii do konzoly obsiahnutej v grafickom používateľskom rozhraní.

Toto chovanie bolo realizované prostredníctvom špeciálnej výnimky `IOInterruption`, ktorá vykonávaciemu systému oznámila, že program požaduje interakciu s grafickým používateľským rozhraním mimo štandardného vykonávania simulácie. K obsluhe tejto výnimky slúži špeciálny objekt `IOManager`, ten poskytuje vstupno výstupné operácie v spolupráci s grafickým používateľským rozhraním. Tieto operácie sú pre vstup ošetrené modálnym oknom dialógu a pre prípad výstupu je vyvolaný zápis do konzoly obsiahnutej v *GUI* (*Graphical User Interface*).

Výnimka obsahuje numerický kód identifikujúci požiadavku na službu (ktorých sumarizácia sa nachádza v prílohe[D]), vykonávací modul následne kód tejto služby poskytne objektu pre manažment vstupov a výstupov. Ten umožní vykonať obsluhu tejto požiadavky a po úspešnej obsluhe sa program navráti k predošlej forme vykonávania programu.

Ak obsluha vytvorí chybu (napríklad ilegálnym prístupom do pamäte), je táto chyba chápaná ako chyba simulovaného programu a simulačný experiment je ukončený štandardným chybovým výstupom.

4.4 Implementácia modulu používateľského rozhrania

Grafické používateľské rozhranie bolo implementované pomocou nástrojov knižnice Swing s využitím natívnych, zdokumentovaných komponentov. Výnimkou je objekt `TextLineNumber`, ktorý rozširuje klasické objekty `JTextArea` o číselník riadkov. Tento komponent bol prevzatý z už existujúceho riešenia, čo je viditeľne označené v kóde.

Používateľské rozhranie je vyvolané objektom `GuiSystem`, ktorý je inicializovaný a aktivovaný v momente spustenia aplikácie. Tento objekt zabezpečuje správu rozhrania a na inicializáciu prvkov rozhrania. Základná štruktúra rozhrania sa skladá z dvoch obrazoviek, medzi ktorými je prepínané na základe požiadaviek od používateľa.

Organizácia obsahu týchto obrazoviek je dosiahnutá pomocou manažéra rozloženia `GridBagLayout` a obsah samotný sa skladá z klasických swingových objektov ako `JTextArea`, `JTextField`, `JLabel`, `JButton` a pod. Pre potreby tohto aplikačného riešenia neboli vytvorené žiadne špeciálne komponenty (s výnimkou TextLineNumber, ako bolo spomenuté vyššie).

V prípade, ak je systém spustený v lokalite, ktorá neobsahuje špeciálnu zložku obsahujúcu súbory s príkladmi, systém zabezpečí vytvorenie tejto zložky spolu s obsahom v rovnakom adresári, v akom sa nachádza .jar archív.

4.4.1 Obrazovka Editoru

Obrazovka, ktorá je automaticky zobrazená po spustení aplikácie dáva používateľovi k dispozícii tieto komponenty:

A. Textové komponenty pre používateľský vstup

B. Textové komponenty pre zobrazenie výstupu interpretácie a správ aplikácie

C. Tlačidlá pre kontrolu interpretácie a simulácie

D. Menu pre riadenie interakcie so súborovým systémom a otváranie príkladov

Táto obrazovka poskytuje funkcionalitu pre vyvolanie interpretácie, spustenie simulácie (ak je možné interpretovať používateľský vstup) a pre načítanie/uloženie z/do súboru v používateľskom systéme. Toho je docielené spracovaním používateľských vstupov ako serializovateľných objektov a využitím vstupných a výstupných tokov.

Obsluha interpretácie (stlačenia k tomu určeného tlačidla) je vykonaná vytvorením novej inštancie strojového súboru a následne vytvorením interpretačného objektu. Tento spracuje používateľský vstup a v prípade chyby vystaví obsah zachytenej výnimky do textovej komponenty slúžiacej ako konzola.

Proces sa opakuje pre obsluhu vyvolania simulácie, kedy je objekt zodpovedný za správu obrazoviek požiadaný o vytvorenie obrazovky simulácie a sú mu dané k dispozícii objekty reprezentujúce používateľský program a strojový súbor.

Inštancia objektu reprezentujúcu používateľskú obrazovku je počas simulácie zachovaná, takže pri ukončení simulácie nedochádza k novej inicializácii obsahu prvej obrazovky, ale je dosadená z vnútornej položky objektu `GuiSystem`, ktorý ju uchováva.

4.4.2 Obrazovka Simulátoru

Obrazovka, ktorá je zobrazená pre potreby vykonávania simulačného experimentu. Táto obrazovka, až na špeciálne prípady, nedáva používateľovi možnosť textového vstupu a je tak

ovládaná cez tlačidlá. Špeciálnym prípadom sú modálne okná, ktoré sú vyvolané v prípade obsluhy zabudovaných vstupno–výstupných procedúr.

Obsah tejto obrazovky je aktualizovaný v špecifických momentoch a to:

- Po požiadavke na vykonanie kroku
- Po požiadavke na vykonanie behu programu do konca
- Po požiadavke na vykonanie behu programu po najbližší *break point*, alebo do konca
- Po požiadavke o reštart
- Ak počas vykonávania simulácie nastane chyba

Simulátor poskytuje komplexné rozhranie pre sledovanie a kontrolu simulačného experimentu. Toho je docielené prostredníctvom štyroch hlavných prvkov:

- Prehľad Vykonávaného programu
- Inšpektor dátového modelu
- Konzola obsahujúca výstup programu a správy simulátora
- Panel s kontrolnými tlačidlami

Vykonávaný program je zobrazovaný tak, ako ho zadal programátor. Jedná sa o reprezentáciu originálneho vstupu, ktorá je doplnená o špeciálny objekt schopný asociácie obsahu inštrukčného ukazovateľa a riadku programu. Tohoto je využité, keď rozhranie zvýrazňuje riadok v kóde, ktorý obsahuje inštrukciu vykonanú v nasledujúcom kroku. Tento riadok je zvýraznený žltou farbou.

Kvôli rozsahu bude inšpektor dátového modelu popísaný v samostatnej sekcii.

Konzola obsahujúca výstup programu je formátovateľný textový komponent, ktorý je využívaný kontrolórom obsiahnutom v grafickom používateľskom rozhraní a objektom `IOManager` zodpovedným za správu vstupno výstupných operácií.

Panel s kontrolnými tlačidlami umožňuje používateľovi riadiť, reštartovať a ukončiť vykonávanie simulačného experimentu. V prípade ukončenia simulačného experimentu je celá obrazovka zahodená, nakoľko je jej inicializácia viazaná na konkrétne inštancie strojového súboru a interpretovaného kódu.

4.4.3 Inšpektor dátového modelu

Primárny zdroj informácii o vykonávanom simulačnom experimente sa skladá z panelu obsahujúceho záložky. Každá zo záložiek predstavuje jeden špecifický pohľad na stav systému. Obsahuje tieto záložky:

A. Inšpektor registrov

B. Inšpektor pamäte

C. Inšpektor pamäte ako polí

D. Inšpektor zásobníka

E. Inšpektor premenných

Inšpektor registrov

Inšpektor registrov zobrazuje všeobecné registre dostupné zo strojového súboru ako stĺpec dlaždíc (kde jedna dlaždica predstavuje panel s jedným objektom typu `PseudoRegister32`). Každá z týchto dlaždíc zobrazuje na základe dostupných kľúčov hodnoty obsiahnuté v daných rozsahoch vnútornej reprezentácie hodnoty uloženej v registry.

Tieto zobrazenia taktiež disponujú vlastným polom objektov typu `JRadioButton`, ktoré umožňujú pre každý zo zobrazených registrov určiť akým spôsobom sú ich dáta reprezentované (binárne, oktálne, decimálne so znamienkom, decimálne bez znamienka a hexadecimálne).

Inšpektor pamäte

Inšpektor pamäte zobrazuje pamäť ako postupnosť bajtov. Táto postupnosť je teda realizovaná, ako náhľad do pamäte bez dekódovania hodnôt prostredníctvom *Little–Endian* kódovania (dáta sú teda zobrazená formou *Big–Endian*). Dáta sú zobrazené ako text, organizované po štvoriciach bajtov, ktorým predchádza adresa v pamäti v hexadecimálnom formáte.

Jednotlivé bajty sú farebne zvýraznené podľa toho, či sa jedná o adresy prístupné programu (zelené podfarbenie), alebo adresy, ku ktorým nemá prístup (červené podfarbenie).

Používateľ môže v rámci inšpektora pamäte prepínať medzi spôsobom, akým sú bajty zobrazené, teda či budú zobrazené ako binárne, oktálne, decimálne so znamienkom, decimálne bez znamienka, hexadecimálne a ako znak podľa ASCII tabuľky.

Inšpektor pamäte ako polí

Inšpektor pamäte ako polí je nástroj, ktorý umožňuje obsah pamäte reprezentovať ako pole hodnôt. Používateľ musí zvoliť parametre poľa, ktoré predstavujú počiatočnú adresu (definovanú symbolom, alebo výrazom), veľkosť prvku poľa (jeden (1), dva (2), alebo štyri (4) bajty) a počtom prvkov poľa. Na základe tejto definície je potom pole zobrazené, pričom hodnoty sú načítané a dekódované prostredníctvom *Little–Endian*.

Inšpektor zásobníka

Inšpektor Zásobníka umožňuje používateľovi skúmať obsah pamäte, v ktorom je realizovaná abstraktná štruktúra zásobníka. Pomocou tabuľky sú zobrazené nielen hodnoty a adresy, na ktorých sú dané hodnoty uložené, no je doplnený aj o kontext vzhľadom k obsahom registrov súvisiacich s obsluhou zásobníka.

Inšpektor teda zobrazuje na ktorú z *buniek* zásobníka ukazujú registre určené ako ukazovatele na vrchol a bázu zásobníka. Mimo toho taktiež umožňuje meniť bázu, v ktorej sú zobrazované dáta analogicky k spôsobu, akým ich implementuje inšpektor registrov.

Inšpektor premenných

Inšpektor premenných je posledným z nástrojov, ktoré umožňujú skúmať obsah strojového súboru. Tento systém, taktiež s využitím tabuľky, umožňuje používateľovi vytvárať náhľady na špecifické súčasti systému.

Tieto náhľady sa chovajú analogicky k tomu, ako sú implementované operandy, čiže si v jednom okne môže používateľ zobraziť hodnotu pamäťovej lokality, alebo dáta obsia-

hnuté v niektorom z registrov. Výhodou je, že takto vytvorené pohľady na špecifické sekcie dátového modelu sú agregované na jednom mieste.

Kapitola 5

Testovanie

Táto kapitola pojednáva o testovaní aplikačného riešenia. Vychádzajúc z materiálov agregovaných pod *International Software Testing Qualifications Board*[1], bola za testovaciu stratégiu zvolená trojica samostatných prístupov, ktorých cieľom bolo zaručiť vysokú mieru spoľahlivosti aplikačnej logiky a funkcionality, spolu s testovaním vhodnosti použiteľnosti z hľadiska používateľa.

Tohoto bolo dosiahnuté tromi testovacími celkami:

A. Komponentové testovanie funkcionality dátového modelu, interpretácie, reprezentácie inštrukcie a vykonávania programu pomocou knižnice `JUnit`

B. Exploratívne testovanie programátorom a vybranou vzorkou používateľov

C. Anonymné testovanie programu pomocou testovacieho protokolu

5.1 Komponentové testovanie

Z hľadiska aplikácie bolo chovanie jednotlivých komponent testované pomocou testovacích tried zahrnutých v riešení tohto projektu. Ako hierarchia testovania bola zvolená abstrakcia, teda postup od najnižších komponent k najvyšším z hľadiska štruktúry aplikačného riešenia.

Pre komponenty, ktorých vnútorná logika je primárne zameraná na aritmetické operácie, boli primárne zvolené prierezy hodnôt s dôrazom na hraničné hodnoty pre odhalenie chýb a defektov spôsobených abstrakciou nízko-úrovňových procesov do prostredia vyššieho programovacieho jazyka.

Komponenty pracujúce s interpretáciou vstupného programu boli testované vyčerpávajúcim porovnávaním možných mutácií vstupu a porovnávaním výsledkov s očakávanými hodnotami. Toto bolo aplikované ako pre testovanie tokenizácie vstupu, tak aj spracovania už existujúcich tokenov podľa pravidiel syntaxe vzťahujúcej sa k základnej štruktúre programu, či konkrétnej inštrukcie.

Testy boli vytvorené a implementované pomocou knižnice `JUnit` a predstavujú vyčerpávajúce testovanie ako korektných vstupov a operácií, tak aj testovanie chybných vstupov a ilegálnych operácií. Cieľom tohoto bolo zaistiť čo najvyššiu mieru robustnosti systému a jeho schopnosti indikovať ilegálny stav (ako napríklad vytvorenie registrov, ktorých množiny prístupových kľúčov sa prekrývajú, negatívnych veľkostí pamätí, nezmyselných výrazov a podobne).

Cieľom tohto testovania bolo zaistenie očakávaného chovania a detekcie chýb a chybami produkovaných defektov.

Toto testovanie je označené za testovanie komponent, nakoľko je analyzované chovanie jednotlivých programových štruktúr ako *čierne skrinky* (anglicky *black box*). Dôvodom je to, že testovanie metód a premenných, ktoré boli v rámci návrhu označené ako privátne by bolo možné len vytvorením reflexie, čo je však na druhú stranu protichodné voči zamýšľanému návrhu.

Negatívny dopad takto zvoleného spôsobu testovania je však vo veľkej miere eliminovaný samotným designom systému, nakoľko overenie funkcionality najmenších, elementárnych súčastí zároveň aj verifikovalo chovanie skryté vo väčších, komplexnejších objektoch, ktoré sa skladajú z týchto súčastí.

Testovanie projektu je možné spustiť prostredníctvom príkazu `ant test` (za predpokladu, že je na systéme, na ktorom je test vykonávaný prítomný program *ant*).

5.2 Exploratívne testovanie

Exploratívne testovanie bolo riešené formou pohovorov v kontrolovanom prostredí, kedy boli používatelia inštruovaní k vykonávaniu špecifických úloh. Tieto úlohy boli založené na testovacích protokoloch, ktoré boli zámerne formulované všeobecne za účelom odhalenia potencionálnych problémov v návrhu rozhrania.

Testovacie protokoly boli založené na štandardnom prístupe k používaniu simulačného prostriedku, teda zahrňovali úkony týkajúce sa prípravy simulácie (tvorenie kódu a odhaľovanie chýb) a samotného vykonávanie simulačného experimentu (vykonávanie programu).

Testovacie podmienky zahrňovali špecifické počiatočné stavy testovania ako:

- Žiaden počiatočný stav, teda aplikácia po spustení bez akýchkoľvek predošlých dát
- Existujúce programy, kedy bolo úlohou používateľa otestovať schopnosť aplikácie zobraziť chyby zanesené vo vstupnom kóde
- Pripravený simulačný experiment, kedy bolo úlohou používateľov identifikovať vzťah medzi simulovaným programom a vstupným programom

Cieľom tohoto testovania bolo identifikovať defekty spôsobené používateľským vstupom (nielen vstupným programom, ale aj ich interakciou s aplikačným riešením) a indikovať nedostatky v návrhu a implementácii používateľského rozhrania. Tieto testovacie protokoly a výsledky sú súčasťou prílohy na pamäťovom médiu.

Súčasťou tohoto exploratívneho testovania bola taktiež aj analógia na používateľské príbehy (anglicky *User Stories*), či *biznis procesy*. V základnej štruktúre opisovali chovanie používateľa, alebo scenár situácie v kombinácii so žiadanou reakciou aplikácie. Súčasťou týchto protokolov boli teda nielen počiatočné body zmienené vyššie, ale aj špecifikácia postupnosti úloh a úkonov s definíciou akceptačných kritérií.

5.3 Anonymné testovanie dotazníkom

Toto testovanie slúžilo ako primárna indikácia vhodnosti riešenia grafického používateľského rozhrania. Cieľom bolo identifikovať ako boli používatelia schopný bez predošlej inštruktáže, či len s obmedzenými znalosťami systému riešiť predstavené problémy. Vyhodnotenie týchto testov bolo numerické, formou dotazníku, za účelom agregácie výsledkov do zmysluplných dát.

testovanie prebiehalo na pracovných staniciach používateľov, kedy obdržali archív s *.jar* súborom a tromi súbormi slúžiacim pre testovanie. Tieto súbory boli:

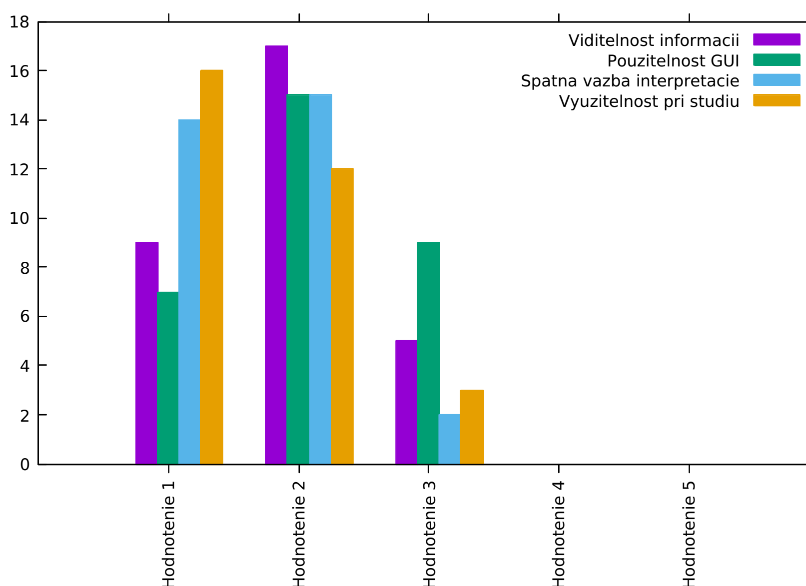
1. `readme.txt`, ktorý obsahoval základnú inštruktáž o použití protokolu a tom, ako overiť, či je prostredie schopné spustiť aplikáciu a samotné spustenie aplikácie.
2. `Protokol.txt`, ktorý obsahoval detailný opis jednotlivých experimentov a testov.
3. `Dotazník.txt`, ktorý obsahoval miesto pre vyhodnotenie testovania.
4. `test_files`, teda adresár obsahujúci dáta pre testovanie aplikácie protokolom.

Súčasťou testovacieho balíka boli taktiež príkladné súbory, teda špecifické testovacie súbory, na ktorých bola demonštrovaná funkcionalita vrámci testovania. Tieto slúžili na demonštráciu špecifických spôsobov zobrazovania dát a vykonávania programu a sú súčasťou prílohy na pamäťovom médiu ako celý testovací archív.

Forma a obsah dotazníku, spolu s vyplnenými dotazníkmi sú taktiež súčasťou prílohy na priloženom pamäťovom médiu.

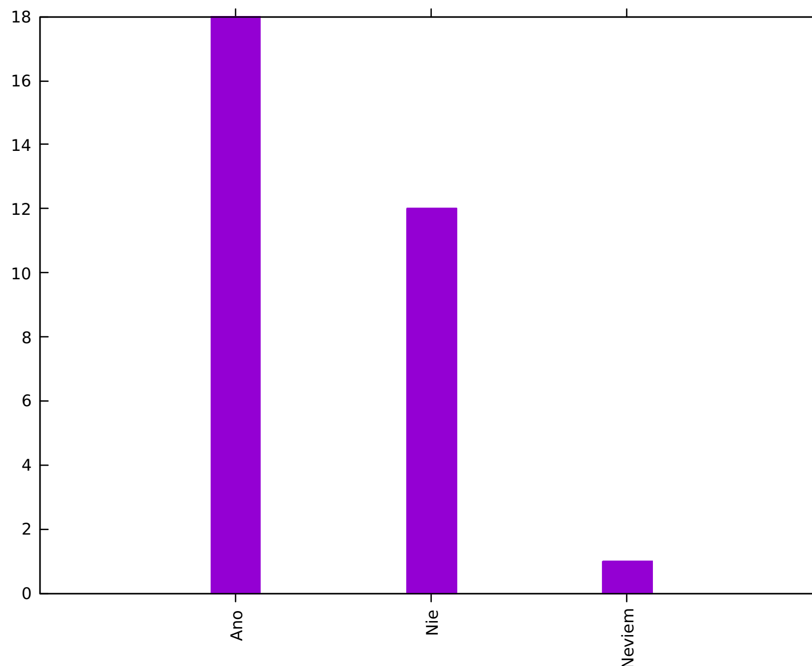
5.4 Vyhodnotenie testovania

Na základe výstupu z automatizovaného testovania vnútornej logiky boli identifikované a odstránené nedostatky, ktoré vznikli počas návrhu a implementácie. Tieto chyby boli často spôsobené iteratívnym prístupom k vývoju samotnej aplikácie.



Obr. 5.1: Vyhodnotenie jednotlivých aspektov používania aplikácie respondentmi. Hodnotenie ako v škole, teda 1 (najlepšie) a 5 (najhoršie)

Využitím exploratívneho testovania boli používateľmi a programátorom identifikované nedostatky, či defekty v grafickom používateľským rozhraní, ktoré boli primárne spôsobené defektným (či chýbajúcim) testovaním vstupných hodnôt a boli odstránené.



Obr. 5.2: Posúdenie intuitívnosti grafického rozhrania respondentmi.

Na základe spätnej väzby z testovania dotazníkom a exploratívneho testovania, bola vyhodnotená použiteľnosť používateľského rozhrania ako dostatočná pre potreby vzdelávania a pochopenia principiálnej funkcionality vykonávania programu na modelovanom systéme. Používatelia taktiež demonštrovali schopnosť pochopiť vzťahy medzi zobrazovanými dátami a spôsobom, akým mali dopad na vykonávaný experiment (ovplyvnenie výsledkov operácii počiatočnými hodnotami a stavy príznakov).

Vyhodnotenie otázok by sa, vzhľadom k ich povahe, ako známky dalo vyjadriť priemernou známkou pre danú tému a to:

- *1,87* je priemerná známka udelená viditeľnosti informácií o dátovom modele
- *2,06* je priemerná známka udelená použiteľnosti grafického rozhrania
- *1,61* je priemerná známka spätnej väzbe interpretácie používateľského vstupu
- *1,58* je priemerná známka udelená využiteľnosti pri štúdiu

Ako ďalším výsledkom je vhodné zmieniť pomer odpovedí na intuitívnosť rozhrania a to *18 ÁNO* proti *12 NIE*, čo indikuje, že návrh a implementácia rozhrania más voje nedostatky z hľadiska podvedomého pochopenia funkcionality. Jeden z respondentov vybral odpoveď „neviem“, ktorá nebola zarátaná do pomeru.

Agregované výsledky z prieskumu dotazníkom je možné vidieť na sérii grafov, ktoré predstavujú vyhodnotenie testov na grafoch opisujúcich viditeľnosť informácií, spätná väzba interpretácie a použiteľnosť aplikácie pri štúdiu[5.1]. Na ďalšom grafe je možné vidieť zobrazenie intuitívneho použitia[5.2]. Súčasťou dotazníka bolo aj to, či by respondenti nástroj použili ako pomôcku pri štúdiu. Tu uniformne odpovedali *áno* všetci z opýtaných.

Kapitola 6

Záver

Na základe informácií z predošlých kapitol je možné zhrnúť, že táto práca bola úspešná v návrhu a implementácií nástroja, ktorý slúži ako simulátor vykonávania programu v prostredí procesorov rodiny x86.

Analýza problematiky bola vyčerpávajúca a založená na technických manuáloch, odborných textoch týkajúcich sa programovania v jazyku symbolických inštrukcií a výňatkoch pojednávajúcich o konkrétnych aspektoch modelovaného systému. Túto informačnú bázu bolo doplnené o experimentovanie s reálnym systémom a nástrojoch z prostredia *Netwide Assembler*, pričom bolo cieľom zistiť, ako sa systém zachová v určitých aspektoch, ktoré neboli v dokumentácii plne objasnené.

Na základe spätnej väzby od používateľov, ktorí sa zúčastnili testovania je možné určiť, že nástroj je reálne využiteľný, aj keď sú isté jeho aspekty ťažkopádne a vyžadujú si zvyknúť na jeho využívanie. Toto je založené primárne na spätnej väzbe, kde používateľa subjektívne hodnotili využiteľnosť grafického používateľského rozhrania.

Vývojový proces bol náročný, nakoľko bol zvolený iteratívny model, kedy bolo cieľom dosiahnuť čo najrýchlejšie použiteľného prototypu, ktorý mohol byť testovaný, vyhodnotený a následne bola na jeho základoch stavaná ďalšia funkcionalita. Toto sa nezaobišlo bez problémov. Počas procesu vývoja bolo nutné dva krát kompletne zmeniť vnútornú štruktúru programu a vytvoriť nový návrh a implementáciu, nakoľko pôvodné riešenie neposkytovalo elegantné riešenie vznikajúcich problémov. Príkladom môže byť spôsob využívania výnimiek v rámci interpretačného procesu, ich funkcionalitu predtým realizovala systém informačných objektov, čo však zaťažovalo riadiacu a vyhodnocovaciu logiku komplexným protokolom pre predávanie správ a kontrolu správ.

Zadanie samotné taktiež prešlo vývojom, kedy sa z jednoduchého simulačného prostriedku na demonštrovanie chovania sekvencie inštrukcií stal systém, ktorý je schopný demonštrovať komplexné algoritmy zapísané podmnožinou implementovaných inštrukcií. Taktiež bolo nutné vyvinúť grafické používateľské rozhranie, ktoré zodpovedá bežným očakávaniam používateľa, je schopné interakcie so súborovým systémom a poskytuje funkcionalitu jednoduchého IDE (*Integrated development environment*) a taktiež poskytuje schopnosť počas vykonávania analyzovať stav dátového modelu.

V závere je vhodné zmieniť, že aplikačné riešenie je životaschopné a navrhnuté robustne, modulárne a so zámerom, že je možné ho v budúcnosti rozširovať bez nutnosti zložitých, rozsiahlych zásahov do primárnej riadiacej logiky.

Literatúra

[1] Foundation Level Syllabus. [Online; navštívené 04.04.2019].
URL
https://www.istqb.org/downloads/syllabi/foundation-level-syllabus.html

[2] The Netwide Assembler Documentation. [Online; navštívené 04.04.2019].
URL https://www.nasm.us/doc/

[3] Combined Volume Set of Intel® 64 and IA-32 Architectures Software Developer's Manuals. Feb 2019, [Online; navštívené 04.04.2019].
URL https://software.intel.com/en-us/articles/intel-sdm

[4] Allen, I. D.: The CARRY flag and OVERFLOW flag in binary arithmetic. [Online; navštívené 04.04.2019].
URL http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt

[5] Duntemann, J.: *Assembly language step by step: programming with Linux*. Wiley Publishing, 2009.

[6] Finley, T.: Two's Complement. Apr 2000, [Online; navštívené 04.04.2019].
URL https://www.cs.cornell.edu/~tomf/notes/cps104/twoscomp.html

[7] Irvine, K. R.: *Assembly language for x86 processors*. Pearson, 2015.

[8] Stelting, S.; Maassen, O.: *Applied Java patterns*. Prentice Hall, 2002.

Príloha A

Obsah priloženého pamäťového média

Priložené pamäťové médium obsahuje nasledujúce súbory:

- technická_správa_tex : adresár obsahujúci súbory pre preklad systémom LaTeX
- technická_správa_pdf : adresárobsahujúci .pdf súbor bakalárskej práce
- projekt_src : adresár obsahujúci zdrojové súbory pre aplikáciu
- projekt_jar : adresár obsahujúci .jar archív aplikácie
- testy : adresár obsahujúci dotazník, protokol, testovacie súbory, vyhodnotené dotazníky a manuálne testovacie zápisy
- readme.txt : súbor opisujúci štruktúru priložených dát a inštrukcie pre preklad
- manuál : adresár obsahujúci návod pre používanie aplikácie

Príloha B

Lexikálne a syntaktické pravidlá jazyka *Netwide Assembler*

Táto kapitola pojednáva o špecifických pravidlách odvodených z dokumentácie jazyka symbolických inštrukcií použitého v prostredí *Netwide Assembler*. Konkrétne ide o lexikálne pravidlá, ktoré umožňujú rozložiť vstupný reťazec na špecifické tokeny a syntaktické pravidlá organizácie tokenov do syntaktických celkov. Tieto pravidlá sú, v upravenej verzii, použité pre potreby implementácie simulátora.

B.1 Lexikálne pravidlá

Lexikálne pravidlá definujú postupnosti symbolov zložených z abecedy, ktoré v špecifickej sekvencii predstavuje slová, teda elementárne prvky nesúce význam. Takéto slová sú lexémy, alebo tokeny, teda najmenší výrazový prvok programovacieho jazyka. Vzhľadom k tomu, že abeceda je konečná množina, je možné opísať lexémy pomocou konečného stavového automatu, ktorý je použitý aj v rámci implementácie pre korektnú identifikáciu, či je vstupný reťazec platným symbolom. Aplikačné riešenie používa takzvaný *chamtivý* (po anglicky *greedy*) algoritmus, ktorý pohlcuje znaky a považuje ich za patriace lexému pokiaľ nenarazí na jednoznačný separátor. Následne vyhodnotí, či je lexém valídny.

B.1.1 Numerické hodnoty

Numerické hodnoty sú reťazce, ktoré svojou štruktúrou predstavujú konkrétnu informáciu vyjadrujúcu, v prípade tohto simulátora, celočíselnú hodnotu. Systém implementuje možnosť zadať takúto numerickú hodnotu vo viacerých formách, ktoré sa líšia bázou. Všetky numerické hodnoty môžu byť pre zjednodušenie čítania rozdelené symbolom „_“. Na základe dokumentácie jazyka *Netwide Assembler* sú to tieto formy (príklady demonštrujú zápis hodnoty ekvivalentnej dvesto (200) v dekadickej sústave):

Dekadické hodnoty

Teda hodnoty zapísané v desiatkovej sústave.

- `00200` : hodnota zadaná iba symbolmi číslic (teda z rozsahu 0-9) sú považované za dekadické hodnoty (aj v prípade, ak sekvencia začína nulou (0).

- `0200d` : hodnota zadaná symbolmi číslic (teda z rozsahu 0-9) a zakončená sufixom „d“ sú explicitne označené za dekadické hodnoty.
- `0d200` : hodnota zadaná symbolmi číslic (teda z rozsahu 0-9) a začínajúce prefixom „0d“ sú explicitne označené za dekadické hodnoty.

Oktalové hodnoty

Teda hodnoty zapísané v osmičkovej sústave.

- `0310o` : hodnota zadaná symbolmi číslic (teda z rozsahu 0-7) a zakončené sufixom „o“ sú explicitne označené za oktalové hodnoty.
- `0310q` : hodnota zadaná symbolmi číslic (teda z rozsahu 0-7) a zakončená sufixom „q“ sú explicitne označené za oktalové hodnoty.
- `0o310` : hodnota zadaná symbolmi číslic (teda z rozsahu 0-7) a začínajúce prefixom „0o“ sú explicitne označené za oktalové hodnoty.
- `0q310` : hodnota zadaná symbolmi číslic (teda z rozsahu 0-7) a začínajúce prefixom „0q“ sú explicitne označené za oktalové hodnoty.

Hexadecimálne hodnoty

Teda hodnoty zapísané v šestnástkovej sústave.

- `0c8h` : hodnota zadaná symbolmi číslic (teda z rozsahu 0-9), vybraných písmen (teda z rozsahu a-f) a sufixom „h“ sú považované za hexadecimálne hodnoty.
- `0xc8` : hodnota zadaná symbolmi číslic (teda z rozsahu 0-9), vybraných písmen (teda z rozsahu a-f) a prefixom „0x“ sú považované za hexadecimálne hodnoty.
- `0hc8` : hodnota zadaná symbolmi číslic (teda z rozsahu 0-9), vybraných písmen (teda z rozsahu a-f) a prefixom „0h“ sú považované za hexadecimálne hodnoty.

Binárne hodnoty

Teda hodnoty zapísané v dvojkovej sústave.

- `011001000b` : hodnota zadaná iba symbolmi číslic (teda z rozsahu 0-1) a zakončená sufixom „b“ sú explicitne označené za binárne hodnoty.
- `011001000y` : hodnota zadaná iba symbolmi číslic (teda z rozsahu 0-1) a zakončená sufixom „y“ sú explicitne označené za binárne hodnoty.
- `0b11001000` : hodnota zadaná iba symbolmi číslic (teda z rozsahu 0-1) a začínajúce prefixom „0b“ sú explicitne označené za binárne hodnoty.
- `0y11001000` : hodnota zadaná iba symbolmi číslic (teda z rozsahu 0-1) a začínajúce prefixom „0y“ sú explicitne označené za binárne hodnoty.

B.1.2 Symboly a návestia

Jazyk *Netwide Assembler* definuje symbol (ktorý môže reprezentovať návestie v kóde, alebo symbol asociovaný s adresou v pamäti) ako sekvenciu znakov z povolenej podmnožiny abecedy, ktorý začína dodatočne definovanou podmnožinou počiatočných symbolov. Návestie môže začať týmito symbolmi:

`[a-z], [A-Z], „.", „_" a „?"`

Symbol „.“ má špecifický význam z hľadiska jazyka, nakoľko tento definuje, že symbol je *lokálny*, teda jeho existencia je zviazaný s iným, nadradeným symbolom, ktorý nezačína znakom „.“. Týmto umožňuje redeklarovať takýto symbol, ak je unikátnym v kontexte nadradeného *globálneho* symbolu.

Nasledujúce znaky sú povolené ako znaky nasledujúce po štartovacom znaku symbolu a to:

`[a-z], [A-Z], [0-9], „.", „_", „$", „#", „@", „~" a „?"`

Sekvencia vygenerovaná touto postupnosťou je považovaná za platný symbol. *Netwide Assembler* uznáva akúkoľvek z týchto sekvencií za platný symbol, avšak implementácia aplikačného riešenia simulátora kladie striktnejšie pravidlá pre identifikáciu symbolu. Konkrétne ide o to, že symbol nemôže byť platným identifikátorom inštrukcie, veľkosti, alebo registra a symbol musí byť zakončený znakom dvojbodky „:“.

B.1.3 Kontrolné symboly

Ako kontrolné symboly sú v aplikačnom riešení chápané singulárne znaky. Pod tieto znaky spadajú:

- *Aritmetické operátory* : „`+`“, „`-`“, „`*`“, „`/`“, „`(`“ a „`)`“
- *Riadiace symboly* : „`,`“, „`:`“ a „`\n`“
- *Vymedzovače* : „`"`“, „`'`“ a vymedzenie pamäťového prístupu „`[`“ a „`]`“

B.1.4 Rezervované identifikátory

Rezervované identifikátory sú špecifické sekvencie znakov, ktoré sú definované v rámci prostriedkov implementovaného aplikačného riešenia. Tieto sekvencie sú dodatočne identifikované pomocou porovnávania so slovníkmi, ktoré interpretačný modul dodáva zo svojich vnútorných systémov. Konkrétne ide o identifikátory dostupných inštrukcií (vymenované v nasledujúcej kapitole), registrov (dostupné z inštancie strojového súboru ako kolekcia všetkých definovaných kľúčov viažúcich sa ku konkrétnym inštanciám objektov reprezentujúcich registre) a definície veľkostí, odvodené z reálnej implementácie jazyka *Netwide Assembler* a to `BYTE`, `WORD` a `DWORD`.

B.2 Syntaktické pravidlá

Vzhľadom k tomu, že aplikačné riešenie poskytuje len dve preddefinované sekcie (či segmenty) pre písanie kódu a neumožňuje používanie direktív, syntax vstupného jazyka je tak

extrémne zjednodušená. Ako bolo spomínané v primárnom texte tejto práce, elementárna jednotka takéhoto programovacieho jazyka je výraz, teda špecifická postupnosť neterminálov. Tieto neterminály sa môžu ďalej rozvinúť do terminálov, teda špecifických výrazov akceptovaných syntaktickými pravidlami.

Základná štruktúra takéhoto výrazu je:

```
návestie inštrukčný_výraz \n
```

V tejto základnej štruktúre sa tu nachádzajú dve základné jednotky (neterminály) zakončené terminálom konca výrazu (v tomto prípade znakom konca riadku). Oba neterminály je možné redukovať na prázdny výraz, teda platným výrazom je aj prázdny riadok.

Návestie je terminálom, ktorý je platným symbolom a terminálom dvojbodky „:“, aby mohol byť považovaný za syntakticky správny.

Inštrukčný_výraz je neterminálom, ktorý sa ďalej rozvíja na terminál jednoznačného identifikátora inštrukcie a neterminálu sekvencie operandov. Tieto sa opäť môžu redukovať do prázdneho výrazu (príkladom je napríklad inštrukcia `RET`), alebo sekvencie operandov oddelených kontrolným terminálom „`,`“.

Tieto operandy sú reprezentované terminálmi rezervovaných identifikátorov (registre), symbolov, algebraických výrazov, alebo prístupov do pamäte. Operand taktiež disponuje prefixom špecifikácie veľkosti, ktorý môže byť redukovaný do prázdneho výrazu.

Príloha C

Implementované inštrukcie

Táto príloha spracováva simulátorom implementované inštrukcie, rozdeľuje ich do celkov a popisuje ich použitie.

C.1 Dátové inštrukcie

Táto sekcia sa zaoberá inštrukciami, ktoré sa dajú definovať ako inštrukcie manipulácie s dátami formou dátového prenosu z jedného miesta systému do druhého.

MOV

Inštrukcia dátového prenosu, teda presunu hodnôt zo zdrojového operandu, do cieľového operandu.

```
MOV DST , SRC
```

Kde:

DST je cieľovým operandom, ktorý môže byť registrom, alebo prístupom do pamäte.

SRC je zdrojovým operandom, ktorý môže byť registrom, prístupom do pamäte, alebo priamou hodnotou (vyjadrenou numericky, či ako symbol).

Operácia podlieha kritériu rovnakej veľkosti operandov a neovplyvňuje príznaky systému.

XCHG

Inštrukcia výmeny dát medzi operandmi, teda presunu hodnôt zo zdrojového operandu, do cieľového operandu a z cieľového do zdrojového.

```
XCHG DST , SRC
```

Kde:

DST je cieľovým operandom, ktorý môže byť registrom, alebo prístupom do pamäte.

SRC je cieľovým operandom, ktorý môže byť registrom, alebo prístupom do pamäte.

Operácia podlieha kritériu rovnakej veľkosti operandov a neovplyvňuje príznaky systému.

C.2 Zásobníkové inštrukcie

Táto sekcia sa zaoberá inštrukciami, ktoré sa dajú definovať ako inštrukcie manipulácie s dátami formou zápisu a načítania zo štruktúry zásobníku poskytovaného prostriedkami modelovaného systému.

PUSH

Inštrukcia vloženia zdrojového operandu do zásobníka pomocou adresy uloženej v registry `ESP`.

```
PUSH SRC
```

Kde:

SRC je zdrojovým operandom, ktorý môže byť registrom, prístupom do pamäte, alebo priamou hodnotou (vyjadrenou numericky, či ako symbol).

Systém (v tomto prípade architektúra *Intel 32*) podporuje tieto inštrukcie v prenose šestnásť (16) a tridsaťdva (32) bitov, avšak simulátor implementuje len tridsaťdva (32) bitový variant. Inštrukcia priamo nemení hodnoty príznakov.

POP

Inštrukcia získania operandu zo zásobníka pomocou adresy uloženej v registry `ESP` a uloženia do cieľu.

```
POP DST
```

Kde:

DST je cieľovým operandom, ktorý môže byť registrom, alebo prístupom do pamäte.

Systém (v tomto prípade architektúra *Intel 32*) podporuje tieto inštrukcie v prenose šestnásť (16) a tridsaťdva (32) bitov, avšak simulátor implementuje len tridsaťdva (32) bitový variant. Inštrukcia priamo nemení hodnoty príznakov.

C.3 Aritmetické inštrukcie

Táto sekcia sa zaoberá inštrukciami, ktoré sa dajú definovať ako inštrukcie vykonávajúce aritmetické transformácie nad dátami.

ADD

Inštrukcia celočíselného súčtu, ktorá sčíta cieľový a zdrojový operand a výsledok uloží do cieľového operandu.

```
ADD DST, SRC
```

Kde:

DST je cieľovým operandom, ktorý môže byť registrom, alebo prístupom do pamäte.

SRC je zdrojovým operandom, ktorý môže byť registrom, prístupom do pamäte, alebo priamou hodnotou (vyjadrenou numericky, či ako symbol).

Inštrukcia podľa výsledku nastavuje príznaky: `OF`, `SF`, `ZF`, `CF` a `PF`.

SUB

Inštrukcia celočíselného rozdielu, ktorá od cieľového operandu odčíta zdrojový operand a výsledok uloží do cieľového operandu.

```
SUB DST, SRC
```

Kde:

DST je cieľovým operandom, ktorý môže byť registrom, alebo prístupom do pamäte.

SRC je zdrojovým operandom, ktorý môže byť registrom, prístupom do pamäte, alebo priamou hodnotou (vyjadrenou numericky, či ako symbol).

Inštrukcia podľa výsledku nastavuje príznaky: `OF`, `SF`, `ZF`, `CF` a `PF`.

MUL

Inštrukcia celočíselného násobenia bez znamienka. Operandom je násobený obsah registra `EAX`, alebo `AX`, alebo `AL`, pričom je výsledok uložený analogicky do `EDX:EAX`, `DX:AX` alebo `AX` na základe veľkosti vstupného operandu (jeden (4), dva (2) alebo jeden (1) bajt).

```
MUL SRC
```

Kde:

SRC je zdrojovým operandom, ktorý môže byť registrom, prístupom do pamäte, alebo priamou hodnotou (vyjadrenou numericky, či ako symbol).

Inštrukcia podľa výsledku nastavuje príznaky: `OF` a `CF` na 0, ak je horná časť výsledku 0, inak 1. `ZF`, `CF` a `PF` podľa výsledku.

IMUL

Inštrukcia celočíselného násobenia so znamienkom. Operandom je násobený obsah registra `EAX`, alebo `AX`, alebo `AL`, pričom je výsledok uložený analogicky do `EDX:EAX`, `DX:AX` alebo `AX` na základe veľkosti vstupného operandu (jeden (4), dva (2) alebo jeden (1) bajt). Táto inštrukcia umožňuje aj využitie varianty s dvomi, alebo tromi operandmi, no simulátor implementuje len verziu s jedným operandom.

```
IMUL SRC
```

Kde:

SRC je zdrojovým operandom, ktorý môže byť registrom, prístupom do pamäte, alebo priamou hodnotou (vyjadrenou numericky, či ako symbol).

Inštrukcia podľa výsledku nastavuje príznaky: `OF` a `CF` na 0, ak nie sú bity nesúce reálnu hodnotu väčšie ako dolný rozsah (to znamená, že výsledok je možné uložiť v dolnej polovici vrátane znamienkového bitu), inak 1. `ZF`, `CF` a `PF` podľa výsledku.

DIV

Inštrukcia celočíselného delenia bez znamienka. Operandom je delený obsah registra `EDX:EAX`, alebo `DX:AX`, alebo `AX`, pričom je výsledok uložený analogicky do `EAX` (zvyšok po delení do `EDX`), `AX` (zvyšok po delení do `DX`) alebo `AL` (zvyšok po delení do `AH`) na základe veľkosti vstupného operandu (jeden (4), dva (2) alebo jeden (1) bajt). Táto inštrukcia v sebe integruje ochranu pred delením nulou a ochranu pred pretečením cieľových registrov (čo vyvolá výnimku). Nie celočíselné výsledky sú zaokrúhlené smerom dole (k nule).

```
DIV SRC
```

Kde:

SRC je zdrojovým operandom, ktorý môže byť registrom, prístupom do pamäte, alebo priamou hodnotou (vyjadrenou numericky, či ako symbol).

Inštrukcia neovplyvňuje príznaky.

IDIV

Inštrukcia celočíselného delenia so znamienkom. Operandom je delený obsah registra `EDX:EAX`, alebo `DX:AX`, alebo `AX`, pričom je výsledok uložený analogicky do `EAX` (zvyšok po delení do `EDX`), `AX` (zvyšok po delení do `DX`) alebo `AL` (zvyšok po delení do `AH`) na základe veľkosti vstupného operandu (jeden (4), dva (2) alebo jeden (1) bajt). Táto inštrukcia v sebe integruje ochranu pred delením nulou a ochranu pred pretečením cieľových registrov (čo vyvolá výnimku). Nie celočíselné výsledky sú zaokrúhlené smerom dole (k nule).

```
DIV SRC
```

Kde:

SRC je zdrojovým operandom, ktorý môže byť registrom, prístupom do pamäte, alebo priamou hodnotou (vyjadrenou numericky, či ako symbol).

Inštrukcia neovplyvňuje príznaky.

INC

Inštrukcia celočíselného inkrementu. Teda zvýšením zdroja o jedna a uložením výsledku do zdroja.

```
INC SRC
```

Kde:

SRC je zdrojovým operandom, ktorý môže byť registrom, alebo prístupom do pamäte.

Inštrukcia podľa výsledku nastavuje príznaky: `OF`, `SF`, `ZF` a `PF`. Príznak `CF` nie je ovplyvnený.

DEC

Inštrukcia celočíselného dekrementu. teda zníženie zdroja o jedna a uložením výsledku do zdroja.

`DEC SRC`

Kde:

SRC je cieľovým operandom, ktorý môže byť registrom, alebo prístupom do pamäte.

Inštrukcia podľa výsledku nastavuje príznaky: `OF`, `SF`, `ZF` a `PF`. Príznak `CF` nie je ovplyvnený.

NEG

Inštrukcia celočíselnej negácie hodnoty podľa kódovania doplnku dvoch. Zdrojová hodnota je transformovaná na svoj negatívny ekvivalent a uložená do zdroja.

`NEG SRC`

Kde:

SRC je cieľovým operandom, ktorý môže byť registrom, alebo prístupom do pamäte.

Inštrukcia podľa výsledku nastavuje príznaky: `OF`, `SF`, `ZF` a `PF`. Príznak `CF` je nastavený podľa výsledku, kde ak zdrojový operand je 0, príznak 0 a naopak.

C.4 Posuvné a rotujúce inštrukcie

Táto sekcia sa zaoberá inštrukciami, ktoré sa dajú definovať ako inštrukcie vykonávajúce logické a aritmetické posuny a rotácie nad cieľovými hodnotami.

SHR

Inštrukcia logického bitového posunu binárneho vektora zdrojovej hodnoty doprava (teda nekopíruje hodnotu najviac významného bitu, ale dosádza vždy nulu (0)).

`SHR DST, CNT (nepovinný)`

Kde:

DST je cieľovým operandom, ktorý môže byť registrom, alebo prístupom do pamäte.

CNT je nepovinným operandom, ktorý môže byť registrom `CL` alebo priamou hodnotou. Ak operand nie je prítomný, je posun vykonaný iba o jeden (1) bit.

Inštrukcia podľa výsledku nastavuje príznaky: `CF` obsahuje hodnotu posledného posunutého bitu. V prípade ak je posun iba o jeden (1) bit, `OF` obsahuje najviac významný bit originálnej hodnoty pred transformáciou.

SHL

Inštrukcia logického bitového posunu binárneho vektora zdrojovej hodnoty doľava, ako nový bit je vždy dosadená nula (0).

```
SHL DST, CNT (nepovinný)
```

Kde:

DST je cieľovým operandom, ktorý môže byť registrom, alebo prístupom do pamäte.

CNT je nepovinným operandom, ktorý môže byť registrom `CL` alebo priamou hodnotou. Ak operand nie je prítomný, je posun vykonaný iba o jeden (1) bit.

Inštrukcia podľa výsledku nastavuje príznaky: `CF` obsahuje hodnotu posledného posunutého bitu. V prípade ak je posun iba o jeden (1) bit, `OF` obsahuje exkluzívny logický súčet najviac významného bitu výsledku a hodnoty `CF`.

SAR

Inštrukcia aritmetického bitového posunu binárneho vektora zdrojovej hodnoty doprava. Hodnota najviac významného bitu je kopírovaná.

```
SAR DST, CNT (nepovinný)
```

Kde:

DST je cieľovým operandom, ktorý môže byť registrom, alebo prístupom do pamäte.

CNT je nepovinným operandom, ktorý môže byť registrom `CL` alebo priamou hodnotou. Ak operand nie je prítomný, je posun vykonaný iba o jeden (1) bit.

Inštrukcia podľa výsledku nastavuje príznaky: `CF` obsahuje hodnotu posledného posunutého bitu. V prípade ak je posun iba o jeden (1) bit, `OF` je nastavený na 0.

SAL

Inštrukcia aritmetického bitového posunu binárneho vektora zdrojovej hodnoty doľava. Táto inštrukcia je zhodná v chovaní s inštrukciou `SHL`.

```
SAL DST, CNT (nepovinný)
```

Kde:

DST je cieľovým operandom, ktorý môže byť registrom, alebo prístupom do pamäte.

CNT je nepovinným operandom, ktorý môže byť registrom `CL` alebo priamou hodnotou. Ak operand nie je prítomný, je posun vykonaný iba o jeden (1) bit.

Inštrukcia podľa výsledku nastavuje príznaky: `CF` obsahuje hodnotu posledného posunutého bitu. V prípade ak je posun iba o jeden (1) bit, `OF` obsahuje exkluzívny logický súčet najviac významného bitu výsledku a hodnoty `CF`.

ROR

Inštrukcia rotácie binárneho vektoru doprava. Najmenej významný bit je zachovaný a po posune doprava dosadený ako najviac významný bit.

`ROR DST, CNT (nepovinný)`

Kde:

DST je cieľovým operandom, ktorý môže byť registrom, alebo prístupom do pamäte.

CNT je nepovinným operandom, ktorý môže byť registrom `CL` alebo priamou hodnotou. Ak operand nie je prítomný, je posun vykonaný iba o jeden (1) bit.

Inštrukcia podľa výsledku nastavuje príznaky: `CF` obsahuje hodnotu posledného posunutého bitu. V prípade ak je posun iba o jeden (1) bit, `OF` obsahuje exkluzívny logický súčet najviac významného bitu výsledku a druhého najviac významného bitu výsledku.

ROL

Inštrukcia rotácie binárneho vektoru doľava. Najviac významný bit je zachovaný a po posune doľava dosadený ako najmenej významný bit.

`ROL DST, CNT (nepovinný)`

Kde:

DST je cieľovým operandom, ktorý môže byť registrom, alebo prístupom do pamäte.

CNT je nepovinným operandom, ktorý môže byť registrom `CL` alebo priamou hodnotou. Ak operand nie je prítomný, je posun vykonaný iba o jeden (1) bit.

Inštrukcia podľa výsledku nastavuje príznaky: `CF` obsahuje hodnotu posledného posunutého bitu. V prípade ak je posun iba o jeden (1) bit, `OF` obsahuje exkluzívny logický súčet najviac významného bitu výsledku a `CF`.

RCR

Inštrukcia rotácie binárneho vektoru doprava cez príznak prenosu (`CF`). Najmenej významný bit je uložený do `CF` a pôvodná hodnota `CF` zachovaná, aby bola po posune doprava dosadená ako najviac významný bit.

`RCR DST, CNT (nepovinný)`

Kde:

DST je cieľovým operandom, ktorý môže byť registrom, alebo prístupom do pamäte.

CNT je nepovinným operandom, ktorý môže byť registrom `CL` alebo priamou hodnotou. Ak operand nie je prítomný, je posun vykonaný iba o jeden (1) bit.

Inštrukcia podľa výsledku nastavuje príznaky: `CF` obsahuje hodnotu posledného posunutého bitu. V prípade ak je posun iba o jeden (1) bit, `OF` obsahuje exkluzívny logický súčet najviac významného bitu pôvodnej hodnoty a `CF`.

RCL

Inštrukcia rotácie binárneho vektoru doľava cez príznak prenosu (`CF`). Najviac významný bit je uložený do `CF` a pôvodná hodnota `CF` zachovaná, aby bola po posune doľava dosadená ako najmenej významný bit.

```
RCL DST, CNT (nepovinný)
```

Kde:

DST je cieľovým operandom, ktorý môže byť registrom, alebo prístupom do pamäte.

CNT je nepovinným operandom, ktorý môže byť registrom `CL` alebo priamou hodnotou. Ak operand nie je prítomný, je posun vykonaný iba o jeden (1) bit.

Inštrukcia podľa výsledku nastavuje príznaky: `CF` obsahuje hodnotu posledného posunutého bitu. V prípade ak je posun iba o jeden (1) bit, `OF` obsahuje exkluzívny logický súčet najviac významného bitu výsledku a `CF`.

C.5 Logické inštrukcie

Táto sekcia sa zaoberá inštrukciami, ktoré sa dajú definovať ako inštrukcie vykonávajúce logické transformácie nad dátami.

AND

Inštrukcia logického súčinu, vykoná logický súčin hodnôt zdroja a cieľa reprezentovaných ako binárne vektory a výsledok tejto operácie je uložený do cieľového operandu.

```
AND DST, SRC
```

Kde:

DST je cieľovým operandom, ktorý môže byť registrom, alebo prístupom do pamäte.

SRC je zdrojovým operandom, ktorý môže byť registrom, prístupom do pamäte, alebo priamou hodnotou (vyjadrenou numericky, či ako symbol).

Inštrukcia podľa výsledku nastavuje príznaky: `CF` a `OF` sú nastavené na 0. `ZF`, `SF` a `PF` sú nastavené podla výsledku.

OR

Inštrukcia logického súčtu, vykoná logický súčet hodnôt zdroja a cieľa reprezentovaných ako binárne vektory a výsledok tejto operácie je uložený do cieľového operandu.

```
OR DST, SRC
```

Kde:

DST je cieľovým operandom, ktorý môže byť registrom, alebo prístupom do pamäte.

SRC je zdrojovým operandom, ktorý môže byť registrom, prístupom do pamäte, alebo priamou hodnotou (vyjadrenou numericky, či ako symbol).

Inštrukcia podľa výsledku nastavuje príznaky: `CF` a `OF` sú nastavené na 0. `ZF`, `SF` a `PF` sú nastavené podla výsledku.

XOR

Inštrukcia exkluzívneho logického súčtu, vykoná logický súčet hodnôt zdroja a cieľa reprezentovaných ako binárne vektory a výsledok tejto operácie je uložený do cieľového operandu.

```
XOR DST, SRC
```

Kde:

DST je cieľovým operandom, ktorý môže byť registrom, alebo prístupom do pamäte.

SRC je zdrojovým operandom, ktorý môže byť registrom, prístupom do pamäte, alebo priamou hodnotou (vyjadrenou numericky, či ako symbol).

Inštrukcia podľa výsledku nastavuje príznaky: `CF` a `OF` sú nastavené na 0. `ZF`, `SF` a `PF` sú nastavené podla výsledku.

NOT

Inštrukcia logickej negácie. Hodnota zdrojového operandu reprezentovaná binárnym vektorom je binárne negovaná a výsledok je uložený do zdrojového operandu.

```
NOT SRC
```

Kde:

SRC je cieľovým operandom, ktorý môže byť registrom, alebo prístupom do pamäte.

Inštrukcia neovplyvňuje žiadne príznaky.

C.6 Porovnávacie inštrukcie

Táto sekcia sa zaoberá inštrukciami, ktoré sa dajú definovať ako inštrukcie vykonávajúce logické a aritmetické porovnávania nad dátami a nastavujú príznaky systému.

CMP

Inštrukcia celočíselného porovnania, ktorá od cieľového operandu odčíta zdrojový operand a výsledok po vyhodnotení príznakov zahodí. Je analogická s inštrukciou `SUB`.

```
CMP DST, SRC
```

Kde:

DST je cieľovým operandom, ktorý môže byť registrom, alebo prístupom do pamäte.

SRC je zdrojovým operandom, ktorý môže byť registrom, prístupom do pamäte, alebo priamou hodnotou (vyjadrenou numericky, či ako symbol).

Inštrukcia podľa výsledku nastavuje príznaky: `OF`, `SF`, `ZF`, `CF` a `PF`.

TEST

Inštrukcia logického súčinu, vykoná logický súčin hodnôt zdroja a cieľa reprezentovaných ako binárne vektory a výsledok tejto operácie po vyhodnotený príznakov zahodí. Je analogická s inštrukciou `AND`.

```
TEST DST, SRC
```

Kde:

DST je cieľovým operandom, ktorý môže byť registrom, alebo prístupom do pamäte.

SRC je zdrojovým operandom, ktorý môže byť registrom, prístupom do pamäte, alebo priamou hodnotou (vyjadrenou numericky, či ako symbol).

Inštrukcia podľa výsledku nastavuje príznaky: `CF` a `OF` sú nastavené na 0. `ZF`, `SF` a `PF` sú nastavené podla výsledku.

C.7 Riadiace a skokové inštrukcie

Táto sekcia sa zaoberá inštrukciami, ktoré sa dajú definovať ako inštrukcie vykonávajúce riadenie chovania systému na zákalde dát poskytnutých stavom systému a hodnotami uložených v jeho prostriedkoch.

CALL

Inštrukcia pre vyvolanie procedúry. Aktuálna hodnota ukazujúca na nasledujúcu inštrukciu, ktorá sa má vykonať je uložená do zásobníka a systém vykoná skok (teda nastaví ukazovateľ na nasledujúcu inštrukciu) na adresu zadanú operandom. Implementácia inštrukcie v simulátore podporuje zadanie adresy začiatku procedúry len platným návestím.

```
CALL SRC
```

Kde:

SRC je cieľovým operandom, ktorý môže byť registrom, prístupom do pamäte,alebo priamou hodnotou (vyjadrenou numericky, či ako symbol), ktorá je platnou adresou v systéme.

Inštrukcia neovplyvňuje žiadne príznaky.

RET

Inštrukcia pre návrat z vyvolanej procedúry. Aktuálna hodnota ukazujúca na nasledujúcu inštrukciu, ktorá sa má vykonať je prepísaná hodnotu z vrcholu zásobníka a systém vykoná skok (teda nastaví ukazovateľ na nasledujúcu inštrukciu) na adresu obsiahnutú v tomto ukazovateli.

```
RET
```

Inštrukcia neovplyvňuje žiadne príznaky.

LOOP

Inštrukcia pre podmienený skok na operandom zadané návestie. Inštrukcia zníži obsah registra `ECX` o jedna a vyhodnotí jeho obsah. Ak sa obsah nerovná nule (0), inštrukcia vykoná skok na adresu zadanú zdrojovým operandom. Ak sa obsah registra rovná nule (0), inštrukcia nevykoná skok a vykonávanie pokračuje nasledujúcou inštrukciou. Implementácia inštrukcie v simulátore podporuje zadanie cieľovej adresy len platným návestím.

```
LOOP SRC
```

Kde:

SRC je cieľovým operandom, ktorý môže byť registrom, prístupom do pamäte,alebo priamou hodnotou (vyjadrenou numericky, či ako symbol), ktorá je platnou adresou v systéme.

Inštrukcia neovplyvňuje žiadne príznaky.

JMP

Inštrukcia nepodmieneného skoku. Inštrukcia zmení hodnotu inštrukčného ukazovateľa na adresu obsiahnutú v zdrojovom operande. Implementácia inštrukcie v simulátore podporuje zadanie cieľovej adresy len platným návestím.

```
JMP SRC
```

Kde:

SRC je cieľovým operandom, ktorý môže byť registrom, prístupom do pamäte,alebo priamou hodnotou (vyjadrenou numericky, či ako symbol), ktorá je platnou adresou v systéme.

Inštrukcia neovplyvňuje žiadne príznaky.

C.7.1 Rodina inštrukcií `Jcc`

Táto pod-sekcia sa zaoberá rodinou skokových inštrukcií, ktorých vykonanie je podmienené špecifickou podmienkou. Ich chovanie je však analogické s nepodmieneným skokom `JMP` a ich vykonávanie je až na overenie platnosti podmienky pre skok rovnaké. Ako pri `JMP`, aj tu implementácia považuje za legálny zdrojový operand pre skok platné návestie.

```
Jcc SRC
```

Kde:

SRC je cieľovým operandom, ktorý môže byť registrom, prístupom do pamäte,alebo priamou hodnotou (vyjadrenou numericky, či ako symbol), ktorá je platnou adresou v systéme.

Inštrukcie skoku, ktoré sú podmienené aritmetickým porovnaním očakávajú, že pred ich použitím je použité inštrukcie `CMP`, ktorá svojím chovaním nastaví adekvátne príznaky v systéme. Týmito príznakmi je potom riadené vyhodnotenie, preto aj opis skokových inštrukcií obsahuje vyjadrenia, ktoré dávajú zmysel pri aritmetickom porovnaní. Výnimkou sú inštrukcie kontrolujúce obsahy registrov, alebo priamo hodnoty príznakov.

Skok podľa aritmetického porovnania bez ohľadu na znamienko

`JE` Skok, ak sú hodnoty rovné, teda príznak nuly `ZF` je 1.

`JNE` Skok, ak hodnoty nie sú rovné, teda príznak nuly `ZF` je 0.

Skok podľa aritmetického porovnania bez znamienka

Tieto skokové inštrukcie pracujú s predpokladom, že porovnávané hodnoty sú chápané zo sémantického hľadiska, ako priame hodnoty.

`JA` Skok, ak je prvá z porovnávaných hodnôt väčšia. Vyhodnotené ak príznaky prenosu `CF` a nuly `ZF` sú rovné 0.

`JAE` Skok, ak je prvá z porovnávaných hodnôt väčšia, alebo rovná. Vyhodnotené ak príznak prenosu `CF` je 0.

`JB` Skok, ak je prvá z porovnávaných hodnôt menšia. Vyhodnotené ak príznak prenosu `CF` je 1.

`JBE` Skok, ak je prvá z porovnávaných hodnôt menšia, alebo rovná. Vyhodnotené ak príznaky prenosu `CF` a nuly `ZF` sú rovné 1.

`JNA` Skok, ak prvá z porovnávaných hodnôt nie je väčšia. Vyhodnotené ak príznaky prenosu `CF` a nuly `ZF` sú rovné 1.

`JNAE` Skok, ak prvá z porovnávaných hodnôt nie je väčšia, alebo nie je rovná. Vyhodnotené ak príznak prenosu `CF` je 1.

`JNB` Skok, ak prvá z porovnávaných hodnôt nie je menšia. Vyhodnotené ak príznak prenosu `CF` je 0.

`JNBE` Skok, ak prvá z porovnávaných hodnôt nie je menšia, alebo nie je rovná. Vyhodnotené ak príznaky prenosu `CF` a nuly `ZF` sú rovné 0.

Skok podľa aritmetického porovnania so znamienkom

Tieto skokové inštrukcie pracujú s predpokladom, že porovnávané hodnoty sú chápané zo sémantického hľadiska, ako hodnoty zakódované pomocou doplnku dvoch.

`JG` Skok, ak je prvá z porovnávaných hodnôt väčšia. Vyhodnotené ak príznaky nuly `ZF` a bitu znamienka `SF` sú rovné 0.

`JGE` Skok, ak je prvá z porovnávaných hodnôt väčšia, alebo rovná. Vyhodnotené ak príznaky nuly `ZF` a bitu znamienka `SF` sú vzájomne rovné.

`JL` Skok, ak je prvá z porovnávaných hodnôt menšia. Vyhodnotené ak príznaky bitu znamienka `SF` a pretečenia `OF` nie sú vzájomne rovné.

`JLE` Skok, ak je prvá z porovnávaných hodnôt menšia, alebo rovná. Vyhodnotené ak je príznak bitu znamienka `SF` rovný 1, alebo ak príznaky bitu znamienka `SF` a pretečenia `OF` nie sú vzájomne rovné.

JNG Skok, ak prvá z porovnávaných hodnôt nie je väčšia. Vyhodnotené ak je príznak nuly `ZF` rovný 1, alebo ak príznaky bitu znamienka `SF` a pretečenia `OF` nie sú vzájomne rovné.

JNGE Skok, ak prvá z porovnávaných hodnôt nie je väčšia, alebo nie je rovná. Vyhodnotené ak príznaky bitu znamienka `SF` a pretečenia `OF` nie sú vzájomne rovné.

JNL Skok, ak prvá z porovnávaných hodnôt nie je menšia. Vyhodnotené ak príznaky bitu znamienka `SF` a pretečenia `OF` sú vzájomne rovné.

JNLE Skok, ak prvá z porovnávaných hodnôt nie je menšia, alebo nie je rovná. Vyhodnotené ak je príznak nuly `ZF` rovný 0 a ak príznaky bitu znamienka `SF` a pretečenia `OF` sú vzájomne rovné.

Skok podľa hodnoty príznaku alebo registra

Tieto skokové inštrukcie operujú priamo s hodnotami príznakov, alebo špecifických registrov.

JC Skok, ak je príznak prenosu `CF` rovný 1.

JCXZ Skok, ak je obsah registra `CX` rovný 0.

JECXZ Skok, ak je obsah registra `ECX` rovný 0.

JNC Skok, ak je príznak prenosu `CF` rovný 0.

JNO Skok, ak je príznak pretečenia `OF` rovný 0.

JNP Skok, ak je príznak parity `PF` rovný 0.

JNS Skok, ak je príznak bitu znamienka `SF` rovný 0

JNZ Skok, ak je príznak nuly `ZF` rovný 0

JO Skok, ak je príznak pretečenia `OF` rovný 1.

JP Skok, ak je príznak parity `PF` rovný 1.

JPE Skok, ak je parita párna, teda príznak `PF` rovný 1.

JPO Skok, ak je parita nepárna, teda príznak `PF` rovný 0.

JS Skok, ak je príznak bitu znamienka `SF` rovný 1.

JZ Skok, ak je príznak nuly `ZF` rovný 1.

Príloha D

Implementované procedúry

Táto kapitola opisuje implementované procedúry, ktoré nie sú súčasťou štandardného prostredia Netwide Assembler, ale v tejto práci slúžia pre zjednodušenie práce so vstupno-výstupnými operáciami. Všetky procedúry sú dostupné pomocou volania inštrukciou `CALL`, akoby šlo o funkcie zahrnuté v samotnom programe.

D.1 Procedúry pre používateľský vstup

Tieto procedúry používajú grafické používateľské rozhranie pre načítanie hodnoty a jej interpretáciu ako dát akceptovateľných modelovaným systémom.

ReadChar

Procedúra načíta znak z používateľského rozhrania a interpretuje ho ako numerickú hodnotu, ktorá je uložená do registra `AL`.

ReadString

Procedúra na základe informácii z registrov `EBX` (maximálna veľkosť čítaného reťazca) a `EDI` (cieľová adresa, kam bude reťazec uložený) vykoná čítanie sekvencie znakov zadaných používateľom. V prípade ak je reťazec dlhší ako zadaná dĺžka, je skrátený (a ukončený teminátorom, teda nulovým bajtom).

ReadInt8, ReadInt16 a ReadInt32

Procedúra načíta zo štandardného vstupu numerickú hodnotu a interpretuje ju ako celé číslo so znamienkom, ktoré uloží do registra `AL`, `AX`, alebo `EAX` na základe toho, ktorá z procedúr bola volaná (osem (8), šestnásť (16), alebo tridsaťdva (32) bitová verzia).

ReadUInt8, ReadUInt16 a ReadUInt32

Procedúra načíta zo štandardného vstupu numerickú hodnotu a interpretuje ju ako celé číslo bez znamienka, ktoré uloží do registra `AL`, `AX`, alebo `EAX` na základe toho, ktorá z procedúr bola volaná (osem (8), šestnásť (16), alebo tridsaťdva (32) bitová verzia).

D.2 Procedúry pre výstup skalárnej hodnoty

Tieto procedúry slúžia k výpisu konkrétnej hodnoty obsiahnutej v systéme na štandardný výstup (teda do konzoly, ktorá je súčasťou používateľského rozhrania).

WriteChar

Procedúra z registra `AL` načíta osem (8) bitovú hodnotu a interpretuje ju ako textový symbol, ktorý vytlačí na konzolu používateľského rozhrania.

WriteNewLine

Procedúra vytlačí na konzolu používateľského rozhrania symbol konca riadku.

WriteString

Na základe informácií z registra `ESI`, systém začne po znakoch načítavať z pamäte bajty, ktoré sekvenčne vynáša na štandardný výstup. Procedúra očakáva terminátorom (teda nulovým bajtom), ukončenú sekvenciu hodnôt. V prípade, ak nie je reťazec ukončený terminátorom, čítanie prejde do ilegálnej pamäte a systém vystaví výnimku.

WriteBin8, WriteBin16 a WriteBin32

Procedúra interpretuje hodnotu uloženú v registroch `AL`, `AX`, alebo `EAX` (podľa toho, či bola volaná osem (8), šestnásť (16), alebo tridsaťdva (32) bitová verzia procedúry) ako reťazec obsahujúci binárnu reprezentáciu hodnoty a vytlačí ju na štandardný výstup.

WriteHex8, WriteHex16 a WriteHex32

Procedúra interpretuje hodnotu uloženú v registroch `AL`, `AX`, alebo `EAX` (podľa toho, či bola volaná osem (8), šestnásť (16), alebo tridsaťdva (32) bitová verzia procedúry) ako reťazec obsahujúci hexadecimálnu reprezentáciu hodnoty a vytlačí ju na štandardný výstup.

WriteInt8, WriteInt16 a WriteInt32

Procedúra interpretuje hodnotu uloženú v registroch `AL`, `AX`, alebo `EAX` (podľa toho, či bola volaná osem (8), šestnásť (16), alebo tridsaťdva (32) bitová verzia procedúry) ako reťazec obsahujúci dekadickú reprezentáciu hodnoty so znamienkom a vytlačí ju na štandardný výstup.

WriteUInt8, WriteUInt16 a WriteUInt32

Procedúra interpretuje hodnotu uloženú v registroch `AL`, `AX`, alebo `EAX` (podľa toho, či bola volaná osem (8), šestnásť (16), alebo tridsaťdva (32) bitová verzia procedúry) ako reťazec obsahujúci dekadickú reprezentáciu hodnoty bez znamienka a vytlačí ju na štandardný výstup.

D.3 Procedúry pre výstup vektorovej hodnoty

Tieto procedúry slúžia k výpisu vektorovej hodnoty obsiahnutej v systéme na štandardný výstup (teda do konzoly, ktorá je súčasťou používateľského rozhrania). Pod vektorovým

výstupom sa tu chápe pole hodnôt uložených v pamäti, ktoré sú dekódované a vystavené do výstupu. K tomuto slúžia procedúry označené ako `WriteArrayInt8`, `WriteArrayInt16` a `WriteArrayInt32`. Analogicky k týmto boli taktiež implementované procedúry `Write-ArrayUInt8`, `WriteArrayUInt16` a `WriteArrayUInt32`.

Procedúra na základe poskytnutých dát v registroch `ESI` (adresa prvého prvku poľa) a `ECX` (počet prvkov poľa) interpretuje prvky uložené v pamäti ako numerické hodnoty a vytlačí ich na štandardný výstup. Na základe názvu vo volaní, tieto hodnoty môžu byť poskytnuté v dekadickom formáte so znamienkom a dekadickom formáte bez znamienka.