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 ORSÁG, Ph.D.**
SUPERVISOR

BRNO 2018

Zadání bakalářské práce/21141/2017/xheste00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2017/2018

Zadání bakalářské práce

Řešitel: **Heštera Andrej**
Obor: Informační technologie
Téma: **Simulátor asembleru x86 pro výuku**
x86 Assembler Simulator for Education
Kategorie: Překladače

Pokyny:

1. Prostudujte následující témata: simulátory procesorů, asembler x86, zásady vizuální prezentace chování procesoru pro výukové účely.
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.
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 bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Orság Filip, Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 16. května 2018

L.S.

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

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 napodobuje chovanie procesora z architektúru 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.

Klíčová slova

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

Citace

HEŠTERA, Andrej. *Simulátor asembleru x86 pro výuku.* Brno, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Filip Orság, Ph.D.

Simulátor asembleru x86 pro výuku

Prohlášení

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
13. května 2018

Poděkování

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

Obsah

Kapitola 1

Úvod

Obsahom tejto práce je analýza prostriedkov architektúry x86, jazyka symbolických inštrukcií pre procesory rodiny x86 a prostredia Netwide Assembler. Na základe tejto interpretácie bude navrhnutý abstraktný model a neskôr implementovaný simulačný model, ktorý umožní používateľom vykonávať simulačné experimenty nad týmito modelmi.

Práca by mala čitateľa zoznámiť s kľúčovými prvkami architektúry x86 a princípmi, ktoré využíva k reprezentácii a práci s dátami a množinou inštrukcií podporovan7ch procesormi implementujúcimi túto architektúru.

V nasledujúcich kapitolách sa pokúsi dielo priblížiť realitu programovania v nízko úrovňových jazykoch a taktiež vytvoriť hodnovernú reprezentáciu, ktorá umožní používateľovi bez hlbšej vedomosti o tejto problematike navrhnúť, otestovať, preložiť a skúmať vykonávanie krátkych sekvencií kódu asembleru x86 pre účely prehĺbenia vedomostí a získania informačnej bázy, ktorá následne umožní ďalšie vzdelávanie.

Účelom nie je plne simulovať prostriedky procesorov rodiny x86, avšak umožniť rýchle a jednoduché testovanie podporovaných inštrukcií za účelom pochopenia ich singulárnej funkcionality a to ako ovplyvňujú stav systému, nad ktorým sú vykonávané.

Kapitola 2

Architektúra x86, Jazyk symbolických inštrukcií a Simulácia

Cieľom tejto práce je vytvoriť simulačný nástroj pre testovanie používateľom navrhnutých programov a prostredníctvom simulácie ich vykonávania sledovať meniaci sa stav na modely fyzického systému. Účelom tohto prostriedku je prehĺbenie znalostí o návrhu a vykonávaní programu. Pre vytvorenie validného modelu je nutné skúmať kľúčové prvky fyzického systému a pochopiť ich funkcionalitu pre korektné napodobenie ich chovania za účelom simulácie.

Z hľadiska tohto simulačného systému je nutné analyzovať prostriedky a chovanie poskytnuté architektúrou x86, spôsob, akým je používateľom definovaný kód interpretovaný na spustiteľný program a samotné vykonávanie programu týmto systémom, pričom je potrebné adresovať zmeny nad fyzickým systémom podmienené týmto vykonávaním.

Je dôležité zmieniť, že účelom tejto práce nie je hĺbková analýza architektúry x86 a prostredia písania, kontroly, kompilovania a spustenia kódu, ale analýza prostriedkov kľúčových pre získanie a pochopenie základných vedomostí o problematike tvorenia programov prostredníctvom asembleru x86. Z toho dôvodu bude aj táto práca analyzovať prostriedky hlavne dostupné v rámci používateľom definovaného kódu.

2.1 Architektúra x86

Architektúra x86, tiež aj Architektúra Inštrukčnej Sady procesorov rodiny x86 (anglicky Instruction Set Architecture) je abstraktný model opisujúci a definujúci chovanie systému a prostriedky dostupné pre programátora v rámci návrhu a spustenia programu. [10]

Tento model definuje dostupné vnútorné prostriedky, ako sú registre, pamäťový priestor, opisuje podporované dátové typy a špecifikuje množinu symbolických inštrukcií, ktoré opisujú operácie schopné vykonávať týmto modelom a dostupné kombinácie dát, nad ktorými je tieto operácie možné vykonávať. [7]

Realizácia tohto modelu ako fyzického systému sa označuje *implementácia*. Z hľadiska reálneho použitia sa jednotlivé implementácie môžu líšiť rýchlosťou, cenou a fyzickými charakteristikami, no z hľadiska dát, s ktorými systém pracuje je jeho stav pred a po vykonaní konkrétnej inštrukcie definovaný v rámci modelu. [1]

Pre potreby navrhované simulačného prostredia bude táto sekcia analyzovať prostriedky tak ako ich má k dispozícii programátor a bude zanedbávať implementačné detaily hardvérového riešenia.

Z hľadiska dátových typov architektúra podporuje celočíselné hodnoty v doplnkovom kóde a hodnoty s pohyblivou desatinnou čiarkou. Pre potreby simulačných prostriedkov bude analyzovaný len spôsob, akým systém pracuje s celočíselnými hodnotami.

2.1.1 Registre a Inštrukčný ukazovateľ

Register predstavuje základný element architektúry. Jedná sa o komponentu v systéme, ktoré umožňuje udržiavať dáta a vykonávať nad nimi aritmetické, logické a dátové operácie (dátové, teda operácie presunu dát medzi registrami navzájom, či registrami a pamäťou).

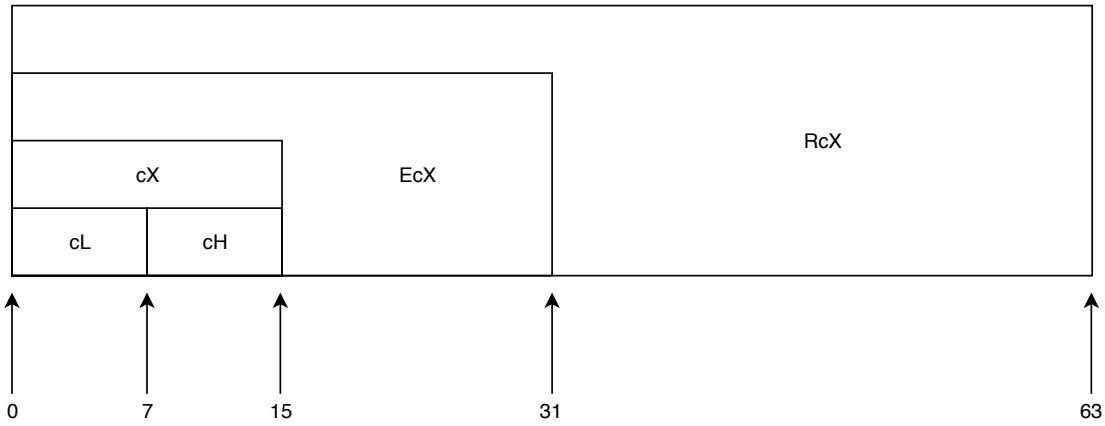
Architektúra rozlišuje niekoľko kategórii registrov: [1]

- Všeobecné registre
- Segmentové registre
- Príznakový register
- Inštrukčný ukazovateľ

Všeobecné registre

Všeobecné registre sú systémové súčasti, ktoré programátorovi umožňujú využívať ich obsah pre ukladanie dát a výsledkov operácií. Jednotlivé registre taktiež z hľadiska architektúry slúžia ako priestory pre ukladanie informácii o upravovaní chovaní procesora v prípade vyvolania špecifického prerušenia alebo obsahujú dáta ovplyvňujúce chovanie špecifickej inštrukcie (ako napríklad dáta riadiace prerušenie pre výpis dát prostredníctvom vstupno-výstupných jednotiek, alebo register kontrolujúci ECX kontrolujúci vykonávanie slučky prostredníctvom inštrukcie LOOP). [1]

Architektúra x86 poskytuje register ako komponentu schopnú uložiť do seba numerická dáta určitej bitovej šírky, pričom z hľadiska vývoja platformy (a rozširovania dostupnej bitovej šírky) došlo k prekrývaniu kľúčov asociovaných s určitou šírkou registra. Obrázok[2.1] ilustruje ako dochádza k tomuto prekrývaniu spolu s indexmi bitov.



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

Toto chovanie umožňuje pristupovať ku konkrétnym bitovým šírkam, kedy je niekoľko registrov, ktoré je možné chápať ako virtuálne pamäťové priestory, asociovaných s jedným zdielaným fyzickým priestorom.

Ako obrázok[2.1] ukazuje, jednotlivé prístupové kľúče (kde znak c predstavuje konkrétne značenie registra) nepredstavujú samostatné registre, ale zdieľaný pamäťový priestor, ktorého maximálna šírka vychádza z konkrétnej architektúry (na obrázku je najväčší zobrazený register šírky šesťdesiatštyri (64) bitov).

Architektúra x86 poskytuje osem všeobecných registrov, pričom každý z týchto registrov, plne prístupný potrebám programátora, má aj vlastnú špecifickú funkcionalitu vychádzajúcu z implementovanej architektúry. Tieto registre sú (vo zozname je použité pomenovanie pre šestnásť (16) bitový register):

`AX:` Akumulátor, tiež aj zberač. Register, ktorého účelom je využitie pre aritmetické operácie.

`CX:` Počítač, či počítadlo. Register používaný pre posuny, rotácie a ovládanie programových slučiek.

`DX:` Dátový register. Používaný pre aritmetické a vstupno-výstupné operácie.

`BX:` Bázový register. Používaný ako ukazovateľ na dáta (uložené v segmentovom registry DS, pre segmentové adresovanie).

`SP:` Ukazovateľ vrcholu zásobníka v pamäti.

`BP:` Ukazovateľ báze zásobníku v pamäti.

`SI:` Zdrojový ukazovateľ/index. Ukazuje na zdroj operácií dátového prenosu.

`DI:` Cieľový ukazovateľ/index. Ukazuje na cieľ operácií dátového prenosu.

Len registre zakončené postfixom X umožňujú prístup k hornej a dolnej polovici svojich šestnásť bitových reprezentácii. V nasledujúcich iteráciách architektúry boli prefixom E identifikované registre šírky tridsaťdva (32) bitov a prefixom R boli identifikované šírky šesťediatštyri (64) bitov.

Z hľadiska skúmania fyzického systému pre potreby návrhu simulačných prostriedkov budú použité len registre maximálnej šírky tridsaťdva (32) bitov a menšie. Z tohto dôvodu budú taktiež opomenuté registre FPU (anglicky *Floating-Point Unit*), ktoré sú používané v rámci rozšírenej aritmetickej funkcionality podporujúcej desatinné čísla.

Segmentové registre

Segmentové registre predstavujú pomôcku pri adresovaní užitočných dát, respektíve je ich obsah používaní pri kalkulácii reálnej/fyzickej adresy v rámci prístupu k dátam programu. Architektúra poskytuje šesť segmentových registrov [1]:

`SS:` Stack Segment. Ukazovateľ na zásobník.

`CS:` Code Segment. Ukazovateľ na inštrukcie strojového kódu.

`DS:` Data Segment. Ukazovateľ na dáta programu.

`ES:` Extra Segment. Ukazovateľ na extra dáta (kde E znamená Extra).

`FS:` F Segment. Ukazovateľ na extra dáta (kde F nasleduje po E).

`GS:` G Segment. Ukazovateľ na extra dáta (kde G nasleduje po F).

V súčasnosti je však používanie týchto registrov v moderných operačných systémoch nahradené systémov *stránkovania*[9], teda obsahy segmentových registrov ukazujú na rovnaké miesto, pričom je v réžii implementácie systému, aby do operačnej pamäti sťahoval dátové rámce zo sekundárnej pamäte.

Z tohto dôvodu je však rozšírené využívanie segmentových registrov **FS** a **GS**, ktorých obsah ukazuje na dátové priestory konkrétnej inštancie vlákna procesu.

Z hľadiska simulačného systému je modelovanie týchto registrov vynechané, keďže programátor má k dispozícii virtuálny pamäťový priestor a znalosti týkajúce sa používateľskej kontroly pamäťových prístupov je považované za informácie mimo cieľovej množiny znalostí, ktoré má simulátor pomôcť nadobudnúť.

Príznakový register

Príznakový register (tiež aj Stavový register, či EFLAG) predstavuje vnútornú komponentu realizovanú registrom (v architektúre x86 implementovaný ako register šírky tridsaťdva (32) bitov), ktorej účelom je monitorovať a poskytovať informácie o stave systému. Tento stav môže byť definovaný ako výsledok aktuálne vykonanej aritmetickej, či logickej operácie, alebo nastavenie príznaku čakania na prerušenie.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	0	0	0	0	0	0	0	0	ID	VIP	VIF	AC	VM	RF

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	NT	IOPL		OF	DF	IF	TF	SF	ZF	0	AF	0	PF	1	CF

Obrázek 2.2: Reprezentácia stavového registra a asociácie indexov s kľúčmi.

Mimo udržiavania stavu systému je tento register aj zdrojom pre riadiacu logiku vykonávania strojového kódu, ktorý umožňuje vetvenie na základe pravdivostnej hodnoty jednotlivých konkrétnych bitov. [10] Z hľadiska návrhu simulácie je táto komponenta kľúčová pre pochopenie procesu vykonávania strojového kódu.

Obrázok[2.2] ilustruje rozloženie kľúčov a asociovaných indexov, pričom hodnota bitu na konkrétnom indexe je chápaná ako pravdivostná hodnota reprezentujúca stav systému. Nepomenované indexy sú chápané z hľadiska architektúru ako rezervované a ich hodnota je nemenná, pričom zodpovedá hodnote obsiahnutej v bunke, ktorá inak obsahuje identifikačnú skratku príznaku.

Príznakové bity sa dajú rozdeliť do troch kategórií a to *Stavové* (CF, PF, AF, ZF, SF a OF), *Systémové* (IOPL, NT, RF, VM, AC, VIF, VIP a ID) a *Riadiace* (TF, IF a DF).

index 00, CF, Carry Flag: Príznak, ktorý je generovaný aritmetickou operáciou v prípade ak v rámci výpočtu nastane tzv. výpožička z/prenos do vyššieho binárneho rádu. Pre programátora tento príznak indikuje chybu aritmetiky znamienkových čísel. [3]

index 02, PF, Parity Flag: Príznak, ktorý indikuje ak binárna reprezentácia výsledku má párny počet bitov.

index 04, AF, Adjust/Auxiliary Flag: Príznak, ktorý indikuje prenos do vyššieho rádu v rámci operácii používajúcich BCD (Binary-Coded Decimal, teda binárne kódované dekadické číslo) hodnoty.

index 06, ZF, Zero Flag: Príznak, ktorý indikuje, či výsledok aritmetickej, alebo logickej operácie bol nula (0).

index 07, SF, Sign Flag: Príznak, ktorý indikuje, či je výsledok operácie záporné číslo.

index 08, TF, Trap Flag: Príznak, ktorý umožňuje procesoru vykonávanie inštrukcií po jednej a následne analyzovať obsah zásobníka a registrov. Využívané aplikáciami typu „debugger“.

index 09, IF, Interrupt Flag: Príznak, ktorý indikuje, či je procesor schopný maskovať prerušenia vyvolané hardvérom.

index 10, DF, Direction Flag: Príznak indikujúci smer čítania reťazca. Ak je nastavený, ukazovateľ čítania sa znižuje, v opačnom prípade, zvyšuje.

index 11, OF, Overflow Flag: Príznak indikujúci pretečenie hodnoty z operačného bitového rozsahu. Tento príznak je produktom aritmetických operácii v prípade, ak je výsledná hodnota širšia ako veľkosť definovanej operácie. Pre programátora tento príznak indikuje chybu aritmetiky nad neznamienkovými hodnotami.[3]

index 12-13, IOPL, I/O Privilege Level: Príznak indikujúci úroveň vstupno-výstupných práv programu/úlohy. Ak procesor pracuje v chránenom režime, tento príznak slúži k overeniu prístupových práv. V prípade, ak *CPL* (Current Privilege Level) hodnota aktuálne spusteného programu/úlohy je nižšia, alebo rovná ako *IOPL* hodnota príznakového registra, je mu umožnený prístup k vstupno-výstupným portom.

index 14, NT, Nested Task Flag: Príznak je nastavený, ak nasledujúca úloha bola vyvolaná predošlou prostredníctvom volania inštrukcie `CALL`, namiesto nepodmieneného skoku, čo vyvoláva zreťazenie.

index 16, RF, Resume Flag: Používaná v rámci debuggovacieho procesu.

index 17, VM, Virtual-8086 Mode: Príznak indikuje spustenie procesoru v móde 8086.

index 18, AC, Alignment Check: Príznak indikujúci dokončenie kontroly zarovnanie dát v pamäti.

index 19, VIF, Virtual Interrupt Flag: Príznak indikujúci virtuálne prerušenie.

index 20, VIP, Virtual Interrupt Pending Flag: Príznak indikuje virtuálne prerušenie čakajúce na obsluhu.

index 21, ID, Identification Flag: Príznak indikuje, či procesor implementuje inštrukciu `CPUID`.

Z hľadiska budovania simulačného modelu sú kľúčové stavové príznaky, ktoré indikujú výsledky operácii a ako také slúžia k riadeniu toku programu prostredníctvom podmienených skokov. Ostatné príznaky vyžadujú pre použitie hlbšie znalosti, ktoré sú mimo rozsahu tejto práce. Preto budú modelované príznaky **CF**, **PF**, **ZF**, **SF** a **OF**.

Inštrukčný ukazovateľ

Inštrukčný ukazovateľ je systémový register, ktorého hodnota je ukazovateľ do pamäte obsahujúcej nasledujúcu inštrukciu programu. V rámci definície chovania, je inštrukčný ukazovateľ zvýšený o jedna (1) pred vykonaním súčasnej inštrukcie. Toto umožní súčasnej inštrukcii, v prípade, ak je to inštrukcia riadiaca tok programu, zmeniť hodnotu ukazovateľa a tým vytvoriť riadenie programu prostredníctvom programovej logiky vetvenia.

2.1.2 Pamäť a Zásobník

Pamäťový priestor a v ňom realizovaný zásobník sú modely, ktorá umožňujú prostredníctvom systémovo integrovaných služieb pristupovať pomocou adresy k dátam uloženým v operačnej pamäti procesoru.

Ako bolo zmienené už v sekcii pojednávajúcej o segmentových registroch[2.1.1], súčasné operačné systémy nahradzujú priame pristupovanie do pamäti formou stránkovania, čo je abstrakcia fyzickej pamäte do podoby virtuálneho modelu pamäťového priestoru. *Virtuálna pamäť* reprezentuje idealizovanú abstrakciu pamäťových prostriedkov dostupných na danom stroji. [4]

Z hľadiska programátora je virtuálna pamäť riešením bezpečnosti a zefektívnenia prístupov do pamäte, pričom je toto implementované kombináciou hardvérových a softvérových riešení. Toho je docielené využívaním tzv. *virtuálnej adresy*, ktorá je použitá v stránkach vyvolaných zo sekundárnych pamäťových médii prostredníctvom stránkovacieho algoritmu. Ten zabezpečí preklad virtuálnej na *fyzickú adresu*, čo už je mimo réžie programátora.

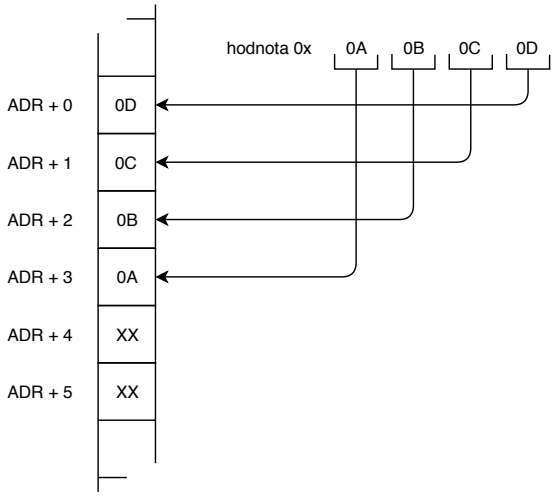
Pre potreby simulátora bude teda modelovaná virtuálna pamäť, ktorá umožňuje izoláciu od pamäťových sekcií ostatných programov/úloh a umožní tým pádom efektívnejšie zobrazenie prístupov do tohto pamäťového bloku. Z hľadiska uložených dát, architektúra x86 implementuje pamäťový model využívajúci konceptu *Little-Endian*, čo je organizácia ukladania viac bytových hodnôt v pamäti a ich následne načítanie. Z principiálneho hľadiska, Little-Endian obracia poradie ukladaných bytov, pričom ako prvý je uložený byt obsahujúci najmenej významné bity. [10]

Účelom tohto je zabezpečiť, aby načítanie tej istej adresy, s rozdielnymi požiadavkami na šírku dátového prístupu (chápaný ako šírka dát v bitoch) vytvorilo zhodné najnižšie bity. Táto organizácia je ilustrovaná na obrázku[2.3].

Zásobník

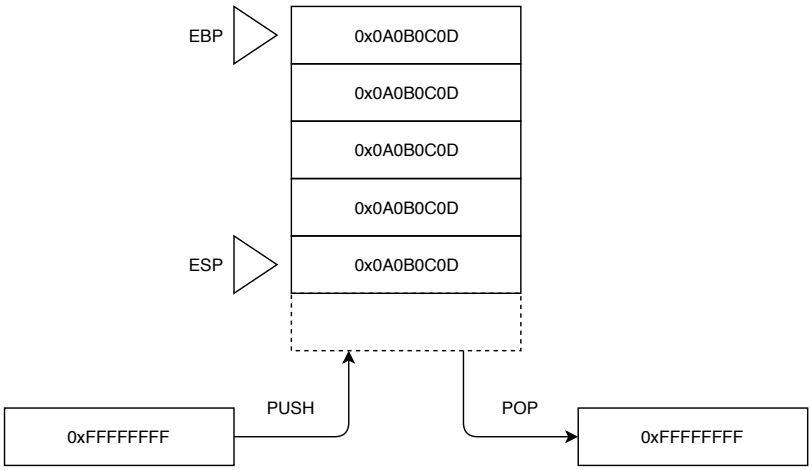
V rámci tejto pamäte je realizovaná aj dátová štruktúra zásobníka. Architektúra x86 definuje zásobník ako pamäťové pole zarovnané na architektúrou špecifikovanú veľkosť homogénnych prvkov, ktorých účelom je zefektívnenie prenosu hodnoty medzi jednotlivými sekciami kódu.

Súčasťou obsluhy zásobníka sú dva registre definované v rámci architektúry a to register udržiavajúci adresu najvyššieho prvku (v rámci architektúry x86 je toto register SP, či ESP)



Obrázek 2.3: Opis princípu ukladania dát do pamäte prostredníctvom metódy Little-Endian.

a register udržiavajúci adresu dna/bázy zásobníku (v rámci architektúry x86 je toto register BP, či EBP). [9]



Obrázek 2.4: Ilustrácia funkcie zásobníku na architektúre x86.

Model zásobníku umožňuje vkladanie a vyberanie hodnôt (ilustrované obrázkom[2.4]), pričom je toto zabezpečené algoritmom *LIFO* („Last In, First Out“ - Posledný Dnu, Prvý Von) a dvomi množinami definovaných inštrukcií. Tie sa dajú rozdeliť podľa charakteristiky dátovej operácie a to množina PUSH, či množina inštrukcií vkladajúcich dáta a množina POP, či množina inštrukcií vyberajúcich dáta zo zásobníka.

Množina inštrukcií typu *PUSH* umožňuje vložiť operand do zásobníka (pričom je, na základe architektúry požadované, aby tieto dáta mali dostačujúcu veľkosť, ktorá neporuší zarovnanie vkladaných informácii).

Mimo používateľom definovaných dát predložených inštrukcii PUSH ako operand, architektúra taktiež umožňuje vložiť na zásobník obsah všeobecných registrov (`PUSHA`, `PUSHAD`), či príznakového registra (`PUSHF`, `PUSHFD`). Táto funkcionalita opäť nadväzuje

na možnosť využitia zásobníka ako systému na riadenie chodu programu prostredníctvom vyvolávania podprogramov cez inštrukciu `CALL`.

Chovanie zásobníku a inštrukcie PUSH, počas jej vykonania je kombinácia úpravy adresy najvyššieho prvku a prístupu do pamäte prostredníctvom tejto adresy. Konkrétne odčítanie šírky bunky zásobníku v bytoch od hodnote registra SP/ESP a uloženie dát do zásobníku prostredníctvom pamäťového prístupu s využitím adresy v registry SP/ESP. Z toho je možné odvodiť, že:

```
PUSH EAX
```
je ekvivalentné:
```
SUB ESP, 4
MOV [ESP], EAX
```

Množina inštrukcií typu *POP* umožňuje odstrániť prvok zo zásobníka a uložiť ho do operandu poskytnutého programátorom. Je analogická k množine operácii typu PUSH. Poskytuje funkcionalitu, ktorá umožňuje načítať obsahy registrov z obrazu uloženého v zásobníku (`POPA`, `POPAD`), alebo načítať príznakový register (`POPF`, `POPFD`).

Chovanie zásobníka a inštrukcie POP, počas jej vykonania je kombinácia načítanie hodnozty z pamäte do operandu a úprava adresy. Konkrétne ide o prístup do pamäte, pričom je ako adresa použitý obsah registra SP/ESP a následne zvýšenie obsahu registra SP/ESP o šírku bunky zásobníka v bytoch. Z toho je možné odvodiť, že:

```
POP EAX
```
je ekvivalentné:
```
MOV EAX, [ESP]
ADD ESP, 4
```

Je nutné podotknúť, že dáta v zásobníku nie sú reálne odstránené, indikácia použitej/-voľnej bunky zásobníka je určená tým, či je adresa konkrétnej bunky pamäte rovná, alebo vyššia ako ukazovateľ na vrchu zásobníku. Toto opisuje chovanie zásobníka, ktorý v rámci architektúry x86 rastie smerom dole v pamäťovom priestore. [9]

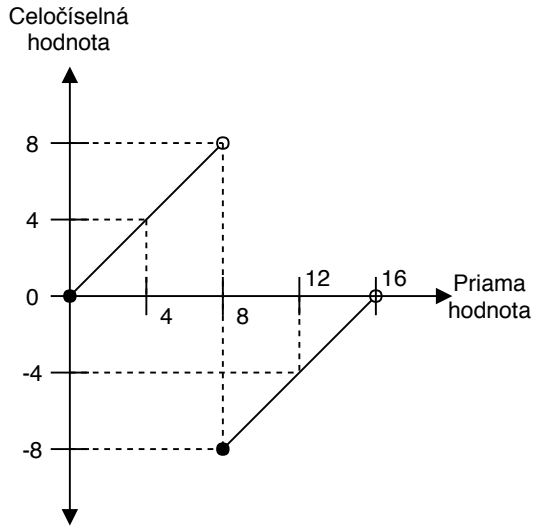
Z hľadiska implementácie procesor umožňuje dve v rámci operácii nad zásobníkom dve pamäťové šírky a to šestnásť (16) bitov a šírku naajväčšieho dostupného bitového rozsahu (tridsať dva (32) bitov pre architektúru IA—32 a šesťdesiatštyri (64) bitov v rámci režimu 64). Toto chovanie však môže spôsobiť stratu zarovnania a tým pádom aj stratu dát pri zle kontrolovanej/navrhnutej práci nad zásobníkom.

2.1.3 Doplnkový kód

Ukladanie hodnôt prostredníctvom kódovania dvojkového doplnku je spôsob reprezentácie celočíselných dát, ktorých rozsah nie je viazaný len na priamu reprezentáciu hodnoty, ale tiež implementuje spôsob ako dané dáta uložiť a zároveň indikovať, či je hodnota kladná, alebo záporná.

Graf[2.5] opisuje ako sú priame hodnoty (pre prehľadnosť je zvolený bitový rozsah štyri (4) bity) mapované prostredníctvom algoritmu dvojkového doplnku na celočíselné hodnoty rozšírené o reprezentáciu záporných hodnôt.

Tento systém taktiež umožňuje určiť, či je hodnota kladná, alebo záporná bez konverzie, ale len kontrolou najviac významného bitu (v chápaní binárnej reprezentácie ide o bit v



Binárna hodnota	Priama hodnota	Celočíselná hodnota
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

Obrázek 2.5: Reprezentácia vzťahu hodnoty v doplnkovom kóde a priamom kóde.

najvyššom ráde skúmaného rozsahu hodnôt), ktorý ak je nastavený na jedna, reprezentuje záporné číslo a v prípade nuly kladné číslo. [9]

Z hľadiska implementácie má takto šifrovaná hodnota výhodu, že nie je potrebné hardvérovo implementovať špecializovaná vetvenie pre ošetrenie možnej kombinácie kladných a záporných vstupov, ale binárne operácie sčítania a odčítania korektne mapujú výstupy do zodpovedajúcich hodnôt tejto operácie.

Táto vlastnosť sa taktiež vzťahuje aj na násobenie realizované prostredníctvom implementácie Boothovho algoritmu.

Hodnoty sú šifrované prostredníctvom nasledujúceho algoritmu [8] (v rámci algoritmu je predpokladané, že máme definovanú maximálnu šírku bitovej reprezentácie hodnôt a túto hodnotu je možné do takéhoto rozsahu zobraziť):

Kódovanie hodnoty:

1. Je vytvorená priama binárna reprezentácia hodnoty bez znamienka.
2. Ak je hodnota kladná, táto reprezentácia je finálna. Ak je hodnota záporná, binárna reprezentácia je bitovo invertovaná.
3. K výslednej hodnote je pripočítaná jednotka.
4. Výsledná hodnota je záporná hodnota kódovaná v doplnkovom kóde.

Dekódovanie hodnoty je analogické vzhľadom k tomu ako je kódovaná, čo činí tento spôsob univerzálnym, avšak prvotné rozhodnutie pre potreby šifrovania je riadené najvyšším bitom analyzovanej binárnej reprezentácie.

Dekódovanie hodnoty:

1. Ak je najvyšší bit binárnej reprezentácie nula, hodnota je interpretovaná priamo. Ak je najvyšší bit jedna, je potrebné hodnotu dekódovať.
2. Binárna reprezentácia zápornej hodnoty je bitovo invertovaná.
3. K výsledku je pripočítaná jednotka.
4. Výsledná hodnota ja priama reprezentácia zápornej hodnoty.

2.1.4 Spôsoby adresovania

Adresovanie operandov reprezentuje spôsob, akým je odkázané na dáta potrebné pre vykonanie operácie definovanej symbolickou inštrukciou. Z hľadiska tvorenia kódu je možné tieto spôsoby rozdeliť do niekoľkých skupín.

Registrový prístup Hodnota operandu je obsiahnutá v niektorom z dostupných registrov definovaných architektúrou. Z hľadiska kódu je táto adresa nahradená použitím prístupového kľúča asociovaného s registrom. Príkladom takéhoto prístupu je:

```
mov ax, bx
```

Okamžitý prístup Operand obsahuje okamžitú, konštantnú hodnotu definovanú programátorom, ktorá je nemenná počas vykonávania kódu. Z hľadiska kódu je táto hodnota zadaná jednou z podporovaných foriem definovaných v lexikálnych pravidlách jazyka. Príkladom takéhoto prístupu je:

```
mov ax, 0x0f0f
```

Priamy pamäťový prístup Dáta operandu sú obsiahnuté vo virtuálnej pamäti stroja a je k ním pristúpené cez symbolickú adresu, ktorá bola definovaná počas kompilácie. Príkladom takéhoto prístupu je:

```
buffer: resb 64
mov ax, [buffer]
```

Priamy pamäťový prístup s ofsetom Adresa použitá na prístup je vypočítaná na základe vstupných dát reprezentovaných algebraickým výrazom adresy a hodnotou ofsetu. Toto umožňuje prechádzať pamäťové úseky väčšie, než je maximálny rozsah registra. Príkladom takéhoto prístupu je:

```
buffer: resb 64
mov ax, [buffer+4]
```

Nepriamy registrový prístup Hodnota adresy je uložená v jednom z dostupných registrov, ktorého obsah je použitý pre prístup do pamäte. Príkladom takéhoto prístupu je:

```
mov ax, [bx]
```

2.2 Jazyk symbolických inštrukcií

Písanie programu pre konkrétnu architektúru je možné realizovať prostredníctvom kombinácie jazyka symbolických inštrukcií a programového prostriedku pre preklad takto zapísaného programu do strojového kódu. Toto prostredie je nazvané asembler a cieľom tejto práce je analyzovať asembler pre procesory rodiny x86.

Jazyk symbolických inštrukcií je nízko-úrovňový programovací jazyk, ktorý je cielený na konkrétnu architektúru. Tento jazyk predstavuje abstrakciu nízkej úrovne a tá umožňuje programátorovi využívať výrazov skladajúcich sa z kombinácii mnemotechnických označení strojových inštrukcií a operandov reprezentujúcich dáta inštrukcií. Tento jazyk tiež umožňuje využívať symbolické hodnoty, ktoré sa až v momente prekladu stanú reálnymi dátami a tým pádom prenechávajú výpočty konkrétnych adries v dátovej a kódovej pamäti nástroju pre preklad.

Množina inštrukcií strojového kódu je izomorfná množine symbolických inštrukcií. To znamená, že jednotlivé symbolické inštrukcie je možné interpretovať ako viacero inštrukcií strojového kódu, pričom konkrétny variant je definovaný kombináciou operandov.

Pre porovnanie je možné ako príklad použiť symbolickú inštrukciu ADD, ktorá reprezentuje aritmetickú operáciu celočíselného sčítania. Túto symbolickú inštrukciu je však možné interpretovať ako štrnásť (14) rôznych inštrukcií strojového kódu, pričom tieto varianty sú odlišné kombináciami operandov.

2.2.1 Strojový kód

Strojový kód, či konkrétne, inštrukcia strojového kódu je numerická hodnota identifikujúca špecifickú inštrukciu (tzv. Operačný kód). Tento operačný kód je mimo identifikátora rozšírený o prefixy a príznaky, ktoré procesoru umožňujú načítať z pamäte, prípadne iných vnútorných prostriedkov, ako sú registre a dáta reprezentujúce operandy.

Tieto dáta reprezentujú užitočné dáta, ktoré budú použité pre vyhodnotenie operácie, alebo kontrolné informácie ovplyvňujúce chovanie vykonávania inštrukcie.

2.2.2 Prostredie Netwide Assembler

Netwide Assembler (ďalej len NASM) je súbor pravidiel a programov, ktoré umožňujú písať, kontrolovať a kompilovať kód cielený na systémy s procesormi rodiny x86. Syntax tohto asembleru je založená na syntaxy Intel. [2]

NASM poskytuje funkcionalitu, ktorá umožňuje programátorovi využívať symbolických hodnôt pre riadenie toku programu (tzv. Návestie) a adries v rámci pamäte dát, ktorých konkrétna hodnota je odvodená až v momente prekladu. Mimo toho poskytuje sadu funkcionalít umožňujúcich vyhodnotenie okamžitých výrazov ako operandy a algebraických výrazov pre výpočet efektívnej adresy v rámci volania prístupov do pamäte.

Symbolická inštrukcia jazyka NASM je identifikovaná ako textový reťazec, pričom postupnosť týchto reťazcov, oddelených symbolom nového riadka predstavuje kód programu. Táto symbolická inštrukcia sa dá opísať ako:

```
NÁVESTIE : IDENTIFIKÁTOR OPERANDY ; KOMENTÁR
```

kde:

Návestie predstavuje alfanumerický reťazec (s niekoľkými ďalšími znakmi povolenými v rámci lexikálnych pravidiel), ktorý umožňuje predstaviť symbolickú hodnotu asocio-

vanú s konkrétnou adresou v rámci pamäte kódu. NASM pozná dva druhy návestí, globálne a lokálne, pričom lokálne návestie je identifikované špecifickým prefixom.

Globálne návestie je symbolická hodnota dostupná a unikátna v rámci celého kódu. Lokálne návestie je vždy prepojené s určitým globálnym návestím a je chápané ako unikátne v rámci rozsahu tohto globálneho návestie. V rámci lexikálnej špecifikácie NASM, dvojbodka nasledujúca za reťazcom návestia nie je nutná.

Identifikátor predstavuje jednoznačné označenie reprezentované textovým reťazcom symbolickej inštrukcie.

Operandy reprezentujú dáta, ktoré inštrukcia využíva v rámci svojho definovaného chovania, z hľadiska syntaxe NASM pozná štyri druhy operandov a to:

- Označenie registra
- Okamžitá hodnota
- Pamäťový prístup
- Symbolická hodnota

Jazyk taktiež umožňuje dodatočne špecifikovať veľkosť operácie, či prostredníctvom kľúčových slov určiť nejednoznačnú, či zmeniť konkrétnu veľkosť dát vstupujúcich do operácie. Tieto kľúčové slová reprezentujú dostupné šírky definované architektúrou. NASM pozná špecifikátory veľkosti ako (`BYTE, WORD, DWORD, QWORD, ...`).

Jednotlivé operandy sú z hľadiska syntaxe rozdelené symbolmi čiarky.

Komentár reprezentuje programátorom vytvorený komentár, ktorý je od užitočného kódu oddelený symbolom bodkočiarky a takýto komentár siaha až po symbol nového riadku, ktorý reprezentuje začiatok nového výrazu.

Jazyk NASM taktiež poskytuje rodinu pseudoinštrukcií, ktoré sú použité v rámci kompilácie pre vytvorenie a inicializáciu pamäťového priestoru dostupného programu.

Pre potreby simulácie budú analyzované inštrukcie rodiny RESc a Dc, ktoré slúžia k definícii pamäťových priestorov.

RESc pseudoinštrukcie, ktoré v rámci sekcie .BSS umožňujú rezervovať pamäťový priestor potrebnej veľkosti. Ako operand tejto inštrukcie je numerická hodnota, ktorá indikuje počet rezervovaných pamäťových buniek a symbol c v názve pseudoinštrukcie identifikuje veľkosť jednej rezervovanej bunky.

- RESB X: Rezervuj X bytov
- RESW X: Rezervuj X celkov veľkosti WORD (teda X * 2)
- RESD X: Rezervuj X celkov veľkosti DWORD (teda X * 4)

Dc pseudoinštrukcie, ktoré v rámci sekcie .DATA umožňujú rezervovať pamäťový priestor a inicializovať hodnoty v pamäti na základe definície zadanej programátorom. Ako operand tejto inštrukcie je postupnosť okamžitých hodnôt, ktorých obsah sa uloží do pamäte, pričom symbol c v názve pseudoinštrukcie identifikuje veľkosť ukladanej hodnoty v rámci pamäte.

- DB a1,a2,a3,...,aX: Rezervuj X bytov
- DW a1,a2,a3,...,aX: Rezervuj X celkov veľkosti WORD (teda X * 2)
- DD a1,a2,a3,...,aX: Rezervuj X celkov veľkosti DWORD (teda X * 4)

2.3 Modelovanie a Simulácia

Na základe získaných informácii o fyzickom systéme je možné začať proces modelovania. Teda vytvorenie abstraktného modelu za účelom opisu kľúčových prvkov fyzického systému. Takto zadefinovaný abstraktný model poslúži ako teoretický základ pre simulačný model. [5]

Výstupom modelovania je simulačný model, ktorý reprezentuje implementáciu abstraktného modelu prostredníctvom vhodne zvolených nástrojov. Samotný simulačný model je prepojený s funkcionalitou umožňujúcou používateľom vytvárať simulačné experimenty za účelom prehĺbenia vedomostí o simulovanom systéme.

2.3.1 Modelovanie

Zvolený fyzický systém, ktorý je modelovaný je kombinácia dvoch celkov a to procesor implementujúci architektúru x86 a jazyk symbolických inštrukcií (v rámci tejto práce zastúpený jazykom NASM).

Účelom modelovania je vytvorenie abstraktného modelu, teda reprezentácie fyzického systému, ktorý opisuje kľúčové prvky tohto systému ako komponenty s definovaným chovaním. Tieto komponenty predstavujú jednotlivé súčasti abstraktného modelu, ktoré spoločne umožňujú pozorovať napodobenie procesov prebiehajúcich skúmanom systéme.

Vytvorenie abstraktného modelu umožní definovať potrebné chovanie a kvality, ktoré je možné v neskoršej fázy interpretovať ako konkrétny simulačný model. Takýto simulačný model je vytvorený napríklad implementáciou prostredníctvom programovacieho jazyka, ktorý umožní opis tohto chovania tak, aby boli jeho kľúčové prvky transparentné pre používateľa.

2.3.2 Simulácia

Simulácia, teda proces vykonávania simulačných experimentov nad modelom je spôsob, akým môže používateľ získať cez simulačné nástroje nové vedomosti o modelovanom systéme. Z hľadiska tohto projektu je teda simulačný experiment chápaný ako pokus používateľa o vytvorenie, preklad a spustenie programu zapísaného jazykom symbolických inštrukcií.

Aby bolo možné získať relevantné informácie zo simulačného experimentu, je nutné, aby bol používateľ schopný sledovať zmeny, ktoré prebiehajú na modelovanom systéme. Tieto zmeny vyvoláva chovanie definované jednotlivými inštrukciami.

Toto je možné docieliť aj bez špecializovaného nástroja (ktorý je cieľom tejto práce), prostredníctvom prekladu programu so špeciálnymi podmienkami, avšak toto vyžaduje, aby používateľ disponoval strojom, ktorý implementuje architektúru x86 a znalosti potrebné pre objektový preklad a linkovanie cez dostupné debuggovacie nástroje. Účelom tejto práce je tento proces zefektívniť.

Kapitola 3

Návrh modelov a Simulačných prostriedkov

Táto kapitola bude reprezentovať výstup modelovacieho procesu, kedy bude nutné zvoliť určitú úroveň abstrakcie, ktorá umožní modelovať konkrétne, kľúčové prvky fyzického systému. Výstupom tohto bude abstraktný model, ktorý bude neskôr implementovaný prostredníctvom jazyka Java SE8 v podobe simulačného modelu a rozšírený o funkcionalitu, ktorá umožní detailné skúmanie stavu systému medzi jednotlivými vykonávaniami inštrukcií.

3.1 Návrh Abstraktného modelu

Abstraktný model reprezentujúci fyzický systém je agregáciou dielčích modelov vnútorných súčastí, ktoré majú definované chovanie a z hľadiska abstraktného modelu disponujú rozhraniami pre presun dát. Abstraktný model sa skladá z troch samostatných celkov, ktoré svojou spoluprácou vytvárajú chovanie imitujúce fyzický systém.

Abstraktný model prostriedkov architektúry x86, teda model reprezentujúci dostupné súčasti procesora, ako sú všeobecné registre, príznakový register a virtuálna pamäť. Jednotlivé komponenty majú definované chovanie, ako je spôsob ukladania dát a spôsoby, akým je možné k týmto dátam pristupovať.

Abstraktný model prekladu jazyka symbolických inštrukcií na strojový kód. Tento celok predstavuje spôsob, ktorým je používateľský kód programu prekladaný na inštrukcie strojového kódu. Súčasťou tohto je aj modelovanie spustiteľného kódu ako štruktúry udržiavajúcej dáta pochádzajúce z používateľom definovaného kódu.

Posledným modelom je model vykonávajúci kód. Predstavuje premostenie medzi prostriedkami a kódom, kedy definované chovanie v rámci modelu inštrukcie a dát tejto inštrukcie vykoná transformácie modelu prostriedkov.

Tieto tri spolupracujúce celky umožňujú vykonávať používateľom definovaný kód nad prostriedkami modelu architektúry x86.

3.2 Návrh Simulačného modelu Prostriedkov

Simulačný model prostriedkov reprezentuje štruktúru, ktorá udržiava stav systému. Procesor vykonávajúci kód je možné chápať ako množinu dát reprezentujúcich stav systému medzi

vyvolaním funkcionality jednotlivých inštrukcií. Pre potreby modelovania tohto chovania je nutné navrhnúť a implementovať jednotlivé súčasti ako spolupracujúce komponenty.

3.2.1 Návrh Registrov

Model všeobecného registra predstavuje štruktúru, ktorá udržiava dáta vo vnútornej premennej modelujúcou hardvérovú realizáciu registra. Teda ide o premennú predstavujúcu zdieľaný fyzický priestor konkrétnej bitovej šírky, pričom sú jednotlivé segmenty tohto priestoru asociované so špecifickými kľúčmi, či štítkami, ktoré umožnia externej komponente prístup k dátam na základe tohto kľúča.

Pre potreby realizácie kontroly kódu je nutné v rámci návrhu zvážiť aj rozšírenie funkcionality o detekciu náležitosti kľúča ku konkrétnej dátovej štruktúre a definovať veľkosť asociovaného priestoru. Pre potreby simulátora budú modelované len všeobecné registre a ich rozsah ohraničený horným maximom tridsaťdva (32) bitov.

3.2.2 Návrh pamäte

Pamäťový priestor, či model virtuálnej pamäte bude reprezentovať dátový priestor, ktorý bude možné použiť pre ukladanie a získanie dát prostredníctvom adresy a požadovanej šírky, či veľkosti manipulovaných dát. Z hľadiska simulácie nebude potrebné simulovať správu pamäte tak ako je realizovaná vo fyzickom systéme, keďže tento aspekt je vhodnejšie skúmať ako reálnu aplikáciu, než simuláciu.

Pamäťový priestor sprostredkuje funkcionalitou rozhranie pre prístupy do pamäte, alokáciu, či umožnenie použitia blokov pamäte na základe pseudoinštrukcií[2.2.2] predstavujúcich mapovanie adries pamäte do symbolických hodnôt.

3.2.3 Návrh príznakového registra

Príznakový register je posledná súčasť modelu prostriedkov, ktoré umožňujú sledovať stav systému. Príznakový register modeluje svoje chovanie analogicky k tomu, ako je modelované vo fyzickom systéme, teda predstavuje štruktúru registra obsahujúceho vnútornú dátovú premennú, ktorej bitová reprezentácia umožňuje analyzovať pravdivostné hodnoty príznakov.

Pre potreby tohto modelu bude funkcionalita rozšírená o systém, ktorý umožní asociovať konkrétne indexy s identifikátormi a následne prostredníctvom týchto identifikátorov zistiť pravdivostnú hodnotu príznaku.

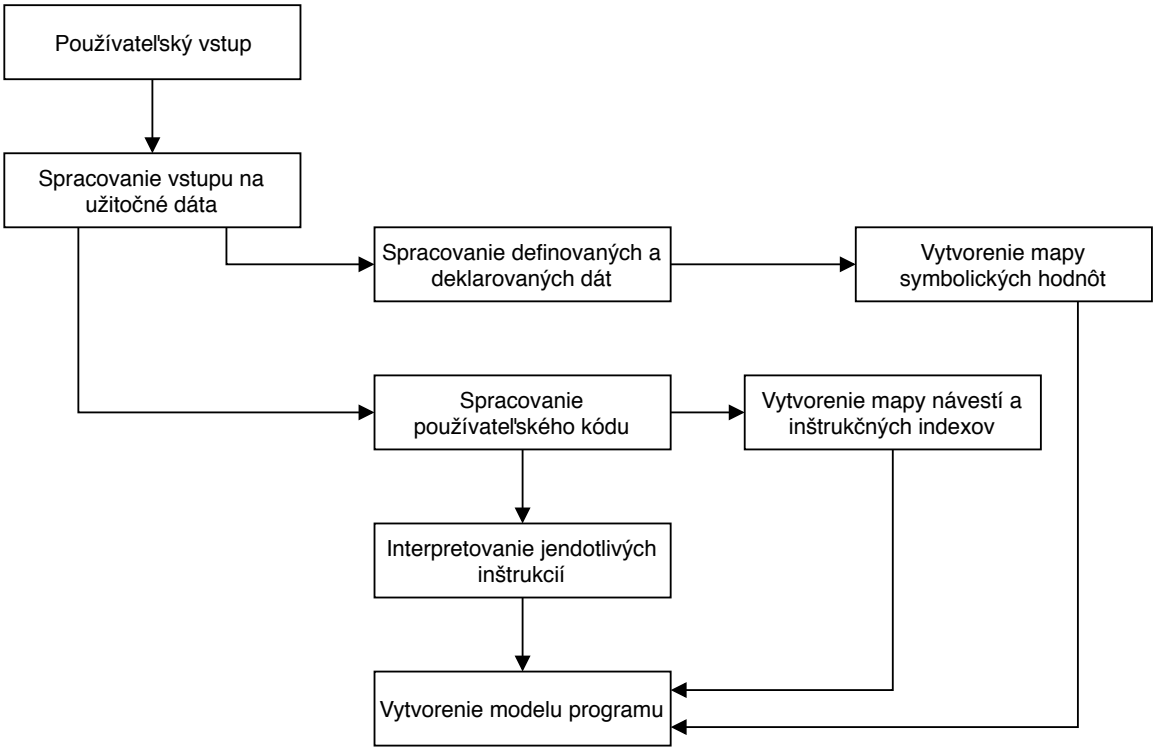
3.3 Návrh Simulačného modelu Interpretácie

Interpretácia predstavuje druhý celok, ktorá umožňuje modelovať používateľom vytvorený kód ako sústavu inštrukcií s definovaným chovaním a dátových premenných opisujúcich operandy.

3.3.1 Interpreter

Interpretácia predstavuje druhý celok, ktorá umožňuje modelovať používateľom vytvorený kód ako sústavu inštrukcií s definovaným chovaním a dátových premenných opisujúcich operandy.

Tento proces je možné opísať diagramom[3.1], ktorý opisuje spôsob, ako je používateľský vstup spracovaný a z týchto dát je sú vytvorené štruktúry udržiavajúce relevantné informácie.



Obrázek 3.1: Ilustrovanie procesu spracovania používateľského vstupu.

Opísaný proces predstavuje spôsob, ktorým je používateľský vstup spracovaný a rozdelený na jednotlivé výrazy jazyka NASM. Tieto výrazy následne podliehajú spracovaniu za účelom vytvorenia užitočných dátových štruktúr a to:

Mapa návestí , čo je štruktúra, ktorá v rámci analýzy kódu vytvorí asociácie medzi jednotlivými výrazmi označujúce návestia a inštrukčnými indexmi v rámci používateľom definovaného kódu. Tieto návesti sú uložené v štruktúre, ktorá konkrétne návestie asociuje s konkrétnym indexom a umožňuje inštrukciám riadiacich tok programu vykonávať zmeny inštrukčného ukazovateľa (teda inštrukcie podmienených a nepodmienených skokov).

Mapa symbolov , čo je štruktúra, ktorej účelom je udržiavať adresy pseudoinštrukciami definovaných priestorov v pamäti a asociovaných symbolických hodnôt. Súčasťou tvorenia tohto objektu je aj modifikácia pamäťového modelu za účelom legalizácie prístupov k týmto konkrétnym adresám.

Ako posledný krok interpretácie je použitie hlavnej továrne, čo je štruktúra, ktorá na základe spracovávaného výrazu poskytuje adekvátnu továreň. Táto konkrétna továreň spracuje výraz a interpretuje ho ako model inštrukcie s náležitými dátami.

3.3.2 Inštrukčná Továreň

Inštrukčná továreň je kľúčový prvok, ktorý zabezpečuje modularitu navrhovaného systému. Na základe existencie adekvátnej továrne navráti hlavná továreň, čo je organizačná štruktúra, špecifický továrenský objekt. Tento objekt je v rámci svojej implementácie schopný identifikovať, či výraz náleží tejto konkrétnej továrne (na základe identifikátora zo štruktúry symbolickej inštrukcie).

Nasleduje proces interpretácie, ktorá kontroluje lexikálnu (rozpoznávanie jednotlivých súčastí výrazu), syntaktickú (zoradenie jednotlivých častí podľa ich typu, či ide o operand, separátor operandov) a sémantickú (použitie legálneho operandu pre túto konkrétnu inštrukciu) stránku výrazu.

V prípade úspešnej interpretácie továreň navráti model reprezentujúci inštrukciu s relevantnými dátami.

3.3.3 Inštrukcia

Model inštrukcie zahrňuje jej statickú a dynamickú zložku. Pod statickou zložkou sa chápe opis chovania inštrukcie, teda to, čo bude inštrukcia vykonávať s dátami. Dynamickou zložkou sú konkrétne dáta realizované ako operandy. Takáto inštrukcia poskytuje len jednu funkcionalitu a to vykonanie seba samej nad prostriedkami modelovaného systému, ktoré sú konkretizované v programátorom definovanom kóde.

3.3.4 Operand

Operand predstavuje dynamickú zložku modelu inštrukcie. Z hľadiska vykonávania sú všetky operandy chápané ako rovnocenné a inštrukcia samotná nie je schopná detegovať konkrétny typ dát, nad ktorými pracuje.

Inštrukcia vykonáva svoje pevne definované chovanie nad dátami, pričom tieto dáta, teda operandy reprezentujú prístupy ku konkrétnym prostriedkom modelovaného systému ako sú napríklad registre, pamäťové prístupy, či okamžité hodnoty.

3.3.5 Interpretovaný Kód

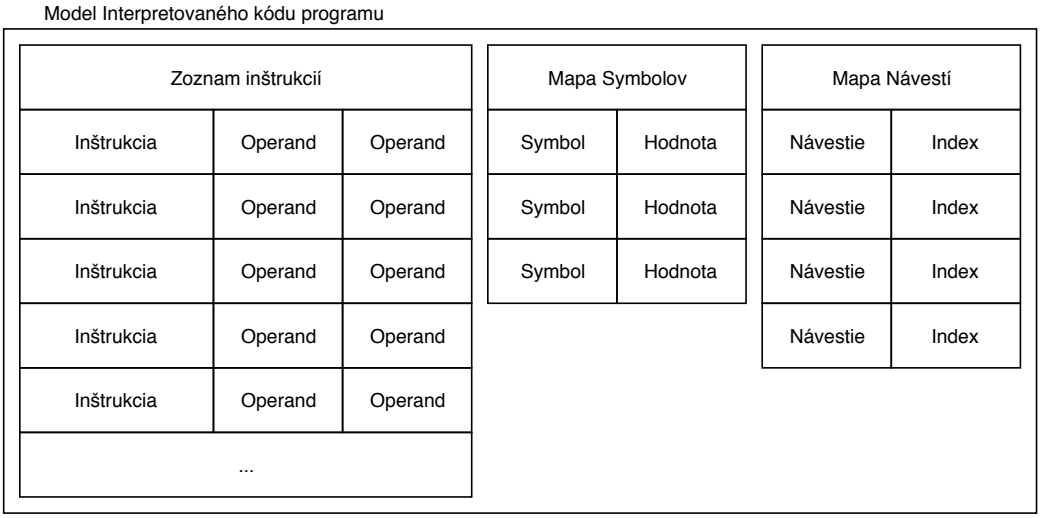
Interpretovaný kód je dátová štruktúra, ktorá umožňuje organizáciu dát súvisiacich s používateľom definovaným kódom. Cieľom je, aby tento model bolo možné poskytnúť modelu simulujúceho vykonávanie programu, teda disponuje dátovými štruktúrami, ktoré sú produktom interpretácie, ako vyššie opísané mapy návestí a symbolických hodnôt a štruktúra udržiavajúca indexovanú sekvenciu inštrukcií.

Táto vnútorná štruktúra je opísaná diagramom[3.2].

3.4 Návrh Vykonávania kódu

Systém, ktorý zabezpečí vykonávanie kódu predstavuje previazanie modelu programu a modelu prostriedkov systému. Tento model predstavuje štruktúru, ktorá na základe dát z modelu prostriedkov (konkrétne inštrukčného ukazovateľa) extrahuje z modelu interpretovaného kódu konkrétnu inštrukciu.

Následne, napodobujúc chovanie fyzického systému. Inkrementuje inštrukčný ukazovateľ a vyvolá metódu opisujúcu chovanie inštrukcie.



Obrázek 3.2: Štruktúra interpretovaného kódu.

Tu model opakuje svoje chovanie, pokým nenarazí na koniec programu, či nenastane situácia, ktorá vyvolá chybu (ako napríklad prístup do ilegálnej pamäte, delenie nulou (0), alebo pretečenie zásobníka).

3.5 Návrh Používateľskho rozhrania

Používateľské rozhranie predstavuje spôsob, ktorým používateľ bude vykonávať simulačné experimenty. Ako simulačná nástroj sa dá rozdeliť do dvoch podôb.

Rozhranie, ktoré umožní používateľovi vkladať vstup vo forme textu reprezentujúceho kód a dáta a rozhranie, ktoré umožní vykonávať samotnú simuláciu vykonávania kódu.

3.5.1 Editor kódu

Rozhranie umožňuje používateľovi definovať tri špecifické vstupy a to:

Kód , čiže užitočný kód, ktorý bude vykonaný simulátorom, tento priestor je rezervovaný pre symbolické inštrukcie a návestia.

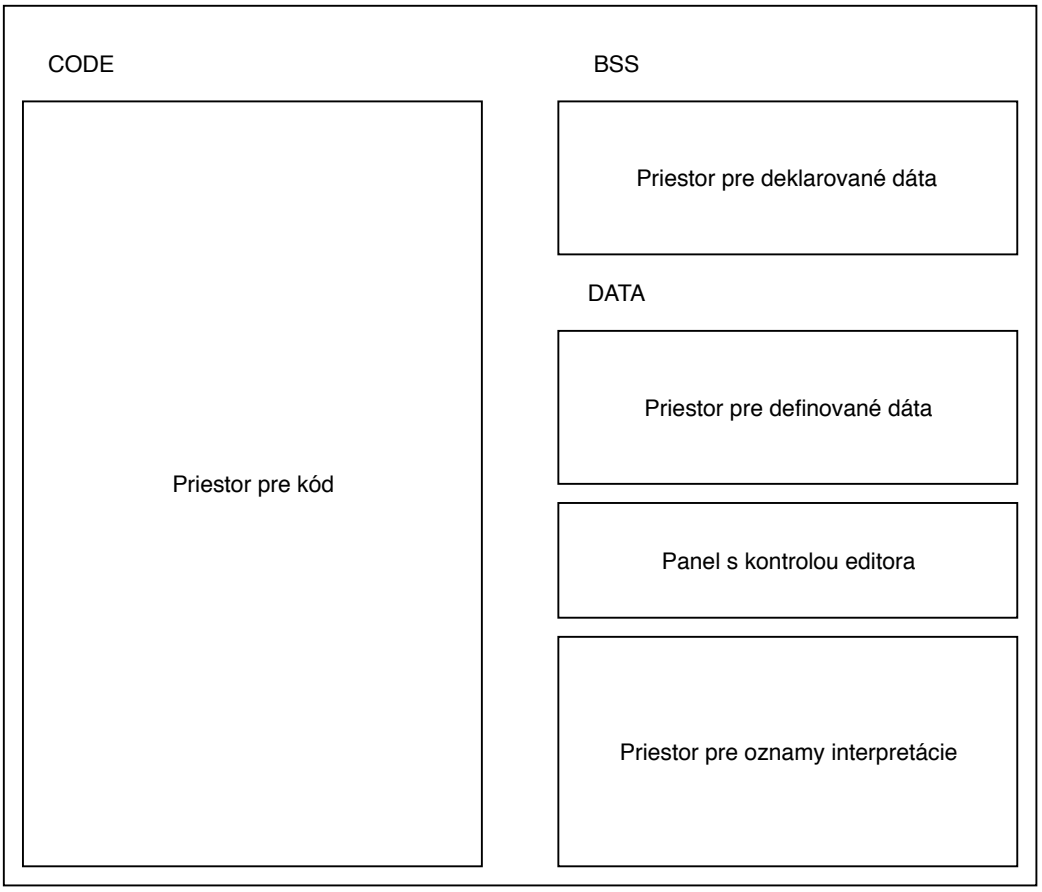
Deklarované dáta , ktoré predstavujú dátové priestory v pamäti bez špecifikovaných hodnôt, teda pre pseudoinštrukcie rodiny RESc.

Definované dáta , ktoré sú určené k definovaniu dátových priestorov s konkrétnymi hodnotami prostredníctvom pseudoinštrukcií rodiny Dc.

Editor umožňuje vytvoriť takéto vstupy a následne ich podrobiť interpretácii. Rozloženie návrhu tohoto rozhrania je možné vidieť v obrázku[3.3].

3.5.2 Simulátor kódu

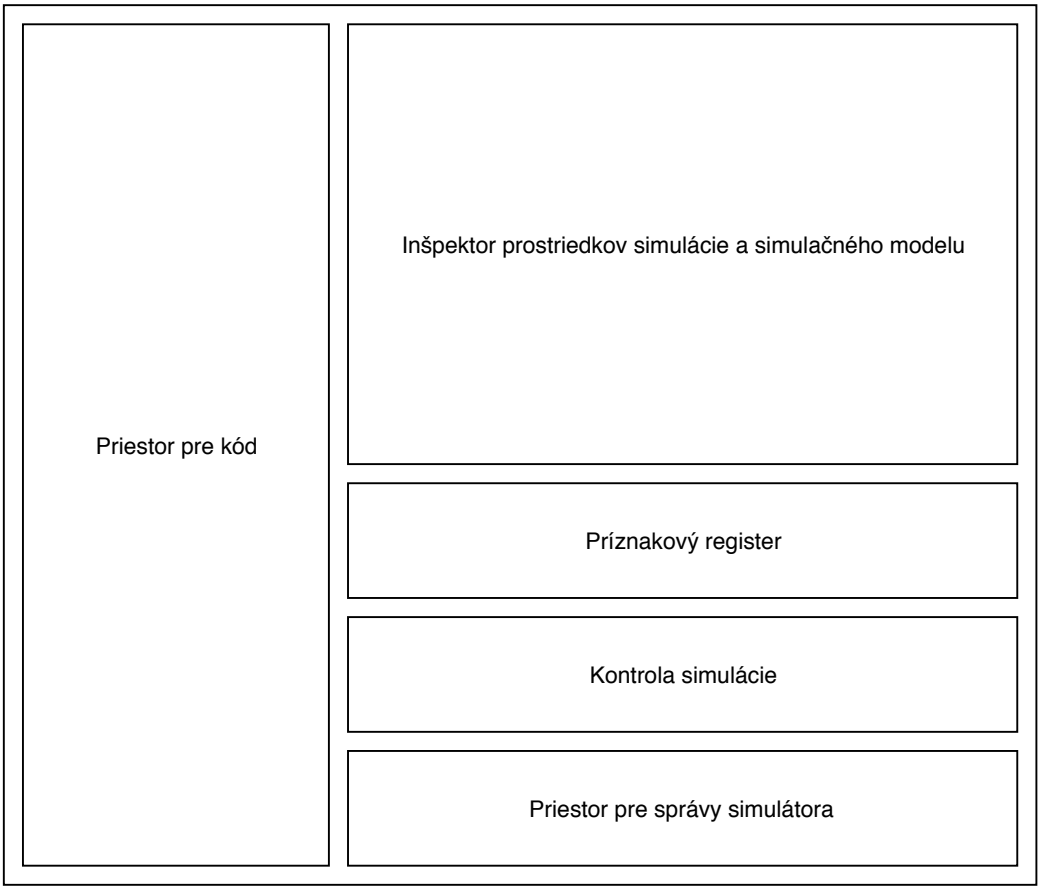
Rozhranie simulačnej obrazovky umožňuje vykonávanie používateľského kódu, pričom umožňuje študovať stav systému, pod týmto sa rozumie:



Obrázek 3.3: Bávrh rozloženia editoru.

- Inštrukcia naplánovaná na vykonanie v nasledujúcom kroku.
- Inšpektor stavu prostriedkov systému.
- Príznakový register.
- Správy simulačného modelu.

Ovládanie simulátora je sprostredkované cez kontrolný panel. Návrh rozloženia grafického požívateľského rozhrania je možné vidieť v obrázku[3.4].



Obrázek 3.4: Návrh rozloženia simulátora.

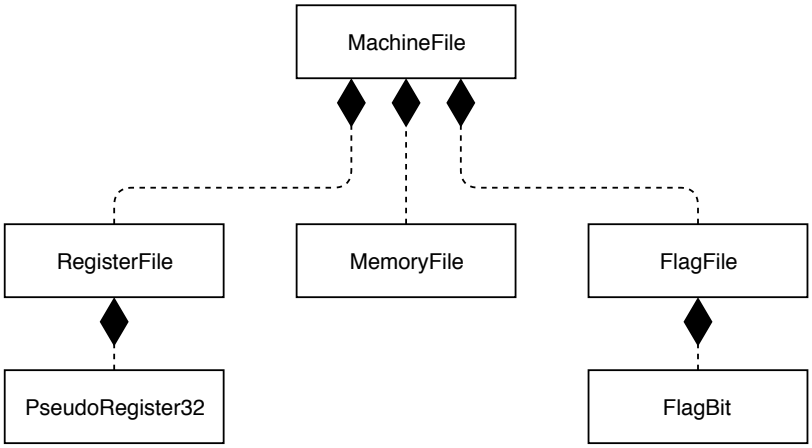
Kapitola 4

Implementácia Simulačných prostriedkov

Táto kapitola pojednáva o konkrétnej implementácii navrhnutých modelov prostredníctvom jazyka Java SE8. V predošlej kapitole bola opísaná principiálna funkcionalita jednotlivých súčastí simulačných prostriedkov a to, ako budú spolupracovať, aby napodobili chovanie fyzického systému.

4.1 Implementácia prostriedkov stroja

Prostriedky stroja chápeme všetky prostriedky definované architektúrou ako súčasti schopné udržiavať dáta. Z hľadiska programového riešenia sú tieto súčasti určené len ako kontajnery, ktoré na základe prístupových dát (či ide o kľúč označujúci register, alebo adresu v pamäti) umožňujú ukladať, načítavať a meniť vnútorné dáta. Hierarchia štruktúry prostriedkov stroja je definovaná obrázkom[4.1].



Obrázek 4.1: Prostriedky stroja.

Tieto súčasti nedefinujú chovanie, čo je súčasťou modelu inštrukcie, ale predstavujú systém dát, nad ktorými bude toto chovanie realizované.

4.1.1 Register

Register je realizovaný triedou PseudoRegister32. Táto trieda obsahuje pole typu byte, ktoré na základe architektúrou definovanej maximálnej bitovej šírky (tridsaťdva (32) bitov), má dĺžku štyri (4).

Táto trieda taktiež disponuje štvoricou reťazcových premenných, ktoré svojím obsahom indikujú konkrétny kľúč a jeho asociáciu s konkrétnym bitových priestorom. Tieto kľúče, či *štítky* sú definované ako:

- `fullTag` – asociovaný s rozsahom 0 - 31
- `halfTag` – asociovaný s rozsahom 0 - 15
- `lowTag` – asociovaný s rozsahom 0 - 7
- `highTag` – asociovaný s rozsahom 8 - 15

Na základe modelovaného registra sú tieto hodnoty definované v rámci inicializácie štruktúry udržiavajúcej strojové prostriedky.

Trieda taktiež implementuje metódy, ktoré umožňujú prístup k týmto bitovým šírkam, ako dvojice pre nastavenie/získanie hodnoty:

- `void setFull(int value) / int getFull()`
- `void setHalf(int value) / int getHalf()`
- `void setLow(int value) / int getLow()`
- `void setHigh(int value) / int getHigh()`

Tieto metódy umožňujú prístup k dátam na základe využitia funkcionality z balíka NIO, ktorý umožňuje zaobaliť bytové pole a konvertovať ho na celočíselnú hodnotu, pričom zohľadňuje doplnkový kód.

Tieto metódy (v rámci získavania hodnoty z registra) taktiež zohľadňujú hodnotu najvyššieho (najviac významného) bitu a doplnia ho pre plný rozsah celočíselnej hodnoty int.

Trieda taktiež implementuje metódy, ktoré indikujú, či daný kľúč náleží konkrétnej inštancii a deteguje veľkosť asociovaného priestoru v bytoch k danému kľúču.

4.1.2 Pamäť

Pamäťový priestor, či model virtuálne pamäte je realizované triedou MemoryFile, ktorá implementuje pamäť prostredníctvom dvoch vnútorných premenných vo forme pola typu byte.

Dátové pole, ktoré reprezentuje adresovateľný priestor, kde jednotlivé byty sú prístupné pamäťové bunky.

Pole povolení umožňuje indikovať, či má program povolený prístup k danej adrese. Povolenie je kódované formou, kedy je konkrétny index bitu v tomto poli asociovaný s adresou pamäťovej bunky v hlavnom dátovom poli.

Táto trieda poskytuje pre simuláciu dve základné metódy a to:

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

Metódy umožňujú simulácii ukladať a čítať hodnoty z pamäte. Tieto prístupové metódy implementujú kontrolu legality prístupu do pamäte na základe validity adresy a požadovaného rozsahu. Je vhodné zmieniť, že metódy vracajú celočíselnú hodnotu typu int a reflektujú najvyšší bit požadovanej sekcie jeho doplnením (teda zachovávajú znamienko).

Metódy taktiež implementujú systém Little-Endian[2.3].

Pre interpretáciu taktiež trieda implementuje metódu pre alokovanie požadovanej veľkosti pamäte.

4.1.3 Príznaky

Príznakový register je implementovaný analogicky k triede PseudoRegister32, avšak namiesto kľúčov, ktoré asociujú konkrétne bitové rozsahy táto trieda implementuje dodatočný systém, ktorý umožňuje asociovať konkrétne indexy bitov špecifickým kľúčom a následne tieto hodnoty sprístupniť prostredníctvom metód používajúcich kľúče ako argumenty.

Tento systém umožňuje využívať namiesto fixných indexov triviálne názvy a asociovať ich s konkrétnym bitom, pričom sa tieto dáta udržiavajú v dodatočnom viazanom zozname poskytovanom programovacím jazykom.

4.2 Implementácia interpretácie

Interpretácia predstavuje proces spracovania používateľského kódu a navrátenia dátovej štruktúry, ktorá bude daný kód reprezentovať.

K tomu využíva komplexnej funkcionality spracovania textu na užitočné informácie, generovania dátových štruktúr udržiavajúcich tieto informácie a následne spracovanie výrazov do inštancií simulačných modelov popísaných v nasledujúcich sekciách. Kontrola toku interpretácie je realizovaná modulárnym systémom, ktorý sa skladá z postupnosti testov a prekladov na virtuálnu reprezentáciu kódu.

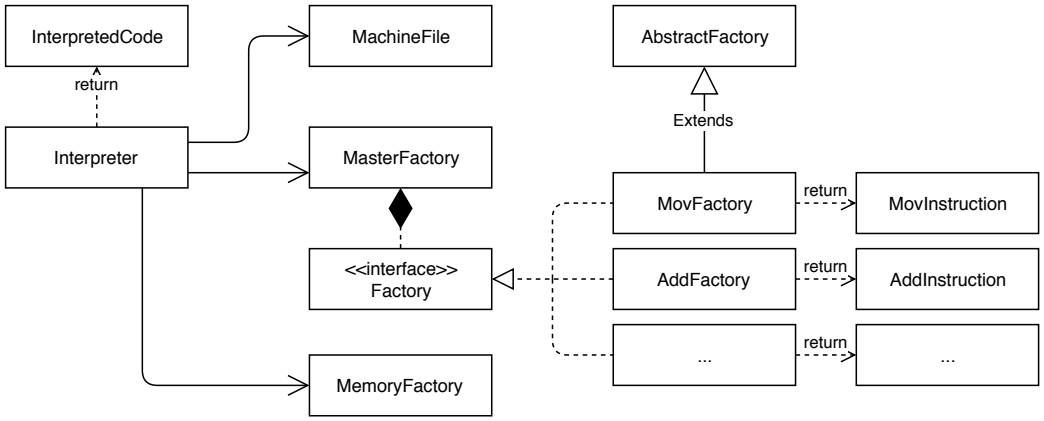
V prípade vyskytnutia používateľom zanesenej chyby je riadenie realizovaný vytvorením výnimiek, ktoré zachytáva nadriadená časť programu a zotavuje sa z nich za účelom vytvorenia chybového hlásenia.

4.2.1 Interpreter

Interpretácia používateľského vstupu je realizovaná triedou Interpreter (ako je definovaný, spolu s podpornými triedami v obrázku[4.2]), ktorá svojou funkcionalitou umožňuje spracovanie používateľských vstupov. Toto je realizované v dvoch fázach.

Prvou fázou je spracovanie vstupných reťazcov pomocou textových funkcionalít umožňujúcich rýchle a efektívne rozloženie vstupných dát na užitočné a neužitočné informácie. Užitočné informácie sú následne podrobené sanitácii, teda procesu očistenia redundantných znakov, ktorá naformátuje text tak, aby ho bolo možné v rámci vnútorných procesov jednoduchšie spracovať. V tomto kroku taktiež dochádza k nedokonalej lexikálnej analýze, teda sú riadky kódu podrobené lexikálnym testom a je odhadnutý ich typ, teda, či sa jedná o riadok obsahujúci návestie, inštrukciu, alebo špeciálny simulačný príkaz.

Nasledujúcou fázou je spracovanie užitočných informácií v dvoch slučkách. Prvou sú používateľom definované pamäťové priestory a dáta, ktoré sú realizované spracovaním pseudoinštrukcií rodiny RESc a Dc. Tieto vytvoria v pamäťovej reprezentácii alokované pamäťové sekcie, ktorých hodnota je nastavená, prípadne ponechaná prázdnou (nulovaná) ak ide o deklaráciu a nie definíciu. Toto vyprodukuje prvú časť dátovej štruktúry interpretovaného



Obrázek 4.2: Štruktúra interpretačného modulu.

kódu, čo je trieda udžiavajúca symbolického názvy a ich hodnoty v rámci pamäťového priestoru.

Návestia sú spracované, pričom je im udelený jednoznačný inštrukčný index. Tento krok taktiež zabezpečuje kontrolu redeklarácie návestí a generuje triedu udržiavajúcu mapy návestí a asociovaných inštrukčných indexov. Produktom je dátová štruktúra slúžiaca ako navigačná mapa asociujúca textový reťazec pomenovania návestie a jeho index v zozname inštrukcií.

V tomto bode dochádza k druhému kroku interpretácie, teda systematickému spracovaniu výrazov obsahujúcich symbolické inštrukcie. To je realizované pomocou triedy MasterFactory, ktorá udržiava referencie na všetky dostupné továrenské objekty. Tieto objekty, implementujúce rozhranie, ktoré im definuje dvojicu metód:

- `boolean isMyStatement(String statement)`
- `Instruction interpretStatement(String stat, MachineFile mF, DataBundle dB)`

Prvá metóda určí na základe identifikátora, či výraz náleží továrni. Druhá metóda interpretuje výraz do konkrétnej inštancie inštrukcie.

Ako argumenty taktiež vyžaduje strojový súbor, čiže reprezentáciu prostriedkov architektúry (potrebné pre referencie na objekty reprezentujúce registre) a dátový balík, ktorý obsahuje symbolické hodnoty adries a indexy návestí (pre kontrolu dátových prístupov do pamäte využívajúce symbolické hodnoty a skokové inštrukcie).

Z hľadiska interpretácie môže trieda počas prekladu vytvoriť výnimku, ktorá identifikuje chybu v používateľom definovanom kóde. Proces kontroly a prekladu je navrhnutý tak, aby sa bol schopný z chyby zotaviť a pokúsiť dokončiť preklad, pričom používateľovi zosumarizuje všetky problémy a chyby, na ktoré počas prekladu narazil v rámci vnútorných premenných výnimky.

V prípade úspešného prekladu Interpreter vráti inštanciu interpretovaného kódu pripraveného na spustenie.

4.2.2 Inštrukčná Továreň

Inštrukčná továreň implementuje rozhranie Factory, ktoré je popísané vyššie v sekcii[**??**]. Tieto metódy predstavujú lexikálny test, či výraz náleží továrni a následne metódu, ktorej

cieľom je analýza lexiky, syntaxe a sémantiky výrazy a následná interpretácia do inštancie inštrukcie.

Lexikálna analýza , ktorá určuje, či sú používateľom vytvorené reťazce rozpoznateľné, ako základné, informáciu nesúce lexémy. Teda, že jednotlivé reťazce je možné interpretovať ako platné dáta.

Syntaktická analýza zabezpečuje to, že používateľom definované lexémy nesú syntakticky korektný význam skladajúci štruktúru výrazu symbolickej inštrukcie. To je možné chápať ako test, či za rozpoznateľným identifikátorom nasledujú platné pole operandov (korektnej dĺžky a typu).

Sémantická analýza zabezpečuje, že syntakticky korektný výraz nesie platnú konfiguráciu informácii. Teda, že konkrétna inštrukcia má platnú kombináciu typov operandov korektných veľkostí.

Inštrukčné továrne majú prostredníctvom abstraktnej triedy AbstractFactory dané k dispozícii metódy spracovania výrazov pre zjednodušenie samotnej interpretácie. Medzi tieto metódy patrí extrakcia dát ako je identifikátor inštrukcie, extrakcia operandov, konverzia okamžitých hodnôt na platné celočíselné reprezentácie a testy používateľom definovaných obmedzení veľkosti.

Cieľom interpretácie výrazu je vyprodukovať platnú inštrukciu viazanú k typu továrne, prípadne vytvoriť chybové hlásenie pre nadradený program.

4.2.3 Inštrukcia

Inštrukcie implementujú rozhranie Instruction, ktoré zaobaľuje konkrétne implementácie špecifických inštrukcií všeobecným rozhraním definujúcim niekoľko metód: `boolean isDebug()`
`boolean isEnd()`
`DebugData getDebugData()`
`void execute(MachineFile mFile, DataBundle dBundle)`
Metóda `void execute(MachineFile mFile, DataBundle dBundle)` vo svojom tele opisuje to, ako sa inštrukcia zachová počas vykonávania a aké zmeny vykoná nad systémom.

Inštrukcie môžu pre potreby dediť špecifickú funkcionalitu od abstraktnej inštrukcie, ktorej úlohou je zjednodušiť kontroly dát pre potreby korektného nastavenia príznakového registra.

Architektúra chovania samotnej inštrukcie je zjednodušená využívaním objektov implementujúcich rozhranie Operand, ktoré maskuje štyri (4) rôzne implmenetácie operandov (ako register, pamäťový prístup, okamžitá hodnota a symbolická hodnota). Účelom tohto prístupu je eliminovať potrebu kontroly typu operandov a špecifikácie chovania pre konkrétne implementácie. Namiesto toho používateľ vyvoláva len metódy pre získanie/nastavenie hodnoty operandu.

Toto môže v špeciálnych prípadoch vytvoriť výnimku, ak sa používateľ pokúša nastaviť hodnotu okamžitej hodnoty, avšak táto situácia by nemala nastať, keďže je toto považované za sémantickú chybu v rámci interpretácie.

4.2.4 Operand

Systém implementuje štyri typy operandov, ktoré sprostredkujú prístup k dátam inštrukcie.

Okamžitá hodnota , tento operand reprezentuje konštantu definovanú programátorom a ako taký, tento operand neumožňuje meniť hodnotu konštanty, keďže je to sémantický chybný výraz.

Register , tento operand sprístupňuje register prostredníctvom platného registrového kľúča. Jeho hodnotu je možné čítať a meniť. Vzhľadom k návrhu funguje tento operand ako zabaľovacia funkcionalita, ktorá vyžaduje platnú inštanciu objektu MachineFile pre získanie/zmenu hodnoty.

Pamäťový prístup , tento operand realizuje prístup do pamäte na základe kalkulácie efektívnej adresy, ktorú definuje programátor.

Operand umožňuje využívať komplexného algebraického výrazu pre priamy prístup do pamäte prostredníctvom symbolickej hodnoty, nepriamy prístup prostredníctvom adresy uloženej v registry a prístup do pamäte s ofsetom, ktorý je realizovaný algebraickým výrazom.

Pre kalkuláciu adresu je použitý algoritmus Shunting Yard, rozšírený o podporu unárnych operácií a zátvoriek. V rámci inicializácie pamäťového operandu je vykonaná kontrola pre vyhodnocovanie okamžitých hodnôt prostredníctvom skalárnych operácií.

Symbolická hodnota , táto hodnota je považovaná za konštantu, či okamžitú hodnotu, ktorá nie je známa v momente písania kódu, ale je známa v momente vykonávania kódu po úspešnej interpretácii. Podobne ako Okamžitá hodnota, tento operand neumožňuje zmeniť konkrétnu hodnotu priradenej symbolickej hodnoty.

4.2.5 Interpretovaný Kód

Interpretovaný kód je trieda, ktorá udržiava všetky relevantné informácie opisujúce kód definovaný používateľom (dátová štruktúra ilustrovaná obrázkom[4.3]).

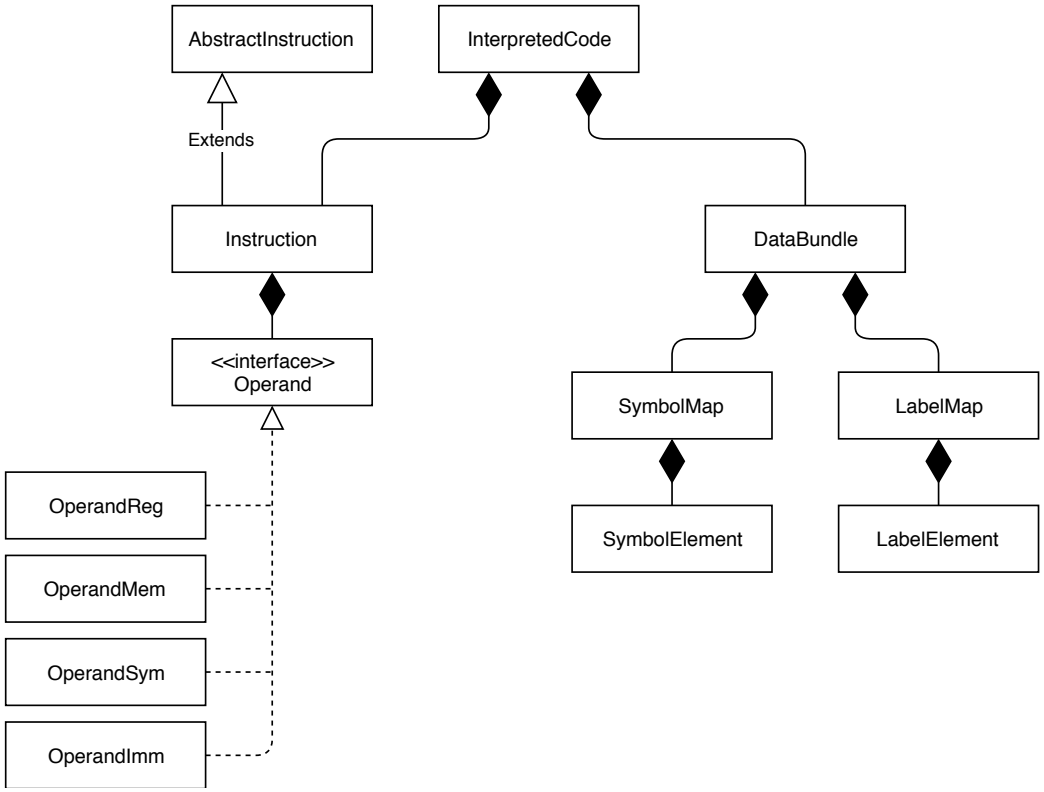
Realizuje sekvenciu inštrukcií ako zoznam, ktorý je možné pristupovať prostredníctvom indexov a obsahuje dátové štruktúry so symbolickými pomenovaniami hodnôt (návestia a pomenované pamäťové adresy) a ich reálnymi hodnotami.

4.3 Implementácia vykonávania

Vykonávanie interpretovaného kódu je riešené triedou vykonávač (Executor). Táto trieda zabezpečuje postupnosť úkonov, ktoré sa dajú označiť ako simulácia vykonávané kódu s ohľadom na definované chovanie fyzického systému.

Vykonávač používa ukazovateľa v strojovom súbore k identifikovanie inštrukcie naplánovanej na vykonanie. Následne inkrementuje tento ukazovateľ a vyvolá chovanie konkrétnej inštancie inštrukcie zo zoznamu inštrukcií interpretovaného kódu. Vykonávač pre potreby simulácie poskytuje sériu metód, ktoré predstavujú primárnu kontrolu simulačného experimentu, teda vykonávania kódu a to:

```
void executeStep()
void executeRun()
void executeContinue()
void restart()
```



Obrázek 4.3: Štruktúra interpretačného modulu.

Kde metódy umožňujú, ako naznačujú ich pomenovania, vykonať jeden krok, vykonať celý, neprerušený beh od aktuálneho stavu, vykonať beh až po najbližšiu značku pre prerušenie vykonávania a reštartovať strojový súbor, čo vykoná aj reštartovanie vykonávania programu.

Z hľadiska vnútornej implementácia vykonávač vyvoláva inštrukcie na základe ukazovateľa a následne vykonáva ich vnútorný opis chovania, pričom testuje či bolo dosiahnuté konca programu realizované špeciálnou inštrukciou definovanou v rámci simulačných prostriedkov.

4.4 Implementácia používateľského rozhrania

Používateľské rozhranie, tak ako bolo opísané v návrhu, predstavuje možnosť ako môže používateľ riadiť simulačné experimenty.

Z hľadiska implementácie bolo použité prostredie Swing, ktoré poskytuje nutné kontajnery, riadiace objekty a chovanie potrebné pre zabezpečenie interakcie používateľa a aplikácie. Je dôležité zmieniť, že grafické rozhranie obsahuje špeciálnu triedu `TextLineNumber`[6], ktorá nepochádza z prostriedkov jazyka Java, ani nebola vytvorená v rámci tejto práce, ale prevzaná a upravená pre potreby projektu.

Základom je primárny rámec, čiže najvyšší kontajner, ktorý udržiava aktuálnu, používateľom vybranú funkčnú obrazovku (teda kompozíciu grafických, dátových a riadiach elementov zabezpečujúcich funkcionalitu).

Vnútorné obrazovky zabezpečujú svojou funkcionalitou dátovú perzistenciu počas prepnutia obrazoviek a predstavujú tak dva oddelené celky. Editor, pre vytvorenie kódu a Simulátor pre vykonávanie simulačných experimentov.

Pre potreby reprezentácie dát zo strojového súboru bola implementovaná trieda umožňujúca, na základe používateľom zvolených kritérií, interpretovať dáta ako textové reťazce.

Kapitola 5

Záver

Výsledkom tejto práce je aplikácia, ktorá umožňuje testovať krátke sekvencie kódu napísané v jazyku symbolických inštrukcií určeného pre procesory rodiny x86.

Tieto sekvencie využívajú aplikáciou podporovaných inštrukcií, pričom spracovanie samotného kódu simuluje navrhnutý abstraktný model reálneho, fyzického prostredia a systému.

Pre potreby zjednodušenia, táto aplikácia sprostredkováva len bežné, používateľovi prístupné súčasti prostredia, ako sú všeobecné registre, či virtualizovaný pamäťový priestor.

Tento prístup bol zvolený za účelom priblíženia kľúčových prvkov používateľom, ktorých znalosti danej problematiky sú len povrchné a slúži k pochopeniu základných konceptov.

Simulátor vyššej kvality by vyžadoval hĺbkové spracovanie fyzického systému a úpravu abstraktného a simulačného modelu v rozsahu, ktorý prevyšuje zvolený rozsah tejto práce.

Testovanie správnosti bolo založené na dostupných dátach, primárne technickej dokumentácii od spoločnosti Intel, ktoré opisuje prostriedky procesorov implementujúcich architektúru x86. [1]

Na základe tohto testovania je možné povedať, že aplikácia, teda simulačný model spĺňuje dostatočnú mieru validity, aby bolo možné pozorovať a skúmať vykonávanie inštrukcií a k tomu náležiace zmeny realizované nad modelom reprezentujúcim prostriedky procesora ako sú registre a pamäť.

Literatura

[1] *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Intel Corporation, Marec 2018.
URL https://software.intel.com/en-us/articles/intel-sdm

[2] *Netwide Assmebler Documentation*. The Netwide Assembler development team, verzia 2.13.03, [Online; navštívené 16.4.2018].
URL https://software.intel.com/en-us/articles/intel-sdm

[3] Allen, I. D.: *The CARRY flag and OVERFLOW flag in binary arithmetic*. [Online; navštívené 15.4.2018].
URL http://teaching.idallen.com/dat2343/10f/notes/040_overflow.txt

[4] Bhattacharjee, Abhishek and Lustig, Daniel: *Architectural and operating system support for virtual memory*. Morgan & Claypool Publishers, 2018, ISBN 9781627056021.

[5] Bossel, H.: *Modeling and simulation*. AK Peters, 1994, ISBN 978-1568810331.

[6] Camick, R.: *Text Component Line Number*. [Online; navštívené 1.3.2018].
URL
https://tips4java.wordpress.com/2009/05/23/text-component-line-number/

[7] Duntemann, J.: *Assembly language step-by-step: programming with Linux*. Wiley, 2009, ISBN 978-0-470-49702-9, 3rd ed.

[8] Finley, T.: *Two's Complement*. Cornell CIS, Duben 2000, [Online; navštívené 15.4.2018].
URL https://www.cs.cornell.edu/~tomf/notes/cps104/twoscomp.html

[9] Hyde, R.: *The Art of Assembly Language ; 2nd Edition*. No Starch Press, 2010, ISBN 978-1-59327-207-4.

[10] Irvine, K. R.: *Assembly language for x86 processors*. Pearson, 2015, ISBN 978-0-13-602212-1.

Příloha A

Manuál

Preklad zdrojových súborov prostredníctvom príkazu `make`.

Výsledkom je spusitetľný archív s príponou .jar, ktorý po spustení ukáže používateľovi okno s obrazovkou Editora, tu je možné:

- Písať kód (oblasť `.CODE`)
- Definovať pamäťové priestory (oblasť `.DATA`)
- Deklarovať pamäťové priestory (oblasť `.BSS`)
- Kontrolným panelom vybrať jeden z predpripravených kódov
- Kontrolným panelom vybrať spustiť kontrolu kódu
- Kontrolným panelom vybrať spustiť kontrolu a vykonanie
- Sledovať prípadne chybové hlásenia na konzole

Po spustení simulácie sa obsah okna zmení na Simulačnú obrazovku, tu je možné:

- Kontrolným panelom spustiť kód
- Kontrolným panelom krokovať kód
- Kontrolným panelom spustiť po najbližší bod break
- Kontrolným panelom reštartovať vykonávanie
- Sledovať stav modelu procesora:
  - Sledovať obsahy registrov
  - Sledovať obsah pamäte
  - Generovať výpis pamäte ako pola prvkov

Příloha B

Podporované inštrukcie

Táto príloha obsahuje sumarizáciu podporovaných inštrukcií v rámci implementovaného simulačného prostredia.

B.1 Dátové inštrukcie

Množina inštrukcií určených k presunom dát medzi pamäťou a registrami. Simulátor implementuje inštrukcie:

`MOV` inštrukcia uloženia dát. Inštrukcia uloží dáta opísané druhým operandom (zdrojovým, povolené sú: Register, Pamät, Okamžitá hodnota, Symbolická hodnota) a uloží ich do prvého operandu (cieľového, povolené sú Register alebo Pamäť). Inštrukcia nepodporuje prenos z pamäte do pamäte.

Inštrukcia neovplyvňuje žiadne príznaky.

`XCHG` inštrukcia výmeny dát. Inštrukcia vymení obsah prvého a druhého operandu. Prípustné sú len Registrové a Pamäťové operandy, pričom inštrukcia nepodporuje výmenu medzi dvomi pamäťovými sekciami.

Inštrukcia neovplyvňuje žiadne príznaky.

`PUSH` inštrukcia uloženia operandu do zásobníku. Inštrukcia uloží hodnotu operandu do zásobníku, ktorého prvky sú zarovnané na šetnásť (16) alebo tridsaťdva (32) bitov. Ako operand sú prípustné všetky typy.

Inštrukcia neovplyvňuje žiadne príznaky.

`POP` inštrukcia načítania dát zo zásobníku do operandu. Inštrukcia uloží hodnotu na vrchole zásobníka do vybraného operandu (ktorý môže byť typu Register alebo Pamäť).

Inštrukcia neovplyvňuje žiadne príznaky.

B.2 Aritmetické inštrukcie

Množina inštrukcií určených k vykonaniu aritmetických operácií nad operandmi. Simulátor implementuje inštrukcie:

`ADD` inštrukcia cločíselného súčtu. Inštrukcia sčíta prvý (cieľový, možný Register alebo Pamäť) operand s druhým (zdrojovým, možné všetky typy) a výsledok uloží do cieľového operandu.

Inštrukcia na základe výsledku nastavuje `Sign`, `Zero` a `Parity` flag. Na základe vstupov a výsledku sú nastavené `Overflow` a `Carry` flag.

`SUB` inštrukcia celočíselného odčítania. Inštrukcia odčíta druhý (zdrojový, možné všetky typy) od prvého (cieľového, možný Register alebo Pamäť) a výsledok uloží do cieľového operandu.

Inštrukcia na základe výsledku nastavuje `Sign`, `Zero` a `Parity` flag. Na základe vstupov a výsledku sú nastavené `Overflow` a `Carry` flag.

`MUL` inštrukcia neznamienkového násobenia. Inštrukcia vynásobí obsah registra AX (zvolený rozsah registra sa určuje od veľkosti operandu) s operandom a výsledok uloží na základe veľkosti operandu.

- 8 bit: Zdrojom je register AL, výsledok sa ukladá do AX
- 16 bit: Zdrojom je register AX, výsledok sa ukladá do DX:AX
- 32 bit: Zdrojom je register EAX, výsledok sa ukladá do EDX:EAX

Inštrukcia na základe výsledku nastavuje `Overflow` a `Carry` flag. Ak je horná polovica výsledku nula (0), príznaky sú nastavené na nula (0), inak jedna (1).

`DIV` inštrukcia neznamienkového delenia. Inštrukcia na základe veľkosti operandu (analogicky s inštrukciou `MUL`) vydelí hodnotu v konkrétnom rozsahu a výsledok delenia uloží ako: celočíselný výsledok do dolnej polovice rozsahu, zvyšok po delení do hornej polovice rozsahu.

- 8 bit: Zdrojom je register AX, výsledok sa ukladá do AL, zvyšok do AH
- 16 bit: Zdrojom je register DX:AX, výsledok sa ukladá do AX, zvyšok do DX
- 32 bit: Zdrojom je register EDX:EAX, výsledok sa ukladá do EAX, zvyšok do EDX

Inštrukcia nemá definované príznaky. V prípade aritmetickej chyby vytvorí procesor výnimku delenia nulou.

`DEC` inštrukcia dekrementu, teda zníženia hodnoty operandu o jedna (1) a uloženie tejto hodnoty do operandu. Ako operand je možné použiť Pamäť alebo Register.

Inštrukcia na základe výsledku nastavuje `Sign`, `Zero`, `Parity` a `Overflow` flag. Príznak `Carry` nie je ovplyvnený.

`INC` inštrukcia inkrementu, teda zvýšenia hodnoty operandu o jedna (1) a uloženie tejto hodnoty do operandu. Ako operand je možné použiť Pamäť alebo Register.

Inštrukcia na základe výsledku nastavuje `Sign`, `Zero`, `Parity` a `Overflow` flag. Príznak `Carry` nie je ovplyvnený.

`NEG` inštrukcia aritmetickej inverzie hodnoty, teda obrátenie znamienka s ohľadom na doplnkový kód, v ktorom sú aritmetické hodnoty ukladané. Ako operand je možné použiť Pamäť alebo Register.

Inštrukcia na základe výsledku nastavuje `Sign`, `Zero`, `Parity` a `Overflow` flag. Príznak `Carry` je nastavený na nula (0), pokiaľ bola hodnota operandu nula (0), inak je príznak nastavený na jedna (1).

B.3 Logické inštrukcie

Množina inštrukcií určených k vykonaniu logických operácií nad operandmi. Simulátor implementuje inštrukcie:

`AND` inštrukcia logického bitového súčinu. Inštrukcia logicky vynásobí prvý (cieľový, možný Register alebo Pamäť) operand s druhým (zdrojovým, možné všetky typy) a výsledok uloží do cieľového operandu.

Inštrukcia na základe výsledku nastavuje `Sign`, `Zero` a `Parity` flag. Príznak `Carry` a `Overflow` je nastavený na nula (0).

`OR` inštrukcia logického bitového súčtu. Inštrukcia logicky sčíta prvý (cieľový, možný Register alebo Pamäť) operand s druhým (zdrojovým, možné všetky typy) a výsledok uloží do cieľového operandu.

Inštrukcia na základe výsledku nastavuje `Sign`, `Zero` a `Parity` flag. Príznak `Carry` a `Overflow` je nastavený na nula (0).

`XOR` inštrukcia logického exkluzívneho bitového súčtu. Inštrukcia logicky exklúzivne sčíta prvý (cieľový, možný Register alebo Pamäť) operand s druhým (zdrojovým, možné všetky typy) a výsledok uloží do cieľového operandu.

Inštrukcia na základe výsledku nastavuje `Sign`, `Zero` a `Parity` flag. Príznak `Carry` a `Overflow` je nastavený na nula (0).

`NOT` inštrukcia logickej negácie. Inštrukcia invertuje bitové hodnoty zdrojového operandu (možný Register alebo Pamäť) a výsledok uloží do operandu.

Inštrukcia neovplyvňuje žiadne príznaky.

B.4 Posuvné inštrukcie

Množina inštrukcií určených k vykonaniu bitových posunov nad operandmi. Simulátor implementuje inštrukcie:

`SHR` inštrukcia logického bitového posunu doprava. Inštrukcia posunie binárnu hodnotu cieľového operandu (Register alebo Pamäť) o počet pozícií definovaných druhým operandom (Okamžitá hodnota alebo Register CL) doprava (teda smerom k najmenej významnému bitu), čo je ekvivalentné neznamienkovému deleniu cieľovej hodnoty číslom dva (2). Nový bit doplnený na najvýznamnejšiu pozíciu je nula (0). Najmenej významný bit je uložený ako hodnota príznaku `Carry`. Výsledok je uložený do zdrojového operandu.

Inštrukcia na základe výsledku nastavuje `Sign`, `Zero` a `Parity` flag. Príznak `Carry` obsahuje hodnotu posledného posunutého bitu. `Overflow` je v prípade jednotkového posunu nastavený na hodnotu najmenej významného bitu pôvodnej hodnoty.

`SHL a SAL` inštrukcia bitového posunu doľava. Inštrukcia posunie binárnu hodnotu cieľového operandu (Register alebo Pamäť) o počet pozícií definovaných druhým operandom (Okamžitá hodnota alebo Register CL) doľava (teda smerom k najvýznamnejšiemu bitu), čo je ekvivalentné vynásobeniu cieľovej hodnoty číslom dva (2). Nový bit doplnený na najmenej významnú pozíciu je nula (0). Najviac významný bit je uložený ako hodnota príznaku `Carry`. Výsledok je uložený do zdrojového operandu.

Inštrukcia na základe výsledku nastavuje `Sign`, `Zero` a `Parity` flag. Príznak `Carry` obsahuje hodnotu posledného posunutého bitu. `Overflow` je v prípade jednotkového posunu nastavený na hodnotu najvýznamnejšieho bitu pôvodnej hodnoty.

`SAR` inštrukcia aritmetického bitového posunu doprava. Inštrukcia posunie binárnu hodnotu cieľového operandu (Register alebo Pamäť) o počet pozícií definovaných druhým operandom (Okamžitá hodnota alebo Register CL) doprava (teda smerom k najvýznamnejšiemu bitu), čo je ekvivalentné znamienkovému deleniu cieľovej hodnoty číslom dva (2). Nový bit doplnený na najvýznamnejšiu pozíciu je rovný druhému najvýznamnejšiemu bitu. Najmenej významný bit je uložený ako hodnota príznaku `Carry`. Výsledok je uložený do zdrojového operandu.

Inštrukcia na základe výsledku nastavuje `Sign`, `Zero` a `Parity` flag. Príznak `Carry` obsahuje hodnotu posledného posunutého bitu. `Overflow` je v prípade jednotkového posunu nastavený na hodnotu najmenej významného bitu pôvodnej hodnoty.

B.5 Rotujúce inštrukcie

Množina inštrukcií určených k vykonaniu bitových rotácii nad operandmi. Simulátor implementuje inštrukcie:

`ROL` inštrukcia bitovej rotácie doľava. Inštrukcia posunie prvý operand (zdroj, možná Pamäť alebo Register) o počet pozícií definovanýd druhým operandom (Okamžitá hodnota alebo register CL) doľava (smerom k najmenej významnému bitu), pričom bity, ktoré sú takto vysunuté mimo bitového rozsahu sú kopírované na začiatok rozsahu.

Inštrukcia na základe výsledku nastavuje `Sign`, `Zero` a `Parity` flag. Príznak `Carry` obsahuje hodnotu posledného posunutého bitu. `Overflow` je v prípade jednotkového posunu nastavený na hodnotu exkluzívneho súčinu `Carry` a najviac významného bitu výsledku.

`ROR` inštrukcia bitovej rotácie doprava. Inštrukcia posunie prvý operand (zdroj, možná Pamäť alebo Register) o počet pozícií definovanýd druhým operandom (Okamžitá hodnota alebo register CL) doprava (smerom k najvýznamnejšiemu bitu), pričom bity, ktoré sú takto vysunuté mimo bitového rozsahu sú kopírované na koniec rozsahu.

Inštrukcia na základe výsledku nastavuje `Sign`, `Zero` a `Parity` flag. Príznak `Carry` obsahuje hodnotu posledného posunutého bitu. `Overflow` je v prípade jednotkového posunu nastavený na hodnotu exkluzívneho súčinu prvého a druhého najviac významného bitu výsledku.

`RCL` inštrukcia bitovej rotácie doľava cez `Carry`. Inštrukcia posunie prvý operand (zdroj, možná Pamäť alebo Register) o počet pozícií definovanýd druhým operandom (Okamžitá hodnota alebo register CL) doľava (smerom k najmenej významnému bitu),

pričom v rámci jednotlivých rotácií je príznak `Carry` kopírovaný do najmenej významného bitu a najvýznamnejší bit je ukladaný do príznaku `Carry`.

Inštrukcia na základe výsledku nastavuje `Sign`, `Zero` a `Parity` flag. Príznak `Carry` obsahuje hodnotu posledného posunutého bitu. `Overflow` je v prípade jednotkového posunu nastavený na hodnotu exkluzívneho súčinu `Carry` a najviac významného bitu výsledku.

`RCR` inštrukcia bitovej rotácie doľava cez `Carry`. Inštrukcia posunie prvý operand (zdroj, možná Pamäť alebo Register) o počet pozícií definovanýd druhým operandom (Okamžitá hodnota alebo register CL) doprava (smerom k najmenej významnému bitu), pričom v rámci jednotlivých rotácií je príznak `Carry` kopírovaný do najvýznamnejšieho bitu a najmenej významný bit je ukladaný do príznaku `Carry`.

Inštrukcia na základe výsledku nastavuje `Sign`, `Zero` a `Parity` flag. Príznak `Carry` obsahuje hodnotu posledného posunutého bitu. `Overflow` je v prípade jednotkového posunu nastavený na hodnotu exkluzívneho súčinu `Carry` a najviac významného bitu pôvodnej hodnoty zdroja.

B.6 Kontrolné inštrukcie

Množina inštrukcií určených k vykonaniu porovnaní operandov. Simulátor implementuje inštrukcie:

- `TEST` inštrukcia testu logického súčinu. Ekvivalentné inštrukcii `AND`, kedy kopíruje chovanie a nastavenie príznakov, no s rozdieľom, že výsledok nie je uložený.
- `CMP` inštrukcia testu aritmetického odčítania. Ekvivalentné inštrukcii `SUB`, kedy kopíruje chovanie a nastavenie príznakov, no s rozdieľom, že výsledok nie je uložený.

B.7 Pseudoinštrukcie

Množina inštrukcií definovaných v rámci jazyka NASM, určených k rezervovaniu pamäťového priestoru a prideleniu adresy symbolickej hodnote. Simulátor implementuje inštrukcie:

- `RESB, RESW, RESD`
- `DB, DW, DC`

Příloha C

Podporované podmienené skoky

Podmienené skoky podporované simulátorom, podmienky sú opísané ako výsledok porovnania dvoch hodnôt, či špecifického stavu systému. Tabuľka[C.1] opisuje konkrétny identifikátor inštrukcie, ako je použitý a aké podmienky sú testované simulátorom pre korektné nastavenie vetvenia kódu.

Pokiaľ nie je povedané inak, podmienka je opísaná ako výsledok použitia testovacej inštrukcie `CMP`, teda porovnanie hodnôt prostredníctvom ich odčítania.

Skoky sú kriticky významným prvkom asembleru x86, keďže je tento jazyk definovaný ako jazyk kontrolovaný vetvením vykonávania programu a pochopenie využitia skokových inštrukcií je kľúčové pre osvojenie si znalostí týkajúcich sa jazyka symbolických inštrukcií pre procesory rodiny x86.

Tabulka C.1: Skoky a ich podmienky

Inštrukcia skoku	Opis podmienky	Strojová podmienka
JA	Priama hodnota je väčšia	(CF==0 and ZF==0)
JAE	Priama hodnota je väčšia alebo rovná	(CF==0)
JB	Priama hodnota je menšia	(CF==1)
JBE	Priama hodnota je menšia alebo rovná	(CF==1 or ZF==1)
JC	Príznaku Carry je 1	(CF==1)
JCXZ	Hodnota registru CX je 0	(CX==0)
JECXZ	Hodnota registru ECX je 0	(ECX==0)
JE	Hodnoty sú si rovné	(ZF==1)
JG	Hodnota je väčšia	(ZF==0 and SF==OF)
JGE	Hodnota je väčšia alebo rovná	(SF==OF)
JL	Hodnota je menšia	(SF!=OF)
JLE	Hodnota je menšia alebo rovná	(ZF==1 or SF!=OF)
JNA	Priama hodnota nie je väčšia	(CF==1 or ZF==1)
JNAE	Priama hodnota nie je väčšia alebo rovná	(CF==1)
JNB	Priama hodnota nie je menšia	(CF==0)
JNBE	Priama hodnota nie je menšia alebo rovná	(CF==1)
JNC	Príznaku Carry je 0	(CF==0)
JNE	Hodnota nie je rovná	(ZF==0)
JNG	Hodnota nie je väčšia	(ZF==1 or SF!=OF)
JNGE	Hodnota nie je väčšia alebo rovná	(SF!=OF)
JNL	Hodnota nie je menšia	(SF==OF)
JNLE	Hodnota nie je menšia alebo rovná	(ZF==0 and SF==OF)
JNO	Príznak Overflow je 0	(OF==0)
JNP	Príznak Parity je 0	(PF==0)
JNS	Príznak Sign je 0	(ZF==0)
JNZ	Príznak Zero je 0	(ZF==0)
JO	Príznak Overflow je 1	(OF==1)
JP	Príznak Parity je 1	(PF==1)
JPE	Príznak Parity je 1	(PF==1)
JPO	Príznak Parity je 0	(PF==0)
JS	Príznak Sign je 1	(SF==1)
JZ	Príznak Zero je 1	(ZF==1)