Univerza v Mariboru

Fakulteta za elektrotehniko,

računalništvo in informatiko

DIPLOMSKO DELO

Maribor, september 2002 Sašo Greiner

Univerza v Mariboru

Fakulteta za elektrotehniko,

računalništvo in informatiko

Arhitektura za objektno orientirane jezike

Avtor:	Sašo Greiner
Študijski program:	Univerzitetni, Računalništvo in informatika
Smer:	Programska oprema
Mentor:	doc. dr. Janez Brest
Somentor:	prof. dr. Viljem Žumer

ZAHVALA

Za pomoč pri opravljanju tega diplomskega dela se zahvaljujem mentorju doc. dr. Janezu Brestu, somentorju prof. dr. Viljemu Žumerju in vsem sodelavcem iz laboratorija za računalniške arhitekture in jezike.

Avtor: Sašo Greiner

Naslov: Arhitektura za objektno orientirane jezike

UDK: 004.3:004.43 (043.2)

Ključne besede: arhitektura računalnikov, programski jeziki, prevajalniki, virtualni stroj

POVZETEK

V diplomskem delu je predstavljena arhitektura abstraktnega stroja, ki je bila modelirana za cilj prevajanja iz splošnega objektno usmerjenega programskega jezika. Izdelan je bil model za definicijo splošnega stroja, zbirniški jezik in instanca navideznega stroja za operacijski sistem Linux na x86 arhitekturi. Opisane so slabosti in prednosti posameznih tipov arhitektur kot tudi celoten postopek načrtovanja nove arhitekture in njene implementacije.

Ideja za novo arhitekturo se je pojavila kot težnja za zmanjšanje semantičnega prepada med splošnim visokim programskim jezikom in strojnim nivojem, za katerega se prevaja. V ta namen smo razvili arhitekturo z velikim številom splošnonamenskih registrov in kompleksnimi instrukcijami, ki operirajo nad sestavljenimi podatkovnimi tipi. Pri implementaciji virtualnega stroja smo se osredotočili na razvoj in razširljivost interpreterja ter ga zasnovali tako, da je čas, potreben za režijo sistema, čimkrajši.

Title: Architecture for object-oriented languages

Keywords: computer architecture, programming languages, compilers, virtual machine

ABSTRACT

This work presents the architecture of an abstract machine, which was modelled as a target platform for compiling a general object-oriented language. A scheme for defining a general processor architecture has been devised as well as an assembly language and lastly a virtual machine for the Linux operating sistem, running on an x86. Advantages and drawbacks of architectural types are presented and also the whole procedure to defining a new machine and its implementation.

The idea has arisen for the purpose of narrowing the semantic gap between a general high level programming language and its machine level representation. On this account an architecture with many general-purpose registers and complex instruction set which itself operates on complex data structures has been built. In the implementation of our virtual machine the main focus has been on the development of the interpreter and its extensibility.

Kazalo

Slike

Tabele

1. Uvod

Ko se vsakodnevno, bodisi doma ali ob delu, srečujemo z računalniško obdelavo podatkov nas zanimata predvsem zmogljivost in zanesljivost delovanja računalnika. Malokdo, celo izmed zahtevnejših uporabnikov, pri tem pomisli na dejansko implementacijo ali organizacijo procesorja in naprav, ki tvorijo jedro slehernega računalniškega sistema. Prav o tem, natančneje o arhitekturi računalnika, govori to diplomsko delo, katerega namen je čimbolje in v najjasnejši obliki predstaviti abstraktni računalnik, postopek načrtovanja, prednosti in slabosti, ter nenazadnje tudi uporabo na praktičnih primerih. Področje, ki ga računalniška arhitektura zajema, je precej široko in ga je zato mogoče klasificirati na več podpodročij in vsakega obravnavati posamezno ter na več načinov. Mi se bomo ustalili na področju specificiranja procesorskih ukazov in njihovega izvajanja. Povodov za pohitritev izvajanj posameznih ukazov je mnogo. Hitrost pa je samo ena izmed množice lastnosti, h katerim težimo. Podoben problem je kompleksnost ukazov in preslikava iz višjenivojskega programskega jezika v zbirni jezik ali strojni kod. Čim višji je programski jezik, tem večja bo razlika med tema dvema nivojema in s tem tudi zahtevnost postopka preslikave. Namen raziskav v okviru diplomskega dela je prav zmanjšanje kompleksnosti te pretvorbe na način, pri katerem se bosta nivoja visokega programskega jezika in strojnega koda karseda približala. V ta namen je mogoče oklestiti semantiko programskega jezika ali jo zvišati oz. narediti abstraktnejšo na zbirniškem nivoju.

Ker želimo programski jezik ohraniti na visoki ravni abstrakcije, se bomo odločili za drugo možnost, tj. posodobitev računalniške arhitekture, na kateri se bo strojni kod izvajal. Jedro dela je potemtakem zmanjševanje t.i. semantičnega prepada, ki se v vse večji meri pojavlja pri vseh novodobnih, še posebno pa pri objektno usmerjenih programskih jezikih, kjer so razlike med strojnim in programskim nivojem zelo velike. Vse to zelo otežuje delo programerjem prevajalnikov, ki v veliki meri zavisijo od ciljne arhitekture, za katero se generira strojni kod in povečuje kompleksnost samega procesa prevajanja. Da bi bilo le-to preprostejše in karseda učinkovito, je potrebno zasnovati

boljšo in bogatejšo arhitekturo, ki bo bolje podpirala objektno okolje, v katerem teče program. Premostiti je potrebno težave, ki se pojavljajo pri majhnem številu registrov, kar daje kot rezultat neoptimalno prevajanje in večkrat neposrečeno določene operande posameznih instrukcij.

Za dosego tega cilja bo načrtovana abstraktnejša računalniška arhitektura, kot jih povečini poznamo danes, s podporo splošnemu objektno usmerjenemu jeziku, ki bo z razumljivimi razlogi podpirala veliko število splošnonamenskih registrov in učinkovite operacije nad njimi brez stranskih učinkov. Kot integralni del procesorja bodo za delo z abstraktnimi podatkovnimi strukturami izdelane razširjene procesne enote. Izdelan bo tudi zbirniški jezik, v katerem bo mogoče napisati poljuben program za definirano arhitekturo in ga prevesti v vmesni kod, primeren za izvajanje na navideznem stroju. V končni fazi bo za namene testiranja implementiran tudi navidezni stroj, s katerim bo simulirano delovanje procesorja nad neko množico instrukcij. Ta bo implementiral abstraktno arhitekuro, ki bo definirana povsem specifično glede na problem.

Zgradba diplomskega dela je podana v preostanku uvoda.

V drugem poglavju bosta predstavljeni arhitekturi tipa RISC in CISC, njune prednosti in slabosti ter podobnosti in razlike, ki ju ločujejo. Prikazani bodo problemi prevajanja za posamezno arhitekturo in morebitne rešitve, ki bi kakorkoli pripomogle k boljši vmesni kodi.

V okviru tretjega poglavja bo predstavljena nova računalniška arhitektura hibridnega tipa. Podrobno bo opisan razvojni pristop do strukture arhitekture in postopek načrtovanja posameznih procesorskih enot. V središče bo torej postavljena definicija arhitekture, zgradba ukazov in operandov ter načini pomnilniških naslavljanj. Podane bodo operacijske kode vseh ukazov in format instrukcij.

Četrto poglavje bo namenjeno za predstavitev virtualnega stroja, ki bo definirano arhitekturo implementiral. Opisane bodo podrobnosti uporabljenega izvajalnega modela, tehnik zaščite, nalaganja in prenaslavljanja. Prav tako bo opisana implementacija interpreterja, ki skrbi za izvajanje instrukcij na virtualnem stroju.

Peto poglavje bo podalo uporabo arhitekture s pomočjo primerov kode, katera bo postavljena ob tisto, ki je bila generirana avtomatsko kot produkt prevajanja iz jezika C ali jave. Opisana bo tudi uporaba zbirnika za prevajanje programa v vmesno obliko in uporaba virtualnega stroja vključno s formatom programskih datotek, ki jih stroj uporablja.

Delo zaključimo s sklepnim poglavjem, v katerem je povzeto jedro diplomskega dela in so predstavljene smernice za nadaljevanje raziskave na tem področju.

2. Opis arhitekture

2.1 Razlike in podobnosti med RISC in CISC

2.1.1 Splošno o CPE

Za razumevanje računalnika kot celote moramo poznati osnovne principe delovanja posameznih procesnih enot. V tem razdelku se bomo osredotočili na centralno procesno enoto, ali krajše CPE, ki je jedro slehernega računalniškega sistema. Poleg centralne enote imamo v računalnikih običajno tudi druge procesorje, ki skrbijo za izvrševanje specifičnih nalog. Sem spadajo predvsem vhodno/izhodni, krmilni in drugi procesorji. Arhitektura je zato v veliki meri določena z naborom in številom ukazov, ki jih procesna enota ponuja. Glede na število in kompleksnost ukazov sta se v načrtovanju računalniških sistemov in kar velja še posebej za področje razvoja mikroprocesorjev, ustvarili dve večji tehnološki veji razvoja - CISC (Complex Instruction Set Computer) in RISC (Reduced Instruction Set Computer). Kot pove ime, gre pri tipu RISC za procesor z zmanjšanim naborom instrukcij, pri drugem pa za procesor s kompleksnim naborom instrukcij. Kriterij za RISC ukaze je naslednji [3] :

- ukaz se mora izvršiti v eni urini periodi in
- operacije, ki jo ukaz izvaja, ni mogoče realizirati hitreje z zaporedjem drugih ukazov

Razlike med arhitekturami so močno pogojene s predstavitvijo podatkov znotraj procesorja. Najbolj razširjene so skladovna, akumulatorska in predstavitev z množico registrov. Operandi so lahko predstavljeni implicitno ali eksplicitno. Pri skladovnem stroju so operandi implicitno na vrhu sklada, v akumulatorski arhitekturi pa se en operand implicitno nahaja v akumulatorju. Arhitekture s splošnonamenskimi registri imajo samo eksplicitne operande, ki so bodisi registri ali pomnilniški naslovi. Do eksplicitnih operandov se lahko dostopa direktno v pomnilniku ali pa morajo biti najprej naloženi v poseben prostor, kar zavisi od tipa instrukcij. Primer izvedbe

Sklad	**Akumulator**	**Register-pomnilnik**	**Naloži-shrani**
push A	load A	load r1,A	load A,r1
push B	add B	add r1,B	load B,r2
add	store C	store r1,C	add r1,r2,r3
pop C			store r3,C

Tabela 1: Zaporedje instrukcij za izračun izraza A=B+C na različnih tipih arhitektur.

preprostega aritmetičnega izraza `A=B+C` na različnih arhitekturah prikazuje tabela 1. Iz tabele je razvidno, da v bistvu obstajata dve vrsti registrskih strojev. Eden lahko dostopa do pomnilnika v vsaki instrukciji (register-memory arhitektura), drugi pa lahko dostopa samo preko instrukcij za nalaganje in shranjevanje (load-store arhitektura). Tretja tovrstna arhitektura bi bila čisto pomnilniško orientirana in bi imela vse operande shranjene v pomnilniku.

Večina starejših strojev je temeljila na skladu ali akumulatorju, medtem ko danes obstajajo skoraj izključno računalniki s splošnonamenskimi registri. Registri se namreč nahajajo v notranjosti samega procesorja in so zato veliko hitreje dostopni, druga dobra lastnost pa je njihova vloga pri prevajanju. Izraz je npr. z uporabo registrov mogoče evaluirati v poljubnem vrstem redu, kar pride večkrat prav. Na skladovnem stroju poteka evaluacija običajno od leve proti desni, kar povzroča, posebej pri prevajanju, nepotrebne omejitve. Registri se lahko uporabljajo tudi za shranjevanje spremenljivk, če jih tako alociramo. Pri tem se občutno zmanjšajo pomnilniški prenosi, poveča se hitrost izvajanja in zmanjša dolžina kode, saj se registri navadno kodirajo z veliko manjšim številom bitov kot nek pomnilniški naslov. Število registrov, potrebnih za optimalno prevajanje, je odvisno od tega, kako jih prevajalnik porabi. Večina prevajalnikov si rezervira nekaj registrov za evaluacijo izrazov, nekaj za pošiljanje parametrov, ostali pa so na voljo za alokacijo spremenljivk.
Pomembna karakteristika je tudi število eksplicitnih operandov v ukazih in tip teh operandov. Večina današnjih računalnikov temelji na 2- ali 3-operandni arhitekturi.

Tip	Prednosti	Slabosti
Register-register	Preprosto kodiranje instrukcij fiksne dolžine in preprost model prevajanja. Vse instrukcije za izvedbo potrebujejo približno enako število ciklov.	Več instrukcij je potrebnih kot pri tistih z pomnilniški operandi.
Register-pomnilnik	Dostop do podatkov je omogočen brez prvotnega nalaganja. Format instrukcij je preprost in kompakten.	Operandi niso enaki kot tudi instrukcije niso fiksne dolžine. Čas izvajanja ukazov prav tako variira.
Pomnilnik-pomnilnik	Kompaktne instrukcije. Ni potrebe po registrih.	Dolžina instrukcij je zelo različna, posebno za 3-operandne verzije. Dostop do pomnilnika pomeni ozko grlo.

Tabela 2: Prednosti in slabosti treh najbolj razširjenih registrskih strojev.

Pri slednji se rezultat operacije nad dvema operandoma prenese v tretjega. Pri 2-operandni pa je en operand istočasno izvor in rezultat operacije. Pri tipu operandov je pomembno, koliko jih lahko referencira pomnilnik; to število se običajno giblje od 1 do 3. Prednosti in slabosti posameznih arhitektur prikazuje tabela 2.

Druga ključna lastnost je naslavljanje pomnilnika, ki se med arhitekturami krepko razlikuje. V uporabi sta dve predstavitvi podatkov - z majhnim in velikim koncem (little in big endian). Pri slednjem je naslov podatka naslov bita z največjo težo, medtem kot je pri predstavitvi z malim koncem ravno obratno. Povsem irelevantno je, katero predstavitev se uporabi; problem lahko nastane le pri prenosu podatkov med stroji z različnimi predstavitvami. Dostopanje do enega zloga običajno ni problematično, pri podatkih, daljših od tega pa se običajno zahteva da so poravnani (aligned) na določeno lokacijo. Podatek velikosti v zlogov na naslovu N je poravnan, če velja $N \ mod \ v \ = \ 0$. Pri neporavnanih podatkih se pojavijo komplikacije pri implementaciji, saj se pomnilniške lokacije običajno začnejo na 2- ali 4-zložnih naslovih. Za RISC računalnike je značilno, da zahtevajo poravnane podatke, medtem

ko npr. x86 arhitektura tega eksplicitno ne zahteva.
CISC arhitekture omogočajo veliko število različnih in kompleksnih naslavljanj pomnilnika. Med naslavljanja, ki se največ uporabljajo, spadajo [4]:

- registrsko (`move r1,r2`),
- takojšnje (`move r2,6`),
- z odmikom (`move r4,10(r5)`),
- registrsko posredno (`move r4,(r5)`),
- indeksno (`move r3,(r1+r2)`),
- direktno ali absolutno (`move r1,(100)`),
- pomnilniško posredno (`move r1,@(r2)`),
- avtomatsko inkrementalno in dekrementalno (`move r1,(r2)+`, `move r1,-(r2)`) in
- skalirano (`move r1,100(r3)[r6]`).

Veliko tipov naslavljanja lahko izjemno zmanjša število potrebnih instrukcij. Nekoliko slabša stran pa je, da za izvedbo običajno potrebujejo večje število urinih period.

Tipična moderna CISC procesna enota vsebuje:

- manjše število registrov (do 16),
- več tipov registrov, od katerih so nekateri dovoljeni samo za posamezne instrukcije,
- aritmetične operacije operirajo nad registri in pomnilniškimi operandi,

- instrukcije z dvema operandoma,
- $r_1 \leftarrow r_1 \bigoplus r_2$,
- veliko število naslavljanj,
- variabilno dolžino instrukcij, pri čemer ima tako operacijska koda kot operandi, spremenljivo dolžino in
- instrukcije s stranskimi učinki.

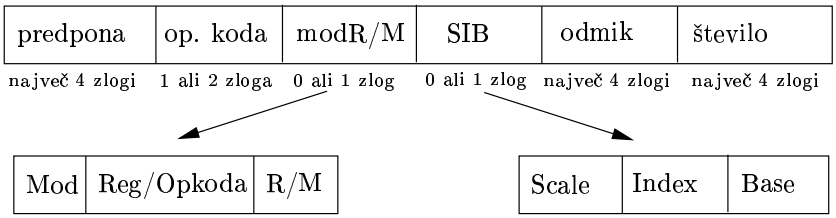
Arhitekturni tehnologiji sta očitno prav nasprotni, zaradi česar se je pojavila dilema, katera je boljša. Boljša je tista, ki je v dani domeni učinkovitejša, kar pa v veliki meri zavisi od kompleksnosti in narave samega problema, pa tudi drugih faktorjev, ki vplivajo na prilagojenost arhitekture za dotični problem.
Običajna RISC topologija procesorja ponuja:

- najmanj 32 registrov,
- samo en tip celoštevilčnih in realnih registrov,
- aritmetične operacije med registri,
- instrukcije za branje in pisanje v pomnilnik v obliki naslavljanja s konstantnim odmikom `Mem[reg+konst]`,
- instrukcije s tremi operandi v obliki $r_1 \leftarrow r_2 \bigoplus r_3$,
- vse instrukcije enake dolžine in
- natanko en rezultat na instrukcijo.

Razlog za variabilni format instrukcij je v tem, da se nekatere dajo zakodirati v 1 zlog, npr. premik iz enega registra v drugega, kjer ni potreben noben naslov, druge pa lahko vsebujejo zaradi kompleksnega naslavljanja naslove in odmike, kar bistveno poveča dolžino celotnega ukaza. Nefiksna dolžina pelje tudi do implementacijskih problemov.

Eden izmed teh je, da je pomnilniška hierarhija zasnovana z dolžinami besed, ki so potence števila 2. Nekatere daljše instrukcije bodo tako zasedale več kot eno pomnilniško besedo in bo za njihov prenos v procesor potreben več kot en poseg v pomnilnik. Drug problem se pojavi, kadar pride, pri uporabi virtualnega pomnilnika, do napake strani med samo instrukcijo, ker se ta zaradi svoje dolžine lahko nahaja na dveh straneh. Ta problem se običajno rešuje s prilagajanjem kode (code alignment).

Računalniki tipa CISC imajo tipično veliko število kompleksnih instrukcij, izmed katerih je običajno veliko takih, ki so razmeroma dolge in počasne tj. v pomnilniku zasedajo precej prostora, za izvedbo pa potrebujejo mnogo več kot 1 urin cikel. Nasprotno pa ima tip RISC razmeroma preproste in kratke instrukcije, ki izvršujejo natančno določene ukaze in zahtevajo malo število ali celo en sam urin cikel. Prav zaradi tega je za izvedbo nekega višjenivojskega konstrukta programskega jezika potrebno več RISC instrukcij, kot tistih iz CISC modela, zaradi česar se med programskim jezikom in strojnim nivojem poveča semantični prepad. V tem delu želimo ta prepad premostiti na ta način, da bomo približali strojni nivo višjemu programskemu jeziku. Abstrahirali bomo torej na strojnem nivoju.



Slika 1: Format instrukcije v Intel 586 procesorjih.

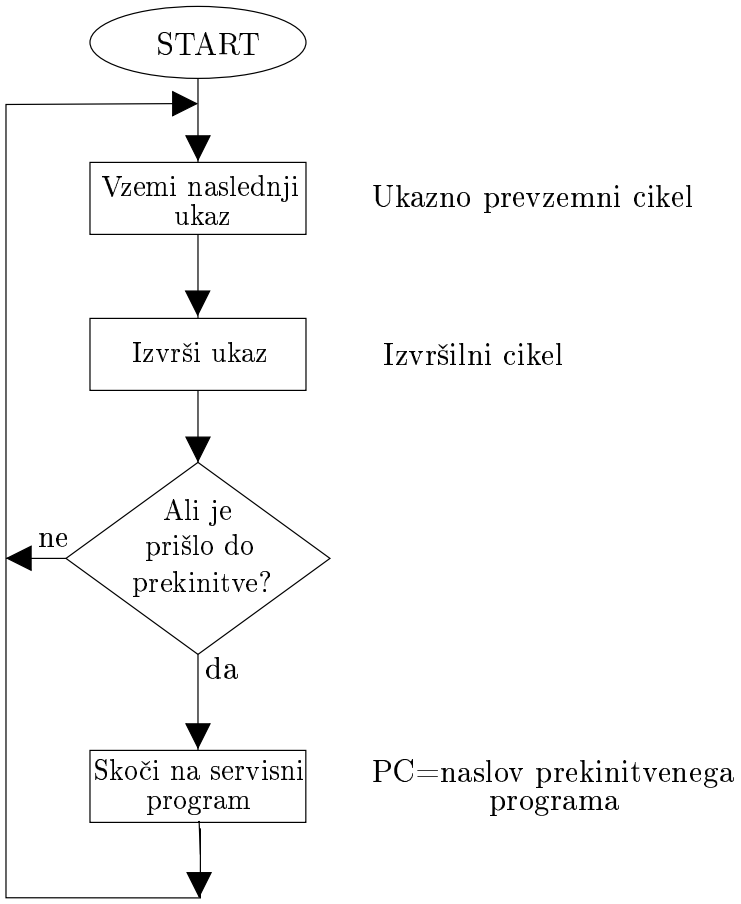
Razlike med hitrostjo in kompleksnostjo posameznih CPE-jev so lahko zelo velike: od najpreprostejših enot z nekaj sto tranzistorji, ki se uporabjajo zgolj za enostavna krmiljenja do pravih mikroprocesorjev z milijoni tranzistorjev na enem čipu. Kljub temu lahko delovanje vsake CPE opišemo s preprostim algoritmom v dveh korakih:

- **jemanje ukaza iz pomnilnika** ali **ukazno prevzemni cikel**, kjer se ukaz prenese v CPE s pomnilniške lokacije, na katero običajno kaže programski števec (register PC) in
- **izvrševanje ukaza** ali **izvršilni cikel**, kjer se ukaz dejansko izvrši [9]. Sestavljen je iz naslednji korakov:
  - dekodiranje ukaza,
  - prenos operandov iz pomnilnika v CPE,
  - izvedba operacije in
  - morebitno shranjevanje rezultatov nazaj v pomnilnik ali registre.

  Po izvedenem ukazu je v registru PC naslov naslednjega ukaza.

Obstaja več modelov osnovnega von Neumannovega računalnika, npr. Harvardska in Princetonska arhitektura, princip delovanja pa se razlikuje le malo. V opisanem algoritmu na sliki 2 se prva dva koraka ponavljata, dokler računalnik deluje. 4. korak se izvede samo v primeru, ko pride do strojne ali programske prekinitve. Takrat se izvajanje prenese na prekinitveno rutino in se po dovršitvi le-te ponovno vrne na prekinjeno mesto.
Tudi alternativne arhitekture, npr. tiste, ki podpirajo funkcijsko programiranje, niso nikoli v celoti nadomestile klasičnega von Neumannovega računalnika, ki je s posameznimi podrobnostmi danes evolviral do zelo različnih arhitektur, ki vključujejo [13]:

- paralelne računalnike,
- paralelizacijo na instrukcijskem nivoju,
- računalnike za pretok podatkov,
- asociativne arhitekture in
- nevronske mreže



Slika 2: Osnovni princip delovanja CPE v von Neumannovem računalniku.

Da bi računalnik popolnoma določili, potrebujemo še dodatne informacije. Tipe CPE-jev ločimo glede na število operandov v ukazih (1, 2, 3 ali več operandni), glede na možnost uporabe registrov (registrski), velikost pomnilniškega prostora, ki so ga sposobne naslavljati in dolžino pomnilniške besede.

2.1.2 Ukazi in njihova zgradba

Množica operacij, ki ji CPE izvaja, je v splošnem lahko zelo široka. Pomnilniški operandi so lahko dolgi 8, 16, 32 ali več bitov. V primeru daljših operandov gre običajno za večkratnike dolžine pomnilniške besede. Operandi so zapisani po pravilu "little endian"

ali “big endian” in so lahko poravnani ali pa ne. Ukazi se delijo na več skupin:

- ukazi za prenos podatkov, ki prenašajo operande iz pomnilnika v procesorjeve registre in obratno. Značilno za njih je, da uporabljajo več vrst naslavljanj,
- aritmetično logični ukazi, ki izvršujejo aritmetične in logične operacije. Običajno so 2 ali več operandni, ker po zaključku shranjujejo tudi rezultat,
- vejitveni ukazi, ki omogočajo skoke in
- kontrolni ter sistemski ukazi.

Dolžina ukazov je odvisna od tipa arhitekture in je lahko fiksna, ki je v skladu z RISC topologijo, ali variabilna ki je značilnejša za računalnike CISC. Vsak ukaz je sestavljen iz operacijske kode in operandov, kot to ponazarja slika 3. Operacijska koda pove, za kateri ukaz gre in vsebuje informacijo o naslavljanju ter formatu operandov, ki sledijo.

Slika 3: Zgradba ukaza.

Število ukazov je enolično določeno s številom različnih operacijskih kod, poleg tega pa lahko na operacijo vplivajo tudi posebni operandi npr. `mod reg/rm` zlog ali t.i. predponski zlogi pri 80x86 arhitekturi. Format instrukcije neposredno vpliva na dolžino prevedenega programa in samo implementacijo dekodirne enote v procesorju, ki je odgovorna za čimhitrejšo razpoznavo ukaza in pripadajočih operandov. Format je odvisen od števila različnih načinov naslavljanja in stopnje neodvisnosti med operacijsko kodo in posameznim načinom naslavljanja. Nekateri stroji imajo 3 ali 4 operande in mnogo naslavljanj za vsakega izmed njih, kar pomeni zelo veliko število različnih kombinacij. V takšnem primeru so v ukazu potrebna posebna polja; to so t.i. specifikatorji

naslova, ki določajo kateri tip naslavljanja je v ukazu uporabljen. Tako lahko prihranimo na številu različnih operacijskih kod, saj je operacija z različnimi tipi naslavljanj določena z eno samo kodo, medtem ko posebno polje v instrukciji pove, za katero naslavljanje gre in kakšni so operandi. Pri strojih s samo enim pomnilniškim operandom (load-store) in z enim ali dvema tipoma naslavljanj pa je lahko celoten ukaz, skupaj z načinom naslavljanja, zakodiran v sami operacijski kodi. Pri določanju formata instrukcij je potrebno upoštevati veliko, tudi nasprotujočih si, teženj. Zaželjeno je imeti čimveč registrov in načinov naslavljanja. Velikost registrskih operandov in specifikatorjev naslavljanj bistveno vpliva na dolžino ukaza in posledično na povprečno velikost programa. Poleg tega pa je zaželjeno, da se instrukcija da preprosto dekodirati in ni čisto poljubne, če že ni fiksne dolžine. Do neugodnih dolžin pridemo tudi pri takojšnjih operandih, kjer so vrednosti podane že v samem ukazu. Če arhitektura uporablja 32 bitne podatkovne tipe, kar je danes zelo pogosto, bi se v operandih večinoma pojavljale 32 bitne vrednosti. Ker pa je veliko števil v praksi manjših, se splača narediti ukaze, ki bodo sprejemali tudi krajša števila, npr. 16 in 8 bitna. Takojšnji operandi se velikokrat uporabljajo neposredno pri naslavljanju z odmikom in ker veliko arhitektur uporablja splošna naslavljanja, pride do nepotrebnih izgub prostora. Odmik v naslavljanju je običajno število dolžine registrov. Če se uporablja splošen tip naslavljanja, potem bo v primeru ničelnega odmika v instrukciji še vedno zakodirana ničla, ki bo večala njeno dolžino, na ukaz pa ne bo imela nikakršnega vpliva.

Tip posameznega operanda je mogoče določiti na dva načina. Informacija je lahko zakodirana že v operacijski kodi, kar se najpogosteje uporablja. Drug način pa je uporaba posebnih značk, ki se interpretirajo strojno, vendar se ta način danes ne uporablja več. Tip operanda se navadno giblje od dolžine 1 do 8 zlogov. Najznačilnejši tipi so znak (1 zlog), polovična beseda (2 zloga), beseda (4 zlogi), realni tip z enojno natačnostjo (4 zlogi) in realni tip z dvojno natančnostjo (8 zlogov). Pri številih s plavajočo vejico se danes skoraj izključno uporablja standard IEEE 754.

2.1.3 Mikroprogram

Direktna implementacija logičnih funkcij v obliki enačb je zelo težavna in pri današnjih procesorjih praktično nemogoča. Povrhu tega bi vsaka kasnejša sprememba povzročila fizičen poseg v digitalna vezja. V ta namen se uvede pojem "mikroprogramiranja". Mikroprogram ali kontrolni program vsebuje mikroukaze, ki se, podobno kot pri CPE, prenašajo iz mikroprogramskega pomnilnika [9]. Mikroukazi so elementarni in atomarni ukazi mikroprogramske enote, ki natančno opisujejo delovanje vsakega ukaza CPE. Pri mikroprogramiranju ima sleherni ukaz torej svoj ekvivalent v množici mikroukazov s čimer načrtovanje kontrolne enote postane pravzaprav pisanje mikroprograma za vsako instrukcijo.

Mikroprogram torej deluje kot vmesna plast med fizičnim nivojem, tj. nivo digitalnih vezij in strojnimi ukazi CPE-ja, kot jih vidijo programerji. Mikroukazi so zaradi tega veliko primitivnejši in preprostejši od običajnih strojnih ukazov. Mikroprogram je v procesorju običajno trdoožičen in je zato vedno prisoten. Ob vklopu se mikroprogram inicializira, nato pa začne z izvajanjem procesorskih ukazov. Vsak ukaz, ki prispe na vhod, se izvede z interpretiranjem preprostih mikroprogramskih ukazov, ki se jemljejo iz pomnilnika. Ker se ukazi v tem pomnilniku ne spreminjajo, je ta običajno realiziran kot bralni pomnilnik (ROM). Mikroprogram za trenutno lokacijo in določanje naslednje uporablja mikroprogramski števec, ki je po vlogi enak programskemu števcu.

Mikroprogram ima enako kot procesor svoj nabor ukazov, od katerih jih je večina za premikanje podatkov in za skoke. Instrukcije za premikanje lahko prenašajo podatke med katerimikoli registri, tistimi za dejansko uporabo na procesorju in mikroprogramskimi, medtem ko vejitvene omogočajo poljubno premikanje po interpreterju.

Format mikroinstrukcije je podoben tistemu za klasično instrukcijo, kot jo vidi programer. Mikroinstrukcij je veliko manj kot pravih ukazov, zato bo število potrebnih bitov za operacijsko kodo veliko manjše. Vsebovani morata biti vsaj še polji za izvor in cilj, običajno pa je dodan še tip operacije.

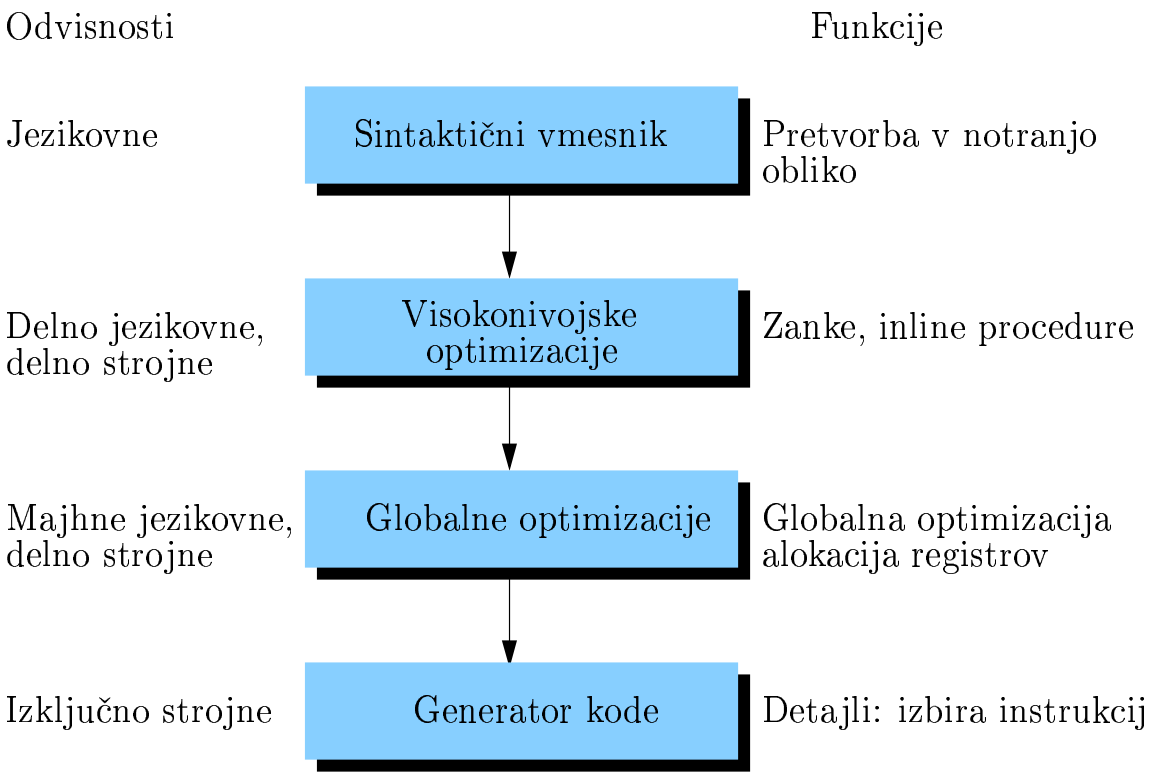
3. Nova arhitektura

Nove strojne arhitekture se lotevamo z namenom zmanjšati semantično razliko med visokonivojskim programskim jezikom in strojnim oz. zbirniškim nivojem. S tem bo poenostavljen postopek prevajanja v strojni kod, veliko pa pridobimo tudi pri procesu optimizacije. Osnovna ideja se prezentira v tem, da visokonivojskim programskim konstruktom definiramo ekvivalente na strojnem nivoju. Semantični model se na ta način spremeni le v nadrobnostih, osnova ostane enaka ali vsaj zelo podobna. Razlike ostanejo, kot v vseh primerih pretvorbe v strojni ali zbirni jezik, na sintaktični ravni in za sam postopek prevajanja niso bistvene.

Z večanjem števila ukazov na nekaterih arhitekturah se kljub drugačnim pričakovanjem stvari niso kaj dosti spremenile, saj pri prevajanju v strojni jezik še zmeraj prevladujejo razmeroma preprosti ukazi in načini naslavljanja. Kompleksni ukazi, ki so dolgi in dokaj počasni, so namreč preveč posplošeni in navadno opravijo več opravil, tudi nepotrebnih, kot bi si želeli. Da bi preprečili ta paradoks, je potrebno operacije, ki jih ukazi izvajajo, bolje dodelati. Operacije naj ostanejo kompleksne, vendar naj opravljajo povsem specifične naloge za posamezne domene.

Drug problem je realizacija kompleksnih ukazov, ki je v primeru trdo ožičene logike zelo zahtevna. To je razlog, zakaj se pri takšnih arhitekturah raje uporablja mikroprogramiranje. Idealna varianta bi bila torej kompleksnost ukazov izvedena s trdo ožičeno logiko, kar pa je težko dosegljivo.

Prvi cilj prevajanja je eksaktnost – vsi pravilni programi se morajo pravilno prevesti. Drugi cilj je nedvomno hitrost prevedene kode, sledi pa podpora razhroščevanju, hitro prevajanje in povezovanje z drugimi jeziki. Prevajalniki običajno opravijo več prehodov za posamezno prevajanje, pri čemer se v vsakem prehodu opravijo določene naloge. Prevajalnik mora nekatere faze poznati vnaprej; telo metode se npr. pri *inline* (vrinjene metode) ključih ekspanzira, preden je poznana velikost kode metode. Optimizacije lahko klasificiramo takole [3]:



Slika 4: Operacije sodobnega prevajalnika.

- visokonivojske optimizacije na izvornem nivoju, ki služijo za vhod naslednjim prehodom,
- lokalne optimizacije operirajo nad posameznimi fragmeni izvorne kode, npr. izrazi,
- globalne optimizacije razširijo lokalne na širše aspekte, npr. celotni bloki ali zanke,
- alokacija registrov in
- strojno odvisne optimizacije se odražajo na specifičnosti arhitekture.

Eden izmed najpomembnejših korakov optimizacije je alokacija procesorskih registrov. Algoritem alokacije je implementiran na podlagi "barvanja grafov", katerega osnovna

ideja je v tem, da konstruiramo graf, ki predstavlja registre, določene za alokacijo in ta graf uporabimo [10]. Čeprav je barvanje grafa NP-poln problem, obstajajo hevristične metode, ki so v praksi zelo učinkovite.

Barvanje deluje dobro, če obstaja vsaj 16 splošnih registrov za celoštevilčne tipe in še nekaj dodatnih registrov za realne tipe. V kolikor je število registrov manjše, barvanje ne deluje dobro, ker bo hevristika v algoritmu običajno zatajila (ne najde barvanja).

Alokacija registrov je učinkovitejša za objekte alocirane na skladu, kot za globalne spremenljivke, za objekte alocirane v pomnilniku pa je praktično nemogoča. Za globalne in skladovne spremenljivke običajno obstaja več "imen" in zato tudi več načinov za dostopanje do njih, kar pomeni, da bi lahko prišlo do nepravilnega dostopanja, če bi jih umestili v registre.

V kolikor želimo zagotoviti kvalitetno podporo objektno usmerjenim programskim jezikom, moramo omogočiti nekompleksno ali celo linearno pretvorbo abstraktnih podatkovnih in semantičnih struktur v nizkonivojske ekvivalente strukture na zbirnem oz. strojnem nivoju. Podobno kot lahko vsak ukaz modeliramo z ustreznim mikroprogramom ali trdoožičenim digitalnim vezjem, lahko na enak način storimo z bogatejšo strukturo.

Abstrakcija podatkov razširja pomen modularnega programiranja v definicijo povsem novega tipa, v katerem so zbrane procedure, ki operirajo nad tem tipom podatka. Tip takega podatka je abstraktni tip in je osnova vseh objektno usmerjenih programskih jezikov. Abstraktni tipi so povsem enakovredni primitivnim tipom in ločijo koncept od implementacije. Pomembna je samo deklaracija tipa, podrobnosti nas ne zanimajo.

Pojmi, ki jih je potrebno upoštevati pri grajenju objektno usmerjenih jezikov, so:

- Nastanek spremenljivk z alokacijo pomnilniškega prostora, ko jih definiramo na nakem mestu v programskem bloku in njihovo uničenje na mestu, ko se blok zaključi. Vrednost se lahko pri tem inicializira na neko določeno vrednost ali ostane nedefinirana. Ker objekt pred definicijo še ne obstaja, mora biti preko klica posebne metode (konstruktor) omogočeno, da se razred instancira v objekt.

- Operatorska in proceduralna mnogoličnost, ki omogočata enaka imena metod in uporabo operatorjev nad različnimi podatkovnimi tipi, tako primitivnimi kot abstraktnimi.

- Dedovanje omogoča razširjanje funkcionalnosti posameznih objektov. Je osnova pri modeliranju "od splošnega k specifičnemu".

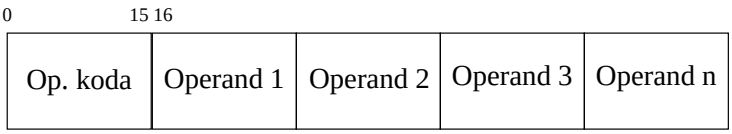
- Enkapsulacija ali ograjevanje omogoča kontrolo nad dostopom do objektovih podatkov in poznavanjem detajlov nekega objekta. S pomočjo mehanizma enkapsulacije je dosežena tudi varnost v tem smislu, da lahko nadzorujemo sleherno spremembo objektovega stanja, saj se te izvajajo izključno preko metod.

- Avtomatsko čiščenje pomnilnika je zelo pomembno pri ustvarjanju velikega števila objektov, kot je pri objektno usmerjenih jezikih običajno, saj nam na ta način ni potrebno skrbeti za manualno sproščanje pomnilniškega prostora. Posledica, da objekti zasedajo različne velikosti pomnilnika, je razdrobljenost oz. fragmentacija pomnilniškega prostora, zaradi česar postopno pride do "lukenj", ot objektov nihče več ne uporablja.

Posamezni konstrukt bomo predstavili z razredom, ki vsebuje podatke v obliki spremenljivk in metode, ki operirajo nad temi podatki, pri čemer bodo ohranjeni osnovni koncepti objektnega programiranja. Razred ali enota procesorja, predstavlja šablono, ki z instanciranjem postane objekt v pomnilniku. Definirane metode znotraj razreda pa se na strojnem nivoju preslikajo v ukaze CPE-ja. Kot je na običajnih procesorjih vsaka operacija realizirana z logično funkcijo v obliki digitalnega vezja, bo sleherni podatkovni ali kontrolni konstrukt programskega jezika prav tako imel svojo realizacijo s specialnim vezjem v procesorju.

Prva stvar, ki se je bomo lotili podrobneje, je zgradba procesorskih ukazov. Realizacija le-teh je nepomembna in se z njo ne bomo posebej ukvarjali; lahko je izvedena z mikroprogramiranjem ali trdo ožičeno logiko. Vsak ukaz vsebuje informacijo o operaciji, ki se imenuje operacijska koda in informacijo o operandih, na katerih se operacija

izvrši. Razdeljen je na več polj, izmed katerih ima vsako svoj pomen, saj lahko nosi operacijsko kodo ali operand. Takšna zgradba se imenuje format ukaza.
Zaradi preprostosti dekodiranja bo operacijska koda v našem primeru vsebovana vedno v prvih dveh zlogih ukaza, iz česar logično sledi, da bo dolga 2 zloga ali 16 bitov, kar nam omogoča 2^{16} različnih kod. Razlog za takšno dolžino je v možnosti razširitve, pri kateri ne bo potrebno spreminjati obstoječega dekodirnika, temveč bo dovolj ukaze z novimi operacijskimi kodami preprosto dodati. Ker bo za izvršitev posamezne metode potrebna še referenca na objekt v pomnilniku, bomo za vsako nestatično metodo potrebovali še kazalec na objekt. Po dogovoru bo kazalec dolg 32 bitov za 32-bitno ciljno arhitekturo in bo sledil neposredno operacijski kodi. Statične metode seveda ne vsebujejo referenc. Operacijski kodi in kazalcu objekta sledijo opcionalni operandi poljubne dolžine. Zgradbo ukaza prikazuje slika 5.



Slika 5: Zgradba ukaza s 16 bitno operacijsko kodo in 32-bitno referenco. Dolžina ostalih operandov je lahko 8, 16, 32 ali 64 bitov.

Operandi so lahko registri, pomnilniške lokacije ali literali v primeru takojšnjega naslavljanja, s čimer se arhitektura uvršča med registrsko-pomnilniške računalnike. Število eksplicitnih operandov je lahko povsem poljubno, kar pogojuje veliko število različnih formatov ukaza. Maksimalno možno število registrov je 256, kar pomeni, da lahko register opišemo z 1 zlogom. Dolžina posameznega registra je 32 bitov in je enaka za vse registre. Množica registrov je lahko globalna ali definirana znotraj razreda kot statična spremenljivka, ki se sklicuje na že definirani register. S tem dosežemo večjo lokalnost in preimenovanje registrov – eden izmed atributov večine RISC arhitektur je namreč v tem, da se nekateri posebni registri preslikajo na navadne registre z drugimi imeni.

Splošnonamenski registri so označeni $r1,$r2, $r3 itd.

3.1 Definicija razredov

Vse procesorske enote so formalno definirane z razredi, ki vsebujejo metode in pripadajoča polja. S tem je zadovoljeno osnovnim zahtevam zaprtosti in skrivanja za uporabnika nepotrebnih informacij. Implementacija razreda, ki pomeni implementacijo razrednih metod, je izvedena na strojnem nivoju preko digitalnih vezij, bodisi kombinatoričnih ali pomnilniških. Vse metode posameznega objekta opisujejo t.i. sporočilni vmesnik oz. sporočilni protokol, če prevedemo klic vsake metode v ustrezno sporočilo objektu. Za kateri objekt gre, je razvidno iz sproročila, ki je sestavljeno iz imena metode in ciljnega objekta, tj. objekta, nad katerim želimo metodo izvršiti. Dostop do razreda je specificiran s ključem dostopa, ki lahko zavzame vrednost *public* ali *private*; pri slednjem je dostop onemogočen.
Prav tako imajo ključ dostopa metode in polja znotraj razreda, s to razliko, da je ta lahko še *protected*. Vsi specifikatorji dostopa se obnašajo "standardno", kar pomeni, da je njihov pomen enak kot pri programskem jeziku C++ ali Javi. Deklaracija metode sestoji iz operacijske kode in množice operandov, ki se sklicujejo na seznam formalnih parametrov; izjema je le operand *this*, ki se lahko uporablja brez definicije v parametrih. Ker je operandov lahko več in so lahko kjerkoli, mora imeti vsak natančno določen položaj znotraj instrukcije. Definicija operandov je izvedena z rezervirano besedo *operand*, pri kateri v oglatih oklepajih povemo številko 32-bitne besede in odmik znotraj te besede.

Primer definicije razreda "alu":

```
public class alu{
       static public method add(register r1,register r2){
              opcode=0x3c;
              operand[0,2]=r1; operand[0,3]=r2;
       }
```

```
        static public method add(register r1,oct4 o){
               opcode=0x3d;
               operand[0,2]=r1;  operand[0,3]=o;

        }
        static public method sub(register r1,register r2){
               opcode=44;
               operand[0,2]=r1;  operand[0,3]=r2;
        }
        static public method sub(register r1,oct4 o){
               opcode=45;
               operand[0,2]=r1;  operand[0,3]=o;
        }
        static public register field flags=r5;
}
```

Metode razreda definiramo z rezervirano besedo *method*. Podati je treba seznam formalnih parametrov, ki je lahko prazen, in telo metode. Seznam parametrov je sestavljen iz, med seboj z vejico ločenih, dvojčkov formata *tip ime_parametra*, kjer se na *ime_parametra* sklicujemo v definiciji operandov. Metode so lahko statične, takrat razreda ni potrebno instancirati za njen klic; nasprotno, pa se lahko nestatična metoda izvrši samo na obstoječem objektu. Klic metode je mogoč samo, če je metoda deklarirana z javnim ključem (*public*). Če je zaščitena (*protected*), je iz razreda, v katerem je definirana, ni mogoče klicati, lahko pa jo naredimo javno v izpeljanem razredu.
Stanja razreda definiramo z besedo *field*, kateri podamo še ključ dostopa in dosega, ime in inicializacijo. Ta je pri statičnih poljih potrebna za sklicevanje v operandih, pri ostalih pa za postavitev vrednosti v pomnilniku pri ustvarjanju novega objekta tega tipa. Tip nestatičnega polja je lahko `oct1, oct2, oct4` ali `oct8`, pri statičnem polju pa je poleg že omenjenih možen še tip `register`.

3.2 Definicija registrov

Register definiramo z rezervirano besedo *register*, kateri sledi ime registra in število, ki ta register predstavlja. Naenkrat lahko definiramo več registrov, enega za drugim,

pri čemer so ti ločeni z vejico. Primer definicije registra je `register r1=1;`, definicije več registrov pa `register r1=1,r2=2,pc=3,sp=4;`. Register lahko definiramo z že obstoječim registrom tako, da mu priredimo njegovo ime. To je način preimenovanja; nekateri registri se reflektirajo na drugih in imajo različna imena. Primer takšnega načrtovanja je naslednji:

```
// definiramo registre r0,r1 in r2
register r0=0,r1=1,r2=2;

// definiramo prvi pc
register pc1=r0;
// drugi pc
register pc2=r1;

// skladovni števec
register sptr=r2;
```

Na definirane registre se sklicujemo pri poljih znotraj razreda in v operandih pri samem programiranju. Ime registra se začne z znakom '$'.

3.3 Sintaksa zbirnega jezika in model programiranja

Za pisanje programske kode je implementiran zbirni jezik. Namenjen je predvsem testiranju razvite arhitekture, lahko pa se uporabi tudi za pisanje večjih programov in posodabljanje obstoječe arhitekture. Zbirnik je napisan dokaj fleksibilno z možnostjo razširitev in dodajanja novih konstruktov. Prevajalnik se nahaja v obliki izvornega programa, ki na vhodu sprejme datoteko z definicijo arhitekture in ukazi, jih obdela in na izhod pošlje preveden program v obliki zlogovne kode, ki je v takšni obliki primeren za izvajanje na virtualnem stroju dotične arhitekture.

Zbirnik omogoča, zaradi večje preglednosti, vključevanje definicijskih datotek in datotek izvorne kode. V spošnem je sintaktično zelo preprost – program namreč sestavljata le dve sekciji, ena za podatke in druga za kodo. Podatkovna sekcija, umeščena med rezervirani besedi `.data` in `.end`, lahko vsebuje spremenljivke primitivnih ali abstraktnih

tipov.

Primer podatkovne sekcije:

```
.data
   myvar1 is oct1
   myvar2 is oct2
   myvar3 is integer
   myvar4 is string
.end
```

Deklaracija spremenljivke sestoji iz imena spremenljivke, kateri sledi rezervirana beseda *is*, in tipa.

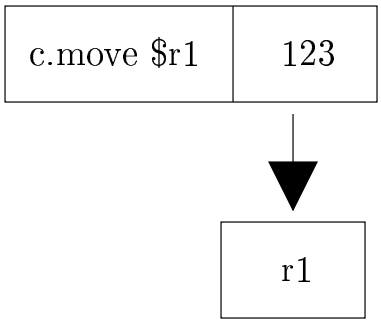
Analogno je sekcija kode oz. "tekstovna" sekcija umeščena med blok `.text` in `.end`. Ukaz v tekstovni je sestavljen iz imena razreda in metode, ki jo želimo izvesti nad objektom tipa razreda. Temu sledijo opcijski operandi, ki so lahko registri, naslovi ali literali. V eni vrstici se lahko nahaja samo en ukaz; vsak ukaz se namreč zaključi s skokom v novo vrstico. V kolikor želimo uporabiti oznako, mora biti ta prav tako v svoji vrstici in se mora končati s dvopičjem. Na naslove se v operandih sklicujemo z imeni spremenljivk ali oznak.

Primer tekstovne sekcije:

```
.text
   alu.add $r7, $r6    // v alu kličemo metodo add nad r7 in r6
   alu.sub $r2, $r7, 123
.end
```

Obstaja več tipov naslavljanj oz. podajanj operandov. Najpreprostejše je takojšnje naslavljanje (immediate addressing), kjer operand določimo tako, da ga v ukazu podamo kar z vrednostjo. Pri tem načinu je operand že del ukaza in se skupaj z njim prenese v CPE. Operandi, podani na ta način, se imenujejo takojšnji operandi ali literali. Primer takojšnjega naslavljanja (slika 6) je ukaz `c.move $r1,123`, kjer se konstanta 123 prenese v register `r1`. Celoštevilčni literali se vnašajo v desetiški, šestnajstiški ali osmiški obliki. Literali so lahko še števila v plavajoči vejici ali znaki. Podobno takojšnjemu naslavljanju je registrsko naslavljanje, kjer so vsi operandi

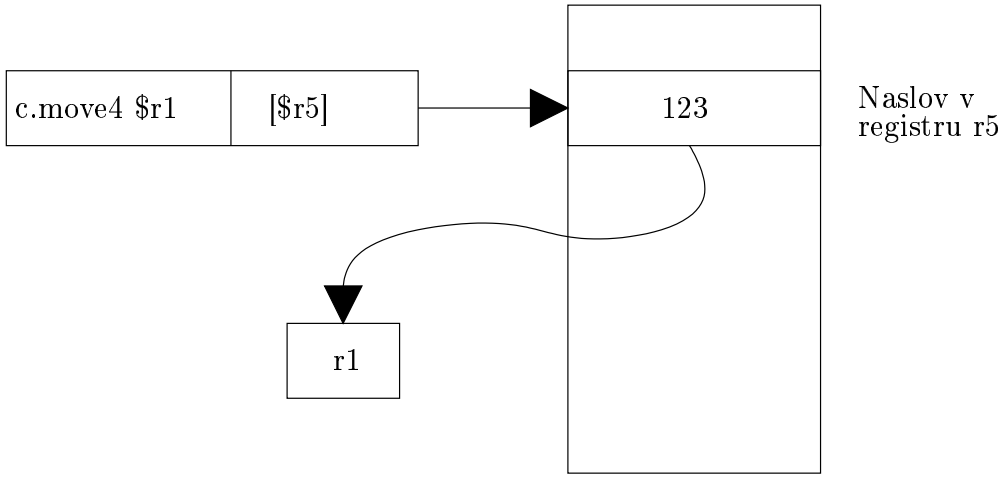
registri in so zato prav tako podani s številko.



Slika 6: Ukaz z takojšnjim naslavljanjem: v register r1 se prenese oz. naloži vrednost 123.

Drug tip naslavljanja je neposredno naslavljanje, kjer je operand v ukazu podan z naslovom. Naslov se ne spreminja, ampak ostaja statičen del ukaza, zato se imenuje tudi absolutni naslov. Primer na sliki 7, je `c.move4 $r1,var1`, ki prenese 4 zloge iz pomnilniškega naslova, podanega s spremenljivko `var1`, v register `r1`. Poudariti je treba, da je neposrednih naslavljanj več vrst, saj lahko naslov podamo na več različnih načinov. Lahko ga navedemo absolutno z literalom, kot je bilo v primeru tokojšnjega naslavljanja, po drugi strani pa ga lahko podamo z registrom. Če se odločimo za zadnjo možnost, je naslov podan v registru: `c.move4 $r1,[$r5]` in bi pomenilo prenos 4 zlogov z lokacije, na katero kaže register `r5`, v register `r1`. Register `r5` se tukaj imenuje bazni register.

Tretji tip naslavljanja je bazno-indeksno naslavljanje, ki je zelo podobno neposrednemu v tem oziru, da tudi tukaj uporabljamo nek bazni register. Dodan je le še indeks oz. odmik od naslova, podanega v baznem registru; v kolikor je ta enak 0, preide bazno-indeksno v bazno naslavljanje. Odmik je v splošnem lahko pozitiven, pri čemer se sklicujemo na naslov, ki je za odmik večji od baznega naslova ali negativen, ki



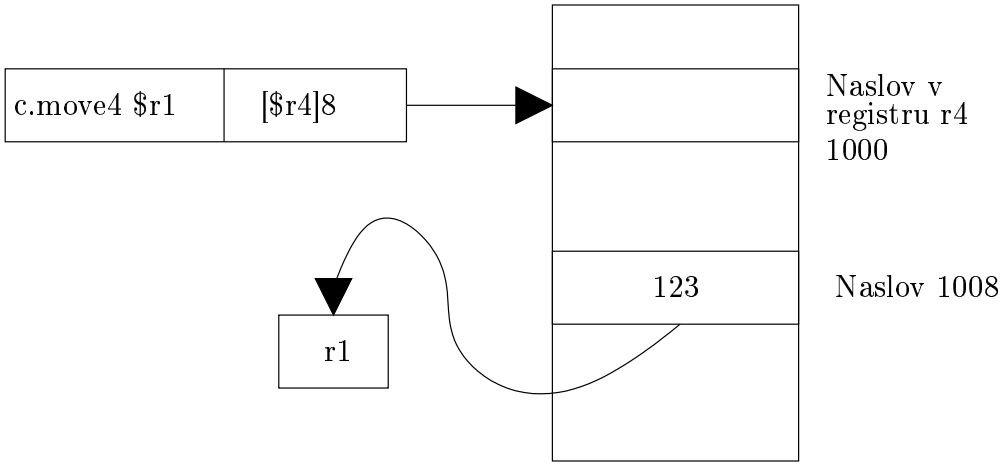
Slika 7: Neposredno naslavljanje. V register r1 se prenese vrednost z lokacije, podane z registrom r5. Lokacija bi bila lahko podana tudi s spremenljivko.

pomeni naslov za odmik manjši od baznega. Primer takšnega naslavljanja je `c.move4 $r1, [r4]8`, ki pomeni, da se z naslova, podanega v registru `r4` + 8, prensesejo 4 zlogi v register `r1` (slika 8).

Tipi operandov se delijo na primitivne in kompleksne. Primitivni tipi so vgrajeni in se lahko uporabljajo neozirajoč se na specifičnosti arhitekture. Definirani so neodvisno od ciljne arhitekture in niso vezani na noben programski jezik. Najmanjša naslovljiva enota je 1 oktet ali 8 bitov, kar pomeni, da je velikost posameznega tipa nek večkratnik enega okteta. Osnovne tipe in njihove dolžine prikazuje tabela 3.

Tipi `oct1, oct2, oct4` in `oct8` se uporabljajo v operandih za takojšnje naslavljanje preko literalov in deklaracije spremenljivk. Tip `register` se uporablja izključno v definiciji razrednih metod v seznamu parametrov in določa, da mora biti posamezen operand register.

Tip `address` se uporablja za operande, ki morajo biti pomnilniški naslovi v obliki spremenljivke. Tipa `base` in `basei` sta uporabljena prav tako za določanje naslova



Slika 8: Primer baznega naslavljanja, kjer se v register r1 prenese vrednost s pomnilniške lokacije, ki je določena z registrom r4 in pozitivnim odmikom 8. Če register r4 vsebuje vrednost 1000, se končna vrednost nahaja na naslovu 1008.

pomnilnika z uporabo baznega in indeksno-baznega naslavljanja. Poleg teh osnovnih tipov so možni seveda še abstraktni tipi, ki so definirani in implementirani v okviru specifične arhitekture. Dolžina vseh abstraktnih tipov je 32 bitov.

V datoteko s programom je mogoče vključiti tudi druge datoteke in na ta način razbiti prevelik del v manjše. Običajno so enote definirane vsaka v svoji datoteke, prav tako

Tip	**Velikost**	**Razred**
oct1	1 zlog	primitiven
oct2	2 zloga	primitiven
oct4	4 zlogi	primitiven
address	4 zlogi	primitiven
integer	4 zlogi	sestavljen
short	2 zloga	sestavljen
byte	1 zlog	sestavljen
string	n zlogov	sestavljen

Tabela 3: Primitivni in sestavljeni tipi.

tudi registri. Datoteko vključimo z rezervirano besedo *import*, kateri sledi ime datoteke. V izvorni kodi lahko uporabljamo enovrstične C++ komentarje, pri katerih je komentar desno od znakov // vse do konca vrstice. Druga možnost je uporaba večvrstičnih komentarjev, ki so umeščeni med znake /* in */.

3.4 Primeri kode

Za lažje razumevanje je spodaj navedenih nekaj preprostih primerov zbirniških stavkov.

Definicije spremenljivk:

```
.data
  var1 is oct1    // spremenljivka tipa oct1
  var2 is oct2    // oct2
  var3 is oct4    // oct4

  vari is integer // kompleksni tip
.end
```

Uporaba različnih tipov naslavljanja:

```
.text
  c.loct1 r6, var1    // naložimo var1
  c.move r6, r7       // damo v r7
  c.laddr r8, var2    // naslov var2
  c.loct4 r6,[r8]2    // naložimo z naslova var2 + 2 (var3)

  vari.new r6         // nov integer
  vari.neg            // negiramo
.end
```

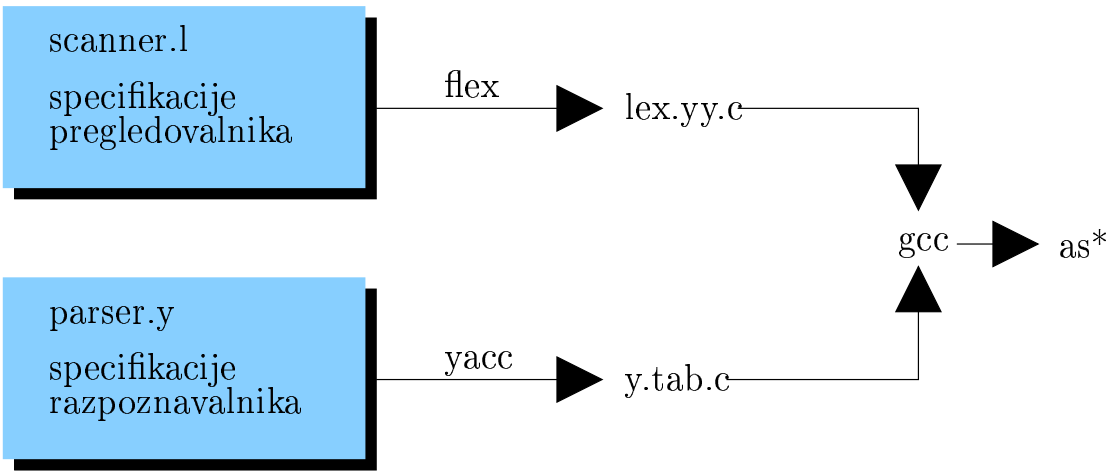
Klici funkcij:

```
.text
  c.call myfunc
  c.move r8,r9   // rezultat v r8

myfunc:
  c.add r10,r12
```

```
  c.move r10,r8
  c.back

.end
```

3.5 Implementacija zbirnika

Zbirnik je implementiran v jeziku C++, z razvojnimi orodji za konstrukcijo pregledovalnikov in razpoznavalnikov *Flex* in *Yacc* [11]. Orodje Flex po specifikacijah generira ustrezen pregledovalnik, ki na izhod pošilja terminalne simbole, Yacc pa te terminale uporablja za razpoznavanje sintakse. Na vhodu sprejme abstraktno sintakso v obliki BNF in na izhodu generira izvorno kodo ustreznega LALR(1) razpoznavalnika. Po konceptih razširljivosti je celoten zbirnik napisan modularno, kar omogoča preprosto dodajanje povsem novih in ažuriranje že obstoječih modulov. Generiranje zbirnika prikazuje slika 9.

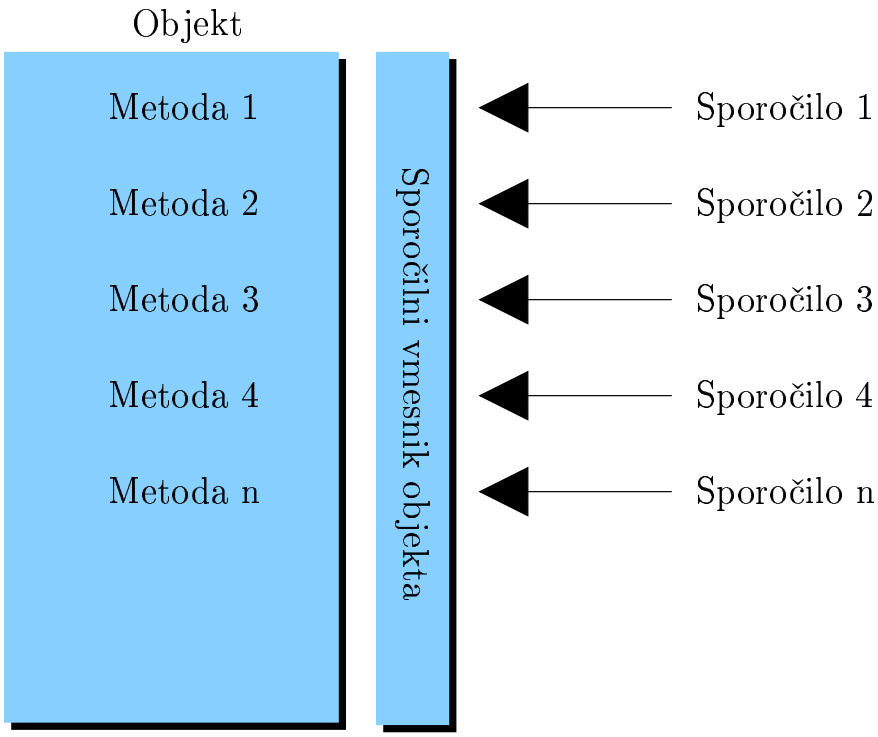


Slika 9: Tvorba zbirnika.

Vsi uporabljeni moduli so napisani v obliki razredov.

3.6 Abstraktni tipi

Z namenom linearizirati ali vsaj poenostaviti preslikavo tipov visokonivojskega programskega jezika na strojni nivo bomo v arhitekturo vpeljali abstraktne tipe, ki se najpogosteje uporabljajo. Abstraktni tipi so sestavljeni iz podatkov in pripadajočih metod, ki te podatke manipulirajo preko sporočilnega vmesnika definiranega z razredom. Klic posamezne metode se v tej terminologiji prelevi v pošiljanje sporočila objektu.



Slika 10: Sporočila.

Proces pošiljanja se mora izvesti atomarno, brez prekinitev, s čimer je zagotovljeno, da objekt sporočilo zares prejme. Ko objekt še ne obstaja, ga je najprej potrebno ustvariti. To storimo preko konstruktorske metode, ki se kliče s posebnim sporočilom. Sporočilo je sestavljeno iz imena sporočila in parametrov, nad katerimi se naj pokliče. Ime sporočila se na implementacijski ravni zaradi kompaktnosti pretvori v indeks v

tabelo sporočil. Realizacij za sporočanje objektom je več. Najočitnejša bi bila, kot je to izvedeno pri večini jezikov, da se parametri najprej shranijo na sklad, nato pa se nad vrednostmi na skladu pokliče ustrezna metoda. Pri tem načinu ni na zbirnem nivoju nobenega sledu več o objektni abstrakciji; vsi principi kapsulacije in skrivanja informacij se izgubijo.

V našem primeru bo referenca na objekt v pomnilniku zakodirana v sam ukaz procesorja. Referenca se pri definiciji nestatične metode poda s parametrom "this" in po dogovoru sledi operacijski kodi. Dolžina reference objekta je 32 bitov. Na enak način so podani vsi preostali parametri.

Primer simboličnega ukaza in njegovega zakodiranega formata:

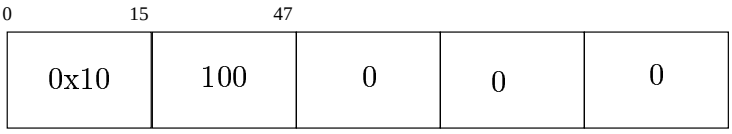
```
.data
  myint1  is integer    // myint1 je spremenljivka tipa integer
.end

.text
  myint1.new       // ustvarimo novo
  myint1.add 22    // prištejemo 22
.end
```

Naj bosta metodi v razredu `integer` definirani na naslednji način:

```
public method new(){
       opcode=0x10;
       operand[0,2]=this;
}

public method add(oct4 o){
       opcode=0x11;
       operand[0,2]=this;  operand[1,2]=o;
}
```

Če je naslov objekta `myint1` npr. 100, se prvi ukaz prevede takole:

```
myint1.new
```



Slika 11: Struktura prevedenega ukaza.

Abstraktni tip se ustvari s konstruktorjem, ki se imenuje "new". Ta poskrbi za alokacijo potrebnega prostora v pomnilniku in inicializira stanje objekta na zahtevano vrednost. Po tem se lahko kličejo metode, ki zavisijo od stanja objekta. Konstruktorjev je v splošnem lahko več; glede na tip parametra pa se pokliče pravi.
Implementirani so osnovni podatkovni tipi *integer, short, byte, char, float* in *string*; njihove metode prikazujejo pripadajoči tabeli 8 in 9.

3.7 Ostale enote procesorja

Za kar največjo učinkovitost so poleg enot abstraktnih tipov implementirane še "klasične" enote. Najobsežnejši med njimi sta ALE in podatkovna enota, ki sta definirani z razredom `core` ali krajše `c`. Vse metode znotraj razreda `c` so deklarirane statično, kar pomeni, da ni potrebno ustvarjati objekta, če jih želimo izvajati. Metode razreda `c` prikazuje tabela 4.
Množico aritmetičnih opracij sestavljajo funkcije za seštevanje, odštevanje, množenje in deljenje. Pri slednjih je treba paziti na rezultate saj npr. množenje dveh 16 bitnih števil v splošnem da rezultat z 32 bitno dolžino. Če imamo 32 bitne registre in množimo 16 bitna števila, bo torej cel rezultat lahko shranjen v registru, v primeru dveh 32 bitnih števil, pa bo rezultat zahteval 2 registra ali pa ovržbo zgornjih 32 bitov. Aritmetične operacije delujejo nad registrskimi in takojšnjimi operandi. Druga skupina so logične operacije, ki delujejo nad Boolovim tipom in imajo samo dve možni vrednosti – 0 ali 1. Čeprav se dajo vse funkcije zapisati z operacijo NAND, so zaradi pogoste uporabe dodane tudi druge, neelementarne. Med te spadajo and, or, xor in not in prav

tako delujejo nad registri in takojšnjimi operandi.

Metode razreda `c` sestavlja še, poleg aritmetično logičnih operacij, skupina za prenos podatkov med pomnilnikom in procesorjem. Ukazi za prenos delujejo nad vsemi primitivnimi tipi podatkov, kot so *oct1, oct2, oct4, oct8* in *register*. Instrukcije za prenos običajno prenesejo nek podatek iz pomnilnika v register, ali pa ga iz registra zapišejo v pomnilnik. Zato obstajajo za te ukaze vse vrste naslavljanja, od takojšnjega do bazno-indeksnega. Poleg naslova sta lahko oba operanda tudi registra, če gre za prenos med registri.

Enota za delo s skladom je definirana z razredom `stk`. Implementira osnovne metode za delo s podatkovno strukturo sklada, ki se nahaja v pomnilniku virtualnega stroja. Sklad je LIFO podatkovna struktura in ima samo dve osnovni operaciji; to sta `push`, ki shrani vrednost na sklad, in `pop`, ki zadnjo vrednost s sklada odstrani. Pri operaciji push se vrednost vrednost najprej shrani na sklad in zatem poveča skladovni števec. Števec torej vedno kaže na naslednjo prosto lokacijo. Analogno se pri odstranjevanju števec najprej zmanjša in šele nato se vrednost z vrha sklada odstrani. Metode sklada so prikazane v tabeli 5.

Na sklad se shranjujejo samo 4 ali 8 zložne vrednosti, kar pomeni, da se pri krajših tipih (oct1, oct2), dolžina podaljša za manjkajoče število zlogov. Metode sklada so v tabeli 5.

Ukazi vejitve se nahajajo v razredu `branch` ali na kratko `b`. Enota branch oz. kontrolna enota vsebuje tako brezpogojne kot pogojne vejitve. Vejitev je edina možnost, s katero programskemu števcu manualno spreminjamo vrednost. Vsak skočni ukaz namreč vsebuje naslov cilja tj. lokacijo, na katero naj skoči. Lokacija je zakodirana kot operand v samem ukazu in je lahko absolutna ali relativna. Skok pri slednji je odvisen od trenutne vrednosti programskega števca in se izračuna tako, da se relativni odmik, ki je lahko pozitiven ali negativen, prišteje vrednosti programskega števca, nato pa se izvajanje prenese na to lokacijo. V primeru absolutnega naslova se samo vrednost le-tega prenese v programski števec.

Ime	Koda	Operand 1	Operand 2	Operand 3	Opis
add	0x10	register	register		seštevanje
add	0x11	oct1	register		seštevanje
add	0x12	oct2	register		seštevanje
add	0x13	oct4	register		seštevanje
add	0x14	register	register	register	seštevanje
sub	0x15	register	register		odštevanje
sub	0x16	oct1	register		odštevanje
sub	0x17	oct2	register		odštevanje
sub	0x18	oct4	register		odštevanje
sub	0x19	register	register	register	odštevanje
mul	0x1a	register	register		množenje
mul	0x1b	oct1	register		množenje
mul	0x1c	oct2	register		množenje
mul	0x1d	oct4	register		množenje
mul	0x1e	register	register	register	množenje
mul_s	0x1f	register	register		predznačeno
mul_s	0x20	oct1	register		predznačeno
mul_s	0x21	oct2	register		predznačeno
mul_s	0x22	oct4	register		predznačeno
mul_s	0x23	register	register	register	predznačeno
div	0x24	register	register		deljenje
div	0x25	oct1	register		deljenje
div	0x26	oct2	register		deljenje
div	0x27	oct4	register		deljenje
div	0x28	register	register	register	deljenje
div_s	0x29	register	register		predznačeno
div_s	0x2a	oct1	register		predznačeno
div_s	0x2b	oct2	register		predznačeno
div_s	0x2c	oct4	register		predznačeno
div_s	0x2d	register	register	register	predznačeno
and	0x2e	register	register		logični in
and	0x2f	oct4	register		logični in
and	0x30	register	register	register	logični in
or	0x31	register	register		logični ali
or	0x32	oct4	register		logični ali
or	0x33	register	register	register	logični ali

Tabela 4: Enota `core`.

Ime	Koda	Operand 1	Operand 2	Operand 3	Opis
xor	0x34	register	register		ekskluzivni ali
xor	0x35	oct4	register		ekskluzivni ali
xor	0x36	register	register	register	ekskluzivni ali
not	0x37	register	register		logični ne
neg	0x38	register			negacija
nop	0x39				brez operacije
rol	0x3a	register	register		rotiraj levo
rol	0x3b	oct1	register		rotiraj levo
ror	0x3c	register	register		rotiraj desno
ror	0x3d	oct1	register		rotiraj desno
shl	0x3e	register	register		pomik levo
shl	0x3f	oct1	register		pomik levo
shr	0x42	register	register		pomik desno
shr	0x43	oct1	register		pomik desno
loct1	0x46	register	address		naloži oct1
loct1	0x47	register	base		naloži oct1
loct1	0x48	register	basei		naloži oct1
loct1_s	0x49	register	address		predznačeno
loct1_s	0x4a	register	base		predznačeno
loct1_s	0x4b	register	basei		predznačeno
loct2	0x4c	register	address		naloži oct2
loct2	0x4d	register	base		naloži oct2
loct2	0x4e	register	basei		naloži oct2
loct2_s	0x4f	register	address		predznačeno
loct2_s	0x50	register	base		predznačeno
loct2_s	0x51	register	basei		predznačeno
loct4	0x52	register	address		naloži oct4
loct4	0x53	register	base		naloži oct4
loct4	0x54	register	basei		naloži oct4
loct4_s	0x55	register	address		predznačeno
loct4_s	0x56	register	base		predznačeno
loct4_s	0x57	register	basei		predznačeno
soct4	0x58	register	address		shrani oct4
soct4	0x59	register	base		shrani oct4
soct4	0x5a	register	basei		shrani oct4
soct1	0x5b	register	address		shrani oct1

Tabela 5: Enota `core` (*nadaljevanje*).

Ime	Koda	Operand 1	Operand 2	Operand 3	Opis
soct1	0x5c	register	base		shrani oct1
soct1	0x5d	register	basei		shrani oct1
soct2	0x5e	register	address		shrani oct2
soct2	0x5f	register	base		shrani oct2
soct2	0x60	register	basei		shrani oct2
move	0x61	register	register		premakni
laddr	0x62	register	address		naloži naslov
call	0x63	address			kliči
call	0x64	base			kliči
call	0x65	basei			kliči
move	0x66	oct1	register		premakni
move	0x67	oct2	register		
move	0x68	oct4	register		
callv	0x69	oct1			virtualno
callv	0x6a	oct2			virtualno

Tabela 6: Enota `core` (*nadaljevanje*).

Ime	Koda	Operand 1	Operand 2	Operand 3	Opis
push	0x80	register			na sklad
push	0x81	oct1			na sklad
push	0x82	oct2			na sklad
push	0x83	oct4			na sklad
pop	0x84	register			s sklada

Tabela 7: Enota `stk`.

Ime	Koda	Operand 1	Operand 2	Operand 3	Opis
to	0x90	address			skoči
ifz	0x91	register	address		če je 0
ifnz	0x92	register	address		če ni 0
ifeq	0x93	register	oct4	address	če je enako
ifeq	0x94	register	oct2	address	če je enako
ifeq	0x95	register	oct1	address	če je enako
ifneq	0x96	register	oct4	address	če ni enako
ifneq	0x97	register	oct2	address	če ni enako
ifneq	0x98	register	oct1	address	če ni enako
back	0x98				vrnitev

Tabela 8: Enota `branch`.

Pogojni skok je lahko realiziran z eno samo ali dvema instrukcijama. V prvem primeru mora ukaz testirati pogoj in glede na rezultat testiranja nadaljevati ali skočiti nazaj. Takšen tip vejitvene instrukcije se imenuje "testiraj in skoči" in ne uporablja posebnih bitov ali "zastavic" za skok. Število oprandov v takšni instrukciji je običajno 3: objekt, ki ga testiramo, vrednost s katero testiramo in lokacija skoka. V drugem primeru pa imamo par ortogonalnih instrukcij, od katerih prva opravi test in nastavi določen register ali kakšne druge indikatorske bite na neko vrednost. Druga instrukcija pa na podlagi nastavljenih bitov ali registra nadaljuje izavajanje ali pa skoči na specificirano lokacijo. Pomembno pri tem je, da lahko drugo instrukcijo postavimo kamorkoli za prvim ukazom. Potrebno je le paziti, da ni zmes nobene druge instrukcije, ki bi utegnila spremeniti nastavljene bite. Rezultate testiranja lahko tako uporabi več sledečih ukazov. Metode razreda `branch` se nahajajo v tabeli 6. Sistemske storitve realizira sistemska enota `sys`. Ukaze v tej enoti prikazuje tabela 7. Sistemski ukazi so podobni sistemskim klicem operacijskega sistema. Ker želimo, da so vsi ukazi neodvisni od ciljne arhitekture in operacijskega sistema, najpogostejše med njimi implementiramo kar v sami arhitekturi in se tako izognemo nepotrebnim soodvisnostim. Metode sestavljenih tipov se nahajajo v svojih enotah. Metode tipa `integer` so v tabeli 8, tipa `string` pa v tabeli 9. Tipa `short` in `byte` sta zelo podobna tipu `integer` in ju zato posebej ne prikazujemo.

Ime	Koda	Operand 1	Operand 2	Operand 3	Opis
print	0xa0	register			izpiši
print	0xa1	address			izpiši
print	0xa2	base			izpiši
print	0xa3	basei			izpiši
exit	0xa4	base			izhod

Tabela 9: Enota `sys`.

3.8 Razširitve arhitekture

V obstoječo arhitekturo je mogoče vpeljati manjše in obsežnejše razširitve, med njimi tudi takšne, ki ne bi podpirale le abstraktnih tipov programskega jezika, temveč tudi druge, kompleksnejše konstrukte. Med te sodijo v prvi vrsti zanke, izrazi in bloki. Ne glede za katero vrsto zanke gre, jo lahko modeliramo kot atomarni sestavni del jezika, ki ima, podobno kot abstraktni tip, pripadajoče metode in stanje. Za ustvarjanje objekta v pomnilniku veljajo enaka pravila kot pri sestavljenih tipih, prav tako za klicanje metod oz. pošiljanje sporočil objektu.

Enako je mogoče modelirati posamezen blok v programskem jeziku. Ta je sestavljen iz stavkov, zank in drugih blokov, ki na ta način sestavljajo hierarhijo od korena do listov. V ta namen uvedemo postopek "pakiranja" izraza ali bloka s pripadajočim okoljem v enoto, ki se lahko prenaša kot celota, hkrati pa se lahko "odpakira" in ovrednoti preko ustreznih postopkov [6]. V našem primeru bi npr. izraz prevedli v objekt, njegovo okolje pa v stanje objekta. Tak izraz se nato pošilja po okolju kot objekt in se ovrednoti, ko je potrebno, preko ustreznih metod. Takšen paket, ki vsebuje stanja in njihovo okolje, se imenuje zaprtje.

```
<zaprtje> := [<izraz>, <okolje>]
<zaprtje> := [<zanka>, <okolje>]
<zaprtje> := [<blok>, <okolje>]
```

`okolje` je tipično predstavljeno kot asociativni seznam dvojic *(spremenljivka, vrednost)* in se imenuje tudi kontekst; posledično je ovrednotenje izraza kontekstno odvisno. Zaprtje omogoča višjo stopnjo abstrakcije, s pomočjo seznamov praktično neskončno

Ime	Koda	Operand 1	Operand 2	Operand 3	Opis
new	0xb0	oct1			konstruktor
new	0xb1	oct2			konstruktor
new	0xb2	oct4			konstruktor
new	0xb3	register			konstruktor
new	0xb4	integer			konstruktor
add	0xb5	oct1			seštevanje
add	0xb6	oct2			seštevanje
add	0xb7	oct4			seštevanje
add	0xb8	register			seštevanje
add	0xb9	integer			seštevanje
sub	0xba	oct1			odštevanje
sub	0xbb	oct2			odštevanje
sub	0xbc	oct4			odštevanje
sub	0xbd	register			odštevanje
sub	0xbe	integer			odštevanje
mul	0xbf	oct1			množenje
mul	0xc0	oct2			množenje
mul	0xc1	oct4			množenje
mul	0xc2	register			množenje
mul	0xc3	integer			množenje
div	0xc4	oct1			deljenje
div	0xc5	oct2			deljenje
div	0xc6	oct4			deljenje
div	0xc7	register			deljenje
div	0xc8	integer			deljenje
ifeq	0xc9	oct4	address		če je enako
ifeq	0xca	integer	address		če je enako
ifneq	0xcb	oct4	address		če ni enako
ifneq	0xcc	integer	address		če ni enako
iflow	0xcd	oct4	address		če je manjše
iflow	0xce	integer	address		če je manjše
ifhigh	0xcf	oct4	address		če je večje
ifhigh	0xd0	integer	address		če je večje
store	0xd1	register			shrani
store	0xd2	integer			shrani
load	0xd3	register			naloži
load	0xd4	integer			naloži
delete	0xd5				zbriši

Tabela 10: Enota `integer`.

Ime	Koda	Operand 1	Operand 2	Operand 3	Opis
new	0x101	address			konstruktor
new	0x102	base			konstruktor
new	0x103	basei			konstruktor
new	0x104	string			konstruktor
new	0x105	integer			konstruktor
new	0x106	short			konstruktor
at	0x107	oct4	register		vrni znak
length	0x108	register			dolžina
upper	0x109				pretvorba
lower	0x10a				pretvorba
left	0x10b	oct4			levi del
right	0x10c	oct4			desni del
mid	0x10d	oct4	oct4		vmesni del
delete	0x10e				zbriši

Tabela 11: Enota `string`.

število objektov v izrazu, poleg tega pa še osnovo za paralelno izvajanje. V primeru paralelnega načrtovanja, se izračunajo odvisnosti med posameznimi zaprtji, z namenom, da se ugotovi, katera izvajanja lahko tečejo sočasno. Ni potrebno posebej poudarjati, da lahko vsako zaprtje vsebuje več podzaprtij.

Podobno bi lahko konstruirali zaprtje za npr. zanko "for". V notaciji programskega jezika java se ta zapiše:

```
for(int var = izraz; bool izraz; izrazi){
    telo;
}
```

Zaprtje, ki predstavlja zanko, je sestavljeno iz okolja, v katerem je spremenljivka `var` in treh ali več podzaprtij v glavi ter enega podzaprtja, ki predstavlja telo zanke.

S tem smo zaključili s predstavitvijo naše arhitekture in programskim modelom, ki je tej arhitekturi namenjen. Podrobno smo predstavili tudi nabor instrukcij vseh procesorskih enot in njihovo vlogo v arhitekturi. V naslednjem poglavju se bomo lotili navideznega stroja, ki bo pravkar predstavljeno arhitekturo implementiral in omogočil izvajanje prevedenega zložnega koda.

4. Virtualni stroj

4.1 Uvod in splošna pojmovanja

Populariziran trend razvoja sodobne programske opreme je pridobitev na programski prenosljivosti, tudi binarnega nivoja, s prevajanjem v vmesno obliko, osnovano na definiciji nekega abstraktnega stroja. Takšni pristopi datirajo še v 70 leta, vendar so šele v zadnjem obdobju pridobili velik zamah v razvoju, predvsem pod vplivom programskega jezika jave. Čeprav so virtualni stroji v principih delovanja zelo podobni, obstajajo med njimi velike razlike.

Virtualni stroj je simulator realnega procesorja, če pa mu dodamo še dodatane funkcije, lahko postane simulator celotnega računalniškega sistema. Običajno služi kot hipotetični računalnik, ki je namenjen za določene naloge. Z njim, običajno enostavneje in veliko ceneje, kot na realnem stroju, simuliramo izvajanje in reševanje teh nalog. Lahko ga hitro prilagodimo za nove probleme in nova izvajalna okolja in mu dodamo ali odvzamemo določene atribute. Eden izmed popularnejših simulatorjev računalniških sistemov je gotovo SPIM, ki simulira 32 bitno RISC arhitekturo mikroprocesorjev MIPS. Procedura generiranja kode za izvajanje na virtualnem stroju je relativno preprosta. Jezikovno odvisen del se semantično preveri in prevede v vmesno jezikovno obliko, iz katere se generira koda. Jezikovni del je praviloma popolnoma neodvisen od ciljne arhitekture, generator kode pa je obratno, neodvisen od jezika. Prevajalnik, ki zadošča takšnim zahtevam, je potemtakem "polovični prevajalnik". Veliko takšnih vmesnih oblik jezika je osnovanih na abstraktnih skladovnih strojih; mogoče najbolj znani primer tega je P-kod, ki je bil izdelan za ETH Pascalov prevajalnik. Prevajanje v vmesno obliko za izvajanje na javinem virtualnem stroju, prikazuje slika 12. Program, izdelan za abstraktni skladovni stroj, je tako lahko preveden na dva načina. Prevajalnik ga lahko prevede popolnoma, do strojnega koda ciljnega procesorja, lahko pa se uporabi interpreter, ki emulira abstraktni stroj na procesorju. Razlika je predvsem v faktorju hitrosti, ki je pri interpretativnem načinu veliko nižji. Pozitivna stran je kompaktnost

samega programa in s tem prenosljivost, saj je na novo arhitekturo potrebno prenesti samo verzijo virtualnega stroja.

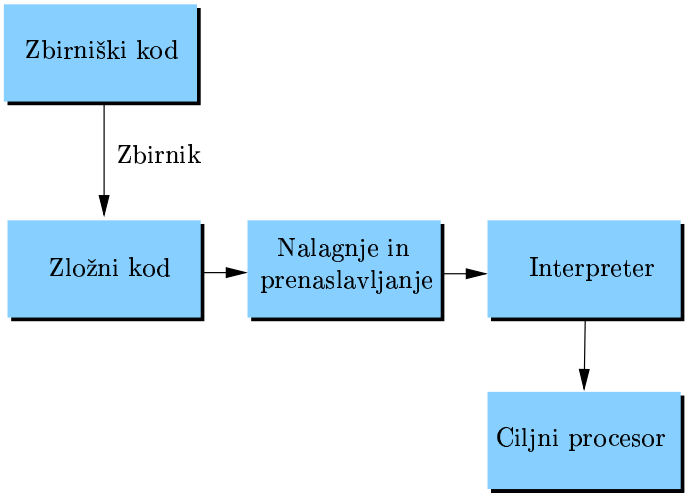
Večina modernih prevajalnikov za virtualne stroje, med njimi tudi javin, tvori vmesni program, ki vsebuje simbolične informacije, ki omogočajo preverjanje in pretvorbo podatkovnih tipov med izvajanjem.

Izvajalni mehanizem skladovnega stroja temelji na evaluaciji sklada s skupino instrukcij, ki ta sklad manipulirajo. Preprost primer za to je naslednji kos JVM kode:

```
iload_0       ; lokalna spremenljivka 0 na sklad
iload_1       ; lokalna spremenljivka 1 na sklad
imul          ; seštejemo
istore_2      ; shranimo
```

Instrukcije JVM (Java Virtual Machine) uporabljajo predpone za posamezne tipe (i = integer, f = float, itd.) in posamezne instrukcije za prvih nekaj lokalnih spremenljivk, ki se največ uporabljajo. Koda, ki jo Javin prevajalnik generira, je zaradi konstantne dolžine 1 zloga, izjemno kompaktna. Podatki, to so razredne spremenljivke in strukture, so predstavljeni v lokalnih spremenljivkah ali na izvršilnem skladu stroja. Agregatni podatki se lahko nahajajo samo v dinamično alociranih objektih, ki se avtomatično odstranijo, ko niso več potrebni.

Za klicanje metod so lahko uporabljeni različni pristopi. Običajno virtualna arhitektura vsebuje posebne instrukcije za klic statičnih in virtualnih metod. V primeru Jave, ta vsebuje še ukaza za klicanje vmesniških metod in statično klicanje virtualnih metod. V primeru klicanja virtualnih metod, je potrebno metodi podati še parameter this, ki kaže na objekt, nad katerim je metoda poklicana. Kako se poda, je povsem implementacijske narave, običajno pa se pošlje kot prvi ali zadnji parameter na skladu. Problemi s skladom se pri prenašanju parametrov po vrednosti lahko pojavijo pri daljših podatkovnih strukturah, ki bi zasedle preveč prostora; to velja še posebej, če je velikost sklada omejena. Da do takšnih težav ne bi prišlo, je bolje prenašati kompleksne podatkovne strukture, kot so polja in abstraktni tipi, po referenci namesto po vrednosti, pri čemer zasedajo samo velikost pomnilniškega naslova.



Slika 12: Prevajanje programa v zložni kod in izvajanje.

V splošnem je zaželjeno, da je arhitektura virtualnega stroja čimmanj specifična in neodvisna od sintakse in semantike programskega jezika, ne glede na to ali se izvaja prevajanje za skladovni, registrski ali kakšen drug tip stroja.

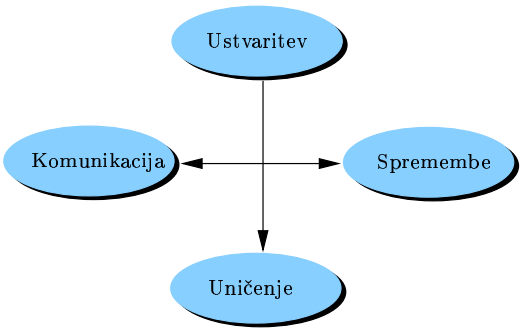
4.2 Načrtovanje virtualnega stroja

4.2.1 Okolje delovanja

V objektno orientiranem programskem okolju obravnavamo celotni sistem kot zbirko objektov. Objekt je integrirana enota podatkov in procedur. Podatki objekta so shranjeni v spremenljivkah, ki lahko vsebujejo podatke osnovnih tipov ali reference na druge objekte. V kolikor imamo samo tipe drugih objektov, govorimo o čistih objektnih jezikih. V spošnem so objekti dinamične entitete v tem oziru, da jih lahko dinamično ustvarimo, imajo spremenljivo stanje med življenjskim ciklom na sliki 13 [14].

En izmed pomembnejših aspektov objektne orientiranosti je onemogočitev neposredne dostopnosti do objektovega stanja. Edini način, kako objekti komunicirajo, je preko pošiljanja sporočil. Sporočilo je zahteva prejemniku za izvršitev ene izmed metod; metoda lahko dostopa do objekta. Takšen mehanizem zagotavlja ločitev med implementacijo objekta in obnašanjem, katerega lahko opazujemo od zunaj. Objekt torej enkapsulira zbirko atributov, ki je shranjena v privatnem pomnilniku.

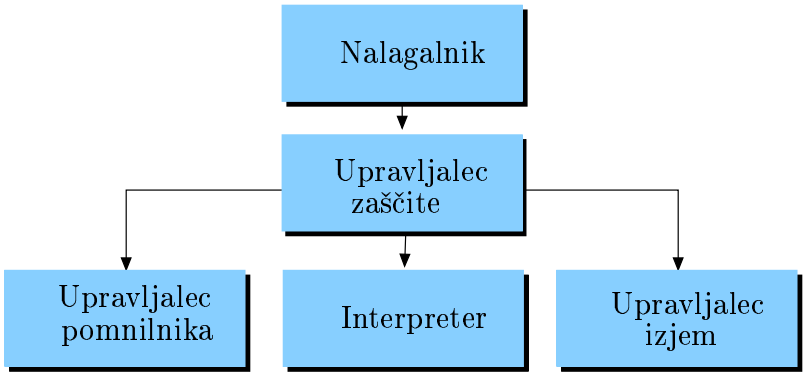
Objekt je ustvarjen in uničen s strani rutin virtualnega stroja, ki to storijo s pomočjo metod *new* in *delete*. Vsak objekt, ki se lahko ustvari, mora imeti definirani metodi new in delete. Metoda new alocira pomnilnik, potreben za shranitev objekta v pomnilniku in inicializira podatke na zahtevane vrednosti. Objekt je po tem dostopen preko kazalca v pomnilniku, s pomočjo katerega se izvajajo metode. Metoda torej potrebuje instanciran razred in morebitne parametre. Omeniti je potrebno, da lahko pri ustvarjanju objekta pride tudi do ustvarjanja objektov, ki jih instancirani razred deduje. Razred lahko definira nek atribut v obliki virtualnega razreda, enako kot ga lahko definira v obliki virtualne procedure [15]. Podrazred lahko nato redefinira oz. refinira ta atribut v bolj splošen razred. Virtualni stroj mora v tem primeru vedeti, za kateri atribut gre in v katerem razredu je definiran.



Slika 13: Življenjski cikel objekta.

4.2.2 Struktura

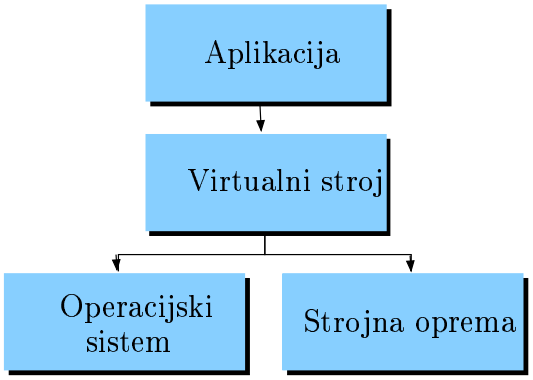
Struktura virtualnega stroja je odvisna od zapletenosti in hitrosti izvajanja operacij, stopnje zaščite, vzdrževanja pomnilnika in drugih faktorjev.



Slika 14: Arhitektura hipotetičnega virtualnega stroja.

Najvidnejša prednost virtualnega stroja pred pravim procesorjem je nedvomno v samem načrtovanju stroja in prenosljivosti kode. Načrtovanje virtualnega stroja je namreč veliko preprostejše in cenejše kakor načrtovanje pravega procesorja s tega stališča, da je virtualni stroj v bistvu samo program, ki ga lahko po volji spreminjamo, nadgrajujemo in dopolnjujemo do želenega delovanja. Velika prednost se pokaže tudi pri prenosljivosti, kjer je potrebno na novo platformo prenesti samo virtualni stroj – obstoječa binarna koda programov ostane enaka in je tako ni potrebno ponovno prevajati.

Virtualni stroj se postavlja med izvajajočim se programom in specifično strojno opremo ter na ta način doda nov nivo abstrakcije in s tem tudi zaščite med fizičnim in programskim sklopom. Učinkovita zaščita je izjemnega pomena, saj preprečuje, da bi program izvajal napačne ali nedovoljene operacije in tako povzročil škodo.
Jeziki, katerih izvajanje temelji na virtualnem stroju, se običajno ne interpretirajo na izvornem nivoju, temveč se prevedejo v zložno kodo (byte code). Prevajanje se izvrši



Slika 15: Slojna struktura virtualnega stroja.

enkrat, bodisi dinamično – preden program zaženemo – ali kot samostojen proces, ki uporablja lasten prevajalnik. Izhod iz prevajalnika je zložna koda (sestavljena iz zlogov), ki jo nalagalnik naloži v pomnilnik, da jo lahko interpreter izvede.
Slabost virtualnega stroja je zagotovo hitrost, ki je običajno bistveno manjša od procesorjeve, kar pomeni, da so tudi zlogovno prevedeni jeziki nekajkrat počasnejši od popolno prevedenih.

4.2.3 Varnost

Eden izmed pomembnejših razlogov za uvedbo virtualnega procesorja je zagotovo varni način izvajanja, ki se izkaže, kadar pride do izjeme. V splošnem se da koncept varnosti zelo elegantno vpeljati v arhitekturo virtualnega stroja, še posebej, če je ta napisan v višjem programskem jeziku. V kolikor je zahtevana visoka stopnja varnosti, je potrebno implementirati dovršene mehanizme zaščite, ki ne upočasnjujejo izvajanja bolj kot je sprejemljivo. Realni procesorji preverjajo format instrukcije neposredno pred njeno izvršitvijo. Če pride do napake, npr. ukaz s prebrano operacijsko kodo sploh ne eksistira ali pa so operandi napačni, procesor signalizira ustrezno izjemno stanje z informacijami, ki povedo za kakšno napako gre. Takšna implementacija bi bila v virtualnem stroju nedopustna s stališča učinkovitosti, saj bi celotni sistem izjemno upočasnila. Potreben

je drugačen pristop. Ena izmed možnosti je ta, da se zložna koda v celoti preveri, namesto med, pred izvajanjem. Na ta način se lahko preverijo tako operacijske kode ukazov kot tudi pripadajoči operandi in njihovi tipi. Vendar vsega ne gre preveriti pred izvedbo, npr. dostopa do pomnilnika. Med izvajanjem se namreč ustvarjajo v pomnilniku novi objekti in pri dostopu do nedovoljenih lokacij bo prišlo do izjeme. Rešitev je preprečitev neposrednih dostopov do pomnilniških lokacij z višjo stopnjo abstrakcije v programskem jeziku. Če imamo namreč dovolj dobro načrtovane metode, nam sploh ne bo potrebno neposredno posegati v pomnilnik. Primerov kode, kjer lahko pride do izjeme, je veliko. Če prenašamo podatke iz ali v pomnilnik preko določenega tipa naslavljanja, obstaja možnost, da naslovimo pomnilnik, ki je izven našega dosega oz. lokacijo, ki je v lasti nekega drugega programa. Del kode

```
c.move 0x7fffffff,$r5
c.loct4 [$r5],$r6
```

je sintaktično povsem pravilen, vendar bo v drugem ukazu prišlo do izjeme, če se pomnilniška lokacija `0x7fffffff` ne nahaja v alociranem pomnilniku za naš program. Prevajalnik takšnih napak ne more preprečiti, saj se pojavijo šele v času izvajanja. Še več, tudi nalagalnik jih ne more preprečiti, ker v samih instrukcijah ni nobenih napačnih tipov operandov. Pravzaprav takšnih vrst napak sploh ni mogoče preprečiti – če obstajajo, se bodo pojavile kvečjemu med izvajanjem kode. Da bi se pravilno interpretirale, jih mora mehanizem virtualnega stroja pravočasno zaznati. Ko bomo naslovili napačen ali neobstoječ del pomnilniškega prostora, bo procesor operacijskemu sistemu signaliziral izjemo. Ta bo proces virtualnega stroja nasilno prekinil. Virtualni stroj mora torej teči v nekakšnem zaščitenem načinu, ki takšne napake preprečuje; podobno teče realni procesor v zaščitenem načinu v večprogramskem okolju.

Način preprečitve takšnih pojavov je lahko ta, da se vsaka instrukcija, ki kakorkoli naslavlja pomnilnik, preveri, ali morda ne izdaja napačnega naslova. Če je to res, se lahko izjema "ujame" in nadzorovano konča program ali pa z njim nadaljuje, brez izvedbe instrukcije, ki je povzročila izjemo. Drug način je, da v arhitekturi preprosto

ne modeliramo takšnih ukazov, ki bi lahko neposredno naslavljale pomnilnik. V kolikor imamo kompleksne ukaze, definirane v razredih, se takšne instrukcije skorajda ne bodo uporabljale. Nazoren primer za to je Javin virtualni stroj, ki ne dopušča neposrednih pomnilniških operandov ampak ima vse shranjene na skladu. Vsi ukazi operirajo nad skladom, ki ga virtualni stroj prej alocira.

Drug izvor napak v objektno orientiranih jezikih so dinamični tipi. Pri polimorfnih lastnostih objektov se iz semantične informacije v času prevajanja ne da razbrati, kateri tip bo ob izvajanju zares uporabljen. Če se del programa prevede za določen tip, ob izvajanju pa se tip spremeni, pride do napake. Metode, ki jih kličemo nad pričakovanim tipom, za spremenjeni tip sploh nimajo pomena ali pa sploh ne obstajajo. Primer za to lahko spet najdemo v javi, če želimo prebrati serializiran objekt nekega tipa iz datoteke. Če zapisan objekt ni pričakovanega tipa, nad njim ne moremo izvajati želenih metod. Pri klicu virtualne metode nad nekim objektom, mora virtualni stroj vedeti katero metodo poklicati, saj je teh v splošnem lahko več. V ta namen služi vsakemu ustvarjenemu objektu tabela vseh metod, ki jih objekt vsebuje. Na podlagi tega lahko virtualni stroj določi, katero metodo je potrebno poklicati in ali ta metoda sploh obstaja. Če je ni v tabeli, se mora sprožiti izjema, saj bi klicanje metode nad napačno strukturo podatkov lahko povzročilo nepredvidljive posledice.

Izjeme pa niso nujno usodne za program. Lahko se prožijo tudi ob lažjih napakah, ki se dajo v času izvajanja odpraviti, če je za njih napisana ustrezna rutina. Takšen pristop uporablja Java. Vsak del programske kode, ki bi utegnil povzročiti izjemo je umeščen med stavka `try` in `catch`, ki izjemo ulovita. Stavek `catch` vsebuje še kodo oz. rutino, ki se izvrši samo, če pride do izjeme. Ko se to zgodi, Javin sistem pokliče ustrezno rutino tako, da začne iskati v metodi, v kateri je do izjeme prišlo, potem pa se pomika navzgor po skladu skozi vse metode, ki so bile klicane. Če se rutina najde, se izvede njena koda, ki lahko [5]:

- uporabi ukaz `goto` za nadaljevanje izvajanja,

- zaključi izvajanje metode z ukazom `return` ali
- uporabi ukaz `athrow`, ki generira novo izjemo.

V kolikor ni bilo najdene ustrezne rutine, se pokliče sistemska, ki običajno izpiše neko sporočilo in zaključi izvajanje programa. Struktura za izjemo je v javi naslednja:

```
class Handler{
      int from_pc;       // začetek kode, kjer se lovi izjema
      int to_pc;         // konec kode, kjer se lovi izjema
      int handler_pc;    // naslov kam skočiti, če pride do izjeme
      Class catch_type;  // razred te izjeme
}
```

Vsaka metoda vsebuje tabelo izjem. Če v deklaraciji metode ni navedenih nobenih izjem, je ta tabela prazna. Tabela je pravzaprav zaporedje struktur `Handler`. V Javi so rutine za napake pozicionirane zunaj kode, ki je zaščitena. Dve območji zaščitene kode se nikoli delno ne prekrivata – ali sta popolnoma ločeni ali pa je eno vsebovano v drugem. Velja tudi, da je vsak dostop do rutine onemogočen; do kode se lahko pride izključno preko izjeme.

4.2.4 Izvajanje kode

Kadarkoli želimo izvesti program napisan v našem zbirnem jeziku, moramo to storiti preko virtualnega stroja. Instrukcije, ki sestavljajo program, se ne izvajajo neposredno na gostiteljskem procesorju v računalniku, ampak jih izvaja poseben interpreter, ki predstavlja jedro virtualnega stroja. Virtualni stroj je torej program, ki koračno izvede vsako instrukcijo posebej z uporabo več, običajno preprostejših in krajših, instrukcij ciljnega procesorja. Interpreter torej predstavlja nekakšen "virtualni procesor".
Postopek izvajanja je naslednji:

1. branje podatkov in kode,

2. nalaganje in relokacija,

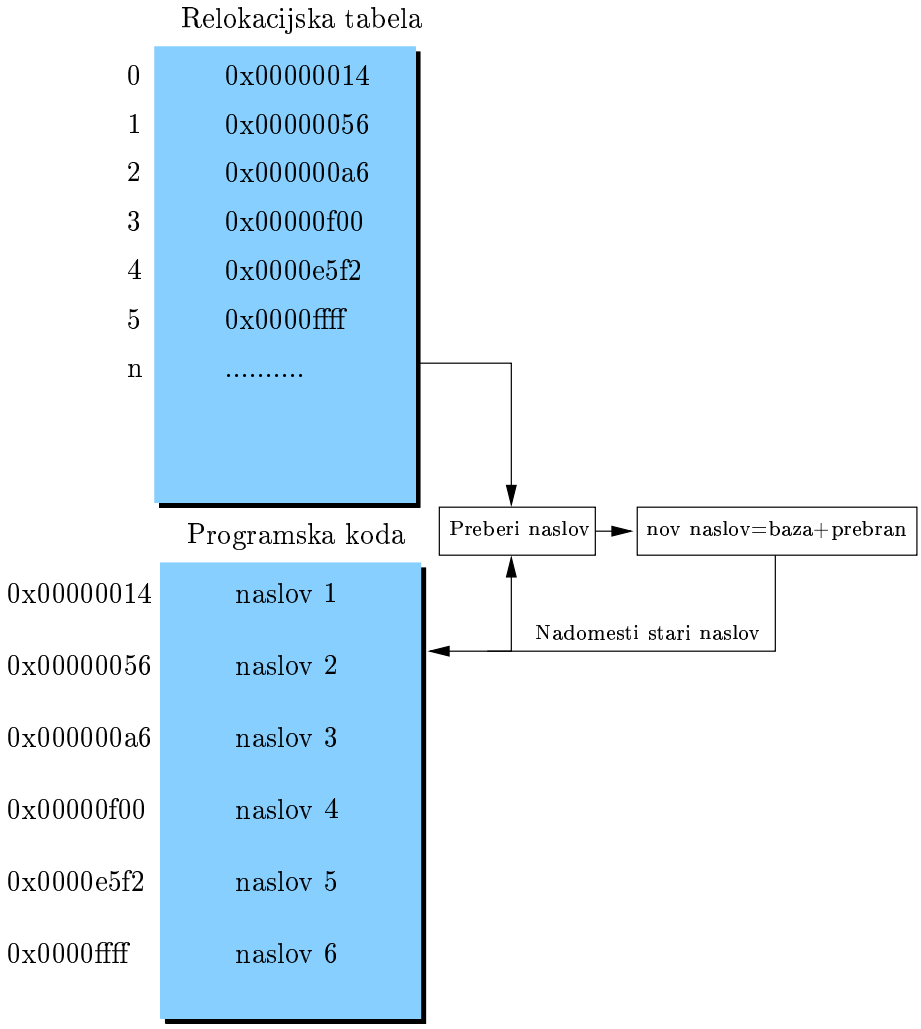
3. preverjanje,

4. izvedba in

5. izhod.

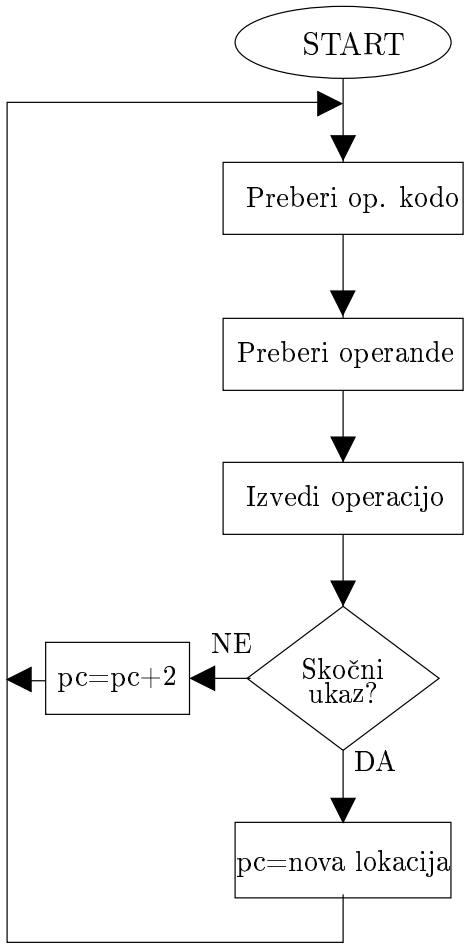
Pri procesu nalaganja se program naloži na alocirano lokacijo nekje v pomnilniku. Ker so naslovi v operandih ukazov relativni, je nujno, da se prekalkulirajo glede na lokacijo, kjer se program naložil. Zaradi tega je lego posameznih ukazov znotraj nekega modula smiselno opisati kot odmik od začetka modula. Prenaslavljanje je potrebno tudi pri nalaganju večih modulov, med katerimi obstajajo določene odvisnosti. Ne glede na tip naslavljanja v operandih, je končen naslov, ki ga procesor izda, vselej absolutni [12]. Do tega lahko pridemo v času prevajanja, kjer se specificira absolutni naslov, na katerega je program potrebno naložiti. Druga možnost je, da se naslovi izračunajo v času nalaganja, tretji način pa je, da pride do tvorbe absolutnega naslova, ko je program že v pomnilniku, pri čemer se vsi naslovi sklicujejo na nek referenčni naslov. V našem virtualnem stroju je uporabljena druga možnost, torej prenaslavljanje ob nalaganju.

Programska shema je sestavljena iz podatkov v *.data* razdelku in izvedljive programske kode v *.text* razdelku. Pri nalaganju se najprej na določen naslov naložijo vsi podatki, takoj za njimi pa izvedljiva koda. Absolutni naslov začetka programa je torej od alociranega naslova večji za toliko, kolikor zasedajo podatki. Razlika med podatki in programom je vidna samo na programski ravni, kjer oboje ločimo z dvema različnima sekcijama. Po drugi strani pa virtualni stroj ne pozna nobene razlike med obojim; pod določenimi pogoji bi lahko izvrševal kodo v podatkovni sekciji in pisal ali bral podatke iz tekstovne sekcije.
Izvedba programa sestoji iz izvedbe posameznih instrukcij. Ta poteka tako, da



Slika 16: Relokacija ob nalaganju v pomnilnik.

interpreter najprej iz pomnilnika prebere operacijsko kodo instrukcije in nato operande. Nato sledi izvršitev operacije in skok na nov ukaz v pomnilniku s povečanjem programskega števca za velikost pravkar izvedenega ukaza. V kolikor je izvrševan vejitveni ukaz, se izvajanje prenese na naslov, določen v operandih vejitvenega ukaza. Izvajanje programa se konča s sistemskim ukazom *exit*. Korake izvajanja prikazuje slika 17.



Slika 17: Koraki izvajanja.

4.2.5 Ciljna arhitektura x86

Virtualni stroj je močno sistemsko in strojno odvisen del programske opreme. Implementirana je bila verzija za PC, torej za arhitekturo, ki temelji na 32 bitni x86 topologiji. Razvoj teh procesorjev se je od 80 let strmo vzpenjal za komercialne potrebe. Te so vselej zahtevale kompatibilnost za starejše programe, kar je močno razvidno v sami arhitekturi. Arhitektura x86 je tipična CISC arhitektura z obsežnim naborom instrukcij in kompleksnostjo delovanja. Arhitektura ima 8 32 bitnih registrov, od katerih so splošno namenski le štirje, pa še za njih to ne velja povsem. Zares splošno namenski je

samo akumulator EAX. Registre prikazuje tabela 11. Vsak izmed registrov je sestavljen iz spodnjih in zgornjih 16 bitov. Spodnjih 16 bitov je možno referencirati s 16 bitnimi registri, ki so označeni enako kot 32 bitni, le da so brez črke E pred imenom. Pomen in uporaba teh registrov je povsem enaka njihovim 32 bitnim različicam. Poleg tega obstajajo še 8 bitni registri, ki so označeni `al, bl, cl, dl`. Poleg osnovnih registrov se uporablja še register zastavic, v katerem posamezne bite nastavljajo nekatere operacije. Ta ni neposredno berljiv, da pa se ga shraniti na sklad in takoj zatem v register. Register programskega števca se na x86 stroju imenuje IP oz. EIP v 32 bitnem načinu delovanja.

Izvorni/ciljni operand	**Drugi izvorni operand**
register	register
register	takojšnji
register	pomnilniški
pomnilniški	register
pomnilniški	takojšnji

Tabela 12: Možne kombinacije tipov operandov v aritmetičnih, logičnih in podatkovnih instrukcijah.

V primerjavi s 3-operandnimi RISC stroji, ima x86 večino instrukcij implementiranih za 2 operanda, kar pomeni, da mora biti ciljni register enak kot prvi izvorni. Večina instrukcij deluje tudi nad pomnilniškimi operandi, kjer je uporabljen format $Mem[r_1 + c] \leftarrow Mem[r_1+c] \bigoplus r_2$ ali $r_1 \leftarrow r_1 Mem[r_2+c]$. Vse možne kombinacije tipov operandov so prikazane v tabeli 10.

V x86, kot tipični CISC arhitekturi, je podprtih je 7 naslovnih načinov:

- absolutno,
- registrsko posredno,
- bazno,
- indeksno,

- bazno indeksno z odmikom,
- bazno s skaliranjem in indeksom in
- bazno s sklairanim indeksom in odmikom.

Odmik je lahko dolg 8 ali 32 bitov v 32 bitnem načinu in 8 ali 16 bitov v 16 bitnem načinu. Vsak pomnilniški operand lahko vsebuje katerikoli naslovni način, so pa omejitve, kateri registri so lahko uporabljeni v posameznem naslavljanju [4]:

- Registrsko posredno - `BX, SI, DI` v 16 bitnem načinu in `EAX, ECX, EDX, EBX, ESI` in `EDI` v 32 bitnem načinu.
- Bazno z 8 ali 32 bitnim odmikom - `BP, BX, SI, DI` v 16 bitnem načinu in `EAX, ECX, EDX, EBX, ESI, EDI` v 32 bitnem načinu.
- Indeksno - naslov je vsota dveh registrov, pri čemer so dovoljene kombinacije `BX+SI, BX+DI, BP+SI` in `BP+DI`.
- Bazno indeksno z 8 ali 16 bitnim odmikom - naslov je vsota odmika in dveh registrov. Registri so enaki kot pri prejšnjem načinu.
- Bazno s sklairanjem in indeksom - naslov je podan z enačbo: $bazni\ register + 2^{skala} * indeksni\ register$. `skala` je lahko 0, 1, 2 ali 3, indeksni register je lahko katerikoli razen `ESP`, bazni register pa je lahko prav tako katerikoli, tudi ESP. Ta tip naslavljanja je možen samo v 32 bitnem načinu.
- Bazno s sklarinim indeksom in 8 ali 32 bitnim odmikom - naslov je vsota odmika in naslova izračunanega enako kot v prejšnjem primeru.

Končni naslov, ki ga procesor da na vodilo, je dolg 32 bitov v formatu z majhnim koncem.

Majhno število registrov pomeni, da bo veliko vmesnih shranjevanj na sklad, pa tudi velika uporabljenost začasnih spremenljivk. Ker instrukcije za množenje in deljenje delujejo samo nad EAX registrom, bo moral ta ostati rezerviran za prav te operacije. Zaradi 2-operandnega formata bo v sami implementaciji instrukcij veliko premikanja med registri, kar povzroča dodatne zakasnitve. Poleg tega lahko obstajajo med zaporednimi instrukcijami odvisnosti v primeru, da naslednja instrukcija zavisi od rezultata prejšnje. S tem se pojavi krajša zakasnitev, ker se instrukciji ne moreta izvesti paralelno.

```
mov edx,[ebx*4]
mov eax,[edx]
```

Odvisnost med registrom edx in eax je mogoče premostiti s vstavitvijo dodatne, seveda uporabne, instrukcije med zgornji dve. Po drugi strani pa imajo novejši Pentium procesorji precej sofisticirano paralelno izvajanje, tako da stara pravila o soodvisnostnih zakasnitvah med instrukcijami ne držijo vedno. Ker lahko aritmetične operacije delujejo nad pomnilniškimi operandi, bomo to s pridom uporabili in s tem prihranili na uporabi registrov, ki so nujni pri drugih operacijah. Prav tako se bomo izognili shranjevanju registrov z uporabo kompleksnih naslavljanj. Nekatere instrukcije ob izvedbi modificirajo druge registre (mul,div), kar je potrebno upoštevati in zato te registre prej shraniti na sklad. Poleg registrov obstaja še 6 segmentnih registrov, ki v zadnjem

Register	**Dolžina**	**Uporaba**
eax	32 bitov	splošna
ebx	32 bitov	splošna
ecx	32 bitov	splošna
edx	32 bitov	kazalec na kodo
esi	32 bitov	kazalec na podatke
edi	32 bitov	splošna
ebp	32 bitov	kazalec na vrh sklada
esp	32 bitov	sistemski sklad

Tabela 13: Uporabljenost procesorjevih registrov znotraj interpreterja.

času v uporabniških aplikacijah vse bolj izgubljajo pomen, saj se koda izvaja v zaščitenem načinu, kjer uporaba segmentov ni eksplicitno zahtevana.

Segment	**Dolžina**	**Uporaba**
cs	32 bitov	koda
ds	32 bitov	podatki
es	32 bitov	podatki
ss	32 bitov	sklad
fs	32 bitov	podatki
gs	32 bitov	podatki

Tabela 14: Uporabljenost segmentnih registrov.

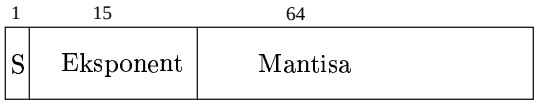
Skoki so v x86 realizirani dvokoračno. Prva instrukcija opravi primerjavo, druga pa, glede na rezultat, izvede skok na zahtevano lokacijo. Ukaza za primerjanje sta `cmp` in `test`, ki nastavita določene bite v registru zastavic. Vsi ukazi za skok se začnejo s črko j, kateri sledi relacija. Skok mora biti izveden, preden kakšna logična ali aritmetična operacija ponastavi bite v zastavicah.

V realizaciji instrukcij s plavajočo vejico se uporabljajo instrukcije za delo z realnimi števili. x86 procesor za to področje nima splošnonamenskih registrov ampak uporablja poseben sklad, na katerem so reflektirani registri. Vseh realnih registrov je 8 in so simbolično poimenovani od `st0` do `st7`, kot je to prikazano v tabeli 13. Njihova skladovna struktura je prikazana na sliki 19. Števila so na tem skladu zapisana v

Register	**Dolžina**	**Uporaba**
st0	80 bitov	splošna
st1	80 bitov	skupaj s st0
st2	80 bitov	skupaj s st0
st3	80 bitov	skupaj s st0
st4	80 bitov	skupaj s st0
st5	80 bitov	skupaj s st0
st6	80 bitov	skupaj s st0
st7	80 bitov	skupaj s st0

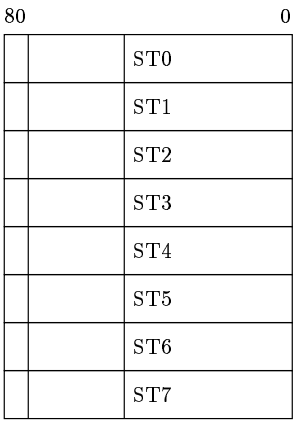
Tabela 15: Registri plavajoče vejice.

formatu IEEE-754. Ne glede na uporabljen tip, se števila, ko jih potisnemo na sklad, vedno pretvorijo v notranji 80-bitni format (slika 18).



Slika 18: Format števila s plavajočo vejico v razširjenem formatu.

Sklad za plavajočo vejico je implementiran v samem procesorju in se zato ne more enačiti z običajnim skladom v pomnilniku. Dostop do njega je namreč veliko hitrejši, kot dostop do lokacij pomnilnika. Operacije plavajoče vejice delujejo nad dvema operandoma in sicer nad prvim registrom `st0` in pomnilniško lokacijo, ali pa nad dvema registroma, od katerih je eden `st0`. Nujna uporaba prvega registra pomeni nove omejitve pri realizaciji s tovrstnimi instrukcijami. Delna kompenzacija se doseže z uvedbo paralelizma med instrukcije, ki delujejo nad celoštevilčnimi tipi in tistimi za realna števila. Ker sta enoti implementirani vsaka zase, x86 omogoča, da se instrukcije izvedejo sočasno.



Slika 19: Struktura registrov plavajoče vejice.

Register ST0 predstavlja vrh registrskega sklada, njegova uporaba pa je implicitna, ka-

dar v posamezni instrukciji ni vključen. Operacije brez operandov, v kolikor ni drugače specificirano, torej funkcionirajo nad tem registrom. Večina realnih instrukcij deluje tudi nad celoštevilčnimi operandi v pomnilniku, vendar so te bistveno počasnejše od tistih, ki delujejo samo nad realnimi števili. Zato je kljub krajši kodi priporočljivo vse celoštevilčne operande pretvoriti v realna števila in nato nad njimi izvajati operacije.

4.2.6 Implementacija virtualnega stroja

V interpreterju so uporabljeni vsi možni registri za dosego večje robustnosti, čeprav je pri takšnem številu registrov to precej oteženo. Za sklad se, kot je privzeto, uporablja register ESP, ki kaže na zadnji element na skladu, in EBP, ki kaže na začetek sklada. V implementaciji posameznih instrukcij se uporabljajo prav vsi registri, kar zahteva, da se nekateri registri shranijo na sklad, to pa v splošnem pomeni počasnejši sistem.
Implementacija instrukcij, ki delajo z registri, je na realnem procesorju z malim številom registrov težavna, še posebej v našem primeru, ko ima virtualni stroj 256 registrov, ciljni procesor pa le 6 uporabnih (eax, ebx, ecx, edx, esi, edi). Virtualnih registrov tako ne bo moč preprosto preslikati na realne registre in s tem zmanjšati kompleksnost. Registrski operand je znotraj ukaza specificiran z 1 zlogom, ki pove številko registra. To število je uporabljeno za odmik v registrski tabeli, ki se nahaja v pomnilniku. Pri vsakem ukazu, če ta operira nad katerimkoli registrom, je torej potrebno najprej referencirati ta register, da bi ga lahko uporabili.
Primer interpreterjeve kode, če register edx kaže na izvajajoč ukaz:

```
movzx eax,byte[edx+2]            ; eax - register
mov ebx,dword[registers+eax*4]   ; ebx - vrednost registra
```

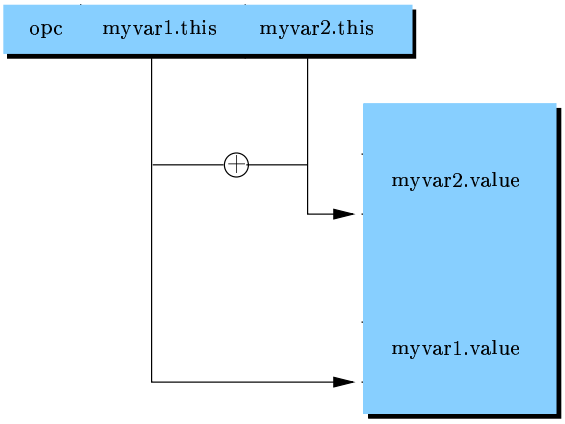
Da bi prebrali vrednost nekega registra, sta torej potrebna kar dva posega v pomnilnik, pri čemer ni nujno, da sta oba naslova že v predpomnilniku. Povrhu je v drugem dostopanju uporabljeno kompleksno naslavljanje, kar pomeni daljšo in počasnejšo instrukcijo. Enako je s pisanjem v registre, le da sta tam operanda zamenjana. V primeru virtualnih instrukcij, ki dostopajo do pomnilnika, pa bo stvar drugačna, saj

ne bo potrebnih posebnih pretvorb v pomnilniške naslove; takšne instrukcije torej ne bodo dosti počasnejše od svojih realnih ekvivalentov.

Instrukcije, ki implementirajo nestatične metode v razredih, so bolj ugodne za izvedbo na stroju, ki podpira pomnilniške operande in kompleksna naslavljanja. Pri prištevanju spremenljivke sestavljenega tipa integer drugi spremenljivki istega tipa ne bo potrebno uporabiti večjega števila registrov. Polje objekta bo namreč na določeni lokaciji v pomnilniku in ga bomo lahko direktno uporabili, brez nalaganja v register, ker ciljna arhitektura dopušča takšne načine naslavljanja. Takšna instrukcija bo precej hitrejša od zaporedja preprostejših instrukcij, ki bi opravile enako nalogo. Treba se namreč zavedati, da virtualni stroj simulira vsak ukaz z zaporedjem realnih ukazov. Vsaka operacija, ki se izvede na virtulnem stroju, mora biti interpretirana, za kar pa je potreben nek čas. Krajše in hitrejše instrukcije torej v celoti ne bodo nujno hitrejše, saj bodo izgubile veliko časa pri sami interpretaciji. Interpreter mora zato biti napisan karseda sofisticirano, da se pri režiji izgubi čim manj časa. Operacijsko kodo ukaza in operande je treba prebrati in skočiti na izvajanje tega ukaza. Najbolje bi bilo to storiti v enem koraku, vendar to ni mogoče, zato je korak realiziran z dvema instrukcijama. Prva prebere operacijsko kodo, druga pa skoči na zahtevano operacijo, ki jo operacijska koda določa. To se zgodi v 4 urinih periodah, kar je občutno bolje, kot če bi za isto stvar uporabili npr. stavek *switch*. Druga bistvena pohitritev je v tem, da je celotni interpreter napisan v zbirniku, kar zmanjša optimizacijske probleme pri prevajanju v primeru, če bi interpreter bil napisan v visokem programskem jeziku. Če imamo dve spremenljivki definirani na naslednji način:

```
myvar1 is integer
myvar2 is integer
```

in nad njima izvedemo seštevanje (vrednost druge prištejemo k prvi), je to ponazorjeno takole:



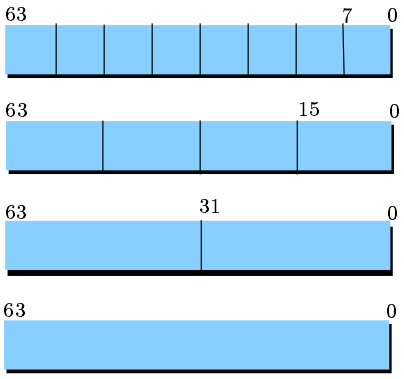
Slika 20: Primer seštevanja dveh tipov integer.

Operacijo bi v zbirnem jeziku zapisali `myvar1.add myvar2`, za kar bomo potrebovali zgolj eno instrukcijo.

Z namenom, da bi izgubili čim manj časa pri režiji interpreterja, kamor spada branje operacijske kode in operandov ter skok, se splača v vsaki interpretirani instrukciji izvesti karseda veliko realnih procesorskih ukazov. Instrukcije kompleksnih podatkovnih tipov so same po sebi dovolj zahtevne, da bodo izkoristile to dejstvo in v enem ukazu opravile čim zahtevnejše operacije. Primer tega bi lahko bil izračun faktoriele nekega realnega števila. Virtualni stroj s preprostimi ukazi bo za to nalogo potreboval velik nabor instrukcij in še zanko povrhu:

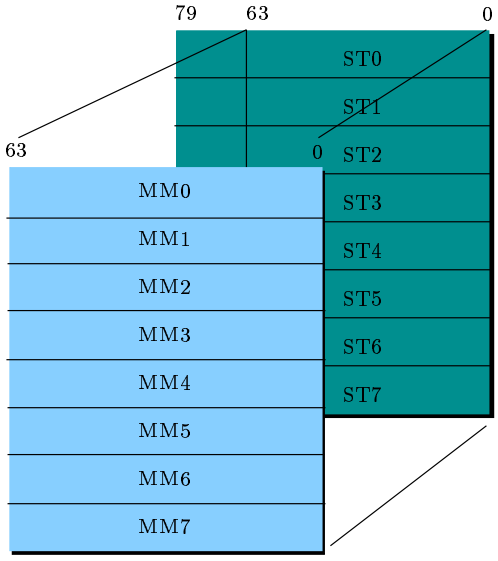
```
; funkcija fact(n) z uporabo registrov
move 1,reg2
cmp n,0
ifeq konec
move n,reg1
.loop:
     mul reg1,reg2
     dec reg1
     cmp reg1,0
ifneq .loop
```

Vsaka izmed instrukcij v zgornji funkciji mora, v kolikor se izvaja na virtualnem stroju, biti interpretirana. Če je skupaj 8 instrukcij, to pomeni 8 branj operacijske kode in operandov oz. 8 posegov v pomnilnik. Ob vsakem prebranem ukazu je potreben še skok na kodo, ki implementira instrukcijo v interpreterju. Čeprav so same operacije posameznih ukazov morda zelo preproste, bo izvajanje zaradi režije bistveno upočasnjeno. Pri izjemno preprostih instrukcijah, kot npr. `dec reg1`, se lahko celo zgodi, da bo sama interpretacija počasnejša kot celotna izvedba instrukcije. Pri operacijah nad kompleksnimi tipi, se lahko za pohitritev uporabijo dodatne razširitve, ki jih arhitektura x86 ponuja. Ena izmed teh je MMX nabor instrukcij, ki operirajo istočasno nad več celoštevilčnimi operandi. S pomočjo teh lahko vpeljemo delni paralelizem za izvajanje osnovnih operacij nad celoštevilčnimi tipi. MMX instrukcije operirajo nad enim 64-bitnim, dvema 32-bitnima, štirimi 16-bitnimi ali osmimi 8-bitnimi števili.



Slika 21: Tipi, nad katerimi operirajo MMX instrukcije.

Vsi podatkovni tipi so "pakirani" v 64 bitov, kar je tudi dolžina MMX registrov (slika 21). Slabost vsega je samo to, da se ti registri nahajajo na registrskem skladu plavajoče vejice, kar pomeni, da realnih in MMX operacij ne moremo opravljati sočasno. MMX registri za razliko od ST registrov funkcionirajo s poljubnimi MMX registrskimi operandi. Operacije sestojijo iz osnovnih aritmetičnih in logičnih operacij, ki se aplicirajo na pakirane podatkovne tipe. Struktura MMX registrov je prikazana na sliki 22.



Slika 22: Struktura MMX registrov.

Dostop do registrov je precej hitrejši kot dostop do predpomnilnika ali celo pomnilnika. V primeru našega virtualnega stroja to ne drži, saj je implementacija virtualnih registrov v pomnilniku. Sklicevanje na register ni torej nič drugega kot sklicevanje na določen pomnilniški naslov. Po drugi strani pa dostop do registrov kljub temu ni tako zamuden, ker se pri pogosti uporabi večina registrov tako ali tako nahaja v predpomnilniku do katerega je dostop precej krajši. Vseh virtualnih registrov je 256, vsak je dolg 4 zloge, kar pomeni, da celotna implementacija v pomnilniku ne zahteva več kot 1024 zlogov. To pa zagotavlja, pri današnjih velikostih predpomnilnikov, da bodo registri praktično ves čas v predpomnilniku.

Postopki klicanja procedur se od sistema do sistema razlikujejo. Vpliv na invokacijo metod ima tako samo arhitektura kot operacijski sistem. Arhitektura določa registre za naslove in instrukcije za klicanje, operacijski sistem pa vpelje še svoje postopke. Na x86 arhitekturi se klic procedure v zaščitenem modelu izvrši z ukazom `call naslov`. Ta na sklad shrani vrednost trenutnega programskega števca (register EIP) in vanj zapiše vrednost, ki je podana z operandom `naslov`. Ko se ukazi v klicani proceduri

zaključijo, se mora tok izvajanja prenesti nazaj, kjer je bil prekinjen. To se stori z instrukcijo `ret`, ki s sklada prebere prejšnjo vrednost EIP registra in se tako vrne na mesto, kjer je bila procedura poklicana.

Natančen postopek pošiljanja in uporabe parametrov je prepuščen operacijskemu sistemu. Konvencije sistema določajo ali se parametri pošiljajo v registrih ali na skladu. Če se pošiljajo v registrih je potrebno vedeti, v katerih registrih, če pa se pošiljajo na skladu so potrebni naslovi. Linux, za katerega je virtualni stroj implementiran, pošilja parametre funckij na skladu in sicer v obratnem vrstnem redu. Funkcija s prototipom `void myfunc(int i,int j,char k)` ima na skladu obliko:

Slika 23: Sklad s parametri.

Prevajalnik ob klicu procedure na sklad pošlje samo parametre, instrukcija `call` pa implicitno shrani še vrednost programskega števca. Zato do prvega parametra dostopamo na lokaciji, ki je za 4 zloge nižje na skladu (če privzamemo 32 bitne naslove). Dostop do prvega parametra je torej na naslovu `[esp+4]`. Lokacije vseh parametrov so:

```
[esp+4] za i
[esp+8] za j
[esp+12] za k
```

Pri skladovnih parametrih je potrebno poudariti, da vsi tipi, katerih dolžina je manjša od 32 bitov, zasedajo na skladu enako prostora, to je 32 bitov ali 4 zlogi. Pri dostopanju je takšne parametre potrebno pretvoriti z ničelno ali predznakovno razširitvijo.

4.2.7 Implementacija objektov

Ob klicu metode new, definirane v razredu, pride do ustvaritve objekta na naslednji način:

1. Alokacija potrebnega pomnilniškega prostora. Objekt je v pomnilniku predstavljen s svojim stanjem, ki je ekvivalentno zaporedju objektovih spremenljivk. Prostor za predstavitev celotnega objekta je torej prostor, potreben za celotno stanje objekta, ki pa je lahko v primeru dedovanja širše, kot stanje osnovnega objekta. Po alokaciji pomnilnika se kazalec priredi spremenljivki objekta "this", preko katere je objekt mogoče referencirati iz drugih delov kode.

2. Število referenc na ustvarjen objekt se postavi na nič, kar pomeni, da se na ta objekt še nihče ne sklicuje.

3. Izvede se telo konstruktorja, s katerim je bil objekt ustvarjen.

Do metod instanciranega razreda se lahko dostopa izključno preko reference. Zato se vsi abstraktni tipi, torej tisti, ki so predstavljeni v obliki objektov, imenujejo tudi referenčni tipi. Instrukcije, ki implementirajo metode razredov, morajo poznati natančno število in strukturo vseh polj v razredu, da bi lahko do njih dostopale in nad njimi izvajale operacije. Na implementacijskem nivoju namreč ni več nikakršne podatkovne abstrakcije. Vsi principi enkapsulacije izginejo, saj se dela samo z naslovi znotraj obstoječega objekta. Pri jezikih, ki neprestano ustvarjajo veliko število objektov, je eksplicitno brisanje le-teh za programerja neugodno in zamudno. Če objekt ni zbrisan, ostane v pomnilniku in po nepotrebnem zaseda prostor, po drugi strani pa, če ga eksplicitno zbrišemo in do njega dostopamo, pride do izjeme. V ta namen prepustimo brisanje objektov virtualnemu stroju, ki implementira strategijo "pobiranja smeti". Ko objekt ni več uporabljan, torej ko nanj ne obstaja nobena referenca več, ga lahko virtualni stroj odstrani iz pomnilnika, da se lahko na njegovo mesto shrani nov objekt.

Avtomatsko brisanje lahko implementiramo s štetjem referenc na objekt. Vsak objekt ima skrito spremenljivko, ki vsebuje število referenc na sebe. Reference izvirajo iz lokalnih in spremenljivk drugih objektov. Število referenc se poveča vedno, kadar se nek objekt priredi drugemu. V kodi

```
String ime="java";
String novo=ime;
```

mora funkcija drugega prireditvenega operatorja povečati števec referenc na objekt `ime`. Ko se objektu `novo` priredi vrednost null, se števec zmanjša za 1. Ko je vrednost števca enaka 0, kar pomeni, da ni več nobene reference na ta objekt, se objekt zbriše. Čeprav je štetje referenc preprosto, lahko pripelje do težav pri cikličnih strukturah. Če npr. objekt A referencira objekt B in obratno, potem ne bo število referenc nikoli doseglo vrednosti 0 in noben izmed objektov ne bo sproščen iz pomnilnika.

Druga metoda je "označi in počisti". Vsak ustvarjen objekt vključuje meta informacijo o razredu, s katerim je bil instanciran. Tako se lahko ugotovi, katera polja objekt vsebuje, njegovo velikost in posledično tudi kje se objekt konča in začne drugi. Ko se začne proces čiščenja se pregledajo reference vseh objektov. Pri vsakem objektu se preveri ali je objekt označen; če to velja, potem se objekt preskoči in pogleda se naslednji objekt. V kolikor objekt ni označen, se ta skopira na novo lokacijo in se označi, da je bil skopiran. Nato se posodobijo vse reference v objektih na objekte, ki so sedaj na novi lokaciji. Nato se na novo generirajo samo objekti, ki se še uporabljajo, ostali se odstranijo.

Metoda ima precej pomanjkljivosti, največja je počasnost celotnega postopka. Ko je v pomnilniku veliko objektov, postane pregledovanje vseh struktur časovno zahtevno, kar bistveno upočasni celoten sistem. Izboljšava algoritma je v lahko v tem, da ne uporablja kopiranja vseh objektov, ampak jih samo premika. Druga rešitev je, da se čiščenje zažene samo, kadar je sistem v stanju mirovanja in ne opravlja časovno zahtevnih opravil. V tem primeru se lahko čiščenje zažene kot nit z nizko prioriteto [5].

Koristno je vpeljati tudi "življenjsko dobo" objektov. Nekateri namreč živijo npr. skozi celotno aplikacijo, drugi pa postanejo neuporabni takoj po njihovi ustvaritvi. Primer slednjega je konkatenacija znakovnih nizov, pri kateri se pri vsaki operaciji '+' ustvari nov objekt kot stranski učinek:

```
"to" + " je" + " niz"
```

Objekt, ki se ustvari po prvem +, je stranski učinek in je zato lahko takoj odstranjen. Nekateri programi zelo hitro alocirajo podatkovne strukture in jih posledično uničujejo. To velja še posebej za funkcijske jezike, kjer se novi podatki ustvarjajo neprestano. Nenehna alokacija in brisanje sta v takšnih primerih zelo neugodna, zato je smiselno imeti prost zvezen pomnilniški prostor, iz katerega se alocirajo strukture. V algoritmu je v ta namen uporabljena vrednost *next*, ki vsebuje naslednjo prosto lokacijo in *limit*, ki pove konec alokacijskega prostora. Algoritem za alokacijo strukture velikosti N je:

1. Pokliči funkcijo za alokacijo.
2. Testiramo če velja `next + N < limit`. Če pogoj ni izpolnjen, se pokliče čistilec.
3. Rezultat=`next`
4. Clear `M[next], M[next+1,...,M[next+N-1]`
5. `next=next+N`
6. Vrnitev iz alokacijske funkcije.
7. Rezultat je nov naslov za strukturo.

V kolikor alociramo prostor za več struktur, se lahko koraka 2 in 5 delita za vse alokacije, s čimer pridobimo na hitrosti. Izvedbo alokacije strukture in njenega čiščenja lahko tako realiziramo s štirimi instrukcijami.

Čistilec mora biti sposoben operirati nad vsemi tipi podatkov, ki so lahko polja, seznami in druge podatkovne strukture. Za vsako strukturo mora vedeti točno število

polj in tip polj; ta je lahko primitiven ali referenca na nek drug objekt. Za močno tipizirane jezike je za identifikacijo posameznih objektov smiselno imeti poseben kazalec, ki kaže na deskriptor objekta oz. razred, iz katerega je možno razbrati velikost objekta in pozicije vseh polj, ki so sami reference na druge objekte. Pri objektno orientiranih jezikih ta strategija ne doprinese nobene težavnosti, saj je kazalec na deskriptor tako ali tako potreben za klicanje dinamičnih metod. Deskriptor, ki se pošlje kot argument funkciji za alokacijo, se ustvari v času prevajanja iz semantične informacije na podlagi tipa, s katerim je objekt ustvarjen.

4.2.8 Primerjava instrukcij

Skupine instrukcij v razredih `core, stack` in `branch` so zelo podobne tistim na realnem procesorju. Velika večina jih operira nad registri in takojšnjimi operandi, izjema so le ukazi za dostop do pomnilnika. Ti omogočajo branje in pisanje preko treh različnih tipov naslavljanj. S stališča kompleksnosti spadajo te instrukcije v RISC topologijo, kljub temu, da imajo variabilno dolžino. Instrukcije, ki implementirajo metode posameznih razredov, pa so zaradi kompleksnih in visokonivojskih operacij, tipične za CISC arhitekturo. Tudi njihova dolžina variira od same operacijske kode, tj. 2 zlogov, do 16 zlogov.

Za realizacijo statičnih in ostalih instrukcij iz procesorskih enot, je običajno potrebnih nekaj realnih ukazov. Če je v povprečju za realizacijo posamezne virtualne instrukcije potrebnih n realnih instrukcij, je preslikava v interpreterju podana z razmerjem 1:n. Primer preslikave na instrukciji za seštevanje dveh registrov bil bil naslednji:

```
movzx eax,byte[edx+2]           ; eax - register 1
movzx ebx,byte[edx+3]           ; ebx - register 2
mov ecx,dword[registers+eax*4]  ; ecx - vrednost registra 1
add dword[registers+ebx*4],ecx  ; seštetje
add edx,4
jmp continue
```

Zaporedje realnih ukazov se preslika v virtualno instrukcijo z dvema operandoma:

```
core.add(register r1,register r2)
```

Za operacijo seštevanja so potrebni samo prvi štirje ukazi, ki najprej pretvorijo številko registra v pomnilniško lokacijo, na kateri se register nahaja nato pa se v četrtem ukazu vrednost enega registra dejansko prišteje drugemu. Zadnja dva ukaza sta potrebna za režijo sistema – povečanje programskega števca za dolžino instrukcije in skok na nov ukaz. Pri uporabi virtualnih registrov se izkaže veliko število različnih naslavljanj za prednost, saj lahko posamezen register referenciramo v enem samem ukazu.
Enak ukaz se z drugimi parametri lahko precej poenostavi. V primeru seštevanja registra in 32-bitnega števila v takojšnjem operandu dobimo:

```
movzx eax,byte[edx+2]
mov ebx,[edx+3]
add dword[registers+eax*4],ebx
add edx,7
jmp continue
```

Zgornja koda se preslika v virtualni ukaz

```
core.add(oct4 o,register r2),
```

ki je po številu zlogov daljši, a se bo hitreje izvedel. Iz tega se da videti, kakšne zmogljivosti lahko pri posameznem ukazu pričakujemo. Nedvomno bo ukaz tekel počasneje, kot njegov realni ekvivalent vendar se da z dobro optimizacijo ta razlika zmanjšati, še posebej, če gre za zahtevne ukaze, ki naenkrat izvršijo celo skupino realnih ukazov.

5. Uporaba in simulacija

5.1 Prevajanje in format datotek

Ko se napiše izvorna koda programa, je to potrebno prevesti v obliko, ki je primerna za izvajanje na virtualnem stroju. Postopek prevajanja in izvajanja bi se lahko izvedel istočasno, vendar bi v tem primeru bila za to potrebna celotna izvorna koda programa. Tega ne želimo, zato se program prevede prej, nato pa se lahko distribuira samo binarna verzija.

Program prevedemo z uporabo zbirnika na naslednji način:

```
as vhodna_datoteka.s
```

ki na izhodu producira binarno datoteko z imenom `vhodna_datoteka.code`. V kolikor želimo berljiv zapis, lahko omogočimo izpis celotne vsebine programa v zložni obliki, ki izpisuje informacije o generirani vsebini datoteke na standardni izhod. Po vrstnem redu se izpisujejo: vhodi relokacijske tabele, podatki in nazadnje še programska koda v skupinah po 1 zlog. Zgornji ukaz uspe samo v primeru, če v izvornem programu ni napak. Če je v program vključena datoteka z definicijo neke enote, mora ta datoteka biti dosegljiva in berljiva. Specifično za našo arhitekturo se uporabljajo definicije v naslednjih datotekah:

- `registers.def` za definicije registrov,
- `core.def` za definicijo ALU in podatkovne enote,
- `branch.def` za vejitveno enoto,
- `stack.def` za sklad in
- `system.def` za sistemske ukaze.

To so osnovni razredi s statičnimi metodami, ki se ne ustvarjajo dinamično med izvajanjem. Razredi, ki deklarirajo kompleksne tipe pa so naslednji:

- `integer.def` za 32 bitni celoštevilčni predznačeni tip,
- `short.def` za 16 bitni celoštevilčni predznačeni tip,
- `byte.def` za 8 bitni celoštevilčni predznačeni tip,
- `string.def` za niz znakov.

Datoteka, ki jo zbirnik pri prevajanju generira, ima strukturo, ki je podana v tabeli 14.

Naslov	**Dolžina**	**Vsebina**	**Predstavitev**
0	4 zlogi	število vstopov v relokacijski tabeli	int nEnt
4	4 zlogi	velikost podatkov v zlogih	int dSize
8	4 zlogi	velikost kode v zlogih	int cSize
12	4*nEnt	relokacijska tabela	int relocTable[nEnt]
12+4*nEnt	dSize	podatki	char data[dSize]
12+4*nEnt+dSize	codeSize	koda	char code[cSize]

Tabela 16: Format in polja generirane datoteke.

5.2 Izvedba programa

Ko je program iz izvorne kode preveden v vmesno binarno obliko, je primeren za izvajanje na virtualnem stroju. To naredimo tako, da virtualnemu stroju podamo ime datoteke s programom, katerega želimo izvesti:

```
run ime_datoteke.code
```

Inicializacijski del virtualnega stroja najprej preveri format datoteke in javi napako, če ta ni pravi. Nato se prebere glava s podatki in preostala vsebina datoteke. Temu sledi nalaganje podatkov in kode v pomnilnik ter relokacija le-te. Tako naložen program je pripravljen na izvajanje. Kontrola se s tem preda interpreterju, ki začne izvajati instrukcije programa. To se nadaljuje dokler ni zahtevan izhod iz programa z instrukcijo `sys.exit`.

5.3 Primeri

Namen naslednjih izsekov kode je prikazati način, na katerega je mogoče posamezno stvar napisati za našo arhitekturo in jo sprogramirati v zbirniku. Prvi primer podaja uporabnost registrov in aritmetičnih instrukcij nad dvema in tremi operandi v rekurzivnih funckijah. Za testni primer naj služi izračun n-tega člena Fibonaccijevega zaporedja, ki je določen:

$$Fib(n) = \begin{cases} 1 & ;\ \text{n=1 ali 2} \\ Fib(n-1) + Fib(n-2) & ;\text{n>2} \end{cases}$$

Funkcijo v C-ju zapišemo na naslednji način:

```
int fib(int n){
  if(n==1 || n==2) return 1;
  return fib(n-2)+fib(n-1);
}
```

V zbirniku bomo N-ti člen izračunali prav tako s funkcijo `fib`, ki na skladu sprejme število n in v registru r2 da končni rezultat. Ko prevedemo program v C-ju, se zgornja funkcija preslika v kodo, katere zbirna oblika je prikazana v levem stolpcu. Na desni strani je ekvivalentna funkcija napisana za to arhitekturo:

```
fib:                            fib:
push   ebp                      stk.pop $r1       // n
mov    esp,ebp                  b.ifeq $r1,1,res1 // 1
mov    0x8(ebp),eax             b.ifeq $r1,2,res1 // 2
push   esi                      c.sub 1,$r1
lea    0xffffffff(eax),esi      stk.push $r1      // shrani r1
cmp    $0x1,esi                 stk.push $r1      // na sklad
push   ebx                      c.call fib
ja     8048418 <fib+0x18>       c.move $r2,$r3
mov    $0x1,eax                 stk.pop $r1       // restavriraj r1
jmp    8048432 <fib+0x32>       stk.push $r3      // shrani r3
nop                             c.sub 1,$r1       // -2
sub    $0xc,esp                 stk.push $r1      // na sklad
sub    $0x2,eax                 c.call fib
push   eax                      stk.pop $r3       // restavriraj r3
call   8048400 <fib>            c.add $r2,$r3,$r2 // r2 - rezultat
mov    eax,ebx                  c.return
mov    esi,(esp,1)              res1: c.move 1,$r2
call   8048400 <fib>            c.return
add    eax,ebx
mov    ebx,eax
lea    0xfffffff8(ebp),esp
pop    ebx
pop    esi
pop    ebp
ret
lea    0x0(esi),esi
```

Razvidno je, da je funkcija na desni zaradi uporabe operacij nad tremi registrskimi operandi krajša in s tem kompaktnejša, čeprav sploh ni optimizirana.

Naslednji primeri prikazuje uporabo kompleksnih tipov, pri katerih največ pridobimo, saj obstajajo za njihovo manipulacijo posebne instrukcije, definirane z metodami v razredih. Do široke uporabe standardnih tipov v visokih programskih jezikih pride zagotovo v aritmetičnih izrazih. V ta namen bomo izračunali vrednost izraza:
`int A = (2 * B + C) / (D + 4).`
Za primer je bil javin prevajalnik, ki je generiral kodo v levem stolpcu. Na desni strani se ponovno nahaja koda v zbirniku, tokrat z uporabo sestavljenih tipov:

```
bipush 33                B.new 33
istore_0                 C.new 44
bipush 44                D.new 11
istore_1                 t0.new B
bipush 11                t0.mul 2
istore_2                 t0.add C
iconst_2                 t1.new D
iload_0                  t1.add 4
imul                     t0.div t1
iload_1                  A.new t0
iadd
iload_2
iconst_4
iadd
idiv
istore_3
iload_3
ireturn
```

Javin virtualni stroj temelji na skladu, zato so potrebne zaporedne instrukcije za potiskanje na sklad in nato shranjevanje v lokalne spremenljivke. V naši arhitekturi je to realizirano s konstrukcijo objektov za vsako spremenljivko in dveh začasnih spremenljivk, ki služita za shranitev delnih rezultatov. Ker metode kompleksnih tipov operirajo tudi nad registrskimi operandi, bi za to lahko uporabili tudi registre in na ta način obšli ustvarjanje začasnih objektov.

Naslednji primer prikazuje vejitvene instrukcije nad kompleksnimi tipi. Izračunali bomo vsoto vseh sodih in lihih števil v zaporedju n števil. Funckija oz. stavki, ki opravijo to nalogo, so v C++ zapisani na naslednji način:

```
int s=0;   // vsota sodih
int l=0;   // vsota lihih
for(int i=0;i<=10;i++){
    if(i % 2 == 0) s+=i; else l+=i;
}
```

Praktično identično je algoritem zapisan v javi. Na levi strani se nahaja kod, ki ga je generiral GNU prevajalnik:

```
movl   $0x0,0xfffffffc(%ebp)
movl   $0x0,0xfffffff8(%ebp)
movl   $0x0,0xfffffff4(%ebp)
nop
cmpl   $0xa,0xfffffff4(%ebp)
jle    80484f4 <main+0x24>
jmp    8048518 <main+0x48>
mov    0xfffffff4(%ebp),%eax
and    $0x1,%eax
test   %eax,%eax
jne    8048508 <main+0x38>
mov    0xfffffff4(%ebp),%edx
lea    0xfffffffc(%ebp),%eax
add    %edx,(%eax)
jmp    8048510 <main+0x40>
mov    0xfffffff4(%ebp),%edx
lea    0xfffffff8(%ebp),%eax
add    %edx,(%eax)
lea    0xfffffff4(%ebp),%eax
incl   (%eax)
jmp    80484ec <main+0x1c>
nop
```

```
n.new    10 // n=10
s.new    0  // soda
l.new    0  // liha
sum:     n.store $r5
c.and    1,$r5
b.ifz    $r5,sodo
l.add    n
b.to     liho
sodo:    s.add n
liho:    n.sub 1
n.ifneq  0,sum
```

Rezultat izvajanja našega programa se nahajata v spremenljivkah `n` in `l`. Naslednji izhod je pri enakem algoritmu dal Javin prevajalnik. Koda se začne na levi strani in nadaljuje na desni:

```
iconst_0
istore_0
iconst_0
istore_1
iconst_0
istore_2
goto 29
iload_2
iconst_2
irem
ifne 22
```

```
iload_0
iload_2
iadd
istore_0
goto 26
iload_1
iload_2
iadd
istore_1
iinc 2 1
iload_2
```

```
bipush 10
if_icmple 9
```

Iz primerjave vseh treh kod je razvidno, da primerjalne operacije nad kompleksnimi tipi privedejo do občutno kompaktnejše prevedene kode. Največja razlika se vidi pri Javini kodi, ki uporablja skladovni stroj, pri čemer mora za vsako operacijo na sklad dati operand ali dva. Z razmeroma velikim številom instrukcij se izgubi precej časa za njihovo interpretacijo, saj se koda izvaja na virtualnem stroju.

6. Sklep

Z razlogi zmanjšati semantično razliko med splošnim visokim programskim jezikom in ukazi na strojnem nivoju, kot jih vidi programer v zbirnem jeziku, smo se odločili razviti novo arhitekturo. Struktura te naj bi pomagala premostiti težave, ki se pojavljajo pri prevajanju v strojni kod in omogočila enotnejšo preslikavo med omenjenima nivojema. Generiran kod programa naj bi s temi predpostavkami bil kompaktnejši in lažji za avtomatsko generiranje v procesu prevajanja. Temu cilju smo se najbolj približali z uvedbo razmeroma velikega števila registrov, skupaj jih je 256, trioperandnih aritmetičnih in logičnih instrukcij, ki operirajo nad temi registri in nenazadnje z novim pristopom kapsulacije podatkov na strojnem nivoju. S tem smo želeli predstaviti najbolj uporabljane kompleksne oz. sestavljene tipe na nivoju procesorja in sicer preko posebnih instrukcij, ki omogočajo manipulacijo teh struktur. Na ta način smo se izognili morebitni vpeljavi teh istih podatkovnih tipov v samem programskem jeziku, ki bi to arhitekturo uporabljal za cilj prevajanja.

Z velikim številom registrov je pri prevajanju mogoče aplicirati algoritme barvanja grafov za računanje odvisnosti med posameznimi registri in tako doseči optimalno alokacijo registrov za vmesne rezultate ali spremenljivke. Zaradi objektnega okolja delovanja bi bilo koristno implementirati avtomatsko čiščenje, za kar je bil v poglavju o virtualnem stroju naveden eden izmed množice algoritmov. Zaradi formalne zgradbe je bila uvedena posebna struktura procesorja, tako da je vsaka tipična skupina instrukcij v svoji enoti. Enoti, ki sta nepogrešljivi sta ALE in podatkovna enota. Prva je odgovorna za aritmetične in logične instrukcije, druga pa za prenos podatkov v in iz pomnilnika. Poleg tega smo implementirali enoto sistemskega sklada, ki upravlja s to podatkovno strukturo in enoto vejitev, v kateri se nahajajo ukazi za pogojne in brezpogojne vejitve. Enote abstraktnih podatkovnih tipov so integer, short, byte in string, nad katerimi je mogoče izvajati najosnovnejše operacije tako, da nad objekti v pomnilniku kličemo ustrezne metode. Arhitekturo bi v celoti specificirali takole:

- 32 bitna širina naslovnega vodila,
- 32 bitna širina registrov,
- 256 splošnonamenskih registrov,
- variabilni format instrukcije s 16 bitno operacijsko kodo,
- 2 in 3 operandne splošne instrukcije,
- 5 vrst naslavljanja pomnilnika in
- instrukcije kompleksnih tipov

Dodatne razširitve, kot npr. razredi za visokonivojske programske konstrukte in še kompleksnejše podatkovne tipe se lahko prav tako izvedejo brez večjih težav. Sama arhitektura in njeni atributi so definirani dokaj formalno, z uvedbo razredov in v njih definiranih metod in pripadajočih spremenljivk. V primeru razširitve ali kakršnihkoli sprememb arhitekture je potreben zgolj poseg v definicijske strukture, kjer se lahko spreminja jedro arhitekture ali zgolj manjši atributi. Dinamično je bil namreč zasnovan format instrukcij, kar pomeni, da se ta lahko spreminja kakor želimo, od operacijske kode do vseh pripadajočih operandov. Na podoben način so bili zasnovani registri, katerih imena lahko poljubno spreminjamo in jih uporabljamo za karkoli ali čisto specifično.

Za programiranje je bil v okviru tega diplomskega dela izdelan zbirnik, ki izvorno kodo programa prevede v zložno obliko, primerno za izvedbo na virtualnem stroju. Za ta namen je bila implementirana verzija virtualnega stroja za operacijski sistem Linux na x86 ciljni arhitekturi, katerega podrobno zgradbo smo opisali v četrtem poglavju, njegovo uporabo in primere programiranja pa v petem poglavju. Formalno smo programski model, kot tudi celotno arhitekturo za virtualni stroj podali v drugem poglavju tega dela, kjer smo se osredotočili tudi na obstoječe RISC in CISC arhitekture

in izpeljali strukturo naše arhitekture.

Tako v zbirniku kot v samem virtualnem stroju so možne razširitve. V zbirniku je sintaktične spremembne, zaradi modularne zgradbe, mogoče aplicirati dokaj preprosto. Za spremembe in razširitve v virtualnem stroju je najbolj odprt interpeter in sama struktura stroja.
Za dodatne razširitve na arhitekturi se odpira veliko možnosti. Samoumevno bi bilo dodajanje novih, še splošnejših abstraktnih tipov, med katere v prvi vrsti spadajo eno in večdimenzionalna polja, množice in seznami, pa tudi druge podatkovne strukture. Za čimvečjo učinkovitost se splača definirati ekvivalente višjenivojskim programskim konstruktom, kot so zanke, izrazi in bloki le-teh.

Literatura

[1] WWW Computer Architecture Page, *http://www.cs.wisc.edu/ arch/www/*

[2] MIT Computer Architecture Group Home Page, *http://www.cag.lcs.mit.edu/*

[3] John L. Hennessy & David A. Patterson, *Computer architecture: A quantitive approach*, Morgan Kaufmann, 1996.

[4] Harvey G. Cragon, *Computer Architecture and Implementation*, Cambridge University Press, 2000.

[5] Jon Meyer & Troy Downing, *JAVA Virtual Machine*, O'Reilly, 1997.

[6] Peter M. Kogge, *The architecture of symbolic computers*, McGraw-Hill, 1991.

[7] Irv Englander, *The Architecture of Computer Hardware and Systems Software*, John Wiley & Sons, Inc., 2000.

[8] Veljko Milutinovič, *200 MHz RISC Microprocessor*, IEEE Computer Society Press, 1997.

[9] Dušan Kodek, *Arhitektura računalniških sistemov*, BIT-TIM, 2000.

[10] Andrew W. Appel, *Modern compiler implementation in Java*, Cambridge, 1997.

[11] John R. Levine, Tony Mason, Doug Brown, *Lex & yacc*, O'Reilly & Associates, Inc., 1992.

[12] Damjan Zazula, *Operacijski sistemi. Maribor: Fakulteta za elektrotehniko, računalništvo in informatiko*, 1997.

[13] Karl E. Grosspietsch, *Unorthodox computer architectures*, 2002.

[14] Pierre America, *Designing an Object-Oriented Programming Language with Behavioural Subtyping*, 1991.

[15] Ole Lehrmann Madsen, *Virtual Classes and Their Implementation*, 2001.