Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Benjamin Kastelic

Pretvornik med SIC/XE in Intel Pentium x86 zbirno kodo

diplomsko delo

univerzitetni študijski program prve stopnje
računalništvo in informatika

Mentor: doc. dr. Tomaž Dobravec

Ljubljana 2012

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00042/2012
Datum: 13.04.2012

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **BENJAMIN KASTELIC**

Naslov: **PRETVORNIK MED SIC/XE IN INTEL PENTIUM X86 ZBIRNO KODO**

SIC/XE TO INTEL PENTIUM X86 ASSEMBLY LANGUAGE TRANSLATOR

Vrsta naloge: Diplomsko delo univerzitetnega študija prve stopnje

Tematika naloge:

SIC/XE je hipotetičen računalnik, ki je bil zasnovan z namenom ponazoritve osnovnih konceptov sistemske programske opreme (zbiranje, povezovanje, nalaganje). Ker pa je SIC/XE zgolj hipotetičen računalnik, lahko programsko kodo, napisano v njegovem zbirnem jeziku, poženemo le v simulatorjih. V diplomskem delu preučite možnost pretvorbe SIC/XE zbirne kode v prevedljivo in izvršljivo Intel Pentium x86 zbirno kodo. V ta namen najprej preučite lastnosti obeh procesorjev, način dela s pomnilnikom in registri, načine naslavljanja in podobno. Preučite in opišite tudi vse ukaze zbirnega jezika za SIC/XE in primerno veliko podmnožico ukazov zbirnega jezika za Intel Pentium x86. Opišite način pretvorbe med omenjenima zbirnima jezikoma ter izdelajte program, ki bo to pretvorbo dejansko izvajal.

Mentor:

doc. dr. Tomaž Dobravec

Fakulteta za računalništvo in informatiko
UNIVERZA V LJUBLJANI
2

Dekan:

prof. dr. Nikolaj Zimic

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Benjamin Kastelic, z vpisno številko **63090024**, sem avtor diplomskega dela z naslovom:

Pretvornik med SIC/XE in Intel Pentium x86 zbirno kodo

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom doc. dr. Tomaža Dobravca,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 18. septembra 2012 Podpis avtorja:

Zahvaljujem se mentorju doc. dr. Tomažu Dobravcu za vse nasvete in pomoč ter družini, ki me je spodbujala pri delu.

Kazalo

Povzetek

Glavni cilj te diplomske naloge je bil izdelati pretvornik zbirnega jezika SIC/XE v zbirni jezik arhitekture Intel Pentium x86 ter se seznaniti z obema zbirnima jezikoma. Spoznali smo se s koncepti prevajalnikov ter njihovo zgodovino razvoja. Omenili smo tudi vrsto izvorno-izvornih pretvornikov, med katere spada izdelek te diplomske naloge. Podrobno smo se seznanili z arhitekturama računalnikov SIC ter Intel Pentium ter spoznali njun zbirni jezik. Izdelali smo prevajalnik ter spoznali njegovo zgradbo. Na kratko smo predstavili tudi programska orodja, s katerimi smo si olajšali razvoj. Kot dodatek smo implementirali še nekaj ukazov, ki olajšajo delo z izpisovanjem ter branjem podatkov.

Ključne besede

prevajalniki, pretvornik, SIC, SIC/XE, Intel, Pentium, x86, zbirni jezik

Abstract

The main goal of this thesis was to develop a SIC/XE to Intel Pentium x86 assembly language translator and to familiarise ourselves with both assembly languages. We learned about the concepts of compilers and their history of development. We mentioned a type of source-to-source compilers, one of which is a product of this thesis. We learned in details about the SIC and Intel Pentium x86 computer architecture and their assembly languages. We developed a translator and learned about its structure. We presented the software tools which were used during the development. We also implemented a couple of additional instructions which make printing and reading data a bit easier.

Keywords

compilers, translator, SIC, SIC/XE, Intel, Pentium, x86, assembly language

Poglavje 1

Uvod

Programerji dandanes povečini pišejo programe v visokonivojskih jezikih, kot so na primer C, Java, C# in podobni. Računalnik, oziroma bolje rečeno centralno procesna enota (v nadaljevanju CPE) teh programov ne zna izvajati, zato je treba vso izvorno kodo pretvoriti v jezik, ki ga CPE razume. To nalogo opravljajo interpreterji ali prevajalniki. Čeprav postajajo prevajalniki čedalje bolj kompleksni, se njihova osnovna zgradba ne razlikuje veliko od prvih prevajalnikov.

Namen te diplomske naloge je bil razviti prevajalnik ali bolje rečeno pretvornik, ki bi vhodni program, napisan v zbirnem jeziku hipotetičnega računalnika SIC, pretvoril v zbirni jezik računalnika Intel Pentium x86. Pri predmetu Sistemska programska oprema smo napisali veliko programov za računalnik SIC, ki smo jih lahko testirali na namenskem virtualnem stroju. To je bilo odlično za testiranje delovanja, še vedno pa ni tisto pravo. Zato se je porodila ideja, da bi se naredil prevajalnik, ki bi prevajal že napisane programe za neki resnični računalnik. Tako bi se lahko prepričali, da so naši programi tudi dejansko uporabni, hkrati pa bi se tudi spoznavali z novim zbirnim jezikom.

Najprej se bomo v poglavju 2 na splošno seznanili s prevajalniki ter njihovo zgradbo. Nato bomo v poglavjih 3 in 4 podrobno opisali arhitekturo hipotetičnega računalnika SIC ter arhitekturo računalnika Intel Pentium x86.

Tako bomo pridobili dovolj znanja, da se bomo lahko v nadaljevanju spoznali s strukturo izdelanega pretvornika. Še prej pa si bomo podrobno pogledali ukaze zbirnega jezika SIC ter podmnožico ekvivalentnih ukazov zbirnega jezika Intel.

Poglavje 2

Prevajalniki

Danes pod besedo prevajalnik razumemo računalniški program oziroma skupek programov, katerih naloga je, da pretvorijo izvorno kodo, ki je napisana v enem programskem jeziku, v neki drug programski jezik. Ponavadi se prevaja iz visokonivojskih jezikov v strojno kodo, ki se lahko takoj izvaja na procesorju, lahko pa se tudi prevaja iz enega visokonivojskega jezika v drugega, iz visokonivojskega v zbirni jezik, iz zbirnega jezika v drug zbirni jezik, iz zbirnega nazaj v visokonivojski jezik in še bi lahko naštevali.

Namesto da se celoten program prevede v ciljni jezik in se šele nato požene, se lahko celoten izvorni program interpretira. Interpretiranje oz. tolmačenje je postopek, pri katerem se vrstice izvornega programa zaporedoma prevaja in sproti izvaja. Znani predstavniki interpretiranih jezikov so PHP, Perl, Python ...

Obstajajo pa tudi taki programski jeziki, ki ob prevajanju tvorijo t.i. vmesno kodo (*bytecode*), ki se lahko izvaja na različnih arhitekturah. Da lahko dosežemo tako neodvisnost, pa potrebujemo namenski virtualni stroj (*virtual machine*), ki izvaja strojne ukaze vmesne kode. Primer takega jezika je Java, njen pripadajoči virtualni stroj pa se imenuje JVM (*Java Virtual Machine*).

2.1 Kratka zgodovina

V zgodnjih dneh računalništva so se programi v večini pisali v zbirnih programskih jezikih. Z razvojem vedno hitrejših in kompleksnejših procesorjev je to opravilo postalo časovno zelo potratno in pa seveda zahtevno, hkrati pa so se višali stroški programske opreme. Zato so se kmalu pojavili visokonivojski programski jeziki ter prevajalniki. Pisanje prevajalnika je bilo zelo naporno predvsem zaradi pomanjkanja delovnega spomina tedanjih računalnikov. Vendar je misel, da bi se programi pisali samo v enem jeziku, nakar bi se jih prevedlo za več ciljnih arhitektur, odtehtala vse stroške, povezane z razvojem. Programer je tako napisal program samo enkrat, prevajalnik pa je poskrbel, da se je program pravilno prevedel za določen procesor. Ker je bilo pisanje programov v višjenivojskih jezikih bistveno lažje in je potekalo hitreje, se je cena programske opreme občutno znižala in je tako postala dostopna širši množici uporabnikov.

V začetku petdesetih let prejšnjega stoletja je bilo razvitih nekaj poskusnih prevajalnikov. Leta 1952 je prvi pravi prevajalnik napisal Grace Hopper za programski jezik A-0. Glavni Fortranov inženir, zaposlen pri podjetju IBM, John Backus, je kot prvi predstavil celoten prevajalnik. V letu 1960 pa je bil programski jezik Cobol prvi preveden za izvajanje na različnih računalniških arhitekturah.

Sprva so bili prevajalniki napisani v zbirnem jeziku. Prvi prevajalnik, ki je prevajal programe napisane v istem programskem jeziku kot prevajalnik sam, je bil razvit leta 1962, in sicer za jezik Lisp. Avtorja tega prevajalnika sta bila Tim Hart in Mike Levin z univerze MIT. Od leta 1970, je postala uveljavljena praksa, da se je prevajalnik za neki jezik napisalo v istem jeziku, ki ga je potem tudi sam prevajal. To je bilo izredno dobro merilo o tem, ali je bil programski jezik pravilno zasnovan ali ne. Vseeno so bili prevajalniki v veliki meri napisani predvsem v programskem jeziku C ali Pascal.

2.2 Groba zgradba

Zgradba prevajalnika je precej odvisna od jezika, ki ga prevaja. Če je jezik enostaven, je tudi prevajalnik lahko preprost program. A skoraj nikoli ni tako. Skoraj vsi današnji jeziki so precej kompleksni in skoraj nemogoče je napisati prevajalnik za tak jezik kot monoliten[1] program in pričakovati, da bo rezultat kvalitetno prevedena izvorna koda. Zato je bolje, da imamo prevajalnik sestavljen po fazah, kjer vsaka faza opravi svojo nalogo in rezultat preda naslednji fazi. Razdelitev prevajalnika na dele nam omogoča, da lahko pozneje enostavno predelamo posamezno fazo oziroma dodamo novo. Hkrati nam taka razdelitev da možnost, da posamezen programer dela na svojem delu in se na koncu vse združi v celoto.

Vsi razen majhnih prevajalnikov so razdeljeni v vsaj dva dela: prednji del (*front end*) in zadnji del (*back end*) prevajalnika. Prednji del je ponavadi odgovoren za sintaktično in semantično analizo, zadnji pa pretvori izvorni program v neko vmesno kodo, opravi optimizacijo ter generira prevod. Točka, ki ločuje ta dela, je ponavadi optimizacija vmesne kode, saj je ta faza neodvisna od izvorne ter prevedene kode. To nam omogoča poljubno menjavanje programskega jezika izvorne ali strojne kode in bo optimizacija še vedno enaka. V teoriji to pomeni, da lahko z m prednjimi deli in n zadnjimi deli prevajalnika dobimo $m * n$ prevajalnikov. Praktični primer tega je GCC (*GNU Compiler Collection*).

2.3 Izvorno-izvorni pretvorniki

Izvorno-izvorni pretvornik (*source-to-source compiler*) je tip prevajalnika, ki mu za vhod podamo izvorno kodo programa v nekem jeziku, na izhodu pa dobimo kodo programa v nekem drugem jeziku, ki delujeta na približno enakem nivoju abstrakcije, medtem ko tipični prevajalniki prevedejo izvorno kodo iz visokonivojskega jezika v neki nižjenivojski jezik. Na primer, lahko bi preva-

[1] Izdelan iz enega kosa.

jal iz Pascala v C.

Tak pretvornik pride prav tudi v primerih, ko imamo recimo neki program napisan v starejši različici nekega programskega jezika in ga želimo popraviti tako, da bi bil kompatibilen z novejšo različico jezika. Če je program kratek, se to da ročno, če pa je kode veliko in je potrebnih veliko sprememb pa je bolje, da to delo namesto nas opravi pretvornik.

Poglavje 3

Hipotetični računalnik SIC

Hipotetični računalnik SIC (Simplified Instruction Computer) je bil zasnovan z namenom, da ilustrira koncepte in značilnosti strojne opreme, s katerimi se največkrat srečujemo, hkrati pa se izogiba večini "navlake", najdene v pravih računalnikih, ki začetnika pogosto le zmedejo. Kot mnogi produkti tudi SIC obstaja v dveh različicah: standardna različica ter XE (*eXtra Equipment*, tudi *"eXtra Expensive"*) različica. Različici sta bili zasnovani tako, da sta navzgor kompatibilni – to pomeni, da je mogoče objektno kodo, zasnovano za računalnik SIC izvajati tudi na sistemu SIC/XE (takšna kompatibilnost je pogosto najdena na pravih sistemih, ki so povezani drug z drugim).

3.1 Arhitektura SIC

Pomnilnik

Pomnilnik je sestavljen iz zaporedno ležečih 8-bitnih bajtov (*byte*). Katerikoli trije zaporedno ležeči bajti sestavljajo eno besedo (*word*). Vsi naslovi na SIC so naslovi bajtov. Besede se naslavlja z lokacijo njihovega "najtežjega" bajta (*big-endian byte ordering*). Celoten pomnilnik obsega 32768 (2^{15}) bajtov.

Naziv	Namen
A	akumulator (*accumulator*); za aritmetične operacije
X	indeksni (*index*) register; za naslavljanje
L	povezavni (*linkage*) register; v ta register ukaz JSUB shrani povratni naslov
PC	programski števec (*program counter*); hrani naslov naslednjega ukaza
SW	statusni register (*status word*); hrani različne informacije, med drugim tudi "condition code" (CC)

Tabela 3.1: Nazivi in uporaba registrov SIC

Registri

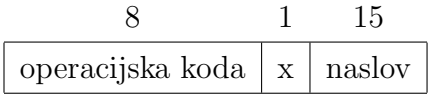
SIC ime pet registrov, vsak od njih služi svojemu namenu. Velikost vsakega registra je 24 bitov. V Tabeli 3.1 so prikazani nazivi registrov ter njihov namen.

Formati podatkov

Cela števila so shranjena kot 24-bitna predznačena binarna števila, za predstavitev negativnih števil pa se uporablja dvojiški komplement. Znaki so shranjeni v 8-bitnem formatu ASCII. Na standardni verziji SIC ni podpore za računanje z realnimi števili.

Formati ukazov

Vsi ukazi uporabljajo isti 24-bitni format, ki je prikazan na sliki 3.1.



Slika 3.1: Format ukazov standardne različice SIC

Načini naslavljanja

V splošnem ločimo dva načina naslavljanja:

- glede na način izračuna uporabnega naslova (*target address*, v nadaljevanju UN),
- glede na način uporabe UN, na podlagi katerega se določi operand.

Glede na način izračuna naslova SIC pozna le neposredni način naslavljanja. Neposredni način naslavljanja lahko kombiramo z indeksnim naslavljanjem, kar določa vrednost bita x v ukazu. Tabela 3.2 prikazuje kako se z neposrednim načinom naslavljanja izračuna UN iz naslova, podanega v ukazu. Oklepaji nakazujejo uporabo **vsebine** nekega registra ali pomnilniške lokacije.

Način	Indikator	Računanje UN
Neposredni	$x = 0$	$UN = naslov$
Neposredni (z indeksiranjem)	$x = 1$	$UN = naslov + (X)$

Tabela 3.2: Načini naslavljanja SIC glede na izračun UN

Glede na način uporabe UN SIC loči en sam način naslavljanja, ki je prikazan v tabeli 3.3.

Način	Indikator	Računanje UN
Enostavno	brez	$operand = (UN)$

Tabela 3.3: Načini naslavljanja SIC glede na uporabo UN

3.2 Arhitektura SIX/XE

Pomnilnik

Pomnilnik je pri SIC/XE urejen enako kot pri prej opisanem SIC, le da je tu povečana velikost na 1 megabajt (2^{20} bajtov). To povečava je privedla do

sprememb v formatu ukazov ter načinov naslavljanja.

Registri

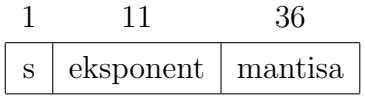
Razširjeni različici SIC so bili dodani štirje registri, ki so opisani v tabeli 3.4.

Naziv	Namen
B	bazni (*base*) register; za naslavljanje
S	splošnonamenski register
T	splošnonamenski register
F	48-bitni register za delo z realnimi števili

Tabela 3.4: Nazivi in uporaba dodatnih registrov SIC/XE

Formati podatkov

Razširjena različica SIC ohranja iste podatkovne formate kot standardni SIC. Dodan je le nov 48-bitni format za predstavitev realnih števil, ki je prikazan na sliki 3.2.



Slika 3.2: Format za predstavitev realnih števil

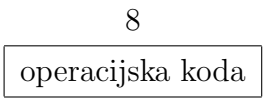
Mantisa je število med 0 in 1 (če predpostavljamo, da je plavajoča vejica takoj pred najvišjim bitom mantise). Za normalizirana števila velja, da je najvišji bit mantise enak 1. Eksponent je nepredznačeno binarno število med 0 in 2047. Če ima eksponent vrednost e in mantisa vrednost m, je vrednost števila predstavljenega s tem zapisom sledeča:

$$s * m * 2^{e-1024}$$

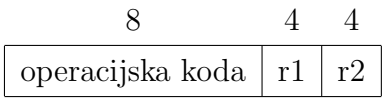
Predznak realnega števila nakazuje vrednost bita s (0 = pozitivno, 1 = negativno). Število 0 je predstavljeno tako, da se vse bite postavi na 0 (s, e in m).

Formati ukazov

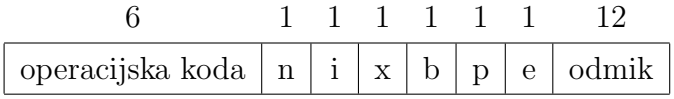
Večji pomnilnik pomeni, da se s 15 biti ne da več naslavljati celotnega pomnilniškega prostora, zato format ukazov standardnega SIC ni več primeren. Sta dve možnosti: ali uporabiti neko obliko relativnega naslavljanja ali pa razširiti polje za naslov na 20 bitov. Obe možnosti sta vključeni v SIC/XE (format 3 in format 4). Če je bit e v ukazu (sliki 3.5 in 3.6) enak 0, gre za format 3, drugače gre za format 4. Poleg tega SIC/XE ponuja tudi možnost uporabe ukazov, ki ne naslavljajo pomnilnika (format 1 in format 2).



Slika 3.3: Format 1



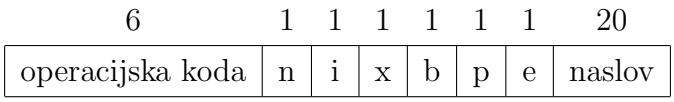
Slika 3.4: Format 2



Slika 3.5: Format 3

Načini naslavljanja

Glede na način izračuna UN sta dodana dva načina relativnega naslavljanja, za uporabo z ukazi formata 3. Oba načina lahko tako kot neposredni



Slika 3.6: Format 4

način kombiniramo z indeksiranjem. Način računanja UN si lahko ogledate v tabeli 3.5.

Način	Indikator	Računanje UN
Bazno-relativni	$b = 1, p = 0$	$UN = (B) + odmik$
PC-relativni	$b = 0, p = 1$	$UN = (PC) + odmik$

Tabela 3.5: Dodatni načini naslavljanja SIC/XE glede na izračun UN

Pri bazno-relativnem naslavljanju, je vrednost polja *odmik* v ukazu formata 3 predstavljena kot 12-bitno nepredznačeno celo število. Pri PC-relativnem naslavljanju je vrednost tega polja predstavljena kot 12-bitno predznačeno celo število. Negativna vrednost je predstavljena z dvojiškim komplementom.

Če sta bita b in p oba enaka 0, se vrednost polja *odmik* pri ukazu formata 3 uporabi kot *uporaben naslov*. Pri ukazih formata 4 sta ta bita ponavadi enaka 0 in se UN vzame iz polja *naslov*. Temu pravimo *neposredno* naslavljanje, v nasprotju z *relativnimi* naslavljanji, opisanimi v prejšnjem odstavku. Vse te načine naslavljanj se lahko kombinira z *indeksnim* naslavljanjem – če je bit $x = 1$, se vrednost registra X prišteje UN.

Bita i in n v ukazih formata 3 in 4 določata, kako naj se UN uporabi. Če je $i = 1$ in $n = 0$, se UN uporabi kot *takojšnji* operand. Temu pravimo *takojšnje* naslavljanje. Če je $i = 0$ in $n = 1$, se prebere besedo z naslova UN; prebrana vrednost pomeni naslov, na katerem je operand. Temu pravimo *posredno* naslavljanje. Če sta bita i in n oba enaka 0 ali oba enaka 1, UN pomeni lokacijo operanda v pomnilniku. Temu pa pravimo *enostavno*

naslavljanje.

Ukazi SIC/XE, ki ne uporabljajo niti takojšnjega niti neposrednega naslavljanja, se prevedejo z bitoma n in i enakima 0. Zbirnik za standardno različico SIC bi take ukaze prevedel ravno nasprotno – z bitoma n in i enakima 1 (to je zato, ker se vse operacijske kode ukazov standardnega SIC končajo z 00). Stroji SIC/XE imajo zaradi kompatibilnosti s SIC vgrajeno posebno funkcionalnost. Če sta prej omenjena bita enaka 0, potem se bite b, p in e uporabi kot del polja *naslov* v ukazu in ne kot zastavice, ki določajo naslavljanje. To povzroči, da so ukazi formata 3 identični ukazom na standardnem SIC, kar zagotavlja željeno kompatibilnost.

Glede na način uporabe UN sta prav tako dodana dva načina naslavljanja, ki sta predstavljena v tabeli 3.6.

Način	Indikator	Računanje UN
Takojšnje	#	$operand = UN$
Posredno	@	$operand = ((UN))$

Tabela 3.6: Dodatni načini naslavljanja SIC/XE glede na uporabo UN

3.3 Ukazi zbirnega jezika

Zbirna jezika SIC in SIC/XE skupaj obsegata 59 zbirniških ukazov. Vsi ukazi ter njihovi učinki in uporaba so opisani v tabeli 3.7.

Sintaksa	Učinek	Primer uporabe
`ADD m`	Prišteje `m` registru A	`ADD #4`
`ADDF m`	Prišteje `m` registru F	`ADDF _X`
`ADDR r1, r2`	Sešteje vsebini reg. `r1` in reg. `r2` in shrani rezultat v reg. `r2`	`ADDR A, S`
`AND m`	Opravi logično "in" operacijo med `m` in reg. A ter shrani rezultat v reg. A	`AND #4`
`CLEAR r1`	V reg. `r1` naloži 0	`CLEAR A`
`COMP m`	Primerja `m` z reg. A in nastavi zastavice v reg. `SW`	`COMP #4`
`COMPF m`	Primerja `m` z reg. F in nastavi zastavice v reg. `SW`	`COMPF _X`
`COMPR r1, r2`	Primerja reg. `r1` z reg. `r2` in nastavi zastavice v reg. `SW`	`COMPR A, S`
`DIV m`	Deli vsebino reg. A z `m` in shrani rezultat v reg. A	`DIV #4`
`DIVF m`	Deli vsebino reg. F z `m` in shrani rezultat v reg. F	`DIVF _X`
`DIVR r1, r2`	Deli vsebino reg. `r2` z reg. `r1` in shrani rezultat v reg. `r2`	`DIVR A, S`
`FIX`	Pretvori realno število v reg. F v celo število in shrani rezultat v reg. A	`FIX`
`FLOAT`	Pretvori celo število v reg. A v realno število in shrani rezultat v reg. F	`FLOAT`
`HIO`	Ustavi V/I kanal, ki je določen z vrednostjo reg. A	`HIO`
`J m`	Naloži `m` v reg. PC .	`J _LOOP`
`JEQ m`	Če zastavica CC v reg. SW predstavlja vrednost "enako", naloži `m` v reg. PC	`JEQ _LOOP`
`JGT m`	Če zastavica CC v reg. SW predstavlja vrednost "večje", naloži `m` v reg. PC	`JGT _LOOP`

`JLT m`	Če zastavica CC v reg. SW predstavlja vrednost "manjše", naloži `m` v reg. PC	`JLT _LOOP`
`JSUB`	Shrani vrednost reg. PC v reg. L in naloži `m` v reg. PC	`JSUB _PROG`
`LDA m`	Naloži `m` v reg. A	`LDA #4`
`LDB m`	Naloži `m` v reg. B	`LDB #4`
`LDCH m`	Naloži `m` v spodnjih 16-bitov reg. A	`LDCH #4`
`LDF m`	Naloži `m` v reg. F	`LDF #4`
`LDL m`	Naloži `m` v reg. L	`LDL #4`
`LDS m`	Naloži `m` v reg. S	`LDS #4`
`LDT m`	Naloži `m` v reg. T	`LDT #4`
`LDX m`	Naloži `m` v reg. X	`LDX #4`
`LPS m`	Naloži CPU status informacijo `m`	`LPS #4`
`MUL m`	Zmnoži `m` z reg. A in shrani rezultat v reg. A	`MUL #4`
`MULF m`	Zmnoži `m` z reg. F in shrani rezultat v reg. F	`MULF _X`
`MULR r1 ,r2`	Zmnoži reg. `r1` z reg. `r2` in shrani rezultat v reg. `r2`	`MULR A, S`
`NORM`	Normalizira vrednost v reg. F	`NORM`
`OR m`	Izvrši logično "ali" operacijo med `m` in reg. A ter shrani rezultat v reg. A	`OR #10`
`RD m`	Zapiše spodnjih 16 bitov reg. A v napravo številka `m`	`RD _IN`
`RMO r1, r2`	Skopira vrednost reg. `r1` v reg `r2`	`RMO A, S`
`RSUB`	Prenese vrednost reg. L v reg. PC	`RSUB`
`SHIFTL r1, n`	Opravi aritmetični pomik registra `r1` za `n` bitov v levo	`SHIFTL A, 4`
`SHIFTR r1, n`	Opravi aritmetični pomik registra `r1` za `n` bitov v desno	`SHIFTR A, 4`

`SIO`	Vzpostavi V/I kanal, ki je določen z vrednstjo reg. A. Naslov kanalskega programa je podan v reg. S	`SIO`
`SSK m`	Nastavi zaščitni ključ za naslov `m`. Ključ je v reg. A	`SSK _X`
`STA m`	Shrani vrednost reg. A na lokacijo `m`	`STA _X`
`STB m`	Shrani vrednost reg. B na lokacijo `m`	`STB _X`
`STCH m`	Shrani spodnjih 16 bitov reg. A na lokacijo `m`	`STCH _X`
`STF m`	Shrani vrednost reg. F na lokacijo `m`	`STF _X`
`STI m`	Shrani vrednost reg. I na lokacijo `m`	`STI _X`
`STL m`	Shrani vrednost reg. L na lokacijo `m`	`STL _X`
`STS m`	Shrani vrednost reg. S na lokacijo `m`	`STS _X`
`STSW m`	Shrani vrednost reg. SW na lokacijo `m`	`STSW _X`
`STT m`	Shrani vrednost reg. T na lokacijo `m`	`STT _X`
`STX m`	Shrani vrednost reg. X na lokacijo `m`	`STX _X`
`SUB m`	Odšteje `m` od reg. A in rezultat shrani v reg. A	`SUB #12`
`SUBF m`	Odšteje `m` od reg. F in rezultat shrani v reg. A	`SUBF _F`
`SUBR m`	Odšteje vrednost reg. `r1` od vrednosti reg. `r2` in rezultat shrani v reg. `r2`	`SUBR A, S`
`SVC n`	Sproži SVC prekinitev	`SVC 4`
`TD m`	Testira, ali je naprava št. `m` pripravljena	`TD _OUT`
`TIO`	Testira, ali je V/I kanal, podan z reg. A, pripravljen	
`TIX m`	Reg. X prišteje 1 in primerja vrednost z `m`	`TIX #4`
`TIXR r1`	Reg. X prišteje 1 in primerja vrednost z vrednostjo reg. `r1`	`TIXR A`

`WD m`	Prebere informacijo z naprave št. `m` in jo zapiše v spodnjih 16 bitov reg A	`WD _OUT`

Tabela 3.7: Seznam ukazov zbirnega jezika SIC/XE

Poglavje 4

Računalnik Intel Pentium x86

4.1 Arhitektura

Pomnilnik

Pomnilnik se lahko na arhitekturi Pentium x86 opiše vsaj na dva načina. Na fizični ravni je sestavljen iz 8-bitnih bajtov. Vsak naslov naslavlja en bajt. Dva zaporedna bajta tvorita besedo (*word*), štirje zaporedni bajti pa tvorijo dvojno besedo (*double word*, tudi *dword*). Nekatere operacije se izvedejo hitreje, če so operandi *poravnani* (npr. da je operand na naslovu, ki je večkratnik števila 4).

Programerji ponavadi gledajo na pomnilnik kot zbirko *segmentov*. Gledano s tega vidika je naslov sestavljen iz dveh delov – številke segmenta in odmika, ki kaže na bajt znotraj segmenta. Segmenti so lahko različnih velikosti ter služijo različnim namenom. Nekateri lahko na primer vsebujejo ukaze, drugi lahko hranijo podatke. Nekateri se obnašajo kot sklad, ki ga lahko uporabimo za shranjevanje vsebine registrov, podajanje parametrov podprogramom in še za druge namene.

Registri

Intel x86 ima 8 splošnonamenskih registrov: *EAX*, *EBX*, *ECX*, *EDX*, *EBP*, *ESI*, *EDI* ter *ESP*. Vsak od teh je dolg 32 bitov (ena dvojna beseda). Registri *EAX*, *EBX*, *ECX* in *EDX* so predvsem namenjeni manipulaciji s podatki; možen je tudi dostop do besede oziroma bajta v teh registrih (za take dostope dobijo registri drugačna imena, kar je vidno na sliki 4.1). Tudi ostale registre se lahko uporablja za manipulacijo s podatki, ampak se ponavadi uporabljajo za hranjenje naslovov.

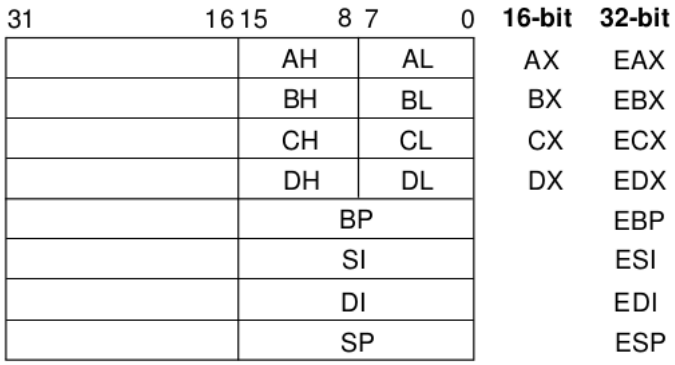
Poleg splošnonamenskih registrov obstaja še nekaj namenskih. Register *EIP* je 32-bitni register, ki vsebuje kazalec (naslov) na naslednji ukaz. Tudi register *EFLAGS* je velik 32 bitov in hrani veliko število zastavic. Nekatere prikazujejo stanje procesorja, nekatere pa se uporabljajo za shranjevanje rezultatov primerjav in aritmetičnih operacij.

Poleg teh je na voljo še 6 16-bitnih *segmentnih* registrov, katerih namen je hranjenje lokacije segmentov v pomnilniku. Register *CS* (Code Segment) hrani naslov segmenta tiste kode, ki se trenutno izvaja, register *SS* (Stack Segment) pa hrani naslov trenutnega skladovnega segmenta. Ostali registri *DS*, *ES*, *FS* ter *GS* so namenjeni hranjenju lokacije podatkovnih segmentov. Operacije s plavajočo vejico opravlja posebna enota *FPU* (Floating Point Unit). Enota vsebuje osem 80-bitnih podatkovnih registrov, ki delujejo na principu sklada ter nekaj kontrolnih in statusnih registrov. Več podrobnosti o FPU si lahko preberete v [4].

Poleg vseh teh registrov, do katerih lahko dostopajo uporabniški programi, obstajajo še registri, ki so namenjeni samo sistemskim programom (npr. OS) ter še drugi, ki kontrolirajo delovanje procesorja.

Formati podatkov

Arhitektura Pentium x86 zagotavlja hrambo celih števil, števil v plavajoči vejici, znakov ter nizov znakov. Cela števila se hranijo kot 8-, 16- ali 32-bitna števila. Podprta so tako predznačena kot nepredznačena cela števila; za zapis negativnih števil je tako kot pri SIC uporabljena predstavitev z dvojiškim



Slika 4.1: Splošnonamenski registri računalnika Intel Pentium x86

komplementom.

Obstajajo trije formati za shranjevanje realnih števil. Format enojne natančnosti (*single-precision*) je dolg 32 bitov. 24 bitov je rezervirano za mantiso ter 7 bitov za eksponent (1 bit predstavlja predznak). Za zagotavljanje dvojne natančnosti (*double-precision*) je na voljo 64-bitni format. 53 bitov predstavlja mantiso, 10 bitov pa ostane za eksponent. Zadnji, razširjeni format, je dolg 80 bitov, kjer je 64 bitov rezerviranih za mantiso in 15 bitov za eksponent.

Znake se shranjuje kot 8-bitne kode ASCII.

Formati ukazov

Vsi strojni ukazi na arhitekturi Pentium x86 uporabljajo variacije enega osnovnega formata. Najvišji biti so neobvezne zastavice, ki spreminjajo izvajanje ukaza. Na primer, nekatere zastavice določajo, kolikokrat naj se izvede ukaz, ali pa določajo segmentni register, ki naj se uporabi za naslavljanje. Sledi operacijska koda ukaza, dolga 1 ali 2 bajta. Za operacijsko kodo so biti, ki določajo operande in načine naslavljanja.

Operacijska koda je edini element ukaza, ki je vedno prisoten v ukazu. Ostali elementi so opcijski. Tako lahko dobimo vrsto različnih formatov dolžine od enega do deset bajtov.

Načini naslavljanja

Arhitektura Pentium x86 nudi veliko načinov naslavljanja. Vrednost operanda se lahko poda kot del ukaza (*takojšnje* naslavljanje) ali pa se ga poda v registru. Operande, ki so shranjeni v pomnilniku, se pogosto naslavlja z variacijami splošnega računanja UN:

$$UN = (bazni\ register) + (indeksni\ register) * skalar + odmik$$

Katerikoli splošnonamenski register se lahko uporabi kot bazni register. Prav tako se lahko katerikoli splošnonamenski register razen *ESP* uporabi kot indeksni register. Kot *skalar* se lahko uporabi števila 1, 2, 4 ali 8, *odmik* pa je 8-, 16- ali 32-bitna vrednost. Številki baznega ter indeksnega registra sta poleg skalarja in odmika zakodirani kot del operanda v ukazu. Ker vsi ti elementi niso obvezni in se lahko izpustijo, tako dobimo osem različnih načinov naslavljanja. Naslov operanda v pomnilniku se lahko poda kot absolutni naslov (*neposredno* naslavljanje) ali kot lokacija, ki je relativna na register *EIP* (*relativno* naslavljanje).

4.2 Ukazi zbirnega jezika

Ker je ukazov zbirnega jezika Intel Pentium preveč, da bi vse opisali, bomo predstavili le tisto podmnožico, ki je zadostovala za pretvorbo ukazov zbirnega jezika računalnika SIC. Ukazi ter njihovi učinki in uporaba so opisani v tabeli 4.1. Preostale ukaze ter dodatne informacije si lahko ogledate v [5].

Sintaksa	Učinek	Primer uporabe
`ADD dst, src`	Sešteje `dst` in `src` ter shrani rezultat v `dst`	`ADD %eax, 4`
`AND dst, src`	Opravi logično "in" operacijo med `dst` in `src` ter shrani rezultat v `dst`	`AND %eax, 4`

`CALL dst`	Shrani reg. EIP na sklad in naloži `dst` v reg. EIP	`CALL _prog`
`CMP op1, op2`	Primerja `op1` in `op2` med sabo in posodobi zastavice v reg. EFLAGS	`CMP _st, 10`
`IDIV dst, src`	Deli vrednost reg. EAX z `op` in shrani rezultat v reg. EAX `dst`	`DIV 2`
`FADD op`	Sešteje `op` in vrh sklada FPU ter shrani rezultat na vrh sklada FPU	`FADD _real`
`FCOM op`	Primerja `op` in vrh sklada FPU ter posodobi zastavice enote FPU	`FCOM _real`
`FDIV op`	Deli vrh sklada FPUz `op` in shrani rezultat na vrh sklada FPU	`FDIV _real`
`FILD op`	Pretvori `op` celo število v realno število in shrani rezultat na vrh sklada FPU	`FILD _int`
`FIST op`	Pretvori realno število na vrhu sklada FPU v celo število in shrani rezultat v `op`	`FIST _int`
`FLD op`	Naloži realno število na vrh sklada FPU	`FLD _real`
`FMUL op`	Zmnoži `op` z vrhom sklada FPU ter shrani rezultat na vrh sklada FPU	`FMUL _real`
`FST op`	Shrani vrh sklada FPU v `op`	`FST _real`
`FSUB dst, src`	Odšteje `op` od vrednosti na vrhu sklada FPU ter shrani rezultat na vrh sklada FPU	`FSUB _real`
`INT num`	Sproži prekinitev določeno z `num`	`INT 0x80`
`Jcc dst`	Naloži `dst` v reg. EIP, če je izpolnjen pogoj.	`JE %eax, 4`
`MOV dst, src`	Kopira `src` in shrani v `dst`	`MOV [_X], %eax`

`IMUL dst, src`	Zmnoži `dst` in `src` ter shrani rezultat v `dst`	`MUL %eax, 4`
`OR dst, src`	Opravi logično "ali" operacijo med `dst` in `src` ter shrani rezultat v `dst`	`OR %eax, 4`
`POP dst`	Odstrani element z vrha sklada in ga shrani v `dst`	`POP %eax`
`PUSH src`	Porine `src` na vrh sklada	`PUSH %eax`
`RET dst, src`	Odstrani naslov s sklada in ga naloži v reg. EIP	`RET`
`SHL src, n`	Opravi aritmetični pomik registra `src` za `n` bitov v levo	`SHL %eax, 2`
`SHR src, src`	Opravi aritmetični pomik registra `src` za `n` bitov v desno	`SHR %eax, 2`
`SUB dst, src`	Odšteje `src` od `src` ter shrani rezultat v `dst`	`SUB %eax, 12`

Tabela 4.1: Seznam izbranih ukazov zbirnega jezika Intel Pentium x86

Poglavje 5

Pretvornik Sic2Intel

Kot praktični del te diplomske naloge je bil izdelan paket, ki pretvori kodo zbirnega jezika SIC v zbirni jezik arhitekture Intel Pentium x86. V nadaljevanju tega poglavja si bomo ogledali samo zgradbo pretvornika, orodja s katerimi smo si pomagali, ter težave, ki smo jih srečevali pri izdelavi.

5.1 Razvojna orodja

Pri delu smo si pomagali z več programskimi orodji, da bi si čimbolj olajšali delo pri tistih stvareh, ki so bile v preteklosti že večkrat temeljito obravnavane in rešene. To se tiče predvsem leksikalne in sintaksne analize izvornega programa. Ta orodja so bila izbrana predvsem zato, ker smo se z njimi spoznali že med študijem pri enem od predmetov. Nekatera uporabljena orodja bomo v nadaljevanju tudi na kratko opisali.

JFlex

JFlex je računalniški program, ki generira leksikalne analizatorje (*lexer, scanner*). Leksikalni analizatorji opravljajo leksikalno analizo, kar je proces pretvarjanja zaporedja vhodnih znakov v izhodno zaporedje osnovnih simbolov (*tokens*). S posebnimi pravili se definira vzorce (*patterns*). V večini primerov so ta pravila kar *regularni izrazi*. S pomočjo regularnih izrazov analizator

Leksem	Žeton
abc	IDENTIFIER
,	COMMA
4	NUMBER

Tabela 5.1: Primer žetonov in pripadajočih leksemov

primerja prebrano zaporedje znakov in tvori osnovne simbole. Če analizator ne najde ustreznega vzorca, s katerim bi lahko povezal prebrano zaporedje znakov, javi napako. Tako se zelo hitro ugotovi, ali so v izvorni kodi nepravilne ključne besede ali simboli. Zelo prav pride tudi odstranjevanje odvečnega belega besedila ter komentarjev. Ne zna pa analizator ugotoviti, ali si simboli sledijo v pravilnem zaporedju (ali je program pravilno strukturiran). Zato tu pride na vrsto drugo orodje. Dodatne podrobnosti o leksikalnih analizatorjih boste našli v [1], več o samem orodju pa v [8].

Java CUP

Java Cup je računalniški program, ki pretvarja tok osnovnih simbolov, ki jih generira leksikalni analizator (v našem primeru JFlex) v neko vmesno predstavitev – ponavadi abstraktno sintaksno drevo. Cup na podlagi gramatičnih pravil, ki jih definira uporabnik, zgradi LALR (Look-Ahead Left to Right) sintaksni analizator. Gramatična pravila oziroma gramatika opisujejo neki jezik. Gramatika sestoji iz množice *produkcij* oblike

$$simbol ::= simbol\ simbol\ \ldots\ simbol\ \{:\ akcija\ :\}$$

kjer je lahko eden ali več simbolov na desni strani produkcije. Neki simbol je lahko ali *terminal*, kar pomeni, da je osnovni simbol, ali *neterminal*, kar pomeni, da se ta simbol pojavlja tudi na levi strani neke produkcije. Osnovni simbol se nikoli ne more pojaviti na levi strani produkcije. Izraz v zavitih oklepajih predstavlja akcijo, ki se zgodi ob vsaki prevedbi – ko se zamenja terminale in neterminale na desni strani z neterminalom na levi

strani produkcije. Če analiza uspe na koncu dobimo abstraktno sintaksno drevo, s katerim lahko v kasnejših fazah prevajanja manipuliramo. Ostale podrobnosti o sintaksni analizi ter orodju najdete v [2] ter [7].

5.2 Struktura projekta

Kot skoraj vsak prevajalnik je tudi ta sestavljen iz več delov. Osnovna struktura projekta je sledeča:

- **lexer** – del, odgovoren za leksikalno analizo, glej razdelek 5.3
- **parser** – del, odgovoren za sintaksno analizo, glej razdelek 5.4
- **structure** – del, odgovoren za generiranje kode, glej razdelek 5.5
- **gui** – del, odgovoren za prikaz grafičnega uporabniškega vmesnika, glej razdelek 5.6

Marsikdo bi opazil, da manjka še precej delov (faz) prevajalnika in prav bi imel. A naj vas opomnimo, da ta prevajalnik ni čisti prevajalnik, ampak le pretvornik, ki pretvarja kodo iz enega zbirnega jezika v drugega in ne direktno v strojno kodo. Zato smo lahko nekatere faze izpustili. Poleg tega je bilo mišljeno, da se pretvarja le pravilne programe, zato tudi ni potrebno toliko preverjanja pravilnosti.

V nadaljevanju bomo opisali glavne razrede ter značilnosti posameznih delov pretvornika.

5.3 Leksikalna analiza

Prvi del je odgovoren za leksikalno analizo. Tu smo si pomagali z orodjem JFlex, ki je opisano v poglavju 5.1 na strani 25. Glavni razred je razred `SicLexer.java`, ki ga avtomatsko zgenerira orodje JFlex iz datoteke

`sic.jflex`. Leksikalni analizator iz podane vhodne datoteke izlušči zaporedje osnovnih simbolov. Vsak osnovni simbol je predstavljen z razredom `Symbol.java`, ki se nahaja v knjižnici `java_cup.runtime`. Da je orodje lahko zgeneriralo analizator (`SicLexer.java`), je bilo treba najprej vzpostaviti združljivost z orodjem Java CUP (izsek kode 5.1). To povzroči, da CUP zapiše terminale, definirane v `sic.cup`, kot konstante v generiran razred `SicTokens.java`, ki jih potem JFlex uporabi za tvorjenje osnovnih simbolov s pomočjo razreda `Symbol.java`. Da bi lahko leksikalni analizator iz vhodne datoteke razpoznal te simbole, jih je treba opisati s pomočjo regularnih izrazov (izsek kode 5.2).

```
1 %cupsym     sic2intel.parser.SicTokens
2 %implements java_cup.runtime.Scanner
3 %function   next_token
4 %type       Symbol
```

Izsek kode 5.1: Vzpostavitev združljivosti z Java CUP

```
1  // whitespace
2  [ \t]+ { }
3  // new lines
4  (\n)+ | (\r)+ | (\r\n)+ { return sym(SicTokens.NL); }
5
6  // ukazi
7  "ADDR" { return sym(SicTokens.ADDR); }
8  "ADDF" { return sym(SicTokens.ADDF); }
9  "ADDR" { return sym(SicTokens.ADDR); }
10 // ...
11
12 // posebni simboli
13 "+" { return sym(SicTokens.PLUS); }
14 "-" { return sym(SicTokens.MINUS); }
15 "/" { return sym(SicTokens.DIVIDE); }
16 // ...
17
18 // labele in stevila
19 [_a-zA-Z][a-zA-Z0-9_]* { return sym(SicTokens.IDENTIFIER); }
20 [0-9]+ | [1-9][0-9]+ { return sym(SicTokens.NUMBER); }
21 "0x"[0-9A-F]+ { return sym(SicTokens.HEXNUMBER); }
22 "X\'"[0-9A-F]+"\'" { return sym(SicTokens.HEXCONST); }
23 // ...
24
25 // komentarji
26 "." {
27   yybegin(COMMENT);
28 }
29
30 <COMMENT> {
31   "<" { yybegin(COMMANDS); }
32   (\n)+ | (\r)+ | (\r\n)+ {
33     yybegin(NORMAL);
34     return sym(SicTokens.NL);
35   }
36   . { } // komentar
37 }
```

Izsek kode 5.2: Primeri regularnih izrazov za JFlex

5.4 Sintaksna analiza

Tudi tu smo si pomagali z orodjem, in sicer z Java CUP, ki je že opisano v poglavju 5.1 na strani 26. Orodje na podlagi gramatike zgenerira razreda `SicSyntax.java` ter `SicTokens.java`. V slednjem so terminali, ki smo jih definirali v datoteki `sic.cup`. Poleg terminalov so v datoteki `sic.cup` tudi definicije raznih neterminalnih simbolov ter celotna gramatika za opis zbirnega jezika SIC (izsek kode 5.3).

```
1  terminal ADD, COMMA, PLUS, HASH, AT, REG_X;
2
3  non terminal SicInstr  add_instr;
4  non terminal epsilon, extended, addressing;
5
6  add_instr ::= extended:e ADD addressing:a identifierOrNumber:id indexed:i
       new_line_non_opt
7                {:
8                  RESULT = new SicInstrADD(e, a, id, i);
9                :};
10
11 extended ::= epsilon {: RESULT = false; :}
12             | PLUS {: RESULT = true; :};
13
14 indexed ::= epsilon {: RESULT = false; :}
15            | COMMA REG_X {: RESULT = true; :};
16
17 addressing ::= epsilon {: RESULT = SicAddressing.SIMPLE; :} // enostavno
18               | HASH {: RESULT = SicAddressing.IMMEDIATE; :} // takojsnje
19               | AT {: RESULT = SicAddressing.INDIRECT; :}; // posredno
20
21 identifierOrNumber ::= identifier:id {: RESULT = id.toString(); :}
22                        | numbers:num {: RESULT = num.toString(); :};
23
24 epsilon ::= ;
```

Izsek kode 5.3: Primer produkcij ter definicij terminalov in neterminalov za

Java CUP.

5.5 Generiranje in pretvarjanje kode

Če je program sintaktično pravilen, sledi faza generiranja kode. Ker je ta del najobsežnejši in je odgovoren za končni rezultat, si ga bomo podrobneje ogledali.

Paket `structure` vsebuje še dve podmapi – `sic` ter `intel`. V obeh najdemo precej razredov, ki so namenjeni opisu posameznih ukazov obeh zbirnih jezikov. Ko pridemo do te faze, je abstraktno sintaksno drevo generirano in lahko se lotimo pretvarjanja ukazov. Tu prideta na vrsto dva obiskovalca (*visitors*), `DataInspector.java` in `CodeGenerator.java`.

Razložimo kaj je obiskovalec. Naše abstraktno sintaksno drevo predstavlja razred `SicProgram.java`, ki vsebuje urejen seznam vseh ukazov izvornega programa. Če bi hoteli pretvoriti te ukaze v ciljni jezik, bi morali napisati metodo, ki bi se sprehodila skozi celoten seznam. Ker je vsak ukaz predstavljen s svojim razredom, bi morali za vsakega napisati svoj `if`-stavek. To ni niti lepo niti praktično. Zakaj ne bi raje izkoristili lastnosti, ki nam jih ponuja Java, kot je recimo redefiniranje metod (*overloading*). Vsakemu od razredov, ki predstavlja ukaz v SIC, dodamo metodo `accept`, ki kot argument sprejme primerek (*instance*) nekega obiskovalca. Tu se kliče metodo `visit` v obiskovalcu s tipom kličočega objekta kot argument. Ker obiskovalec vsebuje toliko metod `visit`, kolikor je različnih razredov, se na podlagi tipa objekta v argumentu odloči za pravo redefinirano metodo `visit`. Tako se elegantno izognemo ponavljajočemu se pisanju `if`-stavkov ter preverjanja tipov z operatorjem tt instanceof. Zdaj ko vemo, kaj pomeni obiskovalec, se lahko vrnemo na prej omenjena razreda `DataInspector.java` ter `CodeGenerator.java`.

Kot že ime pove, je prvi obiskovalec namenjen obdelavi podatkovnih tipov SIC (BYTE, WORD, RESW, RESB in EQU). Tu se podatkovne tipe ter njihove vrednosti shrani v razpršeno tabelo (*hashmap*) za lažjo analizo v drugem obhodu z obiskovalcem `CodeGenerator.java`. Naloga tega obiskovalca je

SIC	Intel
`BYTE val`	`.byte val`
`WORD val`	`.long val`
`RESB n`	`.space n`
`RESW n`	`.space 4*n`
`l EQU e`	`.equ l, e`

Tabela 5.2: Preslikovalna tabela podatkovnih tipov SIC – Intel

preslikava ukazov zbirnega jezika SIC v ukaze zbirnega jezika Intel x86. Ker je preslikavanja kar nekaj, bomo ta proces opisali po delih.

5.5.1 Podatkovni tipi

Kot je že bilo omenjeno, na SIC en bajt predstavlja zaporedje 8 bitov, eno besedo pa predstavlja zaporedje treh bajtov. Preslikavanje bajtov (ukaz `BYTE`) torej ni povzročilo nobene težave, saj je tudi na arhitekturi Pentium x86 velikost bajta 8 bitov. Drugače je pri besedah (ukaz `WORD`). Na arhitekturi Pentium x86 je ena beseda predstavljena s 16 biti, kar je premalo za preslikavo. Zato je bilo treba seči po dvojni besedi (*dword*, ukaz `.long`), ki je dolga 32 bitov.

Drugi problem pri preslikavi zgornjih podatkovnih tipov je način, kako obe arhitekturi shranjujeta ter naslavljata besede. SIC naslavlja besede z lokacijo njihovega "najtežjega" bajta, medtem ko je na arhitekturi Intel x86 to ravno obratno. To najbolj vpliva na shranjevanje ter dostopanje do elementov seznama. Ker nam ni uspelo ugotoviti, kako bi realizirali to preslikavo, je odgovornost na programerju izvornega programa, da prilagodi kodo za pretvorbo.

Ostale preslikave niso povzročale pretiranih preglavic in si jih lahko ogledate v tabeli 5.2.

SIC	Intel
A	EAX
X	EDI
L	EBP
PC	EIP
SW	EFLAGS
B	EBX
S	ECX
T	EDX
F	vrh FPU sklada

Tabela 5.3: Preslikovalna tabela registrov SIC – Intel

5.5.2 Registri

Preslikovanje registrov ni povzročalo večjih težav, saj je registrov na računalniku Pentium več kot na SIC. Treba se je bilo le odločiti za prave preslikave, ki jih lahko vidite v tabeli 5.3.

Problem se je pojavil le pri registru *EFLAGS*, saj ukaz, ki naj bi shranil vsebino na sklad, shrani le polovico registra in tako ne omogoča manipuliranja s celotnim registrom.

5.5.3 Naslavljanje

Kot je bilo povedano v razdelku *Načini naslavljanja* na strani 9, SIC/XE pozna tri načine naslavljanja glede na način uporabe UN – enostavno, takojšnje ter posredno naslavljanje. Ker tudi arhitektura Intel Pentium pozna enostavno in takojšnje naslavljanje, razen pri nekaterih ukazih, tu ni bilo težav. Manjše težave povzročajo ukazi, ki ne podpirajo operandov s takojšnjim naslavljanjem. Pri teh je treba ustvariti dodatne spremenljivke ter shraniti operand v te spremenljivke. Nato se lahko uporabi enostavno naslavljanje za dostop do operanda. Podobno je tudi pri posrednem naslavl-

	SIC	Intel
enostavno	`LDA _X`	`mov %eax, [_X]`
takojšnje	`LDA #5`	`mov %eax, 5`
posredno	`LDA @_X`	`mov %esi, [_X]` `mov %eax, [%esi]`

Tabela 5.4: Preslikovalna tabela naslavljanj SIC – Intel

janju. Posredno naslavljanje na SIC pomeni, da je operand na naslovu, ki ga določa vsebina pomnilniške lokacije na naslovu UN ($operand = ((UN))$). Na računalniku Pentium se lahko operand posredno naslavlja samo v primeru, da je naslov operanda vnaprej zapisan v registru. Temu pravimo registrsko posredno naslavljanje. Na srečo ima Pentium več registrov kot SIC/XE, zato smo lahko register *ESI* rezervirali posebej za pretvarjanje ukazov s posrednim naslavljanjem. Tako se ukaz s posrednim naslavljanjem pretvori v zaporedje dveh ukazov: nalaganje naslova operanda v register *ESI* ter branje pravega operanda z naslova, ki se nahaja v registru *ESI*. Pri enostavnem in posrednem naslavljanju lahko tako kot pri SIC uporabimo tudi indeksiranje. To storimo tako, da naslovu prištejemo indeksni register. V našem primeru je to register *EDI*. Primere pretvorb si lahko ogledate v tabeli 5.4.

5.5.4 Ukazi

Ukazi LDx

`LDA m`, `LDB m`, `LDCH m`, `LDL m`, `LDS m`, `LDT m`, `LDX m`

Vsi ti ukazi se enostavno pretvorijo v ukaze `mov`:

`mov %`X`, [m]`

kjer X predstavlja ustrezen register. Manjša razlika je le pri ukazu `LDCH`, ki se zaradi nalaganja bajta v register A malenkostno razlikuje od ostalih.

Namesto registra `%eax` se uporabi register `%al`, ker se nalaga na spodnjih 8 bitov registra.

Ukazi STx

`STA m`, `STB m`, `STCH m`, `STL m`, `STS m`, `STT m`, `STX m`

Podobno kot ukazi LDx, se tudi vsi tile ukazi enostavno prevedejo v ukaze `mov`:

$$\texttt{mov [m], \%}X$$

kjer X predstavlja ustrezen register. Za ukaz `STCH` tu velja podobno kot za ukaz `LDCH`.

Skočni ukazi

`J m`, `JEQ m`, `JGT m`, `JLT m`

Prvi ukaz se razlikuje od drugih, saj označuje brezpogojni skok in se tako pretvori v ukaz `jmp m`. Ostali ukazi so pogojni skoki in se pretvorijo v `je m`, `jg m` in `jl m`. Pri vseh skočnih ukazih je bil dodan mehanizem za preverjanje neskončnih zank. V primeru najdene neskončne zanke, se skočni ukaz zamenja z zaporedjem ukazov za končanje programa (`sys_exit`), ki prepreči neskončno izvajanje programa.

Ukazi za delo z napravami

`TD m`, `RD m`, `WD m`

Naš pretvornik obravnava naprave kot datoteke. Na primer, če se na SIC bere z naprave `0xAA`, se bo v pretvorjenem programu dejansko bralo iz datoteke z imenom `AA.txt`.

Vsi trije ukazi se pretvorijo v zaporedje ukazov, ki prožijo prekinitev (`int 0x80`) za opravilo določenega sistemskega klica (tabela 5.5). Tako se ukaz `TD`

```
int sys_open(char* filename, int flags, int mode)
ssize_t sys_read(int fd, char* buf, size_t count)
ssize_t sys_write(int fd, char* buf, size_t count)
```

Tabela 5.5: Sistemski klici za delo z datotekami

pretvori v zaporedje ukazov, ki prožijo prekinitev za opravilo sistemskega klica `sys_open`. To povzroči, da se ustvari datoteka s podanim imenom in se vrne njen deskriptor ali, če datoteka že obstaja, se vrne samo deskriptor. Ta informacija se shrani v posebno vnaprej generirano spremenljivko, do katere potem dostopata preostala ukaza. Ko enkrat imamo deskriptor datoteke, je branje ali pisanje v datoteko precej enostavno. Za branje se opravi sistemski klic `sys_read`, za pisanje pa `sys_write`. Za vsak klic je treba v predpisane (ponavadi *EAX, EBX, ECX* ter `EDX`) registre naložiti vse potrebne informacije, kot so na primer identifikator sistemskega klica, deskriptor, naslov polja znakov ... Izjema so naprave 0 (stdin), 1 (stdout) ter 2 (stderr). Ker so te naprave stalno "odprte", ni treba ustvarjati nove datoteke, ampak se lahko takoj uporabijo za pisanje ali branje.

Dodatne informacije o uporabi sistemskih klicev v okolju Linux, si lahko preberete v [3] ter [6].

Preostali ukazi

```
DIVR r1, r2
```

Ukaz `div` na arhitekturi Intel Pentium deli register *EAX* z nekim registrom ali operandom v pomnilniku. Ker ima SIC ukaz `DIVR` oba operanda registra, to povzroča manjšo težavo. Dokler je *r2* enak registru A, lahko preslikamo ukaz tako, kot če bi uporabljali navadno deljenje. Če ni tako, je treba dodati še par ukazov. Najprej je treba register *EAX* shraniti na sklad, nato moramo kopirati *r2* v register *EAX* in šele nato lahko delimo z *r1*. Ko opravimo deljenje, prenesemo rezultat nazaj v *r2* in povrnemo register *EAX* v prvotno stanje.

Do sedaj smo obdelali že skoraj polovico ukazov. Ker preostala polovica ne spada v večjo družino ukazov in ker se dokaj enostavno prevedejo v ciljni jezik, bomo njihove preslikave predstavili v tabeli 5.6.

Če ste si pozorno pregledali tabelo, ste morda opazili, da nekaj ukazov manjka. Vzrok tega je, da nekateri ukazi nimajo ustrezne preslikave za računalnik Intel Pentium nekaterih pa se na SIC-u sploh ne uporablja. Teh ukazov naš pretvornik ne podpira in bo javil napako pri procesu pretvarjanja.

Dodatni ukazi

Kot dodatek k diplomski nalogi je bilo implementiranih še nekaj ukazov, ki na SIC nimajo pomena, saj jih zbirnik za SIC/XE razpozna kot komentarje. Njihov namen je, da bi si z njihovo pomočjo olajšali delo pri delu z vhodom in izhodom. Sintaksa ter njihovi učinki so opisani v tabeli 5.7. Ti ukazi opravljajo isto nalogo kot SIC ukazi `TD`, `RD` ter `WD`. Razlika je le v tem, da so dodatni ukazi lažji za uporabo, saj se med procesom pretvarjanja kode avtomatsko generirajo vse spremenljivke ter procedure, ki so potrebne za izpis oziroma branje.

SIC	Intel
ADD m	add %eax, m
ADDF m	fadd m
ADDR m	add %r2, %r1
AND m	and %eax, m
CLEAR r1	mov %r1, 0
COMP m	cmp %eax, m
COMPF m	fcom m
COMPR r1, r2	cmp %r1, %r2
DIV m	div m
DIVF m	fdiv m
FIX	fist m
FLOAT	fild m
JSUB m	call m
LDF m	fld m

SIC	Intel
MULF m	fmul m
OR m	or %eax, m
RMO r1, r2	mov %r2, %r1
RSUB	ret
SHIFTL r1, n	shl %r1, n
SHIFTR r1, n	shr %r1, n
STF m	fst m
SUB m	sub %eax, m
SUBF m	fsub m
SUBR r1, r2	sub %r2, %r1
TIX m	add %edi, 1 cmp %edi, m
TIXR r1	add %edi, 1 cmp %edi, %r1

Tabela 5.6: Preslikovalna tabela ukazov SIC – Intel

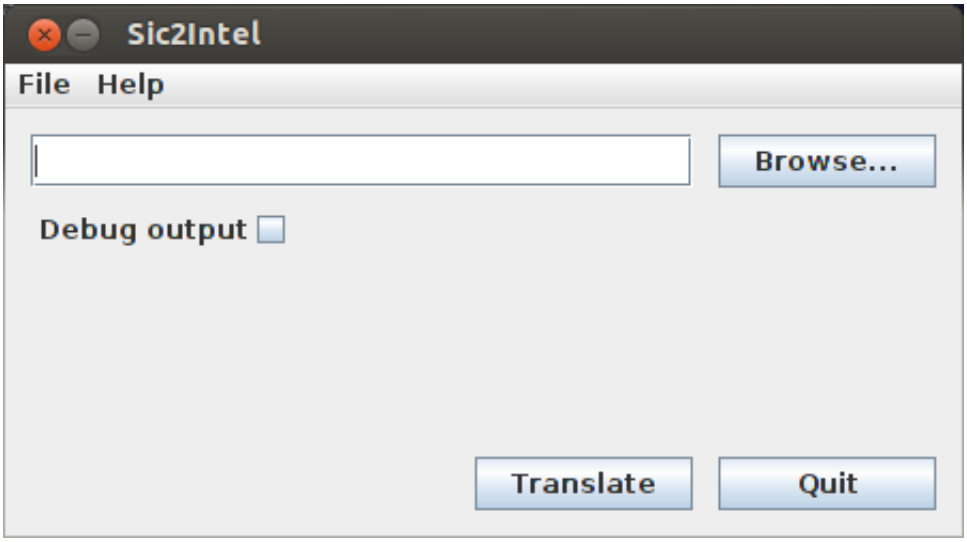
Sintaksa	Učinek
`.<WRITE('message')>`	izpiše *message* na standardni izhod
`.<WRITE(var, len)>`	izpiše *len* dolgo besedilo, ki je na naslovu *var*, na standardni izhod
`.<READ(var, len)>`	prebere *len* znakov s standardnega vhoda in jih shrani na naslov *var*

Tabela 5.7: Razpredelnica dodatnih ukazov

5.6 Grafični vmesnik

Čeprav se da celotni prevajalnik upravljati prek terminala, je bil vseeno izdelan tudi preprost uporabniški vmesnik, saj nekateri uporabniki preferirajo pogled na gumbe kot pa pozivno okno. Razred, ki je odgovoren za prikaz vmesnika je v datoteki `CompilerFrame.java`. Na sliki 5.1 je prikazano glavno okno pretvornika.

Ob kliku na gumb **Browse** se nam odpre okno za iskanje datotek. Poiščemo ustrezno datoteko in kliknemo gumb **Translate**. Pojavi se sporočilo, ki nam pove, ali je bilo pretvarjanje uspešno ali ne. Če proces ni bil uspešen, se nam v direktorju, kjer je izvorna datoteka, namesto pretvorje kode pojavi datoteka z opisom napak. Če obkljukamo **Debug output**, bodo v izhodni datoteki pred pretvorjenimi ukazi dodani komentarji z izvornimi ukazi SIC.



Slika 5.1: Pogled na uporabniški vmesnik

Poglavje 6

Sklepne ugotovitve

V okviru te diplomske naloge smo se spoznali s prevajalniki ter obdelali podrobnosti arhitektur računalnikov SIC ter Intel Pentium. Kot praktični prikaz pridobljenega znanja smo izdelali pretvornik, ki pretvori programe, napisane za hipotetični računalnik SIC, v zbirni jezik računalnika Intel. Pri razvoju smo naleteli na nemalo težav. Postopoma smo nekatere od teh problemov rešili, nekateri trši orehi pa še vedno obstajajo. Pri izdelavi pretvornika smo se seznanili tudi z orodji, ki so nam olajšali delo. Namen pretvornika je, da omogoča prevedbo ter posledično tudi izvajanje programov, napisanih za izmišljen računalnik, na pravem stroju. Tako bi dobili zadovoljstvo opazovanja izvajajanja programa na pravem računalniku, hkrati pa bi se s proučevanjem seznanili z novim zbirnim jezikom.
Kot pri mnogih stvareh je tudi tu precej možnosti za izboljšave. Najpomembnejša izboljšava bi bila, da bi lahko iz izvornega programa razbrali namen posameznih podatkovnih tipov ter tako ustrezno prilagodili zapis (*big to little endianness*). Tako bi razbremenili programerje, saj bi odgovornost za prilagajanje prevzel prevajalnik. Naslednja pomembnejša izboljšava bi bila vpeljava dodatnih faz prevajalnika, ki bi poskrbele za optimizacijo prevedenega programa, saj so sedaj končni programi precej daljši kot izvorni.

Literatura

[1] Andrew W. Appel, Jens Palsberg, *Modern Compiler Implementation in Java*, Cambridge University Press, 2004

[2] Leland L. Beck, *System Software: An Introduction to Systems Programming (3rd Edition)*, Addison Wesley, 1996

[3] Richard Blum, *Professional Assembly Language (Programmer to Programmer)*, Wrox, 2005

[4] *Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*, 1997. Dostopno na:
http://download.intel.com/design/intarch/manuals/24319001.pdf

[5] *Intel Architecture Software Developer's Manual Volume 2: Instruction Set Reference*, 1999. Dostopno na:
http://download.intel.com/design/intarch/manuals/24319101.pdf

[6] *Intel Architecture Software Developer's Manual Volume 3: System Programming*, 1999. Dostopno na:
http://communities.intel.com/servlet/JiveServlet/downloadBody/5061-102-1-8118/Pentium_SW_Developers_Manual_Vol3_SystemProgramming.pdf

[7] *Java CUP manual*, Version 0.11a, 2006. Dostopno na:
http://www2.cs.tum.edu/projects/cup/manual.html

[8] *JFlex manual*, Version 1.4.3, 2009. Dostopno na:
http://jflex.de/manual.pdf

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 00042/2012
Datum: 13.04.2012

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **BENJAMIN KASTELIC**

Naslov: **PRETVORNIK MED SIC/XE IN INTEL PENTIUM X86 ZBIRNO KODO**

SIC/XE TO INTEL PENTIUM X86 ASSEMBLY LANGUAGE TRANSLATOR

Vrsta naloge: Diplomsko delo univerzitetnega študija prve stopnje

Tematika naloge:

SIC/XE je hipotetičen računalnik, ki je bil zasnovan z namenom ponazoritve osnovnih konceptov sistemske programske opreme (zbiranje, povezovanje, nalaganje). Ker pa je SIC/XE zgolj hipotetičen računalnik, lahko programsko kodo, napisano v njegovem zbirnem jeziku, poženemo le v simulatorjih. V diplomskem delu preučite možnost pretvorbe SIC/XE zbirne kode v prevedljivo in izvršljivo Intel Pentium x86 zbirno kodo. V ta namen najprej preučite lastnosti obeh procesorjev, način dela s pomnilnikom in registri, načine naslavljanja in podobno. Preučite in opišite tudi vse ukaze zbirnega jezika za SIC/XE in primerno veliko podmnožico ukazov zbirnega jezika za Intel Pentium x86. Opišite način pretvorbe med omenjenima zbirnima jezikoma ter izdelajte program, ki bo to pretvorbo dejansko izvajal.

Mentor:

doc. dr. Tomaž Dobravec

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:

prof. dr. Nikolaj Zimic