UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Urša Levičnik

Zbirnik za hipotetični računalnik SIC/XE

DIPLOMSKO DELO
NA UNIVERZITETNEM ŠTUDIJU

Mentor: prof. dr. Borut Robič

Ljubljana, 2011

Univerza
v Ljubljani

Fakulteta za računalništvo
in informatiko

Tržaška 25
1000 Ljubljana, Slovenija
telefon: 01 476 84 11
faks: 01 426 46 47
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01729/2011
Datum: 15.03.2011

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

Kandidat: **URŠA LEVIČNIK**

Naslov: **ZBIRNIK ZA HIPOTETIČNI RAČUNALNIK SIC/XE**
ASSEMBLER FOR A HYPOTHETICAL COMPUTER SIC/XE

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Opišite in na modelu hipotetičnega računalnika SIC/XE predstavite zbirnik kot del sistemske programske opreme. Opišite arhitekturo procesorja, jo primerjajte z drugimi procesorji in predstavite vse potrebne pogoje za izdelavo zbirnika. Zbirnik realizirajte v višjem programskem jeziku, na koncu pa predstavite še možne izboljšave.

Mentor:

prof. dr. Borut Robič

Dekan:

prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Tukaj se doda original izdane teme diplomskega dela.

I Z J A V A O A V T O R S T V U

diplomskega dela

Spodaj podpisana **URŠA LEVIČNIK**,

z vpisno številko **63040088**,

sem avtorica diplomskega dela z naslovom:

Zbirnik za hipotetični računalnik SIC/XE

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal/-a samostojno pod mentorstvom (naziv, ime in priimek)

  prof. dr. Borut Robič,

- 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 6.3.2011 Podpis avtorice: _____

Kazalo

Seznam uporabljenih kratic in simbolov

SIC – Simplified Instructional Computer
SIC/XE – Simplified Instructional Computer / Extra Equipment
CPE – Centralna Procesna Enota
RISC – Reduced Instruction Set Computer
CISC – Complex Instruction Set Computer

Povzetek

Prvi računalniki so bili namenjeni, kot je razvidno že iz besede same, računanju. Da so operaterji (tako so se imenovali takratni uporabniki računalnikov) lahko upravljali z računalniki, so morali računalnik na nek način popraviti, sprogramirati, da so lahko izračunali dani problem. V teh primerih so imeli operaterji neposreden dostop do stroja, s tem, ko so računalnik nastavili tako, da je izračunaval nek problem, pa lahko rečemo, da so ga strojno sprogramirali.

Dandanes so računalniki veliko bolj razširjeni in so presegli prvotni namen uporabe, ki je bil hitrejše in učinkovitejše reševanje za človeka preobsežnih problemov. Večina ljudi, ki računalnike uporablja, se pravzaprav ne zaveda, kaj se dogaja za »miško, tipkovnico in ekranom«, od računalnika pričakujejo le, da bodo z njim na lažji način rešili trenutni problem. Za tako delo jim razumevanje procesov, ki se dogajajo v zakulisju, ni potrebno.

Da se lahko ljudje osredotočajo na svoje delo, mora med strojnim in aplikacijskim delom računalnika obstajati neka vez, ki skrbi, da se ta dela razumeta med seboj. Sistemska programska oprema je program, ki se ves čas izvaja na računalniku in operira s strojno opremo, medtem ko se lahko uporabnik posveča le svojemu delu.

Pod sistemsko programsko opremo vključujemo zbirnike, nalagalnike, povezovalnike, prevajalnike, operacijske sisteme ... Vse, kar nam omogoča, da nam ob delu z računalnikom ni potrebno razmišljati o njegovi arhitekturi.

V tem delu se osredotočim na del sistemske programske opreme, ki ga poznamo pod imenom zbirnik. V sklopu tega opišem zgradbo procesorja ter predpotrebne pogoje, ki jih potrebujemo za izdelavo zbirnika. Vmes primerjam arhitekturo SIC/XE s procesorji, s katerimi se srečujemo v vsakdanjem življenju. Vsa ta dognanja predstavim na konkretnem primeru programa v višjem programskem jeziku, ki simulira zbirnik za hipotetični procesor SIC/XE. Na koncu predstavim še možne izboljšave, ki jih lahko vpeljemo pri zbirniku.

Ključne besede:

zbirnik, sistemska programska oprema, hipotetični računalnik, SIC/XE

Abstract

As the name itself suggests, the purpose of the first computers was computing. The operators (as the users were then called) that were working with computers had to in a way fix or program the machine in order to solve a given problem. Operators had a direct access to the machine and the work they were doing can be called hardware programming.

Nowadays, computers have become much more common and are no longer used only for solving mathematical problems that were to difficult for a person to solve. Computer users rarely see beyond the »mouse, keyboard and the monitor« and only expect from their computers to solve their problems more easily. For this kind of work, they do not need to understand the processes that are going on in the background.

In order for the users to be blissfully ignorant and can focus only on their work, a bond between the hardware and the application software must exist. System software is basically a program that is being executed in a computer from the moment we turn the machine on and is responsible for the communication between the machine and the application program.

The term system software includes different parts: assemblers, linkers, loaders, operating systems etc. Therefore, the term denotes everything that enables the users to forget that computer architecture exists.

In my thesis, I focus on the part of system software known as the assembler. First I describe the architecture of the processor and requirements needed to build an assembler. I also compare the architecture of SIC/XE with architectures of commonly used and known processors. Based on my findings, I create a program that simulates the assembler for a hypothetical processor SIC/XE. In the end, I suggest possible improvements which can be implemented in an assembler.

Key words:

assembler, system software, hypothetical computer, SIC/XE

1. Uvod

Najpogosteje so podatki v računalniku predstavljeni v binarnem številskem načinu. Ker je ta zapis človeku zelo neprijazen, se je pojavil zbirni jezik. Ta je programerjem olajšal delo, saj jim ni bilo več potrebno poznati vseh operacijskih kod ukazov (te so nadomestili mnemoniki) in na roke preračunavati naslovov. Dokler se niso pojavili višji programski jeziki, ki se s pomočjo prevajalnikov prevedejo v zbirni jezik, je bilo programiranje v zbirnem jeziku zelo razširjeno. [3]

Zbirnik je definiran kot program, ki na vhod prejme izvorno kodo in jo prevede v strojno kodo, ob tem pa upošteva arhitekturne značilnosti procesorja.

Slika 1: Definicija zbirnika

2. Arhitektura procesorja SIC/XE

SIC (Simplified Instructional Computer) je hipotetični računalnik, namenjen razumevanju sistemske programske opreme. [1] Že njegovo ime nam pove, da je poenostavljen, torej se z njim izognemo nepomembnim posebnostim, ki jih vsebuje večina resničnih procesorjev.

Obstajata dve različici tega procesorja – SIC ter SIC/XE (Extra Equipment). To delo bo osnovano na procesorju SIC/XE, ki ima vključenih več funkcionalnosti, zato je tudi zbirnik zanj bolj razgiban in vsebuje več funkcij. Izboljšana verzija procesorja je kompatibilna z osnovno verzijo SIC procesorja in na njej lahko uporabimo programe, napisane za osnovno verzijo.

2.1. Pomnilnik

Največja velikost pomnilnika SIC/XE je 1MB (2^{20} bajtov), uporablja 8-bitne bajte, katerikoli trije zaporedni bajti pa tvorijo besedo (word), ki je dolžine 24 bitov. Največja velikost pomnilnika je 1MB (2^{20} bajtov).

2.2. Registri

SIC/XE uporablja 9 registrov. Dva registra sta uporabljena ob naslavljanju, dva sta splošno namenska, ostali imajo posebno namembnost. Vseh 9 registrov je predstavljenih v tabeli 1.

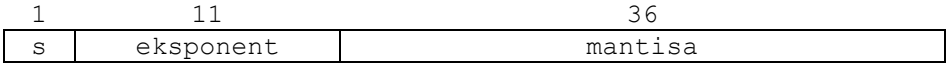
MNEMONIK REGISTRA	ŠTEVILKA REGISTRA	UPORABA
A	0	akumulator; uporablja se za aritmetične operacije
X	1	indeksni register (index register); uporablja se za indeksno naslavljanje
L	2	povezovalni register (linkage register); ukaz JSUB v ta register shrani povratni naslov
B	3	bazni register (base register); uprablja se za naslavljanje
S	4	splošno namenski register; nima posebnega namena uporabe
T	5	splošno namenski register; nima posebnega namena uporabe
F	6	akumulator za plavajočo vejico (48 bitni)
PC	8	programski števec (Program Counter); vsebuje naslov ukaza, ki se bo izvedel naslednji
SW	9	statusni register (Status Word); vsebuje več informacij, vključno s CC (Condition Code)

Tabela 1: Seznam registrov pri SIC/XE

2.3. Predstavitev operandov

Cela števila so shranjena v binarnem zapisu dolžine 24-bitov, negativna števila so zapisana z dvojiškim komplementom. Znaki so zapisani z 8-bitno ASCII kodo.

Pri procesorju SIC/XE je možen zapis operanda tudi v 48-bitnem podatkovnem tipu plavajoča vejica.



Mantisa je število med 0 in 1.Vloga eksponenta je, da pove, na katerem mestu stoji decimalna vejica. Eksponent je število med 0 in 2047.
Če rečemo, da ima eksponent vrednost *e* in mantisa vrednost *m*, je absolutna vrednost zapisanega števila *f*.

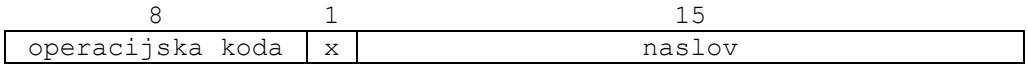
$$f = m*2^{(e-1024)} \qquad (1)$$

Ali bo število negativno ali pozitivno, določa predznak *s*. Če *s* = 0, je število pozitivno, če *s* = 1, potem je število negativno.

Število 0 je predstavljeno tako, da je vseh 48 bitov postavljenih na 0.

2.4. Formati ukazov

Procesor SIC ima le en 24-bitni format ukazov:



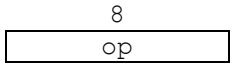
Ker je pomnilnik tega procesorja velik le 32768 (2^{15}) bajtov, to pomeni, da dolžina naslova ne bo večja od 15 bitov.

Pri procesorju SIC/XE je pomnilnik večji - 1MB (2^{20} bajtov). Ker je naslov v teh primerih daljši od 15 bitov, formata ukazov, kot je pri procesorju SIC, ne moremo več uporabljati. Za zapis daljših naslovov imamo dve možnosti: ali uporabimo neko vrsto relativnega naslavljanja ali povečamo naslovno polje na 20 bitov.
Pri procesorju SIC/XE sta mogoči obe možnosti (formata 3 in 4). Imamo pa tudi dva formata, ki ne uporabljata pomnilniških naslavljanj – format 1 in format 2.

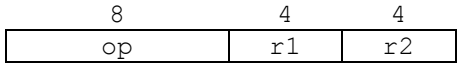
Biti *n*, *i*, *x*, ter *p* se uporabljajo za naslavljanje, bit *e* pa se uporablja za razlikovanje med formatoma 3 in 4. Če *e* = 0, potem je to format 3, če *e* = 1, potem je format 4.

Formati ukazov so torej:

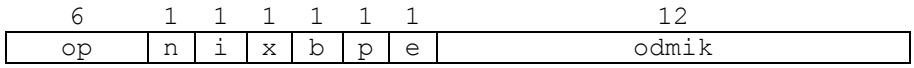
- format 1 (dolžina 1 bajt):



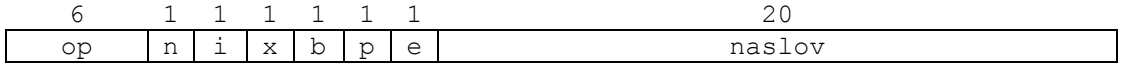
- format 2 (dolžina 2 bajta):



- format 3 (dolžina 3 bajti):



- format 4 (dolžina 4 bajti):



2.5. Tipi ukazov

Osnovna zbirka ukazov SIC/XE vsebuje ukaze za *load* in *store* registrov (LDA, LDX, LDB, STA, STX, STB itd.), ukaze za aritmetične operacije (ADD, SUB, MUL, DIV ...). Vse aritmetične operacije uporabljajo register A in besedo v pomnilniku, rezultat pa se shrani nazaj v register A. Ukaz COMP primerja register A z besedo v pomnilniku in nastavi CC (condition code), ki pove rezultat operacije (<, >, =). Prav tako obstajajo pogojni skoki (JLT, JEQ, JGT), ki za izvedbo pregledajo, kako je nastavljen CC.
Dva ukaza sta namenjena podprogramom: JSUB za skok v podprogram (povratni naslov se zapiše v register L) ter RSUB, ki se vrne iz podprograma na naslov, zapisan v registru L.
Ker SIC/XE pozna zapis števil v plavajoči vejici, obstajajo tudi ukazi za aritmetično računanje s temi števili (ADDF, SUBF, MULF, DIVF).
Obstajajo tudi registrsko-registrski ukazi – to so ukazi, kjer so operandi zapisani v registrih (za registrsko-registrske aritmetične operacije npr. ukazi ADDR, SUBR, MULR, DIVR).

Popoln seznam ukazov je priložen v prilogi A.

2.6. Načini naslavljanj

SIC/XE pozna naslednje načine naslavljanj:

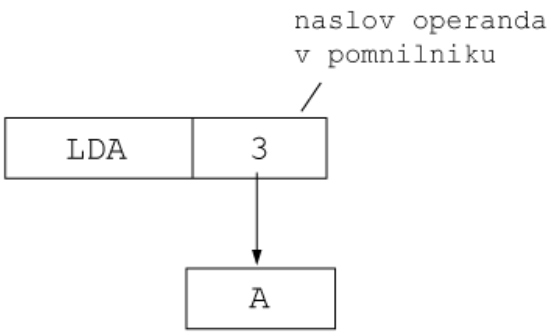
- takojšnje naslavljanje,
- posredno naslavljanje,
- enostavno naslavljanje,
- neposredno naslavljanje,
- bazno relativno naslavljanje ter PC-relativno naslavljanje,
- indeksno naslavljanje.

2.6.1. Takojšnje naslavljanje (immediate addressing)

Pri tem načinu naslavljanja je operand podan z vrednostjo. Operand se ob prevajanju ukaza zapiše v strojno kodo ukaza, torej se v procesor prenese skupaj z ukazom. Dodaten dostop do pomnilnika pri tem naslavljanju ni potreben. [2]

Pri SIC/XE se to naslavljanje uporablja pri formatu 3 in 4. V izvorni kodi ga prepoznamo po znaku #, ki stoji pred operandom. Ob prevajanju v strojno kodo moramo pri tem naslavljanju paziti na bita *i* ter *n*, ki ju nastavimo tako:
$i = 1$, $n = 0$.

Primer uporabe naslavljanja `LDA #3` je predstavljen na sliki 1.



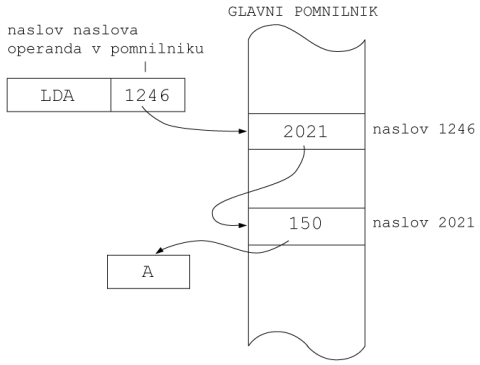
Slika 2: Takojšnje naslavljanje

2.6.2. Posredno naslavljanje (indirect addressing)

Tu je naslov pomnilniškega operanda v ukazu podan posredno preko neke druge vrednosti. Druga vrednost je lahko v glavnem pomnilniku (govorimo o pomnilniškem posrednem naslavljanju) ali pa v registru (govorimo o registrskem posrednem naslavljanju).[2]
SIC/XE uporablja le pomnilniško posredno naslavljanje (slika 2). V ukazu je naslov pomnilniške besede, na katerem je shranjen pomnilniški naslov operanda. V primerjavi z

neposrednim naslavljanjem je pri tem naslavljanju potreben še dodaten dostop do pomnilnika (pri neposrednem je potreben en dostop do pomnilnika, tukaj sta potrebna dva).

To naslavljanje v izvorni kodi prepoznamo po znaku @, ki stoji pred operandom. Ob prevedbi v strojno kodo, moramo postaviti bita *i* in *n* na *i* = 0 ter *n* = 1.



Slika 3: Posredno naslavljanje

2.6.3. Enostavno naslavljanje (simple addressing)

Tako se pri SIC/XE označuje naslavljanje, ki ni niti takojšnje niti posredno. Ne označuje ga poseben znak, ob prevajanju v strojno kodo moramo paziti, da bita *n* in *i* postavimo na 1.

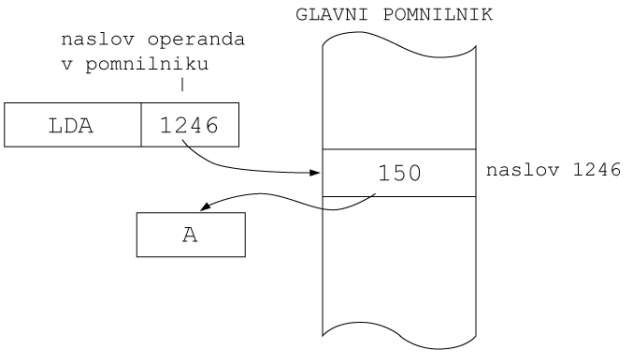
2.6.4. Neposredno naslavljanje (direct addressing)

V ukazu je operand podan z naslovom. Način naslavljanja je tipičen tudi za registrske operande. Takrat se imenuje registrsko naslavljanje – v ukazu je operand podan z naslovom registra v CPE. [2]

To je najpogostejši način naslavljanja pri SIC/XE in se ga ne označuje s posebnim znakom. Uporabljamo ga tako v kombinaciji z enostavnim naslavljanjem kot s posrednim in takojšnjim.

Naslovov ponavadi ne podajamo direktno s številko, temveč kot operand uporabimo labelo, ki ji pripada nek naslov.

Ob prevajanju v strojno kodo ukaza formata 3 so biti *x, b, p* ter *e* postavljeni na 0, če pa uporabljamo format 4, pa se bit *e* postavi na 1.



Slika 4: Neposredno naslavljanje

2.6.5. Bazno relativno naslavljanje (base relative addressing)

To je eno izmed relativnih naslavljanj pri SIC/XE. To naslavljanje je registrsko posredno naslavljanje. Dejanski naslov se izračuna tako, da vsebini baznega registra B prištejemo odmik: [2]

$$dejanski\ naslov = (B) + odmik\ (0<=odmik<=4095) \quad (2)$$

Ta tip naslavljanja se uporablja le pri formatu 3 (pri formatu 4 ni potrebe, saj nam že sam format zagotovi, da lahko uporabimo dovolj dolg naslov v operandu, s katerim zajamemo celoten pomnilniški prostor). Odmik je 12-bitno nepredznačeno število, ob uporabi naslavljanja pa se bita *b* ter *p* postavita na vrednosti $b = 1$, $p = 0$.

Za uporabo baznega relativnega naslavljanja mora poskrbeti programer sam z uporabo psevdoukaza BASE. Ta psevdoukaz sproži zapis trenutnega naslova v bazni register B ter »vključi« bazno naslavljanje. Če želi programer register B uporabiti v druge namene, ga mora sprostiti z ukazom NOBASE, s čimer pove tudi, da se bazno naslavljanje ne bo uporabljalo (do novega zapisa ukaza BASE).

2.6.6. PC-relativno naslavljanje (program counter relative)

Je druga oblika relativnega naslavljanja, ki ga pozna procesor SIC/XE. Kot pri baznem relativnem naslavljanju se tudi ta tip naslavljanja uporablja le pri formatu 3. Odmik je tu prav tako 12-bitno število, le da je predznačeno. Negativno število je predstavljeno z dvojiškim komplementom. Tu se kot register za izračun naslova uporabi register PC, ki v času izvajanja vsebuje trenutni naslov v programu.
To naslavljanje prav tako operira z bitoma *b* in *p*, pri čemer jih postavi na $b = 0$ ter $p = 1$.

$$dejanski\ naslov = (PC) + odmik\ (-2048<=odmik<=2047) \quad (3)$$

Bazno in PC-relativno naslavljanje se uporabljata skupaj z enostavnim, posrednim ter takojšnjim naslavljanjem. Pri obeh je dobro omeniti še to, da med njima izbira zbirnik glede na nastavitve programerja.

2.6.7. Indeksno naslavljanje (indexed addressing)

Indeksno naslavljanje v izvorni kodi določimo tako, da za operandom postavimo vejico ter znak X. Tak zapis pomeni, da želimo pri ukazu uporabiti indeksno naslavljanje. Dejanski naslov se izračuna tako, da se operandu prišteje vsebina registra X. Seveda mora programer poskrbeti za predhodno nastavitev registra X. [2]

$$dejanski\ naslov = operand + (X) \tag{4}$$

Indeksno naslavljanje se lahko uporablja skupaj z enostavnim naslavljanjem, lahko tudi skupaj z baznim relativnim naslavljanjem ter PC-relativnim naslavljanjem pri formatu 3 ter z enostavnim naslavljanjem formata 4. Ob uporabi naslavljanja se postavi bit x na 1.

Indeksnega naslavljanja ne moremo uporabiti skupaj s takojšnjim ter posrednim naslavljanjem, po želji pa ga uporabljamo pri enostavnem naslavljanju skupaj z baznim ter PC-relativnim naslavljanjem.

3. Primerjave SIC z drugimi (realnimi) procesorji

V praksi se redko srečamo s procesorjem, ki je tako eleganten kot SIC/XE. Vsako podjetje, ki izda svoj procesor, zaradi različnih razlogov vgradi v arhitekturo neke posebnosti. Pa naj bo to zaradi cene, ki je na ta račun nižja, zaradi združljivosti s prejšnjimi verzijami ali zaradi naknadnih izboljšav. Prav te posebnosti pa so tiste, ki nam otežujejo delo. Že pri procesorju SIC/XE moramo biti pozorni na kompatibilnost z njegovo prejšnjo verzijo SIC.

Pisanja zbirnika se je vedno potrebno lotiti od začetka za vsak procesor posebej, saj je zahtevnost zbirnika močno odvisna od arhitekture samega procesorja.

V čem pa so si procesorji tako različni? Na to vprašanje se ne da enostavno odgovoriti, temveč je potrebno razjasniti več vidikov.

3.1. RISC in CISC

Procesorje lahko na kratko razdelimo v dve skupini: CISC in RISC. CISC pomeni Complex Instruction Set Computing, RISC pa je kratica za Reduced Data Set Computing. Torej imajo procesorji, ki spadajo v CISC skupino, večje število ukazov kot RISC procesorji. Na prvi pogled bi lahko rekli, da več kot je ukazov, boljše je, saj imajo programerji več svobode in za vsako operacijo lahko najdejo svoj ukaz, za kar bi morali pri RISC procesorju sicer uporabiti množico ukazov. Vendar ni tako. Večje število ukazov pri CISC procesorjih potegne za seboj množico značilnosti. Čeprav lahko bolj obsežne operacije izvedemo z enim ukazom, so pri RISC procesorjih ukazi ponavadi preprostejši, njihova skupna velikost pa je praviloma večja od enega samega ukaza pri CISC. To pomeni, da je pri RISC računalnikih potrebnih več dostopov do glavnega pomnilnika, kar pa spet postane zanemarljivo, ko se pričnemo pogovarjati o predpomnilnikih. Manjši nabor ukazov pri RISC računalnikih povzroči tudi manj razgibane tipe ukazov ter posledično manjše število načinov naslavljanja. [2]

Glede na karakteristike SIC/XE lahko rečemo, da le-ta spada med RISC računalnike. Poleg tega v to skupino spadata tudi študentom naše fakultete dobro znana ARM procesor ter Motorilin 68HC11. V to skupino spada tudi PowerPC. Ravno zaradi svoje arhitekturne preprostosti so ti procesorji velikokrat uporabljeni v študijske namene.

Med najbolj znanimi CISC računalniki pa so Intelova serija x86, Pentium ter starejši, IBM 370, ter v zdajšnjem času ne več tako poznan VAX.

3.2. Pomnilnik, registri ter načini naslavljanja

Večina procesorjev uporablja 8-bitne bajte, velikost pomnilnika pa se močno razlikuje od procesorja do procesorja, glede na letnico izdelave in njegovo namembnost. Velikost pomnilnika vpliva na velikost naslovnega prostora, torej tudi na načine naslavljanj. Pri RISC

procesorjih so naslavljanja večinoma registrska in ne pomnilniška, večina dostopov do pomnilnika je ponavadi le z *load* in *store* ukazi.
Pentium, kot predstavnik CISC procesorjev, uporablja naslednje načine naslavljanja:
neposredno, registrsko operandno (register operand), naslavljanje z odmikom, bazno, bazno z odmikom, skalirano indeksno z odmikom (scaled index with displacement), bazno z indeksnim in odmikom, bazno skalirano indeksno z odmikom (base scaled index with displacement), relativno naslavljanje.

PowerPC uporablja neposredno naslavljanje ter neposredno indeksno naslavljanje. Oba se kombinirata z baznim naslavljanjem.

ARM9 uporablja več različic naslavljanj: za naslavljanje pomnilnika uporablja t. i. *pre-indexed mode*, *pre-indexed with writeback mode* ter *post-index mode*. Poleg tega pa uporablja še naslavljanja: neposredno naslavljanje, naslavljanje z odmikom v registru ter relativno naslavljanje.

Načini naslavljanj se močno razlikujejo od procesorja do procesorja, čeprav v osnovi poznamo le nekaj tipov naslavljanj. Ponavadi posamezni prodajalci drugače poimenujejo načine naslavljanj ali pa združijo nekaj naslavljanj in jim dajo neko drugačno ime.

Z registri je drugače. Razlike so le v številu registrov ter njihovi dolžini. Predvideva se, da ima vsak procesor poleg splošno namenskih registrov tudi statusne registre, vprašanje je le, ali se nahajajo med ostalimi registri ali jih obravnavamo posebej.

ARM9 na primer uporablja 16 programsko dostopnih 32-bitnih registrov. Med njimi so tudi registri, ki imajo posebno uporabo, npr PC (program counter, stack pointer ...), ki pa niso posebej poimenovani.

RISC procesorji imajo le eno dolžino ukazov zaradi cevovodnega izvajanja ukazov. ARM9 uporablja 36 različnih načinov formatov ukazov. Na splošno imajo RISC procesorji veliko število formatov ukazov z zelo različnimi načini naslavljanja.

3.3. Ukazi ter formati ukazov

Kot že omenjeno v razliki med RISC ter CISC procesorji imajo CISC procesorji veliko število ukazov. Čeprav imajo RISC procesorji manjše število ukazov, to ne pomeni, da z njimi ne moremo sprogramirati istih stvari kot s CISC procesorji. Včasih so rešitve bolj elegantne ter hitrejše zaradi manjšega nabora ukazov.

ARM9 uporablja ukaze za skoke, procesiranje podatkov, ukaze za delo s statusnimi registri, ukaze za *load* in *store* operacije, koprocesorske ukaze ter ukaze za generiranje izjem. Vsi ukazi so lahko pogojno izvedljivi – vsak ukaz vsebuje 4-bitno polje s pogoji.

ARM9 uporablja večje število formatov ukazov kot SIC/XE. So pa vsi ukazi enako dolgi (32-bitni), kar olajša uporabo cevovoda. Formati se razlikujejo po številu bitov za podatke ter po številu kontrolnih bitov.

Manjše število ukazov lahko pomeni tudi manj formatov ukazov, kar močno olajša pisanje zbirnika.

Po opisu teh procesorjev vidimo, da so si določene karakteristike procesorjev zelo podobne in da bi lahko za njih začeli pisati zbirnik kot za procesor SIC/XE, le da bi upoštevali drugačne karakteristike. Seveda pa moramo biti pozorni na vse posebnosti. Primer: ARM9 ima več načinov delovanja in pri vsakem od njih imajo določeni registri drugačne namembnosti. [6]

Za primer pisanja zbirnika sem uporabila procesor SIC/XE, saj lahko sklepam, da bi se zbirnik za tu omenjene procesorje lahko hitro zapletel, saj je precej zapletena že njihova arhitektura.

4. Priprava na pisanje zbirnika

Glavna naloga zbirnika je prevesti zbirni program v objektni program, ki se nato zapiše v pomnilnik računalnika in se izvaja korak za korakom.

Poleg mnemonikov operacijskih kod se v kodi programa uporabljajo ukazi zbirniku – psevdo ukazi:

- START – določa ime in začetni naslov programa
- END – določa konec programa in navede prvi ukaz, ki se bo izvedel v programu
- BYTE – generira znakovno konstanto ali konstanto v heksadecimalnem zapisu, ki zasede toliko bajtov prostora, kot ga potrebuje
- WORD – generira celoštevilsko konstanto dolžine ene besede
- RESB – rezerviraj prostor za določeno število bajtov
- RESW – rezerviraj prostor za določeno število besed

Prevod programa v strojno kodo pomeni izvajati naslednje funkcije:

1. pretvoriti mnemonike v zbirni kodi v njihove ekvivalente v strojni kodi
2. pretvoriti simbolne operande v ekvivalentne strojne naslove
3. zgraditi prave strojne ukaze (pravilen format)
4. pretvoriti podakovne konstante v programu v njim ekvivalentne strojne reprezentacije (npr. EOF v 454F46)
5. napisati objektni program

4.1. Večprehodnost zbirnika

Vse te funkcije, razen druge, se lahko izvedejo z zaporednim izvajanjem programa. Pri pretvarjanju simbolnih operandov v strojne naslove pride do problema, če se v programu sklicujemo na oznako, preden je le-ta definirana.

Primer:

```
4                LDA   LENGTH
                 …
20    LENGTH     RESW  1
```

V vrstici 4 se sklicujemo na spremenljivko LENGTH, definiramo pa jo šele v vrstici 20. Temu rečemo »vnaprejšnja referenca« (*forward reference*). Torej do 20. vrstice ne vemo, da spremenljivka LENGTH sploh obstaja. Če bi program izvajali vrstico po vrstico, se ukaza v vrstici 4 ne bi dalo sprocesirati, saj ne poznamo naslova, ki bo dodeljen labeli LENGTH.

Ravno zaradi tega problema večina zbirnikov naredi vsaj dva prehoda čez izvorno kodo programa. Naloga prvega prehoda je, da v izvorni kodi poišče vse definicije label in jim dodeli naslove. Šele drugi prehod nato izvede dejanski prevod ukaza.

Prvi prehod:

1. določi naslove vsem labelam v programu
2. shrani vrednosti (naslove), ki so določene z labelami, da se bodo lahko uporabile v 2. prehodu
3. delno izvede procesiranje ukazov zbirniku (to vsebuje procesiranje, ki vpliva na določevanje naslovov, kot npr. določevanje velikosti podatkovnih območij definiranih z BYTE, RESW itd.)

Drugi prehod:

1. sestavi ukaze (tako da prevede operacijsko kodo in poišče naslove)
2. generira podatkovne vrednosti definirane z BYTE, WORD itd.
3. izvede procesiranje preostalih ukazov zbirniku (tistih, ki se še niso izvedli v prvem prehodu)
4. napiše objektni program

Zbirnik mora poleg tega pravilno interpretirati še *psevdo ukaze* (napotke zbiriku). Ti ukazi niso prevedeni v strojne ukaze (čeprav imajo lahko vpliv na strojno kodo), temveč podajajo navodila samemu zbirniku.
Primeri psevdo ukazov so npr. BYTE in WORD, ki povesta zbirniku, naj generira konstante kot del objektnega programa ter RESB in RESW, ki naročata zbirniku, naj na tem naslovu rezervira prostor. Začetek programa označuje psevdo ukaz START, konec programa pa END.

Končni rezultat zbirnika je objektna koda (strojna koda). Ta koda se nato naloži v pomnilnik, kjer se izvaja.
Enostavni objektni program vsebuje tri tipe zapisov:

- glavo (*header*) – v njej je zapisano ime programa, začetni naslov in velikost (dolžina) programa;
- tekst (*text*) – tu so prevedeni ukazi in podatki programa skupaj z navedbami naslovov, kam se morajo ti podatki zapisati;
- konec (*end*) – označuje konec objektnega programa in določi naslov programa, na katerem naj se začne izvajanje programa.

Sestavni deli glave so:

znak 1	H
znaki 2–7	ime programa
znaki 8–13	začetni naslov objektnega programa (heksadecimalni zapis)
znaki 14–19	dolžina programa v bajtih (heksadecimalni zapis)

Sestavni deli zapisa teksta:

znak 1	T
znaki 2–7	začetni naslov objektne kode v tem zapisu (hexa)
znaki 8–9	dolžina objektne kode tega zapisa v bajtih (hexa)
znaki 10–69	objektna koda, predstavljena v heksadecimalnem zapisu (dva stolpca za bajt objektne kode)

Sestavni deli zapisa konca:
znak 1 E
znak 2–7 naslov prvega ukaza v objektnem programu (hexa)

Primer programa:

vrstica	lokacija	izvorna koda			objektna koda
01	0000	COPY	START	0	
02	0000	FIRST	STL	RETADR	17202D
03	0003		LDB	#LENGTH	69202D
04			BASE	LENGTH	
05	0006	CLOOP	+JSUB	RDREC	4B101036
06	000A		LDA	LENGTH	032026
07	000D		COMP	#0	290000
08	0010		JEQ	ENDFIL	332007
09	0013		+JSUB	WRREC	4B10105D
10	0017		J	CLOOP	3F2FEC
11	001A	ENDFIL	LDA	EOF	032010
12	001D		STA	BUFFER	0F2016
13	0020		LDA	#3	010003
14	0023		STA	LENGTH	0F200D
15	0026		+JSUB	WRREC	4B10105D
16	002A		J	@RETADR	3E2003
17	002D	EOF	BYTE	C'EOF'	454F46
18	0030	RETADR	1	1	
19	0033	LENGTH	1	1	
20	0036	BUFFER	4096	4096	
21	1036	RDREC	CLEAR	X	B410
22	1038		CLEAR	A	B400
23	103A		CLEAR	S	B440
24	103C		+LDT	#4096	75101000
25	1040	RLOOP	TD	INPUT	E32019
26	1043		JEQ	RLOOP	332FFA
27	1046		RD	INPUT	DB2013
28	1049		COMPR	A,S	A004
29	104B		JEQ	EXIT	332008
30	104E		STCH	BUFFER,X	57C003
31	1051		TIXR	T	B850
32	1053		JLT	RLOOP	3B2FEA
33	1056	EXIT	STX	LENGTH	134000
34	1059		RSUB		4F0000
35	105C	INPUT	BYTE	X'F1'	F1
36	105D	WRREC	CLEAR	X	B410
37	105F		LDT	LENGTH	774000
38	1062	WLOOP	TD	OUTPUT	E32011
39	1065		JEQ	WLOOP	332FFA
40	1068		LDCH	BUFFER,X	53C003
41	106B		WD	OUTPUT	DF2008
42	106E		TIXR	T	B850
43	1070		JLT	WLOOP	3B2FEF
44	1073		RSUB		4F0000
45	1076	OUTPUT	BYTE	X'05'	05
46			END	FIRST	

Primer prevedenega programa v objektno kodo:

```
HCOPY  00100000107A
T0010001E14103348202900103628103030101548206 13C100300102A0C103900102D
T00101E150C10364820610810334C0000454F46000003000000
T0020391E041030001030E0205D30203FD8205D2810303020575490392C205E38203F
T0020571C1010364C0000F1001000041030E02079302064509039DC20792C1036
T002073073820644C000005
E001000
```

4.2. Podatkovne strukture, ki jih uporablja zbirnik

Zbirnik uporablja dve glavni tabeli, ki sta v pomoč pri prevajanju programa.

OPTAB je tabela operacijskih kod: v njej so zapisani vsi mnemoniki operacijskih kod skupaj s strojnimi kodami. Ob prevajanju se za vsak mnemonik v tej tabeli poišče mnemoniku ekvivalentno strojno kodo, ki se nato zapiše v datoteko z objektno kodo. V kompleksnejših zbirnikih lahko ta tabela vsebuje tudi informacijo o formatu in dolžini ukaza.

V prvem prehodu se v tej tabeli preverja pravilnost mnemonikov operacijskih kod v izvirnem programu. V drugem prehodu se uporablja za prevod mnemonikov v operacijsko kodo. Oba koraka bi se lahko pri enostavnejšem zbirniku, ki ima vse ukaze enake dolžine, izvedla v istem prehodu, pri tem zbirniku pa je potrebno v prvem prehodu pri vsakem koraku vedeti, za koliko je potrebno povečati LOCCTR. LOCCTR je spremenljivka, v kateri je zapisan programski števec (*program counter*).

V drugem prehodu to tabelo potrebujemo, da vemo, kako zgraditi ukaz.

SYMTAB je tabela simbolnih label: v to tabelo se ob prevajanju zapisujejo simbolne labele, ki jih uporablja programer. Lahko vključuje tudi podatke o napakah (npr. podvojitev labele). V prvem prehodu se nove labele zapišejo v tabelo skupaj z naslovi, na katerih se nahajajo, v drugem prehodu pa se te labele poišče in pridobi naslov, s pomočjo katrega se nato zgradi ukaz.
Ta tabela naj bo zgrajena tako, da bo vstavljanje in iskanje po njej čim hitrejše. Ker se iz te tabele zapisi ne brišejo, je hitrost brisanja nepomembna.

Oba prehoda bi lahko brala izvorno kodo programa. Vseeno pa mora med prehodoma obstajati nekakšna komunikacija (vrednosti LOCCTR, napake ...). Zato prvi prehod generira vmesno datoteko (*intermediate file*), ki vsebuje te vmesne podatke. Ta datoteka se nato v drugem prehodu uporabi kot vhodna datoteka, namesto datoteke z izvorno kodo.

4.3. Načini naslavljanja ter njihov vpliv na objektni program

Prevajanje registrsko-registrskih ukazov ne predstavlja nikakršnega problema. Zbirnik prevede mnemonik ukaza v objektno kodo (s pomočjo OPTAB) ter mnemonike registrov v njihovo številsko predstavitev. Ti ukazi so ukazi v formatu 2.

Večina registrsko-pomnilniških ukazov se prevede s pomočjo PC-relativnega ali baznega naslavljanja. Za uporabo baznega naslavljanja moramo uporabiti psevdo ukaz BASE in ko baznega naslavljanja ne želimo več, uporabimo ukaz NOBASE. Tako pri baznem kot pri PC-relativnem naslavljanju mora zbirnik izračunati odmik, ki ga nato vključi v objektno kodo. Seveda mora biti odmik dovolj majhen, da se lahko zapiše v 12-bitno polje v ukazu. To

pomeni, da mora biti med 0 in 4095 (desetiško) pri baznem naslavljanju ter med -2048 in +2047 (desetiško) pri PC-relativnem naslavljanju.

Če je odmik prevelik, se uporabi podaljšan format 4, za kar mora poskrbeti programer sam. V tem primeru odmika ne potrebujemo, saj se v ukaz zapiše direkten naslov. Če tega programer ne naredi, zbirnik najprej poizkuša uporabiti PC-relativno naslavljanje, nato bazno relativno naslavljanje in če oboje odpove, mora zbirnik vrniti napako.

Kako se izračuna odmik za PC-relativno in bazno relativno naslavljanje?
Primer ukaza (ki je na naslovu 0000):

```
02    0000  FIRST STL   RETADR      17202D
```

Izračun odmika je pravzaprav ravno obraten od računanja dejanskega naslova. Med izvajanjem programa na procesorju se PC poveča po prevzemu vsakega ukaza in predno se ta ukaz izvede. Tako med izvajanjem ukaza PC vsebuje naslov naslednjega ukaza (v tem primeru 0003). Ker je spremenljivka RETADR na naslovu 0030, odmik izračunamo: 0030 - 0003 = 002D. Ko se bo program izvajal, se bo dejanski naslov izračunal iz trenutne vrednosti PC in odmika (PC + odmik), kar bo dalo pravilen naslov RETADR (0030).
Ker je bilo to PC relativno naslavljanje, je bit *p* postavljen na 1 (zato 202D) in ker nimamo ne posrednega ne takojšnjega naslavljanja, sta bita *n* in *i* tudi 1 (zato 17 in ne 14, kot je direkten prevod STL v objektno kodo).

1				7				2				0				2				D			
0	0	0	1	0	1	**1**	**1**	0	0	**1**	0	0	0	0	0	0	0	1	0	1	1	0	1
op. koda						n	i	x	b	p	e	naslov											

Primer, če imamo negativen odmik:

```
10    0017        J      CLOOP       3F2FEC
```

Tu je naslov operanda (CLOOP) 0006. Med izvajanjem programa bo PC vseboval naslov 0001A. Odmik, ki ga izračunamo, je 0006 – 001A = $-14_{(hex)}$, kar je z dvojiškim komplenemtom v 12-bitnem polju predstavljeno kot FEC, kar je vrednost, ki se zapiše v objektno kodo. Seveda tudi tu upoštevamo vse kontrolne bite.

Pri baznem naslavljanju bo izračun odmika bolj kot ne enak, le da ob izvajanju programa zbirnik ve, kakšna bo vrednost v registru PC, vrednost baznega registra pa je pod nadzorom programerja. Torej mora pri baznem naslavljanju programer povedati zbirniku, kaj bo bazni register vseboval ob izvajanju programa. To naredi s psevdoukazom BASE. Npr. ukaz BASE LENGTH pove, da bo bazni register vseboval naslov LENGTH, že pred tem ukazom pa se v register B zapiše ta naslov. Zbirnik torej ob BASE ukazu predvideva, da je naslov podan v registru B. Če se bo ta register v programu potrebovalo za kakšen drug namen, mora programer uporabiti drug psevdo ukaz, da register B sprosti – NOBASE. S tem pove zbirniku, da vsebina registra B ni več uporabna za bazno naslavljanje.

Tipični primer baznega relativnega naslavljanja je npr.

```
30    104E        STCH  BUFFER,X    57C003
```

Ker smo pred tem ukazom uporabili psevdoukaz BASE, se ve, da bo register B med izvajanjem vseboval naslov 0033. Naslov labele BUFFER je 0036, zato bo odmik 0036 – 0033 = 0003. V tem ukazu sta nastavljena bita *b* in *x*, kar pomeni, da imamo bazno ter indeksno naslavljanje.

5				7				C				0				0				3			
0	1	0	1	0	1	1	1	**1**	**1**	0	0	0	0	0	0	0	0	0	0	0	0	1	1
op. koda						n	i	x	b	p	e	naslov											

Primer takojšnjega naslavljanja je enostavnejši, saj se ne sklicujemo na lokacijo v pomnilniku, ampak se v ukaz zapiše direktna vrednost operanda.

```
13    0020        LDA   #3     010003
```

0				1				0				0				0				3			
0	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1
op. koda						n	i	x	b	p	e	takojšni operand											

Poleg operanda se v končni ukaz zapiše še bit *i*, ki pove, da se uporablja takojšnje naslavljanje.

```
24    103C        +LDT  #4096       75101000
```

7				5				1				0				1				0				0				0			
0	1	1	1	0	1	0	**1**	0	0	0	**1**	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
op. koda						n	i	x	b	p	e	takojšnji operand																			

Če je operand prevelik, da bi ga zapisali v 12-bitno polje, moramo uporabiti format 4. Če bi bil operand prevelik tudi za 20-bitno polje v formatu 4, takojšnjega naslavljanja ne bi mogli uporabiti.

Obstaja še en način uporabe takojšnjega naslavljanja:

```
03    0003        LDB   #LENGTH     69202D
```

Tu je takojšen operand labela LENGTH. Ker je vrednost tega operanda naslov labele LENGTH, se pri tem ukazu v register B zapiše naslov te labele. Tu imamo kombinacijo PC-relativnega ter takojšnjega naslavljanja.

V vrstici 13 pri ukazu LDA so biti *x*, *b* in *p* 0, zato je dejanski naslov kar odmik.

Pri posrednem naslavljanju ni pravzaprav nič novega – izračuna se odmik, da dobimo dejanski naslov, nato se postavi bit *n*, kar pomeni, da je vsebina na podanem naslovu kar naslov operanda in ne operand sam.

Vrstica 16 programa predstavlja kombinacijo PC-relativnega ter posrednega naslavljanja.

```
16       002A        J     @RETADR     3E2003
```

5. Realizacija zbirnika SIC/XE

Zaradi lažje prenosljivosti med različnimi operacijskimi sistemi sem programirala v višjem programskem jeziku Java.

Na začetku definiram vse spremenljivke in tabele, ki jih potrebujem.

Definiram štiri podatkovne strukture, ki so tipa HashMap. [5] HashMap je podatkovna struktura, ki uporablja zgoščevalno (*hash*) funkcijo shranjevanje ključev elemetov, ki se v tej podatkovni strukturi uporabljajo. Namesto navadne tabele uporabim to podatkovno strukturo, saj je iskanje ključev in dodajanje novih ključev hitro in učinkovito.

Prva izmed teh podatkovnih struktur je OPTAB – tabela mnemonikov operacijskih kod, skupaj z njihovimi strojnimi kodami in formati ukazov. Vse operacijske kode, ki jih zbirnik uporablja, so zapisane v zunanji datoteki, ki jo program prebere ob vsakem zagonu. S tako implementacijo se lahko hitro in učinkovito spremeni nabor operacijskih kod brez grobega posega v program – spremeni se le zunanjo datoteko.

```
static HashMap OPTAB
```

V zunanji datoteki morajo biti vrednosti zapisane v naslednji obliki:

```
...
DIVF 64 3
DIVR 9C 2
FIX C4 1
...
```

Prva beseda je mnemonik operacijske kode, druga je strojna koda, ki pripada temu mnemoniku, tretja pa je format ukaza.

Vse ukaze formata 3 se lahko uporabi tudi v obliki formata 4, zato je ob ukazih zapisana le vrednost 3 in ne tudi 4.

Vrednosti v OPTAB zapišem tako, da mnemonik uporabim kot ključ, kot vrednost pa se zapišeta strojna koda ter format tega ukaza (vmes stoji presledek).

```
while ((str = br.readLine()) != null){
      st = new StringTokenizer(str);
      OPTAB.put(st.nextToken(), st.nextToken() +" "+ st.nextToken());
}
```

Druga podatkovna struktura je SYMTAB. V to tabelo se bodo ob izvajanju zapisovale vse labele ter njihove vrednosti. Edina posebnost te tabele je, da ob začetku vanjo vpišem še vse registre z njihovimi numeričnimi vrednostmi, sicer pa mora biti tabela pred prvim prehodom prazna.

```
SYMTAB = new HashMap();

SYMTAB.put("A", "0");
SYMTAB.put("X", "1");
SYMTAB.put("L", "2");
SYMTAB.put("B", "3");
SYMTAB.put("S", "4");
SYMTAB.put("T", "5");
SYMTAB.put("F", "6");
SYMTAB.put("PC", "8");
SYMTAB.put("SW", "9");
```

V tretjo (assemblerDirectives) zapišem psevdo ukaze za njihovo lažjo prepoznavo.

```
assemblerDirectives = new HashMap();
```

V četrti so zapisani ASCII znaki ter njihove heksadecimalne vrednosti. [7]

```
ASCII = new HashMap();
```

Komunikacija med prehodoma poteka preko datoteke intermediate.txt, ki jo prvi prehod zgenerira, drugi prehod pa iz nje bere.

Poleg datoteke pa se komunikacija med prehodoma izvaja še s pomočjo nekaj globalnih spremenljivk. Zapomnim si začetni in končni LOCCTR, da laže izračunam dolžino programa, ki se jo zapiše v datoteko z objektno kodo.

```
static int passNO;
static int LOCCTR;
static int LOCCTRLAST;
static int LOCCTRSTART;
static int PROGLENGTH;
static HashMap OPTAB;
static HashMap SYMTAB;
static HashMap assemblerDirectives;
static HashMap ASCII;
static String PROGNAME;
```

5.1. Psevdo koda prvega prehoda

```
Preberi prvo vrstico
IF (vrstica ne vsebuje "START") vrni napako

WHILE (END of file)
      Beri vrstico po vrstico
            Če je vrstica komentar, ignoriraj;
      IF vrstica.hasMoreTokens();
            IF (token ni v mnemonik in ni psevdo ukaz)
                  //token je labela.
                  IF (že v SYMTAB) vrni error,
                  ELSE dodaj labelo v SYMTAB
            ELSE IF (token je psevdo ukaz)
                  IF (RESB) povečaj LOCCTR za število bajtov
                  ELSE IF (RESW) povečaj LOCCTR za število besed
                  ELSE IF (WORD) povečaj LOCCTR za eno besedo
                  ELSE IF (BYTE) povečaj LOCCTR za število bajtov
            ELSE IF (token je mnemonik)
                  IF (podaljšan format) odstrani +
                  IF not mnemonik, vrni error
            ELSE vrni napako, ker beseda ne spada nikamor
      Zapiši LOCCTR pred izvedbo ukaza, LOCCTR po izvedbi ukaza, ukaz  ter
operand v vmesno datoteko;
Konec WHILE
```

Prvi prehod začnem z branjem prve vrstice. V prvi vrstici mora biti zapisan začetek programa, drugače pride do napake.

V začetku programa je zapisano ime programa, psevdoukaz START ter pomnilniška lokacija v heksadecimalni obliki, na kateri naj se prične program.

```
      TEST        START       100A
```

Prve vrstice ne zapišem v vmesno datoteko. Ime programa ter začetni naslov shranim v spremenljivki, ki ju nato uporabim v drugem prehodu. Zato tudi obravnavam to vrstico posebej.

Nato berem izvorno kodo – vrstico po vrstico.

Če je vrstica v izvorni kodi komentar, ga ignoriram. Komentar je vrstica, ki se prične s piko. Komentarji lahko stojijo le v svojih vrsticah. Ne moremo jih postaviti na konec vrstice, v kateri je že ukaz. V tem primeru bo prišlo do napake.

```
.
. to je komentar
.
```

Bistvo prvega prehoda je zapolniti tabelo simbolov (SYMTAB) z labelami in njihovimi naslovi. Zato se v največji meri posvetim temu.

V vsaki vrstici preberem žetone enega za drugim ter vpisujem labele v SYMTAB. Če naletim na mnemonik operacijske kode, pogledam v tabelo OPTAB, katerega formata je in povečam LOCCTR. Če naletim na psevdo ukaz, ustrezno povečam LOCCTR.

Uporabljam tudi spremenljivko *interesting*. Ta se postavi na false, če je bilo v vrstici prebrano, kar je potrebno vedeti v prvem rpehodu. Če je v vrstici zapisano npr.

```
SP1    LDA  #SP1
```

neham brati, ko preberem LDA, ter ustrezno povečam števec. Če je program napisan pravilno, bodo vse labele nekje definirane, če pa je kakšna nedefinirana uporabljena kot operand, bo to preveril drugi prehod.

V prvem prehodu so operandi zanimivi le pri psevdoukazih RESB, RESW, WORD, BYTE, kjer je v operandu definirano, koliko prostora zavzame in glede na to povečamo števec.

```
else if(assemblerDirectives.containsKey(current) ){
      toWrite = toWrite.concat(current+" ");
      tmo = st.nextToken();
      if(current.equals("RESB")){
            LOCCTR = LOCCTR + Integer.parseInt(tmo);
      }
      else if(current.equals("RESW")){
            LOCCTR = LOCCTR + 3*Integer.parseInt(tmo);
      }
      else if(current.equals("WORD"))
      {
            LOCCTR = LOCCTR + 3;
      }
      else if(current.equals("BYTE"))
      {
            if (tmo.charAt(0) == 'C')
            LOCCTR = LOCCTR + (tmo.length() - 3);
      else if (tmo.charAt(0) == 'X')
            LOCCTR = LOCCTR + (int)Math.ceil((double)(tmo.length() -
            3)/2);
      else
      {
            System.out.println("Napaka v vrstici "+currentNumLine+":
            napacno formiran ukaz BYTE!");
      System.exit(1);
      }
}
```

Ko preberem, vsako vrstico zapišem v vmesno datoteko LOCCTR pred izvedbo ukaza, LOCCTR po izvedbi ukaza, ter mnemonik operacijske kode in operand. Label ne zapisujem, saj so varno spravljene v tabeli SYMTAB skupaj s potrebnimi podatki za drugi prehod.

Primer vsebine vmesne datoteke (intermediate file):

```
3F 42 STL RETADR
42 45 LDB #LENGTH
45 45 BASE LENGTH
45 49 +JSUB RDREC
49 4C LDA LENGTH
4C 4F COMP #0
4F 52 JEQ ENDFIL
52 56 +JSUB WRREC
56 59 J CLOOP
59 5C LDA EOF
5C 5F STA BUFFER
5F 62 LDA #3
62 65 STA LENGTH
65 69 +JSUB WRREC
69 6C J @RETADR
6C 6F BYTE C'EOF'
6F 72 RESW 1
72 75 RESW 1
75 1075 RESB 4096
1075 1077 CLEAR X
1077 1079 CLEAR A
1079 107B CLEAR S
107B 107F +LDT #4096
107F 1082 TD INPUT
1082 1085 JEQ RLOOP
1085 1088 RD INPUT
1088 108A COMPR A,S
108A 108D JEQ EXIT
108D 1090 STCH BUFFER,X
1090 1092 TIXR T
1092 1095 JLT RLOOP
1095 1098 STX LENGTH
1098 109B RSUB
109B 109C BYTE X'F1'
109C 109E CLEAR X
109E 10A1 LDT LENGTH
10A1 10A4 TD OUTPUT
10A4 10A7 JEQ WLOOP
10A7 10AA LDCH BUFFER,X
10AA 10AD WD OUTPUT
10AD 10AF TIXR T
10AF 10B2 JLT WLOOP
10B2 10B5 RSUB
10B5 10B6 BYTE X'05'
10B6 10B6 END FIRST
```

V prvem prehodu polovim tudi že večino napak.

5.2. Psevdo koda drugega prehoda

```
Zapiši Header record v datoteko z objektno kodo
WHILE (!endOfCode)
      Zapiši header
      Beri vrstico po vrstico vmesne datoteke
      Zapomni si LOCCTR pred in po izvedbi ukaza (iz datoteke)
      IF (zapis END v kodi) endOfCode = true;
      ELSE IF (zapis BASE v kodi)
            zapomni si naslov,
            baseValidAddress = true
      ELSE IF (zapis NOBASE v kodi) baseValidAddress = false
      ELSE IF(mnemonik)
            IF (mnemonik vsebuje +) odstrani +

            IF (format 1)
                  buildOC = op. koda mnemonika
            ELSE IF (format 2)
                  buildOC = op. koda mnemonika + registri
            ELSE IF (format 3)
                  IF operand začne z # ali @
                        # - nastavi bit I
                        @ - nastavi bit N
                        Ne # in ne @ - nastavi bita I in N
                  IF (operand je labela)
                        Izračunaj odmik
                        IF PC-relativno naslavljanje?
                              Nastavi bit p
                        IF bazno naslavljanje?
                              Nastavi bit b
                  ELSE (operand je številka)
                        Zapiši operand
                  IF (indeksno naslavljanje)
                        Nastavi bit X

            ELSE IF (format 4)
                  IF obstaja operand
                        IF operand začne z # ali @
                              # - nastavi bit I
                              @ - nastavi bit N
                              Ne # in ne @ - nastavi bita I in N
                        IF (operand je labela)
                              zapiši naslov labele
                        ELSE (operand je številka)
                              Zapiši operand
                  ELSE
                        buildOC = ukaz + razširitev z 0
            ELSE IF (psevdo ukaz)
                  IF ukaz = BYTE
                        IF operand začne z X
                              (zapiši to kar je znotraj navednic)
                        ELSE operand začne s C
                              (zapiši vsak znak kot hexa)
                  ELSE IF ukaz = WORD
                        rezerviraj <operand> število besed
                  ELSE
                        Izpiši error
```

```
IF trenutna beseda+trenutna vrstica > 60
      Zapiši trenutno vrstico
      Trenutna vrstice = trenutna beseda
ELSE
      Trenutna vrstica = trenutna vrstica + trenutna beseda
```

V drugem prehodu začnem pisati datoteko z objektnim programom. Ta datoteka mora biti pravilne oblike. Vsebovati mora glavo, tekst in zapis konca.

Najprej zapišem glavo.

- Ime programa je lahko dolgo največ 6 znakov. Če je krajše, se dopolni s presledki.
- Za imenom stoji začetni naslov programa. Dolžina mora biti 6 znakov, če jih nima, se jih dopolni z ničlami.
- Zapiše se še dolžina programa. Tako kot pri začetnem naslovu programa tudi tu dopolnimo z ničlami do dolžine 6 znakov.

Drugi prehod bere vmesno datoteko, ki se je generirala ob koncu prvega prehoda.

Nato pričnem WHILE zanko, ki se sprehaja vse do konca vmesne datoteke. Prva dva žetona v vrstici sta vrednosti LOCCTR pred in po izvajanju ukaza, ki se nahaja v tej vrstici. Naslednja dva žetona sta mnemonik operacijske kode ali psevdo ukaz ter operand.

Najprej preverim, če je programer v kodi slučajno uporabil psevdoukaz BASE, kar pomeni, da imamo bazno naslavljanje. Ta podatek je potreben čimprej, saj se glede na to drugače formira operacijska koda.

Ker je bistvo drugega prehoda izpis objektne kode, pomemben del le-te pa so naslovi, je interpretiranju naslavljanj posvečena večina drugega prehoda.

Pri prvem in drugem formatu sta prevoda v objektno kodo enostavna.

Ukazi prvega formata vsebujejo samo mnemonik operacijske kode in nobenega operanda, zato je tu dovolj le prevedba mnemonika v operacijsko kodo ter zapis le-te v objektni program.

```
if (format == '1' && !extended){
      buildOC = buildOC.concat(opFromMap.substring(0,2));
}
```

Ukazi formata 2 imajo v operandih le registre. Mnemonik operacijske kode se brez nadaljnjih popravkov prevede v operacijsko kodo, zraven pa se zapišejo numerične vrednosti registrov, ki so v operandu.
Če sta v operandu navedena dva registra, ju je potrebno ločiti z vejico.

```
else if(format == '2' && !extended)
{
	buildOC = buildOC.concat(opFromMap.substring(0,2));

	if (st.hasMoreTokens()) next = st.nextToken();
	else {
		next = "";
		buildOC = ((OPTAB.get(code)).toString()).substring(0,2);
	}

	if ((comma = next.indexOf(",")) > 0) {
		first = next.substring(0,comma);
		second = next.substring(comma+1,next.length());

		if (SYMTAB.containsKey(first)){
			buildOC =
			buildOC.concat((SYMTAB.get(first).toString()));
		}
		else
		buildOC =
		buildOC.concat(decToHex(Integer.parseInt(first)));

		if (SYMTAB.containsKey(second)){
			buildOC =
		buildOC.concat((SYMTAB.get(second).toString()));
		}
		else
		buildOC = buildOC.concat(decToHex(Integer.parseInt(second)));
	}
	else{
		if (SYMTAB.containsKey(next)){
			buildOC =
		buildOC.concat((SYMTAB.get(next).toString())+"0");
		}
		else buildOC =
		buildOC.cosncat(decToHex(Integer.parseInt(next))+"0");
	}
}
```

Format 3 je daleč najbolj zanimiv, saj lahko pri njem uporabljamo celo vrsto naslavljanj.

Pri programiranju mi je bila v pomoč tabela 2 s tipi naslavljanj.

način naslavljanja	zastavice						zapis v zbirnem jeziku	izračun dejanskega naslova
	n	i	x	b	p	e		
enostavno	1	1	0	0	0	0	op c	odmik
	1	1	0	0	0	1	+op m	naslov
	1	1	0	0	1	0	op m	(PC) + odmik
	1	1	0	1	0	0	op m	(B) + odmik
	1	1	1	0	0	0	op c,X	odmik + (X)
	1	1	1	0	0	1	+op m,X	naslov + (X)
	1	1	1	0	1	0	op m,X	(PC) + odmik + (X)
	1	1	1	1	0	0	op m,X	(B) + odmik + (X)
	0	0	0	-	-	-	op m	b/p/e/odmik
	0	0	1	-	-	-	op m,X	b/p/e/odmik + (X)
posredno	1	0	0	0	0	0	op @c	odmik
	1	0	0	0	0	1	+op @m	naslov
	1	0	0	0	1	0	op @m	(PC) + odmik
	1	0	0	1	0	0	op @m	(B) + odmik
takojšnje	0	1	0	0	0	0	op #c	odmik
	0	1	0	0	0	1	+op #m	naslov
	0	1	0	0	1	0	op #m	(PC) + odmik
	0	1	0	1	0	0	op #m	(B) + odmik

c = konstanta med vrednostmi 0 ter 4095 (ali znan pomnilniški naslov v tem rangu), m = pomnilniški naslov ali konstanta večja od 4095

Tabela 2: Tipi naslavljanj pri SIC/XE

Preverim, če se operand začne z # ali @, kar pomeni, da imamo posredno ali pa takojšnje naslavljanje.

Pri posrednem naslavljanju nastavim bit *n* na 1 (pomeni, da operacijski kodi prištejem 1), pri takojšnjem naslavljanju pa bit *i* (enako, kot da operacijski kodi prištejem 2).
Če se ne uporablja ne eno ne drugo naslavljanje, se nastavita oba bita. Pri tem preverim še, če se uporabi indeksno naslavljanje.

```
if (next.charAt(0) == '#' || next.charAt(0) == '@') {
      if (next.charAt(0) == '#'){
            hash = true;
            addBits = hexToDec(opC) + 1;
      }
      else{
            at = true;
            addBits = hexToDec(opC) + 2;
      }
      next = next.substring(1,next.length());
}
else{
      addBits = hexToDec(opC) + 3;
      if ((indexOfX = next.indexOf(",X")) > 0) {
            next = next.substring(0,indexOfX);
      }
}
```

Po tem že lahko zapišem operacijsko kodo mnemonika v spremenljivko tipa *string*, v kateri gradim operacijske kode posameznih ukazov (na spremembo operacijske kode mnemonika vplivata le bita zastavic *n* in *i*).

Če je operand labela, izračunam odmik ter preverim, ali se uporablja PC-relativno ali bazno naslavljanje ter nastavim primerne zastavice. Če je odmik predolg, vrnem napako. Prevedem odmik v heksadecimalno obliko (če je negativen, ga zapišem v dvojiškem komplementu) ter dodam v *string*, ki se bo zapisal v datoteko z operacijsko kodo.

```
if (SYMTAB.containsKey(next)){
      loc = hexToDec(nextlocctr);
      addr = Integer.parseInt((SYMTAB.get(next).toString()));
      disp =  addr - loc;

      if ((disp >= -2048) && (disp <= 2047)){
            if (disp >= 0)
                  disp = disp | 8192;
            else
                  disp = disp & 12287;
      }
      else if(baseValidAddr)
      {
            disp = addr - base;
            if (disp >=0 && disp <= 4095){
                  disp = disp | 16384;
            }
      }
      else {
            System.out.println("Predolgi naslov!!");
            System.exit(1);
      }
      if(disp < 0)
      {
            hex = (Integer.toBinaryString(disp)).substring(27,31);
      }
      else
      {
            hex = decToHex(disp);
      }
      if (hex.length() < 4){
            for (int i = hex.length(); i < 4; i++){
                  buildOC = buildOC.concat("0");
            }
      }
      buildOC = buildOC.concat(hex);
}
```

Če je operand število, potem le preverim, da ni predolgo, nato ga spremenim v heksadecimalno obliko.

```
else{
      addr = Integer.parseInt(next);

      if (addr > 4095) {
            System.out.println("Predolgi naslov!!");
```

```
			System.exit(1);
		}

		hex = decToHex(addr);
		if (hex.length() < 4){
			for (int i = hex.length(); i < 4; i++){
				buildOC = buildOC.concat("0");
			}
		}
		buildOC = buildOC.concat(decToHex(addr));
}
```

Ko imam zapisano heksadecimalno kodo za zapis v vmesno datoteko, z operacijsko kodo nastavim še bit x za indeksno naslavljanje.

```
if (indexOfX > 0){
		tmpX = hexToDec(buildOC);
		tmpX = tmpX | 32768;
		buildOC = decToHex(tmpX);
}
```

Če ob mnemoniku ni operanda, se namesto operanda v operacijsko kodo zapišejo ničle.

Format 4 je enostavno prepoznati, saj mora programer sam poskrbeti za njegovo uporabo s tem, da pred mnemonik operacijske kode postavi znak +.

Tu imamo 4 različna naslavljanja: enostavno (operand je naslov), indeksno (k operandu dodamo vsebino registra X), posredno (označuje ga znak #) in takojšnje (označuje ga znak @) naslavljanje.

Če je naslavljanje posredno ali takojšnje, nastavimo ustrezna bita (*i* ter *n*), če ni nobeno od teh, nastavimo oba bita. Če je naslavljanje indeksno, nastavimo še bit *x*.

Do tu je vse enako kot pri formatu 3. Razlika je, da ne preverjam PC-relativnega in baznega naslavljanja, saj se tu daljši naslovni prostor rešuje s podaljšanim formatom, torej sta bita *b* in *p* postavljena na 0. Poleg tega je potrebno nastaviti bit *e*, ki označuje podaljšan format.

```
else if(extended)
{
		if(st.hasMoreTokens()) {
			hash = false;
			at = false;

			next = st.nextToken();
			opC = ((OPTAB.get(code)).toString()).substring(0,2);

			if (next.charAt(0) == '#' || next.charAt(0) == '@') {
				if (next.charAt(0) == '#'){
					hash = true;
					addBits = hexToDec(opC) + 1;
				}
				else{
					at = true;
```

```
                        addBits = hexToDec(opC) + 2;
                  }
                  next = next.substring(1,next.length());
            }
            else{
                  addBits = hexToDec(opC) + 3;
                  if ((indexOfX = next.indexOf(",X")) > 0) {
                        next = next.substring(0,indexOfX);
                  }
            }

            buildOC = buildOC.concat(decToHex(addBits));
            buildOC = buildOC.concat("1");

            if (SYMTAB.containsKey(next)){
                  addr = Integer.parseInt( (SYMTAB.get(next).toString()));

                  hex = decToHex(addr);
                  if (hex.length() < 5){
                        for (int i = hex.length(); i < 5; i++){
                              buildOC = buildOC.concat("0");
                         }
                  }
                  buildOC = buildOC.concat(hex);
                  }
            else {
                  addr = Integer.parseInt(next);

                  hex = decToHex(addr);
                  if (hex.length() < 5){
                        for (int i = hex.length(); i < 5; i++){
                              buildOC = buildOC.concat("0");
                         }
                  }
                  buildOC = buildOC.concat(hex);
                            }
            }
      else {
            next = "";
            buildOC = ((OPTAB.get(code)).toString()).substring(0,2) +
            "000000";
      }
}
```

V prvem prehodu gledam psevdoukaze le zato, da vem, za koliko povečati LOCCTR, tu pa je treba generirati objektno kodo, zato še enkrat preverim psevdoukaza BYTE in WORD (RESW in RESB le rezevirata prostor in se ne prevedeta v objektno kodo).

Če se operand pri psevdoukazu BYTE začne z znakom X, potem se v objektno kodo zapiše število v heksadecimalni obliki, ki stoji med znakoma »'«. Če se začne z znakom C, potem se v objektno kodo zapiše heksadecimalen zapis ASCII znakov, ki jih zapišemo v ukaz.
Če je psevdoukaz WORD, potem se v operacijsko kodo zapiše konstanta, ki stoji v operandu.

```
else if(assemblerDirectives.containsKey(current)){
      if(current.equals("BYTE")){
            current = st.nextToken();
            if(current.charAt(0) == 'X'){
                  int l = current.length();
                  buildOC = current.substring(2,l-1);
            }
            else if(current.charAt(0) == 'C'){
                  int l = current.length();
                  String buildascii = "";
                  String c;
                  for (int i = 2; i < l-1; i++){
                        c = Character.toString(current.charAt(i));
                        String xyz = (ASCII.get(c)).toString();
                        buildascii = buildascii.concat(xyz);
                  }
                  buildOC = buildascii;
            }
      }
      else if(current.equals("WORD")){
            addr = Integer.parseInt(st.nextToken());
            hex = decToHex(addr);

            if (hex.length() < 6)
             hex = fillToLength(hex,6);

            buildOC = buildOC.concat(hex);
      }
      else {
            System.out.println("Napaka - labela ne obstaja!");
            System.exit(1);
      }
}
```

Na koncu zapišem generirano objektno kodo ukaza v datoteko. Datoteka ima poseben format. Na začetku vsake vrstice stoji znak T, za njim stoji začetni naslov zapisov v tej vrstici (heksadecimalno), nato dolžina teh zapisov (prav tako heksadecimalno). Nato objektna koda ukazov. Koda ukazov ene vrstice je lahko dolga največ 60 znakov, zato to upoštevam ob zapisu objektne kode v datoteko z objektnim programom.

```
if((hexToDec(nextlocctr) - hexToDec(prevLineStart)) > 255){
       needNewLine = true;
}
if ((charInLine+buildOC.length() > 60) || endOfCode || needNewLine){
      int xxx = hexToDec(lastInLine) - hexToDec(prevLineStart);

      if (!firstline)
            bw2.write(decToHex(xxx) + lineToWrite);
      if (!endOfCode){
            bw2.newLine();
            if (!needNewLine)
                  bw2.write("T"+fillToLength(lastInLine,6));
            else
                  bw2.write("T"+fillToLength(nextlocctr,6));
      }
```

```
      lineToWrite = buildOC;
      if (needNewLine)
            prevLineStart = nextlocctr;
      else
      prevLineStart = lastInLine;
      charInLine = buildOC.length();
      needNewLine = false;
      firstline = false;
}
else if (charInLine+buildOC.length() <= 60){
      lineToWrite = lineToWrite.concat(buildOC);
      charInLine = charInLine + buildOC.length();
      if (buildOC.length() > 0)
            lastInLine = nextlocctr;
}
```

Ko zapišem vse ukaze, zapišem v datoteko z objektno kodo še končni zapis, ki vsebuje znak E, ki mu sledi naslov prvega ukaza v programu.

```
bw2.write("E"+fillToLength(decToHex(LOCCTRSTART),6));
```

6. Izboljšave

Način, ki sem ga izbrala za izdelavo zbirnika, je najpreprostejši. Še bolj preprosto bi bilo, če ne bi bilo potrebno zagotavljati kompatibilnosti s prejšnjo verzijo procesorja, vendar je primer ravno zaradi tega veliko bolj realen, saj si pri resničnih procesorjih ne smemo zamišljati, kako bi jih izboljšali. To je naloga proizvajalcev procesorjev in ne nas, ki želimo za te procesorje izdelati sistemsko programsko opremo.

V mojem zbirniku gledam na izvorno kodo kot na celoto, čeprav hitro opazim, da je le-ta logično razdeljena na več delov z uporabo podprogramov (subroutines) in podatkovnih polj. V objektnem programu, ki ga generira zbirnik, so ukazi zapisani v istem vrstnem redu kot v izvirni kodi. Ni pa vedno potrebno, da je tako. Objektni program lahko zapišemo tudi tako, da vrstni red ukazov v njem ni enak vrstnemu redu izvorne kode. Prav tako lahko izvorno kodo prevedemo tako, da se kreira več samostojnih delov objektnega programa. Uporabljamo terminologijo **programskih blokov**, če govorimo o segmentih kode, ki so razporejeni v različnem vrstnem redu znotraj ene objektne kode, ter o **kontrolnih sekcijah**, če se generira več samostojnih delov objektne kode.

6.1. Programski bloki

Programski bloki so deli izvorne kode, za katere programer želi, da stojijo skupaj. Ob pisanju programa uporablja psevdoukaz USE, s katerim pove zbirniku, naj od tu naprej uporablja novi programski blok. Če programer tega ne uporabi, zbirnik predvideva, da je cel program en sam programski blok brez imena.

Dobra stran programskih blokov je v tem, da lahko programer združi različne odseke programa (npr. združi podatkovna polja ali definicije spremenljivk, čeprav jih uporablja skozi celoten program). Tako programerju ni potrebno pisati izvorne kode, ki bi imela npr. vse definicije spremenljivk na enem mestu, temveč si olajša delo z uporabo programskih blokov. Z uporabo programskih blokov si nekoliko zmanjšamo probleme pri naslavljanju. Ker lahko postavimo vsa večja podatkovna polja na konec objektnega programa, nam v določenih primerih ni več potrebno uporabljati »podaljšanega« formata za naslavljanje. Poleg tega ne potrebujemo več baznega naslavljanja, saj se sedaj z uporabo programskih blokov ti deli programa pri izračunavanju naslova obravnavajo malce drugače.

Največja sprememba v prvem prehodu je ta, da ne uporabljamo več le enega programskega števca LOCCTR, temveč uporabljamo po en programski števec za vsak programski blok. Ko zbirnik naleti na začetek programskega bloka, se njegov programski števec postavi na 0 in se povečuje le, dokler se nahajamo znotraj tega programskega bloka. Naslov vsake labele je sedaj relativen glede na začetek programskega bloka, v katerem se nahaja, in ne več glede na začetek programa. Ko se te labele zapišejo v SYMTAB, se ob njih zapiše naslov programskega števca programskega bloka, kateremu pripada labela, ter ime ali številka programskega bloka.

Na koncu prvega prehoda imamo dodatno tabelo, v kateri so shranjena imena vseh programskih blokov, skupaj z njihovimi številkami, začetnim naslovom ter dolžino bloka.

V drugem prehodu zbirnik še vedno potrebuje naslove, ki so relativni glede na začetek programa in ne glede na začetek programskega bloka. Zato naslove enostavno izračuna s pomočjo informacije iz SYMTAB – sešteje lokacijo labele (ta je relativna na začetek programskega bloka) ter naslov programskega bloka, kateremu ta labela pripada. Naslov programskega bloka drugi prehod dobi iz tabele, ki je bila generirana v prvem prehodu.

Razlika v objektnem programu je sedaj ta, da se ob vsaki spremembi programskega bloka zapiše nov tekstovni zapis (*text record*), čeprav je v trenutnem še prostor za zapis. Naslov začetka vsakega zapisa vsebuje tako začetni naslov programskega bloka kot relativno lokacijo kode znotraj bloka. V objektni kodi si tekstovni zapisi ne sledijo več po vrsti, po naslovih, temveč so zmešani. To ne predstavlja problema, saj nalagalnik preprosto naloži vsak tekstovni zapis na njegov pravi naslov. Če uporabljamo v enem programskem bloku le ukaze, ki ne generirajo objektne kode, temveč le rezervirajo ustrezen pomnilniški prostor, se bo to seveda upoštevalo ob izračunavanju naslovov in v ta del pomnilnika se zagotovo ne bo prenesla objektna koda.

Z uporabo programskih blokov tako zadostimo tako človeku kot stroju, saj je vrstni red zapisa kode, ki je optimalen za stroj, pogosto težko berljiv človeku. Tako pa izvorna koda ustreza človeku, objektna pa stroju. Lahko bi rekli: »Volk sit in koza cela«.

6.2. Kontrolne sekcije

Kontrolna sekcija je del programa, ki ohrani svojo identiteto (samostojnost) po prevedbi v objektno kodo. Vsaka kontrolna sekcija je lahko naložena in premaknjena v pomnilniku neodvisno od drugih. Kontrolne sekcije se ponavadi uporablja za podprograme ali logične dele nekega programa. Posledica uporabe kontrolnih sekcij v programu je fleksibilnost.

Da zagotovimo neodvisnost posamezne kontrolne sekcije od drugih, je potrebno zagotoviti nekakšno povezavo med njimi – tukaj mislimo predvsem na reference na ukaz ali uporabo spremenljivke, ki se nahaja v neki drugi kontrolni sekciji, saj zbirnik ne ve, kje se bo neka kontrolna sekcija nahajala med izvajanjem.

Kontrolno sekcijo v izvorni kodi definiramo s psevdoukazom CSECT. Vsaki kontrolni sekciji določimo ime.

Sklice na ukaze ali spremenljivke, ki se nahajajo v drugi kontrolni sekciji, rešimo s pomočjo t. i. zunanjih referenc. Uporabimo dva psevdoukaza – EXTDEF ter EXTREF. Za ukazom EXTDEF naštejemo vse t. i. zunanje labele (*external symbols*), ki so definirane v tej kontrolni sekciji ter se bodo uporabljale v drugi kontrolni sekciji. Za ukazom EXTREF pa naštejemo vse labela, ki jih uporabljamo v tej kontrolni sekciji, vendar so definirane v drugih kontrolnih sekcijah.

Ker zbirnik ob izdelavi objektne kode ne ve, kje se bodo kontrolne sekcije ob izvajanju nahajale, tudi ne ve, kakšne vrednosti bodo imele zunanje labele. Zato namesto njihove vrednosti v objektno kodo zapiše vrednost nič. Namesto ničelne vrednosti pa bo nalagalnik poskrbel, da se bo tu upošteval pravi naslov. Zaradi tega uporaba relativnega naslavljanja ni mogoča pri ukazih, ki uporabljajo zunanje labele. Za naslavljanje moramo tu torej vedno uporabiti razširjen format (format 4). Relativno naslavljanje pa je za ostale ukaze, ki ne uporabljajo zunanjih label, še vedno na razpolago.

Zbirnik mora vedeti, v kateri kontrolni sekciji je definirana posamezna labela. Vsaka labela v drugi kontrolni sekciji, na katero se sklicujemo, mora biti označena kot zunanja referenca (EXTREF), če ni, potem vrnemo napako. Imena label, ki niso zunanje, pa se lahko ponavljajo v vseh kontrolnih sekcijah brez napake.

Da bo nalagalnik upošteval prave naslove, mora zbirnik podati še nekaj dodatnih informacij. Za to se v objektni kodi definira 3 nove zapise: definicijski zapis (*define record*), zapis napotka (*refer record*) ter modifikacijski zapis (*modification record*).

Definicijski zapis poda vse definirane zunanje labele v tej kontrolni sekciji (labele definirane z EXTDEF).

znak 1:	D
znaki 2–7:	ime zunanje labele, definirane v tej kontrolni sekciji
znaki 8–13:	relativni naslov labele v tej kontrolni sekciji (heksadecimalno)
znaki 14–73:	enaka informacija, kot je zapisana v znakih 2–14, le za druge zunanje labele

Zapis napotka poda vse labele, uporabljene v tej kontrolni sekciji, ki so definirani drugje (labele definirane z EXTREF).

znak 1:	R
znaki 2–7:	ime zunanje labele iz druge kontrolne sekcije, na katero se sklicujemo iz te kontrolne sekcije
znaki 8–73:	imena ostalih zunanjih label, na katere se sklicujemo v tej kontrolni sekciji

V modifikacijskem zapisu se nahajajo vse ostale informacije, ki jih potrebujemo za povezovanje kontrolnih sekcij.

znak 1:	M
znaki 2–7:	začetni naslov polja, ki bo potrebno modifikacije, glede na začetek te kontrolne sekcije (heksadecimalno)
znaki 8–9:	dolžina polja, ki bo potrebno modifikacije
znak 10:	modifikacijska zastavica (+ ali –)
znaki 11–16:	zunanja labela, katere vrednost se bo prištela ali odštela od polja

Individualnost, kot pomembna lastnost kontrolnih sekcij, se v objektni kodi kaže tako, da ima vsaka kontrolna sekcija svojo objektno kodo. Torej za vsako kontrolno sekcijo obstaja zapis glave, teksta ter konca. Kontrolno sekcijo, ki nima nikakršne povezave z drugimi, lahko torej obravnavamo kot popolnoma samostojen program, ki ne potrebuje dodatnih zapisov v objektni kodi.

7. Zaključek

V svojem diplomskem delu sem predstavila izdelavo zbirnika za hipotetični procesor SIC/XE in ga implementirala v programskem jeziku Java. Ob pisanju zbirnika sem razmišljala, kako bi bil zbirnik drugačen za kakšen drug procesor, ter ugotovila, da je del zbirnika močno odvisen od arhitekture, medtem ko osnovna ideja ostaja vedno enaka.

Pisanje zbirnika za druge procesorje se torej ne bi močno razlikovalo od pisanja zbirnika za SIC/XE. Na začetku je potrebno poznati arhitekturo procesorja, nabor ukazov, načine naslavljanj itd. Predno pričnemo s pisanjem, si naredimo plan – kakšne podatkovne strukture bomo uporabili, koliko prehodov bo imel zbirnik, kako bomo upoštevali naslavljanja, ali bomo uporabljali kotrolne sekcije ali programske bloke. Šele nato se lotimo samega programiranja. Že v naprej moramo dobro poznati vse podrobnosti v arhitekturi, saj nam lahko že manjša stvar močno vpliva na zbirnik. Vedeti moramo tudi, ali želimo na tem procesorju uporabljati programe iz starejših verzij in to upoštevati v zbirniku.

Zbirnik je le manjši del sistemske programske opreme. Z njim začnemo graditi sistemsko programsko opremo za nek procesor. Že pri njem se moramo zavedati, da je le gradnik, in da se bo strojna koda, ki jo generira, naložila v pomnilnik (to opravlja nalagalnik – *loader*), zgenerirale se bodo povezave med deli kode (za to skrbi povezovalnik – *linker*), nato pa se bo izvedla.

Seznam slik

Seznam tabel

Literatura

[1] Leland L. Beck, System Software, An Introduction to Systems Programming, Addison Wesley Longman, Inc., 1997.

[2] Kodek, Dušan, Arhitektura in organizacija računalniških sistemov, Bi-TIM, 2008.

[3] Assembly language
URL: http://en.wikipedia.org/wiki/Assembly_language

[4] Tovarniški uporabniški priročnik za AT91SAM9260. Dostopno na:
URL: http://laps.fri.uni-lj.si/fri-sms/listine.php

[5] HashMap v Javi. Dostopno na:
URL: http://download.oracle.com/javase/6/docs/api/

[6] ARM načini delovanja. Dostopno na:
URL: http://simplemachines.it/doc/arm_inst.pdf

[7] ASCII heksadecimalne kode znakov. Dostopno na:
URL: http://www.asciitable.com/

Priloga A

Tabela ukazov

Mnemonik	Format	Operacijska koda	Opis
ADD m	3/4	18	A <- (A) + (m..m+2)
ADDF m	3/4	58	F <- (F) + (m..m+5)
ADDR r1,r2	2	90	r2 <- (r2) + (r1)
AND m	3/4	40	A <- (A) & (m..m+2)
CLEAR r1	2	B4	r1 <- 0
COMP m	3/4	28	(A) : (m..m+2)
COMPF m	3/4	88	(F) : (m..m+5)
COMPR r1,r2	2	A0	(r1) : (r2)
DIV m	3/4	24	A <-(A)/(m..m+2)
DIVF m	3/4	64	F <-(F)/(m..m+5)
DIVR r1,r2	2	9C	r2 <- (r2) / (r1)
FIX	1	C4	A <- (F) [convert to integer]
FLOAT	1	C0	F <- (A) [ocnver to floating]
HIO	1	F4	Halt I/O channel number(A)
J m	3/4	3C	PC <- m
JEQ m	3/4	30	PC <- m if CC set to =
JGT m	3/4	34	PC <- m if CC set to >
JLT m	3/4	38	PC <- m if CC set to <
JSUB m	3/4	48	L <- (PC); PC <- m
LDA m	3/4	00	A <- (m..m+2)
LDB m	3/4	68	B <- (m..m+2)
LDCH m	3/4	50	A [rightmost byte] <- (m)
LDF m	3/4	70	F <- (m..m+5)
LDL m	3/4	08	L <- (m..m+2)
LDS m	3/4	6C	S <- (m..m+2)
LDT m	3/4	74	T <- (m..m+2)
LDX m	3/4	04	X <- (m..m+2)
LPS	3/4	D0	Load processor status from information beginning at address m
MUL m	3/4	20	A <- (A) * (m..m+2)
MULF m	3/4	60	F <- (F) * (m..m+5)
MULR r1,r2	2	98	r2 <- (r2)*(r1)
NORM	1	C8	F <- (F) [normalized]
OR m	3/4	44	A <- (A) | (m..m+2)
RD m	3/4	D8	A [rightmost byte] <- data from device specified by (m)
RMO r1,r2	2	AC	r2 <- (r1)
RSUB	3/4	4C	PC <- (L)
SHIFTL r1, n	2	A4	r1 <- (r1); left circular shift n bits. [In assembled instruction, r2=n-1]
SHIFTR r1,n	2	A8	r1 <- (r1); right shift n bits, with vacated bit positions set equal to leftmost bit of (r1). [In assembled instruction, r2=n-1]
SIO	1	F0	Start I/O channel number (A); address of chanel program is given

			by (S)
SSK m	3/4	EC	Protection key for address m <- (A)
STA m	3/4	0C	m..m+2 <- (A)
STB m	3/4	78	m..m+2 <- (B)
STCH m	3/4	54	m <- (A) [rightmost byte]
STF m	3/4	80	m..m+5 <- (F)
STI m	3/4	D4	Interval timer value <- (m..m+2)
STL m	3/4	14	m..m+2 <- (L)
STS m	3/4	7C	m..m+2 <- (S)
STSW m	3/4	E8	m..m+2 <- (SW)
STT	3/4	84	m..m+2 <- (T)
STX m	3/4	10	m..m+2 <- X
SUB m	3/4	1C	A <- (A) – (m..m+2)
SUBF m	3/4	5C	F <- (F) – (m..m+5)
SUBR r1, r2	2	94	r2 <- (r2) – (r1)
SVC n	2	B0	Generate SVC interrupt. [In assembled instruction, r1 = n]
TD m	3/4	E0	Test device specified by (m)
TIO	1	F8	Test I/O channel number (A)
TIX m	3/4	2C	X <- (X) + 1; (X): (m..m+2)
TIXR r1	2	B8	X <- (X) + 1; (X): (r1)
WD m	3/4	DC	Device specified by (m) <- (A) [rightmost byte]

Priloga B

```
package assembler;

/**
 * @author ursalevicnik
 */
import java.io.*;
import java.util.*;

public class Assembler {
 //globalne spremenljivke
 static int passNO;
 static int LOCCTR;
 static int LOCCTRLAST;
 static int LOCCTRSTART;
 static int PROGLENGTH;
 static HashMap OPTAB;
 static HashMap SYMTAB;
 static HashMap assemblerDirectives;
 static HashMap ASCII;
 static String PROGNAME;

 static final String pathUkazi = "ukazi.txt";
 static final String pathDirectives = "directives.txt";
 static final String pathSourceCode = "testnakoda.txt";
 static final String pathIntermediate = "intermediate.txt";
 static final String pathObjectProgram = "objectprogram.txt";

public static void main(String[] args) throws IOException{

 initTables();
 pass1();
 pass2();

}//main

static void initTables() throws IOException{
 //samo za prebiranje iz datotek v tabele
 String str;
 StringTokenizer st;

 OPTAB = new HashMap();
 SYMTAB = new HashMap();
 ASCII = new HashMap();
 assemblerDirectives = new HashMap();

 BufferedReader br = new BufferedReader(new FileReader(pathUkazi));

 //preberi ukaze v OPTAB
 while ((str = br.readLine()) != null){
  st = new StringTokenizer(str);
  //mnemonik kot key, hexa vrednost kot vrednost
  OPTAB.put(st.nextToken(), st.nextToken() +" "+ st.nextToken());
```

```
 }//while
 br.close();

 //vpiši registre v SYMTAB
 SYMTAB.put("A", "0");
 SYMTAB.put("X", "1");
 SYMTAB.put("L", "2");
 SYMTAB.put("B", "3");
 SYMTAB.put("S", "4");
 SYMTAB.put("T", "5");
 SYMTAB.put("F", "6");
 SYMTAB.put("PC", "8");
 SYMTAB.put("SW", "9");

//zapiši psevdo ukaze v assemblerDirectives
 br = new BufferedReader(new FileReader(pathDirectives));
 while ((str = br.readLine()) != null){
  st = new StringTokenizer(str);
  assemblerDirectives.put(st.nextToken(), st.nextToken());
 }//while

//kreiraj tabelo ASCII znakov
 br = new BufferedReader(new FileReader("ASCII.txt"));
 while ((str = br.readLine()) != null){
  st = new StringTokenizer(str);
  ASCII.put(st.nextToken(), st.nextToken());
 }//while
}//konec initTables()


static void pass1() throws IOException{
 passNO = 1;

 BufferedReader br1 = new BufferedReader(new FileReader(pathSourceCode));

 String line, current, tmo, toWrite;
 StringTokenizer st;
 boolean endOfCode = false, interesting = true;
 int locctrbefore = 0, currentNumLine = 1;

 //kreiranje vmesne datoteke
 File inter = new File(pathIntermediate);
 BufferedWriter bw1 = new BufferedWriter(new FileWriter(inter));

 //prva vrstica obravnavana posebej
 line = br1.readLine();
 st = new StringTokenizer(line);

 //branje prve vrstice
 if (line.indexOf("START") > 0){
  current = (st.nextToken()).toUpperCase();

  if (current.equals("START")){
   System.out.println("Napaka v vrstici "+currentNumLine+":
     prvi parameter prve vrstice mora biti ime programa!");
     System.exit(1);
  }//if current.equals("START")
  else PROGNAME = current;
```

```
  current = (st.nextToken()).toUpperCase(); //prebere "START"
  if (!current.equals("START")){
   System.out.println("Napaka v vrstici "+currentNumLine+":
    drugi parameter prve vrstice mora biti START!");
   System.exit(1);
  }//if

  if (st.hasMoreTokens()) LOCCTR = hexToDec(st.nextToken());
  else LOCCTR = 0;

  LOCCTRSTART = LOCCTR;
 }//if indexOf("START") > 0
 else{
  System.out.println("Napaka v vrstici"+currentNumLine+": v prvi
   vrstici pricakujem START!");
  System.exit(1);
 }//else

//bere do konca izvorne kode
 while(!endOfCode){
  currentNumLine++;
  line = br1.readLine();
  st = new StringTokenizer(line);

  interesting = true;
  toWrite = "";

  locctrbefore = LOCCTR;

  //če vrstica ni komentar
  if (line.charAt(0) != '.'){
   bw1.write(decToHex(locctrbefore)+" ");

   while (st.hasMoreTokens() && interesting){
    current = st.nextToken();
    if (current.equalsIgnoreCase("END")) endOfCode = true;

    if (!OPTAB.containsKey(current) &&
         !assemblerDirectives.containsKey(current) &&
         (current.charAt(0) != '+')){ //if symbol
     if(SYMTAB.containsKey(current)){
      System.out.println ("Napaka v vrstici "+currentNumLine+":
       ta labela je ze uporabljena!");
      System.exit(1);
     }//if
     else{
      SYMTAB.put(current, LOCCTR);
     }
    }//if symbol

    else if(assemblerDirectives.containsKey(current)){
     toWrite = toWrite.concat(current+" ");
     tmo = st.nextToken();
     if(current.equals("RESB")){
      LOCCTR = LOCCTR + Integer.parseInt(tmo);
     }
```

```
     else if(current.equals("RESW")){
      LOCCTR = LOCCTR + 3*Integer.parseInt(tmo);
     }
     else if(current.equals("WORD")){
      //WORD generates one word integer constant, 1 word = 3 bytes
      LOCCTR = LOCCTR + 3;
     }
     else if(current.equals("BYTE")){
      if (tmo.charAt(0) == 'C')
       //vsak znak je en bajt, -3 zato, da odštejem znake X ali C in ''
       LOCCTR = LOCCTR + (tmo.length() - 3);
      else if (tmo.charAt(0) == 'X')
       LOCCTR = LOCCTR + (int)Math.ceil((double)(tmo.length() - 3)/2);
      else{
       System.out.println("Napaka v vrstici "+currentNumLine+":
         napacno foromiran ukaz BYTE!");
       System.exit(1);
      }
     } //else if BYTE
     toWrite = toWrite.concat(tmo+" ");
     interesting = false;
    }//assemblerDirectives.containsKey(current)

    else if(OPTAB.containsKey(current) || current.charAt(0) == '+'){
     toWrite = toWrite.concat(current+" ");
     //izbriši +, če je prisoten
     int plus = 0;
     if(current.charAt(0) == '+'){
      plus = 1;
      current = current.substring(1,current.length());
     }

     if(!OPTAB.containsKey(current)){
      System.out.println("Napaka v vrstici "+currentNumLine+":
       to ni mnemonik!");
      System.exit(1);
     }

     String value = (String) OPTAB.get(current);
     LOCCTR = LOCCTR +Integer.parseInt(Character.toString(value.charAt(3)))
      + plus;

     if(st.hasMoreTokens()) toWrite = toWrite.concat(st.nextToken() + " ");
     interesting = false;
    } //else if is operand or starts with +
   }
   bw1.write(decToHex(LOCCTR) + " " + toWrite);
   bw1.newLine();
  }//if vrstica ni komentar
 }//while

 LOCCTRLAST = LOCCTR;
 LOCCTR = 0;

 br1.close();
 bw1.close();
}//konec pass1()
```

```
//predvidevam, da so se vsi errorji ujeli že v pass1,
//zato jih tukaj ne obravnavam več
static void pass2() throws IOException{
 passNO = 2;
 LOCCTR = 0;

 String line, buildOC = "", code, opC,  next, first, second, opFromMap;
 String current, currentlocctr, nextlocctr, hex;
 StringTokenizer st;
  String currStartAddr = "";
 boolean endOfCode = false, extended = false, baseValidAddr = false;
 char format;
 int base = 0, comma = 0, indexOfX = 0, tmpX = 0;
 int addBits = 0, loc = 0, addr = 0, disp = 0;
 boolean lojtra = false, afna = false, firstline = true;

 //spremenljivke za oblikovanje vrstice za zapis v datoteko
 int charInLine = 60;
 String lineToWrite = "";
 Sting prevLineStart = fillToLength(decToHex(LOCCTRSTART),6);
 String lastInLine = fillToLength(decToHex(LOCCTRSTART),6);
 boolean needNewLine = false;

 File oc = new File(pathObjectProgram);

 BufferedReader br2 = new BufferedReader(new FileReader(pathIntermediate));
 BufferedWriter bw2 = new BufferedWriter(new FileWriter(oc));

 PROGLENGTH = LOCCTRLAST - LOCCTRSTART;

 //ime programa mora vsebovati 6 znakov - če ne se na koncu doda presledke
 String spaces = "";
 for (int i = 0; i < (6 - PROGNAME.length()); i++){
  spaces = spaces.concat(" ");
 }

 //zapiši header v datoteko z objektno kodo
 bw2.write("H");
 bw2.write(PROGNAME);
 bw2.write(spaces);

 //začetni naslov programa mora vsebovati 6 znakov -
 //če ne se dopolni spredaj z ničlami
 String addressHex = decToHex(LOCCTRSTART);
 if (addressHex.length() < 6){
  for (int i = addressHex.length(); i < 6; i++)
    addressHex = "0".concat(addressHex);
 }
 bw2.write(addressHex);

 //dolžina programa mora vsebovati 6 znakov -
 //če ne, dopolnim spredaj z ničlami
 String lengthHex = decToHex(PROGLENGTH);
 if (lengthHex.length() < 6){
  for (int i = lengthHex.length(); i < 6; i++)
    lengthHex = "0".concat(lengthHex);
 }
 bw2.write(lengthHex);
```

```
//prva vrstica je zapisana, gremo naprej...
while (!endOfCode){
 line = br2.readLine();
 st = new StringTokenizer(line);
 buildOC = "";

 //prvi token je vedno locctr te vrstice,
 //drugi pa locctr po izvajanju tega ukaza
 currentlocctr = st.nextToken();
 nextlocctr = st.nextToken();

 //tretji token je operacijska koda ali ukaz zbirnika
 current = st.nextToken();

 if (current.equals("END")){
  endOfCode = true;
 }
 else if(current.equals("BASE")){
  base = Integer.parseInt((SYMTAB.get(st.nextToken())).toString());
  baseValidAddr = true;
 }
 else if (current.equals("NOBASE")){
  baseValidAddr = false;
 }
 //je mnemonik??
 else if(OPTAB.containsKey(current) || current.charAt(0) == '+'){
  if(current.charAt(0) == '+'){
   code = current.substring(1,current.length());    //odstrani +
   extended = true;
  }
  else{
   code = current;
   extended = false;
  }
  //gledam katerega formata je ukaz, in glede na to razrešujem probleme
  opFromMap = ((OPTAB.get(code)).toString());
  format = opFromMap.charAt(3);

  //format1
  if (format == '1' && !extended){
   //objektna koda vsebuje samo opcode mnemonika...
   buildOC = buildOC.concat(opFromMap.substring(0,2));
  }//konec format 1

  //format2
  else if(format == '2' && !extended){
   buildOC = buildOC.concat(opFromMap.substring(0,2));

   //če naslednji token sploh obstaja
   if (st.hasMoreTokens()) next = st.nextToken();
   else {
    next = "";
    buildOC = ((OPTAB.get(code)).toString()).substring(0,2);
   }

   if ((comma = next.indexOf(",")) > 0) {
    first = next.substring(0,comma);
```

```
   second = next.substring(comma+1,next.length());

   if (SYMTAB.containsKey(first)){
    buildOC = buildOC.concat((SYMTAB.get(first).toString()));
   }
   else buildOC = buildOC.concat(decToHex(Integer.parseInt(first)));

   if (SYMTAB.containsKey(second)){
    buildOC = buildOC.concat((SYMTAB.get(second).toString()));
   }
   else buildOC = buildOC.concat(decToHex(Integer.parseInt(second)));
  }//if

  //če je samo en operand (kot npr. pri ukazu SVC)
  else{
   if (SYMTAB.containsKey(next)){
    buildOC = buildOC.concat((SYMTAB.get(next).toString())+"0");
   }
   else buildOC = buildOC.concat(decToHex(Integer.parseInt(next))+"0");
  }
 }//konec format 2

 //format3
 else if(format == '3' && !extended){
  lojtra = false;
  afna = false;
  //če naslednji token sploh obstaja
  if(st.hasMoreTokens()){
   //code je prvi token - mnemonik, next je drugi token - operand
   indexOfX = 0;
   next = st.nextToken();
   opC = ((OPTAB.get(code)).toString()).substring(0,2);

   if (next.charAt(0) == '#' || next.charAt(0) == '@') {
    if (next.charAt(0) == '#'){
     lojtra = true;
     addBits = hexToDec(opC) + 1;
    }
    else{
     afna = true;
     addBits = hexToDec(opC) + 2;
    }
    next = next.substring(1,next.length());  // izre≈æem # ali @
   }//if
   else{
    addBits = hexToDec(opC) + 3;
    if ((indexOfX = next.indexOf(",X")) > 0) {
     next = next.substring(0,indexOfX);
    }
   }//else


   //dodam vrednost mnemonika v buildOC
   buildOC = buildOC.concat(decToHex(addBits));

   if (SYMTAB.containsKey(next)){
    loc = hexToDec(nextlocctr);
    addr = Integer.parseInt((SYMTAB.get(next).toString()));
```

```
 disp =  addr - loc;

 if ((disp >= -2048) && (disp <= 2047)){
  //nastavi bit p
  if (disp >= 0) disp = disp | 8192;  //nastavi le en bit na 1
  else disp = disp & 12287;
 }
 else if(baseValidAddr){
  disp = addr - base;
  if (disp >=0 && disp <= 4095){
   //nastavi bit b
   disp = disp | 16384;
  }
 }//else if
 else {
  System.out.println("Predolgi naslov!!");
  System.exit(1);
 }

 if(disp < 0) hex = (Integer.toBinaryString(disp)).substring(27,31);
 else hex = decToHex(disp);

 if (hex.length() < 4){
  for (int i = hex.length(); i < 4; i++){
   buildOC = buildOC.concat("0");
  }
 }
  buildOC = buildOC.concat(hex);
}//if(SYMTAB.containsKey(next))

else{
 addr = Integer.parseInt(next);

 if (addr > 4095){
  System.out.println("Predolgi naslov!!");
  System.exit(1);
 }

 hex = decToHex(addr);
 if (hex.length() < 4){
  for (int i = hex.length(); i < 4; i++){
   buildOC = buildOC.concat("0");
  }
 }
 buildOC = buildOC.concat(decToHex(addr));
}//else

 if (indexOfX > 0){
  tmpX = hexToDec(buildOC);
  tmpX = tmpX | 32768;
  buildOC = decToHex(tmpX);
 }
} // konec if st.hasMoreTokens()
else
{
 next = "";
 buildOC = ((OPTAB.get(code)).toString()).substring(0,2);
 int xyz = hexToDec(buildOC);
```

```
  xyz = xyz + 3;
  buildOC = decToHex(xyz) + "0000";
 }
}//konec format 3

//format4
else if(extended){
 //naslednji token sploh obstaja?
 if(st.hasMoreTokens()) {
  lojtra = false;
  afna = false;

  next = st.nextToken();
  opC = ((OPTAB.get(code)).toString()).substring(0,2);

  if (next.charAt(0) == '#' || next.charAt(0) == '@') {
   if (next.charAt(0) == '#'){
    lojtra = true;
    addBits = hexToDec(opC) + 1;
   }
   else{
    afna = true;
    addBits = hexToDec(opC) + 2;
   }
   next = next.substring(1,next.length());  //izrežem # ali @
  }
  else{
   addBits = hexToDec(opC) + 3;
   if ((indexOfX = next.indexOf(",X")) > 0) {
    next = next.substring(0,indexOfX);
   }
  }

  buildOC = buildOC.concat(decToHex(addBits));
  buildOC = buildOC.concat("1");

  if (SYMTAB.containsKey(next)){
   addr = Integer.parseInt((SYMTAB.get(next).toString()));

   hex = decToHex(addr);
   if (hex.length() < 5){
    for (int i = hex.length(); i < 5; i++){
     buildOC = buildOC.concat("0");
    }
   }
   buildOC = buildOC.concat(hex);
  }
  else {
   addr = Integer.parseInt(next);

   hex = decToHex(addr);
   if (hex.length() < 5){
    for (int i = hex.length(); i < 5; i++){
     buildOC = buildOC.concat("0");
    }
   }
   buildOC = buildOC.concat(hex);
  }
```

```
    } //konec if st.hasMoreTokens()

    else {
     next = "";
     buildOC = ((OPTAB.get(code)).toString()).substring(0,2) + "000000";
    }
   }//konec format 4
  }//konec if mnemonik

  //je psevdoukaz?
  else if(assemblerDirectives.containsKey(current)){
   if(current.equals("BYTE")){
    current = st.nextToken();
    if(current.charAt(0) == 'X'){
     int l = current.length();
     buildOC = current.substring(2,l-1); //končni string
    }
    else if(current.charAt(0) == 'C'){
     int l = current.length();
     String buildascii = "";
     String c;
     for (int i = 2; i < l-1; i++){
      c = Character.toString(current.charAt(i));
      String xyz = (ASCII.get(c)).toString();
      buildascii = buildascii.concat(xyz);
     }
     buildOC = buildascii; //končni string
    }
   }//if BYTE
   else if(current.equals("WORD")){
    addr = Integer.parseInt(st.nextToken());
    hex = decToHex(addr);

    if (hex.length() < 6) hex = fillToLength(hex,6);

    buildOC = buildOC.concat(hex);
   }//else if WORD
  }//else if psevdoukaz


  if((hexToDec(nextlocctr) - hexToDec(prevLineStart)) > 255){
   needNewLine = true;
  }




//zapis v datoteko z objektno kodo:
  if ((charInLine+buildOC.length() > 60) || endOfCode || needNewLine){
   int xxx = hexToDec(lastInLine) - hexToDec(prevLineStart);

   if (!firstline){
    bw2.write(decToHex(xxx) + "*" + lineToWrite);
    if (!endOfCode){
     bw2.newLine();
```

```
     if (!needNewLine) bw2.write("T"+fillToLength(lastInLine,6));
     else bw2.write("T"+fillToLength(nextlocctr,6));
    }
    lineToWrite = buildOC;
    if (needNewLine) prevLineStart = nextlocctr;
    else prevLineStart = lastInLine;

    charInLine = buildOC.length();
    needNewLine = false;
    firstline = false;
   }
   else if (charInLine+buildOC.length() <= 60){
   lineToWrite = lineToWrite.concat(buildOC);
   charInLine = charInLine + buildOC.length();
    if (buildOC.length() > 0) lastInLine = nextlocctr;
   }//else if

  }

 bw2.newLine();
 bw2.write("E"+fillToLength(decToHex(LOCCTRSTART),6));
 br2.close();
 bw2.close();
}//konec pass2()

static int hexToDec(String s){
 int i;
 char c;
 int decimal = 0;

 for (i = s.length()-1; i >= 0; i--){
 c = s.charAt(i);

 switch(c){
  case '1': decimal=(int)(decimal + 1*Math.pow(16, s.length()-(i+1)));
      break;
  case '2': decimal=(int)(decimal + 2*Math.pow(16, s.length()-(i+1)));
      break;
  case '3': decimal=(int)(decimal + 3*Math.pow(16, s.length()-(i+1)));
      break;
  case '4': decimal=(int)(decimal + 4*Math.pow(16, s.length()-(i+1)));
      break;
  case '5': decimal=(int)(decimal + 5*Math.pow(16, s.length()-(i+1)));
      break;
  case '6': decimal=(int)(decimal + 6*Math.pow(16, s.length()-(i+1)));
      break;
  case '7': decimal=(int)(decimal + 7*Math.pow(16, s.length()-(i+1)));
      break;
  case '8': decimal=(int)(decimal + 8*Math.pow(16, s.length()-(i+1)));
      break;
  case '9': decimal=(int)(decimal + 9*Math.pow(16, s.length()-(i+1)));
      break;
  case 'A': decimal=(int)(decimal + 10*Math.pow(16, s.length()-(i+1)));
      break;
  case 'B': decimal=(int)(decimal + 11*Math.pow(16, s.length()-(i+1)));
      break;
  case 'C': decimal=(int)(decimal + 12*Math.pow(16, s.length()-(i+1)));
      break;
```

```
  case 'D': decimal=(int)(decimal + 13*Math.pow(16, s.length()-(i+1)));
       break;
  case 'E': decimal=(int)(decimal + 14*Math.pow(16, s.length()-(i+1)));
      break;
  case 'F': decimal=(int)(decimal + 15*Math.pow(16, s.length()-(i+1)));
      break;
 }//switch
 //for
 return decimal;
}//hexToDec

static String decToHex(int dec){
 if(dec == 0) return "00";

 String hexadecimal = "";
 int current = 0;

 while (dec > 0){
  current = dec % 16;
  if (current == 10) hexadecimal = "A".concat(hexadecimal);
  else if(current == 11) hexadecimal = "B".concat(hexadecimal);
  else if(current == 12) hexadecimal = "C".concat(hexadecimal);
  else if(current == 13) hexadecimal = "D".concat(hexadecimal);
  else if(current == 14) hexadecimal = "E".concat(hexadecimal);
  else if(current == 15) hexadecimal = "F".concat(hexadecimal);
  else hexadecimal = Integer.toString(current).concat(hexadecimal);
  dec = dec / 16;
 }

 if (hexadecimal.length()%2 != 0)
  hexadecimal = "0".concat(hexadecimal);

 return hexadecimal;
}//konec decToHex

static String fillToLength(String smallHex, int toNum){
 for (int i = smallHex.length(); i < toNum; i++){
  smallHex = "0".concat(smallHex);
 }
 return(smallHex);
}


}//konec class assembler
```