Универзитет у Крагујевцу
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА
ЧАЧАК

ДИПЛОМСКИ РАД

WEB БАЗИРАНИ АСЕМБЛЕР ПРОЦЕСОРА TFCOMIN

Ментор

проф. др Синиша Ранђић

Студент

Александра Јовановић 32/2012

У Чачку,
2016. године

САДРЖАЈ

1. УВОД

Развојем технологије и људског друштва уопште, свакодневни живот постаје све лакши. Са појавом првих рачунара дошло је до наглог развоја и осталих технологија и за релативно кратко време рачунари су нашли примену у практично свим аспектима друштва, како у професионалној, тако касније и у личној, кућној примени. Може се рећи да је појава рачунара у великој мери обликовала данашње окружење и друштво у коме живимо.

Иако су данашњи рачунари веома једноставни за коришћење, тј. имају *user-friendly* кориснички интерфејс и како у суштини свака новија верзија оперативних система пружа корисницима све лакши и бржи рад на рачунару, рачунар сам по себи постаје све комплекснија машина. Хардвер рачунара постаје све комплекснији, а софтвер све сложенији и захтевнији. Да би рачунар уопште могао да ради, он мора добити све информације у бинарном облику, кодованом конкретно за његову архитектуру. Да би се то постигло, неопходно је извршити поступак који се назива асемблирање, у оквиру кога се програм писан на симболичком машинском језику у инструкције машинског језика, тј. „језик" карактеристичан за дату рачунарску архитектуру. Програм који обавља то превођење се назива асемблер.

Предмет овог дипломског рада је поступак пројектовања и реализације асемблера за архитектуру процесора TFComin. Ова архитектура 16 – битног процесора развијена је у оквиру студентских пројеката из предмета Архитектура рачунара на Факултету техничких наука у Чачку. У раду ће бити размотрени детаљи процеса пројектовања и реализације асемблера за дату архитектуру. Циљ израде овог рада је добијање асемблера који ће омогућити превођење програма писаних на симболичком машинском језику у одговарајући низ машинских инструкција процесора TFComin. Добијени машински код може да се користи као улазни програм за симулатор архитектуре поменутог процесора, који је реализован у оквиру магистарског рада „Прилог симулацији рачунарских архитектура", који је такође реализован на Факултету техничких наука. С друге стране, студенти рачунарства све чешће стичу знања која их удаљавају од рачунарске основе, а то су архитектура и организација рачунара и нису у прилици да упознају принципе на којима се развијају алати за развој програма, пре

свега асемблери и језички преводиоци. Због тога је приказ развоја асемблера, за једну релативно једноставну рачунарску архитектуру, као што је TFComin може да буде згодно наставно средство. Поготову што се при реализацији асемблера користе многи важни алгоритми, који се користе у рачунарској техници. У протеклом периоду на Факултету техничких наука већ су развијани асемблери за сличне архитектуре, али су при томе коришћени стандардни програмски језици, као што су VisualBasic и C#. Овог пута покушано је коришћење језика који омогућава да се рад асемблера може визуелизовати и приказати путем WEB. За ту намену изабран је JAVA програмски језик. При реализацији асемблера коришћењем овог програмског језика циљ је био и стицање искустава на плану процесирања симболичких информација и рада са системом датотека на WEB серверу.

С обзиром да асемблери спадају у системски део софтвера рачунара у глави 2 дат је кратак приказ намене, карактеристика и захтева према којима се пројектује овај сегмент програмског система рачунара. Посебно је указано на значај преводилаца програмских језика за развој комплетне софтверске подршке једног рачунара. Такође је дат приказ и принципа развоја најважнијих елемената системског софтвера посебно у случају када постоји новодефинисана архитектура. То пре свега подразумева:

- Развој асемблера;
- Развој преводилаца за изабране програмске језике;
- Развој новог или „портирање“ неког од постојећих оперативних система.

Посебна пажња је поклоњена принципима и захтевима развоја асемблерског језика и самог асемблера за новопројектовани процесор.

У глави 3 је приказана архитектура процесора TFComin. Наведен је и објашњен скуп машинских инструкција овог процесора уз опис понашања процесора при извршавању сваке инструкције укључујући и начин формирања статусне информације односно тзв. кода услова за извршавање инструкција скока.

У глави 4 указано је на концепт реализације и структуру развијаног асемблера. Дат је приказ операција и одговарајућих програмских модула из којих треба да се састоји асемблер, а који треба да омогуће превођење симболичког програма у машински. У складу са тим описане су функције датих модула.

У глави 5 дат је опис реализације развијаног асемблера. У склопу тога описани су алгоритми најважнијих операција које у асемблеру омогућавају задатак превођења. Посебна пажња је посвећена алгоритмима за формирање HASH табела израчунавањем одговарајућег hash броја, као и питању конверзије бројних система у бинарни код. На крају је на једноставном примеру приказано функционисање реализованог асемблера.

2. СИСТЕМСКО ПРОГРАМИРАЊЕ

Рачунарски систем без свог програмског подсистема практично је неупотребљив. Основна карактеристика рачунарских система је интегрална повезаност његовог хардвера и софтвера. Рачунар разуме инструкције на *машинском језику*, тј. низове битова. Са друге стране, људима тај облик није много подесан како за само разумевање значења ових низова, а посебно ако је тај принцип потребно применити за реализацију рачунарског програма програма. Људи много лакше користе *симболички* приступ у својој комуникацији са окружењем. Па се то може применити и на комуникацију са рачунаром, која се у конкретном случају остварује на нивоу програмирања рада рачунара у оквиру обраде података.

Системски софтвер је општи назив за скуп рачунарских програма намењених стварању општих услова да рачунар може да извршава своје основне функција. У склопу тога системски софтвер између осталог омогућава управљање хардвером рачунара. Такође системски софтвер извршава задатке као што су пренос података са меморије на диск, исписивања текста на екрану и тако даље. У принципу системски софтвер чини више одвојених програма, који координисано обављају своје функције и у њих спадају оперативни системи, драјвери улазно/излазних уређаја, алати за развој програма као што су: едитори текста, асемблери, преводиоци, линкери, дибагери, итд. У принципу програми који чине системски софтвер разликују се од програма који реализују специфичну обраду података према захтевима корисника управо по својој општости у односу на потребе корисника. Јер они реализују функције које су идентификоване као заједничке на нивоу највећег броја корисника једног рачунарског система. Имајући то у виду поједини делови системског софтвера се могу имплементирати у фиксној меморији (ROM). У том случају за тај део системског софтвера се користи енглески термин firmware.

Значај системског софтвера се огледа и у чињеници да он обезбеђује интегративну функцију тј. омогућава деловима рачунара да функционишу као целина. Без оперативног система, као једног од најважнијих делова системског софтвера рачунар не би могао да функционише као целина. За разлику од системског софтвера, софтвер који омогућава да се извршавају послови као што је креирање текстуалних

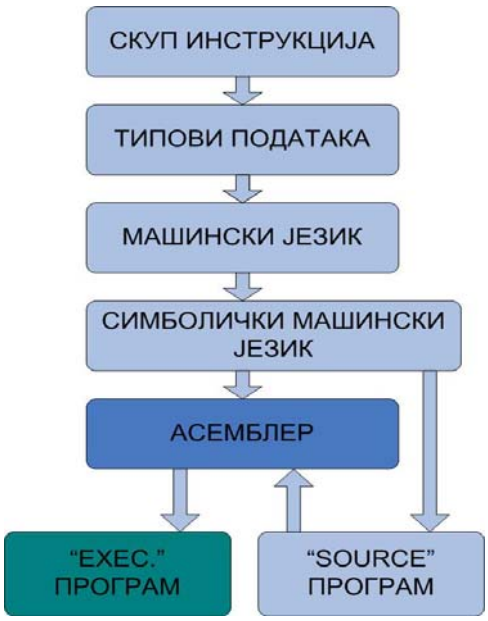
докумената, играње игрица, слушање музике, „пловљење“ Интернетом, зове се апликативни софтвер, јер је окренут одређеној примени и одражава сву њену специфичност. Генерално, програми су логичке целине које омогућују крајњем кориснику да обавља одређене, производне задатке, као што су обрада текста и обрада слика. У склопу тога јасно се може уопштити да системски софтвер извршава оне задатке задатке који су заједнички за поменуте активности, као што су размена података између оперативне меморије и диска, или приказивање текста на екрану уређаја да би корисник могао да има увид у одвијање тока обраде података.

Када се развија потпуно нова архитектура рачунара, неопходно је развити и одговарајуће системске програме за тај рачунар. Код развоја нове архитектуре у неким случајевима може се применити принцип тзв. „портирања” елемената већ постојећих системских програма. Нпр., „портирање” неког већ постојећег оперативног система на нову архитектуру. Међутим, како нова архитектура са собом доноси нови скуп инструкција, структура и типова података које се њоме обрађују јасно је да се морају написати нови програми, који треба да обезбеде функционалност „портираних” системских програма. Иако ће и даље бити коришћен неки од популарних програмских језика (C, C#, ...) мора се написати нови преводилац, који ће моћи да се извршава на новој архитектури, али који ће моћи и да врши превођење програма написаног на неком од програмских језика високог нивоа на ниво машинских инструкција новог рачунара.

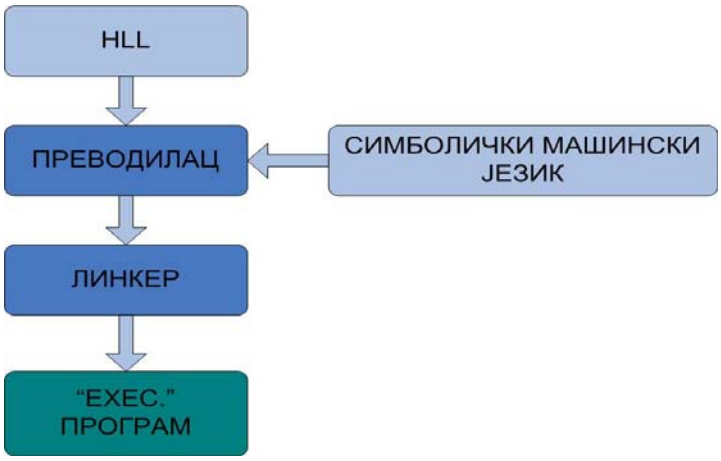
На слици 2.1 су приказани неки од главних интеракционих процеса елемената архитектуре рачунара (а) и системских програма намењених развоју програма (б) односно зависности извршног програма од оперативног система (ц). Зеленом бојом су означени системски програми. Услов да би нова архитектура уопште радила је развијање ових системских програма.

Ови системски програми су:

1. Оперативни систем
2. Асемблер
3. Компајлер
4. Линкер
5. Пуњач (Loader)
6. Програм за отклањање грешака (Debugger)



(а) Процес добијања извршне верзије програма на бази изворног симболичког кода



(б) Ланац креирања линкера и преводиоца уз помоћ неког вишег програмског језика

(ц) Зависност извршног програма од оперативниог система

Слика 2.1 Место системских програма у развоју програмске подршке

2.1 Асемблер

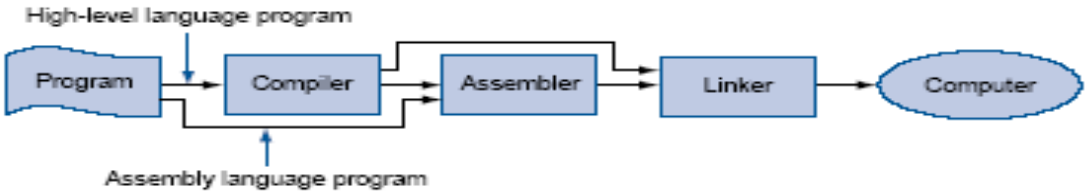
Асемблерски језик је симболичка представа бинарно кодираних инструкција машинског језика. Због своје симболичке базе асемблерски језици су много читљивији од машинског управо због коришћења симбола уместо битова. Симболи одговарају одређеним комбинацијама битова као што су кодови операција или спецификације регистара. Уз то, асемблерски језик омогућава коришћење *симболичких адреса* које представљају симболичка имена појединих меморијских локација било да оне садрже програмске инструкције или податке.

Рачунар, међутим, и даље разуме само машински језик – комбинације битова. Због тога је потребно средство које ће програм написан на асемблерском језику да преведе у програм који се састоји из бинарно кодованих машинских инструкција. Ова врста преводиоца назива се *асемблер*.

Асемблер је програмски језик који машински језик специфичне процесорске архитектуре представља у људима читљивом облику. Из тога произилази да свака процесорска архитектура поседује свој асемблер. Постојање асемблера омогућава писање програма на погоднији начин за програмера. Симболичка имена операција и локација где су смештени подаци су један вид ове презентације. Програмска средства, као што су *макрои*, који представљају груписање често коришћених машинских инструкција у посебне целине, такође побољшавају јасноћу програма и олакшавају програмирање.

Асемблер чита поједине *изворне датотеке* и производи *објектне датотеке* које садрже машинске инструкције и информације које помажу комбиновању неколико објектних фајлова у програм. Већина програма састоји се од неколико датотека или *модула* које се пишу и преводе независно. Модул обично садржи *референце* на потпрограме и податке дефинисане у другим модулима и библиотекама. Програмски код у модулу се не може извршити када постоје *неразрешене референце* на лабелу у другим објектним датотекама или модулима.

Још једно системско средство, *линкер*, комбинује колекцију објектних и библиотечких датотека у *извршну датотеку* чији садржај програм који се може извршити на рачунару. Уз познавање начина бинарног кодирања инструкција, могао би да се добије симболички код. Међутим, много је јасније када се уведу симболичка имена за регистре и меморијске локације. Имена која нпр., почињу тачком (.data, .globl, и др.) уобичајено представљају *асемблерске директиве* које казују асемблеру како да преведе програм, али саме се не преводе у машинске инструкције. Имена праћена двотачком представљају *лабеле* или имена меморијских локација. Поступак добијања извршног програма приказан је на слици 2.2.



Слика 2.2 Поступак добијања извршног програма

У општем случају асемблерски језик има двојаку улогу. Прва улога је да представља излазни језик језичког преводиоца (компајлера). Друга улога је да представља језик на коме се пишу програми. Ова улога је некада била доминантна. Данас то није случај, али се асемблерски језик користи тамо где је брзина рада или

величина програма критична за искоришћење хардверских ресурса, а при томе не постоји одговарајућа алтернатива на нивоу виших програмских језика.

Програм написан у асемблерском језику се често означава и као асемблерски код. Тај код се путем специјалног компајлера претвара у машински код који процесор може да директно изврши. Обрнути процес претварања машинског у асемблер код назива се *дисасемблирање*. При том процесу је рестаурација свих пређашњих делова асемблерског кода немогуће, пошто се они (нпр. коментари) при превођењу неповратно изгубљени. Танође, може доћи до погрешне интерпретације делова машинског кода, операција као података или обратно. Због тога је дисасемблирани код може да буде веома тежак за разумевање.

2.2 Машински и асемблерски језици

Машински језик се састоји од нумеричког кода за операције који одређени рачунар може директно извршити. Тај код представља низ бинарних вредности 0 и 1, или бинарни код на нивоу бајта, који се често претвара у хексадецимални код (представљање бројева у коду са основом 16), ради лакше читљивости и модификације. Инструкције машинских језика обично користе један број бајтова за представљање операција, сабирање на пример, а други за представљање операнда (података над којима се извршава операција) и/или локације где се налази следећа инструкција у програму. Машински језик је тежак за читање и писање, пошто не личи на конвенционално математичко представљање, нити на природни језик, а његов код варира од рачунара до рачунара.

Асемблерски језик представља ниво изнад машинског језика. Користи кратки симболички код за инструкције и омогућава програмеру да уноси имена за блокове меморије који садржи податке. Пројектован је да омогући лако превођење у машински језик. Иако се блокови података у асемблерском језику позивају преко имена, а не преко адресе у меморији, ипак не постоји могућност софистицираног организовања сложених информација. Као и машински језик, асемблерски језик захтева од програмера детаљно познавање рачунарске архитектуре. Користан је када су ти детаљи важни, односно приликом програмирања рачунара за интеракцију са улазним и излазним уређајима, као што су штампачи, скенери, уређаји за чување података и информација (оптички и чврсти дискови), итд.

2.2.1 Разлози коришћења асемблерских језика

Програми написани у асемблерском језику се одликују могућношћу слања директних команди процесору као и потпуно коришћењу свих елемената рачунарске

архитектуре. Пошто ти програми раде практично на нивоу машинског кода, и у себи не имплементирају помоћне конструкције, генерализовање кода, као и друге небитне ствари за ниво хардвера, много су мањи и бржи од програма написаних у неком конвенционалном програмском језику високог нивоа.

Брзина и величина програма могу да буду од критичне важности, као у *уграђеним рачунарима* (embedded computer), какви су рачунари базирани нпр. на микроконтролерима. У програмима где су поједни делови критични у погледу времена извршења користи се хибридни приступ. Преводици у принципу могу да генеришу бољи машински код од просечног програмера, али програмери боље разумеју алгоритме и понашање програма и, за разлику од преводилаца, могу да имају у виду више процедура када пишу код за једну од њих. Додуше, у будућности ће програмери мање бити у стању да овај посао обављају боље од преводилаца. Још једна битна предност асемблерског језика је у могућности да се користе специјализоване инструкције. Најзад, тамо где нема расположивог вишег програмског језика остаје нам само да користимо асемблерски језик.

2.2.2 Недостаци асемблерских језика

Иако су представљали значајан напредак у добијању алата за програмирање рачунара, асемблерски или симболички машински језици у поређењу са програмским језицима високог нивоа имају одређене недостатке, као што су:

- Асемблерски језици су машински оријентисани програмски језици;
- Изворни код је много дужи него код виших програмских језика што смањује продуктивност у процесу програмирања;
- Дужи програми су тежи за читање и разумевање и могу да имају много више грешака него одговарајући програми написани на неком програмском језику високог нивоа.

Неке од главних мана програма написаних асемблерским језицима је и даље присутна лоша читљивост програма, што је посебно изражено при великим пројектима као и сложеност самог програмског кода, као и практична немогућност директног конвертовања програмског кода на другу процесорску архитектуру. Због тих мана се асемблерски језици се данас користе само у временски критичним програмима код којих је битна брзина извршавања програма, као што су хардверски драјвери (енгл. driver), нпр. за графичке адаптере или у просторно критичним програмима, код којих је битна величина извршног програма у интегрисаним системима (енгл. Embedded

Systems). Примери таквих система су програми који се пишу за микроконтролерски базиране рачунарске системе.

2.3 Програмски преводиоци

Језички преводилац (програмски преводилац, компајлер, енг. compiler) представља рачунарски програм који чита изворни програм написан у неком програмском језику и преводи га у у циљани (најчешће машински) језик. Софтвер писан за прве рачунаре је дуго времена писан у асемблерском језику (нижи симболички оријентисан програмски језик намењен програмирању рачунара и прилагођен архитектури централног процесора и треба га разликовати од асемблера – асемблер је системски програм који преводи програм написан на асемблерском језику у програм на машинском језику датог рачунара). Виши програмски језици су створени тек када је могућност коришћења софтвера на различитим процесорима постала већа од трошкова писања преводилаца за те програмске језике. Такође, ограничене меморијске могућности првих рачунара су биле техничка препрека за примену преводилаца.

Код првих преводилаца њихова структура је зависилиа од сложености процесирања коју су требали да изврше, претходног искуства искуства аутора и расположивих ресурса. Преводилац који би писао један аутор, за неки релативно једноставан језик по правилу би био реализован као јединствени софтверски модул. Сложенији језици и захтеви за добијањем квалитетног машинског превода утицали су да се преводиоци пројектују кроз неколико релативно самосталних фаза. За сваку од фаза у пројектовању може бити задужен по један аутор, односно, посао подељен између више сарадника. Рад по фазама омогућава каснију лакшу замену постојећег преводиоца побољшаним верзијама или уметање нових карактеристика.

Формална дефиниција преводиова се заснива на дефинисању односа између три језика:

- изворног језика;
- циљног језик и
- језика изградње.

те се преводилац аналитички може приказати на следећи начин:

$$MJ_{L_g}^{L_i \rightarrow L_c}$$

где је MJ преводилац, Li – изворни језик, Lc – циљни језик, а Lg – језик изградње.

Пошто је преводилац и сам по себи програм, то се као језик изградње може третирати програмски језик који је коришћен за писање самог преводиоца. Осим три

споменута језика, развијају се и посебни метајезици (језици за дефинисање језика) који омогућују једноставан запис правила коришћења програмских језика.

Две основне фазе рада преводиоца су:

- фаза анализе изворног програма;
- фаза синтезе циљног програма

Током фазе анализе изворног програма реализују се се два процеса превођења, један током лексичке анализе, други током синтаксне анализе и семантичке анализе.

Лексичка анализа групише знакове изворног програма у основне елементе језика, које се зову *лексичке јединице* (енгл. token) или *лексеми*. Лексеми су нпр. *променљиве*, *кључне речи*, *константе*, *оператори* и *правописни знакови*. Лексичка јединица се формално задаје као низ знакова. Лексичка правила дефинишу скуп свих правилно написаних лексичких јединица (низова) заданог програмског језика. Дозвољено је да скуп правилно написаних лексичких јединица може да буде бесконачан и у формалном смислу тај скуп дефинише језик лексичких јединки. Формални аутомат је основа лексичког анализатора. Током лексичке анализе, сваки се лексем замени јединственим знаком.

Лексички анализатор такође гради структуру података звану таблица знакова (или таблица симбола) у коју се смештају сви остали подаци важни за променљиве и константе. Током синтаксне анализе изводи се процес прихватања низова јединствених знакова лексичких јединица, а током семантичке анализе изводи се процес генерисања међукода. Формални аутомат је такође окосница и синтаксног анализатора.

Семантички анализатор покреће процес генерисања вишег нивоа међукода. Семантичка правила су интерпретацијска правила која повезују извршавање програма с понашањем рачунара. Семантика језика одређује скуп дозвољених значења. Током процеса генерисања вишег нивоа међукода обично се израчунавају константне вредности и поједностављује се структура наредбе. Током фазе синтезе циљног програма изводе се три процеса превођења:

- превођење вишег нивоа међукода у средњи међукод,
- превођење средњег међукода у нижи међукод и
- превођење нижег међукода у циљни програм.

Током процеса превођења вишег међукода у средњи међукод обавља се конвертовање сложених структура података, као што су низови података, и сложених контролних програмских конструкција у низ наредби које користе искључиво променљиве и једноставне инструкције гранања.

2.4 Оперативни системи

У рачунарству, оперативни систем (ОС) је скуп програма и рутина одговоран за контролу и управљање уређајима и рачунарским компонентама као и за обављање основних системских активности. Оперативни систем обједињује у целину разнородне делове рачунара и сакрива детаље функционисања делова рачунара од крајњег корисника. Оперативни систем ствара за корисника радно окружење које рукује процесима и датотекама, уместо битовима, бајтовима и блоковима.

2.4.1 Структура оперативних система

Већина оперативних система долази са интегрисаним програмским модулом који обезбеђује кориснички интерфејс за руковање оперативним системом од стране корисника, који укључује програме као што су:

- интерпретер командне линије и
- графички оријентисани кориснички интерфејс.

Истовремено, оперативни систем омогућава покретање других, корисничких програма, али и програма који спадају у класу тзв. услужних програма, као што су едитори, преводиоци или Интернет претраживачи.

У основи оперативни систем је суштински састављен из три скупа компоненти:

- Корисничког интерфејса, који, као што је напред речено може бити графички оријентисан или реализован на принципу интерпретера командне линије, који је познат и под називном „шкољка“ (shell);
- Системске рутине ниског нивоа;
- Језгро – kernel које као што и сам назив сугерише представља „срце“ оперативног система

Као што назив указује, „шкољка“ је спољашњи програмски омотач језгра и првенствено је намењено комуникацији са корисником, док језгро непосредно комуницира са уређајима. Код неких оперативних система, као што је UNIX, „шкољка“ и језгро су различити и самостални ентитети, што омогућује произвољне комбинације и лаку замену „шкољке“. Други оперативни системи само формално приказују постојање различитих компоненти док су у суштини компактни програмски системи.

Током пројектовања различитих оперативних система, дошло је до диференцирања концепта језгра на следеће приступе:

- Оперативни системи са монолитним језгром;
- Оперативни системи са микројезгром;
- Оперативни системи са егзојезгром.

Већина најшире коришћених оперативних система, као што су UNIX, Linux и Windows има језгра монолитног типа. Неки новији оперативни системи имају микројезгро, а такви су Apple MacOS X, AmigaOS, QNX и BeOS. Међу истраживачима концепт на бази микројезгра је веома популаран, и примењен је код система као што је нпр., Hurd/GNU. Оба система имају својих предности и успешно се користе на многим рачунарима. На системима посебне намене који подразумевају уградњу система и софтвера на ниво хардвера примењује се концепт егзојезгра.

Први рачунари нису имали оперативни систем. Оператор је био тај који је ручно уносио и покретао програме. Када су развијени програми за учитавање и покретање других програма логично је било да такви програми добију назив у складу са послом који обављају.

Први рачунаром, који су пројектовани према von Neumann – овом концепту архитектуре имали су и програме и податке у истој меморији. Када су се са следећом генерацијом рачунара појавили и први спољашњи уређаји – читачи папирне траке и бушач картица – створили су се услови за прелазак на следећи ниво аутоматизације рада рачунара. Уобичајени скуп картица на којима су се налазили кодови за улазне и излазне операције су биле основа за будући развој оперативних система.

Под појмом „оперативни систем“ данас се, од корисничке и стручне јавности, најчешће подразумева сав софтвер потребан кориснику за управљање системом и покретање свих програма који могу радити на том рачунарском систему. По општеприхваћеним нормама то подразумева не само најниже слојеве језгра (kernel) који непосредно управљају уређајима него и библиотеке неопходне корисничким програмима као и основне програме за рад са датотекама и конфигурисање система.

Граница између оперативног система и корисничких програма није прецизно одређена и представља често предмет расправе. Нпр., једно од кључних питања у антимонополском судском случају Сједињене Америчке Државе против Microsoft – а је било да ли је Microsoft – ов претраживач Интернета Internet Explorer део оперативног система Windows или је део скупа корисничких програма. Други пример је неслагање око означавања GNU/Linux, јер у основи овог система стоји језгро Linux – а, али многи управо цео оперативни систем зову Linux.

Најнижи ниво сваког оперативног система је језгро, први слој овог типа софтвера који се учитава у рачунарску меморију при покретању рачунара. Као први софтверски слој, он обезбеђује свом осталом софтверу, који се потом учита у оперативну меморију заједничко коришћење услуга језгра. Основне услуге које пружа ово заједничко језгро су приступ дисковима, управљање меморијом, управљање процесима и пословима и

приступ осталим рачунарским уређајима. Као и на нивоу целог оперативног система и овде постоји питање шта тачно треба да чини језгро. Постоје ставови стручњака, којима се подржавају већ поменути концепти „микрокернела“ или „монолитног кернел“ или неке друге концепте. Чак се постављају питања као на пример – треба ли систем за управљање датотекама (file system) да буде део кернела?

2.5 Линкер

Под *линкером* или линк – едитором се подразумева програм који узима један или више објектних кодова, које је генерисао преводилац и повезује их у један извршни програм. У IBM – Mainframe окружењима као што је нпр., OS/360 овај програм је познат као linkage editor. На UNIX платформама термин „пуњач“ (енг. loader) се често користи као синоним за линкер. Обично су то два независна програма, али у неким оперативним системима исти програм обавља оба посла повезивање објектних кодова и учитавање извршног програма у меморију.

Рачунарски програми се обично састоји од неколико модула. Сви ти модули не морају бити садржани у једној објектној датотеци. По правилу, објектна датотека може да садржи три врсте симбола:

- дефинисане симболе, који омогућавају да други модули позивају дати модул;
- недефинисане симболе, који позивају друге модуле где су ти симболи дефинисани и
- локалне симболе, који се интерно користе у оквиру објектне датотеке да би се олакшало повезивање.

Када се програм састоји од више објекатних датотека, линкер комбинује ове датотеке у јединствени извршни програм, превођењем симбола редом. Линкери могу да узимају предмете из колекције зване библиотеке. Неки линкери не обухватају целу библиотеку на излазу. Они само обухватају њене симболе који се позивају из других објектних датотека или библиотека. Библиотеке постоје за различите сврхе, и једна или више системских библиотека су обично повезане.

Линкер се брине такође о уређењу објеката у адресном простору програма. Ово може да укључује релокацију кода, који се реферише на одређену базну адресу у односу на другу базну адресу. Пошто преводилац ретко када зна где се објекти налазе, по правилу се претпоставља фиксна базна локација (најчешће нулта адреса). Релокација машинског кода може да подразумева поновно повезивање апсолутних скокова, читања и памћења (Load и Store).

Излаз линкера понекад захтева још један релокацијски пролаз када се коначно учитава у меморију извршни програм (непосредно пре извршења). Овај пролаз се обично обавља у хардверској виртуелној меморији – сваки програм се ставља у свој адресни простор, тако да нема колизије, чак и ако се сви програми учитају у односу на исту базну адресу.

2.5.1 Динамичко повезивање

Многа окружења оперативних система омогућавају динамичко повезивање, тј. одлагање решавања проблема недефинисаних симбола док се не покрене програм. То значи да извршни код још увек има недефинисаних симбола, као и листу објеката или библиотека које обезбеђују њихову дефиниције. Учитавање програма ће учитати и ове објектене датотеке/библиотеке, и извршити коначно повезивање.

Овај приступ има две предности:

- Често коришћене библиотеке (рецимо стандардне системске библиотеке) могу да буду чуване само једном месту, а не да се дуплирају;
- Ако је грешка у функцији библиотеке коригована заменом библиотеке, сви програми који је користе динамички ће имати користи од корекције, након поновног покретања.

Међутим, постоје и недостаци. За Windows платформу везан је појам „dll Пакао“, који настаје у ситуацијама када некомпатибилно ажуриране DLL библиотеке праве проблеме у раду оних програма чији је рад зависио од раније верзије DLL.

Програм, заједно са библиотекама које користи, може бити сертификован (нпр. по основу тачности или перформанси), као пакет, али не и у случају када компоненте могу да се замене. Ово такође представља аргумент против аутоматског ажурирања оперативног система у критичним системима. У оба случаја, оперативни систем и библиотеке чине део квалификоване средине.

Како преводилац нема информације о распореду објекатних датотека у коначном излазу, не могу се искористити краће или ефикасније инструкције које захтевају адресу још неког објекта. На пример, инструкција скока може указивати на апсолутну адресу или померај од тренутне локације. Међутим, померај се може изразити са различитим дужинама зависно од удаљености од позиције. Генерисањем најконзервативније инструкције, могуће је заменити краће или ефикасније инструкције у коначном повезивању. Овај корак се може извршити само након уноса свих објеката и доделе привремених адреса. У принципу, замењене секвенце су краће, што омогућава да овај

процес увек тежи најбољем решењу када је дат фиксни редослед објеката. Ако то није случај, може доћи до конфликта, и линкер треба да процени предности обе опције.

3. АРХИТЕКТУРА ПРОЦЕСОРА TFCOMIN

Инструкције процесора TFComin су фиксне дужине 16 бита. Основни формат инструкције је приказан на слици 3.1.

Слика 3.1 Основни формат инструкције процесора TFaComin

Сходно усвојеном формату инструкције архитектура процесора TFaCo има следеће карактеристике:

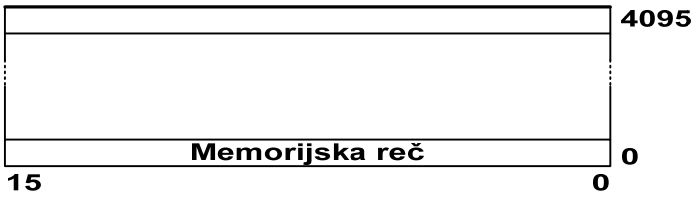
- Дужина операционог дела инструкције (COP) је **4 бита** (**$I_{12} – I_{15}$**).
- Функција осталих битова инструкције (**$I_0 – I_{11}$**) зависи од врсте инструкције и примењеног начина адресирања.
- Дужина процесорске речи је **16 бита**, а исту дужину има и **меморијска реч**, чија је дужина прилагођена дужини инструкције, која се на тај начин може дохватати у оквиру једног меморијског циклуса.

Архитектура процесора TFaComin одговара акумулаторској машини тако да су инструкције једноадресне. Преко **акумулатора** се врши пренос података у/из меморије.

3.1 Начини адресирања

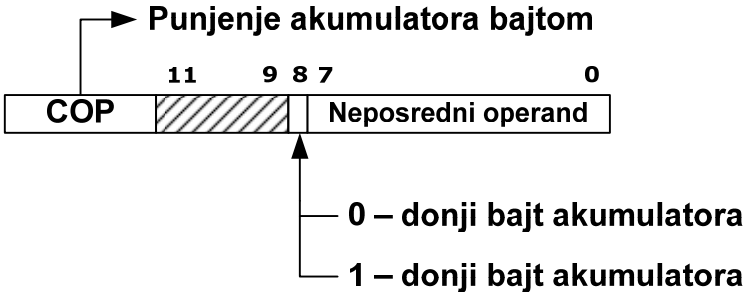
Процесор TFaComin омогућава следеће начине адресирања

1. **Директно меморијско адресирање** – Код овог начина адресирања битови инструкције **0 – 11** представљају адресу адресибилне јединице (меморијске речи дужине 16 бита) унутар странице од **2^{12}** (4096) локација што је истовремено и адресни простор овог процесора. Структура меморије система TFaComin је приказана на слици 3.2. Сходно овоме дужина **меморијског адресног регистра** (MAR) износи 12 битова. Овај начин адресирања примењује се и код инструкција којима се врши пренос података између **акумулатора** и **меморије**.



Слика 2. Структура оперативне меморије

2. **Непосредно адресирање** – Овај начин адресирања за пуњење акумулатора непосредном вредношћу операнда. Усвојено је да дужина непосредног операнда буде 8 бита (1 бајт). У том случају битови инструкције I_8 – I_{11} остају неискоришћени. С обзирмо, да је дужина акумулатора 16 бита ова инструкција може да се искористи да се бира у који бајт акумулатора ће се уписати непосредни операнд. За ту намену се користи бит 8. Ако је $I_8 = 0$ тада се непосредни операнд уписује у доњи бајт акумулатора, а ако је $I_8 = 1$ тада се непосредни операнд уписује у горњи бајт акумулатора, као што је приказано на слици 3.3.



Слика 3.3 Непосредно адресирање

3. **Инструкције скока** – Скокови код TFaComin могу бити безусловни или условни. Условни скокови се реализују на бази статусног регистра RS у који се по извршавању операција уписује одговарајућа статусна информација, према табели 3.1 и слици 3.4.

Табела 3.1 Значење битова у регистру статуса – R_9

Бит	Ознака	Значење
(0)	**Z – Zero Bit**	
	Z=0	Резултат операције различит од 0
	Z=1	Резултат операције једнак 0
(1)	**N – Negative Bit**	
	N=0	Резултат операције позитиван
	N=1	Резултат операције негативан

(2) C – Carry Bit

C=0 Нема преноса из најстаријег разреда резултата

C=1 Постоји пренос из најстаријег разреда резултата

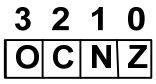
(3) O – Overflow Bit

O=0 Нема прекорачења опсега рачунања

O=1 Дошло је до прекорачења опсега рачунања

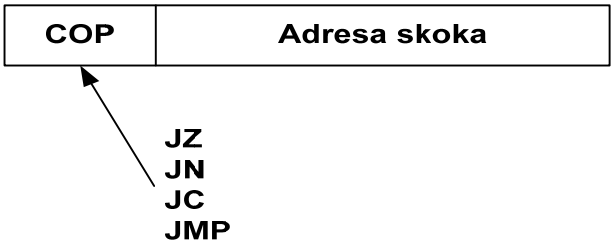
Због ограниченог броја битова намењених кодовању операција предвиђено је постојање само следећих типова скока:

- Безусловни скок;
- Условни скок по биту Z;
- Условни скок по биту N;
- Условни скок по биту C.



Слика 3.4 Структура статусног регистра RS

Спецификација услова који се тестира при условним скоковима дефинисана је кодом операције, а адреса скока је специфицирана у адресном пољу инструкције, као што је приказано на слици 3.5.



Слика 3.5 Формат инструкција скока

3.2 Скуп инструкција TFaCo и опис њиховог извршавања

COP	Симболички код	Синтакса	Опис
0000	**HALT**	HALT	**Заустављање програма**
0001	**LI**	LI lb, podatak	**Операција пуњења акумулатора непосредним операндом** ▪ **lb** – Једнобитна информација, који се налази на локацији $\mathbf{I_8}$, и која дефинише у који бајт се уписује непосредни операнд, који је дужине 1 бајт. o **lb=0** – податак се уписује у доњи бајт акумулатора o **lb=1** – податак се уписује у горњи бајт акумулатора ▪ **podatak** – 8 – битна информација. Помоћу ове наредбе, у општем случају, могу да се формирају следећи подаци у акумулатору: o Неозначена величина дужине 16 бита o Означена величина дужине 16 бита o Бинарни број дужине 16 бита o Четири хексадецимална броја o Два ASCII карактера o Један Unicode карактер
0010	**LOAD**	LOAD adresa	**Пуни акумулатор податком из меморије** ▪ **adresa** – меморијска адреса са које се чита податак
0011	**STORE**	STORE adresa	**Памти садржај акумулатора** 1. **adresa** – меморијска адреса на коју се уписује податак
0100	**ADD**	ADD adresa	**Сабирање** acc←(acc)+(adresa) – Функционални формат инструкције за сабирање
0101	**SUB**	SUB adresa	**Одузимање** acc←(acc)-(adresa) – Функционални формат инструкције за одузимање
0110	**NOT**	NOT adresa	**Логичка негација** acc←not(adresa)
0111	**OR**	OR adresa	**Логичко ИЛИ** acc←(acc) or (adresa) – Функционални формат инструкције за логичко ИЛИ
1000	**AND**	AND adresa	**Логичко И** acc←(acc) and (adresa) – Функционални формат инструкције за логичко И
1001	**XOR**	XOR adresa	**Ексклузивно ИЛИ** acc←(acc) or (adresa) – Функционални формат инструкције за ексклузивно ИЛИ
1010	**IN**	IN adresa	**Пуни акумулатор податком из У/И уређаја** acc←(adresa) ▪ **adresa** - адреса У/И порта са кога се чита податак
1011	**OUT**	OUT adresa	**Шаље податак из акумулатора у У/И уређај** adresa←(acc) ▪ **adresa** - адреса У/И порта на који се шаље податак
1100	**JZ**	JZ adresa	**Скок ако је резултат нула (Z=1)**

COP	Симболички код	Синтакса	Опис
1101	**JN**	JN adresa	**Скок ако је резултат негативан (N=1)**
1110	**JC**	JC adresa	**Скок ако постоји пренос (C=1)**
1111	**JMP**	JMP adresa	**Инструкција безусловног скока**

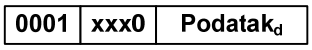
3.3 Описи бинарног формата инструкција TFaCo

При објашњавању начина кодовања инструкција користиће се следећа правила за означавање

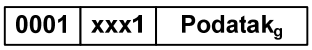
- **0 –** Вредност бита једнака 0
- **1 –** Вредност бита једнака 1
- **x –** Вредност бита није битна

HALT – Заустављање програма

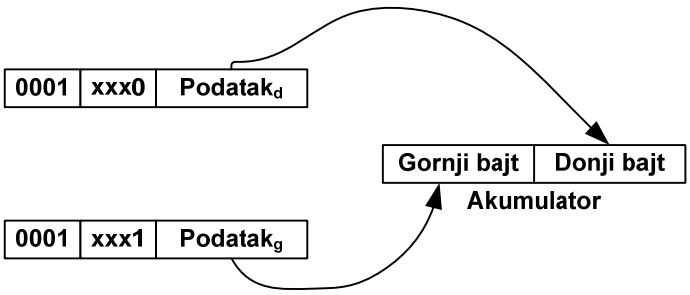
LI – Пуњење акумулатора непосредним операндом



Упис податка у доњи бајт акумулатора



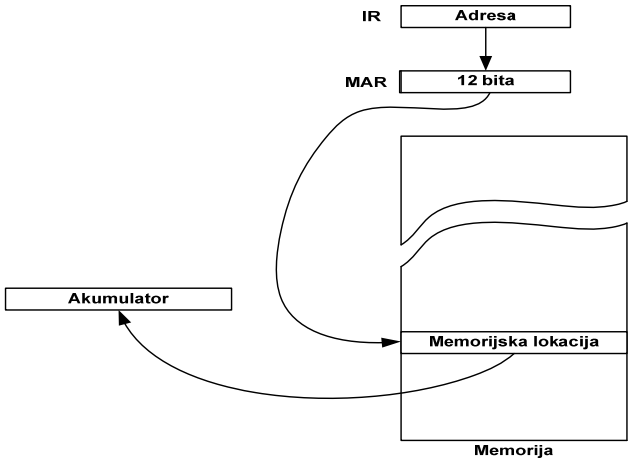
Упис податка у горњи бајт акумулатора



Начин извршења LI инструкције

LOAD – Пуњење акумулатора садржајем меморије

Пуњење акумулатора садржајем адресиране меморијске локације – специфицирана адреса одговара меморијској локацији са које се чита податак

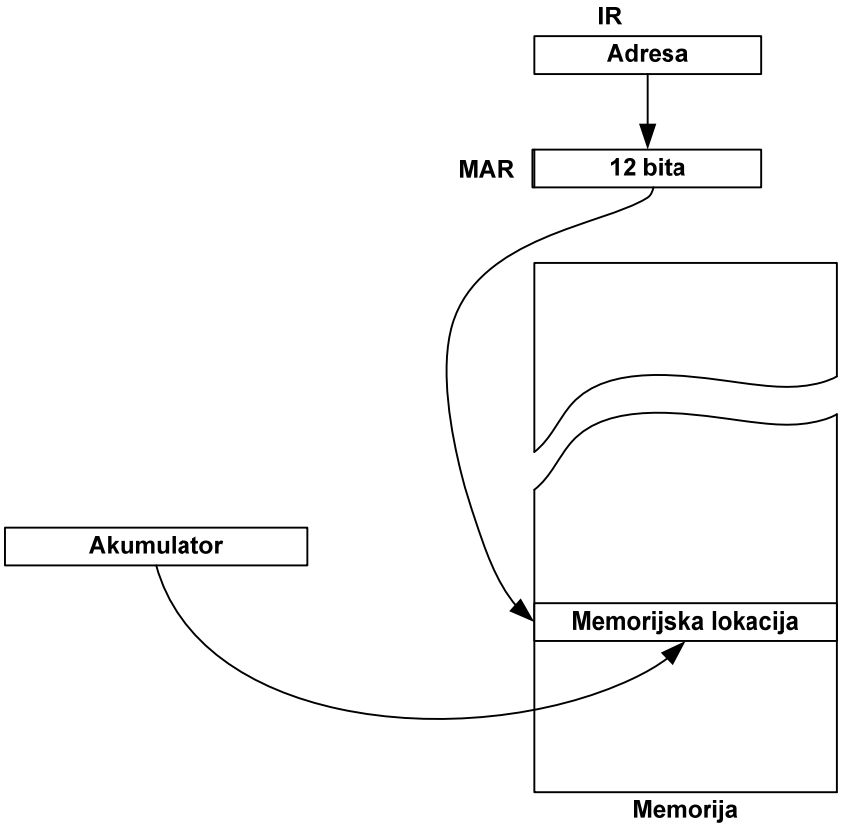


Начин извршавања инструкције LOAD

STORE – Памћење садржаја акумулатора у меморији

0011	Adresa u stranici

Памћење садржаја акумулатора у адресираној меморијској локацији – специфицирана адреса одговара меморијској локацији на којој се памти податак



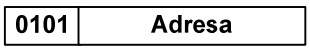
Начин извршења инструкције STORE

ADD – Сабирање

0100	Adresa

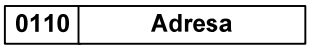
Формат инструкције ADD

SUB – Одузимање



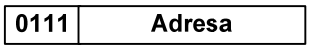
Формат инструкције SUB

NOT – Логичка негација



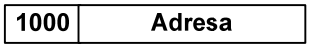
Формат инструкције NOT

OR – Логичко ИЛИ



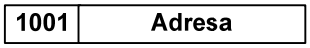
Формат инструкције OR

AND – Логичко И



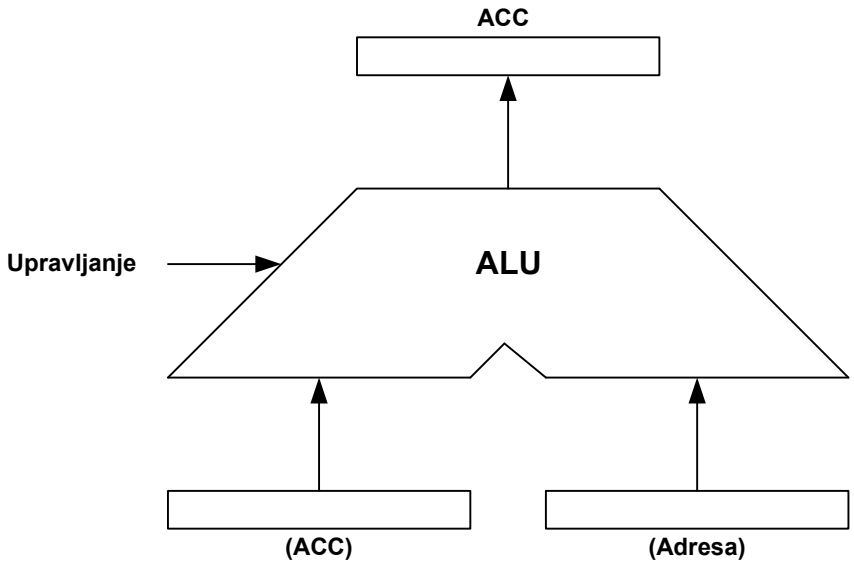
Формат инструкције AND

XOR – Ексклузивно ИЛИ

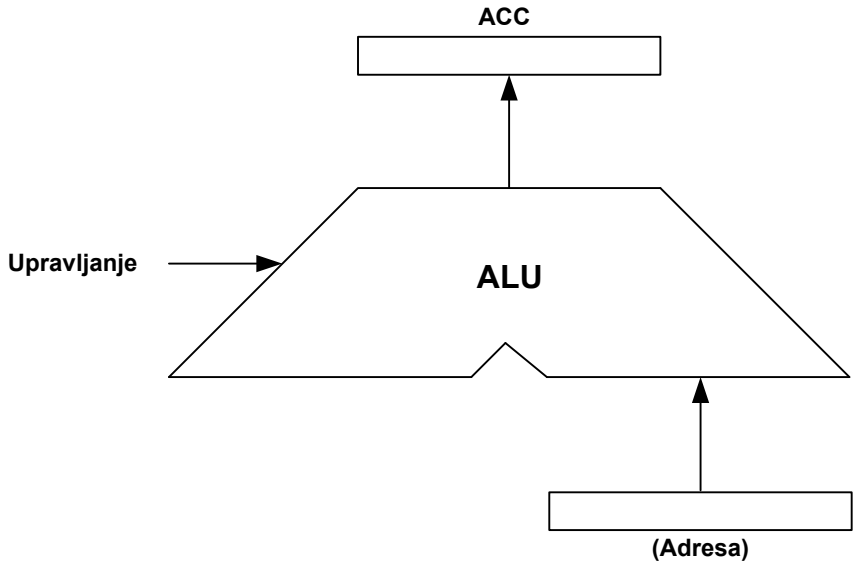


Формат инструкције XOR

Аритметичко – логичке инструкције ADD, SUB, OR, AND и XOR извршавају се према шеми на следећој слици:

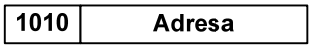


Шема извршавања аритмемтичко – логичких операција

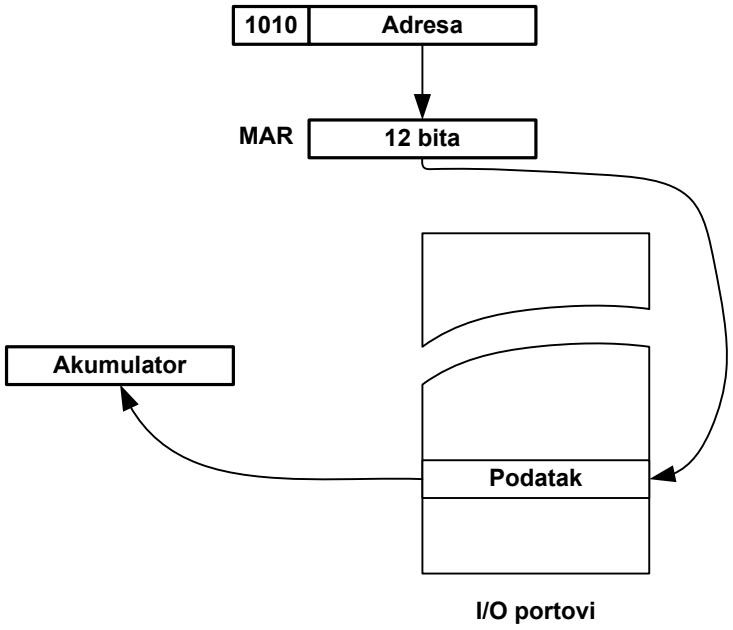


Шема извршавања инструкције NOT

IN – Пуњење акумулатора податком из У/И уређаја

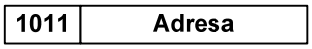


Пуњење акумулатора садржајем адресираог У/И порта

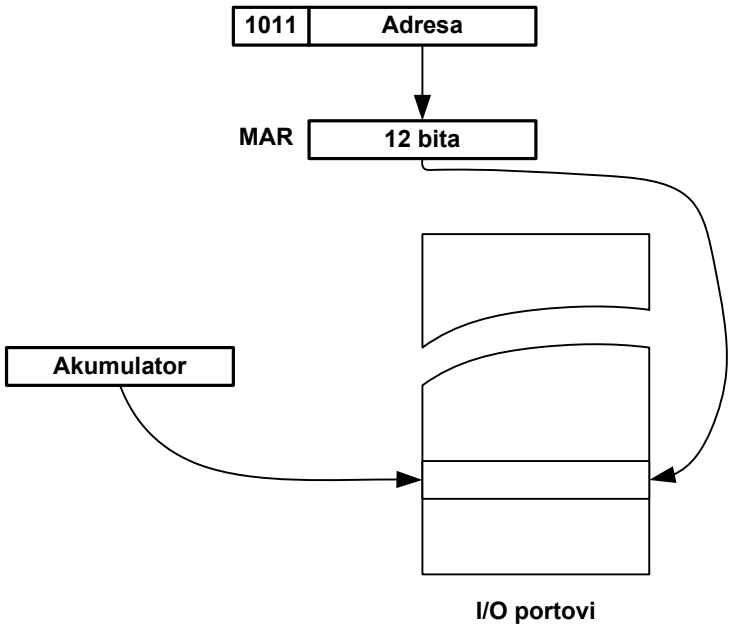


Начин извршавања инструкције IN

OUT – Слање садржаја акумулатора у У/И уређај

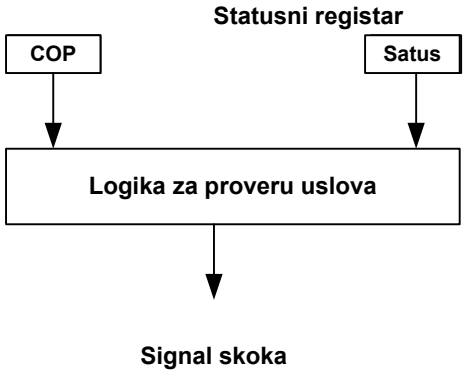


Слање садржаја акумулатора на адресирани У/И порт



Начин извршења инструкције OUT

JZ, JN, JC, JMP – Условни/безусловни скок



Шема извршења инструкције скока

4. КОНЦЕПТ АСЕМБЛЕРА ПРОЦЕСОРА TFCOMIN

Основни задатак сваког асемблера је да изврши превођење програма написаног на симболичком машинском језику у одговарајућу бинарну машински прихватљиву форму. Предмет овог дипломског рада је реализација асемблера за архитектуру процесора TFaComin, која је приказана у претходној глави.

4.1 Формат симболичке машинске инструкције

Да би се сагледали захтеви које асемблер за архитектуру процесора TFaComin односно да би могао да се прикаже концепт његове реализације потребно је подсетити се основних елемената ове архитектуре.

Основни формат инструкције је приказан на слици 4.1.

Слика 4.1 Основни формат инструкције процесора TFaComin

Из приказа архитектуре процесора TFaComin може да се уочи да његових 16 операција могу да се поделе у следеће подгрупе:

1. Операције које немају параметре – HALT;
2. Операције које имају један параметар (адресу меморијске локације која садржи операнд) – ADD, SUB, AND, OR, XOR, NOT, LOAD, STORE;
3. Операције које имају један параметар (адресу меморијске локације где се налази инструкција на којој се наставља програм у случају скока) – JMP, JZ, JN, JC;
4. Операције које имају један параметар (адресу улазно/излазног уређаја) – IN, OUT;
5. Операције које имају два параметра – LI (операција за непосредан унос бајта у акумулатор порд саме вредности има и једнобитни параметар којим је дефинисано да ли се дати бајт уноси у виши или нижи бајт акумулатора).

Сходно овоме асемблерске наредбе поред симболичког имена операције треба да садрже и дате параметре, било да је он дефинисан преко симболичког имена било да се

уноси у инструкцију као одговарајућа нумеричка вредност, која представља адресу или сам податак (LI инструкција).

Сходно задатку који треба да изврши, асемблер врши превођење инструкције која може да буде у целости представљена на симболички начин, коришћењем симболичких имена за операцију и параметре, или само делимично када је симболички представљена само операција. Пре него се дефинише сам асемблер неопходно је дефинисати одговарајући симболички машински или асемблерски језик, помоћу кога се пишу изворни симболички програми. У склопу тога неопходно је дефинисати начин писања симболичких имена начин њиховог коришчења у инструкцији, чиме се дефинише синтакса датог језика.

На слици 4.2 је приказана синтакса асемблерског језика процесора TFaComin:

Labela:	**COP**	**,**	**Simboličko ime parametra/ vrednost parametra**	**;**

Слика 4.2 Синтакса асемблерског језика процесора TFaComin

У принципу симболичка имена која могу да се декларишу код оваквих језика подлежу одређеним ограничењима, као што су: да морају имати неку унапред дефинисану максималну дужину и да обавезно морају почети словом. Као што се види са слике 4.2 у општем случају инструкција асемблерског језика процесора TFaComin се састоји из следећих елемената:

- Симболичког имена ознаке инструкције (лабеле) које се обавезно завршава двотачком (:) да би асемблер могао да дато име препозна као лабелу;
- Симболичког имена операције;
- Симболичког имена или нумеричке вредности параметра.

Између појединих елемената инструкције као гранични симболи користе се било симбол *бланко ( )* или *зарез (,)*. Свака инструкција се завршава знаком *тачка – зарез (;)*.

Пошто су на нивоу архитектуре процесора дефинисани бинарни кодови за поједине операције у табели 4.1 је дата кореспонденција између симболичких имена операција, која су дефинисана на нивоу асемблерског језика и њихових бинарних еквивалената.

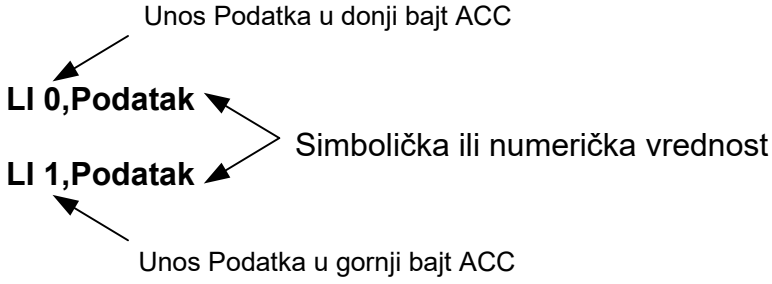
Табела 4.1 Табела симболичких имена операција и бинарних еквивалената

HALT	0000
LI	0001
LOAD	0010
STORE	0011

ADD	0100
SUB	0101
NOT	0110
OR	0111
AND	1000
XOR	1001
IN	1010
OUT	1011
JZ	1100
JN	1101
JC	1110
JMP	1111

У претходној табели дати су кодови операција процесора TFaComin. При писању симболичког кода при позивању одређене инструкције пише се њено симболичко име. Осим тога свака инструкција има своје аргументе, који се пишу иза симболичког имена. Код реализованог асемблера процесора TFaComin постоји 16 инструкција. Од тога, 14 инструкција имају само по један параметар. Инструкција „пуњење акумулатора бајтом" (*Load Immediate* – LI) има два параметра. Док једна инструкција, „заустављање процесора" (*Halt*) нема параметара.

Свака од ових 14 инструкција има као параметар адресу меморијске локације или адресу улазно/излазног порта. LI инструкција врши пуњење акумулатора непосредним операндом величине један бајт (8 бита). Пошто је акумулатор величине 16 битова, а на располагању у адресном делу инструкције остеје 12 битова, користи се само 8 битова за податак у LI инструкцији. У случају да је у акумулатор потребно непосредно унети 16 битни података, то се ради у два корака. Прво се у акумулатор уноси доњи, а затим горњи бајт. Сходно томе ова инструкција има формат, као што је приказано на слици 4.3.



Слиака 4.3 Формати инструкције LI

4.2 Асемблерске директиве

Асемблерске директиве или како се још називају псеудо – инструкције су наредбе које су намењене асемблеру и њима се дефинишу оквири у којима се реализује поступак превођења симболички написаног машинског програма у прави машински код. Они могу да учине да асемблирање програма зависи од параметара које уноси програмер, тако да један програм може да буде асемблиран на различите начине, можда и за различите апликације. Такође се могу користити за манипулацију приказа програма да би се олакшао посао програмеру да чита и одржава програм. На пример, директиве могу да се користие за резервисање области у меморији у коју се учитава програм пре извршавања. Имена директива често почињу са тачком да би се разликовале од машинских инструкција. Неки асемблери подржавају и псеудо – инструкције које генеришу две или више машинских инструкција.

Симболички или асемблерскии језици дозвољавају програмерима да повезују произвољна имена, која представљају лабеле или или имена променљивих са стварним меморијским локацијама. Обично, свакој константи и променљивој је дато име тако да унутар симболички написаног програма инструкције приступају подацима преко симболичких имена. Односно прелази у програму се остварују преко симболичких ознака (лабела) инструкција на којима се програм наставља. Такође, тачка улаза у потпрограм је повезана са његовим именом, тако да се позив потпрограма унутар симболички написаног програма остварује преко имена потпрограма. Неки асемблерски језици подржавају локалне симболе који се лексички разликују од нормалних симбола.

Већина асемблера обезбеђује флексибилну манипулацију симболима, што омогућава програмерима да управљају различитим именима, да аутоматски израчунају померај (offset) унутар структуре података, као и да доделе лабеле које се односе на вредности или резултат једноставног прорачуна извршеног од стране програма. Лабеле се такође могу користити да се иницијализују константе и променљиве са преносивим адресама.

Асемблерски језици, као и већина других програмских језика, дозвољавају стављање коментара изворном коду, који се игнорише од стране асемблера. Добра употреба коментара је још важнија него код виших програмских језика, пошто је значење низа инструкција теже утврдити из симболичког машинског кода. Коришћење ових објеката може знатно поједноставити писање и одржавања програма написаног на симболичком машинском или како се још назива асемблерском језику. Машински код генерисан од стране асемблера или *disassembler* – а без било каквих коментара, или

дефиниције података, прилично тежак за читање у случају да је потребно да се изврше одређене измене у програму.

Од асемблерских директива процесор *TFaComin* поседује асемблерску директиву **EQU**. Асемблерске инструкције су симболичка представа машинског језика. Свака асемблерска инструкција производи једну машинску инструкцију. Асемблерске директиве дају инструкције асемблеру, али се на бази њих не генерише машински код. Оне омогућавају лакше писање и одржавање програма писаног на симболичком машинском језику. У практичном смислу ове наредбе има ју свој пандан код виших програмских језика у виду декларационих инструкција. Због тога постоји више разлога зашто је добро, и у неким случајевима веома корисно, увођење одређених директива. У следећем примеру приказано је коришћење директиве EQU да би се успоставила коресподенција између симболичког имена променљиве и њене стварне вредности.

```
DATA EQU 78h;
ADR1  EQU 001111001101b;
MEM1 EQU FBFh;
MEM2 EQU 767o;
MEM3 EQU 101010101010b;
```

На овај начин се не морају памтити стварне бројне вредности, већ се уместо тога памте симболичка имена, која могу да асоцирају на величину којој је оно придружено, што додатно олакшава памћење. Поред начина коришћења директиве EQU у претходном примеру могуће је видети и неке друге елементе синтаксе асемблерског језика процесора TFaComin. На крају нумеричке вредности додаје се једно од слова *b*, *d*, *h* или *o* што означава да је нумеричка вредност представљена у *бинарном*, *децималном*, *хексадецималном* или *окталном* бројном систему респективно. Такође свака инструкција у овом асемблерском језику се завршава карактером *;* који је индикатор краја програмског реда. Осим тога поједини елементи инструкције се раздвајају или знаком *бланко* или *зарезом*. У претходном примеру прва линија садржи декларацију:

- Променљиве DATA која представља податак чија је вредност 78 хексадецимално. У принципу овде је дефинисан података који ће се вероватно касније у програму појавити као параметар инструкције LI;
- Остале линије садрже декларације променљивих ADR1, MEM1, MEM2 и MEM3, које ће се касније у програму користити као адресе.

Коришћењем асемблерских директива програм постаје значајно читљивији за програмере, како за онога ко изворно пише програма, тако и за оне који врше одређене измене у њему. Веома важно је напоменути да је захваљујући симболичком приступу и на нивоу променљивих олакшано праћење алгоритма који реализује дати симболички написан програм. Такође, пожељно је давање карактеристичних симболичких имена променљивим, тако да програмер може лакше да памти одговарајући податак.

Коришћењем директиве EQU могуће је извршити и редефиницију кодова операција, као што је приказано у следећем примеру:

```
SABIRANJE     EQU ADD;
ODUZIMANJE  EQU SUB;
…
```

На овај начин се и саме наредбе могу заменити са неким другим симболичким именом, које ће можда бити лакше за памћење програмеру, и самим тим му и олакшати посао писања програма. У датом примеру порд симболичких имена операција ADD и SUB у програму је могуће користити и алтернативна симболичка имена SABIRANJE и ODUZIMANJE.

4.3 Специјални карактери и њихово значење

Специјални карактери представљају симболе који имају посебно значење, тј. ако се појаве на одређеном месту у изворном коду, имају специјално значење. Овакви карактери, својим специјалним значењем помажу да се на једноставнији и лакши начин напише неки изворни програм, реши лакше неки проблем, или једноставно скрати дужина изворног програма, итд. На овај начин изворни програм, осим што постаје краћи, постаје разумљивији и логичнији за преглед, за евентуалне исправке и проналажење грешака.

У асемблерском језику процесора *TFaComun* дефинисано је неколико специјалних карактера. Они се пре свега односе на поступак конверзије бројних вредности из једног бројног система у други бројни систем. У конкретном случају омогућено је да се бројне вредности могу представљати у *бинарном*, *децималном*, *окталном* и *хексадецималном* бројном систему. То доноси доста предности у смислу писања изворног програма, али са друге стране доноси и одређене проблеме, које морају бити разрешени и превазиђени током асемблирања. Ти проблеми проистичу из чињенице да захтев за равноправним коришћењем различитих бројних система ствара проблеме на нивоу развоја самог асемблера, јер он мора да препозна који је бројни систем коришћен и да обезбеди

конверзију одговарајуће нумерилке вредности у бинарну вредност одговарајуће дужине.

Када се у изворном коду појави одређена бројна вреденост, може лако доћи до забуне у ком је формату тај број написан. У случају када је дат број у облику A7FF63 анализом је могуће утврдити да се ради о броју написаном у хексадецималном облику. Међутим, када се у програму нађе број 10100, није тешко закључити да дати број у принципу може бити истовремено бинаран, октален, децималан и хексадецималан. Јасно је да се јавио проблем – како препознати стварни бројни формат ове нумеричке вредности и како то детектовати у самом програму, тј како асемблер да зна у ком бројном систему је написан тај број.

Да би се илустровао начин решавања овог проблема у код овог асемблерског језика дат је пример симболичког машинског кода у коме су декларисане вредности неких симболичких променљивих:

```
ATA    EQU 1;
ADR    EQU 0;
ADR1   EQU 000000000001;
MEM1 EQU FFF;
MEM2 EQU 777;
MEM3 EQU 101010101010;
```

Као што се види из овог примера, без детаљене лексичке анализе није могуће знати да ли су одређене вредности написане у бинарном, децималном или неком другом бројном систему. У неким случајевима, као што је напред објашњено то уопште и није могуће извршити, без обзира на примењену анализу. У конкретном случају за решавање овог проблема у асемблерском језику који је дефинисан за архитектуру процесора TFaComin уведени су специјални карактери који се користе за означавање коришћеног бројног система. Сходно усвојеној конвенцији претходни пример се може написати на следећи начин:

```
ATA    EQU 1d;
ADR    EQU 0d;
ADR1   EQU 000000000001b;
MEM1 EQU FFFh;
MEM2 EQU 777o;
MEM3 EQU 101010101010b;
```

Као што се видимо на датом примеру, додавањем специјалног карактера на крају бројне вредности додата је информациј у ком је бројном систему је представљена конкретна бројна вредност. На овај начин се бројна вредност једнозначно одређује, и не може доћи до забуне или грешке.

Специјални карактери који су овде дефинисани имају следеће значење:

- **b** – бинарни бројни систем
- **d** – децимални бројни систем
- **o** – октални бројни систем
- **h** – хексадецимални бројни систем

Асемблер процесора *TFaComun* написан је тако да када детектује бројну вредност у симболичком коду, прво чита карактер који стоји на крају, тј после саме бројне вредности, и у зависности од тог карактера врши конверзију датог броја из датог, тј детектованог бројног система, у бинарни бројни систем. На овај начин ће број бити увек правилно конвертован.

На овај начин се, уз примену специјалних карактера, омогућава се коришћење сва 4 бројна система равноправно, и у многоме се олакшава посао при писању изворног кода. Из овог примера коришћења специјалних карактера се види њихов значај, који још више долази до изражаја што је сама архитектура одређеног процесора сложенија.

4.4 Начин писања лабела

Лабеле су симболичка имена која се користе за идентификацију програмске инструкције на којој се наставља програм у случају успешног извршења инструкције скока. Лабела се састоји од одређеног броја алфанумеричких карактера. SADA, LOK21, R2D2, и C3PO су неки од примера могућих лабела.

```
.....
    ADD ADR1;
L1: AND MEM1;
…….....
```

У претходном примеру је дат случај 2 линије изворног кода, и то прва у којј не постоји лабела и друга у којој се налази лабела L1. Као што је напред објашњено другој линији програмског кода захваљујући што је означена лабелом L1 може се приступити и помоћу наредбе скока, чиме се програмеру омогућава да у програму оствари све конструкције које су дефинисане алгоритмом решења проблема за који се програм пише. Треба обратити пажњу да иза симболичког имена које означава лабелу стоји знак **:**. Овај карактер, као и већ поменути карактери *бланко ( ), зарез (,), тачка – зарез (;)* има

улогу граничног карактера и управо значи да симболичко име испед њега представља лабелу (ознаку) програмског реда у коме се она налази.

4.5 Модули реализованог асемблера

Реачизовани асемблер представља јединствену програмску целину реализовану преко различитих програмских модула. Програмер „пише" симболички машински програм, који помоћу одговарајућег програма преводиоца – асемблера треба превести у машински програм, тј. одговарајућу бинарну форму. Да би асемблер могао да испуни постављени задатака мора да поседује одговарајуће модуле који ће моћи да реализују следеће операције:

- Секвенцијално читају текстуалну датотеку са инструкцијама изворног симболички написаног машинског програма. Секвенцијално читање подразумева читање инструкција по инструкција;
- Лексичку анализу прочитане инструкције, која подразумева идентификовање свих делова инструкције, укључујући проверу исправности синтаксе написаног програма;
- Током лексичке анализе програма формирају се HASH табеле за следеће елементе инструкције:
    - *Скуп симболичких имена операција* које могу да се специфицирају унутар инструкција. У оквиру ове табеле се успоставља кореспонденција између симболичког имена операције и бинарног еквивалента (кода операције);
    - *Скуп симболичких имена дефинисаних у оквиру декларационих инструкција* (коришћењем директиве EQU). У оквиру ове табеле се успоставља кореспонденција између дефинисаних симболичких вредности и додељене им нумеричке вредности;
    - *Скуп лабела.* У оквиру ове табеле се успоставља кореспонденција између симболички дефинисане лабеле и броја инструкције којој је придружена.
- У оквиру лексичке анализе операционих инструкција врши се идентификација елемената инструкције – симболичког имена операције, параметра операције и лабеле ако је специфицирана у инструкцији. Коришћењем одговарајућих HASH табела врши се замена симболичког имена одговарајућом бинарном вредношћу. У случају када је нумеричка вредност није дата у бинарном бројном систему врши се одговарајућа

конверзија. У овом тзв. „првом пролазу" асемблера не врши се замена идентификованих лабела – као одредишта већ се то оставља за „други пролаз".

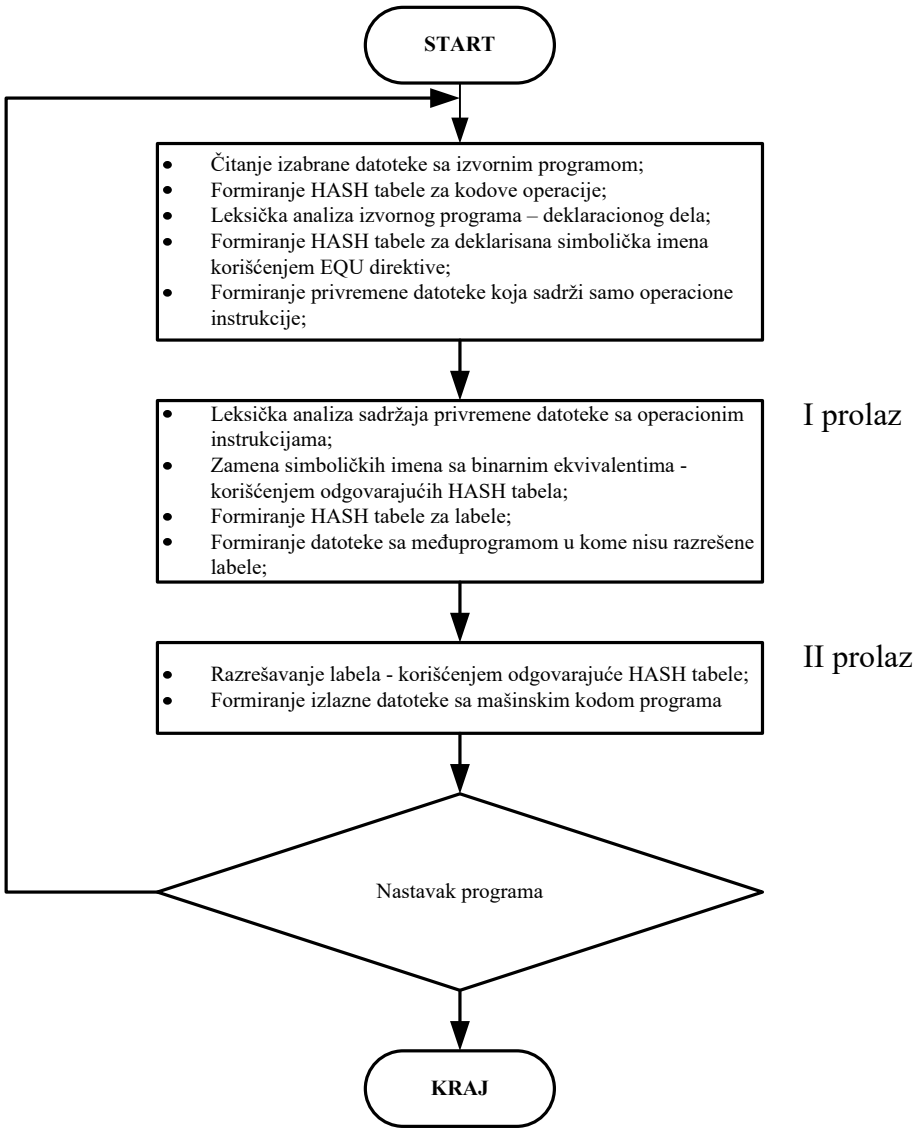
– У оквиру „другог пролаза" врши се тзв. разрешавање лабела, тј. у међу коду се идентификују инструкције код којих су одредишне лабеле остале у симболичком облику па се врши њихова замена одговарајућим бројевима инструкција коришћењем HASH табеле за лабеле.

4.6 Аспекти реализације асемблера

Данас се асемблери по правилу пишу у неком вишем програмском језику. Као што се види из описа операција које асемблер треба да реализује потребно је да коришћени програмски језик има добру подршку за манипулацију карактерима, низовима карактера и битовима. Пошто је циљ био да се направи асемблер који ће моћи да буде солидно образовно средствао у настави архитектуре рачунара, при чему није било битно добијање високо ефикасног извршног кода изабрано је да асемблер за TFaComin архитектуру буде реализован на бази програмског алата JAVA. Овај програмски алат поседује све тражене карактеристике уз могућност одличне графичке визуелизације тока извршавања процеса асемблирања – превођења симболички написаног машинског програма.

Циљ реализације овог асемблера је био да се полазећи од дефиниције инструкцијске архитектуре процесора TFaComin добије одговарајући преводилац који ће читајући текстуалну датотеку са изворним симболичким програмом извши њено превођење у бинарни облик који представља машински програма датог процесора. Избор полазне датотеке са изворним програмом врши се коришћењем одговарајућих контрола у „прозору" програма асемблера тако што се из скупа датотека које садрже изворне програме бира датотека чији програм треба превести. Ове датотеке имају екстензију TFC/tfc. У изабраној датотеци се налази изворни програм написан у симболичком машинском језику дефинисаном у складу са архитектуром процесора TFaComin. Као крајњи резултат требамо да се добије текстуални фајл са истим именом и екстензијом OBJ/obj који треба да представља еквивалент машинском програму за процесор TFaComin. То значи да ће излазна датотека садржати симболичку представу машинског кода у којој су бинарне вредности представљене одговарајућим симболима цифара 0 и 1.

Сходно операцијама које треба да изврши, рад асемблер процесора TFaComin може да се опише на дијаграмом датима на слици 4.4:



Слика 4.4 Ток рада асемблера

Као што се види са слике прво се отвара и чита датотека која садржи симболичка имена операција и бинарне кодове операција – **cop.txt**. На бази њеног садржаја формира се HASH табела као излаз. Код формирања HASH табела веома важан фактор је избор hash функције која неће довести до колизије, тј. неће доћи до преклапања путања унутар формиране табеле. Када се заврши формирање HASH табеле кодова операција, у којој свако симболичко име има свој улаз тако да се сваком од њих можемо приступити преко hash броја, који се при шитању формира већ дефинисаном hash функцијом. HASH табела треба да се памти у **hash_cop.txt**.

Након тога следи обраде декларационог дела програма који садржи асемблерске наредбе које садрже EQU директиву. У случају да постоје ове наредбе формира се HASH табела декларисаних вредности променљивих. Из изворног програма се избацују декларационе наредбе и остатак програма преписује у привремену датотеку **prog.tmp**.

HASH табела за симболичка имена декларисана преко EQU директиву треба да се памти у датотеци **hash_var.txt**.

Пошто се у оквиру програма могу наћи бројне вредности које могу бити у бинарном, децималном, октално или хексадецималном систему, неопходно је обезбедити једнозначну конверзију тих вредности у бинарне бројеве. Ова конверзија се врши у случајевима када су вредности које се додељују симболичким именима дате у разним бројним системима или су у програму у операционим инструкцијама дефинисане као нумеричке вредности које нису дате у бинарном бројном систему. Алтернатива овом приступу је да се конверзија врши приликом узимања вредности из HASH табеле.

Када се асемблира одређени програм писан на симболичком машинском језику, асемблер иде редом читајући инструкцију по инструкцију. Ако у оквиру кода постоји референца на лабелу на одређеној локацији, асемблер не може одмах да зна где се налази та локација на коју се лабела односи. Због тога асемблер мора да изврши више „пролаза" кроз изворни код. У првом пролазу се проналазе лабеле, као ознаке одређених програмских редова и формира се одговарајућа HASH табела за лабеле, тако да се за следећи пролаз зна где се налази дата лабела, ако се у некој инструкцији специфицира прелаз на инструкцију означену лабелом, путем наредбе скока. На овај начин се, разрешавањем лабела, формира коначна објектна датотека. Формирана HASH табела мора бити смештена у датотеку. У конкретном случају изабрано је да одредишна датотека HASH табеле лабела има име **hash_label. txt**.

Као што се види, асемблер да би извршио потребне операције треба да се састоји из више целина, од којих свака има одређену функцију. Кроз цео поступак асемблирања врши се контрола синтаксне исправности изворног симболичког програма.

5. РЕАЛИЗАЦИЈА АСЕМБЛЕРА ПРОЦЕСОРА TFCOMIN

Асемблер преводи програмски код са симболичког машинског језика на машински језик водећи рачуна о свим правилима превођења. Концептуално асемблер није реализован као јединствена програмска целина, већ се састоји из више логичких делова – модула. Сваки модул има своју функцију и извршава одређене радње које су битне за целокупан рад асемблера. Пошто модули обављају одређену функцију она се може описати као посебна целина.

По дефинисању концепта, структуре и функција које асемблер треба да реализује приступило се његовој реализацији. За реализацију асемблера, који је предмет овога рада изабрано је развојно окружење JAVA, које пружа добру подршку за манипулацију карактерима и низовима карактера, што представља једно од најважнијих обележја операција које сваки асемблер мора да изврши у процесу превођења. Реализација асемблера конкретно подразумева, писање програма у вишем програмском језику који ће вршити превођење – асемблирање текстуалне датотеке са симболичким програмским кодом написаним помоћу симболичког језика.

Сагледавањем могућности које треба да поседује реализовани асемблер, ствара се слика о логичким целинама које треба да чине асемблер. Другим речима, долази се до уочавања модуларности коју треба остварити на нивоу асемблера. Сваки модул треба да обавља део посла у процесу асемблирања, а самим тим долази се до кореспонденције између посла који треба да буде извршен и модула у оквиру асемблера који то треба да обави.

Реализовани асемблер се састоји из следећих модула:

- Модули за формирање HASH табела за:
    - симболичка имена операције;
    - декларисана имена променљивих;
    - симболичка имена лабела;
- Модули за конверзију бројних формата у бинарни;
    - из децималног у бинарни;
    - из окталног у бинарни;
    - из хексадецималног у бинарни;

– Главни програмски модул за асемблирање:
    - позвање свих осталих модула уз помоћ којих се врши асемблирање симболичког машинског кода у машински програмски код. На тај начин се сви развијани модули асемблера процесора TFaComin обједињују у једну целину која извршава своју предвиђену функцију – асемблирање смболичког кода

5.1 Опис појединих модула асемблера и њихових алгоритама

Као што је већ објашњено асемблер се састоји из одређеног броја програмских модула. Да би се схватила логика рада асемблера потребно је описати сваки од датих модула и објаснити принципе њиховог функционисања, као и како се остварује њихово повезивање у целину какву представља асемблер. Такође ће бити објашњени и алгоритми по којима раде.

Основа сваког асемблерског језика, су процесорске инструкције. Сам облик и број инструкција зависи од архитектуре датог процесора. Конкретно, процесор TFaComin је 16 – битни, што значи да је његова процесорска реч дужине 16 бита. Такође, је усвојено да је дужина инструкција фиксна и има 16 бита, од којих су 4 бита резервисана за код операције, што значи да је максималан број операција које се могу специфицирати на нивоу овог процесора 16. У табели 5.1 дат је преглед симболичких имена операција и њихових бинарних еквивалената који представљају одговарајуће кодове операција.

Табела 5.1 Табела кодова операција

HALT	0000
LI	0001
LOAD	0010
STORE	0011
ADD	0100
SUB	0101
NOT	0110
OR	0111
AND	1000
XOR	1001
IN	1010
OUT	1011

JZ	1100
JN	1101
JC	1110
JMP	1111

Симболичка имена, уместо бројних или чак директно бинарних кодова, помажу програмеру да лакше напише програм, али да би се тај програм извршио мора се извршити конверзија из симболичког у бинарни облик. Тај поступак се назива асемблирање. Када се у изворном програму или тексту програма детектује симболичко име, асемблер мора да изврши његово превођење у машински облик у складу са операцијом која је специфицирана у датој инструкцији. Као што се види из табеле 5.1 иницијално се на нивоу симболичког језика могу предефинисати имена кодова операције. На тај начин програмер се ослабађа од потребе да врши успостављање кореспонденције између бинарног кода операције и симболичког имена које би сам програмер могао да изабере на нивоу појединих операција. У принципу у оквиру симболичког машинског језика могу да се предефинишу и имена која су везана за неке друге елементе инструкције, а такође могу да се дефинишу и имена асемблерских директива, која у принципу могу да се као таква користе у неизмењеном облику, током писања изворног симболичког програма. Таква имена, која се третирају као симболичке константе у оквиру симболичког језика називају се **кључне речи**.

У принципу асемблери дозвољавају да се мењају предефинисана имена увођењем нових имена која у програмском смислу продукују исти бинарни код. Такође, могуће је увести нова имена која одговарају програмским променљивим и којима се морају доделити одговарајуће нумеричке вредности. У ту сврху се код асемблерских језика, али и код свих виших програмских језика уводе тзв. декларационе инструкције, које de facto представљају само информацију за преводилац и не продукују одговарајући машински код. Код реализованог асемблера функцију идентификатора декларације има асемблерска директива EQU, којом се могу доделити друга симболичка имена за операције, али и извршити декларисање, тј. успостављање кореспонденције између симболичких имена променљивих и њихових вредности. Коришћење директиве EQU је илустровано следећим примерима:

Дефинисање нових имена операција

```
SABIRANJE       EQU  ADD
ODUZIMANJE      EQU  SUB
SKOK            EQU  JMP
```

Дефинисање – декларисање промењљивих

```
ADR    EQU   0d;
ADR1   EQU   000000000001b;
MEM1   EQU   FFFh;
```

Могућност декларисања имена програмер може користити током програмирања, јер се симболичка имена могу лакше памтити, а и вероватноћа грешке код употребе симболичких имена је мања од грешке која се може направити у случају вишеструког коришћења, нрп. дугачких бинарних бројева. Због коришћења симболичких имена уместо непосредних вредности односно бинарних кодова у оквиру асемблера постоји посебан модул који формира HASH табелу која омогућава лакши приступ и замену симболичких имена са бинарним кодовима у излазном – машинском програму. Захваљујући овом приступу промењљива са датим именом и одговарајућом вредношћу, која се у декларацији налази са десне стране директиве EQU смешта се у табелу по индексу који се добија израчунавањем HASH функције. На овај начин се у оквиру симболичког кода може преко имена промењљиве доћи до њене вредности уместо да се она памти у бинарном облику. То је још једна значајна олакшица при писању програма, за програмера.

Да би то могло да се реализује у принципу морају да се памте релације између нових симболичких имена промењљивих и вредности које су им додељене. Због тога се HASH табеле памте у оквиру датотеке **hash_var.txt**. Када се при асемблирању идентификује симболичко име промењљиве, приступа се одговарајућој HASH табели и чита њена вредност.

У оквиру програма се може јавити потреба за коришћењем наредбе скока. Као аргумент наредби скока користи се ознака инструкције на којој се наставља програм у случају успешног скока. Да би се лакше управљало скоковима у програму, испред линије на којој се потенцијално може наставити програм додаје се одговарајућа ознака која се назива **лабела**, као што је приказано у следећем примеру:

```
        JMP POZ1;
        ……....
        ……
        ……....
POZ1:   ADD MEM1;
        ……....
```

Претходни низ наредби извршава скок на инструкцију која је означена са POZ1, и асемблира се линија симболичког кода ADD MEM1. На овај начин се, слично као и у претходним случајевима, олакшава програмирање. Све лабеле се такође морају памтити, и то у оквиру одговарајуће HASH табеле. HASH табела за лабеле се памти у датотеци **hash_label.txt**. Захваљујући овоме у току асемблирања симболичког кода када се идентификује скок на лабелу, тада се приступа HASH табели за лабеле да би се одредила позиција за скок, тј. којој инструкцији у програму кореспондира инструкција означена са датом лабелом.

5.2 Начини формирања HASH табела

Приликом асемблирања, уочено је да је повољно користити HASH табеле као неку врсту складишта за лабеле, променљиве, кодове операције. HASH табеле представљају врсту структуре података у којима се складиште информације. Суштина HASH табеле представља HASH функција, која дефинише место податка у HASH табели. Она генерише HASH број преко кога се приступа елементима табеле. HASH функција генерише HASH број узимајући као улазну променљиву неки параметар елемента који се смешта у табелу, нпр дужину стринга. HASH функцију треба изабрати на тај начин да избегнемо колизије унутар табеле. Под колизијом се подразумева да се за различито симболичко име применом HASH функције добије исти улаз у HASH табелу. Најједноставнији начин превазилажења колизије је да се помери на следећи улаз у HASH табели провери да ли је заузет и ако није онда он користи за смештај жељених информација. У противном се врши померање на следећи улаз и тако редом.

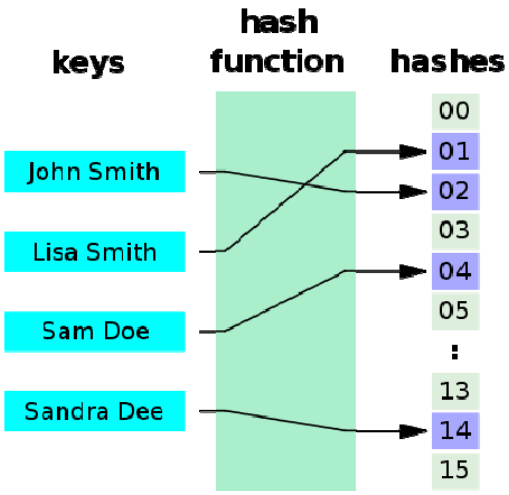
Да би се лакше објаснио појам HASH табеле, тј HASH функције, искористиће се конкретан пример HASH функције која је коришћена при реализацији овог асемблера

Broj = Asc(Slovo$)

*Hash_Broj = Hash_Broj + Broj * Broj * Broj Mod 7777 + Broj * Broj + Broj*

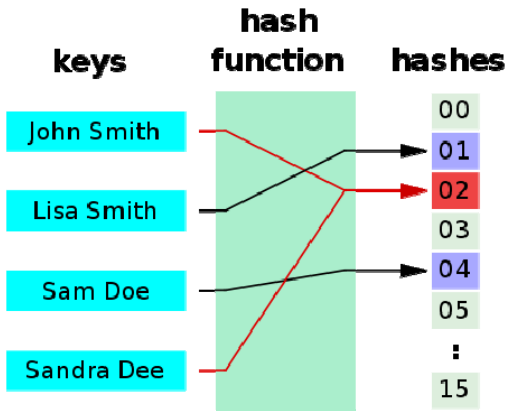
Променљива *Broj* је ASCII вредност одређеног слова из речи која се уноси у табелу. На основу вредности које се добијају за сва слова у речи, израчунава се коначни HASH функција *Hash_Broj.* Затим се елемент, уз напомену везану за проблем колизије, уноси у табелу и може му се приступити преко HASH броја.

Међутим, као што је наведено постоје одређени проблеми о којима морамо водити рачуна при формирању HASH функције. Највећи од њих је колизија. На слици 4.2 је приказана HASH табела без колизија, тј за сваки улазни елемент се израчунава ***јединствен HASH број*** путем HASH функције.



Слика 4.2 HASH табела без колизија (извор Википедија)

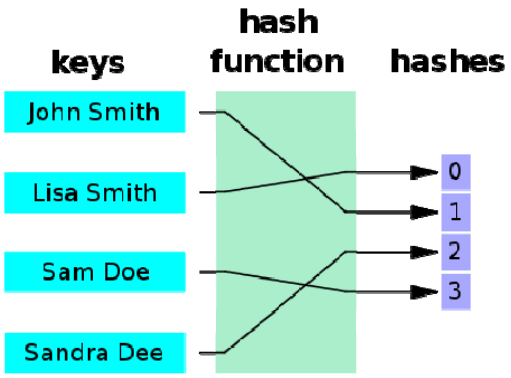
Као што је напред речено у неким случајевима за изабрану HASH функцију може доћи до појаве која се назива колизија. Пример табеле са колизијом је дат на слици 4.3.



Слика 4.3 HASH табела са колизијом (извор Википедија)

Проблем настаје зато што два улазна елемента имају исти HASH број, и када је потребно извршити повратни процес добијања елемента преко HASH броја, добила би се погрешна вредност, зато што постоји више елемената са истим HASH бројем. Разрешавање овог проблема се може урадити на више начина. Сходно томе постоје многе методе о којима овде неће бити речи, али је у суштини најбоље имати функцију која не доводи до колизија или је вероватноћа колизије веома мала.

Идеалан пример HASH функције је када би она генерисала онолико излаза, тј. резултата, колико имамо и улазних елемената, као што је приказано на слици 4.4.



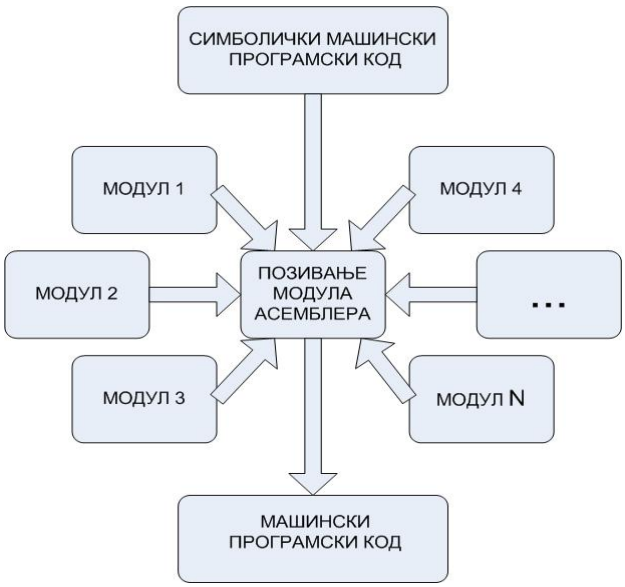
Слика 4.4 Идеална HASH табела (извор Википедија)

5.3 Конверзија бројних формата у бинарни

Када се у изворном коду појави бројна вредност, мора се превести у бинарни облик. Да би се експлицитно знало у ком је бројном формату, тј. бројном систему написан нека нумеричка вредност неопходно је на нивоу симболичког језика имати једнозначну идентификацију по том питању. Пошто у случају овог асемблерског језика број може бити представљен у декадном, окталном, хексадецималном или у бинарном бројном систему неопходно је да уз вредност постоји идентификатор бројног система у коме је она представљена. У реализованом асемблеру, тј. на нивоу одговарајућег симболичког језика, иза сваке бројне вредности стоји једна од словних ознака *d*, *o*, *h*, *b*, које означавају децимални, октални, хексадецимални и бинарни број систем респективно. Приликом асемблирања, када се прочита бројна вредност, прочита се и словна ознака иза. Тада се, на основу словне ознаке, примењује одговарајућа бројна конверзија у бинарни бројни систем. На принципу тог алгоритма ради модул за бројну конверзију, асемблера процесора TFaComin.

5.4 Повезивање модула за асемблирање

Сви модули се развијају као засебне логичке целине. При асемблирању, у зависности од потребе, активирају се сви ови модули. Коришћењем свих њих добијамо програм на машинском језику, прилагођен нашој архитектури. Сваки од ових модула је подједнако важан, и асемблирање не би било могуће без било којег од њих. Модуларност асемблера је принципски приказана на слици 5.5.



Слика 5.5 Позивање модула при асемблирању

5.5 JAVA реализација асемблера за TFComin

Све шира потреба за дистрибуцијом информација путем Интернета утицала је да све више програмских апликација са стоних рачунара почне да мигрира на ниво серверских апликација. С обзиром да се оваквим серверским апликацијама приступа путем Интернет развој таквих апликација добио је атрибуте Интернет или WEB програмирања. Сходно томе савремени програмски језици се специфицирају са претпоставком подршке овом типу апликација. Треба уочити да су до сада Интернет апликације биле релативно профилисане и ограничене на релативно узак скуп примена. Асемблери свакако нису били једна од таквих апликација. Имајући у виду ову чињеницу реализација асемблера је представљао мали програмсерски изазов, поготову са аспекта специфичности обраде података какву захтева развој асемблера.

За развој асемблера веома је важан аспект рада са датотекама, јер се при обради изворног симболичког програма врши формирање неколико нових датотека, без обзира што су оне, осим излазне датотеке са машинским кодом, привременог карактера и бришу се по завршетку рада асемблера. Такође, реализација програма асемблера захтева рад са листама, реализацију неких карактеристичних алгоритама, као што је нпр. HASH алгоритам и интензиван рад са подацима типа – низови карактера (character string).

У наставку текста биће дат комплетан JAVA код асемблера за процесор TFComin. Из листинга програма, који садржи потребне коментаре добро се виде све фазе реализације асемблера. Такође, дат је и изглед подршке, коју апликација даје у погледу визуелизације рада асемблера на нивоу WEB простора.

```
import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.Writer;
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

import javafx.animation.PauseTransition;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
/**
 * Servlet implementation class Servlet2
 */
@WebServlet("/Servlet2")
public class Servlet2 extends HttpServlet {
	private static final long serialVersionUID = 1L;

    /**
     * @see HttpServlet#HttpServlet()
     */
    public Servlet2() {
        super();
        // TODO Auto-generated constructor stub
    }

	/**
	 * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)
	 */
	protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {

		String input_prog=request.getParameter("input_file");
		String path=("D:/WORKSPACE/eclipse kepler/ProjekatM/WebContent/");
		String file_path=(path+input_prog);
		int count_ins, duzina = 0, L_duzina, i, j, k, ind, indi,tk, tokenDuz, indeks;
		long broj, hash_broj = 0;
		String[] OA, COA, VA, VOA, LA, LOA, cifra;
		String linija, rec, tokeni, vrednost, bin_broj, hex_string, oct_string, b =
null, par;
		String[] token;
		List<String> op, cop; // liste su dinamicki nizovi
		int t3_broj;
		char slovo, bs;
		String cValue, value, ins;
		char lastChar;
		String copt, adrt, out_linija, adr_jump, nula, datoteka_obj;

		OA = new String[20]; // HASH tabela za Operacije i kodove operacija
		COA = new String[20]; // -||-
		VA = new String[100];
		VOA = new String[100];
		LA = new String[100]; // HASH tabela za LABEL-e
		LOA = new String[100]; // -||-
		cifra = new String[100];
		for (indeks = 0; indeks < 20; indeks++)
		{
			OA[indeks] = "";
			COA[indeks] = "";
		}
		for (indeks = 0; indeks < 100; indeks++)
		{
			VA[indeks] = "";
			VOA[indeks] = "";
			LA[indeks] = "";
			LOA[indeks] = "";
		}

		op = new ArrayList<String>();
		cop = new ArrayList<String>();


		/* Inicijalizacija HASH tabele za kodove operacija
```

```
            Tabela sa kodovima operacija je smestena u datoteci COP.TXT
            HASH tabela kodova operacija se pamti u datoteci HASH_COP.TXT */

               // Otvaranje i citanje fajla sa kodovima operacija
               int cnt = 0;

               try
               {
                        BufferedReader cop_file=new BufferedReader(new InputStreamReader(new
FileInputStream(path + "COP.txt")));
                        String[] line = new String[20];

                        while ((line[cnt] = cop_file.readLine()) != null) // citanje do kraja
fajla, linija po linija
                        {
                                op.add(line[cnt].split(",")[0]); // OP je prvi deo linije, do
zareza
                                cop.add(line[cnt].split(",")[1]); // COP je drugi deo linije, od
zareza
                                //        tbCOP.AppendText(op[cnt] + "," + cop[cnt] + "\n");
                                cnt++;
                        }
                        cop_file.close();
               }
               catch (Exception e) {
                        e.printStackTrace();
               }
               // Formiranje HASH tabele i ulaza u nju
                                for(indeks = 0; indeks < op.size(); indeks++)
                                {
                                        duzina = op.get(indeks).length();
                                        for (i = 0; i < duzina; i++)
                                        {
                                                slovo = op.get(indeks).charAt(i); // ekvivalent
Mid funkciji u VB, s[i] je ustvari indekser Chars u String klasi
                                                broj = (int)slovo;
                                                hash_broj = hash_broj + broj * broj * broj % 7777
+ (long)Math.pow(broj, 3);
                                        }
                                        hash_broj = hash_broj % 16;

                                        /* Provera da li je polje u HASH tabeli zauzeto. Ukoliko
jeste, pomera se na sledeci ulaz u tabelu */
                                        k = (int)hash_broj;
                                        ind = 1;
                                        while (ind == 1)
                                        {
                                                if (OA[k].equals(""))
                                                        ind = 0;
                                                else k++;
                                        }
                                        OA[k] = op.get(indeks);
                                        COA[k] = cop.get(indeks);
                                }
                                // Kreiranje .txt fajla sa HASH tabelom

                                FileWriter Writer;
                                Writer cop_hash=new FileWriter(path+"hash_cop.txt");

                                {
                                        for (indeks = 0; indeks < 20; indeks++)

        cop_hash.write(Integer.toString(indeks)+","+OA[indeks]+","+COA[indeks]+"\n");

                                        }
                                cop_hash.close();

                                /* Otvaranje realnog programa i njegovo parsiranje
                                 *
                                 * Prvi zadatak je da se obrade asemblerske komande koje sadrze
EQU direktivu
                                 * i da se po potrebi formira HASH tabela deklarisanih vrednosti
PROMENLJIVIH.
                                 * Iz izvornog koda se izbacuju deklaracione naredbe i ostatak
programa
                                 * prepisuje u privremenu datoteku PROG_TEMP.TXT
                                 */

                                // Citanje fajla sa programom, izracunavanje HASH brojeva za COP
i formiranje HASH tabele
```

```
                        Writer prog_temp=new FileWriter(path+"prog_temp.txt");

                        BufferedReader prog=new BufferedReader(new InputStreamReader(new
FileInputStream(path + input_prog)));
                        while ((linija = prog.readLine()) != null)
                        {
                                duzina = linija.length();


                                /* Odredjivanje tokena - Uvode se 4 tokena
                                 * DIREKTIVE
                                 * - Token 0 - Simbolicko ime adresa/podatak
                                 * - Token 1 - Direktiva EQU
                                 * - Token 2 - Vrednost promenljive
                                 */
                                token = new String[4];
                                rec = "";
                                tk = 0;
                                ind = 0;

                                for (i = 0; i < duzina; i++)
                                {
                                        slovo = linija.charAt(i);
                                        if (slovo != ':' && slovo != ' ' && slovo != ','
&& slovo != ';')
                                        {
                                                rec += slovo;
                                                ind = 0;
                                        }
                                        else if (ind == 0)
                                        {
                                                token[tk] = rec;
                                                rec = "";
                                                tk++;
                                                ind = 1;
                                        }
                                }

                                if (token[1].equals("EQU"))
                                {
                                        // Token je uzet
                                        // Eliminisanje direktive EQU
                                        duzina = token[0].length();
                                        for (i = 0; i < duzina; i++)
                                        {
                                                slovo = token[0].charAt(i);
                                                broj = (int)slovo;
                                                hash_broj = hash_broj + broj * broj *
broj % 7777 + (long)Math.pow(broj, 3);
                                        }
                                        hash_broj = hash_broj % 99;

                                        /* Provera da li je polje u HASH tabeli zauzeto.
Ako jeste, pomera se na sledeci ulaz u tabelu */
                                        k = (int)hash_broj;
                                        ind = 1;
                                        while (ind == 1)
                                        {
                                                if (VA[k].equals(""))
                                                        ind = 0;
                                                else k++;
                                        }

                                        VA[k] = token[0];

                                        /* KONVERZIJA
                                         * Ova konverzija se vrsi u slucajevima kada su
vrednosti koje se dodeljuju
                                         * simbolickim imenima date u razlicitim brojnim
sistemima. Alternativa
                                         * ovom sistemu je da se konverzija vrsi prilikom
uzimanja vrednosti iz
                                         * HASH tabele
                                         */

                                        tokeni = token[2];
                                        tokenDuz = tokeni.length();
```

```
                                        lastChar = tokeni.charAt(tokenDuz - 1);
                                        bin_broj = "";

                                        switch (lastChar)
                                        {
                                        case 'd': // provera da li je vrednost decimalna
                                                vrednost = tokeni.substring(0, tokenDuz -
1);
                                //              int.TryParse(vrednost, out t3_broj); //
izdvajanje cifara iz stringa
                                                t3_broj = Integer.parseInt(vrednost);
                                                b = Integer.toString(t3_broj, 2); //
konverzija u binarni format

                                                    duzina = b.length();
                                                     duzina = 12 - duzina;
                                                     for (i = 0; i < duzina; i++)
                                                         b = '0' + b;

                                        //      b = b.PadLeft(12, '0'); // radi isto sto
i iskomentiran kod
                                                bin_broj = b;
                                                break;
                                        case 'o': // provera da li je vrednost oktalna
                                                vrednost = tokeni.substring(0, tokenDuz -
1);
                                                oct_string = vrednost;
                                                int int_valueO =
Integer.parseInt(oct_string, 8); // decimalna vrednost broja
                                                b = Integer.toString(int_valueO, 2);
                                //              b = b.PadLeft(12, '0');

                                                duzina = b.length();
                                        duzina = 12 - duzina;
                                        for (i = 0; i < duzina; i++)
                                            b = '0' + b;

                                                bin_broj = b;
                                                break;
                                        case 'h': // provera da li je vrednost
heksadecimalna
                                                vrednost = tokeni.substring(0, tokenDuz -
1);
                                                hex_string = vrednost;
                                                int int_valueH =
Integer.parseInt(hex_string, 16); // heksadecimalna vrednost broja
                                                b = Integer.toString(int_valueH, 2);
                                        //      b = b.PadLeft(12, '0');

                                                duzina = b.length();
                                        duzina = 12 - duzina;
                                        for (i = 0; i < duzina; i++)
                                            b = '0' + b;

                                                bin_broj = b;
                                                break;
                                        case 'b': // provera da li je vrednost binarna
                                                vrednost = tokeni.substring(0, tokenDuz -
1);
                                                b = vrednost;

                                                duzina = b.length();
                                        duzina = 12 - duzina;
                                        for (i = 0; i < duzina; i++)
                                            b = '0' + b;

//                                    b = b.PadLeft(12, '0');
                                                bin_broj = b;
                                                break;
                                        } // end switch

                                        VOA[k] = bin_broj;

                                    } // end if

                                    else
                                    {
                                Writer hash_var=new FileWriter(path+"hash_var.txt");
                                        {
                                                for (indeks = 0; indeks < 100; indeks++)
```

```
        hash_var.write(Integer.toString(indeks)+","+VA[indeks] + "," + VOA[indeks]+ "\n");
                                        }
                                        prog_temp.write(linija+"\n");
                                        hash_var.close();
                                } // end else


                        }// end while
                        prog_temp.close();
                        prog.close();



                        /* OBRADA PROG_TMP DATOTEKE */

                        count_ins = 0;

                        Writer prog_objtmp=new FileWriter(path+"prog_objtmp.txt");
                        BufferedReader prog_temp2=new BufferedReader(new
InputStreamReader(new FileInputStream(path + "prog_temp.txt")));

                        token=new String[4];

                        while ((linija = prog_temp2.readLine()) != null){
                                count_ins++;
                                linija = linija.trim();
                                duzina = linija.length();

                                //    MessageBox.Show(linija);

                                /* Odredjivanje tokena - Uvode se 4 tokena

                                 * INSTRUKCIJE
                                 * - Token 0 - Labela ili kod operacije (privremeno
labela, posle se zamenjuje sa kodom operacije
                                 * - Token 1 - Parametar u LI
                                 * - Token 2 - Adresa

                                 */

                                // Inicijalizacija promenljivih u koje se upisuju tokeni

                                token = new String[4];
                                rec = "";
                                tk = 0;
                                ind = 0;
                                //Formiranje tokena

                                for (i = 0; i < duzina; i++)
                                {
                                        slovo = linija.charAt(i);
                                        if (slovo != ':' && slovo != ' ' && slovo != ','
&& slovo != ';')
                                        {
                                                rec = rec + slovo;
                                                ind = 0;
                                        }
                                        else
                                        {
                                                if (ind == 0)
                                                {
                                                        token[tk] = rec;

                                                        // Provera da li je rec o tokenu 0
                                                        if (tk == 0 && slovo == ':') //
Postoji labela, formira se HASH tabela za labele
                                                        {
                                                                L_duzina =
token[0].length();
                                                                for (j = 0; j < L_duzina;
j++)
                                                                {
                                                                        slovo =
token[0].charAt(j);
                                                                        broj = (int)slovo;
                                                                        hash_broj =
hash_broj + broj * broj * broj % 7777 + (long)Math.pow(broj, 3);
                                                                }
```

```
                                                        hash_broj = hash_broj %
99;

                                                        // Kod labela nema
kolizije pa ne treba provera

                                                        k = (int)hash_broj;
                                                        LA[k] = token[0];
                                                        LOA[k] =

Integer.toString(count_ins);
                                                        tk--;
                                                    } // end if

                                                    rec = "";
                                                    tk++;
                                                    ind = 1;
                                                } // end if
                                            } // end else


                                        } // end for

                                                    /* U slucaju da je kod
operacije LI - Load Immediate, postoje tri tokena
                                                     * u instrukciji i
potrebno je formirati kod parametra.
                                                     * Parametar = 0 - upis u
nizi bajt - 0000
                                                     * Parametar = 1 - upis u
visi bajt - 0001
                                                     */

                                                    par = "";
                                                    if (token[0].equals("LI"))
                                                    {
                                                            par = "000" +
token[1];
                                                            //
MessageBox.Show(par + "par");
                                                    }



                                                    // Uzimanje binarne
vrednosti koda operacije iz HASH tabele kodova operacija

                                                    duzina =
token[0].length();
                                                    for (i = 0; i < duzina;
i++)
                                                    {
                                                            slovo =
token[0].charAt(i);
                                                            broj = (int)slovo;
                                                            hash_broj =
hash_broj + broj * broj * broj % 7777 + (long)Math.pow(broj, 3);
                                                    }
                                                    hash_broj = hash_broj %
16;

                                                    /* Provera da li je polje
u HASH tabeli zauzeto. Ukoliko jeste, pomera se na sledeci ulaz u tabeli. */

                                                    k = (int)hash_broj;


                                                    ind = 1;
                                                    while (ind == 1)
                                                    {
                                                            if
(OA[k].equals(token[0]))
                                                            {
                                                                    ind = 0;
                                                            }
                                                            else
                                                            {
                                                                    k++;
                                                                    if (k > 19)
                                                                            k =
0;
```

```
                }
            }
            cValue = COA[k];
            // MessageBox.Show(cValue
+ "cvalue");

            /* Provera da li se radi o
instrukciji LI, jer tada je Token 1 parametar, a Token 2 je
             * vrednost podatka koji
se upisuje u ACC */

            int oznaka = 0; /* Ovaj
indikator sluzi za indikaciju da li se radi o instrukciji LI,
             * jer tada je vrednost u
instrukciji 8-bitna */

            if (token[0].equals("LI"))
            {
                //
MessageBox.Show(token[1] + token[2]);
                token[1] =
token[2];
                //
MessageBox.Show(token[1] + token[2]);
                oznaka = 1;
            }


            // Provera da li je Token
1 - LABELA na koju se skace

            indi = 0;
            for (j = 0; j < 100; j++)
                if
(LA[j].equals(token[1]))
                {
                    //
MessageBox.Show(LA[j] + token[1] + "labela");
                    indi = 1;
                }
            if (indi == 0) // Provera
da li je Token 1 sama vrednost, a ne simbolicko ime
            {
                bs =
token[1].charAt(token[1].length() - 1); // Uzima se krajnji desni karakter Token-a 1
                if (bs == 'd' || bs
== 'b' || bs == 'h' || bs == 'o')
                {
                    tokeni =
token[1];
                    tokenDuz =
tokeni.length();
                    lastChar =
tokeni.charAt(tokenDuz - 1);
                    bin_broj =
"";

                    switch
(lastChar)
                    {
                        case 'd':

// provera da li je vrednost decimalna
                            vrednost = tokeni.substring(0, tokenDuz - 1);
                            t3_broj = Integer.parseInt(vrednost);
                            b =
Integer.toString(t3_broj, 2); // konverzija u binarni format

                            duzina = b.length();
                            duzina = 12 - duzina;
                            for (i = 0; i < duzina; i++)
```

```
b = '0' + b;

        bin_broj = b;

        break;
                                                                    case 'o':
// provera da li je vrednost oktalna

        vrednost = tokeni.substring(0, tokenDuz - 1);

        oct_string = vrednost;
                                                                        int
int_valueO = Integer.parseInt(oct_string, 8); // decimalna vrednost broja
                                                                        b =
Integer.toString(int_valueO, 2);

        duzina = b.length();
                                                            duzina = 12 - duzina;
                                                            for (i = 0; i <
duzina; i++)
                                                                b = '0' + b;

        bin_broj = b;

        break;
                                                                    case 'h':
// provera da li je vrednost heksadecimalna

        vrednost = tokeni.substring(0, tokenDuz - 1);

        hex_string = vrednost;
                                                                        int
int_valueH = Integer.parseInt(hex_string, 16); // heksadecimalna vrednost broja
                                                                        b =
Integer.toString(int_valueH, 2);

        duzina = b.length();
                                                            duzina = 12 - duzina;
                                                            for (i = 0; i <
duzina; i++)
                                                                b = '0' + b;

        bin_broj = b;

        break;
                                                                    case 'b':
// provera da li je vrednost binarna

        vrednost = tokeni.substring(0, tokenDuz - 1);
                                                                        b =
vrednost;

        duzina = b.length();
                                                            duzina = 12 - duzina;
                                                            for (i = 0; i <
duzina; i++)
                                                                b = '0' + b;


        bin_broj = b;

        break;
                                                                    } // end
switch
                                                                    value =
bin_broj;
                                                                } // end if
                                                                else
                                                                {
                                                                    duzina =
token[1].length();
```

```
                                                                for (i = 0;
i < duzina; i++)
                                                                {

        slovo = token[1].charAt(i);
                                                                        broj
= (int)slovo;
        hash_broj = hash_broj + broj * broj * broj % 7777 + (long)Math.pow(broj, 3);
                                                                }
                                                                hash_broj =
hash_broj % 99;

                                                                /* Provera
da li je polje u HASH tabeli zauzeto. Ukoliko jeste, pomera se na sledeci ulaz u tabeli. */

                                                                k =
(int)hash_broj;
                                                                ind = 1;
                                                                while (ind
== 1)
                                                                {

                                                                        if
(VA[k].equals(token[1]))
                                                                        {
        // MessageBox.Show(VA[k] + token[1] + "VA");
        ind = 0;


                                                                        }
                                                                        else
                                                                        {
        k++;
        if (k > 99)
                k = 0;


                                                                        }
                                                                }
                                                                value =
VOA[k];
                                                        } // end else
                                                } // end if

                                                else // indi == 1
                                                {
                                                        value = token[1];
                                                }

                                                if (token[0].equals("LI"))
                                                {
                                                        value =
value.substring(value.length() - 8);

                                                        ins = cValue + par
+ value;


                                                }
                                                else
                                                {
                                                        ins = cValue +
value;
                                                }

        prog_objtmp.write(ins+"\n");


                        }
                        prog_objtmp.close();


                        /* Razresavanje LABELA - formiranje konacne OBJEKTNE datoteke */
```

```
                                duzina = file_path.length();
                                datoteka_obj = file_path.substring(0, duzina - 4);
                                datoteka_obj = datoteka_obj + ".obj";

                                Writer obj_dat=new FileWriter(path+"datoteka_obj.txt");
                                BufferedReader prog_objtmp2=new BufferedReader(new
InputStreamReader(new FileInputStream(path + "prog_objtmp.txt")));
                                while ((linija = prog_objtmp2.readLine()) != null)
                                {
                                duzina = linija.length();
                    L_duzina = duzina - 4;
                    copt = linija.substring(0, 4); // kod operacije
                    adrt = linija.substring(L_duzina); // naziv labele
                    out_linija = "";

                    if (L_duzina < 12)
                    {
                        for (i = 0; i < 100; i++)
                        {
                            if (adrt.equals(LA[i]))
                            {
                                adr_jump = LOA[i];


                                t3_broj=Integer.parseInt(adr_jump);
                                nula=Integer.toString(t3_broj,2);
                               out_linija = copt + b;
                            }
                        }
                    }
                    else
                        out_linija = linija;

                    obj_dat.write(out_linija+"\n");


                    }// end while

                           obj_dat.close();
                           prog_objtmp2.close();


FileReader fr= new FileReader("D:/WORKSPACE/eclipse kepler/ProjekatM/WebContent/"+input_prog);
BufferedReader myInput = new BufferedReader(fr);
String str;
StringBuffer b2 = new StringBuffer();
while ((str = myInput.readLine()) != null) {
b2.append(str);
b2.append("\n");

}
String Input_files = b2.toString();
myInput.close();
request.setAttribute("Input_file", Input_files);

FileReader fr2= new FileReader("D:/WORKSPACE/eclipse
kepler/ProjekatM/WebContent/datoteka_obj.txt");
BufferedReader myOutput = new BufferedReader(fr2);
String str2;
StringBuffer b3 = new StringBuffer();
while ((str2 = myOutput.readLine()) != null) {
b3.append(str2);
b3.append("\n");

}
String Output_files = b3.toString();
myOutput.close();
request.setAttribute("Output_file", Output_files);

FileReader fr3= new FileReader("D:/WORKSPACE/eclipse kepler/ProjekatM/WebContent/COP.txt");
BufferedReader Cop = new BufferedReader(fr3);
String str3;
StringBuffer b4 = new StringBuffer();
while ((str3 = Cop.readLine()) != null) {
b4.append(str3);
b4.append("\n");

}
String Cop_file = b4.toString();
```

```
request.setAttribute("Cop_file", Cop_file);
Cop.close();


request.getRequestDispatcher("pocetna_strana.jsp").forward(request, response);


	}

	/**
	 * @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse response)
	 */
	protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
		// TODO Auto-generated method stub
	}

}
```

Следећи део програмског кода реализује динамичку WEB страницу преко које се може пратити рад асемблера.

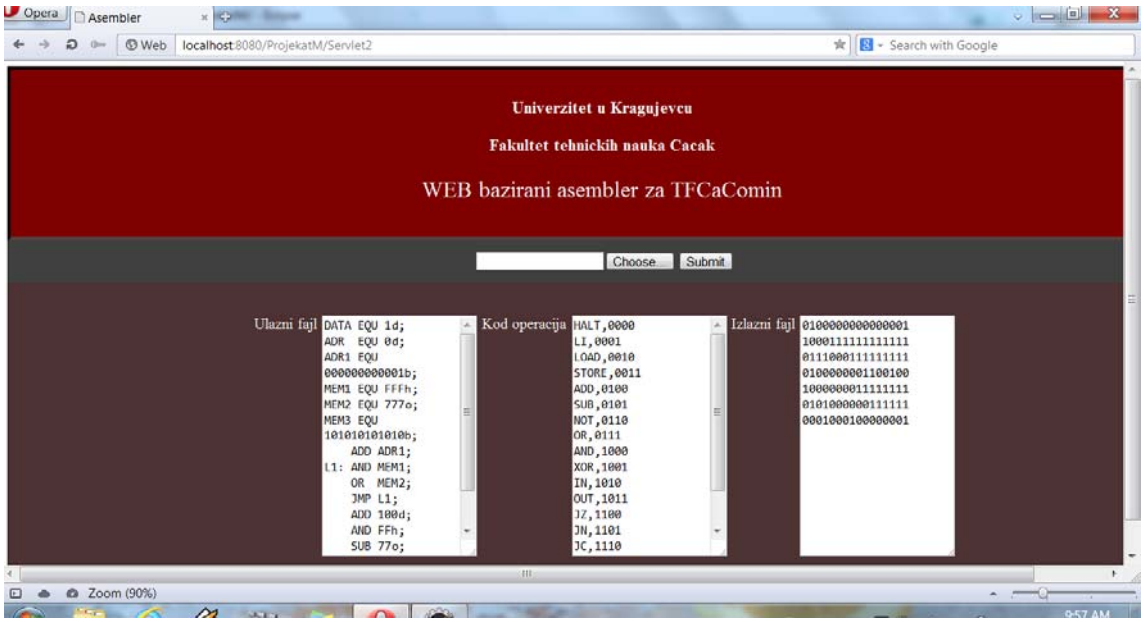
```
<%@page import="java.io.BufferedReader"%>
<%@page import="java.io.FileReader"%>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<link rel="stylesheet" type="text/css" href="Style.css"/>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Asembler</title>
</head>
<body>
<%
String Input_file = (String) request.getAttribute("Input_file");
String Output_file=(String)request.getAttribute("Output_file");
String Cop_file=(String)request.getAttribute("Cop_file");

%>

<div class="zaglavlje">
<h3>Univerzitet u Kragujevcu</h3>
<h3>Fakultet tehnickih nauka Cacak</h3>
<p>WEB bazirani asembler za TFCaComin</p>
</div>
<div class="input_meny">
<form action="Servlet2" method="get">
<input type="file" name="input_file">
<input type="submit">
</form>
</div>
<div class="main_div">
<label for="input">Ulazni fajl</label>
<textarea rows="15" cols="20" name="input">
<%=Input_file%>
</textarea>
<label for="cop">Kod operacija</label>
<textarea rows="15" cols="20" name="input">
<%=Cop_file%>
</textarea>
<label for="output">Izlazni fajl</label>
<textarea rows="15" cols="20" name="output">
<%=Output_file%>
</textarea>


</body>
</html>
```

На слици 5.6 дат је изглед приказа резултата рада апликације у WEB browser – у.



Слика 5.6 Изглед излаза рада апликације

6. ЗАКЉУЧАК

Развој савремених рачунарских технологија довео је ширења примене рачунара на скоро све аспекте људсе делатности. С обзиром да је рачунар комплексна машина, и пошто се састоји из пуно различитих модула, мора се наћи начин да се све те целине обједине у јединствен систем. Пре свега мора да постоји веза између његовог хардвера и софтвера, а да би се то успешно обавило, рачунар мора добијати све информације у бинарном облику или како се уобичајено каже у машинском језику. Због тога је сам процес превођења других облика кодовања информација у бинарни облик један од најважнијих процеса неопходних за исправан рад рачунарског система. Без њега не би могла да се успостави веза између рачунара, као машине, и програма и процеса који требају на њему да се извршавају.

У оквиру овог дипломског рада су објашњене намене и функције једног од најважнијих рачунарских програма – асемблера који је намењен за превођење симболички написаног машинског програма у његов бинарни еквиваленат. Такође је приказан поступак пројектовања и реализације асемблера за архитектуру процесора TFaComin који је пројектован у Лабораторији за рачунарску технику Факултета техничких наука. Да би се добио потпуни увид у начин функционисања асемблера и процеса асемблирања објашњени су сви програмски модули из којих се асемблер састоји. Објашњено је како сви ти модули раде заједно као једна целина, и како се на крају добија објектни код, који је крајњи резултат процеса асемблирања. Пројектовани и реализовани асемблер за дати 16 – битни процесор је потпуно функционалан и као резултат даје објектни машински програмски код који се може извршити на датој процесорској архитектури. Првенствена намена овог асемблера је да генерише објектни код програма процесора TFaComin на одговарајућем симулатору, који је такође развијен на Факултету техничких наука.

Функција добијеног асемблера је уско повезана са процесорском архитектуром за коју је и развијана. Добијени програм за асемблирање је предуслов практичног коришћења процесора TFaComin. Без њега сам процесор би био практично бескористан део хардвера. Асемблер омогућава процесору да „разуме” симболички програмски код, и на тај начин омогућава процесору да обавља обраду података. Захваљујући овом

пројекту сада је могуће писати програме на симболичком машинском језику за дату архитектуру, који ће се превести на машински језик и на тај начин ће се добити објектни програм који се може извршавати.

Искуства стечена кроз овај дипломски рад се огледају кроз више аспеката. За успешно обављање задатака и проблема који су били постављени код при дефинисању захтева за реализацију асемблера, било је потребно поседовати доста знања из разних области рачунарства, као што су архитектура рачунара, машинско програмирање, основе машинског језика, програмирање на неком вишем програмском језику. Значајан аспект реализације овог дипломског рада представља избор алата за његову реализацију. Полазећи од искустава која су постојала у реализацији сличних програмских решења, а код који су коришћени програмски језици Visual Basic и C# одлучено је да се користи JAVA развојно окружење. Овај избор је био двоструко мотивисан: провера могућности JAVA програмског језика у реализацији обраде података, карактеристичне за рад асемблера и добијање апликације која ће бити доступна корисницима у облику Интернет апликације. Први задатак је успешно остварен, тј. утврђено је да JAVA програмски језик даје пуну подршку обради података, каква се захтева на нивоу асемблера. Пре свега у обради текстуалних информација. Такође, реализована је WEB визуелизација рада асемблера.

Нажалост, због ограниченог времена које је стајало за реализацију овог рада други део циља није остварен. Међутим, имајући у виду добру подршку, коју JAVA пружа у визуелизацији информација путем Интернета, овај део задатка је значајно лакши и извеснији за реализацију од основног задатка.

Такође, било је потребно детаљно ући у начин функционисања разматране процесорске архитектуре да би се уопште могло приступити пројектовању асемблера, а касније и његовој реализацији. Сходно томе треба нагласити значај оваквих сложених пројеката, а посебно њихову практичну реализацију за уобличавање знања која су важна за размевање приступа у повлачењу „границе” између хардвера и софтвера рачунара на нивоу дефинисања инструкцијске архитектуре рачунара. Такође, овај рад је био прилика и за сагледавање недостатака архитектуре процесора TFaComin и потребе да се кроз неке наредне студентске пројекте, који би се реализовали у Лабораторији за рачунарску технику, на тим искуствима покушало са дефинисањем неке нове архитектуре, која би уочене недостатке отклонила.

ЛИТЕРАТУРА

[1] John Donovan – *Systems programming,* McGraw-Hill, Inc. New York, NY, USA, 1972.

[2] Kyle Loudon – *Mastering Algorhytms with C,* O' Reilly & Associates, Inc, USA 1999.

[3] Paul A. Carter – *PC Assembly Language,* Self Edition (PDF Format), July 23, 2006

[4] Shimon Schocken – *Machine Language*, Harvard University, 2005

[5] Blum, R., "*Professional Assembly Language*", Weox, 2005

[6] Dandamudi, S., "*Guide to RISC processors: for Programmers and Engineers*", Springer, 2005

[7] Duntemann, J., „*Assembly Language Step – by – Step: Programming with Linux*"3rd edition, Wiley, 2009

Коришћени WEB ресурси

[8] http://searchdatacenter.techtarget.com/definition/assembler, Assembler

[9] www.wikipedia.org, Online enciklopedija

[10] www.osdata.com, OSdata