



## CS120 - ORGANIZACIJA RAČUNARA

Sloj asemblerских језика

Lekcija 11

PRIRUČNIK ZA STUDENTE

# CS120 - ORGANIZACIJA RAČUNARA

## Lekcija 11

### *SLOJ ASEMBLERSKIH JEZIKA*

- ✓ Sloj asemblerских језика
- ✓ Poglavlje 1: NIVO ASEMBLERA
- ✓ Poglavlje 2: FORMATI NAREDBI ASEMBLERA
- ✓ Poglavlje 3: PSEUDOINSTRUKCIJE
- ✓ Poglavlje 4: MAKRO KOMANDE U ASEMBLERSKIM JEZICIMA
- ✓ Poglavlje 5: NASM ASEMBLER
- ✓ Poglavlje 6: Pokazne vežbe
- ✓ Poglavlje 7: Zadaci sa samostalni rad
- ✓ Poglavlje 8: Domaći zadatak
- ✓ Ulaz/Izlaz- Opšta razmatranja

Copyright © 2017 – UNIVERZITET METROPOLITAN, Beograd. Sva prava zadržana. Bez prethodne pismene dozvole od strane Univerziteta METROPOLITAN zabranjena je reprodukcija, transfer, distribucija ili memorisanje nekog dela ili čitavih sadržaja ovog dokumenta., kopiranjem, snimanjem, elektronskim putem, skeniranjem ili na bilo koji drugi način.

Copyright © 2017 BELGRADE METROPOLITAN UNIVERSITY. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, without the prior written permission of Belgrade Metropolitan University.

## ❖ Uvod

### UVOD

*Procesor obrađuje i izvršava mašinski kod (binarni) koji mu govori šta da procesor radi. Jedini razumljivi jezik procesoru je asemblerski jezik.*

U ovoj lekciji ćemo razmatrati potrebu za uvođenjem asemblerskih jezika. generalni pristup organizacije procesora po kome se procesor sastoji iz dve jedinice i to operacione jedinice i upravljačke jedinice.

Nivo asemblerskog jezika značajno se razlikuje od nivoa mikroarhitekture, nivoa ISA i nivoa operativnog sistema računara - on se ne implementira interpretiranjem, već prevođenjem.

Programi koji pretvaraju korisnički program pisan na nekom jeziku u program na nekom drugom jeziku zovu se **prevodioci** (engl. **translators**).

Jezik na kome je originalni program napisan zove se **izvorni jezik** (engl. **source language**), a jezik u koji se on pretvara zove se **ciljni jezik** (engl. **target language**). Izvorni jezik i ciljni jezik definišu dva nivoa. Ako postoji procesor koji može direktno da izvršava program pisan na izvornom jeziku onda nema potrebe da se izvorni program prevodi na ciljni jezik. Prevođenje se koristi kada postoji procesor (kao hardverska komponenta ili kao interpreter) za ciljni jezik, ali ne i za izvorni. Ako se prevođenje izvede ispravno, kada se izvrši prevedeni program dobiće se potpuno isti rezultati kao da je program izvršavan na procesoru koji razume izvorni jezik. Shodno tome, nov nivo za koji ne postoji procesor može se implementirati tako što će se programi pisani za taj nivo prevesti na ciljni jezik a zatim takvi programi na cilnjom jeziku izvršiti.

Treba istaći razliku između prevođenja i interpretiranja. Pri prevođenju se originalni program napisan na izvornom jeziku ne izvršava direktno. On se najpre pretvara u ekvivalentan program koga zovemo **objektniprogram** (engl. **object program**) ili **izvršni binarni program** (engl. **executable binary program**), koji se izvršava tek kada je prevođenje završeno

Nivo asemblera sadrži mašinski jezik iz nižih nivoa preveden u skup tekstualnih simbola. Ovaj nivo pruža ljudima mogućnost da pišu programe za nivoe 1, 2 i 3 u obliku koji nije tako neprijatan kao jezici virtuelnih mašina. Programi napisani u asembleru najpre se prevode na jezik nivoa 1, 2 ili 3, a zatim ih interpretira odgovarajuća virtuelna mašina ili stvarni računar. Program koji obavlja prevođenje zove se **asembler** (engl. **assembler**).

## ▼ Poglavlje 1

### NIVO ASEMBLERA

#### SLOJEVITA STRUKTURA RAČUNARSKOG SISTEMA

*Digitalni računar je mašina koja ljudima pomaže tako što izvršava zadate instrukcije. Niz instrukcija čijim se izvršavanjem obavlja određeni posao naziva se program.*

Elektronska kola računara mogu prepoznati i direktno izvršiti samo ograničen skup jednostavnih instrukcija, pa svaki program, da bi bio izvršen, treba prevesti u takav skup instrukcija. Osnovne instrukcije retko su složenije od sledećih:

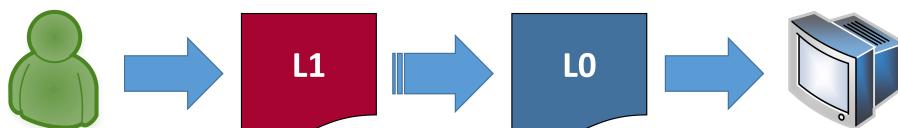
- Saberi dva broja.
- Proveri da li je zadati broj nula.
- Kopiraj skup podataka iz jednog dela memorije računara u drugi.

Sve osnovne instrukcije koje računar razume, obrazuju jezik pomoću koga ljudi komuniciraju s računarom. To je **mašinski jezik**.

**Pošto je mašinski jezik računara izuzetno jednostavan (za računare), ljudi se njime služe teško. Velika je razlika između onoga što odgovara ljudima i onoga što odgovara računaru. Ljudi bi želeli da urade neko X, ali računari mogu da urade samo Y.**

Problemu se može prići na dva načina, a oba zahtevaju da se stvori nov skup instrukcija koji ljudima više odgovara od skupa ugrađenih mašinskih instrukcija.

Skup novih instrukcija takođe formira jedan jezik koji ćemo zvati L1, za razliku od mašinskog jezika L0 koji razume računar.



Slika 1.1 Skup novih instrukcija formira jedan jezik L1 i mašinski jezik L0 koji razume računar [Izvor: Autor]

#### KOMPAJLER I INTERPRETER

*Postoje dva načina za izvršavanje programa pisanih na jeziku L1*

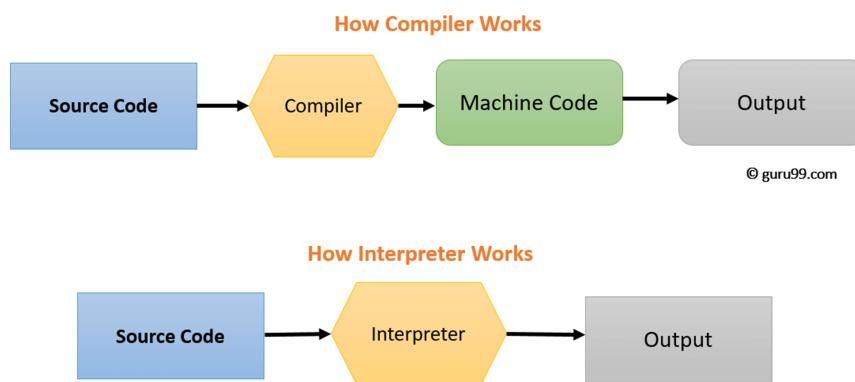
Skup novih instrukcija takođe formira jedan jezik koji ćemo zvati L1, za razliku od mašinskog jezika L0 koji razume računar.

**Prvi način izvršavanja programa** pisanih na jeziku L1 jeste da se najpre svaka njegova instrukcija zameni ekvivalentnim nizom instrukcija na jeziku L0. Rezultujući program sastoji se isključivo od instrukcija na jeziku L0.

Računar tada izvršava nov program na jeziku L0 umesto starog programa na jeziku L1. Ova tehnika se naziva **prevodenje** (en. **translation**). Program koji prevodi sa jezika višeg nivoa na mašinski jezik naziva se kompjajler (en. **compiler**).

**Drugi način izvršavanja** jeste da program pisan na jeziku L1 učitava instrukciju po instrukciju, nakon čega se svaka instrukcija redom ispituje i odmah izvršava njoj ekvivalentan skup instrukcija na jeziku L0.

Na taj način se izbegava prethodno generisanje čitavog novog programa na jeziku L0. Tehnika se zove **interpretiranje** (en. **interpretation**). Program koji je koristi naziva se interpreter (en. **interpreter**).



Slika 1.2 Razlika između prevođenja i interpretiranja [Izvor: Autor]

## SLOJEVITA STRUKTURA RAČUNARSKOG SISTEMA,

*Da bi prevođenje i interpretiranje u praksi bilo efikasno, jezici L0 i L1 ne smeju se međusobno previše razlikovati.*

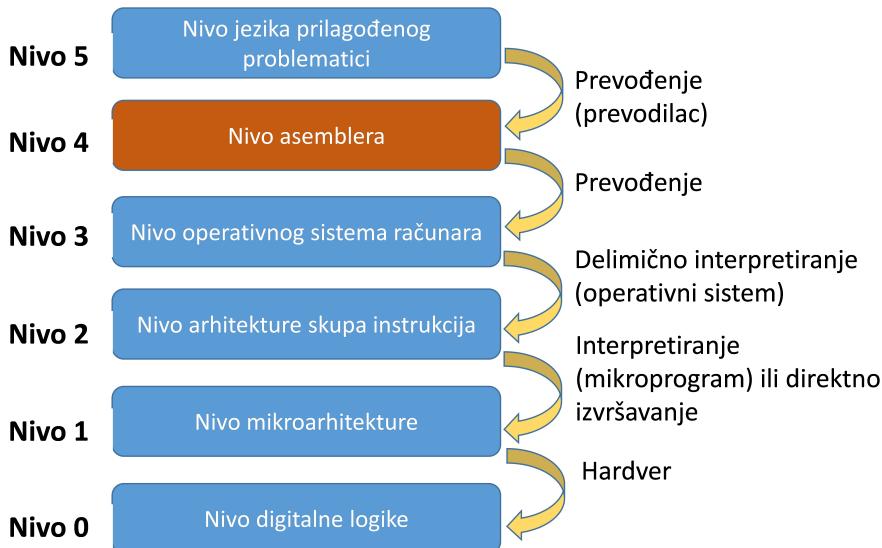
**Savremeni računari se sastoje od dva ili više slojeva .**

Računar sa n slojeva može se u izvesnom smislu posmatrati kao niz od n različitih virtuelnih mašina, od kojih svaka "govori" drugaćijim mašinskim jezikom.

Počevši od sloja 4, programski jezik se sastoji od reči i tekstualnih skraćenica koje za ljudе imaju određeno značenje.

Sloj 4 (**sloj asemblera**) sadrži mašinski jezik iz nižih nivoa preveden u skup tekstualnih simbola.

**Ovaj nivo pruža ljudima mogućnost da pišu programe za nivoe 1, 2 i 3 u obliku koji nije tako neprijatan.**



Slika 1.3 Nivo asemblera u slojevitoj strukturi računarskog sistema [Izvor: Autor]

## UVOD U ASEMBLERSKI JEZIK,

*Programi napisani u asembleru najpre se prevode na jezik nivoa 1, 2 ili 3, a zatim ih interpretira odgovarajuća virtualna mašina ili stvarni računar.*

**Programske prevodioce možemo grubo svrstati u dve grupe, prema međusobnom odnosu izvornog i ciljnog jezika.**

Kada je izvorni jezik u suštini simbolička predstava numeričkog mašinskog jezika, prevodilac se naziva **assembler** (engl. **assembler**), a izvorni jezik se naziva **asemblerški jezik** (engl. **assembly language**).

Kada se kao izvorni jezik koristi programski jezik visokog nivoa, kao što su Java ili C, a ciljni jezik je ili numerički mašinski jezik ili njegova simbolička predstava prevodilac se naziva **kompajler** (engl. **compiler**).

U asemblerškom jeziku, svaka naredba proizvodi tacno jednu mašinsku instrukciju.

Drugim rečima, između mašinskih instrukcija i naredbi u asemblerškom programu postoji preslikavanje jedan na jedan.

Ako svaki red u programu napisanom na asemblerškom jeziku sadrži samo jednu naredbu a svaka mašinska reč sadrži samo jednu mašinsku instrukciju, onda će naredni asemblerški program proizvesti mašinski program od II reči.

**Programi se pišu na asemblerškom jeziku a ne na mašinskom (u obliku heksadecimalnih brojeva) zato što je na asemblerškom jeziku mnogo lakše programirati.**

## UVOD U ASEMBLERSKI JEZIK

*Programske prevodioce možemo grubo svrstati u dve grupe, prema međusobnom odnosu izvornog i ciljnog jezika.*

Većina programera može da zapamti da su **ADD**, **SUB**, **MUL** i **DIV** skraćenice za sabiranje, oduzimanje, množenje i deljenje, ali malo ko pamti ekvivalentne brojčanih vrednosti mašinskog jezika.

**Programer na asemblerском jeziku treba samo da pamti simbolička imena, a asembler ih automatski prevodi u mašinske instrukcije**

Osim svojstva preslikavanja jedan na jedan u mašinske instrukcije, asembleri imaju još jedno svojstvo koje ih jasno razlikuje od programskega jezika visokog nivoa.

Programerima na asemblerском jeziku dostupne su sve mogućnosti i instrukcije ciljnog računara, sto nije slučaj s programerima na jezicima visokog nivoa.

Asembleri program može da izvrši svaku instrukciju iz skupa instrukcija ciljnog računara, što program napisan na jeziku visokog nivoa ne može. Ukratko, sve što se može uraditi na mašinskom jeziku može se uraditi i na asembleru, dok mnoge instrukcije, registri i slične mogućnosti nisu dostupni programeru na jeziku visokog nivoa.

**Takođe program napisan na asembleru može da se izvršava samo na jednoj porodici računara, dok program napisan na jeziku visokog nivoa potencijalno može da se izvršava na više različitih računara.**

## OSNOVE ASEMLBLERA

*Asembleri programi se često izvršavaju brže i zauzimaju manje memorije u odnosu na programe napisane u višim programskim jezicima.*

Assembler na simbolički način predstavlja binarni mašinski jezik određene hardverske platforme.

Simbolički nazivi se mogu dodeliti imenima instrukcija, memorijskim lokacijama, registrima, korisnički definisanim procedurama, itd.

Pored toga, asembler programu daje potpun pristup hardveru.

**Asembleri programi se često izvršavaju brže i zauzimaju manje memorije u odnosu na programe napisane u višim programskim jezicima.**

Uloga asemblera je da sve simboličke adrese u programu zameni numeričkim, simboličke kodove operacija mašinskim kodovima, rezerviše mesto za instrukcije i podatke u memoriji, predstavi konstante u mašinskom obliku, itd.

Rezultat rada asemblera je odgovarajući mašinski kod. Ukoliko ima potrebe, dobijeni kod

treba povezati sa bibliotekama ili drugim programima pomoću posebnog programa koji se naziva **povezivač** (**linker**).

Ovako dobijeni izvršni kod se pomoću **punjača** (**loader**) smešta u memoriju i zatim se aktivira njegovo izvršavanje. Opisani postupak je prikazan na slici.



Slika 1.4 Proces pripreme asemblerskog programa za izvršenje [Izvor: Autor]

## PREDNOSTI ASEMLBLERA

*Prednost programa napisanih u assembler-u je u mogućnosti slanja direktnih komandi procesoru kao i iskorišćavanju celog dijapazona računarske arhitekture.*

**Programi napisani u assembler-u se odlikuju se mogućnošću slanja direktnih komandi procesoru kao i iskorišćavanju celog dijapazona računarske arhitekture.**

Pošto ti programi rade praktično na nivou mašinskog koda, i sa sobom nemaju pomoćne konstrukcije, generalizacije koda i za mašinu slične "nebitne" stvari, su mnogo manji i brži od programa napisanih u nekom "konvencionalnom" programskom jeziku.

Neke od glavnih mana takvih programa su loša čitljivost, (posebno izraženo pri velikim projektima), te složenost koda i praktično nemogućnost konvertiranja istog koda na drugu procesorsku arhitekturu.

**Zbog tih mana se assembler danas koristi samo u sistemima realnog vremena i ostalim specifičnim sistemima.**

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ✓ Poglavlje 2

# FORMATI NAREDBI ASEMLBLERA

## FORMAT NAREDBI ASEMLBLERSKIH JEZIKA

*Precizna sintaksa asemblerskih naredbi zavisi od hardverske platforme.*

Zbog najveće zastupljenosti PC platforme, mi ćemo obratiti pažnju na **Microsoft MASM asembler** za Intelove procesore od 8086/ 8088 do Pentium-a.

Primer asemblerskog programa koji obavlja aritmetičku operaciju  $N = I + J$  je dat na slici 1.

Labela	Opkod	Operandi	Komentari
FORMULA:	MOV	EAX,I	; register EAX = I
	ADD	EAX,J	; register EAX = I + J
	MOV	N,EAX	; N = I + J
I	DD	3	; reserve 4 bytes initialized to 3
J	DD	4	; reserve 4 bytes initialized to 4
N	DD	0	; reserve 4 bytes initialized to 0

Slika 2.1 Izračunavanje izraza  $N=I+J$  na x86 [Izvor: Autor]

Oznake, koje se koriste za obezbeđivanje simboličkih imena za memorijske adrese, potrebne su na izvršnim naredbama kako bi se izrazi mogli granati.

**Pored toga, potrebne su za reći podataka kako bi omogućile da podaci koji se tamo čuvaju budu dostupni pomoću simboličkog imena.**

## FORMAT NAREDBI ASEMLBLERSKIH JEZIKA - INSTRUKCIJE

*Asemblerska instrukcija ssastavljene iz četiri dela.*

Asemblerska instrukcija je u opštem slučaju sačinjena od četiri dela:

1. **labele**
2. **naredbe**
3. **poljaza operande**
4. **polja za komentare**

**Labele** su neophodne da bi se unutar programa mogli realizovati skokovi na određene memorijske lokacije. Te memorijske lokacije su simbolički opisane labelama. Pored toga, labele omogućuju da se podacima dodeli simboličko ime.

**Naredbe** predstavljaju ili simboličke nazine mašinskih instrukcija ili pseudo instrukcije, koje predstavljaju naredbe samom asembleru.

**Operandi** su elementi nad kojima se izvršava instrukcija. Oni mogu biti memorijske adrese, registri, konstante, itd. Dužine operanada su različite.

**U polje za komentare** programeri upisuju objašnjenja o radu programa, korisna drugim programerima, koji će možda kasnije upotrebljavati i podešavati program (ili mogu koristiti i samom autoru nakon izvesnog vremena).

Format naredbi asemblerskih jezika

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ✓ Poglavlje 3

# PSEUDOINSTRUKCIJE

## PSEUDOINSTRUKCIJE

*Pored simboličkog zadavanja instrukcija koje računar treba da izvrši, program na asemblerском jeziku može da sadrži i komande samom asembleru.*

Komande asembleru zovu se **pseudoinstrukcije** ili ponekad direktive asembleru. Tipične pseudoinstrukcije navedene su na slici. One su uzete iz Microsoftovog asemblera **MASM** za grupe Intelovih procesora.

Pseudoinstrukcija	Značenje
SEGMENT	Započinje nov segment (kod, podaci itd.) sa zadatim atributima
ENDS	Završava tekući segment
ALIGN	Upravlja poravnanjem sledeće instrukcije ili sledećih podataka
EQU	Definiše nov simbol jednak zadatom izrazu
DB	Dodeljuje smeštajni prostor za jedan ili više (inicijalizovanih) bajtova
DW	Dodeljuje smeštajni prostor za jednu ili više (inicijalizovanih) 16-bitnih (reči) podataka
DD	Dodeljuje smeštajni prostor za jednu ili više (inicijalizovanih) 32-bitnih (dvostrukih reči) podataka
DO	Dodeljuje smeštajni prostor za jednu ili više (inicijalizovanih) 64-bitnih (četvorostrukih reči) podataka
PROC	Započinje proceduru
ENDP	Završava proceduru
MACRO	Počinje definiciju makroa
ENDM	Završava definiciju makroa
PUBLIC	Izvozi ime definisano u ovom modulu
EXTERN	Uvozi ime iz drugog modula
INCLUDE	Preuzima i uključuje drugu datoteku
IF	Započinje uslovno asembleriranje zasnovano na zadatom izrazu
ELSE	Započinje uslovno asembleriranje ako gornji uslov IF nije ispunjen
ENDIF	Završava uslovno asembleriranje
COMMENT	Definiše nov znak za početak komentara
PAGE	Generiše prelom strane u listingu
END	Završava asembler

Slika 3.1 Asemblerске pseudoinstrukcije [Izvor: Autor]

Zbog ograničenja dužine registara u 8088 procesoru, asemblerски programi se dele na **segmente**.

Pseudo instrukcija **SEGMENT** označava početak segmenta.

**Na ranim Intelovim procesorima se u isto vreme moglo ukazati na četiri segmenta:**

1. **segment koda** (**code segment**)
2. **segment podataka** (**data segment**)

3. **segment steka** (stack segment)
4. **dodatni (extra) segment** (extra segment)

## PSEUDOINSTRUKCIJA PRIMERI #1,

*U asembleru postoje brojni skupovi pseudoinstrukcija koji zavise više od autora nego od arhitekture procesora*

Pseudoinstrukcija **ALIGN** omogućava da se sledeći red, obično sa podacima, veže za adresu koja je umnožak njenog argumenta.

Na primer, ako tekući segment već ima 61 bajt podataka, onda će posle pseudoinstrukcije **ALIGN 4** sledeća dodeljena adresa biti 64.

**EQU** se koristi za davanje simboličkog imena određenom izrazu.

Na primer, posle pseudoinstrukcije **BASE EQU 1000** simbol **BASE** može se svuda koristiti umesto vrednosti 1000. Izraz koji sledi iza **EQU** može da sadrži više definisanih simbola povezanih aritmetičkim i drugim operatorima, kao u izrazu:

**LIMIT EQU 4 \* BASE+ 2000**

Za većinu asemblera, uključujući i MASM, važi pravilo da se simbol definiše pre nego sto se upotrebni u izrazima sličnim prethodnom.

Sledeće pseudoinstrukcije, **DB**, **DW**, **DD** i **DO**, rezervišu smeštajni prostor za jednu ili više promenljivih, veličine 1, 2, 4 ili 8 bajtova.

Na primer, pseudoinstrukcija

**TABLE DB 11, 23, 49**

dodeljuje prostor za 3 bajta i inicijalizuje ih na vrednosti 11, 23 i 49. Ona definiše i simbol **TABLE** izjednačujući ga sa adresom na kojoj je smeštena vrednost II.

Pseudoinstrukcije **PROC** i **ENDP** definišu početak i kraj procedura pisanih na asemblerском jeziku.

Procedure na asemblerском jeziku imaju istu ulogu kao i procedure pisane na drugim programskim jezicima.

Pseudoinstrukcije **MACRO** i **ENDM** u okviru definiciju makroa. O makroima će biti reč kasnije.

## PSEUDOINSTRUKCIJA PRIMERI #2,

*U asembleru postoje brojni skupovi pseudoinstrukcija.*

Pseudoinstrukcije, **PUBLIC** i **EXTERN**, određuju hoće li simbol biti vidljiv. Obično se pišu programi koji su u stvari zbirka datoteka.

U tom slučaju, procedura iz jedne datoteke često treba da pozove proceduru iii da pristupi podacima iz druge datoteke. Da bi se omogućilo ovakvo unakrsno referenciranje simbol koji treba da je raspoloživ drugim datotekama izvozi se pseudoinstrukcijom **PUBLIC**.

Slično tome, da se asembler ne bi bunio što se koristi simbol koji nije definisan u tekućoj datoteci taj simbol se može deklarisati kao spoljni- **EXTERN** - na osnovu čega će asembler znati da je on definisan u drugoj datoteci.

**Simboli koji nisu deklarisani jednom od ove dve pseudo-instrukcije smatraju se lokalnim za tekuću datoteku.**

Pseudoinstrukcija **INCLUDE** nalaze asembleru da preuzme drugu datoteku i daje fizički uključi u tekuću datoteku.

**Takve uključene datoteke obično sadrže svoje definicije, makroe i druge stavke.**

## PSEUDOINSTRUKCIJE PRIMERI 3#,

*Svaki od ovih segmenata ima pridružen pokazivački registar*

Svaki od ovih segmenata ima pridružen pokazivački registar (**CS**, **DS**, **SS** i **ES** respektivno) koji pokazuje na početak segmenta.

Svaki segment završava pseudo instrukcijom **ENDS**.

Primer definisanja segmenta koda i segmenta podataka dat je na slici 2.

```
WORDSIZE EQU 32
IF WORDSIZE GT 32
  WSIZE: DD 64
ELSE
  WSIZE: DD 32
ENDIF
```

Slika 3.2 Primeri uslovnog asembliranja [Izvor: Autor]

Prikazan kod dodeljuje 32-bitnu rec na adresi simboličkog imena **WSIZE**.

Rec se inicijalizuje na 32 ili na 16 bitova, zavisno od vrednosti **WORDSIZE** koja je u ovom slučaju 16.

Stavljući deo koda koji zavisi od računara unutar **IF iENDIF**, i zadajući vrednost **WORDSIZE**, omogucavamo programu da izvrsi asembliranje za jedan ili za drugi računar.

Na ovaj način je moguće održavati sarno jedan izvorni program za više (različitih) ciljnih računara što olakšava razvijanje i održavanje softvera.

U mnogim slučajevima, sve definicije koje zavise od računara, kao **WORDSIZE**, sakupljaju se u jednu datoteku koja postoji u različitim verzijama za različite računare. Kada uključimo odgovarajuću datoteku s definicijama, program možemo lako prevesti za bilo koji računar.

Pseudo instrukcija **EQU** definiše novi simbol jednak izrazu navedenom na desnoj stani instrukcije. Na primer:

**DESET EQU 10**

**STO EQU 10\*DESET**

definiše simbole **DESET** i **STO** čije su vrednosti **10** i **100**.

**PUBLIC** i **EXTERN** definišu vidljivost simbola.

**INCLUDE** u telo programa učitava čitave datoteke u kojima se definišu simboli, makroi i dr.

**END** označava kraj programa napisanog u asembleru.

## ▼ Poglavlje 4

# MAKRO KOMANDE U ASEMBLERSKIM JEZICIMA

## MAKRO NAREDBE

*Makro naredbe obezbeđuju jednostavan i efikasan način za definisanje skupa instrukcija kojeg je neophodno više puta upotrebiti u programu.*

**Makro ili makro naredbe obezbeđuju jednostavan i efikasan način za definisanje skupa instrukcija kojeg je neophodno više puta upotrebiti u programu.**

Ekvivalentan efekat se može postići definisanjem i pozivom procedure. Ipak takav pristup je manje efikasan, jer se procedure pozivaju u vremenu izvršenja.

Svaki poziv procedure zahteva izvršenje call naredbe i ret naredbe za povratak iz procedure.

Usporenje prouzrokovano izvršenjem ovih naredbi dolazi do izražaja kada procedura sadrži mali broj naredbi.

Makro naredbe se mogu shvatiti kao imena dodeljena programskom bloku

Sva obraćanja makrou se razrešavaju u toku prevođenja. Kada asembler prvi put nađe na makro, on ga smešta u tabelu makroa. Pri svakom sledećem nailasku na poziv tog makroa, asembler na to mesto prepisuje telo makroa.

Ovaj proces se naziva **makro** proširenje (engl. **macro expansion**). Izbor između pisanja procedure i makroa je rezultat kompromisa.

**Makroi se brže izvršavaju, ali zbog makro proširenja troše memorijске resurse. Sa druge strane procedure se razrešavaju u vremenu izvršenja, ali je njihovo telo definisano na samo jednom mestu.**

## DEFINISANJE MAKROA

*Primeri programa napisanog bez makroa i programa sa makro naredbom se značajno razlikuju.*

MOV EAX,P	SWAP MACRO
MOV EBX,Q	MOV EAX,P
MOV Q,EAX	MOV EBX,Q
MOV P,EBX	MOV Q,EAX
	MOV P,EBX
MOV EAX,P	ENDM
MOV EBX,Q	
MOV Q,EAX	SWAP
MOV P,EBX	
	SWAP
(a)	(b)

Slika 4.1 Kod na asemblerском језику за dvostruko razmenjivanje vrednosti promenljivih P i Q (a) bez makroa i b) sa rnakroom [Izvor: Autor]

Iako različiti asembleri imaju neznatno različite oznake za definisanje makroa, svi zahtevaju iste osnovne delove u definiciji makroa:

1. Zaglavlje makroa koje daje ime makroa koji se definiše.
2. Tekst koji je telo makroa.
3. Pseudoinstrukcija koja označava kraj definicije (npr. **ENDM**).

Kada asembler nađe na makro definiciju, on je čuva u tabeli makro definicija za kasniju upotrebu. Od tog trenutka, kad god se ime makroa (**SWAP** na slici) pojavi kao opkod, asembler ga zamenjuje telom makroa.

Upotreba imena makroa kao koda operacije poznata je kao **makro** poziv (engl. **macro call**), a njegova zamena telom makroa naziva se **proširenje makroa** (engl. **macro expansion**).

Pozivanje makroa ne treba mešati sa pozivanjem procedura makroa.

**Osnovna razlika je to što pozivanje makroa predstavlja instrukciju asembleru da zameni ime makroa telom makroa.**

**Poziv procedure je mašinska instrukcija koja se ugrađuje u program da bi se kasnije tokom njegovog izvršavanja pozvala procedura.**

## MAKRO SA PARAMETRIMA

*Makro sa ponavljanjem može se koristiti za skraćivanje izvornih programa u kojima se tačno ista sekvenca instrukcija ponavlja.*

MOV EAX,P	CHANGE MACRO P1, P2
MOV EBX,Q	MOV EAX,P1
MOV Q,EAX	MOV EBX,P2
MOV P,EBX	MOV P2,EAX
	MOV P1,EBX
MOV EAX,R	ENDM
MOV EBX,S	
MOV S,EAX	CHANGE P, Q
MOV R,EBX	
	CHANGE R, S
(a)	(b)

Slika 4.2 Program koji zamenjuje vrednosti promenljivih P i Q i promenljivih R i S (b) Program koji menja vrednosti promenljivih upotrebom makroa CHANGE. [Izvor: Autor]

Prethodno opisana makro mogućnost može se koristiti za skraćivanje izvornih programa u kojima se tačno ista sekvenca instrukcija ponavlja.

Često, međutim, program sadrži nekoliko sekvenci instrukcija koje su skoro, ali ne sasvim identične, kao što je **ilustrovano na slici (a)**.

Ovde prva sekvenca razmenjuje P i Q, a druga sekvenca razmenjuje R i S. Makro asembleri obrađuju slučaj skoro identičnih sekvenci dozvoljavajući makro definicijama da obezbede formalne parametre i dozvoljavajući makro pozivima da obezbede stvarne parametre.

Kada se makro proširi, svaki formalni parametar koji se pojavljuje u telu makroa se zamenjuje odgovarajućim stvarnim parametrom. Stvarni parametri se postavljaju u operand polje poziva makroa.

**Slika (b)** prikazuje program sa slike (a) prepisan korišćenjem makroa sa dva parametra. Simboli P1 i P2 su formalni parametri. Svako pojavljivanje P1 unutar tela makroa zamenjuje se prvim stvarnim parametrom kada se makro proširi. Slično, P2 je zamenjen drugim stvarnim parametrom.

#### U makro pozivu:

**CHANGE** P, Q P je prvi stvarni parametar, a K je drugi stvarni parametar. Tako su izvršni programi proizvedeni u oba dela slike identični.

**Oni sadrže potpuno iste instrukcije sa istim operandima**

## MAKRO - NAPREDNE FUNKCIJE

*Većina makro procesora ima čitav niz naprednih funkcija koje olakšavaju rad programeru asemblerorskog jezika.*

## **Jedan od problema koji se javlja kod svih asemblera koji podržavaju makroe je dupliranje oznaka.**

Prepostavimo da makro sadrži instrukciju uslovnog granaanja i oznaku na koju se grana. Ako se makro pozove dva ili više puta, oznaka će se duplirati uzrokujući grešku pri sklapanju.

Jedno rešenje je da programer obezbedi drugu oznaku za svaki poziv kao parametar.

Drugo rešenje (koje koristi **MASM**) je da se dozvoli da se oznaka proglaši **LOCAL**, pri čemu asembler automatski generiše drugačiju oznaku na svakom proširenju makroa.

Neki drugi asembleri imaju pravilo da su numeričke oznake automatski lokalne. MASM i većina drugih asemblera dozvoljavaju definisanje makroa unutar drugih makroa.

Makroi mogu pozvati druge makroe, uključujući i sebe.

**Ako je makro rekurzivan, to jest, poziva sam sebe, mora sebi proslediti parametar koji se menja pri svakom proširenju i makro mora testirati parametar i prekinuti rekurziju kada dostigne određenu vrednost.**

**Inače se asembler može staviti u beskonačnu petlju.**

**Ako se to dogodi, korisnik mora eksplicitno da zatvorи asemblera.**

**Ova lekcija sadrži video materijal. Ukoliko želite da pogledate ovaj video morate da otvorite LAMS lekciju.**

## ▼ Poglavlje 5

# NASM ASEMBLER

## NASM (NETWIDE ASSEMBLER)

*NASM (Netwide Assembler) je assembler koji podržava i 64-bitnu i 32-bitnu platformu.*

- **NASM** ([NetwideAssembler](#)) je assembler koji podržava i 64-bitnu i 32-bitnu platformu. Podržava mnoge izlazne formate pa se može izvršiti i pod Windows i Linux operativnim sistemom.

- **NASM za Windows** platformu možete preuzeti sa:

<https://www.nasm.us/pub/nasm/releasebuilds/2.11.05/win32/nasm-2.11.05-win32.zip>

- **NASM za linux** možete preuzeti pokretanjem komande:

`sudo apt-get install nasm`

- **NASM online** assembler možete koristiti, izmedju ostalih, na adresi:

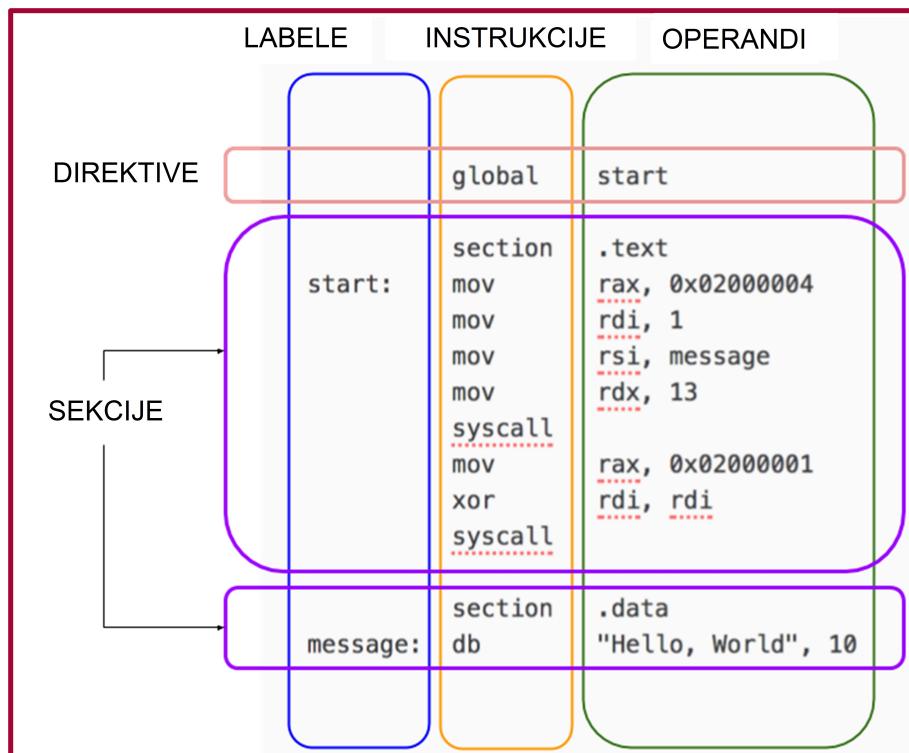
[https://www.tutorialspoint.com/compile\\_assembly\\_online.php](https://www.tutorialspoint.com/compile_assembly_online.php)

U NASM assembleru postoje više različitih segmenta u kodu:

- **Data segment** (.data)
- **BSS segment** (.bss)
- **Text segment** (.text)

## NASM ASEMBLER - STRUKTURA

*NASM (Netwide Assembler) se sastoji od seta Labela, Instrukcija, Operanada, Direktiva i Sekcija*



Slika 5.1 Set operacija kod NASM asemblera [Izvor: Autor]

<https://www.jdoodle.com/compile-assembler-nasm-online/>

[https://www.mycompiler.io/new/asm-x86\\_64](https://www.mycompiler.io/new/asm-x86_64)

[https://rextester.com/l/nasm\\_online\\_compiler](https://rextester.com/l/nasm_online_compiler)

[https://www.tutorialspoint.com/assembly\\_programming/try\\_asm.php](https://www.tutorialspoint.com/assembly_programming/try_asm.php)

## NASM (NETWIDE ASSEMBLER) - SEGMENTI KODA

*U NASM assembleru postoje više različitih segmenta u kodu*

**Data segment** služi za definiciju promenjivih koje već imaju neku određenu vrednost tj. vrednosti koje su inicijalizovane. Ovde obično deklarišemo imena fajlova, veličine buffera ili konstanti.

```
section .data
poruka: db 'Hello world!'           ; Deklarise se poruka (DB) koja
                                         ; sadrzi bajtove teksta 'Hello world!' (bez apostrofa)
velicinaporuke: equ 12              ; Deklarise se konstanta
                                         ; (EQU) velicinaporuke koja ima vrednost 12
buffersize: dw 1024                 ; Deklarise se velicina
                                         ; bufera za reč koja sadrzi 1024 (DW)
```

Slika 5.2 Data segment. [Izvor: Autor]

**BSS segment** služi za definiciju promenjivih koje nemaju u startu određenu vrednost i koje nisu inicijalizovane. Ovde se koriste sledeće instrukcije: RESB, RESW, RESD, RESQ i REST.

```
section .bss
    filename: resb 255          ; Rezerviše 255 bajta za promenjivu filename
    number:   resb 1            ; Rezerviše 1 bajt za promenjivu number
    btgnum:   resw 1            ; Rezerviše 1 reč (word) (1 reč = 2 bajta)
    realarray: resq 10           ; Rezerviše niz od 10 realnih brojeva
```

Slika 5.3 BSS segment[Izvor: Autor]

**Text segment** predstavlja segment u kome se piše sam kod. Ovaj segment mora početi deklaracijom početna (**global\_start**) koja govori kernel-u gde program počinje da se izvršava.

Ovo može da se poređi sa main metodom u C-u ili Javi.

```
section .text
    global _start
_start:
    mov    ebx, 2              ; Ovde program počinje
```

Slika 5.4 Text segment[Izvor: Autor]

## SET INSTRUKCIJA X86 I SISTEMSKI POZIVI

### *Primeri seta instrukcija x86 i sistemske pozivi*

- **ADD** ( Zbir )
- **DEC** ( Dekrement - Smanjiti broj za 1)
- **INC** ( Inkrement - Povećavanje broja za 1)
- **DIV** ( Deljenje brojeva bez znaka)
- **IDIV** ( Deljenje brojeva sa znakom )
- **MUL** ( Množenje brojeva bez znaka )
- **IMUL** ( Množenje brojeva sa znakom )
- **SUB** ( Oduzimanje )
- **CALL** ( Poziv procedure )
- **NOT** ( Logička operacija NE )
- **OR** ( Logička operacija ILI )
- **AND** ( Logička operacija I )
- **XOR** ( Ekskluzivni OR )
- **NEG** ( Drugi komplement )
- **RET** ( Vraćanje iz procedure )
- **ROL** ( Leva rotacija )
- **ROR** ( Desna rotacija )
- **LOOP** ( Petlja )
- **MOV** ( Premesti )

Celu listu možete pogledati na:

[https://en.wikipedia.org/wiki/X86\\_instruction\\_listings](https://en.wikipedia.org/wiki/X86_instruction_listings)

Sistemski pozivi omogućavaju prekide i pozive funkcija sistema (prekid aplikacije, ispis na ekranu i sl).

```
mov    eax,1      ; Sistemski broj za izlaz
mov    ebx,0      ; Izlazni kod 0
int    80h       ; Prekid 80h, koji govori kernelu
                  ; Uradí ovo (Misli se na funkciju i njen kod)
```

Slika 5.5 Sistemski poziv. [Izvor: Autor]

### Za korišćenje Linux sistemskih poziva potrebno je da:

- Postavite broj sistemskog poziva u EAX registar.
- Zapamtite argumente za taj sistemski poziv u registre EBX,ECX, itd.
- Pozvati prekid 80h.
- Rezultat se najčešće nalazi u EAX registru.

Postoji 6 registara koji pamte argumente koje sistemski poziv koristi: EBX, ECX, EDX, ESI, EDI i EB. U njima se respektivno upisuju argumenti, počev od EBX registra. Ako postoji više od 6 argumenata, onda se memoriska lokacija prvog argumenta pamti u EBXregistru.

### Sledeći deo koda prikazuje korišćenje sistemskog poziva sys\_exit:

```
mov eax,1 ; system call number (sys_exit) int 0x80 ; call kernel
```

## SISTEMSKI POZIVI I KOMENTARI

### *Primeri sistemskih poziva i komentara*

### Sledeći deo koda prikazuje korišćenje sistemskog poziva sys\_write:

- mov edx,4** ; dužina poruke
- mov ecx,msg** ; poruka koju treba napisati
- mov ebx,1** ; file descriptor (stdout)
- mov eax,4** ; system call number (sys\_write)
- int 0x80** ; call kernel

```
section .data
    hello: db 'ćao svete!',10      ; Definisane 'ćao svete' poruke
    helloLen: equ $hello           ; Uzimamo veličinu poruke hello
section .text
    global _start
_start:
    mov eax,4      ; Sistemski poziv za pisanje (sys_write)
    mov ebx,1      ; File opis 1 - standard izlaz
    mov ecx,hello  ; Stavljamo offset od promenjive hello u ecx registar
    mov edx,helloLen ; helloLen je konstanta i smeštamo je u edx.
                      ; mov edx,[helloLen] ukoliko želite vrednost
    int 80h       ; Pozivamo kernel
    mov eax,1      ; Sistemski poziv za izlaz (sys_exit)
    mov ebx,0      ; Izlazni kod 0 (nema greške)
    int 80h
```

Slika 5.6 Program koji ispisuje poruku [Izvor: Autor]

### Komentari

Komentari u asembleru počinju tačkom i zarezom ; Može da bude u posebnom redu, ; ovo je program koji ispisuje poruku na ekranu ili u istoj liniji sa instrukcijom, npr.add eax ,ebx ; saberi ebx i eax

Kako bi pokrenuli program najpre se vrši **kompajliranje i linkovanje.**

## ▼ Poglavlje 6

### Pokazne vežbe

#### ZADATAK 1

*Zadatak 1: (10 minuta)*

**Zadatak #1 (10 minuta)**

Napisati program u NASM jeziku koji ispisuje Hello World!

[https://rextester.com/l/nasm\\_online\\_compiler](https://rextester.com/l/nasm_online_compiler)

```
; NASM lab01

section .data
hello    db 'Hello world!', 10 ; 'Hello world!' plus a linefeed character
helloLen equ $-hello           ; Length of the 'Hello world!' string

section .text
    global _start
_start:
    mov eax, 4             ; The system call for write (sys_write)
    mov ebx, 1               ; File descriptor 1 - standard output
    mov ecx, hello           ; Put the offset of hello in ecx
    mov edx, helloLen        ; helloLen is a constant
    int 80h                  ; Call the kernel

    mov eax,1               ; The system call for exit (sys_exit)
    mov ebx,0               ; Exit with return code of 0 (no error)
    int 80h;
```

Slika 6.1 Rešenje zadatka #1 [Izvor: Autor]

#### ZADATAK 2

*Zadatak 2 (10 minuta)*

**Zadatak #2 (10 minuta)**

Napisati program u NASM jeziku koji ispisuje dve linije teksta!

[https://rextester.com/l/nasm\\_online\\_compiler](https://rextester.com/l/nasm_online_compiler)

```
; NASM lab02

section .data
hello:    db 'Hello world!', 0xa, 'Second text', 0xa ; test in two lines
helloLen equ $-hello           ; Length of the 'Hello world!' string

section .text
    global _start
_start:
    mov eax, 4          ; The system call for write (sys_write)
    mov ebx, 1          ; File descriptor 1 - standard output
    mov ecx, hello     ; Put the offset of hello in ecx
    mov edx, helloLen ; helloLen is a constant
    int 80h             ; Call the kernel

    mov eax, 1          ; The system call for exit (sys_exit)
    mov ebx, 0          ; Exit with return code of 0 (no error)
    int 80h;
```

Slika 6.2 Rešenje zadatka #2 [Izvor: Autor]

## PROJEKTNI ZADATAK IZ PREDMETA CS120

*Predmet CS120 od predispitnih obaveza sadrži i PZ koji se sastoji od tri mini-projekta.*

Na predmetu CS120 - Organizacija računara svaki student samostalno radi projektni zadatak (PZ), koji se sastoji od tri mini-projekta.

**Mini-projekti brane se u petoj, desetoj i petnaestoj nedelji semestra!**

Svaki student dužan je da tok izrade projekta izveštava svake nedelje u obliku kratkih izveštaja (do jednog pasusa) preko LAMS lekcija u okviru dodatnih aktivnosti interaktivnih lekcija.

O formatu mini-projekata svi studenti (tradicionalni i Internet) biće obavešteni mejlom od strane predmetnog profesora/asistenta.

## ✓ Poglavlje 7

### Zadaci sa samostalni rad

#### ZADACI ZA SAMOSTALNI RAD ZA LEKCIJU #11

*Zadaci za samostalni rad za lekciju #11 rade se okvirno 30 min.*

##### **Zadaci za samostalni rad:**

###### **Zadatak #1,(10 minuta)**

[https://rextester.com/l/nasm\\_online\\_compiler](https://rextester.com/l/nasm_online_compiler)

- #1. Napisati program u NASM jeziku koji ispisuje devet puta karakter

##### **Zadaci za samostalni rad:**

###### **Zadatak #2,(10 minuta)**

[https://rextester.com/l/nasm\\_online\\_compiler](https://rextester.com/l/nasm_online_compiler)

- #2. Napisati program u NASM jeziku koji dodaje dva broja koji se neposredno upisuju u registre

##### **Zadaci za samostalni rad:**

###### **Zadatak #3,(10 minuta)**

[https://rextester.com/l/nasm\\_online\\_compiler](https://rextester.com/l/nasm_online_compiler)

- #3.Napisati program u NASM jeziku koji ispisuje tri reda teksta koristeći makro poziv

## ✓ Poglavlje 8

### Domaći zadatak

#### DOMAĆI ZADATAK #11

*Domaći zadatak #8 okvirno se radi 20 minuta*

**Domaći zadatak #11 ( 20 minuta)**

Napisati program u NASM asembleru koji u tri reda ispisuje Vaše ime i prezime, broj indexa i godinu studiranja na ekranu.

Koristiti NASM online ili NASM asembler instaliran na Vašem računaru.

[https://rextester.com/l/nasm\\_online\\_compiler](https://rextester.com/l/nasm_online_compiler)

**Predaja domaćeg zadatka**

**Tradicionalni studenti:**

Domaći zadatak treba dostaviti najkasnije 7 dana nakon vežbi, za 100% poena. Nakon toga poeni se umanjuju za 50%.

**Domaći zadatak poslati predmetnim asistentima, sa predmetnim profesorima u CC.**

**Predati domaći zadatak koristeći .doc/.docx uputstvo dato u prvoj lekciji.**

**Internet studenti** treba poslati domaće zadatke najkasnije do 10 dana pred izlazak na ispit predmetnom asistentu zaduženog za internet studente.

Napomena:

Svaki domaći zadatak treba da bude napisan prema dokumentu za predaju domaćih zadataka koji je dat na kraju interaktivne lekcije.

## ▼ Ulaz/Izlaz- Opšta razmatranja

### ZAKLJUČAK

#### *Recime lekcije #11*

U ovoj lekciji istakli smo proizvoljnost granice između hardvera i softvera jer se ona stalno pomera. Današnji softver može postati hardver sutrašnjice i obrnuto. štaviše, nejasne su i granice između pojedinih nivoa.

Kao i u svakom drugom programskom jeziku, program u asembleru je sačinjen od niza uzastopno navedenih instrukcija. Stoga bi se moglo pretpostaviti da je svaku instrukciju dovoljno pročitati i prevesti u mašinski jezik u samo jednom prolazu kroz program.

Postojanje simboličkih imena, labela, makro definicija i operacija skokova u sintaksi asemblera zahteva da se pri prevođenju prođe više puta kroz kod programa. Da bi se program mogao prevesti neophodno je poznavati konkretne vrednosti svih promenljivih.

Na kraju lekcije je dat opis NASM asemblera, procesora 8088 i to u najvećim delom usmeren na opis detalja bitnih za programera.

#### **Literatura:**

A. Tanenbaum, Structured Computer Organization,

Chapter 07, pp. 517 – 529,