PANEVROPSKI UNIVERZITET APEIRON, BANJA LUKA
FAKULTET INFORMACIONIH TEHNOLOGIJA

Student: Ivan Pavlović
92-20/RITP-S

Niži programski jezici i programski prevodioci

(Predstavljanje brojeva i stringova u asemblerskom jeziku)

Mentor: Doc. dr Tijana Talić

Banja Luka, decembar 2021.

SADRŽAJ

UVOD

Sa stalnim napretkom tehnologije smatram da bi svako ljudsko trebalo poznavati programske jezike, jer ipak oni su dio nas i čine našu svakodnevnicu. Može se također navesti da programski jezici nam omogućuju komunikaciju sa računarom.

Programske jezike dijelimo u mašinski zavisne i mašinski nezavisne jezike. Mašinski zavisne jezike dalje dijelimo u mašinske, **asemblerske** i makro jezike.

Mašinski jezik je izgrađen nad binarnom azbukom 0 i 1 (B) i predstavlja nam interni jezik skoro svih savremenih računara i ovaj sistem ćemo najviše spominjati u ovom seminarskom radu.

Asemblerski (simbolički) jezici su veomba bliski mašinskim i nastaju zamjenom koda, skraćenica i adresama sa simboličkim imenima.
Pisanje programa na asemblerskom jeziku je proces sličan pisanju programa na bilo kom programskom jeziku. Također, program na asemblerskom jeziku možemo nazvati izvornim kodom.

1.Asembler

„Svaki računar sadrži mikroprocesor koji upravlja aritmetičkim, logičkim i kontrolnim aktivnostima računara. Svaka porodica procesora ima sopstveni skup instrukcija za rukovanje raznim operacijama kao što su dobijanje unosa sa tastature, prikazivanje informacija na ekranu i obavljanje raznih drugih poslova. Ove instrukcije se drugačije nazivaju *instrukcije mašinskog jezika*.

Procesor razumije samo dvije instrukcije mašinskog jezika, koji su stringovi jedinica (1) i nula (0). Međutim, mašinski jezik je previše nejasan i složen za korištenje u razvoju softvera. Dakle, asemblerski jezik niskog nivoa je dizajniran za određenu porodicu procesora koja predstavlja različite instrukcije u simboličkom kodu i razumljivijem obliku“ [1].

1.1.Prednosti asemblerskog jezika

„Razumijevanjem asemblerskog jezika bićemo svjesni o:

- Kako programi komuniciraju sa OS, procesorom i BIOS-om;
- Kako su podaci predstavljeni u memoriji i drugim uređajima;
- Kako procesor pristupa i izvršava instrukcije;
- Kako program pristupa spoljnim uređajima;

Druge prednosti asemblerskog jezika su:

- Zahtjeva manje memorije i vremena izvršenja;
- Omogućuje složene poslove, specifične za hardver na lakši način“ [1];

1.2.Hardver računara

„Glavni unutrašnji hardver računara sastoji se od procesora, memorije i registara. Registri su komponente procesora koje sadrže podatke i adrese. Da bi izvršili program, sistem ga kopira sa spoljnog uređaja u internu memoriju

Osnovna jedinica računarske memorije je bit. On može biti ON (1) ili OFF (0) i grupa od 8 povezanih bitova čini bajt na većini modernih računara.

Bit parnosti se koristi da bi se broj bitova u bajtu učinio neparnim. Ako je paritet paran, sistem pretpodstavlja da je došlo do greške parnosti, koja je mogla biti uzrokovana hardverskom greškom ili električnim smetnjama“ [1].

Savremeni procesori podržavaju ove tipove podataka:

- Word (2 bajta);
- DoubleWord (4 bajta, 32bita);
- QuadWord (8 bajta, 64 bita);
- Paragraph (16 bajta, 128 bita);
- KiloByte KB (1024 bajta);
- MegaByte MB (1,048,576 bajta);

1.3.Binarni brojevni sistem

„Svaki brojni sistem koristi pozicionu notaciju, odnosno svaka pozicija u kojoj je upisana cifra ima drugačiju pozicionu vrijednost. Svaka pozicija je stepen baze, što je 2 za binarni brojevni sistem, a počinju od 0 i povećavaju se za 1“ [1].

Sljedeća slika prikazuje pozicione vrijednosti za 8 – bitni binarni broj, gdje su svi bitovi postavljeni na uključeno.

Bit value	1	1	1	1	1	1	1	1
Position value as a power of base 2	128	64	32	16	8	4	2	1
Bit number	7	6	5	4	3	2	1	0

Slika 1 - Primjer svih 8bita uključenih

Vrijednost binarnog broja zasniva se na prisustvu 1 bita i njihove pozicione vrijednosti, tj. vrijednost broja navedenog u slici 1 je:

$$1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 = 255$$

1.3.1.Binarna aritmetika

U binarnom brojevnom sistemu koristimo osnovne operacije kao što su:

- Sabiranje
- Oduzimanje
- Množenje
- Djeljenje

1.3.1.1.Sabiranje binarnih brojeva

Za sabiranje binarnih brojeva koristimo istu terminologiju kao i za dekadne brojeve.

```
  100110101
+ 110110110
-----------
 1011101011
```

Slika 2 - Binarno sabiranje

1.3.1.2.Oduzimanje binarnih brojeva

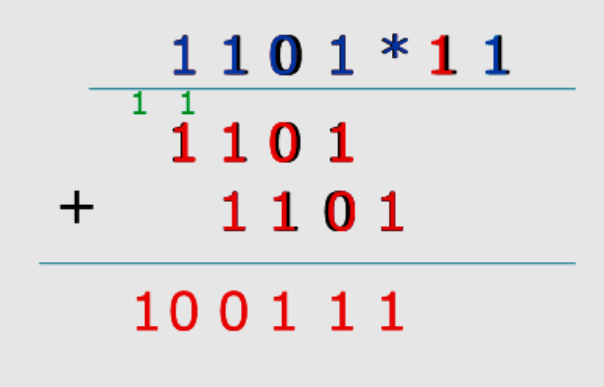
```
  11101001
- 10110111
----------
    110010
```

Slika 3 - Binarno oduzimanje

Kod oduzimanja možemo primjetiti da je 1 – 0 =; 1 – 1 = 0; 0 – 0 = 0; ali se situacija komplikuje ako naiđemo na 0 – 1. Taj problem riješavamo tako što od prve sljedeće cifre pozajmljujemo 1, a kako je u našem primjeru ta cifra 0, ne možemo da pozajmimo od nje, već tražimo sljedeću jedinicu. Kada smo izvršili to pozajmljivanje, vrijednost na onom mjestu sa koga smo prvi put htjeli da pozajmimo je 2. Sada odatle pozajmljujemo 1, tako da na onom mjestu gdje nismo mogli da izvršimo oduzimanje sada dobijamo 2 – 1, a to je 1 i ovaj postupak nastavljamo dok ne završimo traženo oduzimanje.

1.3.1.3.Množenje binarnih brojeva

Kod množenja binarnih brojeva susrećemo se sa istim primjerom kao i kod sabiranja binarnih brojeva u tome što se vrše isto kao dekadni brojevi. Znači da izvršimo množenje binarnih brojeva tretiraćemo ih kao dekadne brojeve.



Slika 4 - Binarno množenje

1.3.1.4. Djeljenje binarnih brojeva

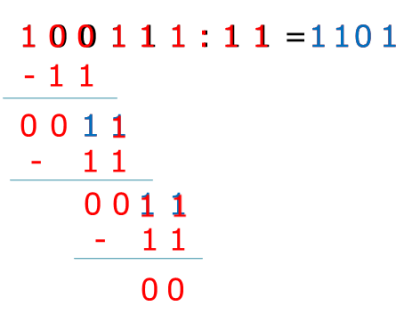


TABELA ODUZIMANJA

0 – 0 = 0
1 – 0 = 1
0 – 1 = 1 (1 se pozajmljuje)
1 – 1 = 0

TABELA MNOŽENJA

0 * 0 = 0
1 * 0 = 0
0 * 1 = 0
1 * 1 = 1

1.4.Heksadecimalni brojevni sistem

„Heksadecimalni sistem brojeva koristi bazu 16 i cifre u ovom sistemu se kreću od 0 do 15. Po konvenciji A do F se koriste za predstavljanje heksadecimalnih cifara koje odgovaraju decimalnim vrijednostima od 10 do 15.

Heksadecimalni brojevni sistem se koristi za skraćivanje dugih binarnih prezentacija. U prevodu, heksadecimalni sistem brojeva predstavlja binarni podatak tako što se svaki bajt dijeli na pola i izražava vrijednost svakog polubajta“ [1].

Decimalni brojevi	Binarna prezentacija	Heksadecimalna prezentacija
0	0	0
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Slika 5 - Heksadecimalna tabela

Da bi bismo konvertovali binarni broj u njegov heksadecimalni ekvivalent, razbijamo ga u grupe od po 4 uzastopne grupe, počevši sa desne strane, i prepisivajući te grupe preko odgovarajućih cifara heksadecimalnog broja.

2.Brojevi u asembleru

„Numerički podaci se uglavnom predstavljaju u binarnom sistemu. Aritmetičke instrukcije rade na binarnim podacima. Kada su brojevi prikazani na ekranu ili unijeti sa tastature, oni dolaze u ASCII obliku“ [1].

Zasada smo konvertovali ove ulazne podatke u ASCII obliku u binarni za aritmetička izračunavanja i konvertovali rezultat nazad u binarni i to je predstavljeno u slici 6.

```
section .text
   global _start        ;must be declared for using gcc

_start:                 ;tell linker entry point
   mov  eax,'3'
   sub     eax, '0'

   mov  ebx, '4'
   sub     ebx, '0'
   add  eax, ebx
   add  eax, '0'

   mov  [sum], eax
   mov  ecx,msg
   mov  edx, len
   mov  ebx,1            ;file descriptor (stdout)
   mov  eax,4            ;system call number (sys_write)
   int  0x80             ;call kernel

   mov  ecx,sum
   mov  edx, 1
   mov  ebx,1            ;file descriptor (stdout)
   mov  eax,4            ;system call number (sys_write)
   int  0x80             ;call kernel

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

section .data
msg db "The sum is:", 0xA,0xD
len equ $ - msg
segment .bss
sum resb 1
```

Slika 6 - Primjer korištenja brojeva u asembleru

Kada se kod sa slike 6 kompajlira i izvrši, dobićemo rezultat: The sum is: 7

„Takve konverzije, nažalost imaju dodatne troškove, a programiranje na asembleru omogućuje obradu brojeva na efikasniji način, u BINARNOM obliku. Decimalni brojevi se mogu predstaviti u dva oblika:

- ASCII formi;
- BCD formi (Binary Coded Decimal)“ [1];

2.1.ASCII Prezentacija

„U ASCII prezentaciji, decimalni brojevi se čuvaju kao niz ASCII znakova. Npr. decimalna vrijednost 1234 se čuva kao: 31 32 33 34H

Gdje je 31H ASCII vrijednost za 1,32H je ASCII vrijednost za 2 i tako dalje. Postoje četri instrukcije za obradu brojeva u ASCII prikazu:

- AAA – ASCII podešavanje nakon dodavanja;
- AAS – ASCII podešavanje nakon oduzimanja;
- AAM – ASCII podešavanje nakon množenja;
- AAD – ASCII podešavanje prije djeljenja“ [1];

Demonstracija koncepta AAA prikazano je u slici 7.

```
section .text
   global _start        ;must be declared for using gcc

_start:                 ;tell linker entry point
   sub     ah, ah
   mov     al, '9'
   sub     al, '3'
   aas
   or      al, 30h
   mov     [res], ax

   mov  edx,len         ;message length
   mov  ecx,msg         ;message to write
   mov  ebx,1           ;file descriptor (stdout)
   mov  eax,4           ;system call number (sys_write)
   int  0x80            ;call kernel

   mov  edx,1           ;message length
   mov  ecx,res         ;message to write
   mov  ebx,1           ;file descriptor (stdout)
   mov  eax,4           ;system call number (sys_write)
   int  0x80            ;call kernel

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

section .data
msg db 'The Result is:',0xa
len equ $ - msg
section .bss
res resb 1
```

Slika 7 - AAA koncept

Kada se kod sa slike 7 kompajlira i izvrši, dobićemo rezultat: The Result is: 6

2.2.BCD Prezentacija

„Postoje dva tipa BCD prezentacije:

- Unpacked BCD (raspakovan);
- Packed BCD (zapakovan);

U raspakovanoj BCD prezentaciji, svaki bajt čuva binarni ekvivalent decimalne cifre. Npr. broj 1234 se čuva kao: 01 02 03 04H

Postoje dva uputstva za obradu ovih brojeva:

- AAM – ASCII podešavanja nakon množenja;
- AAD – ASCII podešavanja nakon djeljenja;

Četiri ASCII instrukcije za podešavanje, AAA, AAS, AAM i AAD, također se mogu koristi sa raspakovanim BCD prikazom. U zapakovanoj BCD prezentaciji, svaka cifra se čuva pomoću četiri bita. Dve decimalne cifre su spakovane u bajt. Npr. broj 1234 se čuva kao: 12 34H

Postoje dve instrukcije za obradu ovih brojeva:

- DAA – Decimalno podešavanje nakon sabiranja
- DAS – Decimalno podešavanje nakon oduzimanja

Ne postoji podrška za množenje i djeljenje u upakovanoj BCD prezentaciji“ [1].

Program prikazan na slici 5 sabira dva petocifrena decimalna broja i prikazuje zbir pomoću prikazanih koncepata gore.

```
section .text
   global _start        ;must be declared for using gcc

_start:                 ;tell linker entry point

   mov     esi, 4       ;pointing to the rightmost digit
   mov     ecx, 5       ;num of digits
   clc
add_loop:
   mov  al, [num1 + esi]
   adc  al, [num2 + esi]
   aaa
   pushf
   or   al, 30h
   popf

   mov  [sum + esi], al
   dec  esi
   loop add_loop

   mov  edx,len         ;message length
   mov  ecx,msg         ;message to write
   mov  ebx,1           ;file descriptor (stdout)
   mov  eax,4           ;system call number (sys_write)
   int  0x80            ;call kernel

   mov  edx,5           ;message length
   mov  ecx,sum         ;message to write
   mov  ebx,1           ;file descriptor (stdout)
   mov  eax,4           ;system call number (sys_write)
   int  0x80            ;call kernel

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

section .data
msg db 'The Sum is:',0xa
len equ $ - msg
num1 db '12345'
num2 db '23456'
sum db '     '
```

Slika 8 - Primjer BCD prezentacije

Kada se kod sa slike 8 kompajlira i izvrši, dobićemo rezultat: The Sum is: 35801

3.Stringovi u asembleru

„Stringovi u asembleru niz potpisanih bajtova tj. znakova (engl. chars) interpretiranih njihovim ASCII kodom. Karakter 'A' se čuva kao 0x41, a karakter 'B' se čuva kao 0x42 itd.

Po konvenciji, obično se završavaju sa NULL karakterom – vrijednost 0. Ova konvencija nam je omogućila da odbacimo velike bafere (komade memorije) i da koristimo samo dijelove koji su nam potrebni, a zatim kasnije identifikujemo koliki je dio bafera zapravo u potrebi.

Primjer:

0x41	0x70	0x70	0x6c	0x65	0x00	0x4f	0x92	0xa0	0x07	0x1c	0x1c	0x3d	0x82	0xbc	0x0f

Mi bismo to protumačili kao 0x41 0x70 0x70 0x6c 0x65 - 0x00 govori *'prestani da čitaš ovdje'*. Sve ostalo u tom baferu bi se ignorisalo.

Standardni način učitavanja stringa je preko .asciz direktorijuma (z označava 0 na kraju). Ono očitava string u memoriju i dodaje nulti bajt 0x00 da ga završi.

Primjer ispod učitava znakove 'A', 'p', 'p', 'l', 'e' i postavlja 0x00 nakon njih“ [2]:

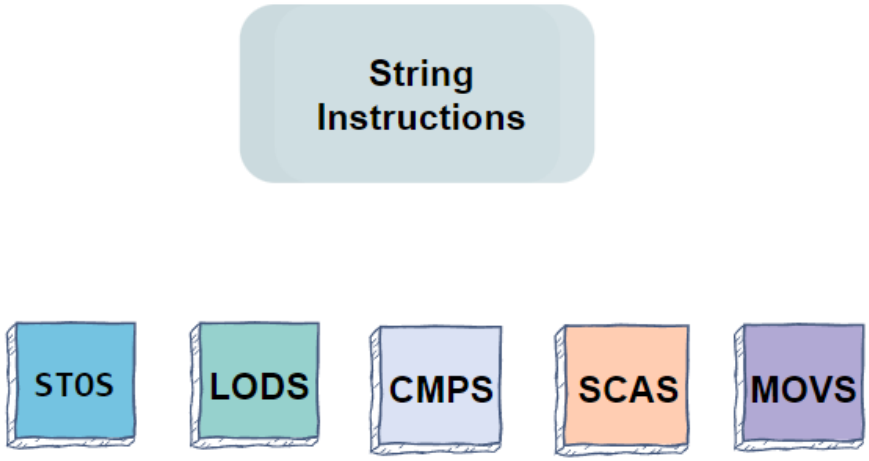
```
.data
.asciz   "Apple"

@use 0xFF to show where string ends
.byte 0xFF
```

3.1.Instrukcije stringova

„Osnovne instrukcije stringova su:

- STOS – Skladišteni string (engl. Stored String);
- LODS – Učitavanje string (engl. Load String);
- CMPS – Upoređivanje string (engl. Compare String);
- SCAS – Skeniranje stringa (engl. Scan String);
- MOVS – Pomjeranje stringa (engl. Move String);



Slika 9 - Instrukcije stringova

Svako od ovih uputstava ima dve varijante, varijantu bajta i varijantu word-a. Varijanta se može dobiti dodavanjem B ili W na kraju instrukcije. Primjer: MOVSB ili MOVSW“ [3].

3.1.1.DI SI Registri

„DI i SI se koriste za pristup memoriji, drugačije se nazivaju izvorni (engl. source) indeks i odredišni (engl. destination) indeks radi string instrukcija.

Kad god instrukciji treba izvor memorije, DS:SI drži pokazivač na nju, a za odredište memorije, pokazivač se postavlja na ES:DI“ [3].

3.1.2.REPE i REPNE

„REPE – ponavlja sljedeću instrukciju stringa dok je postavljen ZERO FLAG.

RENE – ponavlja sljedeću instrukciju stringa dok je ZERO FLAG uklonjen.

Oni se koriste sa SCAS i CMPS instrukcijama“ [3].

3.1.3.STOS

„STOS transferuje bajt ili word iz registra AL ili AX u memorijski element adresiran od strane ES:DI i ažurira DI ka sljedećoj lokaciji. STOS se često koristi za brisanje bloka memorije ili njegovo popunjavanje konstantom. Izvor će uvijek biti AL ili AX. Ako je *data flag* (DF) jasan, DI se povećava. Ako je DF podešen, DI će biti smanjen“ [3].

```
cld: clear data flag
std: set data flag
```

3.1.4.LODS

„LODS prenosi bajt ili word sa izvorne lokacije DS:SI u AL ili AX. U suštini radi obrnuto od STOS“ [3].

```
msg: db 'hello world'
len: dw 11

mov di, ax    # point di to location
mov si, [msg] # point si to string
mov cx, [len] # load length in cx

cld    # auto-increment

char_loop:
lodsb  # load next char in al
loop char_loop
```

3.1.5.SCAS

„SCAS uspoređuje izvorni bajt ili word u registru AL ili AX sa elementom odredišta stringa koji adresira ES:DI i ažurira zastavice (engl. Flags).

DI se ažurira na sljedeću lokaciju. SCAS se koristi za provjeru jednakosti/nejednakosti u stringu pomoću REPE ili REPNE.
Također može se koristiti za pronalaženje 0 u stringu sa nultim krajem da bi se izračunala njegova dužina“ [3].

```
msg: db 'hello world', 0 # null terminated string

mov di, [msg]  # point di to str
mov cx, 0xffff # load MAX in cx
xor al, al     # load 0 in al
repne scasb
mov ax, 0xffff
sub ax, cx     # ax now contains length
```

3.1.6.CMPS

„CMPS oduzima izvornu lokaciju DS:SI od odredišne lokacije ES:DI bez uticaja na sam izvor i odredište. SI i DI se ažuriraju u skladu sa tim.
Ako se koristi sa REPE ili REPNE, može se koristiti za provjeru jednakosti/nejednakosti. CMPS se također može koristiti za pronalaženje podstringa unutar stringa“ [3].

```
msg1: db 'Edpresso'
msg2: db 'EdpressO'
mov si, [msg1]
mov di, [msg2]
mov ax, 1  # true: strings are equal
repe cmpsb
je exit

mov ax, 0 # false: strings are unequal
exit:
mov ax, ox4c00
int 21h
```

3.1.7.MOVS

„MOVS prenosi bajt ili word iz DS:SI u ES:DI i ažurira SI i DI u skladu sa tim, također koristi se za pomjeranje bloka memorije.

REP dozvoljava da se instrukcija ponovi *n* puta, gdje je *n* vrijednost sačuvana u CKS“ [3].

ZAKLJUČAK

Lično smatram da svakom budućem programeru je bitno da bar malo poznaje asemblerski jezik. U ovom predmetu dosta toga sam naučio iako radi posla nisam mogao fizički sudjelovati ali bitno je da naglasim jeste to da ovaj predmet pokaže kako računar funkcioniše, što bi se reklo ispod haube.

POPIS SLIKA

CITATNI IZVORI

[1] »Tutorials point,« [Mrežno]. Available: https://www.tutorialspoint.com/assembly_programming/assembly_introduction.htm.

[2] »Runestone,« [Mrežno]. Available: https://runestone.academy/runestone/books/published/armTutorial/Arrays/Strings.html.

[3] »Educative,« [Mrežno]. Available: https://www.educative.io/edpresso/how-to-use-strings-in-assembly-language.