Uvod u asemblerski jezik

doc. dr Miloš Ivanović, PMF Kragujevac

Oktobar 2012.

1 Mašinski jezik

Svaki tip CPU-a razume isključivo svoj sopstveni mašinski jezik. Instrukcije u mašinskom jeziku su pohranjene kao bajtovi unutar memorije. Svaka instrukcija ima svoj jedinstveni numerički kod, tzv. operacioni kod ili **opcode**. Opcode se uvek nalazi na početku instrukcije. Mnoge instrukcije nakon opcode-a sadrže i podatke. Veoma je teško programirati direktno u mašinskom jeziku. Sledi primer instrukcije koja izračunava zbir EAX i EBX registara i rezultat smešta u EAX registar:

```
03 C3
```

Na sreću, postoji tzv. asembler koji tekstualne instrukcije prevodi direktno u mašinski jezik. Jedan od takvih programa je i NASM (*Netwide ASseMbler*).

2 Asemblerski jezik

Asemblerski program se pohranjuje kao tekst, tj. kao bilo kakav program napisan u višem programskom jeziku kao što je C. Svaka asemblerska instrukcija predstavlja po jednu mašinsku instrukciju, dakle postoji "1-1" preslikavanje. Na primer, gornja instrukcija se u asembleru reprezentuje kao:

```
add eax, ebx
```

Dakle, ovde je značenje instrukcije daleko jasnije nego u mašinskom kodu. Opšti oblik instrukcije u asembleru je:

```
mnemonic operand(s)
```

Zadatak asemblera je da tekstualni kod prevede u mašinski, slično kao što to čine kompajleri viših programskih jezika. Shodno bliskosti asemblera i mašinskog jezika, asembler je dosta jednostavniji od bilo kog kompajlera.

Druga važna razlika između asemblera i viših programskih jezika je u tome što **asemblerski programi zavise od arhitekture CPU-a i nisu portabilni**.

U daljem toku nastave biće korišćen NASM asembler na platformi Linux i386 32bit. Drugi popularni asembleri su MASM (*Microsoft Assembler*) i TASM (*Turbo Assembler*). Postoje izvesne razlike u sintaksi među pomenutim asemblerima.

3 Instrukcijski operandi

Asemblerske instrukcije imaju različit broj i tip operanada, međutim, svaka instrukcija sama po sebi ima fiksni broj operanada (od 0 do 3), koji mogu biti sledećih tipova:

- **registar** - ovi operandi referenciraju direktno CPU registre
- **memorija** - referenciraju podatke u memoriji. Adrese podataka mogu biti konstante zadate u kodu ili zadate vrednostima u registrima. Adrese su uvek dodatak (*offset*) koji se zadaje od početka segmenta.
- **neposredni** - fiksne vrednosti zadate unutar same instrukcije. Čuvaju se u kodnom segmentu (`.text`), a ne u segmentu podataka (`.data, .bss`).
- **implicitni** - operandi koji nisu prikazani na eksplicitan način. Na primer, instrukcija `inc` dodaje jedinicu registru ili podatku u memoriji. Jedinica je implicirana.

4 Osnovne instrukcije

Najosnovnija asemblerska instrukcija je instrukcija `MOV`. Ona pomera podatak sa jedne lokacije na drugu (poput operatora dodele u višem jeziku; :=, =). Uzima dva operanda:

```
mov dest, src
```

Podatak specificiranu `src` kopira se u `dest`. Restrikcija je da ne mogu oba operanda biti podaci u memoriji, već se ta vrsta dodele mora vršiti preko registra. Druga restrikcija se odnosi na veličinu operanada, tj. uslov da operandi moraju biti iste dužine. Ne može se vrednost iz AX registra (16 bita) smestiti u BL registar (8 bita). Evo primera:

```
mov eax, 3 ; smeštanje broja 3 u EAX registar, 3 je neposredan operand
mov bx, ax ; smeštanje vrednosti u AX registru u BX registar
```

`ADD` instrukcija služi za sabiranje celih brojeva:

```
add eax, 4 ; eax=eax+4
add al, ah ; al=al+ah
```

`SUB` služi za oduzimanje celih brojeva:

```
sub bx, 10       ; ebx=ebx-10
sub ebx, edi     ; ebx=ebx-edi
```

`INC` i `DEC` instrukcije služe za inkrementiranje, odn. dekrementiranje za 1:

```
inc ecx      ; ecx++
dec dl       ; dl--
```

5 Direktive

Direktive se tiču samog asemblera, a ne CPU-a. U opštem slučaju se koriste da instruišu asembler da učini neku akciju ili da ga o nečemu informišu. Uobičajene namene direktiva su:

- definisanje konstanti
- definisanje memorijskih resursa za smeštaj podataka
- grupisanje memorije u segmente
- (uslovno) uključivanje izvornog (source) koda

NASM kod prolazi kroz pretprocesor potpuno isto kao C, pri čemu ima i veliki broj sličnih instrukcija kao C. Jedino što kod NASM-a te instrukcije počinju znakom % umesto #.

5.1 `equ` direktiva

`Equ` direktiva se koristi za definisanje simbola. Simboli su imenovane konstante u asemblerskom programu. Format instrukcije je:

```
simbol equ vrednost
```

Simboli se kasnije ne mogu redefinisati.

5.2 `%define` direktiva

Ova direktiva ima jasnu analogiju sa `#define` direktivom u C-u. I način upotrebe je skoro identičan, tj. upotrebljava se za definisanje konstanti i makroa:

```
%define SIZE 100
mov eax, SIZE
```

Makroi su efikasniji od simbola jer se mogu redefinisati.

5.3 Direktive podataka (*data directives*)

Direktive podataka se koriste u segmentu podataka, i to za rezervaciju prostora u memoriji za podatke. Prvi metod definiše samo prostor za njihov smeštaj, dok drugi način rezerviše prostor i dodeljuje inicijalnu vrednost. Prvi metod koristi neku od `RESX` direktiva, gde se `X` zamenjuje slovom koje definiše veličinu objekta (objekata). Tabela prikazuje listu dozvoljenih vrednosti:

Naziv	**Vrednost**
word	2 bajta
double word	4 bajta
quad word	8 bajtova
paragraph	16 bajtova

Drugi metod (onaj koji inicijalizuje vrednost) koristi jednu od `DX` direktiva, gde `X` ima isto značenje kao kod `RESX`.

Uobičajeno je da se memorijske lokacije obeležavaju labelama. Evo nekoliko primera:

```
L1   db   0        ; bajt označen sa L1 sa početnom vrednosti 0
L2   dw   1000     ; word označena sa L2 vrednosti 1000
L3   db   110101b  ; bajt označen sa L3 vrednosti 110101 (53 dekadno)
L4   db   12h      ; bajt označen sa L4 vrednosti 12h (18 dekadno)
L5   db   17o      ; bajt označen sa L4 vrednosti 17 oktalno (15 dekadno)
L6   dd   1A92h    ; double word označen sa L4 vrednosti hex 1A92
L7   resb 1        ; 1 neinicijalizovani bajt
L8   db   "A"      ; bajt inicijalizovan na ASCII kod od A (65)
```

Važno je napomenuti da su dozvoljeni i jednostruki i dvostruki navodnici i da se tretiraju na potpuno isti način. Uzastopni podaci se smeštaju jedan za drugim i u memoriji računara, tj. `L2` se u memoriji smešta odmah posle `L1`.

Takođe se mogu definisati i memorijske sekvence, što je jako važno pri definiciji nizova i stringova:

```
L9   db   0,1,2,3            ; definiše 4 inicijalizovana bajta
L10  db   "w","o","r","d",0  ; definiše C string = "word"
L11  db   'word',0           ; isto što i L10
```

`DD` direktiva može poslužiti kako za definisanje celobrojnih konstanti, tako i za definisanje konstanti u pokretnom zarezu jednostruke preciznosti (`float` u C-u). Nasuprot tome, `DQ` se koristi samo za brojeve u pokretnom zarezu dvostruke preciznosti.

Za dugačke sekvence, NASM direktiva `TIMES` je veoma korisna:

```
L12  times 100 db 0          ; isto sto i 100 (db 0)
L13  resw  100               ; rezervise prostor za 100 reci
```

Treba naglasiti da se labele mogu koristiti u svrhu referisanja podataka u kodu. Postoje dva načina za to:

1. Ako se koristi **čista labela**, ona se odnosi na adresu podatka u memoriji
2. Ako se upotrebi **labela u uglastim zagradama ([])**, interpretira se kao podatak na toj adresi.

Drugim rečima, labela je neka vrsta pokazivača na podatke u memoriji, dok je operator "[]" dereferencira na isti način kako to čini "*" u C-u. U 32-bitnom modu, adresa je 32-bitna. Evo nekoliko primera:

```
mov   al, [L1]        ; kopira bajt na L1 u AL
mov   eax, L1         ; EAX=adresa bajta na L1
mov   [L1], ah        ; kopiraj AH u bajt na L1
mov   eax, [L6]       ; double word na L6 u EAX
add   eax, [L6]       ; EAX=EAX+double word na L6
add   [L6], eax       ; double word na L6 += EAX
mov   al, [L6]        ; kopiraj prvi bajt na L6 u AL
```

Poslednja linija pokazuje joše jednu važnu osobinu asemblera; **ASEMBLER NE VODI RAČUNA O TIPU PODATKA NA KOJI SE LABELA REFERENCIRA**. Na programeru je da o tome vodi računa. Zato je asembler podložniji greškama od C-a.

Primera radi, sledeća instrukcija nije korektna:

```
mov [L6], 1          ; smestiti 1 na adresu L6
```

Što nije u redu? Instrukcija proizvodi grešku "*operation size not specified*" zato što ne zna kako da interpretira "1"; kao bajt, reč ili duplu reč. Da bi se ovo ispravilo, dodaje se specifikator veličine:

```
mov dword [L6], 1     ; smestiti 1 na adresu L6
```

6 Prvi asemblerski program

6.1 Primer programa

Danas, kada je dostupan veliki broj veoma moćnih viših programskih jezika, postavlja se pitanje da li je upošte potrebno programirati u asembleru. Danas se asembler više i ne koristi za pisanje celovitih programa, već pre svega za izvođenje nekoliko kritičnih rutina. Zašto? Zato što je mnogo lakše programirati u višem pogramskom jeziku.

Postavlja se pitanje: Zašto uopšte učiti asembler? Evo i nekoliko odgovora na to pitanje:

1. Ponekad je kod napisan u asembleru manji i brži nego kod koji generiše kompajler.
2. Asembler dozvoljava direktne pozive mogućnosti hardverskih uređaja koji mogu biti otežani ili čak nemogući u višem programskom jeziku.
3. Učenje asemblera pomaže u dubljem razumevanju rada računara.
4. Učenje asemblera pomaže u razumevanju kako rade kompajleri ili jezici kao što je C.

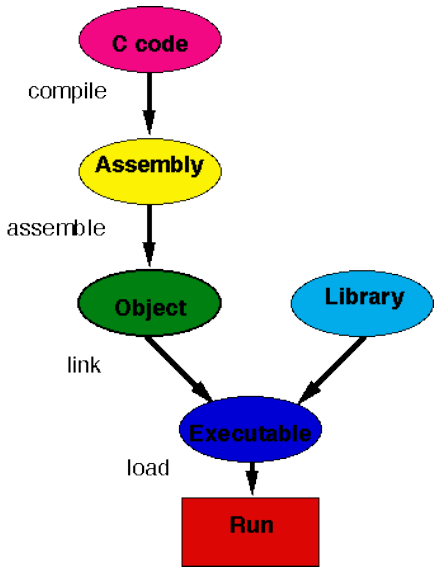
Na Listingu 1 dat je prvi program napisan u asembleru. U komentaru je dat i odgovarajući C program koji ispisuje isti *output* kao i asemblerski program, kao i uputstvo za kompajliranje (asembliranje) i linkovanje.

Izvorni kôd 1: prvi.asm

```
; prvi.asm       Stampa ceo broj iz registra i iz memorije
; Assemble:      nasm -f elf -l prvi.lst  prvi.asm
; Link:          gcc -o prvi.exe prvi.o
; Run:           ./prvi.exe
; Output:        a=5, eax=7

; Ekvivalentan C kod
; /* prvi.c  odtsampaj ceo broj i vrednost izraza */
; #include <stdio.h>
; int main()
; {
;   int a=5;
;   printf("a=%d, eax=%d\n", a, a+2);
;   return 0;
; }

; Deklaracija eksternih funkcija
;
extern        _printf        ; C funkcija koja ce biti pozvana

; Data segment
segment .data                 ; Data segment, inicijalizovane promenljive
      a:      dd     5        ; int a=5;
      fmt:    db     "Izlaz: a=%d, eax=%d", 10, 0 ; Format za printf,'\n','0'

; Kodni segment
segment .text
global  _main                 ; ulazna tacka koju trazi gcc linker

_main:                        ; labela za main
```

```
        enter 0,0               ; kreiranje stek okvira, kao push ebp / mov ebp,esp

; Koristan kod
        mov     eax,[a]         ; prenesi "a" iz memorije u registar EAX
        add     eax, 2          ; eax+=2
        push    eax             ; stavi na stek "a+2"
        push    dword [a]       ; stavi na stek vrednost "a"
        push    dword fmt       ; stavi na stek adresu formata
        call    _printf         ; pozovi C funkciju
        add     esp, 12         ; vrati ESP na vrednost pre 3 push-a
; Kraj korisnog koda

        leave                   ; unistavanje stek okvira, isto kao pop ebp
        mov     eax,0           ; izlazni kod. "0" znaci da nije bilo greske
        ret                     ; return
```

Da bi se od izvornog koda, bilo napisanog u asemblerskom jeziku ili jeziku C dobio izvršni fajl, potrebno je izvršiti niz koraka, datih šematski na Slici 1.



Slika 1: Dijagram prevođenja programskog teksta u izvršni kôd

Asembliranje se vrši komandom `nasm`, pri čemu `elf` predstavlja format izlaznog objaktnog fajla, a parametrom `-l listing-fajl` je dat naziv izlaznog listing fajla. Komandom `gcc` vrši se linkovanje, u ovom slučaju sa standardnom C blbiotekom (u kojoj se i nalazi funkcija `printf()`). Rezultat navedenog niza koraka je izvršni fajl `prvi.exe`.

6.2 Sadržaj listing fajla

`Nasm` parametrom `-l listing-fajl` daje se naziv izlaznog listing fajla. Ovaj fajl pokazuje kako se kôd asemblira. Evo kako izgledaju dve linije iz `data` segmenta:

```
23 00000000 05000000                    a:      dd      5   ; int a=5;
24 00000004 497A6C617A3A20613D-         fmt:    db      "Izlaz: a=%d...
25 0000000D 25642C206561783D25-
26 00000016 640A00
```

Prva kolona u svakoj liniji je redni broj te linije, a druga ofset podatka u `data` segmentu u heksadekadnom obliku. U trećoj koloni se nalaze sirovi podaci u heksadekadnom obliku (2 cifre za svaki bajt). U čestvtoj koloni stoji linija iz `source` fajla. *Offset*-ovi najverovatnije neće

odgovarati pravim *offset*-ovima u *exe* fajlu, jer se izvršni fajl u opštem slučaju sastoji iz više modula.

Takođe, valja prodiskutovai jednu nelogičnost (na pravi pogled) u liniji 23 listing fajla. Naime, vrednost labele `a` umesto da bude predstavljena kao `00000005`, u listing fajlu je data sa `05000000`. Ovo se dešava zbog toga što *Intel*-ovi procesori skladišete podatke u formatu tzv. *little-endian*, tj. na početku ide bajt sa najmanjom težinom, a na kraju bajt sa najvećom težinom, pa 00000005 postaje 05000000.

7 Instalacija potrebnog programskog okruženja

7.1 Ubuntu/Debian Linux

Da bi se uspešno prevodili i linkovali programi napisani na asemblerskom jeziku, potrebno je iz programa `Software Center` ili iz terminala instalirati pakete `nasm`, `gcc` i `build-essential`. Iz terminala se pomenuti paketi instaliraju komandom:

```
sudo apt-get install nasm build-essential gcc
```

nakon čega je potrebno uneti odgovarajuću lozinku. Takođe, ukoliko se radi direktno na nekom Linux sistemu, simboli "_" ispred `main` i `printf` nisu potrebni.

Ukoliko se radi o 64-bitnom sistemu, takodje je potreban i paket gcc-multilib, tako da komanda za instalaciju glasi:

```
sudo apt-get install nasm build-essential gcc gcc-multilib
```

Na 64-bitnim sistemima je potrebno naglasiti da želimo da radimo u 32-bitnom modu, tako da se mora staviti i dodatni argument pri svakom pozivu `gcc` linkera, na primer:

```
gcc -m32 -o prvi.exe prvi.o
```

7.2 MS Windows

Za emulaciju UNIX/Linux radnog okruženja, pod Windows sistemima se koristi paket `Cygwin`. Ovaj paket je besplatan i lako se instalira pokretanjem setup programa sa njegove web stranice `http://cygwin.com/install.html`. Evo sleda koraka koje treba izvesti da bi se došlo do funkcionalnog `Cygwin shell`-a:

1. Sa `http://cygwin.com/install.html` preuzeti program `setup.exe` i postaviti ga na desktop ili u neki drugi direktorijum. Ovaj fajl ne treba brisati, jer se preko njega vrše sve instalacije i ažuriranja paketa `Cygwin`-a.

2. Pokrenuti program `setup.exe` i proći kroz sve korake do ekrana *Choose A Download Site*.

3. Izabrati geografski blizak sajt za preuzimanje paketa, npr. `http://mirrors.mojhosting.sk`.

4. Iz grupe `devel` izabrati pakete `nasm` i `gcc` za instalaciju.

5. Sačekati da se izvrši preuzimanje potrebnih paketa i njihova instalacija

6. Okruženje `Cygwin` se pokreće dvoklikom na `Cygwin` ikonu na desktopu, pri čemu se korisnik automatski pozicionira u direktorijum `C:/cygwin/home/$USERNAME`, na primer `C:/cygwin/home/Pera`.

Literatura

[1] Paul A. Carter, *PC Assembly Language*, `http://www.drpaulcarter.com/pcasm`, 2006.