Arhitektura računara 1

ASEMBLER - 2. termin

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 vežbi biće korišćen NASM asembler na platformi Debian 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 restrikcije 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:

Jedinica	Slovo
byte	B
word	W
double word	D
quad word	Q
ten bytes	T

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        ; byte labeled L1 with initial value 0
L2    dw     1000     ; word labeled L2 with initial value 1000
L3    db     110101b  ; byte initialized to binary 110101 (53 in decimal)
L4    db     12h      ; byte initialized to hex 12 (18 in decimal)
L5    db     17o      ; byte initialized to octal 17 (15 in decimal)
L6    dd     1A92h    ; double word initialized to hex 1A92
L7    resb   1        ; 1 uninitialized byte
L8    db     "A"      ; byte initialized to ASCII code for 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              ; defines 4 bytes
L10   db     "w", "o", "r", 'd', 0   ; defines a C string = "word"
L11   db     'word', 0               ; same as 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                ; equivalent to 100 (db 0)'s
L13   resw   100                    ; reserves room for 100 words
```

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

- Ako se koristi **čista labela**, ona se odnosi na adresu podatka u memoriji
- 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]       ; copy byte at L1 into AL
mov    eax, L1        ; EAX = address of byte at L1
mov    [L1], ah       ; copy AH into byte at L1
mov    eax, [L6]      ; copy double word at L6 into EAX
add    eax, [L6]      ; EAX = EAX + double word at L6
add    [L6], eax      ; double word at L6 += EAX
mov    al, [L6]       ; copy first byte of double word at L6 into 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. Ulaz i izlaz

Ulazno/izlazne aktivnosti su, po prirodi stvari, veoma zavisne od sistema. Ove rutine zahtevaju interakciju sa sistemskim hardverom. Jezici višeg nivoa, kao što je C, isporučuju biblioteku standardnih rutina koje omogućavaju vrlo jednostavan i uniforman interfejs za I/O. Asembler nema nikakvu standardnu biblioteku za ovu svrhu.

Vrlo je uobičajeno da se asemblerske rutine mešaju sa C-ovskim. Jedna od prednosti ovog pristupa je da tada asembler može da poziva C bibloteku za I/O rutine. Bez ulaženja u pravila pozivanja C-a iz asemblera i obrnuto, za I/O koristiće se nezavisna biblioteka koju daje autor knjige *PC Assembly Language*, pod nazivom asm_io. Da bi se pozivale I/O rutine, mora se uključiti fajl sa informacijama o njima:

```
%include "asm_io.inc"
```

Tabela pomenutih rutina sa objašnjenjem sledi:

print_int	štampa na ekran vrednost celog broja u registru EAX
print_char	štampa karakter čija je ASCII vrednost smeštena u AL registar
print_string	štampa string čija se adresa nalazi u EAX. String mora da bude

	terminiran nulom, kao u C-u.
print_nl	prelazak na novu liniju
read_int	čita celi broj sa tastature i smešta ga u EAX registar
read_char	čita jedan karakter i smešta njegov ASCII kod u EAX.

Pozivi I/O rutina se vrše standardnom CALL instrukcijom koja je pandan pozivu funkcije u višem programskom jeziku.

7. Ispravljanje grešaka (debugging)

Biblioteka asm_io sadrži i neke korisne rutine za ispravljanje grešaka. Ove rutine daju informacije o stanju bez modifikacije zatečenog stanja. To su, u stvari, makroi koji ispisuju vrednosti u registrima, memoriji, steku i i matematičkom koprocesoru:

dump_regs	štampa vrednosti u registrima u heksadekadnom zapisu. Takođe štampa vrednosti bitova u FLAG registru. Na primer, ako je ZF=1, štampa se ZF, a ako je ZF=0 ne štampa se ništa. Uzima jedan celobrojni argument koji služi samo za identifikaciju
dump_mem	Štampa vrednosti u regionu memorije u heksadekadnom i ASCII zapisu. Uzima tri argumenta, pri čemu je prvi identifikator (kao u dump_regs), drugi je adresa (labela), a poslednji broj 16-bitnih paragrafa za prikaz.
dump_stack	Štampa vrednosti na CPU steku. Jedinica organizacije steka je dupla reč (double word). Uzima tri argumenta; prvi je identifikator, drugi je broj duplih reči za prikaz ispod adrese u EBP registru, a treći broj duplih reči iznad adrese u EBP registru.
dump_math	štampa vrednosti u registrima matematičkog koprocesora. Jedini argument je identifikator

8. Prvi program

Danas je neuobičajeno razvijati samostalan program napisan u potpunosti u asembleru. Asembler se koristi za rutine koje su kritične za brzinu. Zato će i asemblerski programi koji slede poštovati ovu politiku.

Dakle, asemblerski program će se pozivati iz jednostavnog C programa čija je svrha isključivo pozivanje glavne asemblerske rutine. Više je prednosti ovakvog načina upotrebe. Prvo, C će prvo korektno startovati program u zaštićenom (*protected*) modu, pri čemu će svi segmenti i odgovarajući registri biti ispravno inicijalizovani. Drugo, i važnije, rutine C biblioteke će moći da se pozovu iz asemblera. Biblioteka asm_io koristi baš ovu prednost, tj. standardne C funkcije kao što je **printf**. Evo jednostavnog C drajverskog programa:

```
int main()
{
   int ret_status ;
   ret_status = asm_main();
   return ret_status ;
}
```

A evo i jednostavnog asemblerskog programa first.asm:

```
;
; file: first.asm
; Prvi asemblerski program. Ucitava dva broja i izracunava i stampa zbir
;
; Za prevodjenje programa:
; Koristeci Linux i gcc:
; nasm -f elf first.asm
; gcc -m32 -o first first.o driver.c asm_io.o
;

%include "asm_io.inc"
;
; inicijalizovani podaci se stavljaju u .data segment
;
segment .data
;
; Ovo su stringovi za poruke
;
prompt1 db    "Unesi broj: ", 0       ; ne zaboravi '0' terminator
prompt2 db    "Unesi drugi broj: ", 0
outmsg1 db    "Uneli ste ", 0
outmsg2 db    " i ", 0
outmsg3 db    ", njihov zbir je ", 0

;
; neinicijalizovani podaci se stavljaju u .bss segment
;
segment .bss
;
; Ovo su duple reci (double words) gde se ucitavaju dva broja
;
input1  resd 1
input2  resd 1

;
; kod se nalazi u.text segmentu
;
segment .text
        global  asm_main
asm_main:
        enter   0,0               ; ulazna rutina
        pusha

        mov     eax, prompt1      ; stampanje pitanja
        call    print_string

        call    read_int          ; ucitavanje broja
        mov     [input1], eax     ; snimanje na adresu input1

        mov     eax, prompt2      ; stampanje pitanja
        call    print_string

        call    read_int          ; ucitavanje broja
        mov     [input2], eax     ; snimanje na adresu input2

        mov     eax, [input1]     ; eax = dword na adresi input1
        add     eax, [input2]     ; eax += dword na adresi input2
        mov     ebx, eax          ; ebx = eax
        dump_regs 1               ; odstampaj vrednosti u registrima
        dump_mem 2, outmsg1, 1    ; dodstampaj vrednosti u memoriji
```

```
;
; stampanje izlazne poruke u seriji koraka
;
        mov     eax, outmsg1
        call    print_string      ; stmpanje prve poruke
        mov     eax, [input1]
        call    print_int         ; stampanje input1
        mov     eax, outmsg2
        call    print_string      ; stampanje druge poruke
        mov     eax, [input2]
        call    print_int         ; stampanje input2
        mov     eax, outmsg3
        call    print_string      ; stampanje trece poruke
        mov     eax, ebx
        call    print_int         ; stampanje zbira koji se nalazi u ebx
        call    print_nl          ; stampanje nove linije

        popa
        mov     eax, 0            ; vrati se u C program
        leave
        ret
```

Kao što se može videti, u **.data** segmentu se definišu isključivo podaci kojima se dodeljuje početna vrednost. U linijama koje slede, definiše se nekoliko stringova koji se štampaju pomoću C biblioteke pa moraju biti terminirani *null* karakterom (ASCII kod 0).

Neinicijalizovani podaci idu u **.bss** segment. To su podaci koji tek treba da se pročitaju sa tastature.

U **.text** segmentu nalazi se sam kod, dok **global** direktiva naređuje asembleru da rutina **asm_main** bude globalna kako bi C drajverski program mogao da je pozove. U asembleru, za razliku od C-a, promenljive podrazumevano imaju **interni opseg** (scope), dok ih **global** direktiva čini **eksternim**.

8.1 Proces kompajliranja i linkovanja

Da bi se ovaj program kompajlirao, potrebno je prevesti ga u objektni fajl, kao što to čini kompajler u nekom višem programskom jeziku. Naredba koja izvodi ovu operaciju je:

```
nasm -f elf first.asm
```

Linux koristi ELF (*Executable and Linkable Format*) format objektnih fajlova, pa se takav argument navodi u komandnoj liniji. Nakon izvođenja komande, dobija se **first.o** objektni fajl. U Windowsu je najrašireniji OMF format objektnog fajla.

Drajverski program napisan u C-u takođe se mora prevesti, i to standardnom komandom:

```
gcc -c [-m32] driver.c
```

Argument -c znači da se kod samo kompajlira, bez pokušaja da se linkuje. Argument -m32 je obavezan ukoliko se kompajliranje vrši na 64-bitnoj mašini, jer je mašinski kod koji se generiše 32-bitni.

Nakon što su svi kodovi iskompajlirani, tj. prevedeni u objektne fajlove (first.o, driver.o i asm_io.o), može se pristupiti linkovanju čiji je rezultat izvršni program:

```
gcc [-m32] -o first driver.o first.o asm_io.o
```

Time je izgradnja programa dovedena do kraja.

8.2 Kostur koda

Sledi kostur koda koji će nadalje biti upotrebljavan za sve asemblerske programe:

```
;
; file: kostur.asm
; Kostur asemblerskog programa
;

%include "asm_io.inc"
segment .data
;
; ovde idu podaci koji se inicijalizuju
;

segment .bss
;
; neinicijalizovani podaci se stavljaju u .bss segment
;

segment .text
        global  asm_main
asm_main:
        enter   0,0               ; ulazna rutina
        pusha

;
; U ovom delu ide kod koji nesto radi. Ne modifikovati kod
; unutar .text segmenta pre ili posle ovog komentara
;

        popa
        mov     eax, 0            ; vrati se u C program
        leave
        ret
```