Arhitektura računara 1

ASEMBLER - 3. termin

1. Kontrolne strukture

Jezici visokog nivoa sadrže strukture za kontrolu toka koje su takođe visokog nivoa, npr. *if* i *while* sa preduslovom i postuslovom. Asembler ne omogućava tako "kompleksne" strukture. Umesto toga, koristi se nepopularni *goto* za čitavu kontrolu toka programa.

Ako se *goto* koristi na neodgovarajući način, svaki program (pa i asemblerski) postaje nečitak i težak za održavanje. Uprkos tome, korišćenjem pravila strukturiranog programiranja, može se postići da čak i asemblerski kod bude strukturiran.

2. Poređenja

Kontrolne strukture odlučuju šta da rade bazirajući se na poređenju podataka. U asembleru, rezultat tog poređenja se čuva u FLAGS registru. 80x86 procesori koriste CMP instrukciju za izvođenje poređenja, dok se FLAGS registar menja u skladu sa rezultatom. CMP instrukcija je, u stvari, SUB (oduzimanje drugog od prvog operanda) , s tim što se rezultat oduzimanja nigde ne čuva.

Za neoznačene cele brojeve, dve zastavice (bita u FLAGS registru) su bitne, i to: ZF (*zero flag* - nula) i CF (*carry flag* - prenos). ZF se postavlja na 1 ako je rezultat operacije razlike 0. CF se u operaciji razlike koristi kada dođe do "pozajmice". Na primer:

```
cmp vleft, vright
```

Dakle, izračunava se razlika vleft - vright i FLAGS registar se postavlja prema rezultatu. Ako je razlika 0, tj. vleft=vright, tada se ZF postavlja na 1, dok je CF nula (nema pozajmice). Ako je vleft>vright, i ZF i CF su nule. Ako je vleft<vright, tada je ZF=0, a CF=1 (došlo je do pozajmice).

Ne treba zaboraviti da i druge instrukcije menjaju FLAGS registar, a ne samo CMP! Poređenje označenih brojeva je nešto komplikovanije, pa na ovom mestu neće biti diskutovano.

3. Instrukcije grananja

Instrukcije grananja služe za transfer izvršavanja na bilo koju tačku unutar programa. Drugim rečima, ove instrukcije igraju ulogu kao *goto* u nekim programskim jezicima. Postoje dve vrste ovih instrukcija - uslovne i bezuslovne. Bezuslovna instrukcija je beš kao *goto*, do skoka dolazi uvek. Uslovne instrukcije grananja mogu, ali ne moraju da izvrše grananje, zavisno od stanja FLAGS registra. Ako do grananja ne dođe, kontrola se daje prvoj sledećoj instrukciji.

Instrukcija bezuslovnog skoka je JMP. Njen jedini argument je *labela* instrukcije na koju se skače. Labelu asembler i linker zamenjuju konkretnom memorijskom adresom. Važno je napomenuti da instrukcija odmah posle JMP neće biti nikad izvršena, osim ako se neka druga instrukcija grana na nju.

Validne labele u kodnom segmentu (*.text*) se definišu svojim nazivom i dvotačkom, nakon čega sledi

instrukcija, kao npr:

```
while_granica:      mov     eax,[broj]
```

Neke od instrukcija uslovnog grananja date su u sledećoj tabeli. Kao svoj operand sve one uzimaju labelu, kao i JMP.

JZ	skok ako je ZF=1, zero, nula
JNZ	skok ako je ZF=0
JO	skok ako je ZF=1, overflow, izlaz iz opsega
JNO	skok ako je ZF=0
JS	skok ako je ZF=1, sign, znak
JNS	skok ako je ZF=0
JC	skok ako je ZF=1, carry, prenos
JNC	skok ako je ZF=0
JP	skok ako je ZF=1, parity, parnost nižih 8 bita
JNP	skok ako je ZF=0

Sledeći C-ovski pseudo kod:

```
if ( EAX == 0 )
      EBX = 1;
else
      EBX = 2;
```

u asembleru može da se napiše kao:

```
      cmp eax, 0              ; postavljanje zastavica (ZF=1 ako eax-0=0)
      jz then_blok            ; ako je ZF setovan, skoci na then_blok
      mov ebx, 2              ; ELSE deo IF-a
      jmp next                ; preskoci THEN deo
then_blok:
      mov ebx, 1              ; THEN deo IF-a
next:
```

Međutim, ostala poređenja, npr. <=, >= nije tako lako izvesti koristeći samo instrukcije iz tabele. Problem je što mora da se vodi računa o nekoliko zastavica istovremeno.

Na sreću, procesori tipa 80x86 poseduju instrukcije koje omogućavaju mnogo korisnički orijentisanije grananje. Te instrukcije, i to samo za neoznačne cele brojeve kao operande date su u sledećoj tabeli:

JE	skače ako vleft = vright
JNE	skače ako vleft != vright
JB, JNAE	skače ako vleft < vright
JBE, JNA	skače ako vleft <= vright
JA, JNBE	skače ako vleft > vright
JAE, JNB	skače ako vleft >= vright

Recimo, JB i JNAE su instrukcije sinonimi jer je:

$$x < y \Leftrightarrow \textbf{not}\,(x \geq y)$$

Evo kako bi se u asembler preveo sledeći C kod:

```
// C kod
if ( EAX >= 5 )
      EBX = 1;
else
      EBX = 2;
```

```
; asemblerski kod

      cmp eax, 5
      jae then_blok
      mov ebx, 2
      jmp next
then_blok:
      mov ebx, 1
next:
```

4. Prevođenje standardnih kontrolnih struktura

Ova sekcija data je kao pomoć programeru kod prevođenja standardnih struktura kontrole toka iz viših programskih jezika u asemblerski kod.

4.1 If klauzule

Sledeći pseudo kod:

```
if ( uslov )
      then blok ;
else
      else blok ;
```

može se implementirati ovako:

```
      ; kod za setovanje FLAG zastavica
      jxx   else_blok   ; jxx se postavlja tako da je if uslov false!
      ; kod then bloka
      jmp endif

else_blok:
      ; kod else bloka
endif:
```

U slučaju da nema *else* bloka, instrukcija uslovnog grananja skače na endif:

```
      ; kod za setovanje FLAG zastavica
      jxx   endif       ; jxx se postavlja tako da je if uslov false!
      ; kod then bloka
endif:
```

4.2 While petlje

Opšta forma *while* petlje je:

```
while ( uslov ) {
      telo petlje;
}
```

što se u asembleru implementira ovako:

```
while:
      ; kod za setovanje FLAG zastavica
      jxx endwhile      ; izabrati xx tako da je uslov==false
      ; telo petlje
      jmp while
endwhile:
```

4.3 Do-while petlje

Kod *do-while* petlje uslov se testira na kraju:

```
do {
      telo petlje;
} while ( uslov );
```

što u prevedenom obliku daje:

```
do:
      ; telo petlje
      ; kod za setovanje FLAG zastavica
      jxx do      ; izabrati xx tako da je uslov==true
```

4.4 For petlje

80x86 poseduje nekoliko instrukcija za implementaciju for petlji. Najpoznatija je instrukcija LOOP koja za brojač korsiti ECX registar koji dekrementira, pa ako je ECX!=0 skače na labelu koja se navodi kao argument.

Sledeći pseudo kod:

```
suma = 0;
for ( i =10; i >0; i -- )
      suma += i;
```

bi se u asembler preveo ovako:

```
      mov eax,0         ; eax je suma
      mov ecx, 10       ; ecx je i
loop_start:
      add eax, ecx
      loop loop_start
```

5. Primer: Određivanje da li je broj prost

```
;
; fajl: prost1.asm
; Program koji ispisuje da li je uneseni broj prost
;
; Da bi se napravio izvrsni fajl:
; nasm -f elf prost1.asm
; gcc -m32 -o prost1 prost1.o driver.c asm_io.o
;
; Program radi isto kao sledeci C program:
;#include <stdio.h>
;
;int main()
;{
;  unsigned int broj;
;  unsigned int cinilac;
;
;  printf ("Unesi broj : ");
;  scanf("%u", &broj);
;
;  cinilac = 2;
;  while ( cinilac < broj && broj%cinilac != 0 )
;      cinilac += 1;
;
;  if (broj==cinilac)
;      printf("Broj je prost\n");
;  else
;      printf("Broj nije prost\n");
;
;  return 0;
;}
;
;

%include "asm_io.inc"

segment .data
        poruka         db    "Unesi broj: ", 0
        poruka_nije            db      "Broj nije prost",0
        poruka_jeste   db      "Broj je prost",0


segment .bss
        broj           resd    1       ; broj koji se ispituje


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

        ;Ovde pocinje koristan kod
        mov       eax,  poruka
        call    print_string

        call    read_int             ; scanf("%u", &broj);
        mov     [broj], eax

        mov     ebx, 2               ; u registru ebx se cuva cinilac = 2;
while_cinilac:
        cmp     ebx, [broj]
        jnb     end_while_cinilac    ; if !(cinilac < broj)
        mov     eax,[broj]
        mov     edx,0
        div     ebx                  ; edx = edx:eax % ebx
        cmp     edx, 0
        je      end_while_cinilac    ; if !(broj % cinilac != 0)

        inc     ebx                  ; cinilac++;
        jmp     while_cinilac
end_while_cinilac:
        cmp       [broj], ebx
        je        then_blok                ; if (broj==cinilac) idi na then_blok
        mov     eax, poruka_nije     ; printf("Broj nije prost.\n")
```

```
        call    print_string
        call    print_nl
        jmp     end_if
then_blok:
      mov     eax, poruka_jeste      ; printf("Broj je prost.\n")
      call    print_string
      call    print_nl

end_if:
        popa
        mov     eax, 0               ; vrati se nazad u C
        leave
        ret
```

6. Primer: Nalaženje prostih brojeva manjih od N

Sledeći primer pokazuje kako se jednostavan primer dat u programskom jeziku C u celosti prevodi u asembler. Zadatak je da se pronađu i odštampaju svi prosti brojevi manji od broja koji se unosi sa tastature.

```
;
; fajl: prost.asm
; Program koji ispisuje proste brojeve manje od unetog broja
;
; Da bi se napravio izvrsni fajl:
; nasm -f elf prost.asm
; gcc -m32 -o prost prost.o driver.c asm_io.o
;
; Program radi isto kao sledeci C program:
;#include <stdio.h>
;
;int main()
;{
;  unsigned int broj;
;  unsigned int cinilac;
;  unsigned int granica;
;
;  printf ("Pronadji proste brojeve sve do : ");
;  scanf("%u", &granica);
;  broj = 2;   // Pocinje se od 2
;
;  while ( broj <= granica )
;  {
;    cinilac = 2;
;    while ( cinilac < broj && broj%cinilac != 0 )
;      cinilac += 1;
;    if ( broj == cinilac )
;      printf ("%d\n", broj);
;    broj += 1;
;  }
;
;  return 0;
;}
;

%include "asm_io.inc"

segment .data
       poruka         db      "Pronadji proste brojeve sve do: ", 0


segment .bss
       granica resd    1          ; granica koju unosi korisnik
       broj           resd    1       ; tekuci broj koji se ispituje



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

       ; Ovde pocinje koristan kod
       mov      eax,  poruka
       call    print_string

       call    read_int             ; scanf("%u", &poruka );
       mov     [granica], eax
       mov     dword [broj], 2      ; broj = 2;

while_granica:                      ; while ( broj <= granica )
       mov     eax,[broj]
       cmp     eax, [granica]
       jnbe    end_while_granica    ; skace ako je broj > granica

       mov     ebx, 2               ; u registru ebx se cuva cinilac = 2;
```

```
while_cinilac:
        cmp     ebx, [broj]
        jnb     end_while_cinilac    ; if !(cinilac < broj)
        mov     eax,[broj]
        mov     edx,0
        div     ebx                  ; edx = edx:eax % ebx
        cmp     edx, 0
        je      end_while_cinilac    ; if !(broj % cinilac != 0)

        inc     ebx                  ; cinilac++;
        jmp     while_cinilac
end_while_cinilac:
        cmp      [broj], ebx
        jne      end_if              ; if !(broj==cinilac) izadji iz if-a
        mov     eax,[broj]           ; printf("%u\n")
        call    print_int
        call    print_nl
end_if:
        add      dword [broj],1             ; broj+=1
        jmp     while_granica
end_while_granica:

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


-------------------------
```

Deljenje

Instrukcija DIV je instrukcija celobrojnog deljenja. Deljenik se smešta u kombinaciju registara EDX:EAX (64 bita ukupno), nakon čega se deli operandom instrukcije DIV. Rezultat se smešta u EAX, dok se ostatak u EDX.

Množenje

Slično tome, celobrojno množenje 32-bitnih celobrojnih vredosti se obavlja instrukcijom

`mul source`

gde se 32-bitna vrednost na koju pokazuje *source* množi sadržajem registra EAX, a 64-bitni rezultat se smešta u kombinaciju registara EDX:EAX.