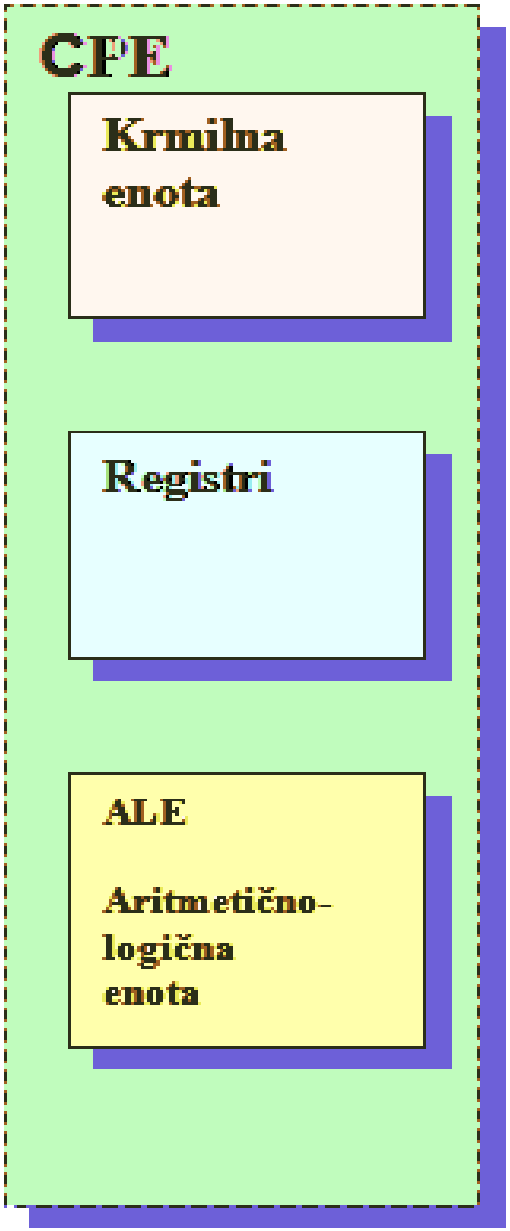
Uvod v programiranje v zbirnem jeziku

Programerjev pogled na zgradbo računalnika



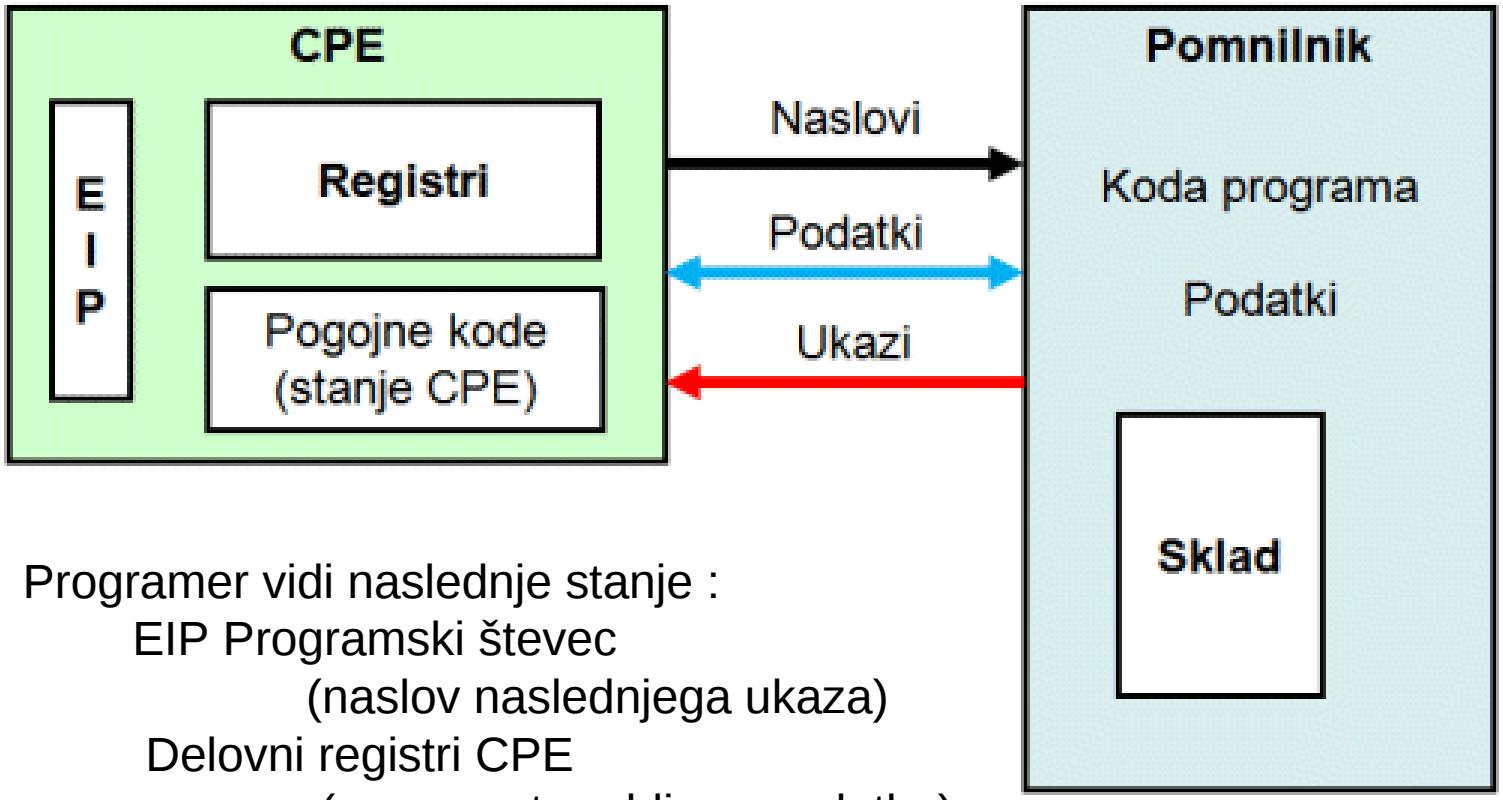
- krmili, nadzoruje in usklajuje delovanje vseh enot računalnika
- organizira prenos podatkov
- razpoznava in analizira ukaze
- skrbi za pravilno izvajanje ukazov

Hitre pomnilniške celice, nad katerimi ALU izvaja operacije. Tipično jih je do nekaj deset, vsak ima svoje ime.
(npr. akumulator(-ji), programski števec, statusni register)

Izvaja osnovne operacije:

- aritmetične: + – * /
- logične: & ∨ ¬
- primerjalne: > = < ≥ ≠ ≤

Primer: procesorji družine Intel x86



Programer vidi naslednje stanje :

EIP Programski števec
(naslov naslednjega ukaza)

Delovni registri CPE
(za pogosto rabljene podatke)

Pogojne kode (Statusni register)

- Pomnijo stanje predhodne aritmetične operacije
- Uporaba pri pogojnih programskih skokih

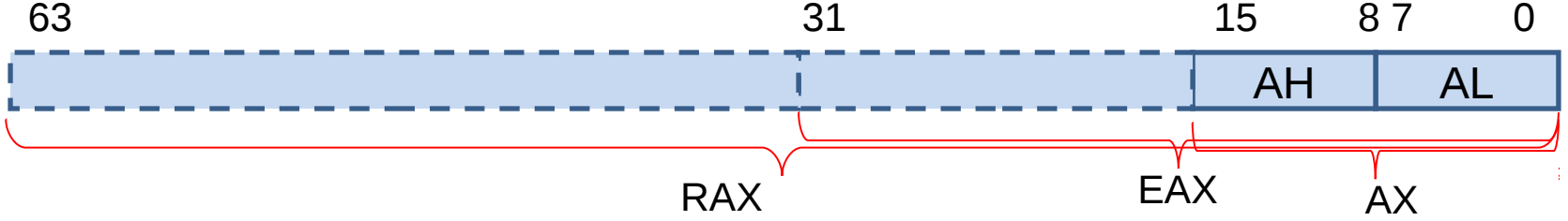
CPE Intel x86

Intelovi 16 bitni procesorji družine x86 so imeli:

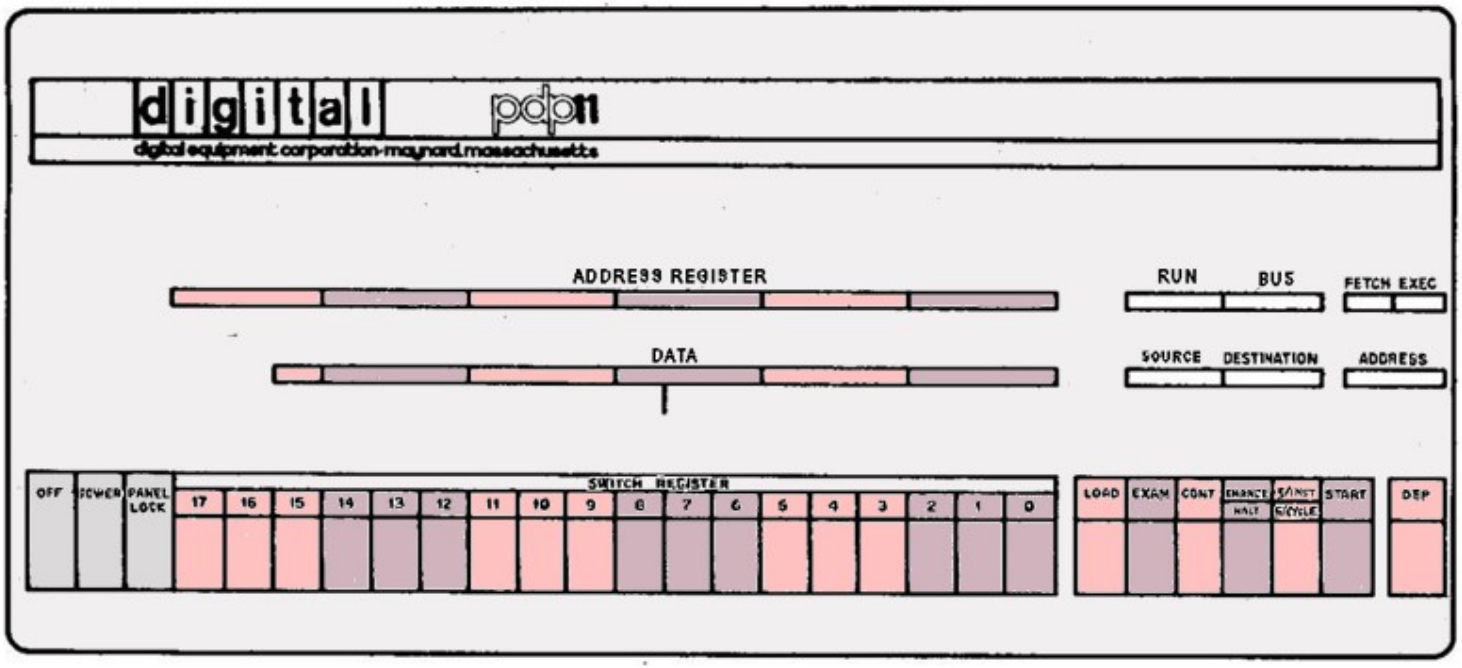
- 6 splošnih delovnih registrov (AX, BX, CX, DX, SI, DI),
- 2 registra za sklad (BP in SP),
- en 16-bitni statusni register (FLAGS),
- in 4 segmentne registre (CS, SS, DS, ES).

S pojavom 32 bitnih procesorjev (začenši z Intel 386 so 16-bitne registre podaljšali na 32 bitov in jih zato preimenovali v (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EFLAGS, EIP).

Podoben prehod so naredili pri uvedbi 64 bitnih procesorjev in uvedli nove, daljše 64 bitne registre z novimi imeni (RAX, RBX, RCX, RDX, RSI, RDI, RBP, RSP, RFLAGS, RIP). Tem pa so dodali še 8 splošnih 64-bitnih registrov (R8, R9, ..., R15).



Programiranje na strojnem nivoju



Nekateri osnovni ukazi vsakega računalnika

Zaenkrat si stvar poenostavimo in spoznajmo najbolj osnovne ukaze:

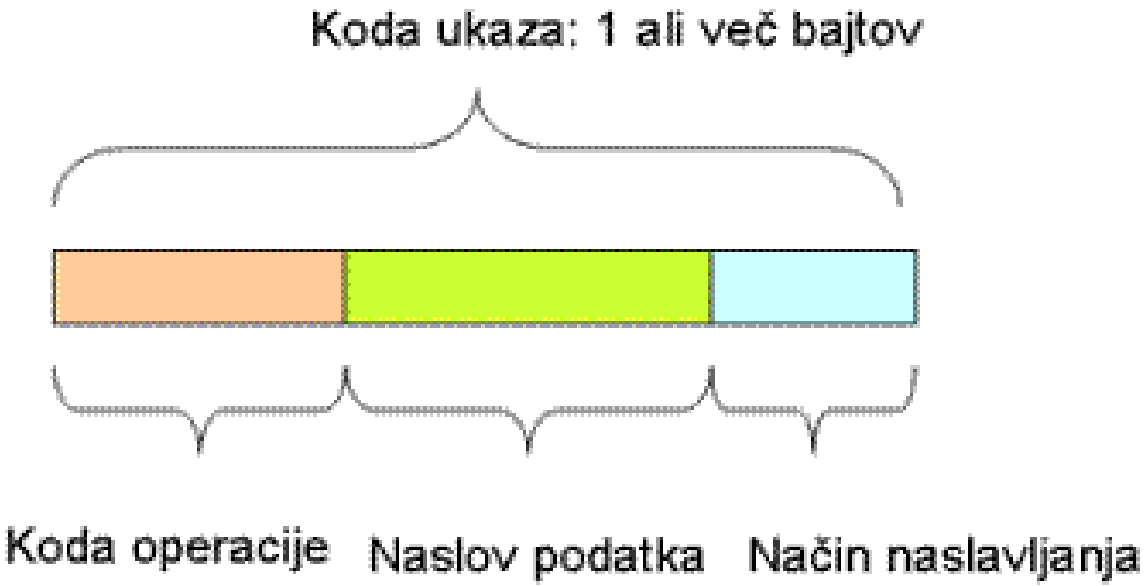
- Vpis dane vsebine v nek register (ali pomnilno lokacijo)
- Prepis vsebine neke lokacije ali registra v nek drug register
- Povečevanje ali zmanjševanje vsebin registrov (ki tako delujejo kot števci, uporabljamo jih lahko za indekse v polja ipd.)
- Izvedba aritmetičnih operacij med dvema registroma (seštevanje, odštevanje, množenje, deljenje..)/
- Izvedba logičnih operacij med dvema registroma (And, Or, Xor)
- (Druge ukaze bomo spoznali kasneje)
- S primernimi ukazi lahko tudi spremenimo vrednost samega programskega števca. To pa pomeni preskok na drug del programa. *Tak skok je lahko pogojen z rezultatom predhodne operacije (kar pomni statusni register procesorja).*

Format ukaza (splošno)

Kako CPE ugotovi, kakšno operacijo naj izvede in katere podatke naj pri tem uporabi?

Kaj moramo v takem ukazu pomniti:

- Kodo operacije (Kaj naj CPE naredi: vpis ali prepis podatka, aritmetična ali logična operacija ipd)
- Kje se morebitni podatek nahaja (naslov podatka ali podatek kot tak)
- Način naslavljanja morebitnega podatka

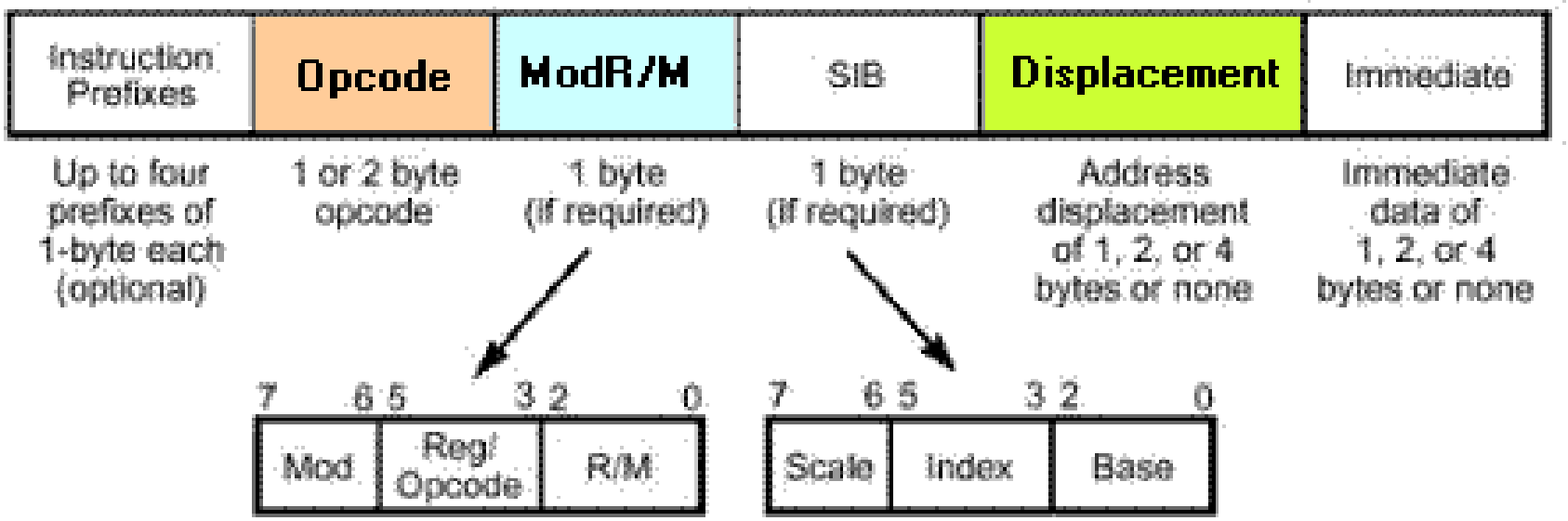


Format ukaza (bolj podrobno)

Koliko bitov potrebujemo za kateri delo ukaza in kje točno v kodi ukaza ležijo ti biti, je odvisno od takoimenovanega formata ukaza. V nekaterih primerih lahko z enim ukazom naslovimo tudi 2 podatka.

Žal je format med različnimi tipi računalnikov (bolje rečeno njihovih centralnih procesnih enot) različen.

Samo za občutek si poglejmo format ukazov pri Intelovih procesorjih Pentium:



Načini naslavljanja

- Registrsko naslavljanje (podatek je v nekem registru)
- Takojšnje naslavljanje (podatek je (podaljšan) del ukaza)
- Absolutno naslavljanje (v podaljšku ukaza je naslov pomnilniške lokacije s podatkom)
- Relativno naslavljanje (podatek je odmaknjen za toliko in toliko mest od ukaza. Odmik (displacement) pomnimo v "podaljšku" ukaza)
- Indeksno naslavljanje (naslov podatka določa vsebina nekega registra, kateri prištejemo odmik)

Še o naslavljanju

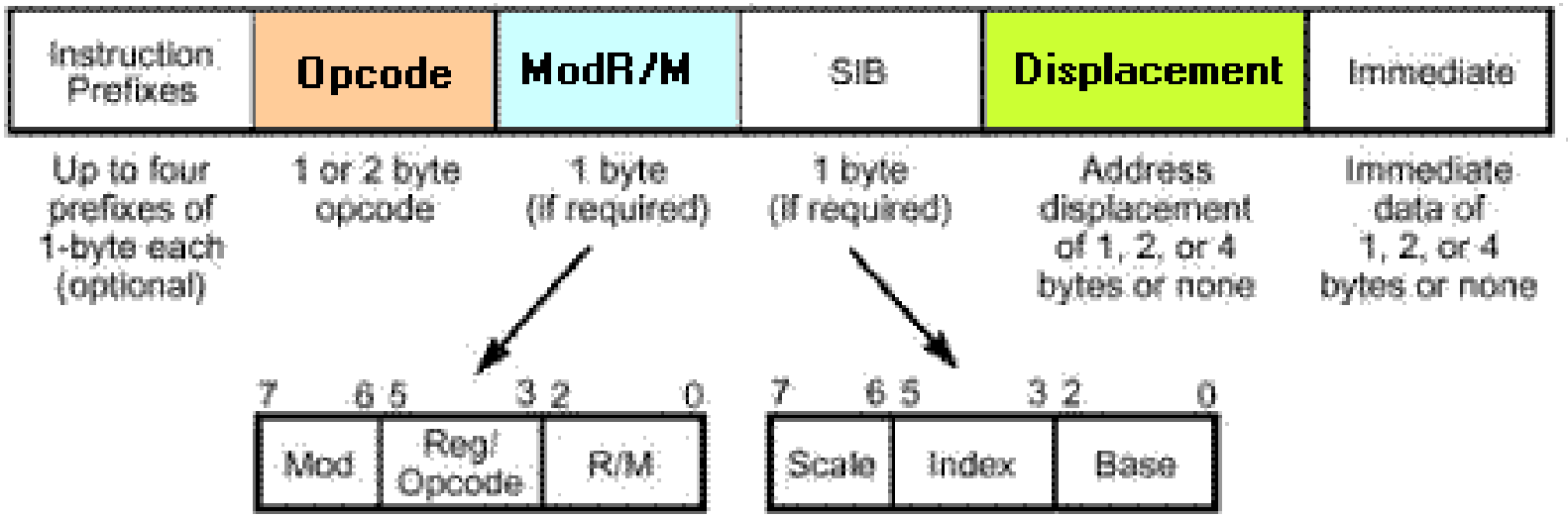
Možne so tudi različne kombinacije in dopolnitve. Včasih govorimo o "indirektnem " naslavljanju. To lahko pomeni, da vzamemo vsebino nekega registra in jo uporabimo kot naslov podatka v pomnilniku.

V nekaterih primerih lahko pred ali po naslavljanju registre avtomatsko povečujemo ali pomanjšujemo (preincrement, postdecrement,..). To je zelo uporabno za delo s polji. Tipičen primer registra, ki tako deluje, je tudi "kazalec na sklad " (stack pointer, SP).

Več o tem kasneje

Naslavljanje pri procesorjih Intel x86

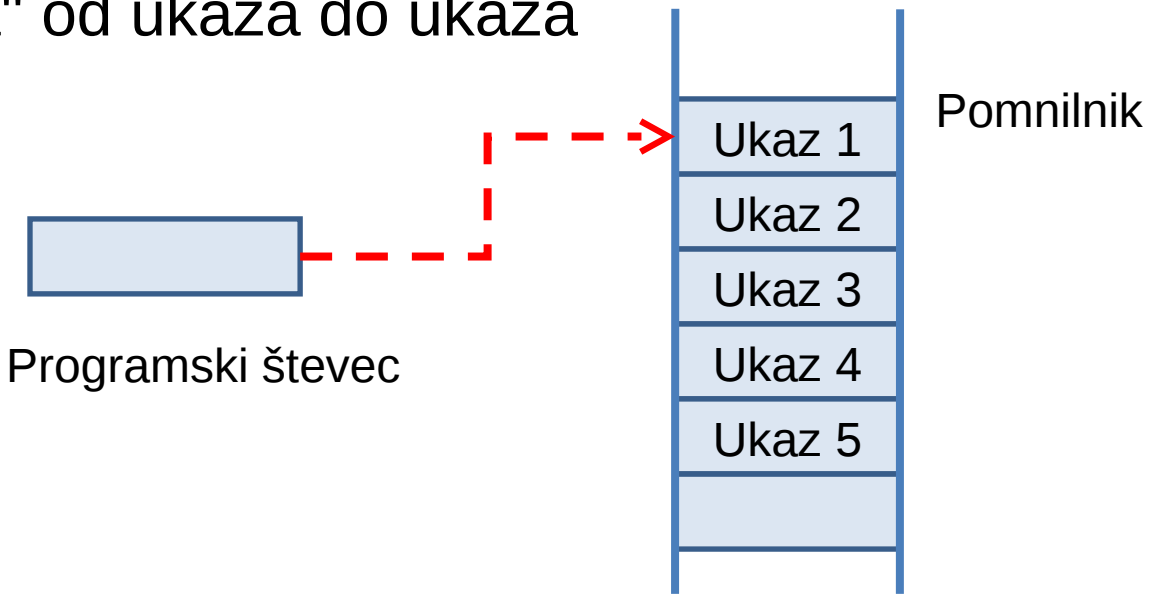
reg – registrski način naslavljanja, 32-bitni register
reg8 - registrski način naslavljanja, 8-bitni register
r/m – splošni, 32 bitni način naslavljanja
r/m8 - splošni, 8 bitni način naslavljanja
immed - 32-bitni takojšnji (immediate) naslov je del ukaza
immed8 - 8-bitni takojšnji (immediate) naslov je del ukaza
m – Simbol v ukazu je dejanski naslov



Program ... Zaporedje ukazov

Program računalnika predstavlja pravzaprav zaporedje ukazov v pomnilniku. Kode teh ukazov so shranjene v zaporednih lokacijah pomnilnika.

Eden od registrov računalnikove CPE (normalno mu pravimo programski števec) ima nalogo, da "kaže" na ukaz, ki naj se izvede. Program torej poteka tako, da programski števec "koraka" od ukaza do ukaza



Kaj je zbirni jezik

To je najnižji nivo programiranja, ki poenostavi izredno nepregledno in zamudno programiranje na nivoju strojnega jezika. Uporabljamo primitivne ukaze s kraticami (mnemoniki), ki nas v angleškem jeziku spominjajo na to, kar želimo od računalnika. Med ukazi v zbirnem jeziku (assembly language) in strojnem jeziku velja običajno kar razmerje 1:1

Primer:

```
mov ax,bx
```

pomeni, da vsebino registra AX prepišemo v register BX. In za to zadošča en ukaz na strojnem nivoju.

Indirektno naslavljanje (2)

- Indirektno naslavljanje dovoljuje, da uporabljamo registre kot “kazalce” na neko lokacijo v pomnilniku.
- Kako pokažemo v zbirnem jeziku, da nas ne zanima vsebina registra, pač pa vsebina lokacije, ki jo register naslavlja (kot kazalec)? Na primer z oglatimi oklepaji ([]).

Primer:

```
mov ax, [ebx]  ; (kar pomeni ax = *ebx   )
```

Naj ima na primer ebx vsebino 1000
V ax register ne bomo vpisali vsebine registra ebx (torej 1000)
pač pa vsebino pomnilniške lokacije 1000

Sklad (Stack)

- Večina CPE ima vgrajeno podporo za sklad (stack). Sklad je seznam, ki deluje po principu “zadnji noter – prvi ven”, kar spominja na sklad krožnikov.
- Ukaz PUSH doda podatek na vrh sklada, ukaz POP vzame en podatek s sklada
- Pri procesorjih Intel x86 imamo register ESP ki vsebuje naslov podatka, ki ga želimo vzeti s sklada (ESP je torej kazalec na sklad).
- Ukaz PUSH avtomatsko zmanjša vrednost ESP za 4 in da na lokacijo [ESP] dvojno besedo (torej 4 byte).
- Ukaz POP prebere s sklada dvojno besedo (double word) in nato avtomatsko zmanjša vrednost ESP za 4.

Uporaba sklada

- Sklad je primeren prostor za začasno shranjevanje podatkov (saj je registrov premalo).
- Sklad omogoča klice podprogramov, posredovanje parametrov tem programom, pomnenje lokalnih spremenljivk.
- Procesorji Intel x86 nudijo tudi ukaze PUSHA in POPA za vlaganje in prevzemanje vrednosti registrov EAX, EBX, ECX, EDX, ESI, EDI in EBP..

Intel x86 – ekvivalent kode v jeziku C in zbirnem jeziku

Primer

```
main (void)
{
    int a = 4, b = 2, c = 0;
    c = a + b;
}
```

Isto v zbirnem jeziku

code segment:

```
...
8048314 <main>:
8048314:    push   %ebp
8048315:    mov    %esp,%ebp
8048317:    sub    $0x18,%esp
804831a:    and    $0xfffffff0,%esp
804831c:    mov    $0x0,%eax
8048322:    sub    %eax,%esp
8048324:    movl   $0x4,0xfffffffc(%ebp)
804832b:    movl   $0x2,0xfffffff8(%ebp)
8048332:    movl   $0x0,0xfffffff4(%ebp)
8048339:    mov    0xfffffff8(%ebp),%eax
804833c:    add    0xfffffffc(%ebp),%eax
804833f:    mov    %eax,0xfffffff4(%ebp)
8048342:    leave
8048343:    ret
...
```

Vstavi vrednost 4 v spremenljivko na skladu
0xfffffffc = -(0xffffffff – 0xfffffffc) = **-0x4**

Kot pomnilniški naslov = EBP - 4

sklad:

0x0...

sub 24 (0x18) bytes
(doda prostor za 24 bytov)

sub 8 bytes

0
2
4
stari EBP
...
...
...

0xfff...

Akumulator za operande in rezultat

EAX: **0**

EBP: 0xffffff**f0** — Kazalec na podatek na skladu

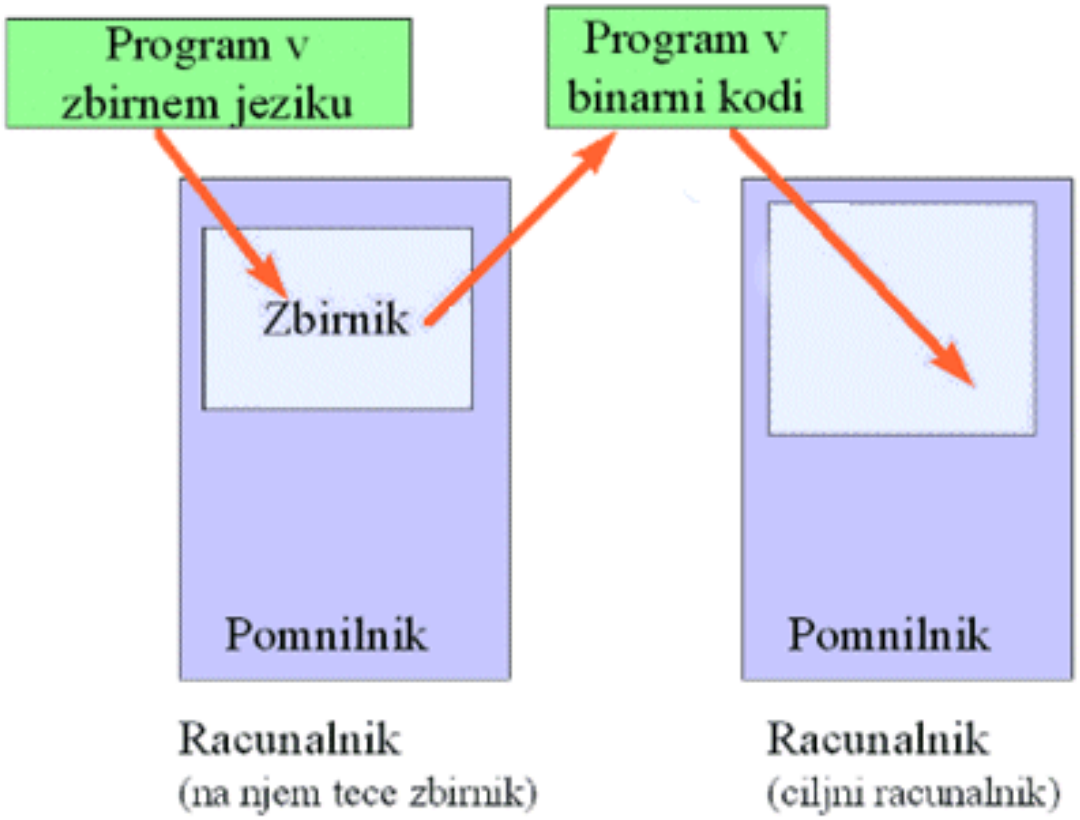
ESP: 0xffffff**f0** — Kazalec na sklad

EPI: 8048**33**8 — Kazalec na naslednji programski ukaz

Priprava programa

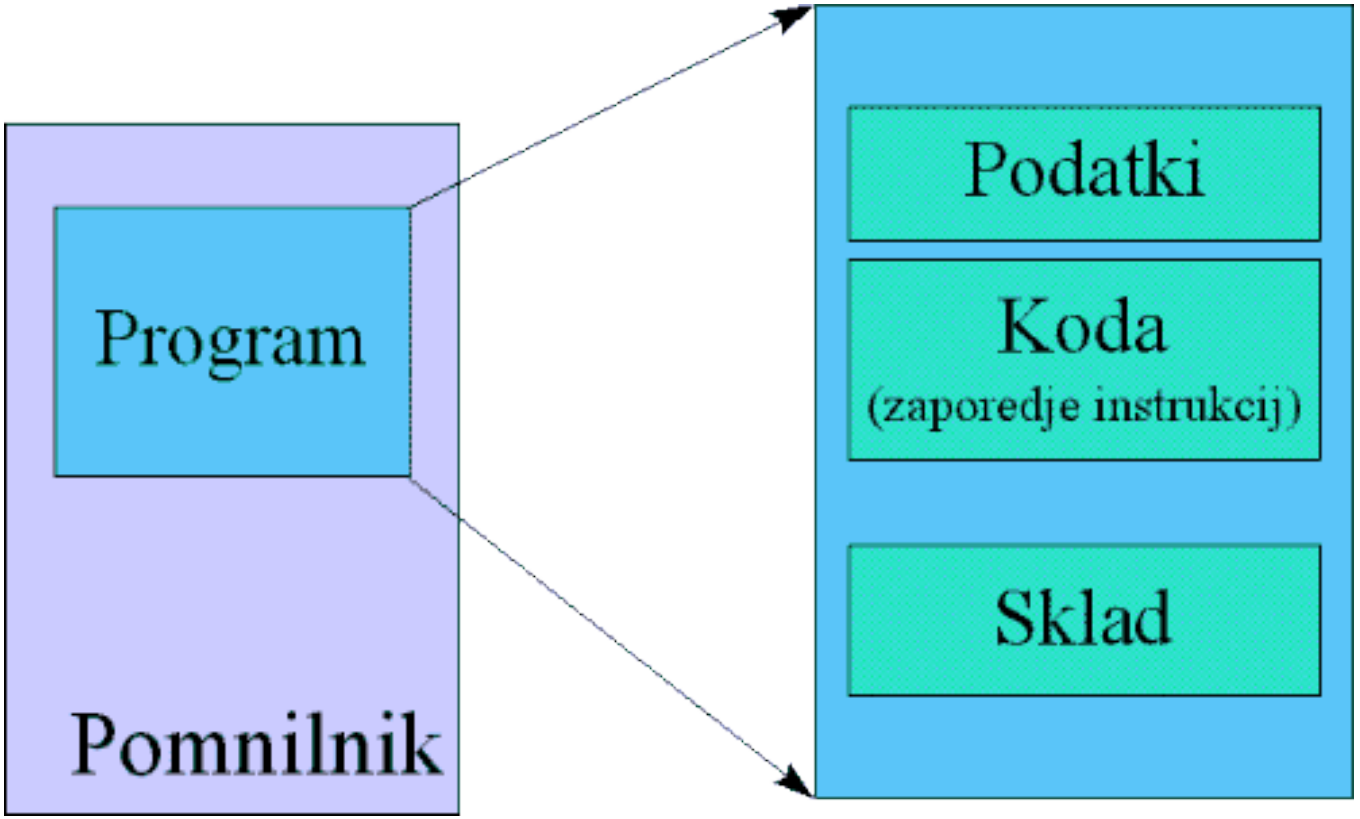
Pripraviti želimo program (v binarni kodi), ki naj bi tekel na našem računalniku.
Najprej s primernim urejevalnikom napisemo program v zbirnem jeziku.

Podobno, kot smo si pomagali z urejevalnikom za pisanje samega programa, si sedaj pomagajmo z zbirnikom (assembler). Zbirnik je program, ki k kodi



Izgled binarnega programa

Če binarni program pogledamo bolj podrobno, zasledimo (vsaj po eno) področje s podatki, programsko kodo ter sklad. Področij s kodo in s podatki je lahko tudi več in se med seboj lahko prepletajo (kar pa je, vsaj za programerje-začetnike,nevarno).



Listanje programa

Po prehodu čez zbirnik je to datoteka z izpisom programa (listing file), ki tipično vsebuje naslednje:

- Izvorno kodo programa
- Koristne naslove (kje je kaj)
- Prevedeno (objektno) kodo (strojni jezik)
- Imena segmentov (to je posebnost Intelove tehnologije)
- Simbole (imena spremenljivk, constant in procedur)

Primer listanja programa

```
00000000                                .code
00000000                            main PROC
00000000  B8  00010000                   mov eax,10000h
00000005  05  00040000                    add eax,40000h
0000000A  2D 00020000                     sub eax,20000h
0000000F  E8  00000000E                   call DumpRegs
                                        exit
00000014  6A 00                          * push +000000000h
00000016  E8 00000000E                    * call ExitProcess
0000001B                                 main ENDP
                                       END main
```

Več o tem kasneje

Kako deluje zbirnik?

S kakšnimi mnemoniki lahko povemo zbirniku, da želimo rezervirati prostor za podatke?

S kakšnimi instrukcijami povemo zbirniku, kakšna naj bo programska koda?

Zbirnik razporeja (alocira prostor za) podatke in kodo sekvenčno, tako kot bere vrstice v našem programu.

Ne pozabimo, da je zbirnik navaden program in ima kot tak svoje spremenljivke. Ena od teh je lokacijski števec. Ta se med zbiranjem povečuje za toliko besed, kolikor jih je potrebnih za posamezen podatek ali kodiran ukaz.

Primer preprostega programa

Delovanje računalnika lahko ponazorimo s pisanjem programa za izračun A+B-C. Ta izraz razstavimo na naslednje operacije:

1. A prepišemo v akumulatorski register,
2. vsebini akumulatorja prištejemo B,
3. nato odštejemo C
4. in končno ustavimo računalnik ter preverimo vsebino akumulatorja.

Kako bi izgledal tak program v zbirnem jeziku za Intel x86?

Program v zbirnem jeziku za Intel x86

```
TITLE Add and Subtract
; This program adds and subtracts 32-bit integers.
INCLUDE unility.inc
.code
Main  PROC
          moveax,10000h                ; EAX = 10000h
          add eax,40000h               ; EAX = 50000h
          sub eax,20000h                ; EAX = 30000h
          call DumpRegs                ; display registers
          exit
main  ENDP
         END main
```

Programske vrstice

Navodila (directive)

- Z njimi povemo zbirniku, kaj naj naredi
- Njim ne ustreza noben strojni ukaz samega računalnika
- Z njimi deklariramo procedure, podatke in model pomnenja (*).
- Različni zbirniki uporabljajo različne, čeprav podobne direktive

Ukazi (instrukcije)

- Jih zbirnik preslika v strojno kodo
- Med izvajanjem programa jih CPE spoznava in izvaja
- Stavek, ki opisuje ukaz, tipično vsebuje oznako (labelo), mnemonik, operande in komentar

Format programa v zbirnem jeziku

Zaradi večje preglednosti lahko te vrstice pišemo tudi s tabulatorjem. Prva kolona je oznaka (label) vrstice. Druga kolona vsebuje operacijo. Tretja kolona vsebuje morebitni operand, četrta kolona vsebuje komentar.

Vsaka vrstica (ki ni le komentar) vsebuje besedico, ki pove zbirniku, kaj naj na danem mestu naredi. Tem besedam, ki so v bistvu okrajšave angleških napotkov, pravimo mnemoniki.

Opazimo, da imajo nekateri stavki oznake (v našem primeru »main«). Zbirnik loči oznake od mnemonikov po tem, da se (vsaj v našem primeru) oznake obvezno začnejo v prvi koloni, mnemoniki pa morajo imeti pred seboj vsaj en presledek.

Za mnemoniki je lahko eden ali več argumentov, odvisno od vrste mnemonika. Če jih je več, so (običajno) ločeni z vejicami.

Pomen navodil (direktiv)

Zbirnik bi naš program v zbirnem jeziku prebral.

Navodila (direktive) povedo zbirniku, kakšno ime ima naš program (vrstica TITLE),

Povedo mu, da bo moral vključiti in uporabljati nekatere, vnaprej pripravljene podprograme (vrstica include unility.inc), ter da program vsebuje neko proceduro oziroma kodo (vrstici .code in PROC Main).

Primer listanja programa

```
00000000                                .code
00000000                            main PROC
00000000  B8  00010000                  mov eax,10000h
00000005  05  00040000                  add eax,40000h
0000000A  2D 00020000                   sub eax,20000h
0000000F  E8  00000000E                 call DumpRegs
                                      exit
00000014  6A 00                         * push +000000000h
00000016  E8 00000000E                  * call ExitProcess
0000001B                                main ENDP
                                      END main
```

pomen vrstice *»call DumpRegs».*

To namreč ni preprost strojni ukaz pač pa klic podprograma z imenom “DumpRegs”

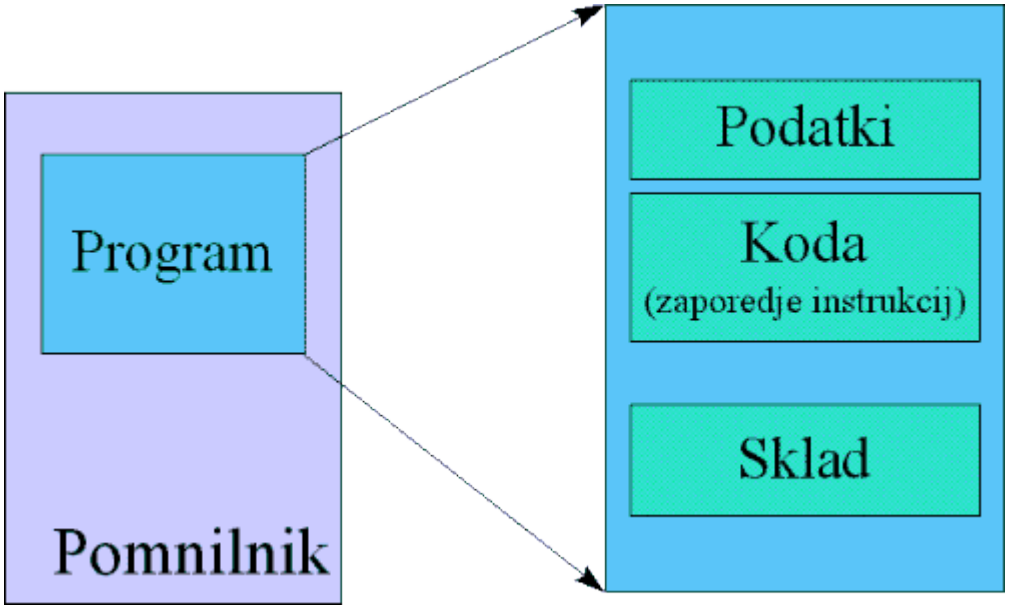
Zasilno si lahko namesto z izpisom pomagamo tako, da nam nekdo naredi podprogram za izpis vsebine registrov računalnika in ugotavljamo, ali je po določenih računalniških korakih njihova vsebina takšna, kot jo pričakujemo.

```
EAX=00030000  EBX=7FFDF000  ECX=00000101  EDX=FFFFFFFF
ESI=00000000  EDI=00000000  EBP=0012FFF0  ESP=0012FFC4
EIP=00401024  EFL=00000206  CF=0  SF=0  ZF=0  OF=0
```

Kako v zbirnem jeziku deklariramo podatke?

V prejšnjem primeru smo imeli podatke »integrirane« v sam program. Če pogledamo prejšnji primer listanja programa, bomo v strojni kodi kaj hitro odkrili podatke 1000, 4000 in 2000 (v šestnajstiškem formatu).

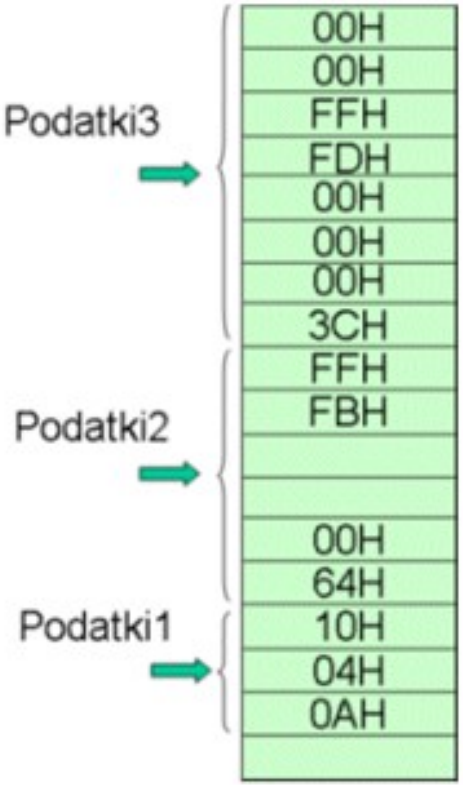
V splošnem pa so podatki v drugem delu pomnilnika kot sama programska koda. Spomnimo se naslednje slike:



Kako v zbirnem jeziku deklariramo podatke?

Za vsak podatek moramo povedati, kje je, kakšnega tipa je in kakšno vrednost ima. To naredimo s primernimi navodili zbirniku.
Za primer zbirnika za Intel, bi to lahko zgledalo tako:

```
Podatki1    DB    10,4,10H
Podatki2    DW    100,?,-5
Podatki3    DD    3*20,0FFFDH
```



Pomen navodil za podatke

Vidimo, da imajo te vrstice podobno obliko, kot ostale vrstice zbirnega jezika. Zaradi večje preglednosti smo mnemonike pobarvali rdeče. Operande pa modro. Pomen mnemonikov je naslednji:

DB (define byte) *Vsak operand zaseda en bajt*
DW (define word) *Vsak operand zasede eno besedo (dva bajta)*
DD (define double word) *Vsak operand zasede 2 besedi (4 bajte)*

Vsaki vrstici lahko po potrebi dodamo oznako (labelo), ki omogoča naslavljanje tako rezerviranih pomnilniških lokacij.

Operandi pri deklaracijah podatkov

Operandi so lahko številske konstante, podane v desetiškem ali šestnajstiškem sistemu (slednje imajo na koncu črko H).

Lahko pa tudi kar rezerviramo pomnilniško lokacijo in ne podamo njene vsebine (za to uporabimo namesto številske vrednosti znak ?).

Poleg tega lahko povemo, da potrebujemo večkratno dolžino (uporabno za rezervacijo polj).

Klicanje podprogramov (CALL, RET)

- Tipično kličemo podprograme z ukazom tipa CALL, povratek iz njih pa povzroči ukaz RET
- Pri klicu podprograma se vrednost programskega števca shrani na sklad, pri povratku pa se vrednost števca rekonstruira s prej pomnjeno vrednostjo na skladu
  - V tem smislu vidimo podobnost z ukazi PUSH in POP.
  - Uporaba sklada omogoča enostavno gnezdenje podprogramov
  - Na isti sklad lahko (za povratnim naslovom) pomnimo lokalne podatke.
  - Ne smemo pozabiti, da mora se mora število POP ukazov ujemati s številom PUSH ukazov, sicer bo to problem *(glej spodnji primer)*

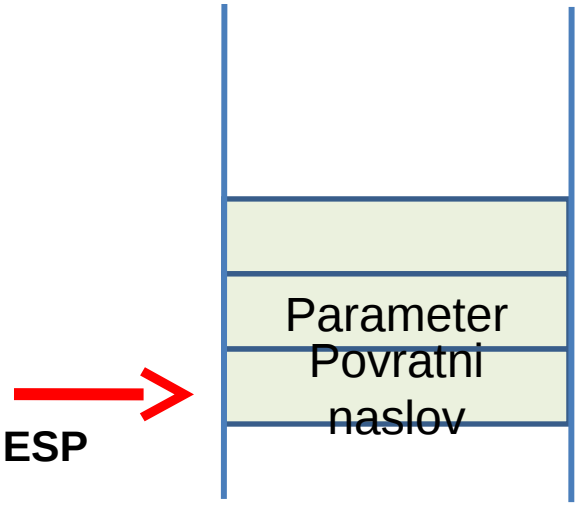
```
get_int:
        call read_int
        mov [ebx], eax
        push eax
        ret         ; Napaka: dobimo EAX vrednost, ne povratnega naslova!!
```

Posredovanje parametrov preko sklada

- Podprogramu lahko preko sklada posredujemo tudi parametre. Na klad jih moramo naložiti (z ukazi PUSH) pred klicem podprograma (z ukazom CALL)
- Če želimo, da podprogram spremeni vrednost neke spremenljivke, moramo podprogramu preko sklada posredovati naslov spremenljivke, ne pa njeno vrednost.

Primer:

Podprogramu posredujemo en parameter. Na skladu bo ta parameter(ali njegov naslov) pred povratnim naslovom kličočega programa



Primerjava podprograma v C in zbirnem jeziku

```
void calc sum( int n, int
    *sump )
{
  int i , sum = 0;

  for ( i=1; i <= n; i++ )
    sum += i;
  *sump = sum;
}
```

```
cal_sum:
        push    ebp
        mov     ebp, esp
        sub     esp, 4                  ; make room for local sum

        mov     dword [ebp - 4], 0      ; sum = 0
        mov     ebx, 1                  ; ebx (i) = 1
for_loop:
        cmp     ebx, [ebp+12]           ; is i >= n?
        jnle    end_for

        add     [ebp-4], ebx            ; sum += i
        inc     ebx
        jmp     short for_loop

end_for:
        mov     ebx, [ebp+8]            ; ebx = sump
        mov     eax, [ebp-4]            ; eax = sum
        mov     [ebx], eax              ; *sump = sum;

        mov     esp, ebp
        pop     ebp
        ret
```

Vhodno izhodne operacije

Niso tako enostavne, kot to velja za programiranje v višjih jezikih.
Programiranje v zbirnem jeziku pač ni temu namenjeno.

Če že moramo predvideti tudi vpis podatkov in izpis rezultatov, mora nekdo sprogramirati primerne podprograme, ki jih po potrebi kličemo.

Nekaj podobnega smo zasledili tudi v našem prejšnjem primeru.

Primer Hello world

Preverjanje in popravljanje programa

Uporabljamo poseben program – razhroščevalnik.
Vsak razhroščevalnik mora omogočati:

- Prikaz vrednosti v registrih in pomnilniku
- Vnašanje vrednosti v registre in pomnilnik
- Izvajanje zaporedja programskih ukazov
- Prekinitev programa, kjerkoli to želimo.

Lastnosti razhroščevalnika

Ena od pomembnih lastnosti razhroščevalnikov je sledenje programa. To pomeni:

- Koračno izvajanje programa ukaz za ukazom in sprotno sledenje vsebin registrov in pomnilniških lokacij.
- Nastavljanje prekinitvenih točk (breakpoints).

Razhroščevalniku povemo, pri katerem programskem ukazu naj se zaustavi. To pride v poštev predvsem pri programih, ki deloma potekajo v zankah in bi bilo koračno sledenje prezamudno.

- Nastavljanje pogojev za zaustavljanje (watchpoint). Zahtevamo, da se program zaustavi, ko pride na primer do spremembe vsebine nekega registra ali kakšne druge opazovane vrednosti.

Primer razhroščevalnika