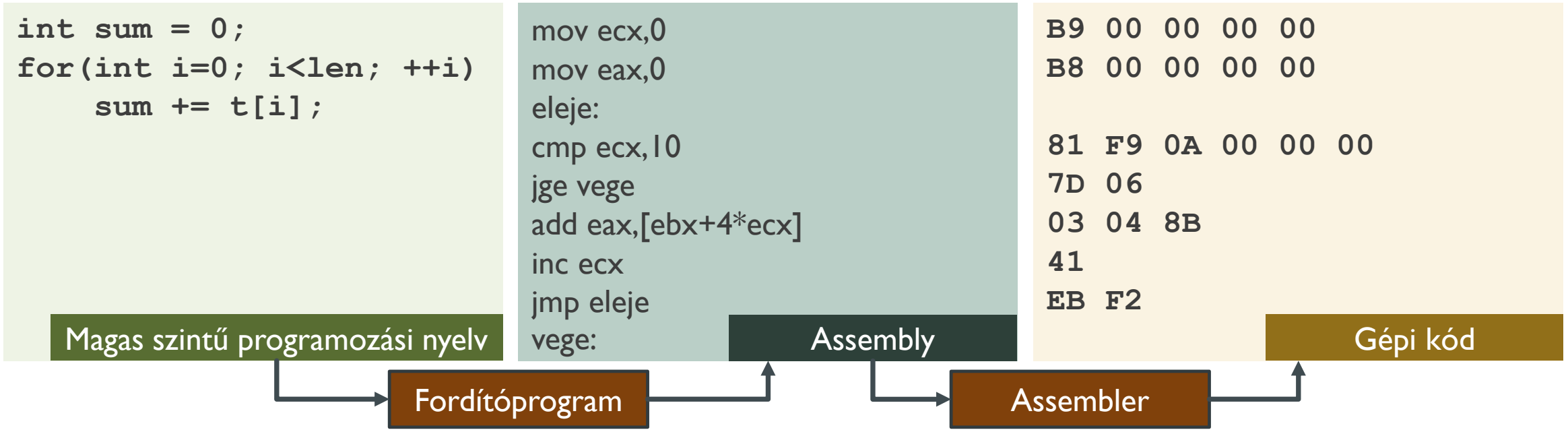
ASSEMBLY BEVEZETŐ

FORMÁLIS NYELVEK ÉS FORDÍTÓPROGRAMOK ALAPJAI

Dévai Gergely
ELTE

ASSEMBLY, ASSEMBLER

- Assembly: Alacsony szintű (hardverközeli) nyelvek
  - Utasításnevek (mnemonikok)
  - Regiszternevek
  - Címkék a memóriacímek azonosítására
- Assembler: Fordítóprogram assemblyről gépi kódra



EBBEN AZ ELŐADÁSBAN...

- 32 bites, x86-os architektúra
- NASM szintaxisú assembly
- Assembly és C programok
- Linux operációs rendszer

PÉLDA: ASSEMBLY FORRÁSFÁJL

addone.asm

```
global main
extern write_natural
extern read_natural

section .text
main:
call read_natural
inc eax
push eax
call write_natural
add esp,4
mov eax,0
ret
```

- Ebben a fájlban definiáljuk a ‘main’ címkét, és szeretnénk, hogy globálisan látható legyen.
- A ‘write_natural’ és ‘read_natural’ címkék máshol vannak definiálva, de itt szeretnénk használni őket.
- Itt kezdődik a programkód.
- Main címke: itt kezdődik a program ‘main’ függvénye.
- Meghívjuk a ‘read_natural’ függvényt, ami egy számot olvas be.
- Eggyel növeljük a visszatérési értékét.
- A verembe tesszük a megnövelt értéket: így lehet paraméterként átadni.
- Meghívjuk a ‘write_natural’ függvényt, ami kiírja a paraméterként kapott számot.
- Az imént a verembe tett paramétert kitöröljük onnan.
- Beállítjuk a ‘main’ függvény visszatérési értékét (0).
- Visszatérünk a függvényből.

PÉLDA: SEGÉDFÜGGVÉNYEK EGY C FORRÁSFÁJLBAN

io.c

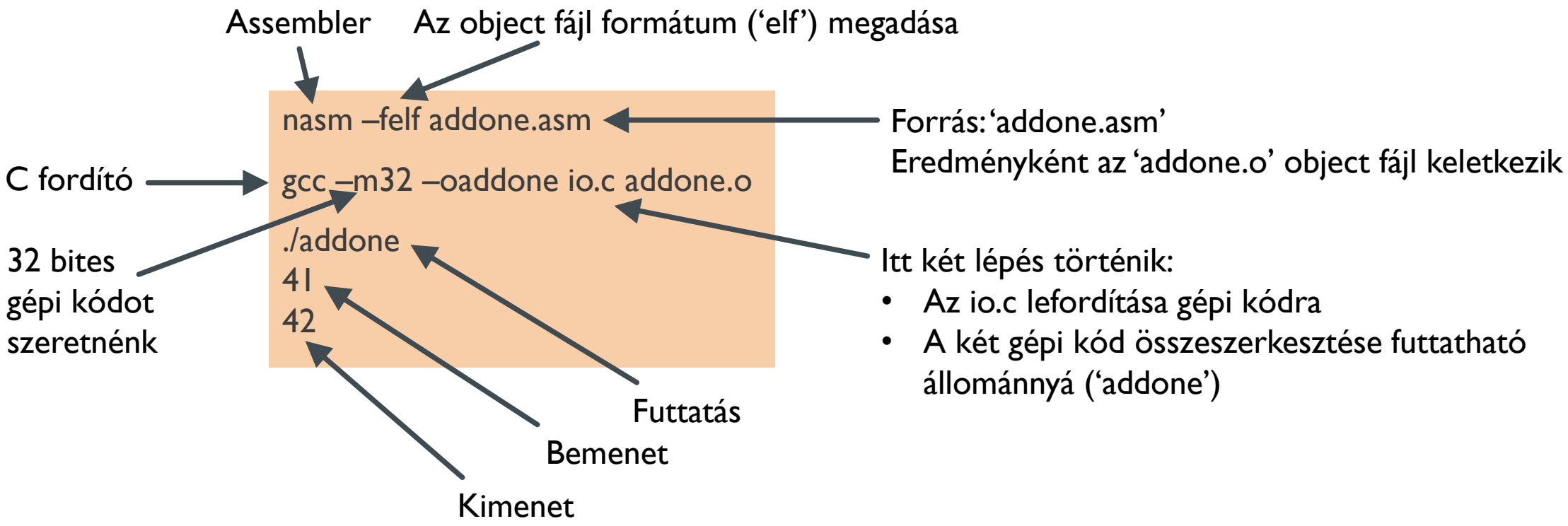
```
#include <stdio.h>

void write_natural(unsigned n) {
    printf("%u\n", n);
}

unsigned read_natural() {
    unsigned ret;
    scanf("%u", &ret);
    return ret;
}
```

- A ‘write_natural’ és ‘read_natural’ függvényeket az egyszerűség kedvéért C-ben írjuk meg.
- Lehetne tisztán assemblyben is, de az jóval bonyolultabb volna.

PÉLDA: FORDÍTÁS, SZERKESZTÉS, FUTTATÁS



ADATTÁROLÁS

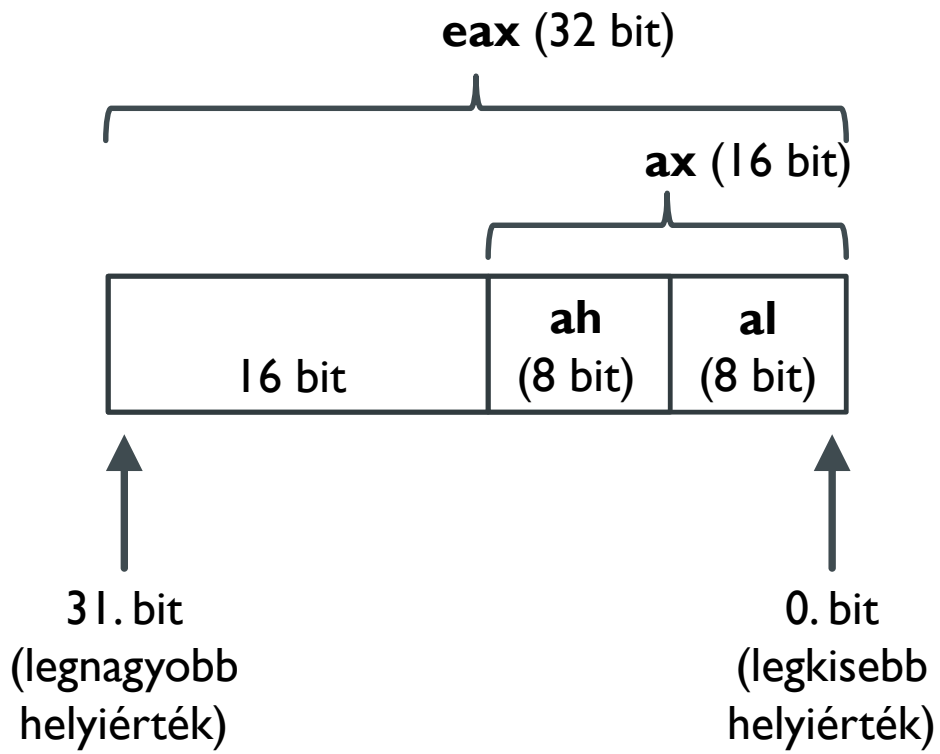
- Regiszterek
  - Gyors elérésű
  - Kicsi tárolókapacitás
  - Azoknak az adatoknak, amikkel éppen műveleteket végzünk
- Memória
  - Közepesen gyors
  - Közepes tárolókapacitás
  - Programkód, Statikus memória, Verem, Heap memória
- Háttértárak
  - Lassú
  - Nagy tárkapacitás

Ezekkel fogunk foglalkozni ebben az előadásban.

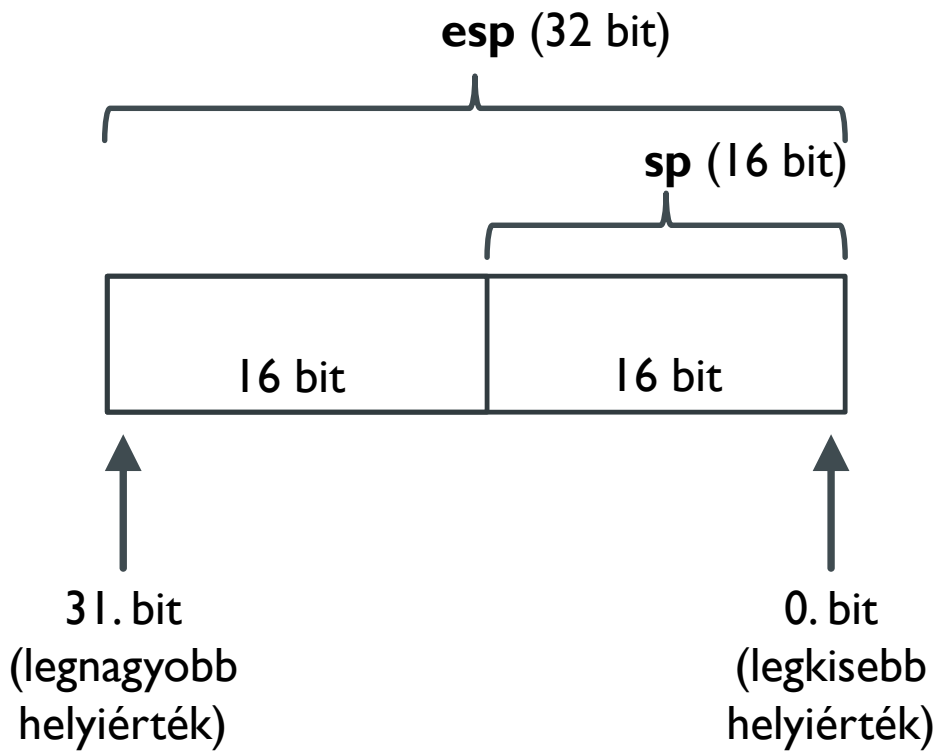
REGISZTEREK

- Általános célú regiszterek: **eax**, **ebx**, **ecx**, **edx**, **esi**, **edi**
  - Egymással felcserélhetően használhatók (többnyire).
- Veremmel kapcsolatos regiszterek:
  - **esp**: Stack pointer, a futási idejű verem tetejét mutatja
  - **ebp**: Base pointer, az éppen aktív eljáráshoz/függvényhez tartozó adatokra mutat a veremben
  - Ezeket csak a veremmel kapcsolatban érdemes használni.
- Egyéb regiszterek:
  - **eip**: Instruction pointer, a következő végrehajtandó utasításra mutat
  - **eflags**: Jelzőbitek gyűjteménye, pl. „az előző aritmetikai utasítás eredménye nulla volt-e?”
  - Ezeket nem érjük el közvetlenül, egyes utasítások olvassák/írják őket.

REGISZTEREK SZERKEZETE

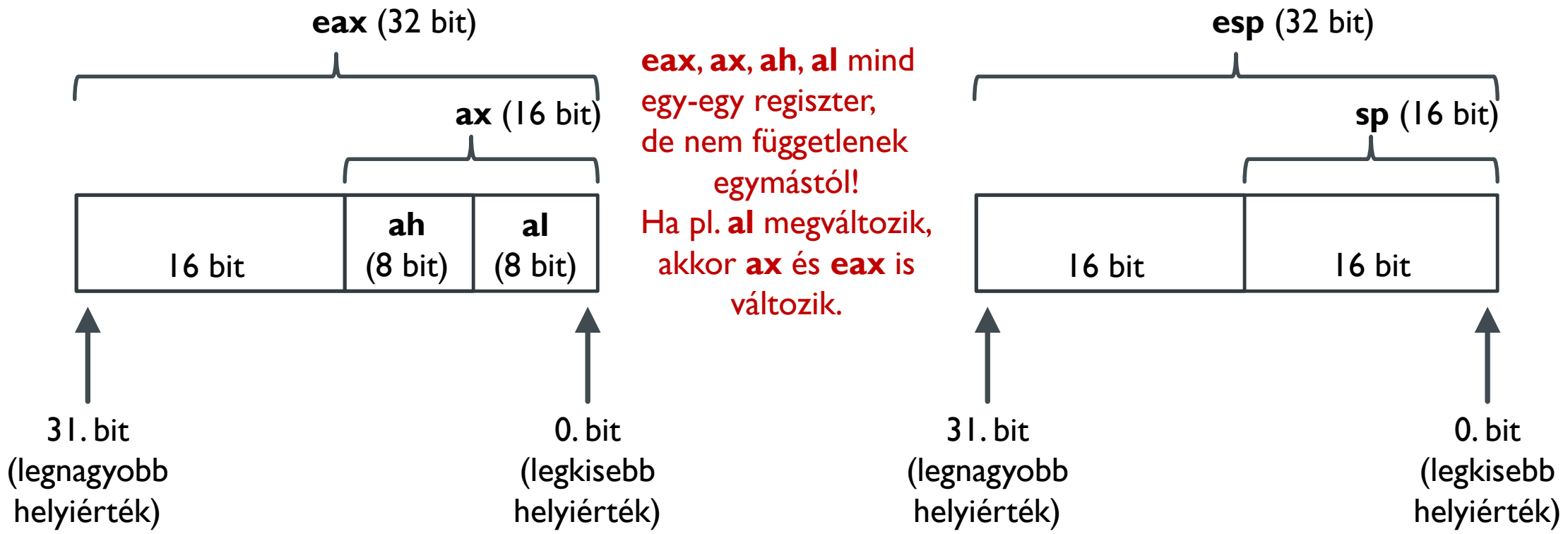


ebx, **ecx**, **edx** hasonlóan...



esi, **edi**, **ebp**, **eip** hasonlóan...

REGISZTEREK SZERKEZETE

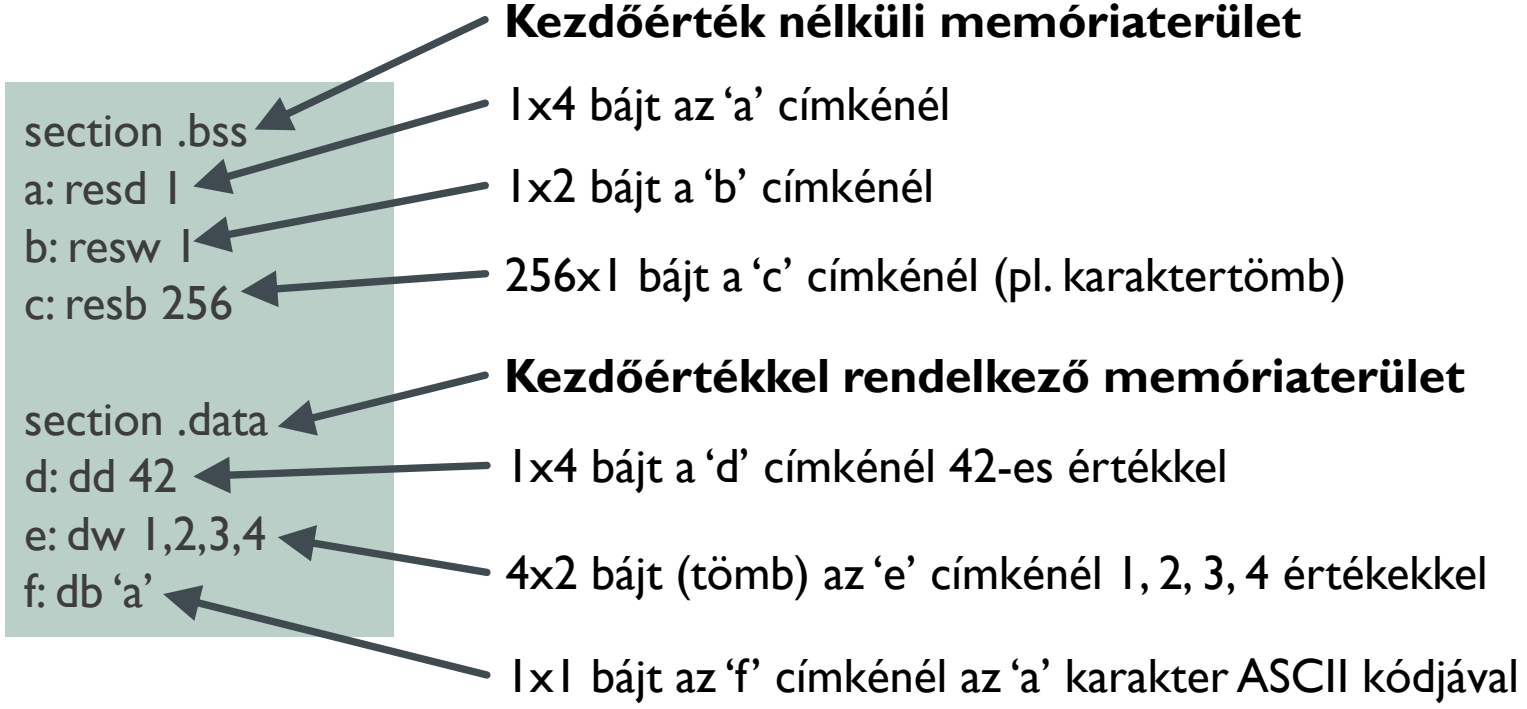


eax, **ax**, **ah**, **al** mind egy-egy regiszter, de nem függetlenek egymástól! Ha pl. **al** megváltozik, akkor **ax** és **eax** is változik.

ebx, **ecx**, **edx** hasonlóan...

esi, **edi**, **ebp**, **eip** hasonlóan...

MEMÓRIA: CÍMKEDEKLARÁCIÓK



- **resb:** "reserve byte"
  1 bájt kezdőérték nélkül
- **resw**: "reserve word"
  2 bájt kezdőérték nélkül
- **resd**: "reserve doubleword"
  4 bájt kezdőérték nélkül

- **db**: "define byte"
  1 bájt kezdőértékkel
- **dw**: "define word"
  2 bájt kezdőértékkel
- **dd**: "define doubleword"
  4 bájt kezdőértékkel

MEMÓRIAHIVATKOZÁSOK

byte [a]	1 bájt az ‘a’ címkétől kezdve.
word [a]	2 bájt az ‘a’ címkétől kezdve.
dword [a]	4 bájt az ‘a’ címkétől kezdve.

A hivatkozásba írhatók regiszterekből, címkékből és számliterálokból álló egyes kifejezések, pl.:

```
dword [a+4*ecx]
```

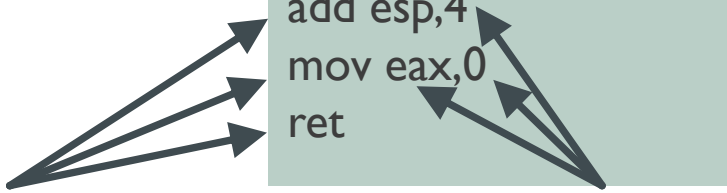
UTASÍTÁSOK

- Az utasításokat a section .text szakaszba írjuk.
- Minden utasításnak lehet címkéje, de nem kötelező.
  Lehet csak címkét tartalmazó sor is.
- Önálló assembly programoknál a program belépési pontja a _start címke,
  de megírhatjuk egy C program main függvényét is, ekkor main címkét használunk.
- Az utasításoknak van neve (mnemonikja) és operandusai.
- Megjegyzések: a ';' karaktertől a sor végéig.

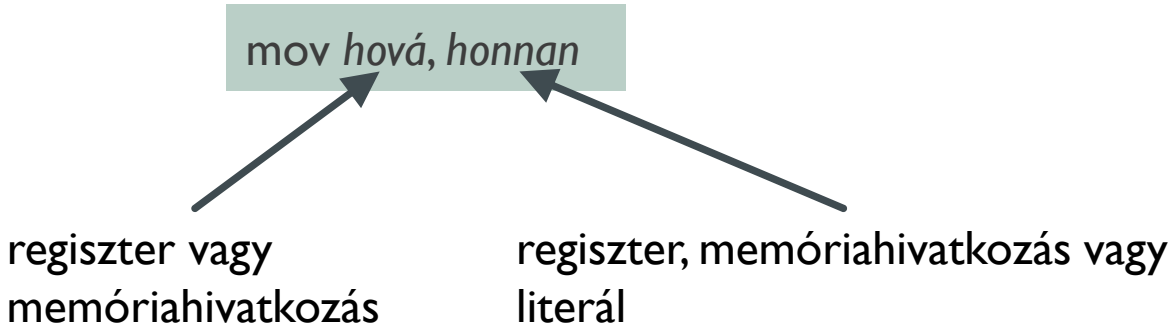
addone.asm

```
global main
extern write_natural
extern read_natural

section .text
main:
call read_natural
inc eax
push eax
call write_natural
add esp,4
mov eax,0
ret
```



ADATMOZGATÓ UTASÍTÁS: MOV



- Legfeljebb az egyik lehet memóriahivatkozás!
- Mekkora méretű adatot mozgatunk (1, 2 vagy 4 bájt)?
  - Ha valamelyik operandus regiszter, akkor abból kiderül.
  - Egyébként a memóriahivatkozásnál meg kell adni (byte, word, dword)
  - Azonos méretű operandusok kellenek
- A „mozgatás” valójában „másolást” jelent:
  A „honnan” nem változik meg

Példák:

```
mov eax,0
mov bx,ax
mov [lab1], al
mov dword [lab2], 987
mov ebx, [lab3]
```

Hibás:

```
mov byte [lab4], byte [lab5]
mov al, ax
```

ARITMETIKAI UTASÍTÁSOK: ÖSSZEADÁS, KIVONÁS

Összeadás:

add *mihez*, *mit*

Kivonás:

sub *miből*, *mit*

Operandusokra vonatkozó szabályok mint a mov utasításnál.

Inkrementálás (+1):

inc *mit*

Dekrementálás (-1):

dec *mit*

Operandus lehet regiszter vagy memóriahivatkozás.

Példák:

```
add eax, 10
add byte [a], 4
add bl, bl
```

```
sub ecx, ebx
sub ax, 40
sub bx, [b]
```

```
inc word [c]
inc edx
```

```
dec dword [d]
dec dl
```

Hibák:

```
add 10, eax
add byte [a], ax
```

```
sub word [b], [c]
```

```
inc 10
```

```
dec
```

ARITMETIKAI UTASÍTÁSOK: SZORZÁS, MARADÉKOS OSZTÁS

Szorzás:

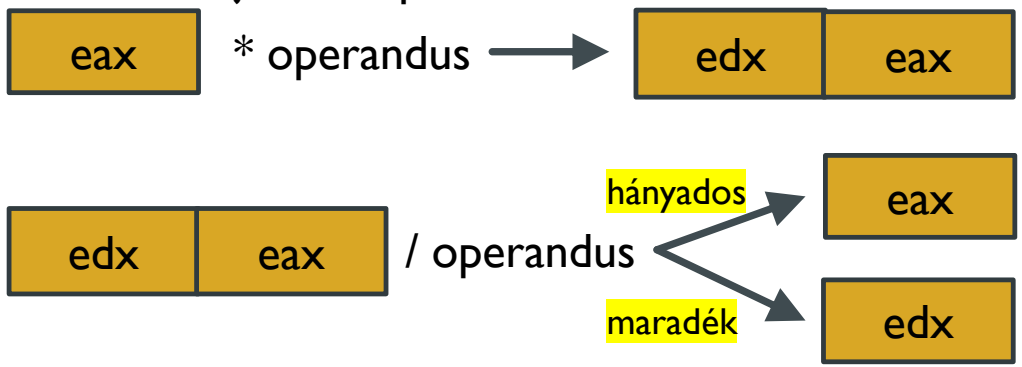
```
mul mivel
```

Maradékos osztás:

```
div mivel
```

Operandus lehet regiszter vagy memóriahivatkozás.

Működés, ha 4 bájtos az operandus:



Példa:

```
mov eax, 5
mov ebx, 6
mul ebx
; itt eax == 30, edx == 0
mov ecx, 4
div ecx
; itt eax == 7, edx == 2
```

Megjegyzés:
A ‘mul’ és ‘div’ előjel nélküli egész számoknak tekinti az operandusait.
Előjeles számok esetén ‘imul’ és ‘idiv’ használandó.

BITMŰVELETEK

Bitenkénti „és”:

```
and mihez, mit
```

Bitenkénti „vagy”:

```
or mihez, mit
```

Bitenkénti „kizáróvagy”:

```
xor mihez, mit
```

Operandusokra vonatkozó szabályok mint a mov utasításnál.

Bitenkénti „tagadás”:

```
not mit
```

Operandus lehet regiszter vagy memóriahivatkozás.

Példák:

```
; ha itt al == 15 és bl == 17
and al, bl
; akkor itt al == 1
```

$$\begin{array}{r} 00001111_b \\ \text{and } \underline{00010001_b} \\ 00000001_b \end{array}$$

```
; ha itt al == 15 és bl == 17
or al, bl
; akkor itt al == 31
```

$$\begin{array}{r} 00001111_b \\ \text{or } \underline{00010001_b} \\ 00011111_b \end{array}$$

```
; ha itt al == 15 és bl == 17
xor al, bl
; akkor itt al == 30
```

$$\begin{array}{r} 00001111_b \\ \text{xor } \underline{00010001_b} \\ 00011110_b \end{array}$$

```
; ha itt al == 15
not al
; akkor itt al == 240
```

$$\begin{array}{r} \text{not } \underline{00001111_b} \\ 11110000_b \end{array}$$

UGRÁS

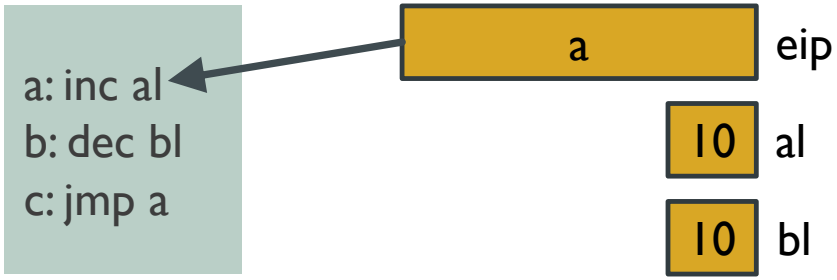
jmp *hová*

Operandusnak mi címkéket fogunk használni
(egyébként lehet regiszter és memóriahivatkozás is).

UGRÁS

jmp *hová*

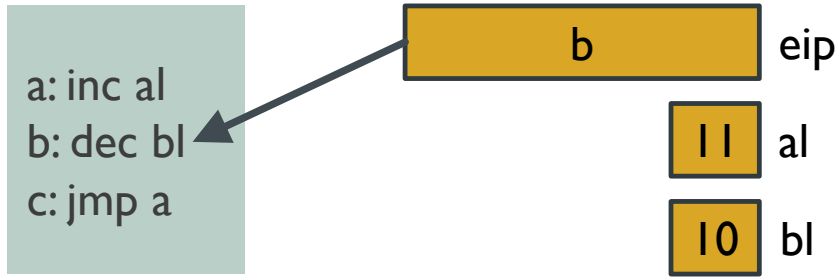
Operandusnak mi címkéket fogunk használni
(egyébként lehet regiszter és memóriahivatkozás is).



UGRÁS

jmp *hová*

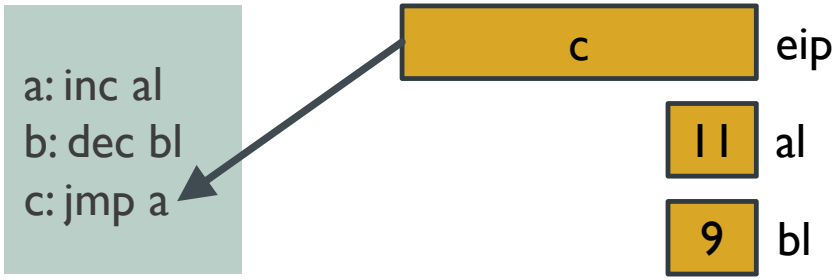
Operandusnak mi címkéket fogunk használni
(egyébként lehet regiszter és memóriahivatkozás is).



UGRÁS

jmp *hová*

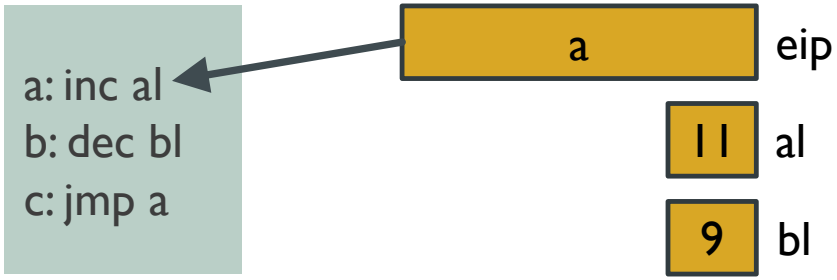
Operandusnak mi címkéket fogunk használni
(egyébként lehet regiszter és memóriahivatkozás is).



UGRÁS

jmp *hová*

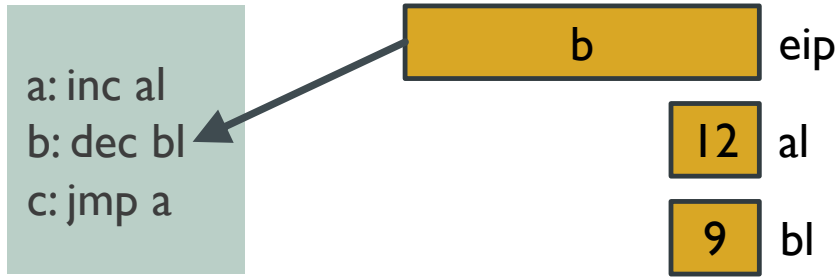
Operandusnak mi címkéket fogunk használni
(egyébként lehet regiszter és memóriahivatkozás is).



UGRÁS

jmp *hová*

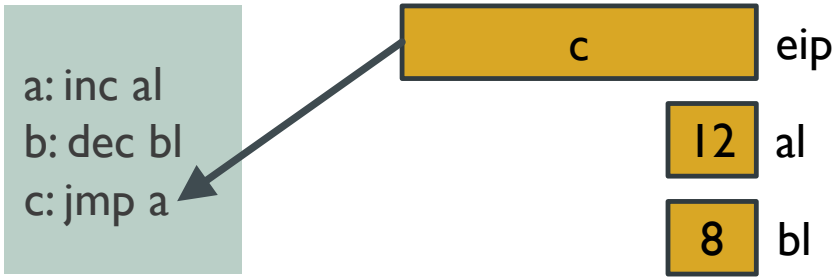
Operandusnak mi címkéket fogunk használni
(egyébként lehet regiszter és memóriahivatkozás is).



UGRÁS

jmp *hová*

Operandusnak mi címkéket fogunk használni
(egyébként lehet regiszter és memóriahivatkozás is).

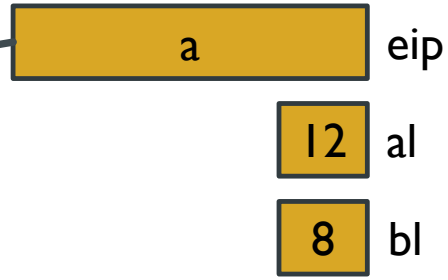


UGRÁS

```
jmp hová
```

Operandusnak mi címkéket fogunk használni
(egyébként lehet regiszter és memóriahivatkozás is).

```
a: inc al
b: dec bl
c: jmp a
```



FELTÉTELES UGRÁSOK

Összehasonlítás:

cmp *mit, mihez*

Operandusokra vonatkozó szabályok mint a mov utasításnál.

Ugorj, ha egyenlő:

je *hová*

Ugorj, ha nem egyenlő:

jne *hová*

Ugorj, ha kisebb:

jb *hová*

Operandus lehet regiszter vagy memóriahivatkozás.

A feltételes ugrások előtt rendszerint összehasonlítás (cmp) áll. Annak eredményétől függ, hogy történik-e ugrás.

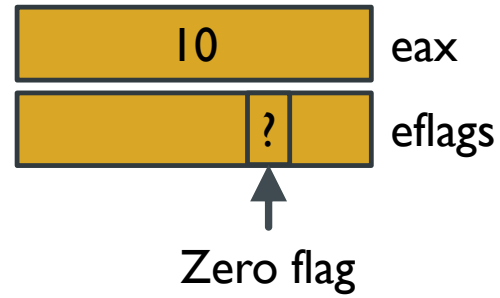
- **je**: jump if **equal** – ugorj, ha egyenlő
- **jne**: jump if **not equal** – ugorj, ha nem egyenlő
- **jb**: jump if **below** – ugorj, ha kisebb
  = **jnae**: jump if **not above or equal** –
  ugorj, ha nem nagyobb egyenlő
- **jnb**: jump if **not below** – ugorj, ha nem kisebb
  = **jae**: jump if **above or equal** –
  ugorj, ha nagyobb egyenlő
- **ja**: jump if **above** – ugorj, ha nagyobb
  = **jnbe**: jump if **not below or equal** –
  ugorj, ha nem kisebb egyenlő
- **jna**: jump if **not above** – ugorj, ha nem nagyobb
  = **jbe**: jump if **below or equal** –
  ugorj, ha kisebb egyenlő

Ezek előjel nélküli egész számokat feltételeznek!

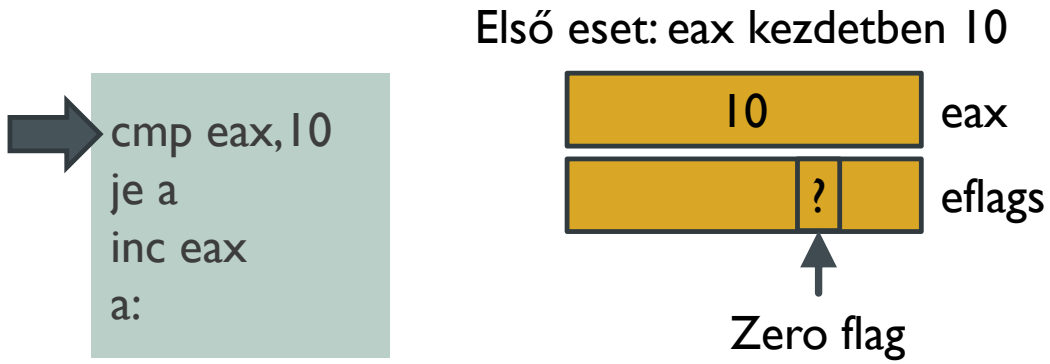
FELTÉTELES UGRÁSOK MŰKÖDÉSE

```
cmp eax,10
je a
inc eax
a:
```

Első eset: eax kezdetben 10

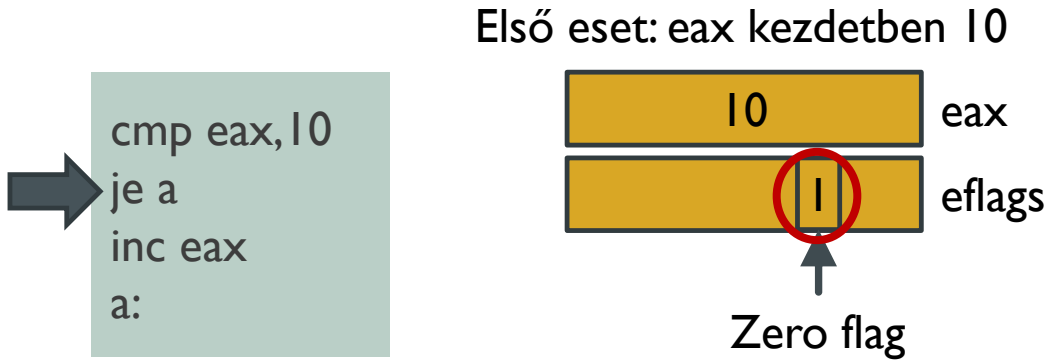


FELTÉTELES UGRÁSOK MŰKÖDÉSE



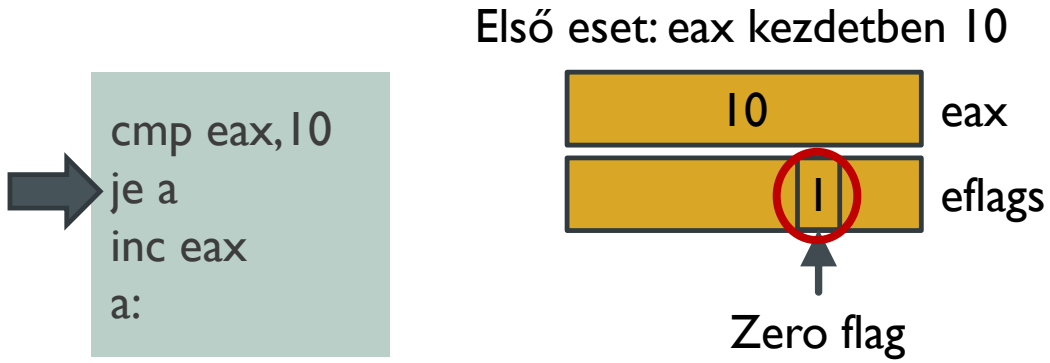
- A **cmp** utasítás kivonást végez, de nem tárolja el az eredményt, hanem annak előjele alapján jelzőbiteket állít át az **eflags** regiszterben.
- Ha a kivonás eredménye 0 (azaz az összehasonlított értékek egyenlők), akkor a **zero flag** 1 lesz, különben 0.

FELTÉTELES UGRÁSOK MŰKÖDÉSE



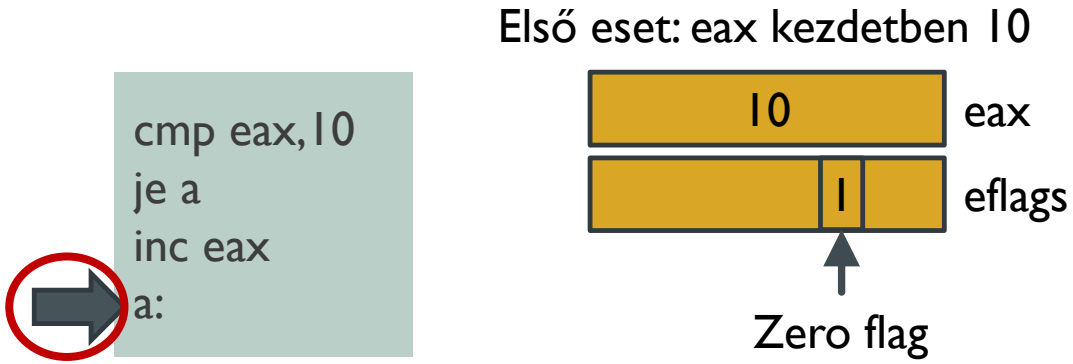
- A **cmp** utasítás kivonást végez, de nem tárolja el az eredményt, hanem annak előjele alapján jelzőbiteket állít át az **eflags** regiszterben.
- Ha a kivonás eredménye 0 (azaz az összehasonlított értékek egyenlők), akkor a **zero flag** 1 lesz, különben 0.

FELTÉTELES UGRÁSOK MŰKÖDÉSE



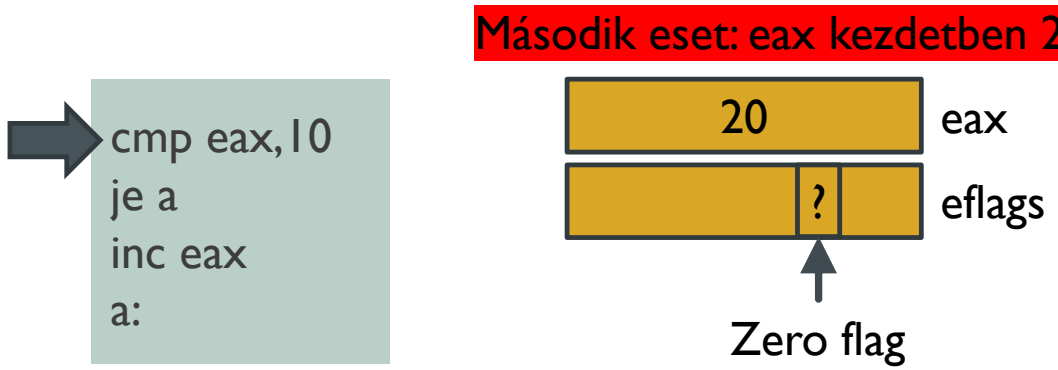
- A **cmp** utasítás kivonást végez, de nem tárolja el az eredményt, hanem annak előjele alapján jelzőbiteket állít át az **eflags** regiszterben.
- Ha a kivonás eredménye 0 (azaz az összehasonlított értékek egyenlők), akkor a **zero flag** 1 lesz, különben 0.
- A feltételes ugró utasítások a jelzőbiteket figyelik.
- A **je** akkor ugrik, ha a **zero flag** 1, egyébként a következő utasításra kerül a vezérlés.

FELTÉTELES UGRÁSOK MŰKÖDÉSE



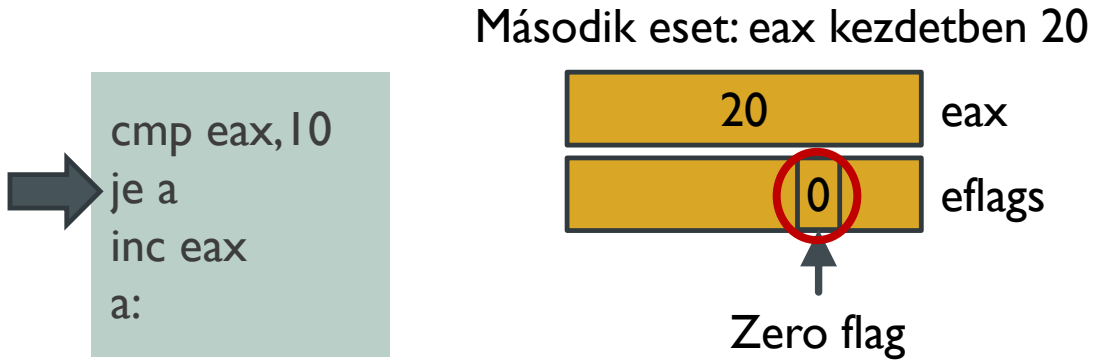
- A **cmp** utasítás kivonást végez, de nem tárolja el az eredményt, hanem annak előjele alapján jelzőbiteket állít át az **eflags** regiszterben.
- Ha a kivonás eredménye 0 (azaz az összehasonlított értékek egyenlők), akkor a **zero flag** 1 lesz, különben 0.
- A feltételes ugró utasítások a jelzőbiteket figyelik.
- A **je** akkor ugrik, ha a **zero flag** 1, egyébként a következő utasításra kerül a vezérlés.

FELTÉTELES UGRÁSOK MŰKÖDÉSE



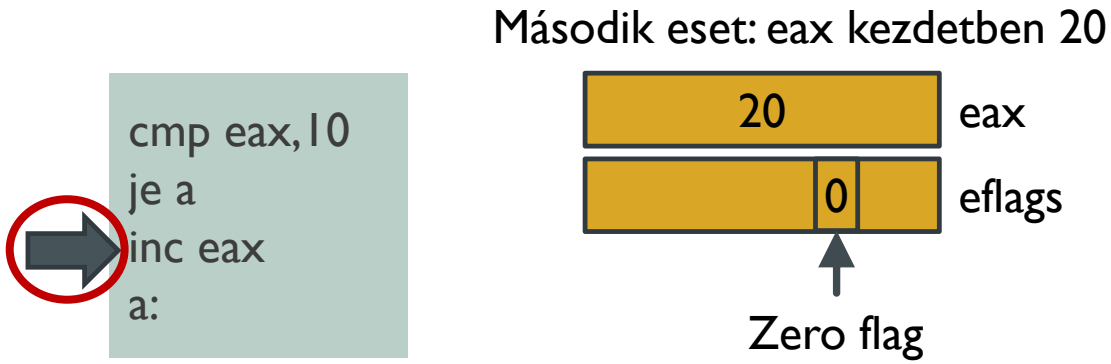
- A **cmp** utasítás kivonást végez, de nem tárolja el az eredményt, hanem annak előjele alapján jelzőbiteket állít át az **eflags** regiszterben.
- Ha a kivonás eredménye 0 (azaz az összehasonlított értékek egyenlők), akkor a **zero flag** 1 lesz, különben 0.
- A feltételes ugró utasítások a jelzőbiteket figyelik.
- A **je** akkor ugrik, ha a **zero flag** 1, egyébként a következő utasításra kerül a vezérlés.

FELTÉTELES UGRÁSOK MŰKÖDÉSE



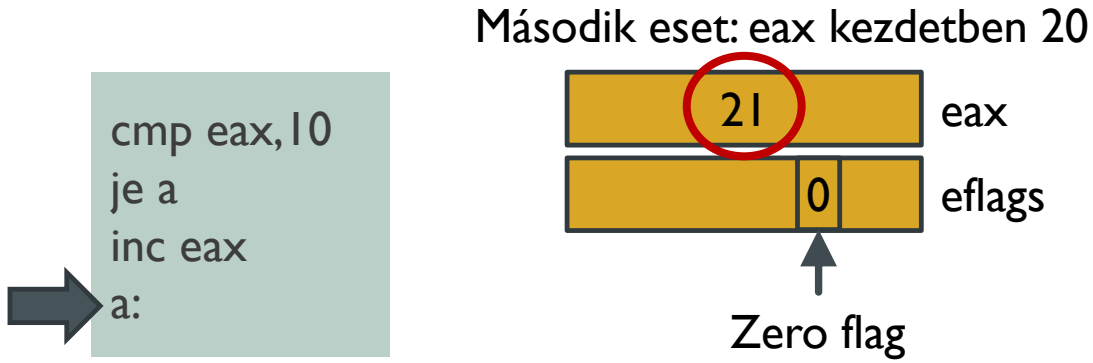
- A **cmp** utasítás kivonást végez, de nem tárolja el az eredményt, hanem annak előjele alapján jelzőbiteket állít át az **eflags** regiszterben.
- Ha a kivonás eredménye 0 (azaz az összehasonlított értékek egyenlők), akkor a **zero flag** 1 lesz, különben 0.
- A feltételes ugró utasítások a jelzőbiteket figyelik.
- A **je** akkor ugrik, ha a **zero flag** 1, egyébként a következő utasításra kerül a vezérlés.

FELTÉTELES UGRÁSOK MŰKÖDÉSE



- A **cmp** utasítás kivonást végez, de nem tárolja el az eredményt, hanem annak előjele alapján jelzőbiteket állít át az **eflags** regiszterben.
- Ha a kivonás eredménye 0 (azaz az összehasonlított értékek egyenlők), akkor a **zero flag** 1 lesz, különben 0.
- A feltételes ugró utasítások a jelzőbiteket figyelik.
- A **je** akkor ugrik, ha a **zero flag** 1, egyébként a következő utasításra kerül a vezérlés.

FELTÉTELES UGRÁSOK MŰKÖDÉSE



- A **cmp** utasítás kivonást végez, de nem tárolja el az eredményt, hanem annak előjele alapján jelzőbiteket állít át az **eflags** regiszterben.
- Ha a kivonás eredménye 0 (azaz az összehasonlított értékek egyenlők), akkor a **zero flag** 1 lesz, különben 0.
- A feltételes ugró utasítások a jelzőbiteket figyelik.
- A **je** akkor ugrik, ha a **zero flag** 1, egyébként a következő utasításra kerül a vezérlés.

ELÁGAZÁSOK ÉS CIKLUSOK

Egy ágú elágazás:

```
cmp eax,ebx
je egyenlo
mov eax,ebx
egyenlo:
```

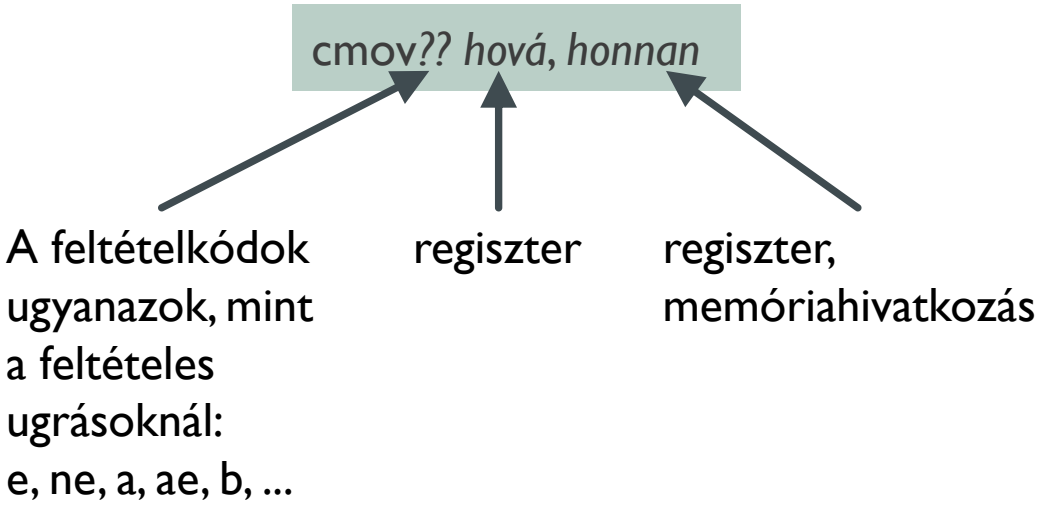
Két ágú elágazás:

```
cmp ebx,ecx
ja nagyobb
mov eax,ecx
jmp vege
nagyobb:
mov eax,ebx
vege:
```

Ciklus:

```
mov eax,0
eleje: cmp ecx,0
je vege
add eax,ecx
dec ecx
jmp eleje
vege:
```

FELTÉTELES ADATMOZGATÁS



Példa:

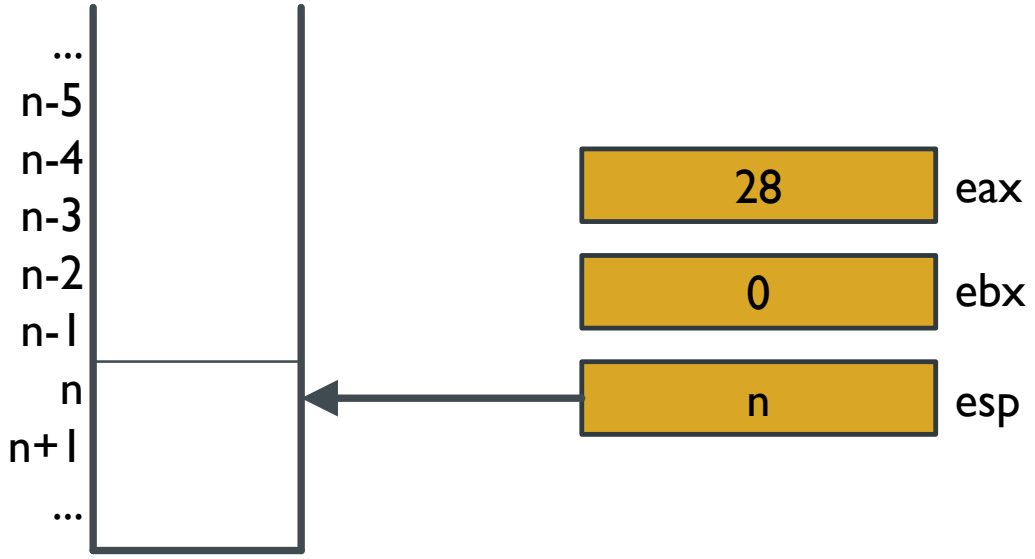
```
cmp eax, ebx
cmovne eax, ebx
```

FUTÁSI IDEJŰ VEREM

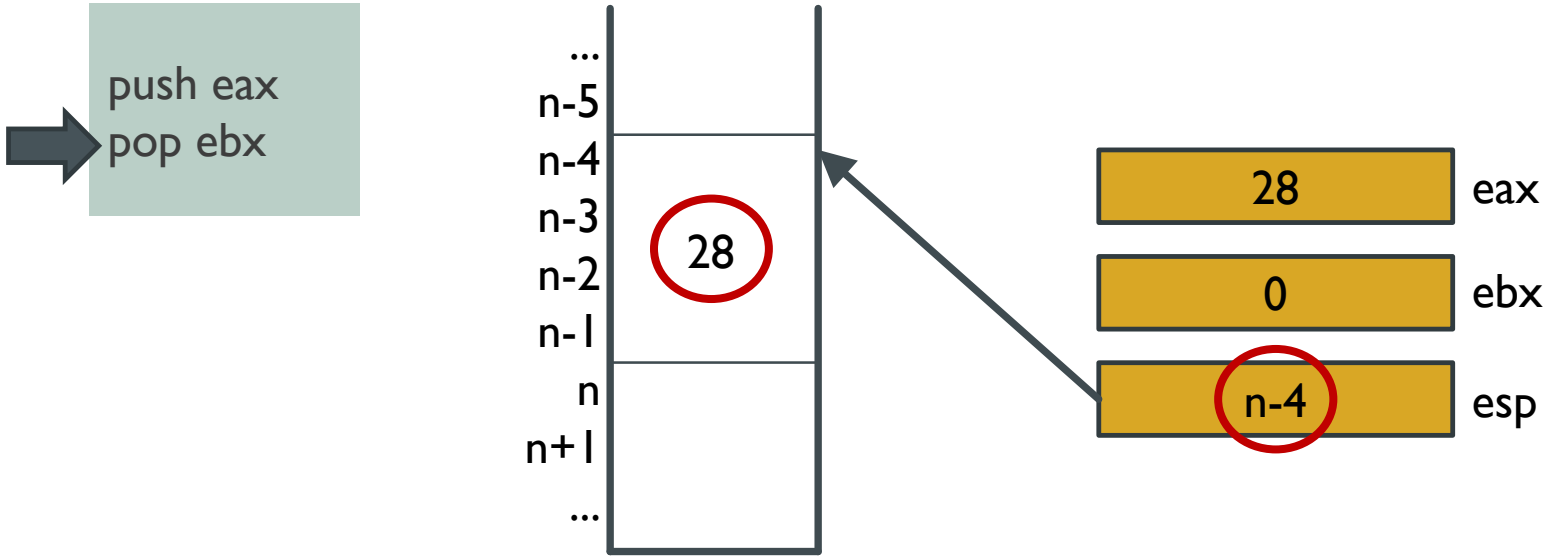
- Minden futó programhoz tartozik egy verem.
- A memóriaterületet az operációs rendszer rendeli hozzá a programhoz.
- Az **esp** regiszter mutatja a verem tetejét.
- Általában a veremben tároljuk:
  - függvények paraméterei
  - függvények visszatérési címe
  - függvények lokális változói
  - időlegesen tárolt adatok
- Veremműveletek: **push** és **pop**

VEREMMŰVELETEK

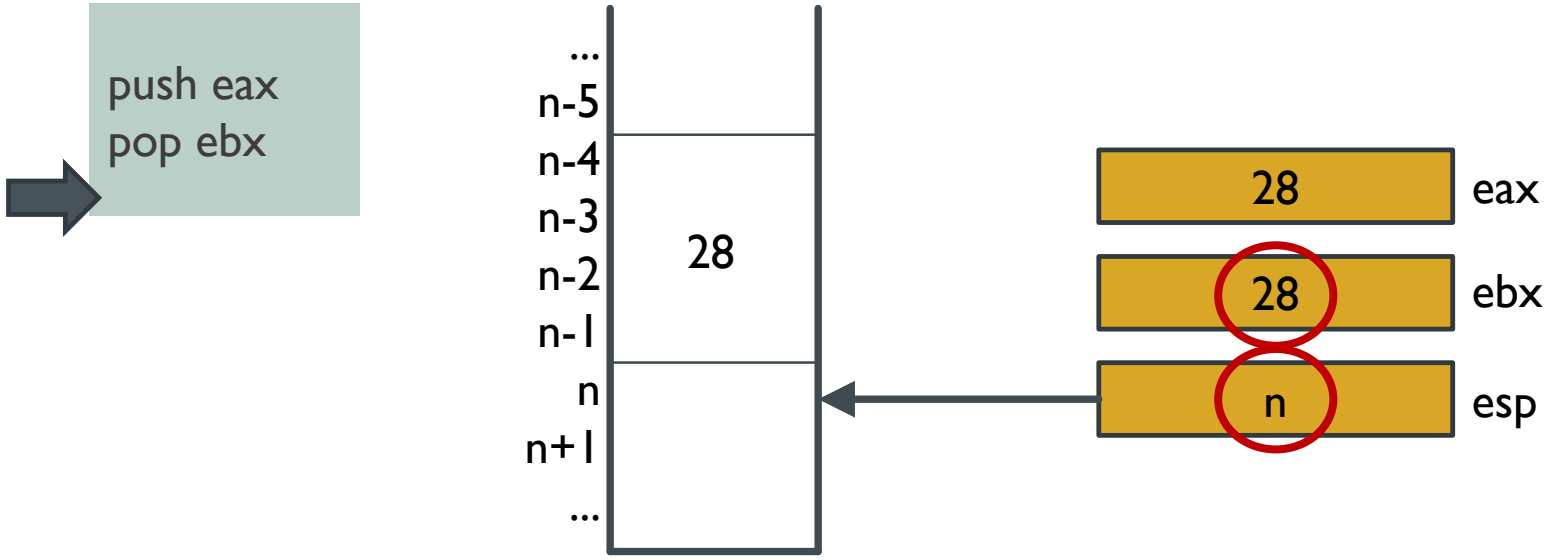
push eax
pop ebx



VEREMMŰVELETEK



VEREMMŰVELETEK

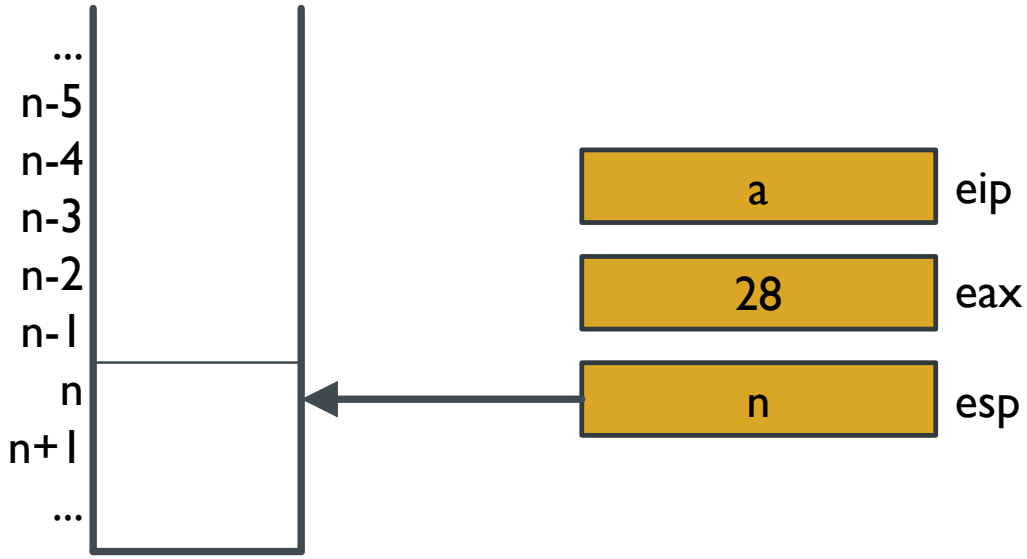


push operandusa: regiszter, memóriahivatkozás vagy literál
pop operandusa: regiszter vagy memóriahivatkozás
Mindkét esetben csak 2 vagy 4 bájtos lehet az operandus!

FÜGGVÉNYHÍVÁS ÉS VISSZATÉRÉS

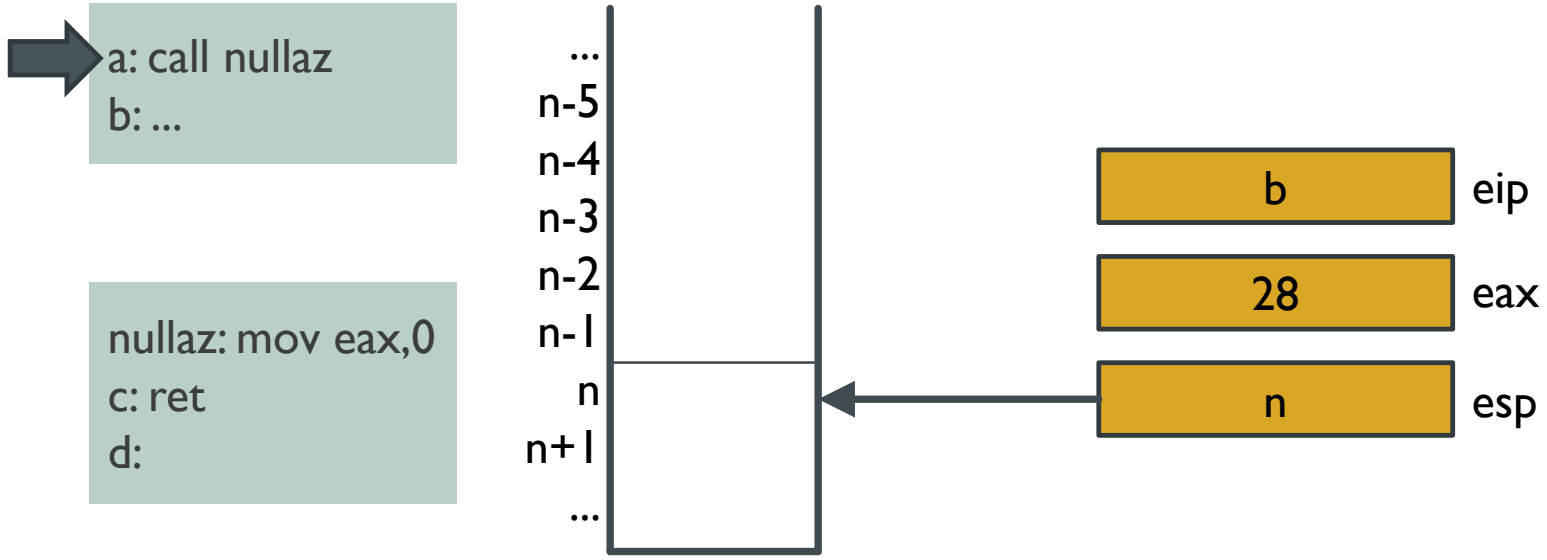
```
a: call nullaz
b: ...

nullaz: mov eax,0
c: ret
d:
```



- A **call** egy speciális ugró utasítás:
  - Az **eip** értékét, azaz a következő utasítás címét beteszi a verembe.
  - Átadja a vezérlést az operandusban adott helyre.

FÜGGVÉNYHÍVÁS ÉS VISSZATÉRÉS

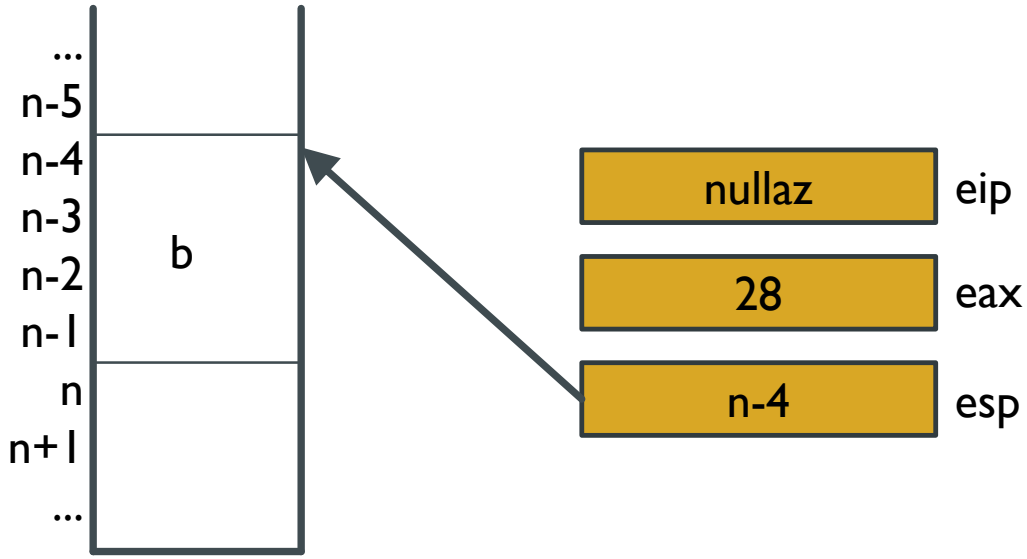


- A **call** egy speciális ugró utasítás:
  - Az **eip** értékét, azaz a következő utasítás címét beteszi a verembe.
  - Átadja a vezérlést az operandusban adott helyre.

FÜGGVÉNYHÍVÁS ÉS VISSZATÉRÉS

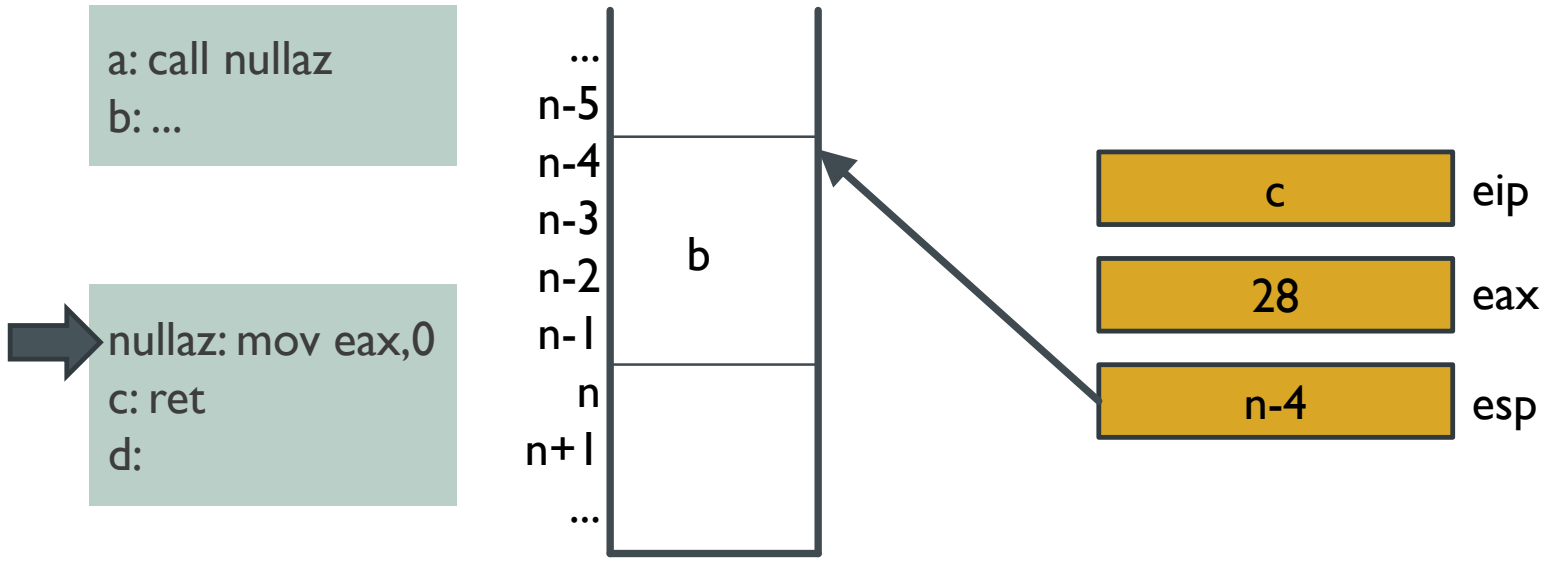
```
a: call nullaz
b: ...

nullaz: mov eax,0
c: ret
d:
```



- A **call** egy speciális ugró utasítás:
  - Az **eip** értékét, azaz a következő utasítás címét beteszi a verembe.
  - Átadja a vezérlést az operandusban adott helyre.

FÜGGVÉNYHÍVÁS ÉS VISSZATÉRÉS

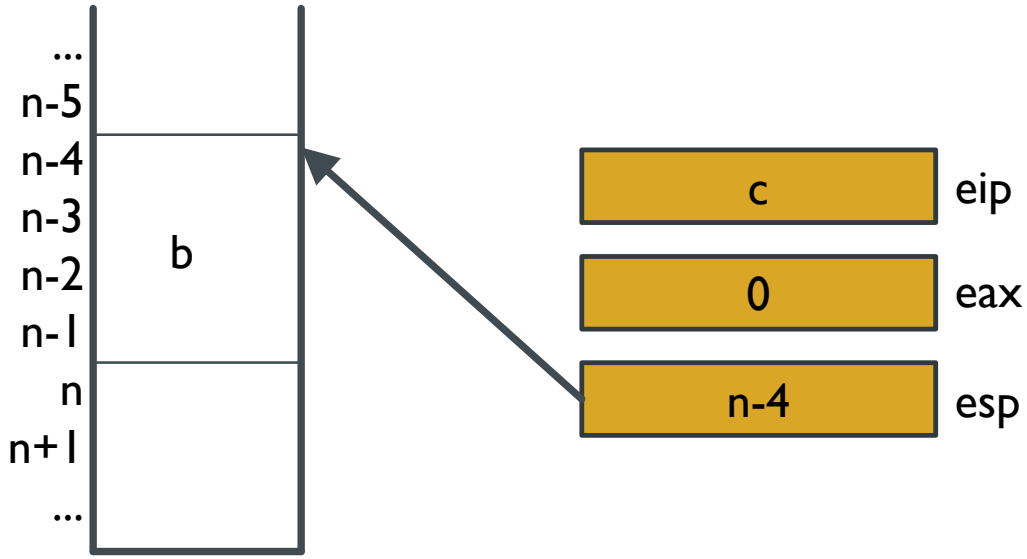


- A **call** egy speciális ugró utasítás:
  - Az **eip** értékét, azaz a következő utasítás címét beteszi a verembe.
  - Átadja a vezérlést az operandusban adott helyre.

FÜGGVÉNYHÍVÁS ÉS VISSZATÉRÉS

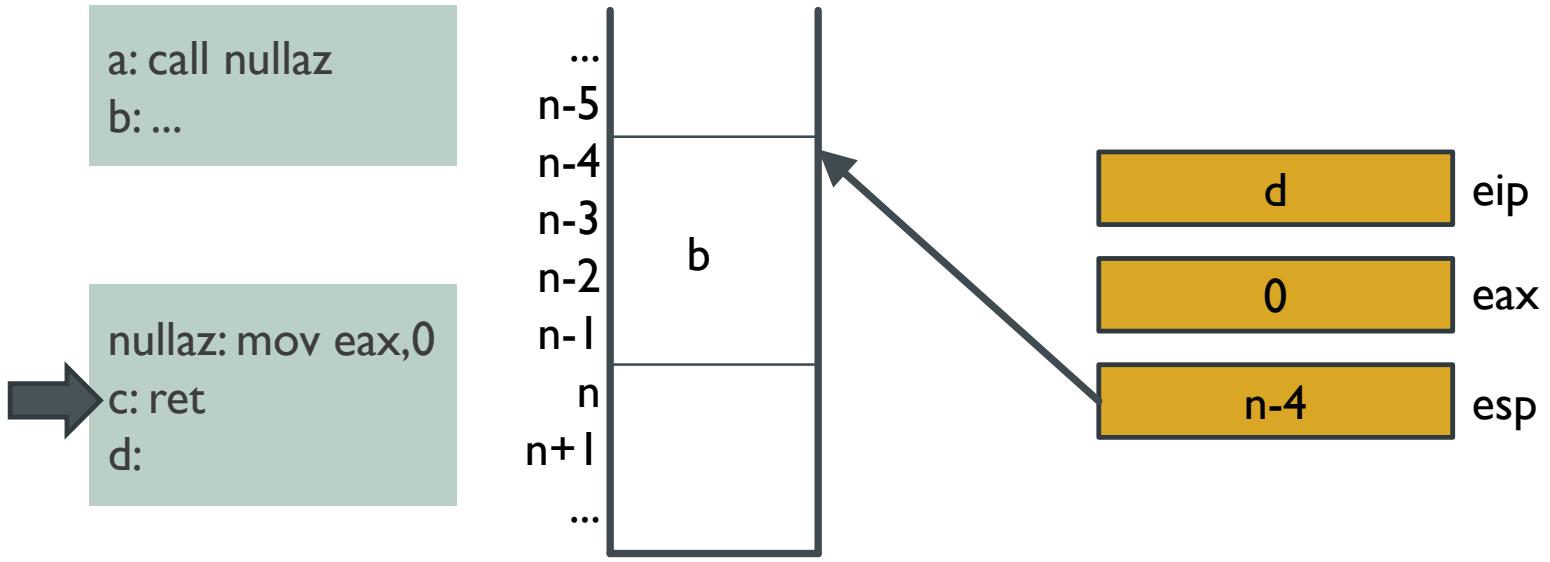
```
a: call nullaz
b: ...

nullaz: mov eax,0
c: ret
d:
```



- A **call** egy speciális ugró utasítás:
  - Az **eip** értékét, azaz a következő utasítás címét beteszi a verembe.
  - Átadja a vezérlést az operandusban adott helyre.

FÜGGVÉNYHÍVÁS ÉS VISSZATÉRÉS

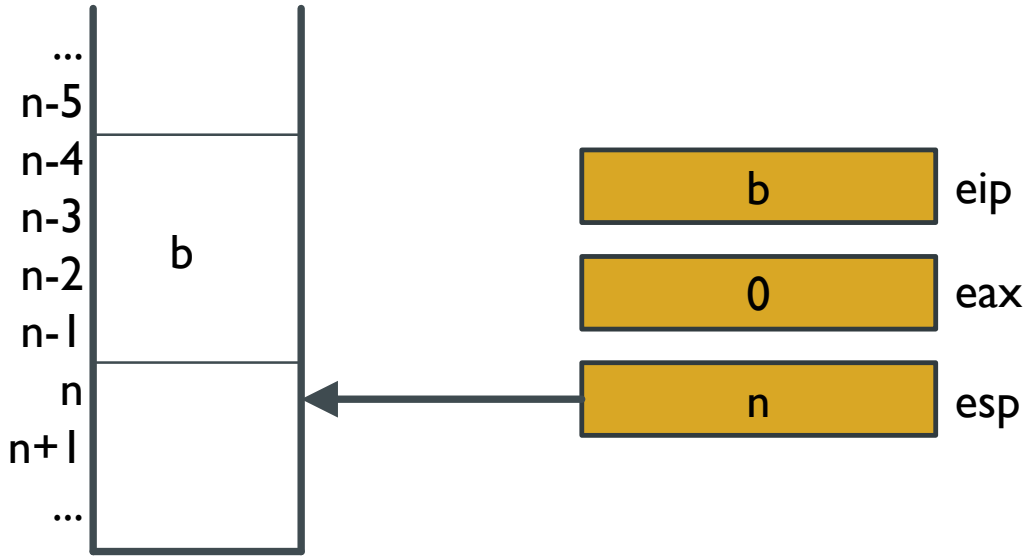


- A **call** egy speciális ugró utasítás:
  - Az **eip** értékét, azaz a következő utasítás címét beteszi a verembe.
  - Átadja a vezérlést az operandusban adott helyre.
- A **ret** kiveszi a verem tetején lévő 4 bájtot, és oda adja át a vezérlést.

FÜGGVÉNYHÍVÁS ÉS VISSZATÉRÉS

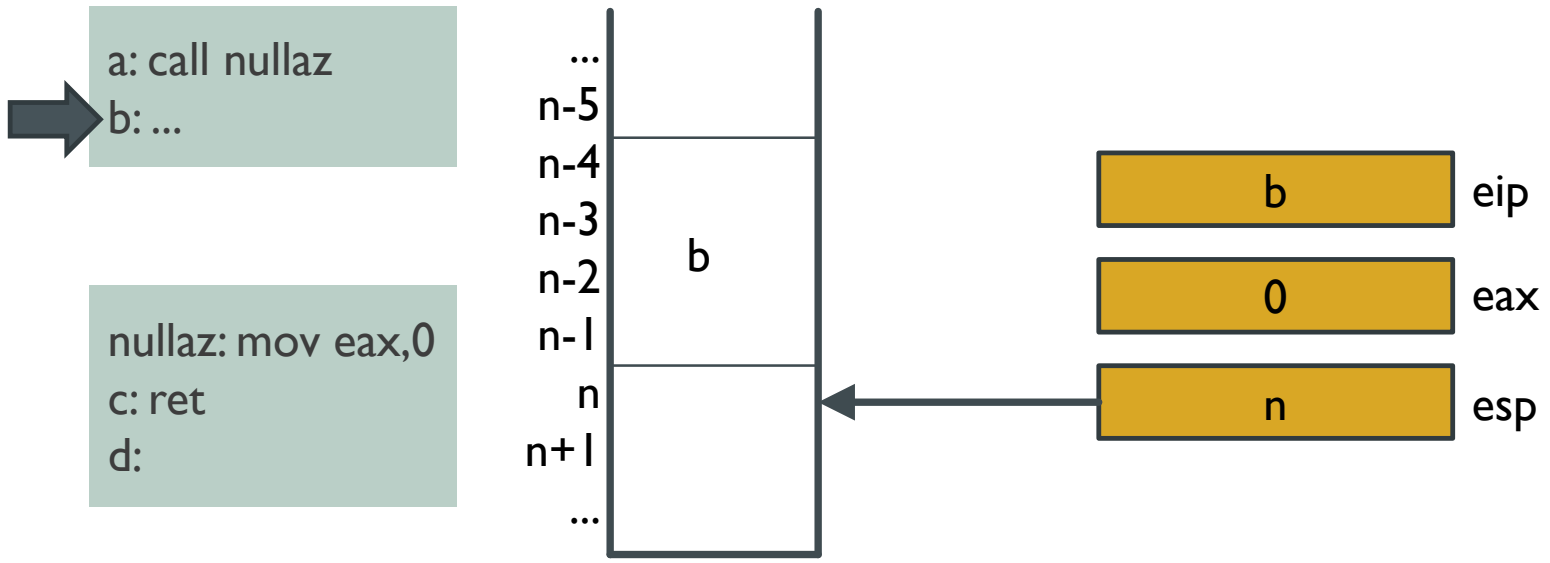
```
a: call nullaz
b: ...

nullaz: mov eax,0
c: ret
d:
```



- A **call** egy speciális ugró utasítás:
  - Az **eip** értékét, azaz a következő utasítás címét beteszi a verembe.
  - Átadja a vezérlést az operandusban adott helyre.
- A **ret** kiveszi a verem tetején lévő 4 bájtot, és oda adja át a vezérlést.

FÜGGVÉNYHÍVÁS ÉS VISSZATÉRÉS

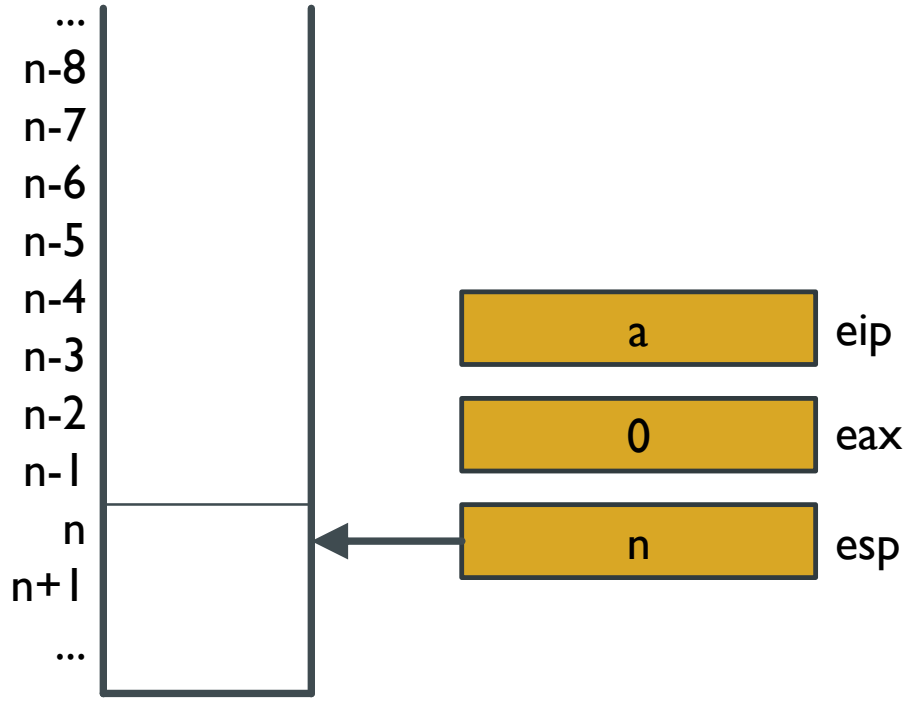


- A **call** egy speciális ugró utasítás:
  - Az **eip** értékét, azaz a következő utasítás címét beteszi a verembe.
  - Átadja a vezérlést az operandusban adott helyre.
- A **ret** kiveszi a verem tetején lévő 4 bájtot, és oda adja át a vezérlést.

FÜGGVÉNYHÍVÁS PARAMÉTERREL

```
a: push dword 5
b: call duplaz
c: add esp,4
d: ...
```

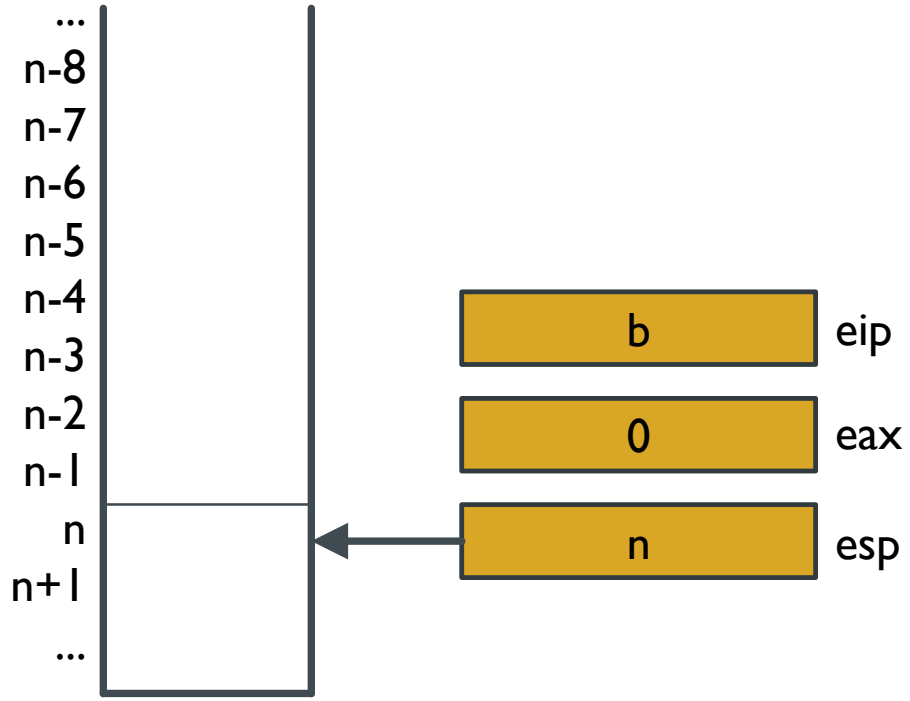
```
duplaz: mov eax, [esp+4]
e: add eax,eax
f: ret
g:
```



FÜGGVÉNYHÍVÁS PARAMÉTERREL

```
a: push dword 5
b: call duplaz
c: add esp,4
d: ...
```

```
duplaz: mov eax, [esp+4]
e: add eax,eax
f: ret
g:
```

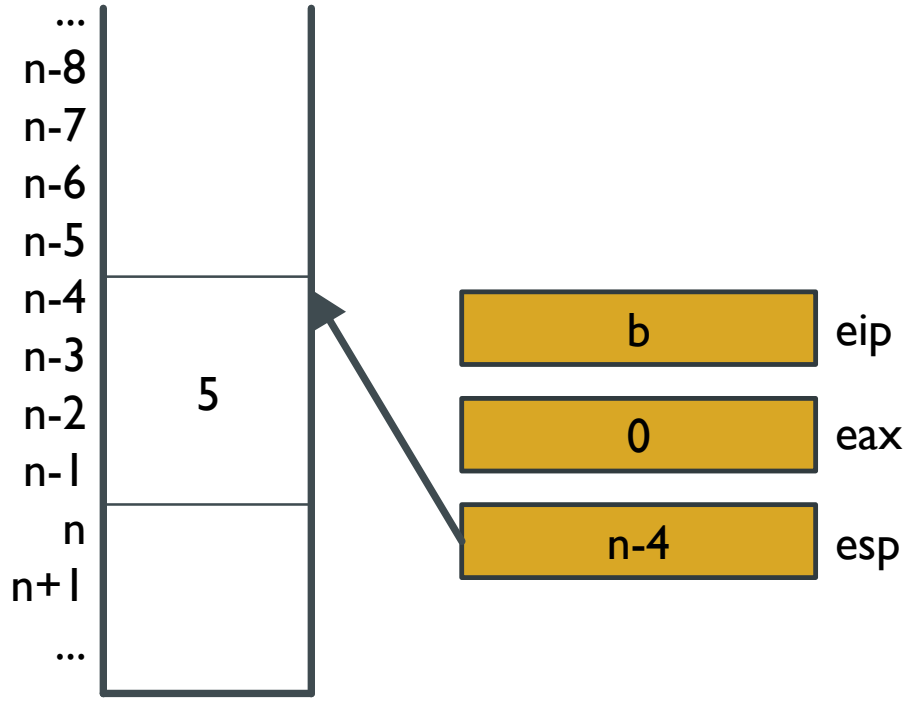


- A függvény paraméterét a verembe tesszük a hívás előtt.

FÜGGVÉNYHÍVÁS PARAMÉTERREL

```
a: push dword 5
b: call duplaz
c: add esp,4
d: ...
```

```
duplaz: mov eax, [esp+4]
e: add eax,eax
f: ret
g:
```

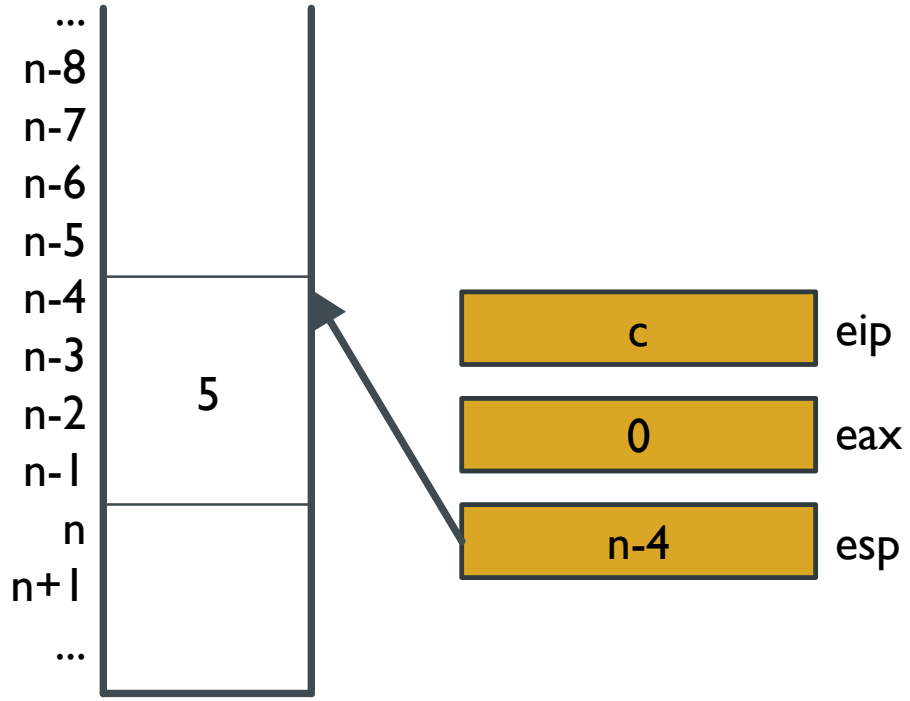


- A függvény paraméterét a verembe tesszük a hívás előtt.

FÜGGVÉNYHÍVÁS PARAMÉTERREL

```
a: push dword 5
b: call duplaz
c: add esp,4
d: ...
```

```
duplaz: mov eax, [esp+4]
e: add eax,eax
f: ret
g:
```

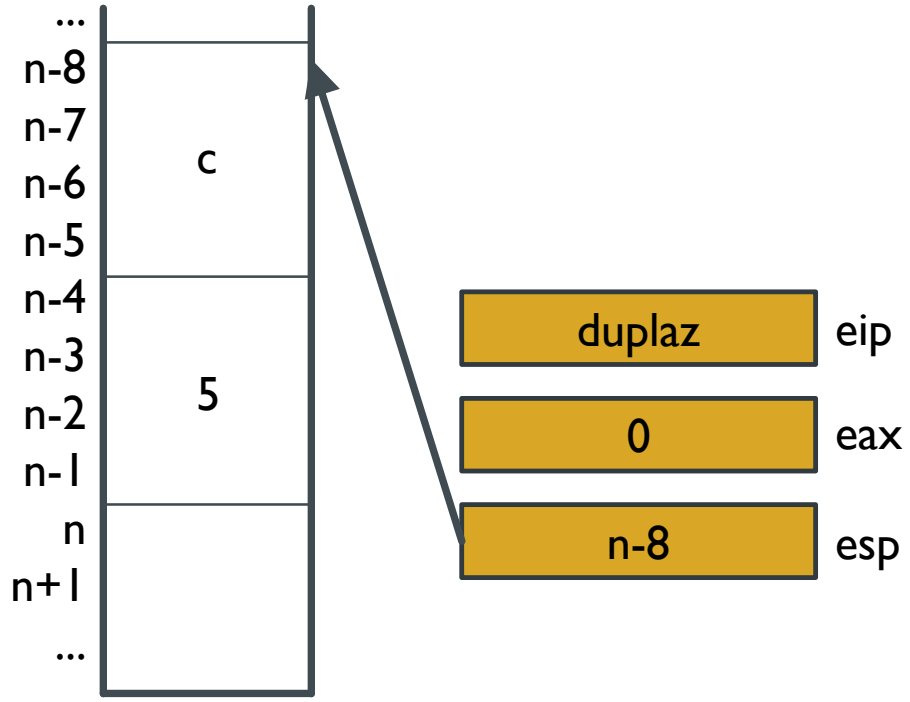


- A függvény paraméterét a verembe tesszük a hívás előtt.

FÜGGVÉNYHÍVÁS PARAMÉTERREL

```
a: push dword 5
b: call duplaz
c: add esp,4
d: ...
```

```
duplaz: mov eax, [esp+4]
e: add eax,eax
f: ret
g:
```

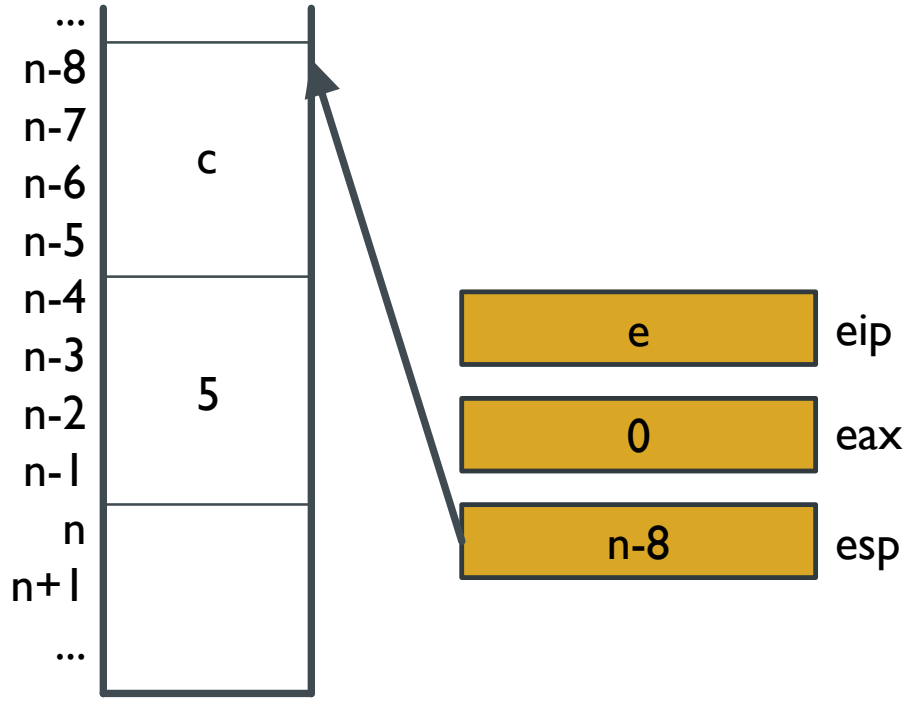


- A függvény paraméterét a verembe tesszük a hívás előtt.

FÜGGVÉNYHÍVÁS PARAMÉTERREL

```
a: push dword 5
b: call duplaz
c: add esp,4
d: ...
```

```
duplaz: mov eax, [esp+4]
e: add eax,eax
f: ret
g:
```

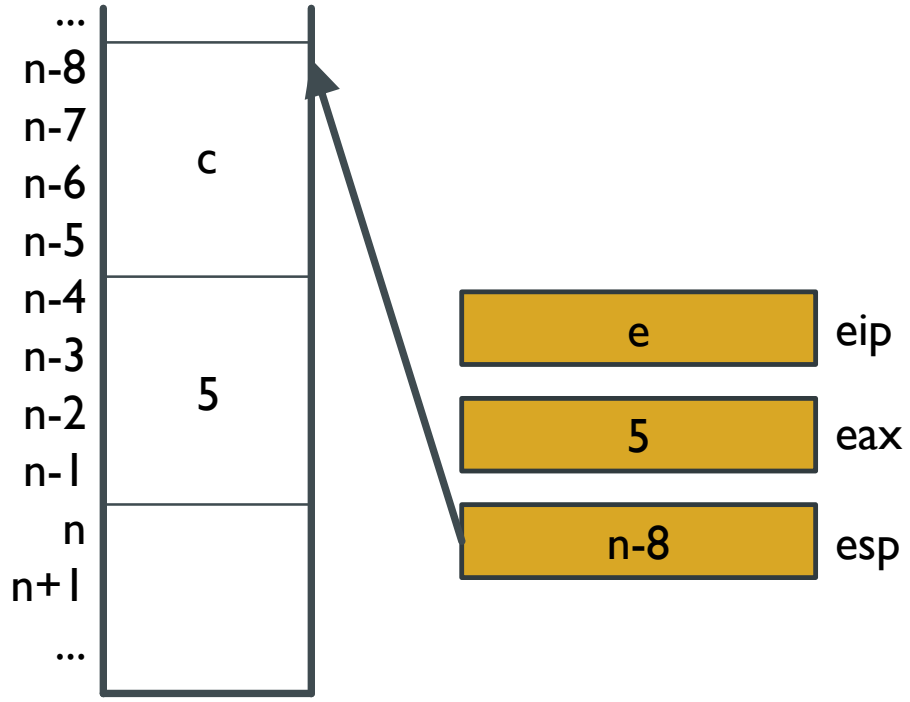


- A függvény paraméterét a verembe tesszük a hívás előtt.
- A függvény törzse kimásolja a paramétert a veremből.

FÜGGVÉNYHÍVÁS PARAMÉTERREL

```
a: push dword 5
b: call duplaz
c: add esp,4
d: ...
```

```
duplaz: mov eax, [esp+4]
e: add eax,eax
f: ret
g:
```

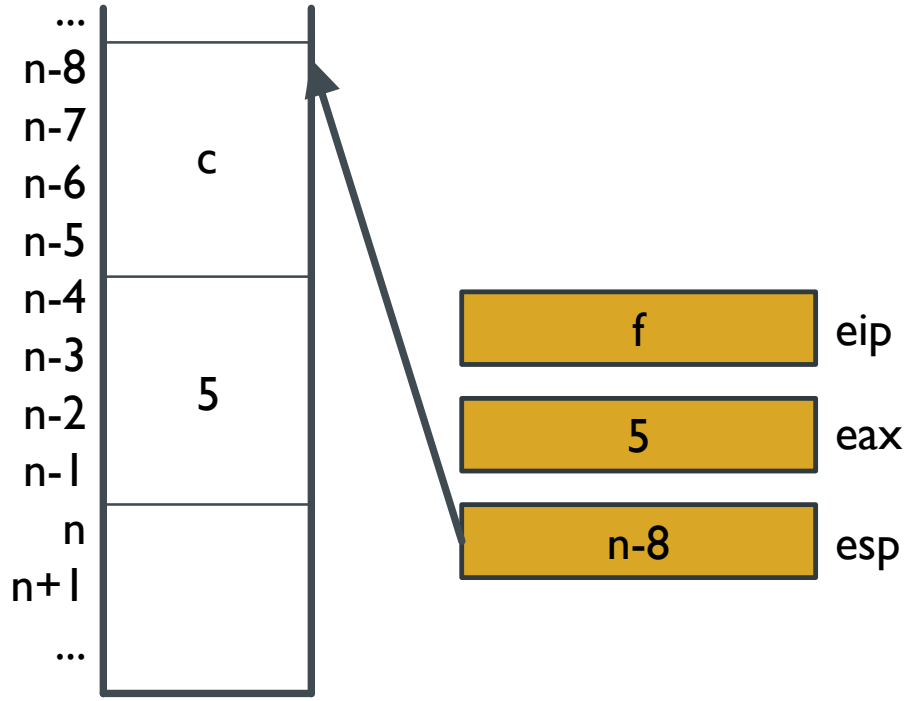


- A függvény paraméterét a verembe tesszük a hívás előtt.
- A függvény törzse kimásolja a paramétert a veremből.

FÜGGVÉNYHÍVÁS PARAMÉTERREL

```
a: push dword 5
b: call duplaz
c: add esp,4
d: ...
```

```
duplaz: mov eax, [esp+4]
e: add eax,eax
f: ret
g:
```

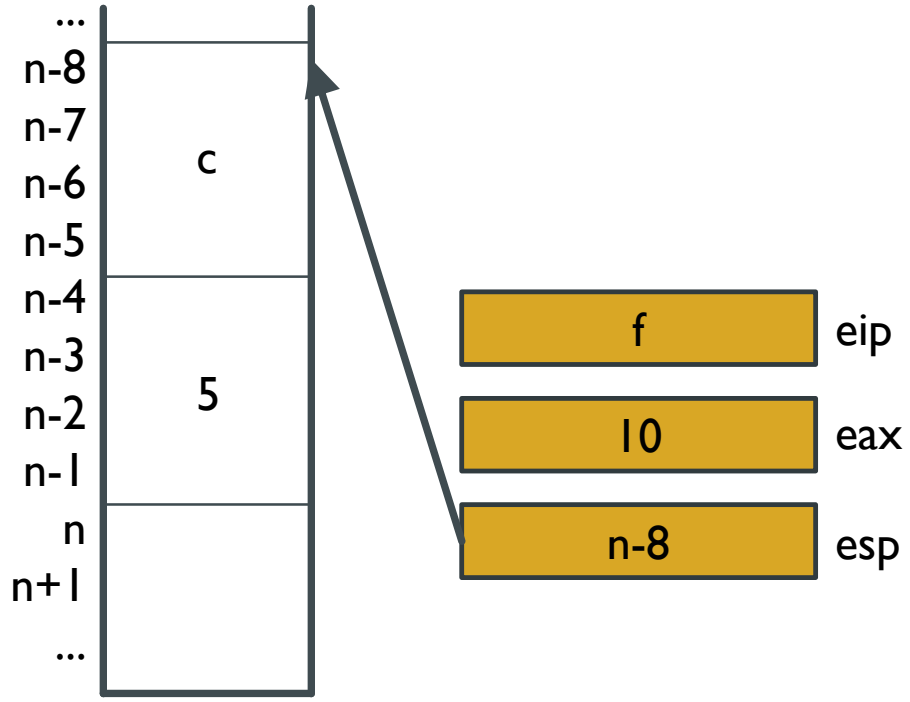


- A függvény paraméterét a verembe tesszük a hívás előtt.
- A függvény törzse kimásolja a paramétert a veremből.

FÜGGVÉNYHÍVÁS PARAMÉTERREL

```
a: push dword 5
b: call duplaz
c: add esp,4
d: ...
```

```
duplaz: mov eax, [esp+4]
e: add eax,eax
f: ret
g:
```

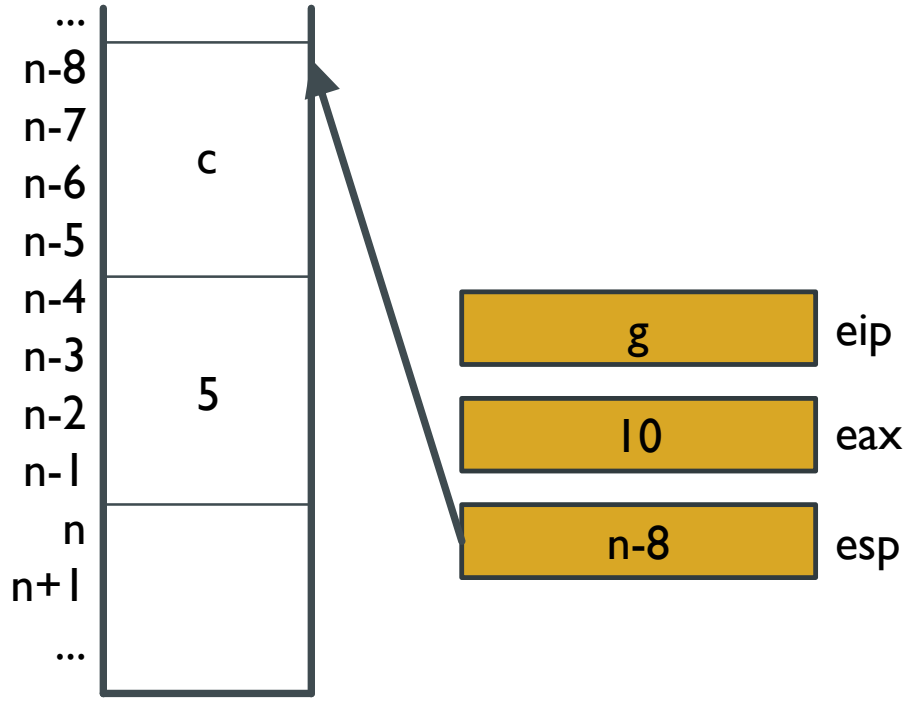


- A függvény paraméterét a verembe tesszük a hívás előtt.
- A függvény törzse kimásolja a paramétert a veremből.
- A visszatérési érték az eax regiszterben van.

FÜGGVÉNYHÍVÁS PARAMÉTERREL

```
a: push dword 5
b: call duplaz
c: add esp,4
d: ...
```

```
duplaz: mov eax, [esp+4]
e: add eax,eax
f: ret
g:
```

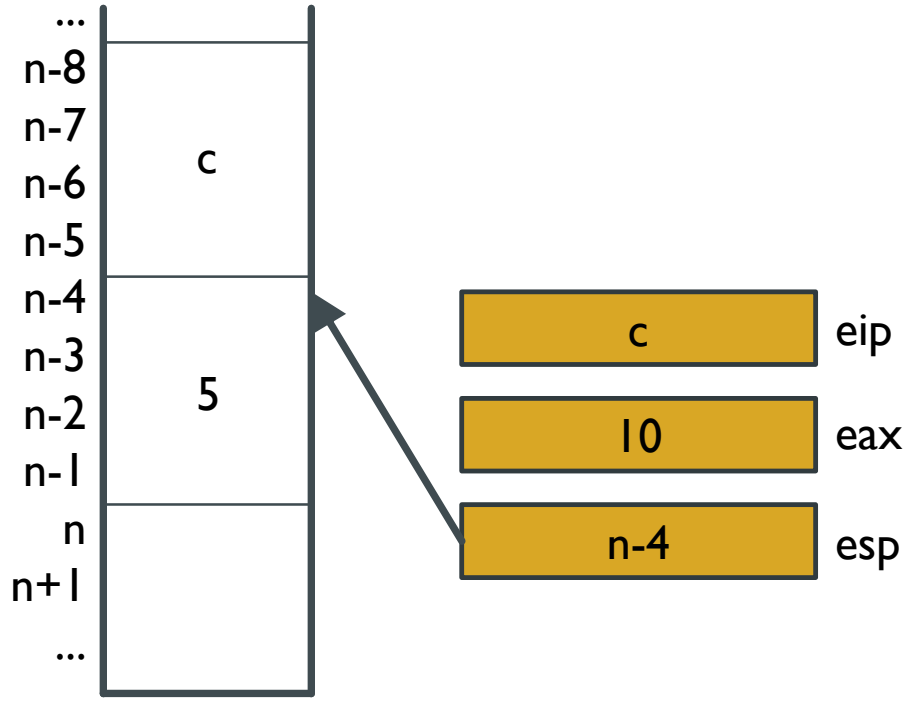


- A függvény paraméterét a verembe tesszük a hívás előtt.
- A függvény törzse kimásolja a paramétert a veremből.
- A visszatérési érték az eax regiszterben van.

FÜGGVÉNYHÍVÁS PARAMÉTERREL

```
a: push dword 5
b: call duplaz
c: add esp,4
d: ...
```

```
duplaz: mov eax, [esp+4]
e: add eax,eax
f: ret
g:
```

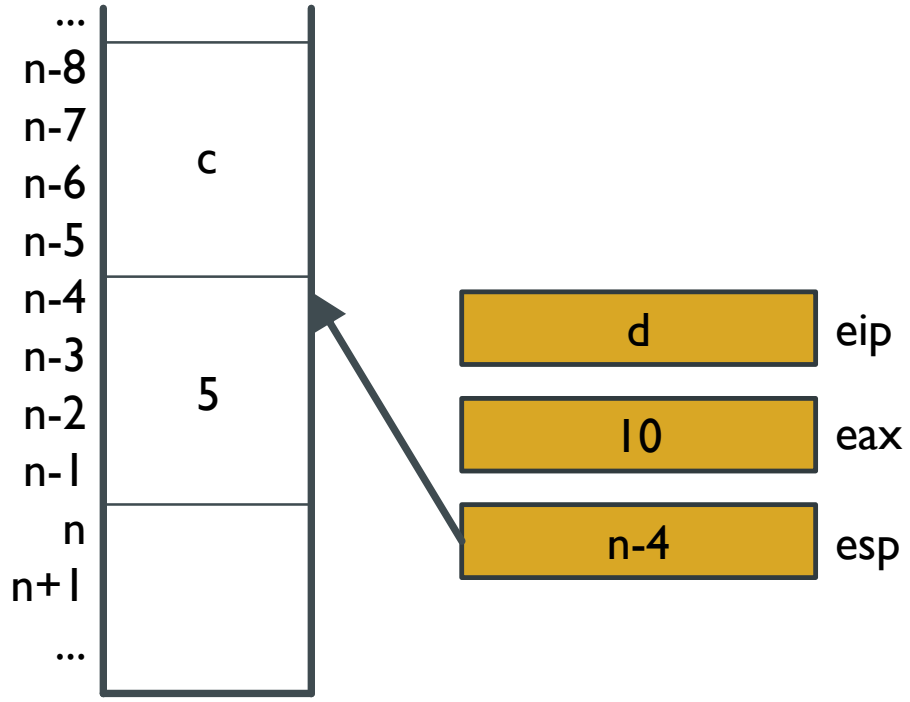


- A függvény paraméterét a verembe tesszük a hívás előtt.
- A függvény törzse kimásolja a paramétert a veremből.
- A visszatérési érték az eax regiszterben van.

FÜGGVÉNYHÍVÁS PARAMÉTERREL

```
a: push dword 5
b: call duplaz
c: add esp,4
d: ...
```

```
duplaz: mov eax, [esp+4]
e: add eax,eax
f: ret
g:
```

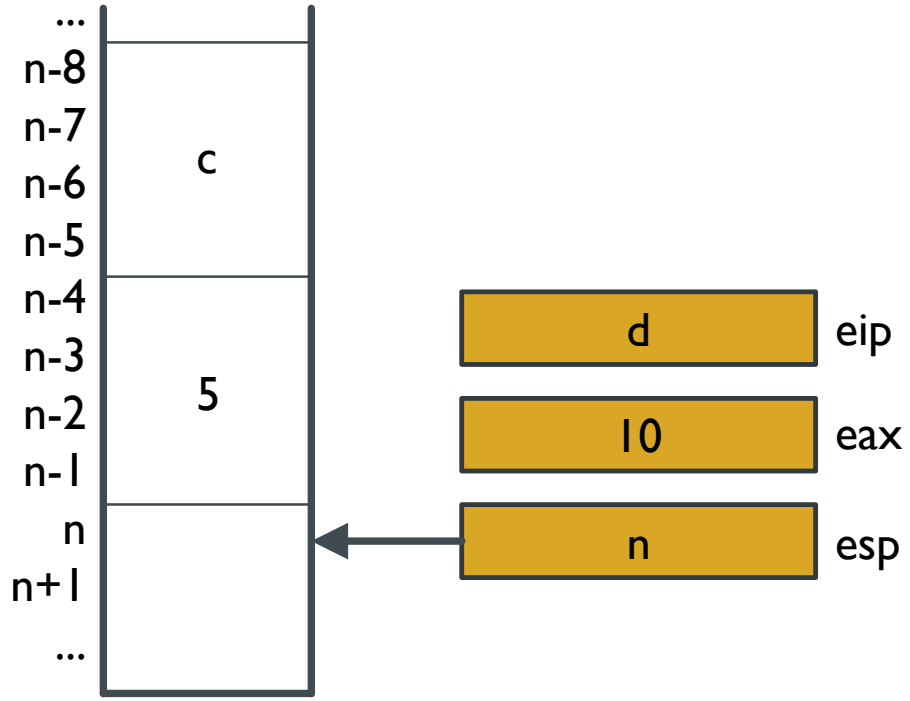


- A függvény paraméterét a verembe tesszük a hívás előtt.
- A függvény törzse kimásolja a paramétert a veremből.
- A visszatérési érték az eax regiszterben van.
- A hívás után kivesszük a paramétert (vagy legalább a veremmutatót visszaállítjuk).

FÜGGVÉNYHÍVÁS PARAMÉTERREL

```
a: push dword 5
b: call duplaz
c: add esp,4
d: ...
```

```
duplaz: mov eax, [esp+4]
e: add eax,eax
f: ret
g:
```



- A függvény paraméterét a verembe tesszük a hívás előtt.
- A függvény törzse kimásolja a paramétert a veremből.
- A visszatérési érték az eax regiszterben van.
- A hívás után kivesszük a paramétert (vagy legalább a veremmutatót visszaállítjuk).

C FÜGGVÉNYEK HÍVÁSI KONVENCIÓI

- A címke a függvény neve.
- A paramétereket fordított sorrendben tesszük a verembe.
- A függvény csak az **eax**, **ecx** és **edx** regisztereket hagyja változatlanul, a többit „elronthatja”.
- A függvény bent hagyja a veremben a paramétereket.
- A visszatérési érték az **eax** regiszterbe kerül.

addone.asm

```
global main
extern write_natural
extern read_natural

section .text
main:
call read_natural
inc eax
push eax
call write_natural
add esp,4
mov eax,0
ret
```