Arhitectura 80x86

Scurt istoric al familiei x86:

- Primul microprocesor **4004** era pe 4 biţi (apărut în 1970);
- doi ani mai târziu i-a urmat **8008** și apoi **8080** care erau procesoare pe 8 biţi;
- **8086** a apărut în 1978 şi apoi i-a urmat **80286** (procesoare pe 16 biţi).

Un caz special este procesorul **8088**, cu aceeaşi structură ca 8086, dar care comunică în exterior printr-o magistrală de 8 biţi şi nu de 16 biţi ca 8086, proiectat astfel din economie.

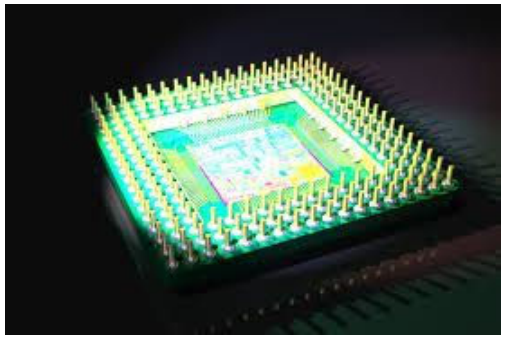
Îmbunătăţirile aduse de la un procesor la altul de-a lungul timpului s-au bazat pe tendinţa de a obţine ***o putere de calcul mai mare*** prin

creşterea numărului de biţi prelucraţi la momentul curent.

- în următoarea etapă s-a trecut la prelucrări pe 32 biţi (începând cu **80386** și continuând apoi cu ***80486***, apoi seria ***Pentium I, II,*** *etc*)
- mai recent: cele pe 64 biţi (de la **Core 2** spre **Core i3, i5, i7 – gen 1, 2, ... , 8**).

Microprocesorul

(Unitatea centrala de procesare – UCP sau CPU)



= Elementul de bază al unui SC

= un *cip* deosebit de complex plasat de obicei ***pe placa de bază (PB)***

din ***unitatea centrala (UC)*** a ***sistemului de calcul (SC)***

- asigură:
  - *procesarea datelor*: interpretarea, prelucrarea şi controlul acestora,
  - *supervizează transferurile* de informaţii şi
    - *controlează activitatea* generală a *celorlalte componente* care alcătuiesc SC

DAR la nivel software ?

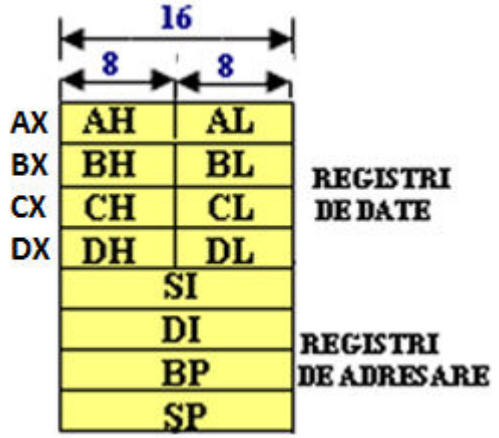
Arhitectura software
8086 – procesor pe 16 biti

Ce inseamna “procesor pe 16 biti” ?

REGISTRII interni sunt de **16 biti**

Registru = o structura interna de baza a CPU,
necesara in efectuarea operatiilor

UCP 8086 are registrii interni: AX, BX, CX, DX



SI, DI, BP, SP – de 16 biti : mov AX, 2 -> AX=0002h

Registrii AX, BX, CX, DX pot fi fol. si ca registrii de 8 biti : mov AL, 2 -> AL= 02h

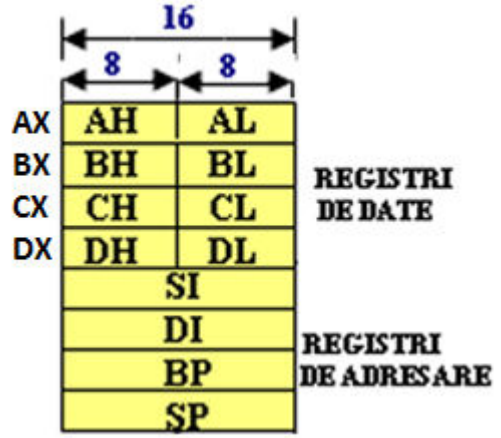
Ca parte HIGH (AH,BH,CH,DH) sau parte LOW (AL,BL,CL,DL) **sau** mov AH, 2 -> AH=02h

se citeste: “muta in registrul ___ valoarea 2”

2 blocuri mari :

- **UNITATE de EXECUTIE (EU – Execution Unit)**
- **UNITATE de INTERFATA cu busurile (BIU – Bus Interface Unit)**

Arhitectura software
8086 – procesor pe 16 biti



- La un procesor pe 16 biti:

REGISTRII interni sunt de **16 biti**

Registru = o structura interna de baza a CPU,
necesara in efectuarea operatiilor

UCP 8086 are registrii interni: AX, BX, CX, DX

SI, DI, BP, SP – de 16 biti : mov AX, 2 -> AX=0002h

Registrii AX, BX, CX, DX pot fi fol. si ca registrii de 8 biti : mov AL, 2 -> AL= 02h

Ca parte HIGH (AH,BH,CH,DH) sau parte LOW (AL,BL,CL,DL) **sau** mov AH, 2 -> AH=02h

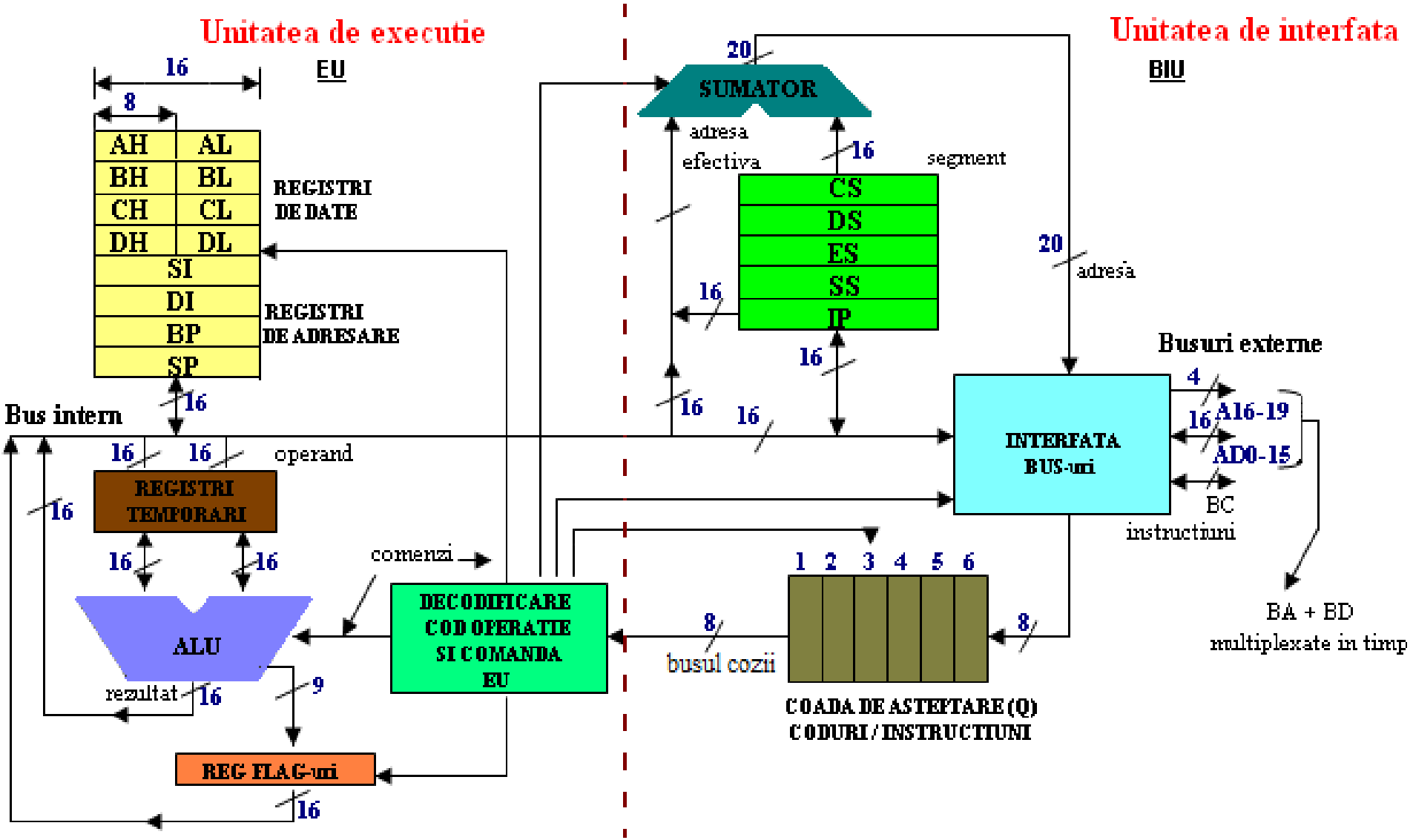
se citeste: "muta in registrul ___ valoarea 2"

2 blocuri mari :

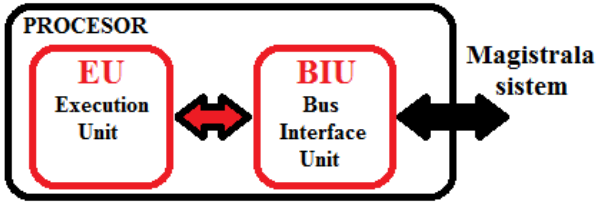
- **UNITATE de EXECUTIE (EU – Execution Unit)**
- **UNITATE de INTERFATA cu busurile (BIU – Bus Interface Unit)**

Microprocesorul I8086 are două componente principale:

unitatea de execuţie (EU) şi ***unitatea de interfaţă cu bus-urile (BIU)***.



EU si BIU

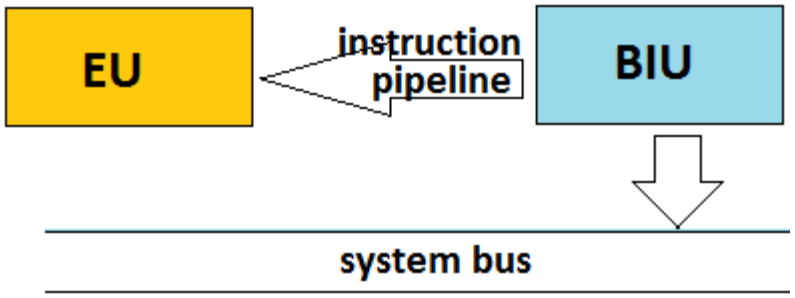


- **EU** are doar rolul de a executa instrucţiunile extrase de BIU, neavând nici o legătură cu magistrala sistemului.
- În timp ce EU îşi îndeplineşte sarcina, BIU extrage noi instrucţiuni pe care le organizează într-o coadă de aşteptare (*queue*).
- **BIU** pregăteşte execuţia fiecărei instrucţiuni, astfel: extrage o instrucţiune din memorie, o depune în coada de instrucţiuni şi calculează adresa din memorie a unui eventual operand. La terminarea execuţiei unei instrucţiuni, EU are deja la dispoziţie o nouă instrucţiune in coada de aşteptare construita de BIU.
- Cele două unităţi, EU şi BIU, lucrează deci în paralel, existând momente de sincronizare şi aşteptare între ele. Funcţionarea paralelă a celor două unităţi (BIU şi EU) este transparentă utilizatorului. Această arhitectură se mai numeşte şi arhitectură cu prelucrare secvenţial – paralelă de tip ***pipeline cu 2 etaje***.
- **Unitatea de execuţie EU** conţine **o unitate logico-aritmetică (ALU)** și **Unitatea de Comandă sau Control** a EU de 16 biţi, **registrul indicatorilor de stare (FLAGS)**, **registrul operatorilor (TEMPORARI)** şi **REGIŞTRII GENERALI (de date si adresare).**
- **Unitatea de interfaţă BIU** conţine indicatorul de instrucţiuni IP (Instruction Pointer), REGISTRII SEGMENT, un bloc de CONTROL SI INTERFATARE AL MAGISTRALEI, un bloc SUMATOR de generare a adresei pe 20 biţi şi o memorie organizată sub forma unei COZI, în care sunt depuse instrucţiunile extrase de BIU (Instruction Queue, de 6 locaţii - octeţi). Singurele momente în care coada trebuie reiniţializată, şi deci EU trebuie să aştepte efectiv citirea unei instrucţiuni, sunt cele care urmează după execuţia unei instrucţiuni de salt (acolo, anticiparea instructiunii ce urmeaza poate fi eronata).

Arhitectura software
8086 – procesor pe 16 biti

2 blocuri mari :

- **UNITATE de EXECUTIE (EU – Execution Unit)**
- **UNITATE de INTERFATA cu busurile (BIU – Bus Interface Unit)**

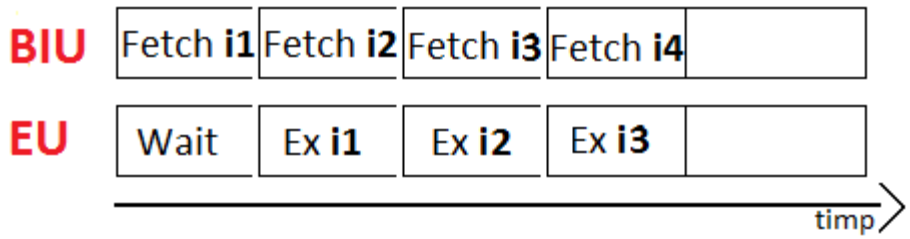


- **In timp ce una lucreaza, si cealalta lucreaza**

=> conceptul "*pipeline de instructiuni*"

Astfel, 8086 implementeaza un

pipeline cu 2 etaje



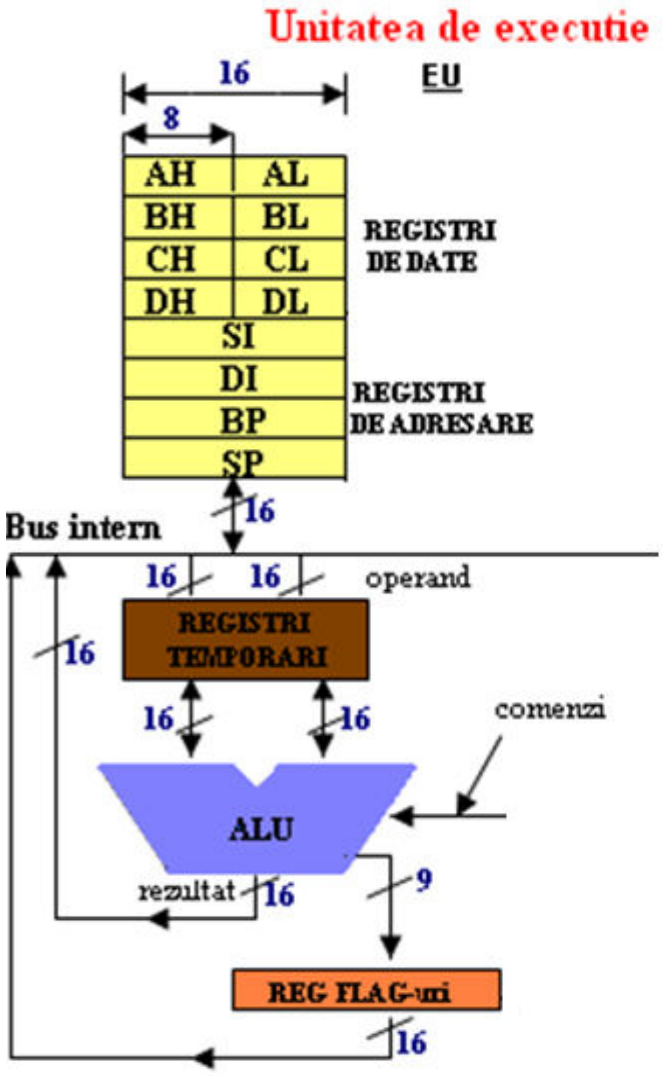
- In timp ce **BIU** aduce din memorie ("***Fetch***") ***instructiunea 1***, **EU** asteapta (Wait)

(= ***perioada de latenta, de umplere a liniei pipleline***), dar apoi :

- In timp ce **BIU** realizeaza **Fetch** pentru ***instructiunea 2***, **EU Executa** ***instructiunea 1***
- In timp ce **BIU** realizeaza **Fetch** pentru ***instructiunea 3***, **EU Executa** ***instructiunea 2***, …

Arhitectura software a microprocesorului 8086

Anca APATEAN - UTCN



EU (Unitatea de Executie)
– contine registrii de date,
de adresare, …

Care este rolul **EU** ?
- sa *execute* operatii
Ce fel de operatii?

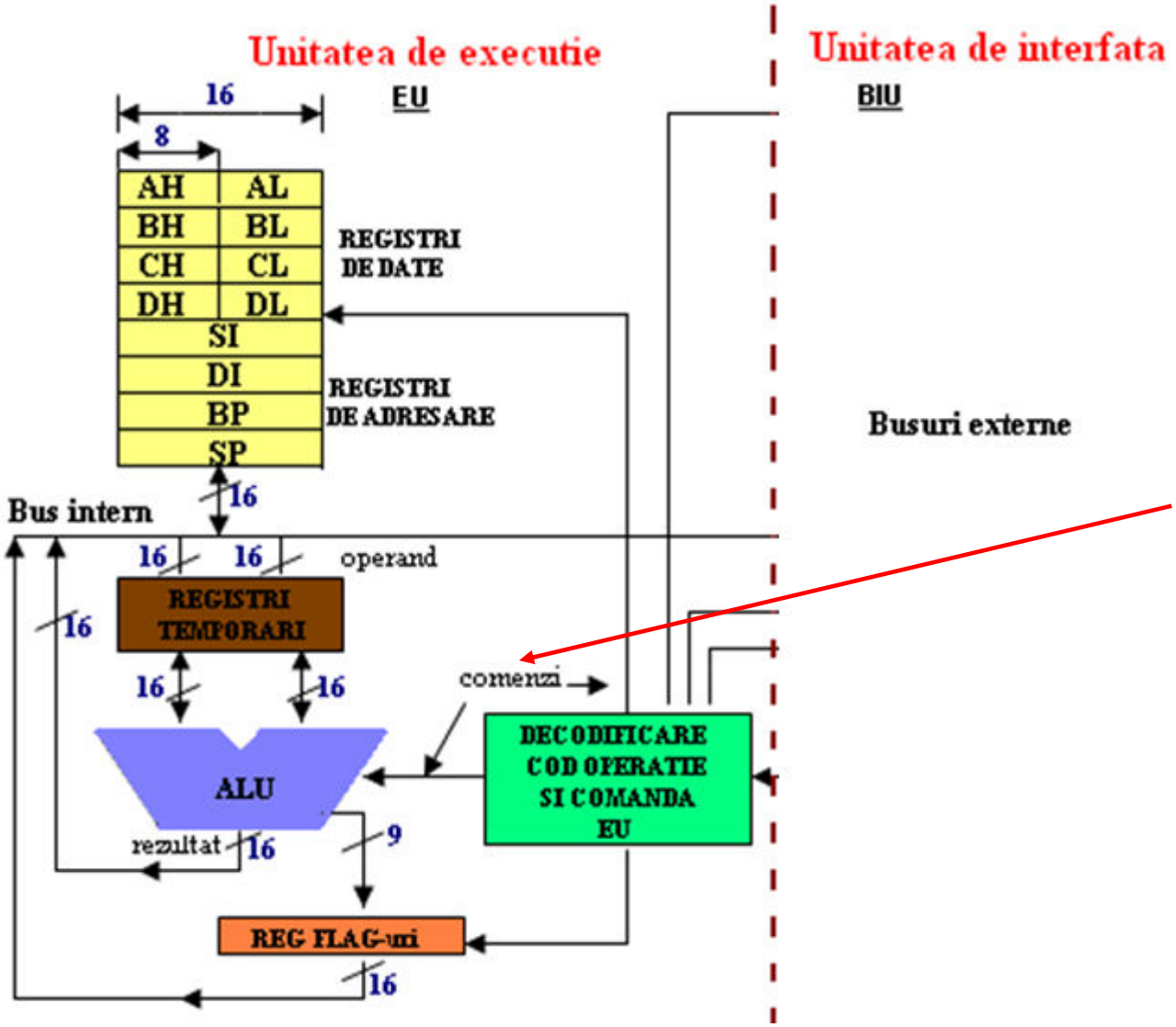
⇒ Are un
ALU Arithmetic & Logic Unit

In gen:
- **Rezultatul - tot pe 16 biti**

+ posibilitatea indicarii
unor ***situatii exceptionale***
(REGISTRUL de ***FLAG-uri***)
- 9 “indicatori” posibili

Arhitectura software a microprocesorului 8086

Anca APATEAN - UTCN



EU (Unitatea de Executie) – contine **registrii de date, de adresare, temporari** – pt operanzi, **ALU**, **FLAGS**

De unde vin comenzile/ instructiunile pe care le are de executat?

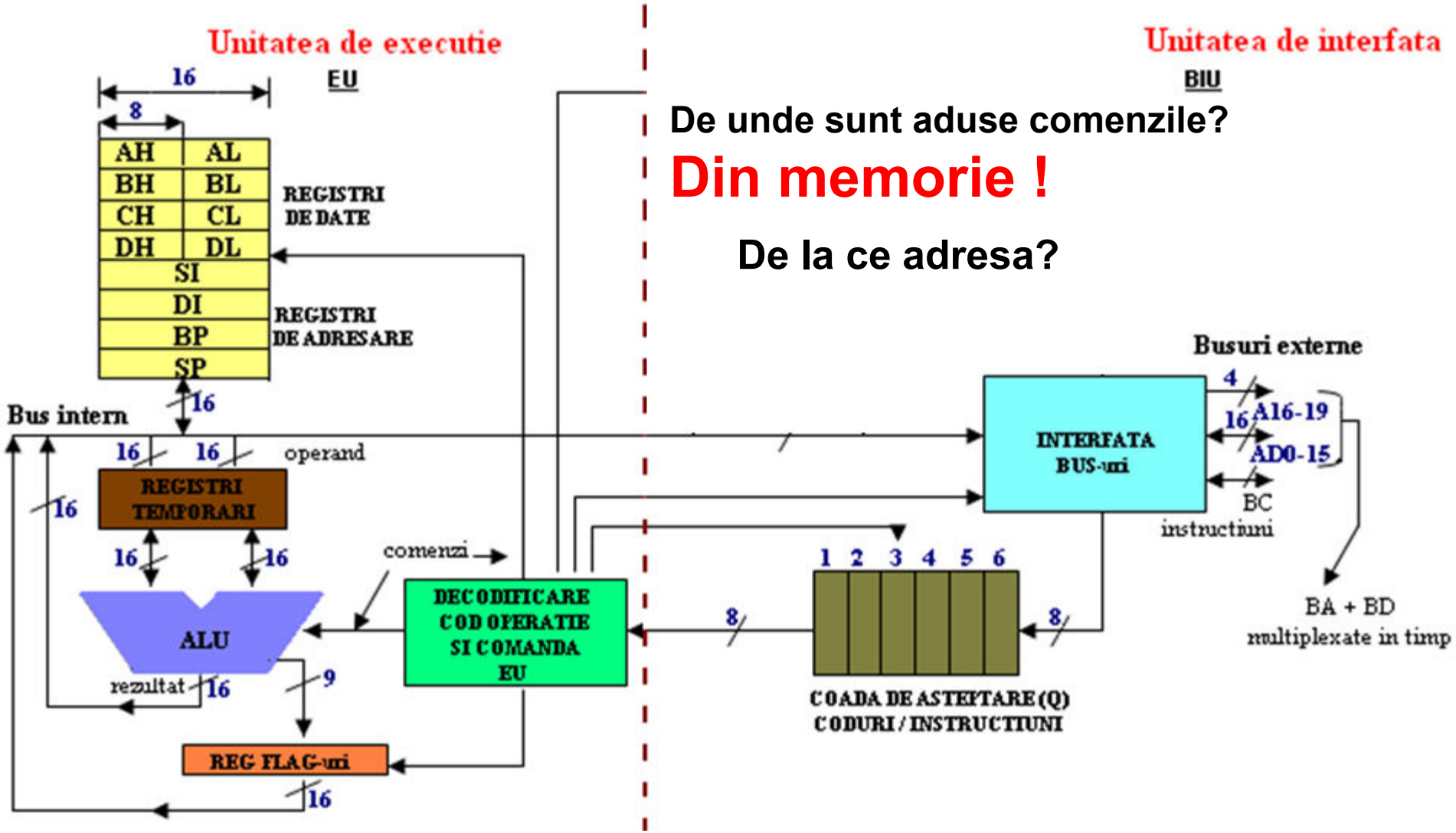
Bloc de *decodificare cod operatie si comanda EU*

(se mai spune si UNITATE DE COMANDA/CONTROL)

Si inainte de a ajunge in acest bloc, de unde vin?

Arhitectura software a microprocesorului 8086

Anca APATEAN - UTCN



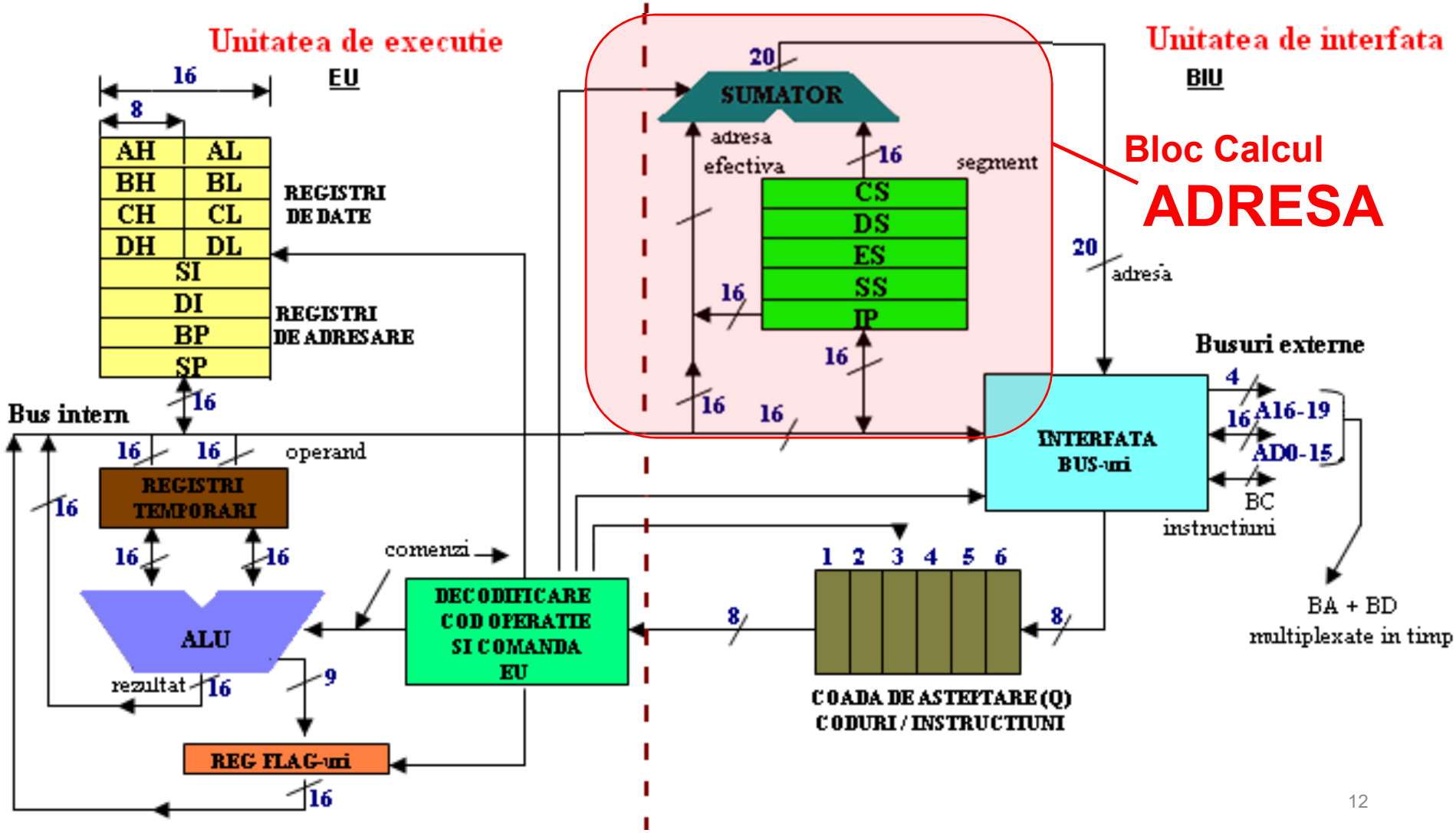
De unde sunt aduse comenzile?

Din memorie !

De la ce adresa?

Arhitectura software a microprocesorului 8086

Anca APATEAN - UTCN



Bloc de calcul ADRESA

Insumeaza 16 b cu 16 b si rezulta o adresa pe 20 b

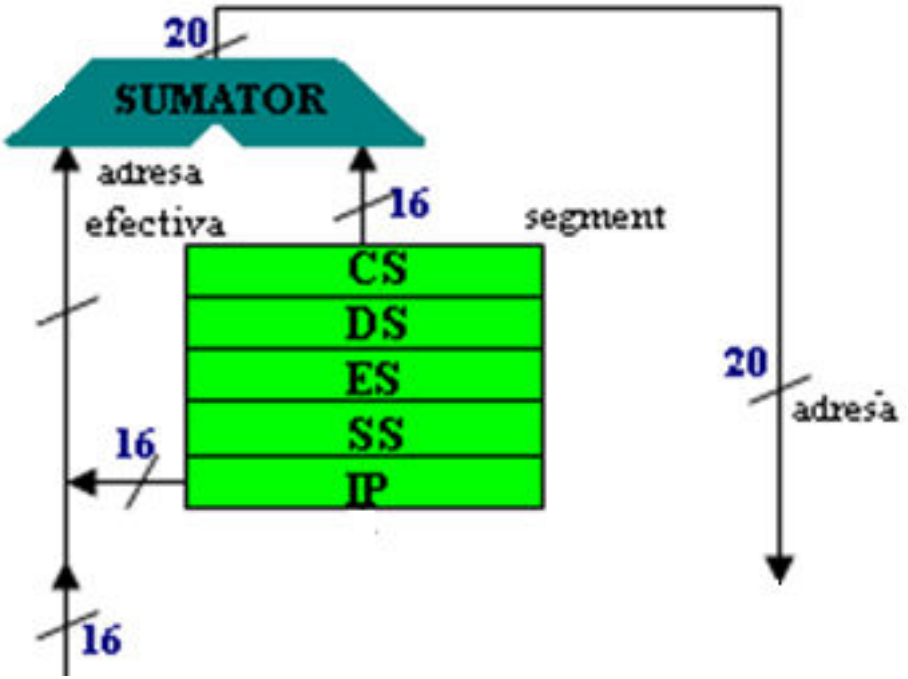
Registrii segment:

de COD: CS (Code Segment),

de DATE: DS (Data Segment),

de STIVA: SS (Stack Segment),

de oricare: ES (Extra Segment)



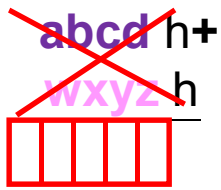
Instruction Pointer IP:

Pointer la instructiunea "curenta"

(ce urmeaza sa se execute)

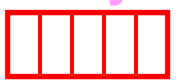
Se aduna 16 b cu 16 b si rezulta o adresa pe 20 b ?!?!?!?!?!

Adica abcd h+
wxyz h



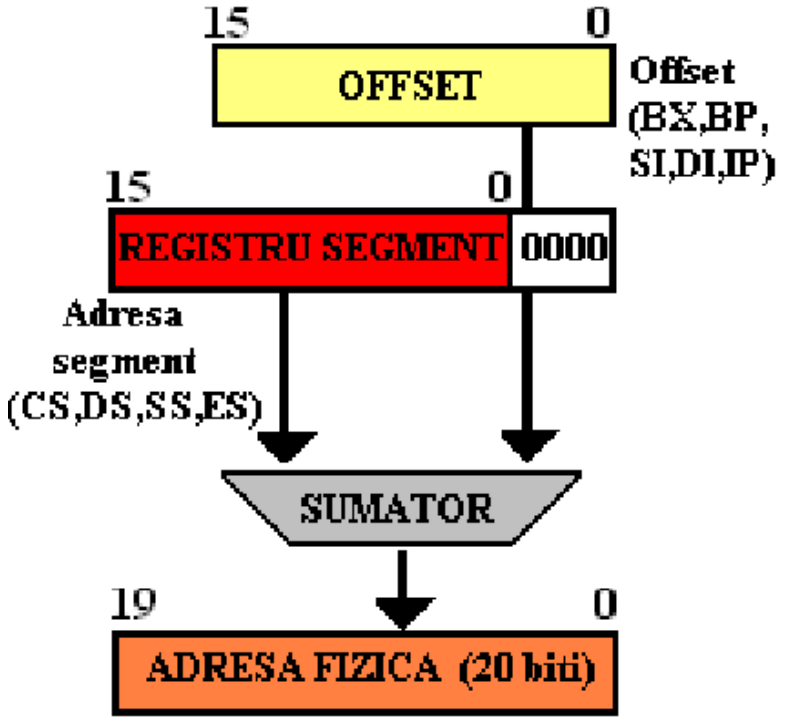
?

abcd0 h+
wxyz h



(calculat in hexa)

Calculul adresei fizice



Adrese logice valide:

- CS:IP – cod
- DS: offset (AE) – date
- SS:SP – stiva
- ES: offset (AE) – extra date

$$AE=[\ BX\ |\ BP\ ]_{opt}+[\ DI\ |\ SI\ ]_{opt}+[\text{deplasament 8/16 biţi}]_{opt}$$

AE este adresa efectiva a unui operand din memorie

- Segmentele sunt locatii contigue de memorie

(de tip ***paragraph***, adica **multiplu de 16** – de aici cele 4 zerouri in binar sau 1 zero in hexa)

- Segmentele au dimensiunea maxima de 64k
  (cat se poate cuprinde cu cei 16 biti ai offsetului)
- Segmentele pot fi adiacente, disjuncte, partial sau complet suprapuse

$$AF_{20}=16\bullet RS_{16}+offset_{16}$$

Calculul – realizat de BIU

Calculul adresei fizice (2)

Adresa = Segm : offset

Offset poate fi format din 3 categorii: deplasament + [BX]/[BP] + [SI]/[DI] **ADRESARE INDIRECTA**

Regula:

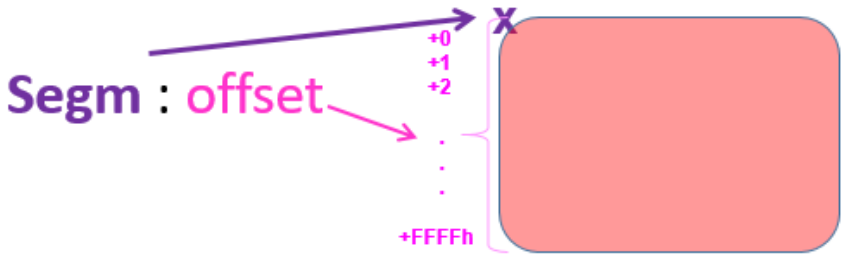
[BX/-]=>Segm=DS

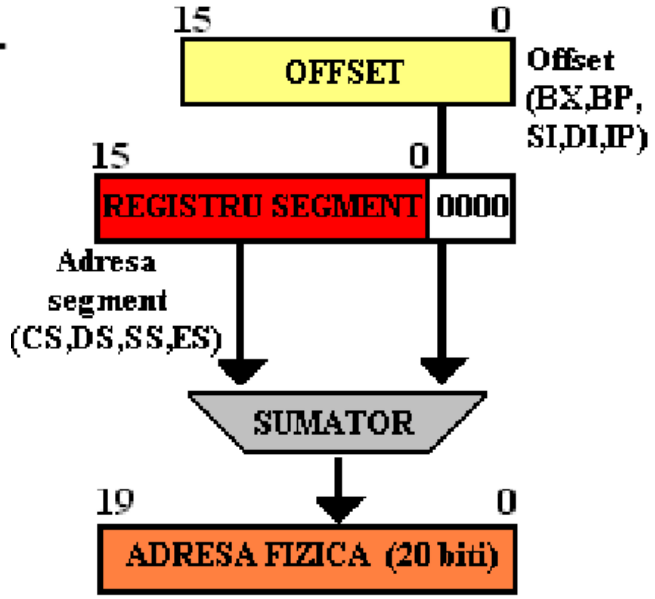
[BP] =>Segm=SS

Exemplu: 1200:0002 => 12000+0002=12002h

12000 h+
0002 h
12002 h

- 12000h+0000h = Adresa fizica de inceput a segmentului
- 12000h+FFFFh = Adresa fizica de sfarsit a segmentului





Adresa logica	Segment	Offset Implicit	Semnificatie
CS:IP	CS	IP	Adresa instructiune curenta
SS:SP	SS	SP	Adresa ultimului element introdus in stiva
SS:BP	SS	BP	Adresa primului element introdus in stiva
DS:offset	DS	BX sau nimic, DI sau SI, (+deplasament)	Adresa datei

Memoria

Zona de memorie – trebuie vazuta ca ***un dulap cu sertare***,
fiecare sertar avand dimensiunea unui octet

– in interiorul sertarului incape un singur octet si numai unul

Adresa sertarului = "locatie" = o eticheta

- **intr-o locatie din memorie incap 8 biti, adica un octet** (sau 1Byte in engl)

=> ***un cuvant (word)*** se va depune la ***2 octeti consecutivi (2B)***

⇒***un dublucuvant (doubleword)*** se va depune la ***4 octeti consecutivi (4B)***
in ordinea data de conventia **Little Endian**

⇒ La 8086 zona de memorie e constituita din ***SEGMENTE***

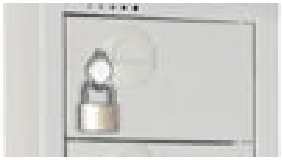
Exista segmente : **de cod**, **de date**, **de stiva**, **extra**

⇒ ***STIVA*** = tot zona de date, dar organizata dupa principiul **LIFO**

Vom spune ca avem 3 tipuri de segmente:

care contin COD, care contin DATE, care contin date din STIVA

Protectia datelor



Segmentul de cod – ***protejat*** – prin ***IP (instruction pointer)***

IP nu e accesibil programatorului, IP se incrementeaza/decrementeaza

in mod automat, deodata cu rularea codului !

Segmentul de stiva – ***protejat prin ordine si dimensiune*** !

SP (stack pointer)

BP (base pointer), BP=0000h

- Nu se pot depune/lua **octeti** (**B**-bytes),

ci **doar 2B, 4B, 8B – deci putere de-a lui 2**

DOAR CUVINTE (8086, 286):

2B, 2B, 2B, 2B, …

sau ***DUBLUCUVINTE*** (386) :

4B, 4B, 4B, 4B, …

sau ***CVADRUPLUCUVINTE*** (Core2)

8B, 8B, 8B, 8B, …

Care sunt “legale“ si care nu ?

Instructiuni de lucru cu stiva:

push operand si ***pop operand***

push operand – depune operandul pe stiva ca un nou element

pop operand – preia de pe stiva un element, depunandu-l in operand

Push AX
~~Push AL~~
Push EAX
Push BX
Push BP
~~Push CL~~

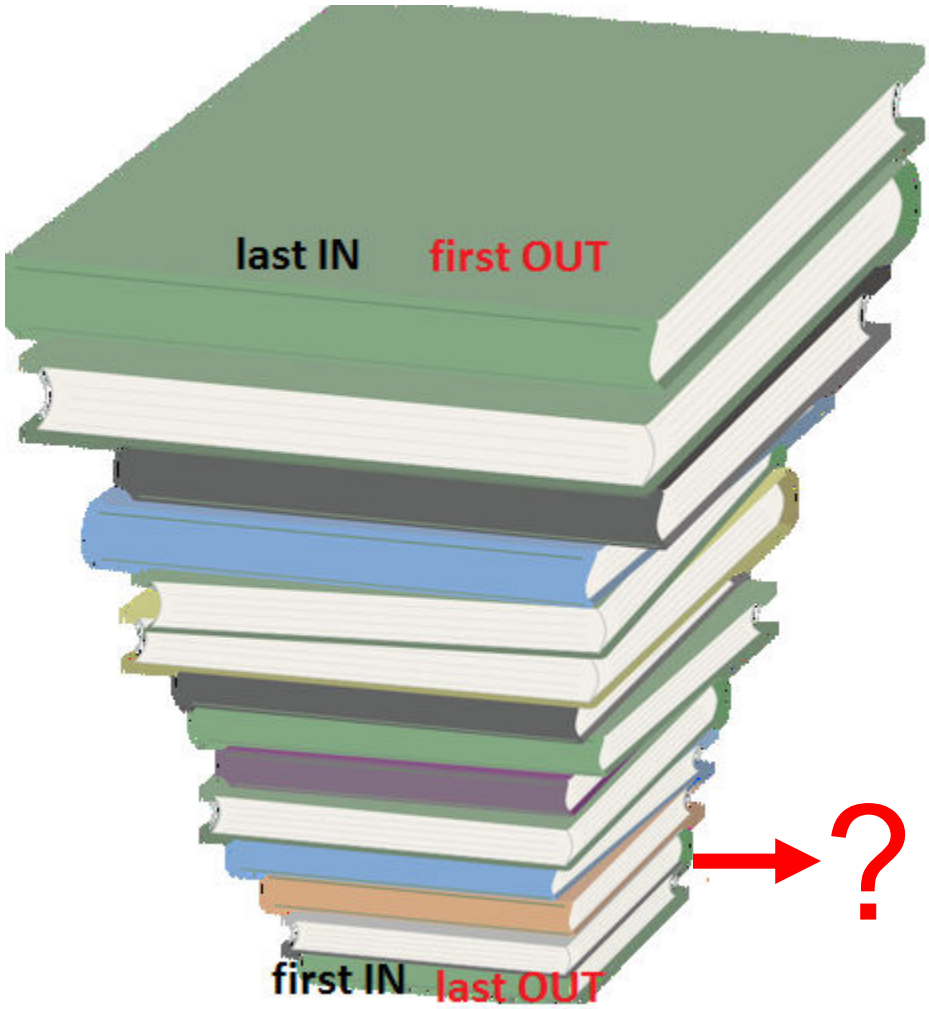
Push DX
~~Push DH~~
Push EAX
Push SI
Push DI
Push SP

similar cu ***pop***

! Operand = 16 biti *(de la 8086 ↑)*
32 biti *(de la 386 ↑)*

Principiul de lucru: *la care* din ele, *cum* avem acces ?

CODUL (8b) **DATELE (8b)** **STIVA (8b) - LIFO**

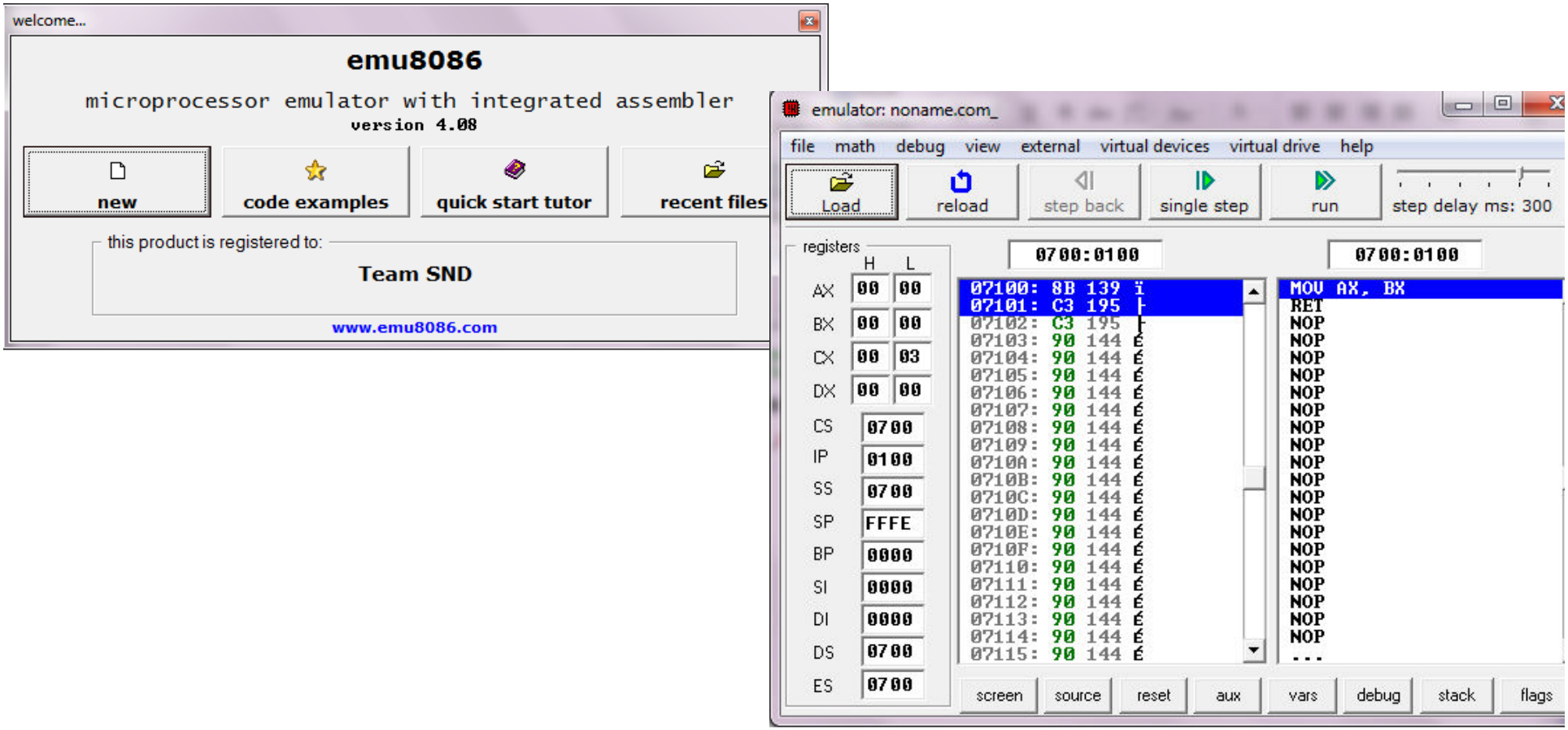


CS:IP **DS: offset sau ES:offset** **SS: BP sau SS:SP**

In aranjarea datelor in memorie, indifferent ca sunt in **zona de date** sau **zona de stiva**, se aplica mereu aceeasi conventie: la 80x86 – **Little END - ian**

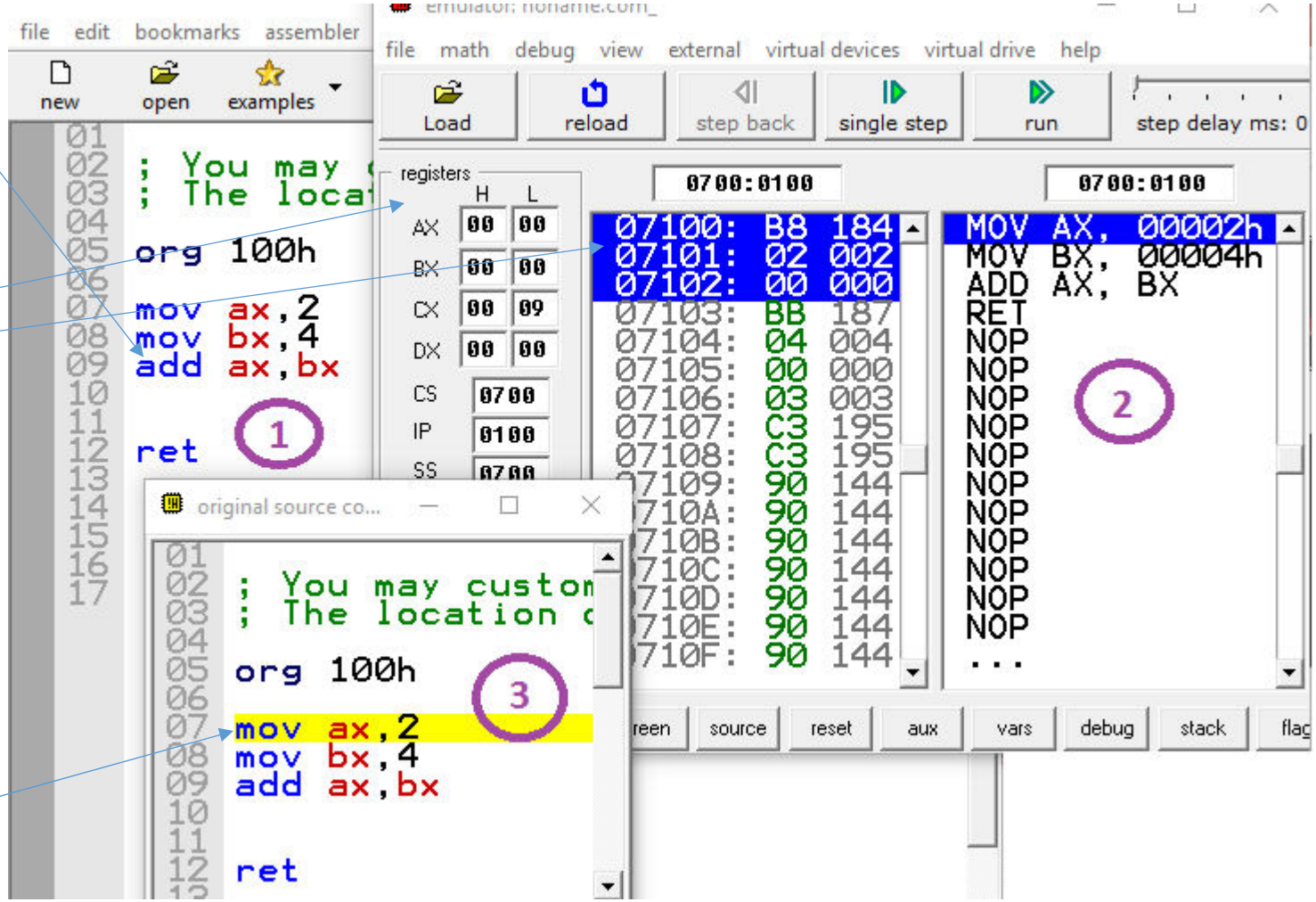
EMU8086

Simulator de procesor pe .???. Biti



EMU – 3 ferestre principale

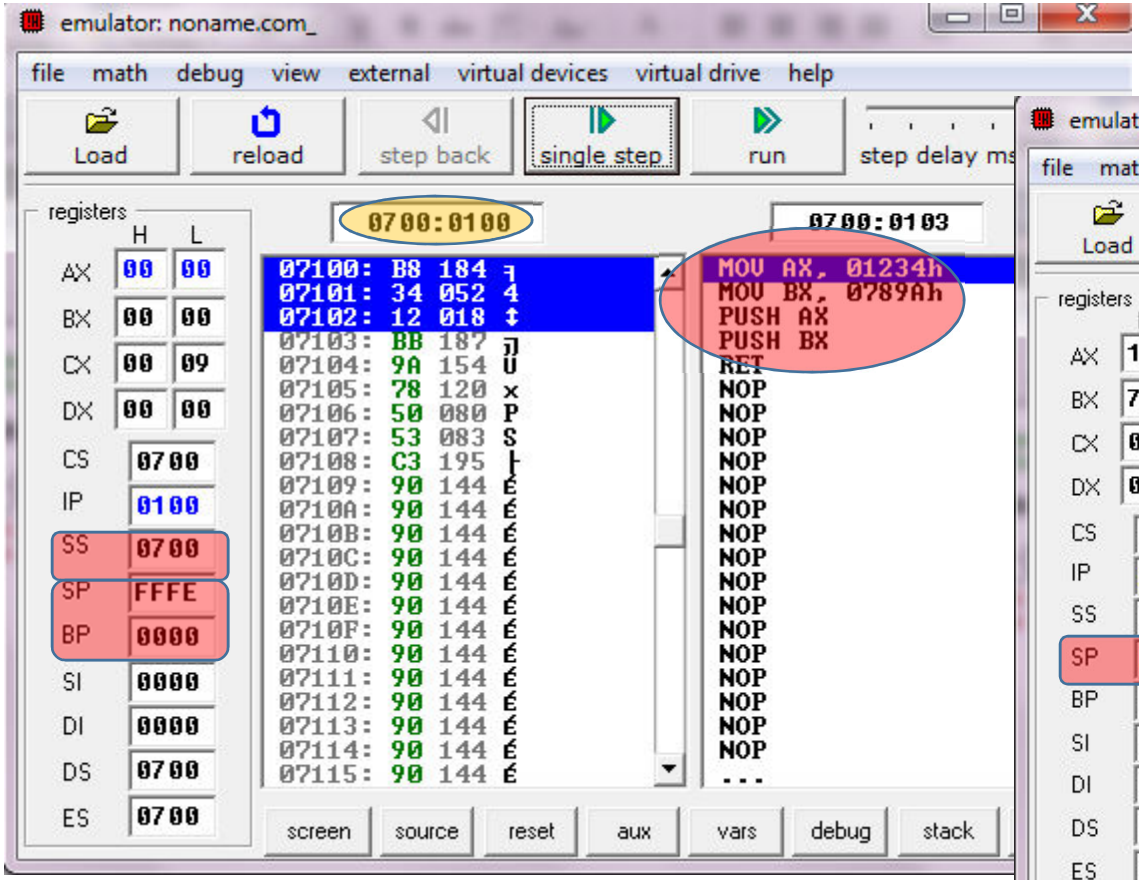
- **1**- unde ***se scrie codul*** efectiv
- **2**- unde ***se observa continutul registrilor, memoriei***, etc si se comanda executia programului
- **3** – unde *se* ***observa instructiunea ce urmeaza a se executa***



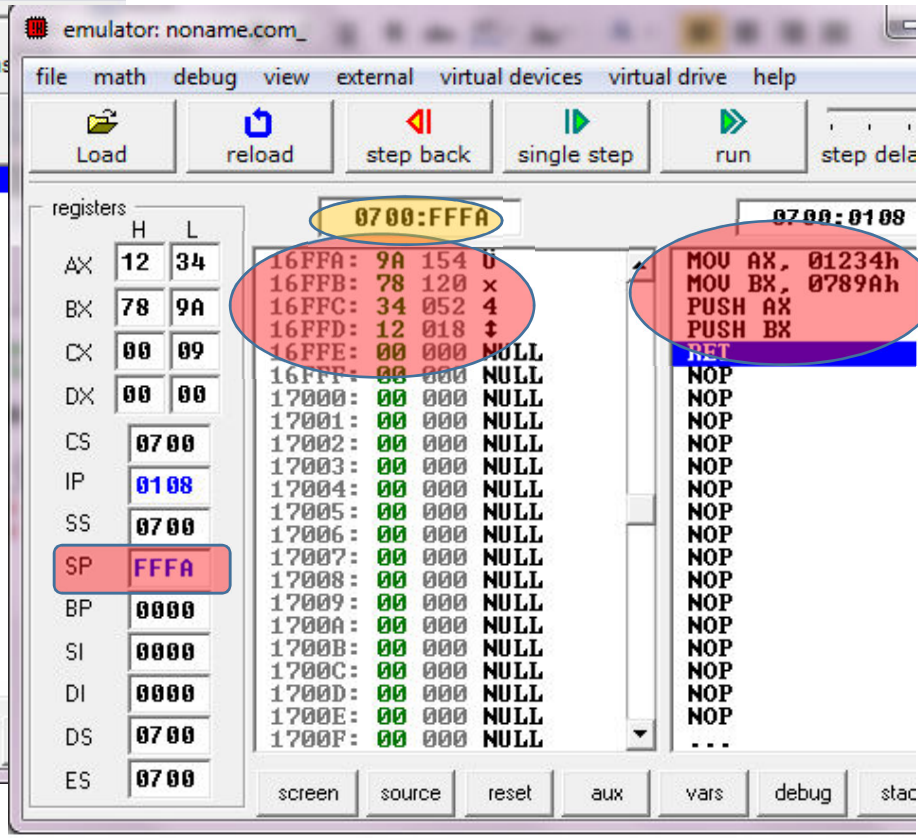
EMU – stiva si “memoria”

Stiva nu e tot in memorie ???? R: ba da (in segm SS), doar ca zona respectiva din memorie se considera de tip LIFO! => lucrul cu ea e diferit (stiva e gestionata altfel decat o zona “*normala*” din memorie)

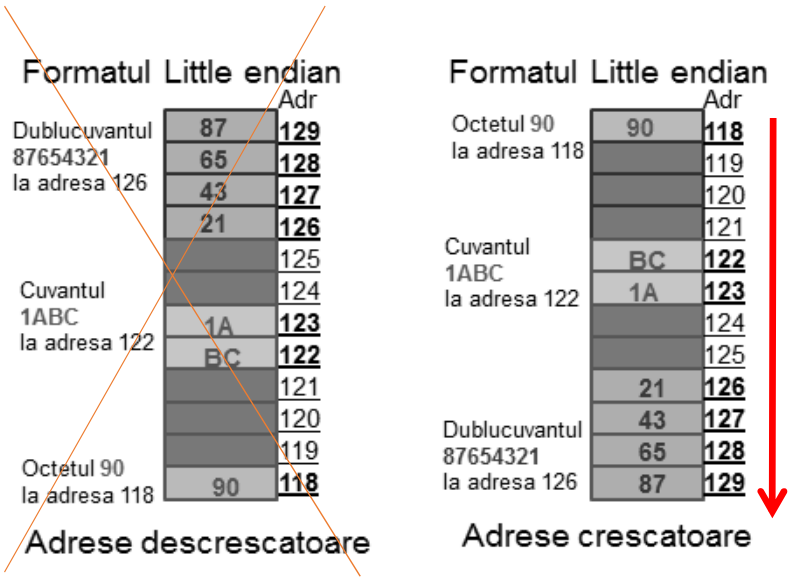
- Inainte de executie:



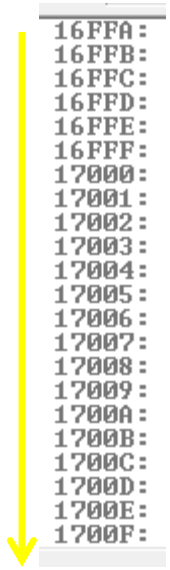
dupa executie:



Depunerea datelor in stiva

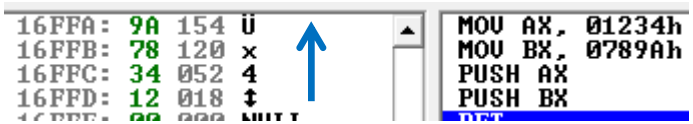


Adresele in EMU



- In memorie: - zona aleatoare
- informatia se insereaza ↓
- (de la adrese mai mici spre adrese mai mari)

- In stiva: LIFO
- informatia se insereaza ↑
- (de la adrese mai mari spre adrese mai mici)



Cum se calculeaza adresa fizica ?

Segm:offset =abcd:wxyz => abcd0+wxyz (calculat in hexa)

abcd0h+
__wxyz

Exemplu 0700h:FFFAh => **16FFA**h pe 20 biti (rezultata din 16b:16b)

Calculul adresei unde se depune informatia

- **Exercitiu :**

Fie registrele CPU in urmatoarea stare:

CS=3456h

DS=1234h

SS=9876h

SP=1210h

DI=3400h

BX=5600h

IP=2400h

Adresa instructiunii urmatoare va fi :
CS:IP adica 3456h:2400h => 36960h

34560 h +
2400 h
36960 h

Adresa unde se afla datele va fi:
Data de DS:DI sau DS:BX sau o combinatie a lor, depinde de modul de adresare.
De ex. avem mov AX, [BX] ; atunci ***data*** va fi din DS:BX, deci 1234h:5600h => 17940h

12340 h +
5600 h
17940 h

Adresa ultimului element introdus in stiva:
SS:SP, deci 9876h:1210h => 99970h

98760 h +
1210 h
99970 h

Adresa primului element introdus in stiva:
SS:BP, deci 9876h:0000h => 98760h

98760 h +
1200 h
99960 h

CODUL
CS:IP

DATELE
DS: offset sau ES:offset

STIVA - LIFO
SS: BP sau SS:SP

Scenariul 1 (*pe jos* tot drumul) versus
Scenariul 2 (cu autobuzul cateva statii) si de acolo *pe jos*

START

STOP

Adresa ***fizica*** – raportat la ***inceput***
(incrementarea incepe de la inceputul zonei de memorie)

Scenariul 1: **36010 pasi**

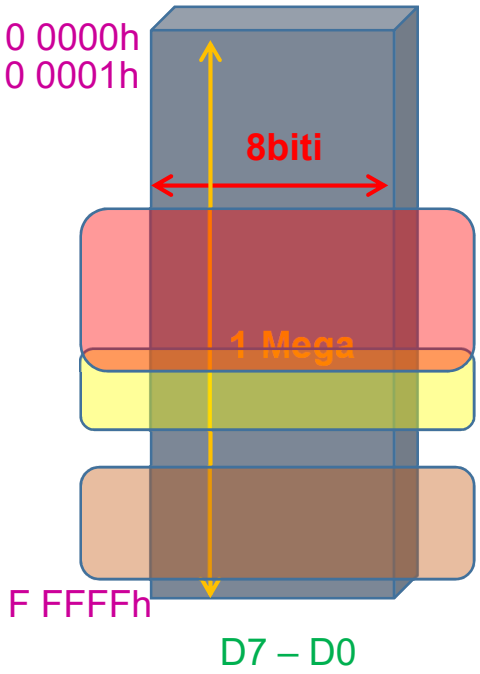



Adresa ***relativa*** – raportat la "***statie***"
(incrementarea incepe de la inceputul zonei segment)

Scenariul 2: **3 *statii cu autobuzul*** + ***10 pasi***

Adresarea segmentata

Logic: Memoria la ***8086***



1 Mega = 2^{20} => sunt necesari 20 biti pt adresare -> **5 cifre hexa**

Dar operatiile se pot realiza doar pe 16 biti ?????

(registrele interne sunt pe maxim 16 biti, fiind procesor pe 16 biti)

Solutia: in loc sa se multumeasca sa acceseze doar ***64kocteti*** o data,
au propus ***adresarea segmentata***

(segmentul = ca o zona de interes la momentul curent = ca o LUPA)

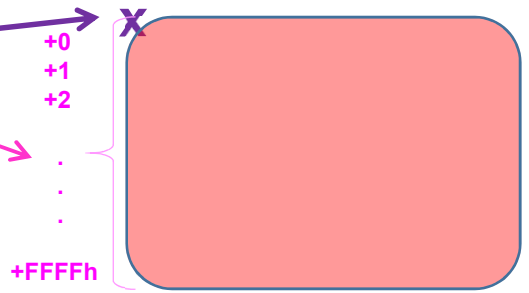
DOAR ca pot exista :

Segmente de **COD**, **DATE**, **STIVA**, **EXTRA**

Pot avea orice lungime = Multiplu(16), dar maxim 64k in interior

➔ o **ADRESA FIZICA** se va exprima sub forma **ADRESA LOGICA**:

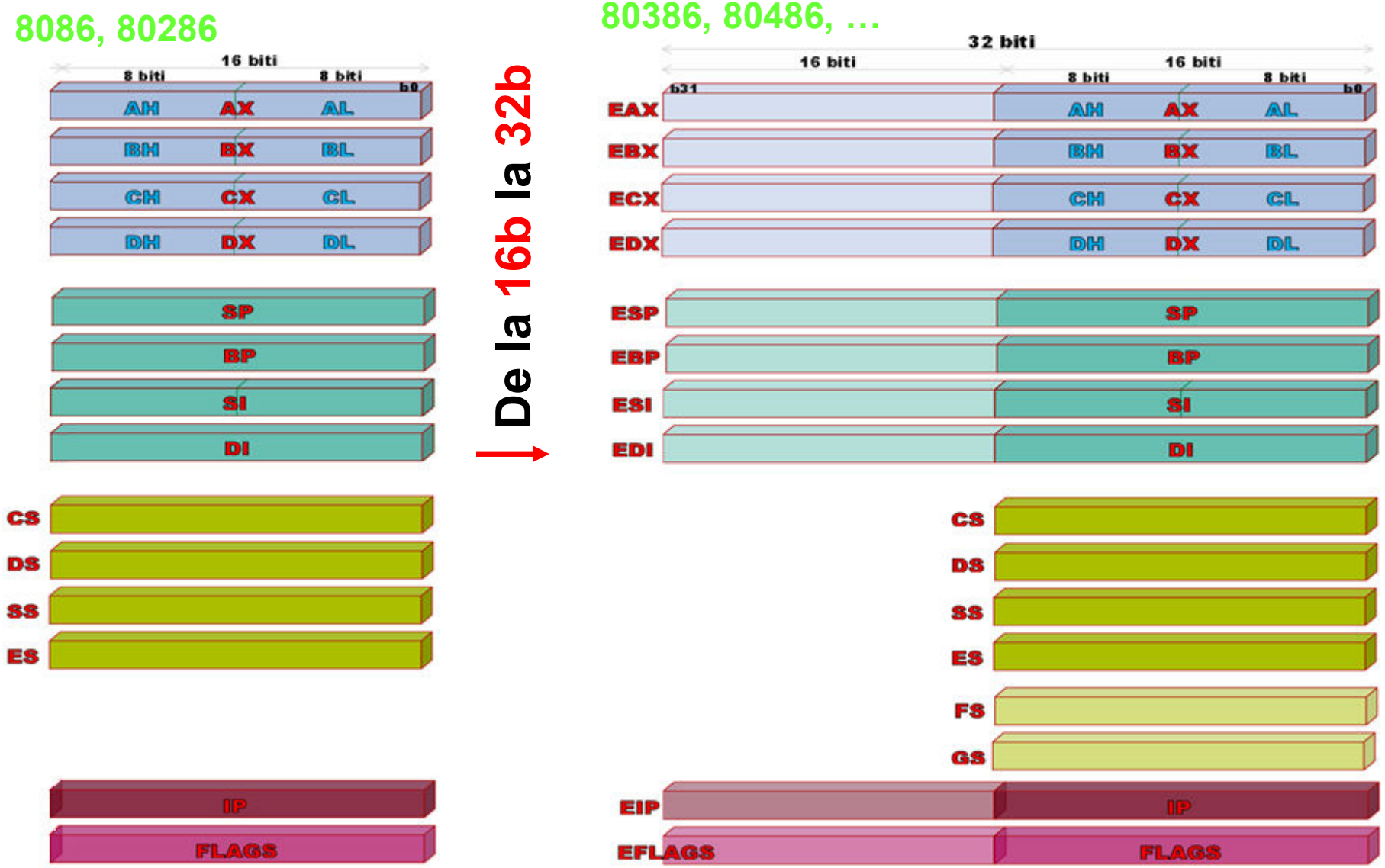
Segm : offset



Adresare ***relativa*** la segment

(raportat la adresa de inceput a segmentului
Si NU la cea de inceput a intregii zone de memorie)

Trecerea de la CPU pe 16 biti la CPU pe 32 biti

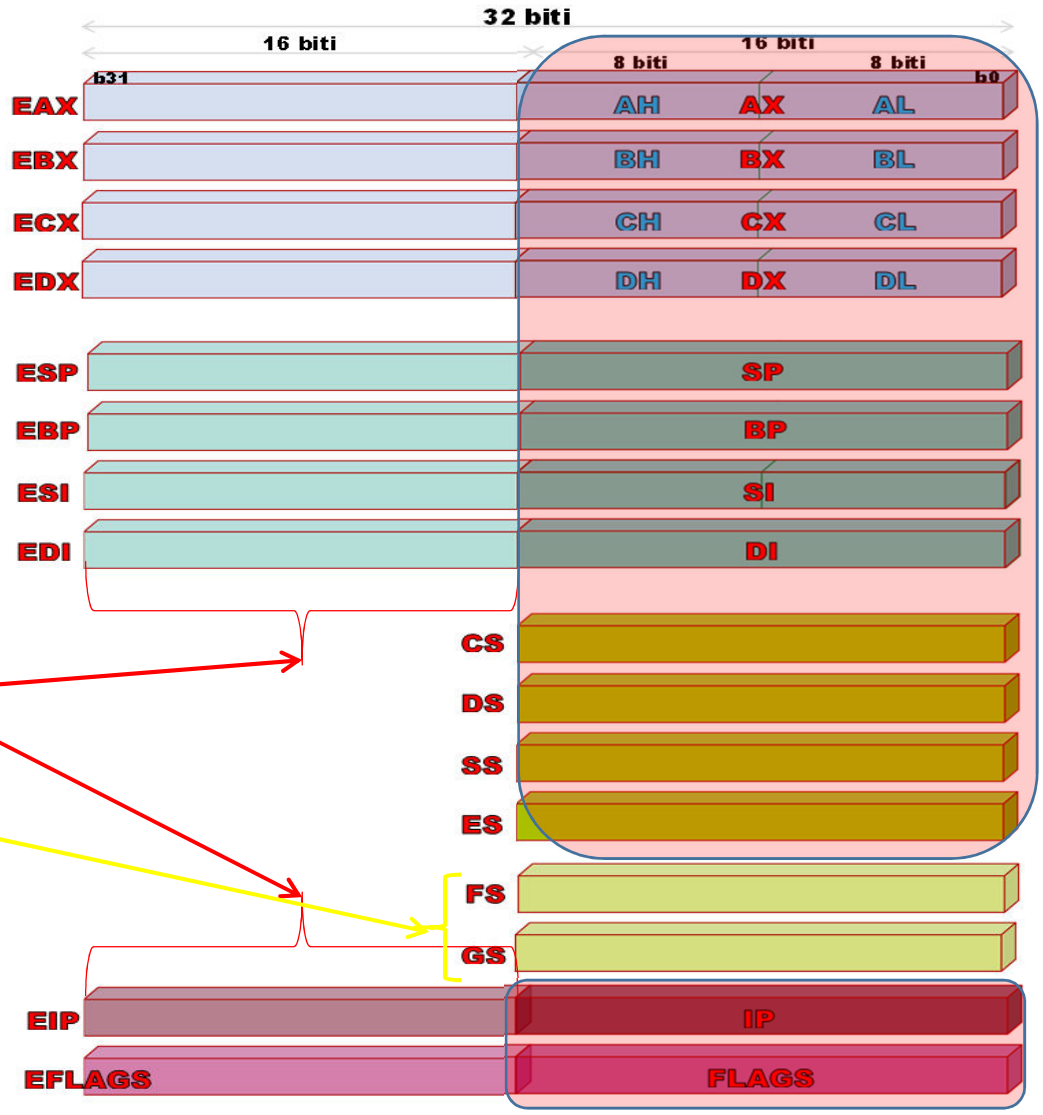


Anca APATEAN -UTCN

Registrii procesorului **80386**
- sugereaza evolutia familiei 80x86
de la **16** biti la **32** biti

-> s-au **extins** toti registrii
(cu exceptia registrilor segment)
de la 16 biti la 32 biti
- extensie specificata prin prefixarea numelor cu litera
E („extended”)

- jumatatea mai transparenta a registrilor
ilustreaza evolutia de la 8086 la 80386:
-> a crescut:
- atat **dimensiunea** unor registri
(de la 16 biti la 32 biti)
- cat si **nr registrilor** segment



- 8086, 80286

Arhitectura procesoarelor din familia 80x86

Evolutia familiei 80x86 pana la 32 biti (2)

Procesoarele din familia 80x86 au ***3 categorii*** principale de *registri*:

I. registrii de UZ GENERAL

II. registrii SEGMENT

III. registrii SPECIALI

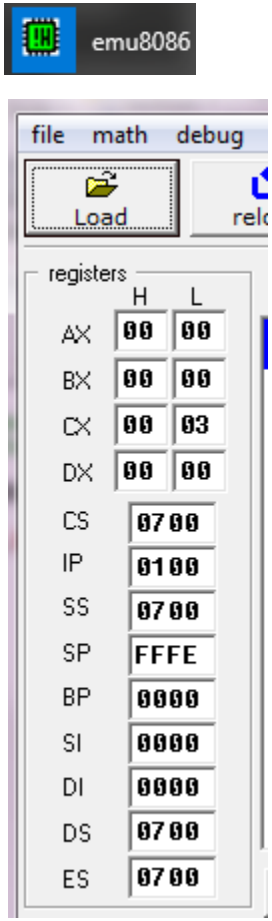
- setul de registri al fiecarui procesor este de fapt **un *superset* al procesoarelor aparute anterior.**

Tabelul 1. Regiştrii procesoarelor din familia 80x86

Registru	Nr biti	Nume Registru	Aparut incepand cu CPU
Registrii de uz general (GPR)			
Date	**8**	**AL,BL,CL,DL,AH,BH,CH,DH**	8086↑
	16	**AX,BX,CX,DX**	8086↑
	32	EAX,EBX,ECX,EDX	80386↑
Pointer	**16**	**SP,BP**	8086↑
	32	ESP,EBP	80386↑
Index	**16**	**SI,DI**	8086↑
	32	ESI,EDI	80386↑
Registrii segment			
Segment	**16**	**CS,DS,SS,ES**	8086↑
	16	FS,GS	80386↑
Registrii speciali			
Pointer la instructiune	**16**	**IP**	8086↑
	32	EIP	80386↑
Indicatori de conditii	**16**	**Flags**	8086↑
	32	Eflags	80386↑

EMU8086

Simulator de procesor pe 16 biti



Ce categorii de registri apar?

De uz general:

Date: *litera **X***

Pointer: *litera **P***

Index: *litera **I***

Segment: *litera **S***

Speciali:

"pointer la instruct":

"*indicator*"ilor de conditii

Registru	Nr biti	Nume Registru	Aparut incepand cu CPU
Registrii de uz general (GPR)			
Date	**8**	**AL,BL,CL,DL,AH,BH,CH,DH**	8086↑
	16	**AX,BX,CX,DX**	8086↑
	32	EAX,EBX,ECX,EDX	80386↑
Pointer	**16**	**SP,BP**	8086↑
	32	ESP,EBP	80386↑
Index	**16**	**SI,DI**	8086↑
	32	ESI,EDI	80386↑
Registrii segment			
Segment	**16**	**CS,DS,SS,ES**	8086↑
	16	FS,GS	80386↑
Registrii speciali			
Pointer la instructiune	**16**	**IP**	8086↑
	32	EIP	80386↑
Indicatori de conditii	**16**	**Flags**	8086↑
	32	Eflags	80386↑

Arhitectura procesoarelor din familia 80x86

Evolutia familiei 80x86 pana la 32 biti (3)

Anca APATEAN -UTCN

I. Registrii de uz general (GPR, General Purpose Registers)
pot fi impartiti in: registrii **DE DATE**, reg. **POINTER**, reg. **INDEX**
(ultimele 2 categorii se mai numesc "de adresare")

Registru	Nr biti	Nume Registru	Aparut incepand cu CPU
Registrii de uz general (GPR)			
Date	8	AL,BL,CL,DL,AH,BH,CH,DH	8086↑
	16	AX,BX,CX,DX	8086↑
	32	EAX,EBX,ECX,EDX	80386↑
Pointer	16	SP,BP	8086↑
	32	ESP,EBP	80386↑
Index	16	SI,DI	8086↑
	32	ESI,EDI	80386↑
Registrii segment			
Segment	16	CS,DS,SS,ES	8086↑
	16	FS,GS	80386↑
Registrii speciali			
Pointer la instructiune	16	IP	8086↑
	32	EIP	80386↑
Indicatori de conditii	16	Flags	8086↑
	32	Eflags	80386↑

I.a) Regiştrii generali DE DATE

- fol pt a stoca temporar operanzi şi rezultate în CPU.

La procesoarele 8086,8088,80186,80188 acesti registri au o dimensiune de 16 biti, avand *funcţii implicite*:

AX – registru "accumulator"
– folosit la înmulţiri şi împărţiri pe 8 biţi, conversii, etc

BX – registru de bază în adresare la operaţii cu memoria, dar poate fi folosit şi la translaţii

CX – registru contor – folosit în general în operaţii cu şiruri sau bucle

DX – registru de date (se mai num. *extensia accumulatorului*)
- este folosit la adresare indirectă a porturilor, la înmulţiri şi împărţiri pe 16 biţi.

Arhitectura procesoarelor din familia 80x86

Evolutia familiei 80x86 pana la 32 biti (3)

Anca APATEAN -UTCN

I. Registrii de uz general (GPR, General Purpose Registers)
I.a) Regiştrii generali DE DATE

Primii 4 registri pot fi adresati la nivel de:
octet (ex: AL, AH, BL, BH, …), deci pe 8 biţi
cuvint (ex:AX, BX, CX, DX), deci pe 16 biţi
(sau dublu-cuvint(EAX, EBX, …)- de la 386↑)

L, H= părţile *LOW*, resp *HIGH*
E - "extended" (extins).

Incepand de la procesoarele 80386↑,
registrii de date au fost **extinsi la 32 biti,**

Registru	Nr biti	Nume Registru	Aparut incepand cu CPU
Registrii de uz general (GPR)			
Date	8	AL,BL,CL,DL,AH,BH,CH,DH	8086↑
	16	AX,BX,CX,DX	8086↑
	32	EAX,EBX,ECX,EDX	80386↑
Pointer	16	SP,BP	8086↑
	32	ESP,EBP	80386↑
Index	16	SI,DI	8086↑
	32	ESI,EDI	80386↑
Registrii segment			
Segment	16	CS,DS,SS,ES	8086↑
	16	FS,GS	80386↑
Registrii speciali			
Pointer la instructiune	16	IP	8086↑
	32	EIP	80386↑
Indicatori de conditii	16	Flags	8086↑
	32	Eflags	80386↑

Arhitectura procesoarelor din familia 80x86

Evolutia familiei 80x86 pana la 32 biti (4)

Anca APATEAN -UTCN

I. Registrii de uz general (GPR, General Purpose Registers)

I.b) Regiştrii POINTER

SP şi BP - **destinaţi lucrului cu stiva,**
dar pot fi folosiţi şi în anumite instrucţiuni de transfer, aritmetice şi logice ca regiştrii pe 16 biţi.

Registrul SP (Stack Pointer) implicit pointează spre ultimul element introdus în stivă

Registrul BP (Base Pointer) implicit pointează către baza stivei

(BP mai poate fi folosit şi pentru adresarea în cadrul altor segmente).

Registru	Nr biti	Nume Registru	Aparut incepand cu CPU
Registrii de uz general (GPR)			
Date		AL,BL,CL,DL,AH,BH,CH,DH	8086↑
		AX,BX,CX,DX	8086↑
		EAX,EBX,ECX,EDX	80386↑
Pointer	16	SP,BP	8086↑
	32	ESP,EBP	80386↑
Index		SI,DI	8086↑
		ESI,EDI	80386↑
Registrii segment			
Segment	16	CS,DS,SS,ES	8086↑
	16	FS,GS	80386↑
Registrii speciali			
Pointer la instructiune	16	IP	8086↑
	32	EIP	80386↑
Indicatori de conditii	16	Flags	8086↑
	32	Eflags	80386↑

Incepand de la procesoarele **80386↑**
registrii pointer au fost extinsi la 32 biti: ESP şi EBP

Arhitectura procesoarelor din familia 80x86

Evolutia familiei 80x86 pana la 32 biti (4)

Anca APATEAN -UTCN

I. Registrii de uz general (GPR, General Purpose Registers)

I.c) Regiştrii INDEX –

SI şi **DI** (**Source Index** şi **Destination Index**)

- destinaţi transferurilor sau lucrului cu şiruri de octeţi sau cuvinte.

Regiştrii SI/DI **conţin adresa relativă curentă a şirului *sursă*/*destinaţie*,** putând fi utilizaţi şi la instrucţiuni de transfer, aritmetice sau logice sau pentru adresare.

Incepand de la procesoarele 80386↑ registrii index au fost extinsi la 32 biti **ESI**, **EDI**.

Reg pointer si **reg index** se mai numesc generic ***registrii de adresare*** deoarece pe lângă utilizarea lor implicită, mai pot fi folosiţi în anumite instrucţiuni aritmetice şi logice.

Registru	Nr biti	Nume Registru	Aparut incepand cu CPU
Registrii de uz general (GPR)			
Date		AL,BL,CL,DL,AH,BH,CH,DH	8086↑
		AX,BX,CX,DX	8086↑
		EAX,EBX,ECX,EDX	80386↑
Pointer		SP,BP	8086↑
		ESP,EBP	80386↑
Index	16	**SI,DI**	8086↑
	32	**ESI,EDI**	80386↑
Registrii segment			
Segment	16	CS,DS,SS,ES	8086↑
	16	FS,GS	80386↑
Registrii speciali			
Pointer la instructiune	16	IP	8086↑
	32	EIP	80386↑
Indicatori de conditii	16	Flags	8086↑
	32	Eflags	80386↑

Anca APATEAN -UTCN

II. Regiştrii SEGMENT

Proc. 8086,8088,80186 si 80188 au definite 4 tipuri de registri segment:

CS (Code Segment)
- adresa segmentului de cod sau program
  - conţine instrucţiuni,

DS (Data Segment)
- adresa segmentului de date curent
  – contine variabilele programului

SS (Stack Segment)
- adresa segmentului de stivă curent şi

ES (Extra Segment)
- adresa segmentului de date suplimentar.

Registru	Nr biti	Nume Registru	Aparut incepand cu CPU
Registrii de uz general (GPR)			
Date		AL,BL,CL,DL,AH,BH,CH,DH	8086↑
		AX,BX,CX,DX	8086↑
		EAX,EBX,ECX,EDX	80386↑
Pointer		SP,BP	8086↑
		ESP,EBP	80386↑
Index		SI,DI	8086↑
		ESI,EDI	80386↑
Registrii segment			
Segment	16	CS,DS,SS,ES	8086↑
	16	FS,GS	80386↑
Registrii speciali			
Pointer la instructiune	16	IP	8086↑
	32	EIP	80386↑
Indicatori de conditii	16	Flags	8086↑
	32	Eflags	80386↑

Anca APATEAN -UTCN

II. Regiştrii SEGMENT

CS contine ***codul ce se executa la momentul curent***.
CPU realizeaza o operatie de „fetch" (aducere) asupra instructiunii data de **CS:IP**.

Regiştrii segment au o dimensiune de 16 biti si reţin **adresele de început ale segmentelor active**, corespunzătoare fiecărui tip de segment.

În fiecare moment al execuţiei este declarat ACTIV câte ***un singur segment din fiecare tip***.

Incepand de la procesoarele 80386 au fost adaugati 2 registrii segment suplimentari: **FS**, **GS**.

Registru	Nr biti	Nume Registru	Aparut incepand cu CPU
Registrii de uz general (GPR)			
Date		AL,BL,CL,DL,AH,BH,CH,DH	8086↑
		AX,BX,CX,DX	8086↑
		EAX,EBX,ECX,EDX	80386↑
Pointer		SP,BP	8086↑
		ESP,EBP	80386↑
Index		SI,DI	8086↑
		ESI,EDI	80386↑
Registrii segment			
Segment	**16**	**CS,DS,SS,ES**	8086↑
	16	**FS,GS**	80386↑
Registrii speciali			
Pointer la instructiune	16	IP	8086↑
	32	EIP	80386↑
Indicatori de conditii	16	Flags	8086↑
	32	Eflags	80386↑

Anca APATEAN -UTCN

III. Registrii speciali

III.a) Registrul POINTER LA INSTRUCTIUNE – IP (Instruction Pointer)

conţine offsetul (adresa relativă) instrucţiunii curente (***instructiunea ce urmează să se execute***) în cadrul **segmentului de cod curent (CS)**,

- e **manipulat exclusiv de către BIU** (programele nu au acces direct la IP).

În mod normal, conţinutul registrului **IP** este ***incrementat*** pe măsură ce instrucţiunile sunt executate (secvenţial) cu numărul de octeţi ai instrucţiunii curente,

Registru	Nr biti	Nume Registru	Aparut incepand cu CPU
Registrii de uz general (GPR)			
Date		AL,BL,CL,DL,AH,BH,CH,DH	8086↑
		AX,BX,CX,DX	8086↑
		EAX,EBX,ECX,EDX	80386↑
Pointer		SP,BP	8086↑
		ESP,EBP	80386↑
Index		SI,DI	8086↑
		ESI,EDI	80386↑
Registrii segment			
Segment		CS,DS,SS,ES	8086↑
		FS,GS	80386↑
Registrii speciali			
Pointer la instructiune	**16**	**IP**	8086↑
	32	**EIP**	80386↑
Indicatori de conditii	**16**	**Flags**	8086↑
	32	**Eflags**	80386↑

dar există şi instrucţiuni care modifică registrul **IP** în diverse moduri, nesecvenţial (de exemplu instrucţiunile de salt JMP, CALL, J[condiţie], RET, IRET).

Incepand de la procesoarele 80386, registrul IP a fost extins la 32 biti si se numeste EIP.

Anca APATEAN -UTCN

III. Registrii speciali

III.b) Registrul INDICATORILOR DE CONDITII

– **Flags** sau **PSW (Program Status Word)**
= un registru special pe 16 biti,
ce contine un numar de 9 indicatori de condiţii (de stare şi control) la 8086.

Un ***flag*** = ***un indicator*** reprezentat pe un bit.
(ca un LED)

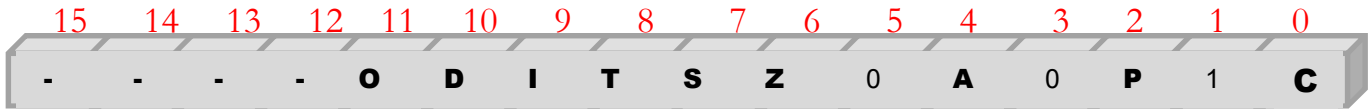
Indicatorii de stare ai PSW ***sintetizează execuţia ultimei instrucţiuni***, ei reţinând **starea curentă a procesorului**.

Registru	Nr biti	Nume Registru	Aparut incepand cu CPU
Registrii de uz general (GPR)			
Date		AL,BL,CL,DL,AH,BH,CH,DH	8086↑
		AX,BX,CX,DX	8086↑
		EAX,EBX,ECX,EDX	80386↑
Pointer		SP,BP	8086↑
		ESP,EBP	80386↑
Index		SI,DI	8086↑
		ESI,EDI	80386↑
Registrii segment			
Segment		CS,DS,SS,ES	8086↑
		FS,GS	80386↑
Registrii speciali			
Pointer la instructiune	**16**	**IP**	8086↑
	32	**EIP**	80386↑
Indicatori de conditii	**16**	**Flags**	8086↑
	32	**Eflags**	80386↑

Anca APATEAN -UTCN

Registrul Indicatorilor de Conditii–Flags/PSW(Program Status Word)

- contine indicatori de conditie (*bistabile*) a caror stare se modifica in urma executiei unor instructiuni



INDICATORII DE STARE - pe bitii **0,2,4,6,7** si **11** :

C – Carry –se seteaza (devine 1) daca in urma unei operatii aritmetice a aparut un **transport (carry) /un imprumut (borrow)** în afara domeniului de reprezentare a rezultatului

P – Parity –se seteaza dacă **nr de biţi de 1** al rezultatului **este par**

A – Auxiliary – indică valoarea **transportului (carry/ borrow) intre bitii 3 si 4** (între cifrele hexazecimale)- folosit in special la cifrele BCD;

Z – Zero – are val 1 dacă rezultatul ultimei instrucţiuni este **egal cu zero**

S – Sign –se seteaza când rezultatul ultimei operaţii este **un nr strict negativ** adică copiază bitul MSB al rezultatului;

O – Overflow – indică **depăşire de gamă** dacă rezultatul ultimei instrucţiuni a depăşit spaţiul rezervat rezultatului, în cazul operanzilor consideraţi numere cu semn (dacă s-a modificat semnul rezultatului):

$$O=C \oplus transport_{MSB-1,\ MSB}$$

INDICATORII DE CONTROL - pe bitii **8,9,10**:

T – Trap – flag pt depanare; dacă are valoarea 1, atunci procesorul se opreşte după fiecare instrucţiune, permitand executia pas cu pas;

I – Interrupt – flag de întrerupere ce controleaza raspunsul CPU la cererea de intreruperi mascabile: permite (dacă IF=1) sau invalidează (dacă IF=0) acceptarea întreruperilor externe mascabile care apar pe intrarea INT a CPU;
- nu afectează întreruperile interne/ externe nemascabile

D - Direction – flag fol la lucrul cu şiruri pt a indica direcţia de parcurgere de-a lungul şirului
- D=0 pt adrese crescătoare,
- D=1 pt adrese descrescătoare.

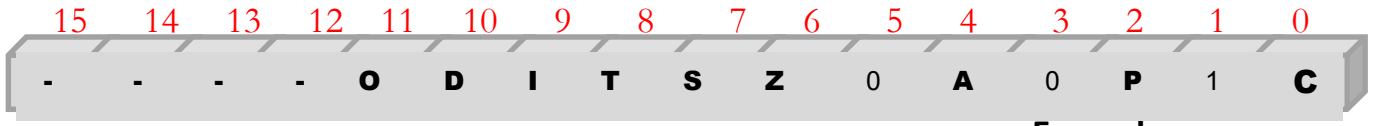
Trap si Interrupt se mai numesc si ***flaguri de sistem***.
Incepand cu 80286, au fost implementati si indicatorii de sistem de pe bitii **12** +**13** (IOPL) si **14** (NT),

incepand de la procesoarele 80386, reg. indicatorilor de conditii a fost **extins la 32 biti** (**E**FLAGS)

Anca APATEAN -UTCN

Registrul Indicatorilor de Conditii–Flags/PSW(Program Status Word)

- contine indicatori de conditie (*bistabile*) a caror stare se modifica in urma executiei unor instructiuni



INDICATORII DE STARE - pe bitii **0,2,4,6,7** si **11** :

C – Carry –se seteaza (devine 1) daca in urma unei operatii aritmetice a aparut un **transport (carry) /un imprumut (borrow)** în afara domeniului de reprezentare a rezultatului

P – Parity –se seteaza dacă **nr de biţi de 1** al rezultatului **este par**

A – Auxiliary – indică valoarea **transportului (carry/ borrow) intre bitii 3 si 4** (între cifrele hexazecimale)- folosit in special la cifrele BCD;

Z – Zero – are val 1 dacă rezultatul ultimei instrucţiuni este **egal cu zero**

S – Sign –se seteaza când rezultatul ultimei operaţii este **un nr strict negativ** adică copiază bitul MSB al rezultatului;

O – Overflow – indică **depăşire de gamă** dacă rezultatul ultimei instrucţiuni a depăşit spaţiul rezervat rezultatului, în cazul operanzilor consideraţi numere cu semn (dacă s-a modificat semnul rezultatului):

$$O=C \oplus transport_{MSB-1,\ MSB}$$

Exemple:

a) 02h+02h=04h -> C=0, Z=0, S=0, O=0

b) 03h+7Ch=7Fh -> C=0, Z=0, S=0, O=0

c) 04h+7Ch=80h -> C=0, Z=0, S=1, O=1

0000 0100b+

0111 1100b

1000 0000b

$O=C \oplus transport_{MSB-1,\ MSB}=0+1$

d) 80h+80h=00h -> C=1, Z=1, S=0, O=1

0000 0000b+

0000 0000b

0000 0000b

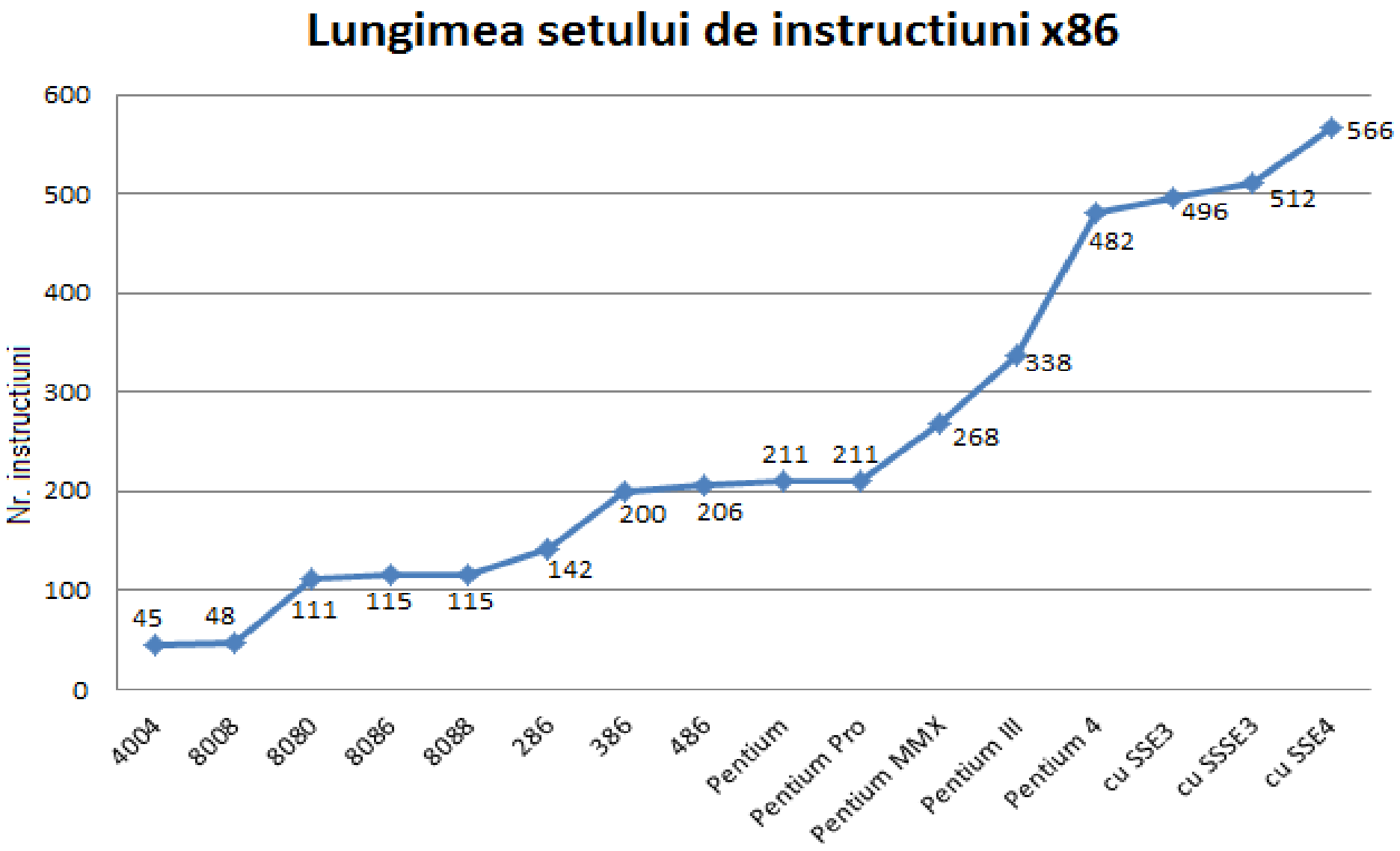
$O=C \oplus transport_{MSB-1,\ MSB}=1+0=1$

Exercitii propuse (0.25p fiecare):
Folosind registre pe 8 biti:
a)7Ah+02h= b)6Bh+14h=
c) 5Bh+2Dh= d) 63h+1Dh=
Folosind registre pe 16 biti
e) 0A4B5h+5B4Bh= f) 0B3A4h+4C5Ch=
g) 3a54h+100h= h) 7965h+210h=
Reanalizati punctele a) …d) folosind registre pe 16 biti. Ce observati

Setul de instructiuni 80x86

Instrucţiuni			Observaţii
de transfer	**generale**	MOV, MOVSX, MOVZX, XCHG	MOVSX, MOVZX - 80386↑
	cu stiva	PUSH, POP, PUSHA,POPA, PUSHAD, POPAD	PUSH, POP - 80186↑ PUSHAD, POPAD - 80386↑
	cu acumulatorul	XLAT, IN, OUT	
	pt adrese	LEA, LDS, LES, LFS, LGS, LSS	LFS, LGS, LSS - 80386↑
	cu flaguri	LAHF,SAHF,PUSHF,POPF, PUSHFD,POPFD	PUSHFD, POPFD - 80386↑
aritmetice	**pt adunare**	ADD, XADD, ADC, INC	XADD - 80486↑
	pt scădere	SUB, SBB, DEC	
	pt negare	NEG	
	pt înmulţire	MUL, IMUL	IMUL - diverse forme - 80286↑
	pt împărţire	DIV, IDIV	
	pt comparare	CMP, CMPXCHG, CMPXCHG8B	CMPXCHG - 80486↑, CMPXCHG8B - Pentium↑
	pt corecţia ACC	DAA, AAA, DAS, AAS, AAM, AAD	
	pt extinderea ACC	CBW, CWD, CWDE, CDQ	CWDE , CDQ– 80386↑
pe biţi	**logice**	NOT, AND, OR, XOR	
	de testare/comparare	TEST, BT,BT[S/R/C],BS[F/R],SETcc	80386↑
	de deplasare (shift)	SHL/SAL, SHR, SAR, SHLD,SHRD	SHLD,SHRD – 80386↑
	de rotire (rotate)	ROL, RCL, ROR, RCR	
de manipulare a şirurilor	**pt operaţii primitive**	MOVSB, MOVSW, MOVSD, movs CMPSB, CMPSW, CMPSD, cmps LODSB, LODSW, LODSD, lods STOSB, STOSW, STOSD, stos SCASB, SCASW, SCASD, scas	MOVSD – 80386↑ CMPSD – 80386↑ LODSD – 80386↑ STOSD – 80386↑ SCASD – 80386↑
	cu şiruri pe port	INSB, INSW, INSD, OUTSB, OUTSW, OUTSD	[IN/ OUT]SB/W – 80186↑ [IN/OUT]SD - 80386↑
	prefixe de repetare	REP,REPE/REPZ, REPNE/REPNZ	
specifice procedurilor/ întreruperilor şi de salt	**de apel şi revenire din procedură/ întrerupere**	CALL, RET INT, IRET	
	de salt (ne)condiţionat	JMP, J[condiţie]	
	pt controlul buclelor de program	JCXZ, LOOP,LOOPZ/LOOPE,LOOPNZ/LOOPNE	
pt controlul proce	**pt controlul explicit al unor indicatori (flaguri)**	CMC, CL[flag], ST[flag], flag=C,D,I	

Numarul de instructiuni suportate de un processor x86



‘15: Core i3, i5, i7 – gen V + **AVX-512** ← **Broadwell**

‘13 : Core i3, i5, i7 – gen IV + **AVX2** si **FMA3** ← **Haswell**

‘11: Core i3, i5, i7 – gen II si III + **AVX** - reg SSE **de la 128b la 256b** ← **SandyBridge**

‘08: Core i3, i5, i7 – prima gen + **SSE4 (54 instr)** ← **Nehalem**

‘06: (Core 2 Duo / Quad) + **SSSE3 (16 instr), fara HT** ← **Core**

‘04: Intel: EM64T:+atomic compare and swap + **SSE3 (13 instr) cu HT**

‘03: AMD: **Arhit pe 64b** + **inca 8 reg de128b** *modul long*

‘00: **P4 = PIII + SSE2 (144 instr.)** – DoublePrecision ← **Netburst**

‘99: **PIII = PII + SSE (70 instr.)** – **8 reg de 128b** – FP-SinglePrecision

Pentium Pro, P II ← **P6**
Superscalar 3cai (’95-’97)

Pentium, Pentium MMX ← **P5**
+ **8 reg de 64b (**+57 instr**)** (‘93-‘97)

Set extins **+**4 instr **486**
(+FPU integrat**)**
(‘89)

Set extins **386**
Arhit pe 32b: instr. pe 32 biti
+altele noi (‘85)

Set extins **286**
adresa pe **24** biti (‘82)

Instructiuni de baza
8086/8088
(1978)

+ 60 instruct.
FPU (‘80)

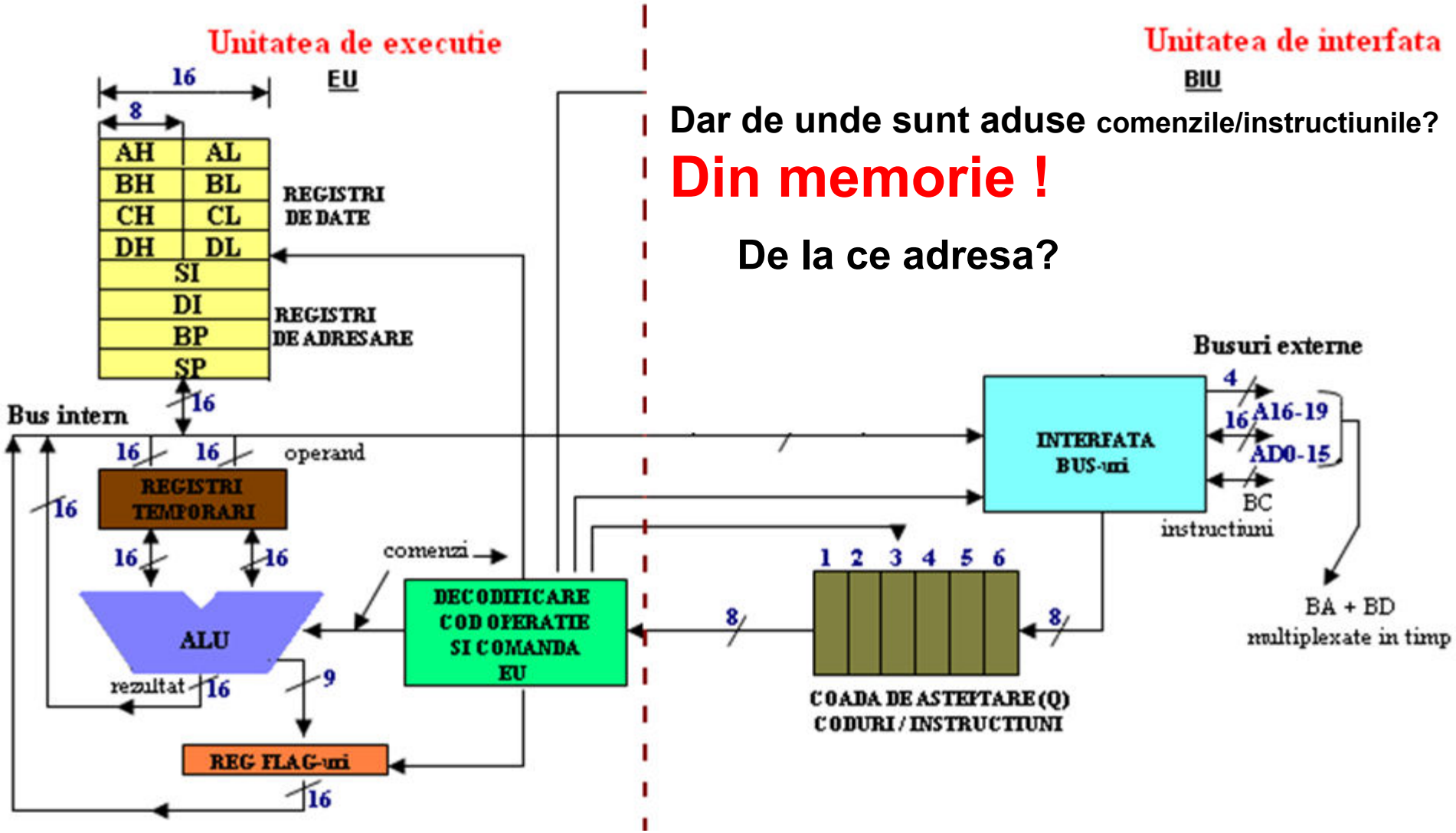
Microarhitectura

FMA = Fused Multiply-Add
AVX = Advanced Vector eXtension
SSE = Streaming SIMD Extension
MMX=MultiMedia eXtention

Arhitectura software a microprocesorului 8086

Anca APATEAN - UTCN

Revenind la schema …



Dar de unde sunt aduse comenzile/instructiunile?

Din memorie !

De la ce adresa?

Extragerea instructiunilor

Organizarea segmentelor in memorie

Exemplu:

La adresa **12002h** este stocat octetul **48h**

Adresa fizica* 12002h *se va putea scrie
ca si ***adresa logica* 1200h:0002h**,
unde segment va fi **1200h**, iar offset va fi **0002h**

Segmentul are dimensiunea maxima de 64k,
deci cat poate fi adresat cu cei 16 biti (4 cifre hexa):
de la 0000h … FFFFh

! Un segment nu poate incepe decat la o adresa multiplu de 16

- Adresele multiplu de 16 se scriu: **abcd0h**

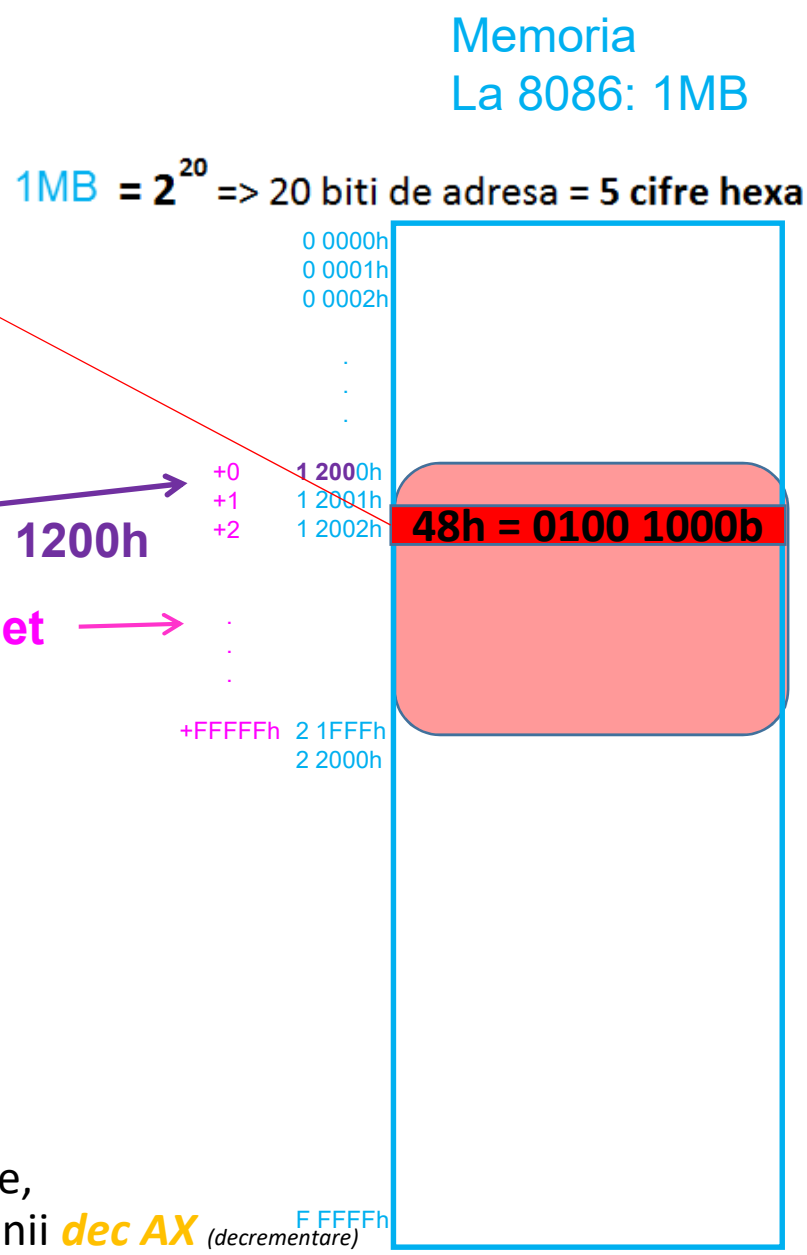
abcd0 h+
wxyz h

12000 h+
0002 h
12002 h

De unde din memorie se extrage urmatoarea instructiune?

Daca exemplul se refera la ***segmentul de cod***,
atunci octetul **48h** aflat la adresa **12002h** in memorie,
ar putea fi octetul obtinut prin codificarea instructiunii ***dec AX*** *(decrementare)*

=> CS=1200h si IP= 0002h



Extragerea instructiunilor

De unde din memorie
se extrage urmatoarea instructiune?

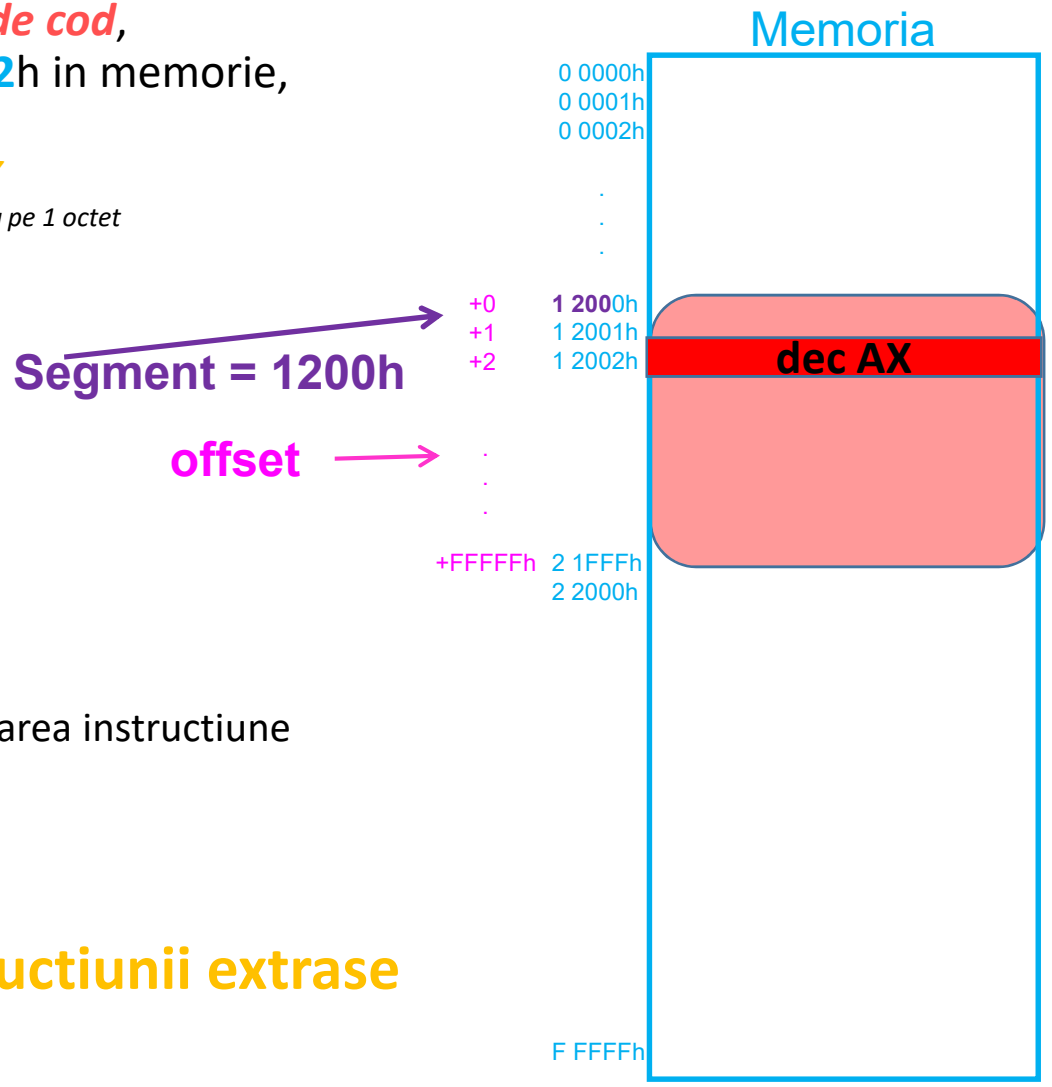
segment**0** h+ offset h

12000 h+ 0002 h

12002h

Daca exemplul se refera la ***segmentul de cod***,
atunci octetul **48h** aflat la adresa **12002**h in memorie,
ar putea fi octetul obtinut prin
codificarea instructiunii *dec AX*
– o instructiune ca oricare alta, codificata pe 1 octet

=> CS=1200h si IP= 0002h



Deci spunem ca din memorie,
de la adresa **12002h** s-a extras
(operatia de FETCH) instructiunea **48h**

Dupa extragerea acestei instructiuni,
registrul **IP va fi actualizat in mod automat**
astfel incat sa se poata apoi extrage urmatoarea instructiune

$\Rightarrow$ deci IP va fi incrementat cu 1 in acest caz

La modul general:

IP = IP + lungimea instructiunii extrase

Codificarea instructiunilor

Instrucţiunile în limbaj de asamblare de la 8086 până la Pentium Pro
au lungime VARIABILA, **între 1 şi 13 octeţi**.
Nu există o listă completă a lor, deoarece există **peste 100.000 combinaţii**.
Modul cum se obţin aceste combinaţii – e dat ca regulă = "***codificarea instrucţiunilor***".

Codificarea instrucţiunilor se realizeaza în funcţie de **modul de adresare** folosit.

Formatul unei instrucţiuni este de forma:

Pentru procesoare **8086-80286**: instrucţiuni pe 16 biţi:

"formatul instructiunilor": forma pe 16b sau pe 32b

Fig a)

Pentru procesoare **80386↑-Pentium Pro**: instrucţiuni **pe 32 biţi**:

Fig b)

Formatul instructiunii la 8086

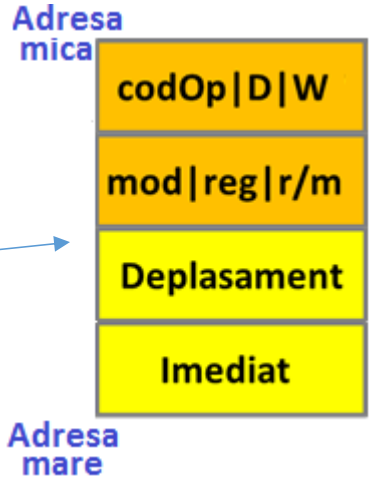
Formatul unei instrucţiuni este de forma sir de octeti

Pentru procesoare 8086-80286: instrucţiuni pe 16 biţi:

“formatul instructiunilor”: forma pe 16b sau pe 32b

codOp|D|W	mod|reg|r/m	Deplasament	Imediat
1-2octeti	0-1octeti	0-1octeti	0-2octeti

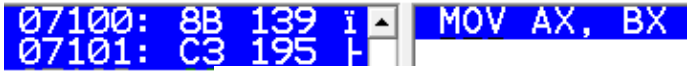
- In memorie se va vedea sub forma:



Exemplu:

instructiunea **MOV AX, BX** -> se va *codifica* sub forma **8B C3**

Asamblorul o converteste in: **8B C3**



8B C3 = **1000 10|1|1| 11|00 0|011** b

Unde **codOp** = **100010** ; cod pentru una din variantele mov
D=1 ; destinatie este un registru
W=1 ; dimensiunea destinatiei = 1 word

Mod=11 ; r/m se va referi la un registru
Reg=000 ; registrul destinatie va fi AX
r/m=011 ; registrul sursa va fi BX

Formatul instructiunii la 8086

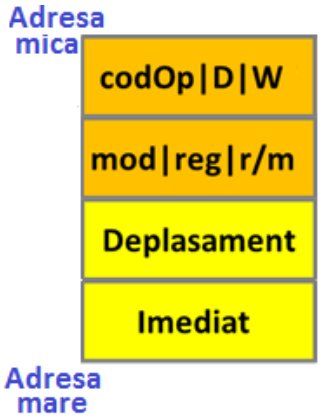
Formatul unei instrucţiuni este de forma sir de octeti

Pentru procesoare 8086-80286: instrucţiuni pe 16 biţi:

“formatul instructiunilor”: forma pe 16b sau pe 32b

codOp|D|W	mod|reg|r/m	Deplasament	Imediat
1-2octeti	0-1octeti	0-1octeti	0-2octeti

- In memorie se va vedea sub forma:



Formatul unei instrucţiuni este de forma *sir de octeti* – poate avea orice dimensiune, nu trebuie neaparat 1B (byte), 2B (word), 4B(doubleword), etc

- Exemple de instructiuni codificate pe 2 octeti, 3 octeti, 5 octeti – mai jos

! Scrieti aceste exemple in Emu si observati incrementarea registrului IP in mod automat !

Exemple:

Instructiunea **MOV AX, BX** -> se va *codifica* sub forma **8B C3** h

Fara **Deplasament**, fara **Imediat**

```
07100: 8B 139 ï  MOV AX, BX
07101: C3 195 ├
```

Instructiunea **MOV AX,1234h** -> se va codifica sub forma **B8 34 12** h

Fara **Deplasament**, cu **Imediat pe 2 octeti (1234h)**

```
07100: B8 184 ╕  MOV AX, 01234h
07101: 34 052 4
07102: 12 018 ↕
```

Instructiunea **MOV [BX+5],1234h** -> se va codifica sub forma **C7 47 05 34 12** h

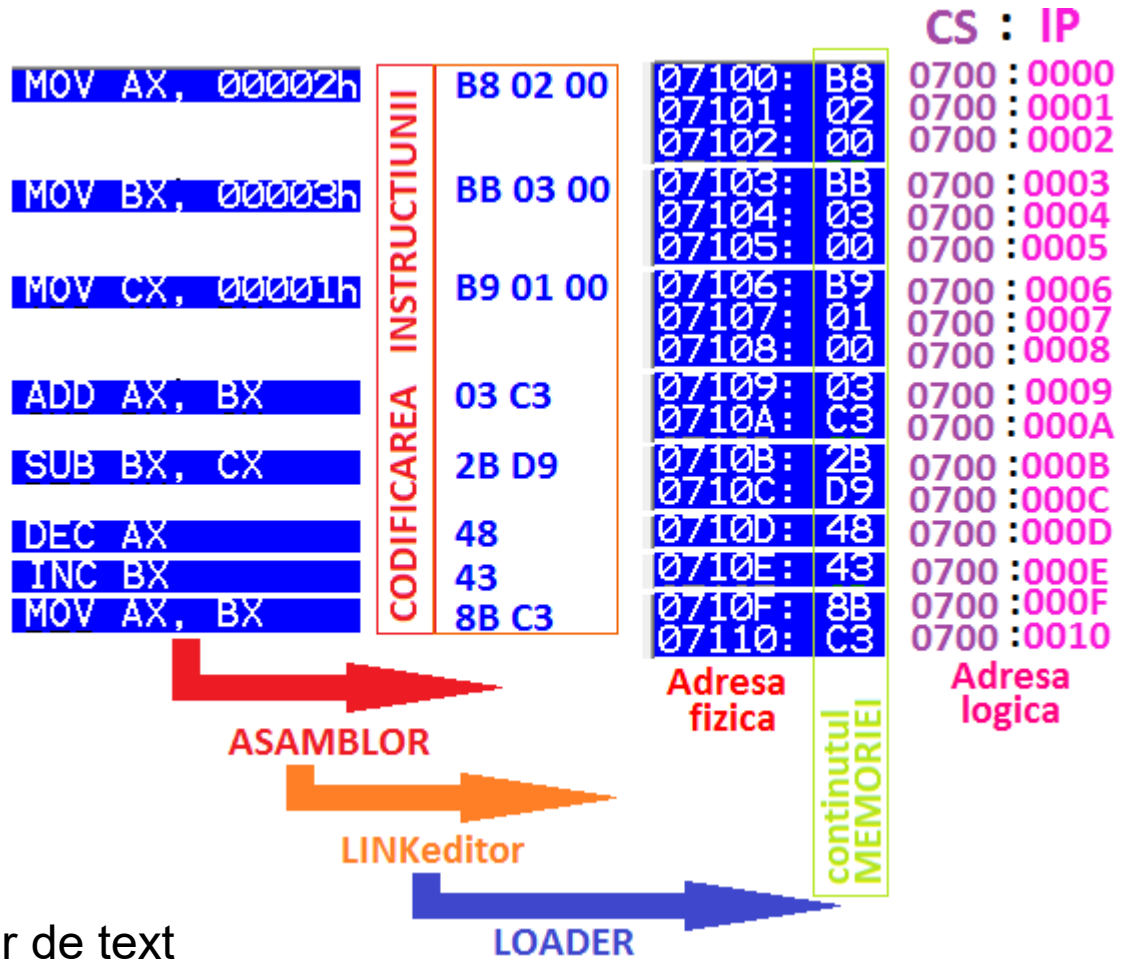
cu **Deplasament pe 1 octet (05h)**, cu **Imediat pe 2 octeti (1234h)**

```
07100: C7 199 ╟  MOV w.[BX]+05h, 01234h
07101: 47 071 G
07102: 05 005 ♣
07103: 34 052 4
07104: 12 018 ↕
```

De la cod sursa in limbaj de asamblare la cod masina

Codul sursa (limbaj asm):

```
org  100h
mov  AX,2     ;  AX:=2
mov  BX,3     ;  BX:=3
mov  CX,1     ;  CX:=1
add  AX,BX    ;  AX:=AX+BX
sub  BX,CX    ;  BX:=BX-CX
dec  AX       ;  AX:=AX-1
inc  BX       ;  BX:=BX+1
mov  AX,BX    ;  AX:=BX
```



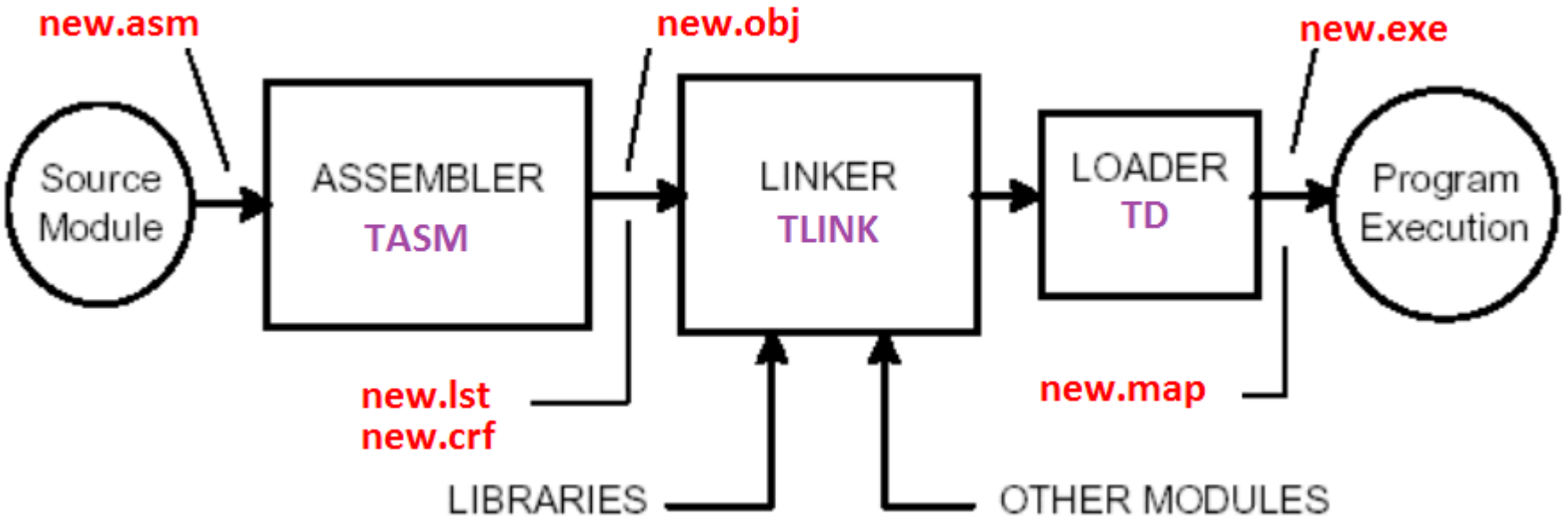
Pasi:

1. Codul sursa se scrie intr-un editor de text gen *Notepad* ++ si apoi
se salveaza fisierul cu extensia **.asm**
2. Se **asambleaza**,
linkediteaza si
se incarca in memorie
programul executabil obtinut – se pot folosi: TASM, TLINK, TD

Adresa LOGICA
segment : offset

segment0 h+
offset h
Adresa FIZICA

Ciclul complet pentru dezvoltarea unui program in limbaj de asamblare



Ce contine **fisierul sursa**, cel de tip ***asm*** ?

OBLIGATORIU un anumit SABLON !

Daca nu e scris corect, nu va functiona !

SABLONUL unui program de tip *asm*

```
ASSUME cs: code, ds:data   ; directiva ASSUME – creare legaturi
                           ; se specifica asamblorului care sunt segmentele folosite

data SEGMENT
                           ; data - segmentul de date in care se vor defini variabilele (datele)

data ENDS

code SEGMENT
                           ; code - numele segmentului de cod
                           ; in care vor aparea instructiunile

start:
  mov ax,data              ; adresa segmentului de date se copiaza in AX
  mov ds,ax                ; continutul lui AX se copiaza in DS

                           ; .....
                           ; aici se vor scrie instructiunile programului
                           ;.....

mov ax,4C00h
  int 21h                  ; pentru iesire din program
code ENDS
END start
```

in EMU:

```
org 100h
.data

      db 12h
a     db 34h
sir   db 56h,78h
b     dw 9ABCh
c     dw ?


.code

    mov al, a+4
    mov ah, [a+4]
    mov bx, offset a

ret
```

Codificarea instructiunilor (2)

Formatul unei instrucţiuni este de forma:

Pentru procesoare 8086-80286: instrucţiuni pe 16 biţi: **“formatul instructiunilor”: forma pe 16b sau pe 32b**

Fig a)

Pentru procesoare 80386↑-Pentium Pro: instrucţiuni pe 32 biţi:

Fig b)

Procesoarele 80386↑ presupun că toate instrucţiunile sunt pe 16 biţi când lucrează în mod real, iar când lucrează în mod protejat se poate selecta ori mod de instrucţiuni pe 16 biţi (Fig a)), ori pe 32 biţi (Fig b))

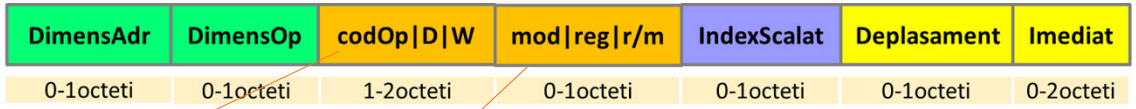
Reguli:

Dacă instrucţiunea e codificată ***pt 16b*** şi se folosesc ***regiştrii de 32 biţi*** : **apare prefixul 66h**

Dacă instrucţiunea e codificată ***pt 32b*** şi se folosesc ***regiştrii de 16 biţi*** : **apare prefixul 67h**

Dacă instrucţiunea e codificată ***pt 32b*** şi se folosesc ***regiştrii de 32 biţi*** : **nu apare prefix**

Codificarea instructiunilor (3)



7	6	5	4	3	2	1	0
mod		reg			r/m		

Octetul (Octeţii) **CodOp|D|W:**

codOp – este codul operaţiei sau mnemonicii

d – indică dacă datele se copiază *în registru* sau *din registru* (sau dacă locaţia de memorie este sursă sau destinaţie):

d=**0** din registru (memoria e destinaţie),

d=**1** în registru (memoria e sursă)

w – indică dacă operaţia se face pe octet (**w=0**) sau pe cuvânt sau dublucuvânt (**w=1**)

– prefixul DimensOp specifică exact dacă e cuvânt sau dublucuvânt prin folosirea *valorii 66h* sau *nimic*

reg	w=0	w=1	
000	AL	AX	EAX
001	CL	CX	ECX
010	DL	DX	EDX
011	BL	BX	EBX
100	AH	SP	ESP
101	CH	BP	EBP
110	DH	SI	ESI
111	BH	DI	EDI

Octetul **Mod de adresare**: este alcătuit *din 3 părţi distincte*:

mod = 00 – adresare cu memoria, fără deplasament

01 – adresare cu memoria, deplasament pe un octet

10 – adresare cu memoria, deplasament pe doi octeţi sau 4 octeţi, funcţie de dimensiunea instrucţiunii

11 – adresare registru

reg – indică registrul operand în instrucţiune

– tabelul se refera si la campul r/m cand mod=11 !

r/m – indică modul de calcul al adresei efective (reg. segm implicit)

r/m	mod de Adresare pe 16 biţi	mod de Adresare pe 32 biţi
000	DS:[BX+SI]	DS:[EAX]
001	DS:[BX+DI]	DS:[ECX]
010	SS:[BP+SI]	DS:[EDX]
011	SS:[BP+DI]	DS:[EBX]
100	DS:[SI]	Scalată
101	DS:[DI]	SS:[EBP]*
110	SS:[BP]*	DS:[ESI]
111	DS:[BX]	DS:[EDI]

Exemple codificarea instructiunilor

Exemple:

Exemplul 1. pe un PC cu S.O. pe 16 biţi apare o instrucţiune pe 2octeţi într-un program:

8BECh

Despre ce instructiune este vorba?

R: deoarece *nu apare nici 67h nici 66h* ca prim octet

=> **8Bh** va fi codOp|D|W --→ **100010 1 1 b**

codOp=**100010** (instruct ***mov reg,reg***),**D**=**1**(în reg),**W**=**1**(pe cuvânt)

=> **ECh** va fi Mod de adresare --→ **11 101 100 b**

mod=**11** (celalalt operand e tot registru),**reg**=**101**(reg BP), **r/m**=**100**(reg SP)

Astfel, instrucţiunea va fi: **mov BP,SP**

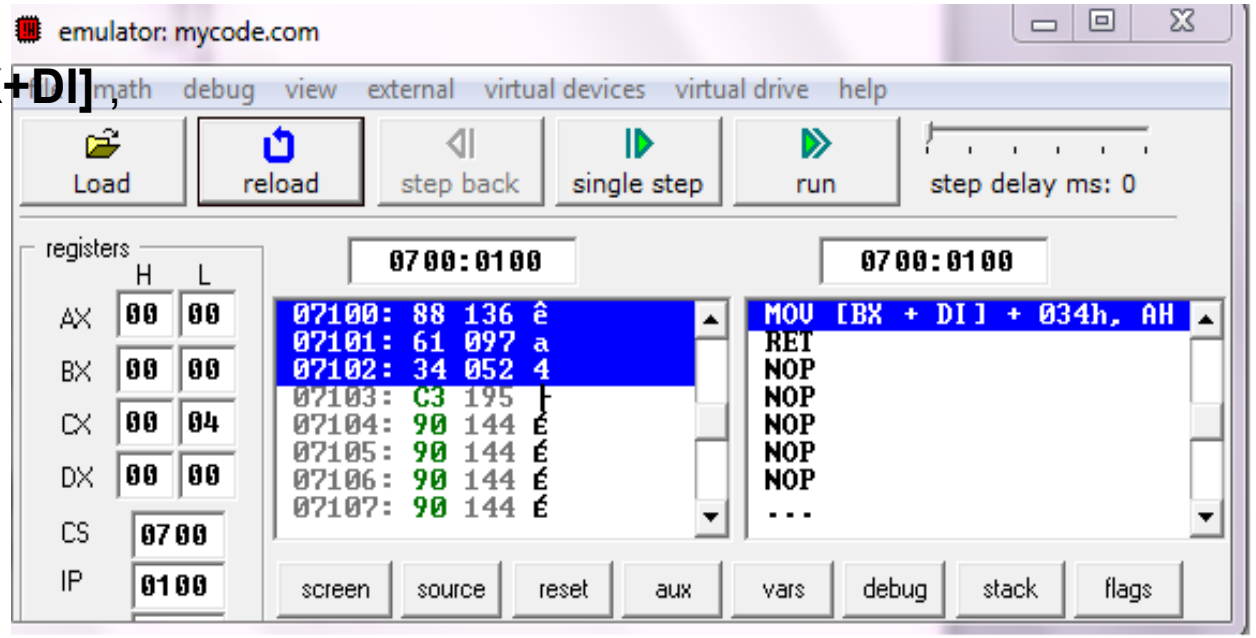
Exemple codificarea instructiunilor (2)

Exemplul 2: se da instructiunea :

mov [bx+di+34h], ah ;

codOp|D|W	mod|reg|r/m	Deplasament	Imediat
1-2octeti	0-1octeti	0-1octeti	0-2octeti

aratati cum se codifica aceasta
(verificati si cu EMU)
R : se transfera ***din reg*** => **d**=0, un octet => **w**=0 ;
se fol deplasament pe octet => **mod**=01,
se fol reg AH=> **reg**=100,
r/m=001 pt ca se fol **DS :[BX+DI]**,
va trebui trecut deplasamentul
ca octet => ***886134h***



Exemplu: mov [bx+di+34h], ah ;

8	8		6	1		3	4
100010	0	0	01	100	001	0011	0100
cod	d	w	mod	reg (AH)	r/m	3	4
"mov"					(bx+di+depl8)		

Lucrurile se complica ...

Doar CodOp a lui mov are cateva variante disponibile:

mov reg,reg ; mov reg,mem ; mov mem, reg ; mov reg,regSegm , etc ...

In functie de tipul instructiunii mov, campurile pot si ele sa difere:

r/m poate avea rol de regSegm sau ca mai devreme

=> peste **100.000 variante de instructiuni**

instructiunile scrise in limbaj de asamblare sunt transformate ***in cod masina***(asamblor)

- - cei care vor sa fie *dezvoltatori de asambloare* pot trata in detaliu acest subiect !

Numar variabil de cicluri de ceas pe instructiune

Procesor / Tipul instructiunii	Codificarea operatiei	8086	8088	80286	80386	80486	Pentium - Core2
mov reg,reg	1000 10DW oorrrmmm depl	2	2	2	2	1	1
mov mem,reg		9+AE	13+AE	3	2	1	1
mov reg,mem		10+AE	12+AE	5	4	1	1
mov mem,imed	1100 011W oo000mmm depl data	10+AE	14+AE	3	2	1	1
mov reg,imed	1011 wrrr data	4	4	3	2	1	1
mov mem,acc	1010 00DW depl	10	14	3	2	1	1
mov acc,mem		10	14	5	4	1	1
mov seg,reg	1000 11D0 oosssmmm depl	2	2	2	2	1	1
mov seg,mem		8+AE	12+AE	2	2	1	2 sau 3
mov reg,seg		2	2	2	2	1	1
mov mem,seg		9+AE	13+AE	3	2	1	1
mov reg,cr	0000 1111 001000D0 11rrrmmm	-	-	-	6	4	4
mov cr,reg		-	-	-	10	4	12-46
mov reg,dr	0000 1111 001000D1 11rrrmmm	-	-	-	22	10	11
mov dr,reg		-	-	-	22	11	11
mov reg,tr	0000 1111 001001D0 11rrrmmm	-	-	-	12	4	11
mov tr,reg		-	-	-	12	6	11

Dar de ce se face ?

Codificarea instructiunilor – se realizeaza pentru ca :

instructiunile scrise in limbaj de asamblare sunt transformate ***in cod masina*** (asamblor)

in final, orice instructiune (indiferent ca e scrisa in LLL sau HLL) se va transforma in siruri de 0 si 1

De exemplu, in LLL: **mov [bx+di+34h], ah** -> **886134h,**

adica 100010000110000100110100b

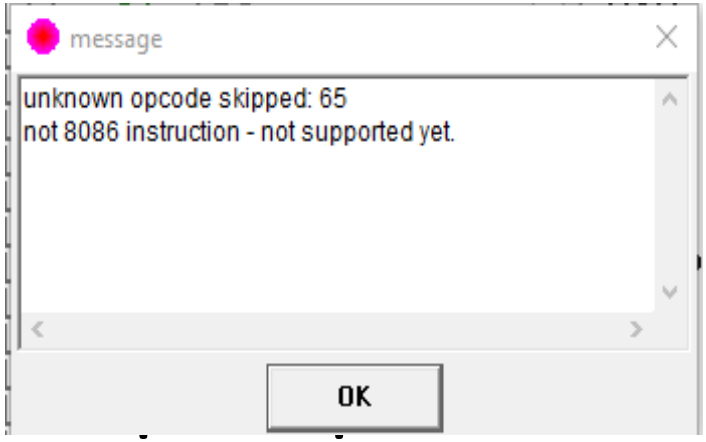
– impartita in 3 octeti -> 10001000 | 01100001 | 00110100b care sunt depusi in memorie

(de fapt tot programul va « suferi » aceleasi modificari)

=> in memorie va fi o zona in care se va gasi «programul» si CPU va fi directionat spre acea zona

- CPU va trebui sa identifice fiecare instructiune in zona de memorie, sa o aduca local (operatie numita ***FETCH***) , sa o decodifice (operatie numita ***DECODE***) – adica sa o transforme inapoi in **« mov [bx+di+34h], ah »** si abia apoi sa o execute (***EXECUTE***)

Diferentierea date vs cod



- Ideea principala la sistemele von Neumann:
  - atat instructiunile cat si datele sunt pastrate in **aceeasi memorie**
- Cum se realizeaza atunci diferenta intre date, resp instructiuni daca ambele sunt in memorie?
- De unde stie CPU ca o valoare din memorie, de exemplu **48h** este **codul** instructiunii **dec AX**, sau **un operand** (de ex. al doilea element al unui sir stocat in memorie si definit cu **sir db 34h, 48h, 86h**) ?
- Din **adresa fizica** va compune **ADRESA LOGICA**
  - -> daca ea corespunde la **CS:IP**, stie ca e **cod**
  - -> daca adresa logica va corespunde la **DS:offset**, atunci sunt **date**
- **TOTUSI ! Am vazut ca din 2 adrese logice diferite putem ajunge la aceeasi adresa fizica !**

2. Calculatorul von Neumann

Unitatea centrala de procesare

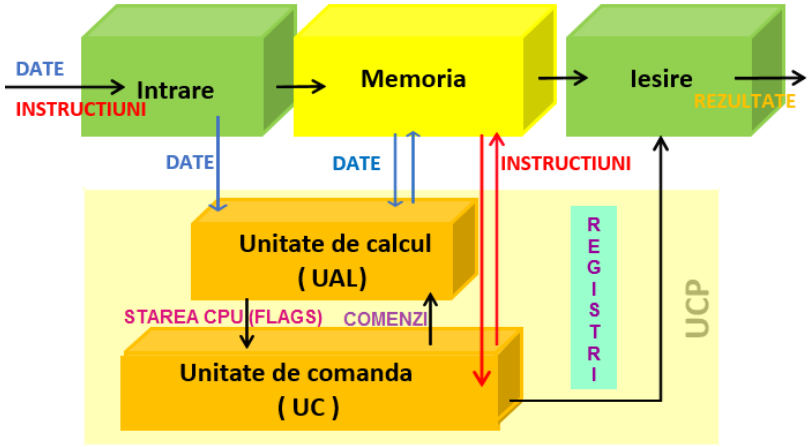
Anca APATEAN -UTCN

UC ii specifica lui **UAL** care este operatia pe care trebuie sa o execute, iar **UAL** dupa obtinerea rezultatului, il poate plasa inapoi in **registrii** sau in memoria principala.

Unitatea de comanda si control (UC) realizeaza operatia de "**fetch**" = extrage instructiunea din memorie si o decodeaza pentru a vedea ce operatie are de executat UCP

- la terminarea executiei unei instructiuni, se trece la urmatoarea (adresa instructiunii e pastrata in reg. IP)

Schema bloc simplificata a UCP



!!! Executia instructiunii implica 1-operatia de fetch, 2-executia instructiunii si 3-scrierea rezultatului !!!

= ***ciclu de executie***

Modelul "von Neumann" - bazat pe operatiile ***Fetch – decode - execute*** pt a executa programe :

1. Fetch, 2. Decode, 3. Data Transfer, 4. Execute – dar in gen sunt 4 pasi , nu 3 !

O iteratie a ciclului se desfasoara astfel:

1. UC aduce ("fetches") urmatoarea instructiune de program din memorie, folosind PC (pt a determina unde este localizata instructiunea)
2. instructiunea este decodificata intr-un limbaj pe care UAL il intelege
3. operanzii necesari executarii instructiunii sunt adusi din memorie si plasati in **registre**, in UCP
4. UAL executa instructiunea si plaseaza rezultatul in **registre** sau in memorie

Bibliografie principala

[Barr2005] Mostafa Abd-El-Barr, Hesham El-Rewini
Fundamentals of Computer Organization and Architecture

[Baruch2000] - Zoltan Baruch
– "Arhitectura calculatoarelor", Editura Todesco, 2000

[Brey1997] - Barry B. Brey
The Intel Microprocessors", 4th edition, 1997

[Henessy2007] - John Hennesy, David Patterson
– "Computer Architecture – A quantitative Approach" , 2007

[Hide2001] Randall Hide
The Art of Assembly Language

[Lupu2012] Eugen Lupu, Simina Emerich , Anca Apatean
Initiere in Limbaj de Asamblare x86. Lucrari practice, teste si probleme

[Mueller2012] Scott Mueller
– "Upgrading and Repairing PCs", 20th edition, 2012

[Null2003] - Linda Null, Julia Lobur
– "The essentials of Computer Organization and Architecture", 2003

[Patterson2009] – David Patterson, John Hennesy
– "Computer Organization and Design – the hardware/software interface", 4th edition, 2009

[Tarnoff2007] - David Tarnoff
– "Computer Organization and Design Fundamentals", editia intai revizuita, 2007
- Cursuri dl prof. E. Lupu , alte cursuri www