ACSO - Laboratories

Balan Gheorghe - UAIC

Info

Administrative:

- maxim 2 absente la laborator
- test final de laborator
- test final la curs

Mediul de lucru:

- VISUAL STUDIO (https://www.youtube.com/watch?v=ang-5_tb52w)

Linkuri utile

http://students.info.uaic.ro/~gheorghe.balan/body/intel-reference-instructions.pdf (documentatie intel)

https://sites.google.com/site/practicahardware/fii-iasi/arhitectura-calculatoarelor-si-sisteme-de-operare (by Alex Baetu)

Testare mediu de lucru

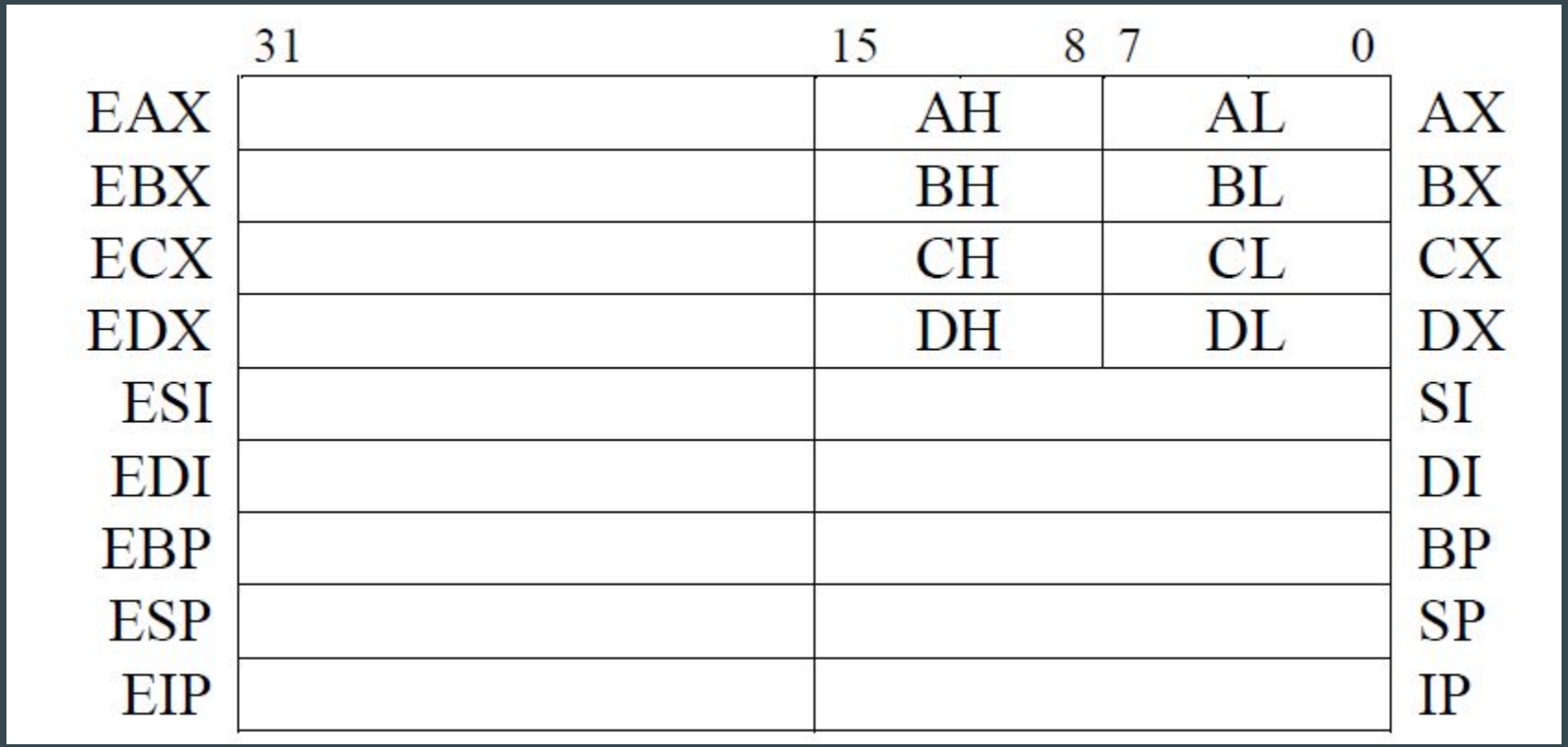
Dupa instalarea Visual Studio, urmariti pasii din acest video pentru a testa daca ati instalat corect Visual Studio:

- https://www.youtube.com/watch?v=ang-5_tb52w

Cu ce tipuri de date vom lucra?

- de dimensiune:
  - 8 biti: char / unsigned char (byte) - size 1
  - 16 biti: short / unsigned short (word) - size 2
  - 32 biti: int / unsigned int (dword) - size 4
- vom lucra in system little-endian:
  - octetii mai mari pe adresa mai mica
  - exemple:
    - word: | 15 ... 8 | 7 ... 0 |
    - dword: | 32 ... 24 | 23 ... 16 | 15 ... 8 | 7 ... 0 |
- numerele negative in ASM sunt retinute in C2

Registri - de uz general



Registri - de uz general

pe 4 bytes (32 de biti):

- EAX - putem sa-l folosim pentru calcule, dar are si anumite roluri:
  - retine valoarea de return a unei functii
  - retine ultimii 32 de biti din rezultatul unei inmultiri (mul)
  - retine catul intr-o impartire (div)
- EBX - putem sa-l folosim pentru calcule
- ECX - putem sa-l folosim pentru calcule, dar are si anumite roluri
  - are rol de counter (pentru loop-uri)
- EDX - putem sa-l folosim pentru calcule, dar are si anumite roluri
  - **trebuie facut 0 inainte unei impartiri!** - utilizat in impartiri: (div)
  - va retine primii 32 de biti din rezultatul unei inmultiri (mul)

Registri - de uz general

pe 4 bytes (32 de biti) (continuare)

- ESI - putem sa-l folosim pentru calcule, dar are si anumite roluri:
  - utilizat in operatiile pe array-uri as source
- EDI - putem sa-l folosim pentru calcule, dar are si anumite roluri:
  - utilizat in operatiile pe array-uri as destination
- EBP - **nu-l folosim in calcule**, este utilizat pentru adresarea stivei (base pointer)
- ESP - **nu-l folosim in calcule**, este utilizat pentru adresarea stivei (stack pointer)
- EIP - **nu-l folosim in calcule**, este utilizat pentru adresarea instructiunilor de executat (instruction pointer)

Registri - de uz general

- **doar** registrii pe 32 de biti pot fi folositi pentru construirea adreselor de memorie (vom discuta detaliat mai tarziu)
- AX, BX, CX, DX, SI, DI, BP, SP, IP - sunt adresarile explicite pentru ultimii 2 bytes (16 biti) din registri mentionati; orice modificare a lor schimba valorile si pentru EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP
- de asemenea, pentru registrii EAX, EBX, ECX, EDX avem posibilitatea adresarii ultimului bit, respectiv penultimului bit, orice modificare a lor schimba registrii:
  - AH, AL
  - BH, BL
  - CH, CL
  - DH, DL

Registri - de uz general - recapitulare

- **vom folosi in practica** pentru calcule doar EAX, EBX, ECX, EDX, ESI, EDI (si adresarile de 2 bytes, respectiv 1 byte aferente)
- EBP, ESP - ii vom folosi pentru lucrul cu stiva (vedem ulterior cum)

Exemplu adresare:

- sa spunem ca punem in eax: 11100011 11100011 11100011 11100011
- rezulta ca ax -ul va avea continutul: 11100011 11100011
- daca modificam ax-ul, in: 11000000 11000000, se va modifica si eax-ul
- noua valoarea a lui eax: 11100011 11100011 11000000 11000000

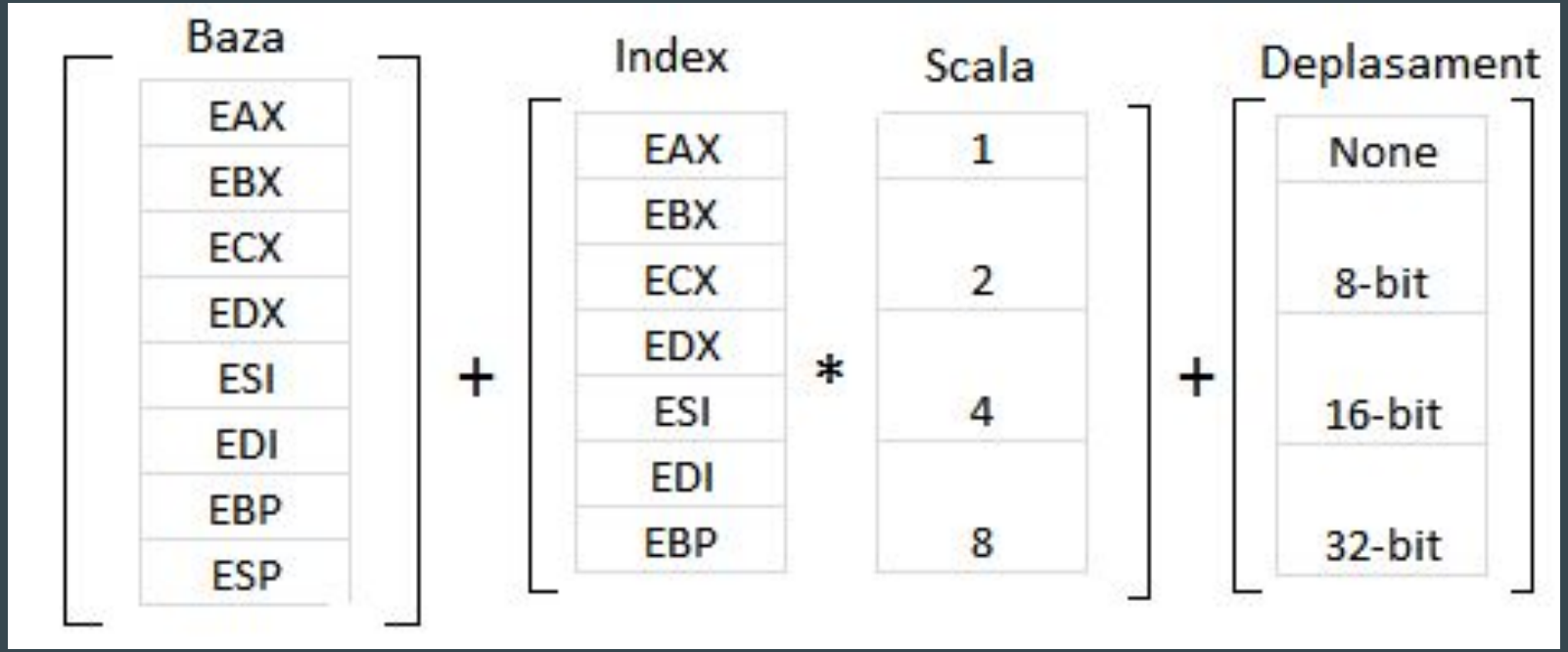
Registri - de segment

- un program poate avea zone de memorie independente, numite segmente
- un program isi poate scrie codul in segemente diferite
- adresa fiecarui segment va fi retinuta intr-un registru de 16 biti (2 bytes)
- exista 6 astfel registri de segment (CS, DS, SS, ES, FS, GS)
- **pe parcursul laboratorului nu vom folosi adresare cu segement, trebuie sa stiti ca exista**

Registri - registrul indicator de conditii (EFLAGS)

- are o dimensiune de 32 de biti, fiecare bit fiind setat cu un anumit scop
- acestia se seteaza (1) sau se reseteaza (0) daca sunt indeplinite anumite conditii dupa executia unei instructiuni (nu le putem accesa sau modifica noi direct)
- de interes sunt:
  - bitul 0 (CF) -> Carry FLAG
    - ia valoarea 1 dacă se generează transport în urma adunării/scăderii a două numere, 0 în caz contrar; poate fi folosit şi de către alte instrucţiuni, pentru indicarea altor situaţii similare
  - birul 6 (ZF) -> Zero FLAG
    - ia valoarea 1 dacă şi numai dacă rezultatul ultimei instrucţiuni este 0, daca nu zero flag este 0
  - bitul 7 (SF) -> Sign FLAG
    - preia valoarea bitul de semn obţinut de ultima instrucţiune executată
  - bitul 11 (OF) -> Overflow FLAG
    - ia valoarea 1 dacă se produce o depăşire în urma adunării/scăderii a două numere (considerate ca numere cu semn), 0 în caz contrar

Adresarea Memoriei



Adresarea memoriei

!!O ADRESA DE MEMORIE TREBUIE SCRISA INTRE PARANTEZE PATRATE!!

Exemple:

- [256] -> avem access la datele de la adresa 256 (adresa in acest caz este constanta)
- [EAX] -> avem access la datele de la adresa precizata prin valoarea lui EAX
- [EAX + 5] -> avem access la datele de la adresa precizata prin valoarea lui EAX la care adaugam constanta 5 (vom avea datele de la adresa EAX + 5)
- [EAX + EBX] -> avem access la datele de la adresa precizata prin rezultatul sumei valorii lui EAX cu EBX

Adresarea memoriei - continuare

Exemple (continuare):

- [EAX + EBX + 1] -> avem access la datele de la adresa precizata prin rezultatul sumei valorii lui EAX, EBX si 1
- [EAX+EBX*2] -> ...
- [ECX+EBX*8+5] -> ...

Adresarea memoriei - continuare

IMPORTANT

De la o adresa de memorie, putem lua un numar de biti. Pentru a specifica numarul de biti pe care-l luam, putem adauga urmatoarele cuvinte keyword:

- byte ptr [] -> va lua un singur byte de la adresa specificata intre []
- word ptr [] -> va lua 2 bytes de la adresa specificata intre []
- dword ptr [] -> va lua 4 bytes de la adresa specificata intre []

In general, operanzii ce sunt folositi in instructiuni trebuie sa aiba acelasi size (de asta folosim byte ptr [], word ptr, ...). De asemenea, nu este posibil ca doi operanzi sa fie adrese de memorie.

Adresarea memoriei - continuare

vector de char: ("abcde") | s[ ] = a b c d e | s[0] = a -> (s + 0) -> pointeaza la abcde | s[1] = b -> (s + 1) -> pointeaza la bcde | s[2] = c -> (s + 2) -> pointeaza la cde | s -> pointeza la tot sirul de caractere (abcde)

[s+0] (a)bcde | [s+1] (b)cde

[eax] (echivalent [s], daca am facut un mov eax, s)

pentru tipul variabilei (specific):

- byte ptr [s+0] "a"
- daca ar fi fost de tipul short: word ptr [s+0] "ab"
- dar ar fi fost de tipul int: dword ptr [s+0] "abcd"
- dar ar fi fost de tipul int: dword ptr [s+1] "bcde"

Adresarea memoriei - continuare

Exemplu:

- intr-un al, ah, bl, bh, cl, ch, dl, dh putem pune doar ceva cu size 1 (asta inseamna ca trebuie sa folosim byte ptr [...])
- intr-un ax, bx, cx, dx, si, di, bp, sp, ip putem pune doar ceva cu size 2 (asta inseamna ca trebuie sa folosim word ptr [...])
- intr-un eax, ebx, ecx, edx, esi, edi, ebp, esp, eip putem pune doar ceva cu size 4 (asta inseamna ca trebuie sa folosim dword ptr [...])

FIECARE INSTRUCTIUNE ARE PRECIZARI LEGATE DE TIPUL SI DIMENSIUNEA OPERANZILOR (detaliat gasiti pe linkul de documentatie intel dat la inceput)

LABORATOR - INCEPUT

http://students.info.uaic.ro/~gheorghe.balan/body/exemple/initial_asm.cpp

Luati codul de la aceasta adresa, si asigurati-va ca poate fi executat in environmentul vostru!

Operanzii instructiunilor trebuie sa aiba acelasi size!! (cu exceptia unor instructiuni, pe care le vom specifica EXPLICIT atunci cand va fi cazul)

LABORATOR - INSTRUCTIUNI - MOV

Keyword: mov

Operanzi: dest, sursa

Rezultat: dest <- sursa

Tipul operanzilor: registri, memorie (nu pot fi simultan adrese de memorie), variabile (sunt intelese in asm ca adrese de memorie, deci nu pot fi simultan 2 variabile)

Exemplu de rulat si de inteles:

http://students.info.uaic.ro/~gheorghe.balan/body/exemple/mov.cpp

Instructiuni Aritmetice

LABORATOR - INSTRUCTIUNI - ADD

Keyword: add

Operanzi: operand1, operand 2

Rezultat: operand1 <- operand1 + operand2

Tipul operanzilor: registri, memorie (nu pot fi simultan adrese de memorie), variabile (sunt intelese in asm ca adrese de memorie, deci nu pot fi simultan 2 variabile)

Exemplu de rulat si de inteles:

http://students.info.uaic.ro/~gheorghe.balan/body/exemple/add.cpp

LABORATOR - INSTRUCTIUNI - SUB

Keyword: sub

Operanzi: operand1, operand 2

Rezultat: operand1 <- operand1 - operand2

Tipul operanzilor: registri, memorie (nu pot fi simultan adrese de memorie), variabile (sunt intelese in asm ca adrese de memorie, deci nu pot fi simultan 2 variabile)

Exemplu de rulat si de inteles:

http://students.info.uaic.ro/~gheorghe.balan/body/exemple/sub.cpp

LABORATOR - INSTRUCTIUNI - INC

Keyword: inc

Operanzi: operand

Rezultat: operand <- operand + 1

Tipul operandului: registri, memorie, variabile

Exemplu de rulat si de inteles:

http://students.info.uaic.ro/~gheorghe.balan/body/exemple/inc_and_dec.cpp

LABORATOR - INSTRUCTIUNI - DEC

Keyword: dec

Operanzi: operand

Rezultat: operand <- operand - 1

Tipul operandului: registri, memorie, variabile

Exemplu de rulat si de inteles:

http://students.info.uaic.ro/~gheorghe.balan/body/exemple/inc_and_dec.cpp

LABORATOR - INSTRUCTIUNI - MUL (inmultirea)

Keyword: mul / imul (e similar cu mul, doar ca operatia se realizeaza cu semn)

Operanzi: operand

Tipul operandului: registri, memorie, variabile

Inmultirea are un operand implicit, ce poate fi al, ax, eax in functie de dimensiunea operandului dat ca parametru (explicit). **De asemenea, rezultatul se pune si in dx (respectiv edx) daca numarul rezultat din inmultire nu poate fi scris pe 2 bytes, respectiv pe 4 bytes. (datele salvate anterior in edx se pierd)**

dimensiune operand explicit	operand implicit	destinaţie rezultat
1	al	ax
2	ax	(dx,ax)
4	eax	(edx,eax)

LABORATOR - INSTRUCTIUNI - MUL (inmultirea)

Cu alte cuvinte, daca dimensiunea operandului dat e 1 (1 byte):

- ax <- operand * al

Daca dimensiunea operandului dat e 2 (2 bytes):

- (dx, ax) <- operand * ax (dx setat daca rezultatul nu poate fi scris pe 2 bytes)

Daca dimensiunea operandului dat e 4 (4 bytes):

- (edx, eax) <- operand * eax (edx setat daca rezultatul nu poate fi scris pe 4 bytes)

http://students.info.uaic.ro/~gheorghe.balan/body/exemple/mul.cpp

LABORATOR - INSTRUCTIUNI - DIV (impartirea)

Keyword: div / idiv (e similar cu div, doar ca operatia se realizeaza cu semn)

Operanzi: operand

Tipul operandului: registri, memorie, variabile

Impartirea are un operand implicit (numit deimpartit), ce poate fi ax, (dx, ax), (edx, eax) in functie de dimensiunea operandului dat ca parametru (explicit). De asemenea, in functie de dimensiunea operandului explicit, catul si restul se afla in: [al, ah] (size 1), [ax, dx] (size 2), [eax, edx] (size 4)

dimensiune împărţitor	deîmpărţit	cît	rest
1	ax	al	ah
2	(dx,ax)	ax	dx
4	(edx,eax)	eax	edx

LABORATOR - INSTRUCTIUNI - DIV (impartirea)

Cu alte cuvinte, daca dimensiunea operandului dat e 1 (1 byte):

- ax : <operand_dat> = al rest ah (al catul, ah restul)

Daca dimensiunea operandului dat e 2 (2 bytes):

- (dx, ax) : <operand_dat> = ax rest dx (ax catul, dx restul)

Daca dimensiunea operandului dat e 4 (4 bytes):

- (edx, eax) : <operand_dat> = eax rest edx (in eax catul, edx restul)

De aceea, inainte de a face o impartire, daca nu ne intereseaza valori mari trebuie sa setam DX / EDX = 0 (altfel acesta ar avea o valoare random, iar impartirea nu ar putea fi realizata)

http://students.info.uaic.ro/~gheorghe.balan/body/exemple/div.cpp

Exercitii

1. Calculati (in ASM) si afisati (in limbaj standard) suma, diferenta, produsul numerelor:
   a. 20 si 56
   b. 16 si -4
   c. 5 si 10
2. Calculati (in ASM) si afisati (in limbaj standard) catul si restul pentru impartirea:
   a. 20 la 4
   b. 21 la 5
   c. 12392 la 1223
3. Calculati (in ASM) si afisati (in limbaj standard) suma numerelor de la 1 la 20 (utilizand formula: (n)(n+1)/2)

Rezolvari: http://students.info.uaic.ro/~gheorghe.balan/body/exemple/tema1.cpp

Instructiuni pe biti

Laborator - Instructiuni pe biti

- **not** <operand> -> neaga bitii lui operand
- **and** <operand1>, <operand2> -> operand1 <- operand1 & operand2
- **or** <operand1>, <operand2> -> operand1 <- operand1 | operand2
- **xor** <operand1>, <operand2> -> operand1 <- operand1 ^ operand2
- **test** <operand1>, <operand2> -> face & intre operand1 si operand2, dar nu modifica operanzii ci seteaza doar zero flag (ZF -> vezi la inceputul prezentarii ce insemna)
- **shl** <operand1>, <numar/registru_cl> <- muta bitii spre stanga cu numar pozitii si completeaza cu 0 pozitiile ramase libere dupa mutare (aka inmultire cu puteri ale lui 2)
- **shr** <operand1>, <numar/registru_cl> <- muta bitii spre dreapta cu numar pozitii si completeaza cu 0 pozitiile ramase libere dupa mutare (aka impartire cu puteri ale lui 2)
- **sar** <operand1>, <numar/registru_cl> <- muta bitii spre dreapta cu numar pozitii pastrand bitul de semn neschimbat

Laborator - Instructiuni pe biti

Tipul operandului: registri, memorie, variabile

Rezultat: sunt executate instructiunile specifice pe biti

http://students.info.uaic.ro/~gheorghe.balan/body/exemple/instructiuni_pe_biti.cpp

Exercitiu:

1. **Fiind date doua numere a si b citite de la tastatura, calculati in asm si afisati pe ecran rezultatul urmatoarelor operatii binare: not a, not b, a & b, a | b, a ^ b, a ^ 1, b ^ 0, a * 2^5, b / 2^2**

Rezolvare: http://students.info.uaic.ro/~gheorghe.balan/body/exemple/tema2.cpp

Instructiuni de salt

Instructiuni de salt

- orice structură de control (testare - ***if*** sau buclă - **for, while**) se poate implementa doar prin salturi
- pot fi doua tipuri de instructiuni de salt:
  - conditionate
  - neconditionate
- aceste instructiuni de start le vom utiliza dupa ca sa putem scrie instructiuni de control:
  - if
  - for, while
  - do while

Instructiuni de salt - Salt neconditionat / LABEL

Keyword: jmp <nume_label>

Rezultat: se continua (sare la) executia de la label (se vor executa instructiunile care incep de la acel label, indiferent de unde sunt din cod)

Definirea unui label: intr-o anumita portiune de cod, un label se defineste astfel:

<nume_label>:

exemplu: _de_aici_incepe_adunarea_a_doua_numere: //!in momentul in care realizez un salt, pot sari la orice label din program!

Nu uitati sa puneti ":" dupa numele labelului! Nu folositi mai multe labels cu acelasi NUME!!

http://students.info.uaic.ro/~gheorghe.balan/body/exemple/jump.cpp

Instructiuni de salt - Salturi conditionate

- salturi conditionate: salturi pe care se verifica o anumita conditie
- daca conditia este evaluata **ADEVARAT,** se sare la labelul dat
- daca conditia este evaluata **FALS,** se continua executia fara a se sari (se ignora instructiunea de salt)
- exista mai multe tipuri de instructiuni de salt:
  - instructiuni ce verifica valori din EFLAGS (registru indicatori de conditii, vezi slide 12)
  - instructiuni ce verifica doi operanzi:
    - cu semn
    - fara semn

NU FOLOSITI IN COD LABELURI CU ACELASI NUME PENTRU ZONE DE COD DIFERITE!!!

Instructiuni de salt - Salturi conditionale - salturi EFLAGS

- si aici, exista 2 tipuri de salturi:
  - salturi care verifica ca flagul sa fie setat (“sari la <label> daca flagul are valoarea 1)
  - salturi care verifica ca flagul sa nu fie setat (“sari la <label> daca flagul are valoarea 0)

http://students.info.uaic.ro/~gheorghe.balan/body/exemple/salturi_conditionale_eflags.cpp

indicator testat	salt pe valoarea 1	salt pe valoarea 0
Carry	jc	jnc
Overflow	jo	jno
Zero	jz	jnz
Sign	js	jns

Instructiuni - Salturi conditionale - instructiunea CMP

- doar salturile EFLAGS nu sunt foarte utili in realitate... avem nevoie si de instructiuni mai complexe
- de acea a fost introdusa si o instructiune de comparare (CMP)
- **Keyword**: CMP
- **Operanzi**: operand1, operand2
- **Tipul operanzilor:** registri, memorie, variabile
- **Ce face:**
  - se simuleaza operand1-operand2 si se analizeaza toti indicatorii din EFLAGS, putand decide **relatia intre cei 2 operanzi**
  - **pentru a testa relatia, au fost introduse mai multe instructiuni de salt (detalii in slide-urile ce urmeaza)**

Instructiuni - Salturi conditionale - fara a lua semnul operanzilor in calcul

- Daca **NU** vrem sa luam in calcul semnul operanzilor, putem folosi instructiunile urmatoare de salt:

relaţie	instrucţiune salt
op1 < op2	jb
op1 <= op2	jbe
op1 > op2	ja
op1 >= op2	jae
op1 == op2	je
op1 != op2	jne

Instructiuni - Salturi conditionale - luam semnul operanzilor in calcul

- Pentru a lua semnul operanzilor in calcul putem folosi urmatoarele instructiuni:

relaţie	instrucţiune salt
op1 < op2	jl
op1 <= op2	jle
op1 > op2	jg
op1 >= op2	jge
op1 == op2	je
op1 != op2	jne

Instructiuni - Salturi conditionale - exemple

Exemple:

http://students.info.uaic.ro/~gheorghe.balan/body/exemple/salturi_conditionale.cpp

In continuare vom vedea cum putem utiliza toate aceste componente pentru a scrie structuri de control (ex: if, while, do-while, for)

Structuri de control

Instructiuni - Structuri de control - If

Explicatii pe cod: http://students.info.uaic.ro/~gheorghe.balan/body/exemple/if.cpp

Instructiuni - Structuri de control - While

Explicatii pe cod: http://students.info.uaic.ro/~gheorghe.balan/body/exemple/while.cpp

Instructiuni - Structuri de control - Do While

Explicatii pe cod:
http://students.info.uaic.ro/~gheorghe.balan/body/exemple/dowhile.cpp

Instructiuni - Structuri de control - For

Explicatii pe cod: http://students.info.uaic.ro/~gheorghe.balan/body/exemple/for.cpp

Exercitii

1. Calculati suma numerelor de la 1 la n, utilizand o structura de control;
2. Calculati suma numerelor: 1, 1+k, 1+2k, ... 1 + nk (unde n si k sunt dati ca parametru), utilizand o structura de control;
3. Calculati maximul si minimul a doua numere;
4. Calculati cmmdc-ul a doua numere;
5. Verificati daca un numar dat este un numar prim;
6. Numarati cati divizori proprii are un numar;

Rezolvare tema 3:
http://students.info.uaic.ro/~gheorghe.balan/body/exemple/tema3.cpp

Lucrul cu stiva

Lucrul cu stiva

- a venit momentul sa invatam sa lucram cu stiva (ultimul element adaugat, este primul element ce poate fi accesat - conceptul de LIFO)
- ne vom folosi de cei doi registri dedicati pentru acest lucru: esp (stack pointer - varful stivei), ebp (base pointer - baza stivei) si instructiunile aferente: **push**, **pop**
- de asemnea, pentru a apela o functie ne vom folosi de instructiunea: **call**
- daca ma duc pe stiva in sus +, in jos -
- vom vorbi in detaliu despre toate aceste aspecte in slideurile urmatoare

Lucrul cu stiva - Instructiunea push

Keyword: push

Operanzi: sursa

Rezultat: se urca pe stiva operandul (pentru instructiunea **push eax**, in realitate, se realizeaza urmatoarea secventa de cod: *sub esp, 4*; *mov[esp], eax*;) (pentru ca size-ul lui eax este 4)

Tipul operanzilor: registri, memorie, variabile

ATENTIE: STIVA POATE LUCRA DOAR CU OPERANZI DE SIZE 2 SAU DE SIZE 4

!!!SE RECOMANDA UTILIZAREA DOAR OPERANZILOR DE SIZE 4 !!!

Lucrul cu stiva - Instructiunea pop

Keyword: pop

Operanzi: destinatie

Rezultat: se **SCOATE** de pe stiva o valoare de size-ul operandului si o stocheaza in operand (in realitate, pentru instructiunea **pop eax**, se realizeaza urmatoarea secventa de cod: *mov eax, dword ptr [esp]*; *add esp, 4*;) (pentru ca size-ul lui eax este 4)

Tipul operanzilor: registri, memorie, variabile

ATENTIE: STIVA POATE LUCRA DOAR CU OPERANZI DE SIZE 2 SAU DE SIZE 4

!!!SE RECOMANDA UTILIZAREA DOAR OPERANZILOR DE SIZE 4 !!!

Lucrul cu stiva - la ce ne ajuta?

La ce ne poate ajuta sa stim sa lucram cu stiva?

- putem crea variabile locale (cu alte cuvinte, sa salvam anumite valori obtinute in registri ca sa putem reutiliza acei registri)
- de exemplu, daca am prea putini registri la dispozitie, si am nevoie de un anumit registru in care am deja o valoare precalculata: de exemplu, eax; cum pot proceda:
  - push eax (salvez valoarea lui eax pe stiva)
  - ... //modific eax-ul si fac orice alt calcul vreau (cu alte cuvinte, acum pot folosi registrul eax asa cum vreau eu)
  - pop eax (restabilesc in eax, valoarea pe care o avea inainte)
- nu uitati: pentru fiecare push executat, trebuie sa aveti si un pop aferent (sau sa curatati stiva) si vice-versa!!! (altfel veti primi eroare de stiva corupta si programul va crapa)

Lucrul cu stiva - alte considerente si exemple

- putem lucra cu stiva si direct cu ajutorul esp-ului / ebp-ului. acest lucru nu este recomandat, dar se poate realiza in special pentru alocarea de spatiu pentru stiva pentru variabile locale sau pentru curatarea stivei dupa un apel de functie (vom vedea imediat cum poate fi folosit)
- valorile de push, se reiau in ordine inversa prin intermediul pop. de exemplu, daca fac: **push eax, push ebx, push ecx**, pentru a pastra **neschimbate** valorile initiale ale registrilor trebuie sa fac pop in ordine inversa: **pop ecx, pop ebx, pop eax**
- EXEMPLE:
  - http://students.info.uaic.ro/~gheorghe.balan/body/exemple/exemple_push_pop.cpp

Apelul de functie

Apelurile de functie - generalitati si keyword (instructiune)

- un apel de functie este de fapt o instructiune de salt pentru a executa instructiunile aflate la o alta adresa. spre deosebire de un salt normal, la sfarsitul executiei instructiunilor, se revine de unde s-a facut apelul si se continua cu instructiunea urmatoare
- KEYWORD: **call <adresa> (in Visual Studio vom folosi call <nume_functie>)**
- (informativ) revenirea dintr-o functie se face utilizand instructiunea **ret** (in laboratoarele pe care le vom face nu ne vom ocupa direct de acest aspect, lasand compilatorul sa rezolve returnul)

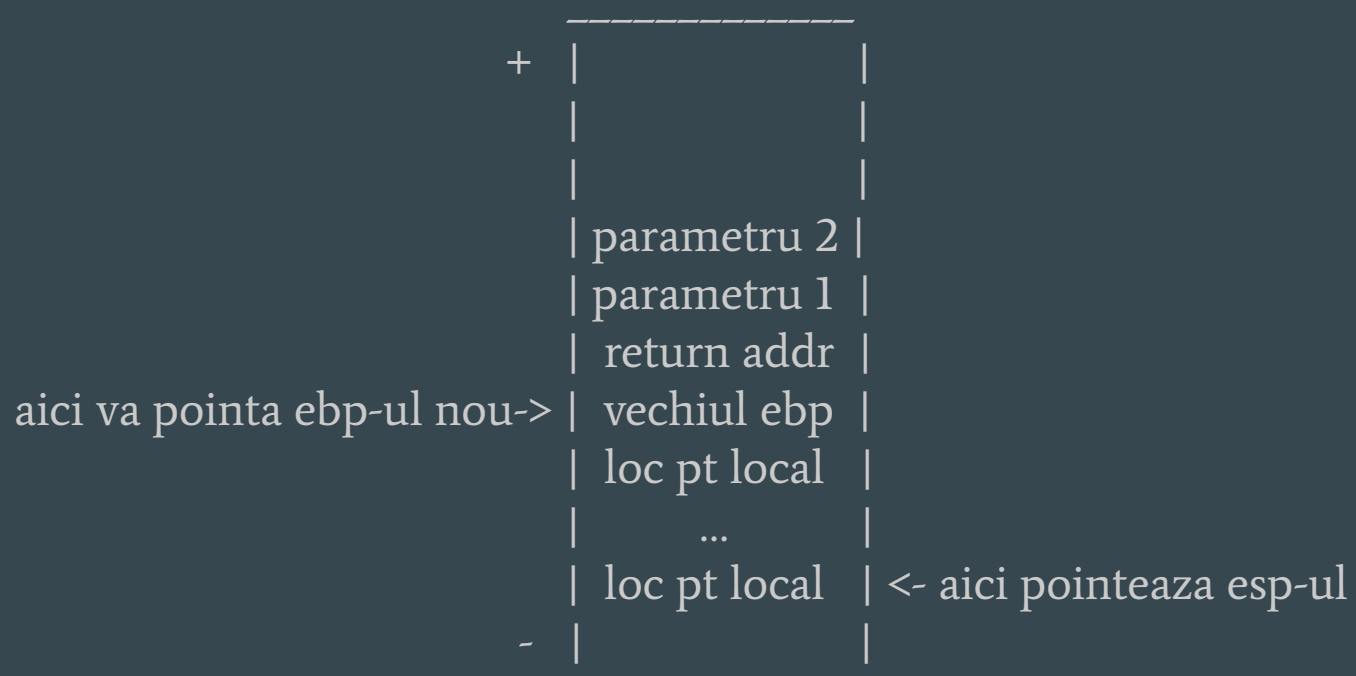
Apelurile de functii - cum apelam o functie

- parametrii cu care apelam o functie sunt variabile locale, asadar, inainte de a apela o functie trebuie sa-i adaugam pe stiva.
  - q: cum adaugam pe stiva?
  - a: utilizand instructiunea push
- dupa ce se termina apelul functiei trebuie sa curatam stiva. cum nu ne intereseaza valorile pe care le-am urcat, vom curata stiva modificand valoarea lui esp (add esp, <suma_size_tip_parametri_urcati_pe_stiva>)
- un alt aspect important de **retinut**: rezultatul unei functii se **returneaza** in registrii edx,eax, by default, in functie de dimensiunea valorii pe care vrem sa o returnam, astfel:
  - size 1 - al
  - size 2 - ax
  - size 4 - eax
  - size 8 - edx si eax
- cu alte cuvinte, trebuie sa punem valoarea pe care vrem sa o returnam fie in eax, fie in edx si eax

Apelurile de functii - ce se intampla cand apelam o functie

- cand se intalneste instructiunea call, se pun in mod automat pe stiva urmatoarele informatii:
  - adresa de return (size 4)
  - vechiul ebp (size 4)
- (informativ) dupa salvarea vechiului ebp, ebp primeste valoarea esp-ului, iar esp-ul este decrementat pentru a se face loc pe stiva variabilelor locale utilizate in acea functie (asta se face de catre compilator, doar sa stiti informativ)
- parametrii dati functiei apelate se vor gasi la ebp + 8, ebp + 12, ebp + 16, samd...
- cum am access la acele adrese? (vezi slide 14 pentru exemplu general), o sa fie ceva de genul: dword ptr [ebp + 8], dword ptr [ebp + 12], ...

Apelul de functie - cum arata stiva la un apel de functie



Apelul de functie - cum arata stiva la un apel de functie

- ce se afla la ebp + x:
  - la ebp + 4 - se afla adresa de return
  - la ebp + 8 - se afla primul parametru dat functiei
  - la ebp + 12 - se afla al doilea parametru dat functiei
  - la ebp + 16 - se afla al treilea parametru dat functiei
  - .... samd
- pe noi ne va interesa doar ebp+8, ebp + 12, epb+16, ...

Apelul de functie - considerente finale si exemple

Ca sa concluzionam, cand vrem sa apelam o functie:

- urcam pe stiva in ordine inversa parametrii (utilizand **push**)
- facem apelul de functie (utilizam **call)**
  - in functie gasim parametrii dati la ebp + 8, ebp + 12, ebp + 16, samd
- curatam stiva (utilizam **add esp, <size-ul parametrilor urcati cu push>**) in general acel size va fi multiplu de 4
- in eax / edx am valoarea returnata (vom utliza doar eax)
- **later edit: inainte de a apela o functie registrii pe care ii folosim in functia din care apelam trebuie sa-i salvam pe stiva utilizand push (dupa apelul functiei si dupa ce luam rezultatul, ii luam de pe stiva cu pop)**

Exemple de rulat: http://students.info.uaic.ro/~gheorghe.balan/body/exemple/apeluri_de_functie.cpp

Exercitii

1. Implementati exercitiile de la slide-ul 48 in functii separate, dupa care scrieti apelul lor tot in ASM
2. Fiind date 3 variabile (a, b, c) reprezentand lungimi de segemente, construiti o functie care returneaza 1 daca acestea pot alcatui un triunghi, 0 altfel (hint: trebuie verificat daca suma a oricaror doua dintre laturi este mai mare decat cea de-a treia latura)
3. (*) Impementati fibonacci recursiv (ex: la apelul fibonacci(n) se va returna al n-lea termen din sirul fibonacci)

Rezolvare tema 4: http://students.info.uaic.ro/~gheorghe.balan/body/exemple/tema4.cpp

Tablouri

Tablouri unidimensionale

- in realitate, un tablou unidimensional este o zona continua de memorie. ca sa putem accesa elementele unui vector, trebuie sa calculam adresa acelui element
- pentru un vector avem asa:
  - <tip_elemente_vector> <nume_vector>[<size_vector>]
  - ca sa accesam un element de pe o anumita pozitie, k, putem scrie asa: <nume_vector> + sizeof(<tip_elemente_vector>) * k
    - reamintim: sizeof(char/unsigned char) = 1, sizeof(short/unsigned short) = 2, sizeof(int / unsigned int) = 4
- exemple:
  - int v[5]; pentru a accesa elementul 3 din v, putem scrie v + sizeof(int) * 3
  - char v[3]; pentru a accesa elementul 2 din v, putem scrie v + 1 * 2
  - short v[10]; pentru a accesa elementul 5 din v, putem scrie v + 2 * 5

Tablouri unidimensionale - cum facem in ASM?

- avem nevoie de 2 notiuni pe care trebuie sa le stim inainte de a lucra in asm cu tablouri:
  - cum incarcam adresa unui tablou (la modul general):
    - folosim instructiunea LEA <registru>, <nume_vector>
    - dupa acea, putem folosi acel registru in lucru cu memoria (vedem imediat cum)
  - sa stim sa calculam adresele asa cum am aratat mai sus si sa precizam inainte de a face o operatie cati biti luam de la acea adresa:
    - dword ptr [adresa] - pentru int
    - word ptr [adresa] - pentru short
    - byte ptr [adresa] - pentru char
  - remindere pentru adresarea memoriei: slide 13
- Exemple de rulat si de inteles: http://students.info.uaic.ro/~gheorghe.balan/body/exemple/vector.cpp

Tablouri unidimensionale - ce urmeaza?

Ce vom vedea data viitoare?

- cum sa lucram cu siruri de caractere? in fapt un caracter este un numar (reprezentarea in codul ascii: http://www.asciitable.com)
- cum lucram cu matrice
- cum lucram cu structuri
- dar pana atunci, va uram

VACANTA PLACUTA SI SARBATORI FERICITE!!


HAPPY HOLIDAYS
Merry
CHRISTMAS
VectorStock®
VectorStock.com/11206364

Tablouri unidimensionale - siruri de caractere

- vom privi sirurile de caractere ca un vector de dimensiune 1 (pentru adresare vom folosi byte ptr [adresa])
- pentru fiecare caracter vom folosi codificarea lui in codul ASCII (http://www.asciitable.com)
- exemplu:
  - `a` - 97
  - `A` - 65
- exemplu de rulat si de inteles:
  - http://students.info.uaic.ro/~gheorghe.balan/body/exemple/siruri_de_caractere.cpp

Tablouri bidimensionale - Matrice

Cand vorbim de matrici, trebuie sa luam in calcul modul in care acestea sunt alocate (avand din modul de alocare, modul de accesare a acestora):

- alocate static (t a[n][m]):
  - t - tip de date (char - size 1, short - size 2, int - size 4)
  - n - numarul de linii, m - numarul de coloane
  - adresa elementului a[i][j] = * (a + i*m * sizeof(t) + j * sizeof(t)) = * (a + (i*m+j) * sizeof(t))
- alocate dinamic (exemplu: int **m): in acest caz trebuie sa obtinem adresa de inceput a liniei, iar dupa putem obtine fiecare element
- exemple de rulat si de inteles:
  - http://students.info.uaic.ro/~gheorghe.balan/body/exemple/matrice.cpp
- sursa complementara de informatii (by Silviu Vitel):
  - http://students.info.uaic.ro/~silviu.vitel/labs/lab11.pdf

Tablouri - considerente finale

- nu uitati de byte ptr, word ptr, dword ptr (in functie de tipul de date al tabloului cu care lucrati)
- nu uitati sa folositi registri corespunzatori:
  - pentru dword - eax, ebx, ecx, edx, esi, edi
  - pentru word - ax, bx, cx, dx, si, di
  - pentru byte - ah, al, bh, bl, ch, cl, dh, dl
  - NU UITATI CA MODIFICAND un word / byte influentam tot registrul
- in adresare *4, *2, *1 sunt date de tipul de date al tabloului

Exercitii

1. Se citeste un cuvant c cu cel mult 20 de litere. Sa se elimine din cuvantul c toate aparitiile primei litere.
    - ex: ana are mere -> n re mere
2. Se citeste de la tastatura un cuvant cu cel putin una si cel mult 20 de litere ale alfabetului englez. Construiti si afisati pe ecran cuvantul obtinut prin interschimbarea primei consoane cu ultima vocala din cuvantul citit. In cazul in care cuvantul este format numai din vocale sau numai din consoane, functia va returna 0, altfel va returna 1. Se considera vocale literele a, e, i, o, u, A, E, I, O, U.
    - ex pentru: Informatica obtinem Iaformaticn
3. Afisati suma elementelor unei matrice (tablou bidimensional)
4. Construiti inversa elementelor unei matrice

Rezolvare tema 5: http://students.info.uaic.ro/~gheorghe.balan/body/exemple/tema5.cpp

Structuri

Structuri - tipul de date struct

- structura este o zona de memorie ce contine tipuri diferite de date aliniate astfel:
  - dimensiunea unei structuri este multiplu de dimensiunea celui mai mare camp din structura (multiplu de 1, 2, 4)
    - ex: daca structura contine un int, un char si un short, dimensiunea structurii va fi multiplu de 4 - sizeof(int)
  - fiecare element din structura va incepe la multiplu de dimensiunea sa:
    - ex: short incepe la multiplu de 2, int la multiplu de 4, char la multiplu de 1
- modul in care dam elementele in structura influenteaza size-ul structurei
  - exemplu:
    - struct ex1 { char a, char b, short c, int d} va arata asa (size char - 1, size short 2, size int - 4), sizeof (ex1) fiind 8:
      - a b c c d d d d
      - 0 1 2 3 4 5 6 7
    - struct ex2 { char a, short c, char b, int d} va arata asa, sizeof (ex2) fiind 12:
      - a _ c c b _ _ _ d d d d
      - 0 1 2 3 4 5 6 7 8 9 10 11
    - struct ex3 { int d, char c} va arata asa, sizeof(ex3) fiind 8:
      - d d d d c _ _ _
      - 0 1 2 3 4 5 6 7
- AM UTILIZAT DOAR CELE 2 REGULI DE ALINIERE DE MAI SUS

Structuri - ASM

- asemanator cu tablourile unidimensionale, numai ca deplasamentul fata de pointerul initial il calculam de mana
- exemplu de rulat si de inteles:
  - http://students.info.uaic.ro/~gheorghe.balan/body/exemple/structuri.cpp
- sursa complementara de informatii (by Silviu Vitel):
  - http://students.info.uaic.ro/~silviu.vitel/labs/lab11.pdf

Exercitii

1. Fiind date 2 puncte a (int x, int y) si b (int x, int y) calculati distanta intre cele doua puncte (veti simula sqrt intreg in ASM)

Rezolvare tema 6: http://students.info.uaic.ro/~gheorghe.balan/body/exemple/tema6.cpp

Siiiiii..... exercitii recapitulative rezolvate:

http://students.info.uaic.ro/~gheorghe.balan/body/recapitulare.cpp

Cum facem debug? (https://youtu.be/M6QKUE_oHyg)

- http://students.info.uaic.ro/~gheorghe.balan/body/exemple/how_to_debug.cpp

Alte notiuni

- Exemplu de lucru cu stiva si variabile `locale`:
  - push 0 //in acest moment, ca sa accesam variabila locala folosim [esp]
  - push eax; //in acest moment, ca sa accesam variabila locala folosim [esp+4]
  - push ebx; //in acest moment, ca sa accesam variabila locala folosim [esp+8]
  - pop ebx; //in acest moment, ca sa accesam variabila locala folosim [esp+4]
  - pop eax; //in acest moment, ca sa accesam variabila locala folosim [esp]
- Recapitulare - 16.01.2020: http://students.info.uaic.ro/~gheorghe.balan/body/exemple/recapitulare2.cpp
- Simulare - 17.01.2020: https://docs.google.com/document/d/1WpPh9p5Ed_5w4EgpeU952qGlDCtMKFVSROguaTB_2XY/edit?usp=sharing

Alte notiuni

- Recapitulare 17.01.2021
  - exemplu complex cu liste inlantuite si structura: http://students.info.uaic.ro/~gheorghe.balan/body/exemple/recapitulare3_exemplu_structuri_liste_inlantuite.cpp
  - exemplu matrice (pune cu - numerele prime din matrice): http://students.info.uaic.ro/~gheorghe.balan/body/exemple/recapitulare3_exemplu_matrice_putin_mai_complex.cpp
  - exemplu apel recursiv: http://students.info.uaic.ro/~gheorghe.balan/body/exemple/recapitulare3_factorial.cpp