asm1

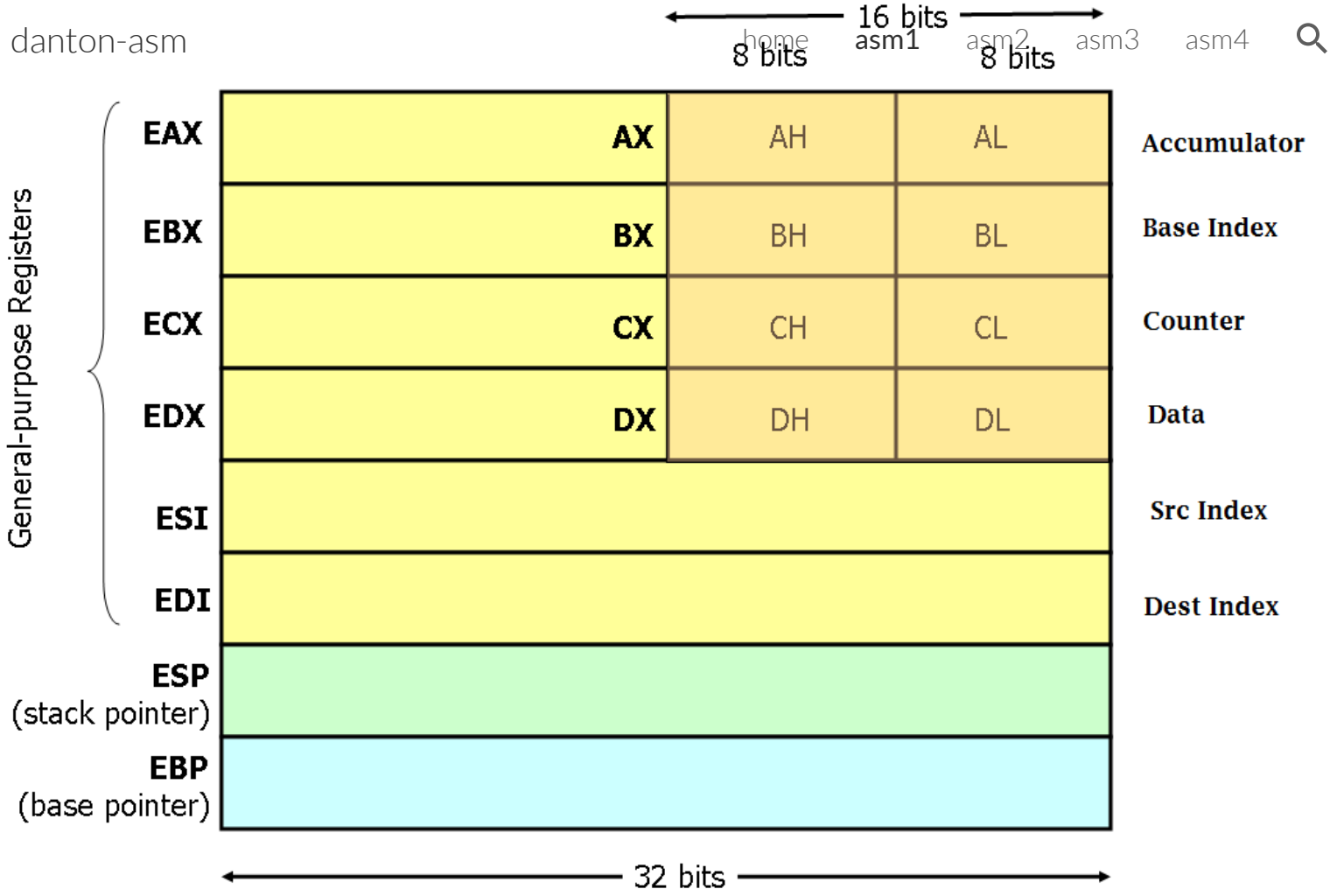
Registri x86

Arhitectura x86 are:

- 8 registri de uz general (**GPR** - General-Purpose Registers)
- 1 registru cu flag-uri/indicatori de conditii (**EFLAGS**)
- 1 registru pointer la instructiunea urmatoare ce va fi executata (**IP** - Instruction Pointer)
- alti registri

Registri General Purpose

- AX - accumulator - folosit in operatii aritmetice
- CX - counter - folosit in bucle si in instructiuni de shiftare si rotatie
- DX - data - folosit in operatii aritmetice si I/O
- BX - base - folosit ca un pointer la date
- SP - stack pointer - pointer la varful stivei
- BP - stack base pointer - pointer la baza stivei
- SI - source index - pointer la sursa in operatii stream
- DI - destination index - pointer la destinatie in operatii stream



(E)FLAGS

Este un registru de 32 de biti care indica "starea" procesorului la un moment dat. Doar o parte din cei 32 de biti sunt folositi pentru a furniza informatii despre rezultatul ultimei operatii executate de procesor. Bitii din EFLAGS se mai numesc si indicatori:

- CF - carry flag (transport) - are valoarea 1 (este setat) daca dupa ultima operatie a aparut transport, 0 (nu este setat) altfel
- PF - parity flag (paritate) - are valoarea 1 daca numarul de biti de 1 din rezultatul ultimei operatii este par
- ZF - zero flag - are valoarea 1 daca rezultatul ultimei operatii a fost 0
- SF - sign flag (semn) - are valoarea 1 daca rezultatul ultimei operatii a fost negativ (bitul cel mai semnificativ este 1)
- OF - overflow flag (depasire) - are valoarea 1 daca ultima operatie a produs depasire aritmetica

Mai multe informatii: https://en.wikipedia.org/wiki/FLAGS_register

Orice adresa este formata din doua componente: segment si offset (deplasament), notatia uzuala fiind *segment:offset*.

Pentru partea de offset exista mai multe variante:

- Constanta numerica
  - [100]
- Valoarea unui registru general pe 32 biti
  - [EAX] (se poate scrie si cu litere mici)
- Suma dintre valoarea unui registru general pe 32 biti si o constanta
  - [EBX+5]
- Suma a doi registri generali pe 32 biti
  - [ECX+ESI]
- Combinatia celor 2 variante anterioare: suma a 2 registri si a unei constante
  - [EDX+EBP+14]
- Suma a 2 registri, dintre care unul inmultit cu 2, 4, sau 8, la care se poate aduna o constanta
  - [EAX+EDI*2]
  - [ECX+EDX*4+5]

Sintaxa

Exista 2 sintaxe diferite pentru limbajul de asamblare.

- Intel - folosit in Visual Studio
- AT&T - folosit in DevC++ / Linux

Pentru a vedea diferentele, consultati pagina: http://asm.sourceforge.net/articles/linasm.html#Syntax

In cadrul laboratorului vom folosi sintaxa Intel, deci vom lucra in Visual Studio. Cei care nu vor sa foloseasca Visual Studio, se vor documenta pentru sintaxa AT&T.

Instructiuni

Instructiunile ASM au urmatorul format:

```
[eticheta]   mnemonic/operatie   [operanzi]
```

MOV - atribuire

Sintaxa: *mov destinatie, sursa*

Efect: pune in destinatie valoarea din sursa.

Destinatia, respectiv sursa, pot fi:

- registru, registru.
  - *mov eax, ebx*
  - *mov al, bh*
- registru, adresa de memorie
  - *mov bl, [eax]*
- adresa de memorie, registru
  - *mov [esi], esx*
- registru, constanta numerica
  - *mov ah, 0*
- memorie, constanta numerica
  - *mov [eax], 3*

Ex. 1:

```
#include <stdio.h>

void main() {
  _asm {
    mov eax, 0;
    mov ah, 1;
  }
}
```

Ex. 2 - eroare compilare

```
#include <stdio.h>

void main() {
  int i = 0;
  _asm {
    mov ax, i;
  }
}
```

Dimensiunea operanzilor / directiva de size:

```
mov byte ptr [eax], 5;   //afecteaza 1 octet
mov word ptr [eax], 5;   //afecteaza 2 octeti
mov dword ptr [eax], 5;  //afecteaza 4 octeti (double word)
```

ADD - adunare

Sintaxa: *add op1, op2*

Efect: op1 = op1 + op2

Ex. 1:

```
#include <stdio.h>

void main(){
  int a=10;
  _asm {
    add a, 5
  }
  printf("%d\n",a);
}
```

Ex. 2:

```
#include <stdio.h>

void main() {
  _asm {
    mov eax, 0xFFFFFFFF
    add eax, 2 // rezultatul este 0x100000001; necesita 33 biti.
               // setare carry
    mov eax, 0
    mov ax, 0xFFFF
    add ax, 2 // doar ax se modifica!
              // desi rezultatul este 0x10001, al 17-lea bit din eax nu se modifica.
              // se seteaza carry
  }
}
```

Sintaxa: *sub op1, op2*

Efect: op1 = op1 - op2

Ex. 1:

```
#include <stdio.h>

void main() {
  int a=10, b=14;
  _asm {
    mov eax, b
    sub a, eax
  }
  printf("%d\n", a);
}
```

Sintaxa:

- *and* destinatie, sursa
- *or* destinatie, sursa
- *xor* destinatie, sursa
- *not* destinatie

Instrucţiunile *and*, *or*, *xor modifică indicatorul ZERO*

Utilitatea principală a acestor instrucţiuni este in lucrul cu măşti. De exemplu, dacă ne interesează valoarea bitului al 5-lea din registrul ax, este suficient să se execute *and* intre *ax* şi valoarea (scrisa binar) 0000000000010000 (aceasta se numeşte mască). Rezultatul operaţiei va fi 0 (iar indicatorul ZERO va deveni 1) dacă bitul al 5-lea din ax are valoarea 0, respectiv va fi diferit de 0 (iar indicatorul ZERO va deveni 0) dacă bitul al 5-lea din ax are valoarea 1.

Dezavantajul abordării de mai sus este acela că instrucţiunea *and* modifică valoarea primului operand, plasind acolo rezultatul operaţiei.

Instrucţiunea *test* are acelaşi efect ca şi *and* (execută AND intre biţii celor doi operanzi, modifică la fel indicatorul ZERO), dar *nu alterează valoarea primului operand*. De exemplu:

```
test ax, 0x0010 // binar: 0000000000010000
```

modifică indicatorul ZERO ca şi

```
and ax, 0x0010
```

fără a altera valoarea din ax.

Informatii extra

- http://en.wikipedia.org/wiki/X86
- http://www.cs.virginia.edu/~evans/cs216/guides/x86.html
- http://asm.sourceforge.net/articles/linasm.html#Syntax