# Optimizări

De citit:
capitol 16 din Ray Sefarth, 15 din Richard Blum

Modificat: 22-Oct-23

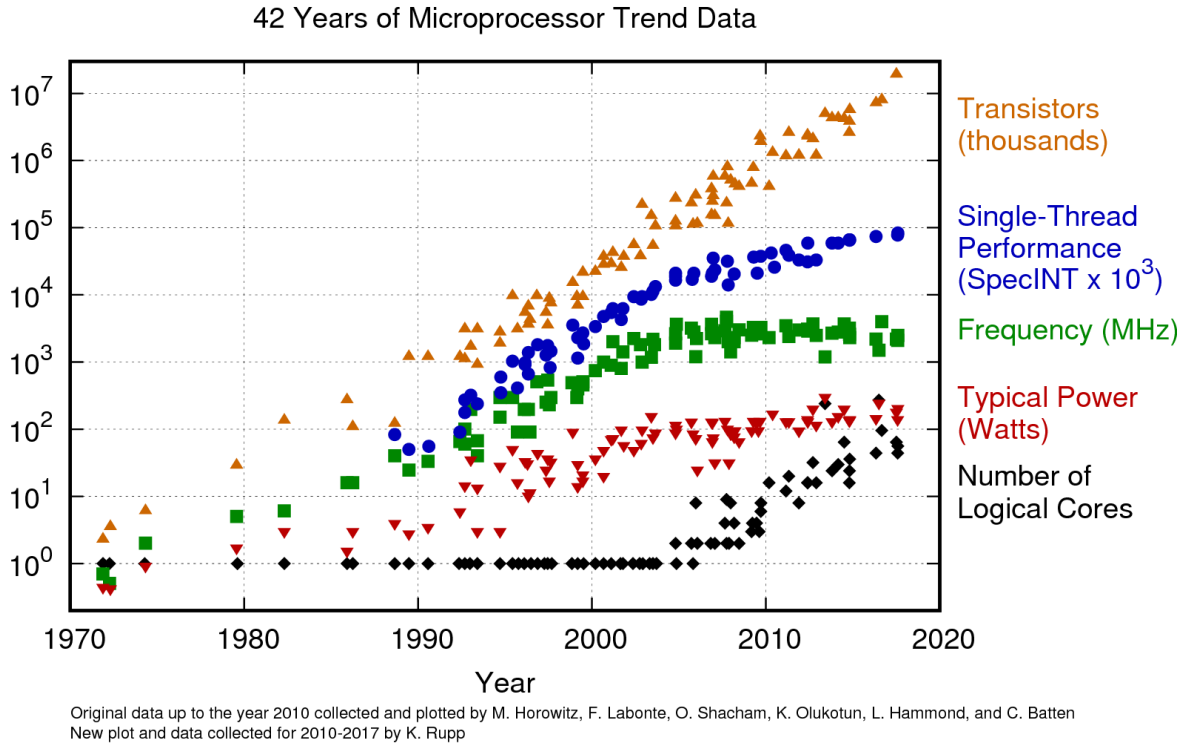
# Bibliografie


- Ray Sefarth, “Introduction to 64 Bit Intel Assembly Language Programming for Linux”, 2011, capitolul 16
- Richard Blum, “Professional Assembly Language” , Wiley 2005, capitolul 15

# Sumar

1. Profilare: *"Testing tells the truth"*
   - **`perf`**
   - **`gprof`**
   - **`clock/rdtscp`**
   - **`papi`**
2. Optimizări
   * comune pentru C/C++ și assembly
   * Optimizări pe care compilatorul le poate face în C
   * optimizări numai pentru assembly
   * extensii hardware SIMD, AVX, SSE

# Tendințele procesoarelor




- Plafonare
  - Performanța per core
  - Frecvența
  - Puterea consumată
- Ce fel de optimizări se pot face?

# Profilare

# Profilare/Masuratori

- Programe mari => greu de profilat
- “Hotpaths”
  - Ce sunt?
  - Cum le identificam?
  - Cum le îmbunătățim?
- Utilizarea unei unelte adecvate contextului

# Perf

- Utilitar Linux
- Access la contoarele de performanta ale CPU
- Evenimente software & evenimente hardware
- Masurare non-invaziva
- Granularitate mare
- Demo

# Gprof

- Utilitar cu suport in compilatorul C
- Profileaza la nivel de functie
- Ofera informatii despre
  - Timpul de executie al fiecarei functii si procentajul din timpul total de executie al programului
  - Un graf de apeluri ale functiilor
- Programele se compileaza cu optiunea `-pg`
- Ne ajuta sa identificam functia in care se petrece cel mai mult timp (aflata pe hotpath)
- Demo

# Masurare timp – functii de nivel inalt

- **clock, gettimeofday** etc.
- Functii de biblioteca
- Masoara timpul
- Masoara la nivel de microsecunda
- Pot fi folosite la o granularitate foarte mica pentru a masura portiuni mici de cod
- Demo

# Masurare timp – instructiune low level

- **rdtscp**
- Instructiune assembly
- Masoara numarul de cicli
- Masoara la nivel de ciclu
- Pot fi folosite la o granularitate foarte mica pentru a masura portiuni mici de cod
- Nu se poate masura efectiv timpul (frecventa se poate schimba)
- Demo

# PAPI – masurare alte criterii

- Intelegerea “bottleneck”-ului depinde de comportamentul codului
- Branch miss, cache miss, loads, stores, stalls etc.
- PAPI (Performance API) ne ofera posibilitatea masurarii granulare
- Similar perf, doar ca invaziv
- Demo

# Optimizari

# Taxonomie optimizari

- Criteriu: unde poate actiona programatorul
- Optimizari algoritmice
  - Cel mai bun algoritm (cu complexitatea cea mai mica)
- Optimizari de implementare
  - Shiftari in loc de inmultiri, cod invariant in afara buclei etc.
- Optimizari de arhitectura
  - Necesita cunostinte despre arhitectura procesorului
  - Analiza codului assembly rezultat
  - Introducerea in cod de instructiuni assembly mai eficiente

# Procesor superscalar

demo/curs-16/0-superscalar

- O secvența de 1000 instrucțiuni
- O buclă de 10^7 ori
- În total 10^10 (10 miliarde instrucțiuni)
- Se afișează numărul de cicli folosit

ciclu = 1/frecvență

Frecvența procesorului cat /proc/cpuinfo

$ time ./superscalar

- Se măsoară durata în secunde a execuției

- Concluzie: se execută mai multe instrucțiuni în același timp (un proces, un thread, un core)!

# Cum se poate optimiza?

- Cine optimizează? compilatorul, programatorul

- Pentru hardware
  - Superscalar? => planificare instrucțiuni
  - Instrucțiuni specifice => SIMD, SSE
  - Registre extra, reguli extra

- Optimizări generice
  - Matematică/logică
  - Generice orice limbaj
  - Algoritmice

# Utilizarea unui algoritm mai bun

- insert sort, implementată eficient, rămâne $O(n^2)$
- Sunt preferabile (rapid de scris, rapid de executat)
  - qsort din C/libc
  - C++ STL sort
- Un lookup în tabela hash => O(1)
- Dictionar sortat => STL map
- Optimizarea unui algoritm $O(n^2)$ în asamblare nu îl va converti la O(n log n)

# Eliminarea sub-expresiilor comune

- Compilatorul e adesea mai bun ca programatorul
  - compilatorul optimizează “neobosit”
- Putem examina codul generat

```
void funct1(int a, int b)
{
   int c = a * b;
   int d = (a * b) / 5;
   int e = 500 / (a * b);
   printf(“The results are c=%d d=%d e=%d\n”, c, d, e);
}
```

# Optimizări matematice simple

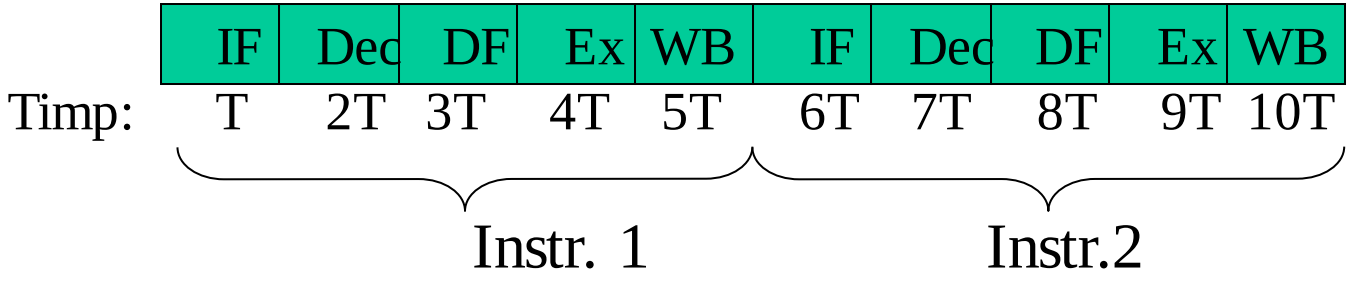
- Tehnici matematice simple (sintaxa C)
  - Împărțirea la 8 poate fi << 3
  - Restul împărțirii la 1024 se poate realiza cu operatrul &
  - În loc de pow(x,3) se folosește x*x*x
  - Pentru $x^4$, calculăm $x^2$ și apoi ridicăm la pătrat
  - Pentru diviziuni repetate la x, calculăm 1/x și refolosim
- Atenție
  - Aceste optimizări fac codul greu de citit
  - compilatorul optimizează "neobosit"
  - Compilatoarele & procesoarele evoluează
  - IMUL = 13-42 cycles(486), 3 cycles(Core I7)

# Folosirea eficientă a registrelor

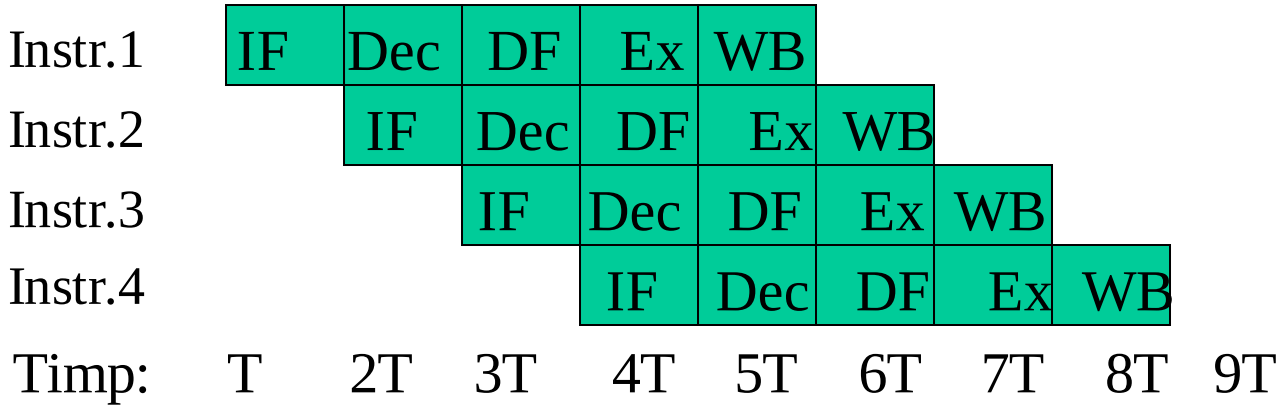
- Aplicată mereu de compilator
- Plasarea valorilor des folosite în registre
- Pentru tehnica loop unrolling, utilizarea registrelor diferite pentru a permite paralelizarea

# ne amintim: execuție secvențială vs pipeline

- Execuție secvențială




- Execuție pipeline



# Costul instrucțiunilor de tip jmp

1. jmp la distanță mare => codul nu mai este în cache
2. Se invalidează execuția pipeline
   * Cost 10-20 cicluri
   * Branch prediction
   * Speculative execution
3. Reducerea folosirii instrucțiunilor de salt
   * Cicluri cu test la sfârșit
   * Loop unrolling
   * Refactorizare bucle
   * Reducere recursivitate

# Reducerea utilizării salturilor(branch)

- compilatorul reordonează blocuri pentru a reduce salturile
- Studiem codul generat cu gcc –S
- Instrucțiunea **cmov**
  - cmov? registru, registru/memorie [?=b/w/d]
  - Testează combinații din EFLAGS, precum jz, jnae

```
mov ebx, MAX
cmp eax, ebx
cmovb ebx, eax
; ebx va conține minimul dintre eax și MAX
; se evită folosirea jb sau jnb
```

# Cicluri cu test la sfârșit

```
for ( i = 0; i < n; i++ ) {
        x[i] = a[i] + b[i];
}
```

```
for:
   <evaluate for loop counter value>
   jxx forcode;
   jmp end
forcode:
   < for loop code to execute>
   <increment for loop counter>
   jmp for
end:
```

```
if ( n > 0 ) {
        i = 0;
        do {
                x[i] = a[i] + b[i];
                i++;
        } while ( i < n );
}
```

- O singură operație de salt
- Nu faceți asta de mână

  gcc știe, chiar fără –O

# Loop unrolling

- gcc –funroll-loops **nu** este parte din –O1, –O2, –O3
- Repetarea corpului buclei pentru date consecutive
  - Numărul de repetări cunoscut la intrarea în loop
- De dorit ca fiecare repetare să folosească alte registre
- Permite execuția în altă ordine a unor instrucțiuni
- Îmbunătățește  pipeline-ul și paralelizarea

# Loop unrolling

```
for(i = 0; i < 4*n; i++){
    x[i] = a[i] + b[i];
}
```

- 4*n instrucțiuni  cmp + jmp

```
for(i = 0; i < 4*n; ){
    x[i] = a[i] + b[i];
    x[i+1] = a[i+1] + b[i+1];
    x[i+2] = a[i+2] + b[i+2];
    x[i+3] = a[i+3] + b[i+3];
    i = i + 4;
}
```

- n instrucțiuni  cmp + jmp

# Combinare de bucle

- Dacă au aceleași limite, se pot combina corpurile
  - Overhead redus (cmp + jmp)
- exemplu

```
for ( i = 0; i < 1000; i++ ) a[i] = b[i] + c[i];
for ( j = 0; j < 1000; j++ ) d[j] = b[j] - c[j];
```

- devin

```
for ( i = 0; i < 1000; i++ ) {
    a[i] = b[i] + c[i];
    d[i] = b[i] - c[i];
}
```

- cmp + jmp pentru i doar o dată
- variabilele b și c se pot refolosi

# Separare de bucle

- Parcă tocmai am propus combinarea de bucle?
- Câteodată datele sunt necorelate și combinarea nu ajută
- Poate combinarea nu permite refolosirea registrelor
- Separarea poate folosi cache-ul mai bine
- Trebuie testat codul generat
  - testing tells the truth
  - dependent de hardware

# Interschimbare bucle

```
for ( j = 0; j < n; j++ ) {
    for ( i = 0; i < n; i++ ) {
        x[i][j] = i+j;
    }
}
```

- Bucla de sus trece prin x cu pași mari
- Bucla de jos trece prin x element cu element

```
for ( i = 0; i < n; i++ ) {
    for ( j = 0; j < n; j++ ) {
        x[i][j] = i+j;
    }
}
```

- Utilizare cache mai bună

# Codul invariant în afara buclei

- Se poate face în C,  compilatorul o va face
- Asamblorul  **NU** mută cod
- Studiați codul generat  cu **gcc –S, godbolt.org**

# Evitarea recursivității

- Recursivitatea folosește stiva
  - Parametri, variabile locale, registre salvate
- Recursivitatea pe coadă (tail recursion)
  - Ultimul apel/instr din funcție este apelul recursiv
  - Nu încarcă stiva
  - Poate fi codată cu o buclă while
  - Generată de –O3
- Evitarea completă (algoritm iterativ)

# Eliminare cadru de stivă

- Funcții frunză = care nu apelează alte funcții
- gcc -fomit-frame-pointer
  - Nu se mai folosesc enter, leave
  - Doar pentru cod deja depanat
- Utilizarea ebp este opțională

# Funcții inline

- Apelul unei functii presupune anumite operatii care pot fi costisitoare:
  - Punerea parametrilor pe stiva
  - Salt la corpul functiei
  - Revenire
  - Refacere stiva
- Function inlining: substituirea apelului cu codul efectiv
- Operatiile de mai sus sunt evitate
- Demo

# Funcții inline

- **macro** în asm, **#define** în C, **inline** în C/C++
- Compilatorul optimizează "neobosit"
- În asamblare face codul mai greu de citit

# Reducerea dependețelor între instrucțiuni

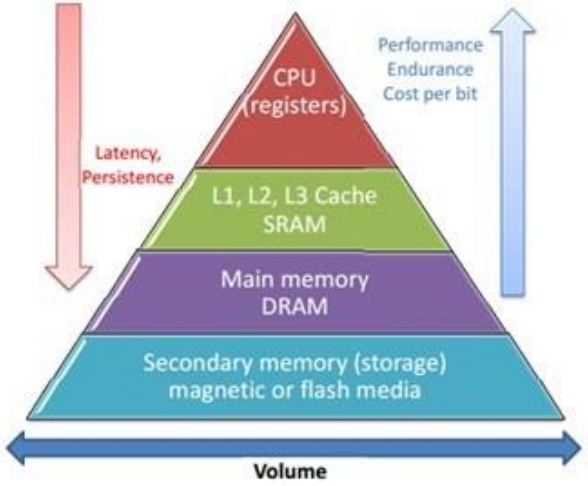
- Pentru a permite execuția superscalară
- Folosirea registrelor diferite pentru a reduce dependențele
- CPU conține ALU multiple într-un core
  - Execuție out-of-order
  - Se optimizează funcționarea pipeline
  - Se păstrează ocupate mai multe ALU
  - Demo curs-16/0-superscalar/
    - » Se execută mai multe instrucțiuni per ciclu

# Folosirea instrucțiunilor specializate

- prefetch
- popcnt
- bsf
- Lzcnt
- Sse: Instrucțiuni SIMD pe întregi
- Dificil de gestionat de compilator
- Se modernizează permanent
- Utilizarea combinată C + assembler

# Instructiunea prefetch

- Ierarhia memoriilor



# Instructiunea prefetch

- Trimite o cerere neblocanta catre cache
- In caz ca datele se afla in cache, actioneaza ca un nop
- In caz contrar, cererea este trimisa la nivelul urmator
- Ideal: toate datele se afla in cache
- Compilatoarele ofera extensii (__builtin_prefetch) pentru utilizarea facila
- Demo

# popcnt, lzcnt, bsf

- Operatii pe biti
  - popcnt: numarul de biti setati dintr-un registru
  - lzcnt: numarul de 0-uri pana la primul 1 incepand de la bitul cel mai semnificativ
  - bsf: pozitia primului bit setat porning de la cel mai putin semnificativ bit
- Mult mai eficiente ca implementarea high-level
- Extensii de compilator disponibile

# Instrucțiuni SIMD pe întregi

Instrucțiuni și registre noi

- xmm0 – xmm7  pe 128 biți
  - Pentium, Celeron >= 1999
- ymm0 – ymm31 pe 256 biți
  - Core i3/i5/i7  >=  2013
- zmm0 – zmm31 pe 512 biți
  - Intel  >= 2016 codename knights landing

# SSE

- Exemplu de optimizare: adunarea a 2 vectori de întregi  demo: curs-17/7-sse/

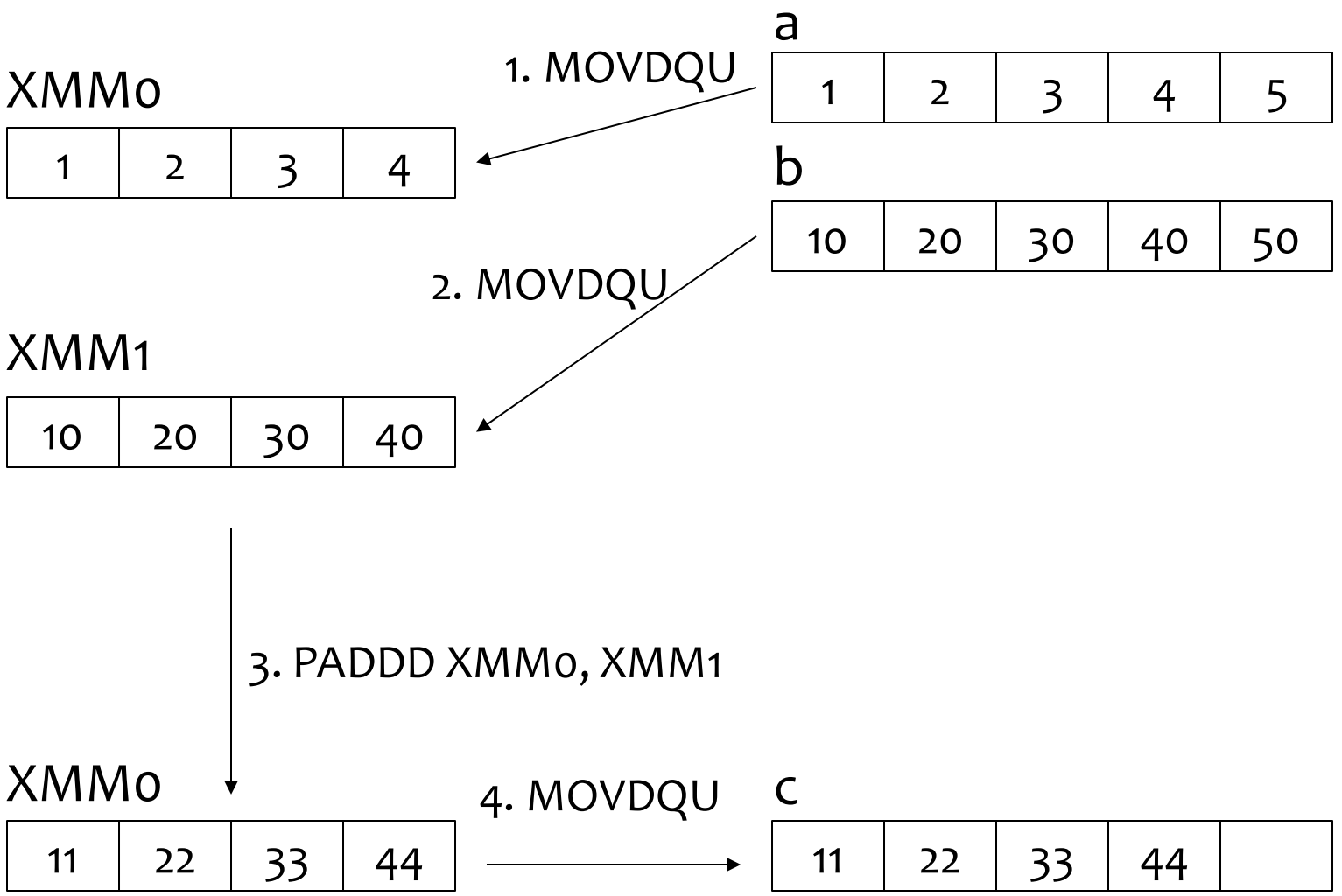
```
void sum_array(uint8_t *a, uint8_t *b, uint8_t *c, int n)
{
	int i;

	for (i = 0; i < n; i++)
		c[i] = a[i] + b[i];
}
```

- Idee: putem aduna câte 16 valori deodată folosind instrucțiunile SSE de adunare vectorială

# SSE

- Folosim registrele XMM (128 de biți văzuți ca 16 întregi pe 8 de biți, sau 4 întregi de 32 biți)
- Instrucțiuni necesare:
  - MOVDQU XMM, mem – citește 128 biți din memorie și îi împachetează într-un registru XMM
  - MOVDQU mem, XMM – idem, dar în direcția opusă
  - PADD**D** XMM0, XMM1 – adună cei 4 întregi împachetați în registrul XMM0 cu cei 4 întregi împachetați în registrul XMM1
  - PADD**B** XMM0, XMM1 – adună cei 8 întregi împachetați în registrul XMM0 cu cei 8 întregi împachetați în registrul XMM1

# SSE


a
1. MOVDQU
XMM0
1 2 3 4 5
1 2 3 4
b
10 20 30 40 50
2. MOVDQU
XMM1
10 20 30 40
3. PADDD XMM0, XMM1
XMM0
4. MOVDQU
c
11 22 33 44
11 22 33 44

# Exemplu sse.asm

```
BITS 32

GLOBAL sum_array_sse
sum_array_sse:
    push ebp
    mov ebp, esp

    push esi
    push edi
    push ebx

    mov ecx, [ebp + 20] ; ecx = n
    mov esi, [ebp + 8] ; esi = a
    mov edi, [ebp + 12] ; edi = b
    mov ebx, [ebp + 16] ; ebx = c

; n = n / 16
    shr ecx, 4
    xor eax, eax

cmp eax, ecx
    jge end
begin:
    movdqu xmm0, [esi]
    movdqu xmm1, [edi]
    paddb xmm0, xmm1
    movdqu [ebx], xmm0
    add esi, 16
    add edi, 16
    add ebx, 16
    inc eax
    cmp eaX, ecx
    jl begin
end:
    pop ebx
    pop edi
    pop esi
    leave
    ret
```

c[i] = a[i] + b[i];
Câte 16 octeți deodată

# Exemplu test_sse.c

```
#include <stdio.h>
#include <stdint.h>

void sum_array_sse(uint8_t *a, uint8_t *b, uint8_t *c, int n);

int main()
{
    uint8_t v1[] = { 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1100, 1200 };
    uint8_t v2[] = { 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000, 11000,
12000};
    uint8_t r[12];
    int n = 12;
    int i;

    sum_array_sse(v1, v2, r, n);

    for (i = 0; i < n; i++)
        printf("%u ", r[i]);
    printf("\n");

    return 0;
}
```

# Sumar optimizări

- matematică / logică
  - complexitate, constante
  - compilator
- software
  - compilator
  - minimizare acces memorie
  - optimizări loop, jmp
  - Unrolling
- hardware
  - specific hardware
  - cache, registre, superscalare
  - SSE

**testing tells the truth**