Optimizări

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

Modificat: 7-Jan-19

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

- Optimizări comune pentru C/C++ și assembly
- Optimizări pe care compilatorul le poate face în C
- Optimizări numai pentru assembly
- SIMD, SSE
- "*Testing tells the truth*"

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)

Utilizare C sau C++

- Un compilator modern implementează multe optimizări generale
- Acoperire exhaustivă, sistematică, "neobosită"
- Majoritatea codului nu este time-critical
- Poate 10% dintr-un program merită optimizat
- Greu de depășit compilatorul
- gcc –S –m32 –masm=intel hello.c
- gcc -g -Wa,-adhln –m32 –masm=intel hello.c
  - gcc -O descoperă metodele compilatorului

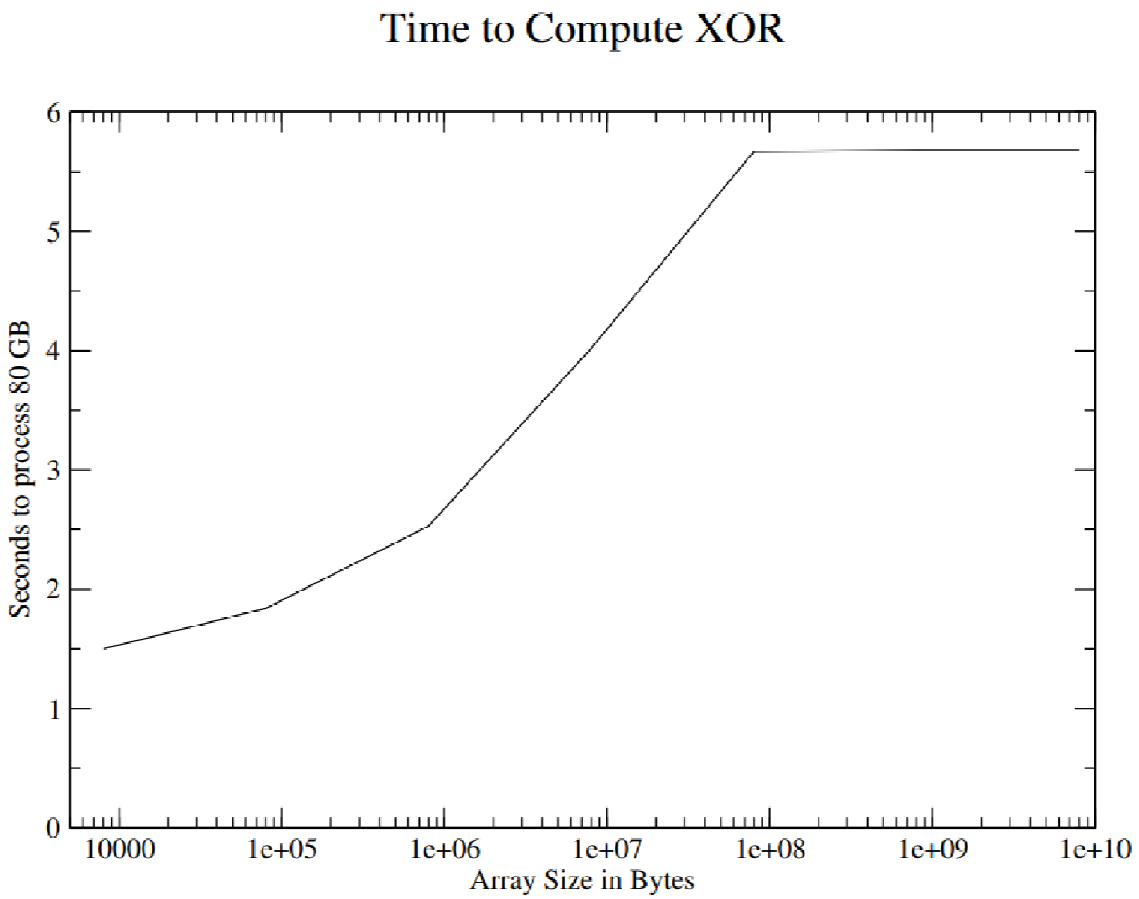
Utilizare C sau C++

- Un compilator modern implementează multe optimizări generale
- Acoperire exhaustivă, sistematică, "neobosită"
- Nivele cumulative de optimizare
  - gcc –O0 compilare rapidă, fără optimizări - default
  - gcc –O1
  - gcc –O2
  - gcc –O3
  - Tradeoffs: viteză cod, dimensiune cod, timp de compilare
  - Le putem face mai bine?
- https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Utilizarea eficientă a cache-ului

- CPU funționează la 3GHz (vezi demo test_freq.c)
- Memoria principală = lentă,
  - 21GBytes/s ~ 7octeți / ciclu
  - CPU=4cores => 2 octeți / ciclu
  - 1 core = 3 ALU, 2MMU
- Memoria cache = rapidă
  - 8MBytes

Timpul de calcul pentru 10G operatii XOR

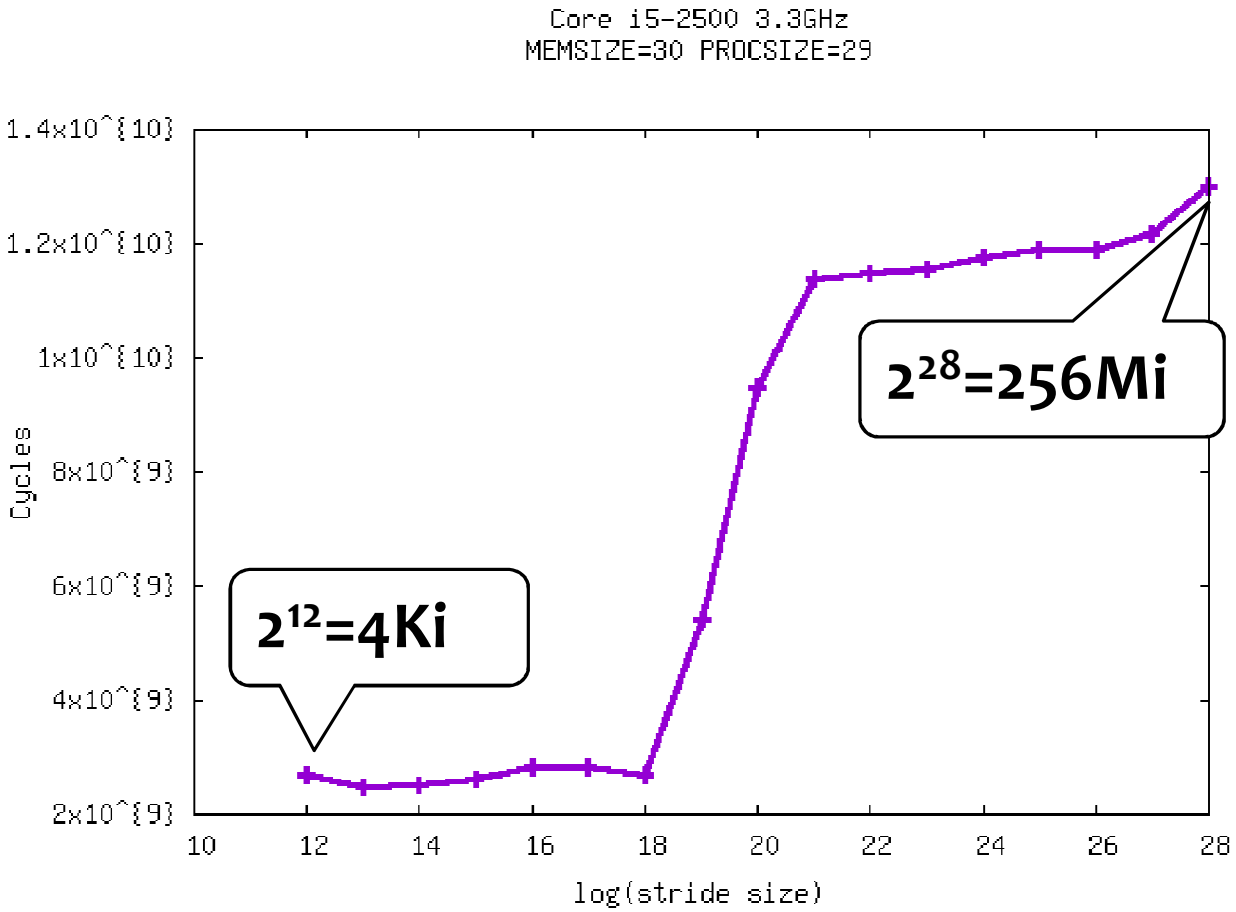


Demo xor_cache.c (similar Sefarth 16.4)

Stride = 2^2

Stride = 2^3

Același număr total de operații pentru fiecare punct

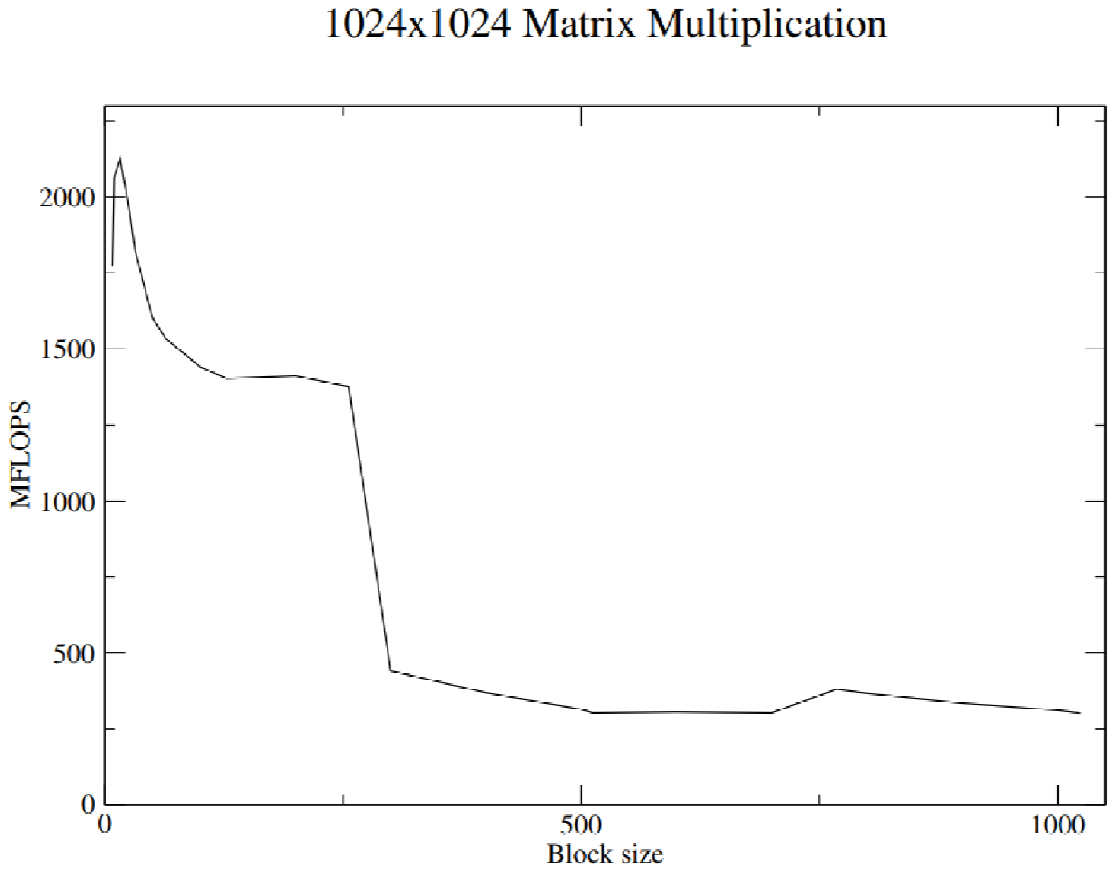


Utilizarea eficientă a cache-ului

- Organizare algoritm pentru a folosi date care încap în cache
  - Cu ce cache va fi executat codul?
  - Pe ce configurație multicore, hyperthreaded?
  - Ce viteze cpu, bus, memorie, cache?

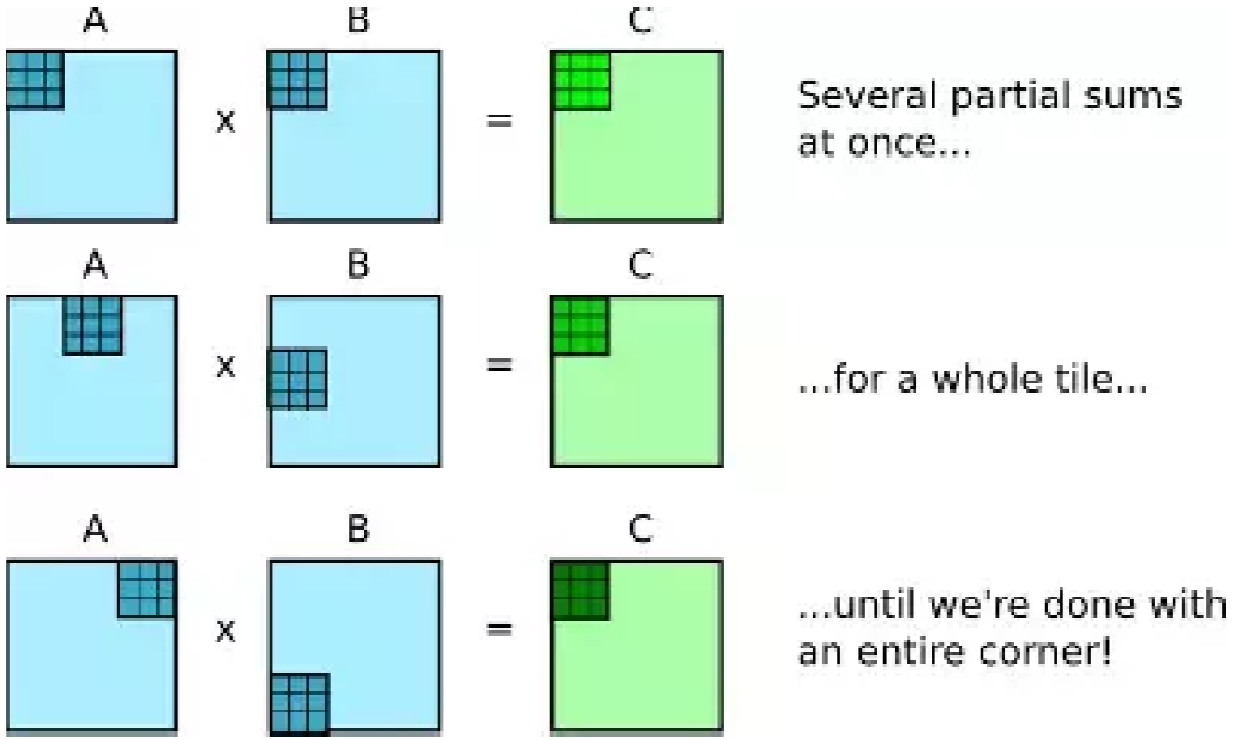
Utilizarea eficientă a cache-ului

- Înmulțire de matrici 1024 × 1024
- Codul standard folosește 3 cicluri for imbricate
- O linie = 8MB (ocupă întreg cache-ul)
- Performanță slabă (la viteza memoriei principale)



Utilizarea eficientă a cache-ului

- Înmulțire de matrici 1024 × 1024
- Codul standard folosește 3 cicluri for imbricate
- O linie = 8MB (ocupă întreg cache-ul)
- Se poate optimiza folosind blocuri mai mici
  - 6 cicluri imbricate



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

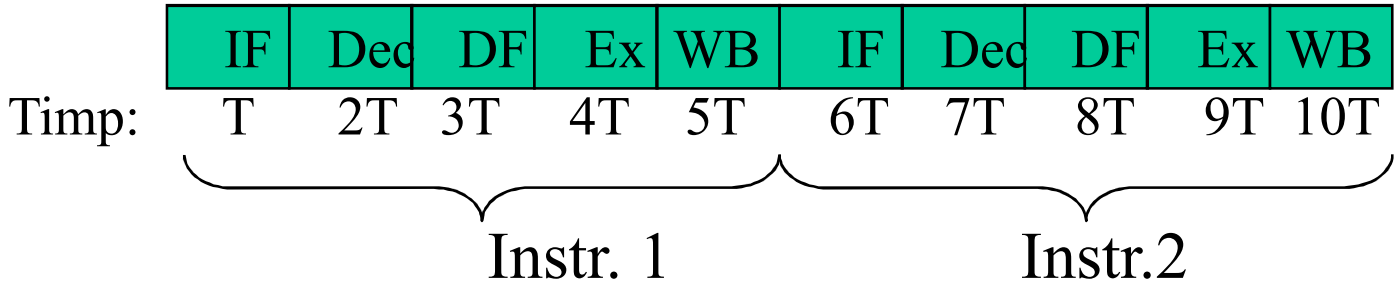
- Tehnici matematice simple (sintaxa C)
  - Împărțirea la 8 poate fi << 3
  - Restul împărțirii la 1024 se poate realiza cu operatrul &
  - În loc de pow(x,3) se folosește x*x*x
  - Pentru x^4, calculăm x^2 și apoi ridicăm la pătrat
  - Pentru diviziuni repetate la x, calculăm 1/x și refolosim
- Atenție
  - 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

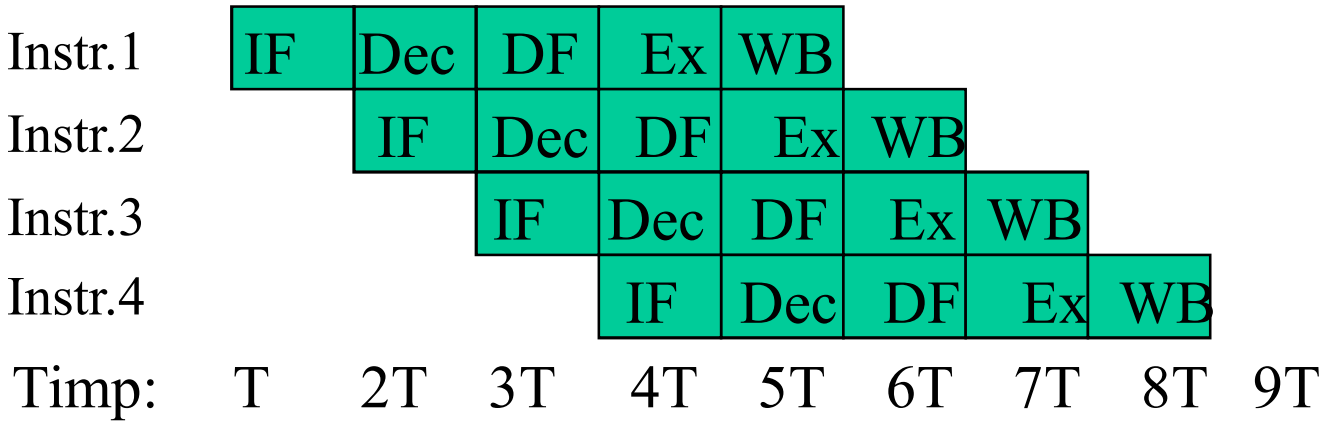
- Aplicată mereu de compilator
- Plasarea valorilor des folosite în registre
- Pentru tehnica loop unrolling, utilizarea registrelor diferite pentru a permite paralelizarea

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 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

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**

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

- **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 test_rdtsc.asm
    - » Se execută mai multe instrucțiuni per ciclu

Măsurători de performanță

- Instrucțiunea **rdtscp**
  - EDX:EAX – cicli petrecuți de la boot
  - Depinde de ceasul procesorului
  - /proc/cpuinfo (Linux), cpu-z (Windows)
  - Atenție la: frecvența variabilă, mașini virtuale, execuție superscalară
  - Atenție la: context switch, întreruperi, etc

- Exemple în curs-12-demo
  - xor_cache.c – exemplu simila Sefarth ch 16
  - test_freq.c – estimare frecvență core
  - test_rdtsc.asm – test execuție hyperthreaded
  - lock.cc – instrucțiunea inc nu este atomică
  - test_sse – instrucțiuni de tip SIMD

Folosirea combinată C + assembler

- Module separate
  - Exemple multiple Dandamudi
- Inline assembly

```
int counter = 0;
void asm_inc(){
  asm ("\
        mov ecx, 1000000                \n\
incagain:                               \n\
        inc dword ptr [ counter]        \n\
        dec ecx                         \n\
        jnz incagain                    \n\
");
}
```

- Exemplu lock.cc

Folosirea combinată C + assembler

- Exemplu lock.cc, de excutat pe multicore
- două threaduri care incrementează același întreg
- "race condition" – cursurile SO, APD
  - Instrucțiunea inc nu este atomică (fetch, decode, execute, write back)
  - Valoarea minimă 2, maximă 2*10^6
- Ce valori rezultă?
  1. Se compilează cu 2^32 iterații, se constată CPU load = 200%
  2. Se compilează cu 10^6 iterații, counter < 2*10^6
  3. Se compilează cu lock inc, incrementarea devine atomică

```
Counter = 2*10^6
```

Programare inline assembly

- pros
  - Trebuie să implementăm în assembly doar partea de cod care ne interesează
- cons
  - Control mai redus asupra codului generat
  - Sintaxa de inline assembly variază în funcție de compilator (sau lipsește complet – Visual C++ 64-bit)

Folosirea instrucțiunilor specializate

- Instrucțiuni SIMD pe întregi
- Instrucțiuni SIMD pe floating point(FP)
- Instrucțiuni AVX – permit mai multe valori FP în registrele SIMD
- Dificil de gestionat de compilator
- Se modernizează permanent
- Utilizarea combinată C + assembler

SSE

- Exemplu clasic de optimizare: adunarea a 2 vectori de întregi

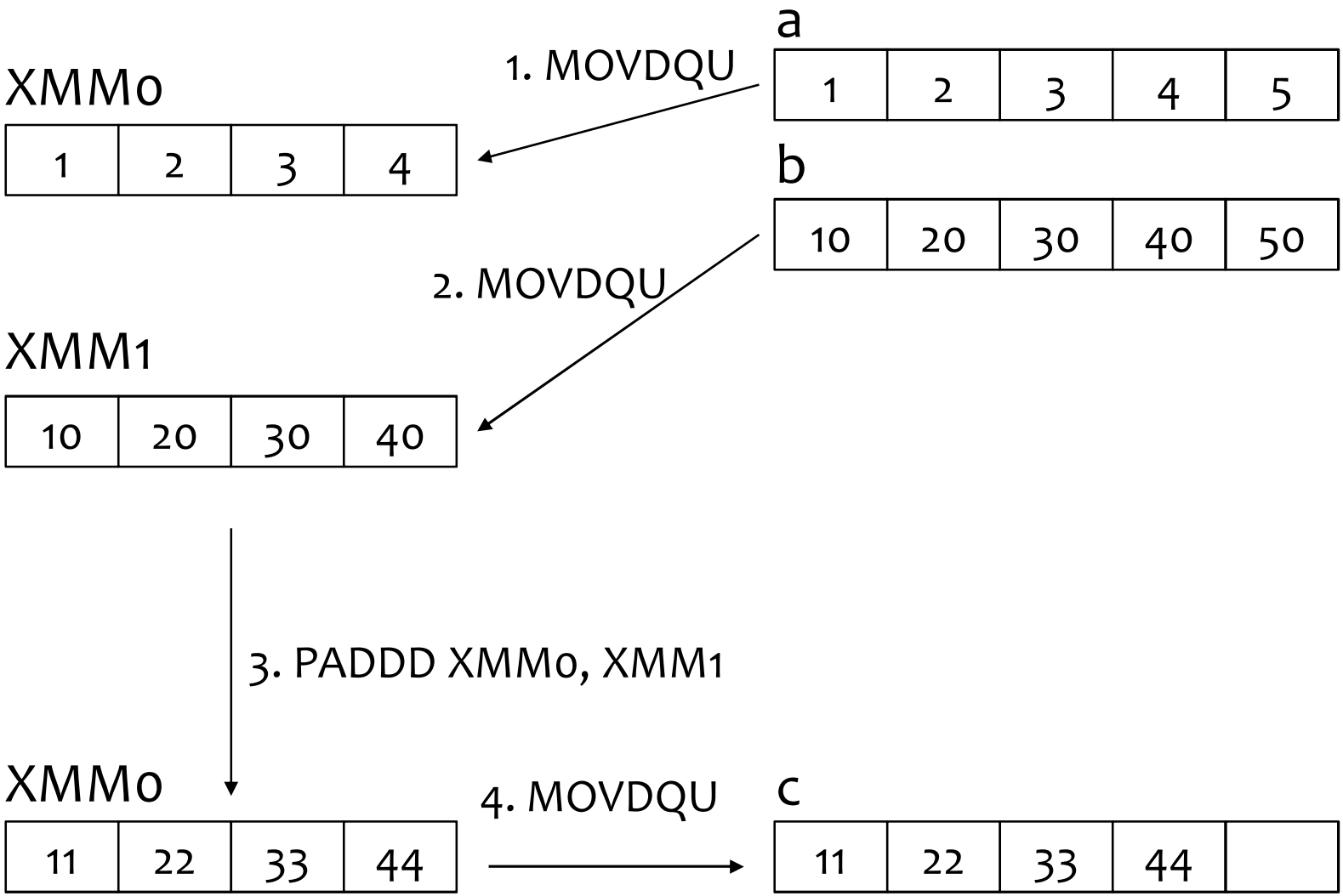
```
void sum_array(uint8_t *a, uint8_t *b, uint8_t *c, int n)
{
	int i;

	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
```

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;
}
```

Intrebari?