Processamento Vectorial:
Single Instruction Multiple Data

Arquitectura de Computadores

Lic. em Engenharia Informática

Luís Paulo Santos

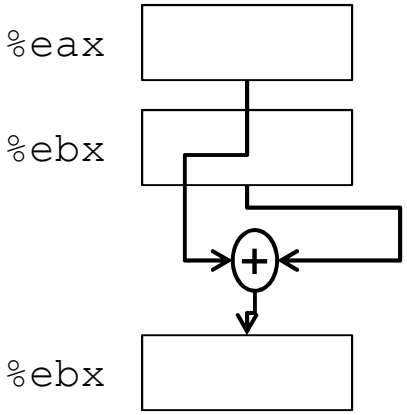
Processamento Vectortial

Conteúdos	**11 – Arquitecturas Actuais** 11.1 – Extensões SIMD ao conjunto de instruções
Resultados de Aprendizagem	R11.1 – Justificar e comparar tendências recentes na arquitectura e organização de sistemas de computação

Processamento Escalar *versus* Vectorial

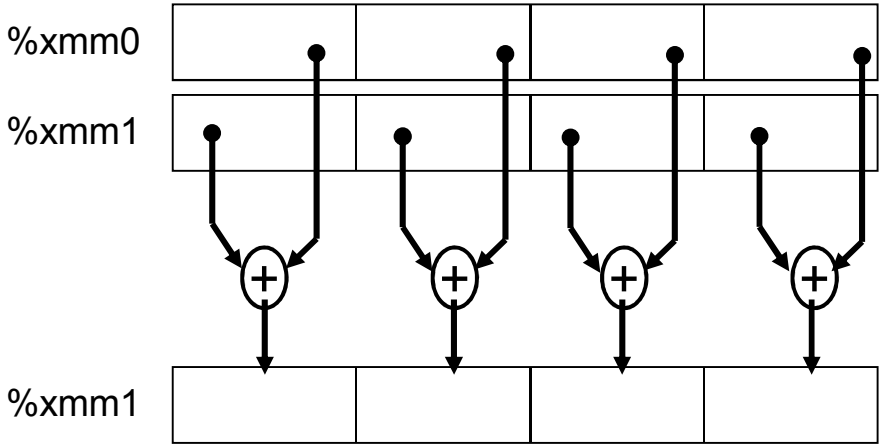
Escalar – os operandos das instruções são constituídos por um único valor, logo escalar.

```
addl %eax, %ebx
```



Vectorial – os operandos das instruções são constituídos por um vector de valores.

```
paddd %xmm0, %xmm1
```



SIMD – *Single Instruction Multiple Data*

Single Instruction Multiple Data

1994 – Pentium II e Pentium with MMX –
Intel introduz a primeira extensão SIMD ao conjunto de instruções (MMX - MultiMedia eXtensions)

1995 – Introdução de Streaming Simd Extensions (SSE) no Pentium III

2000 – Introdução de SSE2 no Pentium IV

2004 - Introdução de SSE3 no Pentium IV HT

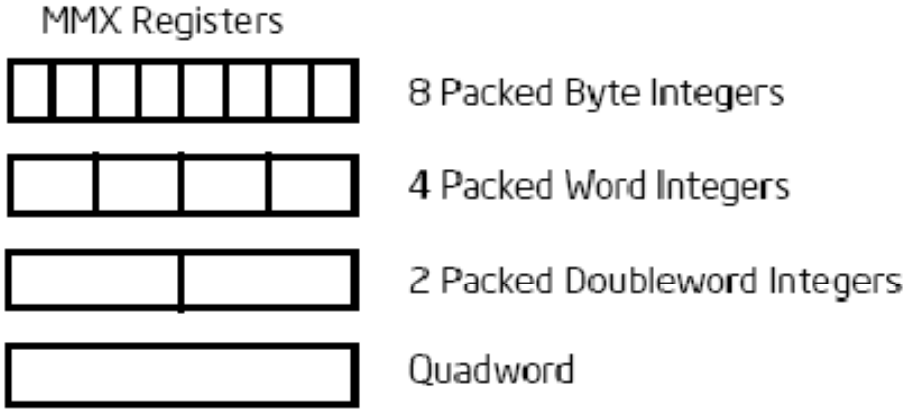
2007 - Introdução de SSE4

MultiMedia eXtensions (MMX)

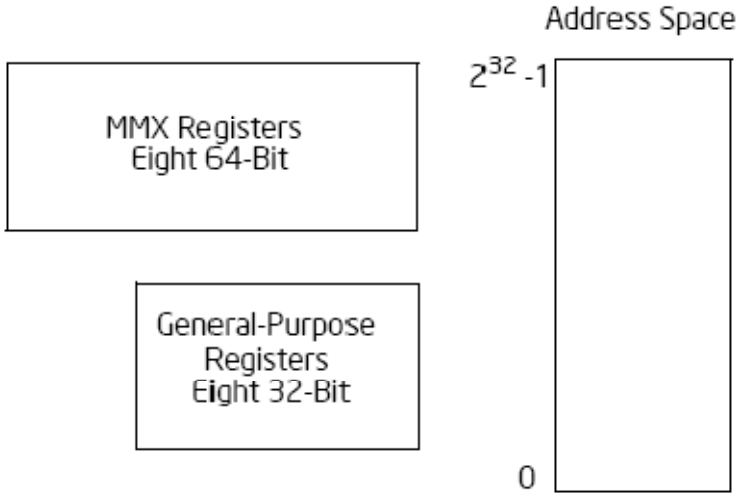
- Operações sobre inteiros

- 8 registos de 8 *bytes* (64 *bits*): mmx0 .. mmx7
  Estes registos são os mesmos da FPU

Tipos de Dados



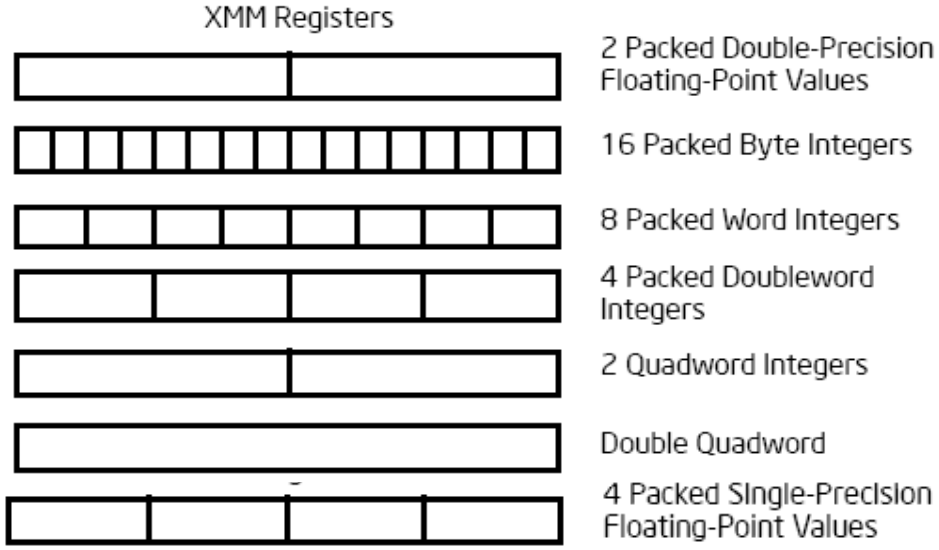
Ambiente de Programação



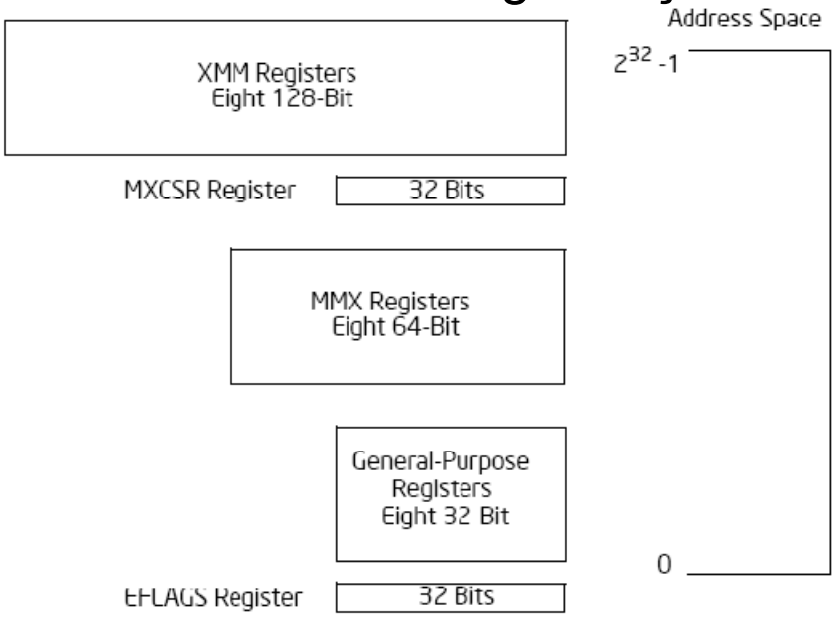
Streaming SIMD Extensions (SSE)

- Operações em vírgula flutuante
- Operações adicionais sobre inteiros
- 8 novos registos de 16 *bytes* (128 *bits*): xmm0 .. xmm7

Tipos de Dados



Ambiente de Programação



Instruções: Transferência de Dados

Instruções	**Operandos** orig, dest	**Obs.**
MOVQ	mm/m64, mm mm, mm/m64	Mover palavra quádrupla (8 bytes) de memória para registo mmx ou de registo mmx para memória (Apenas para inteiros)
MOVDQA MOVDQU	xmm/m128, xmm xmm, xmm/m128	Mover 2 palavras quádruplas (2*8 bytes) Apenas para inteiros A - addr alinhado M16; U – addr não alinhado
MOVAP[S|D] MOVUP[S|D]	xmm/m128, xmm xmm, xmm/m128	Mover 4 FP precisão simples ou 2 FP precisão dupla A – addr alinhado M16 U – addr não alinhado

Instruções: Operações Inteiras

Instruções	**Operandos** orig, dest	**Obs.**

`PADD?`
`PSUB?`
`PAND?`
`POR?`
	`mm/m64, mm`
	`xmm/m128, xmm`
		Adição, subtracção, conjunção ou disjunção do tipo de dados indicado
		Operação realizada sobre o número de elementos determinado pelo registo+tipo de dados
		Endereços em memória alinhados
		O resultado não pode ser em memória

? = B | W | D | Q

B – byte W – 2 bytes

D – 4 bytes Q – 8 bytes

Instruções: Operações FP

Instruções	**Operandos** orig, dest	**Obs.**

`ADDP?`
`SUBP?`
`MULP?`
`DIVP?`
`SQRTP?`
`MAXP?`
`MINP?`
`ANDP?`
`ORP?`
	`xmm/m128, xmm`	Operação sobre o tipo de dados indicado
		
		Operação realizada sobre o número de elementos determinado pelo tipo de dados (S = 4 ; D = 2)
		
		Endereços em memória alinhados
		
		O resultado não pode ser em memória

? = S | D

S – precisão simples

D – dupla precisão

Exemplo SSE

```
MM_ALIGN16 float a[100], b[100], r[100];

func (int n, float *a, float *b, float *r) {
  int i;
  for (i=0 ; i<n ; i++)
    r[i] = a[i] * b[i];
}
```

```
func:
  …
  movl 8(%ebp), %edx
  movl 12(%ebp), %eax
  movl 16(%ebp), %ebx
  movl 20(%ebp), %esi
  movl $0, %ecx
ciclo:
  movaps (%eax, %ecx, 4), %xmm0
  mulps (%ebx, %ecx, 4), %xmm0
  movaps %xmm0, (%esi, %ecx, 4)
  addl $4, %ecx
  cmpl %edx, %ecx
  jle ciclo
  …
```

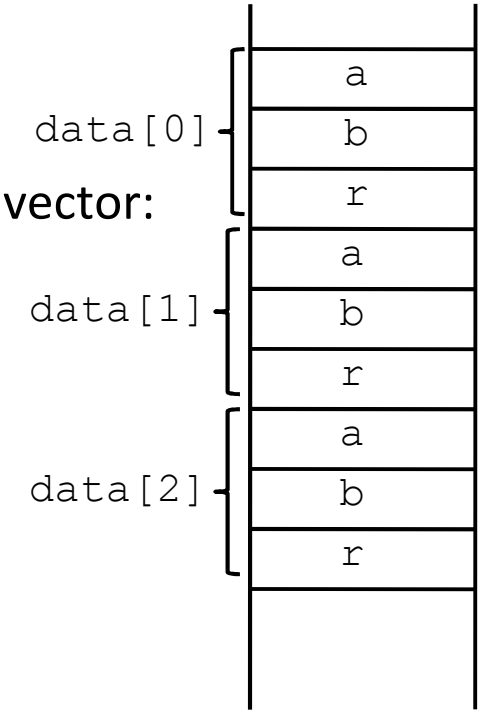
Data layout – AoS versus SoA

Exemplo anterior: `movaps (%eax, %ecx, 4), %xmm0`

- Carrega 4 elementos do vector `b` para `%xmm0`
- Isto requer que estes 4 elementos estejam armazenados em posições consecutivas de memória

- No entanto, o modelo habitual de programação usa vectores de estruturas (AoS – A*rray of Structures*) que resulta na dispersão dos elementos do mesmo vector:

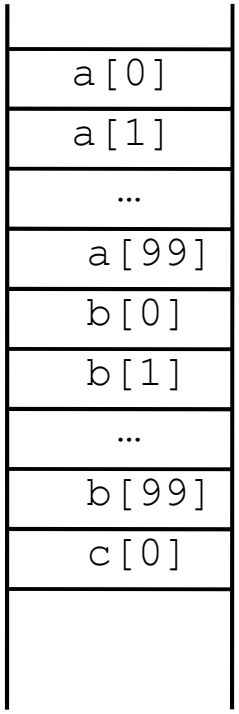
```
struct {
  float a, b, r;
} data[100];
```



Data layout – AoS versus SoA

- Para que os vários elementos do mesmo campo (ou vector) sejam armazenados consecutivamente em mémória é necessário usar uma codificação do tipo (SoA – *Structure of Arrays*)

```
struct {
  float a[100], b[100], r[100];
} data;
```



Compiler Intrinsics

- *Compiler intrinsics* são pseudo-funções que expõem funcionalidades do CPU incompatíveis com a semântica da linguagem de programação usada (C/C++ neste caso)

EXEMPLO:

- A adição ou divisão de dois valores mapeiam perfeitamente nos operadores '+' e '/'

- A multiplicação de um vector de 4 valores inteiros seguida da soma dos produtos (MMX) não mapeia em nenhum operador
  - O compilador disponibiliza o tipo de dados `__m64` e a pseudo-função `_m_pmadddw (__m64, __m64)` utilizáveis pelo programador

Compiler Intrinsics

- As funções e tipos de dados definidos como *intrinsics* são acessíveis incluindo o *header* `<ia32intrin.h>`

Tipos de Dados	
__m64	Vector de 64 bits – inteiros (MMX)
__m128	Vector 128 *bits* – 4 FP SP (SSE)
__m128d	Vector 128 *bits* – 2 FP DP (SSE2)
__m128i	Vector 128 *bits* – inteiros (SSE2)

Compiler Intrinsics

Operações Aritméticas		
Pseudo-função	Descrição	Instrução
`__m128 _mm_add_ps (__m128, __m128)`	Adição	`ADDPS`
`__m128 _mm_sub_ps (__m128, __m128)`	Subtracção	`SUBPS`
`__m128 _mm_mul_ps (__m128, __m128)`	Multiplicação	`MULPS`
`__m128 _mm_div_ps (__m128, __m128)`	Divisão	`DIVPS`
`__m128 _mm_sqrt_ps (__m128)`	Raiz Quadrada	`SQRTPS`
`__m128 _mm_rcp_ps (__m128)`	Inverso	`RCPPS`
`__m128 _mm_rsqrt_ps (__m128)`	Inverso da Raiz Quadrada	`RSQRTPS`

Compiler Intrinsics

Acesso à Memória		
Pseudo-função	Descrição	Instrução
`__m128 _mm_load1_ps (float *)`	Carrega 1 valor para os 4 elementos do vector	`MOVSS + Shuffling`
`__m128 _mm_load_ps (float *)`	Carrega vector de memória para registo (alinhado 16)	`MOVAPS`
`__m128 _mm_loadr_ps (float *)`	Carrega vector de memória para registo em ordem inversa (alinhado 16)	`MOVAPS + Shuffling`
`_mm_store_ps (float *, __m128)`	Escreve registo em vector de memória (alinhado 16)	`MOVAPS`
`_mm_storer_ps (float *, __m128)`	Escreve registo em vector de memória por ordem inversa (alinhado 16)	`MOVAPS + Shuffling`

Compiler Intrinsics

Set		
Pseudo-função	Descrição	Instrução
`__m128 _mm_set1_ps (float)`	Carrega 1 constante para os 4 elementos do registo	`Várias`
`__m128 _mm_set_ps (float, float, float, float)`	Carrega 4 constantes para os 4 elementos do registo	`Várias`
`__m128 _mm_setzero_ps (f)`	Coloca os 4 elementos do registo a zero	`Várias`

Compiler Intrinsics

Comparação		
Pseudo-função	Descrição	Instrução
`__m128 _mm_cmpeq_ps (__m128, __m128)`	Põe a 1 se iguais	`CMPEQPS`
`__mm_cmp[lt, le, gt, gem neq, nlt, ngt, nle, nge]`		

A comparação é feita elemento a elemento dos registos %xmm,
Sendo o resultado um registo %xmm com o elemento correspondente a 0 ou 1

Compiler Intrinsics: Exemplo

```
#include <math.h>
float a[100], b[100], r[100];

func() {
  for (int i=0 ; i<100 ; i++) {
    r[i] = 5. * (a[i] + sqrt(b[i]));
} }
```

```
#include <ia32intrin.h>
MM_ALIGN16 float a[100], b[100], r[100];

func() {
  __m128 cinco = _mm_set1_ps (5.);
  for (int i=0 ; i<100 ; i+=4) {
    __m128 mb = _mm_sqrt_ps(_mm_load_ps (&b[i]));
    __m128 ma = _mm_load_ps(&a[i]);
    __m128 mr = _mm_mul_ps (cinco, _mm_add_ps (ma, mb);
    _mm_store_ps (&r[i], mr);
} }
```

Compiler Intrinsics: Exemplo

```
#include <ia32intrin.h>
struct {
 MM_ALIGN16 union {float a[4], __m128 ma};
 MM_ALIGN16 union {float b[4], __m128 mb};
 MM_ALIGN16 union {float r[4], __m128 mr};
} d[25];

func() {
  __m128 cinco = _mm_set1_ps (5.);
  for (int i=0 ; i<25 ; i++) {
    __m128 aux = _mm_sqrt_ps(d[i].mb);
    d[i].mr = _mm_mul_ps (cinco, _mm_add_ps (d[i].ma, aux);
} }
```

Vectorização pelo compilador

```
typedef struct {
  float a, b; } data;
data arr[100];

func() {
  float mult=1.;
  for (int i=0 ; i<100 ; i++)
    mult *= arr[i].a;
}
```

O compilador será capaz de vectorizar este código automaticamente?

Não, pois o estilo AoS dos dados impede o carregamento de 4 elementos de arr[].a com uma só instrução

```
typedef struct {
  float a[100], b[100];} data;
data arr;

func() {
  float mult=1.;
  for (int i=0 ; i<100 ; i++)
    mult *= arr.a[i];
}
```

O compilador será capaz de vectorizar este código automaticamente?

Sim, pois o estilo SoA dos dados permite o carregamento de 4 elementos de arr.a[] com uma só instrução