

Streaming SIMD Extensions

A taxonomia de Flynn

- Quatro classes de arquiteturas de computadores (Tanenbaum, 2001)
- SISD (Single Instruction Single Data): Fluxo único de instruções sobre um único conjunto de dados.
- SIMD (Single Instruction Multiple Data): Fluxo único de instruções em múltiplos conjuntos de dados.
- MISD (Multiple Instruction Single Data): Fluxo múltiplo de instruções em um único conjunto de dados. Impraticável tecnologicamente.
- MIMD (Multiple Instruction Multiple Data): Fluxo múltiplo de instruções sobre múltiplos conjuntos de dados.

Streaming SIMD Extensions

- A tecnologia MMX
 - Tecnologia Single Instruction, Multiple Data (SIMD);
 - 57 novas instruções;
 - 8 registradores MMX de 64 bits
 - 4 novo tipo de dados empacotados
- As SSE melhoraram a arquitetura Intel x86
 - 8 novos registradores SIMD de 128 bits de ponto flutuante que podem ser endereçado diretamente;
 - 50 novas instruções que trabalham em dados empacotados de ponto flutuante;
 - 8 novas instruções destinadas a controlar cacheabilidade de dados;
 - 12 novas instruções estender o conjunto de instruções MMX.

Desenvolvimento

- MMX (MultiMedia eXtentions) Architecture
- MMX Instructions
- SSE (Streaming SIMD Extensions)
- SSE2
- SSE3
- SSSE3 (Supplemental Streaming SIMD Extensions 3)
- SSE4
- AVX (Advanced Vector Extension)
- AVX2

Registradores SSE

- 16 registradores, cada um pode ser usado para armazenar e operar
 - Dois números em ponto flutuante de 64-bit de precisão dupla ou
 - Quatro números em ponto flutuante de 32-bit de precisão dupla ou
 - Dois inteiros de 64-bit ou
 - Quatro inteiros de 32-bit ou
 - Oito inteiros curtos de 16-bit ou
 - Dezesseis inteiros de 8-bit ou caracteres.

_m128	Float	Float	Float	Float	4x 32-bit float
_m128d	Double	Double			2x 64-bit double
_m128i	B	B	B	B	16x 8-bit byte
_m128i	short	short	short	short	8x 16-bit short
_m128i	int	int	int	int	4x 32bit integer
_m128i	long	long	long	long	2x 64bit long
_m128i					1x 128-bit quad

AVX Data Types (16 YMM Registers)

_mm256	Float	Float	Float	Float	Float	Float	Float	8x 32-bit float
_mm256d	Double	Double	Double	Double				4x 64-bit double
_mm256i								256-bit integer registers. It behaves similarly to _m128i. Out of scope in AVX, useful on AVX2.

Sintaxe das Instruções MMX

Usada para Inteiros

- Todas com 2 operandos
 - Primeiro operando é o destino, segundo a origem
- Uma instrução típica pode ser acompanhada de caracteres indicativos
 - Prefixo: P for Packed
 - Operação: por exemplo - ADD, MIN, or XOR
 - Sufixo:
 - US for Unsigned Saturation
 - S for Signed saturation
 - B, W, D, Q tipo dos dados: packed byte, packed word, packed doubleword, or quadword.

Saturação e Wrap-around

- Wrap-around – o resultado é truncado e somente os bits menos significativos são retornados
 - ex.: PADDB, PADDW, PADDD
- Saturação – Os resultados em *overflow* (de uma adição) ou *underflow* (de uma subtração) são levados para seus maiores ou menores valores que podem ser representados no tipo.
 - ex.: PADD\$B, PADD\$W, PADD\$USB, PADD\$USW
 - Por exemplo, na soma de dois bytes sem sinal
 - 200+100 resultam em overflow
 - Com wrap-around → resultado = 44
 - Com saturação → resultado = 255

Categorias de Instruções

- Movimento de Dados
 - movss, movups, movaps, etc.
- Instruções Aritméticas
 - addps, mulss, divps, sqrtss, minps, maxps, etc.
- Instruções Lógicas
 - Só disponíveis em dados empacotados
 - xorps, orps, andnps, etc.
- Instruções de Comparação
 - cmpss, cmpps, etc.
- Instruções de Conversão Integer-Real
 - cvtss2pi, cvtsi2ss, etc.
- Instruções Shuffle/Rearrange
 - shufps, unpcklps, etc

Movimentos de Dados

- Instruções para copiar os dados entre os registadores MMX e a memória:
 - MOVSS**: copia os 32 bits menos significativos
 - DEST[31-0] ← SRC[31-0]; DEST[127-32] unchanged;
 - MOVLPS**: atribui apenas os 64 bits menos significativos
 - MOVHPS**: atribui apenas os 64 bits mais significativos
 - MOVAPS**: cópia **alinhada** de 16 bytes (rápido)
 - MOVUPS**: cópia **desalinhada** de 16 bytes (lento)
 - MOVHLPS**
 - DEST[63-0] ← SRC[127-64]; DEST[127-64] unchanged;
 - MOVLHPS**
 - DEST[127-64] ← SRC[63-0]; DEST[63-0] unchanged;

Instruções MMX

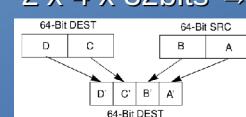
Category	Wraparound	Signed Saturation	Unsigned Saturation
Arithmetic	Addition	PADD\$B, PADD\$W, PADD\$USB, PADD\$USW	PADD\$B, PADD\$W, PSUB\$B, PSUB\$W
	Subtraction	PSUBB, PSUBW, PSUBD PMULL, PMULH PMADD	PSUB\$B, PSUB\$W
	Multiplication Multiply and Add	PCMPEQB, PCMPEQW, PCMPEQD PCMPPGTB, PCMPPGTPW, PCMPPGTPD	
Comparison	Compare for Equal		
	Compare for Greater Than		
Conversion	Pack		PACKSSWB, PACKSSDW
Unpack	Unpack High	PUNPCKHBW, PUNPCKHWD, PUNPCKHDQ, PUNPCKLBW, PUNPCKLWD, PUNPCKLDQ	
	Unpack Low		

Instruções MMX continuação

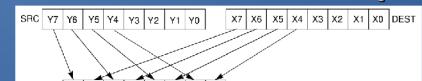
Logical	And And Not Or Exclusive OR	Packed		Full Quadword	
				PAND PANDN POR PXOR	
Shift	Shift Left Logical Shift Right Logical Shift Right Arithmetic	PSLLW, PSLLD PSRLW, PSRLD PSRAW, PSRAD		PSLLQ PSRLQ	
			Doubleword Transfers	Quadword Transfers	
Data Transfer	Register to Register Load from Memory Store to Memory	MOVD MOVD MOVD	MOVQ MOVQ MOVQ		
Empty MMX State		EMMS			

Rearranjo de Dados

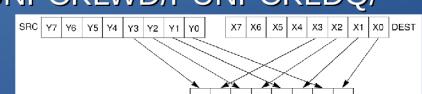
- PACKSSWB – 2 x 8 x 16bits → 1 x 16 x 8bits
- PACKSSDW – 2 x 4 x 32bits → 1 x 8 x 16bits



- PUNPCKHBW/PUNPCKHWD/PUNPCKHDQ/
PUNPCKHQDQ



- PUNPCKLBW/PUNPCKLWD/PUNPCKLDQ/
PUNPCKLQDQ



Exemplo

- PADDB - Soma 16 pares de inteiros de um byte cada

```
char a[N], b[N], c[N];
...
for (i = 0; i < N; i++)
    a[i] = b[i] + c[i];
```

```
char a[N], b[N], c[N];
...
_asm {
    mov    ebx, 0
    mov    ecx, N/16
    repete:
    movups xmm0, c[ebx]
    movups xmm1, b[ebx]
    paddb xmm0, xmm1
    movups a[ebx], xmm0
    add    ebx, 16
    sub    ecx, 1
    jnz   repete
}
```

Carrega 16 bytes em xmm0
Carrega 16 bytes em xmm1
Salva 16 bytes na memória

Instruções Escalares e Empacotadas - SSE

- SSE dois tipos de operações
 - Escalar e Empacotada
 - Operação escalar opera apenas no elemento de dados menos significativo
 - Sufixo SS (Single Scalar)
 - O empacotado calcula os elementos em paralelo
 - Sufixo PS

mulss $xmm1, xmm0$	mulps $xmm1, xmm0$
127 95 63 31 0	127 95 63 31 0
XMM0 [4.0 3.0 2.0 1.0 *]	XMM0 [4.0 3.0 2.0 1.0 *]
XMM1 [5.0 5.0 5.0 5.0 =]	XMM1 [5.0 5.0 5.0 5.0 =]
	= = = =
XMM1 [4.0 3.0 2.0 5.0]	XMM1 [20.0 15.0 10.0 5.0]

Exemplo

- ADDPB - Soma um valor (escalar) com 4 reais (float) em um vetor

```
float a[4], b[4], x;
...
for (i = 0; i < 4; i++)
    a[i] = b[i] + x;
```

```
float a[4], b[4], x;
...
_asm {
    mov    ebx, 0
    movss xmm0, x
    shupss xmm0, xmm0, 0
    movups xmm1, b[ebx]
    addps  xmm0, xmm1
    movups a[ebx], xmm0
}
```

Carrega o float x (4 bytes) na parte baixa do xmm0
Repete os primeiros 4 bytes nos outros bytes do registrador

Não há repetição pois a operação ADDPS já soma os quatro elementos do vetor

Instruções Aritméticas SSE Usada em Reais

- Requerem dois operandos

Arithmetic	Scalar Operator	Packed Operator
$y = y + x$	addss	addps
$y = y - x$	subss	subps
$y = y \times x$	mulss	mulps
$y = y \div x$	divss	divps
$y = \frac{1}{x}$	rcpss	rcpps
$y = \sqrt{x}$	sqrts	sqrtps
$y = \frac{1}{\sqrt{x}}$	rsqrts	rsqrtps
$y = \max(y, x)$	maxss	maxps
$y = \min(y, x)$	minss	minps

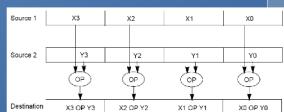
Exemplo

- ADDPB - Soma 4 reais (double) de um vetor com 4 de outro vetor

```
double a[N], b[N], c[N];
...
for (i = 0; i < N; i++)
    a[i] = b[i] + c[i];
```

```
double a[N], b[N], c[N];
...
_asm {
    mov    ebx, 0
    mov    ecx, ND/2
    repete:
    movups xmm0, b[ebx]
    movups xmm1, c[ebx]
    addpd  xmm0, xmm1
    movups a[ebx], xmm0
    add    ebx, 16
    sub    ecx, 1
    jnz   repete
}
```

Carrega 16 bytes em xmm0
Carrega 16 bytes em xmm1
Salva 16 bytes em na memória

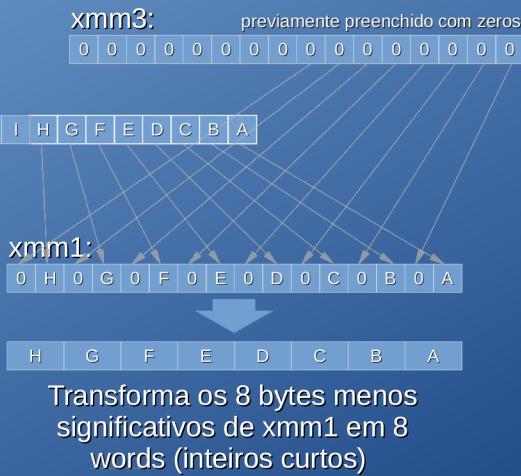


movq xmm1, [eax]

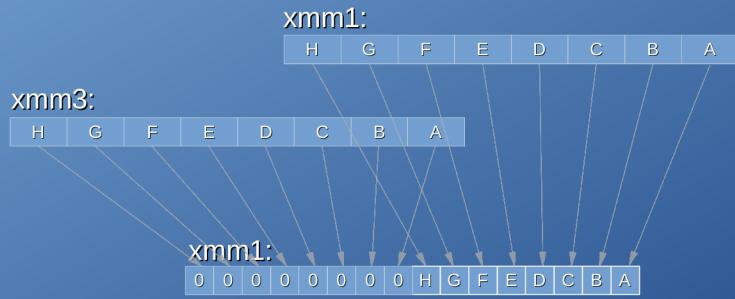


8 bytes são copiados da memória endereçada por eax para o registrador xmm1

punpcklbw xmm1, xmm3



packuswb xmm1, xmm3



Exercícios

- Encontre o menor valor em um vetor de inteiros de 16 bits. O vetor tem 1024 posições.
- Some dois vetores de inteiros de 8 bits sem sinal. Os valores com resultados maiores que 255, devem receber 255.

Intrinsics

- O GCC oferece *Intrinsics* como uma alternativa para a programação em linguagem de montagem de SSE

```
1 #include <iostream>
2
3 #ifdef __SSE2__
4 | #include <emmintrin.h>
5 #else
6 | #warning SSE2 support is not available. Code will not compile
7 #endif
8
9 int main(int argc, char **argv)
10 {
11     __m128 a = _mm_set_ps(4.0, 3.0, 2.0, 1.0);
12     __m128 b = _mm_set_ps(8.0, 7.0, 6.0, 5.0);
13
14     __m128 c = _mm_add_ps(a, b);
15
16     float d[4];
17     _mm_storeu_ps(d, c);
18
19     std::cout << "result equals " << d[0] << "," << d[1]
20     << "," << d[2] << "," << d[3] << std::endl;
21
22     return 0;
23 }
```

Bibliografia

- Intel® 64 and IA-32 Architectures Software Developer's Manual
 - <https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf>
- Intel® Intrinsics Guide
 - <https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html>
- Um resumo SSE (Streaming SIMD Extentions)
 - <http://www.songho.ca/misc/sse/sse.html>