Instruções SIMD

João Canas Ferreira

Abril 2017

U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

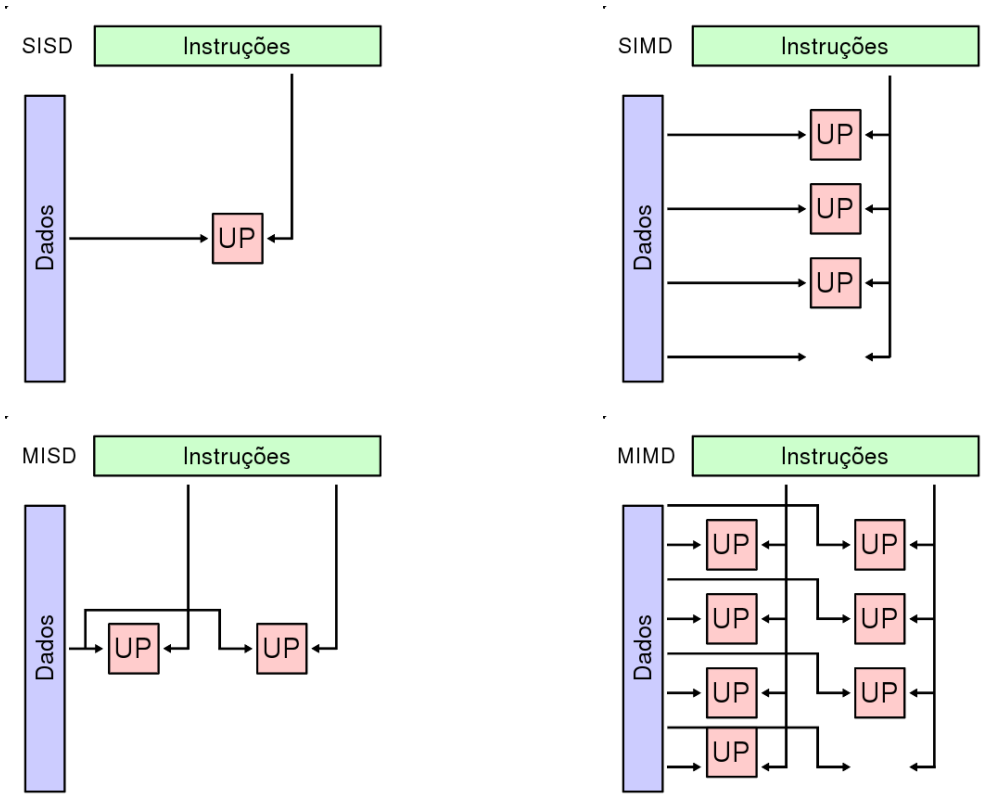
Assuntos

Modelos de execução de instruções

As unidades de processamento podem ser classificadas segundo o número de instruções e de conjuntos de dados tratados em simultâneo:

- **SISD:** *Single instruction stream, single data stream*
  processador convencional: 1 instrução processa 1 conjunto de dados
- **SIMD:** *Single instruction stream, multiple data streams*
  1 instrução processa vários conjuntos de dados (processamento vetorial, também usado em GPUs)
- **MISD:** *Multiple instruction streams, single data stream*
  processamento redundante (pouco usado)
- **MIMD:** *Multiple instruction streams, multiple data streams*
  múltiplas instruções (diferentes) processam múltiplos conjuntos de dados (por exemplo, um processador multi-núcleo)

Representação dos modelos de execução



Adaptado de I, Cburnett, CC BY-SA 3.0, http://wikipedia.org/wiki/Flynn's_taxonomy

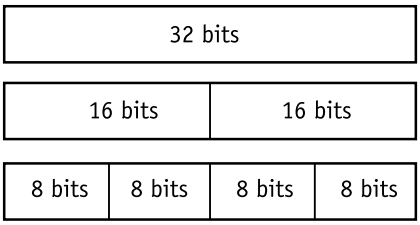
Evolução do suporte para instruções SIMD em IA-32

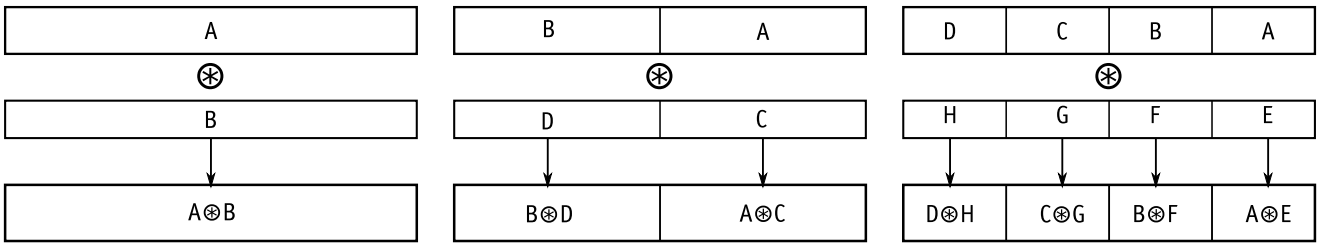
- **MMX** (Multimedia Extensions): introduzido com o processador Pentium II; registos de 64 bits; *apenas valores inteiros*.
- **SSE** (Streaming SIMD Extensions): introduzido com o processador Pentium III; vem substituir as instruções MMX; registos de 128 bits.
- **SSE2**, SSE3/SSE3S e SSE4: SSE2 introduzido com os processadores Pentium 4 e Xeon; versão atual: SSE 4.2
- **AVX/AVX2** (Advanced Vector Extensions): extensão para 256 bits.
- **AVX-512**: extensão para vetores de 512 bits, mais operações e registos.

As instruções SSE também vêm substituir as instruções de vírgula flutuante anteriores (baseadas na unidade de vírgula flutuante x87).

Instruções SIMD: modelo abstrato

➠ Um registo pode ser interpretado como uma unidade ou um vetor com um número fixo (p. ex., 2 ou 4) de registos **independentes**.





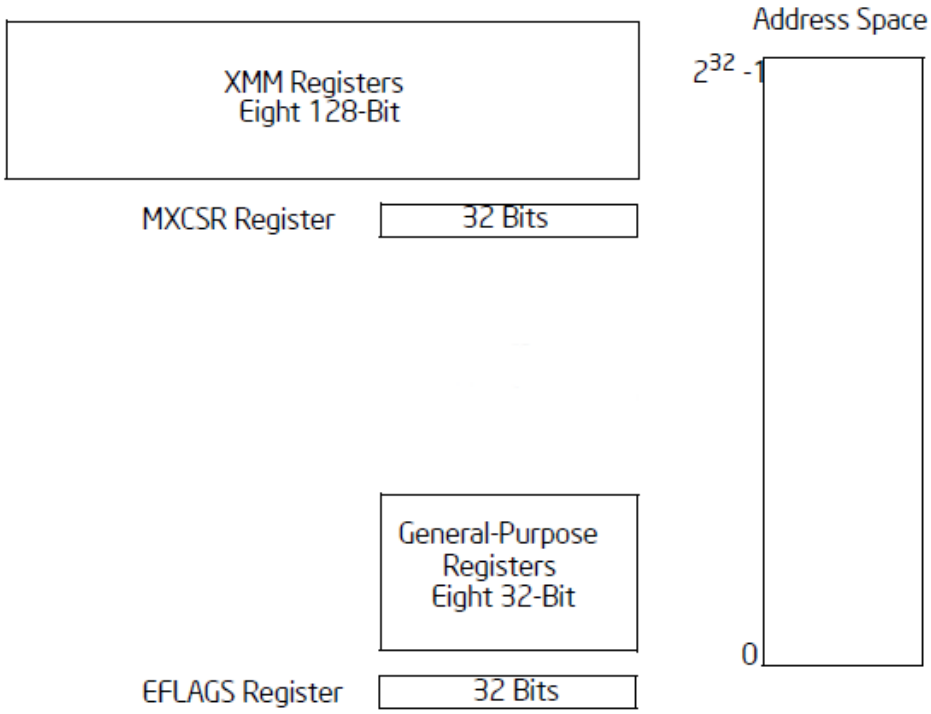
➠ Cada elemento do vetor pode ser combinado com o elemento correspondente de outro vetor com uma *única* instrução.

Instruções SIMD: vantagens e desvantagens

- Vantagens
  - Aumento de desempenho
  - Aproveitamento da capacidade de integração (capacidade de integrar número elevado de ALUs e outras unidades de processamento)
  - Paralelismo explícito é mais fácil de aproveitar (operações são naturalmente independentes)
- Desvantagens
  - Requer algoritmos com processamento de dados "uniforme" (todos os elementos do vetor são processados da mesma forma)
  - Necessidade de adaptar a codificação do algoritmo
  - Alguns compiladores têm dificuldade em aproveitar bem este tipo de instruções: recurso a linguagem *assembly*.

Topics

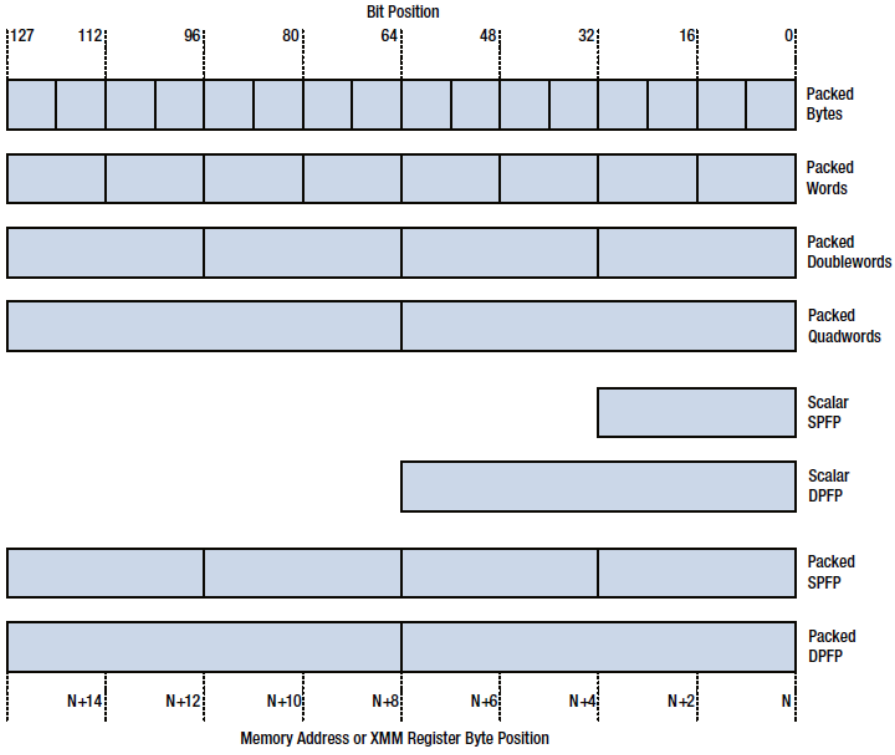
Ambiente de execução SSE



Fonte: [Intel16]

Novos registos

- Oito novos registo de 128 bits: XMM0-XMM7



Fonte: [Kussworm14]

SPFP: single-precision floating point

DPFP: double precision floating point

MXCSR: registo de controlo e *status*

Classes de instruções

➠ Operações para cada tipo de dados (escalares, pacotes de dados em VF, pacotes de dados inteiros):

1. movimentação (transferência) de dados
2. aritmética
3. comparação
4. conversão de formatos de representação
5. operações lógicas
6. mudança de posição no pacote de dados (exceto operandos escalares)
7. outras (controlo do usa da memória *cache*, acesso ao registo de controlo)

➠ Vamos usar a designação genérica SSE para cobrir SSE/SSE2/SSE3/SSE4.

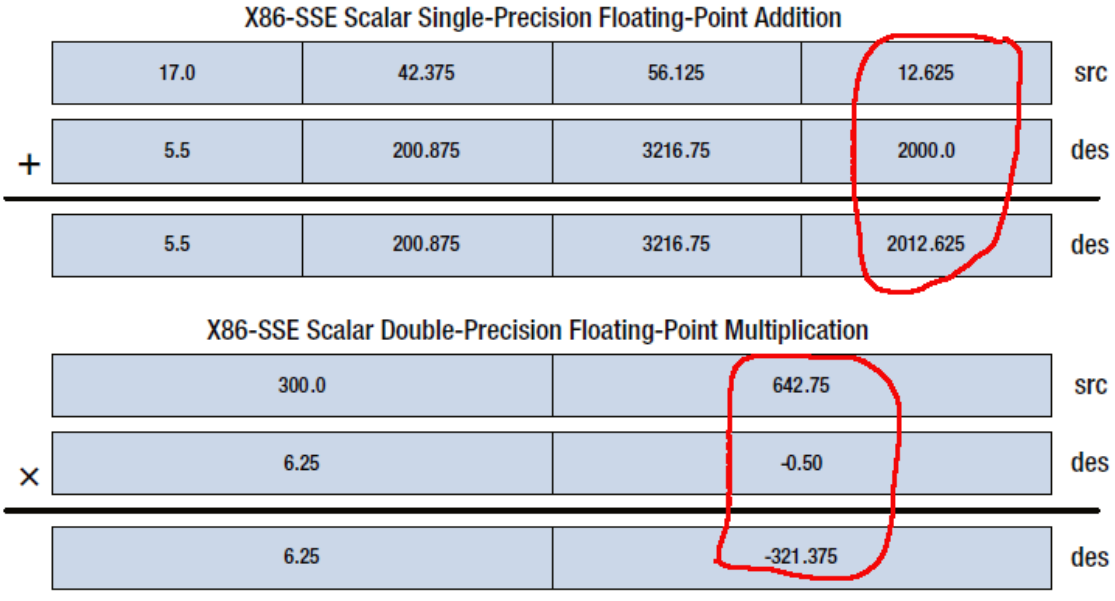
Regras gerais das instruções SSE

- Instruções de 2 operandos; o 1º operando é também o destino.
- Um operando pode provir de um registo XMM ou de memória.
- O resultado fica sempre num registo XMM (exceto para instruções de transferência de dados).
- Dados vetoriais *(packed)* devem estar em posições de memória cujos endereços sejam **múltiplos de 16**.
- a maior parte das mnemónicas para vírgula flutuante termina com:
  - `ps` (packed SPFP)
  - `pd` (packed DPFP)
  - `ss` (scalar SPFP)
  - `sd` (scalar DPFP)
- MASM: Acrescentar a diretiva `.XMM` no início do ficheiro de *assembly*.

Topics

Dados escalares (simples)

➠ As instruções SSE permitem tratar de dados escalares de vírgula flutuante (precisão simples ou dupla).



Fonte: [Kussworm14]

➠ Estas instruções usam apenas a parte menos significativa do registo.

Instruções de transferência de dados

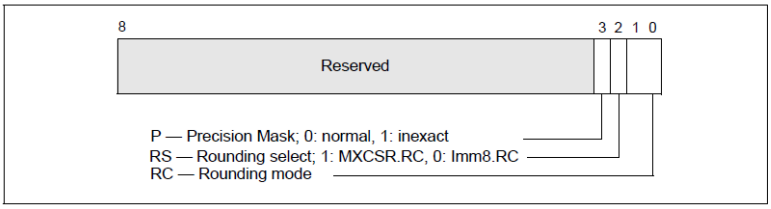
- Estas instruções afetam apenas parte de um registo XMM (como todas as instruções "escalares").
  - precisão simples (tipo `real4`): `movss`
  - precisão dupla (tipo `real8`): `movsd`
- Modo de utilização:
  - `movss xmm1, xmm2`
    Copiar SPFP de registo xmm2 para registo xmm1, sem alterar o restante conteúdo do registo *(merge)*.
  - `movss xmm1, m32`
    Copiar valor de memória (em format IEEE-754) para xmm1 *(load)*
  - `movss mem32, xmm1`
    Copiar SPFP para posição de memória mem32 *(store)*
  - Para dados DPFP (`real8`, 64 bits):
    `movsd xmm1, xmm2` *(merge)*
    `movsd xmm1, mem64` *(load)*
    `movsd mem64, xmm2` *(store)*

Instruções aritméticas para dados VF simples

- Instruções para dados em VF escalares:
  - `addss, addsd`
  - `subss, subsd`
  - `mulss, mulsd`
  - `divss, divsd`
  - `sqrtss, sqrtsd` (raiz quadrada) `sqrtss xmm1, xmm2/m32`
  - `maxss, maxsd`
  - `minss, minsd`
  - `roundss, roundsd` (arredondar) `roundss xmm1,xmm2/m32,imm8`
  - `rcpss` (recíproco) `rcpss xmm1, xmm2/m32`
  - `rsqrtss` (recíproco da raiz quadrada) `rsqrtss xmm1, xmm2/m32`

Modos de arredondamento

➠ Especificados pelo operando imm8 das instruções roundss e roundsd ou por alguns bits do registo MXCSR.



Fonte: [Intel16]

➠ Se P=1 → eventual exceção de precisão não é assinalada.
(Esta exceção ocorre quando o resultado não é exatamente representável.)

Rounding Mode	RC Field Setting	Description
Round to nearest (even)	00B	Rounded result is the closest to the infinitely precise result. If two values are equally close, the result is the even value (i.e., the integer value with the least-significant bit of zero).
Round down (toward −∞)	01B	Rounded result is closest to but no greater than the infinitely precise result.
Round up (toward +∞)	10B	Rounded result is closest to but no less than the infinitely precise result.
Round toward zero (Truncate)	11B	Rounded result is closest to but no greater in absolute value than the infinitely precise result.

Fonte: [Intel16]

Exemplo 1

```
   include mpcp.inc
   .xmm

   .data
varA   REAL8   2.5
varB   REAL8   -1.5
varC   REAL8   ?
msg    BYTE    "Resultado: %f",13,10,0

   .code
main PROC C
       movsd xmm0, varA
       mulsd xmm0, varB
       movsd varC, xmm0

       invoke printf, offset msg, varC
       invoke _getch
       ret
main ENDP
end
```

Exemplo 2 (parte 1)

- Programa em C++ que invoca função que processa dados do tipo `float`

```
#include <iostream>
extern "C" void asmsse(float a, float b,  float c[7]);
using namespace std;
int main()
{
  float a = 3.5f, b = -5.525f;
  float c[7];  // para resultados

  asmsse(a, b, c);  // sub-rotina que realiza cálculos

  cout << "a=" << a << " b=" << b << endl;
  cout << "a+b=" << c[0] << " a-b=" << c[1] << endl;
  cout << "a*b=" << c[2] << " a/b=" << c[3] << endl;
  cout << "min(a,b)=" << c[4] << " max(a,b)=" << c[5] << endl;
  cout << "sqrt(max(a,b))=" << c[6] << endl;
  return 0;
}
```


Exemplo 2 (parte 2)

```
1  asmsse PROC C a:real4, b:real4, varC: ptr real4
2          movss   xmm0, a          ; xmm0 = a
3          movss   xmm1, b          ; xmm1 = b
4          mov     edx, varC        ; endereco base
5          movss   xmm2, xmm0
6          addss   xmm2, xmm1
7          movss   real ptr [edx], xmm2
8          movss   xmm3, xmm0
9          subss   xmm3, xmm1
10         movss   real4 ptr [edx+4], xmm3
11         movss   xmm4,xmm0
12         mulss   xmm4,xmm1        ;xmm4 = a * b
13         movss   real4 ptr [edx+8],xmm4
14         ; ...
15         maxss   xmm7,xmm1        ;xmm7 = max(a, b)
16         movss   real4 ptr [edx+20],xmm7
17         sqrtss  xmm0, xmm7
18         movss   real4 ptr [edx+24],xmm0
19         ret
20 asmsse ENDP
```

Instruções de comparação (1/3)

➠ Instruções de comparação dados em VF escalares:

`cmpss, cmpsd` `cmpss xmm1, xmm2/m32, imm8`

Os bits [2:0] de imm8 especificam o predicado de comparação:

Predicate	imm8 Value	Description	Result: A Is 1st Operand, B Is 2nd Operand			
			A >B	A < B	A = B	Unordered
EQ_OQ (EQ)	0H	Equal (ordered, non-signaling)	False	False	True	False
LT_OS (LT)	1H	Less-than (ordered, signaling)	False	True	False	False
LE_OS (LE)	2H	Less-than-or-equal (ordered, signaling)	False	True	True	False
UNORD_Q (UNORD)	3H	Unordered (non-signaling)	False	False	False	True
NEQ_UQ (NEQ)	4H	Not-equal (unordered, non-signaling)	True	True	False	True
NLT_US (NLT)	5H	Not-less-than (unordered, signaling)	True	False	True	True
NLE_US (NLE)	6H	Not-less-than-or-equal (unordered, signaling)	True	False	False	True
ORD_Q (ORD)	7H	Ordered (non-signaling)	True	True	True	False

Fonte: [Intel16]

Os predicados *unordered* são verdadeiros se pelo menos um dos operandos for NaN.
Os predicados *ordered* só podem ser verdadeiros se nenhum dos operandos for NaN.

Resultado: "máscara" do tamanho dos operandos só com 0s ou só com 1s .

Instruções de comparação (2/3)

As seguintes pseudo-operações evitam o uso do valor imediato:

Pseudo-Op	CMPSS Implementation
CMPEQSS *xmm1, xmm2*	CMPSS *xmm1, xmm2, 0*
CMPLTSS *xmm1, xmm2*	CMPSS *xmm1, xmm2, 1*
CMPLESS *xmm1, xmm2*	CMPSS *xmm1, xmm2, 2*
CMPUNORDSS *xmm1, xmm2*	CMPSS *xmm1, xmm2, 3*
CMPNEQSS *xmm1, xmm2*	CMPSS *xmm1, xmm2, 4*
CMPNLTSS *xmm1, xmm2*	CMPSS *xmm1, xmm2, 5*
CMPNLESS *xmm1, xmm2*	CMPSS *xmm1, xmm2, 6*
CMPORDSS *xmm1, xmm2*	CMPSS *xmm1, xmm2, 7*

Fonte: [Intel16]

➠ Também existem as pseudo-operações correspondentes para DPFP (terminação SD)

Instruções de comparação (3/3)

As seguintes comparações de VF escalares afetam os indicadores ZF, PF e CF:

- `comiss, comisd` — `comiss xmm1, xmm2/m32`
  Comparação "ordenada"de dois valores VF.
- `ucomiss, ucomisd` — `ucomiss xmm1, xmm2/m32`
  Comparação *unordered* entre dois valores VF
  (resultado é *unordered* se algum operando for NaN).

➠ Valores dos indicadores *(flags)* após a execução das instruções:

resultado da comparação	ZF	PF	CF
A>B	0	0	0
A<B	0	0	1
A=B	1	0	0
sem ordem	1	1	1

➠ Estes valores são compatíveis com as operações condicionais para comparações de valores sem sinal (se PF=0).

Exemplo 3 (parte 1)

➠ Programa em C++ que invoca função que compara `floats`.

```
1  #include <iostream>
2  #include <limits>
3  extern "C" void ssecmp(float a, float b, bool *res);
4  using namespace std;
5  const char* Ops[7]={"UO","<","<=","==","!=",">",">="};
6  int main()  {
7      const int n = 4;     bool resultados[7];
8      float a[n] = { 120.0, 250.0, 300.0, 42.0 };
9      float b[n] = { 130.0, 240.0, 300.0, 0.0 };
10     b[n - 1] = numeric_limits<float>::quiet_NaN();
11     for (int i = 0; i < n; i++) {
12        ssecmp(a[i], b[i], results);
13        cout << "a=" << a[i] << "\tb=" << b[i] << endl;
14        for (int j = 0; j < 7; j++)
15          cout << Ops[j] << " ("
16               << (resultados[j]==true?"true)":"false)")
17               << '\t';
18     cout << endl << endl;
19 }
```

Exemplo 3 (parte 2)

```
1  ssecmp PROC C a:real4, b:real4, bptr: ptr byte
2         movss  xmm0, a          ; xmm0 = a
3         mov    edx, bptr        ; endereco base
4         comiss  xmm0, b         ; comparar
5         setp byte ptr [edx]     ; PF = 1 se "unordered"
6         jnp  @F
7         xor al, al
8         mov byte ptr [edx+1],al   ; todos os outros falsos
9        ...
10        jmp fim
11 @@:
12       setb  byte ptr [edx+1]     ; byte=1 se a < b
13       setbe byte ptr [edx+2]     ; byte=1 se a <= b
14       sete  byte ptr [edx+3]     ; byte=1 se a == b
15       setne byte ptr [edx+4]     ; byte=1 se a != b
16       seta  byte ptr [edx+5]     ; byte=1 se a > b
17       setae byte ptr [edx+6]     ; byte=1 se a >= b
18 fim: ret
19 ssecmp ENDP
```


Exemplo 3 (parte 3)

- Resultado da execução do exemplo:

```
A=120  B=130
UO (false)      < (true)        <= (true)       == (false)      != (true)       > (false)       >= (false)

A=250  B=240
UO (false)      < (false)       <= (false)      == (false)      != (true)       > (true)        >= (true)

A=300  B=300
UO (false)      < (false)       <= (true)       == (true)       != (false)      > (false)       >= (true)

A=42   B=nan
UO (true)       < (false)       <= (false)      == (false)      != (false)      > (false)       >= (false)
```

Notar o que sucede quando b é NaN (2 últimas linhas).

Instruções de conversão de formato

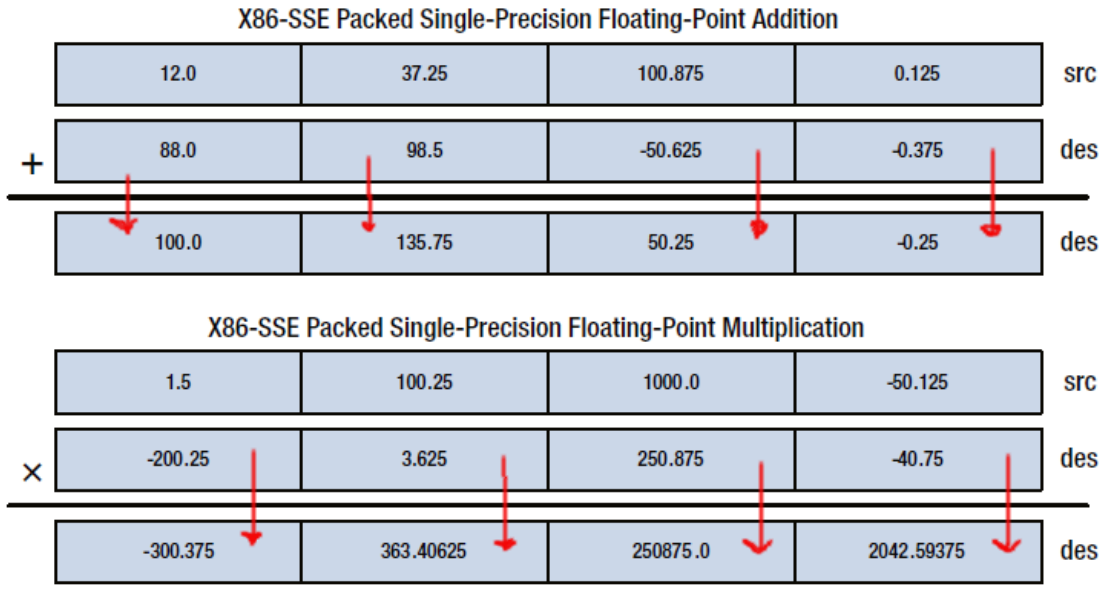
➠ Existem instruções para converter entre representação de números inteiros (complemento para 2) e representação em VF, ou entre formatos VF.

- `cvtsi2ss, cvtsi2sd` — `cvtsi2ss xmm1, r/m32`
  Conversão de SDWORD para SPFP ou DPFP.
- `cvtss2si, cvtsd2si` — `cvtss2si r32, xmm1/m32`
  Conversão de SPFP ou DPFP para SDWORD (com arredondamento)
- `cvttss2si, cvttsd2si` — `cvttss2si r32, xmm1/m32`
  Conversão de SPFP ou DPFP para SDWORD (com truncatura)
- `cvtss2sd` — `cvtss2sd xmm1, xmm2/m32`
  Conversão de SPFP para DPFP.
- `cvtsd2ss` — `cvtsd2ss xmm1, xmm2/m32`
  Conversão de DPFP para SPFP.

Topics

Princípios gerais (1/2)

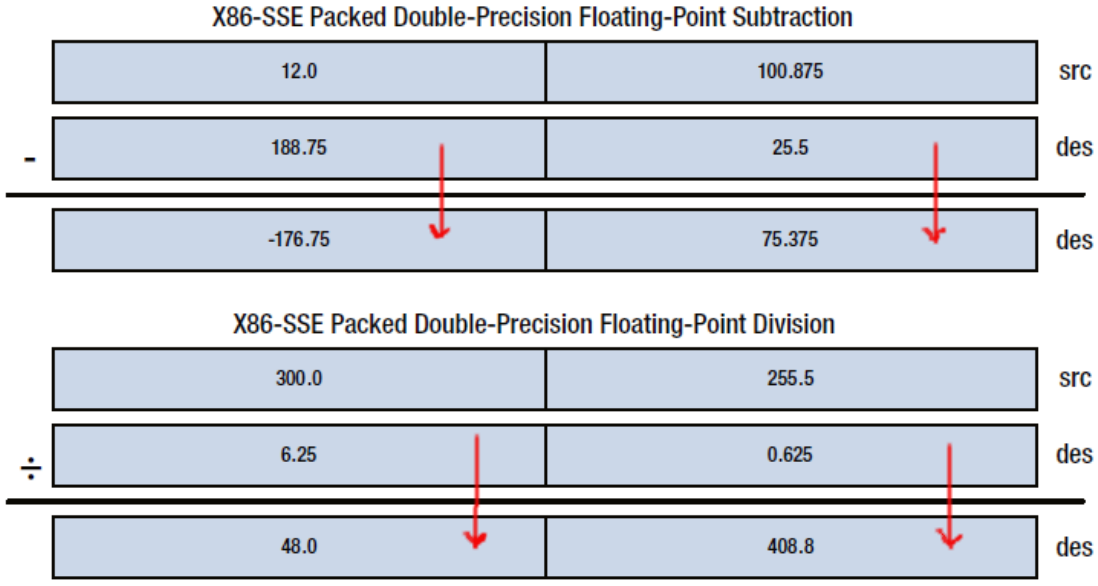
- Quatro números de VF (precisão simples) por operando
- Uma instrução



Fonte: [Kussworm14]

Princípios gerais (1/2)

- Dois números de VF (precisão dupla) por operando
- Uma instrução



Fonte: [Kussworm14]

- Importante: Dados empacotados devem ter endereços múltiplos de 16.

Instruções de transferência de dados empacotados

➠ As versões que trabalham com SPFP terminam em ps (packed, single); as que trabalham com DPFP terminam em pd (packed, double).

- `movaps, movapd` Copiar valores alinhados
  Ex: `movaps xmm1, xmm2/m128` ou `movaps xmm2/m128, xmm1`
- `movups, movupd` Copiar valores não-alinhados
  Ex: `movups xmm1, xmm2/m128` ou `movups xmm2/m128, xmm1`
- `movhps, movhpd`
  Copiar QWORD mais significativa sem alterar a outra parte do registo
  Ex: `movhps xmm1, m64` ou `movhps m64, xmm1`
- `movlhps, movhlps`
  Copiar QWORD menos significativa para posição mais significativa
  Ex: `movlhps xmm1, xmm2` ou `movhlps xmm1, xmm2`
- `movmskps, movmskpd` Extrair os bits de sinal e de cada operando SPFP/DPFP e guardá-los nos bits menos significativos de um registo de uso geral (restantes bits colocados a 0)
  Ex: `movmskps reg32,xmm` (4 bits) ou `movmskpd reg32,xmm` (2 bits)

Instruções aritméticas para dados VF empacotados

➠ Operações aritméticas com dados VF empacotados:

- `addps, addpd`
- `subps, subpd`
- `mulps, mulpd`
- `divps, divpd`
- `sqrtps, sqrtpd` (raiz quadrada) `sqrtps xmm1, xmm2/m32`
- `maxps, maxpd`
- `minps, minpd`
- `roundps, roundpd` (arredondar) `roundps xmm1,xmm2/m32,imm8`
- `rcpps` (recíproco) `rcpps xmm1, xmm2/m32`
- `rsqrtps` (recíproco da raiz quadrada) `rsqrtps xmm1, xmm2/m32`

Exemplo 4 (parte 1)

```
1  sse_pa PROC  varA: ptr dqword, varB:ptr dqword, varPtr: dqword
2      mov     edx, varA
3      mov     ecx, varB
4      mov     eax, varPtr
5      movaps  xmm0, [edx]
6      movaps  xmm1, [ecx]
7
8      movaps  xmm2,xmm0
9      addps  xmm2,xmm1          ; somar
10     movaps  [eax+0],xmm2      ; guradar no vetor de resultados
11     movaps  xmm2,xmm0
12     subps  xmm2,xmm1          ; subtrair
13     movaps  [eax+16],xmm2
14     ...
15     maxps  xmm0,xmm1          ; máximo
16     movaps  [eax+96],xmm0
17     ret
18 sse_pa ENDP
```


Exemplo 4 (parte 2)

- A impressão dos resultados da sub-rotina produz:

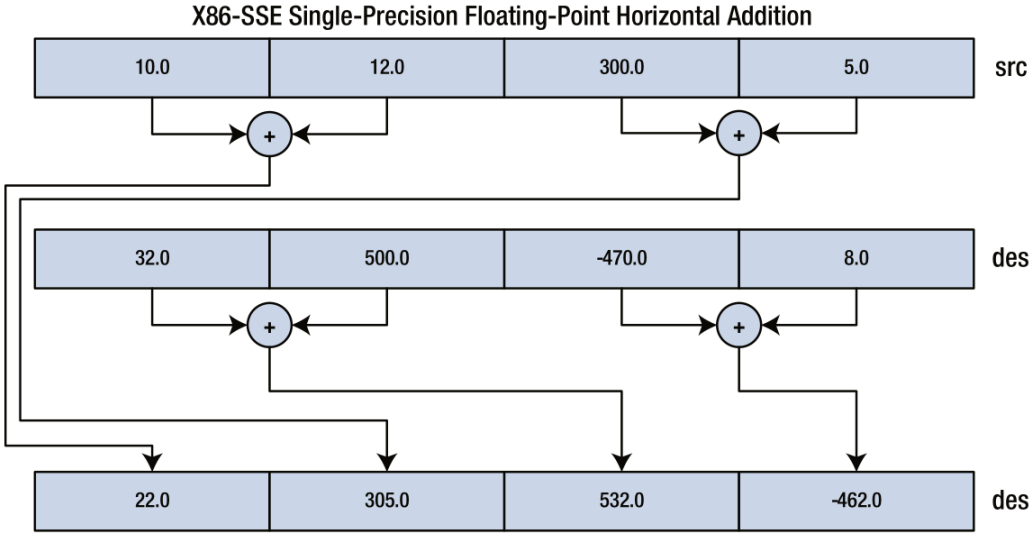
```
A:          36.000000     0.031250 |      2.000000    42.000000
B:          -0.111111    64.000000 |     -0.062500     8.666667

addps       35.888889    64.031250 |      1.937500    50.666668
subps       36.111111   -63.968750 |      2.062500    33.333332
mulps       -4.000000     2.000000 |     -0.125000   364.000000
divps     -324.000000     0.000488 |    -32.000000     4.846154
sqrtps       6.000000     0.176777 |      1.414214     6.480741
minps       -0.111111     0.031250 |     -0.062500     8.666667
maxps       36.000000    64.000000 |      2.000000    42.000000
```

Operações de adição/subtração horizontal (1)

- Há instruções para operar sobre pares de números do mesmo registo.

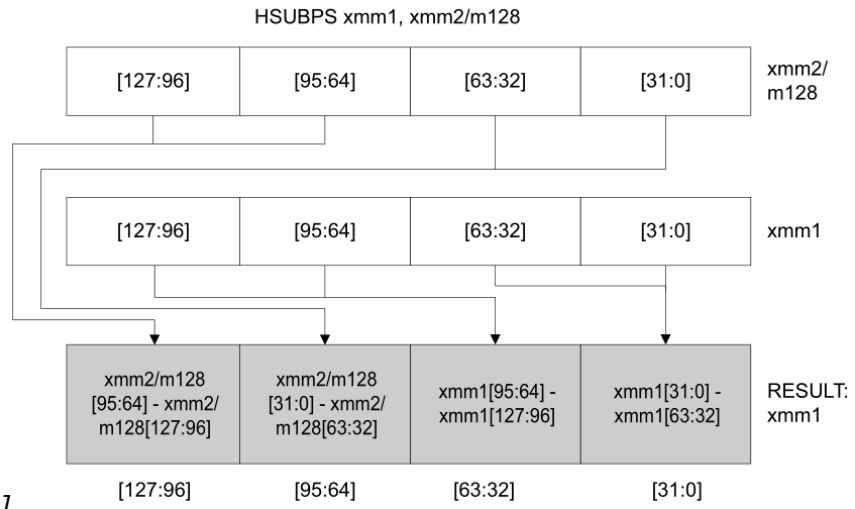
O esquema geral de funcionamento é o seguinte (instrução haddps):



Fonte: [Kussworm14]

Operações de adição/subtração horizontal (2)

- Instruções para operações "horizontais"sobre dados empacotados:
  - `haddps, haddpd` somar pares de valores adjacentes.
    Ex: `haddps xmm1, xmm2/m128` (ver slide anterior)
  - `hsubps, hsubpd` subtrair pares de valores adjacentes
    Ex: `hsubps xmm1, xmm2/m128`

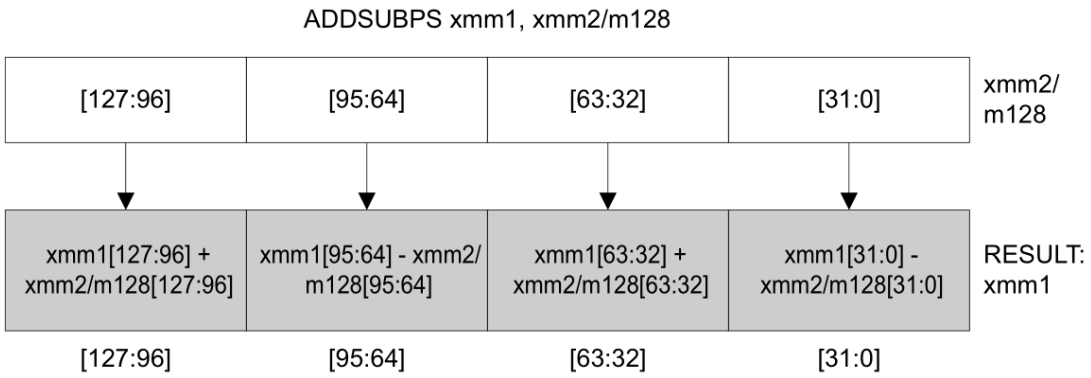


Fonte: [Intel16]

Operações de adição/subtração horizontal (3)

- Também é possível combinar adições e subtrações:
  - `addsubps, addsubpd` Subtrair ou somar valores adjacentes.

    Ex: `addsubps xmm1, xmm2/m128`



Fonte: [Intel16]

- A escolha de quais os elementos usados nas subtrações e quais são usados nas adições e da posição dos resultados é fixa.

Instruções de comparação de dados VF empacotados (1)

- Instruções de comparação dados em VF empacotados:

`cmpps, cmppd` `cmpps xmm1, xmm2/m128, imm8`

Os bits [2:0] de `imm8` especificam o predicado de comparação (da mesma forma que para valores escalares):

Predicate	imm8 Value	Description	Result: A Is 1st Operand, B Is 2nd Operand			
			A >B	A < B	A = B	Unordered
EQ_OQ (EQ)	0H	Equal (ordered, non-signaling)	False	False	True	False
LT_OS (LT)	1H	Less-than (ordered, signaling)	False	True	False	False
LE_OS (LE)	2H	Less-than-or-equal (ordered, signaling)	False	True	True	False
UNORD_Q (UNORD)	3H	Unordered (non-signaling)	False	False	False	True
NEQ_UQ (NEQ)	4H	Not-equal (unordered, non-signaling)	True	True	False	True
NLT_US (NLT)	5H	Not-less-than (unordered, signaling)	True	False	True	True
NLE_US (NLE)	6H	Not-less-than-or-equal (unordered, signaling)	True	False	False	True
ORD_Q (ORD)	7H	Ordered (non-signaling)	True	True	True	False

Fonte: [Intel16]

Os predicados *unordered* são verdadeiros se pelo menos um dos operandos for NaN.
Os predicados *ordered* só podem ser verdadeiros se nenhum dos operandos for NaN.

Resultado: "máscara" de 0s ou de 1s do tamanho dos operandos.

Instruções de comparação de dados VF empacotados (2)

➠ As instruções de comparação dados em VF empacotados também têm mnemónicas individuais.

Para dados de precisão simples são:

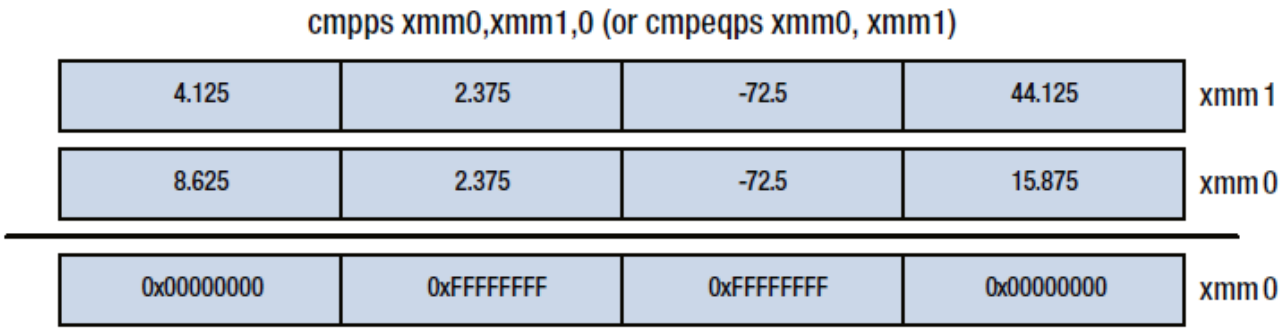
Pseudo-Op	CMPPS Implementation
CMPEQPS *xmm1, xmm2*	CMPPS *xmm1, xmm2, 0*
CMPLTPS *xmm1, xmm2*	CMPPS *xmm1, xmm2, 1*
CMPLEPS *xmm1, xmm2*	CMPPS *xmm1, xmm2, 2*
CMPUNORDPS *xmm1, xmm2*	CMPPS *xmm1, xmm2, 3*
CMPNEQPS *xmm1, xmm2*	CMPPS *xmm1, xmm2, 4*
CMPNLTPS *xmm1, xmm2*	CMPPS *xmm1, xmm2, 5*
CMPNLEPS *xmm1, xmm2*	CMPPS *xmm1, xmm2, 6*
CMPORDPS *xmm1, xmm2*	CMPPS *xmm1, xmm2, 7*

Fonte: [Intel16]

➠ As versões para precisão dupla terminam com "d" em vez de "s".

Comparação simultânea de vários valores

➠ Exemplo do princípio de funcionamento da comparação de dados empacotados



Fonte: [Kussworm14]

Exemplo5 (parte 1)

```
1  sse_pf_cmp PROC ptrA:ptr dqword, ptrB:ptr dqword,
2                  varPtr:ptr dqword
3      mov       eax,ptrA
4      mov       ecx,ptrB
5      mov       edx,varPtr
6      movaps    xmm0,[eax]    ;carregar A para xmm0
7      movaps    xmm1,[ecx]    ;carrregar B para xmm1
8      movaps    xmm2,xmm0      ; igualdade
9      cmpeqps   xmm2,xmm1
10     movdqa    [edx],xmm2     ; copiar para vetor de resultados
11     movaps    xmm2,xmm0      ; menor que
12     cmpltps   xmm2,xmm1
13     ; ...
14     movaps    xmm2,xmm0      ; ordenáveis?
15     cmpordps  xmm2,xmm1
16     movdqa    [edx+112],xmm2
17     ret
18  sse_pf_cmp ENDP
```


Exemplo 5 (parte 2)

```
A:      2.000000     7.000000 |    -6.000000     3.000000
B:      1.000000    12.000000 |    -6.000000     8.000000

        == 00000000 00000000 | FFFFFFFF 00000000
        <= 00000000 FFFFFFFF | 00000000 FFFFFFFF
         < 00000000 FFFFFFFF | FFFFFFFF FFFFFFFF
 sem ordem 00000000 00000000 | 00000000 00000000
        != FFFFFFFF FFFFFFFF | 00000000 FFFFFFFF
       ! < FFFFFFFF 00000000 | FFFFFFFF 00000000
     ! <= FFFFFFFF 00000000 | 00000000 00000000
ordenaveis FFFFFFFF FFFFFFFF | FFFFFFFF FFFFFFFF

Resultados (iteracao 1)
A:          nan     7.000000 |    -6.000000     3.000000
B:      1.000000    12.000000 |    -6.000000     8.000000

        == 00000000 00000000 | FFFFFFFF 00000000
        <= 00000000 FFFFFFFF | 00000000 FFFFFFFF
         < 00000000 FFFFFFFF | FFFFFFFF FFFFFFFF
 sem ordem FFFFFFFF 00000000 | 00000000 00000000
        != FFFFFFFF FFFFFFFF | 00000000 FFFFFFFF
       ! < FFFFFFFF 00000000 | FFFFFFFF 00000000
     ! <= FFFFFFFF 00000000 | 00000000 00000000
ordenaveis 00000000 FFFFFFFF | FFFFFFFF FFFFFFFF
```

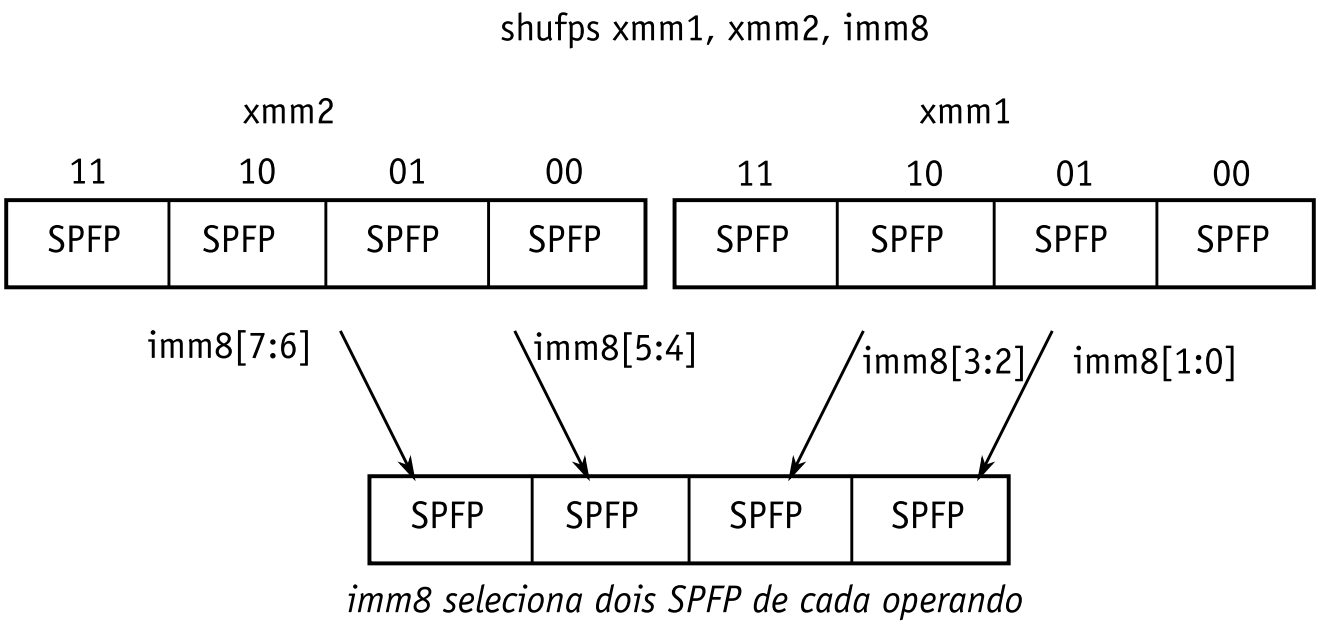
Instruções de conversão para dados VF empacotados

- `cvtpi2ps, cvtpi2pd` `cvtpi2ps xmm, m64`
  Converte um par de SDWORDS para SPFP ou DPSP empacotados (bits não usados ficam a 0)
- `cvtdq2ps, cvtdq2pd` `cvtdq2ps xmm1, xmm2/m128`
  Converte 4 SDWORDs (empacotadas) para 4 FPFP ou 4 DPFP (DQWORD: double quadword)
- `cvtps2dq, cvtpd2dq` `cvtps2dq xmm1, xmm2/m128`
  Converte 4 FPFP ou 2 DPFP para 4 (ou 2) SDWORDs empacotadas; para `cvtpd2dq`, XMM1[127:64]=0.
- `cvttps2dq, cvttpd2dq` `cvttps2dq xmm1, xmm2/m128`
  Como as anteriores, mas usam truncatura para arredondamento
- `cvtps2pd` `cvtps2pd xmm1, xmm2/m64`
  Converte 2 SPFP (bits menos significativos de XMM) para 2 DPFP
- `cvtpd2ps` `cvtpd2ps xmm1, xmm2/m128`
  Converte 2 DPFP para 2 SPFP; no destino, XMM1[127:64]=0.

Rearranjos de dados VF empacotados (1)

- `shufps, shufpd` `shufps xmm1, xmm2/m128, imm8`

Copia elementos dos dois operandos para o destino de acordo com a especificação `imm8`: 2 valores de cada um (`shufps`) ou apenas um valor de cada (`shufpd`).

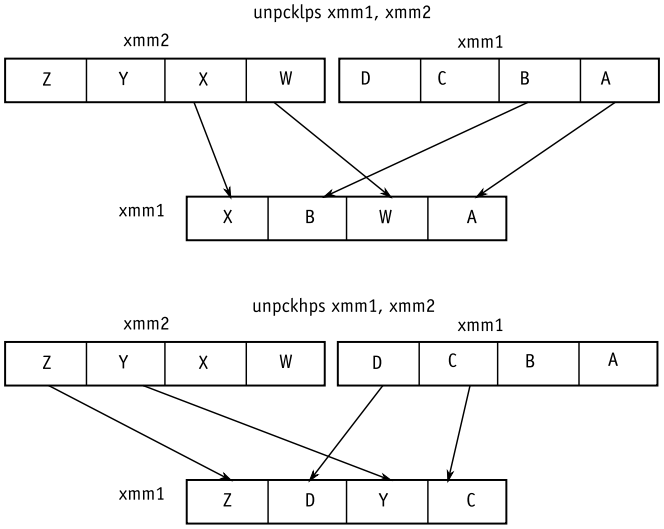


imm8 seleciona dois SPFP de cada operando

Rearranjos de dados VF empacotados (2)

➠ `unpcklps, unpcklpd` `unpcklps xmm1, xmm2/m128`

Intercala os elementos menos significativos dos dois operandos.



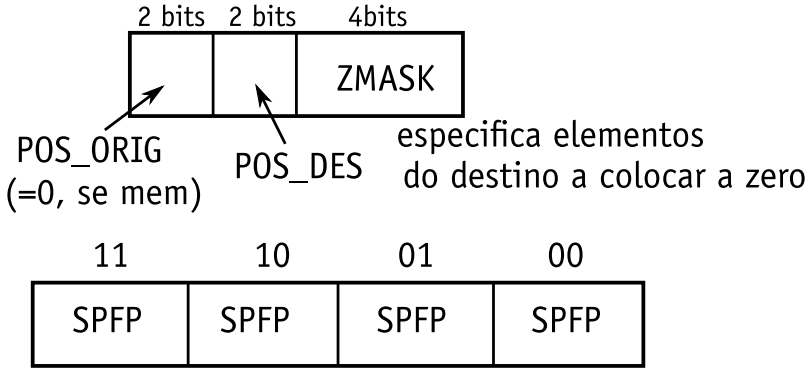
➠ `unpckhps, unpckhpd` `unpckhps xmm1, xmm2/m128`

Intercala os elementos mais significativos dos dois operandos.

Rearranjos de dados VF empacotados (3)

- `insertps` `insertps xmm1,xmm2/m32,imm8`

  Copiar um SPFP para uma dada posição de um registo XMM e, opcionalmente, colocar as outras a zero.



- `extractps` `extractps reg/m32,xmm1,imm8`

  Copiar um SPFP de uma dada posição de um registo XMM (especificada por `imm8`) para memória ou registo de uso geral.

Operações lógicas com dados VF empacotados

- `andps, andpd` `andps xmm1, xmm2/m128`

  AND bit-a-bit dos elementos correspondentes de cada operando.

- `andnps, andnpd` `andnps xmm1, xmm2/m128`

  Negação do registo de destino seguida de operação AND bit-a-bit dos elementos correspondentes de cada operando.

- `orps, orpd` `orps xmm1, xmm2/m128`

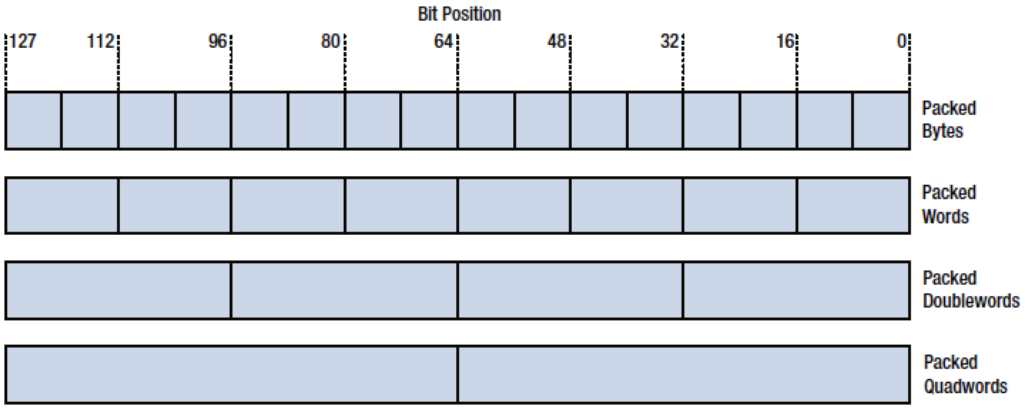
  OR bit-a-bit dos elementos correspondentes de cada operando.

- `xorps, xorpd` `xorps xmm1, xmm2/m128`

  XOR bit-a-bit dos elementos correspondentes de cada operando.

Topics

Transferências de valores inteiros empacotados

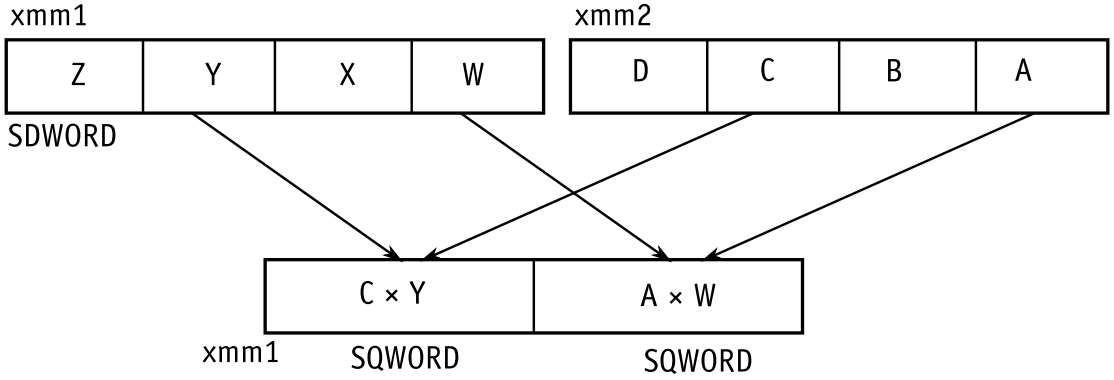


Fonte: [Kussworm14]

- `movdqa` `movdqa xmm1, xmm2/m128` `movdqa xmm1/m128, xmm2`
  Copiar grupos de 128 bits (`m128` é múltiplo de 16)
- `movdqu`
  Como a anterior, mas `mem128` não necessita de estar alinhado (acesso muito demorado)

Aritmética com valores inteiros empacotados (1)

- `paddb, paddw, paddd, paddq` `paddb xmm1, xmm2/m128`
- `psubb, psubw, psubd, psubq` `psubb xmm1, xmm2/m128`
- `pmulhw, pmullw` `pmulhw xmm1, xmm2/m128`
  Multiplicar SWORDS e colocar bits mais (menos, no caso de `pmullw`) significativos no destino.
- `pmuldq` `pmuldq xmm1, xmm2/m128`
  Multiplicar SDWORDs das posições ímpares.



Aritmética com valores inteiros empacotados (2)

- `pminub, pminuw, pminud` `pminud xmm1, xmm2/m128`
  Mínimo de valores sem sinal.
- `pminsb, pminsw, pminsd`
  Mínimo de valores com sinal.
- `pmaxub, pmaxuw, pmaxud`
  Valores máximos de números sem sinal.
- `pmaxsb, pmaxsw, pmaxsd`
  Valores mínimos de números com sinal.
- `pabsb, pabsw, pabsd` `pabsb xmm1, xmm2/m128`
  Valor absoluto.
- `psignb, psignw, psignd` `psignb xmm1, xmm2/m128`
  Preserva, anula, ou calcula o simétrico dos valores de xmm1 conforme o sinal dos valores de xmm2.

Aritmética com saturação

➠ As operações seguintes permitem adicionar e subtrair valores "com saturação": o resultado nunca sai dos limites da representação.

- `paddsb, paddsw` `paddsb xmm1, xmm2/m128`
  Soma com saturação de números com sinal.
- `paddusb, paddusw` `paddusb xmm1, xmm2/m128`
  Soma com saturação de números sem sinal.
- `psubsb, psubsw` `psubsb xmm1, xmm2/m128`
  Subtração com saturação de números com sinal.
- `psubusb, psubusw` `psubusb xmm1, xmm2/m128`
  Subtração com saturação de números sem sinal.

Operações lógicas com valores inteiros empacotados

- `pand` `pand xmm1, xmm2/m128`
  Operação lógica AND bit a bit
- `pandn` `pandn xmm1, xmm2/m128`
  Operação lógica $a \leftarrow \overline{a} \cdot b$ realizada bit a bit
- `por, pxor` `por xmm1, xmm2/m128`
  Operações lógicas XOR e XNOR bit a bit.
- `psllw, pslld, psllq, pslldq` `psllw xmm1, imm8`
  Deslocamento "lógico"para a esquerda (de `imm8` bits) de cada componente.
- `pslrw, pslrd, pslrq, pslrdq` `psrlw xmm1, imm8`
  Deslocamento "lógico"para a direita (de `imm8` bits) de cada componente.
- `psaaw, psrad, psraq` `psraw xmm1, imm8`
  Deslocamento "aritmético"para a direita (de `imm8` bits) de cada componente.

Comparação de valores inteiros empacotados

➠ As instruções de comparações produzem valores "000...0"ou "111...1"consoante o resultado da comparação é falso ou verdadeiro.

- `pcmpeqb, pcmpeqw, pcmpeqd, pcmpeqq`
  `pcmeqb xmm1, xmm2/m128`
  Comparação de igualdade.
- `pcmpgtb, pcmpgtw, pcmgtd, pcmgtq`
  `pcmpgtb xmm1, xmm2/m128`
  Comparação "maior que"para valores com sinal.

Referências

Esta apresentação inclui figuras de:

Kusswurm14 D. Kusswurm, "Modern Assembly Language Programming: 32-bit, 64-bit, SSE, and AVX", 2014, Apress

Intel16 Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual", dezembro de 2016, order number: 325462-061US (disponível on-line)