

---

# Extensions SIMD dans les microprocesseurs

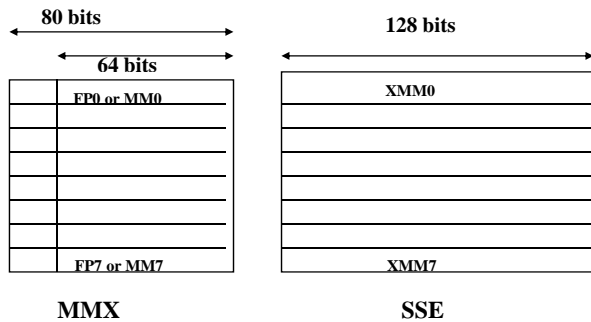
Daniel Etiemble  
de@lri.fr

---

## Extensions SIMD dans les jeux d'instructions

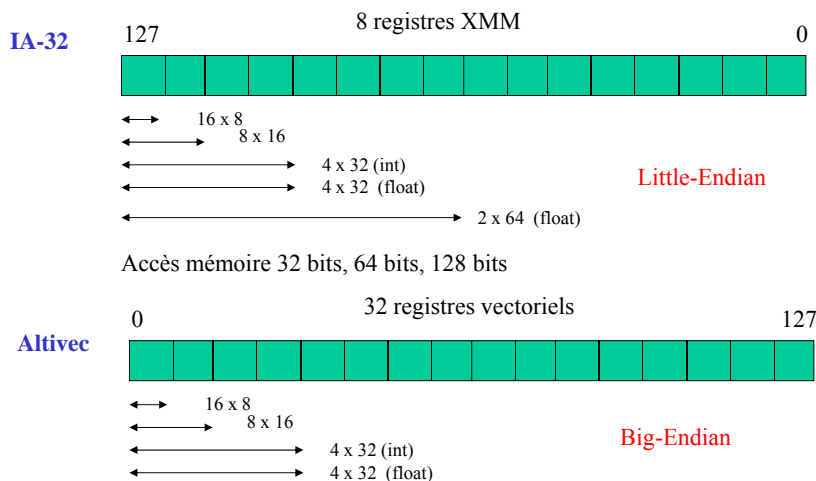
- Dans tous les jeux d'instructions
  - HP, SPARC VIS, MIPS, PowerPC AltiVec
  - IA-32 (Intel MMX, SSE, SSE2, SSE3, AMD 3D Now)
- Applications
  - Audio, Communication, Noyaux DSP, Graphique 2D et 3D, Images, Vidéo, Reconnaissance parole, etc
- Formes limitées d'instructions vectorielles
  - Vecteurs courts (2, 4, 8, 16 éléments)
  - Accès mémoire avec pas unitaire
    - Pas de scatter-gather, pas d'accès avec pas non unitaire
  - Transfert registres, registres mémoire et opérations sont SIMD (tous les éléments du vecteur simultanément)
  - Instructions "spéciales" multimédia
    - Ex : Somme de valeurs absolues de différences

## Les registres SIMD (IA-32)



- Les registres MMX sont partagés avec les registres flottants (pile x87)
- Les registres XMM sont distincts

## Formats de données SIMD ()

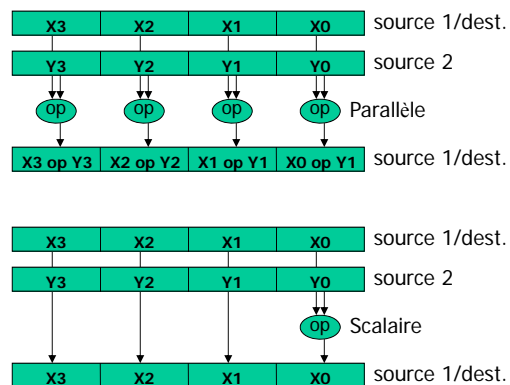


## Les registres SIMD (Intel 64)

- Intel AVX
  - 16 registres 256 bits Ymm0 à Ymm15
  - Formats
    - SSE = AVX 128 bits
    - 8 floats
    - 4 doubles
  - Implanté dans « Sandy Bridge » (32 nm)

## Instructions SIMD parallèles et scalaires

- Instructions arithmétiques et logiques
- Instructions mémoire
- Instructions de formatage et manipulation
- Instructions de conversion



## SIMD IA-32 : opérations arithmétiques

Arithmétique entière	Complément à 2	Saturation signée	Saturation non signée
Addition	PADD(h,w,d)	PADDs(h,w)	PADDUS(h,w)
Soustraction	PSUB(b,w,d)	PSUBS(b,w)	PSUBUS(b,w)
Multiplication	PMUL(lw,lhw)		
Multiplication accumulation	PMADDWD		

Arithmétique flottante	Parallèle SP	Scalaire SP	Parallèle DP	Scalaire DP
Addition	ADDPS	ADDSS	ADDPD	ADDSD
Soustraction	SUBPS	SUBSS	SUBPD	SUBSD
Multiplication	MULPS	MULSS	MULPD	MULSD
Division	DIVPS	DIVSS	DIVPD	DIVSD
Racine carrée	SQRTPS	SQRTSS	SQRTPD	SQRTSD

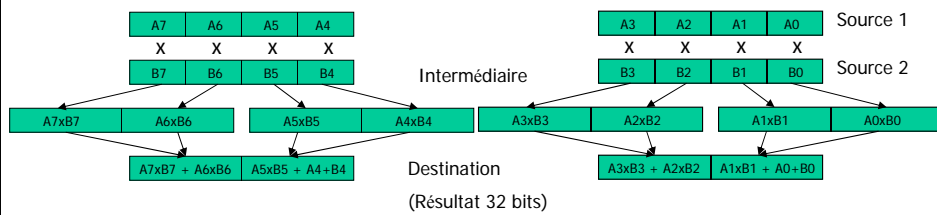
Polytech Info4 - 2012

Architecture des ordinateurs  
Daniel Etiemble

7

## Multiplication - accumulation entière

- 8 multiplications et 4 additions en une instruction PMADDWD



- PMADDWD produit 2 résultats 32 bits
  - Utile pour les applications multimédia et traitement du signal
  - Formats d'entrée et de sortie différents
  - N'existe pas pour les données 8 bits en entrée

Polytech Info4 - 2012

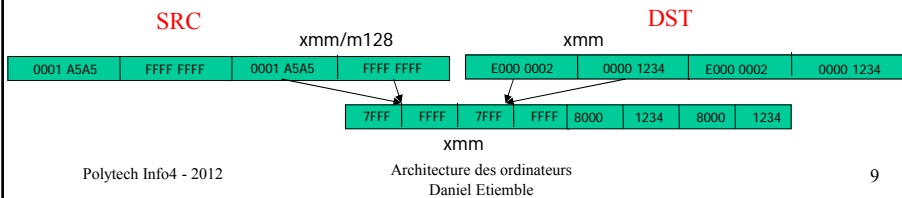
Architecture des ordinateurs  
Daniel Etiemble

8

## Compactage - décompactage

Arithmétique entière	Compl. à 2	Saturation signée	Saturation non signée
Pack		PACKSS(wb,dw)	PACKUS(wb)
Unpack High	PUNPCKH(bw,wd,dq)		
Unpack Low	PUNPCKL(bw,wd,dq)		

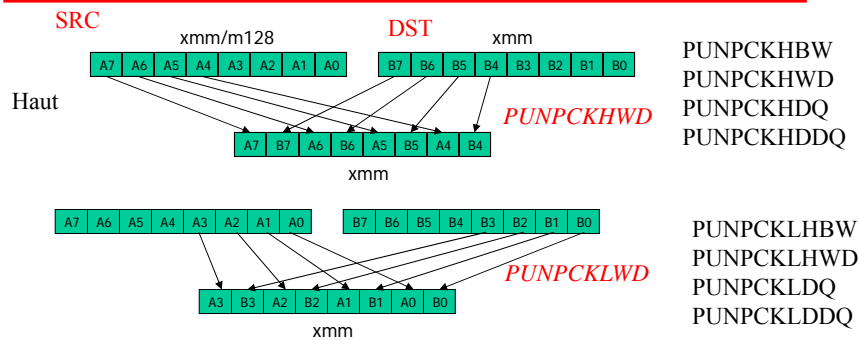
- **PACKSSDW** – Compacte les données 32 bits en données 16 bits avec saturation signée (plus grand/plus petit si débordement par défaut ou par excès). Utile pour les données 16 bits avec calcul sur précision 32 bits.



Polytech Info4 - 2012

9

## Décompactage



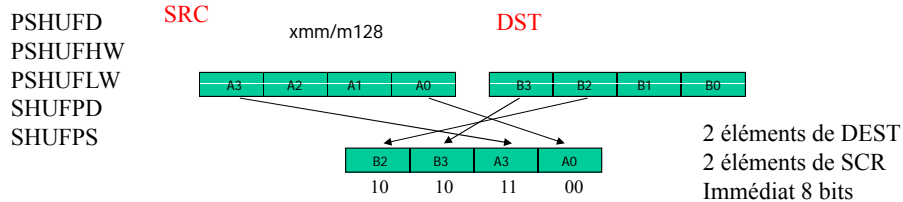
- Utile pour convertir des données, rassembler des données, entrelacer/dupliquer des données, transposer des lignes et des colonnes

Polytech Info4 - 2012

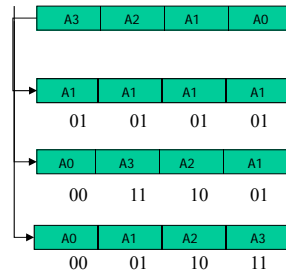
Architecture des ordinateurs  
Daniel Etiemble

10

## Les instructions “shuffle”



- Diffusion
  - shufps xmm1, xmm1, 55H
- Rotation
  - shufps xmm1, xmm1, 39H
- Swap
  - shufps xmm1, xmm1, 1BH



## Transferts mémoire

- Transferts flottants
  - Transferts entre registres XMMi et Mémoire
    - movaps xmm1, [eax]
    - movaps [edi], xmm2
  - Transferts alignés ou non
    - Aligné 4 mots : movaps
    - Aligné 2 mots haut : movhps
    - Aligné 2 mots bas : movlps
    - Non aligné : movups
  - Transfert scalaire
    - movss : charge mot bas du registre, et met à 0 les autres
  - Transfert entre partie haute ou basse de registres
    - movhlps et movlhps
- Transferts entiers
  - Transfert entre mémoire et registres XMM

## Alignement mémoire

- L'accès à un mot de 128 bits est aligné sur une frontière de 16 octets. Les quatre bits de poids faible de l'adresse sont 0000 pour un accès aligné
- IA-32
  - Accès alignés
  - Accès non alignés (plus lents)
- AltiVec
  - Les accès mémoire sont obligatoirement alignés
  - Utilisation de plusieurs instructions pour les accès non alignés

## Comparaisons et opérations logiques

- Compare (eq, lt, le, unord, neq, nlt, nle, ord) and set mask
  - Flottants
    - cmpxxps ou cmpxxss
    - cmpxxpd ou cmpxxsd
  - Entiers
    - PCOMPEQ (B,W,D)
    - PCOMPGT (B,W,D)
- Opérateurs logiques
  - and, andn, or, xor
    - versions ps et ss
    - Version entière

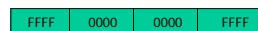
SRC



DST



DST < SRC



## Valeurs absolues

- Si  $x(i) < 0$  alors  $|x(i)| = -x(i)$
- Il n'y a pas d'instruction SIMD de calcul de la valeur absolue
- Calcul des valeurs absolues du contenu d'un registre SIMD
  - `__m128i v1;`
  - Créer une constante SIMD zero
  - Calculer `zero - v1`
  - Calculer `max (v1, zero - v1)`

## Conditionnelles SIMD

Exemple

```
unsigned char X[512], max;  
max = X[0];  
for (i=1 ; i<512 ; i++)  
    if (X[i] >max) max=X[i] ;
```

Programme Scalaire

```
unsigned char X[512], max, byte[16];  
__m128i XS[32], M , a;  
XS=X ;  
M= ld16(&XS[0]) ;  
for(i=0; i<32; i++) {  
    a = ld16(&XS[i]);  
    M = maxbu(M,a);  
}  
  
byte=&M ;  
max = byte[0]  
for (i=1 ; i<16 ; i++)  
    if (byte[i] >max) max=byte[i];
```

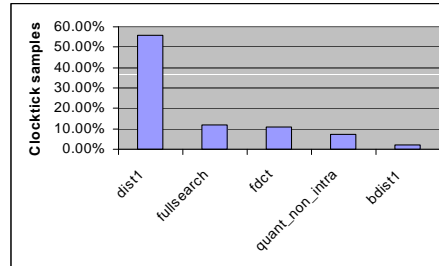
Suppression des  
branchements  
conditionnels

Version SIMD



## Instructions spéciales (ex : PSABW)

- PSABW
  - Valeur absolue des différences des octets (unsigned char) dans DST et SRC
  - Somme des valeurs absolues pour les huit octets bas dans les 32 bits de la partie basse de DST
  - Somme des valeurs absolues pour les huit octets hauts dans les 32 bits de la partie haute de DST



Encodeur MPEG2

$$SAE = \sum_{i=0}^{i=7} \sum_{j=0}^{j=7} |C_{ij} - R_{ij}|$$

0000 RES2 0000 RES1

## Impact de PSABW

Code C pour l'estimation de mouvement

```

for(l=0; l<nVERT; l++)
  for(k=0, c=0; c<nHORZ; k+=8, c++){
    answer = 0;
    for(j=0; j<16; j++)
      for(i=0; i<16; i++)
        answer += abs(x[l+j][k+i] - y[l+j][k+i]);
    result[l][c] = answer;}
    
```

Version C naïve : 271 CPP (cycles par pixel)

Version XMM : 13,5 CPP

Accélération : **20**

## Type d'utilisation des instructions SIMD

---

- C (ou Fortran)
  - Transformation du code pour rendre les boucles vectorisables
- Intrinsics
  - Appel de fonctions de type C
  - Le compilateur traite l'allocation des registres et l'ordonnancement
- Langage assembleur
  - Assembleur avec instructions SIMD dans le code C

### Exemples d'intrinsics

```
__m128 _mm_set_ps1(float f)
__m128 _mm_load_ps(float *mem)
__m128 _mm_mul_ps(__m128 x, __m128 y)
__m128 _mm_add_ps (__m128 x, __m128 y)
void _mm_store_ps(float *mem, __m128 x)
```

## « Vectorisation automatique »

---

- Conditions de « vectorisation »
  - Accès à pas unitaire
  - Pas de dépendances de données
  - Pas de pointeurs
- Performance des caches
  - Accès à pas unitaire

## Eviter les pointeurs

- Pointeurs et incrémentation/décrémentation de pointeurs n'est pas autorisé pour indexer les boucles

```
int a[100];
int *p;
p=a;

for (i=0; i<100;i++)
    *p++ = i;
```

Ne vectorise pas

```
int a[100];

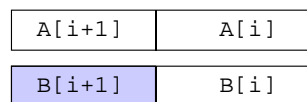
for (i=0; i<100;i++)
    a[i] = i;
```

VECTORISE

## Dépendances propagées entre itérations

```
S1: A[i]=A[i]+ B[i];
S2: B[i+1]= C[i]+ D[i]
```

Pas de  
vectorisation



Pas encore disponible

```
S2: B[i+1]= C[i]+ D[i]
S1*: A[i+1]=A[i+1]+ B[i+1];
```

VECTORISATION

## Problèmes de l'arithmétique entière

- Problème spécifique aux instructions SIMD entières
  - Addition :  $N$  bits +  $N$  bits donnent  $N+1$  bits
    - → soit arithmétique saturée
    - → soit complément à 2 avec PERTE DE LA RETENUE
  - Multiplication :  $N * N$  donnent  $2N$  bits
- Instructions spéciales de multiplication ou multiplication - accumulation
  - Multiplication  $N*N$  et et résultat sur  $2N$  bits (avec éventuellement accumulation)
    - IA-32
      - PMADDWD :  $16 \times 16 + 32$
      - PMULUDQ :  $32 \times 32 \Rightarrow 64$
    - Altivec:
      - plusieurs variantes  $16 \times 16 + 32$
      - Plusieurs variantes de multiplications  $8 \times 8 \Rightarrow 16$

## Exemple : l'inversion d'images

```
void inversion(byte **X, long i0, long i1, long j0, long j1, byte **Y)
{ int i, j; for(i=i0; i<=i1; i++)
  { for(j=j0; j<=j1; j++)
    { Y[i][j] = 255 - X[i][j]; } } }
```

```
void inversionS(byte **X, long i0, long i1, long j0, long j1, byte **Y)
{ int i, j, m, n;
  __m128i **XS, **YS;
  __m128i constante, l1, l2;
  XS=X; YS=Y;
  constante=_mm_set1_epi8 (-1);
  for(i=i0; i<=i1; i++) {
    for(j=j0; j<=j1/16-1; j++) {
      __mm_store_si128(&YS[i][j], _mm_subs_epu8(constante,XS[i][j] ));}} }
```

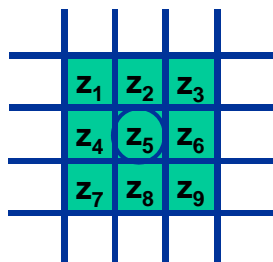
## Problèmes d'alignement

- Variables allouées statiquement
  - `__declspec( align(16) ) V1, V2, V3;`
- Variables allouées dynamiquement
  - `#include malloc.h`
  - Fonctions `_mm_malloc` et `_mm_free`

```
byte** bmatrix(long nrl, long nrh, long ncl, long nch)
{long i, nrow=nrh-nrl+1, ncol=nch-ncl+1;
 byte **m;
 /* allocate pointers to rows */

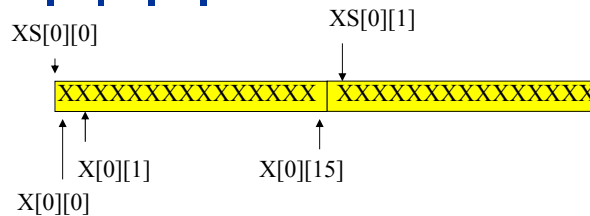
 m=(byte **) _mm_malloc((size_t)((nrow+NR_END)*sizeof(byte*)), 16);
 if (!m) nerror("allocation failure 1 in bmatrix()");
 m += NR_END;
 m -= nrl;
 /* allocate rows and set pointers to them */
 .....
 return m;
```

## Accès aux pixels voisins

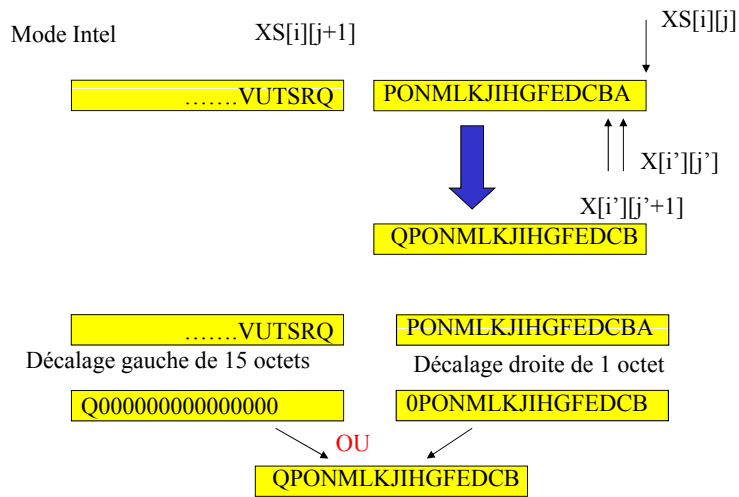


```
byte **X[i][j];
__m128i **XS [i][j];
XS = X;
```

Accès classique ≠ accès Intel



## Accès SIMD aux pixels voisins



Polytech Info4 - 2012

Architecture des ordinateurs  
Daniel Etiemble

27

## Accès aux pixels voisins

```
#define decg(va,vb) _mm_or_si128(_mm_srli_si128(va,1),_mm_slli_si128(vb,15))
#define decd(va,vb) _mm_or_si128(_mm_slli_si128(va,1),_mm_srli_si128(vb,15))
```

EXEMPLE

```
aij= _mm_load_si128(&XS[i][j]);
```

```
aijp = decg(_mm_load_si128(&XS[i][j]),_mm_load_si128(&XS[i][j+1]));
```

```
aijm= decd(_mm_load_si128(&XS[i][j]),_mm_load_si128(&XS[i][j-1]));
```

Polytech Info4 - 2012

Architecture des ordinateurs  
Daniel Etiemble

28

## Un exemple avec « produit scalaire »

### Calculs flottants

```
void postfiltfn (//.....//)
{float *basis, *ppm, *ppm_end;
for(ppm=premult+bshift,y=0; y<sr; ppm+=ppm_yinc, ++y) {
    for (ppm_end = ppm+((tmp3<sr)?tmp3:sr);
        ppm < ppm_end; ++ppm, --tmp3){
        ip = 0;
        for (k=0; k<n; k++){
            ip+= basis[k]*ppm[k];.....}
        // two similar "middle" loops}
```

### OPTIMISATIONS

- Déroulage de la boucle interne
  - Calculer plus vite chaque produit scalaire
- Déroulage de la boucle milieu
  - Calculer quatre produits scalaires simultanément

## Instructions SIMD et produit scalaire

```
for (ppm_end = ppm+((tmp3<sr)?tmp3:sr);
    ppm < ppm_end; ++ppm, --tmp3){
    ip[0] = 0.0; ip[1] = 0.0; ip[2] = 0.0; ip[3] = 0.0;
    for (k=0; k<n/4; k+=4){
        ip[0]+= basis[k]*ppm[k];
        ip[1]+= basis[k+1]*ppm[k+1];
        ip[2]+= basis[k+2]*ppm[k+2];
        ip[3]+= basis[k+3]*ppm[k+3]; }
    ip= ip[0]+ip[1]+ip[2]+ip[3];
```

```
for(ppm_end=ppm+((tmp3<sr)?tmp3:sr);
    ppm<ppm_end-4; ppm+=4, tmp3-=4){
    .....
    ip0 = 0; ip1 = 0; ip2 = 0; ip3 = 0;
    for (k=0; k<n; k++){
        ip0+= basis[k]*ppm[k];
        ip1+= basis[k]*ppm[k+1];
        ip2+= basis[k]*ppm[k+2];
        ip3+= basis[k]*ppm[k+3]; } ...}
```

basis 

k+3	k+2	k+1	k
-----	-----	-----	---

ppm 

k+3	k+2	k+1	k
-----	-----	-----	---

ip[3]	ip[2]	ip[1]	ip[0]
-------	-------	-------	-------

ip

⊕  
Problème d'alignement  
Somme finale

basis 

k	k	k	k
---	---	---	---

ppm 

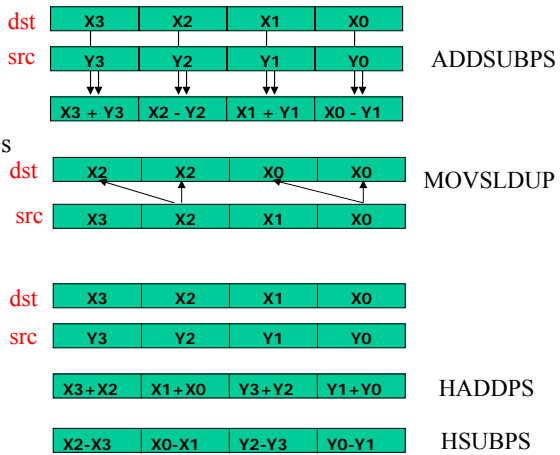
k+3	k+2	k+1	k
-----	-----	-----	---

ip3	ip2	ip1	ip0
-----	-----	-----	-----

Pas de problème  
d'alignement  
Pas de somme finale

## Extensions SIMD SSE3 (IA-32)

- Instructions pour la FFT
  - ADDSUBPD
  - ADDSUBPS
- Duplication de constantes
  - MOVSLLDUP
  - MOVSHDUP
  - MOVSLDUP
- Instructions horizontales
  - HADDPD
  - HADDPS
  - HSUBPD
  - HSUBPS
- Accès non alignés
  - LDDQU



## Réduction des éléments d'un vecteur

- Instruction HADDPS
    - dst: X3, X2, X1, X0
    - src: Y3, Y2, Y1, Y0
    - dst: X3+X2, X1+X0, Y3+Y2, Y1+Y0
- 
- XMM0: D, C, B, A
- HADDPS XMM0, XMM0: C+D, A+B, C+D, A+B
- HADDPS XMM0, XMM0: A+B+C+D, A+B+C+D, A+B+C+D, A+B+C+D



## Registres MIC (Xeon Phi)

---

- 32 registres 512 bits (Zmm0 à Zmm31)
  - 16 int32
  - 8 int64
  - 16 float
  - 8 double
- 8 registres de masque vectoriel k0 à k15 (16 bits)
  - 1 bit par élément du registre  $Zmm_i$  ( 16 ou 8 bits)
    - Contrôle des opérations « élément par élément »
    - Contrôle des écritures registre « élément par élément »
  - Exception
    - k0 correspond au « non masquage »

## Instructions vectorielles

---

- Format général (3,1)
  - $Zmm_i \leftarrow Zmm_j \text{ OP } S (Zmm_k, m)$ 
    - $Zmm_i$  : Registre destination
    - $Zmm_j$  : Registre source 1
    - $Zmm_k$  : Registre source 2
    - $m$  : opérande mémoire
    - $S$  : Fonction de conversion – Swizzle
    - $OP$  : Opération

## Addition vectorielle sur des float32

---

vaddps zmm1 {k1}, zmm2, S.(zmm3/m)

Memory Up-conversion:  $S_{f32}$

$Zmm_i \leftarrow Zmm_j$ , OP Opérandes mémoire

$S_2S_1S_0$	Function:	Usage	disp8*N
000	no conversion	[rax] {16to16} or [rax]	64
001	broadcast 1 element (x16)	[rax] {1to16}	4
010	broadcast 4 elements (x4)	[rax] {4to16}	16
011	float16 to float32	[rax] {float16}	32
100	uint8 to float32	[rax] {uint8}	16
110	uint16 to float32	[rax] {uint16}	32
111	sint16 to float32	[rax] {sint16}	32