Master 1 Informatique IGAI et SIAME

Christophe Collet

- Principes et Technologies des Architectures Spécialisées

Parallélisme de données : SIMD

Introduction : *les architectures*

- *machine* = des opérateurs traitant des données

  opérateurs simultanément actifs → machines parallèles

- Classification de Flynn *(1972)* :

  contrôle des traitements (*flot d'instructions*) ⏩ flots de données

Flot d'instructions \ Flot de données	Unique	Multiple
Unique	**SISD** (Von Neumann)	**SIMD** (tab de processeurs)
Multiple	**MISD** (pipeline)	**MIMD** (multi-processeurs)

Introduction : *les architectures*

- *machines parallèles* :
  - **SIMD** *(Single Instruction stream, Multiple Data stream)*

    1 unité de contrôle

    plrs unités de traitement synchronisées ⏩ plrs flots de données
    - séquenceur unique
    - tableau de processeurs

Introduction : *les architectures*

- *machines parallèles* :
  - **SIMD** : parallélisme de données
    - données stockées en mémoire, manipulées sur place
    - parallélisme important → 1 unité de traitement / donnée
    - traitements bas niveau :
      - opérations sur de combinaisons de données voisines (par ex. filtrage)
      - mouvements de données réguliers, simples et systématiques *(quelles que soient leurs valeurs)*
    - bonne programmation → adaptation des algorithmes séquentiels
    - coût très élevé mais bon rapport performance/coût pour une utilisation intensive *(par ex. images)*

Parallélisme de données : *programmation*

- algorithme ⇒ modèle de programmation :
  - abstraire primitives de traitement et mouvement de données
  - selon les mouvements de données privilégiés de l'architecture de la machine
    - bonnes performances

      ⇒ mécanismes architecturaux efficaces

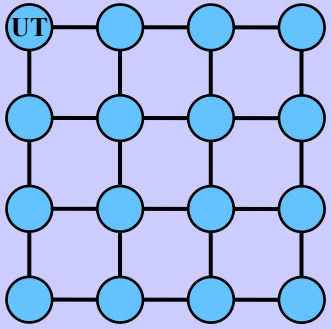
      ⇒ suffisamment généraux pour la programmation des applications

Parallélisme de données : *architectures* SIMD

- ensemble de processeurs synchronisés et communiquant suivant une certaine topologie
- primitives de calcul :
  - une opération sur l'ensemble des processeurs
  - échange une information de chaque processeur vers un voisin dans le graphe de communication

Parallélisme de données : SIMD *tableaux*

- Les machines SIMD tableaux
  - plus anciennes machines spécialisées en Traitement d'Images
  - associe chaque pixel à un processeur simplifié
  - transformations à appliquer : régulières et systématiques
  - topologie en tableau :



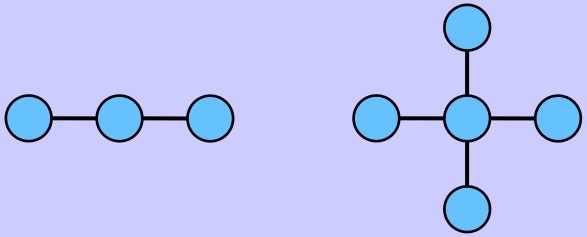
Grille ou tableau ou maille
de processeurs
(plus courante)

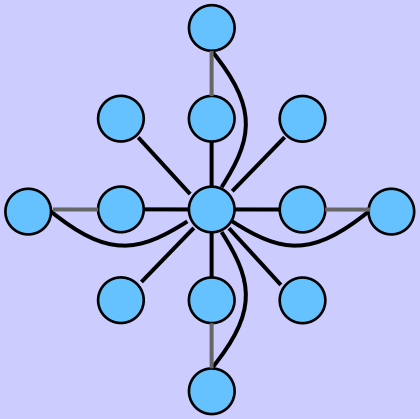
Parallélisme de données : SIMD *tableaux*

- constitué de :
  - unités de traitement :
    - calculs logiques et arithmétiques simples avec des mécanismes d'accélérations des algorithmes complexes :
      - Et, Ou, Xor, Non
      - addition, soustraction, multiplication, division...
      - calculs entiers, flottants
      - racine carrée, multiplication-accumulation (MAC)...
      - comparaisons rapides
      - arithmétique des entiers débordante ou saturée

Parallélisme de données : SIMD *tableaux*

- mémoire locale :
  - stocke un ou plrs ensembles de données et des résultats intermédiaires
  - de qlq dizaines de bits à qlq Ko par processeurs
- liens de communication avec les voisins :
  - pour combiner des valeurs entre données proches dans le tableau *(par ex. pixels voisins)*
  - 2, 4 ou 8 connexité, voire plus





Parallélisme de données : SIMD *tableaux*

- liens de communication globale :
  - transfert de paramètres aux des unités de traitement
  - centralise une info venant des unités de traitement
- entrées-sorties parallèles :
  - potentiel de traitement considérable
  - vitesse adaptée en rapport au temps de traitement pour éviter de brider la machine
  - en générale par décalage par lignes ou colonnes du tableau de données

Parallélisme de données : SIMD *tableaux*

- exemple d'algorithme parallèle : *Convolution*
  - pour chaque pixel *i,j* l'image *y* résultat de la convolution de *x* par le noyau *k* :

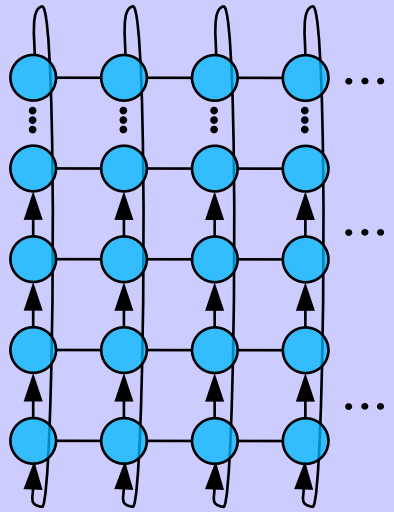
$$y_{i,j} = \sum_{-1 \leqslant u \leqslant 1} \sum_{-1 \leqslant v \leqslant 1} k_{uv} \times x_{i-u,\, j-v}$$

  - les connexions internes autorisant un accès en 8-connexité
  - accès à chaque voisin, multiplication, accumulation du résultat
  - si noyau > 3x3 : décalages successifs des voisins.
  - idem opérations de morphologie mathématique élémentaires (érosion, dilatation)

Parallélisme de données : SIMD *tableaux*

- exemple d'algorithme parallèle : *Histogramme*
  - mauvais adéquation de l'architecture aux opérations globales
  - soit un tableau de *N*x*N* processeurs et une image *I (i,j)* avec *G* niveaux de gris. *N* = *G* pour simplifier. *C* compteur de l'histogramme :

pour $k \leftarrow 1$ à N
pour tous les *(i,j)* en parallèle
si $I(i,j) = G(i,j)$
$C(i,j) \leftarrow C(i,j) + 1$
décaler *G(i,j)* et *C(i,j)* vers le nord suivant un tore
► chaque processeur *(i,j)* contient un *C(i,j)*
accumuler les compteurs *C(i,j)* par ligne



Parallélisme de données : SIMD *tableaux*

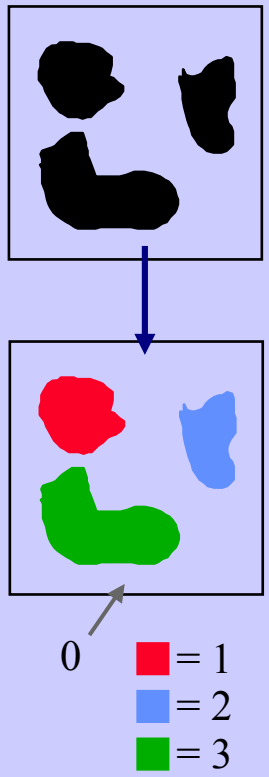
- exemple d'algorithme parallèle :

  Étiquetage de composantes connexes

  - pixels de l'image numérotés de 0 à *N*x*N*

    étiquette ← indice du processeur
    nouvelle ← étiquette
    répéter
        pour tout les voisins *v* du pixel
            si niveau de gris (*v*) = 1
                nouvelle ← max (nouvelle, étiquette (*v*))
            finsi
        finpour
    tant que nouvelle ≠ étiquette

  - complexité : *O*(*N*x*N*)

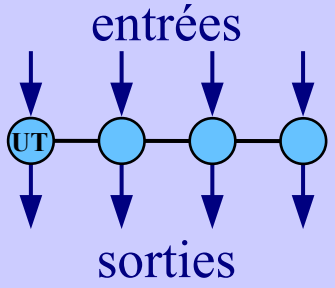


Parallélisme de données : *machines lignes*

- variantes de machines tableaux :
  - chaque processeur s'occupe d'une région
    - 1 ou plrs lignes ou colonnes de données
    - économie de processeurs
  - permet des processeurs plus puissant en fonctionnalités : entiers sur 16, 32 bits... flottants...
  - mémoire contenant une ou plrs lignes/colonnes de données
  - communications améliorées :
    - par mots de 8 ou 16 bits
    - distance > voisinage direct, voire + longue par routage de données

Parallélisme de données : *machines lignes*

- topologie des machines lignes :



- algorithmes de traitement d'image similaires aux machines tableaux
  - gestion plus fine des données : une partie en mémoire locale, l'autre dans les processeurs voisins

  ⇒ peut induire des dissymétries importantes dans les algorithmes
- excellent compromis performances / coût
- complexité matérielle comparable aux machines *pipeline* avec la programmabilité en plus

⇒ solution plus générale et ouverte pour le TI

Parallélisme de données : processeurs *multimedia*

- application des techniques de calcul SIMD en ligne
- amélioration des microprocesseurs :

  extensions multimedia

- besoins pour applications multimedia : image, son, vidéo...
  - exemples :
    - décodage d'images comprimées par blocs (cosinus, filtrage...)
    - zooms (interpolation)
    - correction chromatique gamma (transformées non linéaires/pixel)
    - estimation de mouvement en vidéo (mise en correspondance de blocs)

⇒ introduction d'un mécanisme SIMD dans les µproc

Parallélisme de données : processeurs *multimedia*

- registres 64 ou 128 bits → 8 ou 16 pixels de 8 bits/mot

  données pacquées (plrs données dans un seul mot)
- unité arithmétique et logique : 8 opérations en //

  ⇔ 8 processeurs d'une machine ligne
- instructions de conversion : données pacquées ↔ non
- instructions de communication entre pixels voisins
- masquage de données simule l'inhibition de « processeurs »

Parallélisme de données : processeurs *multimedia*

- la plupart des µproc en possèdent :
  - HP : MAX (*Media ACCelerator*), 1994
  - SUN : VIS (*Visual Instruction Set*), 1995
  - Intel : MMX (*MultiMedia eXtention*), 1996
    - SSE (*Streaming SIMD extensions*), 1999
    - AVX (*Advanced Vector Extensions*), 2010
  - AMD : 3Dnow!, 1998
  - Digital : MVI (*MotionVideo Instructions*), 1998
  - Motorola : AltiVec, 1999 (*VelocityEngine*, Apple)
  - ARM : DSP, NEON (128bits), SIMD (32bits)

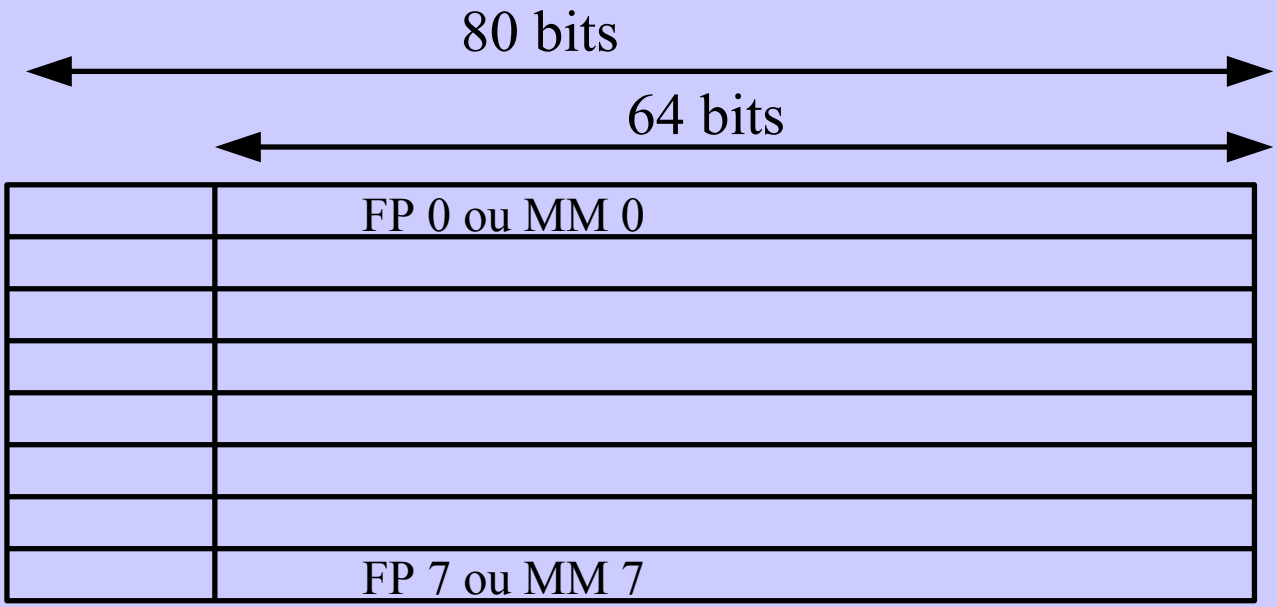
≠ par leur type de données (8/16/32/64 bits, flottant), par les instructions plus ou moins spécialisées

Parallélisme de données : exemple le *MMX*

- études d'un large panel d'applications multimedia : analyse des procédures les plus gourmandes en calculs
- dont caractéristiques communes :
  - données de type entiers courts (pixels sur 8 bits, échantillons de son sur 16 bits...) ;
  - boucles courtes extrêmement répétitives ;
  - multiplications et accumulations fréquentes (MAC : multiplications + additions des résultats) ;
  - algorithmes de calculs intensifs ;
  - opérations extrêmement parallèles

Parallélisme de données : exemple le *MMX*

- ensemble de 57 instructions de base sur les entiers, exploitable facilement dans les applications multimedia et de communication.
- exploite les registres flottant 64 bits *(compatibilité !)*



- bp de passage de portion de programme de calculs flottants en calculs MMX : sauvegardes des registres.

Parallélisme de données : exemple le *MMX*

- Les types de données MMX : entiers signés et non-signés regroupés (*packed*) par paquets de 64bits et sauvegardés dans les registres MMX 64 bits
- quatre types différents :
  - *Packed byte* (Octets pacqués) : 8 octets pacqués dans 64 bits
  - *Packed word* (Mots pacqués) : 4 mots de 16 bits pacqués dans 64 bits ;
  - *Packed doubleword* (Double-mots pacqués) : 2 mots de 32 bits pacqués dans 64 bits ;
  - *Quadword* (Quadruple-mot) : un mot de 64 bits.

Parallélisme de données : exemple le *MMX*

- Instructions:
  - opérations arithmétiques de base : addition, soustraction, multiplication, décalage arithmétique et multiplication-addition;
  - opérations de comparaisons
  - conversions entre types de données : paqueter ou dépaqueter des données;
  - opérations logiques;
  - opérations de décalages;
  - transfert de données (MOV) entre registres MMX ou vers la mémoire sur 64 ou 32 bits.

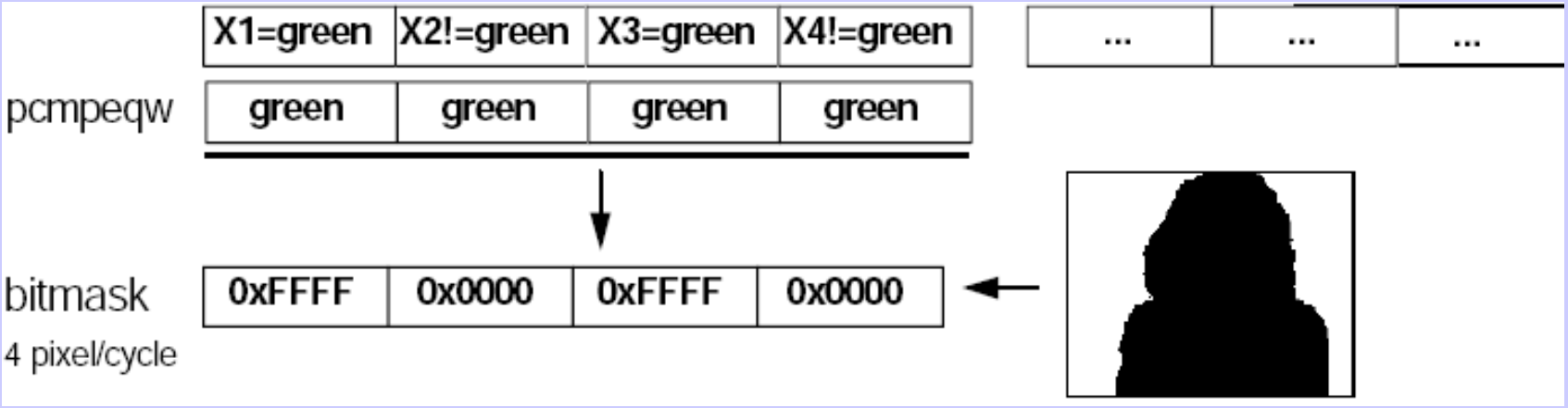
Parallélisme de données : exemple le *MMX*

- Les opérations arithmétiques peuvent gérer les débordements de capacité selon deux modes :
  - *wrap around* : le débordement est ignoré, le résultat correspond au modulo de la valeur max codable :
    - exemple sur 8 bits : $255 + 1 \rightarrow 0$
  - saturation : le débordement donne comme résultat une valeur saturée sur la valeur max (ou min) codable :
    - exemple sur 8 bits : $255 + 1 \rightarrow 255$

Parallélisme de données : exemple le *MMX*

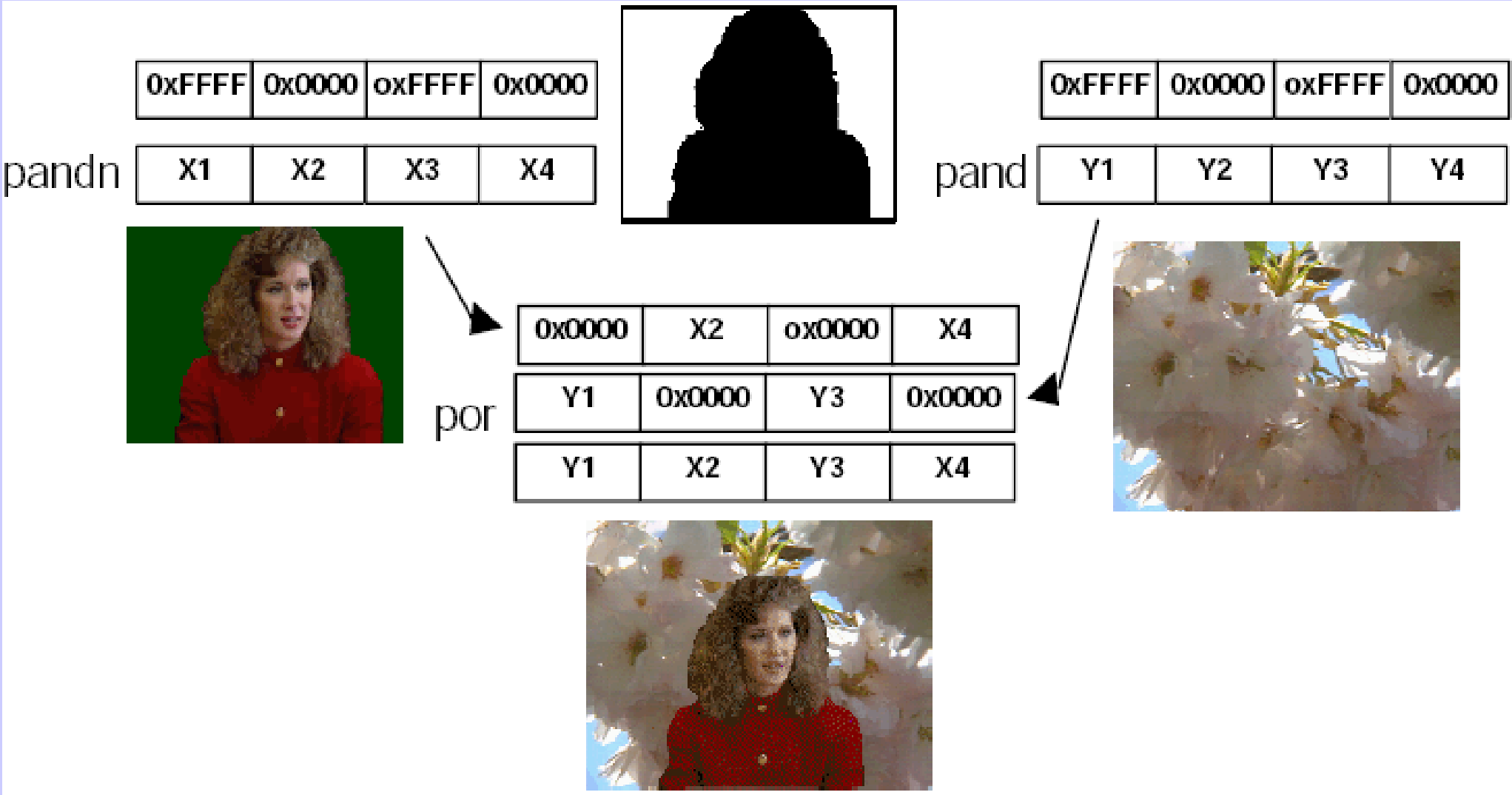
- exemple d'opération avec comparaison : *chroma-key* :

création du masque :



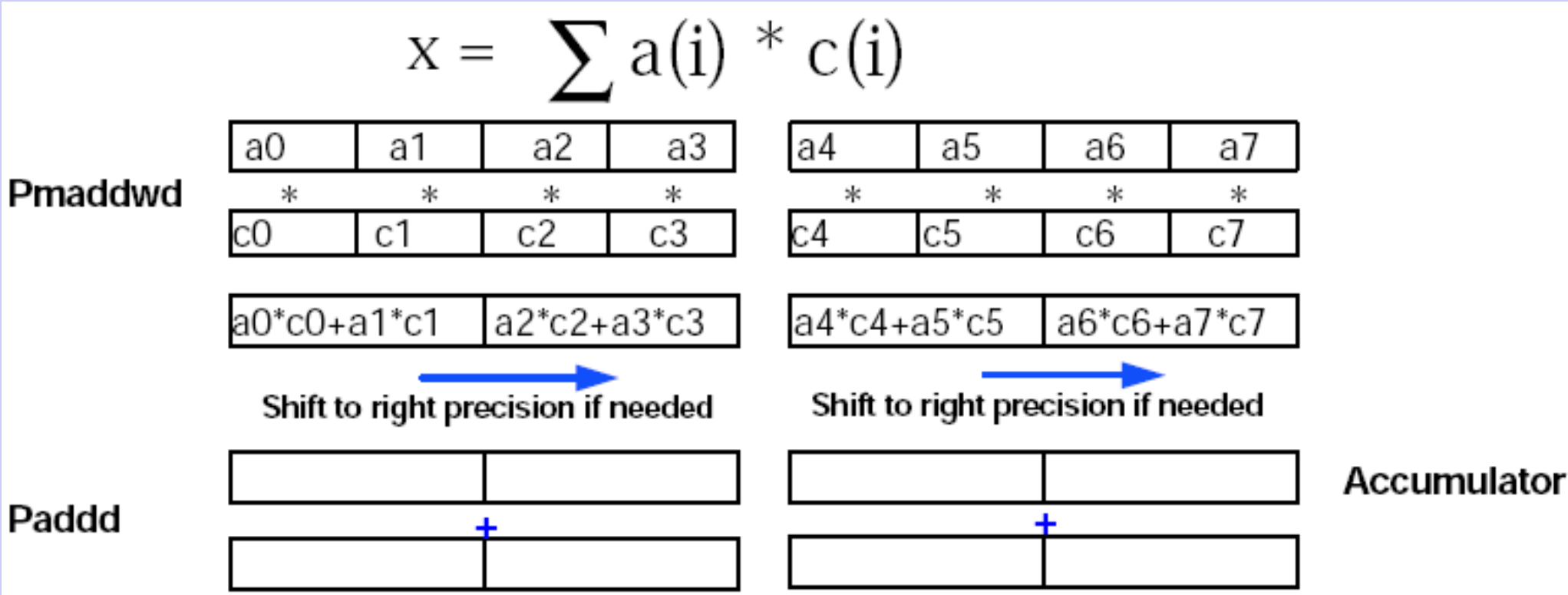
Parallélisme de données : exemple le *MMX*

- exemple d'opération avec comparaison : *chroma-key* :



Parallélisme de données : exemple le *MMX*

- exemple d'opération : calcul matriciel



Parallélisme de données : exemple le *MMX*

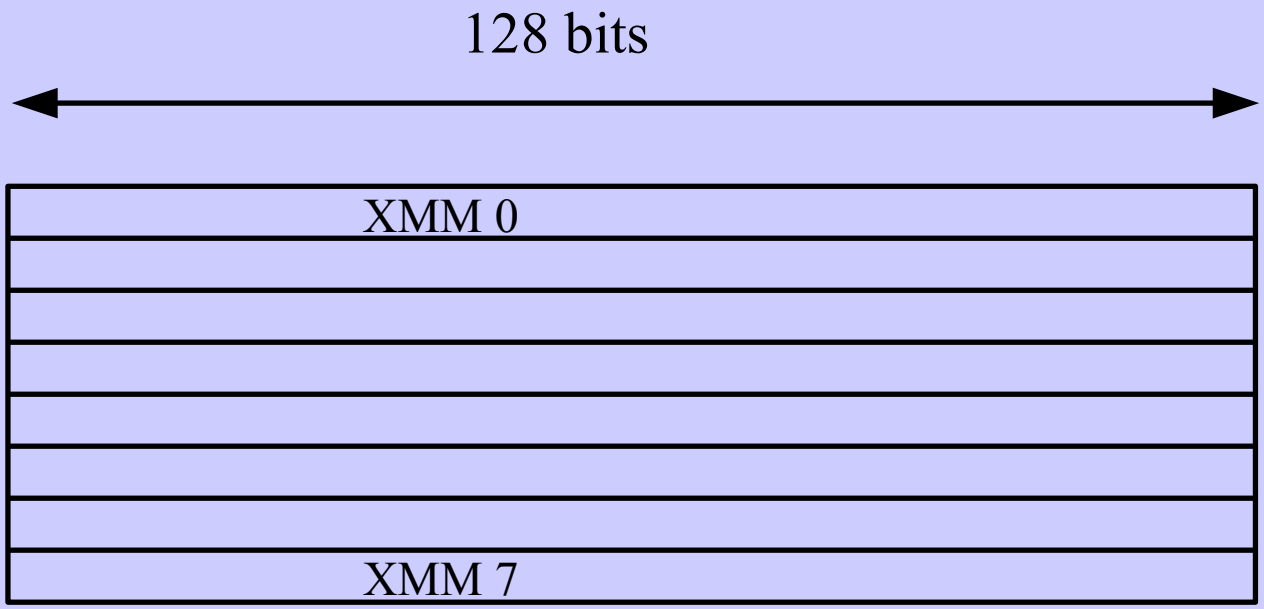
- exemple d'opération : calcul matriciel

nombre d'instruction avec et sans MMX

	Number of Instructions without MMX™ Technology	Number of MMX Instructions
Load	16	4
Multiply	8	2
Shift	8	2
Add	7	1
Miscellaneous	-	3
Store	1	1
Total	40	13

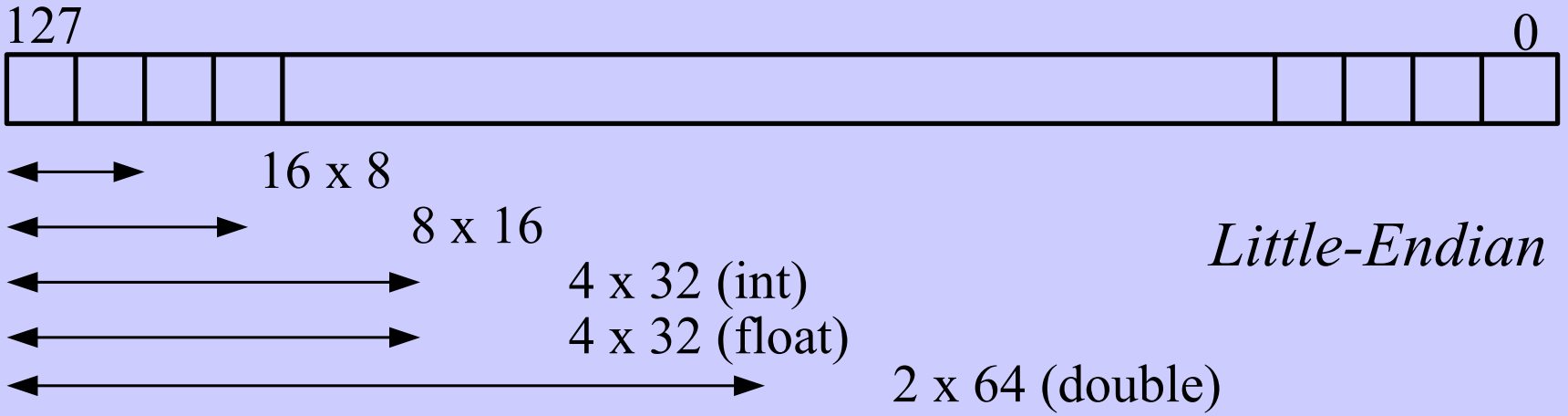
Parallélisme de données : exemple le *SSE*

- nouveaux registres : 128 bits XMM
  - 8 registres sur x86 : XMM0 → XMM7
  - 16 registres sur x86_64 : XMM0 → XMM15
  - distincts des registres flottants



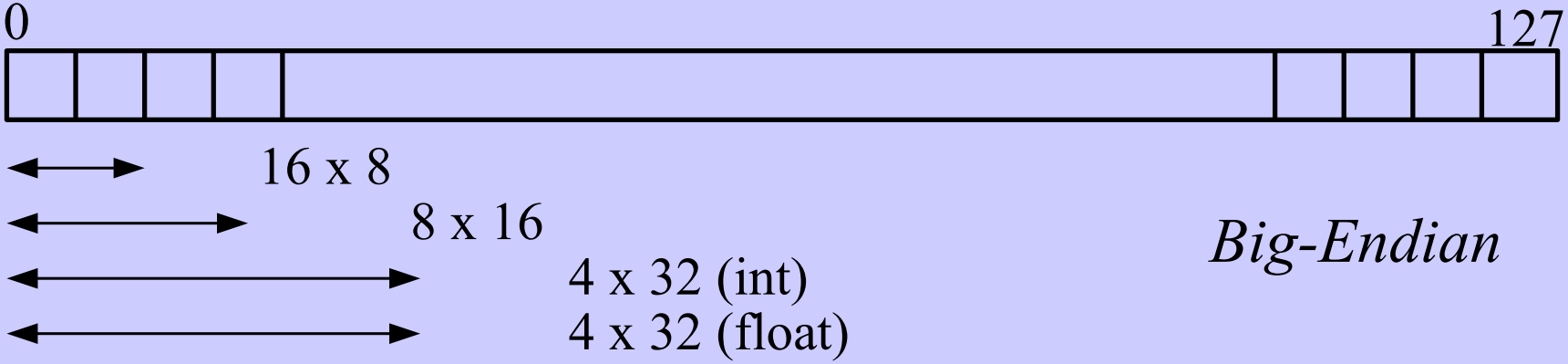
Parallélisme de données : exemple le *SSE*

- SSE : 8 registres XMM



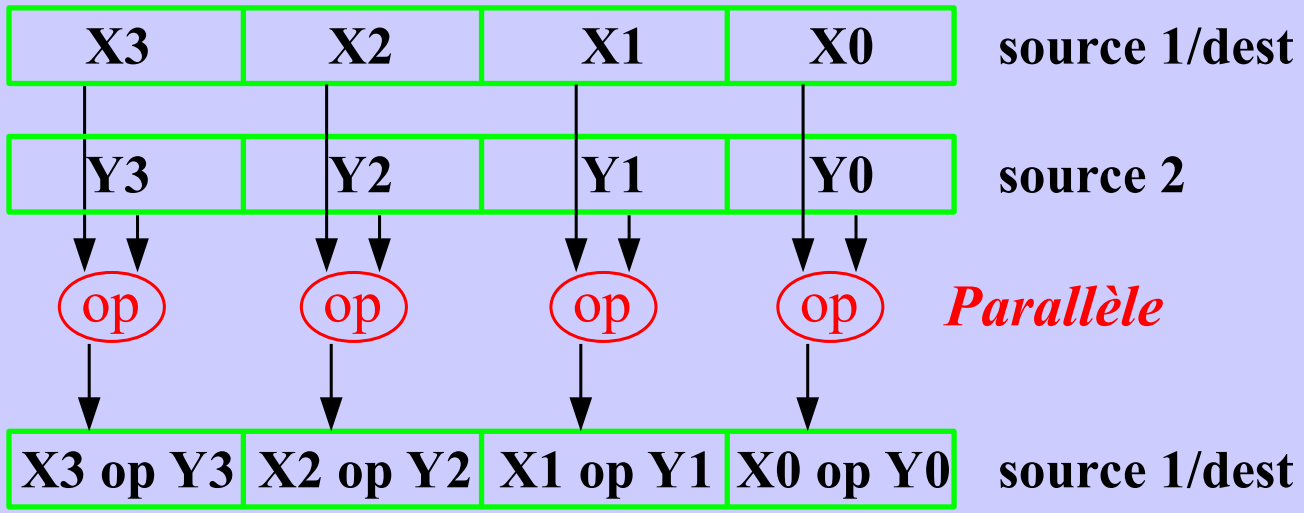
Accès mémoire 32, 64 et 128 bits

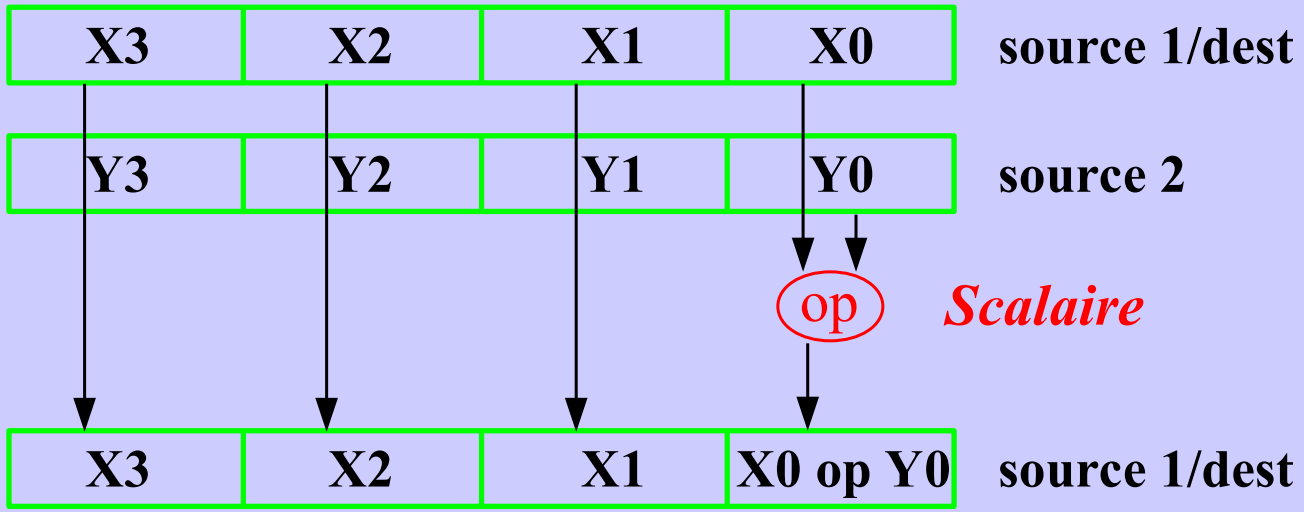
- Alvitec : 32 registre vectoriels



Parallélisme de données : instructions *SSE*

- Instructions SIMD parallèles et scalaires :
  - Instructions arithmétiques et logiques
  - Instructions mémoire
  - Instructions de formatage et manipulation
  - Instructions de conversion





Parallélisme de données : instructions *SSE*

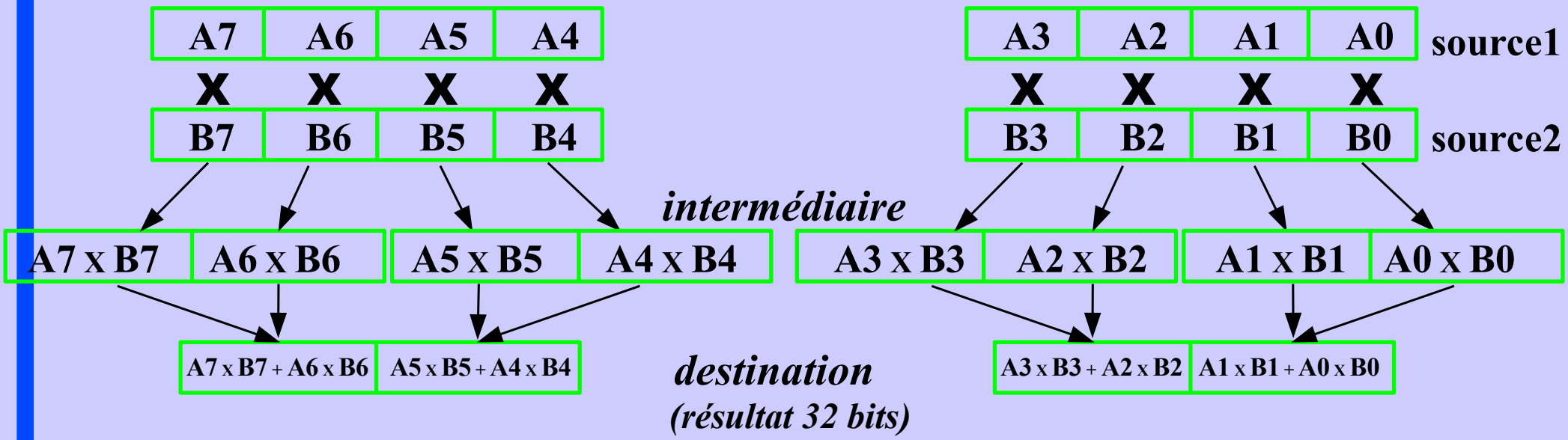
- opérations arithmétiques

Arithmétique entière	Complément à 2	Saturation signée	Saturation non signée
Addition	PADD(b,w,d)	PADDS(b,w)	PADDUS(b,w)
Soustraction	PSUB(b,w,d)	PSUBS(b,w)	PSUBUS(b,w)
Multiplication	PMUL(llw,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

Parallélisme de données : instructions *SSE*

- Multiplication – accumulation entière
  - 8 multiplications et 4 additions en une instruction PMADDWD



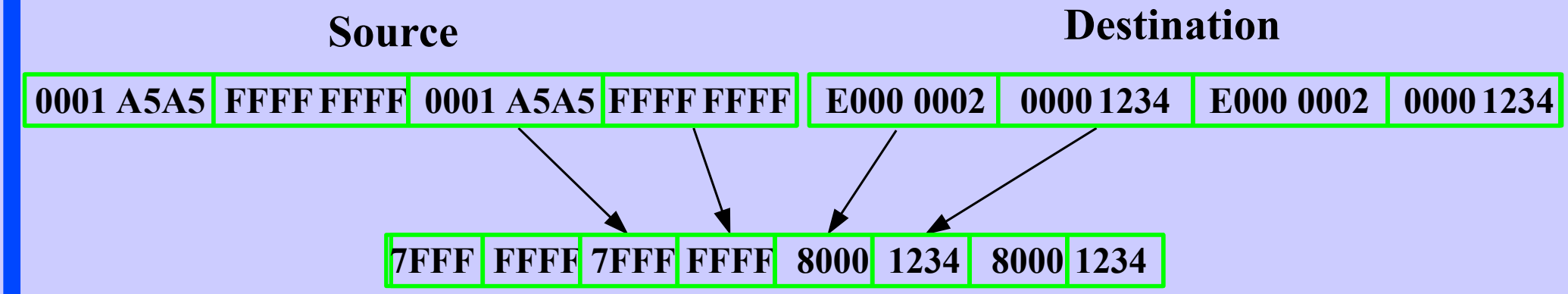
  - PMADDWD produit 2 résultat 32 bits
    - utile pour les applications multimédia et traitement du signal
    - formats d'entrée et de sortie différents
    - n'existe pas pour les données 8 bits en entrée

Parallélisme de données : instructions *SSE*

- Compactage – décompactage

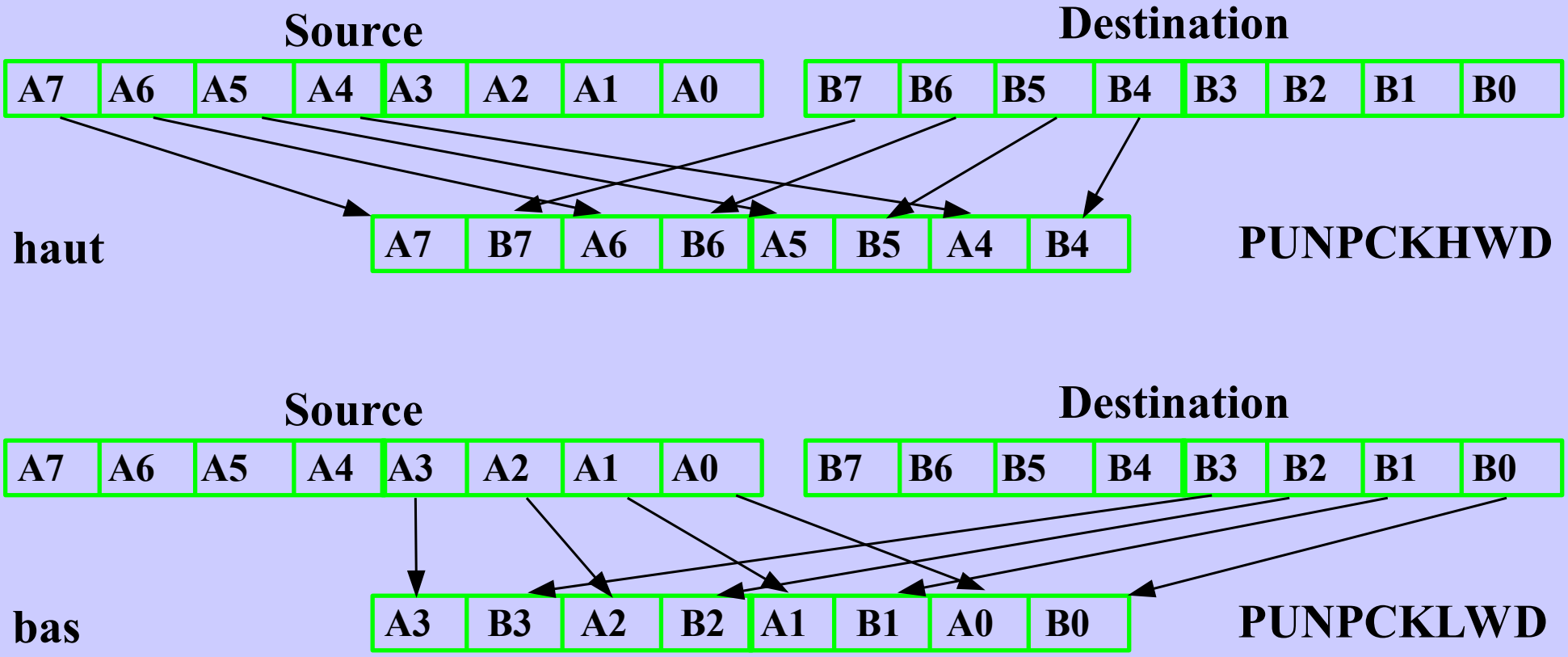
Arithmétique entière	Compl. à 2	Saturation signée	Saturation non signée
Pack		PACKSS(wb,dw)	PACKUS(wb)
Unpack High	PUNPCKH(bw,wd,dq)		
Unpack Low	PUNPCKL(bw,wd,dq)		

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



Parallélisme de données : instructions *SSE*

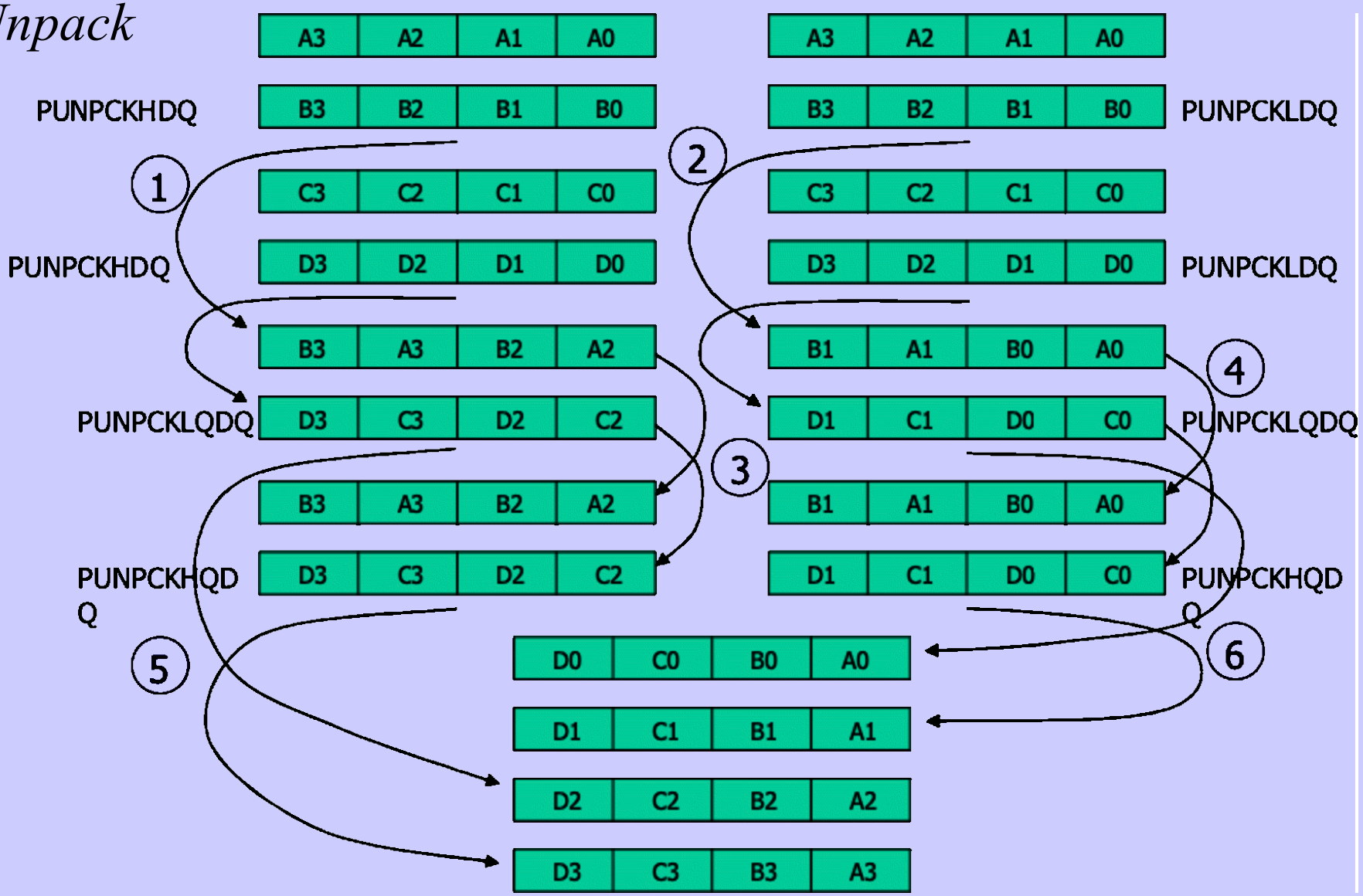
- Décompactage



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

Parallélisme de données : instructions *SSE*

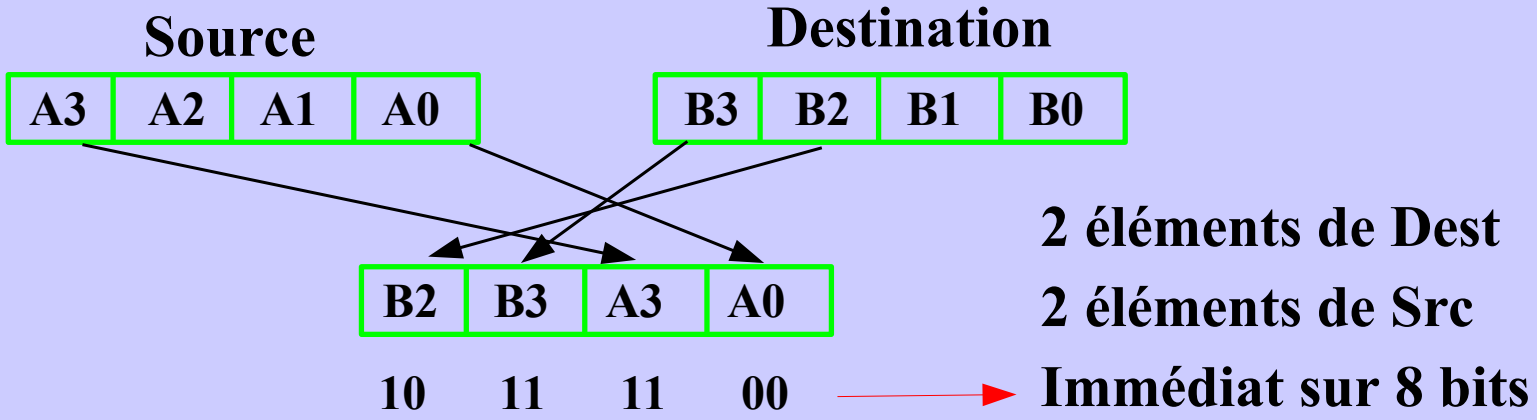
- Décompactage - exemple: transposition de matrice 4x4 avec *Unpack*



Parallélisme de données : instructions *SSE*

- Instruction "*Shuffle*"

PSHUFD
PSHUFHW
PSHUFLW
SHUFPD
SHUFPS



- Diffusion

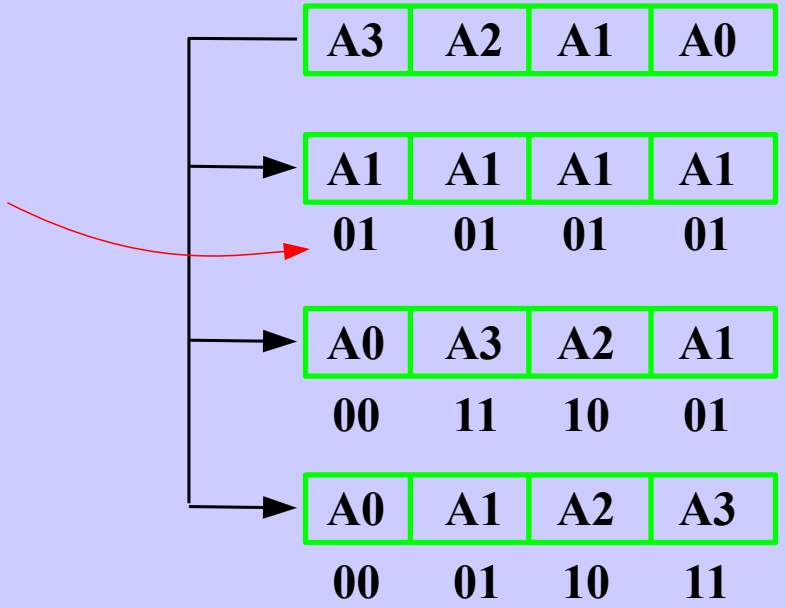
  shufps xmm1, xmm1, 55H

- Rotation

  shufps xmm1, xmm1, 39H

- Swap

  shufps xmm1, xmm1, 1BH



Parallélisme de données : instructions *SSE*

- Transferts mémoire :
  - Flottants :
    - entre registres XMMi et mémoire
      - movaps xmm1, [eax]
      - movaps [edi], xmm2
    - transferts alignés ou non
      - aligné 4 mots : moaps
      - aligné 2 mots haut : movhps
      - aligné 2 mots bas : movlps
      - non aligné : movups
    - transfert scalaire
      - movss : charge mot bas du registre et met à zéro les autres
    - entre partie haute et partie basse des registres
      - movhlps et movlhps
  - Entiers :
    - entre mémoire et registres XMM

Parallélisme de données : instructions *SSE*

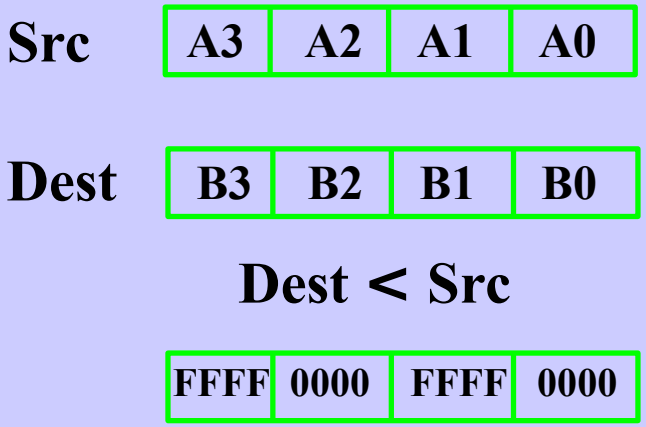
- Accès mémoire *"non write allocate"* :
  - movntps : XMM vers mémoire
  - movntq : MMX vers mémoire
  - sur un défaut de cache en écriture, il y a écriture directe en mémoire
- Préchargement
  - prefetcht0 : L1 et L2
  - prefetcht1 et prefetcht2 : L2 seul
  - prefetchnta : L1 seul

Parallélisme de données : instructions *SSE*

- 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.
  - *SSE* :
    - accès aligné
    - accès non aligné (plus lent)
  - *Alvitec* :
    - Les accès mémoire sont obligatoirement alignés
    - utilisation de plusieurs instructions pour les accès non alignés

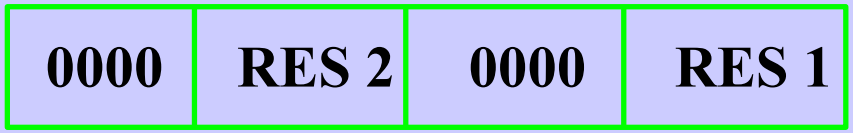
Parallélisme de données : instructions *SSE*

- Comparaison et opérations logiques :
  - *Compare* (eq, lt, le, unord, neq, nlt, nle, ord) *and set mask*
    - flottants
      - cmp*xx*ps ou cmp*xx*ss
      - cmp*xx*pd ou cmp*xx*sd
    - entiers
      - PCOMPEQ (B,W,D)
      - PCOMPGT (B,W,D)
  - Opérateurs logiques
    - and, andn, or, xor
      - versions ps et ss
      - version entière



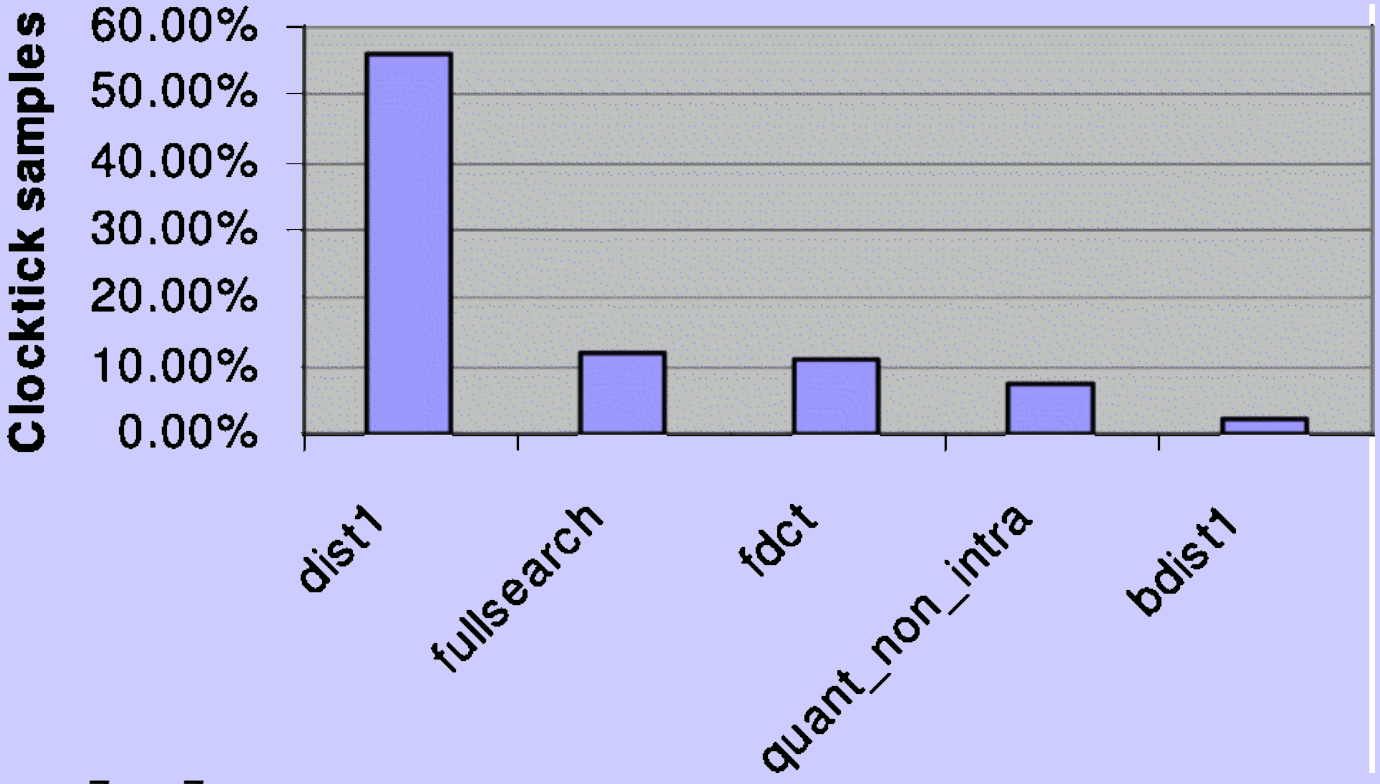
Parallélisme de données : instructions *SSE*

- Instructions spéciales : exemple PSABW
  - Valeur absolue des différences des octets *(unsigned char)* dans Dest et Src
  - Somme des valeurs absolues pour les huit octets bas dans les 32 bits de la partie basse de Dest
  - Somme des valeurs absolues pour les huit octets haut dans les 32 bits de la partie haute de Dest



Parallélisme de données : instructions *SSE*

- Instructions spéciales : exemple PSABW
  - Application à l'encodage mpeg 2



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

ou *SAD* 8x8 ou 16x16

Parallélisme de données : instructions *SSE*

- code C pour l'estimation de mouvement :

```
for (l=0; l<n_vert; l++)
    k=0;
    for (c=0; c<n_horz; c++)
    {
        answer = 0;
        for (j=0; l<16; j++)
            for (i=0; i<16; i++)
                answer += abs (x[l+j][k+i] – y[l+j][k+i])
        result[l][c]=answer;
        k+=8;
    }
```

  - version C naïve : 271 CPP (cycles par pixel)
  - version XMM : 13,5 CPP
  - accélération : 20

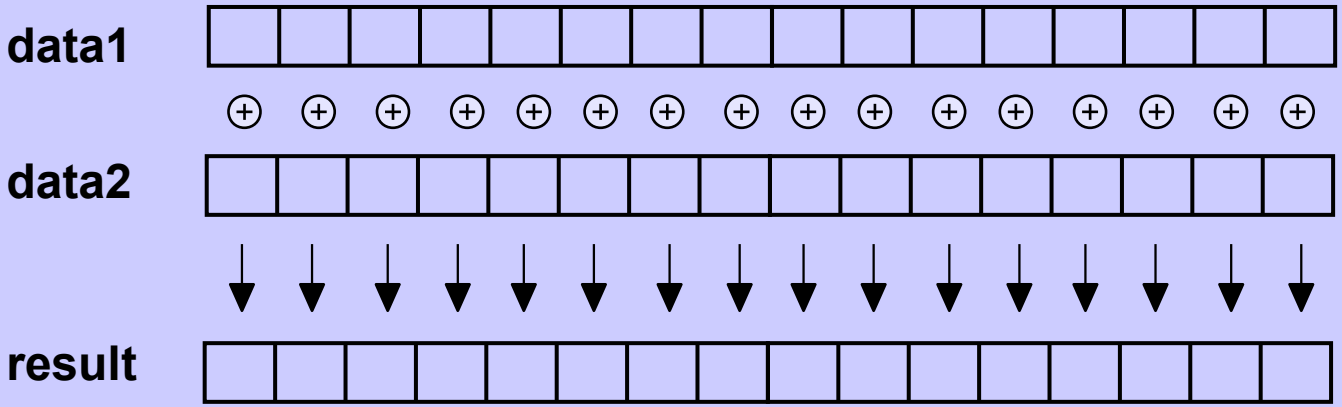
Parallélisme de données : Alvitec

- Intégré à un jeu d'instruction RISC (PowerPC)
- Unité de traitement SIMD
  - 32 registres vectoriels
  - 160 instructions vectorielles
- Opérandes
  - 16 octets
  - 8 entiers 16 bits
  - 4 entiers 32 bits
  - 4 flottants simple précision
- Format d'instructions :

Parallélisme de données : instructions Alvitec

- addition vectorielle
  - existe dans tous les formats flottant et entiers.

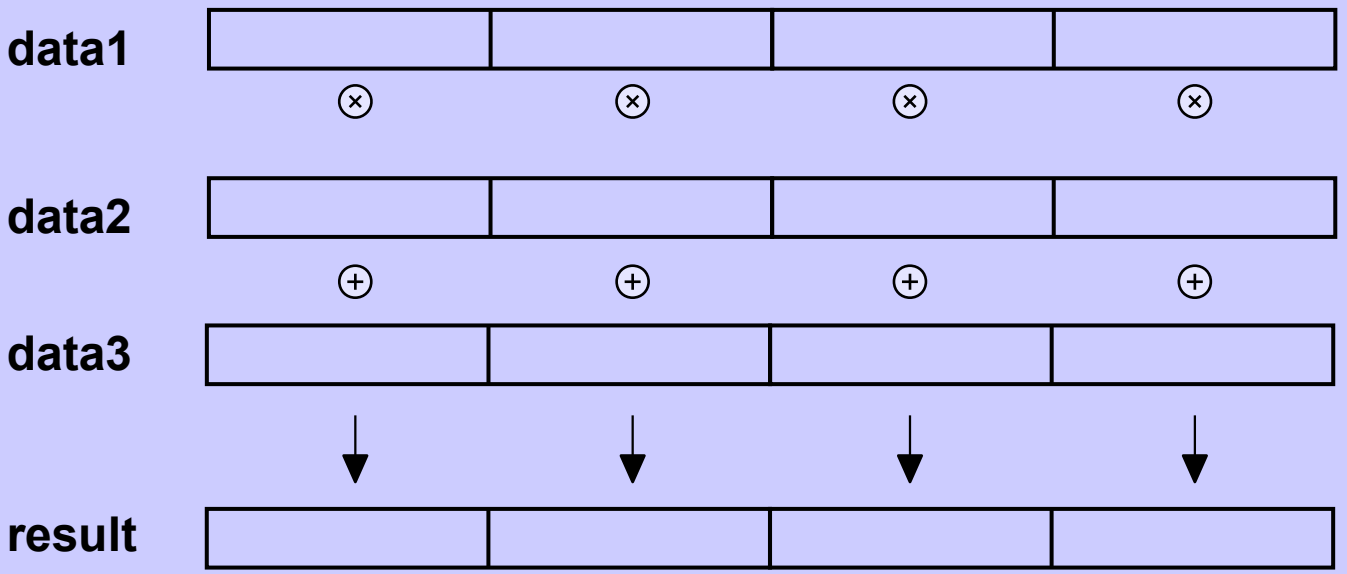
vaddubm result, data1, data2



Parallélisme de données : instructions Alvitec

- multiplication – accumulation :
  - multiplie quatre *floats* par quatre *floats* et ajoute à quatre *floats* en une instruction

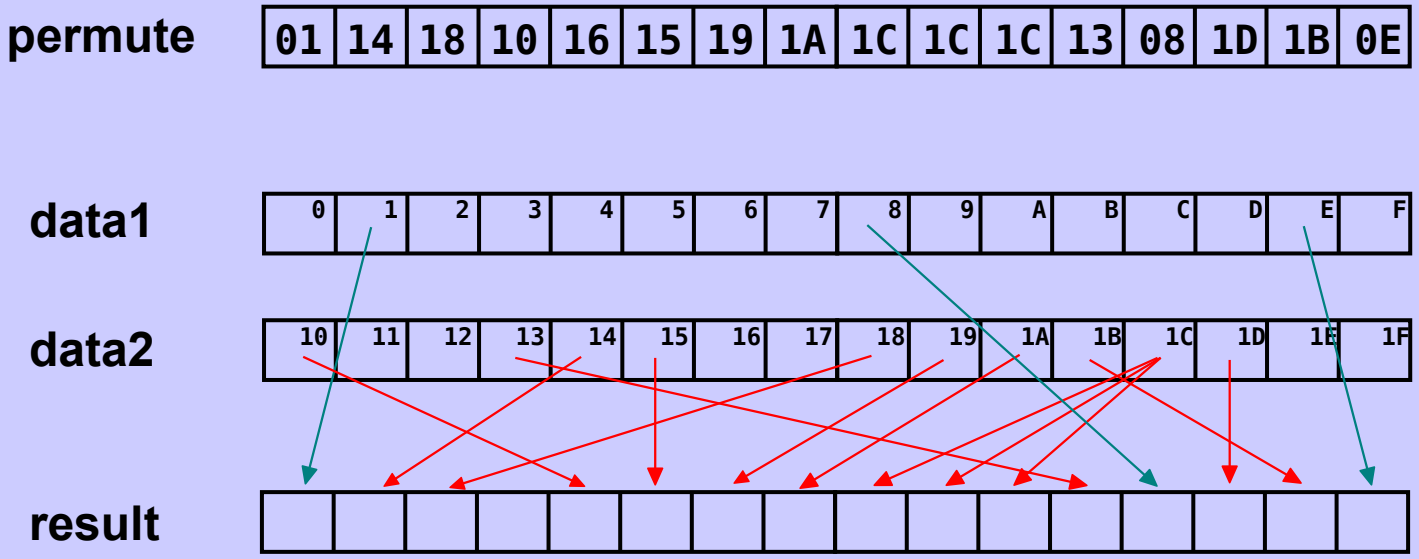
vmaddfp result, data1, data2, data3



Parallélisme de données : instructions Alvitec

- Permute :
  - L'instruction "*permute*" remplit un registre à partir de deux autres registres. Les octets peuvent être spécifiés dans n'importe quel ordre.

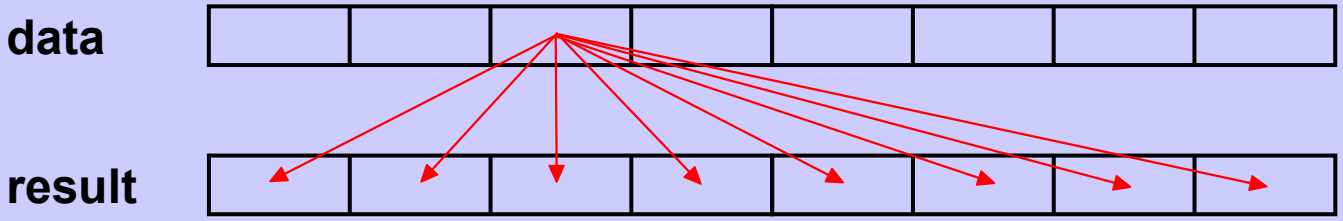
vperm result, data1, data2, permute



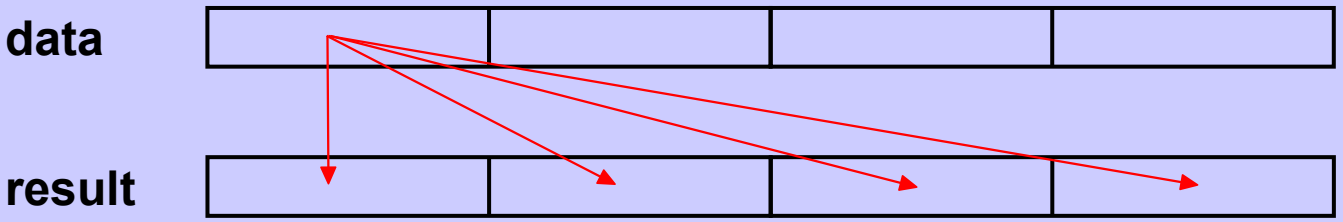
Parallélisme de données : instructions Alvitec

- "SPLAT" :
  - L'instruction *"splat"* est utilisée pour copier un élément d'un registre dans tous les éléments d'un autre registre

vsplth result, data, 2

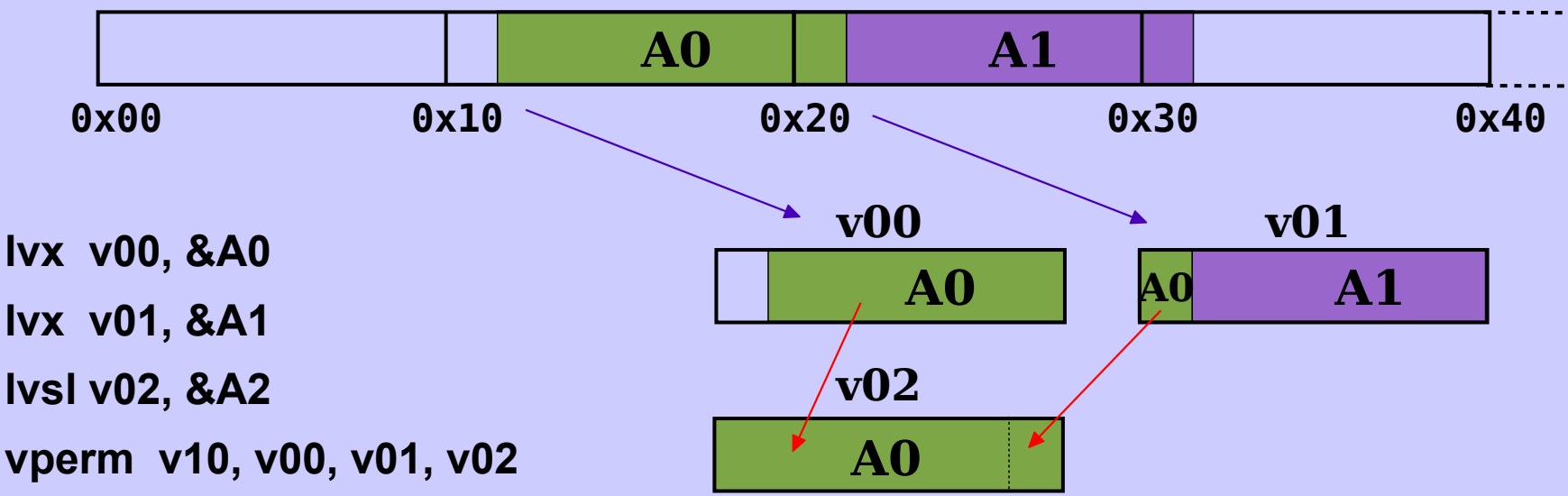


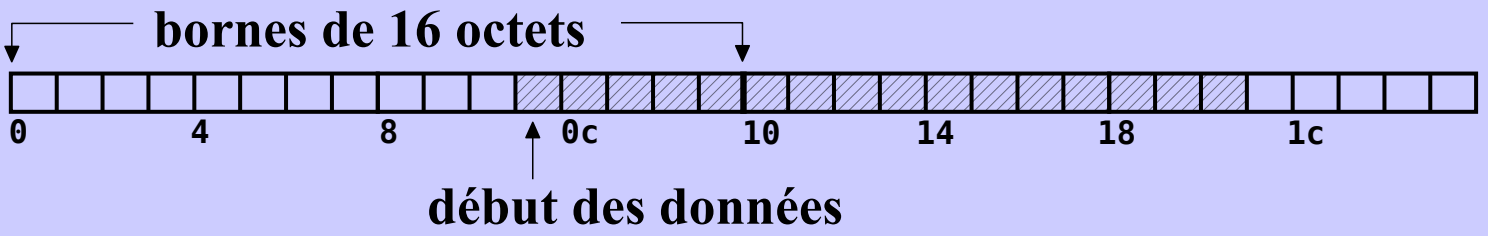
vspltw result, data, 0



Parallélisme de données : instructions Alvitec

- Accès mémoire non aligné





0b	0c	0d	0e	0f	10	11	12	13	14	15	16	17	18	19	1a

masque généré par lvsl
pour la permutation

Type d'utilisation des instructions SIMD

- Vectorisation automatique par compilateur C (ou Fortran)
  - transformation de 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)
```

Exemples SSE : *Intrinsics*

- Compilateur gcc / Intel icc sous x86/x64
  - mmintrin.h : MMX *Pentium MMX*
  - xmmintrin.h : SSE *Pentium III, Athlon XP*
  - emmintrin.h : SSE2 *Pentium 4, Pentium M*
  - pmmintrin.h : SSE3 *Pentium 4 Prescott, atom*
  - tmmintrin.h : SSSE3 *Pentium Dual-Core*
  - smmintrin.h : SSE4.1 *Core2*
  - nmmintrin.h : SSE4.2 *Core i7*
  - immintrin.h *contient les différentes versions d'*AVX *:*
    - avxintrin.h : AVX *Core i7 Sandy Bridge*
    - avx2intrin.h : AVX2 *Core Haswell*
    - avx512*intrin.h : AVX512 *Knights Landing*

Exemples SSE 2 : *Intrinsics*

- type de données - déclaration :

  __m64 registre MMX
  __m128 *packed* flottant simple précision – 32bits (XMM)
  __m128d *packed* flottant double précision – 64bits (XMM)
  __m128i *packed* entier (XMM)

- format général :

_mm_ {*opération*}{*alignement*}_{*organisation données*}{*type*}(...)

- exemple : addition de 2 vecteurs de 4 floats

  _mm_ add_ps(__m128 A, __m128 B)

Exemples SSE 2 : *Intrinsics*

_mm_ {*opération*}{*alignement*}_{*organisation données*}{*type*}(...)

- *alignement* :
  - si vide, les données sont alignées en mémoire
  - *u* : données non alignées en mémoire
- *organisation des données* :
  - *p* : packed
  - *ep* : extended packed (pour les entiers dans les registres XMM)
  - *s* : scalar
- *type d'opération* :
  - *s* : simple précision
  - *d* : double précision
  - *i#* : entier de # bits (8, 16, 32, 64, 128)
  - *u#* : entier non signé de # bits (8, 16, 32, 64, 128)

Exemples SSE 2 : *Intrinsics*

- Charge 2 doubles dans un vecteur:

```
double a[2] = {1.0, 2.0};
__m128d x = _mm_load_pd(a);
```

- Additionne 2 vecteurs de 2 doubles

```
__m128d a, b;
__m128d x = _mm_add_pd(a, b);
```

- Multiplie 2 vecteurs de 4 floats:

```
__m128 a, b;
__m128 x = _mm_mul_ps(a, b);
```

- Additionne 2 vecteurs de 8 entiers signés sur 16 bits avec saturation

```
__m128i a, b;
__m128i x = _mm_adds_epi16(a, b);
```

- Compare 2 vecteurs de 16 entiers signés sur 8 bits

```
__m128i a, b;
__m128i x = _mm_cmpgt_epi8(a, b);
```

Exemples SSE 2 : *Intrinsics*

- Fonctionnement général
  - déclaration des variables SSE (registres)

    ```
    __m128 r1
    ```
  - chargement des données de la mémoire vers les registres

    ```
    r1 = _mm_load...(type* p)
    ```
  - opérations sur les registres
  - placement du contenu des registres en mémoire

    ```
    _mm_store...(type* p, r1)
    ```

Exemples SSE 2 : *Intrinsics*

- Transfert mémoire ↔ registres XMM
  - Alignés ou non-alignés

    _mm_load_... ou _mm_loadu_...

    _mm_store_... ou _mm_storeu_...
  - Par vecteurs : 4xSP, 2xDP, ...

    _mm_load{u}_ps,

    _mm_load{u}_pd, ...

    _mm_store{u}_ps, _mm_store{u}_pd, ...
  - Par éléments scalaires

    _mm_load_ss, _mm_load_sd, ...

    _mm_store_ss, _mm_store_sd, ...

Exemples SSE 2 : *Intrinsics*

- Transfert mémoire ↔ registres XMM
  - Alignement mémoire :
    - Le processeur peut effectuer des transferts efficaces de 16 octets (128 bits) entre la mémoire et un registre SSE sous la condition que le bloc soit aligné sur 16 octets.
    - Cette contrainte est matérielle.
  - Attention
    - L'alignement dépend de l'architecture et du type de variable. Par défaut l'alignement d'un entier 32 bits est effectué sur 4 octets.
    - Pour obtenir l'alignement : `__alignof__ (type)`

`__alignof__ (int[4]) → 4`

Exemples SSE 2 : *Intrinsics*

- Transfert mémoire ↔ registres XMM
  - Déclaration alignée :
    - Alignement de données statiques sur 16 octets :

      gcc : `int x __attribute__ ((aligned (16)))`

      msdn: `__declspec(align(16)) int array[len];`

    - Alignement de données dynamiques sur 16 octets :

```
int posix_memalign(void **memptr, 16, sizeof(type))
```

Exemples SSE : *Intrinsics*

```
#include<stdio.h>
#include<xmmintrin.h>   //  Header pour SSE

Int main(){

   float       a1[4]  __attribute__  ((aligned (16)))
                  ={1.4,2.5,3.6,4.8};
   float       a2[4]  __attribute__ ((aligned (16)));
   __m128   v1, v2;
   v1 = _mm_load_ps(a1);
   _mm_store_ps(a2, v1);
   printf("%f %f %f %f\n", a2[0], a2[1], a2[2], a2[3]);
   return 0;
}
```

Exemples SSE 2 : *Intrinsics*

```
#include <emmintrin.h> // SSE2
...
// tableau d'entiers, 16 octets alignés
__declspec(align(16)) int array[len];
...
__m128i ones4 = _mm_set1_epi32(1);
__m128i *array4 = (__m128i*)array;
for (int i = 0; i < len/4; i++)
    array4[i] = _mm_add_epi32(array4[i], ones4);
```

Exemples SSE 2 : *Intrinsics*

```
#include <emmintrin.h> // SSE2
...
// alloc. statique alignée
float   a1[4] __attribute__((aligned(16))) = {1.f, 2.f, 3.f, 4.f};

// alloc. dynamique alignée
float*  a2  = (float*)_mm_malloc(4*sizeof(float), 16);


float    a3[4] __attribute__((aligned(16)));
__m128 r1, r2;


for(unsigned int i = 0 ; i < 4 ; ++i)
    a2[i] = i;


r1 = _mm_load_ps(a1);
r2 = _mm_load_ps(a2);
r1 = _mm_add_ps(r1, r2);
_mm_store_ps(a3, r1);


for(unsigned int i = 0 ; i < 4 ; ++i)
    cout << a3[i] << endl;


  _mm_free(a2);
```

Exemples SSE 2 : *Intrinsics*

- Opérations arithmétiques
  - Opérations de base
    - `_mm_add_pd` - (Add-Packed-Double)

      Entrée : [A0, A1], [B0, B1]

      Sortie : [A0 + B0, A1 + B1]

    - `_mm_add_ps` - (Add-Packed-Simple)

      Entrée : [A0, A1, A2, A3], [B0, B1, B2, B3]

      Sortie : [A0 + B0, A1 + B1, A2 + B2, A3 + B3]

Exemples SSE 2 : *Intrinsics*

- Opérations arithmétiques
  - Opérations de base
    - _mm_add_epi64 - (Add-Packed-LongInt)

      Entrée : [A0, A1], [B0, B1]

      Sortie : [A0 + B0, A1 + B1]

    - _mm_add_epi32 - (Add-Packed-Int)

      Entrée : [A0, A1, A2, A3], [B0, B1, B2, B3]

      Sortie : [A0 + B0, A1 + B1, A2 + B2, A3 + B3]

Exemples SSE 2 : *Intrinsics*

- Opérations arithmétiques
  - Opérations de base
    - _mm_mul_pd - (Multiply-Packed-Double)

      Entrée : [A0, A1], [B0, B1]

      Sortie : [A0 * B0, A1 * B1]

    - _mm_mul_ps - (Multiply-Packed-Simple)

      Entrée : [A0, A1, A2, A3], [B0, B1, B2, B3]

      Sortie : [A0 * B0, A1 * B1, A2 * B2, A3 * B3]

Exemples SSE 2 : *Intrinsics*

- Opérations logiques
  - _mm_{and | or | xor | ...}_{ps | pd | si128}
- Tests
  - _mm_cmp_{eq | gt | lt | ...}_{ps | pd | epi8 | epi 16 | ...}
- Réorganisation des données
  - _mm_shuffle_...
- Conversions
  - _mm_cvt_...

Exemples SSE 2 : *Intrinsics*

```
float in1[4] = { 1.0, 2.0, 3.0, 4.0 };
float in2[4] = { -1.0, -2.0, -3.0, -4.0 };
float out[4];
for (i = 0; i < 4; i++)
    out[i] = in1[i] + in2[i];

// entrées in1,in2 et sortie out sur 16 octets alignés
#include <emmintrin.h>
__m128 x, y, z;
x = _mm_load_ps(in1);
y = _mm_load_ps(in2);
z = _mm_add_ps(x, z);
_mm_store_ps(out,z);
```

Exemples SSE 2 : *Intrinsics*

```
void toto () {
    for (int j=0; j<SIZE; j++)
        xa[j] = xb[j] + q*xc[j];
}
```

```
void toto () {
    __m128 tmp0, tmp1;
    tmp1 = _mm_set_ps1(q);
    for (int j=0; j<SIZE; j+=VECTOR_SIZE) {
        tmp0 = _mm_mul_ps(*((__m128*)&xc[j]), tmp1);
        *(__m128*)&xa[j] = _mm_add_ps(tmp0, *((__m128*)&xb[j]));
    }
}
```

Exemples SSE 2 : *Intrinsics*

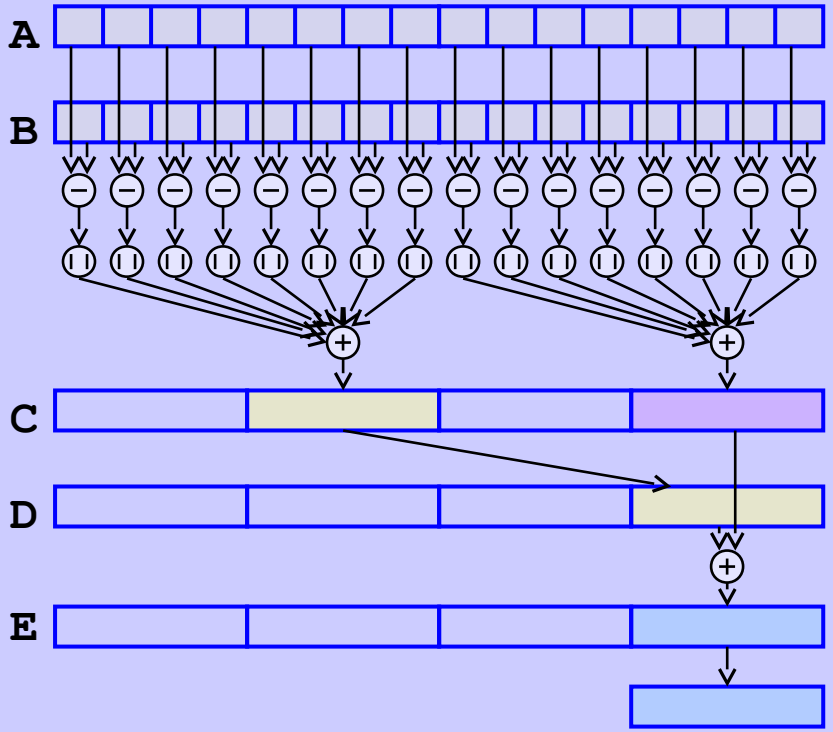
- Calcul du SAD (*Sum of Absolutes Differences*), par exemple pour l'estimation de mouvement en vidéo mpeg – code C :

```
uint8 *a, *b;
int diff, sad;

sad = 0;
for (i=0; i<16; i++)
{
    diff = a[i] – b[i];
    sad += diff > 0 ? diff : -diff;
}
```

Exemples SSE 2 : *Intrinsics*

- Calcul du SAD pour l'estimation de mouvement – version SSE :

```
uint8 *a, *b;
__m128i A, B, C, D, E;
int sad;

A = _mm_load_si128((__m128i *) a);
B = _mm_load_si128((__m128i *) b);
C = _mm_sad_epu8(A, B);
D = _mm_srli_si128(C, 8);
E = _mm_add_epi32(C, D);
sad = mm_cvtsi128_si32(E);
```



Exemples SSE 2 : *Vector Class Library*

- Classes C++ pour le SSE :
  - Classes Entiers I*b*vec*n*

I8vec8	(8 8bit)	I8vec16	(16 8bit)
I16vec4	(4 16bit)	I16vec8	(8 16bit)
I32vec2	(2 32bit)	I32vec4	(4 32bit)
I64vec1	(1 64bit)	I64vec2	(2 64bit)
		I128vec1	(1 128bit)

`s' ou `u' après `I' pour entiers signés ou non, pacqueté

  - Classes Virgule flottante F*b*vec*n*

F32vec4	(4 32bit)	F64vec2	(2 64bit)

Exemples SSE 2 : *Vector Class Library*

- Classes C++ pour le SSE :

```
#include <dvec.h> // SSE2
...
// tableau d'entiers de 16 octets alignés
__declspec(align(16)) int array[len];
...
Is32vec4 *array4 = (Is32vec4*)array;
for (int i = 0; i < len/4; i++)
    array4[i] = array4[i] + 1; // incrémente de 4 entiers
```

Exemples SSE 2 : *Vector Class Library*

```
void toto () {
    for (int j=0; j<SIZE; j++)
        xa[j] = xb[j] + q*xc[j];
}
```

```
void toto () {
    F32vec4 q_xmm = (q, q, q, q);
    F32vec4 *xa_xmm = (F32vec4*)&xa;
    F32vec4 *xb_xmm = (F32vec4*)&xb;
    F32vec4 *xc_xmm = (F32vec4*)&xc;
    for (int j=0; j < SIZE/VECTOR_SIZE; j++)
        xa_xmm[j] = xb_xmm[j] + q_xmm*xc_xmm[j];
}
```

Exemples SSE 3 : *Intrinsics*

- Opérations arithmétiques
  - Addition&Soustraction
    - `_mm_addsub_pd` - (Add-Subtract-Packed-Double)

      Entrée : [A0, A1], [B0, B1]

      Sortie : [A0 – B0, A1 + B1]

    - `_mm_addsub_ps` - (Add-Subtract-Packed-Simple)

      Entrée : [A0, A1, A2, A3], [B0, B1, B2, B3]

      Sortie : [A0 – B0, A1 + B1, A2 – B2, A3 + B3]

Exemples SSE 3 : *Intrinsics*

- Opérations horizontales
  - _mm_hadd_pd - (Horizontal-Add-Packed-Double)

    Entrée : [A0, A1], [B0, B1]

    Sortie : [B0 + B1, A0 + A1]
  - _mm_hadd_ps - (Horizontal-Add-Packed-Simple)

    Entrée : [A0, A1, A2, A3], [B0, B1, B2, B3]

    Sortie : [B0 + B1, B2 + B3, A0 + A1, A2 + A3]
  - _mm_hsub_pd - (Horizontal-Subtract-Packed-Double)

    Entrée : [A0, A1], [B0, B1]

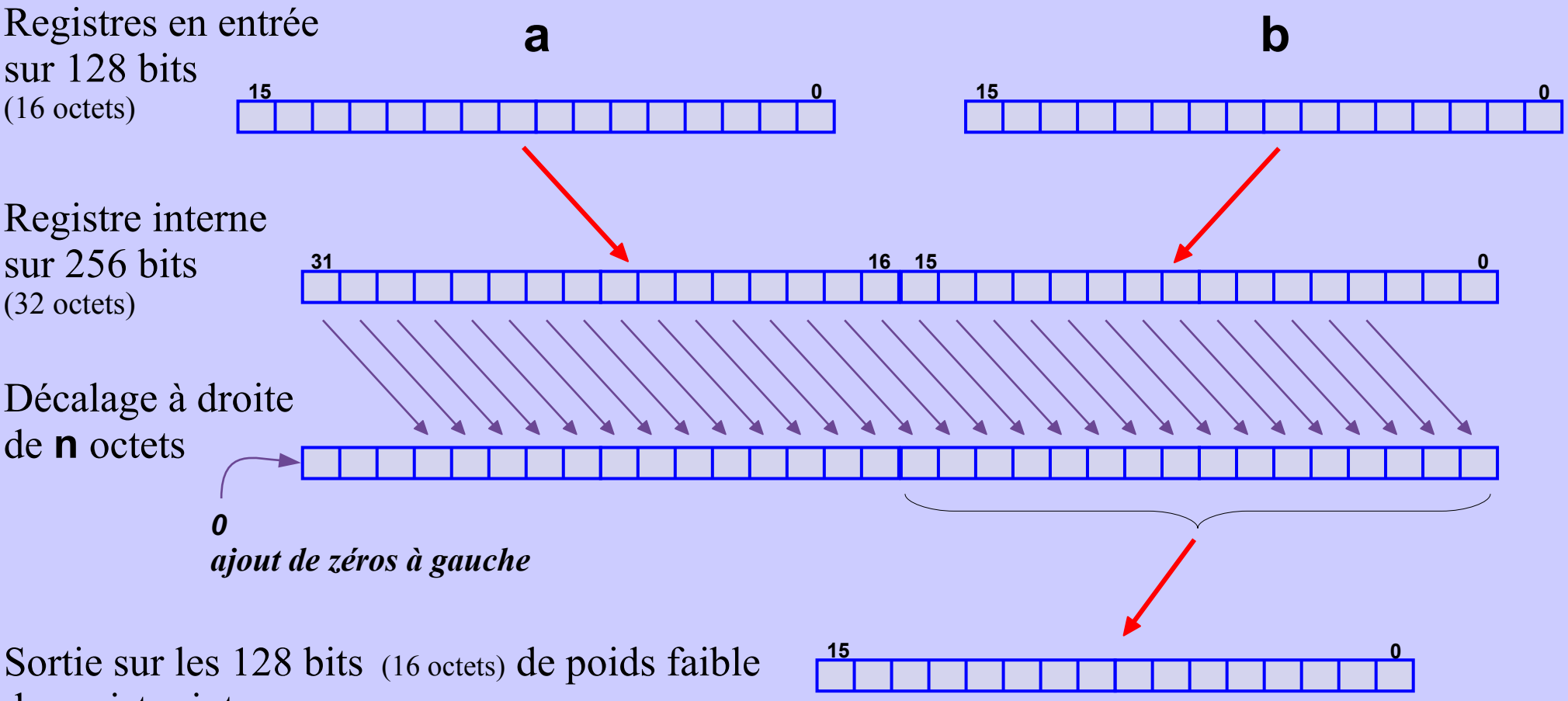
    Sortie : [B0 – B1, A0 – A1]
  - _mm_hsub_ps - (Horizontal-Subtract-Packed-Simple)

    Entrée : [A0, A1, A2, A3], [B0, B1, B2, B3]

    Sortie : [B0 – B1, B2 – B3, A0 – A1, A2 – A3]

Exemples SSE 4.1 : *Intrinsics*

- Alignement horizontale sur deux registres
  - __m128i _mm_alignr_epi8 (__m128i a, __m128i b, int n)



Évolution des extensions SIMD Intel

1999 SSE	2000 SSE2	2004 SSE3	2006 SSSE3	2007 SSE4.1	2008 SSE4.2	2009 AES-NI	2010\11 AVX
70 instr Single-Precision Vectors Streaming operations	144 instr. Double-precision Vectors 8/16/32 64/128-bit vector integer	13 instr. Complex Data	32 instr. Decode	47 instr. Video Graphics building blocks Advanced vector instr.	8 instr. String/ XML processing POP-Count CRC	7 instr. Encryption and Decryption Key Generation	~100 new instr. ~300 legacy sse instr. updated 256-bit vector 3 and 4 operand instr.

2013 AVX2	2016 AVX512	Futur ? AVX1024

AVX : évolutions des extensions SIMD Intel

- AVX : *Advanced Vector Extension*
  - 16 registres 256 bits : YMM0-15 (*XMM en partie basse*)
  - Opérations non destructives à 3 opérandes :
    - $R1 \leftarrow R2$ op $R3$
  - Alignement mémoire moins contraignant
  - Toutes les instructions flottants SSE 128bits implémentées en AVX
  - Deux modes SIMD ≠ AVX et SSE

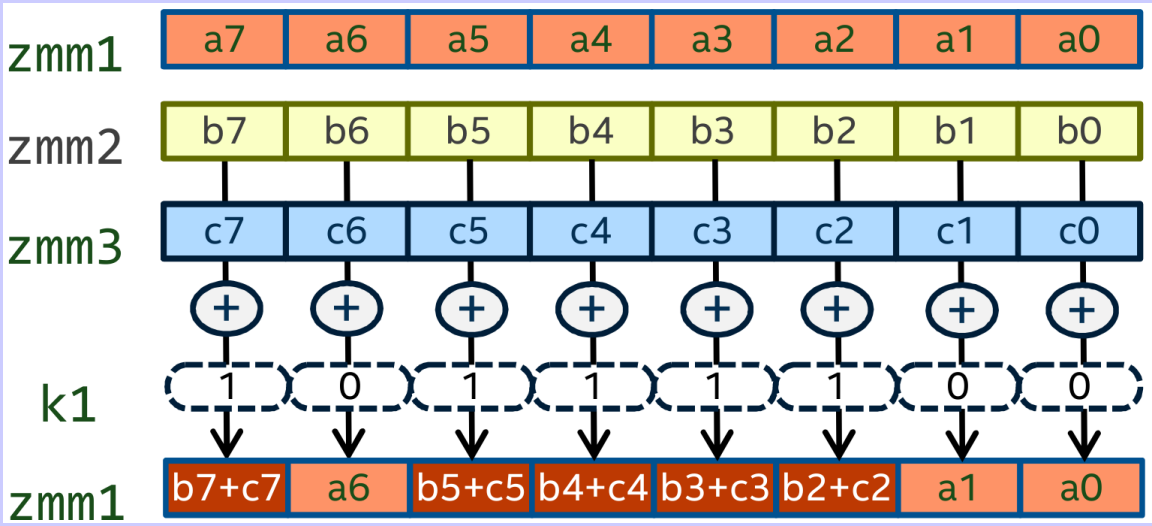
    mélange AVX–SSE → perte de performance, à éviter
- AVX 2 :
  - Extension des instructions entiers SSE 128 sur 256 bits
  - Accès mémoire non contigus des données 32 et 64 bits *(Gather/Scatter)*
  - Diffusion de données entre registres et mémoire *(Broadcast)*
  - Permutation de données entre registres *(Permute)*

AVX : évolutions des extensions SIMD Intel

- AVX 512 : registres 512 bits ZMM0-31
  - 32 registres 512 bits : ZMM0-32 (*YMM et XMM en partie basse*)
  - Opérations à 4 opérandes
    - *R1* ← (*R4*) *R2* op *R3* *R4 masque conditionnel*
  - Mais plusieurs extensions AVX-512 ≠ selon CPU…
    - *Foundation* (F) : instructions SSE et AVX flottants (dans tous les CPUs AVX-512)
    - *Conflict Detection Instructions* (CD) : détections de conflits dans les boucles
    - *Exponential and Reciprocal Instructions* (ER)
    - *Prefetch Instructions* (PF) : préchargement cache
    - *Byte and Word Instructions* (BW) : opérations entiers 8 et 16 bits
    - *Doubleword and Quadword Instructions* (DQ) : opérations entiers 32 et 64 bits
    - *Integer Fused Multiply Add* (IFMA) : a.x + b sans perte de précision
    - *Vector Neural Network Instructions Word variable precision* (4VNNIW) : instructions pour le *deep learning* (équivalentes du FMA)
    - ...
    - + futurs extensions

AVX : évolutions des extensions SIMD Intel

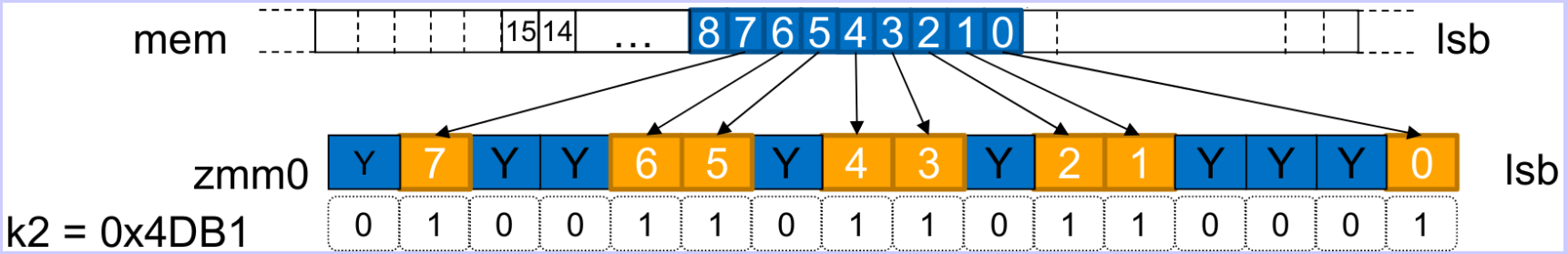
- AVX 512 : Opérations à 4 opérandes et masque conditionnel
  - VADDPD zmm1 {k1}, zmm2, zmm3 k1 *masque conditionnel*



  - VADDPS zmm0 {k1}{z}, zmm3, [mem] masque utile pour :
    - Supprimer individuellement les éléments de mémoire en lecture
    - Éviter des opérations sur certains éléments
    - Éviter la mise à jour de certains éléments en mémoire
    - z : indique s'il faut remplacer la valeur par zéro ou ne pas la recopier (*merge*)

AVX : évolutions des extensions SIMD Intel

- AVX 512 : Opérations *Expand / Compress*
  - VEXPANDPS zmm0 {k2}, [mem]



  - Opérations :
    - *Scatter write*: `a[b[x]] = d[x];`
    - *Histogram*: `a[b[x]]++;`
    - *Expand*: `if (c[i]) a[i] = b[i] * d[j++];`
    - *Compress*: `if (c[i]) a[j++] = b[i] * d[i];`