Architectures non généralistes

SEOC3A – CEAMC

Arthur Perais (arthur.perais@univ-grenoble-alpes.fr)

Certaines slides sont prises de sources diverses et traduites/modifiées (notamment D. Patterson et K. Asanovic)

Architectures de von Neumann

- La majorité des architectures restent des architectures de von Neumann
  - Instructions et données dans la même mémoire
  - Flôt de contrôle (branchements, appels de fonctions…)

➢ Les instructions sont des données qui sont interprétées par la machine comme étant des instructions

Taxonomie de Flynn

- Façon de classifier les machines parallèles proposée par Michael J. Flynn en 1996.
- Classifie en deux dimensions : **Instruction** et **Donnée**, chacune ayant deux états : **Simple** et **Multiple**

	Instruction(s)	
Donnée(s)	**SISD** **S**ingle **I**nstruction **S**ingle **D**ata	**SIMD** **S**ingle **I**nstruction **M**ultiple **D**ata
	MISD **M**ultiple **I**nstruction **S**ingle **D**ata	**MIMD** **M**ultiple **I**nstruction **M**ultiple **D**ata

Single Instruction Single Data

- **Un seul flux d'instructions** => un seul compteur de programme (Program Counter, PC)
- **Un seul flux de données** => un load charge un seul élément (scalaire)

- Par exemple, un processeur généraliste tel qu'on l'a vu jusqu'ici
  - Ou encore, un cœur dans un processeur généraliste multicœur
  - En général, orienté **latence** (minimiser le temps d'exécution d'une instruction)

Single Instruction Multiple Data

- **Un seul flux d'instructions** => un seul compteur de programme (Program Counter, PC)
- **Plusieurs flux de données** => un load charge plusieurs éléments à la fois (vecteur)
- Fonctionne particulièrement bien pour les problèmes réguliers (flôt de contrôle uniforme), e.g., le traitement d'image

- Par exemple, un GPU moderne est une machine SIMD
  - En général, orienté **débit** (maximiser le nombre d'opérations en parallèle

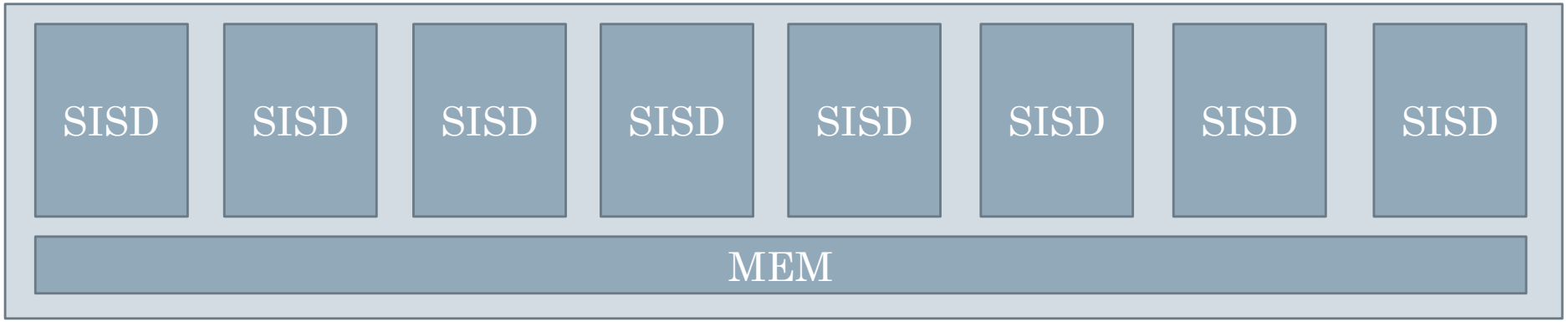
Multiple Instructions Single Data

- Unique flux de donnée envoyé à plusieurs unités fonctionnelles différentes
- Chaque unité opère sur la même donnée de façon indépendante
- Peu d'exemples, mais pourrait servir à :
  - Appliquer plusieurs filtres à un même signal en parallèle

Multiple Instructions Multiple Data

- Une collection de machines SISD indépendantes mais partageant une mémoire

- Par exemple, un processeur multicœur



Plusieurs types de parallélisme

- Chaque paradigme peut tirer parti de différents types de parallélisme au sein du programme

  - **Instruction Level Parallelism – ILP =>** SISD : pipelining, superscalaire, OoO, VLIW (on va y venir)
  - **Data Level Parallelism – DLP =>** SIMD : machines vectorielles
  - **Thread Level Parallelism – TLP =>** MIMD : multicœurs SISD à mémoire partagée

Plusieurs types de parallélisme

- TLP vs. DLP

  - **TLP :** Traiter plusieurs requêtes indépendantes dans une BDD (ou un serveur web) en même temps : instructions différentes, données différentes (MIMD)

  - **DLP :** Traiter plusieurs éléments d'un jeu de données en même temps, e.g., appliquer un filtre à plusieurs pixels en même temps : même instruction, données différentes (SIMD)

Plusieurs types de parallélisme

- TLP vs. DLP
- Suivant le problème, on peut parfois combiner TLP et DLP
- Dans les deux cas, bénéficie aussi de l'ILP (e.g., multicœur où chaque cœur est une machine SISD pipelinée avec exécution out-of-order)

Plusieurs types de parallélisme

- Chaque programme a des parties plus ou moins « parallèle » selon les axes ILP/TLP/DLP

- Quel paradigme choisir pour :
  - Maximiser la performance
  - Minimiser l'énergie consommée
  - Faciliter la programmation/debug
  - Portabilité

Plusieurs types de prallélisme

Ce qu'on a vu jusqu'ici (InO, OoO)

- SISD (ILP)
  - Facile à programmer
  - Performance la « moins pire » pour les programmes avec flot de contrôle complexe

---

- SIMD (DLP, ILP)
  - Un peu moins facile à programmer
  - Performance++ si le programme a en effet du DLP

- MIMD (TLP, ILP)
  - Difficile à programmer (synchronisation)
  - Performance++ si le programme a en effet du TLP

Microarchitectures SISD : Out of order

- Compilateur fait au mieux, le matériel optimisera à l'exécution

- La complexité est dans le matériel
  - Gestion des dépendances
  - Extraction du parallélisme d'instruction (ILP)
  - Spéculation

Microarchitectures SISD : Out of order

- Fenêtre d'instructions avec 100+ instructions
  - Chaque instruction qui rentre dans le pipeline doit déterminer si elle dépend de chacune des autres instructions en vol (*1 vers n)*
- Accès associatifs
  - Wakeup (*n* résultats vers *#ordonnanceur)*
  - Load Queue/Store Queue (*p to #LQ* et *q to #SQ)*
- Ordonnancement dynamique
  - Choisir 6-8-10 instructions à exécuter à chaque cycle *rapidement*

Microarchitectures SISD : Out of order

- Est-ce bien raisonnable ?
  - Consommation et surface explosent par rapport à l'exécution dans l'ordre, pour une performance parfois à peine meilleure
  - Même si la performance est plus élevée, la performance / consommation est souvent pire qu'avec l'exécution dans l'ordre

Microarchitectures SISD : In order

- Retour dans l'ordre
  - Superscalaire : On peut toujours extraire de l'ILP, on compte sur le compilateur pour l'exposer

```
addi x1, x2, 16
add x3, x1, x2
mul x4, x3, x1
addi x5, x2, 32
add x6, x5, x2
mul x4, x6, x1
```

IPC (InO): 1

```
addi x1, x2, 16
addi x5, x2, 32
add x3, x1, x2
add x6, x5, x2
mul x4, x3, x1
mul x4, x6, x1
```

IPC (InO): 2

Microarchitectures SISD : In order

- Retour dans l'ordre
  - Superscalaire : On peut toujours extraire de l'ILP, on compte sur le compilateur pour l'exposer
  - Spéculation : On peut toujours spéculer sur le flot de contrôle

Microarchitectures SISD : In order

- Retour dans l'ordre
  - Superscalaire : On peut toujours extraire de l'ILP, on compte sur le compilateur pour l'exposer
  - Spéculation : On peut toujours spéculer sur le flot de contrôle
- On a toujours besoin de matériel pour déterminer les dépendances dynamiquement
  - A chaque cycle, on teste « est-ce que les *n* instructions les plus vieilles peuvent s'exécuter » ?

Autres Architectures

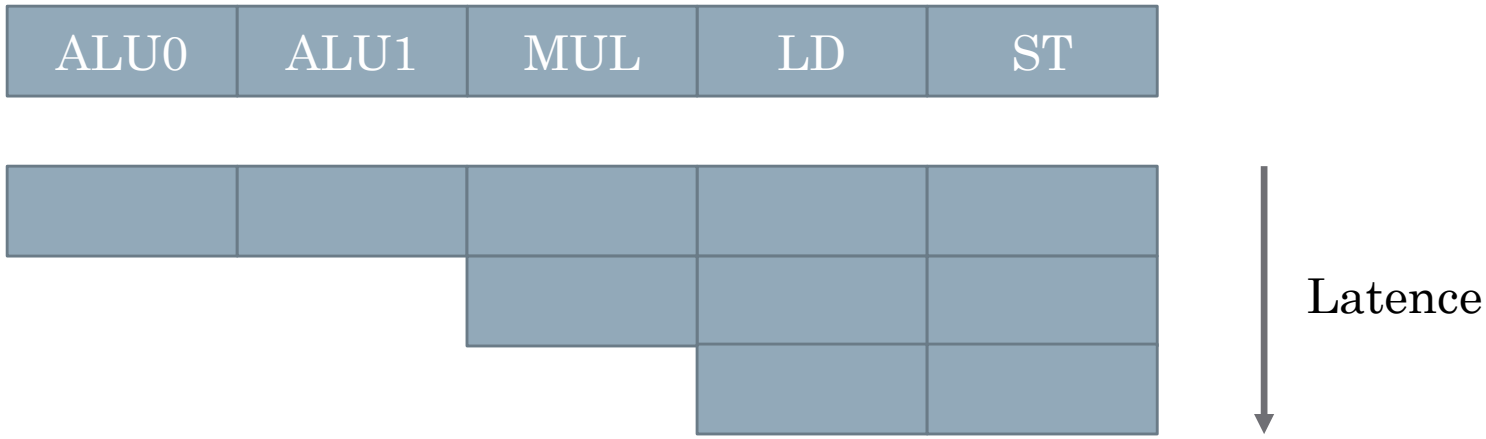
Very Large Instruction Word (VLIW)

Microarchitectures SISD : VLIW

- VLIW : « Very Large Instruction Word »
  - 1 instruction VLIW = plusieurs instructions « classiques »
  - Processeur avec de multiples unités fonctionnelles à **latence connue**
  - Partagent un fichier de registres

Microarchitectures SISD : VLIW

- Une instruction VLIW = plusieurs opérations
  - Format d'instruction liée à une machine spécifique (ici 5 unités)
  - Latences connues à la compilation
  - Mais potentiellement différentes



Indépendance et ISA

- ISA « scalaire » classique (RISV-V, Arm, x86)
  - Sémantique séquentielle (dans l'ordre du programme)
  - On ne peut pas exprimer « instruction A **ne dépend pas** de B »

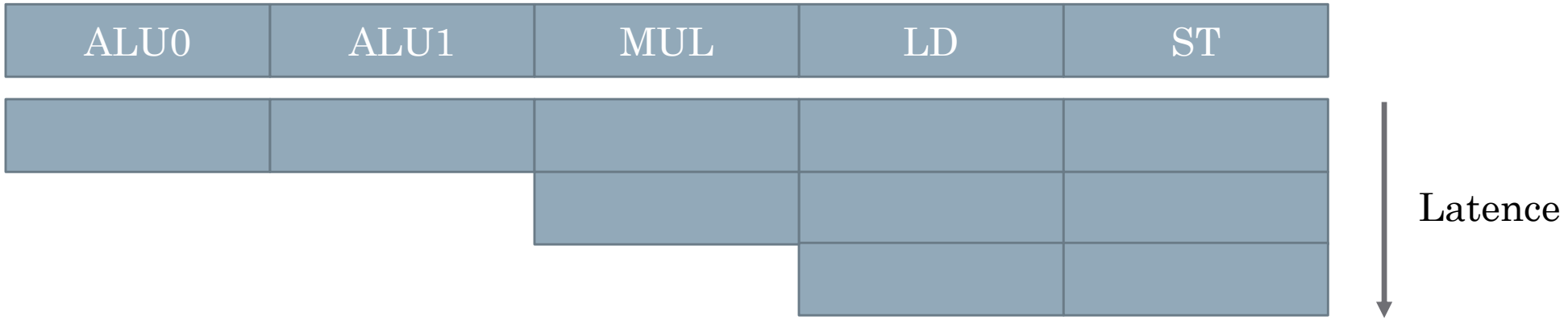
Indépendance et ISA

- ISA « scalaire » classique (RISV-V, Arm, x86)
  - Sémantique séquentielle (dans l'ordre du programme)
  - On ne peut pas exprimer « instruction A **ne dépend pas** de B »
- VLIW
  - Change le modèle d'exécution au niveau ISA
  - Les opérations qui sont dans la même instruction VLIW sont explicitement indépendantes
  - Chaque « voie » du processeur VLIW traite une donnée et une instruction différente => pas du SIMD
  - Mais un seul PC => toujours du SISD, juste une façon différente d'utiliser le parallélisme d'instruction (ILP)

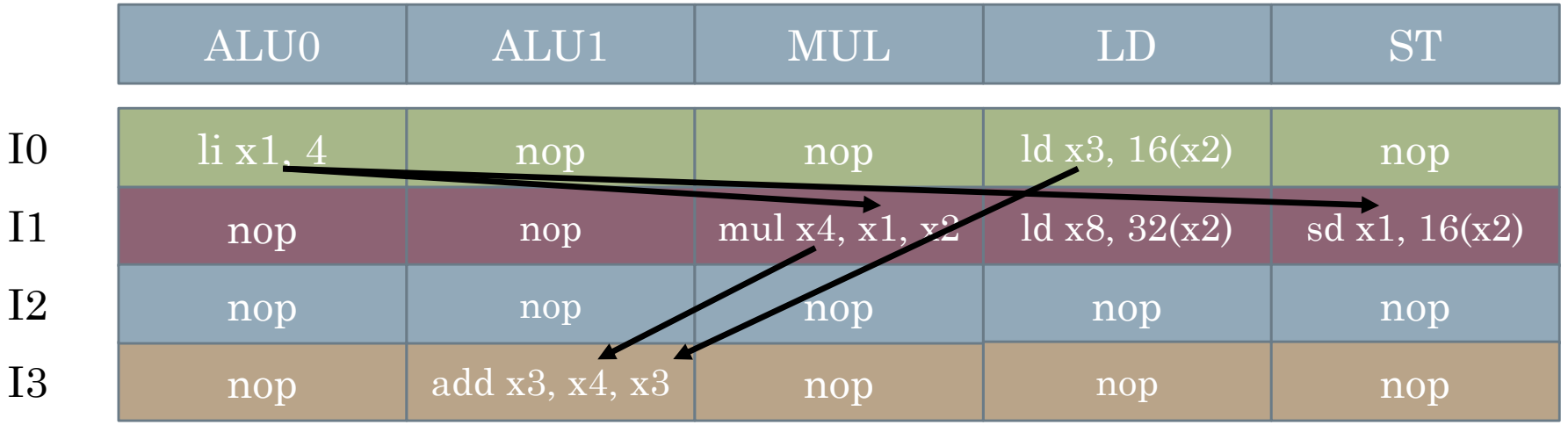
UAL vs. NUAL

- Unit Assumed Latency (latence = 1 cycle)
  - Chaque instruction VLIW est exécutée avant l'exécution de la suivante
  - Sémantique séquentielle conventionnelle
- Non-Unit Assumed Latency (latence = 1+ cycles)
  - Au moins une opération parmi les opérations de l'instruction VLIW prends plus d'un cycle (lat = *L*)
  - *L-1* instructions VLIW suivantes s'exécutent avant que le résultat soit disponible
    - Le compilateur doit donc s'assurer qu'aucune opération dans les *L-1* instructions VLIW suivantes n'a besoin du résultat

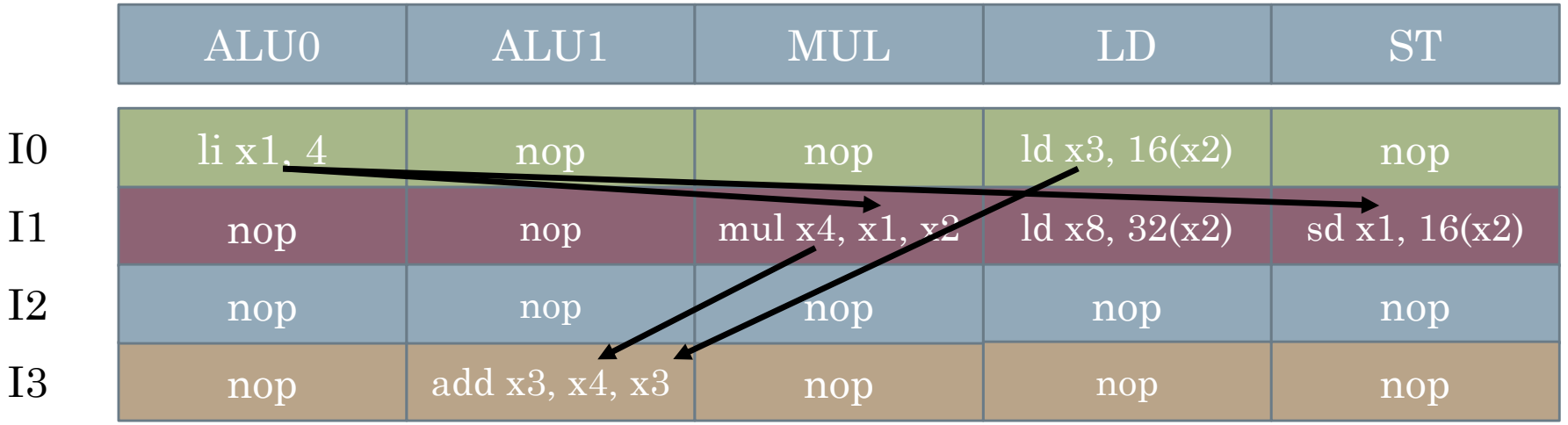
NUAL



NUAL

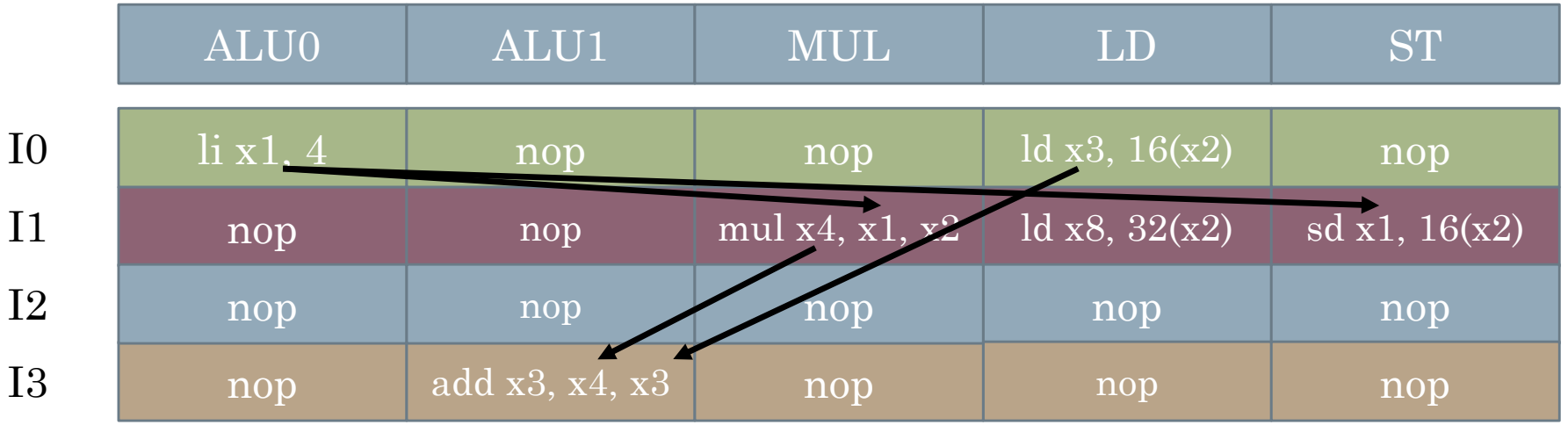

ALU0
ALU1
MUL
LD
ST
I0
li x1, 4
nop
nop
ld x3, 16(x2)
nop
I1
nop
nop
mul x4, x1, x2
ld x8, 32(x2)
sd x1, 16(x2)
I2
nop
nop
nop
nop
nop
I3
nop
add x3, x4, x3
nop
nop
nop

NUAL



- *li x1, 4* de I0 a une latence de 1
  - Le compilateur peut mettre un consommateur de x1 dans l'instruction suivante (I1, par exemple *mul x4, x1, x2)*

NUAL



- *li x1, 4* de I0 a une latence de 1
  - Le compilateur peut mettre un consommateur de x1 dans l'instruction suivante (I1, par exemple *mul x4, x1, x2)*
- *ld x3, 16(x2)* de I0 a une latence de 3
  - ➢Le compilateur ne peut pas mettre de consommateur de x3 avant 3 cycles (consommateur dans I3 au plus tôt)
    - ➢Le compilateur doit trouver d'autres opérations indépendantes pour remplir le pipeline

Architecture VLIW

- Les opérations d'une instruction VLIW sont indépendantes
  - Pas de matériel pour vérifier, le compilateur s'en charge
  - Latences connues
  - Pipelines non bloquants (sauf miss dans le cache de données)

- Le compilateur fait l'ordonnancement pour qu'une instruction qui s'exécute ait la garantie que ses opérandes sources sont disponibles

Architecture VLIW – Avantages

- La matériel n'a pas à « redécouvrir » les dépendances ou à « ré-ordonnancer » les instructions

- Implémentation simplifiée
  - Effort de vérification HW--
  - Surface--
  - Consommation--

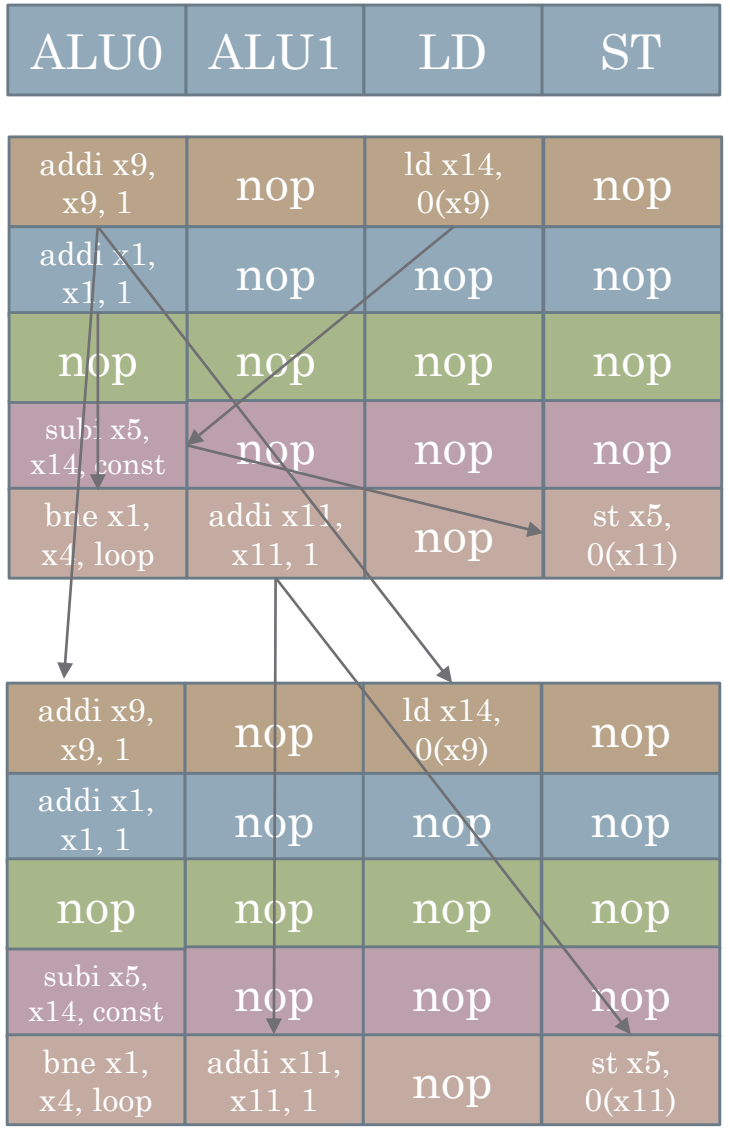
- Performance car parallélisme d'instruction (ILP)

Architecture VLIW

```
for(int i = 0; i < n; i++)
  C[i] = A[i] – const;
```

On considère :

- 3 cycles pour ld/st
- &A[i] dans x9
- &C[i] dans x11
- N dans x4
- i dans x1



Architecture VLIW

```
for(int i = 0; i < n; i++)
  C[i] = A[i] – const;
```

On considère :

- 3 cycles pour ld/st
- &A[i] dans x9
- &C[i] dans x11
- N dans x4
- i dans x1

ALU0	ALU1	LD	ST
addi x9, x9, 1	nop	ld x14, 0(x9)	nop
addi x1, x1, 1	nop	nop	nop
nop	nop	nop	nop
subi x5, x14, const	nop	nop	nop
bne x1, x4, loop	addi x11, x11, 1	nop	st x5, 0(x11)

1 élément / 5 cycles = 0,2 élément/cycle

Pas de dépendances à gérer pour le matériel car le compilateur a correctement ordonnancé les instructions (respect des latences et dépendances)

Cache miss lors d'un ld/st ?

- On bloque tout le pipeline dès qu'on trouve une opération dépendante
- On ne laisse pas les autres opérations de l'instruction VLIW avancer

Architecture VLIW

```
for(int i = 0; i < n; i++)
  C[i] = A[i] – const;
```

On considère :

- 3 cycles pour ld/st
- &A[i] dans x9
- &C[i] dans x11
- N dans x4
- i dans x1

ALU0	ALU1	LD	ST
addi x9, x9, 1	nop	ld x14, 0(x9)	nop
addi x1, x1, 1	nop	nop	nop
nop	nop	nop	nop
subi x5, x14, const	nop	nop	nop
bne x1, x4, loop	addi x11, x11, 1	nop	st x5, 0(x11)

Exercice: Faire mieux que 0,2 élément/cycle

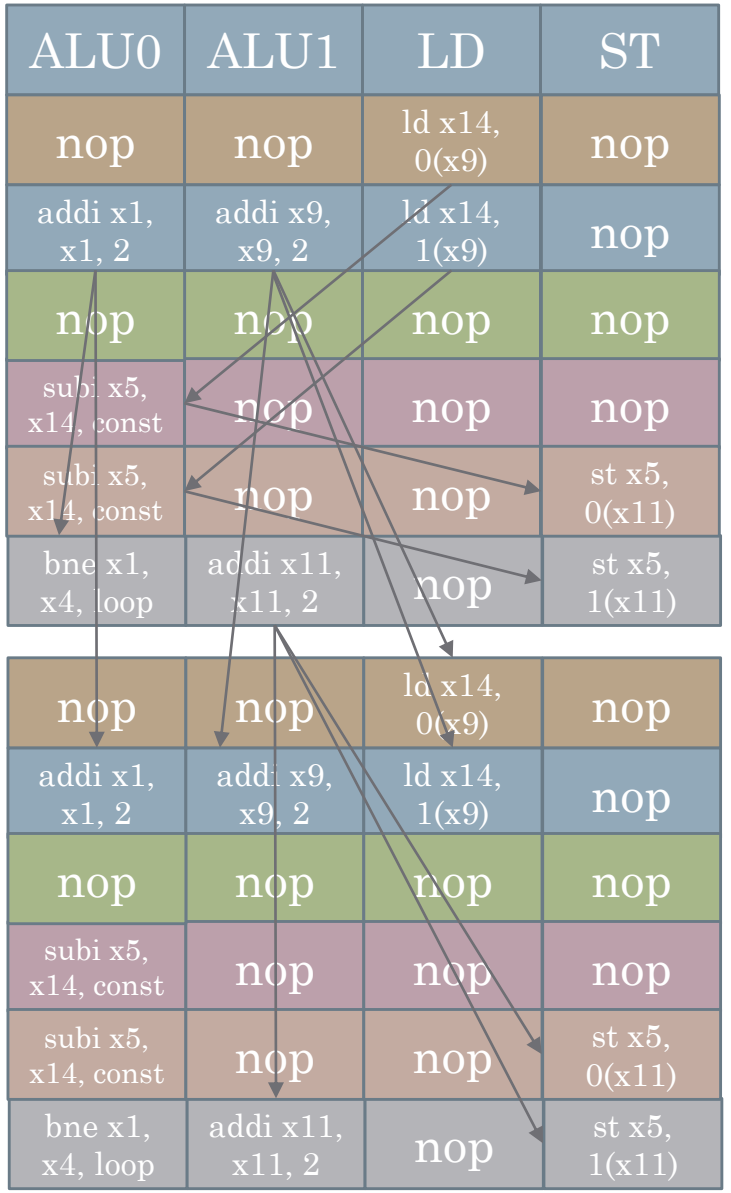
Architecture VLIW - Unrolling

```
for(int i = 0; i < n / 2; i+=2) {
  C[i] = A[i] – const;
  C[i+1] = A[i+1] – const;
}

// Epilogue
if(n % 2 != 0) {
  C[n-1] = A[n-1] – const;
}
```

- On minimise l'impact du branchement de boucle (1 test pour deux éléments)
- On doit ajouter un épilogue pour gérer le cas ou n est impair

2 éléments / 6 cycles = 0,33 élément/cycle



Architecture VLIW - Unrolling

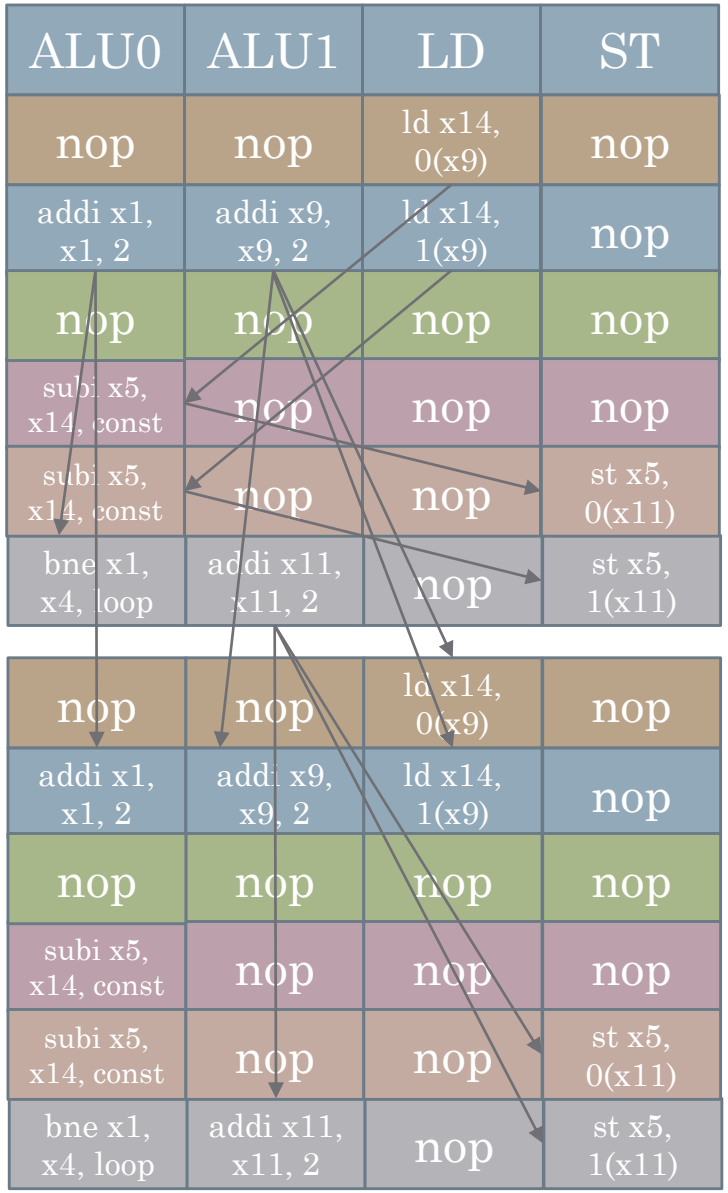
```
for(int i = 0; i < n / 2; i+=2) {
  C[i] = A[i] – const;
  C[i+1] = A[i+1] – const;
}

// Epilogue
if(n % 2 != 0) {
  C[n-1] = A[n-1] – const;
}
```

- On minimise l'impact du branchement de boucle (1 test pour deux éléments)
- On doit ajouter un épilogue pour gérer le cas ou n est impair

2 éléments / 6 cycles = 0,33 élément/cycle

Exercice: Mieux que 0,33 élément/cycle



Software Pipelining (SWP) + Unrolling

- Idée : Faire se chevaucher plusieurs itérations pour cacher les latences (ici trois, car ld prends trois cycles)

```
// Prologue (SWP)
char a = A[0], b = A[1], c = A[2]
int i = 3;

// Boucle (SWP Unrolled 3)
for(; i < n / 3; i+=3) {
   C[i-3] = a – const;
   C[i-2] = b – const;
   C[i-1] = c – const;
   a = A[i], b = A[i+1], c = A[i+2];
}

// Epilogue (SWP)
C[i – 3] = a – const;
C[i – 2] = b – const;
C[i – 1] = c – const;

// Epilogue (Unrolling)
For(int j = 0; j < n % 3; j++) {
 C[n-j-1] = A[n-j-1] – const;
}
```

Software Pipelining (SWP) + Unrolling

- Idée : Faire se chevaucher plusieurs itérations pour cacher les latences (ici trois, car ld prends trois cycles)

```
// Prologue (SWP)
char a = A[0], b = A[1], c = A[2]
int i = 3;

// Boucle (SWP Unrolled 3)
for(; i < n / 3; i+=3) {
   C[i-3] = a – const;
   C[i-2] = b – const;
   C[i-1] = c – const;
   a = A[i], b = A[i+1], c = A[i+2];
}

// Epilogue (SWP)
C[i – 3] = a – const;
C[i – 2] = b – const;
C[i – 1] = c – const;

// Epilogue (Unrolling)
For(int j = 0; j < n % 3; j++) {
 C[n-j-1] = A[n-j-1] – const;
}
```

ALU0	ALU1	LD	ST
nop	subi x23, x20, const	ld x20, 0(x9)	Nop
addi x1, x1, 3	subi x24, x21, const	ld x21, 1(x9)	st x23, 0(x11)
addi x9, x9, 3	subi x25, x22, const	ld x22, 2(x9)	st x24, 1(x11)
bne x1, x4, loop	addi x11, x11, 3	nop	st x25, 3(x11)

Software Pipelining (SWP) + Unrolling

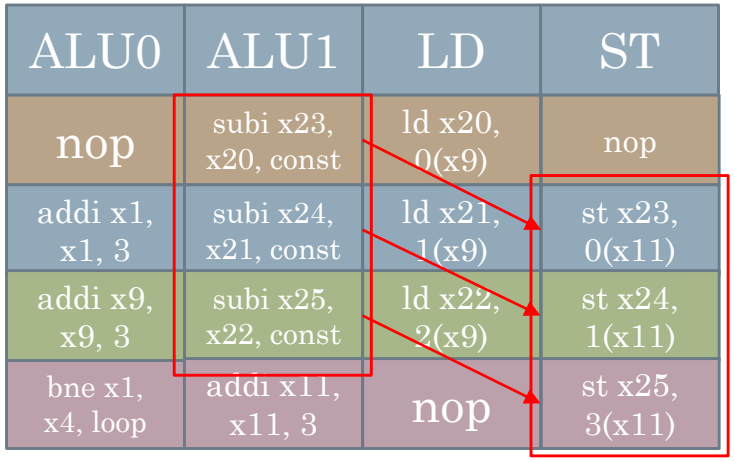
- Idée : Faire se chevaucher plusieurs itérations pour cacher les latences (ici trois, car ld prends trois cycles)

```
// Prologue (SWP)
char a = A[0], b = A[1], c = A[2]
int i = 3;

// Boucle (SWP Unrolled 3)
for(; i < n / 3; i+=3) {
   C[i-3] = a – const;
   C[i-2] = b – const;
   C[i-1] = c – const;
   a = A[i], b = A[i+1], c = A[i+2];
}

// Epilogue (SWP)
C[i – 3] = a – const;
C[i – 2] = b – const;
C[i – 1] = c – const;

// Epilogue (Unrolling)
For(int j = 0; j < n % 3; j++) {
 C[n-j-1] = A[n-j-1] – const;
}
```



Software Pipelining (SWP) + Unrolling

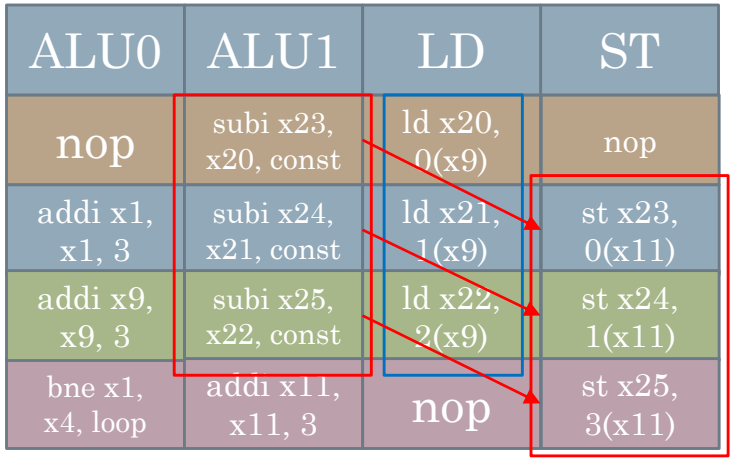
- Idée : Faire se chevaucher plusieurs itérations pour cacher les latences (ici trois, car ld prends trois cycles)

```
// Prologue (SWP)
char a = A[0], b = A[1], c = A[2]
int i = 3;

// Boucle (SWP Unrolled 3)
for(; i < n / 3; i+=3) {
   C[i-3] = a – const;
   C[i-2] = b – const;
   C[i-1] = c – const;
   a = A[i], b = A[i+1], c = A[i+2];
}

// Epilogue (SWP)
C[i – 3] = a – const;
C[i – 2] = b – const;
C[i – 1] = c – const;

// Epilogue (Unrolling)
For(int j = 0; j < n % 3; j++) {
 C[n-j-1] = A[n-j-1] – const;
}
```



Software Pipelining (SWP) + Unrolling

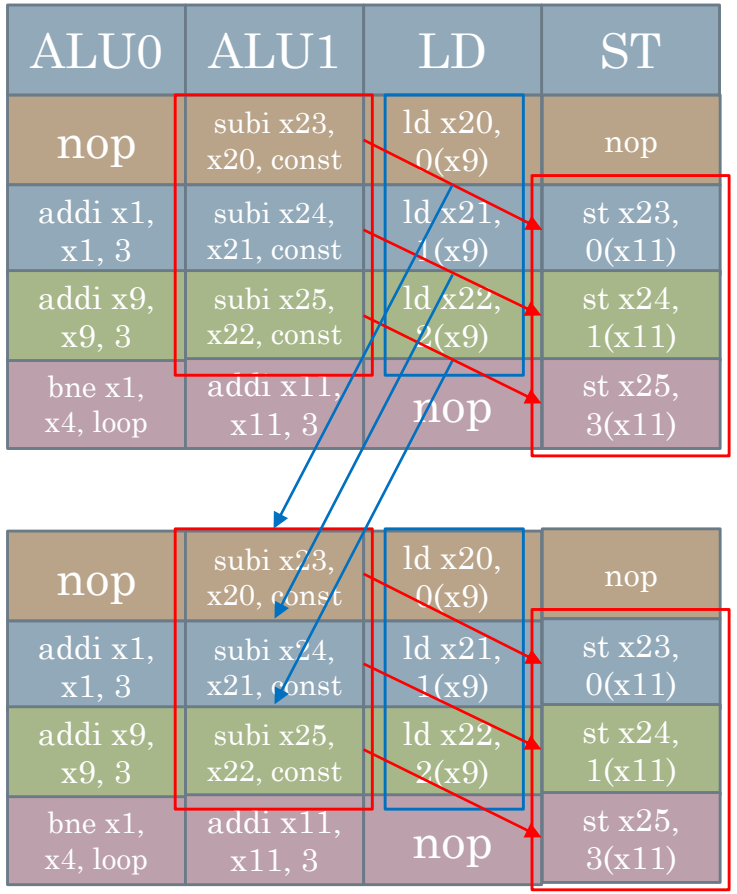
- Idée : Faire se chevaucher plusieurs itérations pour cacher les latences (ici trois, car ld prends trois cycles)

```
// Prologue (SWP)
char a = A[0], b = A[1], c = A[2]
int i = 3;

// Boucle (SWP Unrolled 3)
for(; i < n / 3; i+=3) {
   C[i-3] = a – const;
   C[i-2] = b – const;
   C[i-1] = c – const;
   a = A[i], b = A[i+1], c = A[i+2];
}

// Epilogue (SWP)
C[i – 3] = a – const;
C[i – 2] = b – const;
C[i – 1] = c – const;

// Epilogue (Unrolling)
For(int j = 0; j < n % 3; j++) {
 C[n-j-1] = A[n-j-1] – const;
}
```



Software Pipelining (SWP) + Unrolling

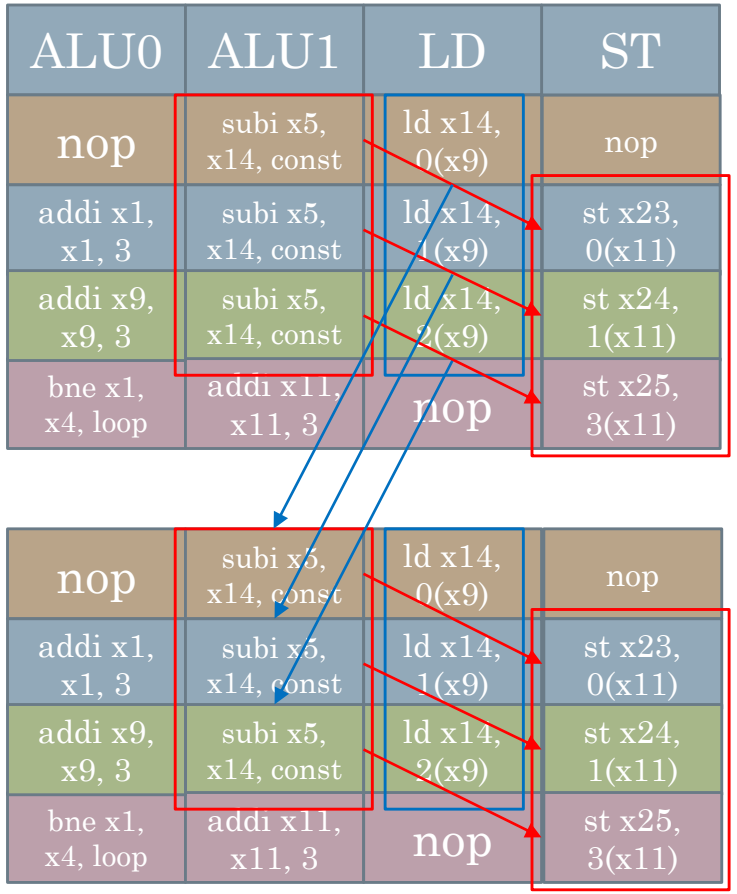
- Idée : Faire se chevaucher plusieurs itérations pour cacher les latences (ici trois, car ld prends trois cycles)

```
// Prologue (SWP)
char a = A[0], b = A[1], c = A[2]
int i = 3;

// Boucle (SWP Unrolled 3)
for(; i < n / 3; i+=3) {
   C[i-3] = a – const;
   C[i-2] = b – const;
   C[i-1] = c – const;
   a = A[i], b = A[i+1], c = A[i+2];
}

// Epilogue (SWP)
C[i – 3] = a – const;
C[i – 2] = b – const;
C[i – 1] = c – const;

// Epilogue (Unrolling)
For(int j = 0; j < n % 3; j++) {
 C[n-j-1] = A[n-j-1] – const;
}
```



3 éléments en 4 cycles = 0,75 élément/cycle !

Architecture VLIW

- Le compilateur ou le programmeur doit utiliser des techniques *logicielles* pour augmenter la performance
  - Unrolling pour limiter des branchements
  - *Software pipelining (SWP)* pour maximiser l'utilisation
    - On ne peut pas tout faire, surtout si il y a du flot de contrôle complexe dans le corps de boucle

- Unrolling aussi applicable aux processeurs classiques InO et OoO (SWP inutile car OoO le fait déjà).

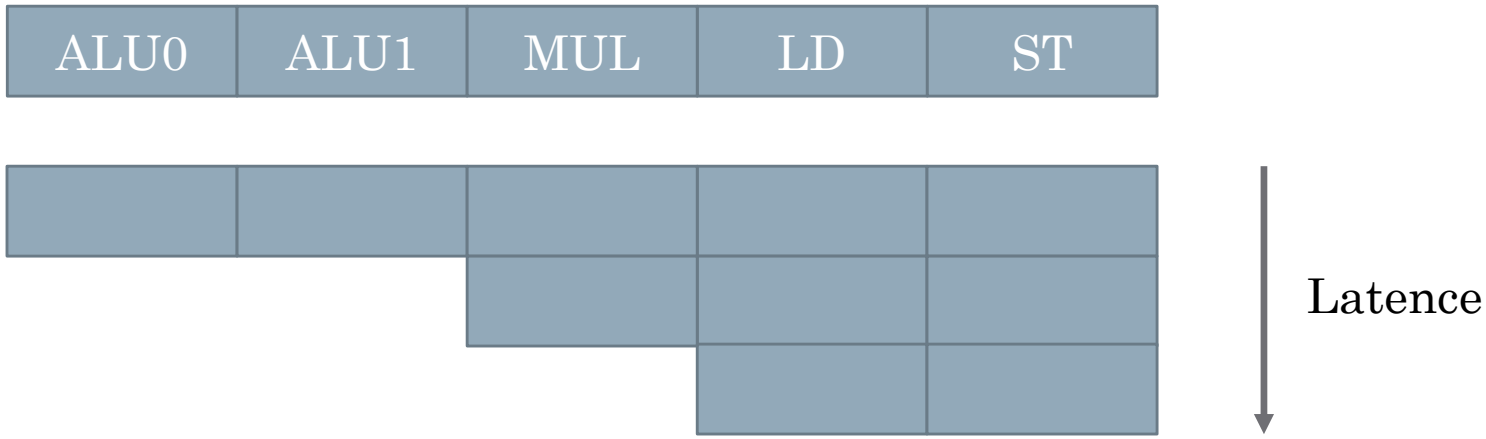
Software Pipelining (+ Unrolling)

- Unrolling vs. Software Pipelining
  - Unrolling simple paie une pénalité à chaque début/fin de tour de boucle
  - SWP paie une pénalité une seule fois (début/fin de boucle)
  - SWP pour les machines dans l'ordre (OoO fait du SWP en matériel)
- Combinaison des deux pour maximiser l'utilisation

VLIW - Désavantages

- Le compilateur doit connaître la microarchitecture (#FUs, latences)
  - Non portable (il faut recompiler le code si on veut le faire tourner sur un autre VLIW)
  - Les évènements non prévisibles à la compilation (cache miss) bloquent complètement le processeur
- Performance moindre que OoO
  - OoO ne bloque pas tant qu'il y a des instructions indépendantes
  - OoO fait du software pipelining en matériel via l'ordonnancement dynamique
- Densité du code
  - Les dépendances empêchent de toujours occuper toutes les FUs à chaque cycle
  - ➢ En pratique, beaucoup de *nop* dans les instructions VLIW

VLIW - Résumé



- Opérations dans une instruction VLIW sont indépendantes
- Latences connues statiquement (sauf load/store)
- Le matériel ne vérifie pas les dépendances
- Le compilateur s'assure que l'ordonnancement est correct et est responsable de la performance

Plusieurs types de parallélisme

- SISD (ILP)
  - Facile à programmer
  - Performance la « moins pire » pour les programmes avec flot de contrôle complexe

Ce qu'on a vu jusqu'ici (InO, OoO, **VLIW**)

- SIMD (DLP, ILP)
  - Un peu moins facile à programmer
  - Performance++ si le programme a en effet du DLP

Ce qu'on va voir maintenant

- MIMD (TLP, ILP)
  - Difficile à programmer (synchronisation)
  - Performance++ si le programme a en effet du TLP

Autres Architectures

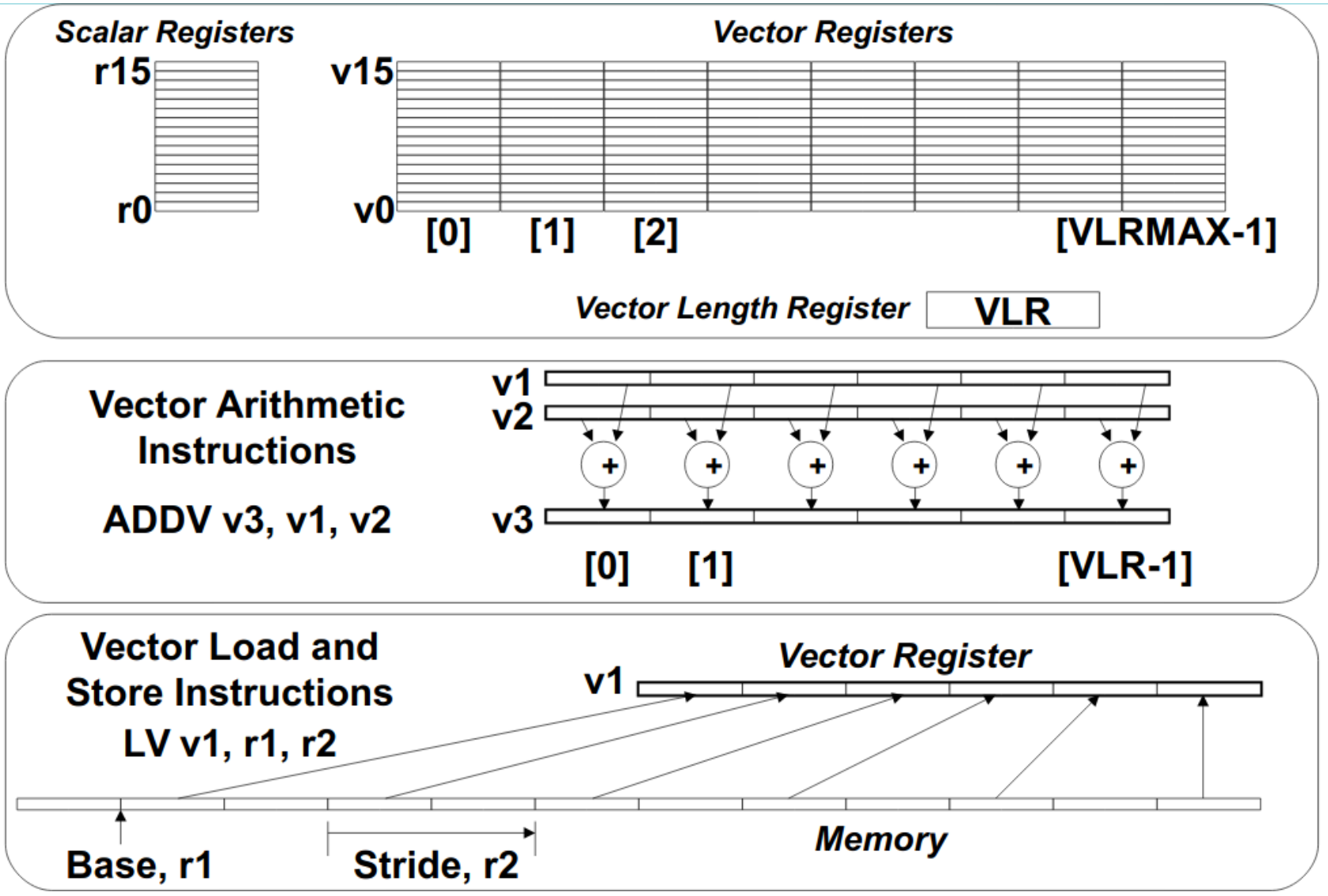
Machines vectorielles et SIMD

Machines vectorielles

- Dans les années 60-70-80, les supercalculateurs étaient construit à base de machines vectorielles (SIMD)
- Plus maintenant (MIMD domine), mais certains supercalculateurs embarquent des GPU (SIMD) pour accélérer les tâches qui s'y prêtent.

- On va tenter de mettre en lumières les différences entre machines vectorielles et généralistes.

Modèle de programmation

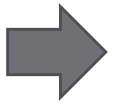


Vectorisation – Exemple (4 elts/vecteur)

```
for(i = 0; i < N; i++)
   A[i] = B[i] + C[i]
```

Vectorisation – Exemple (4 elts/vecteur)

```
for(i = 0; i < N; i++)
  A[i] = B[i] + C[i]
```

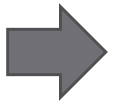


Scalaire : N branchements

```
LD r0, [r1] // r0 = N
LD r2, 0x0 // r2 = i = 0
loop:
  LD r6, [r3] // r6 = B[i]
  LD r7, [r4] // r7 = C[i]
  ADD r6, r7, r6
  ST r6, [r5] // A[i] = r6
  ADD r2, 1
  ADD r3, 1
  ADD r4, 1
  ADD r5, 1
  BNE r2, r0, loop
```

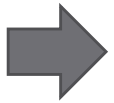
Vectorisation – Exemple (4 elts/vecteur)

```
for(i = 0; i < N; i++)
  A[i] = B[i] + C[i]
```



Scalaire : N branchements

```
LD r0, [r1] // r0 = N
LI r2, 0x0 // r2 = i = 0
loop:
  LD r6, [r3] // r6 = B[i]
  LD r7, [r4] // r7 = C[i]
  ADD r6, r7, r6
  ST r6, [r5] // A[i] = r6
  ADD r2, 1
  ADD r3, 1
  ADD r4, 1
  ADD r5, 1
  BNE r2, r0, loop
```



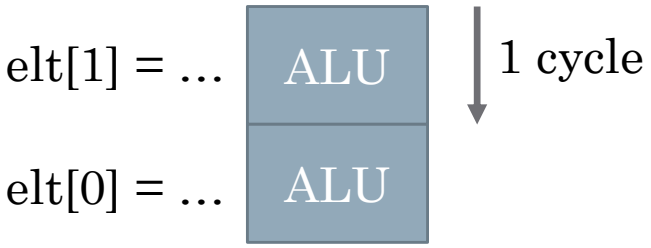
Vectoriel : N/4 branchements

```
LD r0, [r1] // r0 = N
LI r2, 0x0 // r2 = i = 0
loop:
  LV v0, [r3] // v0 = B[i..i+3]
  LV v1, [r4] // v1 = C[i..i+3]
  ADDV v0, v0, v1
  SV v0, [r5] // A[i..i+3] = v0
  ADD r2, 4
  ADD r3, 4
  ADD r4, 4
  ADD r5, 4
  BNE r2, r0, loop
```

Machines vectorielles

- On économise de la bande passante au niveau contrôle
  - Une seule instruction vectorielle traite 4/8/16+ éléments = Une seule instruction à récupérer au lieu de 4/8/16+
- Au sein de la microarchitecture, la machine peut très bien n'avoir que des unités scalaires pipelinées. Par exemple, avec 2 elts/vecteur (abstraction arch/uarch)

Unité scalaire :
Plusieurs cycles pour traiter le vecteur



Unité vectorielle :
Un seul cycle pour traiter le vecteur

Vectorisation – Accès non unitaire avec « pas » (Stride)

```
for(i = 0; i < N; i++)
   A[i * 2] = B[i * 2] + C[i * 2]
```

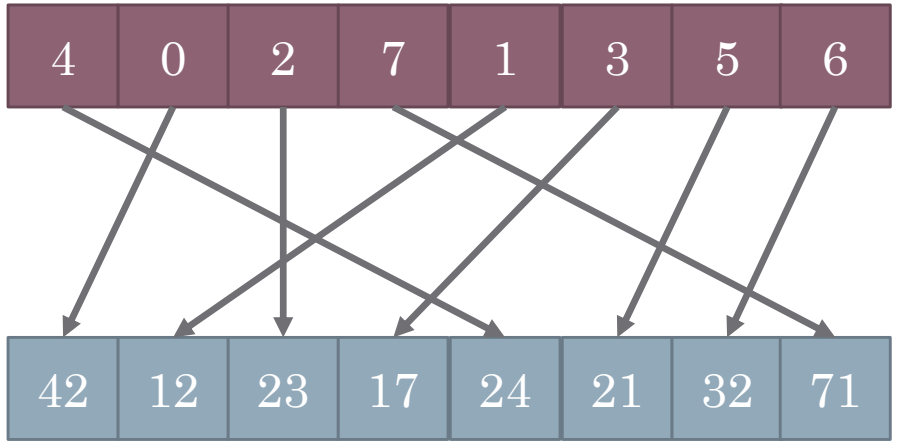
```
LD r0, [r1] // r0 = N
LI r2, 0x0 // r2 = i = 0
LI r8, 2 // r8 = « pas » = 2
loop:
  LV v0, [r3], r8 // v0 = B[i, i+2, i+4, i+6]
  LV v1, [r4], r8 // v1 = C[i, i+2, i+4, i+6]
  ADDV v0, v0, v1
  SV v0, [r5], r8 // A[i, i+2, i+4, i+6] = v0
  ADD r2, 4
  ADD r3, 4
  ADD r4, 4
  ADD r5, 4
  BNE r2, r0, loop
```

Vectorisation – Accès irréguliers

```
for(i = 0; i < N; i++)
    A[i] = B[i] + C[D[i]];
```

D : Parcours séquentiel

C : Parcours aléatoire (indexé)



Vectorisation – Accès irréguliers

```
for(i = 0; i < N; i++)
    A[i] = B[i] + C[D[i]];
```

- « Indexed load » => Gather

```
LV vD, rD           # Load indices in D vector
LVI vC, rC, vD      # Load indirect from rC base
LV vB, rB           # Load B vector
ADDV.D vA, vB, vC   # Do add
SV vA, rA           # Store result
```

Vectorisation – Accès irréguliers

```
for(i = 0; i < N; i++)
  A[B[i]]++;
```

- « Indexed store » => Scatter

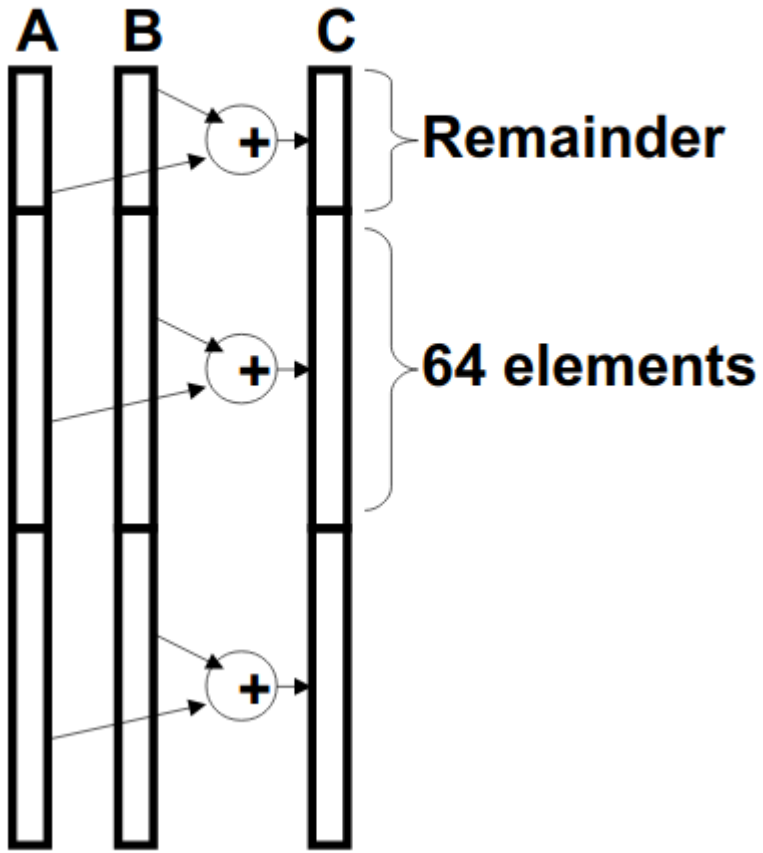
```
LV vB, rB         # Load indices in B vector
LVI vA, rA, vB    # Gather initial A values
ADDV vA, vA, 1    # Increment
SVI vA, rA, vB    # Scatter incremented values
```

Machines vectorielles - Avantages

- Compact
  - 1 op vectorielle = *n* ops scalaires
- Exprime au matériel que ces *n* ops
  - Sont indépendantes : **Data Level Parallelism**
  - Utilisent la même unité fonctionnelle
  - Accèdent à des données disjointes
  - Accèdent à un bloc de mémoire contigu (stride = 1)
  - Accèdent à un bloc de mémoire avec un motif régulier (stride > 1)
- Passe à l'échelle
  - En augmentant *n* (ajout de voies)

Difficultés – Taille de vecteur

```
for (i=0; i<N; i++)
    C[i] = A[i]+B[i];
```



- Taille des opérations vectorielles ne correspond pas toujours à la taille du problème
  - Opérations scalaires pour finaliser une boucle

Difficultés – Flot de contrôle

```
for(i = 0; i < N; i++) {
  if (A[i] > 0) {
   A[i] = B[i];
  }
}
```

- Problème de *divergence* du flot de contrôle au sein des voies d'un même vecteur
- Solution : Vecteur de *masques* (1 bit par voie)
  - Si masque == 0 alors l'opération sur l'élément devient un *nop*
  - Equivalent vectoriel de la **prédication** (voir cours prédiction de branchement)

```
CVM              # Turn on all elements
LV vA, rA        # Load entire A vector
SGTVS.D vA, F0   # Set bits in mask register where A>0
LV vA, rB        # Load B vector into A under mask
SV vA, rA        # Store A back to memory under mask
```

Vectorisation automatique

- Evident sur certaines boucles

```
for(i = 0; i < N; i++)
C[i] = A[i] + B[i]
```

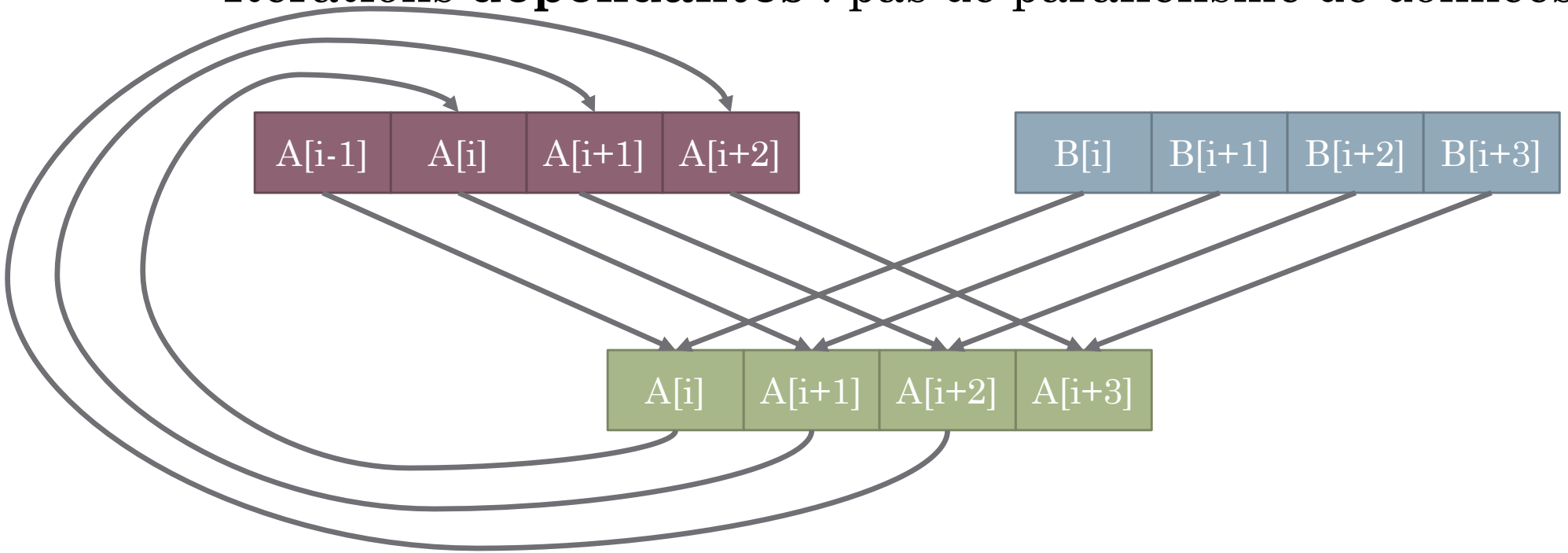
- Itérations **indépendantes**, pas de flot de contrôle dans la boucle
- Il faut juste gérer le cas ou N n'est pas un multiple de N

Vectorisation automatique

- Pas évident sur d'autres boucles

```
for(i = 0; i < N; i++)
  A[i] = A[i-1] + B[i]
```

- Itérations **dépendantes** : pas de parallélisme de données

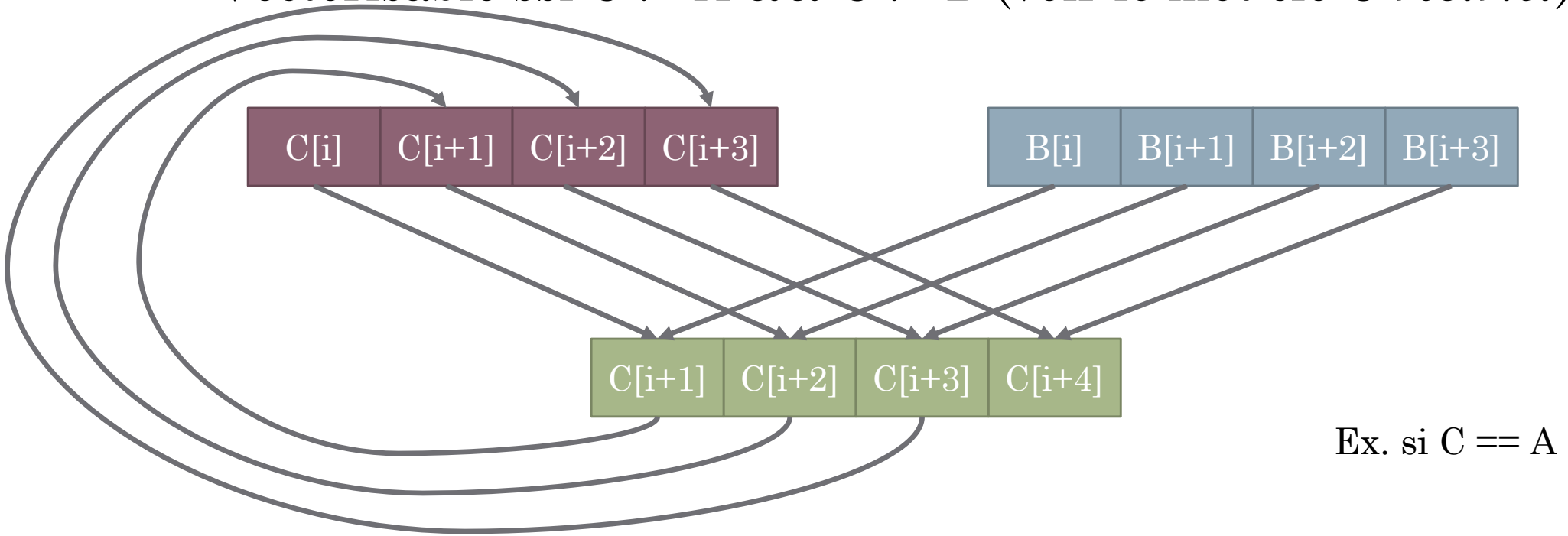


Vectorisation automatique

- Ou encore

```
void add_array(int * A, int * B, int * C) {
  for(i = 0; i < size - 1; i++)
    C[i+1] = A[i] + B[i];
}
```

- Vectorisable ssi C != A && C != B (voir le mot clé C *restrict*)



Ex. si C == A

Vectorisation automatique

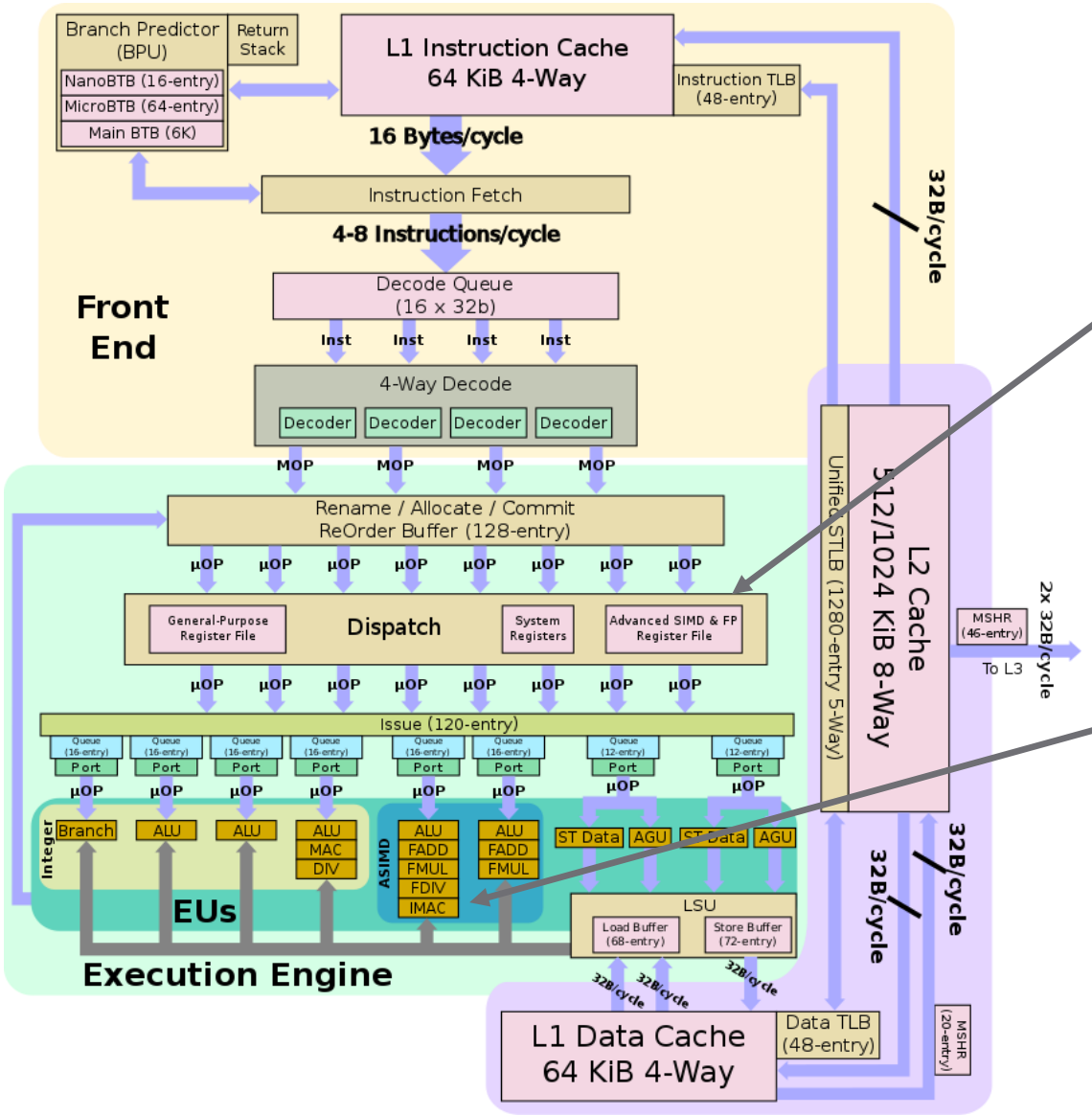
- Le compilateur doit pouvoir prouver certaines propriétés pour pouvoir vectoriser automatiquement
- Sinon : à la main

- Toutes les boucles ne se prêtent pas au paradigme vectoriel
  - Vectoriel => « special purpose architecture » par rapport aux processeurs « general purpose »

Autres Architectures

Unité SIMD dans un processeur SISD

Retour à une machine SISD (ex. ARM N1)



Advanced SIMD Register File ?

Advanced SIMD ?

Unités vectorielles

- Les processeurs généralistes modernes possèdent une unité vectorielle (SIMD) pour accélérer les calculs qui le permettent
  - Nouvelles instructions dédiées
  - Souvent plusieurs taille d'éléments supportés (8b, 16b, 32b, 64b)
  - Registres arch. vectoriels dédiés
  - Taille des vecteurs : 128-, 256-, **512-bits** (Intel AVX512, notamment), scalaire seulement 64-bit

Extensions SIMD

- En général plus limitées que les jeux d'instructions vectoriels complets
  - Pas de strided load/store ni scatter/gather (même si scatter/gather arrivé avec AVX2/AVX512 chez x86)
  - Registres de prédicats (arrivés avec AVX512 chez x86)
  - Taille des registres vectoriels limitée
  - Nombre de registres limité

- Mais un gain en performance élevé pour les problèmes qui s'y prêtent

Exemple – Optimisons memcpy

void * memcpy (void * restrict destination, const void * restrict source, size_t num);

1. Copie “num” octets depuis “source” vers “destination”
2. Résultat indéfini si source et destination se chevauchent (*restrict)*

Exercice : Votre meilleure implementation de memcpy en C

Exemple – Optimisons memcpy

void * memcpy (void * restrict destination, const void * restrict source, size_t num);

```
void * memcpy (void * restrict destination, const void * restrict source, size_t num);
{
    for(size_t i= 0; i < num; i++)
    {
        ((char*) destination)[i] = ((char*) source)[i];
    }
    return destination;
}
```

- (4 * num) + 3 instructions
- Des idées pour faire mieux ?

```
x2 ← source
x3 ← destination
x4 ← source + num

loop:
  ldrb x1, [x2], #1 // ld byte
  strb x1, [x3], #1 // st byte
  sub x6, x4, x2 // i < num
  bnz x6, loop
```

Note : On est passé sur de l'assembleur Armv8

Exemple – Optimisons memcpy

- En utilisant toute la largeur du registre scalaire (« SIMD implicite »)

void * memcpy (void * restrict destination, const void * restrict source, size_t num);

```
void * memcpy (void * restrict destination, const void * restrict source, size_t num);
{
    for(int i= 0; i < num; i+=8) {
        ((uint64_t*) destination)[i] = ((uint64_t*) source)[i];
    }
    for(int 0; i < num % 8; i++) {
        ((char*) destination)[num – i – 1] =
            ((char*) source)[num – i – 1];
    }
    return destination;
}
```

```
x2 ← source
x3 ← destination
x4 ← source + num

loop:
  ldr x1, [x2], #8 // ld dw
  str x1, [x3], #8 // st dw
  sub x6, x4, x2 // i < num
  bnz x6, loop
```

- 4 * (num / 8) + 3 instrs
  - ~8 fois moins d'instructions
- Encore mieux ?

SIMD « Implicite »

- Note : On peut faire du SIMD « implicite » avec certaines instructions scalaires lorsque la taille de l'élément < taille du registre scalaire

Scalaire (elt = byte)

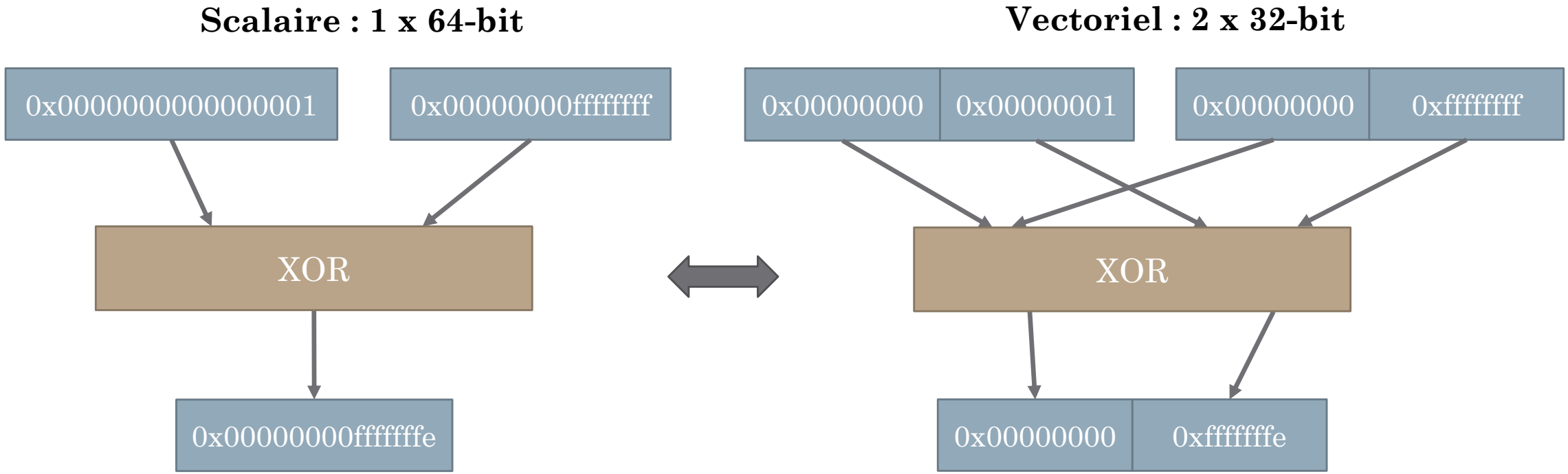
```
loop:
  ldb x1, [x2 + x4] // Load 1B
  stb x1, [x3 + x4] // Store 1B
  addi x4, x4, #1 // 1 tours de boucle
  subs x4, #1
  bnz x6 loop
```

“"Vectoriel” (techniquement scalaire avec elt = doubleword)

```
loop:
  ldr x1, [x2 + x4] // Load 8B
  str x1, [x3 + x4] // Store 8B
  addi x4, x4, #8 // “8” tours de boucle
  subs x4, #8
  bnz x6 loop
```

SIMD « Implicite »

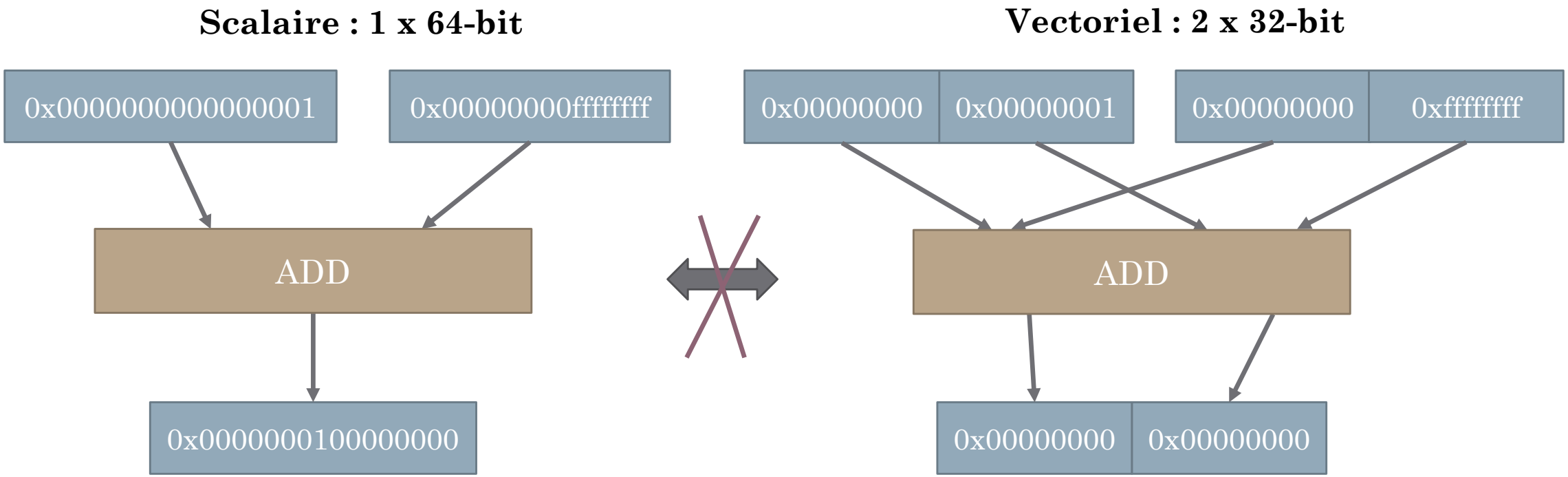
- SIMD « implicite » lorsque l'opération le permet, c'est-à-dire quand les éléments sont **indépendants**
  - On programme avec des instructions **scalaires**, mais on interprète un scalaire de taille x comme plusieurs elts de taille y
  - Logique bit à bit (and, xor, not, or)



SIMD « Implicite »

- SIMD « implicite » lorsque l'opération le permet, c'est-à-dire quand les éléments sont **indépendants**
  - On programme avec des instructions **scalaires**, mais on interprète un scalaire de taille x comme plusieurs elts de taille y
  - Logique bit à bit (and, xor, not, or)
  - Loads/Stores
  - **Pas l'arithmétique (add/sub, shift, etc.)**

SIMD « Implicite »



- Certaines opérations ne peuvent pas se vectoriser en utilisant une taille d'élément plus petite et des instructions scalaires : Il faut des instructions **dédiées**

SIMD « Implicite »

- SIMD « implicite » lorsque l'opération le permet, c'est-à-dire quand les éléments sont **indépendants**
  - On programme avec des instructions **scalaires**, mais on interprète un scalaire de taille x comme plusieurs elts de taille y
  - Logique bit à bit (and, xor, not, or)
  - Loads/Stores
  - **Pas l'arithmétique (add/sub, shift, etc.)**

- Les données doivent être **indépendantes**
  - SIMD « implicite » dans un vecteur scalaire Add[i] = Src1[i] + Src2[i] + **Carry[i-1]**
- Carry dépend de l'opérateur appliqué à d'autres bits : pas de parallélisme SIMD dans l'opérateur lui-même

Exemple – Optimisons memcpy

- En utilisant l'unité vectorielle des processeurs Arm (ici, Neon 128-bit)

void * memcpy (void * restrict destination, const void * restrict source, size_t num);

```
void * memcpy ( void * destination, const void * source, size_t num );
{
    for(int i= 0; i < num; i+=16) {
        ((int64x2_t*) destination)[i] = ((int64x2_t*) source)[i];
    }
    for(int 0; i < num % 16; i++) {
        ((char*) destination)[num – i – 1] =
            ((char*) source)[num – i – 1];
    }
    return destination;
}
```

- 4 * (num / 16) + 3 instrs
  - ~16 fois moins d'instructions
  - Si num multiple de 16 (cas considéré ici)
- Encore mieux avec des registres plus larges

```
x2 ← source
x3 ← destination
x4 ← 0

loop:
  ld1 v0.2d, [x2], #16 // ld qw
  st1 v0.2d, [x3], #16 // st qw
  sub x6, x4, x2 // i < num
  bnz x6, loop
```

SIMD dans un processeur généraliste

- Unités/Registres/Bypass dédiés
  - Surface non négligeable
  - Consommation non-négligeable
  - Sert aussi (en général) d'unité de calcul flottant
- Requiert dans la majorité des cas une réécriture du code pour en bénéficier
- Compilateur tente de vectoriser mais c'est une tâche difficile

SIMD dans un processeur généraliste

- Un “accélérateur” SIMD pour le processeur généraliste, mais pas un coprocesseur
- Intégré dans les CPUs généralistes au fil du temps
  - Cache L2 (puis L3)
  - Unités de calcul flottant
  - Contrôleur mémoire
  - Accélérateur crypto
  - etc.

Questions ?

Autres Architectures

SEOC3A – CEAMC

Arthur Perais (arthur.perais@univ-grenoble-alpes.fr)