# SWP
## Sub Word Parallelism

Lionel Lacassagne

Sorbonne University / LIP6 / ALSOC

https://www.lip6.fr/

lionel.lacassagne@lip6.fr

# SWP & ALU

## les ALU scalaires sont parallèles

- Historique: les DSP (*Digital Signal Processor*)
  - constat (en général) : image = 8 bits, signal = 16 bits
  - $\Rightarrow$ 4 données 8 bits ou 2 données 16 bits dans un registre 32 bits
  - ajout d'instructions spéciales pour traiter ces formats
  - gain: 1 LOAD 32 bits permet de charger plus que une donnée ...
  - exemples: TI TMS320C6x et TMS320C8x: 4 et 6 unités arithmétiques.
- Les GPP (*General Purpose Processors*) aka CPU
  - architecture 32 (voire) 64 bits = registres + ALU + bus
  - les opérations logiques sont intrinsèquement parallèles:
  - 1 registre = $1\times32$ ou $2\times16$ ou $4\times8$ ou ... $32\times1$

# SWP & ALU: adaptation

## le *full adder*

- le *hall adder* 1 bit
- le *full adder* 1 bit
  - constat (en général) : image = 8 bits, signal = 16 bits
  - $\Rightarrow$ 4 données 8 bits ou 2 données 16 bits dans un registre 32 bits
  - ajout d'instructions spéciales pour traiter ces formats
  - gain: 1 LOAD 32 bits permet de charger plus que une donnée ...
  - exemples: TI TMS320C6x et TMS320C8x: 4 et 6 unités arithmétiques.

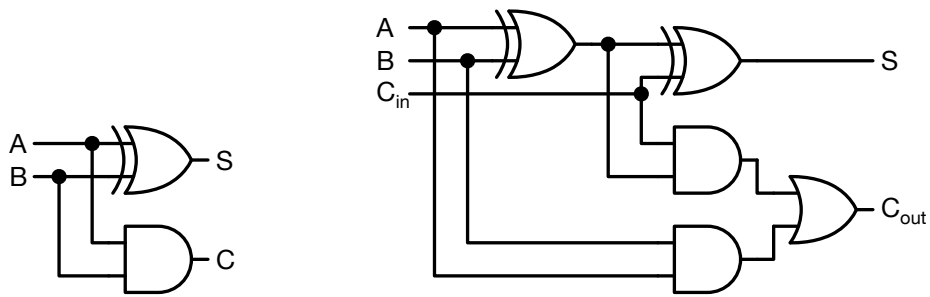
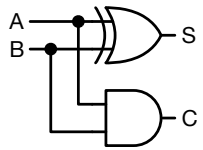



Figure: *half-adder* et *full-adder* 1 bit

# *Half-adder* & full-adder

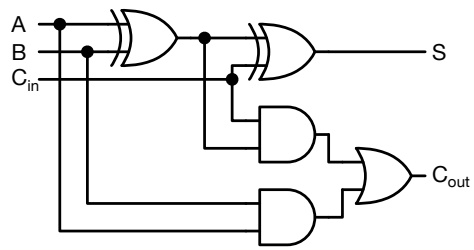
- *Half-adder* 1 bit
  - inputs 1 bit: $A$, $B$
  - outputs 1 bit: $S$, $C$
  - $S = A \oplus B$, $C = A + B$

| $A$ | $B$ | $C$ | $S$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

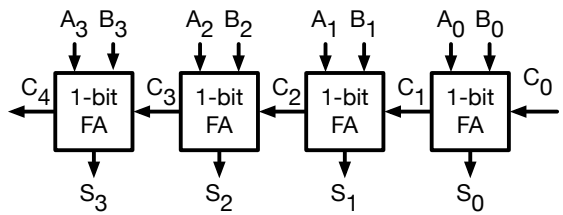



- *Full-adder* 1 bit
  - inputs 1 bit: $A$, $B$, $C_{in}$
  - outputs 1 bit: $S$, $C_{out}$
  - $S = A \oplus B \oplus C$
  - $C_{out} = (A \cdot B) + (C_{in} \cdot (A \oplus B))$

| $A$ | $B$ | $C_{in}$ | $C_{out}$ | $S$ |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

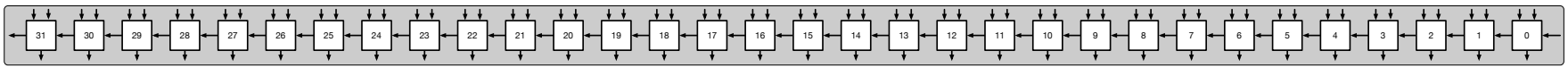



- *Full-adder* n bits
  - n $\times$ *full-adders* 1 bit
  - chainage des retenues
  - exemple : *full-adder* 4 bits

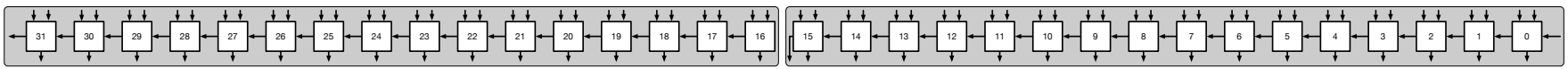


# *Full-adder* 32 bits

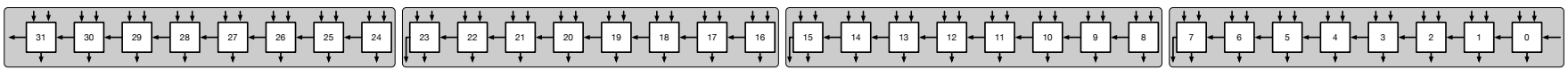
- 1×32 bits



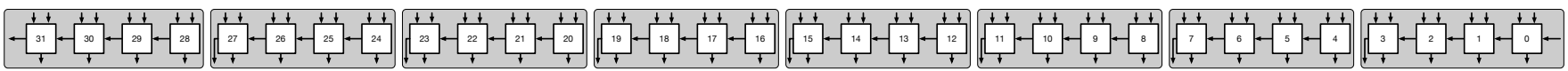

- 2×16 bits




- 4×8 bits




- 8×4 bits




- Le même *full-adder*, mais configuré de 4 manières différentes
  - facilement faisable en matériel (sur un FPGA ou dans un ASIC)
  - impossible à faire en logiciel, mais adaptable

# *Full-adder* & langage C

## le masque à zéro

- Le problème du C:
  - impossible de récupérer la retenue (finale) au niveau C (faisable en assembleur via registre d'état)
  - impossible de paramétrer l'ALU
- Ce qui est faisable
  - faire qu'il n'y ait pas de propagation de retenue
    $\Rightarrow$ le bit de poids fort des mots $A$ et $B$ doit etre à zéro
  - utiliser des masques avant chaque calcul.

```
uint32_t m = 0x7f7f7f7f; // masque pour faire 4 calculs sur 7 bits
uint32_t a = 0x12345678; // 12.34.56.78
uint32_t am = a & m;     // 12.34.56.78
uint32_t b = 0x456789a0; // 45.67.89.a0
uint32_t bm = b & m;     // 45.67.09.20
uint32_t c = am + bm;    // 57.9b.5f.98
uint32_T cm = c & m;     // 57.1b.5f.18

uint32_t m = 0x7fff7fff; // masque pour faire 2 calculs sur 15 bits
uint32_t a = 0x12345678; // 1234.5678
uint32_t am = a & m;     // 1234.5678
uint32_t b = 0x456789a0; // 4567.89a0
uint32_t bm = b & m;     // 4567.09a0
uint32_t c = am + bm;    // 579b.6018
uint32_T cm = c & m;     // 579b.6018
```

# SWP & stencil

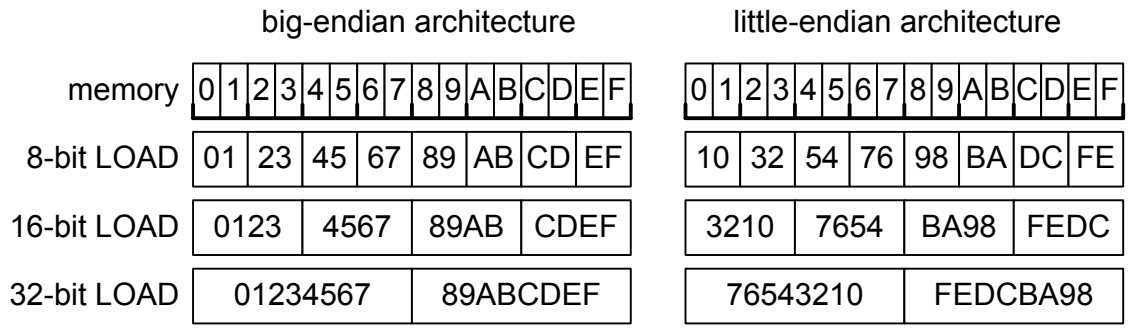
- Création d'une variable dont le contenu provient de 2 variables
  - exemple : `a = 12.34.56.78`, `b = 9a.bc.de.f0` `c = 12.34.56.78`
  - on veut `l == 78.9a.bc.de` et `r == bc.de.f0.12`

| actions | code C | résultat |
|---|---|---|
| shift de 8 vers la droite | `b'= b>>8` | `b'==00.9a.bc.de` |
| shift complémentaire de 32-8 vers la gauche | `a'= a<<(32-8)` | `a'==78.00.00.00` |
| combinaison via un OR | `l = a'\|b'` | `l ==78.9a.bc.de` |
| shift de 8 vers la gauche | `b"= b<<8` | `b"==bc.de.f0.00` |
| shift complémentaire de 32-8 vers la droite | `c"= c>>(32-8)` | `c"==00.00.00.12` |
| combinaison via un OR | `r = b"\|c"` | `r"==bc.de.f0.12` |

- *Design Pattern* left/right
  - version généraliste où n indique le nombre de bit (32 indique la taille des registres)
  - `#define i32left(a,b,n) (a << (32-n) | (b >> n)`
  - `#define i32right(b,c,n) (c >> (32-n) | (b << n)`
  - version spécialisée où n indique le nombre d'éléments (et 8 la taille des éléments)
  - `#define i8left1(a,b) (a << 24 | (b >> 8)`
  - `#define i8right1(b,c) (c >> 24 | (b << 8)`

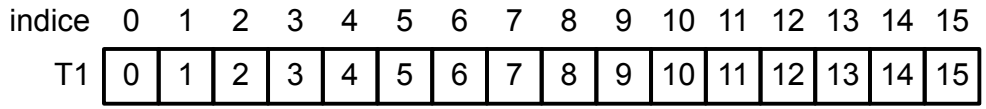
# Mémoire et endianness

- En fonction de l'architecture
  - les données ne seront pas chargées de la même façon
  - et cela pourra aussi dépendre de la taille des loads (en changeant le type du pointeur)

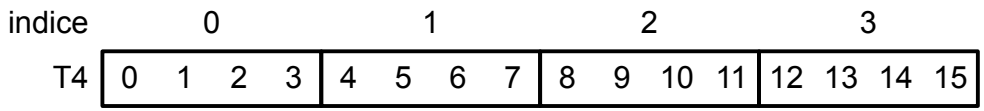



- Architectures big-endian
  - Power, Power PC, ARM (FreeBSD), MIPS, SPARC, PA-RISC, Motorola 68K et 88K
- Architectures little-endian
  - Intel, AMD, ARM (linux), DEC Alpha
- Conclusion
  - concevoir du code portable pour big et little endian
  - les macros précédentes sont pour du big-endian
  - $\Rightarrow$ réfléchir par rapport aux données en mémoire et non en registe

# SWP & stencil




- Soit un tableau `T1` de valeurs 8-bits
  - et une architecture qui est soit SWP,
  - soit généraliste avec additions 7 bits ou émulation d'additions 8 bits
- Stencil d'addition de 3 valeurs (add3)
  - `T1[i-1]+T1[i]+T1[i+1]`: est correct




- Soit un pointeur 32-bits `T4` sur cette même zone mémoire
  - `a=T4[i-1]; b=T4[i]; c=T4[i+1]; y=a+b+c`: est incorrect
  - ce ne sont pas les bonnes additions qui sont réalisées

| | | | | |
|---|---|---|---|---|
| a | 0 | 1 | 2 | 3 |
| b | 4 | 5 | 6 | 7 |
| c | 8 | 9 | 10 | 11 |

# SWP & stencil

*design pattern left/right*

- Avec les instructions de mélange left/right
  - `a=T4[i-1]; b=T4[i]; c=T4[i+1];`
  - `l=i32left1(a,b); r=i32right1(b,c); y=l+b+r`: est correct

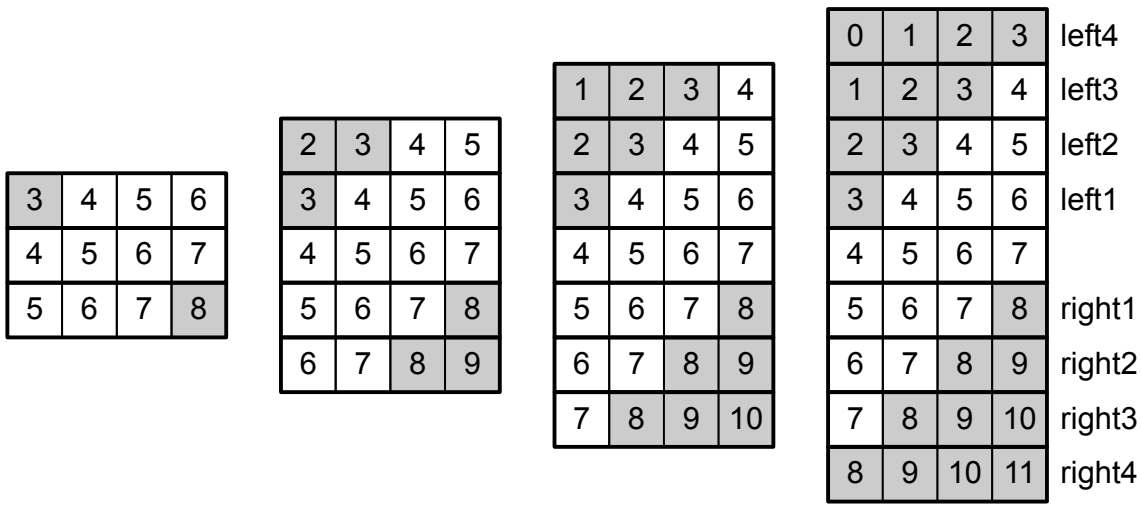
| | | | | |
|---|---|---|---|---|
| l | 3 | 4 | 5 | 6 |
| b | 4 | 5 | 6 | 7 |
| r | 5 | 6 | 7 | 8 |

- vocabulaire
  - les opérations arithmétiques sont des instructions dites verticales
  - les opérations de mélange sont des instructions dites horizontales

# SWP & stencil

## les voisins de mes voisins sont . . . mes voisins

- Pour un registre contenant $c$ valeurs (cardinal)
  - le rayon $r$ du plus grand stencil applicable ne nécessitant que 3 loads est $r = c$




- Amélioration de performance
  - diminution du nombre d'accès mémoire par rapport au nombre d'opérations
  - scalaire: $2r$ ADD pour $2r$ LOAD
  - SWP: $2r$ ADD pour 3 LOAD
  - rappel: objectif des optimisations = diminuer le nombre d'accès mémoire ...

# SWP & émulation

- Soient $a$ et $b$ deux entiers de la même taille que les registres (ici 32 bits)
  - notations $a = a_3 a_2 a_1 a_0$ et $b = b_3 b_2 b_1 b_0$ ou `a=[a3,a2,a1,a0]` et `b=[b3,b2,b1,b0]`
  - l'addition devient:

```
a3=(a>>24)   ; b3=(b>>24)   ; c3=a3+b3; c=   (c3<<24);
a2=(a>>16)&7; b2=(b>>16)&7; c2=a2+b2; c=c|(c2<<16);
a1=(a>> 8)&7; b1=(b>> 8)&7; c1=a1+b1; c=c|(c1<< 8);
a0=(a    )&7; b0=(b    )&7; c0=a0+b0; c=c|(c0    );
```

- Possibilités
  - un masque pour éviter une retenue `ck=(ak+bk) & 0x7F`
  - un masque pour récupérer la retenue `carryk=((ak+bk) & 0x7F) >> 7`

- Implantation sous forme de macro `c=OP(a,b)`
  - remplacer $a_k$ et $b_k$ par leur expression dans $c_k$
  - remplacer $c_k$ par leur expression dans $c$
  - `(((a>>24)+(b>>24))<<24) | ((((a>>16)&7)+((b>>16)&7))<<16) | ...`
  - ne pas oublier de tester et valider la macro avant de l'utiliser ...

# SWP & debug de stencil

## SWP & debug de stencil

- Debug de stencils scalaire à base d'opérateurs logiques
  - relativement facile : remplacer opération logique par opération arithmétique (émulé)
- Debug de stencils binaire et SWP à base d'opérateurs logiques
  - extrêmement compliqué : $n$ valeurs $\{0, 1\}$
  - $\Rightarrow$ remplacer parallélisme de $n$ par parallélisme de $n/8$ ou $n/16$
  - $\Rightarrow$ remplacer l'opérateur logique par opérateur arithmétique (facile si les calculs se font via macro ou fonction)
  - $32 = 4 \times 8$ ou $64 = 4 \times 16$ pour plus de dynamique dans les additions

# SWP de nos jours

## SIMD et bit-slice parallelism

- SWP → SIMD (*Single Instruction Multiple Data*)
  - extension de la largeur des registres à 128, 256 et 512 bits
  - arithmétique et fonctions mathématiques pour flottants 16, 32 et 64 bits
  - arithmétique pour les entiers 8, 16, 32, 64 bits
  - *tous* les processeurs actuels sont SIMD (et même certains micro-contrôleurs)
  - parallélisme très important pour les systèmes embarqués car plus efficace énergétiquement que les processeur multi-coeurs scalaires
- bit-slice parallelism
  - un registre de $n$ bits est interprété comme un registre contenant $n$ valeurs de 1 bit
  - si valeurs sur plus de 1 bit, plusieurs registres pour coder ces valeurs
  - redéfinition d'une arithmétique par bit, (opérateurs logiques intrinsèquement bit-à-bit)
  - très utilisé en cryptographie car permet des accélérations importantes arithmétique modulaire en base 2 (GF2) modulo = XOR
  - tendance actuelle: combiner SIMD et bit-slice parallelism